Python Insight

Overview

Python程序调试

pdb

pdb是python通过库的形式提供的调试工具，只需要通过`python -m pdb demo1.py`就可以借助于交互式命令调试自己的python脚本。

但是如果要分析Python本身的话，或者说你遇到如下一些问题[DWG, DGPPWG]:

- python无法捕获的段错误
- 进程挂起，pdb根本无法返回trace信息
- 后台进程失控

因此这个时候就需要gdb出场。

gdb

gdb调试可以完全参考其官方文档, 通过`gdb python`或者`gdb python -p $pid`进行。 注意编译python的时候指定`--with-pydebu`标签。

效果简单演示如下:

```
$ gdb python3.6
(gdb) !cat demo1.py
def add(a, b):
  return a + b;

print(add(1.0, 2))
(gdb) b main
(gdb) r demo1.py
(gdb) l
15  }
16  #else
17
18  int
19  main(int argc, char **argv)
20  {
21      wchar_t **argv_copy;
22      /* We need a second copy, as Python might modify the first one. */
23      wchar_t **argv_copy2;
24      int i, res;
(gdb) tui enable ## 效果更佳
```

Python程序执行

注意接下来代码涉及到python源码的地方，都是指python3.6版本的源码。

一个python程序执行要经过Initialization、compilation以及execution三个阶段。

- Initialization：完成非交互式模式下对Python进程的初始化工作，源码在Py_InitializeEx；
- compilation：将源码编译成为代码对象的过程；编译过程从run_file开始；
- execution：通过解释器执行代码对象的过程，从 _PyEval_EvalCodeWithName开始，`Python/ceval.c:3855`；

从高度抽象的层面来看，Python的执行过程如下：

图1： 编译执行过程，来自[Book IPVM]

在分析源码的时候，可以通过doxygen生成代码文档和调用依赖信息，然后通过`python -m SimpleHTTPServer`查看代码依赖关系和文档。sss

词法分析

词法分析的目的是将源代码转换为Parser Tree (也叫做CST, concrete syntax tree)。

Python parser是一个符合LL(1)文法的解析器，按照扩展巴恩斯范式(EBNF)进行定义文法。具体文法定义在Grammar/Grammar。

Python的token主要分为以下几种：

- identifiers： 程序里面定义的标识符，例如函数名、变量名、类名等
- operators： 操作符，
- delimiters：定界符，用于进行表达式分组，包括`( , )`, `{,}`, `=`, `*=`等

- literals： 字面量，代表常量值，例如字符串常量等
- comments： 注释
- NEWLINE： 换行符
- INDENT and DEDENT ： 缩进和回退

parser的核心代码在PyParser_ParseFileObject. 从`run_file/PyRun_AnyFileExFlags/PyRun_SimpleFileExFlags/PyRun_FileExFlags/PyParser_ASTFromFileObject`调用而来， 返回node结构体，一个node就是一个parse tree节点，表示一个token, 定义于`Include/node.h:41`。

例如：

```
In [114]: code_str = """
     ...: def add(a, b):
     ...:   return a + b
     ...: add(1, 2)
     ...: """

In [115]: from pprint import pprint
     ...: st = parser.suite(code_str)
     ...: pprint(parser.st2list(st))
```

