深入 Python 编程

（预览版，0.3）

前言

嗯，这个地方一般用来说一些无关紧要但又不得不说的废话。

在很久以前，我还在兴奋地研究 Delphi/VCL。有个在金山的朋友告诉我他选择 Python 作为下一门语言，那是我第一次听到这个名字。后来又陆陆续续在《程序员》杂志上看过几篇介绍文章。老实说，我最初并不看好 "脚本语言"，这可能源自 97 年被 Perl 吓到的经历。我喜欢那种严谨具有学院风格的语言，比如 Pascal。在 MS-DOS 时代，我就是被 Pascal 带入门槛的。

再往后接触 Python，已经是 2006 年，因为当时还在迷恋 .NET/C#，所以并没有什么实质性的结果。直到 2 年前，我纠结着要从 Windows 平台转入 Linux。

从 .NET 1.0 开始，一直到 4.0，大大小小写了几百篇技术文字。从 C# 语言到 FX/BCL，还有 MSIL、SOS.dll，以及 MVC、LINQ 的流程分析，甚至是加密破解，Metadata 手工修复。个人感觉还是比较深入和全面的。也正因为如此，对 .NET 愈发臃肿的发展趋势颇为不满。加上工作的原因，需要将开发平台转移，急需找个接替者，Python 此时作为一个多面手顺利进入我的世界。

因为有 C 和 ASM 的底子，加上十几年的开发经验，我没有费太多的功夫就上手开始工作。2 年后，我重新整理这期间所有的学习心得，打算为 Python 添砖加瓦，顺便为国内不旺的人气添几把干柴，如此就有了这本书。

本书的定位介于入门和精通之间 —— 深入。因此需要读者有一定的 Python 编程经验，起码看过几本经典入门书籍，并写过一些代码。

个人推荐 《Learning Python, 4th Edition》作为入门书籍，貌似有中文版 (没看过，不知道翻译质量如何，我个人比较倾向于看英文版)。该书系统而全面地介绍了 Python 语言开发的各个方面，让您对 Python 有个完整的认识。其中某些复杂的东西可能还很模糊，没关系，这就是我这本书要完成的工作。最后呢，陈儒大仙的《Python 源码剖析 —— 深度探索动态语言核心技术》就是 "葵花宝典" 级别的东西了。

> **入门**：《Learning Python, 4th Edition》
> **深入**：《深入 Python 编程》(本书)
> **精通**：《Python 源码剖析》

好吧，折腾完这三步，您在简历上写 "精通 Python 语言" 就多了些底气。

这本书着重点还是 "Python 编程"，对 Python 虚拟机以及源码级别的东西点到为止，只求能诠释理论实现的过程，而非完整剖析源码（这活陈儒大仙已经做得非常好了）。您最好能阅读一些 C 代码，这样有助于更好地理解。当然，我尽可能添加详细的注释说明。

本书分为三个部分：

1.Python 语言

深入说明语言层面的知识和理论，包括 CPython 虚拟机的一些知识。重要的是，我们应该如何探究一种语言的理论实现，而不仅仅是背诵一些文字规则。

2.Python 标准库

编程不仅仅是语言的事，我们总是需要各种各样的数据结构、框架来完成应用开发和部署。提供一套相对完整而高效的标准库已经是当前所有新兴语言的 "标准"。

3.第三方库

Python 拥有上万种各种各样的第三方库。夸张点说："只有想不到的，没有找不到的"。其中诸如 gevent、tornado 等优秀的第三方库和框架能让你的开发工作有个更高的起点。

强烈建议 Pythoner 要经常去国外的一些论坛溜达，很多优秀的资源少为人知，非常可惜。

在写这本书的时候，我将遵循一些基本的原则：

- 代码能说清楚的，就不要写太多废话。
- 代码尽可能简单，便于阅读。

本书将在网上发布预览版，希望各位神仙大牛们不吝赐教。作者本人颇有些心虚，生怕自身不足生出误导和完全错误的表达。敬谢！

联系方式：

email: qyuhen@hotmail.com
msn: qyuhen@hotmail.com （通常用于工作用途，可能不会回复）
QQ: 1620443 （移动的时候也能用）
weibo: http://weibo.com/qyuhen （这个好，灌水聊天皆可）

另，因工作时间和个人作息习惯问题，如不能及时回复，敬请谅解。

雨痕 Q.yuhen
2012-01-11

更新记录

2012-02-09 第一章截稿，发布预览版 0.1。

2012-02-16 第一章增加和修改了部分内容。

第二章截稿，发布预览版 0.2。

2012-02-27 全书第一部分从 11 章调整为 10 章。

第二章增加和修改了部分内容。

第三章截稿，发布预览版 0.3。

目录

第一部分 Python 语言

不要误会，我不会象大多数书那样去慢条斯理地逐个介绍语法。我们要面对的是一次探险旅程。

收拾工具，整理好行囊，你将是探险团队中的重要一员。我们有计划，但时时刻刻都可能面临 "惊心动魄" 的状况。你可能要一头扎入源码里寻幽探秘，也或者是在调试器里面对一堆十六进制数字发呆。

总之，不要指望这本书会按常规出牌，我希望能有惊悚小说那样的节奏和快感，免得大家翻几页后就丢在一旁，白白浪费了金钱和时间，还可能成为家里的卫生死角。

个人建议，在看本书的同时，把你喜欢的经典书籍放在旁边 (我喜欢纸质书，看电子书无法专心) 互为参考。

好吧，让我们开始探索 Python 语言的秘密。我们的目标是：刨出个二五八万来。

Q.yuhen: 老实说，写这些废话比写代码困难多了……

第 1 章 基础

标题虽然叫 "基础"，但内容却不简单。在本章，我们要为我们的探险旅程打包所有的行李和工具，也就是所谓 "利其器"。如果你没有 Python 编程经验，或者刚刚做完简单的入门 Tour，那么可能会被吓到。

Python 让编程变得简单和享受，但其自身的实现却相当复杂。在我们习以为常的简单优雅背后隐藏了大量的理论和技巧。如果能深入了解这些原理、实现以及限制，可以让我们在开发过程中避免诸多麻烦，绕开不必要的错误陷阱，有效提高代码质量和执行效率。

首先你需要准备如下环境：

> **Linux** 或 **OS X**（当然也可以是 Windows。本书使用 OS X Lion 10.7.3）
> **Python 2.7** （本书所分析的都是 CPython 实现，http://www.python.org）
> **IPython 0.12** （比 CPython 自带交互环境更好用，支持代码补全和高亮）
> **IPdb** （可选，使用 easy_install 安装，比内置的 pdb 包更好用些）
> **GDB** （GCC 带的调试器）
>
> 记得从官方网站下载一份源码 (本书使用的是 2.7.2)，我们在后面的分析过程中会用到。

需要强调的是，本书是针对官方 CPython 版本的分析，Jython 和 IronPython 因各自虚拟机平台的实现异同而导致完全不同的底层数据结构和执行机制。当然，相关理论知识还是共通的。

1.1 基本工具

好了，在我们开始 "折腾" 前，先练习一下手头工具的使用方法，以便在后面的旅途中能坚持下来。

1.1.1 IPython

比起 CPython 自带的命令行交互环境，IPython 要强大许多，各种各样的 Magic Function 让我们可以轻松完成很多工作。比如命令和成员自动完成，查看对象帮助信息、源码等等。你甚至可以把 IPython 当作一种特殊的 IDE 来用。

我习惯于在 IPython 上随手写些实验代码，研究新发现的资源，或者直接当作 *nix Shell 使用。我觉得学习 Python 并不适合用 PyCharm 这类可视化 IDE 环境。静下心来，直接面对终端，将注意力集中在代码上。在 *nix 下工作时，并不总是有 IDE 这样奢侈的工作环境。对于被宠坏的家伙们，我们应该让他习惯在终端模式下用 VIM。当然，这并不表示我反对使用 IDE，毕竟提高生产力的东西总归是好的，重要的是适可而止，别折腾了 n 年，连个函数名都没记清楚。话说我见过一些 .NET 程序员，在 Visual Studio .NET 里面用 Console.WriteLine 来完成调试工作，这不能不说是 IDE 开发者的 "杯具"。

不要多，我们先掌握几个基本的 IPyhton Magic Function 用法就行。作为在终端下生存的一员，要习惯 help、man 等命令。

命令	说明
%quickref	IPython 使用快速导引
?, ??	查看对象、MagicFunction 等信息
!	执行 Shell Command
%sc, %sx	捕获 Shell Command 输出结果
%pwd	查看当前工作目录
%cd	工作目录跳转
%pushd, %popd, %dirs	目录栈操作
%bookmark	目录书签
%save	保存操作历史
%logstart, ... %logstop	log 记录器
%edit	编辑源码文件
%pycat, %pfile	显示源码
%who, %whos	查看当前环境下的所有对象
%run	执行源码文件
%prun, %time, %timeit	性能测试
%pdb	开打或关闭调试器
%reset	重置执行环境，删除全部变量
%quit	退出 IPython

随时可以用 <TAB> 键做代码自动完成，多数时候可以省略 Magic Function 的 "%" 前缀符号。

```
$ ipython
Python 2.7.1 (r271:86832, Jun 16 2011, 16:59:05)
Type "copyright", "credits" or "license" for more information.

IPython 0.12 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]:
```

1.1.1.1 帮助

%quickref 显示 Quick Reference Card，建议仔细阅读，并尽可能把所有的功能都实验一遍。

```
IPython -- An enhanced Interactive Python - Quick Reference Card
================================================================

obj?, obj??      : Get help, or more help for object (also works as
                   ?obj, ??obj).
?foo.*abc*       : List names in 'foo' containing 'abc' in them.
%magic           : Information about IPython's 'magic' % functions.

Magic functions are prefixed by %, and typically take their arguments without
parentheses, quotes or even commas for convenience.

... ...
```

"?" 显示目标的基本信息，而 "??" 返回的信息更加详细，比如查看函数、模块的源码。

```
In [2]: def test():          # 定义一个包含了 __doc__ 的函数
   ...:     """
   ...:     haha, test...
   ...:     """
   ...:     pass
   ...:

In [3]: test?                # 查看函数信息，包括 __doc__。
Type:       function
Base Class: <type 'function'>
String Form:<function test at 0x10ddff668>
Namespace:  Interactive
Definition: test()
Docstring:  haha, test...

In [4]: test??               # 更多信息？比如源码！
Type:       function
Base Class: <type 'function'>
String Form:<function test at 0x10ddff668>
Namespace:  Interactive
Definition: test()
Source:
def test():
    """
    haha, test...
    """
    pass
```

当你阅读某个开源项目（比如 tornado）源码时，使用 pdb 显示 Call Stack Frames 获得执行流程，然后配合 "??" 查看相关对象和函数的源码。岂不比用一个文本编辑器乱哄哄地查找好？
Python 作为一种动态语言，很多东西是在运行时生成和改变的，不能用 Ctags 的老习惯来对待。

想查看某个 Maigc Function 的具体使用方法?

```
In [5]: %bookmark?

Type:       Magic function
Base Class: <type 'instancemethod'>
Namespace:  IPython internal
Definition: %bookmark(self, parameter_s='')
Docstring:
Manage IPython's bookmark system.

%bookmark <name>       - set bookmark to current dir
%bookmark <name> <dir> - set bookmark to <dir>
%bookmark -l           - list all bookmarks
%bookmark -d <name>    - remove bookmark
%bookmark -r           - remove all bookmarks

... ...
```

或者替代那个别扭的 pydoc，用来快速查看开发手册。

```
In [6]: import sys

In [7]: sys._getframe ?
Type:       builtin_function_or_method
Base Class: <type 'builtin_function_or_method'>
String Form:<built-in function _getframe>
Namespace:  Interactive
Docstring:
_getframe([depth]) -> frameobject

Return a frame object from the call stack.  If optional integer depth is
given, return the frame object that many calls below the top of the stack.
If that is deeper than the call stack, ValueError is raised.  The default
for depth is zero, returning the frame at the top of the call stack.

This function should be used for internal and specialized
purposes only.
```

1.1.1.2 系统命令

调用系统命令，通常以 "!" 开头 (某些时候可能省略)。

- 在 Shell Command 中直接以 "$var_name" 格式使用 Python 变量。
- 在 Shell Command 中使用系统环境变量时，格式为 "$$env_var"，以便和 Python 变量区分开来。

```
In [8]: !ls /etc
afpovertcp.cfg        hosts.equiv           pf.anchors
aliases               irbrc                 pf.conf
```

```
... ...

In [9]: path = "/bin"

In [10]: !ls $path           # 注意格式: $var_name
... ...
chmod          df             hostname       ln             ps             sh             test
cp             domainname     kill           ls             pwd            sleep          unlink

In [11]: !echo $$HOME        # 输出 shell 环境变量，注意对 $ 转义，否则就寻找 Python 变量了。
/Users/yuhen
```

用 %sc、%sx 可以捕获 Shell Command 的输出结果。

- %sc: 返回命令输出的完整字符串。
- %sx: 一个 list 对象。

注意两个命令不同的语法格式。

```
In [12]: %sc s = ls ~

In [13]: s
Out[13]: 'Desktop\nDocuments\nDownloads\nLibrary\nMovies\nMusic\nPictures\nPublic\n'

In [14]: l = %sx ls ~

In [15]: l
Out[15]:
['Desktop',
 'Documents',
 'Downloads',
 'Library',
 'Movies',
 'Music',
 'Pictures',
 'Public']
```

1.1.1.3 目录

将 IPython 作为 Shell 使用时，免不了要在不同工作目录下跳转。

- %pwd: 显示当前目录；
- %cd: 前往某个目录；
- %pushd, %popd, %dirs: 目录栈操作。

```
In [16]: pwd
Out[16]: u'/Users/test/python'

In [17]: cd /bin
/bin
```

```
In [18]: cd -         # 回到上个目录
/Users/test/python

In [19]: pushd /bin   # 将当前目录压入栈中，并跳转到指定目录
/bin
Out[19]: [u'~/Documents/Projects/learn/python']

In [20]: dirs         # 查看目录栈内容，可以多次 pushd，按 FILO 弹出。
Out[20]: [u'~/Documents/Projects/learn/python']

In [21]: popd         # 弹出目录占中最后一个目录，并跳转。
/Users/test/python
popd -> ~/Documents/Projects/learn/python
```

如果觉得目录栈不方便，还可以用目录书签 (Bookmark)。

- %bookmark <name>: 将当前目录加入书签。
- %bookmark <name> <dir>: 将指定目录加入书签。
- %bookmark -l: 显示所有书签。
- %bookmark -d <name>: 删除书签。
- %bookmark -r: 删除全部书签。

```
In [22]: bookmark learnPython        # 将当前目录加入书签，并取名为 learnPython

In [23]: bookmark ubin /usr/bin      # 将某个目录加入书签

In [24]: bookmark -l                 # 看看所有的书签
Current bookmarks:
learnPython -> /Users/test/python
ubin        -> /usr/bin

In [25]: cd learnPython              # 使用书签跳转
(bookmark:learnPython) -> /Users/test/python
/Users/test/python

In [26]: cd ubin                     # 再次使用书签跳转
(bookmark:ubin) -> /usr/bin
/usr/bin

In [27]: pwd                         # 跳转无误!
Out[27]: u'/usr/bin'

In [28]: cd -b learnPython           # 如果担心书签名和某个子目录名冲突，可以用 -b 参数。
(bookmark:learnPython) -> /Users/test/python
/Users/test/python

In [29]: bookmark -d ubin            # 删除某个书签

In [30]: bookmark -l
Current bookmarks:
learnPython -> /Users/test/python
```

1.1.1.4 记录

折腾了老半天，想不想把测试成果保存下来？%save 可以指定要保存的行号范围。

格式: %save [options] filename n1-n2 n3-n4 ... n5 .. n6 ...
参数: -r 按照输入的原始样式保存。

```
In [31]: a = 1

In [32]: b = a + 1

In [33]: p a                            # 这行写错了，待会别保存。
  File "<ipython-input-88-b38b07de0377>", line 1
    p a
      ^
SyntaxError: invalid syntax

In [34]: print a
1

In [35]: save -r test.py 31-32 34    # 将 31 ~ 32 行，还有第 34 行保存到 test.py。
The following commands were written to file `test.py`:
a = 1
b = a + 1
print a

In [36]: cat test.py
# coding: utf-8
a = 1
b = a + 1
print a
```

如果希望完整记录实验过程，可以使用 log 功能。

- %logstart: 启动记录器
- %logstop: 停止
- %logon, %logoff: 开启或停止记录
- %logstate: 查看记录器状态

格式: %logstart [-o|-r|-t] [log_name [log_mode]]
参数: -o 保存 Out 内容； -r 按输入格式保存; -t 加入时间戳。
模式: append 追加; backup 将已有的记录文件改名; global 使用 $HOME 目录下的单个记录文件; over 覆盖; rotate 创建 name.1~ name.2~ 循环日志文件。

```
In [37]: %logstart -r -t test.log over   # 启动记录器，over 表示 overwrite
Activating auto-logging. Current session state plus future input saved.
Filename       : test.log
Mode           : over
```

```
Output logging : False
Raw input log  : True
Timestamping   : True
State          : active

In [38]: a = 1                # 这些行会被记录下来

In [39]: b = 2

In [40]: c = a + b

In [41]: logoff               # 暂时关闭记录器，在 logon 之前的命令不会被记录。
Switching logging OFF

In [42]: d = sum([a, b, c])   # 被遗弃的卖姑娘的小火柴

In [43]: logon                # 再次开始记录
Switching logging ON

In [44]: d = 100

In [45]: logstop              # 停止记录

In [46]: cat test.log         # 看看记录文件的内容
# IPython log file

%logstart?
%logstart -r -t test.log over
# Fri, 03 Feb 2012 11:46:09
a = 1
# Fri, 03 Feb 2012 11:46:10
b = 2
# Fri, 03 Feb 2012 11:46:14
c = a + b
# Fri, 03 Feb 2012 11:46:23
logoff
# Fri, 03 Feb 2012 11:46:54
d = 100
# Fri, 03 Feb 2012 11:47:01
logstop
```

1.1.1.5 编辑

%edit 打开文本编辑器 (通常是 vi/vim)，参数可以是源文件名，或某个对象名。

- -n: 指定源文件行号；
- -x: 表示退出编辑器时不执行该源码。

%ed 是缩写别名。%pycat 高亮显示源码文件，%pfile 显示某个对象所在的源码文件。

```
In [47]: ed -x main.py
```

```
Editing...

In [48]: pycat main.py
# -*- coding:utf-8 -*-

import sys
import pprint

def test():
    print "Hello, World!"

def main():
    test()

if __name__ == "__main__":
    main()

In [49]: ed main.py           # 不使用 -x 参数，会导致源码被执行。
Editing... done. Executing edited code...
Hello, World!
```

1.1.1.6 查看

%who 和 %whos 用来查看当前环境下所有的变量。我们会发现 "ed main.py" 被执行后，其成员已经被 "导入" 到当前交互环境下。

```
In [50]: who
main   pprint  sys    test

In [51]: whos
Variable   Type        Data/Info
-------------------------------
main       function    <function main at 0x100b22230>
pprint     module      <module 'pprint' from '/S<...>ib/python2.7/pprint.pyc'>
sys        module      <module 'sys' (built-in)>
test       function    <function test at 0x10042f2a8>
```

1.1.1.7 运行

%run 用于执行一个 py 源码文件，有几个重要的参数。

- -n: 将要执行的模块 __name__ 设置为非 "__main__" 值 (比如 main)，这样就可以不执行常见的入口函数了。
- -i: 则是让模块直接使用当前交互环境的名字空间，如此在源码中可以直接使用和修改交互环境下的变量，对于设置测试场景调试模块很有用，在后面我们会经常用到。
- -d: 启用 pdb，直接进入调试状态。
- -t: 输出执行时间，相当于 %time。
- -p: 启用 profiler 输出性能统计。

```
In [52]: run main.py
Hello, World!

In [53]: run -t main.py
Hello, World!

IPython CPU timings (estimated):
  User   :       0.00 s.
  System :       0.00 s.
Wall time:       0.00 s.
```

1.1.1.8 测试

除了使用 pdb 进行代码调试外，我们还需要对完成的算法函数进行性能测试。

- %prun: 和 "%run -p" 相同，输出 profiler 信息。
- %time: 和 "%run -t" 相同，查看执行代码所消耗时间。
- %timeit: 比 "%time" 更强大一些，可以指定循环的次数。参数 -n 表示单次测试内执行目标的次数，-r 则表示循环测试的次数。

```
In [54]: import time

In [55]: def test(n):
   ....:     for i in xrange(n):
   ....:         time.sleep(0.001)
   ....:

In [56]: %prun test(10)
         13 function calls in 0.012 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
       10    0.012    0.001    0.012    0.001 {time.sleep}
        1    0.000    0.000    0.012    0.012 <ipython-input-42-98d0554b9502>:1(test)
        1    0.000    0.000    0.012    0.012 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}

In [57]: %time test(10)
CPU times: user 0.00 s, sys: 0.00 s, total: 0.00 s
Wall time: 0.01 s

In [58]: %timeit -n 100 -r 5  test(1)
100 loops, best of 5: 1.15 ms per loop
```

%pdb 启动调试模式，这在后面我们会详细说明。
%reset 将删除当前环境所有变量，清理出一个干净的战场。不过不会改变序号。

1.1.2 Pdb / iPdb

还在用 "print" 做调试?

对于熟悉 gdb 的人来说，命令行调试器并不比 IDE 的调试器逊色多少，甚至尤有过之。况且大部分的 IDE Debugger 也是调用命令行的调试工具。

ipdb 和 pdb 的差异，相当于 IPython 和 Python 交互环境的差异。ipdb 提供代码高亮、代码完成，以及更好的异常处理能力。

在代码中启动 ipdb:

```
import ipdb; ipdb.set_trace()
```

在命令行启动 ipdb:

```
$ python -m ipdb main.py
> /Users/test/python/main.py(4)<module>()
      3
----> 4 import sys
      5 import pprint
```

在 IPython 中使用 %pdb 打开 "Automatic pdb calling”，在出错时自动进入调试状态。

```
In [1]: cat main.py
#!/usr/bin/env python
# -*- coding:utf-8 -*-

import sys
import pprint

def test(*args, **kwargs):
    print args
    print kwargs
    raise Exception("Error!")

def main():
    test(1, "a", x = 10)

if __name__ == "__main__":
    main()

In [2]: pdb                   # 打开 pdb
Automatic pdb calling has been turned ON

In [3]: run main.py           # 运行 main.py，在 test() 中会抛出异常。将在异常发生时进入调试状态。
(1, 'a')
{'x': 10}
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)

/Users/test/python/main.py in <module>()
     15
```

```
     16 if __name__ == "__main__":
---> 17     main()

/Users/test/python/main.py in main()
     11
     12 def main():
---> 13     test(1, "a", x = 10)
     14

/Users/test/python/main.py in test(*args, **kwargs)
      8     print args
      9     print kwargs
---> 10     raise Exception("Error!")
     11
     12 def main():

Exception: Error!
> /Users/test/python/main.py(10)test()
      9     print kwargs
---> 10     raise Exception("Error!")
     11
```

运行 "%run -d" 直接进入调试状态。

```
In [3]: run -d main.py
Breakpoint 1 at /Users/test/python/main.py:4
NOTE: Enter 'c' at the ipdb>  prompt to start your script.
> <string>(1)<module>()

ipdb> c        # 注意上面的提示!
> /Users/test/python/main.py(4)<module>()
      3
1---> 4 import sys
      5 import pprint
ipdb>
```

常用的调试命令：

命令	说明
a, args	显示函数参数。
b, break	设置或显示断点列表。
bt, w, where	显示 stack trace。
c, cont, continue	继续执行，直到下一个有效断点。
cl, clear	清除断点。
d, down, u, up	在 call stack frame 列表间移动。
disable, enable	禁用或启用断点。

命令	说明
exit, q, quit	退出 pdb。
h, help	查看命令列表或详细使用方法。
ignore	暂时忽略某个断点，可设置忽略次数。
l, list	显示源码。
n, next	继续执行下一行代码。
p, pp	显示变量或表达式的结果。pp = pretty-print。
r, return	继续执行，直到当前函数结束。
run, restart	运行，重新运行。
s, step	继续执行下一行代码，继续函数内部。
tbreak	创建临时断点，命中后失效。
whatis	查看对象类型。

试试看：

```
In [4]: run -d main.py

Breakpoint 1 at /Users/test/python/main.py:4
NOTE: Enter 'c' at the ipdb>  prompt to start your script.
> <string>(1)<module>()

ipdb> c                # 注意上面的提示信息。
> /Users/test/python/main.py(4)<module>()
      3
1---> 4 import sys
      5 import pprint

ipdb> l                # 查看源码
      1 #!/usr/bin/env python

      2 # -*- coding:utf-8 -*-

      3
1---> 4 import sys
      5 import pprint
      6
      7 def test(*args, **kwargs):
      8     print args
      9     print kwargs
     10     raise Exception("Error!")
     11

ipdb> l                # 继续查看后续源码
     12 def main():
```

```
     13     test(1, "a", x = 10)
     14
     15
     16 if __name__ == "__main__":
     17     main()

ipdb> b 13            # 按行号设置断点
Breakpoint 2 at /Users/test/python/main.py:13

ipdb> b test          # 按函数设置断点
Breakpoint 3 at /Users/test/python/main.py:7

ipdb> b               # 显示所有的断点
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at /Users/test/python/main.py:4
       breakpoint already hit 1 time
2   breakpoint   keep yes   at /Users/test/python/main.py:13
3   breakpoint   keep yes   at /Users/test/python/main.py:7

ipdb> r               # 开始执行，去命中第一个断点吧。
> /Users/test/python/main.py(13)main()
     12 def main():
2--> 13     test(1, "a", x = 10)
     14

ipdb> c               # 继续，在下一个断点折腾。
> /Users/test/python/main.py(8)test()
3     7 def test(*args, **kwargs):
----> 8     print args
      9     print kwargs

ipdb> a               # 显示函数参数
args = (1, 'a')
kwargs = {'x': 10}

ipdb> w               # 查看 CallStack
  /System/Library/Frameworks/.../python2.7/bdb.py(383)run()
    381             cmd = cmd+'\n'
    382         try:
--> 383             exec cmd in globals, locals
    384         except BdbQuit:
    385             pass

  <string>(1)<module>()

  /Users/test/python/main.py(17)<module>()
2    13     test(1, "a", x = 10)
     14
     15
     16 if __name__ == "__main__":
---> 17     main()

  /Users/test/python/main.py(13)main()
     11
```

```
     12 def main():
2--> 13     test(1, "a", x = 10)
     14
     15

> /Users/test/python/main.py(8)test()
      6
3     7 def test(*args, **kwargs):
----> 8     print args
      9     print kwargs
     10     raise Exception("Error!")

ipdb> u               # 上一个 StackFrame
> /Users/test/python/main.py(13)main()
     12 def main():
2--> 13     test(1, "a", x = 10)
     14

ipdb> d               # 回到最后一个 StackFrame
> /Users/test/python/main.py(8)test()
3     7 def test(*args, **kwargs):
----> 8     print args
      9     print kwargs

ipdb> r               # 继续运行
(1, 'a')
{'x': 10}
--Return--
None
> /Users/test/python/main.py(10)test()
      9     print kwargs
---> 10     raise Exception("Error!")
     11

ipdb> q               # 退出。
```

如果没有打开 IPython %pdb，可以使用 ipdb.pm 函数回到错误现场。

```
In [5]: pdb
Automatic pdb calling has been turned OFF

In [6]: run main.py
(1, 'a')
{'x': 10}
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)

/Users/test/python/main.py in <module>()
     15
     16 if __name__ == "__main__":
---> 17     main()
```

```
/Users/test/python/main.py in main()
     11
     12 def main():
---> 13     test(1, "a", x = 10)
     14
     15

/Users/test/python/main.py in test(*args, **kwargs)
      8     print args
      9     print kwargs
---> 10     raise Exception("Error!")
     11
     12 def main():

Exception: Error!

In [7]: ipdb.pm()             # 回到错误现场
> /Users/test/python/main.py(10)test()
      9     print kwargs
---> 10     raise Exception("Error!")
     11

ipdb> a                       # 勘查现场
args = (1, 'a')
kwargs = {'x': 10}
```

我发现一些不习惯使用调试器的人，总是写几行代码就 print 输出一回，查看输出结果，再回到编辑器中继续修改。来来回回折腾，效率很低。远不如用 IPython + ipdb 配合，交互式执行环境让我们随意发挥，实时观测结果。满意了，就将代码整理到源文件中。不满意，随时 del 或 reset。

1.2 常用函数

应该熟悉下面几个函数或操作符的使用，这有助于我们作出准确的判断和分析。

- id: 返回对象标识，在 CPython 里通常是目标对象的内存地址。
- type: 获取类型对象 (Type)。
- is: 判断两个变量 (名字) 是否指向同一个对象。
- dir: 返回对象名字空间字典。
- sys.getsizeof: 返回对象大小，通常和 "id" 配合使用，以便使用 gdb 查看对象的内存状态。
- sys._get_frame: 返回当前线程 Call Stack 列表中某个堆栈帧 (Stack Frame) 对象。
- sys._current_frames: 返回所有线程当前堆栈帧对象，常用于调试死锁。

我们看一个简单的例子。

```
In [1]: help(id)
Help on built-in function id in module __builtin__:

id(...)
    id(object) -> integer
```

```
    Return the identity of an object.  This is guaranteed to be unique among
    simultaneously existing objects.  (Hint: it's the object's memory address.)


In [2]: hex(id(int))
Out[2]: '0x103b0a180'

In [3]: a = 0x1234

In [4]: hex(id(a))
Out[4]: '0x7f83026a3950'

In [5]: hex(id(type(a)))
Out[5]: '0x103b0a180'

In [6]: import sys; sys.getsizeof(a)
Out[6]: 24
```

我们使用 gdb 查看 "hex(id(a))" 输出结果所在位置的内存，你会看到 Python 整数对象的内存布局。至于其具体的含义我们会在后面的章节作出详细说明。

```
$ gdb --pid=`pidof python`
GNU gdb 6.3.50-20050815 (Apple version gdb-1705) (Fri Jul  1 10:50:06 UTC 2011)

(gdb) x/3xg 0x7f83026a3950    # 使用您机器上返回的地址替代
0x7f83026a3950:       0x0000000000000001    0x0000000103b0a180
0x7f83026a3960:       0x0000000000001234
```

注：gdb x 指令用于查看内存数据，3xg 表示输出 3 组 8 字节单位的十六进制数据

Python 支持操作符重载，因此 "==" 只适合用来做值相等判断。

```
In [7]: class A(object):
   ....:     def __eq__(self, o):
   ....:         return True
   ....:

In [8]: a, b = A(), object()

In [9]: a == b
Out[9]: True

In [10]: a is b
Out[10]: False

In [11]: c = a; a is c
Out[11]: True
```

再看 _get_frame 的一个使用案例：通过检查调用函数名来实现权限管理。

```
In [12]: import sys
```

```
In [13]: def test():
   ....:     if sys._getframe(1).f_code.co_name != "a":
   ....:         print "Error!"
   ....:     else:
   ....:         print "OK!"
   ....:

In [14]: def a():
   ....:     test()
   ....:

In [15]: a()
OK!

In [16]: def b():
   ....:     test()
   ....:

In [17]: b()
Error!
```

1.3 环境初始化

在启动 Python 后，我们可以直接使用内置函数和类型……
Python 的 list、tuple、dict 是多么的……
所有载入的模块都存储在 sys.modules 中……
通过 sys.path 搜索目标模块……

好吧，你已经知道这些了，也折腾过很多遍。难道你不好奇，为什么会这样？怎么来的？好奇心不仅仅是 "攻城师" 的必备素质，还能提高我们的生存技能。伙计，美丽的东西，往往有着复杂的组成和演化。随着你在 Python 城池里生活得越久，你就会越发明白了解其初始化和执行过程是多么的有必要。

从哪开始呢？不管是进入 Python 交互环境，还是在命令行用 "python xxx.py" 执行一个文件，亦或者是那个 "化妆版" 的 IPython，其本质上都是启动了一个 Python 进程。

```
$ cat `which ipython`

#!/usr/bin/python
# EASY-INSTALL-ENTRY-SCRIPT: 'ipython==0.12','console_scripts','ipython'
__requires__ = 'ipython==0.12'
import sys
from pkg_resources import load_entry_point

sys.exit(
   load_entry_point('ipython==0.12', 'console_scripts', 'ipython')()
)
```

打开你的 CPython 源码 (不同版本的源码可能存在一些差异。为阅读方便，相关源码皆为节选)，宝藏入口就在下面这段代码里:

Modules/main.c

```
int Py_Main(int argc, char **argv)
{
    // 处理参数，帮助信息等等...
    ... ...

    // 初始化整个环境。
    Py_Initialize();

    ... ...

    // 进入虚拟机执行流程。
    if (command) {
        // 运行 "python -c cmd"。
        sts = PyRun_SimpleStringFlags(command, &cf) != 0;
        ...
    } else if (module) {
        // 运行 "python -m mod"。
        sts = RunModule(module, 1);
        ...
    }
    else {
        // 交互模式，并运行启动文件。
        if (filename == NULL && stdin_is_interactive) {
            ...
            RunStartupFile(&cf);
        }

        ... ...

        // 运行 py 文件。
        sts = PyRun_AnyFileExFlags(
              fp,
              filename == NULL ? "<stdin>" : filename,
              filename != NULL, &cf) != 0;
    }

    ... ...

    Py_Finalize();

    ... ...
}
```

显然这和官方帮助文档中 "Embedding Python in Another Application" 示例类似。

```
#include <Python.h>

int main(int argc, char *argv[])
```

```
{
    Py_Initialize();
    PyRun_SimpleString("from time import time,ctime\n"
                       "print 'Today is',ctime(time())\n");
    Py_Finalize();
    return 0;
}
```

Py_Initialize() 是我们跟踪的第一号目标，在这里 Python 完成了几乎全部的初始化工作。

Python/pythonrun.c

```
void Py_Initialize(void)
{
    Py_InitializeEx(1);
}

void Py_InitializeEx(int install_sigs)
{
    ...

    // 解释器状态对象
    interp = PyInterpreterState_New();
    ...

    // 线程状态对象
    tstate = PyThreadState_New(interp);

    // 创建所有的内置类型对象
    _Py_ReadyTypes();

    // 这些特殊的家伙有些小秘密需要照顾一下
    if (!_PyFrame_Init())
        Py_FatalError("Py_Initialize: can't init frames");

    if (!_PyInt_Init())
        Py_FatalError("Py_Initialize: can't init ints");

    if (!_PyLong_Init())
        Py_FatalError("Py_Initialize: can't init longs");

    if (!PyByteArray_Init())
        Py_FatalError("Py_Initialize: can't init bytearray");

    _PyFloat_Init();

    // 为解释器对象属性分配内存。
    interp->modules = PyDict_New();
    interp->modules_reloading = PyDict_New();

    // 创建 __builtin__ 模块
    bimod = _PyBuiltin_Init();
    interp->builtins = PyModule_GetDict(bimod);
```

```
    // 创建 sys 模块
    sysmod = _PySys_Init();
    interp->sysdict = PyModule_GetDict(sysmod);

    // 设置 sys.path 模块搜索路径，但不包括 site-packages。
    PySys_SetPath(Py_GetPath());

    // sys.modules，终于看见你了。
    PyDict_SetItemString(interp->sysdict, "modules",
                         interp->modules);

    // 设置导入机制
    _PyImport_Init();

    ... ...

    // 初始化内建 Exception
    _PyExc_Init();

    ... ...

    _PyImportHooks_Init();

    // 终于看到 __main__ module 了
    initmain(); /* Module __main__ */

    ...

    // 设置 site module
    if (!Py_NoSiteFlag)
        initsite(); /* Module site */

    ... ...
}
```

通过 Py_Main 和 Py_InitializeEx 这两个函数，基本就能摸清 Python 虚拟机的启动和初始化流程。 在 Py_InitializeEx 中，Python 创建并加载了几个重要的基础模块: __builtin__、sys、site。并为后续的字节码执行准备好了所需的内置类型对象。

接下来，我们看看这几个主要步骤都做了些什么具体工作。

1.3.1 State

PyInterpreterState 可以看成 Python 虚拟机进程的全局状态存储器。你所熟悉的 __builtin__、sys.modules 内容的实际存储位置就是它了。

Python/pystate.c

```
PyInterpreterState * PyInterpreterState_New(void)
{
    PyInterpreterState *interp = (PyInterpreterState *)malloc(sizeof(PyInterpreterState));
```

```
    if (interp != NULL) {
        HEAD_INIT();

        ... ...

        interp->modules = NULL;
        interp->modules_reloading = NULL;
        interp->sysdict = NULL;
        interp->builtins = NULL;

        ... ...
    }

    return interp;
}
```

而 PyThreadState 则存储了线程相关的状态，其中最重要的就是 StackFrame 和 TraceBack 了。

Python/pystate.c

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

static PyThreadState * new_threadstate(PyInterpreterState *interp, int init)
{
    PyThreadState *tstate = (PyThreadState *)malloc(sizeof(PyThreadState));

    if (_PyThreadState_GetFrame == NULL)
        _PyThreadState_GetFrame = threadstate_getframe;

    if (tstate != NULL) {
        tstate->interp = interp;

        tstate->frame = NULL;
        tstate->recursion_depth = 0;

        ...

        tstate->exc_type = NULL;
        tstate->exc_value = NULL;
        tstate->exc_traceback = NULL;
        ... ...
    }

    return tstate;
}
```

通过 PyInterpreterState 和 PyThreadState，Python 虚拟机完成了对 Process、Thread 的初始化，并建立了相关的状态联系，这是代码执行所必需的。

1.3.2 Types

_Py_ReadyTypes 函数完成了所有内置类型对象初始化工作。只有准备好了这些类型 (Type) 对象，我们才能创建对象实例 (instance)。

Objects/object.c

```
void _Py_ReadyTypes(void)
{
    if (PyType_Ready(&PyType_Type) < 0)
        Py_FatalError("Can't initialize type type");

    if (PyType_Ready(&PyBool_Type) < 0)
        Py_FatalError("Can't initialize bool type");

    if (PyType_Ready(&PyString_Type) < 0)
        Py_FatalError("Can't initialize str type");

    ... ...
}
```

要 "准备" 好一个类型对象都需要做些什么?

Objects/typeobject.c

```
int PyType_Ready(PyTypeObject *type)
{
    ... ...

    // 初始化 base object
    base = type->tp_base;
    ...

    bases = type->tp_bases;

    // 初始化 type.__dict__，用于存储类成员。
    dict = type->tp_dict;

    // 添加基本的类型成员，包括操作符、方法等。
    if (add_operators(type) < 0)
        goto error;
    if (type->tp_methods != NULL) {
        if (add_methods(type, type->tp_methods) < 0)
            goto error;
    }
    if (type->tp_members != NULL) {
        if (add_members(type, type->tp_members) < 0)
            goto error;
    }
    if (type->tp_getset != NULL) {
        if (add_getset(type, type->tp_getset) < 0)
            goto error;
    }
```

```
    // 处理 mro (Method Resolution Order) 属性，这关系到如何从基类查找成员。
    if (mro_internal(type) < 0) {
        goto error;
    }

    ... ...

    // 处理 __doc__
    if (PyDict_GetItemString(type->tp_dict, "__doc__") == NULL) {
        ... ...
    }

    ... ...

