summary

shell.c: #IMP

```
/* Non-zero means that this shell has already been run; i.e. you should
   call shell_reinitialize () if you need to start afresh. */
int shell_initialized = 0;

COMMAND *global_command = (COMMAND *)NULL; //#IMP 获取命令的

/* Information about the current user. */
struct user_info current_user =
{
  (uid_t)-1, (uid_t)-1, (gid_t)-1, (gid_t)-1,
--
  this_command_name = shell_name;   /* for error reporting */
  arg_index = parse_shell_options (argv, arg_index, argc);

  /* If user supplied the "--login" (or -l) flag, then set and invert
     LOGIN_SHELL. */
 // #IMP 2019-05-31 以登录方式进入shell
  if (make_login_shell)
    {
      login_shell++;
      login_shell = -login_shell;
    }
--
    no -c command
    no arguments remaining or the -s flag given
    standard input is a terminal
    standard error is a terminal
     Refer to Posix.2, the description of the `sh' utility. */
 // #IMP 2019-05-31 交互shell

  if (forced_interactive ||     /* -i flag */
      (!command_execution_string && /* No -c command and ... */
       wordexp_only == 0 &&     /* No --wordexp and ... */
       ((arg_index == argc) ||      /*   no remaining args or... */
--
 read_and_execute:
#endif /* !ONESHOT */

  shell_initialized = 1;

// #IMP 2019-06-01 开始循环读取指令
  /* Read commands until exit condition. */
  reader_loop ();
```

```
  exit_shell (last_command_exit_value);
}

--
    maybe_execute_file (fn, 1);
      FREE (fn);
    }
}

// #IMP 2019-06-01 读取bash配置文件
// #IMP 执行命令前先读取配置文件
static void
run_startup_files ()
{
#if defined (JOB_CONTROL)
  int old_job_control;
--
    }
}

/* Do whatever is necessary to initialize the shell.
   Put new initializations in here. */
// #IMP ｓｈｅｌｌ初始化
static void
shell_initialize ()
{
  char hostname[256];
  int should_be_restricted;
```

shell.c: #NOTE

```
  Sunday, January 10th, 1988.
  Initial author: Brian Fox
*/
#define INSTALL_DEBUG_MODE

// #NOTE 2019-05-27 config.h是使用configure自动生成的文件，保存了电脑的基本信息
#include "config.h"

#include "bashtypes.h"

// #NOTE 2019-05-31 在没有定义MINIX文件系统但是定义了HAVE_SYS_FILE_H宏的时候引入系统中的文件数据结构
#if !defined (_MINIX) && defined (HAVE_SYS_FILE_H)
#  include <sys/file.h>
#endif

#include "posixstat.h"
--
#if defined (NO_MAIN_ENV_ARG)
extern char **environ;  /* used if no third argument to main() */
#endif
```

```
extern char *dist_version, *release_status;
// #NOTE 2019-05-27 extern引用不在此文件中的全局变量，编译器会自动去
// 其他C文件里面去找，比如这里的patch_level就在version.c里面
extern int patch_level, build_version;
extern int shell_level;
extern int subshell_environment;
extern int running_in_background;
--
  const char *name;
  int type;
  int *int_value;
  char **char_value;
} long_args[] = {
  // #NOTE 2019-05-31 使用预定义语句来确定long_args[]的内容
  { "debug", Int, &debugging, (char **)0x0 },
#if defined (DEBUGGER)
  { "debugger", Int, &debugging_mode, (char **)0x0 },
#endif
  { "dump-po-strings", Int, &dump_po_strings, (char **)0x0 },
--

static void add_shopt_to_alist __P((char *, int));
static void run_shopt_alist __P((void));

static void execute_env_file __P((char *));
// #NOTE 2019-06-01
static void run_startup_files __P((void));//判断是不是由sshd启动的bash
static int open_shell_script __P((char *));
static void set_bash_input __P((void));
static int run_one_command __P((char *));
#if defined (WORDEXP_OPTION)
--
     int argc;
     char **argv;
#else /* !NO_MAIN_ENV_ARG */
int
main (argc, argv, env)
     int argc; // #NOTE 2019-05-31 整型的argc是存放程序运行时发送个main函数的参数个数。
     /*
      * argv存放指向字符串参数的指针数组。
        argv[0]-----指向程序运行的全路径
        argv[1]-----指向执行程序名后的第一个字符串；argv[2]-----指向执行程序名后的第二个字符串…
        argv[argc]-------NULL
--
  register int i;
  int code, old_errexit_flag;
#if defined (RESTRICTED_SHELL)
  int saverst;
#endif
  // #NOTE 2019-05-31 volatile的变量是说这变量可能会被意想不到地改变，这样，编译器就不会去假设这个变量的值了。
```

```
  // volatile 一般用在多线程共享的变量上。volatile可以保证变量在并发修改过程中的可见性，是一种轻量级的同步机制
  volatile int locally_skip_execution;
  volatile int arg_index, top_level_arg_index;
#ifdef __OPENNT
  char **env;
--
  while (debugging_login_shell) sleep (3);

  set_default_locale ();

  running_setuid = uidget ();
// #NOTE 2019-05-31
//  bash的posix模式
//   当bash以posix模式启动时，就像使用--posix选项一样，它的启动文件遵循POSIX标准。
//   这种模式下，交互式shell展开ENV变量的并读取和执行以ENV变量值为文件名的配置文件。不会读取其他启动文件。
  if (getenv ("POSIXLY_CORRECT") || getenv ("POSIX_PEDANTIC"))//POSIXLY_CORRECT如果被设置，bash以POSIX模式启动
    posixly_correct = 1;
--
    }

  shell_reinitialized = 0;

  /* Initialize `local' variables for all `invocations' of main (). */
// #NOTE 2019-05-31  为main函数中调用的所有变量初始化本地变量
  arg_index = 1;
  if (arg_index > argc)
    arg_index = argc;
  command_execution_string = (char *)NULL;
  want_pending_command = locally_skip_execution = read_from_stdin = 0;
--
  /* If this shell has already been run, then reinitialize it to a
     vanilla state. */
  if (shell_initialized || shell_name)
    {
      /* Make sure that we do not infinitely recurse as a login shell. */
    // #NOTE 2019-05-31  保证不把递归作为登录shell
    // #NOTE 2019-05-31 保证不陷入登录shell的死循环
    // 即登录之后就不再打开登录shell了
      if (*shell_name == '-')
    shell_name++;

      shell_reinitialize ();
--
  /* Parse argument flags from the input line. */

  /* Find full word arguments first. */
  arg_index = parse_long_options (argv, arg_index, argc);

  if (want_initial_help)// #NOTE 2019-05-31 初始化的帮助
    {
      show_shell_usage (stdout, 1);// #NOTE 2019-05-31 输出shell基本用法
```

```
      exit (EXECUTION_SUCCESS);
    }

  if (do_version)//版本
    {
--
    }

  echo_input_at_read = verbose_flag;    /* --verbose given */

  /* All done with full word options; do standard shell option parsing.*/
 // #NOTE 2019-05-31 全部使用完整的选项完成；执行标准shell选项解析
  this_command_name = shell_name;   /* for error reporting */
  arg_index = parse_shell_options (argv, arg_index, argc);

  /* If user supplied the "--login" (or -l) flag, then set and invert
     LOGIN_SHELL. */
--
   * '22.1 (term:0.96)' instead of (or in addition to) setting INSIDE_EMACS.
   * They may set TERM to 'eterm' instead of 'eterm-color'.  They may have
   * a now-obsolete command that sets neither EMACS nor INSIDE_EMACS:
   * M-x terminal -> TERM='emacs-em7955' (line editing)
   */
  // #NOTE 2019-06-01 非交互式shell
  if (interactive_shell)
    {
      char *term, *emacs, *inside_emacs;;
      int emacs_term, in_emacs;

--
    {
      change_flag ('i', FLAG_ON);
      interactive = 1;
    }

// #NOTE 2019-06-01 打开严格模式，只要指令出错就退出
#if defined (RESTRICTED_SHELL)
  /* Set restricted_shell based on whether the basename of $0 indicates that
     the shell should be restricted or if the `-r' option was supplied at
     startup. */
  restricted_shell = shell_is_restricted (shell_name);
--
  cmd_init ();      /* initialize the command object caches */
  uwp_init ();

  if (command_execution_string)
    {
      // #NOTE 2019-06-01 参数绑定
      arg_index = bind_args (argv, arg_index, argc, 0);
      startup_state = 2;

// #NOTE 2019-06-01 打开调试
      if (debugging_mode)
```

```
    start_debugger ();

#if defined (ONESHOT)
      executing = 1;
--
  /* Read commands until exit condition. */
  reader_loop ();
  exit_shell (last_command_exit_value);
}

// #NOTE 2019-06-01 解析长选项，比如像--login这样的选项
static int
parse_long_options (argv, arg_start, arg_end)
     char **argv;
     int arg_start, arg_end;
{
--

  return (arg_index);
}

static int
// #NOTE 2019-05-31 解析shell选项
parse_shell_options (argv, arg_start, arg_end)//理解选择的shell操作
     char **argv;
     int arg_start, arg_end;
{
  int arg_index;
--
/* Exit the shell with status S. */
void
exit_shell (s)
     int s;
{
  // #NOTE 2019-05-31 刷新标准输出和标准错误
    /*
     * stdout 标准输出设备，对应终端屏幕。是一个定义在<stdio.h>的宏，它展开到一个 FILE*类型的表达式（不一定是常量），
       这个表达式指向一个与标准输出流（standard output stream）相关连的 FILE 对象。
     * stderr 标准错误输出设备 ，对应终端的屏幕。
       进程将从标准输入文件中得到输入数据，将正常输出数据输出到标准输出文件，而将错误信息送到标准错误文件中。
--
           (find_variable ("SSH2_CLIENT") != (SHELL_VAR *)0);
#else
      run_by_ssh = 0;
#endif

// #NOTE 2019-06-01 读取.bashrc配置文件
      /* If we were run by sshd or we think we were run by rshd, execute
     ~/.bashrc if we are a top-level shell. */
      if ((run_by_ssh || isnetconn (fileno (stdin))) && shell_level < 2)
    {
```

```
#ifdef SYS_BASHRC
--
#endif

  shell_reinitialized = 1;
}

// #NOTE 2019-06-01
/*
 * 展示给用户的shell用法的界面
 */
static void
show_shell_usage (fp, extra)
```

