事实上，这个世界并没有几份 GNU m4 教程。

这个文档系列是我第一次认真学习 GNU m4 并进行了一些实践之后的一些总结。由于我在撰写此文的过程中充满着像 m4 展开一个又一个宏一般的耐心，因此这篇文章会比较长。在这个信息碎片化的时代，似乎没有很多人愿意去看很长的文章，大家更喜欢干货。为了节省大家的时间，必须声明，这个文档系列没有干货，它是写给我自己或者那些像我自己的人看的。

什么是宏

书名是『宏』，它被作者展开为这本书的全部内容。药瓶上的标签是『宏』，将药片从瓶中倾倒出来，就是这个宏的展开结果。被用的最多的『宏』，应该是 Internet 的超级链接。每当你点击一个超级链接，就相当于将这个宏展开为网页中的内容。生活中，类似的例子还有很多，只要你给某种具体的事物贴上了一个标签，那么这个标签就相当于宏。

人类非常喜欢给事物贴标签，尽管无论他们贴与不贴，那些事物本身依然是存在的。在编程中，如果你想给一段代码贴标签，最简单最直接的办法就是使用宏。那些还在用汇编语言编程的人，他们是离不开宏的，因为汇编语言本身就是将一大堆标签贴在了更大的一堆机器代码上。如果所用的编程语言不提供宏功能，可以用这种编程语言为一段代码制作一个标签——函数，不过这种标签就不是宏了，而且要付出一些性能上的代价，因为标签的展开过程被推迟到程序的运行过程。

C 语言自诞生后，只用了 5 年就让汇编语言归隐山林了，这可能要归功于 Unix 的成功以及 Dennis Ritchie 的忽悠。Steve Johnson——yacc, lint, spell 以及 PCC（Portable C Compiler）的作者说：『Dennis Ritchie 告诉所有人，C 函数的调用开销真的很小很小。于是人人都开始编写小函数，搞模块化。然而几年后，我们发现在 PDP-11 中函数的调用开销依然非常大，而 VAX 机器上的代码往往在 CALL 指令上花费掉 50% 的运行时间。Dennis 对我们撒了谎！但为时已晚，我们已经欲罢不能……』

现代的编程语言，几乎都赞同用函数来取代宏。拥护者们往往会给出一些冠冕堂皇的理由是，诸如不必额外实现一个宏处理器，函数比宏更安全并且更容易调试。事实上，他们的理由仅仅是迎合现实而已。如果将这些人扔进时空裂缝让他们穿越到 Ken Thompson 编写 Unix 系统的时代，让他们也在一台废弃的 PDP-7 型号的计算机上写程序。在这种内存只有 8KB 的计算机上，那些冠冕堂皇的理由近乎与科幻小说等价。函数之所以能够取代宏，仅仅是因为 CPU 的计算速度比过去更快了，内存比以前更大了，牺牲一些程序性能，让编程工作更容易一些，这样比较合算而已。编程语言的性能与机器的性能似乎总是成反比的。

宏被很多人主观的弃用了，得益于现代编程语言的表达能力，他们似乎几乎不需要用宏，于是他们作出结论：宏过时了。事实上，宏会永远居于众编程语言之上的，因为前者总是能够生成后者。编程专家总是会告诉我们，要慎用宏。胆子小的程序猿看到宏就躲得远远的，以至于他们总觉得那些使用宏的代码是糟糕的，是不安全的。事实上，在编程中，若能恰如其分的使用宏，可以让代码更加简洁易读，特别是对 C 语言这种表现力不足的语言。

例如下面 C 代码中的宏：

```
#define DEF_PAIR_OF(dtype) \typedef struct pair_of_##dtype { \
        dtype first; \
        dtype second; \
} pair_of_##dtype##_t

DEF_PAIR_OF(int);
DEF_PAIR_OF(double);
DEF_PAIR_OF(MyStruct);
```

是不是有点 C++ 模板的意味？像 C 标准库提供的 `qsort` 函数所接受的回调函数，也可以用类似的方法半自动生成。有关 C 语言宏的基本规则与技巧，可参考『宏定义的黑魔法 - 宏菜鸟起飞手册』。即使是表达能力很强的现代编程语言，在处理复杂问题上，也无法避免代码自身的频繁重复，妥善的使用宏总是可以消除这种重复，甚至可以创造一些 DSL（领域专用语言）。

在代码中适当的运用宏，创造优雅易读的代码，这样或许更能体现编程是一种艺术。虽然有些编程语言未提供宏功能，但是我们总是会有 GNU m4 这种通用的宏处理器可用。

GNU m4 简介

m4 是一种宏处理器，它扫描用户输入的文本并将其输出，期间如果遇到宏就将其展开后输出。宏有两种，一种是内建的，另一种是用户定义的，它们能接受任意数量的参数。**除了做展开宏的工作之外，m4 内建的宏能够加载文件，执行 Shell 命令，做整数运算，操纵文本，形成递归等等。m4 可用作编译器的前端，或者单纯作为宏处理器来用。**

所有的 Unix 系统都会提供 m4 宏处理器，因为它是 POSIX 标准的一部分。通常只有很少一部分人知道它的存在，这些发现了 m4 的人往往会在某个方面成为专家。这不是我说的，这是 m4 手册说的。

有些人对 m4 非常着迷，他们先是用 m4 解决一些简单的问题，然后解决了一个比一个更大的问题，直至掌握如何编写一个复杂的 m4 宏集。若痴迷于此，往往会对一些简单的问题写出复杂的 m4 脚本，然后耗费很多时间去调试，反而不如直接手动解决问题更有效。所以，对于程序猿中的强迫症患者，要对 m4 有所警惕，它可能会危及你的健康。这也不是我说的，是 m4 手册说的。

m4 基本工作过程

上文提到『m4 是一种宏处理器，它扫描用户输入的文本并将其输出，期间如果遇到宏就将其展开后输出』，其实更正式的说，应该是：m4 从文本输入流中获取文本并将其发送到文本输出流，期间如果遇到宏就将其展开后发送到文本输出流。

在 Brian Kernighan 与 Dennis Ritchie 合著的《C Programming Language》中将**流**（Stream）定义为『**与磁盘或其它外围设备关联的数据的源或目的地**』。基于这个定义，m4 的**输入流**就是**与磁盘或其它外围设备关联的数据的源**，其**输出流**就是**与磁盘或其它外围设备关联的数据的源或目的地**，只不过 m4 希望它的输入流与输出流的内容是文本。如果你不那么较真，可以将**流**理解为文件，对于 m4 而言，就是文本文件，但是下文会坚持使用流的概念。

