《源码探秘 CPython》76. 实例对象的属性访问（上）

原创 古明地觉 古明地觉的编程教室 2022-04-26 08:30 发表于北京

收录于合集
#CPython
97个 >

楔子

之前在讨论名字空间的时候提到，在Python中，形如 **x.y** 样式的表达式被称之为**属性引用**，其中 x 指向某个对象，y 为对象的某个属性。并且这个属性可以是很多种，比如：整数、字符串、函数、类、甚至是模块等等。

那么下面来看看虚拟机是怎么实现属性引用的？

属性引用

还是看一个简单的类，然后观察它的字节码。

```
class Girl:

    def __init__(self):
        self.name = "satori"
        self.age = 16

    def get_info(self):
        return f"name: {self.name}, age: {self.age}"

g = Girl()
name = g.name
g.get_info()
```

字节码如下，这里只看模块的字节码。

```
  # class Girl: 对应的字节码, 这里就不赘述了
  0 LOAD_BUILD_CLASS
  2 LOAD_CONST               0 (<code object Girl at 0x00...>)
  4 LOAD_CONST               1 ('Girl')
  6 MAKE_FUNCTION            0
  8 LOAD_CONST               1 ('Girl')
 10 CALL_FUNCTION            2
 12 STORE_NAME               0 (Girl)

  # g = Girl() 对应的字节码, 不再赘述
 14 LOAD_NAME                0 (Girl)
 16 CALL_FUNCTION            0
 18 STORE_NAME               1 (g)

  # name = g.name 对应的字节码
  # 加载变量 g
 20 LOAD_NAME                1 (g)
  # 获取 g.name, 加载属性用的是 LOAD_ATTR
 22 LOAD_ATTR                2 (name)
```

```
  # 将结果交给变量 name 保存
 24 STORE_NAME               2 (name)

  # g.get_info() 对应的字节码
  # 加载变量 g
 26 LOAD_NAME                1 (g)
  # 获取方法 g.get_info，加载方法用的是 LOAD_METHOD
 28 LOAD_METHOD              3 (get_info)
  # 调用，注意指令是 CALL_METHOD，不是 CALL_FUNCTION
  # 但显然 CALL_METHOD 内部也是调用了 CALL_FUNCATION
 30 CALL_METHOD              0
  # 从栈顶弹出返回值
 32 POP_TOP
  # return None
 34 LOAD_CONST               2 (None)
 36 RETURN_VALUE
```

除了 LOAD_METHOD 和 LOAD_ATTR，其它的指令我们都见过了，因此下面重点分析这两条指令。

```
case TARGET(LOAD_METHOD): {
    //从符号表中获取符号，因为是 g.get_info
    //那么这个 name 就指向字符串对象 "get_info"
    PyObject *name = GETITEM(names, oparg);
    //从栈顶获取元素obj，显然这个 obj 就是代码中的 g
    PyObject *obj = TOP();
    //meth 是一个 PyObject * 指针
    //显然它要指向一个方法
    PyObject *meth = NULL;

    //这里是获取 obj 中和符号 name 绑定的方法，然后让meth指向它
    //具体做法就是调用 _PyObject_GetMethod，传入二级指针&meth
    //然后让 meth 存储的地址变成指向具体方法的地址
    int meth_found = _PyObject_GetMethod(obj, name, &meth);

    //如果 meth == NULL，raise AttributeError
    if (meth == NULL) {
        /* Most likely attribute wasn't found. */
        goto error;
    }

    //注意:无论是 Girl.get_info、还是 g.get_info
    //对应的指令都是 LOAD_METHOD
    //类去调用的话，说明得到是一个未绑定的方法，说白了就等价于函数
    //实例去调用的话，会得到一个绑定的方法，相当于对函数进行了封装
    //关于绑定和未绑定我们后面会详细介绍
    if (meth_found) {
        //如果meth_found为1，说明meth是一个绑定的方法，obj就是self
        //将 meth 设置为栈顶元素，然后再将 obj 压入栈中
        SET_TOP(meth);
        PUSH(obj);  // self
    }
    else {
        //否则 meth 是一个未绑定的方法
        //那么将栈顶元素设置为 NULL，然后将 meth 压入栈中
        SET_TOP(NULL);
        Py_DECREF(obj);
        PUSH(meth);
    }
    DISPATCH();
}
```

