Développer un pool de ressources thread-safe

...en comprenant ce que l'on fait !

Olivier Croisier

Table des matières

I Présentation....3
I.1 Pré-requis....3
I.2 Introduction....3
II Mise en place de l'environnement....4
II.1 La Ressource....4
II.2 Les Consommateurs....5
II.3 L'application de test....9
III Développement du pool....10
III.1 Un pool basique....10
III.2 Pool à double liste....11
III.3 Synchronisation du pool....12
IV Amélioration du pool....13
IV.1 Attente maximale pour l'obtention d'une Ressource....13
IV.2 Création dynamique de Ressources....14
V Développement d'un Pool générique....16
V.1 Design du pool générique....16
V.2 Développement du pool générique....17
V.3 Développement d'un pool concret....21
V.4 Adaptation de l'environnement de test....22
VI Amélioration du pool avec les génériques Java5....24
VI.1 Développement d'un pool générique à la mode Java5....24
VI.2 Développement d'un pool concret dérivant du générique....26
VII Conclusion....27

Article rédigé sous OpenOffice.org 2.0 / Linux

I Présentation

I.1 Pré-requis

Cet article s'adresse à un public maîtrisant le langage Java. Une bonne connaissance de la version 5 est un plus.

La compilation et l'exécution du code source nécessitent la machine virtuelle Java5, disponible sur le site de Sun Microsystems : http://java.sun.com/j2se/1.5.0/download.jsp

I.2 Introduction

Il arrive que, dans le cadre d'un projet, on ait besoin de pooler des ressources. L'exemple qui vient immédiatement à l'esprit concerne les connexions aux bases de données; mais le système peut se révéler intéressant pour toutes les ressources disponibles en nombre limité ou ayant un coût de création élevé.

Dans sa version la plus basique, un pool est un objet maintenant une simple liste d'instances pré-initialisées de la ressource concernée.

Dans sa version la plus complexe, un pool peut proposer des options comme l'ajustement entre ressources pré-initialisées et ressources créées dynamiquement, la mise en place d'un temps maximal pour obtention de la ressource, ou encore une certaine généricité.

Mais, par-dessus tout, le pool devra être thread-safe, afin de garantir que les applications y faisant appel ne tombent pas en interblocage (deadlock).

Dans cet article, nous allons développer un pool facilement adaptable à tout projet. La première version sera simple, puis nous y apporterons les améliorations indiquées plus haut, de manière à offrir une grande flexibilité d'utilisation.

Pour terminer, nous étudierons comment Java5 nous permet de développer un pool réellement générique, et sous quelles conditions.

II Mise en place de l'environnement

Pour développer et tester notre pool de ressources, nous allons avoir besoin de :

- une Ressource à pooler,
- un Consommateur de cette ressource; plusieurs seront lancés en parallèle pour tester le comportement du pool en environnement multi-threadé,
- un programme de test, pour initialiser le pool et contrôler les consommateurs.

II.1 La Ressource

En environnement réel, ce pourrait être une connexion à une base de données, ou un objet métier à la création coûteuse. Nous nous contenterons d'une simple classe-marqueur.

```
public class Resource
{
	public Resource()
	{
	}
}
```

Pour faciliter le test de notre pool, nous allons faire en sorte que chaque ressource ait un nom unique. Pour cela, une variable statique est utilisée pour compter le nombre de ressources créées, et affecter à chacune un numéro unique.

```
public class Resource
{
	private static int instanceCount = 0;
	private int num;

	public Resource()
	{
		this.num = ++Resource.instanceCount;
	}

	public String toString()
	{
		return "Resource #" + this.num;
	}
}
```

II.2 Les Consommateurs

Les Consommateurs vont simuler différents programmes accédant simultanément au pool de ressources : nous allons donc en faire des threads.

```
public class Consumer implements Runnable
{
	public Consumer()
	{
	}

	public void run()
	{
		while(true)
		{
			try { Thread.sleep(100); }
			catch (Exception e) {}
		}
	}
}
```

Comme précédemment, pour faciliter le test, nous assignerons un numéro unique à chacun des Consommateurs.

```
public class Consumer implements Runnable
{
	private static int instanceCount = 0;
	private int num;

	public Consumer()
	{
		this.num = ++Consumer.instanceCount;
	}

	public void run()
	{
		while(true)
		{
			try { Thread.sleep(100); }
			catch (Exception e) {}
		}
	}
```

```
	public String toString()
	{
		return "Consumer #" + this.num;
	}
}
```

Un Consommateur (un programme, par exemple) possède une référence sur une Ressource (une connexion à une base de données) dont il a besoin pour effectuer son traitement.