> m4 使用流的概念并非巧合，如果说巧合，也只是因为它的作者恰好也是 Brian Kernighan 与 Dennis Ritchie。

m4 是如何从输入流中获取文本并将其发送到输出流的？肯定不是简单的读取文本就了事，因为 m4 有一个任务是『遇到宏就将其展开』。这意味着 m4 在从输入流中读取文本的过程中至少需要检测所读取的某段文本是不是宏。也就是说，从 m4 的角度来看，它首先要将输入流所提供的文本分为两类：**宏**与**非宏**。如果 m4 读取的是一段文本是**非宏**，它基本上会将它们直接发送到输出流。之所以说是『基本上』，是因为非宏的文本会被进一步分类处理，其中细节后文会讲。如果 m4 读取的文本片段是**宏**，m4 就会将它展开，然后将展开结果发送到输出流。

m4 的工作过程具有一定程度的即时性，它不需要将输入流中全部信息都读取出来，然后再进行处理，而是扮演了一种过滤器的角色。从用户的角度来看，文本流入 m4，然后又流出。

从图灵的角度来看 m4，输入流与输出流可以衔接起来构成一条无限延伸的纸带，m4 是这条纸带的读写头，所以 m4 是一种图灵机。事实上，m4 的确是一种图灵机。因此 m4 的计算能力与任何一种编程语言等同，区别只体现在编程效率以及所编写的程序的运行效率方面。感觉基于 m4 来讲解计算机原理还是挺不错的。

m4 的工作空间

m4 既然是图灵机，它至少需要有一个『状态寄存器』，否则它无法判断当前从输入流中读取的文本是宏还是非宏。为了提高文本处理效率，还应该有一个缓存空间，使得 m4 在这一空间中高效工作。现代的 CPU，没有缓存的应该很罕见。

m4 缓存的容量为 512KB。当它满了的时候，m4 会将自动将其中的内容妥善的保存到一份临时文件中备用。所以，只要你的磁盘或其它外围设备的容量足够，就不要担心 m4 无法处理大文件。

> 注意，m4 缓存，这个概念是我瞎杜撰的。GNU m4 官方文档没这个概念，官方的概念是**转移**（Diversion）。

类似 CPU 的多级缓存，m4 的缓存空间也是划分了级别的。符合 POSIX 标准的 m4，可将缓存空间划分为 10 种级别，编号依次为 `0, 1, 2, ..., 9`。GNU m4 对缓存空间的级别数量不作限制。

m4 默认在 0 号缓存中工作，它在这个缓存对文本进行处理，然后将其发送到输出流。使用 m4 内建的宏 `divert`，可以从当前缓存切换到其他缓存。例如：

```
divert(3)
```

就从当前的缓存切换到 3 号缓存了，然后 m4 就在 3 号缓存中对输入流中的文本进行处理。如果不继续使用 `divert` 进行缓存切换，m4 会一直在 3 号缓存中工作，直到输入流终结。最后，m4 会将各个缓存中的文本汇总到 0 号缓存中。

缓存的汇总过程是按照缓存级别进行的。m4 会根据缓存级别的编号的增序进行汇总。例如，它总是先将 1 号缓存的内容汇总到 0 号缓存中，然后将 2 号缓存的内容汇总到 0 号缓存中，以此类推，最后将 0 号缓存中的内容依序发送到输出流中。

划分了级别的缓存，像是一道一道分水岭，使得文本流像河流一样拥有支流，不同的支流最终又汇集到一起，奔流到海……是不是有些气势恢宏的感觉，然而你也应该考虑到这样的现实：百川东到海，何时复西归？也就是说，文本流经 m4 的过程也像河流入海一样的不可逆。这是宏最大的弱点。在程序中滥用宏，形同过度开采水资源。

软件领域有一门学科，叫**逆向工程**，研究如何借助反汇编技术重现某个程序的原有逻辑。具体技术我不是很了解，但是幸好有这门学科，否则我的显卡很难在新版本的 Linux 内核上工作。因为 Nvidia 官方的 Linux 驱动自某个版本之后就宣布不再支持我这种型号的显卡了，而 Nvidia 官方驱动已经被大神实施逆向工程产生了 Nouveau 驱动，而后者又被集成到了 Linux 内核中。

似乎跑题了，我想表达的是，逆向工程固然能够在一定程度上复原某个程序的源码，但它却永远无法基于宏的展开结果重现宏的定义，只有宏的作者才知道当初究竟发生了什么。

这时，你应该有一个问题。如果你真的想学习 m4，那就必须要有这个问题——m4 为什么要对缓存划分级别？回顾一下上文，各个缓存的汇总过程是遵循特定次序的。有了这种分级的缓存汇总机制，你就有能力借助缓存来控制文本的支流，决定哪条支流先汇入 0 号缓存。你可以说这样你有机会扮演大禹，但是我觉得这更像铁路调度员所做的事。对于铁路调度员而言，文本流是他要调度的一组列车。

暗黑缓存

更有趣的是，m4 也提供了暗黑缓存，它的编号是 `-1`。GNU m4 对暗黑缓存也不限制数量，只要它们的编号是负数就可以。

暗黑缓存，似乎有点恐怖，实际上你可以将它们理解为地下河。也就是流过暗黑缓存的文本，m4 会将它们汇总到 0 号缓存，汇总过程按照暗黑缓存编号的递减次序进行的，但是 m4 不会将暗黑缓存汇总的内容发送到输出流。这没什么不好理解的，现实中没有什么东西是**负数**的。

在 m4 的应用中，暗黑缓存的主要作用就是作为宏定义的空间。如果在 0 号缓存定义一个宏，例如：

```
divert(0)
define(say_hello_world, Hello World!)
```

定义了一个名为 `say_hello_world` 的 m4 宏。宏定义语句『展开』为一个长度为 0 的字符串，然后发送到输出流。长度为 0 的字符串，就是空文本，即使它被发送到输出流，对输出流不会产生任何影响，但是 `say_hello_world` 宏之前，也就是 `divert(0)` 之后存在一个**换行符**，m4 会将这个换行符发送到输出流。除非你原本就希望输出流中需要这个换行符，否则你就在输出流中引入了一个额外的换行符，通常情况下，它不是你想要的结果。为了更好的说明这一点，可以看下面的示例：

```
divert(0)
define(say_hello_world, Hello World!)
say_hello_world
```

这个示例就是在上述代码中又增加了一行文本，它表示调用了上一行所定义的 `say_hello_world` 宏。假设示例代码保存在 hello.m4 文件中，然后执行以下命令：

```
$ m4 hello.m4
```

