如何编写无法维护的代码

让自己稳拿铁饭碗 ;-)

-- Roedy Green
（老码农翻译，略有删节）

简介

> *永远不要（把自己遇到的问题）归因于（他人的）恶意，这恰恰说明了（你自己的）无能。* -- 拿破仑

为了造福大众，在Java编程领域创造就业机会，兄弟我在此传授大师们的秘籍。这些大师写的代码极其难以维护，后继者就是想对它做最简单的修改都需要花上数年时间。而且，如果你能对照秘籍潜心修炼，你甚至可以给自己弄个铁饭碗，因为除了你之外，没人能维护你写的代码。再而且，如果你能练就秘籍中的**全部**招式，那么连你自己都无法维护你的代码了！

你不想练功过度走火入魔吧。那就不要让你的代码**一眼看去**就完全无法维护，只要它**实质上是**那样就行了。否则，你的代码就有被重写或重构的风险！

总体原则

> *Quidquid latine dictum sit, altum sonatur.*
> (随便用拉丁文写点啥都会显得高大上。)

想挫败维护代码的程序员，你必须先明白他的思维方式。他接手了你的庞大程序，没有时间把它全部读一遍，更别说理解它了。他无非是想快速找到修改代码的位置、改代码、编译，然后就能交差，并希望他的修改不会出现意外的副作用。

他查看你的代码不过是管中窥豹，一次只能看到一小段而已。你要确保他永远看不到全貌。要尽量和让他难以找到他想找的代码。但更重要的是，要让他不能有把握**忽略**任何东西。

程序员都被编程惯例洗脑了，还为此自鸣得意。每一次你处心积虑地违背编程惯例，都会迫使他必须用放大镜去仔细阅读你的每一行代码。

你可能会觉得每个语言特性都可以用来让代码难以维护，其实不然。你必须精心地误用它们才行。

命名

> *"当我使用一个单词的时候" Humpty Dumpty 曾经用一种轻蔑的口气说，"它就是我想表达的意思，不多也不少。“*
> – Lewis Carroll –– 《爱丽丝魔镜之旅》， 第6章

编写无法维护代码的技巧的重中之重是变量和方法命名的艺术。如何命名是和编译器无关的。这就让你有巨大的自由度去利用它们迷惑维护代码的程序员。

妙用 *宝宝起名大全*

买本宝宝起名大全，你就永远不缺变量名了。比如 `Fred` 就是个好名字，而且键盘输入它也省事。如果你就想找一些容易输入的变量名，可以试试 `adsf` 或者 `aoeu`之类。

单字母变量名

如果你给变量起名为a,b,c，用简单的文本编辑器就没法搜索它们的引用。而且，没人能猜到它们的含义。

创造性的拼写错误

如果你必须使用描述性的变量和函数名，那就把它们都拼错。还可以把某些函数和变量名拼错，再把其他的拼对(例如 SetPintleOpening 和 SetPintalClosing)，我们就能有效地将grep或IDE搜索技术玩弄于股掌之上。这招超级管用。还可以混

淆不同语言（比如*colour* –– 英国英语，和 *color* –– 美国英语）。

抽象

在命名函数和变量的时候，充分利用抽象单词，例如 *it*, *everything*, *data*, *handle*, *stuff*, *do*, *routine*, *perform* 和数字，例如 e.g. `routineX48`, `PerformDataFunction`, `DoIt`, `HandleStuff` 还有 `do_args_method`。

首字母大写的缩写

用首字母大写缩写（比如GNU 代表 GNU's Not Unix) 使代码简洁难懂。真正的汉子(无论男女)从来不说明这种缩写的含义，他们生下来就懂。

辞典大轮换

为了打破沉闷的编程气氛，你可以用一本辞典来查找尽量多的同义词。例如 *display*, *show*, *present*。在注释里含糊其辞地暗示这些命名之间有细微的差别，其实根本没有。不过，如果有两个命名相似的函数真的有重大差别，那倒是一定要确保它们用相同的单词来命名(例如，对于 "写入文件", "在纸上书写" 和 "屏幕显示" 都用 *print* 来命名)。 在任何情况下都不要屈服于编写明确的项目词汇表这种无理要求。你可以辩解说，这种要求是一种不专业的行为，它违反了结构化设计的*信息隐藏原则*。

首字母大写

随机地把单词中间某个音节的首字母大写。例如 `ComputeReSult()`。

重用命名

在语言规则允许的地方，尽量把类、构造器、方法、成员变量、参数和局部变量都命名成一样。更高级的技巧是在{}块中重用局部变量。这样做的目的是迫使维护代码的程序员认真检查每个示例的范围。特别是在Java代码中，可以把普通方法伪装成构造器。

使用非英语字母

在命名中偷偷使用不易察觉的非英语字母，例如

```
    typedef struct { int i; } ínt;
```

看上去没啥不对是吧？嘿嘿嘿...这里的第二个 ínt 的 í 实际上是东北欧字母，并不是英语中的 i 。在简单的文本编辑器里，想看出这一点点区别几乎是不可能的。

巧妙利用编译器对于命名长度的限制

如果编译器只区分命名的前几位，比如前8位，那么就把后面的字母写得不一样。比如，其实是同一个变量，有时候写成 *var_unit_update()* ，有时候又写成 *var_unit_setup()*，看起来是两个不同的函数调用。而在编译的时候，它们其实是同一个变量 *var_unit*。

