聊一聊字节跳动的面试

一、算法题

一面：

1. lc 里最长上升子序列的变形题

第一种解法：

一、动态规划

观察数组，要求数组的最长上升子序列，可以想到长度为 i 的子序列可以由长度为 i - 1 的递推而来，因此考虑动态规划。那么动态规划的**状态空间**该如何定义呢？

思考最长上升子序列的寻找过程，当我们遍历到数组的 i 位置时，无法判断最终的最长子序列是由 i 结尾的所属的子序列还组成还是由 i - 1 结尾或者其他的子序列组成。因此我们需要把位置 i 之前的所有位置所在的子序列的长度都记录下来。

因此我们这样定义状态空间，dp[i]代表的是以 nums[i]结尾的子序列的最长子序列的长度。

递推公式为：dp[i] = max(dp[i][, dp[j] + 1) if nums[i] > nums [j]

最后取以每个元素结尾的子序列长度的最大值。

解决方法：

1：动态规划

```
class Solution:
    def lengthOfLIS(self, nums: List[int]) -> int:
        if not nums: return 0
        dp  = [1] * len(nums)
        for i in range(len(nums)):
            for j in range(i):
                if nums[i] > nums[j]:
                    dp[i] = max(dp[i], dp[j] + 1)
        return max(dp)
```

2：贪心+二分查找

二、贪心+二分查找

看到这个复杂度要求基本上就是要二分了，O(nlogn)的复杂度，n 给遍历数组，logn 给二分查找。

那么如何贪心呢？在哪儿查找呢？（这个方法太巧妙了）

我们再思考一下动态规划过程中，在寻找以 nums[i]为结尾的最长子序列时，当我们取到元素 i + 1 时，这时加入 dp[i]和 dp[i - 1]的值相同，那么我该选择那个序列向下递推呢？显然我们选**择结尾元素更小的那个序列更有利于子序列长度的增长**。动态规划方法没有利用这个信息，将每种子序列的最长值求出来然后取最大值。

而贪心方法充分利用了这个信息，遇到比当前序列值更大的值，直接添加到序列末尾，如果比当前序列小，那么替换掉当前序列中刚好比这个值大的数。**二分就是用来寻找被替换的位置的。**
这样保存下来的数列并不是最长的序列，但长度是最长序列的长度。因为当前序列中的每一个值代表的是，在这个位置，数值可能到达的最小值。
非常巧妙，数学证明可任意看下官方解释。

官方数学证明

```python
class Solution:
    def lengthOfLIS(self, nums: List[int]) -> int:
        d = []
        for n in nums:
            if not d or n > d[-1]:
                d.append(n)
            else:
                l, r = 0, len(d) - 1
                loc = r
                while l <= r:
                    mid = (l + r) // 2
                    if d[mid] >= n:
                        loc = mid
                        r = mid - 1
                    else:
                        l = mid + 1
                d[loc] = n
        return len(d)
```

第二种解法：

```
def len_of_longest_ascending_subsequences(nums):
        if len(nums)<=1:
            return len(nums)
        # 用来存放各个字串的最长上升子序列个数
        mem = [0 for _ in range(len(nums))]

        for j in range(1,len(nums)):
            for i in range(0,j):
                # 1<2
```

```
                if nums[i]<nums[j]:
                    # max(1,1+1)
                    #  更新各个字串的最长上升子序列个数
                    #  状态转移方程
                    mem[j] = max(mem[j],mem[i]+1)
                    # mem[1] = 2
        print(mem)

        return max(mem)
# list01=[2,-5,-6,-8,-3,3,56,6,3,6,7,100,9,3]
list01=[1,2,6,3]
print(len_of_longest_ascending_subsequences(list01))
```

第三种解法：

```
300.    给定一个无序的整数数组，找到其中最长上升子序列的长度。
301.
302.    示例:
303.
304.    输入: [10,9,2,5,3,7,101,18]
305.    输出: 4
306.    解释: 最长的上升子序列是 [2,3,7,101]，它的长度是 4。
307.    说明:
308.
309.    可能会有多种最长上升子序列的组合，你只需要输出对应的长度即可。
310.    你算法的时间复杂度应该为 O(n2) 。#动态规划
311.    进阶: 你能将算法的时间复杂度降低到 O(n log n) 吗? #二分查找
312.
313.    测试集：
314.    [4,10,4,3,8,9]
315.    [4,10,4,3,2,1]
316.    [10,9,2,5,3,7,101,18,19,20,30]
317.    []
318.    [0]
319.    [10,10,10]
320.    [1,3,6,7,9,4,10,5,6]
```

动态规划类型，第四道：O(n*n)

提交时间	状态	执行用时	内存消耗	语言
几秒前	通过	1836 ms	13.1 MB	python3

	j	0	1	2	3	4	5
i		4	10	4	3	8	9
0	4	1	1+1	1		1+1	1+1
1	10		1				
2	4			1		1+1	1+1
3	3				1	1+1	1+1
4	8					1	2+1
5	9						1
	MAX	1	2	1	1	2	3


```
class Solution:
    def lengthOfLIS(self, nums):
        if not nums: return 0
```

```
        n = len(nums)
        memo = [1]*n

        for j in range(1, n):
            for i in range(j):
                if nums[j] > nums[i]:
                    memo[j] = max(memo[i]+1,memo[j])

        return max(memo)
```

【优化】使用二分查找：O(nlogn)

python 中使用 bisect 二分查找插入位置

提交时间	状态	执行用时	内存消耗	语言
几秒前	通过	64 ms	13.1 MB	python3

```
class Solution:
    def lengthOfLIS(self, nums):
                lst = []
                for num in nums:
                        p = bisect.bisect_left(lst, num)
                        #找到此数字可能插入的位置

                        if p == len(lst):
                                lst.append(num)
                        #66 的 position 是 0 == len([]) or 77 的 position3 == len([0,10,20])
                        #[66] [0,10,20,77]
```

```
                else:
                        lst[p] = num
                #9 的 postion 是 1 != len([0,10,20])
                #[0,9,20]
        return len(lst)
```

举例 nums = [4,10,4,5,2,1]
num = 4, lst = [4]
num = 10, lst = [4, 10]
num = 4, lst = [4, 10]
num = 5, lst = [4, 5]
num = 2, lst = [2, 5]
num = 1, lst = [1, 5]
不难看出，题目要求是长度，所以无碍。

2. 实现输入英文单词联想的功能

python_根据"词库"进行“词联想”

输入法中，当你输入一个字的时候，输入法就能猜出你要输入什么词。这就是词联想。现在，再 python 中简单实现类似这样的功能：根据制定好的词库，输入一个新的词，帮助实现词联想。其中分词用了 jieba 包。

```
# -*-coding:utf-8-*-



"""
    分词的函数 cut_words()
"""
def cut_words(temp):
    import jieba
    import re
    wenben = re.sub("[1234567890\s+\.\!\/_; : ,$%^*(+\"\']+|[+——！ ， 。 ？ 、 ~@#￥%……&*（ ）]+", "", temp)

    seg_list = jieba.cut(wenben, cut_all=True)  # 全模式分词
    # seg_list = jieba.cut(wenben,cut_all=False)#精确模式分词
    # seg_list = jieba.cut_for_search(wenben) #搜索引擎模式
    lists = []
    for item in seg_list:
```

```
        lists.append(item)

    return lists #分词结果返回到一个列表

"""
    统计词频的函数 get_counts1(),参数 lists 为切好的词的列表
"""
def get_counts1(lists):
    counts = {}
    for item in lists:
        if item in counts:
            counts[item] += 1
        else:
            counts[item] = 1
    return counts


"""
    方法 2：统计词频的函数 get_counts2,参数 lists 为切好的词如下：

"""
from collections import defaultdict
def get_counts2(lists):
    counts = defaultdict(int)
    for item in lists:
        counts[item] += 1
    return counts


"""
    统计词频的方法 3：使用 collection 的 Counter 方法,参数 lists 为切好的词
"""


```

```
def get_counts3(lists):
    from collections import Counter
    counter = Counter(lists)
    print(counter.most_common(10))


"""
    按照词频排序的函数 get_top_counts()
"""


def get_top_counts(counts):
    value_keys = sorted([(count, tz) for tz, count in counts.items()], reverse=True)
    result = {}
    for item in value_keys:
        result[item[1]] = item[0]
    return value_keys
    # return result

"""
 获取原始文本的所有汉字的联想词。函数的参数为原始文本。
"""


def get_lianxiang(temp):
    import re
    wenzhang = re.sub("[1234567890\s+\.\!\/_; : ,$%^*(+\"\']+|[+——！ ， 。？ 、 ~@#￥%……&*（）]+", "", temp)
    a = cut_words(wenzhang)  # 分词
    b = get_counts1(a)  # 词频统计
    data = {}
    for i in range(len(wenzhang)):
        data[wenzhang[i]] = {}
        for key in get_counts1(a):
            try:
```

```
                if wenzhang[i] == key[0]:
                    data[wenzhang[i]][key] = b[key]
            except:
                pass
    print(data)

"""
    词联想的函数，输入的参数为输入的词，然后根据词库，按照词频高低，
    输出你想输入的词组，即词联想。词库为《明朝那些事》的部分章节。
"""
def get_lianxiang1():
    import re
    word = input("请出入您想联想的词：")
    print("请稍等...")
    wenzhang = re.sub("[1234567890\s+\.\!\/_; : ,$%^*(+\"\']+|[+——！ ， 。 ？ 、 ~@#￥%……&* （ ） ]+", "", temp)
    a = cut_words(wenzhang)  # 分词
    b = get_counts1(a)  # 词频统计
    data = {}
    for i in range(len(wenzhang)):
        data[wenzhang[i]] = {}
        for key in get_counts1(a):
            try:
                if wenzhang[i] == key[0]:
                    data[wenzhang[i]][key] = b[key]
            except:
                pass
    if word in data:
        dic=data[word]
        a=get_top_counts(dic)
        b=[]
        for item in a:
            b.append(item[1])
        # print(a)
        print("为您联想的几个词为：",b)
```

```
120         # print("联想结果如下:\n",data[word])
121     else:
122         print("没有根据本篇文章，您输入的词的联想词！")
123         get_lianxiang1()
124
125
126 if __name__ == '__main__':
127     temp = "之前我们曾经介绍过，朱棣曾派出两路人去寻找建文帝，一路是胡濙，他的事情我们已经讲过了，这位胡濙的生平很多人都不熟悉，这也不奇怪，因为他从事是秘密工作，大肆宣传是不好的。但另一路人马的际遇却大不相同，不但闻名于当时，还名留青史，千古流芳。这就是鼎鼎大名的郑和舰队和他们七下西洋的壮举同样是执行秘密使命，境遇却如此不同，我们不禁要问：同样是人，差距怎么那么大呢？原因很多，如队伍规模、附带使命等等，但在我看来，能成就如此壮举，最大的功劳应当归于这支舰队的指挥者——伟大的郑和。伟大这个词用在郑和身上是绝对不过分的，他不是皇室宗亲，也没有显赫的家世，但他以自己的努力和智慧成就了一段传奇——中国人的海上传奇，在郑和之前历史上有过无数的王侯将相，在他之后还会有很多，但郑和只有一个。下面就让我们来介绍这位伟大航海家波澜壮阔的一生。郑和，洪武四年（1371）出生，原名马三保，云南人，自小聪明好学，更为难得的是，他从小就对航海有着浓厚的兴趣，按说在当时的中国，航海并不是什么热门学科，而且云南也不是出海之地，为什么郑和会喜欢航海呢？这是因为郑和是一名虔诚的伊斯兰教徒，他的祖父和父亲都信奉伊斯兰教，而所有的伊斯兰教徒心底都有着一个最大的愿望去圣城麦加朝圣。去麦加朝圣是全世界伊斯兰教徒的最大愿望，居住在麦加的教徒们是幸运的，因为他们可以时刻仰望圣地，但对于当时的郑和来说，这实在是一件极为不易的事情。麦加就在今天的沙特阿拉伯境内，有兴趣的朋友可以在地图上把麦加和云南连起来，再乘以比例尺，就知道有多远了。不过好在他的家庭经济条件并不差，他的祖父和父亲都曾经去过麦加，在郑和小时候，他的父亲经常对他讲述那朝圣途中破浪远航、跋山涉水的惊险经历和万里之外、异国他乡的奇人异事。这些都深深的影响了郑和。也正是因此，幼年的郑和与他同龄的那些孩子并不一样，他没有坐在书桌前日复一日的背诵圣贤之言，以求将来图个功名，而是努力锻炼身体，学习与航海有关的知识，因为在他的心中，有着这样一个信念：有朝一日，必定乘风破浪，朝圣麦加。如果他的一生就这么发展下去，也许在十余年后，他就能实现自己的愿望，完成一个平凡的伊斯兰教徒的夙愿，然后平凡地生活下去可是某些人注定是不会平凡地度过一生的，伟大的使命和事业似乎必定要由这些被上天选中的人去完成，即使有时是以十分残忍的方式。洪武十四年（1381），傅友德、蓝玉奉朱元璋之命令，远征云南，明军势如破竹，仅用了半年时间就平定了云南全境，正是这次远征改变了郑和的命运。顺便提一句，在这次战役中，明军中的一名将领戚祥阵亡，他的牺牲为自己的家族换来了世袭武职，改变了自己家族的命运，从此他的子孙代代习武。这位戚祥只是个无名之辈，之所以这里要特意提到他，是因为他有一个十分争气的后代子孙戚继光。历史真是让人难以捉摸啊。对于明朝政府和朱元璋来说，这不过是无数次远征中的一次，但对于郑和而言，这次远征是他人生的转折，痛苦而未知的转折。战后，很多儿童成为了战俘，按说战俘就战俘吧，拉去干苦力也就是了，可当时对待儿童战俘有一个极为残忍的惯例阉割。这种惯例的目的不言而喻，也实在让人不忍多说，而年仅 11 岁的马三保正是这些不幸孩子中的一员。我们不难想象当年马三保的痛苦，无数的梦想似乎都已经离他而去了，但历史已经无数次地告诉我们，悲剧的开端，往往也是荣耀的起点。悲剧，还是荣耀，只取决于你，取决于你是否坚强。从此，这个年仅十一岁的少年开始跟随明军征战四方，北方的风雪、大漠的黄沙，处处都留下了他的痕迹，以他的年龄，本应在家玩耍、嬉戏，却突然变成了战争中的一员，在那血流成河，尸横遍野的战场上飞奔。刀剑和长枪代替了木马和玩偶，在军营里，没有人会把他当孩子看，也不会有人去照顾和看护他，在战争中，谁也不能保证明天还能活下来，所以唯一可以照顾他的就是他自己。可是一个十一岁的孩子怎么能照顾自己呢？我们无法想象当年的马三保吃过多少苦，受过多少累，多少次死里逃生，我们知道的是，悲惨的遭遇并没有磨灭他心中的
```

希望和信念，他顽强地活了下来，并最后成为了伟大的郑和。总结历史上的名人（如朱元璋等）的童年经历，我们可以断言：小时候多吃点苦头，实在不是一件坏事。在度过五年颠沛流离的生活后，他遇到了一个影响他一生的人，这个人就是朱棣当时的朱棣还是燕王，他一眼就看中了这个沉默寡言却又目光坚毅的少年，并挑选他做了自己的贴身侍卫，从此马三保就跟随朱棣左右，成为了他的亲信。金子到哪里都是会发光的，马三保是个注定要成就大事业的人，在之后的靖难之战中，他跟随朱棣出生入死，立下大功，我们之前介绍过，在郑村坝之战中，朱棣正是采用他的计策，连破李景隆七营，大败南军。朱棣从此也重新认识了这个贴身侍卫，永乐元年（1403），　朱棣登基后，立刻封马三保为内官监太监，这已经是内官的最高官职，永乐二年（1404），朱棣又给予他更大的荣耀，赐姓郑，之后，他便改名为郑和，这个名字注定要光耀史册。要知道，皇帝赐姓是明代至高无上的荣耀，后来的郑成功被皇帝赐姓后，便将之作为自己一生中的最大光荣，他的手下也称呼他为国姓爷，可见朱棣对郑和的评价之高。上天要你受苦，往往会回报更多给你，这也是屡见不鲜的，郑和受到了朱棣的重用，成为了朝廷中炙手可热的人物，作为朱棣的臣子，他已经得到了很多别人想都不敢想的荣耀，想来当年的郑和应该也知足了。但命运似乎一定要让他成为传奇人物，要让他流芳千古。更大的使命和光荣将会降临到他的头上，更大的事业将等待他去开创。朱棣安排郑和出海是有着深层次目的的，除了寻找建文帝外，郑和还肩负着威服四海，胸怀远人的使命，这大致也可以算是中国历史上的老传统，但凡强盛的朝代，必定会有这样的一些举动，如汉朝时候贯通东西的丝绸之路，唐朝时众多发展中国家及不发达国家留学生来到我国学习先进的科学文化技术，都是这一传统的表现。中国强盛，万国景仰，这大概就是历来皇帝们最大的梦想吧，历史上的中国并没有太多的领土要求，这是因为我们一向都很自负，天朝上国，万物丰盛，何必去抢人家的破衣烂衫？但正如俗话所说，锋芒自有毕现之日，强盛于东方之中国的光辉是无法掩盖的，当它的先进和文明为世界所公认之时，威服四海的时刻自然也就到来了。实话实说，在中国强盛之时，虽然也因其势力的扩大与外国发生过领土争端和战争（如唐与阿拉伯之战），也曾发动过对近邻国家的战争（如征高丽之战），但总体而言，中国的外交政策还是比较开明的，我们慷慨的给予外来者帮助，并将中华民族的先进科学文化成就传播到世界各地，四大发明就是最大的例证。综合来看，我们可以用四个字来形容中国胸怀远人的传统和宗旨：以德服人。现在中国又成为了一个强盛的国家，经过长期的战乱和恢复，以及几位堪称劳动模范的皇帝的辛勤耕耘和工作，此时的华夏大地已经成为了真正的太平盛世，人民安居乐业，国家粮银充足，是该做点什么的时候了。在我们这个庞大国家的四周到底还有些什么？这是每一个强盛的朝代都很感兴趣的一个问题，明帝国就是一个强盛的朝代，而明帝国四周的陆地区域已由汉唐盛世时的远征英雄们探明，相比而言，帝国那漫长的海岸线更容易引起人们的遐想，在宽阔大海的那一头有着怎样的世界呢？最先映入人们眼帘的就是西洋，需要说明的是西洋这个名词在明朝的意义与今日并不相同，当时的所谓西洋其实是现在的南洋，之前的朝代虽也曾派出船只远航过这些地区，但那只是比较单一的行动，并没有什么大的影响，海的那边到底有些什么，人们并不是十分清楚，而现在强大的明帝国的统治者朱棣是一个与众不同的人，他之所以被认为是历史上少有的英明君主，绝非由于仁慈或是和善，而是因为他做了很多历史上从来没有人做过的事情。现在，朱棣将把一件历史上从来没有人做过的事情交给郑和来完成，这是光荣，也是重托。无论从哪个角度来看，郑和都是最合适的人选，他不但具有丰富的航海知识，还久经战争考验，军事素养很高，性格坚毅顽强，最后，他要去的西洋各国中有很多都信奉伊斯兰教，而郑和自己就是一个虔诚的穆斯林。　　按说这只是一次航海任务而已，何必要派郑和这样一个多样型人才去呢，然而事实证明，郑和此次远航要面对的，绝不仅仅是大海而已。历史将记住这个日子，永乐三年六月十五日（ 1405 年 7 月 11 日），郑和在福建五虎门起航，开始了中国历史上最伟大的远航征程，郑和站在船头，看着即将出发的庞大舰队和眼前的茫茫大海。他明白自己此次航程所负的使命和职责，但他并不知道，此时此刻，他正在创造一段历史，将会被后人永远传颂的历史。　　他的心中充满了兴奋，自幼年始向往的大海现在就在他的眼前，等待着他去征服！一段伟大的历程就要开始了！扬帆！我们之前曾不断用舰队这个词语来称呼郑和的船队，似乎略显夸张，一支外交兼寻人的船队怎么能被称为舰队呢，但看了下面的介绍，相信你就会认同，除了舰队外，实在没有别的词语可以形容他的这支船队。托当年一代枭雄陈友谅的服，朱元璋对造船技术十分重视，这也难怪，当年老朱在与老陈的水战中吃了不少亏，连命也差点搭进去。在他的鼓励下，明朝的造船工艺有了极大的发展，据史料记载，当时郑和的船只中最大的叫做宝船，这船到底有多大呢，大者，长四十

四丈四尺，阔一十八丈；中者，长三十七丈，阔一十五丈。大家可以自己换算一下，按照这个长度，郑和大可在航海之余举办个运动会，设置了百米跑道绝对不成问题。而这条船的帆绝非我们电视上看到的那种单帆，让人难以想象的是，它有十二张帆！它的锚和舵也都是巨无霸型的，转动的时候需要几百人喊口号一起动手才能摆得动，南京市在五十年代曾经挖掘过明代宝船制造遗址，出土过一根木杆，这根木杆长十一米，问题来了，这根木杆是船上的哪个部位呢？鉴定结论出来了，让所有的人都目瞪口呆，这根木杆不是人们预想中的桅杆，而是舵杆！　果你不明白这是个什么概念，我可以说明一下，桅杆是什么大家应该清楚，所谓舵杆只不过是船只舵叶的控制联动杆，经过推算，这根舵杆连接的舵叶高度大约为六米左右。也就是说这条船的舵叶有三层楼高！航空母舰，名副其实的航空母舰。这种宝船就是郑和舰队的主力舰，也就是我们通常所说的旗舰，此外还有专门用于运输的马船，用于作战的战船，用于运粮食的粮船和专门在各大船只之间运人的水船。　和率领的就是这样的一支舰队，舰队之名实在实至名归。这是郑和船队的情况，那么他带了多少人下西洋呢？将士卒二万七千八百余人。说句实话，从这个数字看，这支船队无论如何也不像是去寻人或是办外交的，倒是很让人怀疑是出去找碴打仗的。但事实告诉我们，这确实是一支友好的舰队，所到之处，没有战争和鲜血，只有和平和友善。强而不欺，威而不霸，这才是一个伟大国家和民族的气度与底蕴。郑和的船队向南航行，首先到达了占城，然后他们自占城南下，半个月后到达爪哇（印度尼西亚爪哇岛），此地是马六甲海峡的重要据点，但凡由马六甲海峡去非洲必经此地，在当时，这里也是一个人口稠密，物产丰富的地方，当然，当时这地方还没有统一的印度尼西亚政府。而且直到今天，我们也搞不清当时岛上的政府是由什么人组成的。郑和的船队到达此地后，本想继续南下，但一场悲剧突然发生了，船队的航程被迫停止了，而郑和将面对他的航海生涯中的第一次艰难考验。　　事情是这样的，当是统治爪哇国的有两个国王，互相之间开战，史料记载是东王和西王，至于到底是些什么人，那也是一笔糊涂账，反正是西王战胜了东王。东王战败后，国家也被灭了，西王准备秋后算账，正好此时，郑和船队经过东王的领地，西王手下的人杀红了眼，也没细看，竟然杀了船队上岸船员一百七十多人。郑和得知这个消息后，感到十分意外，手下的士兵们听说这个巴掌大的地方武装居然敢杀大明的人，十分愤怒和激动，跑到郑和面前，声泪俱下，要求就地解决那个什么西王，让他上西天去做一个名副其实的王。郑和冷静地看着围在他四周激动的下属，他明白，这些愤怒的人之所以没有动手攻打爪哇，只是因为还没有接到他的命令。那些受害的船员中有很多人郑和都见过，大家辛辛苦苦跟随他下西洋，是为了完成使命，并不是来送命的，他们的无辜被杀郑和也很气愤，他完全有理由去攻打这位所谓的“西王”，而且毫无疑问，这是一场毫无悬念的战争，自己的军队装备了火炮和火枪等先进武器，而对手不过是当地的一些土著而已，只要他一声令下，自己的舰队将轻易获得胜利，并为死难的船员们报仇雪恨。但他没有下达这样的命令。他镇定地看着那些跃跃欲试的下属，告诉他们，决不能开战，因为我们负有更大的使命。和平的使命。如果我们现在开战，自然可以取得胜利，但那样就会偏离我们下西洋的原意，也会耽误我们的行程，更严重的是，打败爪哇的消息传到西洋各地，各国就会怀疑我们的来意，我们的使命就真的无法达成了。″

```
130. 128      get_lianxiang1()  # 调用获取联想词的函数。
```

二面：

1. **矩阵旋转，要求空间复杂度 O(1)**

   给定一个 *n* × *n* 的二维矩阵表示一个图像。

   将图像顺时针旋转 90 度。

   说明：

   你必须在原地旋转图像，这意味着你需要直接修改输入的二维矩阵。**请不要**使用另一个矩阵来旋转图像。

   示例 1:

   给定 **matrix** =

   ```
   [
     [1,2,3],
     [4,5,6],
     [7,8,9]
   ],

   ```

   原地旋转输入矩阵，使其变为:

   ```
   [
     [7,4,1],
     [8,5,2],
     [9,6,3]
   ]
   ```

   示例 **2:**

   给定 **matrix** =

   ```
   [
     [ 5, 1, 9,11],
     [ 2, 4, 8,10],
   ```

```
 [13, 3, 6, 7],
 [15,14,12,16]
],
```

原地旋转输入矩阵，使其变为:

```
[
 [15,13, 2, 5],
 [14, 3, 4, 1],
 [12, 6, 8, 9],
 [16, 7,10,11]
]
```

思路：这道题最难的地方在于不要使用额外的空间，也就是说空间复杂度的上限是 O1。这样来说，leetcode 上最快解也是达不到标准的。

那么怎么做呢？常规思路可能发现最后一行变成第一列，倒数第二行变第二列。。。。但这样的话我们实在没有办法做到 O1 完成解题。

所以思路是我们每次只移动四个点，这四个点是有要求的，它组成的图形得是个正方形。当所有的内嵌正方形完成旋转，我们的任务也就完成了。 举例[[1,2,3],[4,5,6],[7,8,9]], 我们第一个内嵌正方形选的四个点是 1 , 7, 9, 3。让它们完成旋转，也就是 1 到 3 的位置上去，7 到 1 的位置上去， 9 到 7 的位置上去， 3 到 9 的位置上去。下一个内嵌正方形是哪四个点？答：2， 4， 8， 6。 同样的操作对这四个点。然后下一个内嵌正方形是哪个点？哦，已经不剩下这样的正方形了，因为只有中心一个 5 还没有被转动。

这样的正方形符合一定的数学规律，无论是起点还是转动的四个点，所以贴下代码供大家交流。如果能推导其中的数学原理，欢迎补充。

```python
class Solution ( object ):
    def minCut ( self , s ):
        """
        :type s: str
        :rtype: int
        """
        n = len (s)
        dp = [(i - 1 ) for i in range (n + 1 )]
        for i in range ( 1 , n + 1 ):
            for j in range (i):
            tmp = s[j:i]
            if tmp == tmp[:: - 1 ]:
                dp[i] = min (dp[i], dp[j] + 1 )
                return dp[n]

class Solution :
```

```
    def rotate ( self , matrix ):
        #解法非常非常巧妙
        n = len (matrix)
        #排除特殊情况
        if n == 0 or n == 1 : return
       #这道题其实就是内嵌正方形的旋转
       #i,j 表示转动正方形转动开始的起点左标
       for i in range ( 0 , (n + 1 ) // 2 ):
            for j in range ( 0 , n // 2 ):
            #先保存我们的起点
            tmp = matrix[i][j]
            #不断的赋值，注意到每次赋值后我们都要再重新把赋值的那一项也赋值上
            matrix[i][j] = matrix[n - 1 - j][i] #赋值是连续紧凑的
            matrix[n - 1 - j][i] = matrix[n - 1 - i][n - 1 - j]
            matrix[n - 1 - i][n - 1 - j] = matrix[j][n - 1 - i]
            matrix[j][n - 1 - i] = tmp
```

64.总结：当我们对正方形中进行操作时，我们应该铭记：存在数学规律找出所有内嵌正方形的坐标。

2.无序的数组的中位数。要求时间复杂度尽可能的小

长度为 n 的无序数组，求中位数，如何尽快的估算出中位数，算法复杂度是多少？

算法 1（建立最小堆）：

如果数组中元素有奇数个，可以采用这种算法：

步骤 1 ：可以将数组的前 (n+1)//2 个元素，建立 1 个最小堆；

步骤 2 ：遍历剩余元素，如果剩余元素小于堆顶元素，则丢弃或不作处理；如果剩余元素大于堆顶元素，则将其取代堆顶元素，并将当前堆调整为最小堆。

步骤 3 ：返回堆顶元素，即 nums[0]，就是所要寻找的中位数。

一点解释：

不管是步骤 1、2 还是整个过程中，最小堆的栈顶元素必然满足：

中位数 >= 最小堆的堆顶元素

例如，[7,8,9,10,11,12,13] 中位数是 10 ，n 等于 7 ，(n+1)//2 等于 4 ，不管是取前 4 个数、后 4 个数、任意 4 个数，构造的最小堆的堆顶元素，最小为 7 ，最大为 10。

因此，小于堆顶元素的元素，必然不可能是中位数，可以直接丢弃；中位数只有可能在最小堆、剩余元素中。

实现：

```
1. # coding:utf-8
2. #from heap_sort import filter_down
```

```
3.  def filter_up(nums, p, n):
4.          parentIdx = p
5.          rootVal = nums[parentIdx]
6.
7.          while 2*parentIdx+1 <= n-1:
8.                  kidIdx = 2*parentIdx+1
9.
10.                 if kidIdx != n-1 and nums[kidIdx] > nums[kidIdx+1]:
11.                         kidIdx += 1
12.
13.                 if rootVal < nums[kidIdx]:
14.                         break
15.                 else:
16.                         nums[parentIdx] = nums[kidIdx]
17.
18.                 parentIdx = kidIdx
19.
20.         nums[parentIdx] = rootVal
21. def changeToMinHeap(nums, n):
22.         ''' 建立最小堆 '''
23.         for index in range(n//2-1, -1, -1):
24.                 filter_up(nums, index, n)
25. def find_median(nums, n):
26.         assert n%2 == 1
27.         aboutHalf = (n+1)//2
28.         changeToMinHeap(nums, aboutHalf)
29.         pointer = aboutHalf
30.
31.         for index in range(aboutHalf, n):
32.                 if nums[index] > nums[0]:
33.                         nums[0] = nums[index]
34.                         changeToMinHeap(nums, aboutHalf)
35.         return nums[0]
36.
37. def test():
```

```
        nums = list(range(4, 10)) + list(range(0, 4)) + list(range(10, 15))
        print('nums：', nums)
        assert find_median(nums, 15) == 7
        print('Pass!')
if __name__ == '__main__':
        test()
```

二、计算机网络

1. tcp 怎么保证数据包有序

```
主机每次发送数据时，TCP 就给每个数据包分配一个序列号并且在一个特定的时间内等待接收主机对分配的这个序列号进行确认。
如果发送主机在一个特定时间内没有收到接收主机的确认，则发送主机会重传此数据包。
接收主机利用序列号对接收的数据进行确认，以便检测对方发送的数据是否有丢失或者乱序等。
接收主机一旦收到已经顺序化的数据，它就将这些数据按正确的顺序重组成数据流并传递到高层进行处理。
```

2. tcp 和 udp 的异同

```
TCP 是面向流的可靠数据传输连接
UDP 是面向数据包的不可靠无连接
```

3. tcp 怎么保证可靠性

```
差错检验机制，反馈机制，重传机制，引入序号，滑动窗口协议，选择重传
```

4. tcp 中拥塞避免和流量控制机制

```
拥塞避免和流量控制这两种机制很像，但是流量控制是由接收方的接受能力也就是接收窗口所决定的，如果接收窗口够大，以动态调整发送窗口的大小调整发送速度
拥塞避免主要由网络情况所限制，网络情况良好，则加大发送速率，网络状态差（冗余 ACK 和丢包）则降低发送速率（慢启动，拥塞控制，快恢复，快重传） RENO，BBR
```

5. tcp 四次挥手的详细解释

```
tcp 四次挥手其实可以分为两个阶段
第一：
客户端至服务器的半双工连接关闭
客户端向服务器发送 FIN 信号，进入 FIN_WAIT1 的状态，等待服务器的 ACK 信号
收到服务器的 ACK 后，进入 FIN_WAIT2
第二：
服务器至客户端的半双工连接关闭
客户端收到服务器发来的 FIN 后，发送 ACK，并进入 TIME_WAIT，等待 2msl，若无异常，则客户端认为连接成功关闭
服务器收到客户端发来的 ACK 后，关闭连接
```

6. 四次挥手之后为什么还要等待 2msl

MSL 是报文最大生存时间
1 是因为有可能客户端发往服务器的 ACK 丢失，服务器并不知道客户端已经确认关闭，这时候客户端的关闭会导致服务器端无法正常关闭
2 是为了保证连接中的报文都已经传递。假如短时间关闭又重新实现一个 TCP 还连到了同个端口上，旧连接中尚未消失的数据就会被认为是新连接的数据。

7. 浏览器从输入网址到显示出网页的全过程

1. 输入网址或者 ip。
2. 如果输入的是网址，首先要查找域名的 ip 地址

第一步会在浏览器缓存中查找，如果没有，转至查询系统缓存，如果还是没有，发送请求给路由器，路由器首先会在自身的缓存中查找，如果还是没有，向 ips 发出请求，查询 ips 中的 dns 缓存，如果还是没有递归向上查询直至根服务器。

3. 浏览器与 ip 机器之间建立 TCP 连接（三次握手）（HTTP）或者在 TCP 上进一步建立 SSL/TLS 连接 （HTTPS）

接下来就是发送 HTTP 报文啥的了
GET，POST，DELETE，PUT。

8. 滑动窗口机制的原理和理解

GBN 协议，回退 N 步协议，这是对停等协议的改进，因为停等协议的传输效率非常低下。每次可发送的数据为 N，基数为 base，小于 base 的数据已经发送并且确认，base 是最小的已发送未确认的报文序号。在接收端同样也有一个接收窗口，（解释）GBN 采用的是累计确认方式，这时候说一下选择重传机制。再说一下 TCP 中既不是 GBN 也不是 SR ，而是 GBN 和 SR 的综合体。
N 的大小必须报文序列编号的一半，否则接收端对报文的确认可能发生混淆

9. Https 原理和实现

HTTPS 简介

在日常互联网浏览网页时，我们接触到的大多都是 HTTP 协议，这种协议是未加密，即明文的。这使得 HTTP 协议在传输隐私数据时非常不安全。因此，浏览器鼻祖 Netscape 公司设计了 SSL（Secure Sockets Layer） 协议，用于对 HTTP 协议传输进行数据加密，即 HTTPS 。
HTTPS 和 HTTP 协议相比提供了：

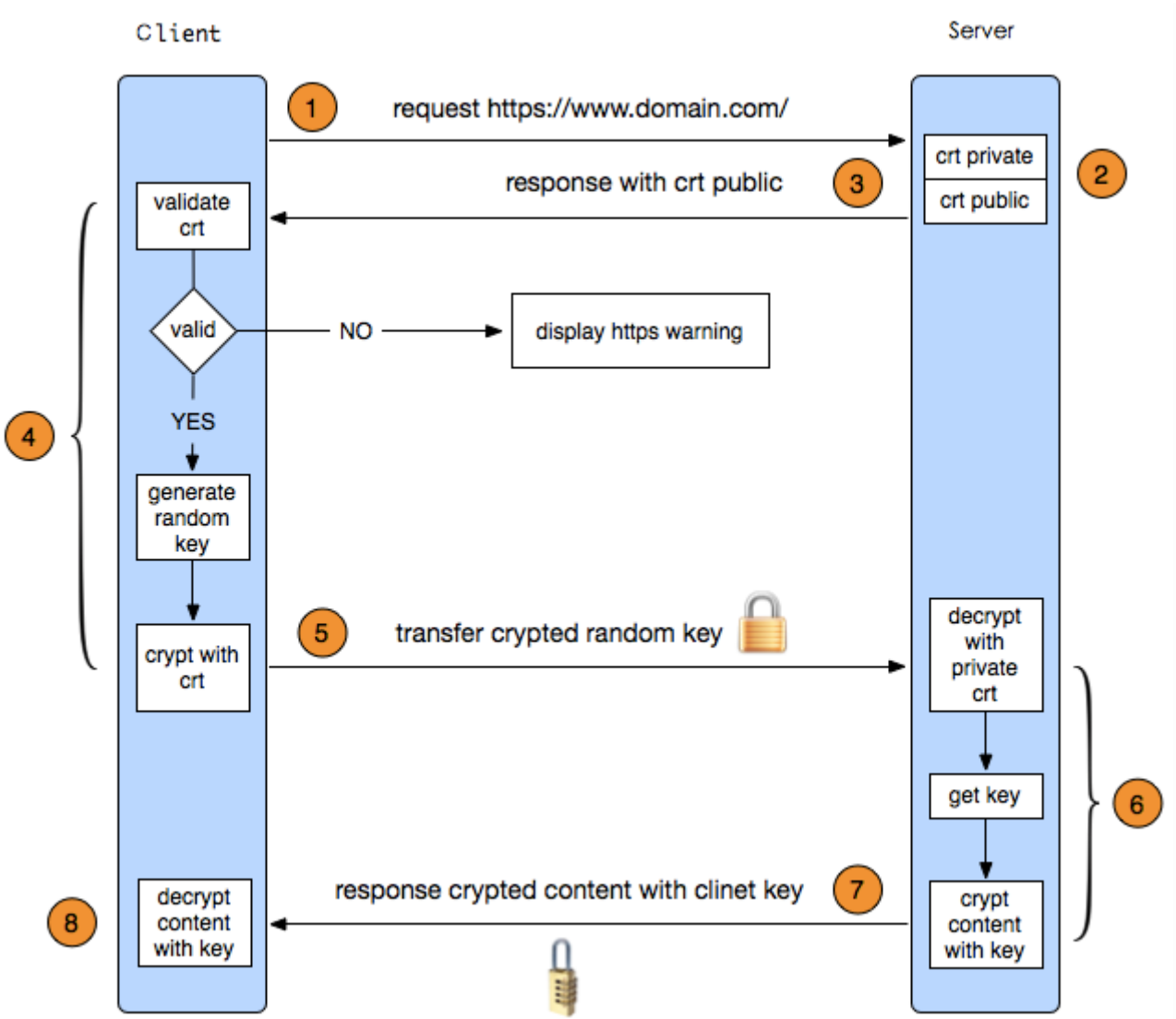
- 数据完整性：内容传输经过完整性校验
- 数据隐私性：内容经过对称加密，每个连接生成一个唯一的加密密钥
- 身份认证：第三方无法伪造服务端（客户端）身份

SSL 目前版本是 3.0，之后升级为了 TLS（Transport Layer Security） 协议，TLS 目前为 1.2 版本。如未特别说明，SSL 与 TLS 均指同一协议。

TTPS 基本工作原理是怎么实现的?我们都知道 HTTPS 能够加密信息，以免敏感信息被第三方获取。所以很多银行网站或电子商务网站都会采用 HTTPS 协议。现在很多浏览器强制要求部署 https，否则会被认定为不安全的网站，那么 HTTPS 基本工作原理是怎么实现的?

