1. cd函数实现（简易文件路径解析器）

中文说明：
实现一个`cd(current_dir, new_dir)`函数，返回最终路径。例子：

- `cd(/foo/bar, baz)` → `/foo/bar/baz`
- `cd(/foo/../, ./baz)` → `/baz`
- `cd(/, foo/bar/../../baz)` → `/baz`
- `cd(/, ..)` → `Null`

拓展要求：

- 加上**symbolic link**（软链接）支持。
- 软链接Map有长短匹配，优先用更长的匹配。
- 如果存在软链接循环（比如A指向B，B又指向A），需要抛出异常检测（可以用DFS检测循环）。

进一步要求：

- 支持~（用户home目录）符号。
- 每次变化都需要重新apply软链接映射。
- 最终路径并不一定最短路径。
- 需要考虑所有corner cases。

提供参考代码（**Python**）：

python
CopyEdit

```
class Solution:
    def simplifyPath(self, path: str) -> str:
        nodes = path.split("/")
```

```
        nodes = [node for node in nodes if node != "" and node != "."]
        results = []
        for node in nodes:
            if node == "..":
                if results:
                    results.pop()
            elif node == "~":
                results.clear()
            else:
                results.append(node)
        return "/" + "/".join(results)

    def cd(self, pwd, input, dictionary=None) -> str:
        path = pwd + '/' + input
        nodes = path.split("/")
        nodes = [node for node in nodes if node != "" and node != "."]
        results = []
        for node in nodes:
            if node == "..":
                if results:
                    results.pop()
            elif node == "~":
                results.clear()
            else:
                results.append(node)
        result = "/" + "/".join(results)

        if dictionary:
            if self.is_cyclic(dictionary):
                raise Exception("Softlink cycle detected")
            keys = sorted(dictionary.keys(), key=lambda k: -len(k))
            while True:
                found = False
                for key in keys:
                    if key in result:
                        result = result.replace(key, dictionary[key])
                        found = True
                        break
```

```
                if not found:
                    break
        return result

    def is_cyclic(self, d):
        visited = {k: False for k in d}
        stack = {k: False for k in d}
        for key in d:
            if not visited[key] and self.is_cyclic_inner(d, key,
visited, stack):
                return True
        return False

    def is_cyclic_inner(self, d, key, visited, stack):
        if not visited[key]:
            stack[key] = True
            visited[key] = True
            if d[key] in d:
                if not visited[d[key]] and self.is_cyclic_inner(d,
d[key], visited, stack):
                    return True
                if stack[d[key]]:
                    return True
        stack[key] = False
        return False
```

测试示例：

python
CopyEdit

```
s = Solution()
assert s.cd("/123/", "..") == "/"
assert s.cd("/123/456", ".") == "/123/456"
assert s.cd("/123/456", "123/~") == "/"
assert s.cd("/123/456", "123/~/67") == "/67"
assert s.cd("/123/", "456", {"123": "abc", "123/456": "cde"}) ==
"/cde"
```

English Version:

Implement a `cd(current_dir, new_dir)` function that returns the final path. Examples:

- `cd(/foo/bar, baz)` → `/foo/bar/baz`
- `cd(/foo/../, ./baz)` → `/baz`
- `cd(/, foo/bar/../../baz)` → `/baz`
- `cd(/, ..)` → `Null`

Further requirements:

- Add **symbolic link** support, given a dictionary mapping.
- Prioritize longer matches for symbolic links.
- Detect cycles (e.g., A → B → A) using DFS and throw exceptions.
- Add ~ home directory symbol support.
- After each change, reapply the symbolic link map.
- The final path does not have to be the shortest one.

Reference Code (Python):
(Same as above.)

Test Cases:
(Same as above.)

---

2. Resumable Iterator (可恢复迭代器)

中文说明:

要求:

- 写一个抽象接口`IteratorInterface`
- 实现`get_state()`和`set_state(state)`，用于恢复迭代
- 不允许用`hasNext()`；需要能正确处理遍历结束的情况
- 写测试函数，测试每一个保存的state是否正确恢复

基本步骤：

1. 写接口
2. 写基于List的Iterator
3. 写多文件（多个Iterator）的可恢复Iterator
4. 加上Coroutine（async迭代器）

参考代码（**Python**）

python
CopyEdit

```
class IteratorInterface:
    def __init__(self):
        pass
    def __iter__(self):
        pass
    def __next__(self):
        pass
    def get_state(self):
        pass
    def set_state(self, state):
        pass

class ListIterator:
    class State:
        def __init__(self, ind):
            self.ind = ind

    def __init__(self, input_list):
```

```
        self.input_list = input_list
        self.state = self.State(0)

    def __iter__(self):
        return self

    def __next__(self):
        if self.state.ind < len(self.input_list):
            self.state.ind += 1
            return self.input_list[self.state.ind - 1]
        raise StopIteration

    def get_state(self):
        return self.state

    def set_state(self, state):
        self.state = state
```

测试示例：

python
CopyEdit

```
l = [4,3,2,1]
it = ListIterator(l)
assert next(it) == 4
state = it.get_state()
assert next(it) == 3
it.set_state(state)
assert next(it) == 3
```

English Version:

Requirements:

- Implement an abstract `IteratorInterface`
- Implement `get_state()` and `set_state(state)` for restoring iteration

- No `hasNext()` is allowed; caller should handle StopIteration
- Write tests that verify resuming works correctly

Steps:

1. Define the interface
2. Implement a List-based resumable iterator
3. Implement a multiple file resumable iterator
4. Upgrade to async (Coroutine) iterator

Reference Code (Python):
(Same as above.)

Test Cases:
(Same as above.)

3. Time Based Key-Value Store（带并发与OOM讨论）

中文说明：

实现一个支持时间戳版本的KV存储（类似LeetCode 981题）。要求：

- `set(key, value)`: 保存当前时间戳的键值对
- `get(key, timestamp)`: 返回 <= timestamp 的最新value

Follow-ups:

1. 多线程并发读写，如何处理？
   → 用`ReadWriteLock`读写锁机制，每个key独立锁。
2. 数据集太大OOM怎么办？
   → 把老版本移到磁盘，内存只保存最近访问频繁的数据（如：LRU策略或SSTable方式）

参考代码：

基础版本（带锁）：

python
CopyEdit

```python
import time
from collections import defaultdict
from threading import Lock

class TimeMap:
    def __init__(self):
        self.values = defaultdict(list)
        self.lock = Lock()

    def set(self, key: str, value: str) -> None:
        with self.lock:
            timestamp = time.time()
            self.values[key].append([value, timestamp])

    def get(self, key: str, timestamp: int) -> str:
        current = time.time()
        if timestamp > current:
            time.sleep(timestamp - current)
        with self.lock:
            if key not in self.values:
                return ""
            vs = self.values[key]
            l, r = 0, len(vs) - 1
            while l <= r:
                mid = (l + r) // 2
                if vs[mid][1] > timestamp:
                    r = mid - 1
                else:
                    l = mid + 1
            return vs[r][0] if r >= 0 else ""
```

优化版本（分shard）：

python
CopyEdit

```python
class ReadWriteLock:
    def __init__(self):
```

```
        self.read_ready = threading.Condition(threading.Lock())
        self.readers = 0

    def acquire_read(self):
        with self.read_ready:
            self.readers += 1

    def release_read(self):
        with self.read_ready:
            self.readers -= 1
            if self.readers == 0:
                self.read_ready.notify_all()

    def acquire_write(self):
        self.read_ready.acquire()
        while self.readers > 0:
            self.read_ready.wait()

    def release_write(self):
        self.read_ready.release()

class TimeMapShard:
    def __init__(self):
        self.values = defaultdict(list)
        self.lock = ReadWriteLock()

    def set(self, key, value):
        self.lock.acquire_write()
        try:
            timestamp = time.time()
            self.values[key].append([value, timestamp])
        finally:
            self.lock.release_write()

    def get(self, key, timestamp):
        current = time.time()
        if timestamp > current:
            time.sleep(timestamp - current)
```

```
        self.lock.acquire_read()
        try:
            if key not in self.values:
                return ""
            vs = self.values[key]
            l, r = 0, len(vs) - 1
            while l <= r:
                mid = (l + r) // 2
                if vs[mid][1] > timestamp:
                    r = mid - 1
                else:
                    l = mid + 1
            return vs[r][0] if r >= 0 else ""
        finally:
            self.lock.release_read()

class TimeMap:
    def __init__(self):
        self.size = 16
        self.maps = [TimeMapShard() for _ in range(self.size)]

    def get_shard(self, key):
        return self.maps[hash(key) % self.size]

    def set(self, key, val):
        self.get_shard(key).set(key, val)

    def get(self, key, timestamp):
        return self.get_shard(key).get(key, timestamp)
```

English Version:

Implement a time-based KV store (similar to LeetCode 981).

Requirements:

- `set(key, value)`: Save a value with the current timestamp.

- `get(key, timestamp)`: Return the most recent value at or before the given timestamp.

Follow-ups:

1. How to handle multithreaded access?
   → Use `ReadWriteLock` per key.

2. What to do if the dataset becomes too large (OOM)?
   → Keep only the most recently accessed data in memory; move older versions to disk.

Reference Code:
(Same as above.)

---

4. In-Memory Database (带Where+OrderBy)

中文说明：

设计一个简易版的内存数据库。要求：

- `insert(table, row)`

- `query(table, columns, where_conditions=[], order_by=[])`

支持功能：

- 单列where
- 多列AND where
- order by一列
- order by多列
- 支持大于小于查询(>, <)
- 无需**SQL**解析，自己定义API

- 所有数据类型都是String

参考代码（**Python**）：

python
CopyEdit

```
from collections import defaultdict

class DataBase:
    def __init__(self, cols):
        self.rows = []
        self.col_map = {v: i for i, v in enumerate(cols)}

    def insert_row(self, row):
        self.rows.append(row)

    def where(self, conds):
        res = []
        for r in self.rows:
            if all(r[self.col_map[k]] == v for k, v in conds.items()):
                res.append(r)
        return res

    def order_by(self, cols):
        inds = [self.col_map[col] for col in cols]
        return sorted(self.rows, key=lambda r: [r[i] for i in inds])

    def greater_than(self, conds):
        res = []
        for r in self.rows:
            if all(r[self.col_map[k]] > v for k, v in conds.items()):
                res.append(r)
        return res
```

测试示例：

python
CopyEdit

```
db = DataBase(["name", "birthday"])
```

```
db.insert_row(["Ada", "1815-12-10"])
db.insert_row(["Charles", "1791-12-26"])
db.insert_row(["Ben", "1715-12-11"])

assert db.where({"name": "Ada"}) == [["Ada", "1815-12-10"]]
assert db.order_by(["birthday"]) == [["Ben", "1715-12-11"],
["Charles", "1791-12-26"], ["Ada", "1815-12-10"]]
```

English Version:

Design a simple **in-memory database**.

Requirements:

- `insert(table, row)`
- `query(table, columns, where_conditions=[], order_by=[])`

Supported features:

- Where by single or multiple columns
- AND condition
- ORDER BY one or more columns
- Greater-than and less-than conditions
- **No SQL parsing**; define your own APIs
- All fields are treated as String

Reference Code:
(Same as above.)

5. Spreadsheet API（设计可依赖单元格的表格计算）

中文说明：

要求实现一个表格系统，单元格可以是：

- 直接的数值
- 两个其他单元格的和 (比如 `C = A + B`)

功能要求：

- `setCell(key, cell)`
- `getCellValue(key)`

特点：

- 每次`setCell`需要更新依赖关系
- 如果有循环引用(A依赖B，B又依赖A)，需要抛异常
- 需要支持高效读取，使用cache缓存计算结果
- 修改单元格时，自动**invalidate**相关依赖的单元格

参考代码(**Python**版)

python
CopyEdit

```
from typing import Dict, Set, Optional

cell_dict: Dict[str, 'Cell'] = {}

class Cell:
    def __init__(self, value=None, child1=None, child2=None):
        self.value = value
        self.child1 = child1
        self.child2 = child2
        self.parents: Set['Cell'] = set()

    def get_value(self) -> int:
        if self.value is not None:
```

```
            return self.value
        if self.child1 not in cell_dict or self.child2 not in
cell_dict:
            raise ValueError(f"Cannot find children {self.child1} or
{self.child2}")
        self.value = cell_dict[self.child1].get_value() +
cell_dict[self.child2].get_value()
        return self.value

    def set_parent(self, parent: 'Cell', visited: Optional[Set] =
None):
        if visited is None:
            visited = set()
        if parent in visited:
            raise ValueError("Circular dependency detected")
        visited.add(parent)
        self.parents.add(parent)
        if self.child1:
            cell_dict[self.child1].set_parent(self, visited)
        if self.child2:
            cell_dict[self.child2].set_parent(self, visited)

    def invalidate(self):
        self.value = None
        for parent in self.parents:
            parent.invalidate()

class SpreadSheet:
    def get_cell_value(self, key: str):
        return cell_dict[key].get_value()

    def set_cell(self, key: str, cell: Cell):
        if key in cell_dict:
            cell_dict[key].invalidate()
        cell_dict[key] = cell
        if cell.child1:
            cell_dict[cell.child1].set_parent(cell)
        if cell.child2:
```

```
            cell_dict[cell.child2].set_parent(cell)
```

测试示例：

python
CopyEdit

```
spreadsheet = SpreadSheet()
spreadsheet.set_cell("A", Cell(6))
spreadsheet.set_cell("B", Cell(7))
spreadsheet.set_cell("C", Cell(None, "A", "B"))

assert spreadsheet.get_cell_value("C") == 13
spreadsheet.set_cell("A", Cell(5))
assert spreadsheet.get_cell_value("C") == 12
```

循环检测测试：

python
CopyEdit

```
spreadsheet.set_cell("D", Cell(10))
spreadsheet.set_cell("E", Cell(15))
spreadsheet.set_cell("F", Cell(None, "D", "E"))

try:
    spreadsheet.set_cell("D", Cell(None, "F", "E"))
except ValueError as e:
    assert str(e) == "Circular dependency detected"
```

English Version:

Design a spreadsheet system where each cell can:

- Hold a direct integer value
- Be the sum of two other cells

Features:

- `setCell(key, cell)`
- `getCellValue(key)`

Additional Requirements:

- Updating a cell must update dependency relationships.
- Detect cycles (e.g., A → B → A) and throw an exception.
- Use caching to improve read performance.
- Invalidate dependent caches when a cell changes.

Reference Code (Python):
(Same as above.)

Test Cases:
(Same as above.)

---

6. LRU Cache改版（带Priority的Heap）

中文说明：

要求实现一个支持访问次数优先级的缓存系统：

- `AddKey(key)`
- `GetCountForKey(key)`
- 维护一个最大堆，优先级是count大的key排在前面

注意：

- 如果访问次数变化了，要动态调整Heap。
- 允许访问次数相同情况下，根据key排序。

参考代码（**Python**）

python
CopyEdit

```
import sys

class Item:
    def __init__(self, key, count):
        self.key = key
        self.count = count

    def __gt__(self, other):
        if self.count == other.count:
            return self.key > other.key
        return self.count > other.count

    def add(self):
        self.count += 1

class MaxHeap:
    def __init__(self):
        self.h = [Item("fake", sys.maxsize)]
        self.pos = {}

    def par(self, pos):
        return pos // 2

    def swap(self, p1, p2):
        self.pos[self.h[p1].key], self.pos[self.h[p2].key] = p2, p1
        self.h[p1], self.h[p2] = self.h[p2], self.h[p1]

    def heapify(self, pos):
        while pos > 1 and self.h[pos] > self.h[self.par(pos)]:
            self.swap(pos, self.par(pos))
            pos = self.par(pos)

    def insert(self, item):
        self.h.append(item)
        self.pos[item.key] = len(self.h) - 1
```

```
        self.heapify(len(self.h) - 1)

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

class Cache:
    def __init__(self):
        self.heap = MaxHeap()

    def AddKey(self, key):
        if key not in self.heap.pos:
            self.heap.insert(Item(key, 1))
        else:
            item = self.heap.get(key)
            item.add()
            self.heap.heapify(self.heap.pos[key])

    def GetCountForKey(self, key):
        return self.heap.get(key).count

    def GetMaxItem(self):
        if len(self.heap.h) > 1:
            return self.heap.h[1].key
        return ""
```

测试示例：

python
CopyEdit

```
c = Cache()
c.AddKey(1)
c.AddKey(2)
c.AddKey(2)
c.AddKey(3)
c.AddKey(3)
c.AddKey(3)
assert c.GetMaxItem() == 3
assert c.GetCountForKey(1) == 1
```

English Version:

Design a cache that supports **priority based on access count**:

- `AddKey(key)`
- `GetCountForKey(key)`
- Maintain a **Max Heap** to track most accessed keys.

Requirements:

- Update heap whenever counts change.
- Tie-break using key ordering when counts are equal.

Reference Code (Python):
(Same as above.)

Test Cases:
(Same as above.)

---

7. Web Crawler多线程版

中文说明：

实现一个并发版的Web爬虫：

- 起始是一个URL
- 抓取所有同host的链接
- 需要用多线程加速抓取

参考代码（**Python ThreadPoolExecutor**版）

python
CopyEdit

```python
from concurrent.futures import ThreadPoolExecutor, as_completed

class Solution:
    def crawl(self, startUrl: str, htmlParser: 'HtmlParser') ->
list[str]:
        seen = set()
        parts = startUrl.split("/")
        host = parts[0] + "//" + parts[2]

        def dfs(url):
            urls = htmlParser.getUrls(url)
            result = []
            for new_url in urls:
                if new_url.startswith(host) and new_url not in seen:
                    seen.add(new_url)
                    result.append(new_url)
            return result

        with ThreadPoolExecutor(max_workers=16) as executor:
            seen.add(startUrl)
            queue = [startUrl]
            while queue:
                futures = [executor.submit(dfs, url) for url in queue]
                queue = []
                for f in as_completed(futures):
                    queue.extend(f.result())

        return list(seen)
```

English Version:

Implement a **concurrent web crawler**:

- Start from a URL
- Crawl all URLs under the same host
- Use multi-threading to accelerate crawling

Reference Code (Python):
(Same as above.)

---

8. 分布式系统Debug面试题（开放性问题）

中文说明：

给定一个大规模分布式系统，可能遇到以下问题：

- 节点宕机（Node failure）
- 网络延迟、丢包（Latency, Packet Loss）
- 负载过高（Overload）
- 数据不一致（Inconsistency）
- 死锁（Deadlock）
- 重试风暴（Retry Storm）

Debug方法举例：

- 监控 Metrics（如QPS, Latency）
- Trace全链路日志
- 加入心跳检查机制
- 多副本副本一致性检测
- 设置熔断器Circuit Breaker
- 建立Slow Request日志收集系统
- 分布式事务与重试策略设计

English Version:

Given a large-scale distributed system, typical issues include:

- Node failures
- Network latency, packet loss
- Overload and throttling
- Data inconsistency
- Deadlocks
- Retry storms

Debugging Approaches:

- Monitor Metrics (QPS, Latency)
- Use Distributed Tracing (end-to-end traces)
- Heartbeat checks
- Consistency checks among replicas
- Circuit breakers for overload control
- Slow request logging
- Careful design of distributed transactions and retry strategies