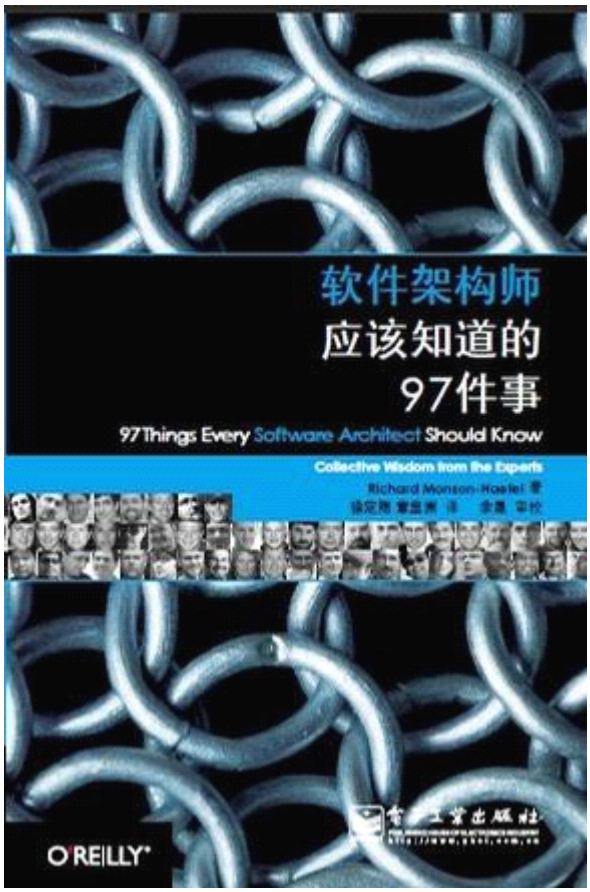

软件架构师应该知道的 97 件事笔记

by: ciahi

http://blog.csdn.net/ciahi

1. 客户需求重于个人简历

不要为了学习新的知识或丰富自己的简历而选择新技术解决问题，要尽量选择切合实际的技术解决客户的难题。
脚踏实地的为客户着想，选择正确的方案可以降低项目的压力，团队工作起来更开心，客户也会更满意，从而你也会有更充裕的时间学习新的知识。

2. 简化根本复杂性，消除偶发复杂性

根本复杂性是问题本身就很复杂，所以它是无法避免的。偶发复杂性是在解决根本复杂性的过程中衍生的，即解决方案本身带来的新问题。
如为了解决某个问题而设计的一个软件框架，设计该框架本身，就是引入的偶发复杂性。所以，如果原本问题比较简单，但设计或引入一个太过灵活的框架，可能得不偿失。(避免过度设计)

3. 关键问题可能不是出在技术上

简单的项目也可能会出现问题，这并不是因为我们选错了某种语言或者系统，根本原因还是“人”
所以，要帮助团队完成项目，多与项目成员沟通，及时改正团队成员不正确的工作方式。
沟通的技巧：
a) 不要把对话看成对抗
改变心态，发现他人的优点，把沟通视为请教，就会有所收获，并且能避免引入对方的戒备之心。
b) 不要带着情绪与人沟通
当你处于愤怒、沮丧、烦恼等情绪中时，对方可能会误认为你的举动不怀好意。
c) 尝试通过沟通设定共同的目标

4. 以沟通为中心，坚持简明清晰的表达方式和开明的领导风格

架构师要避免只发号施令，让开发人员执行的弊病。要让开发人员了解项目的蓝图和决策过程，让大家共同验证你的架构，让开发人员参与你的架构的制订过程，这样开发人员才能更心甘情愿的去执行。
架构师应该提高自己的沟通技巧，帮助大家快速、精准的理解项目的目标。要言简意赅的表达观点，用简单的图表来表达自己的想法。
用相机拍下自己在向团队成员表叙时在白板上写写画画的想法，会后分享给大家，并且自己好好整理一下。

