**Hướng dẫn tạo và sử dụng ThreadPool**

**1. ThreadPool là gì?**

Trong Java, ThreadPool được dùng để giới hạn số lượng Thread được chạy bên trong ứng dụng của chúng ta trong cùng một thời điểm. Nếu chúng ta không có sự giới hạn này, mỗi khi có một Thread mới được tạo ra và được cấp phát bộ nhớ bằng từ khóa new thì sẽ có vấn đề về bộ nhớ và hiệu suất, có thể dẫn đến lỗi crash chương trình.

**Ví dụ:** Khi chúng ta viết chương trình tải các tập tin từ Internet, mỗi tập tin cần 1 Thread để thực hiện quá trình tải, giả sử cần tải 1000 tệp hình ảnh thì chúng ta phải cần tới 1000 Thread hoạt động cùng một thời điểm trong cùng một chương trình. Điều này sẽ dễ dẫn đến lỗi quá tải của chương trình, làm ảnh hưởng đến hiệu suất và chương trình sẽ rất dễ bị crash vì khó kiểm soát.

Vì vậy, để khắc phục hiện tượng này, Java cho phép chúng ta thay vì phải tạo mới Thread cho mỗi nhiệm vụ, quá trình được thực hiện trong cùng một thời điểm thì các nhiệm vụ, quá trình đó có thể được đưa vào trong một ThreadPool để khi trong ThreadPool có bất kỳ Thread nào đang không phải thực hiện một nhiệm vụ nào thì sẽ có nhiệm vụ gán vào một trong số các Thread đó để thực thi. Điều này sẽ giúp khắc phục được sự tắc nghẽn và chương trình sẽ kiểm soát được các luồng thực thi.

Bên trong ThreadPool, các nhiệm vụ sẽ được chèn vào trong một Blocking Queue. Blocking Queue có thể hiểu là nơi chứa các nhiệm vụ mà các Thread sẽ lấy chúng ra và thực thi lần lượt. Mỗi khi có một nhiệm vụ mới được thêm vào Queue và sau đó sẽ chỉ có một Thread đang không phải thực hiện một nhiệm vụ nào vào Queue lấy nhiệm vụ đó ra, còn các Thread còn lại phải chờ đợi cho đến khi Thread đó lấy nhiệm vụ ra thành công.

Ví dụ dưới đây sẽ minh họa cách tạo ThreadPool bằng cách sử dụng ThreadPoolExecutor:

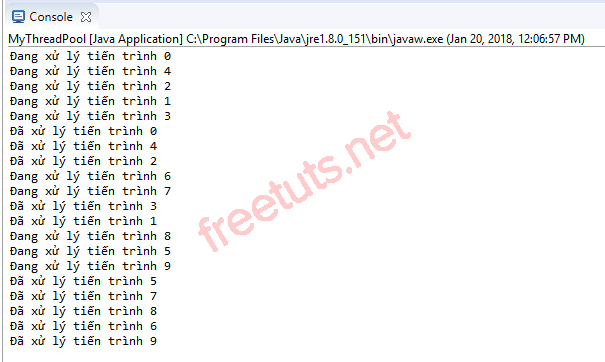
**RunPool.java**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | package vidu;    public class RunPool implements Runnable {        int id;        @Override      public void run() {          System.out.println("Đang xử lý tiến trình " + id);            try {              Thread.sleep(1000);          } catch (InterruptedException e) {              e.printStackTrace();          }            System.out.println("Đã xử lý tiến trình " + id);      }        public RunPool(int id) {          this.id = id;      }  } |

**MyThreadPool.java**

|  |  |
| --- | --- |
| 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 | package vidu;    import java.util.concurrent.ArrayBlockingQueue;  import java.util.concurrent.ThreadPoolExecutor;  import java.util.concurrent.TimeUnit;    public class MyThreadPool {        public static void main(String[] args) {          ArrayBlockingQueue<Runnable> hangDoi = new ArrayBlockingQueue<>(100);            // khi số tiến trình của chúng ta vượt quá maxSize ở đây là 5          // ví dụ như đối số thứ nhất = 6          // thì tất cả những tiến trình mới mà chúng ta tạo ra sẽ được đưa vào hangDoi          ThreadPoolExecutor threadPoolExecutor = new ThreadPoolExecutor(5, 5, 12,              TimeUnit.SECONDS, hangDoi);            // dùng vòng lặp for để có thể chạy các Thread          for (int i = 0; i < 10; i++) {              // trong phương thức execute() thì đối số truyền vào phải là một Runnable              // đó là lý do mà lớp RunPool phải implements từ interface Runnable              threadPoolExecutor.execute(new RunPool(i));          }      }    } |

Kết quả sau khi biên dịch chương trình:



Giải thích hoạt động của chương trình trên:

Trong dòng code khởi tạo ThreadPoolExecutor: ThreadPoolExecutor threadPoolExecutor = new ThreadPoolExecutor(5, 5, 1, TimeUnit.SECONDS, hangDoi); chúng ta có 5 tham số:

* Đối số 1 (*corePoolSize*) là số lượng Thread tối thiểu trong ThreadPool, ở đây corePoolSize = 5. Khi khởi tạo, số lượng Thread có thể là 0. Khi nhiệm vụ được thêm vào thì Thread mới được tạo ra và kể từ đây, nếu số lượng Thread ít hơn corePoolSize thì những Thread mới sẽ được tạo ra đến khi số Thread bằng giá trị của corePoolSize.
* Đối số 2 (*maximumPoolSize*) là số lượng tối đa các Thread trong ThreadPool.
* Đối số 3 (*keepAliveTime*): khi số Thread lớn hơn corePoolSize thì keepAliveTime là thời gian tối đa mà 1 Thread "*nhàn rỗi*" chờ nhiệm vụ. Khi hết thời gian chờ mà Thread đó chưa có nhiệm vụ thì nó sẽ bị hủy.
* Đối số 4 (*unit*) là đơn vị thời gian của keepAliveTime. Trong ví dụ này thì unit của tôi là TimeUnit.SECONDS.
* Đối số 5 (*workQueue*) là hàng đợi dùng để chứa các nhiệm vụ mà các Thread sẽ lấy chúng ra và thực thi lần lượt, ở đây tôi dùng ArrayBlockingQueue.

**2. ExecutorService**

Kể từ Java 5 trở đi, ThreadPool đã được xây dựng sẵn trong gói java.util.concurrent, vì vậy chúng ta không cần phải tạo một ThreadPool mà thay vào đó chúng ta sẽ sử dụng các lớp có sẵn của gói này. Java cung cấp cho chúng ta lớp Executor, interface của lớp Executor là ExecutorService.

Ta có thể tạo ThreadPool thông qua Interface ExecutorService, các nhiệm vụ sẽ được đưa vào ThreadPool và được xử lý bằng một trong những phương thức có sẵn mà Executor cung cấp như sau:

* newSingleThreadExecutor(): Trong ThreadPool chỉ có 1 Thread và các nhiêm vụ sẽ được xử lý một cách tuần tự.
* newCachedThreadPool(): Trong ThreadPool sẽ có nhiều Thread và các nhiệm vụ sẽ được xử lý một cách song song. Các Thread cũ sau khi xử lý xong sẽ được sử dụng lại cho nhiệm vụ mới. Mặc định nếu một Thread không được sử dụng trong vòng 60 giây thì Thread đó sẽ bị tắt.
* newFixedThreadPool(): Trong ThreadPool sẽ được cố định các Thread. Nếu một nhiệm vụ mới được đưa vào mà các Thread đều đang "*bận rộn*" thì nhiệm vụ đó sẽ được gửi vào Blocking Queue và sau đó nếu có một Thread đã thực thi xong nhiệm vụ của nó thì nhiệm vụ đang ở trong Queue đó sẽ được push ra khỏi Queue và được Thread đó xử lý tiếp.
* newScheduledThreadPool(): tương tự như newCachedThreadPool() nhưng sẽ có thời gian delay giữa các Thread.
* newSingleThreadScheduledExecutor(): tương tự như newSingleThreadExecutor() nhưng sẽ có khoảng thời gian delay giữa các Thread.

Interface ExecutorService đại diện cho cơ chế thực thi bất đồng bộ có khả năng thực thi các nhiệm vụ trong background (*nền*). ExecutorService tương tự như một ThreadPool và trong thực tế, việc triển khai ExecutorService là một triển khai ThreadPool.

Sau đây tôi sẽ đưa ra một ví dụ minh họa cách sử dụng ExecutorService:

**RunPool.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 vidu;    public class RunPool implements Runnable {        int id;        @Override      public void run() {          System.out.println("Đang xử lý tiến trình " + id);            try {              Thread.sleep(1000);          } catch (InterruptedException e) {              e.printStackTrace();          }            System.out.println("Đã xử lý tiến trình " + id);      }        public RunPool(int id) {          this.id = id;      }  }  **MyThreadPool.java**   |  |  | | --- | --- | | 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 | package vidu;    import java.util.concurrent.ExecutorService;  import java.util.concurrent.Executors;  import java.util.concurrent.TimeUnit;    public class MyThreadPool {        public static void main(String[] args) {            ExecutorService pool = Executors.newFixedThreadPool(5);          for (int i = 0; i < 10; i++) {              pool.submit(new RunPool(i));    // chay ThreadPool, đối số là 1 Runnable          }            try {              // thời gian sống của mỗi Thread là 1 ngày (nếu nó chưa thực thi xong)              pool.awaitTermination(1, TimeUnit.DAYS);          } catch (InterruptedException e) {              e.printStackTrace();          }            pool.shutdown();    // tắt ThreadPool      }    } | |

Kết quả sau khi biên dịch chương trình:

