Java Thread – Games Series

Amru Rosyada
taka86@gmail.com
http://amrurosyada.blogspot.com

Multithreading sangat diperlukan dalam games engineering, karena dalam sebuah games diperlukan adanya pemrosesan yang berjalan bersama2 misalnya ketika kita membuat games realtime strategi seperti red alert, berapa banyak thread yang dibutuhkan ??, apalagi kalo membuat games seperti football manager (my favourite) dimana setiap club bahkan setiap pemain akan mempunyai perubahan perilaku, skill, usia, mood, technique dll dalam siklus tertentu. Dalam dunia nyata-pun tak dapat dipisahkan, simpelnya saja nih ... ketika kita browsing browser membuka banyak tab dan tiap2 tab mengkses halaman web yang berbeda itu juga merupakan proses multithreading, sedangkan disisi server seperti webserver akan membuat thread jika ada request yang masuk.

Pendahuluan

Dalam java untuk membuat thread ada 3 cara :

1. Menurunkan class Thread (Extends)
2. Mengimplementasikan interface Runnable
3. Menggunakan anonymous inner class

Isi

Sekarang kita akan bahas satu persatu bagaimana ketiga cara itu digunakan :
ex, dengan menurunkan class thread :

```
public class MyThread extends Thread {
            public void run() {
            System.out.println("Ini threading lho !!!");
      }
}
Thread mythread = new MyThread();
mythread.start();
```

ex, dengan mengimplementasikan interface Runnable:
dengan menggunakan cara ini mempunyai keuntungan class yang kita buat masih bisa menurunkan class yang lain, karena java tidak mengijinkan multiple inharitance

```
public class MyThread extends SomeOthersClass implements Runnable {
        public MyThread() {

    }
    public void run() {
        System.out.println("threading pake implementd Runnable");
    }
}

Thread mythread = new Thread(new MyThread);
mythread.start();
```

ex, menggunakan anonymous class:
dengan ini kita tidak perlu untuk menurunkan class Thread dan mengimplementasikan interface Runnable.

```
new Thread() {
        public void run() {
        System.out.println("menggunakan anonymous class");
    }
}.start();
```

keruagian menggunakan cara ini adalah membuat code yang kita bikin menjadi sulit untuk dibaca dan dipahami.
gunakan method join agar thread yang kita gunakan masuk keantrian dan mengunggu sampai thread yang lain selesai.

```
myhtread.join();
```

method ini sangat berguna ketika kita membuat games dan player ingin keluar dari permainan, untuk memastikan bahwa semua thread telah selesai sebelum menjalankan cleanup. Kita juga melakukan sleep untuk thread yang sedang berjalan dengan presisi waktu dalam milidetik.

```
mythread.sleep(1000);
```

Sinkronisasi
Misalkan kita mau membuat maze game (games untuk mencari jalan keluar). Thread manapun bisa mengubah posisi pemain/player, dan thread manapun bisa melakukan pemeriksaan apakah ada pemain yang sudah menemukan pintu keluar. Untuk mempermudah mari kita lihat pada ilustrasi berikut, kita asumsikan bahwa jalan keluar/exit berada di x=0, y=0 :

```
public class Maze {
        private int playerX;
```

```
	private int playerY;

	public boolean isAtExit() {
		return (playerX==0 && playerY==0);
	}

	public void setPosition(int x, int y) {
		playerX=x;
		 playerY=y;
	}
}
```

secara garis besar code diatas tidak akan bermasalah, tapi bagai mana jika terjadi preemtive (banyak thread yang mengakses dan mana yang didahulukan untuk dapat mengubah resource). Misalkan kita ambil sekenario sebagai berikut, pamain berpindah tempat dari (1,0) ke (0,1):

1. Dimulai dari posisi (1,0), variabel playerX=1 dan playerY=0
2. Thread A memanggil setPosition(0,1)
3. Ketika line playerX=x dieksekusi maka playerX bernilai 0
4. Tiba-tiba Thread B melakukan pengecekan pada isAtExit() sebelum A sempat mengubah nilai playerY maka B akan mendapatkan kembalian bernilai true, karena playerX dan playerY sedang dalam keadaan yang sama yaitu bernilai 0.

