《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？

原创 古明地觉 古明地觉的编程教室 2022-05-07 08:30 发表于北京

楔子

我们之前完成了 Python 的字节码、以及虚拟机的剖析工作，但这仅仅只是一部分，而其余的部分则被遮在了幕后。记得我们在分析虚拟机的时候，曾这么说过：

解释器启动时，首先会进行 "运行时环境" 的初始化，关于 "运行时环境" 的初始化是一个非常复杂的过程。并且 "运行时环境" 和 "执行环境" 是不同的， "运行时环境" 是一个全局的概念，而 "执行环境" 是一个栈帧。关于 "运行时环境" 后面会单独分析，这里就假设初始化动作已经完成，我们已经站在了虚拟机的门槛外面，只需要轻轻推动第一张骨牌，整个执行过程就会像多米诺骨牌一样，一环扣一环地展开。

所以这次，我们将回到时间的起点，从 Python 的应用程序被执行开始，一步一步紧紧跟随 Python 的轨迹，完整地展示解释器在启动之初的所有动作。当我们了解所有的初始化动作之后，也就能对 Python 执行引擎执行字节码指令时的整个运行环境了如指掌了。

线程模型

我们知道线程是操作系统调度的最小单元，那么 Python 的线程又是怎么样的呢？

启动一个 Python 线程，底层会启动一个 C 线程，然后启动操作系统的一个原生线程（OS 线程）。所以 Python 的线程实际上是对 **OS 线程**的一个封装，因此 Python 的线程是货真价实的。

然后 Python 还提供了一个 PyThreadState 对象，也就是线程状态对象，维护 OS 线程执行的状态信息，相当于是 OS 线程的一个抽象描述。

虽然真正用来执行的线程及其状态肯定是由操作系统进行维护的，但是 Python 虚拟机在运行的时候总需要另外一些与线程相关的状态和信息，比如是否发生了异常等等，这些信息显然操作系统是没有办法提供的。

而 PyThreadState 对象正是为 OS 线程准备的、在虚拟机层面保存其状态信息的对象，也就是线程状态对象。在 Python 中，当前活动的 OS 线程对应的 PyThreadState 对象可以通过 PyThreadState_GET 获得，有了线程状态对象之后，就可以设置一些额外信息了。具体内容，我们后面会说。

当然除了线程状态对象之外，还有进程状态对象，我们来看看两者在底层的定义是什么？它们位于 Include/pystate.h 中。

```
// ts 是 thread state 的简写
typedef struct _ts PyThreadState;
// is 是 interpreter state 的简写
typedef struct _is PyInterpreterState;
```

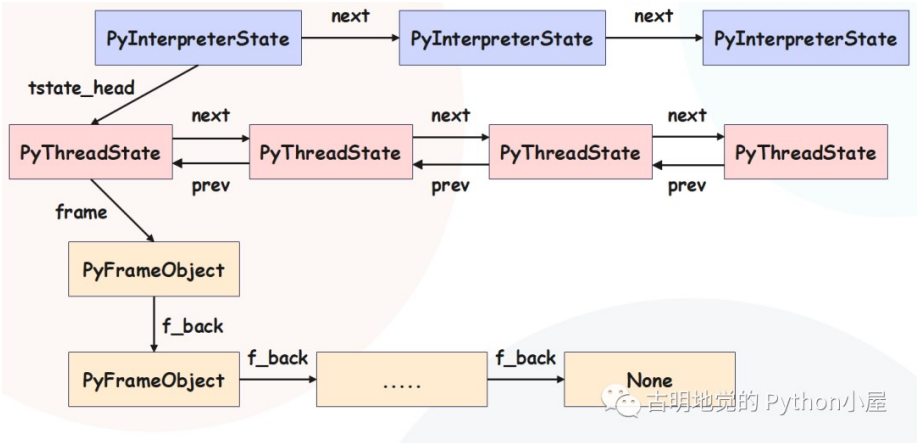
里面的 PyThreadState 表示线程状态对象，PyInterpreterState 表示进程状态对象，但它们都是 typedef 起的一个别名。前者是 struct _ts 的别名，后者是 struct _is 的别名，我们来看一下它们长什么样。

```c
// Include/cpython/pystate.h
struct _ts {
    //多个线程状态对象也会像链表一样串起来
    //因为一个进程里面是可以包含多个线程的
    //prev 指向上一个线程状态对象
    //next 指向下一个线程状态对象
    struct _ts *prev;
    struct _ts *next;
    //进程状态对象，标识该线程属于哪一个进程
    PyInterpreterState *interp;

    //栈帧对象，模拟线程中函数的调用堆栈
    struct _frame *frame;
    //递归深度
    int recursion_depth;
    //.....
    //线程 id
    uint64_t id;
};
```

```c
//Include/internal/pycore_pystate.h
struct _is {
    //Python支持多进程，多个进程也会以链表的形式进行组织
    //当前进程的下一个进程
    struct _is *next;
    //进程环境中的线程状态对象的集合
    //我们说线程状态对象会形成一个链表
    //这里就是链表的头结点
    struct _ts *tstate_head;
    //进程id
    int64_t id;
    //....
    PyObject *audit_hooks;
};
```

所以 PyInterpreterState 对象可以看成是对进程的模拟， PyThreadState 是对线程的模拟。我们之前分析虚拟机的时候说过其执行环境，如果再将运行时环境加进去的话。



进程状态对象的 tstate_head 指向了线程状态对象，对应当前活跃的 Python 线程；每个线程状态对象的 frame 都指向当前正在执行的栈帧对象。

线程环境的初始化

在解释器启动之后，初始化的动作是从 Py_NewInterpreter 函数开始的，然后这个函数调用了 new_interpreter 函数完成初始化。至于这两个函数长什么样一会再聊，先往后看。

