**Lập trình đa luồng trong Java (Java Multi-threading)**

*Đăng vào 12/02/2018 . Được đăng bởi*[**GP Coder**](https://gpcoder.com/author/gpadmin/)*.****61541****Lượt xem . Toàn màn hình*

**Nội dung**[[Ẩn](https://gpcoder.com/3484-lap-trinh-da-luong-trong-java-java-multi-threading/)]

* [1 Giới thiệu](https://gpcoder.com/3484-lap-trinh-da-luong-trong-java-java-multi-threading/#Gioi_thieu)
  + [1.1 Thread là gì? Multi-thread là gì?](https://gpcoder.com/3484-lap-trinh-da-luong-trong-java-java-multi-threading/#Thread_la_gi_Multi-thread_la_gi)
  + [1.2 Đa nhiệm (multitasking)](https://gpcoder.com/3484-lap-trinh-da-luong-trong-java-java-multi-threading/#Da_nhiem_multitasking)
  + [1.3 Ưu điểm của đa luồng](https://gpcoder.com/3484-lap-trinh-da-luong-trong-java-java-multi-threading/#Uu_diem_cua_da_luong)
  + [1.4 Nhược điểm của đa luồng](https://gpcoder.com/3484-lap-trinh-da-luong-trong-java-java-multi-threading/#Nhuoc_diem_cua_da_luong)
* [2 Vòng đời (các trạng thái) của một Thread trong java](https://gpcoder.com/3484-lap-trinh-da-luong-trong-java-java-multi-threading/#Vong_doi_cac_trang_thai_cua_mot_Thread_trong_java)
* [3 Cách tạo luồng trong Java](https://gpcoder.com/3484-lap-trinh-da-luong-trong-java-java-multi-threading/#Cach_tao_luong_trong_Java)
  + [3.1 Tạo luồng bằng cách extend từ lớp Thread](https://gpcoder.com/3484-lap-trinh-da-luong-trong-java-java-multi-threading/#Tao_luong_bang_cach_extend_tu_lop_Thread)
  + [3.2 Tạo luồng bằng cách implement từ Interface Runnable](https://gpcoder.com/3484-lap-trinh-da-luong-trong-java-java-multi-threading/#Tao_luong_bang_cach_implement_tu_Interface_Runnable)
  + [3.3 Khi nào implements từ interface Runnable?](https://gpcoder.com/3484-lap-trinh-da-luong-trong-java-java-multi-threading/#Khi_nao_implements_tu_interface_Runnable)
  + [3.4 Ví dụ minh họa sử dụng đa luồng](https://gpcoder.com/3484-lap-trinh-da-luong-trong-java-java-multi-threading/#Vi_du_minh_hoa_su_dung_da_luong)
    - [3.4.1 Ví dụ Tạo luồng bằng cách extend từ class Thread](https://gpcoder.com/3484-lap-trinh-da-luong-trong-java-java-multi-threading/#Vi_du_Tao_luong_bang_cach_extend_tu_class_Thread)
    - [3.4.2 Ví dụ Tạo luồng bằng cách implement từ Interface Runnable](https://gpcoder.com/3484-lap-trinh-da-luong-trong-java-java-multi-threading/#Vi_dunbspTao_luong_bang_cach_implement_tu_Interface_Runnable)
* [4 Các phương thức của lớp Thread thường hay sử dụng](https://gpcoder.com/3484-lap-trinh-da-luong-trong-java-java-multi-threading/#Cac_phuong_thuc_cua_lop_Thread_thuong_hay_su_dung)
* [5 Một số thông tin liên quan đến luồng](https://gpcoder.com/3484-lap-trinh-da-luong-trong-java-java-multi-threading/#Mot_so_thong_tin_lien_quan_den_luong)
  + [5.1 Định danh của luồng (ThreadId)](https://gpcoder.com/3484-lap-trinh-da-luong-trong-java-java-multi-threading/#Dinh_danh_cua_luong_ThreadId)
  + [5.2 Tên của luồng (ThreadName)](https://gpcoder.com/3484-lap-trinh-da-luong-trong-java-java-multi-threading/#Ten_cua_luong_ThreadName)
  + [5.3 Độ ưu tiên của luồng (Priority)](https://gpcoder.com/3484-lap-trinh-da-luong-trong-java-java-multi-threading/#Do_uu_tien_cua_luong_Priority)
  + [5.4 Ví dụ minh họa](https://gpcoder.com/3484-lap-trinh-da-luong-trong-java-java-multi-threading/#Vi_du_minh_hoa)
* [6 Sử dụng phương thức sleep()](https://gpcoder.com/3484-lap-trinh-da-luong-trong-java-java-multi-threading/#Su_dung_phuong_thuc_sleep)
* [7 Sử dụng join() và join(long)](https://gpcoder.com/3484-lap-trinh-da-luong-trong-java-java-multi-threading/#Su_dung_join_vanbspjoinlong)
* [8 Xử lý ngoại lệ cho Thread](https://gpcoder.com/3484-lap-trinh-da-luong-trong-java-java-multi-threading/#Xu_ly_ngoai_le_cho_Thread)

**Giới thiệu**

**Thread là gì? Multi-thread là gì?**

**Thread** (luồng) về cơ bản là một tiến trình con (sub-process). Một đơn vị xử lý nhỏ nhất của máy tính có thể thực hiện một công việc riêng biệt. Trong Java, các luồng được quản lý bởi máy ảo Java (JVM).