sekarang kita akan melakukan sinkronisasi untuk mencegah terjadinya hal diatas. Kodenya akan berubah menjadi sebagai berikut :

```
public class Maze {
		private int playerX;
	private int playerY;
	public synchronized boolean isAtExit() {
		return (playerX == 0 && playerY == 0);
	}
	public synchronized void setPosition(int x, int y) {
		playerX = x;
		 playerY = y;
	}
}
```

ketika JVM mengeksekusi method yang beratribut sinchronized, maka akan terjadi ackquire lock pada method tersebut dan hanya akan mengijinkan untuk dieksekusi oleh satu object pada suatu waktu.
Jadi jika suatu sinchronized method belum selesai dieksekusi maka method sinchronized lain tidak akan bisa dieksekusi.

```
public synchronized void setPosition(int x, int y) {
		playerX = x;
	playerY = y;
}
```

code diatas bisa juga ditulis dalam bentuk berikut:

```
public void setPosition(int x, int y) {
        synchronized(this) {
        playerX = x;
         playerY = y;
    }
}
```

pada code mempunyai jumlah bytecode yang lebih banyak. Sinkronisasi object seperti kode kedua diatas berguna jika kita mengijinkan lebih dari satu lock, dan tidak membutuhkan untuk sinkronisasi untuk methodnya.

Lock bisa dilakukan pada object apapun kecuali pada tipe primitive, berikut adalah contoh bagaimana lock dilakukan pada object :

```
Object myLock = new Object();
        ...
synchronized (myLock) {
        ...
}
```

Saat bagaimanakah diperlukan sinkronisasi ?
Jawabannya adalah setiap waktu ketika ada dua atau lebih thread melakukan akses pada suatu object/field.

Hal yang perlu diingat jangan pernah melakukan oversinkronisasi (melakukan sinkronisasi pada object/method/field yang telah disinkronisasi). Sebagai contoh jangan lakukan sinkronisasi pada sebuah method jika hanya field tertentu saja yang akan disinkronisasi dalam method tersebut. Sebagai contoh method berikut, lakukan sinkronisasi pada block yang diperlukan saja.

```
public void myMethod() {
        synchronized(this) {
                // code that needs to be synchronized
        }
                // code that is already thread-safe
}
```

jangan lakukan sinkronisasi pada method yang hanya menggunakan local variable, karena local variable akan ditaruh di stack, sedangkan thread punya stack untuk tiap2 thread, jadi tidak perlu untuk dilakukan sinkronisasi. Berikut adalah contoh method yang tidak perlu dilakukan sinkronisasi karena hanya menggunakan variable local.

```
public int square(int n) {
        int s = n * n;
        return s;
}
```

jika kita tidak yakin thread mana yang sedang mengakses kode kita, kita bisa mendapatkan nama dari thread tersebut dengan :

```
Thread.currentThread().getName();
```

Perlu diwaspadai juga adanya deadlock, deadlock adalah adanya thread yang tidak bisa melanjutkan proses karena thread saling menunggu thread lain sampai melepaskan resource. Misalkan pada contoh berikut :
1. Thread A acquire lock 1.
2. Thread B acquire lock 2.
3. Thread B menunggu sampai lock 1 dilepaskan.
4. Thread A menunggu sampai lock 2 dilepaskan.

Dapat kita lihat dari process diatas kedua thread saling menunggu samapai suatu waktu yang tidak bisa ditentukan. Kita bisa mengatasinya dengan cara melakukan sinkronisasi yang tepat sesuai dengan urutannya.

Menggunakan wait() dan notify()
Misalkan kita ambil sekenario sebagai berikut, ada dua thread yang akan saling berkomunikasi satu sama lain, sebagai contoh ada thread A menunggu sampai thread B mengirimkan pesan :

```
// Thread A
public void waitForMessage() {
        while (hasMessage == false) {
        Thread.sleep(100);
        }
}
// Thread B
public void setMessage(String message) {
        ...
        hasMessage = true;
}
```

kode diatas bukan suatu contoh yang baik, karena thread A melakukan pengecekan setiap 100 milisecond atau 10 kali dalam satu detik. Thread A dapat oversleep dan terlambat dalam mendapatkan pesan.
Alangkah lebih baik jika A idle sampai ada notifikasi dari B bahwa pesan sudah bisa dikonsumsi, dan ini bisa dilakukan dengan pasangan method wait() dan notify().

