Bash 源码分析

周荣华

作者简介：10 年通讯底层研发经验，熟悉 linux/vxworks 等实时操作系统的内核原理和实现，在虚拟化的 openstack，kubernetes，docker 等领域也初有涉猎。

摘要：本文讲述当下留下的 linux 的 bash 的源代码，通过代码分析和单步调试解析 bash 的运行流程，适合喜欢研究 linux 原理的高级用户。分析的源代码来自 gnu 的开源项目 https://git.savannah.gnu.org/git/bash.git，例子是作者自己编写，可以随意引用。

1 引言

Bash 这个程序作为一个 linux 的用户，用的实在太频繁了，但一般局限于会用就结束了，一直没机会研究 bash 本身的原理。因工作需要，调试一个 bash 的 cpu 冲高问题，趁此机会对 bash 的源码做了一些研究，希望能对大家有点帮助。

2 linux 的各种主流 shell 介绍

现在一般使用的 shell 有 sh，bash 和 csh 这几种，我们这里主要说的是 bash，其他 shell 的源代码逻辑也差不多。

3 bash 使用到的主要数据结构介绍

3.1 COMMAND

```
00187:
00188: /* What a command looks like. */
00189: typedef struct command {
00190:   enum command_type type;   /* FOR CASE WHILE IF CONNECTION or SIMPLE. */
00191:   int flags;            /* Flags controlling execution environment. */
00192:   int line;         /* line number the command starts on */
00193:   REDIRECT *redirects;      /* Special redirects for FOR CASE, etc. */
00194:   union {
00195:     struct for_com *For;
00196:     struct case_com *Case;
00197:     struct while_com *While;
00198:     struct if_com *If;
00199:     struct connection *Connection;
00200:     struct simple_com *Simple;
00201:     struct function_def *Function_def;
00202:     struct group_com *Group;
00203: #if defined (SELECT_COMMAND)
00204:     struct select_com *Select;
00205: #endif
00206: #if defined (DPAREN_ARITHMETIC)
00207:     struct arith_com *Arith;
00208: #endif
00209: #if defined (COND_COMMAND)
00210:     struct cond_com *Cond;
00211: #endif
00212: #if defined (ARITH_FOR_COMMAND)
00213:     struct arith_for_com *ArithFor;
00214: #endif
00215:     struct subshell_com *Subshell;
00216:     struct coproc_com *Coproc;
00217:   } value;
00218: } COMMAND;
```

COMMAND 是所有数据结构的纲，从这里可以看出一个 bash 实际能执行的语句有 14 种，分别位 for，case，while，fi，connection，simple_com，function，group，select，arith，cond，arith_for，subshell，coproc，其中 select，arith，cond，arith_for 这 4 个命令需要打开对应的编译开关之后才能执行。

除了下面的这个 union 外，另外几个属性分别对应命令类型，行号和执行环境控制参数。其中控制参数有很多，每个控制参数占用一个 bit 位，包括是否启动子 shell，是否忽略 exit 值等。

```
00171: /* Possible values for command->flags. */
00172: #define CMD_WANT_SUBSHELL  0x01 /* User wants a subshell: ( command ) */
00173: #define CMD_FORCE_SUBSHELL 0x02 /* Shell needs to force a subshell. */
00174: #define CMD_INVERT_RETURN  0x04 /* Invert the exit value. */
00175: #define CMD_IGNORE_RETURN  0x08 /* Ignore the exit value.  For set -e. */
00176: #define CMD_NO_FUNCTIONS   0x10 /* Ignore functions during command lookup. */
00177: #define CMD_INHIBIT_EXPANSION 0x20 /* Do not expand the command words. */
00178: #define CMD_NO_FORK    0x40 /* Don't fork; just call execve */
00179: #define CMD_TIME_PIPELINE  0x80 /* Time a pipeline */
00180: #define CMD_TIME_POSIX     0x100 /* time -p; use POSIX.2 time output spec. */
00181: #define CMD_AMPERSAND      0x200 /* command & */
00182: #define CMD_STDIN_REDIR    0x400 /* async command needs implicit </dev/null */
00183: #define CMD_COMMAND_BUILTIN 0x0800 /* command executed by `command' builtin */
00184: #define CMD_COPROC_SUBSHELL 0x1000
00185: #define CMD_LASTPIPE       0x2000
00186: #define CMD_STDPATH     0x4000  /* use standard path for command lookup */
```

这些 flag 可以在 bash 启动 shell 脚本时设置，或者在 shell 脚本内部调用 set 指令来设置，一

般用户不怎么关注，高阶用户可以看看：

```
[root@RNC30 ~]# bash --help
GNU bash, version 3.2.25(1)-release-(x86_64-redhat-linux-gnu)
Usage:  bash [GNU long option] [option] ...
        bash [GNU long option] [option] script-file ...
GNU long options:
        --debug
        --debugger
        --dump-po-strings
        --dump-strings
        --help
        --init-file
        --login
        --noediting
        --noprofile
        --norc
        --posix
        --protected
        --rcfile
        --rpm-requires
        --restricted
        --verbose
        --version
        --wordexp
Shell options:
        -irsD or -c command or -O shopt_option          (invocation only)
        -abefhkmnptuvxBCHP or -o option
Type `bash -c "help set"' for more information about shell options.
Type `bash -c help' for more information about shell builtin commands.
Use the `bashbug' command to report bugs.
```

3.2 FOR_COM

```
00250: /* FOR command. */
00251: typedef struct for_com {
00252:   int flags;        /* See description of CMD flags. */
00253:   int line;     /* line number the `for' keyword appears on */
00254:   WORD_DESC *name;  /* The variable name to get mapped over. */
00255:   WORD_LIST *map_list;  /* The things to map over.  This is never NULL. */
00256:   COMMAND *action;  /* The action to execute.
00257:                During execution, NAME is bound to successive
00258:                members of MAP_LIST. */
00259: } FOR_COM;
```

FOR_COM 对应的 shell 语句是 for name in map_list; do action; done

从结构体定义可以看出，除了和 COMMAND 相同的 flags 和行号外，for 语句是有一个变量名，一个列表和一个递归的 COMMAND 组成的，实际 for 循环执行过程中也是将列表中的每个元素拿出来赋值给变量名，并执行 action 中的脚本段。

从这里的 flags，可以看出，每条命令的 flags 是可以单独设置的，本条命令设置的控制参数可以不影响其他命令的控制参数。

3.3 CASE_COM

```
00234: /* Pattern/action structure for CASE_COM. */
00235: typedef struct pattern_list {
00236:   struct pattern_list *next;    /* Clause to try in case this one failed. */
00237:   WORD_LIST *patterns;      /* Linked list of patterns to test. */
00238:   COMMAND *action;      /* Thing to execute if a pattern matches. */
00239:   int flags;
00240: } PATTERN_LIST;
00241:
00242: /* The CASE command. */
00243: typedef struct case_com {
00244:   int flags;            /* See description of CMD flags. */
00245:   int line;         /* line number the `case' keyword appears on */
00246:   WORD_DESC *word;      /* The thing to test. */
00247:   PATTERN_LIST *clauses;    /* The clauses to test against, or NULL. */
00248: } CASE_COM;
```