可以打印出parse tree如下：

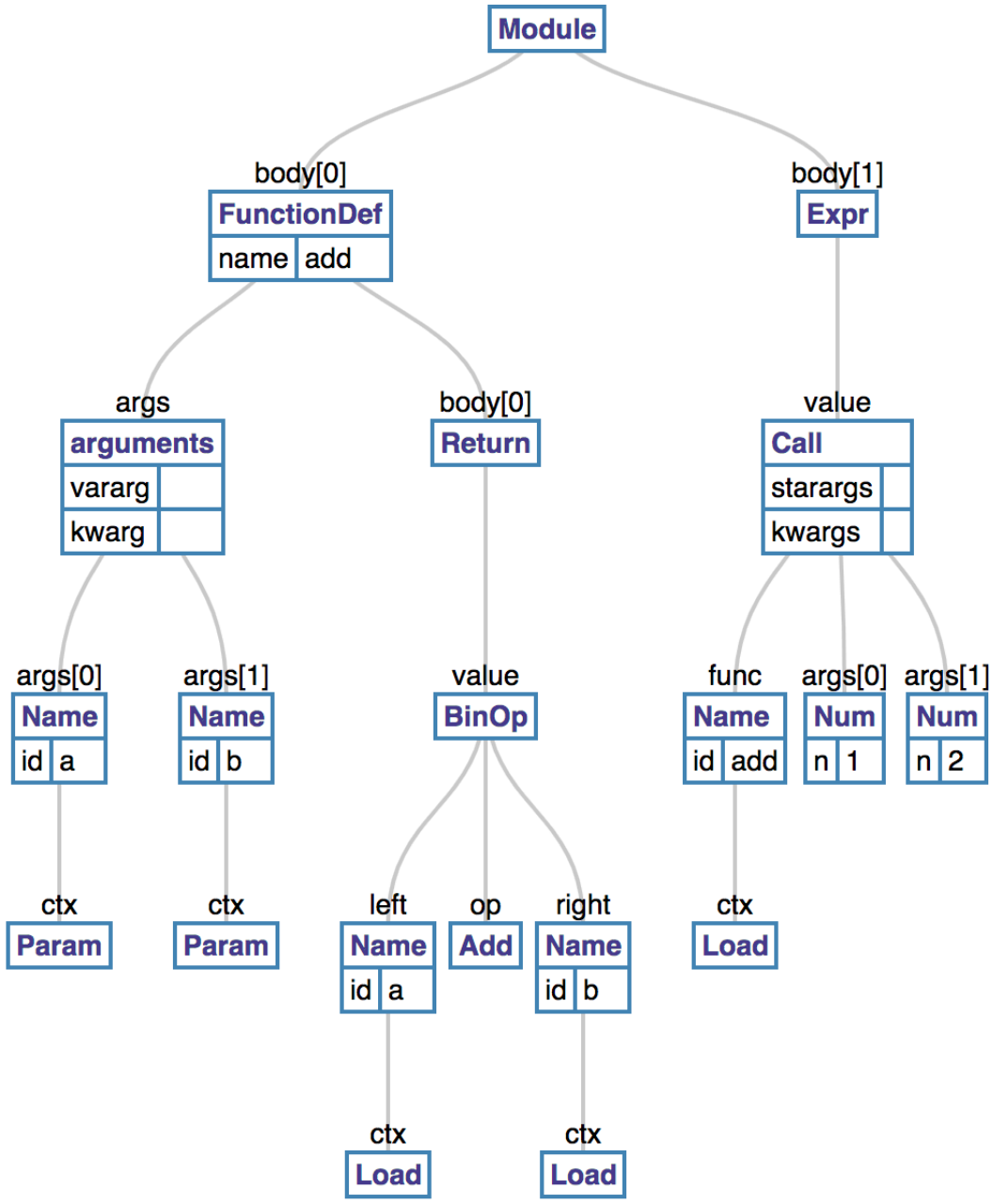
```
In [118]: pprint(parser.st2tuple(st))
(257,
 (269,
  (295,
   (263,
    (1, 'def'),
    (1, 'add'),
    (264,
     (7, '('),
     (265, (266, (1, 'a')), (12, ','), (266, (1, 'b'))),
     (8, ')')),
    (11, ':'),
    (304,
     (4, ''),
     (5, ''),
     (269,
      (270,
       (271,
        (278,
         (281,
          (1, 'return'),
          (274,
           (306,
            (310,
             (311,
```

```
              (312,
               (313,
                (316,
                 (317,
                  (318,
                   (319,
                    (320,
                     (321, (322, (323, (324, (325, (1, 'a')))))),
                     (14, '+'),
                     (321, (322, (323, (324, (325, (1,
'b'))))))))))))))))))),
       (4, ''))),
     (6, ''))))),
 (4, ''),
 (0, ''))
```

其中每个touple的第一个元素的编号信息可以在 graminit 和 token找到。例如257表示file_input等, 通过Language Reference可以查到具体token的用法。