获取方法是LOAD_METHOD，获取属性 LOAD_ATTR，来看一下。

```c
case TARGET(LOAD_ATTR): {
    //可以看到和LOAD_METHOD本质上是类似的，但更简单一些
    //name 依旧是符号，这里指向字符串对象 "name"
    PyObject *name = GETITEM(names, oparg);
    //从栈顶获取变量 g
    PyObject *owner = TOP();
    //res 显然就是获取属性返回的结果了
    //通过PyObject_GetAttr进行获取
    PyObject *res = PyObject_GetAttr(owner, name);
    Py_DECREF(owner);
    //设置到栈顶
    SET_TOP(res);
    if (res == NULL)
        goto error;
    DISPATCH();
}
```

所以这两个指令本身是很简单的，而核心在 PyObject_GetAttr 和 _PyObject_GetMethod 上面，前者用于获取属性、后者用于获取方法。

```c
//Objects/object.c
PyObject *
PyObject_GetAttr(PyObject *v, PyObject *name)
{
    //v: 对象
    //name: 属性名

    //获取类型对象
    PyTypeObject *tp = Py_TYPE(v);

    //name必须是一个字符串
    if (!PyUnicode_Check(name)) {
        PyErr_Format(PyExc_TypeError,
                     "attribute name must be string, not '%.200s'",
                     name->ob_type->tp_name);
        return NULL;
    }
    //通过类型对象的 tp_getattro 成员获取对应的属性
    //所以实例获取属性(包括方法)的时候都是通过类来获取的
    //比如 g.xx()本质上就是 Girl.xx(g)
    //但是 Girl.xx(g)是不是长得有点丑啊，于是 Python 提供了 g.xx()
    //所以 g.xx()就是 Girl.xx(g)的一个语法糖，底层还是通过 Girl.xx(g)执行的
    //虽然 g.xx() 等价于 Girl.xx(g)，但 Girl.xx() 仍是 Girl.xx()
    //实例调用的时候会将自身作为参数传进去，但是类不会
    //如此类获取的话(Girl.xx)叫函数，实例获取(girl.xx)的话叫方法，后面会介绍
    if (tp->tp_getattro != NULL)
        return (*tp->tp_getattro)(v, name);

    //tp_getattro 和 tp_getattr 功能一样，但前者可以支持中文
    if (tp->tp_getattr != NULL) {
        const char *name_str = PyUnicode_AsUTF8(name);
        if (name_str == NULL)
            return NULL;
        return (*tp->tp_getattr)(v, (char *)name_str);
    }

    //属性不存在，抛出异常
    PyErr_Format(PyExc_AttributeError,
                 "'%.50s' object has no attribute '%U'",
                 tp->tp_name, name);
    return NULL;
}
```

PyTypeObject里面定义了两个与属性访问相关的操作：tp_getattro 和 tp_getattr。其中 tp_getattro 是优先选择的属性访问动作，而 tp_getattr 已不推荐使用。

这两者的区别在 PyObject_GetAttr 中已经显示的很清楚了，主要是在属性名的使用上，tp_getattro 所使用的属性名是一个PyUnicodeObject *，而 tp_getattr所使用的属性名是一个 char *。

因此如果某个类型定义了 tp_getattro 和 tp_getattr，那么 PyObject_GetAttr 优先使用 tp_getattro，因为这位老铁写在上面。