Method wait() digunakan didalam blok synchronized. Ketika method wait() dieksekusi, lock akan dilepaskan dan menunggu sampai ada notifikasi.
Method notify() juga digunakan didalam block synchronized. Method notify akan memberikan notifikasi pada thread yang menunggu pada lock yang sama. Jika ada banyak thread yang menunggu maka hanya akan ada satu notifikasi dan akan dipilih satu thread secara acak. Berikut adalah kode yang telah diperbaiki :

```
// Thread A
public synchronized void waitForMessage() {
        try {
        wait();
        }
        catch (InterruptedException ex) { }
        }
        // Thread B
public synchronized void setMessage(String message) {
```

```
            ...
            notify();
}
```

jika kita ingin memberika notifikasi untuk semua thread yang sendang menunggu kita bisa menggunakan notifyAll(), method wait() juga menerima parameter dalam milisecond sebagai waktu tunggu, misalnya kita ingin memberikan timeout sampai 100milisecond maka kita bisa menggunakan wait(100).
Method wait(), notify(), dan notifyAll() merupakan method dari class object, sehingga semua java object mempunyai method2 tersebut.

Kapan kita seharusnya menggunakan thread ?
Jadi begini dari pendekatan games, ketika games play loading untuk kenyamanan pengguna sebaiknya ketika loading dibuatkan thread sendiri sehingga player tidak menyangka bahwa gamesnya sedang ngehang. Kalo dari pendekatan lain sebenarnya juga untuk kenyamanan dan optimasi, nyaman untuk pengguna karena pengguna merasa menggunakan program yang cepet loadingnya (tricky), optimal karena bisa memanfaatkan resource CPU yang belum termanfaatkan.

Sum it up
Oke dengan informasi thread yang telah dibahas sebelumnya, mari kita buat sesuatu yang berguna yaitu thread pool. Thread pool merupakan sebuah group dari thread yang didesain untuk mengeksekursi tugas yang bermacam-macam. Pada thread pool kita bisa memilih jumlah dari thread didalam pool dan menjalankan task yang didefinisikan sebagai Runnable. Berikut adalah contoh menggunakan ThreadPool dengan membuat 8 thread dalam pool, menjalankan task sederhana dan kemudian menunggu samapai task selesai dijalankan.

```
ThreadPool myThreadPool = new ThreadPool(8); myThreadPool.runTask(new Runnable() {
            public void run() {
            System.out.println("Do something cool here.");
            }
});
myThreadPool.join();
```

