Content

CHAPTER 1

python 进程的生命周期

1.1 cpython 主函数

同 Linux kernel 和我们入门时的“hello world”一样，cpython 项目也是由 C 语言编写而成的，所以这里的 cpython 其实和我们曾经玩过的“hello world”没什么区别。额。。。当然啦，这里只是说编程语言一致而已，别太当真。严格意义上讲，cpython 相比于我们以往在 C 语言课程或书籍中所看到的代码片段还是要复杂得多的。

任何 C 语言编制的项目都始于一个名为”main” 的特殊函数，python 也不例外。python 的 main 函数定义在 Prgrams/python.c 文件下：出于可移植性的考虑，python 的 main 函数被划分为两类，适应于 Windows 系统的 wmain 和适应于类 Unix 平台 (如：Linux) 的 main 两类，详细代码如下:

```
#include "Python.h"
#ifdef MS_WINDOWS
int
wmain(int argc, wchar_t **argv)
{
    return Py_Main(argc, argv);
}
#else
int
main(int argc, char **argv)
{
    return _Py_UnixMain(argc, argv);
}
#endif
```

两函数均定义在 Modules/main.c 文件内,通过下面的代码,不难看出,对于不同的平台两函数除了对于 _PyMain 类型的结构体的赋值过程有所不同之外，函数内部均调用了以 Py_Main 结构体为输入参数的 pymain_main 函数，详细代码如下[1]：

```
int
Py_Main(int argc, wchar_t **argv)
{
    _PyMain pymain = _PyMain_INIT;
    pymain.use_bytes_argv = 0;
    pymain.argc = argc;
    pymain.wchar_argv = argv;

    return pymain_main(&pymain);
}
```

[1]个人还是比较喜欢这种应对多平台 (或者多工作环境) 的函数编制模式：即采用统一的函数模块执行公共的代码段，将代码的不同点通过输入参数的变化找到函数体内对应的不同分支

```
int
_Py_UnixMain(int argc, char **argv)
{
    _PyMain pymain = _PyMain_INIT;
    pymain.use_bytes_argv = 1;
    pymain.argc = argc;
    pymain.bytes_argv = argv;

    return pymain_main(&pymain);
    }
```

首先对 _PyMain 结构体进行分析，该结构体定义在 Modules/main.c 文件内。我们可以将 _PyMain 结构体肢解为三个部分：

#3-6　对传入主函数的变量的记录。

#9-11　对程序执行状态的记录。

#13-20　python 程序解析相关参数。

```
typedef struct {
    /* Input arguments */
    int argc;
    int use_bytes_argv;
    char **bytes_argv;
    wchar_t **wchar_argv;

    /* Exit status or "exit code": result of pymain_main() */
    int status;
    /* Error message if a function failed */
    _PyInitError err;

    /* non-zero is stdin is a TTY or if -i option is used */
    int stdin_is_interactive;
    int skip_first_line; /* -x option */
    wchar_t *filename; /* Trailing arg without -c or -m */
    wchar_t *command; /* -c argument */
    wchar_t *module; /* -m argument */

    PyObject *main_importer_path;
} _PyMain;
```

在执行 pymain_main 函数之前，python 进程只是将传入 C 语言主函数的参数赋到 _PyMain 结构体里面并实现对输入主函数变量的记录以及当前程序执行状态的初始化。

```
    /*== Modules/main.c #487 ==*/
    #define _PyMain_INIT {.err = _Py_INIT_OK()}

    /*== Include/pylifecycle.h #25-26 ==*/
    #define _Py_INIT_OK() \
        (_PyInitError){.prefix = NULL, .msg = NULL, .user_err = 0}

    /*== Include/pylifecycle.h #11-15 ==*/
    typedef struct {
        const char *prefix;
        const char *msg;
        int user_err;
    } _PyInitError;
```

其中，对输入主函数变量的记录是由 Py_Main 函数的 #5-7[1]完成。use_bytes_argv 用于区分 python 进程所运行

[1] _Py_UnixMain 函数的 #17-19

的平台[2]。argc 用于记录输入终端的变量个数，即对传入主函数的 argc 变量的记录。wchar_argv 和 bytes_argv 分别用来对传入执行在 Windows 和类 Unix 平台上的主函数的 argv 变量进行记录。

对于当前程序执行状态的初始化，则是由 _PyMain_INIT 宏完成的，其实际上是对 _PyMain 结构体内的 _PyInitError 结构体进行初始化，即指定 _PyInitError 结构体内的 user_err 变量为 0，变量 prefix 以及 msg 均为 NULL。

```
1  /*== Modules/main.c #3020-3040 ==*/
2  static int
3  pymain_main(_PyMain *pymain)
4  {
5      /*-- Initialization --*/
6      int res = pymain_init(pymain);
7      if (res == 1) {
8          goto done;
9      }
10     /*-- Execution --*/
11     pymain_run_python(pymain);
12
13     /*-- Finalization --*/
14     if (Py_FinalizeEx() < 0) {
15         /* Value unlikely to be confused with a non-error exit status or
16            other special meaning */
17         pymain->status = 120;
18     }
19
20 done:
21     pymain_free(pymain);
22
23     return pymain->status;
24 }
```

大体上可以将上述 pymain_main 函数体分为三个部分，后续小节也将依次对三个部分进行分析：

#5-9　　程序初始化。

#11　　程序执行。

#14-21　程序终止。

[2]0 代表 Windows，1 代表类 Unix 平台

1.2 pymain_main 函数初始化

1.2.1 pymain_main 函数主体

python 主进程的初始化主要在 pymain_init 函数中进行，函数的相关代码如下，可以将初始化分成：状态初置[1]和异常处理两部分来分析。

```
1  /*== Modules/main.c #2970-3017 ==*/
2
3  static int
4  pymain_init(_PyMain *pymain)
5  {
6      _PyCoreConfig local_config = _PyCoreConfig_INIT;
7      _PyCoreConfig *config = &local_config;
8
9      /* 754 requires that FP exceptions run in "no stop" mode by default,
10     * and until C vendors implement C99's ways to control FP exceptions,
11     * Python requires non-stop mode. Alas, some platforms enable FP
12     * exceptions by default. Here we disable them.
13     */
14 #ifdef __FreeBSD__
15     fedisableexcept(FE_OVERFLOW);
16 #endif
17
18     config->_disable_importlib = 0;
19     config->install_signal_handlers = 1;
20     _PyCoreConfig_GetGlobalConfig(config);
21
22     int res = pymain_cmdline(pymain, config);
23     if (res < 0) {
24         _Py_FatalInitError(pymain->err);
25     }
26     if (res == 1) {
27         pymain_clear_config(&local_config);
28         return res;
29     }
30
31     pymain_init_stdio(pymain);
32
33     PyInterpreterState *interp;
34     pymain->err = _Py_InitializeCore(&interp, config);
35     if (_Py_INIT_FAILED(pymain->err)) {
36         _Py_FatalInitError(pymain->err);
37     }
38
39     pymain_clear_config(&local_config);
40     config = &interp->core_config;
41
42     if (pymain_init_python_main(pymain, interp) < 0) {
43         _Py_FatalInitError(pymain->err);
44     }
45
46     if (pymain_init_sys_path(pymain, config) < 0) {
47         _Py_FatalInitError(pymain->err);
48     }
49     return 0;
50 }
```

[1] #6-22,31-34

1.2.2 cpython 终端参数解析

状态初置涉及到一个新的结构体 _PyCoreConfig，具体代码如下：

```
/*== Include/pystate.h #29-77==*/

typedef struct {
    int install_signal_handlers; /* Install signal handlers? -1 means unset */

    int ignore_environment; /* -E, Py_IgnoreEnvironmentFlag */
    int use_hash_seed; /* PYTHONHASHSEED=x */
    unsigned long hash_seed;
    const char *allocator; /* Memory allocator: _PyMem_SetupAllocators() */
    int dev_mode; /* PYTHONDEVMODE, -X dev */
    int faulthandler; /* PYTHONFAULTHANDLER, -X faulthandler */
    int tracemalloc; /* PYTHONTRACEMALLOC, -X tracemalloc=N */
    int import_time; /* PYTHONPROFILEIMPORTTIME, -X importtime */
    int show_ref_count; /* -X showrefcount */
    int show_alloc_count; /* -X showalloccount */
    int dump_refs; /* PYTHONDUMPREFS */
    int malloc_stats; /* PYTHONMALLOCSTATS */
    int coerce_c_locale; /* PYTHONCOERCECLOCALE, -1 means unknown */
    int coerce_c_locale_warn; /* PYTHONCOERCECLOCALE=warn */
    int utf8_mode; /* PYTHONUTF8, -X utf8; -1 means unknown */

    wchar_t *program_name; /* Program name, see also Py_GetProgramName() */
    int argc; /* Number of command line arguments,
                          -1 means unset */
    wchar_t **argv; /* Command line arguments */
    wchar_t *program; /* argv[0] or "" */

    int nxoption; /* Number of -X options */
    wchar_t **xoptions; /* -X options */

    int nwarnoption; /* Number of warnings options */
    wchar_t **warnoptions; /* Warnings options */

    /* Path configuration inputs */
    wchar_t *module_search_path_env; /* PYTHONPATH environment variable */
    wchar_t *home; /* PYTHONHOME environment variable,
                          see also Py_SetPythonHome(). */

    /* Path configuration outputs */
    int nmodule_search_path; /* Number of sys.path paths,
                                -1 means unset */
    wchar_t **module_search_paths; /* sys.path paths */
    wchar_t *executable; /* sys.executable */
    wchar_t *prefix; /* sys.prefix */
    wchar_t *base_prefix; /* sys.base_prefix */
    wchar_t *exec_prefix; /* sys.exec_prefix */
    wchar_t *base_exec_prefix; /* sys.base_exec_prefix */

    /* Private fields */
    int _disable_importlib; /* Needed by freeze_importlib */
} _PyCoreConfig;
```

这里主要是利用 _PyCoreConfig_INIT 宏以及 _PyCoreConfig_GetGlobalConfig 函数设置 _PyCoreConfig 的初始参数，可以看到这里基本上仍然将大部分参数设置为-1，即默认或 unknow 状态，等待后续函数的处理。相关代码如下：

```
/*== Include/pystate.h #79-89 ==*/
#define _PyCoreConfig_INIT \
(_PyCoreConfig){ \
    .install_signal_handlers = -1, \
    .ignore_environment = -1, \
    .use_hash_seed = -1, \
    .coerce_c_locale = -1, \
    .faulthandler = -1, \
    .tracemalloc = -1, \
    .utf8_mode = -1, \
    .argc = -1, \
    .nmodule_search_path = -1}

/*== Include/pystate.h #1433-1445 ==*/
void
_PyCoreConfig_GetGlobalConfig(_PyCoreConfig *config)
{
#define COPY_FLAG(ATTR, VALUE) \
       if (config->ATTR == -1) { \
           config->ATTR = VALUE; \
       }

    COPY_FLAG(ignore_environment, Py_IgnoreEnvironmentFlag);
    COPY_FLAG(utf8_mode, Py_UTF8Mode);

#undef COPY_FLAG
}

/*== Python/pylifecycle.c #122==*/
int Py_IgnoreEnvironmentFlag = 0;

/*== Python/bltinmodule.c #35 ==*/
int Py_UTF8Mode = -1;
/* UTF-8 mode (PEP 540): if equals to 1, use the UTF-8 encoding, and change
stdin and stdout error handler to "surrogateescape". It is equal to
-1 by default: unknown, will be set by Py_Main() */
```

在相关结构体初始化之后，后面紧接着将会对 python 主进程所用的存储空间进行申请和分配，并且对程序的执行状态信息记录作初始化操作。这里重点研究 pymain_cmdline_impl 函数的内容，其他部分将结合后续章节内容进行分析。

```
/*== Modules/main.c #2932-2967 ==*/
static int
pymain_cmdline(_PyMain *pymain, _PyCoreConfig *config)
{
    /* Force default allocator, since pymain_free() and pymain_clear_config()
       must use the same allocator than this function. */
    ...

    _PyCmdline cmdline;
    memset(&cmdline, 0, sizeof(cmdline));

    cmdline_get_global_config(&cmdline);

    int res = pymain_cmdline_impl(pymain, config, &cmdline);
   ...
    return res;
}
```

在正式进入对 pymain_cmdline_impl 函数体分析之前，我们先对该函数的三个输入参数进行简要的分析，前两个参数分别为 _PyMain 和 _PyCoreConfig 结构体，分别用于记录主进程的执行情况和基本资源配置，这里我们将对 _PyCmdline 这个新的结构体进行简要的说明。

我们在使用 python 时，经常会考虑程序是如何根据我们在终端的输入指令来判断后续需要执行的操作的，例如：在终端输入 python，就会自动进入 REPL 交互命令行中；如果输入 python –help，终端就会自动回复 python 的相关使用说明，如果输入 python *.py，后面就会自动执行相应 python 代码文件内的程序。按照一般的思路，我们只需要在程序内将指定的字符对应到相应的 python 功能上就 OK 了，cpython 就利用了 _Pycmdline 这一结构体来存储终端 python 命令不同参数所对应的功能，该结构体的定义如下[1]：

```
   /*== Modules/main.c #435-462 ==*/
   typedef struct {
   wchar_t **argv;
   int nwarnoption; /* Number of -W options */
   wchar_t **warnoptions; /* -W options */
   int nenv_warnoption; /* Number of PYTHONWARNINGS options */
   ...
   int quiet_flag; /* Py_QuietFlag, -q */
   const char *check_hash_pycs_mode; /* --check-hash-based-pycs */
#ifdef MS_WINDOWS
   int legacy_windows_fs_encoding; /* Py_LegacyWindowsFSEncodingFlag,
                                   PYTHONLEGACYWINDOWSFSENCODING */
   int legacy_windows_stdio; /* Py_LegacyWindowsStdioFlag,
                                   PYTHONLEGACYWINDOWSSTDIO */
#endif
} _PyCmdline;
```

```
   /*== Modules/main.c #2874-2917 ==*/
   static int
   pymain_cmdline_impl(_PyMain *pymain, _PyCoreConfig *config,
                       _PyCmdline *cmdline)
   {
      pymain->err = _PyRuntime_Initialize();
      ...
      int res = pymain_read_conf(pymain, config, cmdline);
      ...
      if (cmdline->print_help) {
         pymain_usage(0, config->program);
         return 1;
      }

      if (cmdline->print_version) {
         printf("Python %s\n",
               (cmdline->print_version >= 2) ? Py_GetVersion() : PY_VERSION);
         return 1;
      }
      ...
      return 0;
   }
```

除去异常处理的代码段，pymain_cmdline_impl 函数的主体主要分为初始化，终端输入参数解析两个部分，分别由 _PyRuntime_Initialize 和 pymain_read_conf 两函数实现[2]。作为对终端输入参数解析的重要函数之一，pymain_read_conf 函数实现了依据不同的终端 python 指令参数，给出对应的执行指令以另后续的执行函数正式进行

[1]这里仅列出源代码的一部分，后面我们将会看到，cpython 通过 switch 结构将终端输入对应到 _Pycmdline 结构体的不同单元上，以指示不同的 python 操作。

[2]这里重点关注 pymain_read_conf 函数，对于初始化相关的部分在后续章节进行分析。

相关操作的导引功能。这里需要注意的是对于 help 和 version 两指令的实际执行过程已经在 pymain_cmdline_impl 中体现[1]。

```
/*== Modules/main.c #2052-2168 ==*/
static int
pymain_read_conf(_PyMain *pymain, _PyCoreConfig *config, _PyCmdline *cmdline)
{
    ...
    int loops = 0;
    ...
    while (1) {
        ...
        /* Watchdog to prevent an infinite loop */
        loops++;
        if (loops == 3) {
            pymain->err = _Py_INIT_ERR("Encoding changed twice while "
                                       "reading the configuration");
            goto done;
        }

        if (pymain_init_cmdline_argv(pymain, config, cmdline) < 0) {
            goto done;
        }

        int conf_res = pymain_read_conf_impl(pymain, config, cmdline);
        if (conf_res != 0) {
            res = conf_res;
            goto done;
        }
    ...
    }
    res = 0;

done:
    ...
    return res;
}
```

pymain_read_conf 函数中的核心部分为定义在 Modules/main.c 文件 #2077-2157 的包含看门狗机制的循环代码段上。其中对于 python 相关终端输入的解析主要由 pymain_init_cmdline_argv[3]和 pymain_read_conf_impl 两函数配合实现。

```
/*== Modules/main.c #1993-2047 ==*/
static int
pymain_read_conf_impl(_PyMain *pymain, _PyCoreConfig *config,
                      _PyCmdline *cmdline)
{
    ...
    int res = pymain_parse_cmdline(pymain, config, cmdline);
    if (res != 0) {
        return res;
    }

    ...
    return 0;
}
```

[1]help 对应 #10-12，version 对应 #15-19

[3]pymain_init_cmdline_argv 函数定义在 Modules/main.c 文件 #520-567，主要是将类 Unix 系统中 cpython 主函数读取的终端参数由 char 数据类型转换为 wchar_t 类型，方便后续的处理。

```c
/*== Modules/main.c #1637-1657 ==*/
static int
pymain_parse_cmdline(_PyMain *pymain, _PyCoreConfig *config,
                     _PyCmdline *cmdline)
{
    int res = pymain_parse_cmdline_impl(pymain, config, cmdline);
    if (res < 0) {
        return -1;
    }
    ...
    return 0;
}
```

通过源码可以看出 pymain_read_conf_impl 实际是对 pymain_parse_cmdline_impl 函数的“包装”[1]

```c
/*== Modules/main.c #739-911 ==*/
static int
pymain_parse_cmdline_impl(_PyMain *pymain, _PyCoreConfig *config,
                          _PyCmdline *cmdline)
{
    _PyOS_ResetGetOpt();
    do {
        int longindex = -1;
        int c = _PyOS_GetOpt(pymain->argc, cmdline->argv, PROGRAM_OPTS,
                             longoptions, &longindex);
        if (c == EOF) {
            break;
        }

        if (c == 'c') {
            ...
        }
        if (c == 'm') {
            ...
        }
        switch (c) {
        ...
        case 'b':
            cmdline->bytes_warning++;
            break;
        ...
        }
    } while (1);

    if (pymain->command == NULL && pymain->module == NULL
        && _PyOS_optind < pymain->argc
        && wcscmp(cmdline->argv[_PyOS_optind], L"-") != 0)
    {
        pymain->filename = pymain_wstrdup(pymain, cmdline->argv[_PyOS_optind]);
        if (pymain->filename == NULL) {
            return -1;
        }
    }
...
}
```

[1]略去的部分是 pymain_parse_cmdline 中对于 pymain_parse_cmdline_impl 函数异常输出的一个处理，具体的操作为当 cpython 主函数接收到的终端参数不在备选集合范围内时，将会由 pymain_parse_cmdline_impl 返回一个非 0 的 int 类型数据作为错误指示，引导 pymain_parse_cmdline 在后续的操作中调用定义在 Modules/main.c 文件 #151-167 的 pymain_usage 函数以实现在同一终端下输出 python 的终端命令使用导引，引导用户输入正确的 python 终端指令。

pymain_parse_cmdline_impl 通过调用 _PyOS_GetOpt 函数[1]将终端参数转化为对应的特征指示变量 c 输出，然后由后续的三个条件语句[2]以及 switch 分支代码段进行相关操作的标记。

对 pymain_parse_cmdline_impl 函数分析的重点放在了 #23-25 和 #30-37 两部分。对于输入的终端的第二个字符指令非文件名的情况，pymain_parse_cmdline_impl 函数大体采用 #23-25 的做法：即通过将 _PyCmdline 结构体 cmdline 中同指示变量 c 所对应的成员的值由 0 通过“++”操作变为 1，指示后续的操作。对于输入代码文件的情况，则如同 #30-37 所定义的一样通过调用 pymain_wstrdup 函数[3]去除无效字符 [4]仅保留文件名之后，将该文件名所对应的字符串指针传输给 _PyMain 结构体 pymain 的 filename 成员，以便后续操作的进行。

```
1   /*=== Modules/main.c #2932-2967 ===*/
2   static int
3   pymain_cmdline(_PyMain *pymain, _PyCoreConfig *config)
4   {
5       /* Force default allocator, since pymain_free() and pymain_clear_config()
6          must use the same allocator than this function. */
7       PyMemAllocatorEx old_alloc;
8       _PyMem_SetDefaultAllocator(PYMEM_DOMAIN_RAW, &old_alloc);
9   #ifdef Py_DEBUG
10      PyMemAllocatorEx default_alloc;
11      PyMem_GetAllocator(PYMEM_DOMAIN_RAW, &default_alloc);
12  #endif
13
14      _PyCmdline cmdline;
15      memset(&cmdline, 0, sizeof(cmdline));
16
17      cmdline_get_global_config(&cmdline);
18
19      int res = pymain_cmdline_impl(pymain, config, &cmdline);
20
21      cmdline_set_global_config(&cmdline);
22      _PyCoreConfig_SetGlobalConfig(config);
23      if (Py_IsolatedFlag) {
24          Py_IgnoreEnvironmentFlag = 1;
25          Py_NoUserSiteDirectory = 1;
26      }
27
28      pymain_clear_cmdline(pymain, &cmdline);
29
30  #ifdef Py_DEBUG
31      /* Make sure that PYMEM_DOMAIN_RAW has not been modified */
32      PyMemAllocatorEx cur_alloc;
33      PyMem_GetAllocator(PYMEM_DOMAIN_RAW, &cur_alloc);
34      assert(memcmp(&cur_alloc, &default_alloc, sizeof(cur_alloc)) == 0);
35  #endif
36      PyMem_SetAllocator(PYMEM_DOMAIN_RAW, &old_alloc);
37      return res;
38  }
```

[1] _PyOS_GetOpt 函数定义在 Python/getopt.c 文件内。

[2] 仅保留对指示变量 c 是否为“m”以及“c”的条件代码块的主干进行保留，以展示代码的主体结构。

[3] pymain_wstrdup 函数定义在 Modules/main.c 文件 #498-507。

[4] 如 NULL。