3. 定义扩展类型：已分类主题

本章节目标是提供一个各种你可以实现的类型方法及其功能的简短介绍。

这是 C 类型 PyTypeObject 的定义，省略了只用于 调试构建 的字段：

```
typedef struct _typeobject {
    PyObject_VAR_HEAD
    const char *tp_name; /* 用于打印，格式为 "<module>.<name>" */
    Py_ssize_t tp_basicsize, tp_itemsize; /* 用于分配 */

    /* 用于实现标准操作的方法 */

    destructor tp_dealloc;
    Py_ssize_t tp_vectorcall_offset;
    getattrfunc tp_getattr;
    setattrfunc tp_setattr;
    PyAsyncMethods *tp_as_async; /* 原名为 tp_compare (Python 2)
                                    或 tp_reserved (Python 3) */
    reprfunc tp_repr;

    /* 用于标准类的方法集 */

    PyNumberMethods *tp_as_number;
    PySequenceMethods *tp_as_sequence;
    PyMappingMethods *tp_as_mapping;

    /* 更多标准操作 (这些用于二进制兼容) */

    hashfunc tp_hash;
    ternaryfunc tp_call;
    reprfunc tp_str;
    getattrofunc tp_getattro;
    setattrofunc tp_setattro;

    /* 用于以输入/输出缓冲区方式访问对象的函数 */
    PyBufferProcs *tp_as_buffer;

    /* 用于定义可选/扩展特性是否存在的旗标 */
    unsigned long tp_flags;

    const char *tp_doc; /* 文档字符串 */

    /* 在 2.0 发布版中分配的含义 */
    /* 为所有可访问的对象调用函数 */
    traverseproc tp_traverse;

    /* 删除对所包含对象的引用 */
    inquiry tp_clear;

    /* 在 2.1 发布版中分配的含义 */
    /* 富比较操作 */
    richcmpfunc tp_richcompare;
```

```
    /* 启用弱引用 */
    Py_ssize_t tp_weaklistoffset;

    /* 迭代器 */
    getiterfunc tp_iter;
    iternextfunc tp_iternext;

    /* 属性描述器和子类化内容 */
    PyMethodDef *tp_methods;
    PyMemberDef *tp_members;
    PyGetSetDef *tp_getset;
    // 堆类型的强引用，静态类型的借入引用
    PyTypeObject *tp_base;
    PyObject *tp_dict;
    descrgetfunc tp_descr_get;
    descrsetfunc tp_descr_set;
    Py_ssize_t tp_dictoffset;
    initproc tp_init;
    allocfunc tp_alloc;
    newfunc tp_new;
    freefunc tp_free; /* 层级的释放内存例程 */
    inquiry tp_is_gc; /* 用于 PyObject_IS_GC */
    PyObject *tp_bases;
    PyObject *tp_mro; /* 方法解析顺序 */
    PyObject *tp_cache; /* 不再被使用 */
    void *tp_subclasses;  /* 对于静态内置类型这将是一个索引 */
    PyObject *tp_weaklist; /* 不被用于静态内置类型 */
    destructor tp_del;

    /* 类型属性缓存版本标签。 在 2.6 版中添加。
     * 如果为零，则缓存无效并且必须被初始化。
     */
    unsigned int tp_version_tag;

    destructor tp_finalize;
    vectorcallfunc tp_vectorcall;

    /* 类型监视器针对此类型的位设置 */
    unsigned char tp_watched;

    /* 使用的 tp_version_tag 值数量。
     * 如果针对此类型的属性缓存被禁用则设为 _Py_ATTR_CACHE_UNUSED
     * （例如由于自定义的 MRO 条目而被禁用）。
     * 在其他情况下，将被限制为 MAX_VERSIONS_PER_CLASS (在其他地方定义)。
     */
    uint16_t tp_versions_used;
} PyTypeObject;
```