    // 添加所有 subclass 信息
    bases = type->tp_bases;
    n = PyTuple_GET_SIZE(bases);
    for (i = 0; i < n; i++) {
        PyObject *b = PyTuple_GET_ITEM(bases, i);
        if (PyType_Check(b) &&
            add_subclass((PyTypeObject *)b, type) < 0)
            goto error;
    }

    ... ...
}
```

为提高性能和效率，Python 还对某些类型做了特殊的处理，比如对象缓存。我们以 PyInt 为例，看看有什么小动作。

Objects/intobject.c

```
#define NSMALLPOSINTS           257
#define NSMALLNEGINTS           5

int _PyInt_Init(void)
{
    // 看不明白？没关系，第二章有详细的分析。
    ... ...
    for (ival = -NSMALLNEGINTS; ival < NSMALLPOSINTS; ival++) {
        ...
        v = free_list;
        free_list = (PyIntObject *)Py_TYPE(v);
        PyObject_INIT(v, &PyInt_Type);
        v->ob_ival = ival;
        small_ints[ival + NSMALLNEGINTS] = v;
    }
    return 1;
}
```

如你所见，它会缓存 [-5, 257) 之间的所有小整数对象。

1.3.3 __builtin__ module

__builtin__ 模块 "存储" 了所有内置类型和内置函数对象。任何时候，任何地点都能访问这个特殊的 "全局" 模块。

Python/bltinmodule.c

```
// 内置函数表
static PyMethodDef builtin_methods[] = {
    {"__import__",      (PyCFunction)builtin___import__, METH_VARARGS | METH_KEYWORDS, import_doc},
    {"abs",             builtin_abs,        METH_O, abs_doc},
    {"all",             builtin_all,        METH_O, all_doc},
    {"any",             builtin_any,        METH_O, any_doc},
    ... ...
    {"zip",             builtin_zip,        METH_VARARGS, zip_doc},
    {NULL,              NULL},
};

PyObject * _PyBuiltin_Init(void)
{
    // 初始化 __builtin__ 模块，并添加 builtin_methos 表中的所有内置函数。
    mod = Py_InitModule4("__builtin__", builtin_methods,
                         builtin_doc, (PyObject *)NULL,
                         PYTHON_API_VERSION);
    ... ...

    // 添加所有的内置类型对象。看到你熟悉的 int、dict、tuple 了吗?
    SETBUILTIN("None",                  Py_None);
    SETBUILTIN("basestring",            &PyBaseString_Type);
    SETBUILTIN("bool",                  &PyBool_Type);
    ... ...
    SETBUILTIN("dict",                  &PyDict_Type);
    SETBUILTIN("int",                   &PyInt_Type);
    ... ...
    SETBUILTIN("tuple",                 &PyTuple_Type);
    SETBUILTIN("type",                  &PyType_Type);
    SETBUILTIN("xrange",                &PyRange_Type);

    ... ...
}
```

在 Python 源码中有多个 Py_InitModule 函数，它们内部都是通过 PyImport_AddModule 来完成模块对象的创建和维护工作。

Python/import.c

```
PyObject * PyImport_AddModule(const char *name)
{
    // 获取 PyInterpreterState modules，也就是 sys.modules。
    PyObject *modules = PyImport_GetModuleDict();
    PyObject *m;

    // 如果模块已经存在，就直接返回好了。
```

```
    if ((m = PyDict_GetItemString(modules, name)) != NULL && PyModule_Check(m))
        return m;

    // 创建新的模块对象。
    m = PyModule_New(name);
    if (m == NULL) return NULL;

    // 将模块对象添加到 sys.modules。
    if (PyDict_SetItemString(modules, name, m) != 0) {
        Py_DECREF(m);
        return NULL;
    }
    Py_DECREF(m); /* Yes, it still exists, in modules! */

    return m;
}

PyObject * PyImport_GetModuleDict(void)
{
    PyInterpreterState *interp = PyThreadState_GET()->interp;
    if (interp->modules == NULL)
        Py_FatalError("PyImport_GetModuleDict: no module dictionary!");
    return interp->modules;
}
```

Objects/moduleobject.c

```
PyObject * PyModule_New(const char *name)
{
    PyModuleObject *m;
    PyObject *nameobj;

    // 创建模块对象
    m = PyObject_GC_New(PyModuleObject, &PyModule_Type);

    // 设置 __name__、__doc__ ...
    nameobj = PyString_FromString(name);
    m->md_dict = PyDict_New();

    if (PyDict_SetItemString(m->md_dict, "__name__", nameobj) != 0)
        goto fail;
    if (PyDict_SetItemString(m->md_dict, "__doc__", Py_None) != 0)
        goto fail;
    if (PyDict_SetItemString(m->md_dict, "__package__", Py_None) != 0)
        goto fail;

    ... ...
}
```

1.3.4 sys module

sys 应该是 Python 里面最重要的基础模块了。

Python/sysmodule.c

```
// sys module 函数表
static PyMethodDef sys_methods[] = {
    ... ...
    {"displayhook",     sys_displayhook, METH_O, displayhook_doc},
    {"exc_info",        sys_exc_info, METH_NOARGS, exc_info_doc},
    {"exc_clear",       sys_exc_clear, METH_NOARGS, exc_clear_doc},
    {"excepthook",      sys_excepthook, METH_VARARGS, excepthook_doc},
    {"exit",            sys_exit, METH_VARARGS, exit_doc},
    ... ...
    {"getrefcount",     (PyCFunction)sys_getrefcount, METH_O, getrefcount_doc},
    {"_getframe", sys_getframe, METH_VARARGS, getframe_doc},
    ... ...
};

// sys.__doc__ 帮助信息
PyDoc_VAR(sys_doc) =
...
"This module provides access to some objects used or maintained by the\n\
interpreter and to functions that interact strongly with the interpreter.\n\
\n\
Dynamic objects:\n\
...

PyObject * _PySys_Init(void)
{
    // 创建 sys module，并添加所有的方法。
    m = Py_InitModule3("sys", sys_methods, sys_doc);

    ... ...

    // sys.stdin, sys.stdout, sys.stderr，别说没用过啊。
    sysin = PyFile_FromFile(stdin, "<stdin>", "r", NULL);
    sysout = PyFile_FromFile(stdout, "<stdout>", "w", _check_and_flush);
    syserr = PyFile_FromFile(stderr, "<stderr>", "w", _check_and_flush);

    PyDict_SetItemString(sysdict, "stdin", sysin);
    PyDict_SetItemString(sysdict, "stdout", sysout);
    PyDict_SetItemString(sysdict, "stderr", syserr);

    ... ...

    // 设置一些零七八碎的信息。
    SET_SYS_FROM_STRING("version", PyString_FromString(Py_GetVersion()));
    SET_SYS_FROM_STRING("copyright", PyString_FromString(Py_GetCopyright()));
    SET_SYS_FROM_STRING("platform", PyString_FromString(Py_GetPlatform()));
    SET_SYS_FROM_STRING("maxsize", PyInt_FromSsize_t(PY_SSIZE_T_MAX));
    SET_SYS_FROM_STRING("maxint", PyInt_FromLong(PyInt_GetMax()));

    ... ...

    return m;
}
```

sys module 有个很重要的属性 path，这里面存储了 "import <module>" 所需的搜索路径。

```
void PySys_SetPath(char *path)
{
    PyObject *v;
    if ((v = makepathobject(path, DELIM)) == NULL)
        Py_FatalError("can't create sys.path");
    if (PySys_SetObject("path", v) != 0)
        Py_FatalError("can't assign sys.path");
    Py_DECREF(v);
}

static PyObject * makepathobject(char *path, int delim)
{
    // 函数很简单，就是依照 delim 将 path 分割，存储到一个 PyList 对象中。
}
```

Modules/getpath.c

```
char * Py_GetPath(void)
{
    if (!module_search_path)
        calculate_path(); // 好长的一个函数，有兴趣的自己去翻翻看。这里就不列了，省得说我凑字。
    return module_search_path;
}
```

1.3.5 __main__ module

不容易啊，终于看到这个 "__main__" 了，是不是下意识想到 "if __name__ == '__main__': " 了。

initmain 很简单，除了创建模块对象，剩余的工作就是将 __builtin__ 改名为 "__builtins__" 添加到模块名字空间 (__dict__) 中。

等等，这个 __main__ 模块和我们自己写的项目入口文件 (.py) 有什么关系？下面的代码里没有看到任何执行 py 代码的意思。别着急，这个 __main__ 只是为虚拟机执行 py 做一些准备工作。

Python/pythonrun.c

```
static void initmain(void)
{
    PyObject *m, *d;
    m = PyImport_AddModule("__main__");
    ...
    d = PyModule_GetDict(m);
    if (PyDict_GetItemString(d, "__builtins__") == NULL) {
        PyObject *bimod = PyImport_ImportModule("__builtin__");
        if (bimod == NULL || PyDict_SetItemString(d, "__builtins__", bimod) != 0)
            Py_FatalError("can't add __builtins__ to __main__");
        ...
    }
}
```

1.3.6 site module

site.py 有什么用？好像很多 Pythoner 都没有认真地去探究过。

Python/pythonrun.c

```
static void initsite(void)
{
    PyObject *m;
    m = PyImport_ImportModule("site");

    ... ...
}
```

还记得你用 easy_install 安装到 site-packages 目录下的那些第三方库吗？site 的作用就是把它们加入到 sys.path 搜索路径里面。当你 "import pymongo" 时，确保能正确找到 "pymongo-xxx.egg" 这样的文件或目录。

site.py

```
"""Append module search paths for third-party packages to sys.path.

****************************************************************
* This module is automatically imported during initialization. *
****************************************************************
"""

def main():
    ...
    known_paths = addusersitepackages(known_paths)
    known_paths = addsitepackages(known_paths)

    ...
    setencoding()

    ... ...

    // 坑爹的 sys.setdefaultencoding 就是在这被咔嚓掉的！！！
    # Remove sys.setdefaultencoding() so that users cannot change the
    # encoding after initialization.  The test for presence is needed when
    # this module is run as a script, because this code is executed twice.
    if hasattr(sys, "setdefaultencoding"):
        del sys.setdefaultencoding

main()
```

这个被 site 砍掉的 sys.setdefaultencoding 对我们这些非英文 "码农" 造成 "很大" 麻烦。我都有些厌倦了在 package __init__.py 里写 "reload(sys) ..." 了。

1.3.7 Run

不管是交互环境还是执行 py 文件，最终都会通过 run_mod 来启动虚拟机。 PyRun_xxx 函数会将 __main__.__dict__ 作为 globals、locals 名字空间传递给 run_mod，这其中就包括了念念不忘的 __name__ 和 __binutils__。这也是你可以在 py 中做 "if __name__ == '__main__': " 的原因。

Python/pythonrun.c

```
int PyRun_AnyFileExFlags(FILE *fp, const char *filename, int closeit, PyCompilerFlags *flags)
{
    ...
    if (Py_FdIsInteractive(fp, filename)) {
        int err = PyRun_InteractiveLoopFlags(fp, filename, flags);
        ... ...
    }
    else
        return PyRun_SimpleFileExFlags(fp, filename, closeit, flags);
}

int PyRun_SimpleFileExFlags(FILE *fp, const char *filename, int closeit, PyCompilerFlags *flags)
{
    ... ...

    // 获取 __main__ module 和 __main__.__dict__
    m = PyImport_AddModule("__main__");
    d = PyModule_GetDict(m);

    // 对 pyc/pyo、py 文件作出判断处理
    //   __main__.__dict__ 作为 globals/locals 被传递过去了，其中就包含 __name__: __main__
    if (maybe_pyc_file(fp, filename, ext, closeit)) {
        v = run_pyc_file(fp, filename, d, d, flags);
    } else {
        v = PyRun_FileExFlags(fp, filename, Py_file_input, d, d, closeit, flags);
    }

    ...

    // 设置 __file__ 属性
    if (set_file_name && PyDict_DelItemString(d, "__file__"))
        PyErr_Clear();
    return ret;
}

PyObject * PyRun_FileExFlags(FILE *fp, const char *filename, int start, PyObject *globals,
                  PyObject *locals, int closeit, PyCompilerFlags *flags)
{
    ... ...
    // globals, locals 这些名字空间都被带入。
    ret = run_mod(mod, filename, globals, locals, flags, arena);
    ... ...
}
```

run_mod 基于 Abstract Syntax Trees 分析源码并编译成字节码 (Byte Code)，并保存成二进制文件。后面的 "编译" 章节里对此有详细的描述。

Python/pythonrun.c

```
static PyObject * run_mod(mod_ty mod, const char *filename, PyObject *globals, PyObject *locals,
         PyCompilerFlags *flags, PyArena *arena)
{
    ... ...

    // 编译字节码，创建 PyCodeObject。
    co = PyAST_Compile(mod, filename, flags, arena);

    // 执行字节码指令。
    v = PyEval_EvalCode(co, globals, locals);

    return v;
}
```

在无比巨大的虚拟机核心执行函数 PyEval_EvalFrameEx 中，一个超级夸张的 "for + switch" 流程循环执行字节码指令 (看着头晕)。我们可以看到字节码指令和底层函数的对应关系，这有助于了解字节码指令的底层细节。

Python/ceval.c

```
PyObject * PyEval_EvalCode(PyCodeObject *co, PyObject *globals, PyObject *locals)
{
    return PyEval_EvalCodeEx(co, globals, locals, (PyObject **)NULL, 0,
                             (PyObject **)NULL, 0, (PyObject **)NULL, 0, NULL);
}

PyObject * PyEval_EvalCodeEx(PyCodeObject *co, PyObject *globals, PyObject *locals,
           PyObject **args, int argcount, PyObject **kws, int kwcount,
           PyObject **defs, int defcount, PyObject *closure)
{
    ... ...

    // 折腾一大堆东西，就是为了这句话。
    retval = PyEval_EvalFrameEx(f,0);

    ...
    return retval;
}

PyObject * PyEval_EvalFrameEx(PyFrameObject *f, int throwflag)
{
    ... ...

    // 很熟悉的东西吧，嘿嘿……
    co = f->f_code;
    names = co->co_names;
    consts = co->co_consts;
```

```
    fastlocals = f->f_localsplus;
    freevars = f->f_localsplus + co->co_nlocals;

    ... ...

    // 虚拟机主循环
    for (;;) {
        // 看到字节码指令了吗？可以在官方文档的 dis 模块中找到详细说明。
        switch (opcode) {

        case NOP:
            goto fast_next_opcode;

        case LOAD_FAST:
            x = GETLOCAL(oparg);
            if (x != NULL) {
                Py_INCREF(x);
                PUSH(x);
                goto fast_next_opcode;
            }
            ...
            break;

        case LOAD_CONST:
            x = GETITEM(consts, oparg);
            Py_INCREF(x);
            PUSH(x);
            goto fast_next_opcode;

        // ... 忒长了，用来凑字不错。就怕被人打。 ...

        } /* switch */

        ... ...
    } /* main loop */

    ... ...
}
```

执行流程到了这一步，总算看到属于我们的逻辑开始发挥作用了。

1.3.8 Finalize

凡事都得有始有终，在虚拟机执行完用户逻辑，准备结束进程前，还有些清理工作需要完成。

Python/pythonrun.c

```
void Py_Finalize(void)
{
    // 避免多次调用
    if (!initialized) return;

    // 等待线程结束，注意不包括背景线程。
```

```
    wait_for_thread_shutdown();

    // 调用退出函数，相信很多人都用过 atexit。
    call_sys_exitfunc();

    // 将初始化状态设为 False。
    initialized = 0;

    // 让解释器结束工作
    tstate = PyThreadState_GET();
    interp = tstate->interp;
    PyOS_FiniInterrupts();

    // 很多类型都有自己的缓存操作，这些需要清理。
    PyType_ClearCache();

    // 都准备结束进程了，内存整个被操作系统回收，还清理干嘛?
    //   回收内存不是目的，主要是某些对象的 __del__ 和弱引用 callback 需要调用，这也属于正常逻辑的一部分。
    //   如果在 __del__ 和 wearkref.callback 里有 import 操作，这回就麻烦大了，前面已经让某些主放假回家了，
    //     保不齐就搞得天下大乱啊。
    PyGC_Collect();

#ifdef COUNT_ALLOCS
    // 多次回收，直到彻底打扫干净。
    //   在后面详细分析垃圾回收的章节里，你就会发现一次是完不成工作的。
    while (PyGC_Collect() > 0)
        /* nothing */;
#endif

    // 将所有导入的模块释放。
    PyImport_Cleanup();

#if 0
    // 又是垃圾释放。总之 GC 忒不容易了。
    PyGC_Collect();
#endif

    // 对 import 做个彻底了结。
    _PyImport_Fini();

    ... ...

    // 清除进程状态
    PyInterpreterState_Clear(interp);

    // 清除异常
    _PyExc_Fini();

#ifdef WITH_THREAD
    // 虽然 GIL 给我们带来不少麻烦，可还是要一视同仁不是。
    _PyGILState_Fini();
#endif /* WITH_THREAD */

    // 清除线程状态
```

```
    PyThreadState_Swap(NULL);
    PyInterpreterState_Delete(interp);

    // 这些家伙们私下里都有不少小金库，比如 int、float 都有 Block 对象池。
    //   清理干净......
    PyMethod_Fini();
    PyFrame_Fini();
    PyCFunction_Fini();
    PyTuple_Fini();
    PyList_Fini();
    PySet_Fini();
    PyString_Fini();
    PyByteArray_Fini();
    PyInt_Fini();
    PyFloat_Fini();
    PyDict_Fini();

#ifdef Py_USING_UNICODE
    /* Cleanup Unicode implementation */
    _PyUnicode_Fini();
#endif

    ... ...

    // 谁知道这些被 GC 和 Fini 清理的家伙，会不会再次注册退出函数啊。
    //   总之，都是苦逼……
    call_ll_exitfuncs();
}
```

看完这个 Py_Finalize 函数，最大的感触就是，结束进程也不是一件简单的事情。光 GC 和 exit func 就来来回回折腾了好几次。还有那些看着简单的 int、float 也不是好相与的主，和 str、dict、tuple 一样有着许多小秘密有待我们去发掘。

1.4 名字和名字空间

"name 'a' is not defined"，这个错误提示不陌生吧？为什么是 "name"，而不是 "variable" 呢？

```
In [1]: print a
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
NameError: name 'a' is not defined
```

在 Python 中，把变量名、函数名、类型名等等标识符都称为名字。名字总是在名字空间中与一个对象关联。名字本身并不知道目标的具体信息，它仅仅负责 "指路"。

```
In [2]: a = 1234

In [3]: globals()
Out[3]:
{'a': 1234, ...}
```

上面例子中，赋值语句创建了一个值为 1234 的整形对象，并将其与名字 "a" 关联起来。也就是说通过赋值语句，我们得到的是一个 (a, 1234) 这样的对象关联关系。这个关系存储在当前的名字空间 globals 中。没错，名字空间以 dict 实现。

通过名字访问对象时，就从该名字空间中查找其关联的目标对象。我们完全可以绕开赋值语句，直接在名字空间中添加名字和对象的关联。

```
In [4]: globals()["b"] = "Hello, World!"

In [5]: b
Out[5]: 'Hello, World!'
```

名字空间里不仅可以保存 "a: 1234"，实际上可以是任何类型。要知道 Python 将一切都当作对象，比如下面我们动态导入一个模块对象。

```
In [6]: reset
Once deleted, variables cannot be recovered. Proceed (y/[n])?  y

In [7]: globals()["sys"] = __import__("sys")

In [8]: whos
Variable   Type      Data/Info
------------------------------
sys        module    <module 'sys' (built-in)>

In [9]: sys.version
Out[9]: '2.7.1 (r271:86832, Jun 16 2011, 16:59:05) \n[GCC 4.2.1 (Based on Apple Inc. build 5658)
(LLVM build 2335.15.00)]'
```

1.4.1 名字空间

我们已经知道名字和对象的关联总是保存在特定的名字空间中，那么都有哪些名字空间呢?

- globals(): 每个 module 都有这样一个名字空间，存储了 "全局对象"。
- locals(): 当前作用域（通常是函数或方法）的本地名字空间。
- <Instance>.__dict__: 对象实例成员的名字空间。
- <Type>.__dict__: 方法、静态字段等成员的名字空间。

globals() 指向当前模块的 __dict__ 属性。而在函数内部，locals() 指向 StackFrame.f_locals。

```
In [10]: import sys

In [11]: m = sys.modules[__name__]  # 获取当前模块对象

In [12]: m.__dict__ is globals()
Out[12]: True
```

```
In [13]: def test(*args, **kwargs):
   ....:     s = "Hello, World!"
   ....:     fls = sys._getframe(0).f_locals
   ....:     print fls is locals()
   ....:     print fls
   ....:     print locals()
   ....:

In [14]: test(1, x = "abc")
True
{'s': 'Hello, World!', 'args': (1,), 'fls': {...}, 'kwargs': {'x': 'abc'}}
{'s': 'Hello, World!', 'args': (1,), 'fls': {...}, 'kwargs': {'x': 'abc'}}
```

名字空间会导致一些特殊的问题，比如超出代码块的变量引用。

```
In [15]: def test():
   ...:     while True:
   ...:         s = "abc"
   ...:         break
   ...:     print s
   ...:     print locals()
   ...:

In [16]: test()
abc
{'s': 'abc'}
```

本地变量 s 只所以在超出 while 代码快后依然存在，是因为整个函数内部只使用同一个名字空间 StackFrame.f_locals。同样的做法，在 C 语言里面只会引发 undeclared 编译错误。

```
int main(int argc, char* argv[])
{
    while (true)
    {
        int x = 1234;
        printf("%d\n", x);
        break;
    }

    printf("%d\n", x);

    return EXIT_SUCCESS;
}
```

编译:

```
./src/main.c: In function ‘main’:
./src/main.c:21: error: ‘x’ undeclared (first use in this function)
./src/main.c:21: error: (Each undeclared identifier is reported only once
./src/main.c:21: error: for each function it appears in.)
```

1.4.2 名字访问方式

Python 语言具有静态作用域，不同层级的名字空间形成嵌套关系。作用域中的代码可以直接访问外层嵌套名字空间中的对象，内层的名字空间会隐藏外层名字空间中的同名对象。

直接访问又称为 "名字引用"，是指仅用名字访问对象，而不括任何限制前缀。直接访问时，Python 代码依照 LEGB 的顺序去查找名字，它们分别对应:

L: locals -> E: Enclosing Function -> G: globals -> B: builtins

与名字引用对应的还有 "属性引用"。访问对象属性时，必须使用 "." 成员修饰符。属性引用不受 LEGB 约束，采取了另外一种规则，我们会在后面章节详细解说这点。

```
In [17]: x = 1234

In [18]: __builtin__.s = "Hello, World!"

In [19]: def test():
   ....:     y = "abc"
   ....:     print x
   ....:     print y
   ....:     print s
   ....:

In [20]: test()
1234
abc
Hello, World!

In [21]: import dis

In [22]: dis.dis(test)
  2           0 LOAD_CONST               1 ('abc')
              3 STORE_FAST               0 (y)

  3           6 LOAD_GLOBAL              0 (x)
              9 PRINT_ITEM
             10 PRINT_NEWLINE

  4          11 LOAD_FAST                0 (y)
             14 PRINT_ITEM
             15 PRINT_NEWLINE

  5          16 LOAD_GLOBAL              1 (s)
             19 PRINT_ITEM
             20 PRINT_NEWLINE
             21 LOAD_CONST               0 (None)
             24 RETURN_VALUE
```

虽然你可能从没接触过 dis，但这种字节码还是很好理解的。x 和 s 都使用 LOAD_GLOBAL 指令从外层嵌套名字空间中查找。

由于每个模块被创建后，都会导入 __builtins__，所以你能在任何地方访问内置类型和函数。既然如此，__builtins__ 也是用来存储全局变量的好地方 (不推荐这么做)。

LEGB 规则有个限制，就是不能直接修改外层名字空间中的对象。为此，Python 引入了 global 和 local 关键字。可惜 local 只在 Python 3 中有效。

```
In [23]: x = 1234

In [24]: def set_local():
   ....:     x = "abc"
   ....:     print x
   ....:     print locals()
   ....:

In [25]: def set_global():
   ....:     global x
   ....:     x = "abc"
   ....:     print locals()
   ....:

In [26]: set_local()
abc
{'x': 'abc'}

In [27]: set_global()
{}

In [28]: x
Out[28]: 'abc'

In [29]: import dis

In [30]: dis.dis(set_local)
  2           0 LOAD_CONST               1 ('abc')
              3 STORE_FAST               0 (x)

  3           6 LOAD_FAST                0 (x)
              9 PRINT_ITEM
             10 PRINT_NEWLINE

  4          11 LOAD_GLOBAL              0 (locals)
             14 CALL_FUNCTION            0
             17 PRINT_ITEM
             18 PRINT_NEWLINE
             19 LOAD_CONST               0 (None)
             22 RETURN_VALUE

In [31]: dis.dis(set_global)
  3           0 LOAD_CONST               1 ('abc')
```

```
              3 STORE_GLOBAL             0 (x)

  4           6 LOAD_GLOBAL              1 (locals)
              9 CALL_FUNCTION            0
             12 PRINT_ITEM
             13 PRINT_NEWLINE
             14 LOAD_CONST               0 (None)
             17 RETURN_VALUE
```

set_local 函数中 "x = 'abc'" 的对应字节码指令为 "STORE_FAST, LOAD_FAST"，而 set_global 是 "STORE_GLOBAL, LOAD_GLOBAL"。很显然，如果不使用 global 关键字，"x = 'abc'" 将在当前函数作用域 locals 名字空间中创建一个新的名字对象关联。

你一定从多本书上看到了类似下面的错误警告。

```
In [32]: x = 1234

In [33]: def test():
   ....:     print x
   ....:     x = "abc"
   ....:

In [34]: test()
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)

/Users/test/python/<ipython-input-35-3069a7d116f7> in test()
      1 def test():
----> 2     print x
      3     x = "abc"
      4

UnboundLocalError: local variable 'x' referenced before assignment
```

但是，为什么会出错呢？不要总是背诵规则，我们应该看看 Python 编译器到底怎么理解这段代码的行为。

```
In [37]: dis.dis(test)
  2           0 LOAD_FAST                0 (x)
              3 PRINT_ITEM
              4 PRINT_NEWLINE

  3           5 LOAD_CONST               1 ('abc')
              8 STORE_FAST               0 (x)
             11 LOAD_CONST               0 (None)
             14 RETURN_VALUE
```

很显然，"0 LOAD_FAST 0 (X)" 访问的是 locals 名字空间，在这个空间里的确不存在一个叫 "x" 的名字，引发 "local variable 'x' referenced before assignment" 错误自然是合情合理的。

1.4.2 对性能的影响

名字空间的使用大大提高了 Python 的灵活性，我们可以围绕名字空间作出很多在静态语言看来不可思议的魔法。(在某些社区 "大牛" 眼里，这不是魔法，而是需要诅咒的异端，哈哈……)

每次访问对象，都需要从名字空间中查找，貌似会降低性能。就算 dict 在相关数据结构中已经是非常快的了，但总归比不上直接通过指针访问内存来得高效。

弱化指针，或者完全不使用指针，是现在高级语言共同遵守的一个 "约定"。一来，指针造成的问题非常多，常常导致进程 "崩溃"。其次，指针会对垃圾回收造成一定程度的干扰。Java 和 C# 这类语言使用 "引用" 来代替指针，是因为垃圾回收完成后，对内存的压缩整理会导致对象的内存地址发生变化，需要调整引用和指针的映射关系。Python 比 Java 和 C# 做得更彻底一些，直接用一个 dict 把对象和名字装在一起，虽然让我们有了更大的灵活性，但也进一步导致性能损失。

名字引用和属性引用会导致一些性能差异，看下面的例子:

```
In [38]: class A(object):
   ....:     def __init__(self, x):
   ....:         self.x = x
   ....:     def add(self, n):
   ....:         self.x += n
   ....:

In [39]: def test1():
   ....:     a = A(100)
   ....:     for i in xrange(100):
   ....:         a.add(i)
   ....:     print a.x
   ....:

In [40]: def test2():
   ....:     a = A(100)
   ....:     add = a.add
   ....:     for i in xrange(100):
   ....:         add(i)
   ....:     print a.x
   ....:

In [41]: test1()
5050

In [42]: test2()
5050
```

函数 test1 和 test2 的目的完全相同，区别在于 test1 每次循环都从对象 a 的名字空间查找 add，test2 则为其创建了一个 lcoals 名字。看看字节码的区别:

```
In [54]: dis.dis(test1)
  2           0 LOAD_GLOBAL              0 (A)
              3 LOAD_CONST               1 (100)
```

```
              6 CALL_FUNCTION            1
              9 STORE_FAST               0 (a)

  3          12 SETUP_LOOP              33 (to 48)
             15 LOAD_GLOBAL              1 (xrange)
             18 LOAD_CONST               1 (100)
             21 CALL_FUNCTION            1
             24 GET_ITER
        >>   25 FOR_ITER                19 (to 47)
             28 STORE_FAST               1 (i)

  4          31 LOAD_FAST                0 (a)
             34 LOAD_ATTR                2 (add)
             37 LOAD_FAST                1 (i)
             40 CALL_FUNCTION            1
             43 POP_TOP
             44 JUMP_ABSOLUTE           25
        >>   47 POP_BLOCK

  5     >>   48 LOAD_FAST                0 (a)
             51 LOAD_ATTR                3 (x)
             54 PRINT_ITEM
             55 PRINT_NEWLINE
             56 LOAD_CONST               0 (None)
             59 RETURN_VALUE

In [55]: dis.dis(test2)
  2           0 LOAD_GLOBAL              0 (A)
              3 LOAD_CONST               1 (100)
              6 CALL_FUNCTION            1
              9 STORE_FAST               0 (a)

  3          12 LOAD_FAST                0 (a)
             15 LOAD_ATTR                1 (add)
             18 STORE_FAST               1 (add)

  4          21 SETUP_LOOP              30 (to 54)
             24 LOAD_GLOBAL              2 (xrange)
             27 LOAD_CONST               1 (100)
             30 CALL_FUNCTION            1
             33 GET_ITER
        >>   34 FOR_ITER                16 (to 53)
             37 STORE_FAST               2 (i)

  5          40 LOAD_FAST                1 (add)
             43 LOAD_FAST                2 (i)
             46 CALL_FUNCTION            1
             49 POP_TOP
             50 JUMP_ABSOLUTE           34
        >>   53 POP_BLOCK

  6     >>   54 LOAD_FAST                0 (a)
             57 LOAD_ATTR                3 (x)
             60 PRINT_ITEM
```

```
             61 PRINT_NEWLINE
             62 LOAD_CONST               0 (None)
             65 RETURN_VALUE
```

test1 调用 add 方法由 4 条指令组成:

```
  4          31 LOAD_FAST                0 (a)
             34 LOAD_ATTR                2 (add)
             37 LOAD_FAST                1 (i)
             40 CALL_FUNCTION            1
```

test2 仅 3 条指令:

```
  5          40 LOAD_FAST                1 (add)
             43 LOAD_FAST                2 (i)
             46 CALL_FUNCTION            1
```

对于习惯 ASM 的程序员来说，显然 test2 的效率更高，何况还不知道 LOAD_ATTR 指令的性能如何 (有兴趣有时间的可以翻看以下 CPython 源码，我们前面流程分析时已经提到过字节码指令执行位置)。

用 %timeit 测试一下。记得先将两个函数的 print 语句注释掉，否则会被刷屏。

```
In [72]: timeit -n 100 -r 1 test1()
100 loops, best of 1: 79.8 us per loop

In [73]: timeit -n 100 -r 1 test2()
100 loops, best of 1: 69.8 us per loop

In [74]: timeit -n 1000 -r 1 test1()
1000 loops, best of 1: 69.2 us per loop

In [75]: timeit -n 1000 -r 1 test2()
1000 loops, best of 1: 55.2 us per loop

In [76]: timeit -n 10000 -r 1 test1()
10000 loops, best of 1: 55.2 us per loop

In [77]: timeit -n 10000 -r 1 test2()
10000 loops, best of 1: 48.3 us per loop
```

从测试结果看，的确有轻微的性能差异 (单位: us 微秒，1s = 1000000us)。多数时候可以忽略，但对于海量数据处理算法，累计下来就很可观了。

那么 "python -O" 优化会不会有所调整呢？我们把代码存到一个 test.py 中。

```
$ cat test.py
# -*- coding:utf-8 -*-

class A(object):
    def __init__(self, x):
        self.x = x
    def add(self, n):
```

```
        self.x += n

def test1():
    a = A(100)
    for i in xrange(100):
        a.add(i)

    return a.x

def test2():
    a = A(100)
    add = a.add
    for i in xrange(100):
        add(i)

    return a.x

if __name__ == "__main__":
    import dis

    dis.dis(test1)
    print "-" * 30
    dis.dis(test2)
```

输出:

```
$ python -O test.py

...
 13          31 LOAD_FAST                0 (a)
             34 LOAD_ATTR                2 (add)
             37 LOAD_FAST                1 (i)
             40 CALL_FUNCTION            1
...
------------------------------
...

 22          40 LOAD_FAST                1 (add)
             43 LOAD_FAST                2 (i)
             46 CALL_FUNCTION            1
...
```

很遗憾，优化前后的指令完全相同。

与之类似的还有 "import <module>" 和 "from <module> import <name>" 也会带来属性引用和名字引用的性能差异。

1.5 类型和对象

类型 (Type) 也是对象，一种特殊的，不被回收的对象。先有类型对象，然后才能由类型对象 "生育" 出大大小小的对象实例 (Instance)。

1.5.1 对象的内存布局

我们总是通过名字来访问目标对象，但名字仅仅是名字空间中的一个 key，一个 string。名字除了指路，并不能说明目标对象的任何信息，只有对象自身才知道。

> Names have no type, but objects do.

CPython 所有的对象都有一个标准的头信息。

Include/object.h

```
#define PyObject_HEAD                   \
    Py_ssize_t ob_refcnt;               \
    struct _typeobject *ob_type;

typedef struct _object {
    PyObject_HEAD
} PyObject;
```

头部这两个字段，分别代表 "引用计数" 和 "类型指针"。看看下面的演示:

```
In [1]: a = 0x1234

In [2]: b = a

In [3]: hex(id(a)), hex(id(b))
Out[3]: ('0x7fe66b0a3a80', '0x7fe66b0a3a80')

In [4]: hex(id(int))
Out[4]: '0x1024f9180'
```

注意不要使用 [-5, 257) 之间的小整数，它们会被缓存，不适合做这里的测试。在 OS X Lion 64 位系统中，Py_ssize_t 和 指针的长度都是 8 字节。

Include/intobject.h

```
typedef struct {
    PyObject_HEAD
    long ob_ival;
} PyIntObject;
```

用 gdb x/3xg 指令输出 3 组 8 字节数据。

```
$ gdb --pid=`pidof python`
(gdb) x/3xg 0x7fe66b0a3a80
0x7fe66b0a3a80:       0x0000000000000002    0x00000001024f9180
0x7fe66b0a3a90:       0x0000000000001234
```

我们会看到 PyIntObject 各字段数据分别是:

- ob_refcnt = 2
- ob_type = 0x00000001024f9180
- ob_ival = 0x0000000000001234

退出 gdb，我们在 IPython 中减少引用。再次查看内存时，你会发现引用计数的变化。

```
In [5]: del b
```

```
(gdb) x/3xg 0x7fe66b0a3a80
0x7fe66b0a3a80:       0x0000000000000001    0x00000001024f9180
0x7fe66b0a3a90:       0x0000000000001234
```

sys.getrefcount 函数可以获取对象引用计数。其返回的结果总是比你期望的要大 1 号，那是因为 Python 总是按引用传递，getrefcount 形参本身也会引用目标对象一次。

```
In [6]: import sys

In [7]: sys.getsizeof(a)
Out[7]: 24

In [8]: sys.getrefcount(a)
Out[8]: 2
```

并不是所有的引用都会改变引用计数，弱引用就不会，相关内容我们在后面的章节再做说明。

CPython 中还有另外一种变长对象，比如 string、list 等等。它们的头信息有所扩展。

Include/object.h

```
#define PyObject_VAR_HEAD               \
    PyObject_HEAD                       \
    Py_ssize_t ob_size; /* Number of items in variable part */

typedef struct {
    PyObject_VAR_HEAD
} PyVarObject;
```

比 PyObject 多了一个 ob_size 字段，它存储了数据项的数量。注意是 "Number of Items"，而非 "size"。比如存储 list 元素的数量，string 字符的数量等。

我们就以字符串为例，看看内存布局状况。

Include/stringobject.h

```
typedef struct {
    PyObject_VAR_HEAD
    long ob_shash;
    int ob_sstate;
    char ob_sval[1];

    /* Invariants:
     *     ob_sval contains space for 'ob_size+1' elements.
```

```
     *     ob_sval[ob_size] == 0.
     *     ob_shash is the hash of the string or -1 if not computed yet.
     *     ob_sstate != 0 iff the string object is in stringobject.c's
     *       'interned' dictionary; in this case the two references
     *       from 'interned' to this object are *not counted* in ob_refcnt.
     */
} PyStringObject;
```

"char ob_savl[1]" 是 C 语言创建可变长结构体的常用做法。

```
In [9]: s = "Hello, World!"

In [10]: hex(id(s))
Out[10]: '0x102f9a228'

In [11]: hex(id(str))
Out[11]: '0x102501f80'

In [12]: hex(hash(s))
Out[12]: '0x137f24410ef64a3c'

In [13]: len(s)
Out[13]: 13

In [14]: sys.getsizeof(s)
Out[14]: 50
```

先看看 PyObject_VAR_HEAD + ob_shash 的信息 (都是 8 字节)。

```
(gdb) x/4xg 0x102f9a228
0x102f9a228:   0x0000000000000001    0x0000000102501f80
0x102f9a238:   0x000000000000000d    0x137f24410ef64a3c
```

- ob_refcnt = 0x0000000000000001
- ob_type = 0x0000000102501f80
- ob_size = 0x000000000000000d = 13
- ob_shash = 0x137f24410ef64a3c

很好，和 IPython 里面的输出都对上号了，接着看后面的内容。

- PyObject_VAR_HEAD + ob_shash + ob_sstate = 36 (0x24) 字节；
- "ob_sval contains space for 'ob_size+1' elements." 表示其长度应该是 14 字节；
- "ob_sval[ob_size] == 0" 表示总是以 \0 结尾 。

```
(gdb) x/14xb 0x102f9a228 + 0x24
0x102f9a24c:   0x48    0x65    0x6c    0x6c    0x6f    0x2c    0x20    0x57
0x102f9a254:   0x6f    0x72    0x6c    0x64    0x21    0x00

(gdb) x/s 0x102f9a228 + 0x24
0x102f9a24c:    "Hello, World!"
```

```
In [15]: " ".join([hex(ord(c)) for c in s])
Out[15]: '0x48 0x65 0x6c 0x6c 0x6f 0x2c 0x20 0x57 0x6f 0x72 0x6c 0x64 0x21'
```

虽然，我们仅用 PyIntObject 和 PyStringObject 做了简单的说明，但已经初步了解一个 Python 对象在内存中的状态。用 gdb 抓内存不是必须的，但却比看 CPython 源码更更加印象深刻 (挖坟党的习惯行为)。

1.5.2 类型对象

在上一节，我们查看了对象实例 (Instance) 在内存中的布局，那么类型对象又包含了什么信息呢?

Include/object.h

```
typedef struct _typeobject {
    // 一样的标准头
    PyObject_VAR_HEAD

    // 类型名称
    const char *tp_name; /* For printing, in format "<module>.<name>" */

    // 内存分配参数
    Py_ssize_t tp_basicsize, tp_itemsize; /* For allocation */

    // 标准操作函数
    destructor tp_dealloc;
    printfunc tp_print;
    getattrfunc tp_getattr;
    setattrfunc tp_setattr;
    cmpfunc tp_compare;
    reprfunc tp_repr;

    /* More standard operations (here for binary compatibility) */
    hashfunc tp_hash;
    ternaryfunc tp_call;
    reprfunc tp_str;
    getattrofunc tp_getattro;
    setattrofunc tp_setattro;

    ... ...
    // __doc__
    const char *tp_doc; /* Documentation string */

    ... ...

