

## 《源码探秘 CPython》73. 自定义类对象的底层实现与 metaclass（下）

原创 古明地觉 古明地觉的编程教室 2022-04-21 08:30



微信扫一扫  
关注该公众号

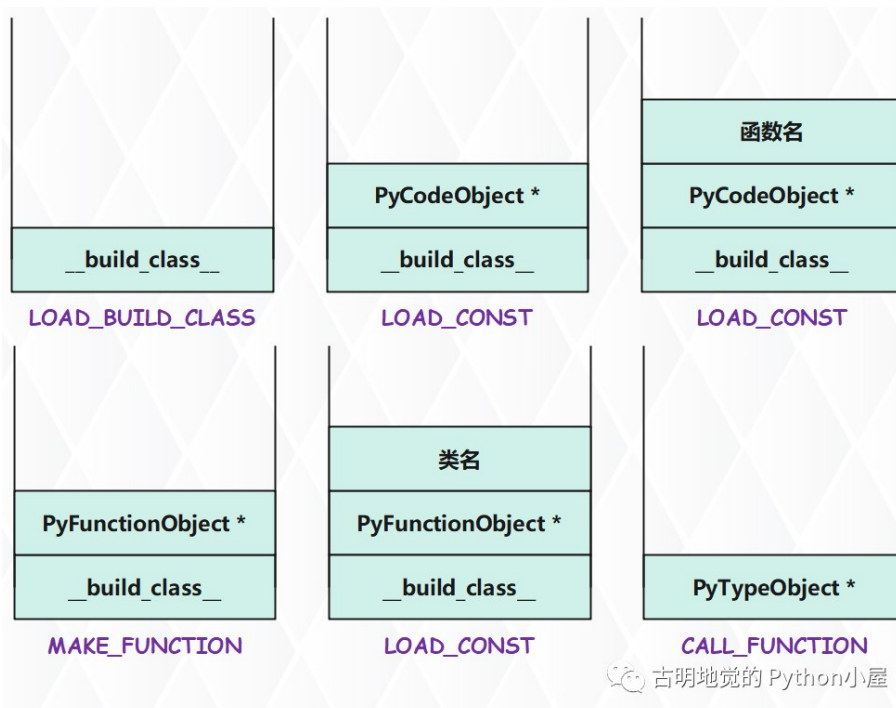
收录于合集  
#CPython

97个 >

回顾一下类是怎么创建的，首先会通过指令LOAD\_BUILD\_CLASS将内置函数\_\_build\_class\_\_压入运行时栈。然后将A对应的PyCodeObject包装成一个PyFunctionObject；最后再调用\_\_build\_class\_\_将PyFunctionObject变成PyTypeObject，也就是我们使用的类对象。

```
1 class A: pass
2 class B: pass
3 class C: pass
4 class D: pass
5 class E: pass
6 class F: pass
7
8 MyClass = __build_class__(lambda: None, "MyClass",
9                           A, B, C, D, E, F)
10 print(MyClass) # <class '__main__.MyClass'>
11
12 for cls in MyClass.__mro__:
13     print(cls)
14 """
15 <class '__main__.MyClass'>
16 <class '__main__.A'>
17 <class '__main__.B'>
18 <class '__main__.C'>
19 <class '__main__.D'>
20 <class '__main__.E'>
21 <class '__main__.F'>
22 <class 'object'>
23 """
```

我们以运行时栈的变化，来描述一下上述过程：



那么接下来的重点就是 `__build_class__`，它是如何将一个函数变成类的，我们来看一下。内置函数的相关实现，位于 `Python/bitinmodule.c` 中。

```

1 static PyMethodDef builtin_methods[] = {
2     {"__build_class__", (PyCFunction)(void*)(void))builtin__build_class
3     __,
4     METH_FASTCALL | METH_KEYWORDS, build_class_doc},
5     //...
6     //...
7 }

