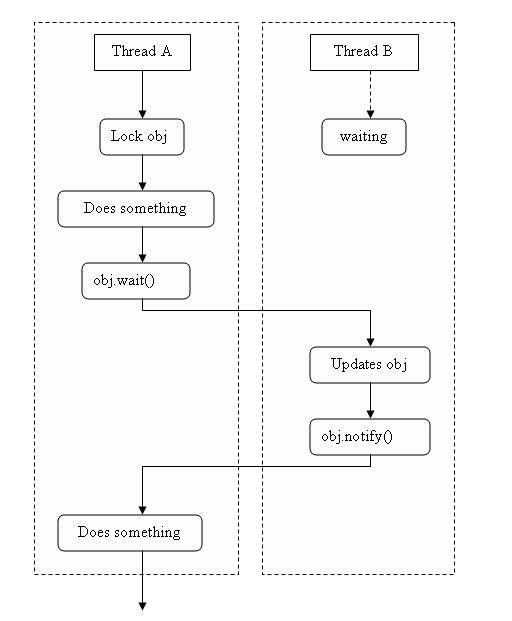
Java Thread - Luồng trong Java  
  
​

Mở đầu  
  
Bạn đã từng nghe nhiều về Multithreading. Ban đã biết về bộ xử lý Dual core của intel của AMD. Bạn biết rằng các công nghệ đó hỗ trợ việc xử lý song song...  
  
Bạn có biết rằng bản thân ngôn ngữ Java cũng hỗ trợ Multithreading. Với bài viết này, tôi hi vọng sẽ giúp các bạn tìm hiểu sơ lược về Thread trong Java. Một lĩnh vực khá thú vị.  
  
Thread  
  
Có một định nghĩa về thread như sau :  
  
Thread là một dòng các điều khiển trong một process hay một ứng dụng. Nguyên văn là : Threads are multiple flows of control within a single program or process.  
  
Với cơ chế multithreading ứng dụng của bạn có thể thực thi đồng thời nhiều dòng lệnh cùng lúc. Có nghĩa là bạn có thể làm nhiều công việc đồng thời trong cùng một ứng dụng của bạn. Có thể hiểu một cách hết sức đơn giản : hệ điều hành với cơ chế đa nhiệm cho phép nhiều ứng dụng chạy cùng lúc. Thì với cơ chế đa luồng, mỗi ứng dụng của bạn có thể thực hiện được nhiều công việc đồng thời.  
  
Bạn sẽ hỏi tại sao không dùng nhiều processes , sao không là multiprocessing mà lại cần đến multithreading ?  
  
Câu trả lời đơn giản nhất là: Việc tạo ra và quản lý các process đòi hỏi nhiều tài nguyên của hệ thống (cả ram và CPU) nhiều hơn rất nhiều so với việc tạo ra một thread. Trong khi đó bạn có thể chỉ cần tạo ra một thread để thực hiện song song một công việc hết sức đơn giản cùng với một công việc chính.  
  
Bạn viết một ứng dụng Java trên bất kỳ nền tảng nào. Khi ứng dụng bạn chạy thì thực sự đã có một bản sao của JVM khởi động và ứng dụng của bạn là một thread nếu bạn không dùng multithreading hoặc là nhiều threads nếu bạn dùng multithreading.  
Tạo một thread  
  
Như đã nói, mỗi khi chạy một ứng dụng trong java thì bạn đã có một thread. Đây là thread chính, nó thực thi các dóng lệnh trong method : public static void main . Đây là một điểm nhập bắt buộc cho mọi ứng dụng độc lập.  
  
  
Để tạo ra một thread khác ngoài thread chính trên, Java cung cấp cho chúng ta hai cách :  
  
- Tạo ra một lớp con của lớp Thread (java.lang.Thread)  
  
- Tạo ra một lớp hiện thực interface Runnable  
  
Chúng ta sẽ tìm hiểu lần lược hai cách trên.  
  
Tạo một lớp con của lớp java.lang.Thread  
  
Bạn khai báo như sau :  
  
class A extends Thread {  
  
public void run() {  
  
... // code for the new thread to execute  
  
}  
  
}  
  
...  
  
