MATLAB 编程风格指南

Richard Johnson

Version 1.5，Oct. 2002

翻译：Genial @ USTC

"Language is like a cracked kettle on which we beat tuned to dance to, while all the time we long to move the stars to pity."

— **Gustave Flaubert, in *Madame Bovary***

目 录：

简介

有关 MATLAB 代码的建议通常强调的是效率，譬如说有关“不要用循环”等的建议，本指南与之不同。本指南主要考虑的是代码（格式）的正确性、清晰性与通用性。本指南的目的在于帮助写出更可能正确、易于理解、更具有共享性与更利于维护的代码。正如 Brian Kernighan 写道：“Well-written programs are better than badly-written ones－－they have fewer errors and are easier to debug and to modify－－so it is important to think about style from the beginning.”（良好的写作规范的程序比糟糕的写作规范的要好，因为他们具有较少的错误、易于调试与修改，因此，从一开始就考虑风格是很重要的）。

本指南列举的 MATLAB 代码编写的建议在软件开发小组实际工作中得到一致性的认可。本指南与 C、C＋＋与 Java 的同类手册在整体上是相似的，但是针对 MATLAB 的特征与历史进行了修正。指南中的建议是基于多种其他代码语言的指南与个人经验而来的。指南主要是针对 MATLAB 而写的，但是它对于相近的语言，如 Octave、Scilab 和 O-Matrix 等的编程也有所帮助。

指南不是戒律，其目的在于简洁易懂地帮助程序员写出好的代码。许多组织有理由偏离这个目的。

“You got to know the rules before you can break 'em. Otherwise it's no fun.”

－－Sonny Crockett in Miami Vice

MATLAB 是 MathWorks 公司的注册商标，在本指南中，缩写 TMW 指 The Mathworks 公司。本著作献给那些致力于仔细提高进步的人们。

命名规则

Pathrick Raume：“A rose by any other name confues the issue。”（？玫瑰如果以其他名字则会导致事情的混乱）

一个开发团体建立一个命名规则可能会成为一个可笑的争议性问题。本节描述了一个常用的规则。它对于单个的程序员理解命名规则特别有用。

变量

变量的名字应该能够反应他们的意义或者用途。

变量名应该以小写字母开头的大小写混合形式

譬如：linearity ，credibleThreat，qualityofLife 等。

这在 C＋＋开发团体中是普遍实用的。TMW 有时候以大写字母开头命名一个变量名，但是这种用法在其他的语言中通常作为类型或者结果的保留用法。

分割复合变量名的各个部分中间通常有两种做法，前面例子的这种（以大写字母分

割）是其中的一种，虽然可读性比较好，但是在其他语言变量名中并不很常用。另外一种考虑的方法是在变量名中采用下划线，但是下划线在 MATLAB 的 Tex 解释程序中会将其翻译为下标转换符。

应用范围比较大的变量应该具有有意义的变量名，小范围应用的变量应该用短的变量名

实际上，大多数变量都应该具有有意义的变量名，短变量名通常作为结构申明时候必须阐明意义的情况下的变量的保留用法。当然，作为“草稿变量”的的临时存储空间或者索引可以用短名字。程序员在读到这样的变量的时候，可以假定这个变量的值在没有几行之后的代码中就不会再用到。通常的“草稿变量”整数的时候用 i，j，k，m，n（不过我认为最好少用 i，j，因为他们作为 MATLAB 中的永久性常量表示虚数单位的。译者注），双精度数的时候常用 x，y 和 z。