    // 基类、子类、方法、静态成员、mro 等等。
    /* Attribute descriptor and subclassing stuff */
    struct PyMethodDef *tp_methods;
    struct PyMemberDef *tp_members;
    struct PyGetSetDef *tp_getset;
    struct _typeobject *tp_base;
    PyObject *tp_dict;
```

```
    descrgetfunc tp_descr_get;
    descrsetfunc tp_descr_set;
    initproc tp_init;
    ...
    PyObject *tp_bases;
    PyObject *tp_mro; /* method resolution order */
    PyObject *tp_subclasses;
    PyObject *tp_weaklist;
    destructor tp_del;

    ... ...
} PyTypeObject;
```

Instance 存储了实例数据，再通过类型指针找到这里，操作就齐活了。两者配合起来，才有了我们所谓的面向对象开发。

在 Include/Object.h 中有这样一段注释文字。

```
An object has a 'type' that determines what it represents and what kind of data it contains.  An
object's type is fixed when it is created. Types themselves are represented as objects; an object
contains a pointer to the corresponding type object.  The type itself has a type pointer pointing to
the object representing the type 'type', which contains a pointer to itself!).
```

int、list 等这些都是 types 中某个具体类型的短名。类型本身也是一个对象，它自然也有类型。比如 IntType 的类型就是 types.TypeType (短名 type)。types.TypeType 也有类型，不过这回是丫自己。

```
In [16]: import types

In [17]: int is types.IntType
Out[17]: True

In [18]: list is types.ListType
Out[18]: True

In [19]: type(int)
Out[19]: type

In [20]: type(type)
Out[20]: type

In [21]: type is types.TypeType
Out[21]: True
```

试着动态创建一个类型看看。

```
In [22]: ClassA = type("ClassA", (object,), {})

In [23]: a = ClassA()
```

```
In [24]: type(a)
Out[24]: __main__.ClassA

In [25]: type(ClassA)
Out[25]: type
```

"class A(object): ..." 和 "type('A', (object,), {...})" 的作用完全相同，区别仅在于后者是运行期动态创建而已。

类型信息保存在 __class__ 属性里 (还记得对象头那个 *ob_type 指针吗)。

```
In [26]: a = 1234

In [27]: type(a)
Out[27]: int

In [28]: a.__class__
Out[28]: int

In [29]: type(a) is a.__class__
Out[29]: True

In [30]: type(int) is int.__class__
Out[30]: True
```

一个对象的类型无法改变吗？当然是否定的，否则怎么叫魔法呢。比如下面例子中，我们把 "儿子" 变 "孙子"，横插一杠子的家伙偷偷改变了某些继承行为。

```
In [31]: class A(object):
   ...:     def test(self):
   ...:         print "A test"

In [32]: class B(A):
   ...:     pass

In [33]: b = B()

In [34]: b.test()
A test

In [35]: type(b)
Out[35]: __main__.B
```

事情到此还很正常，接下来，暗黑巫师 X 来了。通过修改 <Instance>.__class__ 属性来改变对象实例的类型，从而达到动态注入的目的。你想到了什么？AOP？知道为什么会被一些 "老夫子" 当作异端了吧？当然，我们不能修改一个类型对象的类型。

```
In [36]: class X(A):
   ...:     def test(self):
   ...:         print "X test"
```

```
In [37]: b.__class__ = X

In [38]: b.test()
X test

In [39]: type(b)
Out[39]: __main__.X

In [40]: isinstance(b, A)
Out[40]: True
```

我们还可以用 __metaclass__ 自定义类型的类型，以便控制自定义类型对象本身的细节。写个阻止继承的密封类玩玩。

```
In [41]: class SealedClass(type):
   ....:     _types = set()
   ....:     def __init__(cls, name, bases, attrs):
   ....:         cls._types.add(cls)
   ....:         for b in bases:
   ....:             if b in cls._types: raise SyntaxError("太过分了，大内总管的财产你也惦记着。")
   ....:

In [42]: class Caohuachun(object):
   ....:     __metaclass__ = SealedClass
   ....:

In [43]: type(Caohuachun)
Out[43]: __main__.SealedClass

In [44]: class Xiaolizi(Caohuachun): pass
SyntaxError: 太过分了，大内总管的财产你也惦记着。
```

好吧，我知道有点绕头，总结一下:

- 对象实例 (Instance) 是由其对应的类型对象 (比如 IntType 等) 生成的。
- 类型对象 (IntType 等) 本身又是由 ctypes.TypeType (短名 type) 生成的。
- 可以通过自定义 __metaclass__ 来干预类型对象的创建。

__metaclass__ 的用途很多，也算 Python 里面一个比较深的 “坑”。本书有专门的章节详细探讨它的使用方法。

类型相关的东西还有很多，让我们一路走下去，期望能识得 "庐山真面目"。

1.6 内存管理

Python 为了提高性能，在内存管理上倾尽心血。诸多复杂的技术和理念，默默支持着 Python 的简单和优雅。

你是否知道内存池的存在?
你是否知道 Python 有两套内存回收机制?
你是否知道拥有 GC 的 Python 也会造成内存泄漏?

1.6.1 内存分配

暂时抛开种种问题，先看看 Python 如何创建对象?

Objects/object.c

```
PyObject * _PyObject_New(PyTypeObject *tp)
{
    PyObject *op;

    // 很显然，这个 PyObject_MALLOC 就是下一个追捕的目标。
    op = (PyObject *) PyObject_MALLOC(_PyObject_SIZE(tp));
    ...
    // 初始化对象
    return PyObject_INIT(op, tp);
}

#define Py_TYPE(ob)  (((PyObject*)(ob))->ob_type)  // 摘自 object.h

PyObject * PyObject_Init(PyObject *op, PyTypeObject *tp)
{
    // 设置对象头部的类型指针，还记得 PyObject_Head 结构吧。
    Py_TYPE(op) = tp;
    ...
    return op;
}

void _PyObject_Del(PyObject *op)
{
    // 又一个追捕的对象出现了。
    PyObject_FREE(op);
}
```

和我们设想的差不多，按照对象大小为其分配内存空间，然后初始化头部 PyObject_HEAD 的相关参数。不过在分析具体的内存分配函数前，我们先看看 Objects/obmalloc.c 里面的一段 Python 内存架构的说明文字。

```
/* An object allocator for Python.

   Here is an introduction to the layers of the Python memory architecture,
   showing where the object allocator is actually used (layer +2), It is
   called for every object allocation and deallocation (PyObject_New/Del),
   unless the object-specific allocators implement a proprietary allocation
   scheme (ex.: ints use a simple free list). This is also the place where
   the cyclic garbage collector operates selectively on container objects.
```

```
    Object-specific allocators
    _____   ______   ______       ________
   [ int ] [ dict ] [ list ] ... [ string ]       Python core         |
+3 | <----- Object-specific memory -----> | <-- Non-object memory --> |
    ________________________________       |                           |
   [   Python's object allocator   ]      |                           |
+2 | ####### Object memory ####### | <------ Internal buffers ------> |
    ______________________________________________________________    |
   [          Python's raw memory allocator (PyMem_ API)          ]   |
+1 | <----- Python memory (under PyMem manager's control) ------> |   |
    __________________________________________________________________
   [    Underlying general-purpose allocator (ex: C library malloc)   ]
 0 | <------ Virtual memory allocated for the python process -------> |

   ========================================================================
    ______________________________________________________________________
   [                OS-specific Virtual Memory Manager (VMM)               ]
-1 | <--- Kernel dynamic storage allocation & management (page-based) ---> |
    __________________________________   __________________________________
   [                                  ] [                                  ]
-2 | <-- Physical memory: ROM/RAM --> | | <-- Secondary storage (swap) --> |

*/
```

在 "+1"、"+2" 两个层面，Python 分别包装了两套内存管理 API：

raw memory allocator: Include/pymem.h

```
#define PyMem_MALLOC(n)      ((size_t)(n) > (size_t)PY_SSIZE_T_MAX ? NULL \
                             : malloc((n) ? (n) : 1))
#define PyMem_REALLOC(p, n)  ((size_t)(n) > (size_t)PY_SSIZE_T_MAX  ? NULL \
                             : realloc((p), (n) ? (n) : 1))
#define PyMem_FREE           free
```

object allocator: Include/objimp.h

```
#define PyObject_MALLOC         PyObject_Malloc
#define PyObject_REALLOC        PyObject_Realloc
#define PyObject_FREE           PyObject_Free
```

PyMem_ 只是简单的 C 函数包装，我们真正要关心的是 PyObject_。PyObject_ 除了为对象分配存储内存，还要负责初始化对象的类型指针等信息。而最最重要的是，Python 内存管理中最核心的小对象内存池和垃圾回收都与之有关。至于 PyMem_ 作为更底层的 API，也有发挥空间，比如在创建整数内存池时就会用到。

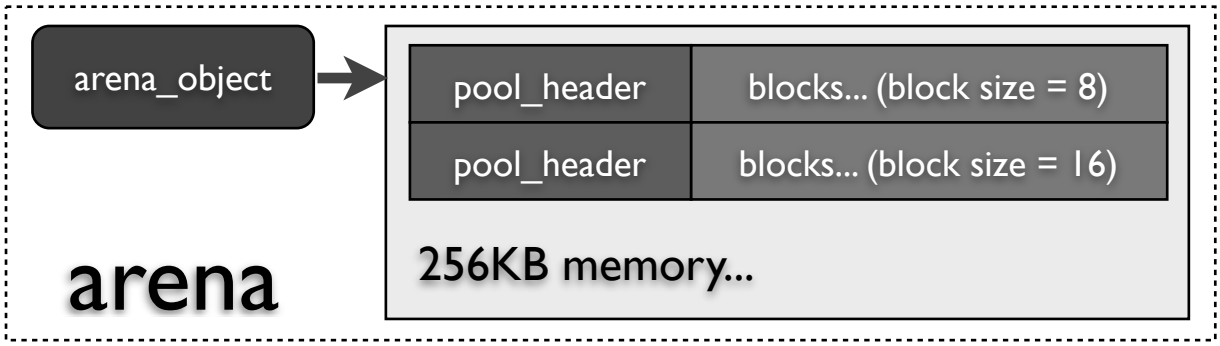
至于 "+3" 所涉及的 "Object-specific memory"，会在本书第二章做全面深入的分析。现在，我们还是要回到本节的主题，继续沿着 PyObject_Malloc 这条线索追踪下去。

obmalloc.c 中还有一段内存分配策略的说明，其中提到大于 256 字节的对象不会使用内存池，而是直接调用系统内存分配。这点很好理解，程序执行过程中，以几十字节的小对象居多，频繁调用 malloc/free 进入内核态申请和释放内存，必然导致性能降低和出现大量内存碎片。

```
/*
 * Allocation strategy abstract:
 *
 * For small requests, the allocator sub-allocates <Big> blocks of memory.
 * Requests greater than 256 bytes are routed to the system's allocator.
 *
 * Small requests are grouped in size classes spaced 8 bytes apart, due
 * to the required valid alignment of the returned address. Requests of
 * a particular size are serviced from memory pools of 4K (one VMM page).
 * Pools are fragmented on demand and contain free lists of blocks of one
 * particular size class. In other words, there is a fixed-size allocator
 * for each size class. Free pools are shared by the different allocators
 * thus minimizing the space reserved for a particular size class.
 *
 * This allocation strategy is a variant of what is known as "simple
 * segregated storage based on array of free lists". The main drawback of
 * simple segregated storage is that we might end up with lot of reserved
 * memory for the different free lists, which degenerate in time. To avoid
 * this, we partition each free list in pools and we share dynamically the
 * reserved space between all free lists. This technique is quite efficient
 * for memory intensive programs which allocate mainly small-sized blocks.
 *
 * For small requests we have the following table:
 *
 * Request in bytes     Size of allocated block      Size class idx
 * ----------------------------------------------------------------
 *        1-8                     8                       0
 *        9-16                   16                       1
 *       17-24                   24                       2
 *       25-32                   32                       3
 *       33-40                   40                       4
 *       41-48                   48                       5
 *       49-56                   56                       6
 *       57-64                   64                       7
 *       65-72                   72                       8
 *        ...                   ...                     ...
 *      241-248                 248                      30
 *      249-256                 256                      31
 *
 *      0, 257 and up: routed to the underlying allocator.
 */
```

上面这段注释已经很好的阐述了 Python 内存池的基本信息，其中提到了 block、pool 等概念。为了让大家更好地理解后续代码的相关内容，我们需要预先了解一些背景知识。

Python Object 内存池架构说明：



arena:

arena 是内存申请单位，每个 arena 包含一个存储状态的 arena_object 对象，还有一块延迟分配的 256KB 大小内存。当 arena 的内存不再使用时，会被释放，交还给操作系统。并下次需要的时候再次分配。

多个 arena 通过 arena_object 内的链表连接起来，共同构成 Python 内存池。

pool:

内存管理机制会将 arena 的 256KB 内存块分割成多个片段来使用。这些片段长度为一个系统内存页大小 (通常是 4KB)，被称为 pool。也就是说每个 arena 最多可以有 0 ~ 64 个 pool。

pool 在其起始位置存储了 pool_header 状态信息。当需要存储小对象时，就是通过检索该区来查看是否符合存储要求。

剩余的内存会按照需要，再次分割成多个更小的区域 block。每个 pool 只包含一种大小规格的 block。基于性能原因，block 大小总是按 8 字节对齐，也就是说总是 8 的倍数。

block 长度在 CPython 源码中称为 "Size of allocated block"，并依照这个长度，将 pool 标上 "Size class idx"。block 大小为 8 字节的 pool idx 等于 0，block 大小等于 16 字节的 pool idx 等于 1..... 如此只要通过这个 idx 就能确定 pool block 的大小，Python 使用该字段配合查找合适的存储位置。

pool 有三种状态：empty 尚未使用；used 起码有一个 block 被使用；full 全部的 block 都被使用。

empty 状态的 pool 并没有确定其 block 大小，只在需要的时候再做决定。而当 pool 从 used 状态变成 empty 时，它会被放入名为 freepools 的链表中，在下次需要时再重新作出合理的分配。Python 内部通过链表来管理这三种不同状态的 pool，以便快速查找合适的存储位置。

block:

block 是 Python 内存池最小的分配单位，每个 block 存储一个合适大小的对象。block 大小总是 8 字节的倍数。比如 8 字节的 block 可存储大小为 1 ~ 8 字节的对象。

我们先看看 Python 是如何创建 arena，并为其分配内存的。

Objects/obmalloc.c

```
struct pool_header {
    union { block *_padding;
            uint count; } ref;          // 已分配 block 数量
    block *freeblock;                   // 下一个可用 block
    struct pool_header *nextpool;       // 相同 size class 构成链表
    struct pool_header *prevpool;
    uint arenaindex;                    // pool 所在 arena 的序号
    uint szidx;                         // size class index
    uint nextoffset;
    uint maxnextoffset;
};

struct arena_object {
    uptr address;                    // 用来存储 pool 的那段内存地址
    block* pool_address;             // 下一个 pool 的切分地址
    uint nfreepools;                 // 可用的 pool 数量
    uint ntotalpools;                // pool 总数
    struct pool_header* freepools;   // 可用 pool 链表
    struct arena_object* nextarena;  // 用双向链表将 arena 串起来。
    struct arena_object* prevarena;
};

#define ARENA_SIZE              (256 << 10)     /* 256KB */

static struct arena_object* new_arena(void)
{
    ... ...

    // 没有可用的 arena 了。
    if (unused_arena_objects == NULL) {

        // 本次要申请的 arena 数量。
        nbytes = numarenas * sizeof(*arenas);

        // 为多个存储状态的 arena_object 申请内存。
        arenaobj = (struct arena_object *)realloc(arenas, nbytes);
        arenas = arenaobj;

        // 将这些 arena 加入 unused_arena_objects 列表中。
        for (i = maxarenas; i < numarenas; ++i) {
            arenas[i].address = 0;                          // 仅有 arena_object，并没分配存储内存。
            arenas[i].nextarena = i < numarenas - 1 ?
                                  &arenas[i+1] : NULL;
        }

        /* Update globals. */
        unused_arena_objects = &arenas[maxarenas];
        maxarenas = numarenas;
    }
```

```
    // 从列表中获取一个未使用的 arena
    arenaobj = unused_arena_objects;

    // 将 arenaobj 从 unused_arena_objects 中剔除
    unused_arena_objects = arenaobj->nextarena;

    // 别忘了为其分配 256KB 的存储内存。
    arenaobj->address = (uptr)malloc(ARENA_SIZE);

    /* 这里有一大堆有关 pool 的设置 ... */

    return arenaobj;
}
```

回到本节开头的问题，我们需要关心一下如何为对象分配存储内存。

Objects/obmalloc.c

```
#define SMALL_REQUEST_THRESHOLD 256
#define SMALL_MEMORY_LIMIT      (64 * 1024 * 1024)         /* 64 MB -- more? */
#define ARENA_SIZE              (256 << 10)                /* 256KB */

#ifdef WITH_MEMORY_LIMITS
#define MAX_ARENAS              (SMALL_MEMORY_LIMIT / ARENA_SIZE)
#endif

void * PyObject_Malloc(size_t nbytes)
{
    ... ...

    // 判断要分配的内存大小是否小于或等于 256 字节
    if ((nbytes - 1) < SMALL_REQUEST_THRESHOLD) {

        // 查看是否有适合的 pool 用来存储小对象
        size = (uint)(nbytes - 1) >> ALIGNMENT_SHIFT;
        pool = usedpools[size + size];

        if (pool != pool->nextpool) {

            // 返回适合存储的 block 指针。
            ...
            ++pool->ref.count;
            bp = pool->freeblock;
            ...
            return (void *)bp;
        }

        // 没有可用 pool，只好分配新的内存。
        if (usable_arenas == NULL) {

#ifdef WITH_MEMORY_LIMITS

            // 判断是否超出了 arenas 数量限制。这得看编译参数设置了，貌似不用管。
```

```
            if (narenas_currently_allocated >= MAX_ARENAS) {
                ...
                // 超出限制，直接跳转到函数尾部，用 malloc 分配。
                goto redirect;
            }
#endif

            // 新分配一个 arena。
            usable_arenas = new_arena();
            ...

            // 将新分配的 arena 加入链表，以便管理。
            usable_arenas->nextarena =
                usable_arenas->prevarena = NULL;
        }

        ...

        // 从 arenas 中获取可用的 pool
        pool = usable_arenas->freepools;
        if (pool != NULL) {

            // 将取出的 pool 从 freepools 链表中剔除。
            usable_arenas->freepools = pool->nextpool;

            // 减少 nfreepools 数量。
            --usable_arenas->nfreepools;
            if (usable_arenas->nfreepools == 0) {

                // 如果没有空闲的 pool 了，就将该 arena 从 usable_arenas 链表中剔除。
                usable_arenas = usable_arenas->nextarena;
                ...
            }
            else {
                ... ...
            }
        init_pool:
            ... ...

            // 初始化 pool
            pool->szidx = size;
            size = INDEX2SIZE(size);
            bp = (block *)pool + POOL_OVERHEAD;
            pool->nextoffset = POOL_OVERHEAD + (size << 1);
            pool->maxnextoffset = POOL_SIZE - size;
            pool->freeblock = bp + size;
            *(block **)(pool->freeblock) = NULL;

            // 返回 block 指针。
            return (void *)bp;
        }
        ... ...
    }
```

```
    /* The small block allocator ends here. */

redirect:
    // 这里直接用 malloc 为超过 256 字节的对象分配内存。
    return (void *)malloc(nbytes);
}
```

有一个需要关心的问题，arena 的内存在不使用时真的会被释放吗？这个问题在诸多社区里被问及，毕竟谁也不想刚买的 4GB 内存条很快就不够用了。

Objects/obmalloc.c

```
void PyObject_Free(void *p)
{
    ... ...

    // 找到对应的 pool
    pool = POOL_ADDR(p);
    if (Py_ADDRESS_IN_RANGE(p, pool)) {
            ... ...

            // 这段注释好。也就是说，如果 arena 内的 pool 都不再使用，就 free 掉，把存储内存还给操作系统。

            /* All the rest is arena management.  We just freed
             * a pool, and there are 4 cases for arena mgmt:
             * 1. If all the pools are free, return the arena to
             *    the system free().
             * 2. If this is the only free pool in the arena,
             *    add the arena back to the `usable_arenas` list.
             * 3. If the "next" arena has a smaller count of free
             *    pools, we have to "slide this arena right" to
             *    restore that usable_arenas is sorted in order of
             *    nfreepools.
             * 4. Else there's nothing more to do.
             */
            if (nf == ao->ntotalpools) {
                ...

                // 注意，arena 的状态对象 arena_object 不会被释放，重新加入 unused_arena_objects 链表。
                ao->nextarena = unused_arena_objects;
                unused_arena_objects = ao;

                // 至于存储 pool、block 的内存是要释放掉滴。
                free((void *)ao->address);
                ao->address = 0;                        // 原来的内存地址已经失效
                --narenas_currently_allocated;

                ... ...
                return;
            }

            /* 这里省略掉很多，基本不用关心。 */
    }
}
```

有关 Python 内存管理更多的细节，可阅读《Python 源码剖析》。对大多数人而言，了解本章所说的内存池，大小对象分配策略，以及 arena 会被释放就够了。

1.6.2 弱引用

通过传递弱引用对象，可以在不改变目标对象引用计数，不影响内存回收的前提下完成操作。可用来打破循环引用，或者设计对象池等等。但不是所有的对象都能使用弱引用，一些内置类型，如 int、tuple 等就在禁止之列。

```
In [1]: import sys, weakref

In [2]: class A(object):
   ....:     def test(self):
   ....:         print "test..."
   ....:

In [3]: a = A()

In [4]: sys.getrefcount(a)
Out[4]: 2

In [5]: r = weakref.ref(a)

In [6]: r
Out[6]: <weakref at 0x102dd9b50; to 'A' at 0x102da9c90>

In [7]: r().test()
test...

In [8]: sys.getrefcount(a)
Out[8]: 2
```

我们可以看到弱引用正常调用目标对象成员，但却没有改变其引用计数值。

Include/weakrefobject.h

```
typedef struct _PyWeakReference PyWeakReference;

struct _PyWeakReference {
    PyObject_HEAD

    // 引用的目标对象，不会改变其引用计数。
    PyObject *wr_object;

    // 目标对象死亡时的回调函数。
    PyObject *wr_callback;

    // 通过双向链表来维护指向同一个目标对象的多个弱引用对象。
    PyWeakReference *wr_prev;
    PyWeakReference *wr_next;
```

```
    ... ...
};
```

_PyWeakReference.wr_object 指针指向目标对象。弱引用对象重载了 __call__ 操作符用于返回目标对象。

```
In [9]: hex(id(a))
Out[9]: '0x102da9c90'

In [10]: r
Out[10]: <weakref at 0x102dd9b50; to 'A' at 0x102da9c90>

In [11]: r.__call__()
Out[11]: <__main__.A at 0x102da9c90>
```

既然弱引用不能改变目标对象的引用计数，也就意味着无法阻止目标对象被回收。那么如何知道目标对象 "死亡" 呢？简单的方法是判断 __call__ 返回值是否为 None，还可使用回调函数让虚拟机在回收目标对象时主动发出通知。

```
In [12]: class A(object):
   ....:     def __del__(self):
   ....:         print hex(id(self)), "destory"
   ....:

In [13]: a = A()

In [14]: def callback(ref):
   ....:     print hex(id(ref)), "dead callback"
   ....:

In [15]: r = weakref.ref(a, callback)

In [16]: r() is a
Out[16]: True

In [17]: del a
0x102bf46d8 dead callback
0x102da9990 destory

In [18]: r
Out[18]: <weakref at 0x102bf46d8; dead>

In [19]: r() is None
Out[19]: True
```

注意回调函数的参数是弱引用对象，而非目标对象。那么虚拟机如何知道对象有哪些弱引用，并及时发出死亡通知呢？

在目标对象的 __weakref__ 字段中存储了最后创建的弱引用对象，通过其内部链表就可以获知所有指向自己的弱引用对象，如此就可以完成所有的回调函数调用。

Include/classobject.h

```
typedef struct {
    PyObject_HEAD
    PyClassObject *in_class; /* The class object */
    PyObject     *in_dict;    /* A dictionary */
    PyObject     *in_weakreflist; /* List of weak references */
} PyInstanceObject;
```

```
In [20]: def cb1(r): print r, "dead callback1"

In [21]: def cb2(r): print r, "dead callback2"

In [22]: a = A()

In [23]: r1 = weakref.ref(a, cb1)

In [24]: r2 = weakref.ref(a, cb2)

In [25]: r1, r2
Out[25]:
(<weakref at 0x1066c5838; to 'A' at 0x1068c4d10>,
 <weakref at 0x1066c5940; to 'A' at 0x1068c4d10>)

In [26]: a.__weakref__
Out[26]: <weakref at 0x1066c5940; to 'A' at 0x1068c4d10>

In [27]: weakref.getweakrefs(a)
Out[27]:
[<weakref at 0x1066c5940; to 'A' at 0x1068c4d10>,
 <weakref at 0x1066c5838; to 'A' at 0x1068c4d10>]

In [28]: del a
<weakref at 0x1066c5940; dead> dead callback2
<weakref at 0x1066c5838; dead> dead callback1
```

如不使用回调函数，那么共用同一个弱引用对象就足够了，weakref.ref 将直接从 __weakref__ 返回这个对象。

```
In [29]: a = A()

In [30]: r1 = weakref.ref(a)

In [31]: a.__weakref__, r1
Out[31]:
(<weakref at 0x1066cb260; to 'A' at 0x1068c4e10>,
 <weakref at 0x1066cb260; to 'A' at 0x1068c4e10>)

In [32]: r2 = weakref.ref(a)

In [33]: r2
Out[33]: <weakref at 0x1066cb260; to 'A' at 0x1068c4e10>
```

```
In [34]: a.__weakref__ is r1 is r2
Out[34]: True
```

weakref 模块还提供了一种和弱引用非常类似的 ProxyType。

```
In [35]: import sys, weakref

In [36]: class A(object):
   ....:     def test(self):
   ....:         print "test"
   ....:     def __del__(self):
   ....:         print "destory"
   ....:

In [37]: a = A()

In [38]: sys.getrefcount(a)
Out[38]: 2

In [39]: def callback(p):
   .....:     print "dead callback"
   .....:

In [40]: p = weakref.proxy(a, callback)

In [41]: sys.getrefcount(a)  # 引用计数一样没有改变。
Out[41]: 2

In [42]: p.test()            # 果然是 proxy，比 weakref 更贴近原对象。
test

In [43]: p is a              # proxy 不是原对象。
Out[43]: False

In [44]: del a               # 原对象死亡，proxy 接获通知。
dead callback
destory
```

1.6.3 垃圾回收

Python 提供了 "引用计数" 和 "垃圾回收" 两套内存回收机制。垃圾回收 "garbage collector"，通常缩写为 GC。

引用计数技术实现简单，通过 Py_INCREF 和 Py_DECREF 两个宏修改对象头 PyObject_HEAD 的 ob_refcnt 字段。一但引用计数变为 0，立即清除对象，把内存返还给内存池或操作系统。某些特殊对象，比如小整数缓存对象、类型对象，不受该机制影响。

Include/Object.h

```
#define PyObject_HEAD                   \
    Py_ssize_t ob_refcnt;               \
```

```
    struct _typeobject *ob_type;

#define Py_INCREF(op) (                         \
    ((PyObject*)(op))->ob_refcnt++)

#define Py_DECREF(op)                                   \
    do {                                                \
        if (--((PyObject*)(op))->ob_refcnt != 0)        \
            _Py_CHECK_REFCNT(op)                        \
        else                                            \
            _Py_Dealloc((PyObject *)(op));              \
    } while (0)
```

基于引用计数机制，对象内存会被立即回收，相比 "标记-清除"、"停止-拷贝" 等垃圾回收算法，具有更高的效率。不过缺点也很明显，就是对付不了循环引用。

先停掉 GC，看看正常的引用计数回收演示。

```
In [1]: import gc

In [2]: gc.disable()

In [3]: class A(object):
   ...:     def __del__(self):
   ...:         print self, "dead"
   ...:

In [4]: a = A()

In [5]: b = a

In [6]: del a

In [7]: del b
<__main__.A object at 0x10f583990> dead
```

再看看由循环引用导致的问题。

```
In [8]: a, b = A(), A()

In [9]: a.b = b

In [10]: b.a = a

In [11]: del a

In [12]: del b
```

"杯具" 了，这就是传说中的内存泄漏 (memory-leak)。怎么解决这个问题呢？还记得弱引用吗？

```
In [13]: a, b = A(), A()
```

```
In [14]: a.b = weakref.ref(b)

In [15]: b.a = weakref.ref(a)

In [16]: a.b() is b, b.a() is a
Out[16]: (True, True)

In [17]: del a
<__main__.A object at 0x10f5b4b10> dead

In [18]: del b
<__main__.A object at 0x10f5b4b50> dead
```

多数时候，引用计数都能很好且高效地工作。尤其是一些简单的小程序，比如海量数据导入之类的，完全可以把 GC 关掉来提高性能。还可以在程序某个位置暂时关掉 GC，等 "性能瓶颈问题" 过后再打开。

很多刚入门的 Pythoner 会被论坛上的某些人警告说: "不要用 __del__，会导致内存泄漏 ....."。说法虽然有些夸张，但的确存在这个风险。我们会在后面分析垃圾回收过程的代码里看到由 __del__ 和循环引用共同导致的 "钉子户" 事件。风险并不是我们抛弃 __del__ 的终究理由，在上面例子中，我们虽然可以用弱引用回调函数来获知目标对象死亡，但这无法取代 __del__ 作为 "析构函数" 的位置，因为回调函数无法获知对象内部状态，也无法了解清理逻辑。

有些时候，循环引用不会像上面例子那么一目了然，可能会通过多个对象间接形成复杂的循环引用。比如 "a 引用 b，b 引用 c，c 又引用 a"，这时候想要找到问题就不那么容易了。也正因为如此，Python 提供了基于 "标记-清除" 算法的垃圾回收机制，用以打破循环引用，让对象能被正常回收。

Modules/gcmodule.c

```
/*
  Reference Cycle Garbage Collection
  ==================================

  Neil Schemenauer <nas@arctrix.com>

  Based on a post on the python-dev list.  Ideas from Guido van Rossum,
  Eric Tiedemann, and various others.

  http://www.arctrix.com/nas/python/gc/
  http://www.python.org/pipermail/python-dev/2000-March/003869.html
  http://www.python.org/pipermail/python-dev/2000-March/004010.html
  http://www.python.org/pipermail/python-dev/2000-March/004022.html

  For a highlevel view of the collection process, read the collect
  function.
*/
```

GC 只关注那些可能会导致循环引用的 "危险对象"，比如 list、dict 等容器，还有就是持有名字空间的 instance、class。至于 int、str 这些不可变对象，自然属于 "安全人群"。GC 会跟踪所有新创建的 "危险对象"，并将它们记录在一个链表中。

在被跟踪对象头部 PyObject_HEAD 之前会被额外添加一个 PyGC_Head，用来管理 GC 所需的信息，其中就有那个用于跟踪的链表。

Include/objimpl.h

```
typedef union _gc_head {
    struct {
        union _gc_head *gc_next;
        union _gc_head *gc_prev;
        Py_ssize_t gc_refs;
    } gc;
    long double dummy;  /* force worst-case alignment */
} PyGC_Head;
```

为此，Python 专门提供了 3 个 _PyObject_GC 函数用来创建对象和分配内存。

Modules/gcmodule.c

```
PyObject * _PyObject_GC_New(PyTypeObject *tp)
{
    PyObject *op = _PyObject_GC_Malloc(_PyObject_SIZE(tp));
    if (op != NULL)
        op = PyObject_INIT(op, tp);
    return op;
}

PyVarObject * _PyObject_GC_NewVar(PyTypeObject *tp, Py_ssize_t nitems)
{
    const size_t size = _PyObject_VAR_SIZE(tp, nitems);
    PyVarObject *op = (PyVarObject *) _PyObject_GC_Malloc(size);
    if (op != NULL)
        op = PyObject_INIT_VAR(op, tp, nitems);
    return op;
}

#define FROM_GC(g) ((PyObject *)(((PyGC_Head *)g)+1))

PyObject * _PyObject_GC_Malloc(size_t basicsize)
{
    ... ...

    // 申请的内存包含了 PyGC_HEAD。
    g = (PyGC_Head *)PyObject_MALLOC(
        sizeof(PyGC_Head) + basicsize);

    ...

    // 设置 track 标记
    g->gc.gc_refs = GC_UNTRACKED;
```

```
    // 虾米东东？代龄啊。我又想起在 .NET CLR 里面用 SOS.dll 挖坟的日子了。
    //   新对象将被加入 gen_0。
    generations[0].count++; /* number of allocated GC objects */

    // 如果 gen_0 超出 threshold 阀值，且 GC 可用，那么就开始执行垃圾回收。
    if (generations[0].count > generations[0].threshold &&
        enabled &&
        generations[0].threshold &&
        !collecting &&
        !PyErr_Occurred()) {
        collecting = 1;
        collect_generations();
        collecting = 0;
    }

    // 修正指针，返回从 PyObject_HEAD 开始的指针。
    op = FROM_GC(g);
    return op;
}
```

虽然我们在 _PyObject_GC_Malloc 里意外看到了触发垃圾回收的原因，但却没有找到任何将其加入跟踪链表的代码，那么 GC 是什么时候把新目标加入黑名单的呢？

我们打开两个需要被跟踪的危险类型对象创建函数。

Objects/listobject.c

```
PyObject * PyList_New(Py_ssize_t size)
{
    ... ...
    _PyObject_GC_TRACK(op);
    return (PyObject *) op;
}
```

Objects/classobject.c

```
PyObject * PyClass_New(PyObject *bases, PyObject *dict, PyObject *name)
     /* bases is NULL or tuple of classobjects! */
{
    ... ...
    op = PyObject_GC_New(PyClassObject, &PyClass_Type);
    ... ...
    _PyObject_GC_TRACK(op);
    return (PyObject *) op;
}
```

很显然，_PyObject_GC_TRACK 具有重大嫌疑。

Modules/gcmodule.c

```
void _PyObject_GC_Track(PyObject *op)
{
    PyObject_GC_Track(op);
```

```
}

void PyObject_GC_Track(void *op)
{
    _PyObject_GC_TRACK(op);
}
```

Include/objimpl.h

```
/* Tell the GC to track this object. */
#define _PyObject_GC_TRACK(o) do { \
    PyGC_Head *g = _Py_AS_GC(o); \
    ...
    g->gc.gc_refs = _PyGC_REFS_REACHABLE; \
    g->gc.gc_next = _PyGC_generation0; \
    g->gc.gc_prev = _PyGC_generation0->gc.gc_prev; \
    g->gc.gc_prev->gc.gc_next = g; \
    _PyGC_generation0->gc.gc_prev = g; \
} while (0);
```

看来想做一个 "恐怖分子" 也很不容易，费尽千辛万苦，终于在 _PyObject_GC_TRACK 宏里挤上 gen_0 的 "黑名单"，PyGC_HEAD 里面的链表起作用了。不过就算上了黑名单，如果不存在循环引用，引用计数就能搞定相关事宜。此时 Python 是不会麻烦 GC，引用计数机制会将目标从黑名单里删除，然后立即回收内存。

Python 和 .NET 一样，共划分出 3 级代龄。从下面的代码里，我们还可以看到默认的代龄阀值。全局变量 generations 数组管理着不同代龄对象的链表。

Modules/gcmodule.c

```
struct gc_generation {
    PyGC_Head head;
    int threshold;    /* collection threshold */
    int count;        /* count of allocations or collections of younger generations */
};

#define NUM_GENERATIONS 3
#define GEN_HEAD(n) (&generations[n].head)

/* linked lists of container objects */
static struct gc_generation generations[NUM_GENERATIONS] = {
    /* PyGC_Head,                                threshold,      count */
    {{{GEN_HEAD(0), GEN_HEAD(0), 0}},           700,            0},
    {{{GEN_HEAD(1), GEN_HEAD(1), 0}},           10,             0},
    {{{GEN_HEAD(2), GEN_HEAD(2), 0}},           10,             0},
};

PyGC_Head *_PyGC_generation0 = GEN_HEAD(0);
```

当垃圾回收被触发时，GC 会从全局引用等根对象 (root object) 开始遍历这些 "黑名单" 上的目标对象，并将它们标记为 “可达 (reachable)" 和 "不可达 (unreachable)" 两种状态。所有不可达的对象都将被尝试回收。

接下来，让我们看看 GC 的完整回收过程。

Modules/gcmodule.c

```
static Py_ssize_t collect_generations(void)
{
    ... ...

    // 回收工作，先从 gen_2 开始。不过回收却是往 gen_0 添加对象时引发的。
    for (i = NUM_GENERATIONS-1; i >= 0; i--) {
        // 判断代龄列表中的对象数量是否超出阀值。
        if (generations[i].count > generations[i].threshold) {
            // ... 一些性能方面的考量 ...

            // 开始执行垃圾回收
            n = collect(i);

            // 这个似乎有点莫名其妙……
            //   如果 2 级代龄可回收，就不再继续循环?
            break;
        }
    }
    return n;
}

static Py_ssize_t collect(int generation)
{
    ... ...

    // 合并代龄?
    //   如果是回收 2 级代龄，那么合并 gen0, gen1, gen2。如果是回收 1 级，则合并 gen0, gen1。
    //   一次处理所有的对象，这似乎是 collect_generations 提前 break 循环的原因。
    for (i = 0; i < generation; i++) {
        gc_list_merge(GEN_HEAD(i), GEN_HEAD(generation));
    }

    ... ...

    // 处理可达对象。
    update_refs(young);

    // 处理因循环引用导致的可达对象。
    subtract_refs(young);

    // 将所有不可达对象移到 unreachable 中。
    gc_list_init(&unreachable);
    move_unreachable(young, &unreachable);

    // 将可达对象 "移到" 老一辈代龄中。
    if (young != old) {
        if (generation == NUM_GENERATIONS - 2) {
            long_lived_pending += gc_list_size(young);
        }
```

```
        gc_list_merge(young, old);
    }
    else {
        long_lived_pending = 0;
        long_lived_total = gc_list_size(young);
    }

    // 将那些包含 __del__ 的家伙转移到 finalizers。
    gc_list_init(&finalizers);
    move_finalizers(&unreachable, &finalizers);

    // 如果被 "finalizers 列表中的对象" 所引用，那么算你倒霉，一同关到这个 "监狱" 里。
    //   因为不知道这些即将 "秋后问斩" 倒霉鬼的 __del__ 函数会不会访问这些被牵连的家伙。
    move_finalizer_reachable(&finalizers);

    ... ...

    // 处理待回收对象的弱引用，包括调用回调函数。
    m += handle_weakrefs(&unreachable, old);

    // 开始回收不可达对象。会打破循环引用，并释放某些 finalizers 里的对象。
    delete_garbage(&unreachable, old);

    ... ...