**Multi-thread** (đa luồng) là một tiến trình (process – được gọi là tiến trình) thực hiện nhiều luồng đồng thời. Một ứng dụng Java ngoài luồng chính có thể có các luồng khác thực thi đồng thời làm ứng dụng chạy nhanh và hiệu quả hơn.

VD: Trình duyệt web hay các chương trình chơi nhạc là 1 ví dụ điển hình về đa luồng.

+ Khi duyệt 1 trang web, có rất nhiều hình ảnh, CSS, javascript… được tải đồng thời bởi các luồng khác nhau.

+ Khi play nhạc, chúng ta vẫn có thể tương tác được với nút điều khiển như: Play, pause, next, back … vì luồng phát nhạc là luồng riêng biệt với luồng tiếp nhận tương tác của người dùng.

**Đa nhiệm (multitasking)**

**Multitasking**: Là khả năng chạy đồng thời một hoặc nhiều chương trình cùng một lúc trên một hệ điều hành. Hệ điều hành quản lý việc này và sắp xếp lịch phù hợp cho các chương trình đó. Ví dụ, trên hệ điều hành Windows chúng ta có làm việc đồng thời với các chương trình khác nhau như: Microsoft Word, Excel, Media Player, …

vậy Multi-thread là gì? Tại sao chúng lại ra đời, bài toán nào sử dụng Multi-Thread nhỉ?

Multi-thread là việc sử dụng nhiều thread trong một ứng dụng để thực hiện các tác vụ đồng thời, đồng bộ và tăng hiệu suất của ứng dụng. Các thread trong ứng dụng có thể chia sẻ các tài nguyên và dữ liệu, từ đó giúp giảm thời gian xử lý và tăng hiệu suất.

Việc sử dụng Multi-thread được đưa ra để giải quyết vấn đề về hiệu suất và tốc độ của ứng dụng. Trong quá trình thực hiện tác vụ, một ứng dụng thường phải đợi đến khi một tác vụ được hoàn thành trước khi thực hiện tác vụ tiếp theo. Việc sử dụng nhiều thread cho phép các tác vụ đồng thời được thực hiện, giảm thời gian chờ đợi và tăng hiệu suất.

Một số bài toán thường sử dụng Multi-thread như:

1. Xử lý video và âm thanh: Một ứng dụng xử lý video hoặc âm thanh có thể sử dụng Multi-thread để xử lý các khung hình và âm thanh đồng thời, giảm thời gian xử lý.
2. Xử lý dữ liệu lớn: Khi xử lý dữ liệu lớn, việc sử dụng Multi-thread giúp chia nhỏ dữ liệu và xử lý nhiều tác vụ đồng thời, giảm thời gian xử lý và tăng hiệu suất.
3. Ứng dụng web: Một ứng dụng web có thể sử dụng Multi-thread để tải các tài nguyên (như hình ảnh, CSS, JavaScript) đồng thời, giảm thời gian tải và tăng tốc độ hiển thị trang web.

Trong cả ba ví dụ trên, việc sử dụng Multi-thread giúp giảm thời gian xử lý và tăng hiệu suất của ứng dụng. Tuy nhiên, việc sử dụng Multi-thread cũng có thể gây ra một số vấn đề về đồng bộ hóa dữ liệu và xử lý, do đó cần phải được quản lý cẩn thận để đảm bảo tính ổn định và đáng tin cậy của ứng dụng.

Chúng ta sử dụng đa nhiệm (Đa nhiệm - Multitasking) để tận dụng tính năng của CPU.

Đa nhiệm có thể đạt được bằng hai cách:

1. Đa nhiệm dựa trên đơn tiến trình (Process) – Đa tiến trình (Multiprocessing).
   * Mỗi tiến trình có địa chỉ riêng trong bộ nhớ, tức là mỗi tiến trình phân bổ vùng nhớ riêng biệt.
   * Tiến trình là nặng.
   * Sự giao tiếp giữa các tiến trình có chi phí cao.
   * Chuyển đổi từ tiến trình này sang tiến trình khác đòi hỏi thời gian để đăng ký việc lưu và tải các bản đồ bộ nhớ, các danh sách cập nhật, …
2. Đa nhiệm dựa trên luồng (Thread) – Đa luồng (MultiThreading).
   * Các luồng chia sẻ không gian địa chỉ ô nhớ giống nhau.
   * Luồng là nhẹ.
   * Sự giao tiếp giữa các luồng có chi phí thấp.

Thế sự khác biệt giữa Đơn Tiến trình - Process và Luồng - Thread là gì?

Trong lập trình, một tiến trình (process) là một chương trình đang chạy trong hệ thống, bao gồm tất cả các tài nguyên cần thiết để thực thi. Một tiến trình có thể có nhiều luồng (thread) để thực hiện các tác vụ khác nhau trong cùng một chương trình. Tuy nhiên, các luồng trong cùng một tiến trình chia sẻ tài nguyên và bộ nhớ của tiến trình đó.

