

Programmation concurrente en Java

Brian Goetz

avec Tim Peierls, Joshua Bloch, Joseph Bowbeer,
David Holmes et Doug Lea

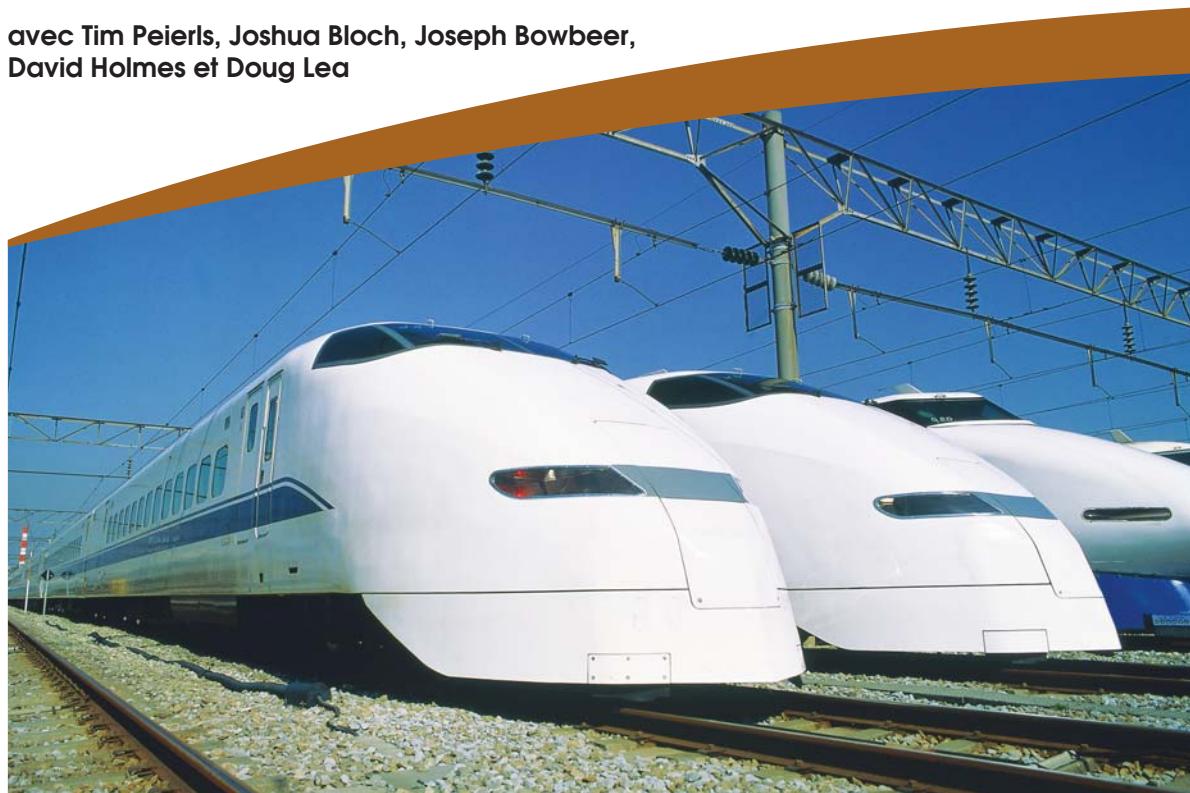
Réseaux
et télécom

Programmation

Génie logiciel

Sécurité

Système
d'exploitation



Programmation concurrente en Java

Brian Goetz

Avec la collaboration de :

Tim Peierls,
Joshua Bloch,
Joseph Bowbeer,
David Holmes
et Doug Lea

Traduction : Éric Jacoboni

Relecture technique : Éric Hébert, Architecte Java JEE
Nicolas de Loof, Architecte Java



Pearson Education France a apporté le plus grand soin à la réalisation de ce livre afin de vous fournir une information complète et fiable. Cependant, Pearson Education France n'assume de responsabilités, ni pour son utilisation, ni pour les contrefaçons de brevets ou atteintes aux droits de tierces personnes qui pourraient résulter de cette utilisation.

Les exemples ou les programmes présents dans cet ouvrage sont fournis pour illustrer les descriptions théoriques. Ils ne sont en aucun cas destinés à une utilisation commerciale ou professionnelle.

Pearson Education France ne pourra en aucun cas être tenu pour responsable des préjudices ou dommages de quelque nature que ce soit pouvant résulter de l'utilisation de ces exemples ou programmes.

Tous les noms de produits ou marques cités dans ce livre sont des marques déposées par leurs propriétaires respectifs.

Publié par Pearson Education France
47 bis, rue des Vinaigriers
75010 PARIS
Tél. : 01 72 74 90 00
www.pearson.fr

Mise en pages : TyPAO

ISBN : 978-2-7440-4109-9

Copyright © 2009 Pearson Education France
Tous droits réservés

Titre original :
Java Concurrency in Practice

Traduit de l'américain par Éric Jacoboni

ISBN original : 978-0-321-34960-6
Copyright © 2006 by Pearson Education, Inc
All rights reserved

Édition originale publiée par
Addison-Wesley Professional,
800 East 96th Street, Indianapolis,
Indiana 46240 USA

Aucune représentation ou reproduction, même partielle, autre que celles prévues à l'article L. 122-5 2° et 3° a) du code de la propriété intellectuelle ne peut être faite sans l'autorisation expresse de Pearson Education France ou, le cas échéant, sans le respect des modalités prévues à l'article L. 122-10 dudit code.

No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

Table des matières

Table des listings	XI
Préface	XIX
Préface à l'édition française	XXI
Présentation de l'ouvrage	XXIII
Structure de l'ouvrage	XXIII
Exemples de code	XXV
Remerciements	XXVI
1 Introduction	1
1.1 Bref historique de la programmation concurrente	1
1.2 Avantages des threads	3
1.2.1 Exploitation de plusieurs processeurs	3
1.2.2 Simplicité de la modélisation	4
1.2.3 Gestion simplifiée des événements asynchrones	5
1.2.4 Interfaces utilisateur plus réactives	5
1.3 Risques des threads	6
1.3.1 Risques concernant la "thread safety"	6
1.3.2 Risques sur la vivacité	9
1.3.3 Risques sur les performances	9
1.4 Les threads sont partout	10

Partie I

Les bases

2 Thread safety	15
2.1 Qu'est-ce que la thread safety ?	17
2.1.1 Exemple : une servlet sans état	19
2.2 Atomicité	19
2.2.1 Situations de compétition	21

2.2.2	Exemple : situations de compétition dans une initialisation paresseuse	22
2.2.3	Actions composées	23
2.3	Verrous	24
2.3.1	Verrous internes	26
2.3.2	Réentrance	27
2.4	Protection de l'état avec les verrous	28
2.5	Vivacité et performances	31
3	Partage des objets	35
3.1	Visibilité	35
3.1.1	Données obsolètes	37
3.1.2	Opérations 64 bits non atomiques	38
3.1.3	Verrous et visibilité	39
3.1.4	Variables volatiles	40
3.2	Publication et fuite	42
3.2.1	Pratiques de construction sûres	44
3.3	Confinement des objets	45
3.3.1	Confinement <i>ad hoc</i>	46
3.3.2	Confinement dans la pile	46
3.3.3	<i>ThreadLocal</i>	47
3.4	Objets non modifiables	49
3.4.1	Champs <i>final</i>	51
3.4.2	Exemple : utilisation de <i>volatile</i> pour publier des objets non modifiables	51
3.5	Publication sûre	53
3.5.1	Publication incorrecte : quand les bons objets deviennent mauvais ..	53
3.5.2	Objets non modifiables et initialisation sûre	54
3.5.3	Idiomes de publication correcte	55
3.5.4	Objets non modifiables dans les faits	56
3.5.5	Objets modifiables	57
3.5.6	Partage d'objets de façon sûre	57
4	Composition d'objets	59
4.1	Conception d'une classe thread-safe	59
4.1.1	Exigences de synchronisation	60
4.1.2	Opérations dépendantes de l'état	61
4.1.3	Appartenance de l'état	62
4.2	Confinement des instances	63
4.2.1	Le patron moniteur de Java	65
4.2.2	Exemple : gestion d'une flotte de véhicules	66

4.3	Délégation de la thread safety	67
4.3.1	Exemple : gestionnaire de véhicules utilisant la délégation	69
4.3.2	Variables d'état indépendantes	70
4.3.3	Échecs de la délégation	71
4.3.4	Publication des variables d'état sous-jacentes	73
4.3.5	Exemple : gestionnaire de véhicules publiant son état	73
4.4	Ajout de fonctionnalités à des classes thread-safe existantes	75
4.4.1	Verrouillage côté client	76
4.4.2	Composition	78
4.5	Documentation des politiques de synchronisation	78
4.5.1	Interprétation des documentations vagues	80
5	Briques de base	83
5.1	Collections synchronisées	83
5.1.1	Problèmes avec les collections synchronisées	83
5.1.2	Itérateurs et <code>ConcurrentModificationException</code>	86
5.1.3	Itérateurs cachés	87
5.2	Collections concurrentes	88
5.2.1	<code>ConcurrentHashMap</code>	89
5.2.2	Opérations atomiques supplémentaires sur les Map	91
5.2.3	<code>CopyOnWriteArrayList</code>	91
5.3	Files bloquantes et patron producteur-consommateur	92
5.3.1	Exemple : indexation des disques	94
5.3.2	Confinement en série	96
5.3.3	Classe <code>Deque</code> et vol de tâches	97
5.4	Méthodes bloquantes et interruptions	97
5.5	Synchroniseurs	99
5.5.1	Loquets	99
5.5.2	<code>FutureTask</code>	101
5.5.3	Sémaphores	103
5.5.4	Barrières	105
5.6	Construction d'un cache efficace et adaptable	107
Résumé de la première partie	113	

Partie II

Structuration des applications concurrentes

6	Exécution des tâches	117
6.1	Exécution des tâches dans les threads	117
6.1.1	Exécution séquentielle des tâches	118

6.1.2	Création explicite de threads pour les tâches	119
6.1.3	Inconvénients d'une création illimitée de threads	120
6.2	Le framework Executor	121
6.2.1	Exemple : serveur web utilisant Executor	122
6.2.2	Politiques d'exécution	123
6.2.3	Pools de threads	124
6.2.4	Cycle de vie d'un Executor	125
6.2.5	Tâches différées et périodiques	127
6.3	Trouver un parallélisme exploitable	128
6.3.1	Exemple : rendu séquentiel d'une page	129
6.3.2	Tâches partielles : Callable et Future	129
6.3.3	Exemple : affichage d'une page avec Future	131
6.3.4	Limitations du parallélisme de tâches hétérogènes	132
6.3.5	CompletionService : quand Executor rencontre BlockingQueue	133
6.3.6	Exemple : affichage d'une page avec CompletionService	134
6.3.7	Imposer des délais aux tâches	135
6.3.8	Exemple : portail de réservations	136
	Résumé	137
7	Annulation et arrêt	139
7.1	Annulation des tâches	140
7.1.1	Interruption	142
7.1.2	Politiques d'interruption	145
7.1.3	Répondre aux interruptions	146
7.1.4	Exemple : exécution avec délai	148
7.1.5	Annulation avec Future	150
7.1.6	Méthodes bloquantes non interruptibles	151
7.1.7	Encapsulation d'une annulation non standard avec newTaskFor()	153
7.2	Arrêt d'un service reposant sur des threads	154
7.2.1	Exemple : service de journalisation	155
7.2.2	Méthodes d'arrêt de ExecutorService	158
7.2.3	Pilules empoisonnées	159
7.2.4	Exemple : un service d'exécution éphémère	160
7.2.5	Limitations de shutdownNow()	161
7.3	Gestion de la fin anormale d'un thread	163
7.3.1	Gestionnaires d'exceptions non capturées	165
7.4	Arrêt de la JVM	166
7.4.1	Méthodes d'interception d'un ordre d'arrêt	166
7.4.2	Threads démons	168
7.4.3	Finaliseurs	168
	Résumé	169

8 Pools de threads	171
8.1 Couplage implicite entre les tâches et les politiques d'exécution	171
8.1.1 Interblocage par famine de thread	173
8.1.2 Tâches longues	174
8.2 Taille des pools de threads	174
8.3 Configuration de ThreadPoolExecutor	176
8.3.1 Création et suppression de threads	176
8.3.2 Gestion des tâches en attente	177
8.3.3 Politiques de saturation	179
8.3.4 Fabriques de threads	181
8.3.5 Personnalisation de ThreadPoolExecutor après sa construction	183
8.4 Extension de ThreadPoolExecutor	183
8.4.1 Exemple : ajout de statistiques à un pool de threads	184
8.5 Parallélisation des algorithmes récursifs	185
8.5.1 Exemple : un framework de jeu de réflexion	187
Résumé	192
9 Applications graphiques	193
9.1 Pourquoi les interfaces graphiques sont-elles monothreads ?	193
9.1.1 Traitement séquentiel des événements	195
9.1.2 Confinement aux threads avec Swing	196
9.2 Tâches courtes de l'interface graphique	198
9.3 Tâches longues de l'interface graphique	199
9.3.1 Annulation	201
9.3.2 Indication de progression et de terminaison	202
9.3.3 SwingWorker	204
9.4 Modèles de données partagées	204
9.4.1 Modèles de données thread-safe	205
9.4.2 Modèles de données séparés	205
9.5 Autres formes de sous-systèmes monothreads	206
Résumé	206

Partie III

Vivacité, performances et tests

10 Éviter les problèmes de vivacité	209
10.1 Interblocages (deadlock)	209
10.1.1 Interblocages liés à l'ordre du verrouillage	210
10.1.2 Interblocages dynamiques liés à l'ordre du verrouillage	212
10.1.3 Interblocages entre objets coopératifs	215

10.1.4	Appels ouverts	216
10.1.5	Interblocages liés aux ressources	218
10.2	Éviter et diagnostiquer les interblocages	219
10.2.1	Tentatives de verrouillage avec expiration	219
10.2.2	Analyse des interblocages avec les traces des threads	220
10.3	Autres problèmes de vivacité	221
10.3.1	Famine	222
10.3.2	Faible réactivité	223
10.3.3	Livelock	223
	Résumé	224
11	Performances et adaptabilité.....	225
11.1	Penser aux performances	225
11.1.1	Performances et adaptabilité	226
11.1.2	Compromis sur l'évaluation des performances	228
11.2	La loi d'Amdahl	230
11.2.1	Exemple : sérialisation cachée dans les frameworks	232
11.2.2	Application qualitative de la loi d'Amdahl	233
11.3	Coûts liés aux threads	234
11.3.1	Changements de contexte	234
11.3.2	Synchronisation de la mémoire	235
11.3.3	Blocages	237
11.4	Réduction de la compétition pour les verrous	237
11.4.1	Réduction de la portée des verrous ("entrer, sortir")	238
11.4.2	Réduire la granularité du verrouillage	240
11.4.3	Découpage du verrouillage	242
11.4.4	Éviter les points chauds	244
11.4.5	Alternatives aux verrous exclusifs	245
11.4.6	Surveillance de l'utilisation du processeur	245
11.4.7	Dire non aux pools d'objets	246
11.5	Exemple : comparaison des performances des Map	247
11.6	Réduction du surcoût des changements de contexte	249
	Résumé	251
12	Tests des programmes concurrents.....	253
12.1	Tests de la justesse	254
12.1.1	Tests unitaires de base	256
12.1.2	Tests des opérations bloquantes	256
12.1.3	Test de la sécurité vis-à-vis des threads	258
12.1.4	Test de la gestion des ressources	263
12.1.5	Utilisation des fonctions de rappel	264
12.1.6	Production d'entrelacements supplémentaires	265

12.2	Tests de performances	266
12.2.1	Extension de <code>PutTakeTest</code> pour ajouter un timing	266
12.2.2	Comparaison de plusieurs algorithmes	269
12.2.3	Mesure de la réactivité	270
12.3	Pièges des tests de performance	272
12.3.1	Ramasse-miettes	272
12.3.2	Compilation dynamique	272
12.3.3	Échantillonnage irréaliste de portions de code	274
12.3.4	Degrés de compétition irréalistes	274
12.3.5	Élimination du code mort	275
12.4	Approches de tests complémentaires	277
12.4.1	Relecture du code	277
12.4.2	Outils d'analyse statiques	278
12.4.3	Techniques de tests orientées aspects	279
12.4.4	Profileurs et outils de surveillance	280
	Résumé	280

Partie IV

Sujets avancés

13	Verrous explicites	283
13.1	<code>Lock</code> et <code>ReentrantLock</code>	283
13.1.1	Prise de verrou scrutable et avec délai	285
13.1.2	Prise de verrou interruptible	287
13.1.3	Verrouillage non structuré en bloc	287
13.2	Remarques sur les performances	288
13.3	Équité	289
13.4	<code>synchronized</code> vs. <code>ReentrantLock</code>	291
13.5	Verrous en lecture-écriture	292
	Résumé	296
14	Construction de synchroniseurs personnalisés	297
14.1	Gestion de la dépendance par rapport à l'état	297
14.1.1	Exemple : propagation de l'échec de la précondition aux appelants	299
14.1.2	Exemple : blocage brutal par essai et mise en sommeil	301
14.1.3	Les files d'attente de condition	303
14.2	Utilisation des files d'attente de condition	304
14.2.1	Le prédictat de condition	304
14.2.2	Réveil trop précoce	306

14.2.3	Signaux manqués	307
14.2.4	Notification	308
14.2.5	Exemple : une classe "porte d'entrée"	310
14.2.6	Problèmes de safety des sous-classes	311
14.2.7	Encapsulation des files d'attente de condition	312
14.2.8	Protocoles d'entrée et de sortie	312
14.3	Objets conditions explicites	313
14.4	Anatomie d'un synchronisateur	315
14.5	<code>AbstractQueuedSynchronizer</code>	317
14.5.1	Un loquet simple	319
14.6	AQS dans les classes de <code>java.util.concurrent</code>	320
14.6.1	<code>ReentrantLock</code>	320
14.6.2	<code>Semaphore</code> et <code>CountDownLatch</code>	321
14.6.3	<code>FutureTask</code>	322
14.6.4	<code>ReentrantReadWriteLock</code>	322
	Résumé	323
15	Variables atomiques et synchronisation non bloquante	325
15.1	Inconvénients du verrouillage	326
15.2	Support matériel de la concurrence	327
15.2.1	L'instruction <i>Compare-and-swap</i>	328
15.2.2	Compteur non bloquant	329
15.2.3	Support de CAS dans la JVM	331
15.3	Classes de variables atomiques	331
15.3.1	Variables atomiques comme "volatiles améliorées"	332
15.3.2	Comparaison des performances des verrous et des variables atomiques	333
15.4	Algorithmes non bloquants	336
15.4.1	Pile non bloquante	337
15.4.2	File chaînée non bloquante	338
15.4.3	Modificateurs atomiques de champs	342
15.4.4	Le problème ABA	342
	Résumé	343
16	Le modèle mémoire de Java	345
16.1	Qu'est-ce qu'un modèle mémoire et pourquoi en a-t-on besoin ?	345
16.1.1	Modèles mémoire des plates-formes	346
16.1.2	Réorganisation	347
16.1.3	Le modèle mémoire de Java en moins de cinq cents mots	349
16.1.4	Tirer parti de la synchronisation	351

16.2	Publication	353
16.2.1	Publication incorrecte	353
16.2.2	Publication correcte	354
16.2.3	Idiomes de publication correcte	355
16.2.4	Verrouillage contrôlé deux fois	356
16.3	Initialisation sûre	358
	Résumé	359
Annexe	Annotations pour la concurrence	361
A.1	Annotations de classes	361
A.2	Annotations de champs et de méthodes	361
Bibliographie	363	
Index	365	

Table des listings

Listing 1 : Mauvaise façon de trier une liste. <i>Ne le faites pas.</i>	XX
Listing 2 : Méthode peu optimale de trier une liste.	XX
Listing 1.1 : Générateur de séquence non thread-safe.	7
Listing 1.2 : Générateur de séquence thread-safe.	8
Listing 2.1 : Une servlet sans état.	19
Listing 2.2 : Servlet comptant le nombre de requêtes sans la synchronisation nécessaire. <i>Ne le faites pas.</i>	20
Listing 2.3 : Situation de compétition dans une initialisation paresseuse. <i>Ne le faites pas.</i>	22
Listing 2.4 : Servlet comptant les requêtes avec <code>AtomicLong</code>	24
Listing 2.5 : Servlet tentant de mettre en cache son dernier résultat sans l'atomicité adéquate. <i>Ne le faites pas.</i>	25
Listing 2.6 : Servlet mettant en cache le dernier résultat, mais avec une très mauvaise concurrence. <i>Ne le faites pas.</i>	27
Listing 2.7 : Ce code se bloquerait si les verrous internes n'étaient pas réentrant.	28
Listing 2.8 : Servlet mettant en cache la dernière requête et son résultat.	32
Listing 3.1 : Partage de données sans synchronisation. <i>Ne le faites pas.</i>	36
Listing 3.2 : Conteneur non thread-safe pour un entier modifiable.	38
Listing 3.3 : Conteneur thread-safe pour un entier modifiable.	38
Listing 3.4 : Compter les moutons.	41
Listing 3.5 : Publication d'un objet.	42
Listing 3.6 : L'état modifiable interne à la classe peut s'échapper. <i>Ne le faites pas.</i>	42
Listing 3.7 : Permet implicitement à la référence <code>this</code> de s'échapper. <i>Ne le faites pas.</i>	43
Listing 3.8 : Utilisation d'une méthode fabrique pour empêcher la référence <code>this</code> de s'échapper au cours de la construction de l'objet.	44
Listing 3.9 : Confinement des variables locales, de types primitifs ou de types références.	47
Listing 3.10 : Utilisation de <code>ThreadLocal</code> pour garantir le confinement au thread.	48
Listing 3.11 : Classe non modifiable construite à partir d'objets modifiables sous-jacents.	50
Listing 3.12 : Conteneur non modifiable pour mettre en cache un nombre et ses facteurs.	52
Listing 3.13 : Mise en cache du dernier résultat à l'aide d'une référence volatile vers un objet conteneur non modifiable.	52
Listing 3.14 : Publication d'un objet sans synchronisation appropriée. <i>Ne le faites pas.</i>	53
Listing 3.15 : Classe risquant un problème si elle n'est pas correctement publiée.	54

Listing 4.1 : Compteur mono-thread utilisant le patron moniteur de Java.	60
Listing 4.2 : Utilisation du confinement pour assurer la thread safety.	64
Listing 4.3 : Protection de l'état à l'aide d'un verrou privé.	66
Listing 4.4 : Implémentation du gestionnaire de véhicule reposant sur un moniteur.	68
Listing 4.5 : Classe Point modifiable ressemblant à java.awt.Point.	69
Listing 4.6 : Classe Point non modifiable utilisée par DelegatingVehicleTracker.	69
Listing 4.7 : Délégation de la thread safety à un objet ConcurrentHashMap.	69
Listing 4.8 : Renvoi d'une copie statique de l'ensemble des emplacements au lieu d'une copie "vivante".	70
Listing 4.9 : Délégation de la thread à plusieurs variables d'état sous-jacentes.	71
Listing 4.10 : Classe pour des intervalles numériques, qui ne protège pas suffisamment ses invariants. <i>Ne le faites pas.</i>	71
Listing 4.11 : Classe point modifiable et thread-safe.	73
Listing 4.12 : Gestionnaire de véhicule qui publie en toute sécurité son état interne.	74
Listing 4.13 : Extension de Vector pour disposer d'une méthode <i>ajouter-si-absent</i>	76
Listing 4.14 : Tentative non thread-safe d'implémenter <i>ajouter-si-absent</i> . <i>Ne le faites pas.</i>	76
Listing 4.15 : Implémentation d' <i>ajouter-si-absent</i> avec un verrouillage côté client.	77
Listing 4.16 : Implémentation d' <i>ajouter-si-absent</i> en utilisant la composition.	78
Listing 5.1 : Actions composées sur un Vector pouvant produire des résultats inattendus.	84
Listing 5.2 : Actions composées sur Vector utilisant un verrouillage côté client.	85
Listing 5.3 : Itération pouvant déclencher ArrayIndexOutOfBoundsException.	85
Listing 5.4 : Itération avec un verrouillage côté client.	86
Listing 5.5 : Parcours d'un objet List avec un Iterator.	86
Listing 5.6 : Itération cachée dans la concaténation des chaînes. <i>Ne le faites pas.</i>	88
Listing 5.7 : Interface ConcurrentMap.	91
Listing 5.8 : Tâches producteur et consommateur dans une application d'indexation des fichiers.	95
Listing 5.9 : Lancement de l'indexation.	96
Listing 5.10 : Restauration de l'état d'interruption afin de ne pas absorber l'interruption.	99
Listing 5.11 : Utilisation de la classe CountDownLatch pour lancer et stopper des threads et mesurer le temps d'exécution.	101
Listing 5.12 : Utilisation de FutureTask pour précharger des données dont on aura besoin plus tard.	102
Listing 5.13 : Coercition d'un objet Throwable non contrôlé en RuntimeException.	103
Listing 5.14 : Utilisation d'un Semaphore pour borner une collection.	104
Listing 5.15 : Coordination des calculs avec CyclicBarrier pour une simulation de cellules.	106
Listing 5.16 : Première tentative de cache, utilisant HashMap et la synchronisation.	107
Listing 5.17 : Remplacement de HashMap par ConcurrentHashMap.	109
Listing 5.18 : Enveloppe de mémoïsation utilisant FutureTask.	110

Listing 5.19 : Implémentation finale de <code>Memoizer</code>	111
Listing 5.20 : Servlet de factorisation mettant en cache ses résultats avec <code>Memoizer</code>	112
Listing 6.1 : Serveur web séquentiel.	118
Listing 6.2 : Serveur web lançant un thread par requête.	119
Listing 6.3 : Interface <code>Executor</code>	121
Listing 6.4 : Serveur web utilisant un pool de threads.	122
Listing 6.5 : <code>Executor</code> lançant un nouveau thread pour chaque tâche.	123
Listing 6.6 : <code>Executor</code> exécutant les tâches de façon synchrone dans le thread appelant.	123
Listing 6.7 : Méthodes de <code>ExecutorService</code> pour le cycle de vie.	126
Listing 6.8 : Serveur web avec cycle de vie.	126
Listing 6.9 : Classe illustrant le comportement confus de <code>Timer</code>	128
Listing 6.10 : Affichage séquentiel des éléments d'une page.	129
Listing 6.11 : Interfaces <code>Callable</code> et <code>Future</code>	130
Listing 6.12 : Implémentation par défaut de <code>newTaskFor()</code> dans <code>ThreadPoolExecutor</code>	131
Listing 6.13 : Attente du téléchargement d'image avec <code>Future</code>	131
Listing 6.14 : La classe <code>QueueingFuture</code> utilisée par <code>ExecutorCompletionService</code>	134
Listing 6.15 : Utilisation de <code>CompletionService</code> pour afficher les éléments de la page dès qu'ils sont disponibles.	134
Listing 6.16 : Récupération d'une publicité dans un délai imparti.	135
Listing 6.17 : Obtention de tarifs dans un délai imparti.	137
Listing 7.1 : Utilisation d'un champ <code>volatile</code> pour stocker l'état d'annulation.	140
Listing 7.2 : Génération de nombres premiers pendant une seconde.	141
Listing 7.3 : Annulation non fiable pouvant bloquer les producteurs. <i>Ne le faites pas.</i>	142
Listing 7.4 : Méthodes d'interruption de <code>Thread</code>	143
Listing 7.5 : Utilisation d'une interruption pour l'annulation.	144
Listing 7.6 : Propagation de <code>InterruptedException</code> aux appelants.	147
Listing 7.7 : Tâche non annulable qui restaure l'interruption avant de se terminer.	147
Listing 7.8 : Planification d'une interruption sur un thread emprunté. <i>Ne le faites pas.</i>	149
Listing 7.9 : Interruption d'une tâche dans un thread dédié.	149
Listing 7.10 : Annulation d'une tâche avec <code>Future</code>	151
Listing 7.11 : Encapsulation des annulations non standard dans un thread par redéfinition de <code>interrupt()</code>	152
Listing 7.12 : Encapsulation des annulations non standard avec <code>newTaskFor()</code>	154
Listing 7.13 : Service de journalisation producteur-consommateur sans support de l'arrêt.	156
Listing 7.14 : Moyen non fiable d'ajouter l'arrêt au service de journalisation.	157
Listing 7.15 : Ajout d'une annulation fiable à <code>LogWriter</code>	157
Listing 7.16 : Service de journalisation utilisant un <code>ExecutorService</code>	158
Listing 7.17 : Arrêt d'un service avec une pilule empoisonnée.	159

Listing 7.18 : Thread producteur pour <code>IndexingService</code> .	159
Listing 7.19 : Thread consommateur pour <code>IndexingService</code> .	160
Listing 7.20 : Utilisation d'un <code>Executor</code> privé dont la durée de vie est limitée à un appel de méthode.	160
Listing 7.21 : <code>ExecutorService</code> mémorisant les tâches annulées après l'arrêt.	161
Listing 7.22 : Utilisation de <code>TrackingExecutorService</code> pour mémoriser les tâches non terminées afin de les relancer plus tard.	162
Listing 7.23 : Structure typique d'un thread d'un pool de threads.	164
Listing 7.24 : Interface <code>UncaughtExceptionHandler</code> .	165
Listing 7.25 : <code>UncaughtExceptionHandler</code> , qui inscrit l'exception dans le journal.	165
Listing 7.26 : Enregistrement d'un hook d'arrêt pour arrêter le service de journalisation.	167
Listing 8.1 : Interblocage de tâches dans un <code>Executor</code> monothread. Ne le faites pas.	173
Listing 8.2 : Constructeur général de <code>ThreadPoolExecutor</code> .	176
Listing 8.3 : Création d'un pool de threads de taille fixe avec une file bornée et la politique de saturation <code>caller-runs</code> .	180
Listing 8.4 : Utilisation d'un <code>Semaphore</code> pour ralentir la soumission des tâches.	180
Listing 8.5 : Interface <code>ThreadFactory</code> .	181
Listing 8.6 : Fabrique de threads personnalisés.	182
Listing 8.7 : Classe de base pour les threads personnalisés.	182
Listing 8.8 : Modification d'un <code>Executor</code> créé avec les méthodes fabriques standard.	183
Listing 8.9 : Pool de threads étendu par une journalisation et une mesure du temps.	184
Listing 8.10 : Transformation d'une exécution séquentielle en exécution parallèle.	185
Listing 8.11 : Transformation d'une récursion terminale séquentielle en récursion parallèle.	186
Listing 8.12 : Attente des résultats calculés en parallèle.	187
Listing 8.13 : Abstraction pour les jeux de type "taquin".	187
Listing 8.14 : Nœud pour le framework de résolution des jeux de réflexion.	187
Listing 8.15 : Résolveur séquentiel d'un puzzle.	188
Listing 8.16 : Version parallèle du résolveur de puzzle.	189
Listing 8.17 : Loquet de résultat partiel utilisé par <code>ConcurrentPuzzleSolver</code> .	190
Listing 8.18 : Résolveur reconnaissant qu'il n'y a pas de solution.	191
Listing 9.1 : Implémentation de <code> SwingUtilities</code> à l'aide d'un <code>Executor</code> .	197
Listing 9.2 : Executor construit au-dessus de <code> SwingUtilities</code> .	197
Listing 9.3 : Écouteur d'événement simple.	198
Listing 9.4 : Liaison d'une tâche longue à un composant visuel.	200
Listing 9.5 : Tâche longue avec effet visuel.	200
Listing 9.6 : Annulation d'une tâche longue.	201
Listing 9.7 : Classe de tâche en arrière-plan supportant l'annulation, ainsi que la notification de terminaison et de progression.	202

Listing 9.8 : Utilisation de <code>BackgroundTask</code> pour lancer une tâche longue et annulable.	203
Listing 10.1 : Interblocage simple lié à l'ordre du verrouillage. <i>Ne le faites pas.</i>	211
Listing 10.2 : Interblocage dynamique lié à l'ordre du verrouillage. <i>Ne le faites pas.</i>	212
Listing 10.3 : Induire un ordre de verrouillage pour éviter les interblocages.	213
Listing 10.4 : Boucle provoquant un interblocage dans une situation normale.	214
Listing 10.5 : Interblocage lié à l'ordre du verrouillage entre des objets coopératifs. <i>Ne le faites pas.</i>	215
Listing 10.6 : Utilisation d'appels ouverts pour éviter l'interblocage entre des objets coopératifs.	217
Listing 10.7 : Portion d'une trace de thread après un interblocage.	220
Listing 11.1 : Accès séquentiel à une file d'attente.	231
Listing 11.2 : Synchronisation inutile. <i>Ne le faites pas.</i>	236
Listing 11.3 : Candidat à l'élosion de verrou.	236
Listing 11.4 : Détenzione d'un verrou plus longtemps que nécessaire.	239
Listing 11.5 : Réduction de la durée du verrouillage.	239
Listing 11.6 : Candidat au découpage du verrou.	241
Listing 11.7 : Modification de <code>ServerStatus</code> pour utiliser des verrous divisés.	242
Listing 11.8 : Hachage utilisant le découpage du verrouillage.	243
Listing 12.1 : Tampon borné utilisant la classe <code>Semaphore</code>	255
Listing 12.2 : Tests unitaires de base pour <code>BoundedBuffer</code>	256
Listing 12.3 : Test du blocage et de la réponse à une interruption.	257
Listing 12.4 : Générateur de nombre aléatoire de qualité moyenne mais suffisante pour les tests.	260
Listing 12.5 : Programme de test producteur-consommateur pour <code>BoundedBuffer</code>	260
Listing 12.6 : Classes producteur et consommateur utilisées dans <code>PutTakeTest</code>	261
Listing 12.7 : Test des fuites de ressources.	263
Listing 12.8 : Fabrique de threads pour tester <code>ThreadPoolExecutor</code>	264
Listing 12.9 : Méthode de test pour vérifier l'expansion du pool de threads.	265
Listing 12.10 : Utilisation de <code>Thread.yield()</code> pour produire plus d'entrelacements	266
Listing 12.11 : Mesure du temps à l'aide d'une barrière.	267
Listing 12.12 : Test avec mesure du temps à l'aide d'une barrière.	267
Listing 12.13 : Programme pilote pour <code>TimedPutTakeTest</code>	268
Listing 13.1 : Interface <code>Lock</code>	283
Listing 13.2 : Protection de l'état d'un objet avec <code>ReentrantLock</code>	284
Listing 13.3 : Utilisation de <code>tryLock()</code> pour éviter les interblocages dus à l'ordre des verrouillages.	285
Listing 13.4 : Verrouillage avec temps imparti.	286
Listing 13.5 : Prise de verrou interruptible.	287

Listing 13.6 : Interface <code>ReadWriteLock</code>	293
Listing 13.7 : Enveloppe d'un <code>Map</code> avec un verrou de lecture-écriture.	295
Listing 14.1 : Structure des actions bloquantes en fonction de l'état.	298
Listing 14.2 : Classe de base pour les implémentations de tampons bornés.	299
Listing 14.3 : Tampon borné qui se dérobe lorsque les préconditions ne sont pas vérifiées.	299
Listing 14.4 : Code client pour l'appel de <code>GrumpyBoundedBuffer</code>	300
Listing 14.5 : Tampon borné avec blocage brutal.	301
Listing 14.6 : Tampon borné utilisant des files d'attente de condition.	304
Listing 14.7 : Forme canonique des méthodes dépendantes de l'état.	307
Listing 14.8 : Utilisation d'une notification conditionnelle dans <code>BoundedBuffer.put()</code> .	310
Listing 14.9 : Porte refermable à l'aide de <code>wait()</code> et <code>notifyAll()</code>	310
Listing 14.10 : Interface <code>Condition</code>	313
Listing 14.11 : Tampon borné utilisant des variables conditions explicites.	314
Listing 14.12 : <code>Semaphore</code> implémenté à partir de <code>Lock</code>	315
Listing 14.13 : Formes canoniques de l'acquisition et de la libération avec <code>AQS</code>	318
Listing 14.14 : Loquet binaire utilisant <code>AbstractQueuedSynchronizer</code>	319
Listing 14.15 : Implémentation de <code>tryAcquire()</code> pour un <code>ReentrantLock</code> non équitable.	321
Listing 14.16 : Les méthodes <code>tryAcquireShared()</code> et <code>tryReleaseShared()</code> de <code>Semaphore</code>	322
Listing 15.1 : Simulation de l'opération CAS.	328
Listing 15.2 : Compteur non bloquant utilisant l'instruction CAS.	329
Listing 15.3 : Préservation des invariants multivariables avec CAS.	333
Listing 15.4 : Générateur de nombres pseudo-aléatoires avec <code>ReentrantLock</code>	334
Listing 15.5 : Générateur de nombres pseudo-aléatoires avec <code>AtomicInteger</code>	334
Listing 15.6 : Pile non bloquante utilisant l'algorithme de Treiber (Treiber, 1986).	337
Listing 15.7 : Insertion dans l'algorithme non bloquant de Michael-Scott (Michael et Scott, 1996).	340
Listing 15.8 : Utilisation de modificateurs atomiques de champs dans <code>ConcurrentLinkedQueue</code>	342
Listing 16.1 : Programme mal synchronisé pouvant produire des résultats surprenants. <i>Ne le faites pas.</i>	348
Listing 16.2 : Classe interne de <code>FutureTask</code> illustrant une mise à profit de la synchronisation.	351
Listing 16.3 : Initialisation paresseuse incorrecte. <i>Ne le faites pas.</i>	353
Listing 16.4 : Initialisation paresseuse thread-safe.	355
Listing 16.5 : Initialisation impatiente.	356
Listing 16.6 : Idiome de la classe conteneur de l'initialisation paresseuse.	356
Listing 16.7 : Antipatron du verrouillage vérifié deux fois. <i>Ne le faites pas.</i>	357
Listing 16.8 : Initialisation sûre pour les objets non modifiables.	358

Préface

À l'heure où ce livre est écrit, les machines de gamme moyenne utilisent désormais des processeurs multicœurs. En même temps, et ce n'est pas une coïncidence, les rapports de bogues signalent de plus en plus de problèmes liés aux threads. Dans un article récent posté sur le site des développeurs de NetBeans, l'un des développeurs principaux indique qu'une même classe a été corrigée plus de quatorze fois pour remédier à ce genre de problème. Dion Almaer, ancien éditeur de *TheServerSide*, a récemment écrit dans son blog (après une session de débogage harassante qui a fini par révéler un bogue lié aux threads) que ce type de bogue est si courant dans les programmes Java que ceux-ci ne fonctionnent souvent que "par accident".

Le développement, le test et le débogage des programmes multithreads peut se révéler très difficile car, évidemment, les problèmes de concurrence se manifestent de façon imprévisible. Ils apparaissent généralement au pire moment – lorsque le programme est en production et doit gérer une lourde charge de travail.

L'une des difficultés de la programmation concurrente en Java consiste à distinguer la concurrence offerte par la plate-forme et la façon dont les développeurs doivent appréhender cette concurrence dans leurs programmes. Le langage fournit des *mécanismes* de bas niveau, comme la synchronisation et l'attente de conditions, qui doivent être utilisés correctement pour implémenter des protocoles ou des *politiques* au niveau des applications. Sans ces politiques, il est vraiment très facile de créer des programmes qui se compileront et sembleront fonctionner alors qu'ils sont bogués. De nombreux ouvrages excellents consacrés à la programmation concurrente manquent ainsi leur but en se consacrant presque exclusivement aux mécanismes de bas niveau et aux API au lieu de s'intéresser aux politiques et aux patrons de conception.

Java 5.0 constitue une étape majeure vers le développement d'applications concurrentes en Java car il fournit à la fois des composants de haut niveau et des mécanismes de bas niveau supplémentaires facilitant la construction des applications concurrentes à la fois pour les débutants et les experts. Les auteurs sont des membres essentiels du *JCP Expert Group*¹, qui a créé ces outils ; outre la description de leur comportement et de

1. N.d.T. : JCP signifie *Java Community Process*. Il s'agit d'un processus de développement et d'amélioration de Java ouvert à toutes les bonnes volontés. Les propositions émises sont appelées JSR (*Java Specification Request*) et leur mise en place est encadrée par un groupe d'experts (*Expert Group*).

leurs fonctionnalités, nous présenterons les cas d'utilisation qui ont motivé leur ajout aux bibliothèques de la plate-forme.

Notre but est de fournir aux lecteurs un ensemble de règles de conception et de modèles mentaux qui facilitent – et rendent plus agréable – le développement de classes et d'applications concurrentes en Java.

Nous espérons que vous apprécierez *Programmation concurrente en Java*.

Brian Goetz
Williston, VT
Mars 2006

Préface à l'édition française

Lors de la première édition de ce livre, nous avions écrit que "les processeurs multi-cœurs commencent à être suffisamment bon marché pour apparaître dans les systèmes de milieu de gamme". Deux ans plus tard, nous pouvons constater que cette tendance s'est poursuivie, voire accélérée.

Même les portables et les machines de bureau d'entrée de gamme disposent maintenant de processeurs multicœurs, tandis que les machines de haut de gamme voient leur nombre de cœurs grandir chaque année et que les fabricants de CPU ont clairement indiqué qu'ils s'attendaient à ce que le nombre de cœurs progresse de façon exponentielle au cours des prochaines années. Du coup, il devient difficile de trouver des systèmes monoprocesseurs.

Cette tendance du matériel pose des problèmes non négligeables aux développeurs logiciels. Il ne suffit plus d'exécuter des programmes existants sur de nouveaux processeurs pour qu'ils aillent plus vite. La loi de Moore continue de développer plus de transistors chaque année, mais elle nous offre désormais plus de cœurs que de cœurs plus rapides. Si nous voulons tirer parti des avantages de la puissance des nouveaux processeurs, nos programmes doivent être écrits pour supporter les environnements concurrents, ce qui représente un défi à la fois en termes d'architecture, de programmation et de tests. Le but de ce livre est de répondre à ces défis en offrant des techniques, des patrons et des outils pour analyser les programmes concurrents et pour encapsuler la complexité des interactions concurrentes.

Comprendre la concurrence est devenu plus que jamais nécessaire pour les développeurs Java.

Brian Goetz
Williston, VT
Janvier 2008

Présentation de l'ouvrage

Structure de l'ouvrage

Pour éviter la confusion entre les mécanismes de bas niveau de Java et les politiques de conception nécessaires, nous présenterons un ensemble de règles *simplifié* permettant d'écrire des programmes concurrents. En lisant ces règles, les experts pourront dire : "Hum, ce n'est pas entièrement vrai : la classe C est thread-safe bien qu'elle viole la règle R !" Écrire des programmes corrects qui ne respectent pas nos règles est bien sûr possible, mais à condition de connaître parfaitement les mécanismes de bas niveau du modèle mémoire de Java, or nous voulons justement que les développeurs puissent écrire des programmes concurrents *sans* avoir besoin de maîtriser tous ces détails. Nos règles simplifiées permettent de produire des programmes concurrents corrects et faciles à maintenir.

Nous supposons que le lecteur connaît déjà un peu les mécanismes de base de la programmation concurrente en Java. *Programmation concurrente en Java* n'est pas une introduction à la programmation concurrente – pour cela, consultez le chapitre consacré à ce sujet dans un ouvrage qui fait autorité, comme *The Java Programming Language* (Arnold et al., 2005). Ce n'est pas non plus un ouvrage de référence sur la concurrence en général – pour cela, lisez *Concurrent Programming in Java* (Lea, 2000). Nous préférons ici offrir des règles de conception pratiques pour aider les développeurs à créer des classes concurrentes, sûres et efficaces. Lorsque cela sera nécessaire, nous ferons référence aux sections appropriées de *The Java Programming Language*, *Concurrent Programming in Java*, *The Java Language Specification* (Gosling et al., 2005) et *Effective Java* (Bloch, 2001) en utilisant les conventions [JPL n.m], [CPJ n.m], [JLS n.m] et [EJ Item n].

Après l'introduction (Chapitre 1), ce livre est découpé en quatre parties.

Les bases. La première partie (Chapitres 2 à 5) s'intéresse aux concepts fondamentaux de la concurrence et des threads, ainsi qu'à la façon de composer des classes "thread-safe"¹ à partir des composants fournis par la bibliothèque de classes. Une "carte de référence" résume les règles les plus importantes présentées dans cette partie.

1. N.d.T : Dans ce livre nous garderons certains termes anglais car ils n'ont pas d'équivalents reconnus en français. C'est le cas de "thread-safe", qui est une propriété indiquant qu'un code a été conçu pour se comporter correctement lorsqu'on y accède par plusieurs threads simultanément.

Les Chapitres 2 (Thread safety) et 3 (Partage des objets) présentent les concepts fondamentaux de cet ouvrage. Quasiment toutes les règles liées aux problèmes de concurrence, à la construction de classes thread-safe et à la vérification du bon fonctionnement de ces classes sont introduites dans ces deux chapitres. Si vous préférez la "pratique" à la "théorie", vous pourriez être tenté de passer directement à la deuxième partie du livre, mais assurez-vous de lire ces chapitres avant d'écrire du code concurrent !

Le Chapitre 4 (Composition d'objets) explique comment composer des classes thread-safe pour former des classes thread-safe plus importantes. Le Chapitre 5 (Briques de base) décrit les briques de base de la programmation concurrente – les collections et les synchronisateurs thread-safe – fournies par les bibliothèques de la plate-forme.

Structuration des applications parallèles. La deuxième partie (Chapitres 6 à 9) explique comment utiliser les threads pour améliorer le rendement ou le temps de réponse des applications concurrentes. Le Chapitre 6 (Exécution des tâches) montre comment identifier les tâches parallélisables et les exécuter. Le Chapitre 7 (Annulation et arrêt) explique comment demander aux tâches et aux threads de se terminer avant leur échéance normale ; la façon dont les programmes gèrent l'annulation et la terminaison est souvent l'un des facteurs permettant de différencier les applications concurrentes vraiment robustes de celles qui se contentent de fonctionner. Le Chapitre 8 (Pools de threads) présente quelques-unes des techniques les plus avancées de l'exécution des tâches. Le Chapitre 9 (Applications graphiques) s'intéresse aux techniques permettant d'améliorer le temps de réponse des sous-systèmes monothreads.

Vivacité, performances et tests. La troisième partie (Chapitres 10 à 12) s'occupe de vérifier que les programmes concurrents font bien ce que l'on veut qu'ils fassent, tout en ayant des performances acceptables. Le Chapitre 10 (Éviter les problèmes de vivacité) explique comment éviter les problèmes de vivacité qui peuvent empêcher les programmes d'avancer. Le Chapitre 11 (Performances et adaptabilité) présente les techniques permettant d'améliorer les performances et l'adaptabilité du code concurrent. Le Chapitre 12 (Tests des programmes concurrents) décrit les techniques de test du code concurrent, qui permettent de vérifier qu'il est à la fois correct et performant.

Sujets avancés. La quatrième et dernière partie (Chapitres 13 à 16) présente des sujets qui n'intéresseront probablement que les développeurs expérimentés : les verrous explicites, les variables atomiques, les algorithmes non bloquants et le développement de synchronisateurs personnalisés.

Exemples de code

Bien que de nombreux concepts généraux exposés dans ce livre s'appliquent aux versions de Java antérieures à Java 5.0, voire aux environnements non Java, la plupart des exemples de code (et tout ce qui concerne le modèle mémoire de Java) supposent que vous utilisez Java 5.0 ou une version plus récente. En outre, certains exemples utilisent des fonctionnalités qui ne sont apparues qu'à partir de Java 6.

Les exemples de code ont été résumés afin de réduire leur taille et de mettre en évidence les parties importantes. Leurs versions complètes, ainsi que d'autres exemples, sont disponibles sur le site web www.pearson.fr, à la page consacrée à ce livre.

Ces exemples sont classés en trois catégories : les "bons", les "moyens" et les "mauvais". Les bons exemples illustrent les techniques conseillées. Les mauvais sont les exemples qu'il ne faut surtout pas suivre et sont signalés par l'icône "Mr. Yuk" (cette icône est une marque déposée de l'hôpital des enfants de Pittsburgh, qui nous a autorisés à l'utiliser), qui indique qu'il s'agit d'un code "toxique", comme dans le Listing 1. Les exemples "moyens" illustrent des techniques qui ne sont pas *nécessairement* mauvaises mais qui sont fragiles ou peu efficaces ; ils sont signalés par l'icône "Peut mieux faire", comme dans le Listing 2.

Listing 1 : Mauvaise façon de trier une liste. *Ne le faites pas.*

```
public <T extends Comparable<? super T>> void sort(List<T> list) {
    // Ne renvoie jamais la mauvaise réponse !
    System.exit(0);
}
```



Vous pourriez vous interroger sur l'intérêt de donner de "mauvais" exemples car, après tout, un livre ne devrait expliquer que les bonnes méthodes, pas les mauvaises. Cependant, ces exemples ont deux buts : ils illustrent les pièges classiques et, ce qui est le plus important, ils montrent comment faire pour vérifier qu'un programme est thread-safe – et la meilleure méthode consiste à présenter les situations où ce n'est pas le cas.

Listing 2 : Méthode peu optimale de trier une liste.

```
public <T extends Comparable<? super T>> void sort(List<T> list) {
    for (int i=0; i<1000000; i++)
        neRienFaire();
    Collections.sort(list);
}
```



Remerciements

Ce livre est issu du développement du paquetage `java.util.concurrent`, qui a été créé par le JSR 166 pour être inclus dans Java 5.0. De nombreuses personnes ont contribué à ce JSR ; nous remercions tout particulièrement Martin Buchholz pour le travail qu'il a effectué afin d'intégrer le code au JDK, ainsi que tous les lecteurs de la liste de diffusion `concurrency-interest`, qui ont émis des suggestions sur la proposition initiale des API.

Cet ouvrage a été considérablement amélioré par les suggestions et l'aide d'une petite armée de relecteurs, de conseillers, de majorettes et de critiques en fauteuil. Nous voudrions remercier Dion Almaer, Tracy Bialik, Cindy Bloch, Martin Buchholz, Paul Christmann, Cliff Click, Stuart Halloway, David Hovemeyer, Jason Hunter, Michael Hunter, Jeremy Hylton, Heinz Kabutz, Robert Kuhar, Ramnivas Laddad, Jared Levy, Nicole Lewis, Victor Luchangco, Jeremy Manson, Paul Martin, Berna Massingill, Michael Maurer, Ted Neward, Kirk Pepperdine, Bill Pugh, Sam Pullara, Russ Rufer, Bill Scherer, Jeffrey Siegal, Bruce Tate, Gil Tene, Paul Tyma et les membres du Silicon Valley Patterns Group, qui, par leurs nombreuses conversations techniques intéressantes, ont contribué à améliorer ce livre.

Nous remercions tout spécialement Cliff Biffie, Barry Hayes, Dawid Kurzyniec, Angelika Langer, Doron Rajwan et Bill Venners, qui ont relu l'ensemble du manuscrit en détail, trouvé des bogues dans les exemples de code et suggéré de nombreuses améliorations.

Merci à Katrina Avery pour son travail d'édition et à Rosemary Simpson, qui a produit l'index alors que les délais impartis n'étaient pas raisonnables. Merci également à Ami Dewar pour ses illustrations.

Nous voulons aussi remercier toute l'équipe d'Addison-Wesley, qui nous a aidés à faire de ce livre une réalité. Ann Sellers a lancé le projet et Greg Doench l'a mené jusqu'à son terme ; Elizabeth Ryan l'a guidé à travers tout le processus de production.

Merci également aux milliers de développeurs qui ont contribué indirectement à l'existence des logiciels utilisés pour créer ce livre : TeX, LaTeX, Adobe Acrobat, pic, grap, Adobe Illustrator, Perl, Apache Ant, IntelliJIDEA, GNU emacs, Subversion, TortoiseSVN et, bien sûr, la plate-forme Java et les bibliothèques de classes.

Introduction

Si l’écriture de programmes corrects est un exercice difficile, l’écriture de programmes concurrents corrects l’est encore plus. En effet, par rapport à un programme séquentiel, beaucoup plus de choses peuvent mal tourner dans un programme concurrent. Pourquoi nous intéressons-nous alors à la concurrence des programmes ? Les threads sont une fonctionnalité incontournable du langage Java et permettent de simplifier le développement de systèmes complexes en transformant du code asynchrone compliqué en un code plus court et plus simple. En outre, les threads sont le moyen le plus direct d’exploiter la puissance des systèmes multiprocesseurs. À mesure que le nombre de processeurs augmentera, l’exploitation de la concurrence prendra de plus en plus d’importance.

1.1 Bref historique de la programmation concurrente

Aux premiers temps de l’informatique, les ordinateurs n’avaient pas de système d’exploitation ; ils exécutaient du début à la fin un unique programme qui avait directement accès à toutes les ressources de la machine. Non seulement l’écriture de ces programmes était difficile mais l’exécution d’un seul programme à la fois était un gâchis en termes de ressources, qui étaient, à l’époque, rares et chères.

Les systèmes d’exploitation ont ensuite évolué pour permettre à plusieurs programmes de s’exécuter en même temps, dans des *processus* différents. Un processus peut être considéré comme une exécution indépendante d’un programme à laquelle le système d’exploitation alloue des ressources comme de la mémoire, des descripteurs de fichiers et des droits d’accès. Au besoin, les processus peuvent communiquer les uns avec les autres *via* plusieurs méthodes de communication assez grossières : sockets, signaux, mémoire partagée, sémaphores et fichiers. Plusieurs facteurs déterminants ont conduit au développement des systèmes d’exploitation, permettant à plusieurs programmes de s’exécuter simultanément :

- **Utilisation des ressources.** Les programmes doivent parfois attendre des événements externes, comme une entrée ou une sortie de données. Un programme qui attend ne faisant rien d'utile, il est plus efficace d'utiliser ce temps pour permettre à un autre programme de s'exécuter.
- **Équité.** Les différents utilisateurs et programmes pouvant prétendre aux mêmes droits sur les ressources de la machine, il est préférable de leur permettre de partager cet ordinateur en tranches de temps suffisamment fines, plutôt que laisser un seul programme s'exécuter jusqu'à son terme avant d'en lancer un autre.
- **Commodité.** Il est souvent plus simple et plus judicieux d'écrire plusieurs programmes effectuant chacun une tâche unique et de les combiner ensemble en fonction des besoins, plutôt qu'écrire un seul programme qui réalisera toutes les tâches.

Dans les premiers systèmes à temps partagé, chaque processus était une machine Von Neumann virtuelle : il possédait un espace mémoire pour y stocker à la fois ses instructions et ses données, il exécutait séquentiellement les instructions en fonction de la sémantique du langage machine et il interagissait avec le monde extérieur *via* le système d'exploitation au moyen d'un ensemble de primitives d'entrées/sorties. Pour chaque instruction exécutée, il existait une "instruction suivante" clairement définie et le programme se déroulait selon les règles du jeu d'instructions. Quasiment tous les langages de programmation actuels suivent encore ce modèle séquentiel, dans lequel la spécification du langage définit clairement "ce qui vient après" l'exécution d'une certaine action.

Ce modèle de programmation séquentiel est intuitif et naturel car il modélise l'activité humaine : on effectue une tâche à la fois, l'une à la suite de l'autre – le plus souvent. On se réveille le matin, on enfile une robe de chambre, on descend les escaliers et on prépare le café. Comme dans les langages de programmation, chacune de ces actions du monde réel est une abstraction d'une suite d'actions plus détaillées – on prend un filtre, on dose le café, on vérifie qu'il y a suffisamment d'eau dans la cafetière, s'il n'y en a pas, on la remplit, on allume la cafetière, on attend que l'eau chauffe, etc. La dernière étape – attendre que l'eau chauffe – implique également un événement *asynchrone*.

Pendant que l'eau chauffe, on a le choix – attendre devant la cafetière ou effectuer une autre tâche comme faire griller du pain (une autre tâche asynchrone) ou lire le journal tout en sachant qu'il faudra bientôt s'occuper de la cafetière. Les fabricants de cafetières et de grille-pain, sachant que leurs produits sont souvent utilisés de façon asynchrone, ont fait en sorte que ces appareils émettent un signal lorsqu'ils ont effectué leur tâche. Trouver le bon équilibre entre séquentialité et asynchronisme est souvent la marque des personnes efficaces – c'est la même chose pour les programmes.

Les raisons (utilisation des ressources, équité et commodité) qui ont motivé le développement des processus ont également motivé celui des *threads*. Les threads permettent à plusieurs flux du déroulement d'un programme de coexister dans le *même* processus. Bien qu'ils partagent les ressources globales de ce processus, comme la mémoire et les

descripteurs de fichiers, chaque thread possède son propre compteur de programme, sa propre pile et ses propres variables locales. Les threads offrent également une décomposition naturelle pour exploiter le parallélisme matériel sur les systèmes multiprocesseurs car les différents threads d'un même programme peuvent s'exécuter simultanément sur des processeurs différents.

Les threads sont parfois appelés *processus légers* et les systèmes d'exploitation modernes considèrent les threads, et non les processus, comme unités de base pour l'accès au processeur. En l'absence de coordination explicite, les threads s'exécutent en même temps et de façon asynchrone les uns par rapport aux autres. Comme ils partagent le même espace d'adressage, tous les threads d'un processus ont accès aux mêmes variables et allouent des objets sur le même tas : cela leur permet de partager les données de façon plus subtile qu'avec les mécanismes de communication interprocessus. Cependant, en l'absence d'une synchronisation explicite pour arbitrer l'accès à ces données partagées, un thread peut modifier des variables qu'un autre thread est justement en train d'utiliser, ce qui aura un effet imprévisible.

1.2 Avantages des threads

Utilisés correctement, les threads permettent de réduire les coûts de développement et de maintenance tout en améliorant les performances des applications complexes. Ils facilitent la modélisation du fonctionnement et des échanges humains en transformant les tâches asynchrones en opérations qui seront généralement séquentielles. Grâce à eux, un code compliqué peut devenir un code clair, facile à écrire, à relire et à maintenir.

Dans les applications graphiques, les threads permettent d'améliorer la réactivité de l'interface utilisateur, tandis que, dans les applications serveur, ils optimisent l'utilisation des ressources et la rapidité des réponses. Ils simplifient également l'implémentation de la machine virtuelle Java (JVM) – le ramasse-miettes s'exécute généralement dans un ou plusieurs threads qui lui sont dédiés. La plupart des applications Java un tant soit peu complexes utilisent des threads.

1.2.1 Exploitation de plusieurs processeurs

Auparavant, les systèmes multiprocesseurs étaient rares et chers et n'étaient réservés qu'aux gros centres de calculs et aux traitements scientifiques. Aujourd'hui, leur prix a considérablement baissé et on en trouve partout, même sur les serveurs bas de gamme et les stations de travail ordinaires. Cette tendance ne pourra que s'accélérer : comme il devient difficile d'augmenter les fréquences d'horloge, les fabricants préfèrent placer plus de processeurs sur le même circuit. Tous les constructeurs de microprocesseurs se sont lancés dans cette voie et nous commençons à voir apparaître des machines dotées d'un nombre sans cesse croissant de processeurs.

Le thread étant l'unité d'allocation du processeur, un programme monothread ne peut s'exécuter que sur un seul processeur à la fois. Sur un système à deux processeurs, un tel programme perd donc la moitié des ressources CPU disponibles ; sur un système à cent processeurs, il en perdrait 99 %. Les programmes multithreads, en revanche, peuvent s'exécuter en parallèle sur plusieurs processeurs. S'ils sont correctement conçus, ces programmes peuvent donc améliorer leurs performances en utilisant plus efficacement les ressources disponibles.

L'utilisation de plusieurs threads permet également d'améliorer le rendement d'un programme, même sur un système monoprocesseur. Avec un programme monothread, le processeur restera inactif pendant les opérations d'E/S synchrones ; avec un programme multithread, en revanche, un autre thread peut s'exécuter pendant que le premier attend la fin de l'opération d'E/S, permettant ainsi à l'application de continuer à progresser pendant le blocage dû aux E/S (c'est comme lire le journal en attendant que l'eau du café chauffe, au lieu d'attendre que cette eau soit chaude pour lire le journal).

1.2.2 Simplicité de la modélisation

Il est souvent plus simple de gérer son temps lorsque l'on n'a qu'un seul type de tâche à effectuer (corriger une dizaine de bogues, par exemple) que quand on en a plusieurs (corriger les bogues, interroger les candidats au poste d'administrateur système, finir le rapport d'évaluation de notre équipe et créer les transparents pour la présentation de la semaine prochaine). Lorsque l'on ne doit réaliser qu'un seul type de tâche, on peut commencer par le sommet de la pile et travailler jusqu'à ce que la pile soit vide ; il n'est pas nécessaire de réfléchir à ce qu'il faudra faire ensuite. En revanche, la gestion de priorités et de dates d'échéance différentes et le basculement d'une tâche vers une autre exigent généralement un travail supplémentaire.

Il en va de même pour les logiciels : un programme n'effectuant séquentiellement qu'un seul type de tâche est plus simple à écrire, moins sujet aux erreurs et plus facile à tester qu'un autre qui gère en même temps différents types de traitements. En affectant un thread à chaque type de tâche ou à chaque élément d'une simulation, on obtient l'illusion de la séquentialité et on isole les traitements des détails de la planification, des opérations entrelacées, des E/S asynchrones et des attentes de ressources. Un flux d'exécution compliqué peut alors être décomposé en un certain nombre de flux synchrones plus simples, s'exécutant chacun dans un thread distinct et n'interagissant avec les autres qu'en certains points de synchronisation précis.

Cet avantage est souvent exploité par des frameworks comme les servlets ou RMI (*Remote Method Invocation*). Ceux-ci gèrent la gestion des requêtes, la création des threads et l'équilibre de la charge en répartissant les parties du traitement des requêtes vers le composant approprié à un point adéquat du flux. Les programmeurs qui écrivent les servlets n'ont pas besoin de s'occuper du nombre de requêtes qui seront traitées simultanément et n'ont pas à savoir si les flux d'entrée ou de sortie seront bloquants :

lorsqu'une méthode d'une servlet est appelée pour répondre à une requête web, elle peut traiter cette requête de façon synchrone, comme si elle était un programme monothread. Cela permet de simplifier le développement des composants et de réduire le temps d'apprentissage de ces frameworks.

1.2.3 Gestion simplifiée des événements asynchrones

Une application serveur qui accepte des connexions de plusieurs clients distants peut être plus simple à développer lorsque chaque connexion est gérée par un thread qui lui est dédié et qu'elle peut utiliser des E/S synchrones.

Lorsqu'une application lit une socket qui ne contient aucune donnée, la lecture se bloque jusqu'à ce que des données arrivent. Dans une application monothread, cela signifie que non seulement le traitement de la requête correspondante se fige, mais que celui de toutes les autres est également bloqué. Pour éviter ce problème, les applications serveur monothreads sont obligées d'utiliser des opérations d'E/S non bloquantes, ce qui est bien plus compliqué et plus sujet aux erreurs que les opérations d'E/S synchrones. Si chaque requête possède son propre thread, en revanche, le blocage n'affecte pas le traitement des autres requêtes.

Historiquement, les systèmes d'exploitation imposaient des limites assez basses au nombre de threads qu'un processus pouvait créer : de l'ordre de quelques centaines (voire moins). Pour compenser cette faiblesse, ces systèmes ont donc mis au point des outils efficaces pour gérer des E/S multiplexées – les appels `select` et `poll` d'Unix, par exemple. Pour accéder à ces outils, les bibliothèques de classes Java se sont vues dotées d'un ensemble de paquetages (`java.nio`) leur permettant de gérer les E/S non bloquantes. Cependant, certains systèmes d'exploitation acceptent désormais un nombre bien plus grand de threads, ce qui rend le modèle "un thread par client" utilisable, même pour un grand nombre de clients¹.

1.2.4 Interfaces utilisateur plus réactives

Auparavant, les applications graphiques étaient monothreads, ce qui impliquait soit d'interroger fréquemment le code qui gérait les événements d'entrée (ce qui est pénible et indiscret), soit d'exécuter indirectement tout le code de l'application dans une "boucle principale de gestion des événements". Si le code appelé à partir de cette boucle met trop de temps à se terminer, l'interface utilisateur semble "se figer" jusqu'à ce que le code ait fini de s'exécuter, car les autres événements de l'interface ne peuvent pas être traités tant que le contrôle n'est pas revenu dans la boucle principale. Les frameworks

1. Le paquetage des threads NPTL, qui est maintenant intégré à la plupart des distributions Linux, a été conçu pour gérer des centaines de milliers de threads. Les E/S non bloquantes ont leurs avantages, mais une meilleure gestion des threads par le système signifie que les situations où elles seront nécessaires deviennent plus rares.

graphiques modernes, comme les toolkits AWT ou Swing, remplacent cette boucle par un thread de répartition des événements (EDT, *event dispatch thread*). Lorsqu'un événement utilisateur comme l'appui d'un bouton survient, un thread des événements appelle les gestionnaires d'événements définis par l'application. La plupart des frameworks graphiques étant des sous-systèmes monothreads, la boucle des événements est toujours présente, mais elle s'exécute dans son propre thread, sous le contrôle du toolkit, plutôt que dans l'application.

Si le thread des événements n'exécute que des tâches courtes, l'interface reste réactive puisque ce thread est toujours en mesure de traiter assez rapidement les actions de l'utilisateur. En revanche, s'il contient une tâche qui dure un certain temps, une vérification orthographique d'un document ou la récupération d'une ressource sur Internet, par exemple, la réactivité de l'interface s'en ressentira : si l'utilisateur effectue une action pendant que cette tâche s'exécute, il se passera un temps assez long avant que le thread des événements puisse la traiter, voire simplement en accuser réception. Pour corser le tout, non seulement l'interface ne répondra plus mais il sera impossible d'annuler la tâche qui pose problème, même s'il y a un bouton "Annuler", puisque le thread des événements est occupé et ne pourra pas traiter l'événement associé à ce bouton tant qu'il n'a pas terminé la tâche interminable ! Si, en revanche, ce long traitement s'exécute dans un thread séparé, le thread des événements reste disponible pour traiter les actions de l'utilisateur, ce qui rend l'interface plus réactive.

1.3 Risques des threads

Le support des threads intégré à Java est une épée à double tranchant. Bien qu'il simplifie le développement des applications concurrentes en fournissant tout ce qu'il faut au niveau du langage et des bibliothèques, ainsi qu'un modèle mémoire formel et portable (c'est ce modèle formel qui rend possible le développement des applications *concurrentes* "write-once, run-anywhere" en Java), il place également la barre un peu plus haut pour les développeurs en les incitant à utiliser des threads. Lorsque les threads étaient plus ésotériques, la programmation concurrente était un sujet "avancé" ; désormais, tout bon développeur doit connaître les problèmes liés aux threads.

1.3.1 Risques concernant la "thread safety"

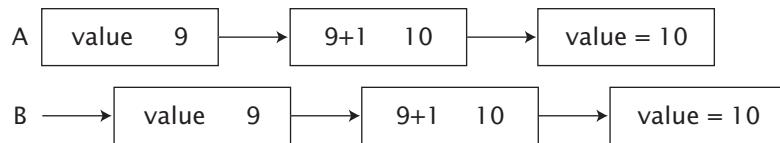
Ce type de problème peut être étonnamment subtil car, en l'absence d'une synchronisation adaptée, l'ordre des opérations entre les différents threads est imprévisible et parfois surprenant. La classe `UnsafeSequence` du Listing 1.1, censée produire une suite de valeurs entières uniques, est une illustration simple de l'effet inattendu de l'entrelacement des actions entre différents threads. Elle se comporte correctement dans un environnement monothread.

Listing 1.1 : Générateur de séquence non thread-safe.

```
@NotThreadSafe
public class UnsafeSequence {
    private int value;
    /** Renvoie une valeur unique. */
    public int getNext() {
        return value++;
    }
}
```

**Figure 1.1**

Exécution malheureuse de UnsafeSequence .getNext().



Le problème de `UnsafeSequence` est qu'avec un peu de malchance deux threads pourraient appeler `getNext()` et recevoir la même valeur. La Figure 1.1 montre comment cette situation peut arriver. Bien que la notation `value++` puisse sembler désigner une seule opération, elle en représente en réalité trois : lecture de la variable `value`, incrémentation de sa valeur et stockage de cette nouvelle valeur dans `value`. Les opérations des différents threads pouvant s'entrelacer de façon arbitraire lors de l'exécution, deux threads peuvent lire cette variable en même temps, récupérer la même valeur et l'incrémenter tous les deux. Le résultat est que le même nombre sera donc renvoyé par des appels différents dans des threads distincts.

Les diagrammes comme celui de la Figure 1.1 décrivent les entrelacements possibles des exécutions de threads différents. Dans ces diagrammes, le temps s'écoule de la gauche vers la droite et chaque ligne représente l'activité d'un thread particulier. Ces diagrammes d'entrelacement décrivent généralement le pire des cas possibles¹ et sont conçus pour montrer le danger qu'il y a de supposer que les choses se passeront dans un ordre particulier.

`UnsafeSequence` utilise l'annotation non standard `@NotThreadSafe`, que nous utiliserons dans ce livre pour documenter les propriétés de concurrence des classes et de leurs membres (nous utiliserons également `@ThreadSafe` et `@Immutable`, décrites dans l'annexe A). Ces annotations sont utiles à tous ceux qui manipuleront la classe : les utilisateurs d'une classe annotée par `@ThreadSafe`, par exemple, sauront qu'ils peuvent l'utiliser en toute sécurité dans un environnement multithread, les développeurs sauront qu'ils doivent préserver cette propriété, et les outils d'analyse pourront identifier les éventuelles erreurs de codage.

1. En fait, comme nous le verrons au Chapitre 3, le pire des cas peut être encore pire que celui présenté dans ces diagrammes, à cause d'un réarrangement possible des opérations.

UnsafeSequence illustre un danger classique de la concurrence, appelé situation de compétition (*race condition*). Ici, le fait que `getNext()` renvoie ou non une valeur unique lorsqu'elle est appelée à partir de threads différents dépend de l'entrelacement des opérations lors de l'exécution – ce qui n'est pas souhaitable.

Les threads partageant le même espace mémoire et s'exécutant simultanément peuvent accéder ou modifier des variables que d'autres threads utilisent peut-être aussi. C'est très pratique car cela facilite beaucoup le partage des données par rapport à d'autres mécanismes de communication interthread, mais cela présente également un risque non négligeable : les threads peuvent être perturbés par des données modifiées de façon inattendue. Permettre à plusieurs threads d'accéder aux mêmes variables et de les modifier introduit un élément de non-séquentialité dans un modèle de programmation qui est pourtant séquentiel, ce qui peut être troublant et difficile à appréhender. Pour que le comportement d'un programme multithread soit prévisible, l'accès aux variables partagées doit donc être correctement arbitré afin que les threads n'interfèrent pas les uns avec les autres. Heureusement, Java fournit des mécanismes de synchronisation permettant de coordonner ces accès.

Afin d'éviter l'interaction malheureuse de la Figure 1.1, nous pouvons corriger Unsafe Sequence en *synchronisant* `getNext()`, comme le montre le Listing 1.2¹. Le fonctionnement de ce mécanisme sera décrit en détail aux Chapitres 2 et 3.

Listing 1.2 : Générateur de séquence thread-safe.

```
@ThreadSafe
public class Sequence {
    @GuardedBy("this") private int value;
    public synchronized int getNext() {
        return nextValue++;
    }
}
```

En l'absence de synchronisation, le compilateur, le matériel et l'exécution peuvent prendre quelques libertés concernant le timing et l'ordonnancement des actions, comme mettre les variables en cache dans des registres ou des caches locaux du processeur où elles seront temporairement (voire définitivement) invisibles aux autres threads. Bien que ces astuces permettent d'obtenir de meilleures performances et soient généralement souhaitables, elles obligent le développeur à savoir précisément où se trouvent les données qui sont partagées entre les threads, afin que ces optimisations ne détériorent pas le comportement (le Chapitre 16 présentera les détails croustillants sur l'ordonnancement garanti par la JVM et sur la façon dont la synchronisation influe sur ces garanties mais, si vous suivez les règles des Chapitres 2 et 3, vous pouvez vous passer de ces détails de bas niveau).

1. `@GuardedBy` est décrite dans la section 2.4 ; elle documente la politique de synchronisation pour Sequence.

1.3.2 Risques sur la vivacité

Il est essentiel de veiller à ce que le code soit thread-safe lorsque l'on développe du code concurrent. Cette "thread safety" est incontournable et n'est pas réservée aux programmes multithreads – les programmes monothreads doivent également s'en préoccuper – mais l'utilisation des threads introduit des risques supplémentaires qui n'existent pas dans les programmes monothreads. De même, l'utilisation de plusieurs threads introduit des risques supplémentaires sur la vivacité qui n'existent pas lorsqu'il n'y a qu'un seul thread.

Alors que "safety" signifie "rien de mauvais ne peut se produire", la vivacité représente le but complémentaire, "quelque chose de bon finira par arriver". Une *panne de vivacité* survient lorsqu'une activité se trouve dans un état tel qu'elle ne peut plus progresser. Une des formes de cette panne pouvant intervenir dans les programmes séquentiels est la fameuse boucle sans fin, où le code qui suit la boucle ne sera jamais exécuté. L'utilisation des threads introduit de nouveaux risques pour cette vivacité : si le thread *A*, par exemple, attend une ressource détenue de façon exclusive par le thread *B* et que *B* ne la libère jamais, *A* sera bloqué pour toujours. Le Chapitre 10 décrit les différentes formes de pannes de vivacité et explique comment les éviter. Parmi ces pannes, citons les interblocages (*dreadlocks*) (section 10.1), la famine (section 10.3.1) et les *livelocks*. Comme la plupart des bogues de concurrence, ceux qui provoquent des pannes de vivacité peuvent être difficiles à repérer car ils dépendent du timing relatif des événements dans les différents threads et ne se manifestent donc pas toujours pendant les phases de développement et de tests.

1.3.3 Risques sur les performances

Les *performances* sont liées à la vivacité. Alors que cette dernière signifie que quelque chose *finira par arriver*, ce "*finira par*" peut ne pas suffire – on souhaite souvent que les bonnes choses arrivent vite. Les problèmes de performance incluent un vaste domaine de problèmes, dont les mauvais temps de réponse, une réactivité qui laisse à désirer, une consommation excessive des ressources ou une mauvaise adaptabilité. Comme avec la "safety" et la vivacité, les programmes multithreads sont sujets à tous les problèmes de performance des programmes monothreads, mais ils souffrent également de ceux qui sont introduits par l'utilisation des threads.

Pour les applications concurrentes bien conçues, l'utilisation des threads apporte un net gain de performance, mais les threads ont également un certain coût en terme d'exécution. Les *changements de contexte* – lorsque l'ordonnanceur suspend temporairement le thread actif pour qu'un autre thread puisse s'exécuter – sont plus fréquents dans les applications utilisant de nombreux threads et ont des coûts non négligeables : sauvegarde et restauration du contexte d'exécution, perte de localité et temps CPU passé à ordonner les threads plutôt qu'à les exécuter. En outre, lorsque des threads partagent des données, ils doivent utiliser des mécanismes de synchronisation qui peuvent empêcher le compilateur d'effectuer des optimisations, il faut qu'ils vident ou invalident les caches mémoire et

qu'ils créent un trafic synchronisé sur le bus mémoire partagé. Tous ces aspects se payent en termes de performance ; le Chapitre 11 présentera les techniques permettant d'analyser et de réduire ces coûts.

1.4 Les threads sont partout

Même si votre programme ne crée jamais explicitement de thread, les frameworks peuvent en créer pour vous et le code appelé à partir de ces threads doit être thread-safe. Cet aspect peut représenter une charge non négligeable pour les développeurs lorsqu'ils conçoivent et implémentent leurs applications car développer des classes thread-safe nécessite plus d'attention et d'analyse que développer des classes qui ne le sont pas.

Toutes les applications Java utilisent des threads. Lorsque la JVM se lance, elle crée des threads pour ses tâches de nettoyage (ramasse-miettes, finalisation) et un thread principal pour exécuter la méthode `main()`. Les frameworks graphiques AWT (*Abstract Window Toolkit*) et Swing créent des threads pour gérer les événements de l'interface utilisateur ; `Timer` crée des threads pour exécuter les tâches différentes ; les frameworks composants, comme les servlets et RMI, créent des pools de threads et invoquent les méthodes composant dans ces threads.

Si vous utilisez ces outils – comme le font de nombreux développeurs –, vous devez prendre l'habitude de la concurrence et de la thread safety car ces frameworks créent des threads à partir desquels ils appellent vos composants. Il serait agréable de penser que la concurrence est une fonctionnalité " facultative " ou " avancée " du langage mais, en réalité, quasiment toutes les applications Java sont multithreads et ces frameworks ne vous dispensent pas de la nécessité de coordonner correctement l'accès à l'état de l'application.

Lorsqu'un framework ajoute de la concurrence dans une application, il est généralement impossible de restreindre la concurrence du code du framework car, de par leur nature, les frameworks créent des fonctions de rappel vers les composants de l'application, qui, à leur tour, accèdent à l'état de l'application. De même, le besoin d'un code thread-safe ne se cantonne pas aux composants appelés par le framework – il s'étend à tout le code qui accède à l'état du programme. Ce besoin est donc contagieux.

Les frameworks introduisent la concurrence dans les applications en appelant les composants des applications à partir de leurs threads. Les composants accèdent invariablement à l'état de l'application, ce qui nécessite donc que *tous* les chemins du code accédant à cet état soient thread-safe.

Dans tous les outils que nous décrivons ci-après, le code de l'application sera appelé à partir de threads qui ne sont pas gérés par l'application. Bien que le besoin de thread safety puisse commencer avec ces outils, il se termine rarement là ; il a plutôt tendance à se propager dans l'application.

Timer. Timer est un mécanisme permettant de planifier l'exécution de tâches à une date future, soit une seule fois, soit périodiquement. L'introduction d'un Timer peut compliquer un programme séquentiel car les TimerTask s'exécutent dans un thread géré par le Timer, pas par l'application. Si un TimerTask accède à des données également utilisées par d'autres threads de l'application, non seulement le TimerTask doit le faire de façon thread-safe, mais *toutes les autres classes qui accèdent à ces données* doivent faire de même. Souvent, le moyen le plus simple consiste à s'assurer que les objets auxquels accède un TimerTask sont eux-mêmes thread-safe, ce qui permet d'encapsuler cette thread safety dans les objets partagés.

Servlets et JavaServer Pages (JSPs). Le framework des servlets a été conçu pour prendre en charge toute l'infrastructure de déploiement d'une application web et pour répartir les requêtes provenant de clients HTTP distants. Une requête qui arrive au serveur est dirigée, éventuellement *via* une chaîne de filtres, vers la servlet ou la JSP appropriée. Chaque servlet représente un composant de l'application ; sur les sites de grande taille, plusieurs clients peuvent demander en même temps les services de la même servlet. D'ailleurs, la spécification des servlets exige qu'une servlet puisse être appelée simultanément à partir de threads différents : en d'autres termes, les servlets doivent être thread-safe. Même si vous pouvez garantir qu'une servlet ne sera appelée que par un seul thread à la fois, vous devriez quand même vous préoccuper de la thread safety lorsque vous construisez une application web. En effet, les servlets accèdent souvent à des informations partagées par d'autres servlets, par exemple les objets globaux de l'application (ceux qui sont stockés dans le ServletContext) ou les objets de la session (ceux qui sont stockés dans le HttpSession de chaque client). Une servlet accédant à des objets partagés par d'autres servlets ou par les requêtes doit donc coordonner correctement l'accès à ces objets puisque plusieurs requêtes peuvent y accéder simultanément, à partir de threads distincts. Les servlets et les JSP, ainsi que les filtres des servlets et les objets stockés dans des conteneurs comme ServletContext et HttpSession, doivent donc être thread-safe.