```

\_\_build\_class\_\_ 是在 Python 里面使用的函数，它在底层对应 builtin\_\_build\_class\_\_。

```

1 static PyObject *
2 builtin__build_class__(PyObject *self, PyObject *const *args, Py_ssize_t
3 nargs,
4                          PyObject *kwnames)
5 {
6     PyObject *func, *name, *bases, *mkw, *meta, *winner, *prep, *ns, *o
7     rig_bases;
8     PyObject *cls = NULL, *cell = NULL;
9     int isclass = 0; /* initialize to prevent gcc warning */
10
11     //我们说了底层调用的是builtin__build_class__
12     //class A: 会被翻译成builtin__build_class__(PyFunctionObject, class
13     name)
14     //所以这个函数至少需要两个参数
15     if (nargs < 2) {
16         //参数不足, 报错, 还记的这个报错信息吗? 之前测试过的
17         PyErr_SetString(PyExc_TypeError,
18                         "__build_class__: not enough arguments");
19         return NULL;
20     }
21     //类对应的PyFunctionObject
22     func = args[0]; /* Better be callable */
23     if (!PyFunction_Check(func)) {
24         //如果不是PyFunctionObject, 报错, 这个信息有印象吗?
25         PyErr_SetString(PyExc_TypeError,
26                         "__build_class__: func must be a function");
27         return NULL;
28     }
29
30     //类对应的名字, __build_class__的时候, 类肯定要有名字
31     name = args[1];
32     if (!PyUnicode_Check(name)) {
33         //必须要是一个PyUnicodeObject, 否则报错
34         PyErr_SetString(PyExc_TypeError,
35                         "__build_class__: name is not a string");
36         return NULL;
37     }
38
39     //args[0]表示 PyFunctionObject*
40     //args[1]表示 class name
41     //args + 2 开始是继承的基类, 基类的个数显然是 nargs - 2
42     //所以这里是拿到所有的基类
43     orig_bases = _PyTuple_FromArray(args + 2, nargs - 2);
44     if (orig_bases == NULL)
45         return NULL;
46
47     //这个update_bases比较有趣, 我们一会单独说
48     bases = update_bases(orig_bases, args + 2, nargs - 2);
49     if (bases == NULL) {
50         Py_DECREF(orig_bases);
51         return NULL;
52     }
53
54     //获取 metaclass

```

```

55     if (kwnames == NULL) {
56         meta = NULL;
57         mkw = NULL;
58     }
59     else {
60         mkw = _PyStack_AsDict(args + nargs, kwnames);
61         if (mkw == NULL) {
62             Py_DECREF(bases);
63             return NULL;
64         }
65
66         meta = _PyDict_GetItemIdWithError(mkw, &PyId_metaclass);
67         if (meta != NULL) {
68             Py_INCREF(meta);
69             if (_PyDict_DelItemId(mkw, &PyId_metaclass) < 0) {
70                 Py_DECREF(meta);
71                 Py_DECREF(mkw);
72                 Py_DECREF(bases);
73                 return NULL;
74             }
75             /* metaclass is explicitly given, check if it's indeed a cl
76 ass */
77             isclass = PyType_Check(meta);
78         }
79         else if (PyErr_Occurred()) {
80             Py_DECREF(mkw);
81             Py_DECREF(bases);
82             return NULL;
83         }
84     }
85     //如果meta为NULL, 这意味着用户没有指定metaclass
86     if (meta == NULL) {
87         //然后尝试获取基类, 如果没有基类
88         if (PyTuple_GET_SIZE(bases) == 0) {
89             //指定 metaclass 为 type
90             meta = (PyObject *) (&PyType_Type);
91         }
92         //否则获取第一个继承的基类的 metaclass
93         else {
94             //拿到第一个基类
95             PyObject *base0 = PyTuple_GET_ITEM(bases, 0);
96             //拿到第一个基类的__class__
97             meta = (PyObject *) (base0->ob_type);
98         }
99         //meta也是一个类
100         Py_INCREF(meta);
101         isclass = 1;
102     }
103
104     //如果设置了元类, 那么isclass为1, if 为真
105     if (isclass) {
106         //选择出了元类, 下面这一步就要解决元类冲突
107         //假设有两个继承type的元类 MyType1 和 MyType2
108         //然后 Base1 的元类是 MyType1、Base2 的元类是 MyType2
109         //那么如果class A(Base1, Base2)的话, 就会报错
110         //因为在Python中有一个要求, 假设class A(Base1, Base2, ..., BaseN)
111         //Base1的元类叫MyType1、...、BaseN的元类叫MyTypeN
112         //那么必须满足:
113         /*
114         MyType1是MyType2的子类或者父类;
115         MyType1是MyType3的子类或者父类;
116         MyType1是MyType4的子类或者父类;
117         ....
118         MyType1是MyTypeN的子类或者父类;

