Python/C API

Выпуск 3.8.8

Гвидо ван Россум
и команда разработчиков Python

августа 12, 2021

https://Digitology.tech
Email: tweakit@bk.ru

Оглавление

В данном руководстве документируются API, используемое C и C++ программистами, которые хотят писать модули расширения или встраивать Python. Это дополнение к extending-index, которое описывает общие принципы написания расширений, но не документирует подробно функции API.

ГЛАВА 1

Введение

Программный интерфейс приложения к Python дает программистам на C и C++ доступ к интерпретатору Python на разных уровнях. API одинаково можно использовать из C++, но для краткости его обычно называют Python/C API. Есть две принципиально разные причины использования API Python/C. Первая причина - написать *модули расширения* для определенных целей; они являются модулями C, расширяющими интерпретатор Python. Это наверное самое общее пользование. Вторая причина - использовать Python как компонент в более крупном приложении; этот метод обычно называют *встраивание* Python в приложение.

Написание модуля расширения является относительно хорошо понятным процессом, где хорошо работает подход «поваренной книги». Существует несколько инструментов, которые в некоторой степени автоматизируют процесс. Хотя люди встраивают Python в другие приложения с самого начала своего существования, процесс встраивания Python менее прост, чем написание расширения.

Многие функции API полезны независимо от того, встраиваете ли вы или расширяете Python; кроме того, большинство приложений, которые встраивают Python, также должны предоставить пользовательское расширение, поэтому, вероятно, рекомендуется ознакомиться с написанием расширения, прежде чем пытаться встроить Python в реальное приложение.

1.1 Стандарты кодирования

Если вы пишете C код для включения в CPython, вы **должны** следовать рекомендациям и стандартам, определенным в **PEP 7**. Эти правила применяются независимо от версии Python, над которой вы работаете. После этих соглашений не требуется для ваших собственных сторонних модулей расширений, если вы в конечном итоге не собираетесь вносить их в Python.

1.2 Подключение файлов

Все определения функций, типов и макросов, необходимые для использования Python/C API, включаются в код следующей строкой:

```
#define PY_SSIZE_T_CLEAN
#include <Python.h>
```

Это подразумевает включение следующих стандартных заголовков: `<stdio.h>`, `<string.h>`, `<errno.h>`, `<limits.h>`, `<assert.h>` и `<stdlib.h>` (при наличии).

Примечание: Поскольку Python могут определять некоторые предпроцессорные определения, которые влияют на стандартные заголовки в некоторых системах, вы *должен* включить `Python.h` перед включением каких-либо стандартных заголовков.

Рекомендуется всегда определять `PY_SSIZE_T_CLEAN` перед включением `Python.h`. Описание этого макроса см. в разделе *Анализ аргументов и сборка значений*.

Все видимые пользователем имена, определенные Python.h (за исключением имен, определенных включенными стандартными заголовками), имеют один из префиксов `Py` или `_Py`. Имена, начинающиеся с `_Py`, предназначены для внутреннего использования Python реализацией и не должны использоваться писателями расширений. Имена членов структуры не имеют зарезервированного префикса.

Примечание: Пользовательский код никогда не должен определять имена, начинающиеся с `Py` или `_Py`. Это сбивает читателя с толку и ставит под угрозу переносимость пользовательского кода на будущие версии Python, которые могут определять дополнительные имена, начиная с одного из этих префиксов.

Заголовочные файлы заголовков обычно устанавливаются с Python. На Unix они расположены в каталогах *`prefix`*`/include/pythonversion/` и *`exec_prefix`*`/include/pythonversion/`, где `prefix` и `exec_prefix` определены соответствующими параметрами к сценария Python **`configure`** и *версии* - `'%d.%d' % sys.version_info[:2]`. В Windows заголовки устанавливаются в *`prefix`*`/include`, где `prefix` - каталог установки, указанный установщиком.

Чтобы включить заголовки, поместите оба каталога (если они отличаются) в путь поиска компилятора для инклудников. *Не* помещайте родительские каталоги в путь поиска, а затем использовать `#include <pythonX.Y/Python.h>`; это приведет к разрыву мультиплатформенных сборок, поскольку независимые от платформы заголовки под `prefix` включают конкретные заголовки платформы из `exec_prefix`.

Пользователи C++ должны отметить, что хотя API определяется полностью с помощью C, заголовочные файлы правильно объявляют точки входа для `extern "C"`. В результате нет необходимости делать что-либо особенное для использования API из C++.

1.3 Полезные макросы

В файлах заголовков Python определено несколько полезных макросов. Многие из них определены ближе к месту их использования (например, *`Py_RETURN_NONE`*). Другие из более общих утилит определены здесь. Это не обязательно полный список.

`Py_UNREACHABLE``()`

Использовать его, если у вас есть путь до кода, который вы не ожидаете получить. Например, в `default:` клаузула в `switch` инструкции, для которого все возможные значения покрыты в `case` инструкции. Использовать это в местах, где у вас может возникнуть искушение вызвать `assert(0)` или `abort()`.

Добавлено в версии 3.7.

Py_ABS(x)
Возвращает абсолютное значение `x`.

Добавлено в версии 3.3.

Py_MIN(x, y)
Возвращает минимального значение между `x` и `y`.

Добавлено в версии 3.3.

Py_MAX(x, y)
Возвращает максимальное значение между `x` и `y`.

Добавлено в версии 3.3.

Py_STRINGIFY(x)
Преобразовать `x` в C строку. Например, `Py_STRINGIFY(123)` возвращает `"123"`.

Добавлено в версии 3.4.

Py_MEMBER_SIZE(type, member)
Возвращает размер структуры `(type)`, `member` в байтах.

Добавлено в версии 3.6.

Py_CHARMASK(c)
Аргумент должен быть символом или целым числом в диапазоне [-128, 127] или [0, 255]. Этот макрос возвращает `c` приведение к `unsigned char`.

Py_GETENV(s)
Как и `getenv(s)`, возвращает `NULL` если `-E` был передан в командной строке (т.е. если `Py_IgnoreEnvironmentFlag` установлен).

Py_UNUSED(arg)
Использовать его для неиспользуемых аргументов в определении функции, чтобы скрыть предупреждения компилятора. Пример: `int func(int a, int Py_UNUSED(b)) { return a; }`.

Добавлено в версии 3.4.

Py_DEPRECATED(version)
Используется для устаревших объявлений. Макрос должен быть размещен перед именем символа.

Пример:

```
Py_DEPRECATED(3.8) PyAPI_FUNC(int) Py_OldFunction(void);
```

Изменено в версии 3.8: Добавлена поддержка MSVC.

PyDoc_STRVAR(name, str)
Создает переменную с именем `name`, которая может использоваться в докстрингах. Если Python построен без докстрингов, значение будет пустым.

Использовать *PyDoc_STRVAR* для докстрингов для поддержки Python сборки без докстрингов, как указано в разделе **PEP 7**.

Пример:

```
PyDoc_STRVAR(pop_doc, "Remove and return the rightmost element.");

static PyMethodDef deque_methods[] = {
    // ...
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
    {"pop", (PyCFunction)deque_pop, METH_NOARGS, pop_doc},
    // ...
}
```

PyDoc_STR(str)

Создает докстринг для данной входной строки или пустой строки, если докстринги отключены.

Использовать *PyDoc_STR* в задании докстринга для поддержки Python сборки без докстрингов, как указано в разделе **PEP 7**.

Пример:

```
static PyMethodDef pysqlite_row_methods[] = {
    {"keys", (PyCFunction)pysqlite_row_keys, METH_NOARGS,
        PyDoc_STR("Returns the keys of the row.")},
    {NULL, NULL}
};
```

1.4 Объекты, типы и ссылочные счетчики

Большинство Python/C функций API имеют один или несколько аргументов, а также возвращает значение типа *PyObject**. Этот тип является указателем на непрозрачный тип данных, представляющий произвольный объект Python. Поскольку все Python типы объектов в большинстве ситуаций обрабатываются языком Python одинаково (например, назначения, правила область видимости и передача аргументов), то их следует представлять одним типом C. Почти все Python объекты находятся в куче: вы никогда не объявляете автоматическую или статическую переменную типа *PyObject*, можно объявлять только переменные указателей типа *PyObject**. Единственным исключением являются объекты типа; поскольку они никогда не должны быть удалены, они обычно являются статическими *PyTypeObject* объектами.

Все Python объекты (даже Python целые числа) имеют *type* и *счетчик ссылок* <*счетчик ссылок*. type объект определяет тип объекта (например, целое число, список или пользовательская функция; существует много других, как объяснено в types). Для каждого из известных типов существует макрос, чтобы проверить, является ли объект такого типа; для сущность `PyList_Check(a)` имеет значение true, если (и только если) объект, на который указывает *a*, является Python списком.

1.4.1 Счетчик ссылок

Счетчик ссылок важен, поскольку современные компьютеры имеют ограниченный (и часто сильно ограниченный) размер памяти; подсчитывает количество различных мест, имеющих ссылку на объект. Таким местом может быть другой объект, или глобальная (или статическая) переменная C, или локальная переменная в некоторой функции C. Когда количество ссылок на объект становится равным нулю, объект освобождается. Если он содержит ссылки на другие объекты, их количество ссылок уменьшается. Эти другие объекты могут быть, по очереди, освобождены, если это уменьшение делает их счетчиком ссылок нулевым, и так далее. (Есть очевидная проблема с объектами, которые ссылаются друг на друга здесь; пока решение - «не делай этого».)

Счетчики ссылок всегда обрабатываются явным образом. Нормальным способом является использование макроса *Py_INCREF()* для приращения количества ссылок на объект на единицу, а *Py_DECREF()* для его уменьшения на единицу. Макрос *Py_DECREF()* является значительно более сложным, чем инкремент, поскольку он должен проверять, становится ли счетчик ссылок нулевым, и затем вызвать деаллокатор объекта. Деаллокатор - это указатель функции, содержащийся в структуре типа объекта. Деаллокатор для конкретного типа обеспечивает уменьшение количества ссылок для других объектах, содер-

жащихся в объекте, если это составной тип объекта, такой как список, а также выполняет любую необходимую дополнительную финализацию. Вероятность переполнения счетчика ссылок отсутствует; в как минимум столько же битов используется для хранения счетчика ссылок, сколько существует различных ячеек памяти в виртуальной памяти (предполагая `sizeof(Py_ssize_t) >= sizeof(void*)`). Таким образом, приращение счетчика ссылок является простой операцией.

Нет необходимости увеличивать счетчик ссылок объекта для каждого локальной переменной, содержащая указатель на объект. Теоретически счетчик ссылок объекта увеличивается на единицу, когда переменная указывает на нее, и она уменьшается на единицу, когда переменная выходит за пределы области видимости. Однако эти два компенсируют друг друга, поэтому в конце счетчик ссылок не изменился. В единственная реальная причина использовать счетчик ссылок - это предотвратит освобождение, пока наша переменная указывает на него. Если мы знаем, что там по крайней мере еще одна ссылка на объект, который живет по крайней мере до тех пор, пока нашей переменной, нет необходимости временно увеличивать счетчик ссылок. Важная ситуация, когда это возникает в объектах, которые передаются как аргументы функций C в модуле расширения, которые вызываются из Python; механизм вызова гарантирует, что он будет содержать ссылку на каждый аргумент для продолжительности вызова.

Однако распространенная ошибка, является извлечение объекта из списка и удержание его в течение некоторого времени без увеличения количества ссылок. Возможно, еще одна операция удалит объект из списка, уменьшив количество ссылок и, возможно, освободив его. Реальная опасность заключается в том, что наивная операция может вызвать произвольный Python код, который может это сделать; существует кодовый путь, который позволяет управлять потоком обратно к пользователю из *Py_DECREF()*, поэтому практически любая операция потенциально опасна.

Безопасный подход заключается в том, чтобы всегда использовать общие операции (функции, имя которых начинается с `PyObject_`, `PyNumber_`, `PySequence_` или `PyMapping_`). Эти операции всегда увеличивают количество ссылок на возвращаемый объект. При этом вызывающий несет ответственность за вызов *Py_DECREF()*, когда он получает результат; это вскоре становится второй натурой.

Подробнее о количестве ссылок

Поведение счетчика ссылок функций в API Python/C лучше всего объяснить с точки зрения *владельца ссылки*. Владение относится к ссылкам, никогда не к объектам (объекты не принадлежат: они всегда являются общими). «Владение ссылкой» означает ответственность за вызов Py_DECREF по ней, когда ссылка больше не нужна. Собственность также может быть передана, что означает, что код, который получает собственность на ссылку, затем становится ответственным за ее окончательное уменьшение путем вызова *Py_DECREF()* или *Py_XDECREF()*, когда он больше не нужен — или передает эту ответственность (обычно вызывающему). Когда функция передает собственность на ссылку своему вызывающему, о вызывающем говорят, что он получает на *новую* ссылку. При отсутствии передачи собственности вызывающий *заимствует* ссылки. Ничего не нужно делать для заимствования ссылки.

И наоборот, когда вызывающая функция передает ссылку на объект, есть две возможности: функция *кражи* ссылки на объект или нет. *Кража ссылки* означает, что при передаче ссылки на функцию эта функция предполагает, что она теперь является владельцем этой ссылки, и вы больше не несет за нее ответственность.

Немногие функции крадут ссылки; два заметных исключения - *PyList_SetItem()* и *PyTuple_SetItem()*, которые крадут ссылку на элемент (но не на кортеж или список, в который помещен элемент!). Эти функции были разработаны для кражи ссылки из-за общей идиомы для заполнения кортежа или списка вновь созданными объектами; например, код создания кортежа `(1, 2, "three")` может выглядеть так (забывая об обработке ошибок на данный момент; лучшая реализация кода этого показана ниже):

```
PyObject *t;

t = PyTuple_New(3);
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
PyTuple_SetItem(t, 0, PyLong_FromLong(1L));
PyTuple_SetItem(t, 1, PyLong_FromLong(2L));
PyTuple_SetItem(t, 2, PyUnicode_FromString("three"));
```

Здесь *PyLong_FromLong()* возвращает новую ссылку, которую тут же крадет *PyTuple_SetItem()*. Если вы хотите сохранить использование объекта, хотя ссылка на него будет украдена, Использовать *Py_INCREF()* для захвата другой ссылки перед вызовом функции кражи ссылок.

Между прочим, *PyTuple_SetItem()* является *только* способом установки элементов кортежа; *PySequence_SetItem()* и *PyObject_SetItem()* отказываются делать это, поскольку кортежи являются неизменяемым типом данных. Вы должны использовать *PyTuple_SetItem()* только для кортежей, которые вы создаете сами.

Эквивалентный код для заполнения списка можно записать с помощью *PyList_New()* и *PyList_SetItem()*.

Однако на практике эти способы создания и заполнения кортежа или списка используются редко. Существует универсальная функция *Py_BuildValue()*, которая может создавать наиболее распространенные объекты из C значения, управляемые *строка формата*. Например, два вышеперечисленных блока кода могут быть заменены следующими блоками (которые также обеспечивают проверку ошибок):

```
PyObject *tuple, *list;

tuple = Py_BuildValue("(iis)", 1, 2, "three");
list = Py_BuildValue("[iis]", 1, 2, "three");
```

Гораздо чаще использовать *PyObject_SetItem()* и друзей с элементами, ссылки на которые вы только заимствуете, как аргументы, которые были переданы в функцию, которую вы пишете. В этом случае их поведение в отношении отсчетов ссылок гораздо безопаснее, так как не нужно увеличивать счетчик ссылок, чтобы можно было выдать ссылку («украсть»). Например, эта функция устанавливает все элементы списка (фактически любую изменяемую последовательность) для данного элемента:

```
int
set_all(PyObject *target, PyObject *item)
{
    Py_ssize_t i, n;

    n = PyObject_Length(target);
    if (n < 0)
        return -1;
    for (i = 0; i < n; i++) {
        PyObject *index = PyLong_FromSsize_t(i);
        if (!index)
            return -1;
        if (PyObject_SetItem(target, index, item) < 0) {
            Py_DECREF(index);
            return -1;
        }
        Py_DECREF(index);
    }
    return 0;
}
```

Ситуация несколько отличается для функции возвращающей значения. Хотя передача ссылки на большинство функций не изменяет ответственности владельца для этой ссылки, многие функции, которые

возвращают ссылку на объект, передают собственность на ссылку. Причина проста: во многих случаях возвращаемый объект создается на лету, а получаемая ссылка является единственной ссылкой на объект. Поэтому общие функции, возвращающие ссылки на объекты, такие как *PyObject_GetItem()* и *PySequence_GetItem()*, всегда возвращают новую ссылку (вызывающий становится владельцем ссылки).

Важно понимать, что независимо от того, владеете ли вы ссылкой, возвращаемое зависит только от того, какую функцию вы вызываете — *оперение* (тип объекта, переданного функции в качестве аргумента) *не входит в него!* Таким образом, если вы извлекаете элемент из списка с помощью *PyList_GetItem()*, вы не владеете ссылкой — но если вы получаете тот же элемент из того же списка с помощью *PySequence_GetItem()* (что происходит при использовании тех же аргументов), вы владеете ссылкой на возвращенный объект.

Вот пример того, как можно написать функцию, которая вычисляет сумму элементов в списке целых чисел; один раз с использованием *PyList_GetItem()* и один раз с использованием *PySequence_GetItem()*.:

```
long
sum_list(PyObject *list)
{
    Py_ssize_t i, n;
    long total = 0, value;
    PyObject *item;

    n = PyList_Size(list);
    if (n < 0)
        return -1; /* Не список */
    for (i = 0; i < n; i++) {
        item = PyList_GetItem(list, i); /* Не может потерпеть неудачу */
        if (!PyLong_Check(item)) continue; /* Пропускать нецелые числа */
        value = PyLong_AsLong(item);
        if (value == -1 && PyErr_Occurred())
            /* Целое число слишком велико, чтобы поместиться в C long,
↪спасайтесь */
            return -1;
        total += value;
    }
    return total;
}
```

```
long
sum_sequence(PyObject *sequence)
{
    Py_ssize_t i, n;
    long total = 0, value;
    PyObject *item;
    n = PySequence_Length(sequence);
    if (n < 0)
        return -1; /* Не имеет длины */
    for (i = 0; i < n; i++) {
        item = PySequence_GetItem(sequence, i);
        if (item == NULL)
            return -1; /* Не последовательность или другой сбой */
        if (PyLong_Check(item)) {
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
            value = PyLong_AsLong(item);
            Py_DECREF(item);
            if (value == -1 && PyErr_Occurred())
                /* Целое число слишком велико, чтобы поместиться в C long,
↪спасайтесь */
                return -1;
            total += value;
        }
        else {
            Py_DECREF(item); /* Отменить собственность на ссылку */
        }
    }
    return total;
}
```

1.4.2 Типы

Существует несколько других типов данных, которые играют значительную роль в Python/C API; большинство из них являются простыми типами C, такими как `int`, `long`, `double` и `char*`. Несколько типов структур используемых для описания статических таблиц используемых для перечисления функций, экспортируемых модулем или атрибутов данных нового типа объекта, а другой используется для описания значения комплексного числа. Они будут обсуждаться вместе с функциями, которые их используют.

1.5 Исключения

Программист Python должен иметь дело только с исключениями, если требуется специальная обработка ошибок; необработанные исключения автоматически передаются вызывающему, затем вызывающему и т.д. до тех пор, пока они не достигнут верхнего уровня интерпретатор, где они сообщаются пользователю вместе со стеком трейсбэка.

Однако для C-программистов проверка ошибок всегда должна быть явной. Все функции в API Python/C могут вызывать исключения, если в документации функции не содержится явного утверждения об обратном. Как правило, когда функция обнаруживает ошибку, она устанавливает исключение, отбрасывает все ссылки на объекты, которыми она владеет, и возвращает индикатор ошибки. Если не задокументировано иначе, этот индикатор является либо `NULL`, либо `-1`, в зависимости от возвращаемого типа функцией. Несколько функций возвращают логический результат true/false, а значение false указывает на ошибку. Очень немногие функции не возвращает явного индикатора ошибки или имеют неоднозначное возвращаемое значение, и требуют явного тестирования на наличие ошибок с помощью *`PyErr_Occurred()`*. Эти исключения всегда задокументированы явно.

Состояние исключения сохраняется в хранилище для каждого потока (это эквивалентно использованию глобального хранилища в непоточном приложении). Поток может находиться в одном из двух состояний: произошло исключение или нет. Функция *`PyErr_Occurred()`* может использоваться для проверки этого: она возвращает заимствованную ссылку на объект типа исключения, когда произошло исключение, и `NULL` в противном случае. Существует ряд функций для установки состояния исключения: *`PyErr_SetString()`* является наиболее распространенной (хотя и не самой общей) функцией для установки состояния исключения, а *`PyErr_Clear()`* очищает состояние исключения.

Состояние полного исключения состоит из трех объектов (все из которых могут быть `NULL`): тип особой ситуации, соответствующий значению исключения и трейсбэк. Они имеют те же значения, что и Python результат `sys.exc_info()`; однако они не одинаковы: объекты Python представляют собой последнее исключение, обрабатываемое Python `try`... `except` инструкцией, в то время как исключение уровня C состояние существует только при передаче исключения между функциями C до тех пор, пока оно не

достигнет основного цикла интерпретатора Python байт-кода, который обеспечивает его передачу `sys.exc_info()` и друзьям.

Обратите внимание, что начиная с Python 1.5 предпочтительный поточно-ориентированный способ доступа к состоянию исключения из кода Python заключается в вызове функции: func: *sys.exc_info*, который возвращает состояние исключения для каждого потока для кода Python. Кроме того, семантика обоих способов доступа к состояние исключений изменилась так, что функция, которая перехватывает исключение, сохранит и восстановит состояние исключения потока, чтобы сохранить состояние исключения своего вызывающего. Это предотвращает распространенные ошибки при обработке исключений кода вызванные тем, что функция, выглядящая невинно, перезаписывает обрабатываемое исключение; это также уменьшает часто нежелательное расширение срока службы объектов, на которые ссылаются кадры стека в трейсбэке.

Как правило, функция, которая вызывает другую функцию для выполнения некоторых задач, должна проверить, вызвало ли вызываемая функция исключение, и если да, передать состояние исключения вызывающей стороне. Следует отказаться от любого объекта ссылки, которыми она владеет, и возвращать индикатор ошибки, но *не* должна устанавливать другое исключение — которое перезапишет только что возникшее исключение, и потерявшее важную информацию о точной причине ошибки.

В приведенном выше `sum_sequence()` примере показан простой пример обнаружения исключений и их передачи. Так бывает, что в этом примере нет необходимости очищать какие-либо собственные ссылки при обнаружении ошибки. В следующем примере функция показывает некоторую очистку от ошибок. Во-первых, чтобы напомнить вам, почему вы любите Python, мы показываем эквивалентный Python код:

```
def incr_item(dict, key):
    try:
        item = dict[key]
    except KeyError:
        item = 0
    dict[key] = item + 1
```

Вот соответствующий C код, во всей красе:

```
int
incr_item(PyObject *dict, PyObject *key)
{
    /* Все объекты инициализированы значением NULL для Py_XDECREF */
    PyObject *item = NULL, *const_one = NULL, *incremented_item = NULL;
    int rv = -1; /* Return value initialized to -1 (failure) */

    item = PyObject_GetItem(dict, key);
    if (item == NULL) {
        /* Обработка только KeyError: */
        if (!PyErr_ExceptionMatches(PyExc_KeyError))
            goto error;

        /* Сбросить ошибку и использовать ноль: */
        PyErr_Clear();
        item = PyLong_FromLong(0L);
        if (item == NULL)
            goto error;
    }
    const_one = PyLong_FromLong(1L);
    if (const_one == NULL)
        goto error;
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
    incremented_item = PyNumber_Add(item, const_one);
    if (incremented_item == NULL)
        goto error;

    if (PyObject_SetItem(dict, key, incremented_item) < 0)
        goto error;
    rv = 0; /* Успех */
    /* Продолжить с кодом очистки */

 error:
    /* Код очистки, общий путь успеха и неудачи */

    /* Использовать Py_XDECREF(), чтобы игнорировать NULL ссылки */
    Py_XDECREF(item);
    Py_XDECREF(const_one);
    Py_XDECREF(incremented_item);

    return rv; /* -1 при ошибке, 0 при успехе */
}
```

Этот пример представляет собой одобренное использование `goto` инструкции в C! Он иллюстрирует использование *`PyErr_ExceptionMatches()`* и *`PyErr_Clear()`* для обработки конкретных исключений, а также использование *`Py_XDECREF()`* для распоряжения принадлежащими им ссылками, которые могут быть `NULL` (обратите внимание на `'X'` в названии; *`Py_DECREF()`* обрушится, когда столкнется с `NULL` ссылкой). Важно, чтобы переменные, использующие для хранения собственных ссылок, были инициализированы `NULL`, чтобы это работало; аналогично, предлагаемое возвращаемое значение инициализируется как `-1` (сбой) и устанавливается как успешный только после успешного выполнения последнего вызова.

1.6 Встраивание Python

Одна важная задача, только встраивания (в отличие от разработчиков расширений) интерпретатора Python, должна переживать за инициализацию и, возможно, доработку интерпретатора Python. Большая часть функциональности интерпретатор можно использовать только после инициализации интерпретатора.

Основной функцией инициализации является *`Py_Initialize()`*. При этом инициализируется таблица загруженных модулей и создаются основные модули `builtins`, `__main__` и `sys`. Она также инициализирует путь поиска модуля (`sys.path`).

`Py_Initialize()` не задает «список аргументов сценария» (`sys.argv`). Если эта переменная необходима для Python кода, который будет выполнен позже, она должна быть явно задана с вызовом `PySys_SetArgvEx(argc, argv, updatepath)` после вызова *`Py_Initialize()`*.

На большинстве систем (в частности, на Unix и Windows, хотя детали немного отличаются), *`Py_Initialize()`* вычисляет путь поиска модуля, основанный на его лучшем предположении для местоположения стандартного исполняемого файла интерпретатора Python, предполагая, что библиотека Python найдена в фиксированном месте относительно исполняемого файла интерпретатора Python. В частности, она ищет каталог с именем `lib/python`*`X.Y`* относительно родительского каталога, где находится в пути поиска команд оболочки исполняемый файл с именем `python` (переменная среды `PATH`).

Например, если Python исполняемый файл находится в `/usr/local/bin/python`, предполагается, что библиотеки находятся в `/usr/local/lib/python`*`X.Y`*. (На самом деле этот конкретный путь - так-

же используется местоположение «отступления», когда никакой исполняемый файл по имени python не найден в PATH.) Пользователь может переопределить это поведение, задав PYTHONHOME переменной среды, или вставив дополнительные каталоги перед стандартным путем, установив PYTHONPATH.

Встраиваемое приложение может управлять поиском, вызывая Py_SetProgramName(file) *перед* вызовом *Py_Initialize()*. Обратите внимание, что PYTHONHOME по-прежнему переопределяет это значение, и PYTHONPATH по-прежнему вставляется перед стандартным путем. Приложение, требующее полного контроля, должно обеспечивать собственную реализацию *Py_GetPath()*, *Py_GetPrefix()*, *Py_GetExecPrefix()* и *Py_GetProgramFullPath()* (все определено в Modules/getpath.c).

Иногда желательно «неинициализировать» Python. Например, приложение может захотеть начаться заново (сделать другой вызов *Py_Initialize()*) или приложение просто сделано с использованием Python и хочет освободить память, аллоцированную Python. Для этого можно вызвать *Py_FinalizeEx()*. Функция *Py_IsInitialized()* возвращает true, если Python находится в данный момент в инициализированном состоянии. Более подробная информация об этих функциях приведена в следующей главе. Заметьте, что *Py_FinalizeEx()* *не* освобождает всей памяти аллоцированную Python интерпретатором, например, память, аллоцированная модулями расширения в настоящее время не может быть освобождена.

1.7 Отладка сборок

Python может быть собран с несколькими макросами для дополнительной проверки модулей интерпретатора и расширения. Эти проверки, как правило, добавляют большой объем накладных расходов во время выполнения, поэтому они не включены по умолчанию.

Полный список различных типов отладочных сборок находится в файле, Misc/SpecialBuilds.txt в Python дистрибутиве исходников. Доступны сборки, поддерживающие трассировку количества ссылок, отладку распределителя памяти или низкоуровневое профилирование основного цикла интерпретатора. В оставшейся части этого раздела будут описаны только самые частоиспользуемые сборки.

Компиляция интерпретатора с заданным макросом Py_DEBUG приводит к тому, что обычно подразумевается под «отладочной сборкой» Python. Py_DEBUG включена в сборке Unix путем добавления --with-pydebug к команде ./configure. Это также подразумевается наличием не-Python-специфичного _DEBUG макроса. Если Py_DEBUG включена в сборке Unix, оптимизация компилятора отключается.

В дополнение к отладке количества ссылок, описанной ниже, выполняются следующие дополнительные проверки:

- Дополнительные проверки добавляются в распределитель объектов.
- Дополнительные проверки добавляются в парсер и компилятор.
- Нисходящие передачи от широких типов к узким типам проверяются на потерю информации.
- В словарь и набор реализаций добавляется ряд утверждений. Кроме того, объект set получает метод test_c_api().
- Проверки работоспособности входных аргументов добавляются к созданию фрейма.
- Хранилище для интов инициализируется с известным недопустимым шаблоном для захвата ссылки на неинициализированные цифры.
- Трассировка низкоуровневой и дополнительной проверки исключений добавляется к виртуальной машине времени выполнения.
- Дополнительные проверки добавляются к реализации арены памяти.
- Дополнительная отладка добавляется в модуль thread.

Могут быть дополнительные проверки, не упомянутые здесь.

Определение `Py_TRACE_REFS` включает трассировку ссылок. При определении циклический список дважды связанных активных объектов сохраняется путем добавления двух дополнительных полей к каждому *`PyObject`*. Также отслеживаются общие аллокации. После выхода печатаются все существующие ссылки. (В интерактивном режиме это происходит после каждого инструкции, выполняемой интерпретатором.) Подразумевается `Py_DEBUG`.

Для получения более подробной информации см. `Misc/SpecialBuilds.txt` в дистрибутиве сорцов Python.