Son travail consiste donc à :

- Obtenir une Ressource auprès du pool,
- Utiliser la Ressource,
- Rendre la Ressource au pool.

Habituellement, ces opérations sont effectuées dans le cadre normal de l'exécution du code du Consommateur. Pour tester notre pool, nous avons besoin de pouvoir les déclencher manuellement, à la demande; pour cela, nous utiliserons une variable qui indiquera au Consommateur quelle action effectuer.

```
public class Consumer implements Runnable
{
	private static int instanceCount = 0;
	private int num;

	private Resource resource;

	public enum Action {	NOTHING,
				GET_RESOURCE,
				RETURN_RESOURCE,
				USE_RESOURCE };
	private Action action = Action.NOTHING;

	public Consumer()
	{
		this.num = ++Consumer.instanceCount;
	}
```

```
	public void run()
	{
		while(true)
		{
			switch (this.action)
			{
				case GET_RESOURCE :
				{
					// Get the ressource here
					this.action = Action.NOTHING;
					break;
				}
				case RETURN_RESOURCE :
				{
					// Return the ressource here
					this.action = Action.NOTHING;
					break;
				}
				case USE_RESOURCE :
				{
					// Use the ressource here
					this.action = Action.NOTHING;
					break;
				}
			}

			try { Thread.sleep(100); }
			catch (Exception e) {}
		}
	}

	public String toString()
	{
		return "Consumer #" + this.num;
	}

	public void setAction(Action action)
	{
		this.action = action;
	}
}
```

Examinons maintenant ces trois actions en détail. Notons que le pool est accédé via un singleton, et que l'action est réinitialisée après avoir été exécutée.

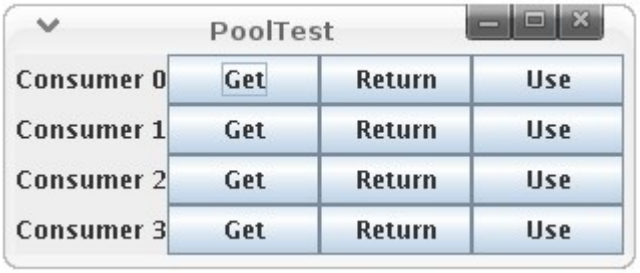
```
case GET_RESOURCE :
{
      if (this.resource == null)
      {
            System.out.println(this + " asks for a Resource.");
            this.resource = ResourcePool.getInstance().getResource();
            System.out.println(this + " received "+this.resource);
      }
      this.action = Action.NOTHING;
      break;
}
```

```
case RETURN_RESOURCE :
{
      if (this.resource != null)
      {
            System.out.println(this + " returns "+this.resource);
            ResourcePool.getInstance().returnResource(this.resource);
            this.resource = null;
      }
      this.action = Action.NOTHING;
      break;
}
```

```
case USE_RESOURCE :
{
      if (this.resource != null)
      {     System.out.println(this + " uses " + this.resource);
      }
      this.action = Action.NOTHING;
      break;
}
```

II.3 L'application de test

Pour finir, nous développerons une application permettant d'une part d'initialiser le pool de Ressources, et d'autre part de créer et piloter des Consommateurs, afin de vérifier le comportement du pool dans les conditions choisies. L'application présente une interface simple en Swing qui représente les Consommateurs.




```
public class PoolTest extends JFrame
{
	public static void main(String[] args)
	{
		new PoolTest();
	}

	public static final int NB_CONSUMERS = 4;

	public PoolTest()
	{
		Container c = getContentPane();
		c.setLayout(new GridLayout(NB_CONSUMERS,4));

		for (int i=0; i<NB_CONSUMERS; i++)
		{
			final Consumer consumer = new Consumer();
			new Thread(consumer).start();

			// Interface creation here for each Consumer
		}

		setTitle("PoolTest");
		pack();
		setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
		setVisible(true);
	}
}
```

Le code pour la création de l'interface Swing n'est pas détaillé ici.

III Développement du pool

III.1 Un pool basique

Un pool, dans sa version la plus simple, maintient une simple liste de Ressources, et propose deux fonctions pour les emprunter et les rendre. Toutes les Ressources sont initialisées dès la création du pool. Leur nombre est fixé par une constante MAX_RESOURCES (ici 2).