我们知道当操作系统在运行一个可执行文件时，首先创建一个进程内核。同理在 Python 中亦是如此，会在 new_interpreter 中调用 PyInterpreterState_New 创建一个崭新的 PyInterpreterState对象。该函数位于 Python/pystate.c 中。

```
PyInterpreterState *
PyInterpreterState_New(void)
{
    //申请进程状态对象所需要的内存
    PyInterpreterState *interp = PyMem_RawMalloc(sizeof(PyInterpreterSta
te));
    if (interp == NULL) {
        return NULL;
    }

    //设置属性
    //......
    //......
    return interp;
}
```

关于进程状态对象我们不做过多解释，只需要知道解释器在启动时，会创建一个 PyInterpreterState 对象。如果开启了多进程，那么内部会继续创建，然后通过 next 指针将多个 PyInterpreterState 串成一个链表结构。

在调用 PyInterpreterState_New 成功创建 PyInterpreterState 之后，会再接再厉，调用 PyThreadState_New 创建一个全新的线程状态对象，相关函数定义同样位于 Python/pystate.c 中。

```
PyThreadState *
PyThreadState_New(PyInterpreterState *interp)
{
    return new_threadstate(interp, 1);
}
```

我们注意到这个函数接收一个 PyInterpreterState，这说明了线程是依赖于进程的，因为需要进程给自己分配资源，然后这个函数又调用了 new_threadstate。除了传递 PyInterpreterState 之外，还传了一个 1，想也不用想这肯定是创建的线程数量。这里创建 1 个，也就是主线程（main thread）。

```
static PyThreadState *
new_threadstate(PyInterpreterState *interp, int init)
{
    _PyRuntimeState *runtime = &_PyRuntime;
    //为线程状态对象申请内存
    PyThreadState *tstate = (PyThreadState *)PyMem_RawMalloc(sizeof(PyTh
readState));
    if (tstate == NULL) {
        return NULL;
    }
    //设置从线程中获取函数调用栈的操作
    if (_PyThreadState_GetFrame == NULL) {
        _PyThreadState_GetFrame = threadstate_getframe;
    }

    //设置该线程所在的进程
    tstate->interp = interp;

    //下面就是设置内部的成员属性
    //比如栈帧、递归深度、线程 id
```

```
    tstate->frame = NULL;
    tstate->recursion_depth = 0;
    tstate->id = ++interp->tstate_next_unique_id;
    //......
    //......
    //......
    //上一个线程状态对象
    tstate->prev = NULL;
    //当前线程状态对象的 next
    //我们看到指向了线程状态对象链表的头结点
    tstate->next = interp->tstate_head;
    if (tstate->next)
        //因为每个线程状态对象的prev指针都要指向上一个线程状态对象
        //如果是头结点的话，那么prev就是 NULL
        //但由于新的线程状态对象在插入之后显然就变成了链表的新的头结点
        //因此还需要将插入之前的头结点的prev指向新插入的线程状态对象
        tstate->next->prev = tstate;
    //将tstate设置为新的线程状态对象(链表的头结点)
    interp->tstate_head = tstate;

    //返回线程状态对象
    return tstate;
}
```

和 PyInterpreterState_New 相同， PyThreadState_New 会申请内存，创建线程状态对象，并且对每个成员进行初始化。其中 prev 指针和 next 指针分别指向了上一个线程状态对象和下一个线程状态对象。

而且也肯定会存在某一时刻，存在多个 PyThreadState 对象形成一个链表，那么什么时刻会发生这种情况呢？显然用鼻子想也知道这是在启动多线程的时候。

> 此外我们看到 Python 在插入线程状态对象的时候采用的是头插法。

从源码中我们看到，虚拟机设置了从线程中获取函数调用栈的操作，所谓函数调用栈就是前面说的 PyFrameObject对象链表。而且在源码中，PyThreadState 关联了 PyInterpreterState ，PyInterpreterState 也关联了 PyInterpreterState 。

到目前为止，仅有的两个对象建立起了联系。对应到操作系统，我们说进程和线程建立了联系。

而在两者建立了联系之后，那么就很容易在 PyInterpreterState 和 PyThreadState 之间穿梭。并且在 Python 运行时环境中，会有一个变量（先卖个关子）一直维护着当前活动的线程，更准确的说是当前活动线程（OS 线程）对应的 PyThreadState 对象。

初始时，该变量为 NULL。在 Python 启动之后创建了第一个 PyThreadState 之后，会调用 PyThreadState_Swap 函数来设置这个变量，函数位于 Python/pystate.c 中。

```
PyThreadState *
PyThreadState_Swap(PyThreadState *newts)
{
    //调用了_PyThreadState_Swap，里面传入了两个参数
    //第一个我们后面说，从名字上看显然这是和GIL相关的
    //第二个参数就是新创建的线程状态对象
    return _PyThreadState_Swap(&_PyRuntime.gilstate, newts);
}
```

内部又调用了 _PyThreadState_Swap。

```
PyThreadState *
_PyThreadState_Swap(struct _gilstate_runtime_state *gilstate, PyThreadSt
ate *newts)
{
    //这里是获取当前的线程状态对象
    //并且保证线程的安全性
    PyThreadState *oldts = _PyRuntimeGILState_GetThreadState(gilstate);
```

```
    //将 GIL 交给 newts，也就是新创建、即将获取执行权的线程状态对象
    _PyRuntimeGILState_SetThreadState(gilstate, newts);
    //....
    return oldts;
}
```

所以逻辑很容易理解，有一个**变量**始终维护着当前活跃线程对应的线程状态对象，初始时它是个 NULL。而一旦解释器启动，并创建了第一个线程状态对象（显然对应主线程），那么就会将创建的线程状态对象交给这个**变量**保存。

如果调用 _PyThreadState_Swap 的时候，发现保存线程状态对象的**变量**不为 NULL，那么说明开启了多线程。**变量**保存的就是代码中的 oldts，也就是当前活动线程对应的线程状态对象，可由于它的时间片耗尽，解释器会剥夺它的执行权，然后交给 newts。那么 newts 就成为了新的当前活跃线程对应的线程状态对象，那么它也要交给**变量**进行保存。

