# GNU bash 实现机制与源代码简析 - ruglcc's blog - CSDN 博客

# GNU bash 实现机制与源代码简析

目录

林健

摘要

本文是本人学习 shell 实现机理，分析 GNU bash 源代码时总结的笔记性文档。通过分析 bash 源代码，阐述了其主要功能模块的组织和实现方式，同时对几个特定的工作流程进行了说明。

# 第 1 章 概述

目录



## 1.1. bash

GNU bash 是各类 UNIX 系统，特别是 Linux 下经典的 shell。作为一个命令行解释器，它提供了强大的可编程功能，为用户提供了操作系统功能的良好接口。作为一个经典的开源项目，它的源代码结构较为清晰，可靠性、性能和易用性经历了考验。

本文分析的 bash 版本为 3.2.0(1)，源代码为 configure 之后的版本，因为部分源代码是在 configure 过程中由辅助工具生成的。builtins 目录下的 *.c 文件是 make 之后的版本（需要注释掉 builtins/Makefile 中删除 *.c 文件的语句），因为生成这些源代码的辅助工具需要在 make 过程中生成。

## 1.2. 环境与工具

项目 configure 和 make 环境为 Ubuntu 7.10（Linux 2.6.22），Intel Pentium III。

源代码分析工具为 Windows 下的 Source Insight 3.5。Source Insight 提供的交叉参考功能和调用链分析功能有助于理清复杂的函数调用和依赖关系。

# 第 2 章 程序结构分析

目录

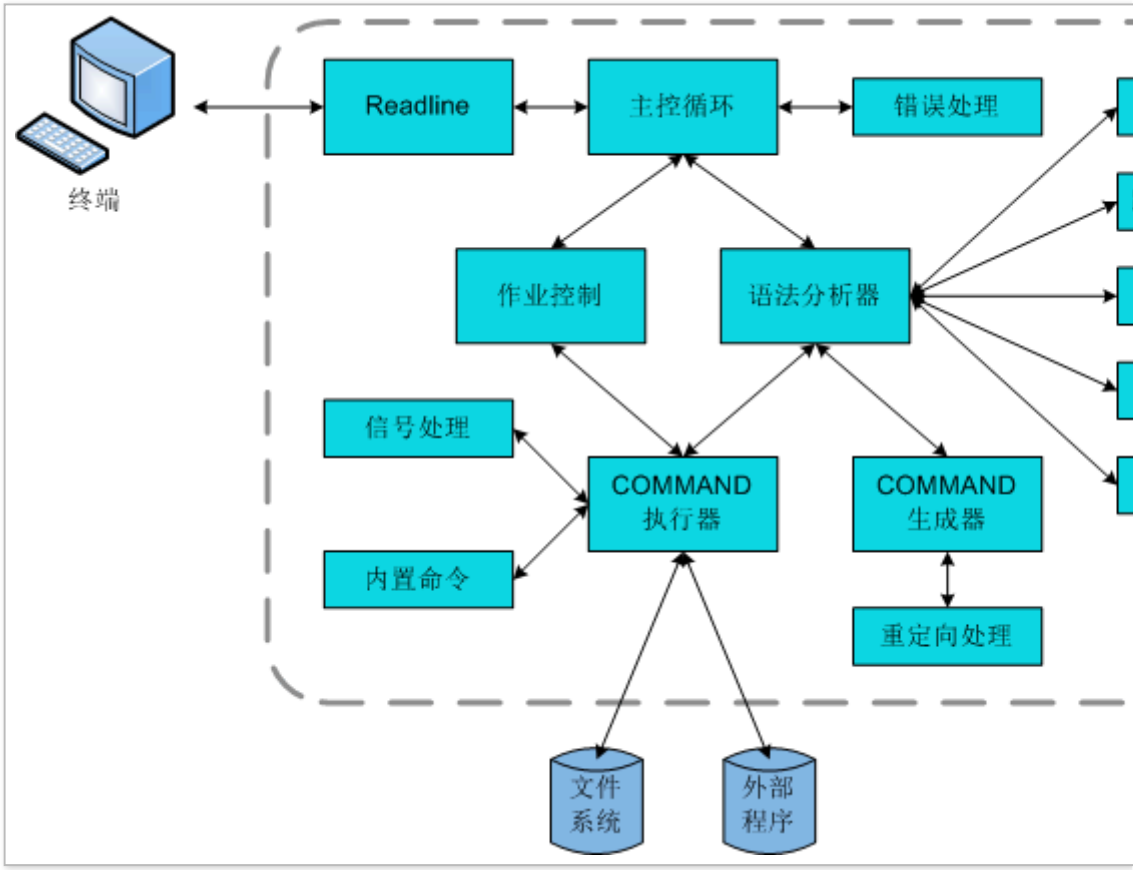


## 2.1. 系统架构

通常而言，shell 的功能是从终端或其它输入取得命令行，将其解析为一系列操作指令，调用系统内核或相应的外部程序执行，然后将执行结果返回给终端或其它输出。因此，实现一个简单的 shell 是一项容易的工作。但 bash 的功能不仅限于此，它支持用管道和重定向协同执行命令，提供了强大的脚本编程能力，具备作业管理功能。一般的 Linux 发行版中，bash 的可执行文件往往是 / bin 中最大的几个实用程序之一，客观反映了它的复杂性。