```

```

119      */
120      //而之所以存在这一限制, 原因就是为了避免属性冲突
121      winner = (PyObject *)_PyType_CalculateMetaclass((PyTypeObject *)
122 )meta,
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137      //寻找__prepare__
138      if (_PyObject_LookupAttrId(meta, &PyId___prepare__, &prep) < 0) {
139          ns = NULL;
140      }
141      //这个__prepare__必须返回一个mapping
142      //如果返回None, 那么等价于返回一个空字典
143      else if (prep == NULL) {
144          ns = PyDict_New();
145      }
146      else {
147          //否则将字典返回
148          PyObject *pargs[2] = {name, bases};
149          //我们看到这里涉及了一个函数调用, 这个函数应该有印象吧
150          ns = _PyObject_FastCallDict(prep, pargs, 2, mkw);
151          Py_DECREF(prep);
152      }
153      if (ns == NULL) {
154          Py_DECREF(meta);
155          Py_XDECREF(mkw);
156          Py_DECREF(bases);
157          return NULL;
158      }
159      if (!PyMapping_Check(ns)) {
160          //如果返回的不是一个字典, 那么报错
161          //这个错误信息我们也见过了
162          PyErr_Format(PyExc_TypeError,
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2610
2611
2612
2613
2614
2615
2616
2617
2618
2619
2620
2621
2622
2623
2624
2625
2626
2627
2628
2629
2630
2631
2632
2633
2634
2635
2636
2637
2638
2639
2640
2641
2642
2643
2644
2645
2646
2647
2648
2649
2650
2651
2652
2653
2654
2655
2656
2657
2658
2659
2660
2661
2662
2663
2664
2665
2666
2667
2668
2669
2670
2671
2672
2673
2674
2675
2676
2677
2678
2679
2680
2681
2682
2683
2684
2685
2686
2687
2688
2689
2690
2691
2692
2693
2694
2695
2696
2697
2698
2699
2700
2701
2702
2703
2704
2705
2706
2707
2708
2709
2710
2711
2712
2713
2714
2
```

为静态元信息。在静态元信息中，隐藏着所有的类对象应该如何创建的信息，注意：是所有的类对象。

从源码中我们可以看到，如果用户指定了metaclass，那么会选择指定的metaclass；如果没有指定，那么会使用第一个继承的基类的metaclass作为该class的metaclass。

对于实例对象，所有的元信息都存储在对应的类对象中。但是对于类对象来说，其元信息的静态元信息存储在对应的元类中，动态元信息则存储在本身的local名字空间中。

可为什么这么做呢？为什么对于类对象来说，其元信息要游离成两部分呢？都存在metaclass里面不香吗？这是因为用户在.py文件中可以定义不同的class，这个元信息必须、且只能是动态的，所以它不适合保存在metaclass中。而存储在metaclass里面的，一定是诸如类对象的创建策略等这些所有class都会共用的元信息。

注意：我们说元信息游离成两部分指的是自定义类对象，内置类对象的元信息也是都存储在metaclass中的。

因为内置类对象都是静态提供的，它们都具备相同的接口集合(底层都是PyTypeObject结构体实例)，支持什么操作一开始就定义好了。只不过有的可以用，有的不能用。比如PyLongObject可以使用nb\_add，但是PyDictObject不能。而PyDictObject可以使用mp\_subscript，但是PyLongObject不可以。

尽管如此，但这不影响它们的所有元信息都存储在类型对象中。而用户自定义的类对象，接口是动态的，不可能在metaclass中静态指定。



然后再来说一说源码中的update\_bases，它比较有意思。

```
1 class Foo:
2     name = "古明地觉"
3
4 class Bar:
5
6     def __mro_entries__(self, bases):
7         return (Foo, tuple,)
8
9 class MyClass(Bar()):
10     pass
11
12 print(
13     MyClass("123"),
14     MyClass.name
15 ) # ('1', '2', '3') 古明地觉
```

我们在继承的时候，都是继承一个类，但是这里的MyClass居然继承了一个实例对象。相信结果你已经猜出来了，如果继承的是实例，那么会去调用实例的\_\_mro\_entries\_\_。因此MyClass继承的其实是Foo、tuple，并且\_\_mro\_entries\_\_必须返回一个元组，否则报错。

另外，如果一个类继承了一个拥有\_\_mro\_entries\_\_的实例，那么该类会多出一个属性叫\_\_orig\_bases\_\_。我们回顾一下type\_new里面的几行关键代码：

```
1 orig_bases = PyTuple_FromArray(args + 2, nargs - 2);
2 if (orig_bases == NULL)
3     return NULL;
4 bases = update_bases(orig_bases, args + 2, nargs - 2);
```

里面有两个变量orig\_bases和bases。我们知道Python的类都有一个\_\_bases\_\_属性，对应这里的bases；但鲜为人知的是，它还有一个属性叫\_\_orig\_bases\_\_，对应这里的orig\_bases。

```
1 class Foo:
2     name = "古明地觉"
3
```

```

4 class Bar:
5
6     def __mro_entries__(self, bases):
7         return (Foo, tuple,)
8
9 class MyClass(Bar()):
10     pass
11
12 print(MyClass.__orig_bases__)
13 print(MyClass.__bases__)
14 """
15 (<__main__.Bar object at 0x000001B4D50BD040>,)
16 (<class '__main__.Foo'>, <class 'tuple'>)
17 """

