《源码探秘 CPython》83. 激活 Python 虚拟机

原创 古明地觉 古明地觉的编程教室 2022-05-08 09:30 发表于北京

收录于合集
#CPython
97个 >

Python 的运行方式有两种，一种是在命令行中输入 python 进入交互式环境；另一种则是以 **python xxx.py** 的方式运行脚本文件。尽管方式不同，但最终殊途同归，进入相同的处理逻辑。

而 Python 在初始化（Py_Initialize）完成之后，会执行 pymain_run_file。

```c
//Modules/main.c
static int
pymain_run_file(PyConfig *config, PyCompilerFlags *cf)
{
    //获取文件名
    const wchar_t *filename = config->run_filename;
    if (PySys_Audit("cpython.run_file", "u", filename) < 0) {
        return pymain_exit_err_print();
    }
    //打开文件
    FILE *fp = _Py_wfopen(filename, L"rb");
    //如果fp为NULL，证明文件打开失败
    if (fp == NULL) {
        char *cfilename_buffer;
        const char *cfilename;
        int err = errno;
        cfilename_buffer = _Py_EncodeLocaleRaw(filename, NULL);
        if (cfilename_buffer != NULL)
            cfilename = cfilename_buffer;
        else
            cfilename = "<unprintable file name>";
        fprintf(stderr, "%ls: can't open file '%s': [Errno %d] %s\n",
                config->program_name, cfilename, err, strerror(err));
        PyMem_RawFree(cfilename_buffer);
        return 2;
    }
    //......
    //调用PyRun_AnyFileExFlags
    int run = PyRun_AnyFileExFlags(fp, filename_str, 1, cf);
    Py_XDECREF(bytes);
    return (run != 0);
}


//Python/pythonrun.c
int
PyRun_AnyFileExFlags(FILE *fp, const char *filename, int closeit,
                     PyCompilerFlags *flags)
{
    if (filename == NULL)
        filename = "???";
    //根据fp是否代表交互环境，对程序进行流程控制
    if (Py_FdIsInteractive(fp, filename)) {
        //如果是交互环境，调用PyRun_InteractiveLoopFlags
        int err = PyRun_InteractiveLoopFlags(fp, filename, flags);
```

```
        if (closeit)
            fclose(fp);
        return err;
    }
    else
        //否则说明是一个普通的python脚本
        //执行PyRun_SimpleFileExFlags
        return PyRun_SimpleFileExFlags(fp, filename, closeit, flags);
}
```

我们看到交互式和执行 py 脚本方式调用的是两个不同的函数，但是别着急，最终你会看到它们又分久必合、走到一起。

看看交互式运行时候的情形，不过在此之前先来看一下提示符。

```
>>> name = "satori"
>>> if name == "satori":
...     pass
...
>>> import sys
>>> sys.ps1 = "+++ "
+++
+++ sys.ps2 = "--- "
+++
+++ if name == "satori":
---     pass
---
+++
```

我们每输入一行，开头都是 >>>，这个是 sys.ps1；而输入语句块，没输入完的时候，那么显示 ...，这个是 sys.ps2。而这两者都支持修改，如果修改了，那么就是我们自己定义的了。