Method runTask() akan dijalankan. Jika semua thread di dalam pool sedang sibuk menproses task, ketika memanggil runTask() akan memasukkan task kedalam antrian sampai ada thread yang mengeksekusinya. Berikut adalah kode ThreadPool.java :

```
import java.util.LinkedList;
/**
A thread pool is a group of a limited number of threads that are used to execute tasks.
*/

public class ThreadPool extends ThreadGroup {
            private boolean isAlive;
            private LinkedList taskQueue;
            private int threadID;
            private static int threadPoolID;

            /**
            Creates a new ThreadPool.
```

```
@param numThreads The number of threads in the pool.
*/

public ThreadPool(int numThreads) {
        super("ThreadPool-" + (threadPoolID++));
        setDaemon(true);
        isAlive = true;
        taskQueue = new LinkedList();
        for (int i=0; i<numThreads; i++) {
                new PooledThread().start();
        }
}

/**
Requests a new task to run. This method returns
immediately, and the task executes on the next available
idle thread in this ThreadPool.
<p>Tasks start execution in the order they are received.
@param task The task to run. If null, no action is taken.
@throws IllegalStateException if this ThreadPool is already closed.
*/

public synchronized void runTask(Runnable task) {
        if (!isAlive) {
                throw new IllegalStateException();
        }
        if (task != null) {
                taskQueue.add(task);
                notify();
        }
}

protected synchronized Runnable getTask() throws InterruptedException
{
        while (taskQueue.size() == 0) {
                if (!isAlive) {
                        return null;
                }
                wait();
        }
        return (Runnable)taskQueue.removeFirst();
}

/**
Closes this ThreadPool and returns immediately.
All threads are stopped, and any waiting tasks are not executed.
Once a ThreadPool is closed, no more tasks can be run on this ThreadPool.
*/

public synchronized void close() {
        if (isAlive) {
                isAlive = false;
                taskQueue.clear();
                interrupt();
        }
```

```
	}

	/**
	Closes this ThreadPool and waits for all running threads to finish.
	Any waiting tasks are executed.
	*/

	public void join() {
		// notify all waiting threads that this ThreadPool is no
		// longer alive
		synchronized (this) {
			isAlive = false;
			notifyAll();
		}

		// wait for all threads to finish
		Thread[] threads = new Thread[activeCount()];
		int count = enumerate(threads);
		for (int i=0; i<count; i++) {
			try {
				threads[i].join();
			} catch (InterruptedException ex) { }
		}
	}

	/**
	A PooledThread is a Thread in a ThreadPool group,
	designed to run tasks (Runnables).
	*/
	private class PooledThread extends Thread {
		public PooledThread() {
			super(ThreadPool.this, "PooledThread-" + (threadID++));
		}

		public void run() {
			while (!isInterrupted()) {
				// get a task to run
				Runnable task = null;
				try {
					task = getTask();
				} catch (InterruptedException ex) { }
				// if getTask() returned null or was interrupted,
				// close this thread by returning.
				if (task == null) {
					return;
				}
				// run the task, and eat any exceptions it throws
				try {
					task.run();
				} catch (Throwable t) {
					uncaughtException(this, t);
				}
			}
		}
	}
```

```
}
```

Sekarang kita akan mencoba untuk melakukan test pada ThreadPool class, berikut adalah kode untuk melakukan test yaitu ThreadPoolTest class.
Berikut adalah cara untuku menjalankan ThreadPoolTest :
java ThreadPoolTest 8 4

8 merupakan jumlah task yang akan dijalankan, 4 adalah jumlah thread yang akan dijalankan. Berikut kode ThreadPoolTest.java :

```
public class ThreadPoolTest {
	public static void main(String[] args) {
		if (args.length != 2) {
			System.out.println("Tests the ThreadPool task.");
			System.out.println(
			"Usage: java ThreadPoolTest numTasks numThreads");
			System.out.println(
			"  numTasks - integer: number of task to run.");
			System.out.println(
			"  numThreads - integer: number of threads " +
			"in the thread pool.");
			return;
		}
		int numTasks = Integer.parseInt(args[0]);
		int numThreads = Integer.parseInt(args[1]);
		// create the thread pool
		ThreadPool threadPool = new ThreadPool(numThreads);
		// run example tasks
		for (int i=0; i<numTasks; i++) {
			threadPool.runTask(createTask(i));
		}
		// close the pool and wait for all tasks to finish.
		threadPool.join();
	}

	/**
	Creates a simple Runnable that prints an ID, waits 500
	milliseconds, then prints the ID again.
	*/

	private static Runnable createTask(final int taskID) {
		return new Runnable() {
			public void run() {
				System.out.println("Task " + taskID + ": start");
				// simulate a long-running task
				try {
					Thread.sleep(500);
				} catch (InterruptedException ex) { }
				System.out.println("Task " + taskID + ": end");
			}
		};
	}
}
```

oke all thing are finished.

Penutup

Diharapkan dengan adanya artikel ini bisa membantu dalam meningkatkan kemajuan teknologi informasi di Indonesia dan mendukung suksesnya IGOS

Referensi

Http://java.sun.com
David Brackeen, Bret Barker, Laurence Vanhelsuwé, Developing Games in Java, 2003, New Riders Publishing

Biografi Penulis

Amru Rosyada. Lahir pada tanggal 22 Mei 1986, menamatkan pendidikan dasar sampai pendidikan menengah akhir di kota Ngawi kemudian terdampar di Jogja mengambil program Diploma tiga Teknik Elektro Universitas Gadjah Mada dan Sekarang masih menamatkan Strata satu di Ilmukomputer Universitas Gadjah Mada.