Sự khác biệt giữa tiến trình và luồng là:

1. Khái niệm: Tiến trình là một chương trình đang chạy trong hệ thống, còn luồng là một phần của tiến trình đó được sử dụng để thực hiện các tác vụ cụ thể.
2. Quản lý tài nguyên: Mỗi tiến trình được quản lý như một đơn vị độc lập với các tiến trình khác, có riêng các tài nguyên như bộ nhớ, tài nguyên mạng và tài nguyên I/O. Trong khi đó, các luồng trong cùng một tiến trình chia sẻ tài nguyên và bộ nhớ của tiến trình đó.
3. Thực thi: Các tiến trình có thể thực thi đồng thời trên nhiều CPU khác nhau, còn các luồng thực thi đồng thời trên cùng một CPU.
4. Khả năng độc lập: Các tiến trình là độc lập với nhau, có thể giao tiếp thông qua các cơ chế như bộ đệm hoặc ổ đĩa, trong khi các luồng chia sẻ cùng một bộ nhớ và tài nguyên nên phải được quản lý cẩn thận để tránh cạnh tranh tài nguyên.

Vì vậy, trong lập trình, người ta sử dụng tiến trình để thực hiện các tác vụ độc lập và sử dụng các luồng để thực hiện các tác vụ cụ thể trong một tiến trình.

Đa tiến trình (multiprocessing) và đa luồng (multithreading) cả hai được sử dụng để tạo ra hệ thống đa nhiệm (multitasking). Nhưng chúng ta sử dụng đa luồng nhiều hơn đa tiến trình bởi vì các luồng chia sẻ một vùng bộ nhớ chung. Chúng không phân bổ vùng bộ nhớ riêng biệt để tiết kiệm bộ nhớ, và chuyển đổi ngữ cảnh giữa các luồng mất ít thời gian hơn tiến trình.

[Diagram

Description automatically generated](https://gpcoder.com/wp-content/uploads/2018/02/MultiThread-SharedModel.png)

**Ưu điểm của đa luồng (Multi-process)**

* Nó không chặn người sử dụng vì các luồng là độc lập và bạn có thể thực hiện nhiều công việc cùng một lúc.
* Mỗi luồng có thể dùng chung và chia sẻ nguồn tài nguyên trong quá trình chạy, nhưng có thể thực hiện một cách độc lập.
* Luồng là độc lập vì vậy nó không ảnh hưởng đến luồng khác nếu ngoại lệ xảy ra trong một luồng duy nhất.
* Có thể thực hiện nhiều hoạt động với nhau để tiết kiệm thời gian. Ví dụ một ứng dụng có thể được tách thành : luồng chính chạy giao diện người dùng và các luồng phụ nhiệm gửi kết quả xử lý đến luồng chính.

**Nhược điểm của đa luồng**

* Càng nhiều luồng thì xử lý càng phức tạp.
* Xử lý vấn đề về tranh chấp bộ nhớ, đồng bộ dữ liệu khá phức tạp.
* Cần phát hiện tránh các luồng chết (dead lock), luồng chạy mà không làm gì trong ứng dụng cả.

Deadlock (Khoá chết) là gì? Deadlock xảy ra khi 2 tiến trình đợi nhau hoàn thành, trước khi chạy. Kết quả của quá trình là cả 2 tiến trình không bao giờ kết thúc.

**Vòng đời (các trạng thái) của một Thread trong java**

[Diagram

Description automatically generated](https://gpcoder.com/wp-content/uploads/2018/02/multithread-lifecylce.jpg)

Vòng đời của thread trong java được kiểm soát bởi JVM. Java định nghĩa các trạng thái của luồng trong các thuộc tính static của lớp **Thread.State**:

* **NEW**: Đây là trạng thái khi luồng vừa được khởi tạo bằng phương thức khởi tạo của lớp Thread nhưng chưa được start(). Ở trạng thái này, luồng được tạo ra nhưng chưa được cấp phát tài nguyên và cũng chưa chạy. Nếu luồng đang ở trạng thái này mà ta gọi các phương thức ép buộc stop,resume,suspend … sẽ là nguyên nhân sảy ra ngoại lệ IllegalThreadStateException .
* **RUNNABLE**: Sau khi gọi phương thức start() thì luồng test đã được cấp phát tài nguyên và các lịch điều phối CPU cho luồng test cũng bắt đầu có hiệu lực. Ở đây, chúng ta dùng trạng thái là Runnable chứ không phải Running, vì luồng không thực sự luôn chạy mà tùy vào hệ thống mà có sự điều phối CPU khác nhau.
* **WAITING** : Thread chờ không giới hạn cho đến khi một luồng khác đánh thức nó.
* **TIMED\_WAITING** : Thread chờ trong một thời gian nhất định, hoặc là có một luồng khác đánh thức nó.
* **BLOCKED**: Đây là 1 dạng của trạng thái “Not Runnable”, là trạng thái khi Thread vẫn còn sống, nhưng hiện tại không được chọn để chạy. Thread chờ một monitor để unlock một đối tượng mà nó cần.
* **TERMINATED**: Một thread ở trong trạng thái terminated hoặc dead khi phương thức run() của nó bị thoát.