```

`__bases__` 表示继承的基类，但很明显我们这里继承的不是 `Foo` 和 `tuple`，而是 `Bar()`，所以 `__bases__` 是 `update_bases` 函数基于 `__orig_bases__` 处理得到的。

`__orig_bases__` 表示继承的对象，该对象可以是一个类对象，也可以是一个实例对象；如果是实例对象，那么在 `update_bases` 函数中，会调用它的 `__mro_entries__` 方法，该方法返回一个包含类对象的元组，然后设置到 `__bases__` 中。

以上就是函数 `update_bases` 的作用，但有两点需要注意。

- 1) 只有继承了拥有 `__mro_entries__` 方法的实例的类，才有 `__orig_bases__` 属性；
- 2) 这样的类，在 Python 里面不能手动调用 `type` 来创建；

我们来解释一下，首先是第一点：

```

1 print(hasattr(MyClass, "__orig_bases__"))
2 print(hasattr(Foo, "__orig_bases__"))
3 print(hasattr(Bar, "__orig_bases__"))
4 """
5 True
6 False
7 False
8 """

```

我们看到只有 `MyClass` 有 `__orig_bases__` 属性，因为它继承了拥有 `__mro_entries__` 方法的实例，而 `Foo` 和 `Bar` 则没有。

然后是第二点，这样的类不可以手动调用 `type` 来创建。

```

1 class Foo:
2     name = "古明地觉"
3
4 class Bar:
5
6     def __mro_entries__(self, bases):
7         return (Foo, tuple,)
8
9 try:
10     MyClass = type("MyClass", (Bar(),), {})
11 except TypeError as e:
12     print(e)
13 # type() doesn't support MRO entry resolution; use types.new_class()
14
15 # 所以这样的类请通过 class 关键字创建
16 # 如果必须手动创建的话, 那么可以使用 types.new_class
17 import types
18 MyClass = types.new_class("MyClass", (Bar(),), {})
19 print(MyClass.name) # 古明地觉
20 print(MyClass("123")) # ('1', '2', '3')

```

以上就是 `update_bases` 函数所干的事情，但是问题来了，如果继承的实例对象没有 `__mro_entries__` 方法怎么办？

```
1 class Base:
2
3     def __init__(self, *args):
4         if len(args) == 0:
5             return
6         elif len(args) == 3:
7             name, bases, attrs = args
8         else:
9             raise ValueError("args 的长度必须是 0 或 3")
10        self.name = name
11        self.bases = bases
12        self.attrs = attrs
13
14 base = Base()
15
16 class MyClass(base):
17     pass
18
19 print(type(MyClass) is Base) # True
20 print(MyClass.name) # MyClass
21 print(MyClass.bases == (base,)) # True
22 print(
23     MyClass.attrs
24 ) # {'__module__': '__main__', '__qualname__': 'MyClass'}
```

显然 `MyClass` 继承的是 `Base` 的实例对象，并且 `Base` 里面也没有定义 `__mro_entries__`，那么虚拟机就不会再使用 `type` 来创建 `MyClass` 了。而是会使用 `Base` 来创建，所以得到的 `MyClass` 就是一个 `Base` 的实例对象。

现在算是彻底理解 `update_bases` 的作用了，因为不能保证继承的都是类，所以还需要进行检测，如果不是类，那么就执行上面的逻辑。但是说实话，这个特性几乎不用，因为既然要继承，那么就应该继承类。虽然通过 `__mro_entries__` 可以整一些花活，甚至也能简化逻辑，但最好还是不要用，因为它会让代码变得难以理解。



`builtin__build_class__` 的逻辑我们上面省略了一部分，至于省略部分的逻辑也很简单，既然元类以及相关参数都准备好了，那么接下来就是对类进行创建了。

我们知道调用一个对象，本质上会执行其类对象的 `__call__`。所以调用类对象创建实例对象，会执行 `type.__call__(cls, ...)`；调用元类创建类对象，会执行 `type.__call__(type, ...)`，因为元类的类对象还是它本身。