前缀 *n* 应该用在作为数值对象的申明的时候

这一符号来自于数学，在数学中这被作为标明数值对象的建立规则。

例如： nFiles，nSegments

MATLAB 一个附加的特别之处在于用 m 来表明行数（来源于 matrix 符号），例如变量名： mRows

应该遵循的一个有关复数（pluralization）变量的惯例

此处的复数应该是指包含多个个体的变量，相对于单数（singular）而言－－译者

一个实用的建议是将所有变量名要么为单数形式，要么为复数形式。两个变量只是最后相差一个字母 s 加以区别的情况应该避免。一个可以接收的选择是对于复数利用后缀 Array。例如：

point， pointArray　　（前者为单数，后者为复数）

只代表单个实体数据的变量可以加以后缀 No 或者是前缀 i

符号 No 来自于数学在表明实体数据的时候的建立规则。例如：

tableNo，empolyeeNo

前缀 i 使得变量命令可以循环进行。例如变量名：

iTable，iEmployee

循环变量应该以 *i*、*j*、*k* 等为前缀

该符号来源于数学在表明一个循环的时候的变量建立规则。例如：

```
for   iFile = 1：nFiles
          …
end
```

注意：当应用中有复数数据的时候，应该禁用 i、j，因为他们都是作为虚数单位使用。

对于嵌套循环，循环变量应该以字母表的顺序。

对于嵌套循环，循环变量应该命名为有帮助意思的变量名。例如：

```
for   iFile = 1：nFiles
  for   jPosition = 1：nPositions
            …
  end
end
```

否定式的布尔变量命名是应该避免的

当采用否定式的布尔变量命名法时，如果采用逻辑运算取非的操作符号对变量进行链接运算的时候，将出现双重否定的情况。例如，用～isNotFound 没有采用 isFound 直观。因此避免使用 类似于 isNotFound 这样的变量名。

缩写形式，即使是通常的大写缩写，也应该于小写字母混合使用

全部使用大写字母作为基本的变量名与上面给出的命名规则相冲突。这种类型的变量只有被命名为类似于 dVD，hTML 等形式的变量名，很显然这将使得其不具有可读性。当这样的变量名与其他的相联合的时候，其可读性严重降低；下面的单词的缩写很难看出他们的本来代表的意思了。

采用： html，isUsaSpecific，checkTiffFromat（）

避免使用：hTML，isUSASpecific，checkTIFFFormat（）

避免使用一个关键字或者特殊意义的字作为变量名

当它的保留字或者内建的特殊值被重新定义的时候，MATLAB 会给出一个模糊的出错信息或者是奇怪的结果。保留字在命令关键字中列出，特殊值在文档中列出了的。

常数

命名常数（包括全局变量）应该采用大写字母，用下划线分割单词

这个规则在 C++开发团体中是非常普遍的。尽管 TMW 的代码中可能会出现一下小写字母命名常数的情况，例如：pi，这种内建常数事实上是函数。

示例： MAX_ITERATIONS，COLOR_RED

参数可以以某些通用类型名作为前缀

这样命名的常数给出了个附加信息，指明它们属于哪类以及他们代表的意义。如：

COLOR_RED，COLOR_GREEN，COLOR_BLUE

结构体

结构体的命名应该以一个大写字母开头

这与 C++实际编程规范一致的，有助于区分结构体与普通变量。

结构体的命名应该是暗示性的（implicit），并且不需要包括字段名（fieldname）

例如下面例子给出的重复是多余的。

例： 应采用 Segment.length

避免用 Segment.SegmentLength

函数

函数名应该那个说明他们的用途。

函数名应该采用小写字母

将函数名与它的 m 文件名保存为相同的是头脑清晰的做法。采用小写字母可以避免混合系统操作时候潜在的文件名问题。

示例：getname（.）， computetotalwidth（）

还有另外两种普遍使用的函数名命名规则。一些人喜欢用下划线在函数名中增加其可读性，另外一些人则根据上面提到的变量的命名规则对函数进行命名。

函数名应该是具有意义的

存在一种不好的 MATLAB 惯例，那就是采用短的函数名，这经常使得其名字含糊不清。这或许是因为 DOS 的 8 个字母的命名规则的限制造成的影响。这种考虑担忧是没有必要的了，为了增加其可读性，这种习惯也应该避免。例如：

