

Python 源码剖析

——编译 Python

本文作者: Robert Chen(search.pythoner@gmail.com)

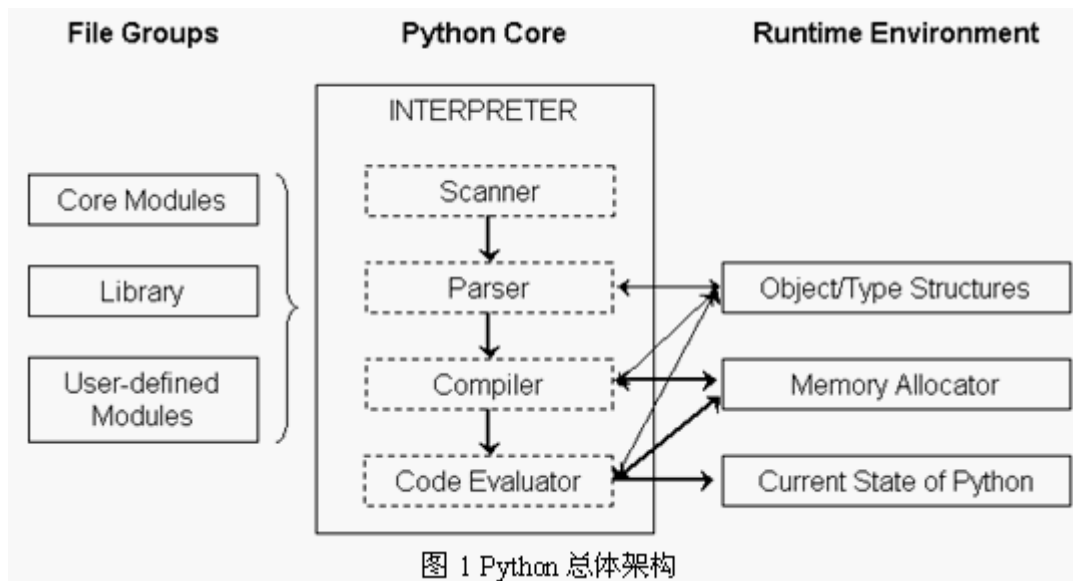
1. Python 总体架构

在最高的层次上, Python 的整体架构可以分为四个主要的部分, 整个架构如图 1 所示。在左边, 是 Python 提供的大量的模块, 库以及用户自定义的模块。比如在执行 `import os` 时, 这个 `os` 就是 Python 内建的模块, 当然用户还可以通过自定义模块来扩展 Python 系统。在本系列文章中, 我们不会对这一部分进行过多的考察。

在图的右边, 是 Python 的运行时环境, 包括对象/类型系统 (Object/Type structures), 内存分配器 (Memory Allocator) 和运行时状态 (Current State of Python)。运行时状态维护了解释器在执行字节码时在不同的状态之间切换的动作, 我们可以将它视为一个巨大而复杂的有穷状态机。内存分配器则全权负责 Python 中创建对象时对内存的申请工作, 实际上它就是 Python 运行时与 C 中 `malloc` 的一层接口。而对象/类型系统则包含了 Python 中存在的各种内建对象, 比如整数, `list` 和 `dict` 等等

在中间的部分, 可以看到 Python 的核心, 解释器 (interpreter)。在解释器中, 箭头的方向指示了 Python 运行时的数据流方向。其中 Scanner 对应词法分析, 将文件输入的 Python 源代码或从命令行输入的一行行 Python 代码切分为一个一个的 token; Parser 对应语法分析部分, 在 Scanner 的分析结果上进行语法分析, 建立抽象语法树 (AST); Compiler 是根据建立的 AST 生成指令集合——Python 字节码 (byte code), 就像 Java 编译器和 C# 编译器所做的那样; 最后由 Code Evaluator 来解释并执行这些字节码。因此, Code Evaluator 又可以被称为执行引擎。

图中, 在 Interpreter 与右边的对象/类型系统, 内存分配器之间的箭头表示“使用”关系; 而与运行时状态之间的箭头表示修改关系, 即 Python 在执行的过程中会不断地修改当前解释器所处的状态, 在不同的状态之间切换。



2. Python 源代码的组织

中国有句老话，巧妇难为无米之炊。要分析 Python 源码，首先当然要获得 Python 源码。Python 源码可以从 Python 的官方网站 <http://www.python.org> 自由下载。当前 Python 的最新版本是 2.4.2，在本书中，我采用的是 Python 2.4.1:

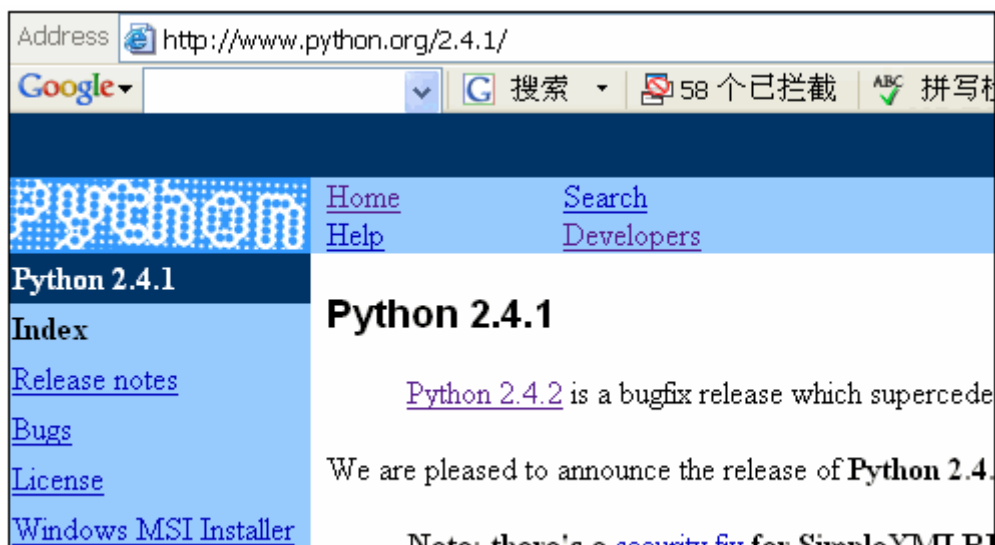


图 2 下载 Python 2.4.1 源码

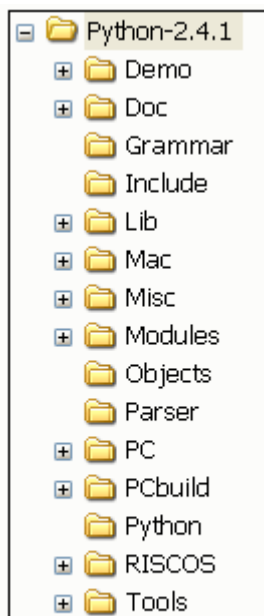


图 3

Python 目录结构

下载了 Python 的源代码压缩包并解压后，可以看到如图 3 所示的目录结构。

Include：该目录下包含了 Python 提供的所有头文件，如果用户需要自己用 C 或 C++ 来编写自定义模块扩展 Python，那么就需要用到这里提供的头文件。

Lib：该目录包含了 Python 自带的所有标准库，Lib 中的库都是用 Python 语言编写的。

Modules：该文件夹中包含了所有用 C 语言编写的模块，比如 random, cStringIO 等，Modules 中的模块是那些对速度要求非常严格的模块。而有一些对速度没有太严格要求的模块，比如 os，就是用 Python 编写，并且放在 Lib 目录下。

Parser：Parser 目录中包含了 Python 解释器中的 Scanner 和 Parser 部分，即对 Python 源代码进行词法分析和语法分析的部分。除了这些，Parser 目录下还包含了一些有用的工具，这些工具能够根据 Python 语言的语法自动生成 Python 语言的词法和语法分析器，与 YACC 非常类似。

Objects：该目录中包含了所有 Python 的内建对象，包括整数，list, dict 等；同时，该目录还包括了 Python 在运行时需要的所有的内部使用对象的实现

Python：该目录下包含了 Python 解释器中的 Compiler 和执行引擎部分，是 Python 运行的核心所在。

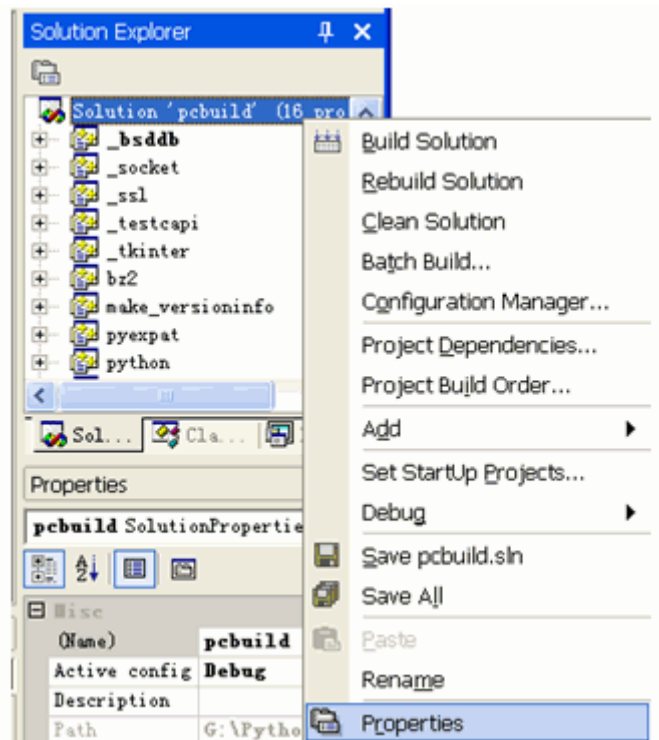
PCBuild：包含了 Visual Studio 2003 工程文件，研究 Python 源代码就从这里开始。

3. 编译 Python

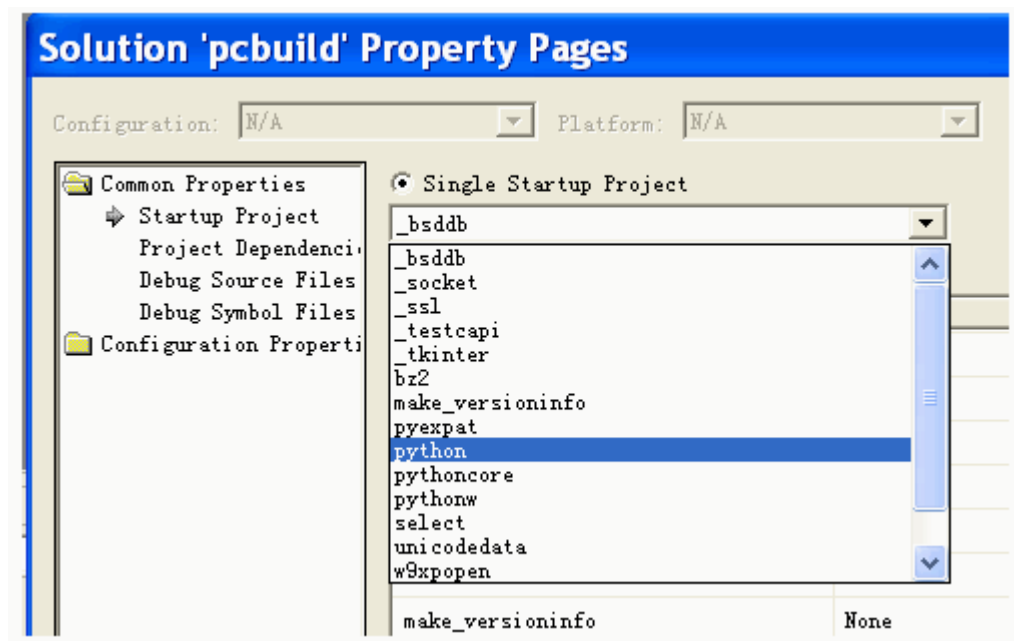
好了，下载了 Python 的源代码之后，我们就可以走出剖析 Python 源码的第一步——编译 Python——了：)

Python2.4.1 是在 Visual Studio 2003 环境下开发的，在 PCBuild 目录下可以看到 VS2003 的工程文件，打开工程后，还需要进行一些设置，才能成功编译。

首先，我们需要激活 VS2003 的配置对话框：



在配置对话框中，首先要做的就是更改 Startup Project，Python2.4.1 中默认设置的是 bsddb，我们需要将其改为 Python。

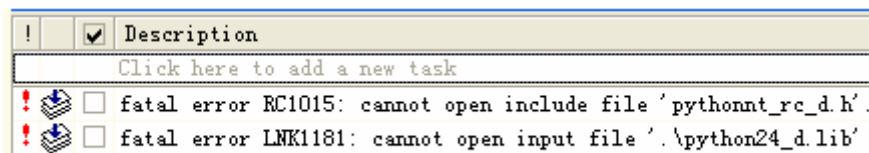


由于我们剖析的只是 Python 的核心部分，不会涉及到工程中的一些标准库和其他的模块，所以我们需要将它们从编译的列表中删除。点击配置对话框左边列表框中的“Configuration Properties”后，会出现当前配置为需要编译的子工程，取消多余的子工程的选中状态，只保留 pythoncore 和 python 的选中状态。

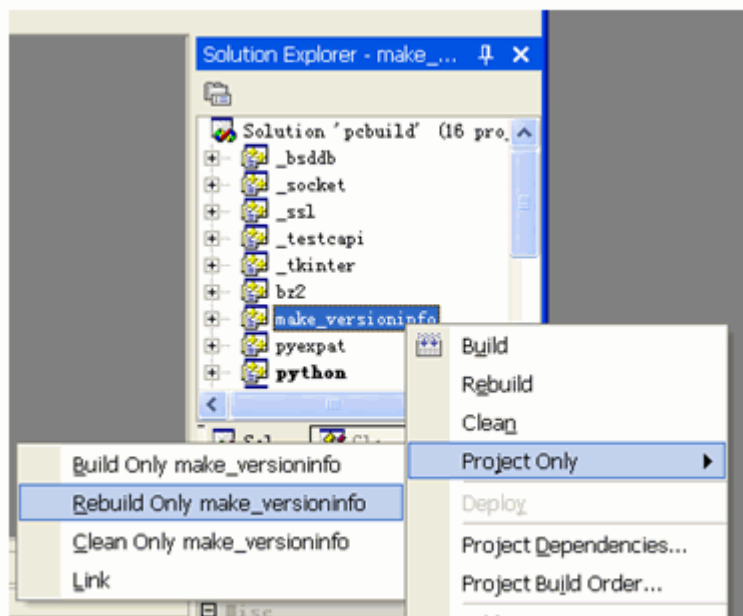
Project Contexts (check the project configurations to build or deploy)

Project	Configuration	Platform	Build
_bsddb	Debug	Win32	<input checked="" type="checkbox"/>
_socket	Debug	Win32	<input checked="" type="checkbox"/>
_ssl	Debug	Win32	<input checked="" type="checkbox"/>
_testcapi	Debug	Win32	<input checked="" type="checkbox"/>
_tkinter	Debug	Win32	<input checked="" type="checkbox"/>
bz2	Debug	Win32	<input checked="" type="checkbox"/>
make_versioninfo	Debug	Win32	<input checked="" type="checkbox"/>
pyexpat	Debug	Win32	<input checked="" type="checkbox"/>
python	Debug	Win32	<input checked="" type="checkbox"/>
pythoncore	Debug	Win32	<input checked="" type="checkbox"/>
pythonw	Debug	Win32	<input checked="" type="checkbox"/>

需要进行的改动就是这么多了，但是完成这些改动后，如果马上开始编译，那么编译还是会失败：



原因是我们还需要一个 pythonnt_rc_d.h，这个文件在 Python2.4.1 的源码包中没有提供，必须要通过一个编译 make_versioninfo 子工程才能自动生成：



好了，现在再编译，一切都会顺利完成了。

Python 源码剖析

——对象机制

本文作者: Robert Chen(search.pythoner@gmail.com)

1. 对象

在 Python 的世界中，一切都是对象，一个整数是一个对象，一个字符串也是一个对象，更为奇妙的是，类型也是一个对象，整数类型是一个对象，字符串类型也是一个对象。从 1980 年 Guido 在那个圣诞节揭开 Python 世界的大幕开始，一直到现在，Python 经历了一次一次的升级，但是其实现语言一直都是 ANSI C。我们知道，C 并不是一个面向对象的语言，那么在 Python 中，它的对象机制是如何实现的呢？

对于人的思维来说，对象是一个比较形象的概念，而对于计算机来说，对象实际上是一个抽象的概念。计算机并不能理解这是一个整数，那是一个字符串，对于计算机来说，它所知道的一切都是字节。通常的说法是，对象是数据以及基于这些数据的操作的集合。在计算机上，一个对象实际上就是一片被分配的内存空间，这些内存可能是连续的，也有可能是离散的，这都不重要，重要的是这片内存存在更高的层次上可以作为一个整体来考虑，这个整体就是一个对象。在这片内存中，存储着一系列的数据以及可以对这些数据进行修改或读取的一系列操作的代码。

在 Python 中，对象就是在堆上申请的结构体，对象不能是被静态初始化的，并且也不能是在栈空间上生存的。唯一的例外就是类型对象(type object)，Python 中所有的类型对象都是被静态初始化的。

在 Python 中，一个对象一旦被创建，它在内存中的大小就是不变的了。这就意味着那些需要容纳可变长度数据的对象只能在对象内维护一个指向一个可变大小的内存区域的指针。为什么要设定这样一条特殊的规则呢，因为遵循这样的规则可以使通过指针维护对象的工作变得非常的简单。因为一旦允许对象的大小可在运行期改变，我们可以考虑如下的情形。在内存中有对象 A，并且其后紧跟着对象 B。如果运行期某个时刻，A 的大小增大了，这意味着必须将整个 A 移动到内存中的其他位置，否则 A 增大的部分将覆盖原本属于 B 的数据。一旦将 A 移动到内存中的其他位置，那么所有指向 A 的指针必须立即得到更新，光是想一想，就知道这样的工作是多么的恐怖。

在 Python 中，所有的东西都是对象，而所有的对象都拥有一些相同的内容，这些内容在 PyObject 中定义，PyObject 是整个 Python 对象机制的核心。

```
[object.h]
typedef struct _object {
    PyObject_HEAD
} PyObject;
```

实际上, PyObject 是 Python 中不包含可变长度数据的对象的基石, 而对于包含可变长度数据的对象, 它的基石是 PyVarObject:

```
[object.h]
typedef struct {
    PyObject_VAR_HEAD
} PyVarObject;
```

这两个结构体构成了 Python 对象机制的核心基石, 从代码中我们可以看到, Python 的对象的秘密都隐藏在 PyObject_HEAD 与 PyObject_VAR_HEAD 中。

```
[object.h]
#ifdef Py_TRACE_REFS
/* Define pointers to support a doubly-linked list of all live heap
objects. */
#define _PyObject_HEAD_EXTRA \
    struct _object *_ob_next; \
    struct _object *_ob_prev;

#define _PyObject_EXTRA_INIT 0, 0,

#else
#define _PyObject_HEAD_EXTRA
#define _PyObject_EXTRA_INIT
#endif

/* PyObject_HEAD defines the initial segment of every PyObject. */
#define PyObject_HEAD \
    _PyObject_HEAD_EXTRA \
    int ob_refcnt; \
    struct _typeobject *ob_type;

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

在 PyObject_HEAD 中定义了每一个 Python 对象都必须有的内容, 这些内容将出现在每一个 Python 对象所占有的内存的最开始的字节中, 从 PyObject_VAR_HEAD 的定义可以看出, 即使对于拥有可变大小数据的对象, 其最开始的字节也含有相同的内容, 这就是说, 在 Python 中, 每一个对象都拥有

相同的对象头部。这就使得在 Python 中，对对象的引用变得非常的统一，我们只需要用一个 PyObject *就可以引用任意的一个对象，而不论该对象实际是一个什么对象。

在 PyObject_HEAD 的定义中，我们注意到有一个 ob_refcnt 的整形变量，这个变量的作用是实现引用计数机制。对于某一个对象 A，当有一个新的 PyObject *引用该对象时，A 的引用计数应该增加；而当这个 PyObject *被删除时，A 的引用计数应该减少。当 A 的引用计数减少到 0 时，A 就可以从堆上被删除，以释放出内存供别的对象使用。

在 PyObject_HEAD 中，我们注意到 ob_type 是一个指向_typeobject 结构体的指针，那么这个结构体是一个什么东西呢？实际上这个结构体也是一个对象，它是用来指定一个对象类型的类型对象。这个类型对象我们将在后边详细地考察。现在我们看到了，在 Python 中实际上对象机制的核心非常的简单，一个是引用计数，一个就是类型。

而对于拥有可变长度数据的对象，这样的对象通常都是容器，我们可以在 PyObject_VAR_HEAD 中看到 ob_size 这个变量，这个变量实际上就是指明了该对象中一共包含了多少个元素。注意，ob_size 指明的是元素的个数，而不是字节的数目。比如对于 Python 中最常用的 list，它就是一个 PyVarObject 对象，如果某一时刻，这个 list 中有 5 个元素，那么 PyVarObject.ob_size 的值就是 5。

2. 类型对象

在上面的描述中，我们看到了 Python 中所有对象的对象头的定义。所以，当内存中存在某一个 Python 的对象时，该对象的开始的几个字节的含义一定会符合我们的预期。但是，当我们把眼光沿着时间轴上溯，就会发现一个问题。当在内存中分配空间，创建对象的时候，毫无疑问地，必须要知道申请多大的空间。显然，这不会是一个定值，因为对于不同的对象，需要不同的空间，一个整数对象和一个字符串对象所需的空间肯定不同。那么，对象所需的内存空间的大小的信息到底在哪里呢？在对象头中显然没有这样的信息。

实际上，内存空间大小这样的对象的元信息是与对象所属类型密切相关的，因此它一定会出现在与对象所对应的类型对象中。现在我们可以来详细考察一下类型对象_typeobject:

```
[object.h]
typedef struct _typeobject {
    PyObject_VAR_HEAD
    char *tp_name; /* For printing, in format "<module>.<name>" */
    int tp_basicsize, tp_itemsize; /* For allocation */

    /* Methods to implement standard operations */
    destructor tp_dealloc;
    printfunc tp_print;
    .....
    /* More standard operations (here for binary compatibility) */
```



```

hashfunc tp_hash;
ternaryfunc tp_call;
.....
} PyTypeObject;

```

在 `_typeobject` 的定义中包含了许多信息，主要可以分为四类：

1. 类型名，`tp_name`，主要是 Python 内部以及调试的时候使用；
2. 创建该类型对象是分配内存空间的大小的信息，即 `tp_basicsize` 和 `tp_itemsize`；
3. 与该类型对象相关联的操作信息，比如 `hashfunc`，`tp_hash` 就指明对于该类型的对象，如何生成其 `hash` 值。在 `Object.h` 中可以看到，`hashfunc` 实际上是一个函数指针：`typedef long (*hashfunc)(PyObject *)`；在 `_typeobject` 中，包含了大量的函数指针，这些函数指针将用来指定某个类型的操作信息。这些操作主要分为标准操作(`dealloc`, `print`, `compare`)，标准操作族(`numbers`, `sequences`, `mappings`)，以及其他操作 (`hash`, `buffer`, `call`...)。
4. 我们在下边将要描述的类型的信息。

有趣的是我们在 `_typeobject` 的头部发现了 `PyObject_VAR_HEAD`，这意味着类型实际上也是一个对象。我们知道在 Python 中，每一个对象都是对应一种类型的，那么一个有趣的问题就出现了，类型对象的类型是什么呢？这个问题听上去很绕口，实际上确非常重要，对于其他的对象，可以通过与其关联的类型对象确定其类型，那么通过什么来确定一个对象是类型对象呢？答案就是

`PyType_Type`:

```

[typeobject.c]
PyTypeObject PyType_Type = {
    PyObject_HEAD_INIT(&PyType_Type)
    0,                          /* ob_size */
    "type",                      /* tp_name */
    sizeof(PyHeapTypeObject),    /* tp_basicsize */
    sizeof(PyMemberDef),         /* tp_itemsize */
    .....,
    PyObject_GC_Del,            /* tp_free */
    (inquiry)type_is_gc,        /* tp_is_gc */
};

```

前面提到，在 Python 中，每一个对象它的开始部分都是一样的。每一个对象都将自己的引用计数，类型信息保存在开始的部分中。为了方便对这部分的初始化，Python 中提供了几个有用的宏：

```

[object.h]
#ifdef Py_TRACE_REFS
#define _PyObject_EXTRA_INIT 0, 0,
#else
#define _PyObject_EXTRA_INIT

```

```
#endif

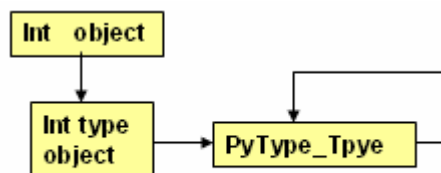
#define PyObject_HEAD_INIT(type)    \
    _PyObject_EXTRA_INIT            \
    1, type,
```

再回顾一下 PyObject 和 PyVarObject 的定义，初始化的动作就一目了然了。实际上，这些宏在类型对象的初始化中被大量地使用着。

如果以一个整数对象为例，可以更清晰地看到一半的类型对象和这个特立独行的 PyType_Type 对象之间的关系：

```
[intobject.c]
PyTypeObject PyInt_Type = {
    PyObject_HEAD_INIT(&PyType_Type)
    0,
    "int",
    sizeof(PyIntObject),
    .....
};
```

现在我们可以放飞想象，看到一个整数对象在运行时的抽象的表示了，下图中的箭头表示 ob_type：



图一、运行时对象类型关系图

3. 对象间的继承和多态

通过 PyObject 和类型对象，Python 利用 C 语言完成了 C++所提供的继承和多态的特性。前面提到，在 Python 中所有的内建对象（PyIntObject 等）和内部使用对象（PyCodeObject 等）的最开始的内存区域都拥有一个 PyObject。实际上，这一点可以视为 PyIntObject，PyCodeObject 等对象都是从 PyObject 继承而来。

在 Python 创建一个对象，比如 PyIntObject 对象时，会分配内存，进行初始化。然后这个对象会由一个 PyObject*变量来维护，而不是通过一个 PyIntObject*指针来维护。其它对象也与此类似，所以在 Python 内部各个函数之间传递的都

是一种范型指针 `PyObject*`。这个指针所指的对象究竟是什么类型的，不知道，只能从指针所指对象的 `ob_type` 域判断，而正是通过这个域，Python 实现了多态机制。

考虑下面的代码：

```
void Print(PyObject* object)
{
    object->ob_type->tp_print(object);
}
```

如果传给 `Print` 的指针实际是一个 `PyIntObject*`，那么就会调用 `PyIntObject` 对象对应的类型对象中定义的输出操作，如果传给 `Print` 的指针实际是一个 `PyStringObject*`，那么就会调用 `PyStringObject` 对象对应的类型对象中定义的输出操作。可以看到，这里同一个函数在不同情况下表现出了不同的行为，这正是多态的核心所在。

在 `object.c` 中，Python 实现了一些对于类型对象中的各种操作的简单包装，从而为 Python 运行时提供了一个统一的多态接口层：

```
[object.c]
long PyObject_Hash(PyObject *v)
{
    PyTypeObject *tp = v->ob_type;
    if (tp->tp_hash != NULL)
        return (*tp->tp_hash)(v);
    .....
}
```

4. 引用计数

在 C 或 C++ 中，程序员被赋予了极大的自由，可以任意地申请内存。但是权利的另一面则对应着责任，程序员必须自己负责将申请的内存释放，并释放无效指针。可以说，这一点正是万恶之源，大量的内存泄露和悬空指针的 bug 由此而生，如黄河泛滥一发不可收拾：)

现代的开发语言中一般都选择由语言本身负责内存的管理和维护，即采用了垃圾收集机制，比如 Java 和 C#。垃圾收集机制使开发人员从维护内存分配和清理的繁重工作中解放出来，但同时也剥夺了程序员与内存亲密接触的机会，并付出了一定的运行时效率作为代价。现在看来，随着垃圾收集机制的完善，对时间要求不是非常高的程序完全可以通过使用垃圾收集机制的语言来完成，这部分程序占了这个地球上大多数的程序。这样做的好处是提高了开发效率，并降低了 bug 发生的机率。Python 同样也内建了垃圾收集机制，代替程序员进行繁重的内存管理工作，而引用计数正式 Python 垃圾收集机制的一部分。

Python 通过对一个对象的引用计数的管理来维护对象在内存中的生存。我们知道在 Python 中每一个东西都是一个对象，都有一个 `ob_refcnt` 变量，正是这个变量维护着该对象的引用计数，从而也最终决定着该对象的生生灭灭。

在 Python 中，主要是通过 `Py_INCREF(op)` 和 `Py_DECREF(op)` 两个宏来增加和减少一个对象的引用计数。当一个对象的引用计数减少到 0 之后，`Py_DECREF`

将调用该对象的析构函数（`dealloc` function）来释放该对象所占有的内存和系统资源。注意这里的析构函数借用了 C++ 的词汇，实际上这个析构动作是通过在对象对应的类型对象中定义的一个函数指针来刻画的，还记得吗？就是那个 `tp_dealloc`。

如果熟悉设计模式中 Observer 模式，可以看到，这里隐隐约约透着 Observer 模式的影子。在 `ob_refcnt` 减为 0 之后，将触发对象销毁的事件；从 Python 的对象体系来看，各个对象又提供了不同的事件处理函数，而事件的注册动作正是在各个对象对应的类型对象中静态完成的。

对于这两个宏的参数 `op` 来说，不允许 `op` 是一个指向空对象的指针(NIL)，如果 `op` 是一个 NIL，那么必须使用 `Py_XINCREF/Py_XDECREF` 这一对宏。

在 `PyObject` 中我们看到 `ob_refcnt` 是一个 32 位的整形变量，这实际是一个 Python 所做的假设，即对一个对象的引用不会超过一个整形变量的最大值。一般情况下，如果不是恶意代码，这个假设显然是不会被突破的。

需要注意的是，在 Python 的各种对象中，类型对象是超越引用计数规则的。类型对象“跳出三界外，不再五行中”，永远不会被析构。每一个对象中指向类型对象的指针不被视为对类型对象的引用。

在每一个对象创建的时候，Python 提供了一个 `_Py_NewReference(op)` 宏来将对象的引用计数初始化为 1。

在 Python 的源代码中可以看到，在不同的编译选项下(`Py_REF_DEBUG`, `Py_TRACE_REFS`)，引用计数的宏还要做许多额外的工作。下面展示的代码是 Python 在最终发行时这些宏所对应的实际的代码：

```
[object.h]
/* Without Py_TRACE_REFS, there's little enough to do that
we expand code
 * inline.
 */
#define _Py_NewReference(op) ((op)->ob_refcnt = 1)
```

```
#define _Py_Dealloc(op)
((*(op)->ob_type->tp_dealloc)((PyObject *) (op)))
```

```
#define Py_INCREF(op) ((op)->ob_refcnt++)
```

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

```
/* Macros to use in case the object pointer may be NULL: */
```

```
#define Py_XINCRF(op) if ((op) == NULL) ; else Py_INCRF(op)
#define Py_XDECRF(op) if ((op) == NULL) ; else Py_DECRF(op)
```

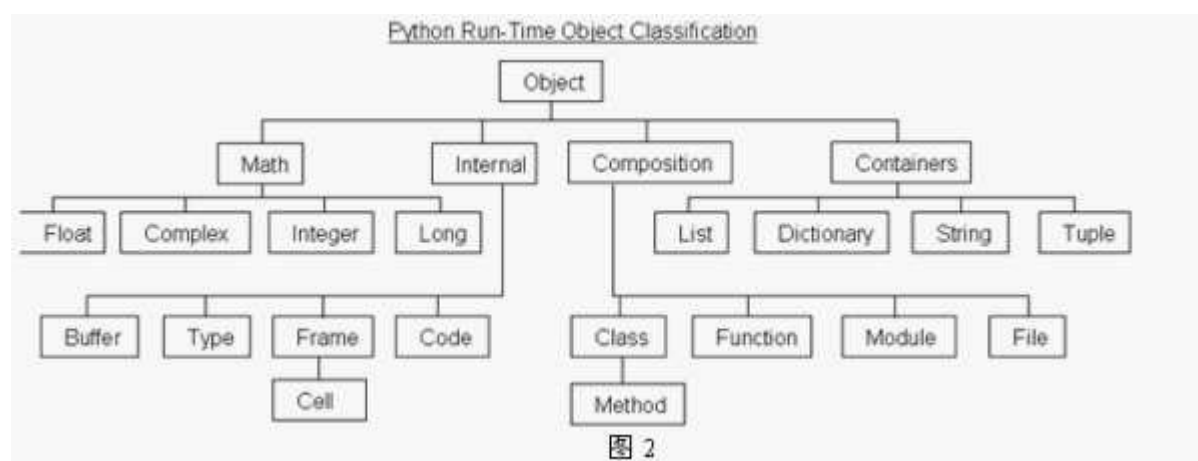
在一个对象的引用计数减为 0 时，与该对象对应的析构函数就会被调用，但是要特别注意的是，调用析构函数并不意味着最终一定会调用 free 释放内存空间，如果真是这样的话，那频繁地申请、释放内存空间会使 Python 的执行效率大打折扣（更何况 Python 已经多年背负了人们对其执行效率的指责：）。一般来说，Python 中大量采用了内存对象池的技术，使用这种技术避免频繁地申请和释放内存空间。因此在析构时，通常都是将对象占用的空间归还到内存池中。这一点在接下来对 Python 内建对象的实现中可以看得一清二楚。

5. Python 对象的分类

我们将 Python 的对象从概念上大致分为四类，需要指出的是，这种分类并不一定完全正确，不过是提供一种看待 Python 中对象的视角而已：

- Math：数值对象
- Container：容纳其他对象的集合对象
- Composition：表示程序结构的对象
- Internal：Python 解释器在运行时内部使用的对象

图 2 列出了我们的对象分类体系，并给出了每一个类别中的一些实例：



6. 通向 Python 之路

对 Python 源码的剖析将分为四部分。

1. 静态对象剖析：首先我们会分析静态的对象，Math 对象和 Container 对象，深刻理解这些对象对我们理解 Python 解释器的运行会有很大的帮助，同时，对我们编写 Python 代码也将大有裨益，在编写 Python 代码时，你会清晰地意识到系统内部这些对象将如何运作，变化。当然，我

们并不会分析所有的 Python 对象，而是选取使用最频繁的四钟对象：
`PyIntObject`, `PyStringObject`, `PyListObject`, `PyDictObject` 进行剖析。

2. 运行时剖析：在分析完静态的对象之后，我们将进入 Python 解释器，在这里我们会详细地考察 Python 的字节码(byte code)以及解释器对字节码的解释和执行过程。这部分将完整地展现 Python 中所有的语法结构，如一般表达式，控制流，异常流，函数，类等等的字节码层面的实现细节。同时，在这部分，我们会考察大部分的 Python 内部对象。
3. 编译期剖析：这部分没什么好打广告的了，目标明确，对象清晰，但是难度呢，绝不简单：)
4. 运行环境剖析：这部分将考察从激活 Python 到 Python 准备就绪，可以接受用户输入或者执行脚本文件，这段时间内，Python 如何建立自己的运行环境，并建立了怎样的运行环境，呵呵透露一下，想想 Python 那个庞大的 builtin 函数集合，这些就是这部分考察的重点。

阅读完这些内容之后，对于 Python，你应该是了如指掌了，在以后编写 Python 代码时，你的脑子里甚至可以出现 Python 解释器将如何一步步解释你的代码的情形。当然，这只是我写作本书的副产品。这本书诞生的真正原因只有一个，兴趣，我对 Python 的实现有浓厚的兴趣。这本书也只是第一步，希望以后还能继续对 Python 系列，如 IronPython、Jython，PyPy 的探索，当然，对于其他动态语言，比如 Ruby 的探索，我希望也会有时间去做。如果你对动态语言的实现有兴趣，你一定会喜欢本书；如果你还没有兴趣，希望它能唤起你的兴趣：)

作为 Python 中最简单的对象，整数对象是研究 Python 对象体系的一个非常好的切入点。直观上会认为整数对象的实现非常简单，如果单纯以整数对象而言，实现确实非常简单。然而在 Python 中，为了运行效率，实际上存在着一个以缓冲池为核心的整数对象的体系结构，实际上，Python 各种对象几乎都拥有这样一个以缓冲池为核心的体系结构，理解这一点对 Python 运行时行为的了解有重要的意义。对这种结构的深入挖掘也是本章的重点所在。

本章分为三个部分：

1. 研究 Python 中的整数对象：`PyIntObject`
2. 通过研究 `PyIntObject` 的创建和维护，深入挖掘整数对象体系结构
3. Hack `PyIntObject`：通过修改 Python 源代码，对第一第二部分的知识加深了解

本文是第一部分。

Python 源码剖析

——整数对象 `PyIntObject`(1)

本文作者: Robert Chen (pythoner.search@gmail.com)

1 `PyIntObject`