下划线，一位真正的朋友

可以拿 _ 和 __ 作为标示符。

混合多语言

随机地混用两种语言（人类语言或计算机语言都行）。如果老板要求使用他指定的语言，你就告诉他你用自己的语言更有利于组织你的思路，万一这招不管用，就去控诉这是语言歧视，并威胁起诉老板要求巨额精神损失赔偿。

扩展 ASCII 字符

扩展 ASCII 字符用于变量命名是完全合法的，包括 ß, Ð, 和 ñ 等。在简单的文本编辑器里，除了拷贝/粘贴，基本上没法输入。

其他语言的命名

使用外语字典作为变量名的来源。例如，可以用德语单词 *punkt* 代替 *point*。除非维护代码的程序员也像你一样熟练掌握了德语. 不然他就只能尽情地在代码中享受异域风情了。

数学命名

用数学操作符的单词来命名变量。例如：

```
openParen = (slash + asterix) / equals;
(左圆括号 = (斜杠 + 星号)/等号;)
```

令人眩晕的命名

用带有完全不相关的感情色彩的单词来命名变量。例如：

```
marypoppins = (superman + starship) / god;
(欢乐满人间 = (超人 + 星河战队)/上帝;)
```

这一招可以让阅读代码的人陷入迷惑之中，因为他们在试图想清楚这些命名的逻辑时，会不自觉地联系到不同的感情场景里而无法自拔。

何时使用 i

永远不要把 `i` 用作最内层的循环变量。 用什么命名都行，就是别用`i`。把 `i` 用在其他地方就随便了，用作非整数变量尤其好。

惯例 -- 明修栈道，暗度陈仓

忽视 Java 编码惯例，Sun 就是这样做的。幸运的是，你违反了它编译器也不会打小报告。这一招的目的是搞出一些在某些特殊情况下有细微差别的名字来。如果你被强迫遵循驼峰法命名，你还是可以在某些模棱两可的情况下颠覆它。例如，*inputFilename* 和 *inputfileName* 两个命名都可以合法使用。在此基础上自己发明一套复杂到变态的命名惯例，然后就可以痛扁其他人，说他们违反了惯例。

小写的 l 看上去很像数字 1

用小写字母 l 标识 long 常数。例如 10l 更容易被误认为是 101 而不是 10L 。 禁用所有能让人准确区分 uvw wW gq9 2z 5s il17|!j oO08 `'" ;,. m nn rn {[()]} 的字体。要做个有创造力的人。

把全局命名重用为私有

在A 模块里声明一个全局数组，然后在B 模块的头文件里在声明一个同名的私有数组，这样看起来你在B 模块里引用的是那个全局数组，其实却不是。不要在注释里提到这个重复的情况。

误导性的命名

让每个方法都和它的名字蕴含的功能有一些差异。例如，一个叫 `isValid(x)`的方法在判断完参数x的合法性之后，还顺带着把它转换成二进制并保存到数据库里。

伪装

当一个bug需要越长的时间才会暴露，它就越难被发现。
– Roedy Green（本文作者）

编写无法维护代码的另一大秘诀就是伪装的艺术，即隐藏它或者让它看起来像其他东西。很多招式有赖于这样一个事实：编译器比肉眼或文本编辑器更有分辨能力。下面是一些伪装的最佳招式。

把代码伪装成注释，反之亦然

下面包括了一些被注释掉的代码，但是一眼看去却像是正常代码。

```
    for(j=0; j<array_len; j+ =8)
        {
        total += array[j+0 ];
```

```
        total += array[j+1 ];
        total += array[j+2 ]; /* Main body of
        total += array[j+3]; * loop is unrolled
        total += array[j+4]; * for greater speed.
        total += array[j+5]; */
        total += array[j+6 ];
        total += array[j+7 ];
        }
```

如果不是用绿色标出来，你能注意到这三行代码被注释掉了么?

用连接符隐藏变量

对于下面的定义

```
#define local_var xy_z
```

可以把 "xy_z" 打散到两行里:

```
#define local_var xy\
_z // local_var OK
```

这样全局搜索 xy_z 的操作在这个文件里就一无所获了。 对于 C 预处理器来说，第一行最后的 "\" 表示继续拼接下一行的内容。

文档

任何傻瓜都能说真话，而要把谎编圆则需要相当的智慧。
– Samuel Butler (1835 – 1902)

不正确的文档往往比没有文档还糟糕。
– Bertrand Meyer

既然计算机是忽略注释和文档的，你就可以在里边堂而皇之地编织弥天大谎，让可怜的维护代码的程序员彻底迷失。

在注释中撒谎

实际上你不需要主动地撒谎，只要没有及时保持注释和代码更新的一致性就可以了。

只记录显而易见的东西

往代码里掺进去类似于 `/* 给 i 加 1 */` 这样的注释，但是永远不要记录包或者方法的整体设计这样的干货。

记录 How 而不是 Why

只解释一个程序功能的细节，而不是它要完成的任务是什么。这样的话，如果出现了一个bug，修复者就搞不清这里的代码应有的功能。

该写的别写

比如你在开发一套航班预定系统，那就要精心设计，让它在增加另一个航空公司的时候至少有25处代码需要修改。永远不要在文档里说明要修改的位置。后来的开发人员要想修改你的代码门都没有，除非他们能把每一行代码都读懂。

计量单位

永远不要在文档中说明任何变量、输入、输出或参数的计量单位，如英尺、米、加仑等。计量单位对数豆子不是太重要，但在工程领域就相当重要了。同理，永远不要说明任何转换常量的计量单位，或者是它的取值如何获得。要想让代码更乱的话，你还可以在注释里写上错误的计量单位，这是赤裸裸的欺骗，但是非常有效。如果你想做一个恶贯满盈的人，不妨自己发明一套计量单位，用自己或某个小人物的名字命名这套计量单位，但不要给出定义。万一有人挑刺儿，你就告诉他们，你这么做是为了把浮点数运算凑成整数运算而进行的转换。

