Z Shell 手册

Table of Contents

---

1 Z Shell 手册

本文档是根据 Zsh 发行版 Doc 子目录中的 texinfo 文件 `zsh.texi` 制作的。

1.1 从 zsh.texi 制作文档

texinfo 源文件可转换成多种格式：

Info 手册

> Info 格式允许从众多索引中搜索主题、命令、函数等。命令 ‘`makeinfo zsh.texi`’用于生成 Info 文档。

印刷手册

> 命令‘`texi2dvi zsh.texi`’将输出 `zsh.dvi`，然后使用 *dvips* 和可选的 *gs*（Ghostscript）对其进行处理，以生成格式精美的打印手册。

HTML 手册

本手册的 HTML 版本可通过 Zsh 网站获取：

https://zsh.sourceforge.io/Doc/。

(HTML 版本通过 *texi2html* 生成，可从 http://www.nongnu.org/texi2html/ 获取。命令为'`texi2html --output . --ifinfo --split=chapter --node-files zsh.texi`'。 如有必要，请升级至 1.78 版 texi2html)

对于那些没有处理 texinfo 所需的工具的用户，可从 zsh 归档站点或其镜像站点获取预编译文档（PostScript、dvi、PDF、info 和 HTML 格式），文件名为 `zsh-doc.tar.gz`。(站点列表请参见可用性）。

---

2 简介

Zsh 是一种 UNIX 命令解释器（shell），可用作交互式登录 shell 和 shell 脚本命令处理器。 在标准 shell 中，zsh 与 *ksh* 最为相似，但包含许多增强功能。 在默认运行模式下，zsh 与 POSIX 或其他 shell 不兼容：请参阅 兼容性 章节。

Zsh 具有命令行编辑、内置拼写校正、可编程命令补全、shell 函数（带自动加载功能）、历史记录机制以及大量其他功能。

2.1 作者

Zsh 最初由 Paul Falstad 编写。 Zsh 现在由 zsh-workers 邮件列表 <zsh-workers@zsh.org> 的成员维护。 目前由 Peter Stephenson <pws@zsh.org>负责协调开发工作。 您可以通过 <coordinator@zsh.org> 与协调人联系，但有关代码的事宜一般应在邮件列表中进行讨论。

2.2 可用性

Zsh 可从以下 HTTP 和匿名 FTP 站点获取。

ftp://ftp.zsh.org/pub/
https://www.zsh.org/pub/

最新源代码可通过 Git 从 Sourceforge 获取。 详情请参见 https://sourceforge.net/projects/zsh/。 有关存档的说明摘要，请参见 https://zsh.sourceforge.io/。

2.3 邮件列表

Zsh 有多个邮件列表：

`<zsh-announce@zsh.org>`

有关版本发布、shell 重大变更以及每月发布的 Zsh 常见问题的公告。（审核）

`<zsh-users@zsh.org>`

用户讨论。

`<zsh-workers@zsh.org>`

黑客、开发、错误报告和补丁。

`<zsh-security@zsh.org>`

私人邮件列表（一般公众无法订阅），用于讨论具有安全影响的错误报告，即潜在漏洞。

如果您发现 zsh 本身存在安全问题，请发送邮件至此地址。

如需订阅或取消订阅，请向邮件列表的相关管理地址发送邮件。

```
<zsh-announce-subscribe@zsh.org>
<zsh-users-subscribe@zsh.org>
<zsh-workers-subscribe@zsh.org>
<zsh-announce-unsubscribe@zsh.org>
<zsh-users-unsubscribe@zsh.org>
<zsh-workers-unsubscribe@zsh.org>
```

您只需加入其中一个邮件列表，因为它们是嵌套的。 所有提交到 *zsh-announce* 的内容都会自动转发给 *zsh-users*。 所有向 *zsh-users* 提交的内容都会自动转发给 *zsh-workers*。

如果您在订阅/退订任何邮件列表时遇到问题，请发送邮件至 `<listmaster@zsh.org>`。

邮件列表已存档；可通过上述管理地址访问存档。 此外，https://www.zsh.org/mla/还提供超文本档案。

2.4 Zsh 常见问题

Zsh 有一个常见问题（FAQ）列表，由 Peter Stephenson <pws@zsh.org> 维护。 它定期发布到 *comp.unix.shell* 新闻组和 *zsh-announce* 邮件列表。 最新版本可在任何 Zsh FTP 站点或 https://www.zsh.org/FAQ/ 上找到。 常见问题相关事宜的联系地址是 <faqmaster@zsh.org>。

2.5 Zsh 网页

Zsh 有一个网页，位于 https://www.zsh.org/。 网站相关事宜的联系地址是 <webmaster@zsh.org>。

2.6 Zsh 用户指南

目前正在编写用户指南。 该指南旨在对手册进行补充，对手册中可能存在的卡巴拉式、等级式或完全神秘化（例如，"等级式 "一词并不存在）的问题进行解释和提示。 您可以通过 https://zsh.sourceforge.io/Guide/ 查看该手册的当前状态。 在撰写本文时，关于启动文件及其内容以及新的补全系统的章节已基本完成。

2.7 另请参见

sh(1), csh(1), tcsh(1), rc(1), bash(1), ksh(1)

IEEE Standard for information Technology - Portable Operating System Interface (POSIX) - Part 2: Shell and Utilities, IEEE Inc, 1993, ISBN 1-55937-255-9.

---

3 路线图

Zsh 手册，就像 shell 本身一样，庞大而复杂。 手册的这一部分为新用户可能特别感兴趣的 shell 领域提供了一些提示，并指出了在手册其余部分在文档中的位置。

3.1 当 shell 启动时

shell 启动时会从各种文件中读取命令。 可以创建或编辑这些文件来定制 shell。 参见 文件。

如果当前用户不存在个人初始化文件，则会运行一个函数来帮助您更改一些最常用的设置。 如果管理员禁用了 zsh/newuser 模块，该函数将不会出现。 该函数的设计不言自明。 你可以使用‘autoload -Uz zsh-newuser-install; zsh-newuser-install -f’手动运行它。 另请参阅 用户配置函数。

3.2 互动式使用

与 shell 的交互使用内置的 Zsh 行编辑器（ZLE）。 Zsh 行编辑器中对此有详细介绍。

用户必须做出的第一个决定是使用 Emacs 还是 Vi 编辑模式，因为两者的编辑键大不相同。 对于初学者来说，Emacs 编辑模式可能更为自然，可以使用 `bindkey -e` 命令明确选择。

历史记录机制可用于检索以前键入的行（最简单的方法是使用向上或向下箭头键）；请注意，与其他 shell 不同，zsh 在退出 shell 时不会保存这些行，除非您设置了适当的变量，而且默认情况下保留的历史行数很少（30 行）。请参阅 Shell 使用的参数 中对 shell 变量（在文档中称为参数）`HISTFILE`、`HISTSIZE` 和 `SAVEHIST` 的描述。 需要注意的是，目前只有在 shell 处于交互状态时才能读写保存历史的文件，也就是说，它无法通过脚本工作。

shell 现在支持 UTF-8 字符集（以及操作系统支持的其他字符集）。 这（大部分）由 shell 透明处理，但终端模拟器的支持程度不一。 shell FAQ `https://www.zsh.org/FAQ/`中对此有一些讨论。 请特别注意，要处理组合字符，需要设置 `COMBINING_CHARS` 选项。 由于 shell 现在对字符集的定义更加敏感，因此如果您是从旧版本的 shell 升级，则应确保设置了 `LANG`（影响 shell 运行的所有方面）或 `LC_CTYPE`（仅影响字符集的处理）这两个变量。(只影响字符集的处理）设置为适当的值。 即使使用的是单字节字符集，包括 ASCII 扩展字符集，如 `ISO-8859-1` 或 `ISO-8859-15`，也是如此。 请参阅 参数 中 `LC_CTYPE` 的描述。

3.2.1 补全

补全是许多 shell 中都有的一项功能。它允许用户只键入单词的一部分（通常是前缀），然后由 shell 来补全其余部分。 zsh 中的补全系统是可编程的。 例如，可以设置 shell 在 `~/.abook/addressbook` 的邮件命令参数中补全电子邮件地址；在 scp 的参数中补全用户名、主机名甚至远程路径，等等。 任何可以用 zsh 写入或粘合在一起的内容都可以成为行编辑器提供的可能补全内容的来源。

Zsh 有两个补全系统，一个是旧的补全系统，即 `compctl`（以作为其完整且唯一用户界面的内置命令命名），另一个是新的补全系统，即 `compsys`，由内置函数库和用户自定义函数库组成。 这两个系统在指定补全行为的界面上有所不同。 新系统的可定制性更强，并为许多常用命令提供了补全功能，因此更受欢迎。

必须在 shell 启动时明确启用补全系统。 更多信息，请参阅 补全系统。

3.2.2 扩展行编辑器

除补全功能外，行编辑器还可通过 shell 函数进行高度扩展。 shell 提供了一些有用的功能，如:

`insert-composed-char`

编写键盘上没有的字符

`match-words-by-style`

在按字移动或删除时，配置行编辑器认为什么是字

`history-beginning-search-backward-end`, 等等。

搜索 shell 历史记录的其他方法

`replace-string`, `replace-pattern`

在命令行中全局替换字符串或模式的函数

`edit-command-line`

使用外部编辑器编辑命令行。

请参阅 ZLE 函数，了解这些函数的描述。

3.3 选项

shell 有大量用于改变其行为的选项。 这些选项涵盖了 shell 的方方面面；浏览完整的文档是熟悉众多选项的唯一好方法。 参见 选项。

3.4 模式匹配

shell 有一套丰富的模式，可用于文件匹配（在文档中称为 '文件名生成'，由于历史原因也称为 'globbing'）和编程。 这些模式在 文件名生成 中有所描述。

Of particular interest are the following patterns that are not commonly supported by other systems of pattern matching:

`**`

用于匹配多个目录

`|`

用于匹配两个备选方案中的任何一个

`~`, `^`

当设置 `EXTENDED_GLOB` 选项时，能够从匹配中排除模式

`(...)`

glob 限定符，包含在模式末尾的括号中，可按类型（如目录）或属性（如大小）选择文件。

3.5 关于语法的一般性评论

尽管 zsh 的语法在某些方面与 Korn shell 相似，因此更接近于最初的 UNIX shell，即 Bourne shell，但其默认行为并不完全与这些 shell 一致。 一般 shell 语法在 Shell 语法 中介绍。

一个常见的区别是，替换到命令行中的变量不会被分割成单词。 参见 参数扩展 中 shell 选项 SH_WORD_SPLIT 的描述。 在 zsh 中，您可以明确请求分割（例如 ${=foo}），或者在希望变量扩展为多个单词时使用数组。 请参见 数组参数。

3.6 编程

为 shell 添加增强功能的最便捷方法通常是编写 shell 函数，并安排其自动加载。 函数在 函数 中有所描述。 从 C shell 及其近似 shell 转变而来的用户应该注意到，别名在 zsh 中使用较少，因为它们不执行参数替换，而只是简单的文本替换。

除了上述用于行编辑器的功能外，Shell 还提供了一些常规功能，这些功能在 用户贡献 中进行了描述。 这些功能包括:

`promptinit`

提示符主题系统，可轻松更改提示符，请参阅 提示符主题

`zsh-mime-setup`

一种 MIME 处理系统，它能像图形文件管理器那样，根据文件的后缀发送命令

`zcalc`

计算器

`zargs`

`xargs` 的另一个版本，使 `find` 命令变得多余

`zmv`

重命名文件的命令。

---

4 调用

4.1 调用

调用 shell 时，shell 会解释以下标志，以确定 shell 从何处读取命令：[译注:调用]

`-c`

将第一个参数作为要执行的命令，而不是从脚本或标准输入中读取命令。 如果还有其他参数，第一个参数会被赋值给 `$0`，而不是作为位置参数使用。[译注:调用.标志.c]

`-i`

强制 shell 为交互式。 仍可指定要执行的脚本。

`-s`

强制 shell 从标准输入端读取命令。 如果没有 `-s` 标志，且给出了参数，则第一个参数将被视为要执行的脚本的路径名。

如果选项处理后还有剩余参数，且未提供 `-c` 或 `-s` 选项，则第一个参数将作为包含要执行的 shell 命令的脚本文件名。 如果设置了选项 `PATH_SCRIPT`，且文件名不包含目录路径（即文件名中没有'`/`'），那么首先会搜索当前目录，然后在变量 `PATH` 给出的命令路径中查找脚本。 如果未设置该选项或文件名中包含'`/`'，则直接使用该脚本。

在前一或两个参数按上述方式分配后，其余参数将分配给位置参数。

有关调用和 `set` 内置函数通用的其他选项，请参见 选项。

可以向 shell 传递长选项'`--emulate`'，后跟仿真模式（用单独的字）。 仿真模式为 `emulate` 内置命令所描述的模式，参见 Shell 内置命令。'`--emulate`'选项必须先于其他选项（否则可能被覆盖），但后面的选项会被执行，因此可以用来修改请求的仿真模式。需要注意的是，与 shell 中的 `emulate` 命令相比，在使用该选项时，会采取一些额外的步骤来确保仿真的顺利进行：例如，不会在 shell 中定义与 POSIX 使用相冲突的变量，如 `path`。

可以使用 `-o` 选项指定选项名称。 `-o` 的作用与单字母选项类似，但选项名称需要使用下面的字符串。 例如

```
zsh -x -o shwordsplit scr
```

运行脚本 `scr`，用相应的字母'`-x`'设置 `XTRACE` 选项，用名称设置 `SH_WORD_SPLIT` 选项。 使用 `+o` 代替 `-o`，可以按名称 **关闭** 选项。 `-o` 可以与前面的单字母选项叠加，例如，'`-xo shwordsplit`'或'`-xoshwordsplit`'等同于'`-x -o shwordsplit`'。

选项名称也可以用 GNU 长选项风格指定，即'`--`*option-name*'。 这样做时，选项名称中的'`-`'字符是允许的：它们会被翻译成'`_`'，因此会被忽略。 因此，举例来说，'`zsh --sh-word-split`'调用的是打开 `SH_WORD_SPLIT` 选项的 zsh。 与其他选项语法一样，可以通过用'`+`'替换开头的'`-`'来关闭选项；因此'`+-sh-word-split`'等同于'`--no-sh-word-split`'。 与其他选项语法不同，GNU 风格的长选项不能与任何其他选项堆叠，因此，例如'`-x-shwordsplit`'是一个错误，而不是像'`-x --shwordsplit`'那样处理。

特殊的 GNU 风格选项‘--version’会被处理；它会向标准输出发送 shell 的版本信息，然后成功退出。 ‘--help’ 也会被处理；它会向标准输出发送调用 shell 时可以使用的选项列表，然后成功退出。

选项处理可以通过两种方式完成，即允许将后面以‘-’ 或 ‘+’开头的参数视为普通参数。 首先，单独的参数‘-’（或‘+’）本身会结束选项处理。 其次，特殊选项‘--’(或 ‘+-’)可以单独指定（这是 POSIX 的标准用法），也可以与前面的选项堆叠（因此‘-x-’等同于‘-x --’）。 选项不允许堆叠在‘--’之后（因此‘-x-f’是错误的），但请注意上文讨论的 GNU 风格选项形式，其中‘--shwordsplit’是允许的，并且不会结束选项处理。

除非启用了 *sh*/*ksh* 模拟单字母选项效果，否则选项‘-b’ (或 ‘+b’) 会结束选项处理。 ‘-b’类似于‘--’，只是在‘-b’ 之后可以堆叠更多的单字母选项，并按正常方式生效。

4.2 兼容性

当 Zsh 分别以 sh 或 ksh 的方式调用时，它会尝试模拟 *sh* 或 *ksh*；更确切地说，它会查看调用名称的第一个字母，不包括任何首字母‘r’（假定代表 "受限"），如果是‘b’、‘s’或‘k’，它就会模拟 *sh* 或 *ksh*。 此外，如果以 su 的方式调用 shell（在某些系统中，通过 su 命令执行 shell 时会出现这种情况），shell 会尝试从 SHELL 环境变量中找到一个替代名称，并以此为基础进行仿真。

在 *sh* 和 *ksh* 兼容模式下，以下参数并不特殊，也不会被 shell 初始化: ARGC, argv, cdpath, fignore, fpath, HISTCHARS, mailpath, MANPATH, manpath, path, prompt, PROMPT, PROMPT2, PROMPT3, PROMPT4, psvar, status.

通常的 zsh 启动/关闭脚本不会被执行。 登录 shell 引入（source）/etc/profile，然后是 $HOME/.profile。 如果在调用时设置了 ENV 环境变量，则 $ENV 会在 profile 脚本之后引入（sourced）。 在将 ENV 的值解释为路径名之前，会对其进行参数扩展、命令替换和算术扩展。 请注意，PRIVILEGED 选项也会影响启动文件的执行。

如果以 sh 或 ksh 调用 shell，则会设置以下选项: NO_BAD_PATTERN, NO_BANG_HIST, NO_BG_NICE, NO_EQUALS, NO_FUNCTION_ARGZERO, GLOB_SUBST, NO_GLOBAL_EXPORT, NO_HUP, INTERACTIVE_COMMENTS, KSH_ARRAYS, NO_MULTIOS, NO_NOMATCH, NO_NOTIFY, POSIX_BUILTINS, NO_PROMPT_PERCENT, RM_STAR_SILENT, SH_FILE_EXPANSION, SH_GLOB, SH_OPTION_LETTERS, SH_WORD_SPLIT. 此外，如果以 sh 的方式调用 zsh，则会设置 BSD_ECHO 和 IGNORE_BRACES 选项。 如果以 ksh 的方式调用 zsh，还会设置 KSH_OPTION_PRINT、LOCAL_OPTIONS、PROMPT_BANG、PROMPT_SUBST 和 SINGLE_LINE_ZLE 选项。

请注意，尽管在出现不兼容问题时，zsh 会尽力解决，但并不保证完全模拟其他 shell，也不保证符合 POSIX。有关 zsh 与其他 shell 之间差异的更多信息，请参阅 shell 常见问题 第 2 章 https://www.zsh.org/FAQ/。

4.3 受限的 Shell

当用于调用 zsh 的命令基名以字母‘r’开头，或在调用时提供了‘-r’命令行选项时，shell 将成为受限模式。 仿真模式是在去掉调用名称中的字母 ‘r’ 后确定的。 在受限模式下，以下功能将被禁用：

- 使用 cd 内置命令更改目录
- 更改或取消设置 EGID, EUID, GID, HISTFILE, HISTSIZE, IFS, LD_AOUT_LIBRARY_PATH, LD_AOUT_PRELOAD, LD_LIBRARY_PATH, LD_PRELOAD, MODULE_PATH, module_path, PATH, path, SHELL, UID 和 USERNAME 参数
- 指定包含 / 的命令名称
- 使用 hash 指定命令路径名
- 将输出重定向到文件
- 使用 exec 内置命令将 shell 替换为另一条命令
- 使用 jobs -Z 覆盖 shell 进程的参数和环境空间
- 使用 ARGV0 参数覆盖外部命令的 argv[0]
- 使用 set +r 或 unsetopt RESTRICTED 关闭受限模式

这些限制会在处理启动文件后执行。 启动文件应设置 PATH，指向可在受限环境中安全调用的命令目录。 启动文件还可以通过禁用选定的内置命令来添加更多限制。

受限模式也可以通过设置 RESTRICTED 选项随时激活。 即使 shell 仍未处理完所有启动文件，也能立即启用上述所有限制。

shell的 **受限模式** 是一种过时的限制用户行为的方法：现代系统有更好、更安全、更可靠的方法来限制用户行为，例如 ***chroot jails***、***containers*** 和 ***zones***。

限制的 shell 很难安全地实现。 该功能可能会在未来的 zsh 版本中删除。

需要注意的是，这些限制只适用于 shell，而不适用于 shell 运行的命令（某些 shell 内置命令除外）。 虽然受限的 shell 只能运行通过预定义的‘PATH’变量访问的受限命令列表，但并不能阻止这些命令运行任何其他命令。

例如，如果 **允许** 命令列表中有‘env’，则允许用户运行任何命令，因为 ‘env’ 不是 shell 内置命令，可以运行任意可执行文件。

因此，在实施受限的 shell 框架时，必须充分了解每个 **允许** 的命令或功能（可视为 **模块**）可以执行哪些操作。

许多命令的行为都会受到环境变量的影响。 除上述少数命令外，zsh 并不限制环境变量的设置。

如果 ‘perl’、‘python’、 ‘bash’或其他通用解释脚本被视为受限命令，用户可以通过设置特制的‘PERL5LIB’、‘PYTHONPATH’、 ‘BASHENV’（等）环境变量来规避限制。 在

GNU 系统中，通过设置 'GCONV_PATH'环境变量，任何命令都可以在执行字符集转换时运行任意代码（包括 zsh 本身）。 以上只是几个例子。

请记住，与其他一些 shell 不同，'readonly'在 zsh 中并不是一种安全功能，因为它可以被撤销，所以不能用来缓解上述问题。

受限的 shell 只有在允许的命令较少且编写谨慎以避免授予用户超出预期的访问权限时才会起作用。 限制用户可以加载的 zsh 模块也很重要，因为其中一些模块（如'zsh/system'、'zsh/mapfile'和 'zsh/files'）可以绕过大部分限制。

---

5 文件

5.1 启动/关闭文件

命令首先从 /etc/zshenv中读取；该选项不可覆盖。 RCS 和 GLOBAL_RCS 选项可修改后续行为；前者影响所有启动文件，后者仅影响全局启动文件（此处显示的文件路径以 / 开头）。 如果在任何时候未设置**[译注:”未设置“在很多时候也是”取消设置“的意思，这时显然是”取消设置“的意思]**其中一个选项，则不会读取相应类型的后续启动文件。 $ZDOTDIR 中的文件也有可能重新启用 GLOBAL_RCS。RCS 和 GLOBAL_RCS 均为默认设置。

然后从 $ZDOTDIR/.zshenv 中读取命令。 如果 shell 是登录 shell，命令将从 /etc/zprofile 读取，然后再从 $ZDOTDIR/.zprofile 读取。 然后，如果 shell 是交互式的，命令将从 /etc/zshrc 读取，然后再从 $ZDOTDIR/.zshrc 读取。 最后，如果 shell 是登录 shell，则会读取 /etc/zlogin 和 $ZDOTDIR/.zlogin。

当登录 shell 退出时，将读取 $ZDOTDIR/.zlogout 和 /etc/zlogout 文件。 这可以通过 exit 或 logout 命令显式退出，也可以通过从终端读取文件结尾隐式退出。 不过，如果由于 exec 操作了另一个进程而导致 shell 终止，则不会读取注销文件。 这些文件也会受到 RCS 和 GLOBAL_RCS 选项的影响。 另外，请注意 RCS 选项会影响历史文件的保存，也就是说，如果 shell 退出时 RCS 未设置，则不会保存任何历史文件。

如果 ZDOTDIR 未设置，则使用 HOME 代替。 上面列出的 /etc 中的文件可能在其他目录下，具体取决于安装情况。

由于 /etc/zshenv 会在 zsh 的所有实例中运行，因此必须尽可能地缩小它的规模。 特别要注意的是，最好将不需要在每个 shell 中运行的代码放在'if [[-o rcs]]; then ...'形式的测试后面，这样当使用'-f'选项调用 zsh 时，它就不会被执行。

5.2 文件

$ZDOTDIR/.zshenv
$ZDOTDIR/.zprofile
$ZDOTDIR/.zshrc
$ZDOTDIR/.zlogin
$ZDOTDIR/.zlogout
${TMPPREFIX}* (默认是 /tmp/zsh*)
/etc/zshenv
/etc/zprofile
/etc/zshrc
/etc/zlogin
/etc/zlogout（根据安装情况而定，/etc 为默认设置）

这些文件中的任何一个都可以使用 zcompile 内置命令（Shell 内置命令）进行预编译。如果存在编译后的文件（以原始文件命名，外加 .zwc 扩展名），且该文件比原始文件新，则将使用编译后的文件。

---

6 Shell 语法

6.1 简单命令和管道

一条 **简单命令** 是一连串可选的参数赋值，后面是空白分隔的词语，中间穿插着可选的重定向。 关于赋值的说明，请参阅 参数 的开头。

第一个单词是要执行的命令，其余单词（如果有）是该命令的参数。 如果给出了命令名，参数赋值将在执行命令时修改命令的环境。 简单命令的值是其退出状态，如果由信号终止，则是 128 加上信号编号。 例如

```
echo foo
```

是一个带有参数的简单命令。

一条 **管道** 既可以是一条简单的命令，也可以是由两条或多条简单命令组成的序列，其中每条命令之间用'|' 或 '|&'分隔。 如果命令之间用'|'分隔，则第一条命令的标准输出与下一条命令的标准输入相连。 '|&'是'2>&1 |'的简写，它将命令的标准输出和标准错误连接到下一条命令的标准输入。 管道的值是最后一条命令的值，除非管道前面有'!'，在这种情况下，管道的值是最后一条命令值的逻辑倒数。 例如

```
echo foo | sed 's/foo/bar/'
```

是一条管道，第一条命令的输出（'foo' 加上换行符）将传递到第二条命令的输入。

如果管道前面有‘coproc’，则它将作为协进程执行；它与父 shell 之间将建立双向管道。shell 可以通过 ‘>&p’和‘<&p’重定向操作符或‘print -p’和‘read -p’读取或写入协进程。一条管道前面不能同时出现 ‘coproc’ 和 ‘!’。 如果作业控制处于激活状态，则协程除了输入和输出外，可以作为普通后台作业处理。

一个 ***sublist*** 既可以是一条管道，也可以是由两条或多条管道组成的序列，中间用‘&&’或‘||’隔开。 如果两条管道由‘&&’分隔，则只有在第一条管道成功（返回零状态）后，才会执行第二条管道。 如果两条管道之间用‘||’隔开，则只有在第一条管道失败（返回非零状态）时，才会执行第二条管道。 这两个操作符具有相同的优先级，并且都是左关联。 子列表的值是最后执行的管道的值。 例如

```
dmesg | grep panic && print yes
```

是一个由两条管道组成的子列表（sublist），第二条管道只是一条简单的命令，只有当 grep 命令返回零状态时才会执行。 如果 grep 返回非零状态，子列表的值就是这个非零返回状态，否则就是 print 返回的状态（几乎肯定是零）。

一个 ***list*** 是由 0 个或多个子列表组成的序列，其中每个子列表都以 ‘;’、‘&’、‘&|’、‘&!’ 或换行符结束。 当列表作为复合命令出现在‘(...)’ 或‘{...}’中时，可以选择省略列表中最后一个子列表的结束符。 当子列表以 ‘;’ 或换行结束时，shell 会等待该子列表结束后再执行下一个子列表。 如果子列表以‘&’, ‘&|’, 或 ‘&!’结束，shell 会在后台执行其中的最后一条管道，而不会等待它结束（注意这与其他 shell 在后台执行整个子列表的做法不同）。 后台管道的返回状态为 0。

更广义地说，list可以看作是任何 shell 命令的集合，包括下面的复杂命令；在后面的描述中，只要出现 ‘list’ 一词，就意味着这一点。 例如，shell 函数中的命令就是一种特殊的列表(list)。

6.2 前置命令修饰符

一条简单命令的前面可以加上 **前置命令修饰符**，它将改变命令的解释方式。 除了 nocorrect 是保留字外，这些修改器都是 shell 内置命令。

-

执行命令时，会在 argv[0] 字符串前加上‘-’。

builtin

命令字被视为内置命令的名称，而不是 shell 函数或外部命令的名称。

command [-pvV]

命令字将被视为外部命令的名称，而不是 shell 函数或内置程序的名称。 如果设置了 POSIX_BUILTINS 选项，内置程序也会被执行，但它们的某些特殊属性会被抑制。 -p 标志会导致搜索默认路径，而不是 $path 中的路径。使用 -v 标志时，command 类似于 whence；使用 -V 时，它等同于 whence -v。

exec [-cl] [-a *argv0*]

以下命令连同参数将代替当前进程运行，而不是作为子进程运行。 shell 不会分叉，并被替换。 shell 不会调用 TRAPEXIT，也不会引入 zlogout 文件。 提供这些选项是为了与其他 shell 兼容。

-c 选项会清除环境。

-l 选项等同于 - 前置命令修饰符，将替换命令视为登录 shell；执行命令时，会在 argv[0] 字符串前加上 -。 如果与 -a 选项一起使用，则该标记无效。

-a 选项用于明确指定替换命令将使用的 argv[0] 字符串（进程本身看到的命令名称），直接等同于为 ARGV0 环境变量设置值。

nocorrect

不对任何单词进行拼写校正。 它必须出现在任何其他前置命令修饰符之前，因为它会在进行任何解析之前立即被解释。 在非交互式 shell 中没有作用。

noglob

不对任何单词执行文件名生成（通配）。

6.3 复杂命令

zsh 中的 **复杂命令**是以下命令之一：

if *list* then *list* [elif *list* then *list*] ... [else *list*] fi

执行if *list*，如果返回零退出状态，则执行 then *list*。 否则，执行 elif *list*，如果状态为零，则执行 then *list* 。 如果每个 elif *list*返回非零状态，则执行 else *list* 。

for *name* ... [in *word* ...] *term* do *list* done

展开 *word* 列表，依次将参数 *name* 设置为其中的每一个，每次都执行 *list*。 如果省略'in *word*'，则将使用位置参数而不是 *word*。[译注:for.示例1]

term 由一个或多个换行符或；组成，用于终止 *word*，当省略 'in *word*' 时是可选的。

在 *word* 之前可以出现多个参数 *name*。 如果给出 *N* 个 *name*，那么在循环的每次执行中，下面每 *N* 个 *words* 都会分配给相应的参数。 如果 *name* 个数多于剩余的 *word*，剩余的参数将被设置为空字符串。 [译注:for.示例2]当没有剩余的 *word* 可以分配给第一个 *name* 时，循环执行结束。 in 作为 *name* 列表中的第一个 *name* 出现时，才能作为 *name* 使用，否则它将被视为 *name* 列表的结束符。[译注:for.示例3]

for (([*expr1*] ; [*expr2*] ; [*expr3*])) do *list* done

首先对算术表达式 *expr1* 进行求值（参见 算术求值）。 算术表达式 *expr2* 会被反复求值，直到求值为零，如果不为零，则执行 *list* 并求值算术表达式 *expr3*。 如果省略了任何表达式，则该表达式的值为 1。

while *list* do *list* done

只要 while *list* 返回的退出状态为零，就执行 do *list* 。

until *list* do *list* done

只要 until *list* 返回非零退出状态，就执行 do *list* 。

repeat *word* do *list* done

word 将被展开并视为算术表达式，其值必须为数值 *n*。 然后 *list* 会被执行 *n* 次。

当 shell 以模拟其他 shell 的模式启动时，repeat 语法默认为禁用。 可以使用‘enable -r repeat’命令启用该语法。

case *word* in [[(] *pattern* [| *pattern*] ...) *list* (;;|;&|;|)] ... esac

执行与第一个匹配 *word* 的 *pattern* 相关的 *list*（如果有）。 模式的形式与文件名生成使用的模式相同。 请参阅 文件名生成。

还需要注意的是，除非设置了 SH_GLOB 选项，否则 shell 会将整个模式视为括号内的模式组对待，尽管在括号和竖线附近可能会出现空白，但这些空白会从模式中删除。 空格可能出现在模式的其他地方，但不会被删除。 如果设置了 SH_GLOB 选项，那么开头的括号明确地视为case语法的一部分，那么表达式将被解析为单独的单词，这些单词将被视为严格的替代词（与其他 shell 相同）。[译注:这里只是shell处理时的内部逻辑，在这里使用时不必关心]

如果执行的 *list* 的结束符是 ;& 而不是 ;;，则也会执行下面的列表。 除非到达 esac，否则将执行下面列表的终止符 ;;、;& 或 ;| 的规则。

如果执行的 *list* 以 ;| 结尾，则 shell 会继续扫描 *pattern* 寻找下一个匹配项，执行相应的 *list* 并应用相应的结尾 ;;、;& 或 ;| 规则。 请注意，*word* 不会重新扩展；所有适用的 *pattern* 都将使用相同的 *word* 进行测试。

select *name* [in *word* ... *term*] do *list* done

其中 *term* 是一个或多个换行符，或 ; ，用以终止 *words*. 打印一组 *word*，每个前面都有一个数字。 如果省略 `in` *word* 则使用位置参数。 如果 shell 是交互式的，并且行编辑器处于激活状态，则会打印 `PROMPT3` 提示符，然后从行编辑器读取一行，否则从标准输入读取一行。 如果该行包含所列 *word* 的编号，那么参数 *name* 将被设置为与该编号对应的 *word*。 如果该行为空，则再次打印选择列表。 否则，参数 *name* 的值将被设置为空。 从标准输入读取的行内容将保存在参数 `REPLY` 中。 *list* 会在每次选择时执行，直到遇到中断或文件结束。[译注:语法.select]

(*list*)

在子shell中执行 *list*。 在执行 *list* 时，由 `trap` 内置函数设置的陷阱将被重置为默认值；但如果设置了 `POSIXTRAPS` 选项，被忽略的信号将继续被忽略。

{ *list* }

执行 *list*.

{ *try-list* } `always` { *always-list* }

首先执行 *try-list*。 无论在 *try-list* 中遇到任何错误、`break` 或 `continue` 命令，都要执行 *always-list*。 换句话说，任何错误、`break` 或 `continue`命令都会以正常方式处理，就好像 *always-list* 不存在一样。 这两段代码被称为 'try 块'和 'always 块'。

在 `always` 之后可以出现可选的换行符或分号，但要注意，换行符或分号 **不可以** 出现在前面的收尾括号和 `always` 之间。

在这个语境中，'错误' 是指诸如语法错误之类的条件，会导致shell 中止当前函数、脚本或列表的执行。shell 解析代码时遇到的语法错误不会导致 *always-list* 被执行。 例如，在 *try-list* 中构造错误的 `if` 块会导致 shell 在解析过程中中止，因此 *always-list* 不会被执行，而错误的替换（如 `${*foo*}`）会导致运行时错误，之后 *always-list* 将被执行。

错误条件可以通过特殊的整数变量 `TRY_BLOCK_ERROR` 进行测试和重置。 在 *always-list* 之外，该变量的值无关紧要，但会被初始化为 `-1`。 在 *always-list* 中，如果 *try-list* 发生错误，则值为 `1`，否则为 `0`。 如果 `TRY_BLOCK_ERROR` 在 *always-list* 中被设置为 `0`，由 *try-list* 引起的错误将被重置，shell 执行将在 *always-list* 结束后继续正常进行。 在 *try-list*期间更改值并无用处（除非它构成了外层 `always` 代码块的一部分）。[译注:语法.always]

无论 `TRY_BLOCK_ERROR` 如何设置，在 *always-list* 结束后，正常的 shell 状态 `$?` 就是 *try-list* 返回的值。 如果出现错误，即使 `TRY_BLOCK_ERROR` 被设置为零，该值也不会为零。

下面的代码会执行给定的代码，并忽略它引起的任何错误。 这与通常在子 shell 中执行代码以保护代码的做法不同。

```
{
    # code which may cause an error
  } always {
    # This code is executed regardless of the error.
    (( TRY_BLOCK_ERROR = 0 ))
}
# The error condition has been reset.
```

当 try 代码块出现在任何函数之外时，在 *try-list* 中遇到 return 或 exit 不会 **不会** 导致 *always-list* 的执行。 相反，shell 会在执行 EXIT 陷阱后立即退出。 否则，与 break 和 continue 一样，在 *try-list* 中执行 return 命令也会导致 *always-list* 的执行。

function [-T] *word* ... [()] [*term*] { *list* }
word ... () [*term*] { *list* }
word ... () [*term*] *command*

其中 *term* 是一个或多个换行符或 ;。 定义一个函数，该函数由 *word* 中的任意一个引用。 通常只提供一个 *word*；多个 *word* 通常只在设置陷阱时有用。 函数的主体是 { 和 } 之间的 *list*。 参见 函数。[译注:语法.function]

function 的选项具有以下含义:

-T

启用对该函数的跟踪，就像使用 functions -T 一样。 请参阅 Shell 内置命令中 typeset 内置命令的 -f 选项文档。

如果为了与其他 shell 兼容而设置了 SH_GLOB 选项，那么当只有一个 *word* 时，左右括号之间可能会出现空白；否则，在这种情况下，括号将被视为形成一个 globbing 模式。[译注:语法.function.shglob]

在上述任何一种形式中，重定向都可能出现在函数体之外，例如

```
func() { ... } 2>&1
```

重定向与函数一起存储，并在执行函数时应用。 重定向中的任何变量都会在函数执行时展开，但不在函数范围内。

time [*pipeline*]

执行 *pipeline*，并以 TIMEFMT 参数指定的形式在标准错误中报告计时统计数据。 如果省略 *pipeline*，则打印 shell 进程及其子进程的统计信息。

[[*exp*]]

计算条件表达式 *exp*，如果为真，则返回零退出状态。 有关 *exp* 的描述，请参阅 条件表达式。

6.4 复杂命令的替代形式

zsh 的许多复杂命令都有替代形式。 这些命令都是非标准的，即使是经验丰富的 shell 程序员也很难看出它们的区别；在任何需要考虑 shell 代码可移植性的地方，都不应该使用它们。

只有当 *sublist* 的形式为'{ *list* }'时，或设置了 SHORT_LOOPS 选项[译注:语法.短循环.shortloops]，下面的简短版本才有效。 对于 `if`、`while` 和 `until` 命令，在这两种情况下，循环的测试部分也必须适当分隔，例如使用 '[[...]]' 或 '((...))'，否则将无法识别测试的结束。 对于 `for`、`repeat`、`case` 和 `select`，不需要为参数设置特殊形式，但其他条件（*sublist* 的特殊形式或使用 SHORT_LOOPS 选项）仍然适用。
SHORT_REPEAT 选项只能用于 `repeat` 命令的简短版本。

`if` *list* { *list* } [`elif` *list* { *list* }] ... [`else` { *list* }]

`if` 的另一种形式。 规则的意思是

```
if [[ -o ignorebraces ]] {
  print yes
}
```

是有效的，但

```
if true {  # Does not work!
  print yes
}
```

则**不会**，因为测试（条件）没有适当的分隔（符）。

`if` *list sublist*

另一种 `if` 的简写形式。 对 *list* 形式的限制与前一种形式相同。

`for` *name* ... (*word* ...) *sublist*

`for` 的简写。

`for` *name* ... [`in` *word* ...] *term sublist*

其中 *term* 是至少一个换行符或 ;。 `for` 的另一种简写形式。

`for` (([*expr1*] ; [*expr2*] ; [*expr3*])) *sublist*

算术 `for` 命令的简写。

`foreach` *name* ... (*word* ...) *list* `end`

for 的另一种形式。

`while` *list* { *list* }

while 的另一种形式。 请注意上述对 *list* 形式的限制。

`until` *list* { *list* }

until 的另一种形式。 请注意上述对 *list* 形式的限制。

`repeat` *word sublist*

这是 repeat 的简写。

`case` *word* { [[(] *pattern* [| *pattern*] ...) *list* (;;|;&|;|)] ... }

case 的另一种形式。

`select` *name* [in *word* ... *term*] *sublist*

其中 *term* 至少是一个换行符或 ;。 select 的简写。

`function` *word* ... [()] [*term*] *sublist*

这是 function 的简写。

6.5 保留字

除非加引号或使用 `disable -r` 禁用，否则下列单词作为命令的第一个单词使用时将被视为保留字：

```
do done esac then elif else fi for case if while function repeat
time until select coproc nocorrect foreach end ! [[ { } declare
export float integer local readonly typeset
```

此外，如果 IGNORE_BRACES 选项和 IGNORE_CLOSE_BRACES 选项均未设置，‘}’可以在任何位置被识别。

6.6 错误

某些错误会被 shell 视为致命错误：在交互式 shell 中，它们会导致控制返回命令行，而在非交互式 shell 中，它们会导致 shell 终止。 在旧版本的 zsh 中，运行脚本的非交互式 shell 不会完全终止，而是会在从脚本读取下一条命令时继续执行，跳过任何函数或 shell 结构（如循环或条件）的剩余部分；这种有点不合逻辑的行为可以通过设置选项 CONTINUE_ON_ERROR 来恢复。

在非交互式 shell 中发现的致命错误包括:

- 无法解析调用 shell 时传递的 shell 选项
- 无法使用 `set` 内置命令更改选项
- 各种解析错误，包括无法解析数学表达式
- 使用 `typeset`、`local`、`declare`、`export`、`integer`、`float` 设置或修改变量行为失败
- 执行位置不正确的循环控制结构（`continue`、`break`）。
- 尝试使用正则表达式，但没有可用的正则表达式模块
- 设置 `RESTRICTED` 选项时禁止的操作
- 未能创建管道所需的管道
- 未能建立一个multio
- 无法自动加载已声明的 shell 功能所需的模块
- 创建命令或进程替换时出现错误
- glob 限定符中的语法错误
- 选项 `BAD_PATTERN` 无法捕捉文件生成错误
- case语句中用于匹配的所有不良模式
- 不是由 `NO_MATCH` 或类似选项造成的文件生成失败
- 所有文件生成错误，其中模式被用于创建multio。
- shell 检测到内存错误
- shell 变量的下标无效
- 尝试为只读变量赋值
- 变量逻辑错误，如赋值给错误类型
- 使用无效的变量名
- 变量替换语法错误
- 无法转换 `$'...'` 表达式中的字符

如果设置了 `POSIX_BUILTINS` 选项，根据 POSIX 标准的规定，更多与 shell 内置命令相关的错误将被视为致命错误。

6.7 注释

在非交互式 shell 或设置了 `INTERACTIVE_COMMENTS` 选项的交互式 shell 中，以 `histchars` 参数（默认为‘#’）的第三个字符开头的单词会导致该单词以及换行符之前的所有后续字符被忽略。

6.8 别名

shell 输入中每一个符合条件的 ***word*** 都会被检查，看是否为其定义了别名。 如果有的话，如果它处于命令位置（如果它可能是简单命令的第一个单词），或者如果别名是全局

的，它就会被别名文本替换。 如果替换文本以空格结束，则 shell 输入中的下一个单词总是可以用于别名扩展。

除非设置了 ALIAS_FUNC_DEF 选项，否则 sh 兼容函数定义语法‘*word* ( ) ...’ 中的函数名 *word*, 如果是由别名扩展产生的单词，则属于错误。[译注:语法.别名]

使用 alias 内置命令可定义别名；使用 -g 选项可定义全局别名。

一个 ***word*** 的定义是:

- 任何普通字符串或 glob 模式
- 任何带引号的字符串，使用任何引号方法（注意，引号必须是别名定义的一部分，才符合条件）
- 任何参数引用或命令替换
- 上述任何系列，连接时中间不留空白或其他符号
- 任何保留字（case、do、else 等）
- 在全局别名定义下，任何命令分隔符、重定向操作符，以及不用作 glob 模式一部分时的‘(’或‘)’

别名扩展是在历史扩展之后，其它扩展之前， 在 shell 输入上进行的。 因此，如果为单词 foo 定义了别名，可以通过引用(转义)单词的一部分来避免别名扩展，例如 \foo。 任何形式的引号(转义)都可以使用，尽管没有什么可以阻止为 \foo 这样的引号(转义)形式定义别名。

特别要注意的是，使用 unalias 删除全局别名时必须使用引号:

```
% alias -g foo=bar
% unalias foo
unalias: no such hash table element: bar
% unalias \foo
%
```

设置 POSIX_ALIASES 时，只有未加引号的普通字符串才有资格使用别名。 alias 内置命令不会拒绝不符合条件的别名，但不会对其进行扩展。[译注:语法.alias.posixaliases]

补全会移除首部反斜线后的非特殊字符，来与补全一起使用，以单引号开头的引用可能更方便，例如 'foo；补全会自动添加尾部的单引号。

6.8.1 别名的问题

虽然别名的使用方式可以超越正常的 shell 语法，但并非所有非空格字符串都可以用作别名。

任何未被上面作为单词列出的字符集都不是单词，因此，无论如何定义（即通过内置命令或 zsh/parameter 模块 中描述的特殊参数 aliases），都不会尝试将其扩展为别名。不

过，正如上文 POSIX_ALIASES 所述，在创建别名时，shell 不会尝试推断字符串是否与单词相对应。

例如，在命令行开头包含 = 的表达式是赋值，不能扩展为别名；单独的 = 不是赋值，只能使用参数设置为别名，否则 = 将被视为内置命令语法的一部分。

目前还无法为‘((’（引入算术表达式的标记）创建别名，因为在解析完整语句之前，无法将其与两个连续的引入嵌套子shell的‘(’标记区分开来。 另外，如果 && 这样的分隔符被别名，\&& 就会变成两个标记 \& 和 &，而每个标记都可能被单独别名。 类似的还有 \<<, \>| 等。

下面的代码说明了一个常见的别名问题：

```
alias echobar='echo bar'; echobar
```

这将打印一条信息：找不到 echobar 命令。 出现这种情况是因为别名是在读入代码时展开的；整行代码是一次性读入的，因此在执行 echobar 时，来不及展开新定义的别名。在 shell 脚本、函数和使用 ‘source’或‘.’执行的代码中，这通常是一个问题。 因此，建议在非交互代码中使用函数而不是别名。

6.9 引用

在一个字符前加上‘\’，该字符就可以被**引用**（即使他表示自己）。 后跟换行符的 ‘\’ 将被忽略。

由 ‘$’’ 和 ‘’’括起来的字符串的处理方式与 print 内置函数的字符串参数相同，生成的字符串被认为是完全加引号的。 通过使用转义符 ‘\’’，可以在字符串中包含一个字面的‘’’字符。

在一对单引号（’’）之间括起来的所有字符，如果前面没有‘$’，都会被加上引号。 单引号不能出现在单引号内，除非设置了 RC_QUOTES 选项，在这种情况下，一对单引号会变成一个单引号。 例如

```
print ''''
```

如果未设置 RC_QUOTES，则除了换行符外不会输出任何内容；如果设置了 RC_QUOTES，则会输出一个单引号。

在双引号（""）内，会出现参数和命令替换， ‘\’ 会引用 ‘\’, ‘`’, ‘"’, ‘$’和 $histchars 的第一个字符（默认为 ‘!’）。

---

7 重定向

如果命令后跟 &，且作业控制未激活，则该命令的默认标准输入为空文件 /dev/null。否则，命令的执行环境将包含调用的 shell 的文件描述符，并根据输入/输出规范进行修改。

以下内容可以出现在简单命令的任何位置，也可以出现在复杂命令的前面或后面。 在使用 *word* 或 *digit* 之前，会进行扩展，以下情况除外。 如果 *word* 的替换结果产生一个以上的文件名，则依次对每个独立的文件名进行重定向。

< *word*

打开文件 *word* 作为标准输入进行读取。 如果文件不存在，用这种方式打开文件是错误的。

<> *word*

打开 *word* 文件，作为标准输入进行读写。 如果文件不存在，则将创建该文件。

> *word*

打开 *word* 文件，将其作为标准输出写入。 如果文件不存在，则创建该文件。 如果文件存在，且 CLOBBER 选项未设置，则会导致错误；否则，文件将被截断为零长度。

>| *word*
>! *word*

与 > 相同，但如果文件存在，无论 CLOBBER 如何，文件都会被截断为零长度。

>> *word*

打开 *word* 文件，作为标准输出以追加模式写入。 如果文件不存在，且 CLOBBER 和 APPEND_CREATE 选项均未设置，则会导致错误；否则将创建文件。

>>| *word*
>>! *word*

与 >> 相同，但如果文件不存在，则会创建文件，与 CLOBBER 和 APPEND_CREATE 无关。

<<[-] *word*

shell 输入将读取至与 *word* 相同的行，或至文件末尾。 不会对 *word* 进行参数扩展、命令替换或文件名生成。 生成的文件称为 ***here-document***，将成为标准输入。

如果 *word* 中的任何字符被单引号、双引号或‘\’引用，则不会对文件中的字符进行解释。 否则，将进行参数替换和命令替换，‘\’后面是换行，则删除‘\’，并且必须使用 ‘\’ 来引用‘\’, ‘$’, ‘`’和 *word* 的第一个字符。

请注意，*word* 本身不进行 shell 扩充。 *word* 中的反引号不会产生通常的效果；相反，它们的行为与双引号类似，只是反引号本身不会被改变。 (提供这些信息只是为了完整，并不建议使用反引号）。 $'...' 形式的引用具有将反斜线引用扩展为特殊字符的标准效果。

如果使用 <<- ，则会从 *word* 和文档中删除所有前导制表符。

<<< *word*

对 *word* 执行 shell 扩展，并将结果传入标准输入。 这被称为 ***here-string***。 对比上述 here-documents 中 *word* 的使用，*word* 并未进行 shell 扩充。 结果后面会有一个换行符。

<& *number*
>& *number*

从文件描述符 *number* 复制标准输入/输出（参见 dup2(2)）。

<& -
>& -

关闭标准输入/输出。

<& p
>& p

协进程的输入/输出被移至标准输入/输出。

>& *word*
&> *word*

(除 '>& *word*' 与上述语法之一相匹配；'&>' 总是可用于避免这种歧义）。 以 '> *word*' 的方式重定向标准输出和标准错误（文件描述符 2）。 请注意，这与存在 multios 时， '> *word* 2>&1' 的效果**不同**（见下文）。

>&| *word*
>&! *word*
&>| *word*
&>! *word*

以'>| *word*'的方式重定向标准输出和标准错误（文件描述符 2）。

>>& *word*
&>> *word*

以 '>> *word*' 的方式重定向标准输出和标准错误（文件描述符 2）。

>>&| *word*

>>&! *word*
&>>| *word*
&>>! *word*

以'>>| *word*'的方式重定向标准输出和标准错误（文件描述符 2）。

如果上述任何一项前面有一个数字，那么所指的文件描述符就是该数字指定的文件描述符，而不是默认的 0 或 1。 指定重定向的顺序很重要。 shell 在计算每个重定向时，都会根据（**文件描述符**, **文件**）关联进行计算。 例如

... 1>*fname* 2>&1

首先将文件描述符 1 与文件 *fname* 关联。 然后将文件描述符 2 和与文件描述符 1 相关联的文件（即 *fname*）关联起来。 如果重定向的顺序颠倒，文件描述符 2 将与终端关联（假设文件描述符 1 已经关联），然后文件描述符 1 将与文件 *fname* 关联。[译注:重定向.顺序]

简单命令和管道 中描述的 '|&' 命令分隔符是 '2>&1 |'的简写。

各种形式的进程替换，即输入的'<(*list*)'和 '=(*list*)' 以及输出的'>(*list*)'，经常与重定向一起使用。 例如，如果输出重定向中的 *word* 的形式为 '>(*list*)'，那么输出将被管道输送到 *list* 所代表的命令。 参见 进程替换。

7.1 使用参数打开文件描述符

在 shell 解析命令参数时，如果未设置 shell 选项 IGNORE_BRACES，则允许使用另一种形式的重定向：用大括号括起来的有效 shell 标识符代替运算符前的数字。 shell 将打开一个新的文件描述符，该文件描述符保证至少为 10，并将标识符命名的参数设置为打开的文件描述符。 在括号和重定向字符之间不允许有空白。 例如

... {myfd}>&1

这将打开一个与文件描述符 1 重复的新文件描述符，并将参数 myfd 设置为文件描述符的编号（至少为 10）。 可以使用 >&$myfd 语法写入新的文件描述符。 该文件描述符在子shell和分叉（forked）的外部可执行文件中保持打开状态。

语法 {*varid*}>&-，例如 {myfd}>&-，可用于关闭以这种方式打开的文件描述符。 需要注意的是，在这种情况下，*varid* 给出的参数必须事先设置为文件描述符。

当参数为只读时，以这种方式打开或关闭文件描述符是错误的。 不过，如果 *param* 为只读参数，则使用 <&$*param* 或 >&$*param* 读取或写入文件描述符不会出错。[译注:重定向.文件描述符.只读参数]

如果未设置选项 CLOBBER，如果参数是一个打开的文件描述符的（通过本机制已经分配的），通过该参数打开文件描述符是错误的。 在使用该参数分配文件描述符前取消参数设置，就不会出错。

请注意，这种机制只是分配或关闭文件描述符，并不执行任何重定向操作。 通常，在将文件描述符作为 exec 的参数使用之前，分配文件描述符会比较方便。 在任何情况下，该语法都不能用于复杂的命令，例如带括号的子shell或循环，在这种情况下，开头的括号会被解释为要在当前 shell 中执行的命令列表的一部分。

下面显示了分配、使用和关闭文件描述符的典型顺序：

```
integer myfd
exec {myfd}>~/logs/mylogfile.txt
print This is a log message. >&$myfd
exec {myfd}>&-
```

请注意，表达式 >&$myfd 中变量的扩展是在重定向打开时进行的。 这是在扩展命令参数和处理命令行左侧的重定向之后。

7.2 Multios

如果用户尝试打开一个文件描述符进行多次写入，shell 会将该文件描述符作为管道打开到一个进程，该进程会将其输入复制到所有指定的输出，类似于 *tee*，前提是设置了 MULTIOS（默认设置）选项。 因此

```
date >foo >bar
```

将日期写入两个文件，分别命名为‘foo’和‘bar’。 请注意，管道是一种隐式重定向；因此

```
date >foo | cat
```

将日期写入文件‘foo’，并将其导入 cat。

请注意，shell 会立即打开所有要在 multio 进程中使用的文件，而不是在即将写入这些文件时才打开。

还要注意的是，重定向总是按顺序展开的。 这与 MULTIOS 选项的设置无关，但如果该选项生效，则会产生额外的后果。例如，表达式 >&1 的含义会在前一次重定向后发生变化：

```
date >&1 >output
```

在上面的例子中，>&1 指的是行首的标准输出；结果与 tee 命令类似。 不过，请注意

```
date >output >&1
```

由于重定向是按顺序计算的，当遇到 >&1 时，标准输出会被设置为 output 文件，因此另一份输出会被发送到该文件。 这可能不是我们想要的结果。

如果设置了 MULTIOS 选项，重定向运算符后的字也会执行文件名生成（通配）。 因此

```
: > *
```

将截断当前目录下的所有文件，前提是至少有一个文件。 (如果没有 MULTIOS 选项，它将创建一个名为'*'的空文件）。 类似地，可以执行

```
echo exit 0 >> *.sh
```

如果用户尝试打开一个文件描述符进行多次读取，shell 会将该文件描述符作为管道打开到一个进程，该进程会按照指定的顺序将所有指定的输入复制到输出，前提是设置了 MULTIOS 选项。 需要注意的是，每个文件都是立即打开的，而不是在即将读取时打开：这种行为与 cat 不同，因此如果需要严格的标准行为，应使用 cat 代替。

因此

```
sort <foo <fubar
```

甚至

```
sort <f{oo,ubar}
```

等价于 'cat foo fubar | sort'.

重定向参数的扩展发生在重定向打开时，也就是上述 >&$myfd 中变量扩展时。

请注意，管道是一种隐式重定向，因此

```
cat bar | sort <foo
```

等价于 'cat bar foo | sort'（注意输入的顺序）。

如果 MULTIOS 选项为 **未设置**，则每次重定向都会替换之前针对该文件描述符的重定向。不过，所有重定向到的文件实际上都会被打开，因此

```
echo Hello > bar > baz
```

当 MULTIOS 未设置时，将截断'bar'，并将 'Hello' 写入'baz'。

当输出multio连接到外部程序时会出现一个问题。 一个简单的例子就能说明问题：

```
cat file >file1 >file2
cat file1 file2
```

在这里，第二个 'cat'可能不会显示 file1 和 file2 的全部内容（即 file 的原始内容重复了两次）。

原因是multios是在 cat 进程从父 shell 分支出来后才生成的，因此父 shell 不会等待 multios完成数据写入。 这意味着所示命令可能会在 file1 和 file2 完全写完之前退出。 作为一种解决方法，可以将 cat 进程作为当前 shell 中作业的一部分运行：

```
{ cat file } >file >file2
```

在此，{…} 作业将暂停，等待两个文件都被写入。

7.3 没有命令的重定向

当一条简单的命令由一个或多个重定向操作符和零个或多个参数赋值组成，但没有命令名称时，zsh 会以几种方式执行。

如果未设置参数 NULLCMD，或设置了选项 CSH_NULLCMD，则会导致错误。 这是 *csh* 的行为，在模拟 *csh* 时，CSH_NULLCMD 默认会被设置。

如果设置了选项 SH_NULLCMD，内置的":"将作为命令插入，并带有给定的重定向。 这是模拟 *sh* 或 *ksh* 时的默认值。

否则，如果设置了参数 NULLCMD，其值将被用作带有给定重定向的命令。 如果同时设置了 NULLCMD 和 READNULLCMD，那么当重定向为输入时，将使用后者的值，而不是前者的值。 NULLCMD 的默认值是 'cat'，而 READNULLCMD 的默认值是'more'。因此

```
< file
```

将 file 的内容显示在标准输出上，如果是终端，还将分页显示。 NULLCMD 和 READNULLCMD 可指 shell 函数。

---

8 命令执行

如果命令名不包含斜线，shell 会尝试查找它。 如果该命令名下存在 shell 函数，则按照 函数 中的说明调用该函数。 如果存在以该命令名命名的 shell 内置命令，则调用该内置命令。

否则，shell 会搜索 $path 的每个元素，查找包含该名称可执行文件的目录。

如果执行失败：则打印错误信息，并返回以下值之一。

127

搜索不成功。 错误信息为 'command not found: *cmd*'。

126

可执行文件的权限不足、是目录或特殊文件、不是脚本和格式未被操作系统识别。 具体条件和错误信息取决于操作系统；请参阅 execve(2)。

如果执行失败是因为文件不是可执行格式，且文件不是目录，则假定该文件是 shell 脚本。 `/bin/sh` 将被生成并执行。 如果程序是以'#!'开头的文件，第一行的其余部分将指定程序的解释器。 在内核不处理这种可执行文件格式的操作系统上，shell 将执行指定的解释器。

如果没有找到外部命令，但存在函数 `command_not_found_handler`，shell 就会使用所有命令行参数执行该函数。 函数的返回状态将成为命令的状态。 请注意，处理程序是在执行外部命令的子 shell 中执行的，因此目录、shell 参数等的更改不会影响主 shell。

---

9 函数

Shell 函数是用 `function` 保留字或特殊语法'*funcname* `()`'定义的。 Shell 函数在内部读取和存储。 读取函数时会解析别名。 函数像命令一样执行，参数作为位置参数传递。 (请参阅 命令执行）。

函数在与调用者相同的进程中执行，并与调用者共享所有文件和当前工作目录。 在函数内部设置的 `EXIT` 陷阱会在函数完成后在调用者的环境中执行。

`return` 内置命令用于从函数调用中返回。

可以使用 `functions` 内置命令列出函数标识符。 函数可以使用 `unfunction` 内置命令清除定义。

9.1 自动加载函数

可以使用 `autoload` 内置命令（或 '`functions -u`' 或 '`typeset -fu`'）将函数标记为 **未定义的** [译注:函数.自动加载.未定义的]。 这样的函数没有主体。 第一次执行函数时，shell 会使用 `fpath` 变量的元素搜索函数的定义。 因此，定义自动加载函数的典型顺序是

```
fpath=(~/myfuncs $fpath)
autoload myfunc1 myfunc2 ...
```

如果 `autoload` 内置命令或其等效命令的选项为 `-U`，读取过程中通常的别名扩展将被抑制[译注:函数.autoload.抑制别名扩展]。建议使用 zsh 发行版提供的函数。 请注意，对于使用 `zcompile` 内置命令预编译的函数，必须在创建 `.zwc` 文件时提供 `-U` 标记，因为相应的信息会编译到 `.zwc` 文件中。

对于 `fpath` 中的每个 *element*，shell 会查找三个可能的文件，其中最新的文件用于加载函数的定义：

element`.zwc`

使用 zcompile 内置命令创建的文件，预计将包含名为 *element* 的目录中所有函数的定义。 该文件的处理方式与包含函数文件的目录相同，并搜索函数的定义。 如果未找到定义，则按下面描述的其他两种可能性搜索定义。

如果 *element* 已经包含 .zwc 扩展名（即用户明确给出了扩展名），则 *element* 将搜索函数的定义，而不会将其与其他文件的时间进行比较；事实上，不需要有任何名为 *element* 而不带后缀的目录。 因此，在 fpath 中加入诸如 '/usr/local/funcs.zwc'这样的元素会加快搜索函数的速度，但缺点是在 shell 发现任何变化之前，必须明确地手工重新编译所包含的函数。

element/*function*.zwc

使用 zcompile 创建的文件，预计包含 *function* 的定义。 它也可能包含其他函数定义，但这些定义既不会被加载，也不会被执行；以这种方式找到的文件**只** 搜索 *function* 的定义。

element/*function*

（含）zsh 命令（的）文本文件，被视为 *function* 的定义。

总之，搜索的顺序是：首先，在 fpath 中的 **父** 目录中搜索-编译的目录或 fpath 目录中-的较新目录；其次，如果其中有多个目录包含所搜索函数的定义，则选择 fpath 中最左边的目录；第三，在一个目录中，使用编译函数或普通函数定义中的较新一个。

如果设置了 KSH_AUTOLOAD选项，或者文件只包含函数的简单定义，文件内容将被执行。 这通常会定义相关函数，但也可能执行初始化，初始化是在函数执行的上下文中执行的，因此可能会定义本地参数。 如果加载文件时，没有定义函数，则会出错。

否则，函数体（周围没有'*funcname*() {...}'）将被视为文件的完整内容。这种形式允许将文件直接用作可执行 shell 脚本。 如果对文件的处理导致函数被重新定义，函数本身不会被重新执行。要强制 shell 执行初始化，然后调用所定义的函数，文件除了包含初始化代码（执行后丢弃）外，还应包含完整的函数定义（保留用于后续函数调用），以及对 shell 函数的调用，包括最后的参数。

例如, 假设自动加载文件 func 包含

```
func() { print This is func; }
print func is initialized
```

那么，如果设置了 KSH_AUTOLOAD，'func; func' 将在第一次调用时产生两条信息，但在第二次及其后的调用中只产生'This is func' 信息。 如果未设置 KSH_AUTOLOAD，则第一次调用时将产生初始化信息，第二次及后续调用时将产生其他信息。[译注:函数.autoload.kshautoload]

通过在 shell 函数中使用 'autoload -X'，也可以创建一个未标记为自动加载的函数，但它可以通过搜索 fpath 加载自己的定义。 例如，以下是等价的:

```
myfunc() {
  autoload -X
}
myfunc args...
```

与

```
unfunction myfunc   # if myfunc was defined
autoload myfunc
myfunc args...
```

事实上，functions 命令会输出‘builtin autoload -X’作为自动加载函数的主体。这样做是为了

```
eval "$(functions)"
```

会产生一个合理的结果。 一个真正的自动加载函数可以通过正文中的注释‘# undefined’来识别，因为定义函数中的所有注释都会被丢弃。

要在不执行 myfunc 的情况下加载自动加载函数 myfunc 的定义，请使用

```
autoload +X myfunc
```

9.2 匿名函数

如果没有给出函数名称，则该函数为 "匿名 "函数，将被特殊处理。 可以使用两种形式的函数定义：一种是前面没有名称的‘()’，另一种是后面紧跟一个大括号的‘function’。函数在定义时立即执行，不会存储起来供将来使用。 函数名称设置为‘(anon)’。

函数的参数可以指定为定义函数的收尾括号后的单词，因此如果没有参数，则不会设置参数（$0 除外）。 这与其他函数的解析方式不同：普通函数定义后可能会出现某些关键字，如‘else’或‘fi’，这些关键字将被视为匿名函数的参数，因此需要换行或分号来强制解释关键字。

还要注意的是，任何外层脚本或函数的参数列表都是隐藏的（此时调用的任何其他函数也是如此）。

对匿名函数进行重定向的方式与对大括号中的当前shell结构进行重定向的方式相同。 匿名函数的主要用途是为局部变量提供作用域。 这在启动文件中尤其方便，因为这些文件不提供自己的局部变量作用域。

例如,

```
variable=outside
function {
  local variable=inside
  print "I am $variable with arguments $*"
```

```
} this and that
print "I am $variable"
```

输出如下:

```
I am inside with arguments this and that
I am outside
```

需要注意的是，如果函数定义的参数扩展为空值，例如‘name=; function $name { ... }’，则不会被视为匿名函数。 相反，它们被视为普通的函数定义，其中的定义会被默默丢弃。

9.3 特殊函数

特殊函数（如果已定义）对 shell 有特殊意义。

9.3.1 钩子函数

对于下面的函数，可以定义一个数组，该数组的名称与附加了‘_functions’的函数相同。 数组中的任何元素都将作为要执行的函数名称；它将在与基本函数相同的上下文、相同的参数和相同的 $? 初始值下执行。 例如，如果 $chpwd_functions 是一个包含‘mychpwd’, ‘chpwd_save_dirstack’值的数组，那么 shell 会尝试依次执行函数‘chpwd’, ‘mychpwd’和‘chpwd_save_dirstack’。 任何不存在的函数都会被静默忽略。通过此机制找到的函数在其他地方被称为 **钩子函数**。 任何函数中的错误都会导致后续函数无法运行。 请进一步注意，如果 precmd 钩子中出现错误，则紧随其后的 periodic 函数将不会运行（尽管它可能会在下一次机会中运行）。

chpwd

每当更改当前工作目录时执行。

periodic

如果设置了参数 PERIOD，该函数将每隔 $PERIOD 秒执行一次，就在提示符之前。需要注意的是，如果使用数组 periodic_functions 定义了多个函数，则整个函数集只适用一个周期，如果函数列表被更改，计划时间也不会重置。 因此，函数集总是一起被调用。

precmd

在每次（打印）提示符前执行。请注意，命令前函数不会因为命令行重绘而被重新执行，例如，在显示-任务退出通知时-就会发生这种情况。

preexec

在命令被读取后并即将执行时执行。 如果历史记录机制处于激活状态（无论该行是否已从历史记录缓冲区中丢弃），用户输入的字符串将作为第一个参数传递，否则就是空字符串。 将被执行的实际命令（包括扩展的别名）以两种不同的形式传递：第二个参数是单行、大小受限的命令版本（省略了函数体等内容）；第三个参数包含正在执行的完整文本。

zshaddhistory

在交互式读取历史行后，但在它执行前执行。 唯一的参数是完整的历史记录行（因此任何终止换行仍将存在）。

如果任何钩子函数返回状态 1（或除 2 之外的任何非零值，但未来版本的 shell 并不保证这一点），历史行将不会被保存，但它会在历史行中停留，直到下一行被执行，从而允许您立即重新使用或编辑它。

如果任何钩子函数返回状态 2，历史行将保存在内部历史列表中，但不会写入历史文件。 如果出现冲突，则取第一个非零状态值。

钩子函数可以调用 'fc -p ...' 来切换历史上下文，使历史记录保存在与全局 HISTFILE 参数不同的文件中。 这种情况会得到特殊处理：历史行处理完成后，历史上下文会自动恢复。

下面的示例函数使用 INC_APPEND_HISTORY 或 SHARE_HISTORY 选项之一，以便在添加历史条目后立即写出该行。 首先，它会将历史行添加到正常的历史记录中，并去掉换行符，这通常是正确的行为。 然后切换历史上下文，将该行写入当前目录下的历史文件。

```
zshaddhistory() {
  print -sr -- ${1%%$'\n'}
  fc -p .zsh_local_history
}
```

zshexit

在主 shell 即将正常退出时执行。 子shell退出或外部命令前使用 exec 前置命令修饰符时，不会调用该命令。 此外，与 TRAPEXIT 不同，函数退出时也不会调用该命令。

9.3.2 陷阱函数

以下函数受到特殊处理，但没有相应的钩子数组。

TRAP*NAL*

如果已定义且非空，每当 shell 捕捉到 SIG*NAL*（其中 *NAL* 是为 kill 内置命令指定的信号名）信号时，就会执行该函数。 信号编号将作为第一个参数传递给函数。

如果定义了这种形式的函数，但其值为空，则 shell 和由其产生的进程将忽略 SIG*NAL*。

函数的返回状态会被特别处理。 如果为零，则假定信号已被处理，并继续正常执行。 否则，除了保留陷阱的返回状态外，shell 的行为与中断时一样。

由未捕获信号终止的程序通常会返回状态 128 +信号编号。 因此，下面的代码会使 SIGINT 的处理程序打印一条信息，然后模仿信号的通常效果。

```
TRAPINT() {
  print "Caught SIGINT, aborting."
  return $(( 128 + $1 ))
}
```

函数 TRAPZERR、TRAPDEBUG 和 TRAPEXIT 绝不会在其他陷阱中执行。

TRAPDEBUG

如果设置了选项 DEBUG_BEFORE_CMD（默认情况），则在每条命令之前执行；否则在每条命令之后执行。 有关调试陷阱提供的其他功能，请参阅 Shell 内置命令 中 trap 内置命令的描述。

TRAPEXIT

在 shell 退出时执行，如果在函数内部定义，则在当前函数退出时执行。 开始执行时 $? 的值是 shell 的退出状态或函数退出时的返回状态。

TRAPZERR

当命令的退出状态为非零时执行。 但是，如果命令出现在子列表中，且后面跟有'&&'或'||'，则不执行该函数；只有这种类型的子列表中的最后一条命令才会导致陷阱被执行。 在没有 SIGERR 的系统中，函数 TRAPERR 的作用与 TRAPZERR 相同（这是通常情况）。

以 'TRAP' 开头的函数也可以用 trap 内置命令来定义：这在某些情况下可能更合适。 使用其中一种形式设置陷阱，会移除另一种形式针对同一信号的陷阱；使用其中任何一种形式移除陷阱，都会移除针对同一信号的所有陷阱。 形式

```
TRAPNAL() {
 # code
}
```

('function traps') 和

```
trap '
 # code
' NAL
```

('list traps') 在大多数方面是等价的，但以下情况除外:

- 函数陷阱具有普通函数的所有属性，会出现在函数列表中，并以自己的函数上下文而不是触发陷阱的上下文被调用。
- 函数陷阱的返回状态是特殊的，而列表陷阱的返回会导致周围的上下文以给定的状态返回。
- 根据 zsh 行为，函数陷阱不会在子shell中重置；根据 POSIX 行为，list的陷阱会重置。

---

10 作业和信号

10.1 作业

如果设置了 MONITOR 选项，交互式 shell 就会为每个管道关联一个 **作业**。 它保存了一个由 jobs 命令打印的当前作业表，并为这些作业分配了小整数编号。 当使用'&'异步启动一个作业时，shell 会在标准错误中打印一行类似下面的内容：

```
[1] 1234
```

表明异步启动的任务是任务编号 1，有一个（顶级）进程，其进程 ID 是 1234。

如果作业以 '&|'或 '&!'启动，则该作业会立即被取消（与 shell 的）关联（disowned, 可以理解成从 shell 中分离进程）。 启动后，它在作业表中没有位置，也不受此处描述的作业控制功能的约束。

如果您正在运行一项作业，并希望做其他事情，可以按 ^Z 键（control-Z），向当前作业发送 TSTP 信号：该键可以通过外部 stty 命令的 susp 选项重新定义。 shell 通常会提示作业已被 '暂停'，并打印另一个提示符。 然后，您就可以操作该作业的状态了 使用 bg 命令将其置于后台，或运行一些其他命令，最后将作业带回前台（用 前台命令 fg）。 ^Z 立即生效，就像中断一样，输入时会丢弃待处理输出和未读输入。

在后台运行的作业如果试图从终端读取数据，就会暂停。

请注意，如果前台运行的作业是 shell 函数，则暂停作业会导致 shell 分叉。 这是为了将函数的状态与执行作业控制的父 shell 的状态分开，以便后者能返回命令行提示符。 因此，即使 fg 被用来继续执行作业，该函数也不再是父 shell 的一部分，父 shell 中也看不到该函数设置的任何变量。 因此，这种行为与函数从未被暂停的情况不同。 在这一点上，Zsh 与许多其他 shell 不同。

一个额外的副作用是:如果作业是通过暂停 shell 代码的方式创建的，那么使用 disown 时就会出现延迟：只有在父 shell 启动的进程结束后，作业才能被放弃所有权。 此时，该作业会从作业列表中悄然消失。

当 shell 作为管道的右侧执行代码或任何复杂 shell 结构（如 `if`、`for` 等）时，也会出现同样的行为，以便将整个代码块作为单个作业管理。 后台作业通常允许产生输出，但可以通过命令‘`stty tostop`’禁用。 如果设置了这个 tty 选项，那么后台作业在试图产生输出时就会暂停，就像试图读取输入时一样。

当命令被挂起并在之后使用 `fg` 或 `wait` 内置命令继续执行时，zsh 会恢复挂起时有效的 tty 模式。 如果通过‘`kill -CONT`’继续执行命令，或使用 `bg` 继续执行命令，则（故意）不适用此规定。

有几种方法可以在 shell 中引用作业。 可以通过作业中任何进程的进程 ID 或以下方式之一来引用作业：

%*number*

具有给定编号的作业。

%*string*

命令行以 *string* 开头的最后一个作业。

%?*string*

命令行包含 *string* 的最后一个作业。

%%

当前作业。

%+

等价于 ‘%%’.

%-

上一个作业。

每当进程改变状态，shell 就会立即得知。 通常情况下，每当作业被阻塞，无法继续执行时，它都会通知您。 如果未设置 `NOTIFY` 选项(注：默认打开），程序会等到打印提示之前才通知你。 所有此类通知都会直接发送到终端，而不是标准输出或标准错误。

开启监控模式后，每个完成的后台任务都会触发为 `CHLD` 设置的陷阱。(注：表示子 shell 状态发生变化的信号)

当作业正在运行或暂停时，如果尝试离开 shell，系统会警告您 ‘You have suspended (running) jobs’。 您可以使用 `jobs` 命令查看这些作业。 如果这样做或立即再次尝试退出，shell 不会再发出警告；暂停的作业将被终止，如果设置了 `HUP` 选项，将向正在运行的作业发送 `SIGHUP` 信号。

要避免 shell 终止正在运行的作业，可以使用 nohup(1) 命令或 `disown` 内置命令。

10.2 信号

如果命令后跟有'&'，且 `MONITOR` 选项未激活，则调用命令的 `INT` 和 `QUIT` 信号将被忽略。 shell 本身始终会忽略 `QUIT` 信号。 否则，信号值将由 shell 从其父级继承（但请参阅 函数 中的 TRAP*NAL* 特殊函数）。

除了那些显式放入后台的工作外，shell 还会异步运行某些工作；即使在 shell 通常会等待这些工作的情况下，显式 `exit` 命令或选项 `ERR_EXIT` 导致的退出也会使 shell 不经等待而退出。 此类异步任务的例子包括进程替换（参见 进程替换）和multios的处理进程（参见 重定向 中的 ***Multios*** 部分）。

---

11 算术求值

shell 可以使用内置的 `let` 或 `$((...))`形式的替代方法，执行整数和浮点运算。 对于整数，通常在编译时使用 8 字节精度，否则精度为 4 字节。 例如，可以通过命令'`print -$(( 12345678901 ))`'来测试；如果数字显示不变，则精度至少为 8 字节。 浮点运算始终使用 'double'类型，其精度由编译器和库提供。

`let` 内置命令将算术表达式作为参数，每个参数都要单独求值。 由于许多算术运算符和空格都需要加引号，因此提供了另一种形式：对于任何以 '`((`'开头的命令，在匹配的'`))`' 之前的所有字符都被视为双引号表达式，并像 `let` 的参数一样执行算术扩展。 更确切地说，'`((...))`' 等同于'`let "..."`'。 如果表达式的算术值不为零，返回状态为 0，如果为零，返回状态为 1，如果出现错误，返回状态为 2。

例如，以下语句

```
(( val = 2 + 1 ))
```

等价于

```
let "val = 2 + 1"
```

都会将值 3 赋给 shell 变量 `val`，并返回 0 状态。

整数的进制可以不是 10。 前导'`0x`' 或'`0X`' 表示十六进制，前导 '`0b`' 或'`0B`'表示二进制。 整数也可以是'*base*`#`*n*'的形式，其中 *base* 是一个介于 2 和 36 之间的十进制数，代表算术基数，*n* 是该基数中的一个数字（例如，'`16#ff`'是十六进制的 255）。 *base*`#`也可以省略，在这种情况下使用基数 10。 为了向后兼容，也可以使用'`[`*base*`]`*n*' 的形式。

以‘*base#n*’形式给出的整数表达式或基数，可以在前导数字后包含下划线（‘_’），以起到视觉引导作用；但在计算中这些下划线将被忽略。 例如 1_000_000 或 0xffff_ffff，它们分别等价于 1000000 和 0xffffffff。

也可以指定一个用于输出的基数，形式为‘[*#base*]’，例如‘[#16]’。 这在输出算术替换或赋值给标量参数时使用，但显式定义的整数或浮点参数不受影响。 如果一个整数变量是由一个算术表达式隐式定义的，那么以这种方式指定的任何基数都将被设置为变量的输出算术基数，就好像typeset内置命令的‘-i *base*’选项一样。 表达式没有优先级，如果在数学表达式中出现多次，则使用最后出现的表达式。 为清晰起见，建议将其放在表达式的开头。 例如

```
typeset -i 16 y
print $(( [#8] x = 32, y = 32 ))
print $x $y
```

首先输出‘8#40’，即给定输出基数中最右边的值，然后输出‘8#40 16#20’，因为 y 已明确声明输出基数为 16，而 x（假设它不存在）是通过算术运算隐式键入的，在算术运算中它获得输出基数为 8。

base 可以用下划线代替或跟在下划线后面，下划线后面可以是一个正整数（如果没有正整数，则使用值 3）。 这表示应在输出字符串中插入下划线，对数字进行分组，以便视觉清晰。 跟随的整数指定了分组的数字个数。 例如：

```
setopt cbases
print $(( [#16_4] 65536 ** 2 ))
```

输出‘0x1_0000_0000’.

该功能可用于浮点数，在这种情况下，必须省略基数；分组以远离小数点的方向进行。 例如，

```
zmodload zsh/mathfunc
print $(( [#_] sqrt(1e7) ))
```

输出‘3_162.277_660_168_379_5’（显示的小数位数可能有所不同）。

如果设置了 C_BASES 选项，十六进制数将以标准 C 格式输出，例如‘0xFF’，而不是通常的‘16#FF’。 如果同时设置了选项 OCTAL_ZEROES（默认情况下没有设置），八进制数也会被类似处理，因此会显示为‘077’，而不是‘8#77’。 该选项对十六进制和八进制以外的其他基数的输出没有影响，这些格式在输入时总是可以理解的。

当使用‘[*#base*]’语法指定输出基数时，如有必要，将输出适当的基数前缀，以便输出的值是有效的输入语法。 如果双写 # ，例如‘[##16]’，则不会输出基数前缀。

浮点常量可以通过小数点或指数来识别。 小数点可以是常数的第一个字符，但指数字符 e 或 E 则不可以，因为它会被当作参数名。 所有数字部分（小数点前后和指数部分）的前导数字后都可以包含下划线，以起到视觉引导作用；在计算中这些下划线将被忽略。

算术表达式使用的语法和表达式的结合性与 C 语言几乎相同。

在原生运算模式下，支持以下运算符（按优先级递减顺序排列）：

+ - ! ~ ++ --

一元加/减、逻辑 NOT、补码、{前,后}缀自{增,减}运算符

<< >>

向左、向右位移

&

按位与

^

按位异或

|

按位或

**

乘幂

* / %

乘法、除法、模数（余数）

+ -

加法、减法

< > <= >=

比较

== !=

等于，不等于

&&

逻辑与

|| ^^

逻辑或，逻辑异或

? :

三元运算符

= += -= *= /= %= &= ^= |= <<= >>= &&= ||= ^^= **=

赋值

,

逗号运算符

运算符 '&&', '||', '&&=', 和'||=' 是短路运算符，三元运算符中后两个表达式中只有一个会被求值。 请注意按位 AND、OR 和 XOR 运算符的优先级。

如果使用选项 C_PRECEDENCES，运算符的优先级（但不包括其他属性）将被修改为与支持相关运算符的大多数其他语言中的运算符相同：

+ - ! ~ ++ --

一元加/减、逻辑 NOT、补码、{前,后}缀自{增,减}运算符

**

乘幂

* / %

乘法、除法、模数（余数）

+ -

加法、减法

<< >>

向左、向右位移

< > <= >=

比较

== !=

等于，不等于

&

按位与

^

按位异或

|

按位或

&&

逻辑与

^^

逻辑异或

||

逻辑或

? :

三元运算符

= += -= *= /= %= &= ^= |= <<= >>= &&= ||= ^^= **=

赋值

,

逗号运算符

注意这两种情况下指数运算的优先级都低于一元运算符，因此'-3**2'的运算结果是'9'，而不是'-9'。 必要时使用括号： '-(3**2)'. 这是为了与其他 shell 兼容。

数学函数的调用语法为 '*func*(*args*)'，其中函数决定 *args* 是用作字符串还是逗号分隔的算术表达式列表。shell 目前默认不定义数学函数，但可以使用 zmodload 内置命令加载 zsh/mathfunc 模块，以提供标准浮点数学函数。

形式为 '##*x*' 的表达式（其中 *x* 是任何字符序列，如 'a'、'^A'或'\M-\C-x' ）给出了该字符的值，形式为'#*name*'的表达式给出了参数 *name* 内容中第一个字符的值。 字符值是根据当前本地使用的字符集确定的；要处理多字节字符，必须设置选项 MULTIBYTE。 需要

注意的是，这种形式不同于 '$#*name*'，后者是一种标准的参数替换形式，会给出参数 *name* 的长度。'#\'可以代替 '##'，但已被弃用。

命名参数和下标数组可以在算术表达式中通过名称引用，而无需使用参数扩展语法。 例如

```
((val2 = val1 * 2))
```

会将 $val1 值的两倍赋值给名为 val2 的参数。

可以使用 integer 内置命令指定已命名参数的内部整数表示。 以这种方式对声明的整数型的命名参数，每次赋值都进行算术运算。 将浮点数赋值给整数会导致四舍五入到零。 (?)

同样，浮点数也可以使用 float 内置命令来声明；浮点数有两种类型，它们的区别仅在于输出格式，正如 typeset 内置命令所描述的那样。 输出格式可以通过使用算术替换代替参数替换来绕过，即'${*float*}'使用定义的格式，而'$((*float*))'使用通用浮点格式。

必要时会将整数值提升为浮点数。 此外，如果任何需要整数参数的运算符（'&', '|', '^', '<<', '>>'及其等价赋值运算符）被赋予浮点参数，除了'~' 会向下舍入外，其他运算符都会被默默地舍入为零。

用户需要注意的是，与许多其他编程语言（但不是为计算而设计的软件）一样，在 zsh 中对表达式的计算是一次一个项进行的，在仅包含整数的项中不会将整数提升为浮点数。 FORCE_FLOAT shell 选项可用于需要浮点运算的脚本或函数中。

标量变量可以在不同时间保存整数或浮点数值；在这种情况下，没有数值类型的内存。

如果一个变量在数值上下文中首次赋值时没有事先声明，那么它将被隐式地类型化为 integer 或 float，并保持这种类型直到类型被显式地更改或作用域结束。 这可能会产生意想不到的后果。 例如，在循环

```
for (( f = 0; f < 1; f += 0.1 )); do
# use $f
done
```

如果 f 尚未被声明，第一次赋值将导致它被创建为一个整数，因此操作 'f += 0.1' 将始终导致结果被截断为零，从而导致循环失败。 一个简单的解决方法是将初始化变成 'f = 0.0'。 因此，最好使用显式类型声明数值变量。

---

12 条件表达式

条件表达示 与 [[复合命令配合使用，可用于测试文件属性和比较字符串。 每个表达式可以由以下一个或多个一元或二元表达式构成：

-a *file*

如果 *file* 存在，则为 true。

-b *file*

如果 *file* 已存在且是块专用文件，则为 true。

-c *file*

如果 *file* 存在且是字符特殊文件，则为 true。

-d *file*

如果 *file* 存在且是一个目录，则为 true。

-e *file*

如果 *file* 存在，则为 true。

-f *file*

如果 *file* 存在且是普通文件，则为 true。

-g *file*

如果 *file* 已存在且设置了 setgid 位，则为 true。

-h *file*

如果 *file* 存在且是符号链接，则为 true。

-k *file*

如果 *file* 存在且设置了粘性位，则为 true。

-n *string*

如果 *string* 的长度非零，则为 true。

-o *option*

如果名为 *option* 的选项已开启，则为 true。 *option* 可以是单字符，在这种情况下就是单字母选项名称。 (参见 指定选项）。

如果不存在名为 *option* 的选项，且 POSIX_BUILTINS 选项尚未设置，则返回 3 并发出警告。 如果设置了该选项，则返回 1，不带警告。

-p *file*

如果 *file* 存在且是 FIFO 特殊文件（命名为管道），则为 true。

`-r` *file*

如果 *file* 存在且当前进程可读取，则为 true。

`-s` *file*

如果 *file* 存在且大小大于零，则为 true。

`-t` *fd*

如果文件描述符编号 *fd* 已打开并与终端设备关联，则为 true。 (注意：*fd* 不是可选项)

`-u` *file*

如果 *file* 已存在且设置了 setuid 位，则为 true。

`-v` *varname*

如果 shell 变量 *varname* 已设置，则为 true。

`-w` *file*

如果 *file* 存在且当前进程可写，则为 true。

`-x` *file*

如果 *file* 存在且当前进程可执行，则为 true。 如果 *file* 存在且是一个目录，则当前进程有权限在该目录中搜索。

`-z` *string*

如果 *string* 的长度为零，则为 true。

`-L` *file*

如果 *file* 存在且是符号链接，则为 true。

`-O` *file*

如果 *file* 存在且为该进程的有效用户 ID 所有，则为 true。

`-G` *file*

如果 *file* 存在，且其组群与此进程的有效组群 ID 一致，则为 true。

`-S` *file*

如果 *file* 存在且是套接字，则为 true。

`-N` *file*

如果 *file* 存在，且其访问时间不新于修改时间，则为 true。

file1 `-nt` *file2*

如果 *file1* 存在且比 *file2* 新，则为 true。

file1 `-ot` *file2*

如果 *file1* 存在且比 *file2* 早，则为 true。

file1 `-ef` *file2*

如果 *file1* 和 *file2* 存在并指向同一文件，则为 true。

string = *pattern*
string == *pattern*

如果 *string* 与 *pattern* 匹配，则为 true。 这两种形式完全等价。 '=' 形式是传统的 shell 语法（因此也是 `test` 和 `[` 内置命令中唯一常用的语法）； '=='形式提供了与其他计算机语言的兼容性。

string `!=` *pattern*

如果 *string* 与 *pattern* 不匹配，则为 true。

string =~ *regexp*

如果 *string* 与正则表达式 *regexp* 匹配，则为 true。 如果设置了 `RE_MATCH_PCRE` 选项，*regexp* 将使用 `zsh/pcre` 模块作为 PCRE 正则表达式进行测试，否则将使用 `zsh/regex` 模块作为 POSIX 扩展正则表达式进行测试。 匹配成功后，某些变量将被更新；匹配失败时，变量不会发生变化。

如果未设置选项 `BASH_REMATCH`，标量参数 `MATCH` 将设置为与模式匹配的子串，整数参数 `MBEGIN` 和 `MEND` 将分别设置为 *string* 中匹配开始和结束的索引， 这样，如果 *string* 包含在变量 `var` 中，表达式 '`${var[$MBEGIN,$MEND]}`' 与 '`$MATCH`'相同。这遵守选项 `KSH_ARRAYS` 的设置。同样，数组 `match` 会被设置为与带括号的子表达式相匹配的子串，数组 `mbegin` 和 `mend` 会被分别设置为 *string* 中子串的开始和结束位置的索引。 如果没有带括号的子表达式，则不会设置数组。例如，如果字符串'`a short string`' 与正则表达式'`s(...)t`'相匹配,那么（假设未设置 `KSH_ARRAYS` 选项），`MATCH`、`MBEGIN` 和 `MEND` 分别是'`short`'、3和 7，而 `match`、`mbegin` 和 `mend` 则是分别包含字符串'`hor`', '4'和'6'的单条目数组。

如果设置了选项 `BASH_REMATCH`，数组 `BASH_REMATCH` 将被设置为与模式匹配的子串，然后是与模式内带括号子表达式匹配的子串。

string1 < *string2*

根据 *string1* 和 *string2* 字符的 ASCII 值，如果 *string1* 在 *string2* 之前，则为 true。

string1 > *string2*

根据 *string1* 和 *string2* 的 ASCII 码值，如果 *string1* 在 *string2* 之后，则为 true。

exp1 `-eq` *exp2*

如果 *exp1* 在数值上等于 *exp2*，则为 true。 请注意，对于纯粹的数值比较，使用 `((...))` 内置命令 算术求值 比条件表达式更方便。

exp1 `-ne` *exp2*

如果 *exp1* 在数值上不等于 *exp2*，则为 true。

exp1 `-lt` *exp2*

如果 *exp1* 在数值上小于 *exp2*，则为 true。

exp1 `-gt` *exp2*

如果 *exp1* 在数值上大于 *exp2*，则为 true。

exp1 `-le` *exp2*

如果 *exp1* 在数值上小于或等于 *exp2*，则为 true。

exp1 `-ge` *exp2*

如果 *exp1* 在数值上大于或等于 *exp2*，则为 true。

(*exp*)

如果 *exp* 为 true，则为 true。

! *exp*

如果 *exp* 为 false，则为 true。

exp1 && *exp2*

如果 *exp1* 和 *exp2* 均为 true，则为 true。

exp1 || *exp2*

如果 *exp1* 或 *exp2* 为 true，则为 true。

为兼容起见，如果有一个在语法上不重要的参数，通常是一个变量，那么该条件将被视为对表达式是否扩展为长度不为零的字符串的测试。 换句话说，`[[ $var ]]` 与 `[[ -n $var ]]` 相同。 建议尽可能使用第二种显式形式。

对 *file*、*string* 和 *pattern* 参数执行正常的 shell 扩展，但每次扩展的结果都限制为一个单词，类似于双引号的效果。

文件名生成不会在任何形式的条件参数中执行。 不过，在正常 shell 扩展有效的情况下，以及在选项 EXTENDED_GLOB 有效的情况下，可以通过在字符串末尾使用 (#q) 形式的显式 glob 限定符来强制生成文件名。 在 'q' 和结尾括号之间可能会出现一个普通的 glob 限定符表达式；如果没有出现，该表达式除了生成文件名外没有任何作用。 与其他形式的扩展结果一样，文件名生成的结果会连接在一起形成一个单词。

只有在使用 `[[` 语法时，才能使用这种特殊的文件名生成方式。 如果条件出现在 `[` 或 `test` 内置命令中，则在条件求值之前，会作为正常命令行扩展的一部分进行套选。 在这种情况下，可能会产生多个单词，从而混淆测试命令的语法。

例如,

```
[[ -n file*(#qN) ]]
```

如果当且仅当当前目录中至少有一个以字符串'file'开头的文件时，才会产生状态 0。 全局(globbing)限定符 N 确保在没有匹配文件的情况下表达式为空。

模式元字符对 *pattern* 参数有效；模式与用于文件名生成的模式相同，请参阅 文件名生成，但不使用 '/'或前导点（initial dots），也不允许使用 glob 限定符。

在上述每个表达式中，如果 *file* 的形式为 '/dev/fd/*n*'，其中 *n* 为整数，那么即使底层系统不支持 /dev/fd 目录，测试也会应用于描述符编号为 *n* 的打开的文件。

在进行数字比较的表格中，表达式 *exp* 会进行算术扩展，就像它们被括入 $((...)) 中一样。

例如，以下内容：

```
[[ ( -f foo || -f bar ) && $report = y* ]] && print File exists.
```

测试文件 foo 或文件 bar 是否存在，如果存在，则测试参数 report 的值是否以'y'开头；如果整个条件为真，则打印信息'File exists.'。

---

13 提示符扩展

13.1 扩展提示符序列

提示符序列会经过一种特殊形式的扩展。 使用 print 内置程序的 -P 选项也可以进行这种扩展。

如果设置了 PROMPT_SUBST 选项，提示符字符串将首先进行 **参数扩展**、**命令替换** 和 **算术扩展**。 参见 扩展。

提示符字符串中可以识别某些转义序列。

如果设置了 PROMPT_BANG 选项，提示符中的 '!' 将被当前历史事件编号取代。 字面的 '!' 可以表示为'!!'。

如果设置了 PROMPT_PERCENT 选项，某些以'%'开头的转义序列将被扩展。 许多转义字符后面只有一个字符，但其中一些转义字符还包含一个可选的整数参数，该参数应出现在'%'和序列的下一个字符之间。 还有更复杂的转义序列可用于提供条件扩展。

13.2 简单的提示符转义

13.2.1 特殊字符

%%

一个 '%'.

%)

一个 ')'.

13.2.2 登录信息

%l

用户登录的线路（tty），不含前缀 '/dev/'。 如果名称以 '/dev/tty' 开头，则去掉前缀。

%M

完整的计算机主机名。

%m

直至第一个'.' 的主机名。 在'%' 后面可以加上一个整数，以指定需要主机名的多少个分量。 如果是负整数，则显示主机名的尾部。

%n

$USERNAME.

%y

用户登录的线路（tty），不含前缀‘/dev/’。 这不会特别处理‘/dev/tty’名称。

13.2.3 shell 状态

%#

如果 shell 以特权运行，则为‘#’；如果不是，则为‘%’。 等同于‘%(!.#.%%)’。 就这些目的而言，‘特权’的定义是有效用户 ID 为零，或者，如果如果支持 POSIX.1e 功能（capabilities），则在有效功能向量或可继承功能向量（capability vectors）中至少有一种功能被提出。

%?

提示符前执行的最后一条命令的返回状态。

%_

解析器的状态，即已在命令行启动的 shell 结构（如‘if’和‘for’）。如果给定的是整数，那么打印的字符串数就是这个数；如果是零、负数或无整数，那么打印所有的字符串。 这在 PS2 续行提示符和 使用 XTRACE 选项调试的PS4 提示符中最有用；在后一种情况下，它也可以非交互式地工作。

%^

解析器的状态（反向）。除了字符串的顺序外，它与‘%_’相同。 它常用于 RPS2。

%d
%/

当前工作目录。 如果在‘%’后面加上一个整数，则表示要显示的当前工作目录的尾部组件数；0 表示整个路径。 负整数表示前导部分，即 %-1d 表示第一个部分。

%~

与 %d 和 %/ 相同，但如果当前工作目录以 $HOME 开头，则该部分会被‘~’替换。此外，如果它的前缀是一个已命名的目录，那么这部分内容会被‘~’紧跟目录名的形式所取代，但前提是结果比完整路径短；文件名扩展。

%e

当前引入的（sourced）文件、shell 函数或 eval 的求值深度。 每次 %N 的值被设置或恢复到之前的值时，该值都会递增或递减。 作为 $PS4 的一部分，它在调试时最为有用。

%h
%!

当前历史事件编号。

%i

%N 给出的脚本、引入的（sourced）文件或 shell 函数中当前正在执行的行号。 作为 $PS4 的一部分，它对调试最有用。

%I

文件 %x 中当前正在执行的行号。 这与 %i 类似，但行号始终是定义代码的文件中的行号，即使代码是 shell 函数。

%j

作业数量。

%L

$SHLVL 的当前值。

%N

zsh 当前执行的脚本、引入的（sourced）文件或 shell 函数的名称，以最近启动者为准。 如果没有，则等同于参数 $0。 ‘%’ 后面可以跟一个整数，用于指定要显示的尾部路径成分的数量；0 表示完整路径。 负整数表示前导成分。

%x

包含当前正在执行的源代码的文件名。 与 %N 相同，但不显示函数名和计算（eval）命令名，而是显示定义这些函数和命令的文件名。

%c
%.
%C

当前工作目录的尾部分量。 ‘%’后面可以跟一个整数，以获取多个分量。 除非使用了‘%C’，否则会先执行转折号收缩。 由于 %c 和 %C 分别等同于 %1~ 和 %1/，而显式正整数与后两个序列的效果相同，因此已被弃用。

13.2.4 日期和时间

%D

日期，格式为 *yy-mm-dd*

%T

当前时间（24 小时制）。

%t
%@

当前时间（12 小时制，上午/下午格式）。

%*

当前时间（24 小时制，含秒）。

%w

日期，格式为 *day-dd*。

%W

日期，格式为 *mm/dd/yy*。

%D{*string*}

string 使用 strftime 函数格式化。 详情请参阅 strftime(3)。 如果数字是个位数，各种 zsh 扩展会提供不带前导零或空格的数字：

%f

月中的一日

%K

24小时制中的小时

%L

12小时制中的小时

此外，如果系统支持 POSIX 的gettimeofday 系统调用，%. 将提供自纪元（epoch）起的秒数（带前导零的小数）。 默认情况下提供三位小数，但也可以在 % 后面提供不超过 9 位的数字；因此 %6. 输出微秒，而 %9. 输出纳秒。 (后者需要纳秒精度的 clock_gettime；缺少clock_gettime的系统将返回一个乘以相应 10的 次方的值）。 一个典型的例子是格式'%D{%H:%M:%S.%.}'。

GNU 扩展名 %N 作为 %9. 的同义词处理。

此外，对于 d, f, H, k, l, m, M, S and y格式字符，GNU 扩展（ % 和 格式字符之间有一个 '-' ）会导致前导零或空格被删除，这会直接由shell处理； 任何其他格式字符都将提供给系统的 strftime(3)，并带有前导'-'，因此处理方法取决于系统。 其他 GNU（或其他）扩展字符也会传递给 strftime(3)，如果系统支持这些扩展字符，它们也可以正常工作。

13.2.5 视觉效果

%B (%b)

启动（停止）粗体模式。

%E

清除到行尾。

%U (%u)

启动（停止）下划线模式。

%S (%s)

启动（停止）突出模式。

%F (%f)

开始（停止）使用不同的前景色（如果终端支持）。 颜色有两种指定方式：一种是以数字参数的形式指定，如正常情况；另一种是在 %F 后用大括号指定，如 %F{red}。 在后一种情况下，所允许的值与 fg zle_highlight 属性所描述的一样；字符高亮。 这意味着第二种格式也允许使用数字颜色。

%K (%k)

开始（停止）使用不同的背景颜色。 语法与 %F 和 %f 相同。

%{...%}

包含一个字符串作为字面转义序列。 括号内的字符串不应改变光标位置。 括号对可以嵌套。

在 % 和 { 之间的正数参数的处理方法与下面的 %G 相同。

%G

在%{...%}序列中，包含一个 '间隙'：即假定只输出一个字符宽度。 这在输出 shell 无法正确处理的字符时非常有用，例如某些终端上的备用字符集。 有关字符可以包含在 %{...%} 序列中，并与适当数量的 %G 序列一起表示正确的宽度。 '%'和'G' 之间的整数表示 1以外的字符宽度。 因此 %{*seq*%2G%} 输出 *seq* 并假定它占两个标准字符的宽度。

%G 的多次使用会以显而易见的方式累加；%G 的位置并不重要。 负整数不作处理。

请注意，在使用提示符截断功能时，最好将每个 %{...%} 组内的输出划分为单个字符，以便找到正确的截断点。

13.3 提示符中的条件子字符串

%v

psvar 数组参数第一个元素的值。 在‘%’后跟一个整数，表示数组中的该元素。 负整数从数组末尾开始计数。

%(*x*.*true-text*.*false-text*)

指定一个三元表达式。 *x* 后面的字符是任意的；‘true’ 结果和‘false’结果的文本是用同一个字符分隔的。 除了作为 %-escape 序列的一部分，该分隔符不得出现在 *true-text* 中。 ‘)’可以以‘%)’的形式出现在 *false-text* 中。 *true-text* 和 *false-text* 都可以包含任意嵌套的转义序列，包括进一步的三元表达式。

左括号前后可以是正整数 *n*，默认值为 0。 负整数将乘以-1，下文中的 ‘l’除外。 测试字符 *x* 可以是以下任何一种字符：

!

如果 shell 以特权运行，则为 True。

#

如果当前进程的有效 uid 为 *n*，则为 True。

?

如果最后一条命令的退出状态为 *n*，则为 True。

_

如果至少 *n* 个 shell 结构体已启动，则为 True。

C
/

如果当前绝对路径至少有 *n* 个相对于根目录的元素，则为 true，因此 / 将被视为 0 个元素。

c
.
~

如果当前路径在替换前缀后至少有 *n* 个相对于根目录的元素，则为 true，因此 / 将被视为 0 个元素。

D

如果月份等于 *n*，则为 true（1 月 = 0）。

d

如果月份的日期等于 *n*，则为 True。

e

如果计算（evaluation）深度至少为 *n*，则为 True。

g

如果当前进程的有效 gid 为 *n*，则为 True。

j

如果作业数量至少为 *n*，则为 True。

L

如果 SHLVL 参数至少为 *n*，则为 True。

l

如果当前行至少已打印 *n* 个字符，则为 true。 当 *n* 为负数时，如果至少 abs(*n*) 字符仍在对边距之前（即 RPROMPT 的左边距），则为 true。

S

如果 SECONDS 参数至少为 *n*，则为 True。

T

如果时间（小时）等于 *n*，则为 True。

t

如果时间（分钟）等于 *n*，则为 True。

v

如果数组 psvar 至少有 *n* 个元素，则为 True。

V

如果数组 psvar 中的元素 *n* 已设置且非空，则为 True。

w

如果星期几等于 *n*（星期日 = 0），则为 True。

%<*string*<

%>*string*>
%[*xstring*]

指定提示符字符串剩余部分的截断行为。 第三种形式已被淘汰，相当于‘%*xstringx*’，即 *x* 可以是 ‘<’ 或 ‘>’。 *string* 将代替任何字符串的截断部分显示出来；请注意，这不会进行提示符扩展。

在第三种形式中，数字参数可能紧接在 ‘[’之后，用于指定提示符中可显示的各种字符串的最大允许长度。 在前两种形式中，数字参数可以是负数，在这种情况下，截断长度是由当前提示符行剩余字符数减去数字参数的绝对值确定的。 如果结果为零或负数，则使用长度 1。 换句话说，如果参数为负数，则在截断后，右边距（RPROMPT的左边距）前至少要保留 *n* 个字符。

带 ‘<’ 的形式在字符串左边截断，带‘>’ 的形式在字符串右边截断。 例如，如果当前目录为 ‘/home/pike’，提示符 ‘%8<..<%/’ 将扩展为 ‘..e/pike’。 在该字符串中，结束符（‘<’, ‘>’ 或 ‘]’）或实际上的任何字符都可以用前面的‘\’来引用；但在使用 `print -P` 时要注意，除了双引号字符串去掉的反斜线外，该字符串还需经过标准的 `print` 处理，因此必须双写：最糟糕的情况是‘`print -P "%<\\\\<<..."`’。

如果 *string* 长度大于指定的截断长度，则会以完整字符串出现，完全替换截断的字符串。

要截断的提示符字符串部分会一直延伸到字符串的末尾，或到下一个 ‘%(’ 结构体的闭合组的末尾，或到下一个在同一分组级别遇到的截断（即 ‘%(’内部的截断是独立的），以先到者为准。 特别是，参数为零的截断（例如，‘%<<’）标志着要截断的字符串范围的结束，并从此关闭截断。例如，提示符 ‘`%10<...<%~%<<%# `’ 将打印当前目录的截断表示，后面跟一个‘%’ 或 ‘#’ ，后面跟一个空格。 如果没有 ‘%<<’ ，这两个字符将包含在被截断的字符串中。 请注意， ‘%-0<<’ 并不等同于 ‘%<<’ ，而是指定在右边空白处截断提示符。

截断仅适用于以内嵌换行符（如有）为分界的每一行提示符。 如果截断后任何一行提示符的总长度大于终端宽度，或者要截断的部分包含内嵌换行符，那么截断行为将是未定义的，并可能在 shell 的未来版本中发生变化。 当可用空间小于 *n* 时，使用 ‘%-*n*(l.*true-text*.*false-text*)’ 删除部分提示符。

---

14 扩展

以下类型的扩展按指定顺序分五个步骤进行：

历史扩展

仅在交互式 shell 中执行。

别名扩展

按照 别名 中的说明，别名会在解析命令行之前立即展开。

进程替换
参数扩展
命令替换
算术扩展
括号扩展

这五个步骤从左到右依次进行。 在每个参数上，所需的五个步骤中的任何一个都会相继执行。 因此，举例来说，在开始命令替换之前，参数扩展的所有部分都已完成。 在这些扩展之后，所有未引用的 ‘\’,‘'’ 和 ‘"’ 都会被删除。

文件名扩展

如果设置了 SH_FILE_EXPANSION 选项，则会修改扩展顺序，以便与 *sh* 和 *ksh* 兼容。 在这种情况下，**文件名扩展**会紧接着 **别名扩展** 之后执行，排在上述五种扩展之前。

文件名生成

这种扩展通常称为 通配，总是最后进行。

以下各节将详细说明扩展的类型。

14.1 历史扩展

历史扩展功能可让你在键入的命令行中使用以前命令行中的单词。 这可以简化拼写更正和复杂命令或参数的重复输入。

每条命令在执行前都会立即保存在历史列表中，其大小由 HISTSIZE 参数控制。 在任何情况下，最近的一条命令都会被保留下来。 历史列表中保存的每一条命令都称为历史 **事件**，并分配一个编号，从 shell 启动时的 1（1）开始。 您可能在提示符中看到的历史记录编号（参见 提示符扩展）就是分配给 **下一个** 命令的编号。

14.1.1 概述

历史扩展以 histchars 参数的第一个字符开始，默认为 ‘!’，可以出现在命令行的任何位置，包括双引号内（但不包括单引号 '...' 或 C风格引号 $'...' ，也不包括反斜线转义）。

第一个字符之后是一个可选的事件指示器（事件指示器），然后是一个可选的单词指示器（单词指示器）；如果这两个指示器都不存在，则不会进行历史扩展。

包含历史扩展的输入行在扩展后，其他扩展发生前和命令执行前，会有回显。 正是这种扩展后的形式被记录为历史事件，供以后引用。

历史扩展不会嵌套。

默认情况下，不带事件指示器的历史记录，会引用与该命令行的前一个历史引用指向的相同事件；如果它是一条命令中唯一的历史记录引用，则指向前一条命令。 但是，如果设置了选项 CSH_JUNKIE_HISTORY，那么每个没有事件规范（specification）的历史引用 **总是** 指向前一条命令。

例如，'!' 是前一条命令的事件指示器，因此 '!!:1' 总是指前一条命令的第一个字，而 '!!$' 总是指前一条命令的最后一个字。 如果设置了 CSH_JUNKIE_HISTORY，则 '!:1' 和 '!$' 的功能分别与 '!!:1' 和 '!!$' 相同。 反之，如果 CSH_JUNKIE_HISTORY 未设置，则 '!:1' 和 '!$' 分别指向当前命令行中最近的历史引用所指向的同一事件的第一个字和最后一个字，如果没有前一个引用，则指向上一条命令。

字符序列 '^*foo*^*bar*'（其中 '^' 实际上是 histchars 参数的第二个字符）重复上一条命令，用 *bar* 替换 *foo* 字符串。 更确切地说，序列 '^*foo*^*bar*^' 与 '!!:s^*foo*^*bar*^' 是同义词，因此在最后的 '^' 后面还可以加上其他修饰符（参见 修饰符）。 特别是，'^*foo*^*bar*^:G' 执行全局替换。

如果 shell 在输入中遇到字符序列 '!"'，历史机制会暂时失效，直到当前列表（参见 Shell 语法）完全解析完毕。 '!"' 将从输入中删除，随后的 '!' 字符将没有特殊意义。

fc 内置程序提供了一种不太方便但更易于理解的命令历史记录支持。

14.1.2 事件指示器

事件指示器是对历史记录列表中命令行条目的引用。 在下面的列表中，请记住，通过设置 histchars 参数，可以将每个条目中的初始 `!' 更改为其他字符。

!

开始历史扩展，除非后面是空白、换行符、'=' 或 '('。 如果后面紧跟一个单词指示器（单词指示器），则形成一个没有事件指示器的历史记录引用（概述）。

!!

引用上一条命令。 该扩展本身重复前一条命令。

!*n*

引用命令行 *n*。

!-*n*

引用当前命令行减 *n*。

!*str*

引用以 *str* 开头的最近命令。

!?*str*[?]

引用包含 *str* 的最近命令。 如果该引用后面有修饰符，或后面有任何不被视为 *str* 一部分的文本，则尾部的 '?' 是必要的。

!#

引用当前键入的命令行。 该命令行将被视为完整的命令行，直到并包括 '!#' 引用之前的单词。

!{...}

将历史引用字符与相邻字符隔开（如有必要）。

14.1.3 单词指示器

单词指示器表示指定命令行中的哪个单词或哪些单词将包含在历史记录引用中。 通常，':' 将事件说明与单词指示器分隔开来。 只有在单词代号以 '^', '$', '*', '-'或 '%' 开头时才可以省略。 单词指示器包括:

0

第一个输入词（命令）。

n

第 *n* 个参数

^

第一个参数。 即 1。

$

最后一个参数。

%

?*str* 搜索(最近一次）匹配到的单词。

x-*y*

字词范围；*x* 默认为 0。

*

所有参数，如果没有参数则为空值。

*x**

缩写为 '*x*-$'。

x-

类似于 '*x**'，但省略了 $。

请注意，'%' 单词指示器只有在和 '!%', '!:%' 或 '!?*str*?:%' 一起使用时有用，并且只能在 !? 扩展（可能在之前的命令中）之后使用。否则会导致错误，尽管错误可能不是最明显的。

14.1.4 修饰符

在可选的单词指示器后，您可以添加以下一个或多个修改器序列，每个修饰符前都有一个 ':'。 这些修改器也适用于 **文件名生成** 和 **参数扩展** 的结果，除非另有说明。

a

将文件名转换为绝对路径：会在必要时对当前目录进行预处理；删除 .' 路径段；删除 '..' 路径段和紧接其后的路径段。

这种转换与文件系统中的内容无关，也就是说，它只针对逻辑目录，而非物理目录。当 CHASE_DOTS 或 CHASE_LINKS 选项均未设置时，转换方式与更改目录的方式相同。例如，'/before/here/../after' 总是被转换为 '/before/after'，而不管 '/before/here' 是否存在，也不管它是哪种对象（目录、文件、符号链接等）。

A

将文件名转换为绝对路径，就像 'a' 修饰符所做的那样，然后 ***then*** 将结果通过 realpath(3) 库函数来解析符号链接。

注意：在没有 realpath(3) 库函数的系统上，符号链接不会被解析，因此在这些系统上，'a' 和 'A' 是等价的。

注意 foo:A 和 realpath(foo) 在某些输入上是不同的。 关于 realpath(foo) 的语义，请参阅 'P' 修饰符。

c

通过搜索 PATH 变量给出的命令路径，将命令名称解析为绝对路径。 这不适用于包含目录部分的命令。 还需注意的是，除非在当前目录下找到同名文件，否则该命令通常不能用作 glob 限定符。

e

除文件名扩展名中 '.' 之后的部分外，全部删除；文件名扩展名的定义请参阅下面对 r 修饰符的描述。 请注意，根据该定义，如果字符串以 '.' 结尾，结果将为空。

h [*digits*]

删除路径名的尾部组件，将路径缩短一级目录：这是路径名的 "头"。 其作用类似于 'dirname'。 如果 h后面紧跟任意数量的十进制数字（没有空格或其他分隔符），且所得数字的值非零，则保留该数量的前导成分，而不是删除最后一个成分。 在绝对路径中，前导成分 '/' 是第一个成分，例如，如果 var=/my/path/to/something，那么 ${var:h3} 将取代 /my/path。 连续的 '/' 与单个 '/' 的处理方式相同。 在参数替换中，只有当表达式位于大括号中时才能使用数字，例如，简式替换 $var:h2 将被视为 ${var:h}2，而不是 ${var:h2}。 在历史替换或 globbing 限定符中使用数字不受限制。 如果请求的组件数多于实际存在的组件数，则会替换整个路径（因此不会在历史扩展中触发 'failed modifier' 错误）。

l

将单词转换为全小写。

p

打印新命令，但不执行。 仅适用于历史扩展。

P

将文件名转换为绝对路径，如 realpath(3)。 生成的路径将是绝对路径，将指向与输入文件名相同的目录条目，并且其任何组成部分都不会是符号链接或等于 '.' 或 '..'。

与 realpath(3)不同的是，允许并保留不存在的尾部成分。

q

引用被替换的词语，并转义进一步的替换。 可用于历史扩展和参数扩展，但对于参数而言，只有在生成的文本需要重新计算（如 eval）时才有用。

Q

去掉替换词的一级引号。

r

删除文件扩展名，保留根名称。 没有文件扩展名的字符串不会被更改。 文件名扩展名是‘.’，后跟任意数量的字符（包括零个），这些字符既不是‘.’，也不是‘/’，并一直延续到字符串的末尾。 例如，‘foo.orig.c’的扩展名是‘.c’，而‘dir.c/foo’没有扩展名。

s/*l*/*r*[/]

如下所述，用 *r* 代替 *l*。 替换只针对与 *l* 匹配的第一个字符串。 对于数组和文件名生成，这适用于扩展文本的每个单词。 有关替换的进一步说明，请参阅下文。

形式‘gs/*l*/*r*’和‘s/*l*/*r*/:G’执行全局替换，即用 *l* 替换 *r* 的每一次出现。 请注意，g 或 :G 必须准确出现在所示位置。

请参阅下文关于这种替代形式的进一步说明。

&

重复前面的 s 替换。 与 s一样，可以在前面紧接一个 g。 在参数扩展中，& 必须出现在大括号内；在文件名生成时，必须用反斜线引用。

t [*digits*]

删除路径名的所有前导成分，留下最后一个成分（尾部）。 其工作原理类似于‘basename’。 首先会删除任何尾部的斜线。 小数位的处理方法如上文 (h) 所述，但在这种情况下，尾部成分的个数将被保留，而不是默认的 1；0 与 1 的处理方法相同。

u

将单词转换为全大写。

x

与 q类似，但在空白处分词。 不支持参数扩展。

s/*l*/*r*/ 替换的工作原理如下。 默认情况下，替换的左侧不是模式，而是字符串。 任何字符都可以代替‘/’作为分隔符。 反斜线引用分隔符。 右侧 *r* 中的字符‘&’将被左侧 *l* 中的文本替换。 ‘&’可以用反斜线引用。空的 *l* 使用前一个 *l* 中的字符串或前一个上下文扫描字符串‘!?*s*’中的*s*。 如果 *r* 后面紧跟换行符，则可以省略最右边的分隔符；同样，也可以省略上下文扫描中最右边的‘?’。 请注意，在所有扩展形式中，最后一个 *l* 和 *r* 的记录都是相同的。

请注意，如果在 glob 限定符中使用了‘&’，则需要额外的反斜杠，因为在这种情况下 & 是一个特殊字符。

还要注意的是，扩展的顺序会影响 *l* 和 *r* 的解释。 在历史扩展中使用时(在其它扩展发生前)，*l* 和 *r* 将被视为字面字符串（下文对 HIST_SUBST_PATTERN 的解释除外）。 在参数扩展中使用时，首先将 *r* 替换为参数值，然后应用附加的进程、参数、命令、算术或括号引用，如果 *l* 在起始值中出现不止一次，可能会对这些替换和扩展进行多次计算。 在 glob 限定符中使用时，即使在 ':s'表达式本身被分为 *l* 和 *r* 两边之前，也会在解析限定符时执行一次替换或扩展。

如果设置了选项 HIST_SUBST_PATTERN，*l* 将被视为 文件名生成 中描述的通常形式的模式。 这可以在所有可以使用修饰符的地方使用；但需要注意的是，globbing 限定符中的参数替换已经发生，因此替换字符串中的参数应该加引号，以确保在正确的时间被替换。还要注意的是，globbing 限定符中使用的复杂模式可能需要使用扩展的 glob 限定符符号 (#q:s/…/…/)，这样 shell 才能将表达式识别为 glob 限定符。 此外，请注意替换中的坏模式不受 NO_BAD_PATTERN 选项的限制，因此会导致错误。

设置 HIST_SUBST_PATTERN 时，*l* 可以以 # 开头，表示模式必须在要替换的字符串的开头匹配；% 可以出现在 # 的开头或后面，表示模式必须在要替换的字符串的结尾匹配。 % 或 # 可以用两个反斜线引用。

例如，下面这段使用 EXTENDED_GLOB 选项的文件名代码：

```
print -r -- *.c(#q:s/#%(#b)s(*).c/'S${match[1]}.C'/)
```

使用扩展 *.c，并应用 (#q…) 表达式中的 glob 限定符，该表达式由锚定在每个单词开头和结尾的替换修饰符 (#%) 组成。 这样就打开了反向引用 ((#b))，这样括号中的子表达式就可以在替换字符串中以 ${match[1]} 的形式出现。 替换字符串带有引号，因此不会在文件名生成之前替换参数。

以下 f、F、w 和 W 修饰符仅用于参数扩展和文件名生成。 在此列出它们是为了给所有修饰符提供一个单一的参考点。

f

重复紧跟修饰符（不带冒号）的词，直到结果词不再变化。

F:*expr*:

与 f类似，但如果表达式 *expr* 的值为 *n*，则只重复 *n* 次。 可以使用任何字符代替 ':'；如果使用'(', '[' 或 '{' 作为开头分隔符，则结尾分隔符应分别为 ')', ']', 或 '}'。

w

使紧随其后的修饰符作用于字符串中的每个单词。

W:*sep*:

与 w 类似，但单词被视为字符串中由 *sep* 分隔的部分。任何字符都可以代替 ':'；开头的括号会特殊处理，见上文。

14.2 进程替换

命令参数中格式为 '<(*list*)'、 '>(*list*)' 或 '=(*list*)' 的每一部分都会进行进程替换。 表达式的前面或后面可以有其他字符串，但为了防止与通常的字符串和模式发生冲突，最后一个形式必须出现在命令参数的开头，而且只有在第一次解析命令或赋值参数时才会展开这个形式。 进程替换可以在重定向操作符之后使用；在这种情况下，替换必须不带尾部字符串。

请注意，'<<(*list*)' 并非特殊语法；它等同于 '< <(*list*)'，从进程替换的结果中重定向标准输入。 因此，以下所有文档都适用。 为清晰起见，建议使用第二种形式（带空格）。

对于 < 或 > 形式，shell 会将 *list* 中的命令作为执行 shell 命令行的作业的子进程运行。 如果系统支持 `/dev/fd` 机制，命令参数就是与文件描述符相对应的设备文件名；否则，如果系统支持命名管道（FIFOs），命令参数就是一个命名管道。 如果选择了带 > 的形式，那么写入该特殊文件将为 *list* 提供输入。 如果使用 <，那么作为参数传递的文件将连接到 *list* 进程的输出。 例如

```
paste <(cut -f1 file1) <(cut -f3 file2) |
tee >(process1) >(process2) >/dev/null
```

分别从文件 *file1* 和 *file2* 中剪切字段 1 和 3，将剪切结果粘贴在一起，然后发送给进程 *process1* 和 *process2* 。

如果使用 =(...)，而不是 <(...)，那么作为参数传递的文件将是一个包含 *list* 进程输出的临时文件的名称。 如果程序希望对输入文件进行 lseek（参见 lseek(2)），则可以用这种形式代替 < 形式。

对 =(<<<*arg*) 形式的替换进行了优化，其中 *arg* 是 here-string 重定向 <<< 的单字参数。 在执行任何替换后，这种形式将生成一个包含 *arg* 值的文件名。 这完全由当前 shell 处理。 这实际上是特殊形式 $(<*arg*) 的反向操作，后者将 *arg* 作为文件名并替换为文件内容。

= 形式非常有用，因为 `/dev/fd` 和 <(...) 的命名管道实现都有缺点。 在前一种情况下，某些程序可能会在检查命令行上的文件之前自动关闭相关的文件描述符，尤其是出于安全原因必须这样做时，例如当程序正在运行 setuid 时。 在第二种情况下，如果程序没有实际打开文件，试图从管道读取或向管道写入的子shell（在典型的实现中，不同的操作系统可能有不同的行为）将永远阻塞，必须明确地被杀死。 在这两种情况下，shell 实际上是通过管道提供信息的，因此那些希望在文件上进行 lseek（见 lseek(2)）的程序将无法运行。

另外要注意的是，前面的例子可以更简洁、更高效地写成（如果设置了 `MULTIOS` 选项）：

```
paste <(cut -f1 file1) <(cut -f3 file2) > >(process1) > >(process2)
```

shell 使用管道而不是 FIFOs 来实现上例中后两个进程的替换。

>(*process*)还有一个问题；当它附加到外部命令时，父 shell 不会等待 *process* 结束，因此紧随其后的命令无法依赖于完整的结果。 问题和解决方案与 重定向 中 ***MULTIOS*** 部分所述相同。 因此，在上述示例的简化版本中：

```
paste <(cut -f1 file1) <(cut -f3 file2) > >(process)
```

(注意不涉及 MULTIOS），则 *process* 将在父 shell 中异步运行。 解决方法是

```
{ paste <(cut -f1 file1) <(cut -f3 file2) } > >(process)
```

这里的额外进程是从父 shell 生成的，父 shell 会等待进程完成。

另一个问题出现在 shell 分离（disowned）需要临时文件的带有替换的作业时，包括在包含替换的命令末尾出现 '&!' 或 '&|' 的情况。 在这种情况下，临时文件将不会被清理，因为 shell 已不再拥有该任务的任何内存。 解决方法是使用子 shell，例如

```
(mycmd =(myoutput)) &!
```

因为分叉(forked)的子shell会等待命令执行完毕，然后删除临时文件。

要确保进程替换能持续适当的时间，一般的变通方法是将其作为参数传递给匿名 shell 函数（一段 shell 代码，在函数作用域下立即运行）。 例如，这段代码

```
() {
   print File $1:
   cat $1
} =(print This be the verse)
```

输出类似下面的内容

```
File /tmp/zsh6nU0kS:
This be the verse
```

进程替换创建的临时文件将在函数退出时删除。

14.3 参数扩展

字符 '$' 用于引入参数扩展。 有关参数的描述，包括数组、关联数组以及访问单个数组元素的下标符号，请参阅 参数。

需要特别注意的是，除非设置了 SH_WORD_SPLIT 选项，否则未引用的参数的单词不会在空白处自动分隔；更多详情请参见下文对该选项的说明。 这是与其他 shell 的一个重要区别。 不过，和其他 shell 一样，空null字会从未引用的参数的扩展中省略。

在分配之后，使用默认选项：

```
array=("first word" "" "third word")
scalar="only word"
```

则 $array 替换两个单词 'first word' 和 'third word' ，而 $scalar 替换一个单词 'only word'。 注意 array 的第二个元素已被省略。 如果标量参数的值为 null（空），也可以省略。 为避免省略，请使用如下引用："$scalar" 用于标量，"${array[@]}" 或 "${(@)array}" 用于数组。 (后两种形式是等价的）。

参数扩展可以涉及 ***flags***，像在 '${(@kv)aliases}'中，和其他操作符，如 '${PREFIX:-"/usr/local"}'。 参数扩展也可以嵌套。 下文将介绍这些主题。 完整的规则比较复杂，将在最后指出。

在下面讨论的需要模式的扩展中，模式的形式与文件名生成时使用的形式相同；参见 文件名生成。 需要注意的是，这些模式以及任何替换的替换文本本身都可以进行参数扩展、命令替换和算术扩展。 除以下操作外，还可以使用 历史扩展 中 修饰符 所描述的冒号修饰符：例如，${i:s/foo/bar/} 对参数 $i 的扩展执行字符串替换。

在以下描述中，'*word*' 指的是命令行中替换的单个单词，不一定是空格分隔的单词。

${*name*}

> 参数 *name* 的值将被替换（如果有）。 如果扩展名后的字母、数字或下划线不能被解释为 *name* 的一部分，则需要使用大括号。 此外，更复杂的替换形式通常需要使用大括号；只有在未设置选项 KSH_ARRAYS 的情况下才适用的例外情况是：在名称后面出现单个下标或任何冒号修饰符，或在名称前面出现任何字符^', '=', '~', '#' 或 '+'，无论是否使用大括号，这些情况都可以工作。
>
> 如果 *name* 是一个数组参数，且未设置 KSH_ARRAYS 选项，那么 *name* 中每个元素的值都会被替换，每个字替换一个元素。 否则，扩展只产生一个字；如果设置 KSH_ARRAYS，则是数组的第一个元素。 除非设置了 SH_WORD_SPLIT 选项，否则不会对结果进行字段分割。 另请参阅 = 和 s:*string*: 标志。

${+*name*}

> 如果 *name* 是一个集合参数（a set parameter，一个已设置的参数？）的名称，则替换为 '1'，否则替换为 '0' 。

${*name*-*word*}
${*name*:-*word*}

> 如果 *name* 已设置，或在第二种形式中为非空，则替换其值；否则替换 *word*。 在第二种形式中，*name* 可以省略，在这种情况下，*word* 总是被替换。

${*name*+*word*}
${*name*:+*word*}

> 如果 *name* 已设置，或第二种形式为非空，则替换 *word*；否则不替换任何内容。

${*name*=*word*}
${*name*:=*word*}
${*name*::=*word*}

在第一种形式中，如果 *name* 未设置，则将其设置为 *word*；在第二种形式中，如果 *name* 未设置或为空，则将其设置为 *word*；在第三种形式中，无条件地将 *name* 设置为 *word*。 在所有形式中，参数值都会被替换。

${*name*?*word*}
${*name*:?*word*}

在第一种形式中，如果 *name* 已设置，或在第二种形式中，如果 *name* 设置且非空，则替换其值；否则，打印 *word* 并退出 shell。 交互式 shell 则返回提示符。 如果省略 *word*，则打印标准信息。

在上述任何一个测试变量并替换可选的 *word* 的表达式中，请注意可以在 *word* 值中使用标准 shell 引用，有选择性地覆盖 SH_WORD_SPLIT 选项和 = 标记的分割，但不覆盖 s:*string*: 标记的分割。

在下面的表达式中，如果 *name* 是数组且替换未加引号，或者使用了 '(@)' 标志或 *name*[@] 语法，则会对每个数组元素分别执行匹配和替换。

${*name*#*pattern*}
${*name*##*pattern*}

如果 *pattern* 与 *name* 值的开头部分匹配，则替换 *name* 值并删除匹配部分；否则，只替换 *name* 值。 在第一种形式中，优先使用匹配度最小的模式；在第二种形式中，优先使用匹配度最大的模式。

${*name*%*pattern*}
${*name*%%*pattern*}

如果 *pattern* 与 *name* 值的末尾匹配，则替换 *name* 值并删除匹配部分；否则，只替换 *name* 值。 在第一种形式中，优先使用匹配度最小的模式；在第二种形式中，优先使用匹配度最大的模式。

${*name*:#*pattern*}

如果 *pattern* 与 *name* 的值匹配，则替换空字符串；否则，只替换 *name* 的值。 如果 *name* 是一个数组，则会删除匹配的数组元素（使用 '(M)' 标志删除不匹配的元素）。

${*name*:|*arrayname*}

如果 *arrayname* 是数组变量的名称（注意，不是内容），那么 *arrayname* 中包含的任何元素都会从 *name* 的替换中删除。 如果是标量替换（因为 *name* 是标量变量或表达式是带引号的），则 *arrayname* 中的元素将针对整个表达式进行测试。

${*name*:**arrayname*}

与前面的替换类似，但意义相反，原始替换中的条目和作为 *arrayname* 元素的条目将保留下来，删除其他条目。

${*name*:^*arrayname*}
${*name*:^^*arrayname*}

对两个数组进行压缩，使输出数组的长度是 name 和 arrayname 中最短（对于 ':^^'，最长）数组的两倍，并交替从中提取元素。对于 ':^'，如果其中一个输入数组较长，输出将在到达较短数组的末尾时停止。 因此

```
a=(1 2 3 4); b=(a b); print ${a:^b}
```

将输出 '1 a 2 b'。 如果是 ':^^'，则重复输入，直到用完所有较长的数组，然后输出 '1 a 2 b 3 a 4 b'。

任一输入或两个输入都可以是标量，它们将被视为长度为 1 的数组，标量是唯一的元素。如果任一数组为空，则输出另一个数组，不插入任何额外元素。

目前，以下代码会将 'a b' 和 '1' 作为两个独立的元素输出，这可能会出乎意料。第二段打印提供了一个变通方法，如果修改了这一方法，它也能继续工作。

```
a=(a b); b=(1 2); print -l "${a:^b}"; print -l "${${a:^b}}"
```

${*name*:*offset*}
${*name*:*offset*:*length*}

该语法的效果类似于 $*name*[*start*,*end*] 形式的参数下标，但与其他 shell 兼容；需要注意的是，*offset* 和 *length* 的解释与下标部分不同。

如果 *offset* 为非负数，那么如果变量 *name* 是标量，则从字符串的第一个字符开始，替换 *offset* 个字符的内容；如果 *name* 是数组，则替换从第一个元素开始，*offset*个元素。 如果给定了 *length*，则替换该数量的字符或元素，否则替换整个标量或数组的其余部分。（offset是指第几个字符开始的内容？）

正数 *offset* 始终被视为 *name* 中的字符或元素从第一个字符或数组的元素开始的偏移量（这与原生的 zsh 下标符号不同）。 因此，无论选项 KSH_ARRAYS 的设置如何，0 都是指第一个字符或元素。

负偏移量从标量或数组的末尾开始向后计数，因此-1 相当于最后一个字符或元素，以此类推。

当为正数时，*length* 从 *offset* 位置开始向标量或数组的末尾计数。 负数时，*length* 从末尾开始向后（back）计数。 如果结果显示的位置小于 *offset*，则会打印诊断信息，并且不会替换任何内容。

遵从 MULTIBYTE 选项，即偏移量和长度会酌情计算多字节字符。

offset 和 *length* 会进行与标量赋值相同的 shell 替换；此外，它们还需要进行算术运算。 因此，例如

```
print ${foo:3}
print ${foo: 1 + 2}
print ${foo:$(( 1 + 2))}
print ${foo:$(echo 1 + 2)}
```

都有相同的效果，如果替换返回的是标量，则从 `$foo` 的第四个字符开始提取字符串；如果 `$foo` 返回的是数组，则从第四个元素开始提取数组。 请注意，使用 KSH_ARRAYS 选项时 `$foo` 返回的总是标量（无论是否使用偏移量语法），因此需要使用 `${foo[*]:3}` 这样的形式来提取名为 `foo` 的数组的元素。

如果 *offset* 为负数，则 `-` 不能紧接在 `:` 之后，因为这表示 `${`*name*`:-`*word*`}`的替换形式。 相反，可以在 `-` 之前插入一个空格。 此外，*offset* 和 *length* 都不能以字母或 & 开头，因为这些字符用于表示历史类型的修饰符。 要从变量中替换一个值，建议在前面加上 `$`，因为这表示了意图（参数替换很容易导致无法读取）；不过，由于执行的是算术替换，表达式 `${var: offs}` 确实有效，可以从 `$offs` 中获取偏移量。

为了进一步与其他 shell 兼容，数组偏移量 0 有一个特殊情况。 这通常会访问数组的第一个元素。 但是，如果替换指向位置参数数组，例如 `$@` 或 `$*`，则偏移量 0 指向 `$0`，偏移量 1 指向 `$1`，依此类推。 换句话说，位置参数数组实际上是通过预置 `$0` 来扩展的。 因此，`${*:0:1}` 替代 `$0`，`${*:1:1}` 替代 `$1`。

`${`*name*`/`*pattern*`/`*repl*`}`
`${`*name*`//`*pattern*`/`*repl*`}`
`${`*name*`:/`*pattern*`/`*repl*`}`

用字符串 *repl* 替换参数 *name* 扩展中可能与 *pattern* 匹配的最长字符串。 第一种形式只替换第一次出现的字符串，第二种形式替换所有出现的字符串，第三种形式仅在 *pattern* 与整个字符串匹配时才替换。 *pattern* 和 *repl* 都会进行双引号替换，因此 `${name/$opat/$npat}` 这样的表达式可以正常工作，但要遵守通常的规则，即 `$opat` 中的模式字符不会被特殊处理，除非设置了 GLOB_SUBST 选项，或者 `$opat` 被替换为 `${~opat}`。

pattern 可以以 '#' 开头，在这种情况下，*pattern* 必须匹配字符串的开头；或者以 '#' 开头，在这种情况下，*pattern* 必须匹配字符串的结尾；或者以 '#%' 开头，在这种情况下，*pattern* 必须匹配整个字符串。 *repl* 可以是空字符串，在这种情况下，也可以省略最后的 '/'。 在其他情况下，要引用最后的 '/'，应在其前面加一个反斜线；如果 '/' 出现在一个被替换的参数中，则不必这样做。 还需注意的是，'#', '%' 和 '#% 如果出现在被替换参数内，即使是在起始位置，也不会被激活。

如果在引用规则应用后，`${`*name*`}` 扩展为一个数组，那么替换将单独作用于每个元素。 请注意下面 I 和 S 参数扩展标志的作用；但 M、R、B、E 和 N 标志并无用处。

例如,

```
foo="twinkle twinkle little star" sub="t*e" rep="spy"
print ${foo//${~sub}/$rep}
print ${(S)foo//${~sub}/$rep}
```

这里，‘~’ 确保 $sub 的文本被视为模式而非普通字符串。 在第一种情况下，t*e 的最长匹配项被替换，结果为 ‘spy star’，而在第二种情况下，最短匹配项被替换，结果为 ‘spy spy lispy star’。

${#*spec*}

如果 *spec* 是上述替换之一，则替换结果的字符长度，而不是结果本身。 如果 *spec* 是数组表达式，则用结果的元素个数代替。 这样做的副作用是，即使是带引号的形式，也会跳过连接，这可能会影响 *spec* 中的其他子表达式。 请注意，下面的 ‘^’，‘=’ 和 ‘~’ 在组合这些形式时必须出现在 ‘#’ 的左边。

如果未设置选项 POSIX_IDENTIFIERS，并且 *spec* 是一个简单的名称，那么大括号是可选的；即使对于特殊参数也是如此，例如 $#- 和 $#* 分别取字符串 $- 和数组 $* 的长度。 如果设置了 POSIX_IDENTIFIERS，则需要使用大括号来处理 #。

${^*spec*}
${^^*spec*}

打开 RC_EXPAND_PARAM 选项，用于 *spec* 的求值；如果 ‘^’ 双写，则关闭该选项。设置该选项后， *foo*${*xx*}*bar*形式的数组扩展会被替换为 ‘*fooa*bar *foob*bar *fooc*bar’ ，而不是默认的 ‘*fooa b c*bar’。其中参数 *xx* 被设置为 (*a b c*)。 请注意，如果数组为空，所有参数都会被移除。

在内部，每个此类扩展都会转换为括号扩展的等效列表。 例如，${^var} 变为 {$var[1],$var[2],…}，并按照下面 括号扩展 中的描述进行处理：但请注意，扩展是立即进行的，任何显式的括号扩展都会在后面进行。 如果同时进行分词，$var[*N*] 本身可能会被分割成不同的列表元素。

${=*spec*}
${==*spec*}

在计算 *spec* 时，使用 SH_WORD_SPLIT 的规则进行分词，但无论参数是否出现在双引号中；如果 ‘=’ 是双引号，则关闭分词。 这将强制在替换前将参数扩展拆分成单独的词，使用 IFS 作为分隔符。 大多数其他 shell 默认都是这样做的。

请注意，在 *spec* 的赋值形式中,在对 *name* 进行赋值**之前**，*word* 将被拆分。 这会影响使用 A 标志的数组赋值结果。

${~*spec*}
${~~*spec*}

打开 GLOB_SUBST 选项，用于计算 *spec*；如果 '~' 双写，则关闭该选项。 设置该选项后，扩展后的字符串将在任何可能的情况下被解释为模式，例如在文件名扩展和文件名生成以及模式匹配上下文中，如条件中的 '=' 和 '!=' 操作符的右侧。

在嵌套替换中，请注意 ~ 的效果适用于当前层次的替换结果。 外围的模式操作可能会抵消该结果。 因此，举例来说，如果参数 foo 设置为 *，${~foo//*/*.c} 将被模式 *.c 取代，该模式可能会通过文件名生成进行扩展，但 ${${~foo}//*/*.c} 将取代为字符串 *.c，该字符串不会被进一步扩展。

如果使用 ${...} 类型的参数表达式或 $(...) 类型的命令替换来代替上述 *name*，则首先展开该表达式，然后将其结果作为 *name* 的值使用。 因此，可以执行嵌套操作： ${${foo#head}%tail} 将替换 $foo 的值，同时删除 'head' 和 'tail'。 带有 $(...) 的形式通常与接下来描述的标志结合使用；请参阅下面的示例。 参数扩展中的每个 *name* 或嵌套 ${...} 后面还可以跟一个下标表达式，如 数组参数 中所述。

请注意，双引号可能会出现在嵌套表达式的周围，在这种情况下，只有内部的部分才会被视为引号；例如，${(f)"$(foo)"} 引用了 $(foo) 的结果，但标记 '(f)'（见下文）是使用无引号扩展的规则。 请进一步注意，在这种情况下，引号本身是嵌套的；例如，在 "${(@f)"$(foo)"}" 中，有两组引号，一组围绕整个表达式，另一组（多余的）围绕 $(foo) 如前。

14.3.1 参数扩展标志

如果开头括号后直接跟了一个开头括号，那么直到匹配的结尾括号为止的字符串将被视为标志列表。 在重复标记有意义的情况下，重复标记不必是连续的； 例如，'(q%q%q)' 与更易读的 '(%%qqq)' 意思相同。 支持以下标志：

#

将得到的单词作为数值表达式进行计算，并将其解释为字符代码。 输出相应的字符。 请注意，这种形式与使用不带括号的 # 完全不同。

如果设置了 MULTIBYTE 选项，且数字大于 127（即不是 ASCII 字符），则会被视为 Unicode 字符。

%

以与提示符相同的方式扩展结果词中的所有 % 转义符（参见 提示符扩展）。如果两次给出该标记，则会根据 PROMPT_PERCENT、PROMPT_SUBST 和 PROMPT_BANG 选项的设置，对生成的单词进行完全的提示符扩展。

@

在双引号中，数组元素被放在单独的词中。 例如，'"${(@)foo}"' 等同于 '"${foo[@]}"'，而 '"${(@)foo[1,2]}"' 等同于 '"$foo[1]" "$foo[2]"' 相同。这与使用 f、s 或 z 标记的 **字段分割** 不同，仍适用于每个数组元素。

A

将替换转换为数组表达式，即使它本来是标量表达式。 这比下标优先级低，因此需要一级嵌套扩展才能使下标适用于数组元素。 因此，当 *name* 是标量时，`${${(A)name}[1]}` 将产生 *name* 的完整值。

这会用 '${...=...}', '${...:=...}' 或 '${...::=...}' 指定一个数组参数。 如果该标志重复出现（如 'AA'），则会分配一个关联数组参数。 赋值在排序或填充之前进行；如果激活了字段分割，*word* 部分会在赋值前被分割。 对于普通数组，*name* 部分可以是下标范围；当赋值一个关联数组时，*word* 部分 **必须** 转换为数组，例如使用 '${(AA)=name=...}' 激活字段分割。

周围环境（如额外嵌套或在标量赋值中使用值）可能会导致数组再次连接回单个字符串。

a

按数组索引顺序排序；与 '0' 结合使用时，按数组索引倒序排序。 请注意， 'a' 因此等同于默认值，但 '0a' 在按相反顺序获取数组元素时非常有用。

b

仅对模式匹配中的特殊字符使用反斜线引用。这在使用 GLOB_SUBST （包括 `${~...}` 开关测试变量内容时非常有用。

由于 GLOB_SUBST 不会从非模式字符中去除引号，因此使用 q 系列标志之一的引号无法实现这一目的。 换句话说

```
pattern=${(q)str}
[[ $str = ${~pattern} ]]
```

如果 `$str` 是 'a*b' 则有效，如果是 'a b'则无效，而

```
pattern=${(b)str}
[[ $str = ${~pattern} ]]
```

对于 `$str` 的任何可能值都是 true。

c

用 `${#name}`， 计算数组中的字符总数，就像元素之间用空格连接一样。 这并不是数组的真正连接，因此在计算之前，使用此标志的其他表达式可能会对数组元素产生影响。

C

将生成的单词大写。 这里的 'Words' 指的是由非字母数字分隔的字母数字字符序列，**不是**指字段分割产生的单词。

D

假定字符串或数组元素包含目录，并尝试用名称替换其前导部分。 路径的其余部分（如果前导部分未被替换，则为整个路径）会被加引号，这样整个字符串就可以用作 shell 参数。 这是'~'替换的反向操作：参见 文件名扩展。

e

对结果执行单字 shell 扩展，即在结果上执行 **参数扩展**、**命令替换** 和 **算术扩展**。此类扩展可以嵌套，但过深的递归可能会产生不可预知的效果。

f

在换行符处分割扩展结果。这是 'ps:\n:' 的简写。

F

使用换行符作为分隔符，将数组的单词连接在一起。 这是 'pj:\n:' 的简写。

g:*opts*:

在未给出任何选项的情况下（g::），像 echo 内置程序一样处理转义序列。 使用 o 选项时，八进制转义字符不带前导零。 使用 c 选项时，类似 '^X' 的序列也会被处理。 如果使用 e 选项，则会处理 '\M-t' 和类似 print 内置命令的序列。 同时使用 o 和 e 选项时，除了不解释 '\c' 以外，其行为与 print 内置程序类似。

i

不区分大小写排序。 可与 'n' 或 '0' 结合使用。

k

如果 *name* 指向关联数组，则替换 ***keys***（元素名）而不是元素的值。 与下标（包括普通数组）一起使用时，即使下标形式指的是值，也会强制替换索引或键。 但是，该标志不能与下标范围结合使用。 使用 KSH_ARRAYS 选项时，需要使用下标 '[*]' 或 '[@]' 对整个数组进行操作。

L

将结果中的所有字母转换为小写。

n

对十进制整数按数值排序；如果两个测试字符串的第一个不同字符不是数字，则按词法排序。 '+' 和 '-' 不会被特殊处理；它们与其他非数字一样被处理。 前导零位较多的整数排序在前导零位较少或没有前导零位的整数之前。 因此，数组 'foo+24 foo1 foo02 foo2 foo3 foo20 foo23' 的排列为所示顺序。 可与 'i' 或 '0' 结合使用。

-

与 n相同，但前导负号表示十进制负整数。 前面的负号后面没有整数，不会触发数字排序。 请注意，‘+’ 符号不会被特殊处理（这点将来可能会改变）。

o

按升序排序结果；如果该选项单独出现，则按字面和大小写排序（除非本地语言不区分大小写）。 升序排序是其他排序形式的默认排序方式，因此，如果与 ‘a’, ‘i’, ‘n’ 或 ‘-’ 一起使用，则会被忽略。

O

按降序排序； ‘O’ 不含 ‘a’, ‘i’, ‘n’ 或 ‘-’ 时，按相反的词序排序。 可与 ‘a’, ‘i’, ‘n’ 或 ‘-’ 结合使用，以颠倒排序顺序。

P

这会强制将参数 *name* 的值解释为另一个参数名，并在适当的地方使用其值。 需要注意的是，使用 typeset 系列命令之一设置的标志（特别是转换）并不适用于以这种方式使用的 *name* 值。

如果与嵌套参数或命令替换一起使用，其结果将以同样的方式作为参数名。 例如，如果有 ‘foo=bar’ 和 ‘bar=baz’，则字符串 ${(P)foo}、${(P)${foo}} 和 ${(P)$(echo bar)} 将扩展为 ‘baz’。

同样，如果引用本身是嵌套的，则带有标志的表达式将被视为直接替换为参数名。 如果这种嵌套替换产生了一个包含多个单词的数组，则属于错误。 例如，如果 ‘name=assoc’ 中的参数 assoc 是一个关联数组，那么 ‘${${(P)name}[elt]}’ 指的就是关联下标 ‘elt’ 中的元素。

q

使用反斜线引用结果字中对 shell 有特殊意义的字符；使用 $'*NNN*' 格式引用无法打印或无效的字符，每个八位位组使用单独的引号。

如果两次使用该标记，生成的单词将用单引号引用；如果三次使用该标记，生成的单词将用双引号引用；在这些形式中，不会对无法打印或无效的字符进行特殊处理。 如果标记出现四次，则以单引号引用，前面加上 $。 需要注意的是，在所有这三种形式中，引用都是无条件执行的，即使这并不改变 shell 解释字符串的方式。

