两个维度：一个维度是编程技能层次（技能），另一个维度是领域知识层次（应用、业务）。

**编程技能层次：指的程序员设计和编写程序的能力。这是程序员的根本。**

**1段—基础程序员：**

      学习过一段时间编程后，接到任务，可以编写程序完成任务。

**2段—数据结构：**

       经过一段时间的编程实践后，程序员会认识到“数据结构+算法=程序”这一古训的含义。他们会使用算法来解决问题。进而，他们会认识到，算法本质上是依附于数据结构的，好的数据结构一旦设计出来，那么好的算法也会应运而生。设计错误的数据结构，不可能生长出好的算法。记得某一位外国先贤曾经说过：“**给我看你的数据结构！**”

**3段—面向对象：**

       再之后，程序员就会领略面向对象程序设计的强大威力。大多数现代编程语言都是支持面向对象的。但并不是说，你使用面向对象编程语言编程，你用上了类，甚至继承了类，你就是在写面向对象的代码了。只有你掌握了接口，掌握了多态，掌握了类和类，对象和对象之间的关系，你才真正掌握了面向对象编程技术。就算你用的是传统的不支持面向对象的编程语言，只要你心中有“对象”，你依然可以开发出面向对象的程序。

      掌握面向对象思想的途径应该有很多种，我是从关系数据库中获得了灵感来理解和掌握面向对象设计思想的。在我看来，关系数据库的表，其实就是一个类，每一行记录就是一个类的实例，也就是对象。表之间的关系，就是类之间的关系。O-Rmapping技术（如Hibernate），用于从面向对象代码到数据库表之间的映射，这也说明了**类和表确实是逻辑上等价的。**

编程语言发展到今天，就是需要模块化，就是需要把“数据结构”和“算法”结合起来。

**4段—设计模式：**

    设计模式是编程的客观规律，是针对特定场景特定问题的有效解决方案。

    不用设计模式，你也可以写出满足需求的程序来。但是，一旦后续需求变化，那么你的程序没有足够的柔韧性，将难以为继。而真实的程序，交付客户后，一定会有进一步的需求反馈。而后续版本的开发，也一定会增加需求。这是程序员无法回避的现实。

设计模式，最重要的思想之一就是解耦，通过接口来解耦。

**5段—架构设计**

还需要掌握架构设计的能力，才能设计出优秀的软件。架构设计有一些技巧：

1、分层：可以解耦各个模块，支持并行开发，易于修改，易于提升性能。一个软件通常分为：

* 表现层--UI部分
* 接口层--后台服务的通讯接口部分
* 服务层--实际服务部分
* 存储层—持久化存储部分，存储到文件或者数据库。

2、SOA

     模块之间通过网络通讯互相连接，松耦合。每一个模块可以独立部署，可以增加部署实例从而提高性能。每一个模块可以使用不同的语言和平台开发，可以重用之前开发的服务。SOA，常用协议有WebService, REST, JSON-RPC等。

3、性能瓶颈

1）化同步为异步，多线程并发

2）用集群来处理计算

3）用cache

### 程序员编程语录

   1. 一个好的程序员是那种过单行线马路都要往两边看的人。

   2. 程序有问题时不要担心。如果所有东西都没问题，你就失业了。

   3. 程序员的麻烦在于，你无法弄清他在捣腾什么，当你最终弄明白时，也许已经晚了。

   4. 我想大部分人都知道通常一个程序员会具有的美德。当然了，有三种：懒惰，暴躁，傲慢。

   5. 一个人写的烂软件将会给另一个人带来一份全职工作。

   6. 工作进度上越早落后，你就会有越充足的时间赶上。

   7. 为什么我们没有时间把事情做对，却总有时间把事情做过头？

   8. 傻瓜都能写出计算机能理解的程序，优秀的程序员写出的是人类能读懂的代码。

### 编程/软件开发语录

   1. 按代码行数来评估软件开发的进度，就如同按重量来评估飞机建造的进度。(比尔-盖茨)

   2. 在水上行走和按需求文档开发软件都很容易——前提是它们都是冻结状态。(Edward V Berard)