所以不管调用的是元类、还是类对象，都会执行元类的 `__call__`，在底层对应 `&PyType_Type` 的 `tp_call` 成员，它指向了 `type_call` 函数。

```
1 //Objects/typeobject.c
2 static PyObject *
3 type_call(PyTypeObject *type, PyObject *args, PyObject *kwargs)
4 {
5     PyObject *obj;
6     //tp_new 负责创建实例, 所以它不能为空
7     if (type->tp_new == NULL) {
8         PyErr_Format(PyExc_TypeError,
9                     "cannot create '%.100s' instances",
10                     type->tp_name);
11         return NULL;
12     }
13     //调用tp_new为实例申请内存
```

```

14  obj = type->tp_new(type, args, kwds);
15  obj = _Py_CheckFunctionResult((PyObject*)type, obj, NULL);
16  if (obj == NULL)
17      return NULL;
18
19  //如果调用的是&PyType_Type, 并且只接收了一个位置参数
20  //那么显然是查看对象类型, 执行完__new__之后直接返回
21  if (type == &PyType_Type &&
22      PyTuple_Check(args) && PyTuple_GET_SIZE(args) == 1 &&
23      (kwds == NULL ||
24       (PyDict_Check(kwds) && PyDict_GET_SIZE(kwds) == 0)))
25      return obj;
26
27  //记得我们之前说过, __new__里面一定要返回类的实例对象
28  //否则是不会执行__init__函数的, 从这里我们也看到了
29  //如果obj的类型不是对应的类、或者其子类, 那么直接返回
30  if (!PyType_IsSubtype(Py_TYPE(obj), type))
31      return obj;
32
33  //然后获取obj的类型
34  type = Py_TYPE(obj);
35  //如果内部存在__init__函数, 那么执行
36  if (type->tp_init != NULL) {
37      int res = type->tp_init(obj, args, kwds);
38      if (res < 0) {
39          assert(PyErr_Occurred());
40          Py_DECREF(obj);
41          obj = NULL;
42      }
43      else {
44          assert(!PyErr_Occurred());
45      }
46  }
47  //执行完构造函数之后, 再将对象返回
48  //返回的 obj 可以是类对象、也可以是实例对象
49  return obj;
50 }

```

type\_call 里面的逻辑非常简单, 就是先调用对象的 tp\_new 创建实例, 然后执行 tp\_init (如果有)。至于返回的是类对象还是实例对象, 则取决于 type\_call 的第一个参数, 如果第一个参数是元类, 那么返回的就是类对象, 否则是实例对象。

因此创建的核心逻辑就隐藏在对象的 tp\_new 中, 不同对象的tp\_new指向的函数不同。但对于创建类对象而言, 显然执行的是 &PyType\_Type 的 tp\_new, 它指向的是 type\_new 函数。

```

3652  offsetof(PyTypeObject, tp_dict),          /* tp_dictoffset */
3653  type_init,                                /* tp_init */
3654  0,                                         /* tp_alloc */
3655  type_new,                                 /* tp_new */
3656  PyObject_GC_Del,                          /* tp_free */
3657  (inquiry)type_is_gc,                      /* tp_is_gc */
3658  };
3659

```

👤 古明地觉的 Python小屋

这个type\_new是我们创建自定义类对象的第一案发现场, 我们看一下它的源码, 位于 typeobject.c 中。这个函数的代码比较长, 我们会有删减, 像那些检测的代码我们就省略掉了。

```

1  static PyObject *
2  type_new(PyTypeObject *metatype, PyObject *args, PyObject *kwds)
3  {
4      //都是类的一些动态元信息
5      PyObject *name, *bases = NULL, *orig_dict, *dict = NULL;
6      PyObject *qualname, *slots = NULL, *tmp, *newslots, *cell;

```



```

7     PyObject *type = NULL, *base, *tmptype, *winner;
8     PyHeapTypeObject *et;
9     PyMemberDef *mp;
10    Py_ssize_t i, nbases, nslots, slotoffset, name_size;
11    int j, may_add_dict, may_add_weak, add_dict, add_weak;
12    _Py_IDENTIFIER(__qualname__);
13    _Py_IDENTIFIER(__slots__);
14    _Py_IDENTIFIER(__classcell__);
15
16    //如果metaclass是type的话
17    if (metatype == &PyType_Type) {
18        //获取位置参数和关键字参数个数
19        const Py_ssize_t nargs = PyTuple_GET_SIZE(args);
20        const Py_ssize_t nkws = nkws == NULL ? 0 : PyDict_GET_SIZE(kwd
21 s);
22
23        //位置参数为1, 关键字参数为0, 你想到了什么
24        //type(xxx), 是不是这个呀
25        if (nargs == 1 && nkws == 0) {
26            PyObject *x = PyTuple_GET_ITEM(args, 0);
27            Py_INCREF(Py_TYPE(x));
28            //这显然是初学Python时就知道的
29            //查看一个变量指向的对象的类型, 获取类型之后直接返回
30            return (PyObject *) Py_TYPE(x);
31        }
32
33        //如果上面的if不满足, 会走这里
34        //表示现在不再是查看类型了, 而是创建类
35        //那么要求位置参数必须是3个, 否则报错
36        if (nargs != 3) {
37            PyErr_SetString(PyExc_TypeError,
38                            "type() takes 1 or 3 arguments");
39            return NULL;
40        }
41    }
42
43    /* Check arguments: (name, bases, dict) */
44    //现在显然是确定参数类型, 因为传递的三个参数是有类型要求的
45    //必须是PyUnicodeObject、PyTupleObject、PyDictObject
46    if (!PyArg_ParseTuple(args, "U0!O!:type.__new__", &name, &PyTuple_T
47 ype,
48                            &bases, &PyDict_Type, &orig_dict))
49        /*
50         TypeError: type.__new__() argument 1 must be str, not xxx
51         TypeError: type.__new__() argument 2 must be tuple, not xxx
52         TypeError: type.__new__() argument 3 must be dict, not xxx
53         */
54        return NULL;
55    //无论是使用 class 关键字创建类, 还是使用 type 创建类
56    //底层都是执行的 type_new
57
58    //处理基类
59    nbases = PyTuple_GET_SIZE(bases);
60    if (nbases == 0) {
61        //如果没有继承基类, 那么会默认继承object
62        //__base__ 设置为 object
63        base = &PyBaseObject_Type;
64        //__bases__ 设置为 (object,)
65        bases = PyTuple_Pack(1, base);
66        if (bases == NULL)
67            return NULL;
68        nbases = 1;
69    }
70    else {