如果给出 q-（只能出现单个 q），则会使用最小形式的单引号，仅在需要保护特殊字符时对字符串加引号。 通常，这种形式的输出最易读。

如果给出 q+，将使用扩展的最小引号形式，使用 $'...' 呈现无法打印的字符。 这种引号与 typeset 系列命令输出值时使用的引号类似。

Q

从得到的单词中删除一级引号。

t

使用描述参数类型的字符串，参数值通常会出现在该字符串中。该字符串由以连字符（‘-’）分隔的关键字组成。字符串中的第一个关键字描述主要类型，可以是‘scalar’, ‘array’, ‘integer’, ‘float’ 或 ‘association’。其他关键字会更详细地描述类型：

local

为局部参数

left

用于左对齐参数

right_blanks

用于右对齐参数，带前导空格

right_zeros

用于右对齐参数，带前导零

lower

用于参数值在展开时被转换为全小写的参数

upper

用于参数值在展开时被转换为全大写的参数

readonly

用于只读参数

tag

用于已标记参数

tied

用于以PATH (冒号分隔列表) 和 path (数组)方式绑定到另一个参数的参数，无论这些参数是特殊参数还是用户用 ‘typeset -T’ 定义的参数。

export

用于导出参数

`unique`

用于只保留重复值首次出现的数组

`hide`

为参数加上 'hide' 标志

`hideval`

用于带有 'hideval' 标志的参数

`special`

用于 shell 定义的特殊参数

`u`

只扩展每个唯一单词的第一次出现。

`U`

将结果中的所有字母转换为大写字母。

`v`

与 `k` 一起使用时，（以两个连续词的形式）替换每个关联数组元素的键和值。 与下标一起使用时，即使下标形式指的是索引或键，也会强制替换值。

`V`

使结果字中的任何特殊字符可见。

`w`

使用 `${#name}`，可以计算数组或字符串中的单词数；可以使用 `s` 标志设置单词分隔符。

`W`

与 `w` 类似，不同之处在于重复分隔符之间的空字也会被计算在内。

`X`

使用该标志时，将报告使用 `Q`, `e` and `#` 标志或模式匹配形式（如 '`${name#pattern}`'）出现的解析错误。 如果没有标志，错误将被忽略。

`z`

使用 shell 解析将扩展结果拆分为单词，即考虑值中的任何引号。 注释不会被特殊处理，而是作为普通字符串处理，类似于未设置 INTERACTIVE_COMMENTS 选项的交互式 shell（不过，相关选项请参见下面的 Z 标志）

需要注意的是，这个过程进行得很晚，甚至比 '(s)' 标志更晚。因此，要访问结果中的单字，请使用嵌套扩展，如 '${${(z)foo}[2]}'。同样，要删除结果单词中的引号，请使用 '${(Q)${(z)foo}}'。

0

在空字节处分割扩展结果。 这是 'ps:\0:' 的简写。

下列标志（p 除外）后跟一个或多个参数，如所示。 任何字符或匹配对 '(...)', '{...}', '[...]' 或 '<...>' 都可以代替冒号作为分隔符，但要注意的是，当一个标志有多个参数时，每个参数周围必须有一对匹配的分隔符。

p

在该参数后面，下面所述的标志，识别为 print 内置函数相同的转义序列。（不怎么通顺）

另外，使用该选项时，字符串参数的形式可以是 $*var*，在这种情况下，变量的值会被替换。 请注意，这种形式是严格的；字符串参数不会进行一般的参数扩展。

例如,

```
sep=:
val=a:b:c
print ${(ps.$sep.)val}
```

在 : 上分割变量。

~

由以下任何标志插入扩展的字符串都将被视为模式。 这也适用于 ~ 之后在同一组括号内的标志的字符串参数。 与括号外的 ~ 相比，括号外的 ~ 会强制将整个替换的字符串作为模式处理。 因此，例如

```
[[ "?" = ${(~j.|.)array} ]]
```

将 '|' 视为模式，并且只有当 $array 包含字符串 '?' 作为一个元素时才会成功。 可以重复使用 ~ 来切换行为；其效果只持续到括号中的组的末尾。

j:*string*:

使用 *string* 作为分隔符，将数组的单词连接在一起。 请注意，这发生在使用 s:*string*: 标志或 SH_WORD_SPLIT 选项进行字段分割之前。

l:*expr*::*string1*::*string2*:

将得到的单词填充到左边。 如果需要，每个单词都将被截断，并放入一个 *expr* 字宽的字段中。

参数 :*string1*: 和 :*string2*: 是可选参数；可以两个参数都不给，也可以给第一个参数，或者两个参数都给。 请注意，三个参数必须使用相同的分隔符。 左边的空格将由 *string1*（根据需要进行连接）填充，如果没有给出 *string1*，则由空格填充。 如果同时给出了*string1*和*string2*，则在使用*string1*生成任何剩余的填充之前，在每个单词的左侧直接插入一次*string2*（如有必要，则截断）。

如果 *string1* 或 *string2* 中的任何一个存在但为空，即在该点有两个分隔符，则使用 $IFS 的第一个字符来代替。

如果 MULTIBYTE 选项有效，也可以给出标志 m，在这种情况下，宽度将被用于计算填充；否则，单个多字节字符将被视为占用一个宽度单位。

如果 MULTIBYTE 选项未生效，字符串中的每个字节将被视为占用一个宽度单位。

控制字符始终被假定为一个单位宽；这使得该机制可用于生成重复的控制字符。

m

当 MULTIBYTE 选项有效时，仅与 l 或 r 标志之一或 # 长度运算符一起使用。 使用系统报告的字符宽度来计算它在字符串中的占用量或字符串的总长度。 大多数可打印字符的宽度为一个单位，但某些亚洲字符集和某些特效字使用更宽的字符；组合字符的宽度为零； 不可打印字符的宽度为零；但实际显示方式会有所不同。

如果 m 重复出现，该字符的宽度要么为零（如果宽度为零），要么为一。 对于可打印字符串，这具有计算字形（明显分开的字符）数量的效果，除非组合字符本身的宽度不为零（在某些字母表中确实如此）。

r:*expr*::*string1*::*string2*:

与 l相同，但将单词填充到右侧，并在要填充的字符串右侧插入 *string2*。

左填充和右填充可以同时使用。 在这种情况下，策略是对每个结果字的前半部分宽度使用左填充，后半部分使用右填充。 如果要填充的字符串宽度为奇数，则额外的填充会应用在左边。

s:*string*:

在分隔符 *string* 处进行字段分割。 请注意，包含两个或多个字符的 *string* 意味着所有字符必须依次匹配；这与 IFS 参数中两个或多个字符的处理方式不同。 另请参阅 = 标志和 SH_WORD_SPLIT 选项。 也可以给出一个空字符串，在这种情况下，每个字符都是一个单独的元素。

由于历史原因，对于通过拆分生成的数组，通常会将空数组元素保留在双引号内，但这一行为被禁用了，因此出现了以下情况：

```
line="one::three"
print -l "${(s.:.)line}"
```

会为 one 和 three 生成两行输出，并忽略空字段。 要覆盖这一行为，请同时提供 ‘(@)’ 标志，即 "${(@s.:.)line}"。

Z:*opts*:

与 z相同，但在后面一对分隔符之间使用选项字母组合。 如果没有选项，效果与 z 相同。 有以下选项

(Z+c+)

会将注释解析为字符串并保留；结果数组中以未加引号的注释字符开头的任何字段都是注释。

(Z+C+)

会解析并删除注释。 注释的规则是标准的：从 $HISTCHARS（默认为 #）的第三个字符开始的单词到下一个换行符之间的任何内容都是注释。

(Z+n+)

将未加引号的换行符视为普通空白，否则会将其视为 shell 代码分隔符并转换为分号。

选项在同一组分隔符内组合，例如 (Z+Cn+)。

_:*flags*:

下划线 (_) 标志保留给将来使用。 在本次修订的 zsh 中，没有有效的 *flags*；除了空的分隔符之外，下划线后面的任何内容都会被视为错误，而标志本身则没有任何作用。

以下标志对 ${...#...} or ${...%...} 形式有意义。 S, I 和 * 标志也可与 ${.../...} 形式一起使用。

S

使用 # 或 ##，搜索最靠近字符串起始位置的匹配项（‘子串匹配’）。 在特定位置的所有匹配中，# 选择最短的，而 ## 则选择最长的：

```
% str="aXbXc"
% echo ${(S)str#X*}
abXc
% echo ${(S)str##X*}
a
%
```

使用 % 或 %%，搜索最靠近字符串末尾的匹配项：

```
% str="aXbXc"
% echo ${(S)str%X*}
aXbc
% echo ${(S)str%%X*}
aXb
%
```

(请注意，% 和 %% 并不像人们想象的那样，搜索最靠近字符串末尾的匹配项）。

通过 ${…/…} or ${…//…} 进行替换时，指定非贪婪匹配，即替换最短匹配而不是最长匹配：

```
% str="abab"
% echo ${str/*b/_}
_
% echo ${(S)str/*b/_}
_ab
%
```

I:*expr*:

搜索的第 *expr* 个匹配项（其中 *expr* 的值为数字）。 这仅适用于搜索子字符串时，使用 S 标志，或 ${…/…}（仅替换第 *expr* 个匹配项）。(只替换第 *expr* 个匹配）或 ${…//…} (从 *expr* 开始的所有匹配都会被替换）。 默认情况下，取第一个匹配项。

第 *expr* 个匹配的计算方法是，从字符串的每个起始位置开始，要么有一个匹配，要么为零，尽管在全局替换时匹配可能会与之前的替换重叠，但重叠部分的替换会被忽略。 对于 ${…%…} 和 ${…%%…} 形式，随着索引的增加，匹配的起始位置会从尾部向后（backward）移动，而对于其他形式，则会从起始位置向前移动。

因此，用字符串

```
which switch is the right switch for Ipswich?
```

当 *N* 从 1 开始增加时，形式为 ${(SI:*N*:)string#w*ch} 的替换，将匹配并移除 ‘which’, ‘witch’, ‘witch’ 和 ‘wich’； 使用 ‘##’ 的形式将匹配并移除 ‘which switch is the right switch for Ipswich’, ‘witch is the right switch for Ipswich’, ‘witch for Ipswich’ 和 ‘wich’。使用 ‘%’ 的形式将删除与 ‘#’ 相同的匹配，但顺序相反；使用 ‘%%’ 的形式将删除与 ‘##’ 相同的匹配，但顺序相反。

*

启用 EXTENDED_GLOB 以通过 ${…/…} or ${…//…} 进行替换。 请注意，‘**’ 不会禁用扩展glob。

B

在结果中包含匹配起始的索引。

E

在结果中包含匹配结束后一个字符的索引（注意这与参数索引的其他用法不一致）。

M

在结果中包含匹配的部分。

N

在结果中包含匹配长度。

R

在结果中包含未匹配的部分（***R***est）。

14.3.2 Rules

以下是替换规则的摘要；假设替换周围有大括号，即 ${…}。 下面给出了一些特殊的示例。 请注意，Zsh 开发小组对阅读以下规则时可能造成的任何脑损伤概不负责。

1. **嵌套替换**

   如果存在多个嵌套的 ${…} 形式，则从内向外进行替换。 在每一层中，替换都会考虑当前值是标量还是数组，整个替换是否使用双引号，以及为当前替换层提供了哪些标志，就像嵌套替换是最外层一样。 这些标志不会向上传播到外层替换；嵌套替换将根据标志返回标量或数组，并可能根据引号进行调整。 下面的所有步骤都将在各级替换中进行。

   请注意，除非存在‘(P)’标记，否则标记和任何下标都直接应用于嵌套替换的值；例如，扩展 ${${foo}} 的行为与 ${foo} 完全相同。 当嵌套替换中出现‘(P)’标记时，其他替换规则将在值被解释为名称**之前**应用到值，因此 ${${(P)foo}} 可能与 ${(P)foo} 不同。

   在每一级嵌套替换中，被替换的单词都会进行所有形式的单字替换（即不生成文件名），包括命令替换、算术扩展和文件名扩展（即前导 ~ 和 = ）。 例如， ${${:-=cat}:h} 扩展到 cat 程序所在的目录。 (解释：内部替换没有参数，只有一个缺省值 =cat，该值通过文件名扩展扩展为完整路径；外部替换应用修饰符 :h，并获取路径中的目录部分）。

2. **内部参数标志**

任何由 typeset 系列命令之一设置的参数标志，特别是用于填充和大写的 -L, -R, -Z, -u 和 -l 选项，都会直接应用于参数值。 请注意，这些标志是命令的选项，例如 'typeset -Z'；它们与参数替换中使用的标志不同。

在替换的最外层，'(P)' 标志（规则 4.）会忽略这些转换，并使用未修改的参数值作为要替换的名称。 这通常是我们所希望的行为，因为填充可能会使值作为参数名在语法上不合法，但如果需要更改大写字母，则应使用 ${${(P)foo}} 形式（规则 25.）。

3. **参数下标**

如果数值是带有下标的原始参数引用，例如 ${*var*[3]}，下标的作用将直接应用于参数。 下标从左到右依次求值；后续下标适用于前一个下标产生的标量或数组值。 因此，如果 var 是一个数组，${var[1][2]} 就是第一个单词的第二个字符，而 ${var[2,4][2]} 就是整个第三个单词（原始数组中第二到第四个单词范围内的第二个单词）。 可以出现任意数量的下标。 '(k)' 和 '(v)' 等标志可用来改变下标结果。

4. **参数名替换**

仅在嵌套的最外层，才使用 '(P)' 标记。 这会将迄今为止的值视为参数名（可能包括下标表达式），并用相应的值替换。 如果 '(P)' 标志出现在嵌套替换中，替换将在后面进行。

如果当前值命名的参数有内部标志（规则 2.），则替换后的新值将应用这些内部标志。

5. **双引号连接**

如果处理后的值是一个数组，且替换出现在双引号中，且当前层级既没有 '(@)' 标志，也没有 '#' l长度操作符，那么值的字将被连接，每个字之间使用参数 $IFS 的第一个字符（默认为空格）（单字数组不会被修改）。 如果存在 '(j)' 标志，则将使用该标志代替 $IFS 进行连接。

6. **嵌套下标**

此时，将根据值是数组还是标量，对任何剩余的下标（即嵌套替换的下标）进行计算。 与 3. 一样，可以出现多个下标。 请注意，${foo[2,4][2]} 等价于 ${${foo[2,4]}[2]}，也等价于 "${${(@)foo[2,4]}[2]}"（嵌套替换在这两种情况下都返回一个数组），但不等价于 "${${foo[2,4]}[2]}"（由于引号的存在，嵌套替换返回一个标量）。

7. **修饰符**

任何修饰符，如尾部的'#', '%', '/'（可能双写）或一组形式为 ':...' 的修饰符（见 历史扩展 中的 修饰符）所指定的，都会应用到这一层的值的字词上。

8. **字符求值**

任何 '(#)' 标记都会被应用，并将目前的结果作为一个字符进行数字求值。

9. **长度**

任何初始 '#' 修饰符，即 ${#var} 形式的修饰符，都会被用来计算表达式的长度。

10. **强制连接**

如果存在 '(j)' 标志，或没有 '(j)' 标志，但字符串要按规则 11. 进行拆分，且未按规则 5. 进行连接，则使用给定的字符串或（如果没有给定字符串）$IFS 的第一个字符连接值中的任何单词。 请注意，'(F)' 标志隐含地提供了一个字符串，以便以这种方式连接。

11. **简单的单词拆分**

如果存在 '(s)' 或 '(f)' 标志之一，或存在 '=' 指示符（例如 ${=var}）），则在出现指定字符串或（对于两个标志均不存在的 =）$IFS中的任何字符时分割单词。

如果没有给出 '(s)', '(f)' 或 '=' ，但单词没有加引号，且设置了选项 SH_WORD_SPLIT，则会在出现 $IFS 中的任何字符时分割单词。 请注意，这一步骤也会在嵌套替换的各级进行。

12. **大小写修改**

任何由 '(L)', '(U)' 或 '(C)' 标志之一产生的大小写修改都会被应用。

13. **转义序列替换**

首先执行 '(g)' 标志中的任何替换，然后应用 '(%)' 标志系列中的任何提示符样式格式。

14. **引号应用**

任何使用 '(q)' 和 '(Q)' 及相关标志的引号或反引号（unquoting）都会被应用。

15. **目录命名**

任何使用 '(D)' 标志的目录名替换都会被应用。

16. **可视化增强**

任何使用 '(V)' 标志使字符可见（ visible ）的修改都会被应用。

17. **词性拆分**

如果存在 '(z)' 标志或 '(Z)' 标记的其中一种形式，单词将像 shell 命令行一样被拆分，因此引号和其他元字符被用来决定什么是单词。 请注意，这种形式的分词与规则 11. 所描述的分词完全不同：它不使用 $IFS，也不会导致强制连接。

18. **唯一性**

如果结果是一个数组，且存在 '(u)' 标记，则会从数组中删除重复元素。

19. **排序**

如果结果仍然是一个数组，并且存在 '(o)' 或 '(O)' 标志，数组将被重新排序。

20. RC_EXPAND_PARAM

此时，将决定是否按照 RC_EXPAND_PARAM 选项或 '^' 标志的指示，将数组元素与周围的文本逐一组合。

21. **重新求值**

任何 '(e)' 标记都会应用于该值，从而迫使它在进行新参数替换以及命令和算术替换时被重新检查。

22. **填充**

使用 '(l.*fill*.)' 或 '(r.*fill*.)' 标志对值进行填充。

23. **语义连接**

在扩展语义要求只产生一个单词的情况下，所有单词之间都会用 IFS 的第一个字符重新连接。 因此，在 '${(P)${(f)lines}}' 中， ${lines} 的值在换行符处被分割，但在应用 '(P)' 标记之前必须再次连接。

如果不需要单个单词，则跳过该规则。

24. **空参数移除**

如果替换不在双引号中出现，任何由此产生的零长度参数，无论是来自标量还是数组元素，都会从插入命令行的参数列表中删除。

严格来说，移除发生在后面，因为其他形式的替换也是如此；这里需要注意的是，移除发生在上述任何参数操作之后。

25. **嵌套参数名替换**

如果存在 '(P)' 标记，且规则 4. 尚未应用，则迄今为止的值将被视为参数名（可能包括下标表达式），并替换为相应的值，内部标记（规则 2.）将应用于新值。

14.3.3 示例

标记 f 用于逐行分割双引号替换。 例如，${(f)"$(<*file*)"} 会替换 *file* 中被分割的内容，这样每一行都是结果数组中的一个元素。 与 $(<*file*) 单独使用的效果相比， $(<*file*) 将文件按单词分割，而在双引号内使用同样的效果，这将文件的全部内容变成一个字符串。

下面说明了嵌套参数展开的规则。 假设 $foo 包含数组 (bar baz)：

"${(@)${foo}[1]}"

结果是 b。 首先，内部替换 "${foo}" (没有数组 (@) 标志) 产生了单字结果 "bar baz"。 外层替换 "${(@)...[1]}" 检测到这是一个标量，因此 (尽管有 '(@)' 标志) 下标选择了第一个字符。

"${${(@)foo}[1]}"

这将产生结果 'bar'。 在这种情况下，内部替换 "${(@)foo}" 产生数组 '(bar baz)'。 外层替换 "${...[1]}" 会检测到这是一个数组，并选择第一个单词。 这与简单情况 "${foo[1]}" 类似。

以分词和连词的规则为例，假设 $foo 包含数组 '(ax1 bx1)'。 那么

${(s/x/)foo}

会产生 'a', '1 b' 和 '1'。

${(j/x/s/x/)foo}

会产生 'a', '1', 'b' 和 '1'。

${(s/x/)foo%%1*}

产生 'a' 和 ' b' (注意多出的空格)。 由于替换发生在连接或拆分之前，因此操作首先生成修改后的数组 (ax bx) ，连接后得到 "ax bx" ，然后拆分得到 'a', ' b' 和 ''。 最后的空字符串将被省略，因为它不在双引号中。

14.4 命令替换

在以美元符号开头的括号中括起来的命令，如 '$(...)' ，或带重音符号的引号，如 '`...`' ，将被替换为它的标准输出，并删除尾部换行符。 如果替换内容没有用双引号括起来，则使用 IFS 参数将输出内容分解成单词。

'$(cat *foo*)' 可以被速度更快的 '$(<*foo*)' 替换。 在这种情况下，*foo* 会进行单字 shell 扩展 (**参数扩展**、**命令替换**和 **算术扩展**) ，但不会执行文件名生成。

如果设置了选项 GLOB_SUBST，则任何未加引号的命令替换结果，包括刚才提到的特殊形式，都可以用于文件名生成。

14.5 算术扩展

形式为 '$[*exp*]' 或 '$((*exp*))' 的字符串会被算术表达式 *exp* 的值所替代。 在求值之前，*exp* 会进行 **参数扩展**、**命令替换** 和 **算术扩展**。 请参阅 算术求值。

14.6 括号扩展

形式为 '*foo*{*xx*,*yy*,*zz*}*bar*' 的字符串会扩展为单独的字 '*fooxxbar*', '*fooyybar*' 和 '*foozzbar*'。 保留从左到右的顺序。 该结构可以嵌套。 逗号可以加引号，以便按字面意思将其包含在单词中。

形式为 '{*n1*..*n2*}' 的表达式，其中 *n1* 和 *n2* 均为整数，会扩展为 *n1* 和 *n2* 之间的所有数字。 如果任一数字以 0 开头，所有生成的数字都会以 0 为前导填充，达到最小宽度，但对于负数，- 字符也会包含在宽度中。 如果数字按递减顺序排列，则产生的序列也将按递减顺序排列。

一个形式为 '{*n1*..*n2*..*n3*}' 的表达式，其中 *n1*、*n2* 和 *n3* 均为整数，将按上述方式展开，但只输出从 *n1* 开始的第 *n3* 个数字。 如果 *n3* 为负数，数字将以相反的顺序输出，这与简单地交换 *n1* 和 *n2* 稍有不同，因为 *n3* 的步长不能平均分割范围。 可以在三个数字中的任何一个指定零填充，在第三个数字中指定零填充可能会很有用，例如 '{-99..100..01}'，如果不能在前两个数字中的任何一个上打 0，就无法指定零填充（即填充为两个字符）。

形式为 '{*c1*..*c2*}' 的表达式，其中 *c1* 和 *c2* 为单个字符（可以是多字节字符），将扩展为内部使用的字符序列中 *c1* 至 *c2* 范围内的每个字符。 对于码位低于 128 的字符，则使用 US ASCII（这是大多数用户唯一需要的情况）。 如果中间的任何字符不可打印，将使用适当的引用使其可打印。 如果字符序列颠倒，则按相反顺序输出，例如，'{d..a}' 被替换为 'd c b a'。

如果括号表达式与上述任何形式都不匹配，则保持不变，除非设置了选项 BRACE_CCL（'brace character class' = '括号字符类' 的缩写）。 在这种情况下，它将扩展为大括号之间的单个字符列表，并按照 ASCII 字符集中的字符顺序进行排序（目前不处理多字节字符）。 该语法类似于文件名生成中的 [...] 表达式：'-' 被特殊处理以表示一系列字符，但作为第一个字符的 '^' 或 '!' 则被正常处理。 例如，'{abcdef0-9}' 扩展为 16 个词 0 1 2 3 4 5 6 7 8 9 a b c d e f。

请注意，括号扩展并不是文件名生成（globbing）的一部分；在文件名生成之前，像 */{foo,bar} 这样的表达式会被拆分成两个独立的词 */foo 和 */bar。 特别要注意的是，如果 ***either*** 与两个表达式不匹配，就会产生 'no match' 错误；这一点与 */(foo|bar) 相反，后者被视为一个单独的模式，但在其他方面具有类似的效果。

要将括号扩展与数组扩展结合起来，请参阅上文 参数扩展中描述的 ${^*spec*} 形式。

14.7 文件名扩展

每个单词都会被检查是否以未加引号的 '~' 开头。 如果是，则检查词头至 '/' 的部分，如果没有 '/'，则检查词尾部分，看是否可以用这里描述的方法之一进行替换。 如果可以，则 '~' 和被检查的部分将被替换为相应的替代值。

'~' 本身会被 $HOME 的值替换。 后跟 '+' 或 '-' 的 '~' 会分别被当前或上一个工作目录替换。

后跟数字的 '~' 会被目录堆栈中该位置的目录替换。 '~0' 等同于 '~+'，而 '~1' 是目录栈的顶层。 后跟数字的 '~+' 会被目录堆栈中该位置的目录替换。 '~+0' 等同于 '~+'， '~+1' 是目录栈的顶层。 后跟数字的 '~-' 会被从堆栈底部起多少个位置的目录替换。 '~-0' 为堆栈底部。 后面跟一个数字，PUSHD_MINUS 选项可以交换 '~+' 和 '~-' 的效果。

14.7.1 动态命名目录

如果函数 zsh_directory_name 已存在，或者 shell 变量 zsh_directory_name_functions 已存在并包含函数名数组，那么这些函数将用于实现动态目录命名。 这些函数会依次尝试，直到其中一个返回状态为零，因此函数必须测试它们是否能处理相关情况，并返回适当的状态。

后跟未加引号的方括号里的字符串 *namstr* 的 '~' 会被视为动态目录名。 请注意，第一个未加引号的结尾方括号总是 *namstr* 的结束符。 shell 函数有两个参数：字符串 n（表示名称）和 *namstr*。 它要么将数组 reply 设置为单个元素，即与名称对应的目录，并返回状态为零（作为最后一条语句执行赋值通常就足够了），要么返回状态非零。 在前一种情况下，reply 的元素被用作目录；在后一种情况下，替换被视为失败。 如果所有函数都失败，且选项 NOMATCH 被设置，则会出现错误。

上述定义的函数还可用于查看目录是否可以转化为名称，例如在打印目录堆栈或在提示符中扩展 %~ 时。 在这种情况下，每个函数都会传递两个参数：字符串 d（表示目录）和动态命名的候选名称。 如果函数无法命名目录，则函数应返回非零状态，或者将数组 reply 设置为由两个元素组成：第一个元素是目录的动态名称（如在 '~[...]' 中显示的名称），第二个元素是要替换的目录的前缀长度。 例如，如果试用目录是 /home/myname/src/zsh，而 /home/myname/src（有 16 个字符）的动态名称是 s，那么函数会设置

```
reply=(s 16)
```

这样返回的目录名称将与目录路径部分可能的静态名称进行比较，如下所述；如果匹配的前缀长度（示例中为 16）长于任何静态名称所匹配的长度，则使用该名称。

并不要求函数同时实现 n 和 d 调用；例如，某些动态扩展形式可能不适合收缩为名称。 在这种情况下，任何带有第一个参数 d 的调用都应返回一个非零状态。

补全系统会调用‘zsh_directory_name c’，后面是数组 zsh_directory_name_functions 中的元素（如果存在）的等价调用，以补全目录的动态名称。 其代码应与 补全系统中描述的其他补全函数相同。

作为一个工作示例，下面的函数可将以 p: 字符串开头的任何动态名称扩展到 /home/pws/perforce 以下的目录。 在这个简单的例子中，目录的静态名称也同样有效。

```
zsh_directory_name() {
  emulate -L zsh
  setopt extendedglob
  local -a match mbegin mend
  if [[ $1 = d ]]; then
    # turn the directory into a name
    if [[ $2 = (#b)(/home/pws/perforce/)([^/]##)* ]]; then
      typeset -ga reply
      reply=(p:$match[2] $(( ${#match[1]} + ${#match[2]} )) )
    else
      return 1
    fi
  elif [[ $1 = n ]]; then
    # turn the name into a directory
    [[ $2 != (#b)p:(?*) ]] && return 1
    typeset -ga reply
    reply=(/home/pws/perforce/$match[1])
  elif [[ $1 = c ]]; then
    # complete names
    local expl
    local -a dirs
    dirs=(/home/pws/perforce/*(/:t))
    dirs=(p:${^dirs})
    _wanted dynamic-dirs expl 'dynamic directory' compadd -S\] -a dir
    return
  else
    return 1
  fi
  return 0
}
```

14.7.2 静态命名目录

‘~’ 后跟任何字母数字字符或下划线 (‘_’)、连字符 (‘-’)或点 (‘.’)组成的任何未包含（covered）的内容，都会作为命名目录被查找，如果找到，则用该命名目录的值替换。命名目录通常是系统中用户的主目录。 如果 ‘~’ 后面的文本是以 ‘/’开头的字符串 shell 参

数的名称，也可以定义命名目录。 请注意，目录路径中的尾部斜线将被删除（但原始参数不会被修改）。

也可以使用 hash 内置命令的 -d 选项来定义目录名。

当 shell 打印路径时（例如，在提示符中扩展 %~ 或打印目录栈时），会检查路径的前缀是否为已命名的目录。 如果有，那么前缀部分就会被替换为 '~'，然后是目录名。 在引用目录的两种方式中，使用较短的一种，即目录名或完整路径；如果两者长度相同，则使用目录名。 参数 $PWD 和 $OLDPWD 绝不会以这种方式缩写。

14.7.3 '=' 扩展

如果一个单词以未加引号的 '=' 开头，且设置了 EQUALS 选项，则该单词的其余部分将作为命令名称。 如果存在以该名称开头的命令，则会用命令的完整路径名替换该单词。

14.7.4 注意

文件名扩展在参数赋值的右侧执行，包括出现在 typeset 系列命令之后的参数。 在这种情况下，右侧将按照 PATH 参数的方式，被视为一个以冒号分隔的列表，因此，'~' 或 '=' 后跟 ':' 均可进行扩展。 所有这些行为都可以通过引用 '~'、 '=' 或整个表达式（而不仅仅是冒号）来禁用；EQUALS选项也同样受到尊守。

如果设置了选项 MAGIC_EQUAL_SUBST，任何未被引用的 shell 参数，其形式为 '*identifier=expression*'，都有资格按照上段所述进行文件扩展。 引用第一个 '=' 也会抑制这种情况。

14.8 文件名生成

如果一个单词包含 '*', '(', '|', '<', '[', or '?'字符 中的一个未加引号实例，除非 GLOB 选项未设置，否则该单词将被视为文件名生成的模式。 如果设置了 EXTENDED_GLOB 选项，'^' 和 '#' 字符也表示模式，否则 shell 不会对它们进行特殊处理。

单词会被匹配模式的经排序后的文件名列表替换。 如果没有找到匹配的模式，shell 会给出错误信息，除非设置了 NULL_GLOB 选项， 在这种情况下，单词将被删除；或者，除非 NOMATCH 选项未被设置，在这种情况下，单词将保持不变。

在文件名生成时，必须明确匹配字符 '/' ；此外，除非设置了 GLOB_DOTS 选项，否则必须明确匹配位于模式开头或 '/' 之后的 '.' 。 没有文件名生成模式与 '.' 或 '..' 文件匹配。在其他模式匹配情况下， '/' 和 '.' 不会被特殊处理。

14.8.1 Glob 操作符

*

匹配任何字符串，包括空字符串。

?

匹配任何字符。

[...]

匹配所包含（enclosed）的任何字符。 可以通过用 '-' 分隔两个字符来指定字符范围。'-' 或 ']' 可作为列表中的第一个字符进行匹配。 还有几类命名的字符，其形式为 '[:*name*:]'，含义如下。 第一组使用操作系统提供的宏来测试给定的字符组合，包括因本地语言设置而进行的任何修改，参见 ctype(3)：

[:alnum:]

字符为字母数字

[:alpha:]

字符为字母

[:ascii:]

字符为 7 位，即未设置最高位的单字节字符。

[:blank:]

该字符为空白字符

[:cntrl:]

该字符为控制字符

[:digit:]

字符为十进制数字

[:graph:]

该字符是除空格外的可打印字符

[:lower:]

字符为小写字母

`[:print:]`

字符是可打印字符

`[:punct:]`

该字符可打印，但既不是字母数字，也不是空格

`[:space:]`

该字符为空白字符

`[:upper:]`

字符为大写字母

`[:xdigit:]`

字符是十六进制数字

另一组命名类，由 shell 内部处理，对本地语言不敏感：

`[:IDENT:]`

允许该字符构成 shell 标识的一部分，例如参数名；该测试尊重 `POSIX_IDENTIFIERS` 选项

`[:IFS:]`

该字符用作输入字段分隔符，即包含在 `IFS` 参数中

`[:IFSSPACE:]`

该字符为 IFS 空白字符；参见 Shell 使用的参数中 `IFS` 的文档。

`[:INCOMPLETE:]`

匹配不完整多字节字符开头的字节。 请注意，可能会有一个以上的字节序列共同构成一个多字节字符的前缀。 要测试可能不完整的字节序列，请使用模式 '`[[:INCOMPLETE:]]*`'。 该模式永远不会匹配以有效多字节字符开头的序列。

`[:INVALID:]`

匹配不以有效多字节字符开头的字节。 请注意，这可能是不完整多字节字符的延续字节，因为由无效和不完整多字节字符组成的多字节字符串的任何部分都被视为单字节。

`[:WORD:]`

该字符被视为单词的一部分；该测试对 WORDCHARS 参数的值很敏感

请注意，方括号是在包围整个字符集的方括号之外的，因此要测试单个字母数字字符，需要使用‘[[:alnum:]]’。 已命名字符集可与其他类型一起使用，例如‘[[:alpha:]0-9]’。

[^…]
[!…]

与 […] 相似，但它匹配的字符不在给定的字符集中。

<[*x*]-[*y*]>

匹配 *x* 至 *y*（含）范围内的任意数字。 可以省略任何一个数字，使范围不受限制；因此，‘<->’ 可以匹配任何数字。 要匹配单个数字，[…] 形式更有效。

在使用与这种形式的模式相近的其他通配符时要小心；例如，<0-9>* 实际上将匹配字符串开头的任何数字， 因为‘<0-9>’ 将匹配第一个数字，而‘*’ 将匹配任何其他数字。这对不小心的人来说是个陷阱，但实际上是 "尽可能长的匹配总是成功的 "这一规则的必然结果。 可以使用‘<0-9>[^[:digit:]]*’ 等表达式来代替。

(…)

与括起来的模式匹配。 用于分组。 如果设置了 KSH_GLOB选项，那么紧接在‘(’ 之前的‘@’, ‘*’, ‘+’, ‘?’ 或" ‘!’ 将被特殊处理，详情如下。选项 SH_GLOB 可以防止以这种方式使用空括号（bare parentheses），但 KSH_GLOB 选项仍然可用。

请注意，分组不能扩展到多个目录：在分组中出现‘/’ 是错误的（这只适用于文件名生成中使用的模式）。 但有一个例外：以 (*pat*/)# 形式出现的组作为一个完整的路径段，可以匹配一系列目录。 例如， foo/(a*/)#bar 匹配 foo/bar、foo/any/bar、foo/any/anyother/bar，以此类推。

x|*y*

匹配 *x* 或 *y*。 该操作符的优先级低于其他操作符。 ‘|’ 字符必须在括号内，以免被解释为管道。 从左到右依次选择。

^*x*

(需要设置 EXTENDED_GLOB）匹配模式 *x* 以外的任何内容。 其优先级高于‘/’，因此‘^foo/bar’ 将搜索‘.’ 中除‘./foo’ 之外的目录，以查找名为‘bar’ 的文件。

x~*y*

(需要设置 EXTENDED_GLOB）匹配任何匹配 *x* 但不匹配 *y* 的模式。 它的优先级比除‘|’ 之外的任何操作符都低，因此‘*/*~foo/bar’ 将搜索‘.’ 中所有目录下的所有文件，如果有匹配，则排除‘foo/bar’。‘*foo*~*bar*~*baz*’ 可以排除多个模式。 在排除模式（*y*）中，‘/’ 和‘.’ 不会像通常的全局模式(globbing)那样被特殊处理。

x#

(需要设置 EXTENDED_GLOB）匹配模式 *x* 的零次或多次出现。 该操作符具有高优先级；‘12#’ 等同于 ‘1(2#)’，而不是 ‘(12)#’。 如果未加引号的 ‘#’ 跟在不能重复的内容后面，则属于错误；这包括空字符串、已跟在 ‘##’ 后面的模式，或 KSH_GLOB 模式中的括号（例如，‘!(*foo*)#’ 无效，必须替换为 ‘*(!(*foo*))’)。

x##

(需要设置 EXTENDED_GLOB）匹配模式 *x* 的一次或多次出现。 该操作符具有高优先级；‘12##’ 等同于 ‘1(2##)’，而不是 ‘(12)##’。 同时出现的 ‘#’ 字符不得超过两个。 (注意可能会与格式为 ‘1(2##)’ 的 glob 限定符发生冲突，因此应避免使用）。

14.8.2 类ksh Glob 操作符

如果设置了 KSH_GLOB选项，括号的效果可以通过前面的 ‘@’, ‘*’, ‘+’, ‘?’ 或 ‘!’ 来修改。 该字符不需要不加引号就能产生特殊效果，但 ‘(’ 必须不加引号。

@(…)

匹配括号中的模式。 (如 ‘(…)’）。

*(…)

匹配任意数量的出现。 (类似于 ‘(…)#’，但不支持递归目录搜索）。

+(…)

至少匹配一次出现。 (类似于 ‘(…)##’，但不支持递归目录搜索）。

?(…)

匹配零次或一次出现。 (类似于 ‘(|…)’）。

!(…)

匹配括号内表达式以外的任何内容。 (类似于 ‘(^(…))’）。

14.8.3 优先级

上述运算符的优先级为（最高）‘^’, ‘/’, ‘~’, ‘|’（最低）；其余运算符从左到右被简单地视为字符串的一部分，其中 ‘#’ 和 ‘##’ 适用于前面尽可能短的单位（即字符、 ‘?’, ‘[…]’, ‘<…>’，或一个带括号的表达式）。 如上所述，作为目录分隔符的 ‘/’ 可能不会出现在括号内，而 ‘|’ 则必须出现在括号内；在文件名生成以外的其他情况下使用的模式中（例如，在 case 语句和 ‘[[…]]’ 中的测试），‘/’ 并不特殊；在文件名模式中括号外出现的 ‘~’ 之后，‘/’ 也不特殊。

14.8.4 Globbing 标志

有多种标志会影响其右侧直至包围组的末尾或模式的末尾的任何文本；这些标志需要使用 EXTENDED_GLOB 选项。所有标志的形式都是 (#*X*)，其中 *X* 可以是以下形式之一：

`i`

大小写不敏感：模式中的大写或小写字符与大写或小写字符匹配。

`l`

模式中的小写字符匹配大写或小写字符；模式中的大写字符仍然只匹配大写字符。

`I`

区分大小写：局部反转 `i` 或 `l` 的效果。

`b`

为模式中的括号组激活反向引用；这在文件名生成中不起作用。 当一个带有激活括号组的模式被匹配时，与括号组匹配的字符串将存储在数组 `$match`中，与括号组匹配的字符串的起始索引存储在数组 `$mbegin`中，结束索引存储在数组 `$mend`中，每个数组的第一个元素对应于第一个括号组，以此类推。 这些数组与 shell 并无其他特殊关系。 索引使用与参数替换相同的约定，因此 `$mend` 和 `$mbegin` 中的元素可以用于下标；KSH_ARRAYS 选项受到尊重。 全局标志集不视为括号组；只能引用前九个有效括号。

例如,

```
foo="a_string_with_a_message"
if [[ $foo = (a|an)_(#b)(*) ]]; then
  print ${foo[$mbegin[1],$mend[1]]}
fi
```

打印‘string_with_a_message’。 请注意，第一组括号位于 (#b) 之前，不会产生反向引用。

除文件名生成外，反向引用适用于所有形式的模式匹配，但请注意，在对整个数组（如 ${*array*#*pattern*}）或全局替换（如 ${*param*//*pat*/*repl*}）执行匹配时，只有最后一次匹配的数据可用。 在全局替换的情况下，这可能仍然有用。 请参阅下面的 `m` 标志示例。

反向引用的编号严格遵循模式字符串中从左到右的开头括号顺序，但括号组可以嵌套。 对于括号后的‘#’ 或‘##’有特殊规定。 只有括号的最后一个匹配才会被记住：例如，在‘[[abab = (#b)([ab])#]]’ 中，只有最后的‘b’ 才会被存储在 `match[1]` 中。 因此，可能需要额外的括号来匹配完整的段：例如，使用‘X((ab|

cd)#)Y' 来匹配 'X' 和 'Y' 之间的整个 'ab' 或 'cd' 字符串，使用 $match[1] 的值，而不是 $match[2]。

如果匹配失败，所有参数都不会改变，因此在某些情况下可能需要事先初始化这些参数。 如果某些反向引用匹配失败–如果这些反向引用位于匹配失败的备用分支中，或者如果这些反向引用紧跟 # 且匹配次数为零–那么匹配的字符串将被设置为空字符串，开始和结束索引将被设置为-1。

有反向引用的模式匹配比没有反向引用的模式匹配稍慢。

B

停用反向引用，并从此反转 b 标记的作用。

cN,M

标志 (#cN,M) 可以在任何可以使用 # 或 ## 操作符的地方使用，但在文件名生成中的表达式 '(*/)#' 和 '(*/)##' 中除外，其中 '/' 有特殊含义； 它不能与其他 globbing 标志结合使用，如果放错位置，就会出现坏模式错误。 它等同于正则表达式中的 {N,M} 。 要求前一个字符或组匹配 N 到 M 次，包括 N 和 M 次。 格式 (#cN) 要求匹配 N 次； (#c,M) 等同于指定 N 为 0； (#cN,) 则表示对匹配次数没有最大限制。

m

设置对整个匹配字符串的匹配数据的引用；这类似于反向引用，在文件名生成中不起作用。 该标记必须在模式末尾有效，即不属于某个组。参数 $MATCH、$MBEGIN 和 $MEND 将分别设置为匹配的字符串以及字符串开始和结束的索引。 这在参数替换中最有用，否则匹配的字符串就显而易见了。

例如,

```
arr=(veldt jynx grimps waqf zho buck)
print ${arr//(#m)[aeiou]/${(U)MATCH}}
```

将所有匹配项（即所有元音）强制改为大写，打印 'vEldt jynx grImps wAqf zhO bUck'。

与反向引用不同，使用匹配引用不会影响速度，只是在所示示例中需要对替换字符串进行额外的替换。

M

停用 m 标志，因此不会创建匹配数据的引用。

a*num*

近似匹配： 在模式匹配的字符串中允许出现 *num* 个错误。 相关规则将在下一小节中介绍。

s, e

与其他标志不同，这些标志只有局部效果，而且每个标志都必须单独出现：‘(#s)’和‘(#e)’是唯一有效的形式。 ‘(#s)’标志只在测试字符串的起始处生效，而‘(#e)’标志只在测试字符串的末尾处生效；它们对应于标准正则表达式中的‘^’和‘$’。 它们适用于匹配文件名生成模式以外的模式中的路径段（在任何情况下，路径段都是单独处理的）。 例如，‘*((#s)|/)test((#e)|/)*’匹配以下任何字符串中的路径段‘test’： test, test/at/start, at/end/test, in/test/middle。

另一个用途是参数替换；例如，‘${array/(#s)A*Z(#e)}’将只删除与完整模式‘A*Z’匹配的数组中的元素。 执行此类操作的方法还有很多，但将替换操作‘/’和‘//’与‘(#s)’和‘(#e)’标志相结合，就提供了一种简单易记的方法。

请注意，‘(^(#s))’形式的断言也可以工作，即匹配字符串开头以外的任何地方，但这实际上意味着‘字符串开头零长度部分以外的任何地方’；您需要使用‘(""~(#s))’来匹配字符串开头以外的零长度部分。

q

模式匹配代码将忽略‘q’和直到 globbing 标志结尾括号内的所有内容。 这样做的目的是支持 glob 限定符的使用，见下文。 因此，‘(#b)(*).c(#q.)’模式既可用于 globbing，也可用于字符串匹配。 在前一种情况下，‘(#q.)’将被视为全局限定符，而‘(#b)’将不起作用；在后一种情况下，‘(#b)’可用于反向引用，而‘(#q.)’将被忽略。 请注意，glob 限定符中的冒号限定符也不适用于普通模式匹配。

u

如果 shell 在编译时使用了 MULTIBYTE_SUPPORT，则在判断模式中是否存在多字节字符时尊重当前的语言环境。 这将覆盖 MULTIBYTE 选项；默认行为取自该选项。 比较 U。 (提示：多字节字符通常来自 UTF-8 编码的 Unicode，但也可以使用系统库支持的任何 ASCII 扩展）。

U

所有字符都被视为单字节长。 与 u 相反。 此选项覆盖 MULTIBYTE 选项。

例如，测试字符串 fooxx 可与模式 (#i)FOOXX 匹配，但不能与 (#l)FOOXX、(#i)FOO(#I)XX 或 ((#i)FOOX)X 匹配。 字符串 (#ia2)readme 指定不区分大小写的 readme 匹配，最多有两个错误。

使用 ksh 语法进行分组时，必须同时设置 KSH_GLOB 和 EXTENDED_GLOB，左括号前应加上 @。 还要注意的是，标志不会影响 [...] 组内的字母，换句话说，(#i)[a-z] 仍然只匹配小写字母。 最后要注意的是，在检查整个路径时，由于对大小写不敏感，所以必

须搜索每个目录中所有匹配的文件，因此格式为 (#i)/foo/bar/... 的模式可能会比较慢。

14.8.5 近似匹配

在近似匹配时，shell 会对发现的错误进行计数，计数不能超过 (#a*num*) 标志中指定的数量。 可识别四种类型的错误：

1. 不同的字符，如 fooxbar 和 fooybar。
2. 字符的转置，如 banana 和 abnana。
3. 目标字符串中缺少一个字符，如模式 road 和目标字符串 rod。
4. 目标字符串中出现的额外字符，如 stove 和 strove。

因此，模式 (#a3)abcd 与 dcba相匹配，错误发生在使用第一条规则两次，第二条规则一次，将字符串分组为 [d][cb][a] 和 [a][bc][d]。

模式中的非字面部分必须完全匹配，包括字符范围中的字符：因此，(#a1)??? 通过对模式的空部分应用规则 4，可以匹配长度为 4 的字符串，但不能匹配长度为 2 的字符串，因为所有 ? 都必须匹配。 其他必须完全匹配的字符包括文件名中的首字母点（除非设置了 GLOB_DOTS 选项），以及文件名中的所有斜线，因此 a/bc 与 ab/c 之间有两个错误（斜线不能与其他字符换位）。 同样，对于模式中的非连续字符串，错误也会单独计算，因此 (ab|cd)ef 以 aebf 来说是两个错误。

当通过 ~ 操作符使用排除时，近似匹配将完全单独处理被排除的部分，并且必须单独激活。 因此，(#a1)README~READ_ME 会匹配 READ.ME，但不会匹配 READ_ME，因为尾部的 READ_ME 不会进行近似匹配。 不过，(#a1)README~(#a1)READ_ME 不会匹配任何形式为 READ?ME 的模式，因为所有此类形式现在都被排除在外。

除排除项外，只有一个总体错误计数；不过，允许的最大错误数可以局部更改，而且可以通过分组来限定。 例如，(#a1)cat((#a0)dog)fox 总共允许出现一个错误，而这个错误可能不会出现在 dog 部分，因此 (#a1)cat(#a0)dog(#a1)fox 模式是等价的。 请注意，首次发现错误的点是确定是否使用近似的关键点； 例如，(#a1)abc(#a0)xyz 不会匹配 abcdxyz，因为错误发生在 'x' 处，而此处的近似是关闭的。

整个路径段可以近似匹配，因此 '(#a1)/foo/d/is/available/at/the/bar' 允许在任何路径段中出现一个错误。 不过，这比不使用 (#a1)时的效率要低得多，因为必须扫

描路径中的每个目录，以寻找可能的近似匹配。 最好将 (#a1) 放在已知正确的路径段之后。

14.8.6 递归 Globbing

形式为 '(*foo*/)#' 的路径名部分匹配由零个或多个符合 *foo* 模式的目录组成的路径。

作为简写，'**/' 等同于 '(*/)#'；请注意，这将匹配当前目录和子目录中的文件。 因此

```
ls -ld -- (*/)#bar
```

或

```
ls -ld -- **/bar
```

进行递归目录搜索,查找 'bar' (可能包括当前目录中的文件 'bar' ）。 这种形式不遵循符号链接；另一种形式 '***/' 遵循符号链接，但在其他方面完全相同。 在同一路径段中，这两种形式都不能与其他形式的 globbing 结合使用；在这种情况下，'*' 操作符将恢复其通常的效果。

如果设置了 GLOB_STAR_SHORT 选项，还可以使用更短的形式。 在这种情况下，如果 ** 或 *** 后面没有紧跟 /，它们就会被当作同时存在 / 和 * 处理。 因此

```
setopt GLOBSTARSHORT
ls -ld -- **.c
```

等价于

```
ls -ld -- **/*.c
```

14.8.7 Glob 限定符

用于生成文件名的模式可以以括号中的限定符列表结尾。 限定符指定哪些与给定模式匹配的文件名将插入参数列表。

如果设置了选项 BARE_GLOB_QUAL，那么尾部不包含 '|' 或 '(' 字符（或 '~' ，如果是特殊符号）的括号将被视为一组 glob 限定符。 通常会被当作 glob 限定符的 glob 子表达式，例如 '(^x)'，可以通过将括号双写来强制当作 glob 模式的一部分，在这种情况下会产生 '((^x))'。

如果设置了选项 EXTENDED_GLOB，glob 限定符就有了不同的语法，即 '(#q*x*)'，其中 *x* 是与其他格式相同的任何 glob 限定符。 限定符仍必须出现在格式末尾。 不过，这种语法可以将多个 glob 限定符串联起来。 它们被视为各组标志的逻辑与。 此外，由于语法不含糊，只要表达式中包含的任何括号是平衡的，表达式就会被视为 glob 限定符；出现 '|'、'(' 或 '~'并不能否定其效果。 请注意，即使在模式末尾存在裸 glob 限定符，限定符

也会以这种形式被识别，例如，如果同时设置了两个选项，‘*(#q*)(.)’将识别可执行的常规文件；不过，为了清晰起见，最好避免使用混合语法。 请注意，在使用‘[[’形式的条件中，如果字符串末尾出现带括号的表达式（#q...），则表示应执行globbing；该表达式可能包含 glob 限定符，但如果只是（#q），也是有效的。 这不适用于模式匹配运算符的右侧，因为该语法已经具有特殊意义。

限定符可以是以下任何一种：

/

directories

F

‘full’（即非空）目录。 请注意，与之相反的（^F）会扩展到空目录和所有非目录。对于空目录，请使用（/^F）。

.

文本文件

@

符号链接

=

套接字

p

命名管道 (FIFOs)

*

可执行普通文件 (0100 or 0010 or 0001)

%

设备文件（字符或块特殊文件）

%b

块特殊文件

%c

字符特殊文件

r

所有者可读文件 (0400)

`w`

所有者可写文件 (0200)

`x`

所有者可执行文件 (100)

`A`

组可读文件 (0040)

`I`

组可写文件 (0020)

`E`

组可执行文件 (0010)

`R`

世界可读文件 (0004)

`W`

世界可写文件 (0002)

`X`

世界可执行文件 (0001)

`s`

setuid 文件 (04000)

`S`

setgid 文件 (02000)

`t`

带 sticky bit 位的文件 (01000)

`f`*spec*

访问权限与 *spec* 匹配的文件。*spec* 可以是一个八进制数，前面可选择加上'=', '+', 或 '-'。如果没有给出这些字符，则行为与 '=' 相同。八进制数描述了预期的模式位，

如果与 ‘=’ 组合，给出的值必须与文件模式完全匹配；如果与 ‘+’ 组合，至少要在文件模式中设置所给数字中的位；如果与 ‘-’ 组合，数字中的位数必须不被设置。如果不在数字的任何位置给出八进制数字，而是给出 ‘?’ ，则不会检查文件模式中的相应位，这只有在与 ‘=’ 结合使用时才有用。

如果限定符 ‘f’ 后跟有任何其他字符，则下一个匹配字符之前的任何字符（ ‘[’, ‘{’ 和 ‘<’ 分别与 ‘]’, ‘}’ 和 ‘>’ 匹配，任何其他字符则与自身匹配）都将作为逗号分隔的 *sub-spec*s 列表。每个 *sub-spec* 既可以是上述八进制数，也可以是 ‘u’, ‘g’, ‘o’ 和 ‘a’ 中的任意字符, 后接 ‘=’ 、 ‘+’ 或 ‘-’ ，后接 ‘r’, ‘w’, ‘x’, ‘s’ 和 ‘t’ 中的任意字符或一个八进制数字。第一个列表字符指定要检查的访问权限。如果给出 ‘u’ ，则使用文件所有者的访问权限；如果给出 ‘g’ ，则检查组的访问权限； ‘o’ 表示检查其他用户的访问权限； ‘a’ 表示检查所有三个组的访问权限。 ‘=’, ‘+’ 和 ‘-’ 再次说明如何检查模式，其含义与上述第一种形式相同。第二个字符列表说明了预期的访问权限： ‘r’ 表示读取权限， ‘w’ 表示写入权限， ‘x’ 表示执行文件（或搜索目录）的权限，‘s’ 表示 setuid 和 setgid 位， ‘t’ 表示 sticky 位。

因此， ‘`*(f70?)`’ 给出了所有者有读取、写入和执行权限，而其他组员没有权限的文件，与其他用户的权限无关。模式 ‘`*(f-100)`’ 给出所有所有者没有执行权限的文件，而 ‘`*(f:gu+w,o-rx:)`’ 给出所有者和组内其他成员至少有写入权限，而其他用户没有读取或执行权限的文件。

e*string*
+*cmd*

string 将作为 shell 代码执行。 当且仅当代码返回零状态（通常是最后一条命令的状态）时，文件名才会包含在列表中。

在第一种形式中， ‘e’ 之后的第一个字符将被用作分隔符，下一个匹配的分隔符之前的任何字符将被用作 *string*； ‘[’, ‘{’ 和 ‘<’ 分别匹配 ‘]’, ‘}’ 和 ‘>’，而任何其他字符则匹配自身。需要注意的是，扩展必须在 *string* 中加上引号，以防止在执行 globbing 之前被扩展。 然后，*string* 将作为 shell 代码执行。 在执行过程中，字符串 `globqual` 会被附加到数组 `zsh_eval_context` 中。

在 *string* 的执行过程中，正在测试的文件名在参数 `REPLY` 中可用；该参数可更改为一个字符串，以代替原始文件名插入到列表中。 此外，参数 `reply` 可以设置为数组或字符串，这将覆盖 `REPLY` 的值。 如果设置为数组，则后者会逐字插入命令行。

例如，假设一个目录包含一个文件 ‘`lonely`’。 那么表达式 ‘`*(e:'reply=(${REPLY}{1,2})':)`’ 将导致在命令行中插入 ‘`lonely1`’ 和 ‘`lonely2`’。 注意 *string* 的引号。

形式 +*cmd* 也有同样的效果，但 *cmd* 周围没有分隔符。 取而代之的是，*cmd* 是 + 后面最长的字母数字或下划线字符序列。 通常情况下，*cmd* 是包含相应测试的 shell 函数名称。 例如

```
nt() { [[ $REPLY -nt $NTREF ]] }
NTREF=reffile
ls -ld -- *(+nt)
```

会列出目录中所有最近修改次数多于 reffile 的文件。

d*dev*

设备 *dev* 上的文件

l[-|+]*ct*

链接数小于 *ct* (-)、大于 *ct* (+) 或等于 *ct* 的文件

U

有效用户 ID 拥有的文件

G

有效组 ID 拥有的文件

u*id*

用户 ID 为 *id* 的所拥有的文件。 否则，*id* 将指定用户名：'u' 后面的字符将作为分隔符，它与下一个匹配分隔符之间的字符串将作为用户名。 起始分隔符 '[', '{' 和 '<' 分别与最终分隔符 ']', '}' 和 '>'匹配；任何其他字符都与自身匹配。 所选文件为该用户拥有的文件。 例如，'u:foo:' 或 'u[foo]' 选择用户 'foo' 所拥有的文件。

g*id*

类似 u*id* ，但使用组 ID 或名称

a[Mwhms][-|+]*n*

（精确的）*n* 天前访问过的文件。使用 *n* 的负值（-*n*）选择最近 *n* 天内访问的文件。超过 *n* 天前访问的文件则使用 *n* 的正值（+*n*）来选择。可选的单位指定符 'M', 'w', 'h', 'm' 或 's'（例如 'ah5'）会分别导致以月（30 天）、周、小时、分钟或秒来代替天进行检查。也可以使用显式的 'd' 来表示天数。

在比较过程中，访问时间与当前时间之间的任何小数部分将被忽略。 例如，'echo *(ah-5)' 将回显最近 5 小时内访问的文件，而 'echo *(ah+5)' 将回显至少 6 小时前访问的文件，因为严格介于 5 至 6 小时之间的时间被视为 5 小时。

m[Mwhms][-|+]*n*

和文件访问限定符一样，只不过它使用的是文件修改时间。

c[Mwhms][-|+]*n*

和文件访问限定符一样，只不过它使用的是文件的 inode 更改时间。

L[+|-]n

长度小于 n 字节（-）、大于 n 字节（+）或正好 n 字节的文件。

如果该标志后直接跟了 ***size* 指定符** 'k' ('K'), 'm' ('M') 或 'p' ('P')（例如'Lk-50'），则检查将以千字节、兆字节或块（512 字节）为单位进行。 (在某些系统中，还可以使用额外的千兆字节（ 'g' 或 'G' ）和百万兆字节（ 't' 或 'T' ）说明符）。 如果使用了文件大小指定符，那么四舍五入到下一单位的文件大小等于测试大小时，文件就被视为大小 "准确 "。 因此，'*(Lm1)' 可以匹配从 1 字节到 1 兆字节（含 1 兆字节）的文件。 还要注意的是，"小于 "测试大小的文件集只包括不符合相等测试的文件；因此， '*(Lm-1)' 只匹配大小为零的文件。

^

否定后面的所有限定符

-

在使限定符作用于符号链接（默认）和符号链接指向的文件之间切换，如果有的话；如果 'stat' 系统调用的目标符号链接失败（无论失败原因如何），该符号链接本身将被视为文件

M

为当前模式设置 MARK_DIRS 选项

T

为文件名添加尾部限定符标记，类似于当前模式的 LIST_TYPES 选项（覆盖 M

N

为当前模式设置 NULL_GLOB 选项

D

设置当前模式的 GLOB_DOTS 选项

n

为当前模式设置 NUMERIC_GLOB_SORT 选项

Yn

启用短路模式：模式最多会扩展到 n 个文件名。 如果存在多于 n 个匹配项，则只考虑目录遍历顺序中的前 n 个匹配项。

在未使用 o*c* 限定符时，意味着 oN。

o*c*

指定文件名的排序方式。*c* 的下列值按以下方式排序：

n

按名称。

L

按文件的大小（长度）。

l

按链接数量。

a

按最后一次访问时间排列，最年轻者优先。

m

按最后一次修改时间排序，最年轻的先修改。

c

按最后一次改变 inode 的时间排列，最年轻的在前。

d

按目录：在每一级搜索中，子目录下的文件都会出现在当前目录下的文件之前–这最好与其他条件结合起来，例如，‘odon’ 可以对同一目录下的文件名进行排序。

N

不执行排序。

e*string*
+*cmd*

按 shell 代码排序（见下文）。

请注意，由于使用了修饰符 ^ 和 -，因此 ‘*(^-oL)’ 给出的是一个按文件大小降序排序的所有文件的列表，任何符号链接都在其后。 除非使用 oN，否则可以用多个顺序指定符来解决并列问题。

默认排序为 n（按名称），除非使用了 Y glob 限定符，在这种情况下，排序为 N（未排序）。

oe 和 o+ 是特殊情况；它们后面都有 shell 代码，分别以 e glob 限定符和 + glob 限定符分隔（见上文）。 每执行一个匹配文件的代码，都会将参数 REPLY 设置为输入文件的名称，并将 globsort 追加到 zsh_eval_context 中。 代码应以某种方式修改参数 REPLY。 返回时，将使用参数值而不是文件名作为排序字符串。 与其他排序运算符不同，oe 和 o+ 可以重复使用，但请注意，在任何 glob 表达式中出现的任何类型的排序运算符的最大数量都是 12。

O*c*

和 'o' 一样，但按降序排序；即 '*(^o*c*)' 与 '*(O*c*)' 相同，而 '*(^O*c*)' 与 '*(o*c*)' 相同；'Od' 将当前目录下的文件排在每一级搜索的子目录下的文件之前。

[*beg*[,*end*]]

指定哪些匹配的文件名应包含在返回的列表中。语法与数组下标相同。*beg* 和可选的 *end* 可以是数学表达式。与参数下标一样，它们可以是负数，以便从最后一次匹配开始反向计数。例如 '*(-OL[1,3])' 给出了三个最大文件的名称列表。

P*string*

string 将作为一个单独的单词被添加到每个 glob 匹配结果中。 *string* 的分隔方式与上述 glob 限定符 e 的参数相同。 限定符可以重复使用；单词会被分别放在前面，因此命令行中的单词顺序与 glob 限定符列表中的顺序相同。

典型的用法是在所有出现的文件名前加上一个选项；例如，模式 '*(P:-f:)' 会产生命令行参数 '-f *file1* -f *file2* ...'

如果修改器 ^ 已激活，那么 *string* 将被追加而非前置。 前置和追加是独立进行的，因此可以在同一个 glob 表达式中同时使用这两种方式；例如，编写 '*(P:foo:^P:bar:^P:baz:)' 会产生命令行参数 'foo baz *file1* bar ...'

可以合并多个列表，并用逗号分隔。如果至少有一个子列表匹配，则整个列表匹配（它们是 "或 "关系，子列表中的限定符是 "和 "关系的）。 不过，有些限定符会影响生成的所有匹配结果，与在哪个子列表中给出限定符无关。 这些限定符包括 'M', 'T', 'N', 'D', 'n', 'o', 'O' 和括号（brackets）中的下标（'[...]'）。

如果限定符列表中出现 ':'，括号中表达式的其余部分将被解释为修饰符（参见 历史扩展 中的 修饰符）。 每个修饰符必须用单独的 ':' 引入。 还要注意的是，修改后的结果不一定是现有文件。 即使没有实际执行文件名生成，也可以在任何现有文件名后面加上形式为 '(:...)' 的修饰符，但要注意的是，括号的存在会导致整个表达式受全局模式匹配选项（如 NULL_GLOB）的影响。因此

```
ls -ld -- *(-/)
```

会列出所有目录和指向目录的符号链接，而

```
ls -ld -- *(-@)
```

会列出所有断开的符号链接，而

```
ls -ld -- *(%W)
```

会列出当前目录下所有世界可写设备文件，而

```
ls -ld -- *(W,X)
```

会列出当前目录下所有世界可写或世界可执行的文件，而

```
print -rC1 /tmp/foo*(u0^@:t)
```

输出 /tmp 中以字符串 'foo' 开头的 root 所有的文件的基名，忽略符号链接，以及

```
ls -ld -- *.*~(lex|parse).[ch](^D^l1)
```

会列出所有链接数为 1 且名称包含点的文件（但不包括以点开头的文件，因为 GLOB_DOTS 已明确关闭），但 lex.c、lex.h、parse.c 和 parse.h 除外。

```
print -rC1 b*.pro(#q:s/pro/shmo/)(#q.:s/builtin/shmiltin/)
```

演示了如何将冒号修饰符和其他限定符串联起来。 首先应用普通限定符 '.'，然后按从左到右的顺序应用冒号修饰符。 因此，如果设置了 EXTENDED_GLOB，且基本模式与常规文件 builtin.pro 匹配，shell 将打印 'shmiltin.shmo'。

---

15 参数

15.1 说明

参数有一个名称、一个值和若干属性。 名称可以是字母数字字符和下划线的任意序列，也可以是单字符'*', '@', '#', '?', '-', '$' 或 '!'。 名称以字母数字或下划线开头的参数也称为 **变量**。

参数的属性决定了其值的 **类型**（通常称为参数类型或变量类型），并控制着在引用该值时可能对其进行的其他处理。 值类型可以是 **标量**（字符串、整数或浮点数）、数组（按数字索引）或 **关联**数组（按名称索引的键值对的无序集合，也称为 **哈希**）。

已命名的标量参数可以使用 ***exported***、-x 属性，将其复制到进程环境中，然后由 shell 传递给启动的任何新进程。 导出参数被称为 **环境变量**。shell 还会在启动时 **导入** 环境变

量，并自动将相应参数标记为导出参数。 出于安全考虑，或由于某些环境变量会干扰其他 shell 功能的正确运行，这样的变量不会导入。

参数也可以是 **特殊的**，即它们对 shell 有预定义意义。 特殊参数的类型不能更改，其只读属性也不能关闭，如果一个特殊参数被取消设置，随后又重新创建，其特殊属性将被保留。

要声明参数的类型，或为标量参数赋于字符串或数值，请使用 `typeset` 内置命令。

标量参数的值也可以通过下面写法来分配：

name=*value*

在标量赋值中，*value* 会展开为单个字符串，其中数组元素被连接在一起；除非设置了选项 `GLOB_ASSIGN`，否则不会执行文件名展开。

为 *name* 设置整数属性 `-i` 或浮点属性 `-E` 或 `-F` 时，*value* 将进行算术求值。 此外，将 '=' 替换为 '+='，还可以递增或追加参数。 有关赋值的其他形式，请参阅 数组参数 和 算术求值。

请注意，赋值可能会隐含地改变参数的属性。 例如，在算术运算中为变量赋一个数值可能会将其类型改为整型或浮型，而用 `GLOB_ASSIGN`选项， 为变量赋值一个模式，可能会将其类型改为数组。

要引用一个参数的值，请写成 '$*name*' 或 '${*name*}'。 详情请参见 参数扩展。 该章节还解释了标量赋值和数组赋值之间的差异对参数扩展的影响。

15.2 数组参数

要为数组赋值，请写入以下其中一个：

`set -A` *name value* ...

name=(*value* ...)

name=([*key*]=*value* ...)

如果不存在参数 *name*，则会创建一个普通数组参数。 如果参数 *name* 存在且是标量，则会被一个新数组取代。

在第三种形式中，*key* 是一个将在算术上下文中求值的表达式（最简单的形式是一个整数），它给出了赋值为 *value* 的元素的索引。 在这种形式中，在赋值的最大索引之前的任何未明确提及的元素都将被赋值为空字符串。 索引可以是任意顺序。 请注意，这种语法是严格的： `[` 和 `]=` 不得加引号，并且 *key* 不得包含未加引号的字符串 `]=`，否则视为简单字符串。 下面 '数组下标 ' 一节中描述的下标表达式增强形式在直接下标变量名时不可用。

有显式键和无显式键的语法可以混合使用。 隐式 *key* 是通过递增先前赋值元素的索引来推导的。 请注意，如果后面的赋值覆盖了前面的赋值，不会被视为错误。

例如，假设未设置选项 KSH_ARRAYS，则会出现以下情况：

```
array=(one [3]=three four)
```

会使数组变量 array 依次包含四个元素 one、空字符串、three 和 four。

在只指定 *value* 的形式中，将执行完整的命令行扩展。

在 [*key*]=*value* 形式中，*key*和*value*都会经历单字 shell 扩展所允许的所有扩展形式（不包括文件名生成）；这些扩展由参数扩展标志（e）执行，如 参数扩展 中所述。*value*周围可能会有嵌套的小括号，这些小括号将作为值的一部分包含在内，并被连接成一个纯字符串；这与允许值本身为数组的 ksh 不同。 未来版本的 zsh 可能会支持这一点。 要将括号解释为文件名生成的字符类，并因此将生成的文件列表视为一组值，请使用任何形式的引号为等号加引号。 例如

```
name=([a-z]'='*)
```

要在不改变现有值的情况下追加到数组，请使用以下方法之一：

```
name+=(value ...)
```

```
name+=([key]=value ...)
```

在第二种形式中，*key* 可以指定一个现有的索引，也可以指定旧数组末尾（之后 off the end of）的索引；任何现有值都会被 *value* 覆盖。 此外，还可以使用 [*key*]+=*value* 追加到该索引处的现有值。

在任何一种赋值形式右侧的括号内，换行符和分号都与空白一样，用于分隔各个 *value*s。任何连续的此类字符序列都具有相同的效果。

普通数组参数也可以明确声明用：

```
typeset -a name
```

必须在赋值前声明关联数组 ，方法是使用：

```
typeset -A name
```

当 *name* 指向关联数组时，赋值中的列表被解释为交替的键和值：

```
set -A name key value ...
```

```
name=(key value ...)
```

```
name=([key]=value ...)
```

请注意，在任何给定的赋值中，只能使用上述两种语法中的一种；这两种形式不能混合使用。 这与数字索引数组的情况不同。

在这种情况下，每个 *key* 都必须有一个 *value*。 需要注意的是，这样做会对整个数组赋值，删除列表中没有出现的元素。追加语法也可用于关联数组：

name+=(*key value* ...)

name+=([*key*]=*value* ...)

如果键还不存在，则添加新的键/值对，如果存在，则替换现有键的值。 在第二种形式中，也可以使用 [*key*]+=*value* 来追加到该键的现有值。 如上所述，扩展的执行方式与普通数组的相应形式相同。

要创建一个空数组（包括关联数组），请使用以下命令之一：

`set -A` *name*

name=()

15.2.1 数组下标

使用下标可以选择数组中的单个元素。 形式为 '[*exp*]' 的下标将选择单个元素 *exp*，其中 *exp* 是一个算术表达式，它将被算术展开，就像它被 '$((...))' 包围一样。 元素的编号从 1 开始，除非设置了 KSH_ARRAYS 选项，在这种情况下，元素的编号从 0 开始。

下标可以用在大括号内，用于限定参数名，因此 '${foo[2]}' 等同于 '$foo[2]'。 如果设置了 KSH_ARRAYS 选项，则括号形式是唯一有效的形式，否则括号内的表达式不会被视为下标。

如果没有设置 KSH_ARRAYS 选项，那么默认情况下，访问下标值为 0 的数组元素时，会返回空字符串，而尝试写入这样的元素则会被视为错误。 为了向后兼容，可以设置 KSH_ZERO_SUBSCRIPT 选项，使下标值 0 和 1 等价；参见 选项说明 中的选项描述。

除了 *exp* 不进行算术扩展外，关联数组使用了相同的下标语法。然而，算术表达式的解析规则仍然适用，这影响了某些特殊字符必须受到保护以免被解释的方式。 详见下面的 **下标解析**。

形式为 '[*]' 或 '[@]' 的下标计算为数组的所有元素；除了出现在双引号内时，两者没有区别。 '"$foo[*]"' 的运算结果为 '"$foo[1] $foo[2] ..."'，而 '"$foo[@]"' 的运算结果为 '"$foo[1]" "$foo[2]" ...'。 对于关联数组，'[*]' 或 '[@]' 计算为所有值，没有特定顺序。 请注意，这并不能替代键值；详情请参见 参数 下的 'k' 标志文档。 当一个数组参数被引用为'$*name*'（不带下标）时，其值将被计算为 '$*name*[*]'、 除非设置了 KSH_ARRAYS选项，在这种情况下，它的值为 '${*name*[0]}'（对于关联数组，这意味着键'0' 的值，这可能不存在，，即使其他键的值存在）。

形式为 '[*exp1*,*exp2*]' 的下标会选择 *exp1* 至 *exp2* 范围内的所有元素，包括 *exp1* 和 *exp2* 。(关联数组是无序的，因此不支持范围。) 如果其中一个下标求值为负数，例如 -*n*，则使用数组末尾的第 *n* 个元素。 因此，'$foo[-3]' 是数组 foo 末尾的第三个元素，而 '$foo[1,-1]' 与 '$foo[*]' 相同。

下标运算也可以用于非数组值，此时下标指定了要提取的子字符串。例如，如果FOO被设置为 'foobar' ，那么 'echo $FOO[2,5]' 将输出 'ooba'。请注意，下面描述的某些形式的下标运算执行模式匹配，在这种情况下，子字符串从第一个下标的匹配开始，一直延伸到第二个下标的匹配结束。例如，

```
string="abcdefghijklm"
print ${string[(r)d?,(r)h?]}
```

会打印 'defghi'。 这显然是对单字符匹配规则的概括。 对于单个下标，只引用单个字符（而不是匹配所覆盖的字符范围）。

请注意，在子串操作中，r 和 R 下标标志对第二个下标有不同的处理方式：前者以最短的匹配值作为长度，后者以最长的匹配值作为长度。 因此，在前一种情况下，末尾的 * 是多余的，而在后一种情况下，它匹配字符串的整个剩余部分。 这不会影响单个下标情况下的结果，因为这里匹配的长度无关紧要。

15.2.2 数组元素赋值

在赋值的左侧可以使用下标，如下所示：

name[*exp*]=*value*

在这种赋值形式中，*exp* 指定的元素或范围会被右侧的表达式替换。 数组（但不是关联数组）可以通过对元素或范围赋值来创建。 数组不会嵌套，因此给元素或范围赋值一列括号中的值会改变数组中元素的数量，并移动其他元素以容纳新值。 (关联数组不支持这种方式）。

该语法也可作为 typeset 命令的参数：

typeset "*name*[*exp*]"=*value*

在这种情况下，*value* 可以 **不**是一个带括号的列表；只能使用 typeset 进行单元素赋值。 请注意，在这种情况下必须使用引号，以防止括号被解释为文件名生成操作符。 可以使用 noglob 前置命令修饰符代替。

要删除普通数组中的一个元素，请为该元素赋值 '()'。 要删除关联数组的元素，请使用 unset 命令：

unset "*name*[*exp*]"

15.2.3 下标标志

如果在任何下标表达式中，开头括号（bracket)或范围中的逗号，后直接跟了一个开头小括号(parenthesis)，则直到匹配的结尾括号的字符串将被视为一个标志列表，如在 '*name*[(*flags*)*exp*]' 中。

标志 s、n 和 b 都有一个参数；分隔符如下所示：':'，但可以使用任何字符或匹配对 '(...)', '{...}', '[...]' 或 '<...>'。 但要注意的是，'<...>' 只能在下标位于双引号表达式或括号内的参数替换中时使用，否则表达式将被解释为重定向。

目前可理解的标志有:

`w`

如果下标的参数是标量，则该标志会使下标作用于单词而不是字符。 默认的单词分隔符是空白。 当与 `i` 或 `I` 标志结合使用时，效果是生成与给定模式匹配的第一个/最后一个单词的第一个字符的索引；请注意，在这种情况下，匹配失败的结果总是 0。

`s:`*string*`:`

这给出了分隔单词的 *string*（与 `w` 标志一起使用）。 分隔符：是任意的，见上文。

`p`

在随后的 's' 标志的字符串参数中，识别与 `print` 内置命令相同的转义序列。

`f`

如果下标的参数是一个标量，那么该标志会使下标作用在行上，而不是字符上，也就是用换行符分隔元素。这是 '`pws:\n:`' 的简写。

`r`

反向（reverse)下标：如果给出此标记，*exp* 将作为模式，结果是第一个匹配的数组元素、子串或单词（如果参数是数组，如果参数是标量，或者如果参数是标量且给出了 'w' 标志）。使用的下标是匹配元素的编号，因此，如果参数不是关联数组，则可以使用成对的下标，如 '`$foo[(r)??,3]`' 和 '`$foo[(r)??,(r)f*]`'。 如果参数是关联数组，则只将每一对的值部分与模式进行比较，结果就是该值。

如果对普通数组的搜索失败，搜索会将下标设置为数组末尾之后的一个，因此 `${array[(r)`*pattern*`]}` 将替换为空字符串。因此，可以使用 `(i)` 标志来测试搜索是否成功，例如（假设选项 KSH_ARRAYS 未生效）：

```
[[ ${array[(i)pattern]} -le ${#array} ]]
```

如果 KSH_ARRAYS 有效，则 `-le` 应替换为 `-lt`。

`R`

类似于 'r'，但给出最后一个匹配项。对于关联数组，会给出所有可能的匹配结果。可用于普通数组元素的赋值，但不能用于关联数组的赋值。 对于普通数组，赋值失败后会返回下标 0 对应的元素；除非使用b KSH_ARRAYS 或 KSH_ZERO_SUBSCRIPT 选项之一，否则返回的元素为空。

需要注意的是，在同时包含 'r' 和 'R' 的下标中，模式字符即使被替换为参数，也是有效的（与在正常模式匹配中控制该功能的 GLOB_SUBST 的设置无关）。可以添加标志 'e' 来抑制模式匹配。 由于该标志并不抑制其他形式的替换，因此仍需小心谨慎；使用参数来保持关键字（key）可以达到预期效果：

```
key2='original key'
print ${array[(Re)$key2]}
```

i

类似于 'r'，但给出的是匹配的索引；不能与第二个参数一起使用。 在赋值的左侧，行为类似于 'r'。 对于关联数组，每个键值对的键部分都会与模式进行比较，找到的第一个匹配键就是结果。 如果失败，则用数组的长度加一代替，这在'r'的描述中已经讨论过，如果是关联数组，则是空字符串。

注意：虽然 'i' 可以应用于标量替换，以查找子串的偏移量，但当在产生空字符串的替换中查找，或查找空子串时，结果很可能会产生误导。

I

与 'i' 类似，但会给出最后一个匹配项的索引，或关联数组中所有可能匹配的键。如果失败，则替换为 0 或对于关联数组是空字符串。 在测试不存在的值或键时，最好使用该标志。

注意：如果选项 KSH_ARRAYS 有效，但未找到匹配，则结果与数组中第一个元素匹配的情况无异。

k

如果在关联数组的下标中使用该标志，则键会被解释为模式，并返回键与 *exp* 匹配的第一个键的值。 请注意，这可能是任何键，因为关联数组没有定义排序。 该标志不适用于关联数组元素赋值的左侧。 如果使用在其他类型的参数上，则行为类似于 'r'。

K

在关联数组中，这与 'k' 类似，但会返回 *exp* 与键匹配的所有值。 对于其他类型的参数，作用与 'R' 相同。

n:*expr*:

如果与 'r', 'R', 'i' 或 'I' 结合使用，则会给出第 n 个或后数第 n 个匹配（如果 *expr* 值为 n）。 当数组是关联数组时，该标志将被忽略。 分隔符 : 是任意的；见上文。

b:*expr*:

如果与 'r', 'R', 'i' 或 'I' 结合使用，会使它们从第 *n* 个或后数第 *n* 个元素、单词或字符（如果 *expr* 值为 *n*）开始。 如果数组是关联数组，该标志将被忽略。 分隔符：是任意的；见上文。

e

该标志会使任何对下标执行的模式匹配改用纯字符串匹配。 因此，'${array[(re)*]}' 只匹配值为 * 的数组元素。 请注意，其他形式的替换（如参数替换）并不受限制。

该标记还可用于强制将 * 或 @ 解释为单键，而不是对所有值的引用。 在赋值的左侧，它可以用于这两种目的。

请参阅 **参数扩展标志** (参数扩展)，了解操作数组下标结果的其他方法。

15.2.4 下标解析

本讨论主要适用于关联数组键字符串和用于反转下标的模式（'r', 'R', 'i' 等标志），但也可能影响作为算术表达式的一部分出现在普通下标中的参数替换。

在对关联数组元素赋值时，为避免下标解析的限制，可使用追加语法：

```
aa+=('key with "*strange*" characters' 'value string')
```

编写下标表达式时要记住的基本规则是，开头的 '[' 和结尾的 ']' 之间的所有文本都被**解释为**双引号引用（引用）。 不过，与通常不能嵌套的双引号不同，下标表达式可能出现在双引号引用字符串内部，也可能出现在其他下标表达式内部（或两者兼而有之！），因此这些规则有两个重要区别。

第一个区别是，括号（'[' 和 ']'）必须作为平衡对出现在下标表达式中，除非它们前面有反斜线（'\'）。 因此，在下标表达式中（与真正的双引号不同），'\[' 序列变为 '['，同样，'\]' 变为 ']'。 这甚至适用于通常不需要反斜线的情况；例如，模式 '[^[]'（匹配开放括号以外的任何字符）在反向下标模式中应写成' [^\[]'。 不过，请注意，'\[^\[\]' 甚至 '\[^[]' 都是 **一样** 的意思，因为反斜线出现在括号前时总是会被去掉！

同样的规则也适用于小括号（'(' 和 ')'）和大括号（'{' 和 '}'）：它们必须以平衡对形式出现，或在前面加上反斜线，在解析过程中，保护小括号或大括号的反斜线会被删除。 这是因为参数扩展可以用平衡大括号包围，而下标标志则由平衡小括号引入。

第二个区别是，双引号 ('")可以作为下标表达式的一部分出现，而不需要在前面加上反斜线，因此 '\"' 这两个字符仍然是下标中的两个字符（在真正的双引号中，'\"' 变成了'"'）。 然而，由于标准的 shell 引用规则，任何出现的双引号都必须成对出现，除非前面有反斜线。 这就增加了编写包含奇数双引号字符的下标表达式的难度，但之所以存在这种差异，是因为当下标表达式出现在真正的双引号内时，我们仍然可以把 '\"' 写成 '\"'（而不是 '\\\"'）。

要在赋值中使用奇数双引号作为键，请使用 typeset 内置命令和一对双引号；要引用该键的值，同样使用双引号：

```
typeset -A aa
typeset "aa[one\"two\"three\"quotes]"=QQQ
print "$aa[one\"two\"three\"quotes]"
```

值得注意的是，当带有下标的参数扩展嵌套在另一个下标表达式中时，引用规则不会改变。 也就是说，不需要在内部下标表达式中使用额外的反斜线；只需从最内层下标向外删除一次反斜线。 参数也是先从最内层的下标开始展开，因为在外层表达式中，每次展开都是从左向右进行的。

下标解析与双引号解析**并没有不同** （not different），这就造成了进一步的复杂性。 与真正的双引号一样，‘*’ 和‘\@’ 序列在下标表达式中出现时仍是两个字符。 要使用字面的‘*’ 或 ‘@’ 作为关联数组的键，必须使用 ‘e’ 标志：

```
typeset -A aa
aa[(e)*]=star
print $aa[(e)*]
```

在执行反向下标时，必须考虑最后一个细节。 下标表达式中出现的参数首先会被展开，然后整个表达式会被解释为一个模式。 这有两个影响：首先，参数的行为就好像 GLOB_SUBST 是开启的一样（而且无法关闭）；其次，反斜线会被解释两次，一次是在解析数组下标时，另一次是在解析模式时。 在反向下标中，必须使用 **四个** 反斜线，才能使单个反斜线与模式中的字面意思相匹配。 对于复杂的模式，通常最简单的做法是将所需模式指定给一个参数，然后在下标中引用该参数，因为只有在将完整表达式转换为模式时才会看到反斜线、花括号、小括号等。 要匹配反向下标中的参数字面值而不是模式，可使用 ‘${(q)*name*}’ （参数扩展）来为扩展后的值加引用。

请注意，‘k’ 和 ‘K’ 标志对于普通数组来说是反向下标，但对于关联数组来说是 **不是** 反向下标！(对于关联数组，数组本身的键被这些标志解释为模式；在这种情况下，下标是一个纯字符串）。

最后注意一点,与下标语法没有直接关系：位置参数（位置参数）的数字名称会被特殊解析，例如，‘$2foo’ 等同于 ‘${2}foo’。 因此，要使用下标语法从位置参数中提取子串，必须用大括号括起来；例如，‘${2[3,5]}’ 的值是第二个位置参数的第三到第五个字符，但 ‘$2[3,5]’ 是整个第二个参数与文件名生成模式 ‘[3,5]’ 的连接。

15.3 位置参数

位置参数用于访问 shell 函数、shell 脚本或 shell 本身的命令行参数；参见 调用，以及 函数。 参数 *n* 是第 *n* 个位置参数，其中 *n* 是一个数字。 参数 ‘$0’ 是一个特例，参见 由 Shell 设置的参数。

参数 *, @ 和 argv 是包含所有位置参数的数组；因此 '$argv[*n*]' 等，等同于 '$*n*'。 需要注意的是，选项 KSH_ARRAYS 或 KSH_ZERO_SUBSCRIPT 也适用于这些数组，因此在设置了这两个选项之一的情况下，'${argv[0]}' 等价于 '$1'，以此类推。

位置参数可以在 shell 或函数启动后通过 set 内置命令、赋值给 argv 数组或 '*n*=*value*' 的形式直接赋值进行修改，其中 *n* 是要修改的位置参数的编号。 这也会创建（用空值）从 1 到 *n* 尚未有值的任何位置。 需要注意的是，由于位置参数组成了一个数组，因此允许使用形式为 '*n*=(*value* ...)' 的数组赋值，其效果是将所有大于 *n* 的位置值移动需要的位置，以容纳新值。

15.4 局部参数

shell 函数的执行为 shell 参数限定了作用域（参数是动态作用域）。 (参数是动态作用域的。 ）typeset 内置命令及其替代形式 declare、integer、local 和 readonly（但不包括 export）可用于将参数声明为最内层作用域的局部参数。

当读取或赋值一个参数时，会使用该参数名的最内层现有参数（也就是说，局部参数会隐藏任何外层（less-local）参数）。 (不过，如果赋值给一个不存在的参数，或使用 export 声明一个新参数，则会在 **最外层**（outermost）作用域创建该参数。

局部参数会在其作用域结束时消失。 unset 可用于删除仍在作用域中的参数；任何同名的外层参数仍会被隐藏。

特殊参数也可以设置为局部参数；除非现有参数或新创建的参数具有 -h（hide）属性，否则特殊参数将保留其特殊属性。 这可能会产生意想不到的效果：变量没有默认值，因此如果在变量局部化时没有赋值，它将被设置为空值（如果是整数，则为零）。 下面是

```
typeset PATH=/new/directory:$PATH
```

可以暂时允许 shell 或从 shell 调用的程序在函数内查找 /new/directory 中的程序。

请注意，旧版本的 zsh 中关于永不导出局部参数的限制已被删除。

15.5 由 Shell 设置的参数

在后面的参数列表中，标记 '<S>' 表示参数是特殊的。 '<Z>' 表示 shell 在 sh 或 ksh 模拟模式下初始化时该参数不存在。

参数 '!', '#', '*', '-', '?', '@', '$', 'ARGC', 'HISTCMD', 'LINENO', 'PPID', 'status', 'TTYIDLE', 'zsh_eval_context', 'ZSH_EVAL_CONTEXT',和 'ZSH_SUBSHELL' 是只读的因此用户无法恢复（restore）, 所以它们不能通过 'typeset -p' 输出。 这也适用于从模块加载的许多只读参数。

以下参数由 shell 自动设置：

! <S>

使用 & 在后台启动、使用 bg 内置命令进入后台或使用 coproc 生成的最后一条命令的进程 ID。

<S>

位置参数的个数（十进制）。 请注意， $#*param* 的语法可能会与 *param* 的长度产生混淆。 请使用 ${#} 来消除歧义。 特别是，算术表达式中的序列 '$#-...' 会被解释为参数 - 的长度，参阅

ARGC <S> <Z>

与 # 一样。

$ <S>

此 shell 的进程 ID，在 shell 初始化时设置。 在不执行新程序的情况下从 shell 分支出来的进程，如命令替换和以 (...) 分组的命令，都是复制当前 shell 的子 shell，因此其 $$ 值与父 shell 相同。

- <S>

调用 shell 时或通过 set 或 setopt 命令提供给 shell 的标志。

* <S>

包含位置参数的数组。

argv <S> <Z>

与 * 相同。 向 argv 赋值会改变局部位置参数，但 argv 本身 **不是** 局部参数。 在任何函数中使用 unset 删除 argv 都会把它从所有位置删除，但只删除最内层的位置参数数组（因此其他作用域中的 * 和 @ 不受影响）。

@ <S>

与 argv[@]相同，即使 argv 未设置。

? <S>

最后一条命令返回的退出状态。

0 <S>

调用当前 shell 时使用的名称，或由 -c 命令行选项在调用时设置的名称。 如果设置了 FUNCTION_ARGZERO 选项，在进入 shell 函数时 $0 将被设置为该函数的名称，在进入引入（sourced）的脚本时 $0 将被设置为该脚本的名称，并在函数或脚本返回时重置为之前的值。

`status` <S> <Z>

与 ? 一样。

`pipestatus` <S> <Z>

数组，包含最后一条管道中所有命令返回的退出状态。

`_` <S>

上一条命令的最后一个参数。 此外，在执行每条命令时，该参数都会在环境中设置为命令的完整路径名。

`CPUTYPE`

运行时确定的机器类型（微处理器类别或机器型号）。

`EGID` <S>

shell 进程的有效组 ID。 如果您有足够的权限，可以通过指定此参数来更改 shell 进程的有效组 ID。 此外（假设有足够权限），还可以通过 '(`EGID=`*gid*`; command`)' 以不同的有效组 ID 启动单个命令。

如果将其设置为局部的，则不会隐式设置为 0，但是可以显式且局部地设置为0。

`EUID` <S>

shell 进程的有效用户 ID。 如果您有足够的权限，可以通过指定此参数来更改 shell 进程的有效用户 ID。 此外（假设有足够权限），还可以通过 '(`EUID=`*uid*`; command`)' 以不同的有效用户 ID 启动单个命令。

如果将其设置为局部的，则不会隐式设置为 0，但是可以显式且局部地设置为0。

`ERRNO` <S>

最近一次失败的系统调用所设置的 `errno`（参见 errno(3)）值。 该值取决于系统，用于调试目的。 在与 `zsh/system` 模块一起使用时也很有用，该模块允许将数字转换为名称或信息。

要使用该参数，必须先为其赋值（通常为 0（零））。 为了与脚本兼容，该参数最初未设置。

`FUNCNEST` <S>

整数。 如果大于或等于零，则为 shell 函数的最大嵌套深度。 超过该值时，将在调用函数时出错。 默认值在配置 shell 时确定，但通常为 500。 增加该值会增加函数递归失控导致 shell 崩溃的危险。 设置负值则会关闭检查。

`GID` <S>

shell 进程的真实组 ID。 如果您有足够的权限，可以通过为该参数赋值来更改 shell 进程的组 ID。 此外（假设有足够权限），还可以通过 '(GID=*gid*; command)' 在不同的组 ID 下启动一条命令。

如果将其设置为局部的，则不会隐式设置为 0，但是可以显式且局部地设置为0。

HISTCMD

交互式 shell 中的当前历史事件编号，换句话说，就是导致 $HISTCMD 被读取的命令的事件编号。 如果当前历史事件修改了历史记录，HISTCMD 将更改为新的最大历史事件编号。

HOST

当前主机名。

LINENO <S>

当前脚本、源文件或正在执行的 shell 函数（以最近启动者为准）中当前行的行号。 请注意，对于 shell 函数，行号指的是函数在原始定义中的行号，而不一定是 functions 内置命令所显示的行号。

LOGNAME

如果 shell 环境中未设置相应变量，则会将其初始化为当前登录会话对应的登录名。默认情况下会导出此参数，但可以使用 typeset 内置命令禁用。 如果 getlogin(3) 系统调用返回的字符串可用，该值将被设置为该字符串。

MACHTYPE

编译时确定的机器类型（微处理器类别或机器型号）。

OLDPWD

上一个工作目录。 它在 shell 初始化和目录更改时设置。

OPTARG <S>

getopts 命令处理的最后一个选项参数的值。

OPTIND <S>

getopts 命令处理的最后一个选项参数的索引。

OSTYPE

编译时确定的操作系统。

PPID <S>

shell 初始化时设置的 shell 父进程 ID。 与 $$ 一样，在作为当前 shell 的副本创建的子 shell 中，该值不会改变。

PWD

当前工作目录。 它在 shell 初始化和目录更改时设置。

RANDOM <S>

一个从 0 到 32767 的伪随机整数，每次引用该参数时都会新生成。 可以通过为 RANDOM 赋值来给随机数生成器一个随机数种子。

RANDOM 的值构成了一个可有意重复的伪随机序列；引用 RANDOM 的子 shell 将产生完全相同的伪随机值，除非在子 shell 调用之间引用 RANDOM 的值或在父 shell 中设置（seed） RANDOM 的值。

SECONDS <S>

调用 shell 后的秒数。 如果为该参数赋值，则引用后返回的值将是赋值的值加上赋值后的秒数。

与其他特殊参数不同，SECONDS 参数的类型可以通过 typeset 命令来更改。 类型只能改为浮点类型或整数类型。 例如，‘typeset -F SECONDS’ 会将数值报告为浮点数。 虽然 shell 可能会根据 typeset 的使用显示更多或更少的位数，但数值的精度可以达到微秒级。 更多详情，请参阅 Shell 内置命令 中的 typeset 文档。

SHLVL <S>

每次启动一个新 shell 时递增 1。

signals

包含信号名称的数组。 请注意，按照标准的 zsh 数组索引编号（第一个元素的索引为 1），信号与操作系统使用的信号编号相差 1。 例如，在典型的类 Unix 系统中 HUP 是信号编号 1，但这里引用为 $signals[2]。 这是因为 EXIT 位于数组的第 1 位，zsh 在内部使用它，但操作系统并不知道它。

TRY_BLOCK_ERROR <S>

在 always 块中，表示前面的代码列表是否导致错误。 值为 1 表示出错，否则为 0。 可以重置该值，清除错误条件。 参见 复杂命令

TRY_BLOCK_INTERRUPT <S>

该变量的工作方式与 TRY_BLOCK_ERROR 类似，但它代表来自 SIGINT 信号的中断状态，通常在用户从键盘键入 ^C 时。 如果设置为 0，任何此类中断都将被重置；否则，中断将在 always 代码块之后传播。

需要注意的是，在 always 块执行过程中可能会出现中断；该中断也会被传播。

TTY

与 shell 关联的 tty 的名称（如果有）。

TTYIDLE <S>

与 shell 关联的 tty 的空闲时间（以秒为单位），如果没有此类 tty，则为-1。

UID <S>

shell 进程的真实用户 ID。 如果您有足够的权限，可以通过为该参数赋值来更改 shell 的用户 ID。 此外（假设有足够权限），还可以通过‘(UID=*uid*; command)’以不同的用户 ID 启动一条命令。

如果将其设置为局部的，则不会隐式设置为 0，但是可以显式且局部地设置为0。

USERNAME <S>

与 shell 进程真实用户 ID 相对应的用户名。 如果您有足够的权限，可以通过指定该参数来更改 shell 的用户名（以及用户 ID 和组 ID）。 此外（假设有足够权限），还可以通过‘(USERNAME=*username*; command)’以不同的用户名（以及用户 ID 和组 ID）启动单个命令。

VENDOR

编译时确定的供应商。

zsh_eval_context <S> <Z> (ZSH_EVAL_CONTEXT <S>)

一个数组（以冒号分隔的列表），表示正在运行的 shell 代码的上下文。 每次执行一段存储在 shell 中的 shell 代码时，都会在数组中临时添加一个字符串，以指示正在执行的操作类型。 按顺序读取时，数组会显示正在执行的操作堆栈，最近的操作放在最后。

需要注意的是，该变量不提供诸如管道或子 shell 等语法上下文信息。 请使用 $ZSH_SUBSHELL 来检测子 shell。

上下文是以下情况之一：

cmdarg

由 -c 选项指定的调用 shell 的命令行代码。

cmdsubst

使用 `...` or $(...) 结构进行的命令替换。

equalsubst

使用 =(...) 结构进行的文件替换。

eval

由 eval 内置命令执行的代码。

evalautofunc

使用 KSH_AUTOLOAD 机制执行的代码，用于定义自动加载函数。

fc

通过 fc 内置命令的 -e 选项执行的 shell 历史代码。

file

直接从文件（例如 source 内置命令）读取的代码行。

filecode

从 .zwc 文件而不是直接从源文件中读取的代码行。

globqual

由 glob 限定符 e 或 + 执行的代码。

globsort

根据glob 限定符 o 对文件进行排序的代码。

insubst

使用 <(...) 结构的文件替换。

loadautofunc

直接从文件中读取以便定义自动加载函数的代码。

outsubst

使用 >(...) 结构进行文件替换。

sched

由 sched 内置命令执行的代码。

shfunc

一个 shell 函数。

stty

由 STTY 环境变量传递给 stty 的代码。 通常情况下，该值会直接传递给系统的 stty 命令，因此在实际应用中不太可能看到这个值。

style

作为 zsh/zutil 模块中 zstyle 内置命令获取的样式的一部分执行的代码。

toplevel

脚本或交互式 shell 的最高执行级别。

trap

以 trap 内置命令定义的陷阱形式执行的代码。 定义为函数的陷阱上下文为 shfunc。 由于陷阱是异步的，它们的层次结构可能与其他代码不同。

zpty

由 zsh/zpty 模块中的 zpty 内置命令执行的代码。

zregexparse-guard

zsh/zutil 模块中的 zregexparse 命令执行时，会被作为保护代码执行。

zregexparse-action

由 zsh/zutil 模块中的 zregexparse 命令作为操作（action）执行的代码。

ZSH_ARGZERO

如果调用 zsh 是为了运行脚本，则这是脚本的名称。 否则，它就是用于调用当前 shell 的名称。 这与设置 POSIX_ARGZERO 选项时 $0 的值相同，但始终可用。

ZSH_EXECUTION_STRING

如果 shell 是使用 -c 选项启动的，则此处包含该选项传递的参数。 否则不设置。

ZSH_NAME

扩展为用于调用此 zsh 实例的命令的基名。

ZSH_PATCHLEVEL

用于构建 shell 的 zsh 仓库的 'git describe --tags --long' 输出。 在开发过程中，为了跟踪 shell 的发布版本，这个输出是最有用的；因此大多数用户不应使用它，而应依赖 $ZSH_VERSION。

zsh_scheduled_events

参见 zsh/sched 模块.

ZSH_SCRIPT

如果调用 zsh 是为了运行脚本，则这是脚本的名称，否则不设置。

ZSH_SUBSHELL

只读整数。 初始值为 0，每次 shell 分叉创建用于执行代码的子 shell 时都会递增。 因此，'(print $ZSH_SUBSHELL)' 和 'print $(print $ZSH_SUBSHELL)' 输出 1，而 '((print $ZSH_SUBSHELL))' 输出 2。

ZSH_VERSION

zsh 版本的版本号。

15.6 Shell 使用的参数

shell 会使用下列参数。 同样，'<S>' 表示该参数是特殊的，'<Z>' 表示当 shell 在 sh 或 ksh 模拟模式下初始化时，该参数不存在。

如果两个参数的是同一个名称的大小写版本，例如 path 和 PATH，则小写形式的参数是一个数组，大写形式的参数是一个标量（以数组中的元素用冒号连接在一起组成）。 这些参数类似于通过 'typeset -T' 创建的绑定参数。 以冒号分隔的形式通常用于导出到环境中，而数组形式则更易于在 shell 中操作。 需要注意的是，取消设置这对参数中的任何一个都会取消设置另一个；它们在重新创建时会保留各自的特殊属性，重新创建这对参数中的一个也会重新创建另一个。

ARGV0

如果导出，其值将用作外部命令的 argv[0]。 通常用于 'ARGV0=emacs nethack' 等结构体。

BAUD

数据到达终端的速率（比特/秒）。 行编辑器将使用该值对缓慢的终端进行补偿，在必要时延迟更新显示内容。 如果参数未设置或值为零，补偿机制将被关闭。 默认情况下不设置该参数。

在某些情况下，例如对于拨号到通信服务器的慢速调制解调器或在慢速广域网中，设置该参数可能是有益的。 应将其设置为链路最慢部分的波特率，以获得最佳性能。

cdpath <S> <Z> (CDPATH <S>)

指定 cd 命令搜索路径的目录数组（以冒号分隔的列表）。

COLUMNS <S>

该终端会话的列数。 用于打印选择列表和行编辑器。

CORRECT_IGNORE

如果设置，则在拼写纠错过程中被视为模式。 任何与该模式匹配的潜在纠错都会被忽略。 例如，如果值为 '_*'，则补全函数（按照惯例，其名称以 '_' 开头）将永远不会作为拼写纠错提供。 该模式不适用于 CORRECT_ALL 选项对文件名的纠错（因此，在刚才的例子中，当前目录中以 '_' 开头的文件仍会被补全）。

CORRECT_IGNORE_FILE

如果设置，在文件名拼写纠错时将被视为模式。 任何与该模式匹配的文件名都不会作为纠错提供。 例如，如果值为 '.*'，则点文件名永远不会作为拼写纠错提供。 这对于和 CORRECT_ALL 选项一起使用非常有用。

DIRSTACKSIZE

目录堆栈的最大大小，默认情况下没有限制。 如果目录栈大于此值，将自动截断。 这和 AUTO_PUSHD 选项一起使用非常有用。

ENV

如果在以 sh 或 ksh 的方式调用 zsh 时设置了 ENV 环境变量，则 $ENV 将在 profile 脚本之后引入。 在将 ENV 的值解释为路径名之前，会对其进行参数扩展、命令替换和算术扩展。 请注意，除非 shell 是交互式的，并且 zsh 正在模拟 *sh* 或 *ksh*，否则 ENV是 **不**被使用的。

FCEDIT

fc 内置编辑器的默认编辑器。 如果未设置 FCEDIT，则使用参数 EDITOR；如果也未设置 ，则使用内置默认编辑器，通常是 vi。

fignore <S> <Z> (FIGNORE <S>)

数组（冒号分隔的列表），包含在文件名补全过程中要忽略的文件后缀。 不过，如果补全在此列表中只生成带后缀的文件，那么这些文件还是会被补全。

fpath <S> <Z> (FPATH <S>)

指定函数定义搜索路径的目录数组（冒号分隔列表）。 当带有 -u 属性的函数被引用时，将搜索该路径。 如果找到可执行文件，则在当前环境下读取并执行该文件。

histchars <S>

shell 的历史和词法分析机制使用的三个字符。 第一个字符表示历史扩展的开始（默认为 '!'）。 第二个字符表示快速历史替换的开始（默认为'^'）。 第三个字符是注释字符（默认为'#'）。

字符必须是 ASCII 字符集中的字符；任何将 histchars 设置为具有区域相关（locale-dependent）的字符的尝试都会被拒绝，并显示错误信息。

HISTCHARS <S> <Z>

与 histchars 相同。 (已废弃）。

HISTFILE

交互式 shell 退出时保存历史记录的文件。 如果未设置，则不保存历史记录。

HISTORY_IGNORE

如果设置，则在写入历史文件时被视为一种模式。 任何与该模式匹配的潜在历史条目都会被跳过。 例如，如果值为 'fc *'，则调用交互式历史编辑器的命令永远不会写入历史文件。

请注意，HISTORY_IGNORE 只定义了一种模式：要指定其他模式，请使用 '(*first*|*second*|...)' 语法。

对比 HIST_NO_STORE 选项或 zshaddhistory 钩子，这两种方法都会阻止此类命令被添加到交互式历史记录中。 如果希望使用 HISTORY_IGNORE 来阻止历史记录的添加，可以定义以下钩子：

```
zshaddhistory() {
  emulate -L zsh
  ## uncomment if HISTORY_IGNORE
  ## should use EXTENDED_GLOB syntax
  # setopt extendedglob
  [[ $1 != ${~HISTORY_IGNORE} ]]
}
```

HISTSIZE <S>

存储在内部历史列表中的最大事件数。 如果使用 HIST_EXPIRE_DUPS_FIRST 选项，将此值设置为大于 SAVEHIST 大小，就能获得差值作为保存重复历史事件的缓冲。

如果将其设置为局部的，则不会隐式设置为 0，但是可以显式且局部地设置为0。

HOME <S>

cd 命令的默认参数。在 sh、ksh 或 csh 模拟中，shell 不会自动设置该参数，但通常它会存在于环境中，如果被设置，那么通常会出现特殊行为。

IFS <S>

内部字段分隔符（默认为空格、制表符、换行符和 NUL），用于分隔命令或参数扩展产生的字和 read 内置命令读取的词。 在 IFS 中出现的空格、制表符和换行符中的任何字符都称为 ***IFS*空白字符**。 一个或多个 IFS 空白字符或一个非 IFS 空白字符连同任何相邻的 IFS 空格字符构成一个字段的分界。 如果一个 IFS 空白字符在 IFS 中连续出现两次，该字符不会被视为 IFS 空格字符。

如果参数未设置，则使用默认值。 请注意，这与将参数设置为空字符串的效果不同。

KEYBOARD_HACK

该变量定义了在解释命令行（仅限交互式 shell）之前从命令行末尾移除的字符。它的目的是解决按键与回车键距离过近的问题，并取代 SUNKEYBOARDHACK 选项，后者只针对反引号。 如果选择的字符是单引号、双引号或反引号之一，那么命令行中必须有奇数个单引号、双引号或反引号，最后一个才会被移除。

为了向后兼容，如果 SUNKEYBOARDHACK 选项被明确设置，KEYBOARD_HACK 的值将恢复为反引号。 如果明确未设置该选项，则该变量将被设置为空。

KEYTIMEOUT

shell 等待的时间， 在读取绑定的多字符序列时等待按下另一个键的时间，以百分之一秒为单位。

LANG <S>

对于未通过以 'LC_' 开头的变量具体选择的区域类别，该变量将决定其本地化类别。

LC_ALL <S>

该变量会覆盖 'LANG' 变量和其他以 'LC_' 开头的变量的值。

LC_COLLATE <S>

该变量用于确定 glob 括弧范围内字符对比信息和排序的本地化类别。

LC_CTYPE <S>

该变量决定字符处理功能的本地化类别。 如果使用 MULTIBYTE 选项，则该变量或 LANG 应包含一个能反映所用字符集的值，即使是单字节字符集，除非只使用 7 位子集（ASCII）。 例如，如果字符集是 ISO-8859-1，合适的值可能是 en_US.iso88591（某些 Linux 发行版）或 en_US.ISO8859-1（MacOS）。

LC_MESSAGES <S>

该变量决定了信息的编写语言。 请注意，zsh 不使用消息类。

LC_NUMERIC <S>

此变量会影响格式化输入/输出函数和字符串转换函数的小数点字符和千位分隔符。请注意，在解析浮点数学表达式时，zsh 会忽略此设置。

LC_TIME <S>

该变量决定提示符转义序列中日期和时间格式化的本地化类别。

LINES <S>

该终端会话的行数。 用于打印选择列表和行编辑器。

LISTMAX

在行编辑器中，无需先询问即可列出的匹配条数。如果该值为负数，则最多只能显示与绝对值相同行数的列表。 如果设置为零，则 shell 仅在列表顶部会滚动出屏幕时才会询问。

MAIL

如果设置了该参数，而 mailpath 未设置，shell 会在指定文件中查找邮件。

MAILCHECK

检查新邮件的间隔时间（秒）。

mailpath <S> <Z> (MAILPATH <S>)

用于检查新邮件的文件名数组（以冒号分隔的列表）。 每个文件名后可跟一个‘?’和一条将被打印的信息。 信息将进行参数扩展、命令替换和算术扩展，变量 $_ 被定义为已更改的文件名。 默认信息为‘You have new mail’。 如果元素是目录而不是文件，shell 将递归检查元素的每个子目录中的每个文件。

manpath <S> <Z> (MANPATH <S> <Z>)

数组（以冒号分隔的列表），其值不被 shell 使用。 不过，manpath 数组可能很有用，因为设置它也会设置 MANPATH，反之亦然。

match
mbegin
mend

在模式匹配中使用 b globbing 标志时由 shell 设置的数组。 参见 文件名生成 中的 ***Globbing* 标志** 小节。

MATCH
MBEGIN
MEND

在模式匹配中使用 m globbing 标志时由 shell 设置。 参见 文件名生成 中的 ***Globbing* 标志** 小节。

module_path <S> <Z> (MODULE_PATH <S>)

一个数组（以冒号分隔的列表），包含 zmodload 用于搜索动态加载模块的目录。它被初始化为一个标准路径名，通常是‘/usr/local/lib/zsh/$ZSH_VERSION’。（‘/usr/local/lib’ 部分因安装情况而异）。出于安全考虑，启动 shell 时在环境中设置的任何值都将被忽略。

只有当安装支持动态模块加载时，这些参数才会存在。

NULLCMD <S>

如果重定向未指定命令，使用的假定的命令名称。 默认为 cat。 若要使用 *sh*/*ksh* 行为，请将其更改为 :。 对于 *csh* 风格的行为，请取消设置此参数；如果输入了空命令，shell 将打印错误信息。

path <S> <Z> (PATH <S>)

用于搜索命令的目录数组（以冒号分隔的列表）。 设置该参数后，将扫描每个目录，并将找到的所有文件放入哈希表中。

POSTEDIT <S>

该字符串在行编辑器退出时输出。 它通常包含重置终端的 termcap 字符串。

PROMPT <S> <Z>

PROMPT2 <S> <Z>

PROMPT3 <S> <Z>

PROMPT4 <S> <Z>

分别与 PS1、PS2、PS3 和 PS4 相同。

prompt <S> <Z>

与 PS1 相同。

PROMPT_EOL_MARK

设置 PROMPT_CR 和 PROMPT_SP 选项后，可以使用 PROMPT_EOL_MARK 参数自定义部分行结束符的显示方式。 在设置 PROMPT_PERCENT 选项后，该参数会进行提示符扩展。 如果未设置，默认行为等同于值 ‘%B%S%#%s%b’。

PS1 <S>

主要提示符字符串，在读取命令前打印。 在显示之前，它会经过特殊形式的扩展；请参阅 提示符扩展。 默认值为“%m%# ”。

PS2 <S>

次级提示符，在 shell 需要更多信息来完成命令时打印。 其展开方式与 PS1 相同。 默认值为 “%_> ”，显示当前正在处理的任何 shell 结构或引号。

PS3 <S>

在 select 循环中使用的选择提示符。 其展开方式与 PS1 相同。 默认为 “?# ”。

PS4 <S>

执行跟踪提示符。 默认为 “+%N:%i> ”，显示当前 shell 结构的名称及其中的行号。 在 sh 或 ksh 模拟中，默认为 “+ ”。

psvar <S> <Z> (PSVAR <S>)

数组（以冒号分隔的列表），其元素可用于 PROMPT 字符串。 设置 psvar 也会设置 PSVAR，反之亦然。

READNULLCMD <S>

如果指定的是无命令的单输入重定向时，假定的命令名称。 默认为 more。

REPORTMEMORY

如果是非负值，则以千字节为单位的最大驻留集大小（粗略地说，就是主内存使用量）大于此值的命令将被报告计时统计信息。 输出统计信息的格式是 TIMEFMT 参数的值，与 REPORTTIME 变量和 time 内置命令相同；注意，默认情况下不输出内存使用情况。 在 TIMEFMT 值后面加上 " max RSS %M"，就会输出触发报告的值。 如果 REPORTTIME 也在使用中，则两个触发器最多只能打印一份报告。 此功能需要使用 getrusage()系统调用，现代类 Unix 系统通常都支持该调用。

REPORTTIME

如果是非负值，则用户和系统执行时间（以秒为单位）之和大于此值的命令将被打印计时统计数据。 在行编辑器中执行的命令（包括补全）的输出将被抑制（suppressed）；在这种情况下，用 time 关键字明确标记的命令仍会打印摘要。

REPLY

按照惯例，该参数用于在不可能或不希望调用函数或重定向的情况下，在 shell 脚本和 shell 内置命令之间传递字符串值。 read 内置命令和 select 复杂命令可以设置 REPLY，而文件名生成器在计算某些表达式时，既可以设置也可以检查其值。 某些模块也使用 REPLY 来达到类似目的。

reply

与 REPLY相同，但用于数组值而非字符串。

RPROMPT <S>

RPS1 <S>

当主提示符显示在屏幕左侧时，该提示将符显示在屏幕右侧。 如果设置了 SINGLE_LINE_ZLE 选项，则此提示符不起作用。 其扩展方式与 PS1 相同。

RPROMPT2 <S>

RPS2 <S>

当次级提示符显示在屏幕左侧时，该提示将显示在屏幕右侧。 如果设置了 SINGLE_LINE_ZLE 选项，则此提示符不起作用。 其扩展方式与 PS2 相同。

SAVEHIST

要保存在历史文件中的历史事件的最大数量。

如果将其设置为局部的，则不会隐式设置为 0，但是可以显式且局部地设置为0。

SPROMPT <S>

用于拼写更正的提示符。 序列 '%R' 扩展为需要拼写更正的字符串，'%r' 展开为建议更正的字符串。 还允许使用所有其他提示符转义。

提示符下可用的操作是 [nyae]：

n ('no') (default)

放弃更正并运行命令。

y ('yes')

更正并运行命令。

a ('abort')

不运行命令行，直接丢弃整个命令行。

e ('edit')

继续编辑命令行。

STTY

如果在命令的环境中设置了该参数，shell 将以该参数的值为参数运行 stty 命令，以便在执行命令前设置终端。模式仅适用于命令，并在命令结束或暂停时重置。如果命令被暂停，随后使用 fg 或 wait 内置命令继续执行，则会看到 STTY 指定的模式，就好像命令没有被暂停一样。 如果通过 'kill -CONT' 继续执行命令，则（故意）不适用此规定。 如果命令在后台运行，或者命令在 shell 环境中但未在输入行中明确指定，则 STTY 将被忽略。这样可以避免在每条外部命令中运行 stty 时意外导出它。还要注意的是，STTY 不应被用于指定窗口大小，因为这些指定并不是命令的本地化指定。

如果参数被设置为空，除了 stty 不被运行外，上述所有功能都适用。这可以用来冻结单个命令周围的 tty，阻止命令对 tty 设置的更改，类似于 ttyctl 内置命令。

TERM <S>

使用的终端类型。 在查找 termcap 序列时使用。 赋值给 TERM 会导致 zsh 重新初始化终端，即使值没有改变（例如，'TERM=$TERM'）。 有必要在终端定义数据库或终端类型发生任何更改时进行这样的赋值，以使新设置生效。

TERMINFO <S>

terminfo 数据库的引用，当系统有该数据库时，'terminfo' 库会使用它；参见 terminfo(5)。 如果设置了这个参数，shell 就会重新初始化终端，这样就不需要使用 'TERM=$TERM'。

TERMINFO_DIRS <S>

以冒号为分隔符的 terminfo 数据库列表，在系统有 'terminfo' 库时使用；参见 terminfo(5)。该变量仅用于某些终端库，尤其是 ncurses；请参见 terminfo(5) 检查系统是否支持该变量。 如果设置了该变量，shell 将重新初始化终端，从而无需使用 'TERM=$TERM'。 请注意，与其他以冒号分隔的数组不同，它并没有绑定到一个 zsh 数组。

TIMEFMT

使用 time 关键字的进程时间报告格式。 默认格式为 '%J %U user %S system %P cpu %*E total'。 可识别下列转义序列，但并非所有系统都能使用所有这些转义序列，而且有些转义序列可能并不实用：

%%

一个 '%'.

%U

用户模式下花费的 CPU 秒数。

%S

在内核模式下占用的 CPU 秒数。

%E

所用时间（秒）。

%P

CPU 百分比，计算公式为 100*(%U+%S)/%E。

%W

进程被交换的次数。

%X

使用的（共享）文本空间的平均数量（千字节）。

%D

使用的（非共享）数据/堆栈空间的平均数量（千字节）。

%K

使用的总空间（%X+%D），单位为千字节。

%M

进程在任何时候使用的最大内存（千字节）。

%F

主要页面故障（需要从磁盘调用页面）的数量。

%R

次要页面故障的数量。

%I

输入操作的数量。

%O

输出操作的次数。

%r

收到的套接字消息数量。

%s

发送的套接字消息数量。

%k

接收到的信号数量。

%w

自愿上下文切换（等待）的次数。

%c

非自愿上下文切换次数。

%J

这个作业的名字。

打印时间j时，可以在百分号和标志之间插入星号（例如，'%*E'）；这将导致时间以 '*hh*:*mm*:*ss*.*ttt*' 格式打印（小时和分钟仅在不为零时才打印）。 也可以使用 'm' 或 'u'（例如 '%mE'）分别以毫秒或微秒为单位输出时间。

TMOUT

如果该参数非零，则在发出提示符后的指定秒数内未输入命令，shell 将收到 ALRM 信号。如果 SIGALRM 上有陷阱，则会执行该陷阱，并在执行陷阱后使用 TMOUT 参数的值调度新的警报。 如果没有设置陷阱，并且终端的空闲时间不小于 TMOUT 参数的值，zsh 就会终止。 否则会在最后一次按键后的 TMOUT 秒内发出新的警报。

TMPPREFIX

路径名前缀，shell 会用于所有临时文件。 请注意，这应包括文件名的首部以及任何目录名。 默认值为 '/tmp/zsh'。

TMPSUFFIX

文件名后缀，shell 会将其用于进程替换创建的临时文件（例如，'=(list)'）。 请注意，如果要将该值解释为文件扩展名，则应包含一个前导点 '.'。 默认情况下不附加任何后缀，因此只有在需要时才分配该参数，然后再取消设置。

WORDCHARS <S>

行编辑器认为属于单词一部分的非字母数字字符列表。

ZBEEP

如果设置了该选项，就会向终端输出一串字符，这串字符可以使用 zsh/zle 模块 中描述的 bindkey 命令的所有相同代码，而不是发出蜂鸣声。 例如，在 vt100 或

xterm 上使用字符串 '\e[?5h\e[?5l' 会产生反向视频闪烁的效果（如果通常使用反向视频，则应使用字符串 '\e[?5l\e[?5h'）。 该选项优先于 NOBEEP 选项。

ZDOTDIR

如果不是 $HOME，则为搜索 shell 启动文件（.zshrc 等）的目录。

zle_bracketed_paste

许多终端模拟器都有一项功能，允许应用程序识别文本是何时粘贴到终端而不是正常键入的。对于 ZLE 来说，这意味着可以插入制表符和换行符等特殊字符，而不是调用编辑器命令。 此外，粘贴的文本会形成单个撤销事件，如果区域处于活动状态，粘贴的文本将取代区域。

该双元素数组包含用于启用和禁用该功能的终端转义序列。这些转义序列用于在 ZLE 处于活动状态时启用括号粘贴功能，而在其他时间禁用该功能。 取消设置该参数可确保括号粘贴功能保持禁用状态。

zle_highlight

一个数组，用于描述 ZLE 应高亮显示输入文本的上下文。 请参阅 字符高亮。

ZLE_LINE_ABORTED

该参数由行编辑器在发生错误时设置。 它包含发生错误时正在编辑的行。 可以使用 'print -zr -- $ZLE_LINE_ABORTED' 来恢复该行。 只有最近的一行才会被记住。

ZLE_REMOVE_SUFFIX_CHARS
ZLE_SPACE_SUFFIX_CHARS

行编辑器会使用这些参数。 在某些情况下，补全系统添加的后缀（通常是空格或斜线）会自动删除，这可能是因为下一个编辑命令不是可插入字符，也可能是因为该字符被标记为需要删除后缀。

这些变量可以包含会导致后缀被移除的字符集。 如果设置了 ZLE_REMOVE_SUFFIX_CHARS，这些字符会导致后缀被移除；如果设置了 ZLE_SPACE_SUFFIX_CHARS，这些字符会导致后缀被移除并替换为空格。

如果未设置 ZLE_REMOVE_SUFFIX_CHARS，则默认行为等同于：

```
ZLE_REMOVE_SUFFIX_CHARS=$' \t\n;&|'
```

如果 ZLE_REMOVE_SUFFIX_CHARS 已设置但为空，则没有字符具有这种行为。 ZLE_SPACE_SUFFIX_CHARS优先，因此下面的内容：

```
ZLE_SPACE_SUFFIX_CHARS=$'&|'
```

会导致字符 '&' 和 '|' 删除后缀，但用空格代替。

为了说明两者的区别，假设 AUTO_REMOVE_SLASH 选项有效，目录 DIR 刚刚补全，并附加了 /，用户随后键入了 '&'。 默认结果为 'DIR&'。 如果设置了 ZLE_REMOVE_SUFFIX_CHARS，但未包含 '&'，则结果为 'DIR/&'。 设置了 ZLE_SPACE_SUFFIX_CHARS 并包含 '&'，结果是 'DIR &'。

请注意，某些补全可能会提供自己的后缀移除或替换行为，从而覆盖此处描述的值。 请参阅 补全系统 中的补全系统文档。

ZLE_RPROMPT_INDENT <S>

如果设置，用于给出 RPS1 或 RPROMPT 所给出的行编辑器右侧提示符与屏幕右侧之间的缩进。 如果未设置，则使用值 1。

通常情况下，可以将该值设置为 0，这样提示符就会与屏幕右侧齐平。 这不是默认值，因为许多终端无法正确处理这种情况，尤其是当提示出现在屏幕右下方时。 最新的虚拟终端更有可能正确处理这种情况。 有必要进行一些试验。

---

16 选项

16.1 指定选项

选项主要用名称表示。 这些名称不区分大小写，下划线将被忽略。 例如，'allexport' 等同于 'A__lleXP_ort'。

选项名称的含义可以通过在其前面加上 'no' 来反转，因此 'setopt No_Beep' 等同 'unsetopt beep'。 这种反转只能进行一次，所以 'nonobeep' 不是 'beep' 的同义词。同样，'tify' 也不是 'nonotify' （'notify' 的反义词）的同义词。

有些选项还有一个或多个单字母名称。 有两组单字母选项：一组默认使用，另一组用于模拟 *sh*/*ksh*（当设置了 SH_OPTION_LETTERS 选项时使用）。 单字母选项可以在 shell 命令行中使用，也可以与 set、setopt 和 unsetopt 内置命令一起使用，就像以 '-' 为前缀的普通 Unix 选项一样。

使用 '+' 而不是 '-' 可以反转单字母选项的含义。 有些单字母选项名称指的是关闭选项，在这种情况下，该名称的反义词指的是开启选项。 例如，'+n' 是 'exec' 的简称，而 '-n' 是其反义词 'noexec' 的简称。

在启动时提供给 shell 的单字母选项字符串中，尾部空白将被忽略；例如，字符串 '-f ' 将被视为 '-f'，但字符串 '-f i' 则是一个错误。 这是因为许多采用 '#!' 机制调用脚本的系统并不清除尾部空白。

可以在函数作用域内设置选项。 请参阅下文对 LOCAL_OPTIONS 选项的描述。

16.2 选项说明

在以下列表中，所有模拟中默认设置的选项都标为 <D>；仅在 csh、ksh、sh 或 zsh 模拟中默认设置的选项则根据情况标为 <C>、<K>、<S>、<Z>。 在列出选项（通过‘setopt’, ‘unsetopt’, ‘set -o’ 或 ‘set +o’）时，默认开启的选项会以 ‘no’ 为前缀出现在列表中。 因此（除非设置了 KSH_OPTION_PRINT），‘setopt’ 会显示所有设置与默认值不同的选项。

16.2.1 改变目录

AUTO_CD (-J)

如果发出的命令不能作为普通命令执行，且该命令是一个目录的名称，则执行 cd 命令以进入该目录。 只有设置了 SHIN_STDIN，即从标准输入读取命令时，该选项才适用。 该选项专为交互式使用而设计；建议在脚本中明确使用 cd，以避免歧义。

AUTO_PUSHD (-N)

让 cd 将旧目录推送到目录栈。

CDABLE_VARS (-T)

如果 cd 命令（或设置了 AUTO_CD 选项的隐式 cd）的参数不是目录，也不是以斜线开头的，则尝试扩展表达式，就好像在表达式前面加上了 ‘~’（参见 文件名扩展）。

CD_SILENT

切勿在 cd（无论是显式还是通过设置 AUTO_CD 选项隐含）之后打印工作目录。当 cd 的参数是 -、堆栈条目或 CDPATH 下的目录名时，cd 通常会打印工作目录。请注意，这与 pushd 的堆栈打印行为不同，后者由 PUSHD_SILENT 控制。此选项会覆盖 POSIX_CD 的打印相关效果。

CHASE_DOTS

当切换到一个目录时，如果该目录中包含路径段 ‘..’，这通常会被认为是取消前一个路径段的意思（也就是说，‘foo/..’ 会在路径中被移除，或者如果 ‘..’ 是路径的第一部分，那么当前工作目录的最后一部分会被移除。），而不是解析物理目录的路径。 该选项被 CHASE_LINKS 覆盖。

例如，假设 /foo/bar 是指向 /alt/rod 目录的链接。 如果没有设置该选项，‘cd /foo/bar/..’ 会更改为 /foo；如果设置了该选项，则会更改为 /alt。如果当前目录是 /foo/bar，并使用了 ‘cd ..’，情况也一样。 请注意，路径中的所有其他符号链接也将被解析。

CHASE_LINKS (-w)

更改目录时，将符号链接转换为其真实值。 这也具有 CHASE_DOTS 的效果，即 '. .' 路径段将被视为指向物理父目录，即使前面的路径段是符号链接。

POSIX_CD <K> <S>

修改 cd、chdir 和 pushd 命令的行为，使其更符合 POSIX 标准。未设置该选项时的行为，参见 Shell 内置命令 中 cd 内置命令的文档。 如果设置了该选项，在 cdpath 中的所有目录都测试完毕之前，shell 不会测试本地目录 ('.')下的目录，并且 cd 和 chdir 命令不会将形式为 '{+|-}*n*' 的参数视为目录栈条目。

此外，如果设置了该选项，shell 打印新目录的条件也会发生变化。 它不再仅限于交互式 shell（尽管使用 pushd 打印目录堆栈仍仅限于交互式 shell）；任何使用 CDPATH 的组件，包括 '.'，但不包括空组件（空组件在其他情况下被视为 '.'），都会导致目录被打印。

PUSHD_IGNORE_DUPS

不要将同一目录的多个副本推送到目录堆栈中。

PUSHD_MINUS

在使用数字指定堆栈中的目录时，交换 '+' 和 '-' 的含义。

PUSHD_SILENT (-E)

不在 pushd 或 popd 之后打印目录堆栈。

PUSHD_TO_HOME (-D)

让不带参数的 pushd 像 'pushd $HOME' 一样运行。

16.2.2 补全

ALWAYS_LAST_PROMPT <D>

如果未设置，则列出补全的关键函数会在给出数字参数时尝试返回最后一个提示符。如果设置，这些函数在**没有**数字参数的情况下会尝试返回最后一个提示符。

ALWAYS_TO_END

如果光标在单词内执行了补全操作，并且插入了完整补全，则光标会移动到单词的末尾。 也就是说，如果插入了单个匹配或执行了菜单补全，光标就会移动到单词的末尾。

AUTO_LIST (-9) <D>

自动列出模棱两可的补全选项。

AUTO_MENU <D>

在连续两次请求补全（例如重复按下制表符键）后自动使用菜单完成。该选项被 MENU_COMPLETE 覆盖。

AUTO_NAME_DIRS

任何被设置为目录绝对名称的参数都会立即成为该目录的名称，将被 '%~' 和相关提示符序列使用，并在对以 '~' 开头的单词进行补全时可用（否则，必须先以 '~*param*' 的形式使用该参数）。

AUTO_PARAM_KEYS <D>

如果参数名称补全后，后面的字符（通常是空格）被自动插入，而下一个输入的字符必须直接位于参数名称之后（如 '}', ':' 等），则自动添加的字符将被删除，这样输入的字符将紧接在参数名称之后。 在括号扩展中补全也会受到类似影响：添加的字符是','，如果接下来输入 '}'，该字符将被删除。

AUTO_PARAM_SLASH <D>

如果补全的参数内容是一个目录的名称，则在后面加上斜线而不是空格。

AUTO_REMOVE_SLASH <D>

如果补全后的最后一个字符是斜线，而输入的下一个字符是单词分隔符、斜线或命令结束符（如分号或括弧），则删除斜线。

BASH_AUTO_LIST

在补全模棱两可时，如果补全函数被连续调用两次时，会自动列出选项。 这优先于 AUTO_LIST。 将尊重 LIST_AMBIGUOUS 的设置。 如果设置了 AUTO_MENU，则菜单行为将从第三次按下时开始。 请注意，在使用 MENU_COMPLETE 时，此功能将无法正常工作，因为在这种情况下，重复执行的补全调用会立即在列表中循环。

COMPLETE_ALIASES

防止命令行上的别名在尝试补全前被内部替换。 这样做的目的是使别名成为一个独立的命令。

COMPLETE_IN_WORD

如果未设置，则在开始补全时将光标设置到字尾。否则，光标将停留在该处，并从两端开始补全。

GLOB_COMPLETE

当当前单词包含一个 glob 模式时，不会插入扩展后的所有单词，而是像 MENU_COMPLETE 一样生成补全匹配并循环播放。在生成匹配时，就好像在单词末尾添加了一个 '*'，或者在设置 COMPLETE_IN_WORD 时在光标处插入了一个 '*'。

这实际上使用的是模式匹配，而不是 globbing 匹配，因此它不仅适用于文件，也适用于任何补全，如选项、用户名等。

请注意，在使用模式匹配器时，不能使用匹配控制（例如，不区分大小写或锚定匹配）。 这一限制仅适用于当前单词包含模式的情况；简单地打开 GLOB_COMPLETE 选项不会产生这种效果。

HASH_LIST_ALL <D>

无论何时尝试命令补全或拼写更正，都要确保先对整个命令路径进行散列(hash)。这样会降低首次补全的速度，但可以避免拼写错误的错误报告。

LIST_AMBIGUOUS <D>

当 AUTO_LIST 或 BASH_AUTO_LIST 也被设置时，该选项才会起作用。 如果要在命令行中插入一个明确的前缀，则无需显示补全列表即可完成；换句话说，只有在不会插入任何前缀的情况下，才会执行自动列表行为。 在 BASH_AUTO_LIST 的情况下，这意味着列表将延迟到函数的第三次调用。

LIST_BEEP <D>

在模棱两可的补全时发出蜂鸣声。 更准确地说，如果同时设置了选项 BEEP，这将强制补全小部件在模棱两可的补全时返回状态 1，从而导致 shell 发出蜂鸣声；如果补全是从用户自定义的小部件调用的，这一点可以修改。

LIST_PACKED

尝试将匹配项打印在不同宽度的列中，使补全列表更小（占用的行数更少）。

LIST_ROWS_FIRST

在补全列表中按水平排序排列匹配项，即第二个匹配项在第一个匹配项的右边，而不是像往常一样在第一个匹配项的下面。

LIST_TYPES (-X) <D>

在列出可能补全的文件时，用尾部识别标记显示每个文件的类型。

MENU_COMPLETE (-Y)

在出现模棱两可的补全时，不要列出可能性或发出蜂鸣声，而是立即插入第一个匹配项。 然后，当再次要求补全时，移除第一个匹配项，插入第二个匹配项，等等。当没有匹配项时，再返回第一个匹配项。 reverse-menu-complete 可用来从另一个方向循环浏览列表。该选项覆盖 AUTO_MENU。

REC_EXACT (-S)

如果命令行上的字符串与其中一个可能的补全完全匹配，这个补全会被接受（即使有另一个补全<即该字符串添加了其他内容>也匹配）。

16.2.3 扩展和 Globbing

BAD_PATTERN (+2) <C> <Z>

如果文件名生成的模式不正确，则打印错误信息。 (如果未设置该选项，则模式将保持不变）。

BARE_GLOB_QUAL <Z>

在 glob 模式中，如果尾部的括号不包含 '|', '(' 或（如果指定）'~' 字符，则将其视为限定符列表。 参见 文件名生成。

BRACE_CCL

将括号中的表达式扩展为一个包含所有字符的字面顺序的列表，否则这些表达式不会进行括号扩展。 参见 括号扩展。

CASE_GLOB <D>

使 globbing（文件名生成）对大小写敏感。 请注意，模式的其他用法始终对大小写敏感。 如果未设置该选项，则文件名生成过程中出现的任何特殊字符都将导致不区分大小写的匹配。 例如，由于存在 globbing 标志，cvs(/) 可以匹配目录 CVS（除非未设置选项 BARE_GLOB_QUAL）。

CASE_MATCH <D>

使使用 zsh/regex 模块的正则表达式（包括使用 =~ 的匹配）对大小写敏感。

CASE_PATHS

如果未设置 CASE_PATHS（默认值），只要 **任意** 组件中出现特殊字符，CASE_GLOB 就会影响 **每一个** 路径组件的解释。 设置 CASE_PATHS 后，**不** 包含特殊文件名生成字符的文件路径组件始终对大小写敏感，因此 NO_CASE_GLOB 只适用于包含 globing 字符的组件。

请注意，如果文件系统本身对大小写不敏感，那么 CASE_PATHS 就没有任何作用。

CSH_NULL_GLOB <C>

如果生成文件名的模式没有匹配项，则从参数列表中删除该模式；除非命令中的所有模式都没有匹配项，否则不报错。 覆盖 NOMATCH。

EQUALS <Z>

执行 = 文件名扩展。 (参见 文件名扩展）。

EXTENDED_GLOB

将 '#', '~' 和 '^' 字符视为文件名生成等的模式的一部分（初始的无引号的 '~' 总是产生命名目录扩展）。

FORCE_FLOAT

即使不使用小数点，算术运算中的常量也将被视为浮点数；整数变量在算术表达式中使用时，其值将转换为浮点数。 任何基数的整数都会被转换。

GLOB (+F, ksh: +f) <D>

执行文件名生成（globbing）。 (参见 文件名生成）。

GLOB_ASSIGN <C>

如果设置了该选项，文件名生成（globbing）将在'*name*=*pattern*'(例如 'foo=*')形式的标量参数赋值的右侧进行。 如果结果有多个单词，参数将变成一个数组，并以这些单词作为参数。提供该选项只是为了向后兼容：globbing 总是在形式为 '*name*=(*value*)' (e.g. 'foo=(*)') 的数组赋值的右侧执行，为了清晰起见，推荐使用这种形式；设置该选项后，无法预测结果是数组还是标量。

GLOB_DOTS (-4)

不要求明确匹配文件名中的前导 '.'。

GLOB_STAR_SHORT

当设置了该选项且默认的 zsh 风格 globbing 有效时，'**/*' 模式可缩写为 '**'，而 '***/*' 模式可缩写为 ***。 因此，'**.c' 可以在任何子目录中找到以 .c 结尾的文件，而 '***.c' 也可以在跟踪符号链接的同时找到以 ***.c 结尾的文件。 紧跟在'**' 或 '***' 之后的 / 会强制将模式视为未缩写形式。

GLOB_SUBST <C> <K> <S>

将参数扩展产生的任何字符视为符合文件名扩展和文件名生成条件，将命令替换产生的任何字符视为符合文件名生成条件。 括号（和中间的逗号）不符合扩展条件。

HIST_SUBST_PATTERN

使用 :s 和 :& 历史修饰符的替换的执行，是使用模式匹配而不是字符串匹配。只要历史修饰符（包括glob限定符和参数）有效，就会发生这种情况。请参见 修饰符。

IGNORE_BRACES (-I) <S>

不执行括号扩展。 由于历史原因，这也包括 IGNORE_CLOSE_BRACES 选项的效果。

IGNORE_CLOSE_BRACES

如果既未设置该选项，也未设置 IGNORE_BRACES，那么在命令行的任何位置，单一的封闭括号字符 '}' 都具有语法意义。 这样一来，在结束函数或当前 shell 结构的括号前就不需要分号或换行符了。 如果设置了这两个选项中的任何一个，则结尾大括号只在命令行位置有语法意义。 与 IGNORE_BRACES 不同，该选项不会禁用括号扩展。

例如，在两个选项都未设置的情况下，可以按以下方式定义一个函数：

```
args() { echo $# }
```

而如果设置了这两个选项中的任何一个，则不起作用，需要类似下面的设置：

```
args() { echo $#; }
```

KSH_GLOB <K>

在模式匹配中，括号的解释受前面的 '@', '*', '+', '?' 或 '!' 的影响。 参见 文件名生成。

MAGIC_EQUAL_SUBST

在命令名之后出现的所有未加引号的参数，其形式为 '*anything*=*expression*'，其文件名扩展（即 *expression* 带有前导 '~' 或 '=' ）在 *expression* 上执行，如同这是参数赋值。 参数不会受到其他特殊处理；它将作为单个参数传递给命令，而不是用作实际的参数赋值。 例如，在 `echo foo=~/bar:~/rod` 中，~ 的两次出现都会被替换。 请注意，typeset 和类似语句都会出现这种情况。

该选项尊守 KSH_TYPESET 选项的设置。 换句话说，如果两个选项都有效，看起来像赋值的参数将不会被分词。

MARK_DIRS (-8, ksh: -X)

在文件名生成（globbing）过程中产生的所有目录名后添加尾部 '/' 。

MULTIBYTE <D>

当字符串中出现多字节字符时，尊守多字节字符。 设置该选项后，将使用系统库检查字符串，以确定一个字符由多少个字节组成，具体取决于当前的本地语言。 这将影响字符在模式匹配、参数值和各种分隔符中的计数方式。

如果 shell 在编译时使用了 MULTIBYTE_SUPPORT，则默认开启该选项；否则默认关闭，开启后也不会有任何影响。

如果关闭该选项，单字节将始终被视为单字符。 这一设置纯粹是为了检查已知包含原始字节或其他值的字符串，这些字节或值在当前本地语言中可能不是字符。 没有必要仅仅因为当前语言的字符集不包含多字节字符而取消设置该选项。

该选项不会影响 shell 的编辑器，它始终使用本地设置来确定多字节字符。 这是因为终端模拟器显示的字符集与 shell 设置无关。

NOMATCH (+3) <C> <Z>

如果文件名生成的模式没有匹配项，则打印错误信息，而不是在参数列表中保持不变。 这也适用于起始为 '~' 或 '='的文件扩展。

NULL_GLOB (-G)

如果文件名生成的模式没有匹配项，则从参数列表中删除该模式，而不是报错。 覆盖 NOMATCH。

NUMERIC_GLOB_SORT

如果数字文件名与文件名生成模式相匹配，则按数字而不是按词法对文件名进行排序。

RC_EXPAND_PARAM (-P)

格式为 '*foo*${*xx*}*bar*' 的数组扩展形式（其中参数 *xx* 被设置为（*a b c*）），将被替换为 '*fooabar foobbar foocbar*' ，而不是默认的 '*fooa b cbar*'。 请注意，如果数组为空，所有参数都会被移除。

REMATCH_PCRE

如果设置，使用 =~ 操作符匹配的正则表达式将使用 PCRE 库中的 Perl 兼容正则表达式（zsh/pcre 模块必须可用）。 如果未设置，正则表达式将使用系统库提供的扩展 regexp 语法。

SH_GLOB <K> <S>

在参数和命令替换的结果进行通配符匹配（globbing），以及其它一些 shell 接受模式的地方，禁用 '(', '|', ')' 和 '<' 的特殊含义。 如果设置了 SH_GLOB，但未设置 KSH_GLOB，则 shell 允许在某些情况下解释括号中的子 shell 表达式，这些情况下括号前没有空格，例如 !(true) 被解释为 ! 后有空格。 如果以 sh 或 ksh 模拟方式调用 zsh，则默认设置该选项。

UNSET (+u, ksh: +u) <K> <S> <Z>

在替换参数时，将未设置参数视为空参数；在算术扩展和算术命令中读取参数值时，将未设置参数视为零。 否则将作为错误处理。

WARN_CREATE_GLOBAL

当函数中通过赋值或 在数学上下文中创建全局参数时，打印警告信息。 这通常表明某个参数本应声明为本地参数，但却未声明。 在函数中使用 typeset -g 明确声

明的全局参数不会导致警告。 需要注意的是，在嵌套函数中赋值给局部参数时也不会发出警告，这也可能表明存在错误。

WARN_NESTED_VAR

在函数中通过赋值或在数学上下文中设置闭合函数作用域或全局的现有参数时，打印警告信息。 对 shell 特殊参数赋值不会引起警告。 这与 WARN_CREATE_GLOBAL 类似，因为在这种情况下，只有在 **没有** 创建参数时才会打印警告信息。 在可能的情况下，使用 typeset -g 设置参数可以抑制错误，但需要注意的是，每次设置参数时都需要使用该方式。 要将此选项的作用限制在单个函数作用域内，请使用 'functions -W'。

例如，以下代码会对nested 函数内的赋值产生警告，因为该赋值会覆盖 toplevel 内的值

```
toplevel() {
  local foo="in fn"
  nested
}
nested() {
     foo="in nested"
}
setopt warn_nested_var
toplevel
```

16.2.4 历史

APPEND_HISTORY <D>

如果设置了此项，zsh 会话将会把它们的历史列表追加到历史文件中，而不是替换它。因此，多个并行的 zsh 会话都会按照退出的顺序，将其历史列表中的新条目添加到历史文件中。 当文件行数增长超过 $SAVEHIST 指定值的 20% 时，文件仍会定期重写以进行修剪（另请参阅 HIST_SAVE_BY_COPY 选项）。

BANG_HIST (+K) <C> <Z>

以 *csh* 风格执行文本性历史扩展，对字符 '!' 进行特别处理。

EXTENDED_HISTORY <C>

将每条命令的起始时间戳（以秒为单位，从纪元<epoch>开始）和持续时间（以秒为单位）保存到历史文件中。 这些前缀数据的格式为:

': *<beginning time>*:*<elapsed seconds>*;*<command>*'.