坑

永远不要记录代码中的坑。如果你怀疑某个类里可能有bug，天知地知你知就好。如果你想到了重构或重写代码的思路，看在老天爷的份上，千万别写出来。切记电影《小鹿斑比》里那句台词 *"如果你不能说好听的话，那就什么也不要说。"*。万一这段代码的原作者看到你的注释怎么办？万一老板看到了怎么办？万一客户看到了怎么办？搞不好最后你自己被解雇了。一句”这里需要修改“的匿名注释就好多了，尤其是当看不清这句注释指的是哪里需要修改的情况下。切记难得糊涂四个字，这样大家都不会感觉受到了批评。

说明变量

永远不要 对变量声明加注释。有关变量使用的方式、边界值、合法值、小数点后的位数、计量单位、显示格式、数据录入规则等等，后继者完全可以自己从程序代码中去理解和整理嘛。如果老板强迫你写注释，就把方法体代码混进去，但绝对不要对变量声明写注释，即使是临时变量！

在注释里挑拨离间

为了阻挠任何雇佣外部维护承包商的倾向，可以在代码中散布针对其他同行软件公司的攻击和抹黑，特别是可能接替你工作的其中任何一家。例如：

```
/* 优化后的内层循环
这套技巧对于SSI软件服务公司的那帮蠢材来说太高深了，他们只会
用 <math.h> 里的笨例程，消耗50倍的内存和处理时间。
*/
class clever_SSInc
    {
    .. .
    }
```

可能的话，除了注释之外，这些攻击抹黑的内容也要掺到代码里的重要部分，这样如果管理层想清理掉这些攻击性的言论然后发给外部承包商去维护，就会破坏代码结构。

程序设计

编写无法维护代码的基本规则就是：在尽可能多的地方，以尽可能多的方式表述每一个事实。
– Roedy Green

编写可维护代码的关键因素是只在一个地方表述应用里的一个事实。如果你的想法变了，你也只在一个地方修改，这样就能保证整个程序正常工作。所以，编写无法维护代码的关键因素就是反复地表述同一个事实，在尽可能多的地方，以尽可能多的方式进行。令人高兴的是，像Java这样的语言让编写这种无法维护代码变得非常容易。例如，改变一个被引用很多的变量的类型几乎是不可能的，因为所有造型和转换功能都会出错，而且关联的临时变量的类型也不合适了。而且，如果变量值要在屏幕上显示，那么所有相关的显示和数据录入代码都必须一一找到并手工进行修改。类似的还有很多，比如由C和Java组成的Algol语言系列，Abundance甚至Smalltalk对于数组等结构的处理，都是大有可为的。

Java 造型

Java的造型机制是上帝的礼物。你可以问心无愧地使用它，因为Java语言本身就需要它。每次你从一个Collection 里获取一个对象，你都必须把它造型为原始类型。这样这个变量的类型就必须在无数地方表述。如果后来类型变了，所有的造型都要修改才能匹配。如果倒霉的维护代码的程序员没有找全（或者修改太多），编译器能不能检测到也不好说。类似的，如果变量类型从`short` 变成 `int`，所有匹配的造型也都要从`(short)` 改成 `(int)`。

利用Java的冗余

Java要求你给每个变量的类型写两次表述。 Java 程序员已经习惯了这种冗余，他们不会注意到你的两次表述有细微的差别，例如

```
    Bubblegum b = new Bubblegom();
```

不幸的是 ++ 操作符的盛行让下面这种伪冗余代码得手的难度变大了：

```
    swimmer = swimner + 1;
```

永远不做校验

永远不要对输入数据做任何的正确性或差异性检查。这样能表现你对公司设备的绝对信任，以及你是一位信任所有项目伙伴和系统管理员的团队合作者。总是返回合理的值，即使数据输入有问题或者错误。

有礼貌，无断言

避免使用 assert() 机制，因为它可能把三天的debug盛宴变成10分钟的快餐。

避免封装

为了提高效率，不要使用封装。方法的调用者需要所有能得到的外部信息，以便了解方法的内部是如何工作的。

复制粘贴修改

以效率的名义，使用 复制+粘贴+修改。这样比写成小型可复用模块效率高得多。在用代码行数衡量你的进度的小作坊里，这招尤其管用。

使用静态数组

如果一个库里的模块需要一个数组来存放图片，就定义一个静态数组。没人会有比512 X 512 更大的图片，所以固定大小的数组就可以了。为了最佳精度，就把它定义成 double 类型的数组。

傻瓜接口

编写一个名为 "WrittenByMe" 之类的空接口，然后让你的所有类都实现它。然后给所有你用到的Java 内置类编写包装类。这里的思想是确保你程序里的每个对象都实现这个接口。最后，编写所有的方法，让它们的参数和返回类型都是这个 WrittenByMe。这样就几乎不可能搞清楚某个方法的功能是什么，并且所有类型都需要好玩的造型方法。更出格的玩法是，让每个团队成员编写它们自己的接口(例如 WrittenByJoe)，程序员用到的任何类都要实现他自己的接口。这样你就可以在大量无意义接口中随便找一个来引用对象了。

巨型监听器

永远不要为每个组件创建分开的监听器。对所有按钮总是用同一个监听器，只要用大量的if...else 来判断是哪一个按钮被点击就行了。

好事成堆™

狂野地使用封装和OO思想。例如

```
    myPanel.add( getMyButton() );
    private JButton getMyButton()
        {
        return myButton;
        }
```