5. 架构决定性能

如果架构有问题，仅仅通过优化局部，不足以获得理想的性能的。找出性能瓶颈，优化瓶颈，如果需要的话，重新设计架构的内在逻辑和部署策略。

6. 分析客户需求背后的意义

分析客户提的需求，为什么提这样的需求。即要知道客户要求的功能和需求的真正意义，这样或许我们可以在达到客户目的的情况下，采取更简单的办法。

7. 起立发言

在多人场合发表意见时，起立发言非常重要，尤其是其他人坐着的时候。站立时，无形中增添了一种权威和自信，容易控制了场面。听众不会轻易打断你的发言。站立时可以更好的利用肢体语言，且在十人以上的场合，起立发言方便与每位听众保持视线接触。眼神交流和肢体语言等表达在沟通中的作用不可小觑。起立发言也更容易控制语气、语调、语速和嗓门，让声音传播的更远。讲到重点时，注意语速的减缓，发声技巧也能改善沟通效果。

8. 故障终究会发生

为了防止 A 程序出错，增加了监控程序 B，但监控程序 B 本身就可能会出错。所以故障及错误是无法避免的，即要做好错误的防范措施。

9. 我们常常忽略了自己在谈判

项目投资人为了削减预算，可能会去掉一些“东西”。当投资人问“我们真的需要这东西吗？”很多工程师把自己摆在了工程师的位置，会回答一些不使用这些东西的替代方案，但往往这些替代方案是极差的。在投资人看来，有其它方案可选，他不关心是什么方案，他往往会让你去掉这些“东西”。
我们这时应该把自己放到谈判者的角度上，说明不使用该方案的坏处，以及使用该方案的好处。让投资人清楚的认识到该方案的重要性。

10. 量化需求

“速度快”、“响应灵敏”都不能算需求，需求一定要可量化。比如响应时间在 500ms 以内。如果没有这些具体数字，也要有一个描述范围，如上限、下限等。

11. 一行代码比 500 行架构说明更有价值

要牢记我们的目标是可工作的代码，设计只是手段。很多架构师往往被抽象的架构所吸引，沉迷于设计过程。没有天生完美的设计，设计都是在实现过程中逐步完善的。如果有机会亲自开发，珍惜这个机会。

12. 不存在放之四海皆准的解决方案

没有通用的解决方案，遇到的问题也是千差万别，架构师需要运用情境意识来解决问题（类似于常识）。情境意识需要培养和训练，架构师要不断的积累案例和经验才能建立足够的情境意识。要解决系统层次的问题，通常需要十年的磨练。

13. 提前关注性能问题

尽早进行性能测试，如果在某个时候性能表现大幅下滑，更容易找到性能下滑是由哪些变化引起的。如果以两周为一个迭代周期，性能测试的开始时间最迟不要晚于第三次迭代。尤其是对性能要求苛刻的系统，验证的早晚直接关系到能否及时交付项目。

14. 架构设计要平衡兼顾多方需求

软件架构不仅包含系统建模、定义接口、划分功能模块、大胜模式、性能优化等，还包括系统的安全性、易用性、产品支持、发布管理、部署方式等问题。
架构师不仅要为用户创造实用的软件，还要平衡兼顾不同部门的目标，如 CEO 要求控制成本，运营部门要求易于管理，二次开发人员要求代码易读易维护等

15. 草率提交任务是不负责任的行为

如果在提交代码前，需要浪费很多时间进行测、验证，则开发人员很可能会为了节省时间草率的提交了任务。一旦出现问题，就会浪费别人很多时间。所以架构师有必要改善这个过程，通过引入自动测试、持续集成等来纠正开发人员的行为。
架构师应沉下心来改善系统的生产效率，缩短流程，缩短开发时间，提升开发体验，减少构建代码时间等，这样也利于杜绝开发员草率提交任务的念头。

16. 不要在一棵树上吊死

没有什么架构、策略、观点能解决所有的业务问题，我们要承认世界是混乱的，解决方案也是多样的、不一致的等。

17. 业务目标至上

架构师必须成为业务部门和技术部门之间沟通的桥梁，兼顾双方的利益，用业务目标来驱动项目开发。
架构师要评估项目商业价值，以高的投资回报率作为目录，避免作出错误的技术决策。要谨慎的站在业务团队一边，用业务目标驱动项目开发，才能保证软件开发团队的长远利益。
在软件形成产品过程中，需要持续的根据反馈制定决策，软件开发人员可以制定微观技术决策，业务决策由业务部门来制定。

