《源码探秘 CPython》10. 浮点数的行为

原创 古明地觉 古明地觉的编程教室 2022-01-13 09:30

楔子

PyFloat_Type中定义了很多的函数指针，比如：type_repr、tp_str、tp_hash等等，这些函数指针将一起决定浮点数的行为，例如：tp_hash决定了浮点数的哈希值计算：

```
>>> e = 2.71
>>> hash(e)
1637148536541722626
>>>
```

tp_hash指向的是float_hash，还是那句话，Python底层的函数命名以及API都是很有规律的，相信你能慢慢发现。

```
static Py_hash_t
float_hash(PyFloatObject *v)
{
    //我们看到调用了_Py_HashDouble
    //计算的就是ob_fval成员的哈希值
    return _Py_HashDouble(v->ob_fval);
}
```

浮点数的运算

由于加减乘除等数值操作很常见，所以Python将其抽象成数值操作簇**PyNumberMethods**，并让内部成员tp_as_number指向。数值操作簇**PyNumberMethods** 在头文件**Include/object.h**中定义：

```
typedef struct {
    binaryfunc nb_add;
    binaryfunc nb_subtract;
    binaryfunc nb_multiply;
    binaryfunc nb_remainder;
    binaryfunc nb_divmod;
    ternaryfunc nb_power;
    unaryfunc nb_negative;
    // ...

    binaryfunc nb_inplace_add;
    binaryfunc nb_inplace_subtract;
    binaryfunc nb_inplace_multiply;
    binaryfunc nb_inplace_remainder;
    ternaryfunc nb_inplace_power;
    //...
} PyNumberMethods;
```

PyNumberMethods定义了各种数学算子的处理函数，数值计算最终由这些函数执行，当然这些函数就是魔法方法的底层实现。

处理函数根据参数个数可以分为：**一元函数(unaryfunc)** 、 **二元函数(binaryfunc)** 和**三元函数(ternaryfunc)**。

对于PyFloat_Type而言，在初始化的时候给成员tp_as_number赋的值为**&float_as_number**，我们来看一看。

```
static PyNumberMethods float_as_number = {
    float_add,          /* nb_add */
    float_sub,          /* nb_subtract */
    float_mul,          /* nb_multiply */
    float_rem,          /* nb_remainder */
    float_divmod,       /* nb_divmod */
    float_pow,          /* nb_power */
    (unaryfunc)float_neg, /* nb_negative */
    // ...

    0,                  /* nb_inplace_add */
    0,                  /* nb_inplace_subtract */
    0,                  /* nb_inplace_multiply */
    0,                  /* nb_inplace_remainder */
    0,                  /* nb_inplace_power */
    // ...
};
```