依据 bash 源代码的文件组织及函数调用关系，可分析出它的基本架构。

图 2.1. bash 基本架构图




bash 使用 GNU Readline 库处理用户命令输入，Readline 提供类似于 vi 或 emacs 的行编辑功能。

bash 运行时的调度中心是其主控循环。主控循环的功能较为简单，它循环读取用户（或脚本）输入，传递给语法分析器，同时处理下层递归返回的错误。

语法分析器对文本形式的输入首先进行通配符、别名、算术和变量展开等工作，然后通过命令生成器得到规范的命令结构，并由专门的重定向处理机制填写重定向语义，交由命令执行器执行。命令执行器依据命令种类不同，执行内部命令函数、外部程序或文件系统调用。在命令执行过程中，执行器要对系统信号进行捕获和处理。

在支持作业管理的操作系统中，命令执行器将进程信息加入作业控制机制，并允许用户使用内部命令或键盘信号来启停作业。如果在不支持作业管理的操作系统中编译 bash，会使用另一套接口相同的机制对进程信息进行简单的维护。

# 2.2. 主要数据结构

## 2.2.1. WORD_DESC 与 WORD_LIST

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

/* A linked list of words. */
typedef struct word_list {
  struct word_list *next;
  WORD_DESC *word;
} WORD_LIST;
```

命令等各种结构都可能引用字符串或字符串数组。bash 对于有必要做进一步语义处理的字符串使用 WORD_DESC、WORD_LIST 结构进行封装。

WORD_DESC 结构是对字符指针的一层封装，可称为字符串描述符。它在字符指针的基础上附加了一个 flags 标志位集合。其标志位包含 W_HASDOLLAR、W_QUOTED、W_DQUOTE、W_GLOBEXP 等，分别表示该字符串中包含了 “$”、引号、双引号、通配符等，目的是便于在变量代入、通配展开等过程中处理相应字符串。

WORD_LIST 结构是 WORD_DESC 对象的链表。

## 2.2.2. COMMAND

```
/* What a command looks like. */
typedef struct command {
  enum command_type type;       /* FOR CASE WHILE IF CONNE
  int flags;                    /* Flags controlling execu
  int line;                     /* line number the command
  REDIRECT *redirects;          /* Special redirects for F
  union {
    struct for_com *For;
    struct case_com *Case;
    struct while_com *While;
    struct if_com *If;
    struct connection *Connection;
    struct simple_com *Simple;
    struct function_def *Function_def;
```

```
    struct group_com *Group;
#if defined (SELECT_COMMAND)
    struct select_com *Select;
#endif
#if defined (DPAREN_ARITHMETIC)
    struct arith_com *Arith;
#endif
#if defined (COND_COMMAND)
    struct cond_com *Cond;
#endif
#if defined (ARITH_FOR_COMMAND)
    struct arith_for_com *ArithFor;
#endif
    struct subshell_com *Subshell;
  } value;
} COMMAND;
```

COMMAND 结构描述一条 bash 命令，这里的 “命令” 概念指语法分析器通过定界符、管道或控制语句分析出的相对独立的执行单元，它可以是内部或外部命令、函数、控制结构、算术表达式等。

枚举参数 type 表示了一个命令对象代表的命令属于上述何种类型。整型变量 flags 记录了一系列有关运行环境的标志位，包括：