这里有 *很多* 方法。但是不要太担心，如果你要定义一个类型，通常只需要实现少量的方法。

正如你猜到的一样，我们正要一步一步详细介绍各种处理程序。因为有大量的历史包袱影响字段的排序，所以我们不会根据它们在结构体里定义的顺序讲解。通常非常容易找到一个包含你需要的字段的例子，然后改变值去适应你新的类型。

```
const char *tp_name; /* 用于打印 */
```

类型的名字 - 上一章提到过的，会出现在很多地方，几乎全部都是为了诊断目的。尝试选择一个好名字，对于诊断很有帮助。

```
Py_ssize_t tp_basicsize, tp_itemsize; /* 用于分配 */
```

这些字段告诉运行时在创造这个类型的新对象时需要分配多少内存。Python为了可变长度的结构（想下：字符串，元组）有些内置支持，这是 `tp_itemsize` 字段存在的原由。这部分稍后解释。

```
const char *tp_doc;
```

这里你可以放置一段字符串（或者它的地址），当你想在Python脚本引用 `obj.__doc__` 时返回这段文档字符串。

现在我们来看一下基本类型方法 - 大多数扩展类型将实现的方法。

3.1. 终结和内存释放

```
destructor tp_dealloc;
```

当您的类型实例的引用计数减少为零并且Python解释器想要回收它时，将调用此函数。如果你的类型有内存可供释放或执行其他清理，你可以把它放在这里。 对象本身也需要在这里释放。 以下是此函数的示例：

```
static void
newdatatype_dealloc(PyObject *op)
{
    newdatatypeobject *self = (newdatatypeobject *) op;
    free(self->obj_UnderlyingDatatypePtr);
    Py_TYPE(self)->tp_free(self);
}
```

如果你的类型支持垃圾回收，则析构器应当在清理任何成员字段之前调用 `PyObject_GC_UnTrack()`：

```
static void
newdatatype_dealloc(PyObject *op)
{
    newdatatypeobject *self = (newdatatypeobject *) op;
    PyObject_GC_UnTrack(op);
    Py_CLEAR(self->other_obj);
    ...
    Py_TYPE(self)->tp_free(self);
}
```

一个重要的释放器函数实现要求是把所有未决异常放着不动。这很重要是因为释放器会被解释器频繁的调用，当栈异常退出时(而非正常返回)，不会有任何办法保护释放器看到一个异常尚未被设置。此事释放器的任何行为都会导致额外增加的Python代码来检查异常是否被设置。这可能导致解释器的误导性错误。正确的保护方法是，在任何不安全的操作前，保存未决异常，然后在其完成后恢复。者可以通过 `PyErr_Fetch()` 和 `PyErr_Restore()` 函数来实现:

```
static void
my_dealloc(PyObject *obj)
{
    MyObject *self = (MyObject *) obj;
    PyObject *cbresult;

    if (self->my_callback != NULL) {
        PyObject *err_type, *err_value, *err_traceback;

        /* 这里保存当前异常状态 */
        PyErr_Fetch(&err_type, &err_value, &err_traceback);

        cbresult = PyObject_CallNoArgs(self->my_callback);
        if (cbresult == NULL) {
           PyErr_WriteUnraisable(self->my_callback);
        }
        else {
            Py_DECREF(cbresult);
        }

        /* 这里恢复被保存的异常状态 */
        PyErr_Restore(err_type, err_value, err_traceback);

        Py_DECREF(self->my_callback);
    }
    Py_TYPE(self)->tp_free(self);
}
```

备注： 你能在释放器函数中安全执行的操作是有限的。 首先，如果你的类型支持垃圾回收 (使用 `tp_traverse` 和/或 `tp_clear`)，对象的部分成员可以在调用 `tp_dealloc` 时被清空或终结。 其次，在 `tp_dealloc` 中，你的对象将处于不稳定状态：它的引用计数等于零。 任何对非琐碎对象或 API 的调用 (如上面的示例所做的) 最终都可能会再次调用 `tp_dealloc`，导致双重释放并发生崩溃。