18. 先确保解决方案简单可用，再考虑通用性和复用性

一味强调通用性而不考虑具体应用，会导致许多令人困惑的可选项和不确定因素。多余的功能往往被闲置或被误用。如果有多个可选解决方案时，“先简单后通用”

开始就追求理论上的通用性，往往会导致解决方案脱离实际的开发目标。而且这种理论上的通用性往往是基于错误的假设之上的，所以无法提供有价值的方案，只会带来问题。通用性而来于对需求的理解，理解之后才能简化。

19. 架构师应该亲力亲为

称职的架构师即要懂技术，才能代表技术团队发言，又要懂业务知识，才能督促技术团队满足业务需求。而且应该能通过示范领导团队，能够胜任团队的所有工作，从网络布线，到配置，从单元测试代码编写，到进行测试工作等。还要有能力发现问题所在，向大家解释问题产生的原因，或者给出解决方案。
架构师应该尽早的参与项目，与团队成员并肩作战，对细节的关注程度往往决定了项目的质量。遇到难题不要仅仅扔给别人，可以亲自动手研究或者咨询其他优秀架构师的意见。

20. 持续集成

源码的提交或变更触发工具对项目进行自动构建、自动测试，反馈结果。

21. 避免进度调整失误

欲速则不达,赶进度会引发很多问题，如拙劣的设计、蹩脚的文档，可能引发的质量问题，仓促完成的代码 Bug 数量也会显著的增加。
作为架构师应该避免赶进度的情况，如果一定要提前发布，则尝试去掉一些不重要的功能，待后续版本再加入。

22. 取舍的艺术

鱼和熊掌不可兼得，没有十全十美的设计：高性能、高可用性、高安全性而且高度抽象。也不要妄想将所有的需求都实现。
架构权衡分析方法(Architecture Tradeoff Analysis Method, ATAM)、成本收益分析方法(Cost Benefit Analysis Method, CBAM)都是帮助架构师做出取舍的工具。

23. 打造数据库堡垒

不管业务如何发展、人员如何变动，数据总是不变的，且会永远保存下来，所以创建牢固的数据模型要从第一天开始。数据库的出错是灾难性的，一旦数据被破坏，即使后面修复了数据层的设计问题，丢失的数据也无法恢复了。
要充分发挥关系型数据库的作用，让它成为应用的一部分，从开始构建数据库时，就要深刻地理解业务需求。虽然普通开发方法中敏捷被证明是非常好的方法，但对数据库的设计还是要保守，开始之前要认真设计。

24. 重视不确定性

在设计过程中，面对多种可能方案举棋不定时，要仔细考虑采用哪一种方案，或

者推迟决定，当收集到更多的信息时，再做出后续的更好的选择。但也不能太迟，要赶在这些信息失效前利用它们。
如果对着代码反复琢磨但仍无法决定采用哪种实现方式时，实现时设法利用分离或封装将决策和最终依赖于决策的代码隔离开，这样在决策变更时可以尽量降低成本。优秀的架构会把这个变更成本降的尽量低。

25. 不要轻易放过不起眼的问题

莫非法则：凡事只要有可能出错，那就一定会出错。
不要忽视小的问题，有可能到后面会演变为重大的灾难。团队的每个人通常关注点不同，当别人提出问题时，要尽量重视。
每个人身上都存在自己难以识别和接受的盲点和不足，所以有"不妥"的感觉时，放大这种感觉，重视起来。

26. 让大家学会复用

让大家知道你的框架、库或设计，然后让大家知道它如何使用。
一般有经验的人都喜欢寻找已有的解决方案。

27. 架构里没有大写的"I"

架构师不能自负，别人批评我们的设计时，要学会接受与学习。
要避免认为自己比客户更懂需求或把开发人员当作资源，要与客户密切合作，驱动架构的是需求，要重视团队合作，架构是属于团队的，经常检查自己的工作以及反省自己，要杜绝"自我意识"引发的一些常见问题。