此时，hello.m4 就是 m4 的输入流。m4 从输入流中读取文本，处理文本，然后将处理结果发送到输出流。此时，输出流是系统的标准输出设备（stdout），也就是当前的终端屏幕。

执行上述命令后，我们期望的结果通常是：

```
$ m4 hello.m4
say_hello_world
```

然而，m4 输出的却是：

```
$ m4 hello.m4


Hello World!
```

`Hello World!` 前面出现了两处空行，一处是 `divert` 语句后面的换行符导致的，另处是 `say_hello_world` 宏定义语句后面的换行符导致的。

如果将 `say_hello_world` 宏定义语句放在暗黑缓存中，可以解决一半问题。例如：

```
 divert(-1)
define(say_hello_world, Hello World!)
divert(0)
say_hello_world
```

再次执行 m4 命令，可得：

```
$ m4 hello.m4

Hello World!
```

现在 `Hello World!` 前面只有 1 处空行了，它是 `divert(0)` 后面的换行符导致的。要消除它，有两种方法。第一种方法就是 `divert(0)` 后面不换行，例如：

```
 divert(-1)
define(say_hello_world, Hello World!)
divert(0)say_hello_world
```

另一种方法是使用 m4 内建的 `dnl` 宏，它会从将它被调用的位置到后面的第一个换行符之间的文本（包括换行符本身）一并删除，例如：

```
 divert(-1)
define(say_hello_world, Hello World!)
divert(0)dnl
say_hello_world
```

这两种方法输出的结果是相同的。为了让文本具有更好的可读性，通常用 `dnl` 来做这样的事。

挑战

(1) 对于以下 m4 代码

```
 divert(-1)
define(say, )
define(hello, HELLO)
define(world, WORLD!)
divert(0)dnl
say hello world
```

推测一下 m4 的处理结果，然后执行 m4 命令检验所做的推测是否正确。

(2) 对于以下 m4 代码

```
 divert(2)
define(say, )
define(hello, HELLO)
divert(1)
define(world, WORLD!)
divert(0)dnl
say hello world
```

推测一下 m4 的处理结果，然后执行 m4 命令检验所做的推测是否正确。

宏

自定义一个 m4 宏所用的基本格式如下：

```
define(宏名, 宏体)
```

上一节，我们定义的一个很简单的 `say_hello_world` 宏：

```
define(say_hello_world, Hello World!)
```

`say_hello_world` 是宏名，`Hello World` 是宏体。如果在宏定义之后的文本中出现了 `say_hello_world`，例如：

```
define(say_hello_world, Hello World!)
blab blab ... say_hello_world
```

假设上述文本均处于非负号缓存，那么当 m4 从输入流中读取到 `say_hello_world` 时，它能够检测出该文本片段是一个被定义了的宏，于是它就将这个宏展开为 `Hello World`，并使用这个展开结果替换文本片段 `say_hello_world`，所以，上述文本经过 m4 处理后发送到输出流，就变成：

```
blab blab ... Hello World!
```

上述输出结果中的空行，应该没什么玄机可言了，只是需要注意：宏定义语句本身也会被 m4 展开，因为 `define` 本身就是一个宏，只不过它的展开结果是一个空的字符串。

有参数的宏

宏可以有参数。遵循 POSIX 标准的 m4，允许一个宏最多有 9 个参数（似乎 Shell 脚本里的函数也最多支持 9 个参数），在**宏体**中可使用用 `$1, ..., $9` 来引用它们。GNU 的 m4 不限制宏的参数数量。

对于下面这段 C 语言的宏定义与调用：

```
#define DEF_PAIR_OF(dtype) \typedef struct pair_of_##dtype { \
        dtype first; \
        dtype second; \
} pair_of_##dtype##_t

DEF_PAIR_OF(int);
DEF_PAIR_OF(double);
DEF_PAIR_OF(MyStruct);
```

用 m4 的有参数的宏可给出等价表示：

```
 divert(-1)
define(DEF_PAIR_OF,
`typedef struct pair_of_$1 {
        $1 first;
        $1 second;
} pair_of_$1')
divert(0)dnl

DEF_PAIR_OF(int);
DEF_PAIR_OF(double);
DEF_PAIR_OF(MyStruct);
```

它们能够展开为同样的 C 代码（C 语言宏由 C 预处理器展开，m4 宏由 m4 展开）：

```
typedef struct pair_of_int {
        int first;
        int second;
} pair_of_int;
typedef struct pair_of_double {
        double first;
        double second;
} pair_of_double;
typedef struct pair_of_MyStruct {
        MyStruct first;
        MyStruct second;
} pair_of_MyStruct;
```

注意，C 宏与 m4 宏的调用有点区别。**在 C 中，调用一个宏，宏名与其后的 `(` 可以有空格，而 m4 宏的调用不允许这样。**

m4 版本的 `DEF_PAIR_OF` 宏的宏体为：

```
`typedef struct pair_of_$1 {
        $1 first;
        $1 second;
} pair_of_$1'
```

这个宏体是一个带引号的字符串，形如：

```
`... ... ...'
```

注意左引号与右引号对应的符号。在大部分键盘上，左引号与 `~` 符号同键，右引号与 `"` 同键，它们是单引号。不要对引号掉以倾心，它在 m4 中的重要地位仅位列宏之下，如果没有它，宏的世界会异常的混乱。后面，我会在单独给引号的基本用法留出一节专门阐述。在此只需将引号理解为一段文本的封装。

事实上，对于 m4 版本的 `DEF_PAIR_OF` 宏体，不用引号也不会出问题（可以去掉引号试一下）。但是，在复杂一些的宏体内，可能会出现 `,` 符号，如果这样的宏体不用引号囊括起来，那么 `,` 会被 m4 误认为宏参数的分隔符。所以，一定要记住：**`,` 会被 m4 捕获为宏参数分隔符，而引号可使之逃逸。**

小实践：reStructuredText 插图标记的简化

reStructuredText 是一种轻量级的文本标记语言，与 Markdown 属于同类，一般用于记笔记，然后以网页的形式发布。之所以要用轻量级文本标记，是因为直接手写 HTML 太繁琐了。我在我的机器上搭建的 Nikola 静态网站，默认用的就是 reStructuredText，我用它来整理我的一些笔记。

在使用 reSturcturedText 写文档时，我觉得它的插图标记过于繁琐。我常用的插图标记如下：

```
.. figure:: 图片文件路径
    :align: center
    :width: 宽度值
```