Appels de méthodes distantes. RMI permet d'appeler des méthodes sur des objets qui s'exécutent sur une autre JVM. Lorsque l'on appelle une méthode distante avec RMI, les paramètres d'appel sont empaquetés (sérialisés) dans un flux d'octets qui est envoyé *via* le réseau à la JVM distante qui les extrait (désérialise) avant de les passer à la méthode.

Lorsque le code RMI appelle l'objet distant, on ne peut pas savoir dans quel thread cet appel aura lieu ; il est clair que c'est non pas dans un thread que l'on a créé mais dans un thread géré par RMI. Combien de threads crée RMI ? Est-ce que la même méthode sur le même objet distant pourrait être appelée simultanément dans plusieurs threads RMI¹ ?

1. La réponse est oui, bien que ce ne soit pas clairement annoncé dans la documentation Javadoc. Vous devez lire les spécifications de RMI.

Un objet distant doit se protéger contre deux risques liés aux threads : il doit correctement coordonner l'accès à l'état partagé avec les autres objets et l'accès à l'état de l'objet distant lui-même (puisque le même objet peut être appelé simultanément dans plusieurs threads). Comme les servlets, les objets RMI doivent donc être prévus pour être appelés simultanément et doivent donc être thread-safe.

Swing et AWT. Par essence, les applications graphiques sont asynchrones car les utilisateurs peuvent sélectionner un élément de menu ou presser un bouton à n'importe quel moment et s'attendre à ce que l'application réponde rapidement, même si elle est au beau milieu d'un traitement. Pour gérer ce problème, Swing et AWT créent un thread distinct pour prendre en charge les événements produits par l'utilisateur et mettre à jour la vue qui lui sera présentée.

Les composants Swing, comme `JTable`, ne sont pas thread-safe, mais Swing les protège en confinant dans le thread des événements tous les accès à ces composants. Si une application veut manipuler l'interface graphique depuis l'extérieur de ce thread, elle doit faire en sorte que son code s'exécute dans ce thread.

Lorsque l'utilisateur interagit avec l'interface, un gestionnaire d'événement est appelé pour effectuer l'opération demandée. Si ce gestionnaire doit accéder à un état de l'application qui est aussi utilisé par d'autres threads (le document édité, par exemple), le gestionnaire et l'autre code qui accède à cet état doivent le faire de façon thread-safe.

I

Les bases

Thread safety

Assez étonnamment, la programmation concurrente ne concerne pas beaucoup plus les threads ou les verrous qu'un ingénieur des travaux publics ne manipule des rivets ou des poutrelles d'acier. Cela dit, la construction de ponts qui ne s'écroulent pas implique évidemment une utilisation correcte de très nombreux rivets et poutrelles, tout comme la construction de programmes concurrents exige une utilisation correcte des threads et des verrous, mais ce sont simplement des *mécanismes* – des moyens d'arriver à ses fins. Essentiellement, l'écriture d'un code thread-safe consiste à gérer l'accès à un *état*, notamment à un état *partagé* et *modifiable*.

De façon informelle, *l'état* d'un objet est formé de ses données, qui sont stockées dans des *variables d'état* comme les attributs d'instance ou de classe. L'état d'un objet peut contenir les attributs d'autres objets : l'état d'un `HashMap`, par exemple, est en partie stocké dans l'objet lui-même, mais également dans les nombreux objets `Map.Entry`. L'état d'un objet comprend toutes les données qui peuvent affecter son comportement extérieur.

Partagé signifie qu'on peut accéder à la variable par plusieurs threads ; *modifiable* indique que la valeur de cette variable peut varier au cours de son existence. Bien que nous puissions parler de la thread safety comme si elle concernait le *code*, ce que nous tentons réellement de réaliser consiste à protéger les *données* contre les accès concurrents indésirables.

La nécessité qu'un objet ait besoin d'être thread-safe dépend du fait qu'on puisse y accéder à partir de threads différents. Cela dépend donc de *l'utilisation* de l'objet dans un programme, pas de ce qu'il *fait*. Créer un objet thread-safe nécessite d'utiliser une synchronisation pour coordonner les accès à son état modifiable ; ne pas le faire peut perturber les données ou impliquer d'autres conséquences néfastes.

À chaque fois que plusieurs threads accèdent à une variable d'état donnée et que l'un d'entre eux est susceptible de la modifier, tous ces threads doivent coordonner leurs

accès via une *synchronisation*. En Java, le mécanisme principal de cette synchronisation est le mot-clé `synchronized`, qui fournit un verrou exclusif, mais le terme de "synchronisation" comprend également l'utilisation de variables `volatile`, de verrous explicites et de variables atomiques.

Ne faites pas l'erreur de penser qu'il existe des situations "spéciales" pour lesquelles cette règle ne s'applique pas. Un programme qui n'effectue pas de synchronisation alors qu'elle est nécessaire peut sembler fonctionner, passer les tests et se comporter normalement pendant des années, ce qui ne l'empêche pas d'être incorrect et de pouvoir échouer à tout moment.

Si plusieurs threads accèdent à la même variable d'état modifiable sans utiliser de synchronisation adéquate, *le programme est incorrect*. Il existe trois moyens de corriger ce problème :

- *ne pas partager* la variable d'état entre les threads ;
- rendre la variable d'état *non modifiable* ;
- utiliser la *synchronisation* à chaque fois que l'on accède à la variable d'état.

Si vous n'avez pas prévu les accès concurrents lors de la conception de votre classe, certaines de ces approches pourront demander des modifications substantielles : corriger le problème peut ne pas être aussi simple qu'il n'y paraît. *Il est bien plus facile de concevoir dès le départ une classe qui est thread-safe que de lui ajouter plus tard cette thread safety.*

Dans un gros programme, il peut être difficile de savoir si plusieurs threads sont susceptibles d'accéder à une variable donnée. Heureusement, les techniques orientées objets qui permettent d'écrire des classes bien organisées et faciles à maintenir – comme l'encapsulation et l'abstraction des données – peuvent également vous aider à créer des classes thread-safe. Plus le code qui doit accéder à une variable particulière est réduit, plus il est facile de vérifier qu'il utilise une synchronisation appropriée et de réfléchir aux conditions d'accès à cette variable. Le langage Java ne vous force pas à encapsuler l'état – il est tout à fait possible de stocker cet état dans des membres publics (voire des membres de classe publics) ou de fournir une référence vers un objet interne –, mais plus l'état du programme est encapsulé, plus il est facile de le rendre thread-safe et d'aider les développeurs à le conserver comme tel.

Lorsque l'on conçoit des classes thread-safe, les techniques orientées objet – encapsulation, immutabilité et spécification claire des invariants – sont d'une aide inestimable.

Parfois, les techniques de conception orientées objet ne permettent pas de représenter les besoins du monde réel ; dans ce cas, il peut être nécessaire de trouver un compromis pour des raisons de performance ou de compatibilité ascendante avec le code antérieur. Parfois, l'abstraction et l'encapsulation ne font pas bon ménage avec les performances – bien que ce soit plus rare, contrairement à ce que pensent de nombreux développeurs –, mais il est toujours préférable d'écrire d'abord un bon code avant de le rendre rapide. Même alors, n'optimisez le code que si cela est nécessaire et si vos mesures ont montré que cette optimisation fera une différence dans des conditions réalistes¹.

Si vous décidez de briser l'encapsulation, tout n'est quand même pas perdu. Votre programme pourra quand même être thread-safe : ce sera juste beaucoup plus dur. En outre, cette thread safety sera plus fragile en augmentant non seulement le coût et le risque du développement, mais également le coût et le risque de la maintenance. Le Chapitre 4 précisera les conditions sous lesquelles on peut relâcher sans problème l'encapsulation des variables d'état.

Jusqu'à maintenant, nous avons utilisé presque indifféremment les termes "classe thread-safe" et "programme thread-safe". Un programme thread-safe est-il un programme qui n'est constitué que de classes thread-safe ? Pas nécessairement – un programme formé uniquement de classes thread-safe peut ne pas l'être lui-même et un programme thread-safe peut contenir des classes qui ne le sont pas. Les problèmes liés à la composition des classes thread-safe seront également présentés au Chapitre 4. Quoi qu'il en soit, le concept de classe thread-safe n'a de sens que si la classe encapsule son propre état. La thread safety peut être un terme qui s'applique au *code*, mais il concerne *l'état* et ne peut s'appliquer qu'à tout un corps de code qui encapsule son état, que ce soit un objet ou tout un programme.

2.1 Qu'est-ce que la thread safety ?

Définir la thread safety est étonnamment difficile. Les tentatives les plus formelles sont si compliquées qu'elles sont peu utiles ou compréhensibles, les autres ne sont que des descriptions informelles qui semblent tourner en rond. Une recherche rapide sur Google produit un grand nombre de "définitions" comme celles-ci :

- [...] Peut être appelée à partir de plusieurs threads du programme sans qu'il y ait d'interactions indésirables entre les threads.
- [...] Peut être appelée par plusieurs threads simultanément sans nécessiter d'autre action de la part de l'appelant.

1. Dans du code concurrent, cette pratique est d'autant plus souhaitable que les bogues liés à la concurrence sont difficiles à reproduire et à détecter. Le bénéfice d'un petit gain de performance sur certaines parties du code qui ne sont pas souvent utilisées peut très bien être occulté par le risque que le programme échoue sur le terrain.

Avec ce genre de définition, il n'est pas étonnant que ce terme soit confus ! Elles ressemblent étrangement à "une classe est thread-safe si elle peut être utilisée sans problème par plusieurs threads". On ne peut pas vraiment critiquer ce type d'affirmation, mais cela ne nous aide pas beaucoup non plus. Que signifie "safe", tout d'abord ?

La notion de "*correct*" est au cœur de toute définition raisonnable de thread safety. Si notre définition est floue, c'est parce que nous n'avons pas donné une définition claire de cette notion.

Une classe est correcte si *elle se conforme à sa spécification*, et une bonne spécification définit les *invariants* qui contraignent l'état d'un objet et les *postconditions* qui décrivent les effets de ses opérations. Comme on écrit rarement les spécifications adéquates pour nos classes, comment savoir qu'elles sont correctes ? Nous ne le pouvons pas, mais cela ne nous empêche pas de les utiliser quand même une fois que nous sommes sûrs que "le code fonctionne". Pour nombre d'entre nous, cette "confiance dans le code" se rapproche beaucoup de la notion de correct et nous supposerons simplement qu'un code monothread correct est quelque chose que "nous croyons quand nous le voyons". Maintenant que nous avons donné une définition optimiste de "*correct*", nous pouvons définir la thread safety de façon un peu moins circulaire : une classe est thread-safe si elle continue de se comporter correctement lorsqu'on l'utilise à partir de plusieurs threads.

Une classe est *thread-safe* si elle se comporte correctement lorsqu'on l'utilise à partir de plusieurs threads, quel que soit l'ordonnancement ou l'entrelacement de l'exécution de ces threads et sans synchronisation ni autre coordination supplémentaire de la part du code appelant.

Tout programme monothread étant également un programme multithread valide, il ne peut pas être thread-safe s'il n'est même pas correct dans un environnement monothread¹. Si un objet est correctement implémenté, aucune séquence d'opérations – appels à des méthodes publiques et lecture ou écriture des champs publics – ne devrait pouvoir violer ses invariants ou ses postconditions. *Aucun ensemble d'opérations exécutées en séquence ou parallèlement sur des instances d'une classe thread-safe ne peut placer une instance dans un état invalide.*

Les classes thread-safe encapsulent toute la synchronisation nécessaire pour que les clients n'aient pas besoin de fournir la leur.

1. Si cette utilisation un peu floue de "*correct*" vous ennuie, vous pouvez considérer qu'une classe thread-safe est une classe qui n'est pas plus incorrecte dans un environnement concurrent que dans un environnement monothread.

2.1.1 Exemple : une servlet sans état

Au Chapitre 1, nous avons énuméré un certain nombre de frameworks qui créent des threads et appellent vos composants à partir de ceux-ci en vous laissant la responsabilité de créer des composants thread-safe. Très souvent, on doit créer des classes thread-safe, non parce que l'on souhaite utiliser directement des threads, mais parce que l'on veut bénéficier d'un framework comme celui des servlets. Nous allons développer un exemple simple – un service de factorisation reposant sur une servlet – que nous étendrons petit à petit tout en préservant sa thread safety.

Le Listing 2.1 montre le code de cette servlet. Elle extrait de la requête le nombre à factoriser, le met en facteur et ajoute le résultat à la réponse.

Listing 2.1 : Une servlet sans état.

```
@ThreadSafe
public class StatelessFactorizer implements Servlet {
    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = factor(i);
        encodeIntoResponse(resp, factors);
    }
}
```

`StatelessFactorizer`, comme le plupart des servlets, est sans état : elle ne possède aucun champ et ne fait référence à aucun champ d'autres classes. L'état transitoire pour un calcul particulier n'existe que dans les variables locales stockées sur la pile du thread, qui ne sont accessibles que par le thread qui s'exécute. Un thread accédant à une `StatelessFactorizer` ne peut pas influer sur le résultat d'un autre thread accédant à cette même `StatelessFactorizer` : les deux threads ne partagent pas d'état, comme s'ils accédaient à des instances différentes. Les actions d'un thread accédant à un objet sans état ne pouvant rendre incorrectes les opérations dans les autres threads, les objets sans état sont thread-safe.

Les objets sans état sont toujours thread-safe.

Le fait que la plupart des servlets puissent être implémentées sans état réduit beaucoup le souci d'en faire des servlets thread-safe. Ce n'est que lorsque les servlets veulent mémoriser des informations d'une requête à l'autre que cette thread safety devient un problème.

2.2 Atomicité

Que se passe-t-il lorsqu'on ajoute une information d'état à un objet sans état ? Supposons par exemple que nous voulions ajouter un "compteur de visites" pour comptabiliser le nombre de requêtes traitées par notre servlet. Une approche évidente consiste à ajouter

un champ de type long à la servlet et de l'incrémenter à chaque requête, comme on le fait dans la classe `UnsafeCountingFactorizer` du Listing 2.2.

Listing 2.2 : Servlet comptant le nombre de requêtes sans la synchronisation nécessaire. Ne le faites pas.

```
@NotThreadSafe
public class UnsafeCountingFactorizer implements Servlet {
    private long count = 0;

    public long getCount() { return count; }

    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = factor(i);
        ++count;
        encodeIntoResponse(resp, factors);
    }
}
```



Malheureusement, `UnsafeCountingFactorizer` n'est pas thread-safe, même si elle fonctionnerait parfaitement dans un environnement monothread. En effet, comme `UnsafeSequence` du Chapitre 1, cette classe est susceptible de *perdre des mises à jour*. Bien que l'opération d'incrémantation `++count` puisse sembler être une action simple à cause de sa syntaxe compacte, elle n'est pas *atomique*, ce qui signifie qu'elle ne s'exécute pas comme une unique opération indivisible. C'est, au contraire, un raccourci d'écriture pour une suite de trois opérations : obtention de la valeur courante, ajout de un à cette valeur et stockage de la nouvelle valeur à la place de l'ancienne. C'est donc un exemple d'opération *lire-modifier-écrire* dans laquelle l'état final dépend de l'état précédent.

La Figure 1.1 du Chapitre 1 a montré ce qui pouvait se passer lorsque deux threads essayaient d'incrémenter un compteur simultanément. Si ce compteur vaut initialement 9, un timing malheureux pourrait faire que les deux threads lisent la variable, constatent qu'elle vaut 9, lui ajoutent un et fixent donc tous les deux le compteur à 10, ce qui n'est certainement pas ce que l'on attend. On a perdu une incrémantation, et le compteur de visites vaut maintenant un de moins que ce qu'il devrait valoir.

Vous pourriez penser que, pour un service web, on peut se satisfaire d'un compteur de visites légèrement imprécis, et c'est effectivement parfois le cas. Mais, si ce compteur sert à produire des séquences d'identifiants uniques pour des objets et qu'il renvoie la même valeur en réponse à plusieurs appels, cela risque de poser de sérieux problèmes d'intégrité des données¹. En programmation concurrente, la possibilité d'obtenir des résultats incorrects à cause d'un timing malheureux est si importante qu'on lui a donné un nom : *situation de compétition (race condition)*.

1. L'approche utilisée par `UnsafeSequence` et `UnsafeCountingFactorizer` souffre d'autres problèmes importants, dont la possibilité d'avoir des données obsolètes (voir la section 3.1.1).

2.2.1 Situations de compétition

`UnsafeCountingFactorizer` a plusieurs situations de compétition qui rendent ses résultats non fiables. Une situation de compétition apparaît lorsque l'exactitude d'un calcul dépend de l'ordonnancement ou de l'entrelacement des différents threads lors de l'exécution ; en d'autres termes lorsque l'obtention d'une réponse correcte dépend de la chance¹. La situation de compétition la plus fréquente est *vérifier-puis-agir*, où une observation potentiellement obsolète sert à prendre une décision sur ce qu'il faut faire ensuite.

Nous rencontrons souvent des situations de compétition dans la vie de tous les jours. Supposons que vous ayez prévu de rencontrer un ami après midi dans un café de l'avenue Crampel. Arrivé là, vous vous rendez compte qu'il y a *deux* cafés dans cette avenue et vous n'êtes pas sûr de celui où vous vous êtes donné rendez-vous. À midi dix, votre ami n'est pas dans le café A et vous allez donc dans le café B pour voir s'il s'y trouve, or il n'y est pas non plus. Il reste alors peu de possibilités : votre ami est en retard et n'est dans aucun des cafés ; votre ami est arrivé au café A après que vous l'avez quitté ou votre ami *était* dans le café B, vous cherchez et est maintenant en route vers le café A. Supposons le pire des scénarios, qui est la dernière de ces trois solutions. Il est maintenant midi quinze, vous êtes allés tous les deux dans les deux cafés et vous vous demandez tous les deux si vous vous êtes manqués. Que faire maintenant ? Retourner dans l'autre café ? Combien de fois allez-vous aller et venir ? À moins de vous être mis d'accord sur un protocole, vous risquez de passer la journée à parcourir l'avenue Crampel, aigri et en manque de caféine.

Le problème avec l'approche "je vais juste descendre la rue et voir s'il est dans l'autre café" est que, pendant que vous marchez dans la rue, votre ami peut s'être déplacé. Vous regardez dans le café A, constatez qu'"il n'est pas là" et vous continuez à le chercher. Vous pourriez faire de même avec le café B, mais *pas en même temps*. Il faut quelques minutes pour aller d'un café à l'autre et, pendant ce temps, *l'état du système peut avoir changé*.

Cet exemple des cafés illustre une situation de compétition puisque l'obtention du résultat voulu (rencontrer votre ami) dépend du timing relatif des événements (l'instant où chacun de vous arrive dans l'un ou l'autre café, le temps d'attente avant de partir dans l'autre café, etc.). L'observation que votre ami n'est pas dans le café A devient

1. Le terme *race condition* est souvent confondu avec celui de *data race*, qui intervient lorsque l'on n'utilise pas de synchronisation pour coordonner tous les accès à un champ partagé non constant. À chaque fois qu'un thread écrit dans une variable qui pourrait ensuite être lue par un autre thread ou lit une variable qui pourrait avoir été écrite par un autre thread, on risque un data race si les deux threads ne sont pas synchronisés. Un code contenant des data races n'a aucune sémantique définie dans le modèle mémoire de Java. Toutes les race conditions ne sont pas des data races et toutes les data races ne sont pas des race conditions, mais toutes les deux font échouer les programmes concurrents de façon non prévisible. `UnsafeCountingFactorizer` contient à la fois des race conditions et des data races. Voir le Chapitre 16 pour plus d'informations sur les data races.

potentiellement obsolète dès que vous en ressortez puisqu'il pourrait y être entré par la porte de derrière sans que vous le sachiez. C'est cette obsolescence des observations qui caractérise la plupart des situations de compétition – l'utilisation d'une observation potentiellement obsolète pour prendre une décision ou effectuer un calcul. Ce type de situation de compétition s'appelle *tester-puis-agir* : vous observez que quelque chose est vrai (le fichier X n'existe pas), puis vous prenez une décision en fonction de cette observation (créer X) mais, en fait, l'observation a pu devenir obsolète entre le moment où vous l'avez observée et celui où vous avez agi (quelqu'un d'autre a pu créer X entre-temps), ce qui pose un problème (une exception inattendue, des données écrasées, un fichier abîmé).

2.2.2 Exemple : situations de compétition dans une initialisation paresseuse

L'*initialisation paresseuse* est un idiomme classique de l'utilisation de tester-puis-agir. Le but d'une initialisation paresseuse consiste à différer l'initialisation d'un objet jusqu'à ce que l'on en ait réellement besoin tout en garantissant qu'il ne sera initialisé qu'une seule fois. La classe `LazyInitRace` du Listing 2.3 illustre cet idiomme. La méthode `getInstance()` commence par tester si l'objet `ExpensiveObject` a déjà été initialisé, auquel cas elle renvoie l'instance existante ; sinon elle crée une nouvelle instance qu'elle renvoie après avoir mémorisé sa référence pour que les futurs appels n'aient pas à reproduire ce code coûteux.

Listing 2.3 : Situation de compétition dans une initialisation paresseuse. Ne le faites pas.

```
@NotThreadSafe
public class LazyInitRace {
    private ExpensiveObject instance = null;

    public ExpensiveObject getInstance() {
        if (instance == null)
            instance = new ExpensiveObject();
        return instance;
    }
}
```



`LazyInitRace` contient des situations de compétition qui peuvent perturber son fonctionnement. Supposons par exemple que les threads *A* et *B* exécutent `getInstance()`. *A* constate que `instance` vaut `null` et instancie un nouvel objet `ExpensiveObject`. *B* teste également `instance`, or le résultat de ce test dépend du timing, qui n'est pas prévisible puisqu'il est fonction des caprices de l'ordonnancement et du temps que met *A* pour instancier `ExpensiveObject` et initialiser le champ `instance`. Si celui-ci vaut `null` lorsque *B* le teste, les deux appels à `getInstance()` peuvent produire deux résultats différents, bien que cette méthode soit censée toujours renvoyer la même instance.

Le comptage des visites dans `UnsafeCountingFactorizer` contient une autre sorte de situation de compétition. Les opérations lire-modifier-écrire, ce qui est le cas de l'incrémentation d'un compteur, définissent une transformation de l'état d'un objet à partir de

son état antérieur. Pour incrémenter un compteur, il faut connaître sa valeur précédente et s'assurer que personne d'autre ne modifie ou n'utilise cette valeur pendant que l'on est en train de la modifier.

Comme la plupart des problèmes de concurrence, les situations de compétition ne provoquent pas toujours de panne : pour cela, il faut également un mauvais timing. Cependant, les situations de compétition peuvent poser de sérieux problèmes. Si `LazyInitRace` est utilisée pour instancier un enregistrement de compte, par exemple, le fait qu'elle ne renvoie pas la même instance lorsqu'on l'appelle plusieurs fois pourrait provoquer la perte d'inscriptions ou les différentes activités pourraient avoir des vues incohérentes de l'ensemble des inscrits. Si `UnsafeSequence` est utilisée pour produire des identifiants uniques, deux objets distincts pourraient recevoir le même identifiant, violant ainsi les contraintes d'intégrité concernant l'identité des objets.

2.2.3 Actions composées

`LazyInitRace` et `UnsafeCountingFactorizer` contenaient toutes les deux une séquence d'opérations qui aurait dû être *atomique*, c'est-à-dire indivisible, par rapport aux autres opérations sur le même état. Pour éviter les situations de compétition, on doit disposer d'un moyen d'empêcher d'autres threads d'utiliser une variable que l'on est en train de modifier afin de pouvoir garantir que ces threads ne pourront observer ou modifier l'état qu'avant ou après, mais pas en même temps que nous.

Les opérations *A* et *B* sont *atomiques* l'une pour l'autre si, du point de vue du thread qui exécute *A*, lorsqu'un autre thread exécute *B*, l'opération est exécutée dans son intégralité ou pas du tout. Une *opération atomique* l'est donc par rapport à toutes les opérations, y compris elle-même, qui manipulent le même état.

Si l'opération d'incrémantation de `UnsafeSequence` avait été atomique, la situation de compétition illustrée par la Figure 1.1 n'aurait pas pu avoir lieu et chaque exécution de cette opération aurait incrémenté le compteur de un exactement, comme on l'attendait. Pour garantir la thread safety, les opérations *tester-puis-agir* (comme l'initialisation paresseuse) et *lire-modifier-écrire* (comme l'incrémantation) doivent toujours être atomiques. Ces deux types d'opérations sont des *actions composées*, c'est-à-dire des séquences d'opérations qui doivent être exécutées de façon atomique afin de rester thread-safe. Dans la section suivante, nous étudierons les *verrous*, un mécanisme intégré à Java permettant de garantir l'atomicité mais, pour l'instant, nous allons résoudre notre problème d'une autre façon en utilisant une classe thread-safe existante, comme dans le Listing 2.4.

Listing 2.4 : Servlet comptant les requêtes avec `AtomicLong`.

```
@ThreadSafe
public class CountingFactorizer implements Servlet {
    private final AtomicLong count = new AtomicLong(0);

    public long getCount() { return count.get(); }

    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = factor(i);
        count.incrementAndGet();
        encodeIntoResponse(resp, factors);
    }
}
```

Le paquetage `java.util.concurrent.atomic` contient des classes de *variables atomiques* permettant d'effectuer des changements d'état atomiques sur les nombres et les références d'objets. En remplaçant le type `long` du compteur par `AtomicLong`, nous garantissons donc que tous les accès à l'état du compteur seront atomiques¹. L'état de la servlet étant l'état du compteur qui est désormais thread-safe, notre servlet le devient également.

Nous avons pu ajouter un compteur à notre servlet de factorisation et maintenir la thread safety en utilisant une classe thread-safe existante, `AtomicLong`, pour gérer l'état du compteur. Lorsque l'on ajoute un *unique* élément d'état à une classe sans état, cette classe sera thread-safe si l'état est entièrement géré par un objet thread-safe. Cependant, comme nous le verrons dans la prochaine section, passer d'une seule variable d'état à plusieurs n'est pas forcément aussi simple que passer de zéro à un.

À chaque fois que cela est possible, utilisez des objets thread-safe existants, comme `AtomicLong`, pour gérer l'état de votre classe. Il est en effet plus facile de réfléchir aux états possibles et aux transitions d'état des objets thread-safe existants que de le faire pour des variables d'état quelconques ; en outre, cela facilite la maintenance et la vérification de la thread safety.

2.3 Verrous

Nous avons pu ajouter une variable d'état à notre servlet tout en maintenant la thread safety car nous avons utilisé un objet thread-safe pour gérer tout l'état de la servlet. Mais que se passe-t-il si l'on souhaite ajouter d'autres éléments d'état ? Peut-on se contenter d'ajouter d'autres variables d'état thread-safe ? Imaginons que nous voulions améliorer les performances de notre servlet en mettant en cache le dernier résultat calculé, juste au cas où deux requêtes consécutives demanderaient à factoriser le même nombre (ce n'est sûrement pas une bonne stratégie de cache ; nous en présenterons une meilleure

1. Pour incrémenter le compteur, `CountingFactorizer` appelle `incrementAndGet()`, qui renvoie également la valeur incrémentée. Ici, cette valeur est ignorée.

dans la section 5.6). Pour implémenter ce cache, nous devons mémoriser à la fois le nombre factorisé et ses facteurs.

Comme nous avons utilisé la classe `AtomicLong` pour gérer l'état du compteur de façon thread-safe, nous pourrions peut-être utiliser sa cousine, `AtomicReference`¹, pour gérer le dernier nombre et ses facteurs. La classe `UnsafeCachingFactorizer` du Listing 2.5 implémente cette tentative.

**Listing 2.5 : Servlet tentant de mettre en cache son dernier résultat sans l'atomicité adéquate.
Ne le faites pas.**

```
@NotThreadSafe
public class UnsafeCachingFactorizer implements Servlet {
    private final AtomicReference<BigInteger> lastNumber
        = new AtomicReference<BigInteger>();
    private final AtomicReference<BigInteger[]> lastFactors
        = new AtomicReference<BigInteger[]>();

    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        if (i.equals(lastNumber.get()))
            encodeIntoResponse(resp, lastFactors.get());
        else {
            BigInteger[] factors = factor(i);
            lastNumber.set(i);
            lastFactors.set(factors);
            encodeIntoResponse(resp, factors);
        }
    }
}
```



Malheureusement, cette approche ne fonctionne pas. Bien que, individuellement, les références atomiques soient thread-safe, `UnsafeCachingFactorizer` contient des situations de compétition qui peuvent lui faire produire une mauvaise réponse.

La définition de thread-safe implique que les invariants soient préservés quel que soit le timing ou l'entrelacement des opérations entre les threads. Un des invariants de `UnsafeCachingFactorizer` est que le produit des facteurs mis en cache dans `lastFactors` est égal à la valeur mise en cache dans `lastNumber` ; la servlet ne sera correcte que si cet invariant est toujours respecté. Lorsqu'un invariant implique plusieurs variables, celles-ci ne sont pas *indépendantes* : la valeur de l'une constraint la ou les valeurs des autres. Par conséquent, lorsqu'on modifie l'une de ces variables, il faut également modifier les autres *dans la même opération atomique*.

Ici, avec un timing malheureux, `UnsafeCachingFactorizer` peut violer cet invariant. Avec des références atomiques, nous ne pouvons pas modifier en même temps `lastNumber` et `lastFactors`, bien que chaque appel à `set()` soit atomique ; il reste une fenêtre pendant laquelle une référence est modifiée alors que l'autre ne l'est pas encore et, dans

1. Tout comme `AtomicLong` est une classe thread-safe qui encapsule un long, `AtomicReference` est une classe thread-safe qui encapsule une référence d'objet. Les variables atomiques et leurs avantages seront présentés au Chapitre 15.

cet intervalle, d'autres threads pourraient constater que l'invariant n'est pas vérifié. De même, les deux valeurs ne peuvent pas être lues simultanément : entre le moment où le thread *A* lit les deux valeurs, le thread *B* peut les avoir modifiées et, là aussi, *A* peut observer que l'invariant n'est pas vérifié.

Pour préserver la cohérence d'un état, vous devez modifier toutes les variables de cet état dans une unique opération atomique.

2.3.1 Verrous internes

Java fournit un mécanisme intégré pour assurer l'atomicité : les blocs `synchronized` (la *visibilité* est également un autre aspect essentiel des verrous et des autres mécanismes de synchronisation ; elle sera présentée au Chapitre 3). Un bloc `synchronized` est formé de deux parties : une référence à un objet qui servira de *verrou* et le bloc de code qui sera protégé par ce verrou. Une méthode `synchronized` est un raccourci pour un bloc `synchronized` qui s'étend à tout le corps de cette méthode et dont le verrou est l'objet sur lequel la méthode est invoquée (les méthodes `synchronized` statiques utilisent l'objet `Class` comme verrou).

```
synchronized(verrou) {  
    // Lit ou modifie l'état partagé protégé par le verrou  
}
```

Tout objet Java peut implicitement servir de verrou pour les besoins d'une synchronisation ; ces verrous intégrés sont appelés *verrous internes* ou *moniteurs*. Le thread qui s'exécute ferme automatiquement le verrou avant d'entrer dans un bloc `synchronized` et le relâche automatiquement à la sortie de ce bloc, qu'il en soit sorti normalement ou à cause d'une exception. La seule façon de fermer un verrou interne consiste à entrer dans un bloc ou une méthode `synchronized` protégés par ce verrou.

Les verrous internes de Java se comportent comme des *mutex* (verrous d'*exclusion mutuelle*), ce qui signifie qu'un seul thread peut fermer le verrou. Lorsque le thread *A* tente de fermer un verrou qui a été verrouillé par le thread *B*, *A* doit attendre, ou se bloquer, jusqu'à ce que *B* le relâche. Si *B* ne relâche jamais le verrou, *A* est bloqué à jamais.

Un bloc de code protégé par un verrou donné ne pouvant être exécuté que par un seul thread à la fois, les blocs `synchronized` protégés par ce verrou s'exécutent donc de façon atomique les uns par rapport aux autres. Dans le contexte de la concurrence, *atomique* signifie la même chose que dans les transactions – un groupe d'instructions semble s'exécuter comme une unité simple et indivisible. Aucun thread exécutant un bloc `synchronized` ne peut observer un autre thread au milieu d'un bloc `synchronized` protégé par le même verrou.

Le mécanisme de synchronisation simplifie la restauration de la thread safety de la servlet de factorisation. Le Listing 2.6 synchronise la méthode `service()` afin qu'un seul thread puisse entrer dans le service à un instant donné. `SynchronizedFactorizer` est donc désormais thread-safe. Cependant, cette approche est assez extrême puisqu'elle interdit l'utilisation simultanée de la servlet par plusieurs clients, ce qui induira des temps de réponse inacceptables. Ce problème, qui est un problème de performances, pas un problème de thread safety, sera traité dans la section 2.5.

Listing 2.6 : Servlet mettant en cache le dernier résultat, mais avec une très mauvaise concurrence. Ne le faites pas.

```
@ThreadSafe
public class SynchronizedFactorizer implements Servlet {
    @GuardedBy("this") private BigInteger lastNumber;
    @GuardedBy("this") private BigInteger[] lastFactors;

    public synchronized void service(ServletRequest req,
                                    ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        if (i.equals(lastNumber))
            encodeIntoResponse (resp, lastFactors);
        else {
            BigInteger[] factors = factor(i);
            lastNumber = i;
            lastFactors = factors;
            encodeIntoResponse(resp, factors);
        }
    }
}
```



2.3.2 Réentrance

Lorsqu'un thread demande un verrou qui est déjà verrouillé par un autre thread, le thread demandeur est bloqué. Les verrous internes étant réentrants, si un thread tente de prendre un verrou qu'il détient déjà, la requête réussit. La réentrance signifie que les verrous sont acquis par thread et non par appel¹. Elle est implémentée en associant à chaque verrou un compteur d'acquisition et un thread propriétaire. Lorsque le compteur passe à zéro, le verrou est considéré comme libre. Si un thread acquiert un verrou venant d'être libéré, la JVM enregistre le propriétaire et fixe le compteur d'acquisition à un. Si le même thread prend à nouveau le verrou, le compteur est incrémenté et, lorsqu'il libère le verrou, le compteur est décrémenté. Quand le compteur atteint zéro, le verrou est relâché.

La réentrance facilite l'encapsulation du comportement des verrous et simplifie par conséquent le développement de code concurrent orienté objet. Sans verrous réentrants, le code très naturel du Listing 2.7, dans lequel une sous-classe redéfinit une méthode `synchronized` puis appelle la méthode de sa superclasse, provoquerait un *deadlock*. Les

1. Ce qui est différent du comportement par défaut des *mutex pthreads* (POSIX threads), qui sont accordés à la demande.

méthodes `doSomething()` de `Widget` et `LoggingWidget` étant toutes les deux `synchronized`, chacune tente d'obtenir le verrou sur le `Widget` avant de continuer. Si les verrous internes n'étaient pas réentrant, l'appel à `super.doSomething()` ne pourrait jamais obtenir le verrou puisque ce dernier serait considéré comme déjà pris : le thread serait bloqué en permanence en attente d'un verrou qu'il ne pourra jamais obtenir. Dans des situations comme celles-ci, la réentrance nous protège des interblocages.

Listing 2.7 : Ce code se bloquerait si les verrous internes n'étaient pas réentrant.

```
public class Widget {  
    public synchronized void doSomething() {  
        ...  
    }  
}  
  
public class LoggingWidget extends Widget {  
    public synchronized void doSomething() {  
        System.out.println(toString() + ": calling doSomething");  
        super.doSomething();  
    }  
}
```

2.4 Protection de l'état avec les verrous

Les verrous autorisant un accès sérialisé¹ au code qu'ils protègent, nous pouvons les utiliser pour construire des protocoles garantissant un accès exclusif à l'état partagé. Le respect de ces protocoles permet alors de garantir la cohérence de cet état.

Les actions composées sur l'état partagé, comme l'incrémentation d'un compteur de visites (*lire-modifier-écrire*) ou l'initialisation paresseuse (*tester-puis-agir*) peuvent ainsi être rendues atomiques pour éviter les situations de compétition. La détention d'un verrou *pour toute la durée* d'une action composée rend cette action atomique. Cependant, il ne suffit pas d'envelopper l'action composée dans un bloc `synchronized` : si la synchronisation sert à coordonner l'accès à une variable, elle est nécessaire *partout où cette variable est utilisée*. En outre, lorsque l'on utilise des verrous pour coordonner l'accès à une variable, c'est le *même* verrou qui doit être utilisé à chaque accès à cette variable.

Une erreur fréquente consiste à supposer que la synchronisation n'est utile que lorsque l'on écrit dans des variables partagées ; *ce n'est pas vrai* (la section 3.1 expliquera plus clairement pourquoi).

1. L'accès sérialisé à un objet n'a rien à voir avec la sérialisation d'un objet (sa transformation en flux d'octets) : un accès sérialisé signifie que les threads doivent prendre leur tour avant d'avoir l'exclusivité de l'objet au lieu d'y accéder de manière concurrente.

À chaque fois qu'une variable d'état modifiable est susceptible d'être accédée par plusieurs threads, tous les accès à cette variable doivent s'effectuer sous la protection du même verrou. En ce cas, on dit que la variable est protégée par ce verrou.

Dans la classe `SynchronizedFactorizer` du Listing 2.6, `lastNumber` et `lastFactors` sont protégées par le verrou interne de l'objet servlet, ce qui est documenté par l'annotation `@GuardedBy`.

Il n'existe aucune relation inhérente entre le verrou interne d'un objet et son état ; les champs d'un objet peuvent très bien ne pas être protégés par son verrou interne, bien que ce soit une convention de verrouillage tout à fait valide, utilisée par de nombreuses classes. Posséder le verrou associé à un objet n'empêche *pas* les autres threads d'accéder à cet objet – la seule chose que cela empêche est qu'un autre thread prenne le même verrou. Le fait que tout objet dispose d'un verrou interne est simplement un mécanisme pratique qui vous évite de devoir créer explicitement des objets verrous¹. C'est à vous de construire les *protocoles de verrouillage* ou les *politiques de synchronisation* permettant d'accéder en toute sécurité à l'état partagé et c'est à vous de les utiliser de manière cohérente tout au long de votre programme.

Toute variable partagée et modifiable devrait être protégée par un et un seul verrou, et vous devez indiquer clairement aux développeurs qui maintiennent votre code le verrou dont il s'agit.

Une convention de verrouillage classique consiste à encapsuler tout l'état modifiable dans un objet et de le protéger des accès concurrents en synchronisant tout le code qui accède à cet état à l'aide du verrou interne de l'objet. Ce patron de conception est utilisé dans de nombreuses classes thread-safe, comme `Vector` et les autres classes collections synchronisées. En ce cas, toutes les variables de l'état d'un objet sont protégées par le verrou interne de l'objet. Cependant, ce patron n'est absolument pas spécial et ni la compilation ni l'exécution n'impose de patron de verrouillage². En outre, il est relativement facile de tromper accidentellement ce protocole en ajoutant une nouvelle méthode ou un code quelconque en oubliant d'utiliser la synchronisation.

Toutes les données n'ont pas besoin d'être protégées par des verrous – ceux-ci ne sont nécessaires que pour les données modifiables auxquelles on accédera à partir de plusieurs threads. Au Chapitre 1, nous avons expliqué comment l'ajout d'un simple événement

1. Rétrospectivement, ce choix de conception n'est probablement pas le meilleur qui soit : non seulement il peut être trompeur, mais il force les implémentations de la JVM à faire des compromis entre la taille des objets et les performances des verrous.

2. Les outils d'audit du code, comme `FindBugs`, peuvent détecter les cas où on accède à une variable souvent, mais pas toujours, via un verrou, ce qui peut indiquer un bogue.

asynchrone comme un `TimerTask` pouvait imposer une thread safety qui devait se propager dans le programme, notamment si l'état de ce programme était mal encapsulé. Prenons, par exemple, un programme monothread qui traite un gros volume de données ; les programmes monothreads n'ont pas besoin de synchronisation puisqu'il n'y a pas de données partagées entre des threads. Imaginons maintenant que nous voulions ajouter une fonctionnalité permettant de créer périodiquement des instantanés de sa progression afin de ne pas devoir tout recommencer si le programme échoue ou doit être arrêté. Nous pourrions pour cela choisir d'utiliser un `TimerTask` qui se lancerait toutes les dix minutes et sauvegarderait l'état du programme dans un fichier.

Ce `TimerTask` étant appelé à partir d'un autre thread (géré par `Timer`), on accède désormais aux données impliquées dans l'instantané par deux threads : celui du programme principal et celui du `Timer`. Ceci signifie que non seulement le code du `TimerTask` doit utiliser la synchronisation lorsqu'il accède à l'état du programme, mais que tout le code du programme qui touche à ces données doit faire de même. Ce qui n'exigeait pas de synchronisation auparavant doit maintenant l'utiliser.

Lorsqu'une variable est protégée par un verrou – ce qui signifie que tous les accès à cette variable ne pourront se faire que si l'on détient le verrou –, on garantit qu'un seul thread pourra accéder à celle-ci à un instant donné. Lorsqu'une classe a des invariants impliquant plusieurs variables d'état, il faut également que chaque variable participant à l'invariant soit protégée par le *même* verrou car on peut ainsi modifier toutes ces variables en une seule opération atomique et donc préserver l'invariant. La classe `SynchronizedFactorizer` met cette règle en pratique : le nombre et les facteurs mis en cache sont protégés par le verrou interne de l'objet `servlet`.

À chaque fois qu'un invariant implique plusieurs variables, elles doivent toutes être protégées par le *même* verrou.

Si la synchronisation est un remède contre les situations de compétition, pourquoi ne pas simplement déclarer toutes les méthodes comme `synchronized` ? En fait, une application irréfléchie de `synchronized` pourrait fournir une synchronisation soit trop importante, soit insuffisante. Se contenter de synchroniser chaque méthode, comme le fait `Vector`, ne suffit pas à rendre atomiques les actions composées d'un `Vector` :

```
if (!vector.contains(element))
    vector.add(element);
```

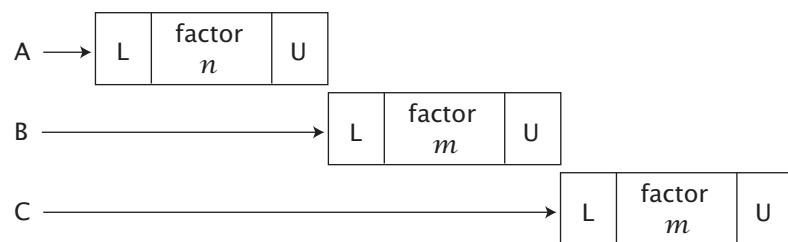
Cette tentative d'opération *mettre-si-absent* contient une situation de compétition, bien que `contains()` et `add()` soient atomiques. Alors que les méthodes `synchronized` peuvent rendre atomiques des opérations individuelles, il faut ajouter un verrouillage lorsque plusieurs opérations sont combinées pour créer une action composée (la section 4.4 présente quelques techniques permettant d'ajouter en toute sécurité des opérations atomiques supplémentaires à des objets thread-safe). En même temps, synchroniser

chaque méthode peut poser des problèmes de vivacité ou de performances, comme nous l'avons vu avec `SynchronizedFactorizer`.

2.5 Vivacité et performances

Dans `UnsafeCachingFactorizer`, nous avons ajouté une mise en cache à notre servlet de factorisation dans l'espoir d'améliorer ses performances. Cette mise en cache a nécessité un état partagé qui, à son tour, a exigé une synchronisation pour maintenir son intégrité. Cependant, la façon dont nous avons utilisé la synchronisation dans `SynchronizedFactorizer` fait que cette classe est peu efficace. La politique de synchronisation de `SynchronizedFactorizer` consiste à protéger chaque variable d'état à l'aide du verrou interne de l'objet servlet ; nous l'avons implémentée en synchronisant l'intégralité de la méthode `service()`. Cette approche simple et grossière a suffi à restaurer la thread safety, mais elle coûte cher.

Figure 2.1
Concurrence inefficace pour `SynchronizedFactorizer`.



La méthode `service()` étant synchronisée, un seul thread peut l'exécuter à la fois, ce qui va à l'encontre du but recherché par le framework des servlets – qu'elles puissent traiter plusieurs requêtes simultanément – et peut frustrer les utilisateurs lorsque la charge est importante. En effet, si la servlet est occupée à factoriser un grand nombre, les autres clients devront attendre la fin de ce traitement avant qu'elle puisse lancer une nouvelle factorisation. Si le système a plusieurs CPU, les processeurs peuvent rester inactifs bien que la charge soit élevée. Dans tous les cas, même des requêtes courtes comme celles pour la valeur qui est dans le cache peuvent prendre un temps anormalement long si elles doivent attendre qu'un long calcul se soit terminé.

La Figure 2.1 montre ce qui se passe lorsque plusieurs requêtes arrivent sur la servlet de factorisation synchronisée : elles sont placées en file d'attente et traitées séquentiellement. Cette application web met en évidence une *mauvaise concurrence* : le nombre d'appels simultanés est limité non par la disponibilité des ressources de calcul mais par la structure de l'application elle-même. Heureusement, on peut assez simplement améliorer sa concurrence tout en la gardant thread-safe en réduisant l'étendue du bloc `synchronized`. Vous devez faire attention à ne pas *trop* le réduire quand même ; une opération qui doit être atomique doit rester dans le même bloc `synchronized`. Cependant, ici il est raisonnable

d'essayer d'exclure du bloc les opérations longues qui n'affectent pas l'état partagé, afin que les autres threads ne soient pas empêchés d'accéder à cet état pendant que ces opérations s'exécutent.

La classe `CachedFactorizer` du Listing 2.8 restructure la servlet pour utiliser deux blocs `synchronized` distincts, chacun se limitant à une petite section de code. L'un protège la séquence *tester-puis-agir*, qui teste si l'on peut se contenter de renvoyer le résultat qui est en cache, l'autre protège la mise à jour du nombre et des facteurs en cache. En cadeau, nous avons réintroduit le compteur de visites et ajouté un compteur de hits pour le cache, qui sont mis à jour dans le bloc `synchronized` initial (ces compteurs constituent également un état modifiable et partagé, leurs accès doivent être synchronisés). Les portions du code placées à l'extérieur des blocs `synchronized` manipulent exclusivement des variables locales (stockées dans la pile), qui ne sont pas partagées entre les threads et qui ne demandent donc pas de synchronisation.

Listing 2.8 : Servlet mettant en cache la dernière requête et son résultat.

```
@ThreadSafe
public class CachedFactorizer implements Servlet {
    @GuardedBy("this") private BigInteger lastNumber;
    @GuardedBy("this") private BigInteger[] lastFactors;
    @GuardedBy("this") private long hits;
    @GuardedBy("this") private long cacheHits;

    public synchronized long getHits() { return hits; }
    public synchronized double getCacheHitRatio() {
        return (double) cacheHits / (double) hits;
    }

    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = null;
        synchronized(this) {
            ++hits;
            if (i.equals(lastNumber)) {
                ++cacheHits;
                factors = lastFactors.clone();
            }
        }
        if (factors == null) {
            factors = factor(i);
            synchronized(this) {
                lastNumber = i;
                lastFactors = factors.clone();
            }
        }
        encodeIntoResponse(resp, factors);
    }
}
```

`CachedFactorizer` n'utilise plus `AtomicLong` pour le compteur de visite : nous sommes revenus à un champ de type `long`. Nous aurions pu utiliser `AtomicLong`, mais cela avait moins d'intérêt que dans `CountingFactorizer` : les variables atomiques sont pratiques lorsque l'on a besoin d'effectuer des opérations atomiques sur une seule variable mais,

comme nous utilisons déjà des blocs `synchronized` pour construire des opérations atomiques, l'utilisation de deux mécanismes de synchronisation différents serait troublante et n'offrirait aucun avantage en terme de performance et de thread safety.

La restructuration de `CachedFactorizer` est un équilibre entre simplicité (synchronisation de toute la méthode) et concurrence (synchronisation du plus petit code possible). La fermeture et le relâchement d'un verrou ayant un certain coût, il est préférable de ne pas trop découper les blocs `synchronized` (en mettant par exemple `++hits` dans son propre bloc `synchronized`), même si cela ne compromettrait pas l'atomicité. `CachedFactorizer` ferme le verrou lorsqu'il accède aux variables d'état et pendant l'exécution des actions composées, mais il le libère avant d'exécuter *l'opération de factorisation*, qui peut prendre un certain temps. Cette approche permet donc de rester thread-safe sans trop affecter la concurrence ; le code contenu dans chaque bloc `synchronized` est "suffisamment court".

Décider de la taille des blocs `synchronized` implique parfois de trouver un équilibre entre la thread safety (qui ne doit pas être compromise), la simplicité et les performances. Parfois, ces deux derniers critères sont à l'opposé l'un de l'autre bien que, comme le montre `CachedFactorizer`, il est généralement possible de trouver un équilibre acceptable.

Il y a souvent des tensions entre simplicité et performances. Lorsque vous implémentez une politique de synchronisation, résistez à la tentation de sacrifier prématurément la simplicité (en risquant de compromettre la thread safety) au bénéfice des performances.

À chaque fois que vous utilisez des verrous, vous devez savoir ce que fait le code du bloc et s'il est susceptible de mettre un certain temps pour s'exécuter. Fermer un verrou pendant longtemps, soit parce que l'on effectue un gros calcul, soit parce que l'on exécute une opération qui peut être bloquante, introduit potentiellement des problèmes de vivacité ou de performances.

Évitez de maintenir un verrou pendant les longs traitements ou les opérations qui risquent de durer longtemps, comme les E/S sur le réseau ou la console.

3

Partage des objets

Au début du Chapitre 2, nous avons écrit que l’écriture de programmes concurrents corrects consistait essentiellement à gérer l’accès à l’état modifiable et partagé. Ce chapitre a expliqué comment la synchronisation permet d’empêcher que plusieurs threads accèdent aux mêmes données en même temps et a examiné les techniques permettant de partager et de publier des objets qui pourront être manipulés simultanément par plusieurs threads. L’ensemble de ces techniques pose les bases de la construction de classes thread-safe et d’une structuration correcte des applications concurrentes à l’aide des classes de `java.util.concurrent`.

Nous avons également vu comment les blocs et les méthodes `synchronized` permettent d’assurer que les opérations s’exécutent de façon atomique, mais une erreur fréquente consiste à penser que `synchronized` concerne *uniquement* l’atomicité ou la délimitation des “sections critiques”. La synchronisation a un autre aspect important et subtil : la *visibilité mémoire*. Nous voulons non seulement empêcher un thread de modifier l’état d’un objet pendant qu’un autre thread l’utilise, mais également garantir que lorsqu’un thread modifie l’état d’un objet, les autres pourront *voir* les changements effectués. Or, sans synchronisation, ceci peut ne pas arriver. Vous pouvez garantir que les objets seront publiés correctement soit en utilisant une synchronisation explicite, soit en tirant parti de la synchronisation intégrée aux classes de la bibliothèque.

3.1 Visibilité

La visibilité est un problème subtil parce que ce qui peut mal se passer n’est pas évident. Dans un environnement monothread, si l’on écrit une valeur dans une variable et qu’on lise ensuite cette variable sans qu’elle ait été modifiée entre-temps, on s’attend à retrouver la même valeur, ce qui semble naturel. Cela peut être difficile à accepter mais, quand les lectures et les écritures ont lieu dans des threads différents, *ce n'est pas le cas*. En général, il n’y a *aucune* garantie que le thread qui lit verra une valeur écrite

par un autre thread. Pour assurer la visibilité des écritures mémoire entre les threads, il faut utiliser la synchronisation.

La classe `NoVisibility` du Listing 3.1 illustre ce qui peut se passer lorsque des threads partagent des données sans synchronisation. Deux threads, le thread principal et le thread lecteur, accèdent aux variables partagées `ready` et `number`. Le thread principal lance le thread lecteur puis initialise `number` à 42 et `ready` à `true`. Le thread lecteur boucle jusqu'à voir `ready` à `true`, puis affiche `number`. Bien qu'il puisse sembler évident que `NoVisibility` affichera 42, il est possible qu'elle affiche zéro, voire qu'elle ne se termine jamais ! Comme cette classe n'utilise pas de synchronisation adéquate, il n'y a aucune garantie que les valeurs de `ready` et `number` écrites par le thread principal soient visibles par le thread lecteur.

Listing 3.1 : Partage de données sans synchronisation. Ne le faites pas.

```
public class NoVisibility {  
    private static boolean ready;  
    private static int number;  
  
    private static class ReaderThread extends Thread {  
        public void run() {  
            while (!ready)  
                Thread.yield();  
            System.out.println(number);  
        }  
    }  
  
    public static void main(String[] args) {  
        new ReaderThread().start();  
        number = 42;  
        ready = true;  
    }  
}
```



`NoVisibility` pourrait boucler sans fin parce que le thread lecteur pourrait ne jamais voir la nouvelle valeur de `ready`. Ce qui est plus étrange encore est que `NoVisibility` pourrait afficher zéro car le thread lecteur pourrait voir la nouvelle valeur de `ready` avant l'écriture dans `number`, en vertu d'un phénomène appelé *réarrangement*. Il n'y a aucune garantie que les opérations dans un thread seront exécutées dans l'ordre du programme du moment que ce réarrangement n'est pas détecté dans *ce thread – même s'il est constaté par les autres*¹. Lorsque le thread principal écrit d'abord dans `number`, puis dans `ready` sans synchronisation, le thread lecteur pourrait constater que les choses se sont produites dans l'ordre inverse – ou pas du tout.

1. On pourrait penser qu'il s'agit d'une mauvaise conception, mais cela permet à la JVM de profiter au maximum des performances des systèmes multiprocesseurs modernes. En l'absence de synchronisation, par exemple, le modèle mémoire de Java permet au compilateur de réordonner les opérations et de placer les valeurs en cache dans des registres ; il permet également aux CPU de réordonner les opérations et de placer les valeurs dans des caches spécifiques du processeur. Le Chapitre 16 donnera plus de détails.

En l'absence de synchronisation, le compilateur, le processeur et l'environnement d'exécution peuvent s'amuser bizarrement avec l'ordre dans lequel les opérations semblent s'exécuter. Essayer de trouver l'ordre selon lequel les opérations sur la mémoire "doivent" se passer dans les programmes multithreads mal synchronisés sera presque certainement voué à l'échec.

NoVisibility est presque aussi simple qu'un programme concurrent peut l'être – deux threads et deux variables partagées –, mais on peut très bien ne pas savoir ce qu'il fait ni même s'il se terminera. Réfléchir sur des programmes concurrents mal synchronisés est vraiment très difficile.

Tout cela peut et devrait vous effrayer. Heureusement, il existe un moyen simple d'éviter tous ces problèmes complexes : *utilisez toujours une synchronisation correcte à chaque fois que des données sont partagées par des threads.*

3.1.1 Données obsolètes

NoVisibility a montré l'une des raisons pour lesquelles les programmes insuffisamment synchronisés peuvent produire des résultats surprenants : les *données obsolètes*. Lorsque le thread lecteur examine `ready`, il peut voir une valeur non à jour. À moins d'utiliser la synchronisation *à chaque fois que l'on accède à une variable*, on peut lire une valeur obsolète de cette variable. Pire encore, cette obsolescence ne fonctionne pas en tout ou rien : un thread peut voir une valeur à jour pour une variable et une valeur obsolète pour une autre, qui a pourtant été modifiée avant.

Quand une nourriture n'est plus fraîche, elle reste généralement comestible – elle est juste moins bonne – mais les données obsolètes peuvent être plus dangereuses. Alors qu'un compteur de visites obsolète pour une application web peut ne pas être trop méchant¹, les valeurs obsolètes peuvent provoquer de sévères problèmes de thread safety ou de vivacité. Dans la classe NoVisibility, elles provoquaient l'affichage d'une valeur erronée ou empêchaient le programme de se terminer, mais la situation peut se compliquer encore plus avec des valeurs obsolètes de références d'objets, comme les liens dans une liste chaînée. *Les données obsolètes peuvent provoquer de graves erreurs, difficiles à comprendre, comme des exceptions inattendues, des structures de données corrompues, des calculs imprécis et des boucles infinies.*

La classe `MutableInteger` du Listing 3.2 n'est pas thread-safe car on accède au champ `value` par `get()` et `set()` sans synchronisation. Parmi les autres risques qu'elle encourt,

1. Lire des données sans synchronisation est analogue à l'utilisation du niveau d'isolation `READ_UNCOMMITTED` dans une base de données lorsque l'on veut sacrifier la précision aux performances. Cependant, dans le cas de lectures non synchronisées, on sacrifie un plus grand degré de précision puisque la valeur visible d'une variable partagée peut être arbitrairement obsolète.

elle peut être victime des données obsolètes : si un thread appelle `set()`, les autres threads appelant `get()` pourront, ou non, constater cette modification.

Nous pouvons rendre `MutableInteger` thread-safe en synchronisant les méthodes de lecture et d'écriture comme dans la classe `SynchronizedInteger` du Listing 3.3. Ne synchroniser que la méthode d'écriture ne serait pas suffisant : les threads appelant `get()` pourraient toujours voir des valeurs obsolètes.

Listing 3.2 : Conteneur non thread-safe pour un entier modifiable.

```
@NotThreadSafe
public class MutableInteger {
    private int value;

    public int get() { return value; }
    public void set(int value) { this.value = value; }
}
```



Listing 3.3 : Conteneur thread-safe pour un entier modifiable.

```
@ThreadSafe
public class SynchronizedInteger {
    @GuardedBy("this") private int value;

    public synchronized int get() { return value; }
    public synchronized void set(int value) { this.value = value; }
}
```

3.1.2 Opérations 64 bits non atomiques

Un thread lisant une variable sans synchronisation peut obtenir une valeur obsolète mais il lira au moins une valeur qui a été placée là par un autre thread, plutôt qu'une valeur aléatoire. Cette garantie est appelée *safety magique* (*out-of-thin-air safety*).

La safety magique s'applique à toutes les variables, à une exception près : les variables numériques sur 64 bits (`double` et `long`), qui ne sont pas déclarées `volatile` (voir la section 3.1.4). En effet, le modèle mémoire de Java exige que les opérations de lecture et d'écriture soient atomiques, or les variables `long` et `double` sur 64 bits sont lues ou écrites à l'aide de deux opérations sur 32 bits. Si les lectures et les écritures ont lieu dans des threads différents, il est donc possible de lire un `long` non volatile et d'obtenir les 32 bits de poids fort d'une valeur et les 32 bits de poids faible d'une autre¹. Par conséquent, même si vous ne vous souciez pas des valeurs obsolètes, il n'est pas prudent d'utiliser des variables modifiables de type `long` ou `double` dans les programmes multithreads, sauf si elle sont déclarées `volatile` ou protégées par un verrou.

1. Lorsque la spécification de la machine virtuelle Java a été écrite, la plupart des processeurs du marché ne disposaient pas d'opérations arithmétiques atomiques efficaces sur 64 bits.

3.1.3 Verrous et visibilité

Comme le montre la Figure 3.1, le verrouillage interne peut servir à garantir qu'un thread verra les effets d'un autre thread de façon prévisible. Lorsque le thread *A* exécute un bloc `synchronized` et qu'ensuite le thread *B* entre dans un bloc `synchronized` protégé par le même verrou, les valeurs des variables visibles par *A* avant de libérer le verrou seront visibles par *B* lorsqu'il aura pris le verrou. En d'autres termes, tout ce qu'a fait *A* avant ou dans un bloc `synchronized` est visible par *B* lorsqu'il exécute un bloc `synchronized` protégé par le même verrou. *Sans synchronisation, cette garantie n'existe pas.*

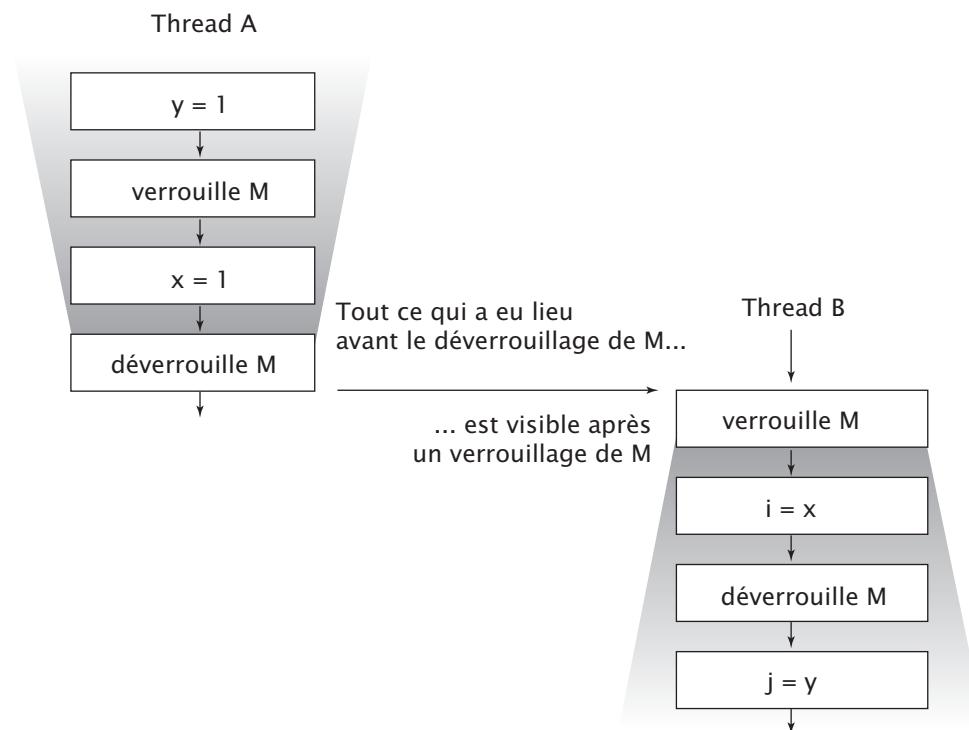


Figure 3.1

Visibilité garantie pour la synchronisation.

Nous pouvons maintenant donner l'autre raison de la règle qui exige que tous les threads se synchronisent sur le *même* verrou lorsqu'ils accèdent à une variable partagée modifiable – garantir que les valeurs écrites par un thread soient visibles par les autres. Sinon un thread lisant une variable sans détenir le verrou approprié risque de voir une valeur obsolète.

Le verrouillage ne sert pas qu'à l'exclusion mutuelle ; il est également utilisé pour la visibilité de la mémoire. Pour garantir que tous les threads voient les valeurs les plus récentes des variables modifiables partagées, les threads de lecture et d'écriture doivent se synchroniser sur le même verrou.

3.1.4 Variables volatiles

Le langage Java fournit également une alternative, une forme plus faible de synchronisation : les *variables volatiles*. Celles-ci permettent de s'assurer que les modifications apportées à une variable seront systématiquement répercutées à tous les autres threads. Lorsqu'un champ est déclaré `volatile`, le compilateur et l'environnement d'exécution sont prévenus que cette variable est partagée et que les opérations sur celle-ci ne doivent pas être réarrangées avec d'autres opérations sur la mémoire. Les variables volatiles ne sont pas placées dans des registres ou autres caches qui les masquerait aux autres processeurs ; la lecture d'une variable volatile renvoie donc toujours la dernière valeur qui y a été écrite par un thread quelconque.

Un bon moyen de se représenter les variables volatiles consiste à imaginer qu'elles se comportent à peu près comme la classe `SynchronizedInteger` du Listing 3.3, en remplaçant les opérations de lecture et d'écriture sur la variable par des appels à `get()` et `set()`¹. Cependant, l'accès à une variable volatile n'utilise aucun verrouillage et ne peut donc pas bloquer le thread qui s'exécute : c'est un mécanisme de synchronisation plus léger que `synchronized`².

La visibilité des variables volatiles va au-delà de la valeur de la variable. Lorsqu'un thread *A* écrit dans une variable volatile et qu'un thread *B* la lit ensuite, les valeurs de toutes les variables qui étaient visibles pour *A* avant d'écrire dans la variable volatile deviennent visibles pour *B* après sa lecture de cette variable. Du point de vue de la visibilité mémoire, l'écriture dans une variable volatile revient donc à sortir d'un bloc `synchronized` et sa lecture revient à entrer dans un bloc `synchronized`. Cependant, nous déconseillons de trop se fier aux variables volatiles pour la visibilité d'un état quelconque ; le code qui utilise des variables volatiles dans ce but est plus fragile et plus difficile à comprendre qu'un code qui utilise des verrous.

N'utilisez les variables volatiles que pour simplifier l'implémentation et la vérification de votre politique de synchronisation ; évitez-les si la vérification du code exige des calculs subtils sur la visibilité. Une bonne utilisation de ces variables consiste à assurer la visibilité de leur propre état, celui auquel se réfèrent les objets, ou pour indiquer qu'un événement important (comme une initialisation ou une fermeture) s'est passé.

1. Cette analogie n'est pas exacte car la visibilité mémoire de `SynchronizedInteger` est, en réalité, un peu plus forte que celle des variables volatiles, comme on l'explique au Chapitre 16.

2. Sur la plupart des processeurs actuels, les lectures volatiles sont à peine un peu plus coûteuses que les lectures non volatiles.

Le Listing 3.4 illustre une utilisation typique des variables volatiles : le test d'un indicateur pour savoir quand sortir d'une boucle. Ici, notre thread à l'apparence humaine essaie de dormir après avoir compté des moutons. Pour que cet exemple fonctionne, l'indicateur `asleep` doit être déclaré `volatile` ; sinon le thread pourrait ne pas remarquer qu'il a été positionné par un autre thread¹. Nous aurions pu utiliser à la place un verrou pour garantir la visibilité des modifications apportées à `asleep`, mais le code serait alors devenu plus lourd.

Listing 3.4 : Compter les moutons.

```
volatile boolean asleep;
...
    while (!asleep)
        countSomeSheep();
```

Les variables volatiles sont pratiques, mais elles ont leurs limites. Le plus souvent, on les utilise pour terminer ou interrompre une exécution, ou pour les indicateurs d'état comme `asleep` dans le Listing 3.4. Elles peuvent être utilisées dans d'autres types d'opérations sur l'état mais, en ce cas, il faut être plus prudent. La sémantique de `volatile`, par exemple, n'est pas suffisamment forte pour rendre une incrémentation (comme `count++`) atomique, sauf si vous pouvez garantir que la variable n'est modifiée que par un seul thread (comme on l'explique au Chapitre 15, les variables atomiques permettent d'effectuer des opérations *lire-modifier-écrire* et peuvent souvent être utilisées comme des "meilleures variables volatiles").

Alors que les verrous peuvent garantir à la fois la visibilité et l'atomicité, les variables volatiles ne peuvent assurer que la visibilité.

Vous ne pouvez utiliser des variables volatiles que lorsque tous les critères suivants sont vérifiés :

- Les écritures dans la variable ne dépendent pas de sa valeur actuelle ou vous pouvez garantir que sa valeur ne sera toujours modifiée que par un seul thread.
- La variable ne participe pas aux invariants avec d'autres variables d'état.

1. Pour les applications serveur, assurez-vous de toujours utiliser l'option `-server` de la JVM lorsque vous lappelez pendant les phases de développement et de tests. En mode serveur, la JVM effectue plus d'optimisations qu'en mode client : elle extrait les invariants de boucle, par exemple. Un code qui peut sembler fonctionner dans l'environnement de développement (client JVM) peut donc ne plus fonctionner dans l'environnement de production (serveur JVM). Dans le Listing 3.4, si nous avions par exemple "oublié" de déclarer la variable `asleep` comme `volatile`, le serveur JVM aurait pu sortir le test de la boucle (qui serait donc devenue une boucle sans fin), alors que le client JVM ne l'aurait pas fait. Or une boucle infinie apparaissant au cours du développement est bien moins coûteuse que si elle n'apparaît qu'à la mise en production.

- Lorsque l'on accède à la variable, le verrouillage n'est pas nécessaire pour d'autres raisons.

3.2 Publication et fuite

Publier un objet signifie le rendre disponible au code qui est en dehors de sa portée courante ; stocker une référence vers lui où un autre code pourra le trouver, par exemple, le renvoyer à partir d'une méthode non privée ou le passer à une méthode d'une autre classe. Dans de nombreuses situations, on veut garantir que les objets et leurs détails internes ne seront *pas* publiés. Dans d'autres, on veut publier un objet pour une utilisation générale, mais le faire de façon thread-safe peut nécessiter une synchronisation. Publier les variables de l'état interne d'un objet peut compromettre l'encapsulation et compliquer la préservation des invariants ; publier des objets avant qu'ils ne soient totalement construits peut compromettre la thread safety. On dit qu'un objet qui est publié alors qu'il n'aurait pas dû l'être s'est *échappé*. La section 3.5 présente les idiomes d'une publication correcte mais, pour l'instant, nous allons étudier comment un objet peut s'échapper.

La forme la plus évidente de la publication consiste à stocker une référence dans un champ statique public, où n'importe quelle classe et n'importe quel thread peut la voir, comme dans le Listing 3.5. Dans cet exemple, la méthode `initialize()` instancie un nouvel objet `HashSet` et le publie en stockant sa référence dans `knownSecrets`.

Listing 3.5 : Publication d'un objet.

```
public static Set<Secret> knownSecrets;

public void initialize() {
    knownSecrets = new HashSet<Secret>();
}
```

Publier un objet peut indirectement en publier d'autres. Si vous ajoutez un `Secret` à l'ensemble `knownSecrets` qui est publié, vous avez également publié ce `Secret` puisque n'importe quel code peut parcourir cet ensemble et obtenir une référence sur le nouveau `Secret`. De même, renvoyer une référence à partir d'une méthode non privée publie également l'objet renvoyé. Dans le Listing 3.6, la classe `UnsafeStates` publie le tableau des abréviations des états américains, qui est pourtant privé.

Listing 3.6 : L'état modifiable interne à la classe peut s'échapper. Ne le faites pas.

```
class UnsafeStates {
    private String[] states = new String[] {
        "AK", "AL" ...
    };
    public String[] getStates() { return states; }
}
```



Publier `states` de cette façon pose un problème puisque n'importe quel appelant peut modifier son contenu. Ici, le tableau `states` s'est échappé de sa portée initiale car ce qui était censé être un état privé a été rendu public.

Publier un objet publie également tous les objets qu'il référence dans ses champs non privés. Plus généralement, tout objet *accessible* à partir d'un objet publié en suivant une chaîne de champs références et d'appels de méthodes non privés est également publié.

Du point de vue d'une classe *C*, une méthode *étrangère* est une méthode dont le comportement n'est pas totalement spécifié par *C*. Cela comprend donc les méthodes des autres classes ainsi que les méthodes redéfinissables (donc ni `private` ni `final`) de *C* elle-même. Passer un objet à une méthode étrangère doit également être considéré comme une publication de cet objet. En effet, comme on ne peut pas savoir quel code sera réellement appelé, on ne sait pas si la méthode étrangère ne publiera pas cet objet ou si elle gardera une référence vers celui-ci, qui pourrait être utilisée plus tard à partir d'un autre thread.

Qu'un autre thread utilise une référence publiée n'est pas vraiment le problème : ce qui importe est qu'il existe un risque de mauvaise utilisation¹. Une fois qu'un objet s'est échappé, vous devez supposer qu'une autre classe ou un autre thread peut, volontairement ou non, le détourner. C'est un argument irréfutable en faveur de l'encapsulation puisque celle-ci permet de tester si les programmes sont corrects et complique la violation accidentelle des contraintes de conception.

Un dernier mécanisme par lequel un objet ou son état interne peuvent être publiés consiste à publier une instance d'une classe interne, comme dans la classe `ThisEscape` du Listing 3.7. Quand `ThisEscape` publie le `EventListener`, elle publie implicitement aussi l'instance `ThisEscape` car les instances des classes internes contiennent une référence cachée vers l'instance qui les englobe.

Listing 3.7 : Permet implicitement à la référence `this` de s'échapper. Ne le faites pas.

```
public class ThisEscape {  
    public ThisEscape(EventSource source) {  
        source.registerListener (  
            new EventListener() {  
                public void onEvent(Event e) {  
                    doSomething(e);  
                }  
            } );  
    }  
}
```



1. Si quelqu'un vole votre mot de passe et le poste sur le forum `alt.free-passwords`, cette information s'est échappée : que quelqu'un l'ait ou non utilisée pour vous causer du tort, votre compte a quand même été compromis. Publier une référence expose au même risque.

3.2.1 Pratiques de construction sûres

ThisEscape illustre un cas particulier important de la fuite d'un objet – lorsque la référence `this` s'échappe lors de sa construction. Quand l'instance `EventListener` interne est publiée, l'instance `ThisEscape` englobante l'est aussi. Mais un objet n'est dans un état cohérent qu'après la fin de l'exécution de son constructeur : publier un objet à partir de son constructeur peut donc publier un objet qui n'a pas été totalement construit et *ceci est vrai même si la publication s'effectue dans la dernière instruction du constructeur*. Si la référence `this` s'échappe au cours de la construction, l'objet est considéré comme *mal construit*¹.

Faites en sorte que la référence `this` ne s'échappe pas au cours de la construction.

Une erreur fréquente pouvant conduire à la fuite de `this` au cours de la construction consiste à lancer un thread à partir du constructeur. Lorsqu'un objet crée un thread dans son constructeur, il partage presque toujours sa référence `this` avec le nouveau thread, soit explicitement (en le passant au constructeur du thread) soit implicitement (parce que le `Thread` ou le `Runnable` est une classe interne de l'objet). Le nouveau thread pourrait alors voir l'objet englobant avant que ce dernier ne soit totalement construit. Cela ne pose aucun problème de créer un thread dans un constructeur, mais il est préférable de ne pas lancer immédiatement ce thread : fournissez plutôt une méthode `start()` ou `initialize()` qui permettra de le lancer (le Chapitre 7 détaillera les problèmes liés au cycle de vie des services). L'appel d'une méthode d'instance redéfinissable (ni `private` ni `final`) à partir du constructeur peut également donner à `this` l'occasion de s'échapper.

Si vous voulez enregistrer un récepteur d'événement ou lancer un thread à partir d'un constructeur, vous pouvez vous en sortir en utilisant un constructeur privé et une méthode fabrique publique, comme dans la classe `SafeListener` du Listing 3.8.

Listing 3.8 : Utilisation d'une méthode fabrique pour empêcher la référence `this` de s'échapper au cours de la construction de l'objet.

```
public class SafeListener {  
    private final EventListener listener;  
  
    private SafeListener() {  
        listener = new EventListener() {  
            public void onEvent(Event e) {  
                doSomething(e);  
            }  
        };  
    }  
}
```

1. Plus précisément, la référence `this` ne devrait pas s'échapper du thread avant la fin du constructeur. Elle peut être stockée quelque part par le constructeur tant qu'elle n'est pas *utilisée* par un autre thread jusqu'à la fin de la construction. La classe `SafeListener` du Listing 3.8 utilise cette technique.

```
public static SafeListener newInstance(EventSource source) {  
    SafeListener safe = new SafeListener();  
    source.registerListener (safe.listener);  
    return safe;  
}  
}
```

3.3 Confinement des objets

L'accès à des données partagées et modifiables nécessite d'utiliser la synchronisation. Un bon moyen d'éviter cette obligation consiste à ne *pas partager* : si on n'accède aux données que par un seul thread, il n'y a pas besoin de synchronisation. Cette technique, appelée *confinement*, est l'un des moyens les plus simples qui soient pour assurer la thread safety. Lorsqu'un objet est confiné dans un thread, son utilisation sera automatiquement thread-safe, même si l'objet confiné ne l'est pas lui-même [CPJ 2.3.2].

Swing utilise beaucoup le confinement. Ses composants visuels et les objets du modèle de données ne sont pas thread-safe, mais on obtient cette propriété en les confinant au thread des événements de Swing. Pour utiliser Swing correctement, le code qui s'exécute dans les threads autres que celui des événements ne devrait pas accéder à ces objets (pour faciliter cette pratique, Swing fournit le mécanisme `invokeLater()`, qui permet de planifier l'exécution d'un objet `Runnable` dans le thread des événements). De nombreuses erreurs de concurrence dans les applications Swing proviennent d'une mauvaise utilisation de ces objets confinés à partir d'un autre thread.

Une autre application classique du confinement est l'utilisation des pools d'objets `Connection` de JDBC (*Java Database Connectivity*). La spécification de JDBC n'exige pas que les objets `Connection` soient thread-safe¹. Dans les applications serveur classiques, un thread prend une connexion dans le pool, l'utilise pour traiter une seule requête et la retourne au pool. La plupart des requêtes, comme les requêtes de servlets ou les appels EJB (*Enterprise JavaBeans*), étant appelées de façon synchrone par un unique thread et le pool ne délivrant pas la même connexion à un autre thread tant qu'elle ne s'est pas terminée, ce patron de gestion des connexions confine implicitement l'objet `Connection` à ce thread pour la durée de la requête.

Tout comme le langage ne possède pas de mécanisme pour imposer qu'une variable soit protégée par un verrou, il n'a aucun moyen de confiner un objet dans un thread. Le confinement est un élément de conception du programme qui doit être imposé par son implémentation. Le langage et les bibliothèques de base fournissent des mécanismes permettant de faciliter la gestion de ce confinement – les variables locales et la classe

1. Les implémentations du pool de connexion fournies par les serveurs d'applications sont thread-safe ; comme on accède nécessairement aux pools de connexion à partir de plusieurs threads, une implémentation non thread-safe n'aurait aucun intérêt.

ThreadLocal – mais il est quand même de la responsabilité du programmeur de s’assurer que les objets confinés à un thread ne s’en échappent pas.

3.3.1 Confinement *ad hoc*

Le *confinement ad hoc* intervient lorsque la responsabilité de gérer le confinement incombe entièrement à l’implémentation. Il peut donc être fragile car aucune des fonctionnalités du langage, tels les modificateurs de visibilité des membres ou les variables locales, ne facilite le confinement de l’objet au thread concerné. En fait, les références à des objets confinés, comme les composants visuels ou les modèles de données dans les applications graphiques, sont souvent contenues dans des champs publics.

La décision d’utiliser le confinement découle souvent de la décision d’implémenter un sous-système particulier (une interface graphique, par exemple) comme un sous-système monothread. Ces sous-systèmes ont parfois l’avantage de la simplicité, ce qui compense la fragilité du confinement *ad hoc*¹.

Un cas spécial de confinement concerne les variables volatiles. Vous pouvez sans problème exécuter des opérations *lire-modifier-écrire* sur des variables volatiles partagées du moment que vous garantissez que la variable volatile n’est écrite qu’à partir d’un seul thread. En ce cas, vous confinez la *modification* dans un seul thread pour éviter les situations de compétition, et les garanties de visibilité des variables volatiles assurent que les autres threads verront la dernière valeur de la variable.

À cause de sa fragilité, le confinement *ad hoc* doit être utilisé avec parcimonie ; si possible, préférez-lui plutôt une des formes plus fortes du confinement (confinement dans la pile ou ThreadLocal).

3.3.2 Confinement dans la pile

Le *confinement dans la pile* est un cas spécial de confinement dans lequel on ne peut accéder à un objet qu’au travers de variables locales. Tout comme l’encapsulation aide à préserver les invariants, les variables locales permettent de simplifier le confinement des objets à un thread. En effet, les variables locales sont intrinsèquement confinées au thread en cours d’exécution ; elles n’existent que sur la pile de ce thread, qui n’est pas accessible aux autres. Le confinement dans la pile (également appelé utilisation *interne au thread ou locale au thread* et qu’il ne faut pas confondre avec la classe ThreadLocal de la bibliothèque standard) est plus simple à maintenir et moins fragile que le confinement *ad hoc*.