```
/* Possible values for command->flags. */
#define CMD_WANT_SUBSHELL  0x01 /* User wants a subshell:
#define CMD_FORCE_SUBSHELL 0x02 /* Shell needs to force a
#define CMD_INVERT_RETURN  0x04 /* Invert the exit value.
#define CMD_IGNORE_RETURN  0x08 /* Ignore the exit value.
#define CMD_NO_FUNCTIONS   0x10 /* Ignore functions during
#define CMD_INHIBIT_EXPANSION 0x20 /* Do not expand the co
#define CMD_NO_FORK        0x40 /* Don't fork; just call e
#define CMD_TIME_PIPELINE  0x80 /* Time a pipeline */
#define CMD_TIME_POSIX     0x100 /* time -p; use POSIX.2 t
#define CMD_AMPERSAND      0x200 /* command & */
#define CMD_STDIN_REDIR    0x400 /* async command needs im
#define CMD_COMMAND_BUILTIN 0x0800 /* command executed by
```

整型变量 line 表示该命令在一个脚本中所处的行数。指针 redirects 指向说明本命令重定向信息的 REDIRECT 结构对象。最后，针对不同的命令类型，COMMAND 结构包含不同命令类型具体内部结构的联合：

对于内部或外部命令，使用联合中的 simple_com 结构，这个结构主要记录了命令名和命令行参数。它们存储于 WORD_LIST 链表结构，表元结点是 WORD_DESC 结构。

对于分支、循环等控制结构，它们的内部结构中主要包含指向相关控制流对应命令的指针。例如 while_com 结构包含了测试条件和循环体对应命令的指针；if_com 结构包含了测试条件、真值执行体和假值执行体对应命令的指针。

对于函数定义，function_def 结构包含了指向函数名的 WORD_DESC 结构指针、函数对应命令的指针以及指定函数所在文件名的指针。

## 2.2.3. REDIRECT 与 REDIRECTEE

```
/* Structure describing a redirection.  If REDIRECTOR is n
   (or translator in redir.c) encountered an out-of-range
typedef struct redirect {
  struct redirect *next;        /* Next element, or NULL.
  int redirector;               /* Descriptor to be redire
  int flags;                    /* Flag value for `open'.
  enum r_instruction  instruction; /* What to do with the
  REDIRECTEE redirectee;        /* File descriptor or file
  char *here_doc_eof;           /* The word that appeared
} REDIRECT;
```

REDIRECT 结构是对命令输入输出重定向的定义。一条命令可以设置多个（输入、输出、错误）重定向，因此 REDIRECT 结构本身包含指向下一个 REDIRECT 对象的 next 指针，以便构成一条命令的重定向链表。

整型参数 redirector 为重定向源的文件描述符，标志位集合 flags 定义目标文件打开方式。instruction 枚举定义了重定向的具体类型，它们包括：

```
/* Instructions describing what kind of thing to do for a
enum r_instruction {
  r_output_direction, r_input_direction, r_inputa_directic
  r_appending_to, r_reading_until, r_reading_string,
  r_duplicating_input, r_duplicating_output, r_deblank_rea
  r_close_this, r_err_and_out, r_input_output, r_output_fo
  r_duplicating_input_word, r_duplicating_output_word,
```

```
  r_move_input, r_move_output, r_move_input_word, r_move_o
};
```

重定向的目标记录在 REDIRECTEE 联合中，它可以是一个文件名或文件描述符。定义如下：

```
/* What a redirection descriptor looks like.  If the redir
   is ri_duplicating_input or ri_duplicating_output, use D
   use the file in FILENAME.  Out-of-range descriptors are
   negative DEST. */

typedef union {
  int dest;                     /* Place to redirect REDIR
  WORD_DESC *filename;          /* filename to redirect to
} REDIRECTEE;
```

此外，对于 Here Document 类型的重定向，REDIRECT 结构中的 here_doc_eof 指针指向 Here Document。

## 2.2.4. VAR_CONTEXT 与 SHELL_VAR

bash 本身的 shell 变量以及其中运行的函数的局部变量上下文存储在 VAR_CONTEXT 结构中。

```
/* A variable context. */
typedef struct var_context {
  char *name;           /* empty or NULL means global cont
  int scope;            /* 0 means global context */
  int flags;
  struct var_context *up;       /* previous function calls
  struct var_context *down;     /* down towards global con
  HASH_TABLE *table;            /* variables at this scope
} VAR_CONTEXT;
```

VAR_CONTEXT 的字符指针 name 如果为空则表示它存储的是 bash 全局上下文，否则表示某一个函数的局部上下文，name 指向函数的名称。整型变量 scope 是本上下文在栈中的层数，0 表示全局上下文，每深入一层函数调用 scope 递增 1，这样可以体现出该上下文的作用域。标志位集合 flags 记录该上下文是否为局部的、是否属于函数、是否属于内部命令，或者是不是临时建立的等信息。up 和 down 指针指向

函数调用栈中上一个和下一个局部上下文。哈希表 table 的内容是该上下文中的变量名值对。

bash 中的变量不强调类型，可以认为都是字符串。其存储结构如下：