上述标记文本块前后必须要留出空行，否则 reStructuredText 的解析器就会抱怨。`align` 与 `width` 前面的缩进也是必须的，否则 reStructuredText 的解析器就会抱怨……

为了简化这个标记，我用 m4 定义了一个宏：

```
divert(-1)
define(place_figure, `
.. figure:: $1
    :align: center
    :width: $2')
divert(0)dnl
```

然后我就可以愉快的像下面这样在 reStructuredText 文本中插入一幅图片了。

```
place_figure(`/images/amenta-needles/0001.png', 480)
```

用这种办法可以简化许多繁琐的文本标记，甚至可以实现 reStructuredText 不具备的功能，例如参考文献的管理。如果你不打算研究如何改造 reStructuredText 解析器来满足自己的需求，在这种前提下，用 m4 简单的 hack 一下，使得 reStructuredText 的易用性显著增强，这就是宏语言最大的用途。

不妨将宏语言视为生活中的方便袋。

宏的陷阱

m4 允许宏的重定义，结果是新的宏定义会覆盖旧的。例如：

```
define(LEFT, [)dnl
LEFT
define(LEFT, {)dnl
LEFT
```

如果你按照我说『新的宏定义会覆盖旧的』来判断，可能会认为上述文本流经 m4 会变为：

```
[
{
```

然而，事实上 m4 的处理结果是：

```
[
[
```

与理解这个诡异的结果是如何产生的，就需要认真的回顾一下 m4 的工作过程。

我将 m4 处理第一个 `LEFT` 宏定义的过程大致拆解为：

1. 在输入流中，m4 遇到了 `define`，在它的记忆里，`define` 是一个宏；
2. 接下来它遇到了一个 `(`，它会认为这是 `define` 宏参数列表的左定界符；
3. 接下来，它遇到了 `LEFT,`，它会认为 `,` 之前的文本是 `define` 的第一个参数；
4. 接下来，它遇到了 `[)`，他会认为 `[` 是 `define` 的第二个参数，而 `)` 是 `define` 参数列表的右定界符；
5. 它现在终于明白了，`define(LEFT, [)` 是在调用 `define` 宏；
6. m4 对 `define(LEFT, [)` 进行展开，具体的展开过程，我们不得而知，因为 `define` 是 m4 内建的宏。我们只知道在 `define(LEFT, [)` 的展开过程中，m4 会为我们定义 `LEFT` 宏，并且 `define(LEFT, [)` 宏展开完成后，m4 会向输出流发送一个空字串。

当 m4 遇到第二个 `LEFT` 宏定义时，它的过程大致如下：

1. 在输入流中，m4 遇到了 `define`，在它的记忆里，`define` 是一个宏；
2. 接下来它遇到了一个 `(`，它会认为这是 `define` 宏参数列表的左定界符；
3. 接下来，它遇到了 `LEFT,`，它会认为 `,` 之前的文本——`LEFT` 是 `define` 的第一个参数。但是 m4 随即发现 `LEFT` 是一个宏，于是它就将这个宏展开，结果为 `[`，它认为 `[` 才是真正的 `define` 的第一个参数；
4. 接下来，它遇到了 `{)`，他会认为 `{` 是 `define` 的第二个参数，而 `)` 是 `define` 参数列表的右定界符；
5. 它现在终于明白了，`define([, {)` 是在调用 `define` 宏；
6. m4 对 `define([, {)` 进行展开，具体的展开过程，我们不得而知，因为 `define` 是 m4 内建的宏。我们只知道在 `define([, {)` 的展开过程中，m4 会为我们定义 `[` 宏，并且 `define([, {)` 宏展开完成后，m4 会向输出流发送一个空字串。

m4 处理输入流的过程，非常像人类，急功近利，目光短浅，一叶障目，不见泰山，管中窥豹，略见一斑……现在明白了吧！第二个 `LEFT` 宏定义，表面上看起来是重定义了 `LEFT` 宏，实际上定义的是 `[` 宏。

由于 m4 允许用任何符号作为宏名，所以定义一个 `[` 宏，这种行为是合法的，只不过 m4 不会真正的将它视为宏。我一直没有提 m4 的宏命名规则，现在是谈谈它的最好的时机，但是没什么好说的，**在 m4 眼里，只有像 C 函数名的宏名才是真正的宏**，也就是说，**m4 的宏名名规则是：只允许使用字母、数字以及下划线构造宏名，并且宏名只能以字母或下划线开头。只有符合宏名规则的宏，m4 才会将它视为真正的宏。**不过，不符合宏名规则的宏，也是有办法调用的，以后再讲。

如果想真正的实现宏定义，需要借助引号，例如：

```
 define(`LEFT', [)dnl
LEFT
define(`LEFT', {)dnl
LEFT
```

在 m4 语法中，**单重引号具有逃逸的作用**：当 m4 读到带单重引号的文本片段 S 时，它会将 S 的引号消除，然后继续处理 S 之后的文本。

现在可以这样来理解引号的作用：

- m4 将一切没有引号的文本都视为宏。对于已定义的宏，m4 会将其展开；对于未定义的宏，m4 会按其字面将其输出。
- 加了引号的文本，m4 不再检测它们是不是宏，而是将其作为普通文本按字面输出。

也就是说，加了引号的文本，可以让 m4 不需要判断它是不是宏。

记号

现在，我们继续探究 m4 究竟对于输入流都做了些什么。这件事，已经讨论了 3 次了，虽然每一次都比前一次更深入一些，但是迄今为止，真相依然未能堪破。现在应该到堪破真相的时候了。

m4 对输入流是以记号（Token）**为单元进行读取的。一般情况下，m4 会将读取的每个记号直接发送到输出流，但是当 m4 发现某个单词是已定义的宏名时，它会将这个宏展开。在对宏进行展开的过程中，m4 可能会需要读入更多的文本以获取宏的参数。宏展开的结果会被插入到输入流剩余部分的前端，也就是说，宏展开后所得到的文本会被 m4 重新读取，解析为记号，继续处理。**

上面这段文字尤为重要。当 m4 不能如你预期的那样展开你定义的宏，都应该重新理解上面这段文字。

什么样的文本对于 m4 而言是一个记号？带引号的字符串、宏名、宏参数列表、空白字符（包括换行符）、数字以及其他符号（包括标点符号），像这些类别的文本，对于 m4 而言都是记号。对于每种记号，m4 都有相应的处理机制。**数字与标点符号（西文的），它们本身是记号，同时也是某些记号的边界，除非它们出现于带引号的字符串或者宏的参数列表中。**

