viettel
software services

CHƯƠNG TRÌNH ĐÀO TẠO JAVA NÂNG CAO

Xử lý đa luồng

Đơn vị tổ chức: Phòng đào tạo và phát triển nguồn nhân lực

viettel
software services

NỘI DUNG HỌC PHẦN

BÀI 1

Xử lý đa luồng

BÀI 2

Xử lý bất đồng bộ

viettel
software services

BÀI 1

Xử lý đa luồng trong Java

viettel
software services

MỤC LỤC

viettel
software services

01

Xử lý đa luồng trong Java

viettel
software services

Xử lý đa luồng trong Java

- Đa luồng hay còn được gọi là Multithreading. Một chương trình đa luồng luôn có 2 tiến trình trở lên chạy song song nhau, mỗi tiến trình đó người ta gọi là một luồng (thread).
- Luồng(thread) là đơn vị nhỏ nhất trong java có thể thực hiện được 1 công việc riêng biệt và các luồng được quản lý bởi máy ảo java (Java Virtual Machine - JVM).
- Một ứng dụng java ngoài luồng chính có thể có các luồng khác thực thi đồng thời. Đa luồng giúp cho các tác vụ được xử lý độc lập giúp công việc được hoàn thành nhanh chóng. Vậy đa luồng có thể hiểu đơn giản là quá trình xử lý nhiều luồng song song nhau và thực hiện các nhiệm vụ khác nhau cùng một lúc.

viettel
software services

Tạo và quản lý đa luồng trong Java

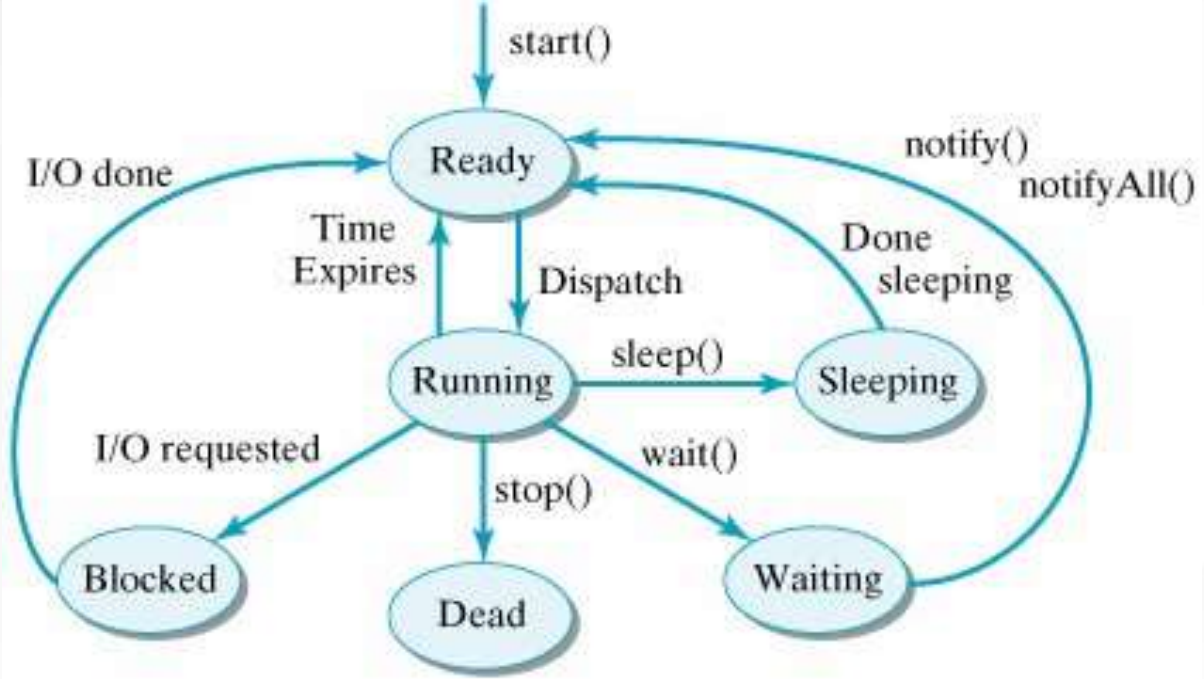
- Ngôn ngữ java cung cấp cho ta đối tượng Thread nhằm thể hiện cơ chế đa luồng. Có hai cách chính để tạo luồng đó là tạo 1 đối tượng của lớp được thừa kế từ lớp Thread hoặc implements từ giao diện Runnable