采用： computetotalwidth（）

避免： compwid（）

但是对于那些在数学中广泛使用的缩写或者首字母缩写的情况是个例外。譬如：

max（），gcd（）等

具有这种短的函数名的函数应该在最开始的注释的地方有完整的整个单词使得其意义清楚并且支持 lookfor 命令的查询搜索。

单输出变量的函数可以根据输出参数命名

这在 TMW 的代码中也是经常采用的，譬如：

mean（.），standarderror（.）等

没有输出变量或者返回值为句柄的函数应该根据其实现的功能命名

这种规则可以增强可读性,使得很清楚函数应该(或者不应该)干什么。这就使得代码很简洁明了并且易于理解其功能。

示例： plot（.）

前缀 *get/set* 应该作为访问对象或者属性的保留前缀

这一条在 TMW 与 C++以及 Java 开发实际中经常使用。一个合理的例外是用 set 作为逻辑置位的操作。

示例： getobj（.）；setappdata（.）

前缀 *compute* 应该用在计算某些量的函数的地方

一致应用这一条规则是为了加强可读性。它给读者一条线索：这里是潜在的比较复杂的、或者比较耗时的操作。

示例： computweightedaverage（）； computespread（）；

前缀 *find* 可以用在那些具有查询功能的函数的地方

这使得读者能够理解得到一条线索：这里是一个查询方法，包含有少量计算。一致应用这条规则可以增强其可读性，是 get 的一个好的替换品。

示例： findoldestrecord（.）； findheaviestelement(.)；

前缀 *initialize* 可以用在对象或者是概念（concept）建立的地方

美语中 *initialize* 就是指的是英国英语中的 *initialise*。应该避免使用缩写形式 *init*。

示例： initializeproblemstate（）

前缀 *is* 应该用在布尔函数的命名的地方

这通常在 TMW 的代码以及 C++与 Java 代码中普遍使用。

示例： isoverpriced（）； iscomplete（）

在某些环境下，存在少量的替代它的前缀，包括 *has*， *can* 以及 *should* 等前缀。

补足型（complement）名称应该用在补足型（complement）操作的地方

下面通过一个小的归纳减少一个一个说明的复杂性。

示例： get/set，add/remove，create/destroy，start/stop，insert/delete，increment/decrement
old/new，begin/end，first/last，up/down，min/max，next/previous，open/close
show/hide，suspend/resume，等

避免无意识地覆盖（shadowing）

通常，函数的命名应该是唯一的（unique）。覆盖（shadowing）（两个或者多个函数具有相同的函数名）会增加不可预测的行为或者错误。可以通过 **which –all** 或者是 **exist** 来检查文件名重复的情况。

概要

命名多维变量与常量应该具有单位后缀

只采用单一的单位集合是一个很不错的想法，但是通常在程序的完整实现很少见的。增加单位后缀可以帮助避免必然的混淆。

示例： incidentAngleRadians

命名中应该避免缩写

利用完整的单词命名可以减少含糊，有利于使得代码自成为文档（self-documenting）。

采用： computearrivaltime（）

避免： comparr（）

特殊领域的常用语的简写或者首字母缩写形式更容易自然地被理解，因此他们应该保持缩写形式。甚至在他们第一次出现的时候的定义注释的时候都是允许的。

示例： html，cpu，cm

考虑使得名字可以拼读

在命名的时候应该至少考虑易于拼读与记忆。

所有的命名都应该以英语的形式写出

MATLAB 是以英语发布的，英语是国际研发交流中最适合的语言。

文件与程序结构

将代码结构化，不只是在文件的内部，也包括在文件之间，都能够使得程序更易于理解。有思想的（thoughtful）程序结构块分割和条理化可以增加代码的质量。

M 文件

模块化

编写一个大程序的最好的方法是将它以好的设计分化为小块（通常采用函数的方式）。这种方式通过减少为了理解代码的作用而必须阅读的代码数量使得程序的可读性、易于理解性和可测试性得到了增强。超过编辑器两屏幕的代码都应该考虑进行分割。并且设计规划很好的函数也使得它在其他的应用中可用性增强了。