   3. 注释代码很像清洁你的厕所——你不想干，但如果你做了，这绝对会给你和你的客人带来更愉悦的体验。

   4. 软件设计最困难的部分是：阻挡新功能的引入。

   5. 测评不会撒谎，但测评的人会。

   6. 培养员工，即使他们有跳槽的风险。什么都不做而留他们在公司，这样风险更大。

   7.  程序必须是为了给人看而写，给机器去执行只是附带任务。

### 软件纠错语录

   1. 删除的代码是没有bug的代码。

   2. 如果纠错是消除软件bug的过程，那编程一定是把它们放进去的过程。

   3. 代码纠错要比新编写代码困难一倍。

   4. 想在自己的代码里找出一个错误是十分困难的。而当你认为你的代码没有错误时，那就更难了。

### 软件产品/成品语录

   1. 软件能够复用前，它必须要可用。

   2. 软件通常在beta测试完成不久后发布。Beta在拉丁语中是“还不能用”的意思。

   3. 最好的性能改进是将软件从不能用的状态变成可用。

   4. 最廉价、最快速、最可信赖的组件是那些还未出现的组件。

   5. 软件和教堂非常相似——建成之后我们就在祈祷。

   6. 除非最后一个用户死掉，软件是不会有完工的时候的。

编程，乐趣何在？

# 1.   什么是软件开发？

软件最基本的目标是让计算机硬件（运算/存储/输入输出）按照人们预想的规则来工作。我们又管软件叫程序，软件工程师定制编写一个“顺序、序列”，机器就按照这个序列来执行。软件开发，就是这个定制编写序列的过程。

# 2. 原本的乐趣：挑战和控制欲

     解数学题,是很多理科学生都很喜欢的一项活动。特别是在高中时期，证明出一道立体几何或者在模拟考试中第一个交卷儿都是非常令人羡慕的，虚荣心和满足感也会随之飘飘然。同时，多数中学的老师和一些大学老师，喜欢把软件开发归于数学的范畴。按这个推理，喜欢数学的都应该喜欢编程。但事实并非如此。

    无论男性还是女性，我们都有**控制的欲望**。在控制不了“人”这种活物的情况下，能够控制一台机器让他按照我们的意愿来运行，会带来极大的快感。我30多岁了还喜欢玩遥控汽车，但一直羞于去玩具店购买，直到我儿子2岁以后。玩跑车，用手臂和脚尖控制一台400马力的怒吼的发动机当然更加过瘾，但显然太昂贵了。编程则可能是达到这一目标最廉价又最冠冕堂皇的一种方式。而且**编程这种活动似乎在发挥创造力和满足自我陶醉心理上有更大的空间**。同样，现在的情况也非如此，越来越多的程序员开始不喜欢他的职业了。

# 3.   为什么软件开发越来越无趣？

## 3.1.首先，软件开发并不是数学

  我们在学校的时候，那些老师们把软件开发归于数学的范畴，这没错，但过于狭隘。在30年前数学或许是软件的80%，但今天我们更倾向于把软件开发称为“工程”。工程与数学是不同的范畴，尽管在工程中我们会用到数学，但并不是全部，而且在软件工业的逐步发展的过程中，由于行业的分工进一步细化，数学的应用在软件工程中的比例越来越小。

       算法是数学在编程中最基基本的一种表现方式，但在80%的软件开发项目数百万行代码中，能真正让你去思考“算法”的部分寥寥无几。来自于“解题”的快感，自然无从寻觅。没有挑战，哪来成就感？

## 3.2.第二，软件工程技术的发展，限制了施展的空间

       计算机给程序员提供了一个广阔的思维空间，但计算机工业则根据自己的发展需求将这个空间切分成非常细小的片断。位于产业链上游和技术前沿的厂商、团体和个人通过中间件产品（如数据库、应用服务器）、开发工具、设计理念、框架、宣传等方式则各自独占了软件技术链上最“有趣”的一部分。多数现代程序员，则随流进入其它一个一个狭小的片断中。那些所谓技术含量较低的管理软件（广义的业务软件）领域，更是集中了大多数的从业者。

       平台、数据库、应用服务器、开发工具、现代软件设计理念、软件框架、技术宣传，这些产品和理念在推动软件工业成熟和发展的同时，一方面在宏观上提高了整个行业的生产率，降低了技术门槛，吸引了更多的从业者；另一方面则在微观上剥夺了多数程序员享受编程乐趣的环境。你应用EJB或者SSH(struts/spring/hibernate)开发项目的过程中，由衷地体会到编程的乐趣了吗？我反正没有。

## 3.3.第三，PHP和Java

       C语言是有趣的，因为它是“计算机科学”发展的产物。

       Python和Ruby是有趣的，因为它是“天才”的产物。

       Delphi是有趣的，因为他是史上“最优美的结构化编程教学语言Pascal”的延伸。 与之不同的是多数的资深程序员认为PHP和Java是无趣的， PHP是快速WEB生产需求催化的产物。Java则是软件工程发展的产物。

       当“编程”遇上“快速生产”和“工程”的时候，乐趣就开始退化了。然而与乐趣无关的是，他们三个却成为了现代软件工业中最成功的三把斧头。一把能快速的砍出一个WEB论坛；一把能跨速的砍出Client界面；一把则通过理念、框架、规范、中间件等等等等，使得软件开发更加模式化和规范化，令软件行业向大规模工业化生产方式向前迈进了一大步。

      PHP和Java本身都是成功的，而陷于两者的程序员则多半难以成功。我们现在常常赞赏地说：XX技术XX框架让程序员更关注于业务逻辑。我们在享受他们所带来的便捷的同时，也正在慢慢丧失程序员的天性——创造力。

# 4.   寻找新乐趣之旅

我们不能选择放弃，那么就让我们开始去寻找新的乐趣吧！

## 4.1.创新：用户UI体验的乐趣

与20年前不同，当年的软件更接近“底层”，而今天我们所开发的软件则更多地接近用户的感官和操作。把成就感从底层的挖掘移向UI层的体验，显得顺理成章。

同时，当今的UI技术和硬件渲染能力非20年前可比。以我们目前接触最多的WEB应用为例，最为普通的HTML/CSS/Ajax/JS/Flex等技术为我们提供了全所未有界面表现能力。我一直坚信优秀的用户体验是成功的一半。最近几年的Web创新很多都集中在表现方式上，如Ajax和Flex。

一些小型的用户体验提升方式已经普及到了“标配”的程度。比如，5年前如果你在一个Web表单中输入了错误的数据，必须在提交后的下一个页面中被提示出错；而今天不能在Input框的右边提供实时交验信息的界面则是令人恼火的经历。

在UI上的创新远不止这些。在Ajax和Flash令界面表现的丰富程度达到VB/Delphi望尘莫及的今天，我们追捧着gmail，研究着google map，效仿着flickr，甚至崇拜着fins的GT Grid。一旦有人能够向UI体验发出挑战性的创新，就会给开发者赢来众多赞赏的目光和追随者的效仿，伴随而来的是开发人员极大的快乐。

## 4.2.探险：扒开“框架”的乐趣

使用Hibernete谈不上乐趣，至少是乐趣有限。但如果你扒开Hibernate的代码，跟着作者的思路在数十万行代码迷宫中探险的时候，当你拨开一层层迷雾，为一段思路一行程序一种理念一个技巧而拍案叫绝的时候，你可能会得到前所未有的乐趣：**这种乐趣可能，**

**来自于“发现”的惊喜，**

**来自于“理解”的激动，**

**来自于“学习”的充实，**

**来自于“顿悟”的爽快！**

**来自于“英雄所见略同”的自豪感！**

**在咱们软件圈儿，大师用书说话，大侠则用代码说话**。“书上得来终觉浅，绝知此事要躬行”。转进大侠的代码里去吧，那里有无穷的乐趣等着我们。

## 4.3.拓展：扩展眼界的乐趣

我一直鼓励身边共事的开发人员多学习一些编程语言，不一定在工作中用，但起码能够见识一下另一种思维方式。这不仅能扩宽眼界，我们更能从中体会到这个职业的乐趣.

## 4.4.协作：大制作的乐趣

钢琴王子的独奏固然经典，气势磅礴的交响乐同样能博得喝彩。跟交响乐一样，软件工程演奏的关键同样是配合。

大制作的软件产品是任何独行侠无法完成的，一个人的精力有限兴趣狭隘，不可能达到面面俱到，也懒于照顾上至UI体验下至数据库优化的每一个细节。这正是我等发挥的乐园。 如何在协作中取得成就感，获得乐趣，正是我们现在不断尝试和孜孜追求的东西，它需要我们共同的努力。

# UNIX 编程艺术

构成文化的是人，一直以来，获取文化的方式大约是：口口相传、潜移默化。

每过 18 个月，就有一半的知识会过时。

在 UNIX 世界里，操作系统以成就感而不是挫折感来回报人们的努力。

UNIX 的哲学：一个程序只做一件事，并把它做好。程序要能协作。

Rob Pike 用六个原则道出了 UNIX 的哲学

* 原则 1： 你无法断定程序会在什么地方耗费运行时间。瓶颈经常出现在想不到的地方，所以别急于胡乱找个地方改代码，除非你已经证实那儿就是瓶颈所在。
* 原则 2： 估量。在你没对代码进行估量，特别是没找到最耗时的那部分之前，别去优化速度。
* 原则 3： 花哨的算法在 n 很小时通常很慢，而 n 通常很小。花哨算法的常数复杂度很大。除非你确定 n 总是很大，否则不要用花哨算法（即使 n 很大，也优先考虑原则 2）排序时 n = 10？还是用冒泡吧！）
* 原则 4： 花哨的算法比简单的算法更容易出 bug、更难实现。尽量使用简单的算法配合简单的数据结构。Ken Thompson 强调说：拿不准就穷举。
* 原则 5： 数据压倒一切。如果已经选择了正确的数据结构并且把一切都组织得井井有条，正确的算法也就不言自明。编程的核心是数据结构，而不是算法。
* 原则 6： 没有原则。