这段很可能看起来不怎么好笑。别担心，只是时候未到而已。

友好的朋友

在C++ 里尽量多使用friend声明。再把创建类的指针传递给已创建类。现在你不用浪费时间去考虑接口了。另外，你应该用上关键字*private* 和 *protected* 来表明你的类封装得很好。

使用三维数组

大量使用它们。用扭曲的方式在数组之间移动数据，比如，用arrayA里的行去填充arrayB的列。这么做的时候，不管三七二十一再加上1的偏移值，这样很灵。让维护代码的程序员抓狂去吧。

混合与匹配

存取方法和公共变量神马的都要给他用上。这样的话，你无需调用存取器的开销就可以修改一个对象的变量，还能宣称这个类是个"Java Bean"。对于那些试图添加日志函数来找出改变值的源头的维护代码的程序员，用这一招来迷惑他尤其有效。

没有秘密!

把每个方法和变量都声明为 public。毕竟某个人某天可能会需要用到它。一旦方法被声明为public 了，就很难缩回去。对不？这样任何它覆盖到的代码都很难修改了。它还有个令人愉快的副作用，就是让你看不清类的作用是什么。如果老板质问你是不是疯了，你就告诉他你遵循的是经典的透明接口原则。

全堆一块

把你所有的没用的和过时的方法和变量都留在代码里。毕竟说起来，既然你在1976年用过一次，谁知道你啥时候会需要再用到呢？当然程序是改了，但它也可能会改回来嘛，你"不想要重新发明轮子"（领导们都会喜欢这样的口气）。如果你还原封不动地留着这些方法和变量的注释，而且注释写得又高深莫测，甭管维护代码的是谁，恐怕都不敢对它轻举妄动。

就是 Final

把你所有的叶子类都声明为 final。毕竟说起来，你在项目里的活儿都干完了，显然不会有其他人会通过扩展你的类来改进你的代码。这种情况甚至可能有安全漏洞。 java.lang.String 被定义成 final 也许就是这个原因吧？如果项目组其他程序员有意见，告诉他们这样做能够提高运行速度。

避免布局

永远不要用到布局。当维护代码的程序员想增加一个字段，他必须手工调整屏幕上显示所有内容的绝对坐标值。如果老板强迫你使用布局，那就写一个巨型的 GridBagLayout 并在里面用绝对坐标进行硬编码。

全局变量，怎么强调都不过分

如果上帝不愿意我们使用全局变量，他就不会发明出这个东西。不要让上帝失望，尽量多使用全局变量。每个函数最起码都要使用和设置其中的两个，即使没有理由也要这么做。毕竟，任何优秀的维护代码的程序员都会很快搞清楚这是一种侦探工作测试，有利于让他们从笨蛋中脱颖而出。

再一次说说全局变量

全局变量让你可以省去在函数里描述参数的麻烦。充分利用这一点。在全局变量中选那么几个来表示对其他全局变量进行操作的类型。

局部变量

永远不要用局部变量。在你感觉想要用的时候，把它改成一个实例或者静态变量，并无私地和其他方法分享它。这样做的好处是，你以后在其他方法里写类似声明的时候会节省时间。C++程序员可以百尺竿头更进一步，把所有变量都弄成全局的。

配置文件

配置文件通常是以 关键字 = 值 的形式出现。在加载时这些值被放入 Java 变量中。最明显的迷惑技术就是把有细微差别的名字用于关键字和Java 变量.甚至可以在配置文件里定义运行时根本不会改变的常量。参数文件变量和简单变量比，维护它的代码量起码是后者的5倍。

子类

对于编写无法维护代码的任务来说，面向对象编程的思想简直是天赐之宝。如果你有一个类，里边有10个属性（成员/方法），可以考虑写一个基类，里面只有一个属性，然后产生9层的子类，每层增加一个属性。等你访问到最终的子类时，你才能得到全部10个属性。如果可能，把每个类的声明都放在不同的文件里。

编码迷局

迷惑 C

从互联网上的各种混乱C 语言竞赛中学习，追随大师们的脚步。

追求极致

总是追求用最迷惑的方式来做普通的任务。例如，要用数组来把整数转换为相应的字符串，可以这么做：

```
char *p;
switch (n)
{
case 1:
    p = "one";
    if (0)
case 2:
    p = "two";
    if (0)
case 3:
    p = "three";
    printf("%s", p);
    break;
}
```

一致性的小淘气

当你需要一个字符常量的时候，可以用多种不同格式： ' ', 32, 0x20, 040。在C或Java里10和010是不同的数（0开头的表示16进制），你也可以充分利用这个特性。

造型

把所有数据都以 void * 形式传递，然后再造型为合适的结构。不用结构而是通过位移字节数来造型也很好玩。

嵌套 Switch

Switch 里边还有 Switch，这种嵌套方式是人类大脑难以破解的。

利用隐式转化

牢记编程语言中所有的隐式转化细节。充分利用它们。数组的索引要用浮点变量，循环计数器用字符，对数字执行字符串函数调用。不管怎么说，所有这些操作都是合法的，它们无非是让源代码更简洁而已。任何尝试理解它们的维护者都会对你感激不尽，因为他们必须阅读和学习整个关于隐式数据类型转化的章节，而这个章节很可能是他们来维护你的代码之前完全忽略了的。

分号!