28. 使用"一千英尺高"的视图

要判断软件的质量，从架构图上看不清楚，太过抽象，从源码上看又容易迷失自我，细节太多。选取一个介于两者之间的视图，即能包含足够的信息，又不至于陷于细节的泥沼之中。

29. 先尝试后决策

越晚决策，可利用的信息就越多，但多并不意味着足够。完美的决策只可能出现在事后。
在制定决策之前，可以和开发人员商讨解决方案，然后尝试一段时间，在决策点临近时，再根据尝试的结果作出较优的决策。
尝试多种方案看似浪费时间，但实际上可能是代价最低的选择。

30. 掌握业务领域知识

掌握业务领域知识，有助于选择合适的架构模式，而且可以更好的制定未来的扩展适应不断变化的产业趋势。还能自如地运用企业高管和用户熟悉的行业术语与

他们沟通。

31. 程序设计是一种设计

代码即文档，写代码即是设计行为，而非生产行为。

32. 让开发人员自己作主

应该给自己的团队足够的自主权，让他们发挥自己的创意和能力。不要过于拘泥于细节，要为开发人员创造一个良好的开发体验，如自己设计的 API 是否易于理解及使用，如果经常被误用，应该怎么修改。而且要创造一个良好的氛围，让大家主动起来，如果遇到什么问题，及时的向你征求意见。

33. 时间改变一切

让结果说话，认真执行 KISS 原则。现在的设计，可能会被两三年之后的自己所否定，同样，以前的设计也容易被自己否定，所以不要跟以前的工作过不去。

34. 设计软件架构专业为时尚早

软件开发仍处于相对初级的尝试阶段

35. 控制项目规模

尽量控制和缩小项目规模，提高项目成功机会。
a) 抓住真正需求
b) 分而治之
c) 设置优先级
d) 尽快交付

36. 架构师不是演员是管家

炫耀和作秀放到市场营销上去，而不是设计中，架构师应该把自己看成管家，承担着管理他人资产的责任，为客户利益着想。

37. 软件架构的道德责任

任何浪费用户时间的行为，就是不道德的行为。即使只浪费用户一点时间，但影响到几百万用户时，累加起来就是一个非常长的时间。应该浪费自己的时间，节省用户的时间。

38. 摩天大厦不可申缩

软件相对建筑，扩展新功能要简单的多，而且产品发布越早，公司的净收益就会

越高，所以，应用软件只要具备了用户要求的关键功能就可以发布了。提早发布，还能持续改善软件架构的品质。

39. 混合开发的时代已经来临

混合编程是指在同一套软件系统中同时采用多种核心编程语言。
把若干强大的工具组合起来，形成更巧妙的解决方案。

40. 性能至上

性能和其它指标一样重要，减少软件响应时间，提高人机交互效率。
不要拿摩尔定律来安慰自己，认为硬件及系统的速度足够快并且以后会更快，而忽略了软件性能。

41. 留意架构图里的空白区域

架构图中的两个模块之间，仍有很多细节需要考虑，比如 A 和 B 的通信，在架构图上可能只是简单的一个箭头，但实际上要考虑两个程序之间的响应时间，如果超时如何处理，如果中间电缆出问题怎么办等。

42. 学习软件专业的行话

使用行话可以提高交流质量及效率。如企业架构模式、应用架构模式、集成模式、设计模式、反模式等。

43. 具体情境决定一切

设计架构的过程就是做出明智决策的过程。脱离了具体的情景比较技术的优劣是无意义的。

44. 侏儒、精灵、巫师和国王

一个团队中要有各种性格和各种特长的人，招聘时尽量招聘不同性格的人。
相同的人在一起，视野往往不够开阔。安排任务时也尽量根据团队成员的特点来安排，让大家相互学习。

45. 向建筑师学习

要想成为伟大的建筑师，优雅丰富的心灵远比聪明才智重要。 --- 弗兰克·劳埃特·赖特
建筑师应该是伟大的雕塑家，或者伟大的画家，否则他不过是个建筑工人。--- 约翰·拉斯金
架构应该蕴涵适当的艺术成份。