从 Python 3.4 开始，推荐不要在 `tp_dealloc` 放复杂的终结代码，而是使用新的 `tp_finalize` 类型方法。

参见： PEP 442 解释了新的终结方案。

3.2. 对象展示

在 Python 中，有两种方式可以生成对象的文本表示: `repr()` 函数和 `str()` 函数。 (`print()` 函数会直接调用 `str()`。) 这些处理程序都是可选的。

```
reprfunc tp_repr;
reprfunc tp_str;
```

`tp_repr` 处理程序应该返回一个字符串对象，其中包含调用它的实例的表示形式。 下面是一个简单的例子:

```
static PyObject *
newdatatype_repr(PyObject *op)
```

```
{
    newdatatypeobject *self = (newdatatypeobject *) op;
    return PyUnicode_FromFormat("Repr-ified_newdatatype{{size:%d}}",
                                self->obj_UnderlyingDatatypePtr->size);
}
```

如果没有指定 tp_repr 处理器，解释器将提供一个使用类型的 tp_name 的表示形式以及对象的唯一标识值。

tp_str 处理器对于 str() 就如上述的 tp_repr 处理器对于 repr() 一样；也就是说，它会在当 Python 代码在你的对象的某个实例上调用 str() 时被调用。 它的实现与 tp_repr 函数非常相似，但其结果字符串是供人类查看的。 如果未指定 tp_str，则会使用 tp_repr 处理器来代替。

下面是一个简单的例子:

```
static PyObject *
newdatatype_str(PyObject *op)
{
    newdatatypeobject *self = (newdatatypeobject *) op;
    return PyUnicode_FromFormat("Stringified_newdatatype{{size:%d}}",
                                self->obj_UnderlyingDatatypePtr->size);
}
```

3.3. 属性管理

对于每个可支持属性操作的对象，相应的类型必须提供用于控制属性获取方式的函数。 需要有一个能够检索属性的函数（如果定义了任何属性）还要有另一个函数负责设置属性（如果允许设置属性）。 移除属性是一种特殊情况，在此情况下要传给处理器的新值为 NULL。

Python 支持两对属性处理器；一个支持属性操作的类型只需要实现其中一对的函数。 两者的差别在于一对接受 char* 作为属性名称，而另一对则接受 PyObject*。 每种类型都可以选择使用对于实现的便利性来说更有意义的那一对。

```
getattrfunc  tp_getattr;        /* char * 版本 */
setattrfunc  tp_setattr;
/* ... */
getattrofunc tp_getattro;       /* PyObject * 版本 */
setattrofunc tp_setattro;
```

如果访问一个对象的属性总是为简单操作（这将在下文进行解释），则有一些泛用实现可被用来提供 PyObject* 版本的属性管理函数。 从 Python 2.2 开始对于类型专属的属性处理器的实际需要几乎已完全消失，尽管还存在着许多尚未理新为使用某种新的可选泛用机制的例子。

3.3.1. 泛型属性管理

大多数扩展类型只使用 **简单** 属性，那么，是什么让属性变得"简单"呢？只需要满足下面几个条件:

1. 当调用 PyType_Ready() 时，必须知道属性的名称。
2. 不需要特殊的处理来记录属性是否被查找或设置，也不需要根据值采取操作。

请注意，此列表不对属性的值、值的计算时间或相关数据的存储方式施加任何限制。

当 PyType_Ready() 被调用时，它会使用由类型对象所引用的三个表来创建要放置到类型对象的字典中的 descriptor。 每个描述器控制对实例对象的一个属性的访问。 每个表都是可选的；如果三个表全都为 NULL，则该类型的实例将只有从它们的基础类型继承来的属性，并且还应当让 tp_getattro 和 tp_setattro 字段保持为 NULL，以允许由基础类型处理这些属性。

表被声明为object::类型的三个字段:

```
struct PyMethodDef *tp_methods;
struct PyMemberDef *tp_members;
struct PyGetSetDef *tp_getset;
```