Sự khác biệt giữa class Thread và Interface Runnable

- Nếu chúng ta cho một class extend lớp Thread, class đó sẽ không thể extend bất kỳ class nào khác vì Java không hỗ trợ đa kế thừa. Nhưng, nếu chúng ta implement giao diện Runnable, lớp của chúng ta vẫn có thể mở rộng các class khác.
- Chúng ta sẽ không phải implement lại một số chức năng cơ bản của luồng nếu chúng ta extend class Thread, vì Thread đã cung cấp sẵn cho chúng ta một số phương thức như field (), interrupt (), v.v... Các phương thức này không có sẵn trong giao diện Runnable.

viettel
software services

Vòng đời của 1 Thread



viettel
software services

Vòng đời của 1 Thread

- **New:** Đây là trạng thái đầu tiên của thread khi nó bắt đầu được khởi tạo (Khởi tạo nằng lệnh new() ấy). Và ở trạng thái này thread chưa được chạy, cũng chưa được lập lịch để chạy.
- **Runnable:** Đây là trạng thái mà thread sẵn sàng và chờ được chạy vì nó đang được lập lịch để chạy. Thread sẽ mang trạng thái này khi mà lần đầu tiên phương thức start() của nó được triệu gọi. và nó sẽ mang trạng thái này nhiều lần nữa khi chuyển từ các trạng thái running, waiting, sleeping, blocking sang.
- **Running:** Đây là trạng thái mà lập trình viên luôn luôn mong muốn khi bắt đầu ra lệnh start() với nó. Ở trạng thái này các lệnh ở trong phương thức run() của thread đươc triệu gọi. Và thread chỉ có trạng thái này khi trước đó nó ở trạng thái ready.

viettel
software services

Vòng đời của 1 Thread

- **Các trạng thái Nonrunnable states:** Là các trạng thái tạm dừng và chưa đủ điều kiện về logic để chạy. Thread muốn được chạy thì phải thỏa mãn một điều kiên nào đó (tùy từng trạng thái) và chuyển sang trạng thái read mới có thể tiếp tục được chạy. Có 3 trạng thái như sau:

    - **Blocked:** đây là trạng thái mà thread phải đợi một tài nguyên (resource) của hệ thống hoặc đợi một đối tượng nhả khóa ra (Object's lock) để có thể chuyển sang trạng thái ready để tiếp tục chạy. Thông thường trong java khi truy cập vào ra một tài nguyên hệ thống thì thread tự động bị block cho tới khi chiếm được quyền truy cập.

    - **Timed Waiting:** Đây là trạng thái ngủ của thread xảy ra khi ta gọi phương thức sleep() của thread.

Vòng đời của 1 Thread

- **Waiting:** Đây là trạng thái mà một object trong thread đang chạy triệu gọi phương thức wait(). Thread ngay lập tức chuyển sang waiting và chỉ được chuyển sang trạng thái ready khi có một thread khác gọi tới phương thức notify() (của object đã gọi wait()) hoặc notifyall(). Lý do mà ta phải đưa thread sang trạng thái wait là vì chúng ta cần một thread khác phải thực hiện xong một việc của nó trước và thread hiện tại mới bắt đầu thực hiện tiếp công việc. Ví dụ như nhiều thread cùng đọc và ghi vào một đối tượng thì ta cần chờ cho các thread khác ghi vào đối tượng xong thì mới đọc, không nên cùng đọc va ghi xảy ra cùng một lúc sẽ dẫn tới sai lệch về mặt dữ liệu hoặc thậm chí xảy ra deadlock (ta sẽ nghiên cứu deadlock sau)

