Tuần 2:

**Bài 1**: Tìm hiểu về Prefix Sum và ứng dụng trong bài toán counting sort

- Prefix Sum là một kĩ thuật “lưu vết" tổng của một tập hợp số đứng trước một phần tử trong dãy có thứ tự (tăng dần hoặc giảm dần).

- Bài toán đặt ra: Tính giá trị của một mảng số nguyên arr có kích cỡ N

arr[a]+arr[a+1]+⋯+arr[b]

với Q cặp [a,b] khác nhau thoả mãn 1≤a≤b≤N. Ta có ví dụ với một mảng N = 6

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **index i** | 1 | 2 | 3 | 4 | 5 | 6 |
| **arr[i]** | 1 | 6 | 4 | 2 | 5 | 3 |

- Với mỗi truy vấn, chúng ta có thể cộng lần lượt từng phần tử trong khoảng từ a đến b, như vậy thì độ phức tạp của phép toán là O(N). Với Q truy vấn (Q cặp [a,b]) sẽ có độ phức tạp là O(QN) → Quá lớn

- Thay vào đó, có thể sử dụng một mảng PrefixSum lưu vết để tính tổng của k phần tử phía trước phần tử thứ k. Đặt Prefix[0] = 0, với chỉ số k (1≤k≤N) ta có thể định nghĩa một prefix như sau:

prefix[k] =

- Có thể hiểu là Prefix[k] là phần tử tại chỉ số k của mảng và nó lưu trữ tổng của tất cả phần tử trong mảng gốc từ chỉ số 1 đến k. Bằng cách này thì độ phức tạp của từng truy vấn sẽ là O(1), với N truy vấn thì độ phức tạp của mảng sẽ là O(N)

prefix[k] = prefix[k-1] + arr[k]

//Cho một mảng số nguyên

int []prefix = new int [arr.length];

prefix = arr[1]

for i =1 to array.length:

prefix[i] = prefix[i-1] + arr[i]

- Ở ví dụ trên, với N = 6, mảng PrefixSum có tính như sau:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **index i** | 1 | 2 | 3 | 4 | 5 | 6 |
| **arr[i]** | 1 | 6 | 4 | 2 | 5 | 3 |
| **prefix[i]** | 1 | 7 | 11 | 13 | 18 | 21 |

- Mảng PrefixSum cho phép ta tìm tổng của bất kỳ mảng con. Nếu muốn tổng của mảng con từ i đến j (bao gồm cả chỉ số i và j) thì:

prefix[i,j] = prefix[j] – prefix[i-1]

- Theo ví dụ trên (N = 6):

prefix[2,5] = prefix[5] – prefix[1] = 18 -1 = 17 (=6+4+2+5)

//Cho một khoảng cho trước từ l đến r (lấy theo vị trí)

print the value of prefix[l,r]

“prefix[r] – prefix[l-1]”

Ứng dụng của PrefixSum trong bài toán CountingSort:

- Trong thuật toán CountingSort, chúng ta sử dụng một mảng đếm để đếm số lần xuất hiện của từng phần tử trong mảng đầu vào. Thay vì đếm số lần xuất hiện, ta có thể ứng dụng của PrefixSum để biết số lượng phần tử nhỏ hơn hoặc bằng giá trị trong mảng đếm, điều này giúp ta biết vị trí chính xác của mỗi phần tử trong mảng đầu ra mà không phải duyệt lại mảng đếm nhiều lần.

**COUNTING SORT (A,B,k)**

1. let *C*[0 .. *k*] be a new array

2. **for** *i* = 0 **to** *k*

3. *C*[i] = 0

4. **for** *j* = 1 **to** *A*.*length*

5. *C*[*A*[*j*]] = *C*[*A*[*j*]] + 1

6. *//C[j] chứa số lượng phần tử bằng i (đếm tần suất xuất hiện của các phần tử trong mảng)*

7. **for** i = 1 **to** k

8. C[i] = C[i] + C[i-1]

9. *//C[j] chứa số lượng của phần tử nhỏ hơn hoặc bằng i*

10. **for** j = A.length **downto** 1

11. *B*[*C*[*A*[*j*]]] = *A*[*j*]

12. *C*[*A*[*j*]] = *C*[*A*[*j*]]

**Bài 2**: Cho một chuỗi s và một từ điển dict. Hãy viết một phương thức add các khoảng trắng vào chuỗi s sao cho thành các câu có thể.

*Input:*

*s = "catsanddog"*

*wordDict = ["cat", "cats", "and", "sand", "dog"]*

*Output:*

*[*

*"cats and dog",*

*"cat sand dog"*

*]*

**Bài 3:**  Tìm hiểu về một số design pattern. Mô tả và implement lại chúng (phân tích so sánh dựa trên 1 case bài toán thực tế)

* SINGLETON
* FACTORY
* Chain of Responsibility
* Adapter Pattern

**• SINGLETON:**

- Mô tả: Là một design pattern cơ bản nhất trong lập trình. được sử dụng để đảm bảo rằng một lớp chỉ có duy nhất một thể hiện và cung cấp một cách truy cập toàn cục đến thể hiện đó.

- Sử dụng Singleton khi muốn:

+ Đảm bảo rằng chỉ có 1 instance(thể hiện) của một lớp

+ Quản lý việc truy cập tốt hơn vì chỉ có một lớp instance duy nhất

+ Có thể quản lý số lượng instance của một lớp trong giới hạn chỉ định

- Sử dụng Singleton khi:

+ Các bài toán cần truy cập vào ứng dụng như: Shared resource, Logger, Configuration, Caching, Thread pool, …

+ Một số design pattern khác cũng sử dụng Singleton để triển khai: Abstract Factory , Buider, Prototype, Facade, …

+ Được sử dụng trong một số class của core java như: java.lang.Runtime, java.awt.Desktop

**• FACTORY**

- Mô tả: Là một design pattern cơ bản thuộc nhóm Creational Design Pattern. Nhiệm vụ của nó là quản lý và trả về các đối tượng theo yêu cầu, giúp cho việc khởi tạo đối tượng một cách linh hoạt hơn

- Factory Method cung cấp một interface, phương thức trong việc tạo nên một đối tượng (object) trong class. Nhưng để cho class con kế thừa của nó có thể ghi đè để chỉ rõ đối tượng (object) nào sẽ được tạo. Factory method giao việc khởi tao một đối tượng (object) cụ thế cho lớp con (subclass)

- Mục đích: Tạo ra một cách khởi tạo Object mới thông qua một Interface chung

- Thành phần trong mô hình:

+ Product : Định nghĩa một khuôn mẫu (interface) của các đối tượng mà factory method tạo ra.

+ Concreteproduct: các lớp được cài đặt khuôn mẫu product.

+ Creator: Đây là lớp trừu tượng hoặc giao diện mô tả phương thức factory method để tạo ra đối tượng. Creator cung cấp một công cụ chung để tạo đối tượng, nhưng không định nghĩa cách chính xác để tạo đối tượng đó.

+ ConcreteCrator: ghi đè factory method để trả về một instance của concreteproduct.

- Ưu điểm:

+ Che giấu quá trình xử lý logic của phương thức khởi tạo

+ Hạn chế sự phụ thuộc giữa creator và concrete products

+ Dễ dàng mở rộng, thêm những đoạn code mới vào chương trình mà không cần phá vỡ các đối tượng ban đầu

+ Giúp gom các đoạn code tạo ra product vào một nơi trong chương trình, nhờ đó giúp dễ theo dõi và thao tác.

+ Giảm khả năng gây lỗi compile, trong trường hợp chúng ta cần tạo một đối tượng mà quên khai báo lớp, chúng ta cũng có thể xử lý lỗi trong Factory và khai báo lớp cho chúng sau.

- Nhược điểm:

+ Source code có thể trở nên phức tạp hơn mức bình thường do đòi hỏi phải sử dụng nhiều class mới có thể cài đặt được pattern này.

+ Việc refactoring ( tái cấu trúc ) một class bình thường có sẵn thành một class có Factory Method có thể dẫn đến nhiều lỗi trong hệ thống, phá vỡ sự tồn tại của clients

+ Factory method pattern lệ thuộc vào việc sử dụng private constructor nên các class không thể mở rộng và kế thừa

- Khi nào sử dụng:

+ Chúng ta có một super class với nhiều class con và dựa trên đầu vào, cần trả về một class con. Mô hình này giúp đưa trách nhiệm của việc khởi tạo một lớp từ phía client sang lớp Factory, giúp tiết kiệm tài nguyên hệ thống vì nhờ vào việc tái sử dụng các object đã có thay vì xây dựng lại mỗi phần có thêm product

+ Chúng ta không biết sau này sẽ cần đến những lớp con nào nữa. Khi cần mở rộng, hãy tạo ra sub class và implement thêm vào factory method cho việc khởi tạo sub class này.

**• CHAIN OF RESPONSIBILITY**

- Chain of resiponsibility là một trong những Design Pattern cơ bản, nó cho phép chúng ta xây dựng một chuỗi các đối tượng, mỗi đối tượng trong chuỗi có khả năng xử lý một yêu cầu nhưng không biết đối tượng tiếp theo trong chuỗi là ai. Yêu cầu sẽ được chuyển qua chuỗi cho đến khi nó được xử lý hoặc không còn đối tượng nào ở trong chuỗi.

- Thành phần:

+ Handler (Xử lý): Đây là giao diện hoặc lớp trừu tượng mô tả các phương thức xử lý yêu cầu và khai báo phương thức thiết lập đối tượng tiếp theo trong chuỗi.

+ Concrete Handlers (Xử lý cụ thể): Đây là các lớp cụ thể triển khai giao diện xử lý. Mỗi lớp xử lý cụ thể quyết định xem liệu nó có thể xử lý yêu cầu hay không. Nếu có thể, nó xử lý yêu cầu; nếu không, nó chuyển yêu cầu cho đối tượng tiếp theo trong chuỗi.

+ Client (Khách hàng): Đây là lớp hoặc đối tượng sử dụng mô hình Chain of Responsibility để gửi yêu cầu và nhận kết quả từ đối tượng xử lý đầu tiên trong chuỗi.

- Ưu điểm:

+ Giảm sự phụ thuộc giữa người gửi yêu cầu và người xử lý yêu cầu: Mô hình Chain of Responsibility giúp tách biệt người gửi yêu cầu và người xử lý yêu cầu. Người gửi yêu cầu không cần biết rõ về người xử lý cụ thể và ngược lại, người xử lý không cần biết ai đã gửi yêu cầu.

+ Mở rộng linh hoạt: Mô hình này cho phép thêm, loại bỏ và thay đổi các xử lý trong chuỗi một cách linh hoạt. Bạn có thể thêm một xử lý mới vào cuối chuỗi hoặc chèn một xử lý ở giữa hai xử lý hiện có mà không làm thay đổi code hiện có.

+ Tăng khả năng tái sử dụng: Các đối tượng xử lý có thể được sử dụng lại trong nhiều chuỗi khác nhau hoặc được thay đổi vị trí trong chuỗi mà không ảnh hưởng đến các đối tượng khác.

- Nhược điểm:

+ Yêu cầu có thể không được xử lý: Trong trường hợp không có xử lý nào trong chuỗi có thể xử lý yêu cầu, yêu cầu sẽ không được xử lý. Điều này có thể dẫn đến một tình huống không mong muốn nếu không có xử lý mặc định hoặc xử lý cuối cùng trong chuỗi.

+ Tăng độ phức tạp: Khi chuỗi trở nên quá lớn và phức tạp, việc theo dõi và quản lý các đối tượng xử lý có thể trở nên khó khăn. Việc sắp xếp các đối tượng xử lý và thiết lập đúng thứ tự trong chuỗi cũng có thể là một thách thức.

- Mô hình Chain of Responsibility được sử dụng phổ biến trong các tình huống như xử lý yêu cầu trong hệ thống giao diện người dùng, xử lý sự kiện, xử lý yêu cầu HTTP, xử lý các tác vụ trong công nghiệp, v.v.

**• ADAPTER PATTERN**

- Là một trong những Pattern thuộc nhóm cấu trúc (Structural Pattern). Mô hình này cho phép các interface không liên quan tới nhau có thể làm việc với nhau. Đối tượng giúp kết nối các Interface gọi là Adapter.

- Adapter Pattern giữ vai trò trung gian giữa hai lớp, chuyển đổi interface của một hay nhiều lớp có sẵn thành một interface khác, thích hợp cho lớp đang viết.

- Còn được gọi là Wrapper Pattern do cung cấp một Interface “bọc ngoài” tương thích cho một hệ thống có sẵn, có dữ liệu hành vi phù hợp nhưng có Interface không tương thích với lớp đang viết

- Một adapter bao gồm những thành phần cơ bản sau:

+ Adaptee: định nghĩa Interface không tương thích, cần được tích hợp vào

+ Adapter: lớp tích hợp, giúp Interface không tương thích với Interface đang làm việc. Thực hiện việc chuyển đổi Interface cho Adaptee và kết nối Adaptee với client.

+ Target: Một Interface chứa các chức năng được sử dụng bởi Client (domain specific)

+ Client: Lớp này sử dụng các đối tượng có Interface Targer

- Lợi ích:

+ Cho phép nhiều đối tượng có Interface giao tiếp khác nhau có thể tuơng tác và giao tiếp với nhau

+ Tăng khả năng sử dụng lại các thư viện với Interface không thay đổi do không có mã nguồn.

- Khuyết điểm:

+ Do tất cả các yêu cầu được chuyển tiếp nên chi phí tăng lên không ít

+ Đôi khi có quá nhiều Adapter được thiết kế trong một chuỗi Adapter trước khi đến được với yêu cầu thật sự

- Sử dụng khi nào:

+ Khi muốn sử dụng một lớp đã tồn tại trước đó nhưng interface sử dụng không phù hợp như mong muốn.

+ Khi muốn tạo ra những lớp có khả năng sử dụng lại, chúng phối hợp với các lớp không liên quan hay những lớp không thể đoán trước được và những lớp này không có những interface tương thích.

+ Cần phải có sự chuyển đổi interface từ nhiều nguồn khác nhau.

+ Khi cần đảm bảo nguyên tắc Open/ Close trong một ứng dụng.

- Một vài class sử dụng Adapter Pattern:

+ java.util.Arrays#asList()

+ java.io.InputStreamReader(InputStream) (returns a Reader)

+ java.io.OutputStreamWriter(OutputStream) (returns a Writer)

+ javax.xml.bind.annotation.adapters.XmlAdapter#marshal() và #unmarshal()