对照下面的脚本，可以看出，先判断一个变量，变量判断晚走到复合语句 clauses，注意 clause 最终实现的时候是一个单向链表，链表中每个元素由一个样式的列表和一个执行体 action 来组成。

```
1   case word in
2       pattern1|pattern2|pattern3 )
3           action1
4           exit
5       ;;
6
7       pattern4 )
8           action4
9           exit
10      ;;
11
12
13      * )
14          actionx
15  esac
```

3.4 WHILE_COM

```
00293: /* WHILE command. */
00294: typedef struct while_com {
00295:   int flags;            /* See description of CMD flags. */
00296:   COMMAND *test;        /* Thing to test. */
00297:   COMMAND *action;      /* Thing to do while test is non-zero. */
00298: } WHILE_COM;
00299:
```

WHILE_COM 比较简单，主体有两部分组成，判断条件和执行体。上篇文章调试过程中导致 CPU 冲高到 100%的例子就是用的 WHILE_COM，不过例子中的 WHILE_COM 的判断条件留空，相当于永远为 true。

```
while :; do i=$((i+1));   done
```

3.5 IF_COM

```
00285: /* IF command. */
00286: typedef struct if_com {
00287:   int flags;            /* See description of CMD flags. */
00288:   COMMAND *test;        /* Thing to test. */
00289:   COMMAND *true_case;       /* What to do if the test returned non-zero. */
00290:   COMMAND *false_case;      /* What to do if the test returned zero. */
00291: } IF_COM;
```

IF_COM 由 3 段组成，条件判断，条件为 true 时的执行体和条件为 false 时的执行体，其中 false 情况下的执行体可以为空。

```
1  if [ -z $1]; then
2      echo "hello true"
3  else
4      echo "hello false"
5  fi
6
```

从 IF_COM 的定义看，3 部分都可以是复杂的 COMMAND 结构，所以嵌套起来也可以做的非常复杂，例如可以在 test 部分通过执行脚本，依靠脚本的返回值来判断是应该执行 true_case 还是 false_case。

3.6 CONNECTION

```
00220: /* Structure used to represent the CONNECTION type. */
00221: typedef struct connection {
00222:   int ignore;           /* Unused; simplifies make_command (). */
00223:   COMMAND *first;       /* Pointer to the first command. */
00224:   COMMAND *second;      /* Pointer to the second command. */
00225:   int connector;        /* What separates this command from others. */
00226: } CONNECTION;
```

CONNECTION 由 4 个属性组成：ignore 字段对应其他命令结构种的 flags，但对连接命令实际上没有用 first 对应第一条命令，second 对应第二条命令，connector 对应两条命令之间的连接符。难道只能两个命令一起用，不能多余两个命令一起调用？显然不是，一个 CONNECTION 对应一个连接符连接起来的两段命令，每段命令又可以是一个 CONNECTION，这样就形成了级联的效果。

CONNECTION 有 3 种：AND_AND 对应“&&”，表示 first 执行返回结果为 0 的时候执行 second；OR_OR 对应“||”，表示 first 执行返回结果为非 0 的时候执行 second；分号对应的 connector 还是分号，表示无论 first 执行结果是 0 还是非 0，都执行 second。是不是有点像 C 语言里面的&&，||和;?

```
6
7   cd $dir && rm -fr *
8   cd $dir || rm -fr *
9   cd $dir ; rm -fr *
10
```

上面的三个例子别真的执行，后果很严重(☺)。第一条表示删除$dir 对应值的目录中的所有文件；第二条表示$dir 不存在的时候删除当前目录下面的所有文件；第三条表示，如果$dir 存在就删除$dir 目录下的所有文件，如果不存在就删除当前目录下面的所有文件。

3.7 SIMPLE_COM

```
00328: /* The "simple" command.  Just a collection of words and redirects. */
00329: typedef struct simple_com {
00330:   int flags;            /* See description of CMD flags. */
00331:   int line;         /* line number the command starts on */
00332:   WORD_LIST *words;     /* The program name, the arguments,
00333:                    variable assignments, etc. */
00334:   REDIRECT *redirects;      /* Redirections to perform. */
00335: } SIMPLE_COM;
```

SIMPLE_COM 按字面意思就是简单命令，结构体由四部分组成，通用的 flags，行号 line，命令队列 WORD_LIST，重定向队列 REDIRECT。

WORD_LIST 队列比较容易理解。一堆命令的集合：

```
00129: /* A linked list of words. */
00130: typedef struct word_list {
00131:   struct word_list *next;
00132:   WORD_DESC *word;
00133: } WORD_LIST;
```

其中单个命令还可以独立设置 flags：

```
00123: /* A structure which represents a word. */
00124: typedef struct word_desc {
00125:   char *word;       /* Zero terminated string. */
00126:   int flags;        /* Flags associated with this word. */
00127: } WORD_DESC;
```

REDIRECT 就是重定向的意思，这里也有很多种重定向，结构体的各个属性的含义：next，组成重定向链表的指针；redirector，重定向的源；rflags，重定向时使用的私有 flags；flags，打开重定向目标文件时的 flags；instruction，重定向的实际功能指令，这个又有很多种，下面会详细描述；redirectee，重定向的目的文件描述符或者文件名；here_doc_eof，本地文件。

```
00152: /* Structure describing a redirection.  If REDIRECTOR is negative, the parser
00153:    (or translator in redir.c) encountered an out-of-range file descriptor. */
00154: typedef struct redirect {
00155:   struct redirect *next;    /* Next element, or NULL. */
00156:   REDIRECTEE redirector;    /* Descriptor or varname to be redirected. */
00157:   int rflags;           /* Private flags for this redirection */
00158:   int flags;            /* Flag value for `open'. */
00159:   enum r_instruction  instruction; /* What to do with the information. */
00160:   REDIRECTEE redirectee;    /* File descriptor or filename */
00161:   char *here_doc_eof;       /* The word that appeared in <<foo. */
00162: } REDIRECT;
```

从 REDIRECTEE 的定义看，它既可以是一个文件描述符，例如 0 表示标准输入，1 表示标准输出，2 表示错误输出，也可以是一个文件名。

```
00147: typedef union {
00148:   int dest;          /* Place to redirect REDIRECTOR to, or ... */
00149:   WORD_DESC *filename;      /* filename to redirect to. */
00150: } REDIRECTEE;
```

Bash 支持二十种不同的重定向，后面会根据 bash 的源代码来一一解释一下具体内容（bash 源代码的注释对重定向的含义理解也有很多帮助）：

```
00027: /* Instructions describing what kind of thing to do for a redirection. */
00028: enum r_instruction {
00029:   r_output_direction, r_input_direction, r_inputa_direction,
00030:   r_appending_to, r_reading_until, r_reading_string,
00031:   r_duplicating_input, r_duplicating_output, r_deblank_reading_until,
00032:   r_close_this, r_err_and_out, r_input_output, r_output_force,
00033:   r_duplicating_input_word, r_duplicating_output_word,
00034:   r_move_input, r_move_output, r_move_input_word, r_move_output_word,
00035:   r_append_err_and_out
00036: };
```