宁花机器一分，不花程序员一秒。

“用错误的方法解决正确的问题总比用正确的方法解决错误的问题好。”——Doug Mcllroy

那些伪称能够给他们提供间接性的操作系统（windows）往往更能吸引人，人们宁愿用几个按钮选择少得可怜的功能，而懒得用文本去配置自己所想要的一切

计算机编程的本质就是控制复杂度——Brain Kernighan

用清晰的接口把若干简单的模块组成一个复杂软件。如此一来，多数问题只会局限于某个局部，那么就还有希望对局部进行改进而不至于牵动全身。

若程序没有什么特别之处可讲，就保持沉默。行为良好的程序应该默默工作，绝不唠唠叨叨，碍手碍脚。沉默是金。

程序应该只在有有足够理由怀疑答案是“不不不！”的时候请求确认。

老是告诉用户他们已经知道的事情是非常糟糕的风格，比如程序 <foo> 正在启动。不但询问的答案几乎都为“是”的请求确认会造成用户根本不假思索地点击“是”，这个习惯会带来非常不幸的结果。

先制作原型，再精雕细琢。优化之前先确保能用。

“过早优化是万恶之源。”——Donald knuth

“先求运行，再求正确，最后求快”——Kent Beck

Keep It Simple and Stupid

当 Dennis Ritchie 告诉所有人 C 中函数调用的开销很小很小之后，人人都开始编写小函数，搞模块化。然而几年以后，人们发现在 PDP-11 中函数调用开销仍然昂贵，但那个时候那大家已经欲罢不能了。

一些有能力的开发者，一开始总是定义接口，然后编写简要注释，对其进行描述，最后才编写代码——因为编写注释的过程就阐明了代码必须达到的目的。这种描述能够帮助你组织思路，本身就是十分有用的模块说明，而且，最终你有可能还想把这些说明做出路标文档（road document），方便以后的人阅读。

紧凑的软件工具让人乐于使用，不会在你的想法和工作之间格格不入，使你工作起来更有成效——完全不像那些蹩脚的工具，用着别扭，甚至还会把你弄伤。Lisp 语言就是这样一个经典的例子。在通用编程语言中，C 和 Python 是半紧凑的；Perl，Java，Emacs Lisp 和 shell 则不是。C++ 是反紧凑的型的，Bjarne Stoustrup 已经承认，根本没人能完全理解 C++。

SPOT（Single Point of Truth）原则倡导寻找一种数据结构，使得模型中状态和真实世界系统的状态能够一一对应。

UNIX 中非常有效的工具都隐含了一个强大的核心算法，比如通过模式匹配从文件中挑选文本的 grep（围绕正则表达式 regular-expression 模式的形式代数问题），和用于生产语法分析器  yacc（围绕 LR 语法形式理论）。

有时候不能找到绝对正确的解决方法，例如，一个算法上完美的垃圾邮件过滤器需要完全解决自然语言的理解问题（但是事实上我们只需要一个有效的算法使其正确率达到 99% 就行了，又比如操作系统在管理内存时也很难做到理论上的完美）

自顶向下的设计者通常先考虑程序的主事件循环，以后才插入具体事件。自底向上的设计者通常先考虑封装具体的任务，以后再按照某种相关次序把这些东西粘合在一起。

程序员工具箱中最强大的优化技术就是不做优化。最聪明、最便宜、常常也是最迅速的性能提升方法，就是等上几个月，期望硬件性能更好。

UNIX 程序员追求简单的激情，源自注重成效的事实：复杂度就是成本。复杂的软件更难于开发，难于测试，难于调试，难于维护——最重要的，难以学习和使用。而复杂所带来的成本，开发时便汹涌而来，部署后更变本加厉。复杂是 bug 滋生的温床，在整个的软件生命周期，世界都将不得安宁。

在UNIX 传统中存在一个强烈的偏好，宁可去掉功能，也不接受肯能复杂性。

用软件包名字或者主题关键字加上 HOWTO 或 FAQ 字符串，用它作为关键字在网上搜索。对初学者来说，通常会发现比手册更有用的资料。

UNIX 开发者喜欢清晰、简单的操作，用户告诉什么他们就做什么，即使用户指令等于“向我开枪”的命令。

# 单元测试的优点

**是一种验证行为**

* 完成的每一项功能都有单元测试来验证它的正确性
* 通过运行单元测试快速的定位问题
* 不用担心修改导致程序结构被破坏
* 不用担心需求变更导致产品不稳定
* 有单元测试做保障，任何人都可以维护项目

**是一种设计行为（test drive design），迫使我们**

* 从调用者角度考虑
* 把程序设计成为可调用、可测试的
* 解除模块之间的耦合

**是一种编写文档的行为**

* 方法接口如何使用的最佳文档
* 文档是可编译、可运行的
* 保持最新、永远与代码同步\

**具有回归性**

* 单元测试用例可以随时、自动化、批量地运行
* 是我们的软件能够持续构建的基石

其它

* **勇气：**单元测试是自动化的回归测试，她让我对自己的代码充满自信，每一个测试就像攀岩者钉在峭壁上的一个楔子，没有了程序衰退的担心，于是我可以大胆的重构、积极的拥抱变化；
* **快速反馈**：每写一段代码，我都可以在几秒钟之内看到他的运行效果，免去了打包、部署、重起server以及在一堆日志里找结果的工作，开发的效率极大提高；
* **测试驱动设计：**通过编写测试可以准确的理解需求、发现问题、发现接口，在不知不觉间做出最合理的设计；
* **文档**：测试是最好的详细设计文档，不会过时、可运行。