语法分析

语法分析是将Parse Tree转换为AST。通过vpyast可以很方便的查看AST的结构。 例如上面的例子的AST结构如下：



也可以借助ast库进行分析, 例如：

```
In [124]: import ast
In [125]: import pprint
In [126]: node = ast.parse(code_str, mode="exec")
In [127]: ast.dump(node)
Out[127]: "Module(body=[FunctionDef(name='add', args=arguments(posonlyargs=[],
args=[arg(arg='a', annotation=None, type_comment=None), arg(arg='b',
annotation=None, type_comment=None)], vararg=None, kwonlyargs=[], kw_defaults=
[], kwarg=None, defaults=[]), body=[Return(value=BinOp(left=Name(id='a',
ctx=Load()), op=Add(), right=Name(id='b', ctx=Load()))))], decorator_list=[],
returns=None, type_comment=None)], type_ignores=[])"
```

AST的解析是从PyAST_FromNodeObject开始的， 传入parser tree的根节点，遍历Parse Tree，按照语法规则Language Reference表示的syntax diagram，转成AST格式，传出一个_mod(`Include/Python-ast.h:44`)对象。也就是上图的Module。

转换成AST之后，源码由`run_mod`函数开始进行Module的编译执行。核心逻辑在`PyAST_CompileObject` (Python/compile.c:301)完成，返回`PyCodeObject`。 在这个过程，首先要进行语义分析，通过`PySymtable_BuildObject`(Python/symtable.c:240)然后建立对应的符号表。

语义分析(semantic analysis)

语义分析就是要分析statement的含义的过程，分析语义自然要涉及到每个stmt所在上下文以及对应的symbol的lifecyle管理。在Python里面， 所有的元素都是对象。 对于c/c++这类语言，一个变量的类型是跟变量名字绑定的，但是在Python里面类型是跟对象绑定的。同时对象也具备自己的属性和方法。del关键字则可以实现解绑。

```
x = 5  # 将x绑定到5对应的对象上；
print(type(x));
x = "hello" #将x绑定到字符串"hello"对应的对象上；
print(type(x))
```

输出如下：

```
<class 'int'>
<class 'str'>
```

在进一步介绍之前，需要先了解下Python程序的结构。基本结构如下：

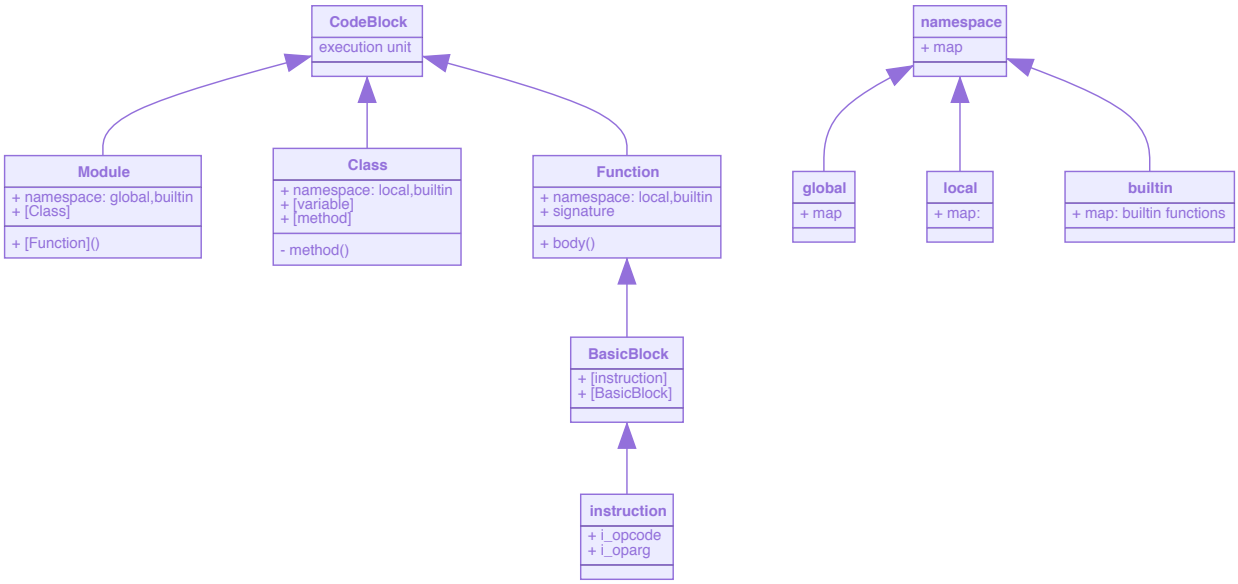


图1： 程序组成

程序中的对象的scopes是该对象按照namespace的访问规则能够看到的对象视图。 大家可能对lamdba会有疑问，本质上lamda就是一个局部匿名结构体，结构体包含了其本身可见的对象作为其成员变量，然后重载`()`成为一个callable对象。