如果 tp_methods 不为 NULL，则它必须指向一个由 PyMethodDef 结构体组成的数组。 表中的每个条目都是该结构体的一个实例:

```
typedef struct PyMethodDef {
    const char  *ml_name;       /* 方法名称 */
    PyCFunction  ml_meth;       /* 实现函数 */
    int          ml_flags;      /* 旗标 */
    const char  *ml_doc;        /* 文档字符串 */
} PyMethodDef;
```

应当为该类型所提供的每个方法都应定义一个条目；从基类型继承来的方法无需定义条目。 还需要在末尾加一个额外的条目；它是一个标记数组结束的哨兵条目。 该哨兵条目的 ml_name 字段必须为 NULL。

第二个表被用来定义要直接映射到实例中的数据的属性。 各种原始 C 类型均受到支持，并且访问方式可以为只读或读写。 表中的结构体被定义为:

```
typedef struct PyMemberDef {
    const char *name;
    int         type;
    int         offset;
    int         flags;
    const char *doc;
} PyMemberDef;
```

对于表中的每个条目，都将构建一个 descriptor 并添加到类型中使其能够从实例结构体中提取值。 type 字段应包含一个类型代码如 Py_T_INT 或 Py_T_DOUBLE；该值将用于确定如何将 Python 值转换为 C 值或反之。 flags 字段用于保存控制属性要如何被访问的旗标：你可以将其设为 Py_READONLY 以防止 Python 代码设置它。

使用 tp_members 表来构建用于运行时的描述器还有一个有趣的优点是任何以这种方式定义的属性都可以简单地通过在表中提供文本来设置一个相关联的文档字符串。 一个应用程序可以使用自省 API 从类对象获取描述器，并使用其 __doc__ 属性来获取文档字符串。

与 tp_methods 表一样，需要有一个值为 NULL 的 ml_name 哨兵条目。

3.3.2. 类型专属的属性管理

为了简单起见，这里只演示 char* 版本； name 形参的类型是 char* 和 PyObject* 风格接口之间的唯一区别。 这个示例实际上做了与上面的泛用示例相同的事情，但没有使用在 Python 2.2 中增加的泛用支持。 它解释了处理器函数是如何被调用的，因此如果你确实需要扩展它们的功能，你就会明白有什么是需要做的。

tp_getattr 处理器会在对象需要进行属性查找时被调用。 它被调用的场合与一个类的 __getattr__() 方法要被调用的场合相同。

例如：

```
static PyObject *
newdatatype_getattr(PyObject *op, char *name)
{
    newdatatypeobject *self = (newdatatypeobject *) op;
    if (strcmp(name, "data") == 0) {
        return PyLong_FromLong(self->data);
    }

    PyErr_Format(PyExc_AttributeError,
                 "'%.100s' object has no attribute '%.400s'",
                 Py_TYPE(self)->tp_name, name);
    return NULL;
}
```

当调用类实例的 __setattr__() 或 __delattr__() 方法时会调用 tp_setattr 处理器。 当需要删除一个属性时，第三个形参将为 NULL。 下面是一个简单地引发异常的例子；如果这确实是你想要的，则 tp_setattr 处理器应当被设为 NULL。

```
static int
newdatatype_setattr(PyObject *op, char *name, PyObject *v)
{
    PyErr_Format(PyExc_RuntimeError, "Read-only attribute: %s", name);
    return -1;
}
```

3.4. 对象比较

```
richcmpfunc tp_richcompare;
```

tp_richcompare 处理器会在需要进行比较时被调用。 它类似于 富比较方法，例如 __lt__()，并会被 PyObject_RichCompare() 和 PyObject_RichCompareBool() 调用。

此函数被调用时将传入两个 Python 对象和运算符作为参数，其中运算符为 Py_EQ, Py_NE, Py_LE, Py_GE, Py_LT 或 Py_GT 之一。 它应当使用指定的运算符来比较两个对象并在比较操作成功时返回 Py_True 或 Py_False，如果比较操作未被实现并应尝试其他对象比较方法时则返回 Py_NotImplemented，或者如果设置了异常则返回 NULL。