46. 避免重复

如果开发人员复制救命代码中的内容，说明这部分还可以简化，甚至全部提取出来。
消灭复制是架构师的责任，如果有重复，则应该重新研究框架，创造更完善的抽象机制。

47. 欢迎来到现实世界

现实世界是不可预知的，随时都可能发生一些让人预料不到的事情，如客户撤消订单，付款时间延误等。
如果现实世界带来了麻烦，不要怕，不要报怨，寻找解决办法应对即可。

48. 仔细观察，别试图控制一切

我们已经进入了分布式、松耦合的时代，不要妄想掌控一切，这样只会让你设计出紧耦合、脆弱的方案。但是撒手不管也是危险的状态。正确的做法是：仔细观察，提取模型，然后检查验证。

49. 架构师好比两面神

要有兼顾前后、过去与未来的能力。善于倾听和观察，思想开放，即要满足当前需求，又要兼顾未来的发展规划。即要让系统易于访问，又要保证系统安全；即要让设计符合当前的业务流程，又要体现管理层对未来发展规划的考虑。融合不同的思想和观念，兼顾不同的设想和目标，才能开发出皆大欢喜的产品。

50. 架构师当聚焦于边界和接口

分而治之，分离关注点。对架构师来说，困难在于找到设置边界的自然之处所需要的接口。情景地图为架构师提供了一个强大的工具，使得他们可以聚焦于“哪些应该归在一起，而哪些应该分开”，从而使他们能够以一种可顺畅沟通的方式，实施明智的分而治之。

51. 助力开发团队

要在职责范围内尽量去助力开发团队，不能仅仅是只说不做。要确保开发人员拥有所需的技能，定期进行培训、讨论、实践等。在选拔开发人员过程中，也尽量选择那些热衷于学习，有亮点的。当不违背软件设计的总目标时，就让开发人员自己做出决策。要保护好开发人员，避免让他们卷入太多不重要的工作之中。

52. 记录决策理由

记录软件架构决策理由的文件，长期来看非常有用，因为架构不会经常变，所以也不用付出过多维护精力。

它一般会记录如下基本问题：1. 我们做了什么决策？ 2. 为什么这样决策？ 3. 还有哪些解决方案没有采用？ 等等
用处：1.沟通工具 2.移交项目给别人 3.写文件也会迫使自己明确这样决策的理由，有助于确保基础是扎实稳固的。 4.当相关条件变化时，需要重新决策时，这份文档是一个不错的起点。

53. 挑战假设，尤其是你自己的

“臆断是事情搞砸的根源” --- 韦森“延期判决”法则
要对一些假设清楚明确，拿出相关的经验数据验证假设，最后再做出决策。事实和假设是构建软件的两大支柱。务必确保软件的基石坚实可靠。

54. 分享知识和经验

软件行业还非常年轻，想要持续发展，则传播经验和知识是非常重要的。
在个人层面，也利于成长，能更好的理解和修正已知的知识和经验。

55. 模式病

避免过度使用模式，适合的才是好的

56. 不要滥用架构隐喻

隐喻对那些通常比较抽象、复杂和变化移动的目标，提供了很好的具体媒介。无论是与其他队员沟通，还是与最终用户讨论架构全局，找到有形实物作为正要构建的东西的隐喻都是十分诱人的。这在开始的时候很有效，都使用一种语言，能让大家感觉到正确的方向，不断演化前进。但隐喻也容易被滥用。
滥用隐喻可能让其他团队成员沉溺于隐喻，且隐喻不能完全展现问题。如业务系统还在构想之中时，和方共享的是最乐观的可能解读，并没有包括任何必要的约束等。

57. 关注应用程序的支持和维护

架构师大多出身于开发人员，而非系统管理员，所以很容易站在开发者的角度上来思考。所以设计出来的系统，会让系统管理员遇到很多问题，导致很多新的问题。
要学会多角度考虑，尽早引入支持负责人，让其参与规划应用程序的支持。

58. 有舍才有得

CAP 定理：在分布式系统中通常期望的 3 个特性，即一致性、可用性和分区容错性是无法同时获得的。
永远不要放弃质疑，因为架构设计的教条往往从根基上削弱了交付能力。权衡是不可避免的，并且接受一些权衡，往往能产生富有创造性和创新性的结果。