确保交互过程清晰

函数通过输入输出参数以及全局变量与其他代码交互通信。使用参数几乎总是比使用全局变量清楚明了。采用结构可以避免那种一长串儿的输入输出参数的形式。

分割（partitioning）

所有的子函数和所有的函数都应该只把一件事情做好。

每个函数应该隐藏（hide）一些东西。

利用现有的函数

开发一个有正确功能的、可读的、合理灵活性的函数在一项有重大意义的任务。或许寻找一个现成的提供了要求的部分、甚至全部功能的函数应该更快也更具有正确性。

任何在多个 m 文件中出现的代码块都应该考虑用函数的形式封装起来

如果代码只在一个文件中出现，那么修改变换起来就会容易得多。“改变是不可避免的，…，除非自动售货机。”

子函数

只被另外一个函数调用的函数应该作为一个子函数写在同一个文件中。这使得代码更加利于理解与维护。

测试脚本

为每一个函数写一个测试脚本。这样可以提高初期版本的质量和改进版本的可靠性。要注意的是，任何函数如果不易于测试的话就可以不易于编写的。Boris Beizer 讲到：“一个好的反 bug 人员知道，设计测试案例比实际的测试需要更多的行动。”

输入输出

编写输入/输出模块

输出要求可以无需特别注意就可以根据变化而改变，输入的格式与内容根据变化的时候经常很混乱。找到处理输出的地方进行改善，提高其可维护性。避免将输入/输出部分的代码与计算功能的代码混淆在一起，单个函数的预处理的时候除外。各种功能混合的函数的可再用性一遍很小。

格式化输出使得其易于利用

如果输出很大可能是人工阅读，那么就让输出采用易于越多的描述性的方式。

如果输出更多的可能是通过其他软件调用而不是人，那么应该使得输出易于解析。

如果这以上两种情况都很重要，将输出表达成易于解析的格式，并编写一个格式化输出的函数用来产生一个人工可读的输出版本。

基本语句

变量与常数

变量不应该重复使用（赋予为不同意义），除非因为内存限制的需要

通过确保所有的概念都只有唯一的意义可以加强代码的可读性，以及通过消除误解的定义可以减少错误的可能。

同种类型的相近的变量可以在同一个语句中定义

不相近的变量应该不要在同一个语句中定义

通过变量分组可以增强其可读性。

示例： persistent x，y，z

global REVENUE_JANUARY，REVENUE_FEBRUARY

注意在文件开始部分的注释中为重要变量编写文档

在其他的编程语言中，在变量申明的地方为他们编写文档是一种标准话的操作。既然 MATLAB 不需要变量申明，这种信息就可以在注释中提供。

示例： % pointArray Points are in rows with coordinates in columns.

注意在语句行注释的最后为常数编写文档

这对参数有关合理性、应用和约束等附加信息。

示例： THRESHOLD = 10； %Maximum noise level found by experiment。

全局变量

应该尽量少地使用全局变量

参数传递在代码清晰性与可维护性方面都比常用全局变量要好。在某些应用 global 变量的地方可以被 ***persisitent*** 和 ***getappdata*** 所代替。

应该尽量少用全局常量

利用 m 文件或者是 mat 文件，这样就可以使得很清楚，常数在什么地方定义的，避免无意识地重复定义。如果文件的访问接口是不令人满意的，那么可以考虑采用全局常数的结构的形式。

循环语句

循环变量应该在循环开始前立即被赋值

这可以提高循环的速度，有助于防止循环没有执行所有的可能索引而产生的虚假值。

示例：

```
refult = zeros( nEntries，1)；
for index = 1：nEntries
    result（index）= foo（index）；
end
```