HTTPS 简介

HTTPS 其实是有两部分组成：HTTP + SSL / TLS，也就是在 HTTP 上又加了一层处理加密信息的模块。服务端和客户端的信息传输都会通过 TLS 进行加密，所以传输的数据都是加密后的数据。具体是如何进行加密，解密，验证的，且看下图。



1. 客户端发起 HTTPS 请求

这个没什么好说的，就是用户在浏览器里输入一个 https 网址，然后连接到 server 的 443 端口。

2. 服务端的配置

采用 HTTPS 协议的服务器必须要有一套数字证书，可以自己制作，也可以向组织申请。区别就是自己颁发的证书需要客户端验证通过，才可以继续访问，而使用受信任的公司申请的证书则不会弹出提示页面(startssl 就是个不错的选择，有 1 年的免费服务)。这套证书其实就是一对公钥和私钥。如果对公钥和私钥不太理解，可以想象成一把钥匙和一个锁头，只是全世界只有你一个人有这把钥匙，你可以把锁头

给别人，别人可以用这个锁把重要的东西锁起来，然后发给你，因为只有你一个人有这把钥匙，所以只有你才能看到被这把锁锁起来的东西。

3. 传送证书

这个证书其实就是公钥，只是包含了很多信息，如证书的颁发机构，过期时间等等。

4. 客户端解析证书

这部分工作是有客户端的 TLS 来完成的，首先会验证公钥是否有效，比如颁发机构，过期时间等等，如果发现异常，则会弹出一个警告框，提示证书存在问题。如果证书没有问题，那么就生成一个随即值。然后用证书对该随机值进行加密。就好像上面说的，把随机值用锁头锁起来，这样除非有钥匙，不然看不到被锁住的内容。

5. 传送加密信息

这部分传送的是用证书加密后的随机值，目的就是让服务端得到这个随机值，以后客户端和服务端的通信就可以通过这个随机值来进行加密解密了。

6. 服务段解密信息

服务端用私钥解密后，得到了客户端传过来的随机值(私钥)，然后把内容通过该值进行对称加密。所谓对称加密就是，将信息和私钥通过某种算法混合在一起，这样除非知道私钥，不然无法获取内容，而正好客户端和服务端都知道这个私钥，所以只要加密算法够彪悍，私钥够复杂，数据就够安全。

7. 传输加密后的信息

这部分信息是服务段用私钥加密后的信息，可以在客户端被还原

8. 客户端解密信息

客户端用之前生成的私钥解密服务段传过来的信息，于是获取了解密后的内容。整个过程第三方即使监听到了数据，也束手无策。

以上，是为大家分享的“HTTPS 基本工作原理是怎么实现的？”的全部内容，如果用户遇到的问题不能解决，可通过 wosign 官网客服寻求帮助，凡是选择 wosign ssl 证书的网站用户，wosign 可提供免费一对一的 ssl 证书技术部署支持，免除后顾之忧。

加密方式

加密算法一般分为对称加密与非对称加密。

对称加密

客户端与服务器使用相同的密钥对消息进行加密

优点：

- 加密强度高，很难被破解
- 计算量小，仅为非对称加密计算量的 0.1%

缺点：

- 无法安全的生成和管理密钥
- 服务器管理大量客户端密钥复杂

非对称加密

非对称指加密与解密的密钥为两种密钥。服务器提供公钥，客户端通过公钥对消息进行加密，并由服务器端的私钥对密文进行解密。

优点：安全

缺点

- 性能低下，CPU 计算资源消耗巨大，一次完全的 TLS 握手，密钥交换时的非对称加密解密占了整个握手过程的 90% 以上。而对称加密的计算量只相当于非对称加密的 0.1%，因此如果对应用层使用非对称加密，性能开销过大，无法承受。
- 非对称加密对加密内容长度有限制，不能超过公钥的长度。比如现在常用的公钥长度是 2048 位，意味着被加密消息内容不能超过 256 字节。

HTTPS 下的加密

HTTPS 一般使用的加密与 HASH 算法如下：

- 非对称加密算法：RSA，DSA/DSS
- 对称加密算法：AES，RC4，3DES
- HASH 算法：MD5，SHA1，SHA256

其中非对称加密算法用于在握手过程中加密生成的密码，对称加密算法用于对真正传输的数据进行加密，而 HASH 算法用于验证数据的完整性。由于浏览器生成的密码是整个数据加密的关键，因此在传输的时候使用了非对称加密算法对其加密。非对称加密算法会生成公钥和私钥，公钥只能用于加密数据，因此可以随意传输，而网站的私钥用于对数据进行解密，所以网站都会非常小心的保管自己的私钥，防止泄漏。

TLS 握手过程中如果有任何错误，都会使加密连接断开，从而阻止了隐私信息的传输。正是由于 HTTPS 非常的安全，攻击者无法从中找到下手的地方，于是更多的是采用了假证书的手法来欺骗客户端，从而获取明文的信息，但是这些手段都可以被识别出来

关于证书

证书需要申请，并由专门的数字证书认证机构 CA 通过非常严格的审核之后颁发的电子证书，证书是对服务器端的一种认证。颁发的证书同时会产生一个私钥和公钥。私钥有服务器自己保存，不可泄露，公钥则附带在证书的信息中，可以公开。证书本身也附带一个证书的电子签名，这个签名用来验证证书的完整性和真实性，防止证书被篡改。此外证书还有个有效期。

证书包含以下信息：

- 使用者的公钥值。
- 使用者标识信息（如名称和电子邮件地址）。
- 有效期（证书的有效时间）。
- 颁发者标识信息。
- 颁发者的数字签名，用来证明使用者的公钥和使用者的标识符信息之间的绑定的有效性。

10. cookie 和 session 的区别是什么

由于 HTTP 协议是无状态的协议，所以服务端需要记录用户的状态时，就需要用某种机制来识具体的用户

cookie 存在本地的上的

session 是存在服务器上的

通俗讲，Cookie 是访问某些网站以后在本地存储的一些网站相关的信息，下次再访问的时候减少一些步骤。另外一个更准确的说法是：Cookies 是服务器在本地机器上存储的小段文本并随每一个请求发送至同一个服务器，是一种在客户端保持状态的方案。

Session 是存在服务器的一种用来存放用户数据的类 HashTable 结构。

二者都用来保持用户的状态，cookie 可更改，对服务器来说并不安全，服务器常见做法有这两种：

1.把 session 加密后放入浏览器的 cookie 中，浏览器重连后将加密的 session 发给服务器

2.cookie 中存储着 session 的 id，浏览器重连时只需要发送 session_id' 即可

三、操作系统

1. 进程和线程的区别

进程就是保存上下文切换的程序执行时间总和 = CPU 加载上下文 + CPU 执行 + CPU 保存上下文

2. 线程是什么呢？

进程的颗粒度太大，每次都要有上下的调入，保存，调出。如果我们把进程比喻为一个运行在电脑上的软件，那么一个软件的执行不可能是一条逻辑执行的，必定有多个分支和多个程序段，就好比要实现程序 A，实际分成 a，b，c 等多个块组合而成。那么这里具体的执行就可能变成：
程序 A 得到 CPU =》CPU 加载上下文，开始执行程序 A 的 a 小段，然后执行 A 的 b 小段，然后再执行 A 的 c 小段，最后 CPU 保存 A 的上下文。
这里 a，b，c 的执行是共享了 A 的上下文， CPU 在执行的时候没有进行上下文切换的。这里的 a，b，c 就是线程，也就是说线程是共享了进程的上下文环境，的更为细小的 CPU 时间段。

3. 进程切换与线程切换

进程切换分两步：

1.切换页目录以使用新的地址空间

2.切换内核栈和硬件上下文

对于 linux 来说，线程和进程的最大区别就在于地址空间，对于线程切换，第 1 步是不需要做的，第 2 是进程和线程切换都要做的。

切换的性能消耗：

1、线程上下文切换和进程上下问切换一个最主要的区别是线程的切换虚拟内存空间依然是相同的，但是进程切换是不同的。这两种上下文切换的处理都是通过操作系统内核来完成的。内核的这种切换过程伴随的最显著的性能损耗是将寄存器中的内容切换出。

2、另外一个隐藏的损耗是上下文的切换会扰乱处理器的缓存机制。简单的说，一旦去切换上下文，处理器中所有已经缓存的内存地址一瞬间都作废了。还有一个显著的区别是当你改变虚拟内存空间的时候，处理的页表缓冲（processor's Translation Lookaside Buffer (TLB)）或者相当的神马东西会被全部刷新，这将导致内存的访问在一段时间内相当的低效。但是在线程的切换中，不会出现这个问题。

系统调用：处于进程上下文

系统调用是在进程上下文中,并没有 tasklet 之类的延迟运行,系统调用本身可以休眠,这些可以参见内核代码

虽然系统调用实与其他中断实现有点类似,通过 IDT 表查找入口处理函数,但是系统调用与其他中断最大的不同是,系统调用是代表当前进程执行的,所以 current 宏/task_struct 是有意义的,这个休眠可以被唤醒

系统调用，异常，中断（其中中断是异步时钟，异常时同步时钟），也可以把系统调用成为异常

中断上下文：在中断中执行时依赖的环境，就是中断上下文（不包括系统调用，是硬件中断）

进程上下文：当一个进程在执行时,CPU 的所有寄存器中的值、进程的状态以及堆栈中的内容被称为该进程的上下文

1、首先，这两个上下文都处于内核空间。

2、其次，两者的区别在于，进程上下文与当前执行进程密切相关，而中断上下文在逻辑上与进程没有关系。

进程上下文主要是异常处理程序和内核线程。内核之所以进入进程上下文是因为进程自身的一些工作需要在内核中做。例如，系统调用是为当前进程服务的，异常通常是处理进程导致的错误状态等。所以在进程上下文中引用 current 是有意义的。

内核进入中断上下文是因为中断信号而导致的中断处理或软中断。而中断信号的发生是随机的，中断处理程序及软中断并不能事先预测发生中断时当前运行的是哪个进程，所以在中断上下文中引用 current 是可以的，但没有意义。事实上，对于 A 进程希望等待的中断信号，可能在 B 进程执行期间发生。例如，A 进程启动写磁盘操作，A 进程睡眠后现在时 B 进程在运行，当磁盘写完后磁盘中断信号打断的是 B 进程，在中断处理时会唤醒 A 进程。

上下文这个词会让人想到进程的 CPU 寄存器状态，但好像进入进程上下文（异常处理系统调用）和进入中断上下文（中断处理），内核所做的工作没有太大区别。所以，这两个上下文的主要区别，我认为在于是否与进程相关。

运行于进程上下文的内核代码是可抢占的，但中断上下文则会一直运行至结束，不会被抢占。因此，内核会限制中断上下文的工作，不允许其执行如下操作：

(1) 进入睡眠状态或主动放弃 CPU;

由于中断上下文不属于任何进程，它与 current 没有任何关系（尽管此时 current 指向被中断的进程），所以中断上下文一旦睡眠或者放弃 CPU，将无法被唤醒。所以也叫原子上下文（atomic context）。

(2) 占用互斥体;

为了保护中断句柄临界区资源，不能使用 mutexes。如果获得不到信号量，代码就会睡眠，会产生和上面相同的情况，如果必须使用锁，则使用 spinlock。

(3) 执行耗时的任务;

进程切换和线程切换？

我们都知道线程切换的开销比进程切换的开销小，那么小在什么地方？切换的过程是怎样的？

无论是在多核还是单核系统中，一个 CPU 看上去都像是在并发的执行多个进程，这是通过处理器在进程间切换来实现的。

- 操作系统实现这种交错执行的机制称为上下文切换。
- 操作系统保持跟踪进程运行所需的所有状态信息，这种状态，也就是上下文，它包括许多信息，例如 PC 和寄存器文件的当前值，以及主存的内容。

在任何一个时刻，单处理器系统都只能执行一个进程的代码。

当操作系统决定要把控制权从当前进程转移到某个新进程时，就会进行上下文切换，即保存当前进程的上下文，恢复新进程的上下文，然后将控制权传递到新进程，新进程就会从上次停止的地方开始

深入计算机系统一书中对上下文切换的表达如下图：

如果现在有两个并发的进程：外壳进程和 hello 进程。

开始只有外壳进程在运行，即等待命令行上的输入，当我们让他运行 hello 程序时，外壳通过调用一个专门的函数，即系统调用，来执行我们的请求，系统调用会将控制权传递给操作系统。

操作系统保存外壳进程的上下文，创建一个新的 hello 进程及其上下文，然后将控制权传递给新的 hello 进程。

hello 进程终止后，操作系统恢复外壳进程的上下文，并将控制权传回给他，外壳进程将继续等待下一个命令行输入。

上下文切换

内核为每一个进程维持一个上下文。**上下文就是内核重新启动一个被抢占的进程所需的状态。**包括一下内容：

- 通用目的寄存器
- 浮点寄存器
- 程序计数器

- 用户栈
- 状态寄存器
- 内核栈
  - 各种内核数据结构：比如描绘地址空间的页表，包含有关当前进程信息的进程表，以及包含进程已打开文件的信息的文件表。

进程切换

系统中的每个程序都是运行在某个进程的上下文中的。

上下文是由程序正确运行所需的状态组成的，这个状态包括存放在存储器中的程序的代码和数据，他的栈，通用目的寄存器的内容，程序计数器，环境变量以及打开文件描述符的集合。

所以进程切换就是上下文切换。

那回到最开始的问题，进程切换和线程切换有什么区别？

当然这里的线程指的是同一个进程中的线程。要想正确回答这个问题，需要理解虚拟内存。

虚拟内存

虚拟内存是操作系统为每个进程提供的一种抽象，每个进程都有属于自己的、私有的、地址连续的虚拟内存，当然我们知道最终进程的数据及代码必然要放到物理内存上，那么必须有某种机制能记住虚拟地址空间中的某个数据被放到了哪个物理内存地址上，这就是所谓的地址空间映射，那么操作系统是如何记住这种映射关系的呢，答案就是页表。

每个进程都有自己的虚拟地址空间，进程内的所有线程共享进程的虚拟地址空间。

现在我们就可以来回答这个面试题了。

进程切换和线程切换的区别

最主要的一个区别在于**进程切换涉及虚拟地址空间的切换而线程不会**。因为每个进程都有自己的虚拟地址空间，而线程是共享所在进程的虚拟地址空间的，因此同一个进程中的线程进行线程切换时不涉及虚拟地址空间的转换。

有的同学可能还是不太明白，为什么虚拟地址空间切换会比较耗时呢？

现在我们已经知道了进程都有自己的虚拟地址空间，把虚拟地址转换为物理地址需要查找页表，页表查找是一个很慢的过程，因此通常使用 Cache 来缓存常用的地址映射，这样可以加速页表查找，这个 cache 就是 TLB（translation Lookaside Buffer，我们不需要关心这个名字只需要知道 TLB 本质上就是一个 cache，是用来加速页表查找的）。由于每个进程都有自己的虚拟地址空间，那么显然每个进程都有自己的页表，那么**当进程切换后页表也要进行切换，页表切换后 TLB 就失效了**，cache 失效导致命中率降低，那么虚拟地址转换为物理地址就会变慢，表现出来的就是程序运行会变慢，而线程切换则不会导致 TLB 失效，因为线程线程无需切换地址空间，因此我们通常说线程切换要比较进程切换块，原因就在这里。

为了控制进程的执行，内核必须有能力挂起正在 CPU 上运行的进程，并恢复以前挂起的某个进程的执行。这种行为被称为进程切换（*process switch*）、任务切换（*task switch*）或上下文切换（*content switch*）。

原文 ： https://www.cnblogs.com/kkshaq/p/4547725.html

进程切换分两步：

1.切换页目录以使用新的地址空间

2.切换内核栈和硬件上下文

对于 linux 来说，线程和进程的最大区别就在于地址空间，对于线程切换，第 1 步是不需要做的，第 2 是进程和线程切换都要做的。

切换的性能消耗：

1、线程上下文切换和进程上下问切换一个最主要的区别是线程的切换虚拟内存空间依然是相同的，但是进程切换是不同的。这两种上下文切换的处理都是通过操作系统内核来完成的。内核的这种切换过程伴随的最显著的性能损耗是将寄存器中的内容切换出。
2、另外一个隐藏的损耗是上下文的切换会扰乱处理器的缓存机制。简单的说，一旦去切换上下文，处理器中所有已经缓存的内存地址一瞬间都作废了。还有一个显著的区别是当你改变虚拟内存空间的时候，处理的页表缓冲（processor's Translation Lookaside Buffer (TLB)）会被全部刷新，这将导致内存的访问在一段时间内相当的低效。但是在线程的切换中，不会出现这个问题。

系统调用：处于进程上下文
系统调用是在进程上下文中,并没有 tasklet 之类的延迟运行,系统调用本身可以休眠,这些可以参见内核代码

虽然系统调用实与其他中断实现有点类似,通过 IDT 表查找入口处理函数,但是系统调用与其他中断最大的不同是,系统调用是代表当前进程执行的,所以 current 宏/task_struct 是有意义的,这个休眠可以被唤醒

系统调用，异常，中断（其中中断是异步时钟，异常时同步时钟），也可以把系统调用成为异常
中断上下文：在中断中执行时依赖的环境，就是中断上下文（不包括系统调用，是硬件中断）
进程上下文：当一个进程在执行时,CPU 的所有寄存器中的值、进程的状态以及堆栈中的内容被称为该进程的上下文
1、首先，这两个上下文都处于内核空间。

2、其次，两者的区别在于，进程上下文与当前执行进程密切相关，而中断上下文在逻辑上与进程没有关系。

进程上下文主要是异常处理程序和内核线程。内核之所以进入进程上下文是因为进程自身的一些工作需要在内核中做。例如，系统调用是为当前进程服务的，异常通常是处理进程导致的错误状态等。所以在进程上下文中引用 current 是有意义的。

内核进入中断上下文是因为中断信号而导致的中断处理或软中断。而中断信号的发生是随机的，中断处理程序及软中断并不能事先预测发生中断时当前运行的是哪个进程，所以在中断上下文中引用 current 是可以的，但没有意义。事实上，对于 A 进程希望等待的中断信号，可能在 B 进程执行期间发生。例如，A 进程启动写磁盘操作，A 进程睡眠后现在时 B 进程在运行，当磁盘写完后磁盘中断信号打断的是 B 进程，在中断处理时会唤醒 A 进程。
上下文这个词会让人想到进程的 CPU 寄存器状态，但好像进入进程上下文（异常处理系统调用）和进入中断上下文（中断处理），内核所做的工作没有太大区别。所以，这两个上下文的主要区别，我认为在于是否与进程相关。
运行于进程上下文的内核代码是可抢占的，但中断上下文则会一直运行至结束，不会被抢占。因此，内核会限制中断上下文的工作，不允许其执行如下操作：
(1) 进入睡眠状态或主动放弃 CPU;
由于中断上下文不属于任何进程，它与 current 没有任何关系（尽管此时 current 指向被中断的进程），所以中断上下文一旦睡眠或者放弃 CPU，将无法被唤醒。所以也叫原子上下文（atomic context）。
(2) 占用互斥体;

为了保护中断句柄临界区资源，不能使用 mutexes。如果获得不到信号量，代码就会睡眠，会产生和上面相同的情况，如果必须使用锁，则使用 spinlock。

(3) 执行耗时的任务;

中断处理应该尽可能快，因为内核要响应大量服务和请求，中断上下文占用 CPU 时间太长会严重影响系统功能。在中断处理例程中执行耗时任务时，应该交由中断处理例程底半部来处理。

(4) 访问用户空间虚拟内存。

因为中断上下文是和特定进程无关的，它是内核代表硬件运行在内核空间，所以在中断上下文无法访问用户空间的虚拟地址

(5) 中断处理例程不应该设置成 reentrant（可被并行或递归调用的例程）。

因为中断发生时，preempt 和 irq 都被 disable，直到中断返回。所以中断上下文和进程上下文不一样，中断处理例程的不同实例，是不允许在 SMP 上并发运行的。

(6)中断处理例程可以被更高级别的 IRQ 中断。（不能嵌套中断）使用软中断，上部分关中断，也就是禁止嵌套，下半部分使用软中断

如果想禁止这种中断，可以将中断处理例程定义成快速处理例程，相当于告诉 CPU，该例程运行时，禁止本地 CPU 上所有中断请求。这直接导致的结果是，由于其他中断被延迟响应，系统性能下降。

软中断是一种延时机制，代码执行的优先级比进程要高，比硬中断要低。相比于硬件中断，软中段是在开中断的环境中执行的（长时间关中断对系统的开销太大）， 代码是执行在中断/线程上下文的，是不能睡眠的，虽然每个 cpu 都有一个对应的 ksoftirqd/n 线程来执行软中断，但是 do_softirq 这个函数也还会在中断退出时调用到，因此不能睡眠(中断上下文不能睡眠的原因是由于调度系统是以进程为基本单位的，调度时会把当前进程的上下文保存在 task_struct 这个数据结构中，当进程被调度重新执行时会找到执行的断点，但是中断上下文是没有特定 task_struct 结构体的，当然现在有所谓的线程话中断，可以满足在中断处理函数执行阻塞操作，但是实时性可能会有问题。还有就是中断代表当前进程执行的概念，个人感觉有点扯淡，毕竟整个内核空间是由所有进程共享的，不存在代表的概念)

上面我们介绍的可延迟函数运行在中断上下文中（软中断的一个检查点就是 do_IRQ 退出的时候），于是导致了一些问题：软中断不能睡眠、不能阻塞。由于中断上下文出于内核态，没有进程切换，所以如果软中断一旦睡眠或者阻塞，将无法退出这种状态，导致内核会整个僵死。但可阻塞函数不能用在中断上下文中实现，必须要运行在进程上下文中，例如访问磁盘数据块的函数。因此，可阻塞函数不能用软中断来实现。但是它们往往又具有可延迟的特性。

4. Linux 中五种 IO 模型

1) 阻塞 I/O（blocking I/O）

2) 非阻塞 I/O （nonblocking I/O）

3) I/O 复用 (select 和 poll) （I/O multiplexing）

4) 信号驱动 I/O （signal driven I/O (SIGIO)）

5) 异步 I/O （asynchronous I/O (the POSIX aio_functions)）

前四种都是同步，只有最后一种才是异步 IO。

同步 IO 和异步 IO 的区别就在于：数据拷贝的时候进程是否阻塞！

阻塞 IO 和非阻塞 IO 的区别就在于：应用程序的调用是否立即返回！

5. 如何实现一个同步非阻塞的请求

BIO 与 NIO

IO 为同步阻塞形式,NIO 为同步非阻塞形式,NIO 并没有实现异步,在 JDK1.7 后升级 NIO 库包，支持异步非阻塞

同学模型 NIO2.0(AIO)

BIO（同步阻塞式 IO）

同步阻塞式 IO，服务器实现模式为一个连接一个线程，即客户端有连接请求时服务器端就需要启动一个线程进行处理，如果这个连接不做任何事情会造成不必要的线程开销，当然可以通过线程池机制改善。

NIO（同步非阻塞式 IO）

同步非阻塞式 IO，服务器实现模式为一个请求一个线程，即客户端发送的连接请求都会注册到多路复用器上，多路复用器轮询到连接有 I/O 请求时才启动一个线程进行处理。

AIO（异步非阻塞式 IO）

异步非阻塞式 IO，服务器实现模式为一个有效请求一个线程，客户端的 I/O 请求都是由 OS 先完成了再通知服务器应用去启动线程进行处理。

BIO（IO）与 NIO 区别

其本质就是阻塞和非阻塞的区别

什么是阻塞?

应用程序在获取网络数据的时候,如果网络传输数据很慢，就会一直等待,直到传输完毕为止。

什么是非阻塞?

应用程序直接可以获取已经准备就绪好的数据,无需等待。

同步时，应用程序会直接参与 IO 读写操作,并且我们的应用程序会直接阻塞到某一个方法上,直到数据准备就绪；或者采用轮训的策略实时检查数据的就绪状态,如果就绪则获取数据.

异步时,则所有的 IO 读写操作交给操作系统,与我们的应用程序没有直接关系，我们程序不需要关系 IO 读写，当操作系统完成了 IO 读写操作时,会给我们应用程序发送通知,我们的应用程序直接拿走数据极即可。

伪异步

问题思考

如何解决同步阻塞 IO?

答：使用伪异步阻塞 IO（多线程）！

由于 BIO 一个客户端需要一个线程去处理，因此我们进行优化，后端使用线程池来处理多个客户端的请求接入，形成客户端个数 M：线程池最大的线程数 N 的比例关系，其中 M 可以远远大于 N，通过线程池可以灵活的调配线程资源，设置线程的最大值，防止由于海量并发接入导致线程耗尽。

原理：

当有新的客户端接入时，将客户端的 Socket 封装成一个 Task（该 Task 任务实现了 java 的 Runnable 接口）投递到后端的线程池中进行处理，由于线程池可以设置消息队列的大小以及线程池的最大值，因此，它的资源占用是可控的，无论多少个客户端的并发访问，都不会导致资源的耗尽或宕机。

使用多线程支持多个请求

服务器实现模式为一个连接一个线程，即客户端有连接请求时服务器端就需要启动一个线程进行处理，如果这个连接不做任何事情会造成不必要的线程开销，当然可以通过线程池机制改善。

1、同步阻塞方式，发送方向接收方发送请求后，一直等待响应；接收方处理请求时进行的 IO 操作如果不能马上得到结果，就一直等到返回结果后，才响应发送方，期间不能进行其他工作。比如，在超市排队付账时，客户（发送方）向收款员（接收方）付款（发送请求）后需要等待收款员找零，期间不能做其他的事情；而收款员等待收款机返回结果（IO）操作后才能把零钱取出来交给客户（响应请求），期间也只能等待，不能做其他事情。这种方式实现简单，但是效率不高。

2、同步非阻塞方式，发送方向接收方发送请求后，一直等待响应；接收方处理请求时进行的 IO 操作如果不能马上得到结果，就立即返回，去做其他事情，但由于没有得到请求处理结果，不响应发送方，发送方一直等待。一直到 IO 操作完成后，接收方获得结果响应发送方后，接收方才进入下一次请求过程。在实际中不使用这种方式。

3、异步阻塞方式，发送方向接收方发送请求后，不用等待响应，可以接着进行其他工作；接收方处理请求时进行的 IO 操作如果不能马上得到结果，就一直等到返回结果后，才响应发送方，期间不能进行其他工作。这种方式在实际中也不使用。

4、异步非阻塞方式，发送方向接收方发送请求后，不用等待响应，可以继续其他工作；接收方处理请求时进行的 IO 操作如果不能马上得到结果，也不等待，而是马上返回去做其他事情。当 IO 操作完成以后，将完成状态和结果通知接收方，接收方再响应发送方。继续使用在超市排队付账的例子。客户（发送方）向收款员（接收方）付款（发送请求）后在等待收款员找零的过程中，还可以做其他事情，比如打电话、聊天等；而收款员在等待收款机处理交易（IO 操作）的过程中还可以帮助客户将商品打包，当收款机产生结果后，收款员给客户结账（响应请求）。在四种方式中，这种方式是发送方和接收方通信效率最高的一种。

再记录一下我对他们的简单理解

1.同步与异步针对的是客户端，同步是指客户端要一直等待服务端返回结果，期间不能做其他事情，异步是指客户端无需等待服务端结果，可以做其他事情

2.阻塞和非阻塞针对的是服务端，阻塞是指服务端对客户的请求执行系统 I/O 操作时要等待系统给出结果，期间不能做其他事情，非阻塞是指服务端把请求交给系统 I/O 后，可以做其他事情，并且会轮询查看之前的请求系统是否给出结果，给出就返回，再处理下一个，没给出就直接处理下一个

3.同步非阻塞方式在实际中不使用是因为这样客户对会一直需要等待，因为服务端不会专门开一个线程服务该客户端的请求，所以客户端体验是最差的

4.异步阻塞方式也不在实际中使用是因为客户端可以一直对服务端进行操作，导致服务端压力很大，需要非常多的线程来维护请求，所以这要求服务端的性能非常高才行

6. 实现进程同步的机制有什么

linux 下进程间的同步机制有哪些?