那么问题来了，自定义类对象的 tp_getattro 对应哪一个 C 函数呢？显然我们要去找 object。PyBaseObject_Type 的 tp_getattro 为 PyObject_GenericGetAttr，因此虚拟机在创建 Girl 这个类时，也会将此操作继承下来。

```
//Objects/object.c
PyObject *
PyObject_GenericGetAttr(PyObject *obj, PyObject *name)
{
    return _PyObject_GenericGetAttrWithDict(obj, name, NULL, 0);
}

PyObject *
_PyObject_GenericGetAttrWithDict(PyObject *obj, PyObject *name,
                                 PyObject *dict, int suppress)
{
    //拿到 obj 的类型对象
    //对于我们的例子来说，显然是class Girl
    PyTypeObject *tp = Py_TYPE(obj);
    //描述符
    PyObject *descr = NULL;
    //返回值
    PyObject *res = NULL;
    //描述符的 __get__ 函数
    descrgetfunc f;
    Py_ssize_t dictoffset;
    PyObject **dictptr;

    //name 必须是字符串
    if (!PyUnicode_Check(name)){
        PyErr_Format(PyExc_TypeError,
                     "attribute name must be string, not '%.200s'",
                     name->ob_type->tp_name);
        return NULL;
    }
    Py_INCREF(name);

    //属性字典为不为空，是初始化是否完成的重要标志
    //如果为空，说明还没有初始化，那么需要先初始化
    if (tp->tp_dict == NULL) {
        if (PyType_Ready(tp) < 0)
            goto done;
    }

    //从 mro 顺序列表中获取属性对应的值，并检测是否为描述符
    //如果属性不存在、或者存在但对应的值不是描述符，则返回 NULL
    descr = _PyType_Lookup(tp, name);

    f = NULL;
    if (descr != NULL) {
        Py_INCREF(descr);
        //如果 descr 不为 NULL，说明该属性被代理了
        //descr 是描述符，f 就是它的 __get__ 方法
        //f = descr.__class__.__get__
```

```
50          f = descr->ob_type->tp_descr_get;
51          //补充:
52          //__get__ 对应 PyTypeObject 的 tp_descr_get
53          //__set__ 对应 PyTypeObject 的 tp_descr_set
54
55          //f 不为 NULL,并且 descr 是数据描述符
56          if (f != NULL && PyDescr_IsData(descr)) {
57              //那么直接调用描述符的 __get__ 方法,返回结果
58              res = f(descr, obj, (PyObject *)obj->ob_type);
59              if (res == NULL && suppress &&
60                      PyErr_ExceptionMatches(PyExc_AttributeError)) {
61                  PyErr_Clear();
62              }
63              goto done;
64          }
65      }
66
67      //走到这说明要获取的属性没有被代理,或者说代理它的非数据描述符
68      //当然还有一种情况,这种情况上一篇文章貌似没提到
69      //就是属性被数据描述符代理,但是该数据描述符没有 __get__
70      //那么仍会优先从实例对象自身的 __dict__ 中寻找属性
71      if (dict == NULL) {
72          /* Inline _PyObject_GetDictPtr */
73          dictoffset = tp->tp_dictoffset;
74          //但如果dict为NULL,并且dictoffset不为0
75          //说明继承自变长对象,那么要调整 tp_dictoffset
76          //这部分代码一会单独说
77          if (dictoffset != 0) {
78              if (dictoffset < 0) {
79                  Py_ssize_t tsize;
80                  size_t size;
81
82                  tsize = ((PyVarObject *)obj)->ob_size;
83                  if (tsize < 0)
84                      tsize = -tsize;
85                  size = _PyObject_VAR_SIZE(tp, tsize);
86                  _PyObject_ASSERT(obj, size <= PY_SSIZE_T_MAX);
87
88                  dictoffset += (Py_ssize_t)size;
89                  _PyObject_ASSERT(obj, dictoffset > 0);
90                  _PyObject_ASSERT(obj, dictoffset % SIZEOF_VOID_P == 0);
91              }
92              dictptr = (PyObject **) ((char *)obj + dictoffset);
93              dict = *dictptr;
94          }
95      }
96      //dict不为NULL,从字典中获取
97      if (dict != NULL) {
98          Py_INCREF(dict);
99          res = PyDict_GetItemWithError(dict, name);
100         if (res != NULL) {
101             Py_INCREF(res);
102             Py_DECREF(dict);
103             goto done;
104         }
105         else {
106             Py_DECREF(dict);
107             if (PyErr_Occurred()) {
108                 if (suppress && PyErr_ExceptionMatches(PyExc_AttributeEr
109 ror)) {
110                     PyErr_Clear();
111                 }
112                 else {
113                     goto done;
```

