OpenAI面经题（4月整理）

OpenAI（ML-SWE） 面试题

1. 实现Linux系统的"cd"command，例如输入: current directory, relative destination, 输出：最终的 path。
    a. 增加 ~ 的支持
    b. 增加 symbolic link（类似于 ~ 的符号映射） 支持

> 实现Linux系统的"cd"command，类似题 https://www.lintcode.com/problem/421/
>
> 第二问：加对 ~ 符号的支持，也就是 user home directory
>
> 第三问：加 symbolic link 支持。Given there is a map of symbolic link path and there respective real path.

OpenAI（HPC 组） 面试题

1. Web 爬虫优化（DFS / BFS 优化）

给 web crawler 一个 URL 会返回和这个页面的相连所有的所有链接的列表。找出给定链接为起始点所有相关的网址。

> 1. **使用 DFS**
>
> 使用DFS（深度优先搜索）来优化网络爬虫**可以避免大量的重复访问**。与BFS不同，DFS优先遍历深度较大的节点，因此在处理某个页面的链接时，它会首先访问该页面的所有子链接，而不是访问该页面的兄弟链接。这意味着在大多数情况下，DFS会更快地发现新的页面。
>
> 下面是使用DFS优化的示例代码：

```python
import requests
from bs4 import BeautifulSoup
import os

def dfs_crawl(url, path, visited=None):
    """
    使用DFS遍历网站，并将所有页面保存到本地
    """
    if visited is None:
        visited = set()
```

```
    if url in visited:
        return
    try:
        response = requests.get(url)
        soup = BeautifulSoup(response.content, 'html.parser')
        filename = url.split('/')[-1]
        if filename == '':
            filename = 'index.html'
        with open(os.path.join(path, filename), 'w', encoding='utf-8') as f
            f.write(str(soup))
        print('Saved:', url)
        visited.add(url)
        for link in soup.find_all('a'):
            href = link.get('href')
            if href is not None:
                if href.startswith('http'):
                    dfs_crawl(href, path, visited)
                elif href.startswith('/'):
                    dfs_crawl(url + href, path, visited)
                else:
                    dfs_crawl(url + '/' + href, path, visited)
    except:
        pass

if __name__ == '__main__':
    url = 'https://example.com'
    path = 'example'
    dfs_crawl(url, path)

```

在这个版本中，我们使用了一个名为 `visited` 的集合来避免访问相同的链接。在函数开始时，我们检查 `visited` 集合中是否存在当前链接，如果存在，则直接返回，否则将该链接添加到 `visited` 集合中。然后，我们继续访问该链接的所有子链接，这是通过递归调用 `dfs_crawl` 函数来完成的。

请注意，使用 DFS 来优化网络爬虫**可能会导致某些页面被忽略，因为 DFS 不一定能够访问到所有的页面**。

2. 使用BFS

BFS（广度优先搜索）是另一种优化网络爬虫的常用算法。与DFS不同，BFS **优先访问离当前页面较近的页面**，这样可以更快地找到整个网站的所有页面。

下面是使用BFS优化的示例代码：

```
```

```python
import requests
from bs4 import BeautifulSoup
import os

def bfs_crawl(url, path):
    """
    使用BFS遍历网站，并将所有页面保存到本地
    """
    visited = set()
    queue = [url]
    while len(queue) > 0:
        curr_url = queue.pop(0)
        if curr_url in visited:
            continue
        try:
            response = requests.get(curr_url)
            soup = BeautifulSoup(response.content, 'html.parser')
            filename = curr_url.split('/')[-1]
            if filename == '':
                filename = 'index.html'
            with open(os.path.join(path, filename), 'w', encoding='utf-8')
                f.write(str(soup))
            print('Saved:', curr_url)
            visited.add(curr_url)
            for link in soup.find_all('a'):
                href = link.get('href')
                if href is not None:
                    if href.startswith('http'):
                        queue.append(href)
                    elif href.startswith('/'):
                        queue.append(url + href)
                    else:
                        queue.append(url + '/' + href)
        except:
            pass

if __name__ == '__main__':
    url = 'https://example.com'
    path = 'example'
    bfs_crawl(url, path)

```

在这个版本中，我们使用了一个名为 `queue` 的列表来存储要访问的链接。在函数开始时，我们将起始链接添加到 `queue` 中，并开始循环。在每次迭代中，我们从 `queue` 的头部取出一个链接，并检查它是否已经访问过。如果已经访问过，我们跳过该链接并继续处理下一个链接。否

则，我们将该链接添加到 `visited` 集合中，并访问它的所有子链接。然后，我们将这些子链接添加到 `queue` 的尾部，以便在下一次迭代中处理。

请注意，BFS **需要更多的存储空间**来存储 `queue` 列表和 `visited` 集合，因此**在处理大型网站时，内存使用可能会很高**。此外，BFS **也需要更多的网络请求，因为它需要访问更多的页面**。

2. 多线程解决 IO 阻塞问题

用 Python 多线程解决 IO 阻塞问题

1. **使用非阻塞式 I/O**：在 Python 中，可以使用 select、poll 或者 epoll 等模块来实现非阻塞式 I/O。这些模块可以让程序在等待 I/O 操作完成的同时继续执行其他任务，提高程序的并发度和性能。
2. **使用多线程或协程**：可以使用多线程或协程来实现并发式 I/O，以提高程序的性能。例如，在多线程或协程中，可以让某个线程或协程负责 I/O 操作，当 I/O 操作完成后再将数据返回给主线程或主协程。
3. **使用异步 I/O**：在 Python 3.4 以后的版本中，引入了 asyncio 模块，可以使用协程实现异步 I/O。使用异步 I/O 可以让程序在等待 I/O 操作完成的同时继续执行其他任务，提高程序的并发度和性能。
4. **使用缓存**：可以使用缓存技术来减少 I/O 操作的次数，从而提高程序的性能。例如，在读取大量文件时，可以使用文件缓存，将文件读取到内存中进行处理，减少磁盘 I/O 操作的次数。

OpenAI（MLE） 面试题

1. 文件操作问题

找重复的文件 文件内容长怎么办 文件太多怎么办 运行慢怎么办

如何查找重复的文件

a. **比较文件内容**：可以使用程序读取文件内容，比较两个或多个文件的内容是否相同。如果内容相同，则可以判断它们是重复的文件。

b. **计算文件的哈希值**：可以使用哈希函数对文件进行计算，得到一个唯一的哈希值。如果两个文件的哈希值相同，则它们可能是重复的文件。

c. **比较文件大小和文件名**：可以通过比较文件大小和文件名，找到可能相同的文件。然后再通过其他方式进一步验证是否是重复的文件。

以上方法可以单独使用，也可以组合使用，以找到重复的文件。

文件内容过长

如果文件的内容太长，在比较过程中会导致内存占用过高，程序运行缓慢，甚至会出现程序崩溃的情况。为了避免这种情况，可以采取以下方法：

a. **分块比较**：将文件按照指定大小分成若干块，逐块进行比较。这样可以避免一次性读取整个文件内容，减少内存占用。
b. **使用哈希值比较**：对于大文件，可以先计算文件的哈希值，然后比较哈希值是否相同。如果哈希值相同，则可以进一步比较文件内容，否则可以判断它们不是重复的文件。
c. **采样比较**：可以随机抽取文件的一部分内容进行比较。如果采样内容相同，则可以进一步比较文件内容，否则可以判断它们不是重复的文件。

以上方法都是针对大文件进行优化的方法。如果文件大小较小，直接比较文件内容也不会有明显的性能问题。同时，比较文件时需要注意文件的安全性，避免对文件进行意外修改或删除。

文件数量过多

a. **并行处理**：可以将文件分成若干批次，分别处理。可以采用多线程或分布式计算的方式进行并行处理，提高处理速度。
b. **增量处理**：可以记录已经处理过的文件，每次只处理新增加的文件。这样可以避免重复处理已经处理过的文件，节省处理时间。
c. **分类处理**：可以根据文件类型、文件大小、文件名等特征进行分类处理，针对不同的类型采取不同的处理方式。例如，可以针对图片、音频、视频等文件类型，使用不同的处理方法来寻找重复文件。
d. **预处理优化**：可以通过索引、缓存、压缩等方式来优化处理过程。例如，可以将文件名、路径等信息存储在数据库中，加快文件查找速度；可以对文件进行压缩处理，减少文件大小，提高处理速度。

运行慢

a. **优化算法**：可以对算法进行优化，使得程序运行更加高效。例如，采用更优的搜索算法、排序算法、哈希算法等，减少程序的时间复杂度和空间复杂度，提高程序的运行效率。
b. **减少 I/O 操作**：可以减少磁盘读写操作，降低程序的 I/O 开销。例如，可以采用缓存、索引等技术来减少磁盘读写操作，提高程序的运行效率。
c. **多线程处理**：可以采用多线程的方式来提高程序的并发度，提高程序的运行效率。例如，可以将任务分成若干部分，分别由不同的线程处理，提高程序的并发度和处理速度。

2. 基本的 probability, bias-variance 等问题。

- free text 解释 kl-divergence 的方程

> KL-Divergence（KL散度）是用于度量两个概率分布之间差异的指标。假设有两个概率分布 P 和 Q，KL散度的计算公式如下：
>
> KL(P||Q) = Σ P(x) * log(P(x) / Q(x))
>
> 其中，x 是所有可能的事件，P(x) 和 Q(x) 分别表示事件 x 在分布 P 和 Q 中的概率。log 是自然对数函数。
>
> KL散度的含义是：将概率分布 P 转化为概率分布 Q 所需的额外信息量。可以理解为，如果我们用概率分布 Q 来拟合概率分布 P，那么在用 Q 来表示 P 时需要额外增加多少信息。
>
> 需要注意的是，KL散度并不是对称的，即 KL(P||Q) ≠ KL(Q||P)。这是因为KL散度对于概率分布 P 和 Q 的位置是敏感的，换句话说，它对于哪个分布用来拟合另一个分布更适合是敏感的。

- free text 解释 crossentropy for classification of n class. what is the worst loss?

> Cross-entropy is a widely used loss function for classification problems with n classes. In this setting, the goal is to predict which of the n classes a given input belongs to. The cross-entropy loss measures the dissimilarity between the true class label and the predicted class probabilities.
>
> The cross-entropy loss for classification of n class can be expressed as:
>
> Cross-entropy = - Σ y_i log(p_i)
>
> Here, y_i is the true probability distribution, which is a one-hot encoded vector indicating the true class of the input, and p_i is the predicted probability distribution, which is a vector containing the predicted probabilities for each class.
>
> The worst loss function in this context would be one that fails to distinguish between the different classes or assigns the same probability to all the classes. In this case, the model would be unable to learn the underlying patterns that differentiate between the different classes and would not be able to make accurate predictions.
>
> Moreover, a loss function that does not properly account for class imbalance could result in a poorly performing model. This is because the model may be biased towards the majority class and not learn to distinguish between the minority classes, leading to poor performance on those classes.
>
> Therefore, it's crucial to choose an appropriate loss function that aligns with the goals of the model and the specific problem being solved. The cross-entropy loss function is a popular choice for classification problems with n classes and is often used in conjunction with other techniques such as regularization to help ensure that the model learns the right patterns and produces useful results.

还有 coding 题目：

- 给你 training, test data，让你写一个 classification model （需要 train 然后 output prediction）。他们会用 accuracy 评分

- largest sub-grid：给你一个2d array, 找出 maximum size of square sub-grid such that all sub-grids of this size must have sub-grid sum <= maxsum，思路可参考 https://www.lintcode.com/problem/943/ 但题目不完全一致

3. 守卫遍历的最少时间（算法题）

墙和门相似题 https://www.lintcode.com/problem/663/

> An isolated prison represented by a 2D array where
>
> - 'W' is wall
> - 'G' is prison guard
> - ' ' is cell
>
> Prison guards are allowed to move ajancently but not diagonally and they can move 1 unit each second
>
> Please return the shortest possible time needed for a guard to get to each cell
>
> Input:
>
> [[W, W, W, W, W],
>
> [W, ' ', 'G', ' ', W]
>
> [W, ' ', 'G', ' ', W]
>
> [W, ' ', ' ', ' ', W]
>
> [W, W, W, W, W]]
>
> Output:
>
> [[W, W, W, W, W],
>
> [W, '1', 'G', '1', W]
>
> [W, '1', 'G', '1', W]
>
> [W, '2', '1', '2 ', W]
>
> [W, W, W, W, W]]

OpenAI（ROBOT-SWE） 面试题

1. ID 发布器问题

实现一个ID发布器，不考虑多线程，只是单纯的讨论时间空间效率

> 简单示例：

```python
class IDGenerator:
    def __init__(self):
        self.counter = 0

    def generate_id(self):
        self.counter += 1
        return self.counter

```

该实现使用一个计数器来生成唯一的标识符。每次调用 `generate_id()` 方法时，计数器会加一，然后返回新的值。这种实现简单易懂，且时间和空间效率都很高，因为它的时间复杂度为O(1)，空间复杂度也为O(1)。

然而，这种实现有一个明显的缺点，**它是不可重用的**。当应用程序关闭后，计数器的值也会被重置。如果需要在应用程序重启后继续生成唯一的标识符，我们需要将计数器的值保存在一个持久化存储中，例如数据库或文件系统中。

因此，为了实现一个可重用的ID发布器，我们需要修改上述实现，将计数器的值保存在持久化存储中。以下是一个简单的修改示例：

```python
import pickle

class IDGenerator:
    def __init__(self):
        try:
            with open('counter.pkl', 'rb') as f:
                self.counter = pickle.load(f)
        except FileNotFoundError:
            self.counter = 0

    def generate_id(self):
        self.counter += 1
        with open('counter.pkl', 'wb') as f:
            pickle.dump(self.counter, f)
        return self.counter

```

该实现使用 `pickle` 模块将计数器的值保存在一个二进制文件中。每次调用 `generate_id()` 方法时，计数器会加一，然后将其值保存在文件中。这种实现的时间和空间效率仍然很高，但它需要进行文件读写操作，因此其性能受到了磁盘I/O速度的影响。

2. 分配和释放 ID

要实现的接口是alloc和release，比如最大ID限制在1百万，用户可以不停的调用int alloc()和void release(id)。ID使用完之后alloc可以return -1之类的，不太重要release要保护同ID已经被release过，防止user error

> 可以使用一个哈希表来跟踪已经分配的 ID 和已经释放的 ID：

```python
class IDAllocator:
    def __init__(self, max_id=1000000):
        self.max_id = max_id
        self.allocated_ids = set()
        self.released_ids = set()

    def alloc(self):
        if self.released_ids:
            # 如果存在已经被释放的 ID，就使用最小的那个 ID。
            id = self.released_ids.pop()
            self.allocated_ids.add(id)
            return id
        elif len(self.allocated_ids) < self.max_id:
            # 如果已经分配的 ID 数量还没有达到最大值，就分配一个新的 ID。
            id = len(self.allocated_ids) + 1
            self.allocated_ids.add(id)
            return id
        else:
            # 如果已经分配的 ID 数量已经达到最大值，就返回 -1。
            return -1

    def release(self, id):
        if id in self.allocated_ids:
            # 如果 ID 已经被分配，就释放它。
            self.allocated_ids.remove(id)
            self.released_ids.add(id)
        elif id in self.released_ids:
            # 如果 ID 已经被释放，就报错。
            raise ValueError("ID has already been released.")
        else:
            # 如果 ID 既没有被分配也没有被释放，就报错。
            raise ValueError("Invalid ID.")

```

> 在上面的实现中，`alloc` 方法首先检查是否有已经被释放的 ID，如果有，就使用最小的那个 ID；如果没有，就检查已经分配的 ID 的数量是否已经达到最大值，如果没有，就分配一个新的 ID；否则，就返回 -1。

> `release` 方法首先检查传入的 ID 是否已经被分配，如果是，就将它从已经分配的 ID 集合中移除，并将它添加到已经释放的 ID 集合中；如果不是，就检查它是否已经被释放，如果是，就报错（ID 已经被释放）；否则，也报错（ID 既没有被分配也没有被释放）。

OpenAI（General）面试题

1. Web 爬虫优化（并行优化）

写一个web crawler. 给一个domain， crawler出所以这个domain下面的page。

> 下面的代码是一个基本的使用Python实现的 Web crawler：

```python
import requests
from bs4 import BeautifulSoup
import os

def get_links(url):
    """
    从URL中获取所有链接
    """
    links = []
    try:
        response = requests.get(url)
        soup = BeautifulSoup(response.content, 'html.parser')
        for link in soup.find_all('a'):
            href = link.get('href')
            if href is not None:
                if href.startswith('http'):
                    links.append(href)
                elif href.startswith('/'):
                    links.append(url + href)
                else:
                    links.append(url + '/' + href)
    except:
        pass
    return links

def crawl(url, path):
    """
    爬取网站，并将所有页面保存到本地
    """
    links = get_links(url)
    os.makedirs(path, exist_ok=True)
    for link in links:
        try:
```

```
            response = requests.get(link)
            soup = BeautifulSoup(response.content, 'html.parser')
            filename = link.split('/')[-1]
            if filename == '':
                filename = 'index.html'
            with open(os.path.join(path, filename), 'w', encoding='utf-8')
                f.write(str(soup))
            print('Saved:', link)
        except:
            pass

if __name__ == '__main__':
    url = 'https://example.com'
    path = 'example'
    crawl(url, path)

```

Follow up: optimize 一个并行版本

为了实现优化上述代码的并行版本，可以使用多线程或异步技术来同时处理多个链接：

```
import threading
import queue
import requests
from bs4 import BeautifulSoup
import os

class CrawlerThread(threading.Thread):
    """
    爬虫线程
    """
    def __init__(self, queue, path):
        threading.Thread.__init__(self)
        self.queue = queue
        self.path = path

    def run(self):
        while True:
            link = self.queue.get()
            if link is None:
                break
            try:
                response = requests.get(link)
                soup = BeautifulSoup(response.content, 'html.parser')
                filename = link.split('/')[-1]
                if filename == '':
```

```
                    filename = 'index.html'
                with open(os.path.join(self.path, filename), 'w', encoding=
                    f.write(str(soup))
                print('Saved:', link)
            except:
                pass
            self.queue.task_done()

def get_links(url):
    """
    从URL中获取所有链接
    """
    links = []
    try:
        response = requests.get(url)
        soup = BeautifulSoup(response.content, 'html.parser')
        for link in soup.find_all('a'):
            href = link.get('href')
            if href is not None:
                if href.startswith('http'):
                    links.append(href)
                elif href.startswith('/'):
                    links.append(url + href)
                else:
                    links.append(url + '/' + href)
    except:
        pass
    return links

def crawl(url, path, num_threads=4):
    """
    爬取网站，并将所有页面保存到本地
    """
    queue = queue.Queue()
    for link in get_links(url):
        queue.put(link)
    os.makedirs(path, exist_ok=True)
    threads = []
    for i in range(num_threads):
        thread = CrawlerThread(queue, path)
        thread.start()
        threads.append(thread)
    for i in range(num_threads):
        queue.put(None)
    for thread in threads:
        thread.join()

```

```
if __name__ == '__main__':
    url = 'https://example.com'
    path = 'example'
    crawl(url, path, num_threads=4)

```

在此并行版本中，我们使用一个队列来存储所有需要爬取的链接，然后创建多个线程来处理这些链接。每个线程都会从队列中获取一个链接并处理它，直到队列为空。当所有线程完成它们的工作时，程序将退出。

请注意，这里我们使用了Python的内置线程库，这是一个比较简单的实现。如果需要更高效的并发性能，可以考虑使用更强大的并发库，如 `asyncio` 或 `gevent`。