59. 先考虑原则、公理和类比，再考虑个人意见和口味

用原则、公理和类比来指导创建过程，有很多优点：
a)架构文档化更容易 b)更容易交流与传奇架构师的个人意见与经验 c)清晰的架构能把架构师解放出来 等
如果单凭个人经验、意见和口味来盲目地创建架构，是无法获得这些好处的。原则和公理还能确保架构上的一致性。

60. 从“可行走骨架”开始开发应用

“可行走骨架”是对系统的最简单实现，它贯串头尾，将所有的主要的构架组件连接起来。然后小步的、增量式的，能不断得到反馈向正确方向前进。

61. 数据是核心

从概念上看，数据要比代码更加精练，也更好理解。即使对地最复杂的系统，从面向数据的视角，即通过底层信息的结构整体看待系统，也可将系统缩减为细节的有形集合。数据在大多数问题中牌核心地位，业务领域问题经由数据蔓延到代码中。只有数据真正构成了每个系统的核心。

62. 确保简单问题有简单的解

简单问题使用简单解，听起来容易做起来难。架构师经常出于炫技心理，容易过度设计。架构师会从主观的判断或潜在不确定需求出发，产生调整解决方案的强烈冲动。不要试图猜测未来的需求，错的概率是 50%，错的离谱的概率是 49%。不要从主观猜测未来需求，而是从反馈中不断生成真实的需求。

63. 架构师首先是开发人员

身居要位仍然要继续跟进各自领域的发展。获得开发人员的尊重和信任，让开发人员自愿接下任务。
时不时的去处理一些比较复杂的任务，目的：a)让自己做到宝刀不老 b)有助于向开发人员证明自己不是只会吹牛皮的
主要目标是创建可行、可维护的解决方案。且自己设计的系统自己应该能编程实现。

64. 根据投资回报率(ROI)进行决策

我们对项目所做的每一个决策--无论是技术、过程、还是与人相关都可以看作是一种投资形式。假设产品上市时间对投资方是至关重要的，那么快速发布就能带来更高的投资回报率，这个时候“完美的架构”就不如架构稍差，但能快速完成的版本。
将架构决策视为投资，并将相关的回报率也一并考虑在内。

65. 一切软件系统都是遗留系统

系统再先进，对接手它的人而言，都是遗留系统，所以不应该排斥这个标签。

66. 起码要有两个可选的解决方案

对某个问题，如果只考虑了一个解决方案，就有麻烦了。如果没有对比其它方案之前就想当然地给出了解决方案，那就要三思了。

67. 理解变化的影响

架构师在解决方案中给出的抽象，应该能为更高的层次提供坚实基础，应该能足够务实地应付未来的变化。了解变化，包括人与人，系统与系统的。要清楚认识解决方案中变化的类型和将带来的影响。

68. 你不能不了解硬件

架构师既要负责连接业务需求和软件解决方案，也要负责连接硬件和软件。硬件方面的一些知识同软件架构一样重要，比如说硬件的容灾能力、容量规划等。如果缺乏硬件规划能力，最好和基础设施架构师紧密合作。

69. 现在走捷径，将来付利息

碰到架构问题或设计缺陷，作为架构师，一定要坚持成本还很低廉时就动手。搁置越久，为之付出的利息也就越高。

70. 不要追求“完美”，“足够好”就行

“足够好”指的是，剩余的不完美之处，对系统的功能、可维护性或性能不会产生任何有深远意义的影响。

71. 小心“好主意”

“好”主意的邪恶之处：它们是好的，不容易被发现它们的问题，如：某框架有升级版本，我们也应该使用新版本。某技术很强大，我们可以把它用于……

72. 内容为王

很多系统无止境地强调需求、设计、开发、安全等，从未关注系统真正的要点---数据。对基于内容的系统，数据尤其重要。在新系统的设计过程中，必须留出一部分精力专心考评内容库。比如系统重点关注哪些内容，能否及时更新，内容主要来源等等。
系统的成功取决于其内容。