A a = new A(); // create the thread object  
  
a.start(); // start the new thread executing  
  
...  
  
  
Với cách này các dòng lệnh của bạn sẽ được đặt trong method run. Method này được override method nguyên thuỷ của lớp Thread.  
  
  
Sau đó bạn sẽ tạo ra một đối tượng từ lớp của bạn.  
  
Bạn gọi phương thức start từ đối tượng đó. Lúc này thread của bạn chính thức được tạo ra và phương thức start sẽ tự gọi method run của bạn và thực thi các dòng lệnh mà bạn đã đặt tả.  
  
Chú ý rằng: method start là method của hệ thống, nó có nhiệu vụ cấp phát bộ nhớ, tạo ra một thread và gọi hàm run của bạn. Vì thế bạn không nên override phương thức này. Điều này có thể dẫn đến ko tạo được thread.  
Hiện thực interface Runnable  
  
Bạn khai báo như sau:  
  
  
class B extends … implements Runnable {  
  
public void run() {  
  
... // code for the new thread to execute  
  
}  
  
}  
  
...  
  
B b = new B(); // create the Runnable object  
  
Thread t = new Thread(b); // create a thread object  
  
t.start(); // start the new thread  
  
...  
  
Cũng giống như cách trên, dòng lệnh của bạn đặt trong method run (có thể gọi đến các phương thức khác, nhưng phải bắt đầu trong phương thức này)  
  
Sau đó bạn tạo một đối tượng B từ lớp đã hiện thực interface Runnable, tạo thêm một đối tượng t của lớp Thread với thông số cho constructor là đối tượng B.  
  
Sau đó khi bạn gọi phương thức t.start() thì chính thức thread được tạo ra và phương thức run của bạn sẽ được triệu gọi một cách tự động.  
  
Bạn sẽ hỏi tại cách thứ hai vẫn phải tạo ra một đối tượng Thread. Vậy tại sao lại đưa ra hai cách hiện thực làm gì ?  
  
Câu trả lời là :  
  
- Bản thân ngôn ngữ Java không hỗ trợ đa thừa kế . Bạn chỉ có thể extends từ một lớp duy nhất. Nhưng bạn lại có thể implements cùng lúc nhiều interface. Khi mà lớp của bạn đã [extends] một lớp nào đó rồi (vd : Applet), thì bạn chỉ có thể implements Runnable để tạo ra Thread.  
  
- Việc extends lớp Thread có thể dẫn đến rủi ro là bạn override các method start, stop, ... thì có thể làm cho việc tạo thread là không thể.  
  
Một lời khuyên là: bạn nên tạo ra một lớp hiện thực interface Runnable (cách thứ hai) khi muốn tạo ra một Thread. Chương trình của bạn sẽ trong sáng và dễ tìm lỗi hơn.  
  
  
Các constructor cho lớp Thread  
  
public Thread ()  
  
- Tạo ra một đối tượng lớp Thread.  
  
- Constructor này có tác dụng giống như Thread(null, null, gname) với gname là một tên được phát sinh tự động bởi JVM (máy ảo java) ở dạng Thread-n , với n là một số nguyên.  
  
public Thread(Runnable target)  
  
- Tạo ra một đối tượng lớp Thread.  
  
- Constructor này có tác dụng giống như Thread(null, target, gname) với gname là tên được phát sinh tự động bởi JVM (máy ảo java) ở dạng Thread-n , với n là một số nguyên.  
  
public Thread(ThreadGroup group, Runnable target)  
  
- Tạo ra một đối tượng lớp Thread.  
  
- Constructor này có tác dụng giống như Thread(group, target, gname) với gname là tên được phát sinh tự động bởi JVM (máy ảo java) ở dạng Thread-n , với n là một số nguyên.  
  
- Constructor này có thể phát sinh một ngoại lệ SecurityException nếu như thread hiện tại không có quyền tạo một thread mới trong nhóm group.  
  
public Thread(String name)  
  
- Tạo ra một đối tượng lớp Thread.  
  
- Constructor này có tác dụng giống như Thread(null, null, name).  
  
public Thread(ThreadGroup group, String name)  
  
- Tạo ra một đối tượng lớp Thread.  
  