```
typedef struct variable {
  char *name;                   /* Symbol that the user ty
  char *value;                  /* Value that is returned.
  char *exportstr;              /* String for the environm
  sh_var_value_func_t *dynamic_value;   /* Function called
                                   value for a variable, l
                                   or $RANDOM. */
  sh_var_assign_func_t *assign_func; /* Function called wh
                                   variable' is assigned a
                                   bind_variable. */
  int attributes;               /* export, readonly, array
  int context;                  /* Which context this vari
} SHELL_VAR;
```

字符指针 name 和 value 分别指向上下文变量的名和值字符串。对于导出（export）环境变量，exportstr 指向一个形如“名 = 值” 的字符串。对于返回一个动态变化值的变量（如 RANDOM），函数指针 dynamic_value 指向生成该值的函数。对于特定的变量，在被赋值的时候可以设置一个回调函数，其指针是 assign_func。整型变量 attributes 记录该上下文变量的可访问性，比如是否为导出的、只读的或隐藏的等。整型变量 context 记录该上下文变量属于可访问的作用域内局部变量栈的哪一层。

# 第 3 章 主要文件分析

目录



## 3.1. 根目录

- shell.c/shell.h

shell.c 是 main() 函数的所在，它定义了 shell 启动和运行过程中的一些状态量，依据不同的启动参数、环境变量等来初始化 shell 的工作状态（包括受限模式等），之后进入 eval.c 中的交互循环函数 reader_loop() 解析命令直到退出。
初始化函数 shell_initialize() 调用了 variables.c 中的 initialize_shell_variables()、set.c 中的 initialize_shell_options() 等一系列子模块初始化函数。如果要新增功能模块，可以将它们的初始化调用放在这里。
run_startup_files() 函数执行~/.profile、~/.bash_profile、~/.bash_login 等配置文件，同时判断了 bash 是否是由 sshd 或 rshd 启动的。对于 login shell，不执行~/.bashrc；对于 non-login 交互式 shell，或通过 sshd、rshd 启动的 shell，执行 ~/.bashrc。
run_one_command() 函数处理了 - c 参数运行一条命令的模式。
open_shell_script() 函数处理运行脚本文件的模式。
退出函数 exit_shell() 处理了挂起作业、保存历史等善后工作。

- eval.c
  读取并解释执行 shell 命令。主循环为 reader_loop() 函数，它调用 read_command()，read_command() 调用 parse_command()，parse_command() 调用语法分析器 y.tab.c 中的 yyparse()。得到命令后，reader_loop() 调用 execute_cmd.c 中的 execute_command() 执行命令。
  注：token 查找优先顺序：别名 > 关键字 > 函数 > 内部命令 > 脚本或可执行程序。

- execute_cmd.c/execute_cmd.h
  执行命令（COMMAND 结构）。外部调用接口是 execute_command()，内部通过 execute_command_internal() 执行命令。

execute_command_internal() 包含可选的管道重定向以及后台运行的参数。
针对不同类型的命令（控制结构、函数、算术等），execute_command_internal() 调用不同的函数来完成相应功能。其中 execute_builtin() 执行内部命令；execute_disk_command() 执行外部文件。
execute_disk_command() 通过调用 jobs.c 或 nojobs.c 中的 make_child() 来 fork 新进程执行。
本文件中维护了一个文件描述符的位图。

- make_cmd.c/make_cmd.h
  构造各类命令、重定向等语法结构实例所需的函数。由语法分析器、redir.c 等调用。
  其中 make_redirection() 填写命令结构的重定向参数。

- copy_command.c
  用来递归复制各种 COMMAND 结构的一系列函数。
  作者注释称：This is needed primarily for making function definitions, but I'm not sure that anyone else will need it.

- dispose_command.c/dispose_command.h
  清理 COMMAND 结构占用的资源，dispose_redirects() 清理重定向语句。

- print_cmd.c
  将命令结构转化为可打印的字符串。在 execute_cmd.c 的 execute_command_internal() 中有调用。

- redir.c/redir.h
  实现输入输出重定向。在执行之前，命令结构的 redirects 参数已由 make_cmd.c 填好，外部主要由 execute_cmd.c 调用以执行重定向操作。
  外部调用接口是 do_redirections()，它解析命令结构的重定向参数，内部交由 do_redirection_internal() 执行。接着依据不同的重定向类型，redir_open() 分别

使用常规的 open()、redir_special_open()、noclobber_open() 等函数打开重定向的文件描述符。
如果要添加新的重定向方式（如重定向到 FTP），可考虑在这里添加代码。
重定向原理参见 参考文献 《Unix/Linux 编程实践教程》10.3 节。

- paser.y
  yacc 语法定义文件。

- y.tab.c/y.tab.h
  yacc 生成的语法分析器。
  解析 token，调用 make_cmd.c 中的函数，生成命令结构，便于 execute_cmd.c 中的函数执行。
  其中包括调用 make_redirection() 填写命令结构的重定向参数。

- alias.c/alias.h
  别名操作相关函数，包括增、删、查、改等。内部命令 alias 的实现是调用本文件中的函数。

- array.c/array.h/arrayfunc.c/arrayfunc.c
  字符串数组定义及相关函数，实现了数组的一些高级操作。bash 程序中一些字符串数组使用了这里定义的 ARRAY 结构。

- bashansi.h
  针对不同的编译器处理一些系统头文件的包含关系。

- bashhist.c/bashhist.h
  命令历史记录功能相关的函数，包括历史记录的启、停、增、查等。

- bashintl.h
  引入 locate.h、gettext.h 等国际化支持。

- bashjmp.h
  对 setjmp.h 的一层封装，定义了 longjmp() 的几种状态参数。

- bashline.c/bashline.h
  与 readline 库的接口，解决命令自动补全、类 emacs 与 vi 行编辑功能等。

- bashtypes.h