以加法为例，最终执行**float_add**，显然它是一个二元函数，我们看一下底层实现。

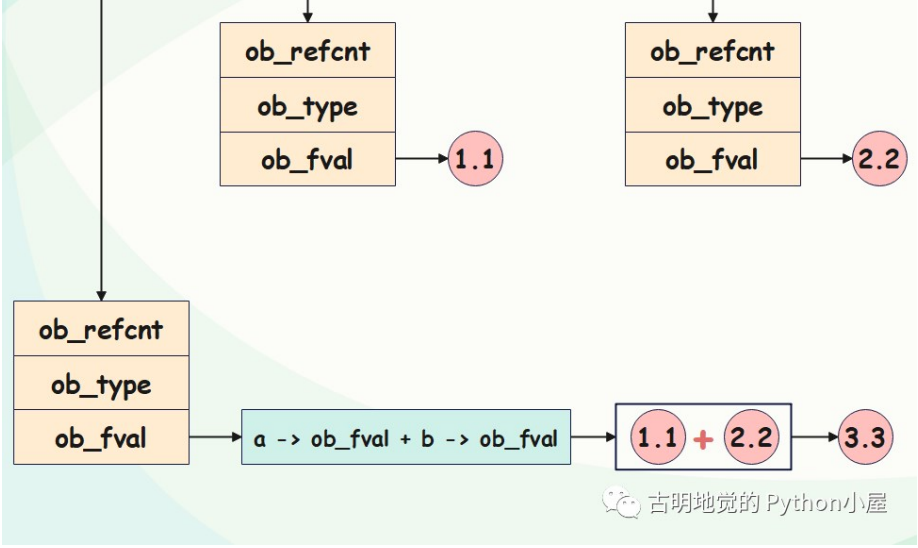
```
static PyObject *
float_add(PyObject *v, PyObject *w)
{
    //显然两个Python对象相加
    //一定是先将其转成C的对象，然后再相加
    //加完之后再根据结果创建新的Python对象
    //所以声明了两个double
    double a,b;
    //CONVERT_TO_DOUBLE是一个宏，从名字上也能看出来它的作用
    //将PyFloatObject里面的ob_fval抽出来，赋值给double变量
    //这个宏有兴趣可以去源码中看一下，也在当前文件中
    CONVERT_TO_DOUBLE(v, a);  // 将ob_fval赋值给a
    CONVERT_TO_DOUBLE(w, b);  // 将ob_fval赋值给b

    //PyFPE_START_PROTECT和下面的PyFPE_END_PROTECT也都是宏，
    //作用我们一会儿说
    PyFPE_START_PROTECT("add", return 0)
    //将a和b相加赋值给a
    a = a + b;
    PyFPE_END_PROTECT(a)
    //根据相加后的结果创建新的PyFloatObject对象
    //当然返回的是泛型指针PyObject *
    return PyFloat_FromDouble(a);
}
```

因此以上就是浮点数的运算，核心就是：

- 1. 定义两个double变量：a、b
- 2. 将用来相加的两个浮点数维护的值（ob_fval）抽出来赋值给a和b
- 3. 让a和b相加，将相加结果传入PyFloat_FromDouble中创建新的PyFloatObject，然后返回其PyObject *

以上便是浮点数的加法运算，所谓的浮点数在底层就是一个PyFloatObject结构体实例。而两个结构体实例无法相加，所以必须先将结构体中维护的值抽出来，对于浮点数而言就是ob_fval，然后转成C的double再进行相加。最后根据相加的结果创建新的结构体实例，于是新的Python对象便诞生了。

假设 a, b = 1.1, 2.2，那么 c=a+b 的流程就如下所示：



但如果是C中的两个浮点数相加，那么a + b在编译之后就是一条简单的机器指令，然而Python则需要额外做很多其它工作。

并且在介绍整数的时候，你会发现Python的整数相加会更麻烦，但对于C而言同样是一条简单的机器码就可以搞定。当然啦，因为Python3的整数是不会溢出的，所以需要额外的一些处理，等介绍整数的时候再说吧。

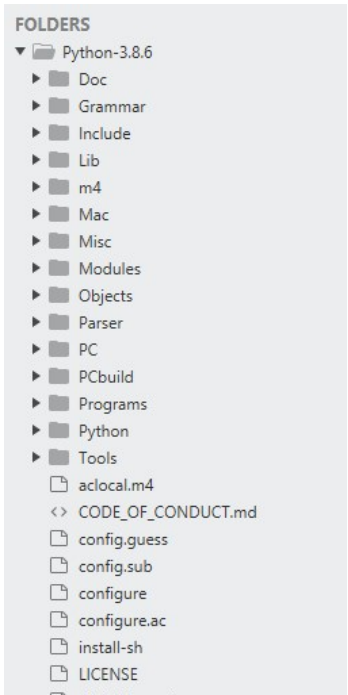
所以这里我们也知道Python为什么会比C慢几十倍了，从一个简单的加法上面就可以看出来。