```

0,          /* tp_traverse */
0,          /* tp_clear */
0,          /* tp_richcompare */
0,          /* tp_weaklistoffset */
0,          /* tp_iter */
0,          /* tp_iternext */
int_methods, /* tp_methods */
0,          /* tp_members */
0,          /* tp_getset */
0,          /* tp_base */
0,          /* tp_dict */
0,          /* tp_descr_get */
0,          /* tp_descr_set */
0,          /* tp_dictoffset */
0,          /* tp_init */
0,          /* tp_alloc */
int_new,    /* tp_new */
(freefunc)int_free, /* tp_free */
};

```

可以看到，在 `PyInt_Type` 中保存着 `PyIntObject` 的元信息，其中有 `PyIntObject` 对象的大小，`PyIntObject` 的文档信息，更多的是 `PyIntObject` 所支持的操作。在下面的列表中，列出了 `PyIntObject` 所支持的操作：

<code>int_dealloc</code>	删除 <code>PyIntObject</code> 对象
<code>int_free</code>	删除 <code>PyIntObject</code> 对象
<code>int_repr</code>	转化成 <code>PyString</code> 对象
<code>int_hash</code>	获得 HASH 值
<code>int_print</code>	打印 <code>PyIntObject</code> 对象
<code>int_compare</code>	比较操作
<code>int_as_number</code>	数值操作
<code>int_methods</code>	成员函数

我们可以看一看比较操作的代码：

```

[intobject.c]
static int int_compare(PyIntObject *v, PyIntObject *w)
{
    register long i = v->ob_ival;
    register long j = w->ob_ival;
    return (i < j) ? -1 : (i > j) ? 1 : 0;
}

```

可以看到,PyIntObject对象实际上就是简单地将它所包装的 long 值进行比较。

需要特别注意的是 int_as_number 这个域,实际上它是一个 PyNumberMethods 结构体:

```
[intobject.c]
static PyNumberMethods int_as_number = {
    (binaryfunc)int_add,      /*nb_add*/
    (binaryfunc)int_sub,      /*nb_subtract*/
    (binaryfunc)int_mul,      /*nb_multiply*/
    .....,
    (binaryfunc)int_div,      /* nb_floor_divide */
    int_true_divide,          /* nb_true_divide */
    0,                        /* nb_inplace_floor_divide */
    0,                        /* nb_inplace_true_divide */
};
```

在 object.h 中,可以找到 PyNumberMethods 的定义。在 Python2.4 中,PyNumberMethods 中一共有 38 个函数指针,也就是说在其中定义了 38 种操作的信息,这些都是作为一个数值(Number)类型的对象可以向世界提供的操作,比如,加法,减法,乘法,模运算等等。

在 int_as_number 中,确定了对于一个整数对象,这些数值操作应该如何进行。当然,在 PyNumberMethods 的 38 中数值操作中,并非所有的操作都要求一定要被实现,在 int_as_number 中我们就可以看到,有相当多的操作是没有实现的。我们可以看一下 PyIntObject 中加法操作是如何实现的:

```
[intobject.h]
/* Macro, trading safety for speed */
#define PyInt_AS_LONG(op) (((PyIntObject *) (op))->ob_ival)
```

```
[intobject.c]
#define CONVERT_TO_LONG(obj, lng) \
    if (PyInt_Check(obj)) { \
        lng = PyInt_AS_LONG(obj); \
    } \
    else { \
        Py_INCREF(Py_NotImplemented); \
        return Py_NotImplemented; \
    }
```

```
static PyObject *
int_add(PyIntObject *v, PyIntObject *w)
{
    register long a, b, x;
```

```

    CONVERT_TO_LONG(v, a);
    CONVERT_TO_LONG(w, b);
    x = a + b;
    if ((x^a) >= 0 || (x^b) >= 0)
        return PyInt_FromLong(x);
    return PyLong_Type.tp_as_number->nb_add((PyObject *)v, (PyObject
*)w);
}

```

如你所想，PyIntObject 实现的加法操作是直接在其维护的 long 值上进行的，可以看到，在完成了加法操作后，还进行了溢出的检查，如果没有溢出，就返回一个新的 PyIntObject，这个 PyIntObject 所拥有的值正好是加法操作的结果。这里清晰地显示了，PyIntObject 是一个 Immutable 的对象，因为在操作完成之后，原来参与操作的任何一个对象都没有发生改变，取而代之的，一个全新的对象诞生了。而如果加法的结果有溢出，那么结果就再不是一个 PyIntObject 对象，而是一个 PyLongObject 了，如图 1 所示：

```

C:\Documents and Settings\Administrator>python
Python 2.4 (#60, Nov 30 2004, 11:49:19) [MSC v.1310 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> a = 0x7fffffff
>>> type(a)
<type 'int'>
>>> c = a+a
>>> type(c)
<type 'long'>
>>>

```

图 1

在 PyInt_Type 中的 int_doc 域中维护着 PyIntObject 的文档信息，你可以在 Python 的交互环境下通过下列命令看到这段文档，如图 2 所示：

```

C:\WINDOWS\system32\cmd.exe - python
Python 2.4 (#60, Nov 30 2004, 11:49:19) [MSC v.1310 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> a = 1
>>> print a.__doc__
int(x[, base]) -> integer

Convert a string or number to an integer, if possible.  A floating point
argument will be truncated towards zero (this does not include a string
representation of a floating point number!)  When converting a string, use
the optional base.  It is an error to supply a base when converting a
non-string.  If the argument is outside the integer range a long object
will be returned instead.
>>>

```

图 2

[python.h]

```

/* Define macros for inline documentation. */
#define PyDoc_VAR(name) static char name[]
#define PyDoc_STRVAR(name,str) PyDoc_VAR(name) = PyDoc_STR(str)
#ifdef WITH_DOC_STRINGS
#define PyDoc_STR(str) str
#else
#define PyDoc_STR(str) ""
#endif

[intobject.c]
PyDoc_STRVAR(int_doc,
"int(x[, base]) -> integer\n\
\n\
Convert a string or number to an integer, if possible. A floating
point\n\
argument will be truncated towards zero (this does not include a
string\n\
representation of a floating point number!) When converting a string,
use\n\
the optional base. It is an error to supply a base when converting
a\n\
non-string. If the argument is outside the integer range a long
object\n\
will be returned instead.");

```

Python 源码剖析

——整数对象 PyIntObject(2)

本文作者: Robert Chen (search.pythoner@gmail.com)

2 PyIntObject 对象的创建和维护

2.1 对象创建的三种途径

在 intobject.h 中可以看到，可以从三种途径获得一个 PyIntObject 对象：

```

PyObject *PyInt_FromLong(long ival)
PyObject* PyInt_FromString(char *s, char **pend, int base)
#ifdef Py_USING_UNICODE
PyObject*PyInt_FromUnicode(Py_UNICODE *s, int length, int base)
#endif

```

分别是从 long 值，从字符串以及 Py_UNICODE 对象生成 PyIntObject 对象。在这里我们只考察从 long 值以及字符串生成 PyIntObject 对象。因为 PyInt_FromString 实际上是先将字符串转换成浮点数，然后再调用 PyInt_FromFloat:

```
[intobject.c]
PyObject* PyInt_FromString(char *s, char **pend, int base)
{
    char *end;
    long x;
    . . . . .
    //convert string to long
    if (base == 0 && s[0] == '0')
    {
        x = (long) PyOS_strtoul(s, &end, base);
    }
    else
        x = PyOS_strtol(s, &end, base);
    . . . . .
    return PyInt_FromLong(x);
}
```

为了理解整数对象的创建过程，必须要深入了解 Python 中整数对象在内存中的组织方式，实际上，一个个的整数对象在内存中并不是独立存在，单兵作战的，而是形成了一个整数对象体系。我们首先就重点考察一下 Python 中的整数对象体系结构。

2.2 小整数对象

在实际的编程中，对于数值比较小的整数，比如 1, 2, 29 等等，可能在程序中会非常频繁地使用。想一想 C 语言中的 for 循环，就可以了解这些小整数会有多么频繁的使用场合。在 Python 中，所有的对象都是存活在系统堆上。这就是说，如果没有特殊机制，对于这些频繁使用的小整数对象，Python 将一次又一次地在使用 malloc 在堆上申请空间，并不厌其烦地一次次 free。这样的操作不仅大大降低了运行效率（Python 本来就以速度慢被人诟病了 :），而且会在系统堆上造成内存碎片，严重影响 Python 的整体性能。

显然，Guido 是决不能容许这样的方案存在的，于是在 Python 中，对小整数对象，使用了对象池技术。刚才我们说了，PyIntObject 对象是 Immutable 对象，这带来了一个天大的喜讯，所以对象池里的 PyIntObject 对象能够被任意地共享。

给你一个整数 100，你说它是个“小”整数吗？那么 101 呢？小整数和大整数的分界线在哪里？Python 的回答是“it's all up on you”，你想它在哪里它就在哪里。

Python 中提供了一种方法，通过这种方法，用户可以动态地调整小整数与大整数的分界线，从而动态确定对象池中到底应该有多少个对象。呃，但是，老实说，Python 提供的这种方法比较原始，为了达到动态调整的目的，你只有自己修改源代码。

```
[intobject.c]
#ifdef NSMALLPOSINTS
#define NSMALLPOSINTS 100
```

```

#endif
#ifndef NSMALLNEGINTS
#define NSMALLNEGINTS      5
#endif
#if NSMALLNEGINTS + NSMALLPOSINTS > 0
/* References to small integers are saved in this array so that they
   can be shared.
   The integers that are saved are those in the range
   -NSMALLNEGINTS (inclusive) to NSMALLPOSINTS (not inclusive).
*/
static PyObject *small_ints[NSMALLNEGINTS + NSMALLPOSINTS];
#endif

```

Python2.4 中，将小整数集合的范围默认设定为[-5, 100)。但是你完全可以修改 NSMALLPOSINTS 和 NSMALLNEGINTS，重新编译 Python，从而将这个范围向两端伸展或收缩。

对于小整数对象，Python 直接将这此整数对应的 PyIntObject 缓存在内存中。那么对于大整数呢？很显然，整数对象是编程中使用得非常多的东西，谁敢保证只有小整数才会被频繁地使用呢。如果将所有的整数对应的 PyIntObject 对象都缓存在内存池中，自然是再理想不过了，但是这样对内存的使用是会被视为败家子，难免遭人鄙视的:)。时间与空间的两难选择，这个计算机领域最基本的矛盾在这里浮出水面。

2.3 大整数对象

Python 的设计者们所做出的妥协是，对于小整数，完全地缓存其 PyIntObject 对象。而对其它整数，Python 运行环境将提供一块内存空间，这些内存空间由这些大整数轮流使用，也就是说，谁需要的时候谁就使用。这样免去了不断地 malloc 之苦，又在一定程度上考虑了效率问题。我们下面将详细剖析其实现机制。

在 Python 中，有一个 PyIntBlock 结构，在这个结构的基础上，实现了一个单向列表。

```

[intobject.c]
#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(PyObject))

struct _intblock {
    struct _intblock *next;
    PyObject objects[N_INTOBJECTS];
};

typedef struct _intblock PyIntBlock;

```

```
static PyIntBlock *block_list = NULL;
static PyIntObject *free_list = NULL;
```

PyIntBlock，顾名思义，就是说这个结构里维护了一块（block）内存，其中保存了一些 PyIntObject 对象。从 PyIntBlock 的声明中可以看到，在一个 PyIntBlock 中维护着 N_INTOBJECTS 对象，做一个简单的计算，就可以知道是 82 个。显然，这个地方也是 Python 的设计者留给你的可以动态调整的地方，不过，你需要再一次地修改源代码并重新编译。

PyIntBlock 的单向列表通过 block_list 维护，而这些 block 中的 PyIntObject 的列表中可以被使用的内存通过 free_list 来维护。最开始的时候，这两个指针都被设置为空指针，如图 3 所示。



图 3

（注意：此后，我们将用红色箭头表示 free_list，蓝色箭头表示 block_list）

2.4 添加和删除

好了，现在我们大体上了解了 Python 中整数对象在内存中的体系结构。下面通过对 PyInt_FromLong 的考察，真实地展现一个个 PyIntObject 对象是如何从无到有地产生并融入到 Python 的整数对象体系中的。

```
[intobject.c]
PyObject* PyInt_FromLong(long ival)
{
    register PyIntObject *v;
    #if NSMALLNEGINTS + NSMALLPOSINTS > 0
        if (-NSMALLNEGINTS <= ival && ival < NSMALLPOSINTS) {
            v = small_ints[ival + NSMALLNEGINTS];
            Py_INCREF(v);
        }
        #ifdef COUNT_ALLOCS
            if (ival >= 0)
                quick_int_allocs++;
            else
                quick_neg_int_allocs++;
        #endif
        return (PyObject *) v;
    }
    #endif
    if (free_list == NULL) {
        if ((free_list = fill_free_list()) == NULL)
```



```

        return NULL;

    }

    /* Inline PyObject_New */
    v = free_list;
    free_list = (PyIntObject *)v->ob_type;
    PyObject_INIT(v, &PyInt_Type);
    v->ob_ival = ival;
    return (PyObject *) v;
}

```

在调用 `PyInt_FromLong` 时，首先会检查传入的 `long` 值是否属于小整数的范围，如果确实是属于小整数，那么很简单了，只需要返回在对象池中的对应的对象就可以了。

如果传入的 `long` 值不是属于小整数，Python 就会转向由 `block_list` 维护的内存。当首次调用 `PyInt_FromLong` 时，因为 `free_list` 为 `NULL`，这时会调用 `fill_free_list`：

```

[intobject.c]
static PyIntObject* fill_free_list(void)
{
    PyIntObject *p, *q;
    /* Python's object allocator isn't appropriate for large blocks.
    */
    p = (PyIntObject *) PyMem_MALLOC(sizeof(PyIntBlock));
    if (p == NULL)
        return (PyIntObject *) PyErr_NoMemory();
    ((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 = &((PyIntBlock *)p)->objects[0];
    q = p + N_INTOBJECTS;
    while (--q > p)
        q->ob_type = (struct _typeobject *) (q-1);
    q->ob_type = NULL;
    return p + N_INTOBJECTS - 1;
}

```

在 `fill_free_list` 中，会申请一个 `PyIntBlock` 结构体的对象，如图 4 所示：

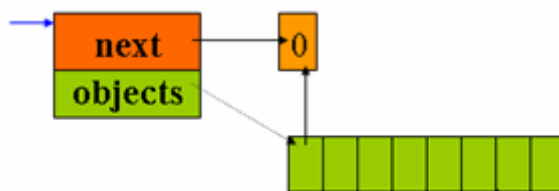


图 4

(注意：图中的虚线并不表示指针关系，虚线表示 `objects` 的更详细地表示方式，下同)

然后，会将该 Block 中的 `PyIntObject` 数组中的所有的 `PyIntObject` 对象通过指针依次连接起来，就像一个单向链表一样。在这里，使用了 `PyObject` 中的 `ob_type` 指针作为连接指针。可以看到，Python 的设计者为了解决问题，也就不再考虑什么类型安全了。就像政治一样，计算机也是一门妥协的艺术：)。 `fill_free_list` 完成后最终的 Block 如图 5 所示。可以看到，这时候，`free_list` 也在它该出现的位置了。从 `free_list` 开始，沿着 `ob_type` 指针，就可以遍历刚刚创建的 `PyIntBlock` 对象中的所有可使用的为 `PyIntObject` 准备的内存了，如图 5 所示：

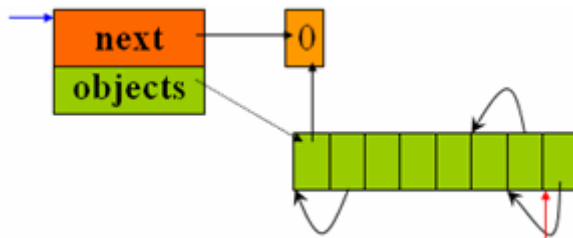


图 5

当一个 Block 中还有剩余的内存没有被一个 `PyIntObject` 占用时，`free_list` 就不会指向 `NULL`。所以在这种情况调用 `PyInt_FromLong` 不会申请新的 Block。只有在一个 Block 中的内存都被占用了，`PyInt_FromLong` 才会再次调用 `fill_free_list` 申请新的空间，为新的 `PyIntObject` 创造新的家园。图 6 展示了两块申请 Block 后 Block 链表的情况。值得注意的是，`block_list` 始终是指向最新创建的 `PyIntBlock` 对象。



图 6

在 `PyInt_FromLong` 中，当必要的空间被申请之后，将会把当前可用的 Block 中的内存空间划出一块，将在这块内存上创建我们需要的 `PyIntObject` 对象，同时，还会调整完成必要的初始化工作，以及调整 `free_list` 指针，使其指向下一块还没有被占用的内存。

从图中我们发现，两个 `PyIntBlock` 处于同一个链表当中，但是每一个 `PyIntBlock` 中至关重要的存放 `PyIntObject` 对象的 `objects` 却是分离的。这样的结构存在着隐患，考虑这样的情况：

现在有两个 `PyIntBlock` 对象，`PyIntBlock1` 和 `PyIntBlock2`，`PyIntBlock1` 中的 `objects` 已经被 `PyIntObject` 对象填满，而 `PyIntBlock2` 种的 `object` 只填充了一部分。所以现在 `free_list` 指针指向的是 `PyIntBlock2.objects` 中空闲得内存块。假设现在 `PyIntBlock1.objects` 中的一个 `PyIntObject` 对象被删除了，现在 `PyIntBlock1` 中有空闲的内存可用了，那么下次创建新的 `PyIntObject` 对象时应该使用 `PyIntBlock1` 中的这块内存。那么如何使 Python 意识到这块重获

自由的内存呢？如果象上图所示的 `PyIntBlock` 对象间的 `objects` 没有任何联系，那么显然不可能实现这样的功能，所以它们之间一定存在联系。实际上，不同 `PyIntBlock` 对象之间的空闲内存块是被链接在一起的，形成了一个单向链表，表头就是 `free_list`。

那么，不同 `PyIntBlock` 中的空闲内存块是在什么时候被链接在一起的呢，这一切都发生在一个 `PyIntObject` 对象被销毁的时候。

列位看官，花开两朵，各表一支：）这里我们先放下自由内存链表，仔细考察一下一个 `PyIntObject` 对象在被销毁时都发生了什么事。在对 Python 中对象机制的分析中，我们已经看到，每一个对象都有一个引用计数与之相连，当这个引用计数减少到 0 时，就意味着这个世上再也没有谁需要它了，于是 Python 运行时负责将这个对象销毁。Python 中不同对象在销毁时会进行不同的动作，销毁动作在与对象对应的类型对象中被定义，这个关键的操作就是类型中的 `tp_dealloc`。

下面看一看 `PyIntObject` 对象的 `tp_dealloc` 操作：

```
[intobject.c]
static void int_dealloc(PyIntObject *v)
{
    if (PyInt_CheckExact(v)) {
        v->ob_type = (struct _typeobject *)free_list;
        free_list = v;
    }
    else
        v->ob_type->tp_free((PyObject *)v);
}
```

在前面我们说了，由 `block_list` 维护的 `PyIntBlock` 的列表中的内存实际是所有的大整数对象所共同分享的。俗话说，皇帝轮流坐，明年到我家。当一个 `PyIntObject` 对象被删除时，它所占有的内存并没有被释放，归还给系统，而是继续被保留着。但是这一块内存现在已经是归 `free_list` 所维护的链表所有了，这表明在 `PyIntObject` 对象被删除后，它所占用的内存成了一块自由内存，可以供别的 `PyIntObject` 使用了。`int_dealloc` 完成的就是这么一个简单的指针维护的工作。当然，这些动作是在删除的对象确实是一个 `PyIntObject` 对象时发生的。如果删除的对象是一个整数的派生类的对象，那么 `int_dealloc` 不做任何动作，只是简单地调用派生类型中制定的释放函数。

在图 7 中我们形象地展示相继创建和删除 `PyIntObject` 对象，在这一过程中，内存中的 `PyIntObject` 对象以及 `free_list` 指针的变化情况。同时我们还展示了 `small_ints` 这个小整数的对象池。

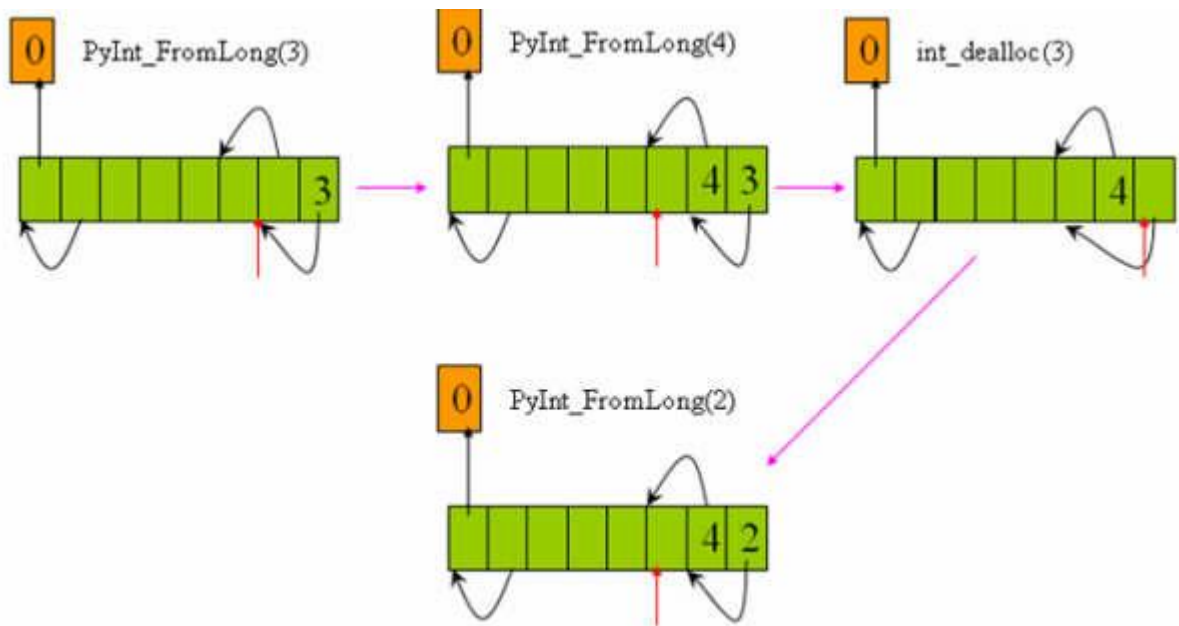


图 7

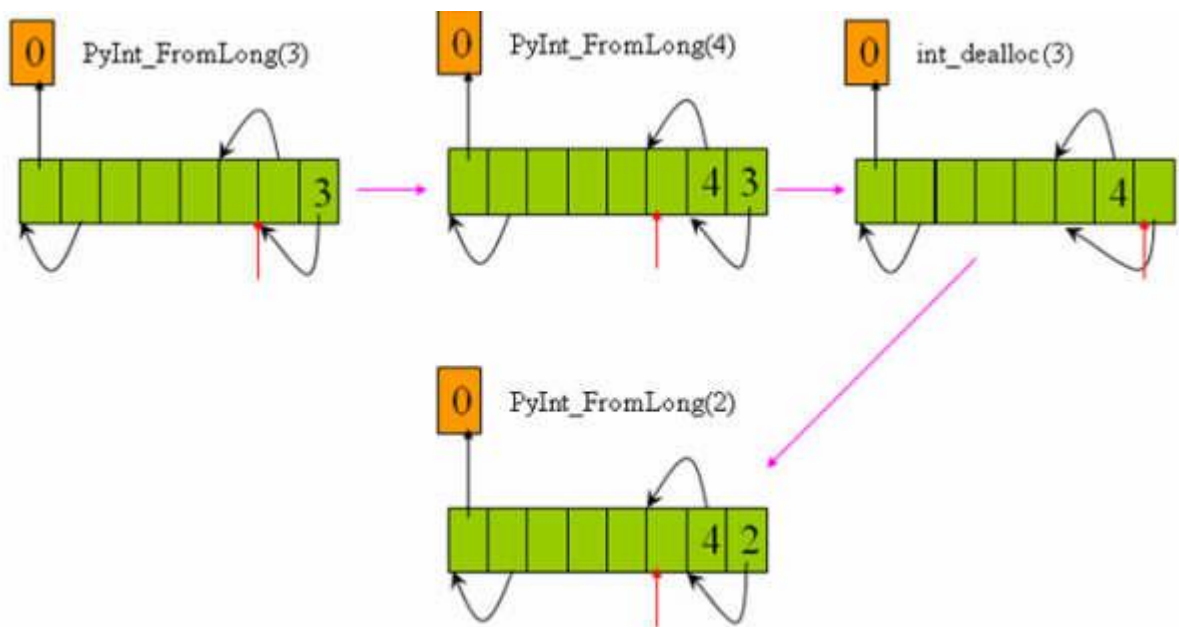


图 7

现在回头来看看刚才提到的不同 `PyIntBlock` 对象的 `objects` 间的空闲内存的互连问题。其实很简单，不同 `PyIntBlock` 对象中空闲内存的互连也是在 `int_dealloc` 被调用时实现的。图 8 展示了这个过程（黄色表示空闲内存）。

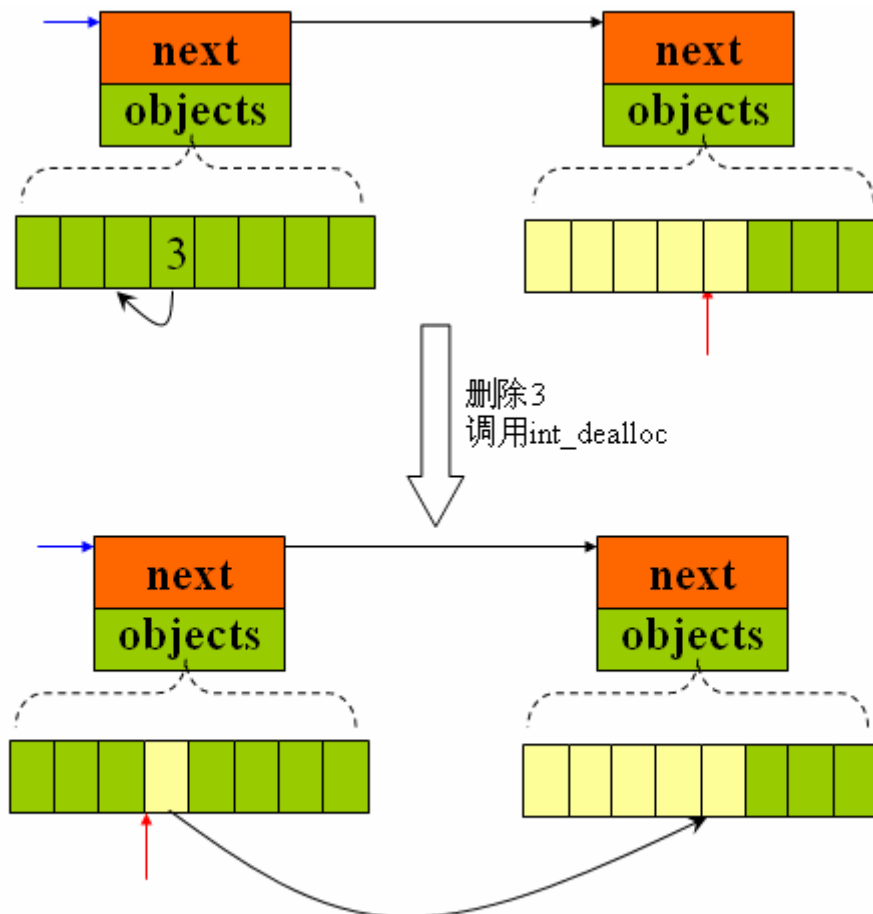


图 8

2.5 小整数对象池的初始化

现在，关于 Python 的整数对象体系，我们只剩下最后一个问题了。在 `small_ints` 中，我们看到，它维护的只是 `PyIntObject` 的指针，那么这些与天地同寿的小整数对象是在什么地方被创建和初始化的呢。完成这一切的神秘的函数正是 `_PyInt_Init`。

```
[intobject.c]
int _PyInt_Init(void)
{
    PyIntObject *v;
    int ival;
    #if NSMALLNEGINTS + NSMALLPOSINTS > 0
    for (ival = -NSMALLNEGINTS; ival < NSMALLPOSINTS; ival++)
    {
        if (!free_list && (free_list = fill_free_list()) == NULL)
            return 0;
        /* PyObject_New is inlined */
        v = free_list;
        free_list = (PyIntObject *)v->ob_type;
        PyObject_INIT(v, &PyInt_Type);
    }
}
```

```

    v->ob_ival = ival;
    small_ints[ival + NSMALLNEGINTS] = v;
}
#endif
return 1;
}

```

这些永生不灭的小整数对象也是生存在由 `block_list` 所维护的内存上，在 Python 运行时初始化的时候，`_PyInt_Init` 被调用，内存被申请，小整数对象被创建，然后就仙福永享，寿与天齐了 :)

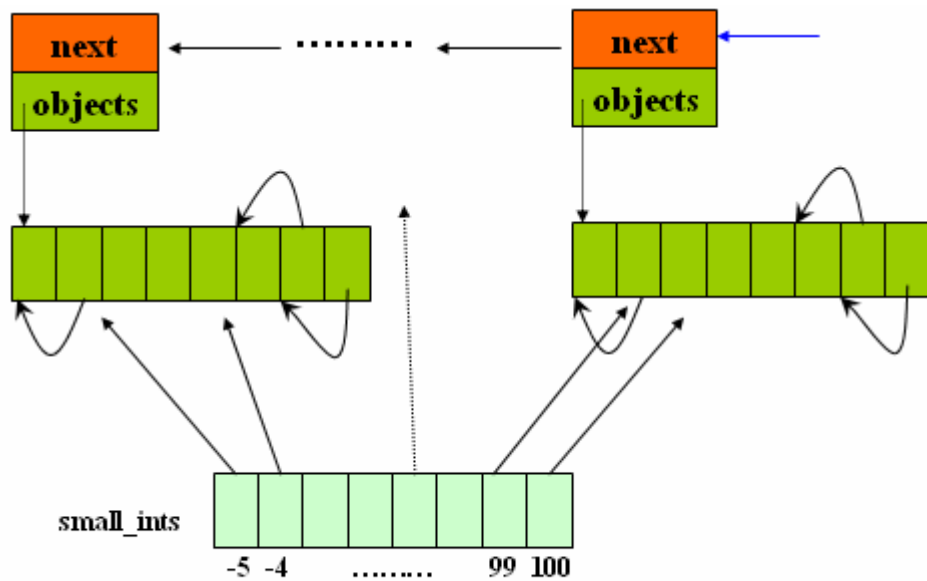


图 9

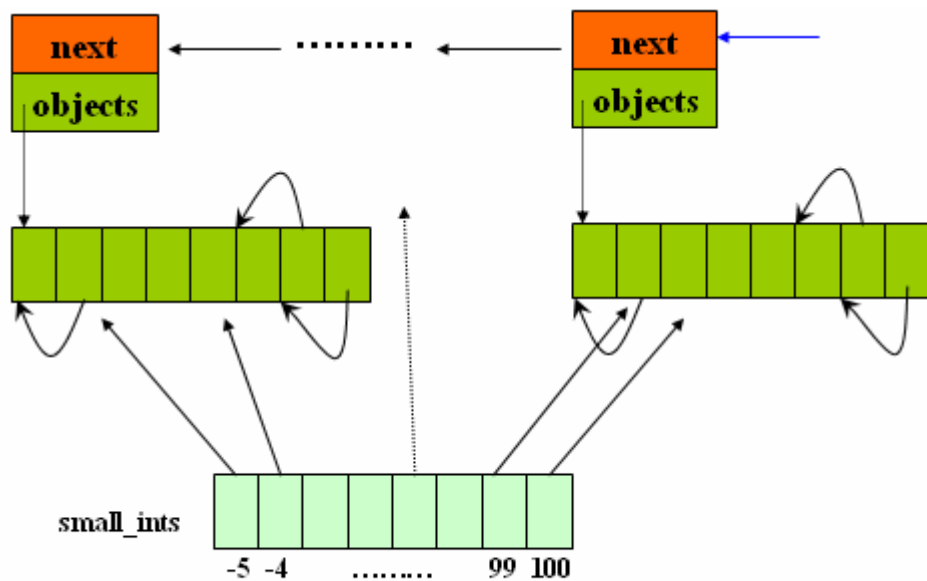


图 9

Python 源码剖析

——整数对象 PyIntObject(3)

本文作者: Robert Chen (search.pythoner@gmail.com)

3 Hack PyIntObject

现在, 让我们荡起双桨, 哦不对, 让我们挽起衣袖和裤脚 ☺, 来和 PyIntObject 大战一场。我们渴望在运行时观察 Python 的整数对象体系的变化。这一点, 完全可以通过修改 Python 源码来实现。我们修改了 int_print 的行为, 让它打印出关于 block_list 和 free_list 的信息, 以及小整数缓冲池的信息:

```
static int int_print(PyIntObject *v, FILE *fp, int flags)
{
    PyIntObject* intObjectPtr;
    PyIntBlock *p = block_list;
    PyIntBlock *last = NULL;
    int count = 0;
    int i;
```

```
    while(p != NULL)
    {
        ++count;
        last = p;
        p = p->next;
    }
```

```
    intObjectPtr = last->objects;
    intObjectPtr += N_INTOBJECTS - 1;
    printf("address %p\n", v);
    printf("***** value\trefCount *****\n");
    for(i = 0; i < 10; ++i, --intObjectPtr)
    {
        printf("%d\t\t%d\n", intObjectPtr->ob_ival,
intObjectPtr->ob_refcnt);
    }
```

```
    printf("block_list count : %d\n", count);
    printf("free_list : %p\n\n", free_list);
```

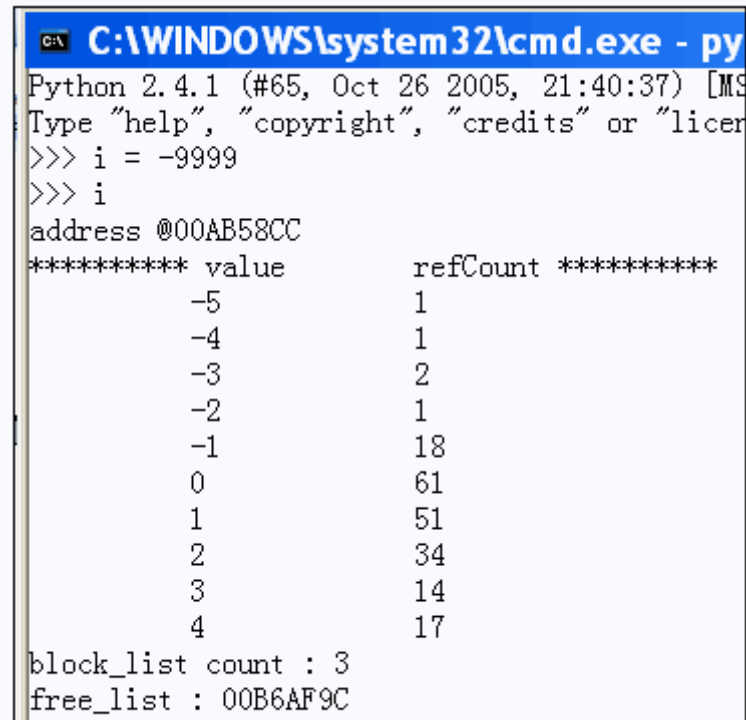
```
    return 0;
```



```
}
```

需要特别注意的是，在初始化小整数缓冲池时，对于 `block_list` 以及每个 `PyIntBlock` 的 `objects`，都是从后往前开始填充的，所以在初始化完成后，-5 应该在最后一个 `PyIntBlock` 对象的 `objects` 内最后一块内存，所以我们需要顺藤摸瓜，一直找到这最后的一块内存，才能观察从-5 到 4 这 10 个小整数。

首先我们创建一个 `PyIntObject` 对象-9999，从图 10 所示的输出信息可以看到，小整数对象很多都被 Python 自身使用多次了。



```
C:\WINDOWS\system32\cmd.exe - py
Python 2.4.1 (#65, Oct 26 2005, 21:40:37) [MS
Type "help", "copyright", "credits" or "licen
>>> i = -9999
>>> i
address @00AB58CC
***** value          refCount *****
          -5             1
          -4             1
          -3             2
          -2             1
          -1            18
           0            61
           1            51
           2            34
           3            14
           4            17
block_list count : 3
free_list : 00B6AF9C
```

图 10

现在的 `free_list` 指向地址为 00B6AF9C 的内存，根据上面对 `PyIntObject` 的分析，那么下一个 `PyIntObject` 会在这个地址安身立命。那么好，我们接着再建立了两个 `PyIntObject` 对象，它们的值分别是-123456:

```
C:\WINDOWS\system32\cmd.e
>>> a = -123456
>>> a
address @00B6AF9C
***** value          refCount ****
          -5           1
          -4           1
          -3           2
          -2           1
          -1          18
           0          61
           1          51
           2          34
           3          14
           4          17
block_list count : 3
free_list : 00B6AF90

C:\WINDOWS\system32\cmd.
>>> b = -123456
>>> b
address @00B6AF90
***** value          refCount **
          -5           1
          -4           1
          -3           2
          -2           1
          -1          18
           0          61
           1          51
           2          34
           3          14
           4          17
block_list count : 3
free_list : 00B6AF84
```

图 11

从图 11 所示的结果中可以看到 a 的地址正是创建 i 后 free_list 所指的地址，而 b 的地址也正是创建 a 后 free_list 所指的地址。虽然 a 和 b 的值都是一样的，但是它们确实是两个完全没有关系的 PyIntObject 对象，这点从地址上看得一清二楚。

现在我们将 b 删除，结果如图 12 所示：

```
C:\WINDOWS\system32\cmd.exe - py
>>> del b
>>> a
address @00B6AF9C
***** value          refCount *****
          -5           1
          -4           1
          -3           2
          -2           1
          -1          18
           0          61
           1          51
           2          34
           3          14
           4          17
block_list count : 3
free_list : 00B6AF90
```

图 12

删除 b 后，free_list 回退到了 a 创建后 free_list 的位置，这一点也跟前面的分析是一致的。

最后我们来看一看对小整数对象的监控，连续两次创建 PyIntObject 对象-5，结果如图 13 所示：

```
C:\WINDOWS\system32\cmd
>>> d1 = -5
>>> d1
address @00AB5778
***** value      refCount *
          -5        2
          -4        1
          -3        2
          -2        1
          -1       18
           0       61
           1       51
           2       34
           3       14
           4       17
block_list count : 3
free_list : 00B6AF90

C:\WINDOWS\system32\cmd
>>> d2 = -5
>>> d2
address @00AB5778
***** value      refCount *
          -5        3
          -4        1
          -3        2
          -2        1
          -1       18
           0       61
           1       51
           2       34
           3       14
           4       17
block_list count : 3
free_list : 00B6AF90
```

图 13

可以看到，两次创建的 `PyIntObject` 对象 `d1` 和 `d2` 的地址都是一样的，这证明它们实际上是同一个对象。同时，我们看到小整数池中 `-5` 的引用计数发生了变化，这证明 `d1` 和 `d2` 实际上都是指向这个对象。此外，`free_list` 没有发生任何变化。这些都与我们对 `PyIntObject` 的分析相符。

字符串对象，在任何一门主流编程语言中，都是整数对象之外使用最广泛的对象。它和整数对象犹如少林，武当，双峰对峙。本章将研究 `Python` 中的字符串对象的实现，同整数对象一样，字符串对象的实现中采用了很多额外的机制来保证性能的优化。

本章内容分为三个部分：

1. 研究 `Python` 中的字符串对象 `PyStringObject`
2. 研究字符串对象的效率加速机制
3. 分析字符串连接操作的效率，Hack `PyStringObject`

Python 源码剖析

——字符串对象 `PyStringObject(1)`

本文作者：Robert Chen(search.pythoner@gmail.com)

1. `PyStringObject` 与 `PyString_Type`

在对 `PyIntObject` 的分析中，我们看到了 Python 中的具有不可变长度数据的对象（定长对象）。在 Python 中，还大量存在着另一种对象，即具有可变长度数据的对象（变长对象）。与定长对象不同，对于变长对象而言，对象维护的数据的长度在对象定义时是不知道的。对于 `PyIntObject` 来说，其维护的数据的长度在对象定义时就已经确定了，是一个 `long` 变量的长度；而可变对象维护的数据的长度只能在对象创建时才能确定，考虑一下，我们只能在创建一个字符串或一个列表时才知道它们所维护的数据的长度，在此之前，对这个信息，我们一无所知。在变长对象中，实际上还可分为可变对象(`mutable`)和不可变对象(`immutable`)，可变对象是在对象创建之后，其维护的数据的长度还能变化的对象，比如一个 `list` 被创建后，可以向其中添加元素或删除元素，这些操作都会改变其维护的数据的长度；而不可变对象所维护的数据在对象创建之后就不能再改变了，比如 Python 中的 `string` 和 `tuple`，它们都不支持添加或删除的操作。本章我们将研究 Python 变长对象中的不可变对象——字符串对象。

在 Python 中，`PyStringObject` 是对字符串对象的抽象和表示。`PyStringObject` 是一个拥有可变长度内存的对象，这一点非常容易理解，因为对于表示“Hi”和“Python”的两个不同的 `PyStringObject` 对象，其内部需要的保存字符串内容的内存空间显然是不一样的。但同时，`PyStringObject` 对象又是一个不变对象(`Immutable`)。当创建了一个 `PyStringObject` 对象之后，该对象内部维护的字符串就不能再被改变了。这一点特性使得 `PyStringObject` 对象能作为 `PyDictObject` 的键值，但同时也使得一些字符串操作的效率大大降低，比如多个字符串的连接操作。

`PyStringObject` 对象的声明如下：

```
[stringobject.h]
typedef struct {
    PyObject_VAR_HEAD
    long ob_shash;
    int ob_sstate;
    char ob_sval[1];
} PyStringObject;
```

在 `PyStringObject` 的定义中我们看到，在 `PyStringObject` 的头部实际上是一个 `PyObject_VAR_HEAD`，其中有一个 `ob_size` 变量保存着对象中维护的可变长度内存的长度。虽然在 `PyStringObject` 的定义中，`ob_sval` 是一个字符的字符数组。但是 `ob_sval` 实际上是作为一个字符指针指向了一段内存，这段内存保存着这个字符串对象所维护的实际字符串，显然，这段内存不会只是一个字节。而这段内存的实际长度（字节），正是由 `ob_size` 来维护，这个机制是 Python 中所有拥有可变长度内存的对象的实现机制。比如对于 `PyStringObject` 对象“Python”，`ob_size` 的值就是 6。

同 C 中的字符串一样，`PyStringObject` 内部维护的字符串在末尾必须以 `'\0'` 结尾，但是由于字符串的实际长度是由 `ob_size` 维护的，所以 `PyStringObject` 表示

的字符串对象中间是可能出现字符'\0'的，这一点与 C 语言中不同，因为在 C 中，只要遇到了字符'\0'，就认为一个字符串结束了。所以，实际上，ob_sval 指向的是一段长度为 ob_size+1 个字节的内存，而且必须满足 ob_sval[ob_size] = '\0'。

PyStringObject 中的 ob_shash 变量其作用是缓存该对象的 HASH 值，这样可以避免每一次都重新计算该字符串对象的 HASH 值。如果一个 PyStringObject 对象还没有被计算过 HASH 值，那么 ob_shash 的初始值是-1。在计算一个对象的 HASH 值时，采用如下的算法：

```
[stringobject.c]
static long string_hash(PyStringObject *a)
{
    register int len;
    register unsigned char *p;
    register long x;

    if (a->ob_shash != -1)
        return a->ob_shash;
    len = a->ob_size;
    p = (unsigned char *) a->ob_sval;
    x = *p << 7;
    while (--len >= 0)
        x = (1000003*x) ^ *p++;
    x ^= a->ob_size;
    if (x == -1)
        x = -2;
    a->ob_shash = x;
    return x;
}
```

PyStringObject 对象的 ob_sstate 变量该对象是否被 Intern 的标志，关于 PyStringObject 的 Intern 机制，在后面会详细介绍。

下面看一下 PyStringObject 对应的类型对象：

```
[stringobject.c]
PyTypeObject PyString_Type = {
    PyObject_HEAD_INIT(&PyType_Type)
    0,
    "str",
    sizeof(PyStringObject),
    sizeof(char),
    .....,
    (reprfunc)string_repr,          /* tp_repr */
}
```

```

    &string_as_number,          /* tp_as_number */
    &string_as_sequence,       /* tp_as_sequence */
    &string_as_mapping,        /* tp_as_mapping */
    (hashfunc)string_hash,     /* tp_hash */
    0,                         /* tp_call */
    .....
    string_new,                /* tp_new */
    PyObject_Del,              /* tp_free */
};

```

可以看到, 在 `PyStringObject` 的类型对象中, `tp_itemsize` 被设置为 `sizeof(char)`, 即一个字节。对于 Python 中的任何一种变长对象, `tp_itemsize` 这个域是必须设置的, `tp_itemsize` 指明了由变长对象保存的元素的单位长度, 所谓单位长度即是指一个对象在内存中的长度。这个 `tp_itemsize` 和 `ob_size` 共同决定了应该额外申请的内存的总大小是多少。

需要注意的是, 我们看到, `tp_as_number`, `tp_as_sequence`, `tp_as_mapping`, 三个域都被设置了。这表示 `PyStringObject` 对数值操作, 序列操作和映射操作都支持。

2. 创建 `PyStringObject` 对象

Python 提供两条路径, 从 C 中原生的字符串创建 `PyStringObject` 对象。我们先考察一下最一般的 `PyString_FromString`:

```

[stringobject.c]
PyObject *
PyString_FromString(const char *str)
{
    register size_t size;
    register PyStringObject *op;

    assert(str != NULL);
    /*判断字符串长度*/
    size = strlen(str);
    if (size > INT_MAX) {
        PyErr_SetString(PyExc_OverflowError,
            "string is too long for a Python string");
        return NULL;
    }
}

```

```

    /*处理 null string*/
    if (size == 0 && (op = nullstring) != NULL) {
#ifdef COUNT_ALLOCS
        null_strings++;
#endif
        Py_INCREF(op);
        return (PyObject *)op;
    }
    if (size == 1 && (op = characters[*str & UCHAR_MAX]) != NULL) {
#ifdef COUNT_ALLOCS
        one_strings++;
#endif
        Py_INCREF(op);
        return (PyObject *)op;
    }

    /* 创建新的 PyStringObject 对象, 并初始化 */
    /* Inline PyObject_NewVar */
    op = (PyStringObject *)PyObject_MALLOCSIZE(sizeof(PyStringObject) +
size);
    if (op == NULL)
        return PyErr_NoMemory();
    PyObject_INIT_VAR(op, &PyString_Type, size);
    op->ob_shash = -1;
    op->ob_sstate = SSTATE_NOT_INTERNERED;
    memcpy(op->ob_sval, str, size+1);

    /* Intern (共享) 长度较短的 PyStringObject 对象 */
    if (size == 0) {
        PyObject *t = (PyObject *)op;
        PyString_InternInPlace(&t);
        op = (PyStringObject *)t;
        nullstring = op;
        Py_INCREF(op);
    } else if (size == 1) {
        PyObject *t = (PyObject *)op;
        PyString_InternInPlace(&t);
        op = (PyStringObject *)t;
        characters[*str & UCHAR_MAX] = op;
        Py_INCREF(op);
    }
    return (PyObject *) op;

```



```
}
```

显然，传给 `PyString_FromString` 的参数必须是一个指向以 `NULL` 结尾的字符串的指针。在从一个原生字符串创建 `PyStringObject` 时，首先需要检查该字符数组的长度，如果字符数组的长度大于了 `MAX_INT`，那么 Python 将不会创建对应得 `PyStringObject` 对象。`MAX_INT` 在系统的头文件中定义，是一个平台相关的值，在 WIN32 系统下，该值为：

```
#define INT_MAX 2147483647
```

嗯，这个界限值确实非常庞大了，如果不是由于变态，几乎没有人会去试图超越这个禁区的：)

接下来，将会检查传入的字符串是不是一个空串，对于空串，Python 并不是每一次都会创建相应得 `PyStringObject`。`Python` 运行时有一个 `PyStringObject` 对象指针 `nullstring` 专门负责处理空的字符数组。如果第一次在一个空字符串基础上创建 `PyStringObject`，由于 `nullstring` 指针被初始化为 `NULL`，所以 Python 会为此空字符串建立一个 `PyStringObject` 对象，将这个 `PyStringObject` 对象通过 Intern 机制进行共享，然后将 `nullstring` 指向这个被共享的对象。如果在以后 Python 检查到需要为一个空字符串创建 `PyStringObject` 对象，这时 `nullstring` 已经存在了，那么就直接返回 `nullstring` 的引用。

接下来需要进行的动作就是申请内存，创建 `PyStringObject` 对象。可以看到，这里申请的内存除了 `PyStringObject` 的内存，还有为字符数组内的元素申请的额外的内存。然后，将 `HASH` 缓存值设为 -1，将 Intern 标志设为 `SSTATE_NOT_INTERNED`。最后将字符数组内的字符拷贝到 `PyStringObject` 所维护的空间中，在拷贝的过程中，将字符数组最后的 `'\0'` 字符也拷贝了。加入我们对于字符数组 `"Python"` 建立 `PyStringObject` 对象，那么对象建立完成后在内存中的状态如图 1 所示：

		ob_size	ob_shash	ob_sstate	ob_sval
ref	type	6	-1	0	P y t h o n \0

深色为 `PyStringObject` 内存，浅色为额外内存

图 1

在 `PyString_FromString` 之外，还有一条创建 `PyStringObject` 对象的途径：
`PyString_FromStringAndSize`：

```
[stringobject.c]
PyObject* PyString_FromStringAndSize(const char *str, int size)
{
    register PyStringObject *op;
    /*处理 null string*/
    if (size == 0 && (op = nullstring) != NULL) {
```

```

#ifdef COUNT_ALLOCS
    null_strings++;
#endif
    Py_INCREF(op);
    return (PyObject *)op;
}

if (size == 1 && str != NULL &&
    (op = characters[*str & UCHAR_MAX]) != NULL)
{
#ifdef COUNT_ALLOCS
    one_strings++;
#endif
    Py_INCREF(op);
    return (PyObject *)op;
}

/* Inline PyObject_NewVar */
op = (PyStringObject *)PyObject_MALLOC(sizeof(PyStringObject) +
size);
if (op == NULL)
    return PyErr_NoMemory();
PyObject_INIT_VAR(op, &PyString_Type, size);
op->ob_shash = -1;
op->ob_sstate = SSTATE_NOT_INTERNERED;
if (str != NULL)
    memcpy(op->ob_sval, str, size);
op->ob_sval[size] = '\0';
/* share short strings */
if (size == 0) {
    PyObject *t = (PyObject *)op;
    PyString_InternInPlace(&t);
    op = (PyStringObject *)t;
    nullstring = op;
    Py_INCREF(op);
} else if (size == 1 && str != NULL) {
    PyObject *t = (PyObject *)op;
    PyString_InternInPlace(&t);
    op = (PyStringObject *)t;
    characters[*str & UCHAR_MAX] = op;
    Py_INCREF(op);
}
return (PyObject *) op;
}

```

PyString_FromStringAndSize 的操作过程和 PyString_FromString 一般无二，只是有一点，PyString_FromString 传入的参数必须是以 NULL 结尾的字符数组的指针，而 PyString_FromStringAndSize 不会有这样的要求，因为通过传入的 size 参数就可以确定需要拷贝的字符的个数。

Python 源码剖析

——字符串对象 PyStringObject(2)

本文作者: Robert Chen(search.pythoner@gmail.com)

3. Intern 机制

无论是 PyString_FromString 还是 PyString_FromStringAndSize,我们都注意到,当字符数组的长度为 0 或 1 时,需要进行了一个特别的动作:

PyString_InternInPlace。这就是前面所提到的 Intern 机制。

PyStringObject 对象的 Intern 机制其目的是对于被 Intern 之后的字符串,在整个 Python 运行时,系统中都只有唯一的与该字符串对应的 PyStringObject 对象。这样当判断两个 PyStringObject 对象是否相同时,如果它们都被 Intern 了,那么只需要简单地检查它们对应的 PyObject*是否相同即可。这个机制既节省了空间,又简化了对 PyStringObject 对象的比较,嗯,可谓是一箭双雕哇 :)

假如在某个时刻,我们创建了一个 PyStringObject 对象 A,其表示的字符串是“Python”,在之后的某一时刻,加入我们想为“Python”再次建立一个 PyStringObject 对象,通常情况下,Python 会为我们重新申请内存,创建一个新的 PyStringObject 对象 B,A 与 B 是完全不同的两个对象,尽管其内部维护的字符数组是完全相同的。

这就带来了一个问题,加入我们在程序中创建了 100 个“Python”的 PyStringObject 对象呢?显而易见,这样会大量地浪费珍贵的内存。因此 Python 对 PyStringObject 对象引入了 Intern 机制。在上面的例子中,如果对于 A 应用了 Intern 机制,那么之后要创建 B 的时候,Python 会首先在系统中记录的已经被 Intern 的 PyStringObject 对象中查找,如果发现该字符数组对应的 PyStringObject 对象已经存在了,那么就将该对象的引用返回,而不会创建对象 B。

PyString_InternInPlace 正是完成对一个对象的 Intern 操作:

```
[stringobjec.c]
void PyString_InternInPlace(PyObject **p)
{
    register PyStringObject *s = (PyStringObject *)(*p);
    PyObject *t;
    if (s == NULL || !PyString_Check(s))
        Py_FatalError("PyString_InternInPlace: strings only
please!");
    /* If it's a string subclass, we don't really know what putting
it in the interned dict might do. */
    if (!PyString_CheckExact(s))
```

```

        return;
    if (PyString_CHECK_INTERNED(s))
        return;

    if (interned == NULL) {
        interned = PyDict_New();
        if (interned == NULL) {
            PyErr_Clear(); /* Don't leave an exception */
            return;
        }
    }
    t = PyDict_GetItem(interned, (PyObject *)s);
    if (t) {
        Py_INCREF(t);
        Py_DECREF(*p);
        *p = t;
        return;
    }

    if (PyDict_SetItem(interned, (PyObject *)s, (PyObject *)s) < 0)
    {
        PyErr_Clear();
        return;
    }
    /* The two references in interned are not counted by refcnt.
       The string deallocator will take care of this */
    s->ob_refcnt -= 2;
    PyString_CHECK_INTERNED(s) = SSTATE_INTERNED_MORTAL;
}

```

首先会进行一系列的检查。首先，会检查传入的对象是否是一个 `PyStringObject` 对象，Intern 机制只能应用在 `PyStringObject` 对象上，甚至对于它的派生类对象系统都不会应用 Intern 机制。然后，会检查传入的 `PyStringObject` 对象是否已经被 Intern 机制处理过了，Python 不会对同一个 `PyStringObject` 对象进行一次以上的 Intern 操作。

从代码中我们可以清楚地看到，Intern 机制的核心在于 `interned` 这个东西，那么这个东西是个什么东西呢？

```
static PyObject *interned;
```

从 `stringobject.c` 中的定义我们完全不知道 `interned` 是个什么东西，然而在这里我们看到，`interned` 实际上指向的是 `PyDict_New` 创建的一个对象。而 `PyDict_New` 实际上创建了一个 `PyDictObject` 对象，这个对象我们将在后面详细

地剖析。其实，现在，一个 PyDictObject 对象完全可以看作是 C++ 中的 map，即 `map<PyObject*, PyObject*>`。

现在一切都清楚了，所谓的 Intern 机制，实际上就是系统中有一个 (Key, Value) 的映射的集合 `interned`。在这个集合中，记录着被应用了 Intern 机制的 PyStringObject 对象。当对一个 PyStringObject 对象 A 应用 Intern 机制时，首先会在 `interned` 中检查是否有满足一下条件的对象 B：B 中维护的原生字符串与 A 相同。如果确实存在对象 B，那么指向 A 的 PyObject 指针将会指向 B，而 A 的引用计数减 1，这样，其实 A 只是一个临时被创建的对象。如果 `interned` 中还不存在这样的 B，那么就将 A 记录到 `interned` 中。

图 2 展示了如果 `Interned` 中存在这样的对象 B，在对 A 进行 Intern 操作时，原本指向 A 的 PyObject 指针的变化：

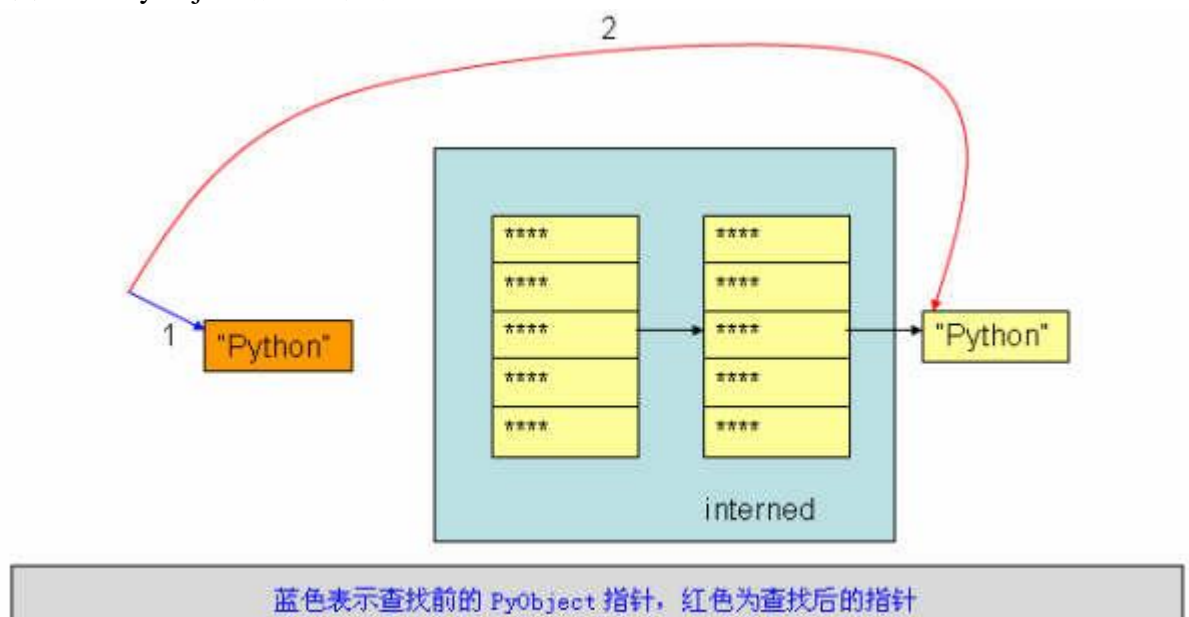


图 2

对于被 Intern 的 PyStringObject 对象，Python 采用了特殊的引用计数机制。在将一个 PyStringObject 对象 A 的 PyObject 指针作为 Key 和 Value 添加到 `interned` 中时，PyDictObject 对象会通过这两个指针对 A 的引用计数进行两次加 1 操作。但是 Python 的设计者规定在 `interned` 中 A 的指针不能被视为对象 A 的有效引用，因为如果是有效引用的话，那么 A 的引用计数在 Python 运行时结束之前永远都不可能为 0，因为至少有 `interned` 中的两个指针引用了 A，那么删除 A 就永远不可能，这显然是没有道理的。因此 `interned` 中的指针不能作为 A 的有效引用。这也就是在 `PyString_InternInPlace` 最后会将引用计数减 2 的原因。当 A 的引用计数在某个时刻减为 0 之后，系统将会销毁对象 A，那么我们可以预期，在销毁 A 的同时，会在 `interned` 中删除指向 A 的指针，显然，这一点在 `string_dealloc` 得到了验证：

```
[stringobject.c]
static void string_dealloc(PyObject *op)
{
    switch (PyString_CHECK_INTERNED(op)) {
        case SSTATE_NOT_INTERNED:
            break;
```

```

    case SSTATE_INTERNED_MORTAL:
        /* revive dead object temporarily for DelItem */
        op->ob_refcnt = 3;
        if (PyDict_DelItem(interned, op) != 0)
            Py_FatalError(
                "deletion of interned string failed");
        break;

    case SSTATE_INTERNED_IMMORTAL:
        Py_FatalError("Immortal interned string died.");

    default:
        Py_FatalError("Inconsistent interned string state.");
}
op->ob_type->tp_free(op);
}

```

前面提到，Python 在创建一个字符串时，会首先在 `interned` 中检查是否已经有该字符串对应得 `PyStringObject` 对象了，如果有，则不用创建新的，这样可以节省内存空间。事到如今，我必须承认，我说谎了：）节省内存空间是没错的，可是 Python 并不是在创建 `PyStringObject` 时就通过 `interned` 实现了节省空间的目的。事实上，从 `PyString_FromString` 中可以看到，无论如何，一个合法的 `PyString_FromString` 对象是会被创建的，同样，我们可以注意到，`PyString_InternInPlace` 也只对 `PyStringObject` 起作用。事实正是如此，Python 始终会为字符串 `S` 创建 `PyStringObject` 对象，尽管 `S` 在 `interned` 中已经有一个与之对应的 `PyStringObject` 对象了。而 `Intern` 机制是在 `S` 被创建后才起作用的，通常 Python 在运行时创建了一个 `PyStringObject` 对象 `Temp` 后，基本上都会调用 `PyString_InternInPlace`，`Intern` 机制会减少 `Temp` 的引用计数，`Temp` 对象会由于引用计数减为 0 而被销毁，它只是作为一个临时对象昙花一现地在内存中闪现，然后湮灭。

那么我们现在有一个疑问了，是否可以直接在 C 的原生字符串上做 `Intern` 的动作，而不再需要再创建这样一个临时对象呢？事实上，Python 确实提供了一个以 `char*` 为参数的 `Intern` 机制相关函数，但是你会相当失望，嗯，因为它基本上是换汤不换药的：

```

[stringobject.c]
PyObject* PyString_InternFromString(const char *cp)
{
    PyObject *s = PyString_FromString(cp);
    if (s == NULL)
        return NULL;
}

```

```
PyString_InternInPlace(&s);  
return s;  
}
```

临时对象照样被创建出来，实际上，仔细一想，就会发现在 Python 中，必须创建这样一个临时的 PyStringObject 对象来完成 Intern 操作。为什么呢？答案就在 PyDictObject 对象 interned 中，因为 PyDictObject 必须以 PyObject 指针作为键。

关于 PyStringObject 对象的 Intern 机制，还有一点需要注意。实际上，被 Intern 的 PyStringObject 对象分为两类，一类是 SSTATE_INTERNERD_IMMORTAL 状态的，而另一类是 SSTATE_INTERNERD_MORTAL 状态的，这两种状态的区别在 string_dealloc 中可以清晰地看到，显然，SSTATE_INTERNERD_IMMORTAL 状态的 PyStringObject 对象是永远不会被销毁的，它将与 Python run time 同年同月同日死。

PyString_InternInPlace 只能创建 SSTATE_INTERNERD_MORTAL 状态的 PyStringObject 对象，如果想创建 SSTATE_INTERNERD_IMMORTAL 状态的对象，必须要通过另外地接口，在调用了 PyString_InternInPlace 后，强制改变 PyStringObject 的 intern 状态。

```
[stringobject.c]  
void PyString_InternImmortal(PyObject **p)  
{  
    PyString_InternInPlace(p);  
    if (PyString_CHECK_INTERNERD(*p) != SSTATE_INTERNERD_IMMORTAL) {  
        PyString_CHECK_INTERNERD(*p) = SSTATE_INTERNERD_IMMORTAL;  
        Py_INCREF(*p);  
    }  
}
```

4. 字符缓冲池

最后需要注意的一点是与 PyIntObject 中的小整数对象的对象池一样，Python 的设计者为 PyStringObject 中的一个字节的字符对象也设计了这样一个对象池 characters。

```
static PyStringObject *characters[ UCHAR_MAX + 1];
```

其中的 UCHAR_MAX 是在系统头文件中定义的常量，这也是一个平台相关的常量，在 Win32 平台下：

```
#define UCHAR_MAX    0xff    /* maximum unsigned char value */
```

在 Python 的整数对象体系中，小整数的缓冲池是在 Python runtime 初始化时被创建的，而字符串对象体系中的字符缓冲池则是以静态变量的形式存在着的。在 Python runtime 初始化完成之后，缓冲池中的所有 PyStringObject 指针都为空。

在创建一个 PyStringObject 时，无论是调用 PyString_FromString 还是 PyString_FromStringAndSize，在创建的字符串实际上是一个字符时，会进行如下的操作：