在循环中应该尽量少用 *break* 与 *continue*

这些结构可以与 **goto** 相比较，只有当他们可以证明用这些结构可以比他们相应的结构化部分有更好的可读性的时候，才可以使用。

在嵌套式循环的时候应该在 end 行加上注释

在长的嵌套循环的 **end** 命令行添加注释可以有助于弄明白哪些语句在那个循环体内、在此处之前已经完成了哪些功能。

条件语句

应该避免复杂的条件表示式，而采用临时逻辑变量进行替代

通过对表达式指定逻辑变量，使得程序更能够自为文档，使得程序结构更易于阅读与调试。

示例：

避免使用：

```
if（value>=lowerLimit）& (values<=upperLimit) &~ismember(value,valueArray)
    ……
    ……
end
```

而应该用如下的方式代替：

```
isValid  =  （value=lowerLimit）& (values<=upperLimit)；
isNew   = ~ismember(value,valueArray)
if ( isValid & isNew)
    ……
    ……
 end
```

在 *if else* 结构的时候，发生较频繁的事件应该放在 *if* 部分，例外情况放在 *else* 部分

这样通过将例外情况排除在常规执行路径之外可以提高程序的可读性。

示例：

```
fid = fopen（fileName）;
if（fid~ = -1）
    …
else
    …
end
```

条件表达式 *if 0* 是应该避免的，除非在对临时程序块进行注释的时候

如果确信表达式在程序正常执行的时候不会发生，首选的方法是采用编辑器的块注释。

一个 *switch* 语句应该包含 *otherwise* 条件

将 *otherwise* 情况遗漏在外是一种通常错误，这或许会导致不可预测结果。

正确示例：

```
switch（condition）
case  ABC
      处理语句;
case  DEF
      处理语句;
otherwise
     处理语句;
end
```

***switch* 变量应该通常是字符串**

字符串在这种情况下能够很有效，通常他们比采用列举值的形式意义更丰富。

小结

避免含糊代码

或许是因为莎士比亚的这句“简单是智慧的灵魂”所感染，在一些程序员之中存在这样一种倾向：将 MATLAB 代码写得很简洁，甚至朦朦胧胧。编写简练的程序是一种表现语言的特色的方式。然而，在很多情况下，清楚是核心问题。正如 TMW 的 Steve Lord 写道：“从现在开始，一个月后，如果我再看这些代码，我能否理解他们是干什么的？”

尽量避免出现 *Cool Hand Luke* 中上尉所描述的那样：“现在我们到了一个无法交流的

地方。”

这个论点在很多作者的著作中都重点强调的。***Martin Fowler*** 写道：“ 任何一个傻子都可以写出一个计算机能够理解的代码。好的程序员写出人能够理解的代码。”***Kreitzberg*** 与 ***Shneiderman*** 写道：“编程可以很有趣，密码学同样也可以很有趣，但是，他们不可以混合在一起。”

采用附加说明

MATLAB 对于操作运算有个优先级的文档，但是谁愿意记住它们的具体内容呢？如果在某些地方有任何疑问，采用附加说明使得表达清楚。特别是在扩展的逻辑表达式的时候尤其有用。

尽量在表达式中少用数字。可能会改变的数字应该用常数代替

如果一个数字它的本身没有明确的意义，采用将它命名为常数可以加强程序的可读性。并且，改变参数的定义比改变文件中所有的相应出现地方的数字要容易得多。

浮点常数应该在小数点前面写上一个阿拉伯数据

这是坚持数学习惯的语法要求，而且，0.5 比 .5 更具有可读性，因为 .5 很有可能被误认为是整数 5。

采用 THRESHOLD = 0.5；

避免使用 THRESHOLD = .5；

浮点数的比较应该要小心

二进制表达可能导致麻烦，如下面的例子所示：

```
shortSide = 3；
longSide = 5；
otherSide = 4；
longSide^2 ＝＝ （shortSide^2+otherSide^2）
ans =
     1
scaleFactor = 0.01；
（scaleFactor*longSide）^2 = = ((scaleFactor*shortSide)^2+(scaleFactor*otherSide)^2
ans =
      0
```