交互式环境会执行 PyRun_InteractiveLoopFlags 函数。

```
int
PyRun_InteractiveLoopFlags(FILE *fp, const char *filename_str, PyCompil
erFlags *flags)
{
    //....
    //创建交互式提示符
    v = _PySys_GetObjectId(&PyId_ps1);
    if (v == NULL) {
        _PySys_SetObjectId(&PyId_ps1, v = PyUnicode_FromString(">>> "));
        Py_XDECREF(v);
    }
    //同理这个也是一样
    v = _PySys_GetObjectId(&PyId_ps2);
    if (v == NULL) {
        _PySys_SetObjectId(&PyId_ps2, v = PyUnicode_FromString("... "));
        Py_XDECREF(v);
    }
    err = 0;
    do {
        //这里就进入了交互式环境
        //我们看到每次都调用了PyRun_InteractiveOneObjectEx
        //直到下面的ret != E_EOF不成立，停止循环
```

```
        //一般情况就是我们输入exit()退出了
        ret = PyRun_InteractiveOneObjectEx(fp, filename, flags);
        if (ret == -1 && PyErr_Occurred()) {
            if (PyErr_ExceptionMatches(PyExc_MemoryError)) {
                if (++nomem_count > 16) {
                    PyErr_Clear();
                    err = -1;
                    break;
                }
            } else {
                nomem_count = 0;
            }
            PyErr_Print();
            flush_io();
        } else {
            nomem_count = 0;
        }
    //......
    } while (ret != E_EOF);
    Py_DECREF(filename);
    return err;
}


static int
PyRun_InteractiveOneObjectEx(FILE *fp, PyObject *filename,
                             PyCompilerFlags *flags)
{
    PyObject *m, *d, *v, *w, *oenc = NULL, *mod_name;
    mod_ty mod;
    PyArena *arena;
    const char *ps1 = "", *ps2 = "", *enc = NULL;
    int errcode = 0;
    _Py_IDENTIFIER(encoding);
    _Py_IDENTIFIER(__main__);

    mod_name = _PyUnicode_FromId(&PyId___main__); /* borrowed */
    if (mod_name == NULL) {
        return -1;
    }

    if (fp == stdin) {
        //......
    }
    v = _PySys_GetObjectId(&PyId_ps1);
    if (v != NULL) {
        //......
    }
    w = _PySys_GetObjectId(&PyId_ps2);
    if (w != NULL) {
        //.....
    }
    //编译用户在交互式环境下输入的python语句
    arena = PyArena_New();
    if (arena == NULL) {
        Py_XDECREF(v);
        Py_XDECREF(w);
        Py_XDECREF(oenc);
        return -1;
    }
    //生成抽象语法树
    mod = PyParser_ASTFromFileObject(fp, filename, enc,
                                     Py_single_input, ps1, ps2,
                                     flags, &errcode, arena);
```

```
    Py_XDECREF(v);
    Py_XDECREF(w);
    Py_XDECREF(oenc);
    if (mod == NULL) {
        PyArena_Free(arena);
        if (errcode == E_EOF) {
            PyErr_Clear();
            return E_EOF;
        }
        return -1;
    }
    //获取<module __main__>中维护的dict
    m = PyImport_AddModuleObject(mod_name);
    if (m == NULL) {
        PyArena_Free(arena);
        return -1;
    }
    d = PyModule_GetDict(m);
    //执行用户输入的python语句
    v = run_mod(mod, filename, d, d, flags, arena);
    PyArena_Free(arena);
    if (v == NULL) {
        return -1;
    }
    Py_DECREF(v);
    flush_io();
    return 0;
}
```

在run_mod之前，Python 会将 __main__ 中维护的 PyDictObject 对象取出，作为参数传递给 run_mod 函数，这个参数关系极为重要，实际上代码中的参数 d 就将作为虚拟机开始执行时、当前活动 frame 对象的 local 空间和 global 空间。

脚本文件运行方式

然后是脚本文件运行方式。

```
//.include/compile.h
#define Py_file_input 257


//Python/pythonrun.c
int
PyRun_SimpleFileExFlags(FILE *fp, const char *filename, int closeit,
                        PyCompilerFlags *flags)
{
    PyObject *m, *d, *v;
    const char *ext;
    int set_file_name = 0, ret = -1;
    size_t len;
    //__main__就是当前文件
    m = PyImport_AddModule("__main__");
    if (m == NULL)
        return -1;
    Py_INCREF(m);
    //还记得这个d吗?
    //当前活动frame对象的local和global名字空间
    d = PyModule_GetDict(m);
```

```
    //在__main__中设置__file__属性
    if (PyDict_GetItemString(d, "__file__") == NULL) {
        PyObject *f;
        f = PyUnicode_DecodeFSDefault(filename);
        if (f == NULL)
            goto done;
        if (PyDict_SetItemString(d, "__file__", f) < 0) {
            Py_DECREF(f);
            goto done;
        }
        if (PyDict_SetItemString(d, "__cached__", Py_None) < 0) {
            Py_DECREF(f);
            goto done;
        }
        set_file_name = 1;
        Py_DECREF(f);
    }
    len = strlen(filename);
    ext = filename + len - (len > 4 ? 4 : 0);
    //如果是pyc
    if (maybe_pyc_file(fp, filename, ext, closeit)) {
        FILE *pyc_fp;
        //二进制模式打开
        if (closeit)
            fclose(fp);
        if ((pyc_fp = _Py_fopen(filename, "rb")) == NULL) {
            fprintf(stderr, "python: Can't reopen .pyc file\n");
            goto done;
        }

        if (set_main_loader(d, filename, "SourcelessFileLoader") < 0) {
            fprintf(stderr, "python: failed to set __main__.__loader__\n"
);
            ret = -1;
            fclose(pyc_fp);
            goto done;
        }
        v = run_pyc_file(pyc_fp, filename, d, d, flags);
    } else {
        if (strcmp(filename, "<stdin>") != 0 &&
            set_main_loader(d, filename, "SourceFileLoader") < 0) {
            fprintf(stderr, "python: failed to set __main__.__loader__\n"
);
            ret = -1;
            goto done;
        }
        //执行脚本文件
        v = PyRun_FileExFlags(fp, filename, Py_file_input, d, d,
                              closeit, flags);
    }
    //.......
}


PyObject *
PyRun_FileExFlags(FILE *fp, const char *filename_str, int start, PyObjec
t *globals,
                  PyObject *locals, int closeit, PyCompilerFlags *flags)
{
    PyObject *ret = NULL;
    //......
    //编译
    mod = PyParser_ASTFromFileObject(fp, filename, NULL, start, 0, 0,
```

```
                                    flags, NULL, arena);
    if (closeit)
        fclose(fp);
    if (mod == NULL) {
        goto exit;
    }
    //执行, 依旧是调用了runmod
    ret = run_mod(mod, filename, globals, locals, flags, arena);

exit:
    Py_XDECREF(filename);
    if (arena != NULL)
        PyArena_Free(arena);
    return ret;
}
```

很显然，脚本文件和交互式之间的执行流程是不同的，但最终都进入了 run_mod，而且同样将 __main__ 中维护的 PyDictObject对象作为 local名字空间和 global名字空间传入了 run_mod。

启动虚拟机

前面的都是准备工作，到这里才算是真正开始启动虚拟机。

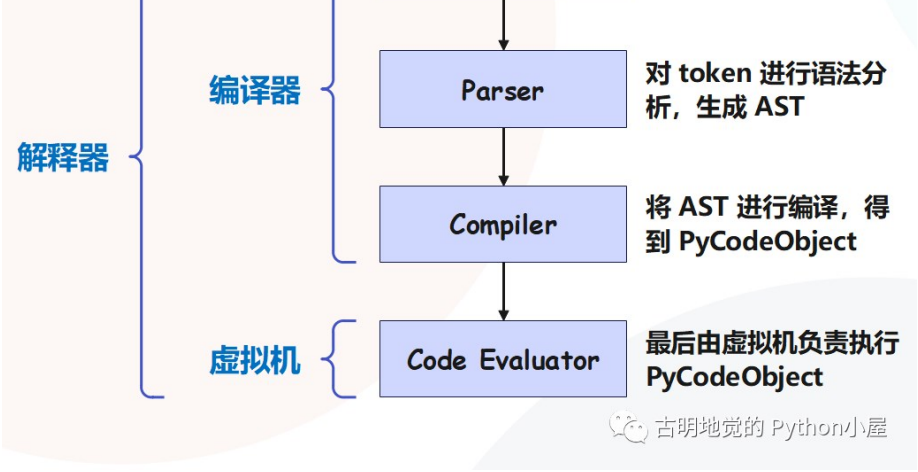
```
static PyObject *
run_mod(mod_ty mod, PyObject *filename, PyObject *globals, PyObject *loc
als,
            PyCompilerFlags *flags, PyArena *arena)
{
    PyCodeObject *co;
    PyObject *v;
    //基于ast编译字节码指令序列，创建PyCodeObject对象
    co = PyAST_CompileObject(mod, filename, flags, -1, arena);
    if (co == NULL)
        return NULL;

    if (PySys_Audit("exec", "O", co) < 0) {
        Py_DECREF(co);
        return NULL;
    }

    //创建PyFrameObject，执行PyCodeObject对象中的字节码指令序列
    v = run_eval_code_obj(co, globals, locals);
    Py_DECREF(co);
    return v;
}
```