Afin d'éviter de passer une référence au pool à chaque Consommateur, celui-ci est implémenté sous la forme d'un singleton.

```
public class Pool
{
	static public final int MAX_RESOURCES = 2;
	private List available = new LinkedList();

	static private Pool instance = new Pool(MAX_RESOURCES);
	static public Pool getInstance()
	{	return instance;
	}

	private Pool(int nbResources)
	{
		for (int i=0; i<nbResources; i++)
		{	available.add(new Resource());
		}
	}

	public Resource getResource()
	{
		return (available.size()>0 ? available.remove(0) : null);
	}

	public void returnResource(Resource resource)
	{
		available.add(resource);
	}
}
```

III.2 Pool à double liste

Le concept de la liste simple montre ses limitations dans le cas de ressources nécessitant d'être surveillées en permanence, ou devant être proprement fermées même dans le cas où des Consommateurs peu scrupuleux « oublient » de les rendre au pool. Cela permet également de vérifier que les Ressources rendues sont bien celles qui ont été empruntées auparavant.

```
public class Pool
{
	static public final int MAX_RESOURCES = 2;
	private List available = new LinkedList();
	private List used = new LinkedList();

	static private Pool instance = new Pool(MAX_RESOURCES);
	static public Pool getInstance()
	{	return instance;
	}

	private Pool(int nbResources)
	{
		for (int i=0; i<nbResources; i++)
		{	available.add(new Resource());
		}
	}

	public Resource getResource()
	{
		Resource ressource = null;
		if(available.size() > 0)
		{
			resource = (Resource) available.remove(0);
			used.add(resource);
		}
		return resource;
	}

	public void returnResource(Resource resource)
	{
		if (! used.remove(resource))
		{
			throw new IllegalArgumentException(
				"The returned Resource is unknown.");
		}
		available.add(resource);
	}
}
```

III.3 Synchronisation du pool

Le pool étant utilisé par plusieurs threads, il faut s'assurer que les données sont correctement synchronisées.

Dans le code de getResource() par exemple, il est possible qu'un thread exécute le test available.size()>0 et se prépare à prendre la Ressource. Avant qu'il n'ait eu le temps de le faire, un autre thread passe également le test de disponibilité. Si à ce moment précis il ne reste effectivement qu'une seule Ressource, il y a un réel problème.

Avec l'introduction des mécanismes de synchronisation, un nouveau choix s'offre à nous : si un Consommateur demande une Ressource alors qu'aucune n'est disponible, on peut lui demander d'attendre que l'une d'elles se libère, au lieu de retourner une référence nulle immédiatement.

La solution présentée ici fait attendre indéfiniment le Consommateur malchanceux. Nous verrons plus loin comment mettre en place une durée d'attente maximale.

```
synchronized public Resource getResource()
{
      while(available.size() == 0)
      {
            try { wait(); }
            catch (InterruptedException iEx)
            {}
      }

      Resource resource = (Resource) available.remove(0);
      used.add(resource);
      return resource;
}

synchronized public void returnResource(Resource resource)
{
      if (! used.remove(resource))
      {
            throw new IllegalArgumentException("
                  The returned Resource is unknown.");
      }

      available.add(resource);
      notify();
}
```

Lorsqu'un Consommateur entre dans la méthode synchronisée getResource() , il prend le lock sur le pool. Si aucune Ressource n'est disponible, il se met en attente et relâche le lock, ce qui permet à d'autres Consommateurs d'exécuter getResource() et surtout returnResource().

Après avoir rendu une Ressource, le Consommateur réveille un des Consommateurs en attente (au choix de la JVM), qui peut alors la récupérer.

IV Amélioration du pool

A ce point, nous disposons d'un pool de Ressources fonctionnel, thread-safe, et testable.

L'archive Pool1.zip contient les sources commentées de toutes les classes vues jusqu'ici. Compilez et exécutez PoolTest, utilisez l'interface graphique pour demander et rendre des Ressources, et observez les messages affichés sur la console.

IV.1 Attente maximale pour l'obtention d'une Ressource

La première amélioration consistera à définir une durée d'attente maximale (paramétrable) pour l'obtention d'une Ressource. Au bout de ce temps, la méthode getResource() renverra une référence nulle si aucune Ressource n'est devenue disponible.