下面是一个示例实现，该数据类型如果内部指针的大小相等就认为是相等的:

```
static PyObject *
newdatatype_richcmp(PyObject *lhs, PyObject *rhs, int op)
```

```
{
    newdatatypeobject *obj1 = (newdatatypeobject *) lhs;
    newdatatypeobject *obj2 = (newdatatypeobject *) rhs;
    PyObject *result;
    int c, size1, size2;

    /* 省略了确保两个参数均为
       newdatatype 类型的代码 */

    size1 = obj1->obj_UnderlyingDatatypePtr->size;
    size2 = obj2->obj_UnderlyingDatatypePtr->size;

    switch (op) {
    case Py_LT: c = size1 <  size2; break;
    case Py_LE: c = size1 <= size2; break;
    case Py_EQ: c = size1 == size2; break;
    case Py_NE: c = size1 != size2; break;
    case Py_GT: c = size1 >  size2; break;
    case Py_GE: c = size1 >= size2; break;
    }
    result = c ? Py_True : Py_False;
    return Py_NewRef(result);
 }
```

3.5. 抽象协议支持

Python 支持多种 *抽象* '协议'；被提供来使用这些接口的专门接口说明请在 抽象对象层 中查看。

这些抽象接口很多都是在 Python 实现开发的早期被定义的。 特别地，数字、映射和序列协议从一开始就已经是 Python 的组成部分。 其他协议则是后来添加的。 对于依赖某些来自类型实现的处理器例程的协议来说，较旧的协议被定义为类型对象所引用的处理器的可选块。 对于较新的协议来说在主类型对象中还有额外的槽位，并带有一个预设旗标位来指明存在该槽位并应当由解释器来检查。 （此旗标位并不会指明槽位值非 `NULL` 的情况，可以设置该旗标来指明一个槽位的存在，但此本位仍可能保持未填充的状态。）

```
PyNumberMethods   *tp_as_number;
PySequenceMethods *tp_as_sequence;
PyMappingMethods  *tp_as_mapping;
```

如果你希望你的对象的行为类似一个数字、序列或映射对象，那么你就要分别放置一个实现了 C 类型 `PyNumberMethods`, `PySequenceMethods` 或 `PyMappingMethods`, 的结构体的地址。 你要负责将适当的值填入这些结构体。 你可以在 Python 源代码发布版的 `Objects` 目录中找到这些对象各自的用法示例。

```
hashfunc tp_hash;
```

如果你选择提供此函数，则它应当为你的数据类型的实例返回一个哈希数值。 下面是一个简单的示例:

```
static Py_hash_t
newdatatype_hash(PyObject *op)
{
    newdatatypeobject *self = (newdatatypeobject *) op;
```

```
    Py_hash_t result;
    result = self->some_size + 32767 * self->some_number;
    if (result == -1) {
        result = -2;
    }
    return result;
}
```

Py_hash_t 是一个在宽度取决于具体平台的有符号整数类型。 从 tp_hash 返回 -1 表示发生了错误，这就是为什么你应当注意避免在哈希运算成功时返回它，如上面所演示的。

```
ternaryfunc tp_call;
```

此函数会在“调用”你的数据类型实例时被调用，举例来说，如果 obj1 是你的数据类型的实例而 Python 脚本包含了 obj1('hello')，则将唤起 tp_call 处理器。

此函数接受三个参数:

1. *self* 是作为调用目标的数据类型实例。 如果调用是 obj1('hello')，则 *self* 为 obj1。
2. *args* 是包含调用参数的元组。 你可以使用 PyArg_ParseTuple() 来提取参数。
3. *kwds* 是由传入的关键字参数组成的字典。 如果它不为 NULL 且你支持关键字参数，则可使用 PyArg_ParseTupleAndKeywords() 来提取参数。 如果你不想支持关键字参数而它为非 NULL 值，则会引发 TypeError 并附带一个提示不支持关键字参数的消息。