73. 对商业方，架构师要避免愤世嫉俗

自我感觉过于良好，往往会忘记倾听，并且会拒绝听取、分析别人的建议。过度自信，会让你在业务领域头破血流。业务是架构师职业存在的原因，为业务服务是我们的生存之本，倾听和了解雇主的业务，是我们必须掌握的最为关键的技能。不能只提批评意见，还需要针对不足提出建设性的意见。

74. 拉伸关键维度，发现设计中的不足

通过某些关键维度被拉伸，可以以此来发现系统设计的限制。(即提前想象某些维度被扩大、拉伸)
如：了解基础设计的规划是否能够应付增长的需求，圈出限制范围。对吞吐量的峰值能否正常处理等等。

75. 架构师要以自己的编程能力为依托

为项目设计架构时，对实现每个设计元素所需的工作量，要做到心中有数：如果以前曾开发过其中某中设计元素，那么估算所需工作量就会容易得多。
不要在设计里使用自己没有亲自实现过的模式，不要使用自己没有用之写过代码的框架等等，如果架构依赖于各种你未曾亲身使用过的设计元素，那其中就存有许多负面影响，如无法估计实现设计所需的时间，无法容易的避免那些设计元素里面的陷阱，开发人员遇到问题时无法向你请教等等。
(架构师平时紧跟职业步伐，平常多关注，多自己使用一些框架、模式等，在需要使用这些东西的时候，肯定是有经验的了)

76. 命名要恰如其分

如果不知道一个东西应该叫什么，那你肯定不知道它究竟是什么。
如果无法给出个合适的命名，那也就无法继续编程。如果发现自己需要多次理性命名，那么最好停下来，直到弄清楚要做的究竟是什么。
一定要起个好名字!!!!!!!!

77. 稳定的问题才能产生高质量的解决方案

最好的架构师不是去解决难题，而是围绕难题开展工作。架构师要能够将四处弥漫的软件问题圈起来，并画出其中的各边界，确保对问题有稳定、完整的认识。如果问题是稳定的，那么问题解决之后，就永远不会再来烦你。

78. 天道酬勤

具有创造力虽然是成功架构师的一项重要特质，不过成功架构师同样还需要具有勤奋的特质。很多时候不是能力不足导致项目失败，而是由于勤奋不够，紧迫感不强导致的。

79. 对决策负责

很多失败的项目，究其根本原因，是因为做出了不当的决策，或无法执行正确的架构决策。
做出有效决策的方法：
a)对决策过程有充分的认识（决策以书面形式记录下来了，与决策执行人沟通过）
b)定期对架构决策进行复审，检查决策的实际效果和预期结果。
c)要贯彻架构决策，架构师不仅要参与设计阶段，后续仍然要持续跟进。
d)可以将一些决策交给问题域专家。

80. 弃聪明，求质朴

小聪明会诱导我们在软件开发中使用奇技淫巧。聪明的设计僵硬难改，细节会对全局产生太多的牵扯。质朴的方案中，每个组件只做一件事，组件耗时少，易于创建，以后维护也更容易。

81. 精心选择有效技术，绝不轻易抛弃

软件架构师工作很大的一部分，是要选择用以攻克难题的合适技术。精心选择熟悉的武器，不到万不得已，不要轻易排序它们。
选择新技术虽然有风险，但其价值在于往往能为你带来质的飞跃。不过仍然要谨慎选择。

82. 客户的客户才是你的客户！

想象你的客户并不是你的客户，而你客户的客户才是你的客户，这样你的客户的客户赢了，你的客户也就赢了。
例如，你为某个机械开发一个网站，你要想到使用这个网站的最终用户！
如果你的客户有意无意的忽视他们的客户所看重的重要事项，这个时候你就需要考虑与你的客户沟通、甚至放弃这个项目。

83. 事物发展总会出人意料

设计是一个不断发现的过程，不要开始试图把设计做的很完美。在开发过程中，可能不断有一些微小的变化，这些变化积累起来就需要对设计进行一次大的变更，如果还是试图维持住已经走样的设计，就会越破坏原设计。

84. 选择彼此间可协调工作的框架

软件框架是软件系统的基础，选择框架时，不仅要考虑每个框架自身的质量和特性，还要关注共同构成系统的各个框架之间是否能和谐共处。而且后面是否方便地向其中加入新的框架。即选择彼此之间没有重叠，而且开放、简洁、精专的框架。