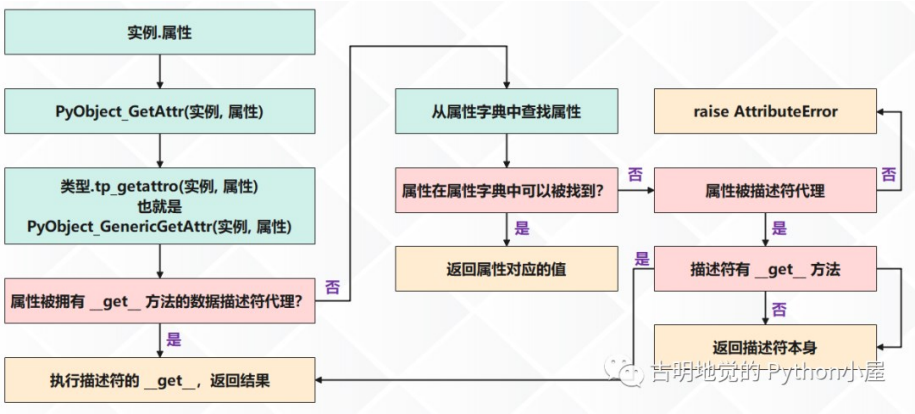
```
                }
            }
        }
    }

    //程序走到这里，说明什么呢？
    //显然意味着实例的属性字典里面没有要获取的属性
    //但如果下面的 f != NULL 成立，说明属性被代理了
    //并且代理属性的描述符是非数据描述符，它的优先级低于实例
    //所以实例会先到自身的属性字典中查找，找不到再去执行描述符的 __get__
    if (f != NULL) {
        // 第一个参数是描述符本身，也就是 __get__ 里面的 self
        // 第二个参数是实例对象，也就是 __get__ 里面的 instance
        // 第三个参数是类对象，也就是 __get__ 里面的 owner
        res = f(descr, obj, (PyObject *)Py_TYPE(obj));
        if (res == NULL && suppress &&
                PyErr_ExceptionMatches(PyExc_AttributeError)) {
            PyErr_Clear();
        }
        goto done;
    }

    //程序能走到这里，说明属性字典里面没有要找的属性
    //并且也没有执行描述符的 __get__
    //但如果 describe 还不为 NULL，这说明什么呢？
    //显然该属性仍被描述符代理了，只是这个描述符没有 __get__
    //如果是这种情况，那么会返回描述符本身
    if (descr != NULL) {
        res = descr;
        descr = NULL;
        goto done;
    }

    //找不到，就报错
    if (!suppress) {
        PyErr_Format(PyExc_AttributeError,
                     "'%.50s' object has no attribute '%U'",
                     tp->tp_name, name);
    }
  done:
    Py_XDECREF(descr);
    Py_DECREF(name);
    return res;
}
```

这里面有两个我们上一篇文章没有提到的地方，下面补充一下：

```
class Descriptor:

    def __set__(self, instance, value):
        print("__set__")

class B:

    name = Descriptor()

b = B()
# b 的属性字典没有 name，描述符也没有 __get__
# 那么这个时候会返回描述符本身
print(b.name)  # <__main__.Descriptor object at 0x0...>

# 此时属性字典里面有 name 了
b.__dict__["name"] = "古明地觉"
# 由于 name 是被数据描述符代理的
```

```
18  # 按理说获取属性时会执行数据描述符的 __get__
19  # 但是这个数据描述符压根没有 __get__
20  # 因此还是会从属性字典中查找
21  print(b.name)  # 古明地觉
```

所以获取属性的流程如下：



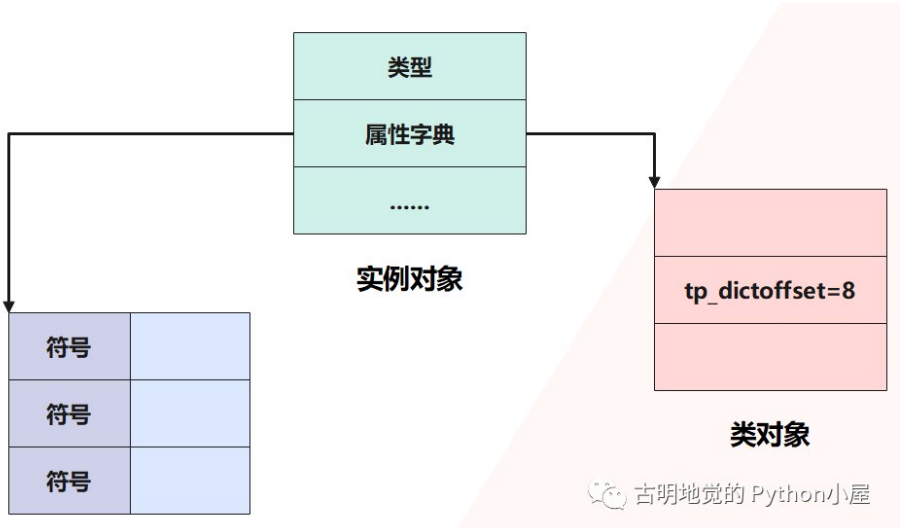
获取方法也与之类似，调用的是 _PyObject_GetMethod，这里就不再看了。