而通过 _PyThreadState_Swap 可以看到，想要实现这一点，主要依赖两个宏。

```
// 通过&(gilstate)->tstate_current获取当前线程
#define _PyRuntimeGILState_GetThreadState(gilstate) \
    ((PyThreadState*)_Py_atomic_load_relaxed(&(gilstate)->tstate_current))

// 将newts设置为当前线程，可以理解为发生了线程的切换
#define _PyRuntimeGILState_SetThreadState(gilstate, value) \
    _Py_atomic_store_relaxed(&(gilstate)->tstate_current, \
                             (uintptr_t)(value))
```


然后这两个宏里面出现了 _Py_atomic_load_relaxed, _Py_atomic_store_relaxed 和 &(gilstate)->tstate_current ，这些又是什么呢？还有到底哪个变量在维护着当前活动线程对应的线程状态对象呢？其实那两个宏已经告诉你了。

```
//Include/internal/pycore_pystate.h
struct _gilstate_runtime_state {
    //...
    //宏里面出现的 gilstate 就是该结构体实例
    //tstate_current指的就是当前活动的 OS 线程对应的状态对象
    //同时也可以理解为获取到 GIL 的 Python线程
    _Py_atomic_address tstate_current;
    //...
};


//Include/internal/pycore_atomic.h
#define _Py_atomic_load_relaxed(ATOMIC_VAL) \
    _Py_atomic_load_explicit((ATOMIC_VAL), _Py_memory_order_relaxed)

#define _Py_atomic_store_relaxed(ATOMIC_VAL, NEW_VAL) \
    _Py_atomic_store_explicit((ATOMIC_VAL), (NEW_VAL), _Py_memory_order_
relaxed)

#define _Py_atomic_load_explicit(ATOMIC_VAL, ORDER) \
    atomic_load_explicit(&((ATOMIC_VAL)->_value), ORDER)

#define _Py_atomic_store_explicit(ATOMIC_VAL, NEW_VAL, ORDER) \
    atomic_store_explicit(&((ATOMIC_VAL)->_value), NEW_VAL, ORDER)
```

不难发现：

- **_Py_atomic_load_relaxed 调用了 _Py_atomic_load_explicit，_Py_atomic_load_explicit 又调用了 atomic_load_explicit；**
- **_Py_atomic_store_relaxed 调用了 _Py_atomic_store_explicit，_Py_atomic_store_explicit 调用了 atomic_store_explicit；**

而atomic_load_explicit和atomic_store_explicit 是系统头文件 stdatomic.h 中定义的 api，这是在系统的 api 中修改的，所以说是线程安全的。

介绍完中间部分的内容，那么我们可以从头开始分析 Python 运行时的初始化了，我们说它是从 Py_NewInterpreter 开始的。

```
// Python/pylifecycle.c
PyThreadState *
Py_NewInterpreter(void)
{
    //线程状态对象
    PyThreadState *tstate = NULL;
    //传入线程对象，调用 new_interpreter
    PyStatus status = new_interpreter(&tstate);
    //异常检测
    if (_PyStatus_EXCEPTION(status)) {
        Py_ExitStatusException(status);
    }
    //返回线程状态对象
    return tstate;
}
```

另外里面出现了一个 PyStatus， 表示程序执行的状态， 会检测是否发生了异常。

```
//Include/cpython/initconfig.h
typedef struct {
    enum {
        _PyStatus_TYPE_OK=0,
        _PyStatus_TYPE_ERROR=1,
        _PyStatus_TYPE_EXIT=2
    } _type;
    const char *func;
    const char *err_msg;
    int exitcode;
} PyStatus;
```

然后我们的重点是 new_interpreter函数，进程状态对象的创建就是在这个函数里面发生的。

```
//Python/pylifecycle.c
static PyStatus
new_interpreter(PyThreadState **tstate_p)
{
    PyStatus status;

    //运行时初始化，如果出现异常直接返回
    status = _PyRuntime_Initialize();
    if (_PyStatus_EXCEPTION(status)) {
        return status;
    }
    //......
    // 创建一个进程状态对象
    PyInterpreterState *interp = PyInterpreterState_New();
    //......
    //根据进程状态对象创建一个线程状态对象
    //维护对应 OS 线程的状态
    PyThreadState *tstate = PyThreadState_New(interp);
    //将GIL的控制权交给创建的线程
    PyThreadState *save_tstate = PyThreadState_Swap(tstate);
    //...
}
```

Python在初始化运行时环境时，肯定也要对类型系统进行初始化等等，整体是一个非常庞大的过程。

到这里，我们对 new_interpreter 算是有了一个阶段性的成功，我们创建了代表进程和线程概念

的 PyInterpreterState 和 PyThreadState 对象，并且在它们之间建立的联系。下面，new_interpreter 将进行入另一个环节，设置系统 module。

在 new_interpreter 中创建了 PyInterpreterState 和 PyThreadState 对象之后，就会开始设置系统的 __builtins__ 了。

```
static PyStatus
new_interpreter(PyThreadState **tstate_p)
{
    //....
    //申请一个PyDictObject对象，用于 sys.modules
    PyObject *modules = PyDict_New();
    if (modules == NULL) {
        return _PyStatus_ERR("can't make modules dictionary");
    }

    //然后让 interp -> modules 维护 modules
    //由于 interp 表示的是进程状态对象，这说明什么？
    //显然是该进程内的多个线程共享同一个 sys.modules
    interp->modules = modules;

    //加载sys模块，所有的module对象都在sys.modules中
    PyObject *sysmod = _PyImport_FindBuiltin("sys", modules);
    if (sysmod != NULL) {
        interp->sysdict = PyModule_GetDict(sysmod);
        if (interp->sysdict == NULL) {
            goto handle_error;
        }
        Py_INCREF(interp->sysdict);
        PyDict_SetItemString(interp->sysdict, "modules", modules);
        if (_PySys_InitMain(runtime, interp) < 0) {
            return _PyStatus_ERR("can't finish initializing sys");
        }
    }

    //加载内置模块 builtins
    //可以import builtins，并且builtins.list等价于list
    PyObject *bimod = _PyImport_FindBuiltin("builtins", modules);
    if (bimod != NULL) {
        //设置 __builtins__
        interp->builtins = PyModule_GetDict(bimod);
        if (interp->builtins == NULL)
            goto handle_error;
        Py_INCREF(interp->builtins);
    }
    //......
}
```