**Cách tạo luồng trong Java**

Trong java ta có thể tạo ra một luồng bằng một trong hai cách sau: tạo 1 đối tượng của lớp được **extend** từ **class Thread** hoặc **implements** từ **interface Runnable**.

Trong Java, tạo luồng có thể thực hiện thông qua hai cách:

1. Kế thừa từ lớp **Thread**:

* Cách này đơn giản hơn khi muốn tạo một luồng mới. Chỉ cần tạo lớp con của **Thread** và ghi đè phương thức **run()**.
* Lớp con sẽ kế thừa tất cả các phương thức của lớp **Thread**, bao gồm cả các phương thức điều khiển luồng (**start()**, **stop()**, **sleep()**, ...) và phương thức khởi tạo.

public class MyThread extends Thread {

@Override

public void run() { //code xử lý trong luồng } }

1. Thực thi interface **Runnable**:

* Cách này linh hoạt hơn khi muốn tạo nhiều luồng có cùng logic xử lý.
* Class mới được tạo ra chỉ cần triển khai phương thức **run()**.
* Không kế thừa các phương thức điều khiển luồng (**start()**, **stop()**, **sleep()**, ...) và phương thức khởi tạo.

public class MyRunnable implements Runnable {

@Override

public void run() { //code xử lý trong luồng } }

Về ưu điểm, việc sử dụng lớp **Thread** cung cấp cho người lập trình nhiều phương thức điều khiển và tùy chỉnh luồng hơn. Tuy nhiên, nó cũng phức tạp hơn và tạo ra nhiều đối tượng. Trong khi đó, thực thi interface **Runnable** đơn giản hơn và giúp tiết kiệm tài nguyên hơn. Bởi vì nó không tạo ra một đối tượng mới, nó có thể được sử dụng để tạo các luồng mới một cách hiệu quả hơn. Do đó, việc sử dụng **Runnable** nên được ưu tiên hơn khi không cần điều khiển nhiều đối tượng luồng.

**Tạo luồng bằng cách extend từ lớp Thread**

Để tạo luồng bằng cách tạo lớp kế thừa từ lớp Thread, ta phải làm các công việc sau :

1. Khai báo 1 lớp mới kế thừa từ lớp Thread
2. Override lại phương thức run ở lớp này, những gì trong phương thức run sẽ được thực thi khi luồng bắt đầu chạy. Sau khi luồng chạy xong tất cả các câu lệnh trong phương thức run thì luồng cũng tự hủy.
3. Tạo 1 thể hiện (hay 1 đối tượng) của lớp ta vừa khai báo.
4. Sau đó gọi phương thức start() của đối tượng này để bắt đầu thực thi luồng.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | **package** com.gpcoder.simple;    **public** **class** TheadSimple **extends** Thread {  **public** **void** run() {          System.out.println("thread is running...");      }    **public** **static** **void** main(String args[]) {          TheadSimple t1 = **new** TheadSimple();          t1.start();      }  } |

**Lưu ý :**

* Tuy ta khai báo những công việc cần làm của luồng trong phương thức run() nhưng khi thực thi luồng ta phải gọi phương thức start(). Vì đây là phương thức đặc biệt mà java xây dựng sẵn trong lớp Thread, phương thức này sẽ cấp phát tài nguyên cho luồng mới rồi chạy phương thức run() ở luồng này. Vì vậy, nếu ta gọi phương thức run() mà không gọi start() thì cũng tương đương với việc gọi 1 phương thức của 1 đối tượng bình thường và phương thức vẫn chạy trên luồng mà gọi phương thức chứ không chạy ở luồng mới tạo ra, nên vẫn chỉ có 1 luồng chính làm việc chứ ứng dụng vẫn không phải là đa luồng.
* Sau khi start một thread, nó không bao giờ có thể được start lại. Nếu bạn làm như vậy, một ngoại lệ **IllegalThreadStateException** sẽ xảy ra.

**Tạo luồng bằng cách implement từ Interface Runnable**

Để tạo luồng bằng cách hiện thực từ Interface Runnable, ta phải làm các công việc sau :

1. Khai báo 1 lớp mới implements từ Interface Runnable
2. Hiện thực phương thức run() ở lớp này, những gì trong phương thức run() sẽ được thực thi khi luồng bắt đầu chạy. Sau khi luồng chạy xong tất cả các câu lệnh trong phương thức run thì luồng cũng tự hủy.
3. Tạo 1 thể hiện (hay 1 đối tượng) của lớp ta vừa khai báo. (VD : Tên đối tượng là r1)
4. Tạo 1 thể hiện của lớp Thread bằng phương thức khởi tạo : Thread(Runnable target)
   * Runnable target: Là 1 đối tượng thuốc lớp được implements từ giao diện Runnable.
   * Ví dụ: Thread t1 = new Thread(r1);
5. Gọi phương thức start() của đối tượng t1.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | **package** com.gpcoder.simple;    **public** **class** RunnableSimple **implements** Runnable {  **public** **void** run() {          System.out.println("thread is running...");      }    **public** **static** **void** main(String args[]) {          RunnableSimple runable = **new** RunnableSimple();          Thread t1 = **new** Thread(runable);          t1.start();      }  } |

**Khi nào implements từ interface Runnable?**

+ Cách hay được sử dụng và được yêu thích là dùng **interface Runnable**, bởi vì nó không yêu cầu phải tạo một lớp kế thừa từ lớp Thread. Trong trường hợp ứng dụng thiết kế yêu cầu sử dụng đa kế thừa, chỉ có interface mới có thể giúp giải quyết vấn đề. Ngoài ra, **Thread Pool** rất hiểu quả và có thể được cài đặt, sử dụng rất hơn giản.  
+ Trong trường hợp còn lại ta có thể kế thừa từ lớp Thread.