```

```

71     _Py_IDENTIFIER(__mro_entries__);
72     //如果继承了基类, 那么循环遍历bases
73     for (i = 0; i < nbases; i++) {
74         //拿到每一个基类
75         tmp = PyTuple_GET_ITEM(bases, i);
76         //如果类型为&PyType_Type, 进行下一次循环
77         if (PyType_Check(tmp)) {
78             continue;
79         }
80         //如果类型不是&PyType_Type, 说明继承的不是类
81         //于是寻找__mro_entries__
82         if (_PyObject_LookupAttrId(tmp, &PyId___mro_entries__, &tmp)
83 < 0) {
84             return NULL;
85         }
86         if (tmp != NULL) {
87             PyErr_SetString(PyExc_TypeError,
88                             "type() doesn't support MRO entry resolut
89 ion; "
90                             "use types.new_class()");
91             Py_DECREF(tmp);
92             return NULL;
93         }
94     }
95     //寻找父类的metaclass
96     //采用之前说的解决元类冲突时所采取的策略
97     winner = _PyType_CalculateMetaclass(metatype, bases);
98
99     //每个类都有 __base__ 和 __bases__
100    //__bases__ 表示直接继承的所有类, __base__ 是继承的第一个类
101    //那么下面这行代码是做什么呢? 直接 base = bases[0]就好了
102    //其实这个 best_base 所做的事情没有这么简单
103    //它还负责检测基类之间是否发生了冲突
104    base = best_base(bases);
105    if (base == NULL) {
106        return NULL;
107    }
108
109    Py_INCREF(bases);
110 }
111
112 dict = PyDict_Copy(orig_dict);
113 if (dict == NULL)
114     goto error;
115
116 //处理用户定义了__slots__属性的逻辑
117 //一旦定义了__slots__, 那么类的实例对象就没有属性字典了
118 slots = _PyDict_GetItemIdWithError(dict, &PyId___slots__);
119 nslots = 0;
120 add_dict = 0;
121 add_weak = 0;
122 may_add_dict = base->tp_dictoffset == 0;
123 may_add_weak = base->tp_weaklistoffset == 0 && base->tp_itemsize ==
124 0;
125 if (slots == NULL) {
126     //.....
127 }
128 else {
129     //.....
130 }
131
132 //为自定义类对象申请内存
133 type = (PyTypeObject *)metatype->tp_alloc(metatype, nslots);
134 if (type == NULL)