整体还是比较清晰和直观的，另外我们说内置名字空间是由进程来维护的，因为进程就是用来为线程提供资源的。但是也能看出，一个进程内的多个线程共享同一个内置作用域。

显然这是非常合理的，不可能每开启一个线程，就为其创建一个 __builtins__。我们来从Python的角度证明这一点：

```
import threading
import builtins

```

```python
def foo1():
    builtins.list, builtins.tuple = builtins.tuple, builtins.list

def foo2():
    print(f"猜猜下面代码会输出什么:")
    print("list:", list([1, 2, 3, 4, 5]))
    print("tuple:", tuple([1, 2, 3, 4, 5]))

f1 = threading.Thread(target=foo1)
f1.start()
f1.join()
threading.Thread(target=foo2).start()
"""
猜猜下面代码会输出什么:
list: (1, 2, 3, 4, 5)
tuple: [1, 2, 3, 4, 5]
"""
```

所有的内置对象和内置函数都在内置名字空间里面，可以通过 import builtins获取、也可以直接通过 __builtins__ 这个变量来获取。

我们在foo1中把list和tuple互换了，而这个结果显然也影响到了foo2函数。这也说明了__builtins__是属于进程级别的，它是被多个线程共享的，所以是**interp -> modules = modules**。当然这个modules是sys.modules，因为不止内置名字空间，所有的module对象都是被多个线程共享的。

而对__builts__的初始化是在 _PyBuiltin_Init 函数中进行的。

```c
//Python/bltinmodule.c
PyObject *
_PyBuiltin_Init(void)
{
    PyObject *mod, *dict, *debug;

    const PyConfig *config = &_PyInterpreterState_GET_UNSAFE()->config;

    if (PyType_Ready(&PyFilter_Type) < 0 ||
        PyType_Ready(&PyMap_Type) < 0 ||
        PyType_Ready(&PyZip_Type) < 0)
        return NULL;

    //创建并设置 __builtins__
    mod = _PyModule_CreateInitialized(&builtinsmodule, PYTHON_API_VERSIO
N);
    if (mod == NULL)
        return NULL;
    //拿到 __builtins__ 的属性字典
    dict = PyModule_GetDict(mod);
    //将所有内置对象加入到 __builtins__ 中
    //......
    //老铁们，下面这些东西应该不陌生吧
    SETBUILTIN("None",                  Py_None);
    SETBUILTIN("Ellipsis",              Py_Ellipsis);
    SETBUILTIN("NotImplemented",        Py_NotImplemented);
    SETBUILTIN("False",                 Py_False);
    SETBUILTIN("True",                  Py_True);
    SETBUILTIN("bool",                  &PyBool_Type);
    SETBUILTIN("memoryview",        &PyMemoryView_Type);
    SETBUILTIN("bytearray",             &PyByteArray_Type);
    SETBUILTIN("bytes",                 &PyBytes_Type);
    SETBUILTIN("classmethod",           &PyClassMethod_Type);
    SETBUILTIN("complex",               &PyComplex_Type);
    SETBUILTIN("dict",                  &PyDict_Type);
```

```
    SETBUILTIN("enumerate",             &PyEnum_Type);
    SETBUILTIN("filter",                &PyFilter_Type);
    SETBUILTIN("float",                 &PyFloat_Type);
    SETBUILTIN("frozenset",             &PyFrozenSet_Type);
    SETBUILTIN("property",              &PyProperty_Type);
    SETBUILTIN("int",                   &PyLong_Type);
    SETBUILTIN("list",                  &PyList_Type);
    SETBUILTIN("map",                   &PyMap_Type);
    SETBUILTIN("object",                &PyBaseObject_Type);
    SETBUILTIN("range",                 &PyRange_Type);
    SETBUILTIN("reversed",              &PyReversed_Type);
    SETBUILTIN("set",                   &PySet_Type);
    SETBUILTIN("slice",                 &PySlice_Type);
    SETBUILTIN("staticmethod",          &PyStaticMethod_Type);
    SETBUILTIN("str",                   &PyUnicode_Type);
    SETBUILTIN("super",                 &PySuper_Type);
    SETBUILTIN("tuple",                 &PyTuple_Type);
    SETBUILTIN("type",                  &PyType_Type);
    SETBUILTIN("zip",                   &PyZip_Type);
    debug = PyBool_FromLong(config->optimization_level == 0);
    if (PyDict_SetItemString(dict, "__debug__", debug) < 0) {
        Py_DECREF(debug);
        return NULL;
    }
    Py_DECREF(debug);

    return mod;
#undef ADD_TO_ALL
#undef SETBUILTIN
}
```

整个 _PyBuiltin__Init 函数的功能就是设置好__builtins__ module，而这个过程是分为两步的。

- **通过 _PyModule_CreateInitialized 函数创建 PyModuleObject 对象，我们知道这是Python模块对象的底层实现；**
- **设置 module，将Python的内置对象都塞到__builtins__中；**