HIST_ALLOW_CLOBBER

添加 '|' 以在历史记录中输出重定向。 这样，即使 CLOBBER 未设置，也能引用历史记录中的 clobber （强制覆盖？重定向？）文件。

HIST_BEEP <D>

当小部件试图访问不存在的历史条目时，在 ZLE 中发出提示音。

HIST_EXPIRE_DUPS_FIRST

如果需要对内部历史记录进行修剪以添加当前命令行，则在丢失列表中的唯一事件之前，设置此选项将导致丢失有重复记录的最旧的历史事件。 请确保 HISTSIZE 的值大于 SAVEHIST 的值，以便为重复事件留出一定的空间，否则一旦历史记录被唯一事件填满，该选项的作用就和 HIST_IGNORE_ALL_DUPS 一样了。

HIST_FCNTL_LOCK

在写出历史文件时，zsh 默认使用临时（ad-hoc)文件锁定，以避免在某些操作系统上出现已知的锁定问题。 使用该选项时，锁定是通过系统的 fcntl 调用（如果该方法可用）来完成的。 在最新的操作系统上，这可能会提供更好的性能，尤其是当文件存储在 NFS 上时，可以避免历史损坏。

HIST_FIND_NO_DUPS

在行编辑器中搜索历史条目时，不要显示与之前找到的行重复的内容，即使重复的内容并不连续。

HIST_IGNORE_ALL_DUPS

如果添加到历史记录列表中的新命令行与旧命令行重复，旧命令行将从列表中删除（即使它不是上一个事件）。

HIST_IGNORE_DUPS (-h)

如果命令行与前一个事件重复，不将其输入历史记录列表。

HIST_IGNORE_SPACE (-g)

当命令行的第一个字符是空格，或扩展后的别名之一包含前导空格时，从历史记录列表中删除命令行。 只有普通别名（不是全局别名或后缀别名）才有这种行为。 需要注意的是，该命令会在内部历史记录中停留，直到输入下一条命令后才会消失，因此可以短暂重用或编辑该行。 如果想在不输入其他命令的情况下让该命令立即消失，请键入空格并按回车键。

HIST_LEX_WORDS

默认情况下，从文件读入的 shell 历史记录会在所有空白处分割成单词。 这意味着有带引号的空白的参数不会被正确处理，结果是从文件读入的历史行中的单词引用可能会不准确。 设置该选项后，从历史文件读入的单词将以类似于正常 shell 命令

行处理的方式分割。 虽然这样分隔的字词更准确，但如果历史文件很大，分隔速度可能会很慢。 这需要反复试验才能决定。

HIST_NO_FUNCTIONS

从历史记录列表中删除函数定义。 请注意，该函数会一直保留在内部历史记录中，直到输入下一条命令后才会消失，因此您可以短暂重用或编辑该定义。

HIST_NO_STORE

从历史记录列表中移除 history （fc -l）命令。(fc -l)命令。 请注意，该命令会一直保留在内部历史记录中，直到输入下一条命令后才会消失，因此您可以短暂重用或编辑该行。

HIST_REDUCE_BLANKS

删除添加到历史记录列表的每行命令中多余的空白。

HIST_SAVE_BY_COPY <D>

重写历史文件时，我们通常会写出一个名为 $HISTFILE.new 的文件副本，然后将其重命名为旧版本。 但如果未设置该选项，我们就会截断旧的历史文件，然后就地写出新版本。 如果启用了其中一个历史附加选项，那么只有在需要重新写入扩大的历史文件以缩小其大小时，该选项才会起作用。 只有在有特殊需要时才禁用此选项，因为这样做如果在保存过程中 zsh 被中断，则可能会丢失历史条目。

在写出历史文件副本时，zsh 会保留旧文件的权限和组信息，但如果会更改历史文件的所有者，则会拒绝写出新文件。

HIST_SAVE_NO_DUPS

在写出历史文件时，会省略与新命令重复的旧命令。

HIST_VERIFY

每当用户输入带有历史扩展的行时，不要直接执行该行，而是执行历史扩展并将该行重新载入编辑缓冲区。

INC_APPEND_HISTORY

此选项的作用与 APPEND_HISTORY 类似，只是新的历史行会以增量方式添加到 $HISTFILE 中（一旦输入），而不是等到 shell 退出。 当文件行数增长超过 $SAVEHIST 指定值的 20% 时，仍会定期重写文件以修剪行数（参见 HIST_SAVE_BY_COPY 选项）。

INC_APPEND_HISTORY_TIME

该选项是 INC_APPEND_HISTORY 的一个变体，在可能的情况下，历史条目会在命令执行完毕后写出到文件，这样命令所花费的时间就会以 EXTENDED_HISTORY 的

格式被正确地记录在历史文件中。 这意味着使用同一历史文件的其他 shell 实例无法立即获得该历史条目。

该选项只有在 INC_APPEND_HISTORY 和 SHARE_HISTORY 关闭时才有用。 这三个选项应视为相互排斥。

SHARE_HISTORY <K>

该选项既可以从历史文件中导入新命令，也可以将输入的命令添加到历史文件中（后者就像指定 INC_APPEND_HISTORY，如果使用该选项，则应关闭 INC_APPEND_HISTORY选项）。 历史记录行还会输出时间戳，如 EXTENDED_HISTORY（这样就能在重写文件后更容易地找到我们停止读取文件的位置）。

默认情况下，历史移动命令会同时访问导入的行和本地的行，但你可以使用 set-local-history zle 绑定打开或关闭这一功能。 也可以创建一个 zle 小部件，让某些命令忽略导入的命令，而另一些命令则包含导入的命令。

如果你想对命令的导入时间有更多控制，不妨关闭 SHARE_HISTORY、打开 INC_APPPEND_HISTORY 或 INC_APPPEND_HISTORY_TIME（见上文），然后在需要时使用 'fc -RI' 手动导入命令。

16.2.5 初始设置

ALL_EXPORT (-a, ksh: -a)

随后定义的所有参数都会自动导出。

GLOBAL_EXPORT <Z>

如果设置了该选项，向内置函数 declare、float、integer、readonly 和 typeset（但不包括 local）传递 -x 标志时，也会设置 -g 标志；因此，导出到环境中的参数将不会在封闭（enclosing）函数中本地化，除非这些参数已经本地化或明确给出了 +g 标志。 如果未设置该选项，导出参数将与其他参数一样被本地化。

默认设置该选项是为了向后兼容；不建议依赖其行为。 请注意，内置的 export 总是同时设置 -x 和 -g 标志，因此其作用超出了封闭函数的范围；这是实现这一行为的最便携（portable）的方法。

GLOBAL_RCS (+d) <D>

如果未设置该选项，则不会运行启动文件 /etc/zprofile、/etc/zshrc、/etc/zlogin 和 /etc/zlogout。 可以随时禁用和重新启用，包括在本地启动文件（.zshrc等）中禁用和重新启用。

RCS (+f) <D>

启动时引入（sourced）/etc/zshenv后，将按照 文件 中的说明，引入 .zshenv、/etc/zprofile、.zprofile、/etc/zshrc、.zshrc、/etc/zlogin、.zlogin 和 .zlogout 文件。 如果未设置该选项，/etc/zshenv 文件仍会被引入，但其他文件将不会被引入；也可随时设置该选项，以防止当前执行文件之后的其他启动文件被引入。

16.2.6 输入/输出

ALIASES <D>

扩展别名。

CLOBBER (+C, ksh: +C) <D>

允许使用 '>' 重定向来截断现有文件。 否则，必须使用 '>!' 或 '>|' 来截断文件。

如果未设置该选项，且选项 APPEND_CREATE 也未设置，则必须使用 '>>!' 或 '>>|' 来创建文件。 如果设置了其中一个选项，则可以使用 '>>' 。

CLOBBER_EMPTY

只有在未设置 CLOBBER 选项时，才会使用该选项：注意，默认情况下是设置该选项的。

如果设置了该选项，那么长度为零的常规文件可能会被改写（'clobbered'）。 需要注意的是，在本次测试和当前进程使用该文件两个时间之间，可能会有另一个进程写入该文件。 因此，如果要被 'clobbered' 的文件可能被异步写入，则不应使用此选项。

CORRECT (-0)

尝试纠正命令的拼写错误。 请注意，如果未设置 HASH_LIST_ALL 选项，或路径中的某些目录不可读，在首次使用某些命令时，可能会错误地报告拼写错误。

shell 变量 CORRECT_IGNORE 可以设置为一个模式，以匹配永远不会作为更正提供的词语。

CORRECT_ALL (-O)

尝试纠正一行中所有参数的拼写。

shell 变量 CORRECT_IGNORE_FILE 可以设置为一个模式，以匹配永远不会作为更正提供的文件名。

DVORAK

为 CORRECT 和 CORRECT_ALL 选项以及 spell-word 编辑器命令，使用 Dvorak 键盘而非标准 qwerty 键盘来检查拼写错误。

FLOW_CONTROL <D>

如果未设置该选项，则 shell 编辑器中通过开始/停止字符（通常分配给 ^S/^Q）进行的输出流控制将被禁用。

IGNORE_EOF (-7)

文件结束（end-of-file)时不退出。 要求使用 exit 或 logout 代替。 不过，连续十次 EOF 还是会导致 shell 退出，以避免在 tty 消失时 shell 挂起。

此外，如果设置了该选项并使用 Zsh 行编辑器，由 shell 函数实现的小部件可以绑定到 EOF（通常为 Control-D），而不会打印正常的警告信息。 这只适用于普通小部件，不适用于补全小部件。

INTERACTIVE_COMMENTS (-k) <K> <S>

即使在交互式 shell 中也允许注释。

HASH_CMDS <D>

首次执行每条命令时，请注意其位置。 以后调用同一命令时将使用保存的位置，避免路径搜索。 如果未设置该选项，则根本不会进行路径散列。 不过，如果设置了 CORRECT，则会对名称未出现在函数或别名哈希表中的命令进行哈希处理，以避免将其报告为拼写错误。

HASH_DIRS <D>

每当对命令名进行散列时，都会对包含该命令名的目录以及路径中前面出现的所有目录进行散列。 如果未设置 HASH_CMDS 和 CORRECT，则不会产生任何影响。

HASH_EXECUTABLES_ONLY

当因为 HASH_CMDS 对命令进行散列时，请检查要散列的文件是否为可执行文件。 默认情况下未设置此选项，因为如果路径包含大量命令，或由许多远程文件组成，额外的测试可能会花费很长时间。 该选项是否有用，还需要反复试验。

MAIL_WARNING (-U)

如果自 shell 上次检查以来有人访问过邮件文件，则打印警告信息。

PATH_DIRS (-Q)

即使命令名称中包含斜线，也会执行路径搜索。 因此，如果用户的路径中存在 '/usr/local/bin'，而用户输入了 'X11/xinit'，则将执行 '/usr/local/bin/X11/xinit' 命令（假设它存在）。 明确以 '/', './' 或 '../' 开头的命令不受路径搜索的影响。 这也适用于 '.' 和 source 内置程序。

请注意，以这种形式指定的可执行文件总是搜索当前目录下的子目录。 该搜索会在本选项指定的任何搜索之前进行,无论命令搜索路径中是否出现 '.' 或当前目录，。

PATH_SCRIPT <K> <S>

如果未设置此选项，作为第一个非选项参数传递给 shell 的脚本必须包含要打开的文件名。 如果设置了此选项，且脚本未指定目录路径，则首先在当前目录下查找脚本，然后在命令路径下查找。 参见 调用。

PRINT_EIGHT_BIT

在补全列表等中按字面打印八位字符。 如果系统能正确返回八位字符的可打印性（参见 ctype(3)），则不需要使用该选项。

PRINT_EXIT_VALUE (-1)

打印退出状态为非零的程序的退出值。 这只有在交互式 shell 的命令行中才能使用。

RC_QUOTES

允许使用字符序列 '' 表示单引号字符串中的单引号。 请注意，这不适用于使用 $'...' 格式的带引号字符串，在这种字符串中可以使用反斜线单引号。

RM_STAR_SILENT (-H) <K> <S>

在执行 'rm *' 或 'rm path/*' 之前不要查询用户。

RM_STAR_WAIT

如果在执行 'rm *' 或 'rm path/*' 之前询问用户，则首先等待十秒钟，并忽略在此期间输入的任何内容。 这就避免了在并非真心实意的情况下条件反射地回答 '是' 的问题。 通过在 ZLE 中扩展 '*'（使用制表符），可以避免等待和查询。

SHORT_LOOPS <C> <Z>

允许 for, repeat, select, if, 和 function 构造的简短形式.

SHORT_REPEAT

像 SHORT_LOOPS 一样允许使用简短形式 repeat，但不在其他结构式中启用。

SUN_KEYBOARD_HACK (-L)

如果一行以反引号结束，而该行的反引号数目为奇数，则忽略尾部的反引号。 在某些键盘上，回车键太小，而反引号键离回车键很近，这时会很有用。 作为替代，变量 KEYBOARD_HACK 可以让你选择要删除的字符。

16.2.7 作业控制

AUTO_CONTINUE

设置该选项后，使用 disown 内置命令从作业表中删除的已停止作业会自动被发送 CONT 信号，使其恢复运行。

AUTO_RESUME (-W)

将没有重定向的单字简单命令视为恢复现有作业的候选命令。

BG_NICE (-6) <C> <Z>

以较低的优先级运行所有后台任务。 该选项为默认设置。

CHECK_JOBS <Z>

在退出受作业控制的 shell 之前，报告后台作业和挂起作业的状态；第二次尝试退出 shell 将成功。 NO_CHECK_JOBS 最好与 NO_HUP 结合使用，否则此类作业会被自动杀死。

如果从上一条命令行运行的命令中包含 'jobs' 命令，则省略检查，因为我们假定用户知道存在后台作业或暂停作业。 从 函数 中 '特殊函数' 一节定义的钩子函数之一运行的 'jobs' 命令不计算在内。

CHECK_RUNNING_JOBS <Z>

启用 CHECK_JOBS 时，会同时检查运行中和暂停中的作业。 禁用此选项后，zsh 只检查暂停的作业，这与 bash 的默认行为一致。

除非设置了 CHECK_JOBS，否则该选项不起作用。

HUP <Z>

当 shell 退出时，向正在运行的作业发送 HUP 信号。

LONG_LIST_JOBS (-R)

默认以长格式打印作业通知。

MONITOR (-m, ksh: -m)

允许作业控制。 在交互式 shell 中默认设置。

NOTIFY (-5, ksh: -b) <Z>

立即报告后台作业的状态，而不是等到打印提示符前才报告。

POSIX_JOBS <K> <S>

该选项使作业控制更符合 POSIX 标准。

未设置该选项时，进入子 shell 时 MONITOR 选项将被取消设置，因此作业控制不再有效。 设置该选项后，MONITOR 选项和作业控制在子 shell 中仍然有效，但需要注意的是，子 shell 无法访问父 shell 中的作业。

不设置该选项时，用 bg 或 fg 放在后台或前台的作业会显示与 jobs 报告的相同信息。 如果设置了该选项，则只打印文本。 jobs 本身的输出不受该选项影响。

不设置该选项时，父 shell 中的作业信息会保存到子 shell（例如管道）中输出。 设置该选项后，jobs 的输出将为空，直到子 shell 中启动作业。

在以前版本的 shell 中，必须启用 POSIX_JOBS，内置命令 wait 才能返回已退出的后台作业的状态。 现在不再需要这样做了。

16.2.8 提示

PROMPT_BANG <K>

如果设置，‘!’将在提示符扩展中被特殊处理。 请参阅 提示符扩展。

PROMPT_CR (+V) <D>

在行编辑器中打印提示符前打印回车。 默认情况下是打开的，因为只有编辑器知道行的起始位置，才能进行多行编辑。

PROMPT_SP <D>

试图保留因 PROMPT_CR 选项而被命令提示符遮盖的部分行（partial line）（即未以换行结束的行）。 这将通过输出一些光标控制字符（包括一系列空格）来实现，当出现部分行时，这些字符将使终端换行到下一行（请注意，这只有在终端有自动页边距的情况下才能成功，而自动页边距是典型的）。

保留部分行时，默认情况下会在部分行的末尾显示一个反转+粗体字符：普通用户为‘%’，root 用户为‘#’。 如果设置了 shell 参数 PROMPT_EOL_MARK，则可以自定义部分行行尾的显示方式。

注意：如果未设置 PROMPT_CR 选项，启用该选项将没有任何效果。 该选项默认为开启。

PROMPT_PERCENT <C> <Z>

如果设置了该选项，‘%’ 将在提示符扩展中被特殊处理。 请参阅 提示符扩展。

PROMPT_SUBST <K> <S>

如果设置，**参数扩展**、**命令替换**和 **算术扩展**将在提示符中执行。 提示符内的替换不会影响命令状态。

TRANSIENT_RPROMPT

在接受命令行时，移除显示的任何右侧提示符。 这对使用其他剪切/粘贴方法的终端可能有用。

16.2.9 脚本和函数

ALIAS_FUNC_DEF <S>

默认情况下，如果 *name* 被展开为别名，zsh 不允许使用 '*name* ( )' 语法定义函数：这将导致错误。 这通常是我们所希望的行为，否则基于相同定义的别名和函数的组合很容易导致问题。

设置该选项后，可以使用别名来定义函数。

例如，考虑启动文件中可能出现的以下定义。

```
alias foo=bar
foo() {
  print This probably does not do what you expect.
}
```

在这里，foo 在遇到 ( ) 之前被扩展为 bar 的别名，因此定义的函数将被命名为 bar。 默认情况下，在 native 模式下这反而是一个错误。 需要注意的是，函数名的任何部分加引号或使用关键字 function 都可以避免这个问题，因此建议在函数名也可以是别名时使用。

C_BASES

以标准 C 格式输出十六进制数，例如 '0xFF' ，而不是通常的 '16#FF'。 如果同时设置了选项 OCTAL_ZEROES (默认情况下没有) ，八进制数也会被类似处理，因此会显示为 '077' 而不是 '8#77'。 该选项不会影响输出基数的选择，也不会影响十六进制和八进制以外基数的输出。 请注意，无论 C_BASES 的设置如何，这些格式在输入时都能被理解。

C_PRECEDENCES

这改变了算术运算符的优先级，使其更像 C 语言和其他编程语言；算术求值有一个显式列表。

DEBUG_BEFORE_CMD <D>

在每条命令之前运行 DEBUG陷阱，否则在每条命令之后运行。 设置该选项后，将模仿 ksh 93 的行为；不设置该选项时，将使用 ksh 88 的行为。

ERR_EXIT (-e, ksh: -e)

如果命令的退出状态为非零，则执行 ZERR 陷阱 (如果已设置) 并退出。 运行初始化脚本时，该功能将被禁用。

在 DEBUG 陷阱中也会禁用该行为。 在这种情况下，会对选项进行特殊处理：在进入陷阱时取消设置。 如果设置了 DEBUG_BEFORE_CMD 选项（默认情况），并且在退出时发现 ERR_EXIT 选项已被设置，则跳过 DEBUG 陷阱正在执行的命令。 该选项会在陷阱退出后恢复。

包含 && 或 || 的命令列表中，不在命令列表末尾的命令，其非零状态将被忽略。 因此

```
false && true
```

不会触发退出。

由于 ERR_EXIT 而退出，会与 作业和信号 中提到的异步作业产生某些交互。

ERR_RETURN

如果命令的退出状态为非零，则立即从闭合函数中返回。 其逻辑与 ERR_EXIT 类似，只是执行的是隐式 return 语句而不是 exit。 这将在非交互式脚本的最外层触发退出。

通常，该选项继承了 ERR_EXIT 的行为，即代码后跟 '&&' '||'不会触发返回。 因此，在下面的代码中

```
summit || true
```

不强制返回，因为综合效果始终为零返回状态。

但请注意，如果上例中的 summit 本身是一个函数，那么其中的代码将被单独考虑：它可能强制从 summit 返回（假设选项仍在 summit 中设置），但不会强制从闭合上下文返回。 这种行为与 ERR_EXIT 不同，后者不受函数作用域的影响。

EVAL_LINENO <Z>

如果设置了该参数，使用内置 eval 求值的表达式的行号将与封闭环境分开跟踪。这同时适用于参数 LINENO 和提示符 %i 输出的行号。 如果设置了该选项，提示符转义 %N 将输出字符串 '(eval)'，而不是脚本或函数名称作为提示。 (这两个提示符转义符通常用于设置选项 XTRACE 时，输出参数 PS4)。 如果未设置 EVAL_LINENO，脚本或函数的行号将在求值过程中保留。

EXEC (+n, ksh: +n) <D>

Do execute commands. Without this option, commands are read and checked for syntax errors, but not executed. This option cannot be turned off in an interactive shell, except when '-n' is supplied to the shell at startup.

FUNCTION_ARGZERO <C> <Z>

执行 shell 函数或引入脚本时，将 $0 暂时设置为函数/脚本的名称。 请注意，将 FUNCTION_ARGZERO 从打开切换到关闭（或从关闭切换到打开）并不会改变 $0 的当前值。 只有进入函数或脚本时的状态才会产生影响。 比较 POSIX_ARGZERO。

LOCAL_LOOPS

如果未设置该选项，break 和 continue 命令的效果可能会传播到函数作用域之外，影响调用函数中的循环。 如果在(正在)调用的函数中设置了该选项，则未被调用函数捕获的 break 或 continue（无论在<已>调用函数中是否设置了该选项）会产生警告并取消效果。

LOCAL_OPTIONS <K>

如果在从 shell 函数返回时设置了此选项，则会恢复大多数在进入函数时有效的选项（包括此选项）；不恢复的选项包括 PRIVILEGED 和 RESTRICTED。 否则，只有该选项、LOCAL_LOOPS、XTRACE 和 PRINT_EXIT_VALUE 选项会被还原。 因此，如果 shell 函数显式地取消了该选项的设置，返回时有效的其他选项仍将保持不变。 shell 函数也可以通过类似于 'emulate -L zsh' 的表述来保证自己拥有已知的 shell 配置；-L 会激活 LOCAL_OPTIONS。

LOCAL_PATTERNS

如果在 shell 函数返回时设置了该选项，则通过内置命令 'disable -p'设置的模式禁用状态将恢复到进入函数时的状态。 该选项的作用类似于 LOCAL_OPTIONS 对选项的作用；因此，'emulate -L sh'（或其他使用 -L 选项的仿真）会激活 LOCAL_PATTERNS。

LOCAL_TRAPS <K>

如果在函数内部设置信号陷阱时设置了该选项，那么当函数退出时，该信号的陷阱状态将被恢复。请注意，必须在更改函数中的陷阱行为之前设置该选项；与 LOCAL_OPTIONS 不同的是，退出函数时的值并不重要。 不过，全局陷阱并不需要在函数正确恢复之前设置。 例如

```
unsetopt localtraps
trap - INT
fn() { setopt localtraps; trap '' INT; sleep 3; }
```

将在函数退出后恢复对 SIGINT 的正常处理。

MULTI_FUNC_DEF <Z>

允许以 'fn1 fn2... ()' 的形式同时定义多个函数；如果未设置该选项，则会导致解析错误。 始终允许使用 function 关键字定义多个函数。 多重函数定义并不常用，可能会导致不明显的错误。

MULTIOS <Z>

当尝试多次重定向时，执行隐式 *tee*s 或 *cat*s （请参阅 重定向）。

OCTAL_ZEROES <S>

根据 IEEE 标准 1003.2-1992（ISO 9945-2:1993），将以 0 开头的整数常量解释为八进制。 默认情况下不启用此功能，因为它会给日期和时间字符串等前导零字符串的解析带来问题。

表示数值基数的数字序列，如 '08#77' 中的 '08' 分量，始终被解释为十进制，与前导零无关。

PIPE_FAIL

默认情况下，当管道退出时，shell 记录并由 shell 变量 $? 返回的退出状态，反映的是管道最右端元素的状态。 如果设置了该选项，退出状态将反映管道最右端元素的非零状态，如果所有元素都以零状态退出，则返回零状态。

SOURCE_TRACE

如果设置了该选项，zsh 将打印一条信息消息，公布所加载的每个文件的名称。 输出格式与 XTRACE 选项类似，信息为 <sourcetrace>。 文件可以在 shell 启动和关闭时由 shell 本身加载（启动/关闭文件），也可以通过使用 'source' 和 'dot' 内置命令加载。

TYPESET_SILENT

如果未设置该选项，在执行任何不带任何选项的 'typeset' 系列命令时，如果参数列表中已经存在的参数没有被赋值，则会显示该参数的值。 如果设置了该选项，则只有在使用 '-m' 选项选择参数时，才会显示这些参数。 无论是否设置了该选项，都可以使用选项 '-p'。

TYPESET_TO_UNSET <K> <S>

当使用 'typeset' 系列相关命令声明一个新参数时，除非在 'typeset' 命令本身或稍后的赋值语句中明确为其赋值，否则该参数将保持未设置状态。

VERBOSE (-v, ksh: -v)

在读取 shell 输入行时将其打印出来。

XTRACE (-x, ksh: -x)

在执行命令时打印命令及其参数。 输出结果的前面是 $PS4 的值，格式如 提示符扩展 中所述。

16.2.10 Shell 仿真

APPEND_CREATE <K> <S>

该选项仅在 NO_CLOBBER （-C）有效时适用。

如果未设置该选项，当在一个不存在的文件上使用追加重定向（>>）时，shell 将报错（传统的 zsh 行为 NO_CLOBBER）。 如果设置了该选项，则不会报错（POSIX 行为）。

BASH_REMATCH

设置后，使用 =~ 操作符执行的匹配将设置 BASH_REMATCH 数组变量，而不是默认的 MATCH 和 match 变量。 BASH_REMATCH 数组的第一个元素将包含整个匹配文本，随后的元素将包含提取的子串。 如果同时设置了 KSH_ARRAYS，整个匹配部分将存储在索引 0 处，第一个子串存储在索引 1 处，那么该选项就更有意义了。 如果不使用该选项，MATCH 变量将包含整个匹配文本，而 match 数组变量将包含子串。

BSD_ECHO <S>

Make the echo builtin compatible with the BSD echo(1) command. This disables backslashed escape sequences in echo strings unless the -e option is specified.

CONTINUE_ON_ERROR

如果遇到致命错误（参见 错误），并且代码是在脚本中运行，shell 将在脚本的下一条语句的顶层恢复执行，换句话说，是在所有函数或 shell 结构（如循环和条件）之外恢复执行。 这模仿了交互式 shell 的行为，即 shell 返回行编辑器读取新命令；这是 5.0.1 之前的 zsh 版本的正常行为。

CSH_JUNKIE_HISTORY <C>

不带事件指示符的历史引用，将始终指向上一条命令。 如果没有该选项，历史记录引用将与当前命令行中的前一个历史记录引用指向相同的事件，默认为前一个命令。

CSH_JUNKIE_LOOPS <C>

允许循环体采用 '*list*; end' 的形式，而不是'do *list*; done'。

CSH_JUNKIE_QUOTES <C>

更改单引号和双引号文本的规则，使其与 *csh* 的规则一致。 这些规则要求内嵌换行符前必须有反斜线；未转义的换行符将导致错误信息。 在双引号字符串中，无法转义'$', '`' 或 '"'（'\' 本身也不再需要转义）。 命令替换只扩展一次，不能嵌套。

CSH_NULLCMD <C>

在运行无命令重定向时，请勿使用 NULLCMD 和 READNULLCMD 的值。 这会导致重定向失败（参见 重定向）。

KSH_ARRAYS <K> <S>

尽可能模拟 *ksh* 数组处理。 如果设置了该选项，数组元素将从零开始编号，不带下标的数组参数将指向第一个元素，而不是整个数组，并且需要使用大括号来分隔下标（ '${path[2]}' 而不只是 '$path[2]' ）或对任何参数应用修饰符（'${PWD:h}' 而不是 '$PWD:h'）。

KSH_AUTOLOAD <K> <S>

模拟 *ksh* 函数的自动加载。 这意味着当一个函数被自动加载时，相应的文件只是被执行，而且必须定义函数本身。 (默认情况下，函数是根据文件内容定义的。 不过，最常见的 *ksh* 风格情况–文件仅包含函数的简单定义–总是以 *ksh* 兼容的方式处理）。

KSH_OPTION_PRINT <K>

改变选项设置的打印方式：不再单独列出已设置和未设置的选项，而是显示所有选项，如果处于非默认状态，则标记为 "on"，否则标记为 "off"。

KSH_TYPESET

该选项现已过时：使用保留字接口 declare、export、float、integer、local、readonly 和 typeset 可以更好地适应其他 shell 的行为。 需要注意的是，只有在**不** 使用保留字接口时，才会应用该选项。

改变 typeset系列命令的参数处理方式，包括 declare、export、float、integer、local 和 readonly。 如果没有此选项，zsh 会在赋值参数中的命令和参数扩展后执行正常的分词；如果有此选项，则不会执行分词。

KSH_ZERO_SUBSCRIPT

将数组或字符串表达式中的零值下标视为对第一个元素的引用，即通常下标为 1 的元素。 如果同时设置了 KSH_ARRAYS 则忽略。

如果既没有设置这个选项，也没有设置 KSH_ARRAYS，那么在访问数组或字符串时，如果下标为零，则会返回空元素或空字符串，而试图设置数组或字符串的零元素则会被视为错误。 但是，如果尝试设置包含零的有效下标范围，则会成功。 例如，如果 KSH_ZERO_SUBSCRIPT 没有被设置，

```
array[0]=(element)
```

是一个错误，而

```
array[0,1]=(element)
```

不是，并且会替换数组的第一个元素。

该选项是为了与旧版 shell 兼容，不建议在新代码中使用。

POSIX_ALIASES <K> <S>

设置该选项后，保留字将不作为别名扩展的候选字：仍可将其中任何一个声明为别名，但该别名将永远不会被扩展。 保留字在 保留字 中有描述。

别名扩展是在读取文本时进行的；因此，当设置了该选项后，直到作为一个单元解析的函数或其他 shell 代码结束时才会生效。 请注意，即使该选项生效，也可能导致与其他 shell 的差异。 例如，当使用 'zsh -c'，甚至 'zsh -o posixaliases -c' 运行命令时，整个命令参数会被解析为一个单元，因此在参数中定义的别名即使在后面的行中也无法使用。 如有疑问，请避免在非交互代码中使用别名。

POSIX_ARGZERO

此选项可用于暂时禁用 FUNCTION_ARGZERO，从而将 $0 的值恢复为调用 shell 时使用的名称（或由 -c 命令行选项设置的名称）。 为了与以前版本的 shell 兼容，仿真会使用 NO_FUNCTION_ARGZERO 而不是 POSIX_ARGZERO，如果在函数或脚本中改变仿真模式，可能会导致 $0 的范围超出预期。 为避免这种情况，请在 emulate 命令中明确启用 POSIX_ARGZERO：

```
emulate sh -o POSIX_ARGZERO
```

请注意，除非 FUNCTION_ARGZERO 在进入函数或脚本时已经启用，否则 NO_POSIX_ARGZERO 不会产生任何影响。

POSIX_BUILTINS <K> <S>

设置该选项后，command 内置命令可用于执行 shell 内置命令。 在 shell 函数和特殊内置命令前指定的参数赋值会在命令补全后保留，除非特殊内置命令前缀为 command 内置命令。 特殊内置命令包括 ., :, break, continue, declare, eval, exit, export, integer, local, readonly, return, set, shift, source, times, trap 和 unset。

此外，与上述内置程序或 exec 相关的各种错误条件会导致非交互式 shell 退出和交互式 shell 返回顶层处理。

此外，函数和 shell 内置命令不会在 exec 前缀后执行；要执行的命令必须是在路径中找到的外部命令。

此外，getopts 内置命令的行为与 POSIX 兼容，相关变量 OPTIND 并非函数的本地变量，其值的计算方式也与其他 shell 不同。

此外，[[-o *non_existent_option*]] 的警告和特殊退出代码也会被抑制。

POSIX_IDENTIFIERS <K> <S>

设置该选项后，标识符（shell 参数和模块的名称）中只能使用 ASCII 字符 a 至 z、A 至 Z、0 至 9 和 _。

此外，设置该选项会限制无括号参数替换的效果，因此表达式 $# 会被视为参数 $#，即使后面跟了一个有效的参数名。 未设置该选项时，zsh 允许 $#*name* 形式的表达式引用 $*name* 的长度，即使是特殊变量，例如 $#- 和 $#* 这样的表达式。

另一个区别是，在算术上下文中对未设置变量赋值时，选项设置会导致变量创建为标量而非数值类型。 因此，在‘unset t; ((t = 3))’之后，如果没有设置 POSIX_IDENTIFIERS，t 将是整数类型，而如果设置了 POSIX_IDENTIFIERS，t 将是标量类型。

如果未设置该选项，但启用了多字节字符支持（即编译了多字节字符支持并设置了 MULTIBYTE 选项），则可在标识符中使用本地字符集中的任何字母数字字符。 需要注意的是，使用此功能编写的脚本和函数不可移植，而且必须在解析脚本或函数之前设置这两个选项；在执行过程中设置这两个选项是不够的，因为语法 *variable*=*value* 已被解析为命令而非赋值。

如果 shell 未编译多字节字符支持，该选项将被忽略；所有设置了最高位的八位字节都可用于标识符。 这是非标准的，但却是传统的 zsh 行为。

POSIX_STRINGS <K> <S>

该选项会影响带引号字符串的处理。 目前，它只影响空字符的行为，即对应于 US ASCII 的可移植字符集中的字符 0。

如果不设置该选项，嵌入 $'…' 格式字符串中的空字符将被视为普通字符。整个字符串将保留在 shell 中，并在必要时输出到文件中，但由于库接口的限制，字符串会在文件名、环境变量或外部程序参数中的空字符处被截断。

设置该选项后， $'…' 表达式将在空字符处截断。 请注意，同一字符串中引号结束后的其余部分不会被截断。

例如，命令行参数 a$'b\0c'd 在选项关闭的情况下被视为 a、b、null、c、d 字符，而在选项开启的情况下被视为 a、b、d 字符。

POSIX_TRAPS <K> <S>

设置此选项后，zsh 通常在退出 shell 函数时执行 EXIT 陷阱的行为将被抑制。 在这种情况下，对 EXIT 陷阱的操作总是会改变用于退出 shell 的全局陷阱；对于 EXIT 陷阱，LOCAL_TRAPS 选项会被忽略。

此外，在没有参数的陷阱中执行 return 语句时，函数会传回周围上下文的值，而不是陷阱中执行的代码的值。

此外，如果陷阱被设置为忽略，那么当进入子 shell 时，这种状态会持续存在。 如果没有该选项，陷阱此时会重置为默认状态。

SH_FILE_EXPANSION <K> <S>

在参数扩展、命令替换、算术扩展和括号扩展**之前**，执行文件名扩展（例如 ~ 扩展）。 如果未设置该选项，则在括号扩展 **之后**执行，因此类似于 '~$USERNAME' 和'~{pfalstad,rc}' 将正常工作。

SH_NULLCMD <K> <S>

重定向时，请勿使用 NULLCMD 和 READNULLCMD 的值，而应使用 ':'（请参阅 重定向）。

SH_OPTION_LETTERS <K> <S>

如果设置了该选项，shell 会尝试像 *ksh* 那样解释单字母选项（与 set 和 setopt 一起使用）。 这也会影响 - 特殊参数的值。

SH_WORD_SPLIT (-y) <K> <S>

使字段分割在无引号的参数扩展上执行。 请注意，此选项与分词无关。 (参见 参数扩展）。

TRAPS_ASYNC

在等待程序退出时，应立即处理信号和运行陷阱。 否则，陷阱将在子进程退出后运行。 请注意，除了 shell 等待子进程外，这不会影响运行陷阱的时间点。

16.2.11 Shell 状态

INTERACTIVE (-i, ksh: -i)

这是一个交互式 shell。 如果标准输入是 tty 且命令是从标准输入读取的，则初始化时会设置该选项（参见 SHIN_STDIN 的讨论）。可以通过在命令行中指定该选项的状态来覆盖这一启发式方法。 该选项的值只能通过调用 shell 时提供的标志来更改。 zsh 运行后就无法更改。

LOGIN (-l, ksh: -l)

这是一个登录 shell。 如果未明确设置此选项，则如果传给 shell 的 argv[0] 第一个字符是 '-'，则 shell 将成为登录 shell。

PRIVILEGED (-p, ksh: -p)

开启特权模式。通常情况下，当脚本需要以提升的权限运行时会用到。具体操作如下：直接使用 zsh 的 -p 选项，使其在启动时生效。

```
#!/bin/zsh -p
```

如果有效用户（组）ID 不等于真实用户（组）ID，则启动时会自动启用该选项。在这种情况下，关闭该选项会将有效用户和组 ID 设置为真实用户和组 ID。需要注意

的是，如果该选项失效，shell 运行时的 ID 可能与预期不同，因此脚本应检查是否失效并采取相应措施：

```
unsetopt privileged || exit
```

PRIVILEGED 选项禁止引入用户启动文件。 如果在设置了该选项后以 'sh' 或 'ksh' 的方式调用 zsh，则会引入 /etc/suid_profile（在交互式 shell 中位于 /etc/profile 之后）。~/.profile 将禁止引入，并且 ENV 变量的内容将被忽略。使用 setopt 和 unsetopt 的 -m 选项无法更改该选项，在函数内部更改该选项时，无论是否使用 LOCAL_OPTIONS 选项，都会全局更改该选项。

RESTRICTED (-r)

启用受限模式。 使用 unsetopt 无法更改此选项，在函数中设置此选项时，无论是否使用 LOCAL_OPTIONS 选项，都会全局更改此选项。 参见 受限的 Shell。

SHIN_STDIN (-s, ksh: -s)

从标准输入端读取命令。 如果没有使用 -c 指定命令，也没有指定命令文件，则会从标准输入端读取命令。 如果在命令行中明确设置了 SHIN_STDIN，任何本应作为运行文件的参数都将被视为普通位置参数。 需要注意的是，在命令行中设置或取消此选项，并不一定会影响 shell 运行时该选项的状态–这纯粹是一个指标，表明命令是否 **真的** 从标准输入端读取。 该选项的值只能通过调用 shell 时提供的标志来更改。 zsh 运行后将无法更改。

SINGLE_COMMAND (-t, ksh: -t)

如果 shell 从标准输入读取数据，则会在执行完一条命令后退出。 除非在命令行中明确设置 INTERACTIVE 选项，否则 shell 是非交互的。 该选项的值只能通过调用 shell 时提供的标志来更改。 zsh 运行后将无法更改。

16.2.12 Zle

BEEP (+B) <D>

ZLE 出错时发出蜂鸣声。

COMBINING_CHARS

假设终端能正确显示组合字符。 具体来说，如果一个基本字母数字字符后面有一个或多个零宽度标点符号字符，则假定零宽度字符将显示为在相同宽度内对基本字符的修改。 并非所有终端都会这样处理。 如果不设置此选项，零宽度字符将以特殊标记单独显示。

如果设置了该选项，则模式测试 [[:WORD:]] 将匹配零宽度标点符号，前提是该字符将作为单词的一部分与单词字符结合使用。 否则，基本 shell 不会特别处理组合字符。

EMACS

如果已加载 ZLE，则打开该选项的效果等同于 'bindkey -e'。 此外，VI 选项是未设置的。 关闭该选项没有任何效果。 该选项的设置不能保证反映当前的键映射。 提供该选项是为了兼容；bindkey 是推荐的接口。

OVERSTRIKE

以叠加模式启动行编辑器。

SINGLE_LINE_ZLE (-M) <K>

使用单行命令行编辑，而不是多行编辑。

请注意，尽管在 ksh 模拟中该选项是默认开启的，但它仅提供了与 ksh 行编辑器的表面兼容性，并降低了 zsh 行编辑器的有效性。 由于它对 shell 语法没有影响，许多用户可能希望在交互式使用 ksh 模拟时禁用该选项。

VI

如果已加载 ZLE，则打开该选项的效果等同于 'bindkey -v'。 此外，EMACS 选项未设置。 关闭该选项没有任何效果。 选项设置并不能保证反映当前的键映射。 提供此选项是为了兼容；bindkey 是推荐的接口。

ZLE (-Z)

使用 zsh 行编辑器。 在连接到终端的交互式 shell 中默认设置。

16.3 Option 别名

有些选项有别名。 这些别名从不用于输出，但在向 shell 指定选项时，可以像普通选项名一样使用。

BRACE_EXPAND

*NO*_IGNORE_BRACES (ksh 和 bash 兼容性)

DOT_GLOB

GLOB_DOTS (bash 兼容性)

HASH_ALL

HASH_CMDS (bash 兼容性)

HIST_APPEND

APPEND_HISTORY (bash 兼容性)

HIST_EXPAND

BANG_HIST (bash 兼容性)

LOG

*NO_*HIST_NO_FUNCTIONS (ksh 兼容性)

MAIL_WARN

MAIL_WARNING (bash 兼容性)

ONE_CMD

SINGLE_COMMAND (bash 兼容性)

PHYSICAL

CHASE_LINKS (ksh 和 bash 兼容性)

PROMPT_VARS

PROMPT_SUBST (bash 兼容性)

STDIN

SHIN_STDIN (ksh 兼容性)

TRACK_ALL

HASH_CMDS (ksh 兼容性)

16.4 单字母选项

16.4.1 默认集(set)

-0

CORRECT

-1

PRINT_EXIT_VALUE

-2

*NO*_BAD_PATTERN

-3

*NO*_NOMATCH

-4

GLOB_DOTS

-5

NOTIFY

-6

BG_NICE

-7

IGNORE_EOF

-8

MARK_DIRS

-9

AUTO_LIST

-B

*NO*_BEEP

-C

*NO*_CLOBBER

-D

PUSHD_TO_HOME

-E

PUSHD_SILENT

-F

*NO*_GLOB

-G

NULL_GLOB

`-H`

RM_STAR_SILENT

`-I`

IGNORE_BRACES

`-J`

AUTO_CD

`-K`

NO_BANG_HIST

`-L`

SUN_KEYBOARD_HACK

`-M`

SINGLE_LINE_ZLE

`-N`

AUTO_PUSHD

`-O`

CORRECT_ALL

`-P`

RC_EXPAND_PARAM

`-Q`

PATH_DIRS

`-R`

LONG_LIST_JOBS

`-S`

REC_EXACT

`-T`

CDABLE_VARS

`-U`

MAIL_WARNING

`-V`

NO_PROMPT_CR

`-W`

AUTO_RESUME

`-X`

LIST_TYPES

`-Y`

MENU_COMPLETE

`-Z`

ZLE

`-a`

ALL_EXPORT

`-e`

ERR_EXIT

`-f`

NO_RCS

`-g`

HIST_IGNORE_SPACE

`-h`

HIST_IGNORE_DUPS

`-i`

INTERACTIVE

`-k`

INTERACTIVE_COMMENTS

`-l`

LOGIN

`-m`

MONITOR

`-n`

NO_EXEC

`-p`

PRIVILEGED

`-r`

RESTRICTED

`-s`

SHIN_STDIN

`-t`

SINGLE_COMMAND

`-u`

NO_UNSET

`-v`

VERBOSE

`-w`

CHASE_LINKS

`-x`

XTRACE

`-y`

SH_WORD_SPLIT

16.4.2 sh/ksh 模拟集

`-C`

*NO*_CLOBBER

`-T`

TRAPS_ASYNC

`-X`

MARK_DIRS

`-a`

ALL_EXPORT

`-b`

NOTIFY

`-e`

ERR_EXIT

`-f`

*NO*_GLOB

`-i`

INTERACTIVE

`-l`

LOGIN

`-m`

MONITOR

`-n`

*NO*_EXEC

`-p`

PRIVILEGED

`-r`

RESTRICTED

`-s`

SHIN_STDIN

`-t`

SINGLE_COMMAND

`-u`

NO_UNSET

`-v`

VERBOSE

`-x`

XTRACE

16.4.3 另请注意

`-A`

由 `set` 使用，用于设置数组

`-b`

在命令行中使用，用于指定选项处理的结束

`-c`

在命令行中使用，用于指定单一命令

`-m`

由 `setopt` 使用，用于模式匹配选项设置

`-o`

在所有地方使用，允许使用长选项名

`-s`

被 `set` 用来对位置参数排序

---

17 Shell 内置命令

某些 shell 内置命令会使用个别条目中描述的选项；这些选项在下面的列表中通常被称为‘标志’，以避免与 shell 选项混淆，后者也可能对内置命令的行为产生影响。 在本介绍性章节中，‘选项’总是指大多数命令行用户熟悉的命令选项。

通常情况下，选项为单个字母，前面有一个连字符（`-`）。 带参数的选项要么紧跟在选项字母之后，要么在空白处之后接受参数，例如，‘`print -C3 {1..9}`’和‘`print -C 3 {1..9}`’是等价的。 选项的参数与命令的参数不同；文档中会说明哪个是哪个。 不带参数的选项可以合并为一个单词，例如，‘`print -rca -- *`’和‘`print -r -c -a -- *`’是等价的。

某些 shell 内置命令的选项也是以‘+’开头，而不是以‘-’开头。 下面的列表清楚地列出了这些命令。

选项（及其单个参数（如果有））必须先于非选项参数出现在一组中；一旦找到第一个非选项参数，选项处理就会终止。

除‘echo’和前置命令修饰符之外的所有内置命令，甚至那些没有选项的命令，都可以使用参数‘`--`’来终止选项处理。 这表明后面的词是非选项参数，但在其他情况下会被忽略。 这在命令参数可能以‘-’开头的情况下非常有用。 由于历史原因，大多数内置命令（包括‘echo’）也会为此目的在单独的单词中识别单个‘-’；请注意，这不是标准的做法，建议使用‘`--`’。

- simple command

请参阅 前置命令修饰符

. file [*arg* ...]

从 *file* 中读取命令，并在当前 shell 环境中执行。

如果 *file* 不包含斜线，或者 `PATH_DIRS` 已被设置，shell 会在 `$path` 的组件中查找包含 *file* 的目录。 除非 `$path` 中出现了‘`.`’，否则不会读取当前目录下的文件。 如果找到名为‘*file*`.zwc`’的文件，该文件比 *file* 新，并且是 *file* 的编译形式（使用 `zcompile` 内置程序创建），则将从该文件而不是 *file* 读取命令。

如果给定了任何参数 *arg*，它们就会成为位置参数；当 *file* 执行完毕后，旧的位置参数会被还原。 但是，如果没有给出参数，位置参数仍然是调用上下文中的参数，不会被还原。

如果 *file* 未找到，则返回状态为 127；如果 *file* 找到了，但包含语法错误，则返回状态为 126；否则返回状态为最后执行命令的退出状态。

: [*arg* ...]

该命令不执行任何操作，但会执行正常的参数扩展，这可能会对 shell 参数产生影响。 返回的退出状态为零。

`alias` [{+|-}`gmrsL`] [*name*[=*value*] ...]

对于每个带有相应 *value* 的 *name*，定义一个带有该值的别名。 如果 *value* 的尾部有空格，则会检查下一个单词的别名扩展情况。 如果存在 `-g` 标志，则定义全局别名；全局别名即使不出现在命令位置，也会被扩展：

```
% perldoc --help 2>&1 | grep 'built-in functions'
    -f   Search Perl built-in functions
% alias -g HG='--help 2>&1 | grep'
% perldoc HG 'built-in functions'
    -f   Search Perl built-in functions
```

如果存在 `-s` 标志，则定义后缀别名：如果命令行中的命令字格式为 '*text*`.`*name*'，其中 *text* 为任何非空字符串，则该命令字将被文本 '*value text*`.`*name*' 替换。 请注意，*name* 被视为字面字符串，而不是模式。 在这种情况下，*value* 中的尾部空格并不特殊。 例如

```
alias -s ps='gv --'
```

将导致命令 '`*.ps`' 扩展为 '`gv -- *.ps`'。 由于别名扩展比 globbing 进行得更早，'`*.ps`' 随后也将被扩展。 后缀别名与其他别名构成不同的名称空间（因此在上述示例中，仍然可以为命令 `ps` 创建一个别名），这两组别名永远不会列在一起。

对于没有 *value* 的 *name*，打印 *name* 的值。 如果没有参数，则打印当前定义的所有别名（后缀别名除外）。 如果给定了 `-m` 标志，参数将作为模式（应加引号以防止被解释为 glob 模式），并打印与这些模式匹配的别名。 在打印别名时，如果有 `-g`、`-r` 或 `-s` 标志，则分别限制为打印全局别名、常规别名或后缀别名；常规别名既不是全局别名，也不是后缀别名。 使用 '`+`' 而不是 '`-`'，或以单个'`+`' 结束选项列表，可以防止打印别名的值。

如果存在 `-L` 标志，则以适合放入启动脚本的方式打印每个别名。 如果给出的 *name*（没有 *value*）没有定义别名，则退出状态为非零。

有关别名的更多信息，包括常见问题，请参阅 别名。

`autoload` [{+|-}`RTUXdkmrtWz`] [`-w`] [*name* ...]

详情请参阅 函数 中的 '自动加载函数' 部分。 首次引用函数时，将搜索 `fpath` 参数以查找函数定义。

如果 *name* 包含一个绝对路径，函数将被定义为从给定文件加载（像往常一样搜索给定位置的转储文件）。 函数名称是文件的基名（非目录部分）。 如果在给定位置找不到函数，通常会出错；但如果给出选项 `-d`，则搜索函数时默认使用 `$fpath`。

如果函数是通过绝对路径加载的，那么从该函数加载的任何标记为 autoload 但没有绝对路径的函数，都会将父函数的加载路径暂时预置为 $fpath。

如果给出选项 -r 或 -R，则会立即搜索函数，并在内部记录位置，供执行函数时使用；相对路径会使用 $PWD 的值展开。 这样可以防止 $fpath 在调用 autoload 后发生变化。 使用 -r 时，如果函数未找到，则会静默地将其搁置，直到执行为止；如果使用 -R，则会打印错误信息，并在搜索失败后立即终止命令处理，即在 autoload 命令执行时终止，而不是在函数执行时终止。

标志 -X 只能在 shell 函数内部使用。 它会将调用的函数标记为自动加载，然后立即加载并执行，并将当前的位置参数数组作为参数。 这将替换之前的函数定义。 如果没有找到函数定义，则会打印错误信息，函数将保持未定义状态并标记为自动加载。 如果给定了一个参数，则该参数将被用作查找函数的目录（即不包含函数名称）；如果函数不在给定的位置，则可以结合 -d 选项，将函数搜索默认为 $fpath。

标志 +X 会尝试将 *name* 作为自动加载函数加载，但 **不会** 执行该函数。 如果函数之前未定义过 **并且** 找到了函数的定义，则退出状态为零（成功）。 这不会 **不会** 替换函数的任何现有定义。 如果函数已被定义或未找到定义，则退出状态为非零（失败）。 在后一种情况下，函数将保持未定义状态，并标记为自动加载。 如果启用了 ksh 样式的自动加载，创建的函数将包含文件内容以及附加到函数本身的调用，从而在首次调用函数时提供正常的 ksh 自动加载行为。 如果同时给出 -m 标志，则每个 *name* 都会被视为一个模式，所有已经标记为自动加载的、与该模式匹配的函数都会被加载。

使用 -t 标志，开启执行跟踪；使用 -T，仅对当前函数开启执行跟踪，在进入任何未开启跟踪的被调用函数时关闭跟踪。

使用 -U 标志时，在加载函数时会抑制别名扩展。

使用 -w 标志时，*name*s 将被视为使用 zcompile 内置程序编译的文件名，其中定义的所有函数都会被标记为自动加载。

标志 -z 和 -k 分别标记函数将使用 zsh 或 ksh 样式自动加载，如同选项 KSH_AUTOLOAD 未设置或已设置。 这些标志会覆盖加载函数时的选项设置。

请注意，autoload 命令并不试图确保在加载或执行文件时设置的 shell 选项具有任何特定值。 为此，可以使用 emulate 命令：

```
emulate zsh -c 'autoload -Uz func'
```

安排在加载 *func* 时使用原生 zsh 模拟 shell，在运行 *func* 时也使用这种模拟。

autoload 的某些功能也由 functions -u 或 functions -U 提供，但 autoload 是一个更全面的接口。

bg [*job* ...]

job ... &

将每个指定的 *job* 置于后台，如果没有指定，则将当前工作置于后台。

bindkey

参见 Zle 内置命令。

break [*n*]

从闭合的 for、while、until、select 或 repeat 循环中退出。如果指定了算术表达式 *n*，则会中断 *n* 层，而不是只中断一层。

builtin *name* [*args* ...]

使用给定的 *args* 执行 *name* 内置命令。

bye

与 exit 相同。

cap

参见 zsh/cap 模块.

cd [-qsLP] [*arg*]
cd [-qsLP] *old new*
cd [-qsLP] {+|-}*n*

更改当前目录。 在第一种形式中，将当前目录更改为 *arg*，如果未指定 *arg*，则更改为 $HOME 的值。 如果 *arg* 是 '-'，则会更改为前一个目录。

否则，如果 *arg* 以斜线开头，则尝试切换到 *arg* 指定的目录。

如果 *arg* 并非以斜线开头，其行为取决于当前目录 '.'是否出现在 shell 参数 cdpath 包含的目录列表中。 如果没有，则首先尝试更改为当前目录下的 *arg* 目录，如果失败，但 cdpath 已设置且包含至少一个元素，则尝试依次更改为 cdpath 的每个组件下的 *arg* 目录，直至成功。 如果 '.' 出现在 cdpath 中，那么 cdpath 将严格按照顺序进行搜索，这样 '.' 只会在适当的位置被尝试。

如果设置了 POSIX_CD 选项，则会修改 cdpath 的测试顺序，具体请参见该选项的文档。

如果未找到目录，但选项 CDABLE_VARS 已设置，且存在以斜线开头的名为 *arg* 的参数，则将其值视为目录。 在这种情况下，该参数会被添加到已命名的目录哈希表中。

cd 的第二种形式是用 *new* 字符串替换当前目录名称中的 *old* 字符串，并尝试更改为这个新目录。

cd 的第三种形式是从目录堆栈中提取一个条目，并更改为该目录。 形式为 '+*n*' 的参数从 dirs 命令显示的列表左侧开始计数(从 0 开始)，以此来识别堆栈条目。 形式为 '-*n*' 的参数从右边开始计数。 如果设置了 PUSHD_MINUS 选项，'+' 和 '-' 在此处的含义将互换。 如果设置了 POSIX_CD 选项，cd 的这种形式将不被识别，并被解释为第一种形式。

如果指定了 -q（quiet）选项，则不会调用钩子函数 chpwd 和数组 chpwd_functions 中的函数。 这对于调用 cd 且不会改变交互式用户所看到的环境非常有用。

如果指定了 -s 选项，cd 会在给定路径名包含符号链接的情况下拒绝更改当前目录。 如果指定了 -P 选项或设置了 CHASE_LINKS 选项，符号链接将被解析为其真实值。 如果给定了 -L 选项，无论 CHASE_LINKS 选项的状态如何，符号链接都会保留在目录中（而不会被解析）。

chdir

与 cd 一样。

clone

参见 zsh/clone 模块.

command [-pvV] *simple command*

简单命令参数将作为外部命令而非函数或内置命令执行。如果设置了 POSIX_BUILTINS 选项，内置命令也会被执行，但它们的某些特殊属性会被抑制。-p 标志会导致搜索默认路径，而不是 $path 中的路径。使用 -v 标志时，command 类似于 whence，而使用 -V 时，则等同于 whence -v。

另请参阅 前置命令修饰符。

comparguments

参见 zsh/computil 模块.

compcall

参见 zsh/compctl 模块.

compctl

参见 zsh/compctl 模块.

compdescribe

参见 zsh/computil 模块.

compfiles

参见 zsh/computil 模块.

compgroups

参见 zsh/computil 模块.

compquote

参见 zsh/computil 模块.

comptags

参见 zsh/computil 模块.

comptry

参见 zsh/computil 模块.

compvalues

参见 zsh/computil 模块.

continue [*n*]

继续 for、while、until、select 或 repeat 循环的下一次迭代。如果指定了算术表达式 *n*，则跳出 *n*-1 循环，继续第 *n* 个封闭循环。

declare

与 typeset 相同。

dirs [-c] [*arg* ...]
dirs [-lpv]

在没有参数的情况下，打印目录栈的内容。 使用 pushd 命令可将目录添加到目录栈，使用 cd 或 popd 命令可将目录移除。 如果指定了参数，则会将参数加载到目录栈，替换原有的参数，并将当前目录推入目录栈。

-c

清空目录栈。

-l

打印完整目录名，而不是使用 ~ 表达式（文件名扩展）。

-p

每行打印一个目录条目。

`-v`

打印时对堆栈中的目录编号。

`disable` [`-afmprs`] *name* ...

暂时禁用名为 *name* 的哈希表元素或模式。 默认情况下禁用内置命令。 这样就可以使用与内置命令同名的外部命令。 `-a` 选项会使 `disable` 作用于常规或全局别名。 `-s` 选项会使 `disable` 作用于后缀别名。 通过 `-f` 选项，`disable` 将作用于 shell 函数。 通过 `-r` 选项，`disable` 将作用于保留字。 在不带参数的情况下，会打印相应哈希表中所有禁用的哈希表元素。 使用 `-m` 标志时，参数将被视为模式（应加引号以防止其进行文件名扩展），相应哈希表中与这些模式匹配的所有哈希表元素都会被禁用。 禁用对象可以使用 `enable` 命令启用。

用选项 `-p`，*name* ... 指的是 文件名生成 中描述的 shell 模式语法元素。 某些元素可以单独禁用，如下所示。

请注意，选项 `EXTENDED_GLOB`、`KSH_GLOB` 和 `SH_GLOB` 的当前设置不允许的模式将永远无法启用，与此处的设置无关。 例如，如果 `EXTENDED_GLOB` 未激活，即使未发出‘`disable -p "^"`’，模式 `^` 也不会生效。 下面列出了限制使用该模式的选项设置。 需要注意的是，设置 `SH_GLOB` 不仅仅会禁用模式，还会产生更广泛的影响，因为某些表达式，特别是涉及括号的表达式，会以不同的方式进行解析。

可以禁用以下模式；所有字符串都需要在命令行中加注引号，以防止被立即解释为模式，下面显示的这些模式用单引号括起来，以示提醒。

`'?'`

无论出现在何处，包括在使用 `KSH_GLOB` 选项时紧跟在括号之前的模式字符 `?`。

`'*'`

模式字符 `*` 出现的任何地方，包括递归 globbing 以及在括号前使用（用 `KSH_GLOB`）时。

`'['`

字符类。

`'<'` (`NO_SH_GLOB`)

数值范围。

`'|'` (`NO_SH_GLOB`)

分组模式、case 语句或 `KSH_GLOB` 括弧表达式中的变换（Alternation）。

'(' (NO_SH_GLOB)

使用单一括号分组。 在 KSH_GLOB 中，如果括号是由特殊字符引入的，禁用此功能并不会禁用括号，也不会禁用 glob 修饰符（使用 'setopt NO_BARE_GLOB_QUAL' 禁用仅使用括号的 glob 修饰符）。

'~' (EXTENDED_GLOB)

以 *A*~*B* 的形式排除。

'^' (EXTENDED_GLOB)

以 *A*^*B* 的形式排除。

'#' (EXTENDED_GLOB)

模式字符 # 出现的地方，既表示重复前一个模式，也表示 globbing 标志。

'?(' (KSH_GLOB)

分组形式 ?(...)。 请注意，如果禁用了 '?'，该功能也会被禁用。

'*(' (KSH_GLOB)

分组形式 *(...)。 请注意，如果 '*' 被禁用，该功能也被禁用。

'+(' (KSH_GLOB)

分组形式 +(...).

'!(' (KSH_GLOB)

分组形式 !(...)。

'@(' (KSH_GLOB)

分组形式 @(...)。

disown [*job* ...]
job ... &|
job ... &!

从作业表中删除指定的 *job*；shell 将不再报告这些作业的状态，如果在这些作业运行或停止的情况下尝试退出交互式 shell，shell 也不会抱怨。 如果未指定 *job*，则删除当前作业。

如果 *job*s 当前已停止运行，且 AUTO_CONTINUE 选项未设置，系统将打印一条警告信息，说明如何在 *job* 被取消所有权后使其继续运行。 如果使用了后两种形式之一，*job*s 将自动运行，与 AUTO_CONTINUE 选项的设置无关。

echo [-neE] [*arg* ...]

将每个 *arg* 写入标准输出，每个参数之间用空格隔开。 如果没有 -n 标志，则在末尾打印换行。 echo 可以识别以下转义序列：

\a

响铃字符

\b

退格键

\c

抑制后续字符和最后的换行符

\e

转义

\f

分页符（form feed）

\n

换行符

\r

回车

\t

水平制表符

\v

垂直制表符

\\

反斜杠

\0*NNN*

八进制字符代码

\x*NN*

十六进制字符代码

`\u`*NNNN*

十六进制 unicode 字符代码

`\U`*NNNNNNNN*

十六进制 unicode 字符代码

`-E` 标志或 `BSD_ECHO` 选项可用于禁用这些转义序列。 在后一种情况下，可以使用 `-e` 标志来启用它们。

请注意，为了符合标准，双破折号不会终止选项处理，而是直接打印出来。 然而，单破折号会终止选项处理，因此第一个破折号（可能是选项后面的破折号）不会打印，但后面的所有内容都会作为参数打印。 单破折号行为与其他 shell 不同。 如需更加可移植的文本打印方式，请参阅 `printf`；如需在 zsh 中更可控的文本打印方式，请参阅 `print`。

`echotc`

参见 zsh/termcap 模块。

`echoti`

参见 zsh/terminfo 模块。

`emulate` [`-lLR`] [{`zsh`|`sh`|`ksh`|`csh`} [*flags* ...]]

不带任何参数打印当前模拟模式。

使用单参数设置 zsh 选项，以尽可能模拟指定的 shell。 *csh* 将不会被完全模拟。 如果参数不是上面列出的 shell，`zsh` 将作为默认值使用；更确切地说，对参数进行的测试与启动时根据 shell 名称确定模拟的测试相同，参见 兼容性 。 除了设置 shell 选项，该命令还能恢复模式启用的原始状态，就像使用 `enable -p` 启用所有模式一样。

如果 `emulate` 命令出现在使用 `functions -t` 标记为执行跟踪的函数内部，那么无论仿真模式或其他选项如何，`xtrace` 选项都将开启。 请注意，通过 `.`、`source` 或 `eval` 命令在函数内部执行的代码不被视为直接从函数运行，因此不会引发此行为。

如果使用 `-R` 开关，除了某些描述交互式环境的选项外，所有可设置的选项都将重置为与指定仿真模式相对应的默认值；否则，只有那些可能导致脚本和函数的可移植性问题的选项才会被修改。 如果给定了 `-L`开关，选项 `LOCAL_OPTIONS`、`LOCAL_PATTERNS` 和 `LOCAL_TRAPS` 也将被设置，从而导致 `emulate` 命令和任何 `setopt`、`disable -p` 或 `enable -p` 以及 `trap` 命令的效果与紧随其后的 shell

函数（如果有的话）本地相关； 通常，除了 ksh 之外，这些选项在所有模拟模式下都是关闭的。 -L 开关与 *flags* 中的 -c 相互排斥。

如果只有一个参数，且给出了 -l 开关，则会列出需要设置或取消设置的选项（后者用前缀 'no' 表示）。 -l 可以与 -L 或 -R 结合使用，列表会以适当方式修改。 请注意，该列表并不取决于当前的选项设置，也就是说，它包括所有原则上可能发生变化的选项，而不仅仅是那些实际会发生变化的选项。

flags 可以是 调用 中描述的任何调用时标志，但不能使用 '-o EMACS' 和 '-o VI'。 在某些情况下，'+r'/'+o RESTRICTED' 等标志可能被禁止使用。

如果 -c *arg* 出现在 *flags* 中，则 *arg* 将在所请求的仿真暂时生效时进行求值。 在这种情况下，在 emulate 返回之前，仿真模式和所有选项都会恢复到之前的值。 -R 开关可以放在要模拟的 shell 名称之前；注意这与 *flags* 中的 -R 意义不同。

使用 -c 可以为在求值表达式中定义的函数启用 'sticky' 仿真模式：仿真模式会随之与函数关联，因此在执行函数时，仿真模式（如果存在 -R 开关，则尊重 -R 开关）和所有选项都会在进入函数前设置（并清除模式禁用），并在退出后恢复状态。 如果调用该函数时粘性（ sticky）仿真已经生效（可以在 'emulate *shell* -c' 中调用，也可以在其他具有相同粘性模拟的函数中调用），则函数的进入和退出不会导致选项发生变化（标准处理（如 LOCAL_OPTIONS 选项）除外）。 这同样适用于在粘性仿真中标记为自动加载的函数；在加载函数和运行函数时，都会应用相应的选项集。

例如：

```
emulate sh -c 'fni() { setopt cshnullglob; }
fno() { fni; }'
fno
```

fni 和 fno 这两个函数是用粘性 sh 仿真定义的。 执行 fno后，与仿真相关的选项将被设置为 sh 中的值。 fno 然后调用 fni；由于 fni 也被标记为粘性 sh 仿真，因此在进入或退出 fno 时，选项不会发生变化。 因此，被 sh 模拟关闭的 cshnullglob 选项将在 fni 中开启，并在返回 fno 时保持不变。 从 fno 退出后，仿真模式和所有选项将恢复到进入临时仿真前的状态。

对于在合适的环境中执行为其他 shell 设计的代码这一预期目的来说，上述文件通常已经足够。 更详细的规则在后面。

1.

   由 'emulate *shell* -c' 提供的粘性仿真环境与被标记为粘性仿真的函数的入口所提供的环境完全相同。 因此，举例来说，在具有粘性仿真的函数中定义的子函数将继承粘性仿真。

2.

在进入或退出未标记为粘性仿真的函数时，除了正常情况下会发生的变化外，选项不会发生任何变化，即使这些函数是在粘性仿真中调用的。

3.

对于标记为 autoload 的函数或 zcompile 命令创建的字码中的函数，不提供特殊处理。

4.

emulate 的 -R 开关的存在与否对应不同的粘性仿真模式，因此，例如，'emulate sh -c'、'emulate -R sh -c' 和 'emulate csh -c' 被视为三种不同的粘性仿真。

5.

除了基本仿真外，提供的 shell 选项不同也意味着粘性仿真不同，例如，'emulate zsh -c' 和 'emulate zsh -o cbases -c' 被视为不同的粘性仿真。

enable [-afmprs] *name* ...

启用 *name*d 哈希表元素，可能是之前使用 disable 禁用了该元素。 默认情况下启用内置命令。 -a 选项会使 enable 作用于常规或全局别名。 -s 选项会使 enable 作用于后缀别名。 通过 -f 选项，enable 可以使用 shell 函数。 通过 -r 选项，enable 可以使用保留字。 在不带参数的情况下，会打印相应哈希表中所有已启用的哈希表元素。 使用 -m 标志时，参数被视为模式（应加引号），相应哈希表中与这些模式匹配的所有哈希表元素都会被启用。 可以使用 disable 内置命令禁用已启用的对象。

enable -p 可以重新启用 disable -p 禁用的模式。 请注意，它不会覆盖 globbing 选项；例如，除非同时设置了 EXTENDED_GLOB 选项，否则 'enable -p "~"' 不会使 ~ 模式字符生效。 要启用所有可能的模式（以便使用 disable -p 单独禁用它们），请使用 'setopt EXTENDED_GLOB KSH_GLOB NO_SH_GLOB'。

eval [*arg* ...]

将参数作为输入读入 shell，并在当前 shell 进程中执行产生的命令。 返回状态与 shell 直接执行命令的状态相同；如果没有 *args* 或不包含命令（即空字符串或空白），则返回状态为零。

exec [-cl] [-a *argv0*] [*command* [*arg* ...]]

用 *command* 替换当前 shell，而不是分叉。 如果 *command* 是 shell 内置命令或 shell 函数，则 shell 会执行该命令，并在命令完成后退出。

使用 `-c` 清除环境；使用 `-l` 在执行命令的 `argv[0]` 字符串前加上 `-`（模拟登录 shell）；使用 `-a` *argv0* 设置所执行命令的 `argv[0]` 字符串。 参见 前置命令修饰符。

如果设置了 `POSIX_BUILTINS` 选项，*command* 将永远不会被解释为 shell 内置命令或 shell 函数。 这意味着其他命令前修饰符，如 `builtin` 和 `noglob` 也不会在 shell 中被解释。 因此 *command* 总是通过搜索命令路径找到。

如果省略了 *command*，但指定了重定向，那么重定向将在当前 shell 中生效。

`exit` [*n*]

使用算术表达式 *n* 指定的退出状态退出 shell；如果未指定退出状态，则使用最后执行的命令的退出状态。 除非设置了 `IGNORE_EOF` 选项，否则 EOF 条件也会导致 shell 退出。

请参阅 作业和信号 末尾的注释，了解 `exit` 命令与作业之间可能存在的意外交互。

`export` [*name*[=*value*] ...]

指定的 *names* 将被标记为自动导出到后续执行命令的环境中。 等同于 `typeset -gx`。 如果指定的参数不存在，则会在全局范围内创建它。

`false` [*arg* ...]

不做任何事并返回退出状态1 。

`fc` [`-e` *ename*] [`-s`] [`-LI`] [`-m` *match*] [*old=new* ...] [*first* [*last*]]
`fc -l` [`-LI`] [`-nrdfEiD`] [`-t` *timefmt*] [`-m` *match*]
[*old=new* ...] [*first* [*last*]]
`fc -p` [`-a`] [*filename* [*histsize* [*savehistsize*]]]
`fc -P`
`fc -ARWI` [*filename*]

`fc` 命令控制交互式历史机制。 请注意，只有在 shell 是交互式的情况下，才会读写历史选项。 这通常是自动检测到的，但也可以通过在启动 shell 时设置 `interactive` 选项来强制执行。

该命令的前两种形式是从历史记录列表中选择 *first* 至 *last* 的一系列事件。 参数 *first* 和 *last* 可以指定为数字或字符串。 负数将用作当前历史事件编号的偏移量。 字符串则指定从给定字符串开始的最新事件。 所有替换 *old=new* 都将作用在事件文本中（如果有的话）。

可以通过以下标志进一步缩小用数字选择的事件范围。

`-I`

仅限于内部事件（非来自 $HISTFILE 的事件）

`-L`

只限制本地事件（不包括来自其他 shell 的事件，参见 选项说明 中的 SHARE_HISTORY – 注意 $HISTFILE 在启动时读取，会被视为本地事件）

`-m`

将第一个参数作为模式（应加引号），只有符合该模式的历史事件才会被考虑

如果未指定 *first*，则将设置为-1（最新事件），如果给定了 `-l` 标志，则将设置为-16。 如果未指定 *last*，则会将其设置为 *first*，如果给定了 `-l` 标志，则会将其设置为-1。 不过，如果当前事件已使用‘`print -s`’或‘`fc -R`’向历史记录添加了条目，那么 `-l` 的默认 *last* 将包含当前事件开始后所有新的历史记录条目。

如果给出 `-l` 标志，则会在标准输出中列出生成的事件。 否则，将在包含这些历史事件的文件上调用 `-e` *ename* 所指定的编辑程序。 如果未给出 `-e`，则使用参数 FCEDIT 的值；如果未设置该参数，则使用参数 EDITOR 的值；如果未设置该参数，则使用内置缺省值，通常是‘vi’。 如果 *ename* 为‘-’，则不调用编辑器。 编辑完成后，将执行已编辑的命令。

标志‘-s’等同于‘-e -’。 标志 `-r` 会颠倒事件的顺序，而标志 `-n` 则会在列出时抑制事件编号。

列出时也是，

`-d`

打印每个事件的时间戳

`-f`

以美国格式‘*MM*/*DD*/*YY* *hh*:*mm*’，打印完整的时间日期戳

`-E`

以欧洲格式‘*dd*.*mm*.*yyyy* *hh*:*mm*’，打印完整的时间日期戳

`-i`

以 ISO8601 格式‘*yyyy*-*mm*-*dd* *hh*:*mm*’打印完整的时间日期戳

`-t` *fmt*

打印指定格式的时间和日期戳；*fmt* 使用 strftime 函数格式化，并使用 提示符扩展 中针对 %D{*string*} 提示符格式描述的 zsh 扩展。 生成的格式化字符串不得超过 256 个字符，否则将无法打印

-D

打印经过时间；可与上述选项之一结合使用

‘fc -p’ 会将当前历史列表推入栈，并切换到新的历史列表。 如果同时指定了 -a 选项，当退出当前函数作用域时，历史列表将自动弹出，这比创建一个陷阱函数来手动调用 ‘fc -P’ 要好得多。 如果未指定参数，则历史列表为空，$HISTFILE 取消设置，$HISTSIZE 和 $SAVEHIST 设置为它们的默认值。 如果指定了一个参数，$HISTFILE 将设置为那个文件名，$HISTSIZE 和 $SAVEHIST 保持不变，并读入历史文件（如果存在）以初始化新列表。 如果指定了第二个参数，$HISTSIZE 和 $SAVEHIST将被设置为指定的数值。 最后，如果指定了第三个参数，$SAVEHIST 将被设置为与 $HISTSIZE 不同的值。 您可以随意更改新历史记录列表的这些环境值，以便对新历史记录列表进行操作。

‘fc -P’ 会将历史列表弹回到 ‘fc -p’ 保存的旧列表。 当前列表会在销毁前保存到 $HISTFILE 中（当然，前提是 $HISTFILE 和 $SAVEHIST 设置得当）。 $HISTFILE、$HISTSIZE 和 $SAVEHIST的值会恢复到调用 ‘fc -p’ 时的值。 需要注意的是，这种恢复可能与将这些变量设置为 "本地 "变量相冲突，因此最好避免在使用 ‘fc -p’ 的函数中对这些变量进行本地声明。 另一个有保障的安全组合是在函数顶层声明这些变量为局部变量，并与 ‘fc -p’ 一起使用自动选项 (-a)。 最后需要注意的是，如果需要在函数退出前手动弹出标记为自动弹出的推送，也是合法的。

‘fc -R’ 从指定文件读取历史记录，‘fc -W’ 将历史记录写入指定文件，‘fc -A’ 将历史记录追加到指定文件。 如果没有指定文件名，则假定使用 $HISTFILE。 如果在 -R 中添加 -I 选项，则只会添加内部历史列表中尚未包含的事件。 如果 -I 选项被添加到 -A 或 -W 中，则只会添加/写入上次增量添加/写入历史文件后新增的事件。 在任何情况下，创建的文件条目都不会超过 $SAVEHIST 。

fg [*job* …]
job …

依次将指定的 *job* 置于前台。 如果未指定 *job*，则恢复当前作业。

float [{+|-}Hghlprtux] [{+|-}EFLRZ [*n*]] [*name*[=*value*] …]

等同于 typeset -E，但不允许使用与浮点数无关的选项。

functions [{+|-}UkmtTuWz] [-x *num*] [*name* …]
functions -c *oldfn newfn*
functions -M [-s] *mathfn* [*min* [*max* [*shellfn*]]]
functions -M [-m *pattern* …]
functions +M [-m] *mathfn* …

等同于 typeset -f，但 -c、-x、-M 和 -W 选项除外。 关于 functions -u 和 functions -U，请参见 autoload，它提供了额外的选项。 关于 functions -t 和 functions -T，请参见 typeset -f。

-x 选项表示任何函数的输出都将以开头的制表符作为缩进，这些缩进由 shell 添加，用于显示语法结构，并扩展为指定数量 *num* 个空格。 *num* 也可以为 0，以抑制所有缩进。

-W 选项仅为指名的函数开启 WARN_NESTED_VAR 选项。 除非被调用函数也具有 -W 属性，否则该选项会在嵌套函数（无名函数除外）的起始位置关闭。

-c 选项会将 *oldfn* 复制到 *newfn* 中。 内部通过引用计数有效地处理复制。 如果 *oldfn* 被标记为自动加载，则会首先加载，如果加载失败，则复制失败。 随后可以重新定义其中一个函数，而不影响另一个函数。 一个典型的惯用手法是 *oldfn* 是一个库 shell 函数的名称，重新定义后调用 newfn，从而安装了该函数的修改版本。

The -M *and* +M *flags*

-M 选项不得与 typeset -f 处理的任何选项结合使用。

functions -M *mathfn* 将 *mathfn* 定义为一个数学函数的名称，该函数可用于所有形式的算术表达式；参见 算术求值。 默认情况下，*mathfn* 可以接受任意多个逗号分隔的参数。 如果给出 *min*，则必须有 *min* 个参数；如果同时给出 *min* 和 *max*，则必须至少有 *min* 个参数，最多有 *max* 个参数。 *max* 可以为 -1 表示没有上限。

默认情况下，函数由同名的 shell 函数实现；如果指定了 *shellfn*，则会给出相应 shell 函数的名称，而 *mathfn* 仍然是算术表达式中使用的名称。 如果选项 FUNCTION_ARGZERO 有效，则 $0 中的函数名称为 *mathfn*（而不是通常的 *shellfn*）。 shell 函数中的位置参数与数学函数调用的参数相对应。

shell 函数内部最后一个算术表达式的求值结果给出了数学函数的结果。这不仅限于形式为 $((...)) 的算术替换，还包括以任何其他方式计算的算术表达式，包括通过 let 内置命令，通过 ((...)) 语句，甚至通过 return 内置命令和数组下标来进行计算。因此，在计算函数结果之后执行算术计算的语法结构必须小心使用。例如

```
# WRONG
zmath_cube() {
  (( $1 * $1 * $1 ))
  return 0
}
functions -M cube 1 1 zmath_cube
print $(( cube(3) ))
```

由于使用了 return，因此将打印 '0'。

注释掉 return 会导致另一个问题：((...)) 语句将成为函数的最后一条语句，因此只要函数的 ***arithmetic result*** 恰好为零（数值上），函数的 ***return status*** ($?) 就会为非零（表示失败）：

```
# WRONG
zmath_cube() {
  (( $1 * $1 * $1 ))
}
functions -M cube 1 1 zmath_cube
print $(( cube(0) ))
```

相反，可以使用 true 内置函数：

```
# RIGHT
zmath_cube() {
  (( $1 * $1 * $1 ))
  true
}
functions -M cube 1 1 zmath_cube
print $(( cube(3) ))
```

如果在 functions -M 的基础上给定了附加选项 -s，那么函数的参数就是一个单独的字符串：开头括号和匹配的结尾括号之间的任何内容都将作为一个单独的参数传递给函数，即使其中包含逗号或空白。 因此，如果给定了最小参数和最大参数，则这两个参数必须为 1。 空参数列表将作为零长度字符串传递。 因此，下面的字符串函数接收单个参数（包括逗号）并打印 11：

```
stringfn() { (( $#1 )); true }
functions -Ms stringfn
print $(( stringfn(foo,bar,rod) ))
```

不带参数的 functions -M 会以与定义相同的形式列出所有此类用户自定义函数。如果使用附加选项 -m 和参数列表，则会列出 *mathfn* 与模式参数之一匹配的所有函数。

function +M 会移除数学函数列表；如果使用附加选项 -m，参数会被视为模式，所有 *mathfn* 与模式匹配的函数都会被移除。 需要注意的是，实现该行为的 shell 函数不会被移除（无论其名称是否与 *mathfn* 相同）。

getcap

参见 zsh/cap 模块.

getln [-AclneE] *name* ...

从缓冲堆栈读取顶部值，并将其放入 shell 参数 *name*。 等同于 read -zr。

getopts *optstring name* [*arg* ...]

检查 *arg*s 是否为合法选项。 如果省略了 *arg*，则使用位置参数。 有效的选项参数以 '+' 或 '-' 开头。 如果参数不是以 '+' 或 '-' 开头，或者参数 '--'，则选项结束。 注意，单个 '-' 不被视为有效的选项参数。 *optstring* 包含 getopts 可以识别的字母。 如果一个字母后面有 ':'，则该选项需要一个参数。 选项与参数之间可以用空格分隔。

每次调用时，getopts 都会将找到的选项字母放入 shell 参数 *name*，当 *arg* 以 '+' 开头时，则以 '+' 作为前缀。 下一个 *arg* 的索引存储在 OPTIND 中。 选项参数（如果有）存储在 OPTARG 中。

可以通过显式赋值 OPTIND 来改变第一个要检查的选项。 OPTIND 的初始值为 1，通常在进入 shell 函数时设置为 1，并在退出时恢复。 (POSIX_BUILTINS 选项禁用了这一功能，同时也改变了计算值的方式，以便与其他 shell 匹配）。 OPTARG 不会重置，并保留最近一次调用 getopts 时的值。 如果 OPTIND 或 OPTARG 中的任何一个被明确取消设置，它将保持未设置状态，索引或选项参数也不会被保存。 在这种情况下，选项本身仍存储在 *name* 中。

optstring 中的前导字母 ':' 会导致 getopts 将无效选项的字母保存在 OPTARG 中，并在出现未知选项时将 *name* 设置为'?'，在缺少必填参数时将 *name* 设置为 ':'。 否则，getopts 会将 *name* 设置为 '?'，并在选项无效时打印错误信息。 当没有更多选项时，退出状态为非零。

hash [-Ldfmrv] [*name*[=*value*]] ...

hash 可用于直接修改命令哈希表和命名目录哈希表的内容。 通常，我们会通过修改 PATH（命令哈希表）或创建适当的 shell 参数（命名目录哈希表）来修改这些表。 使用哪个哈希表由 -d 选项决定；不使用该选项时使用命令哈希表，使用该选项时使用命名目录哈希表。

无论是否明确使用 hash 命令，以 / 开头的命令 *name* 永远不会散列。 这样的命令总是可以在文件系统中直接查找到。

如果没有参数，也没有 -r 或 -f 选项，所选的哈希表将全部列出。

-r 选项会清空所选的哈希表。 随后将以正常方式重建。 -f 选项会立即完全重建所选的哈希表。 对于命令哈希表，这将对 PATH 中的所有绝对目录进行散列；对于命名目录哈希表，这将添加所有用户的主目录。 这两个选项不能与任何参数一起使用。

-m 选项会将参数视为模式（应加引号），并打印出与这些模式匹配的哈希表元素。 这是显示有限选择的哈希表元素的唯一方法。

对于每个带有相应 *value* 的 *name*，将 '*name*' 放入选定的哈希表，并将其与路径名 '*value*' 关联。 在命令哈希表中，这意味着只要将 '*name*' 作为命令参数，shell 就会

尝试执行 '*value*' 给出的文件。 在命名目录哈希表中，这意味着 '*value*' 可以被引用为 '~*name*'。

对于每个没有对应 *value* 的 *name* ，都会尝试将 *name* 添加到哈希表中，并以该哈希表的正常方式检查相应的 value 是什么。 如果找不到合适的 value，哈希表将保持不变。

如果使用 -v 选项，哈希表条目会以显式指定的方式添加到哈希表中。 如果与 -f 一起使用，则不会产生任何影响。

如果存在 -L 标志，那么每个哈希表条目都会以调用哈希的形式打印出来。

history

与 fc -l 相同。

integer [{+|-}Hghlprtux] [{+|-}LRZi [*n*]] [*name*[=*value*] ...]

等同于 typeset -i，但不允许使用与整数无关的选项。

jobs [-dlprs] [*job* ...]
jobs -Z *string*

列出每个给定作业的信息，如果省略 *job*，则列出所有作业的信息。 -l 标志列出进程 ID，-p 标志列出进程组。 如果指定了 -r 标志，则只会列出正在运行的作业；如果指定了 -s 标志，则只会显示已停止的作业。 如果给出 -d 标志，还将显示作业的启动目录（可能不是作业的当前目录）。

-Z 选项会用给定的字符串替换 shell 的参数和环境空间，必要时会截断。 这通常在 ps (ps(1)) 列表中可见。 守护进程通常会使用这一功能来显示其状态。

只有在顶层交互 shell 中才能对作业进行全面控制，而在管道左侧或 (...) 结构中运行的命令则无法对作业进行全面控制。 不过，此时作业状态的快照会被提取出来，因此仍然可以使用 jobs 内置命令或任何提供作业信息的参数。 这将提供子 shell 创建时的作业状态信息。 如果子 shell 中创建了后台进程，则会提供这些进程的相关信息。

例如,

```
sleep 10 &     # Job in background
(              # Shell forks
jobs           # Shows information about "sleep 10 &"
sleep 5 &      # Process in background (no job control)
jobs           # Shows information about "sleep 5 &"
)
```

kill [-s *signal_name* | -n *signal_number* | -*sig*] *job* ...
kill -l [*sig* ...]

向指定作业或进程发送 SIGTERM 或指定信号。 信号可以用数字或名称表示，带或不带前缀 'SIG'。 如果发送的信号不是'KILL' 或'CONT'，那么作业被停止时将发送'CONT' 信号。 参数 *job* 可以是作业列表中没有的作业的进程 ID。 在第二种形式 `kill -l` 中，如果未指定 *sig*，则会列出信号名称。 否则，将列出每个 *sig* 名称对应的信号编号。 每个 *sig* 如果是信号编号或代表被信号终止或停止的进程的退出状态的编号，则打印该信号的名称。

在某些系统中，少数信号可以使用其他信号名称。 典型的例子是 SIGCHLD 和 SIGCLD 或 SIGPOLL 和 SIGIO，假设它们对应的信号编号相同。 `kill -l` 只列出首选形式，但 `kill -l` *alt* 则会显示替代形式是否与信号编号对应。 例如，在 Linux 下 `kill -l IO` 和 `kill -l POLL` 都会输出 29，因此 `kill -IO` 和 `kill -POLL` 的效果相同。

许多系统允许进程 ID 为负数时杀死一个进程组，或为零时杀死当前进程组。

`let` *arg ...*

将每个 *arg* 作为算术表达式进行求值。 有关算术表达式的说明，请参阅 算术求值。 如果最后一个表达式的值不为零，则退出状态为 0；如果为零，则退出状态为 1；如果出现错误，则退出状态为 2。

`limit` [`-hs`] [*resource* [*limit*]] ...

设置或显示资源限制。 除非给出 `-s` 标志，否则限制仅适用于 shell 的子代。 如果给定 `-s` 而未给定其他参数，则当前 shell 的资源限制将设置为先前设置的子 shell 的资源限制。

如果未指定 *limit*，则打印 *resource* 的当前限制值，否则将限制值设置为指定值。 如果给出 `-h` 标志，则使用硬限制而不是软限制。 如果没有指定 *resource*，则打印所有限制。

在对多个资源进行循环时，如果 shell 检测到一个错误的参数，它会立即终止。 但是，如果由于其他原因无法设置限制，它将继续尝试设置剩余的限制。

resource 可以是下面之一:

`addressspace`

已使用的最大地址空间。

`aiomemorylocked`

RAM 中为 AIO 操作锁定的最大内存量。

`aiooperations`

AIO 操作的最大次数。

cachedthreads

缓存线程的最大数量。

coredumpsize

核心转储的最大容量。

cputime

每个进程的最长 CPU 秒数。

datasize

每个进程的最大数据量（包括堆栈）。

descriptors

文件描述符的最大值。

filesize

允许的最大单个文件。

kqueues

已分配 kqueues 的最大数量。

maxproc

进程的最大数量。

maxpthreads

每个进程的最大线程数。

memorylocked

内存中锁定的最大内存量。

memoryuse

最大驻留集大小。

msgqueue

POSIX 报文队列的最大字节数。

posixlocks

每个用户的最大 POSIX 锁数。

`pseudoterminals`

伪终端的最大数量。

`resident`

最大驻留集大小。

`sigpending`

待处理信号的最大数量。

`sockbufsize`

所有套接字缓冲区的最大容量。

`stacksize`

每个进程的最大堆栈大小。

`swapsize`

使用的最大交换区量。

`vmemorysize`

虚拟内存的最大容量。

哪些资源限制可用取决于系统。 *resource* 可以缩写为任何明确的前缀。 它也可以是一个整数，对应于操作系统为资源定义的整数。

如果参数对应的数字超出了 shell 配置的资源范围，shell 会尝试读取或写入限制值，如果失败则会报错。 由于 shell 内部不存储此类资源，因此除非有 `-s` 选项，否则设置限制的尝试将失败。

limit 是一个数字，可选的缩放因子如下：

*n*h

小时

*n*k

千字节 (默认)

*n*m

兆字节或分钟

*n*g

千兆字节

[*mm*:]*ss*

分和秒

当 shell 以模拟其他 shell 的模式启动时，`limit` 命令默认不可用。 可以使用 ‘`zmodload -F zsh/rlimits b:limit`’ 命令来使用它。

`local` [{+|-}`AHUahlprtux`] [{+|-}`EFLRZi` [*n*]] [*name*[=*value*] ...]

与 `typeset` 相同，但不允许使用 `-g` 和 `-f`。 在这种情况下，`-x` 选项并不强制使用 `-g`，即导出变量将是函数的局部变量。

`logout` [*n*]

与 `exit`相同，但它只在登录 shell 中起作用。

`noglob` *simple command*

请参阅 前置命令修饰符

`popd` [`-q`] [{+|-}*n*]

从目录栈中移除一个条目，并执行 `cd` 到新的顶部目录。 如果没有参数，则删除当前的顶部目录。 形式为 ‘+*n*’ 的参数从 `dirs` 命令显示的列表左侧开始计数，从 0 开始识别堆栈条目。 形式为 `-`*n* 的参数从右边开始计数。 如果设置了 `PUSHD_MINUS` 选项，‘+’ 和 ‘-’ 在此处的含义将互换。

如果指定了 `-q`（quiet）选项，则不会调用钩子函数 `chpwd` 和数组 `$chpwd_functions` 中的函数，也不会打印新的目录栈。 这对于调用 `popd` 不会改变交互式用户所看到的环境非常有用。

`print` [`-abcDilmnNoOpPrsSz`] [`-u` *n*] [`-f` *format*] [`-C` *cols*]
[`-v` *name*] [`-xX` *tabstop*] [`-R` [`-en`]] [*arg* ...]

如果使用 ‘`-f`’ 选项，参数将按照 `printf` 的方式打印。 如果没有使用标志或使用标志 ‘`-`’，参数将按照 `echo` 的方式打印在标准输出上，但有以下不同： 转义序列 ‘`\M-`*x*’（或‘`\M`*x*’）会元化字符 *x*（设置最高位），‘`\C-`*x*’（或 ‘`\C`*x*’）会产生一个控制字符（‘`\C-@`’ 和 ‘`\C-?`’ 表示 NULL 和 delete 字符），八进制的字符代码用 ‘`\`*NNN*’ 表示（而不是 ‘`\0`*NNN*’），而 ‘`\E`’ 是 ‘`\e`’ 的同义词。 最后，如果不在转义序列中，‘`\`’ 会转义后面的字符，并不会被打印出来。

`-a`

打印参数，列先递增（with the column incrementing first）。 仅在使用 `-c` 和 `-C` 选项时有用。

`-b`

识别为 bindkey 命令定义的所有转义序列，参见 Zle 内置命令。

`-c`

按列打印参数。 除非同时给出 `-a`，否则参数将按行先递增的方式打印。

`-C` *cols*

打印 *cols* 列中的参数。 除非同时给出 `-a`，否则参数将按行先递增的方式打印。

`-D`

将参数视为路径，酌情用与目录名相对应的 ~ 表达式替换目录前缀。

`-i`

如果同时给出 `-o` 或 `-O`，则排序与大小写无关。

`-l`

打印参数，以换行符而不是空格分隔。 注意：如果参数列表为空，`print -l` 仍会输出一行空行。若要每行打印一个可能为空的参数列表，请使用 `print -C1`，如‘`print -rC1 -- "$list[@]"`’。

`-m`

将第一个参数作为一个模式（应加引号），并将其从参数列表中移除，同时移除与该模式不匹配的后续参数。

`-n`

输出时不要加换行符。

`-N`

打印参数，以空字符分隔和结束。同样，`print -rNC1 -- "$list[@]"` 是将任意列表打印为空字符分隔的记录的典型方法。

`-o`

按升序打印参数。

`-O`

按降序打印参数。

`-p`

将参数打印到协进程的输入端。

`-P`

执行提示符扩展（参见 提示符扩展）。 与 '`-f`' 结合使用时，提示符转义序列只在插值参数中解析，而不在格式字符串中解析。

`-r`

忽略 `echo` 的转义规则。

`-R`

模拟 BSD `echo` 命令，除非给出 `-e` 标志，否则该命令不处理转义序列。 `-n` 标志会抑制尾部换行。 只有 `-e` 和 `-n` 标志会在 `-R` 之后被识别；所有其他参数和选项都会被打印。

`-s`

将结果放在历史记录列表中，而不是标准输出中。 在历史记录中，`print` 命令的每个参数都被视为一个单词，无论其内容如何。

`-S`

将结果放在历史记录列表中，而不是标准输出中。 在这种情况下，只允许使用单个参数；参数将被分割成单词，就像完整的 shell 命令行一样。 其效果与使用启用 `HIST_LEX_WORDS` 选项时从历史文件读取行类似。

`-u` *n*

打印文件描述符 *n* 的参数。

`-v` *name*

将打印的参数存储为参数 *name* 的值。

`-x` *tab-stop*

在打印字符串的每一行输出中展开前导制表符，假设每 *tab-stop* 个字符有一个制表符停止符。 这适用于格式化可能使用制表符缩进的代码。 请注意，即使 `print` 使用空格分隔参数，也会展开要打印的任何参数的前导制表符，而不仅仅是第一个参数的前导制表符（列数在各参数间保持不变，但由于之前的制表符未展开，输出时可能会不正确）。

每条打印命令的输出起始位置假定与制表符停止位置对齐。 如果选项 `MULTIBYTE` 有效，多字节字符的宽度将被处理。 如果使用了其他格式化选项，即列对齐或 `printf` 样式，或者输出到 shell 历史记录或命令行编辑器等特殊位置，则该选项将被忽略。

`-X` *tab-stop*

这与 -x 类似，但打印字符串中的所有制表符都会展开。 如果参数中的制表符被用于生成表格格式，则可以使用这种方法。

-z

将参数推入编辑缓冲堆栈，中间用空格隔开。

如果‘-m’, ‘-o’ 或 ‘-O’ 中的任何一个与 ‘-f’ 结合使用，且没有参数（ ‘-m’ 的情况下是在删除参数后），则不会打印任何内容。

printf [-v *name*] *format* [*arg* ...]

根据格式规范打印参数。格式规则与 C 语言中的相同。格式中可识别与 echo 相同的转义序列。所有以 csdiouxeEfgGn 结尾的 C 转换规范都会被处理。除此之外，还可以使用 ‘%b’ 代替 ‘%s’，以识别参数中的转义序列，并使用 ‘%q’ 引用参数，使其可以作为 shell 输入重复使用。对于数字格式指定符，如果相应参数以引号字符开头，则使用后面字符的数值作为要打印的数字；否则参数将作为算术表达式求值。有关算术表达式的说明，请参阅 算术求值。使用 ‘%n’ 时，相应参数将作为标识符，并作为整数参数创建。

通常，转换规范会按顺序应用于每个参数，但也可以通过将 ‘%’ 替换为 ‘%*n*$’ ，将 ‘*’ 替换为 ‘**n*$’，从而明确指定使用第 *n* 个参数。 建议不要将这种明确样式的引用与正常样式的引用混用，而且对这种混用样式的处理可能会在将来发生变化。

如果参数在格式化后仍未使用，格式化字符串会被重复使用，直到所有参数都被用完。使用 print 内置命令时，可以通过 -r 选项来抑制这种情况。如果格式化所需的参数多于指定的参数，则行为与指定零参数或空字符串相同。

-v 选项会将输出存储为参数 *name* 的值，而不是打印出来。如果 *name* 是一个数组，且格式字符串在使用参数时被重复使用，那么每次使用格式字符串时都会使用一个数组元素。

pushd [-qsLP] [*arg*]
pushd [-qsLP] *old new*
pushd [-qsLP] {+|-}*n*

更改当前目录，并将旧的当前目录推入目录栈。 在第一种形式中，将当前目录更改为 *arg*。 如果未指定 *arg*，则将当前目录更改为目录栈中的第二个目录（即交换前两个条目）；如果设置了 PUSHD_TO_HOME选项，或者目录栈中只有一个条目，则将当前目录更改为 $HOME。 否则，*arg* 将按照 cd 的方式解释。 第二种形式中 *old* 和 *new* 的含义也与 cd 相同。

pushd 的第三种形式是通过旋转目录列表来更改目录。 形式为 ‘+*n*’ 的参数从 dirs 命令显示的目录列表的左边开始计数，从 0 开始，以此确定堆栈条目。 形式为 ‘-*n*’的参数从右边开始计数。 如果设置了 PUSHD_MINUS 选项， ‘+’ 和 ‘-’ 在此处的含义将互换。

如果指定了 -q（quiet）选项，则不会调用钩子函数 chpwd 和数组 $chpwd_functions 中的函数，也不会打印新的目录栈。 这对于调用 pushd 不会改变交互式用户所看到的环境非常有用。

如果未指定选项 -q，也未设置 shell 选项 PUSHD_SILENT，则将在执行 pushd 之后打印目录堆栈。

选项 -s、-L 和 -P 的含义与 cd 内置命令相同。

pushln [*arg* ...]

等价于 print -nz。

pwd [-rLP]

打印当前工作目录的绝对路径名。 如果指定了 -r 或 -P 标志，或设置了 CHASE_LINKS 选项且未给出 -L 标志，则打印的路径将不包含符号链接。

r

与 fc -e - 一样。

read [-rszpqAclneE] [-t [*num*]] [-k [*num*]] [-d *delim*] [-u *n*] [[*name*][?*prompt*]] [*name* ...]

读取一行，并使用 $IFS 中的字符作为分隔符将其分成多个字段,下文所述情况除外。 第一个字段分配给第一个 *name*，第二个字段分配给第二个 *name*，以此类推，剩余字段分配给最后一个 *name*。 如果省略 *name*， REPLY 用于标量， reply 用于数组。

-r

原始模式：行末的 '\' 不表示续行，行中的反斜线不会引述后面的字符，也不会被删除。

-s

如果从终端读取，则不回显字符。

-q

只从终端读取一个字符，如果该字符为 'y' 或 'Y' ，则将 *name* 设置为 'y'，否则设置为 'n'。 设置该标志后，只有当字符为 'y' 或 'Y' 时，返回状态才为零。 该选项可与超时（参见 -t）一起使用；如果读取超时或遇到文件结束，则返回状态 2。 除非有 -u 或 -p，否则将从终端读取输入。 该选项也可在 zle 小部件中使用。

-k [*num*]

只读取一个（或 *num* 个）字符。 所有字符都分配给第一个 *name*，不分词。 当 -q 存在时，该标志将被忽略。 除非有 -u 或 -p，否则将从终端读取输入。 该选项也可在 zle 小部件中使用。

请注意，尽管使用了 'key' 这个助记符，但该选项确实读取全字符，如果设置了 MULTIBYTE 选项，全字符可能由多个字节组成。

-z

从编辑器缓冲堆栈中读取一个条目并将其赋值给第一个 *name*，不进行分词。 文本将通过 'print -z' 或行编辑器（参见 Zsh 行编辑器）中的 push-line 推入堆栈。 当 -k 或 -q 标志存在时，该标志将被忽略。

-e
-E

读取的输入将打印（回显）到标准输出。 如果使用 -e 标志，则不会将输入分配给参数。

-A

第一个 *name* 将作为数组的名称，所有单词都将赋值给它。

-c
-l

只有在用于补全的函数（使用 compctl 的 -K 标志指定 ）中调用时，才允许使用这些标志。 如果给出 -c 标志，将读取当前命令的字词。如果给出 -l 标志，则将整行作为标量分配。 如果两个标志都存在，则使用 -l，忽略 -c。

-n

与 -c 一起使用时，将读取光标所在单词的编号。 使用 -l，将读取光标所在字符的索引。 请注意，命令名称是字 1，而不是字 0，当光标位于行尾时，其字符索引是行长加 1。

-u *n*

输入内容从文件描述符 *n* 读取。

-p

从协进程读取输入。

-d *delim*

输入以 *delim* 的第一个字符结束，而不是换行。

-t [*num*]

在尝试读取数据前测试输入是否可用。 如果存在 *num*，它必须以数字开头，并将被计算为一个秒数（可以是浮点数）；在这种情况下，如果在这段时间内输入不可用，读取将超时。 如果 *num* 不存在，它将被视为零，因此 read 会在没有输入时立即返回。 如果没有输入，则返回状态 1，不设置任何变量。

当使用 -z 从编辑器缓冲区读取数据时，当使用 -c 或 -l 从补全中调用时，当使用 -q 在读取数据前清空输入队列时，或者在使用其他机制测试输入的 zle 中，该选项不可用。

请注意，read 并不试图改变输入处理模式。 默认模式是规范输入，即一次读取整行内容，因此通常在输入整行内容之前，'read -t' 不会读取任何内容。不过，当使用 -k 从终端读取时，输入是一次处理一个键的；在这种情况下，只测试第一个字符是否可用，因此，例如'read -t -k 2' 仍然可以阻塞第二个字符。 如果不希望这样，请使用两个 'read -t -k' 实例。

如果第一个参数包含 '?'，则在 shell 处于交互状态时，该单词的剩余部分将作为 *prompt* 出现在标准错误中。

当遇到文件结束、出现 -c 或 -l 且命令不是由 compctl 函数调用时，或如 -q 所述，read的值（退出状态）为 1。 否则，值为 0。

-k、-p、-q、-u 和 -z 标志的某些组合，其行为未定义。 目前 -q 取消了所有其他标志，-p 取消了 -u，-k 取消了 -z，而 -z 则同时取消了 -p 和 -u。

-c 或 -l 标志会取消所有 -kpquz 标志。

readonly

与 typeset -r 相同。 设置 POSIX_BUILTINS 选项后，与 typeset -gr 相同。

rehash

与 hash -r 一样。

return [*n*]

使 shell 函数或 '.' 脚本以算术表达式 *n* 指定的返回状态返回调用脚本。 例如，下面的代码会打印 '42'：

```
() { integer foo=40; return "foo + 2" }
echo $?
```

如果省略 *n*，则返回最后执行命令的状态。

如果 return 是通过 TRAP*NAL* 函数中的陷阱执行的，那么返回状态为零和非零的效果是不同的。 状态为零（或陷阱结束后隐式返回）时，shell 将返回到之前正在处理的内容；状态为非零时，除了保留陷阱的返回状态外，shell 的行为与中断时一

样。 需要注意的是，引起陷阱的信号的数值会作为第一个参数传递，因此语句 ‘return "128+$1"’ 返回的状态与信号未被捕获时的状态相同。

sched

参见 zsh/sched 模块.

set [{+|-}*options* | {+|-}o [*option_name*]] ... [{+|-}A [*name*]] [*arg* ...]

设置 shell 的选项和/或设置位置参数，或声明和设置数组。 如果给定 -s 选项，则会在将指定参数赋值给位置参数（如果使用 -A，则赋值给数组 *name*）之前对其进行排序。 使用 +s 会按降序对参数排序。 其他标志的含义请参见 选项。 可以使用 -o 选项指定标志的名称。如果没有用 -o 提供选项名称，则会打印当前的选项状态：有关格式的更多信息，请参阅下文对 setopt 的描述。 如果使用 +o，则以可作为 shell 输入的形式打印。

如果指定了 -A 标志，*name* 将被设置为包含给定 *arg* 的数组；如果没有指定 *name*，所有数组将连同其值一起打印。

如果使用 +A，且 *name* 是数组，则给定参数将替换该数组的初始元素；如果未指定 *name*，则打印所有数组，但不包含其值。

-A *name* 或 +A *name* 之后的参数的行为取决于是否设置了 KSH_ARRAYS 选项。 如果未设置，则 *name* 后面的所有参数都会被视为数组的值，无论其形式如何。 如果设置了该选项，则会继续进行正常的选项处理；只有常规参数才会被视为数组的值。 这意味着

```
set -A array -x -- foo
```

如果 KSH_ARRAYS 未设置，则会将 array 设置为 ‘-x -- foo’；如果设置了 KSH_ARRAYS,则会将数组设置为 foo，并开启 ‘-x’ 选项。

如果没有 -A 标志，但在选项之外还有参数，则会设置位置参数。 如果选项列表（如果有）以 ‘--’ 结束，且没有其他参数，位置参数将被取消设置。

如果没有参数，也没有给出 ‘--’，那么所有参数的名称和值都会打印在标准输出上。 如果唯一的参数是 ‘+’，则会打印所有参数的名称。

由于历史原因，在任何其他模拟模式下，而非 zsh 的原生模式下，‘set -’ 被视为 ‘set +xv’，而 ‘set - *args*’ 被视为 ‘set +xv -- *args*’。

setcap

参见 zsh/cap 模块.

setopt [{+|-}*options* | {+|-}o *option_name*] [-m] [*name* ...]

设置 shell 的选项。 所有用标志或名称指定的选项都会被设置。

如果没有提供参数，则会打印当前设置的所有选项名称。 选择这种形式是为了尽量减少与当前模拟器（emulation）的默认选项之间的差异（默认模拟器为原生 zsh，在 选项说明 中显示为 <Z>）。 只有在关闭时，才会以 no 为前缀显示默认开启的模拟选项，而只有在开启时，才会以不带 no 前缀的方式显示其他选项。 除了用户从默认状态修改的选项外，任何由 shell 自动激活的选项（例如 SHIN_STDIN 或 INTERACTIVE）都会显示在列表中。 选项 KSH_OPTION_PRINT 会进一步修改格式，但在这种情况下，选择带或不带 no 前缀的选项的原理是一样的。

如果给定 -m 标志，参数将作为模式（应加引号以防止文件名扩展），所有名称与这些模式匹配的选项都将被设置。

需要注意的是，错误的选项名称不会导致后续 shell 代码的中止执行；这一行为与 'set -o' 不同。 这是因为 POSIX 标准将 set 视为特殊的内置命令，而 setopt 并非如此。

shift [-p] [*n*] [*name* ...]

位置参数 ${*n*+1} ... 更名为 $1 ...，其中 *n* 是默认为 1 的算术表达式。如果指定了 *name*，则会移位具有这些名称的数组，而不是位置参数。

如果给定了选项 -p，参数将从数组的末尾而不是起始位置移除（弹出）。

source *file* [*arg* ...]

与 '.' 相同，不同之处在于总是先搜索当前目录，然后再搜索 $path 中的目录。

stat

参见 zsh/stat 模块。

suspend [-f]

暂停 shell 的执行（向其发送 SIGTSTP），直到收到 SIGCONT。 除非给出 -f 选项，否则将拒绝暂停登录 shell。

test [*arg* ...]
[[*arg* ...]]

类似于 test 的系统版本。 为兼容而添加；请使用条件表达式代替（参见 条件表达式）。 条件表达式语法与 test 和 [内置命令的主要区别在于：这些命令不按语法处理，因此如空变量扩展可能会导致省略参数；语法错误会导致返回状态 2，而不是 shell 错误；算术运算符希望使用整数参数，而不是算术表达式。

该命令试图实现 POSIX 及其扩展（在指定的地方）。 遗憾的是，语法本身存在歧义，特别是测试运算符和类似运算符的字符串之间没有区别。 对于少量参数（最多

四个），该标准试图解决这些问题；对于五个或更多参数，则无法保证兼容性。 建议用户尽可能使用没有这些歧义的 '[[' 测试语法。

`times`

打印 shell 和从 shell 运行的进程的累计用户和系统时间。

`trap` [*arg*] [*sig* ...]

arg 是一系列命令（通常加引号以防止 shell 立即执行），当 shell 接收到一个或多个 *sig* 参数指定的信号时，将读取并执行这些命令。 每个 *sig* 都可以是一个数字，也可以是一个信号的名称，可以在前面加上或不加上 `SIG` 字符串（例如，1、HUP 和 SIGHUP 都是同一个信号）。

如果 *arg* 为 '-'，则重置指定的信号为默认值；如果没有 *sig* 参数，则重置所有陷阱。

如果 *arg* 为空字符串，则 shell（及其调用的命令）将忽略指定的信号。

如果省略了 *arg*，但提供了一个或多个 *sig* 参数（即第一个参数是有效的信号编号或名称），其效果与 *arg* 被指定为 '-' 相同。

无参数的 `trap` 命令将打印与每个信号相关的命令列表。

如果 *sig* 是 `ZERR`，那么 *arg* 将在每条退出状态为非零的命令后执行。 在没有 `SIGERR` 信号的系统中，`ERR` 是 `ZERR` 的别名（通常是这种情况）。

如果 *sig* 是 `DEBUG`，那么如果设置了 `DEBUG_BEFORE_CMD`（默认情况），*arg* 将在每条命令之前执行，否则将在每条命令之后执行。 这里的 '命令' 就是 shell 语法中的 '子列表(sublist)'，参见 简单命令和管道。 如果设置了 `DEBUG_BEFORE_CMD`，就可以使用多种附加功能。 首先，通过设置选项 `ERR_EXIT`，可以跳过下一条命令；请参阅 选项说明 中关于 `ERR_EXIT` 选项的描述。 此外，shell 参数 `ZSH_DEBUG_CMD` 也会被设置为与陷阱后要执行的命令相对应的字符串。 请注意，该字符串是根据内部格式重建的，格式可能与原文不同。 陷阱执行后，该参数将被取消。

如果 *sig* 为 `0` 或 `EXIT`，且 `trap` 语句在函数体内部执行，则在函数完成后执行命令 *arg*。 开始执行时 `$?` 的值是 shell 的退出状态或函数退出时的返回状态。 如果 *sig* 为 `0` 或 `EXIT`，且 `trap` 语句未在函数体内部执行，则在 shell 终止时执行命令 *arg*；陷阱在任何 `zshexit` 挂钩函数之前运行。

`ZERR`、`DEBUG` 和 `EXIT` 陷阱不会在其他陷阱内执行。 `ZERR` 和 `DEBUG` 陷阱会保留在子 shell 内，而其他陷阱会被重置。

请注意，使用 `trap` 内置命令定义的陷阱与 'TRAP*NAL* () { ... }' 定义的陷阱略有不同，因为后者有自己的函数环境（行号、局部变量等），而前者使用的是调用陷阱的命令环境。 例如

```
trap 'print $LINENO' DEBUG
```

会在命令运行后打印该命令的行号，而

```
TRAPDEBUG() { print $LINENO; }
```

将始终打印数字 0。

如上文 `kill` 所述，允许使用可替换信号名称。 使用其中任何一个名称定义陷阱，都会导致使用其他可替换名称的陷阱被删除。 不过，为了保持一致性，我们建议用户只使用一种名称。

`true` [*arg* ...]

不做任何事并返回 0 退出状态。

`ttyctl` [`-fu`]

`-f` 选项会冻结 tty（即终端或终端模拟器），而 `-u` 则会解除冻结。 当 tty 被冻结时，除了屏幕大小的改变外，shell 不会执行外部程序对 tty 设置所做的任何更改；每当命令退出或暂停时，shell 会简单地将设置重置为之前的值。 因此，当 tty 被冻结时，`stty` 和类似程序将不起作用。 冻结 tty 并不会导致当前状态被记忆：相反，它会导致状态的未来更改被阻止。

在没有选项的情况下，它会报告终端是否被冻结。

需要注意的是，无论 tty 是否冻结，shell 都需要在行编辑器启动时更改设置，因此解冻 tty 并不能保证命令行上的设置得到保留。 在编辑命令行之间运行的命令字符串将保持一致的 tty 状态。 另请参阅 shell 变量 `STTY`，了解在运行外部命令前初始化 tty 和/或在单个命令周围冻结 tty 的方法。

`type` [`-wfpamsS`] *name* ...

等价于 `whence -v`。

```
typeset [ {+|-}AHUaghlmrtux ] [ {+|-}EFLRZip [ n ] ]
        [ + ] [ name[=value] ... ]
typeset -T [ {+|-}Uglrux ] [ {+|-}LRZp [ n ] ]
        [ + | SCALAR[=value] array[=(value ...)] [ sep ] ]
typeset -f [ {+|-}TUkmtuz ] [ + ] [ name ... ]
```

设置或显示 shell 参数的属性和数值。

除了下面提到的会改变行为的控制标志外，每一个 *name* 都会创建一个参数，但前提是该参数尚未被引用。 在函数内部时，会为每个 *name*（即使是已经存在的参数）创建一个新参数，并在函数完成后取消设置。 请参阅 局部参数。 同样的规则也适用于特殊 shell 参数，这些参数在局部化后仍保留其特殊属性。

对于每个 *name*=*value* 赋值，参数 *name* 将被设置为 *value*。 如果省略了赋值，并且 *name* **没有** 指向了一个现有参数，那么一个新参数将初始化为空字符串、零或空数组（视情况而定），**除非** shell 选项 TYPESET_TO_UNSET 已被设置。 设置该选项后，参数属性将被记录，但参数仍保持未设置状态。

如果未设置 shell 选项 TYPESET_SILENT，那么对于每一个指向已设置参数的剩余 *name*，都会以赋值的形式打印参数的名称和值。 对于新创建的参数，或 *name* 同时给出以下属性标志时，将不打印任何内容。 使用 '+' 而不是减号来引入属性时，将关闭该属性。

如果没有 *name*，则打印所有参数的名称和值。 在这种情况下，属性标志限制为只显示具有指定属性的参数，使用 '+' 而不是 '-' 引入标志可以在没有参数名时抑制参数值的打印。

该命令的所有形式都能处理标量赋值。 如果在解析命令行时（注意：不是在执行时）匹配了保留字 declare、export、float、integer、local、readonly 或 typeset，则可以进行数组赋值。 在这种情况下，参数将作为赋值进行解析，但不支持 '+=' 语法和 GLOB_ASSIGN 选项，并且 = 之后的标量值**不会**被分割为单词，即使已展开（与 KSH_TYPESET 选项的设置无关；该选项已过时）。

命令和保留字解析差异示例：

```
# Reserved word parsing
typeset svar=$(echo one word) avar=(several words)
```

上面的代码创建了一个标量参数 svar 和一个数组参数 avar，就好像赋值是

```
svar="one word"
avar=(several words)
```

另一方面：

```
# Normal builtin interface
builtin typeset svar=$(echo two words)
```

使用 builtin 关键字后，上述命令将使用 typeset 的标准内置接口，其中参数解析的方式与其他命令相同。 本示例创建了一个包含值 two 的标量 svar，以及另一个无值的标量参数 words。 在这种情况下，数组值要么会导致错误，要么会被视为一组模糊的 glob 限定符。

如果在命令行扩展后采用赋值形式，则允许使用任意参数；不过，这些参数只能执行标量赋值：

```
var='svar=val'
typeset $var
```

上述代码将标量参数 svar 设置为值 val。 在 var 中的值周围使用括号不会导致数组赋值，因为在替换 $var 时，括号将被视为普通字符。 赋值中名称部分的任何非平凡 (non-trivial) 式扩展都会导致以这种方式处理参数：

```
typeset {var1,var2,var3}=name
```

上述语法是有效的，并具有将三个参数设置为相同值的预期效果，但扩展后的命令行会被解析为 typeset 的三个普通命令行参数的集合。 因此，无法通过这种方法为多个数组赋值。

请注意，任何命令的每个接口都可以单独禁用。 例如，'disable -r typeset' 禁用 typeset 的保留字接口，暴露内置接口，而 'disable typeset' 则禁用内置接口。 需要注意的是，禁用 typeset 的保留字接口可能会导致 'typeset -p' 的输出出现问题，因为 'typeset -p' 假定保留字接口可用，以便还原数组和关联数组的值。

与参数赋值语句不同，typeset 在涉及命令替换的赋值中的退出状态并不反映命令替换的退出状态。 因此，要测试命令替换中的错误，应将参数的声明与其初始化分开：

```
# WRONG
typeset var1=$(exit 1) || echo "Trouble with var1"

# RIGHT
typeset var1 && var1=$(exit 1) || echo "Trouble with var1"
```

要将参数 *param* 初始化为命令输出并标记为只读，请在参数赋值语句后使用 `typeset -r` *param* 或 `readonly` *param*。

如果没有给出属性标志，也没有 *name* 参数或使用了 +m 标志，则在打印的每个参数名称前都会列出该参数的属性 (array、association、exported、float、integer、readonly 或 undefined (用于尚未加载的自动加载参数))。 如果 +m 与属性标志一起使用，且所有这些标志都用 + 引入，则会打印匹配的参数名，但不会打印其值。

以下控制标志会改变 typeset 的行为：

+

> 如果 '+' 作为最后一个选项单独出现，那么将打印所有参数 (带 -f 的函数) 的名称，但不打印参数值 (函数体)。 不允许出现 *name* 参数，而且在 '+' 之后出现其他选项都是错误的。'+' 的作用就好像在它前面的所有属性标志都加上了 '+' 前缀。 例如，'typeset -U +' 等同于 'typeset +U'，显示所有具有唯一性属性的数组的名称，而 'typeset -f -U +' 则显示所有可自动加载函数的名称。 如果 + 是唯一的选项，那么每个参数的类型信息 (数组、只读等) 也会被打印出来，打印方式与 'typeset +m "*"' 相同。

-g

-g（全局）表示产生的参数将不限于本地作用域。 请注意，这并不一定意味着参数将是全局的，因为该标志将适用于来自封闭函数的任何现有参数（即使未设置）。 该标志不会影响创建后的参数，因此在列出现有参数时没有任何作用，除了与 -m 结合使用外，标志 +g 也没有任何作用（见下文）。

-m

如果给出 -m 标志，*name* 参数将被视为模式（使用引号防止这些参数被解释为文件模式）。 在没有属性标志的情况下，所有名称匹配的参数（或带有 -f 标志的函数）都会被打印（此时不使用 shell 选项 TYPESET_SILENT）。

如果 +g 标志与 -m 结合使用，则会为每一个尚未本地化的匹配参数创建一个新的本地参数。 否则，-m 会将所有其他标志或赋值应用于现有参数。

除了使用 *name*=*value* 进行赋值外，使用 +m 会强制打印匹配的参数及其属性，即使在函数内部也是如此。 请注意，如果没有给出模式，-m 将被忽略，因此 'typeset -m' 会显示属性，但 'typeset -a +m' 不会。

-p [*n*]

如果给出 -p 选项，参数和值将以带赋值的排版命令形式打印，与其他标志和选项无关。 需要注意的是，会尊守参数上的 -H 标志；这些参数不会显示任何值。

-p 后面可以跟一个可选的整数参数。 目前只支持 1 值。 在这种情况下，为了便于阅读，数组和关联数组会在缩进元素之间使用换行符打印。

-T [*scalar*[=*value*] *array*[=(*value* ...)] [*sep*]]

当与 -f 搭配使用时，该标志的含义有所不同；见下文。 在其他情况下，-T 选项需要 0、2 或 3 个参数。 如果没有参数，将显示以这种方式创建的参数列表。 如果有两个或三个参数，前两个参数是一个标量和一个数组参数的名称（按此顺序），这两个参数将以 $PATH 和 $path 的方式绑定在一起。 第三个可选参数是一个单字符分隔符，用于连接数组元素以形成标量；如果没有分隔符，则使用冒号，与 $PATH 一样。 只有分隔符的第一个字符才有意义，其余字符将被忽略。 目前还不支持多字节字符。

标量参数和数组参数中只有一个可以赋初值（上述赋值形式上的限制也适用）。

标量和数组都可以正常操作。 如果其中一个被取消设置，另一个也会自动取消设置。 我们无法在不取消设置的情况下解除变量的绑定，也无法使用 typeset 命令转换其中一个变量的类型；+T不起作用，将数组赋值给 *scalar* 会出错，将标量赋值给 *array* 会将其设置为单元素数组。

请注意，‘typeset -xT ...’和‘export -T ...’都有效，但只有标量会被标记为导出。 使用标量版本设置值会导致在所有分隔符位置被拆分（不能使用引号）。 可以将 -T 应用于两个先前绑定但使用不同分隔符的变量，在这种情况下，变量仍像以前一样连接，但分隔符会改变。

当现有标量与新数组绑定时，标量的值将被保留，但除导出之外的其他属性将不会被保留。

改变最终值的属性标志（-L、-R、-Z、-L、-u）仅在使用‘$’的参数扩展表达式时应用于扩展值。 当 shell 为任何目的在内部检索参数时，它们都不会被使用。

可指定以下属性标志：

-A

这些名称指的是关联数组参数；请参阅 数组参数。

-L [*n*]

当参数展开时，左对齐并删除值的前导空白。 如果 *n* 非零，则定义字段的宽度。 如果 *n* 为零，则宽度由第一次赋值的宽度决定。 如果是数值参数，则使用赋给参数的完整值的长度来确定宽度，而不是将输出的值。

宽度是字符数，如果使用 MULTIBYTE 选项，则可能是多字节字符。 需要注意的是，字符的屏幕宽度不会考虑在内；如果需要，请使用参数扩展标志 ${(ml…)…} 填充，如 参数扩展 中的‘参数扩展标志’所述。

当参数展开时，会在右侧填充空白，或根据需要截断以适应字段。 请注意，截断会导致数字参数出现意外结果。 如果同时设置了 -Z 标志，前导零将被删除。

-R [*n*]

与 -L 类似，但使用右对齐方式；当参数展开时，字段左侧填充空白或从末尾截断。 不得与 -Z 标志结合使用。

-U

对于数组（但不包括关联数组），只保留每个重复值的第一次出现。 对于绑定参数（参见 -T）或以冒号分隔的特殊参数，如 PATH 或 FIGNORE 等，也可以设置此标志。 需要注意的是，该标志在赋值时生效，被赋值变量的类型起决定作用；因此，对于具有共享值的变量，建议为所有接口设置该标志，例如‘typeset -U PATH path’。

当与 -f 搭配使用时，该标志的含义有所不同；见下文。

-Z [*n*]

如果与 -L 标志一起设置，则进行特殊处理。否则，类似于 -R，只是如果第一个非空字符是数字，则使用前导零进行填充，而不是空格。 数字参数受到特殊处理：它们始终符合使用零进行填充的条件，并且零被插入到输出中的适当位置。

-a

这些名称指的是数组参数。 数组参数可以通过这种方式创建，但只有启用 typeset 的保留字形式（默认情况下是这样）后，才能在 typeset 语句中对其进行赋值。 显示时，普通数组和关联数组都会显示。

-f

这些名称指的是函数而不是参数。 不能进行赋值，唯一有效的标志是 -t、-T、-k、-u、-U 和 -z。 标志 -t 开启了该函数的执行跟踪；标志 -T 的作用与此相同，但会关闭从当前函数调用的任何命名函数（非匿名函数）的跟踪，除非该函数也具有 -t 或 -T 标志。 -u 和 -U 标志会将函数标记为自动加载；-U 还会在加载函数时抑制别名扩展。 详情请参见内置命令 'autoload' 的描述。

请注意，内置的 functions 提供了与 typeset -f 相同的基本功能，但提供了一些额外的选项；autoload 为 typeset -fu 和 typeset -fU 的情况提供了更多的额外选项。

-h

隐藏：仅对特殊参数（在 由 Shell 设置的参数 中的表格中标记为 '<S>' 的参数）和与特殊参数同名的本地参数有用，但对其他参数无害。 带有此属性的特殊参数在本地化后将不会保留其特殊效果。 因此，在 'typeset -h PATH' 之后，包含 'typeset PATH' 的函数将创建一个普通的本地参数，而不具有 PATH 的通常行为。 或者，本地参数本身也可以被赋予此属性；因此，在函数 'typeset -h PATH' 中，将创建一个普通的本地参数，而特殊的 PATH 参数不会以任何方式改变。 也可以使用 'typeset +h *special*' 创建一个本地参数，此时 *special* 的本地副本将保留其特殊属性，而与 -h 属性无关。 为避免名称冲突，从 shell 模块加载的全局特殊参数（目前为 zsh/mapfile 和 zsh/parameter 中的参数）会自动获得 -h 属性。

-H

隐藏值：指定 typeset 在列出参数时不显示参数的值；对此类参数的显示总是如同给定了 '+' 标志。 在其他方面，参数的使用是正常的，如果参数是通过名称或 -m 选项的模式指定的，则该选项不适用。 对于 zsh/parameter 和 zsh/mapfile 模块中的参数，该选项默认为开启。 但需要注意的是，与 -h 标志不同，该标志也适用于非特殊参数。

-i [*n*]

使用内部整数表示。 如果 *n* 非零，则定义输出的算术基数，否则由第一个赋值决定。 允许的基数范围为 2 至 36（含 36）。

`-E` [*n*]

使用内部双精度浮点表示法。 输出时，变量将转换为科学计数法。 如果 *n* 非零，则定义要显示的有效数字个数；默认为 10。

`-F` [*n*]

使用内部双精度浮点表示法。 输出时，变量将转换为定点小数。 如果 *n* 非零，则定义小数点后显示的位数；默认为十位。

`-l`

当参数展开时，将结果转换为小写。 赋值时将 **不** 进行转换。

`-r`

给定的 *name*s 将被标记为只读。 需要注意的是，如果 *name* 是一个特殊参数，那么只读属性可以打开，但之后不能关闭。

如果设置了 `POSIX_BUILTINS` 选项，只读属性的限制性会更强：未设置的变量可以标记为 readonly，但之后不能设置；此外，只读属性不能从任何变量中移除。

但仍有可能更改变量的其他属性，其中一些属性（如 `-U` 或 `-Z` ）会影响变量值。一般来说，不应将只读属性作为一种安全机制。

请注意，在 zsh 中（与 pdksh 类似，但与大多数其他 shell 不同），仍然可以创建一个同名的局部变量，因为这被视为一个不同的变量（尽管这个变量也可以标记为只读）。已设置为只读的特殊变量在设置为本地变量后，其值和只读属性仍会保留。

`-t`

为已命名的参数打标签。 标签对 shell 没有特殊意义。 当与 `-f` 一起使用时，该标志具有不同的含义；参见上文。

`-u`

当参数展开时，将结果转换为大写。 赋值时**不**会进行转换。 当与 `-f` 一起使用时，该标志具有不同的含义；参见上文。

`-x`

标记，用于将随后执行的命令自动导出到环境中。 如果设置了选项 `GLOBAL_EXPORT`，这意味着选项 `-g`，除非同时明确给出 `+g`；换句话说，参数不会被本地化到闭合函数中。 这是为了与以前版本的 zsh 兼容。

ulimit [-HSa] [{ -bcdfiklmnpqrsTtvwx | -N *resource* } [*limit*] ...]

设置或显示 shell 和 shell 启动的进程的资源限制。 *limit* 的值可以是以下面指定的单位表示的数字，也可以是 'unlimited'（删除资源限制）或 'hard'（使用资源硬限制的当前值）中的一个值。

默认情况下，只处理软限制。如果给出 -H 标志，则使用硬限值代替软限值。 如果同时给出 -S 标志和 -H 标志，则同时设置硬限值和软限值。

如果没有使用选项，则假定使用文件大小限制 (-f)。

如果省略 *limit*，则打印指定资源的当前值。 如果打印的资源值不止一个，则会在每个值之前打印限制的名称和单位。

在对多个资源进行循环时，如果 shell 检测到一个错误的参数，它会立即终止。 但是，如果由于其他原因无法设置限制，它将继续尝试设置剩余的限制。

并非所有系统都支持以下所有资源。 运行 ulimit -a 将显示支持哪些资源。

-a

列出所有当前资源限制。

-b

以字节为单位的套接字缓冲区大小（注意：不是千字节）

-c

512 字节块，核心转储大小。

-d

千字节，表示数据段的大小。

-f

写入文件大小的， 512 字节块。

-i

待处理信号的数量。

-k

分配的 kqueues 数量。

-l

千字节，锁定内存大小。

`-m`

千字节，物理内存的大小。

`-n`

打开文件描述符。

`-p`

伪终端的数量。

`-q`

POSIX 消息队列中的字节数。

`-r`

最大实时优先级。 在某些没有这个的系统（如 NetBSD）中，为了与 sh 兼容，该选项与 `-T` 具有相同的效果。

`-s`

千字节，堆栈大小。

`-T`

用户可同时使用的线程数。

`-t`

使用的 CPU 秒数。

`-u`

用户可使用的进程数。

`-v`

虚拟内存大小，千字节。 在某些系统中，这是指称为 '地址空间' 的限制。

`-w`

千字节，表示被交换出去的内存大小。

`-x`

文件锁的数量。

资源也可以通过整数形式 '`-N` *resource*' 指定，其中 *resource* 对应于操作系统为资源定义的整数。 这可以用来设置 shell 已知的、与选项字母不对应的资源的限制。 这些限制将在 '`ulimit -a`' 的输出中以数字显示。

该数字也可能超出编译到 shell 中的限制范围。 shell 会尝试读取或写入限制值，如果失败则会报错。

umask [`-S`] [*mask*]

umask 设置为 *mask*。 *mask* 可以是八进制数，也可以是 chmod(1) 手册页中描述的符号值。 如果省略 *mask*，则打印当前值。 `-S` 选项会将掩码打印为符号值。 否则，掩码将以八进制数的形式打印。 请注意，在符号形式下，您指定的权限是允许（而不是拒绝）给指定用户的权限。

unalias [`-ams`] *name* ...

删除别名。 该命令的作用与 `unhash -a` 相同，但 `-a` 选项会移除所有常规或全局别名，或 `-s` 会移除所有后缀别名：在这种情况下，不能出现 *name* 参数。 选项 `-m`（按模式移除）和 `-s`（不含 `-a`）（移除列出的后缀别名）的作用与 `unhash -a` 相同。 注意 `-a` 的含义在 `unalias` 和 `unhash` 中有所不同。

unfunction

与 `unhash -f` 相同.

unhash [`-adfms`] *name* ...

从内部哈希表中删除名为 *name* 的元素。 默认是从命令哈希表中移除元素。 使用 `-a` 选项，unhash 会移除常规或全局别名；注意移除全局别名时，参数必须加引号，以防止在传递给命令前被展开。 使用 `-s` 选项，unhash 会移除后缀别名。 `-f` 选项会使 unhash 删除 shell 函数。 通过 `-d` 选项，unhash 会移除命名目录。 如果给定了 `-m` 标志，参数将被视为模式（应加引号），相应哈希表中名称匹配的所有元素都将被移除。

unlimit [`-hs`] *resource* ...

每个 *resource* 的资源限制都会设置为硬限制。 如果给定了 `-h` 标志，且 shell 拥有相应权限，则会删除每个 *resource* 的硬资源限制。 只有在给出 `-s` 标志的情况下，shell 进程的资源才会被更改。

当 shell 以模拟其他 shell 的模式启动时，`unlimit` 命令默认不可用。 可以使用 '`zmodload -F zsh/rlimits b:unlimit`' 命令来使用它。

unset [`-fmv`] *name* ...

每个已命名的参数都是未设置的。 即使未设置，本地参数仍是本地参数；它们在作用域内显示为未设置，但当作用域结束时，先前的值仍会重新出现。

关联数组参数的单个元素可以通过 *name* 上的下标语法取消设置，下标应加引号（或整个命令前加 noglob），以防止文件名生成。

如果指定了 -m 标志，参数将被视为模式（应加引号），所有名称匹配的参数将被取消设置。 需要注意的是，该标志不能用于关联数组元素的取消设置，因为下标将被视为模式的一部分。

-v 标志指定 *name* 指向参数。这是默认行为。

unset -f 等价于 unfunction。

unsetopt [{+|-}*options* | {+|-}o *option_name*] [*name* ...]

取消设置 shell 的选项。 所有用标志或名称指定的选项都会被取消设置。 如果没有提供参数，则会打印当前未设置的所有选项名称。 如果给出 -m 标志，参数将作为模式（应加引号以防止被解释为 glob 模式），所有名称与这些模式匹配的选项将被取消设置。

vared

参见 Zle 内置命令。

wait [*job* ...]

等待指定的作业或进程。 如果未给出 *job*，则会等待所有当前活动的子进程。 每个 *job* 既可以是作业指示器，也可以是作业表中作业的进程 ID。 该命令的退出状态是所等待作业的退出状态。 如果 *job* 代表未知作业或进程 ID，则会打印警告（除非设置了 POSIX_BUILTINS 选项），退出状态为 127。

可以等待最近在后台运行的进程（由进程 ID 指定，而不是由作业指定），即使该进程已经退出。 通常情况下，进程 ID 会在进程启动后立即通过捕获变量 $! 的值记录下来。 shell 记忆的进程 ID 数量是有限制的，这个限制由系统配置参数 CHILD_MAX 的值决定。 当达到这个限制时，较旧的进程 ID 将被丢弃，最近启动的进程将被优先丢弃。

注意，没有机制来防止进程ID的回绕，即如果等待操作没有足够快地执行，那么等待的可能是错误的进程。冲突意味着两个进程ID都是由shell生成的，因为其他进程并没有被记录下来，并且用户可能对这两个进程都感兴趣，所以这个问题是进程ID固有的。

whence [-vcwfpamsS] [-x *num*] *name* ...

对于每个 *name*，指出如果用作命令名时，将如何解释。

如果 *name* 不是别名、内置命令、外部命令、shell 函数、散列的命令或保留字，则退出状态应为非零，并且 — 如果传递了 -v、-c 或 -w — 一条信息将被写入标准输出。 (这与其他 shell 将信息写入标准错误不同）。

whence 在 *name* 只是命令的最后一个路径组件，即不包括‘/’时，最有用；特别是，只有在只传递命令的非目录组件时，模式匹配才会成功。

`-v`

生成更详细的报告。

`-c`

以 *csh* 类似的格式打印结果。 其优先级高于 `-v`。

`-w`

对于每个 *name*，打印 ‘*name*: *word*’ ，其中 *word*是 `alias`, `builtin`, `command`, `function`, `hashed`, `reserved` 或 none中的一个，根据 *name* 对应的别名、内置命令、外部命令、shell 函数、使用 `hash` 内置命令定义的命令、保留字或未识别。 其优先级高于 `-v` 和 `-c`。

`-f`

显示 shell 函数的内容，这在不使用 `-c` 标志时，不会显示。

`-p`

对 *name* 进行路径搜索，即使它是别名、保留字、shell 函数或内置命令。

`-a`

在命令路径中搜索 *name* 的所有出现次数。 通常只打印第一次出现的内容。

`-m`

参数被视为模式（模式字符应加引号），每条与其中一个模式匹配的命令的信息都将显示。

`-s`

如果路径名包含符号链接，则同时打印不含符号链接的路径名。

`-S`

与 `-s`相同，但如果需要通过多个符号链接来解析路径名，则也会打印中间步骤。 每一步解析的符号链接可能在路径的任何位置。

`-x` *num*

使用 `-c` 选项输出 shell 函数时展开制表符。 这与 `functions` 内置命令的选项 `-x` 的效果相同。

`where` [`-wpmsS`] [`-x` *num*] *name* ...

等价于 whence -ca.

which [-wpamsS] [-x *num*] *name* ...

等价于 whence -c.

zcompile [-U] [-z | -k] [-R | -M] *file* [*name* ...]
zcompile -ca [-m] [-R | -M] *file* [*name* ...]
zcompile -t *file* [*name* ...]

该内置命令可用于编译函数或脚本，将编译后的形式存储在文件中，并检查包含编译后形式的文件。 这样可以避免在读取文件时对文本进行解析，从而加快函数的自动加载和脚本的引入。

第一种形式（不含 -c、-a 或 -t 选项）会创建一个编译文件。 如果只给出 *file* 参数，输出文件的名称为 '*file*.zwc'，并与 *file* 位于同一目录。 当函数被自动加载时，shell 将加载编译后的文件，而不是普通的函数文件；关于如何搜索自动加载的函数，请参阅 函数。 扩展名 .zwc 代表 'zsh word code'。

如果至少有一个 *name* 参数，所有命名的文件都会被编译到作为第一个参数给出的输出 *file* 中。 如果 *file* 不是以 .zwc 结尾，则会自动添加此扩展名。 包含多个编译函数的文件称为 'digest' 文件，可用作 FPATH/fpath 特殊数组的元素。

第二种形式是使用 -c 或 -a 选项，将所有指定函数的编译定义写入 *file*。 对于 -c，函数名必须是 shell 当前定义的函数，而不是标记为自动加载的函数。 标记为自动加载的未定义函数可以通过使用 -a 选项来编写，在这种情况下，系统会搜索 fpath，如果找到这些函数的定义文件，就会将其内容编译到 *file* 中。 如果同时给出 -c 和 -a，则可能同时给出已定义函数和标记为自动加载的函数的名称。 无论哪种情况，使用 -c 或 -a 选项编写的文件中的函数都将被自动加载，就像未设置 KSH_AUTOLOAD 选项一样。

之所以要用不同的选项来处理已加载和尚未加载的函数，是因为有些自动加载的定义文件会定义多个函数，包括与文件同名的函数，并在最后调用该函数。 在这种情况下，'zcompile -c' 的输出不包括文件中定义的额外函数，文件中的其他初始化代码也会丢失。 使用'zcompile -a' 可以捕获所有这些额外信息。

如果 -m 选项与 -c 或 -a 结合使用，则 *name*s 将被用作模式，所有名称与其中一个模式匹配的函数都将被写入。如果未给出 *name*，则将写入当前定义或标记为自动加载的所有函数的定义。

注意，如果函数将重定向作为定义的一部分而不是在包含在函数主体中，第二种形式不能用于编译这样的一个函数，例如

```
fn1() { { ... } >~/logfile }
```

可以被编译，但是

```
fn1() { ... } >~/logfile
```

不能。 可以使用 zcompile 的第一种形式来编译包含完整函数定义（而不仅仅是函数体）的可自动加载函数。

第三种形式是使用 -t 选项，检查现有的编译文件。 在没有其他参数的情况下，会列出编译到该文件中的原始文件名。 第一行输出显示编译该文件的 shell 版本，以及使用该文件的方式（即直接读取还是映射到内存中）。 如果有参数，则不输出任何内容，如果 ***all*** 和 *names* 的定义被找到，则返回状态设为零；如果没有找到至少一个 *name* 的定义，则返回状态为非零。

其它选项：

-U

编译（名为） *name* 文件时不会扩展别名。

-R

读取编译文件时，其内容会被复制到 shell 的内存中，而不是内存映射（参见 -M）。 这在不支持内存映射的系统上会自动发生。

在编译脚本而不是自动加载函数时，通常最好使用该选项；否则整个文件，包括定义已定义函数的代码，都将保持映射状态，从而浪费内存。

-M

读取编译文件时，该文件会映射到 shell 的内存中。这样一来，在同一主机上运行的多个 shell 实例将共享这个映射文件。 如果既未给出 -R 也未给出 -M，zcompile 内置程序将根据编译文件的大小决定如何处理。

-k
-z

当编译文件包含需要自动加载的函数时，将使用这些选项。如果给出 -z，函数将被自动加载，就像 KSH_AUTOLOAD 选项 **没有** 设置一样，即使该选项在读取编译文件时已被设置；而如果给出 -k，函数将被加载，就像 KSH_AUTOLOAD **已被** 设置一样。这些选项优先于 autoload 内置函数中指定的 -k 或 -z 选项。如果没有给出这两个选项，函数将根据读取编译文件时 KSH_AUTOLOAD 选项的设置来加载。

这些选项还可以根据需要多次出现在列出的 *names* 之间，以指定所有后续函数的加载方式，直至下一个 -k 或 -z。

创建的文件总是包含两个版本的编译格式，一个适用于大二进制（big-endian）机器，另一个适用于小二进制机器。 这样做的结果是，编译后的文件与机器无关，如果对其进行读取或映射，实际上只会使用文件（和映射）的一半。

`zformat`

参见 zsh/zutil 模块.

`zftp`

参见 zsh/zftp 模块。

`zle`

参见 Zle 内置命令。

`zmodload` [`-dL`] [`-s`] [...]
`zmodload -F` [`-alLme -P` *param*] *module* [[+-]*feature* ...]
`zmodload -e` [`-A`] [...]
`zmodload` [`-a` [`-bcpf` [`-I`]]] [`-iL`] ...
`zmodload -u` [`-abcdpf` [`-I`]] [`-iL`] ...
`zmodload -A` [`-L`] [*modalias*[=*module*] ...]
`zmodload -R` *modalias* ...

执行与 zsh 可加载模块相关的操作。 在 shell 运行时加载模块（"动态加载"）并不适用于所有操作系统，也不适用于特定操作系统上的所有安装，尽管 `zmodload` 命令本身总是可用的，并且可以用来操作内置于没有动态加载的 shell 可执行文件版本中的模块。

在不带参数的情况下，将打印当前加载的所有二进制模块的名称。 如果使用 `-L` 选项，则会以一系列 `zmodload` 命令的形式显示该列表。 带参数的形式有：

`zmodload` [`-is`] *name* ...
`zmodload -u` [`-i`] *name* ...

在最简单的情况下，`zmodload` 会加载一个二进制模块。 模块必须位于一个文件中，文件名由指定的 *name* 和标准后缀组成，通常为 '`.so`'（HPUX 上为'`.sl`'）。 如果要加载的模块已经加载，重复的模块将被忽略。 如果 `zmodload` 检测到不一致的情况，例如模块名称无效或循环依赖列表，当前代码块将被中止。 如果模块可用，则会在必要时加载模块；如果模块不可用，则会静默返回非零状态。 选项 `-i` 可以接受，但没有任何作用。

搜索 *name* 模块的方式与搜索命令的方式相同，使用 `$module_path` 而不是 `$path`。 不过，即使模块名称中包含 '`/`'，路径搜索也会执行，而通常情况下模块名称中都会包含 '`/`'。 没有办法阻止路径搜索。

如果模块支持功能（见下文），`zmodload` 会在加载模块时尝试启用所有功能。 如果模块已成功加载，但无法启用所有功能，`zmodload` 将返回状态 2。

如果给出选项 -s，则在模块不可用的情况下不会打印错误信息（但会打印表明模块有问题的其他错误信息）。 返回状态则表明模块是否已加载。 如果调用者认为模块是可选的，则可以使用这种方式。

通过 -u，zmodload 可以卸载模块。 必须使用与加载模块时相同的 *name*，但模块不必存在于文件系统中。 如果模块已被卸载（或从未被加载），则 -i 选项会抑制错误。

每个模块都有启动和清理函数。 如果模块的启动函数失败，模块将无法加载。 同样，只有当模块的清理功能成功运行时，模块才能被卸载。

zmodload -F [-almLe -P *param*] *module* [[+-]*feature* ...]

zmodload -F 允许对模块提供的功能进行更有选择性的控制。 如果除 -F 之外没有其他选项，则会加载名为 *module* 的模块（如果尚未加载），并将 *feature* 列表设置为所需状态。 如果没有指定 *feature*，则会加载尚未加载的模块，但特性（features）的状态保持不变。 每个特性前都可以使用 + 打开该特性，或使用 - 关闭该特性；如果两个字符都不存在，则假定使用 +。 任何未明确提及的特性都将保持当前状态；如果模块之前未加载过，这意味着任何此类特性都将保持禁用状态。 如果所有特性都已设置，则返回状态为 0；如果模块加载失败，则返回状态为 1；如果某些特性无法设置（例如，由于存在同名的其他参数而无法添加参数），但模块已加载，则返回状态为 2。

标准功能包括内置命令、条件、参数和数学函数；它们分别用前缀 'b:'，'c:'（'C:' 表示下位条件）、'p:' 和'f:' 表示，后跟相应特性在 shell 中的名称。 例如，'b:strftime' 表示名为 strftime 的内置函数，p:EPOCHSECONDS 表示名为 EPOCHSECONDS 的参数。 模块可根据其文档提供其他（'抽象'）特性；这些功能没有前缀。

使用 -l 或 -L 时，将列出模块提供的特性。 如果只使用 -l，则会显示特性列表及其状态，每行一个功能。 如果仅使用 -L，则会显示一条 zmodload -F 命令，该命令将开启模块的已启用功能。 如果使用 -lL，则会显示 zmodload -F 命令，该命令会将所有功能设置为当前状态。 如果使用 -P *param* 选项给出上述组合之一 ，则参数 *param* 将被设置为一个特性的数组，可以是特性及其状态，也可以是已启用的特性（如果只给出 -L）。

如果使用选项 -L，则可以省略模块名称；然后以 zmodload -F 命令的形式打印所有提供特性的模块的所有已启用特性列表。 如果同时给出 -l，启用和禁用特性的状态都会以这种形式输出。

在提供一组特性时，可以同时提供 -l 或 -L 和一个模块名称；在这种情况下，只考虑这些特性的状态。 每个特征前都可以加上 + 或 -，但该字符不起作用。 如果没有提供特性集，则会考虑所有特性。

使用 -e，命令首先会测试模块是否已加载；如果未加载，则返回状态 1。 如果模块已加载，则检查作为参数给出的特性列表。 对于不带前缀的特性，只

需测试模块是否提供；对于带前缀 + 或 - 的特性，则测试是否提供并处于指定状态。 如果对列表中所有特性的测试都成功，则返回状态 0，否则返回状态 1。

使用 -m，给定的特性列表中的每个条目都将作为一个模式，与模块提供的特性列表进行匹配。 必须明确给出开头的 + 或 -。 该选项不能与 -a 选项结合使用，因为必须明确指定自动加载。

使用 -a，给定的特性列表将被标记为从指定模块自动加载（该模块可能尚未加载）。 特性名称前可以出现一个可选的 +。 如果特性前缀为 -，则现有的自动加载将被移除。 选项 -l 和 -L 可用来列出自动加载。 自动加载是针对单个特性的；加载模块时，只有请求的特性才会启用。 如果模块随后被卸载，自动加载请求将被保留，直到出现明确的 'zmodload -Fa *module* -*feature*'。 为已加载模块的特性请求自动加载不会出错。

加载模块时，会根据模块实际提供的特性检查每个自动加载请求；如果未提供该特性，自动加载请求将被删除。输出警告信息；如果加载模块是为了提供其他特性，且自动加载成功，则不会影响当前命令的状态。 如果 zmodload -Fa 运行时模块已被加载，则会打印一条错误信息并返回状态 1。

zmodload -Fa 可以与 -l、-L、-e 和 -P 选项一起使用，用于列出和测试是否存在可自动加载的特性。 在这种情况下，如果指定了 -L，则 -l 将被忽略。 zmodload -FaL 不含模块名称，会列出所有模块的自动加载。

请注意，只有上述标准特性可以自动加载；其他特性需要在启用前先加载模块。

zmodload -d [-L] [*name*]
zmodload -d *name dep* ...
zmodload -ud *name* [*dep* ...]

-d 选项可用于指定模块依赖关系。 第二个参数和后续参数中指定的模块将在第一个参数中指定的模块之前加载。

使用 -d 和一个参数时，将列出该模块的所有依赖关系。 如果使用 -d，且没有参数，则会列出所有模块的依赖关系。 该列表默认采用类似 Makefile 的格式。 -L 选项会将格式更改为 zmodload -d 命令的列表。

如果同时使用 -d 和 -u，依赖关系将被移除。 如果只给出一个参数，则会移除该模块的所有依赖关系。

zmodload -ab [-L]
zmodload -ab [-i] *name* [*builtin* ...]
zmodload -ub [-i] *builtin* ...

-ab 选项定义自动加载的内置程序。 它定义了指定的 *builtin*s。 调用其中任何一个内置程序时，第一个参数中指定的模块将被加载并启用其所有特性（如

需选择性控制特性，请使用‘zmodload -F -a’，如上所述）。 如果只给出 *name*，则会定义一个与模块同名的内置程序。 如果内置程序已被定义或自动加载，-i 会抑制错误，但如果另一个同名内置程序已被定义，-i 则不会抑制错误。

如果使用 -ab 且不带参数，则会列出所有自动加载的内置程序，内置程序名称后的括号中会显示模块名称（如果不同）。 使用 -L 选项后，格式将变为 zmodload -a 命令列表。

如果 -b 与 -u 选项一起使用，则会删除之前用 -ab 定义的内置程序。 这只有在内置程序尚未加载的情况下才会发生。 如果内置函数已被移除（或从未存在），则 -i 会抑制错误。

如果模块随后被卸载，直到发出明确的‘zmodload -ub *builtin*’，自动加载请求将被保留。

zmodload -ac [-IL]
zmodload -ac [-iI] *name* [*cond* ...]
zmodload -uc [-iI] *cond* ...

-ac 选项用于定义自动加载的条件代码。*cond* 字符串给出模块定义的条件名称。可选的 -I 选项用于定义中缀条件名。如果没有该选项，则定义前缀条件名。

如果没有给出条件名称，则会列出所有已定义的名称（如果给出 -L 选项，则会列出一系列 zmodload 命令）。

选项 -uc 会删除自动加载条件的定义。

zmodload -ap [-L]
zmodload -ap [-i] *name* [*parameter* ...]
zmodload -up [-i] *parameter* ...

-p 选项与 -b 和 -c 选项类似，但它使 zmodload 对自动加载的参数起作用。

zmodload -af [-L]
zmodload -af [-i] *name* [*function* ...]
zmodload -uf [-i] *function* ...

-f 选项与 -b、-p 和 -c 选项类似，但它使 zmodload 在自动加载的数学函数上工作。

zmodload -a [-L]
zmodload -a [-i] *name* [*builtin* ...]
zmodload -ua [-i] *builtin* ...

等同于 -ab 和 -ub。

`zmodload -e [ -A ] [ string ... ]`

不带参数的 `-e` 选项会列出所有已加载模块；如果同时给出 `-A` 选项，则也会显示与已加载模块相对应的模块别名。 如果提供了参数，则不会打印任何内容；如果参数 *string*s 都是已加载模块的名称，则返回状态设为 0；如果至少有一个 *string* 不是已加载模块的名称，则返回状态设为 1。 这可用于测试模块实现的功能是否可用。 在这种情况下，任何别名都会自动解析，并且不会使用 `-A` 标志。

`zmodload -A [ -L ] [ modalias[=module] ... ]`

对于每个参数，如果同时给出 *modalias* 和 *module*，则定义 *modalias* 为 *module* 模块的别名。 如果随后通过调用 `zmodload` 或隐式的方式请求 *modalias* 模块，shell 将尝试加载 *module* 模块。 如果未给出 *module*，则显示 *modalias* 的定义。 如果未给出参数，则会列出所有已定义的模块别名。 在列出时，如果还给出了 `-L` 标志，则会以 `zmodload` 命令的形式列出定义，以重新创建别名。

模块别名的存在与解析的名称是否实际作为模块加载完全无关：当别名存在时，以任何别名加载或卸载模块都与使用解析后的名称具有完全相同的效果，并且不会影响别名与解析后名称之间的联系，这种联系可以通过 `zmodload -R` 或重新定义别名来移除。 只要不循环，别名链（即第一个解析后名称本身就是一个别名）是有效的。 由于别名采用与模块名相同的格式，因此可以包含路径分隔符：在这种情况下，不要求指定路径的任何部分都存在，因为别名将首先被解析。 例如，‘`any/old/alias`’始终是一个有效的别名。

添加到别名模块的依赖关系实际上会添加到解析后的模块中；如果删除别名，这些依赖关系仍会保留。 创建名称为标准 shell 模块之一的别名，并将其解析为不同的模块是有效的。 但是，如果模块有依赖关系，则无法使用模块名称作为别名，因为模块本身已被标记为可加载模块。

除上述外，`zmodload` 命令还可在任何需要模块名的地方使用别名。 不过，别名将不会显示在已加载模块列表中，只显示 ‘`zmodload`’。

`zmodload -R` *modalias ...*

对于之前通过 `zmodload -A` 定义为模块别名的 *modalias* 参数，删除该别名。 如果未定义任何别名，将导致错误并忽略该行的其余部分。

请注意，`zsh` 并不区分链接到 shell 的模块和动态加载的模块。在这两种情况下，都必须使用这个内置命令来提供内置模块和其他由模块定义的内容（除非模块是根据这些定义自动加载的）。即使不支持动态加载模块的系统也是如此。

`zparseopts`

参见 zsh/zutil 模块.