Pour ce faire, nous allons utiliser une version surchargée de wait(), qui prend une valeur en paramètre. Cette valeur représente le nombre de millisecondes après lesquelles le thread en attente est réveillé, en l'absence d'un notify(). A chaque réveil, on vérifie si le temps imparti (maintenant précisé en paramètre de getResource(), en secondes) est écoulé, auquel cas on renvoie une référence nulle.

```
synchronized public Resource getResource(int timeout)
{
      long time1 = System.currentTimeMillis();
      while(available.size() == 0)
      {
            try { wait(200); }
            catch (InterruptedException iEx)
            {}

            long time2 = System.currentTimeMillis();
            if ( (timeout>0) && (time2-time1 > timeout*1000) )
            {
                  System.out.println("Timeout");
                  return null;
            }
      }

      Resource resource = (Resource) available.remove(0);
      used.add(resource);
      System.out.println("[POOL] avail:" + available.size()
                        + ", used:"+used.size());

      return resource;
}
```

Afin de conserver l'interface publique du pool, on proposera également une version sans paramètre de getResource(), qui appelle la version paramétrée avec une valeur neutre.

```
synchronized public Resource getResource()
{
      return getResource(0);
}
```

Pour tester cette nouvelle fonctionnalité, on peut adapter le code du Consommateur comme suit :

```
case GET_RESOURCE :
{
      if (this.resource == null)
      {
            System.out.println(this + " asks for a Resource.");
            this.resource = Pool.getInstance().getResource(5);
            System.out.println(this + " received "+this.resource);
      }
      this.action = Action.NOTHING;
      break;
}
```

Ensuite, il suffit de demander plus de Ressources que disponible, et de constater que les Consommateurs malchanceux reçoivent une référence nulle au bout de 5 secondes si on n'a libéré aucune Ressource.

IV.2 Création dynamique de Ressources

Pour des questions d'optimisation (de mémoire notamment), il est fréquent qu'un pool ne pré-initialise qu'une certaine fraction des Ressources, les autres étant créées dynamiquement, à la demande, dans la limite de la capacité maximale du pool.

Commençons par ajouter un paramètre indiquant le nombre de Ressources à initialiser.

```
public class Pool
{
      static public final int MAX_RESOURCES = 4;
      static public final int MIN_RESOURCES = 2;

      static private Pool instance = new Pool(MIN_RESOURCES);
      ...
}
```

La condition d'évaluation de la disponibilité d'une Ressource change également. Précédemment, on vérifiait simplement que la liste des Ressource disponibles n'était pas vide. Maintenant, une liste vide ne signifie plus forcément une pénurie : si le nombre maximal de Ressources n'est pas encore atteint, on peut en créer une dynamiquement.

```
synchronized public Resource getResource(int timeout)
{
      long time1 = System.currentTimeMillis();

      while( (available.size()==0) &&
             (available.size()+used.size() >= MAX_RESOURCES) )
      {
            try { wait(200); }
            catch (InterruptedException iEx)
            {}

            long time2 = System.currentTimeMillis();
            if ( (timeout>0) && (time2-time1 > timeout*1000) )
            {     System.out.println("Timeout");
                  return null;
            }
      }

      Resource resource = null;
      if (available.size() > 0)
      {
            System.out.println("Taking an existing Resource from pool");
            resource = (Resource) available.remove(0);
      }
      else
      {
            System.out.println("Creating a new Resource");
            resource = new Resource();
      }
      used.add(resource);

      return resource;
}
```

Le code source de ce pool amélioré est disponible dans l'archive Pool2.zip.

V Développement d'un Pool générique

Notre pool actuel est conçu de manière à ne gérer que des objets d'un type spécifique (dans notre cas, du type Ressource). Si d'aventure nous avions besoin de pooler d'autres types d'objets, il nous faudrait dupliquer le code du pool et l'adapter spécifiquement au nouveau type.

Cette solution est naturellement inacceptable; nous devons donc modifier notre pool afin de le rendre plus souple.