先来五种输出的重定向：普通输出，强制输出，错误和标准输出，叠加输出，错误和标准叠加输出。统一说一下几个概念，标准输出就是 2 级 stdout，错误输出就是 3 级 sdterr，强制的意思是文件存在的情况下会被先清空，再增加，叠加输出的意思是原有内容后面再增加。

```
00712:     case r_output_direction:        /* >foo */
00713:     case r_output_force:           /* >| foo */
00714:     case r_err_and_out:            /* &>filename */
00715:       temp->flags = O_TRUNC | O_WRONLY | O_CREAT;
00716:       break;
00717:
00718:     case r_appending_to:           /* >>foo */
00719:     case r_append_err_and_out:        /* &>> filename */
00720:       temp->flags = O_APPEND | O_WRONLY | O_CREAT;
00721:       break;
00722:
```

再来九种输入和输出重定向：普通输入重定向，后台执行，输入和输出同时重定向，去掉空格的输入重定向，输入重定向，字符串作为输入，关闭重定向源（怎么还有这种应用场景？），复制输入，复制输出。

```
00723:     case r_input_direction:     /* <foo */
00724:     case r_inputa_direction:        /* foo & makes this. */
00725:       temp->flags = O_RDONLY;
00726:       break;
00727:
00728:     case r_input_output:           /* <>foo */
00729:       temp->flags = O_RDWR | O_CREAT;
00730:       break;
00731:
00732:     case r_deblank_reading_until:   /* <<-foo */
00733:     case r_reading_until:          /* << foo */
00734:     case r_reading_string:         /* <<< foo */
00735:     case r_close_this:             /* <&- */
00736:     case r_duplicating_input:         /* 1<&2 */
00737:     case r_duplicating_output:        /* 1>&2 */
00738:       break;
```

紧接着六种输入输出的重定向分别为输入剪切，输出剪切，字符指向的输入剪切和字符指向的输出剪切，字符指向的输入复制和字符指向的输出复制。

```
00740:      /* the parser doesn't pass these. */
00741:      case r_move_input:          /* 1<&2- */
00742:      case r_move_output:         /* 1>&2- */
00743:      case r_move_input_word:     /* 1<&$foo- */
00744:      case r_move_output_word:        /* 1>&$foo- */
00745:        break;
00746:
00747:      /* The way the lexer works we have to do this here. */
00748:      case r_duplicating_input_word:  /* 1<&$foo */
00749:      case r_duplicating_output_word: /* 1>&$foo */
```

Bash 的重定向真是博大精深!! 明天继续其他 COMMAND 的定义讲解。

3.8 FUNCTION_DEF

```
00337: /* The "function definition" command. */
00338: typedef struct function_def {
00339:   int flags;            /* See description of CMD flags. */
00340:   int line;         /* Line number the function def starts on. */
00341:   WORD_DESC *name;      /* The name of the function. */
00342:   COMMAND *command;     /* The parsed execution tree. */
00343:   char *source_file;        /* file in which function was defined, if any */
00344: } FUNCTION_DEF;
```

FUNCTION_DEF 由五部分组成，通用 flags，起始行号，函数名，解析之后的函数执行体，如果函数定义在文件中，最后会有文件名。函数的定义也可以有入参，入参的提取和文件执行时类似的，都是走$1,$2 类似的形式获得的。

```
11  function e {
12      echo $1
13  }
```

3.9 GROUP_COM

```
00346: /* A command that is `grouped' allows pipes and redirections to affect all
00347:    commands in the group. */
00348: typedef struct group_com {
00349:   int ignore;           /* See description of CMD flags. */
00350:   COMMAND *command;
00351: } GROUP_COM;
```

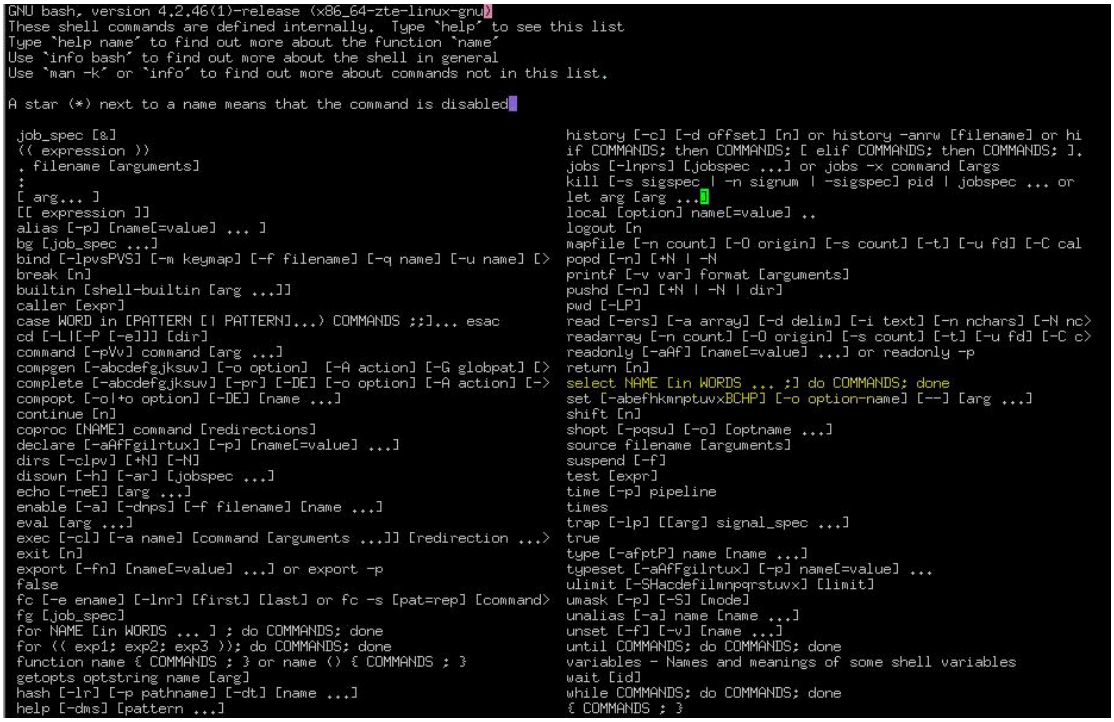
GROUP_COM 是个什么鬼？通过分析 group 的处理函数，发现 group 原来就是多个命令组成的命令段，一般用{}包围起来，从 group_command_nesting 变量的变化看，group 是支持多层嵌套的。

```
00663: static void
00664: print_group_command (group_command)
00665:      GROUP_COM *group_command;
00666: {
00667:   group_command_nesting++;
00668:   cprintf ("{ ");
00669:
00670:   if (inside_function_def == 0)
00671:     skip_this_indent++;
00672:   else
00673:     {
00674:       /* This is a group command { ... } inside of a function
00675:      definition, and should be printed as a multiline group
00676:      command, using the current indentation. */
00677:       cprintf ("\n");
00678:       indentation += indentation_amount;
00679:     }
00680:
00681:   make_command_string_internal (group_command->command);
00682:   PRINT_DEFERRED_HEREDOCS ("");
00683:
00684:   if (inside_function_def)
00685:     {
00686:       cprintf ("\n");
00687:       indentation -= indentation_amount;
00688:       indent (indentation);
00689:     }
00690:   else
00691:     {
00692:       semicolon ();
00693:       cprintf (" ");
00694:     }
00695:
00696:   cprintf ("}");
00697:
00698:   group_command_nesting--;
00699: }
```