**Ví dụ minh họa sử dụng đa luồng**

**Ví dụ Tạo luồng bằng cách extend từ class Thread**

Tạo luồng extend từ class Thead

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | **package** com.gpcoder.flow;    **public** **class** ThreadDemo **extends** Thread {  **private** Thread t;  **private** String threadName;        ThreadDemo(String name) {          threadName = name;          System.out.println("Creating " + threadName);      }        @Override  **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(**this**, threadName);              t.start();          }      }    } |

Chương trình sử dụng đa luồng:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | **package** com.gpcoder.flow;    **public** **class** ThreadDemoTest {  **public** **static** **void** main(String args[]) {          System.out.println("Main thread running... ");            ThreadDemo T1 = **new** ThreadDemo("Thread-1-HR-Database");          T1.start();            ThreadDemo T2 = **new** ThreadDemo("Thread-2-Send-Email");          T2.start();            System.out.println("==&gt; Main thread stopped!!! ");      }  } |

Kết quả thực thi chương trình trên:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | Main thread running...  Creating Thread-1-HR-Database  Starting Thread-1-HR-Database  Creating Thread-2-Send-Email  Starting Thread-2-Send-Email  ==&amp;amp;gt; Main thread stopped!!!  Running Thread-1-HR-Database  Running Thread-2-Send-Email  Thread: Thread-2-Send-Email, 4  Thread: Thread-1-HR-Database, 4  Thread: Thread-1-HR-Database, 3  Thread: Thread-2-Send-Email, 3  Thread: Thread-2-Send-Email, 2  Thread: Thread-1-HR-Database, 2  Thread: Thread-2-Send-Email, 1  Thread: Thread-1-HR-Database, 1  Thread Thread-2-Send-Email exiting.  Thread Thread-1-HR-Database exiting. |

Kết quả chương trình trên được giải thích thông qua hình bên dưới:

[Diagram

Description automatically generated with medium confidence](https://gpcoder.com/wp-content/uploads/2018/02/Multi-Thread-Flow.png)

**Ví dụ Tạo luồng bằng cách implement từ Interface Runnable**

Tạo luồng implement từ Interface Runnable

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | **package** com.gpcoder.flow;    **class** RunnableDemo **implements** Runnable {  **private** Thread t;  **private** String threadName;        RunnableDemo(String name) {          threadName = name;          System.out.println("Creating " + threadName);      }        @Override  **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(**this**, threadName);              t.start();          }      }    } |

Chương trình sử dụng đa luồng:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | **package** com.gpcoder.flow;    **public** **class** RunnableDemoTest {  **public** **static** **void** main(String args[]) {          System.out.println("Main thread running... ");            RunnableDemo R1 = **new** RunnableDemo("Thread-1-HR-Database");          R1.start();            RunnableDemo R2 = **new** RunnableDemo("Thread-2-Send-Email");          R2.start();            System.out.println("==&gt; Main thread stopped!!! ");      }  } |

Kết quả thực thi chương trình trên:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | Main thread running...  Creating Thread-1-HR-Database  Starting Thread-1-HR-Database  Creating Thread-2-Send-Email  Starting Thread-2-Send-Email  ==&amp;amp;gt; Main thread stopped!!!  Running Thread-1-HR-Database  Running Thread-2-Send-Email  Thread: Thread-1-HR-Database, 4  Thread: Thread-2-Send-Email, 4  Thread: Thread-1-HR-Database, 3  Thread: Thread-2-Send-Email, 3  Thread: Thread-1-HR-Database, 2  Thread: Thread-2-Send-Email, 2  Thread: Thread-1-HR-Database, 1  Thread: Thread-2-Send-Email, 1  Thread Thread-1-HR-Database exiting.  Thread Thread-2-Send-Email exiting. |

Kết quả chương trình trên được giải thích thông qua hình bên dưới:

[Diagram

Description automatically generated](https://gpcoder.com/wp-content/uploads/2018/02/Multi-Thread-Flow-Runnable.png)