V.1 Design du pool générique

La transformation de notre pool spécifique en pool générique soulève plusieurs problèmes intéressants; les réponses que nous y apporterons guideront le design du nouveau pool :

- Premièrement, notre singleton actuel interdit d'avoir plusieurs instances simultanées, donc de pooler plusieurs types de ressources simultanément.
- Les nombres initial et maximal de ressources disponibles ne peuvent plus être fixés par constantes au niveau du pool, mais devront plutôt être paramétrés pour chaque type de ressource poolée.
- Les fonctions getResource() et returnResource() ne pourront plus prendre en paramètre ni renvoyer des types spécifiques. Nous emploierons donc le type Object, bien que cela implique *a priori* de nombreux « casts » dans le code appelant.
- La création d'une nouvelle Ressource, que ce soit au cours de l'initialisation ou lors d'une création dynamique, pose désormais problème. En effet, son type exact étant désormais inconnu à la compilation, nous n'avons aucune information sur ses constructeurs. Il faudra donc déléguer cette tâche à un objet possédant cette connaissance.

Pour répondre à ces nouvelles contraintes, nous allons réarchitecturer notre pool sous la forme d'une classe abstraite. Ce nouveau design nous permet en effet de résoudre les différents problèmes soulevés plus haut, et offre une réelle souplesse aux pools concrets en dérivant :

- Les pools concrets seront libres d'adopter à nouveau un pattern de singleton ou non;
- Ils pourront chacun, indépendamment, définir les nombres initial et maximal de ressources à pooler;
- Les pools concrets pourront « caster » à la volée les objets génériques du pool en leur type réel, éliminant ainsi les « casts » dans le code des Consommateurs.
- La création des ressources sera déléguée aux pools concrets, qui connaissent parfaitement les constructeurs des types pour lesquels ils sont spécialisés.

V.2 Développement du pool générique

Reprenons le code de notre Pool pour y apporter les changements décrits plus haut.

Tout d'abord, supprimons le singleton, et changeons le constructeur pour qu'il soit paramétrable. Notez au passage que le constructeur n'est plus privé.

```
public class Pool
{
      protected int initialResources;
      protected int maxResources;

      public Pool(int initialResources, int maxResources)
      {
            this.initialResources = initialResources;
            this.maxResources = maxResources;

            for (int i=0; i<initialResources; i++)
            { ... }
      }

      ...
}
```

Ajoutons ensuite la méthode abstraite permettant de créer une nouvelle ressource; cette méthode devra être implémentée par les pools concrets. Elle est appelée dans le constructeur pour la création des ressources initiales.

```
public abstract class Pool
{
      protected int initialResources;
      protected int maxResources;

      public Pool(int initialResources, int maxResources)
      {
            this.initialResources = initialResources;
            this.maxResources = maxResources;

            for (int i=0; i<initialResources; i++)
            {     Object resource = createResource();
                  available.add(resource);
            }
      }

      protected abstract Object createResource();

      ...
}
```

Modifions maintenant les méthodes de pooling, afin qu'elles acceptent tous les types de ressources. La création dynamique de ressources est également confiée à la méthode createResource vue plus haut.

```
synchronized public Object getResource()
{
      return getResource(0);
}

synchronized public Object getResource(int timeout)
{
      long time1 = System.currentTimeMillis();
      while( (available.size()==0) &&
            (available.size()+used.size() >= this.maxResources) )
      {
            try { wait(200); }
            catch (InterruptedException iEx)
            {}
            ...
      }

      Object resource = null;
      if (available.size() > 0)
      {     System.out.println("Taking an existing Resource from pool");
            resource = available.remove(0);
      }
      else
      {     System.out.println("Creating a new Resource");
            resource = createResource();
      }
      used.add(resource);

      return resource;
}

synchronized public void returnResource(Object resource)
{
      if (! used.remove(resource))
      {     throw new IllegalArgumentException(
                  "The returned resource is unknown.");
      }
      available.add(resource);

      notify();
}
```

Nous en avons maintenant presque terminé avec notre pool générique. En effet, un dernier choix de design se pose.

Les méthodes ci-dessus sont déclarées « public », ce qui implique que les pools concrets en héritent également en « public », et que par conséquence les Consommateurs peuvent y accéder directement. Une solution alternative consiste à les déclarer en « protected », de manière à ce que seuls les pools concrets puissent y avoir accès. Voyons les avantages et inconvénients de chaque choix :

	Avantages	***Inconvénients***
Public	• Pas de code supplémentaire dans les pools concrets. • Dans tous les pools concrets, on trouvera les mêmes noms de méthodes (celles définies au niveau de la classe Pool)	• Les Consommateurs reçoivent des objets génériques (Object) qu'ils doivent « caster ». • Impossible d'effectuer un pré- ou post-traitement garanti sur les ressources au niveau des pools concrets, car les Consommateurs peuvent accéder directement aux méthodes de Pool.
Protected	• Les Consommateurs reçoivent et rendent des objets fortement typés, pas besoin de « casts » dans leur code. • Dans des pools concrets, les méthodes d'emprunt et restitution des ressources peuvent avoir des noms plus explicites, adaptés au type de la ressource gérée. • L'encapsulation des méthodes de Pool permet un pré- ou post-traitement des ressources, adaptés à leur type réel.	• Obligation pour les pools concrets de définir leurs propres méthodes publiques pour l'accès aux ressources.