在所有语法允许的地方都加上分号，例如：

```
if(a);
else;
    {
    int d;
    d = c;
    }
    ;
```

使用八进制数

把八进制数混到十进制数列表里，就像这样：

```
array = new int []
    {
    111,
    120,
    013,
    121,
    };
```

嵌套

尽可能深地嵌套。优秀的程序员能在一行代码里写10层()，在一个方法里写20层{}。

C数组

C编译器会把 `myArray[i]` 转换成 `*(myArray + i)`，它等同于 `*(i + myArray)` 也等同于 `i[myArray]`。 高手都知道怎么用好这个招。可以用下面的函数来产生索引，这样就把代码搞乱了：

```
int myfunc(int q, int p) { return p%q; }
...
myfunc(6291, 8)[Array];
```

遗憾的是，这一招只能在本地C类里用，Java 还不行。

放长线钓大鱼

一行代码里堆的东西越多越好。这样可以省下临时变量的开销，去掉换行和空格还可以缩短源文件大小。记住，要去掉运算符两边的空格。优秀的程序员总是能突破某些编辑器对于255个字符行宽的限制。

异常

我这里要向你传授一个编程中鲜为人知的秘诀。异常是个讨厌的东西。良好的代码永远不会出错，所以异常实际上是不必要的。不要把时间浪费在这上面。子类异常是给那些知道自己代码会出错的低能儿用的。在整个应用里，你只用在main()里放一个try/catch，里边直接调用 System.exit()就行了。在每个方法头要贴上标准的抛出集合定义，到底会不会抛出异常你就不用管了。

使用异常的时机

在非异常条件下才要使用异常。比如终止循环就可以用 `ArrayIndexOutOfBoundsException`。还可以从异常里的方法返回标准的结果。

狂热奔放地使用线程

如题。

测试

在程序里留些bug，让后继的维护代码的程序员能做点有意思的事。精心设计的bug是无迹可寻的，而且谁也不知道它啥时候会冒出来。要做到这一点，最简单的办法的就是不要测试代码。

永不测试

永远不要测试负责处理错误、当机或操作系故障的任何代码。反正这些代码永远也不会执行，只会拖累你的测试。还有，你怎么可能测试处理磁盘错误、文件读取错误、操作系统崩溃这些类型的事件呢？为啥你要用特别不稳定的计算机或者用测试脚手架来模拟这样的环境？现代化的硬件永远不会崩溃，谁还愿意写一些仅仅用于测试的代码？这一点也不好玩。如果用户抱怨，你就怪到操作系统或者硬件头上。他们永远不会知道真相的。

永远不要做性能测试

嘿，如果软件运行不够快，只要告诉客户买个更快的机器就行了。如果你真的做了性能测试，你可能会发现一个瓶颈，这会导致修改算法，然后导致整个产品要重新设计。谁想要这种结果？而且，在客户那边发现性能问题意味着你可以免费到外地旅游。你只要备好护照和最新照片就行了。

永远不要写任何测试用例

永远不要做代码覆盖率或路径覆盖率测试。自动化测试是给那些窝囊废用的。搞清楚哪些特性占到你的例程使用率的90%，然后把90%的测试用在这些路径上。毕竟说起来，这种方法可能只测试到了大约你代码的60%，这样你就节省了40%的测试工作。这能帮助你赶上项目后端的进度。等到有人发现所有这些漂亮的“市场特性”不能正常工作的时候，你早就跑路了。一些有名的大软件公司就是这样测试代码的，所以你也应该这样做。如果因为某种原因你还没走，那就接着看下一节。

测试是给懦夫用的

勇敢的程序员会跳过这个步骤。太多程序员害怕他们的老板，害怕丢掉工作，害怕客户的投诉邮件，害怕遭到起诉。这种恐惧心理麻痹了行动，降低了生产率。有科学研究成果表明，取消测试阶段意味着经理有把握能提前确定交付时间，这对于规划流程显然是有利的。消除了恐惧心理，创新和实验之花就随之绽放。程序员的角色是生产代码，调试工作完全可以由技术支持和遗留代码维护组通力合作来进行。

如果我们对自己的编程能力有充分信心，那么测试就没有必要了。如果我们逻辑地看待这个问题，随便一个傻瓜都能认识到测试根本都不是为了解决技术问题，相反，它是一种感性的信心问题。针对这种缺乏信心的问题，更有效的解决办法就是完全取消测试，送我们的程序员去参加自信心培训课程。毕竟说起来，如果我们选择做测试，那么我们就要测试每个程序的变更，但其实我们只需要送程序员去一次建立自信的培训课就行了。很显然这么做的成本收益是相当可观的。

编程语言的选择

计算机语言正在逐步进化，变得更加傻瓜化。使用最新的语言是不人性的。尽可能坚持使用你会用的最老的语言，先考虑用穿孔纸带，不行就用汇编，再不行用FORTRAN 或者 COBOL，再不行就用C 还有 BASIC，实在不行再用 C++。

FØRTRAN

用 FORTRAN 写所有的代码。如果老板问你为啥，你可以回答说有很多它非常有用的库，你用了可以节约时间。不过，用 FORTRAN 写出可维护代码的概率是0，所以，要达到不可维护代码编程指南里的要求就容易多了。