那么有了以上信息的时候，我们就可以构建符号表， 以便于进一步生成字节码，进行执行。

符号表

符号表的左右就是要将图1中的各种对象的关系描述清楚。Python的符号表结构见symtable, 也可以参考python-internals-symbol-tables了解每个字段的含义. 我们可以借助于symtable显示我们分析的Parse Tree生成的符号表信息。

我们先定义一个符号表打印函数：

```
def describe_symtable(st, recursive=True, indent=0):
    def print_d(s, *args):
        prefix = ' ' * indent
        print(prefix + s, *args)

    assert isinstance(st, symtable.SymbolTable)
    print_d('Symtable: type=%s, id=%s, name=%s' % (
                st.get_type(), st.get_id(), st.get_name()))
    print_d('  nested:', st.is_nested())
    print_d('  has children:', st.has_children())
    print_d('  identifiers:', list(st.get_identifiers()))

    if recursive:
        for child_st in st.get_children():
            describe_symtable(child_st, recursive, indent + 5)
```

然后打印下之前定义的代码串的符号表：

```
In [174]: code_str = """
     ...: class Math(object):
     ...:   def __init__(a: int):
     ...:     self.base = a;
     ...:   def add(self, b):
     ...:     return self.base + b;
     ...: m = Math(10);
     ...: print(m.add(100));
     ...: """
In [179]: st = symtable.symtable(code_str, "<string>", "exec")

In [180]: describe_symtable(st)
Symtable: type=module, id=140641729334304, name=top
  nested: False
  has children: True
  identifiers: ['Math', 'object', 'm', 'print']
     Symtable: type=class, id=140641729333480, name=Math
       nested: False
       has children: True
       identifiers: ['__init__', 'int', 'add']
          Symtable: type=function, id=140641729332896, name=__init__
            nested: False
            has children: False
            identifiers: ['a', 'self']
          Symtable: type=function, id=140641729333408, name=add
```

```
            nested: False
            has children: False
            identifiers: ['self', 'b']
```

可以看到，这种嵌套结构跟之前图1的定义非常的类似。

另外Python是支持多继承的，在进行MRO（方法解析顺序）的时候，容易出现二义性，Python使用C3 算法解决这个问题。

符号表基本上将所有的对象的lifecycle以及对象之间的关系建立起来了，有了这些信息之后，就可以创建代码对象了。

代码对象

有了符号表信息，通过`compiler_mod/assemble/makecode`(Python/compile.c)进一步生成`PyCodeObject`。通过内嵌函数compile函数可以近距离的观察code object的样子。

```
In [230]: code_str = """
     ...: def add(a, b):
     ...:   return a + b
     ...: add(1, 2)
     ...: """

In [238]: code_obj = compile(code_str, 'mycode', 'exec')
In [239]: code_obj.co_filename #文件名
Out[239]: 'mycode'
In [236]: code_obj.co_code  #字节码序列
Out[236]: b'd\x00d\x01\x84\x00Z\x00e\x00d\x02d\x03\x83\x02\x01\x00d\x04S\x00'

In [248]: dis.dis(code_obj)
  2           0 LOAD_CONST               0 (<code object add at 0x12913bea0,
file "mycode", line 2>)
              2 LOAD_CONST               1 ('add')
              4 MAKE_FUNCTION            0
              6 STORE_NAME               0 (add)

  4           8 LOAD_NAME                0 (add)
             10 LOAD_CONST               2 (1)
             12 LOAD_CONST               3 (2)
             14 CALL_FUNCTION            2
             16 POP_TOP
             18 LOAD_CONST               4 (None)
             20 RETURN_VALUE

Disassembly of <code object add at 0x12913bea0, file "mycode", line 2>:
  3           0 LOAD_FAST                0 (a)
              2 LOAD_FAST                1 (b)
              4 BINARY_ADD
              6 RETURN_VALUE
```

```
In [249]: code_obj.co_code[0]  # 下标0对应上面的dis结果的第一列
Out[249]: 100
In [250]: code_obj.co_code[1]
Out[250]: 0
```