但是我们看到设置的东西似乎少了不少，比如 dir、hasattr、setatt r等等，这些明显也是内置的，但是它们到哪里去了。别急，我们刚才说创建__builtins__分为两步，第一步是创建 PyModuleObject，而使用的函数就是 _PyModule_CreateInitialized ，而在这个函数里面就已经完成了大部分的__builtins__设置工作。

```
//Object/moduleobject.c
PyObject *
_PyModule_CreateInitialized(struct PyModuleDef* module, int module_api_v
ersion)
{
    const char* name;
    PyModuleObject *m;

    //初始化
    if (!PyModuleDef_Init(module))
        return NULL;
    //拿到 module 的 name
    //对于当前来说就是__builtins__
    name = module->m_name;
    //这里比较有意思，这是检测模块版本的，针对的是需要导入的py文件。
    //我们说编译成PyCodeObject对象之后
    //会直接从当前目录的__pycache__里面导入，那里面都是pyc文件
    //而pyc文件的文件名是有Python解释器的版本号的
    //这里就是比较版本是否一致，不一致则不导入pyc文件，而是会重新编译py文件
    if (!check_api_version(name, module_api_version)) {
        return NULL;
    }
```

```
    if (module->m_slots) {
        PyErr_Format(
            PyExc_SystemError,
            "module %s: PyModule_Create is incompatible with m_slots", n
ame);
        return NULL;
    }
    //创建一个PyModuleObject
    if ((m = (PyModuleObject*)PyModule_New(name)) == NULL)
        return NULL;

    //.......
    if (module->m_methods != NULL) {
        //遍历methods中指定的module对象中应包含的操作集合
        if (PyModule_AddFunctions((PyObject *) m, module->m_methods) != 0
) {
            Py_DECREF(m);
            return NULL;
        }
    }
    if (module->m_doc != NULL) {
        //设置docstring
        if (PyModule_SetDocString((PyObject *) m, module->m_doc) != 0) {
            Py_DECREF(m);
            return NULL;
        }
    }
    m->md_def = module;
    return (PyObject*)m;
}
```

根据上面的代码我们可以得出如下信息：

- **1）name：module 对象的名称，在这里就是 __builtins__；**
- **2）module_api_version：Python 内部使用的version值，用于比较；**
- **3）PyModule_New：用于创建一个PyModuleObject对象；**
- **4）methods：该module中所包含的函数的集合，在这里是builtin_methods；**
- **5）PyModule_AddFunctions：设置methods中的函数操作；**
- **6）PyModule_SetDocString：设置docstring；**

Python 的 module对象在底层对应的结构体是 PyModuleObject对象，我们来看看它长什么样子吧。

```
//Objects/moduleobject.c
typedef struct {
    //头部信息
    PyObject_HEAD
    //属性字典，所有的属性和值都在里面
    PyObject *md_dict;
     //module对象包含的操作集合，里面是一些结构体
     //每个结构体对应一个函数的相关信息
    struct PyModuleDef *md_def;
    //...
    PyObject *md_name;  //模块名
} PyModuleObject;
```

而这个对象是通过PyModule_New创建的。

```
PyObject *
PyModule_New(const char *name)
{
```

```c
    //module对象的 name、PyModuleObject *
    PyObject *nameobj, *module;
    nameobj = PyUnicode_FromString(name);
    if (nameobj == NULL)
        return NULL;
    //创建 PyModuleObject
    module = PyModule_NewObject(nameobj);
    Py_DECREF(nameobj);
    return module;
}


PyObject *
PyModule_NewObject(PyObject *name)
{
    //module对象的指针
    PyModuleObject *m;
    //为 module 对象申请空间
    //模块具有属性字典，所以是一个变长对象
    m = PyObject_GC_New(PyModuleObject, &PyModule_Type);
    if (m == NULL)
        return NULL;
    //设置相应属性，初始化为NULL
    m->md_def = NULL;
    m->md_state = NULL;
    m->md_weaklist = NULL;
    m->md_name = NULL;
    //属性字典
    m->md_dict = PyDict_New();
    //调用module_init_dict
    if (module_init_dict(m, m->md_dict, name, NULL) != 0)
        goto fail;
    PyObject_GC_Track(m);
    return (PyObject *)m;

 fail:
    Py_DECREF(m);
    return NULL;
}

static int
module_init_dict(PyModuleObject *mod, PyObject *md_dict,
                 PyObject *name, PyObject *doc)
{
    _Py_IDENTIFIER(__name__);
    _Py_IDENTIFIER(__doc__);
    _Py_IDENTIFIER(__package__);
    _Py_IDENTIFIER(__loader__);
    _Py_IDENTIFIER(__spec__);

    if (md_dict == NULL)
        return -1;
    if (doc == NULL)
        doc = Py_None;
    //模块的一些属性、__name__、__doc__等等
    if (_PyDict_SetItemId(md_dict, &PyId___name__, name) != 0)
        return -1;
    if (_PyDict_SetItemId(md_dict, &PyId___doc__, doc) != 0)
        return -1;
    if (_PyDict_SetItemId(md_dict, &PyId___package__, Py_None) != 0)
        return -1;
    if (_PyDict_SetItemId(md_dict, &PyId___loader__, Py_None) != 0)
        return -1;
    if (_PyDict_SetItemId(md_dict, &PyId___spec__, Py_None) != 0)
```

```
68          return -1;
69      if (PyUnicode_CheckExact(name)) {
70          Py_INCREF(name);
71          Py_XSETREF(mod->md_name, name);
72      }
73
74      return 0;
75  }
```