  定义了 word 类型。

- bracecomp.c/braces.c
  使用大括号通配文件名功能的函数。

- builtins.h
  定义内部命令的基本结构 struct builtin。

- command.h
  各类命令（控制结构、函数、算术、重定向等）的结构定义。

- config.h/config-top.h/config-bot.h
  config.h 由 configure 生成，决定有哪些特性要被编译进 bash。如果要新增功能，可以加一个 “开关” 宏定义。

- conftypes.h
  定义主机体系结构和操作系统类型的名称。

- error.c/error.h
  错误处理与报告函数。

- expr.c
  处理算术表达式。外部调用接口是 evalexp()。

- externs.h
  声明一些源文件中的函数，它们在自己的头文件中没有声明。

- findcmd.c/findcmd.h
  通过名字查找命令。主要是从 PATH 变量位置查找外部可执行程序。

- flags.c/flags.h

存储和处理各个运行状态标志，如 Standard Sh Flags 与 Non-Standard Flags。

- general.c/general.h
  很多文件可能公用的一些基础的、不便分类的函数。

- hashcmd.c/hashcmd.h
  管理哈希表，主要用于将命令名字映射到完整路径。

- hashlib.c/hashlib.h
  哈希表的数据结构。

- input.c/input.h
  处理输入流缓冲。

- jobs.c/jobs.h
  作业控制。主要入口是 make_child()，用来创建进程并执行。
  jobs、fg、bg、kill 等命令的内部实现都在这里。
  作业管理详细流程暂未分析。

- nojobs.c
  在未实现作业控制的操作系统中代替 jobs.c 编译。

- list.c
  链表数据结构。

- locale.c
  与国际化相关的函数，包括对 “LC_” 系列环境变量的操作。

- lsignames.h/signames.h
  定义了各种信号的名称字符串。signames.h 是由编译时辅助工具 mksignames 生成的，mksignames 的源代码在 support 子目录。

- mailcheck.c/mailcheck.h
  检查账户邮箱的函数。

- mksyntax.c

用来生成编译时辅助工具 mksyntax。mksyntax 与用来构建词法分析文件 syntax.c。

- syntax.c/syntax.h
  syntax.c 是由 mksyntax 生成的词法分析文件，syntax.h 定义了词法分析工作中需要的宏和标志位等。

- parser.h
  parse.y 和 bashhist.c 所需的定界符栈结构（struct dstack）的定义。

- patchlevel.h
  记录 bash 的修正版本号。

- pathexp.c/pathexp.h
  与通配（globbing）库的接口。

- pathnames.h
  记录一些操作系统配置文件的路径。

- pcomplete.c/pcomplete.h/pcomplib.c
  可编程的命令补全功能。

- quit.h
  定义通用的异常退出宏，是对 SIGINT 信号的响应。

- sig.c/sig.h/siglist.c/siglist.h
  信号处理相关函数。

- stringlib.c
  字符串处理相关函数。包括从字符串 - 整数键值对结构（ALIST）中查找数据项等函数。

- subst.c/subst.h
  负责参数、命令、算术、路径扩展、引号等的代入、展开工作。

- test.c/test.h

GNU test program，各类条件比较条令，在 shell 脚本中常用。

- trap.c
  操作 trap 命令所需的一些对象的函数。

- unwind_prot.c/unwind_prot.h
  通用的函数执行保护和退出处理机制。

- variables.c/variables.h
  处理 shell 变量。用不同的哈希表分别存储不同生命周期的 shell 变量与函数。
  变量列表是由当前环境来初始化的。bash 启动时环境由 main() 的 char **env 参数传入。
  对于函数，使用栈来保存和切换局部变量的上下文。

- version.c/version.h
  显示 bash 版本号。

- xmalloc.c/xmalloc.h
  安全版本的 malloc 封装。

## 3.2. 其它目录

- builtins
  该目录下是内部命令的源代码。
  每个内部命令是一个 def 文件，Makefile 中 DEFSRC 声明了所有内部命令的 def 文件。
  由 mkbuiltins.c 生成编译时辅助工具 mkbuiltins，mkbuiltins 处理 *.def 文件，生成命令的 *.c 源程序以及 builtins.c、builtext.h。builtins.c 和 builtext.h 相当于各个内部命令的索引。
  所有文件最后编译得到 libbuiltins.a。
  此例 试验加入一个了内部命令。

- cross-build
  此目录下的文件是用于为其它系统交叉编译而缓存的 configure 结果。

- CWRU
  杂项文件，可能是来自 CWRU，暂未分析。

- doc
  Te 文档。

- examples
  脚本编程示例（可用于对扩展后的 bash 的验证）。

- include/lib
  bash 所需头文件、库文件（源代码）。

- po
  用于国际化的语言定义文件。

- support
  编译过程所需的支持工具及其源代码。

- tests
  make tests 所用测试脚本，可用于对扩展后的 bash 的验证。

# 第 4 章 主要流程分析

目录



## 4.1. 命令解析与执行

命令解析与执行的外部视图见 参考文献 中《详解 Bash 命令行处理》一文。

bash 启动并初始化完成后，进入 eval.c 中的交互循环函数 reader_loop() 开始解析命令。reader_loop() 不断循环读取和执行命令，直到遇到 EOF。

reader_loop() 中读取命令调用的是 read_command() 函数，read_command() 调用 parse_command()，parse_command() 调用语法分析器 y.tab.c 中的 yyparse()，最终取到命令。read_command() 将读到的命令存入了全局变量 global_command。其中：

read_command()的额外工作是执行 “shell 空闲一段时间后自动登出” 功能（环境变量 TMOUT）。

parse_command() 的额外工作是执行 PROMPT_COMMAND 指定的命令，调用处理 here document 的函数。

yyparse() 由 yacc 通过 parse.y 生成。它分析出命令语法后，调用 make_cmd.c 中的各种函数生成不同的 COMMAND 结构对象，用以执行。

读到命令后，reader_loop() 调用 execute_cmd.c 中的 execute_command() 执行命令。针对不同类型的命令（控制结构、函数、算术、重定向等），execute_command_internal() 调用不同的函数来完成相应功能。其中 execute_builtin() 执行内部命令；execute_disk_command() 执行外部文件。execute_disk_command() 通过调用 jobs.c 或 nojobs.c 中的 make_child() 来 fork 新进程执行。

make_child() 同步了输入流缓冲区，然后 fork 新进程。对于 jobs.c 版的 make_child()，对作业做一些初始化工作，再将待执行的命令通过 add_process() 函数加入启动进程链表。

从 make_child() 返回后，execute_disk_command() 判断 pid，如果是子进程，就调用 shell_execve() 函数在该函数中执行（exec）目标命令，同时做一些错误处理。

源程序对 execute_disk_command() 的注释如下：

