15 和面试官畅谈内建对象背后的算法思想-慕课专栏

笔者曾多年负责 *Python* 开发工程师的面试工作，与 *Python* 相关的面试内容一般这样开始：

日常开发中用过 Python 内置容器对象吗？都在哪些场景，用过哪些容器，解决什么问题？

这几个问题先从 *Python* 内建对象基本用法入手，考察候选人对 *Python* 基础知识的熟悉程度。如果一个以 *Python* 为主要工具的候选人，经过引导后仍然无法准确描述 *list* 、 *dict* 等对象的用法，基本就可以放弃了。

如果候选人表示对 *list* 对象比较了解，则会进一步讨论 *list* 的关键操作以及时间复杂度：

***list* 对象都支持哪些操作？时间复杂度分别是多少？可以在头部插入吗？头部插入有什么需要注意的地方？**

这时候选人需要准确回答 *list* 对象，相应的 **时间复杂度** 以及其中缘由：

- *append* ，从尾部追加，直接将元素添加到动态数组尾部，时间复杂度为 O(1)O(1)O(1) ，必要时 *Python* 负责扩容；
- *insert* ，在指定位置插入，由于需要挪动插入位置以后的所有元素，时间复杂度为 O(N)O(N)O(N) 。

如果候选人无法准确回答 *append* 等关键操作的 **时间复杂度** ，面试官心里可能就会犯嘀咕：对方多半是一位 *API* 调用侠吧？还没搞清楚一个操作的前因后果就写代码，无异于瞎写！指不定这样的代码里藏有多少不知名炸弹！

不少 **初级工程师** 觉得开发中很少用到数据结构和算法，因此没有学习动力。这种想法是非常不可取的。在实际工程中我们完全可以站在别人的肩膀上，但原理掌握与否决定了我们到底能不能站稳，能不能用好。

与资深工程师相比，初级工程师更像是将需求 **翻译** 成代码，而且是直译，这种代码质量可想而知。因此，作为通往高级工程师进阶之路上必不可少的台阶，数据结构和算法等基础知识不应该被忽视。

如果候选人回答还算顺利，接着将讨论 *list* 如何进行容量管理以及扩缩容策略等高级知识点：

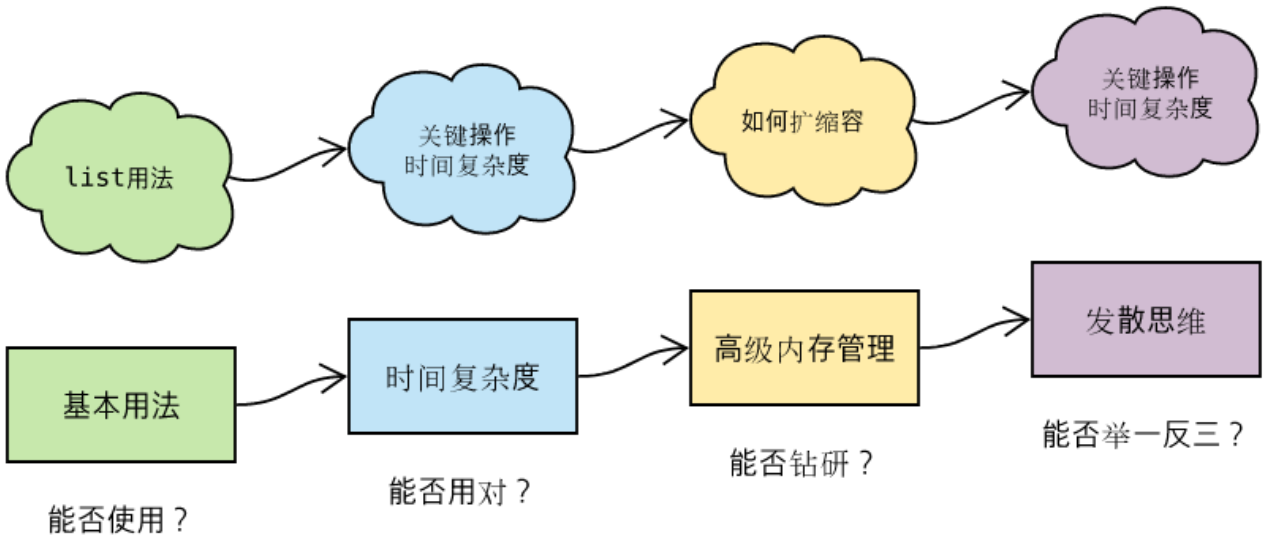
list 如何做到容量自适应？Python 在什么情况下会对底层数组进行扩缩容？

如果候选人能够准确回答《*list*源码解析：动态数组精讲》中介绍的内容，面试官应该就很满意了。再接着，面试官可能会尝试将问题发散，考察候选人的知识面以及对知识点触类旁通的能力：

有个场景需要在尾部追加，在头部删除，用 *list* 对象合适吗？不合适的话有什么替代品？

这个场景需要特别注意 *list* 对象头部操作，不管插入还是删除，时间复杂度都是 O(N)O(N)O(N)，非常不理想。这时，标准库 *collections* 模块中的双端队列 *deque* 更好，头部操作时间复杂度跟尾部操作一样，都是 O(1)O(1)O(1)。如果候选人能够一路打怪升级走到这里，基本上就可以拿到通过面试的门票了。

一般面试过程都是这样，从基础的使用方法开始，逐步深入底层原理，再进一步发散、触类旁通。通过阶梯式面试过程，面试官可以快速评估候选人的能力水平，并据此决定取舍。



退一步讲，就算为了在面试中走得更远，候选人也必须有深入底层的意识，而源码学习是夯实基础的最好途径之一。

list

***list* 对象常用操作有哪些？时间复杂度分别是多少？**

操作	示例	时间复杂度
尾部追加	l.append(x)	O(1)
尾部删除	x = l.pop()	O(1)
头部插入	l.insert(0, x)	O(N)
头部删除	x = l.pop(0)	O(N)
指定位置插入	l.insert(i, x)	O(N)
指定位置删除	x = l.pop(i)	O(N)
清空列表	l.clear()	O(N)
浅拷贝	l.copy()	O(N)

操作	示例	时间复杂度
元素计数	l.count(x)	O(N)
列表扩展	l.extend(l2)	O(N)
元素查找	l.index(x)	O(N)
元素删除	l.remove(x)	O(N)
列表反转	l.reverse()	O(N)
列表排序	l.sort()	O(logN)

常用操作以及对应的时间复杂度是最低门槛，描述必须做到准确无误。如被问及诸如为什么 **尾部插入效率比头部高** 之类的原理性问题，则需要结合列表对象内部 **动态数组** 结构进行回答。

***list* 为什么可以做到容量自适应？什么时机需要扩容、缩容？**

list 对象底层由动态数组实现，对象头部保存数组 **容量** 以及当前已用 **长度** 。

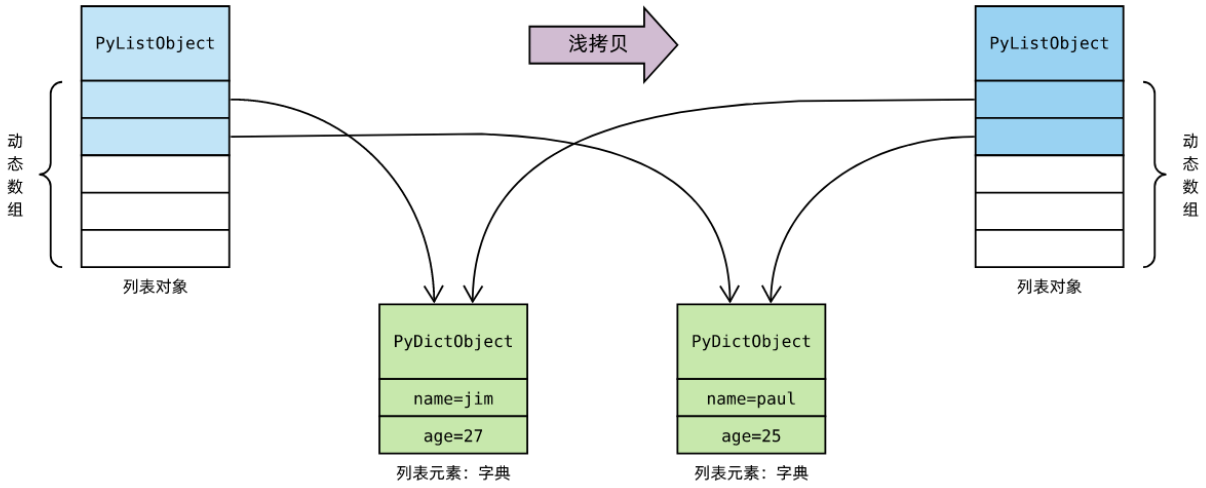
当我们往列表添加新数据时，长度会不断增长。当长度达到容量后， *Python* 会对底层数组进行扩容，分配一个更大的数组，并将元素从旧数组中拷贝过去。为避免频繁扩容，*Python* 每次扩容时都额外分配至少 18\frac{1}{8}81 的空闲空间。

当我们从列表中删除元素时，动态数组慢慢出现很多空闲空间。这时 *Python* 对底层数组进行缩容，以降低内存开销。缩容操作与扩容类似，需要重新分配底层数组，更多细节请复习《*list*源码解析：动态数组精讲》一节。

通过 *copy* 方法复制 *list* ，修改新列表会影响旧列表吗？

list 对象 *copy* 方法实现了浅拷贝，只拷贝列表本身，不拷贝列表中存储的元素对象。

```
>>> users = [{'name': 'jim', 'age': 27}, {'name': 'paul', 'age': 25}]
>>> users2 = users.copy()
```



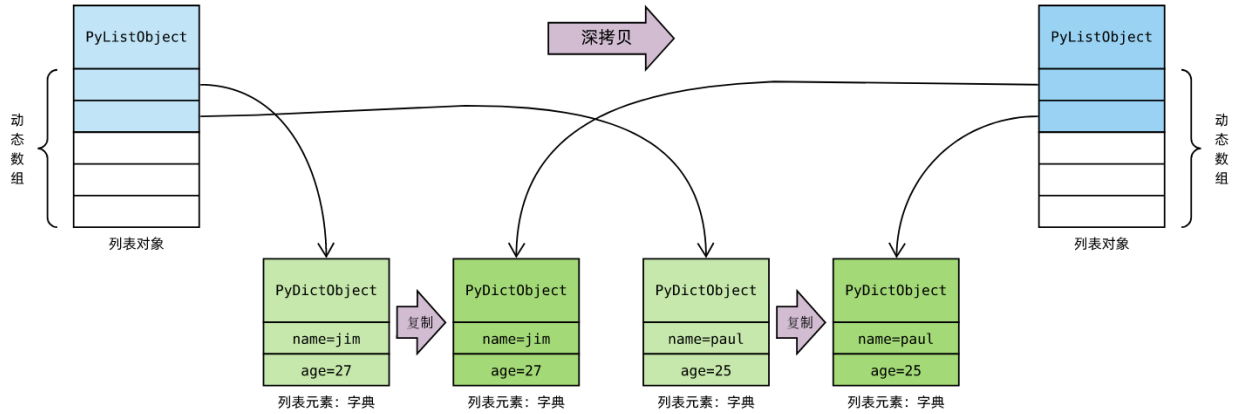
这样一来，新旧列表中的元素是同一个。对新列表中的元素进行修改，必然在旧列表中可见：

```
>>> users2[0]['age'] += 1
>>>
>>> users2
[{'name': 'jim', 'age': 28}, {'name': 'paul', 'age': 25}]
>>> users
[{'name': 'jim', 'age': 28}, {'name': 'paul', 'age': 25}]
```

如果 浅拷贝 不是你想要的行为，可以通过 *copy* 模块中的 *deepcopy* 函数进行 深拷贝 ：

```
>>> from copy import deepcopy
>>> users = [{'name': 'jim', 'age': 27}, {'name': 'paul', 'age': 25}]
>>> users2 = deepcopy(users)
```

与浅拷贝不同，深拷贝不仅负责列表对象，还递归复制列表中存储的每个元素对象：



这样一来，新旧列表就完全独立了。对其中一个的元素元素对象进行修改，不会影响另一个：

```
>>> users2[0]['age'] += 1
>>> users2
[{'name': 'jim', 'age': 28}, {'name': 'paul', 'age': 25}]
>>> users
[{'name': 'jim', 'age': 27}, {'name': 'paul', 'age': 25}]
```

Python 中有“栈”容器吗？如何快速得到一个栈？

list 列表对象是一种 **动态数组** 式容器，类似 *C++* 中的 *vector* 。 *list* 对象具有优秀的尾部操作效率，不管是向尾部追加还是从尾部删除，时间复杂度都是 O(1)O(1)O(1) 。因此，我们可以将 *list* 对象当做一个栈来使用：

```
>>> stack = []


>>> stack.append(1)
>>> stack.append(2)
>>> stack.append(3)


>>> stack[0]


>>> stack[-1]


>>> len(stack)



>>> stack.pop()
3
>>> stack.pop()
2
```

deque

频繁从 *list* 头部删除元素会导致什么问题？如何解决？

由于在 *list* 头部增删元素需要挪动其后所有元素，时间复杂度是 O(N)O(N)O(N)，效率堪忧。因此，我们需要极力避免这类操作。如果实际场景无法避免头部操作，可以考虑用 *collections* 模块中的 *deque* 双端队列。顾名思义， *deque* 也是一种线性容器，头尾两端操作效率都很高，时间复杂度是 O(1)O(1)O(1)。

```
>>> from collections import deque
>>> q = deque()


>>> q.append(1)
>>> q.append(2)
>>> q.append(3)


>>> len(q)
3

>>> q[0]
1

>>> q[-1]
3



>>> q.popleft()
1
>>> q.popleft()
2
```

dict

dict 对象常用操作有哪些？时间复杂度分别是多少？

操作	示例	时间复杂度
键值设置	d[k] = v	O(1)
键值设置（默认值）	d.setdefault(k, v)	O(1)
键值查找	v = d[k]	O(1)
键值查找（默认值）	v = d.get(k)	O(1)
键值删除	v = d.pop(k)	O(1)
清空字典	d.clear()	O(N)

操作	示例	时间复杂度
浅复制	d.copy()	O(N)

空 *dict* 对象是否占用内存？占用多少内存？为什么？

dict 对象内部包含一个哈希表，用于快速定位键值对。就算 *dict* 为空， *Python* 也会为其分配哈希表，最小的哈希表长度为 *8* 。按照《dict 对象，高效的关联式容器》中介绍过的算法，空 *dict* 对象内存刚好是 *240* 字节。

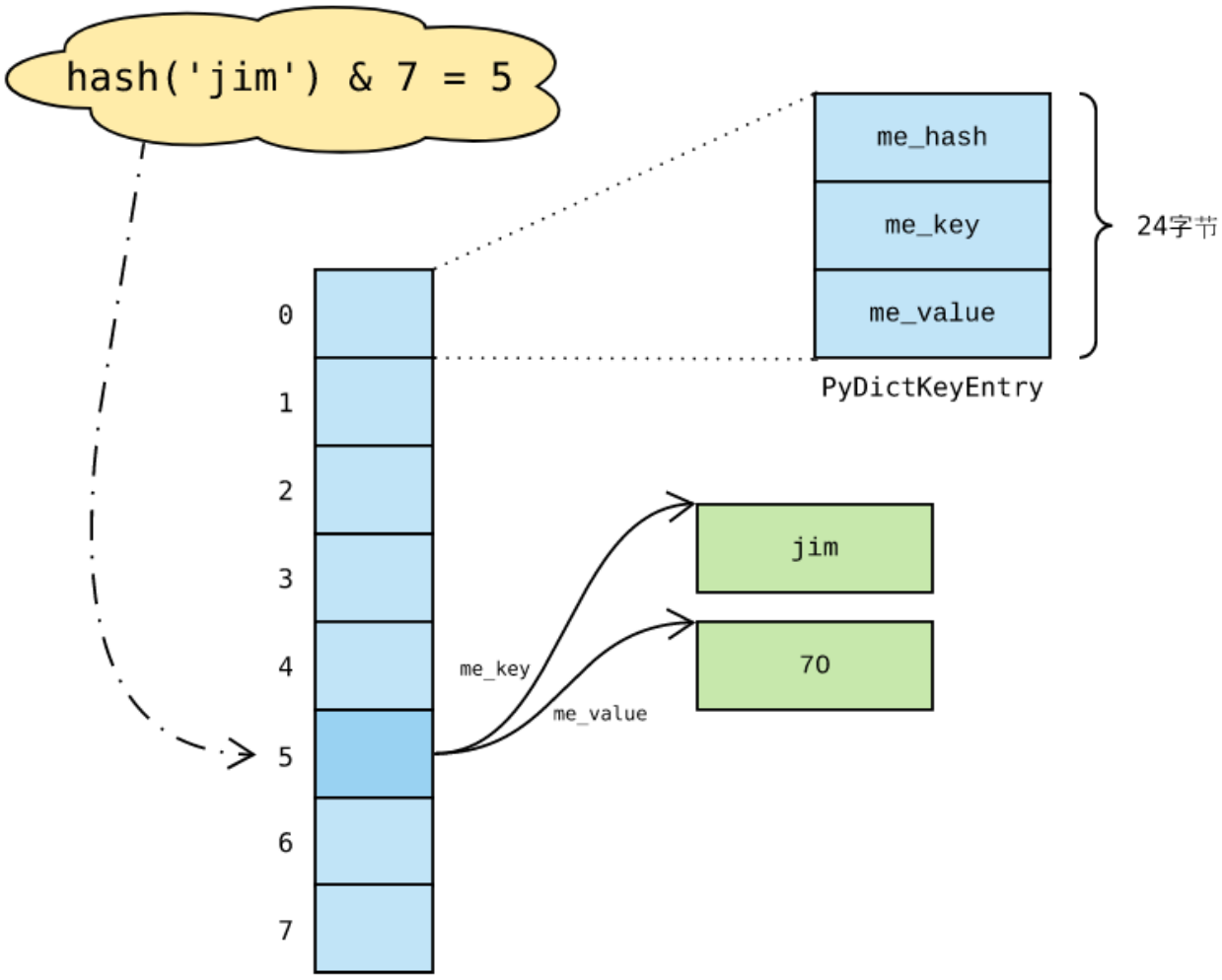
```
>>> import sys
>>> sys.getsizeof({})
240
```

***dict* 内部的哈希表为何分为两个数组来实现？**

答案是 **节约内存** 。

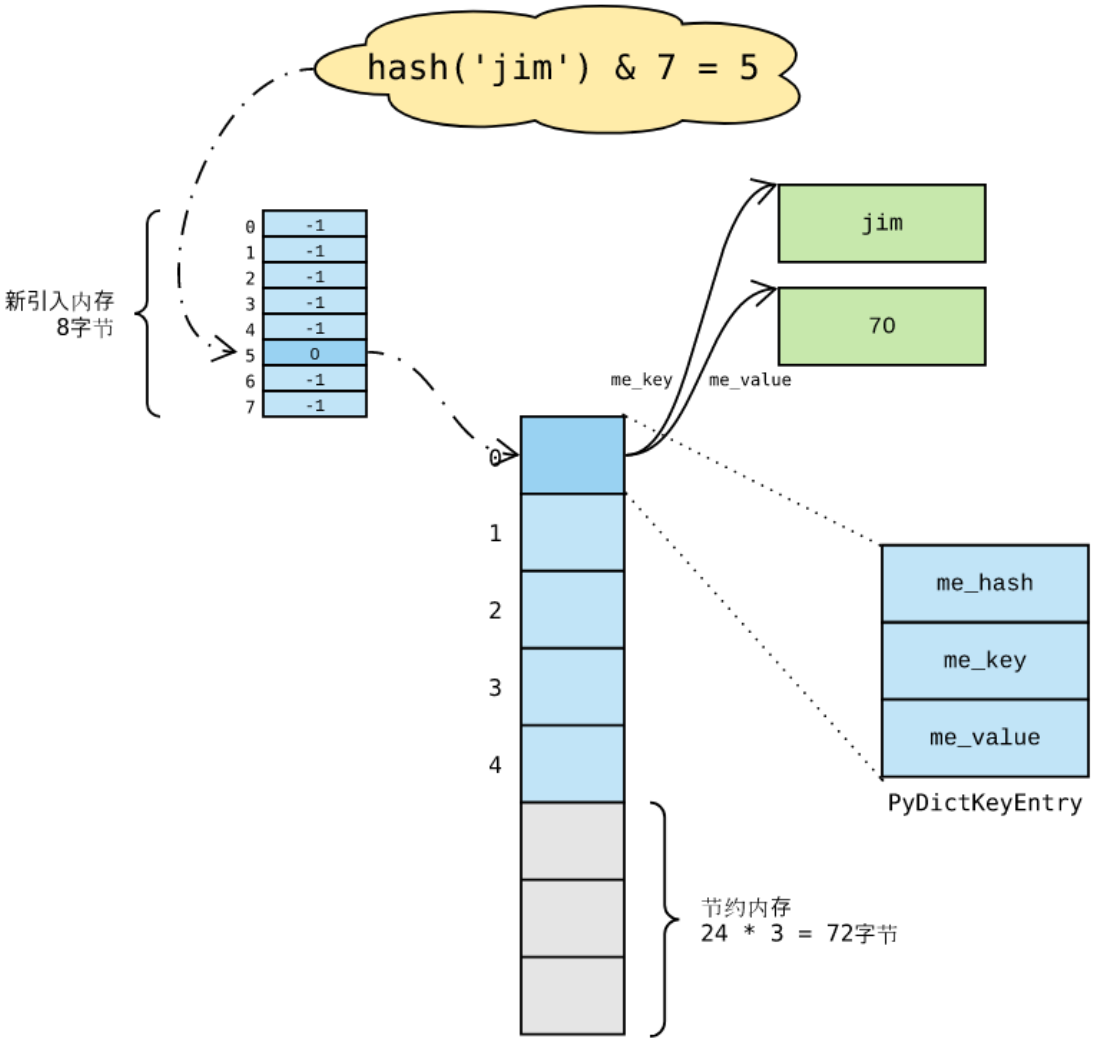
dict 用于保存 **键值对** ，键值对在 *Python* 内部由 *PyDictKeyEntry* 结构体表示，大小为 *24* 字节。为控制 **哈希冲突** 频率，*Python* 只使用哈希表中不超过 23\frac{2}{3}32 的条目。因此，哈希表必然是 **稀疏** 的，至少 13\frac{1}{3}31 的条目是浪费的。如果使用 *PyDictKeyEntry* 作为哈希表条目，将浪费很多内存。

如下图，哈希表规模为 *8* ，而 Python 最多只能使用其中的 8∗23=58 * \frac{2}{3} = 58∗32=5 个，浪费其中 *3* 个。我们知道 *3* 个 *PyDictKeyEntry* 结构体需要 3 x 24 = 72 字节的内存，哈希表规模越大浪费内存越多！



为了优化内存使用， *Python* 将条目存储从哈希表中剥离，只分配相当于哈希表规模 23\frac{2}{3}32 的存储条目。这样一来，需要另一个数组来承担哈希表的角色。哈希表只需存储条目下标，因此使用整数类型即可。整数类型可以根据哈希表规模，选择位数最少的。

如下图，存储条目独立后，节约了 *72* 字节的内存。除去哈希表新引入的 *8* 字节，总体上节约了 *64* 字节内存：



你或许觉得 *72* 字节内存无伤大雅，但由于 *Python* 运行时存在大量空 *dict* 对象，乘数效应的威力便非常明显！更何况，当 *dict* 对象长度很大时，节约的内存将非常可观！