`zprof`

参见 zsh/zprof 模块.

`zpty`

参见 zsh/zpty 模块.

`zregexparse`

参见 zsh/zutil 模块.

`zsocket`

参见 zsh/net/socket 模块.

`zstyle`

参见 zsh/zutil 模块.

`ztcp`

参见 zsh/net/tcp 模块.

---

18 Zsh 行编辑器

18.1 说明

如果设置了 `ZLE` 选项（交互式 shell 的默认设置），并且 shell 输入连接到终端，用户就可以编辑命令行。

有两种显示模式。 第一种是默认的多行模式。 它只有在 `TERM` 参数设置成可以向上移动光标的有效终端类型时才起作用。 第二种是单行模式，如果 `TERM` 无效或无法向上移动光标，或者设置了 `SINGLE_LINE_ZLE` 选项，则会使用这种模式。 该模式与 *ksh* 类似，不使用 termcap 序列。 如果 `TERM` 为 "emacs"，则默认不设置 `ZLE` 选项。

行编辑器也会使用 `BAUD`、`COLUMNS` 和 `LINES` 参数。请参阅 Shell 使用的参数。

行编辑器也使用 `zle_highlight` 参数；请参阅 字符高亮。 特殊字符以及光标和标记之间区域的高亮（在 Emacs 模式下使用 `set-mark-command`，或在 Vi 模式下使用 `visual-mode`）默认为启用；更多信息请参阅本参考文献。 不喜欢高亮的用户可以通过以下设置禁用所有高亮功能：

```
zle_highlight=(none)
```

很多地方都会提到 数字参数。 默认情况下，在 emacs 模式下，可以按住 alt 键并输入数字，或在每个数字前按下 escape 键；在 vi 命令模式下，可以在输入命令前输入数字。 一般来说，除非下文另有说明，否则数字参数会使下一条输入的命令重复指定的次数；这由 digit-argument 小部件实现。有关修改数字参数的其他方法，请参阅 参数。

18.2 键映射

ZLE 中的键映射包含键序列与 ZLE 命令之间的一组绑定。 空键序无法绑定。

任何时候都可以有任意数量的键映射，每个键映射都有一个或多个名称。 如果删除了键映射的所有名称，它就会消失。 bindkey 可用来操作键映射名称。

最初有八个键映射：

emacs

: EMACS 模拟

viins

: vi 模拟 - 插入模式

vicmd

: vi 模拟 - 命令模式

viopp

: vi 模拟 - 操作符待处理（operator pending）

visual

: vi 模拟 - 选择激活

isearch

: 增量搜索模式

command

: 读取命令名

.safe

: 后备键映射

‘.safe’ 键映射是特殊的。 它永远无法更改，名称也永远无法删除。 不过，它可以链接到其他名称，这些名称可以删除。 将来可能会添加其他特殊的键映射；用户应避免在自己的键映射中使用以 ‘.’ 开头的名称。

除这些名称外，‘emacs’ 或 ‘viins’ 也与名称 ‘main’ 相关联。 当 shell 启动时，如果 VISUAL 或 EDITOR 环境变量之一包含字符串 ‘vi’ ，那么它就是 ‘viins’，否则就是 ‘emacs’。 bindkey 的 -e 和 -v 选项提供了一种方便的方式来覆盖默认选择。

当编辑器启动时，它会选择 ‘main’ 键映射。 如果键映射不存在，则会使用 ‘.safe’ 代替。

在 ‘.safe’ 键映射中，除了 ^J（换行）和 ^M（回车）与 accept-line 绑定外，每个单键都与 self-insert 绑定。 这种键映射方式故意让人觉得不好用；如果你正在使用它，就意味着你删除了主键映射，你应该把它放回去。

18.2.1 读取命令

当 ZLE 从终端读取命令时，它可能会读取一个与某些命令绑定的序列，该序列也是一个更长绑定字符串的前缀。 在这种情况下，ZLE 会等待一段时间，看看是否有更多的字符被输入，如果没有（或者它们与更长的字符串不匹配），它就会执行绑定。 超时时间由 KEYTIMEOUT 参数定义；默认值为 0.4 秒。 如果前缀字符串本身未与命令绑定，则不会超时。

当 ZLE 在适当模式下从多字节字符串读取字节时，也会应用按键超时。 (这要求 shell 在编译时启用了多字节模式；通常情况下，locale 中的字符也采用 UTF-8 编码，但也支持操作系统已知的任何多字节编码)。 如果在超时时间内没有读取第二个或后续字节，shell 就会像键入 ? 一样，并重置输入状态。

除了 ZLE 命令外，还可以使用 ‘bindkey -s’ 将按键序列与其他字符串绑定。 当这样的序列被读取时，替换字符串会被推回作为输入，然后命令读取过程会使用这些假按键重新开始。 这种输入本身可以调用更多的替换字符串，但为了检测循环，如果有 20 个这样的替换字符串而没有真正的命令被读取，进程就会停止。

用户输入的按键序列可以通过用 read-command 小部件定义的用户小部件转化为命令名称，在杂项中描述。

18.2.2 本地键映射

虽然在正常编辑中只使用一个键映射，但在许多模式下，本地键映射允许自定义某些键。例如，在增量搜索模式中，isearch 键映射中的绑定将覆盖 main 键映射中的绑定，但所有未覆盖的键仍可使用。

如果在本地键映射中定义了一个键序列，它将隐藏全局键映射中作为该序列前缀的键序列。例如，在 viopp 中绑定 iw 会隐藏 i 在 vicmd 中的绑定。不过，全局键映射中共享

相同前缀的更长序列仍然适用，例如，全局键映射中 ^Xa 的绑定将不受本地键映射中 ^Xb 绑定的影响。

18.3 Zle 内置命令

ZLE 模块包含三个相关的内置命令。bindkey 命令操作键映射和键绑定；vared 命令根据 shell 参数值调用 ZLE；zle 命令操作编辑小部件，并允许从 shell 函数中通过命令行访问 ZLE 命令。

bindkey [*options*] -l [-L] [*keymap ...*]
bindkey [*options*] -d
bindkey [*options*] -D *keymap ...*
bindkey [*options*] -A *old-keymap new-keymap*
bindkey [*options*] -N *new-keymap* [*old-keymap*]
bindkey [*options*] -m
bindkey [*options*] -r *in-string ...*
bindkey [*options*] -s *in-string out-string ...*
bindkey [*options*] *in-string command ...*
bindkey [*options*] [*in-string*]

bindkey 的选项可分为三类：当前命令的键映射选择、操作选择及其他。 键映射选择选项有：

-e

选择键映射 ‘emacs’ 以执行当前命令的任何操作，并将 ‘emacs’ 链接到 ‘main’，以便下次启动编辑器时默认选择该键映射。

-v

选择键映射 ‘viins’ 以执行当前命令的任何操作，并将 ‘viins’ 链接到 ‘main’，以便下次启动编辑器时默认选择该键映射。

-a

为当前命令的任何操作选择键映射 ‘vicmd’。

-M *keymap*

keymap 指定了一个键映射名称，当前命令进行任何操作时都会选择该名称。

如果需要选择键映射，且未使用上述任何选项，则使用 ‘main’ 键映射。 某些操作不允许选择键映射，即

-l

列出所有已存在的键映射名称；如果给出任何参数，则只列出这些键映射。

如果同时使用 -L 选项，则以 bindkey 命令的形式列出创建或链接键映射的命令。 'bindkey -lL main' 显示哪个键映射链接到了 'main'（如果有的话），从而显示标准 emacs 或 vi 模拟是否生效。 该选项不会显示 .safe 键映射，因为它不能以这种方式创建；不过， 'bindkey -lL .safe' 也不会被报告为错误，它只是什么也不输出。

-d

删除所有现有键映射并重置为默认状态。

-D *keymap* ...

删除已命名的 *keymaps* 键映射。

-A *old-keymap new-keymap*

将 *new-keymap* 作为 *old-keymap* 的别名，这样两个名称就都指向同一个键映射。 这两个名称具有同等地位；如果其中一个被删除，另一个将保留。 如果已经有一个键映射使用 *new-keymap* 名称，则该名称将被删除。

-N *new-keymap* [*old-keymap*]

创建一个新的键映射，命名为 *new-keymap*。 如果已有键映射使用该名称，则将其删除。 如果已给出 *old-keymap* 名称，新键映射将被初始化为该名称的副本，否则新键映射将为空。

要使用新创建的键映射，必须将其链接到 main。 因此，创建并使用从 emacs键映射（保持不变）初始化的新键映射 'mymap' 的命令序列如下：

```
bindkey -N mymap emacs
bindkey -A mymap main
```

请注意，虽然 'bindkey -A *newmap* main' 在 *newmap* 为 emacs 或 viins 时有效，但在 vicmd 时无效，因为无法从 vi 插入模式切换到命令模式。

如果没有给出键映射选择选项，下列操作将作用于 'main' 键映射：

-m

将内置的元键绑定集添加到选定的键映射中。 只有未绑定或绑定到 self-insert 的按键才会受到影响。

-r *in-string* ...

解除选定键映射中指定的 *in-strings* 的绑定。 这完全等同于将字符串绑定到 undefined-key。

如果同时使用 -R，则将 *in-string*s 解释为范围。

当同时使用 -p 时，*in-string* 将指定前缀。 任何以给定的 *in-string* 为前缀的绑定（不包括 *in-string* 本身的绑定（如果有））都将被删除。 例如

```
bindkey -rpM viins '^['
```

将删除 vi-insert 键映射中所有以转义字符开头的绑定（可能是光标键），但保留转义字符本身的绑定（可能是 vi-cmd-mode）。 这与选项 -R 不兼容。

-s *in-string out-string ...*

将每个 *in-string* 与每个 *out-string* 绑定。 输入 *in-string* 时，*out-string* 将被推回，并被视为行编辑器的输入。 当 -R 也被使用时，*in-string* 将被解释为范围。

请注意，如下所述，*in-string* 和 *out-string* 都需要进行相同形式的解释。

in-string command ...

将每个 *in-string* 与每个 *command* 绑定。 当使用 -R 时，将 *in-string* 解释为范围。

[*in-string*]

列出按键绑定。 如果指定了 *in-string*，则会显示所选键映射中该字符串的绑定。 否则，将显示所选键映射中的所有按键绑定。 (作为特例，如果单独使用 -e 或 -v 选项，键映射将 **不** 显示–键映射的隐式链接是唯一会发生的情况）。

当使用 -p 选项时，*in-string* 必须存在。 列表显示的是以给定键序为前缀的所有绑定，不包括键序本身的任何绑定。

当使用 -L 选项时，列表将是 bindkey 命令创建键绑定的形式。

如上所述，使用 -R 选项时，有效范围由两个字符组成，两个字符之间还有一个可选的‘-’。 指定的两个字符之间的所有字符（包括在内）都会按指定的方式绑定。

对于 *in-string* 或 *out-string*，可以识别以下转义序列：

\a

响铃字符

\b

退格键

\e, \E

转义

\f

分页符（form feed）

\n

换行符

\r

回车

\t

水平制表符

\v

垂直制表符

NNN

八进制字符代码

\x*NN*

十六进制字符代码

\u*NNNN*

十六进制 unicode 字符代码

\U*NNNNNNNN*

十六进制 unicode 字符代码

\M[-]*X*

设置了元位的字符

\C[-]*X*

控制字符

^*X*

控制字符

在所有其他情况下，'\' 会转义后面的字符。 删除则写成 '^?'。 请注意， '\M^?' 和 '^\M?' 并不相同，而且（与 emacs 不同），'\M-*X*' 和 '\e*X*' 的绑定完全不同，尽管它们被 'bindkey -m' 初始化为相同的绑定。

vared [-Aacghe] [-p *prompt*] [-r *rprompt*]
[-M *main-keymap*] [-m *vicmd-keymap*]
[-i *init-widget*] [-f *finish-widget*]
[-t *tty*] *name*

参数 *name* 的值被载入编辑缓冲区，并调用行编辑器。 编辑器退出后，*name* 将被设置为编辑器返回的字符串值。 如果给出 -c 标志，则会创建尚未存在的参数。 -a 标志可与 -c 一起使用，以创建数组参数，或使用 -A 标志创建关联数组。 如果现有参数的类型与要创建的类型不匹配，参数将被取消设置并重新创建。 可以使用 -g 标志来抑制 WARN_CREATE_GLOBAL 和 WARN_NESTED_VAR 选项的警告。

如果正在编辑数组或数组片段，$IFS 中定义的分隔符会显示为用反斜线引用，反斜线本身也是如此。 相反，当编辑的文本被分割成一个数组时，反斜线会引用紧随其后的分隔符或反斜线；不会对反斜线进行其他特殊处理，也不会对引号进行任何处理。

可以使用 *name* 的下标语法编辑现有数组或关联数组参数的单个元素。 即使没有 -c，也会自动创建新元素。

如果给出 -p 标志，则以下字符串将作为提示符显示在左侧。 如果给出 -r 标志，则下面的字符串将作为提示符显示在右侧。 如果指定了 -h 标志，则可以通过 ZLE 访问历史记录。如果给出 -e 标志，在空行上键入 ^D (Control-D) 会导致 vared 立即退出，并返回一个非零的返回值。

-M 选项给出了一个键映射，在编辑过程中链接到 main 键映射，而 -m 选项给出了一个键映射，在编辑过程中链接到 vicmd 键映射。 对于 vi-style 编辑，这允许一对键映射覆盖 viins 和 vicmd。 对于 emacs 风格的编辑，通常只需要 -M，但仍可使用 -m 选项。 退出时，将恢复之前的键映射。

Vared 会在控制之前和之后调用常规的 'zle-line-init' 和 'zle-line-finish' 钩子。使用 -i 和 -f 选项，可以用其他自定义小部件来替换这些钩子。

如果给出'-t *tty*'，则 *tty* 是要使用的终端设备名称，而不是默认的 /dev/tty。 如果 *tty* 并非指向终端设备，系统将报错。

zle
zle -l [-L | -a] [*string ...*]
zle -D *widget ...*
zle -A *old-widget new-widget*
zle -N *widget* [*function*]
zle -f *flag* [*flag...*]
zle -C *widget completion-widget function*

zle -R [-c] [*display-string*] [*string ...*]
zle -M *string*
zle -U *string*
zle -K *keymap*
zle -F [-L | -w] [*fd* [*handler*]]
zle -I
zle -T [tc *function* | -r tc | -L]
zle *widget* [-n *num*] [-f *flag*] [-Nw] [-K *keymap*] *args ...*

zle 内置命令会执行一系列与 ZLE 有关的不同操作。

在没有选项和参数的情况下，只会设置返回状态。 如果 ZLE 当前处于激活状态，且可以使用该内置命令调用小部件，则状态为 0，否则为非 0。 请注意，即使返回非零状态，zle 仍可能作为补全系统的一部分处于活动状态；这不允许直接调用 ZLE 小部件。

否则，执行哪种操作取决于其选项：

-l [-L | -a] [*string*]

列出所有现有的用户自定义小部件。 如果使用 -L 选项，则以 zle 命令的形式列出创建小部件的命令。

与 -a 选项结合使用时，将列出所有小部件名称，包括内置命令名称。在这种情况下，-L 选项将被忽略。

如果至少给出一个 *string*，且 -a 已存在或 -L 未使用，则不会打印任何内容。 如果所有 *string* 都是已存在的小部件名称，则返回状态为 0；如果至少有一个 *string* 不是已定义的小部件名称，则返回状态为非 0。 如果 -a 也存在，则所有小部件名称都将用于比较，包括内置小部件，否则只使用用户定义的小部件。

如果存在至少一个 *string*，且使用了 -L 选项，则会以 zle 命令的形式列出与任何 *string* 匹配的用户自定义部件创建的命令。

-D *widget ...*

删除已命名的 *widget*s。

-A *old-widget new-widget*

将 *new-widget* 作为 *old-widget* 的别名，这样两个名称都指向同一个小部件。 这两个名称具有同等地位；如果其中一个被删除，另一个将保留。 如果已有一个名称为 *new-widget* 的小部件，它将被删除。

-N *widget* [*function*]

创建一个用户定义的小部件。 如果已经有一个名称为指定名称的小部件，它将被覆盖。 在编辑器中调用新的小部件时，会调用指定的 shell *function*。 如果没有指定函数名称，则默认使用与小部件相同的名称。 更多信息，请参阅 Zle 小部件。

`-f` *flag* [*flag*...]

为运行中的小部件设置各种标志。 *flag* 的可能值是：

yank 用于指示小部件已将文本插入缓冲区。 如果小部件正在封装已有的内部部件，则无需进一步操作，但如果是手动插入文本，则还应注意正确设置 YANK_START 和 YANK_END。 yankbefore 的作用与此相同，但用于在光标后出现复制的文本。

kill 用于指示文本已被杀进入剪切缓冲区。 当重复调用 kill 小部件时，文本会被追加到剪切缓冲区而不是替换掉，但在封装此类小部件时，有必要调用 'zle -f kill' 来保留这种效果。

vichange，用于表示小部件代表的 vi 变化可以用 'vi-repeat-change' 整体重复。在检查 NUMERIC 的值或调用其他小部件之前，应在函数的早期设置该标志。对于从插入模式调用的小部件，该标志不起作用。如果小部件结束时插入模式处于激活状态，则更改会持续到下一次返回命令模式。

`-C` *widget completion-widget function*

创建名为 *widget* 的用户自定义补全部件。该补全小部件的行为与内置的补全小部件类似，其名称为 *completion-widget*。要生成补全，将调用 shell 函数 *function*。 更多信息，请参阅 补全小部件。

`-R` [`-c`] [*display-string*] [*string* ...]

重新显示命令行。 如果给定了 *display-string*，且该字符串不为空，则会显示在状态行中（紧挨着正在编辑的行的下方）。

如果给出了可选的 *string*s，它们将以与打印补全列表相同的方式列在提示符下方。如果没有给出 *string*s，但使用了 `-c` 选项，则会清除该列表。

请注意，运行小部件返回后，命令行会立即重新显示，显示的字符串也会被擦除。 因此，该选项只适用于使用后不会立即退出的小部件。

这条命令可以在用户定义的小部件之外安全地调用；如果 zle 处于活动状态，显示内容将被刷新，而如果 zle 未处于活动状态，这条命令则没有任何作用。在这种情况下，通常不会有其他参数。

如果 zle 处于活动状态，则状态为 0，否则为 1。

`-M` *string*

与 `-R` 选项一样，*string* 将显示在命令行下方；与 `-R` 选项不同的是，字符串不会被放入状态行，而是正常打印在提示符下方。 这意味着小部件返回后，*string* 仍会显示（直到被后续命令覆盖）。

`-U` *string*

这会将 *string* 中的字符推入 ZLE 的输入堆栈。 在当前执行的小部件结束后，ZLE 的行为就好像 *string* 中的字符是由用户输入的一样。

由于 ZLE 使用堆栈，如果重复使用该选项，最后推入堆栈的字符串将首先被处理。 不过，每个 *string* 中的字符将按照它们在字符串中出现的顺序进行处理。

`-K` *keymap*

选择名为 *keymap* 的键映射。 如果没有此键映射，系统将显示错误信息。

键映射的选择会影响 ZLE 在本次调用中对后续按键的解释。 任何后续调用（例如下一条命令行）都将照常以 'main' 键映射开始。

`-F [ -L | -w ] [` *fd* `[` *handler* `] ]`

只有当系统支持 'poll' 或 'select' 系统调用时才可用；大多数现代系统都支持。

安装 *handler*（一个 shell 函数的名称），以处理来自文件描述符 *fd* 的输入。为已处理过的 *fd* 安装处理程序会导致现有处理程序被替换。 可以为任意数量的可读文件描述符安装任意数量的处理程序。 请注意，在安装处理程序时，zle 不会检查 *fd* 是否可读。 用户必须自行安排在zle未激活时如何处理文件描述符。

当 zle 尝试读取数据时，它会同时检查终端和已处理 *fd* 的列表。 如果已处理 *fd* 上的数据可用，zle 会调用 *handler*，并将已准备好读取的 fd 作为第一个参数。 正常情况下，这是唯一的参数，但如果检测到错误，第二个参数会提供详细信息：'hup' 表示断开连接，'nval' 表示描述符已关闭或无效，'err' 表示任何其他情况。 只支持 'select' 系统调用的系统总是使用 'err'。

如果同时给定了 `-w` 选项，*handler* 则是一个行编辑器小部件，通常是使用 'zle -N' 制作成小部件的 shell 函数。 在这种情况下，*handler* 可以使用 zle 的所有功能来更新当前编辑行。 但需要注意的是，由于 *fd* 的处理是在低层次上进行的，因此显示的更改不会自动出现；小部件应调用 'zle -R' 来强制重新显示。 截至本文撰写时，小部件处理程序只支持单个参数，因此不会传递错误状态字符串，所以小部件必须准备好自行测试描述符。

如果处理程序向终端输出内容，则应在输出前调用 'zle -I'（见下文）。 处理程序不应试图从终端读取数据。

如果未给出 *handler*，但存在 *fd*，则会删除该 *fd* 的任何处理程序。 如果没有处理程序，将打印错误信息并返回状态 1。

如果没有给定参数，或者提供了 -L 选项，则会以可存储的形式打印处理程序列表，以供之后执行。

可选择使用 -L 选项给出 *fd*（但不包括 *handler*）；在这种情况下，如果有处理程序，函数将列出该处理程序，否则静默返回状态 1。

请注意，使用该特性时应谨慎。 如果 *fd* 上的某个活动未得到正确处理，可能导致终端无法使用。 从信号陷阱中移除 *fd* 处理程序可能会导致不可预知的行为。

下面是一个使用此功能的简单示例。 使用 ztcp 命令创建与远程 TCP 端口的连接；参见 zsh/net/tcp 模块。 然后安装一个处理程序，简单地打印出该连接上到达的任何数据。 需要注意的是，如果远端关闭了连接，'select' 将指示文件描述符需要处理；我们通过测试读取失败来处理这种情况。

```
if ztcp pwspc 2811; then
  tcpfd=$REPLY
  handler() {
    zle -I
    local line
    if ! read -r line <&$1; then
      # select marks this fd if we reach EOF,
      # so handle this specially.
      print "[Read on fd $1 failed, removing.]" >&2
      zle -F $1
      return 1
    fi
    print -r - $line
  }
  zle -F $tcpfd handler
fi
```

-I

不同的是，这个选项在普通小部件函数之外最有用，不过如果需要向终端正常输出，也可以在内部使用。 它可以使当前的 zle 显示无效，以为输出做准备；通常是通过陷阱函数输出。 如果zle未激活，它将不起作用。 当陷阱退出时，shell 会检查显示是否需要恢复，因此下面，将以不影响正在编辑的行的方式打印输出：

```
TRAPUSR1() {
  # Invalidate zle display
  [[ -o zle ]] && zle -I
```

```
        # Show output
        print Hello
    }
```

一般来说，在使用此方法之前，陷阱函数可能需要测试 zle 是否处于活动状态（如示例所示），因为 zsh/zle 模块可能根本没有加载；如果没有加载，则可以跳过该命令。

在控制权返回编辑器之前，可以多次调用 'zle -I'；为减少干扰，显示只会在第一次无效。

请注意，在 zle 小部件中通常有更好的操作显示的方式；例如，请参阅上文的 'zle -R'。

如果 zle 已失效，则返回状态为零，尽管这可能是由于之前调用了 'zle -I' 或系统通知所致。 要测试此时是否可以调用 zle 小部件，可以不带参数执行 zle，并检查返回状态。

-T

用于添加、列出或删除行编辑器处理过程中的内部转换。 它通常只用于调试或测试，因此一般用户对其兴趣不大。

'zle -T *transformation func*' 指定指定的 *transformation*（见下文）由 shell 函数 *func* 执行（effected）。

'zle -Tr *transformation*' 会移除给定的 *transformation*（如果有的话，如果没有也不会出错）。

'zle -TL' 可用于列出当前正在运行的所有转换。

目前唯一的转换是 tc。 它用来代替向终端输出 termcap 代码。 当转换执行时，shell 函数会将输出的 termcap 代码作为第一个参数传递给函数；如果转换操作需要数字参数，则会将数字参数作为第二个参数传递给函数。 函数应将 shell 变量 REPLY 设置为转换后的 termcap 代码。 通常情况下，这用于生成一些简单格式化的代码版本，以及用于调试或测试的可选参数。 请注意，这种转换不适用于其他非打印字符，如回车符和换行符。

widget [-n *num*] [-f *flag*] [-Nw] [-K *keymap*] *args ...*

调用指定的 *widget*。 只有在 ZLE 处于活动状态时才能调用；通常是在用户自定义的小部件中调用。

使用选项 -n 和 -N，当前数字参数将被保存，并在调用 *widget* 后恢复；'-n *num*' 会将数字参数暂时设置为 *num*，而 '-N' 则会将其设置为默认值，即相当于没有 *num*。

如果使用选项 -K，*keymap* 将在小部件执行期间用作当前键映射。 退出小部件后，将恢复之前的键映射。

通常情况下，以这种方式调用小部件不会设置特殊参数 WIDGET 和相关参数，因此环境看起来就好像用户调用的顶层小部件仍然处于活动状态。 使用 -w 选项后，WIDGET 和相关参数将被设置为反映 zle 调用正在执行的小部件。

通常，当 *widget* 返回时，特殊参数 LASTWIDGET 将指向它。 可以通过选项 -f nolast 来抑制这种情况。

任何进一步的参数都将传递给小部件；注意，由于执行的是标准参数处理，任何一般参数列表都应在前面加上 --。 如果是 shell 函数，这些参数将作为位置参数传递下去；对于内置小部件，则由相关小部件自行决定如何处理这些参数。 目前，只有增量搜索命令，history-search-forward 和 -backward 以及 vi- 前缀的相应函数和 universal-argument 可以处理参数。 如果命令未使用参数，或仅使用了部分参数，则不会标志错误。

返回状态反映了小部件执行操作的成功或失败，如果是用户定义的小部件，则反映 shell 函数的返回状态。

除非未设置 BEEP 选项或通过 zle 命令调用小部件，否则非零返回状态会导致 shell 在小部件退出时发出蜂鸣声。 因此，如果用户定义的小部件需要立即发出蜂鸣声，应直接调用 beep 小部件。

18.4 Zle 小部件

编辑器中的所有操作都由‘widgets’ 执行。 小部件的工作就是执行一些小的操作。 键映射中的键序列所绑定的 ZLE 命令实际上就是小部件。 小部件可以是用户定义的，也可以是内置的。

标准小部件 中列出了 ZLE 内置的标准小部件。 其他内置部件可由其他模块定义（参见 Zsh 模块）。 每个内置小部件都有两个名称：一个是正常的规范名称，另一个是前缀为 ‘.’ 的名称。 ‘.’ 名称比较特殊：它不能重新绑定到其他部件上。 这样，即使重新定义了部件的通常名称，该部件仍可使用。

用户自定义的小部件使用 ‘zle -N’ 定义，并以 shell 函数的形式实现。 当小部件被执行时，相应的 shell 函数也会被执行，并可执行编辑（或其他）操作。 建议用户自定义小部件的名称不要以 ‘.’ 开头。

18.5 用户定义小部件

用户自定义的小部件作为 shell 函数实现，可以执行任何普通的 shell 命令。 它们还可以使用 zle 内置命令运行其他小部件（无论是内置的还是用户自定义的）。 函数的标准输

入是从 /dev/null 重定向的，以防止外部命令通过从终端读取数据而无意中阻塞 ZLE，但 `read -k` 或 `read -q` 也可用于读取字符。 最后，它们可以通过读取和设置下文所述的特殊参数，检查和编辑正在编辑的 ZLE 缓冲区。

这些特殊参数在小部件函数中始终可用，但在 ZLE 外部却没有任何特殊性。 如果它们在 ZLE 外部有一些正常值，那么该值暂时无法访问，但在小部件函数退出时会返回。 事实上，这些特殊参数具有本地作用域，就像在函数中使用 `local` 创建的参数一样。

在 ZLE 处于活动状态时调用的补全小部件和陷阱内，这些参数以只读方式提供。

请注意，这些参数在任何 ZLE 小部件中都是本地参数。 因此，如果要覆盖这些参数，需要在嵌套函数中进行：

```
widget-function() {
  # $WIDGET here refers to the special variable
  # that is local inside widget-function
  () {
     # This anonymous nested function allows WIDGET
     # to be used as a local variable.  The -h
     # removes the special status of the variable.
     local -h WIDGET
  }
}
```

BUFFER (scalar)

编辑缓冲区的全部内容。 如果被写入，光标将保持在相同的偏移量，除非这样会将光标置于缓冲区之外。

BUFFERLINES (integer)

当前屏幕上显示的编辑缓冲区所需的屏幕行数（即上次重新显示后未对前面的参数进行任何更改）；只读。

CONTEXT (scalar)

调用 zle 读取一行的上下文；只读。 值之一：

start

命令行的起始位置（提示符 PS1）。

cont

命令行的续行（提示符 PS2）。

select

在 select 循环中（提示符 PS3）。

vared

编辑 vared 中的变量。

CURSOR (integer)

光标在编辑缓冲区内的偏移量。 范围从 0 到 $#BUFFER，根据定义等于 $#LBUFFER。 如果试图将光标移出缓冲区，光标将被移至缓冲区的适当末端。

CUTBUFFER (scalar)

使用 'kill-' 命令之一剪切的最后一个条目；下一次拖拽(yank)将插入该行的字符串。 删除环（kill ring）中的后续条目位于 killring 数组中。 请注意，命令 'zle copy-region-as-kill *string*' 命令可用于从 shell 函数中设置剪切缓冲区的文本，并以与交互式删除文本相同的方式循环 kill ring。

HISTNO (integer)

当前历史记录编号。 设置它与在历史记录中向上或向下移动到相应的历史记录行具有相同的效果。 如果历史记录中没有该行，则设置该参数的尝试将被忽略。 请注意，这与参数 HISTCMD 并不相同，后者给出的是被添加到主 shell 历史记录中的历史行的编号。 HISTNO 指的是在 zle 中检索的行。

ISEARCHMATCH_ACTIVE (integer)
ISEARCHMATCH_START (integer)
ISEARCHMATCH_END (integer)

ISEARCHMATCH_ACTIVE 表示 BUFFER 中的某个部分当前是否与增量搜索模式相匹配。ISEARCHMATCH_START 和 ISEARCHMATCH_END 显示匹配部分的位置，单位与 CURSOR 相同。只有当 ISEARCHMATCH_ACTIVE 非零时，读取才有效。

所有参数均为只读参数。

KEYMAP (scalar)

当前选中的键映射的名称；只读。

KEYS (scalar)

调用该小部件时键入的字面字符串；只读。

KEYS_QUEUED_COUNT (integer)

被推送回输入队列的字节数，因此在完成任何 I/O 操作之前可以立即读取；只读。另请参阅 PENDING；这两个值是不同的。

killring (array)

之前被删除的条目的数组，最近被删除的条目在前。 yank-pop 会以相同的顺序取出被删除的条目。 但请注意，最近被删除的条目位于 $CUTBUFFER 中；$killring 显示的是之前的条目数组。

删除环的默认大小为 8，但可以通过正常的数组操作改变长度。 删除环中的任何空字符串都会被 yank-pop 命令忽略，因此数组的大小实际上设定了删除环的最大长度，而非零字符串的数量则表示当前的长度，两者都是用户在命令行中看到的长度。

LASTABORTEDSEARCH (scalar)

用户放弃的交互式搜索所使用的最后一个搜索字符串（搜索小部件返回的状态 3）。

LASTSEARCH (scalar)

交互式搜索使用的最后一个搜索字符串；只读。 即使搜索失败（搜索小部件返回的状态为 0、1 或 2）也会设置该字符串，但如果搜索被用户中止，则不会设置该字符串。

LASTWIDGET (scalar)

最后执行的小部件的名称；只读。

LBUFFER (scalar)

位于光标位置左边的缓冲区部分。 如果指定了该部分，则只有该部分缓冲区会被替换，光标将保留在新的 $LBUFFER 和旧的 $RBUFFER 之间。

MARK (integer)

与 CURSOR类似，但用于标记。对于等待移动命令来选择文本区域的 vi 模式操作符，设置 MARK 可以使选择范围从初始光标位置向两个方向扩展。

NUMERIC (integer)

数字参数。如果没有给出数字参数，则不设置该参数。在小部件函数中设置该参数后，使用 zle 内置命令调用的内置小部件将使用指定的数值。如果未在 widget 函数中设置该参数，则调用的内置小部件将视作未给定数字参数。

PENDING (integer)

等待输入的字节数，即已输入并可立即读取的字节数。在 shell 无法获取此信息的系统中，此参数的值始终为 0。 只读。 另请参见 KEYS_QUEUED_COUNT；这两个值是不同的。

PREBUFFER (scalar)

在二级提示符下的多行输入中，该只读参数包含光标当前所在行之前各行的内容。

PREDISPLAY (scalar)

在可编辑文本缓冲区开始前显示的文本。 这不一定是完整的一行；要显示完整的一行，必须明确添加换行符。 每次调用（但不是递归调用）zle 时，文本都会重置。

POSTDISPLAY (scalar)

在可编辑文本缓冲区结束后显示的文本。 这不一定是一行完整的文字；要显示一行完整的文字，必须明确预置换行符。 每次调用（但不是递归调用）zle 时，文本都会重置。

RBUFFER (scalar)

位于光标位置右侧的缓冲区部分。 如果赋值给该部分，则只有这部分缓冲区会被替换，光标将保留在旧的 $LBUFFER 和新的 $RBUFFER 之间。

REGION_ACTIVE (integer)

指示区域当前是否处于激活状态。 可以赋值 0 或 1 以分别停用和激活该区域。如果值为 2，则该区域将以整行模式激活，高亮显示的文本仅延伸至整行；请参阅 字符高亮。

region_highlight (array)

该数组的每个元素都可以设置为一个字符串，用于描述命令行任意区域的高亮效果，该效果将在下一次重新显示命令行时生效。 可以在 PREDISPLAY 和 POSTDISPLAY 中突出显示命令行中不可编辑的部分，但要注意，字符索引需要使用 P 标志才能包含 PREDISPLAY。

每个字符串由以下以空白分隔的部分组成：

- 可选择使用 'P'，表示后面的起始偏移和结束偏移包括 PREDISPLAY 特殊参数设置的任何字符串；如果要高亮显示预显示字符串本身，则需要这样做。 'P' 和起始偏移量之间的空格是可选的。
- 与 CURSOR 单位相同的起始偏移量。
- 与 CURSOR 单位相同的终点偏移量。
- 高亮规范，格式与参数 zle_highlight 中用于上下文的格式相同，参见 字符高亮；例如 standout 或 fg=red,bold。
- 可选择使用 'memo=*token*' 形式的字符串。 *token* 包含从 '=' 到下一个空格、逗号、NUL 或字符串结尾之间的所有内容。 *token* 将被逐字保留，但不会进行任何解析。

  插件可以用它来识别自己添加的数组元素：例如，插件可以将 *token* 设置为自己（插件）的名称，然后使用 'region_highlight=(${region_highlight:#*memo=*token*})' 来删除自己添加的数组元素。

(本例使用了 参数扩展中描述的 '${*name*:#*pattern*}' 数组-grepping 语法）。

例如,

```
region_highlight=("P0 20 bold memo=foobar")
```

指定文本的前 20 个字符（包括任何预显示字符串）以粗体高亮显示。

请注意，region_highlight 的效果不会保存，一旦接受行，它就会消失。

请注意，zsh 5.8 及更早版本不支持 'memo=*token*' 字段，并且在给出备忘录（memo）时可能会误读第三个字段（高亮说明）。

命令行上的最终高亮效果取决于 region_highlight 和 zle_highlight；详情请参见 字符高亮。

registers (associative array)

每个 vi 寄存器缓冲区的内容。通常使用 vi-set-buffer，然后使用删除、更改或复制（yank）命令来设置这些内容。

SUFFIX_ACTIVE (integer)
SUFFIX_START (integer)
SUFFIX_END (integer)

SUFFIX_ACTIVE 表示自动删除的补全后缀当前是否处于激活状态。
SUFFIX_START 和 SUFFIX_END 表示后缀的位置，单位与 CURSOR 相同。只有当 SUFFIX_ACTIVE 非零时，才能读取它们。

所有参数均为只读参数。

UNDO_CHANGE_NO (integer)

代表撤销历史状态的数字。 它的唯一用途是作为参数传递给 undo 小部件，以撤销到记录的点。 只读。

UNDO_LIMIT_NO (integer)

与撤消历史中现有更改相对应的数字；比较 UNDO_CHANGE_NO。 如果设置的值大于零，undo 命令将不允许撤销超出给定更改次数的行。 但仍可以使用 'zle undo *change*' 在小部件中撤销超出该点的操作；在这种情况下，直到 UNDO_LIMIT_NO 被减少，才会完全无法撤销。 设置为 0 则禁用限制。

在小部件函数中该变量的典型用法如下（注意需要附加函数作用域）：

```
() {
  local UNDO_LIMIT_NO=$UNDO_CHANGE_NO
```

```
        # Perform some form of recursive edit.
      }
```

WIDGET (scalar)

当前正在执行的小部件的名称；只读。

WIDGETFUNC (scalar)

实现用 zle -N 或 zle -C 定义的小部件的 shell 函数名称。 在前一种情况下，这是定义小部件的 zle -N 命令的第二个参数，如果没有第二个参数，则是第一个参数。 在后一种情况下，这是定义小部件的 zle -C 命令的第三个参数。 只读。

WIDGETSTYLE (scalar)

描述当前正在执行的补全小部件背后的实现；是定义小部件时 zle -C 后面的第二个参数。 这是内置补全小部件的名称。 对于使用 zle -N 定义的小部件，该参数将被设置为空字符串。 只读。

YANK_ACTIVE (integer)
YANK_START (integer)
YANK_END (integer)

YANK_ACTIVE 表示文本是否刚刚被插入（粘贴）到缓冲区。 YANK_START 和 YANK_END 表示粘贴文本的位置，单位与 CURSOR 相同。 只有当 YANK_ACTIVE 非零时，它们才能读取。 以类似 yank-like 的方式插入文本的小部件（例如 bracketed-paste 的封装）也可以分配它们。 另请参见 zle -f。

YANK_ACTIVE 是只读的。

ZLE_RECURSIVE (integer)

通常为零，但在 recursive-edit 的任何实例中都会递增。 因此，它表示当前的递归级别。

ZLE_RECURSIVE 是只读的。

ZLE_STATE (scalar)

包含描述当前 zle 状态的一组以空格分隔的单词。

目前，显示的状态是由 overwrite-mode 或 vi-replace 小部件设置的插入模式，以及由 set-local-history 小部件控制的历史命令是否会访问已导入的条目。如果要在命令行中插入的字符会将现有字符移到右边，则字符串包含 'insert'；如果要插入的字符会覆盖现有字符，则字符串包含 'overwrite'。如果只访问本地历史命令，则字符串中包含 'localhistory'；如果也访问导入的历史命令，则字符串中包含 'globalhistory'。

子字符串按字母顺序排序，因此，如果您想以一种面向未来的方式测试两个特定的子字符串，就可以通过下面进行匹配：

```
if [[ $ZLE_STATE == *globalhistory*insert* ]]; then ...; fi
```

18.5.1 特殊小部件

有一些用户定义的小部件是 shell 特有的。 如果它们不存在，则不会采取任何特殊行为。所提供的环境与其他编辑小部件相同。

`zle-isearch-exit`

在增量搜索结束时，即从显示屏上删除 isearch 提示符时执行。 有关示例，请参阅 `zle-isearch-update`。

`zle-isearch-update`

当显示屏即将重绘时，在增量搜索中执行。 通过在小部件中使用 '`zle -M`'，可以生成增量搜索提示符下方的附加输出。 例如

```
zle-isearch-update() { zle -M "Line $HISTNO"; }
zle -N zle-isearch-update
```

请注意，在退出增量搜索时，'`zle -M`' 输出的行不会被删除。 这可以通过 `zle-isearch-exit` 小部件来完成：

```
zle-isearch-exit() { zle -M ""; }
zle -N zle-isearch-exit
```

`zle-line-pre-redraw`

每当输入行即将重绘时执行，为更新 region_highlight 数组提供机会。

`zle-line-init`

每次启动行编辑器读取新的输入行时执行。 下面的示例会在行编辑器启动时使其进入 vi 命令模式。

```
zle-line-init() { zle -K vicmd; }
zle -N zle-line-init
```

(函数内部的命令直接设置键映射；相当于 `zle vi-cmd-mode`）。

`zle-line-finish`

与 `zle-line-init`类似，但每次行编辑器读完一行输入时都会执行。

`zle-history-line-set`

当历史行发生变化时执行。

`zle-keymap-select`

每当键映射发生变化，即特殊参数 KEYMAP 被设置为不同值时，且行编辑器活动时，都会执行调用。 行编辑器启动时初始化键映射不会导致小部件被调用。

函数中的 $KEYMAP 值反映了新的键映射。 旧的键映射作为唯一参数传递。

可用于检测 vi 命令（`vicmd`）和插入（通常为 `main`）键映射之间的切换。

18.6 标准小部件

下面列出了所有标准小部件，以及它们在 emacs 模式、vi 命令模式和 vi 插入模式下的默认绑定（分别为‘`emacs`’, ‘`vicmd`’ 和 ‘`viins`’ 键映射）。

请注意，光标键在所有三个键映射中都与移动键绑定；shell 假定光标键发送终端处理库（termcap 或 terminfo）报告的键序。 列表中显示的键序是基于 VT100 的键序，在许多现代终端上都很常见，但实际上这些键序并不一定是绑定的。 在 `viins` 键映射中，序列的初始转义字符也用于返回 `vicmd` 键盘映射：这是否会发生由 KEYTIMEOUT 参数决定，参见 参数。

18.6.1 移动

`vi-backward-blank-word` (unbound) (B) (unbound)

向后移动一个单词，单词的定义是一系列非空白字符。

`vi-backward-blank-word-end` (unbound) (gE) (unbound)

移动到上一个单词的末尾，单词的定义是一系列非空白字符。

`backward-char` (^B ESC-[D) (unbound) (unbound)

反向移动一个字符

`vi-backward-char` (unbound) (^H h ^?) (ESC-[D)

反向移动一个字符，不改变行

`backward-word` (ESC-B ESC-b) (unbound) (unbound)

移动到前一个词的开始。

`emacs-backward-word`

移动到前一个词的开始。

`vi-backward-word` (unbound) (b) (unbound)

移动到前一个词的开始, vi 风格.

`vi-backward-word-end` (unbound) (ge) (unbound)

vi-style 方式，移至前一个单词的末尾。

`beginning-of-line` (^A) (unbound) (unbound)

移至行首。 如果已在行首，则移至上一行（如果有）的行首。

`vi-beginning-of-line`

移动到行首，但不换行。

`down-line` (unbound) (unbound) (unbound)

在缓冲区中向下移动一行。

`end-of-line` (^E) (unbound) (unbound)

移至行尾。 如果已在行尾，则移至下一行（如果有）的行尾。

`vi-end-of-line` (unbound) ($) (unbound)

移至行尾。 如果该命令有一个参数，光标将被移到该行的末尾（参数 - 1）行。

`vi-forward-blank-word` (unbound) (W) (unbound)

前进一个单词，单词的定义是一系列非空白字符。

`vi-forward-blank-word-end` (unbound) (E) (unbound)

移动到当前单词的末尾，如果在当前单词的末尾，则移动到下一个单词的末尾，这里单词定义为一系列非空白字符。

`forward-char` (^F ESC-[C) (unbound) (unbound)

向前移动一个字符

`vi-forward-char` (unbound) (space l) (ESC-[C)

向前移动一个字符

`vi-find-next-char` (^X^F) (f) (unbound)

从键盘上读取一个字符，并移动到该行中出现该字符的下一个位置。

vi-find-next-char-skip (unbound) (t) (unbound)

从键盘上读取一个字符，并移动到该行中下一个该字符出现之前的位置。

vi-find-prev-char (unbound) (F) (unbound)

从键盘上读取一个字符，并移动到该行中出现该字符的前一个位置。

vi-find-prev-char-skip (unbound) (T) (unbound)

从键盘上读取一个字符，并移动到该行中前一个该字符之后的位置。

vi-first-non-blank (unbound) (^) (unbound)

移动到该行第一个非空白字符。

vi-forward-word (unbound) (w) (unbound)

前进一个字，vi-style。

forward-word (ESC-F ESC-f) (unbound) (unbound)

移动到下一个单词的开头。 编辑器对单词的概念由 WORDCHARS 参数指定。

emacs-forward-word

移动到下一个单词的末尾。

vi-forward-word-end (unbound) (e) (unbound)

移动到下一个单词的末尾。

vi-goto-column (ESC-|) (|) (unbound)

移动到数字参数指定的列。

vi-goto-mark (unbound) (`) (unbound)

移动到指定标记。

vi-goto-mark-line (unbound) (') (unbound)

移动到包含指定标记的行的起始位置。

vi-repeat-find (unbound) (;) (unbound)

重复最后一条 vi-find 命令。

vi-rev-repeat-find (unbound) (,) (unbound)

朝相反方向重复上一条 vi-find 命令。

up-line (unbound) (unbound) (unbound)

在缓冲区中向上移动一行。

18.6.2 历史控制

beginning-of-buffer-or-history (ESC-<) (gg) (unbound)

移动到缓冲区的起始位置，如果已经在那个位置，则移动到历史列表中的第一个事件。

beginning-of-line-hist

移至该行开头。 如果已在缓冲区的开头，则移至上一历史行。

beginning-of-history

移动到历史列表中的第一个事件。

down-line-or-history (^N ESC-[B) (j) (ESC-[B)

在缓冲区中向下移动一行，如果已在底行，则移动到历史列表中的下一个事件。

vi-down-line-or-history (unbound) (+) (unbound)

在缓冲区中向下移动一行，如果已在底行，则移动到历史列表中的下一个事件。 然后移动到该行第一个非空白字符处。

down-line-or-search

在缓冲区中向下移动一行，如果已在最下面一行，则在历史记录中向前搜索以缓冲区中第一个单词开始的一行。

如果使用带有参数的 zle 命令从函数中调用，第一个参数将作为要搜索的字符串，而不是缓冲区中的第一个单词。

down-history (unbound) (^N) (unbound)

移动到历史列表中的下一个事件。

history-beginning-search-backward

在历史记录中向后搜索从当前行开始到光标处的一行。 这样光标就会停留在原来的位置。

end-of-buffer-or-history (ESC->) (unbound) (unbound)

移动到缓冲区的末尾，或者如果已经移动到末尾，则移动到历史列表中的最后一个事件。

end-of-line-hist

移至行尾。 如果已到缓冲区末尾，则移至下一个历史行。

end-of-history

移动到历史列表中的最后一个事件。

vi-fetch-history (unbound) (G) (unbound)

获取数字参数指定的历史行。 默认为当前历史行（即尚未成为历史的一行）。

history-incremental-search-backward (^R ^Xr) (unbound) (unbound)

向后递增搜索指定字符串。 如果搜索字符串中没有大写字母，也没有给出数字参数，则搜索不区分大小写。 字符串可以以 '^' 开头，以便将搜索锚定在行首。 从用户自定义函数调用时，将返回以下状态： 0，如果搜索成功；1，如果搜索失败；2，如果搜索项是错误的模式；3，如果搜索被 send-break 命令中止。

在迷你缓冲区中可以使用一组受限的编辑功能。 键将在特殊的 isearch 键映射中查找，如果找不到，则在主键映射中查找（注意 isearch 键映射默认为空）。 由 stty 设置定义的中断信号将停止搜索并返回原始行。 未定义的按键也会产生同样的效果。 请注意，以下功能在增量搜索中始终执行相同的任务，不能由用户定义的小部件代替，也不能扩展函数集。 支持的函数有

accept-and-hold
accept-and-infer-next-history
accept-line
accept-line-and-down-history

退出增量搜索后执行常规函数。 显示的命令行将被执行。

backward-delete-char
vi-backward-delete-char

备份搜索历史中的一个位置。 如果重复搜索，则不会立即删除迷你缓冲区中的字符。

accept-search

退出增量搜索，保留命令行但不再执行其他操作。 请注意，该功能默认情况下未绑定，在增量搜索之外没有任何作用。

backward-delete-word
backward-kill-word

vi-backward-kill-word

备份 minibuffer 中的一个字符；如果在插入该字符后进行了多次搜索，则搜索历史将倒退到输入该字符之前的位置。 因此，这具有重复 backward-delete-char 的效果。

clear-screen

清空屏幕，保持增量搜索模式。

history-incremental-search-backward

查找迷你缓冲区内容的下一次出现。如果迷你缓冲区为空，则恢复最近一次使用的搜索字符串。

history-incremental-search-forward

颠倒搜索的意义。

magic-space

插入一个非魔法空格。

quoted-insert
vi-quoted-insert

为要插入 minibuffer 的字符加引号。

redisplay

重新显示命令行，保持增量搜索模式。

vi-cmd-mode

选择 'vicmd' 键映射；初始时将选择 'main' 键映射（插入模式）。

此外，在 vi 插入模式下所做的修改会合并成一个撤销事件。

vi-repeat-search
vi-rev-repeat-search

重复搜索。 迷你缓冲区会显示搜索方向。

任何未与上述函数、self-insert 或 self-insert-unmeta绑定的字符都会导致模式退出。 然后，该字符将在当时有效的键映射中查找并执行。

当用 zle 命令从小部件函数调用时，增量搜索命令可以接受一个字符串参数。 与 bindkey 命令的参数一样，字符串将被视为键的字符串，并用作命令的初始输入。 字符串中任何未被增量搜索使用的字符都将被忽略。 例如

```
zle history-incremental-search-backward forceps
```

将向后搜索 forceps，留下包含 'forceps' 字符串的迷你缓冲。

history-incremental-search-forward (^S ^Xs) (unbound) (unbound)

向前递增搜索指定字符串。 如果搜索字符串中没有大写字母，也没有给出数字参数，则搜索不区分大小写。 字符串可以以 '^' 开头，以便将搜索锚定在行首。 迷你缓冲区中可用的函数与 history-incremental-search-backward 相同。

history-incremental-pattern-search-backward
history-incremental-pattern-search-forward

这些小部件的行为与不带 -pattern 的相应小部件类似，但用户输入的搜索字符串会被视为一种模式，并尊重影响模式匹配的各种选项的当前设置。 有关模式的描述，请参阅 文件名生成。 如果没有给出数字参数，搜索字符串中的小写字母可能与历史记录中的大写字母匹配。 字符串可以以 '^' 开头，以便将搜索锚定在行首。

提示符改变以指示无效模式；这可能只是表示模式尚未完成。

请注意，只报告不重叠的匹配结果，因此带有通配符的表达式在一行中返回的匹配结果可能少于通过检查可以看到的结果。

history-search-backward (ESC-P ESC-p) (unbound) (unbound)

在历史记录中向后搜索以缓冲区中第一个单词开头的行。

如果使用带有参数的 zle 命令从函数中调用，第一个参数将作为要搜索的字符串，而不是缓冲区中的第一个单词。

vi-history-search-backward (unbound) (/) (unbound)

在历史记录中向后搜索指定字符串。 字符串可以以 '^' 开头，以便搜索锚定到该行的开头。

在迷你缓冲区中可以使用一套受限的编辑功能。 由 stty 设置定义的中断信号将停止搜索。 迷你缓冲区中可用的函数有 accept-line, backward-delete-char, vi-backward-delete-char, backward-kill-word, vi-backward-kill-word, clear-screen, redisplay, quoted-insert 和 vi-quoted-insert。

vi-cmd-mode 的处理方式与 accept-line 相同，而 magic-space 则被视为空格。任何其他未绑定到 self-insert 或 self-insert-unmeta 的字符都会发出哔哔声并被忽略。如果在 vi 命令模式下调用该函数，将使用当前插入模式的绑定。

如果使用带有参数的 zle 命令从函数中调用，第一个参数将作为要搜索的字符串，而不是缓冲区中的第一个单词。

history-search-forward (ESC-N ESC-n) (unbound) (unbound)

在历史记录中向前搜索以缓冲区中第一个单词开头的行。

如果使用带有参数的 zle 命令从函数中调用，第一个参数将作为要搜索的字符串，而不是缓冲区中的第一个单词。

vi-history-search-forward (unbound) (?) (unbound)

在历史记录中向前搜索指定字符串。 字符串可以以 '^' 开头，以便将搜索锚定到行的开头。迷你缓冲区中可用的函数与 vi-history-search-backward 相同。 参数处理也与该命令相同。

infer-next-history (^X^N) (unbound) (unbound)

在历史记录列表中搜索与当前行匹配的行，并获取其后的事件。

insert-last-word (ESC-_ ESC-.) (unbound) (unbound)

在光标位置插入上一个历史事件的最后一个单词。 如果参数为正数，则从上一个历史事件的末尾插入该字。 如果参数为零或负数，则从左侧插入该字（零插入前一个命令字）。 重复该命令后，刚才插入的单词将替换为历史事件中在刚才使用的单词之前的最后一个单词； 数字参数也可以同样的方式从历史事件中选取一个单词。

从用户定义的小部件调用 shell 函数时，命令可以包含一到三个参数。 第一个参数指定历史偏移量，该偏移量适用于对该小部件的连续调用：如果为-1，则使用默认行为；如果为1，则连续调用将在历史中向前移动。 如果数值为 0，则表示将重新检查上一次执行命令时检查过的历史行。 需要注意的是，负数前面应加上 '--' 参数，以免与选项混淆。

如果给定两个参数，第二个参数将以普通数组索引符号指定命令行上的字词（作为数字参数更自然的替代）。 因此，1 是第一个字，-1（默认值）是最后一个字。

如果给出第三个参数，其值将被忽略，但它用于表示历史偏移量是相对于当前历史行的，而不是之前调用 insert-last-word 后记住的历史行。

例如，该命令的默认行为相当于

```
zle insert-last-word -- -1 -1
```

而命令

```
zle insert-last-word -- -1 1 -
```

总是复制紧接着被编辑行之前的历史行的第一个单词。 这样做的副作用是，以后调用小部件时将相对于该行。

vi-repeat-search (unbound) (n) (unbound)

重复上次的 vi 历史记录搜索。

vi-rev-repeat-search (unbound) (N) (unbound)

重复上次的 vi 历史记录搜索，但以相反的方向进行。

up-line-or-history (^P ESC-[A) (k) (ESC-[A)

在缓冲区中向上移动一行，或者如果已经在最上面一行，则移动到历史列表中的上一个事件。

vi-up-line-or-history (unbound) (-) (unbound)

在缓冲区中向上移动一行，如果已在顶行，则移动到历史列表中的上一个事件。 然后移动到该行第一个非空白字符处。

up-line-or-search

在缓冲区中向上移动一行，如果已在顶行，则在历史记录中向后搜索以缓冲区中第一个单词开始的一行。

如果使用带有参数的 zle 命令从函数中调用，第一个参数将作为要搜索的字符串，而不是缓冲区中的第一个单词。

up-history (unbound) (^P) (unbound)

移动到历史列表中的上一个事件。

history-beginning-search-forward

在历史记录中向前搜索从当前行开始到光标处的一行。 光标就会停留在原来的位置。

set-local-history

默认情况下，历史记录移动命令会同时访问导入的行和本地行。这个小部件可以让你切换开关，或者用数字参数设置。如果为零，则本地行和导入行都访问；如果不为零，则只访问本地行。

18.6.3 修改文本

vi-add-eol (unbound) (A) (unbound)

移至行尾，进入插入模式。

vi-add-next (unbound) (a) (unbound)

在当前光标位置后进入插入模式，不换行。

backward-delete-char (^H ^?) (unbound) (unbound)

删除光标后面的字符。

vi-backward-delete-char (unbound) (X) (^H)

删除光标后面的字符，不换行。 如果在插入模式下，则不会删除上次进入插入模式的位置。

backward-delete-word

删除光标后面的单词。

backward-kill-line

从行首删除到光标位置。

backward-kill-word (^W ESC-^H ESC-^?) (unbound) (unbound)

删除光标后面的字。

vi-backward-kill-word (unbound) (unbound) (^W)

删除光标后面的字，但不越过上次进入插入模式的位置。

capitalize-word (ESC-C ESC-c) (unbound) (unbound)

将当前单词大写，然后移过去。

vi-change (unbound) (c) (unbound)

从键盘读取移动命令，从光标位置删除到移动终点。 然后进入插入模式。 如果命令是 vi-change，则更改当前行。

为了与 vi 兼容，如果命令为 vi-forward-word 或 vi-forward-blank-word，则单词后的空白不包括在内。如果您更喜欢包含空白的一致行为，请使用以下按键绑定：

```
bindkey -a -s cw dwi
```

vi-change-eol (unbound) (C) (unbound)

删除到行尾，进入插入模式。

vi-change-whole-line (unbound) (S) (unbound)

删除当前行，进入插入模式。

copy-region-as-kill (ESC-W ESC-w) (unbound) (unbound)

将从光标到标记的区域复制到删除缓冲区。

如果从 ZLE 小部件函数中以 ‘`zle copy-region-as-kill` *string*’ 的形式调用，则 *string* 将作为文本复制到删除缓冲区。 在这种情况下，光标、标记和命令行上的文本将不会被使用。

`copy-prev-word` (`ESC-^_`) (unbound) (unbound)

复制光标左侧的单词。

`copy-prev-shell-word`

与 `copy-prev-word`类似，但单词是通过 shell 解析找到的，而 `copy-prev-word` 则是查找空格。当单词带引号并包含空格时，这一点就会有所不同。

`vi-delete` (unbound) (d) (unbound)

从键盘读取移动命令，并从光标位置到移动终点进行删除。 如果命令是 `vi-delete`，则删除当前行。

`delete-char`

删除光标下的字符。

`vi-delete-char` (unbound) (x) (unbound)

删除光标下的字符，但不越过行尾。

`delete-word`

删除当前单词。

`down-case-word` (`ESC-L ESC-l`) (unbound) (unbound)

将当前单词转换为全小写，然后移动过去。

`vi-down-case` (unbound) (gu) (unbound)

从键盘读取移动命令，并将从光标位置到移动终点的所有字符转换为小写。 如果移动命令是 `vi-down-case`，则交换当前行中所有字符的大小写。

`kill-word` (`ESC-D ESC-d`) (unbound) (unbound)

删除当前单词。

`gosmacs-transpose-chars`

交换光标后面的两个字符。

`vi-indent` (unbound) (>) (unbound)

缩进若干行。

`vi-insert` (unbound) (`i`) (unbound)

进入插入模式。

`vi-insert-bol` (unbound) (`I`) (unbound)

移动到该行第一个非空白字符处，进入插入模式。

`vi-join` (`^X^J`) (`J`) (unbound)

将当前行与下一行连接起来。

`kill-line` (`^K`) (unbound) (unbound)

删除从光标到行尾的字符。 如果已在行尾，则删除换行符。

`vi-kill-line` (unbound) (unbound) (`^U`)

从光标处往回删除到上次进入插入模式的位置。

`vi-kill-eol` (unbound) (`D`) (unbound)

从光标删除到行尾。

`kill-region`

从光标删除到标记。

`kill-buffer` (`^X^K`) (unbound) (unbound)

删除整个缓冲区。

`kill-whole-line` (`^U`) (unbound) (unbound)

删除当前行。

`vi-match-bracket` (`^X^B`) (`%`) (unbound)

移动到与光标下的括号相匹配的括号字符（`{}`、`()` 或 `[]` 之一）。 如果光标不在括号字符上，则在不超过行尾的情况下向前移动，找到一个括号字符，然后移至匹配的括号处。

`vi-open-line-above` (unbound) (`O`) (unbound)

在光标上方打开一行，进入插入模式。

`vi-open-line-below` (unbound) (`o`) (unbound)

在光标下方打开一行，进入插入模式。

`vi-oper-swap-case` (unbound) (g~) (unbound)

从键盘读取移动命令，并交换从光标位置到移动终点的所有字符的大小写。 如果移动命令是 `vi-oper-swap-case`，则交换当前行中所有字符的大小写。

`overwrite-mode` (^X^O) (unbound) (unbound)

在覆盖模式和插入模式之间切换。

`vi-put-before` (unbound) (P) (unbound)

在光标之前插入删除缓冲区的内容。 如果删除缓冲区包含一系列行（而不是字符），则将其粘贴到当前行的上方。

`vi-put-after` (unbound) (p) (unbound)

在光标后插入删除缓冲区的内容。 如果删除缓冲区包含一系列行（而不是字符），则将其粘贴到当前行的下方。

`put-replace-selection` (unbound) (unbound) (unbound)

用删除缓冲区的内容替换当前区域或选区的内容。如果删除缓冲区包含一系列行（而不是字符），则当前行将被粘贴的行分割。

`quoted-insert` (^V) (unbound) (unbound)

将输入的下一个字面字符插入缓冲区。 不会插入中断字符。

`vi-quoted-insert` (unbound) (unbound) (^Q ^V)

在光标位置显示 '^'，并将输入的下一个字面字符插入缓冲区。 不会插入中断字符。

`quote-line` (ESC-') (unbound) (unbound)

为当前行加引号，即在行首和行尾添加一个 ''' 字符，并将所有 ''' 字符转换为 ''\'''。

`quote-region` (ESC-") (unbound) (unbound)

为从光标到标记的区域加引号。

`vi-replace` (unbound) (R) (unbound)

进入覆盖模式。

`vi-repeat-change` (unbound) (.) (unbound)

重复上一次 vi 模式文本修改。 如果修改时使用了计数，该计数将被记住。 如果为该命令指定了计数，则该计数将覆盖已记住的计数，并在以后使用该命令时被记住。 同样，剪切缓冲区规范也会类似的被记住。

`vi-replace-chars` (unbound) (r) (unbound)

用从键盘读取的字符替换光标下的字符。

`self-insert` (printable characters) (unbound) (printable characters and some control characters)

在光标位置向缓冲区中插入一个字符。

`self-insert-unmeta` (`ESC-^I ESC-^J ESC-^M`) (unbound) (unbound)

在去除元位并将 ^M 转换为 ^J 后，将字符插入缓冲区。

`vi-substitute` (unbound) (s) (unbound)

替换下一个字符。

`vi-swap-case` (unbound) (~) (unbound)

调换光标下字符的大小写并移动过去。

`transpose-chars` (^T) (unbound) (unbound)

如果在行尾，交换光标左侧的两个字符，否则交换光标下的字符和左侧的字符。

`transpose-words` (`ESC-T ESC-t`) (unbound) (unbound)

将当前单词与前面的单词交换。

如果使用正数参数 ***N***，光标周围的字（如果光标位于字与字之间，则光标后面的字）将与前面的 ***N*** 个字换位。 光标将被置于结果字组的末尾。

如果使用负数参数 ***-N***，效果与使用正数参数 ***N*** 相同，只是无论如何重新排列单词，都会保留原来的光标位置。

`vi-unindent` (unbound) (<) (unbound)

取消若干行的缩进。

`vi-up-case` (unbound) (gU) (unbound)

从键盘读取移动命令，并将从光标位置到移动终点的所有字符转换为小写。 如果移动命令是 `vi-up-case`，则交换当前行中所有字符的大小写。

`up-case-word` (`ESC-U ESC-u`) (unbound) (unbound)

将当前单词转换为全大写字母，然后移动过去。

yank (^Y) (unbound) (unbound)

在光标位置插入删除缓冲区的内容。

yank-pop (ESC-y) (unbound) (unbound)

删除刚刚复制的文本，旋转删除环（之前删除文本的历史记录），然后复制新的顶部。 仅在 yank、vi-put-before、vi-put-after 或 yank-pop 之后起作用。

vi-yank (unbound) (y) (unbound)

从键盘读取移动命令，并将从光标位置到移动终点的区域复制到删除缓冲区。 如果命令是 vi-yank，则复制当前行。

vi-yank-whole-line (unbound) (Y) (unbound)

将当前行复制到删除缓冲区。

vi-yank-eol

将从光标位置到行尾的区域复制到删除缓冲区。 可以说，这就是 Y 在 vi 中应该做的事情，但实际上它并没有这么做。

18.6.4 实参

digit-argument (ESC-0..ESC-9) (1-9) (unbound)

开始一个新的数字参数，或添加到当前参数。 另请参阅 vi-digit-or-beginning-of-line。 只有绑定到以十进制数字结尾的按键序列时才有效。

在小部件函数内部，调用该函数时会将调用小部件的按键序列中的最后一个按键视为数字。

neg-argument (ESC--) (unbound) (unbound)

更改后面参数的符号。

universal-argument

将下一条命令的参数乘以 4。或者，如果该命令后跟一个整数（正数或负数），则将该整数作为下一条命令的参数。 因此，使用这条命令不能重复输入数字。 例如，如果这条命令出现两次，紧接着是 -forward-char，则向前移动 16 个空格；如果紧接着是 -2，然后是 -forward-char，则向后移动 2 个空格。

在小部件函数内部，如果传递一个参数，即 ‘zle universal-argument *num*’，数字参数将被设置为 *num*；这相当于‘NUMERIC=*num*。*num*’，数字参数将被设置为 *num*；这相当于 ‘NUMERIC=*num*’。

argument-base

使用现有的数字参数作为基数，基数范围必须是 2 至 36（含）。 随后使用 digit-argument 和 universal-argument 将以给定的基数输入新的数字参数。 使用的是通常的十六进制约定：字母 a 或 A 对应 10，以此类推。 使用 universal-argument 输入需要 10 位以上数字的基数参数更方便，因为 ESC-a 等通常不会绑定到 digit-argument。

该函数可与用户自定义小部件内的命令参数一起使用。 下面的代码将基数设置为 16，并让用户输入十六进制参数，直到键入超出数字范围的按键为止：

```
zle argument-base 16
zle universal-argument
```

18.6.5 补全

accept-and-menu-complete

在菜单补全中，将当前补全插入缓冲区，并前进到下一个可能的补全。

complete-word

尝试补全当前单词。

delete-char-or-list (^D) (unbound) (unbound)

删除光标下的字符。 如果光标位于行尾，则列出当前单词可能的补全。

expand-cmd-path

将当前命令扩展为其完整路径名。

expand-or-complete (TAB) (unbound) (TAB)

尝试对当前单词进行 shell 扩展。 如果失败，则尝试补全。

expand-or-complete-prefix

尝试对当前单词直到光标前的部分进行 shell 扩展。

expand-history (ESC-space ESC-!) (unbound) (unbound)

在编辑缓冲区上执行历史记录扩展。

expand-word (^X*) (unbound) (unbound)

尝试对当前单词进行 shell 扩展。

list-choices (ESC-^D) (^D =) (^D)

列出当前单词可能的补全。

list-expand (^Xg ^XG) (^G) (^G)

列出当前单词的扩展。

magic-space

执行历史扩展并在缓冲区中插入一个空格。 这预期与空格绑定。

menu-complete

与 complete-word 类似，但使用菜单补全功能。 参见 MENU_COMPLETE 选项。

menu-expand-or-complete

与 expand-or-complete 类似，只是使用菜单补全。

reverse-menu-complete

执行菜单补全，与 menu-complete 类似，但如果菜单补全已在进行中，则移动到**前一个** 补全，而不是下一个。

end-of-list

当之前的补全在提示符下方显示列表时，可使用此小部件将提示符移到列表下方。

18.6.6 杂项

accept-and-hold (ESC-A ESC-a) (unbound) (unbound)

将缓冲区的内容推入缓冲区堆栈并执行。

accept-and-infer-next-history

执行缓冲区的内容。 然后在历史记录列表中搜索与当前事件相匹配的行，并将后面的事件推入缓冲堆栈。

accept-line (^J ^M) (^J ^M) (^J ^M)

完成对缓冲区的编辑。 通常，这会使缓冲区作为 shell 命令执行。

accept-line-and-down-history (^O) (unbound) (unbound)

执行当前行，并将下一个历史事件推入缓冲堆栈。

auto-suffix-remove

如果前一个操作给命令行中的单词添加了后缀（空格、斜线等），则将其删除。 否则什么也不做。 删除后缀会结束任何活动的菜单补全或菜单选择。

该小部件用于从用户自定义的小部件中调用，以强制执行所需的后缀移除行为。

auto-suffix-retain

如果前一个操作为命令行中的单词添加了后缀（空格、斜线等），则强制保留该后缀。 否则什么也不做。 保留后缀会终止任何激活的菜单补全或菜单选择。

该小部件用于从用户自定义的小部件中调用，以强制执行所需的后缀保护行为。

beep

蜂鸣声，除非 BEEP 选项未设置。

bracketed-paste (^[[200~) (^[[200~) (^[[200~)

当文本粘贴到终端模拟器时，会调用该小部件。它不打算与实际按键绑定，而是与粘贴文本时终端模拟器生成的特殊序列绑定。

当以交互方式调用时，粘贴的文本将插入缓冲区并放入剪切缓冲区。 如果给定了数字参数，则会在插入粘贴文本前对其应用 shell 引号。

如果使用 vi-set-buffer ("x) 指定了一个命名缓冲区，粘贴的文本将存储在该命名缓冲区中，但不是插入。

在小部件函数中以 ‘bracketed-paste *name*‘ 调用时，粘贴的文本将赋值给变量 *name*，而不会进行其他处理。

另请参阅 zle_bracketed_paste 参数。

vi-cmd-mode (^X^V) (unbound) (^[)

进入命令模式；即选择 ‘vicmd’ 键映射。 是的，默认情况下在 emacs 模式下是绑定的。

vi-caps-lock-panic

挂起，直到按下任何小写键。 这是为那些没有心理承受能力去追踪大写锁定键的 vi 用户准备的（比如笔者）。

clear-screen (^L ESC-^L) (^L) (^L)

清除屏幕并重新绘制提示符。

deactivate-region

使当前区域处于非激活状态。如果 vim 风格的可视化选择模式处于激活状态，则会禁用该模式。

describe-key-briefly

读取按键序列，然后打印与该序列绑定的函数。

exchange-point-and-mark (^X^X) (unbound) (unbound)

交换光标位置（点）和标记位置。 除非给出负数参数，否则点和标记之间的区域将被激活，以便高亮显示。 如果给定的数字参数为零，该区域将被激活，但点和标记不会交换。

execute-named-cmd (ESC-x) (:) (unbound)

读取编辑器命令的名称并执行。 在解释按键绑定时，用 'zle -A' 为该小部件建别名或用 'zle -N' 替换该小部件没有任何作用，但 'zle execute-named-cmd' 将调用此类别名或替换。

在迷你缓冲区中可以使用一套受限的编辑功能。 键将在特殊的 command 键映射中查找，如果找不到，则在主键映射中查找。 由 stty 设置定义的中断信号将中止该功能。 请注意，以下函数始终在 executed-named-cmd 环境中执行相同的任务，不能被用户定义的小部件替代，也不能扩展函数集。 允许使用的函数包括: backward-delete-char, vi-backward-delete-char, clear-screen, redisplay, quoted-insert, vi-quoted-insert, backward-kill-word, vi-backward-kill-word, kill-whole-line, vi-kill-line, backward-kill-line, list-choices, delete-char-or-list, complete-word, accept-line, expand-or-complete and expand-or-complete-prefix。

kill-region 会删除最后一个单词，vi-cmd-mode 的处理方式与 accept-line 相同。 空格和制表符如果没有绑定到这些函数中的任何一个，就会补全名称，然后在 AUTO_LIST 选项设置的情况下列出各种可能性。 任何未与 self-insert 或 self-insert-unmeta 绑定的其他字符都会发出蜂鸣声并被忽略。 将使用当前插入模式的绑定。

目前该命令不能重新定义或通过名称调用。

execute-last-named-cmd (ESC-z) (unbound) (unbound)

重做使用 execute-named-cmd 执行的最后一个函数。

与 execute-named-cmd 一样，该命令不能重新定义，但可以通过名称调用。

get-line (ESC-G ESC-g) (unbound) (unbound)

从缓冲堆栈中弹出顶行，并将其插入光标位置。

pound-insert (unbound) (#) (unbound)

如果缓冲区开头没有 # 字符，则在每行开头添加一个 # 字符。 如果有 # 字符，则在有 # 字符的每一行中删除一个 # 字符。 无论哪种情况，都接受当前行。 必须设置 INTERACTIVE_COMMENTS 选项才有用。

vi-pound-insert

如果当前行开头没有 # 字符，则添加一个。 如果有，则将其删除。 必须设置 INTERACTIVE_COMMENTS 选项才有用。

push-input

将整个当前多行结构体推入缓冲堆栈，并返回顶层 (PS1) 提示符。 如果当前解析器结构体只有一行，则与 push-line 完全相同。 下次编辑器启动或使用 get-line 弹出时，该结构体将从缓冲堆栈顶部弹出并加载到编辑缓冲区。

push-line (^Q ESC-Q ESC-q) (unbound) (unbound)

将当前缓冲区推入缓冲区堆栈并清除缓冲区。 下次编辑器启动时，缓冲区将从缓冲区堆栈顶部弹出，并载入编辑缓冲区。

push-line-or-edit

在顶层 (PS1) 提示符下，相当于 push-line。 在二级（PS2）提示符下，将整个当前多行结构移入编辑缓冲区。 后者相当于 push-input 后接 get-line。

read-command

仅对用户自定义小部件有用。 按键操作与正常操作一样被读取，但将被执行的命令名称存储在 shell 参数 REPLY 中，而不是正在执行的命令。 该参数可用作未来 zle 命令的参数。 如果键序未被绑定，则返回状态 1；但通常情况下，REPLY 会被设置为 undefined-key，以表示键序无用。

recursive-edit

仅对用户定义的小部件有用。 此时，编辑器会重新获得控制权，直到执行了一个通常会导致 zle 退出的标准小部件（通常是由按回车键引起的 accept-line）。 相反，控制权会返回到用户定义的小部件。 如果返回是由错误引起的，则返回的状态为非零，但函数仍在继续执行，因此可能会进行整理工作。 这使得用户自定义小部件可以安全地临时更改命令行或按键绑定。

下面这个小部件 caps-lock 就是一个例子。

```
self-insert-ucase() {
  LBUFFER+=${(U)KEYS[-1]}
}
```

```
integer stat

zle -N self-insert self-insert-ucase
zle -A caps-lock save-caps-lock
zle -A accept-line caps-lock

zle recursive-edit
stat=$?

zle -A .self-insert self-insert
zle -A save-caps-lock caps-lock
zle -D save-caps-lock

(( stat )) && zle send-break

return $stat
```

在输入 accept-line（即通常的返回键）或再次调用 caps-lock 小部件之前，输入的字母都会大写。 caps-lock 的处理方式是把旧定义保存为 save-caps-lock，然后重新绑定以调用 accept-line。请注意，递归编辑产生的错误会被检测为非零返回状态，并通过 send-break 小部件传播。

redisplay (unbound) (^R) (^R)

重新显示编辑缓冲区。

reset-prompt (unbound) (unbound) (unbound)

强制重新展开屏幕左右两边的提示符，然后重新显示编辑缓冲区。 这既反映了提示符变量本身的变化，也反映了扩展值的变化（例如时间或目录的变化，或提示符所指变量值的变化）。

否则，提示符只会在每次 zle 启动时展开，以及在显示被 shell 其他部分的输出（如作业通知）打断时展开（这会导致命令行被重新打印）。

reset-prompt 不会改变特殊参数 LASTWIDGET。

send-break (^G ESC-^G) (unbound) (unbound)

终止当前编辑器函数，例如 execute-named-command，或编辑器本身，例如 vared。否则，将中止当前行的解析；在这种情况下，中止的行可以在 shell 变量 ZLE_LINE_ABORTED 中找到。 如果在 vared 中中止编辑器，变量 ZLE_VARED_ABORTED 将被设置。

run-help (ESC-H ESC-h) (unbound) (unbound)

将缓冲区推入缓冲栈，然后执行命令 ‘run-help *cmd*’ 其中 *cmd* 是当前命令。 *cmd*’，其中 *cmd* 是当前命令。 run-help 通常为 man 的别名。

vi-set-buffer (unbound) (") (unbound)

指定要在随后命令中使用的缓冲区。 可指定的缓冲区有 37 个：26 个 ‘命名’ 缓冲区 "a 至 "z、‘yank’ 缓冲区 "0、9 个‘queued’ 缓冲区 "1 至 "9 和 ‘黑洞’ 缓冲区 "_。 命名的缓冲区也可以指定为 "A 至 "Z。

如果为剪切、更改或复制命令指定了缓冲区，相关文本将替换指定缓冲区中以前的内容。如果使用大写字母指定了一个已命名的缓冲区，则新剪切的文本将附加到缓冲区，而不是覆盖缓冲区。使用 "_ 缓冲区时，不会发生任何操作。这对于删除文本而不影响任何缓冲区非常有用。

如果剪切或更改命令没有指定缓冲区，则使用 "1，"1 至 "8 中的内容分别沿一个缓冲区移动；"9 中的内容丢失。如果 yank 命令没有指定缓冲区，则使用 "0。最后，没有指定缓冲区的粘贴命令将粘贴来自最近一条命令的文本，而与该命令可能使用的缓冲区无关。

当使用 zle 命令从小部件函数中调用时，可以选择使用参数指定缓冲区。例如

```
zle vi-set-buffer A
```

vi-set-mark (unbound) (m) (unbound)

在光标位置设置指定标记。

set-mark-command (^@) (unbound) (unbound)

在光标位置设置标记。 如果调用的参数为负数，则不设置标记，而是停用该区域，使其不再高亮显示（仍可用于其他目的）。 否则，该区域将被标记为激活状态。

spell-word (ESC-$ ESC-S ESC-s) (unbound) (unbound)

尝试对当前单词进行拼写更正。

split-undo

在当前更改处打破撤销序列。 这在 vi 模式下非常有用，因为在插入模式下所做的更改会在进入命令模式时合并。 类似地，undo 通常会将用户定义的小部件所做的所有更改统一复原。

undefined-key

当键入未与任何命令绑定的按键序列时，将执行该命令。 默认情况下会发出蜂鸣声。

undo (^_ ^Xu ^X^U) (u) (unbound)

递增撤销最后一次文本修改。 从用户定义的小部件调用时，会接收一个可选参数，该参数表示 UNDO_CHANGE_NO 变量返回的撤销历史的前一个状态；在达到该状态之前，所作修改都会被撤销，但会受到 UNDO_LIMIT_NO 变量的限制。

请注意，在 vi 命令模式下调用时，插入模式下的全部修改将被还原（reverted），因为在选择命令模式时，这些修改已被合并。

redo (unbound) (^R) (unbound)

增量重做未完成的文本修改。

vi-undo-change (unbound) (unbound) (unbound)

撤销上次文本修改。 如果重复，则重做修改。

visual-mode (unbound) (v) (unbound)

切换 vim 风格的可视化选择模式。如果当前已启用按行选择的可视化模式，则会将其改为按字符选择。如果在操作符之后使用，则会强制将随后的移动命令视为按字符移动。

visual-line-mode (unbound) (V) (unbound)

切换 vim 风格的逐行可视化选择模式。如果当前启用的是按字符的可视化模式，则会将其改为按行的可视化模式。如果在操作符之后使用，则会强制将随后的移动命令视为按行移动。

what-cursor-position (^X=) (ga) (unbound)

打印光标下的字符，以八进制、十进制和十六进制数的形式显示其代码，以及光标在缓冲区中的当前位置和当前行中的列数。

where-is

读取编辑器命令的名称，并打印调用指定命令的按键序列列表。 迷你缓冲区中提供了一组受限的编辑功能。 按键在特殊的 command 键映射中查找，如果找不到，则在主键映射中查找。

which-command (ESC-?) (unbound) (unbound)

将缓冲区推入缓冲堆栈，然后执行命令 ‘which-command *cmd*’ 。其中 *cmd* 是当前命令。 which-command 通常是 whence 的别名。

vi-digit-or-beginning-of-line (unbound) (0) (unbound)

如果最后执行的命令是作为参数一部分的数字，则继续执行参数。 否则，执行 vi-beginning-of-line。

18.6.7 文本对象

文本对象是用于根据某些标准选择文本块的命令。它们是 vim 文本编辑器的一项功能，因此主要用于 vi 操作符或可视化选择模式。不过，它们也可以在 vi-insert 或 emacs 模式下使用。下面列出的键绑定适用于 `viopp` 和 `visual` 键映射。

`select-a-blank-word` (aW)

> 选择包括相邻空格的单词，单词的定义是一系列非空格字符。使用数字参数时，将选择多个单词。

`select-a-shell-word` (aa)

> 选择当前的命令参数，并应用正常的引号规则。

`select-a-word` (aw)

> 使用普通的 vi-style 单词定义，选择包括相邻空白在内的单词。使用数字参数时，将选择多个单词。

`select-in-blank-word` (iW)

> 选择一个单词，单词的定义是一系列非空格字符。使用数字参数时，将选择多个单词。

`select-in-shell-word` (ia)

> 按常规引号规则选择当前命令参数。如果参数以匹配的引号字符开始和结束，则不在选择范围内。

`select-in-word` (iw)

> 使用普通的 vi-style 单词定义选择一个单词, 使用数字参数时，将选择多个单词。

18.7 字符高亮

行编辑器可以高亮显示行中具有特殊意义的字符或区域。 如果用户设置了这一功能，则由数组参数 `zle_highlight` 控制。

如果参数只包含 none，则所有高亮都会关闭。 请注意，该参数仍应是一个数组。

否则，数组的每个条目都应由一个表示高亮上下文的单词，然后一个冒号，接着一个用逗号分隔的列表组成，列表中列出了在该上下文中应用的高亮类型。

可用于高亮显示的上下文如下：

default

Any text within the command line not affected by any other highlighting. Text outside the editable area of the command line is not affected.

isearch

当增量历史搜索小部件之一处于活动状态时，搜索字符串或模式所匹配的命令行区域。

region

当前选中的文本。在 emacs 术语中，这被称为区域，以光标（点）和标记为边界。只有在使用 set-mark-command 或 exchange-point-and-mark 命令修改标记后，该区域才会突出显示。 需要注意的是，区域是否激活对其在 emacs 风格小部件中的使用没有影响，它只是决定是否高亮显示。在 vi 模式下，该区域与可视化模式下的选定文本相对应。

special

没有直接可打印表示，但行编辑器以特殊方式显示的单个字符。 下文将对这些字符进行说明。

suffix

该上下文在补全中用于标记为后缀的字符，如果补全在该处结束，这些字符将被移除，最明显的例子是目录名后的斜线 (/)。 请注意，后缀移除是可配置的；不同的补全可能会在不同的情况下移除后缀。

paste

执行粘贴文本的命令后，会显示插入的字符。

设置 region_highlight 时，首先应用描述区域的上下文 — isearch、region、suffix 和 paste — 然后应用 region_highlight，最后应用其余的 zle_highlight 上下文。 如果某个字符受到多个规范的影响，则最后一个规范胜出。

zle_highlight 可能包含其他字段，用于控制如何输出改变颜色的终端序列。 以下每个字段后面都有一个冒号和一个字符串，形式与按键绑定相同。 由于括号中显示的默认值已被广泛使用，因此绝大多数终端都不需要这样做。

fg_start_code (\e[3)

前景色转义序列的起始位置。 随后是代表颜色的一至三位 ASCII 数字。 仅用于调色板颜色，即不是通过颜色三连字符指定的 24 位颜色。

fg_default_code (9)

用来代替颜色的数字，以重置默认前景色。

`fg_end_code` (`m`)

前景色转义序列的终点。

`bg_start_code` (`\e[4`)

背景色转义序列的起始位置。 参见上文 `fg_start_code`。

`bg_default_code` (`9`)

用来代替颜色的数字，以重置默认背景颜色。

`bg_end_code` (`m`)

背景色转义序列的结束。

可用的高亮类型如下。请注意，并非所有终端都支持所有类型的高亮：

`none`

不对给定上下文应用高亮。 与其他类型的高亮同时出现并无用处；它用于覆盖默认值。

`fg=`*colour*

前景色应设置为 *colour*，即一个十进制整数，最广泛支持的八种颜色之一的名称，或者设置为 '#' 后面跟一个十六进制格式的 RGB 三连字符。

并非所有终端都支持此功能，而在支持此功能的终端中，也并非所有终端都提供测试功能，因此用户应根据终端类型来决定。 大多数终端都支持 `black`, `red`, `green`, `yellow`, `blue`, `magenta`, `cyan` 和 `white`，这些颜色可以通过名称来设置。 此外 `default` 可用于设置终端的默认前景色。 可以使用缩写；`b` 或 `bl` 选择黑色。 如果同时存在 `bold` 属性，某些终端可能会生成其他颜色。

在最新的终端和拥有最新终端数据库的系统上，可以通过命令 'echotc Co' 来测试支持的颜色数量；如果测试成功，则表明行编辑器将执行的颜色数量限制。 在任何情况下，颜色数都不得超过 256 种（即范围 0 至 255）。

某些现代终端模拟器支持 24 位真彩色（1600 万色）。在这种情况下，可以使用十六进制三连字符格式。这包括一个 '#'，后面跟一个三位或六位十六进制数，描述颜色的红、绿、蓝分量。通过 `zsh/nearcolor` 模块（参见 zsh/nearcolor 模块），十六进制三连串也可用于 88 和 256 色终端。

Colour 也是 color。

`bg=`*colour*

背景颜色应设置为 *colour*。 其作用与前景色类似，只是背景色通常不受粗体属性的影响。

bold

给定上下文中的字符以粗体显示。 并非所有终端都能区分粗体字体。

standout

给定上下文中的字符以终端的突出模式显示。 实际效果因终端而异；在许多终端上是反向视频。 在某些终端上，如果光标不闪烁，它就会以与突出模式相反出现，从而使光标的实际位置不太清晰。 在这些终端上，其他效果之一可能更适合高亮显示区域和匹配的搜索字符串。

underline

给定上下文中的字符以下划线显示。 有些终端会用不同的颜色显示前景；在这种情况下，空白不会高亮显示。

上述 '特殊' 字符如下。 无论这些字符是否高亮显示，此处描述的格式都将使用：

ASCII 控制字符

ASCII 范围内的控制字符显示为 '^' ，后跟基本字符。

不可打印的多字节字符

该项适用于不在 ASCII 范围内的控制字符，以及以下其他字符。 如果 MULTIBYTE 选项有效，当 COMBINING_CHARS 选项开启时，不在 ASCII 字符集中的多字节字符如果被报告为宽度为零，则会被视为组合字符。 如果该选项处于关闭状态，或者出现的字符不是有效的组合字符，则该字符将被视为不可打印字符。

无法打印的多字节字符以十六进制数字显示在角括号之间。 该数字是该字符在广义字符集中的码位；这可能是也可能不是 Unicode，取决于操作系统。

无效多字节字符

如果 MULTIBYTE 选项生效，那么任何一个或多个字节的序列，如果在当前字符集中不构成有效字符，将被视为一系列以特殊字符显示的字节。这种情况可以与其他不可打印字符加以区分，因为这些字节将以尖括号内的两个十六进制数字表示，与那些在当前字符集中仍然有效但不可打印的字符所用的四个或八个数字是不同的。

并非所有系统都支持这种方法：要使其有效，系统对宽字符的表示必须是 ISO 10646（也称作 Unicode）定义的通用字符集中的代码值。

封装的双宽度字符

当双倍宽度字符出现在一行的最后一列时，它将显示在下一行。原位置留下的空格将作为特殊字符高亮显示。

如果未设置 zle_highlight，或者没有适用于特定上下文的值，则应用的默认值相当于

```
zle_highlight=(region:standout special:standout
suffix:bold isearch:underline paste:standout)
```

即区域字符和特殊字符都以高亮模式显示。

在小部件中，可以通过设置特殊数组参数 region_highlight 来突出显示任意区域；请参阅 Zle 小部件。

---

19 补全小部件

19.1 说明

shell 的可编程补全机制有两种操作方式；这里定义了支持较新的、基于函数的机制的底层特性。 基于这些特性的一整套 shell 函数将在下一章 补全系统 中介绍，没有兴趣添加到该系统（或编写自己的系统 – 参见词典中的 'hubris' 条目<dictionary entry>）的用户可以跳过本节。 基于 compctl 内置命令的旧版系统将在 用 compctl 补全 中介绍。

补全部件由 zsh/zle 模块提供的 zle 内置命令的 -C 选项定义（参见 zsh/zle 模块）。例如

```
zle -C complete expand-or-complete completer
```

定义了一个名为 'complete' 的小部件。 第二个参数是处理补全的任何内置小部件的名称： complete-word、expand-or-complete、expand-or-complete-prefix、menu-complete、menu-expand-or-complete、reverse-menu-complete、list-choices 或 delete-char-or-list。 请注意，即使相关小部件已被重新绑定，该功能仍将有效。

当使用 zsh/zle 模块（Zsh 行编辑器）中定义的 bindkey 内置命令将这个新定义的 小部件 与某个按键绑定时，键入该按键将调用 shell 函数 'completer'。该函数负责使用下文所述的内置命令生成补全匹配。 与其他 ZLE 小部件一样，该函数在调用时关闭了标准输入。

一旦函数返回，补全代码将再次接管控制权，并以与指定的内置小部件（在本例中为 expand-or-complete）相同的方式处理匹配结果。

19.2 补全特殊参数

参数 ZLE_REMOVE_SUFFIX_CHARS 和 ZLE_SPACE_SUFFIX_CHARS 用于补全机制，但并不特殊。请参阅 Shell 使用的参数。

在补全小部件内部，以及从它们调用的任何函数中，有些参数具有特殊意义；在这些函数之外，它们对 shell 没有任何特殊意义。 这些参数用于在补全代码和补全小部件之间传递

信息。某些内置命令和条件代码会使用或更改这些参数的当前值。 在执行补全小部件时，任何现有的值都将被隐藏；除了 compstate 之外，这些参数在每次函数退出时（包括补全小部件内部的嵌套函数调用）都会被重置为进入函数时的值。

CURRENT

这是当前单词的编号，即 words 数组中光标当前所在单词的编号。 请注意，只有在未设置 ksharrays 选项的情况下，此值才是正确的。

IPREFIX

初始值为空字符串。 该参数的功能与 PREFIX 类似；它包含的字符串在 PREFIX 之前，不被视为匹配列表的一部分。 例如，通常情况下，一个字符串会从 PREFIX 的开头转到 IPREFIX 的结尾：

```
IPREFIX=${PREFIX%%\=*}=
PREFIX=${PREFIX#*=}
```

会导致前缀中包括第一个等号在内的部分不被视为匹配字符串的一部分。 compset 内置函数可以自动做到这一点，见下文。

ISUFFIX

与 IPREFIX相同，但后缀不应被视为匹配的一部分；注意 ISUFFIX 字符串位于 SUFFIX 字符串之后。

PREFIX

最初，将设置为当前单词从开头到光标位置的部分；也可以更改，以便为所有匹配提供一个通用前缀。

QIPREFIX

这个参数是只读的，包含引号字符串到正在补全的单词为止。例如，在补全 '"foo' 时，该参数包含双引号。如果使用了compset的-q选项(见下文)，并且原始字符串是 '"foo bar'，且光标在 'bar' 上，那么这个参数包含 '"foo '。

QISUFFIX

与 QIPREFIX 类似，但包含后缀。

SUFFIX

初始，它将被设置为当前单词从光标位置到末尾的部分；也可以进行修改，为所有匹配的单词提供一个共同的后缀。 当设置了 COMPLETE_IN_WORD 选项时，它的作用最大，否则命令行中的整个单词都会被视为前缀。

compstate

这是一个关联数组，包含不同的键和值，补全代码用它来与补全小部件交换信息。键是

all_quotes

compset 内置命令的 -q 选项（见下文）允许将引号字符串分解为不同的单词；如果光标位于其中一个单词上，该单词将被补全，并可能递归调用‘compset -q’。 使用该键可以测试当前以这种方式分解的引号字符串类型。它的值包含每个引号级别的一个字符。 对于使用单引号或双引号加引号的字符串，这些字符是单引号或双引号；对于使用 $'…' 加引号的字符串，这些字符是美元符号；对于不以引号字符开头的字符串，这些字符是反斜杠。 值中的第一个字符始终对应于最内层的引号。

context

这将由补全代码设置为尝试补全的整体上下文。可能的取值包括：

array_value

在数组参数赋值的值内补全时；在本例中，words 数组包含括号内的单词。

brace_parameter

在以 ${ 开头的参数扩展中补全参数名称时，也会设置该上下文。 在${(之后补全参数标志时，也将设置这种上下文；完整的命令行参数会呈现出来，处理程序必须测试要补全的值，以确定情况是否如此。

assign_parameter

在参数赋值中补全参数名称时。

command

补全普通命令时（在命令位置或用于命令参数）。

condition

在‘[[…]]’条件表达式中补全时；在这种情况下，words 数组只包含条件表达式中的单词。

math

在数学环境中完成时，如‘((…))’构造。

parameter

在以 $ 开头而不是 ${ 开头的参数扩展中，补全参数名称时。

redirect

当在重定向操作后补全时。

subscript

在参数下标内补全时。

value

当补全参数赋值的值时。

exact

控制设置 REC_EXACT 选项时的行为。 如果接受精确匹配，则设置为 accept，否则不设置。

如果在生成行上，至少一个匹配与行上字符串相同时设置了该值，则接受该匹配。

exact_string

如果找到完全匹配的字符串，则为该字符串，否则为未设置字符串。

ignored

由于与 compadd 内置命令 -F 选项中给出的模式之一相匹配而被忽略的补全次数。

insert

它控制着将匹配信息插入命令行的方式。 在进入小部件函数时，如果未设置，命令行将不会被更改；如果设置为 unambiguous，则会插入所有匹配的通用前缀；如果设置为 automenu-unambiguous，则会插入通用前缀，下次调用补全代码时可能会启动菜单补全（由于设置了 AUTO_MENU选项）； 如果设置为 menu 或 automenu，则将为当前生成的匹配启动菜单补全（在后一种情况下，这是因为设置了 AUTO_MENU）。 该值还可能包含字符串 'tab'，此时补全代码通常不会真正执行补全，而只会插入 TAB 字符。

退出时，它可以被设置为上述任意值（设置为空字符串与取消设置相同），也可以被设置为一个数字，在这种情况下，给出数字的匹配项将被插入命令行。 负数从最后一个匹配项开始倒数（ '-1' 选择最后一个匹配项），超出范围的数值会被回绕，因此数值为 0 会选择最后一个匹配项，而比最大值大 1 的数值会选择第一个匹配项。除非该键的值以空格结束，否则会像菜单补全一样插入匹配，即不会自动添加空格。

menu 和 automenu 还可以在冒号后指定要插入的匹配项的编号。 例如，'menu:2' 表示从第二个匹配开始菜单补全。

请注意，如果值包含子字符串 'tab' ，则生成的匹配将被忽略，只插入 TAB。

最后，也可以将其设置为 all，这样就可以将生成的所有匹配结果插入到行中。

insert_positions

当补全系统在行中插入一个无歧义的字符串时，可能会有多个位置缺少字符或插入的字符与至少一个匹配字符不同。 此键的值包含所有这些位置的冒号分隔列表，作为命令行的索引。

last_prompt

如果为每一个添加的匹配设置为非空字符串，则在显示补全信息列表后，补全代码会将光标移回上一个提示符。 最初，该值的设置与否取决于 ALWAYS_LAST_PROMPT 选项。

list

用于控制是否显示匹配列表或如何显示匹配列表。 如果未设置或为空，则永远不显示匹配结果；如果以 list 开头，则始终显示匹配结果；如果以 autolist 或 ambiguous 开头，则在 AUTO_LIST 或 LIST_AMBIGUOUS 选项通常会分别显示匹配结果时，显示匹配结果。

如果子字符串 force 出现在值中，即使只有一个匹配项，也会显示列表。通常，只有当至少有两个匹配项时，列表才会显示。

如果设置了 LIST_PACKED 选项，则该值包含子字符串 packed。如果为添加到组中的所有匹配给出了该子串，则该组将显示 LIST_PACKED 行为。同样，带子字符串 rows 的 LIST_ROWS_FIRST 选项也是如此。

最后，如果值中包含字符串 explanations，则只会列出解释字符串（如果有）；如果值中包含 messages，则只会列出消息（通过 compadd 的 -x 选项添加）。 如果同时包含 explanations 和 messages，则会列出两种解释字符串。 在进入补全小部件时，它将被适当设置，并可在此处更改。

list_lines

这给出了显示完整的补全列表所需的行数。 请注意，要计算显示的总行数，需要在此值上加上命令行所需的行数，这可以通过 BUFFERLINES 特殊参数的值获得。

list_max

初始值为 LISTMAX 参数的值。也可以设置为任何其他值；小部件退出时，该值的使用方式与 LISTMAX 值相同。

nmatches

目前补全代码添加的匹配数。

old_insert

在进入小部件时，该值将被设置为当前插入命令行的旧补全列表的匹配编号。如果没有插入匹配项，则不设置。

与 old_list 一样，该键的值只有在是 keep 字符串时才会被使用。如果它被小部件设置为这个值，并且有一个旧的匹配项插入到命令行中，那么这个匹配项将被保留；如果 insert 键的值指定插入另一个匹配项，那么这个匹配项将被插入到旧的匹配项之后。

old_list

如果在调用该小部件时，前一个补全操作的补全列表仍然有效，则将其设置为 yes。 通常只有在上一次编辑操作是使用补全小部件或内置补全函数时，才会出现这种情况。 如果有一个有效的列表，并且当前也显示在屏幕上，则此键的值为 shown。

小部件退出后，该键的值只有在被设置为 keep 时才会被使用。 在这种情况下，补全代码将继续使用旧列表。 如果小部件生成了新的匹配项，则不会使用这些匹配项。

parameter

在下标或参数赋值中值中补全时，参数名称。

pattern_insert

通常情况下，它被设置为 menu，即指定在使用 pattern_match（见下文）生成匹配集时使用菜单补全。 如果用户将其设置为任何其他非空字符串，且没有通过其他选项设置选择菜单补全，则代码会像普通补全一样，为生成的匹配插入任何通用前缀。

pattern_match

本地控制 GLOB_COMPLETE 选项的行为。 初始化时，当且仅当该选项被设置时，它才会被设置为‘*’。 补全小部件可以将其设置为该值、空字符串（与取消设置的效果相同）或任何其他非空字符串。 如果它为非空字符串，命令行中未加引号的元字符将被视为模式；如果它为 ‘*’，则光标位置上会出现通配符 ‘*’ ；如果它为空字符串或未设置，元字符将按字面意思处理。

请注意，如果将这（ compadd ）设置为非空字符串，则不会使用内置命令 compadd 的匹配规范。

quote

在引号内补全时，该值包含引号字符（即单引号、双引号或反引号）。 否则未设置。

引用

如果在单引号内补全，则设置为字符串 single；在双引号内补全，则设置为字符串 double；在反引号内补全，则设置为字符串 backtick。 否则不设置。

redirect

在重定向位置补全时的重定向操作符，即 <、> 等中的一个。

restore

在进入函数前将其设置为 auto，会强制上述特殊参数（words、CURRENT、PREFIX、IPREFIX、SUFFIX 和 ISUFFIX）在函数退出时恢复为之前的值。如果函数取消设置或将其设置为任何其他字符串，它们将不会被恢复。

to_end

指定插入匹配时光标移动到字符串末尾的场合。 在进入小部件函数时，可能是 single（如果在插入单个明确的匹配项时会发生）或者 match（如果在插入任何匹配项时都会发生）。（例如，通过菜单补全；这很可能是 ALWAYS_TO_END 选项的效果）

退出时，它可能会被设置为 single。 也可以设置为 always、空字符串或未设置；在这些情况下，光标将分别移动到字符串的末尾（始终或从不）。 任何其他字符串都将被视为 match。

unambiguous

此键为只读键，将始终设置为补全代码为迄今添加的所有匹配生成的通用（无歧义）前缀。

unambiguous_cursor

如果插入 unambiguous 键中的普通前缀，则光标将位于该键值的相对位置。光标将被置于该键给出的索引的字符之前。

unambiguous_positions

其中包含在无歧义字符串中缺少字符或插入的字符与至少一个匹配字符不同的所有位置。 这些位置以 unambiguous 键值所给字符串的索引形式给出。

vared

如果在使用 vared 内置函数编辑一行时调用了补全，则此键的值将被设置为作为 vared 参数给出的参数的名称。 只有在 vared 命令激活时，才会设置此键。

words

该数组包含当前正在编辑的命令行中出现的单词。

19.3 补全内置命令

compadd [-akqQfenUl12C] [-F *array*]
[-P *prefix*] [-S *suffix*]
[-p *hidden-prefix*] [-s *hidden-suffix*]
[-i *ignored-prefix*] [-I *ignored-suffix*]
[-W *file-prefix*] [-d *array*]
[-J *group-name*] [-X *explanation*] [-x *message*]
[-V *group-name*] [-o [*order*]]
[-r *remove-chars*] [-R *remove-func*]
[-D *array*] [-O *array*] [-A *array*]
[-E *number*]
[-M *match-spec*] [--] [*completions ...*]

该内置命令可用于直接添加匹配信息，并控制补全代码在每个可能补全时存储的所有信息。如果至少添加了一个匹配项，返回状态为 0；如果没有添加匹配项，返回状态为非 0。

补全代码将每个匹配项按顺序分解为七个字段：

<*ipre*><*apre*><*hpre*><*body*><*hsuf*><*asuf*><*isuf*>

第一个字段是来自命令行的忽略的前缀，即 IPREFIX 参数的内容加上 -i 选项的字符串。使用 -U 选项时，只使用 -i 选项中的字符串。字段 <*apre*> 是通过 -P 选项给出的可选前缀字符串。 <*hpre*> 字段是通过 -p 选项给出的字符串，它被视为匹配的一部分，但在列出补全时不应显示；例如，生成文件名的函数可能会以这种方式指定常用路径前缀。 <*body*> 是应出现在向用户显示的匹配列表中的匹配部分。 后缀 <*hsuf*>、<*asuf*> 和 <*isuf*> 与前缀 <*hpre*>、<*apre*> 和 <*ipre*> 相对应，分别由选项 -s、-S 和 -I 指定。

支持的标志是：

-P *prefix*

这将在每次匹配之前插入一个字符串。 给定的字符串不会被视为匹配的一部分，在插入字符串时，其中的任何 shell 元字符都不会被加引号。

`-S` *suffix*

类似于 `-P`，但会在每次匹配后插入一个字符串。

`-p` *hidden-prefix*

这给出了一个应插入到每个匹配字符串之前的字符串，但该字符串不应出现在匹配字符串列表中。除非给出 `-U` 选项，否则该字符串必须作为命令行字符串的一部分进行匹配。

`-s` *hidden-suffix*

类似于 '`-p`'，但会在每次匹配后插入一个字符串。

`-i` *ignored-prefix*

这将给出一个字符串，插入到任何使用 '`-P`' 选项给出的字符串之前。 如果不使用 '`-P`'，字符串会被插入到使用 '`-p`'的字符串之前，或者直接插入到每个匹配的字符串之前。

`-I` *ignored-suffix*

与 `-i`类似，但给出了一个被忽略的后缀。

`-a`

使用该标志后，*completions* 将作为数组的名称，而实际补全则是它们的值。如果只需要数组中的某些元素，*completions* 也可以包含下标，如 '`foo[2,-1]`'。

`-k`

使用此标志时，*completions* 将被视为关联数组的名称，而实际补全则是其键。 与 `-a` 一样，*words* 也可以包含下标，如 '`foo[(R)*bar*]`'。

`-d` *array*

这将添加按补全显示的字符串。每个 *completion* 都应在 *array* 中包含一个元素。补全代码将显示第一个元素，而不是第一个 *completion*，以此类推。*array* 可以作为数组参数的名称给出，也可以直接作为括号中空格分隔的单词列表给出。

如果显示字符串少于 *completions*，剩余的 *completions* 将保持不变；如果显示字符串多于 *completions*，剩余的显示字符串将被静默忽略。

`-l`

该选项只有与 `-d` 选项一起使用时才有效。如果给定了该选项，显示字符串将按行列出，而不是按列排列。

`-o` [*order*]

这将控制匹配结果的排序顺序。*order* 是一个以逗号分隔的列表，包含以下可能的值。 这些值可以缩写为前两个或三个字符。 请注意，排序是组名空间的一部分，因此不同排序的匹配结果不会出现在同一个组中。

`match`

如果给定，输出的顺序由匹配字符串决定；否则由显示字符串（即 `-d` 选项给定的字符串）决定。如果指定了'`-o`' 但省略了 *order* 参数，则默认情况下也是如此。

`nosort`

这表示 *completions* 已预先排序，其顺序应保持不变。 此值只能单独使用，不能与其他值结合使用。

`numeric`

如果匹配项包含数字，则按数字排序，而不是按词典排序。

`reverse`

颠倒排序顺序，反向排列匹配项。

`-J` *group-name*

给出存储匹配信息的组名称。

`-V` *group-name*

与 `-J` 类似，但命名的是一个未排序的分组。该选项与 `-J` 和 `-o nosort` 的组合相同。

`-1`

如果与 `-V` 选项一起使用，则只删除组内连续的重复数据。如果与 `-J` 选项一起使用，则没有明显效果。请注意，有此标志和无此标志的组处于不同的名称空间。

`-2`

如果与 `-J` 或 `-V` 选项一起使用，则会保留所有重复。同样，有此标志和无此标志的组处于不同的名称空间。

`-X` *explanation*

explanation 字符串将与匹配列表一起打印，位于当前选择的组之上。

在 *explanation* 中，可使用以下序列指定输出属性（参见 提示符扩展）：‘%B’, ‘%S’, ‘%U’, ‘%F’, ‘%K’及其小写对应序列，以及‘%{...%}’. ‘%F’, ‘%K’ 并且 ‘%{...%}’ 的参数形式与提示符扩展相同。 (注意，‘%G’ 序列不可用；应使用 ‘%{’ 的参数代替）。 序列 ‘%%’ 产生一个字面意义上的 ‘%’。

用户在自定义 `format` 样式时最常使用这些序列（参见 补全系统），但在编写补全函数时也必须考虑到这些序列，因为将带有未转义 ‘%’ 字符的描述传递给 `_arguments` 和 `_message` 等实用函数时，可能会产生意想不到的结果。如果要在描述（description）中传递任意文本，可以使用 `${my_str//\%/%%}` 等方式转义。

`-x` *message*

与 `-X` 类似，但即使组中没有匹配项，也会打印 *message*。

`-q`

如果键入的下一个字符是空白或没有插入任何内容，或者后缀只有一个字符，且键入的下一个字符是相同的字符，则 `-S` 指定的后缀将自动删除。

`-r` *remove-chars*

这是 `-q` 选项的更加多功能的形式。 如果输入的下一个字符插入了 *remove-chars* 中给出的字符之一，则 `-S` 给出的后缀或补全目录后自动添加的斜线将被自动删除。 该字符串会被解析为字符类，并理解 `print` 命令使用的反斜线序列。 例如，‘`-r "a-z\t"`’ 会在输入的下一个字符为小写字母或 TAB 时删除后缀，‘`-r "^0-9"`’ 会在输入的下一个字符为数字以外的字符时删除后缀。在这个字符串中，有一个额外的反斜杠序列是可以被理解的：‘`\-`’ 代表所有不插入任何内容的字符。因此，‘`-S "=" -q`’ 与‘`-S "=" -r "= \t\n\-"`’ 相同。

该选项也可在不使用 `-S` 选项的情况下使用；这样，当输入列表中的一个字符时，任何自动添加的空格都会被删除。

`-R` *remove-func*

这是 `-r` 选项的另一种形式。在接受匹配并插入后缀后，将在输入下一个字符后调用 *remove-func* 函数。 函数将后缀的长度作为参数传递给它，并可以使用普通（非完成）zle widget（参见 Zsh 行编辑器）中的特殊参数来分析和修改命令行。

`-f`

如果给定了该标志，从 *completions* 生成的所有匹配结果都会被标记为文件名。 它们不一定是实际文件名，但如果是实际文件名，且选项 `LIST_TYPES` 已设置，则会在补全列表中显示描述文件类型的字符。这也会强制在补全目录名称时添加斜线。

`-e`

该标志可用于告诉补全代码所添加的匹配是参数扩展的参数名。这将使 AUTO_PARAM_SLASH 和 AUTO_PARAM_KEYS 选项用于匹配。

`-W` *file-prefix*

该字符串是一个路径名，将与 `-p` 选项指定的任何前缀一起被添加到每个匹配项的前缀中，以形成一个完整的文件名进行测试。 因此，它只有与 `-f` 标志结合使用时才有用，否则不会执行测试。

`-F` *array*

指定一个包含模式的数组。 与其中一个模式匹配的 *completions* 将被忽略，即不被视为匹配项。

array 可以是一个数组参数的名称，也可以是一个用括号和引号括起来的字面模式列表，如‘`-F "(*?.o *?.h)"`’中。如果给出的是数组名称，则数组元素将作为模式。

`-Q`

该标志指示补全代码在将匹配插入命令行时，不对匹配符中的任何元字符加引号。

`-M` *match-spec*

这将提供 补全匹配控制 中描述的本地匹配规范。该选项可以给出多次。 在这种情况下，所有给出的 *match-spec* 都会用空格连接起来，形成要使用的规范字符串。 请注意，只有在未给出 `-U` 选项时，才会使用这些字符串。

`-n`

指定将匹配的 *completions* 加入匹配集，但不向用户列出。

`-U`

如果给出此标志，所有 *completions* 都会被添加到匹配集合中，补全代码将不进行匹配。通常情况下，该标志用于自行完成匹配的函数。

`-O` *array*

如果给定了该选项，*completions* **不会** 被添加到匹配集合中。 相反，匹配将像往常一样进行，所有匹配到的 *completions* 都将存储在数组参数中，该参数的名称为 *array*。

`-A` *array*

与 -O 选项相同，只不过 *completions* 中的匹配字符串不存储在 *array* 中，而是存储补全代码内部生成的字符串。 例如，如果匹配规范为 '-M "L:| no="'，当前单词为'nof'， *completions* 为 'foo' ，则该选项会在数组中存储字符串 'nofoo' ，而 -O 选项则存储最初给出的字符串 'foo' 。

-D *array*

与 -O 一样，*completions* 不会添加到匹配集合中。 相反，只要第 *n* 个 *completion* 不匹配，*array* 的第 *n* 个元素就会被移除。 相应 *completion* 匹配的元素会被保留。 可以多次使用此选项来从多个数组中删除元素。

-C

该选项会添加一个特殊匹配项，在插入一行时会扩展到所有其他匹配项，即使是在使用该选项后添加的匹配项也不例外。 与 -d 选项一起使用时，可以指定在列表中显示该特殊匹配的字符串。 如果没有指定字符串，则会显示为一个字符串，其中包含为其他匹配项插入的字符串，并根据屏幕宽度截断。

-E *number*

该选项会在添加了匹配的 *completions* 之后添加 *number* 个空匹配。 空匹配会占用补全列表的空间，但永远不会插入行中，也不能用菜单补全或菜单选择来选择。 这使得空匹配只在格式化补全列表和在补全列表中显示解释性字符串时有用（因为可以使用 -d 选项为空匹配提供显示字符串）。 由于除了一个空字符串之外的所有字符串都会被删除，因此该选项意味着 -V 和 -2 选项（即使给出了明确的 -J 选项）。 这一点很重要，因为它会影响到添加匹配的名称空间。

-
--

该标志会结束标志和选项列表。其后的所有参数都将作为 *completions* ，即使它们是以连字符开头的。

除 -M 标志外，如果这些标志中的任何一个被多次给出，则将使用第一个标志（及其参数）。

compset -p *number*
compset -P [*number*] *pattern*
compset -s *number*
compset -S [*number*] *pattern*
compset -n *begin* [*end*]
compset -N *beg-pat* [*end-pat*]
compset -q

该命令简化了特殊参数的修改，其返回状态允许对特殊参数进行测试。

这些选项是：

-p *number*

如果 PREFIX 参数的值至少有 *number* 个字符长，则会从中删除前 *number* 个字符，并追加到 IPREFIX 参数的内容中。

-P [*number*] *pattern*

如果 PREFIX 参数的值以与 *pattern* 匹配的内容开头，则匹配的部分会从 PREFIX 中移除，并追加到 IPREFIX 中。

如果不使用可选的 *number*，则取最长的匹配，但如果给出 *number*，则移动第 *number* 个之前的所有匹配。 如果 *number* 为负数，则移动第 *number* 个最长的匹配项。例如，如果 PREFIX 包含字符串 'a=b=c'，那么 `compset -P '*\='` 将把字符串 'a=b=' 移动到 IPREFIX 参数中，但 `compset -P 1 '*\='` 只移动字符串 'a='。

-s *number*

与 -p相同，但会将 SUFFIX 值中的最后 *number* 个字符转到 ISUFFIX 值的前面。

-S [*number*] *pattern*

与 -P 相同，但匹配 SUFFIX 的最后部分，并将匹配的部分转到 ISUFFIX 值的前面。

-n *begin* [*end*]

如果参数 CURRENT 指定的当前字词位置大于或等于 *begin*，则从 words 数组中删除第 *begin* 个字词之前的所有内容，并按 *begin* 递减参数 CURRENT 的值。

如果给出了可选的 *end*，则只有在当前单词位置也小于或等于 *end* 时，才会进行修改。在这种情况下，从 *end* 开始的单词也会从 words 数组中删除。

begin 和 *end* 都可以是负数，以便从 words 数组的最后一个元素开始倒数。

-N *beg-pat* [*end-pat*]

如果 words 数组中位于参数 CURRENT 值给出的索引处的元素之前的一个元素与 *beg-pat* 模式相匹配，则从 words 数组中删除匹配元素之前的所有元素，并更改 CURRENT 的值，使其指向已更改数组中的相同单词。

如果同时给出了可选模式 *end-pat*，并且 words 数组中存在与该模式匹配的元素，那么只有当该词的索引高于 CURRENT 参数给出的索引（因此匹配词必须在游标之后）时，才会修改参数。在这种情况下，从 end-pat 开始的单词

也会从 words 数组中删除。如果 words 数组中没有与 *end-pat* 匹配的单词，则会像没有给出 *end-pat* 一样进行测试和修改。

`-q`

当前正在补全的单词会按空格分割成不同的单词，并遵守通常的 shell 引号约定。 生成的单词存储在 words 数组中，CURRENT、PREFIX、SUFFIX、QIPREFIX 和 QISUFFIX 将被修改，以反映已补全的单词部分。

在上述所有情况下，如果测试成功且参数被修改，则返回状态为 0，否则为非 0。这样就可以在以下测试中使用该内置函数：

```
if compset -P '*\='; then ...
```

这样，补全代码就会忽略直到并包括最后一个等号在内的所有内容。

`compcall` [`-TD`]

这样就可以在补全小部件中使用 compctl 内置函数定义的补全。 除了只使用为特定命令提供的 compctl 外，匹配列表的生成过程与调用非小部件的补全函数（complete-word 等）相同。要强制代码在适当的地方尝试 compctl 的 -T 选项定义的补全函数和/或默认补全函数（无论是 compctl -D 还是内置默认），可以将 -T 和/或 -D 标志传递给 compcall。

返回状态可用于测试是否找到匹配的 compctl 定义。如果找到 compctl，返回状态为非零，否则为零。

请注意，该内置程序是由 zsh/compctl 模块定义的。

19.4 补全条件代码

在补全小部件中，可以使用以下用于 [[...]] 结构的附加条件代码。 这些代码适用于特殊参数。 所有这些测试也可由 compset 内置函数执行，但在条件代码的情况下，特殊参数的内容不会被修改。

`-prefix` [*number*] *pattern*

如果 compset 的 -P 选项测试成功，则为 true。

`-suffix` [*number*] *pattern*

如果 compset 的 -S 选项测试成功，则为 true。

`-after` *beg-pat*

如果只带有 *beg-pat*的 -N 选项的测试成功，则为 true。

`-between` *beg-pat end-pat*

如果两种模式的 `-N` 选项的测试都成功，则为 true。

19.5 补全匹配控制

当用户调用补全时，命令行上的当前 ***word***（即光标当前所在的单词）会被用来生成一个**匹配模式**。 只有那些与模式匹配的 ***completions*** 才会作为 ***matches*** 提供给用户。

默认的匹配模式是由当前单词生成的，其方法是

- 添加 '*'（匹配补全中的任意字符数）**或者，**
- 如果 shell 选项 `COMPLETE_IN_WORD` 已设置，则会在光标位置插入 '*'。

通过 `-M` 选项将 **匹配规范** 传递给 `compadd` 内置命令（参见 补全内置命令），可以有选择性地扩展这种狭义模式。 匹配规范由一个或多个 *matchers* 组成，以空格分隔。 匹配规范中的匹配器从左到右逐个应用。 一旦应用了所有匹配器，就会将补全与最终匹配模式进行比较，并丢弃不匹配的补全。

- 请注意，如果当前单词包含一个 glob 模式，并且 shell 选项 `GLOB_COMPLETE` 已被设置，或者特殊关联数组 `compstate` 的 `pattern_match` 键被设置为非空值（参见 补全特殊参数），则 `-M` 选项将被忽略。
- 补全系统 的用户一般不应直接使用 `-M` 选项，而应使用 `matcher-list` 和 `matcher` 样式（参见 补全系统配置 中的 **标准样式** 小节）。

每个匹配器由以下部分组成

- 一个大小写相关的字母
- 一个 ':',
- 一个或多个用管道 ('|')分隔的模式、
- 等号 ('=')，以及
- 另一个模式。

'=' 之前的模式用于匹配当前单词的子串。 对于每个匹配到的子串，匹配模式的相应部分将通过逻辑 OR 与 '=' 之后的模式进行扩展。

匹配器中的每个模式包括

- 空字符串或
- 一序列
  - 字面字符（可以用一个'\' 加引号），
  - 问号标记 ('?'),
  - 括号表达式（'[...]'；见 文件名生成 中的 ***Glob* 操作符** 小节），和/或
  - 括号表达示 (见下文).

其它 shell 模式是不允许的。

一个大括号表达式，就像一个方括号表达式一样，由下面列表组成

- 字面字符，
- 范围 (‘0-9’)，和/或
- 字符类(‘[:*name*:]’)。

不过，它们之间的区别如下：

- 大括号表达式由一对括号 (‘{...}’) 分隔。
- 括号表达式不支持否定。 也就是说，开头的 ‘!’ 或 ‘^’ 没有特殊含义，将被解释为字面字符。
- 如果当前单词中的一个字符与一个括号表达式中的第 *n* 个模式匹配时，仅与 ‘=’ 另一侧的括号表达式中的第 *n* 个模式（如果有的话）匹配模式的相应部分会进行扩展；如果另一侧没有括号表达式，则该模式为空字符串。 但是，如果任何一个括号表达式的元素多于另一个括号表达式，那么多余的条目将被忽略。 在比较索引时，每个字面字符或字符类都算作一个元素，但每个范围都会扩展为它所代表的字面字符的完整列表。 此外，如果在 ‘=’ 的 **两** 边上，第 *n* 个模式是 ‘[:upper:]’ 或 ‘[:lower:]’，那么这些模式也会扩展为范围。

请注意，尽管匹配系统尚未处理多字节字符，但这很可能是未来的扩展。 因此，建议使用 ‘[:upper:]’ 和 ‘[:lower:]’，而不是 ‘A-Z’ 和 ‘a-z’。

以下是匹配器支持的不同形式。 每种 **大写** 形式的行为都与小写匹配器完全相同，但在匹配模式过滤掉非匹配补全**后**，会增加一个额外步骤： 由大写匹配器的子模式匹配的每个匹配子串都会被当前单词的相应子串替换。 但是，来自 **小写** 匹配器的模式权重更高：如果当前单词的子串同时被小写和大写匹配器的模式匹配，则小写匹配器的模式获胜，匹配的相应部分不会被修改。

除非另有说明，否则列出的每个示例都假定 COMPLETE_IN_WORD 未设置（默认情况下是这样）。

m:*word-pat*=*match-pat*
M:*word-pat*=*match-pat*

对于当前单词中与 *word-pat* 匹配的每个子串，拓宽匹配模式的相应部分，使其额外匹配 *match-pat*。

例如：

m:{[:lower:]}={[:upper:]} 让当前单词中的任何小写字符都能补全为自己或其对应的大写字符。因此，‘foo’, ‘FOO’ 和‘Foo’ 将被视为与单词 ‘fo’ 匹配。

M:_= 将当前单词中的每一个下划线插入到每个匹配中，插入的相对位置相同，由其周围的子串匹配决定。 因此，在给定补全词 ‘foo’ 的情况下，单词 ‘f_o’ 将被补全为匹配词 ‘f_oo’，即使后者并不作为补全词出现。

b:*word-pat=match-pat*
B:*word-pat=match-pat*
e:*word-pat=match-pat*
E:*word-pat=match-pat*

对于当前单词开始（ b:eginning ） 或结尾（ e:nd） 处匹配 *word-pat* 的每个连续子串，扩大匹配模式的相应部分，使其额外匹配 *match-pat*。

例如：

‘b:-=+’ 可以将当前单词开头的任意数量的减号补全为一个减号或加号。

‘B:0=’ 会将当前单词开头的所有 0 添加到每个匹配的开头。

l:|*word-pat=match-pat*
L:|*word-pat=match-pat*
R:*word-pat*|*=match-pat*
r:*word-pat*|*=match-pat*

如果在当前单词的 l:eft (左)或 r:ight （右)边缘有子串与 *word-pat* 匹配，则扩大匹配模式的相应部分，使其额外匹配 *match-pat*。

对于每个 l:、L:、r: 和 R:匹配器（包括下面的匹配器），模式 *match-pat* 也可以是 ‘*’。 它可以匹配补全中的任意数量的字符。

例如：

即使设置了 COMPLETE_IN_WORD，且光标不在当前单词的末尾，‘r:|=*’ 也会在匹配模式后添加 ‘*’ 。

如果当前单词以减号开头， ‘L:|-=’ 将在每次匹配前加上减号。

l:*anchor*|*word-pat=match-pat*
L:*anchor*|*word-pat=match-pat*
r:*word-pat*|*anchor=match-pat*
R:*word-pat*|*anchor=match-pat*

对于当前单词中匹配 *word-pat* 的每个子串，如果其 l:eft 或 r:ight 上有另一个子串匹配 *anchor*，则扩大匹配模式的相应部分，使其额外匹配 *match-pat*。

请注意，这些匹配器（以及下面的匹配器）只修改 *word-pat* 所匹配的内容；它们不会改变 *anchor* （或 *coanchor*；请参阅下面的匹配器）所匹配内容的匹配行为。因此，除非匹配模式的相应部分被修改，否则当前单词中的锚点必须像当前单词的任何其他子串一样，在每次完成时都进行字面匹配。

如果一个匹配器包含至少一个锚点（包括下面有两个锚点的匹配器），那么 *match-pat* 也可以是 ‘*’ 或 ‘**’。‘*’ 可以匹配不包含任何匹配 *anchor* 的子字符串的补全的

任何部分，而 '**' 可以匹配补全的任何部分。 (请注意，这不同于 'l:' 和 'r:'的无锚形式中的 '*' 行为，也不同于 glob 表达式中的 '*' 和 '**'）。

例如：

'r:|.=*' 使补全语句 'comp.sources.unix' 与单词 '..u' 匹配 — 但与单词 '.u'不匹配。

给定补全词 '--foo'，匹配器 'L:--|no-=' 将补全 '--no-' 到匹配的 '--no-foo'。

l:*anchor*||*coanchor*=*match-pat*
L:*anchor*||*coanchor*=*match-pat*
r:*coanchor*||*anchor*=*match-pat*
R:*coanchor*||*anchor*=*match-pat*

对于当前单词中符合 *anchor* 和 *coanchor* 的任意两个连续子串，按照给出的顺序，在匹配模式中的相应部分之间插入模式 *match-pat*。

请注意，与 *anchor* 不同，模式 *coanchor* 并不改变 '*' 可以匹配的内容。

例如：

'r:?|||[[:upper:]]=*' 会将当前单词 'fB' 补全为 'fooBar'，但不会将其补全为 'fooHooBar'（因为此处的 '*' 无法匹配包含 '[[:upper:]]' 的内容），也不会将 'B' 补全为 'fooBar'（因为当前单词中没有与 *coanchor* 匹配的字符）。

给定当前单词 'pass.n' 和补全 'pass.byname'，匹配器 'L:.||[[:alpha:]]=by' 将产生匹配结果 'pass.name'。

x:

忽略该匹配器及其右侧的所有匹配器。

该匹配器用于标记匹配规范的结束。 在一个独立的匹配器列表中，这个匹配器没有任何作用，但在匹配规范被连接起来的情况下（如使用 补全系统 时经常出现的情况），它可以允许一个匹配规范覆盖另一个匹配规范。

19.6 补全小部件举例

第一步是定义小部件：

```
zle -C complete complete-word complete-files
```

然后，可以使用 bindkey 内置命令将小部件绑定到一个键上：

```
bindkey '^X\t' complete
```

之后输入 control-X 和 TAB 键后，将调用 shell 函数 `complete-files`。该函数将生成匹配，例如

```
complete-files () { compadd - * }
```

该函数将补全当前目录中与当前单词匹配的文件。

---

20 补全系统

20.1 说明

本章将介绍 "新 "补全系统的 shell 代码，即 `compsys`。 它是基于前一章 补全小部件中描述的功能， 用 shell 函数编写的。

这些功能与上下文有关，对开始补全的时间点很敏感。 许多补全功能已经提供。 因此，除了如何初始化系统（ 初始化中对此进行了描述）之外，用户无需了解任何细节，就能执行大量任务。

决定执行何种补全的上下文可能是

- 参数或选项位置：这些参数描述了请求补全的命令行上的位置。 例如，‘rmdir 的第一个参数，要补全的单词名称是一个目录’；
- 一个特殊的上下文，表示 shell 语法中的一个元素。 例如 ‘命令位置上的单词’ 或 ‘数组下标’。

完整的上下文规范还包含其他要素，我们将一一介绍。

除了命令名称和上下文，系统还采用了另外两个概念，即样式 (***styles***) 和标签 (***tags***)。 它们为用户提供了配置系统行为的方法。

标签具有双重作用。 它们是匹配的分类系统，通常表示用户可能需要区分的对象类别。例如，在填写 `ls` 命令的参数时，用户可能更愿意在 `directories` 之前尝试 `files`，因此这两个都是标记。 它们也会作为最右边的元素出现在上下文规范中。

样式可以修改补全系统的各种操作，例如输出格式，还可以修改使用哪种补全程序（以及使用顺序），或检查哪些标记。 样式可以接受参数，并使用 zsh/zutil 模块 中描述的 `zstyle` 命令进行操作。

总之，标记描述了 **什么** 是补全对象，以及 如何 补全对象的样式。 在执行的不同阶段，补全系统会检查当前上下文定义了哪些样式和/或标签，并以此修改其行为。 上下文处理决定了上下文中的标记和其他元素如何影响样式的行为，有关上下文处理的完整描述请参阅 补全系统配置。

当请求补全时，会调用一个调度函数；请参阅下面控制函数列表中 _main_complete 的描述。调度程序会决定调用哪个函数来生成补全，并调用该函数。调度结果将传递给一个或多个 ***completters***，这些函数实现了不同的补全策略：简单补全、纠错、带纠错的补全、菜单选择等。

一般来说，补全系统包含两种 shell 函数：

- 以 'comp' 开头的将被直接调用；这样的调用只有少数几个；
- 以 '_' 开头的函数会被补全代码调用。 这组 shell 函数用于实现补全行为并可与按键绑定，被称为 '小部件'。 当需要新的补全功能时，这些小工具就会大量出现。

20.2 初始化

如果系统已完全安装完毕，只需在初始化文件中调用 shell 函数 compinit（参见下一节）即可。 不过，用户也可以运行函数 compinstall 来配置补全系统的各个方面。

通常，compinstall 会将代码插入 .zshrc，但如果该文件不可写，它就会将代码保存到另一个文件中，并告诉你该文件的位置。 需要注意的是，您需要确保添加到 .zshrc 中的行被实际运行；例如，如果 .zshrc 通常提前返回，您可能需要将它们移到文件中较早的位置。 只要将它们放在一起（包括开始和结束时的注释行），就可以重新运行 compinstall，它将正确定位并修改这些行。 不过需要注意的是，如果重新运行 compinstall，手动添加到这一部分的代码很可能会丢失，不过使用命令 'zstyle' 的行应该会得到妥善处理。

新代码将在下次启动 shell 或手动运行 .zshrc 时生效；也可以选择立即生效。 不过，如果 compinstall 删除了定义，则需要重新启动 shell 才能看到更改。

要运行 compinstall，您需要确保它位于 fpath 参数中提到的目录中，如果 zsh 配置正确，只要启动文件没有从 fpath 中移除相应的目录，那么情况应该已经如此。 然后必须自动加载（建议使用 'autoload -U compinstall'）。 在提示符出现时，你可以随时中止安装，你的 .zshrc 完全不会被修改；只有在最后特别要求你确认时才会发生变化。

20.2.1 compinit 的使用

本节将介绍 compinit 的用法，当直接调用时，它将初始化当前会话的补全；如果运行了 compinstall，它将自动从 .zshrc 中调用。

要初始化系统，函数 compinit 应位于 fpath 参数中提到的目录中，并应自动加载（建议使用 'autoload -U compinit'），然后简单的以 'compinit' 的方式运行。 这将定义一些实用程序函数，安排自动加载所有必要的 shell 函数，然后重新定义所有补全的小部件，以便使用新系统。 如果使用 menu-select 小部件（它是 zsh/complist 模块的一部分），则应确保在调用 compinit 之前加载了该模块，以便重新定义该小部件。 如

果补全样式（见下文）被设置为默认执行扩展和补全，且 TAB 键绑定到 expand-or-complete，则 compinit 将把它重新绑定到 complete-word；这对于使用正确的扩展形式是必要的。

如果您需要使用原始的补全命令，您仍然可以在小部件名称前加上‘.’，例如‘.expand-or-complete’，从而将按键绑定到旧的小部件。

为了加快 compinit 的运行速度，我们可以让它生成一个转储配置文件，并在以后的调用中读入；这是默认设置，但也可以通过调用 compinit 并加上 -D 选项来关闭。 转储文件为 .zcompdump，与启动文件位于同一目录（即 $ZDOTDIR 或 $HOME）；也可以通过‘compinit -d *dumpfile*’给出明确的文件名。 下一次调用 compinit 时，将读取转储文件，而不是执行完全初始化。

如果补全文件的数量发生变化，compinit 会识别并生成新的转储文件。 不过，如果函数名称或 #compdef 函数（如下所述）第一行的参数发生变化，最简单的办法是手工删除转储文件，以便 compinit 下次运行时重新创建。 可以通过 -C 选项省略检查是否有新函数的过程。 在这种情况下，只有在没有转储文件的情况下，才会创建转储文件。

转储实际上是由另一个函数 compdump 完成的，但只有在更改配置（例如使用 compdef）后想要转储新配置时，才需要自己运行该函数。 为此，旧转储文件的名称将被记住。

如果设置了 _compdir 参数，compinit 就会将其用作查找补全函数的目录；只有在函数搜索路径中还没有补全函数时，才有必要这样做。

出于安全考虑，compinit 还会检查补全系统是否会使用非根用户或当前用户所有的文件，或者世界或组可写目录中非根用户或当前用户所有的文件。 如果发现此类文件或目录，compinit 将询问是否真的应该使用补全系统。 使用 -u 选项可避免这些测试，并使找到的所有文件都无需询问即可使用；使用 -i 选项可使 compinit 静默忽略所有不安全的文件和目录。 如果使用 -C 选项，只要转储文件存在，就会完全跳过安全检查。

运行 compaudit 函数可随时重试安全检查。 这与 compinit 使用的检查方法相同，但在直接执行 compaudit 时，对 fpath 的任何更改都会在函数本地进行，因此不会持久存在。 需要检查的目录可以作为参数传递；如果没有参数，compaudit 会使用 fpath 和 _compdir 查找补全系统目录，并根据需要将缺失的目录添加到 fpath 中。 要强制检查 fpath 中当前命名的目录，可在调用 compaudit 或 compinit 之前将 _compdir 设置为空字符串。

函数 bashcompinit 与 bash 的可编程补全系统兼容。 运行该函数时，它将定义 compgen 和 complete 函数，这两个函数与同名的 bash 内置函数相对应。 这样就可以使用为 bash 编写的补全规范和函数。

20.2.2 自动加载的文件

补全时使用的自动加载函数的惯例是以下划线开头；如前所述，fpath/FPATH 参数必须包含存储这些函数的目录。 如果 zsh 已在系统中正确安装，那么 fpath/FPATH 就会自动包含标准函数所需的目录。

对于不完整的安装，如果 compinit 无法在搜索路径中找到足够的以下划线开头的文件（少于 20 个），则会尝试在搜索路径中添加 _compdir 目录，以找到更多文件。 如果该目录有一个名为 Base 的子目录，所有子目录都将被添加到路径中。 此外，如果 Base 子目录下有一个名为 Core 的子目录，compinit 将把该子目录的所有子目录添加到路径中：这使得函数的格式与 zsh 源代码发布版中的格式相同。

运行 compinit 时，它会搜索所有可通过 fpath/FPATH 访问的此类文件，并读取每个文件的第一行。 该行应包含下述标记之一。 如果文件的第一行不是以这些标记之一开头，则不会被视为补全系统的一部分，也不会受到特殊处理。

标记是：

#compdef *name* ... [-{p|P} *pattern* ... [-N *name* ...]]

> 该文件将自动加载，在补全 *name*s 时将调用其中定义的函数，每个 *name*s 要么是要补全参数的命令名称，要么是下面描述的 -*context*- 形式的一系列特殊上下文之一。
>
> 每个 *name* 也可以是 '*cmd*=*service*' 的形式。 在补全 *cmd* 命令时，函数的行为通常会被视为正在补全 *service* 命令（或特殊上下文）。 这提供了一种改变函数行为的方法，这些函数可以执行多种不同的补全操作。 它是通过在调用函数时设置参数 $service 来实现的；函数可以自行选择如何解释，而较简单的函数可能会忽略它。
>
> 如果 #compdef 行包含 -p 或 -P 选项之一，则后面的单词将被视为模式。 当试图补全与其中一个模式匹配的命令或上下文时，函数将被调用。 选项 -p 和 -P 分别用于指定在其他补全之前或之后尝试的模式。 因此 -P 可用来指定默认操作。
>
> 选项 -N 用于 -p 或 -P 之后的列表；它指定剩余的词不再定义模式。 可以根据需要在这三个选项之间多次切换。

#compdef -k *style key-sequence* ...

> 该选项会创建一个与内置小部件 *style* 行为类似的小部件，并将其绑定到给定的 *key-sequence*s 上（如果有的话）。 *style* 必须是执行补全的内置小部件之一，即 complete-word、delete-char-or-list、expand-or-complete、expand-or-complete-prefix、list-choices、menu-complete、menu-expand-or-complete 或 reverse-menu-complete。 如果已加载 zsh/complist 模块（请参阅 zsh/complist 模块），小部件 menu-select 也将可用。
>
> 键入 *key-sequence*s 之一时，文件中的函数将被调用以生成匹配。 需要注意的是，如果某个键已经绑定（即绑定到 undefined-key 以外的其他键），则不会重新绑

定。 创建的小部件与文件名相同，可以像往常一样使用 bindkey 绑定到任何其他键。

#compdef -K *widget-name style key-sequence* [*name style seq ...*]

除了每个 *widget-name style* 对只能给出一个 *key-sequence* 参数外，该参数与 -k 类似。不过，整组三个参数可以用不同的参数重复使用。 需要特别注意的是，每组中的 *widget-name* 必须是不同的。 如果不是以 '_' 开头，则会被添加。 *widget-name* 不应与任何现有小部件的名称相冲突：基于函数名称的名称最有用。 例如

```
#compdef -K _foo_complete complete-word "^X^C" \
  _foo_list list-choices "^X^D"
```

(在一行中) 定义了一个用于完成的小部件 _foo_complete，绑定到'^X^C'，以及一个用于列出的小部件 _foo_list，绑定到'^X^D'。

#autoload [*options*]

带有 #autoload 标记的函数会被标记为自动加载，但不会被特殊处理。 通常情况下，这些函数会在某个补全函数中被调用。 提供的任何 *options* 都将传递给 autoload 内置函数；典型用法是 +X 强制立即加载函数。 请注意，-U 和 -z 标志总是隐式添加的。

是标记名的一部分，后面不允许留白。 #compdef 标签使用下面描述的 compdef 函数；主要区别在于函数名称是隐式提供的。

可定义补全函数的特殊上下文有:

-array-value-

数组赋值的右侧 ('*name*=(...)')

-assign-parameter-

赋值中参数的名称，即 '=' 的左边

-brace-parameter-

大括号内的参数扩展名 ('${...}')

-command-

命令位置上的一个单词

-condition-

条件中的单词 ('[[...]]')

-default-

没有其他补全定义的任何单词

-equal-

以等号开头的单词

-first-

该函数会在任何其他补全函数之前被调用。 被调用的函数可以将 _compskip 参数设置为多种值之一： all：不再尝试其他补全函数；包含子串 patterns 的字符串：不调用模式补全函数；包含 default 的字符串：不调用 '-default-' 上下文的函数，但会调用为命令定义的函数。

-math-

在数学语境中，如 '((…))'

-parameter-

参数扩展的名称('$…')

-redirect-

重定向运算符后的单词。

-subscript-

参数下标内容。

-tilde-

在开头的波浪线之后（'~'），单词的第一个斜线之前。

-value-

在赋值的右边。

这些上下文每个都有默认实现。 在大多数情况下，上下文 -*context*- 由相应的函数 _*context* 实现，例如上下文 '-tilde-' 和函数 '_tilde'）。

上下文 -redirect- 和 -value- 允许使用额外的特定上下文信息。 (在内部，这些信息由调用 _dispatch 的每个上下文的函数处理）。 额外信息用逗号分隔。

对于 -redirect- 上下文，额外信息的格式为 '-redirect-,*op*,*command*'，其中 *op* 是重定向操作符，*command* 是该行上的命令名称。 如果行中还没有命令，*command* 字段将为空。

对于 -value- 上下文，其形式为 '-value-,*name*,*command*'，其中 *name* 是赋值左侧的参数名称。 对于关联数组中的元素，例如 'assoc=(key <TAB>'，*name* 会扩展为

‘*name-key*’。 在某些特殊情况下，例如在 ‘make CFLAGS=’ 之后补全，*command* 部分会给出命令名称，此处为 make；否则为空。

没有必要定义完全特定的补全，因为所提供的函数会尝试通过逐步替换 ‘-default-’ 元素来生成补全。 例如，当在 ‘foo=<TAB>’ 后补全时，_value 将依次尝试 ‘-value-,foo,’（注意 *command* 部分为空）、‘-value-,foo,-default-’ 和‘-value-,-default-,-default-’，直到找到可以处理上下文的函数。

作为一个例子：

```
compdef '_files -g "*.log"' '-redirect-,2>,-default-'
```

补全在 ‘2> <TAB>’ 之后与 ‘*.log’ 匹配的文件，适用于任何未定义更多特定处理程序的命令。

还有：

```
compdef _foo -value-,-default-,-default-
```

指定 _foo 为未定义特殊函数的参数值提供补全。 这通常由函数 _value 本身处理。

查找样式时也使用相同的查找规则（如下所述），例如

```
zstyle ':completion:*:*:-redirect-,2>,*:*' file-patterns '*.log'
```

是另一种方法，可让 ‘2> <TAB>’ 后的补全操作补全与 ‘*.log’ 匹配的文件。

20.2.3 函数

以下函数由 compinit 定义，可直接调用。

compdef [-ane] *function name* ... [-{p|P} *pattern* ... [-N *name* ...]]
compdef -d *name* ...
compdef -k [-an] *function style key-sequence* [*key-sequence* ...]
compdef -K [-an] *function name style key-seq* [*name style seq* ...]