实例对象的属性字典

在属性访问的时候，可以通过 g.__dict__ 这种形式访问。但是这就奇怪了，在之前的描述中，我们看到调用 Girl 创建实例的时候，虚拟机并没有为实例创建PyDictObject对象啊。

记得介绍metaclass的时候，我们说过这样一句话，对于任意继承object的自定义类对象来说，为实例对象申请的内存大小为**PyBaseObject_Type->tp_basicsize + 16**，其中的16是**2 * sizeof(PyObject *)**。后面跟着的两个PyObject *的空间被设置给了 tp_dictoffset 和 tp_weaklistoffset，那么现在是时候揭开谜底了。

在创建自定义类对象时我们看到，虚拟机设置了一个名为tp_dictoffset的域，从名字推断，这个可能就是实例对象中__dict__的偏移位置。



图中画出的dict对象就是我们期望的实例对象的属性字典，这个猜想可以在PyObject_GenericGetAttr中得到证实。

```
1  //object.c
2  PyObject *
```

```c
PyObject_GenericGetAttr(PyObject *obj, PyObject *name)
{
    return _PyObject_GenericGetAttrWithDict(obj, name, NULL, 0);
}

PyObject *
_PyObject_GenericGetAttrWithDict(PyObject *obj, PyObject *name,
                                 PyObject *dict, int suppress)
{
  //...
    //那么显然要从实例对象自身的__dict__中寻找属性
    if (dict == NULL) {
        /* Inline _PyObject_GetDictPtr */
        dictoffset = tp->tp_dictoffset;
        if (dictoffset != 0) {
            //但如果dict为NULL，并且dictoffset不为0
            //说明继承自变长对象，那么要调整tp_dictoffset
            if (dictoffset < 0) {
                Py_ssize_t tsize;
                size_t size;

                tsize = ((PyVarObject *)obj)->ob_size;
                if (tsize < 0)
                    tsize = -tsize;
                size = _PyObject_VAR_SIZE(tp, tsize);
                assert(size <= PY_SSIZE_T_MAX);

                dictoffset += (Py_ssize_t)size;
                assert(dictoffset > 0);
                assert(dictoffset % SIZEOF_VOID_P == 0);
            }
            dictptr = (PyObject **) ((char *)obj + dictoffset);
            dict = *dictptr;
        }
    }
  //...
}
```

如果dictoffset小于0，意味着 Girl 是继承自类似 list 这样的变长对象，虚拟机会对dictoffset做一些处理，最终仍然会使dictoffset指向实例的内存中额外申请的位置。
PyObject_GenericGetAttr正是根据这个dictoffset获得了一个dict对象。

而这个设置的动作最终会调用 PyObject_GenericSetAttr ，也就是girl.__dict__最初被创建的地方。

```c
//object.c
int
PyObject_GenericSetAttr(PyObject *obj, PyObject *name, PyObject *value)
{
    return _PyObject_GenericSetAttrWithDict(obj, name, value, NULL);
}


int
_PyObject_GenericSetAttrWithDict(PyObject *obj, PyObject *name,
                                 PyObject *value, PyObject *dict)
{
    PyTypeObject *tp = Py_TYPE(obj);
    PyObject *descr;
    descrsetfunc f;
    PyObject **dictptr;
    int res = -1;