排版、注释与文档

排版

排版的目的是帮助读者理解代码，缩排特别有助于展示程序的机构。

应该将代码内容控制在前 80 列之内

对于一个编辑器、终端仿真器、打印机、调试器以及文件的通常列数是 80 列，因此通常几个人的程序共享的时候，大家通常将内容控制在前 80 列之内。在程序员之间传递文件的时候，避免无意识的分行可以增强程序代码的可读性。

在恰当的地方应该将行进行切分

当语句长度超过 80 列的限制的时候应该切分行。通常：

- 在一个逗号或者空格之后进行断开；
- 在一个操作符之后断开；
- 在表达式开始前的地方重新开始新的一行

示例：

```
totalSum = a +b +c +…
           d+e;
function (param1，param2，…
           param3)
 setText（['Long line split'…
            'into two parts.']）;
```

基本缩排应该是 3 或者 4 个空格

好的缩排或许是唯一的一个展现程序结构的好方法。

1 个空格是缩排太小而不能够强调出代码的逻辑分层；2 个空格的缩排在为了减少因为嵌套循环超过 80 列而切分行的断裂的时候被建议采用，而 MATLAB 通常没有太多太深的循环嵌套。大于 4 个空格的缩排使得因为行切分的机会增大而使得代码的可读性变差。4 个空格的缩排是 MATLAB 编辑器的缺省设置，在以前的一些版本，缺省缩排是 3 个空格。

应该与 MATLAB 编辑器的缩排一致

MATLAB 编辑器提供了使得代码结构清晰的缩排，并且与 C++与 Java 推荐使用的缩排方式相一致。

通常情况下，一行代码应该只包含一个可执行语句

这种方式可以提高可读性，并且允许 JIT 加速。

短的单个 *if*，*for* 或者 *while* 语句可以写在一行

这种方式更加紧凑，但是她失去了缩排格式提示的优点。

示例：

if（condition），statement；end

while（condition），statement；end

for iTest = 1：nTest，statement；end

空白空格

空白空格通过将各个单独组成部分的语句独立出来而增强了程序的可读性。

在 =，&，与 | 前后加上空格

在指定的字符前后加上空格可以增强其可视化的分割提示，明显地将语句左右两部分分开。在二值逻辑操作符前后加上空格可以使得复杂的表达式清晰。

示例：

simpleSum = firstTerm+secondTerm;

常规的操作符两边可以加上空格

这种方式是有争议的。部分人认为它可以增强其可读性。

示例：

simpleAverage = (firstTerm + secondTerm) / two;

1 : nIterations

逗号后面可以加上空格

这些空格可以增强可读性。有些程序员为了避免切分行而不采用这种方式。

示例： foo(alpha, beta, gamma)，也可以 foo(alpha,beta,gama)

分号或者同一行多条指令的逗号之后应该加上一个空格字符

空格加强可读性。

示例：if （pi>1），disp('Yes'), end

关键字后面应该加上空格

这种方式有助于区分关键字与函数。

一个块（block）内部的一个逻辑组语句应该通过一个空白行将其分隔开

在块的逻辑单元之间加入空白行可以增强代码的可读性。

块（blocks）之间应该用多行空白行分隔

一种方式是采用 3 隔空白行。采用大的间隔来与块内分隔相区别，使得在文件中，块看起来非常明显。另外一种方式是采用注释符号后面跟多个诸如*或者 － 。

通过排列成行列整齐的方式来加强可读性

代码排列成行列整齐的形式可以使得切分表达式容易阅读与理解。这种排版也有助于揭示错误。

示例：
```
weithedPopulation = (doctorWeight * nDoctors) + …
                    (layerWeight * nLawyers) + …
                    (chiefWeight * nChiefs);
```