shell.c: #TODO

```
  /* Fix for the `infinite process creation' bug when running shell scripts
     from startup files on System V. */
  login_shell = make_login_shell = 0;

// #TODO 2019-06-01 什么是vanilla state
  /* If this shell has already been run, then reinitialize it to a
     vanilla state. */
  if (shell_initialized || shell_name)
    {
      /* Make sure that we do not infinitely recurse as a login shell. */
--
   * of the getpw* functions, and it's set to be open across execs.  That
   * means one for login, one for xterm, one for shelltool, etc.  There are
   * also systems that open persistent FDs to other agents or files as part
   * of process startup; these need to be set to be close-on-exec.
   */
  // #TODO 2019-05-31 这里是什么意思？
  if (login_shell && interactive_shell)
    {
      for (i = 3; i < 20; i++)
    SET_CLOSE_ON_EXEC (i);
    }
```

eval.c: #IMP

```
#endif

static void send_pwd_to_eterm __P((void));
static sighandler alrm_catcher __P((int));

// #IMP 2019-06-01 非常重要的循环读取指令的循环
/* 读取并执行命令，直到达到EOF。这假定输入源已初始化 */
```

```
//#IMP 主循环reader_loop()函数，调用read_command()
int reader_loop ()
{
  int our_indirection_level;
  COMMAND * volatile current_command;

--
                  fprintf (stderr, "%s", ps0_string);
                  fflush (stderr);
                }
              free (ps0_string);
            }
            //#IMP parse_command()函数里的yyparse将解析的命令保留在全局变量GLOBAL_COMMAND中
            //#IMP 这里得到命令后，reader_loop()调用execute_cmd.c中的execute_command()执行命令。
            execute_command (current_command);

          // #NOTE 2019-05-18 结束标号，指令执行完成
          exec_done:
          /*NOTE 宏调用很多。 SIGINT只设置interrupt_state变量。
--
        /* 解析错误，如果不是交互式的话，可能会丢弃其余的流。
        interactive == 0指非交互式*/
        if (interactive == 0)
          EOF_Reached = EOF;
      }
      /* #IMP命令行上的-t设置变量just_one_command。
      此变量仅在这一个地方使用：（eval.c中循环结束时的if条件中）
      如果给出了-t标志，则在执行第一个命令之后，发出文件结束条件的信号并退出bash。
      reader_loop中的循环继续，直到它收到信号EOF_Reached。
      -t选项的唯一作用是在循环结束时设置此标志，以确保循环仅执行一次。*/
      if (just_one_command)
--
  fprintf (stderr, "\032/%s\n", pwd);
  free (f);
}

// #NOTE 2019-06-01 解析bash指令语法
/* #IMP 调用YACC生成的解析器并返回解析的状态。
   从当前输入流（bash_input）读取输入。
   yyparse将解析的命令保留在全局变量GLOBAL_COMMAND中。
   这是执行PROMPT_COMMAND的地方。 */
//#IMP read_command()调用parse_command()
int parse_command ()
{
  int r;
  char *command_to_execute;//要执行的命令

--
    send_pwd_to_eterm ();   /* Yuck */
      // #TODO 2019-06-01 Yuck是呸的意思，这是什么鬼，程序员爆粗口？
    }

  current_command_line_count = 0;
```

```
  //#IMP parse_command()调用语法分析器y.tab.c中的yyparse()是语法分析的开始
  //#NOTE bash也是利用yacc生成的代码来完成语法分析的，y.tab.c的yyparse()是yacc自动生成的
  r = yyparse ();

  if (need_here_doc)
    gather_here_documents ();

  return (r);//得到命令
}

/*  #IMP read_command ()被主函数调用，读取并解析命令，返回解析的状态。
解析结果的命令字符串保留在全局变量GLOBAL_COMMAND中以供reader_loop使用。
这是执行shell超时代码的地方。*/
int read_command ()
{
  //#NOTE bash中的变量不强调类型，可以认为都是字符串。
--

  /* #NOTE 只有交互时才会超时 */
  tmout_var = (SHELL_VAR *)NULL;
  tmout_len = 0;
  old_alrm = (SigHandler *)NULL;
  //#IMP 非零意味着此时shell是交互式的。
  //#IMP 一般来说，这意味着shell正在读取输入从键盘。
  if (interactive)
    {
      //#NOTE linux设置的空闲等待时间TMOUT
      tmout_var = find_variable ("TMOUT");
      //#NOTE ((tmout_var)->value != 0)，已经连接
--
    }

  QUIT;

  current_command_line_count = 0;
  //#IMP 解析命令
  result = parse_command ();

  //#NOTE 此时shell是交互式的，shell正在读取输入从键盘
  if (interactive && tmout_var && (tmout_len > 0))
    {
--
      //在经过参数指定的秒数后传送给目前的进程.
      //如果参数为0, 则之前设置的闹钟会被取消, 并将剩下的时间返回.
      alarm(0);
      set_signal_handler (SIGALRM, old_alrm);
    }
  //#IMP 解析的命令
  return (result);
}
```