用 ASM

把所有的通用工具函数都转成汇编程序。

用 QBASIC

所有重要的库函数都要用 QBASIC 写，然后再写个汇编的封包程序来处理 large 到 medium 的内存模型映射。

内联汇编

在你的代码里混杂一些内联的汇编程序，这样很好玩。这年头几乎没人懂汇编程序了。只要放几行汇编代码就能让维护代码的程序员望而却步。

宏汇编调用C

如果你有个汇编模块被C调用，那就尽可能经常从汇编模块再去调用C，即使只是出于微不足道的用途，另外要充分利用 goto, bcc 和其他炫目的汇编秘籍。

与他人共事之道

老板才是真行家

如果你的老板认为他20年的 FORTRAN 编程经验对于现代软件开发具有很高的指导价值，你务必严格采纳他的所有建议。投桃报李，你的老板也会信任你。这会对你的职业发展有利。你还会从他那里学到很多搞乱程序代码的新方法。

颠覆技术支持

确保代码中到处是bug的有效方法是永远不要让维护代码的程序员知道它们。这需要颠覆技术支持工作。永远不接电话。使用自动语音答复“感谢拨打技术支持热线。需要人工服务请按1，或在嘀声后留言。”，请求帮助的电子邮件必须忽略，不要给它分配服务追踪号。对任何问题的标准答复是“我估计你的账户被锁定了，有权限帮你恢复的人现在不在。”

沉默是金

永远不要对下一个危机保持警觉。如果你预见到某个问题可能会在一个固定时间爆发，摧毁西半球的全部生命，不要公开讨论它。不要告诉朋友、同事或其他你认识的有本事的人。在任何情况下都不要发表任何可能暗示到这种新的威胁的内容。只发送一篇正常优先级的、语焉不详的备忘录给管理层，保护自己免遭秋后算账。如果可能的话，把这篇稀里糊涂的信息作为另外一个更紧急的业务问题的附件。这样就可以心安理得地休息了，你知道将来你被强制提前退休之后一段时间，他们又会求着你回来，并给你对数级增长的时薪！

每月一书俱乐部

加入一个计算机每月一书俱乐部。选择那些看上去忙着写书不可能有时间真的去写代码的作者。去书店里找一些有很多图表但是没有代码例子的书。浏览一下这些书，从中学会一些迂腐拗口的术语，用它们就能唬住那些自以为是的维护代码的程序员。你的代码肯定会给他留下深刻印象。如果人们连你写的术语都理解不了，他们一定会认为你非常聪明，你的算法非常深奥。不要在你的算法说明里作任何朴素的类比。

自立门户

你一直想写系统级的代码。现在机会来了。忽略标准库， 编写你自己的标准，这将会是你简历中的一个亮点。

推出你自己的 BNF 范式

总是用你自创的、独一无二的、无文档的BNF范式记录你的命令语法。永远不要提供一套带注解的例子（合法命令和非法命令之类）来解释你的语法体系。那样会显得完全缺乏学术严谨性。确保没有明显的方式来区分终结符和中间符号。永远不要用字体、颜色、大小写和其他任何视觉提示帮助读者分辨它们。在你的 BNF 范式用和命令语言本身完全一样的标点符号，这样读者就永远无法分清一段 (...), [...], {...} 或 "..." 到底是你在命令行里真正输入的，还是想提示在你的BNF 范式里哪个语法元素是必需的、可重复的、或可选的。不管怎么样，如果他们太笨，搞不清你的BNF 范式的变化，就没资格使用你的程序。

推出你自己的内存分配

地球人儿都知道，调试动态存储是复杂和费时的。与其逐个类去确认它没有内存溢出，还不如自创一套存储分配机制呢。其实它无非是从一大片内存中 malloc 一块空间而已。用不着释放内存，让用户定期重启动系统，这样不就清除了堆么。重启之后系统需要追踪的就那么一点东西，比起解决所有的内存泄露简单得不知道到哪里去了！而且，只要用户记得定期重启系统，他们也永远不会遇到堆空间不足的问题。一旦系统被部署，你很难想象他们还能改变这个策略。

其他杂七杂八的招

如果你给某人一段程序，你会让他困惑一天；如果你教他们如何编程，你会让他困惑一辈子。 -- Anonymous

1. **不要重编译**

   让我们从一条可能是有史以来最友好的技巧开始：把代码编译成可执行文件。如果它能用，就在源代码里做一两个微小的改动 -- 每个模块都照此办理。**但是不要费劲巴拉地再编译一次了。** 你可以留着等以后有空而且需要调试的时候再说。多年以后，等可怜的维护代码的程序员更改了代码之后发现出错了，他会有一种错觉，觉得这些肯定是他自己最近修改的。这样你就能让他毫无头绪地忙碌很长时间。

2. **挫败调试工具**

   对于试图用行调试工具追踪来看懂你的代码的人，简单的一招就能让他狼狈不堪，那就是把每一行代码都写得很长。特别要把 then 语句 和 if 语句放在同一行里。他们无法设置断点。他们也无法分清在看的分支是哪个 if 里的。

3. **公制和美制**

   在工程方面有两种编码方式。一种是把所有输入都转换为公制（米制）计量单位，然后在输出的时候自己换算回各种民用计量单位。另一种是从头到尾都保持各种计量单位混合在一起。总是选择第二种方式，这就是美国之道！

4. **持续改进**

   要持续不懈地改进。要常常对你的代码做出“改进”，并强迫用户经常升级 -- 毕竟没人愿意用一个过时的版本嘛。即便他们觉得他们对现有的程序满意了，想想看，如果他们看到你又“完善“了它，他们会多么开心啊！不要告诉任何人版本之间的差别，除非你被逼无奈 -- 毕竟，为什么要告诉他们本来永远也不会注意到的一些bug呢?