```

[stringobject.c]
PyObject* PyString_FromStringAndSize(const char *str, int size)
{
    . . . . .
    else if (size == 1 && str != NULL)
    {
        PyObject *t = (PyObject *)op;
        PyString_InternInPlace(&t);
        op = (PyStringObject *)t;
        characters[*str & UCHAR_MAX] = op;
        Py_INCREF(op);
    }
    return (PyObject *) op;
}

```

先对所创建的字符串（字符）对象进行 Intern 操作，再将 Intern 的结果缓存到字符缓冲池 characters 中。图 3 演示了缓存一个字符对象的过程。

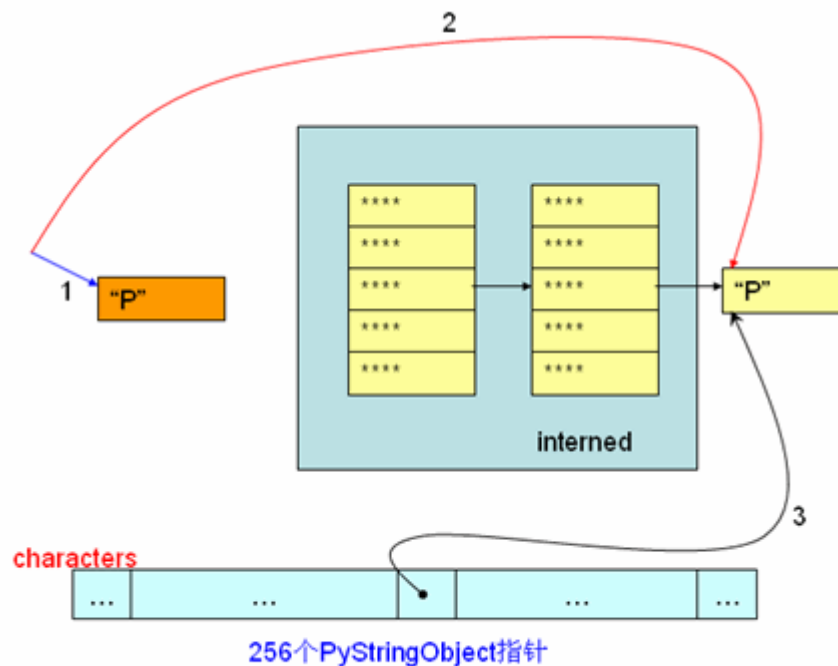


图 3

3 条带有标号的曲线既代表指针，又代表进行操作的顺序：

- 1) 创建 PyStringObject 对象“P”
- 2) 对对象“P”进行 Intern 操作
- 3) 将对象“P”缓存至字符缓冲池中

同样，在创建 PyStringObject 时，会首先检查所要创建的是不是一个字符对象，然后检查字符缓冲池中是否已经有了这个字符的字符对象的缓冲，如果有，则直接返回这个缓冲的对象即可：

```

[stringobject.c]

```

```

PyObject* PyString_FromStringAndSize(const char *str, int size)
{
    register PyStringObject *op;
    .....
    if (size == 1 && str != NULL &&
        (op = characters[*str & UCHAR_MAX]) != NULL)
    {
#ifdef COUNT_ALLOCS
        one_strings++;
#endif
        Py_INCREF(op);
        return (PyObject *)op;
    }
    .....
}

```

Trackback: <http://tb.donews.net/TrackBack.aspx?PostId=667700>

[\[点击此处收藏本文\]](#) 发表于 2005 年 12 月 22 日 11:23 PM

huangyi 发表于 2005-12-25 11:03 PM IP: 222.20.237.*

我发现只有空串和字符会用 intern 机制

如果是空串或字符则检查 nullstring 和 characters 数组是否有
有则直接返回

然后申请空间 如果是空串或字符 则放到 intern 里去

既然有了 nullstring 和 characters 数组 还要 intern 干什么呢

Robert 发表于 2005-12-26 12:46 AM IP: 218.104.52.*

嗯，从 PyStringObject 对象的 code 来看，在创建 PyStringObject 对象的时候，确实只对字符和 nullstring 进行了 Inter 操作。实际上在别的使用 PyStringObject 对象的地方，会在创建了对象后，调用 Inter 机制。比如在 dict 中，当你设置一个 (key, value) 的对时，就会用到这个 Intern 机制。

int

```

PyDict_SetItemString(PyObject *v, const char *key, PyObject *item)
{
    PyObject *kv;
    kv = PyString_FromString(key);
    PyString_InternInPlace(&kv); /* XXX Should we really? */
    err = PyDict_SetItem(v, kv, item);
    Py_DECREF(kv);
    return err;
}

```

可以搜一下 Python 的所有源文件，你会发现还有好几个地方都会用到这个 Intern 机制。

在 Python 交互环境中输入如下的表达式，会发现两个字符串对象的 id 相同，表示 s1 和 s2 指向的是同一个对象，这其实也是 Intern 机制的作用。

```

Python 2.4 (#60, Nov 30 2004, 11:49:19) [MSC v.1310 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> s1 = "abcde"
>>> s2 = "abcde"
>>> id(s1)
11737664
>>> id(s2)
11737664

```

Python 源码剖析

——字符串对象 PyStringObject(3)

本文作者: Robert Chen(search.pythoner@gmail.com)

5. PyStringObject 效率相关问题

关于 PyStringObject，有一个地球人都知道的严重影响 Python 程序执行效率的问题，有一种说法，绝大部分执行效率特别低下的 Python 程序都是由于没有注意到这个问题所致。下面我们就来看看这个在 Python 中举足轻重的问题——字符串连接。

假如现在有两个字符串“Python”和“Ruby”，在 Java 或 C#中，都可以通过使用“+”操作符，将两个字符串连接在一起，得到一个新的字符串

“PythonRuby”。当然，Python 中同样提供了利用“+”操作符连接字符串的功能，然而不幸的是，这样的做法正是万恶之源。

Python 中通过“+”进行字符串连接的方法效率及其低下，其根源在于 Python 中的 PyStringObject 对象是一个不可变对象。这就意味着当进行字符串连接时，实际上是必须要创建一个新的 PyStringObject 对象。这样，如果要连接 N 个 PyStringObject 对象，那么就必须进行 N-1 次的内存申请及内存搬运的工作。这将严重影响 Python 的执行效率。

官方推荐的做法是通过利用 PyStringObject 对象的 join 操作来对存储在 list 或 tuple 中的一组 PyStringObject 对象进行连接操作，这种做法只需要分配一次内存，执行效率将大大提高。

下面我们通过考察源码来更细致地了解这一问题。

通过 “+” 操作符对字符串进行连接时，会调用 string_concat 函数：

```
[stringobject.c]
static PyObject* string_concat(register PyStringObject *a, register
PyObject *bb)
{
    register unsigned int size;
    register PyStringObject *op;
    #define b ((PyStringObject *)bb)
    .....
    size = a->ob_size + b->ob_size;
    /* Inline PyObject_NewVar */
    op = (PyStringObject *)PyObject_MALLOC(sizeof(PyStringObject) +
size);
    if (op == NULL)
        return PyErr_NoMemory();
    PyObject_INIT_VAR(op, &PyString_Type, size);
    op->ob_shash = -1;
    op->ob_sstate = SSTATE_NOT_INTERNERED;
    memcpy(op->ob_sval, a->ob_sval, (int) a->ob_size);
    memcpy(op->ob_sval + a->ob_size, b->ob_sval, (int) b->ob_size);
    op->ob_sval[size] = '\\0';
    return (PyObject *) op;
#undef b
}
```

对于任意两个 PyStringObject 对象的连接，就会进行一次内存申请的动作。而如果利用 PyStringObject 对象的 join 操作，则会进行如下的动作（假设是对 list 中的 PyStringObject 对象进行连接）：

```
[stringobject.c]
static PyObject* string_join(PyStringObject *self, PyObject *orig)
{
    char *sep = PyString_AS_STRING(self);
    const int seplen = PyString_GET_SIZE(self);
    PyObject *res = NULL;
    char *p;
    int seqlen = 0;
    size_t sz = 0;
    int i;
    PyObject *seq, *item;
    ..... //获得 list 中 PyStringObject 对象的个数，保存在 seqlen 中
```

```

for (i = 0; i < seqlen; i++)
{
    const size_t old_sz = sz;
    item = PySequence_Fast_GET_ITEM(seq, i);
    sz += PyString_GET_SIZE(item);
    if (i != 0)
        sz += seplen;
}
/* 申请内存空间 */
res = PyString_FromStringAndSize((char*)NULL, (int)sz);
/* 连接 list 中的每一个 PyStringObject 对象*/
p = PyString_AS_STRING(res);
for (i = 0; i < seqlen; ++i)
{
    size_t n;
    /* 获得 list 中的一个 PyStringObject 对象*/
    item = PySequence_Fast_GET_ITEM(seq, i);
    n = PyString_GET_SIZE(item);
    memcpy(p, PyString_AS_STRING(item), n);
    p += n;
    if (i < seqlen - 1)
    {
        memcpy(p, sep, seplen);
        p += seplen;
    }
}
Py_DECREF(seq);
return res;
}

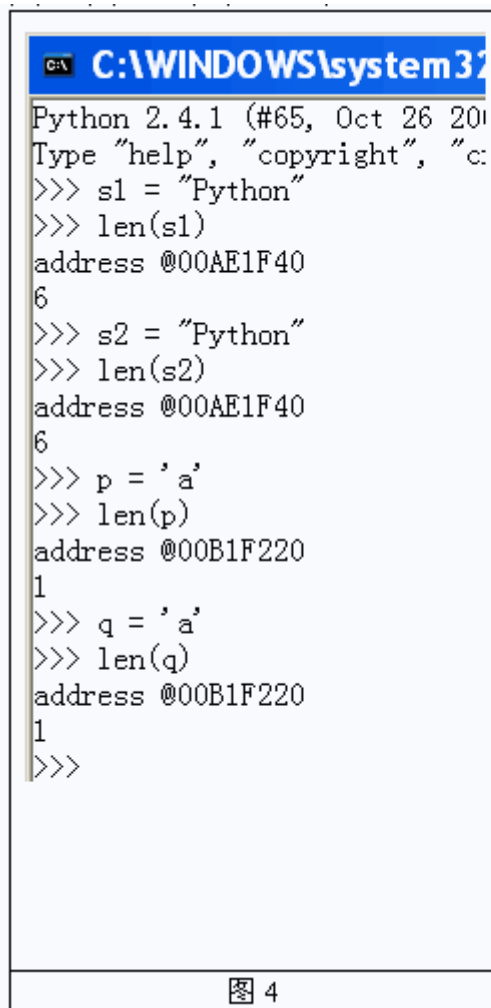
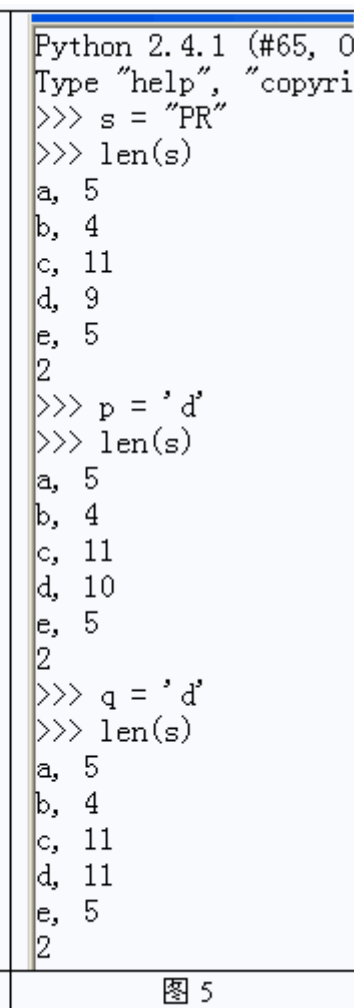
```

执行 join 操作时，会首先统计出在 list 中一共有多少个 PyStringObject 对象，并统计这些 PyStringObject 对象所维护的字符串一共有多长，然后申请内存，将 list 中所有的 PyStringObject 对象维护的字符串都拷贝到新开辟的内存空间中。注意，这里只进行了一次内存空间的申请，就完成了 N 个 PyStringObject 对象的连接操作。相比于“+”操作符，待连接的 PyStringObject 对象越多，效率的提升也越明显。

6. Hack PyStringObject

在这一节，我们对 PyStringObject 在运行时行为进行两项观察。

首先, 观察 Intern 机制, 在 Python Interactive 环境中, 创建一个 PyStringObject 后, 就会对这个 PyStringObject 对象进行 Intern 操作, 所以我们预期内容相同的 PyStringObject 对象在 Intern 后应该是同一个对象, 图 4 是观察的结果:

 <pre> Python 2.4.1 (#65, Oct 26 2001) Type "help", "copyright", "credits()" or "quit()" >>> s1 = "Python" >>> len(s1) 6 >>> s2 = "Python" >>> len(s2) 6 >>> p = 'a' >>> len(p) 1 >>> q = 'a' >>> len(q) 1 >>> </pre>	 <pre> Python 2.4.1 (#65, Oct 26 2001) Type "help", "copyright", "credits()" or "quit()" >>> s = "PR" >>> len(s) 2 >>> p = 'd' >>> len(s) 2 >>> q = 'd' >>> len(s) 2 >>> </pre>
图 4	图 5

通过在 `string_length` 中添加打印地址的代码, 我们可以在运行时获得每一个 PyStringObject 对象的地址, 从观察结果中可以看到, 无论是对于一般的字符串, 还是对于单个字符, Intern 机制最终都使不同的 PyStringObject 指针指向了相同的对象。

然后, 我们观察 Python 中进行缓冲处理的字符对象, 同样是通过在 `string_length` 中添加代码, 打印出缓冲池中从 a 到 e 的字符对象的引用计数信息。需要注意的是, 为了避免执行 `len(...)` 对引用计数的影响, 我们并不会对 a 到 e 的字符对象调用 `len` 操作, 而是对另外的 PyStringObject 对象调用 `len` 操作:

```

static void ShowCharater()
{
    char a = 'a';
    PyStringObject** posA = characters+(unsigned short)a;
    int i;
    for(i = 0; i < 5; ++i)
    {
        PyStringObject* strObj = posA[i];
    }
}

```

```
printf("%s, %d\n", strObj->ob_sval, strObj->ob_refcnt);  
}  
}
```

图 5 展示了观察的结果，可以看到，在创建字符对象时，Python 确实只是使用了缓冲池里的对象，而没有创建新的对象。

Python 源码剖析

——PyListObject 对象(1)

本文作者: Robert Chen (search.pythoner@gmail.com)

1. PyListObject 对象

元素的一个群是一个非常重要的抽象概念，我们可以将符合某一特性的一堆元素聚集为一个群，当然，还要可以向群中添加或删除元素。这样的群的概念对于编程语言十分重要，C 语言就内建了数组的概念，随着编程语言的发展，现在所有的编程语言都会在语言中或标准库中实现这样的群的概念。而且群的概念还进一步地细分为多种实现方式，比如 map, vector 等。每一种实现都为某种目的的元素聚集或元素访问提供了极大的方便。

PyListObject 是 Python 提供的对列表的抽象，熟悉 C++ 的人可能会条件反射地将 PyListObject 与 C++ 中的 list 对应起来（至少它们名字是相同的：）。而实际上，Python 中的列表和 C++ 的 STL 中的 vector 更为相似。

PyListObject 对象可以有效地支持插入，添加，删除等操作，在 Python 的列表中，无一例外地存放的都是 PyObject 的指针。所以实际上，你可以这样看待 Python 中的列表：vector<PyObject*>。

很显然，PyListObject 一定是一个变长对象，因为不同的 list 中存储的元素个数会是不同的。但是，和 PyStringObject 不同的是，PyListObject 对象还支持插入删除等操作，可以在运行时动态地调整其所维护的内存，所以，它还是一个可变对象。

下面看一看 PyListObject 的声明：

```
[listobject.h]  
typedef struct {  
    PyObject_VAR_HEAD  
    /* Vector of pointers to list elements. list[0] is ob_item[0],  
    etc. */  
    PyObject **ob_item;  
    int allocated;  
} PyListObject;
```

如我们所想，在 PyListObject 的头，赫然是一个 PyObject_VAR_HEAD，随后是一个 PyObject**，这个指针正是维护了 PyObject* 列表的关键。我们知道在

PyObject_VAR_HEAD 中，有一个 ob_size，而在 PyListObject 的最后，又有一个 allocated，那么这两个变量之间的关系是什么呢？

其实，ob_size 和 allocated 都和内存的管理有关，PyListObject 所采用的内存管理策略和 C++ 中 vector 采取的内存分配策略是一样的。它并不是存了多少东西就申请对应大小的内存，这样的申请策略显然是低效的，因为我们有理由相信，用户选用列表正是为了频繁地插入删除元素。因此，PyListObject 在每一次需要申请内存的时候，会申请一大块内存，这是申请的总内存的大小记录在 allocated 中，而这些内存中实际被有效的 PyObject* 占用的内存大小被记录在了 ob_size 中。那么不难得到，对于一个 PyListObject 对象，一定存在以下的关系：

```
0 <= ob_size <= allocated
len(list) == ob_size
ob_item == NULL implies ob_size == allocated == 0
```

这里 ob_size 和 allocated 的关系就像 C++ 的 vector 中 size 和 capacity 的关系一样。

Python 源码剖析

——PyListObject 对象(2)

本文作者: Robert Chen (search.pythoner@gmail.com)

2. PyListObject 的创建与维护

2.1 创建

Python 中只提供了唯一一种创建 PyListObject 对象的方法—PyList_New:

```
[listobject.c]
PyObject* PyList_New(int size)
{
    PyListObject *op;
    size_t nbytes;

    nbytes = size * sizeof(PyObject *);
    /* Check for overflow */
    if (nbytes / sizeof(PyObject *) != (size_t)size)
        return PyErr_NoMemory();

    //为 PyListObject 申请空间
    if (num_free_lists) {
        //使用缓冲池
```

```

    num_free_lists--;
    op = free_lists[num_free_lists];
    _PyObject_NewReference((PyObject *)op);
} else {
    //缓冲池中沒有可用的对象，创建对象
    op = PyObject_GC_New(PyListObject, &PyList_Type);
}
//为 PyListObject 对象中维护的元素列表申请空间
if (size <= 0)
    op->ob_item = NULL;
else {
    op->ob_item = (PyObject **) PyMem_MALLOC(nbytes);
    memset(op->ob_item, 0, nbytes);
}
op->ob_size = size;
op->allocated = size;
_PyObject_GC_TRACK(op);
return (PyObject *) op;
}

```

非常清晰的创建过程，首先进行一些例行检查；然后为 `PyListObject` 申请内存空间，创建 `PyListObject` 对象。再成功创建 `PyListObject` 对象之后，就需要为这个对象申请存储 `PyObject*` 的内存空间，内存空间的大小由传入的参数确定，传入的参数决定了新创建的 `PyListObject` 可以容纳多少个 `PyObject*`。最后将 `PyListObject` 对象的存储区域清空，并将 `ob_size` 和 `allocated` 都设置为 `size`，为内存管理做好准备。

我们看到，在 `list` 的实现中，同样用到了缓冲池的技术，创建 `PyListObject` 的时候会首先检查 `free_lists` 中是否还有没有使用的 `PyListObject`，如果有就直接使用，只有在 `free_lists` 中的 `PyListObject` 全部用完之后才会通过 `malloc` 在堆上创建新的 `PyListObject`。`free_lists` 的默认大小为 80，对于一般的小程序而言，基本上只会使用到 `PyListObject` 缓冲池。

```

/* Empty list reuse scheme to save calls to malloc and free */
#define MAXFREELISTS 80
static PyListObject *free_lists[MAXFREELISTS];
static int num_free_lists = 0;

```

有一点非常奇特的是，在 `free_lists` 中缓存的只是 `PyListObject*`，那么这个缓冲池里的 `PyListObject` 究竟是在什么地方被创建的呢？

列位看官，花开两朵，各表一只。我们先把这个问题放一放，看一看，在 `Python` 开始运行后，当第一个 `PyListObject` 对象被创建时的情形。嗯，这有点像上帝创世纪，挺有趣的，不是吗？：)

在第一个 `PyListObject` 创建的时候，我们看到这时 `num_free_lists` 是 0，所以会调用 `PyObject_GC_New` 在堆上创建一个新的 `PyListObject` 对象，假设我们创建的 `PyListObject` 是包含 6 个元素的 `PyListObject`。

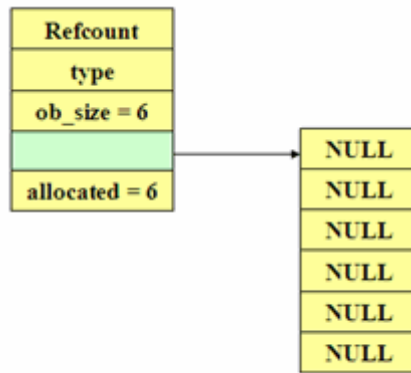


图 1

一个什么东西都没有的 List 当然是很无趣的，我们来尝试向里边添加一点东西，把一个整数对象 100 放到第 3 个位置上去：

```
[listobject.c]
int PyList_SetItem(register PyObject *op, register int i, register
                  PyObject *newitem)
{
    register PyObject *olditem;
    register PyObject **p;
    if (!PyList_Check(op)) {
        .....
    }
    if (i < 0 || i >= ((PyListObject *)op) -> ob_size) {
        Py_XDECREF(newitem);
        PyErr_SetString(PyExc_IndexError,
                        "list assignment index out of range");
        return -1;
    }
    p = ((PyListObject *)op) -> ob_item + i;
    olditem = *p;
    *p = newitem;
    Py_XDECREF(olditem);
    return 0;
}
```

先是进行类型检查，然后进行索引的有效性检查，当一切都 OK 后，将待加入的 PyObject 指针放到指定的位置，然后将这个位置原来存放的对象的引用计数减 1。这里的 olditem 很可能是 NULL，比如向一个新创建的 PyListObject 对象加入元素，就会碰到这样的情况，所以这里必须使用 Py_XDECREF。

好了，现在我们的 PyListObject 对象再不是当年那个一穷二白的可怜虫了：

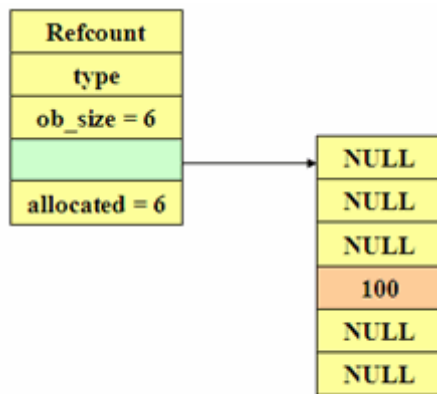


图 2

2.2 添加

接下来我们再试着向这个 PyListObject 中插入一个元素，好吧，就在 100 之前插入 99，99 确实是在 100 之前的，这个地球人都知道：)

```
[listobject.c]
int PyList_Insert(PyObject *op, int where, PyObject *newitem)
{
    .....//类型检查
    return ins1((PyListObject *)op, where, newitem);
}
```

```
static int ins1(PyListObject *self, int where, PyObject *v)
{
    int i, n = self->ob_size;
    PyObject **items;
    .....
    if (list_resize(self, n+1) == -1)
        return -1;

    if (where < 0) {
        where += n;
        if (where < 0)
            where = 0;
    }
    if (where > n)
        where = n;
    items = self->ob_item;
    for (i = n; --i >= where; )
        items[i+1] = items[i];
    Py_INCREF(v);
```

```
items[where] = v;
return 0;
}
```

首先当然是检查指针的有效性，然后是检查当前 `PyListObject` 中已经有多少个元素了，如果这个元素个数已经达到了 `INT_MAX`，那么很遗憾，再不能插入任何元素了。

在通过了检查之后，我们看到，调用了 `list_resize` 函数，从函数名我们可以想象，这个函数改变了 `PyListObject` 所维护的 `PyObject*` 列表的大小：

```
[listobject.c]
static int list_resize(PyListObject *self, int newsize)
{
    PyObject **items;
    size_t new_allocated;
    int allocated = self->allocated;

    /* Bypass realloc() when a previous overallocation is large enough
       to accommodate the newsize.  If the newsize falls lower than half
       the allocated size, then proceed with the realloc() to shrink
       the list.
    */
    if (allocated >= newsize && newsize >= (allocated >> 1)) {
        assert(self->ob_item != NULL || newsize == 0);
        self->ob_size = newsize;
        return 0;
    }

    /* This over-allocates proportional to the list size, making room
       * for additional growth.  The over-allocation is mild, but is
       * enough to give linear-time amortized behavior over a long
       * sequence of appends() in the presence of a poorly-performing
       * system realloc().
       * The growth pattern is: 0, 4, 8, 16, 25, 35, 46, 58, 72, 88, ...
    */
    new_allocated = (newsize >> 3) + (newsize < 9 ? 3 : 6) + newsize;
    if (newsize == 0)
        new_allocated = 0;
    items = self->ob_item;
    if (new_allocated <= ((~(size_t)0) / sizeof(PyObject *)))
        PyMem_RESIZE(items, PyObject *, new_allocated);
    else
        items = NULL;
}
```

```

if (items == NULL) {
    PyErr_NoMemory();
    return -1;
}
self->ob_item = items;
self->ob_size = newsize;
self->allocated = new_allocated;
return 0;
}

```

插入的时候，Python 分四种情况处理：

1. `newsize > ob_size && newsize < allocated` : 简单调整 `ob_size` 值。
2. `newsize < ob_size && newsize > allocated/2` : 简单调整 `ob_size` 值。
3. `newsize < ob_size && newsize < allocated/2` : 调用 `realloc`，重新分配空间。
4. `newsize > ob_size && newsize > allocated` : 调用 `realloc`，重新分配空间。

可以看出，Python 对内存可谓是殚精竭虑了，在某种 `newsize < ob_size` 的情况下还会重新申请内存空间。

将 `PyListObject` 的空间调整之后，接着就应该搬动元素了，才好挪出一个位置，插入我们想要插入的元素。在 Python 中，list 支持一个很有趣的特性，就是负值索引，比如一个 `n` 个元素的 list，`lst[n]`，那么 `lst[-1]` 就是 `lst[n-1]`。这一点在插入元素时得到了体现。

```

static int ins1(PyListObject *self, int where, PyObject *v)
{
    .....
    if (where < 0) {
        where += n;
        if (where < 0)
            where = 0;
    }
    if (where > n)
        where = n;
    items = self->ob_item;
    for (i = n; --i >= where; )
        items[i+1] = items[i];
    Py_INCREF(v);
    items[where] = v;
    return 0;
}

```

可以看到，不管你插入在什么位置，对于 Python 来说，都是合法的，它会自己调整插入的位置。在确定了插入的位置之后，会将插入点之后的所有元素向下挪动一个位置，这样，在插入点就能空出一个位置来，于是大功告成，我们想插入的元素有了容身之地了。

熟悉 C++ 的朋友一定看出来了，这种处理插入的方法实际上与 C++ 中的 `vector` 完全一致。

实际上，Python 中的 PyListObject 与 C++ 中的 vector 非常相似，相反，它和 C++ 中的 list 却是大相径庭的。

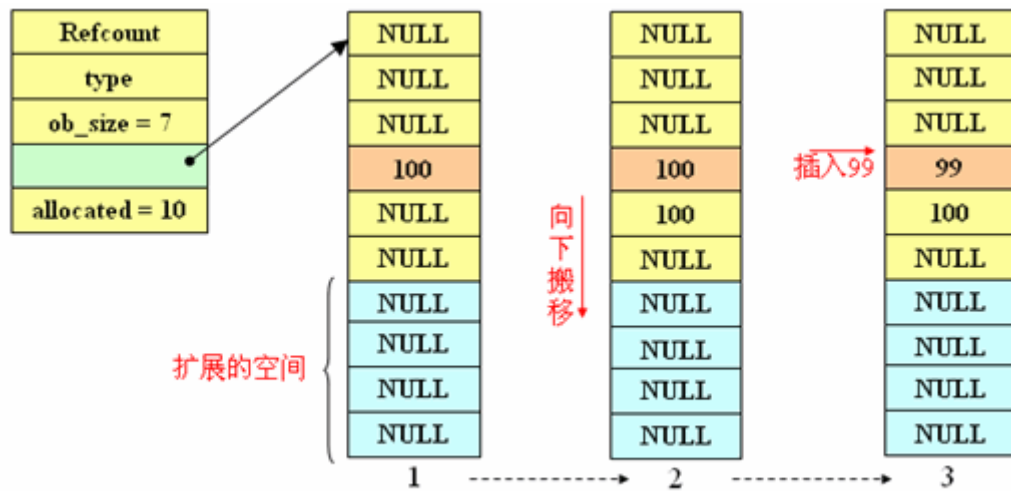


图 3

值得注意的是，通过与 vector 类似的内存管理机制，现在，PyListObject 的 allocated 已经变成 10 了，而 ob_size 却只有 7。

Python 中，list 有一个被广泛使用的操作 append。这个操作与上面所描述的插入操作非常类似：

```
[listobject.c]
int PyList_Append(PyObject *op, PyObject *newitem)
{
    if (PyList_Check(op) && (newitem != NULL))
        return appl((PyListObject *)op, newitem);
    PyErr_BadInternalCall();
    return -1;
}
```

```
static PyObject* listappend(PyListObject *self, PyObject *v)
{
    if (appl(self, v) == 0)
        Py_RETURN_NONE;
    return NULL;
}
```

```
static int appl(PyListObject *self, PyObject *v)
{
    int n = PyList_GET_SIZE(self);
    .....
    if (list_resize(self, n+1) == -1)
        return -1;
}
```



```
Py_INCREF(v);
PyList_SET_ITEM(self, n, v);
return 0;
}
```

只是需要注意的是，在进行 `append` 动作的时候，添加的元素是添加在第 `ob_size` 个位置上的，而不是第 `allocated` 个位置上。下图展示了 `append` 元素 101 之后的 `PyListObject` 对象：

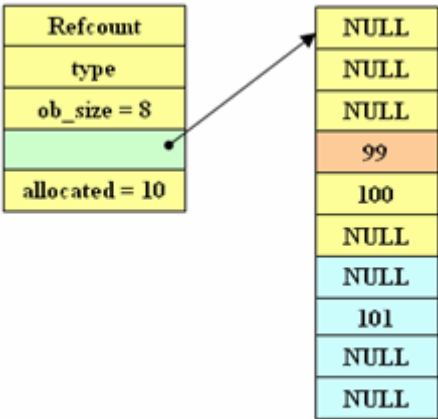


图 4

在 `app1` 中调用 `list_resize` 时，由于 `newsize` (8) 在 7 和 10 之间，所以不需要再分配内存空间。直接将 101 放置在第 8 个位置上即可。

2.3 删除

除了创建，插入这些操作，一个容器至少还应该删除操作，`PyListObject` 自然也不会例外。图 5 展示了一个使用 `PyListObject` 中删除元素功能的例子：

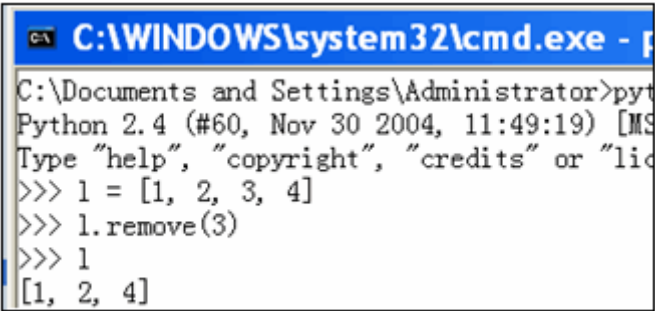


图 5

当在 Python 交互环境中输入 `l.remove(3)` 时，`PyListObject` 中的 `listremove` 操作会被激活：

```
[listobject.c]
static PyObject * listremove(PyListObject *self, PyObject *v)
{
    int i;
    for (i = 0; i < self->ob_size; i++) {
```

```

        int cmp = PyObject_RichCompareBool(self->ob_item[i], v,
Py_EQ);
        if (cmp > 0) {
            if (list_ass_slice(self, i, i+1, (PyObject *)NULL) == 0)
                Py_RETURN_NONE;
            return NULL;
        }
        else if (cmp < 0)
            return NULL;
    }
    PyErr_SetString(PyExc_ValueError, "list.remove(x): x not in
list");
    return NULL;
}

```

在遍历 PyListObject 中所有元素的过程中,将待删除的元素与 PyListObject 中的每个元素一一进行比较,比较操作是通过 PyObject_RichCompareBool 完成的。如果发现了匹配,则调用 list_ass_slice 进行删除操作:

```

[listobject.c]
static int list_ass_slice(PyListObject *a, int ilow, int ihigh,
PyObject *v)
{
    PyObject *recycle_on_stack[8];
    PyObject **recycle = recycle_on_stack; /* will allocate more if
needed */
    PyObject **item;
    int n; /* # of elements in replacement list */
    int norig; /* # of elements in list getting replaced */
    int d; /* Change in size */
#define b ((PyListObject *)v)
    if (v == NULL)
        n = 0;
    else {
        .....
    }

    norig = ihigh - ilow;
    d = n - norig;
    item = a->ob_item;
    //[1]
    s = norig * sizeof(PyObject *);
    if (s > sizeof(recycle_on_stack)) {

```

```

        recycle = (PyObject **)PyMem_MALLOC(s);
        if (recycle == NULL) {
            PyErr_NoMemory();
            goto Error;
        }
    }
    memcpy(recycle, &item[i low], s);

//[2]
    if (d < 0) { /* Delete -d items */
        memmove(&item[i high+d], &item[i high],
                (a->ob_size - i high)*sizeof(PyObject *));
        list_resize(a, a->ob_size + d);
        item = a->ob_item;
    }
    .....
//[3]
    for (k = norig - 1; k >= 0; --k)
        Py_XDECREF(recycle[k]);
#undef b
}

```

当传入的 `v` 是 `NULL` 时，就会进行删除的动作，可以看到，这正是 `listremove` 期望的动作。首先会获得需要删除的元素个数，这是通过 `i high-i low` 得到的，在删除元素这种情况下，这个值显然是 1。在获得了需要删除的元素个数之后，在代码的[2]处，`list_ass_slice` 通过 `memmove` 来达到删除元素的目的。

很明显了，在 `PyListObject` 中，如果是在对象维护的列表中部删除元素的话，一定会引起内存的搬移动作，这一点跟 C++ 中的 `vector` 是完全一致的，而与 C++ 中的 `list` 则完全不同。

图 6 展示了删除元素 100 的过程：

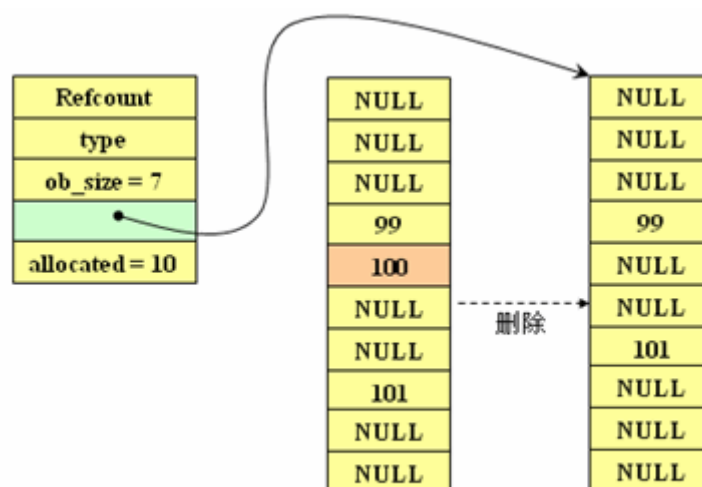


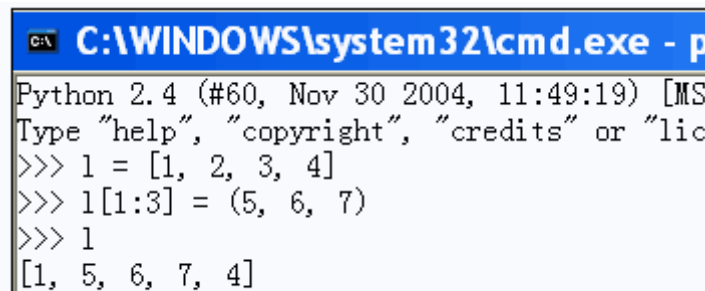
图 6

值得注意的是，实际上，`list_ass_slice` 不仅仅是用做删除元素，它还可以进行插入元素的动作。它的完整功能如下：

```
a[i low:i high] = v if v != NULL.
```

```
del a[i low:i high] if v == NULL.
```

图 7 展示了一个 `list_ass_slice` 进行插入元素操作的例子，有兴趣的朋友可以对 `list_ass_slice` 进行深入研究。



```
C:\WINDOWS\system32\cmd.exe - p
Python 2.4 (#60, Nov 30 2004, 11:49:19) [MS
Type "help", "copyright", "credits" or "lic
>>> l = [1, 2, 3, 4]
>>> l[1:3] = (5, 6, 7)
>>> l
[1, 5, 6, 7, 4]
```

图 7

Python 源码剖析

——PyListObject 对象(3)

本文作者: Robert Chen (search.pythoner@gmail.com)

3. PyListObject 对象缓冲池

还记得吗，刚才我们按下了一个有趣的话题。没错，就是那个缓冲池，`free_list`。现在，是揭开它的神秘面纱的时候呢。我们想知道的问题是：`free_list` 中所缓冲的 `PyListObject` 对象是从哪里获得的，是在何时创建的。答案就在一个 `PyListObject` 被销毁的过程中：

```
[listobject.c]
static void list_dealloc(PyListObject *op)
{
    int i;
    PyObject_GC_UnTrack(op);
    Py_TRASHCAN_SAFE_BEGIN(op)
    if (op->ob_item != NULL) {
        /* Do it backwards, for Christian Tismer.
         * There's a simple test case where somehow this reduces
         * thrashing when a *very* large list is created and
         * immediately deleted. */
        i = op->ob_size;
        while (--i >= 0) {
            Py_XDECREF(op->ob_item[i]);
        }
        PyMem_FREE(op->ob_item);
    }
}
```

```

}
if (num_free_lists < MAXFREELISTS && PyList_CheckExact(op))
    free_lists[num_free_lists++] = op;
else
    op->ob_type->tp_free((PyObject *)op);
Py_TRASHCAN_SAFE_END(op)
}

```

在销毁一个 PyListObject 的时候,当然要做的一件事是为 list 中的每一个元素改变其引用计数。然后,我们就来到了最有趣的部分。Python 会检查我们开始提到的那个缓冲池, free_lists, 查看其中缓存的 PyListObject 的数量是否已经满了, 如果没有, 就将该待删除的 PyListObject 放到缓冲池中, 以备后用。现在一切真相大白了, 那个在 Python 启动是空荡荡的缓冲池原来都是被本应该死去的 PyListObject 对象给填充了, 在以后创建新的 PyListObject 的时候, Python 会首先唤醒这些已经“死去”的 PyListObject。感谢党, 感谢政府, 又给它们一个重新做“人”的机会:) 但是, 需要指出, 这里缓冲的仅仅是 PyListObject 对象, 而没有这个对象曾经拥有的 PyObject*列表, 因为这些 PyObject 指针的引用计数已经减少了, 这些指针所指的对象都要各奔前程, 或生存, 或毁灭, 不再被 PyListObject 所给与的那个引用计数所束缚。PyListObject 如果继续维护一个指向这些对象的指针的列表, 就可能产生悬空指针的问题。所以, PyObject*列表所占用的空间必须归还给系统。

看一下我们刚刚创建的 PyListObject 的最后归宿:

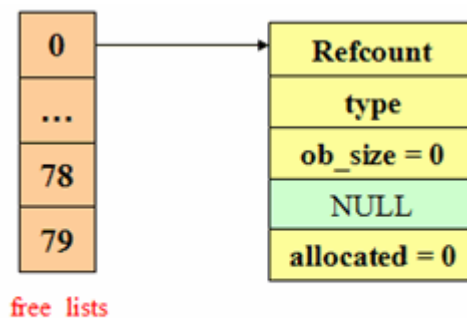


图 8

在下一次创建 PyListObject 时, 这个 PyListObject 将重新被唤醒, 重新分配 PyObject*列表占用的内存, 重新拥抱新的对象。放眼四周, 曾经所拥有过的那些对象, 有的已经容颜苍老, 有的已经烟消云散, 是否有一种“无私人非事事体, 欲语泪先流”的感慨呢?:)

4. Hack PyListObject

首先我们来观察在 PyListObject 中维护的元素数量变化时, PyListObject 中 `ob_size` 和 `allocated` 两个变量的变化情况, 从中窥见 PyListObject 对内存的使用和管理。

在 PyListObject 的输出操作 list_print 中，我们添加了如下代码，以观察 PyListObject 对内存的管理：

```
printf("\nallocated=%d, ob_size=%d\n", op->allocated,
op->ob_size);
```

观察结果如图 9 所示：

```
>>> l = [1]
>>> l
[1]
allocated=1, ob_size=1
```

图 9-1

```
>>> l.append(2)
>>> l
[1, 2]
allocated=5, ob_size=2
```

图 9-2

```
>>> l.append(5)
>>> l
[1, 2, 3, 4, 5]
allocated=5, ob_size=5
```

图 9-3

```
>>> l.remove(3)
Remove object 3
>>> l
[1, 2, 4, 5]
allocated=5, ob_size=4
```

图 9-4

首先创建一个包含一个元素的 list，这时 ob_size 和 allocated 都是 1。这时 list 中拥有的所有内存空间都已被使用完，所以下次插入元素时就一定会调整 list 的内存空间了。

随后向 list 末尾追加元素 2，可以看到，调整内存空间的动作发生了。allocated 变成了 5，而 ob_size 则变成了 2，这里明确地显示出了 PyListObject 所采用的与 C++ 中 vector 一样的内存缓冲池策略。

继续向 list 末尾追加元素 3，4，5，当追加了元素 5 之后，list 所拥有的内存空间又被使用完了，下一次再追加或插入元素时，内存空间调整的动作又会再一次发生。

如果这时在 list 中删除元素 3，可以看到，ob_size 发生了变化，而 allocated 则不发生变化，它始终如一地维护着当前 list 所拥有的全部内存数量。

接下来我们从图 10 的结果中观察一下 PyListObject 对象的创建和删除对于 Python 维护的 PyListObject 对象缓冲池的影响。

```
list1 = [1]
print list1
list2 = [1]
print list1
list3 = [1]
print list1
del list2
print list1
del list3
print list1
```

```
[1]
num_free_lists=3
[1]
num_free_lists=2
[1]
num_free_lists=1
[1]
num_free_lists=2
[1]
num_free_lists=3
```

图 10

这次为了消除 Python 交互环境执行时对 PyListObject 对象缓冲池的影响，我们通过执行 py 脚本来观察，可以看到，当创建新的 PyListObject 对象时，如果缓冲池中有可用的 PyListObject 对象，则会使用缓冲池中的对象；而在销毁一个 PyListObject 对象时，确实将这个对象放到了缓冲池中。

Python 源码剖析

——字典对象 PyDictObject(1)

本文作者: Robert Chen (search.pythoner@gmail.com)

1 散列表概述

元素和元素之间通常可能存在某种联系,这种神秘的联系使本来绝不相同的两个元素捆绑在一起,而别的元素则被排斥在外。比如对应于“2 倍”,这样一种联系,6 和 3 就是这样的两个元素,而 4 和 2 同样也是被这种联系关联起来的一对元素。

为了刻画某种对应关系,现代的编程语言通常都在语言级或标准库中提供某种关联式的容器。关联式的容器中存储着一对对符合该容器所代表的关联规则的元素对。关联式容器中的元素对通常是以(键 key, 值 value)这样的形式存在。比如在一个表示“2 倍”关系的关联容器中(3, 6), (2, 4)就是容器中的两个元素对。其中 3 就是一个“键”,当寻找到 3 之后,我们很轻松地就能获得与 3 有着“2 倍”联系的另一元素。

关联容器的设计总会极大地关注搜索键的效率,因为通常我们使用关联容器,都是希望根据我们手中已有的某个元素来快速获得与之有某种联系的另一元素。一般而言,关联容器的实现都会基于设计良好的数据结构。比如 C++ 的 STL 中的 map 就是一种关联容器, map 的实现基于 RB-tree(红黑树)。RB-tree 是一种平衡二元树,能够提供良好的搜索效率,理论上,其搜索的复杂度为 $O(\log N)$ 。

Python 中同样提供关联式容器,即 PyDictObject 对象。与 map 不同的是, PyDictObject 对搜索的效率要求及其苛刻,这也是因为 PyDictObject 在 Python 本身的实现中被大量地采用,比如会通过 PyDictObject 来建立 Python 字节码的运行环境,其中会存放(变量名, 变量值)的元素对,通过查找变量名获得变量值。因此, PyDictObject 没有如 map 一样采用平衡二元树,而是采用了散列表(hash table),因为理论上,在最优情况下,散列表能提供 $O(1)$ 复杂度的搜索效率。

散列表的基本思想是通过一定的函数将需搜索的键值映射为一个整数,将这个整数视为索引值去访问某片连续的内存区域。看一个简单的例子,如图 1 所示,有 10 个整数 1, 2,, 10。其依次对应 a, b,, j。申请一块连续内存,并依次存储 a, b,, j:

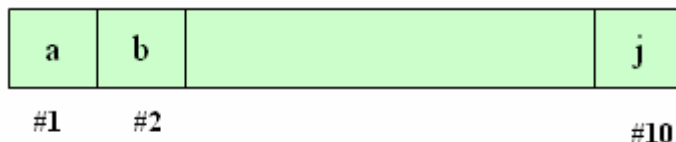


图 1

当需要寻找与 2 对应的字母时,只需通过一定的函数将其映射为整数,显然,2 本身就是一个整数,我们可以使用 2 自身。然后访问这片连续内存的第 2 个位置,就能得到与 2 对应的字母 b。

将元素映射为整数的过程对于散列表来说尤为关键,用于映射的函数成为散列函数(hash function),而映射后的值称为元素的散列值(hash value)。

在使用散列表的过程中，不同的对象经过散列函数的作用，可能被映射为相同的散列值。而且随着需要存储的数据的增多，这样的冲突就会发生得越来越频繁。散列冲突是散列技术与生俱来的问题。

有很多种方法可以用来解决产生的散列冲突，比如说开链法，这是 SGI STL 中的 hash table 所采用的方法；而 Python 中所采用的是另一种方法，即开放定址法。

当产生散列冲突时，Python 会通过一个再次探测函数，计算下一个候选位置，如果这个位置可用，则可将待插入元素放到这个位置；如果这个位置不可用，则 Python 会再次通过探测函数，获得下一个候选位置，如此不断探测，总会找到一个可用的位置。

这样，沿着探测函数，从一个位置出发可以依次到达多个位置，这些位置形成了一个探测链。当需要删除某条探测链上的某个元素时，问题就产生了。假如这条链的首元素位置为 a，尾元素的位置为 c，现在需要删除中间的某个位置 b 上的元素。如果直接将位置 b 上的元素删除，则会导致探测链的断裂，引起严重的问题。想象一下，在下一次搜索 c 时，会从位置 a 开始，通过探测函数，沿着探测链，一步一步向位置 c 靠近，但是在到达位置 b 时，发现这个位置上的元素不属于这个探测链，因此会以为探测链在这里结束，导致不能达到为止 c，自然也不能搜索到位置 c 上的元素，所以结果是搜索失败。而实际上，待搜索的元素确实存在于散列表中。

所以，在采用开放定址的冲突解决策略的散列表中，删除某条探测链上的元素时不能真正地删除，而是进行一种“伪删除”操作，必须要让该元素还存在于探测链上，担当承前启后的重任。对于这种伪删除技术，我们在分析 Python 中的关联容器时会详细讨论。

2 PyDictObject

在此后的描述中，我们将把关联容器中的一个（键，值）元素对称为一个 entry 或 slot。在 Python 中，一个 entry 的定义如下：

```
[dictobject.h]
typedef struct {
    long me_hash;          /* cached hash code of me_key */
    PyObject *me_key;
    PyObject *me_value;
} PyDictEntry;
```

可以看到，在 PyDictObject 中其实存放的都是 PyObject*，这也是 Python 中的 dict 什么都能装得下的原因，因为在 Python 中，不论什么都是一个 PyObject 对象。

me_hash 域中存储的是 me_key 的散列值，利用一个域来记录这个散列值可以避免每次查询的时候都要重新计算一遍散列值。

在 Python 中，在一个 PyDictObject 对象生存变化的过程中，其中的 entry 还会在不同的状态间转换。PyDictObject 中 entry 可以在三种状态间转换：Unused 态，Active 态，Dummy 态。

1. 当一个 entry 的 me_key 和 me_value 都是 NULL 时，entry 处于 Unused 态。表明该 entry 中并没有存储 (key, value) 对，而且在此之前，也没有存储过。每一个 entry 在初始化的时候都会出于这种状态，而且只有在 Unused 态下，entry 的 me_key 域才会为 NULL。

2. 当 entry 中存储了一个 (key, value) 对时，entry 便转换到了 Active 态。在 Active 态下，me_key 和 me_value 都不能为 NULL。更进一步，me_key 不能是 dummy。

3. 当 entry 中存储的 (key, value) 对被删除后，entry 中的 me_key 将指向 dummy，entry 进入 Dummy 态。这是一种惰性删除技巧，是为了保证冲突的探测序列不会在这里被中断。

图 2 展示了三种状态以及它们之间的转换关系：

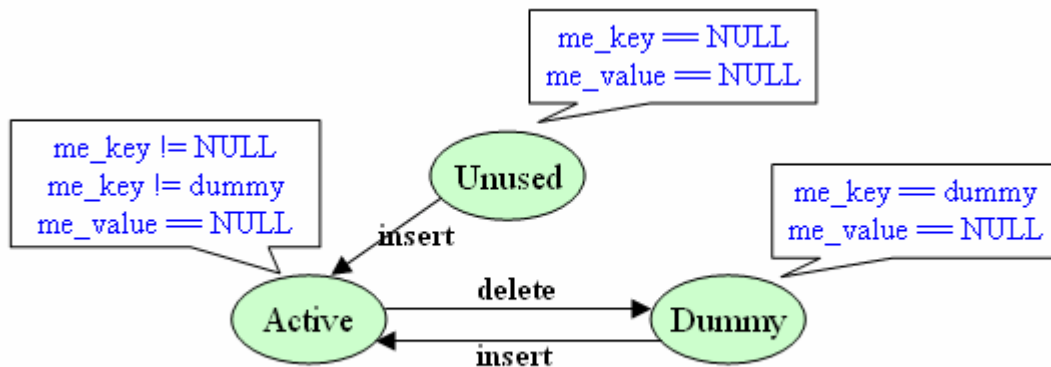


图 2

一个 PyDictObject 对象实际上是一大堆 entry 的集合，总控这些集合的结构如下：

```
[dictobject.h]
#define PyDict_MINSIZE 8
typedef struct _dictobject PyDictObject;
struct _dictobject {
    PyObject_HEAD
    int ma_fill; /* # Active + # Dummy */
    int ma_used; /* # Active */
    int ma_mask;
    PyDictEntry *ma_table;
    PyDictEntry *(*ma_lookup)(PyDictObject *mp, PyObject
*key, long hash);
    PyDictEntry ma_smalltable[PyDict_MINSIZE];
};
```

从注释中可以清楚地看到，ma_fill 域中维护着从 PyDictObject 创建到现在曾经以及正处于 Active 态的 entry 个数，而 ma_used 则维护着当前正处于 Active 态的 entry。

当创建一个 PyDictObject 对象时，至少有 PyDict_MINSIZE 个 entry 被同时创建。在 dictobject.h 中，这个值被设定为 8，这个值被认为是通过大量的实验得出的最佳值。既不会太浪费内存空间，又能很好地满足 Python 中大量使用 PyDictObject 的环境的需求，而不需要再次调用 malloc 申请内存空间。当然，我们可以自己调节这个值来调节 Python 的运行效率。

当一个 PyDictObject 是一个比较小的 dict 时，即 entry 数量少于 8 个，ma_table 域将指向这与生俱来的 8 个 entry。而当 PyDictObject 中的 entry 数量超过 8 个时，Python 认为这家伙是一个大 dict 了，将会额外申请内存空间，并将 ma_table 指向这块空间。这样，无论何时，ma_table 域总不会为 NULL，这带来了一个好处，不用再运行时一次又一次不厌其烦地检查 ma_table 的有效性，因为 ma_table 总是有效的。

PyDictObject 中的 ma_mask 实际上记录了一个 PyDictObject 对象中有 entry 的数量。至于这家伙为什么不叫 ma_size 这么好听的名字，偏偏要叫 ma_mask 这个莫名其妙的名字，此乃后话，这里先按下不表。同样被按下不表的还有 ma_lookup :))

Python 源码剖析

——字典对象 PyDictObject(2)

本文作者: Robert Chen (search.pythoner@gmail.com)

3 PyDictObject 的创建和维护

3.1.1 PyDictObject 对象创建

```
[dictobject.c]
typedef PyDictEntry dictentry;
typedef PyDictObject dictobject;
```

```
#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 dictobject *mp;
    if (dummy == NULL) { /* Auto-initialize dummy */
        dummy = PyString_FromString("<dummy key>");
        if (dummy == NULL)
            return NULL;
    }
    if (num_free_dicts)
    {
        ..... //使用缓冲池
    }
    else
    {
        mp = PyObject_GC_New(dictobject, &PyDict_Type);
        if (mp == NULL)
            return NULL;
        EMPTY_TO_MINSIZE(mp);
    }
    mp->ma_lookup = lookdict_string;
    _PyObject_GC_TRACK(mp);
    return (PyObject *)mp;
}

```

值得注意的是，在第一次调用 `PyDict_New` 时，会创建在前面提到的那个 `dummy` 对象。显而易见，`dummy` 对象仅仅是一个 `PyStringObject` 对象，它作为一种指示标志，表明该 `entry` 曾被使用过，且探测序列下一个位置的 `entry` 有可能是有效的，从而防止探测序列中断。

从 `num_free_dicts` 可以看出，Python 中 `dict` 的实现同样使用了缓冲池。我们把将缓冲池的讨论放到后边。

创建的过程首先申请合适的内存空间，然后在 `EMPTY_TO_MINSIZE` 中，会将 `ma_smalltable` 清零，同时设置 `ma_size` 和 `ma_fill`，当然，在一个 `PyDictObject` 对象刚被创建的时候，这两个变量都应该是 0。然后会将 `ma_table` 指向 `ma_smalltable`，并设置 `ma_mask`，可以看到，`ma_mask` 确实与一个 `PyDictObject` 对象中 `entry` 的数量有关。在创建过程的最后，将 `lookdict_string` 赋给了

ma_lookup。正是 ma_lookup 指定了 PyDictObject 在 entry 集合中搜索某一特定 entry 时要进行的动作，它是 PyDictObject 的搜索策略，万众瞩目。

3.1.2 元素搜索

PyDictObject 引入了两个不同的搜索策略，lookdict 和 lookdict_string。实际上，这两个策略使用的是相同的算法，lookdict_string 只是 lookdict 的一种针对 PyStringObject 对象的特化形式。以 PyStringObject 对象作为 PyDictObject 对象中 entry 的键在 Python 中是如此地广泛和重要，所以 lookdict_string 也就成为了 PyDictObject 创建时所默认采用的搜索策略：

```
[dictobject.c]
static dictentry* lookdict_string(dictobject *mp, PyObject
*key, register long hash)
{
    register int i;
    register unsigned int perturb;
    register dictentry *freeslot;
    register unsigned int mask = mp->ma_mask;
    dictentry *ep0 = mp->ma_table;
    register dictentry *ep;
```

```
    if (!PyString_CheckExact(key)) {
        mp->ma_lookup = lookdict;
        return lookdict(mp, key, hash);
    }
    //[1]
    i = hash & mask;
    ep = &ep0[i];
```

```
    //[2]
    //if NULL or interned
    if (ep->me_key == NULL || ep->me_key == key)
        return ep;

    //[3]
    if (ep->me_key == dummy)
        freeslot = ep;
    else
    {
        //[4]
```

```

        if (ep->me_hash == hash && _PyString_Eq(ep->me_key,
key))
        {
            return ep;
        }
        freeslot = NULL;
    }

```

```

/* In the loop, me_key == dummy is by far (factor of 100s)
the
least likely outcome, so test for that last. */
for (perturb = hash; ; perturb >>= PERTURB_SHIFT)
{
    i = (i << 2) + i + perturb + 1;
    ep = &ep0[i & mask];
    if (ep->me_key == NULL)
        return freeslot == NULL ? ep : freeslot;
    if (ep->me_key == key
        || (ep->me_hash == hash
            && ep->me_key != dummy
            && _PyString_Eq(ep->me_key, key)))
        return ep;
    if (ep->me_key == dummy && freeslot == NULL)
        freeslot = ep;
}
}

```

其中的[1], [2], [3], [4]标注出了搜索过程中的关键步骤, 这些步骤会在后面讲述 PyDictObject 对象的一般搜索策略时详细讨论。

lookdict_string 并不是 PyDictObject 中最一般的搜索策略, 它是一种有条件限制的搜索策略。lookdict_string 背后有一个假设, 即 PyDictObject 对象中每一个 entry 的 key 都是 PyStringObject*。只有在这种假设成立的情况下, lookdict_string 才会被使用。可以看到, lookdict_string 首先会检查需要搜索的 key 是否严格对应一个 PyStringObject 对象, 只有在检查通过后, 才会进行下面的动作; 如果检查不通过, 那么就会转向 PyDictObject 中的通用搜索策略 lookdict:

```

[dictobject.c]
static dictentry* lookdict(dictobject *mp, PyObject *key,
register long hash)
{
    register int i;
    register unsigned int perturb;
    register dictentry *freeslot;

```

```
register unsigned int mask = mp->ma_mask;
dictentry *ep0 = mp->ma_table;
register dictentry *ep;
register int restore_error;
register int checked_error;
register int cmp;
PyObject *err_type, *err_value, *err_tb;
PyObject *startkey;
```

```
//[1]
i = hash & mask;
ep = &ep0[i];
//[2]
if (ep->me_key == NULL || ep->me_key == key)
    return ep;
```

```
//[3]
if (ep->me_key == dummy)
    freeslot = ep;
else
{
    //[4]
    if (ep->me_hash == hash)
    {
        startkey = ep->me_key;
        cmp = PyObject_RichCompareBool(startkey, key,
Py_EQ);
        if (cmp < 0)
            PyErr_Clear();
        if (ep0 == mp->ma_table && ep->me_key == startkey)
        {
            if (cmp > 0)
                goto Done;
        }
        else
        {
            ep = lookdict(mp, key, hash);
            goto Done;
        }
    }
    freeslot = NULL;
}
```

```

    . . . . .
Done:
    return ep;
}

```

PyDictObject 中维护的 entry 的数量是有限的，比如 100 个或 1000 个。而传入 lookdict 中的 key 的 hash 值却不一定会在这个范围内，所以这就要求 lookdict 将 hash 值映射到某个 entry 上去。Lookdict 采取的策略非常简单，直接将 hash 值与 entry 的数量做一个与操作，结果自然落在 entry 的数量之下。由于 ma_mask 会被用来进行大量的与操作，所以这个与 entry 数量相关的变量被命名为 ma_mask，而不是 ma_size。

我们注意到，lookdict 永远都不会返回 NULL，如果在 PyDictObject 中搜索不到待查找的元素，同样会返回一个 entry，这个 entry 的 me_value 为 NULL。

在搜索的过程中 freeslot 是一个重要的变量。如果在探测序列中的某个位置上，entry 处于 Dummy 态，那么如果在这个序列中搜索不成功，就会返回这个处于 Dummy 态的 entry。我们知道，处于 Dummy 态的 entry 其 me_value 是为 NULL 的，所以这个返回结果指示了搜索失败；同时，返回的 entry 也是一个可以立即被使用的 entry，因为 Dummy 态的 entry 并没有维护一个有效的 (key, value) 对。这个 freeslot 正是用来指向探测序列中第一个处于 Dummy 态的 entry，如果搜索失败，freeslot 就会挺身而出，提供一个能指示失败并立即可用的 entry。当然，如果探测序列中并没有 Dummy 态 entry，搜索失败时，一定是在一个处于 Unused 态的 entry 上结束搜索过程的，这时会返回这个处于 Unused 态的 entry，同样是一个能指示失败且立即可用的 entry。

下面是 lookdict 中进行第一次检查时需要注意的动作：

- [1]: 根据 hash 值获得 entry 的序号。
- [2]: 如果 ep->me_key 为 NULL，且与 key 相同，搜索失败。
- [3]: 若当前 entry 处于 Dummy 态，设置 freeslot。
- [4]: 检查当前 Active 的 entry 中的 key 与待查找的 key 是否相同，如果相同，则立即返回，搜索成功。

在[4]中，需要注意那个 PyObject_RichCompareBool，它的函数原形为：

```

int PyObject_RichCompareBool(PyObject *v, PyObject *w, int
op)

```

当(v op w)成立时，返回 1；当(v op w)不成立时，返回 0；如果在比较中发生错误，则返回-1。

现在我们考察的是根据 hash 值获得的第一个 entry 与待查找的元素的比较。实际上，由于对应于某一个散列值，几乎都有一个探测序列与之对应，所以我们现在只是考察了探测序列中第一个位置的 entry。万里长征仅仅迈出了第一步。

如果第一个 entry 与待查找的 key 不匹配，那么很自然地，lookdict 会沿着探测序列，顺藤摸瓜，依次比较探测序列上的 entry 与待查找的 key：

```

[dictobject.c]
static dictentry* lookdict(dictobject *mp, PyObject *key,
register long hash)
{

```



```

register int i;
register unsigned int perturb;
register dictentry *freeslot;
register unsigned int mask = mp->ma_mask;
dictentry *ep0 = mp->ma_table;
register dictentry *ep;
register int restore_error;
register int checked_error;
register int cmp;
PyObject *err_type, *err_value, *err_tb;
PyObject *startkey;
. . . . .
for (perturb = hash; ; perturb >>= PERTURB_SHIFT)
{
    //[5]
    i = (i << 2) + i + perturb + 1;
    ep = &ep0[i & mask];

    //[6]
    if (ep->me_key == NULL)
    {
        if (freeslot != NULL)
            ep = freeslot;
        break;
    }
    if (ep->me_key == key) //[7]
        break;
    if (ep->me_hash == hash && ep->me_key != dummy)
    {
        startkey = ep->me_key;
        cmp = PyObject_RichCompareBool(startkey, key,
Py_EQ);
        if (cmp < 0)
            PyErr_Clear();
        if (ep0 == mp->ma_table && ep->me_key == startkey)
        {
            if (cmp > 0)
                break;
        }
        else {
            ep = lookdict(mp, key, hash);
            break;
        }
    }
}

```

```

        //[8]
        else if (ep->me_key == dummy && freeslot == NULL)
            freeslot = ep;
    }

```

```

Done:
    return ep;
}

```

[5]: 获得探测序列中的下一个待探测的 entry。

[6]: ep 到达一个 Unused 态 entry，表明搜索结束。这是如果 freeslot 不为空，则返回 freeslot 所指 entry。

[7]: entry 与待查找的 key 匹配，搜索成功。

[8]: 在探测序列中发现 Dummy 态 entry，设置 freeslot。

到这里，我们已经清晰地了解了 PyDictObject 中的搜索策略。再回过头去看看那个 lookdict_string，可以很清晰地看到，lookdict_string 实际上就是一个 lookdict 对于 PyStringDict 对象的优化版本。在这里展示的 lookdict 代码经过了删节，实际上，在 lookdict 中有许多捕捉错误并处理错误的代码，因为 lookdict 面对的是 PyObject*，所以会出现很多意外情况。而在 lookdict_string 中，完全没有了这些处理错误的代码。而另一方面，在 lookdict 中，使用的是非常通用的 PyObject_RichCompareBool，而 lookdict_string 使用的是_PyString_Eq，比要简单很多，这些因素使得 lookdict_string 的搜索效率要比 lookdict 高很多。

此外，Python 自身也大量使用了 PyDictObject 对象，用来维护一个作用域中变量名和变量值之间的对应关系，或是用来在为函数传递参数时维护参数名与参数值的对应关系。这些对象几乎都是用 PyStringObject 对象作为 entry 中的 key，所以 lookdict_string 的意义就显得非常重要了，它对 Python 整体的运行效率都有着重要的影响。

3.1.3 插入与删除

PyDictObject 对象中元素的插入动作建立在搜索的基础之上：

```

[dictobject.c]
static void
insertdict(register dictobject *mp, PyObject *key, long hash,
PyObject *value)
{
    PyObject *old_value;
    register dictentry *ep;

    ep = mp->ma_lookup(mp, key, hash);
    //[1]
    if (ep->me_value != NULL) {
        old_value = ep->me_value;
    }
}

```

```

    ep->me_value = value;
    Py_DECREF(old_value); /* which **CAN** re-enter */
    Py_DECREF(key);
}
//[2]
else {
    if (ep->me_key == NULL)
        mp->ma_fill++;
    else
        Py_DECREF(ep->me_key);
    ep->me_key = key;
    ep->me_hash = hash;
    ep->me_value = value;
    mp->ma_used++;
}
}

```

前面我们提到了，搜索操作在成功时，返回相应的处于 Active 态的 entry，而在搜索失败时会返回两种不同的结果：一是处于 Unused 态的 entry；二是处于 Dummy 态的 entry。那么插入操作对应不同的 entry，所需要进行的动作显然也是不一样的。对于 Active 的 entry，只需要简单地替换 me_value 值就可以了；而对于 Unused 或 Dummy 的 entry，则需要完整地设置 me_key, me_hash 和 me_value。

在 insertdict 中，正是根据搜索的结果采取了不同的动作：

[1]：搜索成功，返回处于 Active 的 entry，直接替换 me_value。

[2]：搜索失败，返回 Unused 或 Dummy 的 entry，完整设置 me_key, me_hash 和 me_value。

在 Python 中，对 PyDictObject 中的元素进行插入或设置有两种方式：

```

[python code]
d = {}
d[1] = 1
d[1] = 2

```

第二行 Python 代码是在 PyDictObject 对象中没有这个 entry 的情况下插入元素，第三行是在 PyDictObject 对象中已经有这个 entry 的情况下重新设置元素。可以看到，insertdict 完全可以适应这两种情况，在 insertdict 中，[2]处代码处理第二行 Python 代码，[1]处代码处理第三行 Python 代码。实际上，这两行 Python 代码也确实都调用了 insertdict。

当这两行设置 PyDictObject 对象元素的 Python 代码被执行时，并不是直接就调用 insertdict，因为观察代码可以看到，insertdict 需要一个 hash 值作为调用参数，那么这个 hash 值在什么地方获得的呢？实际上，在调用 insertdict 之前，还会调用 PyDict_SetItem：

```

[dictobject.c]

```

```

int PyDict_SetItem(register PyObject *op, PyObject *key,
PyObject *value)
{
    register dictobject *mp;
    register long hash;
    register int n_used;

```

```

    mp = (dictobject *)op;

    //计算 hash 值

    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;
    }

    n_used = mp->ma_used;
    Py_INCREF(value);
    Py_INCREF(key);
    insertdict(mp, key, hash, value);

    if (!(mp->ma_used > n_used && mp->ma_fill*3 >=
(mp->ma_mask+1)*2))
        return 0;
    return dictresize(mp, mp->ma_used*(mp->ma_used>50000 ?
2 : 4));
}

```

在 PyDict_SetItem 中，会首先获得 key 的 hash 值，在上面的例子中，也就是一个 PyIntObject 对象 1 的 hash 值。

然后再调用 insertdict 进行元素的插入或设置。

PyDict_SetItem 在插入或设置元素的动作结束之后，并不会草草返回了事。接下来，它会检查是否需要改变 PyDictObject 内部 ma_table 所值的内存区域的大小，在以后的叙述中，我们将这块内存称为“table”。那么什么时候需要改变 table 的大小呢。在前面我们说过，如果 table 的装载率大于 2/3 时，后续的插入动作遭遇到冲突的可能性会非常大。所以装载率是否大于或等于 2/3 就是判断是否需要改变 table 大小的准则。在 PyDict_SetItem 中，有如下的代码：

```
if (!(mp->ma_used > n_used && mp->ma_fill*3 >=
(mp->ma_mask+1)*2))
    return 0;
```

经过转换，实际上可以得到：

```
(mp->ma_fill)/(mp->ma_mask+1) >= 2/3
```

这个等式左边的表达式正是装载率。

然而装载率只是判定是否需要改变 table 大小的一个标准，还有一个标准是在 insertdict 的过程中，是否使用了一个处于 Unused 态的 entry。前面我们说过，在搜索过程失败并且探测序列中没有 Dummy 态的 entry 时，就会返回一个 Unused 态的 entry，insertdict 会对这个 entry 进行填充。只有当这种情况发生并且装载率超标时，才会进行改变 table 大小的动作。而判断在 insertdict 的过程中是否填充了 Unused 态的 entry，是通过

```
mp->ma_used > n_used
```

来判断的，其中的 n_used 就是进行 insertdict 操作之前的 mp->ma_used。通过观察 mp->ma_used 是否改变，就可以知道是否有 Unused 态的 entry 被填充。

在改变 table 时，并不一定是增加 table 的大小，同样也可能是减小 table 的大小。更改 table 的大小时，新的 table 的空间为：