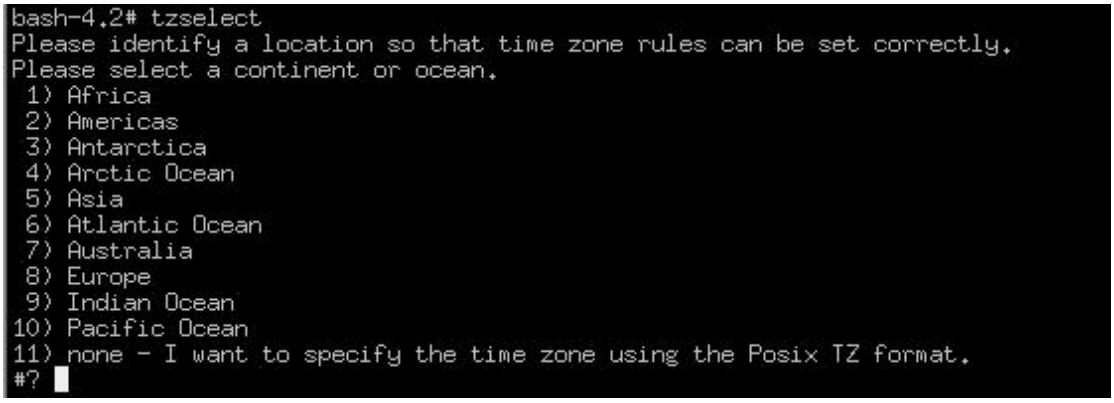
3.10 SELECT_COM

```
00272: #if defined (SELECT_COMMAND)
00273: /* KSH SELECT command. */
00274: typedef struct select_com {
00275:   int flags;        /* See description of CMD flags. */
00276:   int line;     /* line number the `select' keyword appears on */
00277:   WORD_DESC *name;  /* The variable name to get mapped over. */
00278:   WORD_LIST *map_list;  /* The things to map over.  This is never NULL. */
00279:   COMMAND *action;  /* The action to execute.
00280:                During execution, NAME is bound to the member of
00281:                MAP_LIST chosen by the user. */
00282: } SELECT_COM;
00283: #endif /* SELECT_COMMAND */
```

SELECT_COM 并不是每个版本的 bash 都存在，可以通过在 bash 里面敲 help 来确定其是否存在。下面这个 bash 4.2.46 的版本中是打开了 select 的开关的：



SELECT_COM 的作用是为了生成一个简单的菜单，用户通过选择菜单来让系统执行对应的命令，常见的 SELECT_COM 是时区配置时使用的。



具体分析/usr/bin/tzselect 源码时发现，为了做到各个 shell 之间的兼容，这个脚本写的比想象中要复杂的多。

首先要是一下版本号的记录：

跳过紧接着的注释，然后是版本兼容性判断，使用帮助：

```
31
32  # Specify default values for environment variables if they are unset.
33  : ${AWK=awk}
34  : ${TZDIR=/usr/share/zoneinfo}
35
36  # Check for awk Posix compliance.
37  ($AWK -v x=y 'BEGIN { exit 123 }') </dev/null >/dev/null 2>&1
38  [ $? = 123 ] || {
39          echo >&2 "$0: Sorry, your \`$AWK' program is not Posix compatible."
40          exit 1
41  }
42
43  if [ "$1" = "--help" ]; then
44      cat <<EOF
45  Usage: tzselect
46  Select a time zone interactively.
47
48  Report bugs to $REPORT_BUGS_TO.
49  EOF
50      exit
51  elif [ "$1" = "--version" ]; then
52      cat <<EOF
53  tzselect $PKGVERSION$TZVERSION
54  EOF
55      exit
56  fi
57
```

然后很无聊的定义了一个完全不用的变量 IFS，但用来定义 IFS 的 newline 后面倒是用过。为了规避 bug，还要把 PS3 清空。

```
68
69  newline='
70  '
71  IFS=$newline
72
73
74  # Work around a bug in bash 1.14.7 and earlier, where $PS3 is sent to stdout.
75  case $(echo 1 | (select x in x; do break; done) 2>/dev/null) in
76  ?*) PS3=
77  esac
78
```

终于进正题了，先选择大洲或者大洋：

```
# Begin the main loop.  We come back here if the user wants to retry.
while

        echo >&2 'Please identify a location' \
                'so that time zone rules can be set correctly.'

        continent=
        country=
        region=


        # Ask the user for continent or ocean.

        echo >&2 'Please select a continent or ocean.'

        select continent in \
            Africa \
            Americas \
            Antarctica \
            'Arctic Ocean' \
            Asia \
            'Atlantic Ocean' \
            Australia \
            Europe \
            'Indian Ocean' \
            'Pacific Ocean' \
            'none - I want to specify the time zone using the Posix TZ format.'
        do
            case $continent in
            '')
                echo >&2 'Please enter a number in range.';;
            ?*)
                case $continent in
                Americas) continent=America;;
                *' '*) continent=$(expr "$continent" : '\([^ ]*\)')
                esac
                break
            esac
        done
```

根据大洲或者大洋，通过 awk 汇总对应的国家列表：

```
         *)
                # Get list of names of countries in the continent or ocean.
                countries=$($AWK -F'\t' \
                        -v continent="$continent" \
                        -v TZ_COUNTRY_TABLE="$TZ_COUNTRY_TABLE" \
                '
                        /^#/ { next }
                        $3 ~ ("^" continent "/") {
                                if (!cc_seen[$1]++) cc_list[++ccs] = $1
                        }
                        END {
                                while (getline <TZ_COUNTRY_TABLE) {
                                        if ($0 !~ /^#/) cc_name[$1] = $2
                                }
                                for (i = 1; i <= ccs; i++) {
                                        country = cc_list[i]
                                        if (cc_name[country]) {
                                          country = cc_name[country]
                                        }
                                        print country
                                }
                        }
                ' <$TZ_ZONE_TABLE | sort -f)
```

二级 select，选择国家：

```
170
171                 # If there's more than one country, ask the user which one.
172                 case $countries in
173                 *"$newline"*)
174                         echo >&2 'Please select a country.'
175                         select country in $countries
176                         do
177                             case $country in
178                             '') echo >&2 'Please enter a number in range.';;
179                             ?*) break
180                             esac
181                         done
182
183                         case $country in
184                         '') exit 1
185                         esac;;
186                 *)
187                         country=$countries
188                 esac
189
190
```