来看一个例子：

```
define(`definenum', `define(`num', `99')') num
```

若这行文本流经 m4，那么 m4 读到的第一个记号是 `define`。因为 `define` 后面尾随的是 `(`。由于 `(` 即是记号，也是某些记号的边界。m4 读取 `define` 文本之后，就遇到了边界，因此 `define` 是 m4 遇到的一个记号。

然后，m4 开始对 `define` 这个记号进行处理，它发现这个记号是一个带参数的宏。所以它暂停对 `define` 的处理，继续读取并分析 `define` 之后的文本，看是否能获得 `define` 宏的参数列表。

接下来， m4 读取的是 `(`，这是个记号，而且是宏参数列表的左定界符。这对 m4 而言，已经开始经进入了一段可能是参数列表的文本。它期望接下来能遇到一个 `,` 或者 `)`，以得到完整的参数列表记号。

但是接下来，m4 读到的是一个左引号。这时，对 m4 而言，已经开始进入了一个可能是带引号的字符串文本，它期望接下来能遇到一些文本或右引号，以得到一个完整的字符串记号。

但是接下来，m4 读到是文本片段 `definenum`，再读下去，就读到了右引号。这时， m4 很高兴，它确定自己已经读取了一个带引号的字符串记号，然后它就将包围这个字符串的引号消除，继续读取后面的文本。m4 之所以不在这时将 `definenum` 发送到输出端，因为它没有忘记自己还有一个使命：为 `define` 宏搜寻完整的参数列表。

接下来，m4 读到了 `,`——这是宏参数记号的边界。m4 很高兴，它终于得到了 `define` 宏的第一个参数，即 `definenum`。此时，m4 认为刚才读到的 `,` 就没什么用了，于是就将 `,` 消除了，然后它认为后面也许还会有第二个参数，决定继续前进。

接下来，m4 遇到了一个空格。在宏参数列表中，在 `,` 之后的空格是无意义的字符，m4 将这个空格扔掉，继续前进。然后它遇到了左引号……于是就像刚才处理 `definenum` 一样，m4 可以得到一个带引号的字符串：

```
`define(`num', `99')`
```

m4 将这个字符串的引号消除，然后继续前进，结果碰到了 `)`。此时，m4 吁了口气，它终于为 `define` 宏获得了一个完整的参数列表，尽管这个参数列表只含有两个参数。

接下来，m4 对 `define` 宏进行展开。这个过程，我们无法得知，因为 `define` 是 m4 内建的宏，但是我们知道在 `define` 的展开过程中肯定发生了一系列计算，然后 `definenum` 变成了一个宏，最终

```
define(`definenum', `define(`num', `99')')
```

的展开结果是一个空的字符串。由于**宏展开的结果会被插入到输入流剩余部分的前端，也就是说，宏展开后所得到的文本会被 m4 重新读取，解析为记号，继续处理**，因此 m4 会将

```
define(`definenum', `define(`num', `99')')
```

的展开结果视为它下一步继续要读取并处理的文本。当 m4 继续前进时，它就会读到到一个空的字符串。空的字符串，虽然不具备被 m4 发送到输出流的资格，但是它可以作为其他记号的边界记号使用。

接下来，m4 遇到了一个空格字符。空格字符也是个记号，而且是其他记号的边界。m4 将空格记号直接发送到输出流，继续前进。

接下来，m4 一口气读到了输入流的末尾，得到了 `num` 记号。之所以说 `num` 是一个记号，是因为 `num` 的左侧与右侧都有边界，左侧是空格，右侧是输入流终止符。m4 将 `num` 这个记号视为宏，然后它确定这个宏没有被定义，因此无法对其进行展开，所以只好将它作为字符串发送到输出流。

挑战

对于以下 m4 文本

```
define(`definenum', define(`num', `99')) definenum num
```

推测一下 m4 的处理结果，然后执行 m4 命令检验所做的推测是否正确，然后再回顾一次 m4 的工作过程，最后用：

```
$ m4 -dV your-m4-file
```

查看一下输出，根据输出信息再回顾一次 m4 的工作过程。

注释符

`#` 符号是**行注释符**。不过，与我们所熟悉的注释文本不同，m4 的注释文本会被发送到输出流。例如：

```
 define(`VERSION',`A1')
VERSION # VERSION `quote' unmatched`
```

会被展开为：

```
A1 # VERSION `quote' unmatched`
```

可以用 `changecom` 宏修改 m4 默认的注释符，例如

```
changecom(`@@')
```

这样，`@@` 就变成了注释符。

如果你需要**块注释符**，也可以做到，例如：

```
changecom(/*,*/)
```

如果不向 `changecom` 提供任何参数，其他 m4 实现会恢复默认的注释符，但是 GNU m4 不会恢复默认的注释符，而是**关闭** m4 的注释功能。如果要恢复默认的注释符，必须这样：

```
changecom(`#')
```

如果不希望 m4 回显注释文本，可以用 `dnl` 宏替换注释符，例如：

```
define(`VERSION',`A1')
VERSION dnl VERSION `quote' unmatched`
```

`dnl` 会将其后的内容一直连同行尾的换行符统统干掉。

如果让块注释文本不回显，需要基于条件语句进行一些 hack。不过，由于注释这种东西并没有存在的必要，所以就不再理睬它了。之所以说，注释不重要，是因为我们有更强大的注释机制——文式编程！

引号，逃逸以及非 ASCII 字符

m4 有一个不足之处，它**没有专用的逃逸符**。对于非引号字符的字符，引号总是可以作为逃逸符使用。但是，怎么对引号本身进行逃逸呢？毕竟很多场合需要左引号字符作为普通字符出现。

> 事实上，这篇文档是用 Markdown 标记写的，我也无法将左引号符号以 Markdown 行内代码标记表现出来。

虽然可以在引号的外层再封装一层引号从而将前者变为普通字符，例如：

```
I said, ``Quote me.'' # -> I said, `Quote me.'
```

但是，有些时候你只想以普通文本的形式显示左引号，不希望出现一个与之配对的右引号。对于这个问题，可以使用 `changequote` 宏修改 m4 默认的引号定界符，例如：

```
changequote(<!,!>)
a `quoted string
```

m4 会将其处理为：

```
a `quoted string
```

因为此时，真正的引号是 `<!` 与 `!>`。

如果不向 `changequote` 提供任何参数，就恢复了默认的引号定界符。例如：

```
changequote(<!,!>)dnl
a `quoted string

changequote`'dnl
a `quoted string'
```

m4 的处理结果为：

```
a `quoted string

a quoted string
```

一般情况下，应该**避免使用** `changequote`，而是将引号字符定义为宏：

```
define(`LQ', `changequote(<,>)`dnl'
changequote`'')

define(`RQ',`changequote(<,>)dnl`'changequote`'')
```

m4 遇到 `LQ` 宏将其展开为「`」字符，遇到 `RQ` 宏就将其展开为「'」字符。这两个宏的定义所体现的技巧是，临时的改变 m4 默认的引号定界符，然后再改回来。