Dans le cas de variables locales de types primitifs, comme numPairs dans la méthode loadTheArk() du Listing 3.9, il est impossible de violer le confinement sur la pile.

1. Une autre raison de rendre un sous-système monothread est d’éviter les interblocages. C’est d’ailleurs l’une des principales raisons pour lesquelles les frameworks graphiques sont monothreads. Les sous-systèmes monothreads seront présentés au Chapitre 9.

Comme il n'existe aucun moyen d'obtenir une référence vers une variable de type primitif, la sémantique du langage garantit que les variables locales de ce type seront toujours confinées dans la pile.

Listing 3.9 : Confinement des variables locales, de types primitifs ou de types références.

```
public int loadTheArk(Collection<Animal> candidates) {  
    SortedSet<Animal> animals;  
    int numPairs = 0;  
    Animal candidate = null;  
  
    // animals est confiné dans la méthode, ne le laissez pas s'échapper!  
    animals = new TreeSet<Animal>(new SpeciesGenderComparator());  
    animals.addAll(candidates);  
    for (Animal a : animals) {  
        if (candidate == null || !candidate.isPotentialMate (a))  
            candidate = a;  
        else {  
            ark.load(new AnimalPair(candidate, a));  
            ++numPairs;  
            candidate = null;  
        }  
    }  
    return numPairs;  
}
```

Maintenir un confinement dans la pile pour des références d'objets nécessite un petit peu plus de travail de la part du programmeur, afin de s'assurer que le référent ne s'échappe pas. Dans `loadTheArk()`, on instancie un objet `TreeSet` et on stocke sa référence dans `animals`. À ce stade, il n'existe qu'une seule référence vers l'ensemble, contenue dans une variable locale et donc confinée au thread en cours d'exécution. Cependant, si l'on publiait une référence à cet ensemble (ou à l'un de ses composants internes), le confinement serait violé et les animaux pourraient s'échapper.

L'utilisation d'un objet non thread-safe dans un contexte "interne à un thread" est quand même thread-safe. Cependant, vous devez être prudent : l'exigence que l'objet soit confiné au thread ou le fait de savoir que l'objet confiné n'est pas thread-safe n'existe souvent que dans la tête du développeur. Si l'hypothèse que l'utilisation est interne au thread n'est pas clairement documentée, les développeurs ultérieurs du code pourraient, par erreur, laisser l'objet s'échapper.

3.3.3 ThreadLocal

Un moyen plus formel de maintenir le confinement consiste à utiliser la classe `ThreadLocal`, qui permet d'associer à un objet une valeur propre à un thread. `ThreadLocal` fournit des méthodes accesseurs `get()` et `set()` qui maintiennent une copie distincte de la valeur pour chaque thread qui l'utilise : un appel à `get()` renvoie donc la valeur la plus récente passée à `set()` à partir du thread en cours d'exécution.

Les variables locales au thread servent souvent à empêcher le partage dans les conceptions qui reposent sur des singletons modifiables ou sur des variables globales. Une application

monothread, par exemple, pourrait gérer une connexion globale vers une base de données, initialisée au démarrage, afin d'éviter de passer un objet `Connection` à chaque appel de méthode. Les connexions JDBC pouvant ne pas être thread-safe, une application multithread qui utilise une connexion globale sans coordination supplémentaire n'est pas thread-safe non plus. En utilisant un objet `ThreadLocal` pour stocker cette connexion, comme dans la classe `ConnectionHolder` du Listing 3.10, chaque thread disposera de sa propre connexion.

Listing 3.10 : Utilisation de ThreadLocal pour garantir le confinement au thread.

```
private static ThreadLocal<Connection> connectionHolder  
    = new ThreadLocal<Connection>() {  
    public Connection initialValue() {  
        return DriverManager.getConnection (DB_URL);  
    }  
};  
  
public static Connection getConnection() {  
    return connectionHolder.get();  
}
```

Cette technique peut également être utilisée lorsqu'une opération fréquente a besoin d'un objet temporaire comme un tampon et que l'on souhaite éviter de réallouer cet objet temporaire à chaque appel. Avant Java 5.0, par exemple, `Integer.toString()` utilisait un `ThreadLocal` pour stocker le tampon de 12 octets utilisé pour formater son résultat au lieu d'utiliser un tampon statique partagé (qui aurait nécessité un verrou) ou d'allouer un nouveau tampon à chaque appel¹.

Lorsqu'un thread appelle `ThreadLocal.get()` pour la première fois, `initialValue` est lue pour fournir la valeur initiale pour ce thread. Conceptuellement, vous pouvez considérer qu'un `ThreadLocal<T>` contient un `Map<Thread, T>` qui stocke les valeurs spécifiques au thread, bien qu'il ne soit pas implémenté de cette façon. Les valeurs spécifiques au thread sont stockées dans l'objet `Thread` lui-même ; lorsqu'il se termine, ces valeurs peuvent être supprimées par le ramasse-miettes.

Si vous portez une application monothread vers un environnement multithread, vous pouvez préserver la thread safety en convertissant les variables globales en objets `Thread Local` si la sémantique de ces variables le permet ; un cache au niveau de l'application ne serait pas aussi utile s'il était transformé en plusieurs caches locaux aux threads.

Les implémentations des frameworks applicatifs font largement appel à `ThreadLocal`. Les conteneurs J2EE, par exemple, associent un contexte de transaction à un thread

1. Cette technique n'apporte probablement pas un gain en terme de performances, sauf si l'opération est exécutée très souvent ou que l'allocation est exagérément coûteuse. En Java 5.0, elle a été remplacée par l'approche plus évidente qui consiste à allouer un nouveau tampon à chaque appel, ce qui semble indiquer que, pour quelque chose d'aussi banal qu'un tampon temporaire, cela ne permettait pas d'améliorer les performances.

d'exécution pour la durée d'un appel EJB, ce qui peut aisément s'implémenter à l'aide d'un `ThreadLocal` contenant le contexte de la transaction : lorsque le code du framework a besoin de savoir quelle est la transaction qui s'exécute, il récupère le contexte à partir de ce `ThreadLocal`. C'est pratique puisque cela réduit le besoin de passer les informations sur le contexte d'exécution à chaque méthode, mais cela lie au framework tout code utilisant ce mécanisme.

Il est assez simple d'abuser de `ThreadLocal` en considérant sa propriété de confinement comme un laisser-passer pour utiliser des variables globales ou comme un moyen de créer des paramètres de méthodes "cachés". Comme les variables globales, les variables locales aux threads peuvent contrarier la réutilisabilité du code et introduire des liens cachés entre les classes ; pour toutes ces raisons, elles doivent être utilisées avec discernement.

3.4 Objets non modifiables

L'autre moyen d'éviter la synchronisation consiste à utiliser des objets *non modifiables* [EJ Item 13]. Quasiment tous les risques concernant l'atomicité et la visibilité que nous avons décrits, comme la récupération de valeurs obsolètes, la perte de mises à jour ou l'observation d'un objet dans un état incohérent, sont liés au fait que plusieurs threads tentent d'accéder simultanément au même état modifiable. Si l'état d'un objet ne peut pas être modifié, tous ces risques et ces complications disparaissent.

Un objet non modifiable est un objet dont l'état ne peut pas être modifié après sa construction. Par essence, les objets non modifiables sont thread-safe ; leurs invariants sont établis par le constructeur et, si leur état ne peut pas être modifié, ces invariants seront toujours vérifiés.

Les objets non modifiables sont toujours thread-safe.

Les objets non modifiables sont *simples*. Ils ne peuvent être que dans un seul état, qui est soigneusement contrôlé par le constructeur. L'une des parties les plus difficiles de la conception d'un programme consiste à prendre en compte tous les états possibles des objets complexes ; pour l'état des objets non modifiables, cette étape est triviale.

Les objets non modifiables sont également *plus sûrs*. Il est dangereux de passer un objet modifiable à un code non vérifié, ou de le publier à un endroit où un code suspect peut le trouver – ce code pourrait modifier son état ou, pire, garder une référence vers lui et modifier son état plus tard, à partir d'un autre thread. Les objets non modifiables, en revanche, ne peuvent pas être altérés de cette manière par du code malicieux ou bogué et peuvent donc être partagés en toute sécurité ou publiés librement, sans qu'il y ait besoin de créer des copies défensives [EJ Item 24].

Ni la spécification du langage Java ni le modèle mémoire de Java ne définissent formellement cette "immuabilité", mais elle n'est *pas* équivalente à simplement déclarer tous les champs d'un objet comme `final`. Un objet ayant cette caractéristique pourrait quand même être modifiable puisque les champs `final` peuvent contenir des références vers des objets modifiables.

Un objet est *non modifiable* si :

- son état ne peut pas être modifié après sa construction ;
- tous ses champs sont `final`¹ ;
- il est *correctement construit* (sa référence `this` ne s'échappe pas au cours de la construction).

En interne, les objets non modifiables peuvent quand même utiliser des objets modifiables pour gérer leur état, comme l'illustre la classe `ThreeStooges` du Listing 3.11. Bien que le `Set` qui stocke les noms soit modifiable, la conception de `ThreeStooges` rend impossible la modification de cet ensemble après la construction. La référence `stooges` étant `final`, on accède à tout l'état de l'objet *via* un champ `final`. La dernière exigence, une construction correcte, est aisément vérifiée puisque le constructeur ne fait rien qui pourrait faire que la référence `this` devienne accessible à un code autre que celui du constructeur et de celui qui l'appelle.

Listing 3.11 : Classe non modifiable construite à partir d'objets modifiables sous-jacents.

```
@Immutable
public final class ThreeStooges {
    private final Set<String> stooges = new HashSet<String>();

    public ThreeStooges() {
        stooges.add("Moe");
        stooges.add("Larry");
        stooges.add("Curly");
    }

    public boolean isStooge(String name) {
        return stooges.contains(name);
    }
}
```

1. Techniquement, tous les champs d'un objet non modifiable peuvent ne pas être `final` – `String` en est un exemple –, mais cela implique de prendre en compte les situations de compétition bénignes, ce qui nécessite une très bonne compréhension du modèle mémoire de Java. Pour les curieux, `String` effectue un calcul paresseux du code de hachage lors du premier appel de `hashCode()` et place le résultat en cache dans un champ non `final` ; cela ne fonctionne que parce que ce champ ne peut prendre qu'une seule valeur, qui sera la même à chaque calcul puisqu'elle est déterminée à partir d'un état qui ne peut pas être modifié. N'essayez pas de faire la même chose.

L'état d'un programme changeant constamment, on pourrait être tenté de croire que les objets non modifiables ont peu d'intérêt, mais ce n'est pas le cas. Il y a une différence entre un *objet* non modifiable et une *référence* non modifiable vers celui-ci. Lorsque l'état d'un programme est stocké dans des objets modifiables, ceux-ci peuvent quand même être "remplacés" par une nouvelle instance contenant le nouvel état ; la section suivante donne un exemple de cette technique¹.

3.4.1 Champs final

Le mot-clé `final`, une version limitée du mécanisme `const` de C++, permet de construire des objets non modifiables. Les champs `final` ne peuvent pas être modifiés (bien que les objets auxquels ils font référence puissent l'être), mais ils ont également une sémantique spéciale dans le modèle mémoire de Java. C'est l'utilisation des champs `final` qui rend possible la garantie d'une *initialisation sûre* (voir la section 3.5.2) qui permet d'accéder aux objets non modifiables et de les partager sans synchronisation.

Même si un objet est modifiable, rendre certains de ses champs `final` peut quand même simplifier la compréhension de son état puisqu'en limitant la possibilité de modification d'un objet on restreint également l'ensemble de ses états possibles. Un objet qui est "presque entièrement modifiable" mais qui possède une ou deux variables d'états modifiables est plus simple qu'un objet qui a de nombreuses variables modifiables. Déclarer des champs `final` permet également d'indiquer aux développeurs qui maintiennent le code que ces champs ne sont pas censés être modifiés.

Tout comme il est conseillé de rendre tous les champs privés, sauf s'ils ont besoin d'une visibilité supérieure [EJ Item 12], il est conseillé de rendre tous les champs `final`, sauf s'il doivent pouvoir être modifiés.

3.4.2 Exemple : utilisation de `volatile` pour publier des objets non modifiables

Dans la classe `UnsafeCachingFactorizer` du Listing 2.5, nous avons essayé d'utiliser deux `AtomicReferences` pour stocker le dernier nombre factorisé et les derniers facteurs trouvés, mais ce n'était pas thread-safe puisque nous ne pouvions pas obtenir ou modifier ces deux valeurs de façon atomique. Pour la même raison, l'utilisation de variables volatiles pour ces valeurs ne serait pas thread-safe non plus. Cependant, les objets non modifiables peuvent parfois fournir une forme faible de l'atomicité. La servlet de

1. De nombreux développeurs craignent que cette approche pose des problèmes de performance, mais ces craintes sont généralement non justifiées. L'allocation est moins coûteuse qu'on ne le croit et les objets non modifiables offrent des avantages supplémentaires en termes de performances, comme la réduction du besoin de verrous ou de copies défensives, ainsi qu'un impact réduit sur le ramasse-miettes générationnel.

factorisation effectue deux opérations qui doivent être atomiques : la mise à jour du résultat en cache et, éventuellement, la récupération des facteurs en cache si le nombre en cache correspond au nombre soumis à la requête. À chaque fois qu'un groupe de données liées les unes aux autres doit être manipulé de façon atomique, pensez à créer une classe non modifiable pour les regrouper, comme `OneValueCache`¹ du Listing 3.12.

Listing 3.12 : Conteneur non modifiable pour mettre en cache un nombre et ses facteurs.

```
@Immutable
class OneValueCache {
    private final BigInteger lastNumber;
    private final BigInteger[] lastFactors;

    public OneValueCache(BigInteger i, BigInteger[] factors) {
        lastNumber = i;
        lastFactors = Arrays.copyOf(factors, factors.length);
    }

    public BigInteger[] getFactors(BigInteger i) {
        if (lastNumber == null || !lastNumber.equals(i))
            return null;
        else
            return Arrays.copyOf(lastFactors, lastFactors.length);
    }
}
```

Les situations de compétition lors de l'accès ou de la modification de plusieurs variables liées les unes aux autres peuvent être éliminées en utilisant un objet non modifiable pour contenir toutes ces variables. Si cet objet était modifiable, il faudrait utiliser des verrous pour garantir l'atomicité ; avec un objet non modifiable, un thread qui prend une référence sur cet objet n'a pas besoin de se soucier si un autre thread modifiera son état. Si les variables doivent être modifiées, on crée un nouvel objet mais les éventuels threads qui manipulent l'ancien objet le verront quand même dans un état cohérent.

La classe `VolatileCachedFactorizer` du Listing 3.13 utilise un objet `OneValueCache` pour stocker le nombre et les facteurs en cache. Lorsqu'un thread initialise le champ `cache` volatile pour qu'il contienne une référence à un nouvel objet `OneValueCache`, les nouvelles valeurs en cache deviennent immédiatement visibles aux autres threads.

Listing 3.13 : Mise en cache du dernier résultat à l'aide d'une référence volatile vers un objet conteneur non modifiable.

```
@ThreadSafe
public class VolatileCachedFactorizer implements Servlet {
    private volatile OneValueCache cache = new OneValueCache (null, null);

    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = cache.getFactors(i);
```

1. `OneValueCache` ne serait pas non modifiable sans les appels à `copyOf()` dans le constructeur et la méthode d'accès. `Arrays.copyOf()` n'existe que depuis Java 6 mais `clone()` devrait également fonctionner.

```

        if (factors == null) {
            factors = factor(i);
            cache = new OneValueCache (i, factors);
        }
        encodeIntoResponse (resp, factors);
    }
}

```

Les opérations liées au cache ne peuvent pas interférer les unes avec les autres car `OneValueCache` n'est pas modifiable et parce qu'on accède au champ `cache` qu'une seule fois dans chaque partie du code. Cette combinaison d'un objet conteneur non modifiable pour plusieurs variables d'état liées par un invariant et d'une référence volatile pour assurer sa visibilité rend `VolatileCachedFactorizer` thread-safe, bien qu'elle n'utilise pas de verrouillage explicite.

3.5 Publication sûre

Pour l'instant, nous avons fait en sorte de garantir qu'un objet ne sera pas publié lorsqu'il est censé être confiné à un thread ou interne à un autre objet. Cependant, on *veut* parfois partager des objets entre les threads et, dans ce cas, il faut le faire en toute sécurité. Malheureusement, se contenter de stocker une référence dans un champ public, comme dans le Listing 3.14, ne suffit *pas* à publier cet objet de façon satisfaisante.

Listing 3.14 : Publication d'un objet sans synchronisation appropriée. Ne le faites pas.

```

// Publication non sûre
public Holder holder;

public void initialize() {
    holder = new Holder(42);
}

```



Vous pourriez être surpris des conséquences de cet exemple qui semble pourtant anodin. À cause des problèmes de visibilité, l'objet `Holder` pourrait apparaître à un autre thread sous un état incohérent, bien que ses invariants aient été correctement établis par son constructeur ! Cette publication incorrecte pourrait permettre à un autre thread d'observer un objet partiellement construit.

3.5.1 Publication incorrecte : quand les bons objets deviennent mauvais

Vous ne pouvez pas vous fier à des objets qui n'ont été que partiellement construits. Un thread pourrait voir l'objet dans un état incohérent et voir plus tard son état changer brusquement, bien qu'il n'ait pas été modifié depuis sa publication. En fait, si le `Holder` du Listing 3.15 est publié selon la publication incorrecte du Listing 3.14 et qu'un thread autre que celui qui publie appelle `assertSanity()`, celle-ci pourrait lever l'exception `AssertionError`¹ !

1. Le problème, ici, est non pas la classe `Holder` elle-même, mais le fait que l'objet `Holder` ne soit pas correctement publié. Cependant, on pourrait immuniser `Holder` contre une publication incorrecte en déclarant le champ `n` `final`, ce qui rendrait `Holder` non modifiable (voir la section 3.5.2).

Listing 3.15 : Classe risquant un problème si elle n'est pas correctement publiée.

```
public class Holder {  
    private int n;  
  
    public Holder(int n) { this.n = n; }  
  
    public void assertSanity() {  
        if (n != n)  
            throw new AssertionError("This statement is false.");  
    }  
}
```



Comme on n'a pas utilisé de synchronisation pour rendre `Holder` visible aux autres threads, `Holder` n'a pas été *correctement publiée*. Il y a deux choses qui peuvent mal se passer avec les objets mal publiés. Les autres threads pourraient voir une valeur obsolète dans le champ `holder` et donc une référence `null` ou une autre valeur ancienne, bien qu'une valeur ait été placée dans `holder` ; et, ce qui est bien pire, les autres threads pourraient voir une valeur à jour pour la référence à `holder`, mais des valeurs obsolètes pour *l'état* de l'objet `Holder`¹. Pour rendre les choses encore moins prévisibles, un thread peut voir une valeur obsolète la première fois qu'il lit un champ, puis une valeur plus à jour lors de la lecture suivante, ce qui explique pourquoi `assertSanity()` peut lever `AssertionError`.

Au risque de nous répéter, des phénomènes très étranges peuvent survenir lorsque des données sont partagées entre des threads sans synchronisation appropriée.

3.5.2 Objets non modifiables et initialisation sûre

Les objets non modifiables ayant tant d'importance, le modèle mémoire de Java garantit une *initialisation sûre* pour les partager. Comme nous l'avons vu, le fait qu'une référence d'objet devienne visible pour un autre thread ne signifie pas nécessairement que l'état de cet objet sera visible au thread client. Pour garantir une vue cohérente de l'état de l'objet, il faut utiliser la synchronisation.

On peut en revanche accéder en toute sécurité à un objet non modifiable *même si l'on n'utilise pas de synchronisation pour publier sa référence*. Pour que cette garantie d'initialisation sûre puisse s'appliquer, il faut que toutes les exigences portant sur les objets non modifiables soient vérifiées : état non modifiable, tous les champs déclarés comme `final` et construction correcte (si la classe `Holder` du Listing avait été non modifiable, `assertSanity()` n'aurait pas pu lancer `AssertionError`, même si l'objet `Holder` était publié incorrectement).

1. Bien qu'il puisse sembler que les valeurs des champs initialisées dans un constructeur soient les premières écrites dans ces champs et qu'il n'y ait donc pas de valeurs "anciennes" qui puissent être considérées comme des valeurs obsolètes, le constructeur de `Object` initialise d'abord tous les champs avec des valeurs par défaut avant que les constructeurs des sous-classes ne s'exécutent. Il est donc possible de voir la valeur par défaut d'un champ comme valeur obsolète.

Les objets non modifiables peuvent être utilisés en toute sécurité par n'importe quel thread sans synchronisation supplémentaire, même si l'on n'a pas utilisé de synchronisation pour les publier.

Cette garantie s'étend aux valeurs de tous les champs `final` des objets correctement construits ; on peut accéder à ces champs en toute sécurité sans synchronisation supplémentaire. Cependant, si ces champs font référence à des objets modifiables, une synchronisation sera quand même nécessaire pour accéder à l'état des objets référencés.

3.5.3 Idiomes de publication correcte

Les objets modifiables doivent être *publiés correctement*, ce qui implique généralement une synchronisation entre le thread qui publie et le thread qui consomme. Pour l'instant, attachons-nous à vérifier que le thread consommateur puisse voir l'objet dans l'état où il est publié ; nous nous occuperons plus tard de la visibilité des modifications apportées après la publication.

Pour publier un objet correctement, il faut rendre visibles simultanément aux autres threads à la fois la référence à cet objet et son état. Un objet correctement construit peut être publié de façon sûre en respectant l'une des conditions suivantes :

- initialiser une référence d'objet à l'aide d'un initialisateur statique ;
- stocker une référence à cet objet dans un champ volatile ou de type Atomic Reference ;
- stocker une référence à cet objet dans un champ `final` d'un objet correctement construit ;
- stocker une référence à cet objet dans un champ protégé par un verrou.

Avec la synchronisation interne des collections thread-safe, comme `Vector` ou `synchronizedList`, le placement d'un objet dans ces collections vérifie la dernière de ces conditions. Si le thread *A* place l'objet *X* dans une collection thread-safe et que le thread *B* le récupère ensuite, il est garanti que *B* verra l'état de *X* tel que *A* l'a laissé, même si le code qui manipule *X* n'utilise pas de synchronisation *explicite*. Les collections thread-safe offrent les garanties de publication correcte suivantes, même si la documentation Javadoc est peu claire sur ce sujet :

- Le placement d'une clé ou d'une valeur dans un objet `Hashtable`, `synchronizedMap` ou `ConcurrentMap` est publié correctement pour tout thread qui la récupère dans la Map (que ce soit directement ou *via* un itérateur).
- Le placement d'un élément dans un objet `Vector`, `CopyOnWriteArrayList`, `CopyOnWriteArraySet`, `synchronizedList` ou `synchronizedSet` le publie correctement pour tout thread qui le récupère dans la collection.

- Le placement d'un élément dans un objet `BlockingQueue` ou `ConcurrentLinkedQueue` le publie correctement pour tout thread qui le récupère de la file.

D'autres mécanismes de la bibliothèque des classes (comme `Future` et `Exchanger`) constituent également une publication correcte ; nous les identifierons comme tels lorsque nous les présenterons.

L'utilisation d'un initialisateur statique est souvent le moyen le plus simple et le plus sûr de publier des objets pouvant être construits de façon statique :

```
public static Holder holder = new Holder(42);
```

Les initialisateurs statiques sont exécutés par la JVM au moment où la classe est initialisée ; à cause de la synchronisation interne de la JVM, ce mécanisme garantit une publication correcte de tous les objets initialisés de cette manière [JLS 12.4.2].

3.5.4 Objets non modifiables dans les faits

Une publication correcte suffit pour que d'autres threads accèdent en toute sécurité aux objets qui ne seront pas modifiés sans synchronisation supplémentaire après leur publication. Les mécanismes de publication sûre garantissent tous que l'état publié d'un objet sera visible à tous les threads qui y accèdent dès que sa référence est visible ; si cet état n'est pas amené à changer, cela suffit à assurer que tout accès à cet objet est thread-safe.

Les objets qui, techniquement, sont modifiables mais dont l'état ne sera pas modifié après publication sont appelés *objets non modifiables dans les faits*. Ils n'ont pas besoin de respecter la définition stricte des objets non modifiables de la section 3.4 ; il suffit qu'ils soient traités par le programme comme s'ils n'étaient pas modifiables après leur publication. L'utilisation de ce type d'objet permet de simplifier le développement et d'améliorer les performances car cela réduit les besoins de synchronisation.

Les *objets non modifiables dans les faits* qui ont été correctement publiés peuvent être utilisés en toute sécurité par n'importe quel thread sans synchronisation supplémentaire.

Les objets `Date`, par exemple, sont modifiables¹ mais, si vous les utilisez comme s'ils ne l'étaient pas, vous pouvez vous passer du verrouillage qui serait sinon nécessaire pour partager une date entre plusieurs threads. Supposons que vous vouliez utiliser un `Map` pour stocker la dernière date de connexion de chaque utilisateur :

```
public Map<String, Date> lastLogin =
    Collections.synchronizedMap (new HashMap<String, Date>());
```

1. Ce qui est sûrement une erreur de conception.

Si les valeurs Date ne sont pas modifiées après avoir été placées dans le Map, la synchronisation de synchronizedMap suffit pour les publier correctement et aucune synchronisation supplémentaire n'est nécessaire lorsqu'on y accède.

3.5.5 Objets modifiables

Si un objet est susceptible d'être modifié après sa construction, une publication sûre ne peut garantir la visibilité de son état tel qu'il a été publié. Pour garantir la visibilité des modifications ultérieures, il faut utiliser la synchronisation non seulement pour publier un objet modifiable, mais également à chaque fois qu'on y accède. Pour partager des objets modifiables en toute sécurité, ceux-ci doivent avoir été publiés de façon sûre *et* être thread-safe ou protégés par un verrou.

Les exigences de publication d'un objet dépendent du fait qu'il soit modifiable ou non :

- Les objets *non modifiables* peuvent être publiés par n'importe quel mécanisme.
- Les objets *non modifiables dans les faits* doivent être publiés de façon sûre.
- Les objets *modifiables* doivent être publiés de façon sûre et être thread-safe ou protégés par un verrou.

3.5.6 Partage d'objets de façon sûre

À chaque fois que l'on obtient une référence à un objet, il faut savoir ce que l'on a le droit d'en faire. Faut-il poser un verrou avant de l'utiliser ? Est-on autorisé à modifier son état ou peut-on seulement le lire ? De nombreuses erreurs de concurrence proviennent d'une mauvaise compréhension de ces "règles de bonne conduite" avec un objet partagé. Lorsque l'on publie un objet, il faut également indiquer comment on peut y accéder.

Les politiques les plus utiles pour l'utilisation et le partage des objets dans un programme concurrent sont :

- **Confinement au thread.** Un objet confiné à un thread n'appartient et n'est confiné qu'à un seul thread ; il peut être modifié par le thread qui le détient.
- **Partage en lecture seule.** Plusieurs threads peuvent accéder simultanément à un objet partagé en lecture seule sans synchronisation supplémentaire, mais cet objet ne peut être modifié par aucun thread. Ces objets comprennent les objets non modifiables et non modifiables dans les faits.
- **Partage thread-safe.** Un objet thread-safe effectuant une synchronisation interne, plusieurs threads peuvent y accéder via son interface publique sans synchronisation supplémentaire.
- **Protection par verrou.** On ne peut accéder à un objet protégé par un verrou qu'en prenant le contrôle d'un verrou précis. Ces objets comprennent ceux qui sont encapsulés dans d'autres objets thread-safe et les objets publiés protégés par un verrou.

Composition d'objets

Pour l'instant, nous n'avons traité que des aspects de bas niveau de la sécurité par rapport aux threads et de la synchronisation. Cependant, nous ne voulons pas devoir analyser chaque accès mémoire pour garantir que notre programme est thread-safe ; nous voulons pouvoir combiner des composants thread-safe pour créer des composants plus gros ou des programmes thread-safe. Ce chapitre présente donc des patrons de structuration de classes facilitant la création de classes thread-safe qui pourront être maintenues sans risquer de saboter les garanties qu'elles offrent vis-à-vis des threads.

4.1 Conception d'une classe thread-safe

Bien qu'il soit possible d'écrire un programme thread-safe qui stocke tout son état dans des champs statiques publics, vérifier la thread safety d'un tel programme (ou le modifier pour qu'il reste thread-safe) est bien plus difficile qu'avec un programme qui utilise correctement l'encapsulation. Cette dernière permet en effet de déterminer qu'une classe est thread-safe sans avoir besoin d'examiner tout le programme.

Le processus de conception d'une classe thread-safe devrait contenir ces trois éléments de base :

- identification des variables qui forment l'état de l'objet ;
- identification des invariants qui imposent des contraintes aux variables de l'état ;
- mise en place d'une politique de gestion des accès concurrents à l'état de l'objet.

L'état d'un objet commence avec ses champs. S'ils sont tous de type primitif, les champs contiennent l'intégralité de l'état. La classe Counter du Listing 4.1 n'ayant qu'un seul champ, son état est donc entièrement défini par la valeur de ce champ. L'état d'un objet possédant n champs de types primitifs est simplement un n -uplet des valeurs de ces

champs ; l'état d'un point en deux dimensions est formé des valeurs de ses coordonnées (x , y). Si certains champs d'un objet sont des références vers d'autres objets, l'état comprend également les champs des objets référencés. L'état d'un objet `LinkedList`, par exemple, inclut les états de tous les objets appartenant à la liste.

Listing 4.1 : Compteur mono-thread utilisant le patron moniteur de Java.

```
@ThreadSafe
public final class Counter {
    @GuardedBy("this") private long value = 0;

    public synchronized long getValue() {
        return value;
    }
    public synchronized long increment() {
        if (value == Long.MAX_VALUE)
            throw new IllegalStateException ("counter overflow");
        return ++value;
    }
}
```

C'est la *politique de synchronisation* qui définit comment un objet coordonne l'accès à son état sans violer ses invariants ou ses postconditions. Elle précise la combinaison d'immuabilité, de confinement et de verrouillage qu'il faut utiliser pour maintenir la thread safety et indique quelles variables sont protégées par quels verrous. Pour être sûr que la classe puisse être analysée et maintenue, vous devez documenter la politique de synchronisation utilisée.

4.1.1 Exigences de synchronisation

Créer une classe thread-safe implique de s'assurer que ses invariants sont vérifiés lors des accès concurrents, ce qui signifie qu'il faut tenir compte de son état. Les objets et les variables ont un *espace d'état*, l'ensemble des états possibles qu'ils peuvent prendre, et plus cet espace est réduit, plus il est facile de l'appréhender. En utilisant des champs `final` à chaque fois que cela est possible, on simplifie l'analyse des états possibles d'un objet (dans le cas extrême, les objets non modifiables ne peuvent être que dans un seul état).

De nombreuses classes ont des invariants qui considèrent certains états comme valides ou invalides. Le champ `value` de `Counter` étant un `long`, par exemple, l'espace d'état pourrait aller de `Long.MIN_VALUE` à `Long.MAX_VALUE`, mais `Counter` ajoute une contrainte : `value` ne peut pas être négatif.

De même, les opérations peuvent avoir des postconditions qui considèrent certaines *transitions d'état* comme incorrectes. Si l'état courant d'un `Counter` vaut 17, par exemple, le seul état suivant correct est 18. Lorsque l'état suivant est calculé à partir de l'état courant, l'opération est nécessairement composée. Toutes les opérations n'imposent pas de contraintes sur les transitions d'états ; lorsque l'on met à jour une variable qui

contient la température courante, par exemple, son état précédent n'a pas d'influence sur le calcul.

Les contraintes placées sur les états ou les transitions d'états par les invariants et les postconditions créent des besoins supplémentaires de synchronisation ou d'encapsulation. Si certains états sont invalides, alors, les variables d'état sous-jacentes doivent être encapsulées ; sinon le code client pourrait placer l'objet dans un état invalide. Si une opération comprend des transitions d'état invalides, elle doit être atomique. En revanche, si la classe n'impose aucune de ces contraintes, vous pouvez assouplir les besoins d'encapsulation ou de sérialisation, afin de bénéficier de plus de flexibilité ou de meilleures performances.

Une classe peut également posséder des invariants imposant des contraintes sur plusieurs variables d'état. Une classe d'intervalle numérique, comme `NumberRange` dans le Listing 4.10, utilise généralement des variables d'état pour les limites inférieure et supérieure de l'intervalle, et ces variables doivent obéir à la contrainte précisant que la limite inférieure doit être inférieure ou égale à la limite supérieure. Des invariants multivariables comme celui-ci exigent l'atomicité : les variables liées entre elles doivent être lues ou modifiées en une seule opération atomique. Vous ne pouvez pas en modifier une, libérer et reprendre le verrou, puis modifier les autres car l'objet pourrait être dans un état invalide lorsque le verrou est libéré. Lorsque plusieurs variables participent à un invariant, le verrou qui les protège doit être maintenu pendant toute la durée des opérations qui accèdent à ces variables.

Vous ne pouvez pas garantir la thread safety sans comprendre les invariants et les post-conditions d'un objet. Les contraintes sur les valeurs ou les transitions d'état autorisées pour les variables d'état peuvent exiger l'atomicité et l'encapsulation.

4.1.2 Opérations dépendantes de l'état

Les invariants de classe et les postconditions des méthodes imposent des contraintes aux états et aux transitions d'état admis pour un objet. Certains objets possèdent également des méthodes imposant des préconditions à leur état. Vous ne pouvez pas, par exemple, supprimer un élément d'une file vide ; une file doit être dans l'état "non vide" avant de pouvoir y ôter un élément. Les opérations disposant de telles préconditions sont dites *dépendantes de l'état*[CPJ 3].

Dans un programme monothread, lorsqu'une précondition n'est pas vérifiée, l'opération n'a pas d'autre choix que d'échouer. Dans un programme concurrent, en revanche, la précondition peut devenir vraie plus tard, par suite de l'action d'un autre thread. Les programmes concurrents ajoutent donc la possibilité d'attendre qu'une précondition soit vérifiée avant d'effectuer l'opération.

Les mécanismes intégrés permettant d'attendre efficacement qu'une condition soit vérifiée – `wait()` et `notify()` – sont intimement liés au verrouillage interne et peuvent être difficiles à utiliser correctement. Pour créer des opérations qui attendent qu'une pré-condition devienne vraie avant de continuer, il est souvent plus simple d'utiliser des classes existantes de la bibliothèque standard, comme les files d'attente ou les sémaphores, afin d'obtenir le comportement souhaité. Les classes bloquantes de la bibliothèque, comme `BlockingQueue`, `Semaphore` et les autres *synchronisateurs*, sont présentées au Chapitre 5 ; la création de classes dépendantes de l'état utilisant les mécanismes de bas niveau fournis par la plate-forme et la bibliothèque de classes est présentée au Chapitre 14.

4.1.3 Appartenance de l'état

Dans la section 4.1, nous avons laissé entendre que l'état d'un objet pouvait être un sous-ensemble des champs apparaissant dans le graphe des objets ayant cet objet pour racine. Pourquoi un sous-ensemble ? Sous quelles conditions les champs accessibles à partir d'un objet donné ne font *pas* partie de l'état de celui-ci ? Lorsque l'on précise les variables qui forment l'état d'un objet, on souhaite ne prendre en compte que les données *appartenant* à cet objet. L'appartenance est non pas une notion explicite du langage, mais un élément de la conception des classes. Lorsque l'on alloue et remplit un `HashMap`, par exemple, on crée plusieurs objets : l'objet `HashMap`, un certain nombre d'objets `Map.Entry` utilisés par l'implémentation de `HashMap` et, éventuellement, d'autres objets internes. L'état logique d'un `HashMap` inclut l'état de tous ses `Map.Entry` et des objets internes, bien qu'ils soient implémentés comme des objets distincts.

Pour le meilleur ou pour le pire, le ramasse-miettes nous évite de devoir trop nous préoccuper de l'appartenance. En C++, lorsque l'on passe un objet à une méthode, il faut savoir si l'on transfère la propriété, si l'on met en place une location à court terme ou si l'on envisage une copropriété à long terme. En Java, bien que tous ces modèles d'appartenance soient possibles, le ramasse-miettes réduit le coût des nombreuses erreurs lors du partage des références, ce qui nous permet de rester un peu flou sur la question de l'appartenance.

Dans de nombreux cas, l'appartenance et l'encapsulation sont liées – l'objet encapsule l'état qu'il possède et l'état qu'il encapsule lui appartient. C'est le propriétaire d'une variable d'état donnée qui doit décider du protocole de verrouillage utilisé pour maintenir l'intégrité de cette variable. L'appartenance implique le contrôle mais, une fois que nous avons publié une référence vers un objet modifiable, nous n'avons plus le contrôle exclusif de cet objet ; au mieux pouvons-nous avoir une "propriété partagée". Une classe ne possède généralement pas les objets qui sont passés à ses méthodes ou à ses constructeurs, sauf si la méthode a été conçue pour transférer explicitement la propriété des objets qui lui sont passés (c'est le cas, par exemple, des méthodes fabriques de collections synchronisées).

Les classes `Collection` utilisent souvent une forme de "propriété divisée" dans laquelle la collection possède l'état de l'infrastructure de collection alors que le code client possède les objets stockés dans la collection. C'est le cas, par exemple, de la classe `ServletContext` du framework des servlets. Cette dernière fournit aux servlets un objet conteneur assimilé à un `Map`, dans lequel elles peuvent enregistrer et récupérer des objets de niveau application par leurs noms à l'aide des méthodes `setAttribute()` et `getAttribute()`. L'objet `ServletContext` implémenté par le conteneur de servlets doit être thread-safe car plusieurs threads y accéderont nécessairement. Les servlets n'ont pas besoin d'utiliser de synchronisation lorsqu'elles appellent `setAttribute()` et `getAttribute()`, mais elles peuvent devoir en avoir besoin lorsqu'elles *utilisent* les objets stockés dans le `ServletContext`. Ces objets appartiennent à l'application ; ils sont stockés par le conteneur de servlets pour le compte de l'application et, comme tous les objets partagés, ils doivent être partagés correctement ; pour empêcher les interférences dues aux accès concurrents de la part des différents threads, ils doivent être soit thread-safe, soit non modifiables dans les faits, soit protégés explicitement par un verrou¹.

4.2 Confinement des instances

Si un objet n'est pas thread-safe, plusieurs techniques permettent toutefois de l'utiliser en toute sécurité dans un programme multithread. Vous pouvez faire en sorte qu'on n'y accède qu'à partir d'un seul thread (confinement au thread) ou que tous ses accès soit protégés par un verrou.

L'encapsulation simplifie la création de classes thread-safe en favorisant le *confinement des instances*, souvent simplement désigné sous le terme de *confinement* [CPJ 2.3.3]. Lorsqu'un objet est encapsulé dans un autre objet, tout le code qui a accès à l'objet encapsulé est connu et peut donc être analysé plus facilement que si l'objet était accessible par tout le programme. En combinant ce confinement à une discipline de verrouillage adéquate, on peut garantir que des objets qui, sans cela, ne seraient pas thread-safe seront utilisés de façon thread-safe.

L'encapsulation de données dans un objet confine l'accès de ces données aux méthodes de l'objet, ce qui permet de garantir plus facilement qu'on accédera toujours aux données avec le verrou adéquat.

1. L'objet `HttpSession` qui effectue une fonction similaire dans le framework des servlets peut avoir des exigences plus strictes. Le conteneur de servlets pouvant accéder aux objets de `HttpSession` afin de les sérialiser pour la réplication ou la passivation, ceux-ci doivent être thread-safe puisque le conteneur y accédera en même temps que l'application web (nous avons écrit "peut avoir" car la réplication et la passivation ne font pas partie de la spécification des servlets, bien qu'il s'agisse d'une fonctionnalité courante des conteneurs de servlets).

Les objets confinés ne doivent pas s'échapper de leur portée. Un objet peut être confiné à une instance de classe (c'est le cas, par exemple, d'un membre de classe privé), à une portée lexicale (c'est le cas des variables locales) ou à un thread (c'est le cas d'un objet passé de méthode en méthode dans un thread, mais qui n'est pas censé être partagé entre des threads). Les objets ne s'échappent pas tout seuls, bien sûr : ils ont besoin de l'aide du développeur, qui les aide en les publiant en dehors de leur portée.

La classe `PersonSet` du Listing 4.2 illustre la façon dont le confinement et le verrouillage peuvent fonctionner de concert pour créer une classe thread-safe, même lorsque ses variables d'état ne le sont pas. L'état de `PersonSet` est en effet géré par un `HashSet` qui n'est pas thread-safe ; mais, comme `mySet` est privé et ne peut pas s'échapper, ce `HashSet` est confiné dans l'objet `PersonSet`. Le seul code qui peut accéder à `mySet` est celui des méthodes `addPerson()` et `containsPerson()`, or chacune d'elles prend le verrou sur l'objet `PersonSet`. Comme tout son état est protégé par son verrou interne, `PersonSet` est thread-safe.

Listing 4.2 : Utilisation du confinement pour assurer la thread safety.

```
@ThreadSafe
public class PersonSet {
    @GuardedBy("this")
    private final Set<Person> mySet = new HashSet<Person>();

    public synchronized void addPerson(Person p) {
        mySet.add(p);
    }

    public synchronized boolean containsPerson(Person p) {
        return mySet.contains(p);
    }
}
```

Cet exemple ne fait aucune supposition sur la thread safety de `Person` mais, si cette classe est modifiable, une synchronisation supplémentaire sera nécessaire pour accéder à une `Person` récupérée à partir d'un `PersonSet`. Le moyen le plus sûr serait de rendre `Person` thread-safe ; le moins sûr serait de protéger les objets `Person` par un verrou et de s'assurer que tous les clients suivent le protocole consistant à prendre le verrou adéquat avant d'accéder à une `Person`.

Le confinement d'instance est l'un des moyens les plus simples pour créer des classes thread-safe. Il permet également de choisir la stratégie de verrouillage ; `PersonSet` utilise son propre verrou interne pour protéger son état, mais tout verrou correctement utilisé ferait de même. Avec le confinement des instances, vous pouvez aussi protéger les différentes variables d'état par des verrous distincts (la classe `ServerStatus` [voir Listing 11.7] est un exemple de classe utilisant plusieurs objets verrous pour protéger son état).

Les bibliothèques de classes de la plate-forme contiennent plusieurs exemples de confinement ; certaines classes n'existent que pour rendre thread-safe des classes qui ne le

sont pas. Les classes collection de base, comme `ArrayList` et `HashMap`, ne sont pas thread-safe, mais la bibliothèque fournit des méthodes qui enveloppent les méthodes fabriques de ces objets (`Collections.synchronizedList` et d'autres) pour pouvoir les utiliser en toute sécurité dans des environnements multithreads. Ces fabriques utilisent le patron de conception *Décorateur* (voir Gamma et al., 1995) pour envelopper la collection dans un objet synchronisé ; cette enveloppe implémente chaque méthode de l'interface adéquate sous la forme d'une méthode `synchronized` qui fait suivre la requête à l'objet sous-jacent. L'objet enveloppe est thread-safe tant qu'il détient la seule référence accessible à la collection sous-jacente (cette collection est donc confinée dans cette enveloppe). La documentation Javadoc de ces méthodes avertit d'ailleurs que tous les accès à la collection sous-jacente doivent se faire *via* l'enveloppe.

Il est bien sûr toujours possible de violer le confinement en publant un objet censé être confiné ; s'il a été conçu pour être confiné dans une portée spécifique, laisser l'objet s'échapper de cette portée constitue un bogue. Les objets confinés peuvent également s'échapper en publant d'autres objets comme des itérateurs ou des instances de classes internes qui peuvent, indirectement, publier les objets confinés.

Le confinement facilite la création de classes thread-safe car une classe qui confine son état peut être analysée sans avoir besoin d'examiner tout le programme.

4.2.1 Le patron moniteur de Java

En suivant le principe du confinement d'instance jusqu'à sa conclusion logique, on arrive au *patron moniteur de Java*¹. Un objet respectant ce patron encapsule tout son état modifiable et le protège à l'aide de son verrou interne.

La classe `Counter` du Listing 4.1 est un exemple typique de ce patron. Elle encapsule une seule variable d'état, `value`, et tous les accès à cette variable passent par ses méthodes, qui sont toutes synchronisées.

Le patron moniteur de Java est utilisé par de nombreuses classes de la bibliothèque, comme `Vector` et `Hashtable`, mais on peut parfois avoir besoin d'une politique de synchronisation plus sophistiquée : le Chapitre 11 montrera comment améliorer la "scalabilité" des programmes à l'aide de stratégies de verrouillage plus fines. L'avantage de ce patron est sa simplicité.

Le patron moniteur de Java est simplement une convention ; n'importe quel objet verrou peut servir à protéger l'état d'un objet pourvu qu'il soit utilisé correctement. Le Listing 4.3 donne un exemple de classe qui utilise un verrou privé pour protéger son état.

1. Le patron moniteur de Java s'inspire des travaux de Hoare sur les moniteurs (Hoare, 74), bien qu'il existe des différences significatives entre ce patron et un vrai moniteur. Les instructions du pseudo-code pour entrer et sortir d'un bloc `synchronized` s'appellent d'ailleurs `monitorenter` et `monitorexit`, et les verrous internes de Java sont parfois appelés *verrous moniteurs* ou, simplement, *moniteurs*.

Listing 4.3 : Protection de l'état à l'aide d'un verrou privé.

```
public class PrivateLock {  
    private final Object myLock = new Object();  
    @GuardedBy("myLock") Widget widget;  
  
    void someMethod() {  
        synchronized(myLock) {  
            // Accès ou modification de l'état du widget  
        }  
    }  
}
```

Utiliser un objet verrou privé au lieu d'un verrou interne (ou tout autre verrou accessible publiquement) présente quelques avantages. Le fait que l'objet verrou soit privé l'encapsule de sorte que le code client ne peut pas le prendre, alors qu'un verrou public autorise le code client à participer à la politique de synchronisation – correctement ou non. Les clients qui prennent incorrectement le verrou d'un autre objet peuvent en effet poser des problèmes de vivacité, et vérifier qu'un verrou public est correctement utilisé nécessite d'examiner tout le programme au lieu de cantonner l'analyse à une seule classe.

4.2.2 Exemple : gestion d'une flotte de véhicules

La classe Counter du Listing 4.1 est un exemple d'utilisation du patron moniteur de Java, mais il est un peu trop simple. Nous allons donc construire un exemple un peu moins trivial : un "gestionnaire de véhicules" chargé de répartir les véhicules d'une flotte comme des taxis, des voitures de police ou des camions de livraison. Nous utiliserons d'abord le patron moniteur, puis nous verrons comment assouplir un peu l'encapsulation tout en préservant la thread safety.

Chaque véhicule est identifié par une chaîne de caractères et a une position représentée par les coordonnées (x, y). La classe VehicleTracker encapsule l'identité et les emplacements des véhicules connus, ce qui en fait un modèle de données bien adapté à une application graphique *modèle-vue-contrôleur* où elle peut être partagée par un thread vue et plusieurs threads modificateurs. Le thread vue récupère les noms et les emplacements des véhicules pour les afficher :

```
Map<String, Point> locations = vehicles.getLocations();  
for (String key : locations.keySet())  
    renderVehicle(key, locations.get(key));
```

De même, les threads modificateurs mettent à jour les emplacements des véhicules à partir des données reçues par des dispositifs GPS ou entrées manuellement par un opérateur *via* une interface graphique :

```
void vehicleMoved (VehicleMovedEvent evt) {  
    Point loc = evt.getNewLocation();  
    vehicles.setLocation(evt.getVehicleId(), loc.x, loc.y);  
}
```

Le thread vue et les threads modificateurs concourant pour accéder au modèle de données, ce dernier doit être thread-safe. Le Listing 4.4 présente une implémentation du gestionnaire de véhicules qui utilise le patron moniteur de Java. Pour représenter les emplacements des véhicules, ce gestionnaire se sert de la classe `MutablePoint` du Listing 4.5.

Bien que `MutablePoint` ne soit pas thread-safe, la classe du gestionnaire l'est. Ni la carte ni aucun des points modifiables qu'elle contient ne seront jamais publiés. Si l'on doit renvoyer des emplacements de véhicules à un client, les valeurs appropriées seront copiées soit à l'aide du constructeur de copie de la classe `MutablePoint`, soit en utilisant la méthode `deepCopy()`, qui crée une nouvelle carte dont les valeurs sont des copies des clés et des valeurs de l'ancienne¹.

Cette implémentation gère en partie la thread safety en copiant les données modifiables avant de les renvoyer au client. Cela ne pose généralement pas de problème en terme de performances, sauf si l'ensemble des véhicules est très important². Une autre conséquence de la copie des données à chaque appel de `getLocations()` est que le contenu de la collection renvoyée ne changera pas, même si les emplacements sous-jacents sont modifiés ; que ce soit souhaitable ou non dépend de vos besoins. Cela peut être intéressant si la cohérence interne de l'ensemble des emplacements est importante, auquel cas renvoyer un instantané cohérent est essentiel, mais cela peut également être un problème si les clients ont besoin d'informations à jour pour chaque véhicule et qu'ils doivent donc rafraîchir leurs copies plus souvent.

4.3 Délégation de la thread safety

Tous les objets, sauf les plus simples, sont des objets composés. Le patron moniteur de Java est utile lorsque l'on crée des classes en partant de zéro ou que l'on compose des classes à partir d'objets non thread-safe, mais qu'en est-il si les composants de notre classe sont déjà thread-safe ? Faut-il ajouter une couche supplémentaire de thread safety ? La réponse est... "ça dépend". Dans certains cas, un objet composé à partir d'autres objets thread-safe est lui-même thread-safe (voir Listings 4.7 et 4.9) ; dans d'autres, c'est simplement un bon point de départ (voir Listing 4.10).

1. `deepCopy()` ne peut pas se contenter d'envelopper la carte par un objet `unmodifiableMap` puisque cela ne protégerait pas la *collection* contre les modifications ; les clients pourraient modifier les objets modifiables contenus dans cette carte. Pour la même raison, le remplissage du `HashMap` dans `deepCopy()` via un constructeur de copie ne fonctionnerait pas car cela ne copierait que les références aux points, pas les objets points eux-mêmes.

2. `deepCopy()` étant appelée à partir d'une méthode synchronisée, le verrou interne du gestionnaire est détenu pendant toute la durée de l'opération de copie, qui pourrait être assez longue. La réactivité de l'interface utilisateur risque donc d'être dégradée lorsque le nombre de véhicules gérés est très grand.

Dans la classe `CountingFactorizer` du Listing 2.4, nous avions ajouté un champ `AtomicLong` à un objet sans état et l'objet composé obtenu était quand même thread-safe. L'état de `CountingFactorizer` étant l'état contenu dans le champ `AtomicLong`, qui est thread-safe, et `CountingFactorizer` n'imposant aucune contrainte de validité sur le compteur, il est aisément de constater que la classe est elle-même thread-safe. Nous pourrions dire que `CountingFactorizer` délègue la responsabilité de sa thread safety à `AtomicLong` : `CountingFactorizer` est thread-safe parce que `AtomicLong` l'est¹.

Listing 4.4 : Implémentation du gestionnaire de véhicule reposant sur un moniteur.

```
@ThreadSafe
public class MonitorVehicleTracker {
    @GuardedBy("this")
    private final Map<String, MutablePoint> locations;

    public MonitorVehicleTracker(Map<String, MutablePoint> locations) {
        this.locations = deepCopy(locations);
    }

    public synchronized Map<String, MutablePoint> getLocations() {
        return deepCopy(locations);
    }

    public synchronized MutablePoint getLocation(String id) {
        MutablePoint loc = locations.get(id);
        return loc == null ? null : new MutablePoint(loc);
    }

    public synchronized void setLocation(String id, int x, int y) {
        MutablePoint loc = locations.get(id);
        if (loc == null)
            throw new IllegalArgumentException ("No such ID: " + id);
        loc.x = x;
        loc.y = y;
    }

    private static Map<String, MutablePoint> deepCopy(Map<String,
                                                       MutablePoint> m) {
        Map<String, MutablePoint> result = new HashMap<String,
                                                       MutablePoint>();
        for (String id : m.keySet())
            result.put(id, new MutablePoint(m.get(id)));
        return Collections.unmodifiableMap (result);
    }
}

public class MutablePoint { /* Listing 4.5 */ }
```

1. Si `count` n'avait pas été `final`, l'analyse de la thread safety de `CountingFactorizer` aurait été plus compliquée. Si `CountingFactorizer` pouvait modifier `count` pour qu'il fasse référence à un autre `AtomicLong`, nous devrions vérifier que cette modification est visible par tous les threads qui pourraient accéder au compteur et nous assurer qu'il n'y ait pas de situation de compétition concernant la valeur de la référence `count`. C'est une autre bonne raison d'utiliser des champs `final` à chaque fois que cela est possible.

Listing 4.5 : Classe Point modifiable ressemblant à java.awt.Point.

```
@NotThreadSafe
public class MutablePoint {
    public int x, y;

    public MutablePoint() { x = 0; y = 0; }
    public MutablePoint(MutablePoint p) {
        this.x = p.x;
        this.y = p.y;
    }
}
```



4.3.1 Exemple : gestionnaire de véhicules utilisant la délégation

Construisons maintenant une version du gestionnaire de véhicules qui délègue sa thread safety à une classe thread-safe. Comme nous stockons les emplacements dans un objet Map, nous partons d'une implémentation thread-safe de Map, ConcurrentHashMap. À la place de MutablePoint, nous utilisons également une classe Point non modifiable (voir Listing 4.6) pour stocker chaque emplacement.

Listing 4.6 : Classe Point non modifiable utilisée par DelegatingVehicleTracker.

```
@Immutable
public class Point {
    public final int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Point est thread-safe parce qu'elle est non modifiable. Les valeurs non modifiables pouvant être librement partagées et publiées, nous n'avons plus besoin de copier les emplacements lorsqu'on les renvoie.

La classe DelegatingVehicleTracker du Listing 4.7 n'utilise pas de synchronisation explicite ; tous les accès à son état sont gérés par ConcurrentHashMap et toutes les clés et valeurs de la Map sont non modifiables.

Listing 4.7 : Délégation de la thread safety à un objet ConcurrentHashMap.

```
@ThreadSafe
public class DelegatingVehicleTracker {
    private final ConcurrentHashMap<String, Point> locations;
    private final Map<String, Point> unmodifiableMap;

    public DelegatingVehicleTracker (Map<String, Point> points) {
        locations = new ConcurrentHashMap< String, Point>(points);
        unmodifiableMap = Collections.unmodifiableMap(locations);
    }

    public Map<String, Point> getLocations() {
        return unmodifiableMap ;
    }
}
```

Listing 4.7 : Délégation de la thread safety à un objet ConcurrentHashMap. (suite)

```
public Point getLocation(String id) {
    return locations.get(id);
}

public void setLocation(String id, int x, int y) {
    if (locations.replace(id, new Point(x, y)) == null)
        throw new IllegalArgumentException("invalid vehicle name: " + id);
}
}
```

L'utilisation de la classe `MutablePoint` au lieu de `Point` aurait brisé l'encapsulation en autorisant `getLocations()` à publier une référence vers un état modifiable non thread-safe. Vous remarquerez que nous avons légèrement modifié le comportement de la classe gestionnaire des véhicules : alors que la version avec moniteur renvoyait un instantané des emplacements, la version avec délégation renvoie une vue non modifiable mais "vivante" de ces emplacements. Ceci signifie que, si un thread *A* appelle `getLocations()` et qu'un thread *B* modifie ensuite l'emplacement de l'un des points, ces changements seront répercutés dans l'objet `Map` renvoyé au thread *A*. Comme nous l'avons déjà fait remarquer, ceci peut être un avantage (données plus à jour) ou un handicap (vue éventuellement incohérente de la flotte) en fonction de vos besoins.

Si vous avez besoin d'une vue de la flotte qui ne change pas, `getLocations()` pourrait renvoyer une copie de surface de la carte des emplacements. Le contenu de l'objet `Map` étant non modifiable, seule sa structure a besoin d'être copiée. C'est ce que l'on fait dans le Listing 4.8 (où l'on renvoie un vrai `HashMap`, puisque `getLocations()` n'a pas pour contrat de renvoyer un `Map` thread-safe).

Listing 4.8 : Renvoi d'une copie statique de l'ensemble des emplacements au lieu d'une copie "vivante".

```
public Map<String, Point> getLocations() {
    return Collections.unmodifiableMap(
        new HashMap<String, Point>(locations));
}
```

4.3.2 Variables d'état indépendantes

Les exemples de délégation que nous avons donnés jusqu'à présent ne déléguent qu'à une seule variable d'état thread-safe, mais il est également possible de déléguer la thread safety à plusieurs variables d'état sous-jacentes du moment que ces dernières sont *indépendantes*, ce qui signifie que la classe composée n'impose aucun invariant impliquant les différentes variables d'état.

La classe `VisualComponent` du Listing 4.9 est un composant graphique qui permet aux clients d'enregistrer des écouteurs (*listeners*) pour les événements souris et clavier. Elle gère donc deux listes d'écouteurs, une pour chaque type d'événement, afin que les écouteurs appropriés puissent être appelés lorsqu'un événement survient. Cependant, il n'y a aucune relation entre l'ensemble des écouteurs de la souris et celui des écouteurs

du clavier ; ils sont tous les deux indépendants et `VisualComponent` peut donc déléguer ses obligations de thread safety aux deux listes thread-safe sous-jacentes.

Listing 4.9 : Délégation de la thread à plusieurs variables d'état sous-jacentes.

```
public class VisualComponent {
    private final List<KeyListener> keyListeners
        = new CopyOnWriteArrayList <KeyListener>();
    private final List<MouseListener> mouseListeners
        = new CopyOnWriteArrayList <MouseListener>();

    public void addKeyListener(KeyListener listener) {
        keyListeners.add(listener);
    }

    public void addMouseListener (MouseListener listener) {
        mouseListeners.add(listener);
    }

    public void removeKeyListener (KeyListener listener) {
        keyListeners.remove(listener);
    }

    public void removeMouseListener (MouseListener listener) {
        mouseListeners.remove(listener);
    }
}
```

`VisualComponent` utilise un objet `CopyOnWriteArrayList` pour stocker chaque liste d'écouteurs ; cette classe est une implémentation thread-safe de `List` particulièrement bien adaptée à ce type de gestion (voir la section 5.2.3). Chaque liste est thread-safe et, comme il n'y a pas de contrainte reliant l'état de l'une à l'état de l'autre, `Visual Component` peut déléguer ses responsabilités de thread safety aux objets `mouseListeners` et `keyListeners`.

4.3.3 Échecs de la délégation

La plupart des classes composées ne sont pas aussi simples que `VisualComponent` : leurs invariants lient entre elles les variables d'état de leurs composants. Pour gérer son état, la classe `NumberRange` du Listing 4.10 utilise deux `AtomicInteger` mais impose une contrainte supplémentaire : le premier nombre doit être inférieur ou égal au second.

Listing 4.10 : Classe pour des intervalles numériques, qui ne protège pas suffisamment ses invariants. Ne le faites pas.

```
public class NumberRange {
    // INVARIANT: lower <= upper
    private final AtomicInteger lower = new AtomicInteger(0);
    private final AtomicInteger upper = new AtomicInteger(0);

    public void setLower(int i) {
        // Attention : tester-puis-agir non sûr
        if (i > upper.get())
            throw new IllegalArgumentException("can't set lower to " +
                i + " > upper");
    }
}
```



Listing 4.10 : Classe pour des intervalles numériques, qui ne protège pas suffisamment ses invariants. Ne le faites pas. (suite)

```
    lower.set(i);
}

public void setUpper(int i) {
    // Attention : tester-puis-agir non sûr
    if (i < lower.get())
        throw new IllegalArgumentException("can't set upper to " +
                                           i + " < lower");
    upper.set(i);
}

public boolean isInRange(int i) {
    return (i >= lower.get() && i <= upper.get());
}
}
```

NumberRange n'est pas thread-safe car elle ne préserve pas l'invariant qui contraint les valeurs de lower et upper. Les méthodes `setLower()` et `setUpper()` tentent bien de le respecter, mais elles le font mal car ce sont toutes les deux des séquences *tester-puis-agir* qui n'utilisent pas un verrouillage suffisant pour être atomiques. Si le nombre contient (0, 10) et qu'un thread appelle `setLower(5)` pendant qu'un autre appelle `setUpper(4)`, un timing malheureux fera que les deux tests réussiront et que ces deux modifications seront donc autorisées. Le résultat sera alors un intervalle (5, 4), donc dans un état incorrect. Bien que les `AtomicInteger` sous-jacents soient thread-safe, la classe composée ne l'est donc pas. Les variables d'état `lower` et `upper` n'étant pas indépendantes, NumberRange ne peut pas se contenter de déléguer sa thread safety à ses variables d'état thread-safe.

NumberRange pourrait être rendue thread-safe en utilisant le *même* verrou pour protéger `lower` et `upper` et donc ses invariants. Elle doit également éviter de publier `lower` et `upper` pour empêcher le code client de pervertir les invariants.

Si une classe comprend des actions composées, comme c'est le cas de NumberRange, la délégation seule ne suffit pas pour assurer la thread safety. La classe doit fournir son propre verrouillage pour garantir que ces opérations sont atomiques, sauf si l'action composée peut entièrement être déléguée aux variables d'état sous-jacentes.

Si une classe contient plusieurs variables d'état thread-safe *indépendantes* et qu'elle n'ait aucune opération ayant des transitions d'état invalides, elle peut déléguer sa thread safety à ces variables d'état.

Le problème qui empêchait NumberRange d'être thread-safe bien que ses composants d'état fussent thread-safe ressemble beaucoup à l'une des règles sur les variables volatiles de la section 3.1.4 : une variable ne peut être déclarée volatile que si elle ne participe pas à des invariants impliquant d'autres variables d'état.

4.3.4 Publication des variables d'état sous-jacentes

Lorsque l'on délègue la thread safety aux variables sous-jacentes d'un objet, sous quelles conditions peut-on publier ces variables pour que les autres classes puissent également les modifier ? Là encore, la réponse dépend des invariants qu'impose la classe sur ces variables. Bien que le champ `value` sous-jacent de `Counter` puisse recevoir n'importe quelle valeur entière, `Counter` la contraint à ne prendre que des valeurs positives et l'opération d'incrémentation contraint l'ensemble des états suivants admis pour un état donné. Si le champ `value` était public, les clients pourraient le modifier et y placer une valeur invalide ; si ce champ était publié, il rendrait la classe incorrecte. En revanche, si une variable représente la température courante ou l'identifiant du dernier utilisateur à s'être connecté, le fait qu'une autre classe modifie sa valeur ne violera probablement pas les invariants et la publication de cette variable peut donc être acceptable (elle peut quand même être déconseillée, car la publication de variables modifiables contraint les futurs développements et les occasions de créer des sous-classes, mais cela ne rendrait pas nécessairement la classe non thread-safe).

Une variable d'état thread-safe ne participant à aucun invariant qui contraint sa valeur et dont aucune des opérations ne possède de transition d'état interdit peut être publiée sans problème.

On pourrait publier sans problème, par exemple, les champs `mouseListeners` ou `keyListeners` de `VisualComponent`. Cette classe n'imposant aucune contrainte sur les états admis pour ses listes d'écouteurs, ces champs pourraient être publics ou publiés sans compromettre la thread safety.

4.3.5 Exemple : gestionnaire de véhicules publant son état

Nous allons construire une version du gestionnaire de véhicules qui publie son état modifiable. Nous devons donc modifier à nouveau l'interface pour prendre en compte ce changement, cette fois-ci en utilisant des points modifiables mais thread-safe.

Listing 4.11 : Classe point modifiable et thread-safe.

```
@ThreadSafe
public class SafePoint {
    @GuardedBy("this") private int x, y;

    private SafePoint(int[] a) { this(a[0], a[1]); }

    public SafePoint(SafePoint p) { this(p.get()); }

    public SafePoint(int x, int y) {
        this.set(x, y)
    }
```

Listing 4.11 : Classe point modifiable et thread-safe. (suite)

```

    public synchronized int[] get() {
        return new int[] { x, y };
    }

    public synchronized void set(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

```

La classe `SafePoint` du Listing 4.11 fournit un accesseur de lecture qui récupère les deux valeurs `x` et `y` et les renvoie dans un tableau de deux éléments¹. Si nous avions fourni des accesseurs de lecture distincts pour `x` et `y`, les valeurs pourraient être modifiées entre le moment où l'on récupère une coordonnée et celui où l'on récupère l'autre : le code client verrait alors une valeur incohérente, c'est-à-dire un emplacement (`x, y`) où le véhicule n'aurait jamais été. Grâce à `SafePoint`, nous pouvons construire un gestionnaire de véhicules qui publie son état modifiable sans compromettre la sécurité par rapport aux threads, comme le montre la classe `PublishingVehicleTracker` du Listing 4.12.

Listing 4.12 : Gestionnaire de véhicule qui publie en toute sécurité son état interne.

```

@ThreadSafe
public class PublishingVehicleTracker {
    private final Map<String, SafePoint> locations;
    private final Map<String, SafePoint> unmodifiableMap;

    public PublishingVehicleTracker(Map<String, SafePoint> locations) {
        this.locations = new ConcurrentHashMap <String,
                                         SafePoint>(locations);
        this.unmodifiableMap = Collections.unmodifiableMap(this.locations);
    }

    public Map<String, SafePoint> getLocations() {
        return unmodifiableMap;
    }

    public SafePoint getLocation(String id) {
        return locations.get(id);
    }

    public void setLocation(String id, int x, int y) {
        if (!locations.containsKey(id))
            throw new IllegalArgumentException ("invalid vehicle name: " + id);
        locations.get(id).set(x, y);
    }
}

```

`PublishingVehicleTracker` tire sa thread safety de la délégation à un objet `ConcurrentHashMap` sous-jacent mais, cette fois, le contenu de la `Map` sont des points modifiables thread-safe et non plus des points non modifiables. La méthode `getLocation()` renvoie

1. Le constructeur est privé pour éviter la situation de compétition qui surviendrait si le constructeur de copie était implémenté sous la forme `this(p.x, p.y)` ; c'est un exemple de l'*idiome capture du constructeur privé* (Bloch et Gafter, 2005).

une copie non modifiable de la Map sous-jacente. Le code appelant ne peut ni ajouter ni supprimer de véhicules, mais il pourrait modifier l'emplacement d'un véhicule en modifiant les valeurs SafePoint contenues dans la Map. Ici aussi, la nature "vivante" de la Map peut être un avantage ou un inconvénient en fonction des besoins. PublishingVehicleTracker est thread-safe mais elle ne le serait pas si elle imposait des contraintes supplémentaires sur les valeurs valides des emplacements des véhicules. S'il faut pouvoir placer un "veto" sur les modifications apportées aux emplacements des véhicules ou effectuer certaines actions lorsqu'un emplacement est modifié, l'approche choisie par PublishingVehicleTracker ne convient pas.

4.4 Ajout de fonctionnalités à des classes thread-safe existantes

La bibliothèque de classes de Java contient de nombreuses classes "briques" utiles. Il est souvent préférable de réutiliser des classes existantes plutôt qu'en créer de nouvelles : la réutilisation permet de réduire le travail et les risques de développement (puisque les composants existants ont déjà été testés), ainsi que les coûts de maintenance. Parfois, une classe thread-safe disposant de toutes les opérations dont on a besoin existe déjà mais, souvent, le mieux que l'on puisse trouver est une classe fournissant *presque* toutes les opérations voulues : nous devons alors ajouter une nouvelle opération sans compromettre son comportement vis-à-vis des threads.

Supposons, par exemple, que nous ayons besoin d'une liste thread-safe disposant d'une opération *ajouter-si-absent* atomique. Les implémentations synchronisées de List font l'affaire puisqu'elles fournissent les méthodes contains() et add() à partir desquelles nous pouvons construire l'opération voulue.

Le concept *ajouter-si-absent* est assez évident : on teste si un élément se trouve dans la collection avant de l'ajouter et on ne l'ajoute pas s'il est déjà présent (vos voyants d'alarme de *tester-puis-agir* devraient déjà s'être allumés). Le fait que la classe doive être thread-safe ajoute une autre contrainte : les opérations comme *ajouter-si-absent* doivent être *atomiques*. Toute interprétation sensée suggère que, si vous prenez une List qui ne contient pas l'objet X et que vous lui ajoutiez deux fois X avec *ajouter-si-absent*, la collection ne contiendra finalement qu'une seule copie de X. Si *ajouter-si-absent* n'était pas atomique et avec un timing malheureux, deux threads pourraient constater que X n'était pas présent et l'ajouteraient tous les deux.

Le moyen le plus simple d'ajouter une nouvelle opération atomique consiste à modifier la classe initiale pour qu'elle dispose de cette opération, mais ce n'est pas toujours possible car vous pouvez ne pas avoir accès au code source ou ne pas avoir le droit de le modifier. Si vous pouvez modifier la classe originale, vous devez comprendre la politique de synchronisation de son implémentation pour rester cohérent avec sa conception initiale. Pour ajouter directement une nouvelle méthode à la classe, il faut aussi que tout

le code qui implémente la politique de synchronisation de cette classe soit contenu dans le même fichier source, afin de faciliter sa compréhension et sa maintenance.

Une autre approche consiste à étendre la classe en supposant qu'elle a été conçue pour pouvoir être étendue. La classe `BetterVector` du Listing 4.13, par exemple, étend `Vector` pour lui ajouter une méthode `putIfAbsent()`. Étendre `Vector` est assez simple, mais toutes les classes n'exposent pas assez leur état aux sous-classes pour qu'il en soit toujours ainsi.

L'extension d'une classe est plus fragile que l'ajout direct de code à cette classe car elle implique la distribution de la politique de synchronisation entre plusieurs fichiers sources distincts. Si la classe sous-jacente modifie ensuite sa politique de synchronisation en choisissant un verrou différent pour protéger ses variables d'état, la sous-classe tomberait en panne sans prévenir puisqu'elle n'utilisera plus le bon verrou pour contrôler les accès concurrents à l'état de sa classe de base (la politique de synchronisation de `Vector` étant fixée par sa spécification, `BetterVector` ne peut pas souffrir de ce problème).

Listing 4.13 : Extension de `Vector` pour disposer d'une méthode *ajouter-si-absent*.

```
@ThreadSafe
public class BetterVector<E> extends Vector<E> {
    public synchronized boolean putIfAbsent(E x) {
        boolean absent = !contains(x);
        if (absent)
            add(x);
        return absent;
    }
}
```

4.4.1 Verrouillage côté client

Pour un objet `ArrayList` enveloppé dans un objet `Collections.synchronizedList`, aucune des deux approches précédentes ne peut fonctionner puisque le code client ne connaît même pas la classe de l'objet `List` renvoyé par les fabriques de l'enveloppe synchronisée. Une troisième stratégie consiste à étendre la fonctionnalité de la classe sans l'étendre elle-même mais en plaçant le code d'extension dans une classe auxiliaire.

Le Listing 4.14 présente une tentative manquée de créer une classe auxiliaire dotée d'une opération *ajouter-si-absent* atomique portant sur une `List` thread-safe.

Listing 4.14 : Tentative non thread-safe d'implémenter *ajouter-si-absent*. Ne le faites pas.

```
@NotThreadSafe
public class ListHelper<E> {
    public List<E> list = Collections.synchronizedList(new ArrayList<E>());
    ...
    public synchronized boolean putIfAbsent(E x) {
        boolean absent = !list.contains(x);
        if (absent)
            list.add(x);
        return absent;
    }
}
```



Pourquoi cela ne fonctionne-t-il pas, bien que `putIfAbsent()` soit synchronisée ? Le problème est que cette méthode se synchronise sur le *mauvais* verrou. Quel que soit celui que `List` utilise pour protéger son état, il est certain que ce n'est pas le même que celui de `ListHelper`. Cette dernière ne fournit donc qu'une *illusion de synchronisation* : les différentes opérations sur les listes, bien qu'elles soient toutes synchronisées, utilisent des verrous différents, ce qui signifie que `putIfAbsent()` *n'est pas* atomique par rapport aux autres opérations sur la `List`. Il n'y a donc aucune garantie qu'un autre thread ne pourra modifier la liste pendant l'exécution de `putIfAbsent()`.

Pour que cette approche fonctionne, vous devez utiliser le même verrou que `List` en vous servant d'un *verrouillage côté client ou externe*. Grâce au verrouillage côté client, un code client qui utilise un objet X sera protégé avec le verrou que X utilise pour protéger son propre état. Pour mettre en place ce type de verrouillage, il faut connaître le verrou que X utilise.

La documentation de `Vector` et des classes enveloppes synchronisées indique indirectement que ces classes supportent le verrouillage côté client en utilisant le verrou interne du `Vector` ou de la collection enveloppe (pas celui de la collection enveloppée). Le Listing 4.15 présente une méthode `putIfAbsent()` qui utilise correctement le verrouillage côté client lorsqu'elle s'applique à une `List` thread-safe.

Listing 4.15 : Implémentation d'*ajouter-si-absent* avec un verrouillage côté client.

```
@ThreadSafe
public class ListHelper<E> {
    public List<E> list = Collections.synchronizedList(new ArrayList<E>());
    ...
    public boolean putIfAbsent(E x) {
        synchronized (list) {
            boolean absent = !list.contains(x);
            if (absent)
                list.add(x);
            return absent;
        }
    }
}
```

Si l'extension d'une classe en lui ajoutant une autre opération atomique est fragile parce qu'elle distribue le code de verrouillage entre plusieurs classes d'une hiérarchie d'objets, le verrouillage côté client l'est plus encore puisqu'il implique de placer le code de verrouillage d'une classe C dans des classes qui n'ont rien à voir avec C. Faites attention lorsque vous utilisez un verrouillage côté client sur des classes qui ne précisent pas leur stratégie de verrouillage. Le verrouillage côté client a beaucoup de points communs avec l'extension de classe : tous les deux lient le comportement de la classe dérivée à l'implémentation de la classe de base. Tout comme l'extension viole l'encapsulation de l'implémentation [EJ Item 14], le verrouillage côté client viole l'encapsulation de la politique de synchronisation.

4.4.2 Composition

Il existe une autre solution, moins fragile, pour ajouter une opération atomique à une classe existante : la *composition*. La classe `ImprovedList` du Listing 4.16 implémente les opérations de `List` en les déléguant à une instance sous-jacente de `List` et ajoute une opération `putIfAbsent()` (comme `Collections.synchronizedList` et les autres collections enveloppes, `ImprovedList` suppose que le client n'utilisera pas directement la liste sous-jacente après l'avoir passée au constructeur de l'enveloppe et qu'il n'y accédera que par l'objet `ImprovedList`).

Listing 4.16 : Implémentation d'*ajouter-si-absent* en utilisant la composition.