再次祭出 awk，通过国家汇总时区列表：

```
191                   # Get list of names of time zone rule regions in the country.
192                   regions=$($AWK -F'\t' \
193                           -v country="$country" \
194                           -v TZ_COUNTRY_TABLE="$TZ_COUNTRY_TABLE" \
195                   '
196                           BEGIN {
197                                   cc = country
198                                   while (getline <TZ_COUNTRY_TABLE) {
199                                           if ($0 !~ /^#/  &&  country == $2) {
200                                                   cc = $1
201                                                   break
202                                           }
203                                   }
204                           }
205                           $1 == cc { print $4 }
206                   ' <$TZ_ZONE_TABLE)
207
```

第三重 select，选择时区：

```
208
209                 # If there's more than one region, ask the user which one.
210                 case $regions in
211                 *"$newline"*)
212                         echo >&2 'Please select one of the following' \
213                                 'time zone regions.'
214                         select region in $regions
215                         do
216                                 case $region in
217                                 '') echo >&2 'Please enter a number in range.';;
218                                 ?*) break
219                                 esac
220                         done
221                         case $region in
222                         '') exit 1
223                         esac;;
224                 *)
225                         region=$regions
226                 esac
227
```

计算好时区之后，出现第四重 select，确认是否要修改：

```
275
276         # Output TZ info and ask the user to confirm.
277
278         echo >&2 ""
279         echo >&2 "The following information has been given:"
280         echo >&2 ""
281         case $country+$region in
282         ?*+?*)  echo >&2 "      $country$newline        $region";;
283         ?*+*)   echo >&2 "      $country";;
284         +)      echo >&2 "      TZ='$TZ'"
285         esac
286         echo >&2 ""
287         echo >&2 "Therefore TZ='$TZ' will be used.$extra_info"
288         echo >&2 "Is the above information OK?"
289
290         ok=
291         select ok in Yes No
292         do
293             case $ok in
294             '') echo >&2 'Please enter 1 for Yes, or 2 for No.';;
295             ?*) break
296             esac
297         done
298         case $ok in
299         '') exit 1;;
300         Yes) break
301         esac
302 do :
303 done
304
```

还要判断一下当前是 cshell 还是其他 shell，指导用户将当前的时区改到 shell 的启动脚本里面去（老大你写了这么多代码，不能自动把这句加进去么？还是要手动加☹）。

```
305 case $SHELL in
306 *csh) file=.login line="setenv TZ '$TZ'";;
307 *) file=.profile line="TZ='$TZ'; export TZ"
308 esac
309
310 echo >&2 "
311 You can make this change permanent for yourself by appending the line
312         $line
313 to the file '$file' in your home directory; then log out and log in again.
314
315 Here is that TZ value again, this time on standard output so that you
316 can use the $0 command in shell scripts:"
317
318 echo "$TZ"
```

3.11 ARITH_COM

```
00300: #if defined (DPAREN_ARITHMETIC)
00301: /* The arithmetic evaluation command, ((...)).  Just a set of flags and
00302:    a WORD_LIST, of which the first element is the only one used, for the
00303:    time being. */
00304: typedef struct arith_com {
00305:   int flags;
00306:   int line;
00307:   WORD_LIST *exp;
00308: } ARITH_COM;
00309: #endif /* DPAREN_ARITHMETIC */
```

ARITH_COM 也是需要开关打开的，不过当前默认用的 bash 都是支持的，这个命令的意思是算术表达式，算术表达式要用(())包起来，要不然 bash 会不知道你想当做算术表达式使用，如果执行“echo 1+1”会怎么样？bash 认为它是一个文本，直接将文本本身显示出来了☺。

```
-bash-4.2# echo 1+1
1+1
```

加上(())之后 echo 还是失败的：

```
-bash-4.2# echo ((1+1))
-bash: syntax error near unexpected token `('
```

再加一个$之后终于正常了：

```
-bash-4.2# echo $((1+1))
2
```

实际操作的时候，发现[]包起来的算术表达式也能用，但不加$的时候不会报错，加了会触发求值：

```
-bash-4.2# echo [1+1]
[1+1]
-bash-4.2# echo $[1+1]
2
```

通过查看代码，发现，只有(())是算术表达式：

```
03564: #if defined (DPAREN_ARITHMETIC)
03565: static int
03566: execute_arith_command (arith_command)
03567:      ARITH_COM *arith_command;
03568: {
03569:   int expok, save_line_number, retval;
03570:   intmax_t expresult;
03571:   WORD_LIST *new;
03572:   char *exp;
03573:
03574:   expresult = 0;
03575:
03576:   save_line_number = line_number;
03577:   this_command_name = "(("; /* )) */
03578:   line_number = arith_command->line;
03579:   /* If we're in a function, update the line number information. */
03580:   if (variable_context && interactive_shell)
03581:     {
03582:       line_number -= function_line_number;
03583:       if (line_number < 0)
03584:     line_number = 0;
03585:     }
03586:
```

而[]只是为了兼容 posix.2d9 的一种算术替换规则，也就是说$[1+1]是直接替换成了 2，而(())还需要走到算术表达式求值过程（是不是有点饶☺）。

```
08860:     /* Do POSIX.2d9-style arithmetic substitution.  This will probably go
08861:        away in a future bash release. */
08862:     case '[':
08863:       /* Extract the contents of this arithmetic substitution. */
08864:       t_index = zindex + 1;
08865:       temp = extract_arithmetic_subst (string, &t_index);
08866:       zindex = t_index;
08867:       if (temp == 0)
08868:     {
08869:       temp = savestring (string);
08870:       if (expanded_something)
08871:         *expanded_something = 0;
08872:       goto return0;
08873:     }
```

3.12 COND_COM

```
00320: typedef struct cond_com {
00321:   int flags;
00322:   int line;
00323:   int type;
00324:   WORD_DESC *op;
00325:   struct cond_com *left, *right;
00326: } COND_COM;
```

COND_COM 由六个属性组成，通用的 flags 和 line。Type 有 6 种，分别是与、或、一元、二元、最小单元、表达式。这样分类起始让人有点疑惑，其实所有表达式只有一元、二元、三元等等参数个数的区分，bash 源代码为了 yacc 解析方便，把其中的与表达式、或表达式、不带任何表达式的变量或者常量和其他表达式区分开来识别。

```
00311: /* The conditional command, [[...]].  This is a binary tree -- we slipped
00312:    a recursive-descent parser into the YACC grammar to parse it. */
00313: #define COND_AND    1
00314: #define COND_OR     2
00315: #define COND_UNARY  3
00316: #define COND_BINARY 4
00317: #define COND_TERM   5
00318: #define COND_EXPR   6
```