不过，有时候需要全局性的修改 m4 的默认引号定界符，例如有些键盘上没有「`」字符，或者 m4 要处理的文本必须将「`」字符视为普通字符。使用 `changequote` 一定要小心陷阱：GNU m4 提供的 `changequote` 与其早期版本以及 m4 的其他实现有区别。

为了可移植，要么向 `changequote` 提供 2 个参数来调用它，要么就不提供任何参数，例如：

```
changequote
```

`changequoe` 会改变宏的定义，例如：

```
define(x,``xyz'')
x                       # -> xyz
changequote({,})
x                       # -> `xyz'
```

不要用同样的字符作为引号的定界符，这样做，就无法进行引号的嵌套了。

> Markdown 用于格式化行内代码的标记用的就是相同的『左引号』与『右引号』……这样的错误，诞生于上个世纪 70 年代的 m4 没有犯。

不要将引号定界符更改为以字母、下划线或数字开头的字符。m4 虽然不反对这样做，但是它不认为这种字符是引号定界符。数字作为引号定界符，虽然可以被 m4 认可，但是当它作为一个记号本身的组成元素时，它就失去了引号定界符的身份了。

现在的 GNU m4 可以支持非 ASCII 字符，因此也可以用它们来作为引号定界符，例如：

```
changequote(左引号, 右引号)
a 左引号quoted string右引号    # -> a quoted string

define(我是宏, 我知道你是宏)
我是宏
```

但是最好不要这么干，特别是不要将它们用于宏名。因为，使用 8 位宽的字符，就已经让 m4 行为有些怪异了。GNU m4 1.4.17 版本（本文写作过程中所用的 m4 版本）的手册中说：GNU m4 不理解多字节文本，它只是将文本视为以字节为单位的数据，并且支持 8 位宽的字符作为宏名与引号定界符，但 `NUL` 字符（即 `'\0'`）除外。

m4 能处理中文，这是一种巧合。这种巧合应该只发生在 UTF-8 编码的输入流中。因为 UTF-8 的编码机制决定了中文字符的任何一个字节都与 ASCII 码不同。如果是 GB2312，GB18030 这样的字符集，或许就没有这么好的运气了。

条件

m4 提供了两种条件宏，`ifdef` 宏用于判断宏是否定义，`ifelse` 宏是判断表达式的真假。

```
ifdef(`a', b)
```

对于上述条件宏，如果 `a` 是已定义的宏，那么这条语句的展开结果是 `b`。

```
ifdef(`a', b, c)
```

对于上述条件宏，如果 `a` 是未定义的宏，这条语句的展开结果是 `c`。

被测试的宏，它的定义可以是空字串，例如：

```
define(`def')
`def' is ifdef(`def', , not) defined.  # -> def is defined.
```

`ifelse(a,b,c,d)` 会比较字符串 `a` 与 `b` 是否相同，如果它们相同，这条语句的展开结果是字符串 `c`，否则展开为字符串 `d`。

`ifelse` 可以支持多个分支，例如：

```
ifelse(a,b,c,d,e,f,g)
```

它等价于：

```
ifelse(a,b,c,ifelse(d,e,f,g))
```

数字

m4 只认识文本，所以在它看来，数字也是文本。不过 m4 提供了内建宏 `eval`，这个宏可以对整型数的运算表达式进行『求值』——求值结果在 m4 看来依然是文本。

例如：

```
define(`n', 1)dnl
`n' is ifelse(eval(n < 2), 1, less than, eval(n == 2), 1, , greater than) 2
```

`eval(n < 2)` 是对 `n < 2` 这个逻辑表达式进行『求值』，结果是字符串 `1`，因此 `ifelse` 的第一个参数与第二个参数相等，因此 `ifelse` 宏的展开结果是其第三个参数 `less than`，所以展开结果为：

```
n is less than 2
```

我觉得没必要用 m4 来计算，因为它提供的计算功能太孱弱。可以考虑用 GNU bc 来弥补它的不足。在 m4 中，可以通过 `esyscmd` 宏访问 Shell，例如：

```
2.1 ifelse(eval(esyscmd(`echo "2.1 > 2.0" | bc')), 1, `greater than', `less
than') 2.0
```

展开结果为：

```
2.1 greater than 2.0
```

不过，`esyscmd` 是 GNU m4 对 `syscmd` 的扩展，别的 m4 的实现可能没有这个宏。

挑战

(1) 如果用 m4 处理 C 代码文件，将 `#` 符号作为 m4 的行注释符，会有哪些显而易见的好处？

(2) 借助 GNU m4 提供的 `esyscmd` 宏，结合 GNU bc，写一个可以计算数字平方根的的宏。

递归

现在再强调一次，m4 会将当前宏的展开结果插入到待读取的输入流的前端。也就是说，m4 会对当前宏的展开结果再次进行扫描，以处理嵌套的宏调用。这个性质决定了可以写出让 m4 累死的递归宏。

例如：

```
define(`TEST', `TEST')
TEST
```

当 m4 试图对 `TEST` 进行展开时，它就会永无休止的去展开 `TEST`，而每次展开的结果依然是 `TEST`。

既然有递归，那么就可以利用递归来做一些计算，为了让递归能够结束，这就需要 m4 能够支持条件。幸好，我们已经知道 m4 是支持条件的。下面，是一个递归版本的 Fibonacci 宏的实现与应用，它可以产生第 47 个 Fibonacci 数：

```
divert(-1)
define(`FIB',
`ifelse(`$1', `0',
        0,
       `ifelse(`$1', `1',
               1,
              `eval(FIB(eval($1 - 1)) + FIB(eval($1-2)))')')')
divert(0)dnl
FIB(46)
```

m4 的展开结果应该是 `1836311903`。也许你要等很久才会看到这个结果。因为递归的 Fibonacci 数计算过程中包含着大量的重复计算，效率很低。

不过，迭代版本的 Fibonacci 数计算过程也能写得出来：

```
divert(-1)
define(`FIB_ITER',
`ifelse(`$3', 0,
        $2,
     `FIB_ITER(eval($1 + $2), $1, eval($3 - 1))')')
define(`FIB', `FIB_ITER(1, 0, $1)')
divert(0)dnl
FIB(46)
```