```
mp->ma_used*(mp->ma_used>50000 ? 2 : 4)
```

如果一个 PyDictObject 对象的 table 中只有几个 entry 处于 Active 态，而大多数 entry 都处于 Dummy 态，那么改变 table 大小的结果显然就是减小了 table 的空间大小。

在确定新的 table 的大小时，通常选用的策略是时新的 table 中 entry 的数量是现在 table 中 Active 态 entry 数量的 4 倍，选用 4 倍是为了使 table 中处于 Active 态的 entry 的分布更加稀疏，减少插入元素时的冲突概率。当然，这是以内存空间为代价的。由于机器的内存总是有限的，Python 总不能没心没肺地在任何时候都要求 4 倍空间，这样搞，别的程序会有意见的：）所以当 table 中 Active 态的 entry 数量非常巨大时，Python 只会要求 2 倍的空间，这次又是以执行速度来交换内存空间。Python2.4.1 将这个“非常巨大”的标准划定在 50000。如此一来，上帝的归上帝，撒旦的归撒旦，万事大吉：）

至于具体的改变 table 大小的重任，则交到了 dictresize 一人的肩上：

```
[dictobject.c]
static int dictresize(dictobject *mp, int minused)
{
    int newsize;
    dictentry *oldtable, *newtable, *ep;
    int i;
    int is_oldtable_malloced;
    dictentry small_copy[PyDict_MINSIZE];
    //[1]
    for(newsize = PyDict_MINSIZE; newsize <= minused &&
newsize > 0; newsize <= 1)
        ;
    oldtable = mp->ma_table;
```

```
assert(oldtable != NULL);
is_oldtable_malloced = oldtable != mp->ma_smalltable;
```

```
//[2]
if (newsize == PyDict_MINSIZE) {
    newtable = mp->ma_smalltable;
    if (newtable == oldtable) {
        if (mp->ma_fill == mp->ma_used) {
            //没有任何 Dummy 态 entry, 直接返回
            return 0;
        }

        //将 oldtable 拷贝, 进行备份
        assert(mp->ma_fill > mp->ma_used);
        memcpy(small_copy, oldtable,
sizeof(small_copy));
        oldtable = small_copy;
    }
}
else {
    newtable = PyMem_NEW(dictentry, newsize);
}
```

```
//[3]
assert(newtable != oldtable);
mp->ma_table = newtable;
mp->ma_mask = newsize - 1;
memset(newtable, 0, sizeof(dictentry) * newsize);
mp->ma_used = 0;
i = mp->ma_fill;
mp->ma_fill = 0;
```

```
//[4]
for (ep = oldtable; i > 0; ep++) {
    if (ep->me_value != NULL) { /* active entry */
        --i;
        insertdict(mp, ep->me_key, ep->me_hash,
ep->me_value);
    }
    else if (ep->me_key != NULL) { /* dummy entry */
```

```

        --i;
        assert(ep->me_key == dummy);
        Py_DECREF(ep->me_key);
    }
}
if (is_oldtable_mallocated)
    PyMem_DEL(oldtable);
return 0;
}

```

[1] : dictresize 首先会确定新的 table 的大小，很显然，这个大小一定要大于传入的参数 minused，这也是在原来的 table 中处于 Active 态的 entry 的数量。dictresize 从 8 开始，以指数方式增加大小，直到超过了 minused 为止。所以实际上新的 table 的大小在大多数情况下至少是原来 table 中 Active 态 entry 数量的 4 倍。

[2] : 如果在[1]中获得的新的 table 大小为 8，则不需要在堆上分配空间，直接使用 ma_smalltable 就可以了；否则，则需要在堆上分配空间。

[3] : 对新的 table 进行初始化，并调整原来 PyDictObject 对象中用于维护 table 使用情况的变量。

[4] : 对原来 table 中的非 Unused 态 entry 进行处理。对于 Active 态 entry，显然需要将其插入到新的 table 中，这个动作由前面考察过的 insertdict 完成；而对于 Dummy 态的 entry，则略过，不做任何处理，因为我们知道 Dummy 态 entry 存在的唯一理由就是为了不使搜索时的探测序列中断。现在所有 Active 态的 entry 都重新依次插入新的 table 中，它们会形成一条新的探测序列，不再需要这些 Dummy 态的 entry 了。

现在，利用我们对 PyDictObject 的认识，想象一下从 table 中删除一个元素应该怎样操作呢？

```

[dictobject.c]
int PyDict_DelItem(PyObject *op, PyObject *key)
{
    register dictobject *mp;
    register long hash;
    register dictentry *ep;
    PyObject *old_value, *old_key;

    //获得 hash 值

    if (!PyString_CheckExact(key) ||
        (hash = ((PyStringObject *) key)->ob_shash) == -1) {
        hash = PyObject_Hash(key);
        if (hash == -1)
            return -1;
    }
}

```

```

//搜索 entry

mp = (dictobject *)op;
ep = (mp->ma_lookup)(mp, key, hash);

//删除 entry 所维护的元素

old_key = ep->me_key;
Py_INCREF(dummy);
ep->me_key = dummy;
old_value = ep->me_value;
ep->me_value = NULL;
mp->ma_used--;
Py_DECREF(old_value);
Py_DECREF(old_key);
return 0;
}

```

流程非常清晰，先计算 hash 值，然后搜索相应的 entry，最后删除 entry 中维护的元素，并将 entry 从 Active 态变换为 Dummy 态，同时还将调整 PyDictObject 对象中维护 table 使用情况的变量。

下面我们用一个简单的例子来动态地展示对 PyDictObject 中 table 的维护过程，需要提醒的是，这里采用的散列函数和探测函数都与 Python 中 PyDictObject 实际采用的策略不同，这里只是想要从观念上展示对 table 的维护过程。在下面的图中，蓝色代表 Unused 态 entry，绿色为 Active 态，黄色为 Dummy 态。

假如 table 中有 10 个 entry，散列函数为 $\text{HASH}(x) = x \bmod 10$ ，冲突解决方案采用线性探测，且探测函数为 $x = x + 1$ 。假设向 table 中依次加入了以下元素对：(4, 4)，(14, 14)，(24, 24)，(34, 34)，则加入元素后的 entry 为：

			4	14	24	34			
#1	#2	#3	#4	#5	#6	#7	#8	#9	#10

图 3

现在删除元素对 (14, 14)，然后向 table 中插入新的元素对 (104, 104)。则在搜索的过程中，由于在原来维护 14 的 entry#4 处于现在 Dummy 态，所以 freeslots 会指向这个可用的 entry：

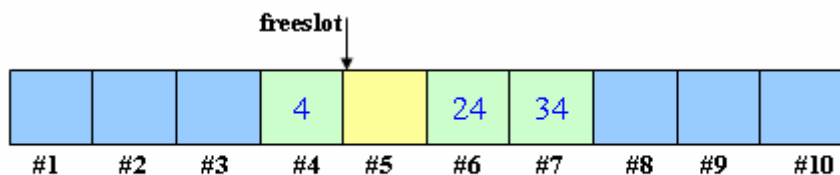


图 4

搜索完成后，填充 freeslot 所指向的 entry:

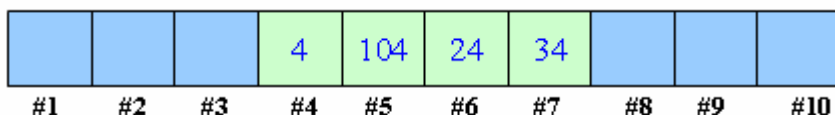


图 5

然后，再向 table 中插入元素对 (14, 14)，这时由于探测序列上已经没有 Dummy 态 entry 了，所以最后返回的 ep 会指向一个处于 Unused 态的 entry:

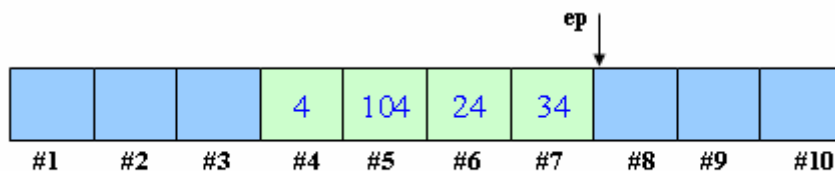


图 6

最后插入元素对 (14, 14)，结果为:

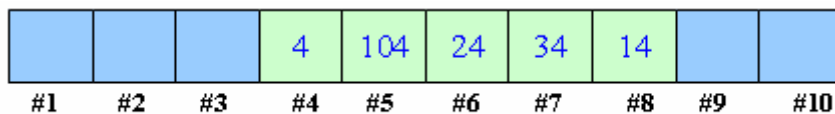


图 7

Python 源码剖析

——字典对象 PyDictObject(3)

本文作者: Robert Chen (search.pythoner@gmail.com)

4 PyDictObject 对象缓冲池

前面我们提到，在 PyDictObject 的实现机制中，同样使用了缓冲池的技术:

```
[dictobject.c]
#define MAXFREEDICTS 80
static PyDictObject *free_dicts[MAXFREEDICTS];
static int num_free_dicts = 0;
```

实际上 PyDictObject 中使用的这个缓冲池机制与 PyListObject 中使用的缓冲池机制是一样的。开始时，这个缓冲池里什么都没有，直到有第一个 PyDictObject 被销毁时，这个缓冲池才开始接纳被缓冲的 PyDictObject 对象:

```
[dictobject.c]
static void dict_dealloc(register dictobject *mp)
```

```

{
    register dictentry *ep;
    int fill = mp->ma_fill;
    PyObject_GC_UnTrack(mp);
    Py_TRASHCAN_SAFE_BEGIN(mp)

    //调整 dict 中对象的引用计数

    for (ep = mp->ma_table; fill > 0; ep++) {
        if (ep->me_key) {
            --fill;
            Py_DECREF(ep->me_key);
            Py_XDECREF(ep->me_value);
        }
    }

    //向系统归还从堆上申请的空间

    if (mp->ma_table != mp->ma_smalltable)
        PyMem_DEL(mp->ma_table);

    //将被销毁的 PyDictObject 对象放入缓冲池

    if (num_free_dicts < MAXFREEDICTS && mp->ob_type ==
        &PyDict_Type)
        free_dicts[num_free_dicts++] = mp;
    else
        mp->ob_type->tp_free((PyObject *)mp);
    Py_TRASHCAN_SAFE_END(mp)
}

```

和 PyListObject 中缓冲池的机制一样，缓冲池中只保留了 PyDictObject 对象，而 PyDictObject 对象中维护的从堆上申请的 table 的空间则被销毁，并归还给系统了。具体原因参见 PyListObject 的讨论。而如果被销毁的 PyDictObject 中的 table 实际上并没有从系统堆中申请，而是指向 PyDictObject 固有的 ma_smalltable，那么只需要调整 ma_smalltable 中的对象引用计数就可以了。

在创建新的 PyDictObject 对象时，如果在缓冲池中有可以使用的对象，则直接从缓冲池中取出使用，而不需要再重新创建：

```

[dictobject.c]
PyObject* PyDict_New(void)
{
    register dictobject *mp;
    .....
    if (num_free_dicts) {
        mp = free_dicts[--num_free_dicts];
        _Py_NewReference((PyObject *)mp);
    }
}

```

```

        if (mp->ma_fill) {
            EMPTY_TO_MINSIZE(mp);
        }
    }
    .....
}

```

5 Hack PyDictObject

现在我们可以根据对 PyDictObject 的了解，在 Python 源代码中添加代码，动态而真实地观察 Python 运行时 PyDictObject 的一举一动了。

我们首先来观察，在 insertdict 发生之后，PyDictObject 对象中 table 的变化情况。由于 Python 内部大量地使用 PyDictObject，所以对 insertdict 的调用会非常频繁，成千上万的 PyDictObject 对象会排着长队来依次使用 insertdict。如果只是简单地输出，我们立刻就会被淹没在输出信息中。所以我们需要一套机制来确保当 insertdict 发生在某一特定的 PyDictObject 对象身上时，才会输出信息。这个 PyDictObject 对象当然是我们自己创建的对象，必须使它有区别于 Python 内部使用的 PyDictObject 对象的特征。这个特征，在这里，我把它定义为 PyDictObject 包含“Python_Robert”的 PyStringObject 对象，当然，你也可以选用自己的特征串。如果在 PyDictObject 中找到了这个对象，则输出信息。

```

static void ShowDictObject(dictobject* dictObject)
{
    dictentry* entry = dictObject->ma_table;
    int count = dictObject->ma_mask+1;
    int i;
    for(i = 0; i < count; ++i)
    {
        PyObject* key = entry->me_key;
        PyObject* value = entry->me_value;
        if(key == NULL)
        {
            printf("NULL");
        }
        else
        {
            (key->ob_type)->tp_print(key, stdout, 0);
        }
    }
}

```

```

printf("\t");

```

```

if(value == NULL)

```

```

        {
            printf("NULL");
        }
        else
        {
            (key->ob_type)->tp_print(value, stdout, 0);
        }
        printf("\n");
        ++entry;
    }
}
static void
insertdict(register dictobject *mp, PyObject *key, long hash,
PyObject *value)
{
    .....
    {
        dictentry *p;
        long strHash;
        PyObject* str = PyString_FromString("Python_Robert");
        strHash = PyObject_Hash(str);
        p = mp->ma_lookup(mp, str, strHash);
        if(p->me_value != NULL && (key->ob_type)->tp_name[0] == 'i')
        {
            PyIntObject* intObject = (PyIntObject*)key;
            printf("insert %d\n", intObject->ob_ival);
        }
    }
}

```

```

        ShowDictObject(mp);
    }
}
}

```

对于 PyDictObject 对象，依次插入 9 和 17，根据 PyDictObject 选用的 hash 策略，这两个数会产生冲突，9 的 hash 结果为 1，而 17 经过再次探测后，会获得 hash 结果为 7。图 7 是观察结果：

```

C:\WINDOWS\system32
G:\PythonSource\Python-2.4.1
Python 2.4.1 (#65, Oct 26 2004)
Type "help", "copyright", "credits()" or "quit()" for more
>>> d = {"Python_Robert":1}
>>> d[9] = 9
insert 9
'Python_Robert' '1'
9 9
NULL NULL
NULL NULL
NULL NULL
NULL NULL
NULL NULL
NULL NULL
>>> d[17] = 17
insert 17
'Python_Robert' '1'
9 9
NULL NULL
NULL NULL
NULL NULL
NULL NULL
NULL NULL
17 17

```

图 8

```

C:\WINDOWS\system32
NULL NULL
17 17
>>> del d[9]
>>> d[17] = 17
insert 17
'Python_Robert' '1'
'<dummy key>' NULL
NULL NULL
NULL NULL
NULL NULL
NULL NULL
17 17
>>> del d[17]
>>> d[17] = 17
insert 17
'Python_Robert' '1'
17 17
NULL NULL
NULL NULL
NULL NULL
NULL NULL
NULL NULL
'<dummy key>' NULL

```

图 9

然后将 9 删除，则原来 9 的位置会出现一个 dummy 态的标识。然后将 17 删除，并再次插入 17，显然，17 应该出现在原来 9 的位置，而原来 17 的位置则是 dummy 标识。图 8 是观察结果。

下面我们观察 Python 内部对 PyDictObject 的使用情况，在 dict_dealloc 中添加代码监控 Python 在执行时调用 dict_dealloc 的频度，图 9 是监测结果。

我们前面已经说了，Python 内部大量使用了 PyDictObject 对象，然而监测的结果还是让我们惊讶不已，原来对于一个简简单单的赋值，一个简简单单的打印，Python 内部都会创建并销毁多达 8 个的 PyDictObject 对象。不过这其中应该有参与编译的 PyDictObject 对象，所以在执行一个完整的 Python 源文件时，并不是每一行都会有这样的八仙过海：当然，我们可以看到，这些 PyDictObject 对象中 entry 的个数都很少，所以只需要使用 ma_smalltable 就可以了。这里，也指出了 PyDictObject 缓冲池的重要性。

```

C:\WINDOWS\system32\cmd
>>> i = 1
dealloc dict with size : 1
dealloc dict with size : 1
dealloc dict with size : 2
dealloc dict with size : 1
dealloc dict with size : 0
dealloc dict with size : 1
dealloc dict with size : 0
dealloc dict with size : 0
>>> print "Hello Python"
dealloc dict with size : 1
dealloc dict with size : 0
dealloc dict with size : 2
dealloc dict with size : 0
dealloc dict with size : 0
dealloc dict with size : 0
dealloc dict with size : 0
dealloc dict with size : 0
Hello Python
>>>

```

图 10

```

C:\WINDOWS\system32\cmd
>>> d1 = {}
>>> print d1
num_free_dicts is : 7
>>> d2 = {}
>>> print d2
num_free_dicts is : 8
>>> d3 = {}
>>> print d3
num_free_dicts is : 8
>>> del d1
>>> print d2
num_free_dicts is : 9
>>> del d2
>>> print d3
num_free_dicts is : 10

```

图 11

所以我们也监控了缓冲池的使用，在 `dict_print` 中添加代码，打印当前的 `num_free_dicts` 值。监控结果见图 10。有一点奇怪的是，在创建了 `d2` 和 `d3` 之后，`num_free_dicts` 的值仍然都是 8。直觉上来讲，它们对应的是应该是 6 和 5 才对。但是，但是，：），看一看左边的图 9，其实在执行 `print` 语句的时候，同样会调用 `dealloc` 8 次，所以每次打印出来，`num_free_dicts` 的值都是 8。在后来 `del d2` 和 `del d1` 时，每次除了 Python 例行的 8 大对象的销毁，还有我们自己创建的对象销毁，所以打印出来的 `num_free_dicts` 的值是 9 和 10。

Python 源码剖析

——Small Python

本文作者: Robert Chen (search.pythoner@gmail.com)

1. Small Python

在详细考察了 Python 中最常用的几个对象之后，我们现在完全可以利用这些对象做出一个最简单的 Python。这一章的目的就是模拟出一个最简单的 Python——Small Python。

在 Small Python 中，我们首先需要实现之前已经分析过的那些对象，比如 `PyIntObject`，与 CPython 不同的是，我们并没有实现像 CPython 那样复杂的机制，

作为一个模拟程序，我们只实现了简单的功能，也没有引入对象缓冲池的机制。这一切都是为了简洁而清晰地展示出 Python 运行时的脉络。

在 Small Python 中，实际上还需要实现 Python 的运行环境，Python 的运行环境是我们在以后的章节中将要剖析的重点。在这里只是展示了其核心的思想——利用 PyDictObject 对象来维护变量名到变量值的映射。

当然，在 CPython 中，还有许多其他的主题，比如 Python 源代码的编译，Python 字节码的生成和执行等等，在 Small Python 中，我们都不会涉及，因为到目前为止，我们没有任何资本做出如此逼真的模拟。不过当我们完成这本书的探索之后，就完全有能力实现一个真正的 Python 了。

在 Small Python 中，我们仅仅实现了 PyIntObject, PyStringObject 以及 PyDictObject 对象，仅仅实现了加法运算和输出操作。同时编译的过程也被简化到了极致，因此我们的 Small Python 只能处理非常受限的表达式。虽然很简陋，但从中可以看到 Python 的骨架，同时，这也是我们深入 Python 解释器和运行时的起点。

2. 对象机制

在 Small Python 中，对象机制与 CPython 完全相同：

```
[PyObject]
#define PyObject_HEAD \
    int refCount;\
    struct tagPyTypeObject *type
```

```
#define PyObject_HEAD_INIT(typePtr) \
    0, typePtr
```

```
typedef struct tagPyObject
{
    PyObject_HEAD;
}PyObject;
```

但是对于类型对象，我们进行了大规模的删减。最终在类型对象中，只定义了加法操作，Hash 操作以及输出操作：

```
[PyTypeObject]
//definition of PyTypeObject
typedef void (*PrintFun)(PyObject* object);
typedef PyObject* (*AddFun)(PyObject* left, PyObject* right);
typedef long (*HashFun)(PyObject* object);
```

```
typedef struct tagPyTypeObject
{
    PyObject_HEAD;
    char* name;
    PrintFun print;
    AddFun add;
    HashFun hash;
}PyTypeObject;
```

PyIntObject 的实现与 CPython 几乎是一样的, 不过没有复杂的对象缓冲机制:

```
[PyIntObject]
typedef struct tagPyIntObject
{
    PyObject_HEAD;
    int value;
}PyIntObject;
```

```
PyObject* PyInt_Create(int value)
{
    PyIntObject* object = new PyIntObject;
    object->refCount = 1;
    object->type = &PyInt_Type;
    object->value = value;
    return (PyObject*)object;
}
```

```
static void int_print(PyObject* object)
{
    PyIntObject* intObject = (PyIntObject*)object;
    printf("%d\n", intObject->value);
}
```

```
static PyObject* int_add(PyObject* left, PyObject* right)
{
    PyIntObject* leftInt = (PyIntObject*)left;
    PyIntObject* rightInt = (PyIntObject*)right;
    PyIntObject* result = (PyIntObject*)PyInt_Create(0);
    if(result == NULL)
    {
        printf("We have no enough memory!!");
    }
}
```



```

    }
    else
    {
        result->value = leftInt->value + rightInt->value;
    }
    return (PyObject*)result;
}

```

```

static long int_hash(PyObject* object)
{
    return (long) ((PyIntObject*)object)->value;
}

```

```

PyTypeObject PyInt_Type =
{
    PyObject_HEAD_INIT(&PyType_Type),
    "int",
    int_print,
    int_add,
    int_hash
};

```

Small Python 中的 PyStringObject 与 CPython 中大不相同，在 CPython 中，它是一个变长对象，而 Small Python 中只是一个简单的定长对象，因为 Small Python 的定位就是个演示的程序：

```

[PyStrObject]
typedef struct tagPyStrObject
{
    PyObject_HEAD;
    int length;
    long hashValue;
    char value[50];
}PyStringObject;

```

```

PyObject* PyStr_Create(const char* value)
{
    PyStringObject* object = new PyStringObject;
    object->refCount = 1;
}

```

```

object->type = &PyString_Type;
object->length = (value == NULL) ? 0 : strlen(value);
object->hashValue = -1;
memset(object->value, 0, 50);
if(value != NULL)
{
    strcpy(object->value, value);
}
return (PyObject*)object;
}

```

```

static void string_print(PyObject* object)
{
    PyStringObject* strObject = (PyStringObject*)object;
    printf("%s\n", strObject->value);
}

```

```

static long string_hash(PyObject* object)
{
    PyStringObject* strObject = (PyStringObject*)object;
    register int len;
    register unsigned char *p;
    register long x;

```

```

    if (strObject->hashValue != -1)
        return strObject->hashValue;
    len = strObject->length;
    p = (unsigned char *)strObject->value;
    x = *p << 7;
    while (--len >= 0)
        x = (1000003*x) ^ *p++;
    x ^= strObject->length;
    if (x == -1)
        x = -2;
    strObject->hashValue = x;
    return x;
}

```

```

static PyObject* string_add(PyObject* left, PyObject* right)
{

```

```

PyStringObject* leftStr = (PyStringObject*)left;
PyStringObject* rightStr = (PyStringObject*)right;
PyStringObject* result = (PyStringObject*)PyStr_Create(NULL);
if(result == NULL)
{
    printf("We have no enough memory!!");
}
else
{
    strcpy(result->value, leftStr->value);
    strcat(result->value, rightStr->value);
}
return (PyObject*)result;
}

```

```

PyTypeObject PyString_Type =
{
    PyObject_HEAD_INIT(&PyType_Type),
    "str",
    string_print,
    string_add,
    string_hash
};

```

在 Python 的解释器工作时，还有一个非常重要的对象，PyDictObject 对象。PyDictObject 对象在 Python 运行时会维护（变量名，变量值）的映射关系，Python 所有的动作都是基于这种映射关系。在 Small Python 中，我们基于 C++ 中的 map 来实现 PyDictObject 对象。当然，map 的运行效率比 CPython 中所采用的 hash 技术会慢上一些，但是对于我们的 Small Python，map 就足够了：

```

[PyDictObject]
typedef struct tagPyDictObject
{
    PyObject_HEAD;
    map<long, PyObject*> dict;
}PyDictObject;

```

```

PyObject* PyDict_Create()
{
    //create object
}

```

```
PyDictObject* object = new PyDictObject;
object->refCount = 1;
object->type = &PyDict_Type;
```

```
return (PyObject*)object;
}
```

```
PyObject* PyDict_GetItem(PyObject* target, PyObject* key)
{
    long keyHashValue = (key->type)->hash(key);
    map<long, PyObject*>& dict = ((PyDictObject*)target)->dict;
    map<long, PyObject*>::iterator it = dict.find(keyHashValue);
    map<long, PyObject*>::iterator end = dict.end();
    if(it == end)
    {
        return NULL;
    }
    return it->second;
}
```

```
int PyDict_SetItem(PyObject* target, PyObject* key, PyObject* value)
{
    long keyHashValue = (key->type)->hash(key);
    PyDictObject* dictObject = (PyDictObject*)target;
    (dictObject->dict)[keyHashValue] = value;
    return 0;
}
```

```
//function for PyDict_Type
static void dict_print(PyObject* object)
{
    PyDictObject* dictObject = (PyDictObject*)object;
    printf("{");
    map<long, PyObject*>::iterator it = (dictObject->dict).begin();
    map<long, PyObject*>::iterator end = (dictObject->dict).end();
    for( ; it != end; ++it)
    {
        //print key
        printf("%d : ", it->first);
        //print value
    }
}
```

```

PyObject* value = it->second;
(value->type)->print(value);
printf(", ");
}
printf("}\n");
}

```

```

PyTypeObject PyDict_Type =
{
    PyObject_HEAD_INIT(&PyType_Type),
    "dict",
    dict_print,
    0,
    0
};

```

Small Python 中的对象机制的所有内容都在上边列出了，非常简单，对吧，这就对了，要的就是这个简单。

3. 解释过程

说 Small Python 中没有编译，对的，它根本就不会进行任何常规的编译动作，没有 token 解析，没有抽象语法树的建立。但说 Small Python 中有那么一点点编译的味道，其实也不错，我们叫这种动作为解释。无论如何，它至少要解析输入的语句，以判断这条语句到底是要干什么，它是要上山打虎呢，还是要下河摸鱼呢，如果连这最基本的都做不到，Small Python 还不如回家卖红薯得了。

然而 Small Python 中的这种解释动作还是被简化到了极致，它实际上就是简单的字符串查找加 if...else...结构：

```

void ExcuteCommand(string& command)
{
    string::size_type pos = 0;
    if((pos = command.find("print ")) != string::npos)
    {
        ExcutePrint(command.substr(6));
    }
    else if((pos = command.find(" = ")) != string::npos)
    {
        string target = command.substr(0, pos);
        string source = command.substr(pos+3);
        ExcuteAdd(target, source);
    }
}

```

```

void ExcuteAdd(string& target, string& source)
{
    string::size_type pos;
    if(IsSourceAllDigit(source))
    {
        PyObject* intValue = PyInt_Create(atoi(source.c_str()));
        PyObject* key = PyStr_Create(target.c_str());
        PyDict_SetItem(m_LocalEnvironment, key, intValue);
    }
    else if(source.find("\\") != string::npos)
    {
        PyObject* strValue = PyStr_Create(source.substr(1,
source.size()-2).c_str());
        PyObject* key = PyStr_Create(target.c_str());
        PyDict_SetItem(m_LocalEnvironment, key, strValue);
    }
    else if((pos = source.find("+")) != string::npos)
    {
        PyObject* leftObject = GetObjectBySymbol(source.substr(0,
pos));
        PyObject* rightObject =
GetObjectBySymbol(source.substr(pos+1));
        if(leftObject != NULL && right != NULL && leftObject->type ==
rightObject->type)
        {
            PyObject* resultValue =
(leftObject->type)->add(leftObject, rightObject);
            PyObject* key = PyStr_Create(target.c_str());
            PyDict_SetItem(m_LocalEnvironment, key, resultValue);
        }
        (m_LocalEnvironment->type)->print(m_LocalEnvironment);
    }
}

```

通过字符串搜索，如果命令中出现“=”，就是一个赋值或加法过程；如果命令中出现“print”，就是一个输出过程。进一步，在 ExcuteAdd 中，又进一步进行字符串搜索，以确定是否需要有一个额外的加法过程。根据这些解析的结果

进行不同的动作，就是 Small Python 中的解释过程。这个过程在 CPython 中是通过正常的编译过程来实现的，而且最后会得到字节码的编译结果。

在这里需要重点指出的是那个 `m_LocalEnvironment`，这是一个 `PyDictObject` 对象，其中维护着 Small Python 运行过程中，动态创建的变量的变量名和变量值的映射。这个就是 Small Python 中的执行环境，Small Python 正是靠它来维护运行过程中的所有变量的状态。在 CPython 中，运行环境实际上也是这样一个机制。当需要访问变量时，就从这个 `PyDictObject` 对象中查找变量的值。这一点在执行输出操作时可以看得很清楚：

```
PyObject* GetObjectBySymbol(string& symbol)
{
    PyObject* key = PyStr_Create(symbol.c_str());
    PyObject* value = PyDict_GetItem(m_LocalEnvironment, key);
    if(value == NULL)
    {
        cout << "[Error] : " << symbol << " is not defined!!" << endl;
        return NULL;
    }
    return value;
}
```

```
void ExcutePrint(string symbol)
{
    PyObject* object = GetObjectFromSymbol(symbol);
    if(object != NULL)
    {
        PyTypeObject* type = object->type;
        type->print(object);
    }
}
```

在这里，通过变量名 `symbol`，获得了变量值 `object`。而在刚才的 `ExcuteAdd` 中，我们将变量名和变量值建立了联系，并存放到 `m_LocalEnvironment` 中。这种一进一出的机制正是 CPython 执行时的关键，在以后对 Python 字节码解释器的详细剖析中，我们将真实而具体地看到这种机制。

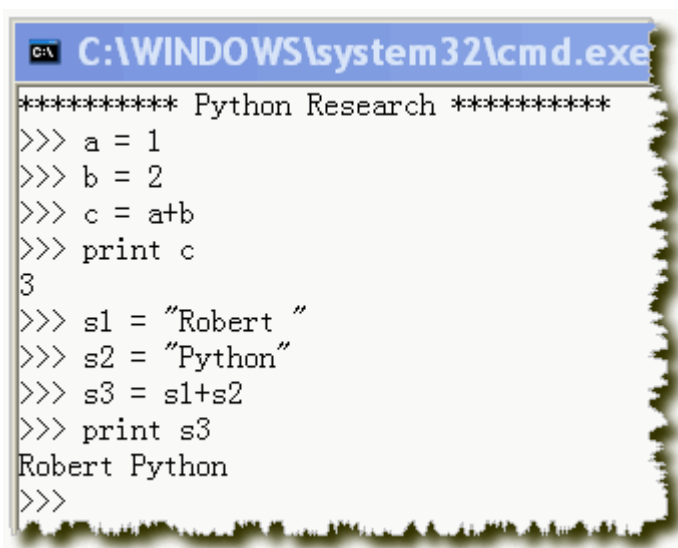
4. 交互式环境

好了，我们的 Small Python 几乎已经完成了，最后所缺的就是一个交互式环境：

```
char* info = "***** Python Research *****\n";
char* prompt = ">>> ";
```

```
void Excute()
{
    cout << info;
    cout << prompt;
    while(getline(cin, m_Command){
        if(m_Command.size() == 0){
            cout << prompt;
            continue;
        }
        else if(m_Command == "exit"){
            return;
        }
        else{
            ExcuteCommand(m_Command);
        }
        cout << prompt;
    }
}
```

有了它，我们的 Small Python 就大功告成了。现在，来看一看我们的成果吧：



```
C:\WINDOWS\system32\cmd.exe
***** Python Research *****
>>> a = 1
>>> b = 2
>>> c = a+b
>>> print c
3
>>> s1 = "Robert "
>>> s2 = "Python"
>>> s3 = s1+s2
>>> print s3
Robert Python
>>>
```

到这里，我们就结束了对 Python 中对象的探索，通过 Small Python 这一个简陋的承前启后的东西，我们将敲开 Python 运行时的大门：那里是字节码，解释器，条件判断语句，函数，类，异常等等的神秘世界，提起精神，我们出发了……

Python 源码剖析

——Pyc 文件解析

本文作者: Robert Chen (search.pythoner@gmail.com)

1. PyCodeObject 与 Pyc 文件

通常认为, Python 是一种解释性的语言, 但是这种说法是不正确的, 实际上, Python 在执行时, 首先会将.py 文件中的源代码编译成 Python 的 byte code (字节码), 然后再由 Python Virtual Machine 来执行这些编译好的 byte code。这种机制的基本思想跟 Java, .NET 是一致的。然而, Python Virtual Machine 与 Java 或.NET 的 Virtual Machine 不同的是, Python 的 Virtual Machine 是一种更高级的 Virtual Machine。这里的高级并不是通常意义上的高级, 不是说 Python 的 Virtual Machine 比 Java 或.NET 的功能更强大, 更拽, 而是说和 Java 或.NET 相比, Python 的 Virtual Machine 距离真实机器的距离更远。或者可以这么说, Python 的 Virtual Machine 是一种抽象层次更高的 Virtual Machine。

我们来考虑下面的 Python 代码:

```
[demo.py]
class A:
    pass
```

```
def Fun():
    pass
```

```
value = 1
str = "Python"
a = A()
Fun()
```

Python 在执行 CodeObject.py 时, 首先需要进行的动作就是对其进行编译, 编译的结果是什么呢? 当然有字节码, 否则 Python 也就没办法在玩下去了。然而除了字节码之外, 还包含其它一些结果, 这些结果也是 Python 运行的时候所必需的。看一下我们的 demo.py, 用我们的眼睛来解析一下, 从这个文件中, 我们可以看到, 其中包含了一些字符串, 一些常量值, 还有一些操作。当然, Python 对操作的处理结果就是自己码。那么 Python 的编译过程对字符串和常量值的处理结果是什么呢? 实际上, 这些在 Python 源代码中包含的静态的信息都会被 Python 收集起来, 编译的结果中包含了字符串, 常量值, 字节码等等在源代码中出现的一切有用的静态信息。而这些信息最终会被存储在 Python 运行期的一个对象中, 当 Python 运行结束后, 这些信息甚至还会被存储在一种文件中。这个对象和文件就是我们这章探索的重点: PyCodeObject 对象和 Pyc 文件。

可以说，PyCodeObject 就是 Python 源代码编译之后的关于程序的静态信息的集合：