**Các phương thức của lớp Thread thường hay sử dụng**

* **suspend()** : Đây là phương thức làm tạm dừng hoạt động của 1 luồng nào đó bằng các ngưng cung cấp CPU cho luồng này. Để cung cấp lại CPU cho luồng ta sử dụng phương thức resume(). Cần lưu ý 1 điều là ta không thể dừng ngay hoạt động của luồng bằng phương thức này. Phương thức suspend() không dừng ngay tức thì hoạt động của luồng mà sau khi luồng này trả CPU về cho hệ điều hành thì không cấp CPU cho luồng nữa.
* **resume()** : Đây là phương thức làm cho luồng chạy lại khi luồng bị dừng do phương thức suspend() bên trên. Phương thức này sẽ đưa luồng vào lại lịch điều phối CPU để luồng được cấp CPU chạy lại bình thường.
* **stop()** : Luồng này sẽ kết thúc phương thức run() bằng cách ném ra 1 ngoại lệ ThreadDeath, điều này cũng sẽ làm luồng kết thúc 1 cách ép buộc. Nếu giả sử, trước khi gọi stop() mà luồng đang nắm giữa 1 đối tượng nào đó hoặc 1 tài nguyên nào đó mà luồng khác đang chờ thì có thể dẫn tới việc sảy ra deadlock.
* **destroy()** : dừng hẳn luồng.
* **isAlive()** : Phương thức này kiểm tra xem luồng còn active hay không. Phương thức sẽ trả về true nếu luồng đã được start() và chưa rơi vào trạng thái dead. Nếu phương thức trả về false thì luồng đang ở trạng thái “New Thread” hoặc là đang ở trạng thái “Dead”
* **yeild()** : Hệ điều hành đa nhiệm sẽ phân phối CPU cho các tiến trình, các luồng theo vòng xoay. Mỗi luồng sẽ được cấp CPU trong 1 khoảng thời gian nhất định, sau đó trả lại CPU cho hệ điều hành (HĐH), HĐH sẽ cấp CPU cho luồng khác. Các luồng sẽ nằm chờ trong hàng đợi Ready để nhận CPU theo thứ tự. Java có cung cấp cho chúng ta 1 phương thức khá đặc biệt là **yeild()**, khi gọi phương thức này luồng sẽ bị ngừng cấp CPU và nhường cho luồng tiếp theo trong hàng chờ Ready. Luồng không phải ngưng cấp CPU như suspend mà chỉ ngưng cấp trong lần nhận CPU đó mà thôi.
* **sleep(long)** : tạm dừng luồng trong một khoảng thời gian millisecond.
* **join()**: thông báo rằng hãy chờ thread này hoàn thành rồi thread cha mới được tiếp tục chạy.
* **join(long)** : Thread cha cần phải đợi millisecond mới được tiếp tục chạy, kể từ lúc gọi join(long). Nếu tham số millis = 0 nghĩa là đợi cho tới khi luồng này kết thúc.
* **getName()** : Trả về tên của thread.
* **setName(String name)** : Thay đổi tên của thread.
* **getId()** : Trả về id của thread.
* **getState():**trả về trạng thái của thread.
* **currentThread()** : Trả về tham chiếu của thread đang được thi hành.
* **getPriority()** : Trả về mức độ ưu tiên của thread.
* **setPriority(int)** : Thay đổi mức độ ưu tiên của thread.
* **isDaemon()** : Kiểm tra nếu thread là một luồng Daemon.
* **setDaemon(boolean)**: xác định thread là một luồng Daemon hay không.
* **interrupt()** : làm gián đoạn một luồng trong java. Nếu thread nằm trong trạng thái sleep hoặc wait, nghĩa là sleep() hoặc wait() được gọi ra. Việc gọi phương thức interrupt() trên thread đó sẽ phá vỡ trạng thái sleep hoặc wait và ném ra ngoại lệ InterruptedException. Nếu thread không ở trong trạng thái sleep hoặc wait, việc gọi phương thức interrupt() thực hiện hành vi bình thường và không làm gián đoạn thread nhưng đặt cờ interrupt thành true.
* **isInterrupted()** : kiểm tra nếu thread đã bị ngắt.
* **interrupted()** : kiểm tra nếu thread hiện tại đã bị ngắt.

**Một số thông tin liên quan đến luồng**

**Định danh của luồng (ThreadId)**

ThreadId là định danh của luồng, nó dùng để phân biệt với các luồng khác cùng tiến trình hoặc cùng tập luồng. Đây là thông số mà máy ảo java tự tạo ra khi ta tạo luồng nên ta không thể sửa đổi cũng như áp đặt thông số này khi tạo luồng. Nhưng ta có thể lấy được nó thông qua phương thức getId() của lớp Thread

**Tên của luồng (ThreadName)**

ThreadName là tên của luồng, đây là thuộc tính mà ta có thể đặt hoặc không đặt cho luồng. Nếu ta không đặt cho luồng thì máy ảo java sẽ tự đặt với quy tắc sau: “Thread-” + Thứ tự luồng được tạo ra, bắt đầu từ 0.

**Độ ưu tiên của luồng (Priority)**

Như đã nói ở phần trước, mỗi luồng có 1 độ ưu tiên nhất định. Đây sẽ là thông số quyết định mức ưu tiên khi cấp phát CPU cho các luồng.

Trong java, đế đặt độ ưu tiên cho 1 luồng ta dùng phương thức: **void setPriority(int newPriority)**

* int newPriority : Là giá trị từ 1 đến 10.