La pertinence des arguments et le choix de la « meilleure » solution en environnement réel sont laissés à l'appréciation du lecteur. Dans le cadre de cet article, nous allons nous contenter des méthodes publiques.

V.3 Développement d'un pool concret

Maintenant que nous avons un pool générique, nous allons développer un pool spécialisé dans la gestion des objets de type Ressource, dont nous allons nous servir dans notre programme de test.

Tout d'abord, créons une nouvelle classe ResourcePool, dérivant de Pool. Elle devra appeler le constructeur à deux arguments de Pool, et implémenter la méthode createResource().

```
public class ResourcePool extends Pool
{
	public ResourcePool(int initialResources, int maxResources)
	{
		super(initialResources, maxResources);
	}

	protected Object createResource()
	{
		return new Resource();
	}
}
```

Ce simple code suffit à obtenir un pool permettant de gérer des objets de type Ressource.

Afin d'épargner aux Consommateurs la nécessité d'un « cast », nous allons implémenter des méthodes fortement typées, qui ne font qu'encapsuler des appels aux méthodes du pool générique :

```
public Resource getResource()
{
	return (Resource) super.getResource();
}

public Resource getResource(int timeout)
{
	return (Resource) super.getResource(timeout);
}

public void returnResource(Resource resource)
{
	super.returnResource(resource);
}
```

Note : nous utilisons ici les types de retour covariants, propres à Java5. Avec les versions précédentes, nous serions obligés de nommer les méthodes différemment de celles de Pool, car leurs types de retour sont incompatibles.

V.4 Adaptation de l'environnement de test

Commençons par adapter le Consommateur.

Le pool de Ressources n'étant plus un singleton, le Consommateur doit désormais recevoir en paramètre une référence représentant le pool à utiliser.

```
private ResourcePool pool;

public Consumer(ResourcePool pool)
{
      this.num = ++Consumer.instanceCount;
      this.pool = pool;
}
```

Cette référence est ensuite utilisée pour récupérer et rendre la Ressource. Notez l'absence de « cast » dans le code, grâce aux fonctions fortement typées que nous avons définies dans ResourcePool.

```
case GET_RESOURCE :
{
      if (this.resource == null)
      {
            System.out.println(this + " asks for a Resource.");
            this.resource = pool.getResource();
            System.out.println(this + " received "+this.resource);
      }
      this.action = Action.NOTHING;
      break;
}
case RETURN_RESOURCE :
{
      if (this.resource != null)
      {
            System.out.println(this + " returns "+this.resource);
            pool.returnResource(this.resource);
            this.resource = null;
      }
      this.action = Action.NOTHING;
      break;
}
```

Pour finir, adaptons le programme de test.

Puisque les Consommateurs prennent maintenant en paramètre le pool à utiliser, nous devons le créer ici.

```
public class PoolTest extends JFrame
{
	public static void main(String[] args)
	{
		new PoolTest();
	}

	public static final int NB_CONSUMERS = 6;

	public PoolTest()
	{
		ResourcePool pool = new ResourcePool(2,4);

		for (int i=0; i<NB_CONSUMERS; i++)
		{
			final Consumer consumer = new Consumer(pool);
			new Thread(consumer).start();
			...
		}
		...
	}

	// Other methods...
}
```

Le code source du pool générique et de l'environnement de test est disponible dans l'archive Pool3.zip.

VI Amélioration du pool avec les génériques Java5

VI.1 Développement d'un pool générique à la mode Java5

Notre pool est fonctionnellement « générique », c'est-à-dire qu'il doit pouvoir gérer tout type de ressource. Java5 propose une solution technique élégante pour décrire ce comportement, permettant de développer du code plus robuste et éliminant les « casts » qui sont souvent source d'erreurs.