> 最后我们再说一下float_add里面的PyFPE_START_PROTECT和PyFPE_END_PROTECT这两个宏，其实它们对于我们了解浮点数在底层的计算没有什么意义。首先浮点数计算一般都遵循IEEE-754标准，如果计算时出现了错误，那么需要将IEEE-754异常转换成Python中的异常，而这两个宏就是用来干这件事情的。
>
> 所以我们不需要管它，这两个宏定义在Include/pyfpe.h中，并且已经在Python3.9的时候被删除掉了。

以上是浮点数的加法操作，至于减法、乘法、除法等操作也是类似的，就是根据Python的浮点数创建C的浮点数，运算完之后根据结果再创建Python的浮点数。

CPython源码结构

最后我们说一下解释器源代码的结构吧，因为我们每一次介绍函数的时候，都会说该函数定义在哪个文件里。所以突然想起来，介绍一下源代码的组织结构也是有必要的。



我们从官网上将源代码下载下来之后，大概长这样，里面有几个目录是我们需要关注的。

- Include：该目录包含了CPython所提供的所有头文件，主要包含了一些实例对象在底层的定义，比如listobject.h、dictobject.h等等。如果用户需要自己使用C或者C++来编写自定义模块来扩展Python，那么也需要用到这里的头文件。
- Lib：这个无需多说，该目录包含了Python自带的所有标准库，Lib中的库基本上都是使用纯Python编写的。
- Modules：该目录中包含了所有用C语言编写的模块，比如**_random**、**_io**等，而且**gc**也在里面。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目录下的。
- Parser：该目录中包含了解释器中的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词法和语法分析器，与YACC非常类似。
- Objects：该目录包含了所有Python的内置类型对象的实现，以及其实例对象相关操作的实现。比如浮点数相关操作就位于文件**floatobject.c**中、列表相关操作就位于文件**listobject.c**中，文件名也很有规律。同时，该目录还包含了Python在运行时需要的所有内部对象的实现，因为有很多对象，比如**<class 'function'>**没有暴露给Python，但是在底层它们肯定是实现了的。
- Python：虚拟机的实现相关，是Python运行的核心所在。

小结

到此浮点数我们就介绍完了，之所以先介绍浮点数，是因为浮点数最简单。至于整数，其实并没有那么简单，因为它的值底层是通过数组存储的，而浮点数底层是用一个double存储对应的值，会更简单一些，所以我们就先拿浮点数**开刀了**。

首先我们介绍了浮点数的创建和销毁，创建有两种方式，使用Python/C API更快一些。

销毁的时候则调用类型对象内部的tp_dealloc，浮点数的话就是float_dealloc。当然为了保证效率，避免内存的创建和回收，解释器为浮点数引入了缓存池机制，我们也分析了背后的原理。

最后浮点数还支持相关的数值型操作，PyFloat_Type中的**tp_as_number**指向了**PyNumberMethods**结构体实例**float_as_number**，里面有大量的函数指针，每个指针指向了具体的函数，专门用于浮点数的运算。

当然整型也有，只不过指针指向的函数是用于整数运算的。比如相加：对于浮点数来说，PyNumberMethods结构体成员nb_add指向了函数float_add；对于整数来说，nb_add则是指向了long_add。

然后我们也以相加为例，看了float_add函数的实现，核心就是将Python对象的值抽出来，转成C的类型，然后运算，最后再根据运算的结果，创建Python的对象、并返回泛型指针。

当然除了加法，它的减法、乘法、除法都是类似的，有兴趣可以**杀入**floatobject.c中，大肆探索一番。

收录于合集 #CPython 97

‹ 上一篇

下一篇 ›

《源码探秘 CPython》11. 整数是怎么设计的，为什么它不会溢出?

《源码探秘 CPython》9. 浮点数的缓存池机制

喜欢此内容的人还喜欢

一文剖析MySQL主从复制异常错误代码13114

TtrOpsStack

力扣 428. 序列化和反序列化 N 叉树 DFS

钰娘娘知识汇总

MySQL · 参数故事 · timed_mutexes

夜雨成诗