这里虽然创建了一个 module对象，但是这仅仅是一个空的 module对象，却并没有包含相应的操作和数据。我们看到只设置了 name 和 doc 等属性。

在 PyModule_New 结束之后，程序继续执行 _PyModule_CreateInitialized 内部的代码，然后通过 PyModule_AddFunctions 完成了对__builtins__大部分属性的设置。这个设置的属性依赖于第二个参数methods，在这里为builtin_methods。然后会遍历builtin_methods，并处理每一项元素，我们还是来看看长什么样子。

```
1  //Python/bltinmodule.c
2
3  static PyMethodDef builtin_methods[] = {
4      {"__build_class__", (PyCFunction)(void(*)(void))builtin___build_clas
5  s__,
6       METH_FASTCALL | METH_KEYWORDS, build_class_doc},
7      {"__import__",      (PyCFunction)(void(*)(void))builtin___import__,
8  METH_VARARGS | METH_KEYWORDS, import_doc},
9      BUILTIN_ABS_METHODDEF
10     BUILTIN_ALL_METHODDEF
11     BUILTIN_ANY_METHODDEF
12     BUILTIN_ASCII_METHODDEF
13     BUILTIN_BIN_METHODDEF
14     {"breakpoint",      (PyCFunction)(void(*)(void))builtin_breakpoint,
15 METH_FASTCALL | METH_KEYWORDS, breakpoint_doc},
16     BUILTIN_CALLABLE_METHODDEF
17     BUILTIN_CHR_METHODDEF
18     BUILTIN_COMPILE_METHODDEF
19     BUILTIN_DELATTR_METHODDEF
20     {"dir",             builtin_dir,        METH_VARARGS, dir_doc},
21     BUILTIN_DIVMOD_METHODDEF
22     BUILTIN_EVAL_METHODDEF
23     BUILTIN_EXEC_METHODDEF
24     BUILTIN_FORMAT_METHODDEF
25     {"getattr",         (PyCFunction)(void(*)(void))builtin_getattr, MET
26 H_FASTCALL, getattr_doc},
27     BUILTIN_GLOBALS_METHODDEF
28     BUILTIN_HASATTR_METHODDEF
29     BUILTIN_HASH_METHODDEF
30     BUILTIN_HEX_METHODDEF
31     BUILTIN_ID_METHODDEF
32     BUILTIN_INPUT_METHODDEF
33     BUILTIN_ISINSTANCE_METHODDEF
34     BUILTIN_ISSUBCLASS_METHODDEF
35     {"iter",            (PyCFunction)(void(*)(void))builtin_iter,
36 METH_FASTCALL, iter_doc},
37     BUILTIN_LEN_METHODDEF
38     BUILTIN_LOCALS_METHODDEF
39     {"max",             (PyCFunction)(void(*)(void))builtin_max,
40 METH_VARARGS | METH_KEYWORDS, max_doc},
41     {"min",             (PyCFunction)(void(*)(void))builtin_min,
42 METH_VARARGS | METH_KEYWORDS, min_doc},
43     {"next",            (PyCFunction)(void(*)(void))builtin_next,
44 METH_FASTCALL, next_doc},
```

```
    BUILTIN_OCT_METHODDEF
    BUILTIN_ORD_METHODDEF
    BUILTIN_POW_METHODDEF
    {"print",           (PyCFunction)(void(*)(void))builtin_print,
METH_FASTCALL | METH_KEYWORDS, print_doc},
    BUILTIN_REPR_METHODDEF
    BUILTIN_ROUND_METHODDEF
    BUILTIN_SETATTR_METHODDEF
    BUILTIN_SORTED_METHODDEF
    BUILTIN_SUM_METHODDEF
    {"vars",            builtin_vars,       METH_VARARGS, vars_doc},
    {NULL,              NULL},
};
```

怎么样，是不是看到了玄机。

总结一下就是：在 Py_NewInterpreter 里面调用 new_interpreter 函数，然后在 new_interpreter 这个函数里面，通过 PyInterpreterState_New 创建 PyInterpreterState，然后传递 PyInterpreterState 调用 PyThreadState_New 创建 PyThreadState。

接着就是执行各种初始化动作，然后在 new_interpreter 中调用 _PyBuiltin_Init 设置内置属性，在代码的最后会设置内置属性(函数、对象)。但是有很多却不在里面，比如：dir、getattr等等。

所以中间调用的 _PyModule_CreateInitialized 不仅仅是初始化一个module对象，还会在初始化之后将我们没有看到的一些属性设置进去。在里面先使用 PyModule_New 创建一个 PyModuleObject，此时它内部只有 __name__ 和 __doc__ 等属性，之后再通过 PyModule_AddFunctions 设置 methods，在这里面我们看到了 dir、getattr 等内置函数。当这些属性设置完之后，退回到 _PyBuiltin_Init 函数中，再设置剩余的部分属性。之后，__builtins__ 就完成了。

另外 builtin_methods 是一个 PyMethodDef 类型的数组，里面是一个个的 PyMethodDef 结构体。

```
//Include/methodobject.h
struct PyMethodDef {
    /* 内置函数的名称 */
    const char  *ml_name;
    //实现对应逻辑的 C 函数，但是需要转成 PyCFunction类型
    //主要是为了更好的处理关键字参数
    PyCFunction ml_meth;

    /* 参数类型
    #define METH_VARARGS  0x0001  扩展位置参数
    #define METH_KEYWORDS 0x0002  扩展关键字参数
    #define METH_NOARGS   0x0004  不需要参数
    #define METH_O        0x0008  需要一个参数
    #define METH_CLASS    0x0010  被classmethod装饰
    #define METH_STATIC   0x0020  被staticmethod装饰
    */
    int         ml_flags;

    //函数的__doc__
    const char  *ml_doc;
};
typedef struct PyMethodDef PyMethodDef;
```

对于这里面每一个 PyMethodDef，_PyModule_CreateInitialized 都会基于它创建一个 PyCFunctionObject 对象。我们说过内置函数在底层对应 PyCFunctionObject。

```
typedef struct {
    //头部信息
    PyObject_HEAD
    //PyMethodDef
```

```
    PyMethodDef *m_ml;
    //self参数
    PyObject    *m_self;
    //__module__属性
    PyObject    *m_module;
    //弱引用列表，不讨论
    PyObject    *m_weakreflist;
    //为了效率而单独实现的调用函数
    vectorcallfunc vectorcall;
} PyCFunctionObject;
```