Java có định nghĩa sẵn 3 mức ưu tiên chuẩn như sau:

* Thread.**MIN\_PRIORITY** (giá trị **01**)
* hread.**NORM\_PRIORITY** (giá trị **05**)
* Thread.**MAX\_PRIORITY** (giá trị **10**)

Để lấy độ ưu tiên của 1 luồng, ta dùng phương thức: **int getPriority()**

**Ví dụ minh họa**

WorkingThread.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | **package** com.gpcoder.info;    **public** **class** WorkingThread **extends** Thread {  **public** WorkingThread(String name) {  **super**(name);      }    **public** **void** run() {  **for** (**int** i = 0; i < 5; i++) {              System.out.printf("Luồng: %s có độ ưu tiên là %d \n",  getName(), getPriority());          }      }  } |

ThreadInfoExample.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | **package** com.gpcoder.info;    **public** **class** ThreadInfoExample {    **public** **static** **void** main(String[] args) {          Thread t1 = **new** WorkingThread("Luồng 1");          Thread t2 = **new** WorkingThread("Luồng 2");          Thread t3 = **new** WorkingThread("Luồng 3");            System.out.println("ID luồng 1: " + t1.getId());          System.out.println("ID luồng 2: " + t2.getId());          System.out.println("ID luồng 3: " + t3.getId());            t1.setPriority(1);          t2.setPriority(5);          t3.setPriority(10);            t1.start();          t2.start();          t3.start();      }    } |

Kết quả thực thi chương trình trên:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | ID luồng 1: 10  ID luồng 2: 11  ID luồng 3: 12  Luồng: Luồng 2 có độ ưu tiên là 5  Luồng: Luồng 2 có độ ưu tiên là 5  Luồng: Luồng 2 có độ ưu tiên là 5  Luồng: Luồng 2 có độ ưu tiên là 5  Luồng: Luồng 2 có độ ưu tiên là 5  Luồng: Luồng 1 có độ ưu tiên là 1  Luồng: Luồng 3 có độ ưu tiên là 10  Luồng: Luồng 3 có độ ưu tiên là 10  Luồng: Luồng 3 có độ ưu tiên là 10  Luồng: Luồng 3 có độ ưu tiên là 10  Luồng: Luồng 3 có độ ưu tiên là 10  Luồng: Luồng 1 có độ ưu tiên là 1  Luồng: Luồng 1 có độ ưu tiên là 1  Luồng: Luồng 1 có độ ưu tiên là 1  Luồng: Luồng 1 có độ ưu tiên là 1 |

**Sử dụng phương thức sleep()**

Phương thức sleep() của lớp Thread được sử dụng để tạm ngừng một thread cho một khoảng thời gian nhất định.

Ví dụ chương trình in ra số từ 1 – 5, tạm ngừng 500 ms trước khi in chữ số tiếp theo.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | **package** com.gpcoder.sleep;    **public** **class** SleepMethodExample **extends** Thread {    **public** **void** run() {  **for** (**int** i = 1; i <= 5; i++) {              System.out.println(i);  **try** {                  Thread.sleep(500);              } **catch** (InterruptedException e) {                  e.printStackTrace();              }          }      }    **public** **static** **void** main(String args[]) {          SleepMethodExample t1 = **new** SleepMethodExample();          t1.start();      }    } |

**Sử dụng join() và join(long)**

**join()** : thông báo rằng hãy chờ thread này hoàn thành rồi thread cha mới được tiếp tục chạy.

Ví dụ:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | **package** com.gpcoder.join;    **public** **class** UsingJoinMethod **extends** Thread {    **public** UsingJoinMethod(String name) {  **super**(name);      }        @Override  **public** **void** run() {          System.out.println(getName());  **for** (**int** i = 1; i <= 5; i++) {  **try** {                  System.out.print(i + " ");                  Thread.sleep(300);              } **catch** (InterruptedException ie) {                  System.out.println(ie.toString());              }          }          System.out.println();      }    **public** **static** **void** main(String[] args) **throws** InterruptedException {          UsingJoinMethod t1 = **new** UsingJoinMethod("Thread 1");          UsingJoinMethod t2 = **new** UsingJoinMethod("Thread 2");          t1.start();          t1.join();          t2.start();          System.out.println("Main Thread Finished");      }  } |

Thực thi chương trình trên:

|  |  |
| --- | --- |
| 1  2  3  4  5 | Thread 1  1 2 3 4 5  Main Thread Finished  Thread 2  1 2 3 4 5 |

**join(long)** : Thread cha cần phải đợi millisecond mới được tiếp tục chạy, kể từ lúc gọi join(long). Nếu tham số millis = 0 nghĩa là đợi cho tới khi luồng này kết thúc.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | **package** com.gpcoder.join;    **public** **class** UsingJoinMethod2 **extends** Thread {    **public** UsingJoinMethod2(String name) {  **super**(name);      }        @Override  **public** **void** run() {          System.out.println(getName());  **for** (**int** i = 1; i <= 5; i++) {  **try** {                  System.out.print(i + " ");                  Thread.sleep(300);              } **catch** (InterruptedException ie) {                  System.out.println(ie.toString());              }          }          System.out.println();      }    **public** **static** **void** main(String[] args) **throws** InterruptedException {          UsingJoinMethod2 t1 = **new** UsingJoinMethod2("Thread 1");          UsingJoinMethod2 t2 = **new** UsingJoinMethod2("Thread 2");          t1.start();            // Main Thread phải chờ 450ms mới được tiếp tục chạy.          // Không nhất thiết phải chờ Thread t1 kết thúc          t1.join(450);            t2.start();          System.out.println("Main Thread Finished");      }  } |