5. **”关于“**

   ”关于“一栏应该只包含程序名、程序员姓名和一份用法律用语写的版权声明。理想情况下，它还应该链接到几 MB 的代码，产生有趣的动画效果。但是，里边永远不要包含程序用途的描述、它的版本号、或最新代码修改日期、或获取更新的网站地

址、或作者的email地址等。这样，所有的用户很快就会运行在不同的版本上，在安装N+1版之前就试图安装N+2版。

6. **变更**

   在两个版本之间，你能做的变更自然是多多益善。你不会希望用户年复一年地面对同一套老的接口或用户界面，这样会很无聊。最后，如果你能在用户不注意的情况下做出这些变更，那就更好了 -- 这会让他们保持警惕，戒骄戒躁。

7. **无需技能**

   写无法维护代码不需要多高的技能。喊破嗓子不如甩开膀子，不管三七二十一开始写代码就行了。记住，管理层还在按代码行数考核生产率，即使以后这些代码里的大部分都得删掉。

8. **只带一把锤子**

   一招鲜吃遍天，轻装前进。如果你手头只有一把锤子，那么所有的问题都是钉子。

9. **规范体系**

   有可能的话，忽略当前你的项目所用语言和环境中被普罗大众所接受的编程规范。比如，编写基于MFC 的应用时，就坚持使用STL 编码风格。

10. **翻转通常的 True False 惯例**

    把常用的 true 和 false 的定义反过来用。这一招听起来平淡无奇，但是往往收获奇效。你可以先藏好下面的定义：

    ```
    #define TRUE 0
    #define FALSE 1
    ```

    把这个定义深深地藏在代码中某个没人会再去看的文件里不易被发现的地方，然后让程序做下面这样的比较

    ```
    if ( var == TRUE )
    if ( var != FALSE )
    ```

某些人肯定会迫不及待地跳出来“修正”这种明显的冗余，并且在其他地方照着常规去使用变量var：

```
if ( var )
```

还有一招是为 `TRUE` 和 `FALSE`赋予相同的值，虽然大部分人可能会看穿这种骗局。给它们分别赋值 1 和 2 或者 -1 和 0 是让他们瞎忙乎的方式里更精巧的，而且这样做看起来也不失对他们的尊重。你在Java 里也可以用这一招，定义一个叫 `TRUE` 的静态常量。在这种情况下，其他程序员更有可能怀疑你干的不是好事，因为Java里已经有了内建的标识符 `true`。

11. **第三方库**

    在你的项目里引入功能强大的第三方库，然后不要用它们。潜规则就是这样，虽然你对这些好的工具仍然一无所知，却还是可以在你简历的“其他工具”一节中写上这些没用过的库。

12. **不要用库**

    假装不知道有些库已经直接在你的开发工具中引入了。如果你用VC++编程，忽略MFC 或 STL 的存在，手工编写所有字符串和数组的实现；这样有助于保持你的指针技术，并自动阻止任何扩展代码功能的企图。

13. **创建一套Build顺序**

    把这套顺序规则做得非常晦涩，让维护者根本无法编译任何他的修改代码。秘密保留 SmartJ ， 它会让 `make`脚本形同废物。类似地，偷偷地定义一个 `javac` 类，让它和编译程序同名。说到大招，那就是编写和维护一个定制的小程序，在程序里找到需要编译的文件，然后通过直接调用 `sun.tools.javac.Main` 编译类来进行编译。

14. **Make 的更多玩法**

    用一个 makefile-generated-batch-file 批处理文件从多个目录复制源文件，文件之间的覆盖规则在文档中是没有的。这样，无需任何炫酷的源代码控制系统，就能实现代码分支，并阻止你的后继者弄清哪个版本的 DoUsefulWork() 才是他需要修改的那个。

15. **搜集编码规范**

尽可能搜集所有关于编写可维护代码的建议，例如 SquareBox 的建议 ，然后明目张胆地违反它们。

16. **规避公司的编码规则**

某些公司有严格的规定，不允许使用数字标识符，你必须使用预先命名的常量。要挫败这种规定背后的意图太容易了。比如，一位聪明的 C++ 程序员是这么写的：

```
#define K_ONE 1
#define K_TWO 2
#define K_THOUSAND 999
```

17. **编译器警告**

一定要保留一些编译器警告。在 make 里使用“-”前缀强制执行，忽视任何编译器报告的错误。这样，即使维护代码的程序员不小心在你的源代码里造成了一个语法错误，make 工具还是会重新把整个包build 一遍，甚至可能会成功！而任何程序员要是手工编译你的代码，看到屏幕上冒出一堆其实无关紧要的警告，他们肯定会觉得是自己搞坏了代码。同样，他们一定会感谢你让他们有找错的机会。学有余力的同学可以做点手脚让编译器在打开编译错误诊断工具时就没法编译你的程序。当然了，编译器也许能做一些脚本边界检查，但是真正的程序员是不用这些特性的，所以你也不该用。既然你用自己的宝贵时间就能找到这些精巧的bug，何必还多此一举让编译器来检查错误呢?