然后 PyCFunctionObject 的创建则是通过 PyCFunction_New 完成的。

```
//Objects/methodobject.c
PyObject *
PyCFunction_New(PyMethodDef *ml, PyObject *self)
{
    return PyCFunction_NewEx(ml, self, NULL);
}


PyObject *
PyCFunction_NewEx(PyMethodDef *ml, PyObject *self, PyObject *module)
{
    vectorcallfunc vectorcall;
    //判断参数类型
    switch (ml->ml_flags & (METH_VARARGS | METH_FASTCALL | METH_NOARGS |
METH_O | METH_KEYWORDS))
    {
        case METH_VARARGS:
        case METH_VARARGS | METH_KEYWORDS:
            vectorcall = NULL;
            break;
        case METH_FASTCALL:
            vectorcall = cfunction_vectorcall_FASTCALL;
            break;
        case METH_FASTCALL | METH_KEYWORDS:
            vectorcall = cfunction_vectorcall_FASTCALL_KEYWORDS;
            break;
        case METH_NOARGS:
            vectorcall = cfunction_vectorcall_NOARGS;
            break;
        case METH_O:
            vectorcall = cfunction_vectorcall_O;
            break;
        default:
            PyErr_Format(PyExc_SystemError,
                         "%s() method: bad call flags", ml->ml_name);
            return NULL;
    }

    PyCFunctionObject *op;
    //我们看到这里也采用了缓存池的策略
    op = free_list;
    if (op != NULL) {
        free_list = (PyCFunctionObject *)(op->m_self);
        (void)PyObject_INIT(op, &PyCFunction_Type);
        numfree--;
    }
    else {
        //否则重新申请
        op = PyObject_GC_New(PyCFunctionObject, &PyCFunction_Type);
        if (op == NULL)
            return NULL;
```

```
    }
    //设置属性
    op->m_weakreflist = NULL;
    op->m_ml = ml;
    Py_XINCREF(self);
    op->m_self = self;
    Py_XINCREF(module);
    op->m_module = module;
    op->vectorcall = vectorcall;
    _PyObject_GC_TRACK(op);
    return (PyObject *)op;
}
```

在 _PyBuiltin__Init 之后，虚拟机会把 PyModuleObject 对象的属性字典抽取出来，赋值给 interp -> builtins 。

```
//moduleobject.c
PyObject *
PyModule_GetDict(PyObject *m)
{
    PyObject *d;
    if (!PyModule_Check(m)) {
        PyErr_BadInternalCall();
        return NULL;
    }
    d = ((PyModuleObject *)m) -> md_dict;
    assert(d != NULL);
    return d;
}


static PyStatus
new_interpreter(PyThreadState **tstate_p)
{
    //......
    PyObject *bimod = _PyImport_FindBuiltin("builtins", modules);
    if (bimod != NULL) {
        //通过PyModule_GetDict获取属性字典，赋值给builtins
        interp->builtins = PyModule_GetDict(bimod);
        if (interp->builtins == NULL)
            goto handle_error;
        Py_INCREF(interp->builtins);
    }
    else if (PyErr_Occurred()) {
        goto handle_error;
    }
    //......
}
```

以后Python在需要访问 __builtins__ 时，直接访问 interp->builtins 就可以了，不需要再到 interp->modules 里面去找了。因为内置函数、属性的使用会很频繁，所以这种加速机制是很有效的。

创建 sys module

Python在创建并设置了__builtins__之后，会照猫画虎，用同样的流程来设置 sys module，并像

设置 interp->builtins 一样设置 interp->sysdict。

```
//Python/pylifecycle.c
static PyStatus
new_interpreter(PyThreadState **tstate_p)
{
    //.......
    PyObject *sysmod = _PyImport_FindBuiltin("sys", modules);
    if (sysmod != NULL) {
        interp->sysdict = PyModule_GetDict(sysmod);
        if (interp->sysdict == NULL) {
            goto handle_error;
        }
        Py_INCREF(interp->sysdict);
        //设置
        PyDict_SetItemString(interp->sysdict, "modules", modules);
        if (_PySys_InitMain(runtime, interp) < 0) {
            return _PyStatus_ERR("can't finish initializing sys");
        }
    }
    //.......
}
```

Python在创建了sys module之后，会在此module中设置一个Python搜索module时的默认路径集合。

```
//Python/pylifecycle.c
static PyStatus
new_interpreter(PyThreadState **tstate_p)
{
    //.......
    status = add_main_module(interp);
    //.......
}


static PyStatus
add_main_module(PyInterpreterState *interp)
{
    PyObject *m, *d, *loader, *ann_dict;
    //将__main__添加进sys.modules中
    m = PyImport_AddModule("__main__");
    if (m == NULL)
        return _PyStatus_ERR("can't create __main__ module");

    d = PyModule_GetDict(m);
    ann_dict = PyDict_New();
    if ((ann_dict == NULL) ||
        (PyDict_SetItemString(d, "__annotations__", ann_dict) < 0)) {
        return _PyStatus_ERR("Failed to initialize __main__.__annotation
s__");
    }
    Py_DECREF(ann_dict);

    if (PyDict_GetItemString(d, "__builtins__") == NULL) {
        PyObject *bimod = PyImport_ImportModule("builtins");
        if (bimod == NULL) {
            return _PyStatus_ERR("Failed to retrieve builtins module");
        }
        if (PyDict_SetItemString(d, "__builtins__", bimod) < 0) {
            return _PyStatus_ERR("Failed to initialize __main__.__builtin
s__");
        }
        Py_DECREF(bimod);
```

```
    }
    loader = PyDict_GetItemString(d, "__loader__");
    if (loader == NULL || loader == Py_None) {
        PyObject *loader = PyObject_GetAttrString(interp->importlib,
                                                  "BuiltinImporter");
        if (loader == NULL) {
            return _PyStatus_ERR("Failed to retrieve BuiltinImporter");
        }
        if (PyDict_SetItemString(d, "__loader__", loader) < 0) {
            return _PyStatus_ERR("Failed to initialize __main__.__loader_
_");
        }
        Py_DECREF(loader);
    }
    return _PyStatus_OK();
}
```