Vòng đời của 1 Thread

- **Terminated:** Khi một thread đã kết thúc phương thức run() thì nó sẽ chuyển về trạng thái dead. ở trạng thái này thread không thể khởi chạy lại một lần nữa, nếu ta cố gọi lại phương thức start() của nó ta sẽ nhận lại một exception. nhưng bản thân object này nó vẫn là một đối tượng đã được khởi tạo và chưa giải phóng bộ nhớ, thế nên ta vẫn có thể gọi được các phương thức khác của đối tượng thread, tuy nhiên khi gọi các phương thức của nó thì phương thức sẽ được thực thi ở trong thread gọi nó

viettel
software services

02

Ví dụ

viettel
software services

Tạo Thread với interface Runnable

```
class RunnableDemo implements Runnable {
    private Thread t;
    private String threadName;

    RunnableDemo( String name) {
        threadName = name;
        System.out.println("Creating " +  threadName );
    }

    public void run() {
        System.out.println("Running " +  threadName );
        try {
            for(int i = 4; i > 0; i--) {
                System.out.println("Thread: " + threadName + ", " + i);
                // Let the thread sleep for a while.
                Thread.sleep( 50);
            }
        } catch (InterruptedException e) {
            System.out.println("Thread " +  threadName + " interrupted.");
        }
        System.out.println("Thread " +  threadName + " exiting.");
    }

    public void start () {
        System.out.println("Starting " +  threadName );
        if (t == null) {
            t = new Thread ( target: this, threadName);
            t.start ();
        }
    }
}
```

viettel
software services

Tạo Thread với interface Runnable

```
public class Solution {
    public static void main(String[] args) throws IOException {
        RunnableDemo R1 = new RunnableDemo( name: "Thread-1");
        R1.start();

        RunnableDemo R2 = new RunnableDemo( name: "Thread-2");
        R2.start();
    }
}
```

viettel
software services

Tạo Thread với class Thread

```
class ThreadDemo extends Thread {
    private Thread t;
    private String threadName;

    ThreadDemo( String name) {
        threadName = name;
        System.out.println("Creating " +  threadName );
    }

    public void run() {
        System.out.println("Running " +  threadName );
        try {
            for(int i = 4; i > 0; i--) {
                System.out.println("Thread: " + threadName + ", " + i);
                // Let the thread sleep for a while.
                Thread.sleep( millis: 50);
            }
        } catch (InterruptedException e) {
            System.out.println("Thread " +  threadName + " interrupted.");
        }
        System.out.println("Thread " +  threadName + " exiting.");
    }

    public void start () {
        System.out.println("Starting " +  threadName );
        if (t == null) {
            t = new Thread ( target: this, threadName);
            t.start ();
        }
    }
}
```

viettel
software services

Tạo Thread với Runnable

```
public class Solution {
    public static void main(String[] args) throws IOException {
        ThreadDemo T1 = new ThreadDemo( name: "Thread-1");
        T1.start();

        ThreadDemo T2 = new ThreadDemo( name: "Thread-2");
        T2.start();
    }
}
```