```
/* Execute a simple command that is hopefully defined in a
   somewhere.
   1) fork ()
   2) connect pipes
   3) look up the command
   4) do redirections
   5) execve ()
   6) If the execve failed, see if the file has executable
   If so, and it isn't a directory, then execute its conte
   a shell script.
   Note that the filename hashing stuff has to take place
   in the parent.  This is probably why the Bourne style s
   don't handle it, since that would require them to go th
   this gnarly hair, for no good reason.
   NOTE: callers expect this to fork or exit(). */
```

## 4.2. 重定向的实现

COMMAND 结构有一个 REDIRECT 类型的指针（redirects），指向了本命令的重定向信息。

REDIRECT 结构记录了重定向的源描述符和目标：redirectee（类型为 REDIRECTEE），REDIRECTEE 是一个联合类型，它可以是目标描述符或目标文件名。REDIRECT 本身包含指向下一个 REDIRECT 对象的指针，因此对于一个 COMMAND 对象，可以有一系列重定向信息构成的链表。

语法分析器在遇到重定向语法时，调用 make_cmd.c 中的 make_redirection() 函数填写 COMMAND 结构的 REDIRECT 参数，并设置表示重定向方式的标志位。

execute_cmd.c 中的函数执行命令时，调用 redir.c 中的 do_redirections() 实现重定向。对于重定向信息链表中的每个 REDIRECT 对象，分别交由 do_redirection_internal() 处理。

do_redirection_internal() 针对重定向方式的标志位，做一些特定的设置，然后调用 redir_open()。

redir_open() 对于不同的重定向目标，调用不同的函数完成文件描述符的打开操作。例如软驱和网络设备文件调用 redir_special_open()，对于 noclobber mode（禁止覆盖变量

模式）调用 redir_special_open()，一般情况下调用常规的 open()，打开系统最小未用的文件描述符，实现重定向。

## 4.3. 内部命令（built-in）的构建

源代码目录（记为 $(srcdir)）下的 builtins 目录存储的是各个内部命令的源代码预定义文件（*.def）。在 make 的过程中，由 mkbuiltins 工具将它们预编译为源程序（*.c），进而编译为目标文件（*.o）。mkbuiltins 工具是由同一目录下的 mkbuiltins.c 编译生成的，它在处理 *.def 文件的同时，还会生成 builtins.c 和 builtext.h 两个文件，用做 bash 主程序调用内部命令的接口以及各个内部命令的索引。

要添加一条新内部命令，只需参考原有命令的存在形式即可，步骤如下：

1、新建预定义文件：$(srcdir)/builtins/[命令名].def。可复制已有命令的预定义文件，修改其中的 $PRODUCES、$BUILTIN、$FUNCTION、$SHORT_DOC 等定义，使之与命令名相符。

2、在预定义文件中建立命令处理函数，原型参考已有命令的处理函数，函数名与 $FUNCTION 的定义一致。参数为 WORD_LIST *list，该结构的定义在 $(srcdir)/command.h 中。处理参数的具体方法同样可参考已有的命令（如 echo）的处理函数。

3、修改 $(srcdir)/builtins/Makefile.in，参照已有的命令，分别在 DEFSRC、OFILES 添加对 [命令名].def、[命令名].o 的定义；添加 [命令名].o 对 [命令名].def 以及其它头文件的依赖关系。

4、回到 $(srcdir) 下，对源代码进行 configure、make，如果一切顺利的话，此时生成的 bash 程序将包含新添加的内部命令。

例 4.1. 新建一条 “linjian” 命令

本例中添加的命令处理函数为：