第一种形式定义了在给定上下文中调用 *function* 来补全的 #compdef 标记。

或者，所有参数的形式可以是 ‘*cmd=service*’。 此处的 *service* 应已由 #compdef 文件中的 ‘*cmd1=service*’ 行定义，如上所述。 *cmd* 的参数将以与 *service* 相同的方式完成。

function 参数也可以是包含几乎所有 shell 代码的字符串。 如果字符串中包含等号，则上述参数优先。 可以使用选项 -e，将第一个参数指定为 shell 代码，即使其中包含等号。 该字符串将使用 eval 内置命令来生成补全。 这样就可以避免定义新的补全函数。 例如，将以 ‘.h’ 结尾的文件作为 foo 命令的参数来补全：

```
compdef '_files -g "*.h"' foo
```

选项 -n 可防止已为命令或上下文定义的补全被覆盖。

选项 -d 会删除为所列命令或上下文定义的任何补全。

*name*s 也可以包含 -p、-P 和 -N 选项，如 #compdef 标记所述。 对参数列表的影响是相同的，即在最初尝试的模式定义、最后尝试的模式定义以及正常命令和上下文之间切换。

参数 $_compskip 可由任何为模式上下文定义的函数设置。 如果将其设置为包含子串 'patterns' 的值，则不会调用任何模式函数；如果将其设置为包含子串 'all' 的值，则不会调用任何其他函数。 在使用 -p 选项时，以这种方式设置 $_compskip 特别有用，否则调度程序会在调用模式上下文函数后继续调用其他函数（可能是默认函数），如果处理不当，可能会导致显示的补全可能混乱。

带有 -k 的形式定义了一个与 *function* 同名的小部件，它将为每个 *key-sequence*s 调用；这就像 #compdef -k 标记一样。 该函数应生成所需的补全，否则其行为将与名称作为 *style* 参数给出的内置小部件相同。 可用的小部件有 complete-word, delete-char-or-list, expand-or-complete, expand-or-complete-prefix, list-choices, menu-complete, menu-expand-or-complete 和 reverse-menu-complete, 以及 menu-select（如果已加载 zsh/complist 模块）。 如果键已绑定到 undefined-key 以外的其他键，则选项 -n 将阻止键被绑定。

带有 -K 的形式与此类似，它基于同一个 *function* 定义多个小部件，每个小部件都需要 *name*、*style* 和 *key-seq*序列三个参数，其中后两个参数与 -k 相同，第一个参数必须是以下划线开头的唯一小部件名称。

在适用的情况下，-a 选项使 *function* 可自动加载，相当于 autoload -U *function*。

函数 compdef 可用于将现有的补全函数与新命令关联起来。 例如

```
compdef _pids foo
```

使用函数 _pids 补全命令 foo 的进程 ID。

还请注意下面介绍的 _gnu_generic 函数，该函数可用于能理解 '--help' 选项的命令补全选项。

20.3 补全系统配置

本节简要概述了补全系统的工作原理，然后详细介绍了用户如何配置匹配的生成方式和时间。

20.3.1 概述

当在命令行的某处尝试补全时，补全系统就会开始构建上下文。 上下文代表了 shell 所知道的命令行含义和光标位置的意义。 其中包括命令字（如"grep"或"zsh"）和当前命令字作为参数的选项（如 zsh 的‘-o’选项，它将 shell 选项作为参数）。

上下文开始时非常通用（"我们正在开始补全"），随着学习的深入，上下文会变得越来越具体（"当前单词所在的位置通常是命令名 "或 "当前单词可能是变量名 "等等）。 因此，在调用同一个补全系统时，上下文会有所不同。

这些上下文信息被浓缩成一个字符串，由多个字段组成，中间用冒号隔开，在本文的其余部分中简称为‘上下文’。 请注意，补全系统的用户很少需要编写上下文字符串，除非编写一个新函数来执行新命令的补全。 用户可能需要做的是编写一个 ***style*** 模式，在需要查找配置补全系统的上下文敏感选项时与上下文进行匹配。

接下来的几段将解释补全函数套件中的上下文是如何构成的。 接下来将讨论如何定义 ***styles***。 样式决定匹配结果的生成方式，与 shell 选项类似，但控制能力更强。 它们由 `zstyle` 内置命令定义（zsh/zutil 模块）。

上下文字符串总是由一组固定的字段组成，字段之间用冒号隔开，第一个字段前有一个前导冒号。 未知字段留空，但周围的冒号还是会出现。 字段的顺序总是 `:completion:`*function*`:`*completer*`:`*command*`:`*argument*`:`*tag*。 其含义如下：

- 字面字符串 `completion`，表示该样式由补全系统使用。 这将该上下文与 zle 小部件和 ZFTP 函数等使用的上下文区分开来。
- *function*，如果补全是通过一个已命名的小部件而不是通过正常的补全系统调用的。 通常情况下，它是空白的，但特殊部件（如 `predict-on`）和发行版 `Widget` 目录中的各种函数会将其设置为该函数的名称，通常是缩写形式。
- 当前激活的 *complete*，即函数名称，去掉前导下划线，并将其他下划线转换为连字符。 一个‘completer’可以全面控制补全程序的执行方式；‘complete’是最简单的补全程序，但也存在其他补全程序来执行相关任务，如更正，或修改后一个补全程序的行为。 更多信息，请参阅 控制函数。
- *command* 或特殊的 *-context-*，就在 `#compdef` 标记或 `compdef` 函数之后出现。 针对具有子命令的命令的补全函数通常会修改该字段，使其包含命令名称、减号和子命令。 例如，`cvs` 命令的补全函数在补全 `add` 子命令的参数时，会将该字段设置为 `cvs-add`。
- *argument*；这表示我们正在补全哪个命令行或选项参数。 对于命令参数，一般采用 `argument-`*n* 的形式，其中 *n* 是参数的编号；对于选项参数，则采用 `option-`*opt*`-`*n* 的形式，其中 *n* 是选项 *opt* 的参数编号。 不过，只有在使用标准 UNIX 样式的选项和参数对命令行进行解析时才会出现这种情况，因此许多补全程序并不设置它。
- *tag*. 如前所述，标记用于区分补全函数在特定上下文中可以生成的匹配类型。 任何补全函数都可以使用自己喜欢的标记名，下面列出了一些比较常用的标记名。

随着函数的执行，上下文逐渐组合在一起，从主入口点开始，必要时添加 `:completion:` 和 *function* 元素。 然后，补全器添加 *completer* 元素。 上下文补全器添加 *command* 和 *argument* 选项。 最后，在知道补全类型时，会添加 *tag*。 例如，上下文名称

```
:completion::complete:dvips:option-o-1:files
```

表示正常补全作为 dvips 命令的 -o 选项的第一个参数进行了尝试：

```
dvips -o ...
```

而补全函数将生成文件名。

通常，补全将按照补全函数给出的顺序对所有可能的标记进行尝试。 不过，可以通过使用 `tag-order` 样式来改变顺序。 这样，补全就会被限制在按给定顺序排列的给定标记列表中。

`_complete_help` 可绑定命令显示了在某一点上可用于补全的所有上下文和标记。 这为查找 `tag-order` 和其他样式的信息提供了一种简便的方法。 该命令在 可绑定命令 中有所描述。

在查找样式时，补全系统使用完整的上下文名称，包括标记。 因此，查找样式的值由两部分组成：上下文（与最具体（最适合）的模式匹配）和样式本身的名称（必须完全匹配）。 下面的示例说明，对于适用范围较广的样式，可以松散地定义模式，而对于适用范围较窄的样式，则可以根据需要严格定义模式。

例如，许多补全函数可以以简单和详细两种形式生成匹配，并使用 `verbose` 样式来决定使用哪种形式。 要使所有此类函数都使用详细形式，请将

```
zstyle ':completion:*' verbose yes
```

放在启动文件（可能是 `.zshrc`）中。 这使得 `verbose` 样式在补全系统中的每种上下文中都具有 `yes` 的值，除非该上下文有更具体的定义。 最好避免将样式定义为 '*'，以防该样式在补全系统之外有其他含义。

只需使用 `compinstall` 函数，就可以配置许多此类通用样式。

使用 `verbose` 样式的一个更具体的例子是 `kill` 内置程序的补全。 如果设置了该样式，内置程序就会列出完整的作业文本和进程命令行；否则，就会只显示作业编号和 PID。 要关闭该样式，请执行以下操作:

```
zstyle ':completion:*:*:kill:*:*' verbose no
```

为了获得更多控制权，样式可以使用标记 'jobs' 或 'processes'。 要关闭仅针对作业的冗长显示，请执行以下操作:

```
zstyle ':completion:*:*:kill:*:jobs' verbose no
```

zstyle 的 -e 选项甚至允许将补全函数代码作为样式的参数；这需要对补全函数的内部结构有一定的了解（参见 补全小部件）。 例如

```
zstyle -e ':completion:*' hosts 'reply=($myhosts)'
```

这就会强制每次需要主机名时从变量 myhosts 中读取 hosts 样式的值。如果 myhosts 的值是动态变化的，这是有用的。 另一个有用的例子，请参阅下面 file-list 样式描述中的示例。 这种形式可能会比较慢，对于 menu 和 list-rows-first 等常用样式，应避免使用。

请注意，样式的 **定义** 顺序并不重要；样式机制使用特定样式的最具体匹配来确定值集。字符串优先于模式（例如，‘:completion::complete:::foo’ 比 ‘:completion::complete:::*' 更具体），较长的模式优先于模式 ‘*’。详情请参见 zsh/zutil 模块。

使用通配符 (*) 以外的其他内容来匹配上下文中间部分的上下文模式 — 即 :completion:*function*:*completer*:*command*:*argument*:*tag* 中的*completer*, *command* 和 *argument* 。应明确包含所有六个冒号 (:)。如果不这样做，像 :completion:*:foo:* 这样的模式就可能匹配到 foo 以外的其他组件（例如，当要匹配 *command* 时，却匹配到 *completer*）。

样式名称与标记名称一样是任意命名的，取决于补全函数。 不过，下面两节列出了一些最常用的标记和样式。

20.3.2 标准标记

以下部分内容仅用于查找特定样式，并不代表某种匹配类型。

帐户

用于查询 users-hosts 样式

all-expansions

在添加包含所有可能扩展的单个字符串时被 _expand 补全器使用

all-files

用于所有文件的名称(有别于特定子集，参见 globbed-files 标记）。

arguments

用于命令的参数

arrays

用于数组参数名

`association-keys`

用于关联数组的键；在补全该类型参数的下标时使用

`bookmarks`

补全书签时（如 URL 和 `zftp` 函数套件）

`builtins`

用于内置命令的名称

`characters`

用于命令参数中的单字符，如 `stty`。 也用于在开头括号后补全字符分类

`colormapids`

用于 X colormap ids

`colors`

用于颜色名称

`commands`

用于外部命令的名称。 复杂命令（如 `cvs`）在补全子命令名称时也会用到。

`contexts`

用于 `zstyle` 内置命令参数中的上下文

`corrections`

用于 `_approximate` 和 `_correct` 补全器使用，以进行可能的修正

`cursors`

用于 X 程序使用的光标名称

`default`

在某些上下文中使用，以便在更具体的标记也有效时提供默认值。 请注意，该标记仅在上下文名称的 *function* 字段被设置时使用

`说明`

用于查找 `format` 样式的值，以生成匹配类型的说明

`设备`

设备特殊文件的名字

directories

目录名 — 当补全 cd 和相关内置命令的参数时，如果 cdpath 数组已设置，将使用 local-directories 代替

目录栈

目录栈中的条目

显示器

X 显示器名字

domains

用于网络域

email-*plugin*

从 _email_addresses 的 '_email-*plugin*' 后端获取电子邮件地址

扩展

用于 _expand 补全器，以补全命令行单词扩展后产生的单个单词（而不是完整的扩展集）。

extensions

用于 X 服务扩展

file-descriptors

表示打开的文件描述符的数量

files

补全文件名的函数使用的通用文件匹配标记

fonts

用于 X 字体名

fstypes

用于文件系统类型（例如用于 mount 命令）

函数

函数名称 — 通常是 shell 函数，但某些命令可能理解其他类型的函数

`globbed-files`

用于文件名是通过模式匹配生成时

`groups`

用于用户组的名字

`history-words`

用于历史中的单词

`hosts`

用于主机名

`indexes`

用于数组索引

`interfaces`

用于网络接口

算术求值

用于作业 (‘jobs’ 内置程序列出的)

键映射

用于 zsh 键映射的名字

`keysyms`

用于 X keysyms 的名字

`libraries`

用于系统库的名字

`limits`

用于系统限制

`local-directories`

在补全 `cd` 和相关内置命令（比较 `path-directories`）的参数时，用于查找当前工作目录的子目录名称 — 当 `cdpath` 数组未设置时，将使用 `directories` 代替

mailboxes

用于电子邮件目录

manuals

用于手册页的名字

maps

用于映射名（例如 NIS 映射）

messages

用于查找 format 样式的信息

修饰符

用于 X 修饰符的名字（X modifiers）

modules

用于模块（如 zsh 模块）

my-accounts

用于查询 users-hosts 样式

named-directories

用于命名的目录（你不会猜到吧？）

names

用于所有类型的名称

newsgroups

用于 USENET 组

nicknames

用于 NIS 映射的昵称

选项

用于命令选项

original

在提供原始字符串作为匹配时,被 _approximate、_correct 和 _expand 补全器使用。

other-accounts

用于查询 users-hosts 样式

packages

用于软件包（例如 rpm 或已安装的 Debian 软件包）

参数

用于参数名称

path-directories

当补全 cd 的参数和相关内置命令的参数时，用于通过搜索 cdpath 数组找到的目录名称（比较 local-directories。

paths

用于查找 expand、ambiguous 和 special-dirs 样式的值

pods

用于 perl pods (文档文件）

ports

用于通讯端口

prefixes

用于前缀（如 URL 的前缀）

printers

用于打印队列名称

processes

用于进程标识符

processes-names

用于为 killall 生成进程名称时查找 command 样式

sequences

用于序列（如 mh 序列）

会话

zftp 函数套件中的会话

signals

用于信号名

字符串

用于字符串（例如 cd 内置命令的替换字符串）

样式

zstyle 内置命令使用的样式

后缀

用于文件扩展名

标记

用于标记（如 rpm 标记）

目标

用于 makefile 目标

时区

用于时区(例如，在设置 TZ 参数时）

类型

关于各种类型的内容（例如 xhost 命令的地址类型）。

urls

用于在补全 URL 时查找 urls 和 local 样式

users

用于用户名

values

用于某些列表中的一组数值之一

variant

用于 _pick_variant 在确定特定命令名安装了什么程序时查找要运行的命令。

visuals

用于 X 视觉效果

warnings

用于为警告查找 format 样式

小部件

用于 zsh 小部件名称

windows

用于 X 窗口的 ID

zsh-options

用于 shell 选项

20.3.3 标准样式

请注意，其中几个样式的值代表布尔值。 任何字符串 'true', 'on', 'yes' 和 '1' 都可用于表示值 'true' ，任何字符串 'false', 'off', 'no' 和 '0' 都可用于表示值 'false' 。 除非明确提及，否则任何其他值的行为都是未定义的。 如果未设置样式，默认值可能是 'true' 或 'false' 。

其中一些样式会首先针对与匹配类型相对应的所有可能标记进行测试，如果未找到样式，则针对 default 标签进行测试。 这类样式中最著名的是 menu、list-colors 和控制补全列表的样式，如 list-packed 和 last-prompt。 在对 default 标记进行测试时，只有上下文的 *function* 字段会被设置，因此使用 default 标记的样式通常会按以下方式定义：

```
zstyle ':completion:*:default' menu ...
```

accept-exact

除了对当前上下文有效的标记外，还对 default 标记进行测试。 如果将其设置为 'true' ，且任何一个试验匹配都与命令行中的字符串相同，则该匹配将立即被接受（即使在其他情况下它会被认为是模棱两可的）。

在补全路径名时（使用的标记为 'paths' ），除布尔值外，该样式还接受任意数量的模式作为值。 即使命令行中包含更多部分键入的路径名组件，且这些组件与所接受目录下的文件不匹配，也会立即接受与这些模式之一匹配的路径名。

_expand 补全器也使用这种样式来决定是否扩展以斜线（tilde）开头的单词或参数扩展。 例如，如果有参数 foo 和 foobar，只有当 accept-exact 设置为 'true' 时，字符串 '$foo' 才会被展开；否则，将允许补全系统将 $foo 补全为 $foobar。如果样式被设置为'continue'，_expand 将添加扩展作为匹配，补全系统也将被允许继续。

accept-exact-dirs

用于文件名补全。 与 accept-exact 不同，它是一个布尔值。 默认情况下，文件名补全会检查路径的所有组件，以查看是否存在该组件的补全，即使该组件与现有目录相匹配。 例如，当补全在 /usr/bin/ 之后时，函数会检查 /usr 的可能补全。

当该样式为 'true' 时，任何与现有目录匹配的路径前缀都会被接受，而不会尝试进一步补全。 因此，在给出的示例中，路径 /usr/bin/ 被立即接受，并尝试在该目录中补全。

这种样式在处理引用后自动出现的目录时也很有用，比如 ZFS 的 .zfs 目录或 NetApp 的 .snapshot 目录。当设置了这种样式时，shell 不会检查父目录中的目录是否存在。

如果希望完全禁止这种行为，请将 path-completion 样式（见下文）设置为 'false'。

add-space

该样式由 _expand 补全器使用。 如果值为 'true'（默认值），则会在扩展后的所有单词后插入空格，如果是目录名，则会插入斜线。 如果该值为 'file'，则补全器只会在现有文件名后添加空格。 布尔值 'true' 或值 'file' 都可以与 'subst' 结合使用，在这种情况下，补全器不会在形式为 '$(...)' 或 '${...}' 的替换扩展产生的单词后添加空格。

_prefix 补全器使用这种样式作为一个简单的布尔值，来决定是否在后缀前插入空格。

ambiguous

这适用于补全文件名路径中的非最终组件，换句话说，就是那些带有尾部斜线的组件。 如果设置了该样式，即使正在使用菜单补全，光标也会留在第一个含混的组件之后。 该样式始终使用 paths 标记进行测试。

assign-list

在等号后补全赋值时，补全系统通常只补全一个文件名。 在某些情况下，该值可能是由冒号分隔的文件名列表，如 PATH 和类似参数。 这种样式可以设置为与此类参数名称相匹配的模式列表。

默认情况下，当一行中的单词已经包含冒号时，列表会补全。

auto-description

如果设置了该样式，则该样式的值将用作未被补全函数描述但有一个参数的选项的描述。 值中的序列 '%d' 将被该参数的描述所取代。 根据个人喜好，将该样式设置为类似于 'specify: %d'。 请注意，这对某些命令可能不起作用。

avoid-completer

_all_matches 补全器使用它来决定是否将所有匹配字符串添加到当前生成的列表中。 它的值是一个补全器名称列表。 如果其中任何一个是在此补全中生成匹配结果的补全器名称，则不会添加该字符串。

该样式的默认值为 '_expand _old_list _correct _approximate'，也就是说，该样式包含的补全器几乎永远不会有匹配的字符串。

cache-path

该样式定义了包含转储补全数据的缓存文件的存储路径。 默认路径为 '$ZDOTDIR/.zcompcache'，如果未定义 $ZDOTDIR，则默认路径为 '$HOME/.zcompcache'。 除非设置了 use-cache 样式，否则不会使用补全缓存。

cache-policy

该样式定义了用于确定缓存是否需要重建的函数。 请参阅下面有关 _cache_invalid 函数的部分。

call-command

这种风格用于函数中的命令，如make和ant，在这些情况下，直接调用命令来生成匹配项会遇到一些问题，例如速度慢，或者像在make的情况下，可能会导致 makefile 中的操作被执行。如果它被设置为 'true'，则会调用命令来生成匹配项。这种风格的默认值是false。

command

在许多地方，补全函数需要调用外部命令来生成补全列表。 在这种情况下，可以使用这种样式来覆盖调用的命令。 值中的元素用空格连接，形成要执行的命令行。 该值也可以以连字符开头，在这种情况下，通常的命令会被添加到最后；这对于在前面加上 'builtin' 或 'command'，以确保调用的是命令的适当版本（例如，避免调用与外部命令同名的 shell 函数）最为有用。

举例来说，进程 ID 的补全函数使用这种样式和 processes 标记来生成要补全的 ID 和要显示的进程列表（如果 verbose 样式为 'true'）。 该命令生成的列表应与 ps 命令的输出相似。 第一行不会显示，但会搜索字符串 'PID'（或 'pid'），以确

定进程 ID 在下面各行中的位置。 如果该行不包含 'PID'，则其他各行中的第一个数字将作为进程 ID 用来补全。

需要注意的是，每次尝试生成补全列表时，补全函数一般都要调用指定的命令。 因此，应注意只指定运行时间较短的命令，尤其要避免任何可能永远不会终止的命令。

`command-path`

这是要搜索补全命令的目录列表。 该样式的默认值是特殊参数 `path` 的值。

`commands`

用于补全系统初始化脚本的子命令的函数（位于 `/etc/init.d` 或其他位置）。 对于补全函数无法自动找出的命令，它的值给出了要补全的默认命令。 该样式的默认值是两个字符串 'start' 和 'stop'。

`complete`

当作为可绑定命令调用时，`_expand_alias` 函数将使用此值。 如果设置为 'true'，且命令行中的单词不是别名的名字，则将补全与别名匹配的名字。

`complete-options`

这是 `cd`、`chdir` 和 `pushd` 的补全器使用的。 对于这些命令，`-` 用于引入目录堆栈条目，补全这些命令远比补全选项更常见。 因此，除非该样式的值为 'true'，否则即使在初始化 `-` 后，也不会补全选项。 如果该样式的值为 'true'，选项将在初始化 `-` 之后补全，除非命令行中有`--`。

`completer`

作为该样式值的字符串提供了要使用的补全函数名称。控制函数 中描述了可用的补全函数。

每个字符串既可以是补全函数的名称，也可以是形式为 '*function*:*name*' 的字符串。 在第一种情况下，上下文的 *completer* 字段将包含补全器的名称，但不包括前导下划线，其他下划线均由连字符代替。 在第二种情况下，*function* 是要调用的完成器的名称，但上下文的 *completer* 字段中将包含用户定义的 *name*。 如果 *name* 以连字号开头，则上下文字符串将与第一种情况一样，由包含 *name* 的补全函数名称追加构建而成。 例如：

```
zstyle ':completion:*' completer _complete _complete:-foo
```

在这里，补全将调用 `_complete` 补全函数两次，一次是使用 'complete'，另一次是在上下文的 *completer* 字段中使用 'complete-foo'。 通常，只有在与 '*functions*:*name*' 形式一起使用时，多次使用同一补全器才有意义，否则在所有对补全器的调用中，上下文名称都将相同；`_ignored` 和 `_prefix` 补全器可能是这一规则的例外。

该样式的默认值为 '_complete _ignored'：只完成补全，首先使用 ignored-patterns 样式和 $fignore 数组，然后在不忽略匹配的情况下进行。

condition

该样式被 _list 补全器函数用于决定是否应无条件延迟插入匹配项。默认为 'true'。

delimiters

该样式用于添加分隔符，以便与历史修饰符或 glob 限定符一起使用(具有分隔的参数)。 它是一个要添加的首选分隔符数组。 首选非特殊字符，否则补全系统可能会混淆。 默认列表为 :、+、/、-、%。 该列表可以为空，以强制输入分隔符。

disabled

如果设置为 'true'，_expand_alias 补全器和可绑定命令也将尝试扩展禁用的别名。 默认值为 'false'。

domains

用于补全的网络域名列表。 如果未设置，域名将取自 /etc/resolv.conf 文件。

environ

environ 样式用于补全 'sudo'。 在调用目标命令的补全之前，它将被设置为 '*VAR*=*value*' 赋值的数组，并导出到本地环境中。

```
zstyle ':completion:*:sudo::' environ \
  PATH="/sbin:/usr/sbin:$PATH" HOME="/root"
```

expand

这种样式用于补全由多个部分（如路径名）组成的字符串。

如果其值之一是字符串 'prefix'，则即使尾部无法完成，也会尽可能扩展该行的部分输入字。

如果其值之一是字符串 'suffix'，则在第一个模棱两可的名称之后也会添加匹配的组件名称。 这意味着生成的字符串可能是最长的无歧义字符串。 不过，可以使用菜单补全来循环查看所有匹配结果。

extra-verbose

如果设置，则补全列表会更加冗长，但可能会降低补全速度。 如果将此样式设置为 'true'，补全性能将受到影响。

fake

可为任何补全上下文设置该样式。 它指定了在该上下文中将始终补全的附加字符串。 每个字符串的形式为 '*value*:*description*'；冒号和描述可以省略，但 *value* 中的任何字面冒号必须用反斜杠引出。 提供的任何 *description* 都会与值一起显示在补全列表中。

在指定虚假字符串时，必须使用足够严格的上下文。 请注意，fake-files 和 fake-parameters 样式可在补全文件或参数时提供额外功能。

fake-always

除了不使用 ignored-patterns 样式外，该样式的作用与 fake 样式相同。 这样，通过将忽略模式设置为 '*'，就可以完全覆盖一组匹配。

下面的示例展示了一种用任意数据补充任何标记的方法，但在显示时，这些数据的表现与单独的标记无异。 在本例中，我们使用 tag-order 样式的功能，在使用标准补全器 complete 对 cd 的参数进行补全时，将 named-directories 标签分为两个。 标签 named-directories-normal 的行为与正常一样，但标签 named-directories-mine 包含一组固定的目录。 这样做的效果是，使用给定的补全添加匹配组 'extra directories'。

```
zstyle ':completion::complete:cd:*' tag-order \
  'named-directories:-mine:extra\ directories
  named-directories:-normal:named\ directories *'
zstyle ':completion::complete:cd:*:named-directories-mine' \
  fake-always mydir1 mydir2
zstyle ':completion::complete:cd:*:named-directories-mine' \
  ignored-patterns '*'
```

fake-files

该样式用于补全文件和查找无标记的文件。 其值的形式为 '*dir*:*names...*'。 这将在目录 *dir* 中补全时添加 *names*（用空格分隔的字符串）作为可能的匹配项，即使确实不存在这样的文件。 dir 可以是一个模式；*dir* 中的模式字符或冒号应使用反斜杠引出，以便按字面意思处理。

这在支持特殊文件系统的系统中非常有用，这些系统的顶层路径名无法用 glob 模式列出或生成（但请参阅 accept-exact-dirs，了解处理这一问题的更普遍方法）。 它还可用于没有读取权限的目录。

模式形式可用于在特定文件系统的所有目录中添加某个 '魔法' 条目。

fake-parameters

补全函数将其用于参数名称。 其值是可能尚未设置但仍应补全的参数名称。 每个名称后面还可以跟一个冒号和一个字符串，该字符串指定了参数的类型（如 'scalar'、'array' 或 'integer'）。 如果给出了类型，只有在特定上下文中需要该类型的参数时，才会补全名称。 未指定类型的名称将始终补全。

file-list

该样式控制使用标准内置机制补全的文件是否以类似 `ls -l` 的长列表形式列出。 需要注意的是，该功能使用 shell 模块 `zsh/stat` 来获取文件信息；这将加载内置的 `stat`，从而取代任何外部 `stat` 可执行文件。 为避免这种情况，可在初始化文件中加入以下代码：

```
zmodload -i zsh/stat
disable stat
```

样式既可以设置为 'true' 值（或 'all'），也可以设置为 'insert' 或'list' 值之一，表示在任何情况下都将以长格式列出文件，或在尝试插入文件名时，或在列出文件名但不尝试插入文件名时。

一般来说，该值可以是由上述任意值组成的数组，可选择在其后加上 *=num*。 如果存在 *num*，则表示将使用长列表样式的最大匹配次数。 例如

```
zstyle ':completion:*' file-list list=20 insert=10
```

指定在列出多达 20 个文件或插入多达 10 个匹配文件时使用长格式（假设要显示列表，例如在模棱两可的补全时），否则将使用短格式。

```
zstyle -e ':completion:*' file-list \
       '(( ${+NUMERIC} )) && reply=(true)'
```

指定在提供数字参数时使用长格式，否则使用短格式。

file-patterns

用于补全文件名的标准函数 `_files`。 如果未设置样式，则会根据 `_files` 的调用者所期望的文件类型，提供三种标记：'globbed-files','directories' 和 'all-files'。 前两种（'globbed-files' 和 'directories'）通常一起提供，以方便补全子目录中的文件。

`file-patterns` 样式提供了默认标记的替代方案，默认标记不会被使用。 其值由形式为 '*pattern*:*tag*' 的元素组成；每个字符串可包含任意数量的此类规范，并用空格分隔。

pattern 是用于生成文件名的模式。 任何出现的序列 '%p' 都会被调用 `_files` 的函数所传递的任何模式所替换。 模式中的冒号必须在前面加上反斜杠，以便与 *tag* 前面的冒号区分开来。 如果需要多个模式，可以将模式放在大括号内，中间用逗号隔开。

值中所有字符串的 *tag*s 将由 `_files` 提供，并在查找其他样式时使用。 同一单词中的任何 *tag*s 都将同时提供，并在后面的单词之前提供。 如果没有给出 ':*tag*' ，则将使用 'files' 标记。

在 *tag* 后面还可以可选择的跟着第二个冒号和描述，描述将用于 format 样式值中的 '%d'（如果已设置），而不是补全函数提供的默认描述。 包含描述也会优先于相关选项，如补全分组，因此它可以用于需要分隔文件的地方。

例如，要使 rm 命令首先只补全对象文件的名称，然后在没有匹配的对象文件时补全所有文件的名称：

```
zstyle ':completion:*:*:rm:*:*' file-patterns \
    '*.o:object-files' '%p:all-files'
```

改变文件补全的默认行为 — 第一次尝试时，提供与模式和目录匹配的文件，然后提供所有文件 — 改为第一次尝试时，只提供匹配的文件，然后提供目录，最后提供所有文件：

```
zstyle ':completion:*' file-patterns \
    '%p:globbed-files' '*(-/):directories' '*:all-files'
```

即使在没有特殊模式的情况下，也能正常工作： _files 第一步会匹配所有使用 '*' 模式的文件，并在看到该模式时停止。 还要注意的是，它不会在一次补全尝试中多次尝试一个模式。

若要将目录与文件分隔成一组，但仍能在第一次尝试时补全，则需要给出说明。 请注意，目录需要明确地从 globbed-files 中排除，因为 '*' 将匹配目录。对于分组，还需要设置 group-name 样式。

```
zstyle ':completion:*' file-patterns \
    '%p(^-/):globbed-files *(-/):directories:location'
```

在执行补全函数期间，EXTENDED_GLOB 选项有效，因此字符 '#', '~' 和 '^' 在模式中具有特殊含义。

file-sort

标准文件名补全函数使用这种不带标记的样式来确定名称的排列顺序；菜单补全将以相同的顺序循环显示这些名称。 可能的值有: 'size',按文件大小排序；'links'，按文件链接数排序；'modification'（或 'time' 或 'date'）,按最后修改时间排序；'access',按最后访问时间排序；'inode'（或 'change'）,按最后 inode 更改时间排序。 如果样式被设置为其他值或未设置，文件将按名称的字母顺序排序。 如果值包含字符串 'reverse'，则按相反顺序排序。 如果值包含字符串 'follow'，时间戳将与符号链接的目标相关联；默认情况下使用链接本身的时间戳。

file-split-chars

一组字符，会导致 **所有** 文件补全在出现其中任何一个字符时被分割。 典型的用法是将样式设置为：；这样，在补全文件时，字符串中直到最后一个：在内的所有内容都将被忽略。 由于这种做法比较粗暴，通常最好是更新补全函数，以适应这种有用的行为。

`filter`

电子邮件地址补全的 `ldap` 插件（请参阅 `_email_addresses`）在过滤条目时使用该样式指定匹配的属性。 例如，如果样式设置为 'sn'，则会根据姓氏进行匹配。 将使用标准 LDAP 过滤，因此绕过了正常的补全匹配。 如果未设置此样式，则会跳过 LDAP 插件。 您可能还需要设置 `command` 样式，以指定如何连接到 LDAP 服务器。

`force-list`

这将强制在进行列表时显示补全列表，即使在通常不显示补全列表的情况下也是如此。 例如，通常只有在至少有两个不同的匹配项时才会显示列表。 将该样式设置为 'always' 后，列表将始终显示，即使只有一个匹配项会立即被接受。 样式也可以设置为数字。 在这种情况下，如果至少有这么多匹配项，即使所有匹配项都插入相同的字符串，列表也会显示。

该样式针对默认标记以及当前补全时有效的每个标签进行测试。 因此，可以只对某些类型的匹配强制应用列表。

`format`

如果为 `descriptions` 标记设置了此项，其值将作为字符串在补全列表中显示以上匹配项。 该字符串中的 '%d' 序列将被替换为这些匹配项的简短描述。 该字符串还可能包含 `compadd -X` 所理解的输出属性序列（参见 补全小部件）。

在对 `descriptions` 标记进行测试之前，会对当前补全时有效的每个标记进行样式测试。 因此，可以为不同类型的匹配定义不同的格式字符串。

还要注意的是，某些补全函数定义了额外的 '%' 序列。 我们将对使用这些序列的补全函数进行说明。

某些补全函数显示的信息可以通过为 `messages` 标记设置这种样式来定制。 在这里，'%d' 会被替换为补全函数给出的信息。

最后，使用 `warnings` 标记查找格式字符串，以便在完全无法生成匹配结果时使用。 在这种情况下，'%d' 会被替换为预期匹配的描述，并用空格分隔。 序列 '%D' 会被替换为同样的描述，并用换行符隔开。

可以使用printf 风格的字段宽度指定符 '%d' 和类似的转义序列。 这由 `zsh/zutil` 模块中的 `zformat` 内置命令处理，参见 zsh/zutil 模块。

`gain-privileges`

如果设置为 `true`，该样式将允许使用 `sudo` 或 `doas` 等命令来获取额外权限，以检索补全信息。只有当 `sudo` 等命令出现在命令行中时，才会这样做。要强制使用 `sudo` 等命令，或覆盖因 `gain-privileges` 而可能添加的任何前缀，可以使用 `command` 样式和以连字符开头的值。

glob

由 _expand 补全器使用。 如果将其设置为 'true'（默认值），则将尝试对之前替换（参见 substitute 样式）后的字词进行 globbing，否则将使用该行的原始字符串。

global

如果设置为 'true'（默认），_expand_alias 补全器和可绑定命令将尝试扩展全局别名。

group-name

补全系统可以将不同类型的匹配分组，分别出现在不同的列表中。 这种样式可用于为特定标记提供分组名称。 例如，在命令位置，补全系统会生成内置和外部命令名称、别名、shell 函数和参数以及保留字作为可能的补全。 若要单独列出外部命令和 shell 函数，可使用以下方法：

```
zstyle ':completion:*:*:-command-:*:commands' \
       group-name commands
zstyle ':completion:*:*:-command-:*:functions' \
       group-name functions
```

因此，任何具有相同标记的匹配都会显示在同一组中。

如果给定的名称是空字符串，匹配标记的名称将被用作组的名称。 因此，要分别显示所有不同类型的匹配信息，只需设置

```
zstyle ':completion:*' group-name ''
```

所有未定义组名的匹配结果都将被归入名为 -default- 的组中。

要在输出中显示组名，请参阅 descriptions 标记下的 format 样式（q.v.）。

group-order

该样式是 group-name 样式的附加样式，用于指定该样式所定义的组的显示顺序（比较 tag-order，它决定显示哪些补全）。 已命名的组按指定顺序显示；其他任何组则按补全函数定义的顺序显示。

例如，让内置命令、shell 函数和外部命令的名称在命令位置补全时按顺序显示：

```
zstyle ':completion:*:*:-command-:*:*' group-order \
       builtins functions commands
```

groups

UNIX 组名称列表。 如果未设置，则组名取自 YP 数据库或文件 '/etc/group'。

hidden

如果设置为 'true'，则不会列出给定上下文的匹配项，但会显示使用 format 样式设置的匹配项的描述。 如果设置为 'all'，则连描述也不会显示。

请注意，这些匹配项仍然会被补全，只是不会显示在列表中。 为了避免将匹配项视为可能的补全项，可以按下文所述修改 tag-order 样式。

hosts

应补全的主机名列表。 如果未设置，主机名将取自文件 '/etc/hosts'。

hosts-ports

这种样式适用于需要或接受主机名和网络端口的命令。 值中的字符串格式应为 '*host*:*port*'。 有效端口由主机名决定；同一主机可能有多个端口。

ignore-line

这将为当前补全有效的每个标记进行测试。 如果设置为 'true'，则该行中已经存在的单词都不会被视为可能的补全。 如果设置为 'current'，光标所在的单词将不被视为可能的补全。 值 'current-shown' 与之类似，但只适用于当前屏幕上显示补全列表的情况。 最后，如果样式设置为 'other'，则该行中除当前单词外的所有单词都将被排除在可能的补全之外。

'current' 和 'current-shown' 有点像 accept-exact 样式的反义词：只有字符缺失的字符串才会被补全。

需要注意的是，在一般情况下，比如 ':completion:*' ，你几乎肯定不想将其设置为 'true' 或 'other' 。 这是因为，即使相关命令不止一次接受选项，它也会禁止多次补全选项。

ignore-parents

补全路径名的函数会在没有标记的情况下对样式进行测试，以确定是忽略当前单词中已提及的目录名，还是忽略当前工作目录名。 该值必须包括以下字符串中的一个或两个：

parent

如果该行中的单词已包含任何目录的路径，则该目录的名称将被忽略。 例如，在 foo/../ 之后补全时，目录 foo 将不被视为有效的补全。

pwd

当前工作目录的名称不会被补全；因此，例如，在 ../ 之后补全将不会使用当前目录的名称。

此外，该值还可包括以下一项或两项：

..

仅当一行中的单词包含子字符串 '../' 时，才忽略指定目录。

directory

仅在补全目录名时忽略指定目录，而不在补全文件名时忽略指定目录。

排除值的作用方式与 ignored-patterns 样式的值类似，因此可以通过 _ignored 补全器恢复对它们的考虑。

ignored-patterns

模式列表；任何与其中一个模式匹配的试验补全都将被排除在外。 _ignored 补全器可以出现在补全器列表中，以恢复被忽略的匹配。 这是 shell 参数 $fignore 的可配置版本。

请注意，EXTENDED_GLOB 选项是在执行补全函数期间设置的，因此字符 '#', '~' 和 '^' 在模式中具有特殊含义。

insert

_all_matches 补全器使用这种样式来决定是否无条件插入所有匹配项的列表，而不是将列表作为另一个匹配项添加。

insert-ids

在填写进程 ID 时，例如作为 kill 和 wait 内置程序的参数，命令名称可能会被转换为相应的进程 ID。 如果键入的进程名称不是唯一的，就会出现问题。 默认情况下（或将此样式显式设置为 'menu'），名称会立即转换为一组可能的 ID，并启动菜单补全以循环浏览这些 ID。

如果样式的值为'single'，shell 将等待用户键入足够多的字符，使命令具有唯一性后才将名称转换为 ID；在此之前，补全尝试将不会成功。 如果值是任何其他字符串，当用户键入的字符串长于相应 ID 的通用前缀时，将启动菜单补全。

insert-sections

在补全手册页面名称时，该样式与格式为 'manuals.*X*' 的标记一起使用。如果设置了该样式，且标签名称中的 *X* 与所补全页面的章节编号一致，则章节编号将与页面名称一起插入。例如，

```
zstyle ':completion:*:manuals.*' insert-sections true
```

man ssh_<TAB> 可被补全为 man 5 ssh_config。

该值也可以设置为 'prepend' 或 'suffix' 之一。 在上例中，'prepend' 的行为与 'true' 相同，而 'suffix' 则会将 man ssh_<TAB> 补全为 man ssh_config.5。

这一点与 separate-sections 配合使用尤其有用，因为它可以确保在有多个同名页面时（例如 printf(1) 和 printf(3)），man 请求的页面与补全列表中显示的页面一致。

该样式的默认值为 'false'。

insert-tab

如果设置为 'true'，当光标左侧没有非空白字符时，补全系统将插入一个 TAB 字符（假设该字符用于开始补全），而不是执行补全。 如果设置为 'false'，即使在这种情况下也会执行补全。

值也可能包含子串 'pending' 或 'pending=*val*'。 在这种情况下，当有未处理的输入待处理时，将插入键入的字符，而不是开始补全。 如果给定了 *val*，那么如果至少有这么多字符的未处理输入，就不会补全。 这在向终端粘贴字符时非常有用。 但需要注意的是，它依赖于 zsh/zle 模块中的 $PENDING 特殊参数的正确设置，而这并非在所有平台上都能保证。

该样式的默认值为 'true' ，但在 vared 内置命令中补全时为 'false'。

insert-unambiguous

这被 _match 和 _approximate 补全器使用。 这些补全器通常与菜单补全一起使用，因为键入的单词可能与最终的补全几乎没有相似之处。 但是，如果该样式为 'true'，则补全器只有在找不到至少与用户键入的原始字符串一样长的明确初始字符串时，才会启动菜单补全。

如果使用 _approximate 补全器，上下文中的补全器字段将被设置为 correct-*num* 或 approximate-*num*，其中 *num* 是已接受的错误数。

对于 _match 补全器，样式也可以设置为字符串 'pattern'。 如果该行上的模式不能明确匹配，则保持不变。

keep-prefix

该样式由 _expand 补全器使用。 如果为 'true'，补全器将尝试保留包含转折号(tilde)或参数扩展的前缀。 例如，字符串 '~/f*' 将扩展为 '~/foo' ，而不是 '/home/user/foo'。 如果样式设置为 'changed'（默认），只有在扩展后的单词与命令行中的原始单词之间有其他变化时，前缀才会保持不变。 任何其他值都会强制无条件扩展前缀。

这个样式为 'true' 时，_expand 的行为是，当恢复的前缀的单个扩展与原始扩展相同时，_expand 会放弃；因此，可以调用任何剩余的补全器。

known-hosts-files

该样式应包含一个文件列表，用于搜索与 ssh known_hosts 文件格式兼容的主机名和 IP 地址（如果设置了 use-ip 样式）。 如果未设置，则使用 /etc/ssh/ssh_known_hosts 和 ~/.ssh/known_hosts 文件。

last-prompt

这是 ALWAYS_LAST_PROMPT 选项的一种更灵活的形式。 如果该选项为 'true'，补全系统将尝试在显示补全列表后将光标返回到前一个命令行。 它会对当前补全有效的所有标记进行测试，然后是 default 标记。 如果该样式对所有类型的匹配都是 'true'，光标将被移回上一行。 需要注意的是，与 ALWAYS_LAST_PROMPT 选项不同，此样式与数字参数无关。

list

该样式由 _history_complete_word 可绑定命令使用。 如果设置为 'true'，则没有任何效果。 如果设置为 'false'，则不会列出匹配结果。 这将覆盖控制列表行为的选项设置，尤其是 AUTO_LIST。 上下文总是以 ':completion:history-words' 开头。

list-colors

如果已加载 zsh/complist 模块，则可使用此样式来设置颜色规格。 这种机制取代了 zsh/complist 模块中 ZLS_COLORS 和 ZLS_COLOURS 参数的使用，但语法相同。

如果为 default 标记设置了这种样式，那么值中的字符串将被视为在任何地方都要使用的规范。 如果为其他标记设置了该样式，则仅在该标记描述的匹配类型中使用规范。 为达到最佳效果，必须将 group-name 样式设置为空字符串。

除了为特定标记设置样式外，还可以使用 group-name 标记明确指定的组名，以及 ZLS_COLORS 和 ZLS_COLOURS 参数允许的 '(group)' 语法，并简单地使用 default 标记。

可以使用已为 GNU 版本 ls 命令设置的任何颜色规格：

```
zstyle ':completion:*:default' list-colors \
       ${(s.:.)LS_COLORS}
```

默认颜色与 GNU ls 命令相同，可以通过将样式设置为空字符串（即 ''）来获得。

list-dirs-first

文件补全时使用，与 file-patterns 样式的特定设置相对应。 如果设置了该样式，则要补全的默认目录将与其他文件分开列出，并先于其他文件补全。

list-grouped

如果该样式为 'true'（默认），补全系统将尝试通过分组匹配使某些补全列表更紧凑。 例如，具有相同描述的命令选项（在 verbose 样式设置为 'true' 时显示）将显示为一个条目。 不过，可以使用菜单选择来循环浏览所有匹配项。

list-packed

这将针对当前上下文中有效的每个标记以及 default 标记进行测试。 如果设置为 'true'，则相应的匹配项会出现在列表中，就像设置了 LIST_PACKED 选项一样。 如果设置为 'false'，则会正常列出。

list-prompt

如果为 default 标记设置了该样式，则无法在屏幕上显示的补全列表可以滚动显示（参见 zsh/complist 模块）。 如果该值不是空字符串，则会在每次刷屏后显示，并且 shell 会提示按键；如果样式设置为空字符串，则会使用默认提示符。

该值可能包含转义序列：'%l' 或 '%L'，它将被最后显示的行数和总行数取代；'%m' 或 '%M'，它将被最后显示的匹配数和总匹配数取代；以及 '%p' 和 '%P'，当位于列表开头时为 'Top'，位于列表结尾时为 'Bottom'，否则将以总长度的百分比显示位置。 在每种情况下，大写字母形式将被固定宽度的字符串替换，并在右侧填充空格，而小写字母形式将被宽度可变的字符串替换。 与其他提示符字符串一样，用于进入和离开显示模式的转义序列 '%S', '%s', '%B', '%b', '%U', '%u' (突出,粗体，下划线)，此外，还有用于改变前景背景颜色的 '%F', '%f', '%K', '%k' ,和"%{"形式。用于括起来的 '%{...%}' 转义序列，其显示宽度为零（或用数字参数表示为其他宽度）。

删除此提示符后，应取消设置变量 LISTPROMPT ,才能生效。

list-rows-first

该样式的测试方法与 list-packed 样式相同，并决定是否以行优先的方式列出匹配结果，就像设置了 LIST_ROWS_FIRST 选项一样。

list-separator

该样式的值在补全列表中用于在可能的情况下（如补全选项时）分隔要补全的字符串和描述。 默认值为 '--'（两个连字符）。

list-suffixes

该样式由补全文件名的函数使用。 如果该样式为 'true'，并且在尝试补全键入的包含多个部分路径名组件的字符串时，将显示所有不明确的组件。 否则，补全将在第一个含糊的部分停止。

local

用于补全 URL 的函数，这些 URL 的相应文件可直接从文件系统中获取。 其值应由三个字符串组成：主机名、服务器默认网页的路径，以及用户在其主页区域放置网页时使用的目录名。

例如：

```
zstyle ':completion:*' local toast \
    /var/http/public/toast public_html
```

在 'http://toast/stuff/' 后补全将查找 /var/http/public/toast/stuff 目录中的文件，而在 'http://toast/~yousir/' 后补全将查找 ~yousir/public_html 目录中的文件。

mail-directory

如果设置了该选项，zsh 会假定邮箱文件在指定的目录中。 默认值为 '~/Mail'。

match-original

这是 _match 补全器使用的。 如果设置为 only，_match 将尝试生成匹配,但不在光标位置插入 '*'。 如果设置为任何其他非空值，它将首先在不插入 '*' 的情况下尝试生成匹配结果，如果没有匹配结果，它将在插入 '*' 的情况下再次尝试。 如果未设置或设置为空字符串，则只在插入 '*' 时执行匹配。

匹配器

该样式对当前上下文中有效的每个标记分别进行测试。 它的值被置于 matcher-list 样式给出的任何匹配规范之前，因此可以通过使用 x: 规范来覆盖这些规范。 该值应采用 补全匹配控制 中描述的形式。 有关示例，请参阅 tag-order 样式的说明。

关于该样式注意与 matcher-list 样式的使用进行比较，请参阅 tag-order 样式的说明。

matcher-list

该样式可设置为匹配规范的列表，这些匹配规范将应用于所有地方。补全匹配控制 中描述了匹配规范。 补全系统将针对所选的每个补全器逐一尝试匹配规范。 例如，首先尝试简单补全，如果没有匹配结果，则尝试不区分大小写的补全：

```
zstyle ':completion:*' matcher-list '' 'm:{a-zA-Z}={A-Za-z}'
```

默认情况下，每一条规范都会取代前一条规范；但是，如果一条规范的前缀是 +，它就会被添加到现有的列表中。 因此，可以在不重复的情况下创建越来越多的通用规范：

```
zstyle ':completion:*' matcher-list \
       '' '+m:{a-z}={A-Z}' '+m:{A-Z}={a-z}'
```

通过使用上下文的第三个字段，可以创建对特定补全程序有效的匹配规范。 这仅适用于覆盖全局匹配器列表的补全程序，截至本文撰写时，全局匹配器列表仅包括

_prefix 和 _ignored。 例如，要使用 _complete 和 _prefix，但只允许使用 _complete 进行大小写不敏感的补全：

```
zstyle ':completion:*' completer _complete _prefix
zstyle ':completion:*:complete:*:*:*' matcher-list \
       '' 'm:{a-zA-Z}={A-Za-z}'
```

用户可自定义名称，如 completer 样式所解释的那样。 这样就可以多次使用同一个补全器，每次使用不同的匹配规范。 例如，先尝试不带匹配规范的正常补全，再尝试带大小写不敏感匹配的正常补全，然后是纠错，最后是部分词补全：

```
zstyle ':completion:*' completer \
    _complete _correct _complete:foo
zstyle ':completion:*:complete:*:*:*' matcher-list \
    '' 'm:{a-zA-Z}={A-Za-z}'
zstyle ':completion:*:foo:*:*:*' matcher-list \
    'm:{a-zA-Z}={A-Za-z} r:|[-_./]=* r:|=*'
```

如果在任何上下文中未设置样式，则不会应用匹配规范。 还需注意的是，某些补全器（如 _correct 和 _approximate）根本不使用匹配规范，不过即使 matcher-list 包含多个元素，这些补全器也只会被调用一次。

在使用多个规范时，请注意 **整个** 补全过程是针对 matcher-list 的每个元素进行的，这会迅速降低 shell 的性能。 根据粗略的经验，一到三个字符串就能提供可接受的性能。 另一方面，将多个以空格分隔的值放入同一字符串不会对性能产生明显影响。

如果当前没有匹配器或匹配器为空，且选项 NO_CASE_GLOB 有效，则在任何情况下都将执行不区分大小写的文件匹配。 不过，如果需要，任何匹配器都必须明确指定大小写不敏感匹配。

关于该样式与 matcher 样式的使用比较，请参阅 tag-order 样式的说明。

max-errors

_approximate 和 _correct 补全函数使用它来确定允许的最大错误数。 补全器在生成补全时，会先允许一个错误，然后是两个错误，以此类推，直到找到一个或多个匹配项，或者达到该样式给出的最大错误数。

如果该样式的值包含字符串 'numeric'，则补全函数会将任何数字参数作为允许的最大错误数。例如

```
zstyle ':completion:*:approximate:::' max-errors 2 numeric
```

如果没有给出数值参数，则允许出现两个错误，但如果数值参数为 6（如 'ESC-6 TAB'），则最多可接受六个错误。 因此，如果值为'0 numeric'，除非给出一个数字参数，否则不会尝试纠正补全。

如果值包含字符串 'not-numeric'，则在给定数字参数时，补全程序将 **不会** 尝试生成更正补全，因此在这种情况下，给定的数字应大于零。 例如，'2 not-numeric' 表示通常会对两个错误进行纠正补全，但如果给出的是数字参数，则不会进行纠正补全。

这个样式的默认值是 '2 numeric'.

max-matches-width

当 verbose 样式生效时，该样式用于确定匹配内容显示宽度与匹配内容描述宽度之间的权衡。 该值给出了为匹配预留的显示列数。 默认值为屏幕宽度的一半。

当多个匹配具有相同的描述，会被分组在一起，这一点影响最大。 增大样式可以将更多的匹配归为一组；减小样式可以显示更多的描述。

menu

如果此值为 'true' ，则将在当前补全定义的任何标记上下文中使用菜单补全。 特定标记的值优先于 'default' 标记的值。

如果通过这种方法找到的值都不是 'true' ，但至少有一个值被设置为 'auto'，那么 shell 的行为就如同设置了 AUTO_MENU 选项。

如果其中一个值被明确设置为 'false'，则菜单补全将被明确关闭，并覆盖 MENU_COMPLETE 选项和其他设置。

在 'yes=*num*' 形式中， 'yes' 可以是任何一个 'true' 值 ('yes', 'true', 'on' 和 '1') ，如果至少有 *num* 个匹配项，则将打开菜单补全功能。 在 'yes=long' 形式中，如果列表无法显示在屏幕上，则将打开菜单补全功能。 如果小部件通常只列出补全信息，则不会激活菜单补全，但在这种情况下，可以使用值 'yes=long-list' 激活菜单补全 (通常情况下，后面介绍的值 'select=long-list' 更有用，因为它提供了对滚动的控制) 。

同样，如果使用任何 'false' 值 (如 'no=10') ，如果有 *num* 或更多个匹配项，将 **不会** 使用完成菜单。

该小部件的值还可以控制菜单选择，由 zsh/complist 模块实现。 以下值可以与上述值同时出现，也可以代替上述值。

如果值包含字符串 'select'，则将无条件启动菜单选择。

在 'select=*num*' 形式中，只有至少有 *num* 个匹配时，才会启动菜单选择。 如果不止一个标记的值提供了一个数字，则取最小的数字。

通过定义包含字符串 'no-select' 的值，可以明确关闭菜单选择功能。

还可以使用值 'select=long'，仅在匹配列表无法在屏幕上显示时启动菜单选择。 要在当前小部件只执行列表功能的情况下启动菜单选择，可使用值 'select=long-list'。

当有一定数量的匹配项 **或** 匹配项列表无法在屏幕上显示时，打开菜单补全或菜单选择功能，则可以同时给出两次 'yes=' 和 'select=' ，一次是数字，另一次是 'long' 或 'long-list'。

最后，还可以激活两种特殊的菜单选择模式。 值中包含 'interactive' 会导致在开始选择菜单时立即进入交互模式；有关交互模式的描述，请参阅 zsh/complist 模块。 包含字符串 'search' 增量搜索模式中行为相同。 要选择后向增量搜索，请加入字符串 'search-backward'。

muttrc

如果设置，则给出 mutt 配置文件的位置。 默认为 '~/.muttrc'。

numbers

与 jobs 标记一起使用。 如果值为 'true'，shell 将补全作业编号，而不是作业命令文本中最短的无歧义前缀。 如果值为数字，则只有在需要从作业描述中找出这么多字来解决歧义时，才会使用作业编号。 例如，如果值为 '1'，则只有在所有作业命令行的第一个单词不同时，才会使用字符串。

old-list

该值由 _oldlist 补全器使用。 如果将其设置为 'always'，那么执行列表的标准小部件将保留当前的匹配列表，无论它们是如何生成的；可以使用值 'never' 显式地关闭此功能，从而在不使用 _oldlist 补全器的情况下提供相应的行为。 如果未设置样式或其他值，则会显示现有的补全列表（如果还没有）；否则，将生成标准的补全列表；这是 _oldlist 的默认行为。 不过，如果存在旧列表，且该样式包含生成该列表的补全函数名称，则将使用旧列表，即使该列表是由不做列表的小部件生成的。

例如，假设键入 ^Xc 来使用 _correct_word 小部件，它会生成光标下单词的更正列表。 通常情况下，键入 ^D 会在命令行中生成该单词的标准补全列表并显示出来。 如果使用 _oldlist，则会显示已生成的更正列表。

再以 _match 补全器为例：当 insert-unambiguous 样式设置为 'true' 时，如果有公共前缀字符串，它只会插入该字符串。 不过，这可能会删除原始模式的部分内容，因此进一步补全可能会产生比第一次更多的匹配结果。 通过使用 _oldlist 补全器并将此样式设置为 _match，将再次使用第一次尝试时生成的匹配列表。

old-matches

_all_matches 补全器使用该值来决定是否在存在匹配列表的情况下使用旧的匹配列表。 这可以通过 'true' 值或字符串 'only' 来选择。 如果值为 'only'，

_all_matches 将只使用旧的匹配列表，而不会对当前生成的匹配列表产生任何影响。

如果设置了这种样式，无条件调用 _all_matches 补全器通常是不明智的。 一种可能的用法是，使用这个样式 或使用 zstyle 的 -e 选项来定义的 completer 样式，从而使样式具有条件性。

old-menu

由 _oldlist 补全器使用。 它可以控制菜单补全在已插入补全且用户键入标准补全键（如 TAB）时的行为。 _oldlist 的默认行为是，菜单补全总是继续使用现有的补全列表。 但如果将此样式设置为 'false'，则如果旧的补全列表是由不同的补全命令生成的，则会启动新的补全；这是没有 _oldlist 补全器时的行为。

例如，假设您键入 ^Xc 来生成一个更正列表，然后菜单补全以常规方式之一启动。 通常情况下，或在此样式设置为'false' 的情况下，此时键入 TAB 将开始尝试补全现在显示的行。 如果使用 _oldlist，则会继续在更正列表中循环。

original

这被 _approximate 和 _correct 补全器用来决定是否应将原始字符串添加为可能的补全。 通常，只有在至少有两个可能的纠正时，才会添加补全，但如果此样式设置为 'true'，则总是会添加补全。 请注意，在检查该样式时，将把上下文名称中的补全器字段设置为 correct-*num* 或 approximate-*num*，其中 *num* 是接受的错误数。

packageset

该样式用于完成 Debian 的 'dpkg' 程序的参数。 它包含对给定上下文的默认软件包集的覆盖。 例如，

```
zstyle ':completion:*:complete:dpkg:option--status-1:*' \
                    packageset avail
```

会导致 'dpkg --status' 补全可用的软件包，而不仅仅是已安装的软件包。

path

补全颜色名称的函数使用这种带有 colors 标记的样式。 该值应是包含颜色名称的文件的路径名，格式为 X11 rgb.txt 文件。 如果未设置样式，但在各种标准位置中找到了该文件，则默认使用该文件。

path-completion

文件名补全时使用。 默认情况下，文件名补全会检查路径的所有组件，查看是否存在该组件的补全。 例如，/u/b/z 可以补全为 /usr/bin/zsh。 如果将此样式明确设置为 'false' ，则光标前的 / 之前的路径组件将无法执行此行为；这将覆盖 accept-exact-dirs 的设置。

即使将样式设置为 'false'，通过设置选项 COMPLETE_IN_WORD，并将光标移回到要补全的路径中的第一个组件，仍然可以补全多个路径。 例如，如果光标位于 /u 之后，则 /u/b/z 可以补全为 /usr/bin/zsh。

pine-directory

如果设置，则指定包含 PINE 邮箱文件的目录。 没有默认设置，因为递归搜索该目录对不使用 PINE 的人来说很不方便。

ports

要补全的互联网服务名称（网络端口）列表。 如果未设置，服务名称将取自文件 '/etc/services'。

prefix-hidden

用于某些有共同前缀的补全，例如以破折号开头的命令选项。 如果设置为 'true'，前缀将不会显示在匹配列表中。

这个样式的默认值是 'false'.

prefix-needed

这种样式也适用于有共同前缀的匹配。 如果设置为 'true' ，则必须由用户输入共同前缀才能生成匹配。

该样式适用于 options, signals, jobs, functions 和 parameters 等补全标记。

对于命令选项而言，这意味着在补全选项名称之前，必须明确键入前导的 '-', '+' 或 '--'。

对于信号，在补全信号名称之前，需要一个前导的 '-' 。

对于作业，在补全作业名称之前，需要输入前导的 '%'。

对于函数名和参数名，在补全以 '_' 或 '.' 字符开头的函数名或参数名之前，需要先输入前导的 '_' 或 '.' 。

对于 function 和 parameter 补全，该样式的默认值为 'false' ，否则为 'true' 。

preserve-prefix

该样式用于补全路径名。 它的值应该是一个与要补全的单词的首字母前缀相匹配的模式，在任何情况下都不应改变。 例如，在某些 Unices 中(指 UNIX 的多种变体)，开头的 '//'（双斜线）具有特殊含义；将该样式设置为字符串 '//' 将保留该含义。 再比如，在 Cygwin 下将该样式设置为 '?:/' 将允许在 'a:/...' 后补全，等等。

range

_history 补全器和 _history_complete_word 可绑定命令会使用该命令来决定哪些词应该被补全。

如果是单个数字，则只补全历史记录中最后 *N* 个字。

如果它是一个形式为 '*max*:*slice*' 的范围，则将补全最后一个 *slice* 词；如果没有匹配结果，则将尝试之前的 *slice* 词，依此类推。 这个过程会在找到至少一个匹配词或 *max* 个词已被尝试后停止。

默认设置是一次性补全历史记录中的所有单词。

recursive-files

如果设置了该样式，其值将是针对 '$PWD/' 进行测试的模式数组：请注意尾部的斜线，它允许通过在两侧包含斜线来明确划分模式中的目录。 如果普通文件补全失败，而命令行中的单词名称中还没有目录部分，那么将使用与刚才尝试的补全相同的标记来检索样式，然后依次根据 $PWD/ 对元素进行测试。 如果有一个匹配，shell 会重新尝试补全，方法是依次将命令行上的单词用 **/*(/) 扩展中的每个目录作为前缀。 通常情况下，样式元素的设置会将当前目录下的目录数量限制在可控范围内，例如 '*/.git/*'。

例如,

```
zstyle ':completion:*' recursive-files '*/zsh/*'
```

如果当前目录是 /home/pws/zsh/Src，则 zle_tr<TAB> 可以补全为 Zle/zle_tricky.c。

regular

该样式由 _expand_alias 补全器和可绑定命令使用。 如果设置为 'true'（默认），正则别名将被展开，但仅限于命令位置。 如果设置为 'false'，则不会扩展常规别名。 如果设置为 'always'，即使不在命令位置，也会展开常规别名。

rehash

如果在补全外部命令时设置了该选项，则每次搜索时都会通过 rehash 命令更新内部命令列表（哈希）。 这样做会降低速度，只有在路径中的目录文件访问速度较慢时才会明显感觉到。

remote-access

如果设置为 'false'，某些命令将无法通过互联网连接获取远程信息。 这包括 CVS 命令的补全。

我们并不总能知道连接是否确实指向远程站点，因此有些连接可能会被不必要地阻止。

remove-all-dups

_history_complete_word 可绑定命令和 _history 补全器利用这一点来决定是否要删除所有重复匹配，而不是只删除连续的重复匹配。

select-prompt

如果为 default 标记设置了该值，则当补全列表无法作为一个整体显示在屏幕上时，其值将在菜单选择时显示（参见上文的 menu 样式）。 与 list-prompt 样式相同的转义符都可以被识别，只是数字指的是标记所在的匹配或行。 当值为空字符串时，将使用默认提示符。

select-scroll

该样式针对 default 标记进行测试，并决定在菜单选择（见上文 menu 样式）时，如果补全列表无法在整个屏幕上显示时，如何滚动补全列表。 如果该值为 '0'（零），则按半屏滚动；如果该值为正整数，则按给定的行数滚动；如果该值为负数，则按给定行数的绝对值减去一屏滚动。 默认滚动行数为单行。

separate-sections

在补全手册页面名称时，该样式与 manuals 标记一起使用。 如果该样式为 'true'，不同章节的条目将使用 'manuals.*X*' 形式的标签名分别添加，其中 *X* 是章节编号。 当 group-name 样式也有效时，不同章节的页面将分别显示。 在为 dict 命令补全单词时，该样式也与 words 样式类似。它允许分别添加来自不同词典数据库的单词。另请参阅 insert-sections。

该样式的默认值为 'false'。

show-ambiguity

如果加载了 zsh/complist 模块，该样式可用于高亮显示补全列表中的第一个模糊字符。其值可以是颜色指示，如 list-colors 样式所支持的颜色；或者，如果值为 'true'，则默认选择下划线。只有在补全显示字符串与实际匹配对应时，才会应用高亮显示。

show-completer

每当尝试一个新的补全器时都会进行测试。 如果为 'true'，补全系统会在列表区输出一条进度信息，显示正在尝试的补全器。 当找到补全器时，该信息会被输出覆盖，并在补全完成后删除。

single-ignored

当只有一个匹配时，_ignored 补全器会使用该值。 如果其值为 'show'，则将显示但不插入单个匹配字符串。 如果其值为 'menu'，则单个匹配字符串和原始字符串都会被添加为匹配字符串，并启动菜单补全，从而方便选择其中任何一个。

sort

这样就可以覆盖匹配的标准排序。

如果其值为 'true' 或 'false'，则表示启用或禁用排序。 此外，还可以列出与 compadd 的 '-o' 选项相关的值： match, nosort, numeric, reverse。 如果上下文未设置该选项，则使用调用小部件的标准行为。

首先针对包含标记的完整上下文测试这个样式，如果测试失败，则针对不包含标记的上下文测试这个样式。

在许多情况下，如果调用小部件明确选择了特定的排序方式，而不是默认的排序方式，'true' 的值就不会被兑现。不是这种情况的例子是，在命令历史中，可以通过将样式设置为 'true' 来覆盖按时间顺序排列匹配的默认值。

在 _expand 补全器中，如果将其设置为 'true'，生成的扩展将始终排序。 如果将其设置为 'menu'，则只有当扩展作为单个字符串提供时，才会对扩展进行排序，而不是在包含所有可能扩展的字符串中进行排序。

special-dirs

通常情况下，补全代码不会将目录名 '.' 和 '..' 作为可能的补全。 如果将该样式设置为 'true'，则会同时添加 '.' 和 '..' 作为可能的补全；如果设置为 '..'，则只会添加 '..'。

下面的示例将 special-dirs 设置为 '..' ，前提是当前前缀为空，是单一的 '.' 或仅由以 '../' 开头的路径组成。 否则，该值为 'false'。

```
zstyle -e ':completion:*' special-dirs \
   '[[ $PREFIX = (../)#(|.|..) ]] && reply=(..)'
```

squeeze-slashes

如果设置为 'true'，文件名路径中的斜线序列（例如在 'foo//bar' 中）将被视为单个斜线。 这是 UNIX 路径的通常行为。 不过，默认情况下，文件补全函数会认为斜线之间有一个 '*' 。

stop

如果设置为 'true'，_history_complete_word 可绑定命令将在到达历史记录的开始或结束时停止一次。 然后，调用 _history_complete_word 将回绕到历史的另一端。 如果将此样式设置为 'false'（默认），_history_complete_word 将像菜单补全一样立即循环。

strip-comments

如果设置为 'true'，该样式会删除补全匹配中的非必要注释文本。 目前，它只用于补全电子邮件地址，在这种情况下，它会删除地址中的任何显示名称，将其缩减为简单的 *user@host* 格式。

`subst-globs-only`

这将被 `_expand` 补全器使用。 如果将其设置为 'true'，则只有通过 globbing 产生的扩展才会被使用；因此，如果扩展是通过使用下面描述的 `substitute` 样式产生的，但这些扩展并没有通过 globbing 进一步改变，则这些扩展将被拒绝。

该样式的默认值为 'false'。

替换

该布尔样式控制 `_expand` 补全器是否首先尝试扩展字符串中的所有替换（如 '`$(...)`' 和 '`${...}`'）。

默认是 'true'.

`suffix`

如果单词以转折号(tilde)开头或包含参数扩展，`_expand` 补全器就会使用它。 如果将其设置为 'true'，则只有在单词没有后缀的情况下才会进行扩展，也就是说，如果单词的后缀是类似 '`~foo`' 或 '`$foo`'，而不是 '`~foo/`' 或 '`$foo/bar`'，才会进行扩展，除非该后缀本身包含符合扩展条件的字符。 该样式的默认值为 'true'。

`tag-order`

这就提供了一种机制，用于对特定上下文中可用标记的使用方式进行排序。

样式的值是一组以空格分隔的标记列表。 每个值中的标签将同时被尝试；如果没有找到匹配，则使用下一个值。 (请参阅 `file-patterns` 样式，了解此行为的例外情况)。

例如：

```
zstyle ':completion:*:complete:-command-:*:*' tag-order \
    'commands functions'
```

指定命令位置的补全首先提供外部命令和 shell 函数。 如果找不到补全，将尝试其余标记。

除标记名称外，值中的每个字符串还可以是以下形式之一：

`-`

如果任何值只包含一个连字符，那么将 **只** 生成其他值中指定的标记。 通常情况下，如果指定的标记无法生成任何匹配结果，则最后尝试所有未明确选择的标记。 这意味着，仅由一个连字符组成的单个值将关闭补全功能。

！*tags...*

以感叹号开头的字符串用于指定 **不会使用的** 标记的名称。 其效果与列出上下文中所有其他可能的标签相同。

tag:*label* ...

这里，*tag* 是标准标签之一，*label* 是任意名称。 匹配结果的生成与正常情况相同，但在上下文中使用 *label* 而不是 *tag*。 这在以！开头的词语中没有用处。

如果 *label* 以连字符开头，则 *tag* 将作为 *label* 的前缀，以形成用于查找的名称。 这可以用来让补全系统多次尝试某个标签，并为每次尝试提供不同的样式设置；请参阅下面的示例。

tag:*label*:*description*

与之前一样，但 `description` 将替换 `format` 样式值中的 '%d'，而不是补全函数提供的默认描述。 描述中的空格必须用反斜线引出。 出现在 *description* 中的 '%d' 会被补全函数给出的描述所替换。

在上述任何一种形式中，标记可以是一个模式或多个模式，其形式为 '{*pat1*,*pat2*...}'。 在这种情况下，除了同一字符串中任何明确给出的标记外，所有匹配标签都将被使用。

这些功能的一个用途是多次尝试一个标记，每次尝试都以不同的方式设置其他样式，但仍可使用所有其他标记，而不必重复所有标记。 例如，在第一次尝试补全时，让命令位置的函数名补全忽略所有以下划线开头的补全函数：

```
zstyle ':completion:*:*:-command-:*:*' tag-order \
    'functions:-non-comp *' functions
zstyle ':completion:*:functions-non-comp' \
    ignored-patterns '_*'
```

第一次尝试时，将提供所有标记，但 `functions` 标记将被 `functions-non-comp` 替换。 为该标记设置的 `ignored-patterns` 样式将排除以下划线开头的函数。 如果没有匹配项，则使用 `tag-order` 样式的第二个值，以使用默认标记补全函数，这次可能包括所有函数名。

一个标记的匹配可以分成不同的组。 例如

```
zstyle ':completion:*' tag-order \
    'options:-long:long\ options
     options:-short:short\ options
```

```
        options:-single-letter:single\ letter\ options'
    zstyle ':completion:*:options-long' \
        ignored-patterns '[-+](|-|[^-]*)'
    zstyle ':completion:*:options-short' \
        ignored-patterns '--*' '[-+]?'
    zstyle ':completion:*:options-single-letter' \
        ignored-patterns '???*'
```

如果设置了 group-names 样式，以‘--’开头的选项、以单个‘-’或‘+’开头但包含多个字符的选项，以及单字母选项，都将以不同的描述分组显示。

模式的另一个用途是连续尝试多个匹配规范。 matcher-list 样式提供了类似的功能，但它在补全系统中很早就进行了测试，因此不能为单个命令或更具体的上下文设置。 下面是如何在没有任何匹配规范的情况下尝试正常补全，如果没有匹配结果，则再次尝试大小写不敏感匹配，将效果限制在命令 foo 的参数上：

```
zstyle ':completion:*:*:foo:*:*' tag-order '*' '*:-case'
zstyle ':completion:*-case' matcher 'm:{a-z}={A-Z}'
```

首先，在 foo 之后补全时提供的所有标记都会使用正常的标记名称进行尝试。 如果没有匹配结果，则使用 tag-order 的第二个值，再次尝试所有标记，但这次每个标记的名称后都附加了 -case，以便查找样式。 因此，这次将使用示例中第二次调用 zstyle 时的 matcher 样式值来完成补全，而不区分大小写。

可以使用 zstyle 内置命令的 -e 选项来指定使用特定标记的条件。 例如:

```
zstyle -e '*:-command-:*' tag-order '
    if [[ -n $PREFIX$SUFFIX ]]; then
      reply=( )
    else
      reply=( - )
    fi'
```

只有当输入的字符串不为空时，才会尝试在命令位置补全。 这将通过 PREFIX 特殊参数进行测试；有关补全小部件内部特殊参数的描述，请参阅 补全小部件。 将 reply 设置为空数组时，默认情况下会一次性尝试所有标记；将其设置为仅包含连字符的数组时，则禁止使用所有标记，因此也无法使用所有补全。

如果没有为上下文定义 tag-order 样式，则将使用字符串‘(|*-)argument-* (|*-)option-* values’和‘options’以及补全函数提供的所有标记来提供合理的默认行为，在大多数命令中，参数（无论是普通命令参数还是选项参数）将在选项名称之前补全。

urls

该标记与 urls 标记一起用于补全 URL 的函数。

如果值由多个字符串组成，或者唯一的字符串没有命名文件或目录，则这些字符串作为 URL 来补全。

如果值中只包含一个字符串，即一个普通文件的名称，URL 将取自该文件（URL 之间可以用空格或换行分隔）。

最后，如果值中唯一的字符串命名了一个目录，则根植于该目录的目录层次结构会给出补全。 顶层目录应是文件访问方式，如 'http', 'ftp', 'bookmark' 等。 在许多情况下，下一级目录将是文件名。 目录层次结构可以根据需要尽可能深入。

例如,

```
zstyle ':completion:*' urls ~/.urls
mkdir -p ~/.urls/ftp/ftp.zsh.org/pub
```

允许在执行适当的命令（如 'netscape' 或'lynx'）后补全 URL `ftp://ftp.zsh.org/pub` 的所有组件。 但需要注意的是，访问方法和文件是分开补全的，因此如果设置了 `hosts` 样式，则可以在不参考 `urls` 样式的情况下补全主机。

更多信息请参阅函数 `_urls` 本身的描述（例如 '`more $^fpath/_urls(N)`'）。

`use-cache`

如果设置了该值，则会为任何使用该值的补全激活补全缓存层（通过 `_store_cache`、`_retrieve_cache` 和 `_cache_invalid` 函数）。 可以使用 `cache-path` 样式更改包含缓存文件的目录。

`use-compctl`

如果将此样式设置为 **不** 等于 `false`、`0`、`no` 和 `off` 的字符串，补全系统可以使用 `compctl` 内置命令定义的任何补全规范。 如果未设置该样式，则只有在加载了 `zsh/compctl` 模块时才会这样做。 该字符串还可以包含子字符串'`first`'，以使用用 '`compctl -T`' 定义的补全，以及子字符串 '`default`'，以使用用 '`compctl -D`' 定义的补全。

请注意，这只是为了从 `compctl` 向新的补全系统平稳过渡，将来可能会消失。

还请注意，只有在相关命令没有特定的补全函数时，才会使用 `compctl` 定义的。 例如，如果有一个函数 `_foo` 来补全命令 `foo` 的参数，那么 `compctl` 将不会被调用。 但是，如果 `foo` 只使用默认补全，则会尝试 `compctl` 版本。

`use-ip`

默认情况下，补全主机名的函数 `_hosts` 会从 NIS 和 ssh 文件等主机数据库读取的条目中剥离 IP 地址。 如果该样式为 'true'，则相应的 IP 地址也可以补全。 在任何设置了 `hosts` 样式的情况下，都不能使用该样式；还需注意的是，必须在生成主机名缓存（通常是第一次补全尝试）之前设置该样式。

users

可将其设置为待补全的用户名列表。 如果不设置，则将补全所有用户名。 请注意，如果设置了该选项，则只能补全该用户列表；这是因为在某些系统中，查询所有用户会耗费大量时间。

users-hosts

该样式的值应为 '*user*@*host*' 或 '*user*:*host*'。它用于需要成对用户名和主机名的命令。 这些命令将补全（仅）该样式中的用户名，并将限制后续主机名的补全，只能补全与该样式值之一中的用户配对的主机。

可以使用 my-accounts 标记对允许远程登录的命令集的值进行分组，如 rlogin 和 ssh。 同样，对于通常指向他人账户的命令集（如 talk 和 finger），也可以使用 other-accounts 标记进行分组。 比较矛盾的命令可以使用 accounts 标记。

users-hosts-ports

与 users-hosts 类似，但用于执行 telnet 等命令，并包含格式为 '*user*@*host*:*port*' 的字符串。

verbose

如果设置(默认)，则补全列表会更加详细。 特别是，如果该样式为 'true'，许多命令都会显示选项说明。

word

_list 补全器使用了这一功能，它可以防止插入补全，直到第二次补全尝试时行内容未发生变化。 正常情况下，要确定该行是否已更改，需要在两次尝试中比较其全部内容。 如果该样式为 'true'，则只对当前单词进行比较。 因此，如果在内容相同的另一个字上执行补全，补全将不会延迟。

20.4 控制函数

初始化脚本 compinit 会重新定义所有执行补全的小部件，以调用提供的小部件函数 _main_complete。 该函数作为一个包装器，调用生成匹配的所谓 'completer' 函数。 如果 _main_complete 是带参数调用的，那么这些参数将作为按给定顺序调用的补全函数名称。 如果没有给定参数，则从 completer 样式中获取要尝试的函数集。 例如，使用正常补全，如果不能生成任何匹配，则进行修正：

```
zstyle ':completion:*' completer _complete _correct
```

在调用 compinit 之后。该样式的默认值为 '_complete _ignored'，即通常只尝试普通补全，先使用 ignored-patterns 样式的效果，然后不使用。 _main_complete 函

数使用补全函数的返回状态来决定是否调用其他补全器。 如果返回状态为零，则不尝试其他补全器，并返回 _main_complete 函数。

如果 _main_complete 的第一个参数是单个连字符，则不会将参数作为补全器的名称。相反，第二个参数会给出一个名称，用于上下文的 *completer* 字段，其他参数会给出一个命令名称和参数s，用于调用生成匹配。

尽管用户可以编写自己的补全函数，但发行版中包含以下补全函数。 请注意，在上下文中，前导下划线会被去掉，例如，基本补全是在上下文‘:completion::complete:…’中执行的。

_all_matches

> 该补全器可用于添加一个由所有其他匹配项组成的字符串。 由于它会影响后面的补全器，因此必须作为列表中的第一个补全器出现。 所有匹配列表都会受到上述 avoid-completer 和 old-matches 样式的影响。
>
> 例如，使用下面描述的 _generic 函数将 _all_matches 与自己的按键绑定，可能会很有用：
>
> ```
> zle -C all-matches complete-word _generic
> bindkey '^Xa' all-matches
> zstyle ':completion:all-matches:*' old-matches only
> zstyle ':completion:all-matches::::' completer _all_matches
> ```
>
> 请注意，这本身并不会生成补全：首先使用任何标准方法生成补全列表，然后使用 ^Xa 显示所有匹配。 您可以在列表中添加一个标准补全器，并要求直接插入所有匹配列表：
>
> ```
> zstyle ':completion:all-matches::::' completer \
>        _all_matches _complete
> zstyle ':completion:all-matches:*' insert true
> ```
>
> 在这种情况下，不应设置 old-matches 样式。

_approximate

> 这与基本的 _complete 补全器类似，但允许对补全内容进行修正。 错误的最大数量可以通过 max-errors 样式指定；关于如何计算错误，请参阅 文件名生成 中关于近似匹配的描述。 通常情况下，该补全器只会在正常的 _complete 补全器之后尝试：
>
> ```
> zstyle ':completion:*' completer _complete _approximate
> ```
>
> 只有在正常补全没有结果的情况下，才会给出修正补全。 当找到修正补全时，补全器通常会启动菜单补全，允许您循环查看这些字符串。

该补全器在生成可能的更正和原始字符串时使用 corrections 和 original 标记。 前者的 format 样式可能包含额外的序列 '%e' 和 '%o' ，它们将分别被生成更正和原始字符串时接受的错误数所取代。

补全器会逐步增加允许的错误数，直至达到 max-errors 样式的限制，因此如果发现一个补全有一个错误，就不会显示有两个错误的补全，以此类推。 它会修改上下文中的补全器名称，以显示正在尝试的错误数：第一次尝试时，补全器字段包含 'approximate-1'，第二次尝试时包含 'approximate-2'，以此类推。

当 _approximate 从其他函数调用时，可以通过 -a 选项传递接受的错误数。 参数的格式与 max-errors 样式相同，都在一个字符串中。

需要注意的是，这个补全器（以及下面提到的 _correct 补全器）的调用代价可能会相当高昂，尤其是在允许出现大量错误的情况下。 避免这种情况的方法之一是使用 zstyle 的 -e 选项来设置 completer 样式，这样只有在对同一字符串进行第二次补全尝试时才会使用某些补全器，例如：

```
zstyle -e ':completion:*' completer '
  if [[ $_last_try != "$HISTNO$BUFFER$CURSOR" ]]; then
    _last_try="$HISTNO$BUFFER$CURSOR"
    reply=(_complete _match _prefix)
  else
    reply=(_ignored _correct _approximate)
  fi'
```

它会使用 HISTNO 参数和 BUFFER 及 CURSOR 特殊参数（这些参数在 zle 和补全小部件中可用）来查找命令行自上次尝试补全后是否未发生变化。 只有这样，才会调用 _ignored、_correct 和 _approximate 补全函数。

_canonical_paths [-A *var*] [-N] [-MJV12nfX] *tag descr* [*paths* ...]

该补全函数会补全所有给定的路径，并尝试提供与给定路径指向相同文件的补全（当给定绝对路径时为相对路径，反之亦然；当待补全的单词中存在 . . 时；以及从符号链接中获得的某些路径）。

如果指定了 -A，则会从指定的数组变量中获取路径。路径也可以如上所示在命令行中指定。 如果指定了 -N，则可以防止在使用所给路径进行补全之前将其规范化，假设它们已经这样规范化。选项 -M、-J、-V、-1、-2、-n、-F、-X 会传递给 compadd。

有关 *tag* 和 *descr* 的说明，请参见 _description。

_cmdambivalent

将剩余的位置参数作为外部命令补全。 如果命令行上有两个或两个以上的剩余位置参数，外部命令及其参数将作为单独的参数（以补全 /usr/bin/env 的适当方式）

补全，否则将作为带引号的命令字符串（以 system(...) 的方式）补全。 另请参见 _cmdstring 和 _precommand。

该函数不带参数。

_cmdstring

以单个参数形式补全外部命令，如 system(...)。

_complete

该补全器以上下文相关的方式生成所有可能的补全，即使用上文解释的 compdef 函数定义的设置以及所有特殊参数的当前设置。 这就给出了正常的补全行为。

为了补全命令的参数，_complete 使用了实用函数 _normal，该函数又负责查找特定的函数；下文将对其进行介绍。 *-context-* 形式的各种上下文会特殊处理。上面提到的这些都是 #compdef 标记的可能参数。

在尝试为特定上下文查找函数之前，_complete 会检查参数'compcontext' 是否已设置。设置 'compcontext' 可以覆盖通常的补全调度，这在使用 vared 作为输入的函数等地方非常有用。如果将其设置为数组，则元素将被视为可能的匹配项，这些匹配项将使用标记'values' 和描述 'value' 来补全。如果设置为关联数组，则键被用作可能的补全，值（如果非空）被用作匹配的描述。 如果将'compcontext' 设置为包含冒号的字符串，则其形式应为 '*tag*:*descr*:*action*'。 在这种情况下，*tag* 和 *descr* 给出了要使用的标记和描述，*action* 则表示应以下面描述的 _arguments 实用程序函数所接受的形式之一来补全。

最后，如果将 'compcontext' 设置为不带冒号的字符串，则该值将作为要使用的上下文名称，并调用为该上下文定义的函数。 为此，有一种名为 -command-line- 的特殊上下文可以补全整条命令行（命令及其参数）。 补全系统本身并不使用该上下文，但在明确调用时会对其进行处理。

_correct

为当前单词生成更正，但不生成补全；这与 _approximate 类似，但不会像该补全器那样允许光标处出现任何数量的额外字符。 其效果类似于拼写检查。 它基于 _approximate，但上下文名称中的补全器字段是 correct。

例如：

```
zstyle ':completion:::::' completer \
       _complete _correct _approximate
zstyle ':completion:*:correct:::' max-errors 2 not-numeric
zstyle ':completion:*:approximate:::' max-errors 3 numeric
```

更正最多接受两个错误。 如果给出了数字参数，则不会执行修正，但会执行更正补全，并接受与数字参数相同数量的错误。 在没有数字参数的情况下，将先尝试修正，然后再尝试更正补全，前者接受两个错误，后者接受三个错误。

当 _correct 作为函数调用时，可以在 -a 选项后给出要接受的错误数量。 参数的形式与 accept 样式的值相同，都在一个字符串中。

该补全函数可以在不使用 _approximate 补全函数的情况下使用，或者像示例中那样，在 _approximate 补全函数之前使用。 在 _approximate 补全函数之后使用该函数是没有用的，因为 _approximate 至少会生成 _correct 补全函数生成的修正字符串 — 可能还会更多。

_expand

该补全函数并不真正执行补全，而是检查命令行中的单词是否符合扩展条件，如果符合，则详细控制如何进行扩展。 为此，需要使用 complete-word 而不是 expand-or-complete（TAB 的默认绑定）来调用补全系统，否则字符串将在补全系统启动之前被 shell 的内部机制展开。 还请注意，此补全函数应在 _complete 补全函数之前调用。

生成扩展时使用的标记有：all-expansions，用于包含所有可能扩展的字符串；expansions，用于将可能的扩展作为单个匹配项添加；original，用于添加该行的原始字符串。 这些字符串的生成顺序（如果有的话）可以像往常一样由 group-order 和 tag-order 样式控制。

all-expansions 和 expansions 的格式字符串可能包含 '%o' 序列，该序列将被该行的原始字符串替换。

要尝试的扩展类型由 substitute、glob 和 subst-globs-only 样式控制。

也可以将 _expand 作为函数调用，在这种情况下，可以通过选项选择不同的模式：-s 表示 substitute，-g 表示 glob，-o 表示 subst-globs-only。

_expand_alias

如果光标所在的单词是别名，则会展开该别名，而不会调用其他补全器。 可以使用 regular、global 和 disabled 样式来控制要展开的别名类型。

该函数也是一个可绑定命令，请参阅 可绑定命令。

_extensions

如果光标跟随字符串 '*.'，则补全文件扩展名。扩展名取自当前目录下的文件或指定以当前单词开头的目录。对于完全匹配，补全继续允许 _expand 等其他补全器扩展模式。标准的 add-space 和 prefix-hidden 样式会被遵守。

_external_pwds

补全属于当前用户的其他 zsh 进程的当前目录。

它通过 _generic 与自定义组合键绑定使用。请注意，模式匹配已启用，因此匹配的执行方式与 _match 补全器类似。

_history

补全 shell 历史命令中的单词。 该补全器可由 remove-all-dups 和 sort 样式控制，正如 _history_complete_word 可绑定命令一样，请参阅 可绑定命令 和 补全系统配置。

_ignored

可以将 ignored-patterns 样式设置为一个模式列表，并将其与可能的补全进行比较；匹配的模式将被删除。 有了这个补全器，就可以恢复这些匹配，就像没有设置 ignored-patterns 样式一样。 补全器实际上会生成自己的匹配列表；调用哪些补全器的决定方与 _prefix 补全器相同。 如上所述，还可以使用 single-ignored 样式。

_list

该补全器允许延迟插入匹配内容，直到第二次尝试补全时，行上的单词才会被更改。 第一次尝试时，只显示匹配列表。 它受 condition 和 word 样式的影响，参见 补全系统配置。

_match

该补全器用于 _complete 补全器之后。 它的行为类似，但命令行上的字符串可以是与试用补全匹配的模式。 它具有 GLOB_COMPLETE 选项的效果。

通常情况下，补全是通过从行中提取模式，在光标位置插入 '*' ，然后将生成的模式与可能的补全进行比较。 这可以通过上述 match-original 样式进行修改。

除非 insert-unambiguous 样式设置为 'true'，否则生成的匹配结果将在菜单补全中提供；有关该样式的其他选项，请参阅上文的说明。

请注意，全局定义的或补全函数使用的匹配器规范（样式 matcher-list 和 matcher），不会被使用。

_menu

该补全器是作为一个简单的示例函数编写的，用于展示如何在 shell 代码中启用菜单补全功能。不过，它也有一个显著的效果，就是禁用了菜单选择功能，这对基于 _generic 的小部件很有用。它应作为列表中的第一个补全器使用。 请注意，这与 MENU_COMPLETE 选项的设置无关，并且不能与 reverse-menu-complete 或 accept-and-menu-complete 等其他菜单补全小部件一起使用。

_oldlist

在存在由特殊补全（即单独绑定的补全命令）生成的现有补全列表时，该补全器可控制标准补全部件的行为。 它允许普通补全键继续使用由此生成的补全列表，而不是生成一个新的普通上下文补全列表。 它应该在任何生成匹配的小部件之前，出现在补全器列表中，。 它使用两种样式： old-list 和 old-menu，请参阅 补全系统配置。

_precommand

用分隔参数（word-separated）补全外部命令，如 exec 和 /usr/bin/env。

_prefix

该补全器可用于在忽略后缀（光标后的所有内容）的情况下尝试补全。 换句话说，后缀不会被视为要补全的单词的一部分。 其效果类似于 expand-or-complete-prefix 命令。

completer 样式用于决定调用哪些其他补全程序来生成匹配。 如果未设置此样式，则使用为当前上下文设置的补全程序列表 — 当然，_prefix 补全器本身除外。 此外，如果该补全器在补全器列表中出现不止一次，则只会调用上次调用 _prefix 时尚未尝试过的补全器。

例如，请看这个全局 completer 样式：

```
zstyle ':completion:*' completer \
    _complete _prefix _correct _prefix:foo
```

在这里，_prefix 补全器会尝试正常补全，但忽略后缀。 如果没有匹配结果，之后调用 _correct 补全器也没有匹配结果，那么 _prefix 将被第二次调用，现在只尝试修正而忽略后缀。 在第二次调用时，上下文的补全部分显示为 'foo'。

将 _prefix 作为最后手段使用，并在调用时只尝试正常补全：

```
zstyle ':completion:*' completer _complete ... _prefix
zstyle ':completion::prefix:*' completer _complete
```

同时，add-space 样式也会受到尊守。 如果设置为 'true' ，_prefix 将在生成的匹配项（如果有）和后缀之间插入空格。

请注意，只有设置了 COMPLETE_IN_WORD 选项，这个补全器才会有用；否则，在调用补全代码之前，光标会被移到当前单词的末尾，因此不会有后缀。

_user_expand

该补全器的行为与 _expand 补全器类似，但它执行的是由用户定义的扩展。 除了补全系统一般处理的其他样式外，_user_expand 还可以使用指定给 _expand 的 _add-space 和 sort 样式。 此外，还可以使用 all-expansions 标记。

扩展取决于为当前上下文定义的数组样式 user-expand；请记住，补全器的上下文没有上下文补全程序那么具体，因为完整的上下文尚未确定。 数组的元素可以是以下形式之一：

$*hash*

hash 是关联数组的名称。 请注意，这并不是一个完整的参数表达式，而只是一个 $，为防止立即展开，该表达式后跟了一个关联数组的名称。 如果试验扩展词与 *hash* 中的键匹配，则扩展结果就是相应的值。

_func

_func 是 shell 函数的名称，其名称必须以 _ 开头，但对补全系统没有其他特殊要求。 调用该函数时，试用词作为参数。 如果要对单词进行扩展，函数应将数组 reply 设置为扩展列表。 可选的，函数还可以将 REPLY 设置为一个单词，该单词将用作扩展集的描述。 函数的返回状态与此无关。

20.5 可绑定命令

除了所提供的与上下文相关的补全（这些补全有望以直观明显的方式运行）外，还有一些实现特殊行为的小部件可以单独绑定到按键上。 下面列出了这些部件及其默认绑定。

_bash_completions

_bash_complete-word 和 _bash_list-choices 这两个小部件使用了该函数。 它的存在是为了与 bash 中的补全绑定兼容。 绑定的最后一个字符决定补全的内容：‘!’，命令名；‘$’，环境变量；‘@’，主机名；‘/’，文件名；‘~’，用户名。 在 bash 中，以 ‘\e’ 开头的绑定给出补全，‘^X’ 开头的绑定列出选项。 由于其中一些绑定与标准的 zsh 绑定冲突，因此默认只绑定了 ‘\e~’ 和 ‘^X~’。 要添加其余选项，应在 compinit 运行后将以下内容添加到 .zshrc 中：

```
for key in '!' '$' '@' '/' '~'; do
  bindkey "\e$key" _bash_complete-word
  bindkey "^X$key" _bash_list-choices
done
```

这包括 ‘~’ 的绑定，以防它们已经绑定到其他东西上；补全代码不会覆盖用户绑定。

_correct_filename (^XC)

更正光标位置的文件名路径。 最多允许在名称中出现六个错误。 也可以调用带参数的命令来修正文件名路径，独立于 zle； 修正结果将打印在标准输出上。

_correct_word (^Xc)

使用通常的上下文补全词作为可能的选择，对当前参数执行校正。这会在上下文名称的 *function* 字段中存储字符串 ‘correct-word’ ，然后调用 _correct 补全器。

_expand_alias (^Xa)

该函数可用作补全器和可绑定命令。 如果光标所在的单词是别名，它将展开该单词。 可以使用 regular、global 和 disabled 样式控制展开的别名类型。

当作为可绑定命令使用时，还有一个额外的功能，可以通过将 complete 样式设置为 ‘true’ 来选择。 在这种情况下，如果单词不是别名的名称，_expand_alias 会尝试将单词补全为完整的别名，而不对其进行扩展。 它会将光标直接留在已补全的单词之后，这样再次调用 _expand_alias 时，就会展开现已补全的别名。

_expand_word (^Xe)

对当前单词进行扩展：等同于标准的 expand-word 命令，但使用 _expand 补全器。 调用前，上下文的 *function* 字段将被设置为‘expand-word’。

_generic

该函数未定义为小部件，默认情况下不绑定。 不过，它可以用来定义一个小部件，然后会将小部件的名称存储在上下文的 *function* 字段中，并调用补全系统。 这样就可以轻松定义具有自己风格设置的自定义补全小部件。 例如，定义一个执行正常补全并启动菜单选择的小部件：

```
zle -C foo complete-word _generic
bindkey '...' foo
zstyle ':completion:foo:*' menu yes select=1
```

需要特别注意的是，可以为上下文设置 completer 样式，以更改用于生成可能匹配的函数集。 如果 _generic 在调用时带有参数，那么这些参数将作为补全函数列表传递给 _main_complete，以代替 completer 样式所定义的补全函数。

_history_complete_word (\e/)

补全 shell 命令历史中的单词。它使用 list、remove-all-dups、sort 和 stop 样式。

_most_recent_file (^Xm)

补全与命令行模式匹配的最近修改的文件名（可能为空）。 如果给定一个数字参数 *N*，则补全第 *N* 个最近修改的文件。 请注意，补全（如果有）总是唯一的。

_next_tags (^Xn)

这条命令会改变下一个标记或标记集的匹配集，可以是 tag-order 样式给出的匹配集，也可以是默认设置的匹配集；否则这些匹配集将不可用。 连续调用该命令将循环使用所有可能的标记集。

_read_comp (^X^R)

提示符会提示用户输入一个字符串，并使用该字符串对当前单词进行补全。 字符串有两种可能。 首先，它可以是一组以 '_' 开头的单词，例如 '_files -/'，在这种情况下，将调用带有任何参数的函数来生成补全。 函数名中不明确的部分将自动补全（此时无法使用普通补全），直到键入空格为止。

其次，任何其他字符串都将作为一组参数传递给 comppadd，因此应该是一个表达式，指定应该补全的内容。

读取字符串时，只能使用一组非常有限的编辑命令：'DEL' 和 '^H' 删除最后一个字符；'^U' 删除行， '^C' 和 '^G' 中止函数，而 'RET' 接受补全。 请注意，字符串是作为命令行逐字使用的，因此参数必须按照标准 shell 规则加引号。

一旦读取了一个字符串，下一次调用 _read_comp 时，将使用现有的字符串，而不会读取新的字符串。 如果要强制读取新字符串，请在调用 _read_comp 时输入数字参数。

_complete_debug (^X?)

该小部件执行普通补全，但会在一个临时文件中捕获补全系统执行的 shell 命令的跟踪信息。 每次补全尝试都有自己的文件。 查看每个文件的命令会被推入编辑器缓冲堆栈。

_complete_help (^Xh)

该小部件显示在当前光标位置补全时使用的上下文名称、标记和补全函数等信息。如果给定的数字参数不是 1（如 'ESC-2 ^Xh'），则还将显示使用的样式和上下文。

请注意，有关样式的信息可能是不完整的；它取决于所调用的补全函数提供的信息，而补全函数的信息又是由用户自己的样式和其他设置决定的。

_complete_help_generic

与此处列出的其他命令不同，它必须作为普通 ZLE 小部件而非补全小部件创建（即使用 zle -N）。 它用于生成与上述 _generic 小部件绑定的帮助。

如果该小部件是使用函数名称创建的（默认情况下是这样），那么执行时它将读取一个按键序列。 这将绑定到使用 _generic 小部件的补全函数的调用。 该小部件将被执行，并以 _complete_help 小部件显示上下文补全的相同格式提供信息。

如果小部件的名称中包含 debug，例如以 'zle -N _complete_debug_generic _complete_help_generic' 的方式创建，它就会像之前一样读取并执行通用小部

件的密钥(keystring)，但随后会像 _complete_debug 为上下文补全所做的那样生成调试信息。

如果小部件的名称中包含 noread，它将不会读取密钥，而是安排下一次使用在同一 shell 中运行的通用小部件时,产生上述效果。

小部件通过设置由 _generic 读取的 shell 参数 ZSH_TRACE_GENERIC_WIDGET 来工作。 取消参数设置后，noread 形式的任何待定效果都将被取消。

例如，执行以下操作后:

```
zle -N _complete_debug_generic _complete_help_generic
bindkey '^x:' _complete_debug_generic
```

键入 'C-x :'，然后输入通用小部件的按键序列，就会将该小部件的跟踪输出保存到文件中。

_complete_tag (^Xt)

该小部件补全由 etags 或 ctags 程序（注意与补全系统的标签没有任何联系）创建的符号标记，这些标记存储在 TAGS 文件中，格式为 etags 使用的格式，或 tags 文件中，格式为 ctags 创建的格式。 它将在路径层次结构中查找第一个出现的文件；如果两个文件都存在，则优先选择 TAGS 文件。 您可以通过设置参数 $TAGSFILE 或 $tagsfile 分别指定 TAGS 或 tags 文件的完整路径。 相应使用的补全标记是 etags 和 vtags，分别与 emacs 和 vi 后缀相同。

20.6 实用函数

下面将介绍在编写补全函数时可能有用的实用函数。 如果函数安装在子目录中，其中大部分位于 Base 子目录中。 与分发版中的命令的示例函数一样，生成匹配的实用函数都遵循这样的惯例：如果生成补全，则返回状态为 0；如果无法添加匹配的补全，则返回非 0。

_absolute_command_paths

该函数将外部命令补全为绝对路径（与 _command_names -e 不同，后者补全的是其基名）。 它不需要参数。

_all_labels [-x] [-12VJ] *tag name descr* [*command arg ...*]

这是下面 _next_label 函数的一个便捷接口，用于实现 _next_label 示例中的循环。 调用 *command* 及其参数可生成匹配结果。 存储在参数 *name* 中的选项会自动插入传给 *command* 的 *args* 中。 通常情况下，这些选项会直接放在 *command* 之后，但如果 *args* 中有一个是连字符，这些选项就会直接插入 *command* 之前。 如果连字符是最后一个参数，则会在调用 *command* 之前从参数

列表中删除。 这样一来，_all_labels 几乎可以在所有情况下使用，只需调用 compadd 内置命令或调用其中一个实用函数，即可生成匹配结果。

例如：

```
local expl
...
if _requested foo; then
  ...
  _all_labels foo expl '...' compadd ... - $matches
fi
```

将补全 matches 参数中的字符串，使用 compadd 和附加选项，这些选项优先于 _all_labels 生成的选项。

_alternative [-O *name*] [-C *name*] *spec* ...

该函数适用于有多个标记的简单情况。 本质上，它实现了一个循环，就像下面描述的 _tags 函数一样。

spec 的形式描述了要使用的标记和请求标记时要执行的操作：'*tag*:*descr*:*action*'。 *tag*s 使用 _tags 提供，如果标记被请求，则执行 *action* 并给出描述 *descr*。 *action*s 是 _arguments 函数（如下所述）接受的参数，但以下情况除外：

- 不支持 '->*state*' 和 '=...' 形式。
- '((a\:bar b\:baz))' 形式不需要转义冒号，因为 *spec*s 在 *action* 之后没有以冒号分隔的字段。

例如，*action* 可能是一个简单的函数调用：

```
_alternative \
    'users:user:_users' \
    'hosts:host:_hosts'
```

提供用户名和主机名作为可能的匹配项，分别由 _users 和 _hosts 函数生成。

与 _arguments 一样，该函数使用 _all_labels 来执行操作，将循环遍历所有标记集。 只有在有额外有效标记的情况下，例如在 _alternative 调用的函数内部，才需要进行特殊处理。

选项 '-O *name*' 的使用方式与 _arguments 函数相同。 换句话说，在执行操作时，*name* 数组中的元素将传递给 compadd。

与 _tags 类似，该函数支持 -C 选项，可为参数上下文字段赋予不同的名称。

_arguments [-nswWCRS] [-A *pat*] [-O *name*] [-M *matchspec*]
[:] *spec* ...

_arguments [*opt* ...] -- [-l][-i *pats*][-s *pair*]
[*helpspec* ...]

该函数可用于为参数遵循标准 UNIX 选项和参数约定的命令提供完整的补全规范。

选项概述

_arguments 本身的选项必须使用单独的单词，即 -s -w，而不是 -sw。 选项后面的 *specs* 描述了已分析命令的选项和参数。 为避免歧义，_arguments 本身的所有选项可以用一个冒号与 *spec* 分开。

'--' 形式用于从被分析命令的帮助输出中直观获取 *spec* 形式，下文将详细介绍。 此外，'--' 形式的 *opts* 与第一种形式相同。 请注意，'--' 后面的 '-s' 与 '--' 前面的 '-s' 含义不同，两者都可能出现。

选项开关 -s、-S、-A、-w 和 -W 会影响 _arguments 分析命令行选项的方式。这些开关对使用标准参数解析的命令非常有用。

_arguments 的选项含义如下：

-n

使用此选项后，_arguments 会将参数 NORMARG 设置为 $words 数组中第一个正常参数的位置，即选项结束后的位置。 如果该参数尚未到达，NORMARG 将被设置为 -1。 如果传递了 -n 选项，调用者应声明 'integer NORMARG'，否则将不使用该参数。

-s

启用单字母选项的 **选项堆叠**，可以将多个单字母选项合并为一个单词。 例如，两个选项 '-x' 和 '-y' 可以合并为一个单词 '-xy'。 默认情况下，每个单词对应一个选项名称（'-xy' 是一个名为 'xy' 的选项）。

以单个连字符或加号开头的选项可叠加；以两个连字符开头的词不可叠加。

请注意，-- 后的 -s 含义不同，这在题为 '从帮助输出中获取 *spec* 形式' 的部分有详细说明。

-w

与 -s 结合使用时，即使一个或多个选项带有参数，也允许堆叠选项。 例如，如果 -x 包含一个参数，在没有 -s 的情况下，'-xy' 被视为一个单独的（未处理的）选项；在有 -s 的情况下，-xy 是一个包含参数 'y' 的选项； 同时使用 -s 和 -w 时，-xy 是选项 -x 和选项 -y，-x 的参数（以及 -y 的参数，如果它需要参数）仍在后面的语句中。

-W

该选项使 -w 更进了一步：即使在出现在同一单词中的参数之后，也可以补全单字母选项。 不过，选项是否真的会在此时补全取决于所执行的操作。 为了获得更多的控制权，可以使用 _guard 这样的实用程序作为操作的一部分。

-C

为 '->*state*' 形式的操作修改 curcontext 参数。 下文将对此进行详细讨论。

-R

当要处理 $state 时，以 '->*string*' 语法返回状态 300 而不是 0。

-S

不要在行中出现的 '--' 之后补全选项，并忽略 '--'。 例如，在行中,带有 -S

```
foobar -x -- -y
```

'-x' 被视为选项， '-y' 被视为参数，而 '--' 既不是选项也不是参数。

-A *pat*

在该行第一个非选项参数之后，选项不再补全。 *pat* 是一个匹配所有不作为参数的字符串的模式。 例如，要使 _arguments 在第一个普通参数之后不再补全选项，但忽略所有以连字符开头的字符串，即使这些字符串没有被 *optspec* 之一描述，其形式为 '-A "-*"'。

-O *name*

将数组 *name* 中的元素作为参数传递给执行 *actions* 的函数。 下文将对此进行详细讨论。

-M *matchspec*

使用匹配规范 *matchspec* 来补全选项名和值。 默认的 *matchspec* 允许在 '_' 和 '-' 之后补全部分单词，例如将 '-f-b' 补全为 '-foo-bar'。 默认的 *matchspec* 是：

```
r:|[_-]=* r:|=*
```

-0

在填充关联数组 'opt_args' 的值时，不要反斜线转义冒号和反斜线，在连接多个值时使用 NUL 而不是冒号。下文将在 ***specs: actions*** 标题下详细介绍该选项。

specs: 概述

以下每种形式都是一个 *spec*，用于描述被分析命令行上的各组选项或参数。

n:*message*:*action*
n::*message*:*action*

这描述了第 *n* 个普通参数。 *message* 将打印在生成的匹配结果上方，而 *action* 则表示在这个位置可以补全的内容（见下文）。 如果 *message* 前面有两个冒号，则该参数为可选参数。 如果 *message* 只包含空格，除非操作本身添加了解释字符串，否则匹配结果上方不会打印任何内容。

:*message*:*action*
::*message*:*action*

类似，但描述的是 **下一个** 参数，不管它是什么数字。 如果所有参数都按正确的顺序以这种形式指定，数字就没有必要了。

*:*message*:*action*
*::*message*:*action*
*:::*message*:*action*

说明在没有提供前两种形式的参数时，如何补全参数（通常是非选项参数，即不是以 - 或 + 开头的参数）。 任何数量的参数都可以用这种方式补全。

如果 *message* 之前有两个冒号，`words` 特殊数组和 `CURRENT` 特殊参数将被修改为在 *action* 执行或求值时仅引用常规参数。 如果在 *message* 之前加上三个冒号，它们将被修改为仅指本说明涵盖的正常参数。

optspec
optspec:...

描述一个选项。 冒号表示处理该选项的一个或多个参数；如果没有冒号，则假定该选项不带参数。

无论选项是否有参数，初始的 *optspec* 都有以下几种形式。

**optspec*

这里的 *optspec* 是以下其余形式之一。 这表示后面的 *optspec* 可以重复。 否则，如果光标左侧的命令行中已有相应选项，则不会再次提供。

-optname
+optname

在最简单的形式中，*optspec* 只是以减号或加号开头的选项名称，如 '`-foo`'。 选项的第一个参数（如果有）必须以 ***separate*** 的形式出现在选项之后。

'*-+optname*' 和 '*+-optname*' 中的任一个都可以用来指定 *-optname* 和 *+optname* 都有效。

在其余所有形式中，前导符 '-' 都可以以这个的方式用 '+' 替换或配对。

-optname-

选项的第一个参数必须直接位于 **同一个词中** 的选项名称之后。例如，'`-foo-:...`' 指定补全的选项和参数看起来像 '`-foo`*arg*'。

-optname+

第一个参数可以紧跟在 *optname* 之后出现在同一个词中，也可以在选项之后作为一个单独的词出现。 例如，'`-foo+:...`' 表示补全后的选项和参数看上去像 '`-foo`*arg*' 或 '`-foo` *arg*'。

-optname=

参数可以作为下一个单词出现，也可以与选项名称出现在同一个单词中，但必须用等号分隔，例如 '`-foo=`*arg*' 或 '`-foo` *arg*'。

-optname=-

选项的参数必须出现在同一单词的等号之后，且不得在下一个参数中给出。

optspec`[`*explanation*`]`

在 *optspec*的任何前述形式中，都可以用括号括起解释字符串，如 '`-q[query operation]`'。

`verbose` 样式用于决定是否在补全列表中与选项一起显示解释字符串。

如果没有给出括号内的解释字符串，但设置了 `auto-description` 样式，且该 *optspec* 只描述了一个参数，则会显示该样式的值，其中出现的任何序列 '%d' 都会被 *optspec* 后面的第一个 *optarg* 的 *message* 所替换；见下文。

选项中可以出现字面的 '+' 或 '=' ，但必须加上引号，例如 '`-\+`'。

在 *optspec* 之后的每个 *optarg* 必须是以下形式之一：

:message:action
::message:action

选项的参数；*message* 和 *action* 与普通参数一样。 在第一种形式中，参数是必选的，而在第二种形式中，参数是可选的。

对于包含多个参数的选项，可以重复使用这组参数。 换句话说，*:message1:action1:message2:action2* 表示该选项有两个参数。

*:*pattern:message:action*

:*pattern::message:action
:*pattern:::message:action

描述多个参数。 对于包含多个参数的选项，只有最后一个 *optarg* 可以用这种形式给出。 如果 *pattern* 为空（即 :*:），则该行剩余的所有单词都将按 *action* 所描述的方式补全；否则，所有单词（直到和包括与 *pattern* 匹配的单词）都将按 *action* 所描述的方式补全。

多个冒号的处理方式与普通参数的 '*:...' 形式相同：当 *message* 前面有两个冒号时，`words` 特殊数组和 `CURRENT` 特殊参数会在 *action* 执行或求值过程中被修改为仅指选项后面的单词。 如果前面有三个冒号，它们将被修改为仅指本说明所涉及的字词。

在 *optname*、*message* 或 *action* 中的任何字面冒号前都必须加上反斜线 '\:'。

上述每种形式的前面都可以用括号列出选项名称和参数编号。 如果给定选项在命令行中，则不提供括号中的选项和参数。 例如，'`(-two -three 1)-one:...`' 补全了选项 '`-one`'；如果该选项出现在命令行中，在它之后将不会补全选项 `-two` 和 `-three` 以及第一个普通参数。 '`(-foo):...`' 指定普通参数补全；如果该参数已经存在，则不会补全 `-foo`。

排除选项列表中还可能出现其他项目，以表示在匹配当前规范时不应应用的其他各种项目：单星号（*）表示其余参数（即格式为 '*:...' 的规范）；冒号（:）表示所有普通（非选项）参数；连字符（-）表示所有选项。 例如，如果'(*)' 出现在某个选项之前，并且该选项出现在命令行中，那么剩余的参数列表（上表中以 '*:' 开头的参数）将不会补全。

为了帮助规范的重复使用，可以在上述任何形式前加上 '!'；这样形式将不再补全，但如果选项或参数出现在命令行中，它们将被正常跳过。 这样做的主要用途是当参数由数组给出时，`_arguments` 会在更具体的上下文中被重复调用：第一次调用时使用 '`_arguments $global_options`'，随后调用时使用 '`_arguments ! $^global_options`'。

***specs:* 行为**

在上述每个形式中，*action* 都决定了补全的生成方式。 除了下面的 '*->string*' 形式，*action* 将通过调用 `_all_labels` 函数处理所有标记标签来执行。 除非函数调用引入了新标签，否则无需对标记进行特殊处理。

执行 *actions* 时调用的函数将以 '`-O` *name*' 选项命名的数组元素作为参数。 例如，这可以用来将 `compadd` 内置函数的同一选项集传递给所有 *actions*。

action 的形式如下。

(single unquoted space)

This is useful where an argument is required but it is not possible or desirable to generate matches for it. The *message* will be displayed but no completions listed. Note that even in this case the colon at the end of the *message* is needed; it may only be omitted when neither a *message* nor an *action* is given.

(*item1 item2 ...*)

例如，可能匹配的列表之一：

```
:foo:(foo bar baz)
```

((item1\:desc1 ...))

与上述类似，但对每种可能的匹配都进行了说明。 注意冒号前的反斜杠。 例如

```
:foo:((a\:bar b\:baz))
```

如果上下文中的 values 标记设置了 description 样式，则匹配项将与其描述一起列出。

->*string*

在这种形式中，_arguments 处理参数和选项，然后将控制权返回给调用函数，并设置参数指示处理状态；正在调用的函数然后自行安排生成补全。 例如，实现状态机的函数就可以使用这种操作。

当 _arguments 遇到格式为 '->*string*' 的 *action* 时，它将从 *string* 中删除所有前导和尾部空白，并将数组 state 设置为要执行操作的所有 *string*s 的集合。 数组 state_descr 中的元素将分配给每个 *optarg* 中包含此类 *action* 的 *message* 字段。

默认情况下，与所有其他行为良好的补全函数一样，_arguments 在能够添加匹配时返回状态 0，否则返回非 0。但是，如果给出 -R 选项，_arguments 将返回状态 300，表示 $state 将被处理。

除了 $state 和 $state_descr 之外，_arguments 还会如下所述设置全局参数 'context', 'line' 和 'opt_args' ，并且不会重置任何对 PREFIX 和 words 等特殊参数的修改。 这使得调用函数可以选择重置这些参数或传播参数的更改。

因此，调用 _arguments 时至少有一个动作包含 '->*string*' 的函数，必须声明适当的本地参数：

```
local context state state_descr line
typeset -A opt_args
```

以防止 _arguments 改变全局环境。

{*eval-string*}

括号中的字符串将作为 shell 代码进行计算，以生成匹配结果。 如果 *eval-string* 本身不是以开小括号或开大括号开头，则会在执行前被分割成单独的单词。

= *action*

如果 *action* 以 '= ' （等号后有空格）开头，_arguments 将插入当前上下文中 *argument* 字段的内容，作为 words 特殊数组中新的第一个元素，并递增 CURRENT 特殊参数的值。 这样做的效果是在补全命令行中插入一个虚词，同时不改变补全正在进行的位置。

对于限制 *action* 在命令行中操作的字词的指定符之一（上述双冒号和三冒号形式），这一点最为有用。 一个特殊的用法是，当 *action* 本身导致 _arguments 在一个受限的范围内运行时；有必要使用这个技巧，在 _arguments 的第二次调用的范围内插入一个适当的命令名称，以便能够解析该行。

word...
word...

这涵盖了除上述形式以外的所有其他形式。 如果 *action* 以空格开头，剩余的单词列表将保持不变。

否则，调用时将在第一个单词后添加一些额外的字符串；这些字符串将作为选项传递给 compadd 内置函数。 它们确保 _arguments 指定的状态，尤其是选项和参数的描述，能正确地传递给补全命令。 这些附加参数取自数组参数 'expl'；该参数将在执行 *action* 之前设置，因此可以在 *action* 中引用，通常以 '$expl[@]' 的形式展开，这样可以保留数组中的空元素。

在执行操作时，数组 'line' 将被设置为命令行中的正常参数，即命令行中命令名称之后的字词，不包括所有选项及其参数。 选项存储在关联数组 'opt_args' 中，选项名称为键，参数为值。 默认情况下，值中的所有冒号和反斜线都用反斜线转义，如果一个选项有多个参数（例如，当使用形式为 '**optspec*' 的 *optspec* 时），它们会用（未转义的）冒号连接起来。 但是，如果传递了 -0 选项，则不会执行反斜杠转义，而是用 NUL 字节连接多个值。 例如，在 'zsh -o foo:foo -o bar:bar -o <TAB>' 之后，'opt_args' 的内容将是

```
typeset -A opt_args=( [-o]='foo\:foo:bar\:bar:' )
```

默认，并且

```
typeset -A opt_args=( [-o]=$'foo:foo\x00bar:bar\x00' )
```

如果 _arguments 在调用时使用了 -0 选项。

参数 'context' 在返回调用函数执行形式为 '->*string*' 的操作时被设置。 它被设置为与 $state 的元素相对应的元素数组。 每个元素都是上下文参数字段的合适名称：或者是表示选项 *-opt* 的第 *n* 个参数的形式为 'option-*opt*-*n*' 的字符串，或者是表示第 *n* 个参数的形式为 'argument-*n*' 的字符串。 对于 'rest' 参数，即列表末尾未被 position 处理的参数，*n* 是字符串 'rest'。 例如，补全 -o 选项的参数时，名称为 'option-o-1'，而对于第二个正常（非选项）参数，名称为 'argument-2'。

此外，在执行 *action* 时，会更改 curcontext 参数中的上下文名称，以附加 context 参数中存储的相同的字符串。

选项 -C 会告诉 _arguments 为形式为 '->*state*' 的操作修改 curcontext 参数。这是用于跟踪当前上下文的标准参数。 在此，它（而非 context 数组）应被设置为调用函数的本地参数，以避免传回修改后的值，并应在函数开始时初始化为当前值：

```
local curcontext="$curcontext"
```

当多个状态不可能同时有效时，这种方法就很有用。

选项分组

可以对选项进行分组，以简化排除列表。一个组可以用 '+' 引入，后面的词是该组的名称。在排除列表中可以引用整个组，也可以使用组名来区分同一选项的两种形式。例如

```
_arguments \
    '(group2--x)-a' \
  + group1 \
    -m \
    '(group2)-n' \
  + group2 \
    -x -y
```

如果以 '(*name*)' 的形式指定了一个组的名称，那么该组中只有一个值会被补全；更正式地说，所有规范与该组中的所有其他规范互斥。这对于定义互为别名的选项非常有用。例如:

```
_arguments \
    -a -b \
  + '(operation)' \
    {-c,--compress}'[compress]' \
    {-d,--decompress}'[decompress]' \
    {-l,--list}'[list]'
```

如果命令行中出现了某个组中的选项，该选项就会以 '*group-option*' 为键存储在关联数组 'opt_args' 中。 在上例中，如果命令行中出现选项 '-c' ，则使用键 'operation--c' 。

指定多个参数集

可以指定多个选项和参数集(set)，各个集(set)之间用单个连字符隔开。这与组(group)不同，集与集之间是相互排斥的。

第一组之前和任何一组中的规格对所有组都是通用的。例如

```
_arguments \
    -a \
  - set1 \
    -c \
  - set2 \
    -d \
    ':arg:(x2 y2)'
```

这定义了两个集。 当命令行包含 '-c' 选项时， '-d' 选项和参数将不被视为可能的补全。 当命令行包含 '-d' 或一个参数时， '-c' 选项将不被考虑。 但是，在 '-a' 之后，这两个集仍将被视为有效。

至于组，集合的名称可以单独出现在排除列表中，或者在普通选项或参数规范之前出现。

补全代码必须为每个集分别解析命令行。这可能会很慢，因此只有在必要时才使用集合。 一个有用的替代方法通常是带有其余参数的选项说明（如 '-foo:*:...' 中）；在这里，选项 -foo 会吞掉 *optarg* 定义中描述的所有其余参数。

从帮助输出中派生 *spec* 形式

选项 '--' 允许 _arguments 计算出支持 '--help' 选项的长选项名称，该选项是许多 GNU 命令的标准选项。 在调用命令字时，会使用参数 '--help' ，并检查输出的选项名称。 显然，将该选项传递给可能不支持该选项的命令是很危险的，因为命令的行为是不明确的。

除选项外，当 '*--opt=val*' 形式有效时，'_arguments --' 将尝试推断出选项的参数类型。 还可以通过检查命令的帮助文本并添加形式为 '*pattern*:*message*:*action*' 的 *helpspec* 来提供提示；注意，其他 _arguments *spec* 形式不被使用。 *pattern* 将与选项的帮助文本进行匹配，如果匹配成功，则使用 *message* 和 *action* 作为其他参数指定符。 '*:' 的特殊情况意味着 *message* 和 *action* 都是空的，这将导致在帮助输出中没有说明的选项被排在有说明的选项之前。

例如：

```
_arguments -- '*\*:toggle:(yes no)' \
              '*=FILE*:file:_files' \
              '*=DIR*:directory:_files -/' \
              '*=PATH*:directory:_files -/'
```

这里，‘yes’ 和 ‘no’ 将作为描述是以星号结尾的选项的参数进行补全；选项的描述中包含子串 ‘=FILE’ 的，文件会被补全；选项的描述中包含 ‘=DIR’ 或 ‘=PATH’ 的，目录会被补全。 后三种模式实际上是默认的，因此不需要明确给出，不过可以覆盖这些模式的使用。 使用这一功能的典型帮助文本是:

```
-C, --directory=DIR           转到目录 DIR
```

因此，上述规范将导致目录在 ‘--directory’ 之后补全，但不会在 ‘-C’ 之后补全。

请注意，_arguments 会自动检测选项的参数是否为可选参数。 可以通过在 *message* 前面加双冒号来明确指定。

如果 *pattern* 以 ‘(-)’ 结尾，则会将其从模式中删除，并且 *action* 只会在 ‘=’ 之后直接使用，而不会在下一个词中使用。 这就是以 ‘=-’ 形式定义的正常规范的行为。

默认情况下，命令（带选项 ‘--help’）会在将所有语言类别（LC_CTYPE除外）重置为 ‘C’ 后运行。 如果已知本地化帮助输出有效，可以在 ‘_arguments --’ 之后指定选项 ‘-l’，这样命令就会以当前的本地化语言运行。

在" ‘_arguments --’ 后面可以加上 ‘-i *patterns*’ 选项,以给出不需要补全的选项模式。 模式可以是数组参数的名称，也可以是括号中的字面列表。 例如,

```
_arguments -- -i \
    "(--(en|dis)able-FEATURE*)"
```

将导致补全忽略选项 ‘--enable-FEATURE’ 和 ‘--disable-FEATURE’（本例在使用 GNU configure 时非常有用）。

‘_arguments --’ 形式后还可以加上 ‘-s *pair*’ 选项来描述选项别名。 *pair* 由一系列可选的模式和相应的替换组成，用双引号括起来，以便在 _arguments 调用中形成一个参数词。

例如，某些 configure 脚本的帮助输出仅将选项描述为 ‘--enable-foo’，但脚本也接受否定形式 ‘--disable-foo’。 允许补全第二种形式：

```
_arguments -- -s "((#s)--enable- --disable-)"
```

杂项说明

最后请注意，_arguments 通常希望自己是处理补全的主要函数。 它可能会产生副作用，改变在它之后调用的其他函数所添加的匹配项的处理方式。 要将 _arguments 与其他函数结合使用，应在 _arguments 之前,作为 *spec* 中的 *action* 调用，或在 ‘->*state*’ 操作的处理程序中调用。

下面是使用 _arguments 的一个更普通的示例：

```
_arguments '-l+:left border:' \
           '-format:paper size:(letter A4)' \
           '*-copy:output file:_files::resolution:(300 600)' \
           ':postscript file:_files -g \*.\(ps\|eps\)' \
           '*:page number:'
```

这里描述了三个选项：‘-l’, ‘-format’ 和 ‘-copy’。 第一个选项需要一个参数，描述为‘*left border*’，由于是空操作，因此不会提供补全。 它的参数可以直接位于 ‘-l’ 之后，也可以作为该行的下一个单词给出。

‘-format’ 选项在下一个单词中接收一个参数，描述为 ‘*paper size*’，对于该参数，只会补全 ‘letter’ 和 ‘A4’ 字符串。

‘-copy’ 选项可以在命令行中出现多次，并包含两个参数。 第一个参数是必选参数，将以文件名形式补全。 第二个参数是可选参数（因为在描述符 ‘*resolution*’ 前有第二个冒号），将由字符串 ‘300’ 和 ‘600’ 补全。

后两处说明了应补全的参数。 第一个说明将第一个参数描述为 ‘*postscript file*’，并要求补全以 ‘ps’ 或 ‘eps’ 结尾的文件。 最后一种描述将所有其他参数描述为 ‘*page number*’，但不提供补全。

_cache_invalid *cache_identifier*

如果给定的补全缓存标识符对应的补全缓存需要重建，则此函数返回状态 0。 它通过查找当前上下文的 cache-policy 样式来确定。 它应提供一个函数名，该函数名的唯一参数是相关缓存文件的完整路径。

例如：

```
_example_caching_policy () {
    # rebuild if cache is more than a week old
    local -a oldp
    oldp=( "$1"(Nm+7) )
    (( $#oldp ))
}
```

_call_function *return name* [*arg* ...]

如果存在函数 *name*，则使用参数 *arg*s 调用该函数。 *return* 参数给出了一个参数的名称，函数 *name* 的返回状态应存储在该参数中；如果 *return* 为空或单个连字符，则该参数将被忽略。

如果函数 *name* 存在并被调用，_call_function 本身的返回状态为 0，否则为非 0。

_call_program [-l] [-p] *tag string* ...

该函数为用户提供了一种覆盖外部命令使用的机制。 它使用提供的 *tag* 查找 command 样式。 如果设置了样式，其值将作为要执行的命令。 调用 _call_program 的 *string*s，或样式（如果已设置）中的 *string*s， 会以空格连接起来，并对得到的字符串进行求值。 返回状态是被调用命令的返回状态。

默认情况下，命令的运行环境是通过调用实用程序 _comp_locale（见下文）将所有语言类别（LC_CTYPE除外）重置为 'C'。如果给出选项 '-l'，命令将以当前的本地语言运行。

如果提供了选项 '-p'，则表示命令输出受运行权限的影响。如果 gain-privileges 样式设置为 true，_call_program 将使用 sudo 等命令（如果命令行中存在），以匹配最终命令可能的运行权限。在查找 gain-privileges 和 command 样式时，zstyle 上下文中的命令组件将以斜线 ('/')结尾，之后是用于获取权限的命令。

_combination [-s *pattern*] *tag style spec ... field opts ...*

该函数用于补全值的组合，例如主机名和用户名的组合。 *style* 参数给出了定义成对值的样式；该样式将在指定了 *tag* 的上下文中查找。

样式名称由用连字符分隔的字段名组成，例如 'users-hosts-ports'。 对于已知值的每个字段，都会给出一个格式为 '*field=pattern*' 的 *spec*。 例如，如果命令行指定了用户 'pws'，则参数 'users=pws' 应当出现。

下一个不带等号的参数将作为应生成补全的字段名（可能不是已知值的 *field*s 中的一个）。

生成的匹配结果将取自样式值。 这些值应按适当顺序（如上例中的用户、主机、端口）包含可能的组合值。 不同字段的值用冒号分隔。 可以使用选项 -s 将其改为 _combination，以指定一个模式。 通常情况下，这是一个字符类，例如，在 users-hosts 样式中使用了 '-s "[:@]"'。 每个 '*field=pattern*' 规范都会限制补全,这些补全适用于具有相应匹配字段的样式的元素。

如果没有为给定标记定义具有给定名称的样式，或者样式值中的字符串都不匹配，但定义了带前导下划线的所需字段的函数名，则将调用该函数生成匹配值。 例如，如果没有 'users-hosts-ports' 或在需要主机时没有匹配的主机名，将自动调用函数 '_hosts'。

如果同一名称用于多个字段，在 '*field=pattern*' 和给出要补全的字段名称的参数中，可以在字段名称后给出字段编号（从 1 开始），并用冒号隔开。

在根据样式值生成匹配时，必填字段名后的所有参数都会传递给 compadd，或者在为字段调用函数时传递给这些函数。

_command_names [-e | -]

该函数补全在命令位置有效的单词：别名、内置命令、散列命令、函数等的名称。 使用 -e 标志时，只补全散列命令。 - 标志将被忽略。

_comp_locale

该函数将 LC_CTYPE 以外的所有区域设置类别重置为 'C'，以便补全系统可以轻松分析外部命令的输出。LC_CTYPE 保留当前值（将 LC_ALL 和 LANG 考虑在内），确保文件名中的非 ASCII 字符仍能得到正确处理。

该函数通常只能在子 shell 中运行，因为新的本地语言(locale)会导出到环境中。典型用法是 '$(_comp_locale; *command* ...)'。

_completers [-p]

该函数可补全补全器的名称。

-p

在匹配中包含前导下划线 ('_')。

_default

该函数对应于 -default- 特殊上下文，适用于未定义补全的情况。 在某些错误条件下调用该函数非常有用，例如在未识别子命令后补全。 这将优美降级的概念应用于补全系统，允许它在基本补全文件名等常用内容时回退。

_describe [-12JVx] [-oO | -t *tag*] *descr name1* [*name2*] [*opt* ...]
[-- *name1* [*name2*] [*opt* ...] ...]

此函数将补全与描述关联起来。 可以提供多个用 -- 分隔的组，可能会有不同的补全选项 *opt*s。

如果 descr 标记的 format 样式被设置，则 *descr* 将作为字符串显示在匹配结果的上方。 接下来是一个或两个数组的名称，后面是传递给 compadd 的选项。 数组 *name1* 包含可能的补全及其描述，格式为 '*completion*:*description*'。 *completion* 中任何字面上的冒号都必须用反斜杠引出。 如果给出 *name2*，则其元素个数应与 *name1* 相同；在这种情况下，相应的元素将作为可能的补全元素添加，而不是 *name1* 中的 *completion* 字符串。 补全列表将保留 *name1* 中的描述。 最后，可以出现一补全选项集。

如果第一个参数前出现了选项 '-o'，则添加的匹配字符串将被视为命令选项名称（注意：不是 shell 选项），通常位于命令行上的 '-', '--' 或 '+' 之后。 在这种情况下，_describe 会使用 prefix-hidden、prefix-needed 和 verbose 样式来查找是否应将字符串添加为补全，以及是否显示说明。 如果不使用 '-o' 选项，则只使用 verbose 样式来决定如何显示说明。 如果使用 '-O' 而不是 '-o'，命令选项的补全方式如上，但 _describe 不会处理 prefix-needed 样式。

使用 -t 选项可以指定一个 *tag* 。 默认值为 'values' ，如果使用 -o 选项，则为 'options'。

选项 -1, -2, -J, -V, -x 将传递给 _next_label。

如果通过使用 list-grouped 样式选择，具有相同描述的字符串将一起出现在列表中。

_describe 使用 _all_labels 函数来生成匹配结果，因此它无需出现在标记标签的循环中。

_description [-x] [-12VJ] *tag name descr* [*spec* ...]

不要将此函数与前一个函数混淆；它是用于为创建 compadd 的选项的辅助函数。它被埋藏在许多高级补全函数中，因此通常不需要直接调用。

下面列出的样式将在当前上下文中使用给定的 *tag* 进行测试。 compadd 产生的选项会被放入名为 *name* 的数组（传统上是 'expl'，但这一约定并不强制执行）。 相应匹配集的描述信息将以 *descr* 的形式传递给函数。

测试的样式有 format、hidden、matcher、ignore-line、ignored-patterns、group-name 和 sort。 首先针对给定的 *tag* 对 format 样式进行测试，如果未找到值，则针对 descriptions 标签进行测试，其余的只针对作为第一个参数给定的标签进行测试。 函数还会调用 _setup，以测试更多样式。

由 format 样式（如果有）返回的字符串将被修改，使 '%d' 序列被作为第三个参数给出的 *descr* 所替换，且不留任何前导或尾部空白。 如果删除空白后，*descr* 为空字符串，则格式样式将不被使用，放入 *name* 数组的选项将不包含显示在匹配结果上方的解释字符串。

如果调用 _description 时有三个以上的参数，则附加的 *specs* 应为 '*char*:*str*' 形式。 这些参数将为 format 样式提供转义序列替换：每次出现 '%*char*' 时，都会被 *string* 替换。 如果没有给出额外的 *specs*，但 *descr* 中的描述符合通用格式，则会为描述中的元素设置更多转义序列。 这些元素分别对应默认值 ('%o')、可接受值的单位('%m') 范围和描述 ('%h')的其余初始部分。 描述的形式包括在括号中指定单位和范围，和在方括号中指定的默认值，例如：

```
_description times expl 'timeout (seconds) (0-60) [20]'
```

在为这些元素设计样式时，可以使用 zformat 条件表达式。因此，举例来说，要添加 'default:' 作为标记，但只在有默认值要显示时，format 样式可能包括 '%(o.default: %o.)'。

如果给出 -x 选项，描述将使用 -x 选项而不是默认的 -X 传递给 compadd。 这意味着即使没有相应的匹配，描述也会显示。

放在 *name* 数组中的选项会考虑 group-name 样式，因此必要时会将匹配结果放在一个单独的组中。 通常情况下，组中的元素是排序的（通过向 compadd 传递 -J 选项），但如果向 _description 传递了以‘-V’, ‘-J’, ‘-1’ 或 ‘-2’ 开头的选项，则该选项将包含在数组中。 因此，可以通过给出选项 ‘-V’, ‘-1V’ 或 ‘-2V’ 来取消补全组的排序。

在大多数情况下，函数的使用方法是这样的：

```
local expl
_description files expl file
compadd "$expl[@]" - "$files[@]"
```

请注意参数 expl、连字符和匹配列表的使用。 补全系统中几乎所有对 compadd 的调用都使用了类似的格式；这样可以确保用户指定的样式正确地传递给实现补全内部结构的内置程序。

_dir_list [-s *sep*] [-S]

补全一个用冒号分隔的目录名列表（格式与 $PATH相同）。

-s *sep*

使用 *sep* 作为项目之间的分隔符。 *sep* 默认为冒号 (‘:’)。

-S

添加 *sep* 代替斜线 (‘/’)，作为可自动删除的后缀。

_dispatch *context string ...*

此操作会将当前上下文设置为 *context*，并通过搜索作为 *string*s 给出的命令名称或特殊上下文（如上文针对 compdef 所述）列表来查找处理此上下文的补全函数。 第一个为列表中某个上下文定义的补全函数将用于生成匹配。 通常情况下，最后一个 *string* 是 -default-，以便使用默认补全函数作为备用。

该函数将参数 $service 设置为正在尝试的 *string*，并将 $curcontext 参数的 *context/command* 字段（第四字段）设置为作为第一个参数给定的 *context*。

_email_addresses [-c] [-n *plugin*]

补全电子邮件地址。 地址由插件提供。

-c

补全裸 localhost@domain.tld 地址，不含名称部分或注释。 如果不使用该选项，RFC822 中的‘*Firstname Lastname* <*address*>’ 字符串将被补全。

-n *plugin*

补全来自 *plugin* 的别名。

默认情况下可使用以下插件：`_email-ldap`（参见 `filter` 样式）、`_email-local`（补全 *user@hostname* Unix 地址）、`_email-mail`（补全 `~/.mailrc` 中的别名）、`_email-mush`、`_email-mutt` 和 `_email-pine`。

来自 `_email-`*foo* 插件的地址会添加到标记 '`email-`*foo*' 下。

编写插件

插件以单独函数的形式编写，名称以 '`_email-`' 开头。 调用时使用 `-c` 选项和 `compadd` 选项。 它们应自行补全或将 `$reply` 数组设置为 '*alias*`:`*address*' 元素列表，并返回 `300`。 新插件将被拾取并自动运行。

`_files`

函数 `_files` 是 `_path_files` 的包装器。它支持所有相同的功能，并做了一些改进 — 尤其是，它尊重 `list-dirs-first` 样式，并允许用户使用 `file-patterns` 样式覆盖 `-g` 和 `-/` 选项的行为。因此，在大多数情况下，`_files` 比 `_path_files` 更受青睐。

该函数接受 `_path_files` 允许的全部选项，具体描述如下。

`_gnu_generic`

该函数是上述 `_arguments` 函数的一个简单封装。 它可用于自动确定命令所能理解的长选项，这些命令在传递选项 '`--help`' 时会产生一个列表。 它本身就是一个顶级补全函数。 例如，要启用 `foo` 和 `bar` 命令的选项补全功能，请使用

```
compdef _gnu_generic foo bar
```

在调用 `compinit` 后。

补全系统在使用该函数时比较保守，因为必须确保命令理解选项 '`--help`'。

`_guard` [*options*] *pattern descr*

如果 *pattern* 与要补全的字符串匹配，该函数将显示 *descr*。 该函数用于 *action* 中传递给 `_arguments` 和类似函数的规范。

如果信息已显示且待补全字不为空，则返回状态为零，否则为非零。

在 *pattern* 之前可以加上 `compadd` 可以理解的任何选项，这些选项是从 `_description` 传递下来的，即 `-M`、`-J`、`-V`、`-1`、`-2`、`-n`、`-F` 和 `-X`。 所有这些选项都将被忽略。 这与 `_arguments` 动作的参数传递约定非常吻合。

例如，考虑一个命令，包含 `-n` 和 `-none` 两个选项，其中 `-n` 必须在同一个单词中后跟一个数值。通过使用：

```
_arguments '-n-: :_guard "[0-9]#" "numeric value"' '-none'
```

_arguments 可以同时显示‘numeric value’和‘-n<TAB>’之后的补全选项。 如果‘-n’后面已经有一个或多个数字（传给 _guard 的模式），则只显示信息；如果‘-n’后面有其他字符，则只补全选项。

_message [-r12] [-VJ *group*] *descr*
_message -e [*tag*] *descr*

descr 的使用方式与 _description 函数的第三个参数相同，不同的是，无论是否生成了匹配结果，都会显示生成的字符串。 这对于在无法生成补全的地方显示帮助信息非常有用。

format 样式与 messages 标记一起检查，以查找消息；只有在前者未设置样式时，才会使用通常的标签 descriptions。

如果给出 -r 选项，则不使用任何样式；*descr* 将作为要显示的字面字符串。 当 *descr* 来自预处理过的参数列表，而该参数列表已包含扩展的描述时，该选项最为有用。 请注意，此选项不会禁用 compadd 的‘%’序列解析。

-12VJ 选项和 *group* 被传递给 comppadd，从而决定信息字符串被添加到哪个组。

第二个 -e 形式为带有 *tag* 标记的补全提供说明，即使该标记没有匹配项也会显示。 如果选项说明中没有任何操作，_arguments 将调用该形式。 标记可以省略，如果省略，则标记取自参数 $curtag；该参数由补全系统维护，因此通常是正确的。 需要注意的是，如果调用此函数时没有匹配项，compstate[insert] 将被清除，因此不会在命令行中插入随后生成的匹配项。

_multi_parts [-i] *sep array*

参数 *sep* 是一个分隔字符。 *array* 可以是一个数组参数的名称，也可以是一个形式为‘(foo bar)’的字面数组，即一个用括号括起来的以空白分隔的单词列表。 可能的补全是数组中的字符串。 不过，由 *sep* 分隔的每个词块都将单独补全。 例如，_tar 函数使用‘_multi_parts / *patharray*’来从给定的完整文件路径数组中补全部分文件路径。

-i 选项会使 _multi_parts 插入唯一匹配，即使需要插入多个分隔符。 这通常不是文件名的预期行为，但某些其他类型的补全，例如有固定可能性的补全集，可能更适合这种形式。

与其他实用用函数一样，该函数接受‘-V’, ‘-J’, ‘-1’, ‘-2’, ‘-n’, ‘-f’, ‘-X’, ‘-M’, ‘-P’, ‘-S’, ‘-r’, ‘-R’和‘-q’选项，并将它们传递给 compadd 内置函数。

_next_label [-x] [-12VJ] *tag name descr* [*option* ...]

该函数用于实现上述在 tag-order 样式的特定标记的不同标记(tag labels)上的循环。 每次调用时，它都会检查是否还有其他标签；如果有，则返回状态 0，否则返

回非 0。 由于该函数需要设置一个当前标签，因此它必须始终跟随 _tags 或 _requested 的调用。

-x12VJ 选项和前三个参数会传递给 _description 函数。 在此调用中，*tag* 将酌情替换为标签(tag label)。 任何以 tag-order 样式给出的描述都优先于传递给 _next_label 的 *descr*。

在 *descr* 后面给出的 *options* 是在 *name* 给出的参数中设置的，因此应传递给 compadd 或任何调用以用于添加匹配的函数。

下面是该函数在标记 foo 中的典型用法。 对 _requested 的调用确定是否需要 foo 标记；对 _next_label 的循环,处理为 tag-order 样式中的标记定义的标签（labels）。

```
local expl ret=1
...
if _requested foo; then
  ...
  while _next_label foo expl '...'; do
    compadd "$expl[@]" ... && ret=0
  done
  ...
fi
return ret
```

_normal [-P | -p *precommand*]

这是一个标准函数，用于处理任何特殊 *-context-* 以外的补全。 它既可用于补全命令字，也可用于补全命令的参数。 在第二种情况下，_normal 会查找该命令的特殊补全，如果没有，则使用 -default- 上下文的补全。

第二个用途是在 $words 数组和 $CURRENT 参数被修改后，重新检查它们指定的命令行。 例如，函数 _precommand 在 nohup 等前置命令指定符之后补全，它会删除 words 数组中的第一个单词，递减 CURRENT 参数，然后调用 '_normal -p $service'。 其效果是： 'nohup *cmd ...*' 的处理方式与 '*cmd ...*' 相同。

-P

重置前置命令列表。如果补全的命令行允许使用内部命令（如内置命令和函数）而不考虑先前的前置命令（如 'zsh -c'），则应使用该选项。

-p *precommand*

将 *precommand* 追加到前置命令列表中。该选项几乎适用于 -P 不适用的所有情况。

如果命令名与 compdef 的 -p 或 -P 的选项之一给出的模式之一匹配，则调用相应的补全函数，然后检查参数 _compskip。 如果参数 _compskip 被设置，即使未找到匹配结果，也会在此时终止补全。 这与 -first- 上下文中的效果相同。

_numbers [*option* ...] [*description*] [*suffix* ...]

当一个数字后面有一个表示单位的后缀时，就可以使用这种方法。 单位后缀可以补全，也可以包含在为前一个数字调用补全时使用的描述中。

除了常用的 compadd 选项外，_numbers 还接受以下选项：

-t *tag*

指定一个要使用的标记，而不是默认的 numbers。

-u *units*

指出数字的默认单位，例如 字节。

-l *min*

指定数字的最小可能值。

-m *max*

指定数字的最大可能值。

-d *default*

指定默认值。

-N

允许负数。 如果范围中包含负数,那么这是隐含的。

-f

允许使用小数。

如果某个后缀代表一个数字的默认单位，则应在其前加上冒号。 此外，后缀后面还可以加上冒号和说明。 因此，举例来说，下面可以指定某事物的年龄，可以用秒或可选的后缀来表示更长的时间单位：

```
_numbers -u seconds age :s:seconds m:minutes h:hours d:days
```

补全时，按大小顺序排列单位通常会有帮助。 为方便起见，我们保留了给出单位的顺序。

使用 descriptions 标记或 -t 指定的标记查找 format 样式时，后缀列表可作为 ‘%x’ 转义序列使用。这是 format 样式下记录的常规序列之外的另一种形式。该列

表的形式也可以配置。为此，首先使用 unit-suffixes 标记查找 format 样式。检索到的格式依次应用于每个后缀，然后将结果连接起来，形成补全列表。对于 unit-suffixes 格式，'%x' 表示单个后缀， '%X' 表示其描述。 '%d' 表示默认后缀，可在条件中使用。索引和反向索引分别设置在 '%i' 和 '%r' 中，适用于只包含列表中第一和最后一个后缀的文本。例如，下文将后缀以逗号分隔的列表形式连接在一起：

```
zstyle ':completion:*:unit-suffixes' format '%x%(r:,)'
```

_options

可用于补全 shell 选项的名称。 它提供了一个忽略前导 'no'、忽略下划线并允许大写字母匹配小写字母的匹配规范（例如，'glob', 'noglob', 'NO_GLOB' 均可补全）。 任何参数都会传播给 compadd 内置函数。

_options_set and _options_unset

这些函数只补全设置或取消设置的选项，其匹配规范与 _options 函数中使用的相同。

请注意，要使这些函数正常工作，需要取消 _main_complete 函数中的几行注释。 在补全小部件本地设置所需的选项之前，这些行用于存储有效的选项设置。 因此，补全系统一般不会使用这些函数。

_parameters

用于补全 shell 参数的名称。

选项 '-g *pattern*' 限制补全的参数类型与 *pattern* 匹配。 参数的类型由 'print ${(t)*param*}' 显示，因此在 *pattern* 中明智地使用 '*' 可能是必要的。

所有其他参数都将传递给 compadd 内置函数。

_path_files

该函数在整个补全系统中用于补全文件名。 它允许补全部分路径。 例如，字符串 '/u/i/sig' 可以补全为 '/usr/include/sys/signal.h'。

_path_files 和 _files 都接受的选项有:

-f

补全所有文件名。 这是默认设置。

-/

指定只补全目录。

-g *pattern*

指定只补全与 *pattern* 匹配的文件。

`-W` *paths*

指定路径前缀，这些路径前缀将被放到命令行字符串前以生成文件名，但不会作为补全插入，也不会显示在补全列表中。 在这里，*paths* 可以是数组参数的名称、用括号括起来的（字面值的）路径列表或绝对路径名。

`-F` *ignored-files*

行为与 `compadd` 内置程序的相应选项相同。 它可以直接控制哪些文件名应被忽略。 如果没有该选项，则使用 `ignored-patterns` 样式。

`_path_files` 和 `_files` 还接受以下传递给 `compadd` 的选项：'`-J`', '`-V`', '`-1`', '`-2`', '`-n`', '`-X`', '`-M`', '`-P`', '`-S`', '`-q`', '`-r`' 和 '`-R`'。

最后，`_path_files` 函数使用了上述 `expand`、`ambiguous`、`special-dirs`、`list-ffixes` 和 `file-sort` 样式。

`_pick_variant` [`-b` *builtin-label*] [`-c` *command*] [`-r` *name*]
label=*pattern* ... *label* [*arg* ...]

该函数用于解决单个命令名称需要不止一种处理方式的情况，原因可能是该命令名称有多个变体，或者是两个不同命令之间存在名称冲突。

要运行的命令取自 `words` 数组的第一个元素，除非被选项 `-c` 改写。 命令运行后，其输出将与一系列模式进行比较。 要传递给命令的参数可以在所有其他参数之后指定。 要依次尝试的模式由参数 *label*=*pattern* 提供；如果 '*command arg* ...' 的输出包含 *pattern*，那么 *label* 将被选为命令变量的标签。 如果没有模式匹配，则选择最后的命令标签并返回状态 1。

如果给出 '`-b` *builtin-label*' 则会测试该命令是否作为 shell 内置命令（可能是自动加载的）提供；如果是，则会选择 *builtin-label* 作为变量的标签。

如果给出 '`-r` *name*'，则选中的 *label* 将存储在名为 *name* 的参数中。

结果也会缓存在 `_cmd_variant` 关联数组中，该数组以运行命令的名称为索引。

`_regex_arguments` *name spec* ...

该函数生成一个补全函数 *name*，该函数与 *spec*s（如下所述的一组正则表达式）的规范相匹配。 运行 `_regex_arguments` 后，应将函数 *name* 作为普通补全函数调用。 要匹配的模式由 `words` 数组中直到当前光标位置的内容和空字符组成；不使用引号。

参数以备选方案集的形式分组，中间用 '`|`' 隔开，一个接一个地测试，直到有一个匹配为止。 每个备选方案由一个或多个规范组成，从左到右依次测试，每个匹配的模式都会从被测试的命令行中依次剥离，直到该组全部成功或有一个失败为止；在后

一种情况下，将测试下一个备选方案。 通过使用括号，这种结构可以重复到任意深度；匹配从内向外进行。

如果没有测试成功，但剩余的命令行字符串不包含空字符（意味着剩余的单词就是要生成补全的单词），则会应用一个特殊程序。 补全目标仅限于剩余的单词，并执行相应模式的任何 *actions*。 在这种情况下，不会从命令行字符串中剥离任何内容。 *actions* 的计算顺序可以由 `tag-order` 样式决定；`_alternative`支持的各种格式都可以在 *action* 中使用。 *descr* 用于设置数组参数 `expl`。

规范参数有以下几种形式，其中 '(', ')', '#' 和 '|' 等元字符应加引号。

/pattern/ [%*lookahead*%] [-*guard*] [:*tag*:*descr*:*action*]

> 这是一个单一的原始组件。 该函数测试组合模式 '`(#b)((#B)`*pattern*`)`*lookahead**' 是否与命令行字符串匹配。 如果匹配，将计算 '*guard*' ，并检查其返回状态，以确定测试是否成功。 *pattern* 字符串 '`[]`' 保证永远不会匹配。 在检查下一个模式之前，*lookahead* 不会从命令行中删除。
>
> 以 : 开头的参数的使用方式与 `_alternative` 的参数相同。
>
> 组件的使用方法如下： 检测 *pattern* ,以查看组件是否已存在于命令行中。 如果存在，则会检查下面的任何规范，以找到需要补全的内容。 如果找到了一个组件，但命令行中还没有这样的模式，则会使用包含 *action* 的字符串来生成匹配项，并在此处插入。

/pattern/+ [%*lookahead*%] [-*guard*] [:*tag*:*descr*:*action*]

> 这与 '*/pattern/ ...*' 类似，但命令行字符串的左侧部分（即已被前面的模式匹配的部分）也被视为补全目标的一部分。

/pattern/- [%*lookahead*%] [-*guard*] [:*tag*:*descr*:*action*]

> 这类似于 '*/pattern/ ...*' ，但是当前和先前匹配模式的 *actions* 会被忽略，即使接下来的 '*pattern*' 匹配空字符串。

(*spec*)

> 括号可用于对 *specs* 进行分组；请注意，每个括号都是 `_regex_arguments` 的一个参数。

spec #

> 这允许 *spec* 有任意次数的重复。

spec spec

> 如上所述，两个 *spec* 将相继匹配。

spec | *spec*

两个 *specs* 中的任何一个都可以匹配。

函数 _regex_words 可用作辅助函数，为一组备选词语生成匹配结果，这些词语可能带有自己的参数作为命令行参数。

例如：

```
_regex_arguments _tst /$'[^\0]#\0'/ \
    /$'[^\0]#\0'/ :'compadd aaa'
```

这会生成一个函数 _tst，该函数的唯一参数是 aaa。 为简洁起见，省略了动作的 *tag* 和 *description*（这样做可行，但不建议在正常使用中这样做）。 第一个组件匹配任意的命令字；第二个组件匹配任何参数。 由于参数也是任意的，因此后面的任何组件都不取决于 aaa 是否存在。

```
_regex_arguments _tst /$'[^\0]#\0'/ \
    /$'aaa\0'/ :'compadd aaa'
```

这是一种更典型的用法；它与此相似，但只有在 aaa 作为第一个参数出现时，才会匹配下面的任何模式。

```
_regex_arguments _tst /$'[^\0]#\0'/ \( \
    /$'aaa\0'/ :'compadd aaa' \
    /$'bbb\0'/ :'compadd bbb' \) \#
```

在本例中，可以补全不确定数量的命令参数。 奇数参数补全为 aaa，偶数参数补全为 bbb。 除非当前参数之前的 aaa 和 bbb 参数集匹配正确，否则补全失败。

```
_regex_arguments _tst /$'[^\0]#\0'/ \
    \( /$'aaa\0'/ :'compadd aaa' \| \
    /$'bbb\0'/ :'compadd bbb' \) \#
```

这一点类似，但 aaa 或 bbb 可以为任意参数补全。 在这种情况下，可以使用 _regex_words 为参数生成合适的表达式。

_regex_words *tag description spec ...*

该函数可用于为 _regex_arguments 命令生成参数，这些参数可插入到任何需要规则集的地方。 *tag* 和 *description* 给出了与当前上下文相关的标准标记和描述。 每个 *spec* 包含两个或三个参数，参数之间用冒号隔开：注意，这里没有前导冒号。

每个 *spec* 都给出了此时可能补全的词的集中的一个，和参数一起。 因此，它大致等同于在普通（非 regex）补全中使用的 _arguments 函数。

spec 中第一个冒号之前的部分是要补全的单词。 其中可能包含一个 *；整个单词，在 * 前后的内容，都会被补全，但只有 * 之前的内容才需要进行上下文匹配，因此在缩写形式之后还可以补全其他参数。

spec 的第二部分是对正在补全的单词的描述。

spec 的第三部分是可选的，它描述了被补全词语后面的词语如何补全。 为了避免引号问题，将对其进行计算。 这意味着它通常包含对一个数组的引用，该数组包含先前生成的正则表达式参数。

选项 `-t` *term* 指定了单词的结束符，而不是通常的空格。这将作为一个可自动删除的后缀来处理，就像 `_values` 的选项 `-s` *sep* 一样。

`_regex_words` 的处理结果会被放入数组 `reply`，该数组应为调用函数的本地数组。 如果词组和参数的集可能重复匹配，则应在当前生成的数组中添加 #。

例如：

```
local -a reply
_regex_words mydb-commands 'mydb commands' \
  'add:add an entry to mydb:$mydb_add_cmds' \
  'show:show entries in mydb'
_regex_arguments _mydb "$reply[@]"
_mydb "$@"
```

这里显示的是命令 `mydb` 的补全函数，该命令需要两个命令参数：`add` 和 `show`。`show` 不需要参数，而 `add` 的参数已经在数组 `mydb_add_cmds` 中准备好了，很可能是之前调用 `_regex_words` 时准备的。

`_requested` [`-x`] [`-12VJ`] *tag* [*name descr* [*command* [*arg ...*]]

调用此函数是为了判断用户是否请求了 `_tags`（见下文）中已经注册的标记，并因此需要对其进行补全。 如果该标记被请求，则返回状态 0，否则返回非 0。 该函数通常作为不同标记循环的一部分使用，如下所示：

```
_tags foo bar baz
while _tags; do
  if _requested foo; then
    ... # perform completion for foo
  fi
  ... # test the tags bar and baz in the same way
  ... # exit loop if matches were generated
done
```

请注意，在 `_tags` 循环结束之前，不会对是否生成匹配进行测试。 这样用户就可以设置 `tag-order` 样式，指定同时补全一组标记。

如果给出 *name* 和 *descr*，_requested 就会调用 _description 函数，其中包含这些参数和传给 _requested 的选项。

如果给出 *command*，将立即调用带有相同参数的 _all_labels 函数。 在简单的情况下，这样就可以一次性完成标记测试和匹配。 例如:

```
local expl ret=1
_tags foo bar baz
while _tags; do
  _requested foo expl 'description' \
      compadd foobar foobaz && ret=0
  ...
  (( ret )) || break
done
```

如果 *command* 不是 comppadd，则必须准备处理相同的选项。

_retrieve_cache *cache_identifier*

此函数从 *cache_identifier* 指定的文件中获取补全信息，该文件存储在 cache-path 样式指定的目录中，默认为 ~/.zcompcache。 如果检索成功，返回状态为零。 只有设置了 use-cache 样式，才会尝试检索，因此调用此函数时无需担心用户是否希望使用缓存层。

更多详情，请参阅下面的 _store_cache。

_sep_parts

该函数的参数是交替传递的数组和分隔符。 数组指定由分隔符分隔的字符串部分的补全。 数组可以是数组参数的名称，也可以是括号中的加引号单词的列表。 例如，使用数组 'hosts=(ftp news)' 调用 '_sep_parts '(foo bar)' @ hosts' 将把字符串 'f' 补全为 'foo' ，把字符串 'b@n' 补全为 'bar@news'。

该函数接受 compadd选项 '-V', '-J', '-1', '-2', '-n', '-X', '-M', '-P', '-S', '-r', '-R' 和 '-q' ，并将它们传递给用于添加匹配的 compadd 内置命令。

_sequence [-s *sep*] [-n *max*] [-d] *function* [-] ...

该函数是其他函数的包装器，用于补全分隔列表中的项目。同一个函数用于补全列表中的每个项目。分隔符用 -s 选项指定。如果省略 -s，将使用 ','。除非指定 -d，否则不会匹配重复值。如果列表中有固定或最多的条目数，可以使用 -n 选项来指定。

常用的 compadd 选项会传递给函数。compadd 也可以直接与 _sequence 搭配使用，不过 _values 在这种情况下可能更合适。

_setup *tag* [*group*]

此函数为第一个参数 *tag* 适当设置补全系统使用的特殊参数。 它使用 list-colors、list-packed、list-rows-first、last-prompt、accept-exact、menu 和 force-list 样式。

可选的 *group* 提供匹配结果所在的组名。 如果没有给出，则使用 *tag* 作为组名。

该函数由 _description 自动调用，因此通常不会明确调用。

_store_cache *cache_identifier param ...*

该函数与 _retrieve_cache 和 _cache_invalid 一起实现了一个缓存层，可用于任何补全函数。 通过代价高昂的操作获得的数据存储在参数中；然后，该函数会将这些参数的值转储到一个文件中。 然后，即使在 shell 的不同实例中，也可以通过 _retrieve_cache 从文件中快速检索数据。

cache_identifier 指定了要转储数据的文件。 文件存储在 cache-path 样式指定的目录中，默认为 ~/.zcompcache。 其余 *param*s 参数是要转储到文件中的参数。

如果存储成功，返回状态为零。 只有设置了 use-cache 样式，函数才会尝试存储，因此可以调用该函数，而不必担心用户是否希望使用缓存层。

当补全函数已经有了作为参数的补全数据时，它可能会避免调用 _retrieve_cache。 不过，在这种情况下，它应该调用 _cache_invalid 来检查参数和缓存中的数据是否仍然有效。

有关缓存层用法的简单示例，请参见 _perl_modules completion 函数。

_tags [[-C *name*] *tag ...*]

如果调用时带有参数，这些参数将被视为当前上下文中有效的补全标记名称。 这些标记将存储在内部，并使用 tag-order 样式进行排序。

接下来，_tags 会在不带参数的情况下被同一个补全函数反复调用。 这将依次选择用户请求的第一、第二等标记的集。 如果至少有一个标记被请求，则返回状态为 0，否则为非 0。 要测试是否要尝试某个标记，应调用 _requested 函数（见上文）。

如果给出 '-C *name*'，则 *name* 会在调用 _tags 时暂时保存在 curcontext 参数中上下文的 *argument* 字段（第五字段）中；退出时会恢复该字段。 这样，_tags 就可以使用更具体的上下文，而无需更改和重置 curcontext 参数（这有相同效果）。

_tilde_files

与 _files类似，但会根据文件名扩展规则解决前导 tildes 问题，因此即使命令行上的文件名以 '~' 开头，建议的补全也不会以 '~' 开头。

_values [-O *name*] [-s *sep*] [-S *sep*] [-wC] *desc spec ...*

用于补全任意关键字（值）及其参数，或此类组合的列表。

如果第一个参数是选项 '-O *name*'，其使用方式将与 _arguments 函数相同。 换句话说，在执行操作时，*name* 数组中的元素将传递给 compadd。

如果第一个参数（或 '-O *name*' 后的第一个参数）是 '-s'，则下一个参数将作为分隔多个值的字符。 该字符会以自动可删除的方式（见下文）自动添加到每个值之后；与使用 _arguments 的补全方式不同，所有通过 '_values -s' 补全的值在命令行中都显示为同一个单词。 如果不使用该选项，每个单词将只补全一个值。

通常情况下，_values 只使用当前单词来确定哪些值已经存在于命令行中，因此不需要再次补全。 如果给出 -w 选项，其他参数也会被检查。

第一个非选项参数 *desc* 用作为一个字符串，在列出值之前作为描述打印。

所有其他参数描述可能的值及其参数的格式与 _arguments 函数描述选项的格式相同（见上文）。 唯一不同的是，开头不需要负号或正号，值只能有一个参数，并且不支持以等号开头的操作形式。

通过选项 -S（类似于 -s，后面跟着用作下一个参数分隔符的字符），可以设置字符,将值与其参数分隔开。默认情况下，等号将被用作值和参数之间的分隔符。

例如：

```
_values -s , 'description' \
        '*foo[bar]' \
        '(two)*one[number]:first count:' \
        'two[another number]::second count:(1 2 3)'
```

这描述了三种可能的值： 'foo', 'one' 和 'two'。 第一个值被描述为 'bar'，不需要参数，可能出现多次。 第二个描述为'number'，可能出现不止一次，需要一个强制参数，描述为'first count'；没有指定操作，因此不会补全。 开头的 '(two)' 表示，如果该行出现了值'one' ，那么值 'two' 将不再被视为可能的补全。 最后，最后一个值（'two'）被描述为 'another number' ，并接受一个被描述为 'second count' 的可选参数，其补全（将出现在 '=' 之后）为 '1', '2' 和 '3'。 _values 函数将补全由逗号分隔的这些值组成的列表。

与 _arguments一样，该函数在执行 *action* 时也会在当前上下文的参数元素（第五个）中临时添加另一个上下文名称组件。 这里这个名称只是补全参数的值的名称。

verbose 样式用于决定是否打印值的说明（但不打印参数的说明）。

关联数组 val_args 用于报告值及其参数；其工作方式与 _arguments 使用的 opt_args 关联数组类似。 因此，调用 _values 的函数应声明本地参数 state、state_descr、line、context 和 val_args：

```
local context state state_descr line
typeset -A val_args
```

当使用形式为 '->*string*' 的操作时。 使用此函数时，context 参数将被设置为要补全的值的名称。 请注意，对于 _values，state 和 state_descr 是标量而不是数组。 只会返回一个匹配的状态。

还需注意的是，_values 通常会在值之间添加分隔符，作为自动删除的后缀（类似于目录后的 '/'）。 但是，'->*string*' 操作不可能这样做，因为参数的匹配是通过调用函数生成的。 要获得通常的行为，调用函数可以通过直接或间接传递选项 '-qS *x*' 给 compadd，以添加分隔符 *x* 作为后缀。

选项 -C 的处理方式与 _arguments 相同。 在这种情况下，应将参数 curcontext 本地化，而不是 context（如上所述）。

_wanted [-x] [-C *name*] [-12VJ] *tag name descr command* [*arg* ...]

在许多情况下，补全只能生成一个特定的匹配集，通常对应于一个标记。 但是，仍有必要确定用户是否需要这种类型的匹配。 在这种情况下，该函数就非常有用。

_wanted 的参数与 _requested 的参数相同，即传递给 _description 的参数。不过，在这种情况下，*command* 并非可选参数；所有的标记处理，包括标记和标记标签上的循环以及匹配结果的生成，都由 _wanted 自动完成。

因此，只提供一个标记，并立即添加与给定描述相对应的匹配项：

```
local expl
_wanted tag expl 'description' \
    compadd -- match1 match2...
```

另请参阅 动态命名目录 示例函数中 _wanted 的使用。

需要注意的是，与 _requested 一样，*command* 必须能够接受向 compadd 传递的选项。

与 _tags 一样，该函数支持 -C 选项，以便为参数上下文字段赋予不同的名称。 -x 选项的含义与 _description 相同。

_widgets [-g *pattern*]

此函数补全 zle 小部件的名称（参见 Zle 小部件）。 如果存在 *pattern*，则将与 $widgets 特殊参数的值进行匹配，该特殊参数在 zsh/zleparameter 模块 中进行了记录。

20.7 补全系统变量

有一些标准变量，由 _main_complete 函数初始化，然后在其他函数中使用。

标准变量是：

_comp_caller_options

> 补全系统使用 setopt 来设置一系列选项。这样，在编写函数时就无需考虑是否与所有可能的用户选项组合兼容。不过，有时补全需要知道用户的选项偏好。这些选项保存在 _comp_caller_options 关联数组中。选项名称以小写字母拼写，不含下划线，并映射到字符串 'on' 和 'off' 中的一个或多个。

_comp_priv_prefix

> _sudo 等补全函数可将 _comp_priv_prefix 数组设置为命令前缀，然后 _call_program 在调用程序时可使用该命令前缀来匹配权限，从而生成匹配结果。

_main_complete 函数还提供了另外两个功能。 数组 compprefuncs 和 comppostfuncs 可以包含在尝试补全之前或之后要立即调用的函数名称。 除非函数明确重新插入数组，否则只会被调用一次。

20.8 补全目录

在源代码发布版中，这些文件包含在 Completion 目录的各个子目录中。 这些文件可能安装在同一结构中，也可能安装在同一个函数目录中。 以下是原始目录结构中文件的说明。 如果要修改已安装的文件，需要将其复制到 fpath 中比标准目录更靠前的目录。

Base

> 核心函数和特殊补全小部件会自动绑定到按键上。 您肯定会用到其中的大部分，但可能不需要对它们进行修改。 其中许多已在上文记录。

Zsh

> 用于补全 shell 内置命令参数的函数，以及用于此的实用函数。 Unix 目录中的函数也会使用其中的一些函数。

Unix

> 用于补全外部命令和命令集参数的函数。 这些函数可能需要针对系统进行修改，不过在许多情况下，它们会尝试判断命令的版本。 例如，mount 命令的补全会尝试确定该命令所运行的系统，而许多其他实用程序的补全则会尝试确定该命令是否使用了 GNU 版本，从而确定是否支持 --help 选项。

X, AIX, BSD, ...

补全和实用函数，用于仅在某些系统上可用的命令。 这些目录并不是按层次排列的，因此，例如 Linux 和 Debian 目录以及 X 目录都可能对你的系统有用。

---

21 用 compctl 补全

21.1 补全的类型

此版本的 zsh 有两种方法来补全命令行上的单词。 shell 的新用户可能更倾向于使用基于 shell 函数的更新、更强大的系统；补全系统将对此进行介绍，而支持该系统的基本 shell 机制将在 补全小部件中进行介绍。 本章将介绍较旧的 compctl 命令。

21.2 说明

compctl [-CDT] *options* [*command ...*]
compctl [-CDT] *options* [-x *pattern options* - ... --]
[+ *options* [-x ... --] ... [+]] [*command ...*]
compctl -M *match-specs ...*
compctl -L [-CDTM] [*command ...*]
compctl + *command ...*

根据提供的 *options* 集来控制编辑器的补全行为。 各种编辑命令，特别是通常与 tab 绑定的 expand-or-complete-word，会尝试补全用户键入的单词，而其他命令，特别是在 EMACS 编辑模式下通常与 ^D 绑定的 delete-char-or-list，则会列出各种可能性；compctl 控制这些可能性是什么。 例如，它们可以是文件名（最常见的情况，因此也是默认情况）、shell 变量或用户指定列表中的单词。

21.3 命令标志

命令参数的补全可以因命令而异，也可以使用默认值。补全命令字本身时的行为也可以单独指定。 这些与下列标志和参数相对应，所有这些标志和参数（-L除外）都可以与随后在 选项标志 中描述的 *options* 的任意组合结合使用：

command ...

控制已命名命令的补全，这些命令必须列在命令行的最后。 如果对路径名包含斜线的命令尝试补全，但未找到补全定义，则使用最后一个路径名组件重试搜索。如果命令以 = 开头，则以命令的路径名尝试补全。

任何 *command* 字符串都可以是通常用于生成文件名的模式。这些命令应加引号，以防止被立即扩展； 例如，命令字符串 'foo*' 用于补全以 foo 开头的命令。 在

尝试补全时，所有模式的补全都会按其定义的相反顺序进行，直到有一个匹配为止。 默认情况下，补全将按正常方式进行，即 shell 将尝试为命令行上的特定命令生成更多匹配项；可以通过在模式补全的标志中包含 -tn 来覆盖该操作。

请注意，除非设置了 COMPLETE_ALIASES 选项，否则别名会在确定命令名之前展开。 命令不得与 -C、-D 或 -T 标志结合使用。

-C

在命令字本身正在补全时控制补全。 如果没有发出 compctl -C 命令，则会补全任何可执行命令的名称（无论是路径中的还是 shell 特有的，如别名或函数）。

-D

控制未指定特殊行为的命令参数的默认补全行为。 如果没有发出 compctl -D 命令，文件名将被补全。

-T

提供补全标志，在进行其他处理之前使用，甚至在处理为特定命令定义的 compctls 之前使用。 当与扩展补全（-x 标志，见下文 扩展补全）结合使用时，这一点尤其有用。 使用该标志，你可以定义适用于所有命令的默认行为，或者改变所有命令的标准行为。 例如，如果访问用户数据库的速度太慢和/或数据库中包含的用户太多（因此 '~' 后的补全速度太慢，无法使用），可以使用

```
compctl -T -x 's[~] C[0,[^/]#]' -k friends -S/ -tn
```

以在 '~' 之后补全数组 friends 中的字符串。 C[…] 参数是必要的，这样, ~-补全，这种形式就不会在目录名完成后再尝试。

-L

以适合放入启动脚本的方式列出现有的补全行为；现有行为不会改变。 可以指定上述形式的任意组合,或指定 -M 标志（必须在 -L 标志之后），否则将列出所有已定义的补全。 其他任何标志都将被忽略。

没有参数

如果没有给出参数，compctl 会以缩写形式列出所有已定义的补全；如果给出 *options* 的列表，则会列出所有设置了这些标志集的补全（不包括扩展补全）。

如果只使用 + 标志，且紧接着使用 *command* 列表，则列表中所有命令的补全行为都将重置为默认。 换句话说，补全随后将使用 -D 标志指定的选项。

以 -M 为第一个也是唯一选项的形式定义了全局匹配规范（参见 补全匹配控制）。 所给出的匹配规范将用于每次补全尝试（仅在使用 compctl 时，新补全系统不使用），并按照定义的顺序进行尝试，直到产生至少一个匹配。例如

```
compctl -M '' 'm:{a-zA-Z}={A-Za-z}'
```

这将先尝试不包含任何全局匹配规范（空字符串）的补全，如果没有匹配结果，将尝试不区分大小写的补全。

21.4 选项标志

[-fcFBdeaRGovNAIOPZEnbjrzu/12]
[-k *array*] [-g *globstring*] [-s *substring*]
[-K *function*]
[-Q] [-P *prefix*] [-S *suffix*]
[-W *file-prefix*] [-H *num pattern*]
[-q] [-X *explanation*] [-Y *explanation*]
[-y *func-or-var*] [-l *cmd*] [-h *cmd*] [-U]
[-t *continue*] [-J *name*] [-V *name*]
[-M *match-spec*]

其余的 *options* 则指定在补全过程中要查找的命令参数类型。 可以指定这些标志的任意组合；结果是一个包含所有可能性的排序列表。 选项如下。

21.4.1 简单标志

这将产生由 shell 本身编制的补全列表：

`-f`

文件名和文件系统路径。

`-/`

只是文件系统路径。

`-c`

命令名称，包括别名、shell 函数、内置命令和保留字。

`-F`

函数名。

`-B`

内置命令名。

`-m`

外部命令名。

`-w`

保留字。

`-a`

别名。

`-R`

普通（非全局）别名。

`-G`

全局别名。

`-d`

可与 `-F`、`-B`、`-w`、`-a`、`-R` 和 `-G` 结合使用，以获取禁用的函数、内置命令、保留字或别名的名称。

`-e`

该选项（显示已启用的命令）默认有效，但可与 `-d` 结合使用；`-de` 与 `-F`、`-B`、`-w`、`-a`、`-R` 和 `-G` 结合使用时，将补全函数、内置程序、保留字或别名的名称，无论它们是否被禁用。

`-o`

shell 选项的名称（参见 选项）。

`-v`

shell 中定义的任何变量的名称。

`-N`

标量（非数组）参数的名称。

`-A`

数组名。

`-I`

整型变量名。

`-O`

只读变量名。

`-p`

shell 使用的参数名称（包括特殊参数）。

`-Z`

shell 特殊参数名。

`-E`

环境变量名。

`-n`

命名的目录。

`-b`

键绑定名。

`-j`

作业名称：作业负责人(job leader's)命令行的第一个单词。 这对 `kill` 内置程序很有用。

`-r`

运行中作业的名称。

`-z`

暂停作业的名称。

`-u`

用户名。

21.4.2 带参数的标志

这些参数由用户提供，用于确定如何组成补全列表：

`-k` *array*

取自 $*array* 元素的名称（注意，‘$’ 不会出现在命令行中）。 或者，参数 *array* 本身可以是一个用空格或逗号分隔的值的集，放在括号中，其中的任何分隔符都可以用反斜杠转义；在这种情况下，参数应加引号。 例如，

```
compctl -k "(cputime filesize datasize stacksize
             coredumpsize resident descriptors)" limit
```

`-g` *globstring*

globstring 使用文件名 globbing 扩展；应加引号以防止立即扩展。生成的文件名将作为可能的补全。 对于目录，请使用 ‘`*(/)`’ 而不是 ‘`*/`’。 `fignore` 特殊参数不适用于生成的文件。 可以使用空格分隔多个模式。(请注意，括号扩展 **不是** globbing 的一部分。 使用语法 ‘`(either|or)`’ 可以匹配其他模式）。

`-s` *substring*

substring 会被分割成单词，然后使用所有 shell 扩展机制（参见 扩展）对这些单词进行扩展。 生成的单词将作为可能的补全。 `fignore` 特殊参数不会应用于生成文件。 请注意，`-g` 对文件名的处理速度更快。

`-K` *function*

调用给定函数获取补全。 除非函数名以下划线开头，否则函数会传入两个参数：要补全的单词的前缀和后缀，换句话说，就是光标位置之前的字符和光标位置之后的字符。 可以使用 `read` 内置函数的 `-c` 和 `-l` 标志访问整个命令行。函数应将变量 `reply` 设置为包含补全信息的数组（每个元素一个补全信息）；注意 `reply` 不应设置为函数的局部变量。 在这样的函数中，可以使用 `read` 内置函数的 `-c` 和 `-l` 标志访问的命令行。 例如，

```
function whoson { reply=(`users`); }
compctl -K whoson talk
```

在 ‘`talk`’ 后只补全的登录用户。 请注意，‘whoson’ 必须返回一个数组，因此 ‘`reply=`users``’ 是不正确的。

`-H` *num pattern*

可能的补全取自最后 *num* 行历史。 只有与 *pattern* 匹配的单词才会被选中。 如果 *num* 为零或负数，则搜索整个历史记录；如果 *pattern* 为空字符串，则搜索所有单词（如 ‘*’）。 典型的用法是

```
compctl -D -f + -H 0 ''
```

如果没有匹配的文件名，则强制补全功能在历史列表中查找单词。

21.4.3 控制标志

这些选项并不直接指定要补全的名称的类型，而是对指定类型的选项进行操作：

`-Q`

这将指示 shell 在可能的补全中不任何元字符加引号。 通常情况下，补全的结果会被插入命令行中，其中的元字符会被加引号，因此会被解释为普通字符。 这适合文件名和普通字符串。 但是，为了达到特殊效果，例如从补全数组（`-k`）中插入反引号表达式，以便在执行完整行之前不对表达式进行求值，则必须使用该选项。

`-P` *prefix*

prefix 插入到已补全的字符串之前；任何已键入的开头部分都将被补全，而整个 *prefix* 将被忽略，以达到补全的目的。 例如，

```
compctl -j -P "%" kill
```

会在删除命令后插入一个 '%'，然后补全作业名称。

`-S` *suffix*

当找到补全时，*suffix* 会插入到补全的字符串之后。 在菜单补全的情况下，后缀会立即插入，但仍可以通过重复按同一按键来循环浏览补全列表。

`-W` *file-prefix*

使用目录 *file-prefix*：对于命令、文件、目录和 globbing 补全（选项 `-c`、`-f`、`-/`、`-g`），会在补全前面隐式添加文件前缀。 例如,

```
compctl -/ -W ~/Mail maildirs
```

补全目录 `~/Mail` 下任意深度的任何子目录，尽管该前缀不会出现在命令行中。 *file-prefix* 也可以是 `-k` 标志所接受的形式，即数组名称或括号中的字面列表。在这种情况下，将搜索列表中的所有目录，以查找可能的补全。

`-q`

如果与 `-S` 选项指定的后缀一起使用，则在下一个键入的字符为空白或未插入任何内容时，或在后缀仅由一个字符组成且下一个键入的字符为相同字符时，后缀将被移除；这与 `AUTO_REMOVE_SLASH` 选项的规则相同。 该选项对列表分隔符（逗号、冒号等）最有用。

`-l` *cmd*

该选项限制了被视为参数的命令行单词的范围。 如果与扩展补全模式 'p[...]'，'r[...]' 或 'R[...]'（见下文 扩展补全）之一结合使用，则范围将限制在括号中指定的参数范围内。补全操作会被执行，就好像这些东西已经作为参数传递给了使用该选

项提供的 *cmd* 命令一样。（系统会将这些内容视为 *cmd* 命令的参数，并执行相应的补全操作。）如果 *cmd* 字符串为空，则范围内的第一个单词将作为命令名称，并在范围内的第一个单词上执行命令名称补全。 例如

```
compctl -x 'r[-exec,;]' -l '' -- find
```

补全‘-exec’和后面的‘;’（如果没有该字符串，则是命令行的末尾）之间的参数，就好像它们是单独的命令行。

-h *cmd*

通常，zsh 会将加引号的字符串作为一个整体补全。有了这个选项，可以分别对字符串的不同部分进行补全。它的作用与 -l 选项类似，但补全代码只针对当前单词中用空格分隔的部分。这些部分将像给定的 *cmd* 的参数一样补全。如果 *cmd* 为空字符串，第一部分将作为命令名称补全，与 -l 一样。

-U

使用整个可能的补全列表，无论它们是否与命令行上的单词实际匹配。 目前输入的单词将被删除。 这在使用函数（由 -K 选项给出）时最有用，该函数可以检查传递给它的单词成分（或通过 read 内置函数的 -c 和 -l 标志），并使用自己的标准来决定哪些匹配。 如果没有补全，则保留原词。 由于产生的可能补全很少有有趣的共同前缀和后缀，因此如果设置了 AUTO_MENU 并使用了该标志，则会立即启动菜单补全。

-y *func-or-var*

只要请求一个列表，就会显示 *func-or-var* 提供的列表，而不是补全列表；实际要插入的补全不受影响。 可以通过两种方式提供该列表。首先，如果 *func-or-var* 以 $ 开头，则定义了一个变量；如果以左括号开头，则定义了一个字面数组，其中包含列表。 变量可能是通过调用使用 -K 选项函数设置的。 否则，它将包含一个函数的名称，该函数将被执行以创建列表。 函数将作为参数传递,列出所有匹配补全，包括前缀和后缀的完整扩展，并应将数组 reply 设置为结果。 在这两种情况下，只有在创建了完整的匹配列表后，才会检索显示列表。

请注意，返回的列表即使在长度上也不必与原始的匹配集一致，可以以标量而不是数组的形式传递。 在这种情况下，不会在输出结果的执行特殊字符格式化处理；特别是，换行符会按字面意思打印，如果换行符出现在列中，换行符将被抑制。

-X *explanation*

在当前选项集上尝试补全时,打印 *explanation*。该字符串中的‘%n’将被该解释字符串的匹配次数取代。 只有在尝试补全但没有唯一匹配时，或列出补全时，才会出现解释。解释字符串将与指定组的匹配一起列在一起，这个组使用 -X 选项（使用 -J 或 -V 选项）指定。如果在多个 -X 选项中使用了相同的解释字符串，则该字符串只出现一次（针对每一组），‘%n’显示的匹配数是每一次使用的所有匹配数的总和。在任何情况下，只有在解释字符串至少有一个匹配项时，才会显示解释字符串。

序列 %B, %b, %S, %s, %U 和 %u 指定输出属性（粗体、突出和下划线），%F, %f, %K, %k 指定前景色和背景色，%{...%} 可用来包含字面转义序列，如提示符中的转义序列。

-Y *explanation*

与 -X 相同，但 *explanation* 会首先按照双引号中字符串的常规规则进行扩展。 扩展将在调用 -K 或 -y 选项的函数后进行，从而允许它们设置变量。

-t *continue*

continue-字符串包含一个字符，用于指定下一步应使用哪一补全标志集。 它非常有用：

(i) 使用 -T，或尝试模式补全列表时，compctl 通常会在找到匹配后继续进行普通处理；可以使用 '-tn' 来抑制这种情况。

(ii) 对于用 + 分隔的备选方案列表，compctl 通常会在其中一个备选方案产生匹配时停止。 可以在 + 之前向备选方案标志中添加 '-t+'，强制它考虑下一组补全。

(iii) 在扩展补全列表（见下文）中，当 compctl 通常会持续尝试直到一组条件成功，则只使用紧随其后的标志。 如果使用 '-t-'，compctl 将在下一个 '-' 之后继续尝试扩展补全；如果使用 '-tx'，它将尝试使用默认标志补全，换句话说，就是使用 '-x' 之前的标志补全。

-J *name*

这给出了应将匹配结果归入的组的名称。组将单独列出并排序；同样，菜单补全将按照组的定义顺序提供组中的匹配结果。如果没有明确给出组名，匹配结果将存储在名为 default 的组中。第一次遇到组名时，就会创建一个具有该名称的组。之后，所有具有相同组名的匹配结果都会存储在该组中。

这对非排他性的替代性补全很有用。 例如，在

```
compctl -f -J files -t+ + -v -J variables foo
```

文件和变量都是可能的补全，因为 -t+ 会强制同时考虑 + 前后的两个备选方案集。不过，由于有 -J 选项，所有文件都会列在所有变量之前。

-V *name*

与 -J 类似，但组内的匹配结果不会在列表或菜单补全中排序。这些未排序的组与已排序的组处于不同的名称空间，因此定义为 -J files 和 -V files 的组是不同的。

-1

如果与 -V 选项一起使用，则只删除组中连续的重复内容。请注意，有此标志和无此标志的组处于不同的名称空间。

-2

如果与 -J 或 -V 选项一起使用，则会保留所有重复。同样，有此标志和无此标志的组处于不同的名称空间。

-M *match-spec*

这定义了额外的匹配控制规范，只应在为该标志所在的标志列表测试词语时使用。*match-spec* 字符串的格式在 补全匹配控制 中描述。

21.5 备选补全

compctl [-CDT] *options* + *options* [+ ...] [+] *command* ...

带 '+' 的形式指定了其他选项。补全将使用第一个 '+' 之前的选项。如果没有匹配结果，则使用 '+' 之后的标志进行补全，依此类推。如果在最后一个 '+' 之后没有标志，且到此为止仍未找到匹配项，则尝试默认补全。 如果标志列表中包含 -t 与 + 字符，即使当前列表产生了匹配，也会使用下一个标志列表。

还可以使用其他选项，将补全限制在命令行的某些部分；这被称为 '扩展补全'。

21.6 扩展补全

compctl [-CDT] *options* -x *pattern options* - ... --
[*command* ...]
compctl [-CDT] *options* [-x *pattern options* - ... --]
[+ *options* [-x ... --] ... [+]] [*command* ...]

带有 '-x' 的形式指定了所给命令的扩展补全；如图所示，它可以与使用 '+' 的可替代补全相结合。 每个 *pattern* 都会被依次检查；当发现匹配时，相应的 *options*（如上文 选项标志 所述）将用于生成可能的补全。 如果没有 *pattern* 匹配，则使用 -x 之前给出的 *options*。

请注意，每个模式应作为一个参数提供，并应加上引号，以防止 shell 扩展元字符。

一个 *pattern* 是由用逗号分隔的子模式组成的；如果这些子模式中至少有一个匹配（它们是 "或" 关系的），那么它就匹配。这些子模式又由其他子模式组成，这些子模式用空格分隔，如果所有子模式都匹配（它们是 "和" 关系），这些子模式也会匹配。 子模式的一

个元素的形式为 '*c*[...] [...]'，其中的括号对可根据需要多次重复，如果任何一组括号匹配（"或"）则匹配。 下面的例子更清楚地说明了这一点。

这些元素可以是以下任何一种:

s[*string*]...

如果命令行中的当前单词以括号中给出的字符串之一开头，则匹配该字符串。 *string* 不会被删除，也不是补全的一部分。

S[*string*]...

与 s[*string*] 类似，只不过 *string* 是补全的一部分。

p[*from*,*to*]...

如果当前单词的编号介于 *from* 和 *to* 之间，则匹配。逗号和 *to* 均为可选项；*to* 的默认值与 *from* 相同。 数字可以是负数： *-n* 指的是该行的最后第 *n* 个字。

c[*offset*,*string*]...

如果 *string* 与从当前字词位置偏移 *offset* 的字词匹配，则匹配。 通常 *offset* 为负值。

C[*offset*,*pattern*]...

与 c 类似，但使用模式匹配。

w[*index*,*string*]...

如果位置 *index* 中的单词等于相应的 *string*，则匹配。 请注意，单词计数是在任何别名扩展之后进行的。

W[*index*,*pattern*]...

与 w 类似，但使用模式匹配。

n[*index*,*string*]...

如果当前单词包含 *string* ，则匹配。 所有东西，直到并包括第 *index* 次该字符串的出现，都不会被视为补全的一部分，但其余部分会被视为补全的一部分。 *index* 可以是负数，以便从结尾开始计数：在大多数情况下，*index* 将是 1 或-1。 例如,

```
compctl -s ``users`` -x 'n[1,@]' -k hosts -- talk
```

通常会补全用户名，但如果在名称后插入 @ ，则会补全 *hosts* 数组（假定包含主机名，但必须自己创建数组）中的名称。 其他命令（如 rcp）的处理方法与此类似。

N[*index*,*string*]...

与 n 类似，但字符串将被视为一个字符类。 在 *string* 中出现的任何字符，直到（包括）第 *index* 次出现的字符，都不会被视为补全的一部分。

`m[`*min*`,`*max*`]`...

如果总字数在 *min* 和 *max* 之间（包括 *min* 和 *max* 在内），则匹配。

`r[`*str1*`,`*str2*`]`...

如果光标位于前缀为 *str1* 的单词之后，则匹配该单词。 如果命令行中在 *str1* 匹配的单词之后还有前缀为 *str2* 的单词，则只有当光标位于该单词之前时才会匹配。如果省略逗号和 *str2*，则只有当光标位于前缀为 *str1* 的单词之后时才会匹配。

`R[`*str1*`,`*str2*`]`...

与 `r` 类似，但使用模式匹配。

`q[`*str*`]`...

当前补全的单词在单引号中且 *str* 以字母 's'开头，则匹配，或者在双引号中补全且 *str* 以字母 'd' 开头，或者在反引号中补全且 *str* 以字母 'b' 开头。

21.7 举例

```
compctl -u -x 's[+] c[-1,-f],s[-f+]' \
  -g '~/Mail/*(:t)' - 's[-f],c[-1,-f]' -f -- mail
```

对这句话的解释如下:

如果当前命令是 `mail`，那么

如果 ((当前单词以 `+` 开头，且前一个单词为 `-f`）或（当前单词以 `-f+` 开头)），则补全 `~/Mail` 目录中文件的非目录部分（ ':t' glob 修饰符）；否则

如果当前单词以 `-f` 开头或前一个单词是 `-f`，则补全任何文件；否则

补全用户名。

---

22 Zsh 模块

22.1 说明

zsh 的某些可选部分以模块的形式存在，与 shell 的核心分开。 这些模块可以在构建时链接到 shell 中，或者在 shell 运行时动态链接（如果安装支持此功能）。 使用 `zmodload` 命令可在运行时链接模块，参见 Shell 内置命令。

与 zsh 发行版捆绑在一起的模块有:

`zsh/attr`

用于操作扩展属性 (xattr) 的内置程序。

`zsh/cap`

用于操作 POSIX.1e (POSIX.6) 能力（权限）集的内置程序。

`zsh/clone`

一个内置程序，可以将一个正在运行的 shell 克隆到另一个终端上。

`zsh/compctl`

用于控制补全的 `compctl` 内置命令。

`zsh/complete`

基本补全代码。

`zsh/complist`

补全列表扩展。

`zsh/computil`

该模块包含基于 shell 函数的补全系统所需的实用内置程序。

`zsh/curses`

curses 窗口命令

`zsh/datetime`

一些日期/时间命令和参数

`zsh/db/gdbm`

用于管理与 GDBM 数据库相关联的关联数组参数的内置程序。

`zsh/deltochar`

重复 EMACS 的 `zap-to-char` 的 ZLE 函数。

`zsh/example`

如何编写模块的示例。

`zsh/files`

一些内置的基本文件操作命令。

`zsh/langinfo`

本地化信息接口。

`zsh/mapfile`

通过一个特殊的关联数组访问外部文件。

`zsh/mathfunc`

用于数学计算的标准科学函数。

`zsh/nearcolor`

将颜色映射到可用调色板中最接近的颜色。

`zsh/newuser`

安排为新用户安装文件。

`zsh/parameter`

通过特殊关联数组访问内部哈希表。

`zsh/pcre`

PCRE 库的接口。

`zsh/param/private`

用于在函数上下文中管理私有范围参数的内置函数。

`zsh/regex`

POSIX regex 库的接口。

`zsh/sched`

在 shell 中提供定时执行功能的内置程序。

`zsh/net/socket`

操纵 Unix 域套接字

`zsh/stat`

`stat` 系统调用的内置命令接口。

`zsh/system`

连接各种底层系统功能的内置接口。

`zsh/net/tcp`

操作 TCP 套接字

`zsh/termcap`

termcap 数据库的接口。

`zsh/terminfo`

terminfo 数据库的接口。

`zsh/watch`

Reporting of login and logout events.

`zsh/zftp`

内置 FTP 客户端。

`zsh/zle`

Zsh 行编辑器，包含 `bindkey` 和 `vared` 内置命令。

`zsh/zleparameter`

通过参数访问 Zsh 行编辑器的内部。

`zsh/zprof`

一个允许对 shell 函数进行性能分析的模块。

`zsh/zpty`

一个内置命令，用于在伪终端中启动一个命令。

`zsh/zselect`

当文件描述符准备就绪时，阻塞并返回。

`zsh/zutil`

一些实用内置程序，例如支持通过样式进行配置的内置程序。

22.2 zsh/attr 模块

zsh/attr 模块用于操作扩展属性。 通过 -h 选项，所有命令都将在符号链接上执行，而不是在目标上执行。 该模块中的内置模块包括:

zgetattr [-h] *filename attribute* [*parameter*]

从指定的 *filename* 获取扩展属性 *attribute*。如果给出了可选参数 *parameter*，则属性将设置在该参数上，而不是打印到 。

zsetattr [-h] *filename attribute value*

将指定 *filename* 上的扩展属性 *attribute* 设置为 *value*。

zdelattr [-h] *filename attribute*

从指定的 *filename* 中移除扩展属性 *attribute*。

zlistattr [-h] *filename* [*parameter*]

列出当前在指定的 *filename* 上设置的扩展属性。如果给出了可选参数 *parameter*，属性列表将设置在该参数上，而不是打印到 stdout。

zgetattr 和 zlistattr 是动态分配内存的。 如果属性或属性列表在分配和调用以获取之间时增长，它们会返回 2。 如果出现其他错误，则返回 1。 这允许调用函数检查这种情况并重试。

22.3 zsh/cap 模块

zsh/cap 模块用于操作 POSIX.1e (POSIX.6) 能力集。 如果操作系统不支持该接口，该模块定义的内置命令将不起任何作用。 该模块中的内置命令包括：

cap [*capabilities*]

将 shell 的进程能力集更改为指定的 *capabilities*，否则显示 shell 的当前能力。

getcap *filename* ...

这是 POSIX 标准实用程序的内置实现。 它会显示每个指定 *filename* 上的能力集。

setcap *capabilities filename* ...

这是 POSIX 标准实用程序的内置实现。 它将每个指定的 *filename* 上的能力集设置为指定的 *capabilities*。

22.4 zsh/clone 模块

`zsh/clone` 模块提供了一条内置命令：

`clone` *tty*

创建当前 shell 的分叉实例，并连接到指定的 *tty*。 在新 shell 中，`PID`, `PPID` 和 `TTY` 特殊参数会相应更改。 在新 shell 中，`$!` 设置为 0，而在原 shell 中，`$!` 则设置为新 shell 的 PID。

如果成功，内置函数在两个 shell 中的返回状态都为 0，如果出错，则返回状态为非 0。

`clone` 的目标应该是一个未使用的终端，例如未使用的虚拟控制台或由下面创建的虚拟终端

```
xterm -e sh -c 'trap : INT QUIT TSTP; tty;
        while :; do sleep 100000000; done'
```

关于这个长长的 xterm 命令行，有必要解释一下：在伪终端上进行克隆时，一些其他会话（"会话"指 unix 会话组或 SID）已经拥有了该终端。因此，克隆的 zsh 无法获取伪终端作为控制 tty。这意味着两件事：

- 作业控制信号将进入 sh-started-by-xterm 进程组（这就是我们禁用 INT QUIT 和 TSTP 陷阱的原因，否则 while 循环可能会被暂停或删除）。
- 克隆的 shell 将禁用作业控制功能，并且作业控制键（control-C、control-\ 和 control-Z）将不起作用。

这不适用于克隆到 **未使用的** vc。

克隆到使用过的（和未准备好的）终端会导致两个进程同时从同一终端读取数据，输入字节会随机进入任一进程。

`clone` 主要是作为 openvt 的 shell 内置替代品。

22.5 zsh/compctl 模块

`zsh/compctl` 模块提供了两个内置命令。`compctl` 是控制 ZLE 补全的旧式、已废弃的方法。 请参阅 用 compctl 补全。 另一条内置命令 `compcall` 可用于用户自定义的补全小部件，参见 补全小部件。

22.6 zsh/complete 模块

`zsh/complete` 模块提供了多个内置命令，这些命令可用于用户自定义的补全小部件，参见 补全小部件。

22.7 zsh/complist 模块

zsh/complist 模块为补全列表提供了三种扩展功能：在此类列表中高亮显示匹配项的功能、滚动浏览长列表的功能以及不同风格的菜单补全功能。

22.7.1 彩色补全列表

只要设置了 ZLS_COLORS 或 ZLS_COLOURS 之一，并且 zsh/complist 模块已加载或链接到 shell 中，补全列表就会进行彩显。 但需要注意的是，如果 complist 没有被链接到 shell 中，则不会被自动加载：在使用动态加载的系统中，需要使用 'zmodload zsh/complist' 。

参数 ZLS_COLORS 和 ZLS_COLOURS 描述了如何高亮匹配。 要打开高亮效果，只需输入空值即可，在这种情况下，将使用下面给出的所有默认值。 这些参数值的格式与 GNU 版本的 ls 命令相同：一个以冒号分隔的列表，其形式为 '*name*=*value*'。 *name* 可以是以下字符串之一，其中大部分字符串指定了 *value* 将被使用的文件类型。 这些字符串及其默认值是:

no 0

用于普通文本（即显示匹配文件以外的内容时）

fi 0

用于常规文件

di 32

用于目录

ln 36

用于符号链接。 如果具有特殊值 target，则符号链接将被解引用，并使用目标文件来确定显示格式。

pi 31

用于命名管道 (FIFOs)

so 33

用于套接字

bd 44;37

用于块设备

`cd` 44;37

用于字符设备

`or` *none*

用于指向不存在文件的符号链接（默认值为 `ln` 中定义的值）

`mi` *none*

用于不存在的文件（默认值为 `fi` 定义的值）；该代码目前未被使用

`su` 37;41

用于设置了 setuid 位的文件

`sg` 30;43

为设置了 setgid 位的文件

`tw` 30;42

用于设置了粘着位的可供所有用户写入的目录

`ow` 34;43

用于没有设置粘着位的可供所有用户写入的目录

`sa` *none*

用于带有相关后缀别名的文件；这只在特定后缀后进行测试，如下所述

`st` 37;44

用于已设置粘着位但不是全局可写的目录

`ex` 35

用于可执行文件

`lc` \e[

用于左侧代码（见下文）

`rc` m

用于右侧代码

`tc` 0

用于在设置 `LIST_TYPES` 选项时在文件名后打印的表示文件类型的字符

`sp 0`

用于匹配后打印的空格，以对齐下一列

`ec` *none*

用于结束代码

除这些字符串外，*name* 也可以是星号 ('*')，后跟任何字符串。为此类字符串给出的 *value* 将用于名称以该字符串结尾的所有文件。 *name* 也可以是等号 ('=')，后面跟一个模式；在计算模式时，将启用 EXTENDED_GLOB 选项。 为该模式提供的 *value* 将用于所有显示字符串与该模式匹配的匹配项（不只是文件名）。 带前导等号的形式的定义优先于为文件类型定义的值，而文件类型定义的值又优先于带前导星号的形式（文件扩展名）。

前导等于形式还允许对显示字符串的不同部分进行不同着色。 为此，模式必须使用 '(#b)' globbing 标志和围绕字符串颜色不同的部分的括号对。 在这种情况下，*value* 可以包含多个颜色代码，并用等号分隔。 第一个代码将用于所有未指定明确代码的部分，后面的代码将用于与括号中子模式匹配的部分。 例如，'=(#b)(?)*(?)=0=3=7' 将用于所有匹配(至少两个字符长度的)，第一个字符使用 '3' 代码，最后一个字符使用 '7' 代码，其余字符使用 '0' 代码。

name 的三种形式前面都可以加上用括号括起来的模式。 如果给出了该模式，*value* 将仅用于名称与括号中给出的模式匹配的组中的匹配。 例如，'(g*)m*=43' 将使用颜色代码 '43' 高亮显示名称以 'g' 开头的组中所有以 'm' 开头的匹配项。 如果使用 'lc','rc',和 'ec' 代码，组模式将被忽略。

还需注意的是，所有模式都会按其在参数值中出现的顺序进行尝试，直到第一个匹配的模式被使用为止。 模式可以与补全、描述（可能追加空格作为填充）或由补全和描述组成的行匹配。 为保持着色的一致性，可能需要使用多个模式或带有反向引用的模式。

在打印匹配结果时，代码会打印 `lc` 的值、文件类型的值或带有 '*' 的最后一个匹配规范、`rc` 的值、要显示的匹配结果本身的字符串，然后是 `ec` 的值（如果已定义）或 `lc`、`no` 和 `rc` 的值（如果未定义 `ec`）。

默认值符合 ISO 6429（ANSI）标准，可用于 vt100 兼容终端（如 `xterms`）。 在单色终端上，默认值不会产生明显影响。 贡献中的 `colors` 函数可用于获取包含 ANSI 终端代码的关联数组（参见 其它函数）。 例如，加载 `colors` 后，可以使用 '`$color[red]`' 为前景色获取红色的代码，使用 '`$color[bg-green]`' 为背景色获取绿色的代码。

如果使用的是由 compinit 调用的补全系统，则不应直接设置这些参数，因为系统会自行控制它们。 相反，应使用 `list-colors` 样式（参见 补全系统配置）。

22.7.2 在补全列表中滚动

要启用滚动浏览补全列表，必须设置 `LISTPROMPT` 参数。 其值将用作提示符；如果是空字符串，则使用默认提示符。 该值可包含 '%x' 形式的转义字符。 它支持转义符 '%B', '%b', '%S', '%s', '%U', '%u', '%F', '%f', '%K', '%k' 和 '%{...%}' 也用于 shell 提示符以及三对附加序

列：‘%l’ 或 ‘%L’ 会被最后显示的行数和总行数替换，形式为 ‘*number*/*total*’；‘%m’ 或 ‘%M’ 会被最后显示的匹配行数和总匹配行数替换；‘%p’ 或 ‘%P’ 分别替换为 ‘Top’、‘Bottom’ 或显示的第一行的位置（占总行数的百分比）。 在上述每种情况下，大写字母形式将被替换为固定宽度的字符串，并在右侧填充空格，而小写字母形式则不填充空格。

如果设置了参数 LISTPROMPT，补全代码将不会询问是否显示列表。 相反，它会立即开始显示列表，在显示第一屏后停止，在底部显示提示符，在临时切换到 listscroll 键映射后等待按键。 在滚动列表时，zle 的某些函数具有特殊意义：

send-break

停止列表，丢弃已按下的键

accept-line, down-history, down-line-or-history
down-line-or-search, vi-down-line-or-history

向前滚动一行

complete-word, menu-complete, expand-or-complete
expand-or-complete-prefix, menu-complete-or-expand

向前滚动一屏

accept-search

停止列表，但不采取其他行动

其他每个字符都会停止列表，并立即像往常一样处理按键。 任何未在 listscroll 键盘映射中绑定或绑定到 undefined-key 的按键都会在当前选定的键盘映射中查找。

至于 ZLS_COLORS 和 ZLS_COLOURS 参数，在使用基于 shell 函数的补全系统时，不应直接设置 LISTPROMPT。 相反，应使用 list-prompt 样式。

22.7.3 菜单选择

zsh/complist 模块还提供了另一种从列表中选择匹配项的方式，称为菜单选择，如果 shell 设置为在显示补全列表后返回最后（近）一个提示符（参见 选项 中的 ALWAYS_LAST_PROMPT 选项），则可以使用这种方式。

本模块定义的小部件 menu-select 可直接调用菜单选择。 这是一个标准的 ZLE 小部件，可以按照 Zsh 行编辑器中描述的常规方式与按键绑定。

或者，也可以将参数 MENUSELECT 设置为一个整数，它给出了在菜单选择自动开启之前必须存在的最小匹配数。 第二种方法要求菜单补全必须启动，可以直接从 menu-complete 等小部件启动，也可以由于设置了 MENU_COMPLETE 或 AUTO_MENU 选项之一而启动。 如果设置了 MENUSELECT，但其值为 0、1 或空，则菜单选择将始终在模糊菜单补全期间启动。

在使用基于 shell 函数的补全系统时，不应使用 MENUSELECT 参数（如上文所述的 ZLS_COLORS 和 ZLS_COLOURS 参数）。 相反，应该与 select=... 关键字一起使用 menu 样式。

菜单选择开始后，将列出匹配结果。如果屏幕上的匹配项较多，则只显示第一屏。 可以从列表中选择要插入命令行的匹配项。 在列表中，将使用 ZLS_COLORS 或 ZLS_COLOURS 参数中的 ma 值突出显示一个匹配项。 默认值为 '7'，这将强制在兼容 vt100 的终端上使用突出显示模式突出显示选定的匹配。 如果既未设置 ZLS_COLORS 也未设置 ZLS_COLOURS，则会使用与提示符中的 '%S' 转义相同的终端控制序列。

如果屏幕上的匹配项较多，且参数 MENUPROMPT 已设置，则其值将显示在匹配项下方。它支持与 LISTPROMPT 相同的转义序列，但显示的匹配项或行的编号将是放置标记的匹配项或行的编号。 如果其值为空字符串，将使用默认提示符。

MENUSCROLL 参数可用于指定列表的滚动方式。 如果未设置该参数，将逐行滚动；如果设置为'0'（零），列表将滚动屏幕行数的一半。 如果参数值为正数，则滚动的行数为正数；如果参数值为负数，则滚动的行数为屏幕行数减去这个值（绝对值）后的行数。

至于 ZLS_COLORS、ZLS_COLOURS 和 LISTPROMPT 参数，在使用基于 shell 函数的补全系统时，不应直接设置 MENUPROMPT 或 MENUSCROLL。 相反，应使用 select-prompt 和 select-scroll 样式。

补全代码有时会决定不在列表中显示所有匹配项。 这些隐藏的匹配项要么是添加匹配项的补全函数明确要求不在列表中显示的匹配项（使用 compadd 内置命令的 -n 选项），要么是与列表中已存在的字符串重复的匹配项（因为它们仅在前缀或后缀等未显示的内容上存在差异）。 不过，在用于菜单选择的列表中，即使是这些匹配也会显示出来，因此可以选择它们。 为了突出显示这类匹配，ZLS_COLORS 和 ZLS_COLOURS 参数中的 hi 和 du 功能分别支持第一类和第二类隐藏匹配。

通过使用 zle 移动函数移动标记来选择匹配项。 当屏幕上不能同时显示所有匹配时，列表会在越过顶线或底线时上下滚动。 下列 zle 函数在菜单选择时具有特殊意义。 请注意，以下函数在菜单选择映射中始终执行相同的任务，不能被用户定义的小部件取代，也不能扩展函数集：

accept-line, accept-search

接受当前匹配并离开菜单选择（但不接受命令行）

send-break

退出菜单选择并恢复之前命令行的内容

redisplay, clear-screen

在不离开菜单选择的情况下执行其正常函数

accept-and-hold, accept-and-menu-complete

接受当前插入的匹配项，并继续选择，允许选择下一个匹配项插入该行

accept-and-infer-next-history

接受当前的匹配结果，然后再次尝试用菜单选择来补全；在文件的情况下，这允许选择一个目录并立即尝试补全其中的文件；如果没有匹配结果，则会显示一条信息，可以使用 undo 返回上一级的补全，其他所有按键都会离开菜单选择（包括菜单选择期间其他特殊的 zle 功能）。

undo

删除在菜单选择过程中由前面三个函数之一插入的匹配项。

down-history, down-line-or-history
vi-down-line-or-history, down-line-or-search

将标记向下移动一行

up-history, up-line-or-history
vi-up-line-or-history, up-line-or-search

将标记向上移动一行

forward-char, vi-forward-char

将标记向右移动一列

backward-char, vi-backward-char

将标记向左移动一列

forward-word, vi-forward-word
vi-forward-word-end, emacs-forward-word

将标记向下移动一屏

backward-word, vi-backward-word, emacs-backward-word

将标记向上移动一屏

vi-forward-blank-word, vi-forward-blank-word-end

将标记移至下一组匹配的第一行

vi-backward-blank-word

将标记移至前一组匹配的最后一行

beginning-of-history

将标记移至第一行

end-of-history

将标记移至最后一行

beginning-of-buffer-or-history, beginning-of-line
beginning-of-line-hist, vi-beginning-of-line

将标记移到最左边一列

end-of-buffer-or-history, end-of-line
end-of-line-hist, vi-end-of-line

将标记移到最右边一列

complete-word, menu-complete, expand-or-complete
expand-or-complete-prefix, menu-expand-or-complete

将标记移动到下一个匹配

reverse-menu-complete

将标记移至上一个匹配

vi-insert

这将在正常模式和交互模式之间切换；在交互模式下，与 self-insert 和 self-insert-unmeta 绑定的按键会像正常编辑模式一样插入命令行，但不会离开菜单选择；每个字符补全后都会再次尝试，列表会更改为只包含新的匹配字符；补全部件会将最长的无歧义字符串插入命令行，而 undo 和 backward-delete-char 则会返回到上一组匹配字符串

history-incremental-search-forward
history-incremental-search-backward

这会开始在显示的补全列表中进行增量搜索；在这种模式下，accept-line 只会离开增量搜索，回到正常的菜单选择模式

所有移动函数都会在边缘回绕；任何其他未列出的 zle 函数都会离开菜单选择并执行该函数。 通过使用前面带有‘.’的小部件形式，可以使上述列表中的小部件具有相同的功能。 例如，小部件‘.accept-line’具有离开菜单选择并接受整个命令行的效果。

在选择过程中，widget 会使用键映射 menuselect。 任何未在此键映射中定义或绑定到 undefined-key 的按键都会在当前选择的键映射中查找。 这样做是为了确保在选择过程中使用的最重要按键（即光标键、回车键和 TAB 键）具有合理的默认值。 不过，可以使用 bindkey 内置命令直接修改 menuselect 键映射中的按键（参见 zsh/zle 模块）。 例如，要使返回键离开菜单选择而不接受当前选择的匹配，可以调用

```
bindkey -M menuselect '^M' send-break
```

加载 zsh/complist 模块后。

22.8 zsh/computil 模块

zsh/computil 模块为基于 shell 函数的补全系统（参见 补全系统 ）中的某些补全函数添加了几条内置命令。 除了 compquote 之外，这些内置命令都非常专业，因此在编写自己的补全函数时并不十分有趣。 总之，这些内置命令包括

comparguments

> _arguments 函数使用它来进行参数和命令行解析。 与 compdescribe 一样，它也有一个选项 -i 来完成解析并初始化一些内部状态，还有各种选项来访问状态信息，以决定应补全哪些内容。

compdescribe

> _describe 函数使用它来构建匹配的显示，并获取要添加为匹配项的字符串及其选项。 第一次调用时，应将 -i 或 -I 选项之一作为第一个参数提供。 在第一种情况下，将生成不带说明的显示字符串；在第二种情况下，必须将用于分隔匹配字符串和说明的字符串作为第二个参数，然后显示说明（如果有的话）。 所有其他参数与 _describe 本身的定义参数相同。
>
> 使用 -i 或 -I 选项调用 compdescribe 后，可以使用 -g 选项和四个参数的名称作为参数反复调用。 这将逐步处理不同的匹配集，并在第一个标量中存储 compstate[list] 的值，在第二个数组中存储 compadd 的选项，在第三个数组中存储匹配结果，在第四个数组中存储要在补全列表中显示的字符串。 然后可以直接将这些数组交给 compadd，以便将匹配结果注册到补全代码中。

compfiles

> 用于 _path_files 函数,以优化复杂递归文件名的生成（globbing）。 它会做三件事。 使用 -p 和 -P 选项，它会构建要使用的 glob 模式，包括已经处理过的路径，并尝试根据行中的前缀和后缀以及当前使用的匹配规范优化模式。 -i 选项会对 ignore-parents 样式进行目录测试，而 -r 选项则会测试某些匹配项的组件是否等于行中的字符串，如果为真，则会移除所有其他匹配项。

compgroups

> 由 _tags 函数使用，用于实现 group-order 样式的内部。 它只接收作为补全分组名称的参数，并为其创建分组（所有六种类型：排序和不排序，均不去除重复、去除全部重复和去除连续重复）。

compquote [-p] *names* ...

编写补全函数时可能会有一些原因，必须使用 -Q 选项将匹配结果添加到 compadd 并自行执行引用（加引号）。 与其解释 compstate 特殊关联数组的 all_quotes 键的第一个字符，并使用 q 标志进行参数扩展，不如使用这条内置命令。 参数是标量或数组参数的名称，这些参数的值根据需要为最内层的加引号。 如果给定了 -p 选项，会加上引号，就像参数值前有一些前缀，因此前导等号不会加引号。

如果出现错误，返回状态为非零，否则为零。

comptags
comptry

它们实现了标记机制的内部功能。

compvalues

与 comparguments 类似，但用于 _values 函数。

22.9 zsh/curses 模块

zsh/curses 模块提供了一条内置命令和各种参数。

22.9.1 内置程序

zcurses init
zcurses end
zcurses addwin *targetwin nlines ncols begin_y begin_x* [*parentwin*]
zcurses delwin *targetwin*
zcurses refresh [*targetwin ...*]
zcurses touch *targetwin ...*
zcurses move *targetwin new_y new_x*
zcurses clear *targetwin* [redraw | eol | bot]
zcurses position *targetwin array*
zcurses char *targetwin character*
zcurses string *targetwin string*
zcurses border *targetwin border*
zcurses attr *targetwin* [[+|-]*attribute* | *fg_col*/*bg_col*] [...]
zcurses bg *targetwin* [[+|-]*attribute* | *fg_col*/*bg_col* | @*char*] [...]
zcurses scroll *targetwin* [on | off | [+|-]*lines*]
zcurses input *targetwin* [*param* [*kparam* [*mparam*]]]
zcurses mouse [delay *num* | [+|-]motion]
zcurses timeout *targetwin intval*
zcurses querychar *targetwin* [*param*]
zcurses resize *height width* [endwin | nosave | endwin_nosave]

操作 curses 窗口。 所有使用该命令的操作都应在括号中加上 ‘zcurses init’，以初始化 curses 的使用，以及 ‘zcurses end’ ，以结束 curses 的使用；省略 ‘zcurses end’ 可能会导致终端处于不需要的状态。

子命令 addwin 将创建一个具有 *nlines* 行和 *ncols* 列的窗口。 窗口的左上角将位于屏幕的 *begin_y* 行和 *begin_x* 列。 *targetwin* 是一个字符串，指向当前未分配的窗口名称。 需要特别注意的是，curses 约定垂直数值显示在水平数值之前。

如果 addwin 的最终参数是一个现有窗口，那么新窗口将作为 *parentwin* 的子窗口创建。 这与普通新建窗口的不同之处在于，窗口内容的内存与父窗口的内存共享。 子窗口必须先于父窗口删除。 请注意，子窗口的坐标是相对于屏幕而非父窗口的，这一点与其他窗口相同。

使用子命令 delwin 删除用 addwin 创建的窗口。 请注意，end **不会** 隐式删除窗口，而且 delwin 不会擦除窗口的屏幕图像。

与完整可见屏幕相对应的窗口称为 stdscr；它始终存在于 ‘zcurses init’ 之后，且无法通过 delwin 删除。

子命令 refresh 将刷新窗口 *targetwin*；这对于在屏幕上显示任何待处理的更改（例如使用 char 准备输出的字符）是必要的。 不带参数的 refresh 会清空并重新绘制屏幕。 如果给定了多个窗口，屏幕会在最后更新一次。

子命令 touch 会将列出的 *targetwin* 标记为已更改。 如果删除了位于另一个窗口（可能是 stdscr）前面的窗口，则在 refresh 操作窗口前必须执行此操作。

子命令 move 会将 *targetwin* 中的光标位置移动到新坐标 *new_y* 和 *new_x* 上。 需要注意的是，子命令 string（但不包括子命令 char）会将光标位置移动到添加的字符上。

子命令 clear 会清除 *targetwin* 中的内容。 可以指定三个选项中的一个（最多一个）。 如果使用选项 redraw，则 *targetwin* 的下一次 refresh 将导致屏幕被清除并重新绘制。 如果使用选项 eol，*targetwin* 只清除到当前光标行的末尾。 如果使用选项 bot，*targetwin* 会被清除到窗口的末尾，即到光标右侧和下方的所有内容都会被清除。

子命令 position 会将与 *targetwin* 相关的各种位置写入名为 *array* 的数组中。 这些位置依次是:

-

    光标相对于 *targetwin* 左上角的 y 和 x 坐标

-

    屏幕上 *targetwin* 左上角的 y 和 x 坐标

-

targetwin 在 y 和 x 维度上的大小。

字符和字符串的输出分别由 char 和 string 实现。

要在 *targetwin* 窗口周围绘制边框，请使用 border。 请注意，边框随后不会被特殊处理：换句话说，边框只是在窗口边缘输出的一组字符。 因此，它可以被覆盖，也可以从窗口滚动出去，等等。

子命令 attr 将为任何连续输出的字符设置 *targetwin* 的属性或前景/背景颜色对。行中给出的每个 *attribute* 都可以用 + 作为前缀来设置或用 - 作前缀来取消设置；如果没有前缀，则默认为 +。 支持的属性有 blink、bold、dim、reverse、standout 和 underline。

每个 *fg_col*/*bg_col* 属性（可理解为 '*fg_col* on *bg_col*'）都设置了字符输出的前景色和背景色。 有时也可以使用颜色 default（特别是当库为 ncurses 时），指定终端启动时的前景色或背景色。 颜色对 default/default 始终可用。要使用多于 8 种指定颜色（红色、绿色等），请构建 *fg_col*/*bg_col* 对，其中 *fg_col* 和 *bg_col* 均为十进制整数，例如 128/200。 如果终端支持 256 种颜色，则最大颜色值为 254。

bg 会覆盖窗口中所有字符的颜色和其他属性。 它通常用于设置初始背景，但在调用时会覆盖任何字符的属性。 除了 attr 允许的参数外，参数 @*char* 还可以指定要在窗口空白区域显示的字符。 由于curses的限制，这个字符不能是多字节字符（建议只使用ASCII字符）。 由于指定的属性集会覆盖现有的背景，因此在参数中关闭属性并无用处，但不会导致错误。

子命令 scroll 可以与 on 或 off 一起使用，当光标因输入或输出而移动到窗口下方时，可以启用或禁用窗口滚动。 它还可以与一个正整数或负整数一起使用，在不改变当前光标位置的情况下向上或向下滚动指定行数的窗口（因此，光标相对于窗口的移动方向看起来是相反的）。 在第二种情况下，如果滚动是 off，则会暂时转为 on，以允许窗口滚动。

子命令 input 从窗口读取一个字符，但不回传。 如果提供了 *param*，则字符会被赋值给参数 *param*，否则会赋值给参数 REPLY。

如果同时提供 *param* 和 *kparam*，按键将以 'keypad' 模式读取。 在这种模式下，特殊按键（如功能键和方向键）会返回参数 *kparam* 中的按键名称。 按键名称是 curses.h 或 ncurses.h 中定义的宏，去掉了前缀 'KEY_'；另请参阅下面参数 zcurses_keycodes 的描述。 其他键值会像之前一样在 *param* 中设置一个值。成功返回时，只有 *param* 或 *kparam* 中的一个包含非空字符串，另一个将被设置为空字符串。

如果同时提供 *mparam*，input 将尝试处理鼠标输入。 只有在使用 ncurses 库时才能使用；可以通过不带参数检查 'zcurses mouse' 的退出状态来检测鼠标处理。如果鼠标按键被点击（或双击或三击，或按下或释放时有可配置的延迟），*kparam* 将被设置为字符串 MOUSE，*mparam* 将被设置为由以下元素组成的数组：

-

  用于区分不同输入设备的标识符；这种标识符很少有用。

-

  鼠标点击时相对于全屏的 x、y 和 z 坐标，这三个元素按顺序排列（即 y 坐标不同寻常地位于 x 坐标之后）。 z 坐标只适用于少数特殊的输入设备，否则将设为零。

-

  作为单独项目发生的任何事件；通常只有一个。 一个事件包括 PRESSED, RELEASED, CLICKED, DOUBLE_CLICKED 或 TRIPLE_CLICKED ，紧接着（在同一元素中）是按钮的编号。

-

  如果按下了 shift 键，则字符串 SHIFT。

-

  如果按下的是 control 键，则字符串 CTRL。

-

  如果按下了 alt 键，则字符串 ALT。

并非所有鼠标事件都会传递到终端窗口；大多数终端模拟器都会自行处理某些鼠标事件。 请注意，ncurses 手册暗示，在有鼠标处理和没有鼠标处理的情况下使用输入时，都可能导致鼠标光标出现和消失。

子命令 mouse 可用于配置鼠标的使用。 没有窗口参数，鼠标选项是全局的。 如果可以处理鼠标，不带参数的‘zcurses mouse’返回状态 0，否则返回状态 1。 否则，可能的参数（可以在同一命令行中组合）如下。 delay *num* 以毫秒为单位设置鼠标按下和松开之间的最大延迟时间，该延迟时间将被视为一次点击；值 0 将禁用点击解析，默认值为六分之一秒。 motion后接一个可选的 ‘+’（默认）或 - ，用于打开或关闭除始终报告的点击、按下和释放之外的鼠标动作报告。 不过，目前似乎还没有实现对鼠标动作的报告。

子命令 timeout 指定了从 *targetwin* 输入的超时值。 如果 *intval* 为负值，‘zcurses input’将无限期等待输入字符；这是默认情况。 如果 *intval* 为零，‘zcurses input’立即返回；如果有预输入数据，则返回该数据；否则不进行输入，并返回状态码1。 如果 *intval* 为正值，‘zcurses input’会等待 *intval* 毫秒，如果等待时间结束时没有输入，则返回状态 1。

子命令 querychar 用于查询当前光标位置的字符。 如果提供了返回值，则将其存储在名为 *param* 的数组中，否则将其存储在 reply 数组中。 第一个值是字符（如

果系统支持多字节字符，则可以是多字节字符）；第二个值是颜色对，采用通常的 *fg_col*/*bg_col* 符号，如果不支持颜色，则为 0。 除颜色外，任何适用于字符的属性（如子命令 attr 所设置）都会作为附加元素出现。

子命令 resize 会将 stdscr 和所有窗口的大小调整到给定尺寸（超出新尺寸的窗口会被缩小）。底层的 curses 扩展（resize_term 调用）可能不可用。为了验证，可以在 *height* 和 *width* 中使用零。如果子命令的结果为 0，则 resize_term 可用（否则为 2）。测试表明，调整大小通常可以通过调用 zcurses end 和 zcurses refresh 来完成。提供 resize 子命令是为了实现多功能性。在检查了多系统配置后，仍需要 zcurses end 和 zcurses refresh ,在调整大小后获得正确的终端状态。要通过 resize 调用它们，请使用 *endwin* 参数。 使用 *nosave* 参数将导致 zcurses 内部不保存新的终端状态。这也是为多功能性而提供的，一般情况下不需要。

22.9.2 参数

ZCURSES_COLORS

只读整数。 终端支持的最大颜色数。 该值由 curses 库初始化，在首次运行 zcurses init 之前不可用。

ZCURSES_COLOR_PAIRS

只读整数。 'zcurses attr' 命令中可定义的颜色对 *fg_col*/*bg_col* 的最大数量；注意该限制适用于所有已使用过的颜色对，无论其当前是否处于活动状态。 该值由 curses 库初始化，在首次运行 zcurses init 之前不可用。

zcurses_attrs

只读数组。 zsh/curses 支持的属性；模块加载后立即可用。

zcurses_colors

只读数组。 zsh/curses 支持的颜色；模块加载后立即可用。

zcurses_keycodes

只读数组。 在 'zcurses input' 的第二个参数中,可能返回一系列值，这些值按照 curses内部定义的顺序排列。 并非所有功能键都会列出，只有 F0；curses 为 F0 至 F63 保留了空间。

zcurses_windows

只读数组。 当前窗口列表，即所有使用'zcurses addwin' 创建、未使用 'zcurses delwin' 删除的窗口。

22.10 zsh/datetime 模块

zsh/datetime 模块提供了一条内置命令：

`strftime` [`-s` *scalar* | `-n`] *format* [*epochtime* [*nanoseconds*]]
`strftime -r` [`-q`] [`-s` *scalar* | `-n`] *format timestring*

以指定的 *format* 格式输出日期。 如果没有 *epochtime*，则使用当前的系统日期/时间；可以选择使用 *epochtime* 指定从纪元开始的秒数，还可以使用 *nanoseconds* 指定秒后的纳秒数（否则该数字被假定为 0）。 详情请参阅 strftime(3)。 还可以使用 提示符扩展 中描述的 zsh 扩展。

`-n`

抑制在格式化字符串后打印换行符。

`-q`

静音运行；禁止打印以下所有错误信息。 对于无效的 *epochtime* 值，错误信息始终会被打印。

`-r`

通过选项 `-r`（反向），使用 *format* 解析输入字符串 *timestring* 并输出自纪元起的秒数。 解析由系统函数 `strptime` 实现；请参阅 strptime(3)。 这意味着无法使用 zsh 格式扩展，但反向查找时不需要这些扩展。

在 `strftime` 的大多数实现中，*timestring* 中的任何时区都会被忽略，而使用 `TZ` 环境变量声明的本地时区；其他参数如果不存在，则设置为 0。

如果 *timestring* 与 *format* 不匹配，命令将返回状态 1 并打印错误信息。 如果 *timestring* 与 *format* 匹配，但未使用 *timestring* 中的所有字符，则转换成功，但也会打印错误信息。

如果系统函数 `strptime` 或 `mktime` 不可用，则返回状态 2 并打印错误信息。

`-s` *scalar*

将日期字符串（或以秒为单位的纪元(epoch)时间，如果给定了 `-r`）赋值给 *scalar*，而不是打印出来。

请注意，根据系统声明的整型时间类型，`strftime` 可能会在纪元时间大于 2147483647（相当于 2038-01-19 03:14:07 +0000 ）时产生错误结果。

zsh/datetime 模块提供了多个参数；所有参数都是只读参数：

`EPOCHREALTIME`

一个浮点数值，代表自纪元(epoch)开始的秒数。 如果 clock_gettime 调用可用，则名义精度为纳秒，否则为微秒，但在实际应用中，双精度浮点范围和 shell 调度延迟可能会产生重大影响。

EPOCHSECONDS

一个整数值，表示自 epoch 开始的秒数。

epochtime

一个数组值，第一个元素包含自纪元(epoch)以来的秒数，第二个元素包含自纪元以来的剩余(remainder)时间（纳秒）。 为确保两个元素的一致性，在使用数值之前，应将数组复制或以其他方式引用为一个单一的替代值。 可以使用以下成语：

```
for secs nsecs in $epochtime; do
  ...
done
```

22.11 zsh/db/gdbm 模块

zsh/db/gdbm 模块用于创建与数据库文件接口的 "绑定 "关联数组。 如果 GDBM 接口不可用，本模块定义的内置程序将报错。 本模块也是用于创建其他数据库接口的原型，因此 ztie 内置函数将来可能会移到更通用的模块。

该模块中的内置程序包括:

ztie -d db/gdbm -f *filename* [-r] *arrayname*

打开由 *filename* 标识的 GDBM 数据库，如果成功，则创建与文件关联的关联数组 *arrayname*。 要创建本地绑定数组，必须先声明参数，因此类似下面的命令将在函数作用域内执行：

```
local -A sampledb
ztie -d db/gdbm -f sample.gdbm sampledb
```

-r 选项打开的数据库文件只能读取，并创建一个带有只读属性的参数。 如果不使用该选项，在用户没有写权限的文件上使用 'ztie' 将导致错误。 如果数据库是可写的，则会同步打开，因此 *arrayname* 中更改的字段会立即写入 *filename*。

打开文件后,对文件模式 *filename* 的更改,不会改变 *arrayname* 的状态，但是 'typeset -r *arrayname*' 如预期一样工作。

zuntie [-u] *arrayname* ...

关闭与每个 *arrayname* 关联的 GDBM 数据库，然后取消设置参数。 -u 选项会强制取消设置用 'ztie -r' 设为只读的参数。

如果参数被显式取消设置(unset)或其本地作用域（函数）结束，则会自动发生。 请注意，只读参数不能显式取消设置，因此取消设置使用 ‘ztie -r’ 创建的全局参数的唯一方法是使用 ‘zuntie -u’。

zgdbmpath *parametername*

将分配给 *parametername* 的数据库文件路径放入 REPLY 标量。

zgdbm_tied

包含所有绑定参数名称的数组。

与 GDBM 绑定的关联数组的字段既不会被缓存，也不会以其他方式存储在内存中，而是在每次引用时从数据库中读取或写入。 因此，例如，同一数据库文件的第二个写入者可能会更改只读数组中的值。

22.12 zsh/deltochar 模块

zsh/deltochar 模块提供了两个 ZLE 函数：

delete-to-char

从键盘读取一个字符，然后从光标位置开始删除，直至并包括该字符的下一个（或，带有重复次数 *n* 时，第 *n* 个）实例。 负重复次数表示向后(backword)删除。

zap-to-char

其行为与 delete-to-char类似，只是不会删除最后出现的字符本身。

22.13 zsh/example 模块

zsh/example 模块提供了一条内置命令：

example [-flags] [*args* ...]

显示调用时的标志和参数。

该模块的目的是作为如何编写模块的范例。

22.14 zsh/files 模块

zsh/files 模块提供了一些用于文件操作的内置常用命令；这些命令在许多正常情况下可能并不需要，但在资源有限的紧急恢复情况下可能很有用。 这些命令并未实现相关标准委员会目前要求的所有功能。

对于所有命令，以 zf_ 开头的变量也是可用的，并会自动加载。 使用 zmodload 的特性功能可以让你只加载你想要的名称。 请注意，使用以下命令可以只加载具有 zsh 特定名称的内置程序：

```
zmodload -m -F zsh/files b:zf_\*
```

默认加载的命令有:

chgrp [-hRs] *group filename ...*

更改指定文件的组。 这等同于 chown，其 *user-spec* 参数为 ':*group*'。

chmod [-Rs] *mode filename ...*

更改指定文件的模式。

指定的 *mode* 必须是八进制。

-R 选项会使 chmod 向下递归到目录，在改变目录本身的模式后，再改变目录中所有文件的模式。

-s 选项是 zsh 对 chmod 功能的扩展。 它启用了偏执行为，旨在避免因 chmod 被欺骗而影响其他文件的安全问题。 它会拒绝跟踪符号链接，因此（例如）如果 /tmp/foo 恰好是 /etc 的链接，"chmod 600 /tmp/foo/passwd" 就不会意外 chmod /etc/passwd。 它还会检查离开目录后的位置，这样深目录树的递归 chmod 就不会因为目录向上移动而对 /usr 进行递归 chmod。

chown [-hRs] *user-spec filename ...*

更改指定文件的所有权和群组。

user-spec 可以有四种形式：

user

将所有者更改为 *user*；不更改组

user::

将所有者更改为 *user*；不更改组

user:

将所有者更改为 *user*；将组更改为 *user* 的主组

user:*group*

将所有者更改为 *user*；将组更改为 *group*

:*group*

不要更改所有者；将组更改为 *group*

在每种情况下，‘:’都可能是‘.’。 规则是，如果有一个‘:’，那么分隔符就是‘:’，否则，如果有一个‘.’，那么分隔符就是‘.’，否则就没有分隔符。

每个 *user* 和 *group* 都可以是用户名（或组名，视情况而定）或十进制用户 ID（组 ID）。 如果存在全数字用户名（或组名），则优先解释为名称。

如果目标是符号链接，-h选项会导致 chown 设置链接的所有权，而不是目标。

-R 选项会使 chown 向下递归到目录中，在更改目录本身的所有权后，再更改目录中所有文件的所有权。

-s 选项是 zsh 对 chown 功能的扩展。 它启用了偏执行为，旨在避免因 chown 被欺骗而影响其他文件的安全问题。 它会拒绝跟踪符号链接，因此（例如），如果 /tmp/foo 恰好是 /etc 的链接，“chown luser /tmp/foo/passwd” 就不会意外地 chown /etc/passwd。 它还会检查离开目录后的位置，这样在对深目录树进行递归 chown 时，就不会因为目录被上移而对 /usr 进行递归 chown。

ln [-dfhins] *filename dest*
ln [-dfhins] *filename ... dir*

创建硬链接（或使用 -s 创建符号链接）。 第一种形式是创建指定 *dest* 目标，作为指向指定 *filename* 的链接。 在第二种形式中，每个 *filename*s 将依次链接到指定 *dir* 目录中具有相同最后路径名组件的路径名。

通常情况下，ln 不会尝试创建指向目录的硬链接。 可以使用 -d 选项覆盖这一检查。 通常只有超级用户才能成功创建指向目录的硬链接。 这在任何情况下都不适用于符号链接。

默认情况下，现有文件不能被链接替换。 -i 选项会询问用户是否替换现有文件。 而 -f 选项则会默默删除现有文件，无需询问。 -f 优先。

-h 和 -n 选项完全相同，都是为了兼容性而存在；其中任何一个都表示，如果目标是一个符号链接，则不应解引用。 通常情况下，该选项与 -sf 结合使用，这样一来，如果现有链接指向一个目录，则该链接将被移除，而不是被跟踪。 如果该选项与多个文件名一起使用，且目标是指向目录的符号链接，则结果会出错。

mkdir [-p] [-m *mode*] *dir ...*

创建目录。 使用 -p 选项时，如果有必要，会首先创建不存在的父目录，如果目录已经存在，则不会有任何抱怨。 可以使用 -m 选项为创建的目录指定一组文件权限（八进制），否则将使用经当前 umask 修改的 777 模式（参见 umask(2)）。

mv [-fi] *filename dest*
mv [-fi] *filename ... dir*

移动文件。 第一种形式是将指定的 *filename* 移动到指定的 *dest* 目的地。 第二种形式是依次将每个 *filename*s 移动到指定 *dir* 目录中具有相同最后路径名组件的路径名中。

默认情况下，在替换任何用户无法写入的文件前都会询问用户，但可写入的文件会被静默删除。 `-i` 选项会在替换任何现有文件时询问用户。 而 `-f` 选项则会在不询问用户的情况下默默删除任何现有文件。 `-f` 优先。

请注意，此 `mv` 不会跨设备移动文件。 历史版本的 `mv` 在无法进行实际重命名时，会退回到复制和删除文件的方式；如果需要这种行为，请手动使用 `cp` 和 `rm`。 未来版本可能会对此进行修改。

`rm` [`-dfiRrs`] *filename* ...

删除指定的文件和目录。

通常情况下，`rm` 不会删除目录（使用 `-R` 或 `-r` 选项时除外）。 `-d` 选项会让 `rm` 尝试使用 `unlink`（参见 unlink(2)）删除目录，这与删除文件的方法相同。 通常只有超级用户才能通过这种方法成功删除目录。 `-d` 优先于 `-R` 和 `-r`。

默认情况下，删除任何用户无法写入的文件前都会询问用户，但可写入的文件会被静默删除。 `-i` 选项会在删除任何文件时询问用户。 `-f`选项会导致静默删除文件，无需询问，并抑制所有错误提示。 `-f` 优先。

`-R` 和 `-r` 选项会使 `rm` 以递归方式进入目录，删除目录中的所有文件，然后再使用 `rmdir` 系统调用删除目录（参见 rmdir(2)）。

`-s` 选项是 zsh 对 `rm` 功能的扩展。 它启用了偏执行为，旨在避免常见的安全问题，即 root 运行的 `rm` 会被诱骗删除预期之外的文件。 它会拒绝跟踪符号链接，因此（例如）如果 `/tmp/foo` 恰好是 `/etc` 的链接，“`rm /tmp/foo/passwd`” 就不会意外删除 `/etc/passwd`。 它还会检查离开目录后的位置，这样在递归删除深目录树时，就不会因为目录被上移而递归删除 `/usr`。

`rmdir` *dir* ...

删除指定的空目录。

`sync`

调用同名系统调用（参见 sync(2)），将脏缓冲区刷新到磁盘。 它可能会在 I/O 实际完成之前返回。

22.15 zsh/langinfo 模块

`zsh/langinfo` 模块提供了一个参数：

`langinfo`

关联数组，用于将 langinfo 元素映射到其值。

您的实现可能支持以下多个键(keys)：

CODESET, D_T_FMT, D_FMT, T_FMT, RADIXCHAR, THOUSEP, YESEXPR, NOEXPR, CRNCYSTR, ABDAY_{1..7}, DAY_{1..7}, ABMON_{1..12}, MON_{1..12}, T_FMT_AMPM, AM_STR, PM_STR, ERA, ERA_D_FMT, ERA_D_T_FMT, ERA_T_FMT, ALT_DIGITS

22.16 zsh/mapfile 模块

zsh/mapfile 模块提供了一个同名的特殊关联数组参数。

mapfile

这个关联数组的键是文件名，结果值是文件的内容。 该值的处理方式与来自参数的任何其他文本相同。 该值也可以被赋值，在这种情况下，相关文件将被写入（无论其是否原本存在）；或者一个元素可以被取消设置，这将删除相关文件。 例如，‘vared 'mapfile[myfile]'’ 就会如预期般编辑文件 ‘myfile’。

当作为一个整体访问数组时，键是当前目录下的文件名，而值是空的（以节省大量内存开销）。 因此 ${(k)mapfile} 与 glob 操作符 *(D) 的效果相同，因为以点开头的文件并不特殊。 在使用 rm ${(k)mapfile} 这样的表达式时必须小心，因为它将删除当前目录下的所有文件，而无需进行通常的 ‘rm *’ 测试。

参数 mapfile 可以设置为只读；在这种情况下，不能写入或删除引用的文件。

文件可以方便地以每行一个元素的形式读入数组，其形式为 ‘*array*=("${(f@)mapfile[*filename*]}")’。 双引号和 ‘@’ 是防止删除空行所必需的。 需要注意的是，如果文件以换行结束，shell 会在最后一个换行处分隔，产生一个额外的空字段；可以使用 ‘*array*=("${(f@)${mapfile[*filename*]%$'\n'}}")’ 来抑制这种情况。

22.16.1 局限性

虽然读写文件的效率很高，zsh 的内部内存管理也可能很随意，但 mapfile 通常比任何涉及循环的操作都要高效。 需要特别注意的是，文件的全部内容在被访问时（由于标准参数替换操作，可能会被多次访问）都会实际存在于内存中。 这尤其意味着处理足够长的文件（大于机器的交换空间或超出指针类型的范围）将是不正确的。

对于不存在、不可读或不可写的文件，不会打印或标志错误，因为参数机制在 shell 执行层次结构中的位置太低，不方便这样做。

遗憾的是，加载模块的机制还不允许用户指定 shell 参数名称来赋予特殊行为的。

22.17 zsh/mathfunc 模块

zsh/mathfunc 模块为数学公式求值提供了标准数学函数。 其语法与 C 和 FORTRAN 的常规语法一致，例如,

```
(( f = sin(0.3) ))
```

将 0.3 的正弦值赋值给参数 f。

大多数函数使用浮点参数并返回浮点数值。 不过，任何必要的整数类型转换都将由 shell 自动执行。 除了带有第二个参数的 atan，以及 abs、int 和 float 函数外，所有函数的行为都与相应 C 语言函数手册中的说明相同，但任何超出函数范围的参数都会被 shell 检测到并报错。

以下函数只接受一个浮点参数： acos, acosh, asin, asinh, atan, atanh, cbrt, ceil, cos, cosh, erf, erfc, exp, expm1, fabs, floor, gamma, j0, j1, lgamma, log, log10, log1p, log2, logb, sin, sinh, sqrt, tan, tanh, y0, y1。 atan 函数可以接受可选的第二个参数，在这种情况下，它的行为类似于 C 语言函数 atan2。 ilogb 函数接收一个浮点参数，但返回一个整数。

函数 signgam 不带参数，返回一个整数，即 gamma(3) 中描述的同名 C 变量。 请注意，该函数只有在调用 gamma 或 lgamma 之后才有用。 还要注意，'signgam()' 和 'signgam' 是不同的表达式。

函数 min、max 和 sum 不是在本模块中定义的，而是在 数学函数 中描述的 zmathfunc 可自动加载函数中定义的。

以下函数使用两个浮点参数： copysign、fmod、hypot、nextafter。

以下函数的第一个参数为整数，第二个参数为浮点数： jn、yn。

以下函数的第一个参数为浮点数，第二个参数为整数： ldexp, scalb.

函数 abs 并不转换其单一参数的类型，而是返回浮点数或整数的绝对值。 函数 float 和 int 分别将参数转换为浮点数或整数（通过截断）。

请注意，C的 pow 函数在普通数学运算中可作为 '**' 运算符使用，此处不再提供。

如果系统的数学库中有 erand48(3)，则函数 rand48 可用。 它返回一个介于 0 和 1 之间的伪随机浮点数。它接受一个字符串作为可选参数。

如果不存在该参数，随机数种子将通过三次调用 rand(3) 函数来初始化 — 这将产生与 $RANDOM 接下来三个值相同的随机数。

如果存在该参数，它将给出一个标量参数的名称，当前随机数种子将存储在该参数中。 第一次调用时，该值必须至少包含 12 个十六进制数字（字符串的其余部分将被忽略），否则种子将以与调用 rand48 (不带参数)时相同的方式初始化。 随后调用

rand48(*param*)时，参数 *param* 中的种子将保持为一个包含十二位十六进制数字的字符串，不含基数符号。 不同参数的随机数序列是完全独立的，也独立于不带参数的 rand48 调用所使用的随机数序列。

例如，考虑

```
print $(( rand48(seed) ))
print $(( rand48() ))
print $(( rand48(seed) ))
```

假设 $seed 不存在，第一次调用将初始化它。 在第二次调用中，将初始化默认种子；但请注意，由于 rand() 的属性，两次初始化使用的种子之间存在关联，因此为了更安全地使用，应自己生成 12 字节的种子。 第三次调用将返回第一次调用中使用的随机数序列，不受中间 rand48() 的影响。

22.18 zsh/nearcolor 模块

zsh/nearcolor 模块会用终端模拟器广泛使用的 88 或 256 色调色板中最接近的颜色替换以十六进制三连字符指定的颜色。 默认情况下，使用十六进制三连字符指定颜色时，会生成 24 位真彩色转义码。 但并非所有终端都支持这些代码。 本模块的目的是让用户能更方便地定义颜色偏好，并能在各种终端模拟器中使用。

除默认颜色外，ANSI 终端转义代码标准还规定了八种颜色。亮色属性将其增加到 16 种。这些基本颜色由于得到广泛支持，在终端应用程序中得到普遍使用。扩展的 88 色和 256 色调色板也很常见，虽然前 16 种颜色在不同终端和配置之间存在一定差异，但它们提供了一套基本一致且可预测的颜色。

要使用 zsh/nearcolor 模块，只需加载该模块即可。此后，每当使用十六进制三元组指定颜色时，都会将其与每个可用颜色进行比较，并选择最接近的颜色。由于无法预测，前 16 种颜色在此过程中永远不会匹配。

在终端模拟器中无法可靠地检测对真彩色的支持。因此，建议有选择性地加载 zsh/nearcolor 模块。例如，下面的代码会检查 COLORTERM 环境变量：

```
[[ $COLORTERM = *(24bit|truecolor)* ]] || zmodload zsh/nearcolor
```

请注意，有些终端接受真彩色转义码，但内部会将其映射到更有限的调色板上，其方式与 zsh/nearcolor 模块类似。

22.19 zsh/newuser 模块

如果 zsh/newuser 模块可用、RCS 选项已设置、PRIVILEGED 选项未设置（默认情况下三者均为 true），则会在启动时加载该模块。 如果全局 zshenv 文件（通常为 /etc/zshenv）中有命令，则在执行完这些命令后立即启动。 如果模块不可用，shell 会静默忽略；如果不需要该模块，管理员可以安全地将其从 $MODULE_PATH 中删除。

加载时，模块会测试 .zshenv、.zprofile、.zshrc 或 .zlogin 中的启动文件是否存在于环境变量 ZDOTDIR 指定的目录中，如果未设置环境变量 ZDOTDIR 则测试用户的主目录。 如果 shell 处于仿真模式（即以其他 shell 而非 zsh 的身份调用），则不会执行测试，模块会停止处理。

如果没有找到任何启动文件，模块会首先在全站目录（通常是 site-functions 目录的父目录）中查找 newuser 文件，如果没有找到，模块会在特定版本目录（通常是包含特定版本函数的 functions 目录的父目录）中查找。 (在构建 zsh 时，可以分别使用 configure 的 --enable-site-scriptdir=*dir* 和 --enable-scriptdir=*dir* 标志来配置这些目录； 默认值为*prefix*/share/zsh 和 *prefix*/share/zsh/$ZSH_VERSION ，其中默认 *prefix* 为 /usr/local。)

如果找到 newuser 文件，则会以与启动文件相同的方式引入该文件。 该文件预计将包含为用户安装启动文件的代码，但任何有效的 shell 代码都将被执行。

随后 zsh/newuser 模块将无条件卸载。

请注意，通过在 /etc/zshenv 中添加代码，可以实现与 zsh/newuser 模块完全相同的效果。 该模块的存在仅仅是为了让 shell 在不需要软件包维护者和系统管理员干预的情况下安排新用户。

与模块一起提供的脚本会调用 shell 函数 zsh-newuser-install。 即使 zsh/newuser 模块被禁用，用户也可以直接调用该函数。 但请注意，如果模块未安装，函数也不会安装。 该函数在 用户配置函数 中有详细说明。

22.20 zsh/parameter 模块

zsh/parameter 模块通过定义一些特殊参数来访问 shell 使用的某些内部哈希表。

选项

> 这个关联数组的键是可使用 setopt 和 unsetopt 内置函数设置和取消设置的选项名称。如果选项当前已设置，则每个键的值都是 on 字符串；如果选项未设置，则每个键的值都是 off 字符串。 将键值设置为其中一个字符串，就相当于设置或取消设置该选项。取消设置数组中的某个键，就相当于将其设置为 off 值。

commands

> 该数组用于访问命令哈希表。键是外部命令的名称，值是调用该命令时将执行的文件的路径名。在此数组中设置键，将在此表中定义一个新条目，方法与 hash 内置函数相同。取消设置键，如‘unset "commands[foo]"’，则从命令哈希表中删除给定键的条目。

函数

这个关联数组将已启用函数的名称映射到其定义。在其中设置一个键，就等于定义了一个名称由键给出、主体由值给出的函数。取消键值设置则会删除键值所指定函数的定义。

dis_functions

与 functions 类似，但用于禁用函数。

functions_source

这个只读关联数组将已启用函数的名称映射到包含函数源代码的文件名。

对于已加载的自动加载函数，或用绝对路径已标记为自动加载的函数，或已使用 'functions -r' 解析其路径的函数，这是找到的自动加载的文件，解析为绝对路径。

对于在脚本或源文件正文中定义的函数，这是文件的名称。 在这种情况下，这是该文件最初使用的确切路径，也可以是相对路径。

对于任何其他函数，包括在交互式提示符下定义的函数或路径尚未解析的自动加载函数，该值为空字符串。 不过，只要函数存在，哈希元素就会被报告为已定义：该哈希的键s与 $functions 的键s相同。

dis_functions_source

与 functions_source 类似，但用于禁用的函数。

builtins

这个关联数组提供了当前已启用的内置命令的信息。键是内置命令的名称，值是 'undefined'（如果调用，将自动从模块加载的内置命令）或 'defined'（已经加载的内置命令）。

dis_builtins

与 builtins 类似，但用于禁用的内置命令。

reswords

该数组包含已启用的保留字。

dis_reswords

与 reswords 类似，但用于禁用的保留字。

patchars

该数组包含启用的模式字符。

dis_patchars

与 patchars 类似，但用于禁用的模式字符。

aliases

这将当前启用的常规别名的名称映射到其扩展。

dis_aliases

与 aliases 类似，但禁用了常规别名。

galiases

类似 aliases，但用于全局别名。

dis_galiases

与 galiases 类似，但禁用了全局别名。

saliases

与 raliases 类似，但用于后缀别名。

dis_saliases

与 saliases 类似，只是禁用了后缀别名。

参数

这个关联数组的键是当前定义的参数名称。值是描述参数类型的字符串，格式与 t 参数标志相同，参见 参数扩展 。 无法设置或取消设置数组中的键。

modules

提供模块信息的关联数组。键是已加载、已注册为自动加载或已有别名的模块名称。值表示被命名模块所处的状态，是字符串'loaded', 'autoloaded' 或 'alias:*name*' 之一，其中 *name* 是模块的别名。

无法设置或取消设置该数组中的键。

dirstack

一个包含目录栈元素的普通数组。请注意，dirs 内置命令的输出还包括一个目录，即当前工作目录。

history

该关联数组将历史事件编号映射到完整的历史行。 虽然它是以关联数组的形式呈现，但所有值的数组（`${history[@]}`）保证按照从最近的历史事件到最古老的历史事件的顺序返回，即按照历史事件编号递减的方式返回。

`historywords`

一个特殊数组，包含存储在历史记录中的单词。 这些词也是按从最近到最久远的顺序排列的。

`jobdirs`

这个关联数组将作业编号映射到作业启动的目录（可能不是作业的当前目录）。

关联数组的键通常是有效的作业编号，这些是,例如 `${(k)jobdirs}` 输出的值。在查找值时，可以使用非数字作业引用；例如，`${jobdirs[%+]}` 指的是当前作业。

有关如何在子 shell 中提供作业信息，请参阅 `jobs` 内置函数。

`jobtexts`

这个关联数组将作业编号映射到用于启动作业的命令行文本。

对关联数组键的处理与上文 `jobdirs` 的描述相同。

有关如何在子 shell 中提供作业信息，请参阅 `jobs` 内置函数。

`jobstates`

这个关联数组提供了当前已知作业的状态信息。键是作业编号，值是格式为 '*job-state*:*mark*:*pid*=*state*...' 的字符串。*job-state*给出了整个作业当前所处的状态，即 'running', 'suspended' 或 'done'。*mark* 中的 '+' 表示当前作业，'-' 表示上一个作业，否则为空。随后，作业中的每个进程都会有一个 ':*pid*=*state*'。*pid*s 当然是进程 ID，而 *state* 则描述该进程的状态。

对关联数组键的处理与上文 `jobdirs` 的描述相同。

有关如何在子 shell 中提供作业信息，请参阅 `jobs` 内置函数。

`nameddirs`

这个关联数组将命名目录的名称映射到它们所代表的路径名。

`userdirs`

这个关联数组将用户名映射到其主目录的路径名。

`usergroups`

该关联数组将当前用户所属的系统组名称映射到相应的组标识符。 其内容与 id 命令输出的组相同。

`funcfiletrace`

该数组包含当前函数、引入的文件或（如果设置了 EVAL_LINENO）eval 命令被调用时的绝对行号和相应文件名。 该数组的长度与 funcsourcetrace 和 functrace 相同，但与 funcsourcetrace 不同的是，行号和文件是调用点，而不是定义点；与 functrace 不同的是，所有值都是文件中的绝对行号，而不是相对于函数（如果有）起始位置的行号。

`funcsourcetrace`

该数组包含定义当前执行的函数、引入的文件和（如果设置了 EVAL_LINENO）eval 命令的文件名和行号。 行号指 'function *name*' 或'*name* ()' 的起始行。 在自动加载函数的情况下，行号报告为零。 每个元素的格式为 *filename*:*lineno*。

对于从原生 zsh 格式文件中自动加载的函数（文件中只有函数的主体），或者通过 source 或 '.' 内置函数执行的文件，跟踪信息将显示为 *filename*:*0*，因为整个文件都是定义文件。 在加载函数时，源文件名会解析为绝对路径，否则就解析为其路径。

大多数用户会对 funcfiletrace 数组中的信息感兴趣。

`funcstack`

该数组包含当前正在执行的函数、引入的文件和（如果设置了 EVAL_LINENO）eval 命令的名称。第一个元素是使用参数的函数的名称。

标准 shell 数组 zsh_eval_context 可用来确定在每个深度中执行的 shell 结构类型：但要注意的是，该数组的顺序正好相反，最新的项目在最后，而且更加详细，例如包含了 toplevel 的条目，即交互式或脚本中执行的主要 shell 代码，而 $funcstack 中并不包含这些代码。

`functrace`

该数组包含与当前正在执行的函数相对应的调用者的名称和行号。 每个元素的格式为 *name*:*lineno*。 对于引入的文件，也会显示调用者；调用者是执行 source 或 '.' 命令的位置。

22.21 zsh/pcre 模块

zsh/pcre 模块提供了一些内置命令：

`pcre_compile` [`-aimxs`] *PCRE*

编译与 perl 兼容的正则表达式。

选项 -a 将强制锚定模式。 选项 -i 将编译不区分大小写的模式。 选项 -m 将编译多行模式，即 ^ 和 $ 将匹配模式内的换行符。 选项 -x 将编译扩展模式，其中空白和 # 注释将被忽略。 选项 -s 使点元字符匹配所有字符，包括表示换行的字符。

pcre_study

研究先前编译的 PCRE，这可能会加快匹配速度。

pcre_match [-v *var*] [-a *arr*] [-n *offset*] [-b] *string*

如果 string 与之前编译的 PCRE 匹配，则成功返回。

匹配成功后，如果表达式捕获了括号内的子字符串，pcre_match 将数组 match 设置为这些子字符串，除非给出 -a 选项，这样的情况下，将设置数组 *arr*。 同样，变量 MATCH 将被设置为字符串的整个匹配部分，除非给出 -v 选项，在这种情况下，变量 *var* 将被设置。 如果没有成功匹配，则不会更改变量。 -n 选项从 *string* 中的 *offset* 字节位置开始搜索匹配。 如果给出 -b 选项，变量 ZPCRE_OP 将被设置为偏移对(offset pair)字符串，代表 *string* 中整个匹配部分的字节偏移位置。 例如，设置为 "32 45" 的 ZPCRE_OP 表示匹配部分从字节偏移位置 32 开始，到字节偏移位置 44 结束。 在这里，字节偏移位置 45 是直接位于匹配部分之后的位置。 请注意，在涉及 UTF-8 字符时，字节位置并不一定与字符位置相同。 因此，字节偏移位置只能用于 *string* 的后续搜索，将偏移位置作为 -n 选项的参数。 这主要用于实现 "查找所有非重叠匹配 " 功能。

"查找所有非重叠匹配项 " 的一个简单示例：

```
string="The following zip codes: 78884 90210 99513"
pcre_compile -m "\d{5}"
accum=()
pcre_match -b -- $string
while [[ $? -eq 0 ]] do
    b=($=ZPCRE_OP)
    accum+=$MATCH
    pcre_match -b -n $b[2] -- $string
done
print -l $accum
```

zsh/pcre 模块提供了以下测试条件：

expr -pcre-match *pcre*

将字符串与 perl 兼容的正则表达式匹配。

例如,

```
    [[ "$text" -pcre-match ^d+$ ]] &&
    print text variable contains only "d's".
```

如果设置了 REMATCH_PCRE 选项，则 =~ 操作符等同于 -pcre-match，并且可以使用 NO_CASE_MATCH 选项。 请注意，NO_CASE_MATCH 并不适用于 pcre_match 内置函数，请使用 pcre_compile 的 -i 开关。

22.22 zsh/param/private 模块

zsh/param/private 模块用于创建参数，其作用域仅限于当前函数体，而**不是**当前函数调用的其他函数。

该模块提供一个自动加载的内置命令：

private [{+|-}AHUahlmrtux] [{+|-}EFLRZi [*n*]] [*name*[=*value*] ...]

除了 '-T' 选项，private 内置命令接受与 local（Shell 内置命令）相同的选项和参数。 绑定的参数不能被私有化。

目前，'-p' 选项是无效的，因为私有参数的状态无法可靠地重新加载。 这也适用于使用 'typeset -p' 打印私有参数。

如果在顶层（函数作用域之外）使用，private 将以与 declare 或 typeset 相同的方式创建一个普通参数。 如果设置了 WARN_CREATE_GLOBAL（选项），系统将对此发出警告。 在函数作用域内使用 private，会创建一个与 local 声明的参数类似的本地参数，但具有以下特殊属性。

暴露或操作 shell 内部状态的特殊参数，例如 ARGC, argv, COLUMNS, LINES, UID, EUID, IFS, PROMPT, RANDOM, SECONDS等，除非使用了 '-h' 选项来隐藏参数的特殊含义，否则不能将其设置为私有。 这种情况将来可能会改变。

与其他 typeset 类似，private 既是内置程序，也是保留字，因此可以使用括号内的词列表 *name*=(*value*...) 语法为数组赋值。 但是，在 zsh/param/private 加载之前，保留字 'private' 是不可用的，因此必须注意使用 private 的函数定义的执行和解析顺序。 为了弥补这一不足，本模块还为内置的 'local' 添加了 '-P' 选项，用于声明私有参数。

例如，如果在定义 'bad_declaration' 时 zsh/param/private 尚未加载，则此构造将失败：

```
bad_declaration() {
  zmodload zsh/param/private
  private array=( one two three )
}
```

这种结构之所以有效，是因为 local 已经是一个关键字，而且在执行语句之前模块已经加载：

```
good_declaration() {
  zmodload zsh/param/private
  local -P array=( one two three )
}
```

以下内容可在脚本中使用，但在使用 autoload 时可能会有问题：

```
zmodload zsh/param/private
iffy_declaration() {
  private array=( one two three )
}
```

private 内置函数始终可用于标量赋值和不带赋值的声明。

用 private 声明的参数具有以下属性：

- 在声明该参数的函数体中，该参数的行为与本地参数相同，但如上所述的绑定参数或特殊参数除外。
- 声明为私有的参数的类型不能在声明该参数的作用域中更改，即使该参数未被设置。 因此，数组不能赋值给私有标量等。
- 在声明函数调用的任何其他函数中，私有参数 **不会** 隐藏同名的其他参数，例如，同名的全局参数是可见的，可以被赋值或取消赋值。 这也包括对匿名函数的调用，不过这种情况将来可能会改变。 不过，如果之前未声明私有名称，则不得在本地作用域之外创建私有名称。
- 导出的私有参数会保留在内部作用域的环境中，但对于这些作用域中的当前 shell 而言，则显示为未设置。 一般来说，应避免导出私有参数。

请注意，这不同于由 C 语言派生的编译语言所定义的静态作用域，在那里，对同一函数的新调用会创建一个新的作用域，也就是说，参数仍然与调用栈相关联，而不是与函数定义相关联。 它与 ksh 的 'typeset -S' 不同，因为定义函数所用的语法与参数作用域是否受到尊重没有关系。

22.23 zsh/regex 模块

zsh/regex 模块提供了以下测试条件：

expr -regex-match *regex*

> 根据 POSIX 扩展正则表达式匹配字符串。 匹配成功后，字符串的匹配部分通常会放入 MATCH 变量。 如果正则表达式中有捕获括号，那么 match 数组变量中将包含捕获。 如果匹配不成功，则不会更改变量。

例如,

```
[[ alphabetical -regex-match ^a([^a]+)a([^a]+)a ]] &&
print -l $MATCH X $match
```

如果未设置 REMATCH_PCRE 选项，那么 =~ 操作符将根据需要自动加载此模块，并调用 -regex-match 操作符。

如果设置了 BASH_REMATCH，那么数组 BASH_REMATCH 将被设置，而不是 MATCH 和 match。

请注意，zsh/regex 模块的逻辑依赖于主机系统。如果给出的 *regex* 使用了非标准语法，相同的 *expr* 和 *regex* 对可能会在不同平台上产生不同的结果。

例如，POSIX 扩展正则表达式标准中没有定义匹配单词边界的语法。GNU libc 和 BSD libc 都提供了此类语法作为扩展（分别为 \b 和 [[:<:]]/[[:>:]] ），但这两种实现都不支持这些语法。

有关本地支持的语法，请参阅系统上的 regcomp(3) 和 re_format(7) 手册。

22.24 zsh/sched 模块

zsh/sched 模块提供一条内置命令和一个参数。

sched [-o] [+]*hh*:*mm*[:*ss*] *command* ...
sched [-o] [+]*seconds command* ...
sched [-*item*]

在计划执行的命令列表中加入一个条目。 时间可以指定为绝对时间或相对时间，也可以指定为用冒号分隔的小时、分钟和秒（可选），或仅指定为秒。 绝对秒数表示自 epoch（1970/01/01 00:00）以来的时间；与 zsh/datetime 模块中的功能结合使用非常有用，参见 zsh/datetime 模块。

在没有参数的情况下，打印计划命令列表。 如果计划命令设置了 -o 标志，则会在命令开始时显示。

使用参数 '-*item*'，从列表中删除给定条目。 列表的编号是连续的，条目按时间顺序排列，因此在添加或删除条目时，编号可能会发生变化。

命令要么在提示符前立即执行，要么在 shell 的行编辑器等待输入时执行。 在后一种情况下，最好能产生不影响正在编辑的行的输出。 提供 -o 选项会使 shell 在事件发生前清除命令行，并在事件发生后重新绘制命令行。 该选项适用于任何向终端产生可见输出的计划事件；例如，不需要用于更新终端模拟器标题栏的输出。

要在事件执行时更改编辑器缓冲区，可使用不带参数的 'zle' 命令测试编辑器是否激活，如果激活，则使用 'zle *widget*' 通过指定的 *widget* 访问编辑器。

当 shell 以模拟其他 shell 的模式启动时，sched 内置函数默认不可用。 可以通过命令 'zmodload -F zsh/sched b:sched' 来使它可用。

zsh_scheduled_events

一个只读数组，对应 sched 内置程序所调度的事件。 数组的索引与不带参数运行 sched 时显示的数字相对应（前提是未设置 KSH_ARRAYS 选项）。 数组的值包括以秒为单位的从 epoch 开始的计划时间（请参阅 zsh/datetime 模块 以了解使用该数字的方法），之后是冒号，之后是任何选项（选项可以为空，但必须在选项前加上 '-' ），之后是冒号，之后是要执行的命令。

应使用 sched 内置函数来操作事件。 请注意，这将对数组内容产生直接影响，因此索引可能会失效。

22.25 zsh/net/socket 模块

zsh/net/socket 模块提供了一条内置命令：

zsocket [-altv] [-d *fd*] [*args*]

zsocket 是作为内置命令实现的，可以充分利用 shell 命令行编辑、文件 I/O 和作业控制机制。

22.25.1 出站连接

zsocket [-v] [-d *fd*] *filename*

为 *filename* 打开一个新的 Unix 域连接。 shell 参数 REPLY 将被设置为与该连接相关联的文件描述符。 目前只支持流连接。

如果指定了 -d，其参数将作为连接的目标文件描述符。

为了获得更详细的输出，请使用 -v。

例如，当不再需要文件描述符时，可以使用正常的 shell 语法关闭文件描述符：

```
exec {REPLY}>&-
```

22.25.2 入站连接

zsocket -l [-v] [-d *fd*] *filename*

zsocket -l 将打开一个监听 *filename* 的套接字。 shell 参数 REPLY 将被设置为与该监听器关联的文件描述符。 该文件描述符在子 shell 和分叉的外部可执行文件中保持打开状态。

如果指定了 -d，其参数将作为连接的目标文件描述符。

为了获得更详细的输出，请使用 -v。

zsocket -a [-tv] [-d *targetfd*] *listenfd*

zsocket -a 将接受与 *listenfd* 关联的套接字的入站连接。 shell 参数 REPLY 将被设置为与入站连接相关的文件描述符。 该文件描述符将在子 shell 和分叉的外部可执行文件中保持打开状态。

如果指定了 -d，其参数将作为连接的目标文件描述符。

如果指定了 -t，zsocket 将在没有新的连入连接时返回。 否则将等待连接。

为了获得更详细的输出，请使用 -v。

22.26 zsh/stat 模块

zsh/stat 模块以两种可能的名称提供一条内置命令：

zstat [-gnNolLtTrs] [-f *fd*] [-H *hash*] [-A *array*] [-F *fmt*]
[+*element*] [*file* ...]
stat ...

该命令是 stat 系统调用（参见 stat(2)）的前端。 同一命令有两个名称；由于 stat 经常被外部命令使用，建议只使用 zstat 形式的命令。 这可以通过使用 'zmodload -F zsh/stat b:zstat' 命令加载模块来实现。

如果 stat 调用失败，将打印相应的系统错误信息并返回状态 1。 struct stat 的字段提供了作为命令参数提供的文件信息。 除了 stat 调用提供的信息外，还提供了一个额外的元素 'link'。 这些元素是:

device

文件所在设备的编号。

inode

该设备上文件的唯一编号（'***inode***' 编号）。

mode

文件的模式，即文件的类型和访问权限。 使用 -s 选项后，将以字符串形式返回，与 ls -l 命令显示的第一列相对应。

nlink

文件的硬链接数量。

`uid`

文件所有者的用户 ID。 如果使用 `-s` 选项，则显示为用户名。

`gid`

文件的组 ID。 如果使用 `-s` 选项，则显示为组名。

`rdev`

原始设备编号。 这只对特殊设备有用。

`size`

文件的大小（字节）。

`atime`
`mtime`
`ctime`

文件的最后访问时间、修改时间和 inode 更改时间，分别为格林尼治标准时间 1970 年 1 月 1 日午夜后的秒数。 使用 `-s` 选项时，这些时间将以当地时区的字符串形式打印；使用 `-F` 选项可更改格式；使用 `-g` 选项时，时间以格林尼治标准时间。

`blksize`

文件所在设备上一个分配块的字节数。

`block`

文件使用的磁盘块数。

`link`

如果文件是链接文件，且 `-L` 选项有效，这将包含链接文件的名称，否则为空。 请注意，如果选择了该元素 (“`zstat +link`”)，则会自动使用 `-L` 选项。

在选项列表中，可以通过在元素名称前加上 ‘+’ 来选择特定元素；只允许选择一个元素。 该元素可以缩短为任意一组唯一的前导字符。 否则，将显示所有文件的所有元素。

选项:

`-A` *array*

与其将结果显示在标准输出上，不如将其分配到 *array* 中，每个数组元素按顺序为每个文件分配一个 `struct stat` 元素。 在这种情况下，除非分别给出 `-t` 或 `-n` 选项，否则元素名称和文件名称都不会出现在 *array* 中。 如果给定了 `-t`，元素名将作为前缀出现在相应的数组元素中；如果给定了 `-n`，文件名将作为单独的数组元素出现在所有其他元素之前。 其他格式化选项也会得到遵守。

`-H` *hash*

与 `-A` 类似，但会将值赋给 *hash*。 键是上面列出的元素。 如果提供 `-n` 选项，文件名将包含在哈希中，键值为 `name`。

`-f` *fd*

使用文件描述符 *fd* 上的文件，而不是命名文件；在这种情况下，不允许使用文件名列表。

`-F` *fmt*

提供用于格式化时间元素的 `strftime`（参见 strftime(3)）字符串。 格式字符串支持 提示符扩展 中描述的所有 zsh 扩展。 特别是 `-F %s.%N`，如果系统支持，可用于显示纳秒精度的时间戳。 `-s` 选项是隐含的。

`-g`

显示 GMT 时区的时间元素。 `-s` 选项是隐含的。

`-l`

列出类型元素的名称（视情况输出到标准输出或数组）并立即返回；忽略参数和 `-A` 以外的选项。

`-L`

执行 `lstat`（见 lstat(2)）而不是 `stat` 系统调用。 在这种情况下，如果文件是链接，返回的是链接本身的信息，而不是目标文件的信息。 要使 `link` 元素发挥作用，就必须使用该选项。 需要注意的是，这与 ls(1) 等完全相反。

`-n`

始终显示文件名。 通常只有在输出到标准输出且列表中有多过一个文件时才会显示这些文件名。

`-N`

从不显示文件名。

`-o`

如果打印的是原始文件模式，则以八进制显示，因为八进制比默认的十进制更适合人类使用。 在这种情况下，将打印一个前导零。 请注意，这并不影响显示原始文件模式还是格式化文件模式（由 -r 和 -s 选项控制），也不影响是否显示任何模式。

-r

打印原始数据（默认格式）和字符串数据（-s 格式）；字符串数据显示在原始数据后的括号内。

-s

将 mode、uid、gid 和三个时间元素打印为字符串而非数字。 每种情况下的格式都与 ls -l 相同。

-t

始终显示 struct stat 元素的类型名称。 通常只有在输出到标准输出且未选择单个元素时才会显示。

-T

从不显示 struct stat 元素的类型名称。

22.27 zsh/system 模块

zsh/system 模块提供各种内置命令和参数。

22.27.1 内置命令

syserror [-e *errvar*] [-p *prefix*] [*errno* | *errname*]

该命令将打印出与 *errno* 相关的错误信息、系统错误编号，并在标准错误后面加换行符。

可以使用名称 *errname* 代替错误编号，例如 ENOENT。 名称集与数组 errnos 的内容相同，见下文。

如果给出 *prefix* 字符串，则会将其打印在错误信息的前面，中间不留空格。

如果提供了 *errvar*，则整个信息（不带换行符）都会分配给参数名 *errvar*，不会输出任何内容。

返回状态为 0 表示信息已成功打印（但如果错误编号超出系统范围，则可能没有用处），返回状态为 1 表示参数出错，返回状态为 2 表示无法识别错误名称（不会打印任何信息）。

`sysopen` [`-arw`] [`-m` *permissions*] [`-o` *options*]
`-u` *fd file*

该命令用于打开文件。`-r`、`-w` 和 `-a` 标志分别表示文件是否应被打开用于读取、写入和追加。`-m` 选项允许以八进制形式在创建文件时使用的指定的初始权限。 文件描述符用 `-u` 指定。既可以指定一个范围在 0 到 9 之间的明确文件描述符，也可以指定一个变量名来分配文件描述符编号。

`-o` 选项允许以逗号分隔的列表形式指定各种系统特定选项。以下是可能的选项列表。请注意，根据系统的不同，有些选项可能不可用。

`cloexec`

在执行其他程序时标记文件已关闭（否则文件描述符在子 shell 和分叉的外部可执行文件中仍处于打开状态）

`create`
`creat`

如果不存在，则创建文件

`excl`

创建文件, 如果文件已存在则出错

`noatime`

抑制文件的 atime 更新

`nofollow`

如果 *file* 是符号链接则失败

`nonblock`

文件以非阻塞模式打开

`sync`

要求写入时等待数据已被实际写入

`truncate`
`trunc`

将文件截断为 0 大小

要关闭文件，请使用以下方法之一：

```
exec {fd}<&-
exec {fd}>&-
```

`sysread` [`-c` *countvar*] [`-i` *infd*] [`-o` *outfd*]
[`-s` *bufsize*] [`-t` *timeout*] [*param*]

从文件描述符 *infd* 执行单次系统读取，如果未指定，则为零。 读取结果存储在 *param* 中，如果未给出，则存储在 REPLY 中。 如果给定了 *countvar*，读取的字节数将分配给 *countvar* 命名的参数。

读取的最大字节数为 *bufsize*，如果没有给出，则为 8192，但只要成功读取任何字节数，命令就会返回。

如果给出 *timeout*，则指定了以秒为单位的超时时间，在轮询文件描述符时，超时时间可以为零。 如果可用，则由 `poll` 系统调用处理，否则由 `select` 系统调用处理。

如果给定了 *outfd*，则会尝试将刚刚读取的所有字节写入文件描述符 *outfd*。 如果由于 EINTR 以外的系统错误或中断期间的 zsh 内部错误而失败，读取但未写入的字节会存储在 *param* 指定的参数（如果提供了该参数，这时不使用默认值）中，读取但未写入的字节数会存储在 *countvar* 指定的参数（如果提供了该参数）中。 如果成功，*countvar* 会像往常一样包含传输的全部字节数，而 *param* 则不会被设置。

错误 EINTR (中断的系统调用）是内部处理的，因此 shell 中断对调用者是透明的。 任何其他错误都会导致返回。

可能的返回状态有

0

成功读取至少一个字节的数据，并酌情写入。

1

命令参数有误。 这是唯一会将信息打印到标准错误中的错误。

2

读取或轮询输入文件描述符超时时出现错误。 参数 ERRNO 提供了错误信息。

3

已成功读取数据，但向 *outfd* 中写入数据时出现错误。 参数 ERRNO 提供了错误信息。

4

尝试读取超时。 注意这不会设置 ERRNO，因为这不是系统错误。

未发生系统错误，但读取的字节数为零。 这通常表示文件已结束。 参数根据常规规则设置；不会尝试向 *outfd* 写入任何内容。

`sysseek` [`-u` *fd*] [`-w` `start`|`end`|`current`] *offset*

未来读写的当前文件位置将调整为指定的字节偏移量。*offset* 将以数学表达式的形式求值。`-u` 选项允许指定文件描述符。默认情况下，偏移量是相对于文件的起点，但通过 `-w` 选项，可以指定偏移量应相对于当前位置或文件的终点。

`syswrite` [`-c` *countvar*] [`-o` *outfd*] *data*

数据（字节的单个字符串）将通过 `write` 系统调用写入文件描述符 *outfd*，如果没有给出，则是 1。 如果第一个写入操作没有写入所有数据，可以使用多个写入操作。

如果给出 *countvar*，写入的字节数将存储在 *countvar* 命名参数中；如果发生错误，这可能不是 *data* 的全长。

`EINTR` 错误（系统调用中断）会通过重试进行内部处理。(系统调用中断）会通过重试进行内部处理，否则错误会导致命令返回。 例如，如果文件描述符设置为非阻塞输出，错误 `EAGAIN`（在某些系统中为 `EWOULDBLOCK`）可能导致命令提前返回。

返回状态可能是 0 以表示成功，1 表示命令参数出错，或 2 表示写入错误；最后一种情况不打印错误信息，但参数 `ERRNO` 将反映发生的错误。

`zsystem flock` [`-t` *timeout*] [`-i` *interval*] [`-f` *var*] [`-er`] *file*
`zsystem flock -u` *fd_expr*

内置的 `zsystem` 的子命令 `flock` 会对给定文件的全部内容执行咨询式文件锁定（advisory file locking）（通过 fcntl(2) 系统调用）。 这种形式的锁定要求访问文件的进程相互配合；其最明显的用途是在 shell 本身的两个实例之间使用。

在第一种形式中，名为 *file* 的文件必须已经存在，通过为该文件打开文件描述符并为文件描述符加锁来锁定该文件。 当创建锁的 shell 进程退出时，锁就会终止；因此，在子 shell 中创建文件锁通常很方便，因为子 shell 退出时，锁会自动释放。需要注意的是，使用 `print` 内置程序和 `-u` 选项的副作用是释放锁，就像重定向到持有锁的 shell 中的文件一样。 要解决这个问题，可以使用子 shell，例如‘`(print message) >>` *file*’。 如果锁定成功，则返回状态 0，否则返回状态 1。

在第二种形式中，算术表达式 *fd_expr* 给出的文件描述符被关闭，释放了锁。 在锁定过程中，可以使用 ‘`-f` *var*’ 形式查询文件描述符；锁定成功后，shell 变量 *var* 将被设置为锁定时使用的文件描述符。 如果以其他方式关闭文件描述符，例如使用 ‘`exec {`*var*`}>&-`’，锁将被释放；不过，这里描述的形式会执行安全检查，确保文件描述符用于文件锁定。

默认情况下，shell 会无限期等待锁定成功。 选项 `-t` *timeout* 以秒为单位指定锁定的超时时间；允许使用小数秒。 在此期间，如果使用 `-i` *interval* 选项，shell 将每隔 *interval* 秒尝试锁定文件一次，否则每秒锁定一次。 (如果需要，这个 *interval* 会在最后一次尝试前缩短，这样 shell 就只能等待 *timeout* 而不是更长的时间）。 如果尝试超时，将返回状态 2。

(注意 *timeout* 限制为 2^30-1 秒（约 34 年），*interval* 限制为 0.999 * LONG_MAX 微秒（在 32 位系统上仅约 35 分钟））。

如果给定了选项 `-e`，当 shell 使用 `exec` 启动新进程时，锁的文件描述符将被保留；否则，它将在此时关闭并释放锁。

如果给出选项 `-r`，锁只用于读取，否则用于读写。 文件描述符也会相应打开。

`zsystem supports` *subcommand*

内置 `zsystem` 的子命令 `supports` 会测试给定的子命令是否受支持。 如果支持，则返回状态 0，否则返回状态 1。这将静默操作，除非出现语法错误（即参数个数错误），这时将返回状态 255。 状态 1 表示以下两种情况之一： *subcommand* 已知但当前操作系统不支持，或者 *subcommand* 未知（可能是因为这是一个尚未实现该命令的旧版 shell）。

22.27.2 数学函数

`systell(fd)`

systell 数学函数返回作为参数传递的文件描述符的当前文件位置。

22.27.3 参数

`errnos`

系统中定义的错误名称的只读数组。 这些通常是通过包含系统头文件 `errno.h` 在 C 语言中定义的宏。 每个名称的索引（假设选项 KSH_ARRAYS 未设置）与错误编号相对应。 在最后一个已知错误之前没有名称的错误编号 *num* 将在数组中命名为 E*num*。

请注意，错误的别名不会被处理，只会使用规范名称。

`sysparams`

一个只读关联数组。 键是

`pid`

返回当前进程的进程 ID，即使在子 shell 中也是如此。 相比之下，`$$` 返回的是主 shell 进程的进程 ID。

ppid

返回当前进程的父进程 ID，即使在子 shell 中也是如此。 比较 $PPID，它返回的是主 shell 进程初始父进程的进程 ID。

procsubstpid

返回进程替换时最后启动的进程的进程 ID，即 <(...) 和 >(...) 扩展。

22.28 zsh/net/tcp 模块

zsh/net/tcp 模块提供了一条内置命令：

ztcp [-acflLtv] [-d *fd*] [*args*]

ztcp 是作为内置命令实现的，可以充分利用 shell 命令行编辑、文件 I/O 和作业控制机制。

如果 ztcp 不带任何选项，它将输出会话表的内容。

如果运行时只使用 -L 选项，则会以适合自动解析的格式输出会话表内容。 如果与打开或关闭会话的命令一起使用，则忽略该选项。 输出由一系列行组成，每个会话一行，每行包含以下内容，以空格分隔：

文件描述符

连接使用的文件描述符。 对于正常的入站（I）和出站（O）连接，可以通过常用的 shell 机制读写该文件描述符。 不过，只能使用 'ztcp -c' 关闭。

连接类型

一个字母，说明会话是如何创建的：

Z

使用 zftp 命令创建的会话。

L

使用 'ztcp -l' 打开的侦听连接。

I

使用 'ztcp -a' 接受的入站连接。

O

使用 'ztcp *host* ...' 创建的出站连接。

本地主机

由于本地主机地址无关紧要，因此通常将其设置为全为零的 IP 地址。

本地端口

除非连接用于侦听，否则可能为零。

远程主机

如果有的话，这是对等体的完全合格域名，否则就是 IP 地址。 对于用于监听的会话，IP 地址全为零。

远程端口

这对于一个用于监听的连接来说是零。

22.28.1 出站连接

套接字、传入 TCP

打开与 *host* 的新 TCP 连接。 如果省略了 *port*，则默认端口为 23。 该连接将被添加到会话表中，shell 参数 REPLY 将被设置为与该连接关联的文件描述符。

如果指定了 -d，其参数将作为连接的目标文件描述符。

为了获得更详细的输出，请使用 -v。

22.28.2 入站连接

ztcp -l [-v] [-d *fd*] *port*

ztcp -l 将在 TCP *port* 上打开一个监听套接字。 该套接字将被添加到会话表中，shell 参数 REPLY 将被设置为与该监听器相关的文件描述符。

如果指定了 -d，其参数将作为连接的目标文件描述符。

为了获得更详细的输出，请使用 -v。

ztcp -a [-tv] [-d *targetfd*] *listenfd*

ztcp -a 将接受与 *listenfd* 关联端口的传入连接。 连接将被添加到会话表，shell 参数 REPLY 将被设置为与入站连接相关的文件描述符。

如果指定了 -d，其参数将作为连接的目标文件描述符。

如果指定了 -t，ztcp 将在没有入站连接的情况下返回。 否则将等待连接。

为了获得更详细的输出，请使用 -v。

22.28.3 关闭连接

`ztcp -cf [ -v ] [` *fd* `]`
`ztcp -c [ -v ] [` *fd* `]`

`ztcp -c` 将关闭与 *fd* 关联的套接字。 该套接字将从会话表中删除。 如果未指定 *fd*，`ztcp` 将关闭会话表中的所有内容。

通常情况下，zftp（参见 zsh/zftp 模块）注册的套接字不能以这种方式关闭。 要强制关闭此类套接字，请使用 `-f`。

为了获得更详细的输出，请使用 -v。

22.28.4 举例

下面介绍如何在两个 zsh 实例之间创建 TCP 连接。 我们需要选择一个未指定的端口；这里我们使用随机选择的 5123。

在 host1 上,

```
zmodload zsh/net/tcp
ztcp -l 5123
listenfd=$REPLY
ztcp -a $listenfd
fd=$REPLY
```

倒数第二个命令会阻塞，直到有连接进入。

现在从 host2（当然可能是同一台机器）创建一个连接：

```
zmodload zsh/net/tcp
ztcp host1 5123
fd=$REPLY
```

现在，在每台主机上，`$fd` 都包含一个文件描述符，用于与另一台主机对话。 例如，在 host1 上：

```
print This is a message >&$fd
```

在 host2 上:

```
read -r line <&$fd; print -r - $line
```

prints ‘`This is a message`’.

在 host1 上进行整理：

```
ztcp -c $listenfd
ztcp -c $fd
```

在 host2 上

```
ztcp -c $fd
```

22.29 zsh/termcap 模块

zsh/termcap 模块提供了一条内置命令：

echotc *cap* [*arg* ...]

输出与能力 *cap* 相对应的 termcap 值，参数可选。

zsh/termcap 模块提供了一个参数：

termcap

关联数组，用于将 termcap 能力代码映射到其值。

22.30 zsh/terminfo 模块

zsh/terminfo 模块提供了一条内置命令：

echoti *cap* [*arg*]

输出与能力 *cap* 相对应的 terminfo 值，如果适用，则使用 *arg* 进行实例化。

zsh/terminfo 模块提供一个参数：

terminfo

关联数组，用于将 terminfo 能力名称映射到其值。

22.31 zsh/watch 模块

zsh/watch 模块可用于报告特定用户何时登录或退出。可通过以下参数进行控制。

LOGCHECK

使用 watch 参数检查登录/注销活动的间隔时间（以秒为单位）。

watch <S> <Z> (WATCH <S>)

要报告的登录/注销事件数组（以冒号分隔的列表）。

如果包含单词 'all'，则报告所有登录/注销事件。 如果包含单词 'notme'，与 'notme' 相同，但报告除 $USERNAME 以外的所有事件。

该列表中的条目可以由用户名、 '@' 、远程主机名、'%'、线路（tty）组成。 其中任何一个都可以是一个模式（在赋值给 watch 时请务必加引号，以免立即生成文件）；EXTENDED_GLOB 选项的设置将受到尊重。 一个条目中可能包含上述任何或所有组件；如果登录/注销事件与所有组件都匹配，则会被报告。

例如，如果设置了 EXTENDED_GLOB 选项，就会出现以下情况：

```
watch=('^(pws|barts)')
```

会导致报告与 pws 或 barts 以外的任何用户相关的活动。

WATCHFMT

如果设置了 watch 参数，登录/注销报告的格式。 默认格式为 '%n has %a %l from %m'。 可识别以下转义序列：

%n

登录/退出的用户名。

%a

观察到的操作，即 "登录 "或 "注销"。

%l

用户登录的线路（tty）。

%M

远程主机的完整主机名。

%m

主机名，直至第一个 '.'。 如果只有 IP 地址或 utmp 字段包含 X-windows 显示器的名称，则打印整个名称。

注意: '%m' 和 '%M' 转义符只有在您的计算机的 utmp 中存在主机名字段时才起作用。 否则，它们将被视为普通字符串。

%F{*color*} (%f)

开始（停止）使用不同的前景色。

%K{*color*} (%k)

开始（停止）使用不同的背景颜色。

%S (%s)

启动（停止）突出模式。

%U (%u)

启动（停止）下划线模式。

%B (%b)

启动（停止）粗体模式。

%t
%@

时间，12 小时制，上午/下午格式。

%T

时间，24 小时制。

%w

日期，格式为 '*day-dd*' 。

%W

日期，格式为 '*mm/dd/yy*' 。

%D

日期，格式为 '*yy-mm-dd*'。

%D{*string*}

使用 `strftime` 函数，以提示符扩展 中描述的 zsh 扩展，将日期格式化为 *string*。

%(*x*:*true-text*:*false-text*)

指定一个三元表达式。*x* 后面的字符是任意的；同样的字符用于分隔 "true " 结果和 "false "结果。 分隔符和右括号都可以用反斜线转义。 三元表达式可以嵌套。

测试字符 *x* 可以是 '!', 'n', 'm' 或 'M' 中的任意一个，如果相应的转义序列会返回一个非空值，则表示结果为 'true' ；也可以是'a'，如果被监视用户已登录，

则表示结果为 'true' ；如果已注销，则表示结果为 'false'。 其他字符的结果既不是 true 也不是 false；在这种情况下，整个表达式将被省略。

如果结果为 'true'，则 *true-text* 将根据上述规则进行格式化并打印，*false-text* 将被跳过。 如果结果为 'false'，则跳过 *true-text*，格式化并打印 *false-text*。 任一分支或两个分支都可以为空，但无论如何，两个分隔符都必须存在。

此外，zsh/watch 模块还提供了一条内置命令：

log

列出受 watch 参数当前设置影响的所有当前登录用户。

22.32 zsh/zftp 模块

zsh/zftp 模块提供了一条内置命令：

zftp *subcommand* [*args*]

zsh/zftp 模块是 FTP（文件传输协议）的客户端。 它以内置命令方式实现，可充分利用 shell 命令行编辑、文件输入/输出和作业控制机制。 通常情况下，用户会通过 shell 函数访问该模块，而 shell 函数提供了更强大的接口；zsh 发行版中提供了一套函数，Zftp 函数系统中对其进行了描述。 不过，zftp 命令本身完全可用。

所有命令都由命令名 zftp 和子命令名组成。 下面列出了这些子命令。 每个子命令的返回状态都应反映远程操作的成功或失败。 有关如何打印服务器响应的更多信息，请参阅变量 ZFTP_VERBOSE 的说明。

22.32.1 子命令

open *host*[: *port*] [*user* [*password* [*account*]]]

打开连接到 *host* 的新 FTP 会话，*host* 可以是 TCP/IP 连接主机的名称，也可以是标准点号形式的 IP 号。 如果参数格式为 *host* : *port*，则将打开连接到 TCP 端口 *port* 而不是标准 FTP 端口 21。 该端口可以是 TCP 服务的名称或数字：更多信息请参阅下文对 ZFTP_PORT 的描述。

如果使用冒号格式的 IPv6 地址，*host* 应该用带引号的方括号括起来，以区别于 *port*，例如 '[fe80::203:baff:fe02:8b56]'。 为了保持一致性，所有形式的 *host* 都允许使用这种方式。

其余参数将传递给 login 子命令。 请注意，如果没有提供 *host* 以外的参数，open 将 **不会** 自动调用 login。 如果没有提供任何参数，open 将使用 params 子命令设置的参数。

成功打开后，shell 变量 ZFTP_HOST、ZFTP_PORT、ZFTP_IP 和 ZFTP_SYSTEM 将可用；请参阅下面的 '变量'。

login [*name* [*password* [*account*]]]
user [*name* [*password* [*account*]]]

使用参数 *password* 和 *account* 登录用户 *name*。 任何参数都可以省略，如果需要，将从标准输入中读取（*name* 始终需要）。 如果标准输入是终端，则会在标准错误中为每个参数打印出提示符，而 *password* 不会被回显。 如果未使用任何参数，系统将打印一条警告信息。

登录成功后，shell 变量 ZFTP_USER、ZFTP_ACCOUNT 和 ZFTP_PWD 可用；请参阅下面的 '变量'。

当用户已经登录时，可以再次发出该命令，服务器将首先为新用户重新初始化。

params [*host* [*user* [*password* [*account*]]]]
params -

存储给定参数，以备以后执行不带参数的 open 命令时使用。 只有命令行中给出的参数才会被记住。 如果没有给定参数，将打印当前设置的参数，但密码将以一行星号显示；如果没有设置参数，返回状态为 1，否则为 0。

任何参数都可以指定为 '?'，可能需要加引号以防止 shell 扩展。 在这种情况下，将像使用 login 子命令一样从 stdin 读取相应的参数，包括对 *password* 的特殊处理。 如果 '?' 后面跟了一个字符串，该字符串将被用作读取参数的提示符，而不是默认信息（任何必要的标点符号和空白都应包含在提示符末尾）。 参数的第一个字母（仅）可以用 '\' 加引号；因此参数 "\\$word" 将保证来自 shell 参数 $word 的字符串将按字面意思处理，无论其是否以 '?' 开头。

如果只给出一个 '-'，现有参数（如果有）将被删除。 在这种情况下，调用不带参数的 open 将导致错误。

在 close 之后，参数列表不会被删除，但如果 zsh/zftp 模块被卸载，它将被删除。

例如，

```
zftp params ftp.elsewhere.xx juser '?Password for juser: '
```

将存储主机 ftp.elsewhere.xx 和用户 juser，然后用给定的提示符提示用户输入相应的密码。

test

测试连接；如果服务器报告已关闭连接（可能是由于超时），则返回状态 2；如果没有打开连接，则返回状态 1；否则返回状态 0。 除了 $ZFTP_VERBOSE 机制打印

的信息或连接关闭时的错误信息外，`test` 子命令是静默(silent)的。 该测试没有网络开销。

只有使用 `select(2)` 或 `poll(2)` 系统调用的系统才支持该测试，否则将打印 'not supported on this system' 信息。

当连接打开时，`test` 子命令将在当前会话的任何其他子命令开始时自动调用。

`cd` *directory*

将远程目录更改为 *directory*。 同时更改 shell 变量 `ZFTP_PWD`。

`cdup`

将远程目录更改为目录树中高一级的目录。 请注意，`cd ..` 在非 UNIX 系统上也能正常工作。

`dir` [*arg* ...]

提供远程目录的（详细）列表。 *arg*s 直接传递给服务器。该命令的行为取决于实现情况，但 UNIX 服务器通常会将 *arg*s 解释为 `ls` 命令的参数，如果没有参数，则返回 '`ls -l`' 的结果。目录会以标准输出的方式列出。

`ls` [*arg* ...]

提供远程目录的（简短）列表。 在没有 *arg* 的情况下，会生成该目录下文件的原始列表，每行一个。 否则会受服务器实现的影响，表现与 `dir` 类似。

`type` [*type*]

将传输类型更改为 *type*，如果没有 *type* 则打印当前类型。 允许的值有： 'A' (ASCII), 'I'（图像，即二进制）或 'B'（'I' 的同义词）。

FTP 的默认传输方式是 ASCII。 不过，如果 `zftp` 发现远程主机是使用 8 位字节的 UNIX 机器，它将在 open 时自动切换为使用二进制传输文件。 这一点随后可以被覆盖。

传输类型只会在建立数据连接时传递给远程主机；该命令不涉及网络开销。

`ascii`

与 `type A` 相同。

`binary`

与 `type I` 相同。

`mode` [`S` | `B`]

将模式类型设置为流（S）或块（B）。 流模式是默认模式；块模式不被广泛支持。

`remote` *file* …
`local` [*file* …]

打印远程或本地文件的大小和最后修改时间。 如果列表中有多个项目，则首先打印文件名。 第一个数字是文件大小，第二个数字是文件的最后修改时间，格式为 CCYYMMDDhhmmSS，由格林尼治标准时间内的年、月、日、时、分、秒组成。 请注意，这种格式（包括长度）是有保证的，因此时间字符串可以通过 [[内置命令的 < 和 > 操作符直接进行比较，即使它们太长而无法用整数表示。

并非所有服务器都支持检索此信息的命令。 在这种情况下，`remote` 命令将不打印任何内容，并返回状态 2，而未找到文件则返回状态 1。

`local` 命令（但不包括 `remote`）可以不带参数，在这种情况下，信息来自于检查文件描述符 0。 这与 `put` 命令看到的文件相同，没有进一步的重定向。

`get` *file* …

从服务器读取所有 *file*s，将它们连接起来并发送到标准输出。

`put` *file* …

对于每个 *file*，从标准输入中读取一个文件，并以给定的名称将其发送到远程主机。

`append` *file* …

与 `put`相同，但如果远程 *file* 已经存在，数据将被附加到其中，而不是覆盖。

`getat` *file point*
`putat` *file point*
`appendat` *file point*

`get`、`put` 和 `append` 的版本，将在远程 *file* 中给定的 *point* 处开始传输。 这对附加到不完整的本地文件非常有用。 但请注意，服务器并不普遍支持这种功能（而且也不完全符合标准规定的行为）。

`delete` *file* …

删除服务器上的文件列表。

`mkdir` *directory*

在服务器上创建一个新目录 *directory*。

`rmdir` *directory*

删除服务器上的 *directory* 目录。

rename *old-name new-name*

将服务器上的文件 *old-name* 重命名为 *new-name*。

site *arg* ...

向服务器发送特定于主机的命令。只有在服务器指示使用时，您才可能需要它。

quote *arg* ...

向服务器发送原始 FTP 命令序列。 在此之前，您应该熟悉 RFC959 中定义的 FTP 命令集。 有用的命令可能包括 STAT 和 HELP。 请注意下面为变量 ZFTP_VERBOSE 所描述的返回信息的机制，特别是所有来自控制连接的信息都会被发送到标准错误中。

close
quit

关闭当前数据连接。 这将取消设置 shell 参数 ZFTP_HOST, ZFTP_PORT, ZFTP_IP, ZFTP_SYSTEM, ZFTP_USER, ZFTP_ACCOUNT, ZFTP_PWD, ZFTP_TYPE 和 ZFTP_MODE.

session [*sessname*]

允许同时使用多个 FTP 会话。 会话名称是任意字符串；默认会话名称为 'default'。 如果调用该命令时不带参数，则会列出所有当前会话；如果带参数，则会切换到名为 *sessname* 的现有会话，或创建一个与该名称相同的新会话。

每个会话都会记住连接的状态、特定于连接的 shell 参数集（与 close 说明中提到的连接关闭时取消设置的参数集相同），以及使用 params 子命令指定的任何用户参数。 如果切换到之前的会话，则会恢复这些值；如果切换到新会话，则会以与 zftp 刚加载时相同的方式初始化这些值。 当前会话的名称由参数 ZFTP_SESSION 提供。

rmsession [*sessname*]

删除会话；如果没有给出名称，则删除当前会话。 如果当前会话被删除，最早存在的会话将成为新的当前会话，否则当前会话不会改变。 如果被删除的会话是唯一的会话，则会创建一个名为 'default' 的新会话，并成为当前会话；请注意，即使被删除的会话也名为 'default'，这也是一个新会话。建议在使用 zftp 的后台命令仍处于活动状态时不要删除会话。

22.32.2 参数

zftp 使用以下 shell 参数。 目前它们都不是特殊参数。

ZFTP_TMOUT

整数。 返回错误前等待网络操作完成的时间（以秒为单位）。 如果模块加载时未设置，则默认值为 60。 如果值为 0，则关闭超时。 如果控制连接时出现超时，将关闭该连接。 如果超时太频繁，请使用较大的值。

ZFTP_IP

只读。 当前连接的 IP 地址，用点号表示。

ZFTP_HOST

只读。 当前远程服务器的主机名。 如果主机是以 IP 地址打开的，ZFTP_HOST 将包含 IP 地址；这样可以节省名称查询的开销，IP 地址在名称服务器不可用时最常用。

ZFTP_PORT

只读。 连接已打开的远程 TCP 端口的编号（即使该端口最初被指定为已命名的服务）。 通常是标准 FTP 端口 21。

如果您的系统不具备相应的转换功能，则会以网络字节顺序显示。 如果您的系统是 little-endian 格式，那么端口将由两个交换字节组成，标准端口将被报告为 5376。 在这种情况下，传给 `zftp open` 的数字端口也需要使用这种格式。

ZFTP_SYSTEM

只读。 服务器响应 FTP SYST 请求时返回的系统类型字符串。 最有趣的情况是以 "UNIX Type: L8" 开头的字符串，以确保与本地 UNIX 主机最大程度兼容。

ZFTP_TYPE

只读。 用于数据传输的类型，可以是 'A' 或 'I'。 使用 `type` 子命令可以更改。

ZFTP_USER

只读。 当前登录的用户名（如果有）。

ZFTP_ACCOUNT

只读。 当前用户的账户名（如果有）。 大多数服务器不需要账户名。

ZFTP_PWD

只读。 服务器上的当前目录。

ZFTP_CODE

只读。 服务器最后一次 FTP 回复的三位数代码字符串。 连接关闭后仍可读取，当前会话发生变化时也不会更改。

ZFTP_REPLY

只读。 服务器发送的最后一次回复的最后一行。 连接关闭后仍可读取，当前会话发生变化时也不会更改。

ZFTP_SESSION

只读。 当前 FTP 会话的名称；请参阅 session 子命令的说明。

ZFTP_PREFS

用于更改 zftp 行为的首选项字符串。每个首选项都是单个字符。 定义如下

P

被动(Passive)：尝试让远程服务器启动数据传输。 这种模式的效率略高于发送端口模式(sendport)。 如果字符串中之后出现字母 S，zftp 将在被动模式不可用时使用发送端口模式。

S

Sendport: 通过 FTP PORT 命令初始化传输。 如果该命令出现在字符串中的 P 之前，则永远不会尝试被动模式。

D

只使用最基本的 FTP 命令。 这将阻止变量 ZFTP_SYSTEM 和 ZFTP_PWD 被设置，并意味着所有连接都默认为 ASCII 类型。 在传输过程中，如果服务器不发送 ZFTP_SIZE（许多服务器都会发送），则可能会阻止设置 ZFTP_SIZE。

如果 ZFTP_PREFS 在 zftp 加载时未设置，则将默认设置为 'PS'，即在可用时使用被动模式，否则退回到发送端口模式。

ZFTP_VERBOSE

由 0 到 5（含 5）之间的数字组成的字符串，用于指定应打印哪些来自服务器的响应。所有响应都将转入标准错误。 如果字符串中出现 1 至 5 中的任何一个数字，则以该数字开头的回复代码的服务器原始回复将被打印到标准错误中。 根据 RFC959 的定义，三位回复代码的第一位对应于:

1.

肯定的初步答复。

2.

肯定的补全回复。

3.

肯定的中间回复。

4.

瞬时否定补全回复。

5.