条件命令使用的一元表达式有 26 个，区分大小写（真的很多，居然只用了一半，没有把 26*2=52 个字母全部用光，设计这个的老大还真是拼啊），下面简单过一下。

a/e 判断文件是否存在；r/w/x 判断文件是否可读或者可写或者可执行；o 表示当前用户是否拥有该文件；G 表示当前用户的组是否拥有该文件；N 表示文件存在而且有新内容（从你上次读，文件被修改过）；f 表示常规文件（设备文件返回 false）。

```
00525:     case 'a':           /* file exists in the file system? */
00526:     case 'e':
00527:       return (sh_stat (arg, &stat_buf) == 0);
00528:
00529:     case 'r':           /* file is readable? */
00530:       return (sh_eaccess (arg, R_OK) == 0);
00531:
00532:     case 'w':           /* File is writeable? */
00533:       return (sh_eaccess (arg, W_OK) == 0);
00534:
00535:     case 'x':           /* File is executable? */
00536:       return (sh_eaccess (arg, X_OK) == 0);
00537:
00538:     case 'O':           /* File is owned by you? */
00539:       return (sh_stat (arg, &stat_buf) == 0 &&
00540:           (uid_t) current_user.euid == (uid_t) stat_buf.st_uid);
00541:
00542:     case 'G':           /* File is owned by your group? */
00543:       return (sh_stat (arg, &stat_buf) == 0 &&
00544:           (gid_t) current_user.egid == (gid_t) stat_buf.st_gid);
00545:
00546:     case 'N':
00547:       return (sh_stat (arg, &stat_buf) == 0 &&
00548:           stat_buf.st_atime <= stat_buf.st_mtime);
00549:
00550:     case 'f':           /* File is a file? */
00551:       if (sh_stat (arg, &stat_buf) < 0)
00552:     return (FALSE);
```

d 表示目录，s 表示文件大小大于 0，S 表示 socket，c 表示是字符设备，b 表示块设备，p

表示有名管道，L/h 表示符号链接：

```
00561:     case 'd':           /* File is a directory? */
00562:       return (sh_stat (arg, &stat_buf) == 0 && (S_ISDIR (stat_buf.st_mode)));
00563:
00564:     case 's':           /* File has something in it? */
00565:       return (sh_stat (arg, &stat_buf) == 0 && stat_buf.st_size > (off_t) 0);
00566:
00567:     case 'S':           /* File is a socket? */
00568: #if !defined (S_ISSOCK)
00569:       return (FALSE);
00570: #else
00571:       return (sh_stat (arg, &stat_buf) == 0 && S_ISSOCK (stat_buf.st_mode));
00572: #endif /* S_ISSOCK */
00573:
00574:     case 'c':           /* File is character special? */
00575:       return (sh_stat (arg, &stat_buf) == 0 && S_ISCHR (stat_buf.st_mode));
00576:
00577:     case 'b':           /* File is block special? */
00578:       return (sh_stat (arg, &stat_buf) == 0 && S_ISBLK (stat_buf.st_mode));
00579:
00580:     case 'p':           /* File is a named pipe? */
00581: #ifndef S_ISFIFO
00582:       return (FALSE);
00583: #else
00584:       return (sh_stat (arg, &stat_buf) == 0 && S_ISFIFO (stat_buf.st_mode));
00585: #endif /* S_ISFIFO */
00586:
00587:     case 'L':           /* Same as -h  */
00588:     case 'h':           /* File is a symbolic link? */
00589: #if !defined (S_ISLNK) || !defined (HAVE_LSTAT)
00590:       return (FALSE);
00591: #else
```

u 表示文件被设置了 uid，g 表示设置了组 id，k 表示 sticky 文件（对目录表示任何人都可以在该目录创建文件，但只能删除自己创建的文件，对可执行程序表示程序执行结束之后还会在内存待一阵子，方便下次再次执行的时候，不用从硬盘读到内存里面），t 表示终端。
n 表示脚本执行时存在至少一个参数，z 表示没有带任何参数，o 表示 arg 选项设置了，v 表示变量存在。最后还有一个 R，表示是否在命名表里面有索引。

```
00596:     case 'u':           /* File is setuid? */
00597:       return (sh_stat (arg, &stat_buf) == 0 && (stat_buf.st_mode & S_ISUID) != 0);
00598:
00599:     case 'g':           /* File is setgid? */
00600:       return (sh_stat (arg, &stat_buf) == 0 && (stat_buf.st_mode & S_ISGID) != 0);
00601:
00602:     case 'k':           /* File has sticky bit set? */
00603: #if !defined (S_ISVTX)
00604:       /* This is not Posix, and is not defined on some Posix systems. */
00605:       return (FALSE);
00606: #else
00607:       return (sh_stat (arg, &stat_buf) == 0 && (stat_buf.st_mode & S_ISVTX) != 0);
00608: #endif
00609:
00610:     case 't':   /* File fd is a terminal? */
00611:       if (legal_number (arg, &r) == 0)
00612:     return (FALSE);
00613:       return ((r == (int)r) && isatty ((int)r));
00614:
00615:     case 'n':           /* True if arg has some length. */
00616:       return (arg[0] != '\0');
00617:
00618:     case 'z':           /* True if arg has no length. */
00619:       return (arg[0] == '\0');
00620:
00621:     case 'o':           /* True if option `arg' is set. */
00622:       return (minus_o_option_value (arg) == 1);
00623:
00624:     case 'v':
00625:       v = find_variable (arg);
00626: #if defined (ARRAY_VARS)
00627:       if (v == 0 && valid_array_reference (arg, 0))
```

二元表达式比一元表达式少了不少，总共

=，>=，<=，>，<，!=，这六个望文生义就知道什么意思了。

另外四个其他语言也经常会见到：-nt（newer than，表示文件修改时间更新），-ot（older than，表示文件修改时间更老），-lt（less than，更小），-gt（greater than，更大）。

```
00395:   if (op[0] == '=' && (op[1] == '\0' || (op[1] == '=' && op[2] == '\0')))
00396:     return (patmatch ? patcomp (arg1, arg2, EQ) : STREQ (arg1, arg2));
00397:   else if ((op[0] == '>' || op[0] == '<') && op[1] == '\0')
00398:     {
00399: #if defined (HAVE_STRCOLL)
00400:       if (shell_compatibility_level > 40 && flags & TEST_LOCALE)
00401:     return ((op[0] == '>') ? (strcoll (arg1, arg2) > 0) : (strcoll (arg1, arg2) < 0));
00402:       else
00403: #endif
00404:     return ((op[0] == '>') ? (strcmp (arg1, arg2) > 0) : (strcmp (arg1, arg2) < 0));
00405:     }
00406:   else if (op[0] == '!' && op[1] == '=' && op[2] == '\0')
00407:     return (patmatch ? patcomp (arg1, arg2, NE) : (STREQ (arg1, arg2) == 0));
00408:
00409:
00410:   else if (op[2] == 't')
00411:     {
00412:       switch (op[1])
00413:     {
00414:     case 'n': return (filecomp (arg1, arg2, NT));       /* -nt */
00415:     case 'o': return (filecomp (arg1, arg2, OT));       /* -ot */
00416:     case 'l': return (arithcomp (arg1, arg2, LT, flags));   /* -lt */
00417:     case 'g': return (arithcomp (arg1, arg2, GT, flags));   /* -gt */
00418:     }
00419:     }
```

