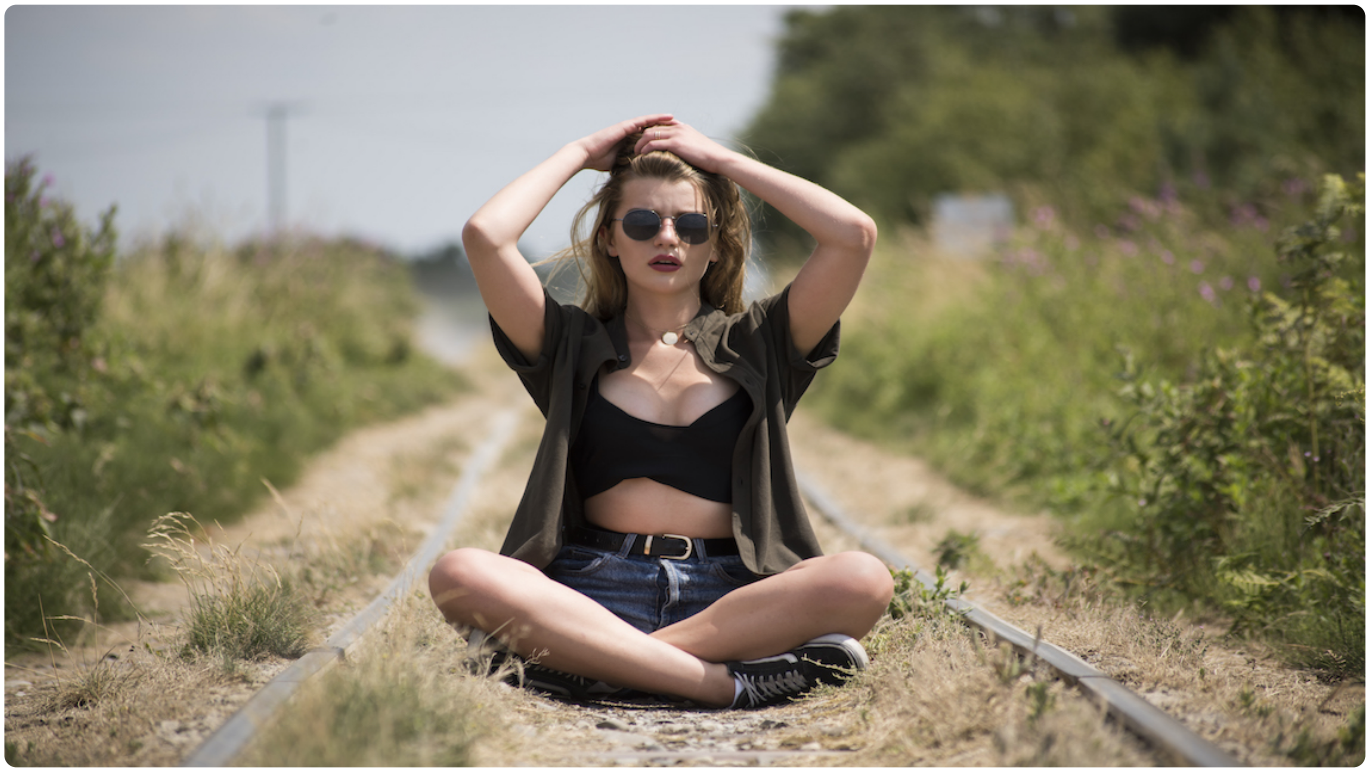
12 | 面向对象（下）：如何实现一个搜索引擎?

2019-06-05 景霄

Python核心技术与实战 进入课程 >



讲述：冯永吉
时长 14:48 大小 13.57M

你好，我是景霄。这节课，我们来实现一个 Python 的搜索引擎（search engine）。

承接上文，今天这节课的主要目的是，带你模拟敏捷开发过程中的迭代开发流程，巩固面向对象的程序设计思想。

我们将从最简单最直接的搜索做起，一步步优化，这其中，我不会涉及到过多的超纲算法，但不可避免会介绍一些现代搜索引擎中的基础概念，例如语料（corpus）、倒序索引（inverted index）等。

如果你对这方面本身有些了解，自然可以轻松理解；即使你之前完全没接触过搜索引擎，也不用过分担心，我会力求简洁清晰，降低学习难度。同时，我希望你把更多的精力放在面向对象的建模思路上。

“高大上”的搜索引擎

引擎一词尤如其名，听起来非常酷炫。搜索引擎，则是新世纪初期互联网发展最重要的入口之一，依托搜索引擎，中国和美国分别诞生了百度、谷歌等巨型公司。

搜索引擎极大地方便了互联网生活，也成为上网必不可少的刚需工具。依托搜索引擎发展起来的互联网广告，则成了硅谷和中国巨头的核心商业模式；而搜索本身，也在持续进步着，Facebook 和微信也一直有意向在自家社交产品架设搜索平台。

关于搜索引擎的价值我不必多说了，今天我们主要来看一下搜索引擎的核心构成。

听 Google 的朋友说，他们入职培训的时候，有一门课程叫做 The life of a query，内容是讲用户在浏览器中键入一串文字，按下回车后发生了什么。今天我也按照这个思路，来简单介绍下。

我们知道，**一个搜索引擎由搜索器、索引器、检索器和用户接口四个部分组成。**

搜索器，通俗来讲就是我们常提到的爬虫（scrawler），它能在互联网上大量爬取各类网站的内容，送给索引器。索引器拿到网页和内容后，会对内容进行处理，形成索引（index），存储于内部的数据库等待检索。

最后的用户接口很好理解，是指网页和 App 前端界面，例如百度和谷歌的搜索页面。用户通过用户接口，向搜索引擎发出询问（query），询问解析后送达检索器；检索器高效检索后，再将结果返回给用户。

爬虫知识不是我们今天学习的重点，这里我就不做深入介绍了。我们假设搜索样本存在于本地磁盘上。

为了方便，我们只提供五个文件的检索，内容我放在了下面这段代码中：

复制代码

```
# 1.txt
I have a dream that my four little children will one day live in a nation where they wi

# 2.txt
I have a dream that one day down in Alabama, with its vicious racists, . . . one day ri

# 3.txt
```

```
I have a dream that one day every valley shall be exalted, every hill and mountain shal

# 4.txt
This is our hope. . . With this faith we will be able to hew out of the mountain of des

# 5.txt
And when this happens, and when we allow freedom ring, when we let it ring from every v
```

我们先来定义 SearchEngineBase 基类。这里我先给出了具体的代码，你不必着急操作，还是那句话，跟着节奏慢慢学，再难的东西也可以啃得下来。

复制代码

```
class SearchEngineBase(object):
    def __init__(self):
        pass

    def add_corpus(self, file_path):
        with open(file_path, 'r') as fin:
            text = fin.read()
        self.process_corpus(file_path, text)

    def process_corpus(self, id, text):
        raise Exception('process_corpus not implemented.')

    def search(self, query):
        raise Exception('search not implemented.')

def main(search_engine):
    for file_path in ['1.txt', '2.txt', '3.txt', '4.txt', '5.txt']:
        search_engine.add_corpus(file_path)

    while True:
        query = input()
        results = search_engine.search(query)
        print('found {} result(s):'.format(len(results)))
        for result in results:
            print(result)
```


SearchEngineBase 可以被继承，继承的类分别代表不同的算法引擎。每一个引擎都应该实现 process_corpus() 和 search() 两个函数，对应我们刚刚提到的索引器和检索器。main() 函数提供搜索器和用户接口，于是一个简单的包装界面就有了。

具体来看这段代码，其中，

add_corpus() 函数负责读取文件内容，将文件路径作为 ID，连同内容一起送到 process_corpus 中。

process_corpus 需要对内容进行处理，然后文件路径为 ID ，将处理后的内容存下来。处理后的内容，就叫做索引（index）。

search 则给定一个询问，处理询问，再通过索引检索，然后返回。

好，理解这些概念后，接下来，我们实现一个最基本的可以工作的搜索引擎，代码如下：

复制代码

```
class SimpleEngine(SearchEngineBase):
    def __init__(self):
        super(SimpleEngine, self).__init__()
        self.__id_to_texts = {}

    def process_corpus(self, id, text):
        self.__id_to_texts[id] = text

    def search(self, query):
        results = []
        for id, text in self.__id_to_texts.items():
            if query in text:
                results.append(id)
        return results

search_engine = SimpleEngine()
main(search_engine)


########## 输出 ##########


simple
found 0 result(s):
little
found 2 result(s):
1.txt
2.txt
```

你可能很惊讶，只需要短短十来行代码居然就可以了吗？

没错，正是如此，这段代码我们拆开来看一下：

SimpleEngine 实现了一个继承 SearchEngineBase 的子类，继承并实现了 process_corpus 和 search 接口，同时，也顺手继承了 add_corpus 函数（当然你想重写也是可行的），因此我们可以在 main() 函数中直接调取。

在我们新的构造函数中，`self.__id_to_texts = {}` 初始化了自己的私有变量，也就是这个用来存储文件名到文件内容的字典。

process_corpus() 函数则非常直白地将文件内容插入到字典中。这里注意，ID 需要是唯一的，不然相同 ID 的新内容会覆盖掉旧的内容。

search 直接枚举字典，从中找到要搜索的字符串。如果能够找到，则将 ID 放到结果列表中，最后返回。

你看，是不是非常简单呢？这个过程始终贯穿着面向对象的思想，这里我为你梳理成了几个问题，你可以自己思考一下，当成是一个小复习。

现在你对父类子类的构造函数调用顺序和方法应该更清楚了吧？

集成的时候，函数是如何重写的？

基类是如何充当接口作用的（你可以自行删掉子类中的重写函数，抑或是修改一下函数的参数，看一下会报什么错）？

方法和变量之间又如何衔接起来的呢？

好的，我们重新回到搜索引擎这个话题。

相信你也能看得出来，这种实现方式简单，但显然是一种很低效的方式：每次索引后需要占用大量空间，因为索引函数并没有做任何事情；每次检索需要占用大量时间，因为所有索引库的文件都要被重新搜索一遍。如果把语料的信息量视为 n，那么这里的时间复杂度和空间复杂度都应该是 O(n) 级别的。

而且，还有一个问题：这里的 query 只能是一个词，或者是连起来的几个词。如果你想要搜索多个词，它们又分散在文章的不同位置，我们的简单引擎就无能为力了。

这时应该怎么优化呢？

最直接的一个想法，就是把语料分词，看成一个个的词汇，这样就只需要对每篇文章存储它所有词汇的 set 即可。根据齐夫定律（Zipf’ s law，https://en.wikipedia.org/wiki/Zipf%27s_law），在自然语言的语料库里，一个单词出现的频率与它在频率表里的排名成反比，呈现幂律分布。因此，语料分词的做法可以大大提升我们的存储和搜索效率。

那具体该如何实现呢？

Bag of Words 和 Inverted Index

我们先来实现一个名叫 Bag of Words 的搜索模型。请看下面的代码：

复制代码

```
import re

class BOWEngine(SearchEngineBase):
    def __init__(self):
        super(BOWEngine, self).__init__()
        self.__id_to_words = {}

    def process_corpus(self, id, text):
        self.__id_to_words[id] = self.parse_text_to_words(text)

    def search(self, query):
        query_words = self.parse_text_to_words(query)
        results = []
        for id, words in self.__id_to_words.items():
            if self.query_match(query_words, words):
                results.append(id)
        return results

    @staticmethod
    def query_match(query_words, words):
        for query_word in query_words:
            if query_word not in words:
                return False
        return True

    @staticmethod
    def parse_text_to_words(text):
        # 使用正则表达式去除标点符号和换行符
        text = re.sub(r'[^\w ]', ' ', text)
        # 转为小写
```

```
        text = text.lower()
        # 生成所有单词的列表
        word_list = text.split(' ')
        # 去除空白单词
        word_list = filter(None, word_list)
        # 返回单词的 set
        return set(word_list)

search_engine = BOWEngine()
main(search_engine)


########## 输出 ##########


i have a dream
found 3 result(s):
1.txt
2.txt
3.txt
freedom children
found 1 result(s):
5.txt
```

你应该发现，代码开始变得稍微复杂些了。

这里我们先来理解一个概念，BOW Model，即 Bag of Words Model，中文叫做词袋模型。这是 NLP 领域最常见最简单的模型之一。

假设一个文本，不考虑语法、句法、段落，也不考虑词汇出现的顺序，只将这个文本看成这些词汇的集合。于是相应的，我们把 id_to_texts 替换成 id_to_words，这样就只需要存这些单词，而不是全部文章，也不需要考虑顺序。

其中，process_corpus() 函数调用类静态函数 parse_text_to_words，将文章打碎形成词袋，放入 set 之后再放到字典中。

search() 函数则稍微复杂一些。这里我们假设，想得到的结果，是所有的搜索关键词都要出现在同一篇文章中。那么，我们需要同样打碎 query 得到一个 set，然后把 set 中的每一个词，和我们的索引中每一篇文章进行核对，看一下要找的词是否在其中。而这个过程由静态函数 query_match 负责。

你可以回顾一下上节课学到的静态函数，我们看到，这两个函数都是没有状态的，它们不涉及对象的私有变量（没有 self 作为参数），相同的输入能够得到完全相同的输出结果。因此设置为静态，可以方便其他的类来使用。

可是，即使这样做，每次查询时依然需要遍历所有 ID，虽然比起 Simple 模型已经节约了大量时间，但是互联网上有上亿个页面，每次都全部遍历的代价还是太大了。到这时，又该如何优化呢？

你可能想到了，我们每次查询的 query 的单词量不会很多，一般也就几个、最多十几个的样子。那可不可以从这里下手呢？

再有，词袋模型并不考虑单词间的顺序，但有些人希望单词按顺序出现，或者希望搜索的单词在文中离得近一些，这种情况下词袋模型现任就无能为力了。

针对这两点，我们还能做得更好吗？显然是可以的，请看接下来的这段代码。

复制代码

```
import re

class BOWInvertedIndexEngine(SearchEngineBase):
    def __init__(self):
        super(BOWInvertedIndexEngine, self).__init__()
        self.inverted_index = {}

    def process_corpus(self, id, text):
        words = self.parse_text_to_words(text)
        for word in words:
            if word not in self.inverted_index:
                self.inverted_index[word] = []
            self.inverted_index[word].append(id)

    def search(self, query):
        query_words = list(self.parse_text_to_words(query))
        query_words_index = list()
        for query_word in query_words:
            query_words_index.append(0)

        # 如果某一个查询单词的倒序索引为空，我们就立刻返回
        for query_word in query_words:
            if query_word not in self.inverted_index:
                return []

        result = []
        while True:
```

```
28
29              # 首先，获得当前状态下所有倒序索引的 index
30              current_ids = []
31
32              for idx, query_word in enumerate(query_words):
33                  current_index = query_words_index[idx]
34                  current_inverted_list = self.inverted_index[query_word]
35
36                  # 已经遍历到了某一个倒序索引的末尾，结束 search
37                  if current_index >= len(current_inverted_list):
38                      return result
39
40                  current_ids.append(current_inverted_list[current_index])
41
42              # 然后，如果 current_ids 的所有元素都一样，那么表明这个单词在这个元素对应的文档中
43              if all(x == current_ids[0] for x in current_ids):
44                  result.append(current_ids[0])
45                  query_words_index = [x + 1 for x in query_words_index]
46                  continue
47
48              # 如果不是，我们就把最小的元素加一
49              min_val = min(current_ids)
50              min_val_pos = current_ids.index(min_val)
51              query_words_index[min_val_pos] += 1
52
53      @staticmethod
54      def parse_text_to_words(text):
55          # 使用正则表达式去除标点符号和换行符
56          text = re.sub(r'[^\w ]', ' ', text)
57          # 转为小写
58          text = text.lower()
59          # 生成所有单词的列表
60          word_list = text.split(' ')
61          # 去除空白单词
62          word_list = filter(None, word_list)
63          # 返回单词的 set
64          return set(word_list)
65
66  search_engine = BOWInvertedIndexEngine()
67  main(search_engine)
68
69
70  ########## 输出 ##########
71
72
73  little
74  found 2 result(s):
75  1.txt
76  2.txt
77  little vicious
78  found 1 result(s):
79  2.txt
```

首先我要强调一下，**这次的算法并不需要你完全理解**，这里的实现有一些超出了本章知识点。但希望你不要因此退缩，这个例子会告诉你，面向对象编程是如何把算法复杂性隔离开来，而保留接口和其他的代码不变。

我们接着来看这段代码。你可以看到，新模型继续使用之前的接口，仍然只在`__init__()`、`process_corpus()`和`search()`三个函数进行修改。

这其实也是大公司里团队协作的一种方式，**在合理的分层设计后，每一层的逻辑只需要处理好分内的事情即可**。在迭代升级我们的搜索引擎内核时， main 函数、用户接口没有任何改变。当然，如果公司招了新的前端工程师，要对用户接口部分进行修改，新人也不需要过分担心后台的事情，只要做好数据交互就可以了。

继续看代码，你可能注意到了开头的 Inverted Index。Inverted Index Model，即倒序索引，是非常有名的搜索引擎方法，接下来我简单介绍一下。

倒序索引，一如其名，也就是说这次反过来，我们保留的是 word -> id 的字典。于是情况就豁然开朗了，在 search 时，我们只需要把想要的 query_word 的几个倒序索引单独拎出来，然后从这几个列表中找共有的元素，那些共有的元素，即 ID，就是我们想要的查询结果。这样，我们就避免了将所有的 index 过一遍的尴尬。

process_corpus 建立倒序索引。注意，这里的代码都是非常精简的。在工业界领域，需要一个 unique ID 生成器，来对每一篇文章标记上不同的 ID，倒序索引也应该按照这个 unique_id 来进行排序。

至于 search() 函数，你大概了解它做的事情即可。它会根据 query_words 拿到所有的倒序索引，如果拿不到，就表示有的 query word 不存在于任何文章中，直接返回空；拿到之后，运行一个“合并 K 个有序数组”的算法，从中拿到我们想要的 ID，并返回。

> 注意，这里用到的算法并不是最优的，最优的写法需要用最小堆来存储 index。这是一道有名的 leetcode hard 题，有兴趣请参考：https://blog.csdn.net/qqxx6661/article/details/77814794)

遍历的问题解决了，那第二个问题，如果我们想要实现搜索单词按顺序出现，或者希望搜索的单词在文中离得近一些呢？

我们需要在 Inverted Index 上，对于每篇文章也保留单词的位置信息，这样一来，在合并操作的时候处理一下就可以了。

倒序索引我就介绍到这里了，如果你感兴趣可以自行查阅资料。还是那句话，我们的重点是面向对象的抽象，别忘了体会这一思想。

LRU 和多重继承

到这一步，终于，你的搜索引擎上线了，有了越来越多的访问量（QPS）。欣喜骄傲的同时，你却发现服务器有些“不堪重负”了。经过一段时间的调研，你发现大量重复性搜索占据了 90% 以上的流量，于是，你想到了一个大杀器——给搜索引擎加一个缓存。

所以，最后这部分，我就来讲讲缓存和多重继承的内容。

复制代码

```
import pylru

class LRUCache(object):
    def __init__(self, size=32):
        self.cache = pylru.lrucache(size)

    def has(self, key):
        return key in self.cache

    def get(self, key):
        return self.cache[key]

    def set(self, key, value):
        self.cache[key] = value

class BOWInvertedIndexEngineWithCache(BOWInvertedIndexEngine, LRUCache):
    def __init__(self):
        super(BOWInvertedIndexEngineWithCache, self).__init__()
        LRUCache.__init__(self)

    def search(self, query):
        if self.has(query):
            print('cache hit!')
            return self.get(query)

        result = super(BOWInvertedIndexEngineWithCache, self).search(query)
```

```
        self.set(query, result)

        return result

search_engine = BOWInvertedIndexEngineWithCache()
main(search_engine)


########## 输出 ##########


little
found 2 result(s):
1.txt
2.txt
little
cache hit!
found 2 result(s):
1.txt
2.txt
```

它的代码很简单，LRUCache 定义了一个缓存类，你可以通过继承这个类来调用其方法。LRU 缓存是一种很经典的缓存（同时，LRU 的实现也是硅谷大厂常考的算法面试题，这里为了简单，我直接使用 pylru 这个包），它符合自然界的局部性原理，可以保留最近使用过的对象，而逐渐淘汰掉很久没有被用过的对象。

因此，这里的缓存使用起来也很简单，调用 has() 函数判断是否在缓存中，如果在，调用 get 函数直接返回结果；如果不在，送入后台计算结果，然后再塞入缓存。

我们可以看到，BOWInvertedIndexEngineWithCache 类，多重继承了两个类。首先需要注意的是构造函数（上节课的思考题，你思考了吗？）。多重继承有两种初始化方法：

> 第一种，`super(BOWInvertedIndexEngineWithCache, self).__init__()`直接初始化该类的第一个父类，不过使用这种方法时，要求继承链的最顶层父类必须要继承 object；
>
> 第二种，对于多重继承，如果有多个构造函数需要调用， 我们就必须用传统的方法 `LRUCache.__init__(self)` 。

第二个需要注意，query() 函数被子类 BOWInvertedIndexEngineWithCache 再次重载，但是我还需要调用 BOWInvertedIndexEngine 的 search() 函数，这时该怎么办呢？请看

下面这行代码：

复制代码

```
1 super(BOWInvertedIndexEngineWithCache, self).search(query)
```

我们可以强行调用被覆盖的父类的函数。

这样一来，我们就简洁地实现了缓存，而且还是在不影响 BOWInvertedIndexEngine 代码的情况下。这部分内容希望你多读几遍，自己揣摩清楚，通过这个例子多多体会继承的优势。

总结

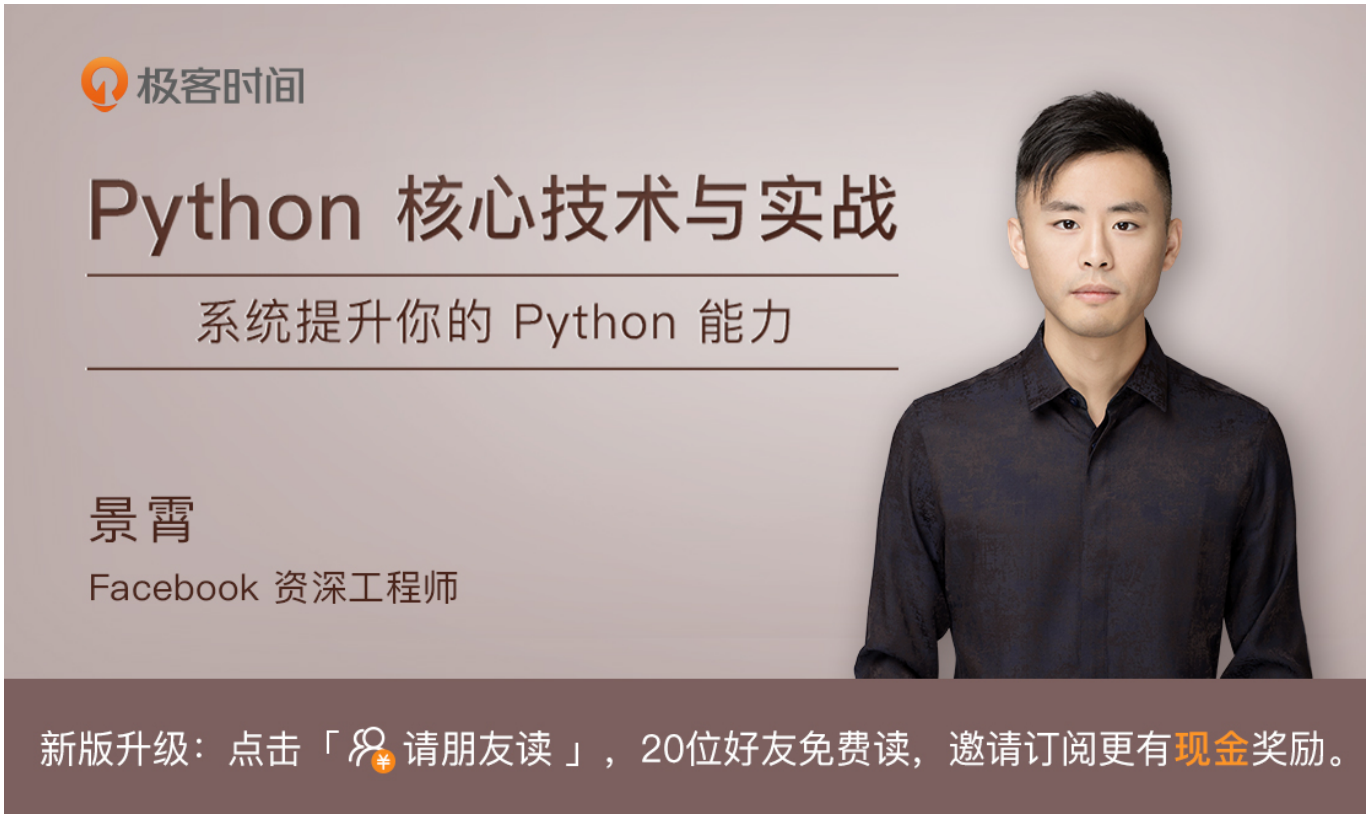
今天这节课是面向对象的实战应用，相比起前面的理论知识，内容其实不那么友好。不过，若你能静下心来，仔细学习，理清楚整个过程的要点，对你理解面向对象必将有所裨益。比如，你可以根据下面两个问题，来检验今天这节课的收获。

> 你能把这节课所有的类的属性和函数抽取出来，自己在纸上画一遍继承关系吗？
>
> 迭代开发流程是怎样的？

其实于我而言，通过构造搜索引擎这么一个例子来讲面向对象，也是颇费了一番功夫。这其中虽然涉及一些搜索引擎的专业知识和算法，但篇幅有限，也只能算是抛砖引玉，你若有所收获，我便欣然满足。

思考题

最后给你留一道思考题。私有变量能被继承吗？如果不能，你想继承应该怎么去做呢？欢迎留言与我分享、讨论，也欢迎你把这篇文章分享给你的同事、朋友，一起交流与进步。



上一篇　11 | 面向对象（上）：从生活中的类比说起

下一篇　13 | 搭建积木：Python 模块化

精选留言 (24)

写留言

益达
2019-06-05
11

看不懂不睡觉

展开

Danpier
2019-06-05
3

BOWInvertedIndexEngine search函数后半部分少了缩进，第40行开始

展开

Wing·三金
2019-06-05

2

思考题：子类继承父类私有变量的方法
- 通过定义 get/set 函数来间接操作私有变量
- 通过 实例名._父类名__私有变量名 来直接调用，所以事实上 python 并不直接私有变量

主要知识点...

展开

fly
2019-06-06

1

是search（）又被重载了，不是query（）

展开

程序员人生
2019-06-05

1

今天内容有点多。对于课后留题，在没有写代码测试时候，觉得是不能继承的。但是在写了以下代码后，好像私有变量是可以继承的，代码如下，请老师指点
class A():

    _privateVar = "privateVariable"...

展开

我是传奇
2019-06-05

1

图不错

展开

tux
2019-06-05

1

运行文中代码，始终在运行状态 --> 因为：query = input() 需要输入内容。 这个自己让自己很尴尬呀
def main(search_engine):

    ...
    while True:...

展开

John Si
2019-06-05

1

关于思考题,子类不能继承父类私有属性，只可透过self._Parent__varname去读取该私有属性的值，或在父类创建方法返回私有属性的值，然后子类调用父类方法去取得该私有属性的值

class Animal():...

展开

farFlight
2019-06-05

1

按照我之前的理解，Python是没有严格的private变量的。子类确实无法直接访问父类"self.__var"变量，但是可以通过"self._Superclass__var"访问。要想比较好地访问或者修改这些变量，可以像JAVA一样写getter和setter吧?
另外我有两个问题:
1. 一些手册中说python的多重继承非常混乱不可靠，尤其涉及很多重载的时候，因此需...

展开

lipan
2019-06-07

既能通过搜索引擎的例子学习Pyhton的继承方法，也能看到令人赏心悦目的图片，我蛮喜欢的

□
2019-06-07

这篇文章就值回票价了。

展开

KaitoShy
2019-06-06

```
class A(object):
    def __init__(self):
        self.__a_private_param = '我是父类私有变量'
        self.a_param = '我是父类变量'
...
```

展开

海生
2019-06-05

可不可以讲解一些开源的框架什么的，python 这边深度学习框架很多。这些框架的代码很优秀，也是语言的核心技术运用最多的地方。

KaitoShy
2019-06-05

“集成的时候，函数是如何重写的？” 是不是继承写成了集成了？

展开

SCAR
2019-06-05

思考题：
本质上来讲是父类的私有属性是可以继承的，因为python的私有只是伪私有，只是用了“改名机制”而已，你用self.__var去访问当然是会报错无此属性，如果你用改名机制的原则在原来属性前面加上一个下划线加父类名是可以访问的到的。看例子：
class A:...

展开

tux
2019-06-05

运行文中代码，始终在运行状态!

修改了一处：
class SearchEngineBase(object):
    ......

展开

lllong33
2019-06-05

收藏，值得反复阅读。
问题解答：私有变量和方法可以通过 obj._className__field 访问。
继承可以通过
```python
```

通过访问公有方法（从父类继承）来访问父类私有属性时却形成了，子类可以继承父类...

展开

GentleCP

2019-06-05

私有变量只能被该类的函数调用，不能被子类继承，子类如果想访问私有变量需在父类设定一个方法用于返回私有变量类似于get。如果变量只想被类本身和子类访问可以定义保护变量（以单下划线开头）

展开

小智e

2019-06-05

这个太棒了，如果能结合设计模式讲一讲，就更好了

展开

夜路破晓

2019-06-05

大地母亲在忽悠着你！
结合上篇多看几遍。代码是其次的，感觉更重要的是思路。