## 一些软件设计的原则

#### Don’t Repeat Yourself (DRY)

DRY 是一个最简单的法则，当我们在两个或多个地方的时候发现一些相似的代码的时候，我们需要把他们的共性抽象出来形一个唯一的新方法。

#### Keep It Simple, Stupid (KISS)

设计上，复杂的东西越来越被众人所BS了，而简单的东西越来越被人所认可。

#### Program to an interface, not an implementation

注重/依赖接口，而不是实现。接口是抽象是稳定的，实现则是多种多样的。

#### Command-Query Separation (CQS)  – 命令-查询分离原则： 读写分离。

#### Common Closure Principle（CCP）– 共同封闭原则

当因为某个原因需要修改时，把需要修改的范围限制在一个最小范围内的包里。

#### Hollywood Principle – 好莱坞原则

好莱坞原则就是一句话——“don’t call us, we’ll call you.”好莱坞的经纪人们不希望你去联系他们，而是他们会在需要的时候来联系你。也就是说，所有的组件都是被动的，所有的组件初始化和调用都由容器负责。组件处在一个容器当中，由容器负责管理。

简单的来讲，就是由容器控制程序之间的关系，而非传统实现中，由程序代码直接操控。这也就是所谓**“控制反转”**的概念所在：

* 不创建对象，而是描述创建对象的方式。
* 在代码中，对象与服务没有直接联系，而是容器负责将这些联系在一起。

**控制权由应用代码中转到了外部容器，控制权的转移，是所谓反转。**

好莱坞原则就是IoC（Inversion of Control）或DI（Dependency Injection ）的基础原则。

#### High Cohesion & Low/Loose coupling & – 高内聚， 低耦合

这是UNIX操作系统设计的经典原则，把模块间的耦合降到最低，而努力让一个模块做到精益求精。

* 内聚：一个模块内各个元素彼此结合的紧密程度
* 耦合：一个软件结构内不同模块之间互连程度的度量

内聚意味着重用和独立，耦合意味着多米诺效应牵一发动全身。

#### Convention over Configuration（CoC）– 惯例优于配置原则

简单点说，就是将一些公认的配置方式和信息作为内部缺省的规则来使用。例如，Hibernate的映射文件，如果约定字段名和类属性一致的话，基本上就可以不要这个配置文件了。

#### Design by Contract (DbC) – 契约式设计

#### Acyclic Dependencies Principle (ADP) – 无环依赖原则

包之间的依赖结构必须是一个直接的无环图形，也就是说，在依赖结构中不允许出现循环依赖。

## 单一职责原则（Single-Responsibility Principle）

**“**对一个类而言，应该仅有一个引起它变化的原因”。本原则是我们非常熟悉地“高内聚性原则”的引申，但是通过将“职责”极具创意地定义为“变化的原因”，使得本原则极具可操作性，尽显大师风范。同时，本原则还揭示了内聚性和耦合性是“一物两面”的关系，为了降低耦合性，基本途径就是提高内聚性；如果一个类承担的职责过多，那么这些职责就会相互依赖，一个职责的变化可能会影响另一个职责的履行。其实OOD（object orient design）的实质，就是合理地进行类的职责分配。

**开放封闭原则（Open-Closed Principle）。**

**“**软件实体应该是可以扩展的，但是不可修改”。本原则紧紧围绕变化展开，变化来临时，如果不必改动软件实体的源代码，就能扩充它的行为，那么这个软件实体的设计就是满足开放封闭原则的。如果我们预测到某种变化，或者某种变化发生了，我们应当创建抽象来隔离以后发生的同类变化。在Java中，这种抽象指抽象基类或接口。

**Liskov替换原则（Liskov-Substitution Principle）。**

**“**子类型必须能够替换掉它们的基类型”。本原则和开放封闭原则关系密切，正是子类型的可替换性，才使得使用基类型的模块无需修改就可扩充。Liskov替换原则从基于契约的设计演化而来，契约通过为每个方法声明“先验条件”和“后验条件”；定义子类时，必须遵守这些“先验条件”和“后验条件”。当前，基于契约的设计发展势头正劲，对实现“软件工厂”的“组装生产”梦想是一个有力的支持。