- Constructor này có tác dụng giống như Thread(group, null, name).  
  
- Constructor này có thể phát sinh một ngoại lệ SecurityException nếu như thread hiện tại không có quyền tạo một thread mới trong nhóm group.  
  
public Thread(Runnable target, String name)  
  
- Tạo ra một đối tượng lớp Thread.  
  
- Constructor này có tác dụng giống như Thread(null, target, name).  
  
public Thread(ThreadGroup group, Runnable target, String name)  
  
- Tạo ra một đối tượng lớp Thread.  
  
- group là nhóm các thread mà thread được tạo mới này sẽ được thêm vào.  
  
- target là đối tượng cần chạy. Tức là trong đối tượng này có method run, chứa các dòng lệnh cần thực thi của chúng ta.  
  
- name là tên của thread.  
  
- Nếu group là null và có một bộ quản lý bảo mật security manager(ta thiết lập cho JVM, mặc định thì không có) thì gruop được xác định bởi method getThreadGroup của security manager. Nếu group null và không có security manager hay phương thức getThreadGroup trả về null thì group là nhóm mà thread đang hoạt động thuộc về.  
  
- Nếu target là null thì sau khi được khởi hoạt, method run của lớp Thread sẽ được gọi. Nếu target không null thì sau khi được khởi hoạt thì method run của target sẽ được gọi.  
  
- Độ ưu tiên (priority) của thread mới sẽ được thiết lập bằng với độ ưu tiên của thread tạo ra nó.  
  
- Constructor này có thể phát sinh một ngoại lệ SecurityException nếu như thread hiện tại không có quyền tạo một thread mới trong nhóm group.  
  
Chi tiết hơn bạn có thể xem ở Java API của bộ JDK  
  
  
Một số phương thức chính của lớp Thread  
  
  
public static Thread currentThread()  
  
- Phương thức này là một phương thức tĩnh. Có nghĩa là bạn có thể gọi nó ở bất đâu mà không cần phải tạo một đối tượng của lớp Thread.  
  
- Cú pháp để gọi phương thức này là : Thread.currentThread()  
  
- Phương thức này trả về một đối tượng của lớp Thread. Đối tượng này thể hiện cho thread mà đang thực thi dòng lệnh này.  
  
public long getId()  
  
- Phương thức này trả về một số nguyên là định danh của thread.  
  
- Con số này là do máy ảo Java (JVM) tạo ra.  
  
public final String getName()  
  
- Phương thức này trả về tên của thread.  
  
public final void setName(String name)  
  
- Thay đổi tên của thread bằng tên name.  
  
public Thread.State getState()  
  
- Trả về trạng thái hiện tại của thread.  
  
- Là một hằng của lớp Thread.  
  
- Các giá trị có thể là :  
  
+ NEW : thread chưa được khởi hoạt.  
  
+ RUNNABLE : thread đang hoạt động trong JVM.  
  
+ BLOCKED : thread chờ một monitor để unlock một đối tượng mà nó cần.  
  
+ WAITING : thread chờ không giới hạn cho đến khi một thread khác đánh thức nó.  
  
+ TIMED\_WAITING : thread chờ trong một thời gian nhất định, hoặc là có một thread khác đánh thức nó.  
  
+ TERMINATED : thread đã kết thúc công việc của nó.  
  
public static void yield()  
  
- Đây là một phương thức tĩnh.  
  
- Phương thức này sẽ làm cho thread hiện thời tạm ngưng hoạt động để nhường cho một thread khác hoạt động.  
  
public static void sleep(long millis) throws InterruptedException  
  
public static void sleep(long millis, int nanos) throws InterruptedException  
  
- Đây là một phương thức tĩnh.  
  
- Phương thức này làm cho thread hiện tại ngừng hoạt động trong một thời gian millis milliseconds.  
  
- Phương thức này có thể phát sinh một ngoại lệ InterruptedException nếu có một thread nào đó làm ngưng hoạt động thread này. (Nhưng không hợp lệ, tức là không dùng các phương thức được cho phép).  
  
public void start()  
  
- Phương thức này chính thức tạo ra một đối tượng thread. Và JVM sẽ tự động gọi phương thức run của đối tượng này nếu nhưng target là null. Hoặc sẽ gọi phương thức run của target nếu như target khác null.  
  