迭代计算很快，在我的机器上只需要 0.002 秒就可以得出 `1836311903` 这个结果。不过，如果想尝试比 46 更大的数，比如 `FIB(47)`，结果就会出现负数。这是因为 m4 目前只支持 32 位的有符号整数，它能表示的最大正整数是 2^31 - 1，而 `FIB(47)` 的结果会大于这个数。

循环

既然有递归，那么就可以用它来模拟循环。例如：

```
define(`for',
`ifelse($#,
        0,
        ``$0'',
        `ifelse(eval($2 <= $3),
                1,
               `pushdef(`$1',$2)$4`'popdef(`$1')$0(`$1', incr($2), $3, `$4')')')')
```

这个 `for` 宏可以像下面这样调用：

```
for(`i', 1, n, `循环内的计算')
```

它类似于 C 语言中的 `for` 循环：

```
for(int i = 1; i <= n; i++) {
        循环内的计算
}
```

例如，可以用 `for` 宏将 64 个 `-` 符号发送到输出流：

```
for(`i', 1, 64, `-')
```

这个宏的展开结果为：

```
----------------------------------------------------------------
```

如果你用过 reStructuredText 标记语言，一定会知道怎么用 `for` 宏构建一个协助你构造一个用于快速撰写 reStructuredText 标题标记的宏。

要理解 `for` 宏的定义，有几个 m4 小知识需要补习一下。请向下看。

宏参数列表的特征值

我们已经知道 `$1, $2, ..., $9` 对应于宏参数列表中的各个参数（GNU m4 不限定参数的个数，其他 m4 实现最多支持 9 个参数）。如果对 C 或 Bash 有所了解，那么当我说 `$0` 是宏本身，估计不会觉得很奇怪。因此，在上一节 `for` 宏定义中，`$0` 表示引用了宏名 `for`。不妨将 `$0` 改成 `for` 试一下。

`$#` 表示宏参数的个数。例如：

```
define(`count', ``$0': $# args')
count         # -> count: 0 args
count()       # -> count: 1 args
count(1)      # -> count: 1 args
count(1,)     # -> count: 2 args
```

> `#` 是注释符，`->` 后面的文本是 m4 对注释符号之前的文本处理后发送到输出流的结果。

值得注意的是，即使 `()` 内什么也没有，m4 也会认为 `count` 宏是有一个参数的，它是空字串。

`for` 的定义中，第一处条件语句为：

```
ifelse($#,
       0,
       ``$0'',
       ... ...)
```

它的作用就是告诉 m4，遇到 `for` 的调用语句，如果 `for` 的参数个数为 0，那么 `for` 的展开结果为带引号的字符串：

```
`for'
```

要理解为什么在条件语句中，`for` 用两重引号包围起来，你需要再认真的复习一次 m4 的宏展开过程。如果用单重引号，那么以无参数的形式调用 `for` 宏时，m4 会陷入对 `for` 宏无限次的展开过程中。

宏的作用域

所有的宏都是全局的。

如果我们需要『局部宏』该怎么做？也就是说，如何将一个宏只在另一个宏的定义中使用？局部宏的意义就类似于编程语言中的局部变量，如果没有局部宏，那么在一个全局的空间中，很容易出现宏名冲突，导致宏被意外的重定义了。

为了避免宏名冲突，一种可选的方法是在宏名之前加前缀，比如使用 `local` 作为局部宏名的前缀。不过，这种方法对于递归宏无效。更好的方法是用**栈**。

m4 实际上是用一个栈来维护宏的定义的。当前宏的定义位于栈顶。使用 `pushdef` 可以将一个临时定义的宏压入栈中，在利用完这个临时的宏之后，再用 `popdef` 将其弹出栈外。例如：

```
 define(`USED',1)
define(`proc',
  `pushdef(`USED',10)pushdef(`UNUSED',20)dnl
``USED' = USED, `UNUSED' = UNUSED`'dnl
`'popdef(`USED',`UNUSED')')
proc      # -> USED = 10, UNUSED = 20
USED      # -> 1
```

如果被压入栈的宏是未定义的宏，那么 `pushdef` 就相当于 `define`。如果 `popdef` 弹出的宏也是未定义的宏，`popdef` 就相当于 `undefine`，它不会产生任何抱怨。

GNU m4 认为 `define(X, Y)` 与 `popdef(X)pushdef(X, Y)` 等价。其他的 m4 实现会认为 `define(X)` 等价于 `undefine(X)define(X, Y)`，也就是说，新的宏的定义会更新整个栈。 `undefine(X)` 就是取消 `X` 宏的定义，使之成为未定义的宏。

让宏名更安全

m4 有一个 `-P` 选项，它可以强制性的在其内建宏名之前冠以 `m4_` 前缀。例如下面的 M1.m4 文件：

```
 define(`M1',`text1')M1           # -> define(M1,text1)M1
m4_define(`M1',`text1')M1        # -> text1
```

直接用 m4 处理，结果为：

```
 $ m4 M1.m4
text1           # -> define(M1,text1)M1
m4_define(M1,text1)text1        # -> text1
```

如果用 `m4 -P` 来处理，结果为：

```
 $ m4 -P test.m4
define(M1,text1)M1           # -> define(M1,text1)M1
text1        # -> text1
```

挑战

理解 `for` 宏的定义。

难以驾驭的引号

对于自己定义的宏，建议先在你的大脑中对它进行逐步展开，确信自己完全理解这个展开过程。如果大脑的堆栈不够用，可以用纸和笔记录展开过程。这样可以在很大程度上提高宏定义的正确性。

m4 宏调用的复杂之处在于嵌套的宏调用——在一个宏的展开结果中调用了其他宏。例如，宏 `A` 的展开结果中调用了宏 `X`，如果期望 `X` 先于 `A` 被 m4 展开，那么在 `A` 的定义中就不要在 `X` 的外围加引号。如果在期望 `A` 展开后，当 m4 再度读取 `A` 的展开结果的过程中再展开 `X`，那么 `X` 的外围必须要有引号。再复杂一些，如果宏 `X` 的展开结果中又调用了宏 `Y`，并且期望 `Y` 是在 m4 再度读取 `X` 展开结果的过程中被展开，那么 `Y` 的外围也必须要有一重引号，此时因为 `X` 外围已经有了一重引号，那么 `Y` 实际上是处于两重引号的包裹之中。

m4 处理引号的基本规则是：**在读取带引号的文本片段 S 时，无论 S 中含有多少重引号，m4 只消除其最外层引号，然后将剩余的文本直接发送到输出流。**这个规则很简单，之前已经提到过一次。需要注意的是，如果在宏的参数列表中出现了引号，一定要记住宏的参数列表总是在宏展开之前被处理的。看下面的例子：

```
define(`echo', `$1')
define(`test', echo($1))
test(test)
```