**依赖倒置原则（Dependency-Inversion Principle）。**

**“**抽象不应依赖于细节，细节应该依赖于抽象”。本原则几乎就是软件设计的正本清源之道。因为人解决问题的思考过程是先抽象后具体，从笼统到细节的，所以我们先生产出的势必是抽象程度比较高的实体，而后才是更加细节化的实体。于是，“细节依赖于抽象”就意味着后来的依赖于先前的，这是自然而然的重用之道。而且，抽象的实体代表着笼而统之的认识，人们总是比较容易正确认识它们，而且它们本身也是不易变的，依赖于它们是安全的。依赖倒置原则适应了人类认识过程的规律，是面向对象设计的标志所在。

**接口隔离原则（Interface-Segregation Principle）。**

**“**多个专用接口优于一个单一的通用接口”。本原则是单一职责原则用于接口设计的自然结果。一个接口应该保证，实现该接口的实例对象可以只呈现为单一的角色；这样，当某个客户程序的要求发生变化，而迫使接口发生改变时，影响到其它客户程序的可能性最小。

**良性依赖原则**。 “不会在实际中造成危害的依赖关系，都是良性依赖”。

**设计正在“腐烂”的征兆（Symptoms of Rotting Design）**

**有四个主要的征兆告诉我们该软件设计正在“腐烂”中。它们并不是互相独立的，而是互相关联，它们是过于僵硬、过于脆弱、不可重用性和粘滞性过高。**

**1. 过于僵硬：致使软件难以更改，每一个改动都会造成一连串的互相依靠的模块的改动，项目经理不敢改动，因为他永远也不知道一个改动何时才能完成。**

**2. 过于脆弱：致使当软件改动时，系统会在许多地方出错。并且错误经常会发生在概念上与改动的地方没有联系的模块中。这样的软件无法维护，每一次维护都使软件变得更加难以维护。（恶性循环）**

**3. 不可重用性：致使我们不能重用在其它项目中、或本项目中其它位置中的软件。工程师发现将他想重用的部分分离出来的工作量和风险太大，足以抵消他重用的积极性，因此软件用重写代替了重用。**

**4. 粘滞性过高：有两种形式：设计的粘滞性和环境的粘滞性.当需要进行改动时，工程师通常发现有不止一个方法可以达到目的。但是这些方法中，一些会保留原有的设计不变，而另外一些则不会。一个设计如果使工程师作错比作对容易得多，那么这个设计的粘滞性就会很高。**

# 系统管理员的 18 个基本准则

**靠规则而活**

     不是仅仅只要知道怎么建立维护服务器和理解系统命令是怎么工作的就可以让你成为一个好的系统管理员—甚至也不是知道当系统宕掉时怎么去修复，怎么去监控系能，怎么去管理备份或者怎么可以写出漂亮的脚本。而是除此之外还要为自己制定一套能让系统良好运行以及让你用户高兴的规则。

可能其中很多你已经听过无数次，也可能有些是你遇到问题时吸取的经验教训。这些规则在过去几十年的系统管理中都已经证明了它们的价值并且能够帮助我们在脑子发热时冷静下来。

**禁止做任何不能回滚的操作**

  除了一些最简单的修改，你应该备有回滚方案。当你进行改动时想好了回滚方案吗？有很多办法可以在修改的途径中留下技术上的“面包屑”，这能够让你随时回到你开始的原点。备份你需要编辑的文件，可能是你记不住的一些复杂的配置文件。记录下遇到的问题。在进行生产环境之前先在测试环境下测试，在你继续之前确保所有的修改都是合理的。

     提前计划好需要改动的地方，最好能够采取同行评审的方式。另外一双眼睛可能可以看到你没有考虑到的问题。

**避免在周五做任何改动**

  不要在你将要不在的几天之前做任何修改。确保在不需要干预的情况下这些修改也可以在系统上正常运行。

**弄清楚根本原因**

去挖掘你所碰到问题的根本原因。当怀疑的时候，用“五个为什么”原则。我的服务器宕了，为什么？是因为内存不够。为什么不够？因为其中一个进程“疯了”。为什么“疯了”？因为它进入到循环中。为什么进入？因为配置文件中有错误。为什么有错误？因为我在周五晚上离开之前修改了文件，但是忘了测试确保所有一切都可以正常工作。

**实践灾备方案**

实践灾备方案在必要的时候能够顺手使用。如果你不实践，有两种可能会发生。第一，你没有信心你的方案是否有用，第二，你可能会不确定你所要采取的措施。比如说，你需要迁移数据库到一个远程站点。你是否知道你要运行的命令？准备好数据备份了吗？还是需要去创建备份？你是否知道迁移文件需要多久？你是否会准备好远程启动数据库？你是否有一套测试来检验其能够很好地运行？

**不要去依赖没有完整测试过的脚本**

这很容易犯错的。即使你已经写了几十年的脚本也要去测试，特别是可能别人在某天会运行这些脚本。带参数测试和不带参数测试。模拟其他人可能会犯的错误。总之，必须要测试脚本。