```

```

135         goto error;
136
137     et = (PyHeapTypeObject *)type;
138     Py_INCREF(name);
139     et->ht_name = name;
140     et->ht_slots = slots;
141     slots = NULL;
142
143     /* 初始化tp_flags */
144     type->tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_HEAPTYPE |
145         Py_TPFLAGS_BASETYPE;
146     if (base->tp_flags & Py_TPFLAGS_HAVE_GC)
147         type->tp_flags |= Py_TPFLAGS_HAVE_GC;
148
149     //设置PyTypeObject中的各个域
150     type->tp_as_async = &et->as_async;
151     type->tp_as_number = &et->as_number;
152     type->tp_as_sequence = &et->as_sequence;
153     type->tp_as_mapping = &et->as_mapping;
154     type->tp_as_buffer = &et->as_buffer;
155     type->tp_name = PyUnicode_AsUTF8AndSize(name, &name_size);
156     if (!type->tp_name)
157         goto error;
158     if (strlen(type->tp_name) != (size_t)name_size) {
159         PyErr_SetString(PyExc_ValueError,
160             "type name must not contain null characters");
161         goto error;
162     }
163
164     /* 设置基类和基类列表 */
165     type->tp_bases = bases;
166     bases = NULL;
167     Py_INCREF(base);
168     type->tp_base = base;
169
170     /* 设置属性字典 */
171     Py_INCREF(dict);
172     type->tp_dict = dict;
173
174     //设置__module__
175     if (_PyDict_GetItemIdWithError(dict, &PyId__module__) == NULL) {
176         //.....
177     }
178
179     //设置__qualname__, 即“全限定名”
180     qualname = _PyDict_GetItemIdWithError(dict, &PyId__qualname__);
181     //.....
182
183     //如果自定义的class中重写了__new__方法
184     //将__new__对应的函数改造为静态方法, 并替换掉默认的__new__
185     tmp = _PyDict_GetItemIdWithError(dict, &PyId__new__);
186     if (tmp != NULL && PyFunction_Check(tmp)) {
187         tmp = PyStaticMethod_New(tmp);
188         if (tmp == NULL)
189             goto error;
190         if (_PyDict_SetItemId(dict, &PyId__new__, tmp) < 0) {
191             Py_DECREF(tmp);
192             goto error;
193         }
194         Py_DECREF(tmp);
195     }
196     else if (tmp == NULL && PyErr_Occurred()) {
197         goto error;
198     }

```

```

199
200 //获取__init_subclass__, 如果子类继承了父类
201 //那么会触发父类的__init_subclass__
202 tmp = _PyDict_GetItemIdWithError(dict, &PyId__init_subclass__);
203 if (tmp != NULL && PyFunction_Check(tmp)) {
204     tmp = PyClassMethod_New(tmp);
205     if (tmp == NULL)
206         goto error;
207     if (_PyDict_SetItemId(dict, &PyId__init_subclass__, tmp) < 0) {
208         Py_DECREF(tmp);
209         goto error;
210     }
211     Py_DECREF(tmp);
212 }
213 else if (tmp == NULL && PyErr_Occurred()) {
214     goto error;
215 }
216
217 //设置__class_getitem__, 这个是什么?类似于__getitem__
218 //__class_getitem__支持通过 类["xxx"] 的方式访问
219 tmp = _PyDict_GetItemIdWithError(dict, &PyId__class_getitem__);
220 //.....
221
222 //为自定义类对象对应的实例对象设置内存大小信息
223 type->tp_basicsize = slotoffset;
224 type->tp_itemsize = base->tp_itemsize;
225 type->tp_members = PyHeapType_GET_MEMBERS(et);
226 //.....
227
228 //调用PyType_Ready对自定义类对象进行初始化
229 if (PyType_Ready(type) < 0)
230     goto error;
231
232 /* Put the proper slots in place */
233 fixup_slot_dispatchers(type);
234
235 if (type->tp_dictoffset) {
236     et->ht_cached_keys = _PyDict_NewKeysForClass();
237 }
238
239 if (set_names(type) < 0)
240     goto error;
241
242 if (init_subclass(type, kwds) < 0)
243     goto error;
244
245 Py_DECREF(dict);
246 return (PyObject *)type;
247
248 error:
249     Py_XDECREF(dict);
250     Py_XDECREF(bases);
251     Py_XDECREF(slots);
252     Py_DECREF(type);
253     return NULL;
254 }

```

虚拟机首先会解析出类名、基类列表和属性字典，然后根据基类列表及传入的metaclass确定最佳的metaclass和base。

随后，虚拟机会调用 `metatype->tp_alloc` 尝试为要创建的类对象分配内存。这里需要注意的是，在 `&PyType_Type` 中，我们会发现 `tp_alloc` 是一个 `NULL`，这显然不正常。但是不要忘记，虚拟机会通过 `PyType_Ready` 对所有的类对象进行初始化。

在这个初始化过程中，有一项动作就是从基类继承各种操作。由于type.\_\_bases\_\_中的第一个基类是object，所以type会继承object的tp\_alloc操作，即 PyType\_GenericAlloc。

对于所有继承object的类对象来说，PyType\_GenericAlloc 将申请metatype->tp\_basicsize + metatype->tp\_itemsize大小的内存空间。

从PyType\_Type的定义中我们看到，这个大小实际就是 sizeof(PyHeapTypeObject) + sizeof(PyMemberDef)。

因此在这里应该就明白了PyHeapTypeObject这个老铁到底是干嘛用的了，之前因为偏移量的问题，折腾了不少功夫，甚至让人觉得这有啥用啊，但是现在意识到了，这个老铁是为自定义类对象准备的。