```
int linjian_builtin (list)
     WORD_LIST *list;
{
  printf ("This is a built-in for test by Lin Jian.\n");
  if (list)
    printf("Parameter: %s\n", list->word->word);
  return (EXECUTION_SUCCESS);
}
```

编译后试验结果如下：

```
#在原版bash下工作:
lj@lj-laptop:~/bash-3.2$ ps
  PID TTY          TIME CMD
 6212 pts/2    00:00:00 bash
 9893 pts/2    00:00:00 ps
lj@lj-laptop:~/bash-3.2$ linjian
-bash: linjian: command not found
#进入修改后的bash:
lj@lj-laptop:~/bash-3.2$ ./bash
lj@lj-laptop:~/bash-3.2$ ps
  PID TTY          TIME CMD
 6212 pts/2    00:00:00 bash
 9904 pts/2    00:00:00 bash
 9922 pts/2    00:00:00 ps
lj@lj-laptop:~/bash-3.2$ linjian hello!
This is a built-in for test by Lin Jian.
Parameter: hello!
lj@lj-laptop:~/bash-3.2$ type linjian
linjian is a shell builtin
```

# 4.4. 环境变量与上下文

Linux 中每个进程都有自己的环境（main 函数的 char *env[] 参数指向），环境是由一组变量组成的，这些变量中存有进程可能需要引用的上下文信息。bash 将环境变量的复本保存在 variables.c 中名为 shell_variables 的全局 VAR_CONTEXT 结构中。要导出给子进程的变量由全局字符串指针 char **export_env 记录，形式是 “名 = 值” 字符串数组，也就是键入 export 命令看到的内容。

bash 启动后，调用 variables.c 中的 initialize_shell_variables() 函数，传入来自 main 函数的 env 参数，将 env 中的环境变量存入 shell_variables。对于 PATH、 IFS 、PS1 之类 bash 本身要使用的环境变量，如果 env 中尚无，则在此时建立。另外一些有关 bash 版本、命令历史、邮件检查等内部辅助功能的环境变量也在这里建立。

execute_cmd.c 中调用各类命令的函数在执行命令之前，首先调用 variables.c 中的 maybe_make_export_env() 函数，构建导出给子进程的环境，即 export_env。shell_execve() 执行外部命令时使用的是 exec 族中的 execve() 函数，因此可以将 export_env 传递给 bash 启动的子进程。

凡需要增改环境变量的地方，调用 variables.c 中的 bind_variable() 函数实现。例如在 cd 命令执行后需要重设 PWD。

## 4.5. 由 sshd 启动 bash 的过程

bash 启动时，shell.c 中的 run_startup_files() 通过查找 SSH_CLIENT、SSH2_CLIENT 环境变量是否存在，来判断自己是不是由 sshd 启动的，记录在变量 run_by_ssh 中。此外可以通过检查 stdin 的文件描述符是否被重定向为网络文件或套接字来判断 bash 是不是由 rshd 启动的。如果 bash 启动自 sshd 或 rshd，并且是顶层 shell（非子 shell），则执行 ~/.bashrc 脚本。

防止~/.bashrc 被多次执行的方法是只在 bash 是顶层 shell 时加载之。作者曾考虑在 initialize_shell_variables() 过程中设置 SSH_CLIENT、SSH2_CLIENT 环境变量为非导出的，来避免子 shell 知道自己的父 shell 是由 sshd 启动的，从而不执行~/.bashrc。但他最终放弃了这个方法。

除此以外，bash 对 ssh 没有其它特殊处理。

## 4.6. 子 shell

shell 启动的 shell 子进程称为子 shell。直接以文件名运行可执行文件时，bash 并不知道它调用的一个可执行是二进制文件还是脚本，只是在 exec 过程中交给系统内核处理。对于 shell 脚本，通常以 “#![shell 可执行文件名]” 开头，“#!”是一种 magic number。当内核通过 magic number 断定执行的是脚本时，就会调用一个新的指定的 shell 的实例来解释执行脚本，这个实例就是子 shell。父子 shell 是两个进程，所以各自的变量是独立的。除非父 shell 将自己的变量导出到环境中，否则子 shell 无法获得父 shell 中定义的变量。

bash 通过变量 SHLVL 记录自己是进程调用栈中哪一层的 shell，即 bash 被嵌套的深度。bash 启动时，调用 variables.c 中的 initialize_shell_level() 设置 SHLVL。系统 login 之后启动的 bash 的 SHLVL 为 1，每层 shell 启动的子 shell 的 SHLVL 在其环境中读到的 SHLVL 基础上加 1。

使用 source 命令（“.” 命令）执行脚本时，不开启子 shell。bash 内部的实现是将脚本文件内容读入一个缓冲区，然后执行语法分析，因此效果与直接从键盘输入脚本内容相同。

# 第 5 章 杂记

目录



## 5.1. bash 编程风格

bash 的 C 语言函数声明使用了 “__P” 宏 ，定义遵循 K&R 规范，因此可以使用旧的非 ANSI 的编译器编译。bash 源代码充分考虑了不同的 CPU 体系结构、不同的操作系统和不同的编译器的差异，使用宏和条件编译处理这类问题，增强可移植性的同时不增加代码复杂性。对于某些可选择编译的特性，bash 通过定义宏作为开关，这些宏可以在 configure 时由用户参数决定取值，不需要编译者显式修改源代码。

bash 的代码缩进风格并不是很统一，可能是有来自不同贡献者的代码。多数代码与常见的 Java 风格差异较大，有些地方

不借助 Source Insight 这样的工具很容易找不到头绪。bash 并不十分避讳 goto 的使用，一些使用 goto 的地方的可读性还是比较好的。

由于多数函数名称都是完整的动宾短语，所以并非每个函数都有注释，通过名称可以了解其大致功能。对于某些复杂的流程，函数内部有一些注释，不少注释具有讨论和建议的性质，对未来的贡献者有启发性。和大多数 GNU 程序一样，bash 中也有不少风趣的注释，体现出西方特色的幽默。

# 附录 A. 学习备注（Q&A）

- Q：__P 是什么?
  A：ANSI C 之前的旧编译器不支持函数原型定义。使用 “__P” 宏为 ANSI 和非 ANSI 的编译器提供一种可移植的方案。“__P”的实现通常如下：