**三次以上的重复和复杂的操作，必须自动化**

**为你的工作建立文档**

     用文档记录下你的日常工作。你做的事里面有哪些对别人来说不好理解？可能你要跑个脚本在日志文件中查找磁盘吃紧或者数据中心湿度太高的警告。

     往脚本中添加注释。你可能会觉得你用的那些命令很显而易见，但是当你隔了一两年没用再回去用的时候就不会那么显而易见了。不要为了简洁而牺牲了可读性。别人可能会要读你的代码。完整地写下你所做的一切，别人能够在你决定跳槽时候很好地接手你的工作。

**重视你犯的错误**

     用你自己的方式去理解错误是避免再犯的唯一方法。找到一种可靠的方式来提醒自己容易忘记的事。

**强迫自己轻微妄想症**

     经常问自己问题，比如“别人会不会误用这个呢？”，“别人会不会弄坏这个呢？”，以及“这个服务会怎么收到攻击？”给所有的脚本设置权限使除管理员以外其他人都无权查看。想一些防守措施能够让你减少很多痛苦。而且，对管理服务器来说，妄想症是优点。

**未雨绸缪**

     并不是每一个问题都会找上你的门。利用空余时间去“招惹”问题并且确保系统还是能够完好运行。想想系统会出问题的各种可能性并且检查这些问题是否已经存在。

     尽可能自动化去检查问题，但是要确保问题在你的掌控之中，你要注意本应该看到的报警是否如期到来。收到你最关键的服务器宕机警报不能让你得到任何奖励。

**特别注意安全性**

     安全方面的投入是要与你所保护的数据成比例的。要知道你在保护什么，知道你所保护数据的“主人”。运用一些最佳的实践方案，比如最小特权、定期补丁、监控关键服务以及漏洞检测。仅仅只运行你所需要的服务。时刻警惕任何强行闯入或者系统受损的迹象。准备好上报渠道能够让你知道什么时候以及向谁汇报这些系统受损现象。

**不要忽视日志文件**

     例行检查日志文件能够在正常的服务器和服务受威胁之前发出警告。检查错误和警告。使用一个监控日志的工具或者自己写脚本。没有人有时间去看你日志中所有的信息。

**备份一切**

     制定一个好的备份策略即使服务器已经做了同步。一个同步服务器上的错误同样也是错误。测试好备份系统。确保在你要使用它们之前是没有问题的。在你能够承担的情况下多雇佣一位员工。尽可能零容忍单点故障，即使是你自己。

**像自己的时间一样考虑其他人的时间**

     系统管理员有点像一个自大的小男孩。我们在自己的神奇领域是奇才。但是尽管如此，我们还是要准时参加会议并且别人向你寻求帮助的时候及时回复别人，即使只是说一句我正在解决他们的问题。礼貌地对待顾客，即使他们找不到怎么运行命令行。他们可能在他们自己领域是个专家级魔术师，如果他们不是的，那就正好说明我们是如此的重要。

**保持时刻通知用户**

     确保用户能够预料将会发生什么，特别是计划会有重大更新时。这些可以让他们对你抱有信心，并且信任他们所依赖的服务。交流、透明化、使用票务系统还有就是要留意多长时间能够解决问题。

**避免太过自我，让别人喜欢你**

两个和尚住在隔壁，所谓隔壁就是隔壁那座山，他们分别住在相邻的两座山上的庙里。两座山之间有一条溪，这两个和尚每天都会在同一时间下山去溪边挑水，久而久之，他们便成为了好朋友。

      就这样，时间在每天挑水中不知不觉已经过了5年。突然有一天，左边这座山的和尚没有下山挑水，右边那座山的和尚心想：“他大概睡过头了。”便不以为意。

      哪知道第二天左边这座山的和尚还是没有下山挑水，第三天也一样。过了一个星期还是一样，直到过了一个月，右边那座山的和尚终于忍不住了，他心想：“我的朋友可能生病了，我要过去拜访他，看看能帮上什么忙。”

      于是他便爬上了左边这座山，去探望他的老朋友。

      等他到了左边这座山的庙里，看到他的老友之后大吃一惊，因为他的老友正在庙前打太极拳，一点也不像一个月没喝水的人。他很好奇地问：“你已经一个月没有下山挑水了，难道你可以不用喝水吗？”左边这座山的和尚说：“来来来，我带你去看。”于是他带着右边那座山的和尚走到庙的后院，指着一口井说： “这5年来，我每天做完功课后都会抽空挖这口井，即使有时很忙，能挖多少就算多少。如今终于让我挖出井水，我就不用再下山挑水，我可以有更多时间练我喜欢的太极拳。”

很多人用一种打工的心态过日子，在公司领薪水。薪水领得再多，那都是“挑水”。人的力气总是有限的，年纪大了，挑水就常常体力不支。  
     很多人之所以人生旅途不如意，或者早年春风得意，老来则风烛残年，就是因为一辈子都在挑水，不懂得挖一口属于自己的井；  
     很多人知道需要挖井，却不知道如何挖井；  
 很多人知道如何挖井，却不知道去哪里挖井，挖什么井；  
 很多人知道去哪里挖井，挖什么井，却总是半途而废。。。。。。