```
[compile.h]
/* Bytecode object */
typedef struct {
    PyObject_HEAD
    int co_argcount;      /* #arguments, except *args */
    int co_nlocals;       /* #local variables */
    int co_stacksize;     /* #entries needed for evaluation stack */
    int co_flags;         /* CO_..., see below */
    PyObject *co_code;     /* instruction opcodes */
    PyObject *co_consts;   /* list (constants used) */
    PyObject *co_names;    /* list of strings (names used) */
    PyObject *co_varnames; /* tuple of strings (local variable names) */
    PyObject *co_freevars; /* tuple of strings (free variable names) */
    PyObject *co_cellvars; /* tuple of strings (cell variable names) */
    /* The rest doesn't count for hash/cmp */
    PyObject *co_filename; /* string (where it was loaded from) */
    PyObject *co_name;     /* string (name, for reference) */
    int co_firstlineno;    /* first source line number */
    PyObject *co_lnotab;   /* string (encoding addr<->lineno mapping) */
} PyCodeObject;
```

在对 Python 源代码进行编译的时候，对于一段 Code（Code Block），会创建一个 PyCodeObject 与这段 Code 对应。那么如何确定多少代码算是一个 Code Block 呢，事实上，当进入新的作用域时，就开始了新的一段 Code。也就是说，对于下面的这一段 Python 源代码：

```
[CodeObject.py]
class A:
    pass
```

```
def Fun():
    pass
```

```
a = A()
Fun()
```

在 Python 编译完成后，一共会创建 3 个 PyCodeObject 对象，一个是对应 CodeObject.py 的，一个是对应 class A 这段 Code（作用域），而最后一个是对应 def Fun 这段 Code 的。每一个 PyCodeObject 对象中都包含了每一个代码块经过编译后得到的 byte code。但是不幸的是，Python 在执行完这些 byte code 后，会销毁 PyCodeObject，所以下次再次执行这个.py 文件时，Python 需要重新编译源代码，创建三个 PyCodeObject，然后执行 byte code。

很不爽，对不对？Python 应该提供一种机制，保存编译的中间结果，即 byte code，或者更准确地说，保存 PyCodeObject。事实上，Python 确实提供了这样一种机制——Pyc 文件。

Python 中的 pyc 文件正是保存 PyCodeObject 的关键所在，我们对 Python 解释器的分析就从 pyc 文件，从 pyc 文件的格式开始。

在分析 pyc 的文件格式之前，我们先来看看如何产生 pyc 文件。在执行一个.py 文件中的源代码之后，Python 并不会自动生成与该.py 文件对应的.pyc 文件。我们需要自己触发 Python 来创建 pyc 文件。下面我们提供一种使 Python 创建 pyc 文件的方法，其实很简单，就是利用 Python 的 import 机制。

在 Python 运行的过程中，如果碰到 import abc，这样的语句，那么 Python 将到设定好的 path 中寻找 abc.pyc 或 abc.dll 文件，如果没有这些文件，而只是发现了 abc.py，那么 Python 会首先将 abc.py 编译成相应的 PyCodeObject 的中间结果，然后创建 abc.pyc 文件，并将中间结果写入该文件。接下来，Python 才会对 abc.pyc 文件进行一个 import 的动作，实际上也就是将 abc.pyc 文件中的 PyCodeObject 重新在内存中复制出来。了解了这个过程，我们很容易利用下面所示的 generator.py 来创建上面那段代码(CodeObjectt.py)对应的 pyc 文件了。

generator.py	CodeObject.py
<pre>import test print "Done"</pre>	<pre>class A: pass def Fun(): pass a = A() Fun()</pre>

图 1 所示的是 Python 产生的 pyc 文件：

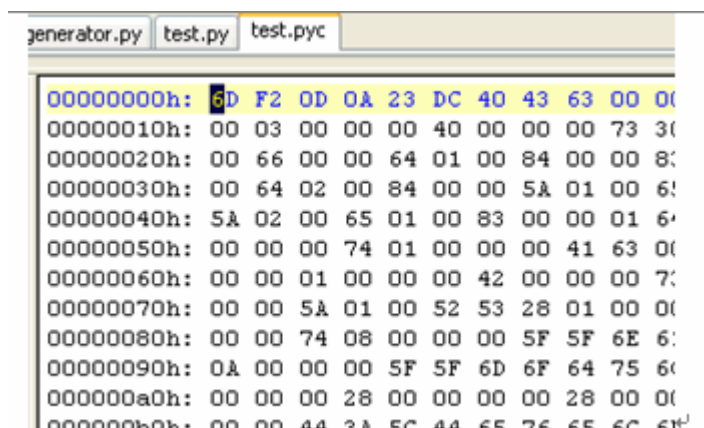


图 1

可以看到，pyc 是一个二进制文件，那么 Python 如何解释这一堆看上去毫无意义的字节流就至关重要了。这也就是 pyc 文件的格式。

要了解 pyc 文件的格式，首先我们必须清楚 PyCodeObject 中每一个域都表示什么含义，这一点是无论如何不能绕过去的。

Field	Content
co_argcount	Code Block 的参数个数，比如说一个函数的参数
co_nlocals	Code Block 中局部变量的个数
co_stacksize	执行该段 Code Block 需要的栈空间
co_flags	N/A
co_code	Code Block 编译所得的 byte code。以 PyStringObject 的形式存在
co_consts	PyTupleObject 对象，保存该 Block 中的常量
co_names	PyTupleObject 对象，保存该 Block 中的所有符号
co_varnames	N/A
co_freevars	N/A
co_cellvars	N/A
co_filename	Code Block 所对应的.py 文件的完整路径
co_name	Code Block 的名字，通常是函数名或类名
co_firstlineno	Code Block 在对应的.py 文件中的起始行
co_lnotab	byte code 与.py 文件中 source code 行号的对应关系，以 PyStringObject 的形式存在

需要说明一下的是 co_lnotab 域。在 Python2.3 以前，有一个 byte code，唤做 SET_LINEENO，这个 byte code 会记录.py 文件中 source code 的位置信息，这个信息对于调试和显示异常信息都有用。但是，从 Python2.3 之后，Python 在编译时不会再产生这个 byte code，相应的，Python 在编译时，将这个信息记录到了 co_lnotab 中。

co_lnotab 中的 byte code 和 source code 的对应信息是以 unsigned bytes 的数组形式存在的，数组的形式可以看作（byte code 在 co_code 中位置增量，代码行数增量）形式的一个 list。比如对于下面的例子：

Byte code 在 co_code 中的偏移	.py 文件中源代码的行数
0	1
6	2
50	7

这里有一个小小的技巧，Python 不会直接记录这些信息，相反，它会记录这些信息间的增量值，所以，对应的 `co_inotab` 就应该是：0, 1, 6, 1, 44, 5。

2. Pyc 文件的生成

前面我们提到，Python 在 `import` 时，如果没有找到相应的 `pyc` 文件或 `dll` 文件，就会在 `py` 文件的基础上自动创建 `pyc` 文件。那么，要想了解 `pyc` 的格式到底是什么样的，我们只需要考察 Python 在将编译得到的 `PyCodeObject` 写入到 `pyc` 文件中时到底进行了怎样的动作就可以了。下面的函数就是我们的切入点：

```
[import.c]
static void write_compiled_module(PyCodeObject *co, char *cpathname,
long mtime)
{
    FILE *fp;
    fp = open_exclusive(cpathname);
    PyMarshal_WriteLongToFile(pyc_magic, fp, Py_MARSHAL_VERSION);

    /* First write a 0 for mtime */
    PyMarshal_WriteLongToFile(0L, fp, Py_MARSHAL_VERSION);
    PyMarshal_WriteObjectToFile((PyObject *)co, fp,
Py_MARSHAL_VERSION);

    /* Now write the true mtime */
    fseek(fp, 4L, 0);
    PyMarshal_WriteLongToFile(mtime, fp, Py_MARSHAL_VERSION);
    fflush(fp);
    fclose(fp);
}
```

这里的 `cpathname` 当然是 `pyc` 文件的绝对路径。首先我们看到会将 `pyc_magic` 这个值写入到文件的开头。实际上，`pyc_magic` 对应一个 `MAGIC` 的值。`MAGIC` 是用来保证 Python 兼容性的一个措施。比如说要防止 Python2.4 的运行环境加载由 Python1.5 产生的 `pyc` 文件，那么只需要将 Python2.4 和 Python1.5 的 `MAGIC` 设为不同的值就可以了。Python 在加载 `pyc` 文件时会首先检查这个 `MAGIC` 值，从而拒绝加载不兼容的 `pyc` 文件。那么 `pyc` 文件为什么会不兼容了，一个最主要的原因是 `byte code` 的变化，由于 Python 一直在不断地改进，有一些 `byte code` 退出了历史舞台，比如上面提到的 `SET_LINEENO`；或者由于一些新的语法特性会加入新的 `byte code`，这些都会导致 Python 的不兼容问题。

`pyc` 文件的写入动作最后会集中到下面所示的几个函数中（这里假设代码只处理写入到文件，即 `p->fp` 是有效的。因此代码有删减，另有一个 `w_short` 未列出。缺失部分，请参考 Python 源代码）：

```
[marshal.c]
```

```

typedef struct {
    FILE *fp;
    int error;
    int depth;
    PyObject *strings; /* dict on marshal, list on unmarshal */
} WFILE;

#define w_byte(c, p) putc((c), (p)->fp)

static void w_long(long x, WFILE *p)
{
    w_byte((char)(x & 0xff), p);
    w_byte((char)((x>> 8) & 0xff), p);
    w_byte((char)((x>>16) & 0xff), p);
    w_byte((char)((x>>24) & 0xff), p);
}

static void w_string(char *s, int n, WFILE *p)
{
    fwrite(s, 1, n, p->fp);
}

```

在调用 PyMarshal_WriteLongToFile 时，会直接调用 w_long，但是在调用 PyMarshal_WriteObjectToFile 时，还会通过一个间接的函数：w_object。需要特别注意的是 PyMarshal_WriteObjectToFile 的第一个参数，这个参数正是 Python 编译出来的 PyCodeObject 对象。

w_object 的代码非常长，这里就不全部列出。其实 w_object 的逻辑非常简单，就是对应不同的对象，比如 string, int, list 等，会有不同的写的动作，然而其最终目的都是通过最基本的 w_long 或 w_string 将整个 PyCodeObject 写入到 pyc 文件中。

对于 PyCodeObject，很显然，会遍历 PyCodeObject 中的所有域，将这些域依次写入：

```

[marshal.c]
static void w_object(PyObject *v, WFILE *p)
{
    .....
    else if (PyCode_Check(v))
    {
        PyCodeObject *co = (PyCodeObject *)v;
        w_byte(TYPE_CODE, p);

```

```

w_long(co->co_argcount, p);
w_long(co->co_nlocals, p);
w_long(co->co_stacksize, p);
w_long(co->co_flags, p);
w_object(co->co_code, p);
w_object(co->co_consts, p);
w_object(co->co_names, p);
w_object(co->co_varnames, p);
w_object(co->co_freevars, p);
w_object(co->co_cellvars, p);
w_object(co->co_filename, p);
w_object(co->co_name, p);
w_long(co->co_firstlineno, p);
w_object(co->co_lnotab, p);
}
.....
}

```

而对于一个 `PyListObject` 对象,想象一下会有什么动作? 没错,还是遍历!!!:

```

[w_object() in marshal.c]
.....
else if (PyList_Check(v))
{
    w_byte(TYPE_LIST, p);
    n = PyList_GET_SIZE(v);
    w_long((long)n, p);
    for (i = 0; i < n; i++)
    {
        w_object(PyList_GET_ITEM(v, i), p);
    }
}
.....

```

而如果是 `PyIntObject`, 嗯, 那太简单了, 几乎没有什么可说的:

```

[w_object() in marshal.c]
.....
else if (PyInt_Check(v))
{
    w_byte(TYPE_INT, p);
    w_long(x, p);
}
.....

```

有没有注意到 `TYPE_LIST`, `TYPE_CODE`, `TYPE_INT` 这样的标志? `pyc` 文件正是利用这些标志来表示一个新的对象的开始, 当加载 `pyc` 文件时, 加载器才能知道在什么时候应该进行什么样的加载动作。这些标志同样也是在 `import.c` 中定义的:

```
[import.c]
#define TYPE_NULL    '0'
#define TYPE_NONE    'N'
. . . . .
#define TYPE_INT     'i'
#define TYPE_STRING  's'
#define TYPE_INTERNEDED  't'
#define TYPE_STRINGREF 'R'
#define TYPE_TUPLE   '('
#define TYPE_LIST    '['
#define TYPE_CODE     'c'
```

到了这里, 可以看到, Python 对于中间结果的导出实际是不复杂的。实际上在 `write` 的动作中, 不论面临 `PyCodeObject` 还是 `PyListObject` 这些复杂对象, 最后都会归结为简单的两种形式, 一个是对数值的写入, 一个是对字符串的写入。上面其实我们已经看到了对数值的写入过程。在写入字符串时, 有一套比较复杂的机制。在了解字符串的写入机制前, 我们首先需要了解一个写入过程中关键的结构体 `WFILE` (有删节):

```
[marshal.c]
typedef struct {
    FILE *fp;
    int error;
    int depth;
    PyObject *strings; /* dict on marshal, list on unmarshal */
} WFILE;
```

这里我们也只考虑 `fp` 有效, 即写入到文件, 的情况。`WFILE` 可以看作是一个对 `FILE*` 的简单包装, 但是在 `WFILE` 里, 出现了一个奇特的 `strings` 域。这个域是在 `pyc` 文件中写入或读出字符串的关键所在, 当向 `pyc` 中写入时, `string` 会是一个 `PyDictObject` 对象; 而从 `pyc` 中读出时, `string` 则会是一个 `PyListObject` 对象。

```
[marshal.c]
void PyMarshal_WriteObjectToFile(PyObject *x, FILE *fp, int version)
{
    WFILE wf;
    wf.fp = fp;
```



```

wf.error = 0;
wf.depth = 0;
wf.strings = (version > 0) ? PyDict_New() : NULL;
w_object(x, &wf);
}

```

可以看到，`strings` 在真正开始写入之前，就已经被创建了。在 `w_object` 中对于字符串的处理部分，我们可以看到对 `strings` 的使用：

```

[w_object() in marshal.c]
.....
else if (PyString_Check(v))
{
    if (p->strings && PyString_CHECK_INTERNED(v))
    {
        PyObject *o = PyDict_GetItem(p->strings, v);
        if (o)
        {
            long w = PyInt_AsLong(o);
            w_byte(TYPE_STRINGREF, p);
            w_long(w, p);
            goto exit;
        }
        else
        {
            o = PyInt_FromLong(PyDict_Size(p->strings));
            PyDict_SetItem(p->strings, v, o);
            Py_DECREF(o);
            w_byte(TYPE_INTERNED, p);
        }
    }
    else
    {
        w_byte(TYPE_STRING, p);
    }
    n = PyString_GET_SIZE(v);
    w_long((long)n, p);
    w_string(PyString_AS_STRING(v), n, p);
}
.....

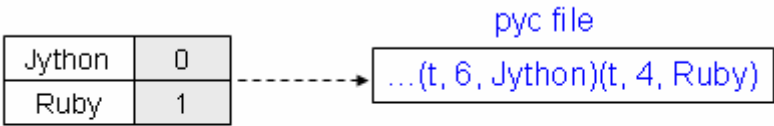
```

真正有趣的事发生在这个字符串是一个需要被进行 `INTERN` 操作的字符串时。可以看到，`WFILE` 的 `strings` 域实际上是一个从 `string` 映射到 `int` 的一个

PyDictObject 对象。这个 int 值是什么呢，这个 int 值是表示对应的 string 是第几个被加入到 WFILE.strings 中的字符串。

这个 int 值看上去似乎没有必要，记录一个 string 被加入到 WFILE.strings 中的序号有什么意义呢？好，让我们来考虑下面的情形：

假设我们需要向 pyc 文件中写入三个 string: "Jython", "Ruby", "Jython"，而且这三个 string 都需要被进行 INTERN 操作。对于前两个 string，没有任何问题，闭着眼睛写入就是了。完成了前两个 string 的写入后，WFILE.strings 与 pyc 文件的情况如图 2 所示：



注意，pyc 文件中的括号和逗号是为了方便理解所添加。

图 2

在写入第三个字符串的时候，麻烦来了。对于这个“Jython”，我们应该怎么处理呢？

是按照上两个 string 一样吗？如果这样的话，那么写入后，WFILE.strings 和 pyc 的情况如图 3 所示：

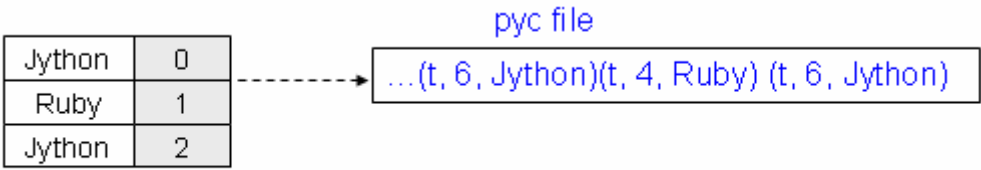


图 3

我们可以不管 WFILE.strings 怎么样了，但是一看 pyc 文件，我们就知道，问题来了。在 pyc 文件中，出现了重复的内容，关于“Jython”的信息重复了两次，这会引起什么麻烦呢？想象一下在 python 代码中，我们创建了一个 button，在此之后，多次使用了 button，这样，在代码中，“button”将出现多次。想象一下吧，我们的 pyc 文件会变得多么臃肿，而其中充斥的只是毫无价值的冗余信息。如果你是 Guido，你能忍受这样的设计吗？当然不能！！于是 Guido 给了我们 TYPE_STRINGREF 这个东西。在解析 pyc 文件时，这个标志表明后面的一个数值表示了一个索引值，根据这个索引值到 WFILE.strings 中去查找，就能找到需要的 string 了。

有了 TYPE_STRINGREF，我们的 pyc 文件就能变得苗条了，如图 4 所示：

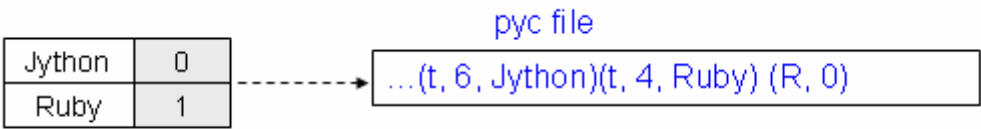
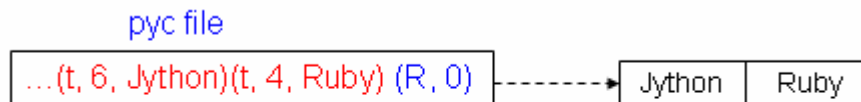


图 4

看一下加载 pyc 文件的过程，我们就能对这个机制更加地明了了。前面我们提到，在读入 pyc 文件时，WFILE.strings 是一个 PyListObject 对象，所以在读入前两个字符串后，WFILE.strings 的情形如图 5 所示：



注：红色部分表示已经加载的部分

图 5

在加载紧接着的 (R, 0) 时，因为解析到是一个 TYPE_STRINGREF 标志，所以直接以标志后面的数值 0 位索引访问 WFILE.strings，立刻可得到字符串“Jython”。

3. 一个 PyCodeObject, 多个 PyCodeObject?

到了这里，关于 PyCodeObject 与 pyc 文件，我们只剩下最后一个有趣的话题了。还记得前面那个 test.py 吗？我们说那段简单的什么都做不了的 python 代码就要产生三个 PyCodeObject。而在 write_compiled_module 中我们又亲眼看到，Python 运行环境只会对一个 PyCodeObject 对象调用 PyMarshal_WriteObjectToFile 操作。刹那间，我们竟然看到了两个遗失的 PyCodeObject 对象。

Python 显然不会犯这样低级的错误，想象一下，如果你是 Guido，这个问题该如何解决？首先我们会假想，有两个 PyCodeObject 对象一定是包含在另一个 PyCodeObject 中的。没错，确实如此，还记得我们最开始指出的 Python 是如何确定一个 Code Block 的吗？对喽，就是作用域。仔细看一下 test.py，你会发现作用域呈现出一种嵌套的结构，这种结构也正是 PyCodeObject 对象之间的结构。所以到现在清楚了，与 Fun 和 A 对应得 PyCodeObject 对象一定是包含在与全局作用域对应的 PyCodeObject 对象中的，而 PyCodeObject 结构中的 co_consts 域正是这两个 PyCodeObject 对象的藏身之处，如图 6 所示：

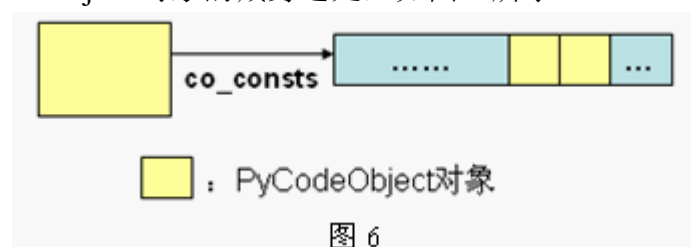


图 6

在对一个 PyCodeObject 对象进行写入到 pyc 文件的操作时，如果碰到它包含的另一个 PyCodeObject 对象，那么就会递归地执行写入 PyCodeObject 对象的操作。如此下去，最终所有的 PyCodeObject 对象都会被写入到 pyc 文件中去。而且 pyc 文件中的 PyCodeObject 对象也是以一种嵌套的关系联系在一起的。

4. Python 字节码

Python 源代码在执行前会被编译为 Python 的 byte code，Python 的执行引擎就是根据这些 byte code 来进行一系列的操作，从而完成对 Python 程序的执行。在 Python2.4.1 中，一共定义了 103 条 byte code：

```
[opcode.h]
#define STOP_CODE    0
#define POP_TOP      1
#define ROT_TWO      2
.....
#define CALL_FUNCTION_KW      141
#define CALL_FUNCTION_VAR_KW  142
#define EXTENDED_ARG  143
```

所有这些字节码的操作含义在 Python 自带的文档中有专门的一页进行描述，当然，也可以到下面的网址察看：<http://docs.python.org/lib/bytecodes.html>。

细心的你一定发现了，byte code 的编码却到了 143。没错，Python2.4.1 中 byte code 的编码并没有按顺序增长，比如编码为 5 的 ROT_FOUR 之后就是编码为 9 的 NOP。这可能是历史遗留下来的，你知道，在咱们这行，历史问题不是什么好东西，搞得现在还有许多人不得不很郁闷地面对 MFC：)

Python 的 143 条 byte code 中，有一部分是需要参数的，另一部分是没有参数的。所有需要参数的 byte code 的编码都大于或等于 90。Python 中提供了专门的宏来判断一条 byte code 是否需要参数：

```
[opcode.h]
#define HAS_ARG(op) ((op) >= HAVE_ARGUMENT)
```

好了，到了现在，关于 PyCodeObject 和 pyc 文件的一切我们都已了如指掌了，关于 Python 的现在我们可以做一些非常有趣的事了。呃，在我看来，最有趣的事莫过于自己写一个 pyc 文件的解析器。没错，利用我们现在所知道的一切，我们真的可以这么做了。图 7 展现的是对本章前面的那个 test.py 的解析结果：

```

- <PycFile>
- <codeObject>
  <argCount value="0" />
  <localCount value="0" />
  <stackSize value="3" />
  <flags value="64" />
+ <code>
- <consts>
  <internStr index="0" length="1" value="A" />
+ <codeObject>
+ <codeObject>
  <NoneObject />
</consts>
- <names>
  <strRef index="0" value="A" />
  <strRef index="4" value="Fun" />
  <internStr index="5" length="1" value="a" />
</names>
+ <varNames>
  <freeVars />
  <cellVars />
- <fileName>
  <strRef index="3" value="D:\Develop\Spy\PycGenerator\test.py" />
</fileName>
- <name>
  <internStr index="6" length="1" value="?" />
</name>
  <firstLineNo />
+ <lnotab>
</codeObject>
</PycFile>

```

图 7

更进一步，我们还可以解析 byte code。前面我们已经知道，Python 在生成 pyc 文件时，会将 PyCodeObject 对象中的 byte code 也写入到 pyc 文件中，而且这个 pyc 文件中还记录了每一条 byte code 与 Python 源代码的对应关系，嗯，就是那个 co_lnotab 啦。假如现在我们知道了 byte code 在 co_code 中的偏移地址，那么与这条 byte code 对应的 Python 源代码的位置可以通过下面的算法得到（Python 伪代码）：

```

lineno = addr = 0
for addr_incr, line_incr in c_lnotab:
    addr += addr_incr
    if addr > A:
        return lineno
    lineno += line_incr

```

下面是对一段 Python 源代码反编译为 byte code 的结果，这个结果也将作为下一章对 Python 执行引擎的分析的开始：

```
i = 1
#  LOAD_CONST    0
#  STORE_NAME    0
```

```
s = "Python"
#  LOAD_CONST    1
#  STORE_NAME    1
```

```
d = {}
#  BUILD_MAP     0
#  STORE_NAME    2
```

```
l = []
#  BUILD_LIST    0
#  STORE_NAME    3
#  LOAD_CONST    2
#  RETURN_VALUE  none
```

再往前想一想，从现在到达的地方出发，实际上我们就可以做出一个 Python 的执行引擎了，哇，这是多么激动人心的事啊。遥远的天空，一抹朝阳，缓缓升起了……

事实上，Python 标准库中提供了对 python 进行反编译的工具 `dis`，利用这个工具，可以很容易地得到我们在这里得到的结果，当然，还要更详细一些，图 8 展示了利用 `dis` 工具对 `CodeObject.py` 进行反编译的结果：

```
IDLE 1.1
>>> import dis
>>> f = open('CodeObject.py').read()
>>> co = compile(f, 'CodeObject.py', 'exec')
>>> dis.dis(co)
1          0 LOAD_CONST           0 (1)
          3 STORE_NAME           0 (i)

2          6 LOAD_CONST           1 ('Python')
          9 STORE_NAME           1 (s)

3         12 BUILD_MAP           0
         15 STORE_NAME           2 (d)

4         18 BUILD_LIST          0
         21 STORE_NAME           3 (l)
         24 LOAD_CONST           2 (None)
         27 RETURN_VALUE
```

图 8

在图 8 显示的结果中，最左面一列显示的是 CodeObject.py 中源代码的行数，左起第二列显示的是当前的字节码指令在 co_code 中的偏移位置。

在以后的分析中，我们大部分将采用 dis 工具的反编译结果，在有些特殊情况下会使用我们自己的反编译结果。

Python 源码剖析

——Python 执行引擎之框架

本文作者: Robert Chen(search.pythoner@gmail.com)

1 PyFrameObject

Python 执行引擎是 Python 的核心，Python 源代码在被编译为 byte code 之后，就将由 Python 的执行引擎接手整个工作，依次读入每一条 byte code，在当前的上下文环境中执行这条 byte code。如此反复推磨，所有由 Python 源代码所规定的动作都会如期望一样，一一展开。

在进入 Python 的执行引擎之前，我们先来看一看在 x86 的机器上，程序是以一种什么方式运行的，在这里，我们主要关注运行时栈的栈帧，如图 1 所示：

帧的栈指针 `esp` 和帧指针 `ebp`。大致上，这就是可执行文件在 x86 机器上的运行原理，而 Python 正是在执行引擎中通过不同的实现方式模拟了这一原理。

前面我们已经知道，Python 源代码经过编译之后，所有的 `byte code` 以及程序的其他信息都存放在 `PyCodeObject` 对象中，那么 Python 的执行引擎是否就是在这个 `PyCodeObject` 对象上进行所有的动作呢？呃，是，又不是。`PyCodeObject` 中包含了最关键的 `byte code` 以及关于程序的所有信息。然而有一点，`PyCodeObject` 没有包含，也不可能包含。这就是执行环境。

什么是执行环境呢，考虑下面的一个例子：

```
i = 'Python'
```

```
def f():  
    i = 999  
    print i #1
```

```
f()  
print i #2
```

在 1 和 2 两个地方，都进行了同样的动作，即 `print i`，显然，它们所对应的字节码是相同的，但是这两条语句的执行效果是不同的。这样的结果正是在执行环境的影响下产生的。在执行 1 处的 `print` 时，执行环境中，`i` 的值为 999；而在执行 2 处的 `print` 时，执行环境中 `i` 的值为“Python”。这种同样的名字对应不同的值，甚至不同的类型的情况，必须在运行时动态地被捕捉和维护。这些则不可能在 `PyCodeObject` 中被静态地存储。

联想到 x86 运行程序的机理，我们可以这样来考虑。当 Python 调用函数 `f` 时，会在当前的运行环境之外重新创建一个新的运行环境，在这个新的运行环境中，有一个新的名字为“`i`”的对象，这个新的运行环境实际上可以看成是一个新的栈帧。所以在 Python 真正执行的时候，它的执行引擎实际上面对的并不是一个 `PyCodeObject` 对象，而是另一个家伙——`PyFrameObject`，这个东西就是 Python 对栈帧的模拟，你看，它的名字中还有个 `Frame` 呢。

当然，对于 Python 而言，`PyFrameObject` 对象不是一个我们在 x86 机器上看到的那个简简单单的栈帧，它实际上包含了其他更多的信息，请看：

```
[frameobject.h]  
typedef struct _frame {  
    PyObject_VAR_HEAD  
    struct _frame *f_back; /* previous frame, or NULL */  
    PyCodeObject *f_code; /* code segment */  
    PyObject *f_builtins; /* builtin symbol table (PyDictObject) */  
    PyObject *f_globals; /* global symbol table (PyDictObject) */  
    PyObject *f_locals; /* local symbol table (any mapping) */  
    PyObject **f_valstack; /* points after the last local */  
    PyObject **f_stacktop;
```

```

.....
int f_lasti;          /* Last instruction if called */
/* As of 2.3 f_lineno is only valid when tracing is active (i.e. when
   f_trace is set) -- at other times use PyCode_Addr2Line instead. */
int f_lineno;         /* Current line number */
int f_nlocals;        /* number of locals */
int f_ncells;
int f_nfreevars;
int f_stacksize;      /* size of value stack */
PyObject *f_localsplus[1]; /* locals+stack, dynamically sized */
} PyFrameObject;

```

从 `f_back` 我们可以窥见一点，在 Python 实际的执行中，会产生很多 `PyFrameObject` 对象，而这些对象会被链接起来，形成一个链表。是不是看出点眉目来了，这似乎正是对 x86 机器上栈帧间关系的模拟，在 x86 上，栈帧间通过 `esp` 指针和 `ebp` 指针建立了关系，使新的栈帧在结束之后能顺利回退到旧的栈帧中，而 Python 似乎正是利用一个指针来完成这个动作。那真实的情况是不是这样呢，这是后话，暂且按下不表 ☺

在 `f_code` 中存放的是一个待执行的 `PyCodeObject` 对象，而接下来的 `f_builtins`, `f_globals`, `f_locals` 正是动态的执行环境，这三个 `PyObject*` 都会指向 `PyDictObject` 对象，而在这些 `PyDictObject` 对象中，分别维护了 `builtin` 的 `name`, `global` 的 `name`, `local` 的 `name` 与对应的值之间的映射关系。想想前面的那段 Python 代码，在执行 `print i` 时，会首先到 `f_locals` 中去寻找 `PyStringObject` 对象 `'i'`，找到了之后，将其对应的值取出，并打印出来。

在 `PyFrameObject` 的开头，有一个 `PyObject_VAR_HEAD`，这表明 `PyFrameObject` 是一个变长的对象，即每次创建的 `PyFrameObject` 对象的大小可能是不一样的。这些变动的内存是用来做什么的呢？实际上，每一个 `PyFrameObject` 对象都维护了一个 `PyCodeObject` 对象，这表明一个 `PyFrameObject` 对象和 Python 源代码中的一段 `Code` 是对应的，更准确地说，是我们在研究 `PyCodeObject` 时提到的那个 `Code Block` 对应的。而在编译一段 `Code Block` 时，会计算出这段 `Code Block` 执行过程中所需要的栈空间的大小（注意，这个栈空间才是和 x86 机器上那个用于函数执行的栈空间相对应的概念），这个栈空间的大小存储在 `f_stacksize` 中，而这个栈正是那段变动的内存。因为不同的 `Code Block` 所需的栈空间的大小是不同的，所以这决定了 `PyFrameObject` 一定有一个 `PyObject_VAR_HEAD`。在 `PyFrameObject` 对象所维护的栈中，存储的都是 `PyObject*`，你可能看出来了，这个栈的起始位置是从 `f_localsplus` 开始的。呃，其实不完全正确，`f_localsplus` 确实维护了一段变动长度的内存，但是这段内存不光是给栈使用的，而且还有别的家伙也会使用：

```

[frameobject.c] (有删节)
PyFrameObject *
PyFrame_New(PyThreadState *tstate, PyCodeObject *code, PyObject
*globals,
            PyObject *locals)