比较好的框架专注于解决某个独立的逻辑领域，且不会侵入其他必须框架的领域关注面。精简、包容、灵活。

85. 着重强调项目的商业价值

利益相关人士往往缺乏软件工程方面的知识，给他们讲软件架构，经常是对牛弹琴，他们认为架构是虚无缥缈的，所以，在为项目争取资金时，要着力强调项目的商业价值，利益相关者一般对这个比较感兴趣，而且更容易达成一致。
将架构提案打造为典型的商业项目步骤：
a)形成价值陈述。即你的决策摘要，用以说明组织的业务为何要采用特定的软件架构。重点要放在该架构如何提高生产力、改进业务效率等。不要强项这种技术如何高明。
b)建立量化的度量标准。量化越具体，项目也将越具有说服力，越能让人相信好的架构可以带来丰厚回报。
c)回过头来关联传统商业的衡量方式。如果能将技术分析转化为财务数据，则更有说服力。
d)知道该在哪里停止。准备好一张路线图，用以捕获远景目标，清楚地知道每一个里程碑将带来的价值。让利益相关者自己决定在哪里停止。如果每处的商业价值都十分显著，就可能获得持续不断的资金支持。
e)寻找恰当的时机。

86. 不仅仅只控制代码，也要控制数据

在架构规划过程中，数据迁移部分经常被架构师忽略。最后数据迁移往往是作为一项事后补救描述，而且整个过程由手工完成，相当脆弱。对数据方案和数据内容的管理，应当尽早无缝集成到自动化的构建和测试过程中，还必须提供回退功能.

87. 偿还技术债务

当已投入实用的项目出现了问题，往往会出现两个选择：
a)花合适的时间进行“一次做对”，可能包含一些重构之类的工作
b)走“捷径”，完全为了满足当前的 bug 而填的一些代码，可以很快的推出修改的产品。
应该尽量选择第一个方案，第二个方案会不断的积累技术债务，越往后就越难以改变。
不过如果时间很紧迫，可以采用第二个方案，但改完之后，不能就此止步，后续仍然要考虑这个技术债务，在适当的时候清理了它。

88. 不要急于求解

首先看看是否可以改变问题。

89. 打造上手的系统

我们是工具制造者，我们制造的系统一定要帮助人们做事，否则就推动存在的意义。
“上手”即容易使用的工具。

90. 找到并留住富有激情的问题解决者

优秀的团队是项目成功非常重要的原因！也要保持团队的稳定！
打造健康的工作环境。好的开发人员，常常能从认可中获得强烈的激励。
提防批评过度，批评过度可能会扼杀开发人员的创造力，降低其生产力。可以提出建设性的批评建议，但不要强求每个解决方案看起来好像都出自你手。
以正确的方式经营开发团队。如同团队成员并肩作战，对他们一视同仁，培养团队精神等。

91. 软件并非真实存在

软件是我们创造的虚拟物，相对于物理世界中的对应物，更易于改变。所以产品的需求可能会不断发生变化，计划不断调整。

92. 学习新语言

语言包括各方面的，如与业务人员交流的语言，与程序员交流的语言，以及扩大自己的知识面需要了解的语言等。

93. 没有永不过时的解决方案

今天的解决方案一定会成为明天的问题，没有“永不过时”的解决方案。

94. 用户接受度问题

去了解与衡量接受度问题带来的威胁，并朝能减轻这些威胁的方向开展工作。比如找代表用户利益的项目拥戴者，与用户进行直接的沟通并影响用户的接受度。（接受度：如用户不想接受一个新的系统，人们不愿意实施新的系统等）

95. 清汤的重要启示

清汤是不断精练与浓缩才成的，软件架构也应该学习清汤的制作方法。

96. 对最终用户而言，界面就是系统

要让界面易用，好的界面能帮助用户提高生产力，用户会因此更加喜欢我们的产品。
用户交互实际和健壮性、性能等一样重要的。

97. 优秀软件不是构建出来的，而是培育起来的

从小的可工作系统开始，逐渐把它推向成功的目标。