注释

注释的目的是为代码增加信息。注释的典型应用是解释用法、提供参考信息、证明结果、阐述需要的改进等。经验表明，在写代码的同时就加上注释比后来再补充注释要好。

注释不能够改变写得很糟糕的代码效果

注释不能够弥补因为代码命名不当、没有清晰的逻辑结构等造成的缺陷。存在这样缺陷的代码应该重写。Steve MaConnell 说道：“提高代码质量，然后再增加文档以进一步使得其清晰。”

注释文字应该简洁易读

一个糟糕的或者是无用的注释反而会影响读者的正常理解。N.Schryer 提到：“如果代码与注释不一致，那么或许两者都是错误的。”一个通常更重要的是注释应该讲的是“为什么”（Why）和“怎么做”（how），而不是“是什么”（what）。

函数头部的注释应该支持利用 *help* 与 *lookfor* 查询

help 命令打印出文件开题的第一块注释行。lookfor 命令搜寻路径上的所有 m 文件的第一个注释行，尽量在这一行中包含可能的搜索关键字。

函数头部的注释应该讨论对输入参数的特殊要求等

使用者通常需要知道输入是否有特殊的单位要求或者矩阵类型要求

示例：
```
% ejectionFraction must be between 0 and 1, not a percentage.
% elepsedTimeSceconds must be one dimensional.
```

函数头的注释应该描述其任何副作用（side effects）

副作用是指函数的行为而不是指输出参数的指定。一个常见的例子是图的产生，在头说明中描述其副作用以便他们可以在用 *help* 指令的打印输出的时候是可见的。

通常情况下，函数头注释的最后一句应该是重申函数语句行

这可以让用户在一眼扫过 *help* 指令的打印输出时可以发现函数的输入输出参数用法。

在函数头注释中讲函数名用大写的形式表达是一个有争议性的规则

在 MathWorks 的开发中采用的是这样的规则，这样可以突出函数名。很多其他作者不遵循这条规则，因为其缺点是通常在代码中函数名是小写字母。

避免在函数头说明的 *help* 打印输出中的混乱

在函数文件开头的附近通常包括版权以及修改日期等信息。在文件头说明和这些信息说明之间应该加入一个空白行，以避免在用 *help* 指令的时候显示出来。

所有的注释语句应该用英语写作

在国际环境中，英语是最提倡使用的语言。

文档

文档规范化

作为有用的文档应该包含一个对如下内容的可读性的描述：代码打算干什么（要求），它是如何工作的（设计），它依赖于什其他什么函数以及怎么被其他代码调用（接口），以及它是如何测试的等。对于额外的考虑，文档可以包含解决方案的选择性的讨论以及扩展与维护的建议。

Dick Brandon 讲到：“Documentation is like sex；when it's good，it's very，very good，and when it's bad，it's better than nothing.”

首先考虑书写文档

一些程序员相信的方法是：“代码第一，回答问题是以后的事情。”而通过经验，我们绝大多数人知道先开发设计然后再实现可以导致更加满意的结果。

如果将测试与文档留在最后，那么开发项目几乎不能够按期完成的。首先书写文档可以确保其按时完成甚至可能减少开发时间。

修改

一个专业的对代码修改进行管理和写文档的方法是采用源程序控制工具。对于很简单的工程，在函数文件的注释中加入修改历史比什么都不做要好。

示例：24 November 1971，D.B.Cooper，exit conditions modified.

参考文献

The Practice of Programming，Brian Kemighan and Rob Pike

The Pragmatic Programmer，Andrew Hunt，David Thomas and Ward Cunningham

Java Programming Style Guidelines，Geotechnical Software Services

Code Complete，Steve McConnel—Microsoft Press

C＋＋Coding Standard，Todd Hoff

由于自己的文字组织能力有限，很多地方都采用了直译的形式，另外有些专业的术语自己也不确定到底该如何翻译，因此错误之处在所难免，欢迎批评指正。

Genial @ USTC

2004-4-25