今天面试，面试官问了这个问题，我的回答是:互斥量，读写锁，信号量，临界区，条件变量。面试官说只有一个是正确的，然后我就不知道其他的了。面试官最后说还有管道，匿名管道，共享内存，套接字。不太懂

面试官说的是对的。参考 APUE 进程间通信一章引言部分的那张表。你的回答大多是线程同步机制。

进程是相互独立的，所以进程间通信大多不需要锁，需要的锁也是文件锁之类的“大锁”，并不需要条件变量、互斥锁这些机制来同步，进程之间对资源的使用由操作系统的进程调度部分来完成。线程需要互斥锁的原因是，除了线程栈里的东西其他都是共享的，需要你自己来完成变量级别的同步。

另外，线程之间的叫同步，进程之间的叫通信。进程间也有同步问题，而 IPC 是同步的实现手段。

你的这些回答大多是用于线程，或者说进程内同步。

这些方法大多对进程与进程间没法用或者不是常规用法。面试官很显然无误的是指的进程间，你的回答当然不准确，进程间确实一般不会用这种方法同步。

进程之间传递信息的各种途径(包括各种 IPC 机制)总结如下:

- 父进程通过 fork 可以将打开文件的描述符传递给子进程
- 子进程结束时,父进程调用 wait 可以得到子进程的终止信息
- 几个进程可以在文件系统中读写某个共享文件,也可以通过给文件加锁来实现进程间同步
- 进程之间互发信号,一般使用 SIGUSR1 和 SIGUSR2 实现用户自定义功能
- 管道
- FIFO mmap 函数,几个进程可以映射同一内存区
- SYS V IPC,以前的 SYS V UNIX 系统实现的 IPC 机制,包括消息队列、信号量和共享内 存,现在已经基本废弃
- UNIX Domain Socket,目前最广泛使用的 IPC 机制

你的题目写的是进程同步，而面试官的说的都是进程通讯。是两码事。不知道是你误解了面试官的意思，还是面试官自己搞错了。
通讯有很多方式
1）管道
2）共享内存
3）消息队列
4）Socket
等，只要能交流就可以了。这个是 IPC（Inter Process Communication）
而同步的方式主要是
1）Mutex（互斥）可以跨进程使用
2）Semphore（信号量）可以跨进程使用
等
两者适用范围场合依据需求决定

一、进程/线程间同步机制。
临界区、互斥区、事件、信号量四种方式
临界区（Critical Section）、互斥量（Mutex）、信号量（Semaphore）、事件（Event）的区别
1、临界区：通过对多线程的串行化来访问公共资源或一段代码，速度快，适合控制数据访问。在任意时刻只允许一个线程对共享资源进行访问，如果有多个线程试图访问公共资源，那么在有一个线程进入后，其他试图访问公共资源的线程将被挂起，并一直等到进入临界区的线程离开，临界区在被释放后，其他线程才可以抢占。
2、互斥量：采用互斥对象机制。 只有拥有互斥对象的线程才有访问公共资源的权限，因为互斥对象只有一个，所以能保证公共资源不会同时被多个线程访问。互斥不仅能实现同一应用程序的公共资源安全共享，还能实现不同应用程序的公共资源安全共享 .互斥量比临界区复杂。因为使用互斥不仅仅能够在同一应用程序不同线程中实现资源的安全共享，而且可以在不同应用程序的线程之间实现对资源的安全共享。
3、信号量：它允许多个线程在同一时刻访问同一资源，但是需要限制在同一时刻访问此资源的最大线程数目 .信号量对象对线程的同步方式与前面几种方法不同，信号允许多个线程同时使用共享资源，这与操作系统中的 PV 操作相同。它指出了同时访问共享资源的线程最大数目。它允许多个线程在同一时刻访问同一资源，但是需要限制在同一时刻访问此资源的最大线程数目。
PV 操作及信号量的概念都是由荷兰科学家 E.W.Dijkstra 提出的。信号量 S 是一个整数，S 大于等于零时代表可供并发进程使用的资源实体数，但 S 小于零时则表示正在等待使用共享资源的进程数。
P 操作申请资源:

（1）S 减 1；
（2）若 S 减 1 后仍大于等于零，则进程继续执行；
（3）若 S 减 1 后小于零，则该进程被阻塞后进入与该信号相对应的队列中，然后转入进程调度。
V 操作 释放资源：
（1）S 加 1；
（2）若相加结果大于零，则进程继续执行；
（3）若相加结果小于等于零，则从该信号的等待队列中唤醒一个等待进程，然后再返回原进程继续执行或转入进程调度。
4、事 件： 通过通知操作的方式来保持线程的同步，还可以方便实现对多个线程的优先级比较的操作 .
总结：
1． 互斥量与临界区的作用非常相似，但互斥量是可以命名的，也就是说它可以跨越进程使用。所以创建互斥量需要的资源更多，所以如果只为了在进程内部是用的话使用临界区会带来速度上的优势并能够减少资源占用量。因为互斥量是跨进程的互斥量一旦被创建，就可以通过名字打开它。
2． 互斥量（Mutex），信号灯（Semaphore），事件（Event）都可以被跨越进程使用来进行同步数据操作，而其他的对象与数据同步操作无关，但对于进程和线程来讲，如果进程和线程在运行状态则为无信号状态，在退出后为有信号状态。所以可以使用 WaitForSingleObject 来等待进程和线程退出。
3． 通过互斥量可以指定资源被独占的方式使用，但如果有下面一种情况通过互斥量就无法处理，比如现在一位用户购买了一份三个并发访问许可的数据库系统，可以根据用户购买的访问许可数量来决定有多少个线程/进程能同时进行数据库操作，这时候如果利用互斥量就没有办法完成这个要求，信号灯对象可以说是一种资源计数器。

二、进程间通信方式
由于比较容易混淆，我们把进程间通信方法也列在这里做比较。
程间通信就是在不同进程之间传播或交换信息，那么不同进程之间存在着什么双方都可以访问的介质呢？进程的用户空间是互相独立的，一般而言是不能互相访问的，唯一的例外是共享内存区。但是，系统空间却是“公共场所”，所以内核显然可以提供这样的条件。除此以外，那就是双方都可以访问的外设了。在这个意义上，两个进程当然也可以通过磁盘上的普通文件交换信息，或者通过“注册表”或其它数据库中的某些表项和记录交换信息。广义上这也是进程间通信的手段，但是一般都不把这算作“进程间通信”。因为那些通信手段的效率太低了，而人们对进程间通信的要求是要有一定的实时性。
进程间通信主要包括管道，系统 IPC(包括消息队列,信号量,共享存储), SOCKET.
管道分为有名管道和无名管道，无名管道只能用于亲属进程之间的通信，而有名管道则可用于无亲属关系的进程之间。
消息队列用于运行于同一台机器上的进程间通信，与管道相似；
共享内存通常由一个进程创建，其余进程对这块内存区进行读写。得到共享内存有两种方式：映射/dev/mem 设备和内存映像文件。前一种方式不给系统带来额外的开销，但在现实中并不常用，因为它控制存取的是实际的物理内存；
本质上，信号量是一个计数器，它用来记录对某个资源（如共享内存）的存取状况。一般说来，为了获得共享资源，进程需要执行下列操作：
（1）测试控制该资源的信号量；
（2）若此信号量的值为正，则允许进行使用该资源，进程将进号量减 1；
（3）若此信号量为 0，则该资源目前不可用，进程进入睡眠状态，直至信号量值大于 0，进程被唤醒，转入步骤（1）；
（4）当进程不再使用一个信号量控制的资源时，信号量值加 1，如果此时有进程正在睡眠等待此信号量，则唤醒此进程。
套接字通信并不为 Linux 所专有，在所有提供了 TCP/IP 协议栈的操作系统中几乎都提供了 socket，而所有这样操作系统，对套接字的编程方法几乎是完全一样的
三、进程/线程同步机制与进程间通信机制比较
很明显 2 者有类似，但是差别很大

同步主要是临界区、互斥、信号量、事件
进程间通信是管道、内存共享、消息队列、信号量、socket
共通之处是，信号量和消息（事件）
其他资料：
进程间通讯(IPC)方法主要有以下几种：
管道/FIFO/共享内存/消息队列/信号
1.管道中还有命名管道和非命名管道(即匿名管道)之分，非命名管道(即匿名管道)只能用于父子进程通讯，命名管道可用于非父子进程，命名管道就是 FIFO，管道是先进先出的通讯方式
2.消息队列是用于两个进程之间的通讯，首先在一个进程中创建一个消息队列，然后再往消息队列中写数据，而另一个进程则从那个消息队列中取数据。需要注意的是，消息队列是用创建文件的方式建立的，如果一个进程向某个消息队列中写入了数据之后，另一个进程并没有取出数据，即使向消息队列中写数据的进程已经结束，保存在消息队列中的数据并没有消失，也就是说下次再从这个消息队列读数据的时候，就是上次的数据！！！！
3.信号量，它与 WINDOWS 下的信号量是一样的，所以就不用多说了
4.共享内存，类似于 WINDOWS 下的 DLL 中的共享变量，但 LINUX 下的共享内存区不需要像 DLL 这样的东西，只要首先创建一个共享内存区，其它进程按照一定的步骤就能访问到这个共享内存区中的数据，当然可读可写
以上几种方式的比较：
1.管道：速度慢，容量有限，只有父子进程能通讯
2.FIFO：任何进程间都能通讯，但速度慢
3.消息队列：容量受到系统限制，且要注意第一次读的时候，要考虑上一次没有读完数据的问题
4.信号量：不能传递复杂消息，只能用来同步
5.共享内存区：能够很容易控制容量，速度快，但要保持同步，比如一个进程在写的时候，另一个进程要注意读写的问题，相当于线程中的线程安全，当然，共享内存区同样可以用作线程间通讯，不过没这个必要，线程间本来就已经共享了同一进程内的一块内存

进程同步的 5 种机制

（1）信号量机制

一个信号量只能置一次初值，以后只能对之进行 p 操作或 v 操作。 由此也可以看到，信号量机制必须有公共内存，不能用于分布式操作系统，这是它最大的弱点。

（2）自旋锁

旋锁是为了保护共享资源提出的一种锁机制。 调用者申请的资源如果被占用，即自旋锁被已经被别的执行单元保持，则调用者一直循环在那里看是否该自旋锁的保持着已经释放了锁，自旋锁是一种比较低级的保护数据结构和代码片段的原始方式，可能会引起以下两个问题；
（1）死锁
（2）过多地占用 CPU 资源

（3）管程

信号量机制功能强大，但使用时对信号量的操作分散，而且难以控制，读写和维护都很困难。因此后来又提出了一种集中式同步进程——管程。其基本思想是将共享变量和对它们的操作集中在一个模块中，操作系统或并发程序就由这样的模块构成。这样模块之间联系清晰，便于维护和修改，易于保证正确性。

（4）会合

进程直接进行相互作用

（5）分布式系统

由于在分布式操作系统中没有公共内存，因此参数全为值参，而且不可为指针。

优缺点：

（1）信号量（Semaphore）及 PV 操作

优：PV 操作能够实现对临界区的管理要求；实现简单；允许使用它的代码休眠，持有锁的时间可相对较长。

缺：信号量机制必须有公共内存，不能用于分布式操作系统，这是它最大的弱点。信号量机制功能强大，但使用时对信号量的操作分散，而且难以控制，读写和维护都很困难。加重了程序员的编码负担；核心操作 P-V 分散在各用户程序的代码中，不易控制和管理；一旦错误，后果严重，且不易发现和纠正。

（2）自旋锁

优：旋锁是为了保护共享资源提出的一种锁机制； 调用者申请的资源如果被占用，即自旋锁已经被别的执行单元保持，则调用者一直循环在那里看是否该自旋锁的保持者已经释放了锁； 低开销；安全和高效；

缺：自旋锁是一种比较低级的保护数据结构和代码片段的原始方式，可能会引起以下两个问题;

（1）死锁

（2）过多地占用 CPU 资源

传统自旋锁由于无序竞争会导致“公平性”问题

（3）管程

优： 集中式同步进程——管程。其基本思想是将共享变量和对它们的操作集中在一个模块中，操作系统或并发程序就由这样的模块构成。这样模块之间联系清晰，便于维护和修改，易于保证正确性。

缺：如果一个分布式系统具有多个 CPU，并且每个 CPU 拥有自己的私有内存，它们通过一个局域网相连，那么这些原语将失效。而管程在少数几种编程语言之外又无法使用，并且，这些原语均未提供机器间的信息交换方法。

（4）会合

进程直接进行相互作用

（5）分布式系统

消息和 rpc ，由于在分布式操作系统中没有公共内存，因此参数全为值参， 而且不可为指针.

python 之路 29 -- 多进程与进程同步（进程锁、信号量、事件）与进程间的通讯（队列和管道、生产者与消费者模型）与进程池

一、理论知识

1.1、什么是进程

进程（Process）是计算机中的程序关于某数据集合上的一次运行活动，是系统进行资源分配和调度的基本单位，是操作系统结构的基础。在早期面向进程设计的计算机结构中，进程是程序的基本执行实体；在当代面向线程设计的计算机结构中，进程是线程的容器。程序是指令、数据及其组织形式的描述，进程是程序的实体。

狭义定义：进程是正在运行的程序的实例（an instance of a computer program that is being executed）。

广义定义：进程是一个具有一定独立功能的程序关于某个数据集合的一次运行活动。它是操作系统动态执行的基本单元，在传统的操作系统中，进程既是基本的分配单元，也是基本的执行单元。

- 进程的概念
- 操作系统引入进程概念的原因
- 进程的特征
- 进程与程序的区别

注意：同一个程序执行两次，就会在操作系统中出现两个进程，所以我们可以同时运行一个软件，分别做不同的事情也不会混乱。

1.2、进程调度

要想多个进程交替运行，操作系统必须对这些进程进行调度，这个调度也不是随即进行的，而是需要遵循一定的法则，由此就有了进程的调度算法。

- 首先来服务调度算法
- 短作业优先调度算法
- 时间片轮转法
- 多级反馈队列

1.3、进程的并行与并发

- **并行**：并行是指两者同时执行，比如赛跑，两个人都在不停的往前跑；（资源够用，比如三个线程，四核的 CPU ）
- **并发**：并发是指资源有限的情况下，两者交替轮流使用资源，比如一段路(单核 CPU 资源)同时只能过一个人，A 走一段后，让给 B，B 用完继续给 A ，交替使用，目的是提高效率。

区别:

- **并行**是从微观上，也就是在一个精确的时间片刻，有不同的程序在执行，这就要求必须有多个处理器。
- **并发**是从宏观上，在一个时间段上可以看出是同时执行的，比如一个服务器同时处理多个 session。

1.4、同步、异步、阻塞、非阻塞

1.4.1、状态介绍

在了解其他概念之前，我们首先要了解进程的几个状态。在程序运行的过程中，由于被操作系统的调度算法控制，程序会进入几个状态：就绪，运行和阻塞。

（1）就绪(Ready)状态

当进程已分配到除 CPU 以外的所有必要的资源，只要获得处理机便可立即执行，这时的进程状态称为就绪状态。

（2）执行/运行（Running）状态当进程已获得处理机，其程序正在处理机上执行，此时的进程状态称为执行状态。

（3）阻塞(Blocked)状态正在执行的进程，由于等待某个事件发生而无法执行时，便放弃处理机而处于阻塞状态。引起进程阻塞的事件可有多种，例如，等待 I/O 完成、申请缓冲区不能满足、等待信件(信号)等。

1.4.2、同步和异步

所谓同步就是一个任务的完成需要依赖另外一个任务时，只有等待被依赖的任务完成后，依赖的任务才能算完成，这是一种可靠的任务序列。要么成功都成功，失败都失败，两个任务的状态可以保持一致。

所谓异步是不需要等待被依赖的任务完成，只是通知被依赖的任务要完成什么工作，依赖的任务也立即执行，只要自己完成了整个任务就算完成了。至于被依赖的任务最终是否真正完成，依赖它的任务无法确定，所以它是不可靠的任务序列。

- 例子

1.4.3、阻塞和非阻塞

阻塞和非阻塞这两个概念与程序（线程）等待消息通知(无所谓同步或者异步)时的状态有关。也就是说阻塞与非阻塞主要是程序（线程）等待消息通知时的状态角度来说的

例子

1.5、同步/异步 与 阻塞/非阻塞

1. **同步阻塞形式**

效率最低。拿上面的例子来说，就是你专心排队，什么别的事都不做。

2. **异步阻塞形式**

如果在银行等待办理业务的人采用的是异步的方式去等待消息被触发（通知），也就是领了一张小纸条，假如在这段时间里他不能离开银行做其它的事情，那么很显然，这个人被阻塞在了这个等待的操作上面；

异步操作是可以被阻塞住的，只不过它不是在处理消息时阻塞，而是在等待消息通知时被阻塞。

3. **同步非阻塞形式**

实际上是效率低下的。

想象一下你一边打着电话一边还需要抬头看到底队伍排到你了没有，如果把打电话和观察排队的位置看成是程序的两个操作的话，这个程序需要在这两种不同的行为之间来回的切换，效率可想而知是低下的。

4. **异步非阻塞形式**

效率更高，

因为打电话是你(等待者)的事情，而通知你则是柜台(消息触发机制)的事情，程序没有在两种不同的操作中来回切换。

比如说，这个人突然发觉自己烟瘾犯了，需要出去抽根烟，于是他告诉大堂经理说，排到我这个号码的时候麻烦到外面通知我一下，那么他就没有被阻塞在这个等待的操作上面，自然这个就是异步+非阻塞的方式了。

很多人会把同步和阻塞混淆，是因为很多时候同步操作会以阻塞的形式表现出来，同样的，很多人也会把异步和非阻塞混淆，因为异步操作一般都不会在真正的 IO 操作处被阻塞。

1.6、进程的创建与结束

1.6.1、进程的创建

但凡是硬件，都需要有操作系统去管理，只要有操作系统，就有进程的概念，就需要有创建进程的方式，一些操作系统只为一个应用程序设计，比如微波炉中的控制器，一旦启动微波炉，所有的进程都已经存在。

而对于通用系统（跑很多应用程序），需要有系统运行过程中创建或撤销进程的能力，主要分为 4 中形式创建新的进程：

1. 系统初始化（查看进程 linux 中用 ps 命令，windows 中用任务管理器，前台进程负责与用户交互，后台运行的进程与用户无关，运行在后台并且只在需要时才唤醒的进程，称为守护进程，如电子邮件、web 页面、新闻、打印）

2. 一个进程在运行过程中开启了子进程（如 nginx 开启多进程，os.fork,subprocess.Popen 等）

3. 用户的交互式请求，而创建一个新进程（如用户双击暴风影音）

4. 一个批处理作业的初始化（只在大型机的批处理系统中应用）

无论哪一种，新进程的创建都是由一个已经存在的进程执行了一个用于创建进程的系统调用而创建的。

创建进程

1.6.2、进程的结束

1. 正常退出（自愿，如用户点击交互式页面的叉号，或程序执行完毕调用发起系统调用正常退出，在 linux 中用 exit，在 windows 中用 ExitProcess）
2. 出错退出（自愿，python a.py 中 a.py 不存在）
3. 严重错误（非自愿，执行非法指令，如引用不存在的内存，1/0 等，可以捕捉异常，try...except...）
4. 被其他进程杀死（非自愿，如 kill -9）

2、在 python 中的进程

上面已经了解了很多进程相关的理论知识，运行中的程序就是一个进程。所有的进程都是通过它的父进程来创建的。因此，运行起来的 python 程序也是一个进程，那么我们也可以在程序中再创建进程。多个进程可以实现并发效果，也就是说，当我们的程序中存在多个进程的时候，在某些时候，就会让程序的执行速度变快。以我们之前所学的知识，并不能实现创建进程这个功能，所以我们就需要借助 python 中强大的模块。

2.1、multiprocess 模块

仔细说来，multiprocess 不是一个模块而是 python 中一个操作、管理进程的包。 之所以叫 multi 是取自 multiple 的多功能的意思,在这个包中几乎包含了和进程有关的所有子模块。由于提供的子模块非常多，为了方便大家归类记忆，我将这部分大致分为四个部分：创建进程部分，进程同步部分，进程池部分，进程之间数据共享。

2.1.1、multiprocess.process 模块介绍

process 模块是一个创建进程的模块，借助这个模块，就可以完成进程的创建。

参数介绍：

1. group 参数未使用，值始终为 None
2. target 表示调用对象，即子进程要执行的任务
3. args 表示调用对象的位置参数元组，args=(1,2,'egon',)
4. kwargs 表示调用对象的字典,kwargs={'name':'egon','age':18}
5. name 为子进程的名称

方法介绍：

1. p.start()：启动进程，并调用该子进程中的 p.run()
2. p.run():进程启动时运行的方法，正是它去调用 target 指定的函数，我们自定义类的类中一定要实现该方法
3. p.terminate():强制终止进程 p，不会进行任何清理操作，如果 p 创建了子进程，该子进程就成了僵尸进程，使用该方法需要特别小心这种情况。如果 p 还保存了一个锁那么也将不会被释放，进而导致死锁

4. p.is_alive():如果 p 仍然运行，返回 True 5 p.join([timeout]):主线程等待 p 终止（强调：是主线程处于等的状态，而 p 是处于运行的状态）。timeout 是可选的超时时间，需要强调的是，p.join 只能 join 住 start 开启的进程，而不能 join 住 run 开启的进程

属性介绍：

1. p.daemon：默认值为 False，如果设为 True，代表 p 为后台运行的守护进程，当 p 的父进程终止时，p 也随之终止，并且设定为 True 后，p 不能创建自己的新进程，必须在 p.start()之前设置
2. p.name:进程的名称
3. p.pid：进程的 pid
4. p.exitcode:进程在运行时为 None、如果为–N，表示被信号 N 结束(了解即可)
5. p.authkey:进程的身份验证键,默认是由 os.urandom()随机生成的 32 字符的字符串。这个键的用途是为涉及网络连接的底层进程间通信提供安全性，这类连接只有在具有相同的身份验证键时才能成功（了解即可）

在 windows 中使用 process 模块的注意事项

2.1.2、使用 multiprocess.process 模块开启子进程

在一个 python 进程中开启子进程，start 方法和并发效果。

在 python 中使用 process 模块启动一个子进程

进程的生命周期：

- 主进程：代码从开头运行到结尾就结束
- 子进程：从 start 运行到子进程中的代码都结束
- 开启了子进程的主进程：主进程自己的代码如果长，等待自己的代码执行结束
- 子进程的执行时间长，主进程会在主进程代码执行完毕之后等待子进程执行完毕之后，主进程才结束

2.1.2.1：join 方法

join 方法：感知子进程的结束

2.1.2.2：开启多个子进程

方法一：

View Code

方法二：

View Code

方法三：

View Code

方法四：

多进程往文件中写入内容

开启多进程的另一中方式：

通过继承 Process 类与 run 方法开启进程

2.1.3、多进程之间的数据隔离问题

View Code

2.1.4、守护进程

会随着主进程的结束而结束。

主进程创建守护进程

其一：守护进程会在主进程代码执行结束后就终止

其二：守护进程内无法再开启子进程,否则抛出异常：AssertionError: daemonic processes are not allowed to have children

注意：进程之间是互相独立的，主进程代码运行结束，守护进程随即终止

守护进程

2.1.5、判断子进程是否存活

View Code

2.1.6、多进程实现 socket 聊天并发 server 端

server 端

client 端

3、关于子进程中有 input 报错问题

```
from multiprocessing import Process
def func():
    info = input('>:')        #当子进程中有 input 时会报错，（即子进程中不能有 input）
    print(info)
if __name__ == "__main__":
    Process(target=func).start()
```

说明：注册子进程的函数中不可以有 input，否则会报（EOFError: EOF when reading a line）错！！

4、进程同步

4.1、锁：multiprocessing.Lock

我们千方百计实现了程序的异步，让多个任务可以同时在几个进程中并发处理，他们之间的运行没有顺序，一旦开启也不受我们控制。尽管并发编程让我们能更加充分的利用 IO 资源，但是也给我们带来了新的问题。

当多个进程使用同一份数据资源的时候，就会引发数据安全或顺序混乱问题。

多进程抢占输出资源

使用锁维护执行顺序

上面这种情况虽然使用加锁的形式实现了顺序的执行，但是程序又重新变成串行了，这样确实会浪费了时间，却保证了数据的安全。

接下来，我们以模拟抢票为例，来看看数据安全的重要性。

⊞ 多进程同时抢购余票

⊞ 使用锁来保证数据安全

- 加锁可以保证多个进程修改同一块数据时，同一时间只能有一个任务可以进行修改，即串行的修改，没错，速度是慢了，但牺牲了速度却保证了数据安全。 虽然可以用文件共享数据实现进程间通信，但问题是：

1.效率低（共享数据基于文件，而文件是硬盘上的数据）

2.需要自己加锁处理

- 因此我们最好找寻一种解决方案能够兼顾：

1、效率高（多个进程共享一块内存的数据）

2、帮我们处理好锁问题。这就是 mutiprocessing 模块为我们提供的基于消息的 IPC 通信机制：队列和管道。 队列和管道都是将数据存放于内存中队列又是基于（管道+锁）实现的，可以让我们从复杂的锁问题中解脱出来， 我们应该尽量避免使用共享数据，尽可能使用消息传递和队列，避免处理复杂的同步和锁问题，而且在进程数目增多时，往往可以获得更好的可获展性。

4.2、信号量：Semaphore（了解）

互斥锁同时只允许一个线程更改数据，而信号量 Semaphore 是同时允许一定数量的线程更改数据 。 假设商场里有 4 个迷你唱吧，所以同时可以进去 4 个人，如果来了第五个人就要在外面等待，等到有人出来才能再进去玩。 实现： 信号量同步基于内部计数器，每调用一次 acquire()，计数器减 1；每调用一次 release()，计数器加 1.当计数器为 0 时，acquire()调用被阻塞。这是迪科斯彻（Dijkstra）信号量概念 P()和 V()的 Python 实现。信号量同步机制适用于访问像服务器这样的有限资源。 信号量与进程池的概念很像，但是要区分开，信号量涉及到加锁的概念

⊞ 信号量例子

4.3、事件：Event（了解）

python 线程的事件用于主线程控制其他线程的执行，事件主要提供了三个方法 set、wait、clear。

事件处理的机制：全局定义了一个“Flag”，如果“Flag”值为 False，那么当程序执行 event.wait 方法时就会阻塞，如果“Flag”值为 True，那么 event.wait 方法时便不再阻塞。

clear：将“Flag”设置为 False set：将“Flag”设置为 True

⊞ 事件的方法与使用

⊞ 红绿灯事件

五、进程间的通讯

5.1、队列

创建共享的进程队列，Queue 是多进程安全的队列，可以使用 Queue 实现多进程之间的数据传递。

⊞ 方法介绍

⊞ 其他方法（了解）

代码实例

⊞ 队列用法
⊞ 进程间使用队列通讯
⊞ 批量生产数据放入队列在批量获取结果

5.2、生产者与消费者模型

在并发编程中使用生产者和消费者模式能够解决绝大多数并发问题。该模式通过平衡生产线程和消费线程的工作能力来提高程序的整体处理数据的速度。

为什么要使用生产者和消费者模式

在线程世界里，生产者就是生产数据的线程，消费者就是消费数据的线程。在多线程开发当中，如果生产者处理速度很快，而消费者处理速度很慢，那么生产者就必须等待消费者处理完，才能继续生产数据。同样的道理，如果消费者的处理能力大于生产者，那么消费者就必须等待生产者。为了解决这个问题于是引入了生产者和消费者模式。

什么是生产者消费者模式

生产者消费者模式是通过一个容器来解决生产者和消费者的强耦合问题。生产者和消费者彼此之间不直接通讯，而通过阻塞队列来进行通讯，所以生产者生产完数据之后不用等待消费者处理，直接扔给阻塞队列，消费者不找生产者要数据，而是直接从阻塞队列里取，阻塞队列就相当于一个缓冲区，平衡了生产者和消费者的处理能力。

⊞ 基于队列实现生产者消费者模型

此时的问题是主进程永远不会结束，原因是：生产者 p 在生产完后就结束了，但是消费者 c 在取空了 q 之后，则一直处于死循环中且卡在 q.get()这一步。

解决方式无非是让生产者在生产完毕后，往队列中再发一个结束信号，这样消费者在接收到结束信号后就可以 break 出死循环。

⊞ 改良版生产者与消费者模型

注意：结束信号 None，不一定要由生产者发，主进程里同样可以发，但主进程需要等生产者结束后才应该发送该信号

⊞ 主进程在生产者生产完毕后发送结束信号 None

但上述解决方式，在有多个生产者和多个消费者时，我们则需要用一个很 low 的方式去解决

⊞ 多个消费者的例子：有几个消费者就需要发送几次结束信号

5.3、JoinableQueue([maxsize])

创建可连接的共享进程队列。这就像是一个 Queue 对象，但队列允许项目的使用者通知生产者项目已经被成功处理。通知进程是使用共享的信号和条件变量来实现的。

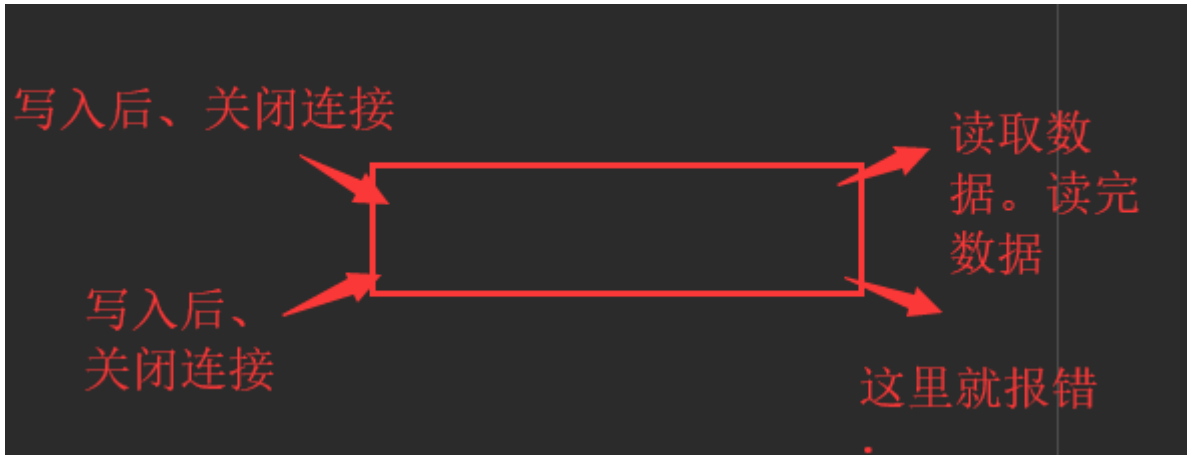
⊞ 方法介绍
⊞ JoinableQueue 队列实现消费之生产者模型

六、管道（了解即可）

⊞ 介绍
⊞ 管道的基础使用

应该特别注意管道端点的正确管理问题。**如果是生产者或消费者中都没有使用管道的某个端点，就应将它关闭。**这也说明了为何在**生产者中关闭了管道的输出端，在消费者中关闭管道的输入端。**如果忘记执行这些步骤，程序可能在消费者中的 recv（）操作上挂起。管道是由操作系统进行引用计数的，必须在所有进程中关闭管道后才能生成 EOFError 异常。因此，在生产者中关闭管道不会有任何效果，除非消费者也关闭了相同的管道端点。

⊞ 引发 EOFError 与用两端通讯实现解决 EOFError 报错问题



⊞ 使用管道实现生产者与消费者模型

⊞ 多个消费者之间的竞争问题带来的数据不安全问题（加锁解决）

七、进程之间的数据共享（了解即可）

展望未来，基于消息传递的并发编程是大势所趋

即便是使用线程，推荐做法也是将程序设计为大量独立的线程集合，通过消息队列交换数据。

这样极大地减少了对使用锁定和其他同步手段的需求，还可以扩展到分布式系统中。

但进程间应该尽量避免通信，即便需要通信，也应该选择进程安全的工具来避免加锁带来的问题。

以后我们会尝试使用数据库来解决现在进程之间的数据共享问题。

⊞ Manager 模块介绍

⊞ Manager 模块的例子

八、进程池（重要）multiprocess.Pool 模块

为什么要有进程池?进程池的概念。

在程序实际处理问题过程中，忙时会有成千上万的任务需要被执行，闲时可能只有零星任务。那么在成千上万个任务需要被执行的时候，我们就需要去创建成千上万个进程么？首先，创建进程需要消耗时间，销毁进程也需要消耗时间。第二即便开启了成千上万的进程，操作系统也不能让他们同时执行，这样反而会影响程序的效率。因此我们不能无限制的根据任务开启或者结束进程。那么我们要怎么做呢?

在这里，要给大家介绍一个进程池的概念，定义一个池子，在里面放上固定数量的进程，有需求来了，就拿一个池中的进程来处理任务，等到处理完毕，进程并不关闭，而是将进程再放回进程池中继续等待任务。如果有很多任务需要执行，池中的进程数量不够，任务就要等待之前的进程执行任务完毕归

来，拿到空闲进程才能继续执行。也就是说，池中进程的数量是固定的，那么同一时间最多有固定数量的进程在运行。这样不会增加操作系统的调度难度，还节省了开闭进程的时间，也一定程度上能够实现并发效果。

更高级的进程池（python 中没有）：

n(下限) m（上限）

下线设置 3 个进程、上线 20 个进程，如果访问量增加了，三个进程处理不过来了就会一点一点增加进程，直到达到进程池设置的上线就不会再加了，如果用不到的太多的进程后会一点点的减少，直到底线

8.1、multiprocess.Pool 模块

```
Pool([numprocess [,initializer [, initargs]]]):创建进程池

1 numprocess:要创建的进程数，如果省略，将默认使用 cpu_count()的值
2 initializer：是每个工作进程启动时要执行的可调用对象，默认为 None
3 initargs：是要传给 initializer 的参数组
```

进程池的基础使用之 map 方法

map 方法说明：默认异步的执行任务，且自带 close 和 join 方法

进程池与进程之间的效率对比

进程池的 map 方法同时执行 a 任务与 b 任务和进程 map 方法传参

进程池的同步执行与异步执行

进程池的返回值

进程池的回调函数

使用进程池实现 socker server 端并发

socket client 端

8.2、进程池的常用方法说明：

1、 p.apply(func [, args [, kwargs]]):在一个池工作进程中执行 func(*args,**kwargs),然后返回结果。

'''需要强调的是：此操作并不会在所有池工作进程中并执行 func 函数。如果要通过不同参数并发地执行 func 函数，必须从不同线程调用 p.apply()函数或者使用 p.apply_async()'''

2、p.apply_async(func [, args [, kwargs]]):在一个池工作进程中执行 func(*args,**kwargs),然后返回结果。

'''此方法的结果是 AsyncResult 类的实例，callback 是可调用对象，接收输入参数。当 func 的结果变为可用时，将理解传递给 callback。callback 禁止执行任何阻塞操作，否则将接收其他异步操作中的结果。''' 、

3、p.close():关闭进程池，防止进一步操作。如果所有操作持续挂起，它们将在工作进程终止前完成

4、P.jion():等待所有工作进程退出。此方法只能在 close（）或 teminate()之后调用

8.3、进程池的其他方法说明（了解即可）

1、 方法 apply_async()和 map_async（）的返回值是 AsyncResul 的实例 obj。实例具有以下方法

2、 obj.get():返回结果，如果有必要则等待结果到达。timeout 是可选的。如果在指定时间内还没有到达，将引发一场。如果远程操作中引发了异常，它将在调用此方法时再次被引发。

3、 obj.ready():如果调用完成，返回 True

4、 obj.successful():如果调用完成且没有引发异常，返回 True，如果在结果就绪之前调用此方法，引发异常
5、 obj.wait([timeout]):等待结果变为可用。
6、 obj.terminate()：立即终止所有工作进程，同时不执行任何清理或结束任何挂起工作。如果 p 被垃圾回收，将自动调用此函数

7. 信号量的实现机制

第一种方法：信号量就是控制同一时刻并发执行的任务数

信号量：互斥锁+容器 容器里同时最大可以存放五把钥匙，谁先拿到钥匙并释放后释放后，外面的才能继续抢钥匙

十个线程，五把钥匙，一开启肯定有五个线程能拿到钥匙，只有这五把钥匙谁先解锁了，之后的五个线程才能有抢钥匙的机会。

```
from threading import Thread,Semaphore,current_thread
import time,random
sm=Semaphore(5) #信号量       互斥锁+队列     相当于一个容器，容器里同时最大可以存在五个钥匙，同时也只能有五个线程
                                                #谁先拿到并释放后，下一个才能拿到钥匙
def task():
    with sm:
        print('%s 正在上厕所' %current_thread().name)
        time.sleep(random.randint(1,4))
if __name__ == '__main__':
    for i in range(20):
        t=Thread(target=task)
        t.start()
```

第二种方法：python 并发——信号量

信号量通常用于保护数量有限的资源，例如数据库服务器。在资源数量固定的任何情况下，都应该使用有界信号量。在生成任何工作线程前，应该在主线程中初始化信号量。

工作线程生成后，当需要连接服务器时，这些线程将调用信号量的 acquire 和 release 方法：

使用有界信号量能减少这种编程错误：信号量的释放次数多于其请求次数。

```
from threading import BoundedSemaphore
maxconnections = 5
pool_sema = BoundedSemaphore(value=maxconnections)
```

```
with pool_sema:
    conn = connectdb()
    try:
        do_some()
    finally:
        conn.close()
```

第三种方法：Python 下进程同步之互斥锁、信号量、事件机制

一、锁机制：multiprocess.Lock

上篇博客中，我们千方百计实现了程序的异步，让多个任务同时在几个进程中并发处理，但它们之间的运行没有顺序。尽管并发编程让我们能更加充分的利用 io 资源，但是也给我我们带来了新问题，**多个进程使用同一份数据资源的时候，就会引发数据安全或顺序混乱问题.** 例：

```
# 多进程抢占输出资源

from multiprocessing import Process
from os import getpid
from time import sleep
from random import random


def work(i):
    print("%s: %s is running" %(i, getpid()))
    sleep(random())
    print("%s: %s is done" %(i, getpid()))


if __name__ == '__main__':
    for i in range(5):
        p = Process(target=work, args=(i,))
        p.start()
```

使用互斥锁维护执行顺序：

```
# 使用锁机制维护执行顺序
```

```
2.
3. from multiprocessing import Process, Lock
4. from os import getpid
5. from time import sleep
6. from random import random
7.
8.
9. def work(lock, i):
10.     lock.acquire()    # ⚠开锁进门
11.     print("%s: %s is running" %(i, getpid()))
12.     sleep(random())
13.     print("%s: %s is done" %(i, getpid()))
14.     lock.release()    # ⚠出门，归还钥匙
15.
16. if __name__ == '__main__':
17.     lock = Lock()    # ⚠实例化一把锁，一把钥匙
18.
19.     for i in range(5):
20.         p = Process(target=work, args=(lock, i))
21.         p.start()
```

上面这种情况虽然使用加锁的形式实现了顺序的执行，却使得进程变成了串行执行，这样确实会浪费些时间，但是保证了数据的安全.

接下来我们以模拟抢票为例，来看看数据安全的重要性:

```
1. # 文件 db 的内容为：{"count": 1} （注意："count"一定要用双引号，否者 json 无法识别）
2. # 并发运行，效率高，但是：竞争抢票只读写同一个文件，造成数据写入错乱
3.
4. from multiprocessing import Process
5. from time import sleep
6. from json import load, dump
7.
8.
9. def search():
```

```
    dct = load(open('db'))
    print("\033[43m剩余票数:%s\033[0m" %dct['count'])


def get():
    dct = load(open('db'))
    sleep(0.01)  # 模拟读数据的网络延迟

    if dct['count'] > 0:
        dct['count'] -=1
        sleep(0.02)  # 模拟写数据的网络延迟
        dump(dct, open('db', 'w'))
        print("\033[32m购票成功\033[0m")


def tack():
    search()
    get()


if __name__ == '__main__':
    for i in range(100):    # 模拟并发100个客户端抢票
        p =Process(target=tack)
        p.start()

# 100个进程同时读写同一个文件，可能会报错
```

使用锁机制保护数据安全：

```
# 文件db的内容为：{"count": 1}（注意："count"一定要用双引号，否者json无法识别）
# 同步运行，效率低，但是可以保证数据安全

from multiprocessing import Process, Lock
```

```
from time import sleep
from json import load, dump


def search():
    dct = load(open('db'))
    print("\033[43m剩余票数:%s\033[0m" %dct['count'])


def get():
    dct = load(open('db'))
    sleep(0.01)  # 模拟读数据的网络延迟

    if dct['count'] > 0:
        dct['count'] -=1
        sleep(0.02)  # 模拟写数据的网络延迟
        dump(dct, open('db', 'w'))
        print("\033[32m购票成功\033[0m")


def tack():
    search()
    lock.acquire()    # 上锁
    get()
    lock.release()    # 释放


if __name__ == '__main__':
    lock = Lock()    # 实例化一把锁

    for i in range(100):    # 模拟并发100个客户端抢票
```

```
        p =Process(target=tack)
        p.start()

# 可能会遇到 json.decoder.JSONDecodeError 报错
```

使用锁机制可以保证多个进程想要修改同一块数据时，在同一时间点只能有一个进程进行修改，即串行的修改，牺牲速度而保证安全性.

虽然可以用文件共享数据实现进程间通讯，但问题是：1. 效率低（共享数据基于文件，而文件是硬盘上的数据）；2. 需要自己加锁处理。因此我们最好找寻一种解决方案能够兼顾：1. 效率高（多个进程共享一块内存的数据）；2. 帮我们处理好锁的问题。这就是 mutiprocessing 模块为我们提供的基于消息的 IPC 通信机制：队列和管道

队列和管道都是将数据存放于内存中，而队列又是基于（管道+锁）实现的，可以让我们从复杂的问题中解脱出来，我们应该尽量避免使用共享数据，尽可能使用消息传递和队列，避免处理复杂的同步和锁问题，而且在进程数目增多时，往往可以获得更好的可扩展性.

二、信号量：multiprocessing.Semaphore

1. 信号量 Semaphore 允许一定数量的线程在同一时间点更改同一块数据，很形象的例子：厕所里有 3 个坑，同时可以 3 个人蹲，如果来了第 4 个人就要在外面等待，直到某个人出来了才能进去.

2.信号量同步基于内部计数器，每调用一次 acquire()，计数器-1，每调用一次 release()，计数器+1，当计数器为 0 时，acquire()调用被阻塞。这是迪克斯彻（Dijkstra）信号量概念 P()和 V()的 Python 实现，信号同步机制适用于访问像服务器这样的有限资源.

3.注意：信号量与进程池的概念很像，要注意区分，信号量涉及到加锁的概念.

- 方法

obj = Semaphore(4)：实例化一把锁，4 把钥匙，钥匙可以指定（最小 0，最大未知）

obj.acquire()：加锁，锁住下方的语句，直到遇到 obj.release()方可解锁

obj.release()：释放，释放 obj.acquire()和 obj.release()的语句数据，此方法如果写在加锁之前，便多了一把钥匙

- 基本用法

```
# 一把锁，多把钥匙

from multiprocessing import Semaphore

s = Semaphore(2)     # 实例化一把锁，配 2 把钥匙

s.release()      # 可以先释放钥匙，变成 3 把钥匙
s.release()      # 再释放一把钥匙，现在变成了 4 把钥匙

s.acquire()
```

```
print(1)

s.acquire()
print(2)

# 加上先释放的钥匙，我门总共有 4 把钥匙
s.acquire() # 顺利执行
print(3)

s.acquire() # 顺利执行
print(4)

# 钥匙全部被占用
s.acquire() # 此处阻塞住，等待钥匙的释放
print(5)    # 不会被打印
```

- 进阶：小黑屋

```
# 小黑屋

from multiprocessing import Process, Semaphore
from time import sleep
from random import uniform


def func(sem, i):
    sem.acquire()
    print("第%s 个人进入了小黑屋" % i)
    sleep(uniform(1, 3))
    print("第%s 个人走出了小黑屋" % i)
    sem.release()


```

```
if __name__ == '__main__':
    sem = Semaphore(5)  # 初始化一把锁，配 5 把钥匙

    for i in range(10): # 启动 10 个子进程，最多只能 5 个人同在小黑屋中
        p = Process(target=func, args=(sem, i))
        p.start()
```

三、事件机制：multiprocessing.Event

Python 线程的事件用于主线程控制其它线程的执行，事件主要提供了三个方法：set、wait，clear

事件处理机制：全局定义了一个“Flag”，如果“Flag”值为 False，程序执行 wait()方法会被阻塞；如果“Flag”值为 True，程序执行 wait()方法便不会被阻塞.

- 方法

obj.is_set()：默认值为 False，事件是通过此方法的 bool 值去标示 wait()的阻塞状态

obj.set()：将 is_set()的 bool 值改为 True

obj.clear()：将 is_set()的 bool 值改为 False

obj.wait()：is_set()的值为 False 时阻塞，否则不阻塞

- **实例：模拟红绿灯**

```
# 模拟红绿灯

from multiprocessing import Process, Event
import time
import random


def Tra(e):
    print("\033[32m 绿灯亮\033[0m")
    e.set()

    while 1:
        if e.is_set():
            time.sleep(3)
            print("\033[31m 红灯亮\033[0m")
```

```
            e.clear()
        else:
            time.sleep(3)
            print("\033[32m 绿灯亮\033[0m")
            e.set()


def Car(e, i):
    e.wait()
    print("第%s 辆小汽车过去了" % i)


if __name__ == '__main__':
    e = Event()

    tra = Process(target=Tra, args=(e,))
    tra.start()

    for i in range(100):    # 模拟一百辆小汽车
        time.sleep(0.5)
        car = Process(target=Car, args=(e, i))
        car.start()
```

8. 共享锁和排他锁

共享锁有时称为"读锁", 而排他锁有时称为"写锁"

共享锁（读锁）和排他锁（写锁）

共享锁（S 锁）：共享（S）用于不更改或不更新数据的操作（只读操作），如 SELECT 语句。

如果事务 T 对数据 A 加上共享锁后，则其他事务只能对 A 再加共享锁，不能加排他锁。获准共享锁的事务只能读数据，不能修改数据。

排他锁（X 锁）：用于数据修改操作，例如 INSERT、UPDATE 或 DELETE。确保不会同时同一资源进行多重更新。

如果事务 T 对数据 A 加上排他锁后，则其他事务不能再对 A 加任任何类型的封锁。获准排他锁的事务既能读数据，又能修改数据。

我们在操作数据库的时候，可能会由于并发问题而引起的数据的不一致性（数据冲突）

乐观锁

乐观锁不是数据库自带的，需要我们自己去实现。乐观锁是指操作数据库时(更新操作)，想法很乐观，认为这次的操作不会导致冲突，在操作数据时，并不进行任何其他的特殊处理（也就是不加锁），而在进行更新后，再去判断是否有冲突了。

通常实现是这样的：在表中的数据进行操作时(更新)，先给数据表加一个版本(version)字段，每操作一次，将那条记录的版本号加 1。也就是先查询出那条记录，获取出 version 字段,如果要对那条记录进行操作(更新),则先判断此刻 version 的值是否与刚刚查询出来时的 version 的值相等，如果相等，则说明这段期间，没有其他程序对其进行操作，则可以执行更新，将 version 字段的值加 1；如果更新时发现此刻的 version 值与刚刚获取出来的 version 的值不相等，则说明这段期间已经有其他程序对其进行操作了，则不进行更新操作。

mysql 锁机制分为表级锁和行级锁，本文就和大家分享一下我对 mysql 中行级锁中的共享锁与排他锁进行分享交流。

共享锁又称为读锁，简称 S 锁，顾名思义，共享锁就是多个事务对于同一数据可以共享一把锁，都能访问到数据，但是只能读不能修改。

排他锁又称为写锁，简称 X 锁，顾名思义，排他锁就是不能与其他所并存，如一个事务获取了一个数据行的排他锁，其他事务就不能再获取该行的其他锁，包括共享锁和排他锁，但是获取排他锁的事务是可以对数据就行读取和修改。

对于共享锁大家可能很好理解，就是多个事务只能读数据不能改数据，对于排他锁大家的理解可能就有些差别，我当初就犯了一个错误，以为排他锁锁住一行数据后，其他事务就不能读取和修改该行数据，其实不是这样的。排他锁指的是一个事务在一行数据加上排他锁后，其他事务不能再在其上加其他的锁。mysql InnoDB 引擎默认的修改数据语句，update,delete,insert 都会自动给涉及到的数据加上排他锁，select 语句默认不会加任何锁类型，如果加排他锁可以使用 select ...for update 语句，加共享锁可以使用 select ... lock in share mode 语句。所以加过排他锁的数据行在其他事务种是不能修改数据的，也不能通过 for update 和 lock in share mode 锁的方式查询数据，但可以直接通过 select ...from...查询数据，因为普通查询没有任何锁机制。

1、什么是共享锁

共享锁又称为读锁。

从多线程的角度来讲，共享锁允许多个线程同时访问资源，但是对写资源只能又一个线程进行。

从事务的角度来讲，若事务 T 对数据 A 加上共享锁，则事务 T 只能读 A； 其他事务也只能对数据 A 加共享锁，而不能加排他锁，直到事务 T 释放 A 上的 S 锁。这就保证了其他事务可以读 A，但是在事务 T 释放 A 上的共享锁之前，不能对 A 做任何修改。

2、什么是排它锁

排他锁又成为写锁。

从多线程的角度来讲，在访问共享资源之前对进行加锁操作，在访问完成之后进行解锁操作。 加锁后，任何其他试图再次加锁的线程会被阻塞，直到当前进程解锁。如果解锁时有一个以上的线程阻塞，那么所有该锁上的线程都被编程就绪状态， 第一个变为就绪状态的线程又执行加锁操作，那么其他的线程又会进入等待。 在这种方式下，只有一个线程能够访问被互斥锁保护的资源。

从事务的角度来讲，若事务 T 对数据对象 A 加上排它锁，则只允许 T 读取和修改数据 A，其他任何事务都不能再对 A 加任何类型的锁，直到事务 T 释放 X 锁。它可以防止其他事务获取资源上的锁，直到事务末尾释放锁。

InnoDB 中的行锁

InnoDB 实现了以下两种类型的行锁：

共享锁（S）： 允许一个事务去读一行，阻止其他事务获得相同数据集的排他锁。

排他锁（X）： 允许获得排他锁的事务更新数据，阻止其他事务取得相同数据集的共享读锁和排他写锁。

另外，为了允许行锁和表锁共存，实现多粒度锁机制，InnoDB 还有两种内部使用的意向锁（Intention Locks），这两种意向锁都是表锁。

意向共享锁（IS）： 事务打算给数据行加行共享锁，事务在给一个数据行加共享锁前必须先取得该表的 IS 锁。

意向排他锁（IX）： 事务打算给数据行加行排他锁，事务在给一个数据行加排他锁前必须先取得该表的 IX 锁。

请求锁 是否兼容当前锁	X	IX	S	IS
X	冲突	冲突	冲突	冲突
IX	冲突	兼容	冲突	兼容
S	冲突	冲突	兼容	兼容
IS	冲突	兼容	兼容	兼容

如果一个事务请求的锁模式与当前的锁兼容，InnoDB 就将请求的锁授予该事务；反之，如果两者不兼容，该事务就要等待锁释放。

意向锁是 InnoDB 自动加的，不需用户干预。对于 UPDATE、DELETE 和 INSERT 语句，InnoDB 会自动给涉及数据集加排他锁（X）；对于普通 SELECT 语句，InnoDB 不会加任何锁；事务可以通过以下语句显示给记录集加共享锁或排他锁。

共享锁（S）：

```
SELECT * FROM table_name WHERE ... LOCK IN SHARE MODE
```

排他锁（X）：

```
SELECT * FROM table_name WHERE ... FOR UPDATE
```

用 SELECT ... IN SHARE MODE 获得共享锁，主要用在需要数据依存关系时来确认某行记录是否存在，并确保没有人对这个记录进行 UPDATE 或者 DELETE 操作。但是如果当前事务也需要对该记录进行更新操作，则很有可能造成死锁，对于锁定行记录后需要进行更新操作的应用，应该使用 SELECT... FOR UPDATE 方式获得排他锁。

使用 Python 实现

1、代码实现

不多说，直接上代码：

```
# -*- coding: utf-8 -*-import threadingclass Source:

# 队列成员标识
__N = None

# 排他锁
__X = 0
# 意向排他锁
__IX = 1
# 共享锁标识
__S = 2
# 意向共享标识
__IS = 3

# 同步排他锁
__lockX = threading.Lock()
# 事件通知
__events = [
threading.Event(),
threading.Event(),
threading.Event(),
threading.Event()
]
# 事件通知队列
__eventsQueue = [
[],
[],
[],
[]
]
# 事件变更锁
```

```
__eventsLock = [
threading.Lock(),
threading.Lock(),
threading.Lock(),
threading.Lock()
]
# 相互互斥的锁
__mutexFlag = {}

# 锁类
class __ChildLock:

# 锁标识
__flag = 0
# 锁定的资源
__source = None

def __init__(self, source, flag):
self.__flag = flag
self.__source = source

# 加锁
def lock(self):
self.__source.lock(self.__flag)

# 解锁
def unlock(self):
self.__source.unlock(self.__flag)

def __init__(self):
self.__initMutexFlag()
```

```
63.  self.__initEvents()
64.
65.  # 不建议直接在外面使用，以免死锁
66.  def lock(self, flag):
67.  # 如果是排他锁，先进进行枷锁
68.  if flag == self.__X: self.__lockX.acquire()
69.  self.__events[flag].wait()
70.  self.__lockEvents(flag)
71.
72.  # 不建议直接在外面使用，以免死锁
73.  def unlock(self, flag):
74.  self.__unlockEvents(flag)
75.  if flag == self.__X: self.__lockX.release()
76.
77.  # 获取相互互斥
78.  def __getMutexFlag(self, flag):
79.  return self.__mutexFlag[flag]
80.
81.  def __initMutexFlag(self):
82.  self.__mutexFlag[self.__X] = [self.__X, self.__IX, self.__S, self.__IS]
83.  self.__mutexFlag[self.__IX] = [self.__X, self.__S]
84.  self.__mutexFlag[self.__S] = [self.__X, self.__IX]
85.  self.__mutexFlag[self.__IS] = [self.__X]
86.
87.  def __initEvents(self):
88.  for event in self.__events:
89.  event.set()
90.
91.  # 给事件加锁， 调用 wait 时阻塞
92.  def __lockEvents(self, flag):
93.  mutexFlags = self.__getMutexFlag(flag)
```

```
94.  for i in mutexFlags:
95.  # 为了保证原子操作，加锁
96.  self.__eventsLock[i].acquire()
97.  self.__eventsQueue[i].append(self.__N)
98.  self.__events[i].clear()
99.  self.__eventsLock[i].release()
100. # 给事件解锁， 调用 wait 不阻塞
101. def __unlockEvents(self, flag):
102. mutexFlags = self.__getMutexFlag(flag)
103. for i in mutexFlags:
104. # 为了保证原子操作，加锁
105. self.__eventsLock[i].acquire()
106. self.__eventsQueue[i].pop(0)
107. if len(self.__eventsQueue[i]) == 0: self.__events[i].set()
108. self.__eventsLock[i].release()
109.
110. # 获取锁
111. def __getLock(self, flag):
112. lock = self.__ChildLock(self, flag)
113. lock.lock()
114. return lock
115.
116. # 获取 X 锁
117. def lockX(self):
118. return self.__getLock(self.__X)
119.
120. # 获取 IX 锁
121. def lockIX(self):
122. return self.__getLock(self.__IX)
123.
124. # 获取 S 锁
```

```
def lockS(self):
return self.__getLock(self.__S)

# 获取 IS 锁
def lockIS(self):
return self.__getLock(self.__IS)
```

使用方式：

```
from lock import Source
# 初始化一个锁资源
source = Source()
# 获取资源的X 锁，获取不到则线程被阻塞，获取到了继续往下执行
lock = source.lockX()
lock.unlock()

lock = source.lockIX()
lock.unlock()
 lock = source.lockS()
 lock.unlock()
 lock = source.lockIS()
 lock.unlock()
```

2、实现思路

以 S 锁为例，获取锁的步骤如下：

- 检测 S 锁是否可以取到，取到了话继续执行，没有取到则阻塞，等待其他线程解锁唤醒。
- 获取与 S 锁相互冲突的锁（IX，X），并将 IX 锁和 X 锁 锁住，后续想获得 IX 锁或者 X 锁的线程就会被阻塞。
- 向 IX 锁和 X 锁的标识队列插入标识，如果此时另外一个线程拿到了 IS 锁，则会继续想 IX 锁队列标识插入标识。
- 完成加锁，返回 S 锁。

以 S 锁为例，解锁的步骤如下：

- 获取与 S 锁相互冲突的锁（IX，X），向 IX 锁和 X 锁的标识队列移除一个标识。
- 判断 IX 锁和 X 锁队列标识是否为空，如果不为空，则继续锁定，为空则解锁并唤醒被 IX 锁和 X 锁阻塞的线程。
- 完成 S 锁解锁。

3、锁兼容测试

测试代码

```
# -*- coding: utf-8 -*-
import threading
import time
from lock import Source

# 初始化资源
source= Source()
maplockname=['X','IX','S','IS']

class MyThread(threading,Thread):
flag = None

def __init__(self, flag):
super().__init__()
self.flag = flag def run(self):
lock = self.lock()
time1 = time.time()
strtime1 = time.strftime("%Y-%m-%d %H:%M:%S", time.localtime(time1))
print('我拿到 %s 锁，开始执行了喔，现在时间是 %s' % (maplockname[self.flag], strtime1))
time.sleep(1)

time2 = time.time()
strtime2 = time.strftime("%Y-%m-%d %H:%M:%S", time.localtime(time2))
print('我释放 %s 锁，结束执行了，现在时间是 %s' % (maplockname[self.flag], strtime2))
lock.unlock()
def lock(self):
if self.flag == 0:
return source.lockX()
elif self.flag == 1:
```

```
30.  return source.lockIX()
31.  elif self.flag == 2:
32.  return source.lockS()
33.  else:
34.  return source.lockIS()
35.  
36.  def unlock(self, lock):
37.  lock.unlock()def test_lock():
38.  for x in range(0, 4):
39.  for y in range(0, 4):
40.  time1 = time.time()
41.  thread1 = MyThread(x)
42.  thread2 = MyThread(y)
43.  thread1.start()
44.  thread2.start()
45.  thread1.join()
46.  thread2.join()
47.  
48.  time2 = time.time()
49.  difftime = time2 - time1
50.  if difftime > 2:
51.  print('%s 锁和 %s 锁 冲突了! ' % (maplockname[x], maplockname[y]))
52.  elif difftime > 1:
53.  print('%s 锁和 %s 锁 没有冲突! ' % (maplockname[x], maplockname[y]))
54.  print('')if __name__ == '__main__':
55.  test_lock()
```

运行结果：

```
1. 我拿到 X 锁了，开始执行了喔，现在时间是 2019-02-17 18:38:09
2. 我释放 X 锁了，结束执行了，现在时间是 2019-02-17 18:38:10
3. 我拿到 X 锁了，开始执行了喔，现在时间是 2019-02-17 18:38:10
4. 我释放 X 锁了，结束执行了，现在时间是 2019-02-17 18:38:11
```

```
X 锁和 X 锁 冲突了!

我拿到 X 锁了，开始执行了喔，现在时间是 2019-02-17 18:38:11
我释放 X 锁了，结束执行了，现在时间是 2019-02-17 18:38:12
我拿到 IX 锁了，开始执行了喔，现在时间是 2019-02-17 18:38:12
我释放 IX 锁了，结束执行了，现在时间是 2019-02-17 18:38:13
X 锁和 IX 锁 冲突了!

我拿到 X 锁了，开始执行了喔，现在时间是 2019-02-17 18:38:13
我释放 X 锁了，结束执行了，现在时间是 2019-02-17 18:38:14
我拿到 S 锁了，开始执行了喔，现在时间是 2019-02-17 18:38:14
我释放 S 锁了，结束执行了，现在时间是 2019-02-17 18:38:15
X 锁和 S 锁 冲突了!

我拿到 X 锁了，开始执行了喔，现在时间是 2019-02-17 18:38:15
我释放 X 锁了，结束执行了，现在时间是 2019-02-17 18:38:16
我拿到 IS 锁了，开始执行了喔，现在时间是 2019-02-17 18:38:16
我释放 IS 锁了，结束执行了，现在时间是 2019-02-17 18:38:17
X 锁和 IS 锁 冲突了!

我拿到 IX 锁了，开始执行了喔，现在时间是 2019-02-17 18:38:17
我释放 IX 锁了，结束执行了，现在时间是 2019-02-17 18:38:18
我拿到 X 锁了，开始执行了喔，现在时间是 2019-02-17 18:38:18
我释放 X 锁了，结束执行了，现在时间是 2019-02-17 18:38:19
IX 锁和 X 锁 冲突了!

我拿到 IX 锁了，开始执行了喔，现在时间是 2019-02-17 18:38:19
我拿到 IX 锁了，开始执行了喔，现在时间是 2019-02-17 18:38:19
我释放 IX 锁了，结束执行了，现在时间是 2019-02-17 18:38:20
我释放 IX 锁了，结束执行了，现在时间是 2019-02-17 18:38:20
IX 锁和 IX 锁 没有冲突!
```

```
我拿到 IX 锁了，开始执行了喔，现在时间是 2019-02-17 18:38:20
我释放 IX 锁了，结束执行了，现在时间是 2019-02-17 18:38:21
我拿到 S 锁了，开始执行了喔，现在时间是 2019-02-17 18:38:21
我释放 S 锁了，结束执行了，现在时间是 2019-02-17 18:38:22
IX 锁和 S 锁 冲突了!

我拿到 IX 锁了，开始执行了喔，现在时间是 2019-02-17 18:38:22
我拿到 IS 锁了，开始执行了喔，现在时间是 2019-02-17 18:38:22
我释放 IX 锁了，结束执行了，现在时间是 2019-02-17 18:38:23
我释放 IS 锁了，结束执行了，现在时间是 2019-02-17 18:38:23
IX 锁和 IS 锁 没有冲突!

我拿到 S 锁了，开始执行了喔，现在时间是 2019-02-17 18:38:23
我释放 S 锁了，结束执行了，现在时间是 2019-02-17 18:38:24
我拿到 X 锁了，开始执行了喔，现在时间是 2019-02-17 18:38:24
我释放 X 锁了，结束执行了，现在时间是 2019-02-17 18:38:25
S 锁和 X 锁 冲突了!

我拿到 S 锁了，开始执行了喔，现在时间是 2019-02-17 18:38:25
我释放 S 锁了，结束执行了，现在时间是 2019-02-17 18:38:26
我拿到 IX 锁了，开始执行了喔，现在时间是 2019-02-17 18:38:26
我释放 IX 锁了，结束执行了，现在时间是 2019-02-17 18:38:27
S 锁和 IX 锁 冲突了!

我拿到 S 锁了，开始执行了喔，现在时间是 2019-02-17 18:38:27
我拿到 S 锁了，开始执行了喔，现在时间是 2019-02-17 18:38:27
我释放 S 锁了，结束执行了，现在时间是 2019-02-17 18:38:28
我释放 S 锁了，结束执行了，现在时间是 2019-02-17 18:38:28
S 锁和 S 锁 没有冲突!
```

```
我拿到 S 锁了，开始执行了喔，现在时间是 2019-02-17 18:38:28
我拿到 IS 锁了，开始执行了喔，现在时间是 2019-02-17 18:38:28
我释放 S 锁了，结束执行了，现在时间是 2019-02-17 18:38:29
我释放 IS 锁了，结束执行了，现在时间是 2019-02-17 18:38:29
S 锁和 IS 锁 没有冲突!

我拿到 IS 锁了，开始执行了喔，现在时间是 2019-02-17 18:38:29
我释放 IS 锁了，结束执行了，现在时间是 2019-02-17 18:38:30
我拿到 X 锁了，开始执行了喔，现在时间是 2019-02-17 18:38:30
我释放 X 锁了，结束执行了，现在时间是 2019-02-17 18:38:31
IS 锁和 X 锁 冲突了!

我拿到 IS 锁了，开始执行了喔，现在时间是 2019-02-17 18:38:31
我拿到 IX 锁了，开始执行了喔，现在时间是 2019-02-17 18:38:31
我释放 IX 锁了，结束执行了，现在时间是 2019-02-17 18:38:32
我释放 IS 锁了，结束执行了，现在时间是 2019-02-17 18:38:32
IS 锁和 IX 锁 没有冲突!

我拿到 IS 锁了，开始执行了喔，现在时间是 2019-02-17 18:38:32
我拿到 S 锁了，开始执行了喔，现在时间是 2019-02-17 18:38:32
我释放 IS 锁了，结束执行了，现在时间是 2019-02-17 18:38:33
我释放 S 锁了，结束执行了，现在时间是 2019-02-17 18:38:33
IS 锁和 S 锁 没有冲突!

我拿到 IS 锁了，开始执行了喔，现在时间是 2019-02-17 18:38:33
我拿到 IS 锁了，开始执行了喔，现在时间是 2019-02-17 18:38:33
我释放 IS 锁了，结束执行了，现在时间是 2019-02-17 18:38:34
我释放 IS 锁了，结束执行了，现在时间是 2019-02-17 18:38:34
IS 锁和 IS 锁 没有冲突!
```

4、公平锁与非公平锁

（1）问题分析

仔细想了想，如果有一种场景，就是用户一直再读，写获取不到锁，那么不就造成脏读吗？这不就是由于资源的抢占不就是非公平锁造成的。如何避免这个问题呢？这就涉及到了公平锁与非公平锁。

对产生的结果来说，如果一个线程组里，能保证每个线程都能拿到锁，那么这个锁就是公平锁。相反，如果保证不了每个线程都能拿到锁，也就是存在有线程饿死，那么这个锁就是非公平锁。

（2）非公平锁测试

上述代码锁实现的是非公平锁，测试代码如下：

```
def test_fair_lock():
threads = []
for i in range(0, 10):
if i == 2:
# 0 代表排他锁（X）
threads.append(MyThread(0))
else:
# 2 代表共享锁（S）
threads.append(MyThread(2))
 for thread in threads: thread.start()
```

运行结果：

```
我拿到 S 锁了，开始执行了喔，现在时间是 2019-02-17 19:06:33
我拿到 S 锁了，开始执行了喔，现在时间是 2019-02-17 19:06:33
我拿到 S 锁了，开始执行了喔，现在时间是 2019-02-17 19:06:33
我拿到 S 锁了，开始执行了喔，现在时间是 2019-02-17 19:06:33
我拿到 S 锁了，开始执行了喔，现在时间是 2019-02-17 19:06:33
我拿到 S 锁了，开始执行了喔，现在时间是 2019-02-17 19:06:33
我拿到 S 锁了，开始执行了喔，现在时间是 2019-02-17 19:06:33
我拿到 S 锁了，开始执行了喔，现在时间是 2019-02-17 19:06:33
我拿到 S 锁了，开始执行了喔，现在时间是 2019-02-17 19:06:33
 我释放 S 锁了，结束执行了，现在时间是 2019-02-17 19:06:34
 我释放 S 锁了，结束执行了，现在时间是 2019-02-17 19:06:34
 我释放 S 锁了，结束执行了，现在时间是 2019-02-17 19:06:34
 我释放 S 锁了，结束执行了，现在时间是 2019-02-17 19:06:34
 我释放 S 锁了，结束执行了，现在时间是 2019-02-17 19:06:34
```

```
我释放 S 锁了，结束执行了，现在时间是 2019-02-17 19:06:34
我释放 S 锁了，结束执行了，现在时间是 2019-02-17 19:06:34
我释放 S 锁了，结束执行了，现在时间是 2019-02-17 19:06:34
我释放 S 锁了，结束执行了，现在时间是 2019-02-17 19:06:34
我拿到 X 锁了，开始执行了喔，现在时间是 2019-02-17 19:06:34
我释放 X 锁了，结束执行了，现在时间是 2019-02-17 19:06:35
```

可以看到由于资源抢占问题，排他锁被最后才被获取到了。

（3）公平锁的实现

实现公平锁，只需要在原有的代码进行小小得修改就行了。

```
class Source:

# ...... 省略
def __init__(self, isFair=False):
self.__isFair = isFair
self.__initMutexFlag()
self.__initEvents()
# ...... 省略
def lock(self, flag):
# 如果是排他锁，先进进行枷锁
if flag == self.__X: self.__lockX.acquire()
if self.__isFair:
# 如果是公平锁则，先将互斥的锁给阻塞，防止其他线程进入
self.__lockEventsWait(flag)
self.__events[flag].wait()
self.__lockEventsQueue(flag)
else:
# 如果是非公平锁，如果锁拿不到，则先等待
self.__events[flag].wait()
self.__lockEvents(flag)
def __lockEventsWait(self, flag):
mutexFlags = self.__getMutexFlag(flag)
```

```
    for i in mutexFlags:
    # 为了保证原子操作，加锁
    self.__eventsLock[i].acquire()
    self.__events[i].clear()
    self.__eventsLock[i].release()
    def __lockEventsQueue(self, flag):
    mutexFlags = self.__getMutexFlag(flag)
    for i in mutexFlags:
    # 为了保证原子操作，加锁
    self.__eventsLock[i].acquire()
    self.__eventsQueue[i].append(self.__N)
    self.__eventsLock[i].release()
```

测试代码：

```
source = Source(True)def test_fair_lock():
threads = []
for i in range(0, 10):
if i == 2:
# 0 代表排他锁（X）
threads.append(MyThread(0))
else:
# 2 代表共享锁（S）
threads.append(MyThread(2))

    for thread in threads: thread.start()
```

运行结果：

```
我拿到 S 锁了，开始执行了喔，现在时间是 2019-02-17 19:35:16
我拿到 S 锁了，开始执行了喔，现在时间是 2019-02-17 19:35:16
我释放 S 锁了，结束执行了，现在时间是 2019-02-17 19:35:17
我释放 S 锁了，结束执行了，现在时间是 2019-02-17 19:35:17
我拿到 X 锁了，开始执行了喔，现在时间是 2019-02-17 19:35:17
我释放 X 锁了，结束执行了，现在时间是 2019-02-17 19:35:18
```

```
我拿到 S 锁了，开始执行了喔，现在时间是 2019-02-17 19:35:18
我拿到 S 锁了，开始执行了喔，现在时间是 2019-02-17 19:35:18
我拿到 S 锁了，开始执行了喔，现在时间是 2019-02-17 19:35:18
  我拿到 S 锁了，开始执行了喔，现在时间是 2019-02-17 19:35:18
  我拿到 S 锁了，开始执行了喔，现在时间是 2019-02-17 19:35:18
  我拿到 S 锁了，开始执行了喔，现在时间是 2019-02-17 19:35:18
  我拿到 S 锁了，开始执行了喔，现在时间是 2019-02-17 19:35:18
  我释放 S 锁了，结束执行了，现在时间是 2019-02-17 19:35:19
  我释放 S 锁了，结束执行了，现在时间是 2019-02-17 19:35:19
  我释放 S 锁了，结束执行了，现在时间是 2019-02-17 19:35:19
  我释放 S 锁了，结束执行了，现在时间是 2019-02-17 19:35:19
  我释放 S 锁了，结束执行了，现在时间是 2019-02-17 19:35:19
  我释放 S 锁了，结束执行了，现在时间是 2019-02-17 19:35:19
  我释放 S 锁了，结束执行了，现在时间是 2019-02-17 19:35:19
```

可以看到排他锁在第二次的时候就被获取到了。

（4）优缺点

非公平锁性能高于公平锁性能。首先，在恢复一个被挂起的线程与该线程真正运行之间存在着严重的延迟。而且，非公平锁能更充分的利用 cpu 的时间片，尽量的减少 cpu 空闲的状态时间。

9. 实现一个读写锁

> Python 实现读写锁

起步

Python 提供的多线程模型中并没有提供读写锁，读写锁相对于单纯的互斥锁，适用性更高，可以多个线程同时占用读模式的读写锁，但是只能一个线程占用写模式的读写锁。

通俗点说就是当没有写锁时，就可以加读锁且任意线程可以同时加；而写锁只能有一个线程，且必须在没有读锁时才能加上。

简单的实现

```
import threading

class RWlock(object):
    def __init__(self):
        self._lock = threading.Lock()
        self._extra = threading.Lock()
```

```
        self.read_num = 0
    def read_acquire(self):
        with self._extra:
            self.read_num += 1
            if self.read_num == 1:
                self._lock.acquire()
    def read_release(self):
        with self._extra:
            self.read_num -= 1
            if self.read_num == 0:
                self._lock.release()
    def write_acquire(self):
        self._lock.acquire()
    def write_release(self):
        self._lock.release()
```

这是读写锁的一个简单的实现，self.read_num 用来保存获得读锁的线程数，这个属性属于临界区，对其操作也要加锁，所以这里需要一个保护内部数据的额外的锁 self._extra 。

但是这个锁是不公平的。理想情况下，线程获得所的机会应该是一样的，不管线程是读操作还是写操作。而从上述代码可以看到，读请求都会立即设置 self.read_num += 1，不管有没有获得锁，而写请求想要获得锁还得等待 read_num 为 0 。

所以这个就造成了只有锁没有被占用或者没有读请求时，可以获得写权限。我们应该想办法避免读模式锁长期占用。

读写锁的优先级

读写锁也有分 读优先 和 写优先。上面的代码就属于读优先。

如果要改成写优先，那就换成去记录写线程的引用计数，读和写在同时竞争时，可以让写线程增加写的计数，这样可使读线程的读锁一直获取不到，因为读线程要先判断写的引用计数，若不为 0，则等待其为 0，然后进行读。这部分代码不罗列了。

但这样显然不够灵活。我们不需要两个相似的读写锁类。我们希望重构我们代码，使它更强大。

改进

为了能够满足自定义优先级的读写锁，要记录等待的读写线程数，并且需要两个条件 threading.Condition 用来处理哪方优先的通知。计数引用可以扩大语义：正数：表示正在读操作的线程数，负数：表示正在写操作的线程数（最多-1）

在获取读操作时，先然后判断时候有等待的写线程，没有，进行读操作，有，则等待读的计数加 1 后等待 Condition 通知；等待读的计数减 1，计数引用加 1，继续读操作，若条件不成立，循环等待；

在获取写操作时，若锁没有被占用，引用计数减 1，若被占用，等待写线程数加 1，等待写条件 Condition 的通知。

读模式和写模式的释放都是一样，需要根据判断去通知对应的 Condition：

```
class RWLock(object):
    def __init__(self):
        self.lock = threading.Lock()
```

```
        self.rcond = threading.Condition(self.lock)
        self.wcond = threading.Condition(self.lock)
        self.read_waiter = 0      # 等待获取读锁的线程数
        self.write_waiter = 0     # 等待获取写锁的线程数
        self.state = 0            # 正数：表示正在读操作的线程数    负数：表示正在写操作的线程数（最多-1）
        self.owners = []          # 正在操作的线程 id 集合
        self.write_first = True # 默认写优先，False 表示读优先

    def write_acquire(self, blocking=True):
        # 获取写锁只有当
        me = threading.get_ident()
        with self.lock:
            while not self._write_acquire(me):
                if not blocking:
                    return False
                self.write_waiter += 1
                self.wcond.wait()
                self.write_waiter -= 1
        return True

    def _write_acquire(self, me):
        # 获取写锁只有当锁没人占用，或者当前线程已经占用
        if self.state == 0 or (self.state < 0 and me in self.owners):
            self.state -= 1
            self.owners.append(me)
            return True
        if self.state > 0 and me in self.owners:
            raise RuntimeError('cannot recursively wrlock a rdlocked lock')
        return False

    def read_acquire(self, blocking=True):
        me = threading.get_ident()
        with self.lock:
            while not self._read_acquire(me):
                if not blocking:
```

```
                    return False
                self.read_waiter += 1
                self.rcond.wait()
                self.read_waiter -= 1
        return True

    def _read_acquire(self, me):
        if self.state < 0:
            # 如果锁被写锁占用
            return False

        if not self.write_waiter:
            ok = True
        else:
            ok = me in self.owners
        if ok or not self.write_first:
            self.state += 1
            self.owners.append(me)
            return True
        return False

    def unlock(self):
        me = threading.get_ident()
        with self.lock:
            try:
                self.owners.remove(me)
            except ValueError:
                raise RuntimeError('cannot release un-acquired lock')

            if self.state > 0:
                self.state -= 1
            else:
                self.state += 1
            if not self.state:
                if self.write_waiter and self.write_first:    # 如果有写操作在等待（默认写优先）
```

```
                    self.wcond.notify()
                elif self.read_waiter:
                    self.rcond.notify_all()
                elif self.write_waiter:
                    self.wcond.notify()

    read_release = unlock
    write_release = unlock
```

10. 设计一个无锁队列

python 多进程没有锁队列范例

假设有一些任务要完成。为了完成这项任务，将使用几个过程。所以，将保持两个队列。一个包含任务，另一个包含已完成任务的日志。
然后实例化流程来完成任务。请注意，python 队列类已经同步。
这意味着，我们不需要使用锁类来阻塞多个进程来访问同一个队列对象。这就是为什么，在这种情况下不需要使用锁类。
下面是将任务添加到队列中的实现，然后创建进程并启动它们，然后使用 join（）完成这些进程。最后，我们将从第二个队列打印日志。

```
from multiprocessing import Process, Queue, current_process
import time
import queue # imported for using queue.Empty exception

def do_job(tasks_to_accomplish, tasks_that_are_done):
    while True:
        try:
            '''
                try to get task from the queue. get_nowait() function will
                raise queue.Empty exception if the queue is empty.
                queue(False) function would do the same task also.
            '''
            task = tasks_to_accomplish.get_nowait()
        except queue.Empty:
            break
```

```
        else:
            '''
                if no exception has been raised, add the task completion
                message to task_that_are_done queue
            '''
            print(task)
            tasks_that_are_done.put(task + ' is done by ' + current_process().name)
            time.sleep(.5)
    return True

def main():
    number_of_task = 10
    number_of_processes = 4
    tasks_to_accomplish = Queue()
    tasks_that_are_done = Queue()
    processes = []
    for i in range(number_of_task):
        tasks_to_accomplish.put("Task no " + str(i))
    # creating processes
    for w in range(number_of_processes):
        p = Process(target=do_job, args=(tasks_to_accomplish, tasks_that_are_done))
        processes.append(p)
        p.start()
    # completing process
    for p in processes:
        p.join()
    # print the output
    while not tasks_that_are_done.empty():
        print(tasks_that_are_done.get())
    return True
if __name__ == '__main__':
```

```
main()
```

11. 协程的原理

协程是什么？用来解决什么问题?

说起进程和线程，大家都比较好理解，因为虽然进程和线程相对抽象，但在计算机运行中指代都很明确。进程是系统运行的最小资源分配单位，线程则是 CPU 的最小调度单元，但协程是什么？比较普遍的说法是，协程是内部运行时可以中断出来并且在适当时刻可以继续的函数。可能会觉得奇怪，因为 python 里的生成器也是这样的，把生成器和协程放到一起比较就像汽车遇到拖拉机。确实，生成器和协程用了同样的语言的特性，但目的完全不同。生成器支持迭代协议，可以减少 list 的内存消耗，也适用于复杂的迭代。而协程则不太一样。

协程基本就是未来异步 I/O 而生。一般来说 IO 任务都会有阻塞，比如网络传输，数据传过去再传回来当然需要时间，如果在这段时间里 CPU 空闲下来岂不是很浪费时间。一开始大家的思路是利用多进程/线程来解决这个问题，但是多进程/线程会占用相当可观的系统资源，同时对于多进程/线程的管理会对系统造成压力，因此这种方案不具备良好的可扩展性。因此，这一思路在服务器资源还没有富裕到足够程度的时候，是不可行的。即便资源足够富裕，效率也不够高。总之，此思路技术实现会使得资源占用过多，可扩展性差。于是协程的概念就横空出世了。当一个 io 子程序传输数据时自动切换到另一个子程序，待传输完成后再切换回来，就能最大限度的利用 CPU 资源达到高并发的效果。事实上，在 python3.4 以前，python3 是没有纯粹的协程概念的，之前协程是利用 yield 和 send 完成。3.4 出现了 asyncio 库，3.5 又有了 async/await 才使 python 的协程逐渐成熟。

提高并发量其实是一个很经典的问题，叫 C10K，由 Dan Kegel 提出。C10K 问题的由来：

大家都知道互联网的基础就是网络通信，早期的互联网可以说是一个小群体的集合。互联网还不够普及，用户也不多，一台服务器同时在线 100 个用户估计在当时已经算是大型应用了，所以并不存在什么 C10K 的难题。互联网的爆发期应该是在 www 网站，浏览器，雅虎出现后。最早的互联网称之为 Web1.0，互联网大部分的使用场景是下载一个 HTML 页面，用户在浏览器中查看网页上的信息，这个时期也不存在 C10K 问题。

Web2.0 时代到来后就不同了，一方面是普及率大大提高了，用户群体几何倍增长。另一方面是互联网不再是单纯的浏览万维网网页，逐渐开始进行交互，而且应用程序的逻辑也变的更复杂，从简单的表单提交，到即时通信和在线实时互动，C10K 的问题才体现出来了。因为每一个用户都必须与服务器保持 TCP 连接才能进行实时的数据交互，诸如 Facebook 这样的网站同一时间的并发 TCP 连接很可能已经过亿。

早期的腾讯 QQ 也同样面临 C10K 问题，只不过他们是用了 UDP 这种原始的包交换协议来实现的，绕开了这个难题，当然过程肯定是痛苦的。如果当时有 epoll 技术，他们肯定会用 TCP。众所周之，后来的手机 QQ、微信都采用 TCP 协议。

实际上当时也有异步模式，如：select/poll 模型，这些技术都有一定的缺点：如 selelct 最大不能超过 1024、poll 没有限制，但每次收到数据需要遍历每一个连接查看哪个连接有数据请求。

这时候问题就来了，最初的服务器都是基于进程/线程模型的，新到来一个 TCP 连接，就需要分配 1 个进程（或者线程）。而进程又是操作系统最昂贵的资源，一台机器无法创建很多进程。如果是 C10K 就要创建 1 万个进程，那么单机而言操作系统是无法承受的

（往往出现效率低下甚至完全瘫痪）。如果是采用分布式系统，维持 1 亿用户在线需要 10 万台服务器，成本巨大，也只有 Facebook、Google、雅虎等巨头才有财力购买如此多的服务器。
基于上述考虑，如何突破单机性能局限，是高性能网络编程所必须要直面的问题。这些局限和问题最早被 Dan Kegel 进行了归纳和总结，并首次成系统地分析和提出解决方案，后来这种普遍的网络现象和技术局限都被大家称为 C10K 问题。

协程的原理

协程拥有自己的寄存器上下文和栈。协程调度切换时，将寄存器上下文和栈保存到其他地方，在切回来的时候，恢复先前保存的寄存器上下文和栈。因此：协程能保留上一次调用时的状态（即所有局部状态的一个特定组合），每次过程重入时，就相当于进入上一次调用的状态，换种说法：进入上一次离开时所处逻辑流的位置。

协程的实现

1、yield/send
2、select
3、yield from
4、asyncio
5、async/await

Python 协程的引入与原理分析

相关概念

- **并发：**指一个时间段内，有几个程序在同一个 cpu 上运行，但是任意时刻只有一个程序在 cpu 上运行。比如说在一秒内 cpu 切换了 100 个进程，就可以认为 cpu 的并发是 100。
- **并行：**值任意时刻点上，有多个程序同时运行在 cpu 上，可以理解为多个 cpu，每个 cpu 独立运行自己程序，互不干扰。并行数量和 cpu 数量是一致的。

我们平时常说的高并发而不是高并行，是因为 cpu 的数量是有限的，不可以增加。

形象的理解：cpu 对应一个人，程序对应喝茶，人要喝茶需要四个步骤（可以对应程序需要开启四个线程）：1 烧水，2 备茶叶，3 洗茶杯，4 泡茶。

并发方式：烧水的同时做好 2 备茶叶，3 洗茶杯，等水烧好之后执行 4 泡茶。这样比顺序执行 1234 要省时间。

并行方式：叫来四个人（开启四个进程），分别执行任务 1234，整个程序执行时间取决于耗时最多的步骤。

- **同步** （注意同步和异步只是针对于 I/O 操作来讲的）值调用 IO 操作时，必须等待 IO 操作完成后才开始新的的调用方式。
- **异步** 指调用 IO 操作时，不必等待 IO 操作完成就开始新的的调用方式。
- **阻塞** 指调用函数的时候，当前线程被挂起。
- **非阻塞** 指调用函数的时候，当前线程不会被挂起，而是立即返回。

IO 多路复用

sllect， poll， epoll 都是 IO 多路复用的机制。IO 多路复用就是通过这样一种机制：一个进程可以监听多个描述符，一旦某个描述符就绪（一般是读就绪和写就绪），能够通知程序进行相应的操作。但 select，poll，epoll 本质上都是同步 IO，因为他们都需要在读写事件就绪后自己负责进行读写（即将数据从内核空间拷贝到应用缓存）。也就是说这个读写过程是阻塞的。而异步 IO 则无需自己负责读写，异步 IO 的实现会负责把数据从内核拷贝到用户空间。

select

select 函数监听的文件描述符分三类：writefds、readfds、和 exceptfds。调用后 select 函数会阻塞，直到描述符就绪（有数据可读、写、或者有 except）或者超时（timeout 指定等待时间，如果立即返回则设置为 null），函数返回。当 select 函数返回后，可以通过遍历 fdset，来找到就绪的描述符。

- 优点：良好的跨平台性（几乎所有的平台都支持）
- 缺点：单个进程能够监听的文件描述符数量存在最大限制，在 linux 上一般为 1024，可以通过修改宏定义甚至重新编译内核来提升，但是这样也会造成效率降低。

poll

不同于 select 使用三个位图来表示 fdset 的方式，poll 使用的是 pollfd 的指针实现

pollfd 结构包含了要监听的 event 和发生的 event，不再使用 select“参数-值”传递的方式。同时 pollfd 并没有最大数量限制（但是数量过大之后性能也是会下降）。和 select 函数一样，poll 返回后，需要轮询 pollfd 来获取就绪的描述符。

从上面看，select 和 poll 都需要在返回后，通过遍历文件描述符来获取已经就绪的 socket。事实上，同时连接的大量客户端在同一时刻可能只有很少的处于就绪状态，因此随着监视的描述符数量的增长，其效率也会下降。

epoll

epoll 是在 linux2.6 内核中国提出的，（windows 不支持），是之前的 select 和 poll 增强版。相对于 select 和 poll 来说，epoll 更加灵活，没有描述符的限制。epoll 使用一个文件描述符管理多个描述符，将用户关系的文件描述符的时间存放到内核的一个时间表中。这样在用户控件和内核控件的 coppy 只需要一次。

如何选择?

①在并发高同时连接活跃度不是很高的请看下，epoll 比 select 好（网站或 web 系统中，用户请求一个页面后随时可能会关闭）

②并发性不高，同时连接很活跃，select 比 epoll 好。（比如说游戏中数据一但连接了就会一直活跃，不会中断）

省略章节：由于在用到 select 的时候需要嵌套多层回调函数，然后印发一系列的问题，如可读性差，共享状态管理困难，出现异常排查复杂，于是引入协程，既操作简单，速度又快。

协程

对于上面的问题，我们希望去解决这样几个问题：

1. 采用同步的方式去编写异步的代码，使代码的可读性高，更简便。
2. 使用单线程去切换任务（就像单线程间函数之间的切换那样，速度超快）

   （1）线程是由操作系统切换的，单线程的切换意味着我们需要程序员自己去调度任务。

   （2）不需要锁，并发性高，如果单线程内切换函数，性能远高于线程切换，并发性更高。

例如我们在做爬虫的时候：

```
def get_url(url):
    html = get_html(url) # 此处网络下载 IO 操作比较耗时，希望切换到另一个函数去执行
    infos = parse_html(html)
# 下载 url 中的 html
def get_html(url):
```

```
    pass
# 解析网页
def parse_html(html):
    pass
```

意味着我们需要一个可以暂停的函数，对于此函数可以向暂停的地方穿入值。（回忆我们的生成器函数就可以满足这两个条件）所以就引入了协程。

生成器进阶

- 生成器不仅可以产出值，还可以接收值，用 send()方法。注意：在调用 send()发送非 None 值之前必须先启动生成器，可以用①next()②send(None)两种方式激活

```
def gen_func():
    html = yield 'http://www.baidu.com' # yield 前面加=号就实现了 1：可以产出值 2：可以接受调用者传过来的值
    print(html)
    yield 2
    yield 3
    return 'bobby'
if __name__ == '__main__':
    gen = gen_func()
    url = next(gen)
    print(url)
    html = 'bobby'
    gen.send(html) # send 方法既可以将值传递进生成器内部，又可以重新启动生成器执行到下一 yield 位置。
```

打印结果：

```
http://www.baidu.com
bobby
```

- close()方法。

```
def gen_func():
    yield 'http://www.baidu.com' # yield 前面加=号就实现了 1：可以产出值 2：可以接受调用者传过来的值
    yield 2
    yield 3
    return 'bobby'
if __name__ == '__main__':
    gen = gen_func()
```

```
    url = next(gen)
    gen.close()
    next(gen)
```

输出结果：

```
StopIteration
```

特别注意：调用 close.()之后， 生成器在往下运行的时候就会产生出一个 GeneratorExit，单数如果用 try 捕获异常的话，就算捕获了遇到后面还有 yield 的话，还是不能往下运行了，因为一旦调用 close 方法生成器就终止运行了（如果还有 next，就会会产生一个异常）所以我们不要去 try 捕捉该异常。（此注意可以先忽略）

```
def gen_func():
    try:
        yield 'http://www.baidu.com'
    except GeneratorExit:
        pass
    yield 2
    yield 3
    return 'bobby'
if __name__ == '__main__':
    gen = gen_func()
    print(next(gen))
    gen.close()
    next(gen)
```

输出结果：

```
RuntimeError: generator ignored GeneratorExit
```

- 调用 throw()方法。用于抛出一个异常。该异常可以捕捉忽略。

```
def gen_func():
    yield 'http://www.baidu.com' # yield 前面加=号就实现了 1：可以产出值 2：可以接受调用者传过来的值
    yield 2
    yield 3
    return 'bobby'
if __name__ == '__main__':
```

```
    gen = gen_func()
    print(next(gen))
    gen.throw(Exception, 'Download Error')
```

输出结果：

```
 Download Error
```

yield from

先看一个函数：from itertools import chain

```
from itertools import chain
my_list = [1,2,3]
my_dict = {'frank':'yangchao', 'ailsa':'liuliu'}
for value in chain(my_list, my_dict, range(5,10)):  chain()方法可以传入多个可迭代对象，然后分别遍历之。
    print(value)
```

打印结果：

```
1
2
3
frank
ailsa
5
6
7
8
9
```

此函数可以用 yield from 实现：yield from 功能 1：从一个可迭代对象中将值逐个返回。

```
my_list = [1,2,3]
my_dict = {'frank':'yangchao', 'ailsa':'liuliu'}
def chain(*args, **kwargs):
    for itemrable in args:
```

```
        yield from itemrable
for value in chain(my_list, my_dict, range(5,10)):
    print(value)
```

看如下代码：

```
def gen():
    yield 1

def g1(gen):
    yield from gen

def main():
    g = g1(gen)
    g.send(None)
```

代码分析：此代码中 main 调用了 g1， main 就叫作调用方， g1 叫做委托方， gen 叫做子生成器 yield from 将会在调用方 main 与子生成器 gen 之间建立一个双向通道。（意味着可以直接越过委托方）

例子：当委托方 middle()中使用 yield from 的时候，调用方 main 直接和子生成器 sales_sum 形成数据通道。

```
final_result = {}
def sales_sum(pro_name):
    total = 0
    nums = []
    while True:
        x = yield
        print(pro_name+'销量', x)
        if not x:
            break
        total += x
        nums.append(x)
    return total, nums #程序运行到 return 的时候，会将 return 的返回值返回给委托方，即 middle 中的 final_result[key]
def middle(key):
    while True: #相当于不停监听 sales_sum 是否有返回数据，（本例中有三次返回）
        final_result[key] = yield from sales_sum(key)
```

```
        print(key +'销量统计完成！！')
def main():
    data_sets = {
        '面膜':[1200, 1500, 3000],
        '手机':[88, 100, 98, 108],
        '衣服':[280, 560,778,70],
    }

    for key, data_set in data_sets.items():
        print('start key', key)
        m = middle(key)
        m.send(None) # 预激生成器
        for value in data_set:
            m.send(value)
        m.send(None)# 发送一个 None 使 sales_sum 中的 x 值为 None 退出 while 循环
    print(final_result)
if __name__ == '__main__':
    main()
```

结果：

```
start key 面膜
面膜销量 1200
面膜销量 1500
面膜销量 3000
面膜销量 None
面膜销量统计完成！！
start key 手机
手机销量 88
手机销量 100
手机销量 98
手机销量 108
```

```
手机销量 None
手机销量统计完成！！
start key 衣服
衣服销量 280
衣服销量 560
衣服销量 778
衣服销量 70
衣服销量 None
衣服销量统计完成！！
{'面膜': (5700, [1200, 1500, 3000]), '手机': (394, [88, 100, 98, 108]), '衣服': (1688, [280, 560, 778, 70])}
```

也许有人会好奇，为什么不能直接用 main()函数直接去调用 sales_sum 呢？加一个委托方使代码复杂化了。看以下直接用 main()函数直接去调用 sales_sum 代码：

```
def sales_sum(pro_name):
    total = 0
    nums = []
    while True:
        x = yield
        print(pro_name+'销量', x)
        if not x:
            break
        total += 1
        nums.append(x)
    return total, nums

if __name__ == '__main__':
    my_gen = sales_sum('面膜')
    my_gen.send(None)
    my_gen.send(1200)
    my_gen.send(1500)
    my_gen.send(3000)
    my_gen.send(None)
```

```
输出结果：
面膜销量 1200
面膜销量 1500
面膜销量 3000
面膜销量 None
Traceback (most recent call last):
  File "D:/MyCode/Cuiqingcai/Flask/test01.py", line 56, in <module>
    my_gen.send(None)
StopIteration: (3, [1200, 1500, 3000])
```

从上述代码可以看出，即使数据 return 结果出来了，还是会返回一个 exception，由此可以看出 yield from 的一个最大优点就是当子生成器运行时候出现异常，yield from 可以直接自动处理这些异常。

yield from 功能总结:

1. 子生成器生产的值，都是直接给调用方；调用发通过.send()发送的值都是直接传递给子生成器，如果传递 None，会调用子生成器的 next()方法，如果不是 None，会调用子生成器的 sen()方法。
2. 子生成器退出的时候，最后的 return EXPR，会触发一个 StopIteration（EXPR）异常
3. yield from 表达式的值，是子生成器终止时，传递给 StopIteration 异常的第一个参数。
4. 如果调用的时候出现了 StopIteration 异常，委托方生成器恢复运行，同时其他的异常向上冒泡。
5. 传入委托生成器的异常里，除了 GeneratorExit 之后，其他所有异常全部传递给子生成器的.throw()方法；如果调用.throw()的时候出现 StopIteration 异常，那么就恢复委托生成器的运行，其他的异常全部向上冒泡
6. 如果在委托生成器上调用.close()或传入 GeneratorExit 异常，会调用子生成器的.close()方法，没有就不调用，如果在调用.close()时候抛出了异常，那么就向上冒泡，否则的话委托生成器跑出 GeneratorExit 异常。

Python 协程实现原理

1. 利用 Python 中 yield 关键字修饰函数使其成为生成器
2. 利用生成器特点：可迭代且是从 yield 修饰处开始
3. 协程近似函数调用，因此所用的资源少于进程和线程
4. 可实现并发操作
5. 以下为原理测试代码

```
def test_1():
        while True:
                print("----1---")
                time.sleep(1)
```

```
            yield

def test_2():
    while True:
            print("----2---")
            time.sleep(1)
            yield

t1 = test_1()
t2 = test_2()
for i in range(4):

    next(t1)
    next(t2
```

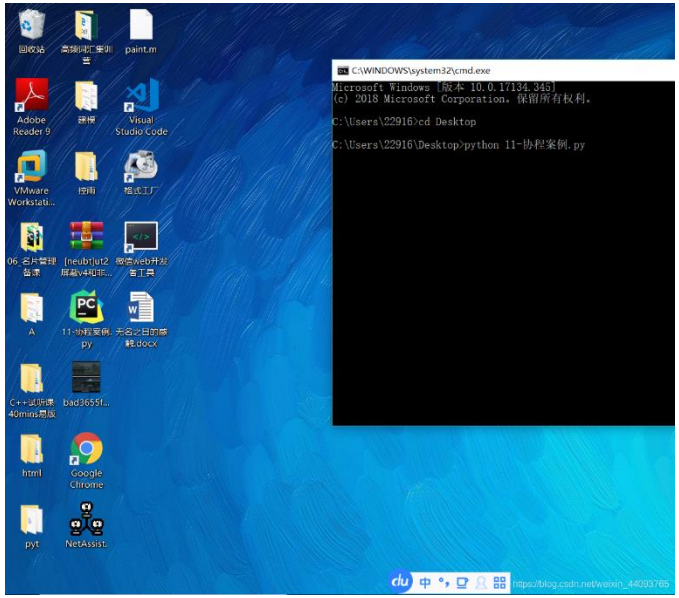
简单利用协程案例

1. 使用利用上述原理封装的库：gevent
2. 接下来 多打代码少动脑
3. monkey.patch_all() 用于协程打补丁
4. gevent.joinall([]) 用于开始多个协程
5. gevent.spawn(function，…)传入协程执行的函数，及其他参数
6. 由于爬取图片耗时最长就在等待响应和下载过程，多张图片的爬取这个过程，通过并发即可提高效率

```
import urllib.request
import gevent
from gevent import monkey

monkey.patch_all()

def download(img_name,url):
    req = urllib.request.urlopen(url)
    img_content = req.read()
    with open(img_name,"wb") as f:
```

```
            f.write(img_content)

def main():
      gevent.joinall([
            gevent.spawn(download,"1.jpg","https://rpic.douyucdn.cn/live-
cover/appCovers/2018/10/09/4403598_20181009234622_small.jpg"),
            gevent.spawn(download,"2.jpg","https://rpic.douyucdn.cn/live-
cover/appCovers/2018/10/11/5763744_20181011182233_small.jpg"),
            gevent.spawn(download,"3.jpg","https://rpic.douyucdn.cn/live-
cover/appCovers/2018/12/13/5960450_20181213135411_small.jpg")
            ])

if __name__ == "__main__":
      main()
```

案例结果示例



四、数据库

1. 索引是什么

索引 （Index） 是帮助 MySQL 高效获取数据的数据结构。

1. 索引能极大的减少存储引擎需要扫描的数据量
2. 索引可以把随机 IO 变成顺序 IO
3. 索引可以帮助我们在进行 分组、 排序等操作时，避免使用临时表

2. 为什么要用 B+ 树（B+ 树的优缺点）

B+ 树是 B- 树的变种 （PLUS 版）多路绝对平衡查找树，好的时间复杂度

B+ 树扫库、表能力更强

B+ 树的磁盘读写能力更强

B+ 树的排序能力更强

B+ 树的查询效率更加稳定（仁者见仁、智者见智，有可能 B-Tree 第一次就命中了，直接返回，而 B+Tree 需要找到叶节点，所以查找效率不一定比 B-Tree 更高）

支持顺序排序，叶节点之间存在链接

3. B+索引和哈希索引的区别？

B-tree 索引

索引是按照顺序存储的，所以，如果按照 B-tree 索引，可以直接返回，带顺序的数据，但这个数据只是该索引列含有的信息。因此是顺序 I/O

适用于：

精确匹配

范围匹配

最左匹配

Hash 索引

索引列值的哈希值+数据行指针：因此找到后还需要根据指针去找数据，造成随机 I/O

适合：

精确匹配

不适合：

模糊匹配

范围匹配

不能排序

1、hash 索引仅满足 “=”、“IN” 和 “<=>” 查询，不能使用范围查询因为 hash 索引比较的是经常 hash 运算之后的 hash 值，因此只能进行等值的过滤，不能基于范围的查找，因为经过 hash 算法处理后的 hash 值的大小关系，并不能保证与处理前的 hash 大小关系对应。
2、hash 索引无法被用来进行数据的排序操作由于 hash 索引中存放的都是经过 hash 计算之后的值，而 hash 值的大小关系不一定与 hash 计算之前的值一样，所以数据库无法利用 hash 索引中的值进行排序操作。
3、对于组合索引，Hash 索引在计算 Hash 值的时候是组合索引键合并后再一起计算 Hash 值，而不是单独计算 Hash 值，所以通过组合索引的前面一个或几个索引键进行查询的时候，Hash 索引也无法被利用。
4、Hash 索引遇到大量 Hash 值相等的情况后性能并不一定就会比 B-Tree 索引高。对于选择性比较低的索引键，如果创建 Hash 索引，那么将会存在大量记录指针信息存于同一个 Hash 值相关联。这样要定位某一条记录时就会非常麻烦，会浪费多次表数据的访问，而造成整体性能低下。
总结：哈希适用在小范围的精确查找，在列数据很大，又不需要排序，不需要模糊查询，范围查询时有用

4. B+ 树中叶子节点间的指针有什么用？

使得访问更加简单，b 树的话需要不断在父节点和叶子节点之间来回移动

5. 聚簇和非聚簇索引的区别？

聚簇索引：将数据存储与索引放到了一块，找到索引也就找到了数据，聚簇索引的叶子节点就是数据节点，由于聚簇索引是将数据跟索引结构放到一块，因此一个表仅有一个聚簇索引，聚簇索引具有唯一性
非聚簇索引：非聚簇索引的叶子节点仍然是索引节点，只不过有指向对应数据块的指针。将数据存储于索引分开结构，索引结构的叶子节点指向了数据的对应行

6. 非聚簇索引的查询都要回表吗？

是的

7. B+ 树和 AVL 树 B 树二叉搜索树有什么区别？

这几种各有交集但又不尽相同。
什么是二叉搜索树？
它或者是一棵空树，或者是具有下列性质的二叉树：
1.左子节点的值必定小于父节点的值
2.右子节点的值必定大于父节点的值
首先 AVL 树是自平衡的二叉搜索树 AVL 在该定义的基础上加入了平衡条件，即：某节点的左右子树的高度差小于等于 1
另一种非严格自平衡的二叉搜索树是红黑树，二者使用场景略微有些不同。
AVL 追求整体的绝对平衡，适合于少量插入，大量查找的应用场景（因为维护全局平衡，插入一个往往需要 O(log n)）
红黑树适用于一部分插入，一部分查询的场景（变色，左旋右旋场景相对少些）
B+ 树是对 B 树的拓展
一棵 m 阶 B 树 (balanced tree of order m) 是一棵平衡的 m 路搜索树。它或者是空树，或者是满足下列性质的树：
1、根结点至少有两个子女；

2、每个非根节点所包含的关键字个数 j 满足：┌m/2┐ - 1 <= j <= m - 1；
3、除根结点以外的所有结点（不包括叶子结点）的度数正好是关键字总数加 1，故内部子树个数 k 满足：┌m/2┐ <= k <= m ；
4、所有的叶子结点都位于同一层。
B+ 树对 B 树的改进就是：只有叶节点存数据，并且维护了两个指针，一个指向根节点另一个指向顺序叶节点的首位。
B+ 树还在叶子节点中加入了链接指针

五、杂项

这一部分和项目比较相关。基本上项目中有什么或者面试官想到什么问什么，问了很多但是不通用。就只写一点。

1. GIL 是什么

全局状态锁

2. 为什么会有?

出现多线程编程之间数据和状态一致性问题，为了解决这个数据不能同步的问题

3. 有什么作用?

全局解释锁，每次只能一个线程获得 cpu 的使用权：为了线程安全，也就是为了解决多线程之间的数据完整性和状态同步而加的锁，因为我们知道线程之间的数据是共享的。

Python 语言和 GIL 解释器锁没有关系，它是在实现 Python 解析器(CPython)时所引入的一个概念，同样一段代码可以通过 CPython，PyPy，Psyco 等不同的 Python 执行环境来执行，然而因为 CPython 是大部分环境下默认的 Python 执行环境。所以在很多人的概念里 CPython 就是 Python，也就想当然的把 GIL 归结为 Python 语言的缺陷，所有 GIL 并不是 python 的特性，仅仅是因为历史原因在 Cpython 解释器中难以移除。

GIL 保证同一时刻只有一个线程执行代码，每个线程在执行过程中都要先获取 GIL

线程释放 GIL 锁的情况： 在 IO 操作等可能会引起阻塞的 system call 之前,可以暂时释放 GIL,但在执行完毕后,必须重新获取 GIL Python 3.x 使用计时器（执行时间达到阈值后，当前线程释放 GIL）或 Python 2.x，tickets 计数达到 100

Python 使用多进程是可以利用多核的 CPU 资源的。

多线程爬取比单线程性能有提升，因为遇到 IO 阻塞会自动释放 GIL 锁

GIL 只对计算密集型的程序有作用，对 IO 密集型的程序并没有影响，因为遇到 IO 阻塞会自动释放 GIL 锁

当需要执行计算密集型的程序时，可以选择：1.换解释器，2.扩展 C 语言，3.换多进程等方案

4. 怎么规避它对于并行的影响?

GIL 是什么：一个防止多线程并发执行机器码的一个 Mutex

一个进程里只有一个 GIL 锁，线程只能拿到 GIL 锁才能运行，所以 python 多线程的假的

首先需要明确的一点是 GIL 并不是 Python 的特性，它是在实现 Python 解析器(CPython)时所引入的一个概念。就好比 C++是一套语言（语法）标准，但是可以用不同的编译器来编译成可执行代码。Python 的代码可以通过 CPython，PyPy，Psyco 等不同的 Python 执行环境来执行。像其中的 JPython 就没有 GIL。然而因为 CPython 是大部分环境下默认的 Python 执行环境。

GIL 锁是历史遗留下来的问题，上世纪九十年代 python 创始人“龟叔”，为了解决线程安全问题写的一个锁。

官方给出的解释大概就是： 在 CPython，全局解释器锁，是一个互斥锁，它阻止多线程同时执行 Python 字节码。这个锁是必要的，主要是因为 CPython 的内存管理不是线程安全的。（然而，由于 GIL 的存在，其他特征已经发展成依赖于它所实施的保证。） ==========》 意思就是 ：在 Cpython 解释器中这个 GIL 锁基本不能移除

问题：描述 Python GIL 的概念， 以及它对 python 多线程的影响？阐明多线程抓取程序是否可比单线程性能有提升，并解释原因。
参考答案:

1. **Python 语言和 GIL 没有半毛钱关系。仅仅是由于历史原因在 Cpython 虚拟机(解释器)，难以移除 GIL。**
2. **GIL：全局解释器锁。每个线程在执行的过程都需要先获取 GIL，保证同一时刻只有一个线程可以执行代码。**
3. **线程释放 GIL 锁的情况： 在 IO 操作等可能会引起阻塞的 system call 之前,可以暂时释放 GIL,但在执行完毕后,必须重新获取 GIL Python 3.x 使用计时器（执行时间达到阈值后，当前线程释放 GIL）或 Python 2.x，tickets 计数达到 100**
4. **Python 使用多进程是可以利用多核的 CPU 资源的。**
5. **多线程爬取比单线程性能有提升，因为遇到 IO 阻塞会自动释放 GIL 锁**

解决 GIL 锁方法：
1:更换解释器 比如使用 jpython(java 实现的 python 解释器)
2:使用多进程完成多任务的处理
结论:

1. 在 处理像科学计算 这类需要持续使用 cpu 的任务的时候 单线程会比多线程快
2. 在 处理像 IO 操作等可能引起阻塞的这类任务的时候 多线程会比单线程快

六、语言相关
1. Python 的内存管理机制
（1）垃圾回收
（2）引用计数
（3）内存池机制
2. 讲一下 Python GC 的原理和详细解释（分代，标记回收，内存划分）
Python 语言默认采用的垃圾收集机制是『引用计数法 Reference Counting』，『引用计数法』的原理是：每个对象维护一个 ob_ref 字段，用来记录该对象当前被引用的次数，每当新的引用指向该对象时，它的引用计数 ob_ref 加 1，每当该对象的引用失效时计数 ob_ref 减 1，一旦对象的引用计数为 0，该对象立即被回收，对象占用的内存空间将被释放。
为了解决对象的循环引用问题，Python 引入了标记-清除和分代回收两种 GC 机制。
标记就是使用有向图的方式，不可达的清除掉（主要用来清理容器对象）
分代分为三代：年轻，中年，老年。年轻对象满，触发 GC，可回收对象回收掉，不可回收移到中年中去，以此类推。结合标记使用
引用计数增加

1.对象被创建： x=4
2.另外的别人被创建： y=x
3.被作为参数传递给函数： foo(x)
4.作为容器对象的一个元素： a=[1,x,'33']
引用计数减少
1.一个本地引用离开了它的作用域。比如上面的 foo(x) 函数结束时，x 指向的对象引用减 1。
2.对象的别名被显式的销毁： del x ；或者 del y
3.对象的一个别名被赋值给其他对象： x=789
4.对象从一个窗口对象中移除： myList.remove(x)
5.窗口对象本身被销毁： del myList，或者窗口对象本身离开了作用域。
垃圾回收
1、当内存中有不再使用的部分时，垃圾收集器就会把他们清理掉。它会去检查那些引用计数为 0 的对象，然后清除其在内存的空间。当然除了引用计数为 0 的会被清除，还有一种情况也会被垃圾收集器清掉：当两个对象相互引用时，他们本身其他的引用已经为 0 了。
2、垃圾回收机制还有一个循环垃圾回收器，确保释放循环引用对象 (a 引用 b, b 引用 a, 导致其引用计数永远不为 0)。

3. Python 中 static_method 、 class_mathod 、和普通 method 有什么区别

python 中的@classmethod 和@staticmethod 和普通方法的区别

普通方法定义：第一个参数必须是实例对象，该参数名一般约定为“self”，通过它来传递实例的属性和方法（也可以传类的属性和方法）；
调用：只能由实例对象调用。
类方法
定义：使用装饰器@classmethod。第一个参数必须是当前类对象，该参数名一般约定为“cls”，通过它来传递类的属性和方法（不能传实例的属性和方法）；
调用：实例对象和类对象都可以调用。
静态方法
定义：使用装饰器@staticmethod。参数随意，没有“self”和“cls”参数，但是方法体中不能使用类或实例的任何属性和方法；
调用：实例对象和类对象都可以调用。

```python
    def ord_func(self):
        """ 定义普通方法，至少有一个 self 参数 """

        # print self.name
        print '普通方法'

    @classmethod
    def class_func(cls):
        """ 定义类方法，至少有一个 cls 参数 """

        print '类方法'

    @staticmethod
    def static_func():
        """ 定义静态方法 ，无默认参数"""

        print '静态方法'
```

```python
# 调用普通方法
f = Foo()
f.ord_func()

# 调用类方法
Foo.class_func()

# 调用静态方法
Foo.static_func()
```

python 中 classmethod、staticmethod 和普通方法的区别

这三种方法在类和实例中都存在，都可被调用，但调用的时候默认默认传递的参数不同。

对于 classmethod 方法

当实例调用 classmethod 方法时，默认会把当前实例所对应的类传进去

当类调用 classmethod 方法时，默认把此类传进去

对于 staticmethod 方法

实例和类调用，没有默认的参数传进函数

对于普通方法

当实例调用时，默认将当前实例传进去

类调用时，只能以 类名.method(类实例) 形式调用

下面通过一个例子清楚说明具体调用情况：

```
class Locker(object):
      a = 'aa'

   @staticmethod
   def show():
      print 'show'

   @classmethod
   def display(cls):
      print 'display'

   def out(self):
      print 'out'

cc = Locker()

#
Locker.show()
cc.show()

#
Locker.display()
cc.display()

#
Locker.out(cc)
cc.out()
```

运行结果：

```
show
```

```
show
display
display
out
out
```

我们修改 **classmethod** 方法：

```
class Locker(object):
      a = 'aa'

   @staticmethod
   def show():
      print 'show'

   @classmethod
   def display(cls):
      print cls.a

   def out(self):
      print 'out'

cc = Locker()

cc.a = 'vv'

#
Locker.display()
cc.display()
```

结果：

```
aa
aa
```

可见当实例调用 classmethod 方法时，默认传入的参数是实例对应的类

总结:

classmethod 和普通函数调用时都有默认参数传入，只有 staticmethod 调用时没有任何默认参数传入

4. 迭代器和生成器有什么区别？

先说结论，生成器式 一种特殊的迭代器：其在使用时生成。

首先明确两个概念：

Iterable ：所有实现了 iter__ 的对象均可称作 Iterablec。Iterator：是指同时实现了 *_iter_* 和 _next 的对象，迭代器也属于 Iterable。

小结

凡是可作用于 for 循环的对象都是 Iterable 类型；

凡是可作用于 next() 函数的对象都是 Iterator 类型，它们表示一个惰性计算的序列

5. 生成器怎么使用？

两种方式：

```
1. a =(i for i in range(100))
2. def a(n):
3.     i = 0
4.     while(i < n):
5.         yield i
6.         i += 1
```

推荐阅读：

字节跳动面试之---Python

文章来源:开源项目 Github https://github.com/wolverinn/Waking-Up

Python

- 什么是 Python 生成器？
- 什么是 Python 迭代器？

- list 和 tuple 有什么区别？
- Python 中的 list 和 dict 是怎么实现的？
- Python 中使用多线程可以达到多核 CPU 一起使用吗？
- 什么是装饰器？
- Python 如何进行内存管理？
- Python 中的垃圾回收机制？
- 什么是 lambda 表达式？
- 什么是深拷贝和浅拷贝？
- 双等于和 is 有什么区别？
- 其它 Python 知识点

什么是 Python 生成器?

generator，有两种产生生成器对象的方式：一种是列表生成式加括号：

`g1 = (x for x in range(10))`

一种是在函数定义中包含 `yield` 关键字：

```
def fib(max):
    n, a, b = 0, 0, 1
    while n < max:
        yield b
        a, b = b, a + b
        n = n + 1
    return 'done'

g2 = fib(8)
```

对于 generator 对象 g1 和 g2，可以通过 `next(g1)`不断获得下一个元素的值，如果没有更多的元素，就会报错 `StopIteration`
也可以通过 for 循环获得元素的值。
生成器的好处是不用占用很多内存，只需要在用的时候计算元素的值就行了。

什么是 Python 迭代器?