```
@ThreadSafe
public class ImprovedList<T> implements List<T> {
    private final List<T> list;

    public ImprovedList(List<T> list) { this.list = list; }

    public synchronized boolean putIfAbsent(T x) {
        boolean contains = list.contains(x);
        if (!contains)
            list.add(x);
        return !contains;
    }

    public synchronized void clear() { list.clear(); }
    // ... délégations similaires pour les autres méthodes de List
}
```

`ImprovedList` ajoute un niveau de verrouillage supplémentaire en utilisant son propre verrou interne. Peu importe que la `List` sous-jacente soit thread-safe puisque `ImprovedList` fournit son propre verrouillage afin d'assurer la thread safety, même si la `List` n'est pas thread-safe ou qu'elle modifie l'implémentation de son verrouillage. Bien que cette couche de synchronisation additionnelle puisse avoir un léger impact sur les performances¹, l'implémentation de `ImprovedList` est moins fragile que les tentatives de mimer la stratégie de verrouillage d'un autre objet. En réalité, nous avons utilisé le patron moniteur de Java pour encapsuler une `List` existante et cela garantit la thread safety tant que notre classe contient la seule référence existante à cette `List` sous-jacente.

4.5 Documentation des politiques de synchronisation

La documentation est l'un des outils les plus puissants qui soit (et malheureusement l'un des moins utilisés) pour gérer la thread safety. C'est dans la documentation que les utilisateurs recherchent si une classe est thread-safe et que les développeurs qui maintiennent le code essaient de comprendre la stratégie de l'implémentation pour éviter de la

1. Cet impact sera peu important car il n'y aura pas de concurrence sur la synchronisation de la `List` sous-jacente. Celle-ci sera donc rapide ; voir le Chapitre 11.

compromettre accidentellement. Malheureusement, la documentation ne contient souvent pas les informations dont on a besoin.

Documentez les garanties de thread safety d'une classe pour ses clients et documentez sa politique de synchronisation pour ses développeurs ultérieurs.

Chaque utilisation de `synchronized`, `volatile` ou d'une classe thread-safe est le reflet d'une *politique de synchronisation* qui définit une stratégie assurant l'intégrité des données vis-à-vis des accès concurrents. Cette politique est un élément de la conception d'un programme et devrait apparaître dans la documentation. Le meilleur moment pour documenter des décisions de conception est, évidemment, la phase de conception. Quelques semaines ou mois plus tard, les détails peuvent s'être estompés et c'est la raison pour laquelle il faut les écrire avant de les oublier.

Mettre au point une politique de synchronisation exige de prendre un certain nombre de décisions : quelles variables rendre volatiles, quelles variables protéger par des verrous, quel(s) verrou(s) protège(nt) quelles variables, quelles variables faut-il rendre non modifiables ou confiner dans un thread, quelles opérations doivent être atomiques, etc. ? Certaines de ces décisions sont strictement des détails d'implémentation et devraient être documentées pour aider les futurs développeurs, mais d'autres affectent le comportement du verrouillage observable d'une classe et devraient être documentées comme une partie de sa spécification.

Au minimum, documentez les garanties de thread safety offertes par la classe. Est-elle thread-safe ? Appelle-t-elle des fonctions de rappel pendant qu'elle détient un verrou ? Y a-t-il des verrous précis qui affectent son comportement ? N'obligez pas les clients à faire des suppositions risquées. Si vous ne voulez pas fournir d'information pour le verrouillage côté client, parfait, mais indiquez-le. Si vous souhaitez que les clients puissent créer de nouvelles opérations atomiques sur votre classe, comme nous l'avons fait dans la section 4.4, indiquez les verrous qu'ils doivent obtenir pour le faire en toute sécurité. Si vous utilisez des verrous pour protéger l'état, signalez-le dans la documentation pour prévenir les futurs développeurs, d'autant que c'est très facile : l'annotation `@GuardedBy` s'en chargera. Si vous utilisez des moyens plus subtils pour maintenir la thread safety, indiquez-les car cela peut ne pas apparaître évident aux développeurs chargés de maintenir le code.

La situation actuelle de la documentation de la sécurité vis-à-vis des threads, même pour les classes de la bibliothèque standard, n'est pas encourageante. Combien de fois avez-vous recherché une classe dans Javadoc en vous demandant finalement si elle était thread-safe¹ ? La plupart des classes ne donnent aucun indice. De nombreuses spécifications

1. Si vous ne vous l'êtes jamais demandé, nous admirons votre optimisme.

officielles des technologies Java, comme les servlets et JDBC, sous-documentent cruellement leurs promesses et leurs besoins de thread safety.

Bien que la prudence suggère de ne pas supposer des comportements qui ne fassent pas partie de la spécification, il faut bien que le travail soit fait et nous devons souvent choisir entre plusieurs mauvaises suppositions. Doit-on supposer qu'un objet est thread-safe parce qu'il nous semble qu'il devrait l'être ? Doit-on supposer que l'accès à un objet peut être rendu thread-safe en obtenant d'abord son verrou (cette technique risquée ne fonctionne que si nous contrôlons tout le code qui accède à cet objet ; sinon elle ne fournit qu'une illusion de thread safety) ? Aucun de ces choix n'est vraiment satisfaisant.

Pour accentuer le tout, notre intuition peut parfois nous tromper sur les classes qui sont "probablement thread-safe" et celles qui ne le sont pas. `java.text.SimpleDateFormat`, par exemple, n'est pas thread-safe, mais sa documentation Javadoc oubliait de le mentionner jusqu'au JDK 1.4. Que cette classe précise ne pas être thread-safe a pris par surprise de nombreux développeurs. Combien de programmes ont été créés par erreur une instance partagée d'un objet non thread-safe et l'ont utilisée à partir de plusieurs threads, sans savoir que cela pourrait donner des résultats incorrects en cas de lourde charge ?

Un problème comme celui de `SimpleDateFormat` pourrait être évité en ne supposant pas qu'une classe soit thread-safe si elle n'indique pas qu'elle l'est. En revanche, il est impossible de développer une application utilisant des servlets sans faire quelques suppositions assez raisonnables sur la thread safety des objets conteneurs comme `HttpSession`. N'obligez pas vos clients ou vos collègues à faire ce genre de supposition.

4.5.1 Interprétation des documentations vagues

Les spécifications de nombreuses technologies Java ne disent rien, ou très peu de choses, sur les garanties et les besoins de thread safety d'interfaces comme `ServletContext`, `HttpSession` ou `DataSource`¹. Ces interfaces étant implémentées par l'éditeur du conteneur ou de la base de données, il est souvent impossible de consulter le code source pour savoir ce qu'il fait. En outre, on ne souhaite pas se fier aux détails d'implémentation d'un pilote JDBC particulier : on veut respecter le standard pour que notre code fonctionne correctement avec n'importe quel pilote. Cependant, les mots "thread" et "concurrent" n'apparaissent jamais dans la spécification de JDBC et très peu dans celle des servlets. Comment faire dans une telle situation ?

Il faut faire des suppositions. Un moyen d'améliorer la qualité de ces déductions consiste à interpréter la spécification du point de vue de celui chargé de l'implémenter (un éditeur de conteneur ou de base de données, par exemple) et non du point de vue de celui qui se contentera de l'utiliser. Les servlets étant toujours appelées à partir d'un thread géré par un conteneur, il est raisonnable de penser que ce conteneur saura s'il y a

1. Il est d'ailleurs particulièrement frustrant que ces omissions perdurent malgré les nombreuses révisions majeures de ces spécifications.

plusieurs threads. Le conteneur de servlet mettant à disposition des objets qui offrent des services à plusieurs servlets, comme `HttpSession` ou `ServletContext`, il devrait s'attendre à ce qu'on accède à ces objets de façon concurrente puisqu'il a créé plusieurs threads qui appelleront des méthodes comme `Servlet.service()` et que celles-ci accèdent sûrement à l'objet `ServletContext`.

Comme il est impossible d'imaginer que ces objets puissent être utilisés dans un contexte monothread, on doit supposer qu'ils ont été conçus pour être thread-safe, bien que la spécification ne l'exige pas explicitement. En outre, s'ils demandaient un verrouillage côté client, quel est le verrou que le client devrait utiliser pour synchroniser son code ? La documentation ne le dit pas et on imagine mal comment le deviner. Cette "supposition raisonnable" est confirmée par les exemples de la spécification et des didacticiels officiels, qui montrent comment accéder à un objet `ServletContext` ou `HttpSession` sans utiliser la moindre synchronisation côté client.

En revanche, les objets placés avec `setAttribute()` dans un objet `ServletContext` ou `HttpSession` appartiennent à l'application web, pas au conteneur de servlets. La spécification des servlets ne suggère aucun mécanisme pour coordonner les accès concurrents aux attributs partagés. Les attributs stockés par le conteneur pour le compte de l'application devraient donc être thread-safe ou non modifiables dans les faits. Si le conteneur se contentait de les stocker pour l'application web, une autre possibilité serait de s'assurer qu'ils sont correctement protégés par un verrou lorsqu'ils sont utilisés par le code de l'application servlet. Cependant, le conteneur pouvant vouloir sérialiser les objets dans `HttpSession` pour des besoins de réplication ou de passivation et le conteneur de servlets ne pouvant pas connaître votre protocole de verrouillage, c'est à vous de les rendre thread-safe.

On peut faire le même raisonnement pour l'interface `DataSource` de JDBC, qui représente un pool de connexions réutilisables. Un objet `DataSource` fournissant un service à une application, il serait surprenant que cela se passe dans un contexte monothread : il est en effet difficilement imaginable que cela n'implique pas un appel à `getConnection()` à partir de plusieurs threads. D'ailleurs, comme pour les servlets, la spécification JDBC ne suggère pas de verrouillage côté client dans les nombreux exemples de code qui utilisent `DataSource`. Par conséquent, bien que la spécification ne promette pas que `DataSource` soit thread-safe et n'exige pas que les éditeurs de conteneurs fournissent une implémentation thread-safe, nous sommes obligés de supposer que `DataSource.getConnection()` n'exige pas de verrouillage côté client.

En revanche, on ne peut pas en dire autant des objets `Connection` JDBC fournis par `DataSource` puisqu'ils ne sont pas nécessairement prévus pour être partagés par d'autres activités tant qu'ils ne sont pas revenus dans le pool. Si une activité obtient un objet `Connection` JDBC et qu'elle se décompose en plusieurs threads, elle doit prendre la

responsabilité de garantir que l'accès à cet objet est correctement protégé par synchronisation (dans la plupart des applications, les activités qui utilisent un objet Connection JDBC sont implémentées de sorte à confiner cet objet dans un thread particulier).

5

Briques de base

Le chapitre précédent a exploré plusieurs techniques pour construire des classes thread-safe, notamment la délégation de la thread safety à des classes thread-safe existantes. Lorsqu'elle est applicable, la délégation est l'une des stratégies les plus efficaces pour créer des classes thread-safe : il suffit de laisser les classes thread-safe existantes gérer l'intégralité de l'état.

Les bibliothèques de la plate-forme Java contiennent un large ensemble de briques de base pour les applications concurrentes, comme les collections thread-safe et divers synchronisateurs qui permettent de coordonner le flux de contrôle des threads qui coopèrent à l'exécution. Nous présenterons ici les plus utiles, notamment celles qui ont été introduites par Java 5.0 et Java 6, et nous montrerons quelques patrons d'utilisation qui permettent de structurer les applications concurrentes.

5.1 Collections synchronisées

Les classes collections synchronisées incluent `Vector` et `Hashtable`, qui étaient déjà là dans le premier JDK, ainsi que leurs cousines ajoutées dans le JDK 1.2, les classes enveloppes synchronisées que l'on crée par les méthodes fabriques `Collections .synchronizedXxx()`. Ces classes réalisent la thread safety en encapsulant leur état et en synchronisant chaque méthode publique de sorte qu'un seul thread puisse accéder à l'état de la collection à un instant donné.

5.1.1 Problèmes avec les collections synchronisées

Bien que les collections synchronisées soient thread-safe, il faut parfois utiliser un verrouillage côté client pour protéger les actions composées comme les itérations (réécupération répétée de tous les éléments de la collection), les navigations (rechercher un élément selon un certain ordre) et les opérations conditionnelles de type *ajouter-si-absent* (vérifier qu'une clé `K` d'un objet `Map` a une valeur associée et ajouter l'association

(K, V) dans le cas contraire). Avec une collection synchronisée, ces actions composées sont techniquement thread-safe, même sans verrouillage côté client, mais elles peuvent ne pas se comporter comme vous vous y attendez lorsque d'autres threads peuvent en même temps modifier la collection.

Le Listing 5.1 présente deux méthodes portant sur un Vector, getLast() et deleteLast(), qui sont toutes les deux des séquences *tester-puis-agir*. Chaque appel détermine la taille du tableau et utilise ce résultat pour récupérer ou supprimer son dernier élément.

Listing 5.1 : Actions composées sur un Vector pouvant produire des résultats inattendus.

```
public static Object getLast(Vector list) {  
    int lastIndex = list.size() - 1;  
    return list.get(lastIndex);  
}  
  
public static void deleteLast(Vector list) {  
    int lastIndex = list.size() - 1;  
    list.remove(lastIndex);  
}
```



Ces méthodes semblent inoffensives et le sont dans un certain sens : elles ne peuvent pas abîmer le Vector, quel que soit le nombre de threads qui les appellent simultanément. Mais le code qui appelle ces méthodes peut avoir une opinion différente. Si un thread A appelle getLast() sur un Vector de dix éléments, qu'un thread B appelle deleteLast() sur le même Vector et que ces opérations soient entrelacées comme à la Figure 5.1, getLast() lancera ArrayIndexOutOfBoundsException. Entre l'appel à size() et celui de get() dans getLast(), le Vector s'est rétréci et l'indice calculé lors de la première étape n'est donc plus valide. Le comportement est parfaitement cohérent avec la spécification de Vector – une exception est levée lorsque l'on demande un élément qui n'existe pas – mais ce n'est pas ce qu'attendait celui qui a appelé getLast(), même en présence d'une modification concurrente (sauf, peut-être, si le Vector était vide).

Les collections synchronisées respectant une politique de synchronisation qui autorise le verrouillage côté client¹, il est alors possible de créer de nouvelles opérations qui seront atomiques par rapport aux autres opérations de la collection du moment que l'on connaît le verrou à utiliser. En effet, ces classes protègent chaque méthode avec un verrou portant sur l'objet collection lui-même : en prenant ce verrou, comme dans le Listing 5.2, nous pouvons donc rendre getLast() et deleteLast() atomiques, ce qui garantit que la taille du Vector ne changera pas entre l'appel à size() et celui de get().

Le risque que la taille du Vector puisse changer entre un appel à size() et l'appel à get() existe aussi lorsque l'on parcourt les éléments du Vector comme dans le Listing 5.3.

1. Ceci n'est documenté que très vaguement dans le Javadoc de Java 5.0, comme exemple d'idiome d'itération correct.

Cet idiome d’itération suppose en effet que les autres threads ne modifieront pas le Vector entre les appels à `size()` et à `get()`. Dans un environnement monothread, cette supposition est tout à fait correcte mais, si d’autres threads peuvent modifier le Vector de façon concurrente, cela posera des problèmes. Comme précédemment, l’exception `ArrayIndexOutOfBoundsException` sera déclenchée si un autre thread supprime un élément pendant que l’on parcourt le Vector et que l’entrelacement des opérations est défavorable.

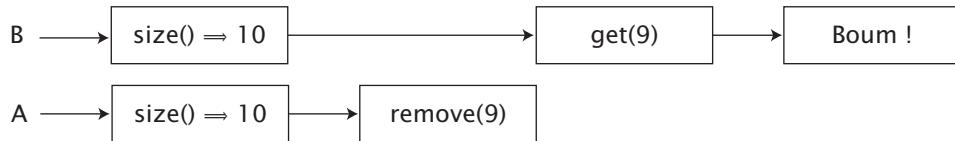


Figure 5.1

Entrelacement de `getLast()` et `deleteLast()` déclenchant `ArrayIndexOutOfBoundsException`.

Listing 5.2 : Actions composées sur Vector utilisant un verrouillage côté client.

```

public static Object getLast(Vector list) {
    synchronized(list) {
        int lastIndex = list.size() - 1;
        return list.get(lastIndex);
    }
}

public static void deleteLast(Vector list) {
    synchronized(list) {
        int lastIndex = list.size() - 1;
        list.remove(lastIndex);
    }
}

```

Listing 5.3 : Itération pouvant déclencher `ArrayIndexOutOfBoundsException`.

```

for (int i = 0; i < vector.size(); i++)
    doSomething(vector.get(i));

```

Le fait que l’itération du Listing 5.3 puisse lever une exception ne signifie pas que Vector ne soit pas thread-safe : l’état du Vector est toujours valide et l’exception est, en réalité, conforme à la spécification. Cependant, qu’une opération aussi banale que la récupération du dernier élément d’une itération déclenche une exception est clairement un comportement indésirable.

Ce problème peut être réglé par un verrouillage côté client, au prix d’un coût supplémentaire en terme d’adaptabilité. En gardant le verrou du Vector pendant la durée de l’itération, comme dans le Listing 5.4, on empêche les autres threads de le modifier pendant qu’on le parcourt. Malheureusement, ceci empêche également les autres threads d’y accéder pendant ce laps de temps, ce qui détériore la concurrence.

Listing 5.4 : Itération avec un verrouillage côté client.

```
synchronized(vector) {  
    for (int i = 0; i < vector.size(); i++)  
        doSomething(vector.get(i));  
}
```

5.1.2 Itérateurs et ConcurrentModificationException

Dans de nombreux exemples, nous utilisons Vector pour des raisons de simplicité, bien qu'elle soit considérée comme une classe collection "historique". Cela dit, les classes plus "modernes" n'éliminent pas le problème des actions composées. La méthode standard pour parcourir une Collection consiste à utiliser un Iterator, soit explicitement, soit via la syntaxe de la boucle "pour-chaque" introduite par Java 5.0, mais l'utilisation d'itérateurs n'empêche pas qu'il faille verrouiller la collection pendant son parcours si d'autres threads peuvent la modifier en même temps. Les itérateurs renvoyés par les collections synchronisées ne sont pas conçus pour gérer les modifications concurrentes et ils échouent rapidement (on dit que ce sont des itérateurs *fail-fast*) : s'ils détectent que la collection a été modifiée depuis le départ de l'itération, ils lancent l'exception non contrôlée ConcurrentModificationException.

Ces itérateurs "fail-fast" sont conçus non pour être infaillibles mais pour capturer "de bonne foi" les erreurs de concurrence ; ce ne sont donc que des indicateurs précoce des problèmes de concurrence. Ils sont implémentés en associant un compteur de modification à la collection : si ce compteur change au cours de l'itération, hasNext() ou next() lancent ConcurrentModificationException. Cependant, ce test étant effectué sans synchronisation, il existe un risque de voir une valeur obsolète du compteur, et l'itérateur peut alors ne pas réaliser qu'une modification a eu lieu. Il s'agit d'un compromis délibéré pour réduire l'impact de ce code de détection sur les performances¹.

Le Listing 5.5 parcourt une collection avec la syntaxe de la boucle pour-chaque. En interne, javac produit un code qui utilise un Iterator et appelle hasNext() et next() pour parcourir la liste. Comme pour le parcours d'un Vector, il faut donc maintenir un verrou sur la collection pendant la durée de l'itération si l'on veut éviter le déclenchement de ConcurrentModificationException.

Listing 5.5 : Parcours d'un objet List avec un Iterator.

```
List<Widget> widgetList  
    = Collections.synchronizedList (new ArrayList<Widget>());  
...  
// Peut lever ConcurrentModificationException  
for (Widget w : widgetList)  
    doSomething(w);
```



1. ConcurrentModificationException peut également survenir dans un code monothread ; elle est levée lorsque des objets sont supprimés directement de la collection plutôt qu'avec Iterator.remove().

Il y a cependant plusieurs raisons pour lesquelles le verrouillage d'une collection pendant la durée de son parcours n'est pas souhaitable. Les autres threads qui ont besoin d'accéder à la collection se bloqueront jusqu'à la fin de l'itération et, si la collection est grande ou que l'opération effectuée sur chaque élément prenne un certain temps, ils peuvent attendre longtemps. En outre, si la collection est verrouillée comme dans le Listing 5.4, `doSomething()` est appelée sous le couvert d'un verrou, ce qui peut provoquer un blocage définitif (ou *deadlock*, voir Chapitre 10). Même en l'absence de risque de famine ou de deadlock, le verrouillage des collections pendant un temps non négligeable peut gêner l'adaptabilité de l'application. Plus le verrou est détenu longtemps, plus il sera disputé et, si de nombreux threads se bloquent en attente de la disponibilité d'un verrou, l'utilisation du processeur peut en pâtrir (voir Chapitre 11).

Au lieu de verrouiller la collection pendant l'itération, on peut cloner la collection et itérer sur la copie. Le clone étant confiné au thread, aucun autre ne pourra le modifier pendant son parcours, ce qui élimine le risque de l'exception `ConcurrentModificationException` (mais il faut quand même verrouiller la collection pendant l'opération de clonage). Cloner une collection a évidemment un coût en termes de performances ; que ce soit un compromis acceptable ou non dépend de nombreux facteurs, dont la taille de la collection, le type de traitement à effectuer sur chaque élément, la fréquence relative de l'itération par rapport aux autres opérations sur la collection et les exigences en termes de réactivité et de débit des données.

5.1.3 Itérateurs cachés

Bien que le verrouillage puisse empêcher les itérateurs de lancer `ConcurrentModificationException`, vous devez penser à l'utiliser à chaque fois qu'une collection partagée peut être parcourue par itération. Ceci est plus difficile qu'il n'y paraît car les itérateurs sont parfois *cachés*, comme dans la classe `HiddenIterator` du Listing 5.6. Cette classe ne contient aucune itération explicite, mais le code en gras revient à en faire une. En effet, la concaténation des chaînes est traduite par le compilateur en un appel à `StringBuilder.append(Object)`, qui, à son tour, appelle la méthode `toString()` de la collection, or l'implémentation de `toString()` dans les collections standard parcourt la collection par itération et appelle `toString()` sur chaque élément pour produire une représentation correctement formatée du contenu de celle-ci.

La méthode `addTenThings()` pourrait donc lancer `ConcurrentModificationException` puisque la collection est itérée par `toString()` au cours de la préparation du message de débogage. Le véritable problème, bien sûr, est que `HiddenIterator` n'est pas thread-safe ; son verrou devrait être pris avant d'utiliser `set()` dans l'appel à `println()`, mais les codes de débogage et de journalisation négligent souvent de le faire.

La leçon qu'il faut en tirer est que plus la distance est importante entre l'état et la synchronisation qui le protège, plus il est probable qu'on oublie d'utiliser une synchronisation adéquate lorsqu'on accédera à cet état. Si `HiddenIterator` enveloppait l'objet

HashSet dans un synchronizedSet en encapsulant ainsi la synchronisation, ce type d'erreur ne pourrait pas survenir.

Tout comme l'encapsulation de l'état d'un objet facilite la préservation de ses invariants, l'encapsulation de sa synchronisation facilite le respect de sa politique de synchronisation.

Listing 5.6 : Itération cachée dans la concaténation des chaînes. Ne le faites pas.

```
public class HiddenIterator {  
    @GuardedBy("this")  
    private final Set<Integer> set = new HashSet<Integer>();  
  
    public synchronized void add(Integer i) { set.add(i); }  
    public synchronized void remove(Integer i) { set.remove(i); }  
  
    public void addTenThings() {  
        Random r = new Random();  
        for (int i = 0; i < 10; i++)  
            add(r.nextInt());  
        System.out.println("DEBUG: added ten elements to " + set);  
    }  
}
```



L'itération est également invoquée indirectement par les méthodes hashCode() et equals() de la collection, qui peuvent être appelées si la collection est utilisée comme un élément ou une clé d'une autre collection. De même, les méthodes containsAll(), removeAll() et retainAll(), ainsi que les constructeurs qui prennent des collections en paramètre, parcouruent également la collection. Toutes ces utilisations indirectes de l'itération peuvent provoquer une exception ConcurrentModificationException.

5.2 Collections concurrentes

Java 5.0 améliore les collections synchronisées en fournissant plusieurs classes collections concurrentes. Les collections synchronisées réalisent leur thread safety en sérialisant tous les accès à l'état de la collection, ce qui donne une concurrence assez pauvre : quand plusieurs threads concourent pour obtenir le verrou de la collection, le débit des données en souffre.

Les collections concurrentes, en revanche, sont conçues pour les accès concurrents à partir de plusieurs threads. Java 5.0 ajoute ConcurrentHashMap pour remplacer les implémentations de Map reposant sur des hachages synchronisés et CopyOnWriteArrayList pour remplacer les implémentations de List synchronisées dans les cas où l'opération prédominante est le parcours d'une liste. La nouvelle interface ConcurrentMap, quant à elle, ajoute le support des actions composées classiques, comme *ajouter-si-absent*, remplacer et suppression conditionnelle.

Le remplacement des collections synchronisées par les collections concurrentes permet d'ajouter énormément d'adaptabilité avec peu de risques.

Java 5.0 ajoute également deux nouveaux types collection, `Queue` et `BlockingQueue`. Un objet `Queue` est conçu pour contenir temporairement un ensemble d'éléments pendant qu'ils sont en attente de traitement. Plusieurs implémentations sont également fournies, dont `ConcurrentLinkedQueue`, une file d'attente classique, et `PriorityQueue`, une file (non concurrente) à priorités. Les opérations sur `Queue` ne sont pas bloquantes ; si la file est vide, une opération de lecture renverra `null`. Bien que l'on puisse simuler le comportement de `Queue` avec `List` (en fait, `LinkedList` implémente également `Queue`), les classes `Queue` ont été ajoutées car éliminer la possibilité d'accès direct de `List` permet d'obtenir des implémentations concurrentes plus efficaces.

`BlockingQueue` étend `Queue` pour lui ajouter des opérations d'insertion et de lecture bloquantes. Si la file est vide, la récupération d'un élément se bloquera jusqu'à ce qu'une donnée soit disponible et, si la file est pleine (dans le cas des files de tailles limitées), une insertion se bloquera tant qu'il n'y a pas d'emplacement libre. Les files bloquantes sont très importantes dans les schémas producteur-consommateur et seront présentées en détail dans la section 5.3.

Tout comme `ConcurrentHashMap` est un remplacement concurrent pour les `Map` synchronisés, Java 6 ajoute `ConcurrentSkipListMap` et `ConcurrentSkipListSet` comme remplacements concurrents des `SortedMap` ou `SortedSet` synchronisés (comme `TreeMap` ou `TreeSet` enveloppés par `synchronizedMap`).

5.2.1 ConcurrentHashMap

Les collections synchronisées maintiennent un verrou pour la durée de chaque opération. Certaines, comme `HashMap.get()` ou `List.contains()`, peuvent impliquer plus de travail qu'on pourrait le penser : parcourir un hachage ou une liste pour trouver un objet spécifique nécessite d'appeler la méthode `equals()` (qui, elle-même, peut impliquer un calcul assez complexe) sur un certain nombre d'objets candidats. Dans une collection de type hachage, si `hashCode()` ne répartit pas bien les valeurs de hachage, les éléments peuvent être distribués inégalement : dans le pire des cas, une mauvaise fonction de hachage transformera un hachage en liste chaînée. Le parcours d'une longue liste et l'appel de `equals()` sur quelques éléments ou sur tous les éléments peut donc prendre un certain temps pendant lequel aucun autre thread ne pourra accéder à la collection.

`ConcurrentHashMap` est un hachage `Map` comme `HashMap`, sauf qu'elle utilise une stratégie de verrouillage entièrement différente qui offre une meilleure concurrence et une adaptabilité supérieure. Au lieu de synchroniser chaque méthode sur un verrou commun, ce qui restreint l'accès à un seul thread à la fois, elle utilise un mécanisme de verrouillage plus fin, appelé *verrouillage partitionné* (*lock striping*, voir la section 11.4.3) afin

d'autoriser un plus grand degré d'accès partagé. Un nombre quelconque de threads lecteurs peuvent accéder simultanément au hachage, en même temps que les écrivains, et un nombre limité de threads écrivains peuvent modifier le hachage de façon concurrente. On obtient ainsi un débit de données bien plus important lors des accès concurrents, moyennant une petite perte de performances pour les accès monothreads.

`ConcurrentHashMap`, ainsi que les autres collections concurrentes, améliore encore les classes collections synchronisées en fournissant des itérateurs qui ne lancent pas `ConcurrentModificationException` : il n'est donc plus nécessaire de verrouiller la collection pendant l'itération. Les itérateurs renvoyés par `ConcurrentHashMap` sont *faiblement cohérents* au lieu d'être *fail-fast*. Un itérateur faiblement cohérent peut tolérer une modification concurrente, il parcourt les éléments tels qu'ils étaient lorsque l'itérateur a été construit et peut (mais ce n'est pas garanti) refléter les modifications apportées à la collection après la construction de l'itérateur.

Comme avec toutes les améliorations, il a fallu toutefois faire quelques compromis. La sémantique des méthodes qui agissent sur tout l'objet `Map`, comme `size()` et `isEmpty()`, a été légèrement affaiblie pour refléter la nature concurrente de la collection. Le résultat de `size()` pouvant être obsolète au moment où il est calculé, cette méthode peut donc renvoyer une valeur approximative au lieu d'un comptage exact. Bien que cela puisse sembler troublant au premier abord, les méthodes comme `size()` et `isEmpty()` sont, en réalité, bien moins utiles dans les environnements concurrents puisque ces valeurs sont des cibles mouvantes. Les exigences de ces opérations ont donc été affaiblies pour permettre d'optimiser les performances des opérations plus importantes que sont `get()`, `put()`, `containsKey()` et `remove()`.

La seule fonctionnalité offerte par les implémentations synchronisées de `Map` qui ne se retrouve pas dans `ConcurrentHashMap` est la possibilité de verrouiller le hachage pour disposer d'un accès exclusif. Avec `Hashtable` et `synchronizedMap`, la détention du verrou de l'objet `Map` empêche tout autre thread d'y accéder. Cela peut être nécessaire dans des cas spéciaux, comme lorsque l'on souhaite ajouter plusieurs associations de façon atomique ou que l'on veut parcourir plusieurs fois le hachage en voyant à chaque fois les éléments dans le même ordre. Cependant, globalement, l'absence de cette fonctionnalité est un compromis acceptable puisque les collections concurrentes sont censées changer leurs éléments en permanence.

`ConcurrentHashMap` ayant de nombreux avantages et peu d'inconvénients par rapport à `Hashtable` ou `synchronizedMap`, vous avez tout intérêt à remplacer ces dernières par `ConcurrentHashMap` afin de disposer d'une plus grande adaptabilité. Ce n'est que lorsqu'une application a besoin de verrouiller un hachage pour y accéder de manière exclusive¹ que `ConcurrentHashMap` n'est pas la meilleure solution.

1. Ou si vous avez besoin des effets de bord de la synchronisation des implémentations synchronisées de `Map`.

5.2.2 Opérations atomiques supplémentaires sur les Map

Un objet ConcurrentHashMap ne pouvant pas être verrouillé pour garantir un accès exclusif, nous ne pouvons pas utiliser le verrouillage côté client pour créer de nouvelles opérations atomiques comme *ajouter-si-absent*, comme nous l'avons fait pour Vector dans la section 4.4.1. Cependant, un certain nombre d'opérations composées comme *ajouter-si-absent*, *supprimer-si-égal* et *remplacer-si-égal* sont implémentées sous forme d'opérations atomiques par l'interface ConcurrentMap présentée dans le Listing 5.7. Si vous constatez que vous devez ajouter l'une de ces fonctionnalités à une implémentation existante de Map synchronisée, il s'agit sûrement d'un signe indiquant que vous devriez utiliser ConcurrentMap à la place.

Listing 5.7 : Interface ConcurrentMap.

```
public interface ConcurrentMap<K,V> extends Map<K,V> {  
    // Insertion uniquement si K n'a pas de valeur associée  
    V putIfAbsent(K key, V value);  
  
    // Suppression uniquement si K est associée à V  
    boolean remove(K key, V value);  
  
    // Remplacement de la valeur uniquement si K est associée à oldValue  
    boolean replace(K key, V oldValue, V newValue);  
  
    // Remplacement de la valeur uniquement si K est associée à une valeur  
    V replace(K key, V newValue);  
}
```

5.2.3 CopyOnWriteArrayList

CopyOnWriteArrayList remplace les List synchronisées et offre une meilleure gestion de la concurrence dans quelques situations classiques. Avec cette classe, il n'y a plus besoin de verrouiller ou de copier une collection pendant les itérations. De la même façon, CopyOnWriteArraySet remplace les Set synchronisés.

La thread safety des collections "copie lors de l'écriture" vient du fait qu'il n'y a pas besoin de synchronisation supplémentaire pour accéder à un objet non modifiable dans les faits qui a été correctement publié. Ces classes implémentent la "mutabilité" en créant et en republant une nouvelle copie de la collection à chaque fois qu'elle est modifiée. Les itérateurs de ces collections stockent une référence au tableau sous-jacent tel qu'il était au début de l'itération et, comme il ne changera jamais, ils doivent ne se synchroniser que très brièvement pour assurer la visibilité de son contenu. Plusieurs threads peuvent donc parcourir la collection sans interférer les uns avec les autres ni avec ceux qui veulent la modifier. Les itérateurs renvoyés par ces collections ne lèvent pas ConcurrentModificationException et renvoient les éléments tels qu'ils étaient à la création de l'itérateur, quelles que soient les modifications apportées ensuite.

Évidemment, la copie du tableau sous-jacent à chaque modification de la collection a un coût, notamment lorsque la collection est importante ; il ne faut utiliser les collections

"copie lors de l'écriture" que lorsque les opérations d'itérations sont bien plus fréquentes que celles de modification. Ce critère s'applique en réalité aux nombreux systèmes de notification d'événements : signaler un événement nécessite de parcourir la liste des écouteurs enregistrés et d'appeler chacun d'eux. Or, dans la plupart des cas, enregistrer ou désinscrire un écouteur d'événement est une opération bien moins fréquente que recevoir une notification d'événement (voir [CPJ 2.4.4] pour plus d'informations sur la "copie lors de l'écriture").

5.3 Files bloquantes et patron producteur-consommateur

Les files bloquantes fournissent les méthodes bloquantes `put()` et `take()`, ainsi que leurs équivalents `offer()` et `poll()`, qui utilisent un délai d'expiration. Si la file est pleine, `put()` se bloque jusqu'à ce qu'un emplacement soit disponible ; si la file est vide, `take()` se bloque jusqu'à ce qu'il y ait un élément disponible. Les files peuvent être bornées ou non bornées ; les files non bornées n'étant jamais pleines, un appel à `put()` sur ce type de file ne sera donc jamais bloquant.

Les files bloquantes reconnaissent le patron de conception "producteur-consommateur". Ce patron permet de séparer l'identification d'un travail de son exécution en plaçant les tâches à réaliser dans une liste "à faire" qui sera traitée plus tard au lieu de l'être immédiatement à mesure qu'elles sont identifiées. Le patron producteur-consommateur simplifie donc le développement puisqu'il supprime les dépendances entre les classes producteurs et consommateurs ; il allège également la gestion de la charge de travail en découplant les activités qui peuvent produire ou consommer des données à des vitesses différentes ou variables.

Dans une conception producteur-consommateur construite autour d'une file bloquante, les producteurs placent les données dans la file dès qu'elles deviennent disponibles et les consommateurs les récupèrent dans cette file lorsqu'ils sont prêts à effectuer les actions adéquates. Les producteurs n'ont pas besoin de connaître ni l'identité, ni le nombre de consommateurs, ni même s'il y a d'autres producteurs : leur seule tâche consiste à placer des données dans la file. De même, les consommateurs n'ont pas besoin de savoir qui sont les producteurs ni d'où provient le travail. `BlockingQueue` simplifie l'implémentation de ce patron avec un nombre quelconque de producteurs et de consommateurs. L'une des applications les plus fréquentes du modèle producteur-consommateur est un pool de threads associé à une file de tâches ; ce patron est d'ailleurs intégré dans le framework d'exécution de tâches `Executor`, qui sera présenté aux Chapitres 6 et 8.

Un exemple d'application du patron producteur-consommateur est la division des tâches entre deux personnes qui font la vaisselle : l'une lave les plats et les pose sur l'égouttoir, l'autre prend les plats sur l'égouttoir et les essuie. Dans ce schéma, l'égouttoir sert de file bloquante ; s'il n'y a aucun plat dessus, le consommateur attend qu'il y ait des plats à essuyer et, lorsque l'égouttoir est plein, le producteur doit s'arrêter de laver jusqu'à ce

qu'il y ait de la place. Cette analogie s'étend à plusieurs producteurs (bien qu'il puisse y avoir un conflit pour l'accès à l'évier) et à plusieurs consommateurs ; chaque personne n'interagit qu'avec l'égouttoir. Personne n'a besoin de savoir quel est le nombre de producteurs ou de consommateurs ni qui a produit un élément particulier.

Les étiquettes "producteur" et "consommateur" sont relatives ; une activité qui agit comme un consommateur dans un contexte peut très bien agir comme un producteur dans un autre. Essuyer les plats "consomme" des plats propres et mouillés et "produit" des plats propres et secs. Une troisième personne voulant aider les deux plongeurs pourrait retirer les plats essuyés, auquel cas celui qui essuie serait à la fois un consommateur et un producteur et il y aurait alors deux files des tâches partagées (chacune pouvant bloquer celui qui essuie en l'empêchant de continuer son travail).

Les files bloquantes simplifient le codage des consommateurs car `take()` se bloque jusqu'à ce qu'il y ait des données disponibles. Si les producteurs ne produisent pas assez vite pour occuper les consommateurs, ceux-ci peuvent simplement attendre que du travail arrive. Parfois, ce fonctionnement est tout à fait acceptable (c'est ce que fait une application serveur lorsqu'aucun client ne demande ses services) et, parfois, cela indique que le nombre de threads producteurs par rapport aux threads consommateurs doit être ajusté pour améliorer l'utilisation (comme pour un robot web ou toute autre application qui a un travail infini à réaliser).

Si les producteurs produisent toujours plus vite que les consommateurs ne peuvent traiter, l'application risque de manquer de mémoire car les tâches s'ajouteront sans fin à la file. Là aussi, la nature bloquante de `put()` simplifie beaucoup le codage des producteurs : si l'on utilise une *file bornée*, les producteurs seront bloqués lorsque cette file sera pleine, ce qui laissera le temps aux consommateurs de rattraper leur retard puisqu'un producteur bloqué ne peut pas générer d'autre tâche.

Les files bloquantes fournissent également une méthode `offer()` qui renvoie un code d'erreur si l'élément n'a pas pu être ajouté. Elle permet donc de mettre en place des politiques plus souples pour gérer la surcharge en réduisant par exemple le nombre de threads producteurs ou en les ralentissant d'une manière ou d'une autre.

Les files bornées constituent un outil de gestion des ressources relativement puissant pour produire des applications fiables : elles rendent les programmes plus résistants à la surcharge en ralentissant les activités qui menacent de produire plus de travail qu'on ne peut en traiter.

Bien que le patron producteur-consommateur permette de découpler les codes du producteur et du consommateur, leur comportement reste indirectement associé par la file des tâches partagée. Il est tentant de supposer que les consommateurs suivront toujours le rythme, ce qui évite de devoir placer des limites à la taille des files de tâches,

mais cette supposition vous amènerait à réarchitecturer votre système plus tard. *Construez la gestion des ressources dès le début de la conception en utilisant des files bloquantes : c'est bien plus facile de le faire d'abord que de faire machine arrière plus tard.* Les files bloquantes facilitent cette gestion dans un grand nombre de situations mais, si elles s'intègrent mal à votre conception, vous pouvez construire d'autres structures de données bloquantes à l'aide de la classe `Semaphore` (voir la section 5.5.3).

La bibliothèque de classes contient plusieurs implémentations de `BlockingQueue`. `LinkedBlockingQueue` et `ArrayBlockingQueue` sont des files d'attente FIFO, analogues à `LinkedList` et `ArrayList` mais avec une meilleure concurrence que celle d'une `List` synchronisée. `PriorityBlockingQueue` est une file à priorités, ce qui est utile lorsque l'on veut traiter des éléments dans un autre ordre que celui de leur insertion dans la file. Tout comme les collections triées, `PriorityBlockingQueue` peut comparer les éléments selon leur ordre naturel (s'ils implémentent `Comparable`) ou à l'aide d'un `Comparator`.

La dernière implémentation de `BlockingQueue`, `SynchronousQueue`, n'est en fait pas une file d'attente du tout car elle ne gère aucun espace de stockage pour les éléments placés dans la file. Au lieu de cela, elle maintient une liste des threads qui attendent d'ajouter et de supprimer un élément de la file. Dans notre analogie de la vaisselle, cela reviendrait à ne pas avoir d'égouttoir mais à passer directement les plats lavés au sécheur disponible suivant. Bien que cela semble être une étrange façon d'implémenter une file, cela permet de réduire le temps de latence associé au déplacement des données du producteur au consommateur car les tâches sont traitées directement (dans une file traditionnelle, les opérations d'ajout et de suppression doivent s'effectuer en séquence avant qu'une tâche puisse être traitée). Ce traitement direct fournit également plus d'informations au producteur sur l'état de la tâche ; lorsqu'elle est prise en charge, le producteur sait qu'un consommateur en a pris la responsabilité au lieu d'être placée quelque part dans la file (c'est la même différence qu'entre remettre un document en main propre à un collègue et le placer dans sa boîte aux lettres en espérant qu'il la relèvera bientôt). Un objet `SynchronousQueue` n'ayant pas de capacité de stockage, les méthodes `put()` et `take()` se bloqueront si un autre thread n'est pas prêt à participer au traitement. Les files synchrones ne conviennent généralement qu'aux situations où il y a suffisamment de consommateurs et où il y en a presque toujours un qui sera prêt à traiter la tâche.

5.3.1 Exemple : indexation des disques

Les agents qui parcourent les disques durs locaux et les indexent pour accélérer les recherches ultérieures (comme *Google Desktop* et le service d'indexation de Windows) sont de bons clients pour la décomposition en producteurs et consommateurs. La classe `FileCrawler` du Listing 5.8 présente une tâche productrice qui recherche dans une arborescence de répertoires les fichiers correspondant à un critère d'indexation et place

leurs noms dans la file des tâches ; la classe `Indexer` du Listing 5.8 montre la tâche consommatrice qui extrait les noms de la file et les indexe.

Le patron producteur-consommateur offre un moyen de décomposer ce problème d'indexation en threads afin d'obtenir des composants plus simples. La factorisation du parcours et de l'indexation de fichiers en activités distinctes produit un code plus lisible et mieux réutilisable qu'une activité monolithique qui s'occuperaient de tout ; chaque activité n'a qu'une seule tâche à réaliser et la file bloquante gère tout le contrôle de flux.

Le patron producteur-consommateur a également plusieurs avantages en termes de performances. Les producteurs et les consommateurs peuvent s'exécuter en parallèle ; si l'un d'eux est lié aux E/S et que l'autre est lié au processeur, leur exécution concurrente produira un débit global supérieur à celui obtenu par leur exécution séquentielle. Si les activités du producteur et du consommateur sont parallélisables à des degrés différents, un couplage trop fort risque de ramener ce parallélisme à celui de l'activité la moins parallélisable.

Le Listing 5.9 lance plusieurs `FileCrawler` et `Indexer` dans des threads différents. Tel que ce code est écrit, le thread consommateur ne se terminera jamais, ce qui empêchera le programme lui-même de se terminer ; nous étudierons plusieurs techniques pour corriger ce problème au Chapitre 7. Bien que cet exemple utilise des threads gérés explicitement, de nombreuses conceptions producteur-consommateur peuvent s'exprimer à l'aide du framework d'exécution des tâches `Executor`, qui utilise lui-même le patron producteur-consommateur.

Listing 5.8 : Tâches producteur et consommateur dans une application d'indexation des fichiers.

```
public class FileCrawler implements Runnable {
    private final BlockingQueue <File> fileQueue;
    private final FileFilter fileFilter;
    private final File root;
    ...
    public void run() {
        try {
            crawl(root);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    private void crawl(File root) throws InterruptedException {
        File[] entries = root.listFiles(fileFilter);
        if (entries != null) {
            for (File entry : entries)
                if (entry.isDirectory())
                    crawl(entry);
                else if (!alreadyIndexed (entry))
                    fileQueue.put(entry);
        }
    }
}
```

Listing 5.8 : Tâches producteur et consommateur dans une application d'indexation des fichiers. (suite)

```
public class Indexer implements Runnable {
    private final BlockingQueue <File> queue;

    public Indexer(BlockingQueue<File> queue) {
        this.queue = queue;
    }

    public void run() {
        try {
            while (true)
                indexFile(queue.take());
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}
```

Listing 5.9 : Lancement de l'indexation.

```
public static void startIndexing(File[] roots) {
    BlockingQueue<File> queue = new LinkedBlockingQueue <File>(BOUND);
    FileFilter filter = new FileFilter() {
        public boolean accept(File file) { return true; }
    };

    for (File root : roots)
        new Thread(new FileCrawler(queue, filter, root)).start();

    for (int i = 0; i < N_CONSUMERS; i++)
        new Thread(new Indexer(queue)).start();
}
```

5.3.2 Confinement en série

Les implémentations des files bloquantes dans `java.util.concurrent` contiennent toutes une synchronisation interne suffisante pour publier correctement les objets d'un thread producteur vers un thread consommateur.

Pour les objets modifiables, les conceptions producteur-consommateur et les files bloquantes facilitent le *confinement en série* pour transférer l'appartenance des objets des producteurs aux consommateurs. Un objet confiné à un thread appartient exclusivement à un seul thread, mais cette propriété peut être "transférée" en le publiant correctement de sorte qu'un seul autre thread y aura accès et en s'assurant que le thread qui publie n'y accédera pas après ce transfert. La publication correcte garantit que l'état de l'objet est visible par son nouveau propriétaire et, le propriétaire initial ne le touchant plus, que cet objet est désormais confiné au nouveau thread. Le nouveau propriétaire peut alors le modifier librement puisqu'il est le seul à y avoir accès.

Les pools d'objets exploitent le confinement en série en "prêtant" un objet à un thread qui le demande. Tant que le pool dispose d'une synchronisation interne suffisante pour publier l'objet correctement et tant que les clients ne publient pas eux-mêmes cet objet

ou ne l'utilisent pas après l'avoir rendu au pool, la propriété peut être transférée en toute sécurité d'un thread à un autre.

On pourrait également utiliser d'autres mécanismes de publication pour transférer la propriété d'un objet modifiable, mais il faut alors s'assurer qu'un seul thread recevra l'objet transféré. Les files bloquantes facilitent cette opération mais, avec un petit peu plus de travail, cela pourrait également être fait en utilisant la méthode atomique `remove()` de `ConcurrentMap` ou la méthode `compareAndSet()` de `AtomicReference`.

5.3.3 Classe Deque et vol de tâches

Java 6 ajoute deux autres types collection, `Deque` (que l'on prononce "deck") et `BlockingDeque`, qui étend `Queue` et `BlockingQueue`. Un objet `Deque` est une file à double entrée qui garantit des insertions et des suppressions efficaces à ses deux extrémités. Ses implémentations s'appellent `ArrayDeque` et `LinkedBlockingDeque`.

Tout comme les files bloquantes se prêtent parfaitement au patron producteur-consommateur, les files doubles sont parfaitement adaptées au patron *vol de tâche*. Alors que le patron producteur-consommateur n'utilise qu'*une seule* file de tâches partagée par tous les consommateurs, le patron vol de tâche utilise une file double *par* consommateur. Un consommateur ayant épuisé toutes les tâches de sa file peut voler une tâche à la fin de la file double d'un autre. Le vol de tâche est plus adaptatif qu'une conception producteur-consommateur traditionnelle car, ici, les travailleurs ne rivalisent pas pour une file partagée ; la plupart du temps, ils n'accèdent qu'à leur propre file double, ce qui réduit donc les rivalités. Lorsqu'un travailleur doit accéder à la file d'un autre, il le fait à partir de la fin de celle-ci plutôt que de son début, ce qui réduit encore les conflits. Le vol de tâche est bien adapté aux problèmes dans lesquels les consommateurs sont également des producteurs – lorsque l'exécution d'une tâche identifie souvent une autre tâche. Le traitement d'une page par un robot web, par exemple, identifie généralement de nouvelles pages à analyser. Les algorithmes de parcours de graphes, comme le marquage du tas par le ramasse-miettes, peuvent aisément être parallélisés à l'aide du vol de tâches. Lorsqu'un travailleur identifie une nouvelle tâche, il la place à la fin de sa propre file double (ou, dans le cas d'un *partage du travail*, sur celle d'un autre travailleur) ; lorsque sa file est vide, il recherche du travail à la fin de la file de quelqu'un d'autre, ce qui garantit que chaque travailleur restera occupé.

5.4 Méthodes bloquantes et interruptions

Les threads peuvent se *bloquer*, ou se mettre en pause, pour plusieurs raisons : ils peuvent attendre la fin d'une opération d'E/S, attendre de pouvoir prendre un verrou, attendre de se réveiller d'un appel à `Thread.sleep()` ou attendre le résultat d'un calcul effectué par un autre thread. Lorsqu'un thread se bloque, il est généralement suspendu et placé dans l'un des états correspondant au blocage des threads (`BLOCKED`, `WAITING` ou `TIMED_WAITING`).

La différence entre une opération bloquante et une opération classique qui met simplement longtemps à se terminer est qu'un thread bloqué doit attendre un événement qu'il ne contrôle pas avant de pouvoir poursuivre – l'opération d'E/S s'est terminée, le verrou est devenu disponible ou le calcul externe s'est achevé. Lorsque cet événement externe survient, le thread est replacé dans l'état RUNNABLE et redevient éligible pour l'accès au processeur.

Les méthodes `put()` et `take()` de `BlockingQueue`, comme un certain nombre d'autres méthodes de la bibliothèque comme `Thread.sleep()`, lancent l'exception contrôlée `InterruptedException`. Lorsqu'une méthode annonce qu'elle peut lancer `InterruptedException`, elle indique qu'elle est bloquante et que, si elle est *interrompue*, elle fera son possible pour se débloquer le plus tôt possible.

La classe `Thread` fournit la méthode `interrupt()` pour interrompre un thread ou savoir si un thread a été interrompu (chaque thread a une propriété booléenne représentant son état d'interruption). L'interruption d'un thread est un mécanisme *coopératif* : un thread ne peut pas forcer un autre à arrêter ce qu'il fait pour faire quelque chose d'autre ; lorsque le thread *A* interrompt le thread *B*, *A* demande simplement que *B* arrête son traitement en cours lorsqu'il trouvera un point d'arrêt judicieux – et s'il en a envie. Bien que l'API ou la spécification du langage ne précisent nulle part à quoi peut servir une interruption dans une application, son utilisation la plus courante consiste à annuler une activité. Les méthodes bloquantes qui répondent aux interruptions facilitent l'annulation à point nommé des activités qui tournent sans fin.

Si vousappelez une méthode susceptible de lever une `InterruptedException`, cette méthode est également bloquante et vous devez avoir un plan pour répondre à l'interruption. Pour le code d'une bibliothèque, deux choix sont essentiellement possibles :

- **Propager l'`InterruptedException`.** C'est souvent la politique la plus raisonnable si vous pouvez le faire – il suffit de propager l'interruption au code appelant. Cela peut impliquer de ne pas capturer `InterruptedException` ou de la capturer afin de la relancer après avoir effectué quelques opérations spécifiques de nettoyage.
- **Restaurer l'interruption.** Parfois, vous ne pouvez pas lancer `InterruptedException` (si votre code fait partie d'un `Runnable`, par exemple). Dans ce cas, vous devez la capturer et restaurer l'état d'interruption en appelant `interrupt()` sur le thread courant, afin que le code plus haut dans la pile des appels puisse voir qu'une interruption est survenue, comme on le montre dans le Listing 5.10.

Vous pouvez aller bien plus loin avec les interruptions, mais ces deux approches suffiront à la grande majorité des situations. Il est toutefois déconseillé de capturer `InterruptedException` pour ne rien faire en réponse. En effet, cela empêcherait le code situé plus haut dans la pile des appels de réagir puisqu'il ne saura jamais que le thread a été interrompu. *La seule situation où l'absorption d'une interruption est acceptable est lorsque*

l'on étend Thread et que l'on veut donc contrôler tout le code situé plus haut dans la pile. L'annulation et les interruptions sont présentées plus en détail au Chapitre 7.

Listing 5.10 : Restauration de l'état d'interruption afin de ne pas absorber l'interruption.

```
public class TaskRunnable implements Runnable {  
    BlockingQueue<Task> queue;  
    ...  
    public void run() {  
        try {  
            processTask(queue.take());  
        } catch (InterruptedException e) {  
            // Restauration de l'état d'interruption  
            Thread.currentThread().interrupt();  
        }  
    }  
}
```

5.5 Synchronisateurs

Les files bloquantes sont uniques parmi les classes collections : non seulement elles servent de conteneurs, mais elles permettent également de coordonner le flux de contrôle des threads producteur et consommateur puisque les méthodes `take()` et `put()` se bloquent jusqu'à ce que la file soit dans l'état désiré (non vide ou non pleine).

Un *synchronisateur* est un objet qui coordonne le contrôle de flux des threads en fonction de son état. Les files bloquantes peuvent donc servir de synchronisateurs ; parmi les autres types, on peut également citer les sémaphores, les barrières et les loquets. Bien que la bibliothèque standard contienne déjà un certain nombre de classes synchronisatrices, vous pouvez créer les vôtres à partir des mécanismes décrits dans le Chapitre 14 si elles ne correspondent pas à vos besoins.

Tous les synchronisateurs partagent certaines propriétés structurelles : ils encapsulent un état qui détermine si les threads qui leur parviennent seront autorisés à passer ou forcés d'attendre, ils fournissent des méthodes pour manipuler cet état et d'autres pour attendre que le synchronisateur soit dans l'état attendu.

5.5.1 Loquets

Un *loquet* est un synchronisateur permettant de retarder l'exécution des threads tant qu'il n'est pas dans son état terminal [CPJ 3.4.2]. Un loquet agit comme une porte : tant qu'il n'est pas dans son état terminal, la porte est fermée et aucun thread ne peut passer ; dans son état terminal, la porte s'ouvre et tous les threads peuvent entrer. Quand un loquet est dans son état terminal, il ne peut changer d'état et reste ouvert à jamais. Les loquets peuvent donc servir à garantir que certaines activités ne se produiront pas tant qu'une autre activité ne s'est pas terminée. Voici quelques exemples d'application :

- Garantir qu'un calcul ne sera pas lancé tant que les ressources dont il a besoin n'ont pas été initialisées. Un simple loquet binaire (à deux états) peut servir à indiquer que

"la ressource R a été initialisée" et toute activité nécessitant R devra attendre ce loquet avant de s'exécuter.

- Garantir qu'un service ne sera pas lancé tant que d'autres services dont il dépend n'ont pas démarré. Chaque service utilise un loquet binaire qui lui est associé ; lancer le service S implique d'abord d'attendre les loquets des autres services dont dépend S, puis de relâcher le loquet de S après son lancement pour que les services qui dépendent de S puissent à leur tour être lancés.
- Attendre que toutes les parties impliquées dans une activité, les joueurs d'un jeu, par exemple, soient prêtes. Dans ce cas, le loquet n'atteint son état terminal que lorsque tous les joueurs sont prêts.

`CountDownLatch` est une implémentation des loquets pouvant être utilisée dans chacune de ces situations ; elle permet à un ou à plusieurs threads d'attendre qu'un ensemble d'événements se produisent. L'état du loquet est formé d'un compteur initialisé avec un nombre positif qui représente le nombre d'événements à attendre. La méthode `countDown()` décrémente ce compteur pour indiquer qu'un événement est survenu, tandis que la méthode `await()` attend que le compteur passe à zéro, ce qui signifie que tous les événements se sont passés. Si le compteur est non nul lorsqu'elle est appelée, `await()` se bloque jusqu'à ce qu'il atteigne zéro, que le thread appelant soit interrompu ou que le délai d'attente soit dépassé.

La classe `TestHarness` du Listing 5.11 illustre deux utilisations fréquentes des loquets. Elle crée un certain nombre de threads qui exécutent en parallèle une tâche donnée et utilise deux loquets, une "porte d'entrée" et une "porte de sortie". Le compteur de la porte d'entrée est initialisé à un, celui de la porte de sortie reçoit le nombre de threads travailleurs. Chaque thread travailleur commence par attendre à la porte d'entrée, ce qui garantit qu'aucun d'eux ne commencera à travailler tant que tous les autres ne sont pas prêts. À la fin de son exécution, chaque thread décrémente le compteur de la porte de sortie, ce qui permet au thread maître d'attendre qu'ils soient tous terminés et de calculer le temps écoulé.

Nous utilisons des loquets dans `TestHarness` au lieu de simplement lancer immédiatement les threads dès qu'ils sont créés car nous voulons mesurer le temps nécessaire à l'exécution de la même tâche *n* fois *en parallèle*. Si nous avions simplement créé et lancé les threads, les premiers auraient été "avantagés" par rapport aux derniers et le degré de contention aurait varié au cours du temps, à mesure que le nombre de threads actifs aurait augmenté ou diminué. L'utilisation d'une porte d'entrée permet au thread maître de lancer simultanément tous les threads travailleurs et la porte de sortie lui permet d'attendre que le *dernier* thread se termine au lieu d'attendre que chacun d'eux se termine un à un.

Listing 5.11 : Utilisation de la classe CountDownLatch pour lancer et stopper des threads et mesurer le temps d'exécution.

```
public class TestHarness {
    public long timeTasks(int nThreads, final Runnable task)
        throws InterruptedException {
        final CountDownLatch startGate = new CountDownLatch(1);
        final CountDownLatch endGate = new CountDownLatch(nThreads);

        for (int i = 0; i < nThreads; i++) {
            Thread t = new Thread() {
                public void run() {
                    try {
                        startGate.await();
                        try {
                            task.run();
                        } finally {
                            endGate.countDown();
                        }
                    } catch (InterruptedException ignored) { }
                }
            };
            t.start();
        }

        long start = System.nanoTime();
        startGate.countDown();
        endGate.await();
        long end = System.nanoTime();
        return end - start;
    }
}
```

5.5.2 FutureTask

FutureTask agit également comme un loquet (elle implémente Future, qui décrit un calcul par paliers [CPJ 4.3.3]). Un calcul représenté par un objet FutureTask est implémenté par un Callable, l'équivalent par paliers de Runnable ; il peut être dans l'un des trois états "attente d'exécution", "en cours d'exécution" ou "terminé". La terminaison englobe toutes les façons par lesquelles un calcul peut se terminer, que ce soit une terminaison normale, une annulation ou une exception. Lorsqu'un objet FutureTask est dans l'état "terminé", il y reste définitivement.

Le comportement de Future.get() dépend de l'état de la tâche. Si elle est terminée, get() renvoie immédiatement le résultat ; sinon elle se bloque jusqu'à ce que la tâche passe dans l'état terminé, puis renvoie le résultat ou lance une exception. FutureTask transfère le résultat du thread qui exécute le calcul au(x) thread(s) qui le récupère(nt) ; sa spécification garantit que ce transfert constitue une publication correcte du résultat.

FutureTask est utilisée par le framework Executor pour représenter les tâches asynchrones et peut également servir à représenter tout calcul potentiellement long pouvant être lancé avant que l'on ait besoin du résultat. La classe Preloader du Listing 5.12 l'utilise pour effectuer un calcul coûteux dont le résultat sera nécessaire plus tard. En

lançant ce calcul assez tôt, on réduit le temps qu'il faudra attendre ensuite, lorsqu'on aura vraiment besoin de son résultat.

Listing 5.12 : Utilisation de FutureTask pour précharger des données dont on aura besoin plus tard.

```
public class Preloader {  
    private final FutureTask<ProductInfo> future =  
        new FutureTask<ProductInfo>(new Callable<ProductInfo>() {  
            public ProductInfo call() throws DataLoadException {  
                return loadProductInfo ();  
            }  
        });  
    private final Thread thread = new Thread(future);  
  
    public void start() { thread.start(); }  
  
    public ProductInfo get()  
        throws DataLoadException , InterruptedException {  
        try {  
            return future.get();  
        } catch (ExecutionException e) {  
            Throwable cause = e.getCause();  
            if (cause instanceof DataLoadException)  
                throw (DataLoadException) cause;  
            else  
                throw launderThrowable(cause);  
        }  
    }  
}
```

La classe Preloader crée un objet FutureTask qui décrit la tâche consistant à charger les informations d'un produit à partir d'une base de données et un thread qui effectuera le calcul. Elle fournit une méthode start() pour lancer ce thread car il est déconseillé de le faire à partir d'un constructeur ou d'un initialisateur statique. Lorsque le programme aura plus tard besoin du ProductInfo, il pourra appeler get(), qui renvoie les données chargées si elles sont disponibles ou attend que le chargement soit terminé s'il ne l'est pas encore.

Les tâches décrisées par Callable peuvent lancer des exceptions contrôlées ou non contrôlées et n'importe quel code peut lancer une instance de Error. Tout ce que lance le code de la tâche est enveloppé dans un objet ExecutionException et relancé à partir de Future.get(). Cela complique le code qui appelle get(), non seulement parce qu'il doit gérer la possibilité d'une ExecutionException (et d'une CancellationException non contrôlée), mais aussi parce que la raison de l'ExecutionException est renvoyée sous forme de Throwable, ce qui est peu pratique à gérer.

Lorsque get() lance une ExecutionException dans Preloader, la cause appartiendra à l'une des trois catégories suivantes : une exception contrôlée lancée par l'objet Callable, une RuntimeException ou une Error. Nous devons traiter séparément ces trois cas mais nous utilisons la méthode auxiliaire launderThrowable() du Listing 5.13 pour encapsuler les parties les plus pénibles du code de gestion des exceptions. Avant d'appeler cette méthode, Preloader teste les exceptions contrôlées connues et les relance : il ne

reste donc plus que les exceptions non contrôlées, que Preloader traite en les transmettant à launderThrowable() et en lançant le résultat renvoyé par cet appel. Si l'objet Throwable passé à launderThrowable() est une Error, la méthode la relance directement ; si ce n'est pas une RuntimeException, elle lance une IllegalStateException pour indiquer une erreur de programmation. Il ne reste donc plus que RuntimeException, que launderThrowable() renvoie à l'appelant, qui, généralement, se contentera de la relancer.

Listing 5.13 : Coercition d'un objet Throwable non contrôlé en RuntimeException.

```
/** Si le Throwable est une Error, on le lance ; si c'est une
 * RuntimeException, on le renvoie ;
 * sinon, on lance IllegalStateException
 */
public static RuntimeException launderThrowable(Throwable t) {
    if (t instanceof RuntimeException)
        return (RuntimeException) t;
    else if (t instanceof Error)
        throw (Error) t;
    else
        throw new IllegalStateException ("Not unchecked", t);
}
```

5.5.3 Sémaphores

Les *sémaphores* servent à contrôler le nombre d'activités pouvant simultanément accéder à une ressource ou exécuter une certaine action [CPJ 3.4.1]. Les sémaphores permettent notamment d'implémenter des pools de ressources ou d'imposer une limite à une collection.

Un objet *Semaphore* gère un ensemble de *jetons* virtuels, dont le nombre initial est passé au constructeur. Les activités peuvent prendre des jetons avec *acquire()* (s'il en reste) et les rendre avec *release()* quand elles ont terminé¹. S'il n'y a plus de jeton disponible, *acquire()* se bloque jusqu'à ce qu'il y en ait un (ou jusqu'à ce qu'elle soit interrompue ou que le délai de l'opération ait expiré). Un *sémaphore binaire* est un sémaphore particulier puisque son nombre de jetons initial est égal à un. Un sémaphore binaire est souvent utilisé comme *mutex* pour fournir une sémantique de verrouillage non réentrant : celui qui détient le seul jeton détient le mutex.

Les sémaphores permettent d'implémenter des pools de ressources comme les pools de connexions aux bases de données. Bien qu'il soit relativement aisé de mettre en place un pool de taille fixe et de faire en sorte qu'une demande de ressource échoue s'il est

1. L'implémentation n'utilise pas de véritables objets jetons et *Semaphore* n'associe pas aux threads les jetons distribués : un jeton pris par un thread peut être rendu par un autre thread. Vous pouvez donc considérer *acquire()* comme une méthode qui consomme un jeton et *release()* comme une méthode qui en ajoute un ; un *Semaphore* n'est pas limité au nombre de jetons qui lui ont été affectés lors de sa création.

vide, on préfère que cette demande se bloque si le pool est vide et se débloque quand il ne l'est plus. En initialisant un objet `Semaphore` avec un nombre de jetons égal à la taille du pool, en prenant un jeton avant de tenter d'obtenir une ressource et en redonnant le jeton après avoir remis la ressource dans le pool, `acquire()` se bloquera jusqu'à ce que le pool ne soit plus vide. Cette technique est utilisée par la classe de tampon borné du Chapitre 12 (un moyen plus simple de construire un pool bloquant serait d'utiliser une `BlockingQueue` pour y stocker les ressources de ce pool).

De même, vous pouvez utiliser un `Semaphore` pour transformer n'importe quelle collection en collection bornée et bloquante, comme on le fait dans la classe `BoundedHashSet` du Listing 5.14. Le sémaphore est initialisé avec un nombre de jetons égal à la taille maximale désirée pour la collection et l'opération `add()` prend un jeton avant d'ajouter un élément à l'ensemble sous-jacent (si cette opération d'ajout sous-jacente n'ajoute rien, on redonne immédiatement le jeton). Inversement, une opération de suppression réussie redonne un jeton, permettant ainsi l'ajout d'autres éléments. L'implémentation `Set` sous-jacente ne sait rien de la limite fixée, qui est gérée par `BoundedHashSet`.

Listing 5.14 : Utilisation d'un `Semaphore` pour borner une collection.

```
public class BoundedHashSet<T> {  
    private final Set<T> set;  
    private final Semaphore sem;  
  
    public BoundedHashSet(int bound) {  
        this.set = Collections.synchronizedSet(new HashSet<T>());  
        sem = new Semaphore(bound);  
    }  
  
    public boolean add(T o) throws InterruptedException {  
        sem.acquire();  
        boolean wasAdded = false;  
        try {  
            wasAdded = set.add(o);  
            return wasAdded;  
        } finally {  
            if (!wasAdded)  
                sem.release();  
        }  
    }  
  
    public boolean remove(Object o) {  
        boolean wasRemoved = set.remove(o);  
        if (wasRemoved)  
            sem.release();  
        return wasRemoved;  
    }  
}
```

5.5.4 Barrières

Nous avons vu que les loquets facilitaient le démarrage ou l'attente de terminaison d'un groupe d'activités apparentées. Les loquets sont des objets *éphémères* : un loquet qui a atteint son état terminal ne peut plus être réinitialisé.

Les *barrières* ressemblent aux loquets car elles permettent de bloquer un groupe de threads jusqu'à ce qu'un événement survienne [CPJ 4.4.3], mais la différence essentielle est qu'avec une barrière tous les threads doivent arriver *en même temps* sur la barrière pour pouvoir s'exécuter. Les loquets attendent donc des *événements* tandis que les barrières attendent les *autres threads*. Une barrière implémente le protocole que certaines familles utilisent pour se donner rendez-vous : "Rendez-vous sur la place du Capitole à 18 heures ; attendez que tout le monde arrive et nous verrons où aller ensuite."

`CyclicBarrier` permet à un nombre donné de parties de se donner des rendez-vous répétés à un *point donné* ; cette classe peut être utilisée par les algorithmes parallèles itératifs qui décomposent un problème en un nombre fixe de sous-problèmes indépendants. Les threads appellent `await()` lorsqu'ils atteignent la barrière ; cet appel est bloquant tant que *tous* les threads n'ont pas atteint ce point. Lorsque tous les threads se sont rencontrés sur la barrière, celle-ci s'ouvre et tous les threads sont libérés. Elle se referme alors pour pouvoir être réutilisée. Si un appel à `await()` dépasse son temps d'expiration ou si un thread bloqué par `await()` est interrompu, la barrière est considérée comme *brisée* et tous les appels à `await()` en attente se terminent avec `BrokenBarrierException`. Si la barrière s'est ouverte correctement, `await()` renvoie un indice d'arrivée unique pour chaque thread, qui peut être utilisé pour "élire" un leader qui aura un rôle particulier lors de l'itération suivante. `CyclicBarrier` permet également de passer une *action de barrière* au constructeur, c'est-à-dire un `Runnable` qui sera exécuté (dans l'un des threads des sous-tâches) lorsque la barrière se sera ouverte, mais avant que les threads bloqués soient libérés.

On utilise souvent les barrières dans les simulations où le travail pour calculer une étape peut s'effectuer en parallèle mais que tout le travail associé à une étape donnée doit être terminé avant de passer à l'étape suivante. Dans les simulations de particules, par exemple, chaque étape calcule une nouvelle valeur pour la position de chaque particule en fonction des emplacements et des attributs des autres particules. Attendre sur une barrière entre chaque calcul garantit que toutes les modifications pour l'étape k se seront terminées avant de passer à l'étape $k + 1$.

La classe `CellularAutomata` du Listing 5.15 utilise une barrière pour effectuer une simulation de cellules, comme celle du jeu de la vie de Conway (Gardner, 1970). Lorsque l'on parallélise une simulation, il est généralement impossible d'affecter un thread par élément (une cellule, dans le cas du jeu de la vie) : cela nécessiterait un trop grand nombre de threads et le surcoût de leur coordination handicaperait les calculs. On choisit donc de partitionner le problème en un certain nombre de sous-parties, chacune étant prise en charge par un thread, et l'on fusionne ensuite les résultats. `CellularAutomata`

partitionne le damier en N_{cpu} parties, où N_{cpu} est le nombre de processeurs disponibles, et affecte chacune d'elles à un thread¹.

À chaque étape, les threads calculent les nouvelles valeurs pour toutes les cellules de leur partie du damier. Quand ils ont tous atteint la barrière, l'action de barrière applique ces nouvelles valeurs au modèle des données. Après cette action, les threads sont libérés pour calculer l'étape suivante, qui implique d'appeler la méthode `isDone()` pour savoir si d'autres itérations sont nécessaires.

Listing 5.15 : Coordination des calculs avec CyclicBarrier pour une simulation de cellules.

```

public class CellularAutomata {
    private final Board mainBoard;
    private final CyclicBarrier barrier;
    private final Worker[] workers;

    public CellularAutomata (Board board) {
        this.mainBoard = board;
        int count = Runtime.getRuntime().availableProcessors ();
        this.barrier = new CyclicBarrier (count, new Runnable() {
            public void run() {
                mainBoard.commitNewValues ();
            }
        });
        this.workers = new Worker[count];
        for (int i = 0; i < count; i++)
            workers[i] = new Worker(mainBoard.getSubBoard(count, i));
    }

    private class Worker implements Runnable {
        private final Board board;

        public Worker(Board board) { this.board = board; }
        public void run() {
            while (!board.hasConverged()) {
                for (int x = 0; x < board.getMaxX(); x++)
                    for (int y = 0; y < board.getMaxY(); y++)
                        board.setNewValue(x, y, computeValue (x, y));
                try {
                    barrier.await();
                } catch (InterruptedException ex) {
                    return;
                } catch (BrokenBarrierException ex) {
                    return;
                }
            }
        }
    }
}

public void start() {
    for (int i = 0; i < workers.length; i++)
        new Thread(workers[i]).start();
    mainBoard.waitForConvergence ();
}

```

1. Pour les traitements comme celui-ci, qui n'effectuent pas d'E/S et n'accèdent à aucune donnée partagée, N_{cpu} ou $N_{cpu}+1$ threads fourniront le débit optimal ; plus de threads n'amélioreront pas les performances et peuvent même les dégrader car ils entreront en compétition pour l'accès au processeur et à la mémoire.

Exchanger est une autre forme de barrière, formée de deux parties qui échangent des données [CPJ 3.4.3]. Les objets Exchanger sont utiles lorsque les parties effectuent des activités asymétriques : lorsqu'un thread remplit un tampon de données et que l'autre les lit, par exemple. Ces threads peuvent alors utiliser un Exchanger pour se rencontrer et échanger un tampon plein par un tampon vide. Lorsque deux threads échangent des données *via* un Exchanger, l'échange est une publication correcte des deux objets vers l'autre partie.

Le timing de l'échange dépend de la réactivité nécessaire pour l'application. L'approche la plus simple est que la tâche qui remplit échange lorsque le tampon est plein et que la tâche qui vide échange quand il est vide ; cela réduit au minimum le nombre d'échanges, mais peut retarder le traitement des données si la vitesse d'arrivée des nouvelles données n'est pas prévisible. Une autre approche serait que la tâche qui remplit échange lorsque le tampon est plein mais également lorsqu'il est partiellement rempli et qu'un certain temps s'est écoulé.

5.6 Construction d'un cache efficace et adaptable

Quasiment toutes les applications serveur utilisent une forme ou une autre de cache. La réutilisation des résultats d'un calcul précédent peut en effet réduire les temps d'attente et améliorer le débit, au prix d'un peu plus de mémoire utilisée.

Comme de nombreuses autres roues souvent réinventées, la mise en cache semble souvent plus simple qu'elle ne l'est en réalité. Une implémentation naïve d'un cache transformera sûrement un problème de performances en problème d'adaptabilité, même si elle améliore les performances d'une exécution monothread. Dans cette section, nous allons développer un cache efficace et adaptable pour y placer les résultats d'une fonction effectuant un calcul coûteux. Commençons par l'approche évidente – un simple `HashMap` – et examinons quelques-uns de ses inconvénients en terme de concurrence et comment y remédier.

L'interface `Computable <A, V>` du Listing 5.16 décrit une fonction ayant une entrée de type A et un résultat de type V. La classe `ExpensiveFunction`, qui implémente `Computable`, met longtemps à calculer son résultat et nous aimerais créer une enveloppe `Computable` qui mémorise les résultats des calculs précédents en encapsulant le processus de mise en cache (cette technique est appelée *mémoïzation*).

Listing 5.16 : Première tentative de cache, utilisant `HashMap` et la synchronisation.

```
public interface Computable<A, V> {  
    V compute(A arg) throws InterruptedException ;  
}  
  
public class ExpensiveFunction  
    implements Computable<String, BigInteger> {  
    public BigInteger compute(String arg) {
```



Listing 5.16 : Première tentative de cache, utilisant HashMap et la synchronisation. (suite)

```

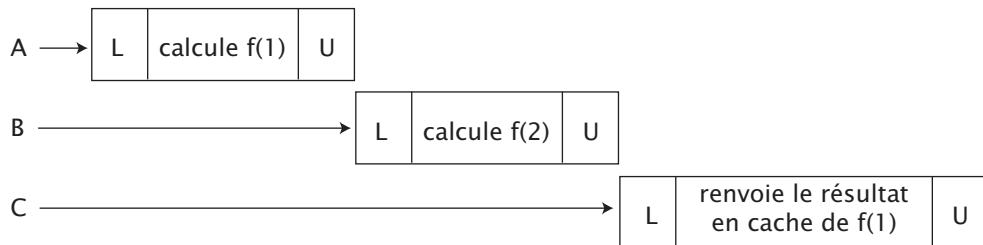
    // Après une longue réflexion...
    return new BigInteger(arg);
}
}

public class Memoizer1<A, V> implements Computable<A, V> {
    @GuardedBy("this")
    private final Map<A, V> cache = new HashMap<A, V>();
    private final Computable<A, V> c;

    public Memoizer1(Computable<A, V> c) {
        this.c = c;
    }

    public synchronized V compute(A arg) throws InterruptedException {
        V result = cache.get(arg);
        if (result == null) {
            result = c.compute(arg);
            cache.put(arg, result);
        }
        return result;
    }
}

```

**Figure 5.2**

Concurrence médiocre de `Memoizer1`.

La classe `Memoizer1` du Listing 5.16 montre notre première tentative : on utilise un `HashMap` pour stocker les résultats des calculs précédents. La méthode `compute()` commence par vérifier si le résultat souhaité se trouve déjà dans le cache, auquel cas elle renvoie la valeur déjà calculée. Sinon elle effectue le calcul et le stocke dans le `HashMap` avant de le renvoyer.

`HashMap` n'étant pas thread-safe, `Memoizer1` adopte l'approche prudente qui consiste à synchroniser *toute* la méthode `compute()` pour s'assurer que deux threads ne pourront pas accéder simultanément au `HashMap`. Cela garantit la thread safety mais a un effet évident sur l'adaptabilité puisqu'un seul thread peut exécuter `compute()` à la fois : si un autre thread est occupé à calculer un résultat, les autres threads appelant cette méthode peuvent donc être bloqués pendant un certain temps. Si plusieurs threads sont en attente pour calculer des valeurs qui n'ont pas encore été calculées, `compute()` peut, en fait,

mettre plus de temps à s'exécuter que si elle n'était pas mémoisée. La Figure 5.2 illustre ce qui pourrait se passer lorsque plusieurs threads tentent d'utiliser une fonction mémoisée de cette façon. Ce n'est pas ce genre d'amélioration des performances que nous espérions obtenir avec une mise en cache.

La classe Memoizer2 du Listing 5.17 améliore la concurrence désastreuse de Memoizer1 en remplaçant le HashMap par un ConcurrentHashMap. Cette classe étant thread-safe, il n'y a plus besoin de se synchroniser lorsque l'on accède au hachage sous-jacent, ce qui élimine la sérialisation induite par la synchronisation de compute() dans Memoizer1.

Listing 5.17 : Remplacement de HashMap par ConcurrentHashMap.

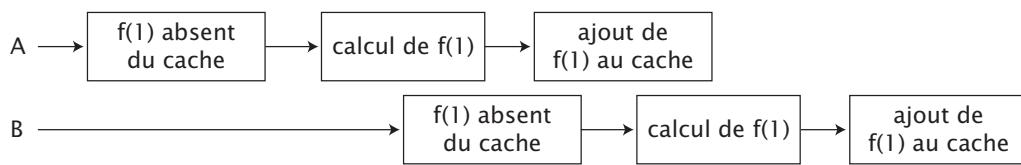
```
public class Memoizer2<A, V> implements Computable<A, V> {  
    private final Map<A, V> cache = new ConcurrentHashMap<A, V>();  
    private final Computable<A, V> c;  
  
    public Memoizer2(Computable<A, V> c) { this.c = c; }  
  
    public V compute(A arg) throws InterruptedException {  
        V result = cache.get(arg);  
        if (result == null) {  
            result = c.compute(arg);  
            cache.put(arg, result);  
        }  
        return result;  
    }  
}
```



Memoizer2 offre certainement une meilleure concurrence que Memoizer1 : plusieurs threads peuvent l'utiliser simultanément. Pourtant, elle souffre encore de quelques défauts : deux threads appelant compute() au même moment pourraient calculer la même valeur. Pour une mémoisation, c'est un comportement assez inefficace puisque le but d'un cache est justement d'empêcher de répéter plusieurs fois le même calcul. C'est encore pire pour un mécanisme de cache plus général ; pour un cache d'objet censé ne fournir qu'une et une seule initialisation, cette vulnérabilité pose également un risque de sécurité.

Le problème avec Memoizer2 est que, lorsqu'un thread lance un long calcul, les autres threads ne savent pas que ce calcul est en cours et peuvent donc lancer le même, comme le montre la Figure 5.3. Nous voudrions donc représenter le fait que "le thread X est en train de calculer f(27)" afin qu'un autre thread voulant calculer f(27) sache que le moyen le plus efficace d'obtenir ce résultat consiste à attendre que X ait fini et lui demande "qu'as-tu trouvé pour f(27) ?".

Nous avons déjà rencontré une classe qui fait exactement cela : FutureTask. On rappelle que cette classe représente une tâche de calcul qui peut, ou non, s'être déjà terminée et que FutureTask.get() renvoie immédiatement le résultat du calcul si celui-ci est disponible ou se bloque jusqu'à ce que le calcul soit terminé puis renvoie son résultat.

**Figure 5.3**

Deux threads calculant la même valeur avec Memoizer2.

La classe `Memoizer3` du Listing 5.18 redéfinit donc le hachage sous-jacent comme un `ConcurrentHashMap<A, Future<V>>` au lieu d'un `ConcurrentHashMap<A, V>`. Elle teste d'abord si le calcul approprié a été lancé (et non terminé comme dans `Memoizer2`). Si ce n'est pas le cas, elle crée un objet `FutureTask`, l'enregistre dans le hachage et lance le calcul ; sinon elle attend le résultat du calcul en cours. Ce résultat peut être disponible immédiatement ou en cours de calcul, mais tout cela est transparent pour celui qui appelle `Future.get()`.

Listing 5.18 : Enveloppe de mémoïsation utilisant FutureTask.

```

public class Memoizer3<A, V> implements Computable<A, V> {
    private final Map<A, Future<V>> cache
        = new ConcurrentHashMap<A, Future<V>>();
    private final Computable<A, V> c;

    public Memoizer3(Computable<A, V> c) { this.c = c; }

    public V compute(final A arg) throws InterruptedException {
        Future<V> f = cache.get(arg);
        if (f == null) {
            Callable<V> eval = new Callable<V>() {
                public V call() throws InterruptedException {
                    return c.compute(arg);
                }
            };
            FutureTask<V> ft = new FutureTask<V>(eval);
            f = ft;
            cache.put(arg, ft);
            ft.run(); // L'appel à c.compute() a lieu ici
        }
        try {
            return f.get();
        } catch (ExecutionException e) {
            throw launderThrowable(e.getCause());
        }
    }
}
  
```



L'implémentation de `Memoizer3` est presque parfaite : elle produit une bonne concurrence (grâce, essentiellement, à l'excellente concurrence de `ConcurrentHashMap`), le résultat est renvoyé de façon efficace s'il est déjà connu et, si le calcul est en cours dans un autre thread, les autres threads qui arrivent attendent patiemment le résultat. Elle n'a qu'un seul défaut : il reste un risque que deux threads puissent calculer la même valeur. Ce risque est bien moins élevé qu'avec `Memoizer2`, mais le bloc `if` de `compute()` étant

toujours une séquence *tester-puis-agir* non atomique, deux threads peuvent appeler `compute()` avec la même valeur à peu près en même temps, voir tous les deux que le cache ne contient pas la valeur désirée et donc lancer tous les deux le même calcul. Ce timing malheureux est illustré par la Figure 5.4.

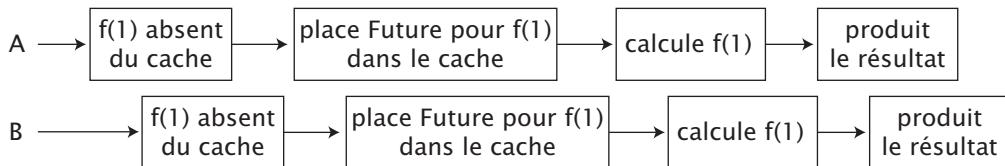


Figure 5.4

Timing malheureux forçant Memoizer3 à calculer deux fois la même valeur.

`Memoizer3` est vulnérable à ce problème parce qu'une action composée (*ajouter-si-absent*) effectuée sur le hachage sous-jacent ne peut pas être rendue atomique avec le verrouillage. La classe `Memoizer` du Listing 5.19 résout ce problème en tirant parti de la méthode atomique `putIfAbsent()` de `ConcurrentMap`.

Listing 5.19 : Implémentation finale de Memoizer.