18. **把 bug 修复和升级混在一起**

永远不要推出什么“bug 修复"版本。一定要把 bug 修复和数据库结构变更、复杂的用户界面修改，还有管理界面重写等混在一起。那样的话，升级就变成一件非常困难的事情，人们会慢慢习惯 bug 的存在并开始称他们为特性。那些真心希望改变这些”特性“的人们就会有动力升级到新版本。这样从长期来说可以节省你的维护工作量，并从你的客户那里获得更多收入。

19. **在你的产品发布每个新版本的时候都改变文件结构**

没错，你的客户会要求向上兼容，那就去做吧。不过一定要确保向下是不兼容的。这样可以阻止客户从新版本回退，再配合一套合理的 bug 修复规则（见上一条），就可以确保每次新版本发布后，客户都会留在新版本。学有余力的话，还可以想办法

让旧版本压根无法识别新版本产生的文件。那样的话，老版本系统不但无法读取新文件，甚至会否认这些文件是自己的应用系统产生的！温馨提示：PC 上的 Word 文字处理软件就典型地精于此道。

20. **抵消 Bug**

不用费劲去代码里找 bug 的根源。只要在更高级的例程里加入一些抵销它的代码就行了。这是一种很棒的智力测验，类似于玩3D棋，而且能让将来的代码维护者忙乎很长时间都想不明白问题到底出在哪里：是产生数据的低层例程，还是莫名其妙改了一堆东西的高层代码。这一招对天生需要多回合执行的编译器也很好用。你可以在较早的回合完全避免修复问题，让较晚的回合变得更加复杂。如果运气好，你永远都不用和编译器前端打交道。学有余力的话，在后端做点手脚，一旦前端产生的是正确的数据，就让后端报错。

21. **使用旋转锁**

不要用真正的同步原语，多种多样的旋转锁更好 -- 反复休眠然后测试一个(non-volatile的) 全局变量，直到它符合你的条件为止。相比系统对象，旋转锁使用简便，”通用“性强，”灵活“多变，实为居家旅行必备。

22. **随意安插 sync 代码**

把某些系统同步原语安插到一些用不着它们的地方。本人曾经在一段不可能会有第二个线程的代码中看到一个临界区（critical section）代码。本人当时就质问写这段代码的程序员，他居然理直气壮地说这么写是为了表明这段代码是很”关键“（也是critical）的！

23. **优雅降级**

如果你的系统包含了一套 NT 设备驱动，就让应用程序负责给驱动分配 I/O 缓冲区，然后在任何交易过程中对内存中的驱动加锁，并在交易完成后释放或解锁。这样一旦应用非正常终止，I/O缓存又没有被解锁，NT服务器就会当机。但是在客户现场不太可能会有人知道怎么弄好设备驱动，所以他们就没有选择（只能请你去免费旅游了）。

24. **定制脚本语言**

在你的 C/S 应用里嵌入一个在运行时按字节编译的脚本命令语言。

25. **依赖于编译器的代码**

如果你发现在你的编译器或解释器里有个bug，一定要确保这个bug的存在对于你的代码正常工作是至关重要的。毕竟你又不会使用其他的编译器，其他任何人也不允许！

26. **一个货真价实的例子**

下面是一位大师编写的真实例子。让我们来瞻仰一下他在这样短短几行 C 函数里展示的高超技巧。

```
void* Realocate(void*buf, int os, int ns)
{
    void*temp;
    temp = malloc(os);
    memcpy((void*)temp, (void*)buf, os);
    free(buf);
    buf = malloc(ns);
    memset(buf, 0, ns);
    memcpy((void*)buf, (void*)temp, ns);
    return buf;
}
```

- 重新发明了标准库里已有的简单函数。
- *Realocate* 这个单词拼写错误。所以说，永远不要低估创造性拼写的威力。
- 无缘无故地给输入缓冲区产生一个临时的副本。
- 无缘无故地造型。 memcpy() 里有 (void*)，这样即使我们的指针已经是 (void*) 了也要再造型一次。另外这样可以传递任何东西作为参数，加10分。
- 永远不必费力去释放临时内存空间。这样会导致缓慢的内存泄露，一开始看不出来，要程序运行一段时间才行。
- 把用不着的东西也从缓冲区里拷贝出来，以防万一。这样只会在Unix上产生core dump，Windows 就不会。
- 很显然，os 和 ns 的含义分别是”old size" 和 "new size"。

- 给 buf 分配内存之后，memset 初始化它为 0。不要使用 calloc()，因为某些人会重写 ANSI 规范，这样将来保不齐 calloc() 往 buf 里填的就不是 0 了。（虽然我们复制过去的数据量和 buf 的大小是一样的，不需要初始化，不过这也无所谓啦）

27. **如何修复 "unused variable" 错误**

    如果你的编译器冒出了 "unused local variable" 警告，不要去掉那个变量。相反，要找个聪明的办法把它用起来。我最喜欢的方法是：
    ```
    i = i;
    ```

28. **大小很关键**

    差点忘了说了，函数是越大越好。跳转和 GOTO 语句越多越好。那样的话，想做任何修改都需要分析很多场景。这会让维护代码的程序员陷入千头万绪之中。如果函数真的体型庞大的话，对于维护代码的程序员就是哥斯拉怪兽了，它会在他搞清楚情况之前就残酷无情地将他们踩翻在地。

29. **一张图片顶1000句话，一个函数就是1000行**

    把每个方法体写的尽可能的长 -- 最好是你写的任何方法或函数都没有少于1000行代码的，而且里边深度嵌套，这是必须的。

30. **少个文件**

    一定要保证一个或多个关键文件是找不到的。利用includes 里边再 includes 就能做到这一点。例如，在你的 main 模块里，你写上：
    ```
        #include <stdcode.h>
    ```
    Stdcode.h 是有的。但是在 stdcode.h 里，还有个引用：
    ```
        #include "a:\\refcode.h"
    ```
    然后，refcode.h 就没地方能找到了。

31. 到处可写，无处可读

至少要把一个变量弄成这样：到处被设置，但是几乎没有哪里用到它。不幸的是，现代编译器通常会阻止你做相反的事：到处读，没处写。不过你在C 或 C++ 里还是可以这样做的。

原始博文发布于：[Roedy Green's Mindproducts](#)。