run_mod 接手传来的 ast，然后再传到 PyAST_CompileObject 中，创建了一个我们已经非常熟悉的 PyCodeObject对象。

关于这个完整的编译过程，就又是另一个话题了，总之先是 Scanner 进行词法分析、将源代码切分成一个个的 token，然后 Parser 在词法分析的结果之上进行语法分析、根据切分好的 token 生成抽象语法树(AST，abstract syntax tree)，然后将 AST 编译 PyCodeObject 对象，最后再由虚拟机执行。



整个流程就是这样，至于到底是怎么分词、怎么建立语法树的，这就涉及到编译原理了，个人觉得甚至比研究Python虚拟机还难。有兴趣的话可以去看源码中的 Parser 目录，如果能把 Python 的分词、语法树的建立给了解清楚，那我觉得你完全可以手写一个正则表达式的引擎、以及各种模板语言。

此时，Python 已经做好一切工作，于是开始通过 run_eval_code_obj 着手唤醒虚拟机。

```
static PyObject *
run_eval_code_obj(PyCodeObject *co, PyObject *globals, PyObject *locals)
{
    PyObject *v;
    //......
    v = PyEval_EvalCode((PyObject*)co, globals, locals);
    if (!v && PyErr_Occurred() == PyExc_KeyboardInterrupt) {
        _Py_UnhandledKeyboardInterrupt = 1;
    }
    return v;
}
```

函数中调用了 PyEval_EvalCode，根据前面的介绍，我们知道最终一定会走到 PyEval_EvalFrameEx。最终调用 _PyEval_EvalFrameDefault，然后进入那个拥有巨型 switch 的 for 循环，不停地执行字节码指令，而运行时栈就是参数的容身之所。

所以整个流程就是先创建进程，进程创建线程，设置 builtins（包括设置__name__、内建对象、内置函数方法等等）、设置缓存池，然后各种初始化，设置搜索路径。最后分词、编译、激活虚拟机执行。

而执行方式就是调用曾经与我们朝夕相处的 PyEval_EvalFrameEx ，掌控 Python 世界中无数对象的生生灭灭。参数 f 就是 PyFrameObject 对象，我们曾经探索了很久，现在一下子就回到了当初，有种梦回栈帧对象的感觉。

目前的话，Python 的骨架我们已经看清了，虽然还有很多细节隐藏在幕后，至少神秘的面纱已经被撤掉了。

小结

当我们在控制台输入 python 的那一刻，背后真的是做了大量的工作。因为Python是动态语言，很多操作都要发生在运行时。

关于运行时环境的初始化和虚拟机的启动就说到这里，接下来我们就要介绍 Python 的多线程了，以及被称为万恶之源的 GIL。

收录于合集 #CPython 97

上一篇
《源码探秘 CPython》84. 初识GIL、以及多个线程之间的调度机制

下一篇
《源码探秘 CPython》82. Python运行时环境的初始化，解释器在启动时都做了什么？

喜欢此内容的人还喜欢

C++使用消息队列实现进程间通信
控制工程研习

浅谈Kotlin协程及首页弹窗中的应用
洋钱罐技术团队

JVM由浅入深最全教程(一文学完系列)
编程攻略