```

public class Memoizer<A, V> implements Computable<A, V> {
    private final ConcurrentMap <A, Future<V>> cache
        = new ConcurrentHashMap <A, Future<V>>();
    private final Computable<A, V> c;

    public Memoizer(Computable<A, V> c) { this.c = c; }

    public V compute(final A arg) throws InterruptedException {
        while (true) {
            Future<V> f = cache.get(arg);
            if (f == null) {
                Callable<V> eval = new Callable<V>() {
                    public V call() throws InterruptedException {
                        return c.compute(arg);
                    }
                };
                FutureTask<V> ft = new FutureTask<V>(eval);
                f = cache.putIfAbsent(arg, ft);
                if (f == null) { f = ft; ft.run(); }
            }
            try {
                return f.get();
            } catch (CancellationException e) {
                cache.remove(arg, f);
            } catch (ExecutionException e) {
                throw launderThrowable(e.getCause());
            }
        }
    }
}
  
```

Mettre en cache un objet Future au lieu d'une valeur fait courir le risque d'une *pollution du cache* : si un calcul est annulé ou échoue, les futures tentatives de calculer ce résultat indiqueront également une annulation ou une erreur. C'est pour éviter ce problème que Memoizer supprime l'objet Future du cache s'il s'aperçoit que le calcul a été annulé ; il pourrait également être judicieux de faire de même si l'on détecte une RuntimeException et que le calcul peut réussir lors d'une tentative extérieure. Memoizer ne gère pas non plus l'expiration du cache, mais cela pourrait se faire en utilisant une sous-classe de Future Task associant une date d'expiration à chaque résultat et en parcourant périodiquement le cache pour rechercher les entrées expirées (de même, on ne traite pas l'éviction du cache consistant à supprimer les anciennes entrées afin de libérer de la place pour les nouvelles et ainsi empêcher que le cache ne consomme trop d'espace mémoire).

Avec cette implémentation concurrente d'un cache désormais complète, nous pouvons maintenant ajouter un véritable cache à la servlet de factorisation du Chapitre 2, comme nous l'avions promis. La classe Factorizer du Listing 5.20 utilise Memoizer pour mettre en cache de façon efficace et adaptative les valeurs déjà calculées.

Listing 5.20 : Servlet de factorisation mettant en cache ses résultats avec Memoizer.

```
@ThreadSafe
public class Factorizer implements Servlet {
    private final Computable<BigInteger, BigInteger[]> c =
        new Computable<BigInteger, BigInteger[]>() {
            public BigInteger[] compute(BigInteger arg) {
                return factor(arg);
            }
        };
    private final Computable<BigInteger, BigInteger[]> cache
        = new Memoizer<BigInteger, BigInteger[]>(c);

    public void service(ServletRequest req, ServletResponse resp) {
        try {
            BigInteger i = extractFromRequest (req);
            encodeIntoResponse (resp, cache.compute(i));
        } catch (InterruptedException e) {
            encodeError(resp, "factorization interrupted");
        }
    }
}
```

Résumé de la première partie

Nous avons déjà présenté beaucoup de choses ! *L'antisèche sur la concurrence* qui suit résume les concepts principaux et les règles essentielles présentées dans cette première partie.

- *C'est l'état modifiable, idiot¹.*

Tous les problèmes de concurrence se ramènent à une coordination des accès à l'état modifiable. Moins l'état est modifiable, plus il est facile d'assurer la thread safety.

- *Créez des champs final sauf s'ils ont besoin d'être modifiables.*

- *Les objets non modifiables sont automatiquement thread-safe.*

Les objets non modifiables simplifient énormément la programmation concurrente.

Ils sont plus simples et plus sûrs et peuvent être partagés librement sans nécessiter de verrous ni de copies défensives.

- *L'encapsulation permet de gérer la complexité.*

Vous pourriez écrire un programme thread-safe avec toutes les données dans des variables globales, mais pourquoi le faire ? L'encapsulation des données dans des objets facilite la préservation de leurs invariants ; l'encapsulation de la synchronisation dans des objets facilite le respect de leur politique de synchronisation.

- *Protégez chaque variable modifiable par un verrou.*

- *Protégez toutes les variables d'un invariant par le même verrou.*

- *Gardez les verrous pendant l'exécution des actions composées.*

- *Un programme qui accède à une variable modifiable à partir de plusieurs threads et sans synchronisation est un programme faux.*

- *Ne croyez pas les raisonnements subtils qui vous expliquent pourquoi vous n'avez pas besoin de synchroniser.*

- *Ajoutez la thread safety à la phase de conception, ou indiquez explicitement que votre classe n'est pas thread-safe.*

- *Documentez votre politique de synchronisation.*

1. En 1992, James Carville, l'un des stratèges de la victoire de Bill Clinton, avait affiché au QG de campagne un pense-bête devenu légendaire, “It's the economy, stupid!”, pour insister sur ce message lors de la campagne.

II

Structuration des applications concurrentes

6

Exécution des tâches

La plupart des applications concurrentes sont organisées autour de l'exécution de *tâches*, que l'on peut considérer comme des unités de travail abstraites. Diviser une application en plusieurs tâches simplifie l'organisation du programme et facilite la découverte des erreurs grâce aux frontières naturelles séparant les différentes transactions. Cette division encourage également la concurrence en fournissant une structure naturelle permettant de paralléliser le travail.

6.1 Exécution des tâches dans les threads

La première étape pour organiser un programme autour de l'exécution de tâches consiste à identifier les *frontières* entre ces tâches. Dans l'idéal, les tâches sont des activités *indépendantes*, c'est-à-dire des opérations qui ne dépendent ni de l'état, ni du résultat, ni des effets de bord des autres tâches. Cette indépendance facilite la concurrence puisque les tâches indépendantes peuvent s'exécuter en parallèle si l'on dispose des ressources de traitement adéquates. Pour disposer de plus de souplesse dans l'ordonnancement et la répartition de la charge entre ces tâches, chacune devrait également représenter une petite fraction des possibilités du traitement de l'application.

Les applications serveur doivent fournir un *bon débit de données* et une *réactivité correcte* sous une charge normale. Les fournisseurs d'applications veulent des programmes permettant de supporter autant d'utilisateurs que possible afin de réduire d'autant les coûts par utilisateur ; ces utilisateurs veulent évidemment obtenir rapidement les réponses qu'ils demandent. En outre, les applications doivent non pas se dégrader brutalement lorsqu'elles sont surchargées mais *réagir le mieux possible*. Tous ces objectifs peuvent être atteints en choisissant de bonnes frontières entre les tâches et en utilisant une *politique raisonnable d'exécution* des tâches (voir la section 6.2.2).

La plupart des applications serveur offrent un choix naturel pour les frontières entre tâches : les différentes requêtes des clients. Les serveurs web, de courrier, de fichiers,

les conteneurs EJB et les serveurs de bases de données reçoivent tous des requêtes de clients distants *via* des connexions réseau. Utiliser ces différentes requêtes comme des frontières de tâches permet généralement d'obtenir à la fois des tâches indépendantes et de taille appropriée. Le résultat de la soumission d'un message à un serveur de courrier, par exemple, n'est pas affecté par les autres messages qui sont traités en même temps, et la prise en charge d'un simple message ne nécessite généralement qu'un très petit pourcentage de la capacité totale du serveur.

6.1.1 Exécution séquentielle des tâches

Il existe un certain nombre de politiques possibles pour ordonner les tâches au sein d'une application ; parmi elles, certaines exploitent mieux la concurrence que d'autres. La plus simple consiste à exécuter les tâches séquentiellement dans un seul thread. La classe `SingleThreadWebServer` du Listing 6.1 traite ses tâches – des requêtes HTTP arrivant sur le port 80 – en séquence. Les détails du traitement de la requête ne sont pas importants ; nous ne nous intéressons ici qu'à la concurrence des différentes politiques d'ordonnancement.

Listing 6.1 : Serveur web séquentiel.

```
class SingleThreadWebServer {  
    public static void main(String[] args) throws IOException {  
        ServerSocket socket = new ServerSocket(80);  
        while (true) {  
            Socket connection = socket.accept();  
            handleRequest(connection);  
        }  
    }  
}
```



`SingleThreadedWebServer` est simple et correcte d'un point de vue théorique, mais serait très inefficace en production car elle ne peut gérer qu'une seule requête à la fois. Le thread principal alterne constamment entre accepter des connexions et traiter la requête associée : pendant que le serveur traite une requête, les nouvelles connexions doivent attendre qu'il ait fini ce traitement et qu'il appelle à nouveau `accept()`. Cela peut fonctionner si le traitement des requêtes est suffisamment rapide pour que `handleRequest()` se termine immédiatement, mais cette hypothèse ne reflète pas du tout la situation des serveurs web actuels.

Traiter une requête web implique un mélange de calculs et d'opérations d'E/S. Le serveur doit lire dans un socket pour obtenir la requête et y écrire pour envoyer la réponse ; ces opérations peuvent être bloquantes en cas de congestion du réseau ou de problèmes de connexion. Il peut également effectuer des E/S sur des fichiers ou lancer des requêtes de bases de données, qui peuvent elles aussi être bloquantes. Avec un serveur monothread, un blocage ne fait pas que retarder le traitement de la requête en cours : il empêche également celui des requêtes en attente. Si une requête se bloque pendant un temps très long, les utilisateurs penseront que le serveur n'est plus disponible

puisque l'il ne semble plus répondre. En outre, les ressources sont mal utilisées puisque le processeur reste inactif pendant que l'unique thread attend que ses E/S se terminent.

Pour les applications serveur, le traitement séquentiel fournit rarement un bon débit ou une réactivité correcte. Il existe des exceptions – lorsqu'il y a très peu de tâches et qu'elles durent longtemps, ou quand le serveur ne sert qu'un seul client qui n'envoie qu'une seule requête à la fois – mais la plupart des applications serveur ne fonctionnent pas de cette façon¹.

6.1.2 Création explicite de threads pour les tâches

Une approche plus réactive consiste à créer un nouveau thread pour répondre à chaque nouvelle requête, comme dans le Listing 6.2.

Listing 6.2 : Serveur web lançant un thread par requête.

```
class ThreadPerTaskWebServer {  
    public static void main(String[] args) throws IOException {  
        ServerSocket socket = new ServerSocket(80);  
        while (true) {  
            final Socket connection = socket.accept();  
            Runnable task = new Runnable() {  
                public void run() {  
                    handleRequest(connection);  
                }  
            };  
            new Thread(task).start();  
        }  
    }  
}
```



La structure de la classe ThreadPerTaskWebServer ressemble à celle de la version monothread – le thread principal continue d'alterner entre la réception d'une connexion entrante et le traitement de la requête. Mais, ici, la boucle principale crée un nouveau thread pour chaque connexion afin de traiter la requête au lieu de le faire dans le thread principal. Ceci a trois conséquences importantes :

- Le thread principal se décharge du traitement de la tâche, ce qui permet à la boucle principale de poursuivre et de venir attendre plus rapidement la connexion entrante suivante. Ceci autorise de nouvelles connexions alors que les requêtes sont en cours de traitement, ce qui améliore les temps de réponse.
- Les tâches peuvent être traitées en parallèle, ce qui permet de traiter simultanément plusieurs requêtes. Ceci améliore le débit lorsqu'il y a plusieurs processeurs ou si des tâches doivent se bloquer en attente d'une opération d'E/S, de l'acquisition d'un verrou ou de la disponibilité d'une ressource, par exemple.

1. Dans certaines situations, le traitement séquentiel offre des avantages en termes de simplicité et de sécurité ; la plupart des interfaces graphiques traitent séquentiellement les tâches dans un seul thread. Nous reviendrons sur le modèle séquentiel au Chapitre 9.

- Le code du traitement de la tâche doit être *thread-safe* car il peut être invoqué de façon concurrente par plusieurs tâches.

En cas de charge modérée, cette approche est plus efficace qu'une exécution séquentielle. Tant que la fréquence d'arrivée des requêtes ne dépasse pas les capacités de traitement du serveur, elle offre une meilleure réactivité et un débit supérieur.

6.1.3 Inconvénients d'une création illimitée de threads

Cependant, dans un environnement de production, l'approche "un thread par tâche" a quelques inconvénients pratiques, notamment lorsqu'elle peut produire un grand nombre de threads :

- **Surcoût dû au cycle de vie des threads.** La création d'un thread et sa suppression ne sont pas gratuites. Bien que le surcoût dépende des plates-formes, la création d'un thread prend du temps, induit une certaine latence dans le traitement de la requête et nécessite un traitement de la part de la JVM et du système d'exploitation. Si les requêtes sont fréquentes et légères, comme dans la plupart des applications serveur, la création d'un thread par requête peut consommer un nombre non négligeable de ressources.
- **Consommation des ressources.** Les threads actifs consomment des ressources du système, notamment la mémoire. S'il y a plus de threads en cours d'exécution qu'il n'y a de processeurs disponibles, certains threads resteront inactifs, ce qui peut consommer beaucoup de mémoire et surcharger le ramasse-miettes. En outre, le fait que de nombreux threads concourent pour l'accès aux processeurs peut également avoir des répercussions sur les performances. Si vous avez suffisamment de threads pour garder tous les processeurs occupés, en créer plus n'améliorera rien, voire dégradera les performances.
- **Stabilité.** Il y a une limite sur le nombre de threads qui peuvent être créés. Cette limite varie en fonction des plates-formes, des paramètres d'appels de la JVM, de la taille de pile demandée dans le constructeur de Thread et des limites imposées aux threads par le système d'exploitation¹. Lorsque vous atteignez cette limite, vous obtiendrez très probablement une exception OutOfMemoryError. Tenter de se rétablir de cette erreur est très risqué ; il est bien plus simple de structurer votre programme afin d'éviter d'atteindre cette limite.

1. Sur des machines 32 bits, un facteur limitant important est l'espace d'adressage pour les piles des threads. Chaque thread gère deux piles d'exécution : une pour le code Java, l'autre pour le code natif. Généralement, la JVM produit par défaut une taille de pile combinée d'environ 512 kilo-octets (vous pouvez changer cette valeur avec le paramètre -xss de la JVM ou lors de l'appel du constructeur de Thread). Si vous divisez les 2^{32} adresses par la taille de la pile de chaque thread, vous obtenez une limite de quelques milliers ou dizaines de milliers de threads. D'autres facteurs, comme les limites du système d'exploitation, peuvent imposer des limites plus contraignantes.

Jusqu'à un certain point, ajouter plus de threads permet d'améliorer le débit mais, au-delà de ce point, créer des threads supplémentaires ne fera que ralentir, et en créer un de trop peut totalement empêcher l'application de fonctionner. Le meilleur moyen de se protéger de ce danger consiste à fixer une limite au nombre de threads qu'un programme peut créer et à tester sérieusement l'application pour vérifier qu'elle ne tombera pas à court de ressources, même lorsque cette limite sera atteinte.

Le problème de l'approche "un thread par tâche" est que la seule limite imposée au nombre de threads créés est la fréquence à laquelle les clients distants peuvent lancer des requêtes HTTP. Comme tous les autres problèmes liés à la concurrence, la création infinie de threads peut *sembler* fonctionner parfaitement au cours des phases de prototypage et de développement, ce qui n'empêchera pas le problème d'apparaître lorsque l'application sera déployée en production et soumise à une forte charge.

Un utilisateur pervers, voire des utilisateurs ordinaires, peut donc faire planter votre serveur web en le soumettant à une charge trop forte. Pour une application serveur supposée fournir une haute disponibilité et se dégrader correctement en cas de charge importante, il s'agit d'un sérieux défaut.

6.2 Le framework Executor

Les tâches sont des unités logiques de travail et les threads sont un mécanisme grâce auquel ces tâches peuvent s'exécuter de façon asynchrone. Nous avons étudié deux politiques d'exécution des tâches à l'aide des threads – l'exécution séquentielle des tâches dans un seul thread et l'exécution de chaque tâche dans son propre thread. Toutes les deux ont de sévères limitations : l'approche séquentielle implique une mauvaise réactivité et un faible débit et l'approche "une tâche par thread" souffre d'une mauvaise gestion des ressources.

Au Chapitre 5, nous avons vu comment utiliser des *files de taille fixe* pour empêcher qu'une application surchargée ne soit à court de mémoire. Un *pool de threads* offrant les mêmes avantages pour la gestion des threads, `java.util.concurrent` en fournit une implémentation dans le cadre du framework Executor. Comme le montre le Listing 6.3, la principale abstraction de l'exécution des tâches dans la bibliothèque des classes Java est *non pas Thread mais Executor*.

Listing 6.3 : Interface Executor.

```
public interface Executor {  
    void execute(Runnable command);  
}
```

Executor est peut-être une interface simple, mais elle forme les fondements d'un framework souple et puissant pour l'exécution asynchrone des tâches sous un grand nombre de politiques d'exécution des tâches. Elle fournit un moyen standard pour

découpler *la soumission des tâches* de leur *exécution* en les décrivant comme des objets Runnable. Les implémentations de Executor fournissent également un contrôle du cycle de vie et des points d’ancrage permettant d’ajouter une collecte des statistiques, ainsi qu’une gestion et une surveillance des applications.

Executor repose sur le patron producteur-consommateur, où les activités qui soumettent des tâches sont les producteurs (qui produisent le travail à faire) et les threads qui exécutent ces tâches sont les consommateurs (qui consomment ce travail). *L'utilisation d'un Executor est, généralement, le moyen le plus simple d'implémenter une conception de type producteur-consommateur dans une application.*

6.2.1 Exemple : serveur web utilisant Executor

La création d’un serveur web à partir d’un Executor est très simple. La classe TaskExecutionWebServer du Listing 6.4 remplace la création des threads, qui était codée en dur dans la version précédente, par un Executor. Ici, nous utilisons l’une de ses implémentations standard, un pool de threads de taille fixe, avec 100 threads.

Dans TaskExecutionWebServer, la soumission de la tâche de gestion d’une requête est séparée de son exécution grâce à un Executor et son comportement peut être modifié en utilisant simplement une autre implémentation de Executor. Le changement d’implémentation ou de configuration d’Executor est une opération bien moins lourde que modifier la façon dont les tâches sont soumises ; généralement, la configuration est un événement unique qui peut aisément être présenté lors du déploiement de l’application, alors que le code de soumission des tâches a tendance à être disséminé un peu partout dans le programme et à être plus difficile à mettre en évidence.

Listing 6.4 : Serveur web utilisant un pool de threads.

```
class TaskExecutionWebServer {  
    private static final int NTHREADS = 100;  
    private static final Executor exec  
        = Executors.newFixedThreadPool(NTHREADS);  
  
    public static void main(String[] args) throws IOException {  
        ServerSocket socket = new ServerSocket(80);  
        while (true) {  
            final Socket connection = socket.accept();  
            Runnable task = new Runnable() {  
                public void run() {  
                    handleRequest(connection);  
                }  
            };  
            exec.execute(task);  
        }  
    }  
}
```

TaskExecutionWebServer peut être facilement modifié pour qu'il se comporte comme ThreadPerTaskWebServer : comme le montre le Listing 6.5, il suffit d'utiliser un Executor qui crée un nouveau thread pour chaque requête, ce qui est très simple.

Listing 6.5 : Executor lançant un nouveau thread pour chaque tâche.

```
public class ThreadPerTaskExecutor implements Executor {
    public void execute(Runnable r) {
        new Thread(r).start();
    }
}
```

De même, il est tout aussi simple d'écrire un Executor pour que TaskExecutionWeb Server se comporte comme la version monothread, en exécutant chaque tâche de façon synchrone dans `execute()`, comme le montre la classe `WithinThreadExecutor` du Listing 6.6.

Listing 6.6 : Executor exécutant les tâches de façon synchrone dans le thread appelant.

```
public class WithinThreadExecutor implements Executor {
    public void execute(Runnable r) {
        r.run();
    }
}
```

6.2.2 Politiques d'exécution

L'intérêt de séparer la soumission de l'exécution est que cela permet de spécifier facilement, et donc de modifier simplement, la politique d'exécution d'une classe de tâches. Une politique d'exécution répond aux questions "quoi, où, quand et comment" concernant l'exécution des tâches :

- Dans quel thread les tâches s'exécuteront-elles ?
- Dans quel ordre les tâches devront-elles s'exécuter (FIFO, LIFO, selon leurs priorités) ?
- Combien de tâches peuvent s'exécuter simultanément ?
- Combien de tâches peuvent être mises en attente d'exécution ?
- Si une tâche doit être rejetée parce que le système est surchargé, quelle sera la victime et comment l'application sera-t-elle prévenue ?
- Quelles actions faut-il faire avant ou après l'exécution d'une tâche ?

Les politiques d'exécution sont un outil de gestion des ressources et la politique optimale dépend des ressources de calcul disponibles et de la qualité du service recherchée. En limitant le nombre de tâches simultanées, vous pouvez garantir que l'application n'échouera pas si les ressources sont épuisées et que ses performances ne souffriront pas de problèmes dus à la concurrence pour des ressources en quantités limitées¹.

1. Ceci est analogue à l'un des rôles d'un moniteur de transactions dans une application d'entreprise ; il peut contrôler la fréquence à laquelle les transactions peuvent être traitées, afin de ne pas épuiser des ressources limitées.

Séparer la spécification de la politique d'exécution de la soumission des tâches permet de choisir une politique d'exécution lors du déploiement adaptée au matériel disponible.

À chaque fois que vous voyez un code de la forme :

```
new Thread(runnable).start()
```

et que vous pensez avoir besoin d'une politique d'exécution plus souple, réfléchissez sérieusement à son remplacement par l'utilisation d'un Executor.

6.2.3 Pools de threads

Un pool de threads gère un ensemble homogène de threads travailleurs. Il est étroitement lié à une *file* contenant les tâches en attente d'exécution. La vie des threads travailleurs est simple : demander la tâche suivante dans la file, l'exécuter et revenir attendre une autre tâche.

L'exécution des tâches avec un pool de threads présente un certain nombre d'avantages par rapport à l'approche "un thread par tâche". La réutilisation d'un thread existant au lieu d'en créer un nouveau amortit les coûts de création et de suppression des threads en les répartissant sur plusieurs requêtes. En outre, le thread travailleur existant souvent déjà lorsque la requête arrive, le temps de latence associé à la création du thread ne retarde pas l'exécution de la tâche, d'où une réactivité accrue. En choisissant soigneusement la taille du pool, vous pouvez avoir suffisamment de threads pour occuper les processeurs tout en évitant que l'application soit à court de mémoire ou se plante à cause d'une trop forte compétition pour les ressources.

La bibliothèque standard fournit une implémentation flexible des pools de threads, ainsi que quelques configurations prédéfinies assez utiles. Pour créer un pool, vous pouvez appeler l'une des méthodes fabriques statiques de la classe Executors :

- **newFixedThreadPool()**. Crée un pool de taille fixe qui crée les threads à mesure que les tâches sont soumises jusqu'à atteindre la taille maximale du pool, puis qui tente de garder constante la taille de ce pool (en créant un nouveau thread lorsqu'un thread meurt à cause d'une Exception inattendue).
- **newCachedThreadPool()**. Crée un pool de threads en cache, ce qui donne plus de souplesse pour supprimer les threads inactifs lorsque la taille courante du pool dépasse la demande de traitement et pour ajouter de nouveaux threads lorsque cette demande augmente, tout en ne fixant pas de limite à la taille du pool.
- **newSingleThreadExecutor()**. Crée une instance Executor monothread qui ne produit qu'un seul thread travailleur pour traiter les tâches, en le remplaçant s'il meurt

accidentellement. Les tâches sont traitées séquentiellement selon l'ordre imposé par la file d'attente des tâches (FIFO, LIFO, ordre des priorités)¹.

- **newScheduledThreadPool()**. Crée un pool de threads de taille fixe, permettant de différer ou de répéter l'exécution des tâches, comme `Timer` (voir la section 6.2.5).

Les fabriques `newFixedThreadPool()` et `newCachedThreadPool()` renvoient des instances de la classe générale `ThreadPoolExecutor` qui peuvent également servir directement à construire des "exécuteurs" plus spécialisés. Nous présenterons plus en détail les options de configuration des pools de threads au Chapitre 8.

Le serveur web de `TaskExecutionWebServer` utilise un `Executor` avec un pool limité de threads travailleurs. Soumettre une tâche avec `execute()` l'ajoute à la file d'attente dans laquelle les threads viennent sans cesse chercher des tâches pour les exécuter.

Passer d'une politique "un thread par tâche" à une politique utilisant un pool a un effet non négligeable sur la stabilité de l'application : le serveur web ne souffrira plus lorsqu'il sera soumis à une charge importante². En outre, son comportement se dégradera moins violemment puisqu'il ne crée pas des milliers de threads qui combattent pour des ressources processeur et mémoire limitées. Enfin, l'utilisation d'un `Executor` ouvre la porte à toutes sortes d'opportunités de configuration, de gestion, de surveillance, de journalisation, de suivi des erreurs et autres possibilités qui sont bien plus difficiles à ajouter sans un framework d'exécution des tâches.

6.2.4 Cycle de vie d'un Executor

Nous avons vu comment créer un `Executor` mais pas comment l'arrêter. Une implémentation de `Executor` crée des threads pour traiter des tâches mais, la JVM ne pouvant pas se terminer tant que tous les threads (non démons) ne se sont pas terminés, ne pas arrêter un `Executor` empêche l'arrêt de la JVM.

Un `Executor` traitant les tâches de façon asynchrone, l'état à un instant donné des tâches soumises n'est pas évident. Certaines se sont peut-être terminées, certaines peuvent être en cours d'exécution et d'autres peuvent être en attente d'exécution. Pour arrêter une application, il y a une marge entre un arrêt en douceur (finir ce qui a été lancé et ne pas accepter de nouveau travail) et un arrêt brutal (éteindre la machine), avec

1. Les instances `Executor` monothreads fournissent également une synchronisation interne suffisante pour garantir que toute écriture en mémoire par les tâches sera visible par les tâches suivantes ; ceci signifie que les objets peuvent être confinés en toute sécurité au "thread tâche", même si ce thread est remplacé épisodiquement par un autre.

2. Même s'il ne souffrira plus à cause de la création d'un nombre excessif de threads, il peut quand même (bien que ce soit plus difficile) arriver à court de mémoire si la fréquence d'arrivée des tâches est supérieure à celle de leur traitement pendant une période suffisamment longue, à cause de l'augmentation de la taille de la file des `Runnable` en attente d'exécution. Avec le framework `Executor`, ce problème peut se résoudre en utilisant une file d'attente de taille fixe – voir la section 8.3.2.

plusieurs degrés entre les deux. Les Executor fournissant un service aux applications, ils devraient également pouvoir être arrêtés, en douceur et brutalement, et renvoyer des informations à l'application sur l'état des tâches affectées par cet arrêt.

Pour résoudre le problème du cycle de vie du service d'exécution, l'interface ExecutorService étend Executor en lui ajoutant un certain nombre de méthodes dédiées à la gestion du cycle de vie, présentées dans le Listing 6.7 (elle ajoute également certaines méthodes utilitaires pour la soumission des tâches).

Listing 6.7 : Méthodes de ExecutorService pour le cycle de vie.

```
public interface ExecutorService extends Executor {  
    void shutdown();  
    List<Runnable> shutdownNow();  
    boolean isShutdown();  
    boolean isTerminated();  
    boolean awaitTermination (long timeout, TimeUnit unit)  
        throws InterruptedException;  
    // ... méthodes utilitaires pour la soumission des tâches  
}
```

Le cycle de vie qu'implique ExecutorService a trois états : *en cours d'exécution*, *en cours d'arrêt* et *terminé*. Les objets ExecutorService sont initialement créés dans l'état *en cours d'exécution*. La méthode `shutdown()` lance un arrêt en douceur : aucune nouvelle tâche n'est acceptée, mais les tâches déjà soumises sont autorisées à se terminer – même celles qui n'ont pas encore commencé leur exécution. La méthode `shutdownNow()` lance un arrêt brutal : elle tente d'annuler les tâches en attente et ne lance aucune des tâches qui sont dans la file et qui n'ont pas commencé.

Les tâches soumises à un ExecutorService après son arrêt sont gérées par le *gestionnaire d'exécution rejetée* (voir la section 8.3.3), qui peut supprimer la tâche sans prévenir ou forcer `execute()` à lancer l'exception non contrôlée `RejectedExecutionException`. Lorsque toutes les tâches se sont terminées, l'objet ExecutorService passe dans l'état *terminé*. Vous pouvez attendre qu'il atteigne cet état en appelant la méthode `awaitTermination()` ou en l'interrogeant avec `isTerminated()` pour savoir s'il est terminé. En général, on fait suivre immédiatement l'appel à `shutdown()` par `awaitTermination()`, afin d'obtenir l'effet d'un arrêt synchrone de ExecutorService (l'arrêt de Executor et l'annulation de tâche sont présentés plus en détail au Chapitre 7).

La classe `LifecycleWebServer` du Listing 6.8 ajoute un cycle de vie à notre serveur web. Ce dernier peut désormais être arrêté de deux façons : par programme en appelant `stop()` ou *via* une requête client en envoyant au serveur une requête HTTP respectant un certain format.

Listing 6.8 : Serveur web avec cycle de vie.

```
class LifecycleWebServer {  
    private final ExecutorService exec = ...;  
  
    public void start() throws IOException {
```

```

ServerSocket socket = new ServerSocket(80);
while (!exec.isShutdown()) {
    try {
        final Socket conn = socket.accept();
        exec.execute(new Runnable() {
            public void run() { handleRequest(conn); }
        });
    } catch (RejectedExecutionException e) {
        if (!exec.isShutdown())
            log("task submission rejected", e);
    }
}
}

public void stop() { exec.shutdown(); }

void handleRequest (Socket connection) {
    Request req = readRequest(connection);
    if (isShutdownRequest(req))
        stop();
    else
        dispatchRequest(req);
}
}

```

6.2.5 Tâches différées et périodiques

La classe utilitaire `Timer` gère l'exécution des tâches différées ("lancer cette tâche dans 100 ms") et périodiques ("lancer cette tâche toutes les 10 ms"). Cependant, elle a quelques inconvénients et il est préférable d'utiliser `ScheduledThreadPoolExecutor` à la place¹. Pour construire un objet `ScheduledThreadPoolExecutor`, on peut utiliser son constructeur ou la méthode fabrique `newScheduledThreadPool()`.

Un `Timer` ne crée qu'un seul thread pour exécuter les tâches qui lui sont confiées. Si l'une de ces tâches met trop de temps à s'exécuter, cela peut perturber la précision du timing des autres `TimerTask`.

Si une tâche `TimerTask` est planifiée pour s'exécuter toutes les 10 ms et qu'une autre `TimerTask` met 40 ms pour terminer son exécution, par exemple, la tâche récurrente sera soit appelée quatre fois de suite après la fin de la tâche longue soit "manquera" totalement quatre appels (selon qu'elle a été planifiée pour une fréquence donnée ou pour un délai fixé). Les pools de thread résolvent cette limitation en permettant d'utiliser plusieurs threads pour exécuter des tâches différées et planifiées.

Un autre problème de `Timer` est son comportement médiocre lorsqu'une `TimerTask` lance une exception non contrôlée : le thread `Timer` ne capturant pas l'exception, une exception non contrôlée lancée par une `TimerTask` met fin au planificateur. En outre, dans cette situation, `Timer` ne ressuscite pas le thread : il suppose à tort que l'objet `Timer` tout entier a été annulé. En ce cas, les `TimerTasks` déjà planifiées mais pas encore

1. `Timer` ne permet d'ordonnancer les tâches que de façon absolue, pas relative, ce qui les rend dépendantes des modifications de l'horloge système ; `ScheduledThreadPoolExecutor` n'utilise qu'un temps relatif.

exécutées ne seront jamais lancées et les nouvelles tâches ne pourront pas être planifiées (ce problème, appelé "fuite de thread", est décrit dans la section 7.3, en même temps que les techniques permettant de l'éviter).

La classe `OutOfTime` du Listing 6.9 illustre la façon dont un `Timer` peut être perturbé de cette manière et, comme un problème ne vient jamais seul, comment l'objet `Timer` partage cette confusion avec le malheureux client suivant qui tente de soumettre une nouvelle `TimerTask`. Vous pourriez vous attendre à ce que ce programme s'exécute pendant 6 secondes avant de se terminer alors qu'en fait il se terminera après 1 seconde avec une exception `IllegalStateException` associée au message "Timer already cancelled". `ScheduledThreadPoolExecutor` sachant correctement gérer ces tâches qui se comportent mal, il y a peu de raisons d'utiliser `Timer` à partir de Java 5.0.

Listing 6.9 : Classe illustrant le comportement confus de Timer.

```
public class OutOfTime {  
    public static void main(String[] args) throws Exception {  
        Timer timer = new Timer();  
        timer.schedule(new ThrowTask(), 1);  
        SECONDS.sleep(1);  
        timer.schedule(new ThrowTask(), 1);  
        SECONDS.sleep(5);  
    }  
  
    static class ThrowTask extends TimerTask {  
        public void run() { throw new RuntimeException(); }  
    }  
}
```



Si vous devez mettre en place un service de planification, vous pouvez quand même tirer parti de la bibliothèque standard en utilisant `DelayQueue`, une implémentation de `BlockingQueue` fournissant les fonctionnalités de planification de `ScheduledThreadPoolExecutor`. Un objet `DelayQueue` gère une collection d'objets `Delayed`, associés à un délai : `DelayQueue` ne vous autorise à prendre un élément que si son délai a expiré. Les objets sortent d'une `DelayQueue` dans l'ordre de leur délai.

6.3 Trouver un parallélisme exploitable

Le framework `Executor` facilite la spécification d'une politique d'exécution mais, pour utiliser un `Executor`, vous devez décrire votre tâche sous la forme d'un objet `Runnable`. Dans la plupart des applications serveur, le critère de séparation des tâches est évident : c'est une requête client. Parfois, dans la plupart des applications classiques notamment, il n'est pas si facile de trouver une bonne séparation des tâches. Même dans les applications serveur, il peut également exister un parallélisme exploitable dans une même requête client ; c'est parfois le cas avec les serveurs de bases de données (pour plus de détails sur les forces en présence lors du choix de la séparation des tâches, voir [CPJ 4.4.1.1]).

Dans cette section, nous allons développer plusieurs versions d'un composant permettant différents degrés de concurrence. Ce composant est la partie consacrée au rendu d'une page dans un navigateur : il prend une page HTML et la représente dans un tampon image. Pour simplifier, nous supposerons que le code HTML ne contient que du texte balisé, parsemé d'éléments `image` ayant des dimensions et des URL préétablies.

6.3.1 Exemple : rendu séquentiel d'une page

L'approche la plus simple consiste à traiter séquentiellement le document HTML. À chaque fois que l'on rencontre un marqueur, on le rend dans le tampon image ; lorsque l'on rencontre un marqueur `image`, on récupère l'image sur le réseau et on la dessine également dans le tampon. Cette technique est facile à implémenter et ne nécessite qu'une seule manipulation de chaque élément d'entrée (il n'y a même pas besoin de mettre le document dans un tampon), mais elle risque d'ennuyer l'utilisateur, qui devra attendre longtemps avant que tout le texte soit affiché. Une approche moins ennuyeuse, mais toujours séquentielle, consiste à afficher d'abord les éléments textuels en laissant des emplacements rectangulaires pour les images puis, après avoir effectué cette première passe, à revenir télécharger les images et à les dessiner dans les rectangles qui leur sont associés. C'est ce que fait la classe `SingleThreadRenderer` du Listing 6.10.

Listing 6.10 : Affichage séquentiel des éléments d'une page.

```
public class SingleThreadRenderer {  
    void renderPage(CharSequence source) {  
        renderText(source);  
        List<ImageData> imageData = new ArrayList<ImageData>();  
        for (ImageInfo imageInfo : scanForImageInfo (source))  
            imageData.add(imageInfo.downloadImage());  
        for (ImageData data : imageData)  
            renderImage(data);  
    }  
}
```



Le téléchargement d'une image implique d'attendre la fin d'une opération d'E/S pendant laquelle le processeur est peu actif. L'approche séquentielle risque donc de sous-employer le processeur et de faire attendre plus que nécessaire l'utilisateur. Nous pouvons optimiser l'utilisation du CPU et la réactivité du composant en découplant le problème en tâches indépendantes qui pourront s'exécuter en parallèle.

6.3.2 Tâches partielles : Callable et Future

Le framework Executor utilise `Runnable` comme représentation de base pour les tâches. `Runnable` est une abstraction assez limitée : `run()` ne peut pas renvoyer de valeur ni lancer d'exception contrôlée, bien qu'elle puisse avoir des effets de bord comme écrire dans un fichier journal ou stocker un résultat dans une structure de données partagée.

En pratique, de nombreuses tâches sont des calculs différés – exécuter une requête sur une base de données, télécharger une ressource sur le réseau ou calculer une fonction

compliquée. Pour ce genre de tâche, `Callable` est une meilleure abstraction : son point d'entrée principal, `call()`, renverra une valeur et peut lancer une exception¹. `Executors` contient plusieurs méthodes utilitaires permettant d'envelopper dans un objet `Callable` d'autres types de tâches, comme `Runnable` et `java.security.PrivilegedAction`.

`Runnable` et `Callable` décrivent des tâches abstraites de calcul. Ces tâches sont généralement finies : elles ont un point de départ bien déterminé et finissent par se terminer. Le cycle de vie d'une tâche exécutée par un `Executor` passe par quatre phases : *créée*, *soumise*, *lancée* et *terminée*. Les tâches pouvant s'exécuter pendant un temps assez long, nous voulons également pouvoir annuler une tâche. Dans le framework `Executor`, les tâches qui ont été soumises mais pas encore lancées peuvent toujours être annulées et celles qui ont été lancées peuvent parfois être annulées si elles répondent à une interruption. L'annulation d'une tâche qui s'est déjà terminée n'a aucun effet (l'annulation sera présentée plus en détail au Chapitre 7).

`Future` représente le cycle de vie d'une tâche et fournit des méthodes pour tester si une tâche s'est terminée ou a été exécutée, pour récupérer son résultat et pour annuler la tâche. Les interfaces `Callable` et `Future` sont présentées dans le Listing 6.11. La spécification de `Future` implique que le cycle de vie d'une tâche progresse toujours vers l'avant, pas en arrière – exactement comme le cycle de vie `ExecutorService`. Le comportement de `get()` dépend de l'état de la tâche (pas encore lancée, en cours d'exécution ou terminée). Cette méthode se termine immédiatement ou lance une `Exception` si la tâche s'est déjà terminée mais, dans le cas contraire, elle se bloque jusqu'à la fin de la tâche. Si la tâche se termine en lançant une exception, `get()` relance l'exception en l'enveloppant dans `ExecutionException` ; si elle a été annulée, `get()` lance `CancellationException`. Si `get()` lance `ExecutionException`, l'exception sous-jacente peut être récupérée par `getCause()`.

Listing 6.11 : Interfaces `Callable` et `Future`.

```
public interface Callable<V> {
    V call() throws Exception;
}

public interface Future<V> {
    boolean cancel(boolean mayInterruptIfRunning );
    boolean isCancelled();
    boolean isDone();
    V get() throws InterruptedException , ExecutionException ,
               CancellationException ;
    V get(long timeout, TimeUnit unit) throws InterruptedException,
                                              ExecutionException, CancellationException ,
                                              TimeoutException ;
}
```

Il y a plusieurs moyens de créer un objet `Future` pour décrire une tâche. Les méthodes `submit()` de `ExecutorService` renvoyant toutes un `Future`, vous pouvez soumettre un

1. Pour indiquer qu'une tâche ne renverra pas de valeur, on utilise `Callable<Void>`.

objet Runnable ou Callable à un Executor et récupérer un Future utilisable pour récupérer le résultat ou annuler la tâche. Vous pouvez également instancier explicitement un objet FutureTask pour un Runnable ou un Callable donné (FutureTask implémentant Runnable, cet objet peut être soumis à un Executor pour qu'il l'exécute ou exécuté directement en appelant sa méthode run()).

À partir de Java 6, les implémentations de ExecutorService peuvent redéfinir newTaskFor() dans AbstractExecutorService afin de contrôler l'instanciation de l'objet Future correspondant à un Callable ou à un Runnable qui a été soumis. Comme le montre le Listing 6.12, l'implémentation par défaut se contente de créer un nouvel objet Future Task.

Listing 6.12 : Implémentation par défaut de newTaskFor() dans ThreadPoolExecutor.

```
protected <T> RunnableFuture <T> newTaskFor(Callable<T> task) {  
    return new FutureTask<T>(task);  
}
```

La soumission d'un Runnable ou d'un Callable à un Executor constitue une publication correcte (voir la section 3.5) de ce Runnable ou de ce Callable, du thread qui le soumet au thread qui finira par exécuter la tâche. De même, l'initialisation de la valeur du résultat pour un Future est une publication correcte de ce résultat, du thread dans lequel il a été calculé vers tout thread qui le récupère via get().

6.3.3 Exemple : affichage d'une page avec Future

La première étape pour ajouter du parallélisme à l'affichage d'une page consiste à la diviser en deux tâches, une pour afficher le texte, une autre pour télécharger toutes les images (l'une étant fortement liée au processeur et l'autre, aux E/S, cette approche peut apporter une grosse amélioration, même sur un système monoprocesseur).

Les classes Callable et Future peuvent nous aider à exprimer l'interaction entre ces tâches coopératives. Dans le Listing 6.13, nous créons un objet Callable pour télécharger toutes les images et nous le soumettons à un ExecutorService afin d'obtenir un objet Future décrivant l'exécution de cette tâche. Lorsque la tâche principale a besoin des images, elle attend le résultat en appelant Future.get(). Avec un peu de chance, ce résultat sera déjà prêt lorsqu'on le demandera ; sinon nous aurons au moins lancé le téléchargement des images.

Listing 6.13 : Attente du téléchargement d'image avec Future.

```
public class FutureRender {
    private final ExecutorService executor = ...;

    void renderPage(CharSequence source) {
        final List<ImageInfo> imageInfos = scanForImageInfo (source);
        Callable<List<ImageData>> task =
            new Callable<List<ImageData>>() {
                public List<ImageData> call() {
```



Listing 6.13 : Attente du téléchargement d'image avec Future. (suite)

```
        List<ImageData> result = new ArrayList<ImageData>();
        for (ImageInfo imageInfo : imageInfos)
            result.add(imageInfo.downloadImage());
        return result;
    }
};

Future<List<ImageData>> future = executor.submit(task);
renderText(source);
try {
    List<ImageData> imageData = future.get();
    for (ImageData data : imageData)
        renderImage(data);
} catch (InterruptedException e) {
    // Réaffirme l'état "interrompu" du thread
    Thread.currentThread().interrupt();
    // On n'a pas besoin du résultat, donc on annule aussi la tâche
    future.cancel(true);
} catch (ExecutionException e) {
    throw launderThrowable(e.getCause());
}
}
```

Le comportement de `get()` dépendant de l'état, l'appelant n'a pas besoin de connaître l'état de la tâche ; en outre, la publication correcte de la soumission de la tâche et la récupération du résultat suffisent à rendre cette approche thread-safe. Le code de gestion d'exception associé à l'appel de `Future.get()` traite deux problèmes possibles : la tâche a pu rencontrer une exception ou le thread qui a appelé `get()` a été interrompu avant que le résultat ne soit disponible (voir les sections 5.5.2 et 5.4).

FutureRender permet d'afficher le texte en même temps que les images sont téléchargées. Lorsque toutes les images sont disponibles, elles sont affichées sur la page. C'est une amélioration puisque l'utilisateur voit rapidement le résultat et on exploite ici le parallélisme, bien que l'on puisse faire beaucoup mieux. En effet, l'utilisateur n'a pas besoin d'attendre que *toutes* les images soient téléchargées : il préférerait sûrement les voir séparément à mesure qu'elles deviennent disponibles.

6.3.4 Limitations du parallélisme de tâches hétérogènes

Dans le dernier exemple, nous avons tenté d'exécuter en parallèle deux types de tâches différents – télécharger des images et afficher la page. Cependant, obtenir des gains de performances intéressants en mettant en parallèle des tâches séquentielles hétérogènes peut être assez compliqué. Deux personnes peuvent diviser efficacement le travail qui consiste à faire la vaisselle : l'une lave pendant que l'autre essuie. Cependant, affecter un type de tâche différent à chaque participant n'est pas très évolutif : si plusieurs personnes proposent leur aide pour la vaisselle, il n'est pas évident de savoir comment les répartir sans qu'elles se gênent ou sans restructurer la division du travail. Si l'on ne trouve pas un parallélisme suffisamment précis entre des tâches similaires, cette approche produira des résultats décevants.

Un autre problème de la division des tâches hétérogènes entre plusieurs participants est que ces tâches peuvent avoir des tailles différentes. Si vous divisez les tâches A et B entre deux participants, mais que A soit dix fois plus longue que B, nous n'aurez accéléré le traitement total que de 9 %. Enfin, diviser le travail entre plusieurs participants implique toujours un certain coût de coordination ; pour que cette division soit intéressante, ce coût doit être plus que compensé par l'amélioration de la productivité due au parallélisme.

FutureRendered utilise deux tâches : l'une pour afficher le texte, l'autre pour télécharger les images. Si la première est bien plus rapide que la seconde, comme c'est sûrement le cas, les performances ne seront pas beaucoup différentes de celles de la version séquentielle alors que le code sera bien plus compliqué. Le mieux que l'on puisse obtenir avec deux threads est une vitesse multipliée par deux. Par conséquent, tenter d'augmenter la concurrence en mettant en parallèle des activités hétérogènes peut demander beaucoup d'efforts et il y a une limite à la dose de concurrence supplémentaire que vous pouvez en tirer (voir les sections 11.4.2 et 11.4.3 pour un autre exemple de ce phénomène).

Le véritable bénéfice en terme de performances de la division d'un programme en tâches s'obtient lorsque l'on peut traiter en parallèle un grand nombre de tâches indépendantes et homogènes.

6.3.5 CompletionService : quand Executor rencontre BlockingQueue

Si vous avez un grand nombre de calculs à soumettre à un Executor et que vous vouliez récupérer leurs résultats dès qu'ils sont disponibles, vous pouvez conserver les objets Future associés à chaque tâche et les interroger régulièrement pour savoir s'ils se sont terminés en appelant leurs méthodes `get()` avec un délai d'expiration de zéro. C'est possible, mais ennuyeux. Heureusement, il y a une meilleure solution : un *service de terminaison*.

CompletionService combine les fonctionnalités d'un Executor et d'une BlockingQueue. Vous pouvez lui soumettre des tâches `Callable` pour les exécuter et utiliser des méthodes sur les files comme `take()` et `poll()` pour obtenir les résultats calculés, empaquetés sous la forme d'objets Future, dès qu'ils sont disponibles. `ExecutorCompletionService` implémente `CompletionService`, en déléguant le calcul à un Executor.

L'implémentation de `ExecutorCompletionService` est assez simple à comprendre. Le constructeur crée une `BlockingQueue` qui contiendra les résultats terminés. `FutureTask` dispose d'une méthode `done()` qui est appelée lorsque le calcul s'achève. Lorsqu'une tâche est soumise, elle est enveloppée dans un objet `QueueingFuture`, une sous-classe de `FutureTask` qui redéfinit `done()` pour placer le résultat dans la `BlockingQueue`, comme le montre le Listing 6.14. Les méthodes `take()` et `poll()` délèguent leur traitement à la `BlockingQueue` et se bloquent si le résultat n'est pas encore disponible.

Listing 6.14 : La classe QueueingFuture utilisée par ExecutorCompletionService.

```
private class QueueingFuture <V> extends FutureTask<V> {
    QueueingFuture(Callable<V> c) { super(c); }
    QueueingFuture(Runnable t, V r) { super(t, r); }

    protected void done() {
        completionQueue .add(this);
    }
}
```

6.3.6 Exemple : affichage d'une page avec CompletionService

CompletionService va nous permettre d'améliorer les performances de l'affichage de la page de deux façons : un temps d'exécution plus court et une meilleure réactivité. Nous pouvons créer une tâche distincte pour télécharger chaque image et les exécuter dans un pool de threads, ce qui aura pour effet de transformer le téléchargement séquentiel en téléchargement en parallèle : cela réduit le temps nécessaire à l'obtention de toutes les images. En récupérant les résultats à partir du CompletionService et en affichant chaque image dès qu'elle est disponible, nous fournissons à l'utilisateur une interface plus dynamique et plus réactive. Cette implémentation est décrite dans le Listing 6.15.

Listing 6.15 : Utilisation de CompletionService pour afficher les éléments de la page dès qu'ils sont disponibles.

```
public class Renderer {
    private final ExecutorService executor;

    Renderer(ExecutorService executor) { this.executor = executor; }

    void renderPage(CharSequence source) {
        List<ImageInfo> info = scanForImageInfo (source);
        CompletionService <ImageData> completionService =
            new ExecutorCompletionService <ImageData>(executor);
        for (final ImageInfo imageInfo : info)
            completionService.submit( new Callable<ImageData>() {
                public ImageData call() {
                    return imageInfo.downloadImage();
                }
            });
        renderText(source);

        try {
            for (int t = 0, n = info.size(); t < n; t++) {
                Future<ImageData> f = completionService.take();
                ImageData imageData = f.get();
                renderImage(imageData);
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        } catch (ExecutionException e) {
            throw launderThrowable(e.getCause());
        }
    }
}
```

Plusieurs ExecutorCompletionServices pouvant partager le même Executor, il est tout à fait sensé de créer un ExecutorCompletionService privé à un calcul particulier tout en partageant un Executor commun. Utilisé de cette façon, un CompletionService agit comme un descripteur d'un ensemble de calculs, exactement comme un Future agit comme un descripteur d'un simple calcul. En mémorisant le nombre de tâches soumises au CompletionService et en comptant le nombre de résultats récupérés, vous pouvez savoir quand tous les résultats d'un ensemble de calculs ont été obtenus, même si vous utilisez un Executor partagé.

6.3.7 Imposer des délais aux tâches

Le résultat d'une activité qui met trop de temps à se terminer peut ne plus être utile et l'activité peut alors être abandonnée. Une application web qui récupère des publicités à partir d'un serveur externe, par exemple, pourrait afficher une publicité par défaut si le serveur ne répond pas au bout de 2 secondes afin de ne pas détériorer la réactivité du site. De la même façon, un portail web peut récupérer des données en parallèle à partir de plusieurs sources mais ne vouloir attendre qu'un certain temps avant d'afficher les données.

Le principal défi de l'exécution des tâches dans un délai imparti consiste à s'assurer que l'on n'attendra pas plus longtemps que ce délai pour obtenir une réponse ou à constater qu'une réponse n'est pas fournie. La version temporisée de Future.get() fournit cette garantie puisqu'elle se termine dès que le résultat est disponible, mais lance TimeoutException si le résultat n'est pas prêt dans le délai fixé.

Le deuxième problème avec les tâches avec délais consiste à les arrêter lorsque le temps imparti s'est écoulé afin qu'elles ne consomment pas inutilement des ressources en continuant de calculer un résultat qui, de toute façon, ne sera pas utilisé. Pour cela, on peut faire en sorte que la tâche gère de façon stricte son propre délai et se termine d'elle-même lorsqu'il est écoulé, ou l'on peut annuler la tâche lorsque le délai a expiré. Une nouvelle fois, Future peut nous aider ; si un appel get() temporisé se termine avec une exception TimeoutException, nous pouvons annuler la tâche via l'objet Future. Si la tâche a été conçue pour être annulable (voir Chapitre 7), nous pouvons y mettre fin précocément afin qu'elle ne consomme pas de ressources inutiles. C'est la technique utilisée dans les Listings 6.13 et 6.16.

Listing 6.16 : Récupération d'une publicité dans un délai imparti.

```
Page renderPageWithAd() throws InterruptedException {
    long endNanos = System.nanoTime() + TIME_BUDGET;
    Future<Ad> f = exec.submit(new FetchAdTask());
    // Affiche la page pendant qu'on attend la publicité
    Page page = renderPageBody();
    Ad ad;
    try {
        // On n'attend que pendant le temps restant
        long timeLeft = endNanos - System.nanoTime();
```

Listing 6.16 : Récupération d'une publicité dans un délai imparti. (suite)

```

        ad = f.get(timeLeft, NANOSECONDS);
    } catch (ExecutionException e) {
        ad = DEFAULT_AD;
    } catch (TimeoutException e) {
        ad = DEFAULT_AD;
        f.cancel(true);
    }
    page.setAd(ad);
    return page;
}

```

Ce dernier listing montre une application typique de la version temporisée de `Future.get()`. Ce code produit une page web composite avec le contenu demandé accompagné d'une publicité récupérée à partir d'un serveur externe. La tâche d'obtention de la publicité est confiée à un `Executor` ; le code calcule le reste du contenu de la page puis attend la publicité jusqu'à l'expiration de son délai¹. Si celui-ci expire, il annule² la tâche de récupération et utilise une publicité par défaut.

6.3.8 Exemple : portail de réservations

L'approche par délai de la section précédente peut aisément se généraliser à un nombre quelconque de tâches. Dans un portail de réservation, par exemple, l'utilisateur saisit des dates de voyages et le portail recherche et affiche les tarifs d'un certain nombre de vols, hôtels ou sociétés de location de véhicule. Selon la société, la récupération d'un tarif peut impliquer l'appel d'un service web, la consultation d'une base de données, l'exécution d'une transaction EDI ou tout autre mécanisme. Au lieu que le temps de réponse de la page ne soit décidé par la réponse la plus lente, il peut être préférable de ne présenter que les informations disponibles dans un certain délai. Dans le cas de fournisseurs ne répondant pas à temps, la page pourrait soit les omettre totalement, soit écrire un texte comme "Air Java n'a pas répondu à temps".

Obtenir un tarif auprès d'une compagnie étant indépendant de l'obtention des tarifs des autres compagnies, la récupération d'un tarif est une bonne candidate au découpage en tâches, permettant ainsi l'obtention en parallèle des différents tarifs. Il serait assez simple de créer n tâches, de les soumettre à un pool de threads, de mémoriser les objets `Future` et d'utiliser un appel `get()` temporisé pour obtenir séquentiellement chaque résultat via son `Future`, mais il existe un moyen plus simple : `invokeAll()`.

Le Listing 6.17 utilise la version temporisée de `invokeAll()` pour soumettre plusieurs tâches à un `ExecutorService` et récupérer les résultats. La méthode `invokeAll()` prend

-
1. Le délai passé à `get()` est calculé en soustrayant l'heure courante de la date limite ; on peut donc obtenir un nombre négatif mais, toutes les méthodes temporisées de `java.util.concurrent` traitant les délais négatifs comme des délais nuls, aucun code supplémentaire n'est nécessaire pour traiter ce cas.
 2. Le paramètre `true` de `Future.cancel()` signifie que le thread de la tâche peut être interrompu même si la tâche est en cours d'exécution (voir Chapitre 7).

en paramètre une collection de tâches et renvoie une collection de Future. Ces deux collections ont des structures identiques ; invokeAll() ajoute les Future dans le résultat selon l'ordre imposé par l'itérateur de la collection des tâches, ce qui permet à l'appelant d'associer un Future à l'objet Callable qu'il représente. La version temporisée de invokeAll() se terminera quand toutes les tâches se seront terminées, si le thread appelant est interrompu ou si le délai imparti a expiré. Toutes les tâches non terminées à l'expiration du délai sont annulées. Au retour de invokeAll(), chaque tâche se sera donc terminée normalement ou aura été annulée ; le code client peut appeler get() ou isCancelled() pour le savoir.

Listing 6.17 : Obtention de tarifs dans un délai imparti.

```
private class QuoteTask implements Callable<TravelQuote> {
    private final TravelCompany company;
    private final TravelInfo travelInfo;
    ...
    public TravelQuote call() throws Exception {
        return company.solicitQuote(travelInfo);
    }
}

public List<TravelQuote> getRankedTravelQuotes (
    TravelInfo travelInfo, Set<TravelCompany> companies,
    Comparator<TravelQuote> ranking, long time, TimeUnit unit)
    throws InterruptedException {
    List<QuoteTask> tasks = new ArrayList<QuoteTask>();
    for (TravelCompany company : companies)
        tasks.add(new QuoteTask(company, travelInfo));

    List<Future<TravelQuote>> futures =
        exec.invokeAll(tasks, time, unit);

    List<TravelQuote> quotes =
        new ArrayList<TravelQuote>(tasks.size());
    Iterator<QuoteTask> taskIter = tasks.iterator();
    for (Future<TravelQuote> f : futures) {
        QuoteTask task = taskIter.next();
        try {
            quotes.add(f.get());
        } catch (ExecutionException e) {
            quotes.add(task.getFailureQuote(e.getCause()));
        } catch (CancellationException e) {
            quotes.add(task.getTimeoutQuote (e));
        }
    }

    Collections.sort(quotes, ranking);
    return quotes;
}
```

Résumé

Structurer les applications autour de l'exécution en tâches permet de simplifier le développement et de faciliter le parallélisme. Grâce au framework Executor, vous pouvez séparer la soumission des tâches de la politique d'exécution ; en outre, de nombreuses politiques

d'exécution sont disponibles : à chaque fois que vous devez créer des threads pour exécuter des tâches, pensez à utiliser un Executor. Pour obtenir le maximum de bénéfice de la décomposition d'une application en tâches, vous devez trouver comment séparer les tâches. Dans certaines applications, cette décomposition est évidente alors que, dans d'autres, elle nécessite une analyse un peu plus poussée pour faire ressortir un parallélisme exploitable.

Annulation et arrêt

Lancer des tâches et des threads est une opération simple. La plupart du temps, on leur permet de décider quand s'arrêter en les laissant s'exécuter jusqu'à la fin. Parfois, cependant, nous voulons stopper des tâches plus tôt que prévu, par exemple parce qu'un utilisateur a annulé une opération ou parce que l'application doit s'arrêter rapidement. Il n'est pas toujours simple de stopper correctement, rapidement et de façon fiable des tâches et des threads. Java ne fournit aucun mécanisme pour forcer en toute sécurité un thread à stopper ce qu'il était en train de faire¹. En revanche, il fournit les *interruptions*, un mécanisme coopératif qui permet à un thread de demander à un autre d'arrêter ce qu'il est en train de faire.

L'approche coopérative est nécessaire car on souhaite rarement qu'une tâche, un thread ou un service s'arrête *immédiatement* puisqu'il pourrait laisser des structures de données partagées dans un état incohérent. Ces tâches et services doivent donc être codés pour que, lorsqu'on leur demande, ils nettoient le travail en cours puis se terminent. Cette pratique apporte une grande souplesse car le code lui-même est généralement plus qualifié que le code demandant l'annulation pour effectuer le nettoyage nécessaire.

Les problèmes de fin de vie peuvent compliquer la conception et l'implémentation des tâches, des services et des applications ; c'est également un aspect important de la conception des programmes qui est trop souvent ignoré. Bien gérer les pannes, les arrêts et les annulations fait partie de ces caractéristiques qui distinguent une application bien construite d'une autre qui se contente de fonctionner. Ce chapitre présente les mécanismes d'annulation et d'interruption et montre comment coder les tâches et les services pour qu'ils répondent aux demandes d'annulation.

1. Les méthodes dépréciées `Thread.stop()` et `suspend()` étaient une tentative pour fournir ce mécanisme, mais on a vite réalisé qu'elles avaient de sérieux défauts et qu'il fallait les éviter. Pour une explication des problèmes avec ces méthodes, consultez la page <http://java.sun.com/j2se/1.5.0/docs/guide/misc/threadPrimitiveDeprecation>.

7.1 Annulation des tâches

Une activité est *annulable*, si un code externe peut l'amener à se terminer avant sa fin normale. Il existe de nombreuses raisons d'annuler une activité :

- **Annulation demandée par l'utilisateur.** Celui-ci a cliqué sur le bouton "Annuler" d'une interface graphique ou a demandé l'annulation *via* une interface de gestion comme JMX (*Java Management Extensions*).
- **Activité limitée dans le temps.** Une application parcourt un espace de problèmes pendant un temps fini et choisit la meilleure solution trouvée pendant ce temps imparti. Quand le délai expire, toutes les tâches encore en train de chercher sont annulées.
- **Événement d'une application.** Une application parcourt un espace de problèmes en le décomposant pour que des tâches distinctes examinent différentes régions de cet espace. Lorsqu'une tâche trouve une solution, toutes celles encore en train de chercher sont annulées.
- **Erreurs.** Un robot web recherche des pages, les stocke ou produit un résumé des données sur disque. S'il rencontre une erreur (disque plein, par exemple), ses autres tâches doivent être annulées, éventuellement après avoir enregistré leur état courant afin de pouvoir les reprendre plus tard.
- **Arrêt.** Lorsque l'on arrête une application ou un service, il faut faire quelque chose pour le travail en cours ou en attente de traitement. Dans un arrêt en douceur, les tâches en cours d'exécution peuvent être autorisées à se terminer alors qu'elles peuvent être annulées dans un arrêt plus brutal.

Il n'y a aucun moyen sûr d'arrêter autoritairement un thread en Java et donc aucun moyen sûr d'arrêter une tâche. Il n'existe que des mécanismes coopératifs dans lesquels la tâche et le code qui demande l'annulation respectent un protocole d'agrément.

L'un de ces mécanismes consiste à positionner un indicateur "demande d'annulation" que la tâche consultera périodiquement ; si elle constate qu'il est positionné, elle se termine dès que possible. C'est la technique qu'utilise la classe `PrimeGenerator` du Listing 7.1, qui énumère les nombres premiers jusqu'à son annulation. La méthode `cancel()` positionne l'indicateur d'annulation qui est consulté par la boucle principale avant chaque recherche d'un nouveau nombre premier (pour que cela fonctionne correctement, l'indicateur doit être `volatile`).

Listing 7.1 : Utilisation d'un champ `volatile` pour stocker l'état d'annulation.

```
@ThreadSafe
public class PrimeGenerator implements Runnable {
    @GuardedBy("this")
    private final List<BigInteger> primes
        = new ArrayList<BigInteger>();
    private volatile boolean cancelled;
```

```

public void run() {
    BigInteger p = BigInteger.ONE;
    while (!cancelled) {
        p = p.nextProbablePrime();
        synchronized (this) {
            primes.add(p);
        }
    }
}

public void cancel() { cancelled = true; }

public synchronized List<BigInteger> get() {
    return new ArrayList<BigInteger>(primes);
}
}

```

Le Listing 7.2 présente un exemple d'utilisation de cette classe dans lequel on laisse une seconde au générateur de nombres premiers avant de l'annuler. Le générateur ne s'arrêtera pas nécessairement après exactement une seconde puisqu'il peut y avoir un léger délai entre la demande d'annulation et le moment où la boucle revient tester l'indicateur. La méthode `cancel()` est appelée à partir d'un bloc `finally` pour garantir que le générateur sera annulé même si l'appel à `sleep()` est interrompu. Si `cancel()` n'était pas appelé, le thread de recherche des nombres premiers continuerait de s'exécuter, ce qui consommerait des cycles processeur et empêcherait la JVM de se terminer.

Listing 7.2 : Génération de nombres premiers pendant une seconde.

```

List<BigInteger> aSecondOfPrimes() throws InterruptedException {
    PrimeGenerator generator = new PrimeGenerator();
    new Thread(generator).start();
    try {
        SECONDS.sleep(1);
    } finally {
        generator.cancel();
    }
    return generator.get();
}

```

Pour être annulable, une tâche doit avoir une *politique d'annulation* précisant le "comment", le "quand" et le "quoi" de l'annulation : comment un autre code peut demander l'annulation, quand est-ce que la tâche vérifie qu'une annulation a été demandée et quelles actions doit entreprendre la tâche en réponse à une demande d'annulation.

Prenons l'exemple d'une annulation d'un chèque : les banques ont des règles qui précisent la façon de demander l'annulation d'un paiement, les garanties sur leur réactivité pour une telle demande et les procédures qui suivront l'annulation du chèque (prévenir l'autre banque impliquée dans la transaction et prélever des frais de dossier sur le compte du demandeur, par exemple). Prises ensemble, ces procédures et ces garanties forment la politique d'annulation d'un paiement par chèque.

`PrimeGenerator` utilise une politique simple : le code client demande l'annulation en appelant `cancel()`, le code principal teste les demandes d'annulation pour chaque nombre premier et se termine lorsqu'il détecte qu'une demande a eu lieu.

7.1.1 Interruption

Le mécanisme d'annulation de `PrimeGenerator` finira par provoquer la fin de la tâche de calcul des nombres premiers, mais cela peut prendre un certain temps. Si une tâche utilisant cette approche appelle une méthode bloquante comme `BlockingQueue.put()`, nous pourrions avoir un sérieux problème : elle pourrait ne jamais tester l'indicateur d'annulation et donc ne jamais se terminer.

Ce problème est illustré par la classe `BrokenPrimeProducer` du Listing 7.3. Le producteur génère des nombres premiers et les place dans une file bloquante. Si le producteur va plus vite que le consommateur, la file se remplira et un appel à `put()` sera bloquant. Que se passera-t-il si le consommateur essaie d'annuler la tâche productrice alors qu'elle est bloquée dans `put()` ? Le consommateur peut appeler `cancel()`, qui positionnera l'indicateur `cancelled`, mais le producteur ne le testera jamais car il ne sortira jamais de l'appel à `put()` bloquant (puisque le consommateur a cessé de récupérer des nombres premiers dans la file).

Listing 7.3 : Annulation non fiable pouvant bloquer les producteurs. Ne le faites pas.

```
class BrokenPrimeProducer extends Thread {
    private final BlockingQueue<BigInteger> queue;
    private volatile boolean cancelled = false;

    BrokenPrimeProducer (BlockingQueue<BigInteger> queue) {
        this.queue = queue;
    }

    public void run() {
        try {
            BigInteger p = BigInteger.ONE;
            while (!cancelled)
                queue.put(p = p.nextProbablePrime());
        } catch (InterruptedException consumed) { }
    }

    public void cancel() { cancelled = true; }
}

void consumePrimes() throws InterruptedException {
    BlockingQueue<BigInteger> primes = ...;
    BrokenPrimeProducer producer = new BrokenPrimeProducer(primes);
    producer.start();
    try {
        while (needMorePrimes())
            consume(primes.take());
    } finally {
        producer.cancel();
    }
}
```



Comme nous l'avions évoqué au Chapitre 5, certaines méthodes bloquantes de la bibliothèque permettent d'être *interrompues*. L'interruption d'un thread est un mécanisme coopératif permettant d'envoyer un signal à un thread pour qu'il arrête son traitement en cours et fasse autre chose, s'il en a envie et quand il le souhaite.

Rien dans l'API ou les spécifications du langage ne lie une interruption à une sémantique particulière de l'annulation mais, en pratique, utiliser une interruption pour autre chose qu'une annulation est une démarche fragile et difficile à gérer dans les applications de taille importante.

Chaque thread a un *indicateur d'interruption* booléen précisant s'il a été interrompu. Comme le montre le Listing 7.4, la classe `Thread` contient des méthodes permettant d'interrompre un thread et d'interroger cet indicateur. La méthode `interrupt()` interrompt le thread cible et `isInterrupted()` renvoie son état d'interruption. La méthode statique `interrupted()` porte mal son nom puisqu'elle *réinitialise* l'indicateur d'interruption du thread courant et renvoie sa valeur précédente : c'est le seul moyen de réinitialiser cet indicateur.

Listing 7.4 : Méthodes d'interruption de Thread.

```
public class Thread {  
    public void interrupt() { ... }  
    public boolean isInterrupted() { ... }  
    public static boolean interrupted() { ... }  
    ...  
}
```

Les méthodes bloquantes de la bibliothèque, comme `Thread.sleep()` et `Object.wait()`, tentent de détecter quand un thread a été interrompu, auquel cas elles se terminent plus tôt que prévu. Elles répondent à l'interruption en réinitialisant l'indicateur d'interruption et en lançant `InterruptedException` pour indiquer que l'opération bloquante s'est terminée précocément à cause d'une interruption. La JVM ne garantit pas le temps que mettra une méthode bloquante pour détecter une interruption mais, en pratique, c'est relativement rapide.

Si un thread est interrompu alors qu'il n'était *pas* bloqué, son indicateur d'interruption est positionné et c'est à l'activité annulée de l'interroger pour détecter l'interruption. En ce sens, l'interruption est "collante" : si elle ne déclenche pas une `InterruptedException`, la trace de l'interruption persiste jusqu'à ce que quelqu'un réinitialise délibérément l'indicateur.

L'appel `interrupt()` n'empêche pas nécessairement le thread cible de continuer ce qu'il est en train de faire : il indique simplement qu'une interruption a été demandée.

En réalité, une interruption n'interrompt pas un thread en cours d'exécution : elle ne fait que *demandez* au thread de s'interrompre lui-même à la prochaine occasion (ces occasions sont appelées *points d'annulation*). Certaines méthodes comme `wait()`, `sleep()` et `join()` prennent ces requêtes au sérieux et lèvent une exception lorsqu'elles les reçoivent ou rencontrent un indicateur d'interruption déjà positionné. D'autres, pourtant tout

à fait correctes, peuvent totalement ignorer ces requêtes et les laisser en place pour que le code appelant puisse en faire quelque chose. Les méthodes mal conçues perdent les requêtes d'interruption, empêchant ainsi le code situé plus haut dans la pile des appels d'agir en conséquence. La méthode statique `interrupted()` doit être utilisée avec précaution puisqu'elle réinitialise l'indicateur d'interruption du thread courant. Si vous lappelez et qu'elle renvoie `true`, vous devez agir (à moins que vous ne vouliez perdre l'interruption) en lançant `InterruptedException` ou en restaurant l'indicateur en rappelant `interrupt()` comme dans le Listing 5.10.

`BrokenPrimeProducer` montre que les mécanismes personnalisés d'annulation ne font pas toujours bon ménage avec les méthodes bloquantes de la bibliothèque. Si vous codez vos tâches pour qu'elles répondent aux interruptions, vous pouvez les utiliser comme mécanisme d'annulation et tirer parti du support des interruptions fourni par de nombreuses classes de la bibliothèque.

Une interruption est généralement le meilleur moyen d'implémenter l'annulation.

Comme le montre le Listing 7.5, on peut facilement corriger (et simplifier) `BrokenPrimeProducer` en utilisant une interruption à la place d'un indicateur booléen pour demander l'annulation.

Listing 7.5 : Utilisation d'une interruption pour l'annulation.

```
class PrimeProducer extends Thread {  
    private final BlockingQueue <BigInteger> queue;  
  
    PrimeProducer(BlockingQueue <BigInteger> queue) {  
        this.queue = queue;  
    }  
  
    public void run() {  
        try {  
            BigInteger p = BigInteger.ONE;  
            while (!Thread.currentThread().isInterrupted())  
                queue.put(p = p.nextProbablePrime());  
        } catch (InterruptedException consumed) {  
            /* Autorise le thread à se terminer */  
        }  
    }  
    public void cancel() { interrupt(); }  
}
```

L'interruption peut être détectée à deux endroits dans chaque itération de la boucle : dans l'appel bloquant à `put()` et en interrogeant explicitement l'indicateur d'interruption dans le test de boucle. Ce test explicite n'est d'ailleurs pas strictement nécessaire ici à cause de l'appel bloquant à `put()` mais il rend `PrimeProducer` plus réactive à une interruption puisqu'il teste l'interruption *avant* de commencer la recherche d'un nombre premier, ce qui peut être une opération assez longue. Lorsque les appels aux méthodes

bloquantes ne sont pas suffisamment fréquents pour produire la réactivité attendue, un test explicite de l'indicateur d'interruption peut améliorer la situation.

7.1.2 Politiques d'interruption

Tout comme les tâches devraient avoir une politique d'annulation, les threads devraient respecter une politique d'interruption. Une telle politique détermine la façon dont un thread interprétera une demande d'interruption : ce qu'il fera (s'il fait quelque chose) lorsqu'il en détectera une, quelles unités de travail seront considérées comme atomiques par rapport à l'interruption et à quelle vitesse il réagira à cette interruption.

La politique d'interruption la plus raisonnable est une forme d'annulation au niveau du thread ou du service : quitter aussi vite que possible, nettoyer si nécessaire et, éventuellement, prévenir l'entité propriétaire que le thread se termine. On peut établir d'autres politiques comme mettre un service en pause ou le relancer mais, en ce cas, les threads ou les pools de threads utilisant une politique d'interruption non standard devront peut-être se limiter à des tâches écrites pour tenir compte de cette politique.

Il est important de faire la différence entre la façon dont les *tâches* et les *threads* devraient réagir aux interruptions. Une simple requête d'interruption peut avoir d'autres destinataires que celui qui est initialement visé : interrompre un thread dans un pool peut signifier "annule la tâche courante" et "termine ce thread".

Les tâches ne s'exécutent pas dans des threads qu'elles possèdent ; elles empruntent des threads qui appartiennent à un service comme un pool de threads. Le code qui ne possède pas le thread (pour un pool de thread, il s'agit du code situé à l'extérieur de l'implémentation du pool) doit faire attention à préserver l'indicateur d'interruption afin que le code propriétaire puisse éventuellement y réagir, même si le code "invité" réagit également à l'interruption (lorsque l'on garde une maison, on ne jette pas le courrier qui arrive pour les propriétaires – on le sauvegarde et on les laisse s'en occuper à leur retour, même si on lit leurs magazines).

C'est la raison pour laquelle la plupart des méthodes bloquantes de la bibliothèque se contentent de lancer `InterruptedException` en réponse à une interruption. Comme elles ne s'exécuteront jamais dans un thread qui leur appartient, elles implémentent la politique d'annulation la plus raisonnable pour une tâche ou un code d'une bibliothèque : elles débarrassent le plancher le plus vite possible et transmettent l'interruption à l'appelant afin que le code situé plus haut dans la pile des appels puisse prendre les mesures nécessaires.

Une tâche n'a pas nécessairement besoin de tout abandonner lorsqu'elle détecte une demande d'interruption – elle peut choisir un moment plus opportun en mémorisant qu'elle a été interrompue, en finissant le travail qu'elle effectuait puis en lançant `InterruptedException` ou tout autre signal indiquant l'interruption. Cette technique permet d'éviter

qu'une interruption survenant au beau milieu d'une modification n'abîme les structures de données.

Une tâche ne devrait jamais rien supposer sur la politique d'interruption du thread qui l'exécute, sauf si elle a été explicitement conçue pour s'exécuter au sein d'un service utilisant une politique d'interruption spécifique. Qu'elle interprète une interruption comme une annulation ou qu'elle agisse d'une certaine façon en cas d'interruption, une tâche devrait prendre soin de préserver l'indicateur d'interruption du thread qui s'exécute. Si elle ne se contente pas de propager `InterruptedException` à son appelant, elle devrait restaurer l'indicateur d'interruption après avoir capturé l'exception :

```
Thread.currentThread().interrupt();
```

Tout comme le code d'une tâche ne devrait pas faire de supposition sur ce que signifie une interruption pour le thread qui l'exécute, le code d'annulation ne devrait rien supposer de la politique d'interruption des threads. Un thread ne devrait être interrompu que par son propriétaire ; ce dernier peut encapsuler la connaissance de la politique d'interruption du thread dans un mécanisme d'annulation adéquat – une méthode d'arrêt, par exemple.

Chaque thread ayant sa propre politique d'interruption, vous devriez interrompre un thread que si vous savez ce que cela signifie pour lui.

Certaines critiques se sont moquées des interruptions de Java car elles ne permettent pas de faire des interruptions préemptives et forcent pourtant les développeurs à traiter `InterruptedException`. Cependant, la possibilité de reporter la prise en compte d'une demande d'interruption permet aux développeurs de créer des politiques d'interruption souples qui trouvent un équilibre entre réactivité et robustesse en fonction de l'application.

7.1.3 Répondre aux interruptions

Comme on l'a mentionné dans la section 5.4, il y a deux stratégies possibles pour traiter une `InterruptedException` lorsqu'on appelle une méthode bloquante interruptible comme `Thread.sleep()` ou `BlockingQueue.put()` :

- propager l'exception (éventuellement après un peu de ménage spécifique à la tâche), ce qui rend également votre méthode bloquante et interruptible ;
- restaurer l'indicateur d'interruption pour que le code situé plus haut dans la pile des appels puisse la traiter.

La propagation de `InterruptedException` consiste simplement à ajouter le nom de cette classe à la clause `throws`, comme le fait la méthode `getNextTask()` du Listing 7.6.

Listing 7.6 : Propagation de InterruptedException aux appelants.

```
BlockingQueue <Task> queue;  
...  
public Task getNextTask() throws InterruptedException {  
    return queue.take();  
}
```

Si vous ne voulez pas ou ne pouvez pas propager `InterruptedException` (parce que votre tâche est définie par un `Runnable`, par exemple), vous devez utiliser un autre moyen pour préserver la demande d'interruption. Pour ce faire, la méthode standard consiste à restaurer l'indicateur d'interruption en appelant à nouveau `interrupt()`. Vous ne devez pas absorber `InterruptedException` en la capturant pour ne rien en faire, sauf si votre code implémente la politique d'interruption d'un thread. La classe `PrimeProducer` absorbe l'interruption, mais le fait en sachant que le thread va se terminer et qu'il n'y a donc pas de code plus haut dans la pile d'appels qui a besoin de savoir que cette interruption a eu lieu. Mais, dans la plupart des cas, un code ne connaît pas le thread qu'il exécutera et il devrait donc préserver l'indicateur d'interruption.

Seul le code qui implémente la politique d'interruption d'un thread peut absorber une demande d'interruption. Les tâches générales et le code d'une bibliothèque ne devraient jamais le faire.

Les activités qui ne reconnaissent pas les annulations mais qui appellent quand même des méthodes bloquantes interruptibles devront les appeler dans une boucle, en réessayant lorsque l'interruption a été détectée. Dans ce cas, elles doivent sauvegarder localement l'indicateur d'interruption et le restaurer juste avant de se terminer, comme le montre le Listing 7.7, plutôt qu'immédiatement lorsqu'elles capturent `InterruptedException`. Positionner trop tôt l'indicateur d'interruption pourrait provoquer une boucle sans fin car la plupart des méthodes bloquantes interruptibles le testent avant de commencer et lancent immédiatement `InterruptedException` s'il est positionné (ces méthodes interrogent cet indicateur avant de se bloquer ou d'effectuer un travail un peu important afin de réagir le plus vite possible à une interruption).

Listing 7.7 : Tâche non annulable qui restaure l'interruption avant de se terminer.

```
public Task getNextTask(BlockingQueue <Task> queue) {  
    boolean interrupted = false;  
    try {  
        while (true) {  
            try {  
                return queue.take();  
            } catch (InterruptedException e) {  
                interrupted = true;  
                // Ne fait rien et réessaie  
            }  
        }  
    }
```

Listing 7.7 : Tâche non annulable qui restaure l'interruption avant de se terminer. (suite)

```
    } finally {
        if (interrupted)
            Thread.currentThread().interrupt();
    }
}
```

Si votre code n'appelle pas de méthodes bloquantes interruptibles, il peut quand même réagir aux interruptions en interrogeant l'indicateur d'interruption du thread courant dans le code de la tâche. Choisir la fréquence de cette consultation relève d'un compromis entre efficacité et réactivité : si vous devez être très réactif, vous ne pouvez pas vous permettre d'appeler des méthodes susceptibles de durer longtemps et qui ne sont pas elles-mêmes réactives aux interruptions ; cela peut donc vous limiter dans vos appels.

L'annulation peut impliquer d'autres états que celui d'interruption ; ce dernier peut servir à attirer l'attention du thread et les informations stockées ailleurs par le thread qui interrompt peuvent être utilisées pour fournir des instructions supplémentaires au thread interrompu (il faut bien sûr utiliser une synchronisation lorsque l'on accède à ces informations). Lorsqu'un thread détenu par un `ThreadPoolExecutor` détecte une interruption, par exemple, il vérifie si le pool est en cours d'arrêt, auquel cas il peut faire un peu de nettoyage avant de se terminer ; sinon il peut créer un autre thread pour que le pool de threads puisse garder la même taille.

7.1.4 Exemple : exécution avec délai

De nombreux problèmes peuvent mettre un temps infini (l'énumération de tous les nombres premiers, par exemple) ; pour d'autres, la réponse pourrait être trouvée assez vite mais ils peuvent également durer éternellement. Or, dans certains cas, il peut être utile de pouvoir dire "consacre dix minutes à trouver la réponse" ou "énumère toutes les réponses possibles pendant dix minutes".

La méthode `aSecondOfPrimes()` du Listing 7.2 lance un objet `PrimeGenerator` et l'interrompt après 1 seconde. Bien que ce dernier puisse mettre un peu plus de 1 seconde pour s'arrêter, il finira par noter l'interruption et cessera son exécution, ce qui permettra au thread de se terminer. Cependant, un autre aspect de l'exécution d'une tâche est que l'on veut savoir si elle lance une exception : si `PrimeGenerator` lance une exception non contrôlée avant l'expiration du délai, celle-ci passera sûrement inaperçue puisque le générateur de nombres premiers s'exécute dans un thread séparé qui ne gère pas explicitement les exceptions.

Le Listing 7.8 est une tentative d'exécuter un `Runnable` quelconque pendant un certain temps. Il lance la tâche dans le thread appelant et planifie une tâche d'annulation pour l'interrompre après un certain délai. Ceci règle le problème des exceptions non contrôlées lancées à partir de la tâche puisqu'elles peuvent maintenant être capturées par celui qui a appelé `timedRun()`.

Listing 7.8 : Planification d'une interruption sur un thread emprunté. Ne le faites pas.