     没想到自己需要一口井、不挖井、不知道去哪里挖井、不知道挖什么井、不知道如何挖井……总是有各种各样的借口或理由；如何在工作之中或工作之余找井源、找工具、挖井，一定有方法和步骤……

       还有一部分人知道应该挖一口属于自己的井，而且一辈子都很努力，但他们之中大多数没有挖出井来，因为他们并不了解，其实，很多东西原本就是“命中注定”的……

作为一名普通的程序开发人员，每天的程序员工作就如同挑水一样，如果只把目光放在每一天的程序开发上，就会一辈子在挑水，并且总有挑不动的那一天。所以要去追求一口属于自己的井，也许最终无法得到，但毕竟努力过了，也人生无憾矣。

因此在工作之余，要利用自己所熟悉的技能，在自己所熟悉的领域，慢慢的钻研和积累，其实就是那句古话：不积小流无以成江海，不积跬步无以至千里。不要因为今天工作加班了，不要因为今天工作累了，不要因为那个朋友来了，等等这样的理由为借口，当然，作为一个现实生活中的人，这些日常的人际交流都是合理的也是必不可少的，只是我们应该利用每一个点滴可以利用的时间去[学习](http://www.html5cn.org/portal.php?mod=list&catid=13" \t "_blank)、去思考、获得知识技能的同时能够提高自己的思想高度。

     不只是程序员，每个人，都该去追求一口属于自己的井。

**为什么要创业（Basecamp）**

我想为自己工作。

     按照自己的节奏。

    走我自己的路。

     我想由我来定义它，不去考虑其他生意人对它的盈利期许。

     关于公司独立性的陈词滥调其实听起来都挺稀奇古怪的，但当你在董事会上被只问如何加快增长，燃得更猛，就像坐上了超音速飞机那样时，你才会了解独立性的好。

     公司的独立性如氧气一般，当它存在时会自然得让你感觉不到其存在。而一旦它消失，鉴于你拿了投资人的钱，你便要接受大师们的指点江山了。并且绝大多创业公司最终都会将公司的独立性双手奉上。

**我当初做产品只是想将它卖给懂得欣赏它品质的人。而品质可能是盈利和用户服务质量之间唯一可能存在的联系。**这和你为了要抓眼球，继而将用户的注意力、隐私和尊严批量打包出卖的行为是完全不同的。

     独立性就是诚信做事，很简单，诚信工作而已。我做一个好产品，你给我一个好价钱。极简的逻辑。我当初做产品时想要落地生根，与同事、顾客、产品都建立起长期的关系。如果我在这些关系之间加入了VC，那它只会成为定时炸弹一般的存在。

创业： 用最初的心，做最永远的事。不被路边美景迷了双眼，坚定走自己的路。

愿景：What I want

天时：When I start    等待时机

地利：How to do      因势利导，顺势而为

人和：Who can help  团结一切可以团结的力量

从内心深处认可自己做的事情，追随自己内心的声音，踏踏实实的按照自己的计划学习、工作、进步。

一些非技术出身的老板，创业者，希望研发人员实现某个产品，功能特性一大堆，以为完成功能就万事大吉，其实，这只是万里长征第一步。很多外包开发，仅限于功能实现！！！

功能实现，其实只是最初级最原始的技术工作。**之后的技术问题，至少有：**

**第一，性能，负载支撑能力。**

可以支持多少用户什么类型的请求，比如每秒多少次浏览，多少次的下单，数据可以支撑多少量级，多大规模。性能瓶颈会在什么规模的数据量和什么规模的请求频次出现。

**第二，稳定性，容错性。**

是否可以在异常输入、异常网络环境中，系统保持稳定健康，不会因为某些偶发异常请求或网络异常导致系统崩溃，雪崩效应，以前我分析数据库案例的时候提过这样的场景。

**第三，系统安全性，灾难备份体系。**

入侵风险，盗号风险，以及即便导致入侵，导致扒库，是否可以有灾难恢复机制以及是否能保持敏感资料不会那么容易被获取。

**第四，可持续维护性。**

后续的升级、维护，换人接手是否可以有足够的维护保障。是否能做到低耦合高复用，能否轻松扩展支持新的业务类型。

**第五，业务安全性，风控体系。**

拼多多的事情大家都看到了，不管具体背后的起因是什么，拼多多业务风控肯定是有不到位的地方。

第一，研发不是一个人打天下的，当然这样的人才不是没有，但大部分情况下，客户端、服务端，是不同的技术人员搭配。

第二，**相当比例大公司背景的研发管理者，不会省钱**

第三，人员是有流动性的，系统的长期可持续维护，需要随时有可替代的接班人，而创业团队，这个条件基本不具备。

第四，研发与产品的配合是需要长期磨合