在 `test` 宏定义过程中，`echo($1)` 先被 m4 展开了，结果为空字串，导致 `test` 宏定义语句中的宏体变成空字串，即：

```
define(`test', `')
```

接下来，`test(test)` 是嵌套的宏调用，括号内的 `test` 会先被展开，展开结果是空字串，导致括号外面的 `test` 被展开之前的形式变为：

```
test()
```

此时，`test` 宏接受了一个参数——空字串，然后它会被 m4 展开，展开结果为空字串。这个结果并非是因为 `test` 宏接受了空字串参数所导致的。

现在改动一下 `test` 的定义：

```
define(`echo', `$1')
define(`test',`echo($1)')
test(test)
```

由于引号的抑制作用，`test` 宏体中的 `echo` 不会先于 `test` 定义完成之前被 m4 展开。`test(test)` 的宏展开次序依然同上，m4 先展开括号里面的 `test`，得到：

```
test(echo())
```

然后，m4 不会去展开括号外层的 `test`，而是先去展开括号里面的 `echo` 宏，因为它认为括号里面的文本是括号外面的 `test` 宏的参数，结果变为：

```
test()
```

接下来，`test()` 会被展开为空字串。

下面改动一下 `test` 宏调用语句：

```
define(`echo', `$1')
define(`test',`echo($1)')
test(`test')
```

这时，括号里面的 `test` 就不再是宏调用了，而是括号外面的 `test` 宏的一个参数。`test(` test `)` 宏会被展开为：

```
echo(test)
```

由于 m4 会将宏的展开结果插入到剩余的输入流中继续读取并处理，所以上述结果被进一步处理为：

```
echo(echo())
```

再进一步处理为：

```
echo()
```

最终的处理结果依然是一个空字串。

虽然这两次改动并没有得到新的结果，但是显然宏展开的过程并不相同。宏参数中的引号的作用并不是那么显而易见。大部分 m4 宏出错，宏参数中的引号往往是首恶元凶。要驾驭它，只能凭借自己的明确的逐步推导。这也导致了一个问题，很难用 m4 描述复杂的宏逻辑。

作为一次小挑战，请用笔在纸上推导下面 m4 宏的展开结果：

```
define(`echo', `$1')
define(`test',`echo($1)')
test(``test'')
```

然后使用 `m4 -dV your-m4-file` 印证自己的推导。注意， `m4 -dV` 所显示的宏展开过程，会对每个宏的展开结果包装一层引号，这其实是多余的引号，它只代表 m4 对宏的展开结果总是字符串。

非法的宏名

下面这个宏定义：

```
define(`?N?', 1)
```

m4 会认为 `?N?` 这个宏是不合法的，因为合法的宏的名字必须要遵守正则表达式 `[_a-zA-Z][_a-zA-Z0-9]*`。不过，GNU m4 是仁慈的，对于不合法的宏，它依然能展开，前提是借助 m4 内建的 `defn` 宏：

```
?N?           # -> ?N?
defn(`?N?')   # -> 1
```

非法的宏名可以用来模拟数组或 Hash 表，例如：

```
define(`_set', `define(`$1[$2]', `$3')')
define(`_get', `defn(`$1[$2]')')
_set(`myarray', 1, `alpha')
_get(`myarray', 1)                  # -> alpha
_set(`myarray', `alpha', `omega')
_get(`myarray', _get(`myarray',1))  # -> omega
defn(`myarray[alpha]')              # -> omega
```

外援

GNU m4 内建了几个与 Shell 交互的宏，诸如 `syscmd`，`esyscmd`，`sysval`，`mkstemp` 等，其中最有用的是 `esyscmd`，因为它不仅能访问 Shell，而且还能获取 Shell 命令产生的输出。例如，下面这行 m4 代码可以借助 Shell 调用 GNU guile——GNU 的 Scheme 解释器来计算阶乘：

```
esyscmd(`guile -c "(define (factorial n)
                     (if (= n 1)
                      1
                     (* n (factorial (- n 1)))))
                   (display (factorial 100))"')
```

如果你的系统中安装了 GNU guile，并且有一个 Shell 可用（既然是 m4 用户，系统中没有 Shell 说不过去的），那么 m4 对上述 `esyscmd` 宏的展开结果为：

```
933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000000000
```

这样写也行：

```
 define(`scheme_code',
`"`(define (factorial n)
   (if (= n 1)
    1
    (* n (factorial (- n 1)))))
 (display (factorial 100))'"')

esyscmd(`guile -c' scheme_code)
```

凡是能在 Shell 中运行并产生输出的程序，皆能被 GNU m4 所用，这是不是很神奇？

文本处理

我一直都忍着不去谈 GNU m4 针对文本处理提供的几个内建宏，主要是因为既然有 `esyscmd` 这样的宏可用，那么类 Unix 系统中那些很无敌的文本处理工具，诸如 tr, cut, paste, wc, md5sum, sed, awk, grep/egrep 等等，它们皆能被 m4 所用，那么何必再多此一举？

倘若是为了让 m4 脚本更具备可移植性，那么最好是将一个比较完整的 Shell 环境移植到目标平台……对于主流操作系统而言，这并不是太困难的事，因为已经有了很多针对不同操作系统的完整的 Shell 环境实现。

如果依然坚持用 m4 的方式处理文本，建议阅读：『GNU m4 Text Handling』。

结束语

这份 GNU m4 指南至此终结。作为学习者，务必要记住 m4 官方手册的告诫之语：**有些人对 m4 非常着迷，他们先是用 m4 解决一些简单的问题，然后解决了一个比一个更大的问题，直至掌握如何编写一个复杂的 m4 宏集。若痴迷于此，往往会对一些简单的问题写出复杂的 m4 脚本，然后耗费很多时间去调试，反而不如直接手动解决问题更有效。所以，对于程序猿中的强迫症患者，要对 m4 有所警惕，它可能会危及你的健康。**

如果不想让 m4 危及你的健康，永远要记住：**宏是用来缩写那些复杂但是又经常重复出现的文本模式的。**

来源：https://segmentfault.com/a/1190000004137562

来源：https://segmentfault.com/a/1190000004131031

来源：https://segmentfault.com/a/1190000004128102

来源：https://segmentfault.com/a/1190000004108113

来源：https://segmentfault.com/a/1190000004104696