永久性的否定补全回复。

值得注意的是，由于不明原因，强制终止连接的回复 'Service not available' 被归类为 421，即 '瞬时否定' ('transient negative')，这是对 '瞬时' ('transient')一词的有趣解释。

代码 0 比较特殊：它表示从服务器读取的多行回复中，除最后一行外的所有内容都将以处理过的格式打印到标准错误中。 按照惯例，服务器使用这种机制发送信息供用户阅读。 适当的回复代码,如果与相同的回复相匹配，则优先使用。

如果在加载 `zftp` 时未设置 `ZFTP_VERBOSE`，则会将其设置为默认值 `450`，即打印给用户的信息和所有错误信息。空字符串有效，表示不打印任何信息。

22.32.3 函数

`zftp_chpwd`

如果用户设置了该函数，那么每次服务器上的目录发生变化时，包括用户登录或连接关闭时，都会调用该函数。 在最后一种情况下，`$ZFTP_PWD` 将被取消设置，否则它将反映新的目录。

`zftp_progress`

如果用户设置了该函数，那么在 `get`、`put` 或 `append` 操作中，每次从主机接收到足够的数据时，都会调用该函数。 在 `get` 操作中，数据会被发送到标准输出，因此该函数必须写入标准错误或直接写入终端，**而不是** 写入标准输出。

在传输过程中调用它时，会设置以下附加 shell 参数：

`ZFTP_FILE`

正在传输的远程文件的名称。

`ZFTP_TRANSFER`

一个 `G` 表示 `get` 操作，一个 `P` 表示 `put` 操作。

`ZFTP_SIZE`

正在传输的完整文件的总大小：与 remote 和 local 子命令为特定文件提供的第一个值相同。 如果服务器无法为正在检索的远程文件提供该值，则不会设置该值。 如果输入来自管道，则该值可能不正确，而仅对应于一个满的管道缓冲区。

ZFTP_COUNT

目前已传输的数据量；介于 0 和 $ZFTP_SIZE 之间的数字(如果设置了 $ZFTP_SIZE)。该数字始终可用。

调用该函数时，首先会适当设置 ZFTP_TRANSFER，并将 ZFTP_COUNT 设置为零。传输完成后，如果函数希望进行整理，则会再次调用 ZFTP_TRANSFER 并将其设置为 GF 或 PF。 在其他情况下，该函数不会以 ZFTP_COUNT 的相同值被调用两次。

有时，进度表可能会造成干扰。 用户可自行决定是否应定义该函数，并在必要时使用 unfunction。

22.32.4 问题

不得在管道左侧打开连接，因为这发生在子 shell 中，文件信息不会在主 shell 中更新。如果类型或模式发生变化，或者在子 shell 中关闭连接，信息会返回，但变量不会更新，直到下一次调用 zftp 时才会更新。 子 shell 中的其他状态变化不会通过变量的变化反映出来（但在其他方面应该是无害的）。

当 zftp 命令在后台活动时删除会话，即使没有使用被删除的会话，也会产生意想不到的效果。 这是因为所有 shell 子进程都共享所有连接的状态信息，而删除会话会改变这些信息的排序。

在某些操作系统上，fork()后控制连接无效，因此子 shell、管道左侧或后台中的操作无法正常进行。 这可能是操作系统的一个错误。

22.33 zsh/zle 模块

zsh/zle 模块包含 Zsh 行编辑器。 参见 Zsh 行编辑器。

22.34 zsh/zleparameter 模块

zsh/zleparameter 模块定义了两个特殊参数，可用于访问 Zsh 行编辑器的内部信息（参见 Zsh 行编辑器）。

键映射

该数组包含当前定义的键映射名称。

小部件

这个关联数组包含每个小部件的一个条目。小部件的名称是键，值则给出了小部件的相关信息。对于内置小部件，它可以是字符串 'builtin' ；对于用户自定义的小部件，它可以是形式为 'user:*name*' 的字符串，其中 *name* 是实现小部件的 shell 函数的名称；对于补全小部件，它可以是形式为 'completion:*type*:*name*' 的字符串；如果小部件尚未完全定义，它可以是空值。 在倒数第二种情况下，*type* 是补全小部件模仿的内置小部件的名称，*name* 是实现补全小部件的 shell 函数的名称。

22.35 zsh/zprof 模块

加载 zsh/zprof 后，将对 shell 函数进行剖析。 可以使用本模块提供的 zprof 内置命令获取剖析结果。 除卸载模块外，无法关闭剖析。

zprof [-c]

如果不使用 -c 选项，zprof 会将剖析结果列到标准输出中。 格式与 gprof 等命令类似。

顶部有一个摘要，列出了至少被调用一次的所有函数。 该摘要按每个函数所用时间的递减顺序排列。 各行依次包含函数的编号（在列表的其他部分中使用后缀 '[*num*]' 形式），然后是函数的调用次数。 接下来的三列列出了该函数及其后代函数的耗时（以毫秒为单位）、该函数及其后代函数每次调用的平均耗时（以毫秒为单位）以及该函数及其后代函数使用的所有 shell 函数的耗时百分比。 下面三列提供了相同的信息，但只计算函数本身所用的时间。 最后一列显示函数名称。

在摘要之后，列出了调用的每个函数的详细信息，并按每个函数及其子函数所用时间的递减顺序排列。 每个条目都包括调用所述函数的函数、函数本身以及从函数调用的函数的描述。 函数本身的描述格式与摘要中的格式相同（显示的信息也相同）。 其他行开头不显示函数的编号，并将其函数名称缩进，以便于从周围的行中区分出显示本节所述函数的行。

这种情况下显示的信息与摘要中的信息几乎相同，但只涉及所显示的调用层次结构。 例如，对于调用函数，显示总运行时间的列仅列出从该特定调用函数调用描述的函数及其子函数时所花费的时间。 同样，对于被调用函数，该列列出了在被调用函数及其后代函数中总共花费的时间，仅针对从所描述的函数中调用它时的时间。

在这种情况下，显示函数调用次数的列还会显示一条斜线，然后是被调用函数的总调用次数。

只要 zsh/zprof 模块已加载，就会进行剖析，多次调用 zprof 内置命令将显示模块加载后的调用时间和次数。 使用 -c 选项后，zprof 内置命令将重置内部计数器，不再显示列表。

22.36 zsh/zpty 模块

`zsh/zpty` 模块提供了一个内置程序：

`zpty` [`-e`] [`-b`] *name* [*arg* ...]

> *name* 后面的参数会用空格连接起来，然后作为一条命令执行，就像传递给 `eval` 内置函数一样。 命令将在新分配的伪终端下运行；这对于运行那些期望在交互式环境下运行的非交互式命令非常有用。 *name* 并非命令的一部分，而是用于在以后调用 `zpty` 时引用该命令。
>
> 使用 `-e` 选项时，伪终端会设置为使输入字符产生回显。
>
> 如果使用 `-b` 选项，伪终端的输入和输出都将是非阻塞的。
>
> shell 参数 `REPLY` 被设置为分配给伪终端主控端的文件描述符。 这样就可以使用 ZLE 描述符处理程序监控终端（参见 Zle 内置命令），或使用 `sysread` 和 `syswrite` 操作终端（参见 zsh/system 模块）。 **警告**： 不建议使用 `sysread` 和 `syswrite`；除非您非常清楚自己在做什么，否则请使用 `zpty -r` 和 `zpty -w`。

`zpty -d` [*name* ...]

> 第二种形式带有 `-d` 选项，通过提供 *name*s 的列表来删除之前启动的命令。 如果没有给出 *name*，则删除所有命令。 删除命令会导致向相应进程发送 HUP 信号。

`zpty -w` [`-n`] *name* [*string* ...]

> `-w` 选项可用于将给定的 *string* 作为输入发送到命令 *name*。如果 **未** 给出 `-n` 选项，则在末尾添加一个换行符。
>
> 如果没有提供 *string*，标准输入会被复制到伪终端；如果伪终端是非阻塞的，可能会在复制完整输入之前停止。 精确输入总是被复制：`-n` 选项不适用。
>
> 请注意，伪终端下的命令会将这些输入视为键入的输入，因此在发送特殊的 tty 驱动程序字符（如字擦除、行删除和文件结束）时要小心。

`zpty -r` [`-mt`] *name* [*param* [*pattern*]]

> `-r` 选项可用于读取 *name* 命令的输出。 如果只有 *name* 参数，读取的输出将被复制到标准输出。 除非伪终端是非阻塞的，否则复制将持续到伪终端下的命令退出为止；如果是非阻塞的，则只复制立即可用的输出。 如果有任何输出被复制，返回状态为零。
>
> 如果同时给出 *param* 参数，则最多读取一行并存储在名为 *param* 的参数中。 如果伪终端是非阻塞的，则可以读取少于一行的内容。 如果 *param* 中至少存储了一个字符，则返回状态为零。

如果同时给出 *pattern*，即使在非阻塞情况下，也会读取输出，直到读取的整个字符串与 *pattern* 匹配为止。 如果读取的字符串与模式匹配，或者命令已退出但仍能读取至少一个字符，则返回状态为零。 如果存在选项 `-m`，只有当模式匹配时，返回状态才为零。 截至本文撰写时，这种方式最多可消耗 1 兆字节的输出；如果读取了整整 1 兆字节，但没有匹配到模式，则返回状态为非零。

在所有情况下，如果没有读取到任何内容，返回状态均为非零，如果是因为命令已执行完毕，返回状态则为 2。

如果 `-r` 选项与 `-t` 选项结合使用，`zpty` 会在尝试读取之前测试输出是否可用。 如果输出不可用，`zpty` 会立即返回状态 1。 与 *pattern* 一起使用时，轮询失败时的行为与命令退出时类似：如果即使模式匹配失败，仍能读取至少一个字符，则返回值为 0。

`zpty -t` *name*

不带 `-r` 选项的 `-t` 选项可用于测试 *name* 命令是否仍在运行。 如果命令正在运行，则返回零状态，否则返回非零值。

`zpty [ -L ]`

最后一种形式不带任何参数，用于列出当前定义的命令。 如果给定了 `-L` 选项，则将以调用 `zpty` 内置程序的形式完成。

22.37 zsh/zselect 模块

`zsh/zselect` 模块提供了一条内置命令：

`zselect [ -rwe ] [ -t` *timeout* `] [ -a` *array* `] [ -A` *assoc* `] [` *fd ...* `]`

`zselect` 内置命令是 'select' 系统调用的前端，它会一直阻塞，直到文件描述符可以读写或出现错误，并可选择超时。 如果p 这在系统中不可用，该命令将打印错误信息并返回状态 2（正常错误返回状态 1）。 更多信息，请参阅系统中的 select(3) 文档。 请注意，该命令与同名的 shell 内置命令没有任何联系。

参数和选项可以任意顺序混合。 非选项参数是文件描述符，必须是十进制整数。 默认情况下，文件描述符将接受读取测试，即 `zselect` 将在可以从文件描述符读取数据时返回，或者更准确地说，在从文件描述符读取操作不会阻塞时返回。 在 `-r`、`-w` 和 `-e` 之后，将对给定的文件描述符进行读、写或错误测试。 这些选项和任意文件描述符列表可以任意顺序给出。

(在许多实现 select 系统调用的文档中，'error condition' 的存在并没有明确定义。根据 POSIX 规范的最新版本，它实际上是一个 ***exception*** 条件，其中唯一的标准示例是通过套接字接收到的带外(out-of-band)数据。 因此，zsh 用户不太可能发现 `-e` 选项有什么用）。

选项 '-t *timeout*' 以百分之一秒为单位指定超时时间。 超时时间可以为零，在这种情况下，文件描述符将被轮询，而 zselect 将立即返回。 可以用没有文件描述符和非零超时的 zselect 来 'sleep' 的更精细替代；但需要注意的是，超时时的返回状态总是 1。

选项 '-a *array*' 表示应设置 *array* 以指示已就绪的文件描述符。 如果未给出该选项，则将使用数组 reply。 数组包含的字符串与 zselect 的参数类似。 例如,

```
zselect -t 0 -r 0 -w 1
```

可能会立即返回状态 0 和包含 '-r 0 -w 1' 的 $reply，以表明两个文件描述符都已准备好执行请求的操作。

选项 '-A *assoc*' 表示应设置关联数组 *assoc* 以指示已准备就绪的文件描述符。 该选项会覆盖 -a 选项，也不会修改 reply。 assoc 的键是文件描述符，对应的值是任意字符 'rwe' ，用于指示条件。

如果某些文件描述符已准备好供读取，则命令返回状态 0。 如果操作超时，或超时时间为 0 但没有文件描述符准备就绪，或出现错误，则返回状态 1，数组不会被设置（也不会以任何方式修改）。 如果选择操作出错，将打印相应的错误信息。

22.38 zsh/zutil 模块

zsh/zutil 模块只增加一些内置命令:

zstyle [-L [*metapattern* [*style*]]]
zstyle [-e | - | --] *pattern style string ...*
zstyle -d [*pattern* [*style ...*]]
zstyle -g *name* [*pattern* [*style*]]
zstyle -{a|b|s} *context style name* [*sep*]
zstyle -{T|t} *context style* [*string ...*]
zstyle -m *context style pattern*

该内置命令用于定义和查找样式。 样式由名称和值组成，其中值由任意数量的字符串组成。 它们与模式一起存储，通过提供一个字符串（称为 '***context***' ）与模式匹配进行查找。 将返回为最匹配的特定模式存储的定义。

如果一个模式包含更多的成分（用冒号分隔的子串），或者如果成分的模式更具体，则该模式被认为比另一个模式更具体，其中简单字符串被认为比模式更具体，而复杂模式被认为比模式 '*' 更具体。 模式中的 '*' 将匹配上下文中的零个或多个字符；冒号在这方面不作特殊处理。 如果两个模式的特定性相同，则先定义的模式优先。

举例

例如，一个虚构的 'weather' 插件可能会在其文档中说明，它会在 ':weather:*continent*:*day-of-the-week*:*phase-of-the-moon*' 上下文中查找 preferred-precipitation 样式。 据此，您可以在 zshrc 中设置以下内容：

```
zstyle ':weather:europe:*' preferred-precipitation rain
zstyle ':weather:*:Sunday:*' preferred-precipitation snow
```

然后，插件会在内部（under the hood)运行一条命令，例如

```
zstyle -s ":weather:${continent}:${day_of_week}:${moon_phase}" pr
```

为了在标量变量 $REPLY 中获取您的偏好。 在周日，$REPLY 将被设置为 'snow'；在欧洲，它将被设置为 'rain'；在欧洲的周日，它将再次被设置为 'snow'，因为模式 ':weather:europe:*' 和 ':weather:*:Sunday:*' 都与 zstyle -s 的 *context* 参数相匹配，具有相同的特异性，而且后者更具特异性（因为它有更多以冒号分隔的成分）。

用法

对模式进行操作的形式如下。

zstyle [-L [*metapattern* [*style*]]]

不带参数，列出样式定义。 样式按字母顺序显示，模式按 zstyle 将测试它们的顺序显示。

如果给出 -L 选项，则将以调用 zstyle 的形式列出。 可选的第一个参数 *metapattern* 是一个模式，将与定义样式时作为 *pattern* 提供的字符串进行匹配。 注意：这意味着，例如，'zstyle -L ":completion:*"' 将匹配以':completion:' 开头的任何提供的模式，而不仅仅是 ":completion:*"：使用 ':completion:*' 来匹配。 可选的第二个参数将输出限制为特定的 *style*（而不是模式）。 -L 与其他选项不兼容。

zstyle [- | -- | -e] *pattern style string ...*

为 *pattern* 定义给定的 *style* 并将 *string*s 作为值。 如果给定了 -e 选项，则 *string*s 将被连接（用空格分隔），生成的字符串将在查找样式时进行评估（与 eval 内置命令的评估方式相同）。 在这种情况下，必须指定参数 'reply'，以设置求值后返回的字符串。 在求值之前，reply 将被取消设置，如果求值后仍未设置，则样式将被视为未设置。

zstyle -d [*pattern* [*style ...*]]

删除样式定义。在不带参数的情况下，所有定义都会被删除；在带 *pattern* 的情况下，该样式的所有定义都会被删除；如果给定了任何 *style*，则只删除 *pattern* 的那些样式。

zstyle -g *name* [*pattern* [*style*]]

读取样式定义。*name* 用作存储结果的数组名称。在没有其他参数的情况下，将返回所有已定义的样式。使用 *pattern* 时，将返回为该模式定义的样式；同时使用 *pattern* 和 *style* 时，将返回该组合的值字符串。

其他形式可用于查找或测试特定环境下的样式。

`zstyle -s` *context style name* [*sep*]

参数 *name* 将被设置为以字符串形式解释的样式值。 如果样式值包含多个字符串，则在字符串之间用空格（或 *sep* 字符串（如果已给出））连接起来。

如果设置了样式，则返回 `0`，否则返回 1。

`zstyle -b` *context style name*

如果参数值只有一个字符串，且该字符串等于‘`yes`’, ‘`true`’, ‘on’ 或 ‘1’ 中的一个，则参数值将以布尔形式保存在 *name* 中，即保存为字符串 ‘`yes`’。如果数值是其他字符串或有多个字符串，参数将被设置为 ‘no’。

如果 *name* 设置为 ‘`yes`’，则返回 `0`，否则返回 1。

`zstyle -a` *context style name*

值以数组形式存储在 *name* 中。如果 *name* 被声明为关联数组，则第一个、第三个等字符串将用作键，其他字符串将用作值。

如果设置了样式，则返回 `0`，否则返回 1。

`zstyle -t` *context style* [*string ...*]
`zstyle -T` *context style* [*string ...*]

测试样式的值，即 `-t` 选项只返回状态（设置 `$?`）。 在不包含任何 *string* 的情况下，如果样式为至少一个匹配模式定义，其值中只有一个字符串，且等于 ‘`true`’, ‘`yes`’, ‘on’ 或 ‘1’ 中的一个，则返回状态为零。如果给出任何 *string*s，则只有当 *string*s 中至少有一个字符串等于值中的至少一个字符串时，状态才为零。如果样式已定义但不匹配，则返回状态为 1。如果未定义样式，则返回状态为 2。

`-T` 选项会像 `-t` 一样测试样式的值，但如果没有为任何匹配模式定义样式，则返回状态 0（而不是 2）。

`zstyle -m` *context style pattern*

匹配一个值。如果 *pattern* 至少匹配值中的一个字符串，则返回状态 0。

`zformat -f` *param format spec ...*
`zformat -F` *param format spec ...*
`zformat -a` *array sep spec ...*

该内置程序提供不同的格式化形式。第一种形式通过 `-f` 选项选择。在这种情况下，*format* 字符串将被修改，用 *spec* 中的字符串替换以百分号开头的序列。 每个 *spec* 都应采用 '*char*:*string*' 的形式，这样 *format* 中每次出现的 '%*char*' 序列都会被 *string* 所替换。 在 '%' 和 '*char*' 之间，'%' 序列还可以包含可选的最小和最大字段宽度说明，其形式为 '%*min*.*max*c'，即首先给出最小字段宽度，如果使用最大字段宽度，则必须在其前加一个点。 如果 *string* 短于要求的宽度，指定最小字段宽度会使结果向右填充空格。 通过给出负的最小字段宽度，可以实现向左填充。 如果指定了最大字段宽度，*string* 将在该字符数之后被截断。 处理完指定 *spec* 的所有 '%' 序列后，生成的字符串将存储在参数 *param* 中。

%转义也能理解提示符所用的三元表达式形式。 % 后面是一个 '(' ，然后是一个普通的格式规范字符，如上所述。 在 '(' 之前或之后可能有一组数字；这些数字指定一个测试数，默认为零。 也允许使用负数。 格式指定符后是一个任意分隔符，之后是一段 'true' 文本、分隔符、一段 'false' 文本和一个闭括号。 因此，完整的表达式（不含数字）看起来就像 '%(*X*.*text1*.*text2*)'，只不过 '.' 字符是任意的。 *char*:*string* 表达式中格式指定符的值将作为数学表达式进行运算，并与测试数进行比较。 如果两者相同，则输出 *text1*，否则输出 *text2*。 *text2* 中的括号可以转义为 %)。 *text1* 或 *text2* 都可能包含嵌套的 % 转义。

例如：

```
zformat -f REPLY "The answer is '%3(c.yes.no)'." c:3
```

由于格式指定符 `c` 的值为 3，与三元表达式的数字参数一致，因此会向 `REPLY` 输出 "The answer is 'yes'."。

使用 `-F` 代替 `-f`，三元表达式会根据格式指定符是否存在且非空来选择 'true' 或 'false' 文本。 测试数表示格式指定符中给定值的最小宽度。负数则相反，因此测试值是否超过最大宽度。

使用 `-a` 选项的形式可用于对齐字符串。 这里，*spec*s 的形式是 '*left*:*right*'，其中 '*left*' 和 '*right*' 是任意字符串。 修改这些字符串的方法是：用 *sep* 字符串替换冒号，并在 *left* 字符串右侧填充空格，这样，如果字符串打印在彼此下方，结果中的 *sep* 字符串（以及后面的 *right* 字符串）都会对齐。 所有不带冒号的字符串都保持不变，所有带空 *right* 字符串的字符串都去掉了后面的冒号。 在这两种情况下，字符串的长度不会用来决定其他字符串的对齐方式。 *left* 字符串中的冒号可以用反斜线转义。 结果字符串将存储在 *array* 中。

`zregexparse`

它实现了 `_regex_arguments` 函数的一些内部功能。

`zparseopts` [`-D` `-E` `-F` `-K` `-M`] [`-a` *array*] [`-A` *assoc*] [`-`] *spec* ...

该内置函数简化了位置参数（即 `$*` 给出的参数集）中选项的解析。 每个 *spec* 描述一个选项，其形式必须是 '*opt*[=*array*]'。 如果在位置参数中找到 *opt* 所描述的选

项，则会将其复制到 -a 选项指定的 *array* 中；如果给出了可选项 '=*array*' ，则会将其复制到该数组中,这里数组应声明为普通数组，而绝不能声明为关联数组。 。

需要注意的是，除非使用 -a 或 -A 选项之一，否则在给出任何 *spec* 的同时不给出 '=*array*' 是错误的。

除非给出 -E 选项，否则解析会在第一个没有被 *specs* 描述的字符串处停止。 即使使用 -E，解析也总是在位置参数等于 '-' 或 '--' 时停止。另请参阅 -F。

opt 说明必须是下列内容之一。 任何特殊字符都可以出现在选项名称中，但必须在前面加上反斜杠。

name
name+

> *name*是不带前导 '-' 的选项名称。 要指定 GNU 风格的长选项，*name* 中必须包含两个前导 '-' 中的一个；例如，'--file' 选项由 *name* 中的 '-file' 表示。
>
> 如果在 *name* 后面出现 '+' ，则每次在位置参数中找到该选项时，都会将其追加到 *array* 中；如果没有 '+' ，则只保留选项的 **最后一欠** 出现。
>
> 如果使用了其中一种形式，则该选项不带参数，因此，如果下一个位置参数也不是以 '-' 开头，解析就会停止（除非使用了 -E 选项）。

name:
name:-
name::

> 如果给出一个或两个冒号，则该选项包含一个参数；如果给出一个冒号，则该参数为必选参数，如果给出两个冒号，则该参数为可选参数。 参数会追加到选项本身之后的 *array* 中。
>
> 可选参数与选项名称放在同一个数组元素中（注意，这使得作为参数的空字符串无法区分）。 除非使用了 ':-' 形式，否则必选参数会作为一个单独的元素加入，在这种情况下，参数会被放入同一个元素。
>
> 在 *name* 和第一个冒号之间可能会出现上文所述的 '+' 。

在任何情况下，选项参数都必须紧接着选项出现在同一位置参数中，或者出现在下一个参数中。即使是可选参数，也可以出现在下一个参数中，除非它以 '-' 开头。在 GNU 风格的参数解析器中，对 '=' 没有特殊处理；给定 *spec* '-foo:'，位置参数 '--foo=bar' 会被解析为参数为'=bar' 的 '--foo' 。

当两个不带参数的选项名称重叠时，最长的选项胜出，因此对于 *specs*，'-foo -foobar'（例如）的解析是明确无误的。然而，由于上述对选项参数的处理，当至

少有一个重叠的 *spec* 包含一个参数时，可能会产生歧义，例如在 '-foo: -foobar' 中。在这种情况下，最后一个匹配的 *spec* 将胜出。

zparseopts 的选项本身无法堆叠（stacked），因为例如，堆栈（选项组合）'-DEK' 与 GNU 风格长选项 '--DEK' 的 *spec* 无法区分。 zparseopts 本身的选项有

-a *array*

如上所述，它命名了存储已识别选项的默认数组。

-A *assoc*

如果给定了这个值，选项及其值也会被放入一个关联数组中，其中选项名称为键，参数（如果有）为值。

-D

如果给定了这个选项，所有找到的选项都会从调用 shell 或 shell 函数的位置参数中移除，直至但不包括 *spec*s 未描述的参数。 如果第一个参数是 '-' 或 '--'，也会被移除。 这与使用 shift 内置函数类似。

-E

这将改变解析规则，使其 **不会** 终止于第一个未被 *spec* 描述的字符串。 它可以用来测试或（如果与 -D 一起使用）提取选项及其参数，而忽略位置参数中的所有其他选项和参数。 如上所述，解析仍会在第一个 *spec* 未描述的 '-' 或 '--' 处停止，但与 -D 一起使用时不会删除。

-F

如果给定了该选项，zparseopts 会在第一个 *spec*s 未描述的类选项参数处立即停止，打印错误信息并返回状态 1。 不执行移除（-D）和提取（-E）操作，也不更新选项数组。 这将对给定的选项进行基本验证。

需要注意的是，无论是否使用选项，如果在位置参数中出现一个不带必要参数的选项，解析过程都会中止，并如上所述返回错误信息。

-K

有了这个选项，当使用 -a 选项和 '=*array*' 形式的数组都没有使用 *spec*s 时，这些数组将保持不变。 否则，在使用任何 *spec*s 时，整个数组都会被替换。使用 -A 选项指定的关联数组的单个元素会被 -K 保留。 这样就可以在调用 zparseopts 之前为数组赋默认值。

-M

这就改变了赋值规则，实现了等价选项名称之间的映射。 如果任何 *spec* 使用了 '=*array*' 形式，则字符串 *array* 会被解释为另一个 *spec* 的名称，用于选择

在何处存储值。 如果找不到其他 *spec*，值将照常存储。 这只会改变值的存储方式，而不会改变 $* 的解析方式，因此，如果 '*name*+' 指定符的使用不一致，结果可能无法预测。

例如,

```
set -- -a -bx -c y -cz baz -cend
zparseopts a=foo b:=bar c+:=bar
```

将产生以下效果

```
foo=(-a)
bar=(-b x -c y -c z)
```

从 'baz' 开始的参数将不会被使用。

以 -E 选项为例，请看

```
set -- -a x -b y -c z arg1 arg2
zparseopts -E -D b:=bar
```

将产生以下效果

```
bar=(-b y)
set -- -a x -c z arg1 arg2
```

即从位置参数中提取选项 -b 及其参数，并将其放入数组 bar 中。

-M 选项可以这样使用：

```
set -- -a -bx -c y -cz baz -cend
zparseopts -A bar -M a=foo b+: c:=b
```

要产生效果

```
foo=(-a)
bar=(-a '' -b xyz)
```

---

23 日历函数系统

23.1 说明

shell 提供了一系列函数来替代和增强传统的 Unix calendar 程序，该程序会在即将发生或未来将发生的事件中向用户发出警告，这些事件的详细信息存储在文本文件中（通常是用户主目录中的 calendar）。 这里提供的版本包含了一种在事件即将发生时提醒用户的机制。

此外，还提供了可在 glob 限定符中使用的 age, `before` 和 `after` 函数；它们允许根据文件的修改时间来选择文件。

`calendar`文件的格式以及其中使用的日期和age函数中使用的日期首先进行描述，然后描述可以调用以检查和修改`calendar`文件的函数。

此处的函数依赖于 `zsh/datetime` 模块的可用性，该模块通常与 shell 一起安装。 库函数 `strptime()` 必须可用；在大多数最新的操作系统中都存在。

23.2 文件和日期格式

23.2.1 日历文件格式

日历文件默认为 `~/calendar`。 可通过 `calendar-file` 样式进行配置，参见 样式。
日历文件的基本格式由一系列独立的行组成，没有缩进，每行包括日期和时间说明，以及事件描述。

根据 Emacs 日历模式的语法，该格式支持多种增强功能。 缩进行表示续行，继续前一行对事件的描述（注意日期不能以这种方式续写）。 为兼容起见，开头的逗号 (&) 将被忽略。

如果缩进行的第一个非空格字符是 #，则该行不会与日历条目一起显示，但仍会被扫描以获取信息。 这可用于隐藏对日历系统有用但对用户无用的信息，例如 `calendar_add` 使用的唯一标识符。

没有说明的日期可能指不同时间的多个后续事件,这样的Emacs扩展是不支持的。

除非更改了 `done-file` 样式，否则任何已处理的事件都会被追加到与日历文件同名的文件中，并带有后缀 `.done`，因此默认情况下是 `~/calendar.done`。

下面是一个例子。

23.2.2 日期格式

日期和时间的格式设计灵活，不会产生歧义。（'date' 日期和 'time' 时间这两个词在下面的文档中都有使用；除非有特别说明，否则这意味着一个字符串可能同时包含日期和时间说明）。 请注意，没有本地化支持；月名和日名必须为英文，且分隔符是固定的。 匹配不区分大小写，只有名称的前三个字母才有意义，但作为特例，以 "month" 开头的形式不会匹配 "Monday"。 此外，不处理时区；所有时间均假定为本地时间。

建议用户不要去探究系统的复杂性，而是找到一种自然的日期格式并坚持使用。 这将避免意想不到的效果。 应注意各种关键事实。

- 特别要注意的是，当月份为数字时， *month*/*day*/*year* 和 *day*/*month*/*year* 之间的混淆；应尽可能避免使用这些格式。 有许多替代格式可供选择。

- 为避免混淆，年份必须完整填写，且只匹配 1900 年至 2099 年（含 2099 年）的年份。

下面给出了一些明显的例子；如果用户在这里找到了自己喜欢的格式，并且不受风格变化的影响，可以跳过完整的描述。 由于日期和时间是分开匹配的（即使时间可能包含在日期中），只要分隔符（空格、冒号、逗号）明确，任何日期格式都可以与任何时间格式混合使用。

```
2007/04/03 13:13
2007/04/03:13:13
2007/04/03 1:13 pm
3rd April 2007, 13:13
April 3rd 2007 1:13 p.m.
Apr 3, 2007 13:13
Tue Apr 03 13:13:00 2007
13:13 2007/apr/3
```

更详细的规则如下。

时间在日期之前进行解析和提取。 它们必须使用冒号来分隔小时和分钟，如果有秒，则允许在秒前加点。 这就将时间格式限制为以下几种：

- *HH*:*MM*[:*SS*[.*FFFFF*]] [am|pm|a.m.|p.m.]
- *HH*:*MM*.*SS*[.*FFFFF*] [am|pm|a.m.|p.m.]

在这里，方括号表示可选要素，可能还有替代要素。 零点几秒是可以识别的，但会被忽略。 对于绝对时间（`calendar` 文件和 `age`、`before` 和 `after` 函数要求的正常格式），日期是必须的，但一天中的时间不是；返回的时间是日期的起始时间。 允许一种变体：如果存在 `a.m.` 或 `p.m.` 或它们的变体之一，则允许使用不带分钟的小时，例如 `3 p.m.`。

不处理时区，但如果在时间规范后匹配到时区，则会将其移除，以便解析周围的日期。 只有在时区格式不太特殊的情况下，才会发生这种情况。 以下是可以理解的格式示例：

```
+0100
GMT
GMT-7
CET+1CDT
```

时区名称中任何非数字部分都必须有三个大写字母。

日期在 *DD*/*MM*/*YYYY* 和 *MM*/*DD*/*YYYY* 之间存在歧义。 建议在使用纯数字日期时避免使用这种形式，但应使用序数词，例如：`3rd/04/2007` 可以消除歧义，因为序数词总是被解析为该月的日期。 年份必须是四位数（前两位必须是 19 或 20）； `03/04/08` 不被识别。 其他数字可以有前导零，但不是必需的。 可处理下列情况：

- *YYYY*/*MM*/*DD*

- *YYYY*-*MM*-*DD*
- *YYYY*/*MNM*/*DD*
- *YYYY*-*MNM*-*DD*
- *DD*[th|st|rd] *MNM*[,] [*YYYY*]
- *MNM DD*[th|st|rd][,] [*YYYY*]
- *DD*[th|st|rd]/*MM*[,] *YYYY*
- *DD*[th|st|rd]/*MM*/*YYYY*
- *MM*/*DD*[th|st|rd][,] *YYYY*
- *MM*/*DD*[th|st|rd]/*YYYY*

在这里，*MNM* 至少是月份名称的前三个字母，并且匹配不区分大小写。 月名的其余部分可能会出现，但其内容无关紧要，因此 janissary, febrile, martial, apricot, maybe, junta 等都可以轻松处理。

如果年份显示为可选项，则假定为当年。 这种情况只有两种： `Jun 20` 或 `14 September`（除了某些英语中的 "the"，目前还不支持 "the"）。 当然，这样的日期将来会变得模棱两可，因此最好避免使用。

时间可以带冒号跟在日期后面，例如 `1965/07/12:09:45`；这是为了提供一种没有空格的格式。 逗号和空格是允许的，例如 `1965/07/12, 09:45`。 目前不检查这些分隔符的顺序，因此 `1965/07/12, : ,09:45` 等不合逻辑的格式也会被匹配。 为简单起见，上面的列表中不显示此类变化。 否则，只有当时间与日期之间只有空白，或者时间被嵌入日期中时，时间才会被识别为与日期相关联。

通常不会扫描星期几，但如果星期几只出现在日期模式的起始位置，则会被忽略。 但是，在需要指定相对于今天的日期时，可以给出没有其他日期说明的星期几。 日期被假定为今天或过去一周内的日期。 同样， `yesterday`, `today` 和 `tomorrow` 也会被处理。 所有匹配都不区分大小写。 因此，如果今天是星期一，那么 `Sunday` 相当于 `yesterday`，`Monday` 相当于 `today`，而 `Tuesday` 则给出六天前的日期。 这种格式在日历文件中一般用处不大。 这种格式的日期可以与时间规范结合使用，例如 `Tomorrow, 8 p.m.`。

例如，标准日期格式：

```
Fri Aug 18 17:00:48 BST 2006
```

处理方法是匹配 *HH*:*MM*:*SS* 并将其与匹配的（但未使用的）时区一起删除。 这样就剩下以下内容：

```
Fri Aug 18 2006
```

`Fri` 将被忽略，其余内容将根据标准规则进行匹配。

23.2.3 相关时间格式

某些地方会处理相对时间。 这里不允许使用日期，而是允许使用各种支持的时段组合，以及可选的时间。 时间段必须按照从最重要到最不重要的顺序排列。

在某些情况下，如果有一个锚定日期，就可以进行更精确的计算：月份或年份的偏移量会选择正确的日期，而不是四舍五入，也可以选择一个月中的某一天作为 '(1st Friday)' 等，下文将详细介绍。

锚点在以下情况下可用。 如果向函数 `calendar` 传递了一个或两个时间，当结束时间是相对的（即使开始时间是隐含的），开始时间将作为结束时间的锚点。 在检查日历文件时，当通过 `WARN` 关键字明确给出警告时间时，被检查的计划事件将锚定警告时间；同样，当通过 `RPT` 关键字给出重复周期时，计划事件将锚定重复周期，因此 `RPT 2 months, 3rd Thursday` 这样的规范可以得到正确处理。 最后，`calendar_scandate` 的 `-R` 参数直接为相对计算提供了锚点。

所处理的时段，可能的缩写是:

年

> `years, yrs, ys, year, yr, y, yearly`. 一年是 365.25 天，除非有锚。

月

> `months, mons, mnths, mths, month, mon, mnth, mth, monthly`。 请注意，`m、ms、mn、mns` 是模棱两可的，**不**会被处理。 除非有锚点，否则 "月 "指的是 30 天，而不是日历月。

星期

> `weeks, wks, ws, week, wk, w, weekly`

日

> `days, dys, ds, day, dy, d, daily`

时

> `hours, hrs, hs, hour, hr, h, hourly`

分

> `minutes, mins, minute, min`, but ***not*** `m, ms, mn or mns`

秒

> `seconds, secs, ss, second, sec, s`

数字之间的空格是可选的，但项目之间需要空格，不过也可以使用逗号（带或不带空格）。

从 yearly 到 hourly 的形式允许省略数字；数字被假定为 1。 例如，1 d 和 daily 是等价的。 需要注意的是，将这些形式与复数一起使用会造成混淆；2 yearly 与 2 years 一样， **不是** 每年两次，因此建议只使用不带数字的形式。

当存在锚点时间时，还可以扩展成第 *n* 个 *some*day 的形式处理常规事件。 这种规范必须紧跟在任何年月规范之后，但在任何日期的时间之前，并且必须以 *n*(th|st|rd) *day* 的形式出现，例如 1st Tuesday 或 3rd Monday。 与在其他地方一样，日的匹配不区分大小写，必须是英文，而且只有前三个字母是重要的，以 'month' 开头的形式不匹配 'Monday'。不会对生成的日期进行净化处理； 如果在一个月内出现过多的日期，则会将日期推到下一个月（但显然会保留正确的星期几）。

这里是一些示例：

```
30 years 3 months 4 days 3:42:41
14 days 5 hours
Monthly, 3rd Thursday
4d,10hr
```

23.2.4 举例

下面是一个日历文件示例。 如上文所建议，它使用了统一的日期格式。

```
Feb 1, 2006 14:30 Pointless bureaucratic meeting
Mar 27, 2006 11:00 Mutual recrimination and finger pointing
  Bring water pistol and waterproofs
Mar 31, 2006 14:00 Very serious managerial pontification
  # UID 12C7878A9A50
Apr 10, 2006 13:30 Even more pointless blame assignment exercise WARN
May 18, 2006 16:00 Regular moaning session RPT monthly, 3rd Thursday
```

第二个条目有一个续行。 第三个条目有一个续行，在显示条目时不会显示，但 calendar_add 函数在更新事件时会使用该唯一标识符。 第四个条目会在事件发生前 30 分钟发出警告（以便您做好适当的准备）。 第五个条目会在一个月后的第三个星期四重复出现，即 2006 年 6 月 15 日的同一时间。

23.3 用户函数

本节介绍用户可直接调用的函数。 第一部分介绍与用户日历相关的函数；第二部分介绍 glob 限定符中的使用。

23.3.1 日历系统函数

`calendar` [`-abdDsv`] [`-C` *calfile*] [`-n` *num*] [`-S` *showprog*]
[[*start*] *end*]
`calendar` `-r` [`-abdDrsv`] [`-C` *calfile*] [`-n` *num*] [`-S` *showprog*]
[*start*]

显示日历中的事件。

如果没有参数，则显示从今天开始到今天之后下一个工作日结束的事件。 换句话说，如果今天是周五、周六或周日，则显示到下周一结束，否则显示今天和明天。

如果给定 *end*，则显示从今天开始到给定时间和日期的事件，给定时间和日期的格式如上一节所述。 需要注意的是，如果是日期，则时间假定为日期开始时的午夜，因此实际上会显示给定日期之前的所有事件。

end 可以以 + 开始，在这种情况下，说明的其余部分是相对时间格式，如上一节所述，表示从开始时间算起的时间范围。

如果同时给出 *start*，则显示从该时间和日期开始的事件。 可以使用 now 表示当前时间。

要在事件到期时发出警报，请在 `~/.zshrc` 文件中加入 `calendar -s`。

选项:

`-a`

显示日历中的所有项目，与 `start` 和 end 无关。

`-b`

简介：不显示续行（即日期/时间行之后的缩进行），只显示第一行。

`-B` *lines*

简介：最多显示日历条目的前 *lines* 行。‘`-B 1`’ 等同于 ‘`-b`’。

`-C` *calfile*

明确指定日历文件，而不是 `calendar-file` 样式或默认的 `~/calendar` 值。

`-d`

将日历文件中的事件移至 "已完成 "文件，如 `done-file` 样式或默认值（即附加 .done 的日历文件）所示。 该选项由 `-s` 选项隐含。

`-D`

关闭 `-d` 选项，即使同时存在 `-s` 选项。

`-n` *num*, `-`*num*

无论 `start` 和 end 如何，至少显示 *num* 个事件(如果日历文件中有)。

`-r`

显示日历中的所有剩余选项，忽略给定的 *end* 时间。 *start* 时间受到尊重；任何给定的参数都会被视为 *start* 时间。

`-s`

使用 shell 的 `sched` 命令调度定时事件，在事件到期时向用户发出警告。 请注意，`sched` 命令只有在 shell 处于交互式提示符时才会运行；前台任务会阻止计划任务的运行，直到它结束。

定时事件通常会运行 `calendar_show` 程序来显示事件，如 实用函数 所述。

默认情况下，事件警告会在到期前五分钟显示。 警告时间可通过 `warn-time` 样式配置，或者对单个日历条目，也可通过在条目的第一行中加入 WARN *reltime* 来配置。其中 *reltime* 是常用的相对时间格式之一。

重复事件可通过在第一行中加入 RPT *reldate* 来表示。 计划事件显示后，将在现有事件之后的 *reldate* 时间重新进入日历文件。 请注意，这是目前重复计数的唯一用途，因此在前一个事件过去之前，无法查询日历中事件的重复发生时间表。

如果使用 RPT，还可以指定某些事件的特定重复发生是否重新安排或取消。使用 OCCURRENCE 关键字完成此，后跟空格和事件在常规序列中的日期和时间，然后是空格，接着是重新安排的事件的日期和时间，或者精确字符串 CANCELLED。在这种情况下，日期和时间必须完全符合 `text/calendar` MIME 类型（RFC 2445）所使用的 "带本地时间的日期" 格式，即 <*YYYY*><*MM*><*DD*>T<*hh*><*mm*><*ss*>（注意文字 T 的存在）。第一个词（正常重复发生）可以是其他内容，而不是正确的日期/时间，以表示事件是正常序列之外的额外事件；保留格式外观的约定是 XXXXXXXXTXXXXXX。

此外，记录下一次定期重复事件也很有用（因为此时显示的日期可能是重新安排的事件，因此不能用于计算定期序列）。 这可以通过 RECURRENCE 和格式相同的时间或日期来指定。 `calendar_add` 在遇到不包含重复事件的重复事件时，会根据标题日期/时间添加这样的指示。

如果 `calendar_add` 被用于更新事件，则现有条目和新增事件中都应包含 UID 关键字，以便识别重复出现的事件序列。

例如,

```
Thu May 6, 2010 11:00 Informal chat RPT 1 week
  # RECURRENCE 20100506T110000
  # OCCURRENCE 20100513T110000 20100513T120000
  # OCCURRENCE 20100520T110000 CANCELLED
```

2010年5月13 日 11:00 的活动改期到一小时后。 一周后发生的事件被取消。 事件发生在以 # 字符开头的续行中，因此通常不会作为事件的一部分显示。 和其他地方一样，时间不考虑时区。下一个事件发生后，头条(headline)日期/时间将是 'Thu May 13, 2010 12:00'，而 RECURRENCE 日期/时间将是 '20100513T110000'（请注意，RECURRENCE 记录的是下一个常规重复事件的时间，而取消和移动的事件不在其中，但它们会在头条日期/时间中显示）。

运行 `calendar -s` 来重新安排现有事件是安全的（例如，如果日历文件已更改），而且还可以在 shell 的多个实例中运行，因为日历文件在使用时是锁定的。

默认情况下，过期事件会被移至 "已完成 "文件；请参阅 `-d` 选项。 使用 `-D` 可以避免这种情况。

`-S` *showprog*

明确指定用于显示事件的程序，而不是 `show-prog` 样式的值或默认的 `calendar_show`。

`-v`

Verbose：显示更多处理阶段的信息。 这有助于确认函数已成功解析日历文件中的日期。

`calendar_add [ -BL ]` *event ...*

在日历的适当位置添加一个事件。 如 文件和日期格式所述，事件可以包含多行。 使用此函数可确保日历文件按日期和时间顺序排序。 在修改文件时，它还会为锁定文件做出特殊安排。 旧版日历将保留在后缀为 `.old` 的文件中。

选项 `-B` 表示备份日历文件的工作将由调用者负责，而不应由 `calendar_add` 执行。 选项 `-L` 表示 `calendar_add` 无需锁定日历文件，因为它已经被锁定。 用户通常不需要这些选项。

如果样式 `reformat-date` 为 true，新条目中的日期和时间将被改写为标准日期格式：请参阅该样式和样式 `date-format` 的说明。

函数可使用与每个事件一起存储的唯一标识符，以确保正确处理对现有事件的更新。 条目应包含 UID 字样，后面是空白，然后是一个完全由任意长度的十六进制数字组成的字样（所有数字都有意义，包括前导零）。 由于 UID 对用户没有直接用处，因此可以将其隐藏在以 # 开头的缩进续行中,例如：

```
Aug 31, 2007 09:30  Celebrate the end of the holidays
  # UID 045B78A0
```

calendar 函数不会显示第二行。

可以指定 RPT 关键字，然后用 CANCELLED 代替相对时间。 这将导致任何匹配的事件或系列事件被取消（使用此方法取消事件时，原始事件不必被标记为重复发生）。 要匹配日历中的现有事件，必须有 UID。

calendar_add 将尝试管理重复事件的发生和重现，就像上文 calendar -s 所描述的事件调度一样。 要重新安排或取消单个事件，calendar_add 应调用包含正确 UID 的条目，但 **不能** 包含 RPT 关键字，因为这意味着该条目适用于一系列重复事件，因此会取代所有现有信息。 每个重新安排或取消的事件都必须在传给 calendar_add 的条目中包含 OCCURRENCE 关键字，并将其合并到日历文件中。 任何对该事件的现有引用都会被替换。 未引用有效事件的事件发生记录将作为一次性事件发生记录添加到同一日历条目中。

calendar_edit

调用用户的编辑器编辑日历文件。 如果有参数，它们将作为要使用的编辑器（文件名会附加到命令中）；否则，编辑器由变量 VISUAL 指定（如果已设置），否则由变量 EDITOR 指定。

如果日历调度程序正在运行，则会在编辑文件后调用 calendar -s 来更新文件。

在编辑过程中，该函数会锁定日历系统。 因此，在编辑日历文件时，如果同时可能发生日历事件，则应使用该函数。 请注意，这会导致另一个已启用日历功能的 shell 挂起等待锁定，因此必须尽快退出编辑器。

calendar_parse *calendar-entry*

这是一个内部函数，用于分析作为唯一参数传递的日历项的各个部分。 如果参数不能被解析为日历条目，函数将返回状态 1；如果传递的参数个数错误，函数将返回状态 2；它还会将参数 reply 设置为空关联数组。 否则返回状态 0，并按如下方式设置关联数组 reply 中的元素：

time

与 $EPOCHSECONDS 单位相同的数字时间字符串

schedtime

定期安排的时间。 如果这是一个重复发生的事件，且下一次事件发生的时间已重新安排，则该时间可能与实际事件时间 time 不同。 那么 time 给出的是实际时间，而 schedtime 给出的是修改前的定期重现时间。

text1

该行的文本，不包括事件的日期和时间，但包括任何 WARN 或 RPT 关键字和值。

`warntime`

任何由 WARN 关键字给出的警告时间，以与 $EPOCHSECONDS 相同的单位表示，包含警告时间的数字字符串。 (注意，这是绝对时间，而不是向下传递的相对时间。 没有 WARN 关键字和值匹配，则不设置。

`warnstr`

在 WARN 关键字之后匹配的原始字符串，否则取消设置。

`rpttime`

任何由 RPT 关键字给出的重复时间，是一个包含重复时间的数字字符串，单位与 $EPOCHSECONDS 相同。 (注意这是绝对时间。) 如果没有 RPT 关键字和值匹配，则不设置。

`schedrpttime`

在修改前，重复事件的下一次定期发生时间。 这可能与 rpttime不同，后者是事件的实际发生时间，可能已从正常时间重新安排。

`rptstr`

在 RPT 关键字之后匹配的原始字符串，否则取消设置。

`text2`

去掉日期、关键字和值后的文本。

`calendar_showdate` [`-r`] [`-f` *fmt*] *date-spec* ...

对给定的 *date-spec* 进行解释并打印相应的日期和时间。 如果初始的 *date-spec* 以 + 或 - 开头，则会被视为相对于当前时间；而第一个 *date-spec* 之后的 *date-spec*s 则被视为相对于迄今计算出的日期，在这种情况下，前导 + 是可选的。 这样就可以将系统用作日期计算器。 例如，`calendar_showdate '+1 month, 1st Friday'` 显示的是下个月第一个星期五的日期。

如果使用选项 `-r`，则不会打印任何内容，但会将日期和时间值（以秒为单位，自纪元epoch起）保存在参数 REPLY 中。

使用选项 `-f` *fmt* 将把指定的日期/时间转换格式传递给 `strftime`；请参阅下文关于 `date-format` 样式的说明。

为了避免负相对日期规范的歧义，选项必须出现在不同的词中；换句话说，`-r` 和 `-f` 不应组合在同一个词中。

`calendar_sort`

将日历文件按日期和时间顺序排序。 旧的日历会保留在后缀为 `.old` 的文件中。

23.3.2 Glob 限定符

`age`

函数 age 可以自动加载，并与日历系统分开使用，不过它使用函数 `calendar_scandate` 进行日期格式化。 它需要 `zsh/stat` 内置命令，但只使用 `zstat` 内置命令。

age 用于选择具有指定修改时间的文件作为 glob 限定符。 日期格式与日历系统理解的格式相同，详见 文件和日期格式。

函数可以接受一个或两个参数，这些参数可以直接作为命令或参数提供，也可以单独作为 shell 参数提供。

```
print *(e:age 2006/10/04 2006/10/09:)
```

以上示例匹配了在这些日期范围的起始时间之后修改过的所有文件。 第二个参数也可以是由 + 引入的相对时间：

```
print *(e:age 2006/10/04 +5d:)
```

上面的例子与前面的例子是等价的。

除了对星期几、`today` 和 `yesterday` 的特殊用法外，还可以指定不含日期的时间；这些时间适用于今天。 显然，这种使用在午夜前后会出现问题。

```
print *(e-age 12:00 13:30-)
```

上例显示的是今天 12:00 至 13:00 之间修改的文件。

```
print *(e:age 2006/10/04:)
```

上述示例匹配了在该日期修改的所有文件。 如果省略第二个参数，则会将其视为第一个参数之后的 24 小时（即使第一个参数包含时间）。

```
print *(e-age 2006/10/04:10:15 2006/10/04:10:45-)
```

上面的示例提供了时间。 请注意，时间和日期说明中的空白必须加引号，以确保 age 接收到正确的参数，因此使用了额外的冒号来分隔日期和时间。

```
AGEREF=2006/10/04:10:15
AGEREF2=2006/10/04:10:45
print *(+age)
```

下面显示的是使用另一种参数传递方式之前的示例。 参数 AGEREF 和 AGEREF2 中的日期和时间在未设置前一直有效，但如果有任何参数作为显式参数传递给 age，则会被覆盖。 任何显式参数都会导致两个参数被忽略。

通过在文件名前加冒号，可以使用文件的修改时间作为任一参数的日期和时间，而不用明确的日期和时间：

```
print *(e-age :file1-)
```

匹配与 file1 在同一天（从午夜开始的 24 小时）创建的所有文件。

```
print *(e-age :file1 :file2-)
```

匹配所有不早于 file1、不晚于 file2修改的文件；此处的精度为最接近的秒。

after
before

函数 after 和 before 是 age 的简化版本，只接受一个参数。 对参数的解析与 age 的参数类似；如果没有给出参数，则会查询变量 AGEREF。 正如函数名称所示，如果文件的修改时间在指定的时间和日期之后或之前，就会匹配该文件。 如果只给出了时间，那么日期就是今天。

因此，下面两个例子是等价的：

```
print *(e-after 12:00-)
print *(e-after today:12:00-)
```

23.4 样式

zsh/zutil 模块 中描述了使用 zstyle 命令的 zsh 样式机制。 这与补全系统中使用的机制相同。

下面的样式都是在 :datetime:*function*: 的上下文中检查的，例如 :datetime:calendar:。

calendar-file

主日历的位置。 默认为 ~/calendar。

date-format

一个 strftime 格式字符串（请参阅 strftime(3)），带有 zsh 扩展，提供各种数字，如果数字是个位数，则不带前导零或空格，如 %D{*string*} 提示符格式在 提示符扩展 中所述。

用于在 calendar 中输出日期，以支持 -v 选项，并将循环事件添加回日历文件，以及在 calendar_showdate 中作为最终输出格式。

如果未设置样式，则默认使用类似 date 命令输出的标准系统格式（也称为 'ctime 格式'）：'%a %b %d %H:%M:%S %Z %Y'。

done-file

追加已完成事件的文件位置。 默认情况下，日历文件的位置加上后缀 .done。 样式可以设置为空字符串，在这种情况下，将不会维护 "已完成" 文件。

reformat-date

布尔值，由 calendar_add 使用。 如果该值为 true，添加到日历中的新条目的日期和时间将按照 date-format 样式或其默认值重新格式化。 只有事件本身的日期和时间会被重新格式化，其他附属日期和时间（如与重复和警告时间相关的日期和时间）将保持不变。

show-prog

由 calendar 运行的程序，用于显示事件。 它将收到所请求事件的开始时间和停止时间，单位为自纪元(epoch)起的秒数，然后是事件文本。 请注意，calendar -s 使用与开始时间和停止时间相等的别一个时间来显示特定事件的警报。

默认使用函数 calendar_show。

warn-time

如果事件的第一行不包含 EVENT *reltime* 文本，则在事件发生前显示警告的时间。默认值为 5 分钟。

23.5 实用函数

calendar_lockfiles

尝试锁定参数中给出的文件。 为防止出现网络文件锁定的问题，将以临时的方式尝试创建一个名为 *file*.lockfile 的文件符号链接。 锁定过程中不使用其他系统级函数，也就是说，任何不使用该机制的实用程序都可以访问和修改文件。 尤其是，除非使用 calendar_edit，否则不会阻止用户同时编辑日历文件。

在放弃之前，会尝试三次锁定文件。 如果 zsh/zselect 模块可用，尝试的时间会被抖动，这样调用函数的多个实例就不太可能同时重试。

锁定的文件会追加到数组 lockfiles，该数组应为调用者的本地数组。

如果所有文件都成功锁定，则返回状态 0，否则返回状态 1。

该函数可用作一般的文件锁定函数，但只有在使用该机制锁定文件时才有效。

`calendar_read`

这是一个后端，用于其他各种函数解析日历文件，这是唯一的参数。 数组 `calendar_entries` 将被设置为文件中的事件列表；除了删除行首的逗号外，不会进行任何剪枝。 每个条目可以包含多行。

`calendar_scandate`

这是一个通用函数，用于解析可与日历系统分开使用的日期和时间。 参数是 文件和日期格式中描述的日期或时间规范。 参数 `REPLY` 将被设置为自该日期或时间对应的纪元起的秒数。 默认情况下，日期和时间可以出现在给定参数的任意位置。

如果成功解析日期和时间，则返回状态 0，否则返回状态 1。

选项:

`-a`

日期和时间锚定在参数的开头；如果前面有文本，则不会匹配。

`-A`

日期和时间被锚定到参数的开始和结束位置；如果参数中有任何其他文本，将不会被匹配。

`-d`

启用额外的调试输出。

`-m`

减 。当 同时给出 `-R` *anchor_time* 时，相对时间将从 *anchor_time* 开始倒推计算。

`-r`

传递的参数将被解析为相对时间。

`-R` *anchor_time*

传入的参数将被解析为相对时间。 时间是相对于 *anchor_time*（以秒为单位的从纪元(epoch)开始的时间）的，返回值是将 *anchor_time* 提前一个相对时间后的绝对时间。 这样就能正确考虑月的长度。 如果给定月份中没有最后一天，则返回最后一个月的最后一天。 例如，如果锚点时间为 2007 年 1 月 31 日，相对时间为 1 个月，则最终时间为 2007 年 2 月 28 日的同一天。

`-s`

除了设置 REPLY 外，还可以将 REPLY2 设置为日期和时间去除后参数的剩余部分。 如果使用了 -A 选项，则此值为空。

`-t`

允许输入没有日期说明的时间。 日期假定为今天。 如果午夜的铁舌正在鸣响，则行为是未指定的。

`calendar_show`

默认用于显示事件的函数。 它接受事件的开始时间和结束时间（均以纪元epoch秒为单位）以及事件描述。

事件总是打印到标准输出中。 如果命令行编辑器处于活动状态（通常是这种情况），命令行将在输出后重新显示。

如果设置了参数 DISPLAY，且开始时间和结束时间相同（表明为计划事件），则函数会使用命令 `xmessage` 显示包含事件详细信息的窗口。

23.6 错误

由于该系统完全基于 shell 函数（`zsh/datetime` 模块提供了少量支持），因此所使用的机制并不像专用日历工具那样强大。 因此，用户不应依赖 shell 来获取重要警报。

没有 `calendar_delete` 函数。

日期和时间不支持本地化，也不支持使用时区。

月和年的相对周期没有考虑到可变的天数。

`calendar_show` 函数目前硬连接到使用 `xmessage` 在 X 窗口系统显示器上显示警报。这应该是可配置的，最好能与桌面更好地集成。

`calendar_lockfiles` 会在等待锁定文件时挂起 shell。 如果从计划任务中调用，则应重新安排导致挂起的事件。

---

24 TCP 函数系统

24.1 说明

`zsh/net/tcp` 模块用于在 shell 中通过 TCP/IP 提供网络 I/O；请参阅 Zsh 模块 中的描述。 本手册介绍了基于该模块的函数套件。 如果安装了该模块，函数通常也会同时安装，在这种情况下，它们会在默认函数搜索路径中自动加载。 除 `zsh/net/tcp` 模块

外，zsh/zselect 模块还用于实现读取操作的超时。 有关故障排除技巧，请参考 Zftp 函数系统 中描述的 zftp 函数的相应建议。

系统提供了与基本 I/O 操作打开、关闭、读取和发送相对应的函数（名为 tcp_open 等），以及用于对作为输入读取的数据进行模式匹配分析的函数 tcp_expect。 该系统可以轻松地同时从多个命名会话接收数据并向其发送数据。 此外，它还可以与 shell 的行编辑器连接，从而在终端自动显示输入数据。 其他可用设施包括日志记录、过滤和可配置的输出提示符。

要使用可用的系统，只需 'autoload -U tcp_open' 并运行 tcp_open 即可启动会话。 tcp_open 函数将自动加载其余函数。

24.2 TCP 用户函数

24.2.1 基本 I/O

tcp_open [-qz] *host port* [*sess*]
tcp_open [-qz] [-s *sess* | -l *sess*[,...]] ...
tcp_open [-qz] [-a *fd* | -f *fd*] [*sess*]

打开一个新会话。 最简单的第一种形式是在端口 *port* 上打开与主机 *host* 的 TCP 连接；数字和符号形式都可以理解。

如果给出 *sess*，它将成为会话的名称，可用于指代多个不同的 TCP 连接。 如果未给出 *sess*，函数将生成一个数字名称值（注意这和与会话所连接的文件描述符 **不** 相同）。 建议会话名称不包含 'funny' 字符， funny 字符没有明确定义，但肯定不包含字母数字或下划线，而且肯定包含空白。

在第二种情况下，需要打开的一个或多个会话可以用名称表示。 在 -s 后给出一个会话名称，在 -l 后给出一个以逗号分隔的列表；这两个选项可根据需要重复多次。 如果无法打开任何会话，tcp_open 就会终止。 主机和端口从 .ztcp_sessions 文件中读取，该文件与用户的 zsh 初始化文件位于同一目录，即通常的主目录，但如果设置了 $ZDOTDIR，也可以从 $ZDOTDIR 中读取。 该文件由几行组成，每行按顺序给出一个会话名称以及相应的主机和端口（注意会话名称在前，而不是在后），中间用空格隔开。

第三种形式允许被动和假 TCP 连接。 如果使用选项 -a，其参数就是一个用于监听连接的文件描述符。 没有提供打开此类文件描述符的前端函数，但调用 'ztcp -l *port*' 将创建一个文件描述符，该文件描述符存储在参数 $REPLY 中。 可以使用 'ztcp -c *fd*' 关闭监听端口。 调用 'tcp_open -a *fd*' 将被阻塞，直到本地计算机上的 *port* 建立了远程 TCP 连接。 此时，会话将以常规方式创建，与使用前两种形式之一创建的活动连接基本没有区别。

如果使用选项 -f，它的参数就是一个文件描述符，直接使用它就像是 TCP 会话一样。 TCP 函数系统的其他部分能否很好地处理这个问题，取决于这个文件描述符的

实际内容。 普通文件可能无法使用； FIFO（管道）效果会更好，但要注意的是，两个不同的会话试图同时从同一个 FIFO 中读取数据并不是一个好主意。

如果选项 -q 与三种形式中的任何一种同时给出，tcp_open 将不会打印信息，但无论如何都会以适当的状态退出。

如果正在使用行编辑器 (zle)，也就是通常的交互式 shell，tcp_open 会在 zle 中安装一个处理程序，在检查键盘输入的同时检查是否有新数据。 这样做很方便，因为 shell 在等待时不会消耗 CPU 时间；测试由操作系统执行。 如果在 tcp_open 的任何形式中添加 -z 选项，则无法安装处理程序，因此必须明确读取数据。 但要注意的是，从函数中执行整套发送和读取命令并不是必须的，因为此时zle并不处于活动状态。 一般来说，只有在 shell 在命令提示符下输入或在 vared 内置命令中时，处理程序才会激活。 如果 zle 未激活，该选项将不起作用；'[[-o zle]]' 将对此进行测试。

第一个打开的会话将成为当前会话，后续调用 tcp_open 不会改变它。 当前会话存储在参数 $TCP_SESS 中；有关系统使用参数的更多详情，请参阅下文。

如果定义了函数 tcp_on_open，则会在会话打开时调用该函数。 请参阅下面的说明。

tcp_close [-qn] [-a | -l *sess*[,...] | *sess* ...]

关闭指定的会话，如果没有指定，则关闭当前会话，如果指定了 -a，则关闭所有打开的会话。 为了与 tcp_open 保持一致，-l 和 -s 选项都会被处理，不过后者是多余的。

如果关闭的会话是当前会话，$TCP_SESS 将被取消设置，即使还有其他会话仍处于打开状态，也不会留下当前会话。

如果会话是通过 tcp_open -f 打开的，那么只要文件描述符在命令行可直接访问的 0 至 9 范围内，就会被关闭。 如果给出 -n 选项，在这种情况下将不会尝试关闭文件描述符。 -n 选项不用于真正的 ztcp 会话；文件描述符总是与会话一起关闭。

如果给出选项 -q，则不会打印任何信息。

tcp_read [-bdq] [-t *TO*] [-T *TO*]
[-a | -u *fd*[,...] | -l *sess*[,...] | -s *sess* ...]

对当前会话执行读操作，或对会话列表执行读操作（如果使用 -u、-l 或 -s)，或所有打开的会话（如果使用 -a 选项）。 任何 -u、-l 或 -s 选项都可以重复或混合使用。 -u 选项直接指定文件描述符（只有本系统管理的文件描述符才有用），其他两个选项指定会话，如上文 tcp_open 所述。

该函数会检查列出的所有会话是否有新数据。 除非给出 -b 选项，否则不会阻塞等待新数据的过程。 除非 $TCP_SILENT 包含一个非空字符串，否则将从任何可用会话中读取任何一行数据，并存储在参数 $TCP_LINE 中，然后显示到标准输出中。

打印到标准输出时，字符串 $TCP_PROMPT 将显示在行首；默认形式包括正在读取的会话名称。 有关这些参数的更多信息，请参阅下文。 在这种模式下，可以反复调用 tcp_read，直到返回状态 2（表示已处理完来自所有指定会话的所有待处理输入）为止。

如果使用选项 -b（相当于无限超时），函数将阻塞，直到从指定会话中读取一行。不过，只会返回一行。

选项 -d 表示应排空所有待处理的输入。 在这种情况下，tcp_read 可以按照上述方式处理多行数据；只有最后一行数据会保存在 $TCP_LINE 中，但整组数据集会保存在数组 $tcp_lines 中。 每次调用 tcp_read 开始时，都会清空该数组。

选项 -t 和 -T 以秒为单位指定超时时间，为提高精确度，可以使用浮点数。 使用 -t，超时将在每次读取行之前应用。 对于 -T，超时适用于整个操作，如果有 -d 选项，则可能包括多个读取操作；如果没有该选项，则 -t 和 -T 没有区别。

该函数不打印信息，但如果给出选项 -q，则不会为不存在的会话打印错误信息。

返回状态为 2 表示超时或无数据可读。 任何其他非零返回状态都表示出现错误条件。

请参阅 tcp_log，了解如何控制 tcp_read 发送数据的位置。

tcp_send [-cnq] [-s *sess* | -l *sess*[,...]] *data* ...
tcp_send [-cnq] -a *data* ...

依次向所有指定会话发送所提供的数据字符串。 其基本操作与向会话的文件描述符发送 'print -r' 几乎没有什么区别，不过它试图防止因试图向已失效的会话写入数据而导致 SIGPIPE 引起的 shell 死机。

选项 -c 会使 tcp_send 的行为与 cat 类似。 它会从标准输入读取行，直到输入结束，然后依次将它们发送到指定的会话，就像将 *data* 作为 tcp_send 命令的参数一样。

选项 -n 可防止 tcp_send 在数据字符串末尾增加换行。

其余选项的行为与 tcp_read 相同。

数据参数传递给 tcp_send 后不会再进一步处理，而是直接传给 print -r。

如果参数 $TCP_OUTPUT 是一个非空字符串，并且启用了日志记录功能，那么发送到每个会话的数据都将以 $TCP_OUTPUT 的方式回传到日志文件，并酌情在前面加上 $TCP_PROMPT。

24.2.2 会话管理

tcp_alias [-q] *alias*=*sess* ...

tcp_alias [-q] [*alias* ...]
tcp_alias -d [-q] *alias* ...

该函数没有经过特别完善的测试。

第一种形式是为会话名称创建别名；然后可以使用 *alias* 来引用现有会话 *sess*。 可以根据需要列出任意多个别名。

第二种形式列出指定的别名，如果没有，则列出所有别名。

第三种形式会删除列出的所有别名。 底层会话不受影响。

选项 -q 会抑制选择不一致的错误信息子集。

tcp_log [-asc] [-n | -N] [*logfile*]

通过参数 *logfile*，将来所有来自 tcp_read 的输入将被记录到指定文件中。 除非给出 -a（追加），否则该文件将首先被截断或创建为空文件。 在没有参数的情况下，显示日志记录的当前状态。

使用选项 -s，可启用按会话记录日志功能。 来自 tcp_read 的输入将输出到文件 *logfile*.*sess*。 由于文件名会自动区分会话，因此文件内容是原始的（没有 $TCP_PROMPT）。 选项 -a 的应用同上。 按会话记录日志和在一个文件中记录所有数据并不相互排斥。

选项 -c 关闭所有日志记录，包括完整日志和每个会话的日志。

选项 -n 和 -N 分别关闭或恢复 tcp_read 读取的数据输出到标准输出；因此，'tcp_log -cn' 关闭 tcp_read 的所有输出。

该函数纯粹是设置参数 $TCP_LOG、$TCP_LOG_SESS、$TCP_SILENT 的便捷前端，下文将对这些参数进行说明。

tcp_rename *old new*

将会话 *old* 重命名为会话 *new*。 旧名称将失效。

tcp_sess [*sess* [*command* [*arg* ...]]]

在没有参数的情况下，列出所有打开的会话和相关文件描述符。 当前会话用星号标记。 在函数中使用时，直接访问参数 $tcp_by_name、$tcp_by_fd 和 $TCP_SESS 可能更方便；请参阅下文。

通过 *sess* 参数，将当前会话设置为 *sess*。 这等同于直接更改 $TCP_SESS。

使用额外的参数，在执行 '*command arg* ...' 时，会临时设置会话。 *command* 会被重新计算，以便扩展别名等，但其余的 *arg*s 会以 tcp_sess 显示的形式传递。 当 tcp_sess 退出时，将恢复原始会话。

24.2.3 高级 I/O

tcp_command *send-option ... send-argument ...*

这是 tcp_send 的便捷前端。 所有参数传递给 tcp_send，然后函数暂停等待数据。 当数据至少每 $TCP_TIMEOUT（默认值 0.3）秒到达时，将根据当前设置处理并打印数据。 始终返回状态 0。

这通常只适用于交互式使用，以防止从连接返回的输出使显示变得支离破碎。 在程序或函数中，通常最好使用更明确的方法来处理读取数据。

tcp_expect [-q] [-p *var* | -P *var*] [-t *TO* | -T *TO*]
[-a | -s *sess* | -l *sess*[,...]] *pattern ...*

等待来自任何指定会话的与给定 *patterns* 匹配的输入。 在输入行与给定模式之一匹配之前，输入将被忽略；此时将返回状态 0，匹配的行将存储在 $TCP_LINE 中，调用 tcp_expect 时读取的所有行将存储在数组 $tcp_expect_lines 中。

会话的指定方式与 tcp_read 相同：默认使用当前会话，否则使用 -a、-s 或 -l 指定的会话。

每个 *pattern* 都是一个标准的 zsh 扩展 globbing 模式；注意需要加引号，以避免在生成文件名时被立即展开。 它必须匹配整行，因此要匹配子串，必须在开头和结尾都有一个 '*'。 匹配的行包括 tcp_read 添加的 $TCP_PROMPT。 可以在模式中包含 globbing 标志 '#b' 或 '#m'，以便在参数 $MATCH、$match 等中提供反向引用，具体说明请参见模式匹配的基础 zsh 文档。

与 tcp_read 不同，tcp_expect 的默认行为是无限阻塞，直到找到所需的输入。 您可以使用 -t 或 -T 指定超时时间；它们的功能与 tcp_read 相同，分别以整数或浮点数指定每次读取或整体超时时间（以秒为单位）。 与 tcp_read 一样，如果超时，函数将返回状态 2。

只要所给模式中有任何一个匹配，函数就会返回。 如果调用者需要知道哪些模式匹配，可以使用 -p *var* 选项;返回时，$var 将使用普通的 zsh 索引设置为模式的编号，即第一个为 1，以此类推。 请注意 *var* 前面没有 '$'。 为避免冲突，参数不能以 '_expect' 开头。 如果出现超时，则使用索引-1，如果没有匹配，则使用索引 0。

选项 -P *var* 的工作原理与 -p类似，但正则参数必须以前缀开头，后跟冒号，而不是以数字索引开头：当参数匹配时，前缀会被用作 *var* 的标记。 如果有超时，则使用 timeout 标记，如果没有匹配，则使用空字符串。 需要注意的是，如果不需要区分不同的匹配，也可以使用以相同前缀开头的参数。

选项 -q 会直接传给 tcp_read。

由于所有输入都是通过 tcp_read 完成的，因此所有关于读取行输出的常规规则都适用。 一个例外是，参数 $tcp_lines 只反映 tcp_expect 实际匹配的行；使用 $tcp_expect_lines 可以获得函数调用期间读取的全部行。

tcp_proxy

这是一个思想简单的函数，用于接受 TCP 连接并执行命令，同时将 I/O 重定向到该连接。 由于没有任何安全性可言，使用时应格外小心，因为这可能会让你的电脑向全世界敞开大门。 理想情况下，它只能在防火墙后使用。

第一个参数是函数将监听的 TCP 端口。

其余参数给出了要执行的命令及其参数，并将标准输入、标准输出和标准错误重定向到接受 TCP 会话的文件描述符。 如果没有给出命令，则会启动一个新的 zsh。 这样，网络上的每个人都可以直接访问你的账户，这在很多情况下是件坏事。

该命令在后台运行，因此 tcp_proxy 可以接受新的连接。 它会持续接受新连接，直至被中断。

tcp_spam [-ertv] [-a | -s *sess* | -l *sess*[,...]] *cmd* [*arg* ...]

依次为每个会话执行 '*cmd* [*arg* ...]'。 请注意，这将执行命令和参数；除非给出 -t（transmit）选项，否则不会将命令行作为数据发送。

会话可以通过标准的 -a、-s 或 -l 选项显式选择，也可以隐式选择。 如果没有给出这三个选项，则规则如下：首先，如果设置了 $tcp_spam_list 数组，则将其作为会话列表，否则将选择所有会话。 其次，$tcp_no_spam_list 数组中的任何会话都将从会话列表中删除。

通常，通过 '-a' 标志添加的会话或隐式选择所有会话时，会按字母顺序列出；通过 $tcp_spam_list 数组或命令行给出的会话，会按给出的顺序列出。 如果使用 -r 标志，则会颠倒列出,不管它是如何得到的。

-v 标志指定在每次会话前输出 $TCP_PROMPT。 在用户定义的 tcp_on_spam 函数对 TCP_SESS 进行任何修改后，都会输出该信息。 (显然，该函数能够生成自己的输出）。

如果存在选项 -e，则使用 eval 执行以 '*cmd* [*arg* ...]' 形式给出的行，否则不做任何进一步处理直接执行。

tcp_talk

这是一个相当简单的尝试，目的是将输入强制发送到默认的 TCP_SESS。

转义字符 $TCP_TALK_ESCAPE（默认为 ':'）用于允许访问正常的 shell 操作。 如果该字符串位于行首，或后面只有空白，行编辑器将返回正常操作。 否则，将跳过该字符串和后面的空白，并将该行的其余部分作为 shell 输入执行，而不改变行编辑器的运行模式。

在使用命令历史记录方面，当前的实现方式存在一定缺陷。 因此，许多用户倾向于使用某种形式的替代方法，以方便地向当前会话发送数据。 一种简单的方法是将某些特殊字符（如 '%'）别名为 'tcp_command --'。

tcp_wait

唯一的参数是一个整数或浮点数，表示延迟的秒数。 在这段时间内，除了通过调用 tcp_read -a 来等待所有 TCP 会话的输入外，shell 不会做任何事情。 这与安装了 zle 处理程序后在命令提示符下的交互行为类似。

24.2.4 'One-shot' 文件传输

tcp_point *port*
tcp_shoot *host port*

这对函数提供了在 shell 内两个主机间传输文件的简单方法。 需要注意的是，目前使用 cat 来进行大量数据传输。 tcp_point 读取到达 *port* 的任何数据，并将其发送到标准输出；tcp_shoot 连接到 *host* 上的 *port* 并发送其标准输入。 可以使用任何未使用的 *port*；选择端口的标准方法是随机想一个高于 1024 的四位数，直到有一个能用为止。

在 springes 上从主机 woodcock 向主机 springes 传输文件：

```
tcp_point 8091 >output_file
```

和 woodcock：

```
tcp_shoot springes 8091 <input_file
```

由于这两个函数不需要 tcp_open 先建立 TCP 连接，因此可能需要单独自动加载。

24.3 TCP 用户定义函数

如果用户定义了某些函数，函数系统会在某些上下文中调用这些函数。 该功能依赖于 zsh/parameter 模块，通常在交互式 shell 中可用，因为补全系统依赖于它。 这些函数都不需要定义；它们只是在必要时提供方便的钩子。

通常情况下，这些参数会在请求的操作完成后被调用，这样各种参数就会反映新的状态。

tcp_on_alias *alias fd*

定义了别名后，调用该函数时将使用两个参数：别名的名称和相应会话的文件描述符。

tcp_on_awol *sess fd*

如果函数 tcp_fd_handler 正在处理来自行编辑器的输入，并检测到文件描述符不再可重复使用，默认情况下会将其从本方法处理的文件描述符列表中删除，并打

印一条信息。 如果定义了函数 tcp_on_awol，则会紧接着在这之前调用该函数。它可能会返回状态 100，表示仍应执行正常的处理；任何其他返回状态都表示不应采取进一步的操作，tcp_fd_handler 应立即以给定的状态返回。 通常情况下，tcp_on_awol 的操作是关闭会话。

如果需要在从函数打印输出之前使用 'zle -I' 来使行编辑器显示无效，则变量 TCP_INVALIDATE_ZLE 将是一个非空字符串。

('AWOL' 是军事术语，意为 '未经请假而缺勤' 或某种变体。 就作者所知，该词并无现成的技术含义) 。

tcp_on_close *sess fd*

调用该函数时，需要输入正在关闭的会话名称以及与该会话相对应的文件描述符。调用该函数时，这两个都将失效。

tcp_on_open *sess fd*

以会话名称和文件描述符为参数定义新会话后调用。 如果返回非零状态，则假定会话打开失败，并再次关闭会话；但 tcp_open 会继续尝试打开命令行中指定的其他会话。

tcp_on_rename *oldsess fd newsess*

会话重命名后会调用该函数，参数包括旧会话名称、文件描述符和新会话名称。

tcp_on_spam *sess command ...*

每个会话都会被调用一次,在 tcp_spam 为会话执行命令之前。 参数包括会话名称和要执行的命令列表。 如果 tcp_spam 在调用时使用了选项 -t，则第一个命令将是 tcp_send。

该函数在 $TCP_SESS 设置以反应要发送的会话后，但在使用该会话之前被调用。因此，可以在此函数中更改 $TCP_SESS 的值。 例如，tcp_spam 的会话参数可以包含额外信息，并在 tcp_on_spam 中进行剥离和处理。

如果函数将参数 $REPLY 设置为 'done'，则不执行命令行；此外，对于 tcp_spam 的 -v 选项，也不打印提示符。

tcp_on_unalias *alias fd*

在删除别名后，会使用别名的名称和相应会话的文件描述符调用该函数。

24.4 TCP 实用函数

TCP 函数系统使用以下函数，但很少需要直接调用。

tcp_fd_handler

这是 tcp_open 安装的函数，用于处理来自行编辑器的输入（如果需要）。 其格式与 Zle 内置命令 中的内置命令 'zle -F' 相同。

当处于激活状态时，函数会将参数 TCP_HANDLER_ACTIVE 设为 1，从而允许内部调用的 shell 代码（例如，通过设置 tcp_on_read）在编辑器提示符下处于空闲状态时判断是否被调用。

tcp_output [-q] -P *prompt* -F *fd* -S *sess*

该函数用于在 tcp_read 和（如果设置了 $TCP_OUTPUT）tcp_send 中记录日志和处理到达标准输出的输出。

要使用的 *prompt* 由 -P 指定；默认为空字符串。 它可以包含:

%c

如果会话是当前会话，则扩展为 1，否则为 0。与三元表达式（如 '%(c.-.+)'）一起使用时，为当前会话输出 '+'，否则输出 '-'。

%f

由会话的文件描述符代替。

%s

用会话名称代替。

%%

由单个 '%' 代替。

选项 -q 会抑制输出到标准输出，但不会抑制输出到任何已配置的日志文件。

-S 和 -F 选项用于传递会话名称和文件描述符，以便在提示符中进行替换。

24.5 TCP 用户参数

参数遵循通常的惯例，即标量和整数使用大写，普通数组和关联数组使用小写。 用户代码可以安全地读取这些参数。 有些参数也可以设置；这些参数已明确注明。 其他参数包括在这一组中，因为它们是由函数系统为了用户的利益而设置的，也就是说，设置它们通常没有什么用处，但却是无害的。

将可设置参数设置为函数的局部参数通常也很有用。 例如，'local TCP_SILENT=1' 指定在函数调用期间读取的数据不会打印到标准输出，与函数外部的设置无关。 同样，'local TCP_SESS=*sess*' 为函数的持续时间设置会话，而 'local TCP_PROMPT=' 则指定在函数期间不使用提示符输入。

tcp_expect_lines

数组。 上次调用 tcp_expect 时读取的行集合，包括最后的行（$TCP_LINE）。

tcp_filter

数组。可直接设置。 一组扩展的 globbing 模式，如果与 tcp_output 中的模式匹配，将导致该行不被打印到标准输出。 这些模式的定义应与 tcp_expect 的参数相同。 向日志文件输出行不受影响。

TCP_HANDLER_ACTIVE

标量。 在 tcp_fd_handler 中设置为 1，以向递归调用的函数表明它们已在编辑器会话中被调用。 否则不设置。

TCP_LINE

tcp_read 读取的最后一行，因此也是 tcp_expect 。

TCP_LINE_FD

$TCP_LINE 读取的文件描述符。 ${tcp_by_fd[$TCP_LINE_FD]} 将给出相应的会话名称。

tcp_lines

数组。最后一次调用 tcp_read 时读取的行的集合，包括最后一次调用（$TCP_LINE）。

TCP_LOG

可以直接设置，但也受 tcp_log 控制。 所有会话的输出将要发送到的文件名。 输出将以通常的 $TCP_PROMPT 方式进行。 如果不是绝对路径名，则将遵循用户的当前目录。

TCP_LOG_SESS

可以直接设置，但也受 tcp_log 控制。 一组文件的前缀，每个会话的输出将分别发送到这些文件；完整的文件名是 ${TCP_LOG_SESS}.*sess*。 每个文件的输出都是原始的，不会添加提示符。 如果不是绝对路径名，则将遵循用户的当前目录。

tcp_no_spam_list

数组。 可直接设置。 有关使用方法，请参见 tcp_spam。

TCP_OUTPUT

可直接设置。 如果是非空字符串，则会记录 tcp_send 发送到会话的任何数据。 此参数给出的提示符将用于 $TCP_LOG 指定的文件，但不用于 $TCP_LOG_SESS 生成的文件。 提示符字符串的格式与 TCP_PROMPT 相同，使用规则也相同。

TCP_PROMPT

可直接设置。 作为 tcp_read 读取数据的前缀，这些数据会被打印到标准输出或 $TCP_LOG 指定的日志文件（如果有）。 字符串中出现的'%s', '%f' 或 '%%' 将分别被会话名称、会话的底层文件描述符或单个 '%' 替换。 如果读取的会话是当前会话，表达式 '%c' 将扩展为 1，否则为 0；这在三元表达式中最有用，例如 '%(c.-.+)'，如果会话是当前会话，则输出 '+'，否则输出 '-'。

如果提示符以 %P 开头，则会将其删除，并在输出前将前一阶段的完整结果通过标准提示符 % 样式的格式进行处理。

TCP_READ_DEBUG

可直接设置。 如果长度不为零，tcp_read 将对读取的数据进行有限的诊断。

TCP_SECONDS_START

该值由 tcp_open 创建并初始化为 0。

函数 tcp_read 和 tcp_expect 会使用 shell 的 SECONDS 参数来实现自己的定(计)时目的。 如果在进入其中一个函数时该参数不是浮点类型，则会创建一个浮点的本地参数 SECONDS，并将参数 TCP_SECONDS_START 设置为 $SECONDS 之前的值。 如果参数已经是浮点数，则无需创建本地副本即可使用，也不会设置 TCP_SECONDS_START。 由于全局值为零，因此可以保证 shell 的运行时间是 $SECONDS 和 $TCP_SECONDS_START 的总和。

使用 'typeset -F SECONDS' 将 SECONDS 全局设置为浮点数，就可以避免这种情况；这样 TCP 函数就不会创建本地副本，也不会将 TCP_SECONDS_START 设置为非零值。

TCP_SESS

可直接设置。 当前会话；必须是 tcp_open 建立的会话之一。

TCP_SILENT

可以直接设置，但也受 tcp_log 控制。 如果长度不为零，tcp_read 读取的数据将不会写入标准输出，但仍有可能写入日志文件。

tcp_spam_list

数组。 可直接设置。 具体用法请参见函数 tcp_spam 的描述。

TCP_TALK_ESCAPE

可直接设置。 使用方法请参阅函数 tcp_talk 的说明。

TCP_TIMEOUT

可直接设置。 目前只有 tcp_command 函数会使用它，见上文。

24.6 TCP 用户定义参数

以下参数不是由函数系统设置，但如果由用户设置，则会产生特殊效果。

tcp_on_read

这应该是一个关联数组；如果不是，则行为未定义。 每个键都是 shell 函数或其他命令的名称，相应的值是 shell 模式（使用 EXTENDED_GLOB）。 直接或间接使用 tcp_read 从 TCP 会话读取的每一行（包括通过 tcp_expect 读取的行）都会与模式进行比较。 如果匹配，则调用关键字中给出的命令，命令包含两个参数：读取该行的会话名称和该行本身。

如果为处理某一行而调用的函数返回非零状态，则不输出该行。 因此，只包含指令 'return 1' 的 tcp_on_read 处理程序可以用来抑制特定行的输出（参见上文的 tcp_filter）。 不过，该行仍会存储在 TCP_LINE 和 tcp_lines 中；这发生在所有 tcp_on_read 处理之后。

24.7 TCP 实用参数

这些参数由函数系统控制；可以直接读取，但通常不应由用户代码设置。

tcp_aliases

关联数组。 键是使用 tcp_open 建立的会话名称；每个值是以空格分隔的别名列表，引用了该会话。

tcp_by_fd

关联数组。 键是会话文件描述符；每个值是该会话的名称。

tcp_by_name

关联数组。 键是会话的名称，每个值是与该会话相关的文件描述符。

24.8 TCP 示例

下面是一个使用远程计算器的微不足道的例子。

在 7337 端口创建计算器服务器（请参阅 dc 手册页面，了解底层命令有多令人恼火）：

```
tcp_proxy 7337 dc
```

从同一主机连接到同样名为 'dc' 的会话：

```
tcp_open localhost 7337 dc
```

向远程会话发送命令并稍候输出（假设 dc 是当前会话）：

```
tcp_command 2 4 + p
```

要关闭会话：

```
tcp_close
```

tcp_proxy 需要被删除才能停止。 请注意，这通常不会删除任何已接受的连接，而且端口也不会立即被重用。

下面的代码会将会话列表放入 xterm 头中，当前会话后面跟一个星号。

```
print -n "\033]2;TCP:" ${(k)tcp_by_name:/$TCP_SESS/$TCP_SESS\*} "\a"
```

24.9 TCP 问题

函数 tcp_read 使用 shell 的常规 read 内置函数。 由于该函数一次读取一整行的数据，如果数据到达时没有换行结束符，就会导致函数无限阻塞。

虽然该函数套件在交互式使用和传输少量数据时运行良好，但在交换大量数据时，其性能可能极差。

---

25 Zftp 函数系统

25.1 说明

这里介绍的是随源码发行版提供的一组 shell 函数，它们是 zftp 内置命令的接口，允许你在 shell 命令行或函数或脚本中执行 FTP 操作。 该界面类似于传统的 FTP 客户端（例如 ftp 命令本身，参见 ftp(1)），但由于它完全是在 shell 中完成的，因此所有熟悉的补全、编辑和 globbing 功能等都一应俱全，宏的编写也特别简单，因为它们只是普通的 shell 函数。

前提条件是 zftp 命令（如 zsh/zftp 模块 中所述）必须在您站点上安装的 zsh 版本中可用。 如果 shell 被配置为在运行时加载新命令，那么它很可能是可用的：键入 'zmodload zsh/zftp' 将确保这一点（如果运行时没有提示，则说明已经成功）。 如果不是这种情况，则有可能 zftp 已经链接到 shell 中：要测试这一点，请键入 'which zftp'，如果 zftp 可用，您将收到 'zftp: shell built-in command' 的信息。

直接使用 zftp 内建的命令可能会穿插在本套件的函数之间；在少数情况下，直接使用 zftp 可能会导致存储在 shell 参数中的某些状态信息失效。 请特别注意zftp 的变量 $ZFTP_TMOUT、$ZFTP_PREFS 和 $ZFTP_VERBOSE 的描述。

25.2 安装

您应确保源代码发行版 Functions/Zftp 目录中的所有函数都可用；它们都以两个字母 'zf' 开头。 它们可能已经安装在你的系统中；否则，你需要找到并复制它们。 目录应作为 $fpath 数组的元素之一出现（如果已经安装，则应如此），并且至少应自动加载 zfinit 函数；它将自动加载其他函数。 最后，要初始化系统的使用，需要调用 zfinit 函数。 .zshrc 中的以下代码将为此做出安排；假定函数存储在 ~/myfns 目录中：

```
fpath=(~/myfns $fpath)
autoload -U zfinit
zfinit
```

需要注意的是，zfinit 假设你正在使用 zmodload 方法加载 zftp 命令。 如果 shell 已内置该命令，请将 zfinit 改为 zfinit -n。 如果在任何初始化新补全系统的代码之后调用 zfinit ，则会有所帮助（但并非必要），否则会给出不必要的 compctl 命令。

25.3 函数

执行文件传输的操作顺序与标准 FTP 客户端的操作顺序基本相同。 需要注意的是，由于 shell 的 getopts 内置函数的特殊性，对于那些处理选项的函数，必须使用 '--' 而不是 '-' ，以确保剩余的参数按字面意思处理（单个'-' 被视为一个参数）。

25.3.1 打开连接

zfparams [*host* [*user* [*password* ...]]]

设置或显示未来 zfopen 的参数，不带实参。 如果未给出实参，则显示当前参数（密码将显示为一行星号）。 如果给定了 *host*，但未给定 *user* 或 *password*，则会提示输入这两个参数；此外，以 '?' 形式给定的任何参数都会被提示输入，如果 '?' 后面跟了一个字符串，则该字符串将被用作提示符。 由于 zfopen 会调用 zfparams 来存储参数，因此通常不需要直接调用它。

单个参数 '-' 将删除存储的参数。 这也会导致删除另一台主机上最后一个目录的记忆（以此类推）。

zfopen [-1] [*host* [*user* [*password* [*account*]]]]

如果存在 *host*，则以用户名 *user* 和密码 *password*（以及在极少数必要情况下的账户 *account*）打开与该主机的连接。 如果缺少必要的参数或参数为 '?'，系统将提示输入。 如果 *host* 不存在，则使用以前存储的参数集。

如果命令成功执行，且终端与 xterm 兼容或为 sun-cmd，标题栏中将显示摘要，给定本地 host:directory 和远程 host:directory；这由 zftp_chpwd 函数处理，详情如下。

通常，内部会记录 *host*、*user* 和 *password*，以便日后通过不带参数的 zfopen 或自动（见下文）重新打开。 如果使用选项 '-1'，则不会存储任何信息。 此外，如果带参数的打开命令失败，参数将不会被保留（之前的参数也将被删除）。 zfopen 本身或 zfopen -1，都不会更改存储的参数。

zfopen 和 zffanon（但不包括 zfparams）都能将 ftp://*host*/*path...* 形式的 URL 理解为连接到 *host*，然后将目录更改为 *path*（必须是目录，而非文件）。 'ftp://' 可以省略；尾部的 '/' 足以触发对 *path* 的识别。 需要注意的是，'ftp:' 以外的前缀不会被识别，而且 *host* 以外第一个斜线后的所有字符在 *path* 中都有意义。

zfanon [-1] *host*

为匿名 FTP 打开 *host* 连接。 使用的用户名为 'anonymous'。 密码（将在第一次报告）生成为 *user*@*host*；然后存储在 shell 参数 $EMAIL_ADDR 中，也可手动设置为合适的字符串。

25.3.2 目录管理

zfcd [*dir*]
zfcd -
zfcd *old new*

更改远程服务器上的当前目录：该功能具有 shell 内置 cd 的许多功能。

在存在 *dir* 的第一种形式中，更改为 *dir* 目录。 命令 'zfcd ..' 会被特殊处理，因此保证能在非 UNIX 服务器上运行（注意这将由 zftp 内部处理）。 如果省略 *dir*，则效果与 'zfcd ~' 相同。

第二个形式会更改为之前的当前目录。

第三种形式是尝试用 *new* 替换当前目录中第一个出现的 *old* 字符串，从而更改当前目录。

请注意，在这条命令中，以及在任何需要远程文件名的地方，本地主机上对应于 '~' 的字符串在传给远程机器之前都会被转换回 '~'。 这样做很方便，因为在 zfcd 接收字符串之前，会在命令行上执行扩展。 例如，假设命令是 'zfcd ~/foo'。 shell 会将其扩展为完整路径，如 'zfcd /home/user2/pws/foo'。 在此阶段，zfcd 会将识别初始路径为对应于 '~'，并将目录以 ~/foo 的形式发送到远程主机，这样服务器就会将 '~' 扩展为正确的远程主机目录。 其他形式为 '~name' 的命名目录不按此方式处理。

zfhere

将远程服务器上的目录更改为与当前本地目录相对应的目录，并对'~' 进行特殊处理，就像在 zfcd 命令中一样。 例如，如果当前的本地目录是 ~/foo/bar，那么 zfhere 就会执行 'zfcd ~/foo/bar' 的效果。

zfdir [-rfd] [-] [*dir-options*] [*dir*]

生成长目录列表。 参数 *dir-options* 和 *dir* 直接传递给服务器，其效果取决于实现，但通常可以指定特定的远程目录 *dir*。 输出通过环境变量 $PAGER 指定的分页器传递，如果未设置环境变量，则通过 'more' 传递。

目录通常被缓存起来，以便重复使用。 事实上，会维护两个缓存。 一个是在没有 *dir-options* 或 *dir* 时使用的缓存，即当前远程目录的完整列表；当当前远程目录发生变化时，缓存会被刷新。 另一个缓存用于重复使用相同参数的 zfdir；例如，重复使用 'zfdir /pub/gnu' 时，只需在第一次调用时检索目录。 另外，也可以使用 -r 选项重新查看这个缓存。 由于相对目录会混淆 zfdir，因此可以使用 -f 选项强制在列出目录之前刷新缓存。 选项 -d 会在不显示目录列表的情况下删除两个缓存；它还会删除当前远程目录中的文件名缓存（如果有的话）。

zfls [*ls-options*] [*dir*]

列出远程服务器上的文件。 在没有参数的情况下，它会产生一个当前远程目录的简单文件名列表。 任何参数都将直接传递给服务器。 不使用分页器和缓存。

25.3.3 状态命令

zftype [*type*]

在没有参数的情况下，显示要传输的数据类型，通常是 ASCII 或二进制数据。 如果有参数，则更改类型：对于 ASCII 数据，类型为 'A' 或'ASCII' ；对于二进制数据，类型为 'B' 或 'BINARY'、'I' 或 'IMAGE' ,大小写无关。

zfstat [-v]

显示当前或上次连接的状态，以及 zftp 的某些状态变量的状态。 使用 -v 选项，还可以通过查询服务器的事件版本，生成更详细的列表。

25.3.4 检索文件

检索文件的命令都至少包含两个选项。 -G 会抑制本应执行的远程文件名扩展（更详细的说明见下文）。 -t 尝试将本地文件的修改时间设置为远程文件的修改时间：更多信息，请参阅下文对函数 zfrtime 的描述。

zfget [-Gtc] *file1 ...*

从远程服务器一次检索所有列出的文件 *file1 ...* 。 如果文件包含 '/' ，全名将传递给远程服务器，但文件将以最后的 '/' 之后的部分给出的名称存储在本地。 选项 -c （cat）会强制将所有文件作为单一数据流发送到标准输出；在这种情况下，-t 选项不起作用。

zfuget [-Gvst] *file1 ...*

与 zfget相同，但只检索远程服务器上版本较新（修改时间较晚）或本地不存在的文件。 如果远程文件较早但文件大小不同，或者文件大小相同但远程文件较新，通常会询问用户。 如果使用选项 -s，命令将静默运行，并始终在这两种情况下检索文件。 如果使用选项 -v，命令在决定是否传输文件时会打印更多文件信息。

zfcget [-Gt] *file1* ...

与 zfget相同，但如果本地文件存在，且比相应远程文件短，命令会认为这是部分传输完成的结果，并尝试传输文件的其余部分。 这在连接不畅、不断失败的情况下非常有用。

请注意，这需要一个常用但非标准的 FTP 协议版本，因此不能保证在所有服务器上都能运行。

zfgcp [-Gt] *remote-file local-file*
zfgcp [-Gt] *rfile1 ... ldir*

该命令从远程服务器检索文件，参数行为与 cp 命令类似。

在第一种形式中，将 *remote-file* 从服务器复制到本地文件 *local-file*。

在第二种形式中，将所有远程文件 *rfile1* ... 复制到本地目录 *ldir* 中，并保留相同的基名。 这假定了 UNIX 的目录语义。

25.3.5 发送文件

zfput [-r] *file1* ...

将所有给定的 *file1* ... 分别发送到远程服务器。 如果文件名包含 '/'，本地将使用完整的文件名查找该文件，但远程文件名只使用基名。

使用选项 -r，如果 *files* 中有任何一个是目录，则会递归发送所有子目录，包括以 '.' 开头的文件。 这要求远程机器理解 UNIX 文件语义，因为 '/' 被用作目录分隔符。

zfuput [-vs] *file1* ...

与 zfput相同，但只发送比远程文件更新的文件，或者远程文件不存在时才发送。逻辑与 zfuget相同，但本地文件和远程文件的逻辑相反。

zfcput *file1* ...

与 zfput相同，但如果任何远程文件已经存在，且比本地文件短，则假定这是传输不完整的结果，并发送文件的其余部分以追加到现有部分。 由于 FTP 追加命令是标准集的一部分，因此原则上这比 zfcget 更有可能奏效。

zfpcp *local-file remote-file*
zfpcp *lfile1 ... rdir*

它会将文件发送到远程服务器，其参数行为与 cp 命令类似。

使用两个参数，将 *local-file* 复制到服务器上的 *remote-file* 中。

在有两个以上参数的情况下，将所有本地文件 *lfile1* ... 复制到现有远程目录 *rdir* 中，并保留相同的基名。 这假定了 UNIX 的目录语义。

如果尝试使用 zfpcp *lfile1 rdir* 就会出现问题，即复制的第二种形式，但有两个参数，因为命令无法简单判断 *rdir* 对应的是目录还是文件名。 命令试图通过多种方式解析这个问题。 首先，如果 *rdir* 参数是 '.' 或 '..'，或以斜线结束，则假定它是一个目录。 其次，如果以第一种形式复制到远程文件的操作失败，并且远程服务器发送了预期的失败代码 553 ，并包含字符串 'Is a directory' 的回复，那么 zfpcp 将使用第二种形式重试。

25.3.6 关闭连接

zfclose

关闭连接

25.3.7 会话管理

zfsession [-lvod] [*sessname*]

允许你同时管理多个 FTP 会话。 默认情况下，连接是在名为 'default' 的会话中进行的。 通过给出 'zfsession *sessname*' 命令，你就可以用自己选择的名称切换到一个新的或现有的会话。 新会话会记住自己的连接、相关的 shell 参数以及 zfparams 设置的 host/user 参数。 因此，你可以设置不同的会话来连接不同的主机，每个会话都会记住相应的主机、用户和密码。

在没有参数的情况下，zfsession 会打印当前会话的名称；在使用选项 -l 的情况下，它会列出当前存在的所有会话；在使用选项 -v 的情况下，它会给出一个详细的列表，显示每个会话的主机和目录，其中当前会话用星号标记。 如果使用 -o，则会切换到最近的上一个会话。

使用 -d，给定会话（或当前会话）将被移除；与之相关的一切都将被彻底遗忘。 如果它是唯一的会话，则会创建一个名为 'default' 的新会话，并使其成为当前会话。 在使用 zftp 的后台命令处于活动状态时，最好不要删除会话。

zftransfer *sess1*:*file1 sess2*:*file2*

在两个会话之间传输文件；不创建本地副本。 文件以 *file1* 的形式从会话 *sess1* 读取，并以文件 *file2* 的形式写入会话 *sess2*；*file1* 和 *file2* 可以是会话当前目录的相对文件。 *sess1* 或 *sess2* 均可省略（但如果文件名中可能出现冒号，则应保留冒号），并默认为当前会话；*file2* 可省略或以斜线结尾，在这种情况下，将添加 *file1* 的基名。 会话 *sess1* 和 *sess2* 必须是不同的。

该操作是使用管道执行的，因此要求连接在子 shell 中仍然有效，而在某些操作系统版本下，情况并非如此，这可能是由于系统漏洞造成的。

25.3.8 书签

通过 zfmark 和 zfgoto 这两个函数，您可以为当前 FTP 连接的当前位置（主机、用户和目录）打上 'bookmark'，以便日后使用。 用于存储和检索书签的文件由参数 $ZFTP_BMFILE 提供；如果调用这两个函数之一时未设置，则将设置为 zsh 启动文件所在目录（通常为 ~）中的文件 .zfbkmarks。

zfmark [*bookmark*]

如果给定了参数，则会在 *bookmark* 名称下标记当前主机、用户和目录，供 zfgoto 后续使用。 如果没有打开连接，则使用最后一个连接关闭前的值。 如果没有，则是错误。任何已存在的同名书签都将被无声替换。

如果没有给定参数，则会以 *user*@*host*:*directory* 的形式列出现有书签及其指向的点。这就是它们的存储格式，可以直接对文件进行编辑。

zfgoto [-n] *bookmark*

返回 zfmark 之前设置的 *bookmark* 指定的位置。 如果该位置有用户为 'ftp' 或 'anonymous'，则使用 zfanon 打开连接，因此无需密码。 如果用户和主机参数与当前会话存储的参数（如果有）匹配，则使用这些参数，同样无需密码。 否则将提示输入密码。

使用选项 -n，书签将被视为 ncftp 程序存储在其书签文件中的昵称，该文件被假定为 ~/.ncftp/bookmarks。 该函数在其他方面的工作原理相同。 请注意，zftp 函数中没有添加或修改 ncftp 书签的机制。

25.3.9 其它函数

大多数情况下，这些函数不会被直接调用（除了 zfinit），但为完整起见，还是在此介绍一下。 您可能希望对 zftp_chpwd 和 zftp_progress 进行修改。

zfinit [-n]

如上所述，它用于初始化 zftp 函数系统。 如果 shell 已内置 zftp 命令，则应使用 -n 选项。

zfautocheck [-dn]

调用该函数是为了实现自动重新打开行为，下文将详细介绍。 选项必须出现在第一个参数中；-n会阻止命令切换到旧目录，而 -d 则会阻止命令设置变量 do_close，否则命令会将其作为传输后自动关闭连接的标志。 最后一次会话的主机和目录保存在变量 $zflastsession 中，但内部 host/user/password 参数也必须正确设置。

`zfcd_match prefix suffix`

这将执行远程目录名的补全匹配。 如果远程服务器是 UNIX，它将尝试说服服务器列出远程目录，并标注子目录，这通常有效，但不能保证。 在其他主机上，它只需调用 `zfget_match`，从而补全所有文件，而不仅仅是目录。 在某些系统中，目录可能看起来甚至不像文件名。

`zfget_match prefix suffix`

该功能用于补全远程文件名的匹配。 它会将当前目录下的文件缓存到 shell 参数 `$zftp_fcache` 中。 它采用 `compctl` 的 `-K` 选项调用的形式，但也可在适当设置 *prefix* 和 *suffix* 的小部件式补全函数中调用。

`zfrglob varname`

执行远程 globbing，详情如下。 *varname* 是一个变量的名称，包含要扩展的模式；如果有任何匹配，返回时将把同一变量设置为扩展后的文件名集。

`zfrtime` *lfile rfile* [*time*]

设置本地文件 *lfile* 的修改时间与远程文件 *rfile* 的修改时间相同，或为 GMT 时区设置 FTP 格式 `CCYYMMDDhhmmSS` 的显式时间 *time* 。 这将使用 shell 的 `zsh/datetime` 模块执行从 GMT 到本地时间的转换。

`zftp_chpwd`

每次打开、关闭连接或更改远程目录时，都会调用该函数。 该版本会更改 `xterm` 兼容终端或 `sun-cmd` 终端模拟器的标题栏，以反映本地和远程主机名及当前目录。 与函数 `chpwd` 结合使用效果最佳。 特别是，下面形式的函数

```
chpwd() {
  if [[ -n $ZFTP_USER ]]; then
    zftp_chpwd
  else
    # usual chpwd e.g put host:directory in title bar
  fi
}
```

非常适合。

`zftp_progress`

此函数显示传输状态。 除非输出到终端，否则不会写入任何内容；不过，如果在后台传输文件，则应使用 ‘`zstyle ':zftp:*' progress none`’ 手动关闭进度报告。 还要注意的是，如果更改了进度表，任何输出都 **必须** 输出到标准错误，因为标准输出可能是正在接收的文件。 进度表的形式，或者是否使用进度表，都可以在不修改函数的情况下进行配置，详见下一节。

`zffcache`

用于为每个会话分别缓存当前目录中的文件。 它被 `zfget_match` 和 `zfrglob` 使用。

25.4 杂项功能

25.4.1 配置

使用标准 shell 样式机制可获得各种样式，详见 zsh/zutil 模块。简而言之，命令 ‘`zstyle ':zftp:*' style value ...`’. 定义 *style* 的值为 *value*；可以给出一个以上的值，但在这里描述的情况下并无用处。 这些值将在整个 zftp 函数系统中使用。 为了实现更精确的控制，可以修改第一个参数（该参数给出了与 ***contexts*** 匹配的模式，在该模式中，样式适用）以包含特定函数，例如 ‘`:zftp:zfget`’： 这样，样式将仅在 `zfget` 函数中具有给定值，并覆盖在 ‘`:zftp:*`’下设置的样式。 需要注意的是，只有用户调用的顶级函数名称才会被使用；低级函数的调用对用户是透明的。 因此，对 `zftp_chpwd` 中标题栏的修改会使用 `:zftp:zfopen`、`:zftp:zfcd` 等上下文，具体取决于从何处调用。 可理解以下样式：

`progress`

控制 `zftp_progress` 报告传输进度的方式。 如果为空、未设置或 ‘none’，则不会报告传输进度；如果为‘bar’，则会显示一个不断增长的反向视频条；如果为 ‘percent’（或其他字符串，但将来可能会更改），则会显示已传输文件的百分比。 条形仪表要求通过 `$COLUMNS` 参数提供终端的宽度（通常会自动设置）。 如果无法提供正在传输的文件大小，`bar` 和 `percent` 仪表将只显示目前传输的字节数。

运行 `zfinit` 时，如果没有为上下文 `:zftp:*` 定义此样式，则会将其设置为 ‘bar’。

`update`

指定进度表更新的最小时间间隔（秒）。 除非收到新数据，否则不会进行更新，因此实际时间间隔仅受 `$ZFTP_TIMEOUT` 的限制。

如 `progress` 所述，`zfinit` 会强制将其默认为 1。

`remote-glob`

如果设置为 ‘1’, ‘yes’ 或 ‘true’，文件名生成（globbing）将在远程机器上执行，而不是由 zsh 本身执行；见下文。

`titlebar`

如果设置为 '1', 'yes' 或 'true'，那么 zftp_chpwd 将在 xterm 或 sun-cmd 等终端模拟器的标题栏中显示远程主机和远程目录。

如 progress 所述，zfinit 会强制将其默认为 1。

chpwd

如果设置为 '1' 'yes' 或'true'，zftp_chpwd 将在连接关闭时调用函数 chpwd。 如果远程主机的详细信息被 zftp_chpwd 放进了终端标题栏，而您常用的 chpwd 也会修改标题栏，那么这将非常有用。

运行 zfinit 时，它会判断 chpwd 是否存在，如果存在，则会将样式的默认值设为 1,如果尚未存在样式。

请注意，还有一个关联数组 zfconfig，其中包含函数系统使用的值。 该数组不应被修改或覆盖。

25.4.2 远程 globbing

检索文件的命令通常会对其参数执行文件名生成（globbing）；通过向每个命令传递选项 -G 可以关闭该功能。 通常情况下，该操作是检索相关目录的完整文件列表，然后将这些文件与提供的模式进行本地匹配。 这样做的好处是可以使用全部的 zsh 模式（与选项 EXTENDED_GLOB 的设置一致）。 不过，这意味着文件名的目录部分不会被展开，必须准确给出。 如果远程服务器不支持 UNIX 目录语义，那么目录处理就会出现问题，因此建议只在当前目录下使用 globbing。 如果检索到当前目录下的文件列表，则会将其缓存起来，这样在同一目录下进行后续的 globb 操作时，不需要 zfcd 的介入，速度会更快。

如果设置了 remote-glob 样式（见上文），则会在远程主机上执行 globbing：要求服务器提供匹配文件的列表。 这在很大程度上取决于服务器是如何实现的，不过通常 UNIX 服务器会提供对基本 glob 模式的支持。 在某些情况下，这可能会更快，因为它避免了检索整个目录内容列表。

25.4.3 自动和临时重新打开

如 zfopen 命令所述，后续不带参数的 zfopen 将重新打开与上一个主机的连接（包括使用 zfanon 命令建立的连接）。 以这种方式打开的连接会从默认远程目录开始，并一直保持打开状态，直到显式关闭为止。

也可以自动重新打开。 如果连接当前未打开，而给出的命令需要连接，则会隐式地重新打开上一个连接。 在这种情况下，关闭连接时的当前目录将再次成为当前目录（当然，除非所下达的命令更改了该目录）。 如果远程服务器因某种原因（如超时）关闭了连接，也会自动重新打开连接。 如果使用了 zfopen 或 zfanon 的 -1 选项，则不可用。

此外，如果发出的命令是文件传输，连接将在传输完成后关闭，从而为传输提供了一次性模式。 这不适用于目录更改或列表命令；例如，zfdir 可能会重新打开连接，但会保持

打开状态。 此外，自动关闭只会发生在与自动打开相同的命令中，即 zfdir 之后直接执行 zfget 永远不会自动关闭连接。

zfstat 函数提供了前一次连接的信息。 因此，举例来说，如果报告:

```
Session:        default
Not connected.
Last session:   ftp.bar.com:/pub/textfiles
```

那么命令 zfget file.txt 将尝试重新打开与 ftp.bar.com 的连接，获取文件 /pub/textfiles/file.txt，然后立即再次关闭连接。 另一方面， zfcd .. 将在 /pub 目录中打开连接，并保持打开状态。

请注意，上述所有内容都是每个会话的本地内容；如果返回到以前的会话，则会重新打开该会话的连接。

25.4.4 补全

支持本地和远程文件、目录、会话和书签的补全。 调用 zfinit 时定义了较早的、compctl 风格的补全；对基于小部件的新补全系统的支持在函数 Completion/Zsh/Command/_zftp 中提供，该函数应该已经与补全系统的其他函数一起安装，因此应自动可用。

---

26 用户贡献

26.1 说明

Zsh 源代码发行版包含许多由用户社区贡献的项目。 这些项目本身并不是 shell 的一部分，而且有些项目可能并不是每个 zsh 安装都可用。 这里记录了其中最重要的部分。 有关其他贡献项目（如 shell 函数）的文档，请查看函数源文件中的注释。

26.2 实用程序

26.2.1 访问在线帮助

键序 ESC h 通常被 ZLE 绑定为执行 run-help 小部件（参见 Zsh 行编辑器）。 这将调用 run-help 命令，并将当前输入行中的命令字作为其参数。 默认情况下，run-help 是 man 命令的别名，因此当命令字是 shell 内置命令或用户自定义函数时，调用 run-help 往往会失败。 通过重新定义 run-help 别名，可以改进 shell 提供的联机帮助。

helpfiles 工具位于发行版的 Util 目录中，是一个 Perl 程序，可用于处理 zsh 手册，为每个 shell 内置功能和许多其他 shell 功能生成单独的帮助文件。 可自动加载的

Functions/Misc 目录下的 run-help 函数会搜索这些帮助文件，并执行其他一些测试，为命令生成尽可能完整的帮助。

帮助文件默认安装在 /usr/share/zsh 或 /usr/local/share/zsh 的子目录下。

要使用 helpfiles 创建自己的帮助文件，请选择或创建一个存放各个命令帮助文件的目录。 例如，你可以选择 ~/zsh_help。 如果将 zsh 发行版解压到你的主目录，则可以使用以下命令：

```
mkdir ~/zsh_help
perl ~/zsh-5.9/Util/helpfiles ~/zsh_help
```

HELPDIR 参数用于告诉 run-help 从何处查找帮助文件。未设置时，将使用默认安装路径。 要使用自己的帮助文件集，请在启动文件中将其设置为相应的路径：

```
HELPDIR=~/zsh_help
```

要使用 run-help 函数，需要在 .zshrc 或类似的启动文件中添加如下行文：

```
unalias run-help
autoload run-help
```

请注意，要使 'autoload run-help' 生效，run-help 文件必须位于 fpath 数组中指定的目录下（参见 Shell 使用的参数）。 如果您安装的是标准 zsh，则应已存在该目录；如果没有，请将 Functions/Misc/run-help 复制到适当的目录。

26.2.2 重新编译函数

如果您经常编辑您的 zsh 函数，或定期更新您的 zsh 安装以跟踪最新开发进展，您可能会发现使用 zcompile 内置命令编译的函数摘要经常会比函数源文件过时。 这通常不是问题，因为 zsh 在加载函数时总是会查找最新的文件，但它可能会导致 shell 启动和函数加载速度变慢。 此外，如果摘要文件被显式地用作 fpath 的元素，zsh 将不会检查其源文件是否已更改。

Functions/Misc 中的 zrecompile 可自动加载函数可用于更新函数摘要。

zrecompile [-qt] [*name* ...]
zrecompile [-qt] -p *arg* ... [-- *arg* ...]

> 此功能会尝试查找 *.zwc 文件，并在至少一个原始文件比编译文件新的情况下自动重新编译它们。 只有当编译文件中存储的名称是全路径或相对于包含 .zwc 文件的目录时，此功能才会起作用。

在第一种形式中，每个 *name* 是一个编译文件的名称，或者是一个包含 `*.zwc` 文件的目录的名称,这些应该被检查。 如果没有给出参数，则使用 `fpath` 中的目录和 `*.zwc` 文件。

如果给出 `-t`，则不执行编译，但如果有文件需要重新编译，则返回状态为零（true），否则返回状态为非零（false）。 使用 `-q` 选项时，`zrecompile` 会静默输出。

如果不使用 `-t` 选项，所有需要重新编译的文件都编译成功，则返回状态为零；如果至少有一个文件编译失败，则返回状态为非零。

如果给出 `-p` 选项，则 *args* 将被解释为 `zcompile` 的一个或多个参数集，并用‘--’分隔。 例如:

```
zrecompile -p \
           -R ~/.zshrc -- \
           -M ~/.zcompdump -- \
           ~/zsh/comp.zwc ~/zsh/Completion/*/_*
```

如果 `~/.zshrc.zwc` 不存在或比 `~/.zshrc` 旧，则会将 `~/.zshrc` 编译成 `~/.zshrc.zwc`。编译后的文件将被标记为用于读取文件，而不是映射文件。对于 `~/.zcompdump` 和 `~/.zcompdump.zwc` 也是如此，但该编译文件会被标记为用于映射文件。最后一行将重新创建文件 `~/zsh/comp.zwc`，前提是任何与给定模式匹配的文件都比它新。

如果没有 `-p` 选项，`zrecompile` 不会创建不存在的函数的摘要，也不会向摘要中添加新函数。

下面的 shell 循环是为 `fpath` 中的所有函数创建函数摘要的方法示例，前提是你对目录有写入权限：

```
for ((i=1; i <= $#fpath; ++i)); do
  dir=$fpath[i]
  zwc=${dir:t}.zwc
  if [[ $dir == (.|..) || $dir == (.|..)/* ]]; then
    continue
  fi
  files=($dir/*(N-.))
  if [[ -w $dir:h && -n $files ]]; then
    files=(${${(M)files%/*/*}#/})
    if ( cd $dir:h &&
         zrecompile -p -U -z $zwc $files ); then
      fpath[i]=$fpath[i].zwc
    fi
  fi
done
```

-U 和 -z 选项适用于默认 zsh 安装 fpath 中的函数；对于个人函数目录，可能需要使用不同的选项。

一旦创建了摘要，并修改了 fpath 以引用它们，就可以通过运行不带参数的 zrecompile 来保持更新。

26.2.3 键盘定义

由于键盘、工作站、终端、模拟器和窗口系统可能存在大量组合，zsh 不可能为每种情况都提供内置的按键绑定。 Functions/Misc 中的 zkbd 工具可以帮助你快速创建适合你配置的按键绑定。

以自动加载函数或 shell 脚本的形式运行 zkbd：

```
zsh -f ~/zsh-5.9/Functions/Misc/zkbd
```

运行 zkbd 时，它首先会要求你输入终端类型；如果它提供的默认类型正确，只需按回车键即可。然后，它会要求你按下一些不同的键，以确定键盘和终端的特征； zkbd 如果发现任何异常，例如 Delete 键既不发送 ^H 也不发达 ^?，就会发出警告。

zkbd 读取的按键记录成名为 key 的关联数组的定义，并写入 HOME 或 ZDOTDIR 目录下 .zkbd 子目录中的文件。 文件名由 TERM、VENDOR 和 OSTYPE 参数组成，并用连字符连接。

你可以使用 'source' 或 '.' 命令将该文件读入 .zshrc 或其他启动文件，然后在 bindkey 命令中引用 key 参数，就像这样：

```
source ${ZDOTDIR:-$HOME}/.zkbd/$TERM-$VENDOR-$OSTYPE
[[ -n ${key[Left]} ]] && bindkey "${key[Left]}" backward-char
[[ -n ${key[Right]} ]] && bindkey "${key[Right]}" forward-char
# etc.
```

请注意，要使 'autoload zkbd' 正常工作，zkdb 文件必须位于 fpath 数组中指定的目录下（参见 Shell 使用的参数）。 如果您安装的是标准 zsh，则应该已经存在；如果没有，请将 Functions/Misc/zkbd 复制到适当的目录。

26.2.4 转储 shell 状态

有时您可能会在 shell 中遇到看似错误的问题，尤其是当您使用的是 zsh 测试版或开发版时。 通常情况下，将问题描述发送到 zsh 邮件列表（请参阅 邮件列表）就足够了，但有时 zsh 开发人员需要重新创建您的环境才能跟踪问题。

发行版 Util 目录中名为 reporter 的脚本就是为此目的而提供的。 (也可以 autoload reporter，但 reporter 默认并未安装在 fpath 中）。这个脚本输出了shell状态的详细转储，以另一个脚本的形式呈现，可以使用 'zsh -f' 读取，以重建该状态。

要使用 reporter，请使用‘.’命令将脚本读入 shell，并将输出重定向到文件中：

```
. ~/zsh-5.9/Util/reporter > zsh.report
```

您应该检查 zsh.report 文件中是否有密码等敏感信息，并在将脚本发送给开发人员之前手动删除它们。 此外，由于输出可能非常庞大，最好等开发人员询问这些信息后再发送。

您还可以使用 reporter 只转储 shell 状态的子集。 这对于首次创建启动文件有时很有用。 大多数 reporter 输出的内容都比启动文件通常需要的内容要详细得多，但 aliases、options 和 zstyles 状态可能会很有用，因为它们只包含了默认值之外的更改。 如果你创建了自己的键映射，bindings 状态可能会有用，因为 reporter 会转储键映射创建命令以及每个键映射的绑定。

按照自动化工具的惯例，如果使用 reporter 创建启动文件，则应编辑结果以删除不必要的命令。 请注意，如果您使用的是新的补全系统，则不应使用 reporter 将 functions 状态转储到启动文件中；而应使用 compdump 函数（参见 补全系统）。

reporter [*state* ...]

将当前 shell 状态的指定子集打印到标准输出。 *state* 参数可以是一个或多个：

all

输出下面列出的所有内容。

aliases

输出别名定义。

bindings

输出 ZLE 键映射和绑定。

completion

输出旧式 compctl 命令。 新的补全由 functions 和 zstyles 涵盖。

函数

输出自动加载和函数定义。

limits

输出 limit 命令。

选项

输出 setopt 命令。

样式

与 zstyles 相同。

变量

输出 shell 参数赋值，为任何环境变量的加上 export 命令。

zstyles

输出 zstyle 命令。

如果省略 *state*，则假定使用 all。

除 'all' 外，每个 *state* 都可以用任何前缀缩写，甚至一个字母；因此 a 与 aliases 相同，z 与 zstyles 相同，等等。

26.2.5 操作勾子函数

add-zsh-hook [-L | -dD] [-Uzk] *hook function*

如 函数 中特殊函数一节所述，有几个函数对 shell 来说是特殊的，它们会在 shell 执行过程中的特定时刻被自动调用。 每个函数都有一个关联数组，由在同一时刻被调用的函数名称组成；这些函数就是所谓的 '钩子函数'。 shell 函数 add-zsh-hook 提供了一种从数组中添加或删除函数的简单方法。

hook 是 chpwd、periodic、precmd、preexec、zshaddhistory、zshexit 或 zsh_directory_name 中的一个，即相关的特殊函数。 请注意 zsh_directory_name 的调用方式与其他函数不同，但仍可作为钩子进行操作。

function 是普通 shell 函数的名称。 如果没有给定选项，该函数将被添加到要在给定上下文中执行的函数的数组中。 函数将按照添加的顺序被调用。

如果给出 -L 选项，钩子数组的当前值将与 typeset 一起列出。

如果给出选项 -d，*function* 将从要执行的函数数组中删除。

如果给出选项 -D，*function* 将被视为一种模式，任何匹配的函数名称都将从要执行的函数数组中删除。

选项 -U、-z 和 -k 将作为 *function* 的参数传递给 autoload。 对于使用 zsh 提供的函数，选项 -Uz 是合适的。

add-zle-hook-widget [-L | -dD] [-Uzk] *hook widgetname*

正如 "特殊小部件" Zle 小部件 ）一节所述，有几个小部件名称对于行编辑器来说是特殊的，它们会在编辑过程中的特定点自动调用。 与函数钩不同的是，这些不使用

预定义的其他名称数组来在同一点调用; ; shell 函数 add-zle-hook-widget 维护一个类似的数组，并安排特殊小部件调用这些额外的小部件。

hook 是 isearch-exit、isearch-update、line-pre-redraw、line-init、line-finish、history-line-set 或 keymap-select 中的一个，分别对应 zle-isearch-exit 等特殊小部件。 这些特殊小部件的名称也可作为 *hook* 参数使用。

widgetname 是 ZLE 小部件的名称。 如果没有给定选项，这将被添加到要在给定钩子上下文中调用的小部件数组中。 小部件会按照添加的顺序被调用，并以

```
zle widgetname -Nw -f "nolast" -- "$@"
```

请注意，这意味着在调用小部件函数时，'WIDGET' 特殊参数会跟踪 *widgetname*，而不是跟踪相应特殊钩子小部件的名称。

如果给出选项 -d，*widgetname* 将从要执行的小部件数组中删除。

如果给出选项 -D，*widgetname* 将被视为一种模式，任何匹配的小部件名称都将从数组中删除。

如果 *widgetname* 在添加到数组时没有命名一个现有的小部件，则会假定一个同样命名为 *widgetname* 的 shell 函数将提供该小部件的实现。 因此，该名称被标记为自动加载，选项 -U、-z 和 -k 将作为 autoload 的参数传递给 autoload，与 add-zsh-hook 一样。 小部件也是通过'zle -N *widgetname*' 创建的，以便在第一次调用钩子时加载相应的函数。

widgetname 的数组目前保存在 zstyle 上下文中，每个 *hook* 上下文有一个，样式为'widgets'。 如果给出 -L 选项，这组样式将以 'zstyle -L' 列出。 这种实现方式可能会发生变化，只有调用 add-zle-hook-widget 添加至少一个小部件时，才会创建引用样式的特殊小部件，因此如果此函数用于任何钩子，那么所有钩子都应只通过此函数进行管理。

26.3 记住最近的目录

通过函数 cdr，可以从自动维护的列表中将工作目录更改为前一个工作目录。 它的概念类似于 pushd、popd 和 dirs 内置程序所控制的目录堆栈，但可配置性更强，而且由于它将所有条目都存储在文件中，因此可以跨会话维护，（默认情况下）也可以在当前会话的终端模拟器之间维护。 重复的条目会自动删除，因此列表反映的是每个目录最近一次使用的情况。

请注意，pushd 目录栈实际上不会被 cdr 修改或使用，除非按照下文配置部分所述进行配置。

26.3.1 安装

该系统通过一个钩子函数工作，每次目录发生变化时都会调用该函数。 要安装该系统，请自动加载所需的函数，并使用上述 add-zsh-hook 函数：

```
autoload -Uz chpwd_recent_dirs cdr add-zsh-hook
add-zsh-hook chpwd chpwd_recent_dirs
```

现在，每次以交互方式直接更改目录时，无论使用哪条命令，都会以,最近的优先,这样的顺序记住更改的目录。

26.3.2 用法

所有直接的用户交互都是通过 cdr 函数进行的。

cdr 的参数是一个数字 N，与第 N 个最近更改过的目录相对应。 1 指的是最近的目录；当前目录会被记住，但不会作为目标目录提供。 需要注意的是，如果打开了多个窗口，1 可能指的是另一个窗口中已更改的目录；为避免出现这种情况，可以使用基于每个终端的文件来存储目录，详见以下 recent-dirs-file 样式的描述。

如果你设置了下面描述的 recent-dirs-default 风格，那么当使用非数字参数或多个参数调用 cdr 时，它将与 cd 的行为相同。但无论你如何切换目录，最近目录列表仍将得到更新。

如果省略参数，则假定为 1。 这与 pushd 交换堆栈中最近的两个目录的行为类似。

如果已运行 compinit，则 cdr 的参数可以补全；建议使用菜单选择, 用：

```
zstyle ':completion:*:*:cdr:*:*' menu selection
```

允许你循环查看最近的目录；顺序会被保留，所以第一个选择是当前目录之前最近的目录。 此外，还建议使用 "verbose" 样式，以确保显示目录；该样式默认为打开，因此除非更改，否则无需执行任何操作。

26.3.3 选项

cdr 的行为可以通过以下选项进行修改。

-l

> 以缩写形式（即重新使用 ~ 替换）列出数字和相应的目录，每行一个。 这里的目录不加引号（只有在目录名包含换行符时才会出现问题）。 这是由补全系统使用的。

-r

> 将变量 reply 设置为当前目录集。 不会打印任何内容，也不会更改目录。

-e

可以编辑目录列表，每行一个。 该列表可任意编辑，但不进行正确性检查。 可以补全。 无需加引号（换行符除外，我对换行符没有任何同情心）；目录采用非缩写形式，包含绝对路径，即以 / 开头。 通常第一个条目应作为当前目录。

`-p '`*`pattern`*`'`

删除目录列表中任何与给定扩展 glob 模式匹配的条目；该模式需要加引号，以避免在命令行中立即展开。 该模式与列表中每个完全展开的文件名匹配；必须补全整个字符串，因此需要在末尾添加通配符（例如 `'*removeme*'`），以删除带有给定子字符串的条目。

如果输出到终端，函数将在修剪后打印新列表，并提示用户确认。 使用 `-P` 代替 `-p` 可以跳过输出和确认步骤。

26.3.4 配置

配置是通过样式机制来完成的，这一点大家应该很熟悉；如果不熟悉，请参阅 zsh/zutil 模块 中 `zstyle` 命令的描述。设置样式的上下文应为 `':chpwd:*'`，以防将来上下文的含义被扩展，例如：

```
zstyle ':chpwd:*' recent-dirs-max 0
```

将 `recent-dirs-max` 样式的值设置为 0。实际上，该样式名称足够具体，使用 '*' 上下文就可以了。

`recent-dirs-insert` 是个例外，它是由补全系统专用的，因此有通常的补全系统上下文（`':completion:*'`，如果不需要更具体的内容），不过 `'*'` 在实际使用中也应该没问题。

`recent-dirs-default`

如果为 "true"，并且该命令期待的是一个最近的目录索引，而参数不只一个或参数不是整数，则转为 "cd"。 这样，懒人就可以只使用一条命令来更改目录。 补全也能识别这一点；关于使用该选项时如何控制补全，请参阅 recent-dirs-insert。

`recent-dirs-file`

保存目录列表的文件。 默认值为 `${ZDOTDIR:-$HOME}/.chpwd-recent-dirs`，也就是说，除非你将变量 ZDOTDIR 设为指向其他地方，否则该文件将保存在你的主目录中。 目录名以 `$'...'` 的引号形式保存，因此文件中的每一行都可以作为参数直接提供给 shell。

该样式的值可以是一个数组。 在这种情况下，列表中的第一个文件将始终用于保存目录，而其他文件则保持不变。 读取最近目录列表时，如果第一个文件中的条目数少于最大值，数组中后面文件的内容将被追加，并从显示的列表中删除重复的文件。 两个文件的内容不会一起排序，即第一个文件中的所有条目都会首先显示。 特

殊值 + 可以出现在列表中，表示在该点应读取默认文件。 这样就可以达到如下效果：

```
zstyle ':chpwd:*' recent-dirs-file \
~/.chpwd-recent-dirs-${TTY##*/} +
```

最近的目录是从一个根据终端编号的文件中读取的。 如果条目不足，则从默认文件中补充。

使用 zstyle -e 可以使目录在运行时可配置：

```
zstyle -e ':chpwd:*' recent-dirs-file pick-recent-dirs-file
pick-recent-dirs-file() {
  if [[ $PWD = ~/text/writing(|/*) ]]; then
    reply=(~/.chpwd-recent-dirs-writing)
  else
    reply=(+)
  fi
}
```

在本例中，如果当前目录是 ~/text/writing 或其下的一个目录，则使用特殊文件保存最近的目录，否则使用默认文件。

recent-dirs-insert

补全时使用。 如果 recent-dirs-default 为 true，则将其设置为 true，会在命令行中插入实际目录，而不是目录索引；这与使用相应索引的效果相同，但会使历史记录更清晰，行更易于编辑。 在此设置下，如果参数的一部分已经输入，则会执行正常的目录补全，而不是最近的目录补全；这是因为最近的目录补全预计会以菜单方式条目执行。

如果样式的值是 always，那么只会补全最近的目录；在这种情况下，当你想补全其他目录时，请使用 cd 命令。

如果值为 fallback，将首先尝试最近的目录，如果最近目录的补全未能找到匹配，则执行正常目录的补全。

最后，如果值为 both，则会同时显示两组补全；可以使用常用的标记机制来区分结果，最近的目录会被标记为 recent-dirs。 需要注意的是，插入的最近目录会根据情况用目录名缩写。

recent-dirs-max

要保存到文件中的最大目录数。 如果为零或负数，则没有最大值。 默认值为 20。 请注意，这包括当前目录，因为当前目录不提供，所以提供的最高目录数比最大值少一个。

recent-dirs-prune

该样式是一个数组，用于确定哪些目录应（或不应）添加到最近列表中。 数组的元素可以包括:

`parent`

从最近列表中剪除父目录（更准确地说，是祖目录）。 如果存在，直接向下更改任意数量的目录会导致当前目录被覆盖。 例如，从 ~pws 改为 ~pws/some/other/dir 会导致 ~pws 不再留在最近目录堆栈中。 这只适用于对后代目录的直接更改；列表中较早的目录不会被剪枝。 例如，从 ~pws/yet/another 更改为 ~pws/some/other/dir 不会导致 ~pws 被剪除。

`pattern:`*pattern*

为不应添加到最近列表（如果尚未添加）的目录提供一个 zsh 模式。 此元素可以重复添加到不同的模式。 例如，`'pattern:/tmp(|/*)'` 会阻止 `/tmp` 或其子目录被添加。 对于这些模式，`EXTENDED_GLOB` 选项始终处于开启状态。

`recent-dirs-pushd`

如果设置为 true，`cdr` 将使用 `pushd` 而不是 `cd` 来更改目录，因此目录将保存在目录栈中。 由于目录栈与本文件使用的机制所保存的文件列表是完全独立的，因此没有明显的理由这样做。

26.3.5 与动态目录命名一起使用

可以使用动态目录名语法引用最近的目录, 通过使用提供的函数 `zsh_directory_name_cdr` 勾子：

```
autoload -Uz add-zsh-hook
add-zsh-hook -Uz zsh_directory_name zsh_directory_name_cdr
```

完成后，`~[1]` 将指向 `$PWD` 以外的最新目录，依此类推。 在 `~[...` 之后补全也同样有效。

26.3.6 目录处理细节

这部分内容是为好奇或困惑的用户准备的，大多数用户并不需要了解这些信息。

最近的目录会立即保存到文件中，因此会跨(across)会话保留。 需要注意的是，目前没有应用文件锁定：列表会在交互式命令中立即更新，而不会在其他地方更新（与历史记录不同），并且假定你一次只会在一个窗口中更改目录。 这在共享账户中并不安全，但无论如何，当别人背着你更改不同的目录集时，该系统的作用就很有限了。

为了让这个过程更安全，只有通过命令行直接或间接地（但不是通过子shell、evals、traps、补全函数等）进行的目录更改才会被保存。Shell函数应该使用 `cd -q` 或 `pushd`

-q 来避免副作用，如果目录的更改在命令行上是不可见的。更多详情请参考函数 chpwd_recent_dirs 的内容。

26.4 目录的简略动态引用

动态目录命名系统在 文件名扩展 的 **动态命名目录** 小节中进行了描述。 其中，~[...] 的引用由钩子机制找到的函数展开。

贡献函数 zsh_directory_name_generic 提供了一个系统，用户只需编写少量新代码即可引用目录。 它支持目录命名的所有三个标准接口：从名称到目录的转换、反向转换以查找简短名称，以及名称的补全。

该函数的主要特点是采用类似路径的语法，将多级缩写以 ":" 分隔。 例如， ~[g:p:s] 可以指定：

g

: git 区域的顶级目录。 第一个组件必须匹配，否则函数将返回，指示应尝试另一个目录名钩子函数。

p

: git 区域内的项目名称。

s

: 该项目中的源区。

这样就可以将长层次结构的引用折叠成非常紧凑的形式，尤其是在磁盘不同区域的层次结构相似的情况下。

名称组件可以补全：如果在补全列表顶部显示说明，则包括之前组件扩展的路径，而单个补全的说明则显示它将添加的路径段。 由于补全系统知道动态目录名称机制，因此无需为此进行额外配置。

26.4.1 用法

要使用该函数，首先要针对具体情况定义一个封装函数。 我们假设它是自动加载的。 它可以有任何名称，但我们将其称为 zdn_mywrapper。 该封装函数将定义各种变量，然后使用封装函数获得的相同参数调用该函数。 下面将介绍这种配置。

然后安排封装器作为 zsh_directory_name 钩子运行：

```
autoload -Uz add-zsh-hook zsh_directory_name_generic zdn_mywrapper
add-zsh-hook -U zsh_directory_name zdn_mywrapper
```

26.4.2 配置

封装函数应定义一个本地关联数组 zdn_top。 或者，也可以使用名为 mapping 的样式来设置。 样式的上下文为 :zdn:*wrapper-name*，其中 *wrapper-name* 是调用 zsh_directory_name_generic 的函数；例如

```
zstyle :zdn:zdn_mywrapper: mapping zdn_mywrapper_top
```

该关联数组的键与名称的第一个组件相对应。 值是匹配的目录。 它们可以有一个可选的后缀，后缀是一个斜线，后面跟一个冒号和一个格式相同的变量名，以便给出下一个组件。 (冒号前的斜线是为了消除歧义，这种情况，驱动器路径中需要冒号。 除此之外，没有任何语法可以将其转义，因此不支持名称以冒号开头的路径组件）。 一个特殊的组件 :default: 以 /:*var* 的形式指定了一个变量（路径部分将被忽略，因此通常为空），如果没有为路径指定变量，下一个组件将使用该变量。 zdn_top 中引用的变量格式与 zdn_top 本身相同，但包含相对路径。

例如,

```
local -A zdn_top=(
  g   ~/git
  ga  ~/alternate/git
  gs  /scratch/$USER/git/:second2
  :default: /:second1
)
```

这指定了被称为 ~[g:...] 或 ~[ga:...] 或 ~[gs:...]的目录的行为。 后面的路径组件是可选的；在这种情况下，~[g] 会扩展为 ~/git，以此类推。 gs 扩展为 /scratch/$USER/git，并使用关联数组 second2 匹配第二个组件；g 和 ga 使用关联数组 second1 匹配第二个组件。

在将名称扩展为目录时，如果第一个组件不是 g、ga 或 gs，则不会出错；函数只会返回 1，以便尝试后面的钩子函数。 不过，匹配第一个组件会提交函数，因此如果后面的组件不匹配，就会打印错误信息（尽管这仍不能阻止后面的钩子执行）。

对于第一个组件之后的组件，预计会使用相对路径，但注意仍可能出现多个层级。 下面是 second1 的示例：

```
local -A second1=(
  p   myproject
  s   somproject
  os  otherproject/subproject/:third
)
```

从 zdn_top 中找到的路径会用匹配目录进行扩展，因此 ~[g:p] 变成了 ~/git/myproject。 中间的斜线是自动添加的（不可能让后面的组件修改已匹配目录的名

称）。 只有 os 为第三个组件指定了变量，而且没有 :default:，因此使用 ~[g:p:x] 或 ~[ga:s:y] 这样的名称是错误的，因为没有地方可以查找 x 或 y。

关联数组需要在该函数中可见；因此，为了避免冲突，泛型函数使用了以 _zdn_ 开头的内部变量名。 需要注意的是，变量 reply 需要传回 shell，因此不应在调用函数中本地化。

该函数并不测试由组件组成的目录是否实际存在；这使得系统可以在自动挂载的文件系统中运行。 试图使用不存在的目录时，命令会出现错误，这足以说明问题所在。

26.4.3 补全示例

以下是上述代码定义的示例函数的完整虚构但可用的自动加载定义。 因此， ~[gs:p:s] 会展开为 /scratch/$USER/git/myscratchproject/top/srcdir（$USER 也会展开）。

```
local -A zdn_top=(
  g   ~/git
  ga  ~/alternate/git
  gs  /scratch/$USER/git/:second2
  :default: /:second1
)

local -A second1=(
  p   myproject
  s   somproject
  os  otherproject/subproject/:third
)

local -A second2=(
  p   myscratchproject
  s   somescratchproject
)

local -A third=(
  s   top/srcdir
  d   top/documentation
)

# autoload not needed if you did this at initialisation...
autoload -Uz zsh_directory_name_generic
zsh_directory_name_generic "$@
```

也可以使用全局关联数组（以适当的方式命名），并将封装函数的上下文样式设置为引用到这个数组。 这样，您的设置代码将包含以下内容：

```
typeset -A zdn_mywrapper_top=(...)
# ... and so on for other associative arrays ...
zstyle ':zdn:zdn_mywrapper:' mapping zdn_mywrapper_top
autoload -Uz add-zsh-hook zsh_directory_name_generic zdn_mywrapper
add-zsh-hook -U zsh_directory_name zdn_mywrapper
```

而函数 zdn_mywrapper 将只包含以下内容：

```
zsh_directory_name_generic "$@"
```

26.5 从版本控制系统中收集信息

在很多情况下，从版本控制系统（VCS）（如 subversion、CVS 或 git）中自动获取信息并提供给用户（可能是在用户提示符中）是件好事。例如，这样你就能立即知道当前在哪个分支上。

为此，您可以使用 vcs_info 函数。

系统支持以下版本控制系统，并显示了系统内使用的简称：

Bazaar (bzr)

https://bazaar.canonical.com/

Codeville (cdv)

http://freecode.com/projects/codeville/

Concurrent Versioning System (cvs)

https://www.nongnu.org/cvs/

Darcs (darcs)

http://darcs.net/

Fossil (fossil)

https://fossil-scm.org/

Git (git)

https://git-scm.com/

GNU arch (tla)

https://www.gnu.org/software/gnu-arch/

Mercurial (hg)

https://www.mercurial-scm.org/

Monotone (mtn)

https://monotone.ca/

Perforce (p4)

https://www.perforce.com/

Subversion (svn)

https://subversion.apache.org/

SVK (svk)

https://svk.bestpractical.com/

此外，还支持补丁管理系统 quilt (https://savannah.nongnu.org/projects/quilt)。详情请参见下面的 Quilt 支持。

载入 vcs_info:

```
autoload -Uz vcs_info
```

它可以在任何现有提示符中使用，因为它不需要任何特定的 $psvar 条目。

26.5.1 快速开始

要快速使用此功能（包括颜色），可以执行以下操作（假设已正确加载 vcs_info - 参见上文）：

```
zstyle ':vcs_info:*' actionformats \
    '%F{5}(%f%s%F{5})%F{3}-%F{5}[%F{2}%b%F{3}|%F{1}%a%F{5}]%f '
zstyle ':vcs_info:*' formats       \
    '%F{5}(%f%s%F{5})%F{3}-%F{5}[%F{2}%b%F{5}]%f '
zstyle ':vcs_info:(sv[nk]|bzr):*' branchformat '%b%F{1}:%F{3}%r'
precmd () { vcs_info }
PS1='%F{5}[%F{2}%n%F{5}] %F{3}%3~ ${vcs_info_msg_0_}%f%# '
```

很明显，最后两行是用来演示的。您需要从 precmd 函数中调用 vcs_info。调用完成后，您需要在提示符中为 '${vcs_info_msg_0_}' **加单引号**。

要像这样在提示符中直接使用 '${vcs_info_msg_0_}' ，需要启用 PROMPT_SUBST 选项。

现在从命令行调用 vcs_info_printsys 工具：

```
% vcs_info_printsys
## list of supported version control backends:
## disabled systems are prefixed by a hash sign (#)
bzr
cdv
cvs
darcs
fossil
git
hg
mtn
p4
svk
svn
tla
## flavours (cannot be used in the enable or disable styles; they
## are enabled and disabled with their master [git-svn -> git])
## they *can* be used in contexts: ':vcs_info:git-svn:*'.
git-p4
git-svn
hg-git
hg-hgsubversion
hg-hgsvn
```

您可能不需要所有这些，因为运行代码来检测您不使用的系统毫无意义。 因此，有一种方法可以完全禁用某些后端：

```
zstyle ':vcs_info:*' disable bzr cdv darcs mtn svk tla
```

您也可以从列表中挑选几项，并只启用那几项：

```
zstyle ':vcs_info:*' enable git cvs svn
```

如果你在执行完这些命令后重新运行 vcs_info_printsys，你会看到 disable 样式中列出的后端（或 enable 样式中没有列出的后端（如果你使用了））以哈希符号标记为禁用。 这意味着将 **完全** 跳过对这些系统的检测。不会浪费时间。

26.5.2 配置

可通过 zstyle 配置 vcs_info 功能。

首先是我们工作的上下文：

```
:vcs_info:vcs-string:user-context:repo-root-name
```

vcs-string

是下列之一：`git`、`git-svn`、`git-p4`、`hg`、`hg-git`、`hg-hgsubversion`、`hg-hgsvn`、`darcs`、`bzr`、`cdv`、`mtn`、`svn`、`cvs`、`svk`、`tla`、`p4` 或 `fossil`。在 Quilt 模式（详见 Quilt 支持）下，后跟‘`.quilt-`*quilt-mode*’；在钩子激活时，后跟‘`+`*hook-name*’（详见 vcs_info 中的勾子）。

目前，quilt 模式下的钩子不会添加‘`.quilt-`*quilt-mode*’信息。 这点未来可能会有所改变。

user-context

是一个可自由配置的字符串，用户可将其指定为 `vcs_info`的第一个参数（参见下文的描述）。

repo-root-name

是您希望匹配样式的版本库名称。因此，如果你想对 `/usr/src/zsh` 进行特定设置，并将其作为 CVS 检出，你可以将 *repo-root-name* 设置为 `zsh`，这样就可以了。

vcs-string 有三个特殊值： 第一种名为 `-init-`，在尚未决定使用哪种 VCS 后端时有效。第二个是 `-preinit-`；它在 `vcs_info` **之前**，初始化数据导出变量时， 使用。第三个特殊值是 `formats`，用于 `vcs_info_lastmsg` 查找样式。

repo-root-name 的初始值为 `-all-`，一旦知道实际名称，就会立即替换为该名称。只有在定义 `formats`、`actionformats` 或 `branchformat` 样式时才会使用这部分上下文，因为只有在这些情况下 *repo-root-name* 才会被正确设置。对于其他样式，只需使用 `'*'` 代替即可。

user-context 有两个预定义值：

`default`

如果没有指定，则使用

`command`

被 vcs_info_lastmsg 用于查找其样式

当然，您也可以使用 `':vcs_info:*'` 一次匹配所有用户上下文中的所有 VCS。

这是对所有查询样式的描述。

`formats`

格式列表，用于不使用 actionformats 时（大多数情况下）。

actionformats

格式列表，用于当前版本库中正在进行的特殊操作，如交互式重置或合并冲突。

branchformat

在上述 formats 和 actionformats 样式中，有些后端不仅会用分支名称替换 %b，还会用版本号替换 %b。这种样式允许你修改该字符串的外观。

nvcsformats

当我们没有检测到当前目录的版本控制系统或 vcs_info 被禁用时，就会设置这些"格式"。如果你想让 vcs_info 完全接管提示符的生成，这一点非常有用。为此，你可以使用PS1='${vcs_info_msg_0_}' 这样的方法。

hgrevformat

hg 同时使用哈希值和版本号来引用版本库中的特定变更集。使用这个样式，你可以格式化版本号字符串（参见 branchformat），使其中之一或两者。只有当 get-revision 为 true 时才有用。需要注意的是，由于每次提示符执行 hg 多于一次会导致速度过慢，因此无法使用完整的 40 个字符的修订版本 ID（使用 use-simple 选项时除外）；你可以使用钩子自定义此行为。

max-exports

定义 vcs_info 将设置的 vcs_info_msg_*_ 变量的最大数量。

enable

要使用的后端列表。在 -init- 上下文中进行检查。如果该列表包含名为 NONE 的项，则不会使用任何后端，vcs_info 也不会做任何操作。如果该列表包含 ALL，vcs_info 将使用所有已知的后端。只有 enable 中包含 ALL 时，disable 样式才会生效。ALL 和 NONE 不区分大小写。

disable

不想让 vcs_info 测试版本库的 VCS 列表（在 -init- 上下文中也会检查）。仅在 enable 包含 ALL 时使用。

disable-patterns

根据 $PWD 检查的模式列表。如果一个模式匹配，vcs_info 将被禁用。此样式在 :vcs_info:-init-:*:-all- 上下文中进行检查。

假设 ~/.zsh 是一个受版本控制的目录，您不希望 vcs_info 在其中处于活动状态，请执行此操作：

```
zstyle ':vcs_info:*' disable-patterns "${(b)HOME}/.zsh(|/*)"
```

use-quilt

如果启用，quilt 支持代码将在 '插件' 模式下激活。 有关详情，请参阅 Quilt 支持。

quilt-standalone

如果启用，在指定目录中如果没有激活 VCS，将尝试进行 '独立' 模式检测。有关详情，请参阅 Quilt 支持。

quilt-patch-dir

覆盖 $QUILT_PATCHES 环境变量的值。有关详情，请参阅 Quilt 支持。

quiltcommand

在 quilt 支持中调用 quilt 本身时，该样式的值将被用作命令名称。

check-for-changes

启用该样式后，当工作目录中有未提交的更改时，将显示 %c 和 %u 格式转义符。可以通过 stagedstr 和 unstagedstr 样式控制这些转义字符显示的字符串。目前支持此选项的后端只有 git、hg 和 bzr（后两者仅支持未提交的）。

要使用 hg 后端对该样式进行评估，需要设置 get-revision 样式，并取消设置 use-simple 样式。后者是默认设置，前者则不是。

使用 bzr 后端时，只有设置了 use-server 样式，***lightweight checkouts*** 才会采用这种样式。

需要注意的是，如果启用该样式，所采取的操作可能会很昂贵（可能会很慢，这取决于当前版本库有多大）。 因此，默认情况下是禁用的。

check-for-staged-changes

该样式与 check-for-changes 类似，但它从不检查工作树文件，只检查 .${vcs} 目录中的元数据。 因此，该样式只初始化 %c 转义符（使用 stagedstr），而不初始化 %u 转义符。 该样式比 check-for-changes 更快。

在 git 后端，该样式会检查索引中的更改。 其他后端目前还未实现此样式。

这个样式默认禁用。

stagedstr

如果存储库中有暂存的更改，该字符串将用于 %c 转义。

unstagedstr

如果版本库中有未暂存的更改，该字符串将用于 %u 转义。

command

这种样式会导致 vcs_info 使用所提供的字符串作为 VCS 的二进制命令。需要注意的是，在 ':vcs_info:*' 中设置此参数并不是一个好主意。

如果该样式的值为空（默认值），则使用的二进制名是正在使用的后端名称（例如 svn 用于 svn 版本库）。

在查找此样式时，上下文中的 repo-root-name 部分总是默认的 -all-。

例如，这种样式可用于使用非默认安装目录中的二进制文件。假设 git 安装在 /usr/bin，但系统管理员在 /usr/local/bin 安装了更新的版本。与其改变 $PATH 参数的顺序，不如这样做：

```
zstyle ':vcs_info:git:*:-all-' command /usr/local/bin/git
```

use-server

Perforce 后端（p4）使用它来决定是否要联系 Perforce 服务器，以查明某个目录是否由 Perforce 管理。 这是唯一可靠的方法，但如果找不到服务器名称，就会有延迟的风险。 如果无法联系到服务器（更具体地说，就是描述服务器的 *host*:*port* 对），其名称就会被放入关联数组 vcs_info_p4_dead_servers ，并且在会话期间不会再被联系，直到被手动删除。 如果不设置此样式，只有将环境变量 P4CONFIG 设置为文件名，并且在每个 Perforce 客户端的根目录中都有相应的文件时，才能使用 p4 后端。 详情请参见函数 VCS_INFO_detect_p4 中的注释。

Bazaar 后端（bzr）使用此功能允许就轻量级检出联系服务器，参见 check-for-changes 样式。

use-simple

如果有两种不同的信息收集方式，可以通过将此样式设置为 true 来选择更简单的一种；默认情况下使用不那么简单的代码，这可能会慢很多，但在所有可能的情况下都可能更准确。bzr、hg 和 git 后端都使用这种样式。对于 hg，它将调用外部 hexdump 程序来解析二进制 dirstate 缓存文件；该方法不会返回本地版本号。

get-revision

如果设置为 true，vcs_info 会额外计算版本库工作树的修订版本（目前适用于 git 和 hg 后端，这类信息并不总是很重要）。对于 git，当前已检出提交的哈希值可通过 %i 扩展获得。对于 hg，则可通过 %i 获取本地版本号和相应的全局哈希值。

get-mq

如果设置为 true，hg 后端将查找 Mercurial Queue (mq) 补丁目录。信息将通过 '%m' 替换提供。

get-bookmarks

如果设置为 true，hg 后端将尝试获取当前书签的列表。这些书签将通过 '%m' 替换获得。

默认设置是生成一个以逗号分隔的列表，其中包含所有指向当前已检出修订版的书签名称。 如果某个书签处于活动状态，则其名称后缀星号，并置于列表首位。

`use-prompt-escapes`

决定是否假设 `vcs_info` 汇编的字符串包含提示符转义。(被 `vcs_info_lastmsg` 使用) 。

`debug`

启用调试输出以跟踪可能出现的问题。目前只有 `vcs_info` 的钩子系统使用这种样式。

`hooks`

定义钩子函数名称的样式列表。详情请参阅下面的 vcs_info 中的勾子 。

`patch-format`
`nopatch-format`

这对样式格式化了 `%m` 扩展在 `git` 和 hg 后端 formats 和 actionformats 中使用的补丁信息。 该值受下文所述的某些 % 扩展的限制。 扩展后的值将以 `${backend_misc[patches]}` 的形式出现在全局 `backend_misc` 数组中。(如果使用了 `set-patch-format` 钩子) 。

`get-unapplied`

该布尔样式控制后端是否应尝试收集未应用补丁的列表 (例如 Mercurial 队列补丁) 。

由 `quilt`、hg 和 `git` 后端使用。

在所有上下文中，这些样式的默认值是:

`formats`

" (%s)-[%b]%u%c-"

`actionformats`

" (%s)-[%b|%a]%u%c-"

`branchformat`

"%b:%r" (for bzr, svn, svk and hg)

`nvcsformats`

    ""

`hgrevformat`

    "%r:%h"

`max-exports`

    2

`enable`

    ALL

`disable`

    (empty list)

`disable-patterns`

    (empty list)

`check-for-changes`

    false

`check-for-staged-changes`

    false

`stagedstr`

    (string: "S")

`unstagedstr`

    (string: "U")

`command`

    (empty string)

`use-server`

    false

`use-simple`

    false

`get-revision`

false

`get-mq`

true

`get-bookmarks`

false

`use-prompt-escapes`

true

`debug`

false

`hooks`

(empty list)

`use-quilt`

false

`quilt-standalone`

false

`quilt-patch-dir`

empty - use `$QUILT_PATCHES`

`quiltcommand`

quilt

`patch-format`

backend dependent

`nopatch-format`

backend dependent

`get-unapplied`

false

在正常的 formats 和 actionformats 中，会进行以下替换：

%s

使用的 VCS（git、hg、svn 等）。

%b

有关当前分支的信息。

%a

描述操作的标识符。只有在 actionformats 中才有意义。

%i

当前版本号或标识符。对于 hg，可使用 hgrevformat 样式自定义输出。

%c

如果版本库中存在阶段性变更，则使用 stagedstr 样式中的字符串。

%u

如果版本库中存在未暂存的变更，则使用 unstagedstr 样式中的字符串。

%R

版本库的基本目录。

%r

版本库名称。如果 %R 是 /foo/bar/repoXY，%r 就是 repoXY。

%S

版本库中的一个子目录。如果 $PWD 是 /foo/bar/repoXY/beer/tasty，%S 就是 beer/tasty。

%m

"杂项" 替换。后台可自行决定将此替换扩展为哪些内容。

hg 和 git 后端使用此扩展来显示补丁信息。 hg 从 mq 扩展获取补丁信息；git 从正在进行的 rebase 和 cherry-pick 操作以及 stgit 扩展获取补丁信息。 patch-format 和 nopatch-format 样式控制生成的字符串。 前者在补丁队列中至少有一个补丁被应用时使用，后者在其他情况下使用。

hg 后端还会在此扩展中显示书签信息(除了 mq 信息外)。 请参阅 get-mq 和 get-bookmarks 样式。 这两种样式可同时启用。 如果同时启用这两种样式，生成的两个字符串都将以分号分隔（目前无法自定义）。

quilt "独立" 后端，会将此扩展设置为与 %Q 扩展相同的值。

%Q

Quilt 系列信息。 当使用 quilt 时（无论是在 "插件" 模式下还是作为 "独立" 后端），该扩展项将被设置为 quilt 系列的 patch-format 字符串。 set-patch-format 钩子和 nopatch-format 样式将被保留。

有关详细信息，请参阅下面的 Quilt 支持。

在 branchformat 中，这些替换已经完成：

%b

分支名称。对于 hg，分支名称可以包括主题名称。

%r

当前版本号或 hg 的 hgrevformat 样式。

在 hgrevformat 中，这些替换将被完成：

%r

当前本地修订号。

%h

当前的全局修订标识符。

在 patch-format 和 nopatch-format 中，这些替换将被完成：

%p

最上层应用补丁的名称；可由 applied-string 钩子覆盖。

%u

未应用的补丁数量；可通过 unapplied-string 钩子覆盖。

%n

已应用补丁的数量。

%c

未应用的补丁数量。

%a

所有补丁的数量 (%a = %n + %c)。

%g

活动 mq 守卫（hg 后端）的名称。

%G

活动 mq 守卫（hg 后台）的数量。

并非所有 VCS 后端都必须支持所有替换。对于 nvcsformats，根本不会进行替换，它只是一个字符串。

26.5.3 奇特现象

如果要使用 formats中的 %b（粗体关闭）提示符扩展（它会扩展 %b 本身），请使用 %%b。这将导致 vcs_info 扩展用 %b 替换 %%b ，这样 zsh 的提示符扩展机制可以处理它。同样，要从 branchformat 传递 %b，请使用 %%%%b。很抱歉给您带来不便，但这是无法轻易避免的。幸运的是，我们与很多提示符扩展并不冲突，因此只需要在这些情况下才这样做。

当 gen-applied-string, gen-unapplied-string 和 set-patch-format 钩子中的一个被定义时，将 %-转义（ ('foo=${foo//'%'/%%}') 应用到用于提示符的插值是这些钩子（共同）的责任；当这些钩子都未被定义时，vcs_info 将自行处理转义。 我们对这种耦合感到遗憾，但这是向后兼容所必需的。

26.5.4 Quilt 支持

Quilt 并不是一个版本控制系统，因此并没有作为后端实现。它可以帮助跟踪一系列补丁。人们可以用它来在软件包上顶部保存一组他们想要使用的修改（这与软件包的构建过程紧密结合 - Debian 项目就为大量软件包做了这项工作）。Quilt 还能帮助个人开发者在真正的版本控制系统上跟踪自己的补丁。

vcs_info 集成试图通过两种略有不同的操作模式（'addon' 模式和 'standalone' 模式）来支持两种使用 quilt 的方式。

Quilt 集成默认为关闭；要启用它，请设置 use-quilt 样式，并在 formats 或 actionformats 样式中添加 %Q：

```
zstyle ':vcs_info:*' use-quilt true
```

从 Quilt 支持代码中查找的样式包括上下文中 *vcs-string* 部分的 '.quilt-*quilt-mode*'，其中 *quilt-mode* 是 addon 或 standalone。 例如 :vcs_info:git.quilt-addon:default:*repo-root-name*。

要使 'addon' 模式生效，vcs_info 必须已经检测到控制该目录的真正版本控制系统。如果是这种情况，则需要找到一个存放 quilt 补丁的目录。该目录可通过 'QUILT_PATCHES' 环境变量进行配置。如果存在该变量，则使用其值，否则假定使用'patches' 值。$QUILT_PATCHES的值可以用 'quilt-patch-dir' 样式覆盖。(注：您可以通过 post-quilt 钩子使用 vcs_info 来保持 $QUILT_PATCHES 的值始终正确）。

当找到相关目录时，quilt 会被认为处于活动状态。为了收集更多信息，vcs_info 会查找名为 '.pc' 的目录；Quilt 会使用该目录来跟踪其当前状态。如果该目录不存在，我们就知道 quilt 还没有对工作目录做任何操作（还没有打补丁）。

如果补丁已应用，vcs_info 将尝试找出哪些补丁已应用。如果想知道某个系列中哪些补丁尚未应用，则需要在适当的上下文中激活 get-unapplied 样式。

vcs_info 允许对如何展示收集到的信息进行非常详细的控制（参见 配置 和 vcs_info 中的勾子），所有这些都将在下文中详细介绍。请注意，还有许多其他补丁跟踪系统是在特定版本控制系统之上运行的（如 stgit 用于 *git*，或 mq 用于 *hg*）；此类系统的配置方式通常与 *quilt* 支持的方式相同。

如果 *quilt* 支持在 'addon' 模式下工作，生成的字符串可作为简单的格式替换（准确地说，是 %Q），可用于 formats 和 actionformats；详情见下文）。

另一方面，如果支持代码是在 'standalone' 模式下工作，vcs_info 将假装 quilt 是一个真正的版本控制系统。这意味着版本控制系统标识符（否则会是类似于'svn' 或 'cvs'）将被设置为 '-quilt-'。这对所使用的样式上下文有影响，在该样式上下文中，该标识符是第二个元素。vcs_info 将为 "版本库 "的根目录填入一个合适的值，包含被子状态信息的字符串将作为 'misc' 替换（以及 %Q 以兼容 'addon' 模式）。

剩下要讨论的是如何检测 'standalone' 模式。检测本身就是对目录的一系列搜索。你可以在每个不受版本控制的目录中一直启用这种检测。如果你知道只有一组有限的目录树，你希望 vcs_info 在 'standalone' 模式下尝试查找 Quilt 以最小化每次调用 vcs_info 时的搜索量，有几种方法可以实现这一目的：

基本上，'standalone' 模式检测是由一种名为 'quilt-standalone' 的样式控制的。这是一个字符串样式，其值可以产生不同的效果。最简单的值是: 'always' 表示每次运行 vcs_info 时都进行检测，'never' 表示完全关闭检测。

如果 quilt-standalone 的值是其他值，则会有不同的解释。如果值是标量变量的名称，则会检查该变量的值，并以上述 'always'/'never' 的方式使用该值。

如果 quilt-standalone 的值是一个数组，则该数组的元素将用作您希望激活检测的目录名。

如果 quilt-standalone 是一个关联数组，则键将作为您希望检测激活的目录名，但前提是相应的值是字符串 'true'。

最后但并非最不重要的一点是，如果 quilt-standalone 的值是一个函数的名称，那么该函数将不带参数地被调用，而返回值则决定检测是否应被激活。返回值为'0' 的表示为 "true"；返回值为非零的表示为 "false"。

注意，如果同时存在名为 quilt-standalone 的函数和变量，则函数优先。

26.5.5 函数说明（公共 API）

vcs_info [*user-context*]

> 主函数，用于运行所有后端并将所有数据汇总到 ${vcs_info_msg_*_} 中。如果想在提示符中包含最新信息（参见下面的 变量说明），就需要从 precmd 调用该函数。 如果给定了参数，则将在样式上下文的 *user-context* 字段中使用该字符串，而不是 default。

vcs_info_hookadd

> 向给定钩子静态注册多个函数。钩子需要作为第一个参数给出；接下来是要注册到钩子的钩子函数名称列表。此处应省略前缀 '+vi-'。详情请参阅下面的 vcs_info 中的勾子。

vcs_info_hookdel

> 从给定钩子中删除钩子函数。钩子需要作为第一个非选项参数给出；接下来是要从钩子中取消注册的钩子函数名称列表。如果第一个参数为 '-a'，则 所有 出现的函数都将被取消注册。否则，只删除最后一次出现的函数（如果一个函数在钩子上注册了不止一次）。此处应省略前缀 '+vi-'。 详情请参阅下面的 vcs_info 中的勾子。

vcs_info_lastmsg

> 输出 ${vcs_info_msg_*_} 的当前值。 会考虑 ':vcs_info:formats:command:-all-' 中 use-prompt-escapes 样式的值。它还只打印 max-exports 值。

vcs_info_printsys [*user-context*]

> 打印所有支持的版本控制系统列表。可用于查找可能的上下文（以及其中哪些已启用）或 disable 样式的值。

vcs_info_setsys

> 初始化 vcs_info 的可用后端的内部列表。使用该函数，您可以在不重启 shell 的情况下添加对新 VCS 的支持。

所有名为 VCS_INFO_* 的函数仅供内部使用。

26.5.6 变量说明

${vcs_info_msg_*N*_} (注意尾部的下划线)

> 其中 *N* 是一个整数，例如 vcs_info_msg_0_。这些变量用于存储最后一次 vcs_info 调用所生成的信息。这些变量与上述 formats、actionformats 和 nvcsformats 样式密切相关。这些样式都是列表。该列表的第一个成员会扩展为 ${vcs_info_msg_0_}，第二个成员会扩展为${vcs_info_msg_1_} ，第 N 个成员会扩展为 ${vcs_info_msg_N-1_}。(参见上面的 max-exports 样式) 。

所有名为 VCS_INFO_* 的变量仅供内部使用。

26.5.7 vcs_info 中的勾子

钩子是 vcs_info 中可以运行自己代码的地方。这些代码可以与调用它的代码通信，从而改变系统的行为。

对于配置，钩子可更改样式上下文：

```
:vcs_info:vcs-string+hook-name:user-context:repo-root-name
```

要向钩子注册函数，需要在适当的上下文中以 hooks 样式列出这些函数。

例如：

```
zstyle ':vcs_info:*+foo:*' hooks bar baz
```

This registers functions to the hook ‘foo’ for all backends. In order to avoid namespace problems, all registered function names are prepended by a ‘+vi-’, so the actual functions called for the ‘foo’ hook are ‘+vi-bar’ and ‘+vi-baz’.