dis库可以将代码对象格式化打印出来，code_obj.co_code是字节码序列，按照`opcode`:`oparg`，的形式放置，例如第一个字节是100，通过查找opcode表 看到其对应的指令是`LOAD_CONST`,对应dis出来的第一行的第二列，其对应的参数个数oparg是0，对应第三列，表示没有参数，如果有的话，就在第四列。 例如：

```
In [255]: code_obj.co_code[14]
Out[255]: 131  #CALL_FUNCTION,
https://github.com/python/cpython/blob/3.6/Include/opcode.h#L104
In [256]: code_obj.co_code[15]
Out[256]: 2
```

CALL_FUNCTION需要一个参数argc，表示参数个数，也就是code_obj.co_code[15]，实际的参数已经通过`LOAD_CONST`将常量压栈，切换函数帧(Function Frame)进入函数体之后，通过`LOAD_FAST`找到再将frame.f_code.co_varnames[0] 和frame.f_code.co_varnames[1]压栈，然后通过加法运算, 更多细节见解释器部分。

注意代码对象的*co_varnames*是当前code block的局部变量，*co_names*则是当前block用到的非局部变量。

字节码

代码对象的`code_obj.co_code`保存了字节码。字节码是由符号表转换而来，通过`assemble`(Python/compile.c:5315)函数生成。生成真实的字节码之前，首先要对前面创建的符号表进行后序遍历(`dfs`)，然后将被遍历到的节点按照`opcode`:`oparg`的格式插入(`assemble_emit`)到字节码数组的末尾。

Python的优化逻辑主要在`PyCode_Optimize`(Python/peephole.c:425)，主要执行了basic peephole 优化。peehole更多的细节可以参考龙书或者llvm相关的介绍。

解释过程

字节码可以理解为一种已经经过优化了的中间代码，但是不能直接执行，需要借助于虚拟机/解释器进行执行。 前面的compile_mod执行完成之后，就开始进入了`PyEval_EvalCode/PyEval_EvalCodeEx/_PyEval_EvalCodeWithName`, 在这个函数里面，首先给当前的Code Object通过`PyFrame_New`创建一个frame。

帧frame

帧是维护当前指令集上下文信息的容器，每个code block都需要创建一个帧。 它至少要包含当前指令所在的代码对象、所有可见的命名空间、全局变量、本地变量、内嵌函数等信息。同时为了能够在执行完成当前帧之后返回，到调用的地方，还定义了f_back字段，指向上一个frame。同时还需要引用到当前指令执行的进程、栈和字节码等信息。

帧通过`PyFrame_New`创建的时候，首先要判断下代码对象的`code->co_zombieframe`是否为空，不为空就用它了，否则再去通过`PyObject_GC_NewVar`创建一个新。 然后再进行新的frame的初始化。co_zombieframe可以起到缓存的作用。

frame创建完成之后，通过`PyEval_EvalFrameEx`开始执行解释流程。

解释过程

解释线程

前面提到在Initialization环节，会对Python解释进程(Process)进行初始化，构造一个全局的`PyInterpreterState`对象。这个对象会维护一个元素为`PyThreadState`的双向队列。每次执行一个Code Block的时候，要创建一个对应的`PyThreadState`, 作为任务创建一个线程进行处理。

PyThreadState对应的进程的入口函数是`t_bootstrap`，其中通过`PyEval_AcquireThread/take_gil`获得GIL， GIL实际上是`pthread_cond_t* gil_cond`, 通过`pthread_cond_timedwait`去等待条件满足。如果满足表示获得锁。用完通过`drop_gil`释放。

创建完线程和代码对象之后，就可以通过默认的执行器`_PyEval_EvalFrameDefault`(Python/ceval.c:722)进行字节码执行了。

解释流程

TODO

持续更新

后面持续分析annotation、lib库、gc等实现。同时也会不停完善上面的部分。

参考

[Book IPVM] https://leanpub.com/insidethepythonvirtualmachine

[DWG] : https://wiki.python.org/moin/DebuggingWithGdb

[DGPPWG] : Debugging Hung Python Processes With GDB https://pycon.org.il/2016/static/sessions/john-schwarz.pdf

[PTC] : Python Type Checking (Guide) https://realpython.com/python-type-checking