缓存是一种将定量数据加以保存以备迎合后续请求的处理方式，旨在加快数据的检索速度。在今天的文章中，我们将一同从简单示例出发，了解如何使用缓存机制。在此之后，我们将进一步利用Python标准库的functools模块创建适合自己需要的缓存。作为起步工作，我们首先创建一个类，用于构建我们的缓存字典，而后根据需要进行扩展。以下为具体代码：



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | ########################################################################  class MyCache:      """"""      #----------------------------------------------------------------------      def \_\_init\_\_(self):          """Constructor"""          self.cache = {}          self.max\_cache\_size = 10 |



在以上类示例中没有包含什么特别之处。我们只是创建一个简单类，同时设置两个类变量或者说属性，即cahce与max\_cache\_size。其中cache属于一套空字典，而max\_cache\_size显然代表着最大缓存容量。下面让我们进一步充实该代码，使其具备一定功能：



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45 | import datetime  import random  ########################################################################  class MyCache:      """"""      #----------------------------------------------------------------------      def \_\_init\_\_(self):          """Constructor"""          self.cache = {}          self.max\_cache\_size = 10      #----------------------------------------------------------------------      def \_\_contains\_\_(self, key):          """          根据该键是否存在于缓存当中返回True或者False          """          return key in self.cache      #----------------------------------------------------------------------      def update(self, key, value):          """          更新该缓存字典，您可选择性删除最早条目          """          if key not in self.cache and len(self.cache) >= self.max\_cache\_size:              self.remove\_oldest()          self.cache[key] = {'date\_accessed': datetime.datetime.now(),                             'value': value}      #----------------------------------------------------------------------      def remove\_oldest(self):          """          删除具备最早访问日期的输入数据          """          oldest\_entry = None          for key in self.cache:              if oldest\_entry == None:                  oldest\_entry = key              elif self.cache[key]['date\_accessed'] < self.cache[oldest\_entry][                  'date\_accessed']:                  oldest\_entry = key          self.cache.pop(oldest\_entry)      #----------------------------------------------------------------------      @property      def size(self):          """          返回缓存容量大小          """          return len(self.cache) |



在这里，我们导入了datetime与random模块，而后我们即可看到之前创建完成的类。这一次，我们向其中添加几种方法。其中一种方法具备神奇的效果，名为\_contains\_。虽然在这里并不一定要使用该方法，但其基本思路在于允许我们检查该类实例，从而了解其中是否包含有我们正在寻找的键。另外，update方法负责利用新的键/值对进行缓存字典更新。一旦达到或者超出缓存最大容量，其还会删除日期最早的输入数据。另外，remove\_oldest方法负责具体的字典内早期数据删除工作。最后，我们还引入了名为size的属性，其能够返回缓存的具体容量。

在添加了以下代码之后，我们就能够测试该缓存是否按预期起效：

1. if \_\_name\_\_ == ‘\_\_main\_\_’:
2. #测试缓存
3. keys = [‘test’, ‘red’, ‘fox’, ‘fence’, ‘junk’,
4. ‘other’, ‘alpha’, ‘bravo’, ‘cal’, ‘devo’,
5. ‘ele’]
6. s = ‘abcdefghijklmnop’
7. cache = MyCache()
8. for i, key in enumerate(keys):
9. if key in cache:
10. continue
11. else:
12. value = ”.join([random.choice(s) for i in range(20)])
13. cache.update(key, value)
14. print(“#%s iterations, #%s cached entries” % (i+1, cache.size))
15. print

在本示例当中，我们设置了大量预定义键与循环。如果键尚不存在，我们会将其添加到缓存当中。不过以上示例代码并没有提到如何更新访问日期，感兴趣的朋友们可以将其作为练习自行探索。在运行这段代码之后，大家会注意到当缓存被占满时，其会正确删除时间更早的条目。

现在，我们继续前进，看看如何利用另一种方式使用Python的内置functools模块创建缓存。

使用functools.lru\_cache

Python的functools模块提供一种非常实用的装饰器，即lru\_cache。需要注意的是，其在3.2版本当中才被添加进来。根据说明文档所言，该装饰器能够“利用可调用内存对函数进行打包，从而削减最近调用的最大尺寸。”接下来，我们将根据说明文档中提到的示例编写一项基本功能，其中包含多个网络页面。在这种情况下，我们可以直接从Python说明文档站点处获取页面。



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | import urllib.error  import urllib.request  from functools import lru\_cache  @lru\_cache(maxsize=24)  def get\_webpage(module):      """      获取特定Python模块网络页面      """      webpage = "https://docs.python.org/3/library/{}.html".format(module)      try:          with urllib.request.urlopen(webpage) as request:              return request.read()      except urllib.error.HTTPError:          return None  if \_\_name\_\_ == '\_\_main\_\_':      modules = ['functools', 'collections', 'os', 'sys']      for module in modules:          page = get\_webpage(module)          if page:              print("{} module page found".format(module)) |



在以上代码当中，我们利用lru\_cache对get\_webpage函数进行了装饰，并将其最大尺寸设置为24条调用。在此之后，我们设置了一条网页字符串变量，并将其传递至我们希望函数获取的模块当中。根据我的个人经验，如果大家将其运行在某种Python解释器当中——例如IDLE——那么效果会更好。如此一来，我们就能够针对该函数运行多次循环。可以看到在首次运行上述代码时，输出结果的显示速度相对比较慢。但如果大家在同一会话中再次加以运行，那么其显示速度将极大加快——这意味着lru\_cache已经正确对该调用进行了缓存处理。大家可以在自己的解释器实例当中进行试验并亲自查看结果。

另外，我们还可以将一条typed参数传递至该装饰器。其属于一条Boolean，旨在通知该装饰器在typed为设定为True时对不同类型参数进行分别缓存。

总结

现在大家已经初步了解了如何利用Python编写自己的缓存机制。这是一款有趣的工具，而且能够在各位面对大量高强度I/O调用或者希望对登录凭证等常用信息进行缓存时发挥重要作用。

--------------------- 本文来自 alextongtong 的CSDN 博客 ，全文地址请点击：https://blog.csdn.net/wangzhaotongalex/article/details/50864355?utm\_source=copy