```
# if defined(__STDC__) || defined(__GNUC__)
#   definec__P(x) x
# else
#   define __P(x) ()
# endif
```

- Q：IFS 是什么?
  shell 环境变量 IFS（Internal Field Separator）中可能存储的值是空格、TAB、换行符等，在 bash 中默认是 0x0a。它用来在语法分析、变量代入等过程中界定命令。

- Q：Here Document 是什么?
  一个 here document 就是一段带有特殊目的的代码段。它使用 I/O 重定向的形式将一个命令序列传递到一个交互程序或者命令中，比如 ftp、cat 或者 ex 文本编辑器。例如：

```
COMMAND <<InputComesFromHERE
...
InputComesFromHERE
```

# 附录 B. 参考文献

- Mendel Cooper．Advanced Bash-Scripting Guide，2006．
  http://personal.riverusers.com/~thegrendel/abs-guide.pdf
- Bruce Molay．Unix/Linux 编程实践教程，北京：清华大学出版社，2004．
- Eric S. Raymond．Unix 编程艺术，北京：电子工业出版社，2006．
- home_king．详解 Bash 命令行处理，2005．
  http://www.linuxsir.org/main/?q=node/134
- Chet Ramey．Bash FAQ version 3.36，2006．
  ftp://ftp.cwru.edu/pub/bash/FAQ


# 附录 C. 作者信息


林健，北京理工大学计算机科学技术学院学生。转载本文请注明出处。如发现文中不妥或错误之处，望不吝赐教。

作者网站： http://www.linjian.cn

作者 Blog： http://blog.linjian.cn


全文完