- Tuyến được bắt đầu hoạt động sau khi phương thức này được gọi thành công.  
  
- Phương thức này có thể phát sinh một ngoại lệ IllegalThreadStateException nếu thread này đã được khởi hoạt.  
  
public final boolean isAlive()  
  
- Phương thức này kiểm tra xem thread có còn hiệu lực hay không (còn sống).  
  
- Một thread còn sống khi mà nó đã được khởi hoạt và chưa kết thúc công việc của nó.  
  
public final ThreadGroup getThreadGroup()  
  
- Trả về nhóm Thread mà thread này thuộc về  
  
public final void join(long millis) throws InterruptedException  
  
public final void join(long millis, int nanos) throws InterruptedException  
  
public final void join() throws InterruptedException  
  
- Đây là phương thức được gọi bởi một thread khác.  
  
- Phương thức này làm cho thread gọi phải ngưng hoạt động và chờ trong một khoảng thời gian millis millisecond hoặc chờ trong khoảng thời gian millis millisecond và nanos nanosseconds, hoặc chờ cho đến khi thread này kết thúc thì mới tiếp tục hoạt động.  
  
- Phương thức này có thể phát sinh một ngoại lệ InterruptedException nếu nó bị một thread khác làm ngưng hoạt động.  
  
public String toString()  
  
- Trả về một thể hiện dạng String của thread này.  
  
- Thể hiện này bao gồm tên, độ ưu tiên và nhóm.  
  
public final void stop()  
  
public final void stop(Throwable obj)  
  
public void destroy()  
  
public final void suspend()  
  
public final void resume()  
  
- Các methods này đều được khuyến cáo là không nên dùng. Sẽ bị bỏ trong các phiên bản sau của bộ JRE và JDK.  
  
- Bởi vì các methods này làm cho các monitor của hệ thống (được JVM tạo ra để giám sát các đối tượng của người dùng) bị giải phóng. Làm cho các đối tượng không được thống nhất và dẫn đến các lỗi không mong muốn, không an toàn khi sử dụng.  
  
Tập tin đính kèm DemoThreads.jar (19921 bytes):  
  
<http://www.javavietnam.org/javavn/mvnforum/getattachment?attach=1535>  
  
Tập tin đính kèm DemoThreads.zip (26938 bytes):  
  
<http://www.javavietnam.org/javavn/mvnforum/getattachment?attach=1536>  
  
Ở phần trước , mình đã giới thiệu sơ qua về Thread và cách tạo một Thread. Bây giờ mình sẽ giới thiệu thêm một số vấn đề liên quan khi lập trình với Java Threads.  
Thread Group  
  
- ThreadGroup là một lớp trong gói java.lang.  
  
- Lớp này cho phép chúng ta quản lý một nhóm các Threads cho ứng dụng của chúng ta.  
  
- Các threads được gom thành một nhóm thông thường dựa trên một mối quan hệ nào đó giữa chúng.Vd như: người tạo ra chúng, chức năng chính của chúng, hay khi nào chúng sẽ được start hay stop.  
Tạo một ThreadGroup  
  
ThreadGroup group = new ThreadGroup( "A group”);  
  
Thread one = new Thread(group, “one");  
  
Thread two = new Thread(group, "two");  
  
...  
  
  
Một số phương thức liên quan  
  
public int activeCount()  
  
- Trả về số thread hiện tại có trong ThreadGroup.  
public final int getMaxPriority()  
  
- Trả về độ ưu tiên cao nhất của các threads trong nhóm.  
  
- Một thread thuộc nhóm này thì không thể có độ ưu tiên cao hơn độ ưu tiên cao nhất của nhóm.  
  
public final void setMaxPriority(int pri)  
  
- Thiết lập độ ưu tiên cao nhất cho nhóm.  
  
- pri phải nằm trong khoảng Thread.MIN\_PRIORITY Thread.MAX\_PRIORITY.  
  
public final void checkAccess()  
  
- Kiểm tra xem thread đang chạy có quyền thay đổi nhóm thread này hay không.  
  
public final boolean isDaemon()  
  
- Kiểm tra xem nhóm có phải là nhóm các Daemon Thread hay không.  
  
- Daemon thread sẽ được đề cập tới.  
  
public final void setDaemon(boolean daemon)  
  
- Thiết lập nhóm thread là Daemon Thread.  
  
  
Chi tiết hơn bạn có thể xem ở Java API  
  
  
Thread Priority  
  
- Mỗi một thread trong java đều có một độ ưu tiên nhất định.  
  
- Độ ưu tiên này nằm trong khoảng Thread.MIN\_PRIORITY (0) và MAX\_PRIORITY (10).  
  
- Độ ưu tiên của một thread được tạo ra sẽ mặc định là bằng với độ ưu tiên với thread tạo ra nó.  
  
- Việc thiết lập cũng như lấy về độ ưu tiên của một thread được thực hiện thông qua hai phương thức:  
  
+ public final void setPriority(int newPriority)  
  
+ public final int getPriority()  
Daemon Thread  
  
- Mọi thread đều có thể là Daemon Thread.  
  
- Daemon threads là các threads có nhiệm cụ cung cấp dịch vụ cho các threads khác cùng hoạt động trong cùng một process.  
  
- Một điều quan trọng là : máy ảo JVM sẽ tự kết thúc hoạt động nếu như tất cả các thread trong ứng dụng đều là Daemon Threads.  
  
Một câu hỏi đặt ra là : Tại sao lại cần Daemon Thread trong khi thread bình thường vẫn có thể cung cấp dịch vụ cho thread khác?  
  
Trả lời :  
  
- Ví dụ bạn có một thread A có nhiệm vụ cung cấp dịch vụ trao đổi dữ liệu với một máy khác từ xa thông qua mạng internet. Khi đó thread A phải có nhiệm vụ chờ yêu cầu trao đổi dữ liệu từ một thread nào đó. Vậy bạn không biết lúc nào sẽ có yêu cầu trao đổi dữ liệu, khi nào sẽ không còn yêu cầu nào nữa.  
  
- Khi thiết lập A là Daemon thread thì khi chắc chắn không còn yêu cầu nào nữa, tức là không còn thread nào sử dụng dịch vụ này nữa. JVM sẽ kết thúc hoạt động của thread A và kết thúc hoạt động của ứng dụng.  
  
- Để thiết lập và kiểm tra một thread có phải là Daemon Thread hay không bạn có thể dùng :  
  
+public final void setDaemon(boolean on)  
  
+ public final boolean isDaemon()  
  
  
Synchronization  
  
- Khi tạo ra nhiều thread hoạt động song song độc lập (tức là không có dữ liệu chia sẻ) thì không có vấn đề gì.  
  
- Tuy nhiên khi hai hay nhiều thread cùng sử dụng chung một biến hay một phương thức thì : bởi vì các thread trong một Process có thể được thực thi đồng thời (hệ thống nhiều CPU) hoặc được luân phiên gọi trong một thời gian rất ngắn (hệ thống đơn CPU) cho nên việc sử dụng dữ liệu chia sẻ là gần như đồng thời. Điều này dẫn đến vấn về : Race condition.  
  
- Race condition (tạm dịch là điều kiện tranh chấp) : xảy ra khi có hai hày nhiều thread cùng chia sẻ dữ liệu, và khi chúng cùng đọc , cùng ghi dữ liệu chia sẻ đó đồng thời. Kết quả nhận được sẽ phụ thuộc vào tác động lên biến chia sẻ là cái gì và lúc nào. Và người lập trình không thể biết được chính xác điều gì xảy ra.  
  
- Java đưa ra một cách giải quyết là dùng từ khoá synchronized cho phương thức hay đoạn mã mà bạn cần bảo vệ .  
  
Synchronized Method  
  
Bạn khai báo như sau :  
  
public synchronized void protectedMethod(object obj)  
  
- Method protectedMethod sẽ được đảm bảo thực thi tuần tự.  
  
- Nghĩa là khi có 2 thread A và B gọi method protectedMethod thì nếu method A gọi trước thì khi nào method protectedMethod thực thi xong thì thread B mới được gọi. Và trong thời gian đó thì thread B phải chờ.  
  
Ví dụ:  
  
class Test {  
  
static int i = 0, j = 0;  
  
static void one() { i++; j++; }  
  
static void two() {  
  
System.out.println("i=" + i);  
  
//Do something  
  
System.out.println("j=" + j);  
  
}  
  
}  
  
  
- Khi thread A gọi method one. Thread B gọi method two.  
  
- Nhưng ở khoảng giữ hai dòng lệnh xuất method B làm một công việc gì đó khá lâu. Trong khoảng thời gian đó method A có thể được gọi nhiều lần.  
  
- Khi đó giá trị i và j được xuất ra là không theo ý bạn. Tức là giá trị j sẽ lớn hơn giá trị i. Bạn có "cảm giác" j được cộng trước i ( thực tế i vẫn được cộng trước).  
  
- Để đảm bảo i và j được xuất ra có giá trị luôn bằng nhau. Ta dùng từ khoá synchronized như sau  
  
class Test {  
  
static int i = 0, j = 0;  
  
static synchronized void one() { i++; j++; }  
  
static synchronized void two() {  
  
System.out.println("i=" + i);  
  
//Do something  
  
System.out.println("j=" + j);  
  
}  
  
}  
  
  
- Khi đó cho dù method two thực hiện một công việc lâu đến thế nào thì method one sẽ không được gọi lần nữa cho đến khi method two thực hiện xong.  
  
- Sở dĩ ta có kết quả như vậy và bởi vì khi method two được gọi thì từ khoá synchronized se khiến cho đối tượng của lớp Test bị khoá. Khi đó một thread khác không thể sử dụng đối tượng này nữa (gọi method two hay one) và thread này phải chờ.  
  
  
Synchronized Statements  
  
  
- Bạn khai báo như sau :  
  
  
synchronized (someObject) {  
  
// this code will execute in one thread at a  
  
//time and someObject will be locked.  
  
}  
  
  
- Đối tượng someObject sẽ bị khoá và đoạn mã trong khối này sẽ được thưc thi tuần tự. Tuần tự ở đây có nghĩa là nếu có 2 thread cùng thực thi đoạn mã trên thì nếu A đến trước thì A thực thi xong thì B mới được thực thi.  
  
- Chú ý rằng chỉ có đoạn mã trong khối này mới được thực thi tuần tự (tuần tự theo nghĩa trên).  
  
Wait and Notify  
  
- Như bạn đã biết bản thân ngôn ngữ Java hỗ trợ multithreading. Điều này được thấy rõ ràng nhất thông qua việc lớp Object có các phương thức wait() , notify() và notifyAll().  
  
- Như phần trước đã nói từ khoá synchronized sẽ làm cho một đối tượng bị khoá và các thread khác phải chờ nếu muốn sử dụng đối tượng này.  
  
- Vậy nếu thread A đã khoá đối tượng obj nhưng trong quá trình xử lý của nó A cần obj được cập nhật bởi một thread B khác, rồi dựa trên giá trị này A mới xử lý tiếp.  
  
- Phương thức wait() cho phép ta làm điều này.  
  
- wait() làm cho thread hiện tại (thread đang khoá đối tượng obj) phải tạm ngưng hoạt động, đồng thời đối tượng obj tạm thời được mở khoá và cho phép các thread khác trong hàng đợi sử dụng obj được hoạt động.  
  
- notify() sẽ làm cho một thread đang chờ kết quả từ obj (thread đã gọi wait()) sẽ được thực thi tiếp (tiếp tục khoá obj). Với phương thức notify() thì thread được chọn đánh thức là ngẫu nhiên tuỳ ý. do đó có thể dẫn đến deadlock.  
  
- notifyAll() sẽ làm cho tất cả các thread chờ được hoạt động (do obj bị khoá) sẽ đều có cơ hội hoạt động. Điều này sẽ giảm thiểu rủi ro cho việc chọn ngẫu nhiên một thread trong hàng đợi.  
  
Trên đây là một số vấn đề khi lập trình Java Thread  
------------------------  
Theo JavaVietNam