还有五个：-ef（equal file，同一文件，不只是文件内容完全一样，而且需要文件指向的 inode 节点也完全一样），-eq（equal，算术相等），-ne（not equal，算术不相等），-ge（greater or equal，算术大于等于），-le（算术小于等于）。上面的算术计算，对字符串也试用。

```
00420:   else if (op[1] == 'e')
00421:     {
00422:       switch (op[2])
00423:     {
00424:     case 'f': return (filecomp (arg1, arg2, EF));       /* -ef */
00425:     case 'q': return (arithcomp (arg1, arg2, EQ, flags));   /* -eq */
00426:     }
00427:     }
00428:   else if (op[2] == 'e')
00429:     {
00430:       switch (op[1])
00431:     {
00432:     case 'n': return (arithcomp (arg1, arg2, NE, flags));   /* -ne */
00433:     case 'g': return (arithcomp (arg1, arg2, GE, flags));   /* -ge */
00434:     case 'l': return (arithcomp (arg1, arg2, LE, flags));   /* -le */
00435:     }
```

3.13 ARITH_FOR_COM

```
00261: #if defined (ARITH_FOR_COMMAND)
00262: typedef struct arith_for_com {
00263:   int flags;
00264:   int line; /* generally used for error messages */
00265:   WORD_LIST *init;
00266:   WORD_LIST *test;
00267:   WORD_LIST *step;
00268:   COMMAND *action;
00269: } ARITH_FOR_COM;
00270: #endif
```

ARITH_FOR_COM 有点像 C 语言里面的 for 循环，几个属性中有初始化，测试边界命令，步进命令和实际的执行体。除了算术运算要求的两层小括号和 do，done 关键字，完整代码和 C 语言里面的 for 是不是没啥明显差别？☺

```
31  for ((i=0;i<3;i++))
32  do
33      echo $i
34  done
```

3.14 SUBSHELL_COM

```
00353: typedef struct subshell_com {
00354:   int flags;
00355:   COMMAND *command;
00356: } SUBSHELL_COM;
```

SUBSHELL_COM 的结构体定义比较简单，就是一个命令属性，看来关键的复杂度还是在写 shell 脚本本身上☺，不过对 bash 本身而言，就是把脚本文件读进来，后面一行一行执行的时候，和其他普通命令没有差别。

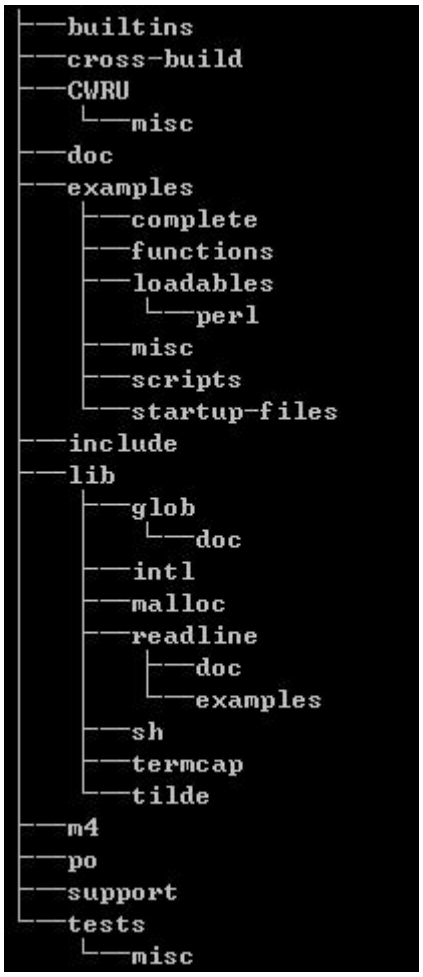
3.15 COPROC_COM

```
00373: typedef struct coproc_com {
00374:   int flags;
00375:   char *name;
00376:   COMMAND *command;
00377: } COPROC_COM;
```

COPROC_COM 的全称是 coprocess，翻译成中文应该是协程的意思，不过 shell 里面的协程没有像高级语言那么复杂，对 bash 而言，执行结果和执行命令后面加一个&类似，不过可以制定协程的名字。

4 bash 源代码的目录结构

现在才说源代码的目录结构是不是晚了点☺。
前面分析数据结构的时候，基本上把每个命令的执行过程也简单过了一下，这样大家读代码的时候先有一个概貌，不至于一叶障目不见泰山。
用 tree 命令打印出来的目录结构如下，其中 builtins 目录里面是大多数内置命令（例如 cd，pwd 等）的实现，但没有看到 ls 命令，难道部分复杂命令还另外建了 git 库来实现？
Cross-build 是交叉编译的设置。CWRU 是修改记录，很多人在修改记录里面还留了自己的邮箱，有兴趣的读者是否可以和他们聊聊☺。Doc 是文档目录，包括 html 格式的和 pdf 格式的文档。Examples 目录是各种脚本的例子，不知道怎么写复杂 bash 的读者有福了。Include 目录里面是一些实现 bash 过程中经查会用到的一些结构体或者宏的定义，一般都是写和 bash 不直接联系的，直接要用到的定义都在最顶级目录下面。
Lib 是实现 bash 种要用到的各种公共库，这些库的实现本身和 bash 的解析没有直接关系，统一放在 lib 目录。
M4 是 GUN 的一种编程语言，类似宏，m4 目录下面是用 m4 语言写的 timespec 结构体的定义相关头文件校验和时间统计相关的头文件校验宏。



Po 又是 GNU 的一种编程语言，字面意思是可扩展组件，bash 主要用它来实现多语种的扩展，每个语种都有自己的一个 po 文件，分别负责将代码里面的打印字符串转换成对应的语言。例如 zh_CN.po 里面定义了中文的各种字符串：

```
#: arrayfunc.c:54
msgid "bad array subscript"
msgstr "数组下标不正确"

#: arrayfunc.c:368 builtins/declare.def:574 variables.c:2092 variables.c:2118
#: variables.c:2730
#, c-format
msgid "%s: removing nameref attribute"
msgstr ""

#: arrayfunc.c:393 builtins/declare.def:780
#, c-format
msgid "%s: cannot convert indexed to associative array"
msgstr "%s: 无法将索引数组转化为关联数组"

#: arrayfunc.c:578
#, c-format
msgid "%s: invalid associative array key"
msgstr "%s: 无效的关联数组键"
```

Support 目录负责 bash 的手册 html 页面的生成。

Tests 是很多自测用例。

最重要的代码都放在最突出的位置，顶级目录下面有 44 个 c 文件，其中最重要的有四个：

shell.c，脚本的解析；make_cmd.c，命令生成；execute_cmd.c，命令执行；copy_cmd.c 命令拷贝。前面对语法的具体用法代码多数都来自 execute_cmd.c 文件。

y.tab.c	trap.c	sig.c	pcomplete.c	mailcheck.c	hashlib.c	expr.c	copy_cmd.c	assoc.c
version.c	test.c	shell.c	pathexp.c	locale.c	hashcmd.c	execute_cmd.c	braces.c	arrayfunc.c
xmalloc.c	subst.c	redir.c	nojobs.c	list.c	general.c	eval.c	bracecomp.c	array.c
variables.c	stringlib.c	print_cmd.c	mksyntax.c	jobs.c	flags.c	error.c	bashline.c	alias.c
unwind_prot.c	siglist.c	pcomplib.c	make_cmd.c	input.c	findcmd.c	dispose_cmd.c	bashhist.c	