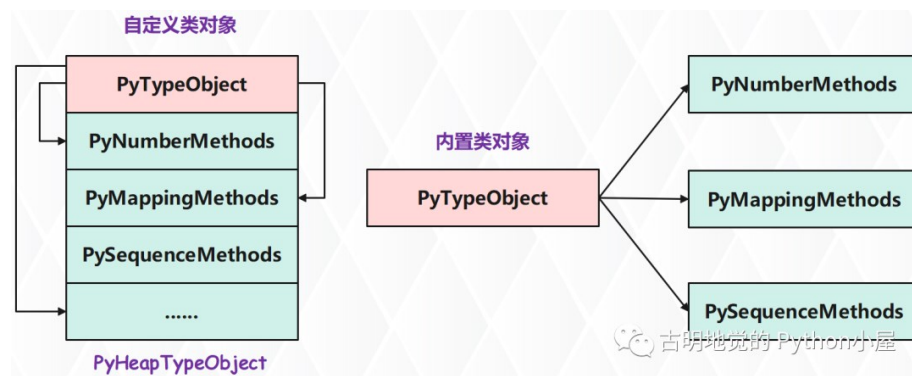
接下来就是设置自定义类对象的各个域，其中包括了在tp\_dict上设置属性字典，也就是\_\_dict\_\_。另外注意的是，这里还计算了类对象对应的实例对象所需要的内存大小信息，换言之，自定义类在创建一个实例对象时，需要为这个实例对象申请多大的内存空间呢？

对于任意继承object的自定义类对象来说，这个大小为PyBaseObject\_Type->tp\_basicsize + 16，其中的16是2 \* sizeof(PyObject \*)。

而之所以后面要跟着两个PyObject \*的空间，是因为这些空间的地址被设置给了 tp\_dictoffset，以及 tp\_weaklistoffset。这一点将在介绍实例对象时进行解析，它是和实例对象的属性字典密切相关的。

最后，虚拟机还会调用PyType\_Ready对自定义类对象进行和内置类对象一样的初始化动作，到此自定义类对象才算正式创建完毕。因此内置类对象是底层静态定义好的，启动之后再调用PyType\_Ready 完善一下即可；但自定义类对象则不同，它需要运行时动态创建，这是一个复杂的过程。但最后，两者都会调用 PyType\_Ready。

那么内置类对象和自定义对象在内存布局上面有什么区别呢？毕竟都是类对象。



本质上，无论是自定义类对象还是内置类对象，在虚拟机内部，都可以用一个PyTypeObject来表示。

但不同的是，内置类对象对应的PyTypeObject以及与其关联的PyNumberMethods等操作簇的内存位置都是在编译时确定的，它们在内存中的位置是分离的。而自定义类对象对应的PyTypeObject和PyNumberMethods等操作簇的内存位置是连续的，必须在运行时动态分配内存。

另外，自定义类对象对应的 PyTypeObject 和相关操作簇组合起来，被称为PyHeapTypeObject。

```
1 typedef struct _heaptypobject {
2     PyTypeObject ht_type;
3     PyAsyncMethods as_async;
4     PyNumberMethods as_number;
5     PyMappingMethods as_mapping;
6     PySequenceMethods as_sequence;
7     PyBufferProcs as_buffer;
8     PyObject *ht_name, *ht_slots, *ht_qualname;
```

```
9     struct _dictkeysobject *ht_cached_keys;  
10 } PyHeapTypeObject;
```

内置类对象有哪些操作是静态定义好的，所以相关操作是分离的；但自定义类对象的相关操作簇必须紧随其后，且顺序也有讲究，只有这样才能通过偏移量 `offset` 准确找到指定的操作。

现在我们也对Python的可调用(callable)这个概念有一个感性认识了，可调用这个概念是一个相当通用的概念，不拘泥于对象、大小，只要类型对象定义了`tp_call`操作，就能进行调用操作。我们已经看到，调用`metaclass`得到类对象，调用类对象得到实例对象。如果类对象也定义了`tp_call`，那么还可以继续对实例对象进行调用。

收录于合集 [#CPython 97](#)

[< 上一篇](#)

《源码探秘 CPython》74. 彻底搞懂描述符

[下一篇 >](#)

《源码探秘 CPython》72. 自定义类对象的  
底层实现与 `metaclass` (上)

喜欢此内容的人还喜欢

python 7天进阶之路-对象和json转换  
缪斯之子



[系列]微服务·深入理解 gRPC - Part2  
走向架构师的每一天



Abaqus python脚本开发 第三章 各类指令的方法对象变量 (3)  
山石结构