如果要将某个函数注册到钩子中，而不考虑当前上下文，可以使用 vcs_info_hookadd 函数。要移除这样添加的函数，可以使用 vcs_info_hookdel 函数。

如果有什么奇怪的地方，可以在适当的上下文中启用 ‘debug’ 布尔样式，这样调用钩子的代码就会打印出试图执行的内容以及相关函数是否存在。

如果向钩子注册了多个函数，所有函数将相继执行，直到其中一个函数返回非零或所有函数都被调用。对上下文敏感的钩子函数会在静态注册的函数（由 vcs_info_hookadd 添加的函数）之前 执行。

您可以通过关联数组 user_data 在函数之间传递数据。 例如

```
+vi-git-myfirsthook(){
    user_data[myval]=$myval
}
+vi-git-mysecondhook(){
    # do something with ${user_data[myval]}
}
```

有一些变量在钩子上下文中比较特殊：

`ret`

钩子系统将返回给调用者的返回值。默认值为整数 'zero'。如果 `ret` 值发生变化，调用者的执行将如何变化，取决于具体的钩子。详情请参见下面的钩子文档。

`hook_com`

关联数组，用于从调用方到钩子函数的双向通信。使用的键取决于具体的钩子。

`context`

钩子的活动上下文。希望更改此变量的函数应首先将其设置为本地作用域。

`vcs`

检测到 VCS 后的当前 VCS。使用与 enable/disable 样式相同的值。可用于除 `start-up` 以外的所有钩子。

最后，是当前可用钩子的完整列表：

`start-up`

在 `vcs_info` 启动后，但在确定该目录中的 VCS 之前调用。必要时，它可用于暂时停用 `vcs_info`。当 `ret` 设置为 1 时，`vcs_info` 会终止并不执行任何操作；当 `ret` 设置为 2 时，`vcs_info` 会将一切设置为未激活版本控制，并退出。

`pre-get-data`

与 `start-up` 相同，但在检测到 VCS 之后。

`gen-hg-bookmark-string`

生成书签字符串时在 Mercurial 后台调用；`get-revision` 和 `get-bookmarks` 样式必须为 true。

此钩子获取 `vcs_info` 从 'hg'中收集的 Mercurial 书签的名称。

如果书签处于活动状态，则键 `${hook_com[hg-active-bookmark]}` 将被设置为其名称。 否则，键值会被取消设置。

当 ret 设置为非零时，${hook_com[hg-bookmark-string]} 中的字符串将被用于 formats 和 actionformats 中的 %m 转义，并将作为 ${backend_misc[bookmarks]} 出现在全局 backend_misc 数组中。

gen-applied-string

在 git（使用 stgit，或在 rebase 或合并过程中）和 hg（使用 mq）后端以及 quilt 支持中生成 applied-string 时调用；对于 quilt，use-quilt zstyle 必须为 true（mq 和 stgit 后端默认为激活状态）。

该钩子的参数以相反的顺序描述应用的补丁，即第一个参数是最上面的补丁，以此类推。

当可以提取补丁的日志信息时，这些消息嵌入在空格后的每个参数中，因此每个参数的形式为 '*patch-name first line of the log message*', 其中 *patch-name* 不含空格。日志信息的第一行'，其中 *patch-name* 不含空格。mq 后端传递的参数形式为"补丁名称"，可能包含空格，但不会提取补丁的日志信息。

当 ret 设置为非零时， ${hook_com[applied-string]} 中的字符串将以 %p 的形式出现在 patch-format 和 nopatch-format 样式中。 该钩子与 set-patch-format 一起负责 %-转义，以便在提示符中使用该值（参见 奇特现象）。

quilt 后端将输入传递给此钩子，其中包括 ${hook_com[quilt-patches-dir]} 输入，如果已确定的话，还包括${hook_com[quilt-pc-dir]} 输入。

gen-unapplied-string

当 unapplied-string 生成时，在 git（使用 stgit）和 hg（使用 mq）后端以及 quilt 支持中调用；get-unapplied样式必须为 true。

此钩子按顺序获取 vcs_info 中所有未应用补丁的名称，这意味着第一个参数是下一个要应用的补丁，以此类推。

每个参数的格式与上述 gen-applied-string 相同。

当 ret 设置为非零时， ${hook_com[unapplied-string]} 中的字符串将以 %u 的形式出现在 patch-format 和 nopatch-format 样式中。 该钩子与 set-patch-format 一起负责 %-转义，以便在提示符中使用该值（参见 奇特现象）。

quilt 后端将输入传递给此钩子，其中包括 ${hook_com[quilt-patches-dir]} 输入，如果已确定的话，还包括${hook_com[quilt-pc-dir]} 输入。

gen-mqguards-string

在 hg 后台生成 guards-string 时调用；get-mq 样式必须为 true（默认）。

此钩子会获取任何活动的 mq 守卫（guards)的名称。

当 ret 设置为非零时， ${hook_com[guards-string]} 中的字符串将在 patch-format 和 nopatch-format 样式的 %g 转义中使用。

no-vcs

当检测不到版本控制系统时，会调用此钩子。

‘hook_com’ 参数未被使用。

post-backend

后台完成信息收集后立即调用。

‘hook_com’ 键与 set-message 钩子相同。

post-quilt

在 quilt 支持完成后调用。钩子将传递以下信息作为参数：1. quilt 支持模式（‘addon’ 或 ‘standalone’）； 2. 包含补丁系列的目录；3. 保存 quilt 状态信息的目录（ ‘.pc’ 目录），如果未找到该目录，则传递字符串 "-nopc-"。

‘hook_com’ 参数未被使用。

set-branch-format

在设置 ‘branchformat’ 之前调用。钩子的唯一参数是此时配置的格式。

‘hook_com’ 键是 ‘branch’ 和 ‘revision’。 它们将被设置为 vcs_info 目前得出的值，任何更改都将在实际替换时直接使用。

如果 ret 设置为非零， ${hook_com[branch-replace]} 中的字符串将被原封不动地用作 vcs_info 设置的变量中 ‘%b’ 的替换。

set-hgrev-format

在设置 ‘hgrevformat’ 之前调用。钩子的唯一参数是此时配置的格式。

‘hook_com’ 键考虑的是 ‘hash’ 和 ‘localrev’。 它们被设置为 vcs_info 目前计算出的值，任何更改都将在实际替换时直接使用。

如果 ret 设置为非零， ${hook_com[rev-replace]} 中的字符串将被原封不动地用作 vcs_info 设置的变量中 ‘%i’ 的替换。

pre-addon-quilt

当 vcs_info 的 quilt 功能在 "插件"模式（ quilt 用于真正的版本控制系统之上）下激活时，就会使用此钩子。该钩子会在执行任何与 quilt 相关的操作之前被激活。

将此钩子中的 'ret' 变量设置为非零值，可避免运行任何 quilt 特定操作。

`set-patch-format`

此钩子用于控制像quilt、mqueue等补丁队列系统中 `patch-format` 和 `nopatch-format` 样式中的某些可能扩展。

该钩子用于 `git`、`hg` 和 `quilt` 后端。

钩子允许控制所有使用钩子的后端中的 %p (`${hook_com[applied]}`) 和 %u (`${hook_com[unapplied]}`) 扩展。对于 mercurial 后端，除此以外还可控制 %g (`${hook_com[guards]}`) 扩展。

如果 `ret` 设置为非零，则将不加修改地使用 `${hook_com[patch-replace]}` 中的字符串，而不是 `patch-format` 或 `nopatch-format` 中的扩展格式。

该钩子与 `gen-applied-string` 或 `gen-unapplied-string` 钩子（如果定义了这两个钩子）共同负责 %-转义，以便最终的 `patch-format` 值用于提示符中（请参阅 奇特现象）。

`quilt` 后端将输入传递给此钩子，其中包括 `${hook_com[quilt-patches-dir]}` 输入，如果已确定的话，还包括`${hook_com[quilt-pc-dir]}` 输入。

`set-message`

每次在设置 '`vcs_info_msg_`*N*`_`' 信息前调用。 它需要两个参数：第一个参数是信息变量名中的 '*N*'，第二个参数是当前配置的 `formats` 或 `actionformats`。

这里使用了许多 'hook_com' 键：'action', 'branch', 'base', 'base-name', 'subdir', 'staged', 'unstaged', 'revision', 'misc', 'vcs' 以及每个特定后端数据字段的一个 'miscN' 条目（N从 0 开始）。它们将被设置为 `vcs_info` 目前计算出的值，任何更改都将在实际替换时直接使用。

由于该钩子会被多次触发（每个配置的 `formats` 或 `actionformats` 都会触发一次），因此上文提到的每个 'hook_com' 键（`miscN` 条目除外）都有一个对应的 '_orig'，所以即使你按照自己的喜好修改了某个值，在下一次运行时仍然可以得到原来的值。更改 '_orig' 值可能不是一个好主意。

如果`ret`设置为非零值，则`vcs_info`将使用 `${hook_com[message]}` 中的字符串作为消息，不会进行更改。

如果这一切听起来令人困惑，请查看 示例 以及 Zsh 源代码中的 `Misc/vcs_info-examples` 文件。 它们包含一些解释性代码。

26.5.8 示例

完全不要使用 `vcs_info`（即使它在你的提示符中）：

```
zstyle ':vcs_info:*' enable NONE
```

禁用 bzr 和 svk 的后端：

```
zstyle ':vcs_info:*' disable bzr svk
```

禁用 **除** bzr 和 svk 外的所有功能：

```
zstyle ':vcs_info:*' enable bzr svk
```

为 git 提供了一种特殊格式：

```
zstyle ':vcs_info:git:*' formats       ' GIT, BABY! [%b]'
zstyle ':vcs_info:git:*' actionformats ' GIT ACTION! [%b|%a]'
```

所有格式（formats、actionformats、branchformat等）的 %x 扩展都是通过 'zsh/zutil' 模块中的 'zformat' 内置程序完成的。这意味着你可以用这些 %x 项做任何 zformat 支持的事情。特别是，如果你希望很长的内容有固定的宽度，比如 多变的 branchformat 中的哈希值，你可以这样做： %12.12i。这样就能将 40 个字符的哈希值缩减为 12 个前导字符。形式实际上是 '%*min*.*max*x'。还可以有更多。 详情请参见 zsh/zutil 模块。

使用更快的 bzr 后端

```
zstyle ':vcs_info:bzr:*' use-simple true
```

如果使用 use-simple，请报告它是否做了 'the-right-thing[tm]'。

以黄色显示 bzr 和 svn 的版本号：

```
zstyle ':vcs_info:(svn|bzr):*' \
       branchformat '%b%%F{yellow}:%r'
```

奇特现象 中解释了双百分号。

或者，也可以直接使用原始颜色代码：

```
zstyle ':vcs_info:(svn|bzr):*' \
       branchformat '%b%{'${fg[yellow]}'%}:%r'
```

通常情况下，当变量被插值到格式字符串中时，需要对变量进行 % 转义。在本例中，我们跳过了这一步骤，因为我们假定 ${fg[yellow]} 的值不包含任何 % 符号。

如果要在提示符中使用 vcs_info 提供的字符串，请确保将颜色代码括在 %{...%} 中。

下面介绍如何以命令形式（而不是在提示符中）打印 VCS 信息：

```
vcsi() { vcs_info interactive; vcs_info_lastmsg }
```

这样，您甚至可以通过 ':vcs_info:*:interactive:*'命名空间中的 vcs_info_lastmsg 来定义不同的输出格式。

现在，正如承诺的那样，一些使用钩子的代码：比如，你想在 vcs_info 的 %s formats 中用 'subversion' 替换字符串 'svn' 。

首先，我们要告诉 vcs_info 在用收集到的信息补全消息变量时调用一个函数：

```
zstyle ':vcs_info:*+set-message:*' hooks svn2subversion
```

什么也不会发生。这是合理的，因为我们还没有定义实际函数。要查看钩子子系统正在尝试做什么，请启用 'debug' 样式：

```
zstyle ':vcs_info:*+*:*' debug true
```

这应该能让你知道发生了什么。具体来说，我们要找的函数是 '+vi-svn2subversion'。注意，前缀是 '+vi-' 。所以，一切都井然有序，就像文档中描述的那样。检查完调试输出后，再次禁用它：

```
zstyle ':vcs_info:*+*:*' debug false
```

现在，我们来定义函数：

```
function +vi-svn2subversion() {
    [[ ${hook_com[vcs_orig]} == svn ]] && hook_com[vcs]=subversion
}
```

足够简单。如果我们在不那么通用的上下文中注册我们的函数，它甚至可以更简单。如果我们只在 'svn' 后端上下文中注册，就不需要测试哪个后端处于活动状态：

```
zstyle ':vcs_info:svn+set-message:*' hooks svn2subversion

function +vi-svn2subversion() {
    hook_com[vcs]=subversion
}
```

最后是一个更复杂的示例，使用钩子为 hg 后台创建自定义书签字符串。

同样，我们先注册一个函数：

```
zstyle ':vcs_info:hg+gen-hg-bookmark-string:*' hooks hgbookmarks
```

然后我们定义 '+vi-hgbookmarks' 函数：

```
function +vi-hgbookmarks() {
    # The default is to connect all bookmark names by
```

```
        # commas. This mixes things up a little.
        # Imagine, there's one type of bookmarks that is
        # special to you. Say, because it's *your* work.
        # Those bookmarks look always like this: "sh/*"
        # (because your initials are sh, for example).
        # This makes the bookmarks string use only those
        # bookmarks. If there's more than one, it
        # concatenates them using commas.
        # The bookmarks returned by `hg' are available in
        # the function's positional parameters.
        local s="${(Mj:,:)@:#sh/*}"
        # Now, the communication with the code that calls
        # the hook functions is done via the hook_com[]
        # hash. The key at which the `gen-hg-bookmark-string'
        # hook looks is `hg-bookmark-string'. So:
        hook_com[hg-bookmark-string]=$s
        # And to signal that we want to use the string we
        # just generated, set the special variable `ret' to
        # something other than the default zero:
        ret=1
        return 0
    }
```

Zsh 源代码目录下 Misc/vcs_info-examples 中的示例文件提供了一些较长的示例和可能有用的代码片段。

至此，我们的 zsh 的 vcs_info 之旅就结束了。

26.6 提示符主题

26.6.1 安装

您应确保源代码发布版 Functions/Prompts 目录中的所有函数都可用；除了特殊函数 'promptinit'，它们都以字符串 'prompt_' 开头。 你还需要 Functions/Misc 中的 'colors' 和 'add-zsh-hook' 函数。 您的系统可能已经安装了所有这些函数；如果没有，则需要找到并复制它们。 目录应作为 fpath 数组的元素之一出现（如果已经安装，则应如此），至少 promptinit 函数应自动加载，其余函数将自动加载。 最后，要初始化系统的使用，需要调用 promptinit 函数。 假设函数存储在 ~/myfns 目录中，则 .zshrc 中的以下代码将为此做出安排：

```
fpath=(~/myfns $fpath)
autoload -U promptinit
promptinit
```

26.6.2 主题选择

使用 prompt 命令选择您喜欢的主题。 可以在调用 promptinit 之后将此命令添加到 .zshrc 中，以便在已选择主题的情况下启动 zsh。

prompt [-c | -l]
prompt [-p | -h] [*theme* ...]
prompt [-s] *theme* [*arg* ...]

设置或检查提示符主题。 在没有任何选项和 *theme* 参数的情况下，以该名称命名的主题将被设置为当前主题。 可用主题在运行时确定；使用 -l 选项可查看列表。 特殊的 *theme* ‘random’ 会从可用主题中随机选择一个，并将提示符设置为该主题。

在某些情况下，*theme* 可能会被一个或多个参数修改，这些参数应在主题名称后给出。 有关这些参数的说明，请参阅每个主题的帮助。

选项是:

-c

显示当前选择的主题及其参数（如果有）。

-l

列出所有可用的提示符主题。

-p

预览由 *theme* 命名的主题，如果没有给出 *theme* 则预览所有主题。

-h

显示由 *theme* 命名的主题的帮助，如果未给出 *theme* 则显示 prompt 函数的帮助。

-s

将 *theme* 设置为当前主题并保存状态。

prompt_*theme*_setup

每个可用的 *theme* 都有一个设置函数，由 prompt 函数调用来安装该主题。 该函数可根据维护提示符的需要定义其他函数，包括用于预览提示符或提供使用帮助的函数。 通常情况下，不应直接调用主题的设置函数。

26.6.3 实用主题

```
prompt off
```

主题 'off' 会将所有提示符变量设置为没有特殊效果的最小值。

prompt default

主题 'default' 会将所有提示符变量设置为与启动无初始化文件的交互式 zsh 相同的状态。

prompt restore

特殊主题 'restore' 会清除所有主题设置，并将提示符变量设置为首次运行 'prompt' 函数前的状态，前提是每个主题都已正确定义了其清理方式（见下文）。

请注意，可以通过 'prompt restore'来撤销 'prompt off' 和 'prompt default'，但第二次还原不会撤销第一次。

26.6.4 编写主题

添加自己主题的第一步是为其取名，并在 fpath 目录下创建文件 'prompt_name_setup'，如上例中的 ~/myfns。 该文件至少应包含主题希望修改的提示符变量的赋值。 按照惯例，主题使用 PS1、PS2、RPS1 等，而不是更长的 PROMPT 和 RPROMPT。

该文件将作为当前 shell 上下文中的一个函数自动加载，因此其中可能包含自定义主题所需的任何命令，包括定义其他函数。 为了简化一些复杂的任务，您的设置函数还可以执行以下任何操作：

为 prompt_opts 赋值

数组 prompt_opts 的值可以是 "bang"、"cr"、"percent"、"sp" 和/或 "subst"。 相应的 setopts（promptbang 等）被打开，所有其他与提示符相关的选项被关闭。 如果您的函数需要，prompt_opts 数组甚至会在 localoptions 的范围之外保留 setopts。

修改勾子

建议使用 add-zsh-hook 和 add-zle-hook-widget（参见上文 操作勾子函数 部分）。 所有遵循命名模式 prompt_*theme_hook* 的钩子都会在提示符主题更改或禁用时自动移除。

声明清理

如果您的函数进行了任何其他更改，而这些更改在主题禁用时应该被撤销，那么您的设置函数可能会调用

```
prompt_cleanup command
```

其中 *command* 应适当加引号。 如果您的主题被禁用或被另一个替换，*command* 将与 eval 一起执行。 您可以声明多个这样的清理钩子。

定义预览

定义或自动加载函数 prompt_*name*_preview，以显示模拟版本的提示符。 promptinit 为未定义预览器的主题定义了一个简单的默认预览器。 该预览函数由 'prompt -p' 调用。

提供帮助

定义或自动加载函数 prompt_*name*_help ，以显示你主题的文档或帮助文本。 该帮助函数由 'prompt -h' 调用。

26.7 ZLE 函数

26.7.1 小部件

这些函数都实现了用户定义的 ZLE 小部件（请参阅 Zsh 行编辑器），可与交互式 shell 中的按键绑定。 要使用它们，您的 .zshrc 应包含以下格式的行

```
autoload function
zle -N function
```

后接适当的 bindkey 命令，将函数与按键序列关联起来。 建议的绑定说明如下。

bash 风格字函数

如果您正在寻找以 bash 的方式移动和编辑单词的函数（在 bash 中，只有字母数字字符才被视为单词字符），您可以使用下一节中描述的函数。 以下函数就足够了：

```
autoload -U select-word-style
select-word-style bash
```

forward-word-match, backward-word-match
kill-word-match, backward-kill-word-match
transpose-words-match, capitalize-word-match
up-case-word-match, down-case-word-match
delete-whole-word-match, select-word-match
select-word-style, match-word-context, match-words-by-style

前八个 '-match' 函数可以直接替换不带后缀的内置小部件。 默认情况下，它们的行为方式类似。 不过，通过使用样式和函数 select-word-style，可以改变单词的匹配方式。select-word-match的目的是在 vi 模式下作为文本对象使用，但可自定义单词样式。相比之下，文本对象 中描述的小部件使用固定的单词定义，与 vim 编辑器兼容。

配置这些函数的最简单方法是使用 select-word-style，它既可以作为一个普通函数与适当的参数一起调用，也可以作为一个用户定义的小部件调用，该部件会提

示要使用的单词样式的第一个字符。 首次调用时，前八个 -match 函数将自动替换内置版本，因此无需明确加载。

可用的单词样式如下。 只检查第一个字符。

bash

单词字符仅限字母数字字符。

normal

与正常的 shell 操作一样：单词字符是字母数字字符加上参数 $WORDCHARS 给出的字符串中的任何字符。

shell

单词是完整的 shell 命令参数，可能包括完整的加引号字符串或 shell 特有的任何标记。

whitespace

单词是以空格分隔的任意一组字符。

default

恢复默认设置；通常与 'normal'相同。

除 'default' 外，其他所有字符都可以作为大写字符输入，其效果与开启子字匹配时相同。 在这种情况下，带有大写字符的单词会被特殊处理：每一个单独的大写字符，或一个大写字符后跟任意数量的其他字符，都会被视为一个单词。 样式 subword-range 可以为默认的 '[:upper:]' 提供一个替代字符范围；该样式的值将被视为'[...]' 模式的内容（注意，不应提供外层括号，只应提供围绕命名范围的括号）。

如 zsh/zutil 模块 所述，使用 zstyle 命令可以获得更多控制权。 每种样式都在 :zle:*widget* 上下文中查找，其中 *widget* 是用户定义的 小部件的名称，而不是实现该小部件的函数的名称，因此在 select-word-style 提供定义的情况下，相应的上下文是 :zle:forward-word，以此类推。 函数 select-word-style 本身总是为上下文 ':zle:*' 定义样式，这些样式可以被更具体（更长）的模式和显式上下文覆盖。

样式 word-style 指定要使用的规则。 它可以有以下值。

normal

使用标准 shell 规则，即字母数字和 $WORDCHARS，除非被 word-chars 或 word-class 样式覆盖。

specified

与 normal类似，但 **只** 将指定的字符（而不是字母数字）视为单词字符。

unspecified

指定的否定。 给出的字符将 **不** 被视为单词的一部分。

shell

单词是通过使用生成 shell 命令参数的语法规则获得的。 此外，诸如 '()' 等从来都不是命令参数的特殊标记也会被视为单词。

whitespace

单词是以空白字符分隔的字符串。

前三条规则通常使用 $WORDCHARS，但参数中的值可以被样式 word-chars 改写，其工作方式与 $WORDCHARS 完全相同。 此外，word-class 样式使用字符类语法对字符进行分组，如果同时设置了 word-chars 和 word-class，则 word-class优先于 word-chars。 word-class 样式不包括字符类周围的括号；例如，'-:[:alnum:]' 是一个有效的 word-class，用以包括所有字母数字以及字符 '-' 和 ':'。 包含 ']', '^' 和 '-' 时要注意，因为它们是字符类内部的特殊字符。

如上所述，word-style 也可以附加 '-subword'，以开启子词匹配。

skip-chars 样式主要用于 transpose-words 和类似函数。 如果设置了该样式，它将给出从光标位置开始的字符计数，这些字符将不被视为单词的一部分，并被视为空格，而不管它们实际上是什么。 例如，如果

```
zstyle ':zle:transpose-words' skip-chars 1
```

已设置，且 transpose-words-match 被调用时光标位于 foo*X*bar 的 *X* 上，其中 *X* 可以是任何字符，那么结果表达式为 bar*X*foo。

通过将 word-context 样式设置为成对条目的数组，可以实现更精细的控制。 每对条目由 *pattern* 和 *subcontext* 组成。 光标所在的 shell 参数依次与每个 *pattern* 匹配，直到有一个匹配为止；如果匹配，则用冒号和相应的 *subcontext* 扩展上下文。 请注意，测试是针对行中的原始单词进行的，不会去掉引号。 单词之间会进行特殊处理：检查当前上下文，如果其中包含字符串 between，则将单词设置为单个空格；否则，如果其中包含字符串 back，则考虑光标前的单词，否则考虑光标后的单词。下面给出了一些示例。

skip-whitespace-first 样式仅用于 forward-word 小部件。如果设置为 true，那么 forward-word 将跳过任何非单词字符，然后跟着任何非单词字符：这类似于其他面向单词的小部件的行为，也类似于其他编辑器使用的行为，但与标准的 zsh 行为不同。这与其他以单词为导向的小部件以及其他编辑器的行为类似，但与标准的 zsh 行为不同。 使用 select-word-style 时，如果单词样式为

bash，则小部件在上下文 :zle:* 中被设置为 true，否则为 false。 可以通过在更具体的上下文 :zle:forward-word* 中进行设置来覆盖该设置。

通过定义一个由相应泛型函数实现的新小部件，然后为特定小部件的上下文设置样式，就可以创建具有特定行为的小部件。 例如，下面使用 backward-kill-word-match（实现 backward-kill-word 行为的通用部件）定义了一个小部件 backward-kill-space-word ，并确保新小部件始终实现空格分隔行为。

```
zle -N backward-kill-space-word backward-kill-word-match
zstyle :zle:backward-kill-space-word word-style space
```

小部件 backward-kill-space-word 现在可以与按键绑定。

以下是一些更进一步的样式使用示例，实际上是从 select-word-style 中简化的接口中提取的。

```
zstyle ':zle:*' word-style standard
zstyle ':zle:*' word-chars ''
```

为所有小部件实现类似于 Bash 风格的单词处理，即只有字母数字字符被视为单词字符；等效于在给定上下文中将参数 WORDCHARS 设置为空。

```
style ':zle:*kill*' word-style space
```

对名称中含有 'kill' 一词的小部件使用以空格分隔的单词。 在这种情况下，不会使用 word-chars 或 word-class 样式。

下面是一些使用 word-context 样式扩展上下文的示例。

```
zstyle ':zle:*' word-context \
       "*/*" filename "[[:space:]]" whitespace
zstyle ':zle:transpose-words:whitespace' word-style shell
zstyle ':zle:transpose-words:filename' word-style normal
zstyle ':zle:transpose-words:filename' word-chars ''
```

这提供了两种使用 transpose-words 的不同方法，取决于光标是在单词之间的空白处还是在文件名上，这里是指包含 / 的任何单词。 在空白处，标准 shell 规则定义的完整参数将被转置。 在文件名中，只有字母数字会被换码。 在其他地方，将使用 :zle:transpose-words 的默认样式转置单词。

词匹配和 zstyle 设置的所有处理实际上都是通过函数 match-words-by-style 实现的。 该函数可用于创建新的用户自定义小部件。 调用函数应将本地参数 curcontext 设置为 :zle:*widget*，创建本地参数 matched_words 并调用 match-words-by-style (不带参数)。 返回时，matched_words 将被设置为一个包含以下元素的数组：(1) 行的起始位置；(2) 光标前的单词；(3) 该单词与光标之间的任何非单词字符；(4) 光标位置上的任何非单词字符以及下一个单词前的任何剩余非单词字符，包括 skip-chars 样式指定的所有字符；(5) 光标处或

光标后的单词；（6）该单词后的任何非单词字符；（7）行的剩余部分。 其中任何一个元素都可能是空字符串；调用函数应对此进行测试，以决定是否可以执行其功能。

如果 match-words-by-style 的调用者将变量 matched_words 定义为关联数组(local -A matched_words)，那么上面给出的七个值应当从这个数组中获取，元素名为 start、word-before-cursor、ws-before-cursor、ws-after-cursor、word-after-cursor、ws-after-word 和 end。 此外，如果光标位于单词或子单词的起始位置，或位于单词或子单词之前的空白处（可通过测试 ws-after-cursor 元素来区分这两种情况），则 is-word-start 元素的值为 1，否则为 0。 为了今后的兼容性，建议使用这种形式。

可以将带参数的选项传递给 match-words-by-style，以覆盖样式的使用。 选项包括：

-w

word-style

-s

skip-chars

-c

word-class

-C

word-chars

-r

subword-range

例如，match-words-by-style -w shell -c 0 可用于提取光标周围的命令参数。

word-context 样式由函数 match-word-context 实现。 通常不需要直接调用该函数。

bracketed-paste-magic

bracketed-paste 小部件（请参阅 标准小部件 中的 杂项）会将粘贴的文本字面值直接插入编辑器缓冲区，而不是将其解释为按键。 这就禁用了一些常见的用法，即替换自插入部件以完成一些额外的处理。 下面介绍的 url-quote-magic 小部件就是一个例子。

bracketed-paste-magic 小部件旨在用一个封装器取代 bracketed-paste，重新启用这些 self-insert 操作以及 zstyles 选择的其他操作。 因此，安装该小部件时应

```
autoload -Uz bracketed-paste-magic
zle -N bracketed-paste bracketed-paste-magic
```

除了启用一些小部件处理外，bracketed-paste-magic 尝试尽可能忠实地复制 bracketed-paste。

可以设置以下 zstyles 来控制粘贴文本的处理。 所有样式均在上下文 ':bracketed-paste-magic' 中查找。

active-widgets

与粘贴时应激活的小部件名称相匹配的模式列表。 所有其他按键序列都将作为 self-inert-unmeta 处理。 默认值为 'self-*' ，因此任何以该前缀命名的用户自定义小部件都会与内置的 self-insert 一起激活。

如果未设置此样式（显式删除）或设置为空值，则不会激活任何小部件，粘贴的文本将按字面意思插入。 如果值中包含 'undefined-key'，粘贴文本中的未知序列将被丢弃。

inactive-keys

active-widgets 的反义词，是一个键序列列表，即使绑定到活动小部件上，也始终使用 self-insert-unmeta。 请注意，这是一个字面键序列列表，而不是模式。

paste-init

函数名称列表，在小部件上下文中调用（但不作为小部件）。 这些函数按顺序调用，直到其中一个函数返回非零状态。 参数 'PASTED' 包含粘贴文本的初始状态。 所有其他 ZLE 参数（如 'BUFFER' ）都有其正常值和副作用，并且可以使用完整的历史记录，因此，例如， paste-init 函数可以将 BUFFER 中的字词移动到 PASTED 中，使 active-widgets 可以看到这些字词。

如果 paste-init 函数的返回值为非 0，则 **不会** 阻止粘贴本身继续进行。

加载 bracketed-paste-magic 定义了 backward-extend-paste，这是一个在 paste-init 中使用的辅助函数。

```
zstyle :bracketed-paste-magic paste-init \
       backward-extend-paste
```

当粘贴会插入到一个单词的中间或将文本附加到已在行上的单词时，backward-extend-paste 会将 LBUFFER 中的前缀移到 PASTED 中，这样

active-widgets 就能看到完整的单词了。 这与 url-quote-magic 一起可能有用。

paste-finish

另一个按顺序调用的函数名列表，直到其中一个函数的返回值不为零。 在 active-widgets 处理完粘贴的文本 **后**，但在将其插入 'BUFFER' **之前**，***after*** 会调用这些函数。 ZLE 参数具有正常值和副作用。

如果 paste-finish 函数的返回值非零，则 **不会** 阻止粘贴本身继续进行。

加载 bracketed-paste-magic 时还定义了 quote-paste，这是一个用于 paste-finish 的辅助函数。

```
zstyle :bracketed-paste-magic paste-finish \
       quote-paste
zstyle :bracketed-paste-magic:finish quote-style \
       qqq
```

当粘贴的文本插入 BUFFER 时，会根据 quote-style 值进行加引号处理。 要强制关闭 bracketed-paste 的内置数字前缀引号，请使用:

```
zstyle :bracketed-paste-magic:finish quote-style \
       none
```

Important: 在 active-widgets 处理粘贴过程中（paste-init 之后、paste-finish 之前），BUFFER 开始为空，且历史记录受到限制，因此光标移动等操作可能不会超出粘贴内容的范围。 活动小部件分配给 BUFFER 的文本会在 paste-finish 之前复制回 PASTED 中。

copy-earlier-word

该小部件的工作原理类似于 insert-last-word 和 copy-prev-shell-word 的组合。 重复调用该小部件可检索相关历史行中较早的字词。 通过数字参数 N，插入历史行中第 N 个字；N 可以是负数，从行末开始计算。

如果 insert-last-word 已被用于检索上一行历史记录中的最后一个单词，则重复调用时将用同一行中较早的单词替换该单词。

否则，小部件适用于当前正在编辑的行上的单词。 widget 样式可以设置为另一个小部件的名称，该小部件应被调用来获取单词。 该小部件必须接受与 insert-last-word 相同的三个参数。

cycle-completion-positions

在命令行中插入一个无歧义的字符串后，新的基于函数的补全系统可能会知道这个字符串中有多个地方缺少字符或至少与一个可能的匹配字符不同。 然后，系统会将

光标放在它认为最有趣的位置上，即可以用尽可能少的键入在尽可能多的匹配字符之间进行消歧的位置。

通过这个小部件，可以轻松地将光标移动到其他感兴趣的位置。 可以反复调用它，在补全系统报告的所有位置之间循环。

delete-whole-word-match

这是另一个函数，其工作原理与上文描述的 -match 函数类似，即使用样式来决定单词边界。 不过，它不能替代任何现有函数。

基本操作是删除光标周围的单词。 不处理数字参数；只考虑光标周围的单字。 如果小部件中包含字符串 kill，则删除的文本将被放入剪切缓冲区，以便将来提取。 这可以通过定义 kill-whole-word-match 来实现：

```
zle -N kill-whole-word-match delete-whole-word-match
```

然后绑定小部件 kill-whole-word-match。

up-line-or-beginning-search, down-line-or-beginning-search

这些小部件类似于内置函数 up-line-or-search 和 down-line-or-search：如果在多行缓冲区内，它们会在缓冲区内向上或向下移动，否则它们会搜索与当前行起始位置相匹配的历史行。 不过，在这种情况下，它们会按照 history-beginning-search-backward 和 -forward 的方式，搜索与当前行直至当前光标位置相匹配的行，而不是该行的第一个单词。

edit-command-line

使用可视化编辑器编辑命令行，如在 ksh。

```
bindkey -M vicmd v edit-command-line
```

也可以在小部件的上下文中使用 editor 样式指定要使用的编辑器。编辑器以命令和参数数组的形式指定：

```
zstyle :zle:edit-command-line editor gvim -f
```

expand-absolute-path

将光标下的文件名扩展为绝对路径，解析符号链接。 在可能的情况下，将开头路径段转换为命名目录或用户主目录的引用。

history-search-end

该函数实现了小部件 history-beginning-search-backward-end 和 history-beginning-search-forward-end。 这些命令的工作方式是，首先调用相应的内置小部件（参见 历史控制），然后将光标移动到行尾。 在第二次调用

内置小部件之前，光标的原始位置会被记住并恢复，这样就可以重复同样的搜索，在历史记录中查找更远的位置。

虽然只 autoload 一个函数，但使用它的命令略有不同，因为它实现了两个小部件。

```
zle -N history-beginning-search-backward-end \
       history-search-end
zle -N history-beginning-search-forward-end \
       history-search-end
bindkey '\e^P' history-beginning-search-backward-end
bindkey '\e^N' history-beginning-search-forward-end
```

history-beginning-search-menu

该函数实现了另一种形式的历史搜索。 光标前的文本用于从历史记录中选择行，与 history-beginning-search-backward 相同，但所有匹配行都显示在一个编号菜单中。 输入相应的数字可插入完整的历史行。 请注意，必须输入前导零（只有在需要消除歧义时才会显示）。 整个历史记录都会被搜索；没有向前和向后之分。

使用数字参数时，搜索不固定在行的起始位置；用户键入的字符串可以出现在历史记录行中的任何位置。

如果小部件名称中包含‘-end’，光标就会移动到插入行的末尾。 如果小部件名称中包含‘-space’，则输入文本中的任何空格都将被视为通配符，可以匹配任何内容（因此前导空格等同于给出一个数字参数）。 两种形式可以结合使用，例如:

```
zle -N history-beginning-search-menu-space-end \
       history-beginning-search-menu
```

history-pattern-search

函数 history-pattern-search 实现了一个小部件，可提示以前向或后向搜索历史记录的模式。 模式采用常用的 zsh 格式，但第一个字符可以是 ^，以便将搜索锚定到行的开始，最后一个字符可以是 $，以便将搜索锚定到行的结束。 如果搜索没有锚定到行的末尾，光标就会定位在找到的模式之后。

创建可绑定小部件的命令与上面例子中的命令类似：

```
autoload -U history-pattern-search
zle -N history-pattern-search-backward history-pattern-search
zle -N history-pattern-search-forward history-pattern-search
```

incarg

在光标位于一个整数上或其左侧时输入该小部件的按键，会使该整数递增 1。 如果输入的是数字参数，则数字会按参数的大小递增（如果数字参数为负数，则数字会递减）。 可以设置 shell 参数 incarg，将默认增量改为 1 以外的值。

```
bindkey '^X+' incarg
```

incremental-complete-word

这样就可以递增补全一个单词。 启动此命令后，每输入一个字符后都会显示一个补全选项列表，您可以用 ^H 或 DEL 删除这些选项。 按回车键可接受到目前为止的补全，并返回正常编辑状态（即命令行 **不会** 立即执行）。 您可以按 TAB 进行正常补全，按 ^G 终止返回开始时的状态，按 ^D 列出匹配结果。

这只适用于新的基于函数的补全系统。

```
bindkey '^Xi' incremental-complete-word
```

insert-composed-char

使用该函数可以将键盘上没有的字符编辑插入命令行。 命令后面有两个与 ASCII 字符相对应的键（没有提示符）。 对于重音字符，这两个键是一个基本字符，后面是一个重音代码，而对于其他特殊字符，这两个字符共同构成要插入字符的助记符。 双字符代码是 RFC 1345 提供的代码的子集（例如 http://www.faqs.org/rfcs/rfc1345.html 中的例子）。

函数后面可以选择最多两个字符，用来替换从键盘读取的一个或两个字符；如果两个字符都有，则不读取输入。 例如，insert-composed-char a: 可以在小部件中使用，在命令行中插入带元音的 a。 与使用字面字符相比，这种方法的优点是更可移植。

为达到最佳效果，zsh 应编译为支持多字节字符（使用 --enable-multibyte 进行配置）；不过，该功能适用于 ISO-8859-1 等单字节字符集中的有限字符范围。

该字符被转换为本地表示法，并插入光标位置的命令行中。 (转换在 shell 中进行，使用 C 库提供的任何功能）。 对于数字参数，字符及其代码将在状态行中预览

这个函数可以在zle之外运行，在这种情况下，它会将字符（连同换行符）打印到标准输出。输入仍然来自按键输入。

请参阅 insert-unicode-char，了解使用十六进制字符编号插入 Unicode 字符的另一种方法。

在 Unicode 字符 U+0180 之前，重音字符集相当完整，而特殊字符集则不那么完整。 不过，从那个位置起就非常零散了。 不过，添加新字符很容易，请参见函数 define-composed-chars。 如有任何添加，请发送至 zsh-workers@zsh.org。

第二个字符与第一个字符重读时的代码如下。 请注意，并非每个字符都可以使用重音。

!

重音符号（如 "à"）。

'

重音符号（如 "á"）。

>

圆周率。

?

波浪符（如"ñ"）。 (这不是 ~，因为 RFC 1345 并不假定键盘上存在该字符）。

-

Macron（如"ā"）。(在基本字符上加一横杠）。

(

Breve（如"ă"）。(一个浅盘形状的底座上的字符）。

.

基本字符上方的点，或者 i 中没有点，或者 L 和 l 中居中的点。

:

Umlaut，也称为 Diaeresis（两点）是一种放置在元音字母上方的双点符号，用于表示发音的变化或分隔两个相邻的元音。（如"ä")

c

Cedilla（如"ç"）。（放置在字符底部的小尾巴符号）

_

下划线，但目前没有下划线字符。

/

笔画贯穿基本字符。

"

Double acute 是一种放置在字符上方的双重尖音符号，用于表示某些语言中特定字母的发音变化。(如ő）

;

Ogonek 是一种放置在字符底部右侧的小型前向挂钩符号，看起来像一个向前弯曲的小钩子。（如"ǫ"）

<

Caron，是一种放置在字符上方的小型折线符号，看起来像一个小 V 字母。(如"č")

0

在基本字符上画圆圈。

2

在基本字符上挂钩。

9

在基本字符上加 "号角"。

阿拉伯文、西里尔文、希腊文和希伯来文字母中最常见的字符均可使用；有关适当的序列，请查阅 RFC 1345。 此外，还有一组 RFC 1345 中没有的双字母代码，可用于与 ASCII 字符 ！至 ~ （0x21 至 0x7e）相对应的双倍宽度字符，方法是在字符前加上 ^，例如 ^A 表示双倍宽度的 A。

还可以理解以下其他双字符序列。

ASCII 字符

这些大多数键盘上都有：

<(

左方括号

//

反斜线（实线）

)>

右方括号

(!

左括号（大括号）

!!

竖条（管道符号）

`!)`

右括号（大括号）

`'?`

波浪号（‘~’）

特殊字母

拉丁字母的各种变体中出现的字符：

`ss`

Eszett，又称为 "scharfes S"（尖的 S），是德语中的一种特殊字符

`D-`, `d-`

Eth（Ð）是国际音标和一些非英语文字中使用的一个字符，表示发音为浊齿擦音的辅音。

`TH`, `th`

Thorn（Þ，þ）是一种字母，最初出现在古英语和北欧语言中

`kk`

Kra

`'n`

’n

`NG`, `ng`

Ng

`OI`, `oi`

Oi

`yr`

yr

`ED`

ezh

货币符号

Ct

美分

Pd

英镑（也包括里拉和其他货币）

Cu

货币

Ye

日元

Eu

欧元（注：不在 RFC 1345 中）

标点符号

指右引号，表示引号的形状（如 9 而不是 6），而不是语法用法。(例如，在德语中，右 低双引号用于开始引用）。

!I

倒置的感叹号

BB

断开的竖条

SE

部分

Co

版权

-a

西班牙文阴性序号指示符

<<

左栅栏

--

软连字符

Rg

注册商标

PI

Pilcrow（¶）是一个标点符号，通常表示段落的开头或分隔不同段落。

-o

西班牙文阳性序号指示符

>>

右栅栏

?I

倒置问号

-1

连字符

-N

En dash是一个连字符，长度通常等于所用字体的字母 "n" 的宽度

-M

Em dash是一个长破折号，长度通常等于所用字体的字母 "M" 的宽度

-3

一种水平线条或横线

:3

垂直省略号

.3

水平中线省略号

!2

双竖线

=2

双下划线

'6

左单引号

'9

右单引号

.9

右低引

9'

反右引号

"6

左双引号

"9

右双引号

:9

"Right" low double quote

9"

反右双引号

/-

一种印刷符号(†)

/=

一种印刷符号(‡)

数学符号

DG

度

-2, +-, -+

- sign, +/- sign, -/+ sign

2S

上标2

Superscript 2

上标3

1S

上标1

My

微

.M

中点，"·"号

14

刻

12

半

34

3刻

*X

乘号

-:

除号

%0

千分号

FA, TE, /0

对所有, 存在的, 空集

dP, DE, NB

偏导数, delta（增量）, del（纳布拉）

(-, -)

元素, 包含

*P, +Z

产品, 总合

*-, Ob, Sb

星号, 圆环, 子弹

RT, 0(, 00

根号, 比例, 无穷大

其它符号

cS, cH, cD, cC

花色：黑桃, 红心, 方块, 梅花

Md, M8, M2, Mb, Mx, MX

音乐符号：四分音符、八分音符、十六分音符、旗号、自然符号、升号符号

Fm, Ml

女性, 男性

重音

'>

插入符号（与caret相同，^）

'!

重音符号（与反引号相同，`）

',

小尾巴符号Cedilla

':

分音符Diaeresis (Umlaut)

'm

长音符（Macron）

''

尖音符Acute

insert-files

该函数允许您键入文件模式，并查看每一步的扩展结果。 按回车键后，所有扩展结果都会插入命令行。

```
bindkey '^Xf' insert-files
```

insert-unicode-char

首次执行时，用户输入一组十六进制数字。 最后再调用 insert-unicode-char 以终止。 然后，这些数字会被转换成相应的 Unicode 字符。 例如，如果小部件绑定到 ^XU，则字符序列 '^XU 4 c ^XU' 会插入 L（Unicode U+004c）。

有关使用双字符助记符插入字符的方法，请参阅 insert-composed-char。

narrow-to-region [-p *pre*] [-P *post*]
[-S *statepm* | -R *statepm* | [-l *lbufvar*] [-r *rbufvar*]]
[-n] [*start end*]
narrow-to-region-invisible

将缓冲区的可编辑部分缩小到光标和标记之间的区域，该区域可以是任一顺序。 该区域不能为空。

narrow-to-region 可以作为一个小部件使用，也可以作为用户定义的小部件中的函数调用；默认情况下，可编辑区域外的文字仍然可见。 执行 recursive-edit 后，将恢复原来的加宽状态。 以函数形式调用时，有多种选项和参数可供选择。

选项 -p *pretext* 和 -P *posttext* 可用于在函数执行期间替换显示前后的文本；任一选项或两个选项都可以是空字符串。

如果同时给出选项 -n，*pretext* 或 *posttext* 将仅在区域之前或之后有文本时插入，而这些文本将被隐藏(区域内文本）。

可以给出两个数字参数，它们将代替光标和标记位置。

选项 -S *statepm* 用于根据其他选项缩小范围，同时将原始状态保存在名称为 *statepm* 的参数中，而选项 -R *statepm* 则用于从参数中恢复状态。而 -R 选项则用于从参数中恢复状态；请注意，这两种情况下都需要使用参数的 ***name***。 在第二种情况下，其他选项和参数无关紧要。 使用此方法时，不执行 recursive-edit；调用的小部件应使用选项 -S 调用此函数，在命令行上执行自己的编辑或通过 'zle recursive-edit' 将控制权传递给用户，然后使用选项 -R 调用此函数。 参数 *statepm* 必须是普通参数的合适名称，但以 _ntr_ 开头的参数被保留给 narrow-to-region 使用。 通常情况下，该参数是调用函数的本地参数。

选项 -l *lbufvar* 和 -r *rbufvar* 可用于指定参数，小部件将在其中存储操作产生的文本。 参数 *lbufvar* 将包含 LBUFFER，而 *rbufvar* 将包含 RBUFFER。 这两个选项都不能与 -S 或 -R 一起使用。

narrow-to-region-invisible 是一个简单的小部件，它调用带有参数的 narrow-to-region，用‘...’替换区域外的任何文本。 它不需要任何参数。

任何 zle 命令通常都会导致该行被接受或终止，此时显示会恢复（小部件会返回）。 因此，要接受或终止当前行，还需要额外的此类命令。

如果该行被接受，则两个小部件的返回状态都为 0，否则为非 0。

下面是一个使用该功能的小部件的微不足道的例子。

```
local state
narrow-to-region -p $'Editing restricted region\n' \
  -P '' -S state
zle recursive-edit
narrow-to-region -R state
```

predict-on

这个函数集使用历史搜索实现预测键入。 在 predict-on 之后，键入字符会导致编辑器在历史记录中向后(backward)查找与迄今键入内容相同的第一行。 predict-off后，编辑器会恢复正常，查找到哪一行就编辑哪一行。 事实上，很多时候甚至不需要使用 predict-off，因为如果该行与历史记录中的内容不匹配，添加键就会执行标准补全，如果没有找到补全，就会插入其本身。 不过，在行中间编辑可能会混淆预测；请参阅下面的 toggle 样式。

有了基于函数的补全系统（这是需要的），你应该可以在几乎任何地方输入 TAB，将光标推进到下一个 “有趣的” 字符位置（通常是当前单词的末尾，但有时也会在单词中间的某个地方）。 当然，只要整行都是您想要的，您就可以用回车接受，而无需先将光标移到末尾。

首次使用 predict-on 时，会创建几个额外的小部件函数：

delete-backward-and-predict

替换 backward-delete-char 小部件。 您无需自行绑定。

insert-and-predict

通过替换 self-insert 小部件实现预测键入。 您无需自行绑定。

predict-off

关闭预测键入功能。

虽然只 autoload 了 predict-on 函数，但也有必要为 predict-off 创建一个按键绑定。

```
zle -N predict-on
zle -N predict-off
bindkey '^X^Z' predict-on
bindkey '^Z' predict-off
```

read-from-minibuffer

当作为一个小部件内部的函数调用时最有用，但作为一个小部件本身也能正常工作。 它在当前命令行下方提示输入一个值；可以使用所有标准的 zle 操作（而不仅仅是在执行时所选的有限集合, 例如,execute-named-cmd ）输入一个值。 然后，数值会通过参数 $REPLY 返回给调用函数，编辑缓冲区也会恢复到之前的状态。 如果键盘中断（通常为 ^G）导致读取中止，函数将返回状态 1，且不设置 $REPLY。

如果为函数提供了一个参数，则将其作为提示符，否则使用‘? ’。 如果提供两个参数，它们分别是提示符和 $LBUFFER 的初始值，如果提供第三个参数，则是 $RBUFFER 的初始值。 这提供了默认值和光标的起始位置。 返回时，整个缓冲区就是 $REPLY 的值。

有一个选项可用：‘-k *num*’ 指定读取 *num* 个字符而不是整行。 在这种情况下，行编辑器不会递归调用，因此根据终端设置，输入可能不可见，而且只有输入键被放入 $REPLY，而不是整个缓冲区。 需要注意的是，与 read 内置函数不同的是，必须给出 *num*；没有默认值。

这个名字有点名不副实，因为事实上并没有使用 shell 自身的 minibuffer。 因此，在读取数值时，仍然可以调用 executed-named-cmd 和类似函数。

replace-argument, replace-argument-edit

函数 replace-argument可用于替换当前命令行中的命令行参数，如果当前命令行为空，则替换最后执行的命令行中的参数（新命令行不执行）。 参数按标准 shell 语法分隔、

如果给出数字参数，则表示要替换的参数。 0 表示命令名称，如在历史扩展中。 负数参数从最后一个字开始倒数。

如果没有给出数字参数，则替换当前参数；如果使用的是上一行历史记录，则替换最后一个参数。

函数提示要求替换参数。

如果小部件包含 edit 字符串，例如定义为

```
zle -N replace-argument-edit replace-argument
```

则函数会显示参数的当前值供编辑，否则替换的编辑缓冲区初始为空。

replace-string, replace-pattern
replace-string-again, replace-pattern-again

函数 replace-string 实现了三个小部件。 如果与函数同名定义，它将提示两个字符串；第一个（源）字符串将在行编辑缓冲区中出现的任何地方被第二个字符串替换。

如果小部件名称包含 'pattern'，例如使用命令 'zle -N replace-pattern replace-string' 定义小部件，那么匹配将使用 zsh 模式进行。 源字符串中可以使用所有 zsh 扩展的 globbing 模式；需要注意的是，与文件名生成不同，模式不需要匹配整个单词，glob 限定符也没有任何作用。 此外，替换字符串可以包含参数或命令替换。 另外，替换字符串中的 '&' 将被替换为匹配的源字符串，反引号数字 '*N*' 将被替换为匹配的第 *N* 个带括号的表达式。 可以使用 '\{*N*}' 形式来保护数字不受后面数字的影响。

如果小部件中包含 'regex'（或 'regexp'），则使用正则表达式进行匹配，并遵守选项 RE_MATCH_PCRE 的设置（参见下文对函数 regexp-replace 的描述）。 上述用于模式匹配的特殊替换功能也可以使用。

默认情况下，前一个源字符串或替换字符串不会提供给编辑。 不过，可以通过将上下文 :zle:*widget*（例如 :zle:replace-string）中的 edit-previous 样式设置为 true，来激活这一功能。 此外，正数参数会强制提供前一个值，负数或零参数则不会。

函数 replace-string-again 可用于重复之前的替换；不做提示。 与 replace-string 一样，如果小部件的名称包含 'pattern' 或 'regex'，则执行模式匹配或正则表达式匹配，否则执行字面字符串替换。 请注意，无论使用模式匹配、正则表达式匹配还是字符串匹配，前面的源文本和替换文本都是一样的。

此外，只要在当前会话期间有替换，replace-string 就会在提示符上方显示上一次的替换；如果源字符串为空，该替换将被重复，小部件不会替换字符串提示。

例如，从这行开始

```
print This line contains fan and fond
```

用源字符串 'f(?)n' 调用 replace-pattern，替换字符串为 'c\1r'，会产生一行不太有用的内容：

```
print This line contains car and cord
```

可以使用 narrow-to-region-invisible 小部件来限制替换字符串的范围。 当前版本的一个限制是，undo 在撤销替换之前，会循环浏览替换和源字符串的更改。

send-invisible

它与 read-from-minibuffer 类似，可以作为一个小部件中的函数或小部件本身来调用，并以交互方式读取键盘上的输入。 不过，输入的内容会被隐藏，取而代之的

是显示一串星号（'*'）。 该值将保存在参数 $INVISIBLE 中，并在恢复的光标位置将其引用插入编辑缓冲区。 如果由于键盘中断（通常为 ^G）或其他退出编辑的操作（如 push-line）而中止了读取，$INVISIBLE 将被设置为空，并恢复原始缓冲区，保持不变。

如果为函数提供了一个参数，则将其作为提示符，否则使用'Non-echoed text: '（与 emacs 中相同）。 如果提供了第二个和第三个参数，它们将用于开始和结束插入缓冲区的 $INVISIBLE 引用。 默认是以 ${ 开始，然后是 INVISIBLE，最后用 } 关闭，但也可以实现其他多种效果。

smart-insert-last-word

这个函数可以替代 insert-last-word 小部件，就像这样：

```
zle -N insert-last-word smart-insert-last-word
```

如果使用数字参数，或在调用其他小部件时传递命令行参数，其行为与 insert-last-word 类似，但如果设置了 INTERACTIVE_COMMENTS，注释中的单词将被忽略。

否则，将查找并插入上一条命令中最右边的 "interesting" 字词。 "interesting" 的默认定义是，单词至少包含一个字母、斜线或反斜线字符。 使用 match 样式可以覆盖这一定义。 用于查找样式的上下文是小部件名称，因此上下文通常是 :insert-last-word。 不过，您可以将此函数绑定到不同的小部件，以使用不同的模式：

```
zle -N insert-last-assignment smart-insert-last-word
zstyle :insert-last-assignment match '[[:alpha:]][[:alnum:]]#=*
bindkey '\e=' insert-last-assignment
```

如果未找到感兴趣的单词，且 auto-previous 样式设置为 true 值，则搜索将继续在历史中向上进行。 如果 auto-previous 未设置或设置为 false（默认值），则必须反复调用小部件才能搜索到更早的历史行。

transpose-lines

仅对多行编辑缓冲区有用；此处的行是当前屏幕缓冲区内的行，而不是历史行。 其效果类似于 Emacs 中的同名函数。

将当前行与上一行平移，并将光标移到下一行的开始位置。 重复上述操作（可以通过提供一个正数参数来实现）的效果是将光标上方的行向下移动若干行。

如果参数为负数，则需要在光标上方的两行。 这两行将被移位，光标将移至上一行的起始位置。 使用小于-1 的数字参数的效果是将光标上方的行向上移动减去该数的行数。

url-quote-magic

这个小部件取代了内置的 self-insert，使输入 URL 作为命令行参数变得更容易。 输入时，系统会分析输入字符，如果需要加引号，则会检查当前单词是否包含 URI 方案。 如果发现有URI方案，且当前单词尚未加引号，则会在输入字符前插入反斜杠。

控制引号行为的样式：

url-metas

> 该样式在上下文':url-quote-magic:*scheme*' 中查找（其中 *scheme* 是当前 URL 的模式，例如"ftp"）。 值是一个字符串，其中列出了在使用该模式的 URL 中出现时应作为 globbing 元字符处理的字符。 默认引用(quote)所有 zsh 扩展 globbing 字符，不包括 '<' 和 '>' ，但包括大括号（如大括号扩展）。 另请参阅 url-seps。

url-seps

> 与 url-metas 类似，但列出了应视为命令分隔符、重定向、历史引用等的字符。 默认引用标准的 shell 分隔符集，不包括与扩展 globbing 字符重叠的字符，但包括 '<' 和 '>' 以及 $histchars 的第一个字符。

url-globbers

> 该样式在上下文 ':url-quote-magic' 中查找。 这些值构成了一个命令名称列表，这些命令名称应自行对 URL 字符串进行 globbing。 这意味着它们被别名为使用 'noglob' 修饰符。 当一行中的第一个单词与这些值之一匹配, **并且** URL 指向本地文件（参见 url-local-schema），只有 url-seps 字符会被加引号；而 url-metas 则不会被加引，允许它们影响命令行解析、补全等。 默认值是字面意义上的'noglob' 加上（当 zsh/parameter 模块可用时）别名到辅助函数 'urlglobber' 或其别名 'globurl' 的所有命令。

url-local-schema

> 尽管 url-quote-magic 和 urlglobber 都使用这种样式，但它总是在上下文 ':urlglobber' 中查找。 这些值构成了一个 URI 模式(schema)列表，这些模式应被视为是指向本地文件的真实本地路径名，而不是指相对于网络服务器定义的文档根目录的文件。 默认值为 "ftp" 和 "file"。

url-other-schema

> 类似于 url-local-schema，但列出了 urlglobber 和 url-quote-magic 应执行的所有其他 URI 模式(schema)。 如果命令行中的 URI 模式没有出现在该列表或 url-local-schema 中，则不会被神奇地加引号。 默认值为"http"、"https"和"ftp"。 当某一模式同时出现在此处和 url-local-schema 中时，将根据命令名称是否出现在 url-globbers 中，以不同方式加引号。

加载 url-quote-magic 还定义了一个辅助函数 'urlglobber' ，并将 'globurl' 别名为 'noglob urlglobber'。 该函数拆分本地 URL，尝试对 URL 路径中的本地文件部分进行模式匹配，然后将结果重新放回 URL 格式。

vi-pipe

此函数从键盘读取移动命令，然后提示外部命令。移动所覆盖的缓冲区部分被管道输送到外部命令，然后被命令的输出所取代。如果移动命令与 vi- 绑定，则使用当前行。

该函数作为从用户定义的小部件中读取 vi 移动命令的示例。

which-command

该函数可直接替换内置小部件 which-command。 它的功能有所增强，能正确检测命令字是否需要扩展为别名；如果需要，则会从扩展后的别名继续追踪命令字，直到找到要执行的命令。

whence 样式在上下文 :zle:$WIDGET 中可用；可将其设置为数组，以给出用于调查找到的命令字的命令和选项。 默认值为 whence -c。

zcalc-auto-insert

该函数与 数学函数 中描述的 zcalc 函数一起使用很有用。 它应该绑定到一个代表二元运算符的键上，例如 '+', '-', '*' 或 '/' 。 在 zcalc 中运行时，如果关键字出现在行的开头或紧跟在开括号之后，则在关键字本身的表示之前插入文本 "ans"。 这样可以方便地在当前行中使用前一个计算的答案。 输入符号前插入的文本可以通过设置变量 ZCALC_AUTO_INSERT_PREFIX 进行修改。

因此，举例来说，输入 '+12'，然后回车，就会在前面的结果上加上 12。

如果 zcalc 处于 RPN 模式（ -r 选项），该绑定的效果将被自动抑制，因为一行中单独的运算符是有意义的。

当不在 zcalc 中时，按键只是插入符号本身。

26.7.2 实用函数

这些函数在构建小部件时非常有用。 应使用 'autoload -U *function*' 加载这些函数，并根据用户自定义 widget 的指示调用它们。

split-shell-arguments

该函数将当前正在编辑的行分割为 shell 参数和空白。 结果存储在数组 reply 中。该数组按顺序包含该行的所有部分，从第一个参数之前的空白处开始，到最后一个参数之后的空白处结束。 因此（只要未设置 KSH_ARRAYS 选项），数组中的奇数

索引表示空白，偶数索引表示参数。 需要注意的是，数组中的引号不会被删除；将 reply 中的所有元素按顺序连接在一起，可以保证产生原始行。

参数 REPLY 设置为 reply 中包含光标后字符的字的索引，其中第一个元素的索引为 1。 参数 REPLY2 将被设置为该字词中光标下字符的索引，其中第一个字符的索引为 1。

因此，reply、REPLY 和 REPLY2 都应本地化到封闭函数中。

有关如何调用该函数的示例，请参阅下文介绍的函数 modify-current-argument。

modify-current-argument [*expr-using-*$ARG | *func*]

该函数提供了一种简单的方法，允许用户定义的小部件修改光标下的命令行参数（如果光标位于参数之间，则紧靠光标左侧）。

参数可以是一个表达式，该表达式在求值时，在 shell 参数 ARG 上进行操作，而 shell 参数 ARG 已被设置为光标下的命令行参数。 表达式应适当加引号，以防止过早求值。

或者，如果参数不包含 ARG 字符串，则假定它是一个 shell 函数，当前命令行参数是唯一的参数。 函数应将变量 REPLY 设置为命令行参数的新值。 如果函数返回非零状态，则调用函数也返回非零状态。

例如，包含以下代码的用户自定义小部件会将光标下参数中的字符转换为全大写：

```
modify-current-argument '${(U)ARG}'
```

下面的命令会删除当前单词中的任何引号（无论是反斜线还是引号样式之一），代之以整个单引号：

```
modify-current-argument '${(qq)${(Q)ARG}}'
```

下面的命令会对命令行参数执行目录扩展，并用绝对路径取而代之：

```
expand-dir() {
  REPLY=${~1}
  REPLY=${REPLY:a}
}
modify-current-argument expand-dir
```

实际上，函数 expand-dir 很可能不会在调用 modify-current-argument 的小部件中定义。

26.7.3 样式

上述几个小部件的行为可以通过使用 zstyle 机制来控制。 特别是，与补全系统交互的小部件会将它们的上下文传递给它们调用的任何补全。

break-keys

> 该样式由 incremental-complete-word 小部件使用。它的值应该是一个模式，所有与该模式匹配的键都将导致小部件停止增量补全，而该键不会有任何进一步的影响。与 incremental-complete-word 直接使用的所有样式一样，该样式通过上下文 ':incremental' 进行查找。

completer

> incremental-complete-word 和 insert-and-predict 小部件会在调用补全之前设置其顶级上下文名称。 这样就可以为普通补全和这些小部件定义不同的补全函数集。 例如，要在正常补全中使用补全、近似和修正，在增量补全中使用补全和修正，在预测中只使用补全，可以使用:
>
> ```
> zstyle ':completion:*' completer \
>        _complete _correct _approximate
> zstyle ':completion:incremental:*' completer \
>        _complete _correct
> zstyle ':completion:predict:*' completer \
>        _complete
> ```
>
> 限制预测中使用的补全器是个好主意，因为它们可能会在输入时自动调用。 _list 和 _menu 补全器绝对不能用于预测。 可以使用 _approximate、_correct、_expand 和 _match 补全器，但要注意它们可能会更改光标后单词中任何位置的字符，因此需要仔细观察结果是否与您的预期一致。

cursor

> insert-and-predict小部件在上下文 ':predict' 中使用这种样式，以决定在尝试补全后将光标置于何处。 值为:
>
> complete
>
> > 光标会留在补全完成时的位置，但前提是光标必须位于与用户刚刚插入的字符相同的字符之后。 如果光标位于其他字符之后，则该值与 'key' 相同。
>
> key
>
> > 光标会停留在刚刚插入的字符的第 n 次出现之后，其中 n 是尝试补全之前该字符在单词中出现的次数。 简而言之，这样做的效果是，即使补全代码发现在该位置不需要插入其他字符，光标也会留在刚刚输入的字符之后。
>
> 该样式的任何其他值都会无条件地将光标停留在补全代码离开的位置。

list

当使用 incremental-complete-word 小部件时，该样式表示是否应在每次按键时列出匹配项（如果它们适合显示在屏幕上）。 使用上下文前缀 ‘:completion:incremental’。

insert-and-predict小部件使用这种样式来决定是否显示补全，即使只有一种可能的补全。 如果该样式的值是字符串 always，则会显示补全。 在这种情况下，上下文为‘:predict’ （**不是** ‘:completion:predict’）。

match

该样式由 smart-insert-last-word 使用，用于提供与有趣的单词相匹配的模式（使用完整的 EXTENDED_GLOB 语法）。 上下文是 smart-insert-last-word 所绑定的小部件的名称（见上文）。 smart-insert-last-word 的默认行为相当于:

```
zstyle :insert-last-word match '*[[:alpha:]/\\]*'
```

但是，您可能希望包含含有空格的单词：

```
zstyle :insert-last-word match '*[[:alpha:][:space:]/\\]*'
```

或者包含数字，只要单词至少有两个字符长：

```
zstyle :insert-last-word match '*([[:digit:]]?|[[:alpha:]/\\])*'
```

上述示例会导致包含 "2>" 等重定向。

prompt

incremental-complete-word 小部件会在递增补全期间在状态行中显示该样式的值。 字符串值可以按照 PS1 和其他提示符参数的方式包含以下任意子串：

%c

替换为生成匹配的补全函数名称（不含前导下划线）。

%l

当 list 样式被设置时，如果匹配列表太长而无法在屏幕上显示，则替换为 ‘...’，否则替换为空字符串。 如果 list 样式为 ‘false’ 或未设置，则始终删除 ‘%l’。

%n

用生成的匹配数代替。

%s

如果没有与该行单词匹配的补全，如果匹配的单词没有与该行单词不同的共同前缀，或者如果有这样的共同前缀，则分别用 '-no match-', '-no prefix-' 或空字符串替换。

%u

如果该部分与该行上的词不同，替换为所有匹配词中的不明确部分（如果有的话）。

与 'break-keys' 一样，它使用 ':incremental' 上下文。

stop-keys

该样式由 incremental-complete-word 小部件使用。 其值的处理方式与 break-keys 样式类似（并使用相同的上下文：':incremental'）。 不过，在这种情况下，所有与作为其值给出的模式匹配的键都将停止增量补全，然后执行它们的常规功能。

toggle

该布尔样式由 predict-on 及其相关小部件在上下文 ':predict' 中使用。 如果将其设置为标准的 'true' 值之一，则预测键入功能会在不太可能有用的情况下自动关闭，例如编辑多行缓冲区时，或移动到行中间并删除一个字符后。 默认情况下，在明确调用 predict-off 之前，预测键入都是打开的。

verbose

该布尔样式由 predict-on 及其相关小部件在上下文 ':predict' 中使用。 如果设置为标准的 'true' 值之一，这些小部件就会在切换预测状态时在提示符下方显示一条信息。 这在与 toggle 样式结合使用时最为有用。 默认情况下不显示这些信息。

widget

这种样式类似于 command 样式： 对于使用 zle 调用其他小部件的小部件函数，有时可以使用这种样式来覆盖被调用的小部件。 此样式的上下文是调用小部件的名称（不是调用函数的名称，因为一个函数可能绑定多个小部件名称）。

```
zstyle :copy-earlier-word widget smart-insert-last-word
```

请查看调用的小部件或函数的文档，以确定是否使用 widget 样式。

26.8 异常处理

提供了两个函数，使 zsh 能够以其他语言相熟的形式提供异常处理。

throw *exception*

函数 throw 会抛出名为 *exception* 的异常。 该名称是一个任意字符串，仅用于 throw 和 catch 函数。 在大多数情况下，异常与 shell 错误的处理方式相同，即未处理的异常将导致 shell 中止函数或脚本中的所有处理，并返回交互式 shell 的顶层。

catch *exception-pattern*

函数 catch 的返回状态为 0，前提是有异常抛出，且模式 *exception-pattern* 与异常名称相匹配。 否则返回状态 1。 *exception-pattern* 是标准的 shell 模式，与 EXTENDED_GLOB 选项的当前设置一致。 此外，还定义了一个别名 catch，以防止函数参数匹配文件名，因此可以不加引号地使用模式。 需要注意的是，由于异常与其他 shell 错误没有本质区别，因此可以使用空字符串作为异常名称来捕获 shell 错误。 shell 变量 CAUGHT 将被 catch 设置为捕获的异常名称。 捕获异常后，可以再次调用 throw 函数来重新抛出异常。

这些函数旨在与 复杂命令 中描述的 always 结构一起使用。 这一点非常重要，因为只有这种结构才能为异常提供所需的支持。 下面是一个典型示例。

```
{
  # "try" block
  # ... nested code here calls "throw MyExcept"
} always {
  # "always" block
  if catch MyExcept; then
    print "Caught exception MyExcept"
  elif catch ''; then
    print "Caught a shell error.  Propagating..."
    throw ''
  fi
  # Other exceptions are not handled but may be caught further
  # up the call stack.
}
```

如果所有异常都应被捕获，下面的习语可能更合适。

```
{
  # ... nested code here throws an exception
} always {
  if catch *; then
    case $CAUGHT in
      (MyExcept)
      print "Caught my own exception"
      ;;
      (*)
      print "Caught some other exception"
      ;;
```

```
    esac
  fi
}
```

与其他语言的异常处理一样，异常也可以由深度嵌套在 'try' 代码块中的代码抛出。 但要注意的是，异常必须在当前 shell 中抛出，而不是在为管道、加括号的当前 shell 结构或某种形式的命令或进程替换而分叉的子 shell 中抛出。

系统内部使用 shell 变量 EXCEPTION 来记录抛出和捕获之间的异常名称。 这种方案的一个缺点是，如果异常未被处理，变量 EXCEPTION 将保持设置，如果随后出现 shell 错误，可能会被错误地识别为异常名称。 在任何使用异常处理的代码的最外层开头添加 `unset EXCEPTION` 就可以解决这个问题。

26.9 MIME 函数

有三个函数可用于处理扩展名识别的文件，例如，在作为命令执行时，将文件 `text.ps` 分派给适当的查看器。

`zsh-mime-setup` [`-fv`] [`-l` [*suffix* ...]]
`zsh-mime-handler` [`-l`] *command argument* ...

这两个函数使用 `~/.mime.types` 和 `/etc/mime.types` 文件，它们将类型和扩展名联系起来；还使用 `~/.mailcap` 和 `/etc/mailcap` 文件，它们将类型和处理这些类型的程序联系起来。 这些，通过多媒体互联网邮件扩展（Multimedia Internet Mail Extensions），在许多系统中都有提供。

要启用该系统，应自动加载并运行函数 `zsh-mime-setup`。 这样，带有扩展名的文件就会被视为可执行文件，并由函数补全系统补全。 用户无需调用 `zsh-mime-handler` 函数。

该系统通过使用 'alias -s' 设置后缀别名来运行。 用户已安装的后缀别名不会被覆盖。

对于以小写定义的后缀，大写变体也将自动得到处理（例如，如果定义了对后缀 pdf 的处理，则 PDF 将自动得到处理），但反之不成立。

除非给出 `-f` 选项，否则重复调用 `zsh-mime-setup` 不会覆盖后缀和可执行文件之间的现有映射。 但请注意，这不会覆盖分配给 `zsh-mime-handler` 以外的处理程序的现有后缀别名。

使用选项 `-l` 调用 `zsh-mime-setup` 会列出现有的映射，但不会更改它们。 要列出的后缀（可能包含模式字符，在命令行中应加注引号以避免直接解释）可以作为附加参数给出，否则将列出所有后缀。

使用选项 `-v` 调用 `zsh-mime-setup` 会在设置(setup)操作过程中显示冗长输出。

系统尊守 mailcap 标志 needsterminal 和 copiousoutput；请参阅 mailcap(4) 或 mailcap(5)（不同平台的手册页名称不同）。

函数使用以下样式，这些样式通过 zstyle 内置命令（zsh/zutil 模块）定义。 它们应在 zsh-mime-setup 运行前定义。 使用的上下文均以 :mime: 开头，在某些情况下还会使用附加组件。 建议在样式模式后加上 *（适当加引号），以防系统将来扩展。 下面给出了一些例子。

对于有多个后缀的文件，例如 .pdf.gz，如果上下文包含后缀，则将从可能的最长后缀开始查找，直到找到与样式匹配的后缀。 例如，如果 .pdf.gz 产生了与处理程序匹配的后缀，则使用该后缀；否则将使用 .gz 的处理程序。 需要注意的是，由于后缀别名的工作方式，总是要求为尽可能短的后缀提供处理程序， 因此在本例中，只有在 .gz 也被处理（但处理方式不一定相同）的情况下，.pdf.gz 才能被处理。 另外，如果不需要处理 .gz 本身，只需添加命令

```
alias -s gz=zsh-mime-handler
```

初始化代码已经足够了；.gz 不会单独处理，但可能会与其他后缀一起处理。

current-shell

如果该布尔值为 true，相关上下文的 mailcap 处理程序将使用 eval 内置程序运行，而不是启动一个新的 sh 进程。 这种方式效率更高，但在 mailcap 处理程序使用严格 POSIX 语法的情况下偶尔可能无法运行。

disown

如果该布尔样式为 true，那么在后台启动的mailcap处理程序将被从Shell的作业控制中移除，即不受父 shell 中作业的控制。 此类处理程序几乎总是产生自己的窗口，因此设置该样式唯一可能产生的有害副作用是，从Shell内部更难以终止这些作业。

execute-as-is

该样式提供了一个模式列表，用于匹配与处理程序一起传递执行的文件。 如果文件与模式匹配，则整个命令行将以当前形式执行，不使用处理程序。 这对可能有后缀但本身仍可执行的文件非常有用。 如果未设置样式，则使用 *(*) *(/) 模式；因此，可执行文件将直接执行，而不会传递给处理程序，并且可以使用选项 AUTO_CD 来更改碰巧带有 MIME 后缀的目录。

execute-never

该样式可与 execute-as-is 结合使用。 它被设置为与文件全路径相对应的模式数组，即使传递给 MIME 处理程序的文件符合 execute-as-is，这些文件也不应被视为可执行文件。 这对于不处理执行权限或包含来自其他操作系统的可执行文件的文件系统非常有用。 例如，如果 /mnt/windows 是 Windows 挂载点，那么

```
zstyle ':mime:*' execute-never '/mnt/windows/*'
```

将确保在该区域找到的任何文件都将作为 MIME 类型执行，即使这些文件是可执行的。 如本例所示，完整的文件名将与模式相匹配，无论文件是如何传递给处理程序的。 文件将使用 修饰符 中描述的 :P 修饰符解析为完整路径；这意味着尽可能解析符号链接，以便以正确的方式链接到其他文件系统。

file-path

当 find-file-in-path 样式在同样的上下文中为真时使用。 设置为用于搜索待处理文件的目录数组；默认为特殊参数 path 指定的命令路径。shell 选项 PATH_DIRS 会被遵从； 如果设置了 shell 选项 PATH_DIRS，即使要处理的文件名在命令行中包含 '/'，也会搜索相应的路径。 完整的上下文为 :mime:.*suffix*:，如 handler 样式所述。

find-file-in-path

如果设置，则允许在命令路径或 file-path 样式指定的路径中搜索名称不包含绝对路径的文件。 如果在路径中找不到文件，将在本地查找（无论当前目录是否在路径中）；如果在本地找不到文件，除非设置了 handle-nonexistent 样式，否则处理程序将终止。 路径中找到的文件将按照 execute-as-is 样式进行测试。 完整上下文为 :mime:.*suffix*:，如 handler 样式所述。

flags

定义处理程序的标志；上下文与 handler 样式相同，格式与 mailcap 中的标志相同。

handle-nonexistent

默认情况下，与文件不对应的参数不会传递给 MIME 处理程序，以防止它拦截路径中恰好有后缀的命令。 该样式可设置为一个扩展 glob 模式数组，用于将参数（即使不存在）传递给处理程序。 如果未明确设置，则默认为 [[:alpha:]]#:/*，允许将 URL 传递给 MIME 处理程序，即使这些 URL 在文件系统中并不存在该格式。 完整的上下文为 :mime:.*suffix*:，如 handler 样式所述。

handler

指定后缀的处理程序；后缀由上下文给出，如 :mime:.*suffix*:，处理程序的格式与 mailcap 中的格式完全相同。 请特别注意 '.' 和后面的冒号，以区分这种上下文用法。 这将覆盖 mailcap 文件指定的任何处理程序。 如果处理程序需要终端，则 flags 样式应设置为包含 needsterminal 字样；如果输出要通过分页程序显示（但如果处理程序本身就是分页程序，则不需要），则应包含 copiousoutput。

`mailcap`

在设置过程中读取的 `~/.mailcap` 和 `/etc/mailcap` 格式的文件列表，取代默认的由这两个文件组成的列表。 上下文为 `:mime:`。 列表中的 `+` 将被默认文件替换。

`mailcap-priorities`

该样式用于解析同一 MIME 类型的多个 mailcap 条目。 它由下列元素组成的数组构成，按优先级降序排列；如果前面的条目无法解析正在比较的条目，后面的条目将被使用。 如果所有测试都无法解析条目，则保留遇到的第一个条目。

`files`

读取文件（`mailcap`样式中的条目）的顺序。较早的文件优先。 (请注意，这并不能解决[resolve]同一文件中的条目）。

`priority`

mailcap 条目中的优先级标志。 优先级是一个从 0 到 9 的整数，默认值为 5。

`flags`

`mailcap-prio-flags` 选项给出的测试用于解析条目。

`place`

较晚的条目优先；由于条目是严格有序的，因此该测试总是成功的。

请注意，由于这种样式是在初始化过程中处理的，因此上下文总是 `:mime:`，没有后缀区分。

`mailcap-prio-flags`

当在 `mailcap-priorities` 样式指定的测试列表中遇到关键字 `flags` 时，就会使用这种样式。 应将其设置为一个模式列表，每个模式都要根据 mailcap 条目中指定的标志（换句话说，在 mailcap 文件的某些条目中发现的赋值集）进行测试。 列表中较早的模式优先于较晚的模式，匹配的模式优先于不匹配的模式。

`mime-types`

在设置过程中读取的 `~/.mime.types` 和 `/etc/mime.types` 格式的文件列表，取代默认的由这两个文件组成的列表。上下文为 `:mime:`。 列表中的 `+` 将被默认文件替换。

`never-background`

如果设置了这个布尔样式，给定上下文的处理程序将始终在前台运行，即使 mailcap 条目中提供的标志表明不需要这样做（例如，它不需要终端）。

pager

如果设置，将代替 $PAGER 或 more 来处理设置了 copiousoutput 标志的后缀。 上下文与 handler 相同，即 :mime:.*suffix*: 用于处理带有给定 *suffix* 的文件。

例如：

```
zstyle ':mime:*' mailcap ~/.mailcap /usr/local/etc/mailcap
zstyle ':mime:.txt:' handler less %s
zstyle ':mime:.txt:' flags needsterminal
```

随后运行 zsh-mime-setup 时，它会在给出的两个文件中查找 mailcap 条目。 后缀名为 .txt 的文件将通过运行 'less *file.txt*'来处理。 设置 needsterminal 标志是为了表明该程序必须连接终端运行。

由于调度命令有多个步骤，如果尝试执行扩展名为 .*ext* 的文件而没有达到预期效果，则应检查以下内容。

命令 'alias -s *ext*' 应显示 'ps=zsh-mime-handler'。 如果显示其他内容，则说明已安装了另一个后缀别名，且未被覆盖。 如果没有显示任何内容，则说明没有安装处理程序：这很可能是因为在 .mime.types 和 mailcap 组合中没有为 .ext 文件找到处理程序。 在这种情况下，应在 ~/.mime.types 和 mailcap 中添加适当的处理程序。

如果扩展名由 zsh-mime-handler 处理，但文件却无法正确打开，要么是为该类型定义的处理程序不正确，要么是与之相关的标志不合适。 运行 zsh-mime-setup -l 将显示处理程序和标志（如果有）。 处理程序中的 %s 将被文件替换（必要时适当加引号）。 检查列出的处理程序是否按所示方式运行。 如果处理程序需要在终端下运行，还应检查是否设置了 needsterminal 或 copiousoutput 标志；如果输出应发送到分页程序(pager)，则使用第二个标志。 以下是此类程序的适合 mailcap 条目的示例：

```
text/html; /usr/bin/lynx '%s'; needsterminal
```

运行 'zsh-mime-handler -l *command line*' 会打印出将要执行的命令行，经过简化以去除任何标志的影响，并加上引号，以便输出（内容）可以作为完整的 zsh 命令行运行。 补全系统使用此命令行来决定如何在 zsh-mime-setup 处理文件后补全。

pick-web-browser

该函数独立于上述两个 MIME 函数，可直接分配给后缀：

```
autoload -U pick-web-browser
alias -s html=pick-web-browser
```

它是作为智能前端提供的，用于调度网络浏览器。它可以作为函数或 shell 脚本运行。 如果无法启动浏览器，则返回状态 255。

有多种样式可供选择，以便自定义浏览器：

`browser-style`

样式的值是一个数组，按照浏览器类型的递减顺序给出首选项。 元素的值可以是

`running`

当 X 窗口显示可用时，使用已在运行的图形用户界面浏览器。 `x-browsers` 样式中列出的浏览器将依次尝试，直到找到一个为止；如果找到了，文件将在该浏览器中显示，因此用户可能需要检查它是否已经出现。 如果没有找到正在运行的浏览器，则不会启动该浏览器。 除 Firefox、Opera 和 Konqueror 外，其他浏览器都被假定为能理解 Mozilla 远程打开 URL 的语法。

`x`

当 X 窗口显示可用时，启动一个新的图形用户界面浏览器。 搜索 `x-browsers` 样式中列出的浏览器是否可用，然后启动找到的第一个浏览器。 不会检查是否有已运行的浏览器。

`tty`

启动终端浏览器 。搜索 `tty-browsers` 样式中列出的浏览器是否可用，然后启动找到的第一个浏览器。

如果未设置样式，则使用默认的 `running x tty`。

`x-browsers`

在 X 窗口系统下运行时，按优先级递减的浏览器数组。 数组包括启动浏览器的命令名称。 它们是在 `:mime:` 的上下文中查找的（将来可能会扩展，因此建议在 `:mime:` 中添加‘*’）。 例如,

```
zstyle ':mime:*' x-browsers opera konqueror firefox
```

指定 `pick-web-browser`应首先依次查找 Opera、Konqueror 或 Firefox 的运行实例，如果未找到，则尝试启动 Opera。 默认值为 `firefox mozilla netscape opera konqueror`。

`tty-browsers`

与 x-browsers 类似的数组，但它提供了在没有 X 窗口显示时使用的浏览器。 默认值为 elinks links lynx。

command

如果设置了该样式，就可以选择用于打开浏览器页面的命令。 上下文为 :mime:browser:new:$browser: ，用于启动新浏览器；或 :mime:browser:running:$browser: ，用于在当前 X 显示器上已运行的浏览器中打开 URL，其中 $browser 是 x-browsers 或 tty-browsers 样式中匹配的值。 样式值中的转义序列 %b 将被浏览器替换，而 %u 将被 URL 替换。 如果未设置样式，所有新实例的默认值相当于 %b %u，而使用运行中浏览器的默认值相当于 Konqueror 的 kfmclient openURL %u、Firefox 的 firefox -new-tab %u、Opera 的 opera -newpage %u，以及所有其他浏览器的 %b -remote "openUrl(%u)" 值。

26.10 数学函数

zcalc [-erf] [*expression* ...]

一个基于 zsh 算术运算工具的相当强大的计算器。 其语法与大多数编程语言中的公式类似；详情请参见 算术求值。

非编程人员应注意，与许多其他编程语言一样，只涉及整数的表达式（无论是不带 '.' 的常量、包含这样字符串常量的变量，还是声明为整数的变量）默认使用整数运算，这与普通台式计算器的操作方式不同。 要强制使用浮点运算，可通过选项 -f；请参阅下面的进一步说明。

如果 ~/.zcalcrc 文件存在，一旦函数建立并准备处理命令行，就会在函数内部引入该文件。 例如，这可以用来设置 shell 选项；emulate -L zsh 和 setopt extendedglob 此时生效。 任何在存在文件的情况下未能引入文件的失败都被视为致命的。与其他初始化文件一样，如果设置了目录 $ZDOTDIR ，则使用该目录而不是$HOME。

如果数学库 zsh/mathfunc 可用，它将被加载；请参阅 zsh/mathfunc 模块。 数学函数与原始系统库相对应，因此三角函数使用弧度求值，等等。

输入的每一行都作为一个表达式进行计算。 提示符会显示一个数字，该数字与存储计算结果的位置参数相对应。 例如，以 '4> ' 开头的一行的计算结果以 $4 的形式提供。 最后一个计算值以 ans 的形式提供。 可以使用完整的命令行编辑功能，包括以前的计算历史；历史记录保存在 ~/.zcalc_history 文件中。 要退出，请输入空行或单独输入 ':q'（为了历史兼容性，允许输入 'q'）。

以单个反斜杠结束的行的处理方式与命令行编辑相同：删除反斜杠，函数提示输入更多内容（提示符前有 '...' 表示），然后将各行合并为一行，得到最终结果。 此外，如果目前输入的开括号多于闭括号，zcalc 将提示要求输入更多内容。

如果在 zcalc 启动时给定了参数，那么这些参数将被用于初始化前几个位置参数。计算器启动时会给出直观提示。

提供常量 PI (3.14159...) 和 E (2.71828...)。 可以进行参数赋值，但需要注意的是，除非使用 :local 特殊命令，否则所有参数都将放入全局命名空间。 函数会创建名称以 _ 开头的局部变量，因此用户应避免这样做。 变量 ans（最后一个回应）和 stack（RPN 模式下的堆栈）可以直接引用；stack 是一个数组，但其中的元素是数字。其他各种特殊变量在本地使用时都有其标准含义，例如 compcontext、match、mbegin、mend、psvar。

输出基数可以通过传递选项 '*-#base*' 来初始化，例如 'zcalc -#16'（'#' 可能需要加引号，取决于设置的 globbing 选项）。

如果设置了选项 '-e'，函数将以非交互方式运行：参数被视为表达式，如同逐行交互输入一样进行计算。

如果设置了选项 '-f'，所有数字都将被视为浮点数，因此例如表达式 '3/4' 的值为 0.75，而不是 0。 选项必须以单独的词出现。

如果设置了选项 '-r'，则会进入 RPN（反波兰语符号）模式。 该模式具有多种附加属性：

栈

> 计算值保存在一个堆栈中；堆栈包含在一个名为 stack 的数组中，最新值保存在 ${stack[1]} 中。

操作符和函数

> 如果输入的行与运算符（+、-、*、/、**、^、| 或 &）或 zsh/mathfunc 库提供的函数相匹配，则会弹出堆栈的底部元素作为参数。 堆栈中较高的元素（最近的元素）会被用作较早的参数。 然后将结果推入 ${stack[1]}。

表达式

> 其他表达式将正常求值、打印并作为数值加入堆栈。 单行上的表达式语法是正常的 shell 运算（而不是 RPN）。

栈列表

> 如果选项 -r 后面的整数没有空格，那么在每次运算时，堆栈中的所有元素（如果有的话）都会被打印出来，而不是只打印最近的结果。因此，例如 zcalc -r4 在每次打印结果时将 $stack[4] 显示为 $stack[1]。

复制： =

> 伪操作符 = 会将栈中最近的元素复制到栈上。

pop

伪函数 pop 将弹出堆栈中最近的元素。 单独使用 '>' 也有同样的效果。

>*ident*

表达式 >（不带空格）后跟 shell 标识符，会导致弹出堆栈中最近的元素，并赋值给该名称的变量。 该变量是 `zcalc` 函数的局部变量。

<*ident*

表达式 < 后面（没有空格）跟一个 shell 标识符，会将该标识符名的变量的值推入堆栈。 *ident* 可以是一个整数，在这种情况下，带有该数字的前一个结果（如标准 `zcalc` 提示符中 > 之前所示）会被放入堆栈。

Exchange: xy

伪函数 xy 会交换堆栈中最近的两个元素。 '<>' 具有相同的效果。

提示符可通过参数 ZCALCPROMPT 进行配置，并进行标准的提示符扩展。 当前条目的索引存储在本地数组 `psvar` 的第一个元素中，可在 ZCALCPROMPT 中称为引用为 '%1v'。 默认提示符为 '%1v> '。

变量 ZCALC_ACTIVE 在函数内部设置，可以通过嵌套函数进行测试；如果 RPN 模式处于激活状态，则变量值为 `rpn`，否则为 1。

有一些特殊命令可用；这些命令以冒号开头。 为了向后兼容，某些命令可以省略冒号。 如果运行了 `compinit`，则可以进行。

输出精度可以在 zcalc 中通过许多计算器常用的特殊命令来指定。

`:norm`

默认输出格式。 它与 printf %g 规范相对应。 通常显示六位小数。

`:sci` *digits*

科学计数法，对应 printf %g 输出格式，精度由 *digits* 给定。 根据输出值的不同，会产生定点或指数符号。

`:fix` *digits*

定点符号，对应 printf %f 输出格式，精度由 *digits* 给出。

`:eng` *digits*

指数符号，对应 printf %E 输出格式，精度由 *digits* 给出。

`:raw`

原始输出：这是数学运算输出的默认形式。 它显示的精度可能高于实际数字。

其他特殊命令：

:!*line...*

将 *line...* 作为普通 shell 命令行执行。 请注意，它是在函数的上下文中执行的，即与局部变量一起执行。 :! 后面的空格是可选项。

:local *arg ...*

声明函数的局部变量。 也可以使用其他变量，但这些变量将从全局作用域中获取或放到全局作用域中。

:function *name* [*body*]

定义一个数学函数，或（在没有 *body* 的情况下）删除该函数。 :function 可以缩写为 :func，或简化为 :f。 *name* 可以包含与 shell 函数名相同的字符。 函数使用 zmathfuncdef 进行定义，见下文。

请注意，zcalc 会处理所有引号。 因此，举例来说

```
:f cube $1 * $1 * $1
```

定义的函数的唯一参数。 如此定义的函数，或者任何直接或间接使用 functions -M 定义的函数，在 RPN 模式下只需在行上键入名称即可执行；这将从堆栈中弹出适当数量的参数传递给函数，例如,在 cube 函数的示例中为 1。 如果有可选参数，则只提供必选参数。

[*#base*]

这不是一条特殊的命令，而是正常算术语法的一部分；不过，当该形式单独出现在一行中时，默认输出基数(radix)将设置为 *base*。 例如，使用 '[#16]' 显示十六进制输出，并在前面标明基数，或使用'[##16]' 只显示以给定基数表示的原始数字。 基数本身总是以十进制指定。'[#]' 则恢复正常的输出格式。 注意，设置输出基数会抑制浮点输出；使用 '[#]' 可恢复正常操作。

$*var*

按字面意思打印 var 的值，不影响计算。 要使用 var 的值，请省略前导句 '$'。

请参阅函数中的注释，了解一些额外的提示。

```
 min(arg, ...)
max(arg, ...)
sum(arg, ...)
```

zmathfunc

函数 zmathfunc 定义了三个数学函数 min、max 和 sum。 函数 min 和 max 接受一个或多个参数。 函数 sum 接受 0 个或多个参数。 参数可以是不同类型（整数和浮点数）。

不要与 zsh/mathfunc 模块混淆，该模块在 zsh/mathfunc 模块 中有所描述。

zmathfuncdef [*mathfunc* [*body*]]

functions -M 的便捷前端。

使用两个参数，定义一个名为 *mathfunc* 的数学函数，该函数可用于任何形式的算术运算。 *body* 是实现函数的数学表达式。 它可以引用位置参数 $1、$2, ... 来表示必选参数， ${1:-*defvalue*} ... 来表示可选参数。请注意，必须严格遵守这些形式，函数才能计算出正确的参数数目。 函数的实现被保存在一个名为 zsh_math_func_*mathfunc* 的 shell 函数中；通常用户不需要直接引用 shell 函数。 任何现有的同名函数都会被静默替换。

使用一个参数，删除数学函数 *mathfunc* 以及 shell 函数的实现。

在没有参数的情况下，以适合还原定义的形式列出所有 *mathfunc* 函数。 这些函数不一定由 zmathfuncdef 定义。

26.11 用户配置函数

zsh/newuser 模块自带一个函数，用于帮助新用户配置 shell 选项。 如果安装了该模块，也可以手动运行该函数。 即使模块的默认行为（即为没有启动文件的新用户登录运行该函数）被禁止，也可以使用该函数。

zsh-newuser-install [-f]

该函数为用户提供了多种自定义初始化脚本的选项。 目前只处理 ~/.zshrc。 如果设置了参数 ZDOTDIR，则会使用 $ZDOTDIR/.zshrc；这为用户提供了一种无需更改现有 .zshrc 即可配置文件的方法。

默认情况下，如果在相应目录中找到 .zshenv、.zprofile、.zshrc 或 .zlogin，则函数会立即退出。 需要使用选项 -f 才能强制函数继续执行。 请注意，即使 .zshrc 本身不存在，也可能出现这种情况。

按照目前的配置，如果用户拥有 root 权限，函数将立即退出；该行为不可重写。

一旦激活，函数的行为应该是不言自明的。 菜单出现以允许用户更改选项和参数的值。 欢迎提出改进建议。

脚本退出时，用户有机会选择是否保存新文件；在此之前，更改并不是不可逆的。不过，脚本会注意把更改限制在‘# Lines configured by zsh-newuser-

install’和‘# End of lines configured by zsh-newuser-install’这两行标记的组内。 此外，旧版本的 .zshrc 会保存到一个后缀名为 .zni 的文件中。

如果函数编辑了现有的 .zshrc，用户必须确保所做的更改会生效。 例如，如果控制权通常会提前从现有的 .zshrc 中返回，那么这些行将不会被执行；或者稍后的初始化文件可能会覆盖选项或参数，等等。 函数本身并不试图检测任何此类冲突。

26.12 其它函数

在 zsh 发行版的 Functions/Misc 目录中有大量有用的函数。 大多数函数都非常简单，不需要在此提供文档，但有几个函数值得特别一提。

26.12.1 说明

colors

该函数初始化多个关联数组，用于将颜色名称映射到（或从）ANSI 标准八色终端代码。 提示符主题系统（提示符主题）将使用这些颜色。 您很少需要多次运行 colors。

八种基础色是black, red, green, yellow, blue, magenta, cyan和 white。 每个属性都有前景和背景代码。 此外，还有七个强度属性： bold, faint, standout, underline, blink, reverse, 和 conceal。 最后，有七个代码用于否定属性： none （将所有属性重置为默认值）、normal（既不加粗也不模糊）、no-standout、no-underline、no-blink、no-reverse和 no-conceal。

某些终端不支持所有颜色和强度组合。

关联数组是:

color
colour

将所有颜色名称映射到其整数代码，并将整数代码映射到颜色名称。 八个基本名称映射到前景色代码，名称的前缀为 ‘fg-’ ，如 ‘fg-red’。 名称前缀为 ‘bg-’ 的，如 ‘bg-blue’ ，则表示背景颜色代码。 从代码到颜色的反向映射产生前景代码的基本名称和背景代码的 bg- 形式。

虽然将它们称为 ‘颜色’ 有些名不副实，但这些数组也将其他 14 个属性从名称映射到代码，并将代码映射到名称。

fg
fg_bold
fg_no_bold

将八种基本颜色名称映射到 ANSI 终端转义序列，以设置相应的前景文本属性。 fg 序列在不改变八个强度属性的情况下改变颜色。

bg
bg_bold
bg_no_bold

将八种基本颜色名称映射到设置相应背景属性的 ANSI 终端转义序列。 bg 序列可在不改变八个强度属性的情况下改变颜色。

此外，标量参数 reset_color 和 bold_color 被设置为 ANSI 终端转义符，分别用于关闭所有属性和打开粗体强度。

fned [-x *num*] *name*

与 zed -f 相同。 此函数未出现在 zsh 发行版中，但可以通过将 zed 链接到 fpath 中某个目录下的 fned 来创建。

histed [[*name*] *size*]

与 zed -h 相同。 此函数未出现在 zsh 发行版中，但可以通过将 zed 链接到 fpath 中某个目录下的 histed 名称来创建。

is-at-least *needed* [*present*]

对两个字符串进行 '大于等于' 比较，这两个字符串的格式为 zsh 版本号，即由数字和文本组成的字符串，以点或破折号分隔。 如果未提供 *present* 字符串，则使用 $ZSH_VERSION。 两个字符串中的段落从左到右配对，忽略前导的非数字部分。 如果一个字符串的段数少于另一个字符串，则缺失的段视为零。

这在启动文件中非常有用，可用于设置选项和其他状态（这些选项和状态在所有版本的 zsh 中都不可用）。

```
is-at-least 3.1.6-15 && setopt NO_GLOBAL_RCS
is-at-least 3.1.0 && setopt HIST_REDUCE_BLANKS
is-at-least 2.6-17 || print "You can't use is-at-least here."
```

nslookup [*arg* ...]

nslookup 命令的封装函数需要 zsh/zpty 模块（参见 zsh/zpty 模块）。 除了提供可定制的提示符（包括右侧提示符）和 nslookup 命令、主机名等的补全（如果使用基于函数的补全系统）外，它的行为与标准 nslookup 完全相同。 可以使用上下文前缀 ':completion:nslookup' 设置补全样式。

另请参阅下面的 pager、prompt 和 rprompt 样式。

regexp-replace *var regexp replace*

使用正则表达式对变量执行全局搜索和替换操作。 使用 POSIX 扩展正则表达式（ERE），除非设置了 RE_MATCH_PCRE 选项，这时将使用 Perl 兼容的正则表达式（这要求 shell 与 pcre 库链接）。

var 是包含要匹配字符串的变量名。 函数将直接修改该变量。 应避免使用变量 MATCH、MBEGIN、MEND、match、mbegin、mend，因为这些变量会被正则表达式使用。

regexp 是要与字符串匹配的正则表达式。

replace 是替换文本。 它可以包含将被替换的参数、命令和算术表达式：特别是，对 $MATCH 的引用将被模式匹配的文本替换。

如果至少进行了一次匹配，则返回状态为 0，否则为 1。

请注意，如果使用 POSIX EREs，^ 或字边界运算符（如果有）可能无法正常工作。

run-help *cmd*

该函数可由 run-help ZLE 小部件调用，以取代默认的别名。有关设置说明，请参阅 '获取在线帮助'（实用程序）。

在接下来的讨论中，如果 *cmd* 是一个文件系统路径，则首先将其简化为最右边的部分（文件名）。

首先会在 HELPDIR 参数指定的目录下查找名为 *cmd* 的文件，以获得帮助。 如果没有找到文件，则会查找名为 run-help-cmd 的辅助函数、别名或命令。 如果找到，将以当前命令行的其余部分（命令名 *cmd* 后面的所有内容）作为参数执行该辅助函数。 如果既未找到文件也未找到辅助函数，则执行外部命令 'man *cmd*'。

"ssh" 命令的助手示例：

```
run-help-ssh() {
    emulate -LR zsh
    local -a args
    # Delete the "-l username" option
    zparseopts -D -E -a args l:
    # Delete other options, leaving: host command
    args=(${@:#-*})
    if [[ ${#args} -lt 2 ]]; then
        man ssh
    else
        run-help $args[2]
    fi
}
```

Functions/Misc 目录中提供了多个此类助手。 这些辅助工具必须自动加载，或作为可执行脚本放在搜索路径中，以便 run-help 查找和使用。

```
run-help-btrfs
run-help-git
run-help-ip
run-help-openssl
run-help-p4
run-help-sudo
run-help-svk
run-help-svn
```

btrfs, git, ip, openssl, p4, sudo, svk, 和 svn 命令的辅助函数.

tetris

Zsh 曾被指责不如 Emacs 完整，因为它缺少俄罗斯方块游戏。 编写这个函数就是为了驳斥这种恶毒的诽谤。

该功能必须作为 ZLE 小部件使用：

```
autoload -U tetris
zle -N tetris
bindkey keys tetris
```

要开始游戏，请输入 *keys* 执行小部件。 你正在编辑的命令行会暂时消失，你的键映射也会被俄罗斯方块控制键暂时取代。 退出游戏（按下 'q'）或输掉游戏后，编辑器将恢复之前的状态。

如果您在游戏中途退出，下一次调用 tetris 小部件时，游戏将继续进行。 如果输了，则会重新开始游戏。

tetriscurses

这是上面内容的 zcurses 移植版。 输入处理稍微改进了一点，这样移动方块时不会自动推进一个时间步，并且图形使用了 Unicode 方块图形。

该版本不会在两次调用之间保存游戏状态，也不会以小部件的形式调用，而是以：

```
autoload -U tetriscurses
tetriscurses
```

zargs [*option* ... - -] [*input* ...] [- - *command* [*arg* ...]]

该函数的作用与 GNU xargs 类似。 它不是从标准输入中读取参数行，而是从命令行中获取参数。 这很有用，因为 zsh（尤其是使用递归 glob 操作符时）经常可以为 shell 函数构造出比外部命令所能接受的更长的命令行。

option 列表代表 zargs 命令本身的选项，与 xargs 的选项相同。 *input* 列表是字符串（通常是文件名）的集合，它们将成为 command 的参数，类似于 xargs 的标准输入。 最后，*arg* 列表包含每次运行 *command* 时传递给它的参数（通常是选项）。 每次运行时，*arg* 列表都位于 input 列表元素之前。如果没有提供 *command*，则不能提供 *arg* 列表。在这种情况下，默认命令是带有参数 '-r --' 的 'print'。

例如，要获取当前目录或其子目录中所有非隐藏普通（plain）文件的 ls 长列表：

```
autoload -U zargs
zargs -- **/*(.) -- ls -ld --
```

第一和第三次出现的 '--' 分别用来标记 zargs 和 ls 的选项的末尾，以防止文件名以 '-' 开头，而第二次出现的 '-' 则用来分隔文件列表和要运行的命令（'ls -ld --'）。

如果列表有可能是空的，也需要第一个 '--'：

```
zargs -r -- ./*.back(#qN) -- rm -f
```

如果字符串 '--' 是或可能是 *input*（输入），可以使用 -e 选项来更改输入结束标记。 需要注意的是，这**不会** 改变选项结束标记。 例如，使用 '..' 作为标记：

```
zargs -e.. -- **/*(.) .. ls -ld --
```

在该示例中，这是一个不错的选择，因为任何普通文件都不能命名为 '..'，但最佳的结束标记取决于具体情况。

选项 -i、-I、-l、-L 和 -n 与 xargs 中的用法略有不同。 zargs 没有输入行可供计数，因此 -l 和 -L 会对 *input* 列表进行计数，而 -n 则会对 *command* 的每次执行所传递的参数进行计数，**包括** 任何 *arg* 列表。 此外，在使用 -i 或 -I 时，每个 *input* 都会被单独处理，就像使用 '-L 1' 一样。

有关其他 zargs 选项的详情，请参阅 xargs(1) 手册（但请注意 zargs 和 xargs 在功能上的区别），或使用 --help 选项运行 zargs。

zed [-f [-x *num*]] *name*
zed [-h [*name*] *size*]
zed -b

该函数使用 ZLE 编辑器编辑文件或函数。

只允许使用一个 *name* 参数。 如果给出 -f 选项，名称将被视为函数名称；如果函数被标记为自动加载，zed 将在 fpath 中搜索并加载该函数。 请注意，以这种方式编辑的函数会安装到当前 shell 中，但 **不** 会写回自动加载文件。 在这种情况下，-x 选项指定将根据语法对函数进行缩进，前导制表符转换为给定数量的空格；'-x 2' 与 shell 随附的函数布局一致。

在没有 -f 的情况下，*name* 是要编辑的文件的路径名，该文件不一定存在；如有必要，会在写入时创建。 如果使用 -h，则假定文件包含历史事件。

如果没有为 -h 提供文件名，当前的 shell 历史记录将被编辑(in place)。 当 zed 成功退出后，历史记录将重新编号。

编辑历史记录时，多行事件在最后一行之前的每一行都必须有一个尾部反斜线。

编辑时，函数会将主键映射设置为 zed，将 vi 命令键映射设置为 zed-vicmd。 如果首次运行 zed 时 main 和 vicmd 键映射不存在，则将从现有的 main 和 vicmd 键映射中复制这些映射。 它们可用于提供仅在 zed 中使用的特殊按键绑定。

如果创建了键映射，zed 会重新绑定回车键以插入换行符，并绑定 '^X^W' 以接受 zed 键映射中的编辑，同时绑定 'ZZ' 以接受 zed-vicmd 键映射中的编辑。

运行 'zed -b' 可单独安装绑定。 这适合放入启动文件。 请注意，如果重新运行，将覆盖现有的 zed 和 zed-vicmd 键映射。

可使用补全功能，并可使用上下文前缀 ':completion:zed:' 设置样式。

可使用一个 zle 小部件 zed-set-file-name。你可以在 zed 内部使用'\ex zed-set-file-name' 调用它，也可以在运行 'zed -b' 后将它绑定到 zed 或 zed-vicmd 键映射中的某个键上。 调用该小部件时，它会提示为正在编辑的文件取一个新名称。 当 zed 退出时，文件将以该名称写入，而原始文件将保持不变。 从 'zed -f' 调用时，该小部件无效。 补全上下文更改为 ':completion:zed-set-file-name:'。 使用'zed -h' 编辑当前历史时，首先更新历史，然后写入文件，但 HISTFILE 的全局设置不会改变。

当 zed-set-file-name 运行时，zed 会使用 zed-normal-keymap 的键映射，该键映射与 zed 初始化绑定时有效的主键映射相连。 (这是为了让返回键正常工作。) 这样做的结果是，如果主键映射发生变化，小部件不会察觉。 这对大多数用户来说并不重要。

zcp [-finqQvwW] *srcpat dest*
zln [-finqQsvwW] *srcpat dest*

分别与 zmv -C 和 zmv -L 相同。 这些函数并未出现在 zsh 发行版中，但可以通过将 zmv 链接到 fpath 中的某个目录里的zcp 和 zln 来创建。

zkbd

请参阅 '键盘定义' (实用程序）。

zmv [-finqQsvwW] [-C | -L | -M | -{p|P} *program*] [-o *optstring*]
srcpat dest

移动（通常是重命名）与 *srcpat* 模式匹配的文件，到名称与 *dest* 所给形式相同的相应文件，其中 *srcpat* 包含围绕模式的括号，这些模式将依次被 *dest* 中的 $1、$2...替换。 例如

```
zmv '(*).lis' '$1.txt'
```

将 'foo.lis' 重命名为 'foo.txt' ，将'my.old.stuff.lis' 重命名为'my.old.stuff.lis' ，以此类推。

该模式始终被视为 EXTENDED_GLOB 模式。 任何文件,名称未通过替换而改变的，都会被忽略。 任何错误（一次替换结果为空字符串、两次替换结果相同、目标文件为现有常规文件且未给出 -f）都会导致整个函数终止，不会执行任何操作。

除了模式替换外，还可以在第二个（替换）参数中引用变量 $f。 这使得使用变量替换来更改参数成为可能；请参阅下面的示例。

选项:

-f

强制覆盖目标文件。 由于实现方式的不同，目前还不能向 mv/cp/ln 命令传递（但可以使用 -o-f 来实现）。

-i

交互式：显示要执行的每一行，并询问用户是否执行。 'Y' 或 'y' 将执行该行，否则将跳过该行。 请注意，您只需键入一个字符即可。

-n

不执行：打印会发生的事情，但不要去做。

-q

关闭裸 glob 限定符：现在默认为关闭，因此没有影响。

-Q

强制开启裸 glob 限定符。 除非您真的要在模式中使用 glob 限定符，否则不要打开它。

-s

符号，向下传递至 ln；仅适用于 -L。

-v

详细：执行每条命令时打印执行的命令。

`-w`

如上文所述，挑出模式中的通配符部分，并隐式添加括号用于引用它们。

`-W`

就像 `-w`一样，只是把替换模式中的通配符变成了连续的 `${1}` .. `${N}` 引用。

`-C`
`-L`
`-M`

分别强制执行 `cp`、`ln` 或 `mv`，与函数名称无关。

`-p` *program*

调用 *program* 而不是 `cp`、`ln` 或 `mv`。 无论它做什么，至少应该理解以下形式

```
program -- oldname newname
```

其中 *oldname* 和 *newname* 是 `zmv` 生成的文件名。 *program* 将被分割成单词，因此可能是归档工具的名称加上复制或重命名子命令。

`-P` *program*

`-p` *program* 除了 *program* 不接受后面的 `--` 来表示选项的结束外，其他都与 *program* 相同。 在这种情况下，文件名必须已经是相关程序的正常形式。

`-o` *optstring*

optstring 会被分割成单词，并逐字传给 `cp`、`ln` 或 `mv` 命令来执行工作。 这也许应该以'-'开头。

更多例子:

```
zmv -v '(* *)' '${1// /_}'
```

对于当前目录中名称中至少有一个空格的文件，用下划线替换每个空格，并显示执行的命令。

```
zmv -v '* *' '${f// /_}'
```

通过引用 `$f` 中存储的文件名，可以实现完全相同的功能。

有关更完整的示例和其他实现细节，请参阅 `zmv` 源文件，该文件通常位于 `fpath` 中命名的某个目录中，或 zsh 发行版中的 `Functions/Misc/zmv` 中。

`zrecompile`

请参阅 '重新编译函数' (实用程序）。

`zstyle+` *context style value* [+ *subcontext style value* ...]

通过使用单个 '+' 作为特殊标记，可以将上下文名称追加到之前使用的上下文名称中，从而使样式定义变得更加简单。 就像这样

```
zstyle+ ':foo:bar' style1 value1 \
       +':baz'     style2 value2 \
       +':frob'    style3 value3
```

与往常一样，这里为上下文 `:foo:bar` 定义了 *style1* 和 *value1*，但同时也为上下文 `:foo:bar:baz` 定义了 *style2* 和 *value2*，并为 `:foo:bar:frob` 定义了 *style3* 和 *value3*。 任何 *subcontext* 都可以是空字符串，以便在不改变上下文的情况下重新使用第一个上下文。

26.12.2 样式

`insert-tab`

zed 函数 在'`:completion:zed:*`'中**设置** 了该样式，以便在行首键入 TAB 时关闭补全功能。 您可以为该上下文和样式设置自己的值，从而覆盖该设置。

`pager`

nslookup 函数在上下文'`:nslookup`'中查找这种样式，以确定用于显示无法在单个屏幕上显示的输出的程序。

`prompt`
`rprompt`

nslookup 函数在上下文'`:nslookup`'中查找此样式，分别设置提示符和右侧提示符。 可以使用 PS1 和 RPS1 参数的常规扩展（参见 提示符扩展）。

概念索引

D

E

F

G

T

U

W

Z

历

参

整

文

行

补

选

递

通

邮

重

钩

变量索引

Index Entry	Section
_	
_	由 Shell 设置的参数
-	
-	由 Shell 设置的参数
!	
!	由 Shell 设置的参数
?	
?	由 Shell 设置的参数
@	
@	由 Shell 设置的参数
*	
*	由 Shell 设置的参数
#	
#	由 Shell 设置的参数

J

K

L

M

N

O

P

Q

R

Y

Z

选项索引

B

C

D

E

F

G

H

I

K

L

M

N

NO_AUTO_RESUME	选项说明
NO_BAD_PATTERN	选项说明
NO_BANG_HIST	选项说明
NO_BARE_GLOB_QUAL	选项说明
NO_BASH_AUTO_LIST	选项说明
NO_BASH_REMATCH	选项说明
NO_BEEP	选项说明
NO_BG_NICE	选项说明
NO_BRACE_CCL	选项说明
NO_BRACE_EXPAND	Option 别名
NO_BSD_ECHO	选项说明
NO_C_BASES	选项说明
NO_C_PRECEDENCES	选项说明
NO_CASE_GLOB	选项说明
NO_CASE_MATCH	选项说明
NO_CASE_MATCH	zsh/pcre 模块
NO_CASE_PATHS	选项说明
NO_CD_SILENT	选项说明
NO_CDABLE_VARS	选项说明
NO_CHASE_DOTS	选项说明
NO_CHASE_LINKS	选项说明
NO_CHECK_JOBS	选项说明
NO_CHECK_RUNNING_JOBS	选项说明
NO_CLOBBER	选项说明
NO_CLOBBER_EMPTY	选项说明
NO_COMBINING_CHARS	选项说明
NO_COMPLETE_ALIASES	选项说明
NO_COMPLETE_IN_WORD	选项说明
NO_CONTINUE_ON_ERROR	选项说明
NO_CORRECT	选项说明
NO_CORRECT_ALL	选项说明
NO_CSH_JUNKIE_HISTORY	选项说明
NO_CSH_JUNKIE_LOOPS	选项说明
NO_CSH_JUNKIE_QUOTES	选项说明
NO_CSH_NULL_GLOB	选项说明
NO_CSH_NULLCMD	选项说明
NO_DEBUG_BEFORE_CMD	选项说明

NO_HIST_LEX_WORDS	选项说明
NO_HIST_NO_FUNCTIONS	选项说明
NO_HIST_NO_STORE	选项说明
NO_HIST_REDUCE_BLANKS	选项说明
NO_HIST_SAVE_BY_COPY	选项说明
NO_HIST_SAVE_NO_DUPS	选项说明
NO_HIST_SUBST_PATTERN	选项说明
NO_HIST_VERIFY	选项说明
NO_HUP	选项说明
NO_IGNORE_BRACES	选项说明
NO_IGNORE_CLOSE_BRACES	选项说明
NO_IGNORE_EOF	选项说明
NO_INC_APPEND_HISTORY	选项说明
NO_INC_APPEND_HISTORY_TIME	选项说明
NO_INTERACTIVE	选项说明
NO_INTERACTIVE_COMMENTS	选项说明
NO_KSH_ARRAYS	选项说明
NO_KSH_AUTOLOAD	选项说明
NO_KSH_GLOB	选项说明
NO_KSH_OPTION_PRINT	选项说明
NO_KSH_TYPESET	选项说明
NO_KSH_ZERO_SUBSCRIPT	选项说明
NO_LIST_AMBIGUOUS	选项说明
NO_LIST_BEEP	选项说明
NO_LIST_PACKED	选项说明
NO_LIST_ROWS_FIRST	选项说明
NO_LIST_TYPES	选项说明
NO_LOCAL_LOOPS	选项说明
NO_LOCAL_OPTIONS	选项说明
NO_LOCAL_PATTERNS	选项说明
NO_LOCAL_TRAPS	选项说明
NO_LOG	Option 别名
NO_LOGIN	选项说明
NO_LONG_LIST_JOBS	选项说明
NO_MAGIC_EQUAL_SUBST	选项说明
NO_MAIL_WARN	Option 别名
NO_MAIL_WARNING	选项说明

NO_MARK_DIRS	选项说明
NO_MENU_COMPLETE	选项说明
NO_MONITOR	选项说明
NO_MULTI_FUNC_DEF	选项说明
NO_MULTIBYTE	选项说明
NO_MULTIOS	选项说明
NO_NOMATCH	选项说明
NO_NOTIFY	选项说明
NO_NULL_GLOB	选项说明
NO_NUMERIC_GLOB_SORT	选项说明
NO_OCTAL_ZEROES	选项说明
NO_ONE_CMD	Option 别名
NO_OVERSTRIKE	选项说明
NO_PATH_DIRS	选项说明
NO_PATH_SCRIPT	选项说明
NO_PHYSICAL	Option 别名
NO_PIPE_FAIL	选项说明
NO_POSIX_ALIASES	选项说明
NO_POSIX_ARGZERO	选项说明
NO_POSIX_BUILTINS	选项说明
NO_POSIX_CD	选项说明
NO_POSIX_IDENTIFIERS	选项说明
NO_POSIX_JOBS	选项说明
NO_POSIX_STRINGS	选项说明
NO_POSIX_TRAPS	选项说明
NO_PRINT_EIGHT_BIT	选项说明
NO_PRINT_EXIT_VALUE	选项说明
NO_PRIVILEGED	选项说明
NO_PROMPT_BANG	选项说明
NO_PROMPT_CR	选项说明
NO_PROMPT_PERCENT	选项说明
NO_PROMPT_SP	选项说明
NO_PROMPT_SUBST	选项说明
NO_PROMPT_VARS	Option 别名
NO_PUSHD_IGNORE_DUPS	选项说明
NO_PUSHD_MINUS	选项说明
NO_PUSHD_SILENT	选项说明

NOALIASES	选项说明
NOALIASFUNCDEF	选项说明
NOALLEXPORT	选项说明
NOALWAYSLASTPROMPT	选项说明
NOALWAYSTOEND	选项说明
NOAPPENDCREATE	选项说明
NOAPPENDHISTORY	选项说明
NOAUTOCD	选项说明
NOAUTOCONTINUE	选项说明
NOAUTOLIST	选项说明
NOAUTOMENU	选项说明
NOAUTONAMEDIRS	选项说明
NOAUTOPARAMKEYS	选项说明
NOAUTOPARAMSLASH	选项说明
NOAUTOPUSHD	选项说明
NOAUTOREMOVESLASH	选项说明
NOAUTORESUME	选项说明
NOBADPATTERN	选项说明
NOBANGHIST	选项说明
NOBAREGLOBQUAL	选项说明
NOBASHAUTOLIST	选项说明
NOBASHREMATCH	选项说明
NOBEEP	选项说明
NOBGNICE	选项说明
NOBRACECCL	选项说明
NOBRACEEXPAND	Option 别名
NOBSDECHO	选项说明
NOCASEGLOB	选项说明
NOCASEMATCH	选项说明
NOCASEPATHS	选项说明
NOCBASES	选项说明
NOCDABLEVARS	选项说明
NOCDSILENT	选项说明
NOCHASEDOTS	选项说明
NOCHASELINKS	选项说明
NOCHECKJOBS	选项说明
NOCHECKRUNNINGJOBS	选项说明

NOCLOBBER	选项说明
NOCLOBBEREMPTY	选项说明
NOCOMBININGCHARS	选项说明
NOCOMPLETEALIASES	选项说明
NOCOMPLETEINWORD	选项说明
NOCONTINUEONERROR	选项说明
NOCORRECT	选项说明
NOCORRECTALL	选项说明
NOCPRECEDENCES	选项说明
NOCSHJUNKIEHISTORY	选项说明
NOCSHJUNKIELOOPS	选项说明
NOCSHJUNKIEQUOTES	选项说明
NOCSHNULLCMD	选项说明
NOCSHNULLGLOB	选项说明
NODEBUGBEFORECMD	选项说明
NODOTGLOB	Option 别名
NODVORAK	选项说明
NOEMACS	选项说明
NOEQUALS	选项说明
NOERREXIT	选项说明
NOERRRETURN	选项说明
NOEVALLINENO	选项说明
NOEXEC	选项说明
NOEXTENDEDGLOB	选项说明
NOEXTENDEDHISTORY	选项说明
NOFLOWCONTROL	选项说明
NOFORCEFLOAT	选项说明
NOFUNCTIONARGZERO	选项说明
NOGLOB	选项说明
NOGLOBALEXPORT	选项说明
NOGLOBALRCS	选项说明
NOGLOBASSIGN	选项说明
NOGLOBCOMPLETE	选项说明
NOGLOBDOTS	选项说明
NOGLOBSTARSHORT	选项说明
NOGLOBSUBST	选项说明
NOHASHALL	Option 别名

NOHASHCMDS	选项说明
NOHASHDIRS	选项说明
NOHASHEXECUTABLESONLY	选项说明
NOHASHLISTALL	选项说明
NOHISTALLOWCLOBBER	选项说明
NOHISTAPPEND	Option 别名
NOHISTBEEP	选项说明
NOHISTEXPAND	Option 别名
NOHISTEXPIREDUPSFIRST	选项说明
NOHISTFCNTLLOCK	选项说明
NOHISTFINDNODUPS	选项说明
NOHISTIGNOREALLDUPS	选项说明
NOHISTIGNOREDUPS	选项说明
NOHISTIGNORESPACE	选项说明
NOHISTLEXWORDS	选项说明
NOHISTNOFUNCTIONS	选项说明
NOHISTNOSTORE	选项说明
NOHISTREDUCEBLANKS	选项说明
NOHISTSAVEBYCOPY	选项说明
NOHISTSAVENODUPS	选项说明
NOHISTSUBSTPATTERN	选项说明
NOHISTVERIFY	选项说明
NOHUP	选项说明
NOIGNOREBRACES	选项说明
NOIGNORECLOSEBRACES	选项说明
NOIGNOREEOF	选项说明
NOINCAPPENDHISTORY	选项说明
NOINCAPPENDHISTORYTIME	选项说明
NOINTERACTIVE	选项说明
NOINTERACTIVECOMMENTS	选项说明
NOKSHARRAYS	选项说明
NOKSHAUTOLOAD	选项说明
NOKSHGLOB	选项说明
NOKSHOPTIONPRINT	选项说明
NOKSHTYPESET	选项说明
NOKSHZEROSUBSCRIPT	选项说明
NOLISTAMBIGUOUS	选项说明

NOPOSIXIDENTIFIERS	选项说明
NOPOSIXJOBS	选项说明
NOPOSIXSTRINGS	选项说明
NOPOSIXTRAPS	选项说明
NOPRINTEIGHTBIT	选项说明
NOPRINTEXITVALUE	选项说明
NOPRIVILEGED	选项说明
NOPROMPTBANG	选项说明
NOPROMPTCR	选项说明
NOPROMPTPERCENT	选项说明
NOPROMPTSP	选项说明
NOPROMPTSUBST	选项说明
NOPROMPTVARS	Option 别名
NOPUSHDIGNOREDUPS	选项说明
NOPUSHDMINUS	选项说明
NOPUSHDSILENT	选项说明
NOPUSHDTOHOME	选项说明
NORCEXPANDPARAM	选项说明
NORCQUOTES	选项说明
NORCS	选项说明
NORECEXACT	选项说明
NOREMATCHPCRE	选项说明
NORESTRICTED	选项说明
NORMSTARSILENT	选项说明
NORMSTARWAIT	选项说明
NOSHAREHISTORY	选项说明
NOSHFILEEXPANSION	选项说明
NOSHGLOB	选项说明
NOSHINSTDIN	选项说明
NOSHNULLCMD	选项说明
NOSHOPTIONLETTERS	选项说明
NOSHORTLOOPS	选项说明
NOSHORTREPEAT	选项说明
NOSHWORDSPLIT	选项说明
NOSINGLECOMMAND	选项说明
NOSINGLELINEZLE	选项说明
NOSOURCETRACE	选项说明

O

P

R

S

T

函数索引

_external_pwds	控制函数
_files	补全函数
_generic	可绑定命令
_gnu_generic	补全函数
_guard	补全函数
_history	控制函数
_history_complete_word (\e/)	可绑定命令
_ignored	控制函数
_list	控制函数
_match	控制函数
_menu	控制函数
_message	补全函数
_most_recent_file (^Xm)	可绑定命令
_multi_parts	补全函数
_next_label	补全函数
_next_tags (^Xn)	可绑定命令
_normal	补全函数
_numbers	补全函数
_oldlist	控制函数
_options	补全函数
_options_set	补全函数
_options_unset	补全函数
_parameters	补全函数
_path_files	补全函数
_pick_variant	补全函数
_precommand	控制函数
_prefix	控制函数
_read_comp (^X^R)	可绑定命令
_regex_arguments	补全函数
_regex_words [-t term]	补全函数
_requested	补全函数
_retrieve_cache	补全函数
_sep_parts	补全函数
_sequence	补全函数
_setup	补全函数
_store_cache	补全函数
_tags	补全函数

C

U

V

W

Z

编辑器函数索引

backward-word	移动
backward-word-match	ZLE 函数
beep	杂项
beginning-of-buffer-or-history	历史控制
beginning-of-history	历史控制
beginning-of-line	移动
beginning-of-line-hist	历史控制
bracketed-paste	杂项
bracketed-paste-magic	ZLE 函数

C

capitalize-word	修改文本
capitalize-word-match	ZLE 函数
clear-screen	杂项
complete-word	补全
copy-earlier-word	ZLE 函数
copy-prev-shell-word	修改文本
copy-prev-word	修改文本
copy-region-as-kill	修改文本
cycle-completion-positions	ZLE 函数

D

deactivate-region	杂项
delete-char	修改文本
delete-char-or-list	补全
delete-to-char	zsh/deltochar 模块
delete-whole-word-match	ZLE 函数
delete-whole-word-match	ZLE 函数
delete-word	修改文本
describe-key-briefly	杂项
digit-argument	实参
down-case-word	修改文本
down-case-word-match	ZLE 函数
down-history	历史控制
down-line	移动
down-line-or-beginning-search	ZLE 函数
down-line-or-history	历史控制

history-incremental-search-backward	历史控制
history-incremental-search-forward	历史控制
history-pattern-search	ZLE 函数
history-pattern-search-backward	ZLE 函数
history-pattern-search-forward	ZLE 函数
history-search-backward	历史控制
history-search-forward	历史控制

I

incarg	ZLE 函数
incremental-complete-word	ZLE 函数
infer-next-history	历史控制
insert-composed-char	ZLE 函数
insert-files	ZLE 函数
insert-last-word	历史控制
insert-unicode-char	ZLE 函数

K

kill-buffer	修改文本
kill-line	修改文本
kill-region	修改文本
kill-whole-line	修改文本
kill-word	修改文本
kill-word-match	ZLE 函数

L

list-choices	补全
list-expand	补全

M

magic-space	补全
match-word-context	ZLE 函数
match-words-by-style	ZLE 函数
menu-complete	补全
menu-expand-or-complete	补全
menu-select	zsh/complist 模块
modify-current-argument	ZLE 函数

N

O

P

Q

R

S

V

vi-add-eol	修改文本
vi-add-next	修改文本
vi-backward-blank-word	移动
vi-backward-blank-word-end	移动
vi-backward-char	移动
vi-backward-delete-char	修改文本
vi-backward-kill-word	修改文本
vi-backward-word	移动
vi-backward-word-end	移动
vi-beginning-of-line	移动
vi-caps-lock-panic	杂项
vi-change	修改文本
vi-change-eol	修改文本
vi-change-whole-line	修改文本
vi-cmd-mode	杂项
vi-delete	修改文本
vi-delete-char	修改文本
vi-digit-or-beginning-of-line	杂项
vi-down-case	修改文本
vi-down-line-or-history	历史控制
vi-end-of-line	移动
vi-fetch-history	历史控制
vi-find-next-char	移动
vi-find-next-char-skip	移动
vi-find-prev-char	移动
vi-find-prev-char-skip	移动
vi-first-non-blank	移动
vi-forward-blank-word	移动
vi-forward-blank-word-end	移动
vi-forward-char	移动
vi-forward-word	移动
vi-forward-word-end	移动
vi-goto-column	移动
vi-goto-mark	移动
vi-goto-mark-line	移动
vi-history-search-backward	历史控制

vi-history-search-forward	历史控制
vi-indent	修改文本
vi-insert	修改文本
vi-insert-bol	修改文本
vi-join	修改文本
vi-kill-eol	修改文本
vi-kill-line	修改文本
vi-match-bracket	修改文本
vi-open-line-above	修改文本
vi-open-line-below	修改文本
vi-oper-swap-case	修改文本
vi-pipe	ZLE 函数
vi-pound-insert	杂项
vi-put-after	修改文本
vi-put-before	修改文本
vi-quoted-insert	修改文本
vi-repeat-change	修改文本
vi-repeat-find	移动
vi-repeat-search	历史控制
vi-replace	修改文本
vi-replace-chars	修改文本
vi-rev-repeat-find	移动
vi-rev-repeat-search	历史控制
vi-set-buffer	杂项
vi-set-mark	杂项
vi-substitute	修改文本
vi-swap-case	修改文本
vi-undo-change	杂项
vi-unindent	修改文本
vi-up-case	修改文本
vi-up-line-or-history	历史控制
vi-yank	修改文本
vi-yank-eol	修改文本
vi-yank-whole-line	修改文本
visual-line-mode	杂项
visual-mode	杂项

样式和标签索引

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

样

目

类

设

说

Jump to: -
A B C D E F G H I J K L M N O P Q R S T U V W
Z 会 信 修 匹 后 字 帐 时 显 替 标 样 目 类 设 说