```
private static final ScheduledExecutorService cancelExec = ...;

public static void timedRun(Runnable r, long timeout, TimeUnit unit) {
    final Thread taskThread = Thread.currentThread();
    cancelExec.schedule(new Runnable() {
        public void run() { taskThread.interrupt(); }
    }, timeout, unit);
    r.run();
}
```



Cette approche est d'une simplicité séduisante, mais elle viole la règle qui énonce que l'on devrait connaître la politique d'interruption d'un thread avant de l'interrompre. `timedRun()` pouvant être appelée à partir de n'importe quel thread, elle ne peut donc connaître la politique d'interruption du thread appelant. Si la tâche se termine avant l'expiration du délai, la tâche d'annulation qui interrompt le thread dans lequel `timedRun()` a rendu la main à son appelant peut continuer à fonctionner en dehors de tout contexte. Nous ne savons pas quel code s'exécutera lorsque cela se passera, mais le résultat ne sera pas correct (il est possible, mais assez compliqué, d'éliminer ce risque en utilisant l'objet `ScheduledFuture` renvoyé par `schedule()` pour annuler la tâche d'annulation).

En outre, si la tâche ne répond pas aux interruptions, `timedRun()` ne se terminera pas tant que la tâche ne s'est pas terminée, ce qui peut être bien après le délai souhaité (voire jamais). Un service d'exécution temporisé qui ne se termine pas après le temps indiqué risque d'irriter ses clients.

Le Listing 7.9 corrige le problème de gestion des exceptions de `aSecondOfPrimes()` et les défauts de la tentative précédente. Le thread créé pour exécuter la tâche peut avoir sa propre politique d'exécution et la méthode `run()` temporisée peut revenir à l'appelant, même si la tâche ne répond pas à l'interruption. Après avoir lancé le thread de la tâche, `timedRun()` exécute un `join` temporisé avec le thread nouvellement créé. Lorsque ce `join` s'est terminé, elle teste si une exception a été lancée à partir de la tâche, auquel cas elle la relance dans le thread qui a appelé `timedRun()`. L'objet `Throwable` sauvegardé est partagé entre les deux threads et est donc déclaré comme `volatile` pour pouvoir être correctement publié du thread de la tâche vers celui de `timedRun()`.

Listing 7.9 : Interruption d'une tâche dans un thread dédié.

```
public static void timedRun(final Runnable r, long timeout,
                           TimeUnit unit)
                           throws InterruptedException {
    class RethrowableTask implements Runnable {
        private volatile Throwable t;
        public void run() {
            try { r.run(); }
            catch (Throwable t) { this.t = t; }
        }
        void rethrow() {
            if (t != null)
                throw launderThrowable(t);
        }
    }
    final RethrowableTask task = new RethrowableTask();
    Thread taskThread = new Thread(task);
    taskThread.start();
    taskThread.join(timeout, unit);
    if (task.t != null)
        throw task.t;
}
```



Listing 7.9 : Interruption d'une tâche dans un thread dédié. (suite)

```
RethrowableTask task = new RethrowableTask();
final Thread taskThread = new Thread(task);
taskThread.start();
cancelExec.schedule(new Runnable() {
    public void run() { taskThread.interrupt(); }
}, timeout, unit);
taskThread.join(unit.toMillis(timeout));
task.rethrow();
}
```

Cette version corrige les problèmes des exemples précédents mais, comme elle repose sur un `join` temporisé, elle partage une lacune de `join` : on ne sait pas si l'on est revenu de son appel parce que le thread s'est terminé normalement ou parce que le délai du `join` a expiré¹.

7.1.5 Annulation avec Future

Nous avons déjà utilisé une abstraction pour gérer le cycle de vie d'une tâche, traiter les exceptions et faciliter l'annulation – `Future`. Si l'on suit le principe général selon lequel il est préférable d'utiliser les classes existantes de la bibliothèque plutôt que construire les siennes, nous pouvons écrire `timedRun()` en utilisant `Future` et le framework d'exécution des tâches.

`ExecutorService.submit()` renvoie un objet `Future` décrivant la tâche et `Future` dispose d'une méthode `cancel()` qui attend un paramètre booléen, `mayInterruptIfRunning`. Cette méthode renvoie une valeur indiquant si la tentative d'annulation a réussi (cette valeur indique seulement si l'on a été capable de délivrer l'interruption, pas si la tâche l'a détectée et y a réagi). Si `mayInterruptIfRunning` vaut `true` et que la tâche soit en cours d'exécution dans un `thread`, celui-ci est interrompu. S'il vaut `false`, cela signifie "n'exécute pas cette tâche si elle n'a pas encore été lancée" – on ne devrait l'utiliser que pour les tâches qui n'ont pas été conçues pour traiter les interruptions.

Comme on ne devrait pas interrompre un `thread` sans connaître sa politique d'interruption, quand peut-on appeler `cancel()` avec un paramètre égal à `true` ? Les threads d'exécution des tâches créés par les implémentations standard de `Executor` utilisant une politique d'interruption qui autorise l'annulation des tâches avec des interruptions, vous pouvez positionner `mayInterruptIfRunning` à `true` lorsque vous annulez des tâches en passant par leurs objets `Future` lorsqu'elles s'exécutent dans un `Executor` standard. Lorsque vous voulez annuler une tâche, vous ne devriez pas interrompre directement un `thread` d'un `pool` car vous ne savez pas quelle tâche s'exécute lorsque la demande d'interruption est délivrée – vous devez passer par le `Future` de la tâche. C'est encore une autre raison

1. Il s'agit d'un défaut de l'API `Thread` car le fait que le `join` se termine avec succès ou non a des conséquences sur la visibilité de la mémoire dans le modèle mémoire de Java, or `join` ne renvoie rien qui puisse indiquer s'il a réussi ou non.

pour laquelle il faut coder les tâches pour qu'elles traitent les interruptions comme des demandes d'annulation : elles peuvent ainsi être annulées *via leur Future*.

Le Listing 7.10 présente une version de `timedRun()` qui soumet la tâche à un Executor Service et récupère le résultat par un appel `Future.get()` temporisé. Si `get()` se termine avec une exception `TimeoutException`, la tâche est annulée *via son Future* (pour simplifier le codage, cette version appelle sans condition `Future.cancel()` dans un bloc `finally` afin de profiter du fait que l'annulation d'une tâche déjà terminée n'a aucun effet). Si le calcul sous-jacent lance une exception avant l'annulation, celle-ci est relancée par `timedRun()`, ce qui est le moyen le plus pratique pour que l'appelant puisse traiter cette exception. Le Listing 7.10 illustre également une autre pratique : l'annulation des tâches du résultat desquelles on n'a plus besoin (cette technique était également utilisée dans les Listing 6.13 et 6.16).

Listing 7.10 : Annulation d'une tâche avec Future.

```
public static void timedRun(Runnable r, long timeout, TimeUnit unit)
    throws InterruptedException {
    Future<?> task = taskExec.submit(r);
    try {
        task.get(timeout, unit);
    } catch (TimeoutException e) {
        // La tâche sera annulée en dessous
    } catch (ExecutionException e) {
        // L'exception lancée dans la tâche est relancée
        throw launderThrowable (e.getCause());
    } finally {
        // Sans effet si la tâche s'est déjà terminée
        task.cancel(true); // interruption si la tâche s'exécute
    }
}
```

Lorsque `Future.get()` lance une exception `InterruptedException` ou `TimeoutException` et que l'on sait que le résultat n'est plus nécessaire au programme, on annule la tâche avec `Future.cancel()`.

7.1.6 Méthodes bloquantes non interruptibles

De nombreuses méthodes bloquantes de la bibliothèque répondent aux interruptions en se terminant précocément et en lançant une `InterruptedException`, ce qui facilite la création de tâches réactives aux annulations. Cependant, toutes les méthodes ou tous les mécanismes bloquants ne répondent pas aux interruptions ; si un thread est bloqué en attente d'une E/S synchrone sur une socket ou en attente d'un verrou interne, l'interruption ne fera que positionner son indicateur d'interruption. Nous pouvons parfois convaincre les threads bloqués dans des activités non interruptibles de se terminer par un moyen ressemblant aux interruptions, mais cela nécessite de savoir plus précisément pourquoi ils sont bloqués.

- **E/S sockets synchrones de java.io.** C'est une forme classique d'E/S bloquantes dans les applications serveur lorsqu'elles lisent ou écrivent dans une socket. Malheureusement, les méthodes `read()` et `write()` de `InputStream` et `OutputStream` ne répondent pas aux interruptions ; cependant, la fermeture de la socket sous-jacente forcera tout thread bloqué dans `read()` ou `write()` à lancer une `SocketException`.
- **E/S synchrones de java.nio.** L'interruption d'un thread en attente d'un `InterruptibleChannel` le force à lancer une exception `ClosedByInterruptException` et à fermer le canal (tous les autres threads bloqués sur ce canal lanceront également `ClosedByInterruptException`). La fermeture d'un `InterruptibleChannel` force les threads bloqués dans des opérations sur ce canal à lancer `AsynchronousCloseException`. La plupart des `Channel` standard implémentent `InterruptibleChannel`.
- **E/S asynchrones avec Selector.** Si un thread est bloqué dans `Selector.select()` (dans `java.nio.channels`), un appel à `close()` le force à se terminer prématièrement.
- **Acquisition d'un verrou.** Si un thread est bloqué en attente d'un verrou interne, vous ne pouvez pas l'empêcher de s'assurer qu'il finira par prendre le verrou et de progresser suffisamment pour attirer son attention d'une autre façon. Cependant, les classes `Lock` explicites disposent de la méthode `lockInterruptibly()`, qui vous permet d'attendre un verrou tout en pouvant répondre aux interruptions – voir le Chapitre 13.

La classe `ReaderThread` du Listing 7.11 présente une technique pour encapsuler les annulations non standard. Elle gère une connexion par socket simple, lit dans la socket de façon synchrone et passe à `processBuffer()` les données qu'elle a reçues. Pour faciliter la terminaison d'une connexion utilisateur ou l'arrêt du serveur, `ReaderThread` redéfinit `interrupt()` pour qu'elle délivre une interruption standard et ferme la socket sous-jacente. L'interruption d'un `ReaderThread` arrête donc ce qu'il était en train de faire, qu'il soit bloqué dans `read()` ou dans une méthode bloquante interruptible.

Listing 7.11 : Encapsulation des annulations non standard dans un thread par redéfinition de `interrupt()`.

```
public class ReaderThread extends Thread {  
    private final Socket socket;  
    private final InputStream in;  
  
    public ReaderThread(Socket socket) throws IOException {  
        this.socket = socket;  
        this.in = socket.getInputStream();  
    }  
  
    public void interrupt() {  
        try {  
            socket.close();  
        }  
        catch (IOException ignored) { }  
        finally {  
    }
```

```
        super.interrupt();
    }
}

public void run() {
    try {
        byte[] buf = new byte[BUFSZ];
        while (true) {
            int count = in.read(buf);
            if (count < 0)
                break;
            else if (count > 0)
                processBuffer(buf, count);
        }
    } catch (IOException e) { /* Permet au thread de se terminer */ }
}
}
```

7.1.7 Encapsulation d'une annulation non standard avec newTaskFor()

La technique utilisée par ReaderThread pour encapsuler une annulation non standard peut être améliorée en utilisant la méthode de rappel newTaskFor(), ajoutée à Thread PoolExecutor depuis Java 6. Lorsqu'un Callable est soumis à un ExecutorService, submit() renvoie un Future pouvant servir à annuler la tâche. newTaskFor() est une méthode fabrique qui crée le Future représentant la tâche ; elle renvoie un Runnable Future, une interface étendant à la fois Future et Runnable (et implémentée par Future Task).

Vous pouvez redéfinir Future.cancel() pour personnaliser la tâche Future. Personnaliser le code d'annulation permet d'inscrire dans un fichier journal des informations sur l'annulation, par exemple, et vous pouvez également en profiter pour annuler des activités qui ne répondent pas aux interruptions. Tout comme ReaderThread encapsule l'annulation des threads qui utilisent des sockets en redéfinissant *interrupt()*, vous pouvez faire de même pour les tâches en redéfinissant Future.cancel().

Le Listing 7.12 définit une interface CancellableTask qui étend Callable en lui ajoutant une méthode cancel() et une méthode fabrique newTask() pour créer un objet Runnable Future. CancellingExecutor étend ThreadPoolExecutor et redéfinit newTaskFor() pour qu'une CancellableTask puisse créer son propre Future.

SocketUsingTask implémente CancellableTask et définit Future.cancel() pour qu'elle ferme la socket et appelle super.cancel(). Si une SocketUsingTask est annulée via son Future, la socket est fermée et le thread qui s'exécute est interrompu. Ceci augmente la réactivité de la tâche à l'annulation : non seulement elle peut appeler en toute sécurité des méthodes bloquantes interruptibles tout en restant réceptive à l'annulation, mais elle peut également appeler des méthodes d'E/S bloquantes sur des sockets.

Listing 7.12 : Encapsulation des annulations non standard avec newTaskFor().

```

public interface CancellableTask <T> extends Callable<T> {
    void cancel();
    RunnableFuture<T> newTask();
}

@ThreadSafe
public class CancellingExecutor extends ThreadPoolExecutor {
    ...
    protected<T> RunnableFuture <T> newTaskFor(Callable<T> callable) {
        if (callable instanceof CancellableTask )
            return ((CancellableTask <T>) callable).newTask();
        else
            return super.newTaskFor(callable);
    }
}

public abstract class SocketUsingTask <T>
    implements CancellableTask <T> {
    @GuardedBy("this") private Socket socket;

    protected synchronized void setSocket(Socket s) { socket = s; }

    public synchronized void cancel() {
        try {
            if (socket != null)
                socket.close();
        } catch (IOException ignored) { }
    }

    public RunnableFuture <T> newTask() {
        return new FutureTask<T>(this) {
            public boolean cancel(boolean mayInterruptIfRunning ) {
                try {
                    SocketUsingTask .this.cancel();
                } finally {
                    return super.cancel(mayInterruptIfRunning );
                }
            }
        };
    }
}

```

7.2 Arrêt d'un service reposant sur des threads

Les applications créent souvent des services qui utilisent des threads – comme des pools de threads – et la durée de vie de ces services est généralement plus longue que celle de la méthode qui les a créés. Si l'application doit s'arrêter en douceur, il faut mettre fin aux threads appartenant à ces services. Comme il n'y a pas moyen d'imposer à un thread de s'arrêter, il faut les persuader de se terminer d'eux-mêmes.

Les bonnes pratiques d'encapsulation enseignent qu'il ne faut pas manipuler un thread – l'interrompre, modifier sa priorité, etc. – qui ne nous appartient pas. L'API des threads ne définit pas formellement la propriété d'un thread : un thread est représenté par un objet `Thread` qui peut être librement partagé, exactement comme n'importe quel autre objet. Cependant, il semble raisonnable de penser qu'un thread a un propriétaire,

qui est généralement la classe qui l'a créé. Un pool de threads est donc le propriétaire de ses threads et, s'ils doivent être interrompus, c'est le pool qui devrait s'en occuper.

Comme pour tout objet encapsulé, la propriété d'un thread n'est pas transitive : l'application peut posséder le service et celui-ci peut posséder les threads, mais l'application ne possède pas les threads et ne peut donc pas les arrêter directement. Le service doit donc fournir des *méthodes de cycle de vie* pour se terminer lui-même et arrêter également les threads qu'il possède ; l'application peut alors arrêter le service, qui se chargera lui-même d'arrêter les threads. La classe `ExecutorService` fournit les méthodes `shutdown()` et `shutdownNow()` ; les autres services possédant des threads devraient proposer un mécanisme d'arrêt similaire.

Fournissez des méthodes de cycle de vie à chaque fois qu'un service possédant des threads a une durée de vie supérieure à celle de la méthode qui l'a créé.

7.2.1 Exemple : service de journalisation

La plupart des applications serveur écrivent dans un journal, ce qui peut être aussi simple qu'insérer des instructions `println()` dans le code. Les classes de flux comme `PrintWriter` étant thread-safe, cette approche simple ne nécessiterait aucune synchronisation explicite¹. Cependant, comme nous le verrons dans la section 11.6, une journalisation en ligne peut avoir un certain coût en termes de performances pour les applications à fort volume. Une autre possibilité consiste à mettre les messages du journal dans une file d'attente pour qu'ils soient traités par un autre thread.

La classe `LogWriter` du Listing 7.13 montre un service simple de journalisation dans lequel l'activité d'inscription dans le journal a été déplacée dans un thread séparé. Au lieu que le thread qui produit le message l'écrive directement dans le flux de sortie, `LogWriter` le passe à un thread d'écriture *via* une `BlockingQueue`. Il s'agit donc d'une conception de type "plusieurs producteurs, un seul consommateur" où les activités productrices créent les messages et l'activité consommatrice les écrit. Si le thread consommateur va moins vite que les producteurs, la `BlockingQueue` bloquera ces derniers jusqu'à ce que le thread d'écriture dans le journal rattrape son retard.

1. Si le même message se décompose en plusieurs lignes, vous pouvez également avoir besoin d'un verrouillage côté client pour empêcher un entrelacement indésirable des messages des différents threads. Si deux threads inscrivent, par exemple, des traces d'exécution sur plusieurs lignes dans le même flux avec une instruction `println()` par ligne, le résultat serait entrelacé aléatoirement et pourrait donner une seule grande trace inexploitable.

Listing 7.13 : Service de journalisation producteur-consommateur sans support de l'arrêt.

```

public class LogWriter {
    private final BlockingQueue <String> queue;
    private final LoggerThread logger;

    public LogWriter(Writer writer) {
        this.queue = new LinkedBlockingQueue <String>(CAPACITY);
        this.logger = new LoggerThread(writer);
    }

    public void start() { logger.start(); }

    public void log(String msg) throws InterruptedException {
        queue.put(msg);
    }

    private class LoggerThread extends Thread {
        private final PrintWriter writer;
        ...
        public void run() {
            try {
                while (true)
                    writer.println(queue.take());
            } catch(InterruptedException ignored) {
            } finally {
                writer.close();
            }
        }
    }
}

```



Pour qu'un service comme `LogWriter` soit utile en production, nous avons besoin de pouvoir terminer le thread d'écriture dans le journal sans qu'il empêche l'arrêt normal de la JVM. Arrêter ce thread est assez simple puisqu'il appelle `take()` en permanence et que cette méthode répond aux interruptions ; si l'on modifie ce thread pour qu'il s'arrête lorsqu'il capture `InterruptedException`, son interruption arrêtera le service.

Cependant, faire simplement en sorte que le thread d'écriture se termine n'est pas un mécanisme d'arrêt très satisfaisant. En effet, cet arrêt brutal supprime les messages qui pourraient être en attente d'écriture et, ce qui est encore plus important, les threads producteurs bloqués parce que la file est pleine *ne seront jamais débloqués*. L'annulation d'une activité producteur-consommateur exige d'annuler à la fois les producteurs et les consommateurs. Interrrompre le thread d'écriture règle le problème du consommateur mais, les producteurs n'étant pas ici des threads dédiés, il est plus difficile de les annuler.

Une autre approche pour arrêter `LogWriter` consiste à positionner un indicateur "arrêt demandé" pour empêcher que d'autres messages soient soumis, comme on le montre dans le Listing 7.14. Lorsqu'il est prévenu de cette demande, le consommateur peut alors vider la file en inscrivant dans le journal les messages en attente et en débloquant les éventuels producteurs bloqués dans `log()`. Cependant, cette approche a des situations de compétition qui la rendent non fiable. L'implémentation du journal est une séquence *tester-puis-agir* : les producteurs pourraient constater que le service n'a pas encore été arrêté et continuer à placer des messages dans la file après l'arrêt avec le risque de se

trouver à nouveau bloqués indéfiniment dans `log()`. Il existe des astuces réduisant cette probabilité (en faisant, par exemple, attendre quelques secondes le consommateur avant de déclarer la file comme épuisée), mais elles changent non pas le problème fondamental, mais uniquement la probabilité que ce problème survienne.

Listing 7.14 : Moyen non fiable d'ajouter l'arrêt au service de journalisation.

```
public void log(String msg) throws InterruptedException {
    if (!shutdownRequested)
        queue.put(msg);
    else
        throw new IllegalStateException("logger is shut down");
}
```



Pour fournir un arrêt fiable à `LogWriter`, il faut régler le problème de la situation de compétition, ce qui signifie qu'il faut que la soumission d'un nouveau message soit *atomique*. Cependant, nous ne voulons pas détenir un verrou pendant que l'on place le message dans la file car `put()` pourrait se bloquer. Nous pouvons, en revanche, vérifier de façon atomique qu'il y a eu demande d'arrêt et incrémenter un compteur, afin de nous "réserver" le droit de soumettre un message, comme dans le Listing 7.15.

Listing 7.15 : Ajout d'une annulation fiable à `LogWriter`.

```
public class LogService {
    private final BlockingQueue <String> queue;
    private final LoggerThread loggerThread;
    private final PrintWriter writer;
    @GuardedBy("this") private boolean isShutdown;
    @GuardedBy("this") private int reservations;

    public void start() { loggerThread.start(); }

    public void stop() {
        synchronized (this) { isShutdown = true; }
        loggerThread.interrupt();
    }

    public void log(String msg) throws InterruptedException {
        synchronized (this) {
            if (isShutdown)
                throw new IllegalStateException (...);
            ++reservations;
        }
        queue.put(msg);
    }

    private class LoggerThread extends Thread {
        public void run() {
            try {
                while (true) {
                    try {
                        synchronized (LogService.this) {
                            if (isShutdown && reservations == 0)
                                break;
                        }
                        String msg = queue.take();
                        synchronized (LogService.this) { --reservations ; }
                    }
                }
            } catch (InterruptedException e) {
                // ...
            }
        }
    }
}
```

Listing 7.15 : Ajout d'une annulation fiable à LogWriter. (suite)

```

        writer.println(msg);
    } catch (InterruptedException e) { /* réessaie */ }
}
} finally {
    writer.close();
}
}
}
}
}
```

7.2.2 Méthodes d'arrêt de ExecutorService

Dans la section 6.2.4, nous avons vu que `ExecutorService` offrait deux moyens de s'arrêter : un arrêt en douceur avec `shutdown()` et un arrêt brutal avec `shutdownNow()`. Dans ce dernier cas, `shutdownNow()` renvoie la liste des tâches qui n'ont pas encore commencé après avoir tenté d'annuler toutes les tâches qui s'exécutent.

Ces deux options de terminaison sont des compromis entre sécurité et réactivité : la terminaison brutale est plus rapide mais plus risquée car les tâches peuvent être interrompues au milieu de leur exécution, tandis que la terminaison normale est plus lente mais plus sûre puisque `ExecutorService` ne s'arrêtera pas tant que toutes les tâches en attente n'auront pas été traitées. Les autres services utilisant des threads devraient fournir un choix équivalent pour leurs modes d'arrêt.

Les programmes simples peuvent s'en sortir en lançant et en arrêtant un `ExecutorService` global à partir de `main()`. Ceux qui sont plus sophistiqués encapsuleront sûrement un `ExecutorService` derrière un service de plus haut niveau fournissant ses propres méthodes de cycle de vie, comme le fait la variante de `LogService` dans le Listing 7.16, qui délègue la gestion de ses propres threads à un `ExecutorService`. Cette encapsulation étend la chaîne de propriété de l'application au service et au thread en ajoutant un autre lien ; chaque membre de cette chaîne gère le cycle de vie des services ou des threads qui lui appartiennent.

Listing 7.16 : Service de journalisation utilisant un ExecutorService.

```

public class LogService {
    private final ExecutorService exec = newSingleThreadExecutor ();
    ...
    public void start() { }
    public void stop() throws InterruptedException {
        try {
            exec.shutdown();
            exec.awaitTermination(TIMEOUT, UNIT);
        } finally {
            writer.close();
        }
    }
    public void log(String msg) {
        try {
            exec.execute(new WriteTask(msg));
        } catch (RejectedExecutionException ignored) { }
    }
}
```

7.2.3 Pilules empoisonnées

Un autre moyen de convaincre un service producteur-consommateur de s'arrêter consiste à utiliser une *pilule empoisonnée*, c'est-à-dire un objet reconnaissable placé dans la file d'attente, qui signifie "quand tu me prends, arrête-toi". Avec une file FIFO, les pilules empoisonnées garantissent que les consommateurs finiront leur travail sur leur file avant de s'arrêter, puisque tous les travaux soumis avant la pilule seront récupérés avant elle ; les producteurs ne devraient pas soumettre de nouveaux travaux après avoir mis la pilule dans la file. Dans les Listings 7.17, 7.18 et 7.19, la classe IndexingService montre une version "un producteur-un consommateur" de l'exemple d'indexation du disque que nous avions présenté dans le Listing 5.8. Elle utilise une pilule empoisonnée pour arrêter le service.

Listing 7.17 : Arrêt d'un service avec une pilule empoisonnée.

```
public class IndexingService {
    private static final File POISON = new File("");
    private final IndexerThread consumer = new IndexerThread();
    private final CrawlerThread producer = new CrawlerThread();
    private final BlockingQueue <File> queue;
    private final FileFilter fileFilter;
    private final File root;

    class CrawlerThread extends Thread { /* Listing 7.18 */ }
    class IndexerThread extends Thread { /* Listing 7.19 */ }

    public void start() {
        producer.start();
        consumer.start();
    }

    public void stop() { producer.interrupt(); }

    public void awaitTermination() throws InterruptedException {
        consumer.join();
    }
}
```

Listing 7.18 : Thread producteur pour IndexingService.

```
public class CrawlerThread extends Thread {
    public void run() {
        try {
            crawl(root);
        } catch (InterruptedException e) { /* ne fait rien */ }
        finally {
            while (true) {
                try {
                    queue.put(POISON);
                    break;
                } catch (InterruptedException e1) { /* réessaie */ }
            }
        }
    }

    private void crawl(File root) throws InterruptedException {
        ...
    }
}
```

Listing 7.19 : Thread consommateur pour IndexingService.

```
public class IndexerThread extends Thread {
    public void run() {
        try {
            while (true) {
                File file = queue.take();
                if (file == POISON)
                    break;
                else
                    indexFile(file);
            }
        } catch (InterruptedException consumed) { }
    }
}
```

Les pilules empoisonnées ne conviennent que lorsque l'on connaît le nombre de producteurs et de consommateurs. L'approche utilisée par `IndexingService` peut être étendue à plusieurs producteurs : chacun d'eux place une pilule dans la file et le consommateur ne s'arrête que lorsqu'il a reçu $N_{\text{producteurs}}$ pilules. Elle peut également être étendue à plusieurs consommateurs : chaque producteur place alors $N_{\text{consommateurs}}$ pilules dans la file, bien que cela puisse devenir assez lourd avec un grand nombre de producteurs et de consommateurs. Les pilules empoisonnées ne fonctionnent correctement qu'avec des files non bornées.

7.2.4 Exemple : un service d'exécution éphémère

Si une méthode doit traiter un certain nombre de tâches et qu'elle ne se termine que lorsque toutes ces tâches sont terminées, la gestion du cycle de vie du service peut être simplifiée en utilisant un `Executor` privé dont la durée de vie est limitée par cette méthode (dans ces situations, les méthodes `invokeAll()` et `invokeAny()` sont souvent utiles).

La méthode `checkMail()` du Listing 7.20 teste en parallèle si du nouveau courrier est arrivé sur un certain nombre d'hôtes. Elle crée un exécuteur privé et soumet une tâche pour chaque hôte : puis elle arrête l'exécuteur et attend la fin, qui intervient lorsque toutes les tâches de vérification du courrier se sont terminées¹.

Listing 7.20 : Utilisation d'un Executor privé dont la durée de vie est limitée à un appel de méthode.

```
boolean checkMail(Set<String> hosts, long timeout, TimeUnit unit)
    throws InterruptedException {
    ExecutorService exec = Executors.newCachedThreadPool ();
    final AtomicBoolean hasNewMail = new AtomicBoolean(false);
    try {
        for (final String host : hosts)
            exec.execute(new Runnable() {
                public void run() {
                    if (checkMail(host))
                        hasNewMail.set(true);
                }
            });
    } catch (RejectedExecutionException e) {
        // ...
    }
    exec.shutdown();
    if (!exec.awaitTermination(timeout, unit))
        return false;
    return hasNewMail.get();
}
```

1. On utilise un `AtomicBoolean` au lieu d'un booléen `volatile` car, pour accéder à l'indicateur `hasNewMail` à partir du `Runnable` interne, celui-ci devrait être `final`, ce qui empêcherait de le modifier.

```

        }
    });
} finally {
    exec.shutdown();
    exec.awaitTermination(timeout, unit);
}
return hasNewMail.get();
}

```

7.2.5 Limitations de shutdownNow()

Lorsqu'un ExecutorService est arrêté brutalement par un appel à `shutdownNow()`, il tente d'annuler les tâches en cours et renvoie une liste des tâches qui ont été soumises mais jamais lancées afin de pouvoir les inscrire dans un journal ou les sauvegarder pour un traitement ultérieur¹.

Cependant, il n'existe pas de méthode générale pour savoir quelles sont les tâches qui ont été lancées et qui ne sont pas encore terminées. Ceci signifie que vous ne pouvez pas connaître l'état des tâches en cours au moment de l'arrêt, sauf si ces tâches effectuent elles-mêmes une sorte de pointage. Pour connaître les tâches qui ne sont pas encore terminées, vous devez savoir non seulement quelles tâches n'ont pas démarré mais également celles qui étaient en cours lorsque l'exécuteur a été arrêté².

La classe `TrackingExecutor` du Listing 7.21 montre une technique permettant de déterminer les tâches en cours au moment de l'arrêt. En encapsulant un ExecutorService et en modifiant `execute()` (et `submit()`, mais ce n'est pas montré ici) pour qu'elle se rappelle les tâches qui ont été annulées après l'arrêt, `TrackingExecutor` peut savoir quelles tâches ont été lancées mais ne se sont pas terminées normalement. Lorsque l'exécuteur s'est terminé, `getCancelledTasks()` renvoie la liste des tâches annulées. Pour que cette technique fonctionne, les tâches doivent préserver l'indicateur d'interruption du thread lorsqu'elles se terminent, ce que font toutes les tâches correctes de toute façon.

Listing 7.21 : ExecutorService mémorisant les tâches annulées après l'arrêt.

```

public class TrackingExecutor extends AbstractExecutorService {
    private final ExecutorService exec;
    private final Set<Runnable> tasksCancelledAtShutdown =
        Collections.synchronizedSet (new HashSet<Runnable>());
    ...
    public List<Runnable> getCancelledTasks () {
        if (!exec.isTerminated())
            throw new IllegalStateException (...);
        return new ArrayList<Runnable>(tasksCancelledAtShutdown);
    }
}

```

1. Les objets `Runnable` renvoyés par `shutdownNow()` peuvent ne pas être les mêmes que ceux qui ont été soumis à l'ExecutorService : il peut s'agir d'instances enveloppées de ces tâches.

2. Malheureusement, il n'existe pas d'option d'arrêt permettant de renvoyer à l'appelant les tâches qui n'ont pas encore démarré et de laisser terminer celles qui sont en cours. Cette option permettrait d'éliminer cet état intermédiaire incertain.

```

public void execute(final Runnable runnable) {
    exec.execute(new Runnable() {
        public void run() {
            try {
                runnable.run();
            } finally {
                if (isShutdown() && Thread.currentThread().isInterrupted())
                    tasksCancelledAtShutdown.add(runnable);
            }
        }
    });
}
// Délègue à exec les autres méthodes de ExecutorService
}

```

La classe WebCrawler du Listing 7.22 met en œuvre TrackingExecutor. Le travail d'un robot web est souvent une tâche sans fin : s'il doit être arrêté, il est préférable de sauvegarder son état courant afin de pouvoir le relancer plus tard. CrawlTask fournit une méthode getPage() qui identifie la page courante. Lorsque le robot est arrêté, les tâches qui n'ont pas été lancées et celles qui ont été annulées sont analysées et leurs URL sont enregistrées afin que les tâches de parcours de ces URL puissent être ajoutées à la file lorsque le robot repart.

Listing 7.22 : Utilisation de TrackingExecutorService pour mémoriser les tâches non terminées afin de les relancer plus tard.

```

public abstract class WebCrawler {
    private volatile TrackingExecutor exec;
    @GuardedBy("this")
    private final Set<URL> urlsToCrawl = new HashSet<URL>();
    ...
    public synchronized void start() {
        exec = new TrackingExecutor (
            Executors.newCachedThreadPool());
        for (URL url : urlsToCrawl) submitCrawlTask (url);
        urlsToCrawl.clear();
    }

    public synchronized void stop() throws InterruptedException {
        try {
            saveUncrawled(exec.shutdownNow());
            if (exec.awaitTermination(TIMEOUT, UNIT))
                saveUncrawled(exec.getCancelledTasks());
        } finally {
            exec = null;
        }
    }

    protected abstract List<URL> processPage(URL url);

    private void saveUncrawled(List<Runnable> uncrawled) {
        for (Runnable task : uncrawled)
            urlsToCrawl.add(((CrawlTask) task).getPage());
    }
    private void submitCrawlTask (URL u) {
        exec.execute(new CrawlTask(u));
    }
    private class CrawlTask implements Runnable {
        private final URL url;
        ...
    }
}

```

```
public void run() {
    for (URL link : processPage(url)) {
        if (Thread.currentThread().isInterrupted())
            return;
        submitCrawlTask (link);
    }
}
public URL getPage() { return url; }
}
```

TrackingExecutor a une situation de compétition inévitable qui pourrait produire des faux positifs, c'est-à-dire des tâches identifiées comme annulées alors qu'elles sont en fait terminées. Cette situation se produit lorsque le pool de threads est arrêté entre le moment où la dernière instruction de la tâche s'exécute et celui où le pool considère que la tâche est terminée. Ceci n'est pas un problème si les tâches sont *idempotentes* (les exécuter deux fois aura le même effet que ne les exécuter qu'une seule fois), ce qui est généralement le cas dans un robot web. Dans le cas contraire, l'application qui récupère les tâches annulées doit être au courant de ce risque et prête à gérer les faux positifs.

7.3 Gestion de la fin anormale d'un thread

On s'aperçoit immédiatement qu'une application monothread en mode console se termine à cause d'une exception non capturée – le programme s'arrête et affiche une trace de la pile d'appels qui est très différente de la sortie d'un programme normal. L'échec d'un thread dans une application concurrente, en revanche, n'est pas toujours aussi évident. La trace de la pile peut s'afficher sur la console mais on peut très bien ne pas regarder la console. En outre, lorsqu'un thread échoue, l'application peut sembler continuer à fonctionner et cet échec peut ne pas être remarqué. Heureusement, il existe des moyens de détecter et d'empêcher les "fuites" de threads d'une application.

La cause principale de la mort prématuée d'un thread est `RuntimeException`. Ces exceptions indiquant une erreur de programmation ou un autre problème irrécupérable, elles ne sont généralement pas capturées et se propagent donc jusqu'au sommet de la pile des appels, où le comportement par défaut consiste à afficher une trace de cette pile sur la console et à mettre fin au thread.

Selon le rôle du thread dans l'application, les conséquences de sa mort anormale peuvent être bénignes ou désastreuses. Perdre un thread d'un pool de threads peut avoir des conséquences sur les performances, mais une application qui s'exécute correctement avec un pool de 50 threads fonctionnera sûrement tout aussi bien avec un pool de 49 threads. En revanche, perdre le thread de répartition des événements dans une application graphique ne passera pas inaperçu – l'application ne pourra plus traiter les événements et l'interface se figera. La classe `OutOfTime` au Listing 6.9 montrait une conséquence grave de la fuite d'un thread : le service représenté par le `Timer` était définitivement hors d'état.

Quasiment n'importe quel code peut lancer une exception `RuntimeException`. À chaque fois que l'on appelle une autre méthode, on suppose qu'elle se terminera normalement ou lancera l'une des exceptions contrôlées déclarées dans sa signature. Moins l'on connaît le code qui est appelé, moins on doit avoir confiance en son comportement.

Les threads de traitement des tâches, comme ceux d'un pool de threads ou le thread de répartition des événements Swing, passent leur existence à appeler du code inconnu *via* une barrière d'abstraction comme `Runnable` : ils devraient donc être très méfiants sur le comportement du code qu'ils appellent. Il serait désastreux qu'un service comme le thread des événements Swing échoue uniquement parce qu'un gestionnaire d'événement mal écrit a lancé une exception `NullPointerException`. En conséquence, ces services devraient appeler les tâches dans un bloc `try-catch` pour garantir que le framework sera prévenu si le thread se termine anormalement et qu'il pourra prendre les mesures nécessaires. C'est l'une des rares fois où l'on peut envisager de capturer `RuntimeException` – lorsque l'on appelle un code non fiable au moyen d'une abstraction comme `Runnable`¹.

Le Listing 7.23 montre une façon de structurer un thread dans un pool de threads. Si une tâche lance une exception non contrôlée, il autorise le thread à mourir mais pas avant d'avoir prévenu le framework de sa mort. Le framework peut alors remplacer ce thread par un nouveau ou ne pas le faire parce que le pool s'arrête ou parce qu'il y a déjà suffisamment de threads pour combler la demande courante. `ThreadPoolExecutor` et Swing utilisent cette technique pour s'assurer qu'une tâche qui se comporte mal n'empêchera pas l'exécution des tâches suivantes. Lorsque vous écrivez une classe `thread` qui exécute les tâches qui lui sont soumises ou que vous appelez un code externe non vérifié (comme des greffons chargés dynamiquement), utilisez l'une de ces approches pour empêcher les tâches ou les greffons mal écrits de tuer le thread qui les a appelés.

Listing 7.23 : Structure typique d'un thread d'un pool de threads.

```
public void run() {
    Throwable thrown = null;
    try {
        while (!isInterrupted())
            runTask(getTaskFromWorkQueue());
    } catch (Throwable e) {
        thrown = e;
    } finally {
        threadExited(this, thrown);
    }
}
```

1. La sécurité de cette technique fait débat : lorsqu'un thread lance une exception non contrôlée, toute l'application peut être compromise. Mais l'autre alternative – arrêter toute l'application – n'est généralement pas envisageable.

7.3.1 Gestionnaires d'exceptions non capturées

La section précédente a proposé une approche proactive du problème des exceptions non contrôlées. L'API des threads fournit également le service `UncaughtExceptionHandler`, qui permet de détecter la mort d'un thread à cause d'une exception non capturée. Les deux approches sont complémentaires : prises ensemble, elles fournissent une défense en profondeur contre la fuite de threads.

Lorsqu'un thread se termine à cause d'une exception non capturée, la JVM signale cet événement à un objet `UncaughtExceptionHandler` fourni par l'application (voir Listing 7.24) ; si aucun gestionnaire n'existe, le comportement par défaut consiste à afficher la trace de la pile sur `System.err`¹.

Listing 7.24 : Interface `UncaughtExceptionHandler`.

```
public interface UncaughtExceptionHandler {  
    void uncaughtException(Thread t, Throwable e);  
}
```

Ce que doit faire le gestionnaire d'une exception non capturée dépend des exigences de la qualité du service. La réponse la plus fréquente consiste à écrire un message d'erreur et une trace de la pile dans le journal de l'application, comme le montre le Listing 7.25. Les gestionnaires peuvent également agir plus directement, en essayant de relancer le thread, par exemple, ou en éteignant l'application, en appelant un opérateur ou toute autre action corrective ou de diagnostic.

Listing 7.25 : `UncaughtExceptionHandler`, qui inscrit l'exception dans le journal.

```
public class UEHLogger implements Thread.UncaughtExceptionHandler {  
    public void uncaughtException(Thread t, Throwable e) {  
        Logger logger = Logger.getAnonymousLogger ();  
        logger.log(Level.SEVERE,  
                   "Thread terminated with exception: " + t.getName(), e);  
    }  
}
```

1. Avant Java 5.0, la seule façon de contrôler le `UncaughtExceptionHandler` consistait à hériter de `ThreadGroup`. À partir de Java 5.0, on peut obtenir un `UncaughtExceptionHandler` thread par thread avec `Thread.setUncaughtExceptionHandler()` et l'on peut également configurer le `UncaughtExceptionHandler` par défaut avec `Thread.setDefaultUncaughtExceptionHandler()`. Cependant, un seul de ces gestionnaires est appelé – la JVM recherche d'abord un gestionnaire par thread, puis un gestionnaire pour un `ThreadGroup`. L'implémentation du gestionnaire par défaut dans `ThreadGroup` délègue le travail à son groupe de thread parent et ainsi de suite jusqu'à ce que l'un des gestionnaires de `ThreadGroup` traite l'exception non capturée ou la fasse remonter au groupe situé le plus haut. Le gestionnaire du groupe de premier niveau délègue à son tour le traitement au gestionnaire par défaut du système (s'il en existe un car il n'y en pas par défaut) et, sinon, affiche la trace de la pile sur la console.

Dans les applications qui s'exécutent en permanence, utilisez toujours des gestionnaires d'exceptions non capturées pour que les threads puissent au moins inscrire l'exception dans le journal.

Pour configurer un `UncaughtExceptionHandler` pour les threads de pool, on fournit un `ThreadFactory` au constructeur de `ThreadPoolExecutor` (comme pour toute manipulation de threads, seul le propriétaire du thread peut changer son `UncaughtExceptionHandler`). Les pools de threads standard permettent à une exception non capturée dans une tâche de mettre fin à un thread du pool. Cependant, utilisez un bloc `try-finally` pour être prévenu de cet événement afin de pouvoir remplacer le thread car, sans gestionnaire d'exception non capturée ou autre mécanisme de notification d'erreur, les tâches peuvent sembler échouer sans rien dire, ce qui peut être très troublant. Si vous voulez être prévenu qu'une tâche échoue à cause d'une exception, enveloppez la tâche avec un `Runnable` ou un `Callable` qui capture l'exception ou redéfinissez la méthode `afterExecute()` dans `ThreadPoolExecutor`.

Bien que ce soit un peu confus, les exceptions levées depuis le gestionnaire d'exceptions non capturées d'une tâche ne concernent que les tâches lancées par la méthode `execute()` ; pour les tâches soumises avec `submit()`, toute exception lancée, qu'elle soit contrôlée ou non, est considérée comme faisant partie de l'état renvoyé par la tâche. Si une tâche soumise par `submit()` se termine par une exception, celle-ci est relancée par `Future.get()`, enveloppée dans une exception `ExecutionException`.

7.4 Arrêt de la JVM

La machine virtuelle Java (JVM) peut s'arrêter *en douceur* ou *brutalement*. Un arrêt en douceur a lieu lorsque le dernier thread "normal" (non démon) se termine, lorsque quelqu'un a appelé `System.exit()` ou par un autre moyen spécifique à la plate-forme (l'envoi du signal `SIGINT` ou la frappe de `Ctrl+C`, par exemple). Bien qu'il s'agisse du moyen standard et conseillé d'arrêter la JVM, celle-ci peut également être arrêtée brutalement par un appel à `Runtime.halt()` ou en tuant son processus directement à partir du système d'exploitation (en lui envoyant le signal `SIGKILL`, par exemple).

7.4.1 Méthodes d'interception d'un ordre d'arrêt

Dans un arrêt en douceur, la JVM lance d'abord les *hooks*¹ d'arrêt enregistrés, qui sont des threads non lancés et enregistrés avec `Runtime.addShutdownHook()`. La JVM ne garantit pas l'ordre dans lequel ces hooks seront lancés. Si des threads de l'application (démons ou non) sont en cours d'exécution au moment de l'arrêt, ils continueront à

1. N.d.T. : Nous laissons le terme *hook* dans le texte car c'est un terme assez courant. Une traduction possible est "méthode d'interception".

fonctionner en parallèle avec le processus d’arrêt. Lorsque tous les hooks d’arrêt se sont terminés, la JVM peut choisir de lancer les finaliseurs si `runFinalizersOnExit` vaut `true`, puis s’arrête. La JVM ne tente pas d’arrêter ou d’interrompre les threads de l’application qui sont en cours d’exécution au moment de l’arrêt ; ils seront brutalement terminés lorsque la JVM finira par s’arrêter. Si les hooks ou les finaliseurs ne se terminent pas, le processus d’arrêt en douceur "se fige" et la JVM doit alors être arrêtée brutalement. Dans un arrêt brutal, la JVM n’est pas censée faire autre chose que s’arrêter ; les hooks d’arrêt ne s’exécuteront pas.

Ceux-ci doivent être thread-safe : ils doivent utiliser la synchronisation lorsqu’ils accèdent à des données partagées et faire attention à ne pas créer de deadlocks, exactement comme n’importe quel autre code concurrent. En outre, ils ne doivent faire aucune supposition sur l’état de l’application (comme supposer que les autres services se sont déjà arrêtés ou que tous les threads normaux se sont terminés) ni sur la raison pour laquelle la JVM s’arrête : ils doivent donc être codés pour être très prudents. Enfin, ils doivent se terminer le plus vite possible car leur existence retarde l’arrêt de la JVM à un moment où l’utilisateur s’attend à ce qu’elle s’arrête rapidement.

Les hooks d’arrêt peuvent être utilisés pour nettoyer le service ou l’application – pour supprimer les fichiers temporaires ou fermer des ressources qui ne sont pas automatiquement fermées par le système d’exploitation, par exemple. Le Listing 7.26 montre comment la classe `LogService` du Listing 7.16 pourrait enregistrer un hook d’arrêt à partir de sa méthode `start()` pour s’assurer que le fichier journal est fermé à la fin de l’application.

Listing 7.26 : Enregistrement d’un hook d’arrêt pour arrêter le service de journalisation.

```
public void start() {
    Runtime.getRuntime().addShutdownHook(new Thread() {
        public void run() {
            try { LogService.this.stop(); }
            catch (InterruptedException ignored) {}
        }
    });
}
```

Les hooks d’arrêt s’exécutant tous en parallèle, la fermeture du fichier journal pourrait poser des problèmes aux autres hooks qui veulent l’utiliser. Pour éviter cette situation, les hooks ne devraient pas utiliser des services qui peuvent être arrêtés par l’application ou les autres hooks. Une façon de procéder consiste à utiliser un seul hook pour tous les services au lieu d’un hook par service et à lui faire appeler une série d’actions d’arrêt. Cela garantit que ces actions s’exécuteront dans le même thread et évitera les situations de compétition ou les deadlocks entre elles. Cette technique peut d’ailleurs être utilisée avec ou sans hooks d’arrêt car une exécution d’actions en séquence plutôt qu’en parallèle élimine de nombreuses sources potentielles d’erreur. Dans les applications qui gèrent des informations de dépendances entre les services, cette technique permet également de s’assurer que les actions d’arrêt s’exécuteront dans le bon ordre.

7.4.2 Threads démons

Parfois, on souhaite créer un thread qui effectue une fonction utilitaire mais sans que l'existence de ce thread empêche la JVM de s'arrêter. C'est la raison d'être des *threads démons*.

Les threads se classent en deux catégories : les threads normaux et les threads démons. Lorsque la JVM démarre, tous les threads qu'elle crée (comme le ramasse-miettes et les autres threads de nettoyage) sont des threads démons, sauf le thread principal. Tout nouveau thread créé hérite de l'état démon de celui qui l'a créé : par défaut, tous les threads créés par le thread principal sont donc également des threads normaux. Ces deux catégories ne diffèrent que par ce qui se passe lorsqu'ils se terminent. À la fin d'un thread, la JVM effectue un inventaire des threads en cours d'exécution et lance un arrêt en douceur si les seuls qui restent sont des threads démons. Lorsque la JVM s'arrête, tous les threads démons restants sont abandonnés – les blocs `finally` ne sont pas exécutés, les piles ne sont pas libérées et la JVM se contente de s'arrêter.

Les threads démons doivent être utilisés avec modération car peu d'activités peuvent être abandonnées brutalement sans aucun nettoyage. Il est notamment dangereux d'utiliser des threads démons susceptibles d'effectuer des opérations d'E/S. Les threads démons doivent être réservés aux tâches "domestiques", pour supprimer périodiquement des entrées d'un cache mémoire, par exemple.

Les threads démons ne doivent pas se substituer à une bonne gestion du cycle de vie des services dans une application.

7.4.3 Finaliseurs

Le ramasse-miettes permet de récupérer les ressources mémoire lorsqu'elles ne sont plus nécessaires, mais certaines, comme les descripteurs de fichiers ou de sockets, doivent être restituées explicitement au système d'exploitation lorsqu'elles ne sont plus utilisées. Pour cette raison, le ramasse-miettes traite de façon spéciale les objets qui disposent d'une méthode `finalize()` non triviale : après avoir été récupérés par le ramasse-miettes, celui-ci appelle leur méthode `finalize()` pour que les ressources persistantes puissent être libérées.

Les finaliseurs pouvant s'exécuter dans un thread géré par la JVM, tout état auquel un finaliseur accède sera accédé par plusieurs threads et doit donc être synchronisé. Les finaliseurs ne garantissent pas l'instant où ils seront exécutés ni même s'ils le seront, et ils ont un impact non négligeable en terme de performances lorsque leur code est un peu important. Il est également très difficile de les écrire correctement¹. Dans la plupart des cas, la combinaison de blocs `finally` et d'appels explicites à des méthodes `close()`

1. Voir (Boehm, 2005) pour certains des défis qu'il faut relever lorsque l'on écrit des finaliseurs.

est plus efficace que les finaliseurs pour la gestion des ressources ; la seule exception concerne la gestion d'objets qui détiennent des ressources acquises par des méthodes natives. Pour toutes ces raisons, essayez de toujours faire en sorte d'éviter d'écrire ou d'utiliser des classes ayant des finaliseurs (à part celles de la bibliothèque standard) [EJ Item 6].

Évitez les finaliseurs.

Résumé

Les problèmes de fin de vie des tâches, des threads, des services et des applications peuvent compliquer leur conception et leur implémentation. Java ne fournit pas de mécanisme préemptif pour annuler les activités ou terminer les threads ; il offre un mécanisme d'interruption coopératif permettant de faciliter l'annulation, mais c'est à vous de construire les protocoles d'annulation et de les utiliser de façon cohérente. FutureTask et le framework Executor simplifient la construction de tâches et de services annulables.

8

Pools de threads

Le Chapitre 6 a introduit le framework d'exécution des tâches et a montré qu'il simplifie la gestion du cycle de vie des tâches et des threads et qu'il permet de dissocier facilement la soumission des tâches de la politique d'exécution. Le Chapitre 7 a présenté quelques-uns des détails épineux du cycle de vie des services, liés à l'utilisation de ce framework dans les applications réelles. Ce chapitre s'intéresse aux options avancées de configuration et de réglage des pools de threads, décrit les dangers auxquels il faut prendre garde lorsque l'on utilise le framework d'exécution des tâches et fournit quelques exemples plus élaborés de l'utilisation d'Executor.

8.1 Couplage implicite entre les tâches et les politiques d'exécution

Nous avons prétendu plus haut que le framework Executor séparait la soumission des tâches de leur exécution. Comme beaucoup de tentatives de découpler des processus complexes, notre affirmation était un peu exagérée. Bien que le framework Executor offre une souplesse non négligeable pour la spécification et la modification des politiques d'exécution, toutes les tâches ne sont pas compatibles avec toutes les politiques d'exécution. Les types de tâches qui exigent des politiques d'exécution spécifiques sont :

- **Les tâches dépendantes.** La plupart des tâches qui se comportent correctement sont *indépendantes* : elles ne dépendent ni du timing, ni du résultat, ni des effets de bord d'autres tâches. Lorsque l'on exécute des tâches indépendantes dans un pool de thread, on peut faire varier librement la taille du pool et sa configuration sans rien affecter d'autre que les performances. En revanche, lorsque l'on soumet à un pool des tâches qui dépendent d'autres tâches, on crée implicitement des contraintes sur la politique d'exécution qui doivent être soigneusement gérées pour éviter des problèmes de vivacité (voir la section 8.1.1).

- **Les tâches qui exploitent le confinement aux threads.** Les exécuteurs monothreads s'engagent plus sur la concurrence que les pools de threads. Comme ils garantissent que les tâches ne s'exécuteront pas en parallèle, il n'y a pas besoin de se soucier de la thread safety du code des tâches. Les objets pouvant être confinés au thread de la tâche, celles conçues pour s'exécuter dans ce thread peuvent donc accéder à ces objets sans synchronisation, même si ces ressources ne sont pas thread-safe. Ceci forme un couplage implicite entre la tâche et la politique d'exécution – les tâches demandent à leur exécuteur d'être monothread¹. En ce cas, si vous remplacez un Executor monothread par un pool de threads, vous risquez de perdre cette thread safety.
- **Les tâches sensibles au temps de réponse.** Les applications graphiques sont sensibles au temps de réponse : les utilisateurs n'apprécient pas qu'il s'écoule un long délai entre un clic sur un bouton et son effet. Soumettre une longue tâche à un exécuteur monothread ou plusieurs longues tâches à un pool ayant peu de threads peut détériorer la réactivité du service géré par cet Executor.
- **Les tâches qui utilisent ThreadLocal.** ThreadLocal autorise chaque thread à posséder sa propre "version" privée d'une variable. Cependant, les exécuteurs peuvent réutiliser les threads en fonction de leurs besoins. Les implémentations standard d'Executor peuvent supprimer les threads inactifs lorsque la demande est faible et en ajouter de nouveaux lorsque la demande s'accroît ; elles peuvent également remplacer un thread par un autre si une tâche lance une exception non contrôlée. Il ne faut donc utiliser ThreadLocal dans les threads de pool que si la valeur locale au thread a une durée de vie limitée à celle d'une tâche ; il ne faut pas s'en servir dans les threads de pool pour transmettre des valeurs entre les tâches.

Les pools de threads fonctionnent mieux lorsque les tâches sont *homogènes* et *indépendantes*. Mélanger des tâches longues et courtes risque en effet d'"embouteiller" le pool, sauf si ce dernier est très grand ; soumettre des tâches dépendant d'autres tâches risque de provoquer un interblocage (*deadlock*), sauf si le pool n'est pas borné. Heureusement, les requêtes adressées aux applications serveur classiques – serveurs web, de courrier, de fichiers – correspondent généralement à ces critères.

Les caractéristiques de certaines tâches nécessitent ou excluent une politique d'exécution précise. Les tâches dépendant d'autres tâches exigent que le pool de threads soit suffisamment important pour que les tâches ne soient jamais placées en attente ni rejetées ; les tâches qui exploitent le confinement au thread exigent une exécution séquentielle. Il faut documenter ces exigences pour que les développeurs qui reprendront le code ne perturbent pas sa thread safety ni sa vivacité en remplaçant sa politique d'exécution par une autre qui serait incompatible.

1. Cette exigence n'est pas si forte que cela : il suffit de s'assurer que les tâches ne s'exécutent pas en parallèle et fournissent une synchronisation suffisante pour que les effets mémoire de l'une soient visibles dans la suivante – c'est précisément la garantie qu'offre newSingleThreadExecutor.

8.1.1 Interblocage par famine de thread

Des tâches qui dépendent d'autres tâches et qui s'exécutent dans un pool de threads risquent de s'*interbloquer*. Dans un exécuteur monothread, une tâche qui en soumet une autre au même exécuteur et qui attend son résultat provoquera toujours un interblocage : la seconde tâche attendra en effet dans la file d'attente que la première se termine, mais cette dernière ne pourra jamais finir puisqu'elle attend le résultat de la seconde. Il peut se passer le même phénomène dans des pools plus grands si tous les threads qui s'exécutent sont bloqués en attente d'autres tâches encore dans la file d'attente. C'est ce que l'on appelle un *interblocage par famine de thread* ; il peut intervenir à chaque fois qu'une tâche du pool lance une attente bloquante non bornée sur une ressource ou une condition qui ne peut réussir que grâce à l'action d'une autre tâche du pool – l'attente de la valeur de retour ou de l'effet de bord d'une autre tâche, par exemple, sauf si l'on peut garantir que le pool est suffisamment grand.

La classe ThreadDeadlock du Listing 8.1 illustre un interblocage par famine de thread. RenderPageTask soumet deux tâches supplémentaires à l'Executor pour récupérer l'en-tête et le pied de page, elle traite le corps de la page, attend la fin des tâches header et footer puis combine l'en-tête, le corps et le pied de page dans la page finale. Avec un exécuteur monothread, ThreadDeadlock produira toujours un interblocage. De même, des tâches qui se coordonnent entre elles avec une barrière pourraient également provoquer un interblocage par famine de thread si le pool n'est pas assez grand.

Listing 8.1 : Interblocage de tâches dans un Executor monothread. Ne le faites pas.

```
public class ThreadDeadlock {  
    ExecutorService exec = Executors.newSingleThreadExecutor ();  
  
    public class RenderPageTask implements Callable<String> {  
        public String call() throws Exception {  
            Future<String> header, footer;  
            header = exec.submit(new LoadfileTask("header.html"));  
            footer = exec.submit(new LoadfileTask("footer.html"));  
            String page = renderBody();  
            // Interblocage - la tâche attend la fin de la sous-tâche.  
            return header.get() + page + footer.get();  
        }  
    }  
}
```



À chaque fois que vous soumettez des tâches non indépendantes à un Executor, vous risquez un interblocage par famine de thread et vous devez préciser dans le code ou dans le fichier de configuration de cet Executor toutes les informations sur la taille du pool et les contraintes de configuration.

Outre les éventuelles bornes explicites sur la taille d'un pool de threads, il peut également exister des limites implicites provenant de contraintes sur d'autres ressources. Si une application utilise un pool de dix connexions JDBC et que chaque tâche ait besoin

d'une connexion à une base de données, par exemple, c'est comme si le pool de threads n'avait que dix threads car les tâches excédentaires se bloqueront en attente d'une connexion.

8.1.2 Tâches longues

Les pools de threads peuvent rencontrer des problèmes de réactivité si les tâches peuvent se bloquer pendant de longues périodes, même s'il n'y a pas d'interblocage. Les tâches longues peuvent embouteiller un pool de threads, ce qui augmentera le temps de réponse même pour les tâches courtes. Si la taille du pool est trop petite par rapport au nombre attendu de tâches longues, tous les threads du pool finiront par exécuter des tâches longues et la réactivité s'en ressentira.

Une technique pouvant atténuer les effets désagréables des tâches longues consiste à faire en sorte que les tâches utilisent des attentes temporisées pour les ressources au lieu d'attentes illimitées. La plupart des méthodes bloquantes de la bibliothèque standard, comme `Thread.join()`, `BlockingQueue.put()`, `CountDownLatch.await()` et `Selector.select()`, existent en versions à la fois temporisée et non temporisée. Lorsque le délai d'attente expire, on peut considérer que la tâche a échoué et il s'agit alors de l'annuler ou de la replacer en file d'attente pour qu'elle soit réexécutée plus tard. Ceci garantit que chaque tâche finira par avancer, soit vers une fin normale, soit vers un échec, permettant ainsi de libérer des threads pour les tâches qui peuvent se terminer plus rapidement. Si un pool de threads est souvent rempli de tâches bloquées, cela peut également vouloir dire qu'il est trop petit.

8.2 Taille des pools de threads

La taille idéale d'un pool de threads dépend des types de tâches qui seront soumis et des caractéristiques du système sur lequel il est déployé. Les tailles des pools sont rarement codées en dur ; elles devraient plutôt être indiquées par un mécanisme de configuration ou calculées dynamiquement à partir de `Runtime.availableProcessors()`.

Le calcul de la taille des pools de threads n'est pas une science exacte mais, heureusement, il suffit d'éviter les valeurs "trop grandes" ou "trop petites". Si un pool est trop grand, les threads lutteront pour des ressources processeur et mémoire peu abondantes, ce qui provoquera une utilisation plus forte de la mémoire et un éventuel épuisement des ressources. S'il est trop petit, le débit s'en ressentira puisque des processeurs ne seront pas utilisés pour effectuer le travail disponible.

Pour choisir une taille correcte, vous devez connaître votre environnement informatique, votre volume des ressources et la nature de vos tâches. De combien de processeurs le système de déploiement dispose-t-il ? Quelle est la taille de sa mémoire ? Est-ce que les tâches effectuent principalement des calculs, des E/S ou un mélange des deux ? A-t-on besoin de ressources rares, comme une connexion JDBC ? Si l'on a des catégories

différentes de tâches avec des comportements très différents, il est préférable d'utiliser plusieurs pools de threads pour que chacun puisse être réglé en fonction de sa charge de travail.

Pour les tâches de calcul intensif, un système à N_{cpu} processeurs est utilisé de façon optimale avec un pool de $N_{cpu}+1$ threads (les threads de calcul intensif pouvant aussi avoir une faute de page ou se mettre en pause pour une raison ou une autre, un thread "supplémentaire" empêche que des cycles processeurs soient inutilisés lorsque cela arrive). Les tâches qui contiennent également des E/S ou d'autres opérations bloquantes ont besoin d'un pool plus grand puisque les threads ne seront pas tous disponibles à tout moment. Pour choisir une taille correcte, vous devez estimer le rapport du temps d'attente des tâches par rapport à leur temps de calcul ; cette estimation n'a pas besoin d'être précise et peut être obtenue en analysant le déroulement du programme. La taille peut également être ajustée en lançant l'application avec différentes tailles de pools et en observant l'utilisation des processeurs.

Étant donné les définition suivantes :

N_{cpu} = nombre de processeurs

U_{cpu} = Utilisation souhaitée des processeurs, $0 \leq U_{cpu} \leq 1$

W/C = rapport du temp d'attente par rapport au temps de calcul

La taille de pool optimale pour que les processeurs atteignent l'utilisation souhaitée est :

$$N_{threads} = N_{cpu} \times U_{cpu} \times (1 + W/C)$$

Vous pouvez connaître le nombre de processeurs à l'aide de la classe `Runtime` :

```
int N_CPUTS = Runtime.getRuntime().availableProcessors();
```

Les cycles processeurs ne sont, bien sûr, pas la seule ressource que vous pouvez gérer avec les pools de threads. Les autres ressources qui peuvent entrer en ligne de compte pour les tailles des pools sont la mémoire, les descripteurs de fichiers et de sockets et les connexions aux bases de données. Le calcul des contraintes de taille associées à ces types de ressources est plus facile : il suffit d'additionner le nombre de ces ressources exigé par chaque tâche et de diviser par la quantité disponible. Le résultat sera une limite supérieure de la taille du pool.

Lorsque des tâches demandent une ressource en pool, comme des connexions à des bases de données, la taille du pool de threads et celle du pool de ressources s'affectent mutuellement. Si chaque tâche demande une connexion, la taille effective du pool de thread est limitée par celle du pool de connexion. De même, si les seuls consommateurs des connexions sont des tâches du pool, la taille effective du pool de connexions est limitée par celle du pool de threads.

8.3 Configuration de ThreadPoolExecutor

ThreadPoolExecutor fournit l'implémentation de base des exécuteurs renvoyés par les méthodes fabriques `newCachedThreadPool()`, `newFixedThreadPool()` et `newScheduledThreadPool()` de Executors. C'est une implémentation souple et robuste d'un pool, qui est également très adaptable.

Si la politique d'exécution par défaut ne vous convient pas, vous pouvez instancier un ThreadPoolExecutor au moyen de son constructeur et l'adapter à vos besoins ; consultez le code source de Executors pour voir les politiques d'exécution des configurations par défaut et les utiliser comme point de départ. ThreadPoolExecutor dispose de plusieurs constructeurs, le plus général est présenté dans le Listing 8.2.

Listing 8.2 : Constructeur général de ThreadPoolExecutor.

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler) { ... }
```

8.3.1 Crédit et suppression de threads

La création et la suppression des threads dépendent des tailles normale et maximale du pool, ainsi que du temps de maintien. La taille normale est la taille souhaitée ; l'implémentation tente de maintenir le pool à cette taille, même s'il n'y a aucune tâche à exécuter¹, et ne créera pas plus de threads tant que la file d'attente n'est pas pleine². La taille maximale du pool est la limite supérieure du nombre de threads pouvant être actifs simultanément. Un thread qui est resté inactif pendant plus longtemps que le temps de maintien devient candidat à la suppression et peut être terminé si la taille courante du pool dépasse sa taille normale.

1. Lorsqu'un ThreadPoolExecutor est créé, les threads initiaux sont lancés non pas immédiatement mais à mesure que les tâches sont soumises, à moins que vous n'appeliez `prestartAllCoreThreads()`.

2. Les développeurs sont parfois tentés d'utiliser une taille normale non nulle pour que les threads soient supprimés et n'empêchent pas la JVM de s'arrêter, mais cela peut provoquer un comportement étrange avec les pools de threads qui n'utilisent pas une `SynchronousQueue` comme file d'attente (ce qui est le cas de `newCachedThreadPool()`). Si le pool a déjà sa taille normale, ThreadPoolExecutor ne crée un nouveau thread que si la file d'attente est pleine : les tâches soumises à un pool ayant une file avec une capacité quelconque et une taille normale de zéro ne s'exécuteront pas tant que la file n'est pas pleine, ce qui n'est généralement pas ce que l'on souhaite. Avec Java 6, `allowCoreThreadTimeOut` permet de demander que tous les pools de thread aient un délai d'expiration ; vous pouvez activer cette fonctionnalité avec une taille normale de zéro si vous voulez obtenir un pool de threads avec une file d'attente bornée, mais qui supprimera quand même tous les threads lorsqu'il n'y a rien à faire.

En configurant la taille normale du pool et le temps de maintien, vous pouvez encourager le pool à réclamer les ressources utilisées par les threads inactifs afin de les rendre disponibles pour le travail utile (comme d'habitude, c'est un compromis : supprimer les threads inactifs ajoute une latence supplémentaire due à la création des threads si ceux-ci doivent être plus tard créés lorsque la demande augmente).

La fabrique `newFixedThreadPool()` fixe les tailles normale et maximale à la taille de pool demandée, ce qui donne l'effet d'un délai d'expiration infini ; la fabrique `newCachedThreadPool()` fixe la taille maximale du pool à `Integer.MAX_VALUE` et sa taille normale à zéro avec un délai d'expiration de 1 minute, ce qui a pour effet de créer un pool de threads qui peut s'étendre indéfiniment et qui se rétrécira à nouveau lorsque la demande chutera. D'autres combinaisons sont possibles en utilisant le constructeur explicite de `ThreadPoolExecutor`.

8.3.2 Gestion des tâches en attente

Les pools de threads bornés limitent le nombre de tâches pouvant s'exécuter simultanément (les exéuteurs monotâches sont un cas particulier puisqu'ils garantissent que les tâches ne s'exécuteront jamais en parallèle, ce qui permet d'obtenir une thread safety au moyen du confinement au thread).

Dans la section 6.1.2, nous avons vu qu'une création d'un nombre illimité de threads pouvait conduire à une instabilité et avons réglé ce problème en utilisant un pool de threads de taille fixe au lieu de créer un nouveau thread pour chaque requête. Cependant, ce n'est qu'une solution partielle : même si cela est moins probable, l'application peut quand même se retrouver à court de ressources en cas de forte charge. En effet, si la fréquence d'arrivée des nouvelles requêtes est supérieure à celle de leur traitement, ces requêtes seront placées en file d'attente. Avec un pool de threads, elles attendront dans une file de `Runnable` gérée par l'`Executor` au lieu d'être placées dans une file de threads concourant pour l'accès au processeur. Représenter une tâche en attente par un objet `Runnable` et un nœud d'une liste est certainement beaucoup moins coûteux que la représenter par un thread, mais le risque d'épuiser les ressources reste quand même présent si les clients peuvent envoyer des requêtes au serveur plus vite qu'il ne peut les gérer ; en effet, les requêtes arrivent souvent par paquet, même lorsque leur fréquence moyenne est assez stable. Les files d'attente peuvent aider à atténuer les pics temporaires de tâches mais, si celles-ci continuent d'arriver trop rapidement, vous devrez ralentir leur fréquence d'arrivée pour éviter de tomber en panne de mémoire¹. Avant même

1. Ceci est analogue au contrôle du flux dans les communications sur le réseau : vous pouvez placer un certain volume de données dans un tampon, mais vous devrez quand même trouver un moyen pour que l'autre extrémité cesse d'envoyer des données ou supprimer les données excédentaires en espérant que l'expéditeur les retransmettra lorsque vous ne serez plus aussi saturé.

d'être à court de mémoire, le temps de réponse empirera progressivement à mesure que la file des tâches grossira.

ThreadPoolExecutor vous permet de fournir une BlockingQueue pour stocker les tâches en attente d'exécution. Il existe trois approches de base pour la mise en attente des tâches : utiliser une file non bornée, une file bornée ou un transfert synchrone. Le choix de la file influe sur d'autres paramètres de configuration, comme la taille du pool.

Par défaut, `newFixedThreadPool()` et `newSingleThreadExecutor()` utilisent une file `LinkedBlockingQueue` non bornée : les tâches sont placées dans cette file si tous les threads sont occupés mais elle peut grossir sans limite si les tâches continuent d'arriver plus vite qu'elles ne peuvent être traitées.

Une stratégie de gestion des ressources plus stable consiste à utiliser une liste bornée, comme une `ArrayBlockingQueue` ou une `LinkedBlockingQueue` ou une `PriorityBlockingQueue` bornées. Ces files empêchent l'épuisement des ressources mais posent le problème du devenir des nouvelles tâches lorsque la file est pleine (dans la section 8.3.3, nous verrons qu'il existe un certain nombre de *politiques de saturation* possibles pour régler ce problème). Avec une file d'attente bornée, la taille de la file et celle du pool doivent être mutuellement adaptées : une grande file associée à un petit pool permet de réduire la consommation mémoire, l'utilisation du processeur et le basculement de contexte au prix d'une contrainte sur le débit.

Pour les pools très gros ou non bornés, vous pouvez également complètement vous passer d'une file d'attente et transmettre directement les tâches des producteurs aux threads en utilisant une `SynchronousQueue` qui, en réalité, n'est absolument pas une file mais est un mécanisme permettant de gérer les échanges entre threads. Pour placer un élément dans une `SynchronousQueue`, il faut qu'il y ait déjà un autre thread en attente du transfert. Si aucun thread n'attend et si la taille courante du pool est inférieure à la taille maximale, `ThreadPoolExecutor` crée un nouveau thread ; sinon la tâche est rejetée selon la politique de saturation. L'utilisation d'un transfert direct est plus efficace car la tâche peut être passée directement au thread qui l'exécutera plutôt qu'être placée dans une file où elle devra attendre qu'un thread vienne la récupérer. `SynchronousQueue` n'est possible que si le pool n'est pas borné ou si l'on peut rejeter les tâches excédentaires. La méthode fabrique `newCachedThreadPool()` utilise une `SynchronousQueue`.

L'utilisation d'une file d'attente FIFO comme `LinkedBlockingQueue` ou `ArrayBlockingQueue` force les tâches à être lancées dans leur ordre d'arrivée. Pour avoir plus de contrôle sur leur ordre d'exécution, vous pouvez utiliser une `PriorityBlockingQueue` qui les ordonnera en fonction de leurs priorités respectives, définies selon l'ordre naturel (si les tâches implémentent `Comparable`) ou par un `Comparator`.

La méthode fabrique `newCachedThreadPool()` est un bon choix par défaut pour un Executor, car les performances obtenues sont meilleures que celles d'un pool de threads de taille fixe¹. Ce dernier est un choix correct lorsque l'on veut limiter le nombre de tâches concurrentes pour gérer les ressources, comme dans le cas d'une application serveur qui reçoit des requêtes provenant de clients réseaux et qui, sinon, pourrait se retrouver surchargée.

Borner le pool de threads ou la file d'attente n'est souhaitable que lorsque les tâches sont indépendantes. Lorsque les tâches dépendent d'autres tâches, les pools de threads ou les files bornées peuvent provoquer des interblocages par famine de threads ; utilisez plutôt une configuration de pool non borné, comme `newCachedThreadPool()`².

8.3.3 Politiques de saturation

La *politique de saturation* entre en jeu lorsqu'une file d'attente bornée est remplie. Pour un `ThreadPoolExecutor`, vous pouvez la modifier en appelant la méthode `setRejectedExecutionHandler()` (elle est également utilisée lorsqu'une tâche est soumise à un Executor qui a été arrêté). Il existe plusieurs versions de `RejectedExecutionHandler` pour mettre en œuvre les différentes politiques de saturation, `AbortPolicy`, `CallerRunsPolicy`, `DiscardPolicy` et `DiscardOldestPolicy`.

La politique par défaut, *abort*, force `execute()` à lancer l'exception non contrôlée `RejectedExecutionException` qui peut être capturée par l'appelant pour implémenter sa propre gestion du dépassement en fonction de ses besoins. La politique *discard* supprime en silence la nouvelle tâche soumise si elle ne peut pas être mise dans la file d'attente ; la politique *discard-oldest* supprime la prochaine tâche qui serait exécutée et tente de resoumettre la nouvelle tâche (si la file d'attente est une file a priorités, cela supprimera l'élément le plus prioritaire et c'est la raison pour laquelle la combinaison de cette politique avec une file a priorités n'est pas un bon choix).

La politique *caller-runs* implémente une forme de ralentissement qui ne supprime pas de tâche et ne lance pas d'exception, mais qui tente de ralentir le flux des nouvelles tâches en renvoyant une partie du travail à l'appelant. La nouvelle tâche soumise s'exécute non pas dans un thread du pool mais dans le thread qui appelle `execute()`. Si nous avions modifié notre exemple `WebServer` pour qu'il utilise une file d'attente bornée et la politique *caller-runs*, la nouvelle tâche se serait exécutée dans le thread

1. Cette différence de performances provient de l'utilisation de `SynchronousQueue` au lieu de `LinkedBlockingQueue`. Avec Java 6, `SynchronousQueue` a été remplacée par un nouvel algorithme non bloquant qui améliore le débit d'un facteur de trois par rapport à l'implémentation de `SynchronousQueue` en Java 5 (Scherer et al., 2006).

2. Une autre configuration possible pour les tâches qui soumettent d'autres tâches et qui attendent leurs résultats consiste à utiliser un pool de threads borné, une `SynchronousQueue` comme file d'attente et la politique de saturation *caller-runs*.

principal au cours de l'appel à `execute()` si tous les threads du pool étaient occupés et la file d'attente, remplie. Comme cela prendrait sûrement un certain temps, le thread principal ne peut pas soumettre d'autres tâches pendant un petit moment, afin de donner le temps aux threads du pool de rattraper leur retard. Le thread principal n'appellera pas non plus `accept()` pendant ce laps de temps et les requêtes entrantes seront donc placées dans la file d'attente de la couche TCP au lieu de l'être dans l'application. Si la surcharge persiste, la couche TCP finit par décider qu'elle a placé en attente suffisamment de demandes de connexions et commence à supprimer également des requêtes. À mesure que le serveur devient surchargé, cette surcharge est graduellement déplacée vers l'extérieur – du pool de threads vers la file d'attente, vers l'application et vers la couche TCP et, enfin, vers le client –, ce qui permet une dégradation plus progressive en cas de forte charge.

Choisir une politique de saturation ou effectuer des modifications sur la politique d'exécution peut se faire au moment de la création de l'Executor. Le Listing 8.3 montre la création d'un pool de threads de taille fixe avec une politique de saturation *caller-runs*.

Listing 8.3 : Création d'un pool de threads de taille fixe avec une file bornée et la politique de saturation *caller-runs*.

```
ThreadPoolExecutor executor
    = new ThreadPoolExecutor(N_THREADS, N_THREADS,
                           0L, TimeUnit.MILLISECONDS,
                           new LinkedBlockingQueue<Runnable>(CAPACITY));
executor.setRejectedExecutionHandler(
    new ThreadPoolExecutor.CallerRunsPolicy());
```

Il n'existe pas de politique de saturation pré définie pour faire en sorte que `execute()` se bloque lorsque la file d'attente est pleine. Cependant, on peut obtenir le même résultat en utilisant un Semaphore pour borner la fréquence d'injection des tâches, comme dans le Listing 8.4. Avec cette approche, on utilise une file non bornée (il n'y a aucune raison de borner à la fois la taille de la file et la fréquence d'injection) et on limite le semaphore pour qu'il soit égal à la taille du pool plus le nombre de tâches en attente que l'on veut autoriser, car le semaphore borne à la fois le nombre de tâches en cours et en attente d'exécution.

Listing 8.4 : Utilisation d'un Semaphore pour ralentir la soumission des tâches.

```
@ThreadSafe
public class BoundedExecutor {
    private final Executor exec;
    private final Semaphore semaphore;

    public BoundedExecutor(Executor exec, int bound) {
        this.exec = exec;
        this.semaphore = new Semaphore(bound);
    }

    public void submitTask(final Runnable command)
        throws InterruptedException {
        semaphore.acquire();
        exec.execute(command);
    }
}
```

```
try {
    exec.execute(new Runnable() {
        public void run() {
            try {
                command.run();
            } finally {
                semaphore.release();
            }
        }
    });
} catch (RejectedExecutionException e) {
    semaphore.release();
}
}
```

8.3.4 Fabriques de threads

À chaque fois qu'il a besoin de créer un thread, un pool de threads utilise une *fabrique de thread* (voir Listing 8.5). La fabrique par défaut crée un thread non démon sans configuration spéciale. En indiquant une fabrique spécifique, vous pouvez adapter la configuration des threads du pool. `ThreadFactory` n'a qu'une seule méthode, `newThread()`, qui est appelée à chaque fois qu'un pool de threads doit créer un nouveau thread.

Listing 8.5 : Interface ThreadFactory.

```
public interface ThreadFactory {
    Thread newThread(Runnable r);
}
```

Il y a un certain nombre de raisons d'utiliser une fabrique de thread personnalisée : vous pouvez, par exemple, vouloir préciser un `UncaughtExceptionHandler` pour les threads du pool ou créer une instance d'une classe `Thread` adaptée à vos besoins, qui inscrit des informations de débogage dans un journal. Vous pourriez également vouloir modifier la priorité (ce qui n'est généralement pas souhaitable, comme on l'explique dans la section 10.3.1) ou positionner l'état démon (ce qui n'est pas non plus conseillé, comme on l'a indiqué dans la section 7.4.2) des threads du pool. Vous pouvez aussi simplement vouloir donner des noms plus significatifs aux threads afin de simplifier l'interprétation des traces et des journaux d'erreur.

La classe `MyThreadFactory` du Listing 8.6 présente une fabrique de threads personnalisés. Elle crée un nouvel objet `MyAppThread` en passant au constructeur un nom spécifique au pool afin de le repérer plus facilement dans les traces et les journaux d'erreur. `MyAppThread` peut également être utilisée ailleurs dans l'application pour que tous les threads puissent bénéficier de ses fonctionnalités de débogage.