Thực thi chương trình trên:

|  |  |
| --- | --- |
| 1  2  3  4  5 | Thread 1  1 2 Main Thread Finished  Thread 2  1 3 2 4 3 5 4  5 |

**Xử lý ngoại lệ cho Thread**

Phương thức Thread.setDefaultUncaughtExceptionHandler() thiết lập mặc định xử lý khi luồng đột ngột chấm dứt do một ngoại lệ xảy ra mà không có xử lý khác đã được xác định cho luồng đó.

Ví dụ:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | **package** com.gpcoder.exception;    **import** java.util.Random;    **public** **class** WorkingThread **implements** Runnable {        @Override  **public** **void** run() {  **while** (**true**) {              processSomething();          }      }    **private** **void** processSomething() {  **try** {              System.out.println("Processing working thread");              Thread.sleep(500);          } **catch** (InterruptedException e) {              e.printStackTrace();          }            Random r = **new** Random();  **int** i = r.nextInt(100);  **if** (i &gt; 70) {  **throw** **new** RuntimeException("Simulate an exception was  not handled in the thread");          }      }    } |

Chương trình minh họa xử lý Thread Exception

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | **package** com.gpcoder.exception;    **public** **class** ThreadExceptionDemo {    **public** **static** **void** main(String[] args) {          System.out.println("==> Main thread running...");            Thread thread = **new** Thread(**new** WorkingThread());          Thread.setDefaultUncaughtExceptionHandler(**new** Thread.UncaughtExceptionHandler() {              @Override  **public** **void** uncaughtException(Thread t, Throwable e) {                  System.out.println("#Thread: " + t);                  System.out.println("#Thread exception message: " + e.getMessage());              }          });            thread.start();          System.out.println("==&amp;gt; Main thread end!!!");      }  } |

Thực thi chương trình trên:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | ==&amp;amp;gt; Main thread running...  ==&amp;amp;gt; Main thread end!!!  Processing working thread  Processing working thread  Processing working thread  Processing working thread  Processing working thread  Processing working thread  #Thread: Thread[Thread-0,5,main]  #Thread exception message: Have a problem... |

Trên đây là những kiến thức cơ bản về đa luồng (Multi-thread) trong java. Chúng ta sẽ tiếp tục tìm hiểu về các vấn đề khác của đa luồng trong Java ở các bài viết tiếp theo. Cám ơn các bạn đã quan tâm và theo dõi bài viết.

**Tài liệu tham khảo:**

* <https://docs.oracle.com/javase/tutorial/essential/concurrency/>
* <https://www.javatpoint.com/multithreading-in-java>
* <https://o7planning.org/vi/10269/huong-dan-lap-trinh-da-luong-trong-java>
* http://sinhvienit.net/forum/da-luong-trong-java-multithreading-in-java-phan-1-cach-tao-luong-va-threadid-threadname-piority-stacksize.251150.html
* http://sinhvienit.net/forum/da-luong-trong-java-multithreading-in-java-phan-2-trang-thai-cua-luong-va-phuong-thuc-suspend-resume-stop-sleep.251295.html

**4.9**

42

**Nếu bạn thấy hay thì hãy chia sẻ bài viết cho mọi người nhé!**

SHARES

*Chuyên mục:*[**Multi-Thread**](https://gpcoder.com/category/java-core/multi-thread/)*Được gắn thẻ:*[**Multithreading**](https://gpcoder.com/tag/multithreading/)

[Integer Constant Pool trong Java](https://gpcoder.com/3468-integer-constant-pool-trong-java/)

[Luồng Daemon (Daemon Thread) trong Java](https://gpcoder.com/3513-luong-daemon-daemon-thread-trong-java/)

**Có thể bạn muốn xem:**

* [Sử dụng CyclicBarrier trong Java](https://gpcoder.com/3603-su-dung-cyclicbarrier-trong-java/) **(13/03/2018)**
* [Đồng bộ hóa các luồng trong Java](https://gpcoder.com/3514-dong-bo-hoa-cac-luong-trong-java/) **(20/02/2018)**
* [Vấn đề Nhà sản xuất (Producer) – Người tiêu dùng (Consumer) và đồng bộ hóa các luồng trong Java](https://gpcoder.com/6180-van-de-nguoi-san-xuat-producer-nguoi-tieu-dung-consumer-va-dong-bo-hoa-cac-luong-trong-java/) **(23/09/2019)**
* [Thực thi nhiều tác vụ cùng lúc như thế nào trong Java?](https://gpcoder.com/5100-thuc-thi-nhieu-tac-vu-cung-luc-nhu-the-nao-trong-java/) **(01/02/2019)**
* [Sử dụng CountDownLatch trong Java](https://gpcoder.com/3594-su-dung-countdownlatch-trong-java/) **(09/03/2018)**

**Bình luận**

2 bình luận

**Tìm kiếm**