Python 中可以用于 for 循环的，叫做可迭代 `Iterable`，包括 list/set/tuple/str/dict 等数据结构以及生成器；可以用以下语句判断一个对象是否是可迭代的：

```
from collections import Iterable
isinstance(x, Iterable)
```

迭代器 Iterator，是指可以被 next()函数调用并不断返回下一个值，直到 StopIteration；生成器都是 Iterator，而列表等数据结构不是；可以通过以下语句将 list 变为 Iterator：

```
iter([1,2,3,4,5])
```

生成器都是 Iterator，但迭代器不一定是生成器。

list 和 tuple 有什么区别?

- list 长度可变，tuple 不可变；
- list 中元素的值可以改变，tuple 不能改变；
- list 支持 append; insert; remove; pop 等方法，tuple 都不支持

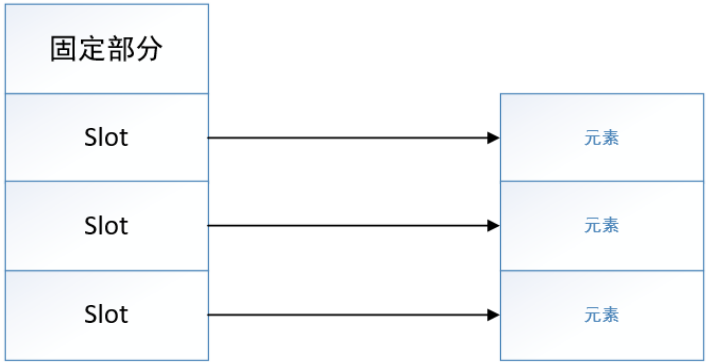
Python 中的 list 和 dict 是怎么实现的?

dict 查找速度快,占用的内存较大,list 查找速度慢,占用内存较小,dict 不能用来存储有序集合。Dict 用{}表示,list 用[]表示。列表是 Python 中最基本的数据结构，列表是最常用的 Python 数据类型，列表的数据项不需要具有相同的类型。列表中的每个元素都分配一个数字 - 它的位置，或索引，第一个索引是 0，第二个索引是 1，依此类推。

list 数据结构

list 的实现是一个可扩展的顺序表，表里除了表头+分配的 slot 的个数（固定大小）之外，还存在一个数组，数组元素是指针。

如下图所示：



Lis结构

因此，我们先创建一个空的 list，可获取固定部分的大小。

```
```

```
import sys

l = []

print(sys.getsizeof(l))
64

l.append(10)

print(sys.getsizeof(l))
96
```

可见，list 的固定部分大小为 64
此时：slot 的个数为 0，当 append 一个元素时，按照 resize 规则（0，4，8，16，25，35，46，58，72，88……），此时有四个 slot，而一个指针占有 8 字节。因而此时大小变为 64+4*8 = 96

dict 是通过 hash 表实现的,dict 为一个数组,数组的索引键是通过 hash 函数处理后得到的,hash 函数的目的是使键值均匀的分布在数组中。字典（Dictionary）是一种映射结构的数据类型，由无序的“键 - 值对”组成。字典的键必须是不可改变的类型，如：字符串，数字，tuple；值可以为任何 python 数据类型。
dict 的实现用的是 hash 结果，而且解决 hash 冲突使用的是开放寻址法。
hash 的一个节点的结果如下

```
typedef struct {
    Py_ssize_t me_hash;
    PyObject *me_key;
    PyObject *me_value;
} PyDictEntry;
```

第一个字段为 key 经过 hash 函数计算之后的 hash 码
第二个字段为指向 key 的指针
第三个字段为指向 value 的指针
hash 的结果如下

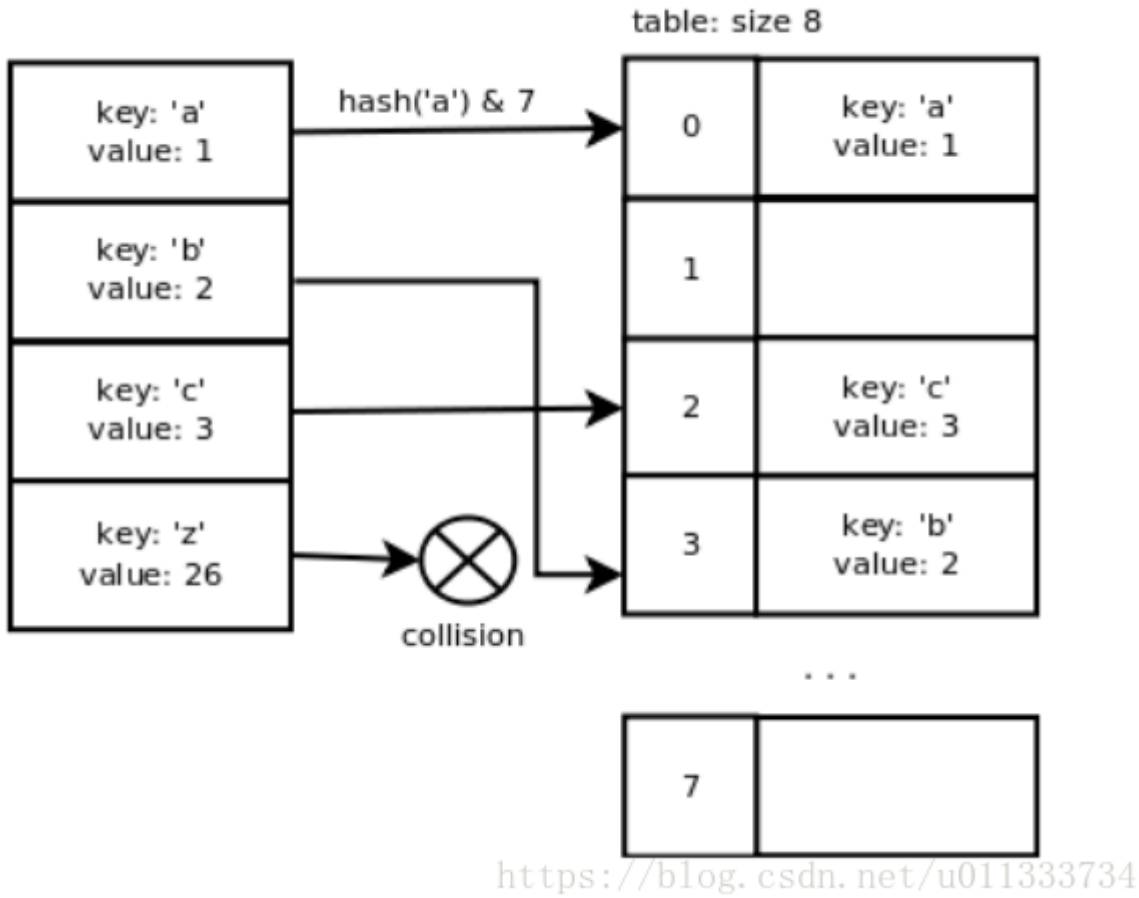
```
struct _dictobject {
   PyObject _HEAD
```

```
    Py_ssize_t ma_fill; /* # Active + # Dummy */
    Py_ssize_t ma_used; /* # Active */
    Py_ssize_t ma_mask;

    PyDictEntry *ma_table;
    PyDictEntry *(*ma_lookup)(PyDictObject *mp, PyObject *key, long hash);
    PyDictEntry ma_smalltable[PyDict_MINSIZE];
};

作者：lintong
链接：https://www.jianshu.com/p/02af9673ab34
來源：简书
简书著作权归作者所有，任何形式的转载都请联系作者获得授权并注明出处。
```

可见，hash 的 size 都是固定的

```
stu = {}

print(sys.getsizeof(stu))
240

stu = {"a":10,"b":20}

print(sys.getsizeof(stu))
240
```

如图：



key:value 经过 hash 函数计算，得到应该放在 hash 的对应 slot 中，如果发生了两个不同的 key 对应相同的 slot，则使用开放寻址法进行解决冲突：寻找其他未被使用的 slot。

list 和 dict 是 python 中常用的列表和字典。

这里讨论一下他们的原理及一些高级用法，供大家查询参考。

list 的切片

list 的切片格式为：

```
list[start:end:step]
```

其中 step 如果没有，则默认为 1

下面举例说明：

先定义一个 list：

```
list = [1,2,3,4,5,6,7,8,9]
```

那么 list[1:6:2]，就表示从 1 位置开始到 3 位置结束，间隔 2，结果如下：

```
list[1:6:2]
[2, 4, 6]
```

注意：list 的切片操作会返回一个新的列表，切片并不会改变原 list 本身。

dict 的浅拷贝及深拷贝

什么是 dict（字典）的浅拷贝和深拷贝？

如果我们在已有指针的情况下，增加一个指针指向已经存在的内存，称为浅拷贝。

如果我们增加一个指针的同时也新增了内存，并且新指针指向到新内存，称为深拷贝。

注意区分 dict 的浅拷贝及深拷贝的区别。

下面举个例子说明一下：

```
import copy

dict = {"a":{"a1":"a123"}, "b":{"b1":"b123"}}
a_dict = dict.copy()              # 浅拷贝
a_dict["a"]["a1"]="newa123"

b_dict = copy.deepcopy(dict)       #深拷贝
b_dict["b"]["b1"]="newb123"

print(dict)
print(a_dict)
print(b_dict)
```

浅拷贝会替换掉原来的的 dict，而深拷贝会新建一个 dict，不会替换掉原来的 dict。

list 转化为 dict

在 python3 的环境下，list 转化成 dict。

下面每一步都复制了执行语句后的结果。便于参考。

先定义一个 list：

```
>>> list = ["a1", "b1"]
```

通过 dict.fromkeys 方法创建一个新字典：

```
>>> dict_list = dict.fromkeys(list,{"aa1":"1234556"})
>>> dict_list
{'a1': {'aa1': '1234556'}, 'b1': {'aa1': '1234556'}}
```

通过 get 方法获取 value 的值，注意 get 的用法：

```
>>> value = dict_list.get("a1")
>>> value
{'aa1': '1234556'}
```

用 update 方法来修改 dict：

```
>>> dict_list.update(a1="new123",b1="bbb123")
>>> dict_list
{'a1': 'new123', 'b1': 'bbb123'}

>>> dict_list.update((("a1","aaa123"),("b1","bbb123")))
>>> dict_list
{'a1': 'aaa123', 'b1': 'bbb123'}
```

完整代码如下：

```
list = ["a1","b1"]
dict_list = dict.fromkeys(list,{"aa1":"1234556"})
value = dict_list.get("a1")
dict_list.update(a1="new123",b1="bbb123")
dict_list.update((("a1","aaa123"),("b1","bbb123")))
```

list 和 dict 的实现原理

当我们定义一个 list 或者 dict 的时候，数据会存入内存中。
在 list 中随着 list 数据的增大，查找时间会增大，
而在 dict 中查找元素的时候，不会随着 dict 的增大而增大，
所以说 dict 查找的性能远大于 list。
那么，为什么 dict 中查找元素的时候，时间会比查找 list 少呢？
dict 背后实现原理是哈希表（hash），其特点是：
1、dict 的 key 或者 set 的值，都必须是可以 hash 的，

就是说对不可变对象都是可 hash 的，比如 str, fronzenset（不可变集合）, tuple,
如果是自己实现的类，我们可以在类中加__hash__，这样我们的对象就是可 hash 对象
2、dict 的内存花销大，但是查询速度快，自定义对象或 python 内部的对象都是用 dict 包装的。
3、dict 的存储顺序和元素添加顺序有关。
4、添加数据有可能改变已有数据的顺序。
当表元（列表中的元素）小于三分之一，就会重新申请空间，在重新分配内存中，有可能重新改变顺序。
字典与列表 dict 和 list
字典通过偏移量直接寻址。
查找原理大致步骤如下，原谅我不会画图：
1、计算键的散列值（哈希值）–>
2、使用散列值的一部分来定位散列表中的一个表元 -->
3、表元为空 --> 抛出 keyerror
4、表元不为空 -->
5、判断键是否相等 -->
如果是 --> 返回表元里的值
如果否，表示散列冲突 -->
6、使用散列值的另一部分来定位散列表中的另一行 -->
7、表元为空 -->
8、返回前面第 3 步骤继续循环

set 集合和 frozenset 集合

1、set 集合是可变集合，可以添加，可以删除。
set 集合没有 hash 值，它是无序的。不支持序列，不记录元素的位置。
2、frozenset 集合是不可变集合，不能添加，一旦创建就不能删除。
frozenset 可以作为字典的 key，也可以作为其他集合的元素。
他们特点都是无序，不能重复的，如：
set 集合是无序且不能重复的：

```
>>> s=set('abcdcee')
>>> s
{'d', 'e', 'b', 'a', 'c'}
```

set 集合可以添加：

```
>>> s.add('g')
```

```
>>> s
{'d', 'e', 'b', 'a', 'c', 'g'}
```

frozenset 集合也是是无序且不能重复的：

```
>>> ss=frozenset('123332')
>>> ss
frozenset({'3', '1', '2'})
```

注意 frozenset 集合是没有 add 方法的，也就是不能添加。

通过 update 来给 set 集合添加集合：

```
>>> ss=set('zyxzce')
>>> ss
{'e', 'z', 'y', 'x', 'c'}
>>> s.update(ss)
>>> s
{'d', 'e', 'z', 'y', 'b', 'x', 'a', 'c', 'g'}
```

difference 可以查看集合之间的区别：

```
>>> s.difference(ss)
{'g', 'b', 'a', 'd'}
```

list 和 dict 的类继承

list 和 dict 本身也是类。

在我们定义类的时候，通常不建议继承 list 和 dict。

因为在某些情况之下，用 c 语言写的 dict，不会调用我们自定义的重新覆盖的方法。

如果实在要继承，可以用 UserDict，如：

```
from collections import UserDict

class MyDict(UserDict):
   def __setitem__(self, key, value):
      super().__setitem__(key, value*2)

mydict = MyDict(a=1)
```

不建议的用法，这里只是顺带一提。

Python 中使用多线程可以达到多核 CPU 一起使用吗?

Python 中有一个被称为 Global Interpreter Lock（GIL）的东西，它会确保任何时候你的多个线程中，只有一个被执行。线程的执行速度非常之快，会让你误以为线程是并行执行的，但是实际上都是轮流执行。经过 GIL 这一道关卡处理，会增加执行的开销。
可以通过多进程实现多核任务。

什么是装饰器?

先上目录

1、装饰器是什么？
2、如何使用装饰器？
3、内置装饰器

装饰器是什么？

装饰器，顾名思义，就是增强函数或类的功能的一个函数。

这么说可能有点绕。
举个例子：如何计算函数的执行时间？
如下，你需要计算 add 函数的执行时间。

```
# 函数
def add(a, b):
    res = a + b
    return res
```

你可能会这么写

```
import time
def add(a, b)
    start_time = time.time()
    res = a + b
    exec_time = time.time() - start_time
    print("add 函数，花费的时间是：{}".format(exec_time))
    return res
```

这个时候，老板又让你计算减法函数（sub）的时间。不用装饰器的话，你又得重复写一段减法的代码。

```
def sub(a, b)
```

```
    start_time = time.time()
    res = a - b
    exec_time = time.time() - start_time
    print("sub 函数，花费的时间是：{}".format(exec_time))
    return res
```

这样显得很麻烦，也不灵活，**万一计算时间的代码有改动，你得每个函数都要改动。**

所以，我们需要引入装饰器。

使用装饰器之后的代码是这样的

```
import time
# 定义装饰器
def time_calc(func):
    def wrapper(*args, **kargs):
        start_time = time.time()
        f = func(*args,**kargs)
        exec_time = time.time() - start_time
        return f
    return wrapper

# 使用装饰器
@time_calc
def add(a, b):
    return a + b

@time_calc
def sub(a, b):
    return a - b
```

是不是看起来清爽多了？

装饰器的作用：增强函数的功能，确切的说，可以装饰函数，也可以装饰类。

装饰器的原理：函数是 python 的一等公民，函数也是对象。

定义装饰器

```
def decorator(func):
```

```
    def wrapper(*args,**kargs):
        # 可以自定义传入的参数
        print(func.__name__)
        # 返回传入的方法名参数的调用
        return func(*args,**kargs)
    # 返回内层函数函数名
    return wrapper
```

使用装饰器

假设 decorator 是定义好的装饰器。

方法一：不用语法糖@符号

```
# 装饰器不传入参数时
f = decorator(函数名)
# 装饰器传入参数时
f = (decorator(参数))(函数名)
```

方法二：采用语法糖@符号

```
# 已定义的装饰器
@decorator
def f():
    pass
# 执行被装饰过的函数
f()
```

装饰器可以传参，也可以不用传参。

自身不传入参数的装饰器（采用两层函数定义装饰器）

```
def login(func):
    def wrapper(*args,**kargs):
        print('函数名:%s'% func.__name__)
        return func(*args,**kargs)
    return wrapper
@login
def f():
    print('inside decorator!')
```

```
f()

# 输出:
# >> 函数名:f
# >> 函数本身:inside decorator!
```

自身传入参数的装饰器（采用三层函数定义装饰器）

```
def login(text):
    def decorator(func):
        def wrapper(*args,**kargs):
            print('%s----%s'%(text, func.__name__))
            return func(*args,**kargs)
        return wrapper
    return decorator
# 等价于 ==> (login(text))(f) ==> 返回 wrapper
@login('this is a parameter of decorator')
def f():
    print('2019-06-13')
# 等价于 ==> (login(text))(f)() ==> 调用 wrapper() 并返回 f()
f()

# 输出:
# => this is a parameter of decorator----f
# => 2019-06-13
```

内置装饰器

常见的内置装饰器有三种，@property、@staticmethod、`@classmethod`

@property

把类内方法当成属性来使用，必须要有返回值，相当于 getter;

假如没有定义 @func.setter 修饰方法的话，就是只读属性

```
class Car:
    def __init__(self, name, price):
        self._name = name
```

```
        self._price = price
    @property
    def car_name(self):
        return self._name
     # car_name 可以读写的属性
     @car_name.setter
     def car_name(self, value):
         self._name = value
     # car_price 是只读属性
     @property
     def car_price(self):
         return str(self._price) + '万'
benz = Car('benz', 30)
print(benz.car_name)   # benz
benz.car_name = "baojun"
print(benz.car_name)   # baojun
print(benz.car_price)  # 30 万
```

@staticmethod

静态方法，不需要表示自身对象的 self 和自身类的 cls 参数，就跟使用函数一样。

@classmethod

类方法，不需要 self 参数，但第一个参数需要是表示自身类的 cls 参数。

例子

```
class Demo(object):
    text = "三种方法的比较"
    def instance_method(self):
        print("调用实例方法")
    @classmethod
    def class_method(cls):
        print("调用类方法")
        print("在类方法中 访问类属性 text: {}".format(cls.text))
        print("在类方法中 调用实例方法 instance_method: {}".format(cls().instance_method()))
```

```
    @staticmethod
    def static_method():
        print("调用静态方法")
        print("在静态方法中 访问类属性 text: {}".format(Demo.text))
        print("在静态方法中 调用实例方法 instance_method: {}".format(Demo().instance_method()))
if __name__ == "__main__":
    # 实例化对象
    d = Demo()
    # 对象可以访问 实例方法、类方法、静态方法
    # 通过对象访问 text 属性
    print(d.text)
    # 通过对象调用实例方法
    d.instance_method()
    # 通过对象调用类方法
    d.class_method()
    # 通过对象调用静态方法
    d.static_method()
    # 类可以访问类方法、静态方法
    # 通过类访问 text 属性
    print(Demo.text)
    # 通过类调用类方法
    Demo.class_method()
    # 通过类调用静态方法
    Demo.static_method()
```

@staticmethod 和 @classmethod 的 **区别** 和 **使用场景**：

在上述例子中，我们可以看出，

区别

在定义静态类方法和类方法时，@staticmethod 装饰的静态方法里面，想要访问类属性或调用实例方法，必须需要把类名写上；

而@classmethod 装饰的类方法里面，会传一个 cls 参数，代表本类，这样就能够避免手写类名的硬编码。

在调用静态方法和类方法时，实际上写法都差不多，一般都是通过 类名.静态方法() 或 类名.类方法()。

也可以用实例化对象去调用静态方法和类方法，但为了和实例方法区分，最好还是用类去调用静态方法和类方法。

使用场景

所以，在定义类的时候，

假如**不需要用到与类相关的属性或方法**时，就用**静态方法@staticmethod**；

假如**需要用到与类相关的属性或方法，然后又想表明这个方法是整个类通用的**，而**不是对象特异的**，就可以使用**类方法@classmethod**。

在解释@函数装饰器之前，先说一下，类中的类方法和静态方法。

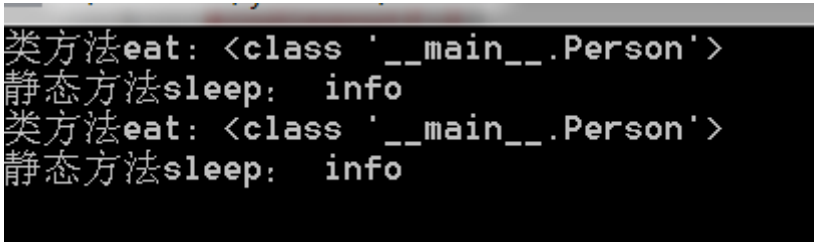
在 Python 中完全支持定义类方法、静态方法。这两种方法很相似，Python 它们都使用类来调用（ps：用对象调用也可以）。

区别在于：Python 会自动绑定**类方法**的第一个参数，类方法的第一个参数会自动绑定到类本身；但对于静态方法则不会自动绑定。

类方法用@classmethod 修饰，**静态方法**用@staticmethod 修饰，如下：

```
#coding=utf-8
class Person:
    @classmethod
    def eat(cls):
        print("类方法 eat:",cls)

    @staticmethod
    def sleep(p):
        print("静态方法 sleep: ",p)


Person.eat()
Person.sleep("info")

p = Person()

p.eat()
p.sleep('info')
```

控制台打印如下：



解释：eat 方法因为是@classmethod 修饰所以它是类方法，所以第 12 行调用时，不用传入任何参数，即可以调用，因为它会自动绑定类本身到第一个参数。
sleep 方法因为是@staticmethod 修饰所以它是静态方法，所以在第 13 行调用时，需要传入参数。再看 17、18 行，可以得出结论：不管是用类或者对象调用静态方法，Python 都不会对静态方法的第一个参数进行自动绑定。

上面的@classmethod 和 @staticmethod 其实就是函数装饰器，其中 classmethod 和 staticmethod 为 Python 中内置的函数。

使用@符号引用已有函数后，可用于修饰其他函数。

例如@函数 A 装饰 函数 B ，实际完成的步骤为：

1.将被装饰的函数 B 作为参数传给函数 A

2.将函数 B 替换为第 1 步的返回值。

什么是 Python 装饰器

一、引出装饰器概念

引入问题： 定义了一个函数，想在运行时动态的增加功能，又不想改动函数本身的代码？

示例：

希望对下列函数调用增加 log 功能，打印出函数调用：

```
1
2
3
```

```
def f1(x):  return x*2

def f2(x):  return x*x

def f3(x):  return x*x*x
```

方法一：直接修改原函数的定义

```
1
2
```

```
def f1(x):  print'call f1()'return x*2
```

```
3
```

```
def f2(x):  print 'call f2()' return x*x

def f3(x):  print 'call f3()' return x*x*x
```

思考：高阶函数

(1)、可以接受函数作为参数

(2)、可以返回函数

(3)、那么可否通过接收一个函数，内部对其包装，然后返回一个新函数，这样子动态的增强函数功能

方法二：通过高阶函数返回新函数

```
1
2
3
4
5
6
7
8
9
10
11
```

```
def f1(x):  return x*2



def new_fn(f):
        '''装饰器函数'''
        def fn(x):
                print 'call' + f.__name__  + '()'
                return f(x)
        return fn



g1  = new_fn(f1)

print g1(5)
```

输出结果：

```
========== RESTART: C:\Users\11xin\Desktop
callf1()
10
```


通过高阶函数达到我们的目的。

但是我们修改程序如下：

```
def f1(x):  return x*2



def new_fn(f):
        '''装饰器函数'''
        def fn(x):
                print 'call' + f.__name__  + '()'
                return f(x)
        return fn



f1  = new_fn(f1)
print f1(5)
```

输出结果：

```
========== RESTART: C:\Users\11xin\Desktop
callf1()
10
```


是一样的。但是这样子的话，f1 的原始定义函数就被彻底隐藏了。

二、装饰器概念

(1)、python 内置的 @ 语法就是为了简化装饰器的使用。

进行装饰器的修饰之后：

下面的代码一和代码二在 python 中是等价的。

```
                    代码一：
                    @new_fn
                    def f1(x):   return x*2
1
2
3
4                   print f1(5)
5
6
7
8                   代码二：
9                   def f1(x):   return x*2
10
11
12
                    f1   = new_fn(f1)

                    print f1(5)
```

输出结果：

```
callf1()
10
```


是一样的。

装饰器可以这么理解：通过高阶函数传递函数参数，新函数添加旧函数的需求，然后执行旧函数。

三、装饰器作用：

可以极大简化代码，避免每个函数编写重复性的代码

(1)、打印日志：@log (2)、检测性能：@performance

(3)、数据库事务：@transaction (4)、URL 路由：@post(’/register’)

四、装饰器函数三步走

(1)定义自己先要执行的函数

(2)

```
def new_fn(要执行的函数名字):
    def fn(要执行的参数函数参数/或者没有参数):
        添加要添加的函数功能
        return f(x)    函数的执行结果
    return fn
```

(3)装饰器进行修饰

Python 中的垃圾回收机制?

垃圾回收机制 GC 作为现代编程语言的自动内存管理机制，专注于两件事：1. 找到内存中无用的垃圾资源 2. 清除这些垃圾并把内存让出来给其他对象使用。GC 彻底把程序员从资源管理的重担中解放出来，让他们有更多的时间放在业务逻辑上。但这并不意味着码农就可以不去了解 GC，毕竟多了解 GC 知识还是有利于我们写出更健壮的代码。

Python 的垃圾回收机制

引子:#

我们定义变量会申请内存空间来存放变量的值，而内存的容量是有限的，当一个变量值没有用了（简称垃圾）就应该将其占用的内存给回收掉，而变量名是访问到变量值的唯一方式，所以当一个变量值没有关联任何变量名时，我们就无法再访问到该变量值了，该变量值就是一个垃圾会被 Python 解释的垃圾回收机制自动回收。。。

一、什么是垃圾回收机制？#

垃圾回收机制（简称 GC）是 Python 解释器自带一种机制，专门用来回收不可用的变量值所占用的内存空间

二、为什么要用垃圾回收机制？#

程序运行过程中会申请大量的内存空间，而对于一些无用的内存空间如果不及时清理的话会导致内存使用殆尽（内存溢出），导致程序崩溃，因此管理内存是一件重要且繁杂的事情，而 python 解释器自带的垃圾回收机制把程序员从繁杂的内存管理中解放出来。

三、垃圾回收机制原理分析#

Python 的 GC 模块主要运用了“**引用计数**”（reference counting）来跟踪和回收垃圾。在引用计数的基础上，还可以通过“**标记-清除**”（mark and sweep）解决容器对象可能产生的循环引用的问题，并且通过“**分代回收**”（generation collection）以空间换取时间的方式来进一步提高垃圾回收的效率。

1、什么是引用计数？#

引用计数就是：**变量值被变量名关联的次数**

如：name= ‘jason'

变量值 jason 被关联了一个 name，称之为引用计数为 1

引用计数增加：

x=10 （此时，变量值 10 的引用计数为 1）

y=x （此时，把 x 的内存地址给了 y，此时，x,y 都关联了 10，所以变量值 10 的引用计数为 2）

引用计数减少：

x=3（此时，x 与 10 解除了关联，与 3 建立了关联，变量 10 的引用计数为 1）

del y（del 的意思是解除变量名 y 与变量值 10 的关联关系，此时，变量 10 的引用计数为 0）

这样变量值 10 的引用计数为 0，其占用的内存地址就会被回收

2、引用计数扩展阅读？（折叠）#

引用计数机制执行效率问题：变量值被关联次数的增加或减少，都会引发引用计数机制的执行，这存在明显的效率问题 如果说执行效率还仅仅是引用计数机制的一个软肋的话，那么很不幸，引用计数机制还存在着一个致命的弱点，即**循环引用**（也称交叉引用）。

```
# 变量名 l1 指向列表 1，变量名 l2 指向列表 2，如下
>>> l1=['列表 1 中的第一个元素']  # 列表 1 被引用一次
>>> l2=['列表 2 中的第一个元素']  # 列表 2 被引用一次
>>> l1.append(l2)                # 把列表 2 追加到 l1 中作为第二个元素，列表 2 的引用计数为 2
>>> l2.append(l1)                # 把列表 1 追加到 l2 中作为第二个元素，列表 1 的引用计数为 2
# l1 与 l2
# l1 = ['列表 1 中的第一个元素',列表 2 的内存地址]
# l2 = ['列表 2 中的第一个元素',列表 1 的内存地址]
```

循环引用可以使一组对象的引用计数不为 0，然而这些对象实际上并没有被任何外部对象所引用，它们之间只是相互引用。这意味着不会再有人使用这组对象，应该回收这组对象所占用的内存空间，然后由于相互引用的存在，每一个对象的引用计数都不为 0，因此这些对象所占用的内存永远不会被释放。比如：

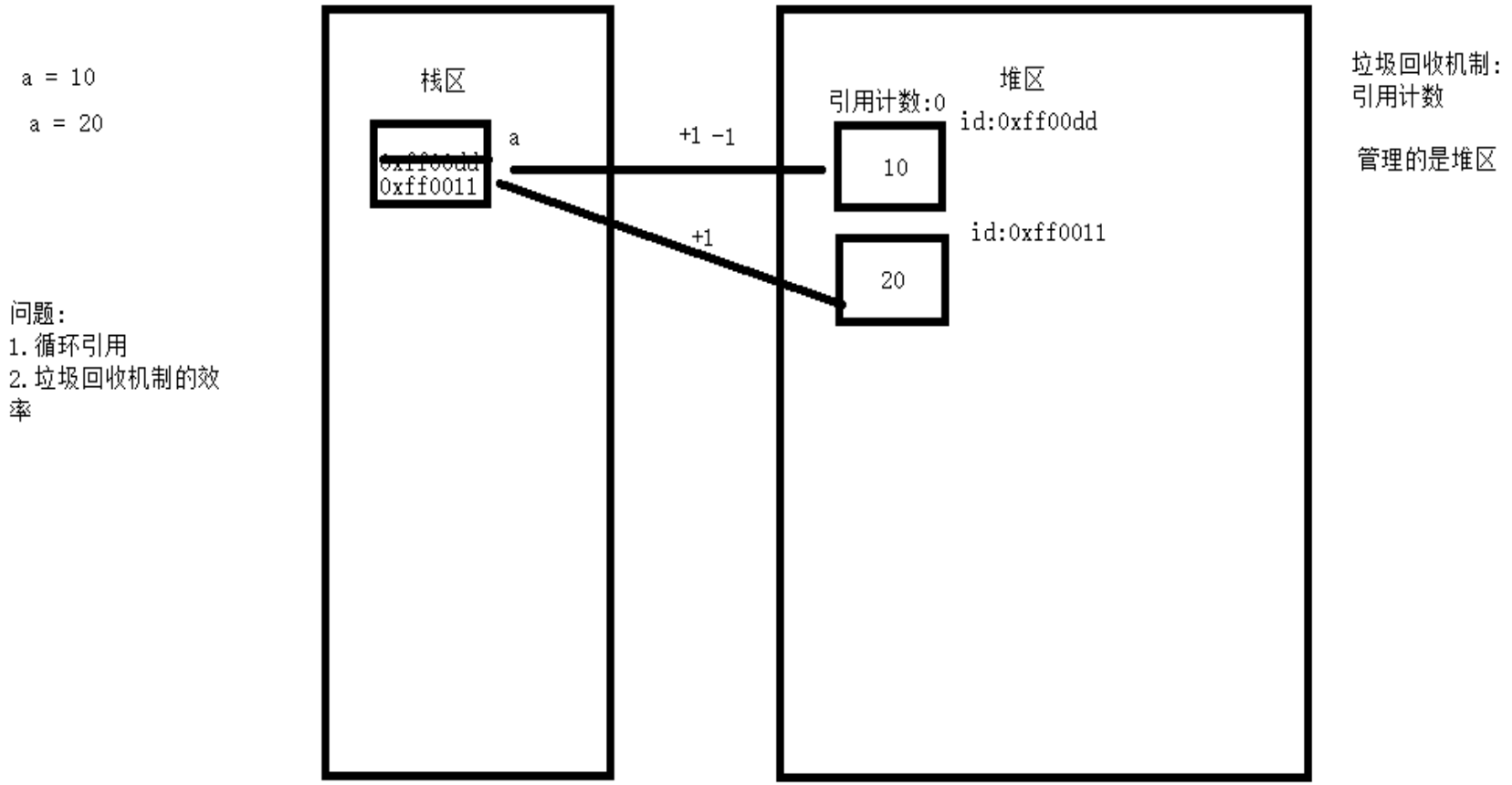
```
>>> l1
['列表 1 中的第一个元素', ['列表 2 中的第一个元素', [...]]]
>>> l2
['列表 2 中的第一个元素', ['列表 1 中的第一个元素', [...]]]
>>> l1[1][1][0]
'列表 1 中的第一个元素'
```

如果我们执行 del l1，列表 1 的引用计数=2-1，即列表 1 不会被回收，同理 del l2，列表 2 的引用计数=2-1，此时无论列表 1 还是列表 2 都没有任何名字关联，但是引用计数均不为 0，所以循环引用是致命的，这与手动进行内存管理所产生的内存泄露毫无区别 要解决这个问题，Python 引入了其他的垃圾收集机制来弥补引用计数的缺陷：**1、 “标记-清除” 2、 “分代回收”**

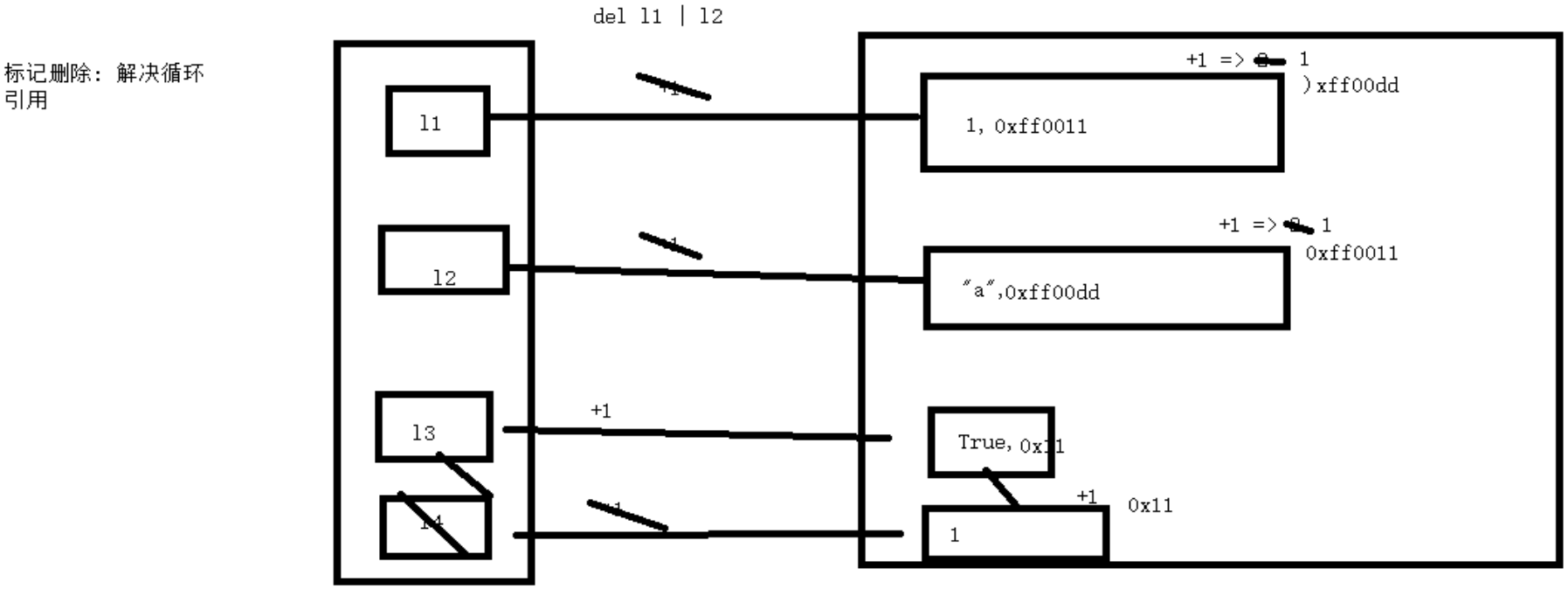
标记-清除#

容器对象（比如：list，set，dict，class，instance）都可以包含对其他对象的引用，所以都可能产生循环引用。而“标记-清除”计数就是为了解决循环引用的问题。

在了解标记清除算法前，我们需要明确一点，内存中有两块区域：堆区与栈区，在定义变量时，变量名存放于栈区，变量值存放于堆区，内存管理回收的则是堆区的内容，详解如下图

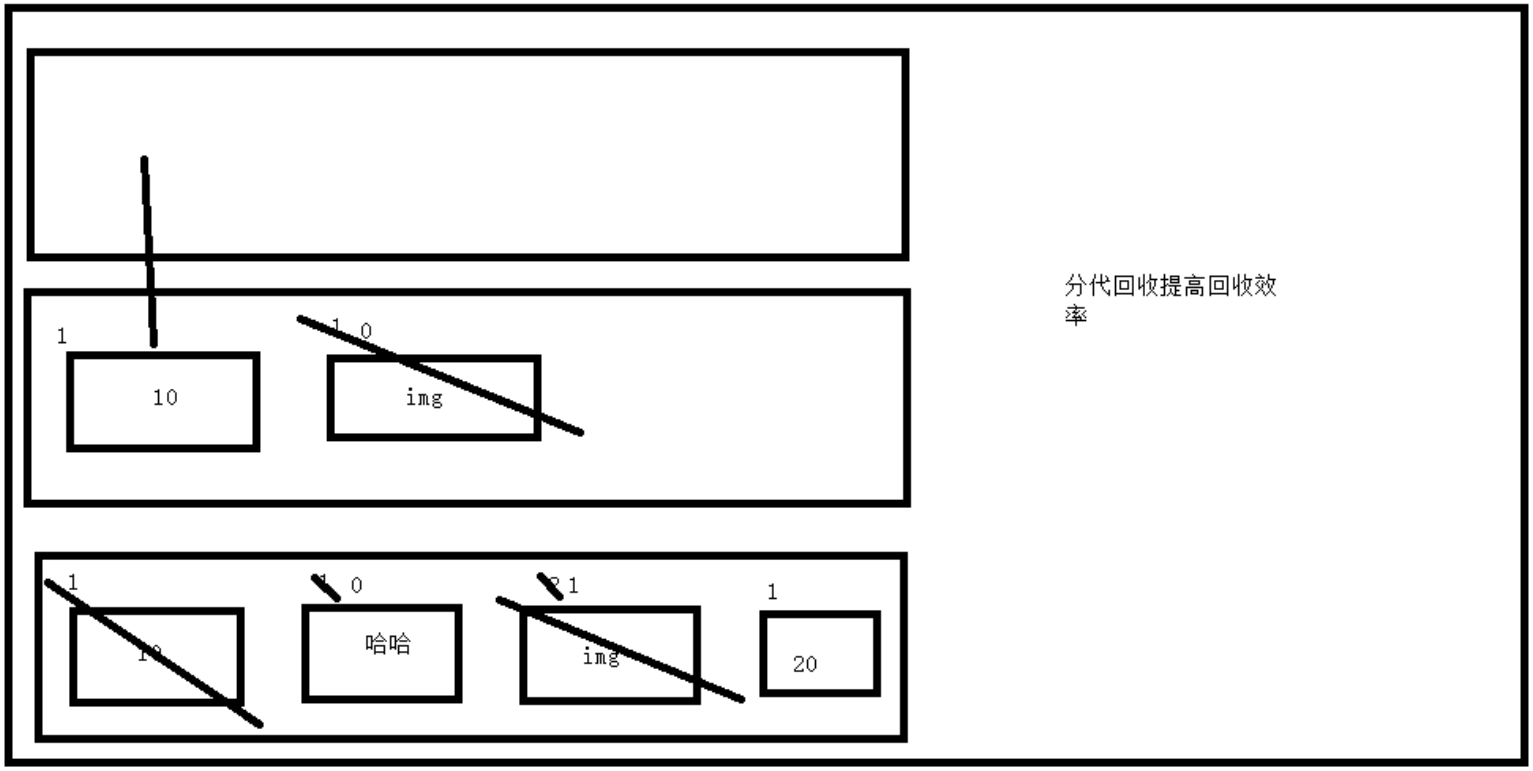


标记/清除算法的做法是当有效内存空间被耗尽的时候，就会停止整个程序，然后进行两项工作，第一项则是标记，第二项则是清除
标记：标记的过程其实就是，遍历所有的 GC Roots 对象(栈区中的所有内容或者线程都可以作为 GC Roots 对象），然后将所有 GC Roots 的对象可以直接或间接访问到的对象标记为存活的对象。
清除：清除的过程将遍历堆中所有的对象，将没有标记的对象全部清除掉。
GC roots 对象直接访问到的对象，插图如下



用图形解释，环引用的例子中的 l1 与 l2，在什么时候启动标记清除，标记清除的整个过程
分代回收#
背景：
基于引用计数的回收机制，每次回收内存，都需要把所有对象的引用计数都遍历一遍，这是非常消耗时间的，于是引入了分代回收来提高回收效率，采用“空间换时间的策略”。
什么是分代回收?
分代：

分代回收的核心思想是：在多次扫描的情况下，都没有被回收的变量，gc 机制就会认为，该变量是常用变量，gc 对其扫描的频率会降低，具体实现原理如下：
分代指的是根据存活时间来为变量划分不同等级（也就是不同的代）
新定义的变量，放到新生代这个等级中，假设每隔 1 分钟扫描新生代一次，如果发现变量依然被引用，那么该对象的权重（权重本质就是个整数）加一，当变量的权重大于某个设定得值（假设为 3），会将它移动到更高一级的青春代，青春代的 gc 扫描的频率低于新生代（扫描时间间隔更长），假设 5 分钟扫描青春代一次，这样每次 gc 需要扫描的变量的总个数就变少了，节省了扫描的总时间，接下来，青春代中的对象，也会以同样的方式被移动到老年代中。也就是等级（代）越高，被垃圾回收机制扫描的频率越低
回收：
回收依然是使用引用计数作为回收的依据
图示：



缺点：

例如一个变量刚刚从新生代移入青春代，该变量的绑定关系就解除了，该变量应该被回收，青春代的扫描频率低于新生代，所以该变量的回收时间被延迟。

Python 垃圾回收机制--完美讲解!

什么是 lambda 表达式?

简单来说，lambda 表达式通常是当你需要使用一个函数，但是又不想费脑袋去命名一个函数的时候使用，也就是通常所说的匿名函数。

lambda 表达式一般的形式是：关键词 lambda 后面紧接一个或多个参数，紧接一个冒号“：”，紧接一个表达式

什么是深拷贝和浅拷贝?

赋值（=），就是创建了对象的一个新的引用，修改其中任意一个变量都会影响到另一个。

浅拷贝 copy.copy：创建一个新的对象，但它包含的是对原始对象中包含项的引用（如果用引用的方式修改其中一个对象，另外一个也会修改改变）

深拷贝：创建一个新的对象，并且递归的复制它所包含的对象（修改其中一个，另外一个不会改变）{copy 模块的 deep.deepcopy() 函数}

双等于和 is 有什么区别?

==比较的是两个变量的 value，只要值相等就会返回 True

is 比较的是两个变量的 id，即 id(a) == id(b)，只有两个变量指向同一个对象的时候，才会返回 True

但是需要注意的是，比如以下代码：

```
a = 2
b = 2
print(a is b)
```

按照上面的解释，应该会输出 False，但是事实上会输出 True，这是因为 Python 中对小数据有缓存机制，-5~256 之间的数据都会被缓存。

其它 Python 知识点

类型转换

- list(x)
- str(x)
- set(x)
- int(x)

- tuple(x)

try...except

list

- lst[a:b]：左闭右开
- lst.append(value)：在末尾添加元素，复杂度 O(1)
- lst.pop()：弹出列表末尾元素，复杂度 O(1)
- lst.pop(index)：弹出任意位置元素，将后面的元素前移，复杂度 O(n)
- lst.insert(index, value)：插入元素，后面的元素后移，复杂度 O(n)
- lst.remove(value)：移除等于 value 的第一个元素，后面的元素前移，复杂度 O(n)
- lst.count(value)：计数值为 value 的元素个数
- lst.sort(reverse = False)：排序，默认升序

参考

- 生成器 - 廖雪峰的官方网站
- Python 中的 is 和==的区别

InnoDB 引擎和 Myisam 有什么区别？简历上写着就被问了 SQL Alchemy 的实现原理？Python 语言的什么机制？Flask 路由的原理和过程？路由匹配里面的正则式干嘛的？正则匹配的优先级？Linux 的 I/O 复用机制？Select poll epoll 的区别和实现原理？select 为什么只能 1024？触发方式上有什么不同？Restful 是啥？HTTP 中 Get 和 Post 的区别？对于频繁的 GET 和 POST 请求会有什么区别？HTTP1.0 和 1.1 和 2.0 的区别？HTTP1.1 里面和 cache 有关的属性？HTTP 和 HTTPS 的区别？HTTPS 建立连接的过程？证书认证是啥？最近在看什么书？算法题：最长递增子序列的数量（Longest Increasing Subsequence）。

一面：
1.自我介绍。
2.介绍“工大小美”项目相关。
3.Python 中的 GIL(全局解释器锁)，以及哪一种状况下使用 python 的多线程性能有较大的提高。
4.项目中用到了 SQLite 数据库，若是有多个副本，怎么保证数据的一致性。

数据一致性-分区可用性-性能——多副本强同步数据库系统实现之我见

1 背景 1

2 问题一：数据一致性 3
3 问题二：分区可用性 6
4 问题三：性能 8
5 总结 10
6 问题四：一个极端场景的分析 10

1. 背景

最近，@阿里正祥（阳老师）发了上面的一条微博，谁知一石激起千层浪，国内各路数据库领域的朋友在此条微博上发散出无数新的话题，争吵有之，激辩有之，抨击有之，不一而足。总体来说，大家重点关注其中的一点：
在不使用共享存储的情况下，传统 RDBMS（例如：Oracle/MySQL/PostgreSQL 等），能否做到在主库出问题时的数据零丢失。
这个话题被引爆之后，我们团队内部也经过了激烈的辩论，多方各执一词。辩论的过程中，差点就重现了乌克兰议会时场景...略
庆幸的是，在我的铁腕统治之下，同学们还是保持着只关注技术，就事论事的撕逼氛围，没有上升到相互人身攻击的层次。激辩的结果，确实是收获满满，当时我就立即发了一条微博，宣泄一下自己愉悦的心情☺略
微博发出之后，也有一些朋友回复是否可以将激辩的内容写出来，独乐乐不如众乐乐。我一想也对，强数据同步，数据一致性，性能，分区可用性，Paxos，Raft，CAP 等一系列知识，我也是第一次能够较好的组织起来，写下来，一来可以加深自己的印象，二来也可以再多混一点虚名，何乐而不为☺
这篇博客文章接下来的部分，将跳出任何一种数据库，从原理的角度上来分析下面的几个问题：

- 问题一：数据一致性。在不使用共享存储的情况下，传统 RDBMS（例如：Oracle/MySQL/PostgreSQL 等），能否做到在主库出问题时的数据零丢失。
- 问题二：分区可用性。有多个副本的数据库，怎么在出现各种问题时保证系统的持续可用？
- 问题三：性能。不使用共享存储的 RDBMS，为了保证多个副本间的数据一致性，是否会损失性能？如何将性能的损失降到最低？
- 问题四：一个极端场景的分析。

1. 问题一：数据一致性

问：脱离了共享存储，传统关系型数据库就无法做到主备强一致吗？
答：我的答案，是 No。哪怕不用共享存储，任何数据库，也都可以做到主备数据的强一致。Oracle 如此，MySQL 如此，PostgreSQL 如此，OceanBase 也如此。
如何实现主备强一致？大家都知道数据库中最重要的一个技术：WAL（Write-Ahead-Logging）。更新操作写日志（Oracle Redo Log，MySQL Binlog 等），事务提交时，保证将事务产生的日志先刷到磁盘上，保证整个事务的更新操作数据不丢失。那实现数据库主备数据强一致的方法也很简单：

1. 事务提交的时候，同时发起两个写日志操作，一个是将日志写到本地磁盘的操作，另一个是将日志同步到备库并且确保落盘的操作；
2.
3. 主库此时等待两个操作全部成功返回之后，才返回给应用方，事务提交成功；
4.

整个事务提交操作的逻辑，如下图所示：略
上图所示，由于事务提交操作返回给应用时，事务产生的日志在主备两个数据库上都已经存在了，强同步。因此，此时主库 Crash 的话，备库提供服务，其数据与主库是一致的，没有任何事务的数据丢失问题。主备数据强一致实现。用过 Oracle 的朋友，应该都

知道 Oracle 的 Data Guard，可工作在 最大性能，最大可用，最大保护 三种模式下，其中第三种 最大保护 模式，采用的就是上图中的基本思路。
实现数据的强同步实现之后，接下来到了考虑可用性问题。现在已经有主备两个数据完全一致的数据库，备库存在的主要意义，就是在主库出故障时，能够接管应用的请求，确保整个数据库能够持续的提供服务：主库 Crash，备库提升为主库，对外提供服务。此时，又涉及到一个决策的问题，主备切换这个操作谁来做？人当然可以做，接收到主库崩溃的报警，手动将备库切换为主库。但是，手动的效率是低下的，更别提数据库可能会随时崩溃，全部让人来处理，也不够厚道。一个 HA（High Availability）检测工具应运而生：HA 工具一般部署在第三台服务器上，同时连接主备，当其检测到主库无法连接，就切换备库，很简单的处理逻辑，如下图所示：略
HA 软件与主备同时连接，并且有定时的心跳检测。主库 Crash 后，HA 探测到，发起一个将备库提升为主库的操作（修改备库的 VIP 或者是 DNS，可能还需要将备库激活等一系列操作），新的主库提供对外服务。此时，由于主备的数据是通过日志强同步的，因此并没有数据丢失，数据一致性得到了保障。
有了基于日志的数据强同步，有了主备自动切换的 HA 软件，是不是就一切万事大吉了？我很想说是，确实这个架构已经能够解决 90%以上的问题，但是这个架构在某些情况下，也埋下了几个比较大的问题。
首先，一个一目了然的问题，主库 Crash，备库提升为主库之后，此时的数据库是一个单点，原主库重启的这段时间，单点问题一直存在。如果这个时候，新的存储再次 Crash，整个系统就处于不可用状态。此问题，可以通过增加更多副本，更多备库的方式解决，例如 3 副本（一主两备），此处略过不表。
其次，在主备环境下，处理主库挂的问题，算是比较简单的，决策简单：主库 Crash，切换备库。但是，如果不是主库 Crash，而是网络发生了一些问题，如下图所示：略
若 Master 与 Slave 之间的网络出现问题，例如：断网，网络抖动等。此时数据库应该怎么办？Master 继续提供服务？Slave 没有同步日志，会数据丢失。Master 不提供服务？应用不可用。在 Oracle 中，如果设置为 最大可用 模式，则此时仍旧提供服务，允许数据不一致；如果设置为 最大保护 模式，则 Master 不提供服务。因此，在 Oracle 中，如果设置为 最大保护 模式，一般建议设置两个或以上的 Slave，任何一个 Slave 日志同步成功，Master 就继续提供服务，提供系统的可用性。
网络问题不仅仅出现在 Master 和 Slave 之间，同样也可能出现在 HA 与 Master，HA 与 Slave 之间。考虑下面的这种情况：略
HA 与 Master 之间的网络出现问题，此时 HA 面临两个抉择：

1. HA 到 Master 之间的连接不通，认为主库 Crash。选择将备库提升为主库。但实际上，只是 HA 到 Master 间的网络有问题，原主库是好的（没有被降级为备库，或者是关闭），仍旧能够对外提供服务。新的主库也可以对外提供服务。两个主库，产生双写问题，最为严重的问题。
2. HA 到 Master 之间的连接不通，认为是网络问题，主库未 Crash。HA 选择不做任何操作。但是，如果这时确实是主库 Crash 了，HA 不做操作，数据库不对外提供服务。双写问题避免了，但是应用的可用性受到了影响。

最后，数据库会出现问题，数据库之间的网络会出现问题，那么再考虑一层，HA 软件本身也有可能出现问题。如下图所示：略
如果是 HA 软件本身出现了问题，怎么办？我们通过部署 HA，来保证数据库系统在各种场景下的持续可用，但是 HA 本身的持续可用谁来保证？难道我们需要为 HA 做主备，然后再 HA 之上再做另一层 HA？一层层加上去，子子孙孙无穷尽也 … …
其实，上面提到的这些问题，其实就是经典的分布式环境下的一致性问题（Consensus），近几年比较火热的 Lamport 老爷子的 Paxos 协议，Stanford 大学最近发表的 Raft 协议，都是为了解决这一类问题。（对 Raft 协议感兴趣的朋友，可以再看一篇 Raft 的动态演示 PPT：Understandable Distributed Consensus）

1. 问题二：分区可用性

前面，我们回答了第一个问题，数据库如果不使用共享存储，能否保证主备数据的强一致？答案是肯定的：可以。但是，通过前面的分析，我们又引出了第二个问题：如何保证数据库在各种情况下的持续可用？至少前面提到的 HA 机制无法保证。那么是否可以引入类似于 Paxos，Raft 这样的分布式一致性协议，来解决上面提到的各种问题呢？
答案是可以的，我们可以通过引入类 Paxos，Raft 协议，来解决上面提到的各类问题，保证整个数据库系统的持续可用。考虑仍旧是两个数据库组成的主备强一致系统，仍旧使用 HA 进行主备监控和切换，再回顾一下上一节新引入的两个问题：

1. HA 软件自身的可用性如何保证？
2.
3. 如果 HA 软件无法访问主库，那么这时到底是主库 Crash 了呢？还是 HA 软件到主库间的网络出现问题了呢？如何确保不会同时出现两个主库，不会出现双写问题？
4.
5. 如何在解决上面两个问题的同时，保证数据库的持续可用？
6.

为了解决这些问题，新的系统如下所示：略
相对于之前的系统，可以看到这个系统的复杂性明显增高，而且不止一成。数据库仍旧是一主一备，数据强同步。但是除此之外，多了很多变化，这些变化包括：

1. 数据库上面分别部署了 HA Client;
2.
3. 原来的一台 HA 主机，扩展到了 3 台 HA 主机。一台是 HA Master，其余的为 HA Participant;
4.
5. HA 主机与 HA Client 进行双向通讯。HA 主机需要探测 HA Client 所在的 DB 是否能够提供服务，这个跟原有一致。但是，新增了一条 HA Client 到 HA 主机的 Master Lease 通讯。
6.

这些变化，能够解决上面的两个问题吗？让我们一个一个来分析。首先是：HA 软件自身的可用性如何保证？
从一台 HA 主机，增加到 3 台 HA 主机，正是为了解决这个问题。HA 服务，本身是无状态的，3 台 HA 主机，可以通过 Paxos/Raft 进行自动选主。选主的逻辑，我这里就不做赘述，不是本文的重点，想详细了解其实现的，可以参考互联网上洋洋洒洒的关于 Paxos/Raft 的相关文章。总之，通过部署 3 台 HA 主机，并且引入 Paxos/Raft 协议，HA 服务的高可用可以解决。HA 软件的可用性得到了保障。
第一个问题解决，再来看第二个问题：如何识别出当前是网络故障，还是主库 Crash？如何保证任何情况下，数据库有且只有一个主库提供对外服务？
通过在数据库服务器上部署 HA Client，并且引入 HA Client 到 HA Master 的租约（Lease）机制，这第二个问题同样可以得到完美的解决。所谓 HA Client 到 HA Master 的租约机制，就是说图中的数据库实例，不是永远持有主库（或者是备库）的权利。当前主库，处于主库状态的时间是有限制的，例如：10 秒。每隔 10 秒，HA Client 必须向 HA Master 发起一个新的租约，续租它所在的数据库的主库状态，只要保证每 10 秒收到一个来自 HA Master 同意续租的确认，当前主库一直不会被降级为备库。
第二个问题，可以细分为三个场景：

- 场景一：主库 Crash，但是主库所在的服务器正常运行，HA Client 运行正常
-

主库 Crash，HA Client 正常运行。这种场景下，HA Client 向 HA Master 发送一个放弃主库租约的请求，HA Master 收到请求，直接将备库提升为主库即可。原主库起来之后，作为备库运行。

- 场景二：主库所在的主机 Crash。（主库和 HA Client 同时 Crash）
-

此时，由于 HA Client 和主库同时 Crash，HA Master 到 HA Client 间的通讯失败。这个时候，HA Master 还不能立即将备库提升为主库，因为区分不出场景二和接下来的场景三（网络问题）。因此，HA Master 会等待超过租约的时间（例如：12 秒），如果租约时间之内仍旧没有续租的消息。那么 HA Master 将备库提升为主库，对外提供服务。原主库所在的主机重启之后，以备库的状态运行。

- 场景三：主库正常，但是主库到 HA Master 间的网络出现问题
-

对于 HA Master 来说，是区分不出场景二和场景三的。因此，HA Master 会以处理场景二同样的逻辑处理场景三。等待超过租约的时间，没有收到续租的消息，提升原备库为主库。但是在提升备库之前，原主库所在的 HA Client 需要做额外的一点事。原主库 HA Client 发送给 HA Master 的续租请求，由于网络问题，一直没有得到响应，超过租约时间，主动将本地的主库降级为备库。如此一来，待 HA Master 将原备库提升为主库时，原来的主库已经被 HA Client 降级为备库。双主的情况被杜绝，应用不可能产生双写。

通过以上三个场景的分析，问题二同样在这个架构下被解决了。而解决问题二的过程中，系统最多需要等待租约设定的时间，如果租约设定为 10 秒，那么出各种问题，数据库停服的时间最多为 10 秒，基本上做到了持续可用。这个停服的时间，完全取决于租约的时间设置。

到这儿基本可以说，要实现一个持续可用（分区可用性保证），并且保证主备数据强一致的数据库系统，是完全没问题的。在现有数据库系统上做改造，也是可以的。但是，如果考虑到实际的实现，这个复杂度是非常高的。数据库的主备切换，是数据库内部实现的，此处通过 HA Master 来提升主库；通过 HA Client 来降级备库；保证数据库崩溃恢复后，恢复为备库；通过 HA Client 实现主库的租约机制；实现 HA 主机的可用性；所有的这些，在现有数据库的基础上实现，都有着相当的难度。能够看到这儿，而且有兴趣的朋友，可以针对此问题进行探讨☺

1. 问题三：性能

数据一致性，通过日志的强同步，可以解决。分区可用性，在出现任何异常情况时仍旧保证系统的持续可用，可以在数据强同步的基础上引入 Paxos/Raft 等分布式一致性协议来解决，虽然这个目前没有成熟的实现。接下来再让我们来看看一个很多朋友都很感兴趣的问题：如何在保证强同步的基础上，同时保证高性能？回到我们本文的第一幅图：略

为了保证数据强同步，应用发起提交事务的请求时，必须将事务日志同步到 Slave，并且落盘。相对于异步写 Slave，同步方式多了一次 Master 到 Slave 的网络交互，同时多了一次 Slave 上的磁盘 sync 操作。反应到应用层面，一次 Commit 的时间一定是增加了，具体增加了多少，要看主库到备库的网络延时和备库的磁盘性能。

为了提高性能，第一个很简单的想法，就是部署多个 Slave，只要有一个 Slave 的日志同步完成返回，加上本地的 Master 日志也已经落盘，提交操作就可以返回了。多个 Slave 的部署，对于消除瞬时的网络抖动，非常有效果。在 Oracle 的官方建议中，如果使用最大保护模式，也建议部署多个 Slave，来最大限度的消除网络抖动带来的影响。如果部署两个 Slave，新的部署架构图如下所示：略

新增一个 Slave，数据三副本。两个 Slave，只要有一个 Slave 日志同步完成，事务就可以提交，极大地减少了某一个网络抖动造成的影响。增加了一个副本之后，还能够解决当主库 Crash 之后的数据安全性问题，哪怕主库 Crash，仍旧有两个副本可以提供服务，不会形成单点。
但是，在引入数据三副本之后，也新引入了一个问题：主库 Crash 的时候，到底选择哪一个备库作为新的主库？当然，选主的权利仍旧是 HA Master 来行使，但是 HA Master 该如何选择？这个问题的简单解决可以使用下面的几个判断标准：

1. 日志优先。两个 Slave，哪个 Slave 拥有最新的日志，则选择这个 Slave 作为新的主库。
2.
3. 主机层面排定优先级。如果两个 Slave 同时拥有最新的日志，那么该如何选择？此时，选择任何一个都是可以的。例如：可以根据 Slave 主机 IP 的大小进行选择，选择 IP 小的 Slave 作为新的主库。同样能够解决问题。
4.

新的主库选择出来之后，第一件需要做的事，就是将新的 Master 和剩余的一个 Slave，进行日志的同步，保证二者日志达到一致状态后，对应用提供服务。此时，三副本问题就退化为了两副本问题，三副本带来的防止网络抖动的红利消失，但是由于两副本强同步，数据的可靠性以及一致性仍旧能够得到保障。
当然，除了这一个简单的三副本优化之外，还可以做其他更多的优化。优化的思路一般就是同步转异步处理，例如事务提交写日志操作；使用更细粒度的锁；关键路径可以采用无锁编程等。
多副本强同步，做到极致，并不一定会导致系统的性能损失。极致应该是什么样子的？我的想法是：

- 对于单个事务来说，RT 增加。其响应延时一定会增加（至少多一个网络 RT，多一次磁盘 Sync）；
-
- 对整个数据库系统来说，吞吐量不变。远程的网络 RT 和磁盘 Sync 并不会消耗本地的 CPU 资源，本地 CPU 的开销并未增大。只要是异步化做得好，整个系统的吞吐量，并不会由于引入强同步而降低。
-
-

1. 总结

洋洋洒洒写了一堆废话，最后做一个小小的总结：

- 能够看到这里的朋友，绝逼都是真爱，谢谢你们!!
-
- 各种主流关系型数据库系统是否可以实现主备的强一致，是否可以保证不依赖于存储的数据一致性？
-

  可以。Oracle 有，MySQL 5.7，阿里云 RDS，网易 RDS 都有类似的功能。

- 目前各种关系型数据库系统，能否在保证主备数据强一致的基础上，提供系统的持续可用和高性能？
-

  可以做，但是难度较大，目前主流关系型数据库缺乏这个能力。

-

1. 问题四：一个极端场景的分析

意犹未尽，给仍旧在坚持看的朋友预留一个小小的作业。考虑下面这幅图：如果用户的提交操作，在图中的第 4 步完成前，或者是第 4 步完成后第 5 步完成前，主库崩溃。此时，备库有最新的事务提交记录，崩溃的主库，可能有最新的提交记录（第 4 步完成，第 5 步前崩溃），也可能没有最新的记录（第 4 步前崩溃），系统应该如何处理？略

文章在博客上放出来之后，发现大家尤其对这最后一个问题最感兴趣。我选择了一些朋友针对这个问题发表的意见，仅供参考。

@淘宝丁奇

最后那个问题其实本质上跟主备无关。简化一下是，在单库场景下，db 本地事务提交完成了，回复 ack 前 crash，或者 ack 包到达前客户端已经判定超时...所以客户端只要没有收到明确成功或失败，临界事务两种状态都是可以接受的。主备环境下只需要保证系统本身一致。

将丁奇意见用图形化的方式表示出来，就是下面这幅图：略

此图，相对于问题四简化了很多，数据库没有主备，只有一个单库。应用发起 Commit，在数据库上执行日志落盘操作，但是在返回应用消息时失败（网络原因？超时？）。虽然架构简化了，但是问题大同小异，此时应用并不能判断出本次 Commit 是成功还是失败，这个状态，需要应用程序的出错处理逻辑处理。

@ArthurHG

最后一个问题，关键是解决服务器端一致性的问题，可以让 master 从 slave 同步，也可以让 slave 回滚，因为客户端没有收到成功消息，所以怎么处理都行。服务器端达成一致后，客户端可以重新提交，为了实现幂等，每个 transaction 都分配唯一的 ID；或者客户端先查询，然后根据结果再决定是否重新提交。

其实，最终的这个问题，更应该由做应用的同学来帮助解答：

如果应用程序在提交 Commit 操作，但是最后 Catch 到网络或者是超时的异常时，是怎么处理的？

5.怎样保证客户端和服务器端数据的一致性（数据的同步）

可以把客户端的数据当作一个整体，然后用 MD5 对它进行加密，服务器端也一样，这样，当他们的 MD5 值不一样时就能判断出他们的数据不同步了。然后再通过其他的方法把不同的部分上传或下载下来

5.MySQL 中的索引，B+树，事务。

6.TCP 三次握手，四次挥手。

7.单链表反转。

单链表反转的原理和 python 代码实现

链表是一种基础的数据结构，也是算法学习的重中之重。其中单链表反转是一个经常会被考察到的知识点。

单链表反转是将一个给定顺序的单链表通过算法转为逆序排列，尽管听起来很简单，但要通过算法实现也并不是非常容易。现在来给大家简单介绍一下单链表反转算法实现的基本原理和 python 代码实现。

算法基本原理及 python 代码

1、方法一：三个指针遍历反转

算法思想：使用 3 个指针遍历单链表，逐个链接点进行反转。

（1）分别用 p,q 两个指针指定前后两个节点。其中 p.next = q

```
（2）将 p 指针指向反方向。
（3）将 q 指针指向 p。q.next = p，同时用 r 代表剩余未反转节点。
（4）将 p,q 指针同时后移一位，回到步骤（2）的状态。
（5）r 指针指向剩余未反转节点。循环执行（3）之后的操作。
# 详细版
def reverse01(head):
    if head == None:
    return None
# 分别用 p,q 两个指针指定先后两个节点
    p = head
    q = head.next

# 将 p 节点反转，head 节点只能指向 None
    p.next = None

# 当存在多个后续节点时，循环执行
    while q:
        r = q.next # 用 r 表示后面未反转节点
        q.next = p # q 节点反转指向 p
        p = q
        q = r # p,q 节点后移一位，循环执行后面的操作
        return p

# 精简版
def reverse01(head):
    if not head:
        return None
        p,q,p.next = head,head.next,None
        while q:
            q.next,p,q = p,q,q.next
            return p
```

2、方法二：尾插法反转
算法思想：固定头节点，然后将后面的节点从前到后依此插入到次节点的位置，最后再将头节点移动到尾部。

```
# 详细版
def reverse02(head):
# 判断链表的节点个数
    if head == None or head.next == None:
        return head

        p = head.next
# 循环反转
        while p.next:
            q = p.next
            p.next = q.next
            q.next = head.next
            head.next = q

            # 将头节点移动到尾部
            p.next = head
            head = head.next
            p.next.next = None

            return head

# 精简版
def reverse02(head):
    if not head or not head.next:
        return head
        p = head.next
        while p.next:
            q = p.next
```

```
        p.next,q.next,head.next = q.next,head.next,q
        p.next,head,p.next.next = head,head.next,None
        return head
```

3、方法三：递归方式反转

算法思想：把单链表的反转看作头节点 head 和后续节点 head.next 之间的反转，循环递归。

```
def reverse03(head):
    if head.next == None:
        return head

new_head = reverse03(head.next)
head.next.next = head
head.next = None

return new_head
```

leetcode 精简代码示例

单链表的反转逻辑思路比较清晰，因此关于单链表反转重在考查代码的经精简度，而 Python 可以实现代码的极度简化，如下：

```
def reverse04(head):
    curr,pre = head,None
    while curr:
        curr.next,pre,curr = pre,curr,curr.next
        return pre
```

leetcode 相关算法习题（92.反转链表 II）

利用以上算法思想完成 leetcode 习题：92.反转链表 II

习题描述：

反转从位置 m 到 n 的链表。请使用一趟扫描完成反转。

说明:

1 ≤ m ≤ n ≤ 链表长度。

```
# 算法思想：采用尾插法反转思想（方法二）

# Definition for singly-linked list.
# class ListNode(object):
```

```
# def __init__(self, x):
# self.val = x
# self.next = None

class Solution(object):
def reverseBetween(self, head, m, n):
"""
:type head: ListNode
:type m: int
:type n: int
:rtype: ListNode
"""
root = ListNode(0)
root.next = head
Head = root #  指定一个头部变量（方法二中固定的 head）

for i in range(m-1):
Head = Head.next

if Head.next == None:
return head

pre = Head.next
while pre.next and m < n:
curr = pre.next
pre.next = curr.next
curr.next = Head.next
Head.next = curr
m += 1

#  由于 m 之前的元素不需要反转，因此用 root.next 代替方法二中的 head
```

```
return root.next
```

算法 1（非递归实现）：使用 3 个指针，分别指向前序结点、当前结点和后序结点，遍历链表，逐个逆转，时间复杂度 O(n)：

```
def reverse1(head):
    if head is None or head.next is None:
        return head
    current = head
    pre = None
    pnext = None
    while current is not None:
        pnext = current.next
        current.next = pre
        pre = current
        current = pnext

    return pre
```

算法 2（递归实现）： 不断逆转当前结点，直到链表尾端，时间复杂度 O(n)：

```
def reverse2(current):
    if current.next is None:
        return current
    pnext = current.next
    current.next = None
    reversed = reverse2(pnext)
    pnext.next = current

    return reversed
```

算法 3（递归实现）：和算法 2 类似，不过递归传入当前结点和前序结点，代码可读性要好点，时间复杂度 O(n)：

```
def reverse3(current, pre):
    if current.next is None:
        current.next = pre
```

```
            return current
        else:
            pnext = current.next
            current.next = pre
            return reverse3(pnext, current)
```

所有完整代码如下:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# @File       : ReverseSingleLinkImpl.py

class Node:
    data = 0
    next = None


def build_single_link_list(nodeArr):
    head_node = None
    next_node = None
    for i in nodeArr:
        node = Node()
        node.data = i
        node.next = None
        if head_node is None:
            head_node = node
            next_node = head_node
        else:
            next_node.next = node
            next_node = node

    return head_node
```

```
def reverse1(head):
    if head is None or head.next is None:
        return head
    current = head
    pre = None
    pnext = None
    while current is not None:
        pnext = current.next
        current.next = pre
        pre = current
        current = pnext

    return pre

def reverse2(current):
    if current.next is None:
        return current
    pnext = current.next
    current.next = None
    reversed = reverse2(pnext)
    pnext.next = current

    return reversed

def reverse3(current, pre):
    if current.next is None:
        current.next = pre
        return current
    else:
        pnext = current.next
```

```
        current.next = pre
        return reverse3(pnext, current)

if __name__ == '__main__':
    head = build_single_link_list([1, 2, 3, 4, 5])
    head = reverse1(head)
    print("impl1:")
    while head is not None:
        print(head.data)
        head = head.next
    print("impl2:")
    head = build_single_link_list([1, 2, 3, 4, 5])
    head = reverse2(head)
    while head is not None:
        print(head.data)
        head = head.next
    print("impl3:")
    head = build_single_link_list([1, 2, 3, 4, 5])
    head = reverse3(head,None)
    while head is not None:
        print(head.data)
        head = head.next
```

8.广度优先周游打印二叉树。

从上到下打印出二叉树的每个节点，同一层的节点按照从左到右的顺序打印。

例如:

给定二叉树: [3,9,20,null,null,15,7],

```
    3
   / \
  9  20
    /  \
```

```
  15   7
```
返回：
```
[3,9,20,15,7]
```
显然，这道题就是广度优先遍历(BFS),那么问题就是代码怎么去写

思路 1： 用字典的方式遍历

思路

将第一层的节点存入到一个字典中，

之后对第一层的每一个节点，按从左到右的顺序：

- 将该节点的左右节点的值添加到添加到最终要返回的 list 中(遍历值);
- 将该节点的左右子节点(如果有的话)添加到下一层的字典中。

然后用包含了下一层节点的字典替代这一层的字典。

代码

```python
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution:
    def levelOrder(self, root: TreeNode) -> List[int]:
        nodeSet = {}
        nodeSet['0'] = root
        if not root:
            return []
        result = [root.val]
        while nodeSet:
            print('nodeSet:', nodeSet)
            nodeList, nodeSetNew = self.transLine(nodeSet)
            nodeSet = nodeSetNew
            result = result + nodeList
```

```
        return result

    def transLine(self,nodeSet):
        nodeSetNew = {}
        nodeList = []
        num = 0
        for key in nodeSet:
            node = nodeSet[key]
            if node.left:
                nodeList.append(node.left.val)
                nodeSetNew[str(num)] = node.left
                num = num+1
            if node.right:
                nodeList.append(node.right.val)
                nodeSetNew[str(num)] = node.right
                num = num+1
        return nodeList, nodeSetNew
```

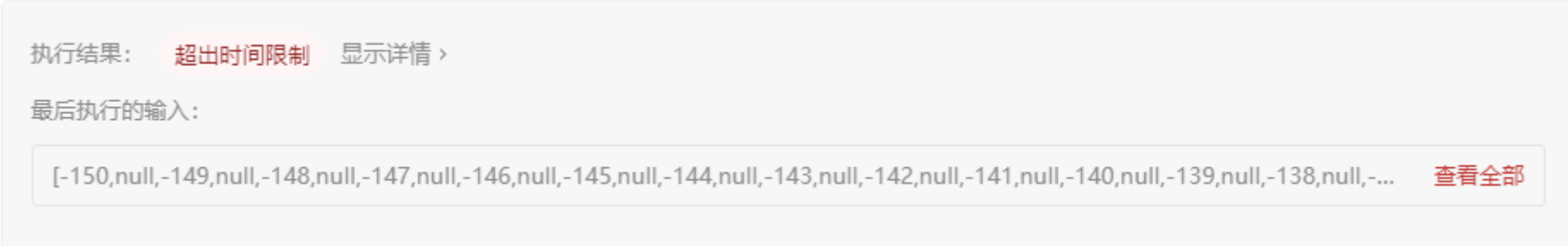
复杂度分析

对一个有 M 个节点的层，时间复杂度为 O(4N)

空间复杂度 O(2D)

结果

超出时间限制



思路 2： 辅助队列 BFS

这才是 BFS 的标准做法，利用队列先入先出的特性完成 BFS。

思路

将根节点放入队列 queue，
对队列中的第一个节点 queue[0]，将其左节点和右节点依次 append 到队列中；
将其值加入到最终要返回的列表中。
直到队列为空，程序结束。

代码

```python
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution:
    def levelOrder(self, root: TreeNode) -> List[int]:
        if not root:
            return []
        queue = []
        res = []
        queue.append(root)
        while queue:
            # for i in queue:
            node = queue[0]
            queue = queue[1:]
            res.append(node.val)
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)

        if not queue:
```

```
            return res
```

复杂度分析

虽然看上去循环次数和第一种方法相同，但从 list 中取第一个元素出来的时间复杂度和从列表中找到对应 key 的 value 的复杂度不同，因此节约了时间。

时间复杂度 O(N)

空间复杂度 O(N)

结果

执行结果： 通过 显示详情 >

执行用时： 40 ms ，在所有 Python3 提交中击败了 81.84% 的用户

内存消耗： 13.8 MB ，在所有 Python3 提交中击败了 87.27% 的用户

leetcode Python 广度优先遍历打印二叉树

Given a binary tree, return the *level order* traversal of its nodes' values. (ie, from left to right, level by level).

Example

Given binary tree `{3,9,20,#,#,15,7}`,

```
    3
   / \
  9  20
    /  \
   15   7
```

return its level order traversal as:

```
[
  [3],
  [9,20],
  [15,7]
]
```

```
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None


class Solution:
    def levelOrder(self, root):
        """
        :type root: TreeNode
        :rtype: List[List[int]]
        """
        res=[]
        if root is None:
            return res
        q=[root]
        while len(q)!=0:
            res.append([node.val for node in q])
            new_q=[]
            for node in q:
                if node.left:
                    new_q.append(node.left)
                if node.right:
                    new_q.append(node.right)
            q=new_q
        return res
```

二面：

1.自我介绍。

2.介绍“工大小美”项目相关。

3.浏览器中输入域名后发生了什么，整个流程。

4.cookie 能够跨域吗？flask 中的 cookie 怎么实现？

cookie 不能自己跨域，可以手动设置跨域。

flask cookie 的跨域请求

就一个地方需要设置:就是请求源.

1. 独立跨域(单独的某个响应实现跨域)

```
@app.route("/user", methods=["get", "post", "OPTIONS","PUT","DELETE"])
def user():
    data = {
        "username": "aa",
    }
    res = make_response(data)
    res.headers['Access-Control-Allow-Origin'] = 'http://localhost:8080'  # 关键就是这一句.
    res.headers["Access-Control-Allow-Headers"] = "Content-Type"
    res.headers['Access-Control-Allow-Credentials'] = 'true'  # 有这个,可以 cookie 跨域
    res.headers['Access-Control-Allow-Methods'] = "GET,POST,PUT,DELETE,OPTIONS" # 对于复杂请求必须加
    return res
```

那么 `res.headers["Access-Control-Allow-Headers"] = "Content-Type"`是什么呢?

没有的话,浏览器就会报错:

`Request header field content-type is not allowed by Access-Control-Allow-Headers in preflight response 预检响应的 Access-Control-Allow-Headers 不允许请求头字段 content-type`

需要注意,“http://localhost:8080"后面不能带有”/"

网上大部分文章是: `res.headers['Access-Control-Allow-Origin'] = "*"`指的是允许所有域注意星号不要写错

请求有两种,一种是简单请求,比如 get,post 还有复杂请求,比如 put

简单请求跨域,没有要求声明方法,但是复杂请求跨域必须声明允许方法

`res.headers['Access-Control-Allow-Methods'] = "GET,POST,PUT,DELETE,OPTIONS"`

前端也一定要有对应的方法,不然也还是会提示跨域访问被拒绝.比如

`@app.route("/user", methods=["get", "post", "OPTIONS","PUT","DELETE"])`

2.全局跨域

全局跨域,就在@app.after_request 在个装饰器下添加个全局的函数

after_request 虽然叫请求后,实际上应该叫 before_response 响应前,是在返回响应之前处理响应

```
@app.after_request
def after_request(res):
    resp = make_response(res)
   res.headers['Access-Control-Allow-Origin'] = 'http://localhost:8080'
    res.headers["Access-Control-Allow-Headers"] = "Content-Type"
    res.headers['Access-Control-Allow-Credentials'] = 'true'  //有这个,可以 cookie 跨域
    res.headers['Access-Control-Allow-Methods'] = "GET,POST,PUT,DELETE,OPTIONS" //对于复杂请求必须加
    return resp

@app.route('/user', methods=["POST", "OPTIONS"])
def hello_world():
    data = {
        "user": "user",
        "password": "000000"
    }
    return data
```

3 cookie 和 session 跨域

这里的 session 是指利用 cookie 存储的 session

后端(服务端)参考 2 全局跨域

客户端(前端) 需要注意的是,添加 withCredentials = true

比如 vue 中使用 axios:

```
import axios from 'axios'
Vue.prototype.$axios = axios
axios.defaults.withCredentials = true
```

对于 cookie 跨域,'Access-Control-Allow-Origin'的域名不能为"*"!

5.flask 框架路由实现原理是什么。

Flask 路由原理

一、添加 url 方式

1. @app.route('xxx')
2. app.add_url_rule('xxx', view_func=demo)

二、原理

1. url 和视图函数通过 endpoint 关联起来。url<---->endpoint<---->view_func;
2. url 和 endpoint 的对应关系存入到 url_map 中;
3. 以 endpoint 为 key，view_function 为值存入 view_functions 字典中。
4. 添加 url 的两种方法都可以指定 endpoint，如果 endpoint 未指定，会把视图函数的名称做为默认的 endpoint 的值。

6.项目中为何要使用 uwsgi。

uWSGI、uwsgi、WSGI、之间的关系，为什么要用 nginx 加 uWSGI 部署。

WSGI 协议

WSGI：是一种协议规范，起到规范参数的作用，就像告诉公路一样，规定超车靠右行，速度不低于 90km/h，等。但这一切都是对双方进行沟通，比如，重庆到武汉这条高速路，这儿重庆和武汉就各为一端，他们之间的行车规范就按照 WSGI 规则即可。我们现在需要记住，WSGI 沟通的双方是 wsgi server (比如 uWSGI） 要和 wsgi application（比如 django ）

wsgi server (比如 uWSGI）实现 wsgi 协议规范的服务器我们叫做 wsgi 服务器，也就是 uWSGI 服务器，
wsgi application（比如 django ）实现 wsgi 协议的应用，我们叫做 wsgi 应用，比如 Django，Falsk

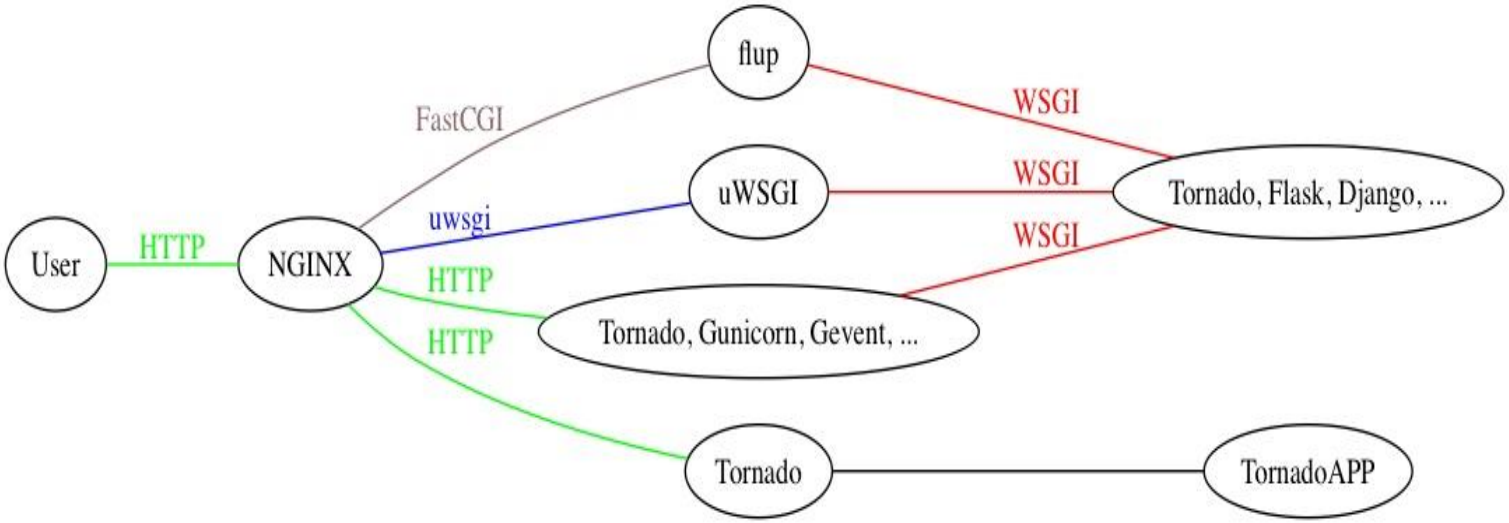
uWSGI

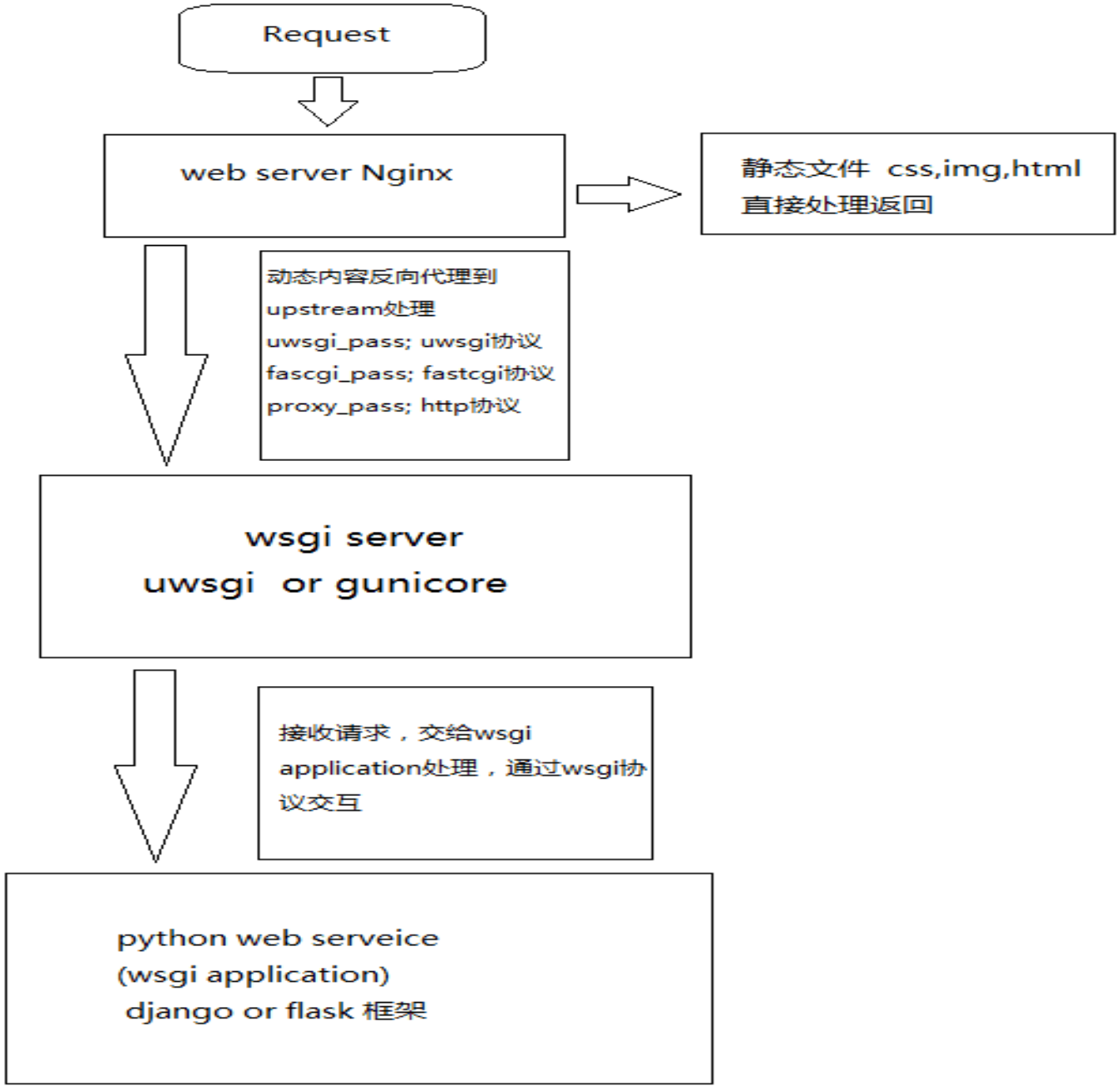
uWGSI：是一个 web 服务器，或者 wsgi server 服务器，他的任务就是接受用户请求，由于用户请求是通过网络发过来的，其中用户到服务器端之间用的是 http 协议，所以我们 uWSGI 要想接受并且正确解出相关信息，我们就需要 uWSGI 实现 http 协议，没错，uWSGI 里面就实现了 http 协议。所以现在我们 uWSGI 能准确接受到用户请求，并且读出信息。现在我们的 uWSGI 服务器需要把信息发给 Django，我们就需要用到 WSGI 协议，刚好 uWSGI 实现了 WSGI 协议，所以。uWSGI 把接收到的信息作一次简单封装传递给 Django，Django 接收到信息后，再经过一层层的中间件，于是，对信息作进一步处理，最后匹配 url，传递给相应的视图函数，视图函数做逻辑处理.....后面的就不叙述了，然后将处理后的数据通过中间件一层层返回，到达 Djagno 最外层，然后，通过 WSGI 协议将返回数据返回给 uWSGI 服务器，uWSGI 服务器通过 http 协议将数据传递给用户。这就是整个流程。
这个过程中我们似乎没有用到 uwsgi 协议，但是他也是 uWSGI 实现的一种协议，鲁迅说过，存在即合理，所以说，他肯定在某个地方用到了。我们过一会再来讨论
我们可以用这条命令：python manage.py runserver，启动 Django 自带的服务器，具体叫什么名字，我真不知道（知道的可以留言）。DJango 自带的服务器（runserver 起来的 HTTPServer 就是 Python 自带的 simple_server）。是默认是单进程单多线程的，对于同一个 http 请求，总是先执行一个，其他等待，一个

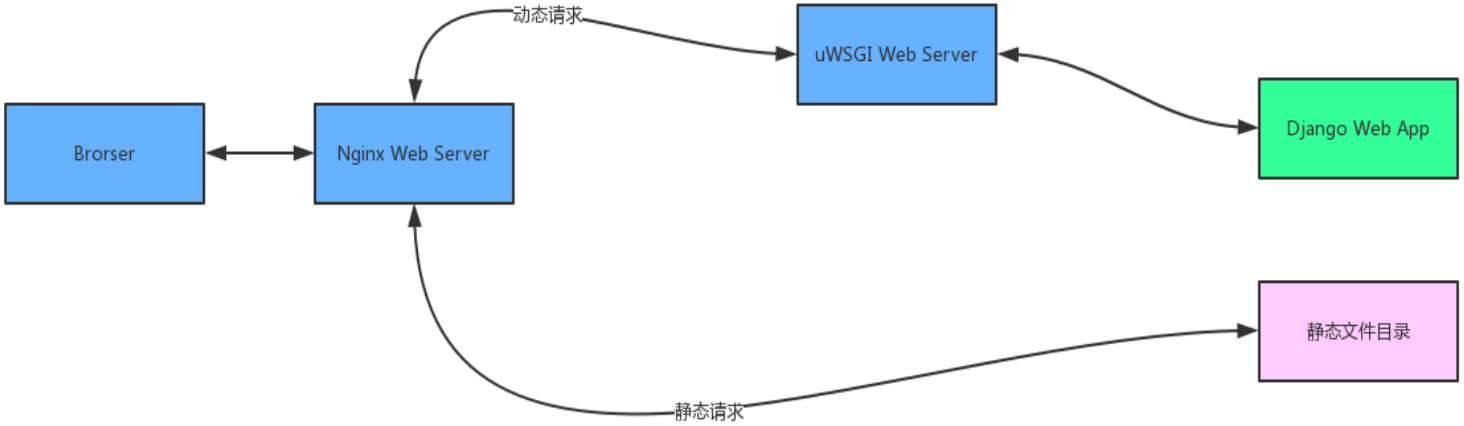
一个串行执行。无法并行。而且 django 自带的 web 服务器性能也不好，只能在开发过程中使用。于是我们就用 uWSGI 代替了。但是 uWSGI 也不够好，为什么看下图。

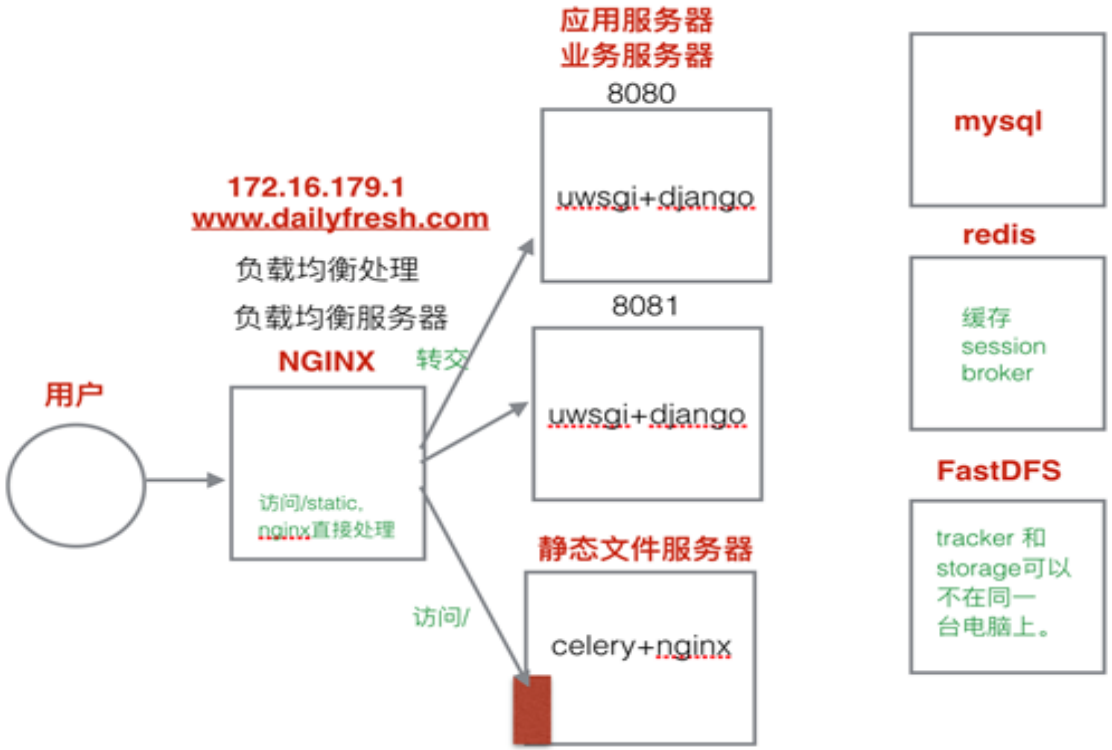
为什么有了 uWSGI 为什么还需要 nginx？

因为 nginx 具备优秀的静态内容处理能力，然后将动态内容转发给 uWSGI 服务器，这样可以达到很好的客户端响应。支持的并发量更高，方便管理多进程，发挥多核的优势，提升性能。这时候 nginx 和 uWSGI 之间的沟通就要用到 uwsgi 协议。




Request
web server Nginx
静态文件 css,img,html
直接处理返回
动态内容反向代理到
upstream处理
uwsgi_pass; uwsgi协议
fascgi_pass; fastcgi协议
proxy_pass; http协议
wsgi server
uwsgi or gunicore
接收请求，交给wsgi
application处理，通过wsgi协
议交互
python web serveice
(wsgi application)
django or flask 框架





杂谈

django 的并发能力真的是令人担忧，这里就使用 nginx + uwsgi 提供高并发

nginx 的并发能力超高，单台并发能力过万（这个也不是绝对），在纯静态的 web 服务中更是突出其优越的地方，由于其底层使用 epoll 异步 IO 模型进行处理，使其深受欢迎

做过运维的应该都知道，

Python 需要使用 nginx + uWSGI 提供静态页面访问，和高并发

php 需要使用 nginx + fastcgi 提供高并发，

java 需要使用 nginx + tomcat 提供 web 服务

django 原生为单线程序，当第一个请求没有完成时，第二个请求辉阻塞，知道第一个请求完成，第二个请求才会执行。

Django 就没有用异步，通过线程来实现并发，这也是 WSGI 普遍的做法，跟 tornado 不是一个概念

官方文档解释 django 自带的 server 默认是多线程
django 开两个接口,第一个接口 sleep(20),另一个接口不做延时处理(大概耗时几毫秒)
先请求第一个接口,紧接着请求第二个接口,第二个接口返回数据,第一个接口 20 秒之后返回数据
证明 django 的 server 是默认多线程
启动 uWSGI 服务器
在 django 项目目录下 Demo 工程名
uwsgi --http 0.0.0.0:8000 --file Demo/wsgi.py

经过上述的步骤测试,发现在这种情况下启动 django 项目,uWSGI 也是单线程,访问接口需要"排队"
不给 uWSGI 加进程,uWSGI 默认是单进程单线程
uwsgi --http 0.0.0.0:8000 --file Demo/wsgi.py --processes 4 --threads 2
processes: 进程数 # processes 和 workers 一样的效果 # threads : 每个进程开的线程数
经过测试,接口可以"同时"访问,uWSGI 提供多线程

- Python 因为 GIL 的存在,在一个进程中,只允许一个线程工作,导致单进程多线程无法利用多核
- 多进程的线程之间不存在抢 GIL 的情况,每个进程有一个自己的线程锁,多进程多 GIL

7.一个 NxM 的矩阵，每一个格子里面能够放一个字符，按照特定的规则，给定一个字符串，找给矩阵中有没有该字符串。

8.一个栈，在 O(1)的时间负责度，找出其中最小的数。

如何在 O(1)的时间复杂度求栈中的最小数据元素：
我们知道栈里面的数据我们只能访问栈顶端的数据，这样时间复杂度就为 O(N)。如何用 O(1)的事件复杂度求出栈中的最小元素，在算法中经常会采用用空间来换取时间的方式来提高时间的复杂度。也就是用两个栈，一个存放数据，一个存放栈的最小元素。
题目：用 O(1)的时间复杂度求栈中最小元素
思路：要找到栈里的最小元素，最简单的方法是对栈进行遍历，找出最小值，但是这样的方法的时间复杂度为 O(n)。在算法设计中，我们可以采取用空间换时间的方法来提高算法的时间复杂度，也就是采用额外的存储空间来降低操作的时间复杂度。具体实现就是使用两个栈结构，一个栈用来存储数据，另一个栈用来存储栈的最小元素，如果当前入栈的元素比原来栈的最小值还小，则把这个栈压入最小元素的栈中在出栈时，如果当前出栈的元素刚好是最小元素，保存最小值的栈顶元素也出栈，使得当前最小值变成它入栈之前的最小值
要求：
由于栈具有后进先出（Last In First Out, LIFO）的特点，因此 push 和 pop 只需要对栈顶元素进行操作，只能访问到栈顶元素，而无法得到栈中最小的元素。那么，如何求栈中最小元素呢？
解析：
1.利用一个变量记录栈底的位置，通过遍历栈中的所有元素找出最小值。但是这种方法的时间复杂度为 O(n)。
2.使用两个栈结构，一个栈用来存储数据，另一个栈用来存储栈的最小元素

实现思路如下：

(1)使用 elemStack 和 minStack 两个栈结构，elemStack 用来存储数据，minStack 用来存储 elem 栈的最小元素。

(2)在入栈时，如果当前入栈的元素比原来栈中的最小值还小，则把这个值压入 minStack 栈中；

(3)在出栈时，如果当前出栈的元素恰好为 minStack 栈中的最小值，则 minStack 栈的栈顶元素也出栈，使得当前最小值变为当前最小值入栈之前的那个最小值。

假设 elem 栈中元素为 3,2,5,1,6 那么 min 栈中对应的元素为 3,2,1。对应的图示如下：

如上图所示， elem 栈顶元素 6 出栈时，由于当前栈的最小值是 1，所以 min 栈元素不动。elem 栈顶元素变为 1，当 1 出栈时，由于 1 是当前栈的最小元素，所以 min 栈的栈顶元素 1 也要出栈。 以此类推。

```
# -*- coding:utf-8 -*-
#模拟栈
class Stack():
    def __init__(self):
        self.items = []
    #判断栈是否为空
    def empty(self):
        return len(self.items) == 0
    #返回栈大小
```

```
    def size(self):
        return len(self.items)
    #返回栈顶元素
    def peek(self):
        if not self.empty():
            return self.items[len(self.items)-1]
        else:
            return None
    #出栈
    def pop(self):
        if len(self.items)>0:
            return self.items.pop()
        else:
            print("栈已经为空！")
            return None
    #入栈
    def push(self,item):
        self.items.append(item)

class MySatck():
    def __init__(self):
        self.elemStack = Stack() #用来存储栈中元素
        self.minStack = Stack() #栈顶永远存储当前elemStack中最小值
    def push(self,data):
        self.elemStack.push(data)
        #更新保存最小元素的栈
        if self.minStack.empty():
            self.minStack.push(data)
        else:
            if data < self.minStack.peek():
                self.minStack.push(data)
    def pop(self):
        topData = self.elemStack.peek()
        self.elemStack.pop()
        if topData == self.mins():
```

```
            self.minStack.pop()
        return topData
    def mins(self):
        if self.minStack.empty():
            return 2 ** 32
        else:
            return self.minStack.peek()

if __name__ == "__main__":
    stack = MySatck()
    stack.push(3)
    print("栈中最小值为："+str(stack.mins()))
    stack.push(2)
    print("栈中最小值为：" + str(stack.mins()))
    stack.push(5)
    print("栈中最小值为：" + str(stack.mins()))
    stack.push(1)
    print("栈中最小值为：" + str(stack.mins()))
    stack.push(6)
    print("栈中最小值为：" + str(stack.mins()))
    stack.pop()
    print("栈中最小值为：" + str(stack.mins()))
    stack.pop()
    print("栈中最小值为：" + str(stack.mins()))
```

运行效果：

```
栈中最小值为：3
栈中最小值为：2
栈中最小值为：2
栈中最小值为：1
栈中最小值为：1
栈中最小值为：1
栈中最小值为：2
```

性能分析：

时间复杂度为 O(1)，额外申请了一个栈空间保存栈中最小元素，空间复杂度为 O(N)

8.12 python 字节跳动笔试后笔记

APP 内打开 3 25 2 分享

1.球场分块用 DFS 搜索

复制代码

```
import sys
try:
        while True:
                m,n=list(map(int, raw_input().split(',')))
                mat=[]
                for i in range(m):
                        mat.append(map(int, raw_input().split(',')))
                def search(i,j):
                        if i<0 or j<0 or i>=m or j>=n or mat[i][j]!=1:
                                return 0
                        temp=mat[i][j]
                        mat[i][j]=-1
                        manCount=temp+search(i-1,j)+search(i+1,j)+search(i,j-1)\
                                                +search(i,j+1)+search(i-1,j-1)+search(i+1,j+1)\
                                                +search(i-1,j+1)+search(i+1,j-1)
                        return manCount
                blockNum=0
                maxNum=0
                for i in range(m):
                        for j in range(n):
                                manNum=search(i,j)
                                if manNum>0:
                                        blockNum+=1
                                        if manNum>maxNum:
```

```
25                                                  maxNum=manNum
26                  print str(blockNum)+','+str(maxNum)
27except Exception:
28          pass
```

2.重叠范围数组合并

复制代码

```
1 try:
2           while True:
3                   m=int(raw_input())
4                   mat=[]
5                   for i in range(m):
6                           s=raw_input().split(';')
7                           for item in s:
8                                   t=item.split(',')
9                                   mat.append(map(int, t))
10                  mat.sort()
11                  ind=1
12                  res=[mat[0]]
13                  while ind<len(mat)
14                        if mat[ind][0]<=res[-1][1]:
15                                if mat[ind][1]>=res[-1][1]:
16                                          x=res[-1][0]
17                                          y=mat[ind][1]
18                                          res.pop(-1)
19                                          res.append([x, y])
20                          else:
21                                  res.append(mat[ind])
22                          ind+=1
23                  for ind in range(len(res)):
24                          bl=(ind!=len(res)-1)
25                          sys.stdout.write(str(res[ind][0])+','+str(res[ind][1])+';'*bl)
26except Exception:
```

27 pass

3. 题听说是背包问题，取牌问题。二维数组，一维是个人加分项，一维是集体积分项。甲乙两人从中选取当个人加分项相等时，集体加分最大值。

4. 题穷举 a 数组窗口内部最大值<b 数组窗口最小值，时间复杂度过高 30%