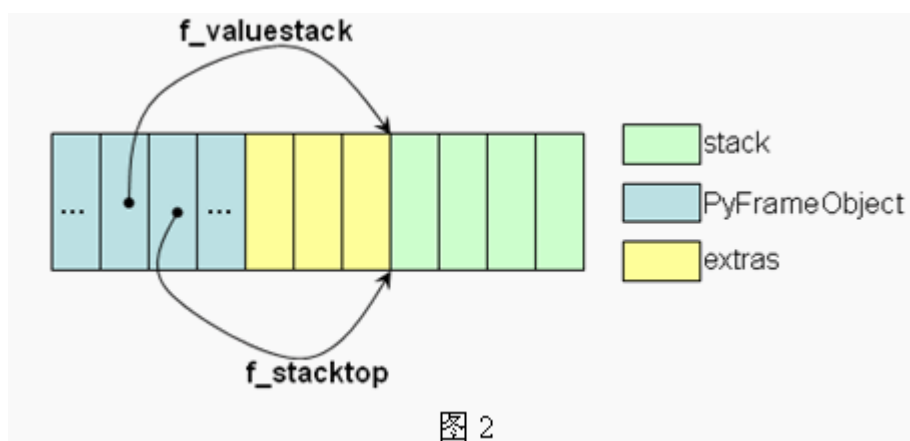
```

```

{
    PyFrameObject *f;
    int extras, ncells, nfreess, i;
    ncells = PyTuple_GET_SIZE(code->co_cellvars);
    nfreess = PyTuple_GET_SIZE(code->co_freevars);
    extras = code->co_stacksize + code->co_nlocals + ncells + nfreess;
    f = PyObject_GC_NewVar(PyFrameObject, &PyFrame_Type, extras);
    f->f_nlocals = code->co_nlocals;
    f->f_stacksize = code->co_stacksize;
    f->f_ncells = ncells;
    f->f_nfreevars = nfreess;
    extras = f->f_nlocals + ncells + nfreess;
    f->f_valuestack = f->f_localsplus + extras;
    f->f_stacktop = f->f_valuestack;
    return f;
}

```

可见，在创建 `PyFrameObject` 对象时，额外申请的那部分内存有一部分是给 `PyCodeObject` 对象中存储的那些 `co_names` 啊，`co_freevars` 啊，`co_cellvars` 啊使用的，而另一部分才是给栈使用的。所以，`PyFrameObject` 对象中的栈的起始位置是由 `f_valuestack` 维护的，而 `f_stacktop` 维护了当前的栈顶。图 2 是一个刚被创建的 `PyFrameObject` 对象的示意图：



2 Python 执行引擎框架

当 Python 启动后，会首先进行 Python 运行时环境的初始化，这个过程非常地复杂，我们将在后面用单独的一章来剖析，这里我们架设初始化的动作已经完成，我们已经站在了执行引擎的门槛外，只需要轻轻推动一下第一张骨牌，整个执行过程就像多米诺骨牌一样，一环扣一环地展开。

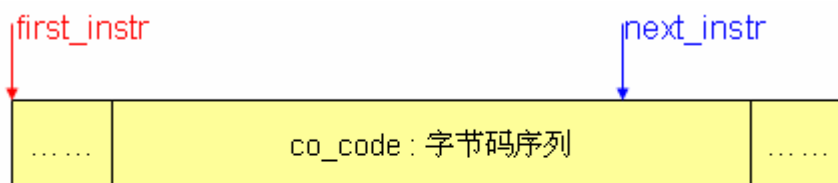
这个推动第一张骨牌的地方在一个名叫 `PyEval_EvalFrame` 的函数中，`PyEval_EvalFrame` 首先会初始化一些变量，其中 `PyFrameObject` 对象中的 `PyCodeObject` 对象包含的重要信息都被照顾到了。当然，另一个重要的动作就是初始化了堆栈的栈顶指针，使其指向 `f->f_stacktop`：

```
[PyEval_EvalFrame in ceval.c]
    co = f->f_code;
    names = co->co_names;
    consts = co->co_consts;
    fastlocals = f->f_localsplus;
    freevars = f->f_localsplus + f->f_nlocals;
    first_instr = PyString_AS_STRING(co->co_code);
    /* An explanation is in order for the next line.
```

```

    f->f_lasti now refers to the index of the last instruction
    executed. You might think this was obvious from the name, but
    this wasn't always true before 2.3! PyFrame_New now sets
    f->f_lasti to -1 (i.e. the index *before* the first instruction)
    and YIELD_VALUE doesn't fiddle with f_lasti any more. So this
    does work. Promise. */
    next_instr = first_instr + f->f_lasti + 1;
    stack_pointer = f->f_stacktop;
    assert(stack_pointer != NULL);
    f->f_stacktop = NULL; /* remains NULL unless yield suspends frame
    */
```

前面我们说过，在 `PyCodeObject` 对象的 `co_code` 域中保存着字节码和字节码的参数，Python 执行引擎执行字节码的过程就是从头到尾遍历整个 `co_code` 域，依次执行字节码的过程。在 Python 的执行引擎中，利用三个变量来完成整个遍历过程，图 3 展示了这三个变量在遍历中某时刻的情形：



`f->f_lasti` : 上一条已执行的字节码在 `co_code` 中的索引

图 3

那么这个一步一步的动作是如何完成的呢，我们来看一看 Python 执行引擎执行字节码的整体架构，其实就是一个 `for` 循环加上一个巨大的 `switch/case` 结构，熟悉 Windows 编程的朋友可以想象一下 Windows 下那个巨大的消息循环，没错，就是那样的结构：

```
[ceval.c]
/* Interpreter main loop */
PyObject* PyEval_EvalFrame(PyFrameObject *f)
{
    .....
    why = WHY_NOT;

```

```

for (;;) {
    .....
    fast_next_opcode:
        f->f_lasti = INSTR_OFFSET();
        /* Extract opcode and argument */
        opcode = NEXTOP();
        oparg = 0; /* allows oparg to be stored in a register because
                    it doesn't have to be remembered across a full loop */
        if (HAS_ARG(opcode))
            oparg = NEXTTARG();
    dispatch_opcode:
        switch (opcode) {

```

```

        case NOP:
            goto fast_next_opcode;

```

```

        case LOAD_FAST:
            .....
        }
    }
}

```

注意，这只是一个极其简化之后 Python 执行引擎的样子，如果想看 Python 执行引擎的尊容，请参考 `ceval.c` 中的源码。

在这个执行架构中，对字节码的一步一步地遍历正是通过几个宏来实现的：

```
[ceval.c]
#define INSTR_OFFSET() (next_instr - first_instr)
#define NEXTOP() (*next_instr++)
#define NEXTTARG() (next_instr += 2, (next_instr[-1]<<8) +
next_instr[-2])

```

在对 PyCodeObject 对象的分析中我们说过,Python 的字节码有的是带参数的,有的是没有参数的,而判断是否是带参字节码是通过 HAS_ARG 这个宏实现的。注意对不同的字节码, next_instr 的位移是不同的,但是无论如何, next_instr 总是指向 Python 下一条要执行的字节码,很像 x86 机器中的那个 PC 寄存器,对吗?

需要提到的一点是那个名叫 why 的神秘变量,这个家伙指示了在退出这个巨大的 for 循环时 Python 执行引擎的状态。你知道,世事难料,Python 执行引擎不一定每次执行都会寿终正寝,很有可能在执行到某条字节码的时候,产生了错误,没错,就是我们熟悉的那个“异常”, exception。这个时候,就需要知道 Python 执行引擎到底是因为什么原因结束了对字节码的执行,是正常结束呢,还是因为有错误发生,实在是活不下去了, why 义无反顾地负担起这一重任。

变量 why 的取值范围在 ceval.c 中被定义,其实也就是 Python 结束字节码执行时的状态:

```
[ceval.c]
/* Status code for main loop (reason for stack unwind) */
enum why_code {
    WHY_NOT = 0x0001, /* No error */
    WHY_EXCEPTION = 0x0002, /* Exception occurred */
    WHY_RERAISE = 0x0004, /* Exception re-raised by 'finally' */
    WHY_RETURN = 0x0008, /* 'return' statement */
    WHY_BREAK = 0x0010, /* 'break' statement */
    WHY_CONTINUE = 0x0020, /* 'continue' statement */
    WHY_YIELD = 0x0040 /* 'yield' operator */
};
```

好了,到现在,想必你已经对 Python 的执行引擎的大体框架了然于胸了。当 Python 的执行流程进入了 PyEval_EvalFrame 中的那个 for 循环,取出第一条字节码之后,第一张多米诺骨牌已经被推倒,命运不可阻挡地降临了。一条接一条的字节码像潮水一样汹涌而来,浩浩荡荡。天玄地黄,长风浩荡,我们展开双臂,迎接那些在天地间游荡的,那些在最平凡的面孔下蕴藏着最惊人的力量的精灵——字节码。

Python 源码剖析

——Python 执行引擎之一般表达式(1)

本文作者: Robert Chen(search.pythoner@gmail.com)

1 Declare.py

```
[declare.py]
i = 1
# LOAD_CONST 0
# STORE_NAME 0
```

```
s = "Python"
# LOAD_CONST 1
# STORE_NAME 1
```

```
d = {}
# BUILD_MAP 0
# STORE_NAME 2
```

```
l = []
# BUILD_LIST 0
# STORE_NAME 3
# LOAD_CONST 2
# RETURN_VALUE none
```

对于 `declare.py`，我们可以解析其生成的 `pyc` 文件，解析的结果如图 4 所示，看一看 `co_consts` 和 `co_names` 究竟都有些什么：

```
- <consts>
  <int value="1" />
  <internStr index="0" length="6" value="Python" />
  <NoneObject />
</consts>
- <names>
  <internStr index="1" length="1" value="i" />
  <internStr index="2" length="1" value="s" />
  <internStr index="3" length="1" value="d" />
  <internStr index="4" length="1" value="l" />
</names>
```

图 4

这些就是 `declare.py` 中关于程序运行的重要信息。这些信息在 `byte code` 的执行过程中必不可少，在随后的描述中可以看得很清楚。

`PyEval_EvalFram` 中还定义了一些宏，这些宏包括对栈的各种操作以及对 `tuple` 元素的访问操作，在执行 `byte code` 时，会大量使用这些宏。

```
#define GETITEM(v, i) PyTuple_GET_ITEM((PyTupleObject *) (v), (i))
```

```
#define BASIC_STACKADJ(n) (stack_pointer += n)
#define BASIC_PUSH(v) (*stack_pointer++ = (v))
#define BASIC_POP() (*--stack_pointer)
```

```
#define PUSH(v)      BASIC_PUSH(v)
#define POP()        BASIC_POP()
#define STACKADJ(n)  BASIC_STACKADJ(n)
```

我们首先来看一看对第一行 Python 代码的执行：

```
i = 1
#  LOAD_CONST  0      (1)
#  STORE_NAME  0      (i)
```

其中红色的部分表示当前的字节码指令对源文件中的哪个符号或常量进行了操作，或产生了影响。

在开始之前，我们先通过图 5 观察一下内存中栈以及 f->f_locals 的情况，前面说了，f->f_locals 将存储程序执行过程中的局部变量，实际也就是 Python 执行引擎的局部变量表，也是一举足轻重的主儿：)

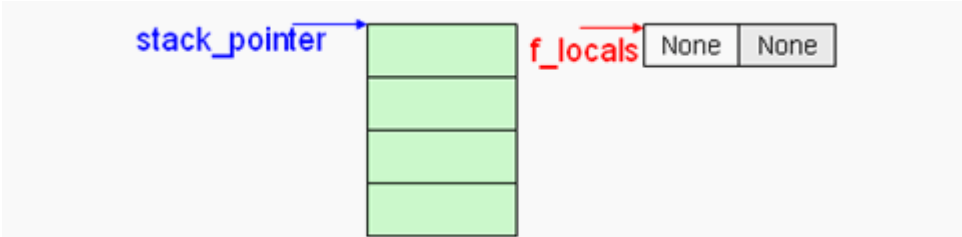


图 5

(注意：以后的阐述中，红色指针代表 f_locals，蓝色指针代表 stack_pointer)

对于 LOAD_CONST，执行引擎的动作如下

```
[LOAD_CONST]
x = GETITEM(consts, oparg);
Py_INCREF(x);
PUSH(x);
```

其中，GETITEM(consts, oparg)显然就是 GETITEM(consts, 0)。LOAD_CONST 的意图很明显，就是从 consts 中读取一个元素，然后将其压入到 Python 的运行时栈中。LOAD_CONST 完成后的情况如图 6 所示：

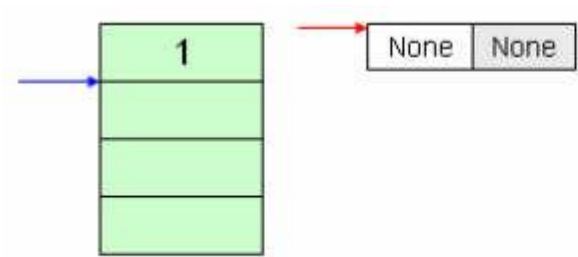


图 6

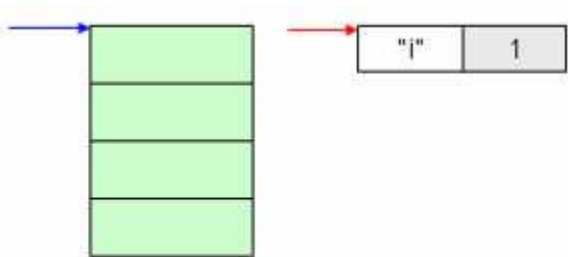


图 7

LOAD_CONST 只改变了运行时栈，而对 f->f_locals 没有任何改变。这个对 f->f_locals 的改变将在 STORE_NAME 中完成：

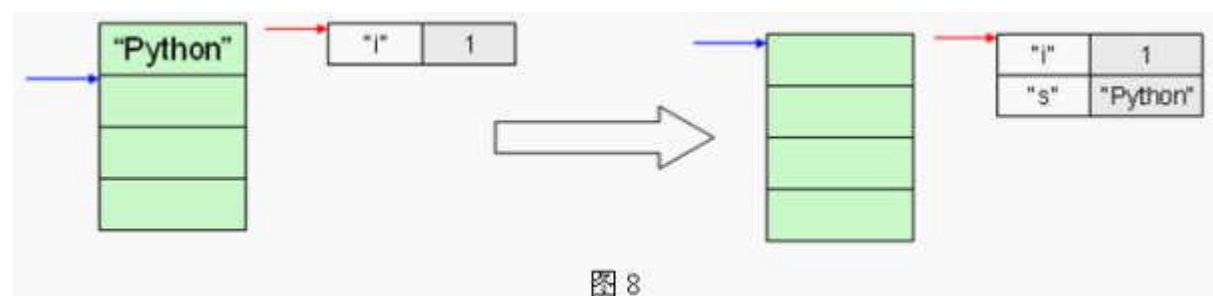
```
[STORE_NAME]
w = GETITEM(names, oparg);
v = POP();
if ((x = f->f_locals) != NULL)
{
    if (PyDict_CheckExact(x))
    {
        PyDict_SetItem(x, w, v);
    }
    else
    {
        PyObject_SetItem(x, w, v);
    }
    Py_DECREF(v);
}
```

在这里，我们只考虑 f->f_locals 确实是 PyDictObject 对象的情况。STORE_NAME 首先从 names 中读取一个元素，作为变量名，然后将刚才 LOAD_CONST 读取的元素作为变量值，将（变量名，变量值）元素对添加到 f->f_locals 中。

好了，到了这里，可以很清晰地看到 Python 代码中变量名与变量值在内存中是通过怎样的一种方式捆绑在一起的了。LOAD_CONST 完成后的情况如图 7 所示。现在，declare.py 中第一行代码执行完毕，第 2 行代码所产生的 byte code 实际上跟第一行代码是一样的，只是操作的参数不同了：

```
s = "Python"
#  LOAD_CONST  1      ('Python')
#  STORE_NAME  1      (s)
```

图 8 展示了这段 byte code 执行时栈和 f->f_locals 的动态变化：



在 declare.py 的第三行，我们见到了一点新鲜的东西，在这里，我们并不是简单地 Load 了，而是凭空创建了一个 PyDictObject 对象：

```
d = {}  
# BUILD_MAP 0  
# STORE_NAME 2 (d)
```

对于 BUILD_MAP, Python 运行时会创建一个空的 PyDictObject 对象，并把这个对象压入到运行时堆栈中：

```
[BUILD_MAP]  
x = PyDict_New();  
PUSH(x);
```

发现了吗？这里有一件很奇怪的事。从 Python 源代码编译出的字节码中，可以发现，BUILD_MAP 是一个带有参数的 byte code，从 opcode.h 中也能证实这一点，但是在这里，我们看到根本没有使用这个参数，可能，这又是“历史遗留”问题：）接下来的 STORE_NAME 我们已经非常熟悉了，看到这条 byte code，实际上我们就可以看到执行完毕后的情形了，如图 9 所示：

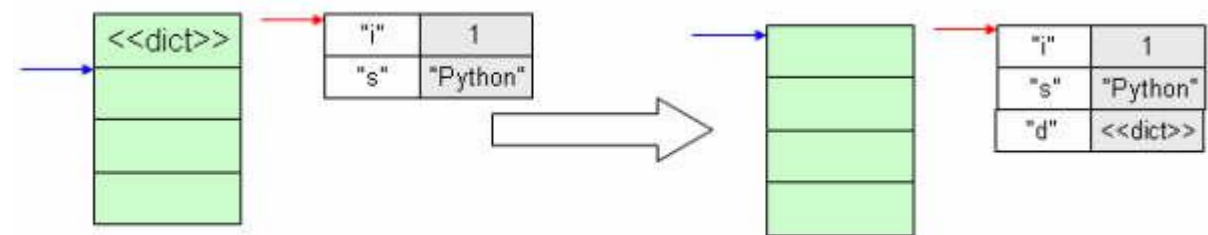


图 9

对于 declare.py 最后一行 Python 代码，很奇怪，居然编译出了四条 byte code：

```
l = []  
# BUILD_LIST 0  
# STORE_NAME 3 (1)  
# LOAD_CONST 2 (none)  
# RETURN_VALUE
```

对于 BUILD_LIST，猜想上会与 BUILD_MAP 是一路货色，但是，BUILD_LIST 比 BUILD_MAP 好太多了，它善待了字节码所带的参数，真正利用了这个参数，而不是 BUILD_MAP 那样，仅仅做个摆设：

```
[BUILD_LIST]  
x = PyList_New(oparg);  
if (x != NULL)  
{  
    for (; --oparg >= 0;)
```

```

{
    w = POP();
    PyList_SET_ITEM(x, oparg, w);
}
PUSH(x);
}

```

可以看到，如果 Python 源代码中创建的不是一个空的 list，那么在 BUILD_LIST 之前一定会有许多 LOAD_CONST 的操作，将元素压入栈中，在真正执行 BUILD_LIST 时，会一一从栈中弹出元素，加入到创建的 PyListObject 对象中。这一点我们下面会详细考察。

在执行了接下来的 STORE_NAME 后，似乎 declare.py 中指示的所有工作都完成了，最后两行是干嘛的呢？原来 Python 在执行了一段 Code Block 后，一定要返回一点东西，这两行 byte code 就是用来返回某些东西的：

```

[RETURN_VALUE]
retval = POP();
why = WHY_RETURN;

```

实际的返回值在 retval 中，是从栈中取得的，所以 RETURN_VALUE 前的那条 LOAD_CONST 就很清楚了，这家伙将返回值压入了栈了。可以看到，压入栈中的返回值是一个 NoneObject，实际上什么有价值的东西也没有返回，但这个过场还是要走的，不走，人民是不答应的：)

执行完整个 declare.py 的瞬间，栈已经变空了，而所有有用的信息都已经到了 f->f_locals 的掌握之中，如图 10 所示：

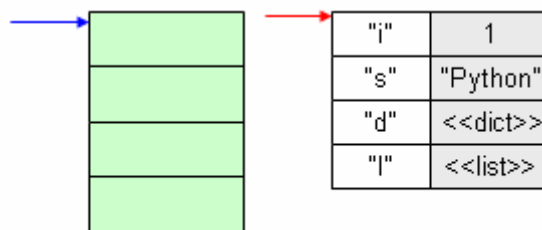


图 10

Python 源码剖析

——Python 执行引擎之一般表达式(2)

本文作者: Robert Chen(search.pythoner@gmail.com)

3.2 Simple.py

前面我们看了创建空的 dict 对象和空的 list，那么如果是创建非空的 dict 和 list 时，行为又是如何的呢。这个问题很有趣，我们通过 simple.py 来研究：

```

[simple.py]
i = 1
s = "Python"

```

```
d = {"1":1, "2":2}
l = [1, 2]
```

对于 simple.py, 执行时的常量表 co_consts 自然和 declare.py 不同了, 图 11 显示了与 simple.py 对应的 co_consts 和 co_names:

```
- <consts>
  <int value="1" />
  <internStr index="0" length="6" value="Python" />
  <internStr index="1" length="1" value="1" />
  <internStr index="2" length="1" value="2" />
  <int value="2" />
  <NoneObject />
</consts>
- <names>
  <internStr index="3" length="1" value="i" />
  <internStr index="4" length="1" value="s" />
  <internStr index="5" length="1" value="d" />
  <internStr index="6" length="1" value="l" />
</names>
```

图 11

编译得到的字节码中, 前两行 Python 代码的字节码都是相同的, 在创建非空的 dict 时, 字节码与 declare.py 中的不同了:

```
d = {"1":1, "2":2}
# BUILD_MAP 0
# DUP_TOP
# LOAD_CONST 2 ('1')
# LOAD_CONST 0 (1)
# ROT_THREE
# STORE_SUBSCR
# DUP_TOP
# LOAD_CONST 3 ('2')
# LOAD_CONST 4 (2)
# ROT_THREE
# STORE_SUBSCR
# STORE_NAME 2 (d)
```

对于 DUP_TOP, Python 会进行如下的动作:

```
[DUP_TOP]
v = TOP();
Py_INCREF(v);
PUSH(v);
```

不要注意的是, DUP_TOP 不光增加了栈顶元素的引用计数, 还将栈顶元素又一次压入到栈中。由于在 DUP_TOP 之前是一个 BUILD_MAP, 所以会将创建的 PyDictObject 对象的引用计数增加 1, 并再次压入该 PyDictObject 对象。

然后会从 `consts` 中将需要插入到 `PyDictObject` 对象中的第一个元素对的键和值读取出来，都压入到栈中，完成后情形如图 12 所示：

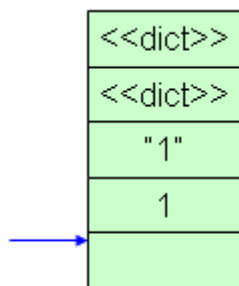


图 12

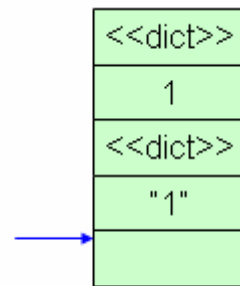
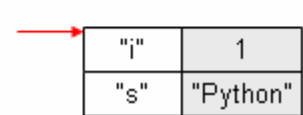


图 13



在接下来的 `ROT_THREE` 中，会做一些奇怪的动作：

```
[ROT_THREE]
v = TOP();
w = SECOND();
x = THIRD();
SET_TOP(w);
SET_SECOND(x);
SET_THIRD(v);
```

其中的 `SET_TOP` 等宏也是在 `PyEval_EvalFrame` 中定义的：

```
[ceval.c]
#define TOP()      (stack_pointer[-1])
#define SECOND()   (stack_pointer[-2])
#define THIRD()    (stack_pointer[-3])
#define FOURTH()   (stack_pointer[-4])
#define SET_TOP(v)  (stack_pointer[-1] = (v))
#define SET_SECOND(v) (stack_pointer[-2] = (v))
#define SET_THIRD(v) (stack_pointer[-3] = (v))
#define SET_FOURTH(v) (stack_pointer[-4] = (v))
```

其实 `ROT_THREE` 并没有干什么有实际意义的事，所做的就是将栈顶的三个元素来了个乾坤大挪移。`ROT_THREE` 操作完成后情形如图 13 所示。

随后的 `STORE_SUBSCR` 会将元素插入到 `PyDictObject` 对象中去：

```
[STORE_SUBSCR]
w = TOP();
v = SECOND();
u = THIRD();
STACKADJ(-3);
/* v[w] = u */
```

```
PyObject_SetItem(v, w, u);
Py_DECREF(u);
Py_DECREF(v);
Py_DECREF(w);
```

随着 `STACKADJ` 的执行，栈顶指针回退了 3 格，所以 `STORE_SUBSCR` 执行完后，运行时栈里又只剩下了最初由 `BUILD_MAP` 创建的 `PyDictObject` 对象。到这里，就完成了将一个元素对插入到 `PyDictObject` 的操作。剩下的不过是重复上面的动作，将第二个元素对插入到 `PyDictObject` 对象中去。最后由我们的老朋友 `STORE_NAME` 完成将这个 `PyDictObject` 对象添加到 `f->f_locals` 中的工作。最后的情形如图 14 所示：

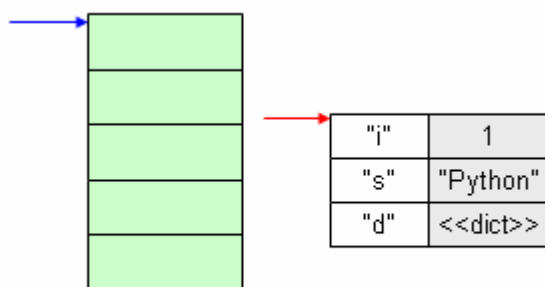


图 14

在成功创建了 `PyDictObject` 对象之后，会创建一个非空的 `PyListObject` 对象：

```
l = [1, 2]
# LOAD_CONST 0 (1)
# LOAD_CONST 4 (2)
# BUILD_LIST 2
# STORE_NAME 3 (1)
```

从这里可以看到，确实 Python 会首先将会填入 `PyListObject` 对象的元素先从 `consts` 读入，压入运行时栈，然后如之前描述的，`BUILD_LIST` 再创建 `PyListObject` 对象之后，会从栈中依次将元素读出，从后至前地填入 `PyListObject` 对象中。

3.3 simple2.py

到了这里，对于一般的声明语句或常量赋值表达式，我们都已经了如指掌。下面，通过研究如下的 `simple2.py`，考察变量赋值，变量运算以及最基本的 `print` 操作，完成对一般 Python 语句的考察：

```
[simple2.py]
a = 5
b = a
c = a + b
print c
```

编译完成后，其 `co_consts` 和 `co_names` 如图 15 所示：

```

- <consts>
  <int value="5" />
  <NoneObject />
</consts>
- <names>
  <internStr index="0" length="1" value="a" />
  <internStr index="1" length="1" value="b" />
  <internStr index="2" length="1" value="c" />
</names>

```

图 15

第一条 Python 代码现在我们已经很熟悉了，不用再费唇舌。现在看第二条，变量间的赋值：

```

b = a
#  LOAD_NAME    0      (a)
#  STORE_NAME   1      (b)

```

对于 STORE_NAME，已经摸清楚它到底干了些什么。而对于 LOAD_NAME，我们还是第一次遇到：

```

[LOAD_NAME]
    w = GETITEM(names, oparg);
    if ((v = f->f_locals) == NULL) {
        //report error
        break;
    }
    if (PyDict_CheckExact(v)) {
        x = PyDict_GetItem(v, w); //[1]
        Py_XINCREF(x);
    }
    else {
        x = PyObject_GetItem(v, w);
        if (x == NULL && PyErr_Occurred()) {
            if (!PyErr_ExceptionMatches(PyExc_KeyError))
                break;
            PyErr_Clear();
        }
    }
    if (x == NULL) {
        x = PyDict_GetItem(f->f_globals, w); //[2]
        if (x == NULL) {
            x = PyDict_GetItem(f->f_builtins, w); //[3]
            if (x == NULL) {
                format_exc_check_arg(
                    PyExc_NameError,

```

```

NAME_ERROR_MSG ,w);

break;

}

}

Py_INCREF(x);

}

PUSH(x);

```

首先，会从 `names` 中取出第 0 个元素，参考前面的 `simple2.py` 编译的结果，可知是一个 `PyStringObject` 对象“a”。然后检查当前 `PyFrameObject` 中所维护的局部变量集合 `f->f_locals`，如果这个东西不存在，那么直接返回（why?），如果局部变量集合存在，就会进行一系列的搜索动作：

[1]：在局部变量集合 `f_locals` 中搜索“a”。

[2]：如果 `f_locals` 中没有“a”，则在全局变量集合 `f_globals` 中搜索“a”。

[3]：如果 `f_globals` 中没有“a”，则在 Python 运行时内建变量集合 `f_builtins` 中搜索“a”。

如果搜索到了与“a”对应的元素，那么就将该元素返回。在 `simple2.py` 这个例子中，第一条 Python 代码的执行会在 `f_locals` 中插入（“a”，1）的元素对，所以这里会返回 `PyIntObject` 对象 1。

如果到了最后还搜索不到“a”，那么表示有错误发生，程序引用了一个不存在的符号。

这样的行为正是 Python 官方文档中所描述的变量的搜索会沿着局部作用域，全局作用域，内建作用域依次上溯，直至搜索成功或全部搜完三个作用域。

现在很清楚了，在 `b = a` 执行完成后，内存中的情形如图 16 所示：

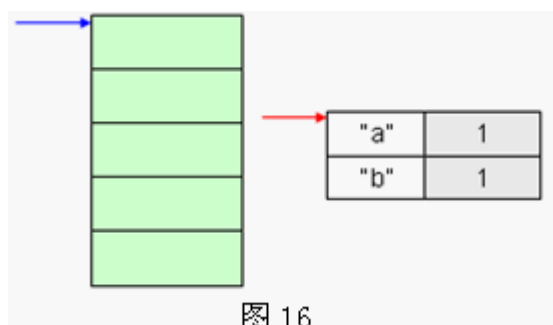


图 16

而从前面我们对 `PyIntObject` 对象的分析可知，这两个 1 实际上指向内存中的同一个 `PyIntObject` 对象。

现在 `a`，`b` 都是合法而有效的变量了，它们的结合就变得很有趣了：

```

c = a + b
#  LOAD_NAME    0          (a)
#  LOAD_NAME    1          (b)
#  BINARY_ADD
#  STORE_NAME   2          (c)

```

首先会将 `a` 和 `b` 所对应的变量值从 `f_locals` 中读取出来，压入运行时栈，然后通过 `BINARY_ADD` 计算两个变量的和，假设结果为 `num`，最后就通过 `STORE_NAME` 将（“c”，`num`）元素对插入到 `f_locals` 中。过程非常清晰，而现在我们感兴趣的就是那个从两个现有对象创造出新的对象的 `BINARY_ADD`：


```

[BINARY_ADD]
    w = POP();
    v = TOP();
    if (PyInt_CheckExact(v) && PyInt_CheckExact(w)) {
        /* INLINE: int + int */
        register long a, b, i;
        a = PyInt_AS_LONG(v);
        b = PyInt_AS_LONG(w);
        i = a + b;
        if ((i^a) < 0 && (i^b) < 0)
            goto slow_add;
        x = PyInt_FromLong(i);
    }
    else if (PyString_CheckExact(v) &&
        PyString_CheckExact(w)) {
        x = string_concatenate(v, w, f, next_instr);
        /* string_concatenate consumed the ref to v */
        goto skip_decref_vx;
    }
    else {
        slow_add:
        x = PyNumber_Add(v, w);
    }
    Py_DECREF(v);
skip_decref_vx:
    Py_DECREF(w);
    SET_TOP(x);
    if (x != NULL) continue;
    break;

```

如果参与运算的两个对象都是 `PyIntObject` 对象，会直接将 `PyIntObject` 中的 `value` 提取出来相加，然后根据相加的结果创建新的 `PyIntObject` 对象作为结果返回；同样，如果是 `PyStringObject` 对象，也会选择 `string_concatenate` 加快速度。如果参与运算的对象落在了这两种有加速机制的情况之外，那么很不幸，只能通过 `PyNumber_Add` 完成运算，`PyNumber_Add` 会进行大量的类型判断，寻找操作函数等额外工作，速度会比前两种加速机制慢上很多。一般来说，Python 在 `PyNumber_Add` 中会首先检查 `PyNumberMethods` 中的 `nb_add` 能否完成在 `v` 和 `w` 上的加法运算，如果不能，还会检查 `PySequenceMethods` 中的 `sq_concat` 能否完成，如果都不能，Python 也是有心杀敌，无力回天了，那也只好报告错误了。可以看到，Python 2.4.1 在这里假设了用户在使用 Python 时，大量进行的加法操作是整数的加法和字符串的连接。其实如果你的程序中涉及到了大量的浮点运算，那你完全可以修改 `BINARY_ADD` 的代码，为浮点加法运算建立快速通道。实际上，对于加，减，乘，除这些操作，你都可以根据自己程序的实际情况对 Python 的代码进行改动，这样的改动应该对提升程序运行效率有很大的好处。

最后看一看 `print` 的动作，前面分析的 `byte code` 都是 Python 自己跟自己玩，这个 `print` 则是

Python 跟咱们用户玩，也是 Python 中最常用和易用的用户交互方式：

```
print c
#  LOAD_NAME      2          (c)
#  PRINT_ITEM
#  PRINT_NEWLINE
```

其中 `LOAD_NAME` 首先将刚才获得的那个加法运算的和 `c` 从 `f_locals` 中读取出来，压入运行时栈，然后通过 `PRINT_ITEM` 完成输出的魔法：

```
[PRINT_ITEM]
    v = POP();
    if (stream == NULL || stream == Py_None) {
        w = PySys_GetObject("stdout");
    }
    Py_XINCREF(w);
    if (w != NULL && PyFile_SoftSpace(w, 0))
        err = PyFile_WriteString(" ", w);
    if (err == 0)
        err = PyFile_WriteObject(v, w, Py_PRINT_RAW);
    .....
    stream = NULL;
```

实际上输出的时候会进行很多动作，但是在这里，我们只考虑其大致的流程。首先会将待输出的对象从运行时栈中取出。然后，判断如果 `stream` 为 `NULL`，则将 `w` 设为标准输出流。这个 `stream` 是什么东西呢？它实际上也是一个 `PyObject` 对象：

```
PyObject *stream = NULL; /* for PRINT opcodes */
```

如果输出的时候，是通过如下的 Python 代码：`print >> file, "str"`

那么所产生的 `byte code` 中还有一个 `PRINT_ITEM_TO`：

```
[PRINT_ITEM_TO]
w = stream = POP();
```

显然，这时在运行时栈中，在待输出对象之前，还会有一个对象，即输出的目标。在执行 `PRINT_ITEM_TO` 时，输出的目标就赋给了 `stream`，同时也赋给了 `w`。所以实际上 `stream` 是作为一个判断条件来使用的，真正使用的输出目标是 `w`。要多使用这一个 `stream` 的原因是 `w` 在别的 `byte code` 中可能还会使用到，所以无法通过判断 `w` 是否为 `NULL` 来确定是否输出到标准输出流。

可以看到，在 `PRINT_ITEM` 最后，又将 `stream` 设为了 `NULL`，又可以为下次输出时的判断做准备了。

在获得了输出的目标和待输出的对象后，`PRINT_ITEM` 将通过 `PyFile_WriteObject -> PyObject_Print -> internal_print` 的调用序列最终调用 `v->ob_type->tp_print`，即待输出对象自身所携带的输出函数进行输出。如果对象没有定义 `tp_print`，那么会先调用 `tp_str` 或 `tp_repr` 获得对象的字符串表示形式，将字符串输出，如果这也失败了，Python 也无能为力了，只好返回失败信息。