redir.c 主要是各种重定向的定义和执行，前面讲到 SIMPLE_COM 的时候，曾经详细讲解过。subst.c 主要是对[]表达式的值替换。值得一提的是，这里对$开头的变量做了完整的说明。首先$0 到$9 分别对应脚本文件名，第一个入参，…，第九个入参。

```
08540:      /* $0 .. $9? */
08541:      case '0':
08542:      case '1':
08543:      case '2':
08544:      case '3':
08545:      case '4':
08546:      case '5':
08547:      case '6':
08548:      case '7':
08549:      case '8':
08550:      case '9':
08551:        temp1 = dollar_vars[TODIGIT (c)];
08552:        if (unbound_vars_is_error && temp1 == (char *)NULL)
08553:      {
08554:        uerror[0] = '$';
08555:        uerror[1] = c;
08556:        uerror[2] = '\0';
08557:        last_command_exit_value = EXECUTION_FAILURE;
```

$$执行 shell 的 pid；$#执行脚本时传入的参数总数；$?上一次同步命令的执行结果，同步命令的意思是这个命令不执行完，就无法继续下面的执行，如果命令执行的过程中加了&，那$?无法得到其执行结果；$-脚本执行时的 flags；$!，和$?相对，$!指的是上一个异步命令执行的结果。

```
08570:     /* $$ -- pid of the invoking shell. */
08571:     case '$':
08572:       temp = itos (dollar_dollar_pid);
08573:       break;
08574:
08575:     /* $# -- number of positional parameters. */
08576:     case '#':
08577:       temp = itos (number_of_args ());
08578:       break;
08579:
08580:     /* $? -- return value of the last synchronous command. */
08581:     case '?':
08582:       temp = itos (last_command_exit_value);
08583:       break;
08584:
08585:     /* $- -- flags supplied to the shell on invocation or by `set'. */
08586:     case '-':
08587:       temp = which_set_flags ();
08588:       break;
08589:
08590:       /* $! -- Pid of the last asynchronous command. */
08591:     case '!':
08592:       /* If no asynchronous pids have been created, expand to nothing.
08593:        If `set -u' has been executed, and no async processes have
08594:        been created, this is an expansion error. */
```

$*和$@都是打印出剩余所有参数，按代码的说法，$*和$@的差别就是是否将带引号的参数去掉引号，实测结果打印出来的结果似乎是一样的。

```
08614:     /* The only difference between this and $@ is when the arg is quoted. */
08615:     case '*':       /* `$*' */
08616:       list = list_rest_of_args ();
```

```
08697:     /* When we have "$@" what we want is "$1" "$2" "$3" ... This
08698:        means that we have to turn quoting off after we split into
08699:        the individually quoted arguments so that the final split
08700:        on the first character of $IFS is still done.  */
08701:     case '@':       /* `$@' */
08702:       list = list_rest_of_args ();
```

实测效果：

```
[root@wsbcx10 zrh]# cat -n test.sh
     1  #!/bin/bash
     2  echo "0:"
     3  echo $0
     4  echo "1:"
     5  echo $1
     6  echo $$
     7  echo $#
     8  echo $?
     9  echo $-
    10  echo $!
    11  echo $*
    12  echo "$*"
    13  echo $@
[root@wsbcx10 zrh]# bash test.sh 0 1 3 "testargs"
0:
test.sh
1:
0
14971
4
0
hB

0 1 3 testargs
0 1 3 testargs
0 1 3 testargs
[root@wsbcx10 zrh]#
```

5 bash 脚本的执行过程分析

一个环境上多个 sh 的 cpu 占用达到 99%，但实际通过 ps 看，这个 sh 并没有带任何参数，如果想要知道这个 sh 在干什么活，为何会一直冲高，还是 gdb 调试一下比较靠谱（还有一种可选的方法是不断的敲 cat /proc/*/stack 来反复查看堆栈，多敲敲之后总能抓到几次上下文，其中*换成对应进程的 pid）。

通过下面的调试，可以看到当前执行是一个简单命令(cm_simple)，通过 p *command->value->Simple->words->word 看到当前执行的简单命令在字符串是 true。

Breakpoint 1, execute_command (command=0x10946f0) at execute_cmd.c:386

386 result = execute_command_internal (command, 0, NO_PIPE, NO_PIPE, bitmap);
(gdb) p *command
$9 = {type = cm_simple, flags = 8, line = 0, redirects = 0x0, value = {For = 0x1094760, Case = 0x1094760, While = 0x1094760,
If = 0x1094760, Connection = 0x1094760, Simple = 0x1094760, Function_def = 0x1094760, Group = 0x1094760, Select = 0x1094760,
Arith = 0x1094760, Cond = 0x1094760, ArithFor = 0x1094760, Subshell = 0x1094760, Coproc = 0x1094760}}
(gdb) p *command->value->Simple
$10 = {flags = 8, line = 3, words = 0x1094740, redirects = 0x0}
(gdb) p *command->value->Simple->words->word
$11 = {word = 0x10946d0 "true", flags = 0}
(gdb) c
Continuing.
重新运行，可以看到又跑到了一个简单命令，其命令是"i=i+1"（汗）。
Breakpoint 1, execute_command (command=0x10947b0) at execute_cmd.c:386
386 result = execute_command_internal (command, 0, NO_PIPE, NO_PIPE, bitmap);
(gdb) p *command
$12 = {type = cm_simple, flags = 0, line = 0, redirects = 0x0, value = {For = 0x1094ad0, Case = 0x1094ad0, While = 0x1094ad0,
If = 0x1094ad0, Connection = 0x1094ad0, Simple = 0x1094ad0, Function_def = 0x1094ad0, Group = 0x1094ad0, Select = 0x1094ad0,
Arith = 0x1094ad0, Cond = 0x1094ad0, ArithFor = 0x1094ad0, Subshell = 0x1094ad0, Coproc = 0x1094ad0}}
(gdb) p *command->value->Simple->words->word
$13 = {word = 0x1094ab0 "i=i+1", flags = 20}
(gdb) c
Continuing.
通过 shell 的进程号，查询进程的上下文，发现是从另外一个虚机链接过来的 ssh，咨询环境负责人，该虚机是跑测试用例的，之前跑的测试用例不知道为何没有正常停止，测试用例确实就是简单的一行命令：

```
while :; do i=$((i+1));   done
```

cat /proc/27538/environ
XDG_SESSION_ID=24620SHELL=/bin/bashSSH_CLIENT=192.168.9.13 45494 22USER=rootPATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/binMAIL=/var/mail/rootPWD=/rootHOME=/rootSHLVL=2LOGNAME=rootSSH_CONNECTION=192.168.9.13 45494 192.168.9.196 22XDG_RUNTIME_DIR=/run/user/0_=/bin/sh-bash-4.2#

6 结束语

Bash 的源码解析到这里就结束了，更详细的内容，需要各位读者在实际使用时一一对应 bash 的源代码来得到更详细的解读，源码都来自于 GNU 的社区：https://git.savannah.gnu.org/git/bash.git，欢迎感兴趣的同事一起讨论分析。