```

```
    //name必须指向PyUnicodeObject对象
    if (!PyUnicode_Check(name)){
        PyErr_Format(PyExc_TypeError,
                     "attribute name must be string, not '%.200s'",
                     name->ob_type->tp_name);
        return -1;
    }

    //字典为空,则进行初始化
    if (tp->tp_dict == NULL && PyType_Ready(tp) < 0)
        return -1;

    Py_INCREF(name);

    //获取描述符
    descr = _PyType_Lookup(tp, name);

    //如果描述符不为空,并且内部有 __set__
    //那么执行 __set__
    if (descr != NULL) {
        Py_INCREF(descr);
        f = descr->ob_type->tp_descr_set;
        if (f != NULL) {
            res = f(descr, obj, value);
            goto done;
        }
    }

    if (dict == NULL) {
        //PyObject_GenericGetAttr中的关键代码
        //根据dictoffset获取dict对象
        dictptr = _PyObject_GetDictPtr(obj);
        if (dictptr == NULL) {
            if (descr == NULL) {
                PyErr_Format(PyExc_AttributeError,
                             "'%.100s' object has no attribute '%U'",
                             tp->tp_name, name);
            }
            else {
                PyErr_Format(PyExc_AttributeError,
                             "'%.50s' object attribute '%U' is read-only",
                             tp->tp_name, name);
            }
            goto done;
        }
        res = _PyObjectDict_SetItem(tp, dictptr, name, value);
    }
    else {
        Py_INCREF(dict);
        if (value == NULL)
            res = PyDict_DelItem(dict, name);
        else
            res = PyDict_SetItem(dict, name, value);
        Py_DECREF(dict);
    }
    if (res < 0 && PyErr_ExceptionMatches(PyExc_KeyError))
        PyErr_SetObject(PyExc_AttributeError, name);

  done:
    Py_XDECREF(descr);
    Py_DECREF(name);
    return res;
}
```

这部分的内容有点抽象，介绍的不是很详细，感兴趣的话可以试着深挖一下。但其实也没有太大必要，大概理解就行。

再论描述符

前面我们看到，在 PyType_Ready 中，虚拟机会填充tp_dict，其中与操作名对应的是一个个的描述符。那时我们看到的是描述符这个概念在Python内部是如何实现的，现在我们将要剖析的是描述符在Python的类机制中究竟会起到怎样的作用。

虚拟机对自定义类对象或实例对象进行属性访问时，描述符将对属性访问的行为产生重大影响。一般而言，如果一个类存在__get__、__set__、__delete__操作(不要求三者同时存在)，那么它的实例便可以称之为描述符。在slotdefs中，我们会看到这三种魔法方法对应的操作。

```
//typeobject.c

TPSLOT("__get__", tp_descr_get, slot_tp_descr_get, wrap_descr_get,
       "__get__($self, instance, owner, /)\n--\n\nReturn an attribute of
instance, which is of type owner."),
TPSLOT("__set__", tp_descr_set, slot_tp_descr_set, wrap_descr_set,
       "__set__($self, instance, value, /)\n--\n\nSet an attribute of ins
tance to value."),
TPSLOT("__delete__", tp_descr_set, slot_tp_descr_set,
       wrap_descr_delete,
       "__delete__($self, instance, /)\n--\n\nDelete an attribute of inst
ance."),
```

而在虚拟机访问实例对象的属性时，描述符的一个作用就是影响虚拟机对属性的选择。从 PyObject_GenericGetAttr 源码中可以看到，虚拟机会先在实例对象自身的__dict__中寻找属性，也会在实例对象的类型对象的mro顺序列表中寻找属性，我们将前一种属性称之为**实例属性**，后一种属性称之为**类属性**。所以在属性的选择上，有如下规律：

- 虚拟机优先按照实例属性、类属性的顺序选择属性，即实例属性优先于类属性；
- 如果发现有一个同名、并且被数据描述符代理的类属性，那么该描述符会优先于实例属性被虚拟机选择；

这两条规则在对属性进行设置时仍然会被严格遵守，换句话说，如果执行 **ins.xxx = yyy** 时，在 type(ins) 中也出现了为 xxx 属性、并且还被数据描述符代理了。那么不好意思，此时虚拟机会选择描述符，并执行它的 __set__ 方法；如果是非数据描述符，那么就不再走 __set__ 了，而是设置属性（因为压根没有 __set__），也就是 a.__dict__['xxx'] = yyy。

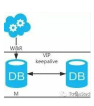
关于描述符的相关内容，上一篇文章介绍的很详细了，这里不再赘述。

由于这部分内容比较多，我们分两篇文章介绍，目前就先说到这里。

收录于合集 #CPython 97

‹ 上一篇
《源码探秘 CPython》77. 实例对象的属性访问（下）

下一篇 ›
《源码探秘 CPython》75. 实例对象是如何创建的?



MySQL · 参数故事 · timed_mutexes

夜雨成诗

力扣 428. 序列化和反序列化 N 叉树 DFS

钰娘娘知识汇总