# JAVA CORE BASIC

### 1. Kỹ thuật lập trình cơ bản

### a. Công cụ lập trình

Công cụ lập trình đã cài đặt là intellij.

### b. Các kiểu dữ liệu trong java

* Kiểu dữ liệu nguyên thủy:

|  |  |  |
| --- | --- | --- |
| **Kiểu dữ liệu** | **Kích thước** | **Giá trị mặc định** |
| byte | 1 byte | 0 |
| short | 2 byte | 0 |
| int | 4 byte | 0 |
| long | 8 byte | 0L |
| float | 4 byte | 0.0f |
| double | 8 byte | 0.0d |
| char | 2 byte | '\u0000' |
| boolean | 1 bit | false |

* Kiểu dữ liệu đối tượng: Là kiểu dữ liệu được lưu trữ trong bộ nhớ heap, các kiểu dữ liệu String, Array, Đối tượng được tạo bằng từ khoá new, Wrapper Classes(Integer, double, Double ,char , Character ,boolean , Boolean)

### c. Câu lệnh if, for, while, do-while, và switch-case:

* **Câu lệnh If**: được sử dụng để kiểm tra một điều kiện và thực hiện một đoạn mã nếu điều kiện đó đúng.

**Ví dụ**:

int X= 1;

if (x >= 3) {

System.out.println("X lớn hơn 3");

}

Dùng để kiểm tra một điều kiện đơn giản.Nếu điều kiện đúng, sẽ thực thi mã bên trong khối if.

* **Câu lệnh For:** For được sử dụng để lặp qua một dãy giá trị, thường dùng khi bạn biết trước số lần lặp.

Ví dụ:

for (int i = 0; i < 5; i++) { System.out.println(i); }

Thường dùng để lặp qua một tập hợp giá trị xác định.

* **Câu lệnh while:** while thực hiện lặp lại một đoạn mã miễn là điều kiện đúng.

**Ví dụ:**

int i = 0;

while (i < 5) {

System.out.println(i);

i++;

}

Thực hiện vòng lặp khi điều kiện đúng.

Điều kiện được kiểm tra trước khi vào vòng lặp, nên nếu điều kiện sai ngay từ đầu, vòng lặp sẽ không được thực thi.

* **Câu lệnh do-while:** do-while giống như while, nhưng nó kiểm tra điều kiện sau khi thực thi đoạn mã ít nhất một lần.

**Ví dụ:**

int i = 0;

do {

System.out.println(i);

i++;

} while (i < 5);

Thực thi đoạn mã ít nhất một lần, sau đó mới kiểm tra điều kiện.

Dùng khi bạn cần thực hiện một hành động trước khi kiểm tra điều kiện.

* **Câu lệnh** switch-case: switch-case được sử dụng để kiểm tra nhiều điều kiện, và có thể thay thế cho nhiều câu lệnh if-else khi kiểm tra một biến với nhiều giá trị khác nhau.

**Ví dụ:**

int day = 2;

switch (day) {

case 1:

System.out.println("Monday");

break;

case 2:

System.out.println("Tuesday");

break;

case 3:

System.out.println("Wednesday");

break;

default:

System.out.println("Invalid day");

}

Dùng để thay thế cho nhiều câu lệnh if-else khi kiểm tra một biến với nhiều giá trị khác nhau.

Break dùng để thoát khỏi switch sau khi thực hiện một case.

### d. Một số thuật toán làm việc với mảng

* Thuật toán Bubble Sort: Duyệt qua mảng và so sánh từng cặp phần tử kề nhau, nếu phần tử trước lớn hơn phần tử sau thì đổi chỗ chúng. Tiếp tục như vậy cho đến khi mảng được sắp xếp.

int[] arr = {5, 3, 8, 4, 2};

for (int i = 0; i < arr.length - 1; i++) {

for (int j = 0; j < arr.length - 1 - i; j++) {

if (arr[j] > arr[j + 1]) {

int temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

}

}

}

* Thuật toán Selection Sort: Tìm phần tử nhỏ nhất trong mảng và hoán đổi nó với phần tử đầu tiên, sau đó tiếp tục tìm phần tử nhỏ nhất trong phần mảng còn lại.

int[] arr = {5, 3, 8, 4, 2};

for (int i = 0; i < arr.length - 1; i++) {

int minIndex = i;

for (int j = i + 1; j < arr.length; j++) {

if (arr[j] < arr[minIndex]) {

minIndex = j;

}

}

int temp = arr[minIndex];

arr[minIndex] = arr[i];

arr[i] = temp;

}

* Thuật toán Insertion Sort: Duyệt qua từng phần tử, so sánh nó với các phần tử đã sắp xếp, và chèn nó vào vị trí phù hợp trong phần đã sắp xếp.

int[] arr = {5, 3, 8, 4, 2};

for (int i = 1; i < arr.length; i++) {

int key = arr[i];

int j = i - 1;

while (j >= 0 && arr[j] > key) {

arr[j + 1] = arr[j];

j--;

}

arr[j + 1] = key;

}

* Thêm phần tử vào mảng: Trong Java, mảng có kích thước cố định, khi cần thêm phần tử cần phải tạo mảng mới có kích thước lơn hơn.
* Xoá phần tử vào mảng: khi xóa một phần tử trong mảng, cần phải di chuyển các phần tử còn lại để lấp đầy chỗ trống.
* Tìm kiếm nhị phân: Tìm kiếm nhị phân là phương pháp tìm kiếm nhanh hơn khi dữ liệu đã được sắp xếp. Thuật toán này phân chia tập hợp dữ liệu thành hai nửa và so sánh giá trị trung bình với phần tử cần tìm. Nếu phần tử nhỏ hơn giá trị trung bình, tìm kiếm tiếp tục ở nửa trái, ngược lại, tìm kiếm tiếp tục ở nửa phải.

### e. Streams Input/Out put, làm việc với file mức cơ bản.

Trong Java, **Streams** là một khái niệm được sử dụng để đọc và ghi dữ liệu. **Streams** có thể làm việc với nhiều loại dữ liệu như file. Java cung cấp một hệ thống **Streams** mạnh mẽ cho phép xử lý các luồng dữ liệu theo cả hai hướng: **Input** (đọc) và **Output** (ghi).

Các loại stream trong java:

* **FileInputStream và FileOutputStream**: Đọc và ghi dữ liệu dưới dạng byte
* **BufferedInputStream và BufferedOutputStream:** Đọc và ghi dữ liệu từ một luồng vào và lưu vào bộ đệm để giảm thiểu số lần truy cập vào nguồn dữ liệu.
* **ObjectInputStream và ObjectOutputStream:** Đọc và ghi đối tượng đã được tuần tự hóa (serialization).
* **DataInputStream và DataOutputStream:** Đọc và ghi dữ liệu cơ bản (int, float, double, boolean...) từ một nguồn nhập liệu.

## 2. Lập trình hướng đối tượng

### a. Lập trình hướng đối tượng và class

* **Đối tượng**:

Đ**ối tượng** (object) là một thực thể cụ thể của một **class**. Nó là một thể hiện của class, mang các thuộc tính và hành vi của class đó.

* **Thuộc tính** : Là các biến hoặc dữ liệu lưu trữ thông tin của đối tượng.
* **Hành vi** : Là các phương thức hoặc chức năng mà đối tượng có thể thực hiện.
* **Class**

Class trong Java là một khuôn mẫu hoặc bản thiết kế dùng để tạo ra các đối tượng. Class định nghĩa các thuộc tính và phương thức mà các đối tượng của class đó sẽ có.Thuộc tính trong class là các biến mà tất cả các đối tượng của class này sẽ chia sẻ.Phương thức trong class là các hàm hoặc chức năng mà các đối tượng của class này có thể thực thi.Class có thể được coi là một mô tả hoặc khung mẫu của đối tượng. Một đối tượng thực tế được tạo ra từ một class khi chương trình chạy.

### b. Các đặc điểm trong Lập trình Hướng Đối Tượng

Lập trình hướng đối tượng có 4 tính chất:

* Tính đa hình: khả năng của một đối tượng trong Java có thể thể hiện nhiều hình thức khác nhau
  + **Overloading:** lviệc **định nghĩa nhiều phương thức trong cùng một lớp** với cùng tên nhưng khác nhau về số lượng tham số hoặc kiểu tham số. Điều này cho phép bạn sử dụng cùng một tên phương thức nhưng thực hiện các hành động khác nhau tùy theo số lượng hoặc loại tham số.
  + **Overriding:** khả năng của lớp **con** rong việc **định nghĩa lại phương thức đã có trong lớp cha** . Phương thức trong lớp con phải có cùng tên, kiểu trả về và tham số như trong lớp cha. **Overriding** cho phép lớp con cung cấp cách cài đặt cụ thể của phương thức mà lớp cha đã khai báo.
* Tính kế thừa: **Thừa kế** là cơ chế cho phép một lớp con kế thừa các thuộc tính và phương thức từ lớp cha, giúp tái sử dụng mã nguồn và mở rộng các tính năng mà không cần phải viết lại mã.
* Tính đóng gói: **Đóng gói** là quá trình ẩn giấu các chi tiết thực thi và chỉ cung cấp các phương thức để tương tác với dữ liệu bên trong lớp. Mục đích chính của đóng gói là bảo vệ dữ liệu khỏi sự truy cập trái phép và sai lệch, đồng thời cung cấp một giao diện rõ ràng cho người dùng tương tác, chỉ.
* Tính trừu tượng: **Trừu tượng** là quá trình ẩn các chi tiết cụ thể và chỉ cung cấp các phương thức và thuộc tính cần thiết cho người dùng. Lớp trừu tượng chỉ định nghĩa các phương thức (không có phần thân hàm) mà lớp con sẽ phải thực thi. Điều này giúp che giấu các chi tiết cài đặt phức tạp, đồng thời chỉ hiển thị các phương thức hoặc tính năng quan trọng.

### c. Cách khai báo lớp, thuộc tính, phương thức, constructor, cách phân chia các class?

* Khai báo lớp có các cách khai báo:
  + Cách Khai báo lớp thường : class ClassName { }
  + Cách khai báo lớp trừu tượng: abstract class ClassName { }
  + Cách khai Interface: interface InterfaceName { }
  + Cách khai lớp tĩnh: static class InnerClass { }
  + Cách khai lớp ẩn danh: ClassName obj = new ClassName() { ... }
* Khai báo thuộc tính:
  + Private
  + Public
  + Protected
  + Default

## 3. Exception handling

### a. Khai báo ngoại lệ

Từ khoá throws: thông báo cho chương trình là một phương thức có thể **nén ra** một ngoại lệ cụ thể trong quá trình thực thi, dừng xử lý luồng hiện tại. Ví dụ

* public void funciton() **throws IOException {}**

Trong Java có 3 loại exception là Error, checked exception, và unchecked exception

Error: lỗi nghiêm trọng mà không kiểm soat được như lỗi phần cứng, tràn bộ nhớ. Kế thừa lớp Error không phải từ lớp expection

Checked exceptions: là những ngoại lệ mà chương trình có thể **dự đoán và xử lý trước,** xảy ra trong quá trình compile time, và cần xử lý ngay khi thực thi code nếu không xử lý thì chương trình sẽ không chạy.

**UnChecked exceptions:** Là loại exception xảy ra tại thời điểm thực thi chương trình, không bị biên dịch yêu cầu phải xử lý,Ví dụ ngoại lệ: NullPointerException, ArrayIndexOutOfBoundsException, DivideByZeroException...

### b. Cơ chế try-catch

Cấu trúc của try catch:

try {

// Đoạn mã có thể gây ra ngoại lệ

} catch (Exception e) {

// Xử lý ngoại lệ

} finally {}

* **Khối try**: Đầu tiên, chương trình sẽ thực thi đoạn mã trong khối try. Nếu không có ngoại lệ xảy ra, chương trình sẽ tiếp tục thực thi các câu lệnh sau đó và sẽ không vào khối catch.
* **Khối catch**: Nếu có ngoại lệ xảy ra trong khối try, chương trình sẽ nhảy vào khối catch tương ứng với loại ngoại lệ đó và thực thi mã trong khối catch, chương trình sẽ không bị dừng và chạy tiếp.
* **Khối finally**: Được sử dụng để thực thi mã dọn dẹp hoặc mã cần chạy bất kể có xảy ra ngoại lệ hay không.

### c. Throw ngoại lệ

**Throw** là từ khóa trong Java được sử dụng để **ném** một ngoại lệ (exception) ra ngoài từ một phương thức, khiến chương trình dừng lại tại điểm ném và chuyển điều khiển tới phần xử lý ngoại lệ.

**throw** được sử dụng để ném một ngoại lệ từ trong một phương thức hoặc khối mã, và chương trình sẽ chuyển sang khối catch của ngoại lệ đó nếu nó được khai báo và xử lý.

Khi **throw** được sử dụng, bạn có thể ném ra bất kỳ loại ngoại lệ nào, bao gồm **checked exceptions** (ngoại lệ cần khai báo hoặc xử lý) và **unchecked exceptions** (ngoại lệ không cần khai báo hoặc xử lý).

### d. Chuyển tiếp ngoại lệ

**Chuyển tiếp ngoại lệ** (Exception Propagation) trong Java là một cơ chế mà theo đó ngoại lệ được **truyền từ phương thức này sang phương thức khác** nếu nó không được xử lý ở phương thức hiện tại. Nếu một phương thức ném ra một ngoại lệ và phương thức đó không xử lý được ngoại lệ, thì ngoại lệ sẽ được "chuyển tiếp" lên phương thức gọi nó cho đến khi nó được bắt và xử lý (hoặc nếu không được xử lý thì chương trình sẽ dừng lại.

* Khi một ngoại lệ xảy ra trong phương thức, nếu phương thức đó không xử lý ngoại lệ, nó có thể **ném ngoại lệ** ra ngoài bằng cách sử dụng từ khóa throw.
* Nếu phương thức có thể ném ra một ngoại lệ, cần khai báo throw, phương thức bắt ngoại lệ của phương thức loại lệ sẽ xử lý nó.
* Nếu một phương thức không xử lý ngoại lệ, ngoại lệ đó sẽ được chuyển tiếp lên phương thức gọi nó

4. Lập tình xử lý với database

* **Stored Procedure** trong Oracle là một đoạn mã Procedural Language/Structured Query Language được **lưu trữ trong cơ sở dữ liệu** và có thể được **gọi lại và thực thi** khi cần thiết. Stored procedure giúp tách biệt mã xử lý khỏi ứng dụng, giúp giảm sự phụ thuộc vào ứng dụng và đảm bảo mã thực thi được chia sẻ và tái sử dụng.

Ưu điểm:

Tái sử dụng mã: Giúp giảm sự trùng lặp mã trong ứng dụng.

Tăng hiệu suất: Quá trình biên dịch chỉ thực hiện một lần, giúp giảm thời gian xử lý.

Bảo mật: Quyền truy cập có thể được kiểm soát để hạn chế người dùng thực hiện các thao tác trực tiếp trên bảng.

Nhược điểm:

Khó bảo trì: Khi ứng dụng phát triển, việc quản lý thủ tục lưu trữ có thể trở nên khó khăn.

Khó khăn trong debug: Quá trình kiểm tra lỗi có thể phức tạp khi có vấn đề xảy ra trong Stored Procedure.

* **Function** trong Oracle tương tự như một thủ tục lưu trữ, nhưng khác biệt ở chỗ nó luôn phải **trả về một giá trị**. Hàm có thể nhận tham số và thực hiện một tác vụ, sau đó trả về một kết quả. Hàm có thể được sử dụng trong các câu lệnh SQL, điều này tạo ra sự linh hoạt cao trong việc xử lý dữ liệu trực tiếp trong cơ sở dữ liệu.

**Ưu điểm:**

**Tính linh hoạt**: Có thể được sử dụng trực tiếp trong câu lệnh SQL, giúp xử lý dữ liệu ngay trong cơ sở dữ liệu.

**Khả năng tái sử dụng**: Giảm thiểu sự lặp lại mã khi cần thực hiện phép toán hoặc xử lý dữ liệu.

**Nhược điểm:**

**Không thể thay đổi trạng thái cơ sở dữ liệu**: Function không thể thay đổi dữ liệu trong bảng, do đó chỉ thích hợp với các tác vụ tính toán hoặc xử lý.

**Giới hạn trong xử lý lỗi**: Quản lý lỗi trong Function có thể không linh hoạt như trong Stored Procedure.

* **Trigger** trong Oracle là một đoạn mã PL/SQL tự động thực thi khi có sự kiện cụ thể xảy ra trong cơ sở dữ liệu, chẳng hạn như chèn, cập nhật hoặc xóa dữ liệu. Triggers được sử dụng để thực thi các hành động tự động trong cơ sở dữ liệu mà không cần yêu cầu thao tác trực tiếp từ người dùng, như ghi log, kiểm tra dữ liệu hợp lệ, hoặc tự động cập nhật các bảng khác khi dữ liệu thay đổi.

Ưu điểm:

* + Tự động hóa: Thực thi tự động khi có sự kiện xảy ra mà không cần thao tác từ người dùng.
  + Đảm bảo tính toàn vẹn dữ liệu: Có thể tự động kiểm tra dữ liệu hoặc thực hiện các hành động khi dữ liệu thay đổi, giúp duy trì tính toàn vẹn cho cơ sở dữ liệu.

Nhược điểm:

* + Ảnh hưởng đến hiệu suất: Nếu sử dụng quá nhiều Trigger hoặc các Trigger phức tạp, có thể làm giảm hiệu suất của hệ thống.
  + Khó theo dõi: Triggers có thể làm phức tạp việc theo dõi và debug hệ thống do chúng thực thi tự động.
* **Sequence** là một đối tượng trong Oracle giúp tạo ra một dãy số tăng dần, thường được sử dụng để tạo giá trị cho các cột khóa chính tự động . Sequence có thể được sử dụng để lấy các giá trị số liên tục trong cơ sở dữ liệu mà không cần sự can thiệp của người dùng.

Ưu điểm:

Tạo giá trị tự động: Giúp tạo ra các giá trị duy nhất và tự động cho các khóa chính.

Tiết kiệm thời gian: Không cần phải theo dõi hoặc tạo số thủ công.

Nhược điểm:

Không thể giảm giá trị: Một khi số đã được tạo ra, không thể giảm lại, điều này có thể gây khó khăn khi cần điều chỉnh lại dữ liệu.

Không sử dụng được trong trường hợp phân tán: Trong các hệ thống phân tán, Sequence có thể gặp vấn đề khi đồng bộ hóa giữa các cơ sở dữ liệu.

* **Partition** trong Oracle giúp chia bảng lớn thành các phần nhỏ hơn . Mỗi **Partition** lưu trữ một phần dữ liệu của bảng, giúp cải thiện hiệu suất khi truy vấn hoặc thao tác với dữ liệu. Có nhiều cách phân vùng bảng, như phân vùng theo dải giá trị, phân vùng theo danh sách , hoặc phân vùng theo hash
* **Job** trong Oracle là một tác vụ được lên lịch để thực thi tự động vào một thời điểm cụ thể hoặc lặp lại theo lịch trình. Jobs có thể được sử dụng để tự động hóa các công việc trong cơ sở dữ liệu như sao lưu, tạo báo cáo, hoặc thực hiện các tác vụ bảo trì định kỳ. Các jobs này có thể được quản lý qua Oracle Scheduler.
* **Index** trong Oracle là một cấu trúc dữ liệu được sử dụng để tăng tốc quá trình truy vấn dữ liệu. Index giúp giảm thời gian tìm kiếm dữ liệu trong các bảng lớn bằng cách tạo ra một bản sao sắp xếp của một hoặc nhiều cột trong bảng. Khi thực hiện các truy vấn, Oracle sẽ sử dụng chỉ mục để truy xuất dữ liệu nhanh chóng thay vì phải quét toàn bộ bảng .

Đã thực hành trong file OracleCore.sql.

Kết nối java với oracle: Cài đặt Oracle JDBC Driver để kết nối với oracle.

Đang thực hành trong file config-java-oracle.

6. Đóng gói chương trình, run file trên windows, linux.

**Đóng gói:**

1. Sử dụng lệnh javac để biên dịch mã nguồn Java thành bytecode (.class).
2. Để đóng gói chương trình Java thành một file duy nhất có thể chạy được, Cần có thể tạo một file JAR. Đầu tiên, Cần tạo file MANIFEST.MF trong thư mục META-INF.
3. Sử dụng jar để đóng gói.

Chạy chương trình trên window:  
 1. Chạy JAR trên windows: java -jar Main.jar, Main.jar là thư mục chứa java đóng gói.

# JAVA CORE ADVANCE

## Streams Input/out put, làm việc với file mức nâng cao ưu nhược điểm.

Streams trong Java là các khái niệm để **đọc** và **ghi dữ liệu** từ các nguồn và đến các đích (ví dụ như từ file, bộ nhớ, mạng). Khi làm việc với Streams, không cần làm việc trực tiếp với từng byte hoặc ký tự mà thay vào đó sử dụng các Streams để xử lý chúng một cách hiệu quả.

Các loại Stream chính:

* **Byte Streams: Byte Streams** được sử dụng để đọc và ghi **dữ liệu nhị phân** (binary data). Dữ liệu nhị phân có thể là bất kỳ dạng dữ liệu nào không phải là văn bản, như ảnh, video, âm thanh… Các lớp **Byte Streams** trong Java bao gồm:
  + **InputStream**: Đọc dữ liệu từ nguồn.
  + **OutputStream**: Ghi dữ liệu vào đích.
* **Character Streams** (Reader/Writer): **Character Streams** được sử dụng để đọc và ghi **dữ liệu văn bản** (text data). Dữ liệu văn bản được mã hóa thành các ký tự (characters) trong các bộ mã như **UTF-8** hoặc **ASCII**. **Character Streams** trong Java bao gồm:
  + Reader: Đọc văn bản từ nguồn.
  + Writer: Ghi văn bản vào đích.
* Java **NIO (New I/O)** là một API mạnh mẽ hơn và hiệu quả hơn để làm việc với I/O trong Java, đặc biệt khi bạn làm việc với các file lớn hoặc cần xử lý nhiều kết nối đồng thời. NIO hoạt động thông qua các lớp **ByteBuffer** và **FileChannel.** Thay vì xử lý từng byte dữ liệu như trong **Byte Streams,** NIO cho phép bạn xử lý dữ liệu trong bộ nhớ.
* Buffers: Đọc và ghi dữ liệu vào bộ nhớ (Memory Buffers).
* Channels: Thực hiện việc giao tiếp I/O không đồng bộ (asynchronous I/O) và tối ưu hơn trong việc làm việc với các file lớn.
* Selectors: Giúp lựa chọn các kênh (channels) và thực hiện các tác vụ I/O đồng thời.
* NIO.2 (File I/O improvements):

Ưu điểm và nhược điểm:

**Byte Streams**

**Ưu điểm**:

* Dễ dàng sử dụng để làm việc với dữ liệu nhị phân như hình ảnh, video, và âm thanh.
* Tối ưu cho các tác vụ xử lý dữ liệu byte.

**Nhược điểm**:

* Không thuận tiện khi làm việc với dữ liệu văn bản, vì không tự động xử lý mã hóa ký tự.

**Character Streams**

**Ưu điểm**:

* Tiện lợi khi làm việc với các dữ liệu văn bản, tự động xử lý mã hóa và giải mã ký tự.
* Tối ưu cho các file văn bản như .txt, .csv, .xml.

**Nhược điểm**:

* Không phù hợp khi làm việc với dữ liệu nhị phân vì có thể làm mất dữ liệu nếu chuyển đổi không đúng.

**NIO**

**Ưu điểm**:

* Hiệu suất cao, đặc biệt là khi làm việc với các file lớn và xử lý đồng thời.
* **Buffers** và **Channels** cung cấp khả năng tối ưu trong việc xử lý I/O.
* **Non-blocking I/O**: Cung cấp khả năng xử lý không chặn, thích hợp cho các ứng dụng yêu cầu đồng thời.

**Nhược điểm**:

* Cần hiểu rõ hơn về các khái niệm như **ByteBuffer** và **Channel**.
* Lập trình phức tạp hơn so với **Byte Streams** và **Character Streams**.

**NIO.2**

**Ưu điểm**:

* Các tính năng mới như **WatchService** giúp theo dõi sự thay đổi trong hệ thống file.
* Cải tiến trong việc làm việc với các hệ thống file đặc biệt hoặc không phải hệ thống file truyền thống.

**Nhược điểm**:

* NIO.2 yêu cầu nhiều mã lệnh hơn và có thể làm tăng sự phức tạp trong ứng dụng.

## 3. Thread

**a. Khái niệm:**

**Thread:**

**-** Đơn vị nhỏ nhất của một tiến trình, Mỗi tiến trình có ít nhất một luồng một tiến trình có thể tạo ra nhiều luồng con để thực hiện các tác vụ đồng thời.

Trong Java là một đơn vị thực thi cơ bản của chương trình. cho phép thực hiện các tác vụ riêng biệt đồng thời, thực thi song song tận dụng khả năng đa lõi của CPU thay vì thực thi theo luồng(single thread).

- **Luồng** là một phần của tiến trình và chia sẻ không gian bộ nhớ của tiến trình cha. Vì thế, việc tạo luồng mới tốn ít tài nguyên hơn so với việc tạo một tiến trình mới. Vì chia sẻ bộ nhớ nên khi một lồng bị treo thì ảnh hưởng đến toàn bộ tiến trình chứa luồng đó.

- Process : một process bị treo thì sẽ không ảnh hưởng đến tiến trình khác.

**Time Slicing:**

**- Đối với hệ thống chỉ có một cpu thì sẽ chia sẻ thời gian xử lý của cpu cho nhiều tiến tình và luồng trong hệ thống,** mỗi khoảng này có thể chỉ vài mili giây. một hệ thống với ba tiến trình A, B, C và một lõi CPU. Mỗi tiến trình sẽ được cấp một **time slice là 10ms.**

**- Đối với hệ thống có nhiều nhân và luồng:**

* Các process sẽ chạy song song trên các nhân
* Hệ điều hành vẫn sử dụng **time slicing** để phân bổ tài nguyên giữa các tiến trình. Tuy nhiên, thay vì chỉ một nhân chia sẻ thời gian, thời gian xử lý sẽ được chia giữa các nhân và luồng trong toàn bộ hệ thống.

Process:

- Giống như một application nhưng cũng có thể application được tạo bảo nhiều quy trình.

- Một **process** là một môi trường thực thi độc lập. Mỗi tiến trình có một không gian bộ nhớ riêng biệt(Ram) và bộ tài nguyên cần thiết để thực thi chương trình(CPU). trừ khi sử dụng **IPC(có thể giao tiếp với nhau).**

- Ví dụ : **Trình duyệt web**: Khi mở một trình duyệt web như **Google Chrome** hoặc **Mozilla Firefox**, hệ điều hành sẽ tạo ra một **process** cho mỗi trình duyệt. Nếu bạn mở nhiều cửa sổ hoặc tab, mỗi cửa sổ/tab có thể là một **sub-process** của tiến trình chính.

Ví dụ:

* Một ứng dụng chat có thể có một thread xử lý việc nhận tin nhắn và một thread khác xử lý việc gửi tin nhắn, cho phép hai tác vụ này xảy ra cùng lúc.
* Ứng dụng web server có thể xử lý hàng nghìn yêu cầu đồng thời bằng cách sử dụng các thread khác nhau.

**b. Khởi tạo một Thread:**

* Cách 1: Kế thừa lớp Thread
  + Tạo một lớp con kế thừa từ lớp Thread và ghi đè phương thức run().
  + Sau khi tạo đối tượng từ lớp con, gọi phương thức start() để bắt đầu thực thi.
* Cách 2: Triển khai Interface Runnable:
* Tạo một lớp triển khai interface Runnable và ghi đè phương thức run().
* Sau đó, tạo đối tượng Thread và truyền đối tượng Runnable vào constructor của Thread, gọi start() để thực thi.

|  |  |  |
| --- | --- | --- |
|  | **ƯU điểm** | **Nhược điểm** |
| **EXTEND THREAD** | - Đơn giản và dễ sử dụng | - Không thể kế thừa từ lớp khác - Khó tái sử dụng mã |
| **IMPLEMENT RUNABLE** | - Linh hoạt và tái sử dụng tốt hơn - Hỗ trợ chia sẻ tài nguyên giữa các thread | - Cần phải tạo đối tượng Thread  - Phức tạp hơn với trường hợp đơn giản. |

**c. Thread Priority:**

**Thread Priority** (Ưu tiên của thread) là mức độ ưu tiên mà thread có trong quá trình thực thi. Java cho phép đặt độ ưu tiên cho thread từ 1 (thấp) đến 10 (cao). Mặc dù, JVM và hệ điều hành sẽ quyết định thread nào sẽ thực thi dựa trên các ưu tiên này, nhưng có thể sử dụng phương thức setPriority() để điều chỉnh mức độ ưu tiên của một thread. Giúp ưu tiên các tác vụ quan trọng hơn trong các ứng dụng đa nhiệm. Không phải lúc nào thread có độ ưu tiên cao cũng sẽ chạy trước, điều này phụ thuộc vào lịch trình của hệ điều hành.

**d. Vòng đời của 1 Thread và các trạng thái:**

Trong Java, **Thread** là một đơn vị thực thi độc lập, và mỗi thread có một **vòng đời** riêng. Vòng đời của một thread bắt đầu từ khi nó được tạo ra và kết thúc khi nó hoàn thành công việc của mình. Trong suốt vòng đời của mình, thread có thể trải qua nhiều **trạng thái** khác nhau, và mỗi trạng thái phản ánh các bước tiến hành trong quá trình thực thi.

**Các trạng thái của Thread:**

**New:** Khi một **Thread** được tạo ra, nó ở trạng thái **New**. Trong trạng thái này, thread chưa được khởi động và không thực thi bất kỳ công việc nào. Đây là bước đầu tiên trong vòng đời của một thread.

**Runnable**: Khi phương thức start() được gọi, thread chuyển sang trạng thái **Runnable**. và ở trong trạng thái này, nó được xem như đang thực hiện tác vụ của chính mình.

**Block**: Thread có thể chuyển sang trạng thái **Blocked** khi nó đang chờ tài nguyên bị khóa. Điều này xảy ra khi một thread cần truy cập tài nguyên mà thread khác đang giữ. Khi tài nguyên được giải phóng, thread sẽ chuyển lại về trạng thái **Runnable** để tiếp tục thực thi.

**Waiting**: Khi thread chờ một sự kiện nào đó (chẳng hạn như chờ một thread khác thực hiện xong công việc hoặc chờ một điều kiện nào đó được đáp ứng), nó sẽ chuyển sang trạng thái Waiting. Thread trong trạng thái này sẽ không thực thi cho đến khi có một thread khác gửi tín hiệu cho nó tiếp tục.

Các phương thức có thể khiến thread chuyển sang trạng thái Waiting:

* wait(): Thread sẽ tạm dừng và chờ cho đến khi có một thread khác gọi notify() hoặc notifyAll().
* join(): Thread sẽ đợi một thread khác hoàn thành trước khi tiếp tục.

**Terminated** :chuyển sang trạng thái **Terminated** khi nó đã hoàn thành công việc của mình hoặc bị dừng đột ngột (có thể do lỗi hoặc khi run() thực thi xong). Sau khi kết thúc, thread không thể quay lại trạng thái **Runnable** nữa.

**e. Deadlock**

xảy ra khi hai hoặc nhiều thread không thể tiến hành vì chúng đang chờ nhau giải phóng tài nguyên mà chúng đang giữ Điều này dẫn đến một vòng lặp vô hạn, có thể sẽ không được giải phóng vào lúc này., nơi các thread không thể tiếp tục thực thi và chương trình bị "đóng băng".

Điều kiện để deadlock xảy ra:

* **Mutual Exclusion** Các tài nguyên không thể được chia sẻ giữa các thread. Mỗi tài nguyên chỉ có thể được sử dụng bởi một thread tại một thời điểm.
* **Hold and Wait**: Một thread giữ ít nhất một tài nguyên và đang chờ tài nguyên khác mà hiện tại đang bị thread khác giữ.
* **No Preemption**: Các tài nguyên không thể bị cưỡng chế từ tay thread đang giữ tài nguyên. Chúng chỉ có thể được giải phóng khi thread tự nguyện trả lại tài nguyên.
* **Circular Wait**: Các thread tạo thành một vòng tròn chờ tài nguyên. Ví dụ, thread A chờ tài nguyên mà thread B đang giữ, thread B chờ tài nguyên mà thread C đang giữ, và thread C lại chờ tài nguyên mà thread A đang giữ.

**Các xử lý deadlock**:

* Một trong những cách hiệu quả nhất để tránh deadlock là không đồng bộ hóa tài nguyên một cách không cần thiết. Khi đồng bộ hóa quá mức, các thread có thể dễ dàng chặn nhau trong quá trình chờ tài nguyên, làm tăng khả năng xảy ra deadlock. Do đó, chỉ nên sử dụng đồng bộ hóa khi cần thiết, chẳng hạn như khi một tài nguyên được chia sẻ giữa nhiều thread và yêu cầu đồng bộ để đảm bảo tính toàn vẹn của dữ liệu.
* Khi nhiều thread cần truy cập vào nhiều tài nguyên cùng lúc, deadlock có thể xảy ra nếu các thread yêu cầu tài nguyên theo các thứ tự khác nhau. Để tránh điều này, bạn có thể thiết lập một thứ tự yêu cầu tài nguyên cho tất cả các thread, đảm bảo rằng tất cả các thread luôn yêu cầu tài nguyên theo cùng một thứ tự. Điều này sẽ ngăn chặn chu trình tuần hoàn trong việc yêu cầu tài nguyên, một trong những nguyên nhân chính gây ra deadlock.
* Để đảm bảo rằng thread không bị chặn quá lâu và tránh việc bị deadlock, bạn có thể sử dụng cơ chế timeout khi một thread chờ tài nguyên. Nếu một thread không thể có được tài nguyên trong thời gian nhất định, nó sẽ dừng chờ và có thể thực hiện hành động thay thế, hoặc thực hiện một chiến lược khác để xử lý sự cố. Việc sử dụng timeout giúp giảm thiểu tình trạng các thread bị chặn vô thời hạn.
* Một phương pháp khác để xử lý deadlock là sử dụng các cơ chế đồng bộ hóa linh hoạt hơn, chẳng hạn như ReentrantLock. Khác với việc đồng bộ hóa với từ khóa synchronized, ReentrantLock cho phép thread có thể tước quyền tạm thời (interruptible lock), nghĩa là khi một thread bị chặn, nó có thể bị gián đoạn và từ bỏ quyền truy cập tài nguyên nếu cần. Điều này giúp cải thiện khả năng quản lý tài nguyên và tránh các tình huống deadlock phức tạp. ReentrantLock cũng hỗ trợ tính năng tryLock(), cho phép thread thử lấy lock mà không bị chặn mãi nếu lock không sẵn sàng.

**f. Timer, Time Task, Schedule**

**Timer** và **TimerTask** là hai lớp trong Java được sử dụng để lên lịch thực hiện các tác vụ (tasks) trong tương lai, với độ chính xác cao. Chúng chủ yếu được sử dụng khi bạn cần thực hiện các tác vụ theo lịch trình định kỳ hoặc sau một khoảng thời gian nhất định.

* **Timer**: Lớp này cung cấp một phương thức để lên lịch thực hiện các tác vụ. Nó có thể thực thi một tác vụ sau một khoảng thời gian cố định hoặc định kỳ.
* **TimerTask**: Lớp này là một lớp trừu tượng đại diện cho một tác vụ mà bạn muốn thực thi. Bạn cần kế thừa lớp này và ghi đè phương thức run() để định nghĩa tác vụ.

**ScheduledExecutorService** là một giải pháp thay thế mạnh mẽ và linh hoạt hơn cho **Timer** và **TimerTask** trong Java. Nó cung cấp khả năng lên lịch các tác vụ để thực thi trong tương lai, với khả năng quản lý nhiều tác vụ đồng thời một cách dễ dàng và an toàn.

**Ưu điểm của ScheduledExecutorService**:

* Có thể xử lý các tác vụ lặp lại hoặc một lần với độ chính xác cao.
* Hỗ trợ các tác vụ đồng thời với nhiều thread.
* Đảm bảo các tác vụ không bị bỏ lỡ khi hệ thống gặp lỗi hoặc tác vụ trước đó mất nhiều thời gian hơn dự tính.

**Khái niệm ScheduledExecutorService**

* **ScheduledExecutorService** là một dịch vụ thực thi các tác vụ định kỳ, giúp bạn dễ dàng lên lịch và thực thi các tác vụ theo yêu cầu.
* Dịch vụ này thay thế **Timer** và **TimerTask**, giúp xử lý tốt hơn các vấn đề liên quan đến đồng bộ hóa và quản lý nhiều tác vụ

## 3. Lập trình mạng

### a. socket

Socket là một điểm cuối (endpoint) trong liên kết truyền thông hai chiều (two-way communication) và đại diện cho kết nối giữa Server và Client trong lập trình mạng. Các đối tượng Socket được ràng buộc với một cổng port cụ thể, để cho phép các tầng TCP (TCP Layer) có thể xác định ứng dụng mà dữ liệu sẽ được gửi đến.

Socket có khả năng hỗ trợ đa nền tảng, người dùng có thể sử dụng nhiều Socket cùng một lúc, cho phép tăng hiệu suất làm việc và tiết kiệm thời gian

Phương thức hoạt động của Socket:

điểm cuối vận hành như một cơ chế kết nối giữa server và client thông qua giao thức UDP hoặc TCP/IP để truyền và nhận dữ liệu qua internet. Để Socket hoạt động, hai ứng dụng cần đáp ứng những điều kiện cơ bản sau:

* Hai ứng dụng có thể nằm trên cùng một máy tính hoặc trên hai máy tính khác nhau.
* Trong trường hợp hai ứng dụng cùng nằm trên một máy tính, số hiệu cổng của cả hai không được trùng nhau

Socket cho phép hai ứng dụng thiết lập kết nối và truyền thông tin qua mạng. Mỗi Socket chỉ được liên kết với **một địa chỉ IP**(một thiết có một địa chỉ Ip định danh), một socket chỉ được gắn với một port, có thể tạo ra nhiều socket bằng nhiều cổng khác nhau

Xử lý nhiều máy khách: Trong khi ServerSocket có thể chấp nhận một kết nối máy khách tại một thời điểm, nó có thể xử lý nhiều máy khách bằng cách tạo một Socket mới bằng nhiều port cho mỗi máy khách. Các ứng dụng máy chủ thường sử dụng luồng hoặc nhóm luồng để xử lý nhiều máy khách đồng thời.

Server Socket:

Lắng nghe kết nối: ServerSocket lắng nghe trên một cổng cụ thể để nhận các yêu cầu kết nối đến từ máy khách. Khi máy khách cố gắng kết nối, socket máy chủ chấp nhận kết nối và tạo một đối tượng Socket mới biểu diễn kênh giao tiếp giữa máy chủ và máy khách.

Hành vi chặn: Phương thức accept() của ServerSocket đang chặn, nghĩa là nó chờ máy khách kết nối. Sau khi máy khách kết nối, máy chủ có thể tiếp tục giao tiếp.

Liên kết cổng: ServerSocket được liên kết với một cổng cụ thể trên máy chủ. Chỉ một ứng dụng có thể liên kết với một cổng cụ thể tại một thời điểm.

### b. Giao thức TCP (Stream socket)

TCP là giao thức truyền tải hướng kết nối (connection-oriented), nghĩa là phải thực hiện thiết lập kết nối với đầu xa trước khi thực hiện truyền dữ liệu. Tiến trình thiết lập kết nối ở TCP được gọi là tiến trình bắt tay 3 bước (threeway handshake).

**Cơ chế:**

**Kết nối 3 bước (Three-way handshake):** Trước khi dữ liệu được gửi, TCP phải thiết lập kết nối giữa client và server qua một quá trình gọi là "three-way handshake". Quá trình này giúp xác định rằng cả hai bên đều sẵn sàng gửi và nhận dữ liệu.

* + **SYN:** Client gửi yêu cầu kết nối.
  + **SYN-ACK:** Server phản hồi với xác nhận kết nối.
  + **ACK:** Client gửi xác nhận kết nối thành công.

**Dữ liệu được chia nhỏ:** Dữ liệu gửi đi qua TCP được chia thành các gói tin nhỏ, mỗi gói có số thứ tự để đảm bảo dữ liệu được ghép lại đúng thứ tự.

**Đảm bảo dữ liệu đến đúng nơi:** TCP sử dụng cơ chế kiểm tra và xác nhận (acknowledgement) để đảm bảo các gói tin được gửi thành công, và nếu có gói tin nào mất, nó sẽ được gửi lại.

**Kiểm soát lưu lượng và điều chỉnh:** TCP sử dụng cơ chế kiểm soát lưu lượng (flow control) để đảm bảo rằng tốc độ gửi dữ liệu không vượt quá khả năng của người nhận.

**Ví dụ:**

* **Truy cập web (HTTP/HTTPS):** Khi mở trình duyệt và truy cập một website, TCP đảm bảo rằng các dữ liệu như trang web, hình ảnh, video... được gửi từ server tới máy của bạn một cách chính xác và đầy đủ.
* **Gửi email (SMTP, IMAP, POP3):** Các giao thức này sử dụng TCP để đảm bảo rằng các email được gửi và nhận một cách chính xác.

### c. UDP (Datagram Sockets):

UDP là một giao thức không kết nối và không đảm bảo tính đáng tin cậy trong việc truyền tải dữ liệu. Dữ liệu được gửi mà không có sự xác nhận hoặc kiểm tra việc mất gói. Vì vậy, UDP có độ trễ thấp và thích hợp cho các ứng dụng yêu cầu tốc độ cao mà không cần quá nhiều độ chính xác, ví dụ như streaming video, trò chơi trực tuyến.

Khi tạo một DatagramSocket mà không chỉ định cổng, Java sẽ tự động chọn một cổng tạm thờ

**Cơ chế:**

1. **Không có kết nối:** UDP không thiết lập kết nối trước khi gửi dữ liệu, dữ liệu được gửi ngay lập tức từ sender đến receiver mà không có bất kỳ sự xác nhận nào.
2. **Không có kiểm tra lỗi:** UDP không kiểm tra xem gói tin có đến nơi hay không, nếu mất gói tin, nó không gửi lại gói tin đó.
3. **Không đảm bảo thứ tự:** Các gói tin có thể đến đích không theo thứ tự, và người nhận cần xử lý các gói tin bị mất hoặc bị sai thứ tự nếu cần.

**Ví dụ thực tế:** **Streaming video** ,**Trò chơi trực tuyến**

d. So sánh TCP và UDP:

|  |  |  |
| --- | --- | --- |
| **Tiêu chí** | **TCP (Transmission Control Protocol)** | **UDP (User Datagram Protocol)** |
| **Kết nối** | Có kết nối (Connection-oriented), yêu cầu thiết lập kết nối trước khi gửi dữ liệu. | Không có kết nối (Connectionless), gửi dữ liệu ngay lập tức mà không cần thiết lập kết nối. |
| **Độ tin cậy** | Đảm bảo dữ liệu đến đúng nơi và đúng thứ tự, có cơ chế xác nhận và retransmission. | Không đảm bảo độ tin cậy, không kiểm tra lỗi hoặc retransmission. |
| **Tốc độ** | Chậm hơn, do phải xác nhận và kiểm soát lưu lượng, có overhead. | Nhanh hơn, không cần xác nhận hoặc kiểm tra lỗi, ít overhead. |
| **Kiểm soát lưu lượng** | Có, giúp điều chỉnh tốc độ truyền tải, tránh tắc nghẽn. | Không có kiểm soát lưu lượng. |
| **Đảm bảo thứ tự** | Đảm bảo thứ tự của các gói tin, không có sự thay đổi về thứ tự dữ liệu. | Không đảm bảo thứ tự của các gói tin, gói tin có thể đến ngoài thứ tự. |
| **Ứng dụng** | Ứng dụng cần độ tin cậy cao như web (HTTP/HTTPS), email (SMTP, IMAP), FTP, truyền tệp lớn. | Ứng dụng cần tốc độ cao, yêu cầu ít độ trễ như video streaming (YouTube, Netflix), trò chơi online, Voice over IP (VoIP), DNS. |
| **Quản lý lỗi** | Có cơ chế kiểm tra và xử lý lỗi (checksum, retransmission). | Không có cơ chế kiểm tra lỗi hoặc xử lý lỗi. |
| **Đảm bảo tốc độ truyền tải** | Không luôn tối ưu cho tốc độ, vì phải đợi xác nhận và kiểm tra lỗi. | Tối ưu cho tốc độ, có thể mất một số dữ liệu nhưng không bị gián đoạn quá nhiều. |
| **Giao thức truyền tải** | Sử dụng giao thức truyền tải an toàn với các kiểm tra toàn vẹn (error detection) và điều khiển luồng (flow control). | Không có cơ chế kiểm tra toàn vẹn, chỉ tập trung vào việc gửi dữ liệu nhanh chóng. |
| **Tính linh hoạt** | Phù hợp với các ứng dụng yêu cầu chính xác và an toàn, như giao dịch tài chính hoặc các dịch vụ quan trọng. | Phù hợp cho các ứng dụng yêu cầu tính linh hoạt và tốc độ, chẳng hạn như live stream, games hoặc dịch vụ VoIP. |
| **Quản lý phiên làm việc** | TCP duy trì trạng thái kết nối, giúp quản lý các phiên giao dịch. | UDP không duy trì trạng thái kết nối, mỗi gói dữ liệu được xử lý độc lập. |
| **Khả năng chịu lỗi** | Chịu lỗi tốt, có khả năng phục hồi sau khi xảy ra sự cố (recovery). | Chịu lỗi kém hơn, không có cơ chế phục hồi khi dữ liệu bị mất. |
| **Quản lý băng thông** | Tối ưu băng thông bằng cách điều chỉnh tốc độ truyền tải và phản hồi từ phía người nhận. | Không quản lý băng thông, gửi dữ liệu với tốc độ tối đa có thể. |
| **Sử dụng cổng** | TCP sử dụng cổng để thiết lập và duy trì kết nối giữa các thiết bị. | UDP cũng sử dụng cổng, nhưng không cần duy trì kết nối. |
| **Đảm bảo hoàn thành nhiệm vụ** | Nếu dữ liệu không đến đích, TCP sẽ tự động gửi lại. | Nếu dữ liệu bị mất, UDP không cố gắng gửi lại. |
| **Sử dụng trong Streaming** | Không được sử dụng trong các ứng dụng streaming vì độ trễ cao và yêu cầu độ chính xác cao. | Phù hợp với các ứng dụng streaming vì tốc độ truyền tải nhanh và ít độ trễ. |

## 4. MultiThread

### a. MultiThreading

Là việc sử dụng nhiều thread trong một chương trình để thực hiện các tác vụ đồng thời.

### b. Queue

Queue là một cấu trúc dữ liệu theo nguyên lý "FIFO" (First In First Out), tức là phần tử được thêm vào đầu tiên sẽ được lấy ra đầu tiên. Queue thường được sử dụng để quản lý các tác vụ cần thực hiện tuần tự, như trong hệ thống các tác vụ chờ đợi, gửi/nhận tin nhắn, hay xử lý các job trong môi trường đa luồng.

Các loại Queue phổ biến:

* BlockingQueue: Đây là loại Queue mà khi không có phần tử nào trong Queue, một thread sẽ phải chờ đợi (block) cho đến khi có phần tử được thêm vào.
* PriorityQueue: Queue này cho phép các phần tử có thứ tự ưu tiên cao hơn được xử lý trước.
* ConcurrentQueue: Là Queue được thiết kế để làm việc với nhiều thread mà không gặp phải vấn đề về đồng bộ hóa.

Cách làm việc với Queue:

* Thêm phần tử vào Queue (Enqueue): Sử dụng các phương thức như add(), offer() để thêm phần tử vào Queue.
* Lấy phần tử từ Queue (Dequeue): Sử dụng các phương thức như remove(), poll() để lấy phần tử ra khỏi Queue.
* Xem phần tử đầu tiên trong Queue: Sử dụng peek() để kiểm tra phần tử đầu tiên mà không loại bỏ nó.

### c. Tìm hiểu và làm ví dụ với Thread Pool

Thread Pool là một mô hình trong lập trình đa luồng, nơi một số lượng nhất định các thread được tạo sẵn và sẵn sàng xử lý các tác vụ khi chúng đến. Khi một tác vụ (job) cần được xử lý, nó sẽ được gửi đến thread trong pool, thay vì phải tạo mới một thread mỗi khi cần xử lý một tác vụ, giúp tiết kiệm tài nguyên và giảm overhead.

Cách sử dụng Thread Pool:

* Tạo một pool các thread: Thường xuyên sử dụng các thư viện như concurrent.futures trong Python hoặc ThreadPoolExecutor trong Java để tạo thread pool.
* Gửi các tác vụ (tasks) vào pool: Các tác vụ này có thể là các hàm hoặc đối tượng callable.
* Quản lý thread và kết quả: Thread pool sẽ xử lý các tác vụ này theo cách hiệu quả và khi hoàn thành, có thể lấy kết quả từ các tác vụ.

Ưu điểm:

* Giảm thiểu chi phí tạo thread: Không cần phải tạo mới thread cho mỗi tác vụ.
* Quản lý tốt tài nguyên: Thread pool giúp quản lý số lượng thread tối ưu, tránh tạo quá nhiều thread dẫn đến lãng phí tài nguyên.
* Xử lý đồng thời hiệu quả: Các tác vụ được phân phối và xử lý nhanh chóng mà không phải chờ đợi.

Nhược điểm:

* Giới hạn số lượng thread: Số lượng thread trong pool bị giới hạn, nếu không đủ thread, các tác vụ mới sẽ phải chờ.
* Quản lý phức tạp: Cần phải quản lý tốt việc tái sử dụng các thread để tránh tình trạng tắc nghẽn.

**d.Thế nào là đồng bộ và bất đồng bộ? Làm thế nào để đồng bộ khi nhiều thread cùng sử dụng một tài nguyên? Các vấn đề cần xử lý khi chương trình có nhiều thread?**

**d. Synchronous and Asynchronous**

**Race condition** xảy ra khi hai hoặc nhiều thread cố gắng truy cập và thay đổi dữ liệu chung mà không có sự đồng bộ, dẫn đến kết quả không mong muốn hoặc sai lệch.

**Đồng bộ (Synchronous):**

Trong lập trình đồng bộ, các tác vụ thực thi tuần tự, nghĩa là mỗi tác vụ phải hoàn thành trước khi tác vụ tiếp theo bắt đầu. Điều này có thể gây chậm trễ trong trường hợp có các tác vụ tốn thời gian (ví dụ như I/O hoặc tải dữ liệu từ mạng). Đồng bộ là mô hình phổ biến trong lập trình đơn giản vì nó dễ hiểu và dễ kiểm soát.

Ví dụ: Khi một chương trình đọc dữ liệu từ tệp, chương trình sẽ chờ đợi quá trình đọc dữ liệu hoàn tất trước khi tiếp tục thực hiện các tác vụ khác.

**Bất đồng bộ (Asynchronous):**

Trong lập trình bất đồng bộ, các tác vụ có thể được thực thi độc lập mà không cần phải đợi tác vụ trước đó hoàn thành. Mỗi tác vụ sẽ được gửi cho hệ thống quản lý để thực hiện và sẽ nhận được kết quả khi hoàn thành, mà không cần phải chờ đợi.

Ví dụ: Trong một ứng dụng web, khi người dùng nhấn nút tải trang, trình duyệt có thể tiếp tục xử lý các tác vụ khác như tải ảnh hoặc thông báo mà không cần phải chờ đợi tất cả dữ liệu được tải về từ server.

**Lợi ích của lập trình bất đồng bộ:**

Hiệu suất cao hơn: Trong các tác vụ I/O như đọc tệp, gửi email, hay gửi yêu cầu HTTP, bất đồng bộ giúp tiết kiệm thời gian và tài nguyên vì chương trình không cần chờ đợi các tác vụ hoàn tất.

Giảm độ trễ: Các ứng dụng yêu cầu phản hồi nhanh, chẳng hạn như trò chơi trực tuyến hoặc hệ thống xử lý thời gian thực, sẽ hoạt động hiệu quả hơn khi sử dụng mô hình bất đồng bộ.

### d. Đồng bộ khi nhiều thread cùng sử dụng một tài nguyên:

Khi nhiều thread cùng truy cập vào tài nguyên chung (ví dụ như một biến toàn cục hoặc cơ sở dữ liệu), vấn đề race condition có thể xảy ra. Race condition là một tình huống trong đó các thread tương tác với tài nguyên chung một cách không đồng bộ, dẫn đến kết quả sai. Ví dụ, nếu hai thread cùng thay đổi giá trị của một biến mà không có cơ chế bảo vệ, thì kết quả cuối cùng có thể không đúng.

**Cách đồng bộ khi nhiều thread sử dụng tài nguyên chung:**

* **Synchronize (Đồng bộ hóa):**

Đồng bộ hóa là cơ chế dùng để bảo vệ các phần của mã mà có thể bị truy cập đồng thời bởi nhiều thread. Một phương pháp phổ biến trong Java là sử dụng từ khóa synchronized. Khi một phương thức hoặc khối mã được đồng bộ hóa, chỉ có một thread tại một thời điểm có thể thực thi đoạn mã đó.

* **Lock (Khóa):**

Trong Python, cơ chế Lock giúp đồng bộ hóa các thread. Khóa (lock) đảm bảo rằng chỉ một thread có thể truy cập tài nguyên trong một khoảng thời gian nhất định. Sau khi một thread lấy khóa, các thread khác phải đợi đến khi khóa được giải phóng.

* **Cơ chế Wait và Notify:**

Wait và Notify là cơ chế đồng bộ trong Java giúp điều phối nhiều thread khi làm việc với tài nguyên chung. Một thread có thể vào trạng thái "chờ" (waiting) khi không thể tiếp tục do thiếu tài nguyên, và các thread khác có thể thông báo (notify) cho các thread đang chờ đợi rằng tài nguyên đã sẵn sàng.

Các vấn đề cần xử lý khi chương trình có nhiều thread

* **Race Condition (Điều kiện đua):**

Vấn đề: Khi nhiều thread cùng truy cập tài nguyên chung mà không có cơ chế đồng bộ, kết quả cuối cùng có thể bị lỗi.

Giải pháp: Sử dụng cơ chế đồng bộ hóa như synchronized, Lock, hoặc Semaphore để bảo vệ tài nguyên chung khỏi sự truy cập đồng thời.

* **Deadlock (Tắc nghẽn):**

Vấn đề: Deadlock xảy ra khi hai hoặc nhiều thread bị mắc kẹt trong một tình huống mà chúng phải chờ đợi nhau giải phóng tài nguyên. Điều này khiến các thread không thể tiến hành và chương trình bị treo.

Giải pháp: Tránh deadlock bằng cách sử dụng thứ tự nhất quán trong việc yêu cầu khóa, tránh việc giữ nhiều khóa trong cùng một thời gian, hoặc sử dụng timeout khi yêu cầu khóa.

* **Starvation (Đói tài nguyên):**

Vấn đề: Một hoặc nhiều thread không có cơ hội truy cập tài nguyên vì các thread khác luôn chiếm tài nguyên đó.

Giải pháp: Cung cấp cơ chế quản lý ưu tiên cho các thread, hoặc sử dụng các thuật toán chia sẻ tài nguyên công bằng (fair scheduling).

* **Thread Safety (An toàn cho thread):**

Vấn đề: Khi nhiều thread chia sẻ cùng một bộ dữ liệu hoặc tài nguyên, cần đảm bảo rằng dữ liệu không bị thay đổi một cách không kiểm soát, dẫn đến lỗi hoặc sự không nhất quán.

Giải pháp: Sử dụng các lớp hoặc cấu trúc dữ liệu an toàn cho thread, như ConcurrentHashMap trong Java hoặc threading.Lock trong Python.

* **Quản lý tài nguyên:**

Vấn đề: Khi có nhiều thread, mỗi thread có thể yêu cầu tài nguyên hệ thống (như bộ nhớ hoặc CPU), dẫn đến việc sử dụng tài nguyên không hiệu quả.

Giải pháp: Sử dụng Thread Pool để quản lý số lượng thread tối đa có thể tạo ra và sử dụng tài nguyên hiệu quả. Thread Pool giúp tái sử dụng các thread đã hoàn thành công việc, thay vì tạo mới thread cho mỗi tác vụ.

5. Logging

logging giúp ghi lại thông tin quan trọng về hoạt động của chương trình để người phát triển có thể theo dõi, gỡ lỗi, giám sát quá trình hệ thống, hỗ trợ phân tích và tối ưu hoá và bảo trì ứng dụng dễ dàng hơn. Logger sẽ cung cấp các thông tin về trạng thái của chương trình, lỗi, hoặc các sự kiện quan trọng.