viettel
software services

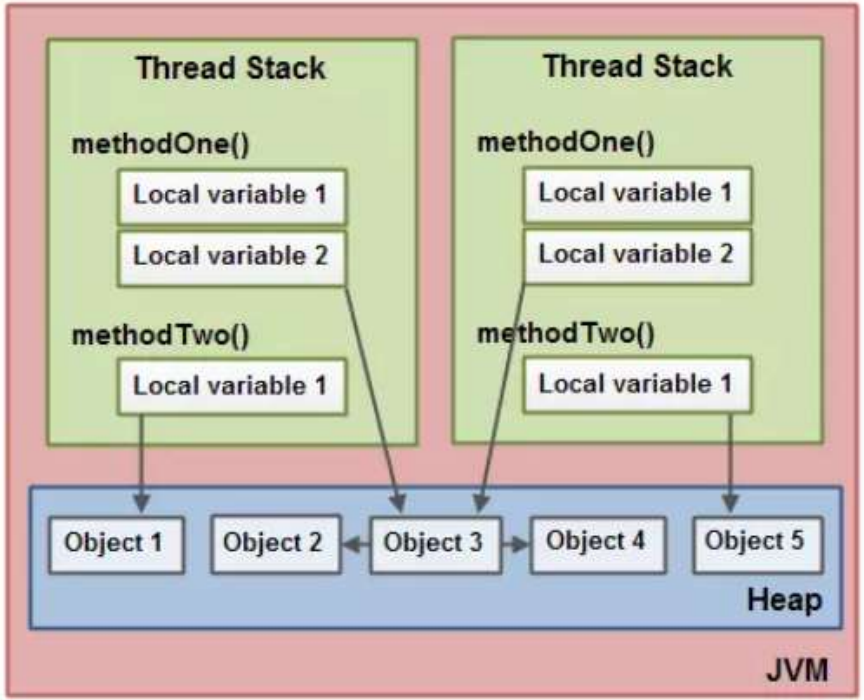
03

Bộ nhớ trong xử lý đa luồng

viettel
software services

Bộ nhớ trong xử lý đa luồng

Java Memory Model được sử dụng trong trong các JVM chia bộ nhớ thành 2 thành phần Thread stacks và Heap. Biểu đồ sau minh họa Java Memory Model từ góc độ logic:



viettel
software services

Stack trong xử lý đa luồng

- Mỗi Thread khi chạy trên JVM sẽ có một Thread stack riêng. Thread stack chứa các thông tin về các methods mà thread đã gọi khi thực thi. Khi các thread thực thi mã của nó, các stack sẽ thay đổi.

- Thread stack cũng bao gồm tất cả các local variables của mỗi method được execute (all methods on the call stack). Một thread có thể chỉ có quyền truy cập tới thread stack của nó. Local variables được tạo bởi thread chỉ available với chính Thread tạo ra nó. Ngay cả khi nếu 2 Threads cùng thực thi một đoạn code giống nhau thì 2 Threads vẫn tạo ra các local variables riêng bên trong Thread stack. Do đó, mỗi thread có một version cho các local variables.

viettel
software services

Heap trong xử lý đa luồng

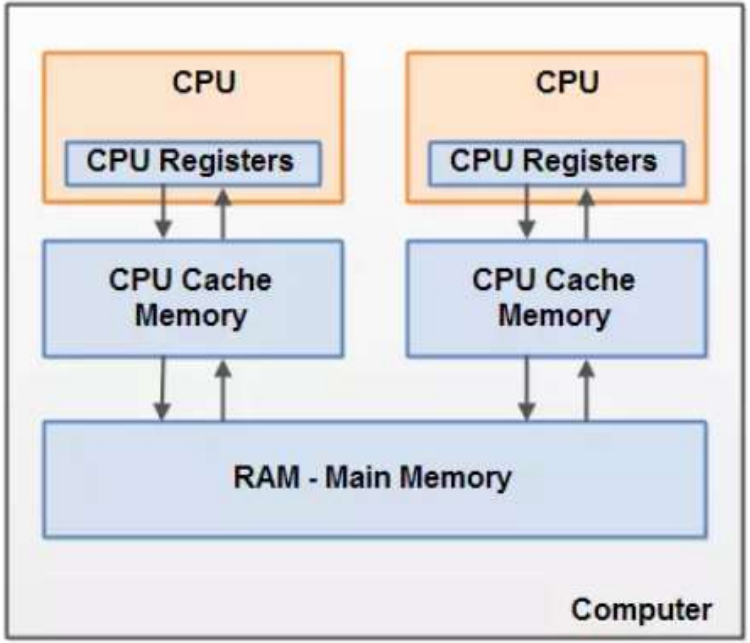
- Heap chứa tất cả các objects được tạo ra bởi Java application. Gồm có các object versions kiểu primitive (e.g. Byte, Integer, Long etc.). Không thành vấn đề nếu một object được tạo ra và gán cho một local variable, hoặc tạo ra như là một member variable của đối tượng khác, các object vẫn được lưu trên heap.
- Objects trên Heap có thể được truy cập bởi tất cả các Threads có reference đến object. Khi một thread có truy cập tới object, nó có thể có quyền truy cập tới các object's member variables. Nếu 2 threads gọi một method trên một object vào cùng một thời điểm, chúng sẽ có cả 2 quyền truy cập tới object's member variables, nhưng mỗi thread sẽ có một bản sao của local variables nghĩa là nếu một thread thay đổi giá trị của một member variable thì giá trị đó sẽ ảnh hưởng đến các threads khác truy cập đến object đó

viettel software services

Hardware Memory Architecture

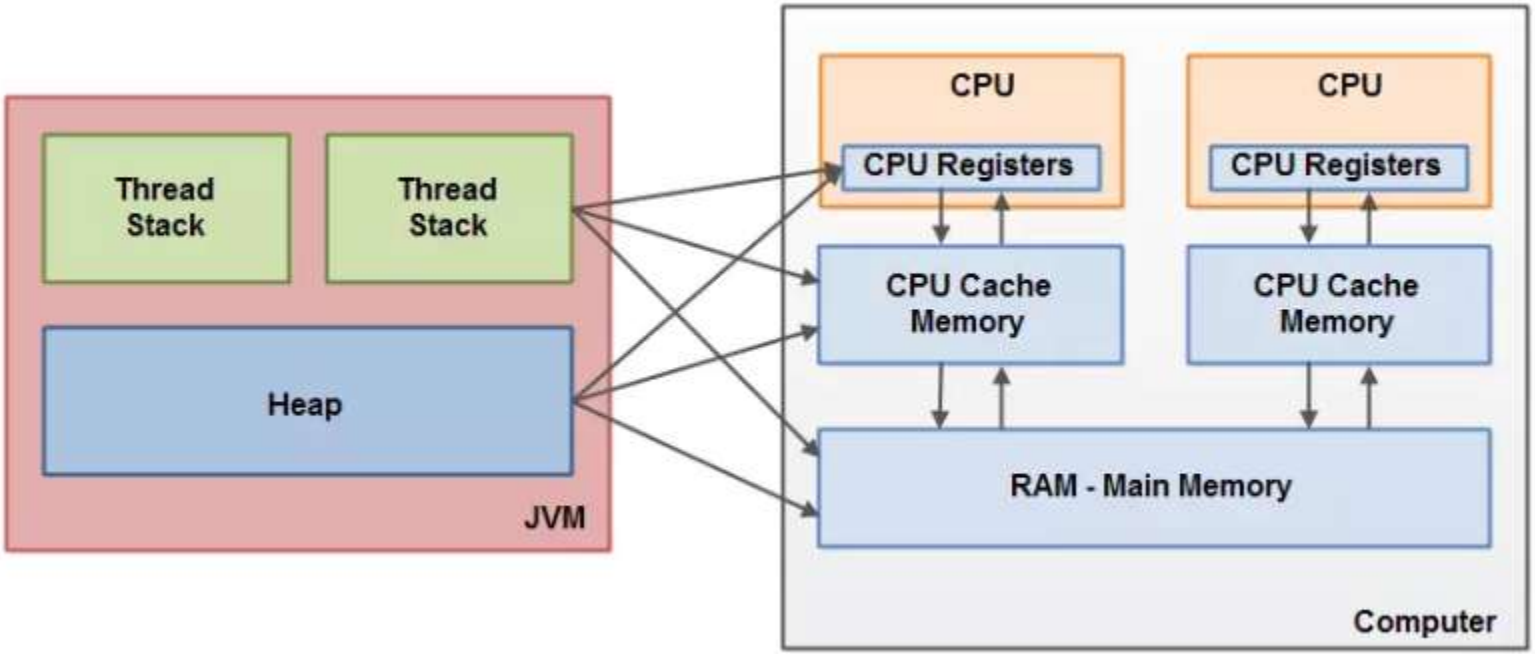
- Với mô hình trên, ta có một hình máy tính với 2 CPU, mỗi CPU có khả năng chạy một thread trong một thời điểm, mỗi CPU có chứa một tập thanh ghi register (thành phần tối thiểu trong CPU-memory), nhờ vào thanh ghi, CPU có thể thực hiện tính toán nhanh hơn là sử dụng bộ nhớ chính. Ngoài ra. mỗi CPU cũng bao gồm CPU cache memory layer. Thông thường, tốc độ truy cập dữ liệu sẽ có thứ tự sau: Main memory < Cache memory < Internal registers

- Khi CPU cần truy cập main memory, nó sẽ đọc một phần main memory vào trong cache, rồi lại đọc một phần cache vào trong internal registers, sau đó mới tiến hành các tác vụ tính toán và xử lý dữ liệu.



viettel
software services

Mối liên hệ giữa Java Memory Model và Hardware Memory Architecture



Mối liên hệ giữa Java Memory Model và Hardware Memory Architecture

- Java memory model và hardware memory architecture không giống nhau. Hardware memory architecture không phân biệt giữa thread stacks và heap. Trên phương diện phần cứng, cả thread stack và heap được đặt trong main memory. Đôi khi, một phần thread stacks và heap xuất hiện trên CPU caches và internal CPU registers.

viettel
software services

BÀI 2

Xử lý bất đồng bộ

viettel
software services

MỤC LỤC

viettel
software services

01

Xử lý bất đồng bộ trong Java

viettel
software services

Xử lý bất đồng bộ trong Java

- Lập trình bất đồng bộ là cách lập trình cho phép các hoạt động thực hiện không theo tuần tự.
- Có thể các đoạn code ở dưới chạy trước đoạn code viết ở phía trên(bất đồng bộ), các hoạt động không phải đợi lẫn nhau.
- Bởi vì không bị block thread và các hoạt động có thể không phải đợi nhau nên khi xử lý các tác vụ có thời gian thực hiện lâu không bị đứng chương trình, đem lại trải nghiệm người dùng tốt hơn.

viettel
software services

Sử dụng CompletableFuture

- **CompletableFuture** được sử dụng cho lập trình bất đồng bộ trong Java.
- **CompletableFuture** là kết quả trả về của phép tính / method không đồng bộ, cho phép kiểm tra trạng thái của phép tính (đã thực hiện xong chưa, kết quả trả về là gì…), bắt sự kiện khi method hoàn thành…
- **CompletableFuture** có từ phiên bản Java 8 có khoảng 50 phương thức để kết hợp, thực hiện các tính toán bất đồng bộ và xử lý lỗi

viettel software services

Sử dụng CompletableFuture

```java
import java.util.concurrent.*;

class Calculator {
    public static int add(int a, int b) {
        int sum = a + b;
        System.out.println("result: " + a + " + " + b + " = " + sum);
        return sum;
    }
}


public class Demo {

    public static void main(String[] args) throws InterruptedException, ExecutionException {
        // Create a CompletableFuture
        CompletableFuture<Integer> future1 = CompletableFuture.supplyAsync(() -> Calculator.add( a: 1, b: 2));
        CompletableFuture<Integer> future2 = CompletableFuture.supplyAsync(() -> Calculator.add( a: 1, b: 3));
        CompletableFuture<Integer> future3 = CompletableFuture.supplyAsync(() -> Calculator.add( a: 2, b: 3));

        System.out.println("Done");

    }
}
```

viettel
software services

Xử lý kết quả

- Với CompletableFuture ta có thể bắt được sự kiện khi việc tính toán hoàn thành
- **thenRun** là thực hiện làm gì khi CompletableFuture hoàn thành (không cần quan tâm kết quả là gì).
- **thenAccept** là xử lý kết quả khi CompletableFuture hoàn thành.
- **handle** dùng xử lý kết quả hoặc lỗi khi CompletableFuture hoàn thành.

viettel
software services

Xử lý kết quả

```
future1.thenRun(()->{
    System.out.println("future1 completed!");
});
future2.thenAccept(result ->{
    System.out.println("future2 completed, result = " + result);
});
future3.handle((data, error) ->{
    if (error != null){
        System.out.println("future3 error, error: " + error);
        return null;
    } else {
        System.out.println("future3 completed, result = " + data);
        return data;
    }

});
```

viettel
software services

Kết hợp nhiều CompletableFuture

- Trường hợp muốn bắt sự kiện tất cả các CompletableFuture hoàn thành thì ta dùng method allOf()

```
CompletableFuture<Void> futureAll = CompletableFuture.allOf(future1, future2, future3);
futureAll.thenRunAsync(()->{
    System.out.println("All future is Done!");
});
```

viettel
software services

Xử lý bất đồng bộ thích hợp cho các hoạt động I/0:

02

Non-Blocking IO

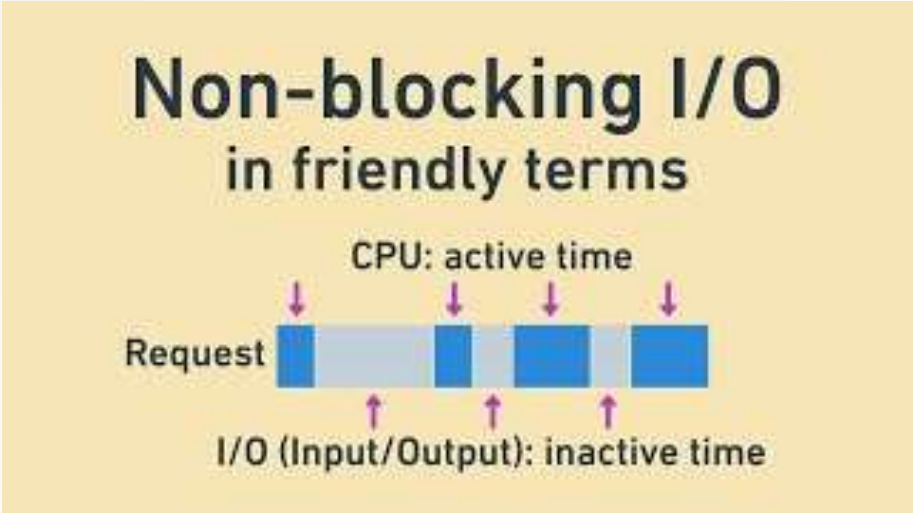
viettel
software services

Non-Blocking IO

- IO chức là input/output, tức bất kì thao tác đọc/ghi nào tới hệ thống. IO là thao tác tốn thời gian và nó sẽ block tất cả các hàm khác đang chạy. IO có nghĩa dữ kiện chạy đi chạy lại từ một nơi này đến một nơi khác như từ: bộ nhớ vào ổ cứng hay ngược lại, server đến client và ngược lại....
- Non-Blocking I/O, một cách đơn giản, có nghĩa những công việc về input output mà nó không ngăn chặn những công việc khác. Trong các loại công việc của máy tính, IO là chậm nhất. Chậm hơn rất nhiều so với tính toán. Vì lý do đó nếu IO được điều hành làm sao mà không làm trễ những công việc khác của máy tính thì cả hệ thống tính toán sẽ bớt trì trệ, hay nói cách khác là nhanh hơn
- Để sử dụng được Non-Blocking IO cần sử dụng các hàm bất đồng bộ khi thực các thao tác với IO

viettel
software services

Non-Blocking IO



viettel
software services

Xử lý đa luồng trong Java

- **Khái niệm:** Một chương trình đa luồng luôn có 2 tiến trình trở lên chạy song song nhau, mỗi tiến trình đó người ta gọi là một luồng (thread).
- Cách sử dụng **Runable và Thread**
- **Bộ nhớ trong xử lý đa luồng:** bộ nhớ trong xử lý đa luồng Heap và Stack, bộ nhớ vật lý trong xử lý đa luồng

viettel
software services

Xử lý bất đồng bộ

- **Khái niệm:** Lập trình bất đồng bộ là cách lập trình cho phép các hoạt động thực hiện không theo tuần tự.
- Cách sử dụng **CompletableFuture** và các hàm cơ bản của **CompletableFuture**
- **Ứng dụng của xử lý đa luồng:** dung cho các trường hợp liên quan đến IO
- Khái niệm về **Non-Blocking IO**

viettel
software services

XIN CẢM ƠN