根据我们使用Python的经验，我们知道最终Python肯定会创建一个PyListObject对象，也就是sys.path，里面包含了一组 PyUnicodeObject *，指向的每一个PyUnicodeObject的内容都代表了一个搜索路径。但是这一步不是在这里完成的，至于是在哪里完成的，我们后面会说。

另外需要注意的是：在上面的逻辑中，解释器将 __main__ 这个模块添加进去了，这个 __main__ 估计不用我多说了。之前在 PyModule_New 中，创建一个 PyModuleObject对象之后，会在其属性字典中插入一个名为 "__name__" 的 key，value 就是 "__main__"。但是对于当然模块来说，这个模块也叫做 __main__。

```
name = "古明地觉"
import __main__
print(__main__.name)  # 古明地觉

import sys
print(sys.modules["__main__"] is __main__)  # True
```

我们发现这样也是可以导入的，因为这个__main__就是模块本身。

```
static PyStatus
add_main_module(PyInterpreterState *interp)
{
    PyObject *m, *d, *loader, *ann_dict;
    //创建__main__ module，并将其加入到 interp->modules 中
    m = PyImport_AddModule("__main__");
    if (m == NULL)
        return _PyStatus_ERR("can't create __main__ module");
    //获取__main__的属性字典
    d = PyModule_GetDict(m);

    //获取interp->modules中的__builtins__ module
    if (PyDict_GetItemString(d, "__builtins__") == NULL) {
        PyObject *bimod = PyImport_ImportModule("builtins");
        if (bimod == NULL) {
            return _PyStatus_ERR("Failed to retrieve builtins module");
        }
        //将<"__builtins__", __builtins__>
        //加入到__main__ module的dict中
        if (PyDict_SetItemString(d, "__builtins__", bimod) < 0) {
            return _PyStatus_ERR("Failed to initialize __main__.__builtin
s__");
        }
        Py_DECREF(bimod);
    }
    //......
}
```

因此我们算是知道了，为什么 python xxx.py 执行的时候，__name__ 是 "__main__" 了，因为我

们这里设置了。而Python沿着名字空间寻找的时候，最终会在 __main__ 的 local空间中发现 __name__，且值为字符串 "__main__"。但如果是以 import 的方式加载的，那么 __name__ 则不是 "__main__"，而是模块名。

其实这个__main__我们是再熟悉不过的了，当输入dir()的时候，就会显示__main__的内容。dir是可以不加参数的，如果不加参数，那么默认访问当前的 local 空间，也就是__main__。

```
>>> __name__
'__main__'
>>>
>>> __builtins__.__name__
'builtins'
>>>
>>> import numpy as np
>>> np.__name__
'numpy'
>>>
```

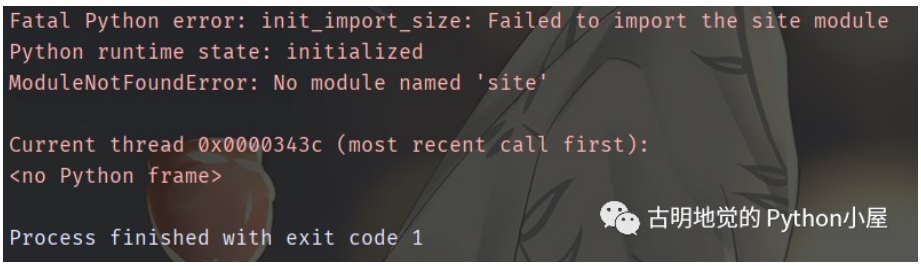
所以说，访问模块就类似访问变量一样。modules 里面存放了所有的**<模块名，模块>**。当我们调用 np 的时候，是会找到 name 为 "numpy" 的模块，然后这个值里面也维护了一个字典，其中也有一个 key 为 __name__ 的 entry，value 为 "numpy"。

设置site-specific的module的搜索路径

Python是一个非常开放的体系，它的强大来源于丰富的第三方库，这些库由外部的 py文件来提供。当使用这些第三方库的时候，只需要简单地进行import即可。一般来说，这些第三方库都放在Lib/site-packages 中，如果程序想使用这些库，直接导入即可。

但是到目前为止，我们好像也没看到 Python 将 site-packages 路径设置到搜索路径里面去啊。其实在完成了 __main__ 的创建之后，Python 才腾出手来收拾这个site-package。这个关键的动作在于Python的一个标准库：site.py。

我们先来将Lib目录下的site.py删掉，然后导入一个第三方模块，看看会有什么后果。



因此 Python 在初始化的过程中确实导入了 site.py，所以才有了如下的输出。而这个 site.py 也正是 Python 能正确加载位于 site-packages 目录下第三方包的关键所在。我们可以猜测，应该就是这个 site.py 将 site-packages 目录加入到了sys.path中，而这个动作是由 init_import_size 完成的。

```
static PyStatus
new_interpreter(PyThreadState **tstate_p)
{
    //......
        if (config->site_import) {
            status = init_import_size();
            if (_PyStatus_EXCEPTION(status)) {
                return status;
            }
```

```
        }
    //......
}


static PyStatus
init_import_size(void)
{
    PyObject *m;
    m = PyImport_ImportModule("site");
    if (m == NULL) {
        //这里的报错信息是不是和上图中显示的一样呢？
        return _PyStatus_ERR("Failed to import the site module");
    }
    Py_DECREF(m);
    return _PyStatus_OK();
}
```

在 init_import_size 中，只调用了 PyImport_ImportModule 函数，这个函数是import机制的核心所在。

比如 PyImport_ImportModule("numpy") 等价于 import numpy。

以上就是运行时环境的初始化所做的事情，但是需要注意：此时虚拟机还没有启动。对，上面的那些工作都只是前戏，是虚拟机启动之前所做的一些准备工作。而在准备工作做完之后，虚拟机就会正式启动。那么怎么启动呢？我们下一篇文章再聊。

收录于合集 #CPython 97

‹ 上一篇
《源码探秘 CPython》83. 激活 Python 虚拟机

下一篇 ›
《源码探秘 CPython》81. import 机制是怎么实现的?