下面是一个演示性的 tp_call 实现:

```
static PyObject *
newdatatype_call(PyObject *op, PyObject *args, PyObject *kwds)
{
    newdatatypeobject *self = (newdatatypeobject *) op;
    PyObject *result;
    const char *arg1;
    const char *arg2;
    const char *arg3;

    if (!PyArg_ParseTuple(args, "sss:call", &arg1, &arg2, &arg3)) {
        return NULL;
    }
    result = PyUnicode_FromFormat(
        "Returning -- value: [%d] arg1: [%s] arg2: [%s] arg3: [%s]\n",
        self->obj_UnderlyingDatatypePtr->size,
        arg1, arg2, arg3);
    return result;
}
```

```
/* Iterators */
getiterfunc tp_iter;
iternextfunc tp_iternext;
```

这些函数提供了对迭代器协议的支持。 这两个处理器都只接受一个形参，即它们被调用时所使用的实例，并返回一个新的引用。 当发生错误时，它们应设置一个异常并返回 NULL。 tp_iter 对应于 Python __iter__() 方法，而 tp_iternext 对应于 Python __next__() 方法。

任何 iterable 对象都必须实现 tp_iter 处理器，该处理器必须返回一个 iterator 对象。 下面是与 Python 类所应用的同一个指导原则:

- 对于可以支持多个独立迭代器的多项集（如列表和元组），则应当在每次调用 tp_iter 时创建并返回一个新的迭代器。
- 只能被迭代一次的对象（通常是由于迭代操作的附带影响，例如文件对象）可以通过返回一个指向自身的新引用来实现 tp_iter -- 并且为此还应当实现 tp_iternext 处理器。

任何 iterator 对象都应当同时实现 tp_iter 和 tp_iternext。 一个迭代器的 tp_iter 处理器应当返回一个指向该迭代器的新引用。 它的 tp_iternext 处理器应当返回一个指向迭代操作的下一个对象的新引用，如果还有下一个对象的话。 如果迭代已到达末尾，则 tp_iternext 可以返回 NULL 而不设置异常，或者也可以在返回 NULL 的基础上 *额外* 设置 StopIteration；避免异常可以产生更好的性能。 如果发生了实际的错误，则 tp_iternext 应当总是设置一个异常并返回 NULL。

3.6. 弱引用支持

One of the goals of Python 弱引用实现的目标之一是允许任意类型参与弱引用机制而不会在重视性能的对象（例如数字）上产生额外开销。

参见: weakref 模块的文档。

对于可被弱引用的对象，扩展类型必须设置 tp_flags 字段的 Py_TPFLAGS_MANAGED_WEAKREF 比特位。 旧式的 tp_weaklistoffset 字段应当保持为零。

具体地说，以下就是静态声明的类型对象的样子:

```
static PyTypeObject TrivialType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    /* ... 省略了其他成员以使代码简短 ... */
    .tp_flags = Py_TPFLAGS_MANAGED_WEAKREF | ...,
};
```

唯一的额外补充是 tp_dealloc 需要清除任何弱引用 (通过调用 PyObject_ClearWeakRefs()):

```
static void
Trivial_dealloc(PyObject *op)
{
    /* 在调用任何析构器之前先清除弱引用 */
    PyObject_ClearWeakRefs(op);
    /* ... 省略了析构代码的其余部分以保持简短 ... */
    Py_TYPE(op)->tp_free(op);
}
```

3.7. 更多建议

为了学习如何为你的新数据类型实现任何特定方法，请获取 CPython 源代码。 进入 Objects 目录，然后在 C 源文件中搜索 tp_ 加上你想要的函数 (例如，tp_richcompare)。 你将找到你想要实现的函数的例子。

当你需要验证一个对象是否为你实现的类型的具体实例时，请使用 `PyObject_TypeCheck()` 函数。它的一个用法示例如下:

```
if (!PyObject_TypeCheck(some_object, &MyType)) {
    PyErr_SetString(PyExc_TypeError, "arg #1 not a mything");
    return NULL;
}
```

参见:

下载CPython源代码版本。

https://www.python.org/downloads/source/

GitHub上开发CPython源代码的CPython项目。

https://github.com/python/cpython