eval.c: #NOTE

```
#include "execute_cmd.h"

#if defined (HISTORY)
#  include "bashhist.h"
#endif
//#NOTE extern可以置于变量或者函数前
//#NOTE以表示变量或者函数的定义在别的文件中
//#NOTE提示编译器遇到此变量和函数时在其他模块中寻找其定义。
extern int EOF_reached;
extern int indirection_level;
extern int posixly_correct;
extern int subshell_environment, running_under_emacs;
extern int last_command_exit_value, stdin_redir;
--
// #TODO 2019-06-01 把它转换为void*的宏？
  USE_VAR(current_command);

  current_command = (COMMAND *)NULL;

  //#NOTE 命令的递归深度
  our_indirection_level = ++indirection_level;
// #NOTE 2019-06-01 没有到达EOF（end of file文件结尾）时一直读，就是没有读到指令结尾时就一直读
  //#NOTE 到达文件末尾时，这个全局变量非零。为零即当未到末尾时
  while (EOF_Reached == 0)
    {
      int code;

// #NOTE 2019-06-01 处理bash信号
      code = setjmp_nosigs (top_level);

#if defined (PROCESS_SUBSTITUTION)
      unlink_fifo_list ();
#endif /* PROCESS_SUBSTITUTION */

      /* #TODO  - 为什么每次都通过循环设置这个？为什么呢
如果SIGINT被困在交互式shell中？
 */
      //#NOTE interactive_shell非零意味着shell作为交互式shell启动
      if (interactive_shell && signal_is_ignored (SIGINT) == 0)
    set_signal_handler (SIGINT, sigint_sighandler);

      if (code != NOT_JUMPED)
    {
    //#NOTE命令的递归深度
      indirection_level = our_indirection_level;

// #NOTE 2019-06-01 根据信号来确定bash的操作
      switch (code)
      {
        // #NOTE 2019-06-01 退出shell
          /* Some kind of throw to top_level has occurred.
        -发生了某种抛向top_level。- */
        case FORCE_EOF://#NOTE停止解析，-1
```

```
        case ERREXIT://#NOTE由于错误情况退出，-4
        case EXITPROG://#NOTE现在无条件退出程序。-3
          current_command = (COMMAND *)NULL;
          if (exit_immediately_on_error)
            //#NOTE 当前的变量上下文。 这实际上是我们执行函数的深度计算。
                variable_context = 0;   /* 不是在一个功能 */
          EOF_Reached = EOF;//直到达到EOF
        // #NOTE 2019-06-01 跳转到exec_done的标号处
          goto exec_done;

        case DISCARD://#NOTE丢弃当前命令。-2
          /* 确保退出状态重置为非零值，但是
        仅保留现有的非零值（例如，信号> 128） */
        //#NOTE如果上一个同步命令返回的值==0。
          if (last_command_exit_value == 0)
                last_command_exit_value = EXECUTION_FAILURE;//execute_command（）返回的值
          //如果刚刚分叉并且当前正在子shell中运行（非零环境）。
        if (subshell_environment)
        {
          current_command = (COMMAND *)NULL;
          EOF_Reached = EOF;
          goto exec_done;
        }
    // #NOTE 2019-06-01 丢弃当前指令
          /* 不受阻碍的命令元素等 */
          if (current_command)
        {
          dispose_command (current_command);
          current_command = (COMMAND *)NULL;
--
          sigprocmask (SIG_SETMASK, &top_level_mask, (sigset_t *)NULL);
#endif
          break;

        default:
      // #NOTE 2019-06-01 打印错误信息
          command_error ("reader_loop", CMDERR_BADJUMP, code, 0);
        }
    }

      executing = 0;
      //#NOTE 一组shell分配，仅在单个命令的环境中进行。
      if (temporary_env)
            dispose_used_env_vars ();

#if (defined (ultrix) && defined (mips)) || defined (C_ALLOCA)
      /* 尝试回收使用alloca（）分配的内存。 */
      (void) alloca (0);
#endif

      if (read_command () == 0)
      {
```

```
        //#NOTE interactive_shell为零意味着非交互式，read_but_dont_execute非零表示读命令，但不执行
它们
        if (interactive_shell == 0 && read_but_dont_execute)
          {
            //last_command_exit_value是上一个同步命令返回的值。
            last_command_exit_value = EXECUTION_SUCCESS;//0
            dispose_command (global_command);
            global_command = (COMMAND *)NULL;
          }
        else if (current_command = global_command)
          {
            global_command = (COMMAND *)NULL;
            //#NOTE 到目前为止执行的命令数++
            current_command_number++;

            executing = 1;//#NOTE 当我们执行顶级命令时非零
            /*如果fd 0是重定向到子shell的主题，则设置为1。
            全局使reader_loop可以在执行命令之前将stdin_redir设置为零。 */
            stdin_redir = 0;

            /* 如果shell是交互式的，
            则在读取命令（可能是列表或管道）之后并在执行之前展开并显示$ PS0 */
     // #NOTE 2019-06-01 读取完指令后就马上展开提示(提示就是你指令前面的一些字符)
            if (interactive && ps0_prompt)
            {
              char *ps0_string;

              ps0_string = decode_prompt_string (ps0_prompt);
--
            }
            //#IMP parse_command()函数里的yyparse将解析的命令保留在全局变量GLOBAL_COMMAND中
            //#IMP 这里得到命令后，reader_loop()调用execute_cmd.c中的execute_command()执行命令。
            execute_command (current_command);

          // #NOTE 2019-05-18 结束标号，指令执行完成
          exec_done:
          /*NOTE 宏调用很多。 SIGINT只设置interrupt_state变量。
          当它是安全的时，将QUIT放在代码中，然后“中断”地点。
          相同的方案用于终止信号（例如，SIGHUP）和终止信号变量。
          这会调用一个最终会退出shell的函数。*/
            QUIT;

// #NOTE 2019-06-01 释放当前指令的空间
// 每条指令的长短不一样，无法复用指令空间，只能每次重新申请
            if (current_command)
            {
              dispose_command (current_command);
              current_command = (COMMAND *)NULL;
--
      if (just_one_command)
            EOF_Reached = EOF;
    }
  indirection_level--;//是命令的递归深度
```

```
  return (last_command_exit_value);//上一个同步命令返回的值。
}//#NOTE主函数结束

// #NOTE 2019-06-01 捕获警报
static sighandler
alrm_catcher(i)
     int i;
{
  printf (_("\007timed out waiting for input: auto-logout\n"));
--
    f = pwd = get_working_directory ("eterm");
  fprintf (stderr, "\032/%s\n", pwd);
  free (f);
}

// #NOTE 2019-06-01 解析bash指令语法
/* #IMP 调用YACC生成的解析器并返回解析的状态。
   从当前输入流（bash_input）读取输入。
   yyparse将解析的命令保留在全局变量GLOBAL_COMMAND中。
   这是执行PROMPT_COMMAND的地方。 */
//#IMP read_command()调用parse_command()
int parse_command ()
{
  int r;
  char *command_to_execute;//要执行的命令

// #NOTE 2019-06-01 heredoc是指写在脚本里面的输入数据
  need_here_doc = 0;
  run_pending_traps ();

  /* 允许在打印每个主要提示之前执行随机命令。
  如果设置了shell变量PROMPT_COMMAND，
--
      command_to_execute = get_string_value ("PROMPT_COMMAND");
      if (command_to_execute)
    execute_variable_command (command_to_execute, "PROMPT_COMMAND");

      if (running_under_emacs == 2)
      //#NOTE 将转义序列发送到emacs术语模式以告诉它当前的工作目录。
    send_pwd_to_eterm ();   /* Yuck */
      // #TODO 2019-06-01 Yuck是呕的意思，这是什么鬼，程序员爆粗口？
    }

  current_command_line_count = 0;
  //#IMP parse_command()调用语法分析器y.tab.c中的yyparse()是语法分析的开始
  //#NOTE bash也是利用yacc生成的代码来完成语法分析的，y.tab.c的yyparse()是yacc自动生成的
  r = yyparse ();

  if (need_here_doc)
    gather_here_documents ();

--
/*  #IMP read_command ()被主函数调用，读取并解析命令，返回解析的状态。
```

```
解析结果的命令字符串保留在全局变量GLOBAL_COMMAND中以供reader_loop使用。
这是执行shell超时代码的地方。*/
int read_command ()
{
  //#NOTE bash中的变量不强调类型，可以认为都是字符串。
  //#NOTE typedef struct variable SHELL_VAR，变量
  SHELL_VAR *tmout_var;
  int tmout_len, result;
  //#NOTE typedef <error-type> SigHandler(int)，错误类型
  SigHandler *old_alrm;

  //#NOTE 配置左侧命令提示符的内容，显示当前目录名字
  set_current_prompt_level (1);
  //#NOTE typedef struct command COMMAND,命令保留变量
  global_command = (COMMAND *)NULL;

  /* #NOTE 只有交互时才会超时 */
  tmout_var = (SHELL_VAR *)NULL;
  tmout_len = 0;
  old_alrm = (SigHandler *)NULL;
  //#IMP 非零意味着此时shell是交互式的。
  //#IMP 一般来说，这意味着shell正在读取输入从键盘。
  if (interactive)
    {
      //#NOTE linux设置的空闲等待时间TMOUT
      tmout_var = find_variable ("TMOUT");
      //#NOTE ((tmout_var)->value != 0)，已经连接
      if (tmout_var && var_isset (tmout_var))
      {
        //#NOTE atoi字符串转换函数，((tmout_var)->value)，value_cell访问变量值
        tmout_len = atoi (value_cell (tmout_var));
        if (tmout_len > 0)
          {
            //#NOTE =>>(SigHandler *)signal (SIGALRM, alrm_catcher)
            old_alrm = set_signal_handler (SIGALRM, alrm_catcher);
            //#NOTE 设置时限
            alarm (tmout_len);
          }
      }
    }

--

  current_command_line_count = 0;
  //#IMP 解析命令
  result = parse_command ();

  //#NOTE 此时shell是交互式的，shell正在读取输入从键盘
  if (interactive && tmout_var && (tmout_len > 0))
    {
      //#NOTE alarm()用来设置信号SIGALRM
      //在经过参数指定的秒数后传送给目前的进程.
      //如果参数为0, 则之前设置的闹钟会被取消, 并将剩下的时间返回.
```

```
      alarm(0);
      set_signal_handler (SIGALRM, old_alrm);
    }
```

eval.c: #TODO

```
int reader_loop ()
{
  int our_indirection_level;
  COMMAND * volatile current_command;

// #TODO 2019-06-01 把它转换为void*的宏？
  USE_VAR(current_command);

  current_command = (COMMAND *)NULL;

  //#NOTE 命令的递归深度
--

#if defined (PROCESS_SUBSTITUTION)
      unlink_fifo_list ();
#endif /* PROCESS_SUBSTITUTION */

      /* #TODO  - 为什么每次都通过循环设置这个？为什么呢
如果SIGINT被困在交互式shell中？
 */
      //#NOTE interactive_shell非零意味着shell作为交互式shell启动
      if (interactive_shell && signal_is_ignored (SIGINT) == 0)
    set_signal_handler (SIGINT, sigint_sighandler);
--
    execute_variable_command (command_to_execute, "PROMPT_COMMAND");

      if (running_under_emacs == 2)
      //#NOTE 将转义序列发送到emacs术语模式以告诉它当前的工作目录。
    send_pwd_to_eterm ();   /* Yuck */
      // #TODO 2019-06-01 Yuck是呸的意思，这是什么鬼，程序员爆粗口？
    }

  current_command_line_count = 0;
  //#IMP parse_command()调用语法分析器y.tab.c中的yyparse()是语法分析的开始
  //#NOTE bash也是利用yacc生成的代码来完成语法分析的，y.tab.c的yyparse()是yacc自动生成的
```

execute_cmd.c: #IMP

```
#endif

#if defined (HAVE_MBSTR_H) && defined (HAVE_MBSCHR)
#  include <mbstr.h>        /* mbschr */
#endif
```

```
/* #IMP extern 关键字可以在一个函数中引用此函数外或此函数所在外的文件的变量*/
extern int dollar_dollar_pid;
extern int posixly_correct;
extern int expand_aliases;
extern int autocd;
extern int breaking, continuing, loop_level;
--
   return values.  Executing a command with nothing in it returns
   EXECUTION_SUCCESS.
//# NOTE 这里可以解释为什么bash里面命令执行成功后什么显示也没有，默认“没有就是最好”
*/

/* #IMP 内部命令调用*/
int
execute_command (command)
     COMMAND *command;
{
  struct fd_bitmap *bitmap;
--
    case cm_arith:
#endif
#if defined (COND_COMMAND)
    case cm_cond:
#endif
// #IMP shell 能执行的所有控制语句
    case cm_case:
    case cm_while:
    case cm_until:
    case cm_if:
    case cm_for:
--

  ignore_return = (command->flags & CMD_IGNORE_RETURN) != 0;

  QUIT;

// #IMP 2019-06-01 一个超长的switch，用于执行不同类型的指令
//比如for，while，select，switch指令
  switch (command->type)
    {
    case cm_simple:
      {
--
}

/* The meaty part of all the executions.  We have to start hacking the
   real execution of commands here.  Fork a process, set things up,
   execute the command. */
// #IMP execute_simple_command
static int
execute_simple_command (simple_command, pipe_in, pipe_out, async, fds_to_close)
     SIMPLE_COM *simple_command;
     int pipe_in, pipe_out, async;
```

```
     struct fd_bitmap *fds_to_close;
--
     We have already found special builtins by this time, so we do not
     set builtin_is_special.  If this is a function or builtin, and we
     have pipes, then fork a subshell in here.  Otherwise, just execute
     the command directly. */
  if (func == 0 && builtin == 0)
    builtin = find_shell_builtin (this_command_name); /* #IMP 通过命令的名字找到要执行的命令在
shell的那个地方*/

  last_shell_builtin = this_shell_builtin;
  this_shell_builtin = builtin;

  if (builtin || func)
--
      break;
    }
  return (r);
}

/* #IMP 执行内建命令*/
static int
execute_builtin (builtin, words, flags, subshell)
     sh_builtin_func_t *builtin;
     WORD_LIST *words;
     int flags, subshell;
--
    }

  executing_builtin++;
  // #NOTE 2019-06-01 执行builtin指令，builtin其实就是一个函数指针
  executing_command_builtin |= builtin   == command_builtin;
  result = ((*builtin) (words->next)); // #IMP 执行执行的语句，会调用不同的内置方法执行具体的命令，
  //#IMP 如cd命令会执行cd.def中的cd_builtin方法，内置方法的文件都在builtins目录下

  /* This shouldn't happen, but in case `return' comes back instead of
     longjmp'ing, we need to unwind. */
  if (posixly_correct && subshell == 0 && builtin == return_builtin && temporary_env)
    discard_unwind_frame ("return_temp_env");
--
   are the redirections to perform.  FDS_TO_CLOSE is the usual bitmap of
   file descriptors to close.

   If BUILTIN is exec_builtin, the redirections specified in REDIRECTS are
   not undone before this function returns. */
/* #IMP
execute_builtin_or_function 方法有一个分支，分为执行内建命令和执行函数
if (builtin)
result = execute_builtin (builtin, words, flags, 0);
else
result = execute_function (var, words, flags, fds_to_close, 0, 0);
--
      else
```

```
    goto parent_return;
    }
#endif /* RESTRICTED_SHELL */

// #IMP 2019-06-01 寻找磁盘上的外部命令
  command = search_for_command (pathname, CMDSRCH_HASH|(stdpath ? CMDSRCH_STDPATH : 0));

  if (command)
    {
      maybe_make_export_env ();
```

execute_cmd.c: #NOTE

```
extern char *glob_argv_flags;
#endif

extern int job_control; /* XXX */

 // #NOTE 2019-06-01 之前的C编译器不支持函数的原型定义，于是发明了__P来解决问题，现在基本上用不上了
extern int close __P((int));

/* Static functions defined and used in this file. */
static void close_pipes __P((int, int));
static void do_piping __P((int, int));
static void bind_lastarg __P((char *));
static int shell_control_structure __P((enum command_type));
static void cleanup_redirects __P((REDIRECT *)); /* #NOTE 清除重定向*/

#if defined (JOB_CONTROL)
static int restore_signal_mask __P((sigset_t *));
#endif

--
      }
    }
}

/* Return the line number of the currently executing command. */
// #NOTE 记录当前正在执行的指令行号，用于继续后面的指令
int
executing_line_number ()
{
  if (executing && showing_function_line == 0 &&
      (variable_context == 0 || interactive_shell == 0) &&
--
  current_fds_to_close = (struct fd_bitmap *)NULL;
  bitmap = new_fd_bitmap (FD_BITMAP_DEFAULT_SIZE);
  begin_unwind_frame ("execute-command");
  add_unwind_protect (dispose_fd_bitmap, (char *)bitmap);

// #NOTE 2019-05-18 非异步执行
```

```
  /* Just do the command, but not asynchronously. */
  result = execute_command_internal (command, 0, NO_PIPE, NO_PIPE, bitmap);

  dispose_fd_bitmap (bitmap);
  discard_unwind_frame ("execute-command");
--
  QUIT;
  return (result);
}

/* Return 1 if TYPE is a shell control structure type. */
// #NOTE  判断是否是控制语句，控制语句跳转到不同地方执行
static int
shell_control_structure (type)
     enum command_type type;
{
  switch (type)
--
}
#endif /* JOB_CONTROL */

#ifdef DEBUG
/* A debugging function that can be called from gdb, for instance. */
// #NOTE 调试？？？
void
open_files ()
{
  register int i;
  int f, fd_table_size;
--
    }
  fprintf (stderr, "\n");
}
#endif

// #NOTE 对标准输入的处理
static void
async_redirect_stdin ()
{
  int fd;
// fd0 :standered input-keyboard
--
// 举例来说，如果是value是for_commmand,即这是一个for循环控制结构命令，则调用execute_for_command函数。
// 在该函数中，将枚举每一个操作域中的元素，对其再次调用execute_command函数进行分析。
// 即execute_for_command这一类函数实现的是一个命令的展开以及流程控制以及递归调用execute_command的功
能。
*/

//#NOTE asynchronous: 异步的
int
execute_command_internal (command, asynchronous, pipe_in, pipe_out,
               fds_to_close)
     COMMAND *command;
```

```
      int asynchronous;
 --
       begin_unwind_frame ("return_temp_env");
       add_unwind_protect (merge_temporary_env, (char *)NULL);
     }

   executing_builtin++;
   // #NOTE 2019-06-01 执行builtin指令，builtin其实就是一个函数指针
   executing_command_builtin |= builtin   == command_builtin;
   result = ((*builtin) (words->next)); // #IMP 执行执行的语句，会调用不同的内置方法执行具体的命令，
   //#IMP 如cd命令会执行cd.def中的cd_builtin方法，内置方法的文件都在builtins目录下

   /* This shouldn't happen, but in case `return' comes back instead of
```

parse.y: #IMP

```
/* Reserved words.  Members of the first group are only recognized
   in the case that they are preceded by a list_terminator.  Members
   of the second group are for [[...]] commands.  Members of the
   third group are recognized only under special circumstances. */

     /* #IMP 2019-06-01 bash中的关键字声明 */
%token IF THEN ELSE ELIF FI CASE ESAC FOR SELECT WHILE UNTIL DO DONE FUNCTION COPROC
%token COND_START COND_END COND_ERROR
%token IN BANG TIME TIMEOPT TIMEIGN

/* More general tokens. yylex () knows how to make these. */
--
            { $$ = make_word_list ($1, (WORD_LIST *)NULL); }
    |   word_list WORD
            { $$ = make_word_list ($2, $1); }
    ;

/* #IMP 2019-06-01 重定向的语法声明 */
redirection:    '>' WORD
            {
              source.dest = 1;
              redir.filename = $2;
              $$ = make_redirection (source, r_output_direction, redir, 0);
```

parse.y: #NOTE

```
              global_command = (COMMAND *)NULL;
              if (parser_state & PST_CMDSUBST)
                parser_state |= PST_EOFTOKEN;
              YYACCEPT;
            }
            /* #NOTE 2019-06-01 error这些在 | 后面的都是编译原理中的非终结符，后面是非终结符对应的一些动
作 */
```

```
    |   error '\n'
            {
              /* Error during parsing.  Return NULL command. */
              global_command = (COMMAND *)NULL;
              eof_encountered = 0;
--
              redir.filename = $2;
              $$ = make_redirection (source, r_input_direction, redir, 0);
            }
    |   NUMBER '>' WORD
            {
                /* #NOTE 2019-06-01 $1指的是NUMBER，$3指的是WORD */
              source.dest = $1;
              redir.filename = $3;
              $$ = make_redirection (source, r_output_direction, redir, 0);
            }
    |   NUMBER '<' WORD
```

parse.y: #TODO

```
/* The list of shell variables that the user has created at the global
   scope, or that came from the environment. */
VAR_CONTEXT *global_variables = (VAR_CONTEXT *)NULL;

// #IMP 2019-06-01 变量和函数的列表
/* The current list of shell variables, including function scopes */
VAR_CONTEXT *shell_variables = (VAR_CONTEXT *)NULL;

/* The list of shell functions that the user has created, or that came from
   the environment. */
--
    var = var_lookup (name, shell_variables);
      else
    {
      /* essentially var_lookup expanded inline so we can check for
         att_invisible */
      // #IMP 2019-06-01 shell_variables存了当前shell的变量列表，是一个哈希表
      for (vc = shell_variables; vc; vc = vc->down)
        {
          var = hash_lookup (name, vc->table);
          if (var && invisible_p (var))
        var = 0;
--
    return ((SHELL_VAR *)NULL);

  return (var->dynamic_value ? (*(var->dynamic_value)) (var) : var);
}

// #IMP 2019-06-01 查找环境变量
/* Look up the variable entry named NAME.  Returns the entry or NULL. */
SHELL_VAR *
```

```
find_variable (name)
     const char *name;
{
```

---

variables.c: #NOTE

```
/* Set to non-zero if an assignment error occurs while putting variables
   into the temporary environment. */
int tempenv_assign_error;

// #NOTE 2019-06-01 $符号的语法定义
/* Some funky variables which are known about specially.  Here is where
   "$*", "$1", and all the cruft is kept. */
char *dollar_vars[10];
WORD_LIST *rest_of_args = (WORD_LIST *)NULL;

--
static void push_func_var __P((PTR_T));
static void push_exported_var __P((PTR_T));

static inline int find_special_var __P((const char *));

// #NOTE 2019-06-01 建立环境变量表
static void
create_variable_tables ()
{
  if (shell_variables == 0)
    {
--
      /* Now, name = env variable name, string = env variable value, and
     char_index == strlen (name) */

      temp_var = (SHELL_VAR *)NULL;

// #NOTE 2019-06-01 载入函数
#if defined (FUNCTION_IMPORT)
      /* If exported function, define it now.  Don't import functions from
     the environment in privileged mode. */
      if (privmode == 0 && read_but_dont_execute == 0 &&
          STREQN (BASHFUNC_PREFIX, name, BASHFUNC_PREFLEN) &&
--
  SHELL_VAR *v;
  int flags;

  last_table_searched = 0;
  flags = 0;
  // #NOTE 2019-06-01 判断是在运行环境中声明的还是内置的变量
  if (expanding_redir == 0 && (assigning_in_environment || executing_builtin))
    flags |= FV_FORCETEMPENV;
  v = find_variable_internal (name, flags);
```

```
  if (v && nameref_p (v))
    v = find_variable_nameref (v);
```

---

variables.h: #IMP

```
/* Shell variables and functions are stored in hash tables. */
#include "hashlib.h"

#include "conftypes.h"

// #IMP 2019-06-01 存储变量列表的数据结构，可以看出来是使用哈希表存储的
/* A variable context. */
typedef struct var_context {
  char *name;       /* empty or NULL means global context */
  int scope;        /* 0 means global context */
  int flags;
```

---

variables.h: #NOTE

```
#define uppercase_p(var)    ((((var)->attributes) & (att_uppercase)))
#define lowercase_p(var)    ((((var)->attributes) & (att_lowercase)))
#define capcase_p(var)      ((((var)->attributes) & (att_capcase)))
#define nameref_p(var)      ((((var)->attributes) & (att_nameref)))

// #NOTE 2019-06-01 判断变量的可见性
#define invisible_p(var)    ((((var)->attributes) & (att_invisible)))
#define non_unsettable_p(var)   ((((var)->attributes) & (att_nounset)))
#define noassign_p(var)     ((((var)->attributes) & (att_noassign)))
#define imported_p(var)     ((((var)->attributes) & (att_imported)))
#define specialvar_p(var)   ((((var)->attributes) & (att_special)))
```

---

command.h: #IMP

```
typedef struct redirect {
  struct redirect *next;    /* Next element, or NULL. */
  REDIRECTEE redirector;    /* Descriptor or varname to be redirected. */
  int rflags;           /* Private flags for this redirection */
  int flags;            /* Flag value for `open'. */
  // #IMP 文件打开方式：readonly, rdwr, ex
  enum r_instruction  instruction; /* What to do with the information. */
  REDIRECTEE redirectee;    /* File descriptor or filename */
  char *here_doc_eof;       /* The word that appeared in <<foo. */
} REDIRECT;

--
#define CMD_LASTPIPE        0x2000
#define CMD_STDPATH     0x4000  /* use standard path for command lookup */
```

```
/* What a command looks like. */
//#NOTE 19-05-18 指令的数据结构
/* What a command looks like. */ // #IMP 定义bash命令结构
typedef struct command {
  enum command_type type;   /* FOR CASE WHILE IF CONNECTION or SIMPLE. */
  int flags;            /* Flags controlling execution environment. */
  int line;         /* line number the command starts on */
  REDIRECT *redirects;      /* Special redirects for FOR CASE, etc. */
--
    struct arith_for_com *ArithFor;
#endif
    struct subshell_com *Subshell;
    struct coproc_com *Coproc;
  } value;
  //#IMP 指令的类型值，shell根据不同value类型执行不同种类的指令，为simple时又会判断时候是builtin指令来调
用
  // execute_command 和  execute_internal_command
} COMMAND;

/* Structure used to represent the CONNECTION type. */
typedef struct connection {
```

command.h: #NOTE

```
#define CMD_COPROC_SUBSHELL 0x1000
#define CMD_LASTPIPE        0x2000
#define CMD_STDPATH     0x4000  /* use standard path for command lookup */

/* What a command looks like. */
//#NOTE 19-05-18 指令的数据结构
/* What a command looks like. */ // #IMP 定义bash命令结构
typedef struct command {
  enum command_type type;   /* FOR CASE WHILE IF CONNECTION or SIMPLE. */
  int flags;            /* Flags controlling execution environment. */
  int line;         /* line number the command starts on */
```

findcmd.c: #NOTE

```
  hashed_file = command = (char *)NULL;

  /* If PATH is in the temporary environment for this command, don't use the
     hash table to search for the full pathname. */
     // #NOTE 2019-06-01 在PATH环境变量中找
  path = find_variable_tempenv ("PATH");
  temp_path = path && tempvar_p (path);

  /* Don't waste time trying to find hashed data for a pathname
     that is already completely specified or if we're using a command-
```

```
--
    }
    }

  if (hashed_file)
    command = hashed_file;
    // #NOTE 2019-06-01 含有/的命令都是绝对命令，不需要查找PATH，直接调用
  else if (absolute_program (pathname))
    /* A command containing a slash is not looked up in PATH or saved in
       the hash table. */
    command = savestring (pathname);
  else
--
      else if (temp_path || path)
    pathlist = value_cell (path);
      else
    pathlist = 0;

// #NOTE 2019-06-01 查找指令
      command = find_user_command_in_path (pathname, pathlist, FS_EXEC_PREFERRED|FS_NODIRS);

      if (command && hashing_enabled && temp_path == 0 && (flags & CMDSRCH_HASH))
    {
      /* If we found the full pathname the same as the command name, the
--
          st = file_status (command);
          if (st & FS_EXECABLE)
            phash_insert ((char *)pathname, command, dot_found_in_search, 1);
        }
      else
    // #NOTE 2019-06-01 将搜索到的指令插入哈希表中，方便下一次查找
        phash_insert ((char *)pathname, command, dot_found_in_search, 1);
    }

      if (flags & CMDSRCH_STDPATH)
    free (pathlist);
```

---

general.h: #NOTE

```
typedef int sh_wassign_func_t __P((WORD_DESC *, int));

typedef int sh_load_func_t __P((char *));
typedef void sh_unload_func_t __P((char *));

// #NOTE 2019-06-01 定义返回值为int，参数为WORD_LIST* 的函数指针为sh_builtin_func_t
typedef int sh_builtin_func_t __P((WORD_LIST *)); /* sh_wlist_func_t */

#endif /* SH_FUNCTION_TYPEDEF */

#define NOW ((time_t) time ((time_t *) 0))
```

---

hashlib.h: #NOTE

```
  unsigned int khash;       /* What key hashes to */
  int times_found;      /* Number of times this item has been found. */
} BUCKET_CONTENTS;

typedef struct hash_table {
  // #NOTE 2019-06-01 哈希桶，存放变量的线性数据结构
  BUCKET_CONTENTS **bucket_array;   /* Where the data is kept. */
  int nbuckets;         /* How many buckets does this table have. */
  int nentries;         /* How many entries does this table have. */
} HASH_TABLE;
```

---

redir.c: #IMP

```
   You should have received a copy of the GNU General Public License
   along with Bash.  If not, see <http://www.gnu.org/licenses/>.
*/


/* #IMP 2019-05-29重定向的定义与执行，以下是关于重定向实现原理的一些文字描述。
在linux中，COMMAND结构用于描述一条bash命令，其中有很多参数来描述这个命令，比如其中的type用于表示一个命令对象代表的命令属于何种类型，
又比如整型变量flags用于记录一系列有关运行环境的标志。同时在COMMAND结构还有一个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。
当语法分析器在遇到重定向语法时，将调用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。
--
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。*/


/* #IMP 2019-05-29 REDIRECT与REDIRECTEE的结构注释
1.REDIRECT的结构体如下所示：
typedef struct redirect {
  struct redirect *next;
  int redirector;
  int flags;
--
#define RF_DEVUDP   6

/* A list of pattern/value pairs for filenames that the redirection
   code handles specially. */

  /*  #IMP 2019-05-30当bash发现重定向时它会先查看重定向的文件是不是特殊文件，
  如果是截获它，自行解释，否则就打开这个文件。此处只是声明，真正的解释还在其他文件中，
  比如真正解释下面第542行/dev/tcp/HOST/PORT 的代码是在 lib/sh/netopen.c 中。*/

  /* #IMP 2019-05-30  bash 在处理重定向的时候，除了支持本地文件外，如果在编译的时候 enable 这些选项：
    H*E_DEV_FD(控制是否支持 /dev/fd/[0-9]*)
    H*E_DEV_STDIN(控制是否支持 /dev/stderr /dev/stdin /dev/stdout)
    NETWORK_REDIRECTIONS(控制是否支持 /dev/(tcp|udp)/*)
    则会支持下面代码中几个特殊的形式：
    /dev/fd/[0-9]*
--

  /* If we are in noclobber mode, you are not allowed to overwrite
     existing files.  Check before opening. */


    //  #IMP 2019-05-30 noclobber用于防止文件被覆盖，所以在这种防止覆盖模式下，当我们打开某一文件之前要判
断它是否已经存在，如果存在就不打开。

  if (noclobber && CLOBBERING_REDIRECT (ri))
    {
      fd = noclobber_open (filename, flags, mode, ri);
      if (fd == NOCLOBBER_REDIRECT)
```

---

redir.c: #NOTE

```
  r_move_input, r_move_output, r_move_input_word, r_move_output_word
};

2.重定向的目标记录存放在REDIRECTEE联合中，它可以是一个文件名或文件描述符。REDIRECTEE的定义如下所示：
typedef union {
  int dest;     #NOTE 2019-05-30 当文件描述符溢出时候dest为负
  WORD_DESC *filename;  #NOTE 2019-05-30此处的文件名为目标文件名
} REDIRECTEE;
```

```
*/

#include "config.h"

--
static REDIRECTEE rd;

/* Set to errno when a here document cannot be created for some reason.
   Used to print a reasonable error message. */

//  #NOTE 2019-05-28用于打印错误信息
static int heredoc_errno;

#define REDIRECTION_ERROR(r, e, fd) \
do { \
  if ((r) < 0) \
--
      last_command_exit_value = EXECUTION_FAILURE;\
      return ((e) == 0 ? EINVAL : (e));\
    } \
} while (0)

void//#NOTE 2019-05-30该模块用于判断各种错误情况，涉及到的错误情况有溢出，覆盖，描述符模糊等，并做出相应的处理。
redirection_error (temp, error)
     REDIRECT *temp;
     int error;
{
  char *filename, *allocname;
  int oflags;

  allocname = 0;
  if ((temp->rflags & REDIR_VARASSIGN) && error < 0)
    filename = allocname = savestring (temp->redirector.filename->word);
    // #NOTE 2019-05-28 当文件描述符溢出时的处理
  else if ((temp->rflags & REDIR_VARASSIGN) == 0 && temp->redirector.dest < 0)
    /* This can happen when read_token_word encounters overflow, like in
       exec 4294967297>x */
    filename = _("file descriptor out of range");
#ifdef EBADF
--
}

/* Return non-zero if the redirection pointed to by REDIRECT has a
   redirectee.filename that can be expanded. */

/* #NOTE 2019-05-30  在REDIRECT这个结构中内部有redirectee这个小的结构用于存放目标描述符或目标文件名。也就说通过redirectee可以找到目标文件。
 当目标文件名可以被扩展的时候以下这个函数会返回1.否则返回0.个人认为是在重定向的过程中，要求返回的文件名应是唯一，容易找到的，如果目标文件还可以被扩
 展的话，是否会对重定向带来一些麻烦？*/
static int
expandable_redirection_filename (redirect)
```

```
     REDIRECT *redirect;
--
  expanding_redir = 1;
  /* Now that we've changed the variable search order to ignore the temp
     environment, see if we need to change the cached IFS values. */


   /* #NOTE 2019-05-30 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*/
  sv_ifs ("IFS");
  tlist2 = expand_words_no_vars (tlist1);
  expanding_redir = 0;
  /* Now we need to change the variable search order back to include the temp
--

/* Write the text of the here document pointed to by REDIRECTEE to the file
   descriptor FD, which is already open to a temp file.  Return 0 if the
   write is successful, otherwise return errno. */

  // #NOTE 2019-05-30 写下目标文件的内容，当成功写入的时候返回0，否则返回写入失败的原因。

static int
write_here_document (fd, redirectee)
     int fd;
     WORD_DESC *redirectee;
--

/* Create a temporary file holding the text of the here document pointed to
   by REDIRECTEE, and return a file descriptor open for reading to the temp
   file.  Return -1 on any error, and make sure errno is set appropriately. */

 /*  #NOTE 2019-05-30 首先创建一个临时文件，用来记录目标文件里的信息，并且返回一个文件描述符fd，通过这个fd
可以找到刚才那个临时文件。
 同时在这个过程中，如果有错误情况发生，也应该做出相应的处理。*/
static int
here_document_to_fd (redirectee, ri)
     WORD_DESC *redirectee;
     enum r_instruction ri;
--

  /* In an attempt to avoid races, we close the first fd only after opening
     the second. */
  /* Make the document really temporary.  Also make it the input. */

  //#NOTE 2019-05-30在重定向实现过程中，为了避免竞争，必须保证在复制之前，应先把之前的关闭。
  fd2 = open (filename, O_RDONLY|O_BINARY, 0600);

  if (fd2 < 0)
    {
      r = errno;
--
/* Open FILENAME with FLAGS in noclobber mode, hopefully avoiding most
   race conditions and avoiding the problem where the file is replaced
```

```
   between the stat(2) and open(2). */


//   #NOTE 2019-05-30 防止文件被覆盖
static int
noclobber_open (filename, flags, mode, ri)
     char *filename;
     int flags, mode;
     enum r_instruction ri;
{
  int r, fd;
  struct stat finfo, finfo2;

   //  #NOTE 2019-05-30 如果文件存在并且正常，（r=0表示文件存在）那么就不再下载已经存在的文件，也就说防止
文件被覆盖。
  r = stat (filename, &finfo);
  if (r == 0 && (S_ISREG (finfo.st_mode)))
    return (NOCLOBBER_REDIRECT);

  /* If the file was not present (r != 0), make sure we open it
--
     will fail.  Make sure that we do not truncate an existing file.
     Note that we don't turn on O_EXCL unless the stat failed -- if
     the file was not a regular file, we leave O_EXCL off. */


   //  #NOTE 2019-05-30当r!=0的时候，表示文件不存在，那么就打开它，并返回fd.
  flags &= ~O_TRUNC;
  if (r != 0)
    {
      fd = open (filename, flags|O_EXCL, mode);
      return ((fd < 0 && errno == EEXIST) ? NOCLOBBER_REDIRECT : fd);
--
    return (NOCLOBBER_REDIRECT);
    }
  else
    {
      do
     /* #NOTE 2019-05-30 关于int open(char *filename, int flags, mode_t mode);
      #中的参数flags为以下值或者组合时：
      #O_RDONLY ：没有文件时返回 -1；有文件时跳过，并且保留文件原来内容
      #O_WRONLY ：没有文件时返回 -1；有文件时跳过，并且保留文件原来内容
      #O_RDWR ：没有文件时返回 -1；有文件时跳过，并且保留文件原来内容

--
   says to remember how to undo each redirection.  If flags & RX_CLEXEC is
   non-zero, then we set all file descriptors > 2 that we open to be
   close-on-exec.  */


  /*  #NOTE 2019-05-30  这个模块用于处理特殊的文件重定向，其中用到了三个flag，它们在redir.h中曾被定义，
      它们的定义如下所示：
         RX_ACTIVE  0x01    /* do it; don't just go through the motions
```

```
        RX_UNDOABLE    0x02    /* make a list to undo these redirections
        RX_CLEXEC  0x04    /* set close-on-exec for opened fds > 2 */
```

---

redir.c: #TODO

```
  else if (error != NOCLOBBER_REDIRECT && temp->redirector.dest >= 0 && error == EBADF)
    {
      /* If we're dealing with two file descriptors, we have to guess about
         which one is invalid; in the cases of r_{duplicating,move}_input and
         r_{duplicating,move}_output we're here because dup2() failed. */
         //#TODO 2019-05-30当使用到dup2()函数时候应该考虑到的错误处理机制？
      switch (temp->instruction)
        {
        case r_duplicating_input:
        case r_duplicating_output:
        case r_move_input:
--
   how to undo the redirections later, if non-zero.  If flags & RX_CLEXEC
   is non-zero, file descriptors opened in do_redirection () have their
   close-on-exec flag set. */


  // #TODO 2019-05-30
int
do_redirections (list, flags)
     REDIRECT *list;
     int flags;
{
--
}

/* Expand the word in WORD returning a string.  If WORD expands to
   multiple words (or no words), then return NULL. */

//#TODO 2019-05-30


char *
redirection_expand (word)
     WORD_DESC *word;
--

static int
write_here_string (fd, redirectee)
     int fd;
     WORD_DESC *redirectee;
{#TODO
  char *herestr;
  int herelen, n, e, old;
```

```
expanding_redir = 1;
/* Now that we've changed the variable search order to ignore the temp
```