Listing 8.6 : Fabrique de threads personnalisés.

```
public class MyThreadFactory implements ThreadFactory {
    private final String poolName;

    public MyThreadFactory(String poolName) {
        this.poolName = poolName;
    }

    public Thread newThread(Runnable runnable) {
        return new MyAppThread(runnable, poolName);
    }
}
```

La partie intéressante se situe dans la classe `MyAppThread`, qui est présentée dans le Listing 8.7 et qui permet de donner un nom au thread, de mettre en place un `UncaughtExceptionHandler` personnalisé qui écrit un message sur un `Logger`, de gérer des statistiques sur le nombre de threads créés et supprimés et qui inscrit éventuellement un message de débogage dans le journal lorsqu'un thread est créé ou terminé.

Listing 8.7 : Classe de base pour les threads personnalisés.

```
public class MyAppThread extends Thread {
    public static final String DEFAULT_NAME = "MyAppThread";
    private static volatile boolean debugLifecycle = false;
    private static final AtomicInteger created = new AtomicInteger();
    private static final AtomicInteger alive = new AtomicInteger();
    private static final Logger log = Logger.getAnonymousLogger();

    public MyAppThread(Runnable r) { this(r, DEFAULT_NAME); }

    public MyAppThread(Runnable runnable, String name) {
        super(runnable, name + "-" + created.incrementAndGet ());
        setUncaughtExceptionHandler(
            new Thread.UncaughtExceptionHandler() {
                public void uncaughtException(Thread t, Throwable e) {
                    log.log(Level.SEVERE, "UNCAUGHT in thread " + t.getName(), e);
                }
            });
    }

    public void run() {
        // Copie l'indicateur debug pour garantir la cohérence de sa valeur.
        boolean debug = debugLifecycle ;
        if (debug) log.log(Level.FINE, "Created "+getName());
        try {
            alive.incrementAndGet ();
            super.run();
        } finally {
            alive.decrementAndGet ();
            if (debug) log.log(Level.FINE, "Exiting "+getName());
        }
    }

    public static int getThreadsCreated() { return created.get(); }
    public static int getThreadsAlive() { return alive.get(); }
    public static boolean getDebug(){ return debugLifecycle ; }
    public static void setDebug(booleanb) { debugLifecycle = b; }
}
```

Si votre application utilise une *politique de sécurité* pour accorder des permissions à du code particulier, vous pouvez utiliser la méthode fabrique `privilegedThreadFactory()` de `Executors` pour construire votre fabrique de threads. Celle-ci crée des threads de pool ayant les mêmes permissions, `AccessControlContext` et `contextClassLoader`, comme le thread créant la `privilegedThreadFactory`. Sinon les threads créés par le pool héritent des permissions du client qui appelle `execute()` ou `submit()` lorsqu'un nouveau thread est nécessaire, ce qui peut provoquer des exceptions troublantes, liées à la sécurité.

8.3.5 Personnalisation de `ThreadPoolExecutor` après sa construction

La plupart des options passées aux constructeurs de `ThreadPoolExecutor` (notamment les tailles normale et maximale du pool, le temps de maintien, la fabrique de threads et le gestionnaire d'exécution rejetée) peuvent également être modifiées après la construction de l'objet *via* des méthodes modificatrices. Si l'Executor a été créé au moyen de l'une des méthodes fabriques de `Executors` (sauf `newSingleThreadExecutor`), il est possible de le transyster en `ThreadPoolExecutor` afin d'avoir accès aux méthodes modificatrices, comme dans le Listing 8.8.

Listing 8.8 : Modification d'un Executor créé avec les méthodes fabriques standard.

```
ExecutorService exec = Executors.newCachedThreadPool();
if (exec instanceof ThreadPoolExecutor)
    ((ThreadPoolExecutor) exec).setCorePoolSize (10);
else
    throw new AssertionError("Oops, bad assumption");
```

`Executors` fournit la méthode fabrique `unconfigurableExecutorService()`, qui prend en paramètre un `ExecutorService` existant et l'enveloppe dans un objet ne proposant que les méthodes de `ExecutorService` afin qu'il ne puisse plus être configuré. À la différence des implémentations utilisant des pools, `newSingleThreadExecutor()` renvoie un objet `ExecutorService` enveloppé de cette façon au lieu d'un `ThreadPoolExecutor` brut. Bien qu'un exécuteur monothread soit en fait implémenté comme un pool ne contenant qu'un seul thread, il garantit également que les tâches ne s'exécuteront pas en parallèle. Si un code mal inspiré tentait d'augmenter la taille du pool d'un exécuteur monothread, cela ruinerait la sémantique recherchée pour l'exécution.

Vous pouvez utiliser cette technique avec vos propres exécuteurs pour empêcher la modification de la politique d'exécution : si vous exposez un `ExecutorService` à du code et que vous ne soyez pas sûr que ce code n'essaie pas de le modifier, vous pouvez envelopper l'`ExecutorService` dans un `unconfigurableExecutorService`.

8.4 Extension de `ThreadPoolExecutor`

La classe `ThreadPoolExecutor` a été conçue pour être étendue et fournit plusieurs "points d'attache" (*hooks*) destinés à être redéfinis par les sous-classes – `beforeExecute()`, `afterExecute()` et `terminated()` – pour étendre le comportement de `ThreadPoolExecutor`.

Les méthodes `beforeExecute()` et `afterExecute()` sont appelées dans le thread qui exécute la tâche et peuvent servir à ajouter une journalisation, à mesurer le temps d'exécution, à surveiller les tâches ou à récolter des statistiques. `afterExecute()` est appelée à chaque fois que la tâche se termine en quittant normalement `run()` ou en lançant une `Exception` (elle n'est pas appelée si la tâche se termine à cause d'une `Error`). Si `beforeExecute()` lance une `RuntimeException`, la tâche n'est pas exécutée et `afterExecute()` n'est pas appelée.

La méthode `terminated()` est appelée lorsque le pool de threads a terminé son processus d'extinction, lorsque toutes les tâches se sont terminées et que tous les threads se sont éteints. Elle peut servir à libérer les ressources allouées par l'Executor au cours de son cycle de vie, à envoyer des notifications, à inscrire des informations dans le journal ou à terminer la récolte des statistiques.

8.4.1 Exemple : ajout de statistiques à un pool de threads

La classe `TimingThreadPool` du Listing 8.9 montre un pool de threads personnalisé qui utilise `beforeExecute()`, `afterExecute()` et `terminated()` pour ajouter une journalisation et une récolte de statistiques. Pour mesurer le temps d'exécution d'une tâche, `beforeExecute()` doit enregistrer le temps de départ et le stocker à un endroit où `afterExecute()` pourra le récupérer. Ces hooks d'exécution étant appelés dans le thread qui exécute la tâche, une valeur placée dans un `ThreadLocal` par `beforeExecute()` peut être relue par `afterExecute()`. `TimingThreadPool` utilise deux `AtomicLong` pour mémoriser le nombre total de tâches traitées et le temps total de traitement ; il se sert également de la méthode `terminated()` pour inscrire un message dans le journal contenant le temps moyen de la tâche.

Listing 8.9 : Pool de threads étendu par une journalisation et une mesure du temps.

```
public class TimingThreadPool extends ThreadPoolExecutor {
    private final ThreadLocal<Long> startTime = new ThreadLocal<Long>();
    private final Logger log = Logger.getLogger("TimingThreadPool ");
    private final AtomicLong numTasks = new AtomicLong();
    private final AtomicLong totalTime = new AtomicLong();

    protected void beforeExecute(Thread t, Runnable r) {
        super.beforeExecute(t, r);
        log.fine(String.format("Thread %s: start %s", t, r));
        startTime.set(System.nanoTime());
    }

    protected void afterExecute(Runnable r, Throwable t) {
        try {
            long endTime = System.nanoTime();
            long taskTime = endTime - startTime.get();
            numTasks.incrementAndGet();
            totalTime.addAndGet(taskTime);
            log.fine(String.format("Thread %s: end %s, time=%dns",
                t, r, taskTime));
        } finally {
    }
}
```

```

        super.afterExecute(r, t);
    }
}

protected void terminated() {
    try {
        log.info(String.format("Terminated: avg time=%dns",
            totalTime.get() / numTasks.get()));
    } finally {
        super.terminated();
    }
}
}

```

8.5 Parallélisation des algorithmes récursifs

Les exemples d'affichage de page de la section 6.3 passaient par une série d'améliorations tendant à rechercher un parallélisme exploitable. La première tentative était entièrement séquentielle ; la deuxième utilisait deux threads mais continuait à télécharger les images en séquence ; la dernière traitait chaque téléchargement d'image comme une tâche distincte afin d'obtenir un parallélisme plus fort. Les boucles dont les corps contiennent des calculs non triviaux ou qui effectuent des E/S potentiellement bloquantes sont généralement de bonnes candidates à la parallélisation du moment que les itérations sont indépendantes.

Si vous avez une boucle dont les itérations sont indépendantes et que vous n'ayez pas besoin qu'elles se terminent toutes avant de continuer, vous pouvez utiliser un Executor pour la transformer en boucle parallèle, comme le montrent les méthodes `processSequentially()` et `processInParallel()` du Listing 8.10.

Listing 8.10 : Transformation d'une exécution séquentielle en exécution parallèle.

```

void processSequentially(List<Element> elements) {
    for (Element e : elements)
        process(e);
}

void processInParallel(Executor exec, List<Element> elements) {
    for (final Element e : elements)
        exec.execute(new Runnable() {
            public void run() { process(e); }
        });
}

```

Un appel à `processInParallel()` redonne le contrôle plus rapidement qu'un appel à `processSequentially()` car il se termine dès que toutes les tâches ont été mises en attente dans l'Executor au lieu d'attendre qu'elles se terminent toutes. Si vous voulez soumettre un ensemble de tâches et attendre qu'elles soient toutes terminées, vous pouvez utiliser la méthode `ExecutorService.invokeAll()` ; pour récupérer les résultats dès

qu'ils sont disponibles, vous pouvez vous servir d'un CompletionService comme dans la classe `Renderer` du Listing 6.15.

Les itérations d'une boucle séquentielle peuvent être mises en parallèle si elles sont mutuellement indépendantes et si le travail effectué dans chaque itération du corps de la boucle est suffisamment significatif pour justifier le coût de la gestion d'une nouvelle tâche.

La parallélisation des boucles peut également s'appliquer à certaines constructions récursives ; dans un algorithme récursif, il y a souvent des boucles séquentielles qui peuvent être parallélisées comme dans le Listing 8.10. Le cas le plus simple est lorsque chaque itération n'a pas besoin des résultats des itérations qu'elle invoque récursivement. La méthode `sequentialRecursive()` du Listing 8.11, par exemple, effectue un parcours d'arbre en profondeur d'abord en réalisant un calcul sur chaque nœud puis en plaçant le résultat dans une collection. Sa version transformée, `parallelRecursive()`, effectue le même parcours mais en soumettant une tâche pour calculer le résultat de chaque nœud, au lieu de le calculer à chaque fois qu'elle visite un nœud.

Listing 8.11 : Transformation d'une récursion terminale séquentielle en récursion parallèle.

```
public<T> void sequentialRecursive(List<Node<T>> nodes,
                                    Collection<T> results) {
    for (Node<T> n : nodes) {
        results.add(n.compute());
        sequentialRecursive(n.getChildren(), results);
    }
}

public<T> void parallelRecursive(final Executor exec,
                                List<Node<T>> nodes,
                                final Collection<T> results) {
    for (final Node<T> n : nodes) {
        exec.execute(new Runnable() {
            public void run() {
                results.add(n.compute());
            }
        });
        parallelRecursive(exec, n.getChildren(), results);
    }
}
```

Lorsque `parallelRecursive()` se termine, chaque nœud de l'arbre a été visité (le parcours est toujours séquentiel : seuls les appels à `compute()` sont exécutés en parallèle) et le résultat pour chacun d'eux a été placé dans la file de l'Executor. Les appelants de cette méthode peuvent attendre tous les résultats en créant en Executor spécifique au parcours et utiliser `shutdown()` et `awaitTermination()` comme on le montre dans le Listing 8.12.

Listing 8.12 : Attente des résultats calculés en parallèle.

```
public<T> Collection<T> getParallelResults (List<Node<T>> nodes)
    throws InterruptedException {
    ExecutorService exec = Executors.newCachedThreadPool ();
    Queue<T> resultQueue = new ConcurrentLinkedQueue <T>();
    parallelRecursive(exec, nodes, resultQueue);
    exec.shutdown();
    exec.awaitTermination(Long.MAX_VALUE, TimeUnit.SECONDS);
    return resultQueue;
}
```

8.5.1 Exemple : un framework de jeu de réflexion

Une application amusante de cette technique permet de résoudre des jeux de réflexion consistant à trouver une suite de transformations pour passer d'un état initial à un état final, comme dans les jeux du taquin¹, du solitaire, etc.

On définit un "puzzle" comme une combinaison d'une position initiale, d'une position finale et d'un ensemble de règles qui définit les coups autorisés. Cet ensemble est formé de deux parties : un calcul de la liste des coups autorisés à partir d'une position donnée et un calcul du résultat de l'application d'un coup à une position. L'interface `Puzzle` du Listing 8.13 décrit notre abstraction : les paramètres de type `P` et `M` représentent les classes pour une position et un coup. À partir de cette interface, nous pouvons écrire un simple résolveur séquentiel qui parcourt l'espace du puzzle jusqu'à trouver une solution ou jusqu'à ce que cet espace soit épuisé.

Listing 8.13 : Abstraction pour les jeux de type "taquin".

```
public interface Puzzle<P, M> {
    P initialPosition ();
    boolean isGoal(P position);
    Set<M> legalMoves(P position);
    P move(P position, M move);
}
```

Dans le Listing 8.14, `Node` représente une position qui a été atteinte *via* une suite de coups ; elle contient une référence au coup qui a créé la position et une référence au `Node` précédent. En remontant les liens à partir d'un `Node`, nous pouvons donc reconstruire la suite de coups qui a mené à la position courante.

Listing 8.14 : Nœud pour le framework de résolution des jeux de réflexion.

```
@Immutable
static class Node<P, M> {
    final P pos;
    final M move;
    final Node<P, M> prev;

    Node(P pos, M move, Node<P, M> prev) {...}
```

1. Voir <http://www.puzzlegame.org/SlidingBlockPuzzles>.

Listing 8.14 : Nœud pour le framework de résolution des jeux de réflexion. (suite)

```

List<M> asMoveList() {
    List<M> solution = new LinkedList<M>();
    for (Node<P, M> n = this; n.move != null; n = n.prev)
        solution.add(0, n.move);
    return solution;
}
}

```

La classe `SequentialPuzzleSolver` du Listing 8.15 est un résolveur séquentiel qui effectue un parcours en profondeur d'abord de l'espace du puzzle. Elle se termine lorsqu'elle a trouvé une solution (qui n'est pas nécessairement la plus courte). La réécriture de ce résolveur pour exploiter le parallélisme nous permettrait de calculer les coups suivants et d'évaluer si le but a été atteint en parallèle car le processus d'évaluation d'un coup est presque indépendant de l'évaluation des autres coups (nous écrivons "presque" car les tâches partagent un état modifiable, comme l'ensemble des positions déjà étudiées). Si l'on dispose de plusieurs processeurs, cela peut réduire le temps nécessaire à la découverte d'une solution.

Listing 8.15 : Résolveur séquentiel d'un puzzle.

```

public class SequentialPuzzleSolver <P, M> {
    private final Puzzle<P, M> puzzle;
    private final Set<P> seen = new HashSet<P>();

    public SequentialPuzzleSolver (Puzzle<P, M> puzzle) {
        this.puzzle = puzzle;
    }

    public List<M> solve() {
        P pos = puzzle.initialPosition ();
        return search(new Node<P, M>(pos, null, null));
    }

    private List<M> search(Node<P, M> node) {
        if (!seen.contains(node.pos)) {
            seen.add(node.pos);
            if (puzzle.isGoal(node.pos))
                return node.asMoveList();
            for (M move : puzzle.legalMoves(node.pos)) {
                P pos = puzzle.move(node.pos, move);
                Node<P, M> child = new Node<P, M>(pos, move, node);
                List<M> result = search(child);
                if (result != null)
                    return result;
            }
        }
        return null;
    }

    static class Node<P, M> { /* Listing 8.14 */ }
}

```

La classe `ConcurrentPuzzleSolver` du Listing 8.16 utilise une classe interne `SolverTask` qui étend `Node` et implémente `Runnable`. L'essentiel du travail a lieu dans `run()` : évaluation des prochaines positions possibles, suppression des positions déjà étudiées,

évaluation du but atteint (par cette tâche ou une autre) et soumission des positions non étudiées à un Executor.

Pour éviter les boucles sans fin, la version séquentielle mémorisait un Set des positions déjà parcourues alors que ConcurrentPuzzleSolver utilise un ConcurrentHashMap. Ce choix garantit une thread safety et évite les situations de compétition inhérentes à la mise à jour conditionnelle d'une collection partagée en utilisant putIfAbsent() pour ajouter de façon atomique une position uniquement si elle n'est pas déjà connue. Pour stocker l'état de la recherche, ConcurrentPuzzleSolver utilise la file d'attente interne du pool de threads à la place de la pile d'appels.

Listing 8.16 : Version parallèle du résolveur de puzzle.

```
public class ConcurrentPuzzleSolver <P, M> {
    private final Puzzle<P, M> puzzle;
    private final ExecutorService exec;
    private final ConcurrentMap <P, Boolean> seen;
    final ValueLatch<Node<P, M>> solution = new ValueLatch<Node<P, M>>();

    ...
    public List<M> solve() throws InterruptedException {
        try {
            P p = puzzle.initialPosition ();
            exec.execute(newTask(p, null, null));
            // bloque jusqu'à ce qu'une solution soit trouvée
            Node<P, M> solnNode = solution.getValue();
            return (solnNode == null) ? null : solnNode.asMoveList();
        } finally {
            exec.shutdown();
        }
    }

    protected Runnable newTask(P p, M m, Node<P,M> n) {
        return new SolverTask(p, m, n);
    }

    class SolverTask extends Node<P, M> implements Runnable {
        ...
        public void run() {
            if (solution.isSet() || seen.putIfAbsent(pos, true) != null)
                return; // already solved or seen this position
            if (puzzle.isGoal(pos))
                solution.setValue(this);
            else
                for (M m : puzzle.legalMoves(pos))
                    exec.execute(newTask(puzzle.move(pos, m), m, this));
        }
    }
}
```

L'approche parallèle échange également une forme de limitation contre une autre qui peut être plus adaptée au domaine du problème. La version séquentielle effectuant une recherche en profondeur d'abord, cette recherche est limitée par la taille de la pile. La version parallèle effectue, quant à elle, une recherche en largeur d'abord et est donc affranchie du problème de la taille de la pile (bien qu'elle puisse quand même se retrouver

à court de mémoire si l'ensemble des positions à parcourir ou déjà parcourues dépasse la quantité de mémoire disponible).

Pour stopper la recherche lorsque l'on a trouvé une solution, il faut pouvoir déterminer si un thread a déjà trouvé une solution. Si nous voulons accepter la première solution trouvée, nous devons également ne mettre à jour la solution que si aucune autre tâche n'en a déjà trouvé une. Ces exigences décrivent une sorte de *loquet* (voir la section 5.5.1) et, notamment, un *loquet de résultat partiel*. Nous pourrions aisément construire un tel loquet grâce aux techniques présentées au Chapitre 14, mais il est souvent plus simple et plus fiable d'utiliser les classes existantes de la bibliothèque au lieu des mécanismes de bas niveau du langage. La classe `ValueLatch` du Listing 8.17 utilise un objet `CountDownLatch` pour fournir le comportement souhaité du loquet et se sert du verrouillage pour garantir que la solution n'est trouvée qu'une seule fois.

Listing 8.17 : Loquet de résultat partiel utilisé par `ConcurrentPuzzleSolver`.

```
@ThreadSafe
public class ValueLatch<T> {
    @GuardedBy("this")
    private T value = null;
    private final CountDownLatch done = new CountDownLatch (1);

    public boolean isSet() {
        return (done.getCount() == 0);
    }

    public synchronized void setValue(T newValue) {
        if (!isSet()) {
            value = newValue;
            done.countDown();
        }
    }

    public T getValue() throws InterruptedException {
        done.await();
        synchronized (this) {
            return value;
        }
    }
}
```

Chaque tâche consulte d'abord le loquet de la solution et s'arrête si une solution a déjà été trouvée. Le thread principal doit attendre jusqu'à ce qu'une solution ait été trouvée ; la méthode `getValue()` de `ValueLatch` se bloque jusqu'à ce qu'un thread ait initialisé la valeur. `ValueLatch` fournit un moyen de mémoriser une valeur de sorte que seul le premier appel fixe en réalité la valeur ; les appelants peuvent tester si elle a été initialisée et se bloquer en attente si ce n'est pas le cas. Au premier appel de `setValue()`, la solution est mise à jour et l'objet `CountDownLatch` est décrémenté, ce qui libère le thread résolveur principal dans `getValue()`.

Le premier thread à trouver une solution met également fin à l'`Executor` pour empêcher qu'il accepte d'autres tâches. Pour éviter de traiter `RejectedExecutionException`, le

gestionnaire d'exécution rejetée doit être configuré pour supprimer les tâches soumises. De cette façon, toutes les tâches non terminées s'exécuteront jusqu'à leur fin et toutes les nouvelles tentatives d'exécuter de nouvelles tâches échoueront en silence, permettant ainsi à l'exécuteur de se terminer (si les tâches mettent trop de temps à s'exécuter, nous pourrions les interrompre au lieu de les laisser finir).

ConcurrentPuzzleSolver ne gère pas très bien le cas où il n'y a pas de solution : si tous les coups possibles et toutes les solutions ont été évalués sans trouver de solution, solve() attend indéfiniment dans l'appel à getSolution(). La version séquentielle se terminait lorsqu'elle avait épousé l'espace de recherche, mais faire en sorte que des programmes parallèles se terminent peut parfois être plus difficile. Une solution possible consiste à mémoriser le nombre de tâches résolveur actives et d'initialiser la solution à null lorsque ce nombre tombe à zéro ; c'est ce que l'on fait dans le Listing 8.18.

Listing 8.18 : Résolveur reconnaissant qu'il n'y a pas de solution.

```
public class PuzzleSolver<P,M> extends ConcurrentPuzzleSolver <P,M> {
    ...
    private final AtomicInteger taskCount = new AtomicInteger (0);

    protected Runnable newTask(P p, M m, Node<P,M> n) {
        return new CountingSolverTask (p, m, n);
    }

    class CountingSolverTask extends SolverTask {
        CountingSolverTask(P pos, M move, Node<P, M> prev) {
            super(pos, move, prev);
            taskCount.incrementAndGet ();
        }
        public void run() {
            try {
                super.run();
            } finally {
                if (taskCount.decrementAndGet () == 0)
                    solution.setValue(null);
            }
        }
    }
}
```

Trouver la solution peut également prendre plus de temps que ce que l'on est prêt à attendre ; nous pourrions alors imposer plusieurs conditions de terminaison au résolveur. L'une d'elles est une limite de temps, qui peut aisément être réalisée en implémentant une version temporisée de getValue() dans ValueLatch (cette version utiliserait la version temporisée de await()) et en arrêtant l'Executor en mentionnant l'échec si le délai imparti à getValue() a expiré. Une autre mesure spécifique aux jeux de réflexion consiste à ne rechercher qu'un certain nombre de positions. On peut également fournir un mécanisme d'annulation et laisser le client choisir quand arrêter la recherche.

Résumé

Le framework Executor est un outil puissant et souple pour exécuter des tâches en parallèle. Il offre un certain nombre d'options de configuration, comme les politiques de création et de suppression des tâches, la gestion des tâches en attente et ce qu'il convient de faire des tâches excédentaires. Il fournit également des méthodes d'interception (*hooks*) permettant d'étendre son comportement. Cela dit, comme avec la plupart des frameworks puissants, certaines combinaisons de configuration ne fonctionnent pas ensemble ; certains types de tâches exigent des politiques d'exécutions spécifiques et certaines combinaisons de paramètres de configuration peuvent produire des résultats curieux.

Applications graphiques

Si vous avez essayé d'écrire une application graphique, même simple, avec Swing, vous savez que ce type de programme a ses propres problèmes avec les threads. Pour maintenir la thread safety, certaines tâches doivent s'exécuter dans le thread des événements de Swing, mais vous ne pouvez pas lancer des tâches longues dans ce thread sous peine d'obtenir une interface utilisateur peu réactive. En outre, les structures de données de Swing n'étant pas thread-safe, vous devez vous assurer de les confiner au thread des événements.

Quasiment tous les toolkits graphiques, dont Swing et AWT, sont implémentés comme des sous-systèmes monothreads dans lesquels toute l'activité de l'interface graphique est confinée dans un seul thread. Si vous n'avez pas l'intention d'écrire un programme entièrement monothread, certaines activités devront s'exécuter en partie dans le thread de l'application et en partie dans le thread des événements. Comme la plupart des autres bogues liés aux threads, votre programme ne se plantera pas nécessairement immédiatement si cette division est mal faite ; il peut se comporter bizarrement sous certaines conditions qu'il est souvent difficile d'identifier. Bien que les frameworks graphiques soient des sous-systèmes monothreads, votre application peut ne pas l'être et vous devrez quand même vous soucier des problèmes de thread lorsque vous écrirez du code graphique.

9.1 Pourquoi les interfaces graphiques sont-elles monothreads ?

Par le passé, les applications graphiques étaient monothreads et les événements de l'interface étaient traités à partir d'une "boucle d'événements". Les frameworks graphiques modernes utilisent un modèle qui est peu différent puisqu'ils créent un thread dédié (EDT, pour *Event Dispatch Thread*) afin de gérer les événements de l'interface.

Les frameworks graphiques monothreads ne sont pas propres à Java : Qt, NextStep, Cocoa de Mac OS, X-Window et de nombreux autres sont également monothreads. Ce n'est

pourtant pas faute d'avoir essayé ; il y a eu de nombreuses tentatives de frameworks graphiques multithreads qui, à cause des problèmes persistants avec les situations de compétition et les interblocages, sont toutes arrivées au modèle d'une file d'événements monothread dans lequel un thread dédié récupère les événements dans une file et les transmet à des gestionnaires d'événements définis par l'application (au départ, AWT a essayé d'autoriser un plus grand degré d'accès multithread et la décision de rendre Swing monothread est largement due à l'expérience d'AWT).

Les frameworks graphiques multithreads ont tendance à être particulièrement sensibles aux interblocages, en partie à cause de la mauvaise interaction entre le traitement des événements d'entrée et le modèle orienté objet utilisé pour représenter les composants graphiques, quel qu'il soit. Les actions de l'utilisateur ont tendance à "remonter" du système d'exploitation à l'application – un clic de souris est détecté par le SE, transformé en événement "clic souris" par le toolkit et finit par être délivré à un écouteur de l'application sous la forme d'un événement de plus haut niveau, comme un événement "bouton pressé".

Les actions initiées par l'application, en revanche, "descendent" de l'application vers le SE – la modification de la couleur de fond d'un composant est demandée dans l'application, puis est envoyée à une classe de composant spécifique et finit dans le SE, qui se charge de l'affichage. Combiner cette tendance des activités à accéder aux mêmes objets graphiques dans un ordre opposé avec la nécessité de rendre chaque objet thread-safe est la meilleure recette pour obtenir un ordre de verrouillage incohérent menant à un interblocage (voir Chapitre 10). C'est exactement ce que quasiment tout effort de développement d'un toolkit graphique a redécouvert par expérience.

Un autre facteur provoquant les interblocages dans les frameworks graphiques multi-threads est la prévalence du patron *modèle-vue-contrôleur* (MVC). Le regroupement des interactions utilisateur en objets coopératifs modèle, vue et contrôleur simplifie beaucoup l'implémentation des applications graphiques mais accroît le risque d'un ordre de verrouillage incohérent. En effet, le contrôleur peut appeler le modèle, qui prévient la vue que quelque chose a été modifié. Mais le contrôleur peut également appeler la vue, qui peut à son tour appeler le modèle pour interroger son état. Là encore, on obtient un ordre de verrouillage incohérent avec le risque d'interblocage qui en découle.

Dans son blog¹, le vice-président de Sun, Graham Hamilton, résume parfaitement ces défis en expliquant pourquoi le toolkit graphique multithread est l'un des "rêves déçus" récurrents de l'informatique :

"Je crois que vous pouvez programmer correctement avec des toolkits graphiques multithreads si le toolkit a été soigneusement conçu, si le toolkit expose en moult détails sa méthode de verrouillage, si vous êtes

1. Voir <http://weblogs.java.net/blog/kgh/archive/2004/10/>.

très malin, très prudent et que vous ayez une vision globale de la structure complète du toolkit. Si l'un de ces critères fait défaut, les choses marcheront pour l'essentiel mais, de temps en temps, l'application se figera (à cause des interblocages) ou se comportera bizarrement (à cause des situations de compétition). Cette approche multithreads marche surtout pour les personnes qui ont été intimement impliquées dans le développement du toolkit.

Malheureusement, je ne crois pas que cet ensemble de critères soit adapté à une utilisation commerciale à grande échelle. Le résultat sera que les programmeurs normaux auront tendance à produire des applications qui ne sont pas tout à fait fiables et qu'il ne sera pas du tout évident d'en trouver la cause : ils seront donc très mécontents et frustrés et utiliseront les pires mots pour qualifier le pauvre toolkit innocent."

Les frameworks graphiques monothreads réalisent la thread safety via le confinement au thread ; on n'accède à tous les objets de l'interface graphique, y compris les composants visuels et les modèles de données, qu'à partir du thread des événements. Cela ne fait, bien sûr, que repousser une partie du problème de la thread safety vers le développeur de l'application, qui doit s'assurer que ces objets sont correctement confinés.

9.1.1 Traitement séquentiel des événements

Les interfaces graphiques s'articulent autour du traitement d'événements précis, comme les clics de souris, les pressions de touche ou l'expiration de délais. Les événements sont un type de tâche ; du point de vue structurel, la mécanique de gestion des événements fournie par AWT et Swing est semblable à un Executor.

Comme il n'y a qu'un seul thread pour traiter les tâches de l'interface graphique, celles-ci sont gérées en séquence – une tâche se termine avant que la suivante ne commence et il n'y a jamais recouvrement de deux tâches. Cet aspect facilite l'écriture du code des tâches car il n'y a pas besoin de se soucier des interférences entre elles.

L'inconvénient du traitement séquentiel des tâches est que, si l'une d'entre elles met du temps à s'exécuter, les autres devront attendre qu'elle se soit terminée. Si ces autres tâches sont chargées de répondre à une saisie de l'utilisateur ou de fournir une réponse visuelle, l'application semblera figée. Si une tâche longue s'exécute dans le thread des événements, l'utilisateur ne peut même pas cliquer sur "Annuler" puisque l'écouteur de ce bouton ne sera pas appelé tant que la tâche longue ne s'est pas terminée. Les tâches qui s'exécutent dans le thread des événements doivent donc redonner rapidement le contrôle à ce thread. Une tâche longue, comme une vérification orthographique d'un gros document, une recherche dans le système de fichiers ou la récupération d'une ressource sur le réseau, doit être lancée dans un autre thread afin de redonner rapidement le contrôle au thread des événements. La mise à jour d'une jauge de progression pendant qu'une longue tâche s'exécute ou la création d'un effet visuel lorsqu'elle s'est

terminée, en revanche, doit s'exécuter dans le thread des événements. Tout ceci peut devenir assez rapidement compliqué.

9.1.2 Confinement aux threads avec Swing

Tous les composants de Swing (comme JButton et JTable) et les objets du modèle de données (comme TableModel et TreeModel) étant confinés au thread des événements, tout code qui les utilise doit s'exécuter dans ce thread. Les objets de l'interface graphique restent cohérents non grâce à la synchronisation mais grâce au confinement. L'avantage est que les tâches qui s'exécutent dans le thread des événements n'ont pas besoin de se soucier de synchronisation lorsqu'elles accèdent aux objets de présentation ; l'inconvénient est que l'on ne peut pas du tout accéder à ces objets en dehors de ce thread.

Règle monothread de Swing : les composants et modèles Swing devraient être créés, modifiés et consultés uniquement à partir du thread des événements.

Comme avec toutes les règles, il y a quelques exceptions. Un petit nombre de méthodes Swing peuvent être appelées en toute sécurité à partir de n'importe quel thread ; elles sont clairement identifiées comme thread-safe dans Javadoc. Les autres exceptions à cette règle sont :

- `SwingUtilities.isEventDispatchThread()`, qui détermine si le thread courant est le thread des événements ;
- `SwingUtilities.invokeLater()`, qui planifie une tâche Runnable pour qu'elle s'exécute dans le thread des événements (appelable à partir de n'importe quel thread) ;
- `SwingUtilities.invokeAndWait()`, qui planifie une tâche Runnable pour qu'elle s'exécute dans le thread des événements et bloque le thread courant jusqu'à ce qu'elle soit terminée (appelable uniquement à partir d'un thread n'appartenant pas à l'interface graphique) ;
- les méthodes pour mettre en attente un réaffichage ou une demande de revalidation dans la file des événements (appelables à partir de n'importe quel thread) ;
- les méthodes pour ajouter et ôter des écouteurs (appelables à partir de n'importe quel thread, mais les écouteurs seront toujours invoqués dans le thread des événements).

Les méthodes `invokeLater()` et `invokeAndWait()` fonctionnent beaucoup comme un Executor. En fait, comme le montre le Listing 9.1, l'implémentation des méthodes de `SwingUtilities` liées aux threads est triviale si l'on se sert d'un Executor monothread. Ce n'est pas de cette façon que `SwingUtilities` est réellement implémentée car Swing est antérieur au framework Executor, mais elle le serait sûrement de cette manière si Swing était écrit aujourd'hui.

Listing 9.1 : Implémentation de SwingUtilities à l'aide d'un Executor.

```

public class SwingUtilities {
    private static final ExecutorService exec =
        Executors.newSingleThreadExecutor(new SwingThreadFactory ());
    private static volatile Thread swingThread;

    private static class SwingThreadFactory implements ThreadFactory {
        public Thread newThread(Runnable r) {
            swingThread = new Thread(r);
            return swingThread;
        }
    }

    public static boolean isEventDispatchThread () {
        return Thread.currentThread() == swingThread;
    }

    public static void invokeLater(Runnable task) {
        exec.execute(task);
    }

    public static void invokeAndWait(Runnable task)
        throws InterruptedException , InvocationTargetException {
        Future f = exec.submit(task);
        try {
            f.get();
        } catch (ExecutionException e) {
            throw new InvocationTargetException (e);
        }
    }
}

```

Le thread des événements Swing peut être vu comme un Executor monothread qui traite les tâches en attente dans la file des événements. Comme pour les pools de threads, le thread exécutant meurt parfois et est remplacé par un nouveau, mais cela devrait être transparent du point de vue des tâches. Une exécution séquentielle, monothread, est une politique d'exécution raisonnable lorsque les tâches sont courtes, que la prévision de l'ordonnancement n'a pas d'importance ou qu'il est impératif que les tâches ne s'exécutent pas en parallèle.

La classe GuiExecutor du Listing 9.2 est un Executor qui délègue les tâches à Swing Utilities pour les exécuter. Elle pourrait également être implémentée à l'aide d'autres frameworks graphiques ; SWT, par exemple, fournit la méthode `Display.asyncExec()`, qui ressemble à `SwingUtilities.invokeLater()`.

Listing 9.2 : Executor construit au-dessus de SwingUtilities.

```

public class GuiExecutor extends AbstractExecutorService {
    // Les singletons ont un constructeur privé et une fabrique publique.
    private static final GuiExecutor instance = new GuiExecutor();

    private GuiExecutor() { }

    public static GuiExecutor instance() { return instance; }
}

```

Listing 9.2 : Executor construit au-dessus de SwingUtilities. (suite)

```

public void execute(Runnable r) {
    if (SwingUtilities.isEventDispatchThread ())
        r.run();
    else
        SwingUtilities.invokeLater(r);
}

// Plus des implémentations triviales des méthodes du cycle de vie.
}

```

9.2 Tâches courtes de l'interface graphique

Dans une application graphique, les événements partent du thread des événements et remontent vers les écouteurs fournis par l'application, qui effectueront probablement certains traitements affectant les objets de présentation. Dans le cas des tâches simples et courtes, toute l'action peut rester dans le thread des événements ; pour les plus longues, une partie du traitement devrait être déchargée vers un autre thread.

Dans le cas le plus simple, le confinement des objets de présentation au thread des événements est totalement naturel. Le Listing 9.3, par exemple, crée un bouton dont la couleur change de façon aléatoire lorsqu'il est enfoncé. Lorsque l'utilisateur clique sur le bouton, le toolkit délivre un événement ActionEvent dans le thread des événements à destination de tous les écouteurs d'action enregistrés. En réponse, l'écouteur d'action choisit une couleur et modifie celle du fond du bouton. L'événement part donc du toolkit graphique, est délivré à l'application et celle-ci modifie l'interface graphique en réponse à l'action de l'utilisateur. Le contrôle n'a jamais besoin de quitter le thread des événements, comme le montre la Figure 9.1.

Listing 9.3 : Écouteur d'événement simple.

```

final Random random = new Random();
final JButton button = new JButton("Change Color");
...
button.addActionListener(new ActionListener () {
    public void actionPerformed(ActionEvent e) {
        button.setBackground(new Color(random.nextInt()));
    }
});

```

**Figure 9.1**

Flux de contrôle d'un simple clic sur un bouton.

Cet exemple trivial représente la majorité des interactions entre les applications et les toolkits graphiques. Tant que les tâches sont courtes et n'accèdent qu'aux objets de l'interface graphique (ou à d'autres objets de l'application confinés au thread ou thread-safe), vous pouvez presque totalement ignorer les problèmes liés aux threads et tout faire à partir du thread d'événement : tout se passera bien.

La Figure 9.2 illustre une version un peu plus compliquée de ce scénario car elle implique un modèle de données formel comme `TableModel` ou `TreeModel`. Swing divise la plupart des composants visuels en deux objets, un modèle et une vue. Les données à afficher résident dans le modèle et les règles gouvernant l'affichage se situent dans la vue. Les objets modèles peuvent déclencher des événements indiquant que les données du modèle ont été modifiées et les vues réagissent à ces événements. Lorsqu'elle reçoit un événement indiquant que les données du modèle ont peut-être changé, la vue interroge le modèle pour obtenir les nouvelles données et met l'affichage à jour. Dans un écouteur de bouton qui modifie le contenu d'une table, l'écouteur mettrait à jour le modèle et appellerait l'une des méthodes `fireXXX()` qui, à son tour, invoquerait les écouteurs du modèle de la table de la vue, qui modifieraient la vue. Là encore, le contrôle ne quitte jamais le thread des événements (les méthodes `fireXXX()` de Swing appelant toujours directement les écouteurs au lieu d'ajouter un nouvel événement dans la file des événements, elles ne doivent être appelées qu'à partir du thread des événements).

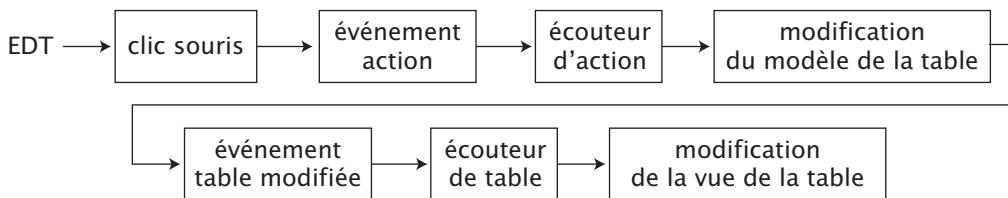


Figure 9.2

Flux de contrôle avec des objets modèle et vue séparés.

9.3 Tâches longues de l'interface graphique

Si toutes les tâches étaient courtes (et si la partie non graphique de l'application n'était pas significative), l'application entière pourrait s'exécuter dans le thread des événements et l'on n'aurait absolument pas besoin de prêter attention aux threads. Cependant, les applications graphiques sophistiquées peuvent exécuter des tâches qui durent plus longtemps que ce que l'utilisateur est prêt à attendre ; c'est le cas, par exemple, d'une vérification orthographique, d'une compilation en arrière-plan ou d'une récupération de ressources distantes. Ces tâches doivent donc s'exécuter dans un autre thread pour que l'interface graphique puisse rester réactive pendant leur déroulement.

Swing facilite l'exécution d'une tâche dans le thread des événements, mais (avant Java 6) il ne fournit aucun mécanisme pour aider les tâches à exécuter leur code dans d'autres threads. Cependant, nous n'avons pas besoin que Swing nous aide, ici, car nous pouvons créer notre propre Executor pour traiter les tâches longues. Un pool de threads en cache est un bon choix pour ce type de tâche ; les applications graphiques ne lançant que très rarement un grand nombre de tâches longues, il y a peu de risques que le pool grossisse sans limite.

Nous commencerons par une tâche simple qui ne supporte pas l'annulation ni d'indication de progression et qui ne modifie pas l'interface graphique lorsqu'elle est terminée, puis nous ajouterons ces fonctionnalités une à une. Le Listing 9.4 montre un écouteur d'action lié à un composant visuel qui soumet une longue tâche à un Executor. Malgré les deux couches de classes internes, il est assez facile de comprendre comment une tâche graphique lance une tâche : l'écouteur d'action de l'interface est appelé dans le thread des événements et soumet l'exécution d'une tâche Runnable au pool de threads.

Listing 9.4 : Liaison d'une tâche longue à un composant visuel.

```
ExecutorService backgroundExec = Executors.newCachedThreadPool ();
...
button.addActionListener(new ActionListener () {
    public void actionPerformed(ActionEvent e) {
        backgroundExec .execute(new Runnable() {
            public void run() { doBigComputation (); }
        });
    }
});
```

Cet exemple sort la longue tâche du thread des événements en s'en "débarrassant", ce qui n'est pas très pratique car on attend généralement un effet visuel lorsque la tâche s'est terminée. Ici, on ne peut pas accéder aux objets de présentation à partir du thread en arrière-plan ; quand elle s'est terminée, la tâche doit donc soumettre une autre tâche pour qu'elle s'exécute dans le thread des événements afin de modifier l'interface utilisateur. Le Listing 9.5 montre la façon évidente de le faire, qui commence quand même à devenir compliquée car nous avons maintenant besoin de trois couches de classes internes. L'écouteur d'action désactive d'abord le bouton et lui associe un texte indiquant qu'un calcul est en cours, puis il soumet une tâche à l'exécuteur en arrière-plan. Lorsque cette tâche se termine, il met en file d'attente une autre tâche pour qu'elle s'exécute dans le thread des événements, ce qui réactive le bouton et restaure le texte de son label.

Listing 9.5 : Tâche longue avec effet visuel.

```
button.addActionListener(new ActionListener () {
    public void actionPerformed(ActionEvent e) {
        button.setEnabled(false);
        label.setText("busy");
        backgroundExec .execute(new Runnable() {
            public void run() {
                try {
                    doBigComputation ();
                } finally {
```

```
        GuiExecutor.instance().execute(new Runnable() {
            public void run() {
                button.setEnabled(true);
                label.setText("idle");
            }
        });
    });
}
});
```

La tâche déclenchée lorsque le bouton est enfoncé est composée de trois sous-tâches séquentielles dont l'exécution alterne entre le thread des événements et le thread en arrière-plan. La première sous-tâche met à jour l'interface utilisateur pour indiquer qu'une opération longue a commencé et lance la seconde dans un thread en arrière-plan. Lorsqu'elle est terminée, cette seconde sous-tâche met en file d'attente la troisième sous-tâche pour qu'elle s'exécute à nouveau dans le thread des événements. Cette sorte de "ping-pong de threads" est typique de la gestion des tâches longues dans les applications graphiques.

9.3.1 Annulation

Toute tâche qui dure assez longtemps dans un autre thread peut inciter l'utilisateur à l'annuler. Nous pourrions implémenter directement l'annulation à l'aide d'une interruption du thread, mais il est bien plus simple d'utiliser Future, qui a été conçue pour gérer les tâches annulables.

Lorsque l'on appelle `cancel()` sur un objet `Future` dont `mayInterruptIfRunning` vaut `true`, l'implémentation de `Future` interrompt le thread de la tâche s'il est en cours d'exécution. Si la tâche a été écrite pour pouvoir répondre aux interruptions, elle peut donc se terminer précocément si elle est annulée. Le Listing 9.6 montre une tâche qui interroge l'indicateur d'interruption du thread et se termine rapidement s'il est positionné.

Listing 9.6 : Annulation d'une tâche longue.

```
Future<?> runningTask = null; // confinée au thread
...
startButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        if (runningTask == null) {
            runningTask = backgroundExec .submit(new Runnable() {
                public void run() {
                    while (moreWork()) {
                        if (Thread.interrupted()) {
                            cleanUpPartialWork();
                            break;
                        }
                        doSomeWork();
                    }
                }
            });
        };
    }
});
```

Listing 9.6 : Annulation d'une tâche longue. (suite)

```
cancelButton.addActionListener(new ActionListener () {
    public void actionPerformed(ActionEvent event) {
        if (runningTask != null)
            runningTask.cancel(true);
    }});
}
```

runningTask étant confinée au thread des événements, il n'y a pas besoin de synchronisation pour la modifier ou la consulter et l'écouteur du bouton de démarrage garantit qu'une seule tâche en arrière-plan s'exécutera à un instant donné. Cependant, il serait préférable d'être prévenu lorsque la tâche se termine pour que, par exemple, le bouton d'annulation puisse être désactivé. C'est ce que nous allons implémenter dans la section qui suit.

9.3.2 Indication de progression et de terminaison

L'utilisation d'un objet Future pour représenter une tâche longue simplifie beaucoup l'implémentation de l'annulation et FutureTask dispose également d'un hook done() qui facilite la notification de la terminaison d'une tâche puisqu'il est appelé lorsque le Callable en arrière-plan s'est terminé. En faisant en sorte que done() lance une tâche de terminaison dans le thread des événements, nous pouvons donc construire une classe BackgroundTask fournissant un hook onCompletion() qui sera appelé dans le thread des événements, comme on le montre dans le Listing 9.7.

Listing 9.7 : Classe de tâche en arrière-plan supportant l'annulation, ainsi que la notification de terminaison et de progression.

```
abstract class BackgroundTask <V> implements Runnable, Future<V> {
    private final FutureTask<V> computation = new Computation();

    private class Computation extends FutureTask<V> {
        public Computation() {
            super(new Callable<V>() {
                public V call() throws Exception {
                    return BackgroundTask.this.compute();
                }
            });
        }
        protected final void done() {
            GuiExecutor.instance().execute(new Runnable() {
                public void run() {
                    V value = null;
                    Throwable thrown = null;
                    boolean cancelled = false;
                    try {
                        value = get();
                    } catch (ExecutionException e) {
                        thrown = e.getCause();
                    } catch (CancellationException e) {
                        cancelled = true;
                    } catch (InterruptedException consumed) {
                    } finally {
                        onCompletion(value, thrown, cancelled);
                    }
                }
            });
        }
    }
}
```

```

protected void setProgress(final int current, final int max) {
    GuiExecutor.instance().execute(new Runnable() {
        public void run() { onProgress(current, max); }
    });
}
// Appelée dans le thread en arrière-plan
protected abstract V compute() throws Exception;
// Appelée dans le thread des événements
protected void onCompletion(V result, Throwable exception,
                           boolean cancelled) { }
protected void onProgress(int current, int max) { }
// Autres méthodes de Future déléguées à Computation
}

```

BackgroundTask permet également d'indiquer la progression de la tâche. La méthode compute() peut appeler setProgress(), qui indique la progression par une valeur numérique. onProgress() sera appelée à partir du thread des événements, qui peut mettre à jour l'interface utilisateur pour montrer visuellement la progression.

Pour implémenter une BackgroundTask il suffit d'écrire compute(), qui est appelée dans le thread en arrière-plan. On peut également redéfinir onCompletion() et onProgress(), qui, elles, sont invoquées dans le thread des événements.

Faire reposer BackgroundTask sur FutureTask permet également de simplifier l'annulation car, au lieu d'interroger l'indicateur d'interruption du thread, compute() peut se contenter d'appeler Future.isCancelled(). Le Listing 9.8 reformule l'exemple du Listing 9.6 en utilisant BackgroundTask.

Listing 9.8 : Utilisation de BackgroundTask pour lancer une tâche longue et annulable.

```

public void runInBackground(final Runnable task) {
    startButton.addActionListener(new ActionListener () {
        public void actionPerformed(ActionEvent e) {
            class CancelListener implements ActionListener {
                BackgroundTask<?> task;
                public void actionPerformed(ActionEvent event) {
                    if (task != null)
                        task.cancel(true);
                }
            }
            final CancelListener listener = new CancelListener ();
            listener.task = new BackgroundTask <Void>() {
                public Void compute() {
                    while (moreWork() && !isCancelled())
                        doSomeWork();
                    return null;
                }
                public void onCompletion(boolean cancelled, String s,
                                       Throwable exception) {
                    cancelButton.removeActionListener(listener);
                    label.setText("done");
                }
            };
            cancelButton.addActionListener(listener);
            backgroundExec.execute(task);
        }
    });
}

```

9.3.3 SwingWorker

Nous avons construit un framework simple utilisant FutureTask et Executor pour exécuter des tâches longues dans des threads en arrière-plan sans détériorer la réactivité de l'interface graphique. Ces techniques peuvent s'appliquer à n'importe quel framework graphique monothread, elles ne sont pas réservées à Swing. Cependant, la plupart des fonctionnalités développées ici – annulation, notification de terminaison et indication de progression – sont fournies par la classe SwingWorker de Swing. Plusieurs versions de cette classe ont été publiées dans la *Swing Connection* et dans le *didacticiel Java* et une version mise à jour a été intégrée à Java 6.

9.4 Modèles de données partagées

Les objets de présentation de Swing, qui comprennent les objets de modèles de données comme `TableModel` ou `TreeModel`, sont confinés dans le thread des événements. Dans les programmes graphiques simples, tout l'état modifiable est contenu dans les objets de présentation et le seul thread qui existe à part le thread des événements est le thread principal. Dans ces programmes, il est facile de respecter la règle monothread : *ne manipulez pas les composants du modèle de données ou de la présentation à partir du thread principal*. Les programmes plus complexes peuvent utiliser d'autres threads pour déplacer les données à partir de ou vers une zone de stockage persistante, comme un système de fichiers ou une base de données, sans pour autant compromettre la réactivité de l'application.

Dans le cas le plus simple, les données du modèle de données sont saisies par l'utilisateur ou chargées statiquement à partir d'un fichier ou d'une autre source au démarrage de l'application, auquel cas les données ne seront jamais touchées par un autre thread que celui des événements. Cependant, l'objet du modèle de présentation n'est parfois qu'une vue vers une autre source de données, comme une base de données, un système de fichiers ou un service distant. Dans ce cas, plusieurs threads manipuleront sûrement les données à mesure qu'elles entrent et sortent de l'application.

Vous pourriez, par exemple, afficher le contenu d'un système de fichiers distant en utilisant un modèle arborescent. Pour cela, vous ne souhaiterez pas énumérer tout le système de fichiers avant de pouvoir afficher toute l'arborescence : cela serait trop long et consommerait trop de mémoire. Au lieu de cela, l'arborescence peut être remplie à mesure que les noeuds sont développés. Même l'énumération d'un petit répertoire situé sur un volume distant pouvant prendre du temps, il est préférable de l'effectuer dans une tâche en arrière-plan. Lorsque celle-ci se termine, il faut trouver un moyen de stocker les données dans le modèle arborescent, ce qui peut se faire au moyen d'un modèle arborescent thread-safe, en "poussant" les données de la tâche en arrière-plan vers le thread des événements. Pour cela, on poste une tâche avec `invokeLater()` ou l'on fait en sorte que le thread des événements regarde si les données sont disponibles.

9.4.1 Modèles de données thread-safe

Tant que la réactivité n'est pas trop affectée par les blocages, le problème des threads multiples manipulant les données peut se régler par un modèle de données thread-safe. Si ce modèle supporte des accès concurrents précis, le thread des événements et ceux en arrière-plan devraient pouvoir le partager sans perturber la réactivité. La classe `DelegatingVehicleTracker` du Listing 4.7, par exemple, utilise un objet `ConcurrentHashMap` sous-jacent dont les opérations de récupération autorisent un haut degré de concurrence. L'inconvénient est qu'il n'offre pas un instantané cohérent des données, ce qui peut être, ou non, une nécessité. En outre, les modèles de données thread-safe doivent produire des événements lorsque le modèle a été modifié, afin que les vues puissent également l'être.

On peut parfois obtenir la thread safety, la cohérence et une bonne réactivité en utilisant un modèle de données *versionné* comme `CopyOnWriteArrayList` [CPJ 2.2.3.3]. Un itérateur sur ce type de collection la parcourt telle qu'elle était lorsqu'il a été créé. Cependant, ces collections n'offrent de bonnes performances que lorsque le nombre des parcours est bien supérieur à celui des modifications, ce qui ne sera sûrement pas le cas avec, par exemple, une application de suivi de véhicules. Des structures de données versionnées plus spécialisées peuvent éviter ce défaut, mais construire des structures qui à la fois fournissent un accès concurrent efficace et ne mémorisent pas les anciennes versions des données plus longtemps qu'il n'est nécessaire n'est pas chose facile ; cela ne devrait être envisagé que si les autres approches ne sont pas utilisables.

9.4.2 Modèles de données séparés

Du point de vue de l'interface graphique, les classes de Swing comme `TableModel` et `TreeModel` sont les dépôts officiels pour les données à afficher. Cependant, ces objets modèles sont souvent eux-mêmes des "vues" d'autres objets gérés par l'application. Un programme qui possède à la fois un modèle de données pour la présentation et un autre pour l'application repose sur un *modèle de séparation* (Fowler, 2005).

Dans un modèle de séparation, le modèle de présentation est confiné dans le thread des événements et l'autre modèle, le *modèle partagé*, est thread-safe et à la fois le thread des événements et les threads de l'application peuvent y accéder. Le modèle de présentation enregistre les écouteurs avec le modèle partagé afin de pouvoir être prévenu des mises à jour. Il peut alors être modifié à partir du modèle partagé en intégrant un instantané de l'état pertinent dans le message de mise à jour ou en faisant en sorte de récupérer directement les données à partir du modèle partagé lorsqu'il reçoit un événement de mise à jour. L'approche par instantané est simple mais possède quelques limitations. Elle fonctionne bien quand le modèle de données est petit, que les mises à jour ne sont pas trop fréquentes et que les structures des deux modèles se ressemblent. Lorsque le modèle de données a une taille importante, que les mises à jour sont très fréquentes ou que l'une ou l'autre des deux parties de la séparation contient des informations non visibles à l'autre, il peut être plus efficace d'envoyer des mises à jour incrémentales au lieu d'instantanés entiers.

Cette approche revient à sérialiser les mises à jour sur le modèle partagé et à les recréer dans les threads des événements par rapport au modèle de présentation. Un autre avantage des mises à jour incrémentales est qu'une information plus fine sur ce qui a été modifié permet d'améliorer la qualité perçue de l'affichage – si un seul véhicule se déplace, nous n'avons pas besoin de réafficher tout l'écran, mais uniquement les régions concernées.

Un modèle de séparation est un bon choix lorsqu'un modèle de données doit être partagé par plusieurs threads et qu'implémenter un modèle thread-safe serait inadapté pour des raisons de blocage, de cohérence ou de complexité.

9.5 Autres formes de sous-systèmes monothreads

Le confinement au thread n'est pas réservé aux interfaces graphiques : il peut être utilisé à chaque fois qu'une fonctionnalité est implémentée sous la forme d'un sous-système monothread. Parfois, ce confinement est imposé au développeur pour des raisons qui n'ont rien à voir avec la synchronisation ou les interblocages : certaines bibliothèques natives, par exemple, exigent que tout accès à la bibliothèque, même son chargement avec `System.loadLibrary()`, ait lieu dans le même thread.

En empruntant l'approche des frameworks graphiques, on peut aisément créer un thread dédié ou un exécuteur monothread pour accéder à cette bibliothèque native et fournir un objet mandataire qui intercepte les appels à l'objet confiné au thread et les soumet sous forme de tâches au thread dédié. `Future` et `newSingleThreadExecutor()` facilitent cette implémentation : la méthode mandataire peut soumettre la tâche et immédiatement appeler `Future.get()` pour attendre le résultat (si la classe qui doit être confinée à un thread implémente une interface, on peut automatiser le processus consistant à ce que chaque méthode soumette un objet `Callable` à un thread en arrière-plan et à attendre le résultat en utilisant des mandataires dynamiques).

Résumé

Les frameworks graphiques sont quasiment toujours implémentés comme des sous-systèmes monothreads dans lesquels tout le code lié à la présentation s'exécute sous forme de tâches dans un thread des événements. Comme il n'y a qu'un seul thread des événements, les tâches longues peuvent compromettre la réactivité de l'interface et doivent donc s'exécuter dans des threads en tâche de fond. Les classes utilitaires comme `SwingWorker` ou la classe `BackgroundTask` que nous avons construite, qui fournissent un support pour l'annulation, l'indication de la progression et de la terminaison d'une tâche, permettent de simplifier le développement des tâches longues qui possèdent des composants graphiques et non graphiques.

III

Vivacité, performances et tests

10

Éviter les problèmes de vivacité

Il y a souvent des conflits entre la thread safety et la vivacité. On utilise le verrouillage pour garantir la première, mais son utilisation irréfléchie peut causer des interblocages liés à l'ordre des verrous. De même, on utilise des pools de threads et des sémaphores pour limiter la consommation des ressources, mais une mauvaise conception de cette limitation peut impliquer des interblocages de ressources. Les applications Java ne pouvant pas se remettre d'un interblocage, il faut s'assurer que l'on a pris soin d'écartier les situations qui peuvent les provoquer. Dans ce chapitre, nous présenterons certaines causes des problèmes de vivacité et nous verrons comment les éviter.

10.1 Interblocages (**deadlock**)

Les interblocages sont illustrés par le problème classique, bien que peu hygiénique, du "dîner des philosophes". Cinq philosophes sont assis autour d'une table ronde dans un restaurant chinois, il y a cinq baguettes sur la table (pas cinq paires), chacune étant placée entre deux assiettes ; les philosophes alternent leurs activités entre réfléchir et manger. Pour manger, un philosophe doit posséder les deux baguettes de chaque côté de son assiette suffisamment longtemps, puis il les repose sur la table et repart dans ses réflexions. Certains algorithmes de gestion des baguettes permettent à chacun de manger plus ou moins régulièrement (un philosophe affamé essaiera de prendre ses deux baguettes mais, s'il n'y en a qu'une de disponible, il reposera l'autre et attendra une minute avant de réessayer), d'autres peuvent faire que tous les philosophes meurent de faim (chacun d'eux prend immédiatement la baguette située à sa gauche et attend que la droite soit disponible avant de la reposer). Cette dernière situation, où chacun possède une ressource nécessaire à l'autre, attend une ressource détenue par un autre et ne relâche pas la sienne tant qu'il n'obtient pas celle qu'il n'a pas, illustre parfaitement le problème des interblocages.

Lorsqu'un thread détient un verrou indéfiniment, ceux qui tentent de l'obtenir seront bloqués en permanence. Si le thread *A* détient le verrou *V* et tente d'acquérir le verrou *W* alors qu'au même moment le thread *B* détient *W* et tente d'obtenir *V*, les deux threads attendront indéfiniment. Cette situation est le cas d'interblocage le plus simple (on l'appelle "étreinte fatale"), où plusieurs threads sont bloqués définitivement à cause d'une dépendance cyclique du verrouillage (considérez les threads comme les nœuds d'un graphe orienté dont les arcs représentent la relation "le thread *A* attend une ressource détenue par le thread *B*". Si ce graphe est cyclique, alors, il y a interblocage).

Les systèmes de gestion de bases de données sont conçus pour détecter les interblocages et s'en échapper. Une transaction peut acquérir plusieurs verrous qui seront détenus jusqu'à ce qu'elle soit validée : il est donc possible, et fréquent, que deux transactions se bloquent mutuellement. Sans intervention, elles attendraient donc indéfiniment (en détenant des verrous qui sont sûrement nécessaires à d'autres transactions). Cependant, le serveur de base de données ne laissera pas cette situation arriver : quand il détecte qu'un ensemble de transactions sont interbloquées (en recherchant les cycles dans le graphe des verrous), il choisit une victime et annule sa transaction, ce qui libère le verrou qu'elle détenait et permet aux autres de continuer. L'application peut ensuite retenter la transaction annulée, qui peut se poursuivre maintenant que les transactions concurrentes se sont terminées.

La JVM n'est pas tout à fait aussi efficace que les serveurs de bases de données pour résoudre les interblocages. Lorsqu'un ensemble de threads Java se bloquent mutuellement, c'est la fin du jeu – ces threads sont définitivement hors de course. Selon ce qu'ils faisaient, toute l'application ou simplement l'un de ses sous-systèmes peut se figer, ou les performances peuvent chuter. La seule façon de remettre les choses en place est alors de l'arrêter et de la relancer – en espérant que le problème ne se reposera pas de nouveau.

Comme la plupart des autres problèmes de concurrence, les interblocages se manifestent rarement immédiatement. Le fait qu'une classe puisse provoquer un interblocage ne signifie pas qu'elle le fera ; c'est uniquement une possibilité. Lorsque les interblocages apparaissent, c'est généralement au pire moment – en production et sous une charge importante.

10.1.1 Interblocages liés à l'ordre du verrouillage

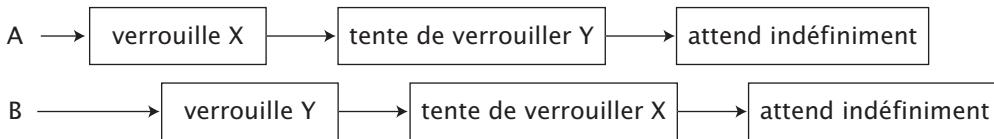
La classe `LeftRightDeadlock` du Listing 10.1 risque de provoquer un interblocage car les méthodes `leftRight()` et `rightLeft()` prennent, respectivement, les verrous `left` et `right`. Si un thread appelle `leftRight()` et un autre, `rightLeft()`, et que leurs actions s'entrelacent comme à la Figure 10.1, ils se bloqueront mutuellement.

Listing 10.1 : Interblocage simple lié à l'ordre du verrouillage. Ne le faites pas.

```
// Attention : risque d'interblocage !
public class LeftRightDeadlock {
    private final Object left = new Object();
    private final Object right = new Object();

    public void leftRight() {
        synchronized (left) {
            synchronized (right) {
                doSomething();
            }
        }
    }

    public void rightLeft() {
        synchronized (right) {
            synchronized (left) {
                doSomethingElse();
            }
        }
    }
}
```

**Figure 10.1***Timing malheureux dans LeftRightDeadlock.*

L'interblocage de `LeftRightDeadlock` provient du fait que les deux threads ont tenté de prendre les mêmes verrous *dans un ordre différent*. S'ils les avaient demandés dans le même ordre, il n'y aurait pas eu de dépendance cyclique entre les verrous et donc aucun interblocage. Si l'on peut garantir que chaque thread qui a besoin simultanément des verrous V et W les prendra dans le même ordre, il n'y aura pas d'interblocage.

Un programme est immunisé contre les interblocages liés à l'ordre de verrouillage si tous les threads prennent les verrous dont ils ont besoin selon un ordre global bien établi.

La vérification d'un verrouillage cohérent exige une analyse globale du verrouillage dans le programme. Il ne suffit pas d'inspecter individuellement les extraits du code qui prennent plusieurs verrous : `leftRight()` et `rightLeft()` acquièrent "correctement" les deux verrous, ce qui ne les empêche pas d'être incompatibles. Avec le verrouillage, la main gauche doit savoir ce que fait la main droite.

10.1.2 Interblocages dynamiques liés à l'ordre du verrouillage

Il n'est pas toujours évident d'avoir un contrôle suffisant sur l'ordre du verrouillage afin d'éviter les interblocages. Étudiez par exemple le code apparemment inoffensif du Listing 10.2, qui transfère des fonds d'un compte à un autre. Ce code verrouille les deux objets Account avant d'exécuter le transfert pour s'assurer que les comptes seront modifiés de façon atomique sans violer des invariants comme "un compte ne peut pas avoir une balance négative".

Listing 10.2 : Interblocage dynamique lié à l'ordre du verrouillage. Ne le faites pas.

```
// Attention : risque d'interblocage !
public void transferMoney(Account fromAccount,
                           Account toAccount,
                           DollarAmount amount)
                           throws InsufficientFundsException {
    synchronized (fromAccount) {
        synchronized (toAccount) {
            if (fromAccount.getBalance().compareTo(amount) < 0)
                throw new InsufficientFundsException();
            else {
                fromAccount.debit(amount);
                toAccount.credit(amount);
            }
        }
    }
}
```



Comment l'interblocage peut-il survenir ? Il peut sembler que tous les threads prennent leurs verrous dans le même ordre, or cet ordre dépend de celui des paramètres passés à `transferMoney()`, qui à leur tour dépendent de données externes. Un interblocage peut donc survenir si deux threads appellent `transferMoney()` en même temps, l'un transférant de X vers Y, l'autre dans le sens inverse :

```
A : transferMoney(monCompte, tonCompte, 10);
B : transferMoney(tonCompte, monCompte, 20);
```

Avec un timing malheureux, A prendra le verrou sur `monCompte` et attendra celui sur `tonCompte`, tandis que B détiendra le verrou sur `tonCompte` et attendra celui sur `monCompte`.

Ce type d'interblocage peut être détecté de la même façon que celui du Listing 10.1 – il suffit de rechercher les prises de verrous imbriquées. Comme, ici, l'ordre des paramètres ne dépend pas de nous, il faut, pour résoudre le problème, induire un ordre sur les verrous et toujours les prendre dans cet ordre tout au long de l'application.

Un bon moyen d'induire un ordre sur les objets consiste à utiliser `System.identityHashCode()`, qui renvoie la valeur qui serait retournée par `Object.hashCode()`. Le Listing 10.3 montre une version de `transferMoney` qui utilise cette méthode pour induire un ordre de verrouillage. Bien qu'elle ajoute quelques lignes de code, elle élimine l'éventualité d'un interblocage.

Listing 10.3 : Induire un ordre de verrouillage pour éviter les interblocages.

```
private static final Object tieLock = new Object();

public void transferMoney(final Account fromAcct,
                           final Account toAcct,
                           final DollarAmount amount)
                           throws InsufficientFundsException {
    class Helper {
        public void transfer() throws InsufficientFundsException {
            if (fromAcct.getBalance().compareTo(amount) < 0)
                throw new InsufficientFundsException ();
            else {
                fromAcct.debit(amount);
                toAcct.credit(amount);
            }
        }
    }
    int fromHash = System.identityHashCode(fromAcct);
    int toHash = System.identityHashCode(toAcct);

    if (fromHash < toHash) {
        synchronized (fromAcct) {
            synchronized (toAcct) {
                new Helper().transfer();
            }
        }
    } else if (fromHash > toHash) {
        synchronized (toAcct) {
            synchronized (fromAcct) {
                new Helper().transfer();
            }
        }
    } else {
        synchronized (tieLock) {
            synchronized (fromAcct) {
                synchronized (toAcct) {
                    new Helper().transfer();
                }
            }
        }
    }
}
```

Pour les rares cas où deux objets auraient le même code de hachage, il faut utiliser un moyen arbitraire d'ordonner les prises de verrou, ce qui réintroduit la possibilité d'un interblocage. Pour empêcher un ordre incohérent dans ce cas précis, on utilise donc un troisième verrou pour éviter les "ex aequo". En prenant le verrou `tieLock` avant n'importe quel verrou `Account`, on garantit qu'un seul thread à la fois prendra le risque d'acquérir deux verrous dans un ordre quelconque, ce qui élimine le risque d'interblocage (à condition que ce mécanisme soit utilisé de façon cohérente). Si les collisions de hachages étaient fréquentes, cette technique pourrait devenir un goulot d'étranglement pour la concurrence (exactement comme l'utilisation d'un seul verrou pour tout le programme) mais, ces collisions étant très rares avec `System.identityHashCode()`, elle fournit la dernière touche de sécurité à moindre coût.

Si `Account` a une clé unique, comparable et non modifiable, comme un numéro de compte bancaire, l'induction d'un ordre sur les verrous est encore plus facile : il suffit d'ordonner les objets selon leurs clés, ce qui élimine le recours à un verrou d'égalité.

Vous pourriez penser que nous exagérons le risque d'interblocage car les verrous ne sont généralement détenus que pendant un temps très court, mais ces interblocages posent de sérieux problèmes dans les vrais systèmes. Une application en production peut réaliser des millions de cycles verrouillage-déverrouillage par jour et il en suffit d'un seul qui arrive au mauvais moment pour bloquer tout le programme. En outre, même des tests soigneux en régime de charge peuvent ne pas déceler tous les interblocages possibles¹. La classe `DemonstrateDeadlock`² du Listing 10.4, par exemple, provoquera assez rapidement un interblocage sur la plupart des systèmes.

Listing 10.4 : Boucle provoquant un interblocage dans une situation normale.

```
public class DemonstrateDeadlock {
    private static final int NUM_THREADS = 20;
    private static final int NUM_ACCOUNTS = 5;
    private static final int NUM_ITERATIONS = 1000000;

    public static void main(String[] args) {
        final Random rnd = new Random();
        final Account[] accounts = new Account[NUM_ACCOUNTS];

        for (int i = 0; i < accounts.length; i++)
            accounts[i] = new Account();

        class TransferThread extends Thread {
            public void run() {
                for (int i=0; i<NUM_ITERATIONS ; i++) {
                    int fromAcct = rnd.nextInt(NUM_ACCOUNTS);
                    int toAcct = rnd.nextInt(NUM_ACCOUNTS);
                    DollarAmount amount = new DollarAmount(rnd.nextInt(1000));
                    transferMoney(accounts[fromAcct],
                                  accounts[toAcct], amount);
                }
            }
        }
        for (int i = 0; i < NUM_THREADS; i++)
            new TransferThread().start();
    }
}
```

1. Ironiquement, la détention de verrous pendant des instants brefs, comme on est supposé le faire pour réduire la compétition pour ces verrous, augmente la probabilité que les tests ne découvrent pas les risques potentiels d'interblocages.

2. Pour rester simple, `DemonstrateDeadlock` ne tient pas compte du problème des balances de comptes négatives.

10.1.3 Interblocages entre objets coopératifs

L'acquisition de plusieurs verrous n'est pas toujours aussi évidente que `LeftRightDead lock` ou `transferMoney()` car les deux verrous peuvent ne pas être pris par la même méthode. Le Listing 10.5 contient deux classes qui pourraient coopérer au sein d'une application de répartition de taxis. La classe `Taxi` représente un taxi particulier avec un emplacement et une destination, la classe `Dispatcher` représente une flotte de taxis.

Listing 10.5 : Interblocage lié à l'ordre du verrouillage entre des objets coopératifs. Ne le faites pas.

```
// Attention : risque d'interblocage !
class Taxi {
    @GuardedBy("this") private Point location, destination;
    private final Dispatcher dispatcher;

    public Taxi(Dispatcher dispatcher) {
        this.dispatcher = dispatcher;
    }

    public synchronized Point getLocation() {
        return location;
    }

    public synchronized void setLocation(Point location) {
        this.location = location;
        if (location.equals(destination))
            dispatcher.notifyAvailable (this);
    }
}

class Dispatcher {
    @GuardedBy("this") private final Set<Taxi> taxis;
    @GuardedBy("this") private final Set<Taxi> availableTaxis ;

    public Dispatcher() {
        taxis = new HashSet<Taxi>();
        availableTaxis = new HashSet<Taxi>();
    }

    public synchronized void notifyAvailable (Taxi taxi) {
        availableTaxis .add(taxi);
    }

    public synchronized Image getImage() {
        Image image = new Image();
        for (Taxi t : taxis)
            image.drawMarker(t.getLocation());
        return image;
    }
}
```



Bien qu'aucune méthode ne prenne explicitement deux verrous, ceux qui appellent `set Location()` et `getImage()` peuvent très bien le faire. Un thread appelant `setLocation()` en réponse à une mise à jour d'un récepteur GPS commence par modifier l'emplacement du taxi puis vérifie s'il a atteint sa destination. Si c'est le cas, il informe le répartiteur qu'il attend une nouvelle destination. Comme `setLocation()` et `notifyAvailable()` sont deux méthodes `synchronized`, le thread qui appelle `setLocation()` prend le verrou

Taxi puis le verrou Dispatcher. De même, un thread appelant `getImage()` prend le verrou Dispatcher puis chaque verrou Taxi (un à la fois). Tout comme dans `LeftRightDeadlock`, deux verrous sont donc pris par deux threads dans un ordre différent, ce qui crée un risque d'interblocage.

Il était assez simple de détecter la possibilité d'un interblocage dans `LeftRightDeadlock` ou `transferMoney()` en examinant les méthodes qui prenaient deux verrous. Dans `Taxi` et `Dispatcher`, en revanche, c'est un peu plus difficile ; le signal rouge est qu'une méthode étrangère est appelée pendant que l'on détient un verrou, ce qui risque de poser un problème de vivacité. En effet, cette méthode étrangère peut prendre d'autres verrous (et donc risquer un interblocage) ou se bloquer pendant un temps inhabituellement long et figer les autres threads qui ont besoin du verrou que l'on détient.

10.1.4 Appels ouverts

`Taxi` et `Dispatcher` ne savent évidemment pas qu'ils sont les deux moitiés d'un interblocage possible. Ils ne devraient pas le savoir, d'ailleurs : un appel de méthode est une barrière d'abstraction conçue pour masquer les détails de ce qui se passe de l'autre côté. Cependant, comme on ne sait pas ce qui se passe de l'autre côté de l'appel, *l'invocation d'une méthode étrangère quand on détient un verrou est difficile à analyser et donc risqué*.

L'appel d'une méthode lorsqu'on ne détient pas de verrou est un *appel ouvert* [CPJ 2.4.1.3] et les classes qui utilisent ce type d'appel se comportent et se combinent mieux que celles qui font des appels en détenant des verrous. L'utilisation d'appels ouverts pour éviter les interblocages est analogue à l'utilisation de l'encapsulation pour la thread safety : bien que l'on puisse construire un programme thread-safe sans aucune encapsulation, il est bien plus simple d'analyser la thread safety d'un programme qui l'utilise. De même, l'analyse de la vivacité d'un programme qui utilise uniquement des appels ouverts est bien plus facile. Se limiter à des appels ouverts permet de repérer beaucoup plus facilement les endroits du code qui prennent plusieurs verrous et donc de garantir que ces verrous sont pris dans le bon ordre¹.

Les classes `Taxi` et `Dispatcher` du Listing 10.5 peuvent aisément être modifiées pour utiliser des appels ouverts et ainsi éliminer le risque d'interblocage. Pour cela, il faut réduire les blocs `synchronized` aux seules opérations portant sur l'état partagé, comme on le fait dans le Listing 10.6. Très souvent, les problèmes comme ceux du Listing 10.5 proviennent de l'utilisation de méthodes `synchronized` au lieu de blocs `synchronized` plus petits, souvent parce que c'est plus simple à écrire et non parce que la méthode entière doit être protégée par un verrou (en outre, la réduction d'un bloc `synchronized` permet également d'améliorer l'adaptabilité, comme nous le verrons dans la section 11.4.1).

1. La nécessité d'utiliser des appels ouverts et d'ordonner soigneusement les verrous est le reflet du désordre fondamental consistant à composer des objets synchronisés au lieu de synchroniser des objets composés.

Efforcez-vous d'utiliser des appels ouverts partout dans vos programmes, car ils seront bien plus simples à analyser pour rechercher les éventuels risques d'interblocages que ceux qui appellent des méthodes étrangères en détenant un verrou.

Listing 10.6 : Utilisation d'appels ouverts pour éviter l'interblocage entre des objets coopératifs.

```
@ThreadSafe
class Taxi {
    @GuardedBy("this") private Point location, destination;
    private final Dispatcher dispatcher;
    ...
    public synchronized Point getLocation() {
        return location;
    }

    public void setLocation(Point location) {
        boolean reachedDestination ;
        synchronized (this) {
            this.location = location;
            reachedDestination = location.equals(destination);
        }
        if (reachedDestination )
            dispatcher.notifyAvailable(this);
    }
}

@ThreadSafe
class Dispatcher {
    @GuardedBy("this") private final Set<Taxi> taxis;
    @GuardedBy("this") private final Set<Taxi> availableTaxis ;
    ...
    public synchronized void notifyAvailable(Taxi taxi) {
        availableTaxis .add(taxi);
    }

    public Image getImage() {
        Set<Taxi> copy;
        synchronized (this) {
            copy = new HashSet<Taxi>(taxis);
        }
        Image image = new Image();
        for (Taxi t : copy)
            image.drawMarker(t.getLocation());
        return image;
    }
}
```

Restructurer un bloc `synchronized` pour permettre des appels ouverts peut parfois avoir des conséquences néfastes car cela transforme une opération qui était atomique en opération non atomique. Dans la plupart des cas, la perte de l'atomicité est tout à fait acceptable ; il n'y a aucune raison que la mise à jour de l'emplacement d'un taxi et le fait de prévenir le répartiteur qu'il est prêt pour une nouvelle destination soit une opération atomique. Dans d'autres cas, la perte d'atomicité est notable mais la modification reste acceptable ; dans la version sujette aux interblocages, `getImage()` produit un instantané

complet des emplacements de la flotte à cet instant, alors que, dans la version modifiée, elle récupère les emplacements de chaque taxi à des instants légèrement différents.

Dans certains cas, en revanche, la perte d'atomicité pose problème et il faut alors utiliser une autre technique. L'une consiste à structurer un objet concurrent pour qu'un seul thread puisse exécuter le morceau de code qui suit l'appel ouvert. Lorsque l'on arrête un service, par exemple, on peut vouloir attendre que les opérations en cours se terminent, puis libérer les ressources utilisées par le service. Détenir le verrou du service pendant que l'on attend que les opérations se terminent est intrinsèquement une source d'interblocage, alors que relâcher ce verrou avant l'arrêt du service peut permettre à d'autres threads de lancer de nouvelles opérations. La solution consiste alors à conserver le verrou suffisamment longtemps pour mettre l'état du service à "en cours d'arrêt", afin que les autres threads qui veulent lancer de nouvelles opérations – y compris l'arrêt du service – sachent que le service n'est pas disponible et n'essaient donc pas. On peut alors attendre la fin de l'arrêt en sachant que seul le thread d'arrêt a accès à l'état du service après la fin de l'appel ouvert. Au lieu d'utiliser un verrouillage pour maintenir les autres threads en dehors d'une section critique, cette technique repose donc sur la construction de protocoles qui empêchent les autres threads de tenter d'y entrer.

10.1.5 Interblocages liés aux ressources

Tout comme les threads peuvent se bloquer mutuellement lorsque chacun d'eux attend un verrou que l'autre détient et ne relâchera pas, ils peuvent également s'interbloquer lorsqu'ils attendent des ressources. Les pools de ressources sont généralement implementés à l'aide de sémaphores (voir la section 5.5.3) pour faciliter le blocage lorsque le pool est vide. Supposons que l'on dispose de deux ressources en pool pour des connexions à des bases de données différentes : si une tâche a besoin de connexions vers les deux bases de données et que les deux ressources ne sont pas toujours demandées dans le même ordre, le thread *A* pourrait détenir une connexion vers la base *B1* tandis qu'il attendrait une connexion vers la base *B2* ; le thread *B*, quant à lui, pourrait posséder une connexion vers *B2* et attendre une connexion vers *B1* (plus les pools sont grands, plus ce risque est réduit ; si chaque pool a N connexions, un interblocage nécessitera N ensembles de threads en attente cyclique et un timing vraiment très malheureux).