    // 将那些无法清除的 "钉子户" 丢到 garbage 列表中。
    (void)handle_finalizers(&finalizers, old);

    ... ...
}
```

头晕眼花了吧？还是用 Python 做些实验吧？不用的 IPython 的原因是避免那一大堆的环境变量带来干扰。

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-

import gc

gc.set_debug(gc.DEBUG_STATS | gc.DEBUG_LEAK)

class A(object):
    pass

def main():
    a, b, c = A(), A(), A()
    print a, b, c

    a.x = b
    b.x = c
    c.x = a

    gc.collect()
```

```
if __name__ == "__main__":
    main()
```

输出:

```
$ ./main.py
<__main__.A object at 0x103b6dcd0> <__main__.A object at 0x103b89090> <__main__.A object at
0x103b892d0>
gc: collecting generation 2...
gc: objects in each generation: 770 3064 0
gc: done, 0.0009s elapsed.
gc: collecting generation 2...
gc: objects in each generation: 0 0 3338
gc: collectable <A 0x103b6dcd0>
gc: collectable <A 0x103b89090>
gc: collectable <A 0x103b892d0>
gc: collectable <dict 0x103a84e60>
gc: collectable <dict 0x103a84ac0>
gc: collectable <dict 0x103a84d00>
gc: done, 6 unreachable, 0 uncollectable, 0.0006s elapsed.
```

从输出结果，我们可以看到 a, b, c 这个间接循环引用被 GC 正确回收。回收对象中还包括三个 dict，它们分别是 a, b, c 的 <instance>.__dict__。

再试试由 "循环引用 + __del__" 共同打造的 "钉子户"。

```
class A(object):
    def __del__(self):
        print "dead"

def main():
    a, b, c = A(), A(), A()
    print a, b, c

    a.x = b
    b.x = c
    c.x = a

    gc.collect()
```

输出:

```
$ ./main.py
gc: collecting generation 0...
gc: objects in each generation: 770 3064 0
gc: done, 0.0002s elapsed.
<__main__.A object at 0x10ecf92d0> <__main__.A object at 0x10ecf9310> <__main__.A object at
0x10ecf9350>
gc: collecting generation 2...
gc: objects in each generation: 4 3367 0
gc: done, 0.0008s elapsed.
gc: collecting generation 2...
```

```
gc: objects in each generation: 0 0 3334
gc: uncollectable <A 0x10ecf92d0>
gc: uncollectable <A 0x10ecf9310>
gc: uncollectable <A 0x10ecf9350>
gc: uncollectable <dict 0x10eb84060>
gc: uncollectable <dict 0x10eb848d0>
gc: uncollectable <dict 0x10eb84c30>
gc: done, 6 unreachable, 6 uncollectable, 0.0006s elapsed.
```

一堆 "uncollectable ..."。哎！谁说 "拆迁队" 就一定牛叉，这不就没干过 "钉子户" 吗。

总结一下 GC 可能导致的性能问题：

- 为了跟踪对象，增加了 PyGC_HEAD 头和链表操作，导致更多的内存消耗。
- 当对象 (被跟踪类型) 大量创建时，会因 gen_0 频发超出阀值而导致多次垃圾回收操作。

凡事有利有弊，了解 GC 的相关背景，有助于我们更加合理使用垃圾回收机制。该关闭的时候，就不要犹豫。

1.7 编译和反编译

不用怀疑，Python 是一种 "编译型" 语言，和 Java、.NET 一样基于字节码 (Byte Code) 和栈式虚拟机 (Stack-based VM) 架构。

栈式虚拟机是对真实机器的一种抽象实现，它使得字节码可以运行在不同的硬件平台上。其执行机制和我们所熟悉的 80x86 有所不同，没有寄存器概念，所有指令集 (Instruction Set) 都通过堆栈 (Stack) 进行运算。首先将参数入栈，然后进行指令运算。

相较 Java 和 .NET 而言，Python 虚拟机抽象层次更高，它的字节码指令恐怕也是最好 "阅读" 的。因为缺少将字节码编译成本地代码 (native code) 的 JIT 编译器，Python 性能一直是个很大的 "缺陷"。不过有一些优秀的第三方产品如 Psyco、PyPy 等可供选择，他们的 JIT 编译器在某些时候甚至有上百倍的性能提升。虽然在源码兼容上有些许限制，但这不应该是我们忽略他们的理由。

本节要关注的依然是官方 CPython 对于编译的实现，至于 JIT 话题，可另文详谈。

1.7.1 编译

所有源代码都必须转换成字节码才能被虚拟机执行，这点我们在前面做虚拟机流程分析时就已经知道。虚拟机执行核心是 Python/ceval.c 中的 PyEval_EvalFrameEx 函数，它会读取所有的字节码指令和参数，并调用相关底层函数来获得最终的执行结果。

对于 Python 编译，你的第一反应可能是 pyc/pyo 文件。这个先不忙，我们首先要分析的是：编译什么时候发生，编译结果是什么？

如果没有通过相关命令进行预先编译，那么这个事件通常发生在导入模块那一刻。

Python/import.c

```
/* Load a source module from a given file and return its module
   object WITH INCREMENTED REFERENCE COUNT.  If there's a matching
   byte-compiled file, use that instead. */

static PyObject * load_source_module(char *name, char *pathname, FILE *fp)
{
    ... ...

    // pyc/pyo 的文件名
    cpathname = make_compiled_pathname(pathname, buf,
                                       (size_t)MAXPATHLEN + 1);

    // check_compiled_module 函数检查 pyc/pyo 是否存在，并核对 magic、修改时间等信息。
    if (cpathname != NULL && (fpc = check_compiled_module(pathname, st.st_mtime, cpathname))) {

        // 检查通过，直接读取编译好的文件。
        co = read_compiled_module(cpathname, fpc);
        ... ...
    }
    else {
        // 编译文件不存在，或者 maigc、修改时间什么的对不上，那么重新分析并编译。
        co = parse_source_module(pathname, fp);
        ...
        if (cpathname) {
            // 生成编译文件
            PyObject *ro = PySys_GetObject("dont_write_bytecode");
            if (ro == NULL || !PyObject_IsTrue(ro))
                write_compiled_module(co, cpathname, &st);
        }
    }
    m = PyImport_ExecCodeModuleEx(name, (PyObject *)co, pathname);
    return m;
}

static FILE * check_compiled_module(char *pathname, time_t mtime, char *cpathname)
{
    fp = fopen(cpathname, "rb");

    // 打开失败，表示编译文件不存在。
    if (fp == NULL) return NULL;

    // 核对 magic
    magic = PyMarshal_ReadLongFromFile(fp);
    if (magic != pyc_magic) {
        ...
        return NULL;
    }

    // 核对修改时间。如果和源码时间不符，表示源码被修改过，自然要重新编译。
    pyc_mtime = PyMarshal_ReadLongFromFile(fp);
```

```
    if (pyc_mtime != mtime) {
        ...
        return NULL;
    }
    ...
}

static PyCodeObject * parse_source_module(const char *pathname, FILE *fp)
{
    ...
    // 源码 AST 分析
    mod = PyParser_ASTFromFile(fp, pathname, Py_file_input, 0, 0, &flags, NULL, arena);

    // 编译成 PyCodeObject，这个类型很重要，我们稍后分析。
    if (mod) {
        co = PyAST_Compile(mod, pathname, NULL, arena);
    }
    ...

    return co;
}

static void write_compiled_module(PyCodeObject *co, char *cpathname, struct stat *srcstat)
{
     // 获取源码文件的修改时间和权限
    time_t mtime = srcstat->st_mtime;
    mode_t mode = srcstat->st_mode & ~S_IEXEC;

    // 开始写编译文件
    fp = open_exclusive(cpathname, mode);

    // 写入 magic
    PyMarshal_WriteLongToFile(pyc_magic, fp, Py_MARSHAL_VERSION);

    // 先写入 0L，为修改时间占位。
    PyMarshal_WriteLongToFile(0L, fp, Py_MARSHAL_VERSION);

    // 将 PyCodeObject 对象写入文件。
    PyMarshal_WriteObjectToFile((PyObject *)co, fp, Py_MARSHAL_VERSION);

    ...

    // 回到开始位置，写入源码修改时间。以便下次载入 pyc/pyo 时判断，源码是否被更新过。
    fseek(fp, 4L, 0);
    PyMarshal_WriteLongToFile((long)mtime, fp, Py_MARSHAL_VERSION);
}
```

PyAST_Compile 生成的这个 PyCodeObject 到底是什么?

Include/code.h

```
/* Bytecode object */
typedef struct {
```

```
    PyObject_HEAD
    int co_argcount;          /* 参数个数，不包括 *args, **kwargs */
    int co_nlocals;           /* 局部变量数量 */
    int co_stacksize;         /* 执行所需的栈空间 */
    int co_flags;             /* 编译标志，在创建 Frame 时用得着 */
    PyObject *co_code;        /* 字节码指令 */
    PyObject *co_consts;      /* 常量列表 */
    PyObject *co_names;       /* 符号列表 */
    PyObject *co_varnames;    /* 局部变量名列表 */
    PyObject *co_freevars;    /* 为闭包准备的东西... */
    PyObject *co_cellvars;    /* 还是闭包要的东西... */
    /* The rest doesn't count for hash/cmp */
    PyObject *co_filename;    /* 源码文件名 */
    PyObject *co_name;        /* PyCodeObject 的名字，函数名、类名什么的 */
    int co_firstlineno;       /* 这个 PyCodeObject 在源码文件中的起始位置，也就是行号 */
    PyObject *co_lnotab;      /* 字节码指令偏移量和源码行号的对应关系，反汇编时用得着 */
    void *co_zombieframe;     /* 为优化准备的特殊 Frame 对象 */
    PyObject *co_weakreflist; /* 为弱引用准备的... */
} PyCodeObject;
```

很显然，PyCodeObject 除了字节码指令，还包含了一个源码对象的所有细节。在编译一个模块时，其内部的函数、类同样也会被编译成 PyCodeObject 对象，并嵌套保存在 co_consts 字段中。

看下面的示例：

```
In [1]: !cat test.py
# -*- coding:utf-8 -*-

def add(a, b):
    return a + b

add(100, 200)


In [2]: code = compile(open("test.py").read(), "test.py", "exec")

In [3]: code.co_filename, code.co_name, code.co_names
Out[3]: ('test.py', '<module>', ('add',))

In [4]: code.co_consts
Out[4]: (<code object add at 0x102745c30, file "test.py", line 3>, 100, 200, None)

In [5]: code.co_consts[0].co_name
Out[5]: 'add'

In [6]: code.co_consts[0].co_varnames
Out[6]: ('a', 'b')
```

除了使用内置函数 compile 直接编译源码外，还可以使用 py_compile、compileall 编译 py 文件。这两个模块都支持在命令行运行，compileall 还可以递归编译多级目录中的所有源文件。

```
In [7]: import py_compile, compileall

In [8]: py_compile.compile("test.py", "test.pyo")

In [9]: ls
main.py*      test.py       test.pyo

In [10]: compileall.compile_dir(".", 0)
Listing . ...
Compiling ./main.py ...
Compiling ./test.py ...
Out[10]: 1
```

对于 "C 语言记忆模糊" 的兄弟，可通过 "...lib/python2.7/compiler/pycodegen.py" 文件来研究字节码生成、pyc 文件格式等内容。

接着说 pyc 文件格式。在前面的 write_compiled_module 函数中，我们已经看到了生成 pyc 的过程。

- 写入 magic。magic 用于判断 pyc 的版本，这里面存在一个不同 Python 版本的兼容问题。
- 写入源文件修改时间。用于判断自上次编译后，源码是否被修改过，以便重新编译。
- 写入 PyCodeObject 对象。递归写入，包括其 co_consts 中的 PyCodeObject 对象。

下面是 pycodegen.py 中的一段代码，除了可以看到 pyc 文件的写入过程外，我们还注意到 PyCodeObject 是通过 marshal.dump 进行序列化写入的。

```
class Module(AbstractCompileMode):

    mode = "exec"

    def compile(self, display=0):
        tree = self._get_tree()
        gen = ModuleCodeGenerator(tree)
        if display:
            import pprint
            print pprint.pprint(tree)
        self.code = gen.getCode()

    def dump(self, f):
        f.write(self.getPycHeader())
        marshal.dump(self.code, f)

    MAGIC = imp.get_magic()

    def getPycHeader(self):
        mtime = os.path.getmtime(self.filename)
        mtime = struct.pack('<i', mtime)
        return self.MAGIC + mtime
```

如果有兴趣，可以翻看 Python/marshal.c 来研究 PyCodeObject 序列化细节。

除了 pyc，还有一种字节码优化后的编译格式 pyo，两者有什么区别?

```
$ python --help
usage: python [option] ... [-c cmd | -m mod | file | -] [arg] ...
Options and arguments (and corresponding environment variables):
... ...
-O     : optimize generated bytecode slightly; also PYTHONOPTIMIZE=x
-OO    : remove doc-strings in addition to the -O optimizations
```

如果我告诉你，多数时候两者生成的字节码没有任何区别，会不会很失望？老实说，CPython 编译器优化能力比起 gcc 这些 "老牛" 要差不少。

看下面这个测试用例。其中的 test 函数实际上没有任何用处，理论上来说优化后的代码除了抛弃 test 外，连 test() 这个调用代码都不应该存在。实际效果如何呢?

```
In [11]: rm *.pyc *.pyo
rm: *.pyc: No such file or directory
rm: *.pyo: No such file or directory

In [12]: cat test.py
# -*- coding:utf-8 -*-

def test():
    i = 0
    i = i + 1

test()



In [13]: import py_compile

In [14]: py_compile.compile("test.py", "test.pyc")

In [15]: py_compile.compile("test.py", "test.pyo")

In [16]: !md5 test.pyc test.pyo
MD5 (test.pyc) = 47d9f61236690ccc5ecdcca06b43f993
MD5 (test.pyo) = 47d9f61236690ccc5ecdcca06b43f993
```

通过比对 MD5 值，可以确定编译生成的 pyc、pyo 完全相同。如果担心 py_compile.compile 的正确性，看看命令行的结果。

```
$ rm *.pyc *.pyo

$ python -c "import test"
$ python -O -c "import test"

$ md5 test.pyc test.pyo
MD5 (test.pyc) = 47d9f61236690ccc5ecdcca06b43f993
MD5 (test.pyo) = 47d9f61236690ccc5ecdcca06b43f993
```

是不是 pyc 就已经优化过了呢？pyo 已经没啥可做得了，所以一样？继续看，要有心理准备。

```
In [17]: import test

In [18]: test?               # 确认是从 pyc 载入
Type:       module
String Form:<module 'test' from 'test.pyc'>

In [19]: import dis

In [20]: dis.dis(test)       # 反编译
Disassembly of test:
  4           0 LOAD_CONST               1 (0)
              3 STORE_FAST               0 (i)

  5           6 LOAD_FAST                0 (i)
              9 LOAD_CONST               2 (1)
             12 BINARY_ADD
             13 STORE_FAST               0 (i)
             16 LOAD_CONST               0 (None)
             19 RETURN_VALUE
```

看着 dis.dis 的输出结果，有没有泪奔的冲动？或者你会想，生成 test 的字节码是傻了点，但如果模块中根本不调用这个毫无用处的函数，不也就达到优化的目的了吗？

```
In [21]: code = compile(open("test.py").read(), "test.py", "exec")

In [22]: dis.dis(code)
  3           0 LOAD_CONST               0 (<code object test, file "test.py", line 3>)
              3 MAKE_FUNCTION            0
              6 STORE_NAME               0 (test)

  7           9 LOAD_NAME                0 (test)
             12 CALL_FUNCTION            0
             15 POP_TOP
             16 LOAD_CONST               1 (None)
             19 RETURN_VALUE
```

受打击啊。当然，某些时候 pyo 与 pyc 还是有点差别的，况且 Python 的 "-O" 参数早已说明是 "optimize generated bytecode slightly"。有关具体的优化内容可以参考 Python/peephole.c 文件。

想要提升 Python 代码的执行性能，寄希望于 "python -O" 是没有用的。我再次推荐各位兄弟去尝试 PyPy 等项目。当然，我并不是贬低 CPython。因为大多数时候，我们并不需要去关注这些性能问题，Python 语言设计初衷也不是冲着和 C 语言一样的高性能去的。

在一个系统中，语言往往不是最大的性能瓶颈，反而是算法和架构造成的损失更多。比如，我们可以用 greenlet/gevent 来改良多线程模型，以提升系统并发和处理能力。等等……

我没有漏掉字节码的内容，往下看。

1.7.2 反编译

又常被称作 "反汇编"，非常重要的逆向工程手段，可用来查看编译器优化结果，或者 "还原" 某段感兴趣的代码，甚至跳过某些 "限制"。反汇编面对的通常是二进制可执行文件，比如 Windows 下的 PE (exe/dll)，Linux 下的 ELF 等。

可为什么要反编译 Python Byte Code?

- 原因一：就像前面章节所做的那样，看看虚拟机是如何执行代码的。可更 "清晰" 的观察分析类似 LEGB、闭包这些 "奇怪" 的语言概念。
- 原因二：只有 pyc/pyo，但好奇心又很重，就可以一窥究竟了。(其实就是闲得无聊)

看下面一个简单的 "代码还原" 过程演示。

```
In [1]: import test, dis

In [2]: test?                # 从 pyc 导入的，显然没有源码。(废话！有源码就不废这劲了)
Type:       module
String Form:<module 'test' from 'test.pyc'>
File:       /Users/test/python/test.pyc

In [3]: test.                # 按 <TAB> 看看里面都有什么? test.add? 就看这个吧。
test.add  test.pyc

In [4]: test.add?            # 查看 test.add 的相关信息。是个函数，有 x、y 两个参数，这很重要。
Type:       function
File:       /Users/test/python/test.py
Definition: test.add(x, y)

In [5]: dis.dis(test.add)    # 反汇编
  4           0 LOAD_FAST                0 (x)             # 载入本地参数 x, y
              3 LOAD_FAST                1 (y)
              6 BINARY_ADD                                 # 将 x, y 相加
              7 STORE_FAST               2 (z)             # 存到本地变量 z
                                                           # 代码: z = x + y

  5          10 LOAD_CONST               1 ('add:')        # 载入常量 "add:"
             13 PRINT_ITEM                                 # 显示该常量
             14 LOAD_FAST                2 (z)             # 载入本地变量 z
             17 PRINT_ITEM                                 # 显示 z
             18 PRINT_NEWLINE                              # 还得在加个换行
                                                           # 代码: print "add:", z 这个默认就换行
  6          19 LOAD_FAST                2 (z)             # 载入本地变量 z
             22 RETURN_VALUE                               # 作为函数返回值
                                                           # 代码: return z
```

这个反编译过程虽然简单，但基本上可以作为 "Hello, World!" 级别的教程了。

先解释一下 dis 输出的结果：

- 第 1 列：该段指令的源码在 py 文件中的行号。我们依此与源代码文件建立关联。
- 第 2 列：指令偏移量。我们可以使用这些位置来 "手工调整" 字节码指令。(嘿嘿……)
- 第 3 列：字节码指令。在标准库 dis 帮助文件里，我们可以看到所有指令帮助和参数说明。
- 第 4 列：指令参数。数字表示参数在变量列表中的序号，括号内是参数对应符号。

1.7.2.1 偏移量、行号

不需要折腾 code.co_lnotab，dis.findlinestarts 函数就可以输出 (偏移量，行号) 的对应关系。

```
In [6]: [(line, offset) for offset, line in dis.findlinestarts(test.add.func_code)]
Out[6]: [(4, 0), (5, 10), (6, 19)]
```

1.7.2.2 字节码指令

可以通过 dis.opname 输出指令集编号，或者查看 opcode.h 文件。

```
In [7]: for index, op in enumerate(dis.opname):
   ....:     print "{0:02X} {1:03}: {2}".format(index, index, op)
   ....:
00 000: STOP_CODE
... ...
17 023: BINARY_ADD
... ...
7C 124: LOAD_FAST
7D 125: STORE_FAST
... ...
FF 255: <255>
```

Include/opcode.h

```
#define BINARY_ADD    23
#define LOAD_FAST     124    /* Local variable number */
#define STORE_FAST    125    /* Local variable number */
```

标准库手册 "31.12. dis — Disassembler for Python bytecode" 包含全部指令说明。

LOAD_FAST(var_num)

Pushes a reference to the local co_varnames[var_num] onto the stack.

BINARY_ADD()

Implements TOS = TOS1 + TOS.

STORE_FAST(var_num)

Stores TOS into the local co_varnames[var_num].

Python 字节码指令并不多，格式如下：

- op < 90 (0x5A): 指令长度 1 byte，无参数。
- op >= 90 : 指令长度 1 byte，1 个 2 bytes 长的参数。

至于偏移量、指令和参数，只需输出 co_code 十六进制数据，和 dis 结果一对比这明白了。

```
In [8]: " ".join(["%02x" % ord(c) for c in test.add.func_code.co_code])
Out[8]: '7c 00 00 7c 01 00 17 7d 02 00 64 01 00 47 7c 02 00 47 48 7c 02 00 53'
#偏移量:  0        3        6  7        10 ...
#指  令:  7c       7c       17 7d
#参  数:     0000     0001        0002
```

```
  4           0 LOAD_FAST                0 (x)
              3 LOAD_FAST                1 (y)
              6 BINARY_ADD
              7 STORE_FAST               2 (z)
  ... ...
```

1.7.2.3 指令参数

查看 code 对象的相关信息，很容易找到指令参数序号和名字的对应关系。

```
In [9]: test.add.func_code.co_nlocals       # locals 名字空间成员数量
Out[9]: 3

In [10]: test.add.func_code.co_varnames     # 本地变量名列表
Out[10]: ('x', 'y', 'z')

In [11]: test.add.func_code.co_consts       # 常量列表
Out[11]: (None, 'add:')
```

想要修改字节码也不难，要么直接修改 pyc 文件，要么先用 marshal 序列化 code 对象，待修改完成后再反序列化。调整字节码时，多余的位置用 NOP 填充，如此就不需要修改偏移量信息了。有兴趣的可以自己动手试试。

1.7.3 动态执行

作为动态语言的 Python 还需要动态执行吗？不要争论这个话题，我们先看标准库中的一个例子。

```
In [1]: from collections import namedtuple

In [2]: User = namedtuple("User", "id, name, age")

In [3]: u1 = User(1, "User1", 10)

In [4]: u1
Out[4]: User(id=1, name='User1', age=10)
```

```
In [5]: u1.name
Out[5]: 'User1'
```

namedtuple 的实现代码 (节选)。

lib/python2.7/collections.py

```
def namedtuple(typename, field_names, verbose=False, rename=False):
    """ Returns a new subclass of tuple with named fields. """

    # 分割 field_names
    if isinstance(field_names, basestring):
        field_names = field_names.replace(',', ' ').split()

    # 转换为字符串
    field_names = tuple(map(str, field_names))

    ...

    # 一段 "class ..." 字符串模版。
    template = '''
        class %(typename)s(tuple):
            __slots__ = () \n
            _fields = %(field_names)r \n

            def __new__(_cls, %(argtxt)s):
                return _tuple.__new__(_cls, (%(argtxt)s)) \n

            ... 省略部分内容 ...

            def __getnewargs__(self):
                return tuple(self) \n\n
    ''' % locals()

    # 循环 field_names，将 property 代码添加到模版字符串后面。
    for i, name in enumerate(field_names):
        template += "%s = _property(_itemgetter(%d), doc='Alias for field number %d')\n" %
                    (name, i, i)
    ...

    try:
        exec template in namespace
    except SyntaxError, e:
        raise SyntaxError(e.message + ':\n' + template)

    result = namespace[typename]
    # ...
    return result
```

简单点说，就是 "exec source_code_str in namespace"。类似的东西还有什么？代码生成器？ORM？

Python 提供了 eval、exec、execfile、runpy 等多种动态执行方法。

eval 执行一个表达式并返回结果，可带入自定义的名字空间 (默认使用 globals、locals)。

```
In [6]: x = 1

In [7]: eval("x + 10")
Out[7]: 11

In [8]: eval("x + y", None, {"y":10})
Out[8]: 11
```

经常在论坛上看到有人问，如何将一个 dict 字符串转换为 dict 对象。

```
In [9]: s
Out[9]: "{'a': 1, 'b': 'abc'}"

In [10]: eval(s)
Out[10]: {'a': 1, 'b': 'abc'}
```

exec 执行代码语句，可使用 in 带入名字空间。

```
In [11]: code = """
   ....: def add():
   ....:     global z
   ....:     z = x + y
   ....: add()"""
   ....:

In [12]: namespace = dict(x = 10, y = 20, z = None)

In [13]: exec code in namespace

In [14]: namespace.keys()
Out[14]: ['y', 'x', 'add', 'z', '__builtins__']

In [15]: namespace["z"]
Out[15]: 30
```

execfile 执行一个 py 文件，使用方法类似，此处就不凑字了。

还记得在分析虚拟机执行流程时提到的那个 run_mod 吗？(1.3.7 Run)

Python/pythonrun.c

```
static PyObject *
run_mod(mod_ty mod, const char *filename, PyObject *globals, PyObject *locals,
         PyCompilerFlags *flags, PyArena *arena)
```

runpy.run_module 和这个类似。在无需 "import <module>" 的情况下 "执行" 模块，通常返回模块的 globals 名字空间字典。当然，如果模块是 "可执行" 的，那么就相当于执行 "python -m <module_name>"。

```
In [16]: runpy.run_module("pdb", None, "__main__")
usage: pdb.py scriptfile [arg] ...
An exception has occurred, use %tb to see the full traceback.

SystemExit: 2

To exit: use 'exit', 'quit', or Ctrl-D.
```

```
$ python -m pdb
usage: pdb.py scriptfile [arg] ...

$ echo $?
2
```

runpy 里面还有个 run_path，用于执行某个 py 文件。

1.7.4 代码混淆

代码混淆 (Obfuscated Code) 在 Java、.NET 圈子被广泛使用，算是一种初级的软件安全加密手法。总有那么些时候，我们需要为我们的代码设置一点 “人为障碍”，避免被 "意外修改"。

当然，我们可以只发布 pyc/pyo 二进制文件。但某些时候还是会存在 "源码 + 保护" 的尴尬。这也就是本节的目的，保护源码文件中某些敏感的算法。

先看下面混淆后的源码文件。

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-

import marshal

def rekey(s, key):
    if len(key) < len(s):
        d, m = divmod(len(s), len(key))
        key *= m and d + 1 or d

    return key


def dec(code, key):
    key = rekey(code, key)
    chrs = [chr(a ^ ord(b)) for a, b in zip(code, key)]
    return marshal.loads("".join(chrs))


code = [
```

```
    2, 98, 99, 97, 98, 99, 97, 98, 99, 96, 98, 99, 97, 34, 99, 97, 98, 16,
    108, 98, 99, 97, 6, 99, 97, 230, 99, 97, 56, 99, 97, 6, 98, 97, 49, 75,
    99, 98, 99, 97, 1, 97, 97, 98, 99, 99, 98, 99, 97, 96, 99, 97, 98, 32,
    97, 98, 99, 18, 115, 99, 97, 98, 7, 96, 98, 36, 29, 98, 99, 29, 99, 99,
    118, 37, 43, 5, 98, 99, 50, 74, 97, 97, 98, 99, 47, 17, 103, 97, 98, 99,
    0, 6, 7, 91, 74, 99, 97, 98, 99, 73, 96, 99, 97, 98, 23, 96, 98, 99, 97,
    3, 23, 96, 98, 99, 97, 0, 75, 97, 98, 99, 97, 74, 99, 97, 98, 99, 21, 98,
    99, 97, 98, 23, 98, 98, 99, 97, 3, 7, 5, 99, 99, 97, 98, 16, 97, 98, 99,
    97, 44, 75, 96, 98, 99, 97, 48, 96, 97, 98, 99, 73, 98, 99, 97, 98, 75,
    97, 98, 99, 97, 74, 99, 97, 98, 99, 51, 96, 99, 97, 98, 23, 105, 98, 99,
    97, 94, 14, 14, 6, 22, 13, 7, 93, 96, 98, 99, 97, 17, 99, 97, 98, 99
]

def main():
    exec dec(code, "abc")
    add(10, 11)

if __name__ == "__main__":
    main()
```

被保护的是一个非常简单的函数:

```
def add(a, b):
    print "add:", a + b
```

执行结果:

```
$ ./main.py
add: 21
```

从解密函数 dec 我们也能推导出加密过程:

- 首先使用 compile 将被保护函数代码编译成 code 对象。
- 使用 marshal.dumps 将其序列化成 str。
- 使用 key 对字符串进行 xor 处理，返回一个被 "加密" 的整数列表。

简单吧，但的确起到了 "代码保护" 目的。当然，这种简单的加密方法并不能阻止有心人的窥探，只要通过 dis 反汇编字节码就可以 "还原" 算法的本来面目。如果你需要更安全的软件加密措施，不妨考虑将核心算法用 C 来编写，或者干脆使用更 "安全" 的语言。

就上面的演示而言，你还需要思考如何保护 key，比如运行时从服务器获取。还可以在加密结果中添加 MD5 签名，以避免源码中的混淆数据被意外修改。

下面给出加密函数，你可以为不同的用户设置不同的 key/sn，然后写个小程序替换源码中的整数列表，进行批量 "生产"。

```
def enc(source, key):
    code = compile(source, "", "exec")
    s = marshal.dumps(code)
    key = rekey(s, key)
    return [ord(a) ^ ord(b) for a, b in zip(s, key)]
```

```
enc('def add(a, b): print "add:", a + b', key)
```

1.8 小结

还好，我们都还活着。

在我发布这章预览版的时候，一些兄弟建议将其中某些内容放到后续章节中。我对此倒有不同的想法:

首先，这本书的定位并非入门书籍，因此不会按部就班地从易到难。对于读者已经做了一定的预设条件，应该是有一定编程经验，期望有所提高的伙计。

其次，在后续章节的分析中，会大量使用本章所录的一些知识背景。如果不了解这些，免不了又流于表面。那就和作者本意相去甚远了。

最后呢，我觉得这章的内容虽稍有深度，但并没有什么迈不过去的门槛。有些基础的兄弟，相信都能看懂。无非是耐心的问题，能否踏踏实实去翻看 CPython 的源码，能否把所有的测试代码尝试一遍。

我个人觉得本章的内容确实是基础，无非是对一些看着复杂的东西做次 "全景" 介绍而已，比如类型的很多东西都未曾细说。有了 "全局印象"，再看后面的局部细节，自然就 "了然于胸"。否则前后牵扯，就会像很多书那样巴拉巴拉絮叨一遍，痛苦死了。

这章里面留下了许多 "坑"，足够让好奇心重的兄弟们打发时间了。随后几章的内容虽然看上去平和一些，但归根结底还是在本章的基础上继续发挥而已。

好了，继续上路吧…… 前面铺满鲜花的山坡上，貌似有 "魔鬼" 在挥手召唤…… 哈哈……

第 2 章 内置类型

Python 将所有的对象都丢在堆上。就算一个整数 0，也是一个标准的，有类型指针和引用计数的对象。很显然，这远比在栈上通过 esp、ebp 寄存器操作一段内存复杂很多。如何在保持面向对象模型的前提下，有效提高 int 的性能呢？Java、C# 为此搞出了个 "box/unbox"，Python 又会做什么？

还有 list、dict 这些我们使用频率极高的数据结构，又是如何处理的？在我们创建一个 "巨大" 的列表时，需要注意哪些性能问题？

很多人在学习 Python 的时候，对这些内置数据类型，一带而过。在你喋喋不休炫耀 OOP 魔法的同时，可能正在累积一场灾难。

按照用途不同，可以将 Python 内置类型分为 "数据结构" 和 "程序结构" 两种。本章将探索 int、dict、set 这类数据结构的实现细节，而 FunctionType、MethodType 这些则留待后续章节再做详述。

2.1 数字

Python 中的数字类型包括 bool、int、long、float、complex。没错，我将 bool 也放在数字这里，因为 Python 把 bool 当作一种特殊的数字对待。

2.1.1 bool

Objects/boolobject.h

```
typedef PyIntObject PyBoolObject;
```

所有零值或内容长度为零的对象都被视为 False。内容长度为零？意味着什么？

Objects/object.c

```
int PyObject_IsTrue(PyObject *v)
{
    Py_ssize_t res;
    if (v == Py_True) return 1;
    if (v == Py_False) return 0;

    if (v == Py_None)
        return 0;
    else if (v->ob_type->tp_as_number != NULL && v->ob_type->tp_as_number->nb_nonzero != NULL)
        res = (*v->ob_type->tp_as_number->nb_nonzero)(v);
    else if (v->ob_type->tp_as_mapping != NULL && v->ob_type->tp_as_mapping->mp_length != NULL)
        res = (*v->ob_type->tp_as_mapping->mp_length)(v);
    else if (v->ob_type->tp_as_sequence != NULL && v->ob_type->tp_as_sequence->sq_length != NULL)
        res = (*v->ob_type->tp_as_sequence->sq_length)(v);
    else
```

```
        return 1;
    /* if it is negative, it should be either -1 or -2 */
    return (res > 0) ? 1 : Py_SAFE_DOWNCAST(res, Py_ssize_t, int);
}
```

```
In [1]: any([None, 0, "", (), {}, set(), frozenset()])
Out[1]: False

In [2]: bool(-1)      # 这个别搞错了，负数可不是零。
Out[2]: True
```

既然 bool 是数字，下面的做法也没什么可奇怪的。

```
In [3]: "abc"[True]
Out[3]: 'b'

In [4]: "abc"[False]
Out[4]: 'a'
```

2.1.2 int

在分析虚拟机初始化流程时，我们就发现 Python 对 PyInt_Type 的特殊照顾。这个看似简单的整数类型背后貌似隐藏着诸多的秘密。

老规矩，先看看创建 PyIntObject 时到底都发生了什么。

Include/intobject.h

```
typedef struct {
    PyObject_HEAD
    long ob_ival;
} PyIntObject;
```

Objects/intobject.c

```
#define NSMALLPOSINTS           257
#define NSMALLNEGINTS           5

PyObject * PyInt_FromLong(long ival)
{
    register PyIntObject *v;

    if (-NSMALLNEGINTS <= ival && ival < NSMALLPOSINTS) {

        // [-5, 257) 之间的小整数，直接从缓存数组中返回。
        v = small_ints[ival + NSMALLNEGINTS];
        ...
        return (PyObject *) v;
    }

    // 这是什么?
    if (free_list == NULL) {
        if ((free_list = fill_free_list()) == NULL) return NULL;
```

```
    }

    // 后面这些看着云山雾罩的，@#%$#*&( ...

    /* Inline PyObject_New */
    v = free_list;
    free_list = (PyIntObject *)Py_TYPE(v);
    PyObject_INIT(v, &PyInt_Type);
    v->ob_ival = ival;
    return (PyObject *) v;
}
```

"小整数" 缓存已经是老熟人了，直接通过 small_ints 数组下标访问。问题是其他整数对象貌似也不简单啊。这个神秘的 free_list 究竟是什么？不管了，先拿 fill_free_list 函数开刀。

```
#define BLOCK_SIZE      1000    /* 1K less typical malloc overhead */
#define BHEAD_SIZE      8       /* Enough for a 64-bit pointer */
#define N_INTOBJECTS    ((BLOCK_SIZE - BHEAD_SIZE) / sizeof(PyIntObject))

// PyIntBlock 类型，包含了链表字段和一个数组字段。
struct _intblock {
    struct _intblock *next;
    PyIntObject objects[N_INTOBJECTS];
};

typedef struct _intblock PyIntBlock;

// 将多个 PyIntBlock 组成链表
static PyIntBlock *block_list = NULL;

// 将所有 PyIntBlock.objects 数组可用元素空间串成链表
static PyIntObject *free_list = NULL;

static PyIntObject * fill_free_list(void)
{
    PyIntObject *p, *q;

    // 使用 PyMem_MALLOC 直接调用 malloc 分配内存。
    p = (PyIntObject *) PyMem_MALLOC(sizeof(PyIntBlock));

    // 看样子会在需要的时候申请多个 PyIntBlock 对象，并构成一个单向链表。
    ((PyIntBlock *)p)->next = block_list;
    block_list = (PyIntBlock *)p;

    /* Link the int objects together, from rear to front, then return
       the address of the last int object in the block. */

    // p 指向刚申请的 PyIntBlock.objects[0]。
    p = &((PyIntBlock *)p)->objects[0];

    // q 指向 objects 尾部。
    q = p + N_INTOBJECTS;
```

```
    // 通过指针 q 反向循环这个 objects 数组。
    while (--q > p)
        // Py_TYPE 宏返回头部的 ob_type 指针。
        //   #define Py_TYPE(ob) (((PyObject*)(ob))->ob_type)
        // 将数组元素的 ob_type 指针指向前一个元素。用 ob_type 指针将所有数组元素构建成链表，这是想干嘛?
        Py_TYPE(q) = (struct _typeobject *)(q-1);

    // 第一个元素的 ob_type = NULL，链表结束。
    Py_TYPE(q) = NULL;

    // 返回 objects 最后一个元素指针。
    return p + N_INTOBJECTS - 1;
}
```

看上去像个简化版的 arena 设计。

- 申请一块类型为 PyIntBlock 的内存，在 64 位平台上足以保存 41 个 PyIntObject 对象内容。
- 多个 PyIntBlock 对象通过其 next 字段和全局变量 block_list 构成一个链表。
- 将 PyIntBlock.objects 数组元素空间串成链表。做法很巧妙，就是将 ob_type 这个原本用来保存类型指针的字段指向前一个元素地址。最后一个元素 (因为是反向的，其实是 objects[0]) 的 ob_type 被设置 NULL，表示链表结束。
- fill_free_list 函数返回这个链表的第一个元素地址 (其实是 objects 的最后一个元素)。

就算是用脚想，也能明白这个 PyIntBlock.objects 数组就是 PyIntObject 对象的实际存储位置了。回到 PyIntObject 创建函数的尾部，看看非小整数对象如何使用这块内存。

```
PyObject * PyInt_FromLong(long ival)
{
    ... ...

    // 将刚申请的 PyIntBlock 和 free_list 关联起来。
    // 如果为 NULL，则调用 fill_free_list 申请新的 PyIntBlock 空间交给 free_list。
    if (free_list == NULL) {
        if ((free_list = fill_free_list()) == NULL) return NULL;
    }

    // 从 free_list 获取一个可用存储位置。
    v = free_list;

    // 宏操作 Py_TYPE(v) 返回 v->ob_type，我们已经知道这个字段已经被当作链表使用。
    // free_list = v->ob_type，也就是说 free_list 指向链表下一个元素，如此就将 v 从链表中剔除。
    // 提醒一下，free_list 最后一个元素的 ob_type == NULL。如果 v 已经是最后一个可用元素，那么意味着下一次
    //   调用本函数时，就需调用 fill_free_list 函数申请新的 PyIntBlock 内存了。
    free_list = (PyIntObject *)Py_TYPE(v);

    // 通过 PyObject_INIT 宏修正 v 的 ob_type、ob_refcnt 等信息。
    PyObject_INIT(v, &PyInt_Type);

    // 为这个对象赋值。
    v->ob_ival = ival;
```

```
    return (PyObject *) v;
}
```

全局变量 free_list 接管了 fill_free_list 返回的链表。当我们需要存储 PyObjectInt 对象时，只需从该链表 "摘" 一个可用空间即可。看到这，我们应该佩服设计者对 ob_type 使用上的巧思。

那么当某个 PyIntObject 对象被回收时，这空出来的空间又该怎么办呢？显然不可能直接还给操作系统的，别忘了是按照 PyIntBlock 一次申请的大块内存。

Include/intobject.h

```
#define PyInt_CheckExact(op) ((op)->ob_type == &PyInt_Type)
```

Objects/intobject.c

```
static void int_dealloc(PyIntObject *v)
{
    // PyInt_Type 类型检查
    if (PyInt_CheckExact(v)) {
        // 将这个对象的存储空间再次加入 free_list 链表头部，如此就可以重复使用了。
        //   注意: 如果有多个 PyIntBlock，此时 free_list 就成了一个跨界的链表了。
        Py_TYPE(v) = (struct _typeobject *)free_list;
        free_list = v;
    }
    else
        // 如果不是 int 对象，则调用其类型对象的 free 函数。意外保护?
        Py_TYPE(v)->tp_free((PyObject *)v);
}
```

果然，"释放" 的这块内存重新加入 free_list 链表，如此就实现了可重复使用的 "对象池"。

此时我到能理解设计者为什么不直接使用 arena 了。PyIntObject 是固定大小的对象，而且使用频率是所有类型中最高的。用 PyIntBlock 即简单，又具备更高的效率，因为只需操作 free_list 链表即可。而 arena 因为需要存储未知类型的对象，所以其本身设计有很大的复杂性。

PyIntBlock 不像 arena 那样有机会释放内存，它会一直存在，直到虚拟机 Py_Main 主函数结束进程前才会通过 Py_Finalize 唤醒 PyInt_Fini 出来收拾残局。

```
block_list is a singly-linked list of all PyIntBlocks ever allocated, linked via their next members.
PyIntBlocks are never returned to the system before shutdown (PyInt_Fini)
```

Python/pythonrun.c

```
void Py_Finalize(void)
{
    ... ...

    PyInt_Fini();

    ... ...
}
```

2.1.2.1 小整数缓存初始化

虽然我们已经看过 "小整数缓存" 很多次了，但一直都没有注意到一个问题，那就是 small_ints 数组保存的是 PyIntObject 指针，还从没研究过小整数缓存对象的分配内存。要解释这个问题，须得问那个被虚拟机初始化时特殊照顾的 _PyInt_Init 函数。

```
// 这里存储的是指针，而非对象内容。
static PyIntObject *small_ints[NSMALLNEGINTS + NSMALLPOSINTS];

int _PyInt_Init(void)
{
    PyIntObject *v;
    int ival;

#if NSMALLNEGINTS + NSMALLPOSINTS > 0

    // 循环 [-5, 257) ...
    for (ival = -NSMALLNEGINTS; ival < NSMALLPOSINTS; ival++) {

        // 一样使用 PyIntBlock 和 free_list
        if (!free_list && (free_list = fill_free_list()) == NULL) return 0;

        // 一样将小整数对象保存在 PyIntBlock 中。
        v = free_list;
        free_list = (PyIntObject *)Py_TYPE(v);
        PyObject_INIT(v, &PyInt_Type);
        v->ob_ival = ival;

        // 再将指针添加到 small_ints。
        small_ints[ival + NSMALLNEGINTS] = v;
    }

#endif
    return 1;
}
```

很好，不管大小 PyIntObject 都存储在 PyIntBlock 中。又因为这些小整数被全局变量 small_ints 持有，除了直接使用整数下标快速访问外，还导致它们的引用计数永远不可能为 0，就彻彻底底成为 "长老" 了。至此，本书多次提及的 "[-5, 257)" 缓存问题，才算是彻底交待清楚了。

依照上一节 PyInt_FromLong 函数的分析结果，我们可以证实：

- Small int 是 "特权阶级"，有自己专用的 "房子"，是 "独一无二" 的。
- 其他 int 虽然同住一座城，但却是 "漂"，在 PyIntBlock "临时租房"，互不来往。
- PyIntBlock 按需创建，虽然可重复使用，但从不被拆除。(你马上就会看到隐患)

```
In [1]: a = 10; b = 10; a is b
Out[1]: True
```

```
In [2]: a = 1234; b = 1234; a is b
Out[2]: False
```

2.1.2.2 range 引发的灾难

当我们一次创建并持有很多的整数对象时，你会发现内存出问题了。

用 range 返回一个 "超大" 整数列表演示一下。代码中的 psutil 是一个第三方包，用来获取进程的相关信息，可用 easy_install 安装。

```
# -*- coding:utf-8 -*-

import os
import sys
import psutil
import gc

gc.set_debug(gc.DEBUG_STATS | gc.DEBUG_LEAK)

p = psutil.Process(os.getpid())
mem = lambda: [s % (m / 1024 / 1024) for s, m in zip(("RSS: %dMB", "VMS: %dMB"),
               p.get_memory_info())]

def test():
    x = 0
    count = 100000
    for i in range(count): x += i
    print "ints: {0}, size: {1} MB".format(count, count * sys.getsizeof(1) / 1024 / 1024)


print "A:", mem()
test()
print "B:", mem()
gc.collect()
print "C:", mem()
```

执行输出：

```
$ python test.py

A: ['RSS: 7MB', 'VMS: 2385MB']
ints: 100000, size: 2 MB

B: ['RSS: 11MB', 'VMS: 2388MB']
gc: collecting generation 2...
gc: objects in each generation: 342 1768 4564
gc: done, 0.0021s elapsed.

C: ['RSS: 11MB', 'VMS: 2388MB']
gc: collecting generation 2...
gc: objects in each generation: 0 0 6535
gc: done, 0.0019s elapsed.
```

创建 10 万个 int 对象，这种模拟场景还是比较常见的吧。通过 A、B 点的输出结果，观察 test 调用前后的内存占用，会发现内存并没有被收回。很显然，这是申请过多的 PyIntBlock 造成的。

> Q.yuhen：明知 GC 不管这茬还加进去，是免得某些兄弟们 "贼心不死"。
> 某些兄弟：好歹 GC 把新人统统搬到 gen_2 了吗…… 哈哈……。

换成 xrange，因为迭代器只在每次调用时才返回一个结果，如此就可重复使用 PyIntBlock 的空间了，结果自然好看多了。(Python 3 已经将 range 改造成 xrange 了)

```
A: ['RSS: 7MB', 'VMS: 2385MB']
ints: 100000, size: 2 MB

B: ['RSS: 7MB', 'VMS: 2385MB']
gc: collecting generation 2...
gc: objects in each generation: 342 1768 4564
gc: done, 0.0020s elapsed.

C: ['RSS: 7MB', 'VMS: 2385MB']
gc: collecting generation 2...
gc: objects in each generation: 0 0 6535
gc: done, 0.0018s elapsed.
```

再用 1千万的 range 测试，GC 依然在打酱油，却意外让内存占用有所回落。兴许是操作系统看不下去了。(orz…，还剩 88MB 呢，这算啥回收啊)

```
A: ['RSS: 7MB', 'VMS: 2385MB']
ints: 10000000, size: 228 MB

B: ['RSS: 318MB', 'VMS: 2695MB']
gc: collecting generation 2...
gc: objects in each generation: 342 1768 4564
gc: done, 0.0020s elapsed.

C: ['RSS: 88MB', 'VMS: 2465MB']
gc: collecting generation 2...
gc: objects in each generation: 0 0 6535
gc: done, 0.0019s elapsed.
```

好了，不继续深挖洞了。只是大家要明白，哪怕是最平凡的一个整数背后，也有着 Python 开发者的诸多智慧和心血。当你嚷嚷 "内存杀手" 时，或许要问问自己对 "她" 究竟了解多少。

2.1.3 long

long 唯一特殊的地方就是，丫是个 "吃货"。变长对象，理论上我们可以创建一个仅受进程内存限制的超大号整数。没有缓存，没有 PyLongBlock 之类的。

Include/longintrepr.h

```
struct _longobject {
        PyObject_VAR_HEAD
        digit ob_digit[1];
};
```

Include/longobject.h

```
typedef struct _longobject PyLongObject; /* Revealed in longintrepr.h */

PyLongObject * _PyLong_New(Py_ssize_t size)
{
    ... ...
    return PyObject_NEW_VAR(PyLongObject, &PyLong_Type, size);
}
```

很普通的创建函数，也就是通过 PyObject_MALLOC 分配内存而已。为什么同是整数，待遇差这么多？要知道在 64 位平台上，int 能存储最大为 9223372036854775807 的整数，够大了吧，没多少机会用到 long。反正底层已经有 arena 这层内存管理机制了，没必要再为一个少数派增加成本了。

Include/pyport.h

```
#define LONG_MAX 0X7FFFFFFFFFFFFFFFL
```

当超出 int 的存储范围时，Python 会自动使用 long 存储。留点演示吧，实在没其他的可说了。

```
In [1]: import sys

In [2]: sys.maxint, hex(sys.maxint)
Out[2]: (9223372036854775807, '0x7fffffffffffffff')

In [3]: type(sys.maxint), type(sys.maxint + 1)
Out[3]: (int, long)

In [4]: 1 << 1000                   # 这个忒吓人了。
Out[4]:
107150860718626732094842504906000181056140481170553360744375038837035105112493612249319837881569585
812759467291755314682518714528569231404359845775746985748039345677748242309854210746050623711418779
541821530464749835819412673987675591655439460770629145711964776865421676604298316526243868372056680
69376L

In [5]: sys.getsizeof(1 << 1000)     # 好吧，我知道这很酷。
Out[5]: 160

In [6]: sys.getsizeof(1 << 2000)     # 够了，不要再显摆了。
Out[6]: 292
```

2.1.4 float

打开 float 的相关源码，你可能会很失望，这里面都是熟悉的面孔，没有惊喜。

Include/floatobject.h

```
typedef struct {
    PyObject_HEAD
    double ob_fval;
} PyFloatObject;
```

Objects/floatobject.c

```
#define BLOCK_SIZE      1000    /* 1K less typical malloc overhead */
#define BHEAD_SIZE      8       /* Enough for a 64-bit pointer */
#define N_FLOATOBJECTS  ((BLOCK_SIZE - BHEAD_SIZE) / sizeof(PyFloatObject))

// 与 PyIntBlock 类似
struct _floatblock {
    struct _floatblock *next;
    PyFloatObject objects[N_FLOATOBJECTS];
};

typedef struct _floatblock PyFloatBlock;

// 这个名字也很眼熟
static PyFloatBlock *block_list = NULL;
static PyFloatObject *free_list = NULL;

static PyFloatObject * fill_free_list(void)
{
    PyFloatObject *p, *q;

    // 这个注释算勾引吗？是否意味着，Python 将使用 arena 替代 PyIntBlock、PyFloatBlock，
    //   形成一个统一的底层对象池？
    /* XXX Float blocks escape the object heap. Use PyObject_MALLOC ??? */

    // 为 PyFloatBlock 分配内存
    p = (PyFloatObject *) PyMem_MALLOC(sizeof(PyFloatBlock));

    // 和 block_list 串起来，组成链表。
    ((PyFloatBlock *)p)->next = block_list;
    block_list = (PyFloatBlock *)p;

    // 依旧是利用 ob_type 将 objects 构成一个反向的链表
    p = &((PyFloatBlock *)p)->objects[0];
    q = p + N_FLOATOBJECTS;
    while (--q > p)
        Py_TYPE(q) = (struct _typeobject *)(q-1);

    // 链表结束
    Py_TYPE(q) = NULL;

    return p + N_FLOATOBJECTS - 1;
}

PyObject * PyFloat_FromDouble(double fval)
{
    ... ...
```

```
    // 依旧是用 free_list 链表管理所有可用的存储空间。
    if (free_list == NULL) {
        if ((free_list = fill_free_list()) == NULL) return NULL;
    }

    // 从链表中取出一个存储位置，并将其从链表中剔除。
    op = free_list;
    free_list = (PyFloatObject *)Py_TYPE(op);

    // 重置 ob_type 和 ob_refcnt
    PyObject_INIT(op, &PyFloat_Type);

    // 赋值
    op->ob_fval = fval;

    return (PyObject *) op;
}
```

除了没有类似 small_ints 小整数对象缓存，其他基本和 int 的处理方式相同。也就是说 float 拥有 int 同样的性能优势，也背上 PyFloatBlock 可能成为内存杀手的恶名。

2.2 字符串

字符串在任何语言里都需要小心对待，其引发的种种问题也早已耳熟能详。大多数语言，其中也包括 Python，都将其作为一种不可变对象。并且使用池 (string intern pool) 技术，共享字符串对象，减少内存占用，提升性能。

2.2.1 str

在前面分析对象内存布局时，已经将 PyStringObject 扒开来看过一回。简单点说，它是对 C 字符数组的一种对象化包装。

Include/stringobject.h

```
typedef struct {
    PyObject_VAR_HEAD
    long ob_shash;
    int ob_sstate;
    char ob_sval[1];

    /* Invariants:
     *     ob_sval contains space for 'ob_size+1' elements.
     *     ob_sval[ob_size] == 0.
     *     ob_shash is the hash of the string or -1 if not computed yet.
     *     ob_sstate != 0 iff the string object is in stringobject.c's
     *       'interned' dictionary; in this case the two references
     *       from 'interned' to this object are *not counted* in ob_refcnt.
     */
} PyStringObject;
```

因为 PyStringObject 是不可变对象，所以设计者使用了变长 struct，将包括字符串内容在内的整个对象存储在一块连续内存中。

PyStringObject 的创建过程并不复杂，只是对空字符串和单字符做了 intern 和缓存处理。

Objects/stringobject.c

```
static PyStringObject *characters[UCHAR_MAX + 1];
static PyStringObject *nullstring;

PyObject * PyString_FromStringAndSize(const char *str, Py_ssize_t size)
{
   register PyStringObject *op;

    ... ...

    // 直接返回空字符串对象
    if (size == 0 && (op = nullstring) != NULL) {
        ...
        return (PyObject *)op;
    }

    // UCHAR_MAX = 255，显然单个字符是被缓存的。
    if (size == 1 && str != NULL && (op = characters[*str & UCHAR_MAX]) != NULL)
    {
        return (PyObject *)op;
    }

    ... ...

    // 使用 PyObject_MALLOC，试图在 arena 上分配。
    op = (PyStringObject *)PyObject_MALLOC(PyStringObject_SIZE + size);

    // 设置 ob_type 和 ob_size
    PyObject_INIT_VAR(op, &PyString_Type, size);

    // 还没有计算 hash 值，状态为 NOT_INTERNED，也就是说没有被池化。
    op->ob_shash = -1;
    op->ob_sstate = SSTATE_NOT_INTERNED;

    // 将 c chars 拷贝到 ob_sval 中。
    if (str != NULL)
        Py_MEMCPY(op->ob_sval, str, size);

    // 依旧是 Null-Terminated。
    op->ob_sval[size] = '\0';

    if (size == 0) {
        PyObject *t = (PyObject *)op;

        // intern 池化
        PyString_InternInPlace(&t);
```

```
        op = (PyStringObject *)t;

        // 看到那个 nullstring 了吗？就在这里初始化的。
        nullstring = op;
        Py_INCREF(op);
    } else if (size == 1 && str != NULL) {
        PyObject *t = (PyObject *)op;

        // 单个字符将被 intern。
        PyString_InternInPlace(&t);
        op = (PyStringObject *)t;

        // 放到 characters 数组中，下次就可以直接用数组下标返回了。
        characters[*str & UCHAR_MAX] = op;
        Py_INCREF(op);
    }

    return (PyObject *) op;
}
```

在 PyString_Fini 函数里，必须将这些被缓存的家伙清理掉。

Objects/stringobject.c

```
void PyString_Fini(void)
{
    int i;
    for (i = 0; i < UCHAR_MAX + 1; i++) {
        Py_XDECREF(characters[i]);
        characters[i] = NULL;
    }
    Py_XDECREF(nullstring);
    nullstring = NULL;
}
```

2.2.1.1 intern

除了空字符串、单字符会被自动池化外，我们还可使用内置函数 intern 主动将字符串放入池中。

Python/bltinmodule.c

```
static PyObject * builtin_intern(PyObject *self, PyObject *args)
{
    ... ...
    PyString_InternInPlace(&s);
    return s;
}
```

Objects/stringobject.c

```
static PyObject *interned;

void PyString_InternInPlace(PyObject **p)
{
```

```
    register PyStringObject *s = (PyStringObject *)(*p);
    PyObject *t;

    // 检查对象的 ob_state 字段，确定是否已经被池化。
    if (PyString_CHECK_INTERNED(s)) return;

    // 创建池化字典
    if (interned == NULL) {
        interned = PyDict_New();
        ...
    }

    // 如果已经在池化字典中，则调整相关引用计数。
    t = PyDict_GetItem(interned, (PyObject *)s);
    if (t) {
        Py_INCREF(t);
        Py_DECREF(*p);
        *p = t;
        return;
    }

    // 将字符串保存到池化字典中
    if (PyDict_SetItem(interned, (PyObject *)s, (PyObject *)s) < 0) {
        PyErr_Clear();
        return;
    }

    // 为啥要引用技术减去 2？因为作为被字典保存的 key、value 对象时都被加上 1 了。
    //   考虑到要被回收，所以这个 interned 字典不能纳入引用计数机制。
    Py_REFCNT(s) -= 2;

    // 调整 ob_state 状态。
    PyString_CHECK_INTERNED(s) = SSTATE_INTERNED_MORTAL;
}

void PyString_InternImmortal(PyObject **p)
{
    PyString_InternInPlace(p);
    if (PyString_CHECK_INTERNED(*p) != SSTATE_INTERNED_IMMORTAL) {
        PyString_CHECK_INTERNED(*p) = SSTATE_INTERNED_IMMORTAL;
        Py_INCREF(*p);
    }
}

PyObject * PyString_InternFromString(const char *cp)
{
    PyObject *s = PyString_FromString(cp);
    if (s == NULL) return NULL;
    PyString_InternInPlace(&s);
    return s;
}
```

PyStringObject.ob_state 除了 SSTATE_NOT_INTERNED，还有两种 intern 状态：

- SSTATE_INTERNED_MORTAL：虽然被 intern，但依然是可回收的。
- SSTATE_INTERNED_IMMORTAL：传说中的 "老不死"。

看看如何回收被 intern 的字符串。

Objects/stringobject.c

```
static void string_dealloc(PyObject *op)
{
    switch (PyString_CHECK_INTERNED(op)) {
        // 没啥说的，直接去本函数尾部，free 掉。
        case SSTATE_NOT_INTERNED:
            break;

        // 虽然被池化，但依然会被回收。
        case SSTATE_INTERNED_MORTAL:
            // 为啥等于 3？因为 PyDict_DelItem 函数内部移除 item 时，会减去 key、value 对象的引用计数。
            //   还有 1 个呢？哥们，当前函数的形参你没忘了吧？否则在 DelItem 里面就被咔嚓掉了，
            //   那本函数尾部的 tp_free 岂不哭死。
            Py_REFCNT(op) = 3;

            // 先从 interned 字典中删除，然后该怎么 free，就是 tp_free 的事情了。
            if (PyDict_DelItem(interned, op) != 0)
                Py_FatalError("deletion of interned string failed");

            break;

        // 选自某词典 "Immortal: n. 神仙；不朽人物"，显然是惹不起的家伙。
        case SSTATE_INTERNED_IMMORTAL:
            Py_FatalError("Immortal interned string died.");

        default:
            Py_FatalError("Inconsistent interned string state.");
    }

    Py_TYPE(op)->tp_free(op);
}
```

看看下面那些使用 intern 的字符串，是不是觉得太有必要了。

Objects/stringobject.c

```
PyObject * PyClass_New(PyObject *bases, PyObject *dict, PyObject *name)
     /* bases is NULL or tuple of classobjects! */
{
    docstr= PyString_InternFromString("__doc__");
    modstr= PyString_InternFromString("__module__");
    namestr= PyString_InternFromString("__name__");
    getattrstr = PyString_InternFromString("__getattr__");
    setattrstr = PyString_InternFromString("__setattr__");
    delattrstr = PyString_InternFromString("__delattr__");
    ... ...
}
```

```
... ...
```

好了，回到 IPython，我们试试 intern 这个函数。

```
In [1]: s = "".join(["a", "b", "c"])

In [2]: s is "abc"
Out[2]: False

In [3]: intern(s) is "abc"
Out[3]: True
```

2.2.1.2 join

"使用 join 函数合并字符串……"，想必这是很多和我一样有代码洁癖伙计们的金科玉律。可凡事总要有个原因，我向来不靠背诵来遵守规矩。

先看看 "+" 号，也就是 concat 函数是如何处理的。

Objects/stringobject.c

```
static PyObject * string_concat(register PyStringObject *a, register PyObject *bb)
{
    ... ...

#define b ((PyStringObject *)bb)

    ... ...

    // 计算两个字符串的总长度
    size = Py_SIZE(a) + Py_SIZE(b);

    ... ...

    // 请注意，上面的 size 只是两个 C chars 的长度，还得加上 PyStringObject 本身的长度。
    op = (PyStringObject *)PyObject_MALLOC(PyStringObject_SIZE + size);

    // 初始化相关头部信息
    PyObject_INIT_VAR(op, &PyString_Type, size);
    op->ob_shash = -1;
    op->ob_sstate = SSTATE_NOT_INTERNED;

    // 将两个 C chars 依次拷贝进来。
    Py_MEMCPY(op->ob_sval, a->ob_sval, Py_SIZE(a));
    Py_MEMCPY(op->ob_sval + Py_SIZE(a), b->ob_sval, Py_SIZE(b));

    // 别忘了这个尾巴。
    op->ob_sval[size] = '\0';

    return (PyObject *) op;
}
```

创建新的 PyStringObject 对象保存合并结果。如果是多个字符串，那么必然要多次创建对象，这就是影响性能的问题所在。当然，事情也不是绝对的，比如用 "+"、"*" 这些操作符连接多个字符串常量，编译器在编译期就会将其合并成一个完整的字符串常量。

```
In [4]: def test():
   ...:     a = "abc" + "def"
   ...:     b = "a" * 3
   ...:     print a, b
   ...:

In [5]: dis.dis(test)
  2           0 LOAD_CONST               5 ('abcdef')
              3 STORE_FAST               0 (a)

  3           6 LOAD_CONST               6 ('aaa')
              9 STORE_FAST               1 (b)

              ... ...
```

再看看 join 的实现。

```
static PyObject * string_join(PyStringObject *self, PyObject *orig)
{
    // 分隔符对象和长度
    char *sep = PyString_AS_STRING(self);
    const Py_ssize_t seplen = PyString_GET_SIZE(self);

    ... ...

    // 将要合并的多个字符串 orig 处理成一个序列对象
    seq = PySequence_Fast(orig, "");
    seqlen = PySequence_Size(seq);

    // 如果只有一个字符串，直接返回就行了，没什么要连接的。
    if (seqlen == 1) {
        item = PySequence_Fast_GET_ITEM(seq, 0);
        if (PyString_CheckExact(item) || PyUnicode_CheckExact(item)) {
            Py_INCREF(item);
            Py_DECREF(seq);
            return item;
        }
    }

    // 循环累加序列对象中各项的长度
    for (i = 0; i < seqlen; i++) {
        ... ...

        item = PySequence_Fast_GET_ITEM(seq, i);    // 当前项的长度
        sz += PyString_GET_SIZE(item);              // 累加长度
        if (i != 0) sz += seplen;                   // 别忘了加上分隔符的长度
        ... ...
    }
```

```
    // 按照累加后的长度，创建一个 PyStringObject 对象。因为 sz != 0，因此不会返回 nullstring。
    res = PyString_FromStringAndSize((char*)NULL, sz);

    ... ...

    // 返回 res->ob_sval，也就是 C chars 指针。
    p = PyString_AS_STRING(res);

    // 再次循环序列对象
    for (i = 0; i < seqlen; ++i) {
        size_t n;
        item = PySequence_Fast_GET_ITEM(seq, i);
        n = PyString_GET_SIZE(item);

        // 将当前项字符串拷贝到新建 PyStringObject.ob_sval 中。
        Py_MEMCPY(p, PyString_AS_STRING(item), n);
        p += n;

        // 当然不能少了分隔符
        if (i < seqlen - 1) {
            Py_MEMCPY(p, sep, seplen);
            p += seplen;
        }
    }

    Py_DECREF(seq);
    return res;
}
```

join 函数能一次接收多个字符串参数 (list 或 tuple 等序列对象)，计算总长度后，一次性分配存储内存。如此就能大大减少中间临时对象的创建和内存分配，自然就有更好的性能了。与 join 类似的方法还有 format，当你要拼接类似 SQL、HTML 这类具有模版格式的字符串时，就不要犹豫。

如果只连接两个字符串，反正都要创建一次 PyStringObject 对象，就没必要把问题复杂化，"+" 操作符足矣。

2.2.2 unicode

在折腾相关操作前，我们应该了解什么是 Unicode，和 UTF 又有什么区别。

字符串编码方案是个即简单又麻烦的话题，这主要涉及到字节 (byte) 和字符 (character) 两个不同的概念。

字节是计算机软件系统的最小存储单位，我们可以用 n 个字节表达一个概念，诸如 4 字节的 32 位整数或着其他什么对象之类的，任何对象最终都是通过字节组合来完成存储的。字符也是一种对象，它和具体的语言有关，比如在中文里，"我" 是一个字符，不能生生将其当作几个字节的拼接。

早期的计算机系统编码方案主要适应英文等字母语言体系，ASCII 编码方案的容量足以表达所有的字符，于是每个字符占用 1 个字节被固定下来，这也造成了一定程度上的误解，将字符和字节等同起来。要知道，这个世界上并非只有英文一种字符，中文等东亚语言体系的字符数量就无法用 ASCII 容纳，人们只好用 2 个或者更多的字节来表达一个字符，于是就有了 gb2312、big5 等等种类繁多且互不兼容的编码方案。

随着计算机技术在世界范围内的广泛使用，国际化需求越发重要，以往的字符编码方案已经不适应现代软件开发。于是国际标谁化组织 (ISO) 在参考很多现有方案的基础上统一制定了一种可以容纳世界上所有文字和符号的字符编码方案，这就是 Unicode 编码方案。Unicode 使得我们在一个系统中可以同时处理和显示不同国家的文字。

Unicode 本质上就是通过特定的算法将不同国家不同语言的字符都映射到数字上。比如中文字符 "我" 对应的数字是 25105 (\u6211)。Unicode 字符集 (UCS, Unicode Character Set) 有 UCS-2、UCS-4 两种标准。其中 UCS-2 最多可以表达 U+FFFF 个字符，而 UCS-4 最多可以有 U+FFFFFFFF 个字符，几乎容纳了全世界所有的字符。

Unicode 只是将字符和数字建立了映射关系，但对于计算机而言，要存储和操作任何数据，都得用字节来表示，这其中还涉及到不同计算机架构的大小端问题 (Big Endian, Little Endian)。于是有了几种将 Unicode 字符数字转换成字节的方法：Unicode 字符集转换格式 (UCS Transformation Format)，缩写 UTF。

目前常用的 UTF 格式有：

- **UTF-8**: 不等长方案，西文字符通常只用一个字节，而东亚字符 (CJK) 通常需要三个字节表示。
- **UTF-16**: 用 2 字节无符号整数存储 Unicode 字符，与 UCS-2 对应，适合处理中文。
- **UTF-32**: 用 4 字节无符号整数存储 Unicode 字符。与 UCS-4 对应，多数时候有点浪费内存空间。

UTF-8 具有非常良好的平台适应性和交互能力，因此已经成为很多操作系统平台的默认存储编码方案。而 UTF-16 因为等长，浪费空间较少，拥有更好的处理性能，包括 .NET、JVM 等都使用 2 字节的 Unicode Char。

另外，按照大小端划分，UTF 又有 BE 和 LE 两种格式，比如 UTF-16BE、UTF-16LE 等。为了让系统能自动识别出 BE 和 LE，通常在字符串头部添加 BOM 信息，"FE FF" 表示 BE，"FF FE" 表示 LE。后面还会有一两个字符用来表示 UTF-8 或 UTF-32。

UTF-8 现在大行其道，Linux、OS X 都将其作为默认编码方案，还有越来越多的网页也采取此种格式。Python 能非常方便地在不同的编码间转换，只不过先要解决悲催的 default encoding 问题。(Python 3 取消了 str，全部改用 unicode，貌似这已经是一种趋势)

```
In [1]: s = "中国人"
```

```
In [2]: unicode(s)
---------------------------------------------------------------------------
UnicodeDecodeError: 'ascii' codec can't decode byte 0xe4 in position 0: ordinal not in range(128)

In [3]: import sys; sys.getdefaultencoding()
Out[3]: 'ascii'
```

要想重新设置默认编码，必须找回被 site 干掉的 sys.setdefaultencoding 函数。

```
In [4]: import sys, locale

In [5]: encoding = locale.getdefaultlocale()        # 获取当前系统默认编码

In [6]: encoding
('zh_CN', 'UTF8')

In [7]: reload(sys)                                 # 重新 import sys
<module 'sys' (built-in)>

In [8]: sys.setdefaultencoding(encoding[1])         # 将默认编码设为 UTF-8

In [9]: s = "中国人"

In [10]: s                                          # 每个中文字符占 3 个字节，浪费啊！
'\xe4\xb8\xad\xe5\x9b\xbd\xe4\xba\xba'

In [11]: unicode(s)                                 # 转换成 Unicode
u'\u4e2d\u56fd\u4eba'

In [12]: s.encode("UTF-16")                         # UTF-8 -> UTF-16
'\xff\xfe-N\xfdV\xbaN'

In [13]: s.encode("gb2312")                         # UTF-8 -> GB2312  熟悉吧，做网页的人都知道。
'\xd6\xd0\xb9\xfa\xc8\xcb'

In [14]: s.decode()                                 # UTF-8 -> Unicode
u'\u4e2d\u56fd\u4eba'
```

str、unicode 都提供了 encode 和 decode 方法。encode 用于将 "默认编码" 的字符串转换为其他编码，而 decode 则还原为 Unicode。如果为 decode 提供 encoding 参数，则表示从指定的编码转换回 unicode。

```
In [15]: s.encode("gb2312").decode("gb2312")        # UTF-8 -> GB2312 -> Unicode
u'\u4e2d\u56fd\u4eba'

In [16]: unicode(s.encode("gb2312"), "gb2312")      # GB2312 -> Unicode
u'\u4e2d\u56fd\u4eba'

In [17]: s.encode("gb2312").encode("utf-32")        # 为虾米不行呢？因为 encode("utf-32") 会把前一个结果
                                                    # 当作默认 utf-8 编码，但实际上是 gb2312。
ERROR: An unexpected error occurred while tokenizing input
```

```
The following traceback may be corrupted or invalid
The error message is: ('EOF in multi-line statement', (2, 0))
---------------------------------------------------------------------------
UnicodeDecodeError: 'utf8' codec can't decode byte 0xd6 in position 0: invalid continuation byte

In [18]: s.encode("gb2312").decode("gb2312").encode("utf-32") # 使用 decode 还原为 Unicode 就行了。
'\xff\xfe\x00\x00-N\x00\x00\xfdV\x00\x00\xbaN\x00\x00'
```

在 codecs 里，我们可以看到当前平台上 CPython 的 BOM 定义。

```
In [19]: import codecs

In [20]: codecs.BOM           # 按 <TAB> 显示所有 BOM 开头的成员。
codecs.BOM       codecs.BOM64_BE  codecs.BOM_LE        codecs.BOM_UTF16_LE  codecs.BOM_UTF32_LE
codecs.BOM32_BE  codecs.BOM64_LE  codecs.BOM_UTF16     codecs.BOM_UTF32     codecs.BOM_UTF8
codecs.BOM32_LE  codecs.BOM_BE    codecs.BOM_UTF16_BE  codecs.BOM_UTF32_BE

In [21]: codecs.BOM           # 我的 MackBook Pro 使用 intel CPU，显然是小端。
'\xff\xfe'

In [22]: codecs.BOM_UTF8
'\xef\xbb\xbf'

In [23]: codecs.BOM_UTF32
'\xff\xfe\x00\x00'

In [24]: codecs.BOM_UTF16     # 默认的，哈哈～～～
'\xff\xfe'
```

看看包含 BOM 头的操作演示。

```
In [25]: s = "中国人"

In [26]: s
'\xe4\xb8\xad\xe5\x9b\xbd\xe4\xba\xba'

In [27]: s.encode("utf-32")                                          # IPython 显示结果有点问题
'\xff\xfe\x00\x00-N\x00\x00\xfdV\x00\x00\xbaN\x00\x00'

In [28]: " ".join([hex(ord(c)) for c in s.encode("utf-32")])       # 这样就清楚多了
'0xff 0xfe 0x0 0x0 0x2d 0x4e 0x0 0x0 0xfd 0x56 0x0 0x0 0xba 0x4e 0x0 0x0'

In [29]: " ".join([hex(ord(c)) for c in s.encode("utf-16")])
'0xff 0xfe 0x2d 0x4e 0xfd 0x56 0xba 0x4e'
```

Unicode 好啊，要知道在 MS-DOS 时代，为了识别中英文混排，避免出现半个中文字符，可是费尽了心思。那时没有互联网，没有这么多的书籍，某些大牛也藏着掖着，怀揣软盘到处求教的菜鸟苦啊～～～

在 Linux、OSX 平台，CPython 默认采用 UCS-2 方案，也就是说用 2 个字节表示一个 Unicode 字符。

PC/pyconfig.h

```
#define Py_UNICODE_SIZE 2
```

看看 PyUnicodeObject 的具体实现。

Include/unicodeobject.h

```
typedef struct {
    PyObject_HEAD
    Py_ssize_t length;          // Unicode 字符数量 (不是字节)
    Py_UNICODE *str;            // 这回使用分离的两块内存了
    long hash;                  // 哈希值
    PyObject *defenc;           /* (Default) Encoded version as Python
                                   string, or NULL; this is used for
                                   implementing the buffer protocol */
} PyUnicodeObject;
```

Objects/unicodeobject.c

```
static PyUnicodeObject *free_list;
static int numfree;
static PyUnicodeObject *unicode_empty;

static PyUnicodeObject *_PyUnicode_New(Py_ssize_t length)
{
    register PyUnicodeObject *unicode;

    // unicode_empty 在 _PyUnicode_Init 被初始化为 _PyUnicode_New(0)
    if (length == 0 && unicode_empty != NULL) {
        Py_INCREF(unicode_empty);
        return unicode_empty;
    }

    ...

    // 据我们以往的经验来看，free_list 多半是个内存重复利用的链表。
    if (free_list) {
        // 从链表中取出一个位置。
        unicode = free_list;

        // 修改链表，剔除已取出来的那个位置。看这意思是直接将下一个链表元素的指针保存在这段内存中。
        free_list = *(PyUnicodeObject **)unicode;

        // 还有个计数器
        numfree--;

        // 查看取出来的 PyUncodeObject.str 是否可用。(链表下一个元素的指针只覆盖了 ob_refcnt)
        if (unicode->str) {
            // 检查 str 大小是否符合要求，如果不符合则进行调整。
            if ((unicode->length < length) && unicode_resize(unicode, length) < 0) {
                // 调整失败就置为 NULL
                PyObject_DEL(unicode->str);
                unicode->str = NULL;
            }
```

```
        }
        else {
            // 为 unicode->str 分配存储空间，记得为 "0x0000" 这个尾巴也留个位子。
            size_t new_size = sizeof(Py_UNICODE) * ((size_t)length + 1);
            unicode->str = (Py_UNICODE*) PyObject_MALLOC(new_size);
        }

        // 重新初始化头部信息
        PyObject_INIT(unicode, &PyUnicode_Type);
    }
    else {
        // 没有可重复使用的内存，只好从头开始了。
        ...

        // 为 PyUnicodeObject 分配内存。
        unicode = PyObject_New(PyUnicodeObject, &PyUnicode_Type);

        // 为 PyUnicodeObject.str 分配内存。
        new_size = sizeof(Py_UNICODE) * ((size_t)length + 1);
        unicode->str = (Py_UNICODE*) PyObject_MALLOC(new_size);
    }

    // 前面提到过，str 可能会因 resize 失败而为 NULL。
    if (!unicode->str) {
        PyErr_NoMemory();
        goto onError;
    }

    // 初始化相关信息
    unicode->str[0] = 0;
    unicode->str[length] = 0; // 同样以 \0 结尾。
    unicode->length = length;
    unicode->hash = -1;
    unicode->defenc = NULL;
    return unicode;

  onError:
    ... ...
}

PyObject *PyUnicode_FromStringAndSize(const char *u, Py_ssize_t size)
{
    if (u != NULL) {

        // unicode_empty 共享
        if (size == 0 && unicode_empty != NULL) {
            Py_INCREF(unicode_empty);
            return (PyObject *)unicode_empty;
        }

        // 单字符缓存
        if (size == 1 && Py_CHARMASK(*u) < 128) {
            unicode = unicode_latin1[Py_CHARMASK(*u)];
            if (!unicode) {
```

```
                unicode = _PyUnicode_New(1);
                unicode->str[0] = Py_CHARMASK(*u);
                unicode_latin1[Py_CHARMASK(*u)] = unicode;
            }
            Py_INCREF(unicode);
            return (PyObject *)unicode;
        }

        // 从 UTF8 解码
        return PyUnicode_DecodeUTF8(u, size, NULL);
    }

    ... ...
}

PyObject *PyUnicode_DecodeUTF8(const char *s, Py_ssize_t size, const char *errors)
{
    return PyUnicode_DecodeUTF8Stateful(s, size, errors, NULL);
}

PyObject *PyUnicode_DecodeUTF8Stateful(const char *s, Py_ssize_t size,
                                       const char *errors, Py_ssize_t *consumed)
{
    ... ...

    unicode = _PyUnicode_New(size);

    ... 巴拉巴拉一大堆 utf-8 解码过程 ...

    return (PyObject *)unicode;
}
```

除了这个 free_list 来得莫名奇妙外 (没看到创建 free_list 的过程)，似乎没什么不妥。我们在 _PyUnicode_Init 中并没有找到任何有价值的线索，只好去看看 unicode_dealloc 了。

```
#define PyUnicode_MAXFREELIST       1024
#define KEEPALIVE_SIZE_LIMIT        9

static void unicode_dealloc(register PyUnicodeObject *unicode)
{
    // 看样子有戏，这个 numfree 不就是 free_list 的计数器吗。
    // 显然，free_list 有数量限制。
    if (PyUnicode_CheckExact(unicode) && numfree < PyUnicode_MAXFREELIST) {

        // free_list 只接纳 unicode->length < 9 的对象?
        if (unicode->length >= KEEPALIVE_SIZE_LIMIT) {

            // 看样子是误会了，如果 unicode->length 长度大于 9，就释放掉 str 所占用的内存。
            //   PyUnicodeObject 依然保留的。难怪从 free_list 取出缓存对象后，还要重新分配 str 内存。
            PyObject_DEL(unicode->str);
            unicode->str = NULL;
            unicode->length = 0;
        }
```

```
        ...

        // 将原来的链表地址保存到当前 PyUnicodeObject 内存中。
        *(PyUnicodeObject **)unicode = free_list;

        // 将当前这个被 "释放" 的内存块添加到链表头部，并调整计数器。
        free_list = unicode;
        numfree++;
    }
    else {
        // 如果超出 free_list 数量限制，就直接将对象释放，回收内存。
        PyObject_DEL(unicode->str);
        Py_XDECREF(unicode->defenc);
        Py_TYPE(unicode)->tp_free((PyObject *)unicode);
    }
}
```

使用一个 free_list 链表来管理最多 1024 个可重复使用的对象缓存，且只有字符数量小于 9 的缓存对象会保留其存储字符内容的那块内存。选择 9 这个数字，想必是在缓存和避免内存浪费之间找平衡。

当进程世界 Game Over 时，_PyUnicode_Fini 会调用 PyUnicode_ClearFreeList 回收这些缓存。

```
int PyUnicode_ClearFreeList(void)
{
    int freelist_size = numfree;
    PyUnicodeObject *u;

    // 循环 free_list 链表
    for (u = free_list; u != NULL;) {
        PyUnicodeObject *v = u;
        u = *(PyUnicodeObject **)u;

        // 释放 str，注意长度大于等于 9 的已经释放过了。
        if (v->str) PyObject_DEL(v->str);
        Py_XDECREF(v->defenc);

        // 释放 PyUnicodeObject。
        PyObject_Del(v);
        numfree--;
    }

    free_list = NULL;
    assert(numfree == 0);
    return freelist_size;
}
```

unicode 和 str 一样，也是不可变对象 (immutable object)，需要注意的性能问题一个不少。

2.3 列表

为什么 Python 用 list 取代 array 在内置类型中的位置?
为什么有了 list 还要有 tuple? 两者除了只读，还有什么区别?
你确定能正确使用 list，而不会引发性能问题吗?
其实我们不该忘记被驱逐到标准库里的 array。

2.3.1 list

我个人觉得，Python 里面除了 int，就属 list 的使用频率最高了。这个看上去精巧的数据类型，既不是数组，也不是链表，或许应该用 Vector<void*> 来表达更准确一些。

PyListObject 采取预分配内存策略，也就是说通常会分配比实际使用更多的内存。ob_item 是用来保存元素的指针数组，只在元素数量大于 0 时才会分配所需内存。ob_size 记录了当前正在使用的元素数量，而 allocated 则是 ob_item 预分配的总容量。

Include/listobject.h

```
typedef struct {
    PyObject_VAR_HEAD      // ob_size 已使用数量
    PyObject **ob_item;    // 保存元素的指针数组
    Py_ssize_t allocated;  // ob_item 预分配数量
} PyListObject;
```

在创建函数 PyList_New 里，一贯用作对象缓存管理器的 free_list 再次出现。不过这回不是链表，而是一个指针数组。同时我们会看到 ob_item 是新分配的，并没有出现在缓存里。

Objects/listobject.c

```
PyObject * PyList_New(Py_ssize_t size)
{
    ... ...

    nbytes = size * sizeof(PyObject *);

    // 对象缓存 numfree 和 free_list 无所不在。
    if (numfree) {
        numfree--;

        // 从 free_list 数组取出一个 PyListObject 对象。
        //   显然通过 numfree 这个数组下标来确定最后一个可用位置。
        op = free_list[numfree];

        // 重新调整引用计数等头部信息。
        _Py_NewReference((PyObject *)op);
        ...
    } else {
        // 没有多余的缓存对象，那只好创建一个了。
        op = PyObject_GC_New(PyListObject, &PyList_Type);
```

```
        ...
    }

    if (size <= 0)
        // 空 list，无需分配元素指针数组。
        op->ob_item = NULL;
    else {
        // 申请元素指针数组所需内存。
        op->ob_item = (PyObject **) PyMem_MALLOC(nbytes);

        // 全部初始化为 0，因为这里面保存的是元素对象的指针。
        memset(op->ob_item, 0, nbytes);
    }

    // 调整相关信息
    Py_SIZE(op) = size;
    op->allocated = size;

    // 设置 PyGC_HEAD 信息，进入 GC 跟踪黑名单。
    _PyObject_GC_TRACK(op);

    return (PyObject *) op;
}
```

对象缓存行为依旧发生在 dealloc 函数中。最多有 80 个 PyListObject 对象被 free_list 数组缓存起来，不过它们的 ob_item 指针数组将被释放掉。PyListObject 使用 PyObject_GC_New 在 arena 上分配，而 ob_item 则是直接使用 PyMem_MALLOC 从堆上申请内存。

Objects/listobject.c

```
#define PyList_MAXFREELIST 80
static PyListObject *free_list[PyList_MAXFREELIST];
static int numfree = 0;

static void list_dealloc(PyListObject *op)
{
    // 准备回收，从黑名单中剔除。
    PyObject_GC_UnTrack(op);
    ...

    if (op->ob_item != NULL) {
        // 减少所有元素的引用计数，以便它们可以被回收。
        i = Py_SIZE(op);
        while (--i >= 0) {
            Py_XDECREF(op->ob_item[i]);
        }

        // 释放元素指针数组内存
        PyMem_FREE(op->ob_item);
    }

    if (numfree < PyList_MAXFREELIST && PyList_CheckExact(op))
        // 如果缓存数组中还有空位，就将 PyListObject 加到末尾。
```

```
        free_list[numfree++] = op;
    else
        // 没有位子，就只好拉出去砍了。
        Py_TYPE(op)->tp_free((PyObject *)op);

    ...
}
```

2.3.1.1 resize

list 和 array 最大的外在区别是，我们随时可以插入、添加和移除元素。PyListObject 会根据需要自动扩容，无须担心越界访问。

从下面的代码里，我们能看到 Python 设计者的苦心。在需要的时候扩容，但又不允许过度的浪费，适当的内存回收是非常必要的。

Objects/listobject.c

```
int PyList_Append(PyObject *op, PyObject *newitem)
{
    if (PyList_Check(op) && (newitem != NULL))
        return app1((PyListObject *)op, newitem);

    ...
}

static int app1(PyListObject *self, PyObject *v)
{
    ... ...

    // 检查可用空间，如果有必要，list_resize 会进行扩容。
    if (list_resize(self, n+1) == -1)
        return -1;

    Py_INCREF(v);
    PyList_SET_ITEM(self, n, v);
    return 0;
}

static int list_resize(PyListObject *self, Py_ssize_t newsize)
{
    PyObject **items;
    size_t new_allocated;
    Py_ssize_t allocated = self->allocated;

    // 如果还有可用空间，且使用率超过一半，那么直接修改 ob_size 就行了。
    //   这都用了一半了，咋还不扩容呢？汗～～
    //   真正的意图是告诉后边，如果利用率过低，就用 realloc 收缩 (shrink) 内存。哈哈～～～
    if (allocated >= newsize && newsize >= (allocated >> 1)) {
        assert(self->ob_item != NULL || newsize == 0);
        Py_SIZE(self) = newsize;
        return 0;
```

```
    }

    /* 这个确定调整后的空间大小算法很有意思。
     *
     *   调整后大小 (new_allocated) = 新元素数量 (newsize) + 预留空间 (new_allocated)
     *
     *   调整后的空间肯定能存储 newsize 个元素。要关注的是预留空间的增长状况。
     *   将预留算法改成 Python 版就更清楚了：(newsize // 8) + (newsize < 9 and 3 or 6)。
     *
     *   当 newsize >= allocated，自然按照这个新的长度 "扩容" 内存。
     *   而如果 newsize < allocated，且利用率低于一半呢？
     *
     *   allocated    newsize        new_size + new_allocated
     *   10           4              4 + 3
     *   20           9              9 + 7
     *
     *   很显然，这个新长度小于原来的已分配空间长度，自然会导致 realloc 收缩内存。(不容易啊)
     */
    new_allocated = (newsize >> 3) + (newsize < 9 ? 3 : 6);
    if (new_allocated > PY_SIZE_MAX - newsize) {
        PyErr_NoMemory();
        return -1;
    } else {
        new_allocated += newsize;
    }

    if (newsize == 0) new_allocated = 0;
    items = self->ob_item;

    // 检查 new_allocated 没有超过内存最大限制。
    if (new_allocated <= ((~(size_t)0) / sizeof(PyObject *)))
        // 通过 PyMem_REALLOC 调整内存，可能增大，也可能减少。
        PyMem_RESIZE(items, PyObject *, new_allocated);
    else
        items = NULL;

    ...

    // 调整相关参数。
    self->ob_item = items;
    Py_SIZE(self) = newsize;
    self->allocated = new_allocated;

    return 0;
}
```

2.3.1.2 realloc

熟悉 C realloc 函数的兄弟肯定会发现 list_resize 函数潜在的性能问题。当因为追加元素导致的扩容行为发生时，realloc 会尝试在当前内存后面连续分配新增的内存空间。但如果失败，就必须另

址建屋，还得将原有的内容拷贝过去。尽管 ob_item 保存的只是一些指针，但如果是个巨大的 list 就有可能引发性能问题。更何况，在添加元素循环的过程中，是有可能发生多次内容拷贝事件的。

如何避免类似的问题呢？可以预先申请足够大的地方，然后用 set_item 方式赋值，避免多次内存分配调整。

```
In [1]: import itertools, dis

In [2]: def test(size):
   ...:     return list(itertools.repeat(0, size))
   ...:

In [3]: a = test(10)

In [4]: a
Out[4]: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

In [5]: dis.dis(test)
  2           0 LOAD_GLOBAL              0 (list)
              3 LOAD_GLOBAL              1 (itertools)
              6 LOAD_ATTR                2 (repeat)
              9 LOAD_CONST               1 (0)
             12 LOAD_FAST                0 (size)
             15 CALL_FUNCTION            2
             18 CALL_FUNCTION            1
             21 RETURN_VALUE
```

上面的例子使用了不太常见的做法，那是因为你所习惯的方式与本意相差甚远。

```
In [6]: def test2(size):
   ...:     return [0 for x in xrange(size)]
   ...:

In [7]: dis.dis(test2)
  2           0 BUILD_LIST               0                 # list 对象已经被创建。
              3 LOAD_GLOBAL              0 (xrange)
              6 LOAD_FAST                0 (size)
              9 CALL_FUNCTION            1
             12 GET_ITER
        >>   13 FOR_ITER                12 (to 28)
             16 STORE_FAST               1 (x)
             19 LOAD_CONST               1 (0)
             22 LIST_APPEND              2                 # 看到 append，估计你也明白了。
             25 JUMP_ABSOLUTE           13
        >>   28 RETURN_VALUE
```

我们修改一下函数返回值 (避免刷屏)，然后测试一下两者的性能。

```
In [8]: def test(size):
   ....:     l = list(itertools.repeat(0, size))
   ....:     return len(l)
   ....:
```

```
In [9]: def test2(size):
   ....:     l = [0 for x in xrange(size)]
   ....:     return len(l)
   ....:

In [10]: %timeit -r 10 test(10000000)
1 loops, best of 10: 196 ms per loop

In [11]: %timeit -r 10 test2(10000000)
1 loops, best of 10: 736 ms per loop
```

两者的性能差异还是非常明显的。当然，你也可以关掉 gc 尝试进一步优化。不过提升很有限，因为除了 list，我们只是重复使用同一个小整数对象 0，没什么值得 gc 跟踪的。

考虑到 list 完全按照 Vector 的方式实现，它不适合用作为一个需要频繁插入和删除的超大列表，或许你需要实现一个链表 (LinkList)。

2.3.1.3 bisect

list 提供了一个 sort 排序方法，那么如何在保证排序的情况下快速插入元素呢？可借助 bisect 二分法查找适合的插入位置。

```
In [12]: import random, bisect

In [13]: l = random.sample(xrange(100), 10); l
Out[13]: [84, 60, 8, 90, 71, 3, 61, 54, 37, 43]

In [14]: l.sort(); l
Out[14]: [3, 8, 37, 43, 54, 60, 61, 71, 84, 90]

In [15]: bisect.insort_left(l, 52); l
Out[15]: [3, 8, 37, 43, 52, 54, 60, 61, 71, 84, 90]

In [16]: bisect.insort_left(l, 91); l
Out[16]: [3, 8, 37, 43, 52, 54, 60, 61, 71, 84, 90, 91]
```

如果是自定义类型，则需要重载比较运算符。

```
In [17]: class A(object):
   ....:     def __init__(self, x):
   ....:         self.x = x
   ....:     def __repr__(self):
   ....:         return "{0:X} {1}".format(id(self), self.x)
   ....:     def __cmp__(self, o):
   ....:         if o is None or not isinstance(o, A):
   ....:             raise Exception()
   ....:         return cmp(self.x, o.x)
   ....:

In [18]: l = [A(3), A(1), A(4), A(2)]; l
```

```
Out[18]: [11080CE90 3, 11080CED0 1, 11080CF10 4, 11080CF50 2]

In [19]: l.sort(); l
Out[19]: [11080CED0 1, 11080CF50 2, 11080CE90 3, 11080CF10 4]

In [20]: bisect.insort_left(l, A(2)); l
Out[20]: [11080CED0 1, 11080CF90 2, 11080CF50 2, 11080CE90 3, 11080CF10 4]
```

用 Python 写过一致性哈希算法 (consistent hashing) 的兄弟，想必对 bisect 是很熟悉的。

2.3.2 tuple

tuple 就像 list 的影子，除了不能修改，多数时候都是可以替换的。tuple 是不可变的，因此可以像 str 那样使用一块连续内存。

Include/tupleobject.h

```
typedef struct {
    PyObject_VAR_HEAD
    PyObject *ob_item[1];
} PyTupleObject;
```

Objects/tupleobject.c

```
#define PyTuple_MAXSAVESIZE     20   /* Largest tuple to save on free list */
#define PyTuple_MAXFREELIST  2000    /* Maximum number of tuples of each size to save */

static PyTupleObject *free_list[PyTuple_MAXSAVESIZE];
static int numfree[PyTuple_MAXSAVESIZE];

PyObject * PyTuple_New(register Py_ssize_t size)
{
    ... ...

    // 难道 free_list[0] 存储了一个默认的空 PyTupleObject?
    if (size == 0 && free_list[0]) {
        op = free_list[0];
        ...
        return (PyObject *) op;
    }

    // op = free_list[size] ? numfree[size]-- ?
    // 似乎有不寻常的意味，和我们前面见过的对象缓存方式不大一样。
    //   numfree 是个计数器数组，按照 size 划分出不同的计数器。
    //   而 free_list[size] 是否意味着该数组每个元素都是一个链表，保存了 numfree[size] 个缓存的对象。
    //   这个设计看上去像链表法哈希表。
    if (size < PyTuple_MAXSAVESIZE && (op = free_list[size]) != NULL) {

        // 果然如此，使用了 ob_item[0] 作为链表指针。
        // 将 op 从链表中移除，并减小对应的计数器。
        free_list[size] = (PyTupleObject *) op->ob_item[0];
        numfree[size]--;
```

```
        ... ...

        // 初始化 op 的相关信息
        Py_SIZE(op) = size;
        Py_TYPE(op) = &PyTuple_Type;
        _Py_NewReference((PyObject *)op);
    }
    else
    {
        // 没有缓存可用，则直接创建新的对象。注意包括了指针数组内存。
        Py_ssize_t nbytes = size * sizeof(PyObject *);
        op = PyObject_GC_NewVar(PyTupleObject, &PyTuple_Type, size);

        ... ...
    }

    // 初始化 ob_items。
    for (i=0; i < size; i++)
        op->ob_item[i] = NULL;

    // 看样子 free_list[0] 只保存了一个唯一的空 PyTupleObject 对象实例进行共享。
    //   倒也正常，空 tuple 没有元素，又是只读的，共享一个对象足矣。
    if (size == 0) {
        free_list[0] = op;
        ++numfree[0];
        Py_INCREF(op);          /* extra INCREF so that this is never freed */
    }

    _PyObject_GC_TRACK(op);
    return (PyObject *) op;
}
```

PyTupleObject 在对象缓存上使用完全不同于我们前面所见的方式。size < 20 的对象可以被缓存，并且 size 相同的对象通过 ob_item[0] 构成一个链表，保存在 free_list[size] 这样一个对应位置。取用缓存对象时，只需以 size 作下标即可非常方便地从 free_list 中找到合适大小的缓存链表，然后从链表投标返回对象即可。每个链表最多可以保存 2000 个缓存对象。

tuple 不可变指的是 Python 语言层面，这和底层 PyTupleObject 对象复用不是一个概念，并不冲突。脑经千万要转过弯来。

我们继续看看 dealloc，看看对象如何被放入缓存链表中。(函数命名规则有变化？难道不是同一个人写的)

```
static void tupledealloc(register PyTupleObject *op)
{
    register Py_ssize_t len =  Py_SIZE(op);

    ... ...

    // 从 GC 黑名单划掉名字，准备开刀问斩。
    PyObject_GC_UnTrack(op);
```

```
    // 小于 0 的空对象就一个，肯定不需要释放的。
    if (len > 0) {
        i = len;

        // 减小所有元素计数器，使其可以被回收。
        while (--i >= 0)
            Py_XDECREF(op->ob_item[i]);

        // 如果长度小于 20，且对应的计数器没有超出 2000 个数量限制，哇哈哈～～～～
        if (len < PyTuple_MAXSAVESIZE && numfree[len] < PyTuple_MAXFREELIST &&
            Py_TYPE(op) == &PyTuple_Type)
        {
            // 将当前这个对象的 ob_item[0] 指向原来的链表。
            op->ob_item[0] = (PyObject *) free_list[len];

            // 计数器自然要增长。
            numfree[len]++;

            // 当前对象顺利成为 "匪头"，坐到 free_list[size] 宝座，管理一堆同样体重的 "瘦子" (小于20，能胖吗)。
            free_list[len] = op;

            // 被缓存了，自然不能 free 了。
            goto done; /* return */
        }
    }

    // 超胖，或者没有空位，就要被咔嚓掉。
    Py_TYPE(op)->tp_free((PyObject *)op);

done:
    ...
}
```

发现很多 Pythoner 都有个 "坏习惯"，总是下意识用 list，尤其是初学者更甚。我个人觉得应该尽可能使用 tuple 而不是 list。首先，tuple 是不可变对象，作为函数参数时，无需关心同步问题，天生并发安全。当我们使用 yield 设计协程，或者多线程并发时，函数可重入是很重要的问题。其次，tuple 的内存布局和缓存方式都比 list 更高效，就算是被缓存的 list，其 items 也面临二次内存分配的情况，远不如 tuple 的等长缓存链表轻便。绝大多数时候，我们所使用的列表都很小，也不总是在创建后还需要修改，那么为啥不考虑 tuple 一下呢？

现在比较流行的函数式编程，同样将不可变对象作为基石之一。

2.3.2.1 逗号少不得

在创建只有一个元素的 tuple 时，千万别忘了加个 "," ，否则就出乱子了。

```
In [1]: t1 = ("a"); t2 = ("a",); t3 = ()

In [2]: type(t1), type(t2), type(t3)
```

```
Out[2]: (str, tuple, tuple)
```

2.3.2.2 namedtuple

在标准库中还有一个 tuple 的弟子 namedtuple，我们早在第一章 "动态编译" 一节就有接触。实现虽然简单，但用来做数据类是极方便的。

通过动态添加属性来创建数据对象一样方便，何必用 namedtuple?

理由有二：首先，namedtuple 动态创建类型，而非实例对象，我们可以判断多个数据对象是否属于同一个类型，动态属性 "做不到"。其次，namedtuple 内部已经包含了 __slots__，可以避免意外增加或删除属性。(__slots__ 详见后续章节)

```
In [3]: import collections

In [4]: User = collections.namedtuple("User", "name age sex", verbose = True)
class User(tuple):
        'User(name, age, sex)'

        __slots__ = ()

        _fields = ('name', 'age', 'sex')

        def __new__(_cls, name, age, sex):
            'Create new instance of User(name, age, sex)'
            return _tuple.__new__(_cls, (name, age, sex))

        @classmethod
        def _make(cls, iterable, new=tuple.__new__, len=len):
            'Make a new User object from a sequence or iterable'
            result = new(cls, iterable)
            if len(result) != 3:
                raise TypeError('Expected 3 arguments, got %d' % len(result))
            return result

        def __repr__(self):
            'Return a nicely formatted representation string'
            return 'User(name=%r, age=%r, sex=%r)' % self

        def _asdict(self):
            'Return a new OrderedDict which maps field names to their values'
            return OrderedDict(zip(self._fields, self))

        def _replace(_self, **kwds):
            'Return a new User object replacing specified fields with new values'
            result = _self._make(map(kwds.pop, ('name', 'age', 'sex'), _self))
            if kwds:
                raise ValueError('Got unexpected field names: %r' % kwds.keys())
            return result

        def __getnewargs__(self):
```

```
            'Return self as a plain tuple.  Used by copy and pickle.'
            return tuple(self)

        name = _property(_itemgetter(0), doc='Alias for field number 0')
        age = _property(_itemgetter(1), doc='Alias for field number 1')
        sex = _property(_itemgetter(2), doc='Alias for field number 2')


In [5]: a, b = User("User1", 10, 1), User("User2", 20, 0)

In [6]: type(a), type(b)
Out[6]: (__main__.User, __main__.User)

In [7]: a.phone = "12345678"
---------------------------------------------------------------------------
AttributeError: 'User' object has no attribute 'phone'

In [8]: del a.name
---------------------------------------------------------------------------
AttributeError: can't delete attribute
```

可问题是 namedtuple 是只读的，想要修改咋办?

```
In [9]: a
Out[9]: User(name='User1', age=10, sex=1)

In [10]: a = a._replace(age = 11)           # 创建了一个新的对象

In [11]: a
Out[11]: User(name='User1', age=11, sex=1)
```

或许你会觉得 _replace 很麻烦，还不如创建一个自定义类型。诚然，在参考 namedtuple 模版的基础上写一个可读写数据类很简单。但我觉得在处处按引用传递的 Python 中用 "只读对象" 实现数据 "Copy-On-Write" 是很有必要的，关键还是用在合适的场合。

2.3.3 array

不知道 Python 是不是第一种将 array 打入冷宫的编程语言。

"攻城师" 多半是 "喜新厌旧" 的，当 Pythoner 拥着 list "纵情声色" 时，哪里还记得 "别在长门宫，愁闷悲思" 的昔日皇后陈阿娇。其实 Python array 非但没有人老珠黄，反而脱胎换骨，别有一番风流体韵。细看下面的例子，对比前后的内存占用，是否有些惊讶?

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-

import sys
import os
import array
import psutil
```

```
def mem():
    rss, vms = psutil.Process(os.getpid()).get_memory_info()
    print "RSS:{0}, VMS:{1}".format(rss / 1024 / 1024, vms / 1024 / 1024)

SIZE = 10000000

def test1():
    l = list(xrange(SIZE))
    return "test1", len(l), l[SIZE - 1]

def test2():
    l = array.array("l", xrange(SIZE))
    return "test2", len(l), l[SIZE - 1]

def main():
    mem()
    print len(sys.argv) == 1 and test1() or test2()
    mem()

if __name__ == "__main__":
    main()
```

输出:

```
$ ./main.py
RSS:8, VMS:2385
('test1', 10000000, 9999999)
RSS:318, VMS:2705

$ ./main.py 2
RSS:8, VMS:2385
('test2', 10000000, 9999999)
RSS:84, VMS:2463
```

list 保存的是什么？是对象指针。而每个整数对象又有 PyObject_HEAD，原本只需 8 字节就能存储的整数 (long)，一下子 "虚胖" 了 3 倍，整整 24 字节。而 array 呢？int[n] 里面保存的显然是数值本身，8 字节还是 8 字节。粗略算来，list 保存一个 64 位整数的开销是 array 的 4 倍。

```
In [1]: import array

In [2]: ints = array.array("l", xrange(10))

In [3]: ints
Out[3]: array('l', [0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [4]: hex(id(ints))
Out[4]: '0x10e6b4570'
```

```
(gdb) x/4xg 0x10e6b4570
0x10e6b4570:   0x0000000000000005    0x000000010e4594b0
0x10e6b4580:   0x000000000000000a    0x00007fc2d3ea8270
```

```
(gdb) x/10xg 0x00007fc2d3ea8270
0x7fc2d3ea8270:       0x0000000000000000    0x0000000000000001
0x7fc2d3ea8280:       0x0000000000000002    0x0000000000000003
0x7fc2d3ea8290:       0x0000000000000004    0x0000000000000005
0x7fc2d3ea82a0:       0x0000000000000006    0x0000000000000007
0x7fc2d3ea82b0:       0x0000000000000008    0x0000000000000009
```

array 可存储 char、short、long、float 等多种数字类型，因此需要专门记录元素的相关信息。为此引入了 arraydescr 元数据类型，其中 typecode 记录元素类型，itemsize 记录元素类型长度，同时还包含类型专用的读写函数指针。

Modules/arraymodule.c

```
struct arraydescr {
    int typecode;
    int itemsize;
    PyObject * (*getitem)(struct arrayobject *, Py_ssize_t);
    int (*setitem)(struct arrayobject *, Py_ssize_t, PyObject *);
};

typedef struct arrayobject {
    PyObject_VAR_HEAD
    char *ob_item;                    // 数组元素实际存储内存
    Py_ssize_t allocated;             // 预分配内存大小
    struct arraydescr *ob_descr;      // 元素不再是对象，必须有元数据对其描述，包括大小和读写函数。
    PyObject *weakreflist;
} arrayobject;

static PyObject * newarrayobject(PyTypeObject *type, Py_ssize_t size, struct arraydescr *descr)
{
    ... ...

    // 计算实际需要的内存大小 (byte)
    nbytes = size * descr->itemsize;

    ...

    // 创建 array 对象，并设置相关属性。最重要的是 descr。
    op = (arrayobject *) type->tp_alloc(type, 0);
    op->ob_descr = descr;
    op->allocated = size;
    op->weakreflist = NULL;
    Py_SIZE(op) = size;

    if (size <= 0) {
        op->ob_item = NULL;
    }
    else {
        // 分配元素实际存储内存。
        op->ob_item = PyMem_NEW(char, nbytes);
        ...
    }
    return (PyObject *) op;
```

```
}
```

除没有对象缓存外，整体实现和 list 非常类似。实际上 array 和我们以往熟悉的数组也有很大不同，我觉得大可改名为 Vector，如此才能名正言顺地待在标准库的 "Data Types" 中。作为经典的数据结构，根本无需与 list 争宠，也没必要顶着 array 的名头被人 "莫名其妙" 地冷落。array 具备 list 的全部功能接口，包括 insert、extend、pop 等。一样会在需要的时候自动扩容， 还可以非常方便地与常用类型进行转换。

```
In [1]: import array

In [2]: array.array.
array.array.append       array.array.fromunicode  array.array.reverse
array.array.buffer_info  array.array.index        array.array.tofile
array.array.byteswap     array.array.insert       array.array.tolist
array.array.count        array.array.itemsize     array.array.tostring
array.array.extend       array.array.mro          array.array.tounicode
array.array.fromfile     array.array.pop          array.array.typecode
array.array.fromlist     array.array.read         array.array.write
array.array.fromstring   array.array.remove
```

看看 array 如何读写 PyIntObject，是不是有 C#、Java "box/unbox" 的感触。

```
static PyObject * l_getitem(arrayobject *ap, Py_ssize_t i)
{
    // long -> PyIntObject
    return PyInt_FromLong(((long *)ap->ob_item)[i]);
}

static int l_setitem(arrayobject *ap, Py_ssize_t i, PyObject *v)
{
    long x;

    // PyIntObject -> long
    if (!PyArg_Parse(v, "l;array item must be integer", &x))
        return -1;

    if (i >= 0)
        ((long *)ap->ob_item)[i] = x;

    return 0;
}
```

不过也正因为多了这道转换过程 (还得把 PyIntObject 临时对象回收掉)，所以 array 比直接使用对象指针的 list 性能要稍差一些。array 和 list 各有优势，因为在存储方式上的差别，我们当按需选用，没必要强用 array，也不能 list 一路走到黑。如果要实现自己的数据结构，array 应该有资格成为候选底层结构。

2.4 字典

将 dict 拿掉，会有什么后果？CPython 2012! 因为其最核心的名字空间理念依赖于 dict 实现。

2.4.1 dict

说到 dict，有两个比较容易混淆的名字：map 和 hashtable。map 通常采用平衡二叉树实现，而 hashtable 则是用数组实现的哈希表。前者具备更好的插入性能，后者则强调搜索效率。Python dict 是一种基于开放地址法的哈希表。(有关哈希表的具体实现，请参考相关专业数据结构书籍)

首先要分配一段合适大小的数组，每个数组元素都将被初始化为 PyDictEntry。

Include/dictobject.h

```
#define PyDict_MINSIZE 8

typedef struct {
    Py_ssize_t me_hash;      // me_key hashcode 缓存，避免重复计算，空间换时间。
    PyObject *me_key;
    PyObject *me_value;
} PyDictEntry;

/*
To ensure the lookup algorithm terminates, there must be at least one Unused
slot (NULL key) in the table.

The value ma_fill is the number of non-NULL keys (sum of Active and Dummy);
ma_used is the number of non-NULL, non-dummy keys (== the number of non-NULL
values == the number of Active items).

To avoid slowing down lookups on a near-full table, we resize the table when
it's two-thirds full.
*/
typedef struct _dictobject PyDictObject;
struct _dictobject {
    PyObject_HEAD
    Py_ssize_t ma_fill;                             // Active + Dummy Entry 数量
    Py_ssize_t ma_used;                             // Active Entry 数量
    Py_ssize_t ma_mask;                             // 线性探测所需的位置掩码，不等于 Entry 总数量。
    PyDictEntry *ma_table;                          // 默认指向 ma_smalltable，需要时指向另外分配的内存。
    PyDictEntry *(*ma_lookup)(...);                 // 线性探测函数指针。
    PyDictEntry ma_smalltable[PyDict_MINSIZE];      // 自带的容量为 8 的 entry 数组。
};
```

从 PyDictObject 源码的相关注释中，我们可以看到用来保存 key-value 的 Entry 分为 Active、Dummy 和 Unused 三种状态。Active 表示当前 Entry 保存了一个有效的 key-value 对象，而 Unused 自然是 "空闲" 位置。那么 Dummy 是做什么的呢?

开放地址法如何确定一个有效的存储位置？首先通过特定算法计算出有效的存储位置 (数组序号)，如果该位置已被占用，则继续计算下一个位置，直到某个空闲位置出现。查找时采取相同路线，要么在多次线性探测后找到目标，要么直到空位查无此人。

假设，某个key-value 对象在经过 [10]、[23]、[30] 三次线性探测后，终于在 [45] 这个位置安家落户。后来 [30] 号人家搬迁，此处位置变成空闲。当我们查找 [45] 所在的对象时，麻烦出现了，因为在经过 [10]、[20] 两次比对失败后，[30] 的空位表明查找可以结束。如何避免这种糟糕的状况呢？就是将 [30] 号空位设置为一种特殊的空闲状态，告诉查找算法，继续查找后面的位置。同时这个特殊状态又可以作为有效的存储位置。嗯，这个特殊的状态就是 Dummy。

ma_mask 值用于线性探测，并不等于 Entry 总数，实际上 ma_table 会分配 "ma_mask + 1" 个 Entry 存储空间。在线性探测函数 ma_lookup 中使用 "index = hashcode & ma_mask" 公式来计算有效存储位置 (ma_table 数组序号)，那么有可能的结果就是 "0 ~ ma_mask"，所以实际容量必然是 ma_mask + 1。

ma_table 默认指向自带的 ma_smalltable。多数时候，我们所使用的 dict 都很小，直接使用 smalltable 可减少二次内存操作，提升整体性能。

Objects/dictobject.c

```
#define PyDict_MINSIZE 8
#define PyDict_MAXFREELIST 80
static PyDictObject *free_list[PyDict_MAXFREELIST];
static int numfree = 0;

#define INIT_NONZERO_DICT_SLOTS(mp) do {                                \
    (mp)->ma_table = (mp)->ma_smalltable;                               \
    (mp)->ma_mask = PyDict_MINSIZE - 1;                                 \
    } while(0)

#define EMPTY_TO_MINSIZE(mp) do {                                       \
    memset((mp)->ma_smalltable, 0, sizeof((mp)->ma_smalltable));        \
    (mp)->ma_used = (mp)->ma_fill = 0;                                  \
    INIT_NONZERO_DICT_SLOTS(mp);                                        \
    } while(0)

PyObject * PyDict_New(void)
{
    register PyDictObject *mp;

    // 原来 Dummy 是一个 key 为 "<dummy key>" 的专用 Entry 对象。
    if (dummy == NULL) { /* Auto-initialize dummy */
        dummy = PyString_FromString("<dummy key>");
        ... ...
    }

    // 想让我说 "咋又是 numfree + free_list 这对缓存老搭档……"，我偏不说。
    if (numfree) {

        // 用数组缓存，通过减小 numfree 计数器来调整下次获取缓存对象的位置。
        mp = free_list[--numfree];

        // 重新初始化缓存对象的相关信息。
        _Py_NewReference((PyObject *)mp);
```

```
        if (mp->ma_fill) {
            EMPTY_TO_MINSIZE(mp);
        } else {
            INIT_NONZERO_DICT_SLOTS(mp);
        }

        ... ...

    } else {

        // 创建新对象
        mp = PyObject_GC_New(PyDictObject, &PyDict_Type);

        // 初始化相关信息
        EMPTY_TO_MINSIZE(mp);

        ...
    }

    // 默认使用 lookdict_string 探测函数。
    mp->ma_lookup = lookdict_string;

    ... ...

    return (PyObject *)mp;
}

static void dict_dealloc(register PyDictObject *mp)
{
    register PyDictEntry *ep;
    Py_ssize_t fill = mp->ma_fill;

    // 从 GC 黑名单删除
    PyObject_GC_UnTrack(mp);

    ...

    // 遍历 ma_table，释放掉所有 Active、Dummy Entry。
    for (ep = mp->ma_table; fill > 0; ep++) {
        if (ep->me_key) {
            --fill;
            Py_DECREF(ep->me_key);
            Py_XDECREF(ep->me_value);
        }
    }

    // 如果 ma_table 指向额外申请的内存，则释放掉。
    if (mp->ma_table != mp->ma_smalltable)
        PyMem_DEL(mp->ma_table);

    // 最多可以存储 80 个缓存对象。
    if (numfree < PyDict_MAXFREELIST && Py_TYPE(mp) == &PyDict_Type)
        free_list[numfree++] = mp;
    else
```

```
        // 缓存数组装不下就拉出去砍了吧，留着心烦。
        Py_TYPE(mp)->tp_free((PyObject *)mp);

    ...
}
```

从本章开始的 int 一路折腾到现在，我不得不感叹做这类 "底层" 所需要的细腻心思。别说写了，我看着都觉得累，光缓存方法就有三四种了吧?

2.4.1.1 lookup

在存储 key-value 对象时，需要通过探测函数找到第一个状态为 Unused 或 Dummy 的可用位置。而在查找时，依然需要它进行多次线性探测才能找到真正的目标对象。可见，线性探测函数是 dict 中最重要的核心算法。

鉴于大多数情况下，都会使用字符串作为 key，因此 CPython 专门提供跟了一个 lookdict_string 版本。它和 lookdict 的算法原理完全相同，只是少了一些错误处理。我们就用它来分析具体的探测过程。

Objects/dictobject.c

```
#define PERTURB_SHIFT 5

static PyDictEntry * lookdict_string(PyDictObject *mp, PyObject *key, register long hash)
{
    register size_t i;
    register size_t perturb;
    register PyDictEntry *freeslot;
    register size_t mask = (size_t)mp->ma_mask;
    PyDictEntry *ep0 = mp->ma_table;
    register PyDictEntry *ep;

    // 如果 key 不是字符串，就使用通用版本 lookdict。
    if (!PyString_CheckExact(key)) {
        mp->ma_lookup = lookdict;
        return lookdict(mp, key, hash);
    }

    // 探测第一个存储位置
    i = hash & mask;
    ep = &ep0[i];

    // 如果是 Unused 表示没找到，这个位置可用来存储。
    // 如果 key 指向同一个对象，自然返回这个 entry。
    if (ep->me_key == NULL || ep->me_key == key)
        return ep;

    // 如果是 Dummy，则用 freeslot 记下这个第一存储位置。
    if (ep->me_key == dummy)
        freeslot = ep;
    else {
```

```
        // 如果 hash 相同，且字符串内容也相同，返回这个 Entry。
        //   仅判断 hash 是不够的。哈希算法可能会出现不同字符串出现相同哈希值，这被称作哈希碰撞。
        //   如果两个不同 key 对象的值相同，也返回。
        if (ep->me_hash == hash && _PyString_Eq(ep->me_key, key))
            return ep;

        freeslot = NULL;
    }

    // 进行循环线性探测，要么找到，要么结束。
    for (perturb = hash; ; perturb >>= PERTURB_SHIFT) {

        // 计算下一个位置
        i = (i << 2) + i + perturb + 1;
        ep = &ep0[i & mask];

        // 如果是 Unused，探测结束。
        if (ep->me_key == NULL)
            // 要么返回以前记录的第一存储位置 freeslot，要么返回当前 entry。
            return freeslot == NULL ? ep : freeslot;

        // 如果 key 指向同一个对象，返回。
        // 如果 hash 相同，值相同，且不是 Dummy (那就是 Active 了)，返回。
        if (ep->me_key == key ||
            (ep->me_hash == hash && ep->me_key != dummy && _PyString_Eq(ep->me_key, key)))
            return ep;

        // 如果是 Dummy，且还没有第一存储位置，那么用 freeslot 记下。然后继续查找下一个位置。
        if (ep->me_key == dummy && freeslot == NULL)
            freeslot = ep;
    }

    // 没有任何可用位置
    return 0;
}
```

算法实现并不复杂，只是要注意判断: key 对象相同，或者 key 值相同。有很多种哈希计算方法，我们并不能预设哈希值和字符串是唯一对应的，所以在判断 hash 后还要进一步判断内容。当然，如果 hash 不同，则内容肯定也不同。通过短路条件，可提高算法性能。

2.4.1.2 resize

前面已经提到过，dict 会在需要的时候重新分配一个更大的 ma_table。我们看看是在什么时候，又是如何重新建立 ma_table 的。

Objects/dictobject.c

```
int PyDict_SetItem(register PyObject *op, PyObject *key, PyObject *value)
{
    ... ...

    mp = (PyDictObject *)op;
```

```
    // 计算 hash code
    if (PyString_CheckExact(key)) {
        hash = ((PyStringObject *)key)->ob_shash;
        if (hash == -1) hash = PyObject_Hash(key);
    }
    else {
        hash = PyObject_Hash(key);
        if (hash == -1) return -1;
    }

    ...

    // 获取一个可用位置，保存 key-value。(新增 或 修改)
    n_used = mp->ma_used;
    Py_INCREF(value);
    Py_INCREF(key);
    if (insertdict(mp, key, hash, value) != 0)
        return -1; // 如果没有可用位置，则返回 -1 表示失败。

    /* 1. 当 ma_fill (Active + Dummy) 占用率超过 2/3 时，就准备调整 ma_table 大小。
     * 2. 按照 ma_used (Active) 的数量，调整 2 倍 或 4 倍。
     * 3. 如果 ma_fill 比 ma_used 大出太多，意味着 Dummy 数量极多，那么就缩小 ma_table。
     */

    // 检查占用率: ma_fill / (ma_mask + 1) >= 2/3；如果未超过，无需调整。
    if (!(mp->ma_used > n_used && mp->ma_fill*3 >= (mp->ma_mask+1)*2))
        return 0;

    // 如果 ma_used (也就是 Active 的数量) 超过 50000，那么 2 倍扩容，否则 4 倍。
    return dictresize(mp, (mp->ma_used > 50000 ? 2 : 4) * mp->ma_used);
}

static int insertdict(register PyDictObject *mp, PyObject *key, long hash, PyObject *value)
{
    ... ...

    ep = mp->ma_lookup(mp, key, hash);

    // 没有可用位置，返回 -1。
    if (ep == NULL) {
        Py_DECREF(key);
        Py_DECREF(value);
        return -1;
    }

    ...

    // Active，直接修改 me_value 即可。
    if (ep->me_value != NULL) {
        old_value = ep->me_value;
        ep->me_value = value;
        ...
    }
```

```
    else {
        // Unused 变成 Active，需要调整 ma_fill 计数器。
        if (ep->me_key == NULL)
            mp->ma_fill++;
        else {
            // Dummy -> Active
            assert(ep->me_key == dummy);
            Py_DECREF(dummy);
        }

        // 设置 entry 内容。
        ep->me_key = key;
        ep->me_hash = (Py_ssize_t)hash;
        ep->me_value = value;

        // 累加 Active 计数。
        mp->ma_used++;
    }

    return 0;
}
```

总结一下上面的 ma_table 调整的相关内容：

- 条件：当占用率 (ma_fill, Active + Dummy) 超过 ma_table 容量的 2/3。
- 方式：如果 ma_used (Active) 大于 50000，扩容 2 倍，否则 4 倍。

注意调整方式的计算条件是 Active Entry 的数量，而不是当前已经分配的 ma_table 大小。如果 Active * 4 比 ma_table 容量小，那么就不是扩容，而是收缩了。出现这种状况，应该是大量执行 del 操作，使得许多 Entry 从 Active 变成 Dummy 造成的。

```
static int dictresize(PyDictObject *mp, Py_ssize_t minused)
{
    Py_ssize_t newsize;
    PyDictEntry *oldtable, *newtable, *ep;
    PyDictEntry small_copy[PyDict_MINSIZE];

    ... ...

    // 从 8 开始，逐步调整，直到 newsize > minused。确保容量总是 8 的倍数。
    for (newsize = PyDict_MINSIZE; newsize <= minused && newsize > 0; newsize <<= 1)
        ;

    ... ...

    oldtable = mp->ma_table;

    // 检查当前 mp_table 是否指向 ma_smalltable。
    is_oldtable_malloced = oldtable != mp->ma_smalltable;

    // 如果新分配的空间等于 8，则使用 smalltable。
    if (newsize == PyDict_MINSIZE) {
```

```
    newtable = mp->ma_smalltable;

    // 如果调整前后都是 smalltable ...
    if (newtable == oldtable) {

        // 没有 Dummy，直接返回。
        if (mp->ma_fill == mp->ma_used) {
            return 0;
        }

        // 将所有 Entry 全部拷贝到一个临时数组中。
        memcpy(small_copy, oldtable, sizeof(small_copy));
        oldtable = small_copy;
    }
}
else {
    // newsize 大于 8，显然需要重新分配内存了。
    newtable = PyMem_NEW(PyDictEntry, newsize);
    ...
}

... ...

// 此时，不管是新分配的，还是 smalltable，原有的 entry 都在 oldtable 里了。

// 调整 PyDictObject 状态
mp->ma_table = newtable;
mp->ma_mask = newsize - 1;

// 将新分配的 ma_table 初始化
memset(newtable, 0, sizeof(PyDictEntry) * newsize);

mp->ma_used = 0;
i = mp->ma_fill;
mp->ma_fill = 0;

// 将 oldtable 里的 Active Entry 重新 "存" 到新的 ma_table 里。
for (ep = oldtable; i > 0; ep++) {

    if (ep->me_value != NULL) {             // Active Entry 保存
        --i;
        insertdict_clean(mp, ep->me_key, (long)ep->me_hash, ep->me_value);
    }
    else if (ep->me_key != NULL) {          // Dummy Entry 放弃
        --i;
        assert(ep->me_key == dummy);
        Py_DECREF(ep->me_key);
    }
    /* else key == value == NULL:  nothing to do */
}

// 别忘了把旧 ma_table 给释放掉。当然 smalltable 就不用管了。
if (is_oldtable_malloced) PyMem_DEL(oldtable);
```

```
    return 0;
}
```

一个调整过程走下来，当真不容易啊。先得在请求参数的基础上按 8 的倍数计算实际分配容量。接下来就是分配新的空间，初始化。最后把原来的 Active Entry 写到新表中。为啥要重新写入呢？直接 memcpy 不行吗？那是因为前后两个表的容量不同，也就是说 ma_mask 不同，依据 "hash & ma_mask" 公式计算出的存储位置自然发生变化，因此需要重新探测后再保存。

当然，如果新分配容量没有超过 8，自然还用 ma_smalltable。可重新整理，剔除 Dummy 的事情依旧逃不掉。按说这个容量并没有改变，为何也要用一个临时表重新整理 Entry？一来是调整 ma_fill 的值，减少 2/3 这个触发机关。其次是清理掉 Dummy，减少线性探测往后搜索的次数，以提升性能。

内存调整并不是直接使用 realloc，而是重新分配一块更大或更小的内存，然后替换 ma_table。

2.4.1.3 del

移除操作其实并没有什么奥秘可挖，无非是亲眼见证一下 Entry 从 Active 到 Dummy 的过程。

Objects/dictobject.c

```
int PyDict_DelItem(PyObject *op, PyObject *key)
{
    ... ...

    // 计算 hash
    if (!PyString_CheckExact(key) ||
        (hash = ((PyStringObject *) key)->ob_shash) == -1) {
        hash = PyObject_Hash(key);
        if (hash == -1)
            return -1;
    }

    // 查找实际存储 Entry
    mp = (PyDictObject *)op;
    ep = (mp->ma_lookup)(mp, key, hash);

    // 没找到
    if (ep == NULL) return -1;

    // 找到，但 value 错误，引发异常。
    if (ep->me_value == NULL) {
        set_key_error(key);
        return -1;
    }

    // 通过设置 me_key = dummy，me_value = NULL 完成 Active -> Dummy 转换。
    old_key = ep->me_key;
    Py_INCREF(dummy);
```

```
    ep->me_key = dummy;
    old_value = ep->me_value;
    ep->me_value = NULL;

    // 减少 ma_used (Ative) 计数。
    mp->ma_used--;

    // 调整原 key-value 引用计数，以便被回收。
    Py_DECREF(old_value);
    Py_DECREF(old_key);

    return 0;
}
```

2.4.1.4 view

Python 2.7 新增了 Dictionary View Object，一种动态引用视图，支持对 dict keys、values 和 items 进行 set 集合运算。

```
In [1]: d1 = dict(a = 1, b = 2, c = 3); v1 = d1.viewitems(); v1
Out[1]: dict_items([('a', 1), ('c', 3), ('b', 2)])

In [2]: d2 = dict(b = 2, d = 4); v2 = d2.viewitems()

In [3]: d1 & d2       # dict 并不支持这些操作符
---------------------------------------------------------------------------
TypeError: unsupported operand type(s) for &: 'dict' and 'dict'

In [4]: v1 & v2
Out[4]: set([('b', 2)])

In [5]: v1 | v2
Out[5]: set([('a', 1), ('b', 2), ('c', 3), ('d', 4)])

In [6]: v1 - v2
Out[6]: set([('a', 1), ('c', 3)])

In [7]: v1 ^ v2
Out[7]: set([('a', 1), ('d', 4), ('c', 3)])
```

2.4.1.5 setdefault

只所以把这个方法拿出来说事，是因为我不止一次看到有人无意犯错。看下面的例子。

```
In [8]: class A(object):
   ....:     _cache = {}
   ....:
   ....:     def __init__(self, key):
   ....:         self.key = key
   ....:         print "init:", hex(id(self)), key
   ....:
```

```
   ....:     @classmethod
   ....:     def get(cls, key):
   ....:         return cls._cache.setdefault(key, cls(key))
   ....:

In [9]: a1 = A.get("a")
init: 0x10e6c3190 a           # 注意 id(self) 的输出结果。

In [10]: a2 = A.get("a")
init: 0x10e6c3090 a           # 注意 id(self) 的输出结果和 [2] 并不相同。

In [11]: a1 is a2             # 相同 key 共享对象的目的达到了。
Out[11]: True

In [12]: b1 = A.get("b")
init: 0x10e6c3290 b

In [13]: a1 is b1             # 不同 key 不受影响。
Out[13]: False
```

一段按 key 缓存和共享对象实例的代码。在 get 中，原作者使用 setdefault 完成两件事，首先是返回 key 已存在时的 value 缓存对象。其次，如果 key 不存在，则返回 cls(key) 创建的新对象，同时将其添加到 _cache 中。

老实说，这个 setdefault 很 Pythonic，用简单的代码完成了一个相对复杂的逻辑。但问题出在哪？注意到第 [9]、[10] 两条语句的输出了吗？按设计本意 [10] 直接由 setdefault 返回 [9] 所创建的对象，为何再次执行了 __init__？而且输出的 id(self) 的结果也不同。这肯定意味着每次调用 get 都会导致一个临时对象被创建。

```
In [14]: import dis

In [15]: dis.dis(A.get)
  8           0 LOAD_FAST                0 (cls)
              3 LOAD_ATTR                0 (_cache)
              6 LOAD_ATTR                1 (setdefault)
              9 LOAD_FAST                1 (key)
             12 LOAD_FAST                0 (cls)
             15 LOAD_FAST                1 (key)
             18 CALL_FUNCTION            1         # call cls(key)
             21 CALL_FUNCTION            2         # call setdefault
             24 RETURN_VALUE
```

从 dis 的反编译结果来看，每次都会在 setdefault 前调用 cls(key)，这就是生成临时对象的原因。从某种程度上导致了一个 "良性错误"，这个不会触发异常的错误并不影响对象共享功能，但却无端丧失了一些性能。

不管是从执行结果，还是做单元测试，都 "很难" 发现这个潜在的问题。而类似于上面这样的代码并不少见。诸位 Pythoner 在大唱 Pythonic 的时候，是否真的搞明白发生了什么？

2.4.2 defaultdict

前面我们看到了 setdefault 误用所引发的小小问题。想来某些伙计正在鄙视作者没有提供一个修正版本，这当真是误会了。常规修复方法非常简单，我写不写并没多大区别。

```
cls._cache.get(key) or cls._cache.setdefault(key, cls(key))
```

不过，我想让各位看的是下面这个特别版本。

```
In [16]: from collections import defaultdict

In [17]: from sys import _getframe

In [18]: class A(object):
   .....:     _cache = defaultdict(lambda: A(_getframe(1).f_locals["key"]))
   .....:
   .....:     def __init__(self, key):
   .....:         print "init:", hex(id(self)), key
   .....:
   .....:     @classmethod
   .....:     def get(cls, key):
   .....:         return cls._cache[key]
   .....:

In [19]: a1 = A.get("a")
init: 0x10e724bd0 a

In [20]: a2 = A.get("a")

In [21]: a1 is a2
Out[21]: True

In [22]: b1 = A.get("b")
init: 0x10e724990 b

In [23]: a1 is b1
Out[23]: False
```

从测试结果来看，我们成功修复了前一节遗留的问题。

关键先生就是 collections.defaultdict，这个 dict 的继承类提供了一个 __missing__。当找不到 key 时，将通过它调用 default_factory Callable 对象来返回结果，并添加到字典中。

```
In [17]: defaultdict.__missing__?
String Form:<method '__missing__' of 'collections.defaultdict' objects>
Docstring:
__missing__(key) # Called by __getitem__ for missing key; pseudo-code:
if self.default_factory is None: raise KeyError((key,))
self[key] = value = self.default_factory()
return value
```

我们要做的就是为 _cache 提供一个 default_factory 参数。由于 default_factory 是没有参数的，所以我必须从 stack frame 中获取 key 的值。

在这个案例中使用 defaultdict 并不合适，有卖弄技巧的嫌疑。不过作为演示，勉强合格。嘿嘿~~~

2.4.3 OrderedDict

谁说 dict 和 "有序" 两者不可兼得？肯定没认真阅读标准库文档。

```
In [24]: from string import letters

In [25]: from collections import OrderedDict

In [26]: kv = zip(letters[:6], range(6)); kv
Out[26]: [('a', 0), ('b', 1), ('c', 2), ('d', 3), ('e', 4), ('f', 5)]

In [27]: d = dict(kv); d.keys()
Out[27]: ['a', 'c', 'b', 'e', 'd', 'f']

In [28]: od = OrderedDict(kv); od.keys()
Out[28]: ['a', 'b', 'c', 'd', 'e', 'f']
```

注意 "有序" 和 "排序" 不是一回事，有序是指添加的顺序。

```
In [29]: od
Out[29]: OrderedDict([('a', 0), ('b', 1), ('c', 2), ('d', 3), ('e', 4), ('f', 5)])

In [30]: od["h"] = 7

In [31]: od["g"] = 6

In [32]: od.keys()
Out[32]: ['a', 'b', 'c', 'd', 'e', 'f', 'h', 'g']

In [33]: od.values()
Out[33]: [0, 1, 2, 3, 4, 5, 7, 6]

In [34]: del od["b"]

In [35]: od.keys()
Out[35]: ['a', 'c', 'd', 'e', 'f', 'h', 'g']

In [36]: od["b"] = 2

In [37]: od.keys()
Out[37]: ['a', 'c', 'd', 'e', 'f', 'h', 'g', 'b']
```

有序不是没代价的，如没必要，还是踏踏实实用 dict 吧。有关 OrderedDict 的实现方式，可在 IPython 里用 "??" 查看源码，我就不在这凑字了。

2.5 集合

集合用来保存一组无序的不重复对象。所谓不重复，除了不会有同一对象的两个引用外，还可能包括值不能相同。Python 提供了 set 和 frozenset 两种集合类型，后者是只读版本。

除了 Entry 没有 value 外，set 整个实现机制和 dict 极其相似，甚至可以说是特殊的 dict。

Include/setobject.h

```
#define PySet_MINSIZE 8

typedef struct {
    long hash;      /* cached hash code for the entry key */
    PyObject *key;
} setentry;

typedef struct _setobject PySetObject;
struct _setobject {
    PyObject_HEAD

    Py_ssize_t fill;  /* # Active + # Dummy */
    Py_ssize_t used;  /* # Active */
    Py_ssize_t mask;
    setentry *table;
    setentry *(*lookup)(PySetObject *so, PyObject *key, long hash);
    setentry smalltable[PySet_MINSIZE];

    long hash;                  /* only used by frozenset objects */
    PyObject *weakreflist;      /* List of weak references */
};
```

而集合特有的操作，大多也是通过循环来完成比较检查，并无新奇之处。

Objects/setobject.c

```
static PyObject * set_issubset(PySetObject *so, PyObject *other)
{
    ...

    // 如果 other 不是 set 类型，则通过创建一个临时 set 对象进行比较。
    if (!PyAnySet_Check(other)) {
        PyObject *tmp, *result;
        tmp = make_new_set(&PySet_Type, other);
        result = set_issubset(so, tmp);
        ...
        return result;
    }

    // 子集长度大于父集，那肯定没啥结果了。
    if (PySet_GET_SIZE(so) > PySet_GET_SIZE(other)) Py_RETURN_FALSE;
```

```
    // 循环检查自己的每个 Entry 是否都 "存在" 于父集中。
    while (set_next(so, &pos, &entry)) {
        int rv = set_contains_entry((PySetObject *)other, entry);
        if (rv == -1) return NULL;
        if (!rv) Py_RETURN_FALSE;
    }

    Py_RETURN_TRUE;
}
```

2.5.1 hash

默认情况下，set 和 list 都无法进行 hash 计算。如想将其作为 dict key，就必须转换成相应的只读版本。

```
In [1]: l = range(6); s = set(l)

In [2]: hash(l)
---------------------------------------------------------------------------
TypeError: unhashable type: 'list'

In [3]: hash(s)
---------------------------------------------------------------------------
TypeError: unhashable type: 'set'

In [4]: d = {}; d[tuple(l)] = 1; d[frozenset(s)] = 2       # tuple、frozenset 可以作为 dict key

In [5]: d
Out[5]: {(0, 1, 2, 3, 4, 5): 1, frozenset([0, 1, 2, 3, 4, 5]): 2}

In [6]: s2 = frozenset(l)

In [7]: s is s2                                            # 创建一个新的，内容相同的 frozenset。
Out[7]: False

In [8]: d[s2]                                              # 复合线性探测需求
Out[8]: 2
```

2.5.2 class

我们已经了解 dict 和 set 的线性探测函数对 key 的比较方法。因此，想将值唯一的自定义类型对象实例放入 set，就必须重载 __hash__ 和 __eq__ 两个方法。

```
In [9]: class A(object):
   .....:     def __init__(self, x):
   .....:         self.x = x
   .....:
   .....:     def __hash__(self):
   .....:         return hash(self.x)
   .....:
```

```
   .....:     def __eq__(self, o):          # 简易版本，实际上还需要判断 None 和 type。
   .....:         return self.x == o.x
   .....:

In [10]: s = set()

In [11]: s.add(A(1))

In [12]: s.add(A(1))

In [13]: s
Out[13]: set([<__main__.A at 0x10e775290>])

In [14]: s.add(A(2))

In [15]: s
Out[15]: set([<__main__.A at 0x10e775290>, <__main__.A at 0x10e775b10>])

In [16]: for o in s: print o.x
1
2
```

对于这节内容，一直很纠结，反复删删改改，不知是否要合并到 dict 中。不管底层实现如何类同，但从类型角度，放一起又着实不合适。

2.6 小结

写完这章，我最大的感觉就是累，心累。

不管我们曾写过多少程序，学过多少语言，似乎从没为一个整数变量操过心。我们大谈特谈着架构、模式以及许多玄之又玄的理论，却极少去关注过微观世界，去了解一个整数对象的生死往来。正如 雅克·贝汉 的《微观世界》那般，当我们用 "放大镜" 去观察这些被我们忽视的 "尘埃" 时，却惊讶地发现，竟有这样一个奇妙的世界。

或许你会不屑于那大段雷同的代码，鄙夷它缺乏可复用模式。但你可想过，自然界的生物何曾使用同一套皮肤，同一对翅膀。每个看似相同的东西，在微观世界里，总是有着许许多多的不同，它总是和目标相契合的，是唯一的，是造物主专门定制的。

"螺丝壳里做道场"，只有对每一个字节斤斤计较，才有了汇流成海的提升。CPython 的设计者们，并没有为了所谓的模式而套用规则。他们尝试用既有的东西去完成目标，从不曾为了漂亮而去浪费一个字节。我们看到他们留在注释里的纠结，感受他们追求最佳的尝试，也体悟到某些不得已的凑活。一切的一切，都源自 1989 年开始的坚持。

由细胞构建起的高等生命，同样可能毁于不起眼的微小病毒。

第 3 章 函数

函数人人会写，但函数到底是什么恐怕就不是个个都能说得清的了。与函数有关的概念不少，什么堆栈帧、调用堆栈、方法、匿名函数、Lambda 表达式等等。对了，还有让初学者头疼的闭包。很显然，函数是有大秘密可待发掘的。

在第一章我们看到过 Python 编译器会将模块和其内部成员编译成 PyCodeObject，那么这个是不是函数的真身呢？还有那个神秘的 PyFrameObject 与 locals 名字空间的关系？多次调用函数，名字空间又如何处理?

要搞清这一切，还得从 "def" 说起。

3.1 创建

在 Python 源码中，我们使用 def 来定义函数或者方法。在其他语言中，类似的东西往往只是一个语法声明关键字，但 def 却一个可执行的指令。也因为这条指令，引发了后面一连串的故事。

我们先准备一个简单的样本。

```
$ cat test.py
# -*- coding:utf-8 -*-

def test(s):
    print s
```

使用 compile 将其编译成 PyCodeObject，然后看看编译器是如何理解 def 指令的。

```
In [1]: code = compile(open("test.py").read(), "test.py", "exec")

In [2]: import dis; dis.dis(code)
  3           0 LOAD_CONST               0 (<code object test, file "test.py", line 3>)
              3 MAKE_FUNCTION            0
              6 STORE_NAME               0 (test)
              9 LOAD_CONST               1 (None)
             12 RETURN_VALUE
```

源码第 3 行 "def hello()" 被编译器理解为：

- 将编译的 "PyCodeObject test" 入栈。
- 使用 MAKE_FUNCTION 指令创建函数对象。
- 将这个函数对象用 "test" 名字保存起来。

这就是为什么说 def 是可执行指令的理由。通过这几条指令，我们同样可以确定一个结果，Code 和 Function 肯定不是一回事，def 和下面的函数体是完全不同的两个意思。

看看虚拟机核心函数是如何处理 MAKE_FUNCTION 这条指令的。

Python/ceval.c

```
PyObject * PyEval_EvalFrameEx(PyFrameObject *f, int throwflag)
{
     ... ...

    case MAKE_FUNCTION:
        // 从栈上弹出 PyCodeObject 对象。
        v = POP(); /* code object */

        // 创建函数对象，注意全局名字空间被带入。
        x = PyFunction_New(v, f->f_globals);

    ... ...
}
```

调用 PyFunction_New 创建 PyFunctionObject 对象，传入的参数分别是 PyCodeObject 和当前堆栈帧所持有的全局名字空间。

Include/funcobject.h

```
typedef struct {
    PyObject_HEAD
    PyObject *func_code;            // PyCodeObject
    PyObject *func_globals;         // 所在模块的全局名字空间
    PyObject *func_defaults;        // 参数默认值列表
    PyObject *func_closure;         // 闭包相关设置
    PyObject *func_doc;             // __doc__
    PyObject *func_name;            // __name__
    PyObject *func_dict;            // __dict__
    PyObject *func_weakreflist;     // 弱引用链表
    PyObject *func_module;          // 所在 Module
} PyFunctionObject;
```

Objects/funcobject.c

```
PyObject * PyFunction_New(PyObject *code, PyObject *globals)
{
    // 创建函数对象。
    PyFunctionObject *op = PyObject_GC_New(PyFunctionObject, &PyFunction_Type);

    // 一大堆的参数需要设置和初始化。
    if (op != NULL) {

        ... ...

        op->func_weakreflist = NULL;
        op->func_code = code;
        op->func_globals = globals;
        op->func_name = ((PyCodeObject *)code)->co_name;
        op->func_defaults = NULL; /* No default arguments */
        op->func_closure = NULL;
        op->func_doc = doc;
        op->func_dict = NULL;
```

```
        op->func_module = NULL;

        ... ...

        // 和所在的 module 对象关联起来。
        module = PyDict_GetItem(globals, __name__);
        if (module) {
            op->func_module = module;
        }
    }
    else
        return NULL;

    _PyObject_GC_TRACK(op);
    return (PyObject *)op;
}
```

很显然，PyFunctionObject 是对 PyCodeObject 的一种 "包装"。

PyCodeObject 本质上依然是一种静态源代码，只不过以字节码方式存储，因为它面向虚拟机。因此 Code 关注的是如何执行这些字节码，比如栈空间大小，各种常量变量符号列表，以及字节码与源码行号的对应关系等等。

PyFunctionObject 是运行期产生的。它提供一个动态环境，让 PyCodeObject 与运行环境关联起来。同时为函数调用提供一系列的上下文属性，诸如所在模块、全局名字空间、参数默认值等等。PyFunctionObject 让函数面向逻辑，而不仅仅是虚拟机。

PyFunctionObject 和 PyCodeObject 组合起来才是一个完整的函数。

3.2 堆栈帧

仅有 PyFunctionObject 是不够的。CPython 作为栈式虚拟机，也必须如同 x86 那样为函数在其所在线程上创建一个 "堆栈帧 (Stack Frame)" 来维持执行状态，这就是 PyFrameObject。

Include/frameobject.h

```
typedef struct _frame {
    PyObject_VAR_HEAD
    struct _frame *f_back;          // 调用堆栈 (Call Stack) 链表
    PyCodeObject *f_code;           // PyCodeObject
    PyObject *f_builtins;           // builtins 名字空间
    PyObject *f_globals;            // globals 名字空间
    PyObject *f_locals;             // locals 名字空间
    PyObject **f_valuestack;        // 和 f_stacktop 共同维护运行帧空间。保存栈底，相当于 RBP 寄存器。
    PyObject **f_stacktop;          // 运行栈顶，相当于 RSP 寄存器的作用。
    PyObject *f_trace;              // Trace function

    PyObject *f_exc_type, *f_exc_value, *f_exc_traceback; // 记录当前栈帧的异常信息

    PyThreadState *f_tstate;        // 所在线程状态
```

```
    int f_lasti;                    // 上一条字节码指令在 f_code 中的偏移量，类似 RIP 寄存器。
    int f_lineno;                   // 与当前字节码指令对应的源码行号

    ... ...

    PyObject *f_localsplus[1]; // 动态申请的一段内存，用来模拟 x86 堆栈帧所在内存空间。
} PyFrameObject;
```

除开 f_globals、f_locals、f_builtins 等名字空间外，我们还看到与 x86 寄存器相对应的几个字段。比如维持运行栈底位置的 f_valuestack/RBP，栈顶 f_stacktop/RSP，以及类似 RIP 指令寄存器的 f_lasti。所有的堆栈帧对象通过 f_back 构建调用堆栈链，我们在使用 sys._getframe 和 pdb 调试时早有接触。

函数执行所需的内存则是通过 f_localsplus 来实现的。

我们看看如何创建一个堆栈帧。

Objects/frameobject.c

```
PyFrameObject * PyFrame_New(PyThreadState *tstate, PyCodeObject *code, PyObject *globals,
                            PyObject *locals)
{
    // 从当前线程状态中获取上一个栈帧，通过这个链表，形成完整的调用堆栈。
    PyFrameObject *back = tstate->frame;

    ... 处理 binutils 名字空间 ...

    // 检查是否有优化处理。
    if (code->co_zombieframe != NULL) {
        // 貌似是重复使用一个挂在 PyCodeObject 上的 frame 对象。
        f = code->co_zombieframe;
        code->co_zombieframe = NULL;
        ...
    }
    else {
        Py_ssize_t extras, ncells, nfrees;

        ncells = PyTuple_GET_SIZE(code->co_cellvars); // 被内部嵌套函数使用的名字列表。
        nfrees = PyTuple_GET_SIZE(code->co_freevars); // 引用外层嵌套函数的名字列表。

        // 栈空间长度 = 字节码执行所需栈帧大小 + 函数局部变量个数 + 嵌套函数名字引用数量。
        extras = code->co_stacksize + code->co_nlocals + ncells + nfrees;

        // 创建 PyFrameObject 对象。
```

```
    //   又见 free_list。可以理解，超频繁的 frame 不使用缓存岂不奇怪。
    if (free_list == NULL) {
        f = PyObject_GC_NewVar(PyFrameObject, &PyFrame_Type, extras);
        ...
    }
    else {
        // 从缓存链表中获取一个 PyFrameObject 对象。
        --numfree;
        f = free_list;
        free_list = free_list->f_back;

        // 如果空间不够，则需要调整大小。Py_Size 检查 ob_size，也就是变长部分的大小。
        if (Py_SIZE(f) < extras) {
            f = PyObject_GC_Resize(PyFrameObject, f, extras);
            ...
        }
    }

    extras = code->co_nlocals + ncells + nfrees;

    // f_localsplus + extras 正好运行指令的那段内存开始处。
    // 起到 rbp 寄存器作用的 f_valuestack 将以此为栈底。
    f->f_valuestack = f->f_localsplus + extras;

    // 初始化用来保存参数等局部变量的内存区。
    for (i=0; i<extras; i++)
        f->f_localsplus[i] = NULL;

    f->f_code = code;
    f->f_locals = NULL;
    f->f_trace = NULL;
    f->f_exc_type = f->f_exc_value = f->f_exc_traceback = NULL;
}

// 将栈顶指针也指向栈底。
//   和我们通常所熟悉的堆栈帧地址从大到小不同，这回显然是反过来的。
//   对照上面的图示，就大概搞清楚 CPython 的堆栈帧内存模型了。
f->f_stacktop = f->f_valuestack;

// 设置相关属性。
f->f_back = back;
f->f_builtins = builtins;
f->f_globals = globals;

... 处理 lcoals 名字空间 ...

// 关联当前线程状态对象。
f->f_tstate = tstate;

// 设置执行指令初始化信息等。
f->f_lasti = -1;
f->f_lineno = code->co_firstlineno;
f->f_iblock = 0;
```

```
    _PyObject_GC_TRACK(f);
    return f;
}

#define PyFrame_MAXFREELIST 200

static void frame_dealloc(PyFrameObject *f)
{
    ... ...

    co = f->f_code;

    if (co->co_zombieframe == NULL)
        // 将优化版重复使用的 frame 还给 PyCodeObject。
        co->co_zombieframe = f;
    else if (numfree < PyFrame_MAXFREELIST) {
        // 将对象还给缓存链表。
        ++numfree;
        f->f_back = free_list;
        free_list = f;
    }
    else
        // 缓存链表存不下的，就释放掉。
        PyObject_GC_Del(f);

    ...
}
```

依旧请出 freelist + numfree 这对老搭档来缓存最多 200 个 PyFrameObject 对象。显然这对使用概率超频繁的堆栈帧来说十分重要。

一个初始化完成的堆栈帧对象，拥有存储局部变量和运行代码所需的内存，以及管理运行栈操作所需的指针。虽说 CPython 栈式虚拟机没有明确的堆栈寄存器，但依然用类似作用的指针完成了对 x86 架构的模拟。

万事具备，且看函数如何开始调用。

3.2 调用

为前面的例子加上函数调用。

```
$ cat test.py
# -*- coding:utf-8 -*-

def test(s):
    print s

test("Hello, World!")
```

这次的反汇编结果，多了我们所关心的内容。

```
In [1]: code = compile(open("test.py").read(), "test.py", "exec")

In [2]: import dis; dis.dis(code)
  3           0 LOAD_CONST               0 (<code object test, file "test.py", line 3>)
              3 MAKE_FUNCTION            0
              6 STORE_NAME               0 (test)

  6           9 LOAD_NAME                0 (test)
             12 LOAD_CONST               1 ('Hello, World!')
             15 CALL_FUNCTION            1
             18 POP_TOP
             19 LOAD_CONST               2 (None)
             22 RETURN_VALUE
```

源码第 6 行通过两条 LOAD 指令，分别将 PyFunctionObject test、常量字符串 "Hello, World!" PUSH 到当前运行栈。

Python/ceval.c

```
PyObject * PyEval_EvalFrameEx(PyFrameObject *f, int throwflag)
{

    ... ...

#define BASIC_PUSH(v)     (*stack_pointer++ = (v))
#define BASIC_POP()       (*--stack_pointer)

#define PUSH(v)           (void)(BASIC_PUSH(v))   // 简化了一下，便于略读。
#define POP()             (void)(BASIC_POP())

    ... ...

    stack_pointer = f->f_stacktop;   // 指向栈顶。默认情况下栈顶处于栈底位置，然后向右(地址增大)扩展。

    ... ...

    case LOAD_NAME:
        w = GETITEM(names, oparg);
        x = PyDict_GetItem(v, w);    // 依次从 LEGB 中查找目标对象，此处暂略。
        ... ...
        PUSH(x);                     // PUSH、POP 实际上就是操作 f_localsplus 中由
                                     //   f_stacktop/f_valuestack 管理的那段运行栈内存。
        continue;

    case LOAD_CONST:
        x = GETITEM(consts, oparg);
        Py_INCREF(x);
        PUSH(x);
        goto fast_next_opcode;

    ... ...
}
```

准备好调用参数后，再看看 CALL_FUNCTION 又将做什么。

Python/ceval.c

```
PyObject * PyEval_EvalFrameEx(PyFrameObject *f, int throwflag)
{
    ... ...

    stack_pointer = f->f_stacktop;          // 运行栈栈顶指针。

    case CALL_FUNCTION:
    {
        PyObject **sp;
        sp = stack_pointer;                 // 当前栈顶指针。目标函数的 code 等参数都需要通过该指针提取。

        x = call_function(&sp, oparg);      // 执行调用。传入栈顶指针和字节码参数 (记录了参数数量信息)。

        stack_pointer = sp;
        PUSH(x);                            // 返回值依然保存在调用堆栈帧上。

        break;
    }

    ... ...
}

static PyObject * call_function(PyObject ***pp_stack, int oparg)
{
    PyObject **pfunc = (*pp_stack) - n - 1;
    PyObject *func = *pfunc;

    ... ...

    if (PyFunction_Check(func))
        x = fast_function(func, pp_stack, n, na, nk);

    ... ...
}

static PyObject * fast_function(PyObject *func, PyObject ***pp_stack, int n, int na, int nk)
{
    ... ...

    // 针对没有参数默认值，没有 键参数(keyword arguments) 的优化调用。
    if (argdefs == NULL && co->co_argcount == n && nk==0 &&
        co->co_flags == (CO_OPTIMIZED | CO_NEWLOCALS | CO_NOFREE)) {

        // 为目标函数创建堆栈帧。
        f = PyFrame_New(tstate, co, globals, NULL);

        ... 处理栈帧所需的参数等数据 ...

        // 进入虚拟机核心函数 PyEval_EvalFrameEx，开始执行 code。
        retval = PyEval_EvalFrameEx(f,0);
```

```
        ... ...

        return retval;
    }

    ... ...

    // PyEval_EvalCodeEx 对付那些有参数的函数调用。
    return PyEval_EvalCodeEx(co, globals, (PyObject *)NULL, (*pp_stack)-n, na,
                             (*pp_stack)-2*nk, nk, d, nd, PyFunction_GET_CLOSURE(func));
}

PyObject * PyEval_EvalCodeEx(PyCodeObject *co, PyObject *globals, PyObject *locals,
           PyObject **args, int argcount, PyObject **kws, int kwcount,
           PyObject **defs, int defcount, PyObject *closure)
{
    ... ...

    // 创建 PyFrameObject 对象。
    f = PyFrame_New(tstate, co, globals, locals);

    ... 好长的参数处理过程，包括 位置参数(positional argument)、键参数(keyword arguments) 等等 ...
    ... 留待后面分析函数参数时再说 ...

    retval = PyEval_EvalFrameEx(f,0);

    ... ...

    return retval;
}

PyObject * PyEval_EvalFrameEx(PyFrameObject *f, int throwflag)
{
    ... ...

    PyThreadState *tstate = PyThreadState_GET();

    // 设置当前线程默认堆栈帧。这个很重要！sys._getframe(0)
    tstate->frame = f;

    ... ...
}
```

整理一下调用过程：

- 使用 LOAD 指令将目标函数 PyCodeObject 对象和所需参数依次压入当前运行栈。
- 执行 CALL_FUNCTION 指令，虚拟机用栈顶指针和指令参数调用 call_function 函数。
- call_function 通过栈顶指针提取包括 PyCodeObject 在内的执行参数，创建新的堆栈帧。
- 使用新堆栈帧再次进入虚拟机核心函数 PyEval_EvalFrameEx 执行目标函数字节码指令。
- 目标函数退出后，将其返回值压入调用者运行栈。

很熟悉吧，对分析过 _cdecl、_stdcall 函数调用的童鞋肯定没啥障碍。无非栈帧不是连续地址，不能像 RBP、RSP 那样直接做地址操作，改由传递当前运行栈栈顶指针来完成。

其实参数传递还另有玄机，如何将保存在调用者运行栈上的参数变成本地名字空间中的变量？请继续往下看。

3.4 参数

Python 支持多种函数形参表达方式，比如：

```
In [1]: def test(a, b, *args, **kwargs):
   ...:     pass
   ...:

In [2]: test(1, 2, "hello", "world", x = 100, y = 200)
```

形参 1、2 称为位置参数 (positional argument)，x、y 则被称为键参数 (keyword argument)。而函数声明中 *args、**kwargs 则是两者对应的扩展方式。

当函数被编译成 PyCodeObject 时，有几个相应参数信息会被记录下来。

- **co_argcount**: 参数个数，但不包括 *args 和 **kwargs。
- **co_nlocals**: 局部变量数量，其中包括参数数量。形参实际上就是局部变量。注意 *args 和 **kwargs 都只能算一个变量，因为它们在函数内部是用 tuple 和 dict 存储的。
- **co_varnames**: 所有局部变量名称，包括参数名。

```
In [3]: test.func_code.co_argcount
Out[3]: 2

In [4]: test.func_code.co_nlocals
Out[4]: 4

In [5]: test.func_code.co_varnames
Out[5]: ('a', 'b', 'args', 'kwargs')
```

看看编译器如何处理这些参数，并进行函数调用的。

```
In [6]: code = compile('test(1, 2, "hello", "world", x = 100, y = 200)', "test.py", "exec")

In [7]: dis.dis(code)
  1           0 LOAD_NAME                0 (test)
              3 LOAD_CONST               0 (1)
              6 LOAD_CONST               1 (2)
              9 LOAD_CONST               2 ('hello')
             12 LOAD_CONST               3 ('world')
             15 LOAD_CONST               4 ('x')
             18 LOAD_CONST               5 (100)
             21 LOAD_CONST               6 ('y')
```

```
             24 LOAD_CONST               7 (200)
             27 CALL_FUNCTION          516
             30 POP_TOP
             31 LOAD_CONST               8 (None)
             34 RETURN_VALUE
```

首先压入 test code 对象，然后将调用时所给出的参数依次入栈。键参数比位置参数多了压参数名的指令。

但是 CALL_FUNCTION 指令参数 516 是什么意思?

CALL_FUNCTION(argc)

Calls a function. The low byte of argc indicates the number of positional parameters, the high byte the number of keyword parameters.

Python 字节码指令最多只能有一个参数，因此用一个整数的高低位来分别记录函数调用时的键参数和位置参数数量。将 516 转换成十六进制 0x0204 以方便查看高低位，正好位置参数 4 个，键参数 2 个。

函数 call_function 会将 CALL_FUNCTION 指令参数 oparg 分解成相应的参数数量信息，并计算栈数据偏移量来获取第一个入栈的 PyCodeObject 对象。

Python/ceval.c

```
static PyObject * call_function(PyObject ***pp_stack, int oparg)
{
    int na = oparg & 0xff;                  // 获取低位字节，也就是位置参数的数量。
    int nk = (oparg>>8) & 0xff;             // 获取高位字节，键参数的数量。
    int n = na + 2 * nk;                    // 计算入栈的指令总数，键参数总是 2 个一组。

    PyObject **pfunc = (*pp_stack) - n - 1; // 直到参数指令数量，就可以通过偏移量计算出 Code 所在位置。
    PyObject *func = *pfunc;

    ... ...

    x = fast_function(func, pp_stack, n, na, nk);

    ... ...
}
```

到目前为止，所有的函数参数都压入原调用函数的运行栈内存中。虚拟机会将它们全部复制到被调用函数自己堆栈帧的 f_localsplus 内存中，以形成函数自己的名字空间。由于 CPython 总是按名字传递，底层实现上总是传递指针，因此并不需要付出太多的代价。

不同类型的参数复制策略也有不同。

Python/ceval.c

```
static PyObject * fast_function(PyObject *func, PyObject ***pp_stack, int n, int na, int nk)
{
```

```
    ... ppstack 指向调用函数栈顶位置 ...

    // 如果没有参数默认值，且参数数量等于指令数，那也就是说没有键参数。
    //   剩下的只有位置参数，自然是按序对应。这个最简单。
    if (argdefs == NULL && co->co_argcount == n && nk==0 &&
        co->co_flags == (CO_OPTIMIZED | CO_NEWLOCALS | CO_NOFREE)) {

        ... ...

        f = PyFrame_New(tstate, co, globals, NULL);
        fastlocals = f->f_localsplus;

        // 减去指令偏移量，指向调用者运行栈中第一个形参位置。(不是 code 啊)
        stack = (*pp_stack) - n;

        // 循环推进，将所有位置参数复制到 fastlcoals 头部。
        for (i = 0; i < n; i++) {
            fastlocals[i] = *stack++;
        }

        retval = PyEval_EvalFrameEx(f,0);

        ... ...
    }

    ... 有默认值或者键参数的进入 PyEval_EvalCodeEx 再做处理 ...

    if (argdefs != NULL) {
        d = &PyTuple_GET_ITEM(argdefs, 0);
        nd = Py_SIZE(argdefs);
    }

    return PyEval_EvalCodeEx(co, globals, (PyObject *)NULL, (*pp_stack)-n, na,
                             (*pp_stack)-2*nk, nk, d, nd, PyFunction_GET_CLOSURE(func));
}
```

看看默认参数和键参数会被如何处理。

Python/ceval.c

```
#define GETLOCAL(i)         (fastlocals[i])
#define SETLOCAL(i, value)  do { PyObject *tmp = GETLOCAL(i); \
                                 GETLOCAL(i) = value;         \
                                 Py_XDECREF(tmp);             \
                               } while (0)

PyObject * PyEval_EvalCodeEx(PyCodeObject *co, PyObject *globals, PyObject *locals,
           PyObject **args, int argcount, PyObject **kws, int kwcount,
           PyObject **defs, int defcount, PyObject *closure)
{
    ... ...

    f = PyFrame_New(tstate, co, globals, locals);
    fastlocals = f->f_localsplus;
```

```
if (co->co_argcount > 0 || co->co_flags & (CO_VARARGS | CO_VARKEYWORDS)) {

    // 先处理 **kwargs 参数
    if (co->co_flags & CO_VARKEYWORDS) {
        kwdict = PyDict_New();

        // 计算所有位置参数的数量，包括 *args。
        i = co->co_argcount;
        if (co->co_flags & CO_VARARGS) i++;

        // 因为所有 **kwargs 参数都保存在这个字典中，所以只需将该字典指针保存到 f_localsplus 即可。
        SETLOCAL(i, kwdict);
    }

    // 如果实际位置参数超过 co_argcount，那么肯定存在 *args。
    if (argcount > co->co_argcount) {
        // 用 n 记录函数声明中的位置参数数量。
        n = co->co_argcount;
    }

    // 先按顺序将位置参数写入 f_localplus 开头。
    for (i = 0; i < n; i++) {
        x = args[i];
        Py_INCREF(x);
        SETLOCAL(i, x);
    }

    // 在处理剩余的 *args。
    if (co->co_flags & CO_VARARGS) {

        // 实际数量 = 接收的位置参数总数 - co_argcount
        u = PyTuple_New(argcount - n);

        // *args 实际保存在一个 tuple 中，写到 f_localsplus。
        SETLOCAL(co->co_argcount, u);

        // 提取所有 *args 参数，保存到 tuple。
        for (i = n; i < argcount; i++) {
            x = args[i];
            PyTuple_SET_ITEM(u, i-n, x);
        }
    }

    // 处理键参数 (注意键参数只是位置参数的命名表示方式而已)
    for (i = 0; i < kwcount; i++) {
        ... ...

        // 键参数总是 name:value 成对出现。
        PyObject *keyword = kws[2*i];
        PyObject *value = kws[2*i + 1];

        // 提取所有变量名字，其实是为了核查键参数。
        co_varnames = ((PyTupleObject *)(co->co_varnames))->ob_item;
```

```
            // 循环检查是否有对应的键参数。
            for (j = 0; j < co->co_argcount; j++) {

                // 如果找到对应的键参数名，goto kw_found...
                PyObject *nm = co_varnames[j];
                if (nm == keyword) goto kw_found;
            }

            ... ...

            // 如果不是位置参数里的，自然要放到 **kwargs 了。
            PyDict_SetItem(kwdict, keyword, value);
            continue;

          kw_found:
            if (GETLOCAL(j) != NULL) {
                ... 目标位置已经有人了，自然表示重复给同一个参数赋值了，抛异常，没得说的 ...
                goto fail;
            }

            // 将该键参数指针写入指定位置参数内存。
            SETLOCAL(j, value);
        }

        // 处理默认参数。
        if (argcount < co->co_argcount) {
            ... ...

            // 循环检查所有有默认值的位置参数
            for (; i < defcount; i++) {
                // 如果其在 f_lcoalsplus 的存储位置为 NULL，就用默认值补上。
                if (GETLOCAL(m+i) == NULL) {
                    PyObject *def = defs[i];
                    SETLOCAL(m+i, def);
                }
            }
        }
    }

    ... 闭包相关，下一节再做分析 ...

}
```

所有调用参数都已复制到当前堆栈帧 f_localsplus 专门用来保存局部变量的内存中，可以继续处理名字空间问题了。在第一章我们只是介绍了名字空间的作用，但现在需要分析的是名字空间的实现机制。

3.5 名字空间

Python 语言的执行逻辑与名字空间息息相关，我们也早已知道大名鼎鼎的 LEGB 规则。在相关章节的分析过程中，明确了 B 对应 __builtins__.__dict__，G 则是 <module>.__dict__。那么 L 和 E 呢？

我们补上前面略过的内容，看看在堆栈帧中是如何体现 locals 名字空间的。

Objects/frameobject.c

```
PyFrameObject * PyFrame_New(PyThreadState *tstate, PyCodeObject *code, PyObject *globals,
                            PyObject *locals)
{
    PyFrameObject *f;

    ... ...

    // 处理 builtins 名字空间
    if (back == NULL || back->f_globals != globals) {

        // 如果是第一个 Frame，或者是动态执行提供自定义 globals 名字空间字典。

        // 看看 globals 中有没有 __builtins__ 模块。
        builtins = PyDict_GetItem(globals, builtin_object);

        // 如果有，就返回 __builtins__.__dict__。
        if (builtins) {
            if (PyModule_Check(builtins)) {
                builtins = PyModule_GetDict(builtins);
            }
            else if (!PyDict_Check(builtins))
                builtins = NULL;
        }

        // 如果是我们自定义的一个 global dict，自然没有 __builtins__。新建一个字典作为 builtins。
        if (builtins == NULL) {
            builtins = PyDict_New();
            ... ...
        }
    }
    else {
        // 正常状态下，我们总是共享 __builtins__.__dict__。
        builtins = back->f_builtins;

        ... ...
    }

    ... 创建或从缓存中获取 PyFrameObject 对象 ...

    /* Most functions have CO_NEWLOCALS and CO_OPTIMIZED set. */
    if ((code->co_flags & (CO_NEWLOCALS | CO_OPTIMIZED)) == (CO_NEWLOCALS | CO_OPTIMIZED)) {
        /* f_locals = NULL; will be set by PyFrame_FastToLocals() */
        // 也就是说大多数函数的 f_locals == NULL？难道是延迟分配？
        // 记住注释提到的这个 PyFrame_FastToLocals，待会用得着。
    }
```

```
    else if (code->co_flags & CO_NEWLOCALS) {
        // 如果有必要，则提前创建 locals 字典。
        locals = PyDict_New();
        f->f_locals = locals;
        ...
    }
    else {
        // 直接在 module，而不是某个函数里调用 locals() 时，locals == globals。
        if (locals == NULL) locals = globals;
        f->f_locals = locals;
        ...
    }

    ... ...
}
```

PyFrame_New 函数交待了 globals 和 builtins 名字空间的来源，但也给我们探索 locals 留下一个巨大的问号。

- 如果 f_locals 是延迟分配，那么延迟到什么时候?
- 相关的名字又是什么时候放进去的?

这回不想走弯路，暴力一点，直接从内置函数 locals 入手，反推。

Python/bltinmodule.c

```
static PyObject * builtin_locals(PyObject *self)
{
    ...

    d = PyEval_GetLocals();
    return d;
}
```

Python/ceval.c

```
PyObject * PyEval_GetLocals(void)
{
    PyFrameObject *current_frame = PyEval_GetFrame();
    if (current_frame == NULL) return NULL;

    PyFrame_FastToLocals(current_frame);
    return current_frame->f_locals;
}
```

很好，我们总算绕回到 PyFrame_New 注释里提到的那个 PyFrame_FastToLocals 函数。

Objects/frameobject.c

```
void PyFrame_FastToLocals(PyFrameObject *f)
{
    /* Merge fast locals into f->f_locals */

    ... ...
```

```
    // 延迟分配的 f_locals 字典果然在这里被创建。
    locals = f->f_locals;
    if (locals == NULL) {
        locals = f->f_locals = PyDict_New();
        ...
    }

    co = f->f_code;
    map = co->co_varnames;   // 局部变量名列表
    fast = f->f_localsplus;

    ... ...

    j = PyTuple_GET_SIZE(map);
    if (j > co->co_nlocals)
        j = co->co_nlocals;

    // 将存储在 f_localsplus 上的局部变量(包括参数)对象指针复制到 f_locals。
    //   参数: map_to_dict(map, nmap, dict, values, deref)
    if (co->co_nlocals)
        map_to_dict(map, j, locals, fast, 0);

    // 继续将 f_localsplus 上的嵌套函数变量引用复制到 f_locals。
    ncells = PyTuple_GET_SIZE(co->co_cellvars);
    nfreevars = PyTuple_GET_SIZE(co->co_freevars);
    if (ncells || nfreevars) {
        map_to_dict(co->co_cellvars, ncells, locals, fast + co->co_nlocals, 1);
        if (co->co_flags & CO_OPTIMIZED) {
            map_to_dict(co->co_freevars, nfreevars, locals, fast + co->co_nlocals + ncells, 1);
        }
    }

    ... ...
}
```

原来 f_localsplus 这块内存又被称做 "FAST"，它保存了所有复制过来的参数，以及当前函数创建的局部变量。 而函数 PyFrame_FastToLocals 会创建 f_locals 字典，并复制所有的局部变量。

当我们在 Python 源码中访问 f_locals 时，将通过 PyFrame_Type 激活下面这些函数来创建所需的名字空间字典。

Objects/frameobject.c

```
PyTypeObject PyFrame_Type = {
    frame_getsetlist,                                   /* tp_getset */
};

static PyGetSetDef frame_getsetlist[] = {
    {"f_locals", (getter)frame_getlocals, NULL, NULL},     // 访问 frame.f_locals 所使用的 get 函数
};

static PyObject * frame_getlocals(PyFrameObject *f, void *closure)
```

```
{
    // 依旧回到 PyFrame_FastToLocals。
    PyFrame_FastToLocals(f);
    return f->f_locals;
}
```

知道了 FAST 的实际含义，那么在 dis 中频繁出现的 LOAD_FAST、STORE_FAST 就很好解释了。

Python/ceval.c

```
PyObject * PyEval_EvalFrameEx(PyFrameObject *f, int throwflag)
{
    ... ...

#define GETLOCAL(i)         (fastlocals[i])
#define SETLOCAL(i, value)  do { PyObject *tmp = GETLOCAL(i); \
                                 GETLOCAL(i) = value;         \
                                 Py_XDECREF(tmp);             \
                            } while (0)
    ... ...

    fastlocals = f->f_localsplus;

    ... ...

    case LOAD_FAST:
        x = GETLOCAL(oparg);
        ...

    case STORE_FAST:
        v = POP();
        SETLOCAL(oparg, v);
        ...

    ... ...
}
```

绝大多数时候，函数只是通过 LOAD_FAST、STORE_FAST 等指令操作 f_localsplus 保存的局部变量，无需用到 f_locals。而且用索引号访问 f_lcoalsplus 指针数组也比字典快得多。

解决了 locals 名字空间的 "出生" 问题，就剩下 LEGB 的查找规则了。就拿 LOAD_NAME 这个典型案犯开刀吧。

Python/ceval.c

```
PyObject * PyEval_EvalFrameEx(PyFrameObject *f, int throwflag)
{
    ... Python 城堡的居民注意了，现在通缉在逃犯 ...

    co = f->f_code;
    names = co->co_names;

    case LOAD_NAME:
```

```
        // w: 目标的名字。
        w = GETITEM(names, oparg);

        // v: f_locals 名字空间
        if ((v = f->f_locals) == NULL) {
            ... ...
            break;
        }

        // 先从 v，也就是 f_locals 中查找。
        if (PyDict_CheckExact(v)) {
            x = PyDict_GetItem(v, w);
            Py_XINCREF(x);
        }
        else {
            // 从 v.__dict__ 中查找，这涉及到 class 的查找规则，后面章节会提到。
            x = PyObject_GetItem(v, w);
            ...
        }

        // 没找到?
        if (x == NULL) {
            // 继续翻 f_globals。
            x = PyDict_GetItem(f->f_globals, w);

            // 还没找到? 这死孩子跑哪去了。
            if (x == NULL) {
                // f_builtins 是最后一家了。如果还找不到就只能作为失踪人口抛出异常了。
                x = PyDict_GetItem(f->f_builtins, w);
                if (x == NULL) {
                    format_exc_check_arg(PyExc_NameError, NAME_ERROR_MSG, w);
                    break;
                }
            }
        }

        PUSH(x);

    ... ...
}
```

查找过程简单明了。只是整个过程只看到了 L -> G -> B，外层嵌套函数的 E 呢?

忘了? PyFrame_FastToLocals 函数生成 f_lcoals 时就已经复制了 co_cellvars、co_freevars 这两个嵌套函数变量引用列表。这就是 E，哪里真需要去引用嵌套函数的整个名字空间。LEGB 的说法只是相对于程序员编写 Python 源码而言的，CPython 底层实现自然不会照搬。同时这也是闭包的实现原理，至于细节如何，且待下回分解。

3.6 闭包

作为现代编程语言，不支持闭包会被人骂死。除了 JAVA 社区曾经闹哄哄的口水仗外，连定位于系统开发的 GoLang 都毫不迟疑地支持了。

好了，不废话了，先整个闭包出来再说。

```
In [1]: def test(s):
   ....:     def a():
   ....:         print s
   ....:     return a
   ....:

In [2]: dis.dis(test)
  2           0 LOAD_CLOSURE             0 (s)
              3 BUILD_TUPLE              1
              6 LOAD_CONST               1 (<code object a)
              9 MAKE_CLOSURE             0
             12 STORE_FAST               1 (a)

  4          15 LOAD_FAST                1 (a)
             18 RETURN_VALUE
```

嗯？怎么没有 MAKE_FUNCTION？难道生成内层函数 a 不需要创建 PyFunctionObject？

事情有点复杂，一步步来，先看看 LOAD_CLOSURE。

Python/ceval.c

```
PyObject * PyEval_EvalFrameEx(PyFrameObject *f, int throwflag)
{
    ... ...

    register PyObject **fastlocals, **freevars;
    freevars = f->f_localsplus + co->co_nlocals;

    case LOAD_CLOSURE:
        x = freevars[oparg];         // 从 f_lcoalsplus 指针数组中取?
        PUSH(x);
        break;

    ... ...
}
```

从 f_localsplus 取被内层函数所引用的变量？这数据从何而来？

首先，编译器在生成 PyCodeObject test 时就已经确定了被内层函数引用的变量，并将它们的名字保存在 co_cellvars 字段里。

```
In [3]: test.func_code.co_cellvars
Out[3]: ('s',)
```

其次，在调用 test 函数时，会将所有参数从调用者的运行栈拷贝到当前函数堆栈帧，其中就包括了被嵌套函数所引用的参数 s。前面章节分析参数复制时，我们省略了有关闭包的代码，现在补上。

Python/ceval.c

```
#define GETLOCAL(i)         (fastlocals[i])
#define SETLOCAL(i, value)  do { PyObject *tmp = GETLOCAL(i); \
                                 GETLOCAL(i) = value;         \
                                 Py_XDECREF(tmp);             \
                            } while (0)

PyObject * PyEval_EvalCodeEx(PyCodeObject *co, PyObject *globals, PyObject *locals,
           PyObject **args, int argcount, PyObject **kws, int kwcount,
           PyObject **defs, int defcount, PyObject *closure)
{
    ... ...

    f = PyFrame_New(tstate, co, globals, locals);
    fastlocals = f->f_localsplus;
    freevars = f->f_localsplus + co->co_nlocals;

    ... 处理位置参数和键参数 ...

    if (PyTuple_GET_SIZE(co->co_cellvars)) {

        // 统计所有函数调用参数数量（加上 *args, **kwargs 这两个参数)。
        nargs = co->co_argcount;
        if (co->co_flags & CO_VARARGS)
            nargs++;
        if (co->co_flags & CO_VARKEYWORDS)
            nargs++;

        // 循环 co_cellvars
        for (i = 0; i < PyTuple_GET_SIZE(co->co_cellvars); ++i) {

            // 获取被内层函数引用的变量名字。
            cellname = PyString_AS_STRING(PyTuple_GET_ITEM(co->co_cellvars, i));
            found = 0;

            // 从所有调用参数中匹配被引用的变量名字。
            for (j = 0; j < nargs; j++) {
                argname = PyString_AS_STRING(PyTuple_GET_ITEM(co->co_varnames, j));
                if (strcmp(cellname, argname) == 0) {

                    // 如果匹配成功，则取出参数值，创建 PyCellObject。
                    c = PyCell_New(GETLOCAL(j));
                    if (c == NULL) goto fail;

                    // 将 PyCellObject 保存到 f_localsplus 对应位置。
                    GETLOCAL(co->co_nlocals + i) = c;
                    found = 1;
                    break;
```

```
                }
            }

            // 没找到则写入一个空的 PyCellObject。
            if (found == 0) {
                c = PyCell_New(NULL);
                if (c == NULL) goto fail;
                SETLOCAL(co->co_nlocals + i, c);
            }
        }
    }

    ... ...
}
```

现在明白被内层函数所引用的参数 s 是如何进入 f_localsplus 专门用来保存 cellvars 的内存空间了。但如果 s 不是 test 的参数，而是一个本地局部变量呢？那么上面扫描 test 参数的代码显然是不起作用的。

```
In [4]: def test2():
   ....:     s = datetime.datetime.now()
   ....:     def a():
   ....:         print s
   ....:

In [5]: dis.dis(test2)
  2           0 LOAD_GLOBAL              0 (datetime)
              3 LOAD_ATTR                0 (datetime)
              6 LOAD_ATTR                1 (now)
              9 CALL_FUNCTION            0
             12 STORE_DEREF              0 (s)

  3          15 LOAD_CLOSURE             0 (s)
             18 BUILD_TUPLE              1
             21 LOAD_CONST               1 (<code object a)
             24 MAKE_CLOSURE             0
             27 STORE_FAST               0 (a)
             30 LOAD_CONST               0 (None)
             33 RETURN_VALUE
```

事情落在 STORE_DEREF 的头上。

Python/ceval.c

```
PyObject * PyEval_EvalFrameEx(PyFrameObject *f, int throwflag)
{
    ... ...

    register PyObject **fastlocals, **freevars;
    freevars = f->f_localsplus + co->co_nlocals;

    case STORE_DEREF:
    {
```

```
        PyObject **freevars = (f->f_localsplus + f->f_code->co_nlocals);
        PyObject *c = freevars[PEEKARG()];
        if (PyCell_GET(c) == v) PyCell_Set(c, NULL);
        break;
    }

    ... ...
}
```

直接操作 f_localsplus，倒省了我们一番手脚。不过这一再提到的这个 PyCellObject 是什么?

Include/cellobject.h

```
typedef struct {
       PyObject_HEAD
       PyObject *ob_ref;     /* Content of the cell or NULL when empty */
} PyCellObject;
```

很简单的一个类型，保存被引用的变量对象。

在虚拟机创建内层函数之前，相关被引用变量已经被打包成 PyCellObject 存放在 f_localsplus 上准备待命了。LOAD_CLOSURE 不过是取出这些 cell 对象，然后由 BUILD_TUPLE 指令打包成一个列表 (元组类型)，压入运行栈。

接下来，瞅瞅 MAKE_CLOSURE 是何方神圣，如何使用这个被引用的变量列表。

Python/ceval.c

```
PyObject * PyEval_EvalFrameEx(PyFrameObject *f, int throwflag)
{
    ... ...

    case MAKE_CLOSURE:
    {
        v = POP();                                  // 弹出 LOAD_CONST 压入的 PyCodeObject。
        x = PyFunction_New(v, f->f_globals);        // 创建函数对象。

        if (x != NULL) {
            v = POP();                              // 注意: 该值 BUILD_TUPLE 指令创建的 Cell 列表。
            if (PyFunction_SetClosure(x, v) != 0) { // 存入 PyFunctionObject.func_closure。
               ...
            }
        }

        ... ...

        PUSH(x);
    }

    ... ...
}
```

用 PyFunction_New 创建内层函数对象，然后调用 PyFunction_SetClosure 函数将所引用的外层变量列表保存到 func_closure。

此处有个小小的问题，内层函数 a.func_closure 持有具体的外层参数 s 的引用。如果我多次调用 test 返回 a 函数，那么 func_closure 怎么办？被后面的调用覆盖？这显然不符合闭包的逻辑。

```
In [6]: a1 = test("hello, world!"); a2 = test(100)

In [7]: a1 is a2
Out[7]: False

In [8]: a1.func_closure, a2.func_closure
Out[8]:
((<cell at 0x102a31e18: str object at 0x102a31e30>,),
 (<cell at 0x102a31d00: int object at 0x7ff883413930>,))
```

MAKE_CLOSURE 每次都会通过 PyFunction_New 创建一个新的 PyFunctionObject 对象，因此不会存在 func_closure 被覆盖的问题。

编译器将外层函数被引用的变量名字保存到 PyCodeObject.co_cellvars，而内存函数则对应的将所引用的外层变量保存到 co_freevars。

```
In [9]: a.func_code.co_freevars
Out[9]: ('s',)
```

在复制内层函数参数时，同样会把 func_closure 中的引用变量拷贝到 f_localsplus。

Python/ceval.c

```
#define GETLOCAL(i)         (fastlocals[i])
#define SETLOCAL(i, value)  do { PyObject *tmp = GETLOCAL(i); \
                                 GETLOCAL(i) = value;         \
                                 Py_XDECREF(tmp);             \
                            } while (0)

PyObject * PyEval_EvalCodeEx(PyCodeObject *co, PyObject *globals, PyObject *locals,
           PyObject **args, int argcount, PyObject **kws, int kwcount,
           PyObject **defs, int defcount, PyObject *closure)
{
    ... ...

    f = PyFrame_New(tstate, co, globals, locals);
    fastlocals = f->f_localsplus;
    freevars = f->f_localsplus + co->co_nlocals;

    ... 处理位置参数和键参数 ...

    // 处理所引用的外层变量
    if (PyTuple_GET_SIZE(co->co_freevars)) {

        // 循环所引用的外部变量名称列表
```

```
        for (i = 0; i < PyTuple_GET_SIZE(co->co_freevars); ++i) {
            // 从 closure 中提取对应的 Cell 对象
            PyObject *o = PyTuple_GET_ITEM(closure, i);

            // 保存到 f_localsplus 专门用来存储 freevars 的内存段。
            freevars[PyTuple_GET_SIZE(co->co_cellvars) + i] = o;
        }
    }

    ... ...
}
```

现在，我们搞清了闭包所引用的外部变量是如何被一步一步存储到内层函数的堆栈帧中的。剩下的，就是看看内层函数 a 如何使用这个闭包变量了。

```
In [10]: a = test("Hello, World!")

In [11]: dis.dis(a)
  3           0 LOAD_DEREF               0 (s)
              3 PRINT_ITEM
              4 PRINT_NEWLINE
              5 LOAD_CONST               0 (None)
              8 RETURN_VALUE
```

不是 LOAD_FAST，嘿嘿，看样子还要拆 PyCellObject。

Python/ceval.c

```
PyObject * PyEval_EvalFrameEx(PyFrameObject *f, int throwflag)
{
    ... ...

    freevars = f->f_localsplus + co->co_nlocals;

    case LOAD_DEREF:
        x = freevars[oparg];
        w = PyCell_Get(x);
        if (w != NULL) {
            PUSH(w);
            continue;
        }

    ... ...
}
```

CPython 实现闭包的原理并不复杂，说白了就是将所引用的外层对象附加到每次都重新创建的内层函数对象身上 (func_closure)。

3.7 lambda

Python lambda 除了不支持语句外，和普通函数并没有多大区别，相当于简化版的匿名函数。

```
In [1]: code = compile("func = lambda x: x + 1", "test.py", "exec")

In [2]: dis.dis(code)
  1           0 LOAD_CONST               0 (<code object <lambda>, file "test.py", line 1>)
              3 MAKE_FUNCTION            0
              6 STORE_NAME               0 (func)
              9 LOAD_CONST               1 (None)
             12 RETURN_VALUE

In [3]: exec code

In [4]: whos
Variable   Type        Data/Info
------------------------------
code       code        <code object <module>, line 1>
dis        module      <module 'dis'>
func       function    <function <lambda> at 0x10dc4b1b8>

In [5]: dis.dis(func)
  1           0 LOAD_FAST                0 (x)
              3 LOAD_CONST               1 (1)
              6 BINARY_ADD
              7 RETURN_VALUE
```

使用 lambda 时，需要注意延迟执行造成的问题。

```
In [5]: fs = [lambda: i for i in range(4)]

In [6]: for f in fs: print f()
3
3
3
3
```

咋都是 3 呢？列表生成表达式中的循环变量 i 在当前名字空间中是唯一的，也就是说每次生成 lambda 函数时，引用的都是同一个变量。在执行 lambda 函数前，i 已经等于 3，所以输出这个结果很正常。这就是所谓的延迟执行问题。

看看生成的 lambda 函数字节码指令就明白了。

```
In [7]: i
Out[7]: 3

In [8]: dis.dis(fs[0])
  1           0 LOAD_GLOBAL              0 (i)
              3 RETURN_VALUE

In [9]: dis.dis(fs[1])
  1           0 LOAD_GLOBAL              0 (i)
              3 RETURN_VALUE
```

解决方法很简单，就是每次生成 lambda 函数时，创建一个自己的局部变量来持有 i 的值。(因为 lambda 仅支持表达式，所以用参数这个特殊的局部变量就行了)

```
In [10]: fs = [lambda x = i: x for i in range(4)]

In [11]: dis.dis(fs[0])
  1           0 LOAD_FAST                0 (x)
              3 RETURN_VALUE

In [12]: dis.dis(fs[1])
  1           0 LOAD_FAST                0 (x)
              3 RETURN_VALUE

In [13]: for f in fs: print f()
0
1
2
3
```

闭包也有同样的问题，所有内层函数都持有同一个外层对象。

```
In [14]: def test():
   ....:     fs = []
   ....:     for i in range(4):
   ....:         def a():
   ....:             print i
   ....:         fs.append(a)
   ....:     return fs
   ....:

In [15]: fs = test()

In [16]: for f in fs: f()
3
3
3
3

In [17]: fs[0].func_closure
Out[17]: (<cell at 0x10eac8bb0: int object at 0x7fde63c132d8>,)

In [18]: fs[1].func_closure
Out[18]: (<cell at 0x10eac8bb0: int object at 0x7fde63c132d8>,)
```

解决方法和 lambda 类似。

```
In [19]: def test():
   ....:     fs = []
   ....:     for i in range(4):
   ....:         def a(x = i):
   ....:             print x
   ....:         fs.append(a)
   ....:     return fs
```

```
   ....:

In [20]: fs = test()

In [21]: for f in fs: f()
0
1
2
3

In [22]: dis.dis(test)
  4          25 LOAD_FAST                1 (i)                          # 区别就在这，将 i 入栈
             28 LOAD_CONST               2 (<code object a, line 4>)
             31 MAKE_FUNCTION            1                              # 创建函数 a 时，i 的值被当作
             34 STORE_FAST               2 (a)                          #    缺省参数值保存起来了。


In [23]: fs[0].func_defaults                                            # 执行 f()，自然是用缺省值了。
Out[23]: (0,)

In [24]: fs[1].func_defaults
Out[24]: (1,)
```

如果将 x 从参数移到函数体，则问题依旧。为啥？因为依然是闭包的延迟引用。

嵌套函数和 lambda 表达式每次都通过 MAKE_FUNCTION 指令生成新的 PyFunctionObject 对象，尽可能避免在循环中被重复创建。

3.8 其他

基于我们前面所获知的 "秘密"，看看日常用 Python 函数时，需要注意些什么。

3.8.1 引用传递

和 C 总是按值传递不同，Python 总是按 "引用传递"，形参不过是原对象的另外一个名字而已。

```
In [1]: def test(x):
   ...:     print hex(id(x))
   ...:

In [2]: a = 258                      # 注意避开 small_ints 小整数缓存啊！

In [3]: hex(id(a))
Out[3]: '0x7ff8840b0670'

In [4]: test(a)
0x7ff8840b0670
```

在折腾完前面有关堆栈帧和函数调用的内容后，想必你对此深有体会。用来存储各类参数、局部变量以及运行栈的 f_localsplus 是个指针数组，整个函数调用过程中，倒来倒去的也都是指针。毕竟所有的对象都在堆上分配，value-copy 的代价忒可怕了。

如果不希望你的对象被意外修改，那么尽可能用不可变对象，诸如 tuple、frozenset 之类的，或者有必要使用深拷贝 (deep copy)。

```
In [5]: import copy

In [6]: a = [1, 2, 3]; b = {"a":a}

In [7]: cb = copy.deepcopy(b)        # 深度递归复制，包括 b 包含的 a。

In [8]: cb is b, cb["a"] is a        # 检查一下，显然和 a、b 都脱离关系了。
Out[8]: (False, False)

In [9]: cb["a"].append(100)          # 对复制品的修改，不再影响原来的对象。

In [10]: cb
Out[10]: {'a': [1, 2, 3, 100]}

In [11]: b, a
Out[11]: ({'a': [1, 2, 3]}, [1, 2, 3])
```

与深度复制对应的还有一个浅拷贝 (shallow copy)。

```
In [12]: sb = copy.copy(b)           # 浅拷贝会创建一个复制品对象，但对其成员只不过是引用复制。

In [13]: sb["x"] = 100               # 修改复制品自身不会影响原对象。

In [14]: sb, b
Out[14]: ({'a': [1, 2, 3], 'x': 100}, {'a': [1, 2, 3]})

In [15]: sb["a"].append(100)         # 但其成员依旧与原对象共享一个目标

In [16]: sb, a, b
Out[16]: ({'a': [1, 2, 3, 100], 'x': 100}, [1, 2, 3, 100], {'a': [1, 2, 3, 100]})
```

标准库对两者的不同说明：

- A shallow copy constructs a new compound object and then (to the extent possible) inserts references into it to the objects found in the original.
- A deep copy constructs a new compound object and then, recursively, inserts copies into it of the objects found in the original.

某些时候也可以用 marshal 等序列化方式代替 deep-copy。尽可能让参数对象不可变只是一种理想化的编程范式，但凡事都不是绝对的，毕竟深度复制和序列化本身会引发一定的性能问题。

只有了解才能应用自如，切莫出错了，还浑然不知所以。

3.8.2 默认参数

似乎有不少人遇到过由默认参数引发的 "惨案"。

```
In [1]: def get_list(a, b, l = []):          # 本意是如果不提供列表，就默认创建一个。
   ....:     l.append(a)
   ....:     l.append(b)
   ....:     return l
   ....:

In [2]: get_list(1, 2, [3, 4])               # 显式提供，默认值自然不会搀和。
Out[2]: [3, 4, 1, 2]

In [3]: get_list(1, 2)                       # 使用默认值提供的 list，感觉良好。
Out[3]: [1, 2]

In [4]: get_list("a", "b")                   # 杯具了，历史遗留问题出现了。
Out[4]: [1, 2, 'a', 'b']
```

故障原因很简单，就是默认值对象保存在 PyFunctionObject.func_defaults 中，除非创建新的函数对象，否则会一直使用同一个默认值对象，而非我们想像中每次都会重新创建新的实例。

```
In [5]: get_list.func_defaults
Out[5]: ([1, 2, 'a', 'b'],)

In [6]: get_list(1, 2) is get_list(3, 4)
Out[6]: True

In [7]: get_list(1, 2) is get_list.func_defaults[0]
Out[7]: True
```

默认值是个好东西，从一定程度上解决了函数重载 (overload) 的问题。想当初 C# 不支持默认值，开发类库就是个苦逼的活。一堆无聊透顶的函数重载代码，看着都心烦。

尽量避免使用可变对象作为参数默认值。

3.8.3 多变参数

keyword argument、*args、**kwargs 让我们调用函数时非常方便，不过也有不称心的时候。

```
In [1]: def test(a, b):
   ....:     print a, b
   ....:

In [2]: d = { "a":1, "b":2, "c":3 }

In [3]: test(**d)            # test 没有 *args 和 **kwargs 收集多余的值。
--------------------------------------------------------------------------
```

```
TypeError: test() got an unexpected keyword argument 'c'

In [4]: d.values()            # dict 是无序存储的，也没法使用。
Out[4]: [1, 3, 2]
```

当然，你可以手工从字典取值，但那不符合 Pythonic 精神 (有时我也搞不清是精神还是神经，权且当作一种技巧吧。假如 test 参数和 d 的数量都很多，手工的确傻了点)。

```
In [5]: test(**{k:d[k] for k in test.func_code.co_varnames if k in d})
1 2
```

这个做法很 Pythonic 吧。test 的参数名肯定保存在 co_varnames 中，如果这个名字同时出现在 d.keys 里面，那么就是我们要的参数。用一个字典生成表达式就可以创建完全复合目标函数需求的参数字典了。别忘了用 "**" 展开。

Python 2.6.x 不支持 "{k:v for ...}"，改一下即可。

```
In [6]: test(**dict([(k, d[k]) for k in test.func_code.co_varnames if k in d]))
1 2
```

下面是一个初始化开关参数的案例。

- 需要设置 n 多个开关参数。
- 显式初始化部分参数，未被初始化的参数值取反。
- 也可以提供一个默认初始化值。

```
In [7]: def gen_options(keys, value, default = None):
   ....:     vars = ("a", "b", "c", "d", "e", "f")
   ....:     if default is None: default = not value
   ....:     return {k: value if k in keys else default for k in vars}
   ....:

In [8]: gen_options((), True)
Out[8]: {'a': False, 'b': False, 'c': False, 'd': False, 'e': False, 'f': False}

In [9]: gen_options(("a", "c"), True)
Out[9]: {'a': True, 'b': False, 'c': True, 'd': False, 'e': False, 'f': False}

In [10]: gen_options(("a", "c"), True, default = 0)
Out[10]: {'a': True, 'b': 0, 'c': True, 'd': 0, 'e': 0, 'f': 0}
```

某些时候，我们需要对某个函数进行整容，比如使其复合某个 callback 的签名要求。或者为某些参数提供默认值，免得累死人。(有没有想起 Windows API 那些非常可怕的冗长参数列表)

```
In [11]: def test(a, b, c, d):
   ....:     print a, b, c, d
   ....:

In [12]: import functools
```

```
In [13]: t2 = functools.partial(test, c = 3, d = 4)

In [14]: t2(1, 2)
1 2 3 4

In [15]: t3 = functools.partial(test, a = 1, b = 2)

In [16]: t3(c = 3, d = 4)                    # 不能直接用 t3(3, 4)，那样就是对 a 和 b 参数赋值了。
1 2 3 4

In [17]: t4 = functools.partial(test, 1, 2, 3, 4)

In [18]: t4()
1 2 3 4
```

3.8.4 global

使用 global 关键字，允许我们在函数内部修改或创建全局变量。但如果我还想使用同名的局部变量咋办?

```
In [1]: import sys

In [2]: x = 100

In [3]: def test(x):
   .....:     print "local x:", x
   .....:     sys.modules[__name__].x = x
   .....:

In [4]: x
Out[4]: 100

In [5]: test(200)
local x: 200

In [6]: x
Out[6]: 200
```

如果是先引用了 global x，然后又想创建本地局部变量 x 呢？用 lcoals() 返回当前函数的本地名字空间，自己加吧。

因为这些做法有点无厘头，我就此打住了，免得被人骂。呵呵～～～～

3.8.5 nonlocal

Python 2.7 迟迟不添加对 nonlocal 的支持，这让我很不满意。

```
Python 3.2.2 (default, Nov 26 2011, 22:40:16)

>>> def test():
```

```
...     x = 0
...     def a():
...         nonlocal x
...         x = 100
...     a()
...     print(x)
...
>>> test()
100
```

换到 Python 2.7 就 invalid syntax，哼哼！

```
Python 2.7.1 (r271:86832, Jun 16 2011, 16:59:05)

In [1]: def test():
   ...:     x = 0
   ...:     def a():
   ...:         nonlocal x
   ...:         x = 100
   ...:     a()
   ...:     print x
   ...:
  File "<ipython-input-1-72119faa3ee9>", line 4
    nonlocal x
             ^
SyntaxError: invalid syntax
```

那 Python 2.7 如何修改外层嵌套函数的变量？global 显然是不行的，用 sys._getframe(1) 获取外层函数的堆栈帧，然后通过 f_locals 名字空间进行修改呢？

```
In [2]: def test():
   ....:     x = 0
   ....:     def a():
   ....:         e_locals = sys._getframe(1).f_locals
   ....:         print e_locals
   ....:         e_locals["x"] = 100
   ....:         print sys._getframe(1).f_locals
   ....:     a()
   ....:     print x
   ....:

In [3]: test()
{'a': <function a at 0x10eb2ff50>, 'x': 0}
{'a': <function a at 0x10eb2ff50>, 'x': 0}
0
```

可耻地失败了。为什么？因为每次访问 f_locals 都会调用 PyFrame_FastToLocals 函数，以便将 f_localsplus 上的变量值刷新到 f_locals 字典。

Objects/frameobject.c

```
void PyFrame_FastToLocals(PyFrameObject *f)
{
```

```
    ... ...

    // 尽管不是每次都创建新的字典。
    locals = f->f_locals;
    if (locals == NULL) {
        locals = f->f_locals = PyDict_New();
        ...
    }

    ... ...

    // 但每次都用 f_localsplus 上的变量值刷新 f_locals 字典。
    if (co->co_nlocals)
        map_to_dict(map, j, locals, fast, 0);

    ... ...
}
```

也就是说，必须修改 f_localsplus 上的结果，否则是没戏的。在 CPython 源码 frameobject.c 中除了 PyFrame_FastToLocals 外，还有一个 PyFrame_LocalsToFast。作用正好相反，是把 f_locals 的内容重新写回 f_localsplus。我们直接调用 CPython API 试试。

```
In [1]: def test():
   ...:     x = 0
   ...:     def a():
   ...:         from ctypes import pythonapi, py_object
   ...:         f = sys._getframe(1)
   ...:         e_locals = f.f_locals
   ...:         e_locals["x"] = 100
   ...:         pythonapi.PyFrame_LocalsToFast(py_object(f), 0)
   ...:     a()
   ...:     print x
   ...:

In [4]: test()        # 哈哈！搞定！
100
```

当然，在 f_locals 中新增的名字是可以保留的，不会被 PyFrame_FastToLocals 刷掉。

```
In [4]: def test():
   ....:     def a():
   ....:         e_locals = sys._getframe(1).f_locals
   ....:         e_locals["s"] = "Hello, World!"
   ....:     a()
   ....:     print locals()["s"]
   ....:

In [5]: test()
Hello, World!
```

怎么不直接 "print s"？这是因为内部函数 a 里面创建新名字是动态行为，在编译时，这个 s 是不存在的，因此 "print s" 的结果就成下面这样了。

```
In [6]: def test():
   ....:     def a():
   ....:         e_locals = sys._getframe(1).f_locals
   ....:         e_locals["s"] = "Hello, World!"
   ....:     a()
   ....:     print s
   ....:

In [7]: test()
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
----> 6     print s
NameError: global name 's' is not defined

In [8]: dis.dis(test)
  6          16 LOAD_GLOBAL              0 (s)
             19 PRINT_ITEM
             20 PRINT_NEWLINE
```

不知道 Python 2.7.x 还能不能用上 nonlocal，难道真的要一直用 CPython API?

3.8.6 Sandbox

执行一个函数，同时要避免它对外部环境造成干扰。什么叫干扰？通常指的是修改了外部环境的数据，最简单的途径自然是 globals 名字空间。

```
In [1]: def test():
   ....:     global x
   ....:     x = 100
   ....:

In [2]: x = "秘密档案"

In [3]: test()

In [4]: x
Out[4]: 100
```

方法1：使用 exec 执行，提供自定义名字空间。

```
In [5]: x = "秘密档案"

In [6]: d = {}

In [7]: exec test.func_code in d

In [8]: print x                       # 老沙：哼哼！守住阵地！
秘密档案

In [9]: d["x"]                        # 小黑：你丫个骗子。
```

```
Out[9]: 100
```

方法2：创建一个新的函数对象，同样使用自定义名字空间。

```
In [10]: import types

In [11]: types.FunctionType?
Type:       type
Docstring:
function(code, globals[, name[, argdefs[, closure]]])

Create a function object from a code object and a dictionary.
The optional name string overrides the name from the code object.
The optional argdefs tuple specifies the default argument values.
The optional closure tuple supplies the bindings for free variables.

In [12]: d = {}

In [13]: t2 = types.FunctionType(test.func_code, d)

In [14]: t2()

In [15]: print x                    # 老沙：换个思路也不错啊。
秘密档案

In [16]: d["x"]                     # 小黑：老一套，有啥不同？
Out[16]: 100
```

对 test 函数来说，要突破这个关卡太容易了。

```
In [17]: def test():
   .....:     import sys
   .....:     f_globals = sys._getframe(1).f_globals
   .....:     f_globals["x"] = 100
   .....:

In [18]: x = "秘密档案"

In [19]: exec test.func_code in {}

In [20]: print x                    # 小黑：哈哈！被攻破了吧。
100
```

小样，我直接把内置函数给封掉。看丫得瑟。

```
In [110]: d = {"__builtins__": types.ModuleType("Anti-XXX")}

In [111]: x = "秘密档案"

In [112]: exec test.func_code in d   # 小黑：太过分了，不玩了！
---------------------------------------------------------------------------
ImportError                               Traceback (most recent call last)
```

```
      1 def test():
----> 2     import sys
      3     f_globals = sys._getframe(1).f_globals
      4     f_globals["x"] = 100
      5

ImportError: __import__ not found
```

我们伪造了一个 __builtins__ 模块，test 函数自然找不到 import 这个内置命令了。当然，在实际应用中，应该把正常使用的内置函数和内置类型复制进来。或者全部复制，然后删除掉危险的函数，比如 import、reload 等等。

其他可选方案：

- 用 os.fork 新建进程彻底隔离。
- 用 AST 分析函数源码，检查危险的指令。(并不靠谱，在第一章我们就用过代码混淆了)
- 其他……

这个沙箱演示，只是个原始版本，大家可以继续发挥。要是某论坛搞个攻防比赛，貌似也挺好玩的啊。别看 Python 是 "脚本语言"，一样可以搞加密和破解玩。

3.9 小结

这章不太好写，与虚拟机核心相关的东西太多，涉及的概念又稍微有点深，一不小心就跑得没边没际了。最麻烦的是如何安排章节顺序，相关概念相互依存，相互纠结得厉害，实在有点顾头不顾尾的感觉。幸好多次调整之后，慢慢理出个稍微满意的次序。

对本书的定位而言，深浅把握实在有点难。在发布了两个预览版之后，已经有兄弟反应看不懂，昏昏欲睡。对此，我有些无语。一来，这书的确不适合初学者，哪怕是认真学过 Python，却没有系统学习计算机基础的同学也不适合。因为诸多的概念远不是一门编程语言所能通透的。其次，市面上入门书已经很多。虽说 Python 小众了些，但相关书籍并不缺。何况入门好书一本足矣，其他全都是代码上的功夫，不如参照官方手册来得直接。

多数人并不喜欢研究源码，尤其是 CPython 这类看着繁复无比，在日常工作时又 "用不到" 的底层代码。他们只希望获得一个规则，一个模式，然后照本宣科去完成某些工作。如果是这样，我个人建议换本书。因为不读源码，不看作者辛苦添加的注释，这书毫无价值。

有些啰嗦了，不过不是抱怨，而是作者的大实话。我只是想写一本记录探索过程，能将理论和实现结合起来的书。这书做不得教程，兴许也就满足那么一点好奇心。

Q.yuhen：原定的第三章《表达式》被暂时删除，要控制全书的篇幅。