Nous allons donc déclarer notre pool abstrait comme étant générique au sens Java5 du terme. Il manipulera désormais des objets de type E, E étant défini par les pools concrets qui en dériveront.

Le code lui-même ne subira que peu de modifications. Tout d'abord, les listes seront des « listes d'objets de type E ».

```
public abstract class Pool<E>
{
	private List<E> available = new LinkedList<E>();
	private List<E> used = new LinkedList<E>();
}
```

Ensuite, nous devons signaler aux classes dérivant de Pool (les pools concrets) qu'elles devront renvoyer des ressources de type E lorsqu'elles implémentent createResource(). Comme nous appelons cette méthode dans le constructeur, celui-ci doit aussi être modifié en conséquence :

```
public Pool(int initialResources, int maxResources)
{
	this.initialResources = initialResources;
	this.maxResources = maxResources;

	for (int i=0; i<initialResources; i++)
	{
		E resource = createResource();
		available.add(resource);
	}
}

protected abstract E createResource();
```

Enfin, les méthodes de gestion du pool subissent également une légère modification, qui consiste essentiellement à remplacer Object par E :

```
synchronized public E getResource()
{
      return getResource(0);
}



synchronized public E getResource(int timeout)
{
      long time1 = System.currentTimeMillis();
      while( (available.size()==0) &&
             (available.size()+used.size() >= this.maxResources) )
      {
            ...
      }

      E resource = null;
      if (available.size() > 0)
      {
            System.out.println("Taking an existing Resource from pool");
            resource = available.remove(0);
      }
      else
      {
            System.out.println("Creating a new Resource");
            resource = createResource();
      }
      used.add(resource);

      return resource;
}



synchronized public void returnResource(E resource)
{
      if (! used.remove(resource))
      {     throw new IllegalArgumentException(
                  "The returned resource is unknown.");
      }


      available.add(resource);


      notify();
}
```

VI.2 Développement d'un pool concret dérivant du générique

Le code du pool concret change très peu : il suffit de définir le type réel du Pool générique dont on dérive :

```
public class ResourcePool extends Pool<Resource>
{
	public ResourcePool(int initialResources, int maxResources)
	{
		super(initialResources, maxResources);
	}

	protected Resource createResource()
	{
		return new Resource();
	}
}
```

Le code « extends Pool<Resource> » permet de fixer le type réel de « E » dans la classe Pool. Grâce à cette déclaration, les méthodes héritées de Pool (getResource() et returnResource()) sont considérées comme manipulant des objets de type Resource : en conséquence, nous pouvons nous débarrasser de toutes les méthodes effectuant des « casts ». Notre code est donc plus clair et plus robuste.

Le code de l'environnement de test ne nécessite, quant à lui, aucun changement.

Le code source est disponible dans l'archive Pool4.zip.

VII Conclusion

Tout au long de cet article, nous avons suivi pas à pas le développement d'un pool d'objets correctement synchronisé.

Tout d'abord, nous avons développé un pool à liste simple. Cette approche se révélant insuffisante, nous avons mis en place une liste double pour gérer les objets poolés.

Nous avons ensuite abordé les problématiques liées aux environnements multi-threadés : une synchronisation stricte des threads est indispensable pour garantir la cohérence des données. Java propose des outils puissants pour résoudre ce problème, et nous avons vu comment les mettre en oeuvre.

Pour finir, nous avons fait évoluer ce pool simple de manière à le rendre plus flexible, notamment en permettant de définir une durée maximale d'attente lorsqu'un consommateur demande une ressource au pool.

Notre pool simple fonctionnant bien, nous avons cherché à le rendre générique, de façon à pouvoir pooler facilement différents types d'objets.

Nous avons vu quels problèmes soulevait la généralisation du code du pool, et l'avons modifié en conséquence pendant sa transformation en classe abstraite.

Puis nous avons développé un pool concret dérivant de ce pool abstrait, et constaté que cette solution imposait de nombreux « casts ».

Pour finir, nous avons tiré parti du mécanisme des « génériques » propre à Java5, constatant qu'ils permettaient d'obtenir du code plus robuste et maintenable.

Je vous remercie d'avoir suivi cet article, j'espère que vous en avez apprécié la lecture.

Le texte et le code source sont disponibles sur http://olivier.croisier.free.fr/articles.php

N'hésitez pas à me faire part de vos remarques et commentaires à : olivier.croisier@free.fr