L'*interblocage par famine de thread* est une autre forme d'interblocage lié aux ressources. Nous avons vu dans la section 8.1.1 un exemple de ce problème dans lequel une tâche qui soumet une tâche et attend son résultat s'exécute dans un Executor monothread. En ce cas, la première tâche attendra indéfiniment, ce qui la bloquera, ainsi que toutes celles qui attendent de s'exécuter dans cet Executor. Les tâches qui attendent les résultats d'autres tâches sont les principales sources d'interblocage par famine de threads ; les pools bornés et les tâches interdépendantes ne vont pas bien ensemble.

10.2 Éviter et diagnostiquer les interblocages

Un programme qui ne prend jamais plus d'un verrou à la fois ne risque pas d'interblocage lié à l'ordre du verrouillage. Ce n'est, évidemment, pas toujours possible mais, si l'on peut s'en contenter, cela demande beaucoup moins de travail. Si l'on doit prendre plusieurs verrous, leur ordre doit faire partie de la conception : on doit essayer de minimiser le nombre d'interactions potentielles entre eux et respecter et décrire un protocole pour ordonner les verrous susceptibles d'être pris ensemble.

Dans les programmes qui utilisent un verrouillage fin, on utilise une stratégie en deux parties : on identifie d'abord les endroits où plusieurs verrous pourraient être pris (en faisant en sorte qu'ils soient les moins nombreux possible), puis on effectue une analyse globale de toutes ces instances pour s'assurer que l'ordre de verrouillage est cohérent dans l'ensemble du programme. L'utilisation d'appels ouverts à chaque fois que cela est possible permet de simplifier cette analyse de façon non négligeable. Avec des appels non ouverts, il est assez facile de trouver les endroits où plusieurs verrous sont pris, soit en relisant le code soit par une analyse automatique du pseudo-code ou du code source.

10.2.1 Tentatives de verrouillage avec expiration

Une autre technique pour détecter les interblocages et en repartir consiste à utiliser la fonctionnalité `tryLock()` temporisée des classes `Lock` explicites (voir Chapitre 13) au lieu du verrouillage interne. Alors que les verrous internes attendent indéfiniment s'ils ne peuvent pas prendre le verrou, les verrous explicites permettent de préciser un délai d'expiration après lequel `tryLock()` renverra un échec. En utilisant un délai à peine supérieur à celui dans lequel on s'attend à prendre le verrou, on peut reprendre le contrôle en cas d'inattendu (le Listing 13.3 montre une autre implémentation de `transferMoney()` utilisant `tryLock()` avec de nouvelles tentatives pour augmenter la probabilité d'éviter les interblocages).

On ne peut pas toujours savoir pourquoi une tentative de verrouillage avec délai échoue. Il y a peut-être eu un interblocage ou un thread est entré par erreur dans une boucle infinie alors qu'il détenait ce verrou ou une activité s'exécute plus lentement que prévu. Quoi qu'il en soit, on a au moins la possibilité d'enregistrer que la tentative a échoué, d'inscrire dans un journal des informations utiles sur ce que l'on essayait de faire et de recommencer le traitement autrement qu'en tuant tout le processus.

La prise de verrous avec délais d'expiration peut se révéler efficace même lorsqu'elle n'est pas utilisée partout dans le programme. Si une prise de verrou ne s'effectue pas dans le délai imparti, on peut relâcher les verrous, attendre un peu et réessayer : la situation d'interblocage aura peut-être disparu et le programme pourra ainsi repartir (cette technique ne fonctionne que lorsque l'on prend deux verrous ensemble ; si l'on prend plusieurs verrous à cause d'une imbrication d'appels de méthodes, on ne peut pas se contenter de relâcher le verrou le plus externe, même si l'on sait qu'on le détient).

10.2.2 Analyse des interblocages avec les traces des threads

Bien qu'éviter les interblocages soit essentiellement *notre* problème, la JVM peut nous aider à les identifier à l'aide des *traces de threads*. Une trace de thread contient une pile d'appels pour chaque thread en cours d'exécution, qui ressemble à la pile des appels qui accompagne une exception. Cette trace contient également des informations sur le verrouillage, notamment quels sont les verrous détenus par chaque thread, dans quel cadre de pile ils ont été acquis et quel est le verrou qui bloque un thread¹. Avant de produire une trace de thread, la JVM recherche les cycles dans le graphe des threads en attente afin de trouver les interblocages. Si elle en trouve un, elle inclut des informations qui permettent d'identifier les verrous et les threads concernés, ainsi que l'endroit du programme où ont lieu ces prises de verrous.

Pour déclencher une trace de thread, il suffit d'envoyer le signal `SIGQUIT` (`kill -3`) au processus de la JVM sur les plates-formes Unix (ou `Ctrl+\`) ou `Ctrl+Break` avec Windows. La plupart des environnements intégrés de développement permettent également de créer une trace de thread.

Java 5.0 ne permet pas d'obtenir des informations sur les verrous explicites : ils n'apparaîtront donc pas du tout dans les traces de thread. Bien que Java 6 ait ajouté le support de ces verrous et la détection des interblocages avec les Lock explicites, les informations sur les emplacements où ils sont acquis sont nécessairement moins précises qu'avec les verrous internes. En effet, ces derniers sont associés au cadre de pile dans lequel ils ont été pris alors que les Lock explicites ne sont associés qu'au thread qui les a pris.

Le Listing 10.7 montre des parties d'une trace de thread tirée d'une application J2EE en production. L'erreur qui a provoqué l'interblocage implique trois composants – une application J2EE, un conteneur J2EE et un pilote JDBC, chacun étant fourni par des éditeurs différents (les noms ont été modifiés pour protéger le coupable). Tous les trois étaient des produits commerciaux qui sont passés par des phases de tests intensifs ; chacun comprenait un bogue inoffensif jusqu'à ce qu'ils soient mis ensemble et causent une erreur fatale du serveur.

Listing 10.7 : Portion d'une trace de thread après un interblocage.

```
Found one Java-level deadlock:  
=====  
"ApplicationServerThread " :  
    waiting to lock monitor 0x080f0cdc (a MumbleDBConnection ),  
    which is held by "ApplicationServerThread "  
"ApplicationServerThread " :  
    waiting to lock monitor 0x080f0ed4 (a MumbleDBCallableStatement ),  
    which is held by "ApplicationServerThread "
```

1. Ces informations sont utiles pour le débogage, même s'il n'y a pas d'interblocage. Le déclenchement périodique des traces de threads permet d'observer le comportement d'un programme vis-à-vis du verrouillage.

```
Java stack information for the threads listed above:  
"ApplicationServerThread ":  
    at MumbleDBConnection .remove_statement  
    - waiting to lock <0x650f7f30> (a MumbleDBConnection )  
    at MumbleDBStatement .close  
    - locked <0x6024ffb0> (a MumbleDBCStatement )  
...  
  
"ApplicationServerThread ":  
    at MumbleDBCStatement .sendBatch  
    - waiting to lock <0x6024ffb0> (a MumbleDBCStatement )  
    at MumbleDBConnection .commit  
    - locked <0x650f7f30> (a MumbleDBConnection )  
...
```

Nous n'avons montré que la partie de la trace permettant d'identifier l'interblocage. La JVM nous a beaucoup aidés en le diagnostiquant – elle nous montre les verrous qui ont causé le problème, les threads impliqués et nous indique si d'autres threads ont été indirectement touchés. L'un des threads détient le verrou sur l'objet `MumbleDBConnection` et attend d'en prendre un autre sur l'objet `MumbleDBCStatement` ; l'autre détient le verrou sur le `MumbleDBCStatement` et attend celui sur `MumbleDBConnection`.

Le pilote JDBC utilisé ici souffre clairement d'un bogue lié à l'ordre du verrouillage : différentes chaînes d'appels passant par ce pilote acquièrent plusieurs verrous dans des ordres différents. Mais ce problème ne se serait pas manifesté s'il n'y avait pas un autre bogue : les différents threads tentent d'utiliser le même objet `Connection` JDBC en même temps. Ce n'est pas comme cela que l'application était censée fonctionner – les développeurs ont été surpris de voir cet objet `Connection` utilisé simultanément par deux threads. Rien dans la spécification de JDBC n'exige que `Connection` soit thread-safe et il est assez courant de confiner son utilisation dans un seul thread, comme c'était prévu ici. Cet éditeur a tenté de fournir un pilote JDBC thread-safe, comme on peut le constater par la synchronisation sur les différents objets JDBC utilisée dans le code du pilote. Malheureusement, l'éditeur n'ayant pas pris en compte l'ordre du verrouillage, le pilote est exposé aux interblocages et c'est son interaction avec le partage incorrect de `Connection` par l'application qui a provoqué le problème. Comme aucun des deux bogues n'était fatal pris séparément, tous les deux sont passés à travers les mailles du filet des tests.

10.3 Autres problèmes de vivacité

Bien que les interblocages soient les problèmes de vivacité les plus fréquents, il en existe d'autres que vous pouvez rencontrer dans les programmes concurrents : la famine, les signaux manqués et les *livelocks* (les signaux manqués seront présentés dans la section 14.2.3).

10.3.1 Famine

La *famine* intervient lorsqu'un thread se voit constamment refuser l'accès à des ressources dont il a besoin pour continuer son traitement ; la famine la plus fréquente est celle qui concerne les cycles processeurs.

Dans les applications Java, la famine peut être due à une mauvaise utilisation des priorités des threads. Elle peut également être causée par l'exécution de structures de contrôles infinies (des boucles sans fin ou des attentes de ressources qui ne se terminent jamais) pendant la détention d'un verrou puisque les autres threads en attente de ce verrou ne pourront jamais le prendre.

Les priorités définies dans l'API Thread sont simplement des indications pour l'ordonnancement. L'API définit dix niveaux de priorités que la JVM, si elle le souhaite, peut faire correspondre aux priorités d'ordonnancement du système d'exploitation. Cette mise en correspondance étant spécifique à chaque plate-forme, deux priorités Java peuvent donc très bien correspondre à la même priorité du SE sur un système et à des priorités différentes sur un autre. Certains systèmes d'exploitation ont, en effet, moins de dix niveaux de priorité, auquel cas plusieurs priorités Java correspondent à la même priorité du SE.

Les ordonnanceurs des systèmes d'exploitation s'appliquent à fournir un ordonnancement bien plus évolué que celui qui est requis par la spécification du langage Java. Dans la plupart des applications Java, tous les threads ont la même priorité, `Thread.NORM_PRIORITY`. Le mécanisme de priorité des threads est un instrument tranchant et il n'est pas toujours évident de savoir quels seront les effets d'un changement de priorité ; augmenter celle d'un thread peut ne rien donner ou faire en sorte qu'un seul thread soit planifié de préférence à l'autre, d'où une famine de ce dernier.

Il est généralement conseillé de résister à la tentation de jouer avec les priorités des threads car, dès que l'on commence à les modifier, le comportement d'une application dépend de la plate-forme et l'on prend le risque d'une famine. La présence de `Thread.sleep` ou `Thread.yield` dans des endroits inhabituels est souvent un indice qu'un programme essaie de se sortir d'une manipulation des priorités et d'autres problèmes de réactivité en tentant de donner plus de temps aux threads moins prioritaires¹.

Évitez d'utiliser les priorités des threads car cela augmente la dépendance vis-à-vis de la plate-forme et peut poser des problèmes de vivacité. La plupart des applications concurrentes peuvent se contenter d'utiliser la priorité par défaut pour tous les threads.

1. Les sémantiques de `Thread.yield` (et `Thread.sleep(0)`) ne sont pas définies [JLS 17.9] ; la JVM est libre de les implémenter comme des opérations n'ayant aucun effet ou de les traiter comme des indices d'ordonnancement. Ces méthodes n'ont notamment pas nécessairement la même sémantique que celle de `sleep(0)` sur les systèmes Unix – qui place le thread courant à la fin de la file d'exécution pour cette priorité et donne le contrôle aux autres threads de même priorité – bien que certaines JVM implémentent `yield` de cette façon.

10.3.2 Faible réactivité

Si l'on recule d'un pas par rapport à la famine, on obtient la *faible réactivité*, qui est assez fréquente dans les applications graphiques qui utilisent des threads en arrière-plan. Au Chapitre 9, nous avons développé un framework pour décharger les tâches longues dans des threads en arrière-plan afin de ne pas figer l'interface utilisateur. Les tâches gourmandes en processeur peuvent malgré tout affecter la réactivité car elles sont en compétition avec le thread des événements pour l'accès au CPU. C'est l'un des cas où la modification des priorités des threads peut avoir un intérêt : lorsque des calculs en arrière-plan utilisent de façon intensive le processeur. Si le travail effectué par les autres threads sont vraiment des tâches en arrière-plan, baisser leur priorité peut rendre les tâches de premier plan plus réactives.

La faible réactivité peut également être due à une mauvaise gestion du verrouillage. Si un thread détient un verrou pendant longtemps (pendant qu'il parcourt une grosse collection en effectuant un traitement non négligeable sur chaque élément, par exemple), les autres threads ayant besoin d'accéder à cette collection devront attendre très longtemps.

10.3.3 Livelock

Les *livelock* sont une forme de perte de vivacité dans laquelle un thread non bloqué ne peut quand même pas continuer son traitement car il continue de retenter une opération qui échouera toujours. Ils interviennent souvent dans les applications de messagerie transactionnelle, où l'infrastructure de messagerie annule une transaction lorsqu'un message ne peut pas être traité correctement, pour le replacer en tête de la file. Si un bogue du gestionnaire de message provoque son échec, la transaction sera annulée à chaque fois que le message est sorti de la file et passé au gestionnaire bogué. Comme le message est remplacé en tête de file, le gestionnaire sera sans cesse appelé et produira le même résultat (ce type de situation est parfois appelé *problème du message empoisonné*). Le thread de gestion du message n'est pas bloqué mais ne progressera plus non plus. Cette forme de livelock provient souvent d'un code de récupération d'erreur trop zélé qui confond une erreur non récupérable avec une erreur récupérable.

Les livelock peuvent également intervenir lorsque plusieurs threads qui coopèrent changent leur état en réponse aux autres de sorte qu'aucun thread ne peut plus progresser. Cela ressemble à ce qui se passe lorsque deux personnes trop polies marchent l'une vers l'autre dans un couloir : chacune fait un pas de côté pour laisser passer l'autre et toutes les deux se retrouvent à nouveau en face l'une de l'autre. Elles font alors un autre écart et peuvent ainsi recommencer éternellement...

La solution à ce type de livelock consiste à introduire un peu de hasard dans le mécanisme de réessai. Si deux stations d'un réseau Ethernet essaient par exemple d'envoyer en même temps un paquet sur le support partagé, il y aura une collision. Les stations détectent cette collision et essaient de réémettre leurs paquets : si chacune réémet exactement une seconde plus tard, la collision recommencera sans fin et le paquet ne partira

jamais, même s'il y a suffisamment de bande passante. Pour éviter ce problème, on fait en sorte que chacune attende pendant un certain moment comprenant une partie aléatoire (le protocole Ethernet inclut également une attente exponentielle après les collisions répétées, ce qui réduit à la fois la congestion et le risque d'un nouvel échec avec les différentes stations impliquées). Selon le même principe, une attente aléatoire peut également permettre d'éviter les livelocks dans les applications concurrentes.

Résumé

Les pertes de vivacité sont un sérieux problème car il n'y a aucun moyen de s'en remettre, à part terminer l'application. La forme la plus fréquente est l'interblocage lié à l'ordre du verrouillage et il ne peut être résolu que lors de la conception : il faut s'assurer que les threads qui prennent plusieurs verrous le feront toujours dans le même ordre. Le meilleur moyen de le garantir consiste à utiliser des appels ouverts tout au long du programme car cela réduit beaucoup le nombre des emplacements où plusieurs verrous sont détenus simultanément et rend plus facile leur repérage.

Performances et adaptabilité

L'une des principales raisons d'utiliser les threads est l'amélioration des performances¹. Grâce à leur emploi, on améliorera l'utilisation des ressources en permettant aux applications d'exploiter plus facilement la capacité de traitement disponible et la réactivité en les autorisant à lancer immédiatement de nouvelles tâches pendant que d'autres s'exécutent encore.

Ce chapitre explore les techniques permettant d'analyser, de surveiller et d'améliorer les performances des programmes concurrents. Malheureusement, beaucoup de techniques d'optimisation des performances augmentent également la complexité et, par conséquent, les risques de problèmes de sécurité vis-à-vis des threads et les risques de perte de vivacité. Pire encore, certaines techniques destinées à améliorer les performances sont, en fait, contre-productives ou échangent un type de problème contre un autre. Bien qu'il soit souvent souhaitable d'optimiser les performances, la sécurité vis-à-vis des threads est toujours prépondérante. Il faut d'abord que le programme soit correct avant de l'accélérer – et uniquement s'il y en a besoin. Lorsque l'on conçoit une application concurrente, extraire la dernière goutte de performance est souvent le dernier de nos soucis.

11.1 Penser aux performances

Améliorer les performances consiste à en faire plus avec moins de ressources. La signification de "ressource" peut varier ; pour une activité donnée, certaines ressources spécifiques sont généralement trop peu nombreuses, que ce soient les cycles processeurs, la mémoire, la bande passante du réseau, celle des E/S, les requêtes aux bases de données, l'espace disque ou n'importe quelle autre quantité de ressource. Lorsque les performances

1. Certains pourraient rétorquer que c'est la *seule* que nous supportons avec la complexité que les threads introduisent.

d'une activité sont limitées par la disponibilité d'une ressource particulière, on dit qu'elle est *liée* à cette ressource : on parle alors d'activité liée au processeur, à une base de données, etc.

Bien que le but puisse être d'améliorer les performances globales, l'utilisation de plusieurs threads a toujours un coût par rapport à l'approche monothread. Ce prix à payer inclut le surcoût dû à la coordination entre les threads (verrouillage, signaux et synchronisation de la mémoire), aux changements de contexte supplémentaires, à la création et à la suppression des threads et, enfin, à leur planification. Lorsque les threads sont employés correctement, ces surcoûts s'effacent devant l'accroissement du débit des données, de la réactivité ou de la capacité de traitement. En revanche, une application concurrente mal conçue peut avoir des performances bien inférieures à celles d'un programme séquentiel¹.

En utilisant la concurrence pour accroître les performances, on essaie de réaliser deux buts : utiliser plus efficacement les ressources de traitement disponibles et permettre au programme d'exploiter les ressources de traitement supplémentaires lorsqu'elles sont mises à disposition. Du point de vue de la surveillance des performances, ceci signifie que nous tentons d'occuper le plus possible le processeur (ce qui ne signifie pas, bien sûr, qu'il faille consommer des cycles avec des calculs sans intérêt ; nous voulons l'occuper avec du travail *utile*). Si le programme est lié au processeur, nous devrions donc pouvoir augmenter sa capacité de traitement en ajoutant plus de processeurs, mais, s'il ne peut même pas occuper les processeurs disponibles, en ajouter d'autres ne lui servira à rien. Les threads offrent un moyen de "faire chauffer" les CPU en décomposant l'application de sorte qu'un processeur disponible ait toujours du travail à faire.

11.1.1 Performances et adaptabilité

Les performances d'une application peuvent se mesurer de différentes façons : en tant que temps de service, de latence, de débit, d'efficacité, d'adaptabilité ou de capacité. Certaines (le temps de service et la latence) mesurent la "rapidité" à laquelle une unité de travail peut être traitée ou prise en compte : d'autres (la capacité et le débit) évaluent la "quantité" de travail qui peut être effectuée avec un nombre donné de ressources de calcul.

L'adaptabilité décrit la faculté d'améliorer le débit ou la capacité lorsque l'on ajoute de nouvelles ressources (processeurs, mémoire, disques ou bande passante).

1. Un collègue m'a relaté cette anecdote amusante : il avait été chargé du test d'une application chère et complexe qui effectuait ses traitements *via* un pool de threads configurable. Lorsque le système a été terminé, les tests ont montré que le nombre optimal de threads pour le pool était... 1. Cela aurait dû être évident dès le début puisque le système cible était monoprocesseur et que l'application était presque entièrement liée au processeur.

La conception et la configuration d'applications concurrentes adaptables peuvent être très différentes d'une optimisation classique des performances. En effet, améliorer les performances consiste généralement à faire en sorte d'effectuer le *même* traitement avec *moins* d'effort, en réutilisant par exemple des résultats mis en cache ou en remplaçant un algorithme en $O(n^2)$ ¹ par un autre en $O(n \log n)$. Pour l'adaptabilité, en revanche, on recherche des moyens de paralléliser le problème afin de tirer parti des ressources supplémentaires : le but consiste donc à en faire *plus* avec *plus* de ressources.

Ces deux aspects des performances – *rapidité* et *quantité* – sont totalement disjoints et, parfois, opposés l'un à l'autre. Pour obtenir une meilleure adaptabilité ou une utilisation optimale du matériel, on finit souvent par *augmenter* le volume de travail à faire pour traiter chaque tâche, en divisant par exemple les tâches en plusieurs sous-tâches individuelles "en pipeline". Ironiquement, la plupart des astuces qui améliorent les performances des programmes monothreads sont mauvaises pour l'adaptabilité (la section 11.4.4 présentera un exemple).

Le modèle trois-tier classique – dans lequel la couche présentation, la couche métier et la couche accès aux données sont séparées et peuvent être prises en charge par des systèmes différents – illustre bien comment des améliorations en termes d'adaptabilité se font souvent aux dépens des performances. Une application monolithique dans laquelle ces trois couches sont mélangées aura presque toujours de meilleures performances qu'une autre multi-tier, bien conçue, distribuée sur plusieurs systèmes. Comment pourrait-il en être autrement ? L'application monolithique ne souffre pas de la latence du réseau inhérente au passage des tâches entre les couches et n'a pas à payer le prix de la séparation d'un traitement en plusieurs couches abstraites (notamment les surcoûts dus à la mise en file d'attente, à la coordination et à la copie des données).

Cependant, il se posera un sérieux problème lorsque ce système monolithique aura atteint sa capacité de traitement car il peut être extrêmement difficile de l'augmenter de façon significative. On accepte donc généralement de payer le prix d'un plus long temps de service ou d'utiliser plus de ressources par unité de travail afin que notre application puisse s'adapter à une charge plus lourde si on lui ajoute plus de ressources.

Des différents aspects des performances, ceux concernant la quantité – l'adaptabilité, le débit et la capacité – sont généralement plus importants pour les applications serveur que ceux liés à la rapidité (pour les applications interactives, la latence est le critère le plus important car les utilisateurs n'ont pas besoin d'indicateurs de progression ou de se demander ce qui se passe). Dans ce chapitre, nous nous intéresserons essentiellement à l'adaptabilité plutôt qu'aux performances pures dans un contexte monothread.

1. Une complexité en $O(n^2)$ signifie que les performances de l'algorithme sont proportionnelles au carré du nombre des éléments.

11.1.2 Compromis sur l'évaluation des performances

Quasiment toutes les décisions techniques impliquent un compromis. Utiliser de l'acier plus épais pour un pont, par exemple, peut augmenter sa capacité et sa sécurité, mais également son coût de construction. Si les décisions informatiques n'impliquent généralement pas de compromis entre le prix et les risques de vies humaines, on dispose également de moins d'informations permettant de faire les bons choix. L'algorithme de "tri rapide", par exemple, est très efficace sur les grands volumes de données alors que le "tri à bulles", moins sophistiqué, est en fait plus performant que lui sur les petits ensembles. Si l'on vous demande d'implémenter une fonction de tri efficace, vous devez donc avoir une idée du volume de données que vous devrez traiter et savoir si vous voulez optimiser le cas moyen, le pire des cas ou la prédictibilité. Malheureusement, ces informations ne font souvent pas partie du cahier des charges remis à l'auteur d'une fonction de tri, et c'est l'une des raisons pour lesquelles la plupart des optimisations sont prématurées : *elles sont souvent entreprises avant d'avoir une liste claire des conditions à remplir.*

Évitez les optimisations prématurées. Commencez par faire correctement les choses, puis accélérez-les – si cela ne va pas assez vite.

Lorsque l'on prend des décisions techniques, on doit parfois troquer une forme de coût contre une autre (le temps de service contre la consommation mémoire, par exemple) ; parfois, on échange le coût contre la sécurité. Cette sécurité ne signifie pas nécessairement que des vies humaines sont en jeu comme dans l'exemple du pont. De nombreuses optimisations des performances se font au détriment de la lisibilité ou de la facilité de maintenance – plus le code est "astucieux" ou non évident, plus il est difficile à comprendre et à maintenir. Les optimisations supposent parfois de remettre en cause les bons principes de conception orientée objet, de casser l'encapsulation, par exemple ; elles augmentent quelquefois les risques d'erreur car les algorithmes plus rapides sont généralement plus compliqués (si vous ne pouvez pas identifier les coûts ou les risques, vous n'y avez sûrement pas assez réfléchi pour continuer).

La plupart des décisions liées aux performances impliquent plusieurs variables et dépendent fortement de la situation. Avant de décider qu'une approche est "plus rapide" qu'une autre, vous devez vous poser les questions suivantes :

- Que signifie "plus rapide" ?
- Sous quelles conditions cette approche sera vraiment plus rapide ? Sous une charge légère ou lourde ? Avec de petits ou de gros volumes de données ? Pouvez-vous étayer votre réponse par des mesures ?
- Combien de fois ces conditions risquent-elles d'avoir lieu dans votre situation ? Pouvez-vous étayer votre réponse par des mesures ?

- Est-ce que le code sera utilisé dans d'autres situations, où les conditions peuvent être différentes ?
- Quels coûts cachés, comme l'augmentation des risques de développement ou de maintenance, échangez-vous contre cette amélioration des performances ? Le jeu en vaut-il la chandelle ?

Toutes ces considérations s'appliquent à n'importe quelle décision technique liée aux performances, mais il s'agit ici d'un livre sur la programmation concurrente : pourquoi recommandons-nous une telle prudence vis-à-vis de l'optimisation ? Parce que *la quête de performance est sûrement l'unique source majeure des bogues de concurrence*. La croyance que la synchronisation était "trop lente" a produit des idiom es apparemment astucieux, mais dangereux, pour la réduire (comme le verrouillage contrôlé deux fois, que nous présenterons dans la section 16.2.4) et elle est souvent citée comme excuse pour ne pas respecter les règles concernant la synchronisation. Les bogues de concurrence faisant partie des plus difficiles à repérer et à supprimer, tout ce qui risque de les introduire doit être entrepris avec beaucoup de précautions.

Pire encore, lorsque vous échangez la sécurité contre les performances, vous pouvez n'obtenir aucune des deux. Lorsqu'il s'agit de concurrence, l'intuition de nombreux développeurs concernant l'emplacement d'un problème de performances ou l'approche qui sera plus rapide ou plus adaptable est souvent incorrecte. Il est donc impératif que toute tentative d'amélioration des performances soit accompagnée d'exigences concrètes (afin de savoir quand tenter d'améliorer et quand arrêter) et d'un programme de mesures utilisant une configuration réaliste et un profil de charge. Mesurez après la modification pour vérifier que vous avez obtenu l'amélioration escomptée. Les risques de sécurité et de maintenance associés à de nombreuses optimisations sont suffisamment graves – vous ne souhaitez pas payer ce prix si vous n'en n'avez pas besoin et vous ne voudrez vraiment pas le payer si vous n'obtenez aucun bénéfice.

Mesurez, ne supposez pas.

Bien qu'il existe des outils de profilage sophistiqués permettant de mesurer les performances et de repérer les goulets d'étranglement qui les pénalisent, vous n'avez pas besoin de dépenser beaucoup d'argent pour savoir ce que fait votre programme. L'application gratuite perfbar, par exemple, peut vous donner une bonne idée de l'occupation du processeur : votre but étant généralement de l'occuper au maximum, c'est un très bon moyen d'évaluer le besoin d'ajuster les performances et l'effet de cet ajustement.

11.2 La loi d'Amdahl

Certains problèmes peuvent se résoudre plus rapidement lorsqu'on dispose de plus de ressources – plus il y a d'ouvriers pour faire les récoltes, plus elles sont terminées rapidement. D'autres tâches sont fondamentalement séquentielles – ce n'est pas un nombre plus grand d'ouvriers qui feront pousser plus vite la récolte. Si l'une des raisons principales d'utiliser les threads est d'exploiter la puissance de plusieurs processeurs, il faut également s'assurer que le problème peut être parallélisé et que le programme exploite effectivement ce parallélisme.

La plupart des programmes concurrents ont beaucoup de points communs avec l'agriculture, qui est un mélange de parties parallélisables et séquentielles. La *loi d'Amdahl* précise l'amélioration théorique de la vitesse d'un programme par rapport au nombre de ressources supplémentaires d'après la proportion de ses composantes parallélisables et séquentielles. Si F est la fraction de calcul qui doit être exécutée séquentiellement, cette loi indique que, sur une machine à N processeurs, on peut obtenir un gain de vitesse d'au plus :

$$\text{gain de vitesse} \leq \frac{1}{F + \frac{(1 - F)}{N}}$$

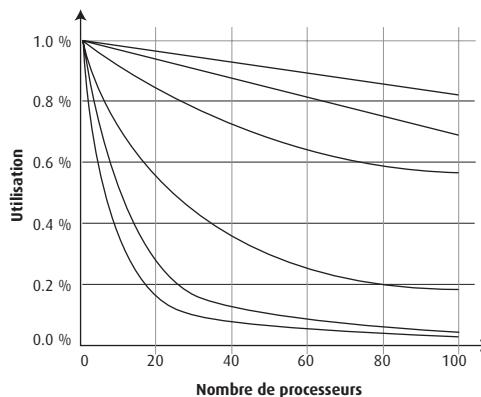
Lorsque N tend vers l'infini, le gain de vitesse maximal converge vers $1/F$, ce qui signifie qu'un programme dont cinquante pour-cent des calculs doivent s'effectuer en séquence ne peut être accéléré que par un facteur de deux, quel que soit le nombre de processeurs disponibles ; un programme où dix pour-cent doit être exécuté en série peut être accéléré par, au plus, un facteur de dix. La loi d'Amdahl quantifie également le coût de l'efficacité de la sérialisation. Avec dix processeurs, un programme séquentiel à 10 % peut espérer au plus une accélération de 5,3 (à 53 % d'utilisation) ; avec cent processeurs, il peut obtenir un gain maximal de vitesse de 9,2 (à 9 % d'utilisation). Il faut beaucoup de processeurs mal utilisés pour ne jamais obtenir ce facteur de 10.

La Figure 11.1 montre l'utilisation maximale des processeurs pour différents degrés d'exécution séquentielle et différents nombres de processeurs (l'utilisation est définie comme le gain de vitesse divisé par le nombre de processeurs). Il apparaît clairement que, à mesure que le nombre de CPU augmente, même un petit pourcentage d'exécution séquentielle limite l'amélioration possible du débit lorsqu'on ajoute des ressources supplémentaires.

Le Chapitre 6 a montré comme identifier les frontières logiques qui permettent de décomposer les applications en tâches. Cependant, il faut également identifier les sources de sérialisation dans les tâches afin de prévoir le gain qu'il est possible d'obtenir en exécutant une application sur un système multiprocesseurs.

Figure 11.1

Utilisation maximale selon la loi d'Amdahl pour différents pourcentages de sérialisation.



Imaginons une application où N threads exécutent la méthode `doWork()` du Listing 11.1, récupèrent les tâches dans une file d'attente partagée et les traitent en supposant que les tâches ne dépendent pas des résultats ou des effets de bords des autres tâches. Si l'on ne tient pas compte pour le moment de la façon dont les tâches sont placées dans la file, est-ce que cette application s'adaptera bien lorsque l'on ajoutera des processeurs ? Au premier abord, il peut sembler qu'elle est totalement parallélisable : les tâches ne s'attendent pas les unes et les autres et, plus il y aura de processeurs, plus l'application pourra traiter de tâches en parallèle. Cependant, l'application comprend également une composante séquentielle – la récupération d'une tâche dans la file d'attente. Cette dernière est partagée par tous les threads et nécessite donc un peu de synchronisation pour maintenir son intégrité face aux accès concurrents. Si l'on utilise un verrou pour protéger l'état de cette file, un thread voulant prendre une tâche dans la file forcera les autres qui veulent prendre la leur à attendre – et c'est là que le traitement des tâches est sérialisé.

Listing 11.1 : Accès séquentiel à une file d'attente.

```
public class WorkerThread extends Thread {
    private final BlockingQueue <Runnable> queue;

    public WorkerThread(BlockingQueue <Runnable> queue) {
        this.queue = queue;
    }

    public void run() {
        while (true) {
            try {
                Runnable task = queue.take();
                task.run();
            } catch (InterruptedException e) {
                break; /* Permet au thread de se terminer */
            }
        }
    }
}
```

Le temps de traitement d'une seule tâche inclut non seulement le temps d'exécution de la tâche `Runnable`, mais également celui nécessaire à l'extraction de la tâche à partir de la file d'attente partagée. Si cette file est une `LinkedBlockingQueue`, l'opération d'extraction peut être moins bloquante qu'avec une `LinkedList` synchronisée car `LinkedBlockingQueue` utilise un algorithme mieux adapté ; cependant, l'accès à toute structure de données partagée introduit fondamentalement un élément de sérialisation dans un programme.

Cet exemple ignore également une autre source de sérialisation : la gestion du résultat. Tous les calculs utiles produisent un résultat ou un effet de bord – dans le cas contraire, ils pourraient être supprimés puisqu'il s'agirait de code mort. `Runnable` ne fournissant aucun traitement explicite du résultat, ces tâches doivent donc avoir un certain effet de bord, en écrivant par exemple leur résultat dans un fichier journal ou en le plaçant dans une structure de données. Les fichiers journaux et les conteneurs de résultats étant généralement partagés par plusieurs threads, ils constituent donc également une source de sérialisation. Si chaque thread gérait sa propre structure de résultat et que toutes ces structures étaient fusionnées à la fin des tâches, cette fusion finale serait aussi une source de sérialisation.

Toutes les applications concurrentes comprennent des sources de sérialisation. Si vous pensez que la vôtre n'en a pas, regardez mieux.

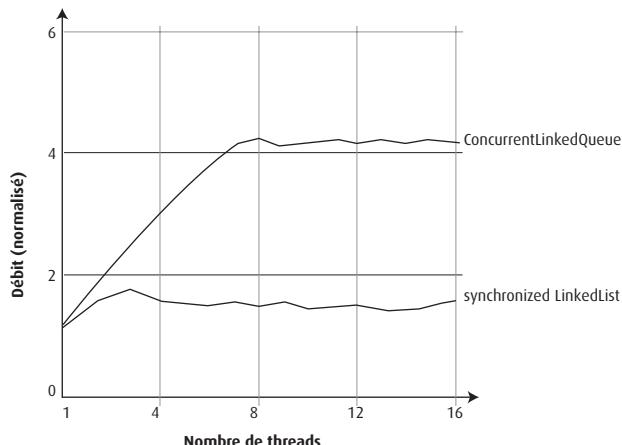
11.2.1 Exemple : sérialisation cachée dans les frameworks

Pour voir comment la sérialisation peut se cacher dans la structure d'une application, nous pouvons comparer les débits à mesure que l'on ajoute des threads et déduire les différences en termes de sérialisation d'après les différences observées dans l'adaptabilité. La Figure 11.2 montre une application simple dans laquelle plusieurs threads répètent la suppression d'un élément d'une Queue partagée et le traitent, un peu comme dans le Listing 11.1. L'étape de traitement n'est qu'un calcul local au thread. Si un thread trouve la file vide, il y ajoute un lot de nouveaux éléments pour que les autres aient quelque chose à traiter lors de leur prochaine itération. L'accès à la file partagée implique évidemment un peu de sérialisation, mais l'étape de traitement est entièrement parallélisable puisqu'elle n'utilise aucune donnée partagée.

Les courbes de la Figure 11.2 comparent les débuts des deux implémentations thread-safe de `Queue` : une `LinkedList` enveloppée avec `synchronizedList()` et une `ConcurrentLinkedQueue`. Les tests ont été effectués sur une machine Sparc V880 à huit processeurs avec le système Solaris. Bien que chaque exécution représente la même somme de "travail", nous pouvons constater que le simple fait de changer d'implémentation de file peut avoir de lourdes conséquences sur l'adaptabilité.

Figure 11.2

Comparaison des implémentations de files d'attente.



Le débit de `ConcurrentLinkedQueue` continue en effet d'augmenter jusqu'à ce qu'il atteigne le nombre de processeurs, puis reste à peu près constant. Le débit de la `LinkedList` synchronisée, en revanche, montre une amélioration jusqu'à trois threads, puis chute à mesure que le coût de la synchronisation augmente. Au moment où elle passe à quatre ou cinq threads, la compétition est si lourde que chaque accès au verrou de la file est très disputé et que le débit est dominé par les changements de contexte. La différence de débit provient des degrés de sérialisation différents entre les deux implémentations. La `LinkedList` synchronisée protège l'état de toute la file avec un seul verrou qui est détenu pendant la durée de l'ajout ou de la suppression ; `ConcurrentLinkedQueue`, en revanche, utilise un algorithme non bloquant sophistiqué (voir la section 15.4.2) qui se sert de références atomiques pour mettre à jour les différents pointeurs de liens. Dans la première, toute l'insertion ou toute la suppression est sérialisée ; dans la seconde, seules les modifications des différents pointeurs sont traitées séquentiellement.

11.2.2 Application qualitative de la loi d'Amdahl

La loi d'Amdahl quantifie le gain de vitesse possible lorsque l'on dispose de ressources de traitement supplémentaires – à condition de pouvoir estimer précisément la fraction séquentielle de l'exécution. Bien qu'il puisse être difficile d'évaluer directement cette sérialisation, la loi d'Amdahl peut quand même être utile sans cette mesure.

Nos modèles mentaux étant influencés par notre environnement, nombre d'entre nous pensons qu'un système multiprocesseurs est doté de deux ou quatre processeurs ou, en cas de gros budget, de plusieurs dizaines puisque c'est cette technologie qui nous est proposée un peu partout depuis quelques années. Mais, lorsque les processeurs multicœurs se seront répandus, les systèmes auront des centaines, voire des milliers de processeurs¹. Les

1. Mise à jour : à l'heure où ce livre est écrit, Sun vend des serveurs d'entrée de gamme reposant sur le processeur Niagara à 8 cœurs et Azul vend des serveurs haut de gamme (96, 192 et 384 processeurs) exploitant le processeur Vega à 24 cœurs.

algorithmes qui semblent adaptables sur un système à quatre processeurs peuvent donc avoir des goulets d'étranglement pour leur adaptabilité qui n'ont tout simplement pas encore été rencontrés.

Lorsqu'on évalue un algorithme, penser à ce qui se passera avec des centaines ou des milliers de processeurs permet d'avoir une idée des limites d'adaptabilité qui peuvent apparaître. Les sections 11.4.2 et 11.4.3, par exemple, présentent deux techniques pour réduire la granularité du verrouillage : la division des verrous (diviser un verrou en deux) et le découpage des verrous (diviser un verrou en plusieurs). En les étudiant à la lumière de la loi d'Amdahl, on s'aperçoit que la division d'un verrou en deux ne nous amène pas bien loin dans l'exploitation de nombreux processeurs, alors que le découpage de verrous semble plus prometteur puisque le nombre de découpages peut être augmenté en fonction du nombre de processeurs (les optimisations des performances devraient, bien sûr, toujours être considérées à la lumière des exigences de performances réelles ; dans certains cas, diviser un verrou en deux peut donc suffire).

11.3 Coûts liés aux threads

Les programmes monothreads n'impliquent pas de surcoûts dus à la planification ou à la synchronisation et n'utilisent pas de verrous pour préserver la cohérence des structures de données. La planification et la coordination entre les threads ont un coût en termes de performances : pour que les threads améliorent la rapidité d'un programme, il faut donc que les bénéfices de la parallélisation soient supérieurs aux coûts induits par la concurrence.

11.3.1 Changements de contexte

Si le thread principal est le seul thread, il ne sera quasiment jamais déprogrammé. Si, en revanche, il y a plus de threads que de processeurs, le système d'exploitation finira par préempter un thread pour permettre à un autre d'utiliser à son tour le processeur. Cela provoque donc un *changement de contexte* qui nécessite de sauvegarder le contexte du thread courant et de restaurer celui du thread nouvellement planifié.

Les changements de contexte ne sont pas gratuits car la planification implique une manipulation de structures de données partagées dans le SE et dans la JVM. Tous les deux utilisant les mêmes processeurs que le programme, plus il y a de temps CPU passé dans le code du SE et de la JVM, moins il en reste pour le programme. Cependant, les activités du SE et de la JVM ne sont pas les seuls coûts induits par un changement de contexte. Lorsqu'un thread est planifié, les données dont il a besoin ne se trouvent sûrement pas dans le cache local du processeur : le changement de contexte produira donc une rafale d'erreurs de cache et le thread s'exécutera alors un peu plus lentement au début. C'est l'une des raisons pour lesquelles les planificateurs octroient à chaque thread qui s'exécute un quantum de temps minimal, même si de nombreux autres threads sont en attente : cela permet d'amortir le coût du changement de contexte en le diluant dans un

temps d'exécution plus long et non interrompu, ce qui améliore le débit (aux dépens de la réactivité).

Lorsqu'un thread se bloque parce qu'il attend de prendre un verrou disputé par d'autres threads, la JVM le suspend afin de pouvoir le déprogrammer. Si des threads se bloquent souvent, ils ne pourront donc pas utiliser tout le quantum de temps qui leur est alloué. Un programme qui se bloque plus souvent (à cause d'E/S bloquantes, d'attentes de verrous très disputés ou de variables conditions) implique par conséquent plus de changements de contexte qu'un programme lié au processeur, ce qui augmente le surcoût de la planification et réduit le débit (les algorithmes non bloquants peuvent permettre de réduire ces changements de contexte ; voir Chapitre 15).

Le véritable coût d'un changement de contexte varie en fonction des plates-formes, mais une bonne estimation consiste à considérer qu'il est équivalent à 5 000-10 000 cycles d'horloge, soit plusieurs microsecondes sur la plupart des processeurs actuels.

La commande `vmstat` des systèmes Unix et l'outil `perfmon` de Windows permettent d'évaluer le nombre de changements de contexte et le pourcentage de temps passé dans le noyau. Une forte utilisation du noyau (plus de 10 %) indique souvent une forte activité de planification, qui peut être due à des E/S bloquantes ou à une lutte pour l'obtention de verrous.

11.3.2 Synchronisation de la mémoire

Le coût de la synchronisation en termes de performances a plusieurs sources. Les garanties de visibilité offertes par `synchronized` et `volatile` peuvent impliquer des instructions spéciales appelées *barrières mémoires* qui peuvent vider ou invalider des caches, vider les tampons physiques d'écriture et figer les pipelines d'exécution. Les barrières mémoires peuvent également avoir des conséquences indirectes sur les performances car elles inhibent certaines optimisations du compilateur ; elles l'empêchent de réordonner la plupart des opérations.

Lorsque l'on veut estimer l'impact de la synchronisation sur les performances, il est important de faire la distinction entre synchronisation *avec compétition* et *sans compétition*. Le mécanisme `synchronized` est optimisé pour le cas sans compétition (`volatile` est toujours sans compétition) et, au moment où ce livre est écrit, le coût d'une synchronisation "rapide" sans compétition varie de 20 à 250 cycles d'horloge sur la plupart des systèmes. Bien qu'il ne soit certainement pas nul, l'effet d'une synchronisation sans compétition est donc rarement significatif dans les performances globales d'une application ; son alternative implique de compromettre la sécurité et de prendre le risque de devoir rechercher des bogues par la suite, ce qui est une opération très pénible pour vous ou vos successeurs.

Les JVM actuelles peuvent réduire le coût de la synchronisation annexe en optimisant le verrouillage quand il est certain qu'il sera sans compétition. Si un objet verrou n'est

accessible que par le thread courant, la JVM peut optimiser sa prise puisqu'il n'est pas possible qu'un autre thread puisse se synchroniser dessus. L'acquisition du verrou dans le Listing 11.2, par exemple, peut donc toujours être éliminée par la JVM.

Listing 11.2 : Synchronisation inutile. Ne le faites pas.

```
synchronized (new Object()) {
    // faire quelque chose
}
```



Les JVM plus sophistiquées peuvent *analyser les échappements* pour déceler quand une référence d'objet local n'est jamais publiée sur le tas et est donc locale au thread. Dans la méthode `getStoogeNames()` du Listing 11.3, par exemple, la seule référence à l'objet `List` est la variable locale `stooges` et les variables confinées à la pile sont automatiquement locales au thread. Une exécution naïve de `getStoogeNames()` prendrait et relâcherait quatre fois le verrou sur le `Vector`, un pour chaque appel à `add()` ou `toString()`. Cependant, un compilateur un peu malin peut traduire ces appels en ligne, constater que `stooges` et son état interne ne s'échapperont jamais et donc éliminer les quatre prises du verrou¹.

Listing 11.3 : Candidat à l'élosion de verrou.

```
public String getStoogeNames() {
    List<String> stooges = new Vector<String>();
    stooges.add("Moe");
    stooges.add("Larry");
    stooges.add("Curly");
    return stooges.toString();
}
```

Même sans analyse des échappements, les compilateurs peuvent également effectuer *un épaisissement de verrou*, c'est-à-dire fusionner les blocs synchronisés adjacents qui utilisent le même verrou. Dans le cas de `getStoogeNames()`, une JVM effectuant un épaisissement de verrou pourrait combiner les trois appels à `add()` et l'appel à `toString()` en une prise et un relâchement d'un seul verrou en se servant d'heuristiques sur le coût relatif de la synchronisation par rapport aux instructions dans le bloc `synchronized`². Non seulement cela réduit le surcoût de la synchronisation, mais cela fournit également au compilateur un plus gros bloc à traiter, ce qui permet d'activer d'autres optimisations.

1. Cette optimisation, appelée *élosion de verrou*, est effectuée par la JVM IBM et est attendue dans Hostpot de Java 7.

2. Un compilateur dynamique astucieux pourrait se rendre compte que cette méthode renvoie toujours la même chaîne et recompiler `getStoogeNames()` après sa première exécution pour qu'elle renvoie simplement la valeur renvoyée par son premier appel.

Ne vous faites pas trop de souci sur le coût de la synchronisation sans compétition. Le mécanisme de base est déjà assez rapide et les JVM peuvent encore effectuer des optimisations supplémentaires pour réduire ou éliminer ce coût. Intéressez-vous plutôt à l'optimisation des parties comprenant des verrous avec compétition.

La synchronisation par un seul thread peut également affecter les performances des autres threads car elle crée du trafic sur le bus de la mémoire partagée, or ce bus a une bande passante limitée et il est partagé par tous les processeurs. Si les threads doivent se battre pour la bande passante de la synchronisation, ils en souffriront tous¹.

11.3.3 Blocages

La synchronisation sans compétition peut être entièrement gérée dans la JVM (Bacon et al., 1998) alors que la synchronisation avec compétition peut demander une activité du système d'exploitation, qui a également un coût. Lorsque le verrouillage est disputé, le ou les threads qui perdent doivent se bloquer. La JVM peut implémenter le blocage par une *attente tournante* (elle réessaie d'obtenir le verrou jusqu'à ce que cela réussisse) ou en suspendant le thread bloqué via le système d'exploitation. La méthode la plus efficace dépend de la relation entre le coût du changement de contexte et le temps qu'il faut attendre avant que le verrou soit disponible ; l'attente tournante est préférable pour les attentes courtes alors que la suspension est plus adaptée aux longues. Certaines JVM choisissent entre les deux en utilisant des informations sur des temps d'attente passés, mais la plupart se contentent de suspendre les threads qui attendent un verrou.

La suspension d'un thread parce qu'il ne peut pas obtenir un verrou ou parce qu'il est bloqué sur une attente de condition ou dans une opération d'E/S bloquante implique deux changements de contexte supplémentaires, avec toute l'activité du SE et du cache qui leur est associée : le thread bloqué est sorti de l'unité centrale avant la fin de son quantum de temps et il y est remis plus tard, lorsque le verrou ou une autre ressource devient disponible (le blocage dû à la compétition pour un verrou a également un coût pour le thread qui détient ce verrou : quand il le relâche, il doit en plus demander au SE de relancer le thread bloqué).

11.4 Réduction de la compétition pour les verrous

Nous avons vu que la sérialisation détériorait l'adaptabilité et que les changements de contexte pénalisaient les performances. Le verrouillage avec compétition souffrant de ces deux problèmes, le réduire permet d'améliorer à la fois les performances et l'adaptabilité.

1. Cet aspect est parfois utilisé comme argument contre l'utilisation des algorithmes non bloquants sous une forme ou sous une autre de repli car, en cas de compétition importante, ces algorithmes produisent plus de trafic de synchronisation que ceux qui utilisent des verrous. Voir Chapitre 15.

L'accès aux ressources protégées par un verrou exclusif est sérialisé – un seul thread peut y accéder à la fois. Nous utilisons bien sûr les verrous pour de bonnes raisons, comme empêcher la corruption des données, mais cette sécurité a un prix. Une compétition persistante pour un verrou limite l'adaptabilité.

La principale menace contre l'adaptabilité des applications concurrentes est le verrou exclusif sur une ressource.

La compétition pour un verrou est influencée par deux facteurs : la fréquence à laquelle ce verrou est demandé et le temps pendant lequel il est conservé une fois qu'il a été pris¹. Si le produit de ces facteurs est suffisamment petit, la plupart des tentatives d'acquisition se feront sans compétition et la lutte pour ce verrou ne posera pas de problème significatif pour l'adaptabilité. Si, en revanche, le verrou est suffisamment disputé, les threads qui tentent de l'acquérir seront bloqués ; dans le pire des cas, les processeurs resteront inactifs bien qu'il y ait beaucoup de travail à faire.

Il y a trois moyens de réduire la compétition pour les threads :

- réduire la durée de possession des verrous ;
- réduire la fréquence des demandes de verrous ;
- remplacer les verrous exclusifs par des mécanismes de coordination permettant une plus grande concurrence.

11.4.1 Réduction de la portée des verrous ("entrer, sortir")

Un moyen efficace de réduire la probabilité de compétition consiste à maintenir les verrous le plus brièvement possible, ce qui peut être réalisé en déplaçant le code qui ne nécessite pas de verrouillage en dehors des blocs `synchronized` – surtout lorsqu'il s'agit d'opérations coûteuses et potentiellement bloquantes, comme des E/S.

Il est assez facile de constater que détenir trop longtemps un verrou "chaud" peut limiter l'adaptabilité ; nous en avons vu un exemple avec la classe `SynchronizedFactorizer` du Chapitre 2. Si une opération détient un verrou pendant 2 millisecondes et que chaque opération ait besoin de ce verrou, le débit ne pourra pas être supérieur à 500 opérations par seconde, quel que soit le nombre de processeurs disponibles. Réduire le temps de

1. C'est un corollaire de la *loi de Little*, un résultat de la théorie des files qui énonce que "le nombre moyen de clients dans un système stable est égal à leur fréquence moyenne d'arrivée multipliée par leur temps moyen dans le système" (Little, 1961).

détention du verrou à 1 milliseconde repousse la limite du débit à 1 000 opérations par seconde¹.

La classe `AttributeStore` du Listing 11.4 montre un exemple de détention d'un verrou plus longtemps que nécessaire. La méthode `userLocationMatches()` recherche l'emplacement de l'utilisateur dans un `Map` en utilisant des expressions régulières. Toute la méthode est synchronisée alors que la seule portion du code qui a réellement besoin du verrou est l'appel à `Map.get()`.

Listing 11.4 : Détection d'un verrou plus longtemps que nécessaire.

```
@ThreadSafe
public class AttributeStore {
    @GuardedBy("this") private final Map<String, String>
        attributes = new HashMap<String, String>();

    public synchronized boolean userLocationMatches(String name,
                                                   String regexp) {
        String key = "users." + name + ".location";
        String location = attributes.get(key);
        if (location == null)
            return false;
        else
            return Pattern.matches(regexp, location);
    }
}
```



La classe `BetterAttributeStore` du Listing 11.5 réécrit `AttributeStore` en réduisant de façon significative la durée de détention du verrou. La première étape consiste à construire la clé du `Map` associée à l'emplacement de l'utilisateur – une chaîne de la forme `users.nom.location` –, ce qui implique d'instancier un objet `StringBuilder`, de lui ajouter plusieurs chaînes et de récupérer le résultat sous la forme d'un objet `String`. Puis l'expression régulière est mise en correspondance avec cette chaîne. La construction de cette clé et le traitement de l'expression régulière n'utilisant pas l'état partagé, ces opérations n'ont pas besoin d'être exécutées pendant la détention du verrou : `BetterAttributeStore` les regroupe par conséquent à l'extérieur du bloc `synchronized` afin de réduire la durée du verrouillage.

Listing 11.5 : Réduction de la durée du verrouillage.

```
@ThreadSafe
public class BetterAttributeStore {
    @GuardedBy("this") private final Map<String, String>
        attributes = new HashMap<String, String>();

    public boolean userLocationMatches(String name, String regexp) {
        String key = "users." + name + ".location";
        String location;
```

1. En réalité, ce calcul *sous-estime* le coût d'une longue détention des verrous car il ne prend pas en compte le surcoût des changements de contexte produits par la compétition accrue pour l'obtention du verrou.

Listing 11.5 : Réduction de la durée du verrouillage. (suite)

```

synchronized (this) {
    location = attributes.get(key);
}
if (location == null)
    return false;
else
    return Pattern.matches(regexp, location);
}
}

```

Réduire la portée du verrou dans `userLocationMatches()` diminue de façon non négligeable le nombre d'instructions qui sont exécutées pendant que le verrou est pris. Selon la loi d'Amdahl, cela supprime un obstacle à l'adaptabilité car le volume de code sérialisé est moins important.

`AttributeStore` n'ayant qu'une seule variable d'état, `attributes`, nous pouvons encore améliorer la technique en *délégant la thread safety* (voir la section 4.3). En remplaçant `attributes` par une Map thread-safe (`Hashtable`, `synchronizedMap` ou `ConcurrentHashMap`), `AttributeStore` peut déléguer toutes ses obligations de sécurité vis-à-vis des threads à la collection thread-safe sous-jacente, ce qui élimine la nécessité d'une synchronisation explicite dans `AttributeStore`, réduit la portée du verrou à la durée de l'accès à l'objet Map et supprime le risque qu'un futur développeur sape la thread safety en oubliant de prendre le bon verrou avant d'accéder à `attributes`.

Bien que la réduction des blocs `synchronized` permette d'améliorer l'adaptabilité, un bloc `synchronized` peut être trop petit – il faut que les opérations qui doivent être atomiques (comme la modification de plusieurs variables participant à un invariant) soient contenues dans un même bloc `synchronized`. Par ailleurs, le coût de la synchronisation n'étant pas nul, découper un bloc `synchronized` en plusieurs peut, à un moment donné, devenir contre-productif en termes de performances¹. L'équilibre idéal dépend, bien sûr, de la plate-forme mais, en pratique, il est raisonnable de ne se soucier de la taille d'un bloc `synchronized` que lorsque l'on peut déplacer d'importants calculs ou des blocs d'opérations en dehors de ce bloc.

11.4.2 Réduire la granularité du verrouillage

L'autre moyen de réduire la durée d'un verrouillage (et donc la probabilité de la compétition pour ce verrou) est de faire en sorte que les threads le demandent moins souvent. Pour cela, on peut utiliser la *division* ou le *découpage* des verrous, qui permettent d'utiliser des verrous distincts pour protéger les différentes variables auparavant protégées par un verrou unique. Ces techniques aident à réduire la granularité du verrouillage et autorisent une meilleure adaptabilité – cependant, utiliser plus de verrous augmente également le risque d'interblocages.

1. Si la JVM effectue un épaissement de verrou, elle peut de toute façon annuler la division des blocs `synchronized`.

À titre d'expérience, imaginons ce qui se passerait s'il n'y avait qu'un seul verrou pour toute l'application au lieu d'un verrou distinct pour chaque objet. L'exécution de tous les blocs `synchronized`, quels que soient leurs verrous, s'effectuerait en série. Si de nombreux threads se disputent le verrou global, le risque que deux threads veuillent prendre ce verrou en même temps augmente, ce qui accroît la compétition. Si les demandes de verrouillage étaient partagées sur un *plus grand* ensemble de verrous, il y aurait moins de compétition. Moins de threads seraient bloqués en attente des verrous et l'adaptabilité s'en trouverait améliorée.

Si un verrou protège plusieurs variables *indépendantes*, il doit être possible d'améliorer l'adaptabilité en le divisant en plusieurs verrous protégeant, chacun, une variable particulière. Chaque verrou sera donc demandé moins souvent.

La classe `ServerStatus` du Listing 11.6 montre une portion de l'interface de surveillance d'un serveur de base de données qui gère l'ensemble des utilisateurs connectés et l'ensemble des requêtes en cours d'exécution. Lorsqu'un utilisateur se connecte ou se déconnecte, ou lorsqu'une requête commence ou finit, l'objet `ServerStatus` est modifié en appelant la méthode `add()` ou `remove()`. Les deux types d'informations sont totalement différents ; `ServerStatus` pourrait même être divisée en deux classes sans perdre de fonctionnalités.

Listing 11.6 : Candidat au découpage du verrou.

```
@ThreadSafe
public class ServerStatus {
    @GuardedBy("this") public final Set<String> users;
    @GuardedBy("this") public final Set<String> queries;
    ...
    public synchronized void addUser(String u) { users.add(u); }
    public synchronized void addQuery(String q) { queries.add(q); }
    public synchronized void removeUser(String u) {
        users.remove(u);
    }
    public synchronized void removeQuery(String q) {
        queries.remove(q);
    }
}
```

Au lieu de protéger `users` et `queries` avec le verrou de `ServerStatus`, nous pouvons protéger chacun d'eux par un verrou distinct, comme dans le Listing 11.7. Après la division du verrou, chaque nouveau verrou plus fin verra moins de trafic que le verrou initial, plus épais (utiliser une implémentation de `Set` thread-safe pour `users` et `queries` au lieu d'employer une synchronisation explicite produirait implicitement une division du verrou car chaque `Set` utiliserait un verrou différent pour protéger son état). Diviser un verrou en deux offre la meilleure possibilité d'amélioration lorsque celui-ci est moyennement disputé. La division de verrous qui sont peu disputés produit peu d'amélioration en termes de performances ou de débit, bien que cela puisse augmenter le seuil de charge à partir duquel les performances commenceront à chuter à cause de la compétition.

Diviser des verrous moyennement disputés peut même en faire des verrous quasiment sans compétition, ce qui est l'idéal pour les performances et l'adaptabilité.

Listing 11.7 : Modification de ServerStatus pour utiliser des verrous divisés.

```

@ThreadSafe
public class ServerStatus {
    @GuardedBy("users") public final Set<String> users;
    @GuardedBy("queries") public final Set<String> queries;
    ...
    public void addUser(String u) {
        synchronized (users) {
            users.add(u);
        }
    }

    public void addQuery(String q) {
        synchronized (queries) {
            queries.add(q);
        }
    }
    // Autres méthodes modifiées pour utiliser les verrous divisés.
}

```

11.4.3 Découpage du verrouillage

Diviser en deux un verrou très disputé produira sûrement deux verrous très disputés. Bien que cela puisse apporter une petite amélioration de l'adaptabilité en autorisant deux threads à s'exécuter en parallèle au lieu d'un seul, cela n'améliore pas énormément les perspectives de concurrence sur un système ayant de nombreux processeurs. L'exemple de division du verrou de la classe `ServerStatus` n'offre pas d'opportunité évidente pour diviser à nouveau chaque verrou.

La division des verrous peut parfois être étendue en un partitionnement du verrouillage sur un ensemble de taille variable d'objets indépendants, auquel cas on parle de *découpage du verrouillage*. L'implémentation de `ConcurrentHashMap`, par exemple, utilise un tableau de 16 verrous protégeant chacun un seizième des entrées du hachage ; l'entrée N est protégée par le verrou $N \bmod 16$. Si l'on suppose que la fonction de hachage fournit une répartition *raisonnable* et qu'on accède aux clés de façon uniforme, cela devrait réduire la demande de chaque verrou d'un facteur de 16 environ. C'est cette technique qui permet à `ConcurrentHashMap` de supporter jusqu'à 16 écrivains concurrents (le nombre de verrous pourrait être augmenté pour fournir une concurrence encore meilleure sur les systèmes ayant un grand nombre de processeurs, mais le nombre de verrous ne doit dépasser 16 que si l'on est sûr que les écrivains concurrents sont suffisamment en compétition).

L'un des inconvénients du découpage des verrous est qu'il devient plus difficile et plus coûteux de verrouiller la collection en accès exclusif qu'avec un seul verrou. Généralement, une opération peut s'effectuer en prenant au plus un verrou mais, parfois, vous devrez verrouiller toute la collection – lorsque `ConcurrentHashMap` doit agrandir le

hachage et donc recalculer les valeurs des clés, par exemple. En règle générale, on prend alors tous les verrous de l'ensemble¹.

La classe `StripedMap` du Listing 11.8 illustre l'implémentation d'un map à l'aide du découpage de verrouillage. Chacun des N_LOCKS protège un sous-ensemble d'entrées. La plupart des méthodes, comme `get()`, n'ont besoin de prendre qu'un seul verrou d'entrée ; certaines peuvent devoir prendre tous les verrous mais, comme le montre `clear()`, pas forcément en même temps².

Listing 11.8 : Hachage utilisant le découpage du verrouillage.

```
@ThreadSafe
public class StripedMap {
    // Politique de synchronisation : buckets[n] protégé par
    // locks[n % N_LOCKS]
    private static final int N_LOCKS = 16;
    private final Node[] buckets;
    private final Object[] locks;

    private static class Node { ... }

    public StripedMap(int numBuckets) {
        buckets = new Node[numBuckets];
        locks = new Object[N_LOCKS];
        for (int i = 0; i < N_LOCKS; i++)
            locks[i] = new Object();
    }

    private final int hash(Object key) {
        return Math.abs(key.hashCode() % buckets.length);
    }

    public Object get(Object key) {
        int hash = hash(key);
        synchronized (locks[hash % N_LOCKS]) {
            for (Node m = buckets[hash]; m != null; m = m.next)
                if (m.key.equals(key))
                    return m.value;
        }
        return null;
    }

    public void clear() {
        for (int i = 0; i < buckets.length; i++) {
            synchronized (locks[i % N_LOCKS]) {
                buckets[i] = null;
            }
        }
    }
    ...
}
```

1. La seule façon de prendre un ensemble quelconque de verrous internes consiste à utiliser la récursivité.
2. Nettoyer la Map de cette façon n'étant pas atomique, il n'existe pas nécessairement un instant précis où l'objet `StripedMap` est vraiment vide si d'autre threads sont en même temps en train de lui ajouter des éléments ; rendre cette opération atomique nécessiterait de prendre en même temps tous les verrous. Cependant, les collections concurrentes ne pouvant généralement pas être verrouillées en accès exclusif par les clients, le résultat de méthodes comme `size()` ou `isEmpty()` peut, de toute façon, être obsolète au moment où elles se terminent ; bien qu'il puisse être surprenant, ce comportement est donc généralement acceptable.

11.4.4 Éviter les points chauds

La division et le découpage du verrouillage peuvent améliorer l'adaptabilité car ces techniques permettent à plusieurs threads de manipuler des données différentes (ou des portions différentes de la même structure de données) sans interférer les uns avec les autres. Un programme qui bénéficierait d'une division du verrouillage met nécessairement en évidence une compétition qui intervient plus souvent pour un *verrou* que pour les *données* protégées par ce verrou. Si un verrou protège deux variables indépendantes *X* et *Y* et que le thread *A* veuille accéder à *X* pendant que *B* veut accéder à *Y* (comme cela serait le cas si un thread appelait `addUser()` pendant qu'un autre appelle `addQuery()` dans `ServerStatus`), les deux threads ne seraient en compétition pour aucune donnée, bien qu'ils le soient pour un verrou.

La granularité du verrouillage ne peut pas être réduite lorsque des variables sont nécessaires pour chaque opération. C'est encore un autre domaine où les performances brutes et l'adaptabilité s'opposent ; les optimisations classiques, comme la mise en cache des valeurs souvent calculées, peuvent produire des "points chauds" qui limitent l'adaptabilité.

Si vous aviez implémenté `HashMap`, vous auriez eu le choix du calcul utilisé par `size()` pour renvoyer le nombre d'entrées dans le Map. L'approche la plus simple compte le nombre d'entrées à chaque fois que la méthode est appelée. Une optimisation classique consiste à modifier un compteur séparé à mesure que des entrées sont ajoutées ou supprimées ; cela augmente légèrement le coût d'une opération d'ajout et de suppression mais réduit celui de l'opération `size()`, qui passe de $O(n)$ à $O(1)$.

Utiliser un compteur séparé pour accélérer des opérations comme `size()` et `isEmpty()` fonctionne parfaitement pour les implémentations monothreads ou totalement synchronisées, mais cela complique beaucoup l'amélioration de l'adaptabilité car chaque opération qui modifie le hachage doit maintenant mettre à jour le compteur partagé. Même en utilisant un découpage du verrouillage pour les chaînes de hachage, la synchronisation de l'accès au compteur réintroduit les problèmes d'adaptabilité des verrous exclusifs. Ce qui semblait être une optimisation des performances – mettre en cache le résultat de l'opération `size()` – s'est transformé en un véritable boulet pour l'adaptabilité. Ici, le compteur est un point chaud car chaque opération de modification doit y accéder.

`ConcurrentHashMap` évite ce problème car sa méthode `size()` énumère les découpages en additionnant les nombres de leurs éléments au lieu de gérer un compteur global. Pour éviter de compter les éléments dans chaque découpage, `ConcurrentHashMap` gère un compteur pour chacun d'eux, qui est également protégé par le verrou du découpage¹.

1. Si `size()` est souvent appelée par rapport aux opérations de modification, les structures de données découpées peuvent optimiser son traitement en mettant en cache la taille de la collection dans une variable volatile à chaque fois que la méthode est appelée et en invalidant le cache (en le mettant à -1) lorsque la collection est modifiée. Si la valeur en cache est positive à l'entrée dans `size()`, c'est qu'elle est exacte et qu'elle peut être renvoyée ; sinon elle est recalculée.

11.4.5 Alternatives aux verrous exclusifs

Une troisième technique pour atténuer l'effet de la compétition sur un verrou consiste à renoncer aux verrous exclusifs au profit de mécanismes de gestion de l'état partagé plus adaptés à la concurrence, comme les collections concurrentes, les verrous de lecture/écriture, les objets non modifiables et les variables atomiques.

La classe `ReadWriteLock` du Chapitre 13 met en application une discipline "plusieurs lecteurs-un seul écrivain" : plusieurs lecteurs peuvent accéder simultanément à la ressource partagée du moment qu'aucun d'entre eux ne veut la modifier ; les écrivains, en revanche, doivent prendre un verrou exclusif. Pour les structures de données qui sont essentiellement lues, `ReadWriteLock` permet une plus grande concurrence que le verrouillage exclusif ; pour les structures de données en lecture seule, l'immuabilité élimine totalement le besoin d'un verrou.

Les variables atomiques (voir Chapitre 15) permettent de réduire le coût de modification des "points chauds" comme les compteurs, les générateurs de séquences ou la référence au premier lien d'une structure de données chaînée (nous avons déjà utilisé `AtomicLong` pour gérer le compteur de visites dans l'exemple des servlets du Chapitre 2). Les classes de variables atomiques fournissent des opérations atomiques très fines (et donc plus adaptables) sur les entiers ou les références d'objets et sont implémentées à l'aide des primitives de concurrence de bas niveau (comme *compare-et-échange*), disponibles sur la plupart des processeurs modernes. Si une classe possède un petit nombre de points chauds qui ne participent pas aux invariants avec d'autres variables, les remplacer par des variables atomiques permettra d'améliorer l'adaptabilité (un changement d'algorithme pour avoir moins de points chauds peut l'améliorer encore plus – les variables atomiques réduisent le coût de la modification des points chauds mais ne l'éliminent pas).

11.4.6 Surveillance de l'utilisation du processeur

Lorsque l'on teste l'adaptabilité, le but est généralement de faire en sorte que les processeurs soient utilisés au maximum. Des outils comme `vmstat` et `mpstat` avec Unix ou `perfmon` avec Windows permettent de connaître le taux d'utilisation des processeurs. Si ces derniers sont utilisés de façon asymétrique (certains processeurs sont au maximum de leur charge alors que d'autres ne le sont pas), le premier but devrait être d'améliorer le parallélisme du programme. Une utilisation asymétrique indique en effet que la plupart des calculs s'exécutent dans un petit ensemble de threads et que l'application ne pourra pas tirer parti de processeurs supplémentaires.

Si les processeurs ne sont pas totalement utilisés, il faut en trouver la raison. Il peut y avoir plusieurs causes :

- **Charge insuffisante.** L'application testée n'est peut-être pas soumise à une charge suffisante. Cela peut être vérifié en augmentant la charge et en mesurant l'impact sur l'utilisation, le temps de réponse ou le temps de service. Produire une charge

suffisante pour saturer une application peut nécessiter une puissance de traitement non négligeable ; le problème peut être que les systèmes clients, pas celui qui est testé, s'exécutent avec cette capacité.

- **Liaison aux E/S.** Pour déterminer si une application est liée au disque, on peut utiliser iostat ou perfmon et surveiller le niveau du trafic sur le réseau pour savoir si elle est limitée par la bande passante.
- **Liaison externe.** Si l'application dépend de services externes comme une base de données ou un service web, le goulet d'étranglement n'est peut-être pas dans son code. Pour le tester, on utilise généralement un profileur ou les outils d'administration de la base de données pour connaître le temps passé à attendre les réponses du service externe.
- **Compétition pour les verrous.** Les outils d'analyse permettent de se rendre compte de la compétition à laquelle sont soumis les verrous de l'application et les verrous qui sont "chauds". On peut souvent obtenir ces informations *via* un échantillonnage aléatoire, en déclenchant quelques traces de threads et en examinant ceux qui combattent pour l'accès aux verrous. Si un thread est bloqué en attente d'un verrou, le cadre de pile correspondant dans la trace de thread indiquera "waiting to lock monitor ... ". Les verrous qui sont peu disputés apparaîtront rarement dans une trace de thread alors que ceux qui sont très demandés auront toujours au moins un thread en attente et seront donc souvent présents dans la trace.

Si l'application garde les processeurs suffisamment occupés, on peut utiliser des outils de surveillance pour savoir s'il serait bénéfique d'en ajouter d'autres. Un programme qui n'a que quatre threads peut occuper totalement un système à quatre processeurs, mais il est peu probable qu'on note une amélioration des performances en passant à un système à huit processeurs puisqu'il faudrait qu'il y ait des threads exécutables en attente pour tirer parti des processeurs supplémentaires (il est également possible de reconfigurer le programme pour diviser sa charge de travail sur plus de threads, en ajustant par exemple la taille du pool de threads). L'une des colonnes du résultat de vmstat est le nombre de threads exécutables mais qui ne s'exécutent pas parce qu'il n'y a pas de processeur disponible ; si l'utilisation du CPU est élevée et qu'il y ait toujours des threads exécutables qui attendent un processeur, l'application bénéficiera sûrement de l'ajout de processeurs supplémentaires.

11.4.7 Dire non aux pools d'objets

Dans les premières versions de la JVM, l'allocation des objets et le ramasse-miettes étaient des opérations lentes¹ mais leurs performances se sont beaucoup améliorées depuis. En fait, l'allocation en Java est désormais plus rapide que malloc() en C : la portion de

1. Comme tout le reste – synchronisation, graphisme, lancement de la JVM, introspection – l'est immuablement dans la première version d'une technologie expérimentale.

code commune entre l'instruction `new Object` de HotSpot 1.4.x et celle de la version 5.0 est d'environ dix instructions machine. Pour contourner les cycles de vie "lents" des objets, de nombreux développeurs se sont tournés vers les *pools d'objets*, où les objets sont recyclés au lieu d'être supprimés par le ramasse-miettes et alloués de nouveau si nécessaire. Même si l'on prend en compte leur surcoût réduit du ramassage des objets, il a été prouvé que dans les programmes monothreads les pools d'objets diminuaient les performances¹ pour tous les objets, sauf les plus coûteux (et cette perte de performance est très importante pour les objets légers et moyens) (Click, 2005).

La situation est encore pire avec les applications concurrentes. Lorsque les threads allouent de nouveaux objets, il n'y a besoin que de très peu de coordination entre eux car les allocateurs utilisent généralement des blocs d'allocation locaux aux threads afin d'éliminer la plupart de la synchronisation sur les structures de données du tas. Si un thread demande un objet du pool, en revanche, il faut une synchronisation pour coordonner l'accès à la structure de données du pool, ce qui introduit la possibilité de blocage du thread. Le blocage d'un thread à cause de la compétition sur un verrou étant des centaines de fois plus coûteux qu'une allocation, même une petite compétition introduite par un pool serait un goulet d'étranglement pour l'adaptabilité (une synchronisation sans compétition est généralement plus coûteuse que l'allocation d'un objet). Il s'agit donc encore d'une autre technique conçue pour améliorer les performances, mais qui s'est traduite en risque pour l'adaptabilité. Les pools ont leur utilité², mais pas pour améliorer les performances.

Allouer des objets est généralement moins coûteux que la synchronisation.

11.5 Exemple : comparaison des performances des Map

Dans un programme monothread, les performances de `ConcurrentHashMap` sont légèrement meilleures que celles d'un `HashMap` synchronisé, mais c'est dans un contexte concurrent que sa suprématie apparaît. L'implémentation de `ConcurrentHashMap` supposant que l'opération la plus courante est la récupération d'une valeur existante, il est donc optimisé

1. Outre qu'ils provoquent une perte des performances en termes de cycles processeurs, les pools d'objets souffrent d'un certain nombre d'autres problèmes, dont la difficulté de configurer correctement les tailles des pools (s'ils sont trop petits, ils n'ont aucun effet ; s'ils sont trop grands, ils surchargent le ramasse-miettes et monopolisent de la mémoire qui aurait pu être utilisée plus efficacement pour autre chose) ; le risque qu'un objet ne soit pas correctement réinitialisé dans son nouvel état (ce qui introduit des bogues subtils) ; le risque qu'un thread renvoie un objet au pool tout en continuant à l'utiliser. Enfin, les pools d'objets demandent plus de travail aux ramasse-miettes générationnels car ils encouragent un motif de références ancien-vers-nouveau.
2. Dans les environnements contraints, comme certaines cibles J2ME ou RTSJ, les pools d'objets peuvent quand même être nécessaires pour gérer la mémoire de façon efficace ou pour contrôler la réactivité.

pour fournir les meilleures performances et la concurrence la plus élevée dans le cas d'opérations `get()` réussies.

Le principal obstacle à l'implémentation des Map synchronisées est qu'il n'y a qu'un seul verrou pour tout l'objet et qu'un seul thread peut donc y accéder à un instant donné. `ConcurrentHashMap`, en revanche, ne verrouille pas la plupart des opérations de lecture réussies et utilise le découpage des verrous pour les opérations d'écriture et les rares opérations de lecture qui ont besoin du verrouillage. Plusieurs threads peuvent donc accéder simultanément à l'objet Map sans risquer de se bloquer.

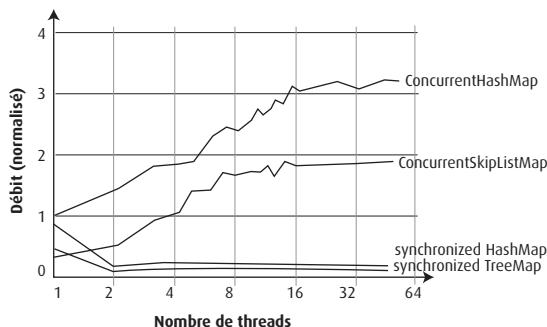
La Figure 11.3 illustre les différences d'adaptabilité entre plusieurs implémentations de Map : `ConcurrentHashMap`, `ConcurrentSkipListMap` et `HashMap` et `TreeMap` enveloppées avec `synchronizedMap()`. Les deux premières sont thread-safe de par leur conception ; les deux dernières le sont grâce à l'enveloppe de synchronisation. Dans chaque test, N threads exécutent simultanément une boucle courte qui choisit une clé au hasard et tente de récupérer la valeur qui lui correspond. Si cette valeur n'existe pas, elle est ajoutée à l'objet Map avec la probabilité $p = 0,6$; si elle existe, elle est supprimée avec la probabilité $p = 0,02$. Les tests se sont déroulés avec une version de développement de Java 6 sur une machine Sparc V880 à huit processeurs et le graphique affiche le débit normalisé par rapport au débit d'un seul thread avec `ConcurrentHashMap` (avec Java 5.0, l'écart d'adaptabilité est encore plus grand entre les collections concurrentes et synchronisées).

Les résultats de `ConcurrentHashMap` et `ConcurrentSkipListMap` montrent que ces classes s'adaptent très bien à un grand nombre de threads ; le débit continue en effet d'augmenter à mesure que l'on en ajoute. Bien que le nombre de threads de la Figure 11.3 ne semble pas élevé, ce programme de test produit plus de compétition par thread qu'une application classique car il se contente essentiellement de maltraiter l'objet Map ; un programme normal ajouterait des traitements locaux aux threads lors de chaque itération.

Les résultats des collections synchronisées ne sont pas aussi encourageants. Avec un seul thread, les performances sont comparables à celles de `ConcurrentHashMap` mais, dès que la charge passe d'une situation essentiellement sans compétition à une situation très disputée – ce qui a lieu ici avec deux threads –, les collections synchronisées souffrent beaucoup. C'est un comportement classique des codes dont l'adaptabilité est limitée par la compétition sur les verrous. Tant que cette compétition est faible, le temps par opération est dominé par celui du traitement et le débit peut augmenter en même temps que le nombre de threads. Lorsque la compétition devient importante, ce sont les délais des changements de contexte et de la planification qui l'emportent sur le temps de traitement ; ajouter des threads a alors peu d'effet sur le débit.

Figure 11.3

Comparaison de l'adaptabilité des implémentations de Map.



11.6 Réduction du surcoût des changements de contexte

De nombreuses tâches utilisent des opérations qui peuvent se bloquer ; le passage de l'état "en cours d'exécution" à "bloqué" implique un changement de contexte. Dans les applications serveur, une source de blocage est la production des messages du journal au cours du traitement des requêtes ; pour illustrer comment il est possible d'améliorer le débit en réduisant ces changements de contexte, nous analyserons le comportement des deux approches pour l'écriture de ces messages.

La plupart des frameworks de journalisation ne font qu'envelopper légèrement `println()` ; lorsqu'il faut inscrire quelque chose dans le journal, on se contente d'y écrire. La classe `LogWriter` du Listing 7.13 présentait une autre approche : l'inscription des messages s'effectuait dans un thread dédié en arrière-plan et non dans le thread demandeur. Du point de vue du développeur, ces deux approches sont à peu près équivalentes, mais elles peuvent avoir des performances différentes en fonction du volume de l'activité de journalisation, du nombre de threads qui inscrivent des messages et d'autres facteurs comme le coût des changements de contexte¹.

Le temps de service d'une inscription dans le journal inclut tous les calculs associés aux classes des flux d'E/S ; si l'opération d'E/S est bloquée, il comprend également la durée de blocage du thread. Le système d'exploitation déprogrammera le thread bloqué jusqu'à ce que l'E/S se termine – probablement un peu plus longtemps. À la fin de l'E/S, les threads qui sont actifs seront autorisés à terminer leur quantum de temps ; les threads qui sont déjà en attente s'ajouteront au temps de service. Une compétition pour

1. La construction d'un logