《源码探秘 CPython》86. 内置函数解析

原创 古明地觉 古明地觉的编程教室 2022-05-11 08:30 发表于北京

Python 自带了很多的内置函数，极大地方便了我们的开发，下面就来挑几个内置函数，看看底层是怎么实现的。

内置函数位于 Python/bitlinmodule.c 中。

abs

abs 的功能是取一个整数的绝对值，或者取一个复数的模。

```
static PyObject *
builtin_abs(PyObject *module, PyObject *x)
{
    return PyNumber_Absolute(x);
}
```

该函数调用了 PyNumber_Absolute。

```
//Objects/abstract.c
PyObject *
PyNumber_Absolute(PyObject *o)
{
    PyNumberMethods *m;
    if (o == NULL) {
        return null_error();
    }
    //通过类型对象获取操作簇 PyNumberMethods
    m = o->ob_type->tp_as_number;
    //调用 nb_absolute
    if (m && m->nb_absolute)
        return m->nb_absolute(o);

    return type_error("bad operand type for abs(): '%.200s'", o);
}
```

我们以整型为例，它的 nb_absoulte 指向 long_absolute。

```
//Objects/longobject.c
static PyObject *
long_abs(PyLongObject *v)
{
    if (Py_SIZE(v) < 0)
        //如果 v 小于 0，那么取相反数
        return long_neg(v);
    else
        //否则返回本身
        return long_long((PyObject *)v);
}
```

由于 Python3 的整数是以数组的方式存储的，所以不会直接取相反数，还要做一些额外的处理，但从数学上直接理解为取相反数即可。

all

接收一个可迭代对象，如果里面的元素全部为真，则返回 True；只要有一个不为真，则返回 False。

```
static PyObject *
builtin_all(PyObject *module, PyObject *iterable)
{
    PyObject *it, *item;
    PyObject *(*iternext)(PyObject *);
    int cmp;
    //获取可迭代对象的迭代器
    it = PyObject_GetIter(iterable);
    if (it == NULL)
        return NULL;
    //拿到内部的 __next__ 方法
    iternext = *Py_TYPE(it)->tp_iternext;

    for (;;) {
        //迭代元素
        item = iternext(it);
        //返回 NULL，说明出异常了
        //一种是迭代完毕抛出的 StopIteration
        //另一种是迭代过程中出现的异常
        if (item == NULL)
            break;
        //判断 item 的布尔值是否为真
        //cmp > 0 表示为真
        //cmp == 0表示为假
        //cmp < 0 表示解释器调用出错(极少发生)
        cmp = PyObject_IsTrue(item);
        Py_DECREF(item);
        if (cmp < 0) {
            Py_DECREF(it);
            return NULL;
        }
        //只要有一个元素为假，就返回 False
        if (cmp == 0) {
            Py_DECREF(it);
            Py_RETURN_FALSE;
        }
    }
    Py_DECREF(it);
    //PyErr_Occurred() 为真表示出现异常了
    if (PyErr_Occurred()) {
        //判断异常是不是 StopIteration
        if (PyErr_ExceptionMatches(PyExc_StopIteration))
            //如果是，那么表示迭代正常结束
            //PyErr_Clear() 负责将异常清空
            PyErr_Clear();
        else
            return NULL;
    }
    //走到这，说明所有的元素全部为真
    //返回 True，等价于 return Py_True
    Py_RETURN_TRUE;
}
```

因此 all 就是一层 for 循环，但它是 C 的循环，所以比我们写的 Python 代码快。

any

接收一个可迭代对象，只要里面有一个元素为真，则返回 True；如果全为假，则返回 False。

```c
static PyObject *
builtin_any(PyObject *module, PyObject *iterable)
{
    //源码和 builtin_all 是类似的
    PyObject *it, *item;
    PyObject *(*iternext)(PyObject *);
    int cmp;
    //获取可迭代对象的迭代器
    it = PyObject_GetIter(iterable);
    if (it == NULL)
        return NULL;
    //拿到内部的 __next__ 方法
    iternext = *Py_TYPE(it)->tp_iternext;

    for (;;) {
        //迭代元素
        item = iternext(it);
        if (item == NULL)
            break;
        cmp = PyObject_IsTrue(item);
        Py_DECREF(item);
        if (cmp < 0) {
            Py_DECREF(it);
            return NULL;
        }
        //只要有一个为真，则返回 True
        if (cmp > 0) {
            Py_DECREF(it);
            Py_RETURN_TRUE;
        }
    }
    Py_DECREF(it);
    if (PyErr_Occurred()) {
        if (PyErr_ExceptionMatches(PyExc_StopIteration))
            PyErr_Clear();
        else
            return NULL;
    }
    //全部为假，则返回 False
    Py_RETURN_FALSE;
}
```

判断一个对象是否可调用。

```c
static PyObject *
builtin_callable(PyObject *module, PyObject *obj)
{
    return PyBool_FromLong((long)PyCallable_Check(obj));
}
```

PyBool_FromLong 是将一个整数转成布尔值，所以就看 PyCallable_Check 是返回 0，还是返回非 0。

```c
int
PyCallable_Check(PyObject *x)
{
    if (x == NULL)
        return 0;
    return x->ob_type->tp_call != NULL;
```

```
7 }
```

逻辑非常简单，一个对象是否可调用，就看它的类型对象有没有实现 __call__。

dir

如果不接收任何对象，返回当前的 local 空间；否则返回某个对象的所有属性的名称。

```
1  static PyObject *
2  builtin_dir(PyObject *self, PyObject *args)
3  {
4      PyObject *arg = NULL;
5      //要么不接收参数，要么接收一个参数
6      //如果没有接收参数，那么 arg 就是 NULL，否则就是我们传递的参数
7      if (!PyArg_UnpackTuple(args, "dir", 0, 1, &arg))
8          return NULL;
9      return PyObject_Dir(arg);
10 }
```

该函数调用了 PyObject_Dir。

```
1  //Objects/object.c
2  PyObject *
3  PyObject_Dir(PyObject *obj)
4  {
5      //当 obj 为 NULL，说明我们没有传参，那么返回 local 空间
6      //否则返回对象的所有属性的名称
7      return (obj == NULL) ? _dir_locals() : _dir_object(obj);
8  }
```

先来看看 _dir_locals 函数。

```
1  //Objects/object.c
2  static PyObject *
3  _dir_locals(void)
4  {
5      PyObject *names;
6      PyObject *locals;
7      //获取当前的 local 空间
8      locals = PyEval_GetLocals();
9      if (locals == NULL)
10         return NULL;
11     //拿到所有的 key，注意:PyMapping_Keys 返回的是列表
12     names = PyMapping_Keys(locals);
13     if (!names)
14         return NULL;
15     if (!PyList_Check(names)) {
16         PyErr_Format(PyExc_TypeError,
17             "dir(): expected keys() of locals to be a list, "
18             "not '%.200s'", Py_TYPE(names)->tp_name);
19         Py_DECREF(names);
20         return NULL;
21     }
22     //排序
23     if (PyList_Sort(names)) {
24         Py_DECREF(names);
25         return NULL;
26     }
27     //返回
28     return names;
29 }
```

还是比较简单的，然后是 _dir_object，它的代码比较多，这里就不看了。但是逻辑很简单，就是调用对象的 __dir__ 方法，将得到的列表排序后返回。

id

查看对象的内存地址，我们知道 Python 虽然一切皆对象，但是我们拿到的都是指向对象的指针。比如 id(name) 是查看变量 name 指向对象的地址，说白了不就是 name 本身吗？所以直接将指针转成整数之后返回即可，

```
static PyObject *
builtin_id(PyModuleDef *self, PyObject *v)

{
    //将 v 转成整数，返回即可
    PyObject *id = PyLong_FromVoidPtr(v);

    if (id && PySys_Audit("builtins.id", "O", id) < 0) {
        Py_DECREF(id);
        return NULL;
    }

    return id;
}
```

locals 和 globals

这两者是查看当前的 local 空间和 global 空间，显然直接通过栈帧的 f_locals 和 f_globals 字段即可获取。

```
static PyObject *
builtin_locals_impl(PyObject *module)
{
    PyObject *d;
    //在内部会通过线程状态对象拿到栈帧
    //再通过栈帧的 f_locals 字段拿到 local 空间
    d = PyEval_GetLocals();
    Py_XINCREF(d);
    return d;
}

static PyObject *
builtin_globals_impl(PyObject *module)
{
    PyObject *d;
    //和 PyEval_GetLocals 类似
    d = PyEval_GetGlobals();
    Py_XINCREF(d);
    return d;
}
```

hash

获取对象的哈希值。

```
static PyObject *
builtin_hash(PyObject *module, PyObject *obj)
{
```

```
    Py_hash_t x;
    //在内部会调用 obj -> ob_type -> tp_hash(obj)
    x = PyObject_Hash(obj);
    if (x == -1)
        return NULL;
    return PyLong_FromSsize_t(x);
}
```

接收一个可迭代对象，计算它们的和。但是这里面有一个需要注意的地方。

```
print(sum([1, 2, 3]))  # 6

try:
    print(sum(["1", "2", "3"]))
except TypeError as e:
    print(e)  # unsupported operand type(s) for +: 'int' and 'str'
```

咦，字符串明明也支持加法呀，为啥不行呢？其实 sum 还可以接收第二个参数，我们不传的话就是 0。

也就是说 **sum([1, 2, 3])** 其实是 **0 + 1 + 2 + 3**；那么同理，**sum(["a", "b", "c"])** 其实是 **0 + "a" + "b" + "c"**；所以上面的报错信息是不支持类型为 int 和 str 的实例进行相加。

```
try:
    print(sum(["1", "2", "3"], ""))
except TypeError as e:
    print(e)  # sum() can't sum strings [use ''.join(seq) instead]

# 我们看到还是报错了，只能说明理论上是可以的
# 但 Python 建议我们使用 join

# 我们用列表举例吧
try:
    print(sum([[1], [2], [3]]))
except TypeError as e:
    print(e)  # unsupported operand type(s) for +: 'int' and 'list'

# 告诉我们 int 的实例和 list 的实例不可以相加
# 将第二个参数换成空列表
print(sum([[1], [2], [3]], []))  # [1, 2, 3]

# 如果不是空列表呢？
print(
    sum([[1], [2], [3]], ["古明地觉"])
)  # ['古明地觉', 1, 2, 3]
```

所以 sum 是将第二个参数和第一个参数（可迭代对象）里面的元素依次相加，然后看一下底层实现。

```
static PyObject *
builtin_sum_impl(PyObject *module, PyObject *iterable, PyObject *start)
{
    //result 就是返回值，初始等于第二个参数
    //如果可迭代对象为空，那么返回的就是第二个参数
    //比如 sum([], 123) 得到的就是 123
    PyObject *result = start;
    PyObject *temp, *item, *iter;
    //获取可迭代对象的类型对象
    iter = PyObject_GetIter(iterable);
```

```c
    if (iter == NULL)
        return NULL;

    //如果 result 为 NULL，说明我们没有传递第二个参数
    if (result == NULL) {
        //那么 result 赋值为 0
        result = PyLong_FromLong(0);
        if (result == NULL) {
            Py_DECREF(iter);
            return NULL;
        }
    } else {
        //否则的话，检测是不是 str、bytes、bytearray 类型
        //如果是的话，依旧报错，并提示使用 join 方法
        if (PyUnicode_Check(result)) {
            PyErr_SetString(PyExc_TypeError,
                "sum() can't sum strings [use ''.join(seq) instead]");
            Py_DECREF(iter);
            return NULL;
        }
        if (PyBytes_Check(result)) {
            PyErr_SetString(PyExc_TypeError,
                "sum() can't sum bytes [use b''.join(seq) instead]");
            Py_DECREF(iter);
            return NULL;
        }
        if (PyByteArray_Check(result)) {
            PyErr_SetString(PyExc_TypeError,
                "sum() can't sum bytearray [use b''.join(seq) instead]");
            Py_DECREF(iter);
            return NULL;
        }
        Py_INCREF(result);
    }

#ifndef SLOW_SUM
    //这里是快分支
    //假设所有元素都是整数
    if (PyLong_CheckExact(result)) {
        //将所有整数都迭代出来，依次相加
        //...
    }

    if (PyFloat_CheckExact(result)) {
        //将所有浮点数都迭代出来，依次相加
        //...
    }
#endif

    //如果不全是整数或浮点数，执行通用逻辑
    for(;;) {
        //迭代元素
        item = PyIter_Next(iter);
        if (item == NULL) {
            /* error, or end-of-sequence */
            if (PyErr_Occurred()) {
                Py_DECREF(result);
                result = NULL;
            }
            break;
        }
        //和 result 依次相加
        temp = PyNumber_Add(result, item);
        Py_DECREF(result);
```

```
        Py_DECREF(item);
        result = temp;
        if (result == NULL)
            break;
    }
    Py_DECREF(iter);
    //返回
    return result;
}
```

一个小小的 sum，代码量还真不少呢，我们还省略了一部分。

getattr、setattr、delattr

这几个应该已经很熟悉了，先来看看 getattr，它是获取对象的某个属性，并且还可以指定默认值。

```
static PyObject *
builtin_getattr(PyObject *self, PyObject *const *args, Py_ssize_t nargs)
{
    PyObject *v, *name, *result;
    //参数个数必须是 2 或 3
    //对象、属性名、可选的默认值
    if (!_PyArg_CheckPositional("getattr", nargs, 2, 3))
        return NULL;
    //获取对象和属性名
    v = args[0];
    name = args[1];
    //name必须是字符串
    if (!PyUnicode_Check(name)) {
        PyErr_SetString(PyExc_TypeError,
                        "getattr(): attribute name must be string");
        return NULL;
    }
    //调用对象的 __getattr__，找不到返回默认值
    if (nargs > 2) {
        if (_PyObject_LookupAttr(v, name, &result) == 0) {
            PyObject *dflt = args[2];
            Py_INCREF(dflt);
            return dflt;
        }
    }
    else {
        result = PyObject_GetAttr(v, name);
    }
    return result;
}
```

同理 setattr 是调用对象的 __setattr__，delattr 是调用对象的 __delattr__。

关于内置函数就介绍这么多，当然实际肯定不止我们说的这些，有兴趣的话可以进入源码中查看。

收录于合集 #CPython 97

‹ 上一篇
《源码探秘 CPython》87. 解密 map、filter、zip 底层实现，对比列表解析式

下一篇 ›
《源码探秘 CPython》85. Python线程的创建、销毁、调度，以及 GIL 的实现原理

喜欢此内容的人还喜欢

真香！超全，Python 中常见的配置文件写法

Python丹卿

客户给100块要做个百度，我用10行Python代码搞定

Python丹卿

百看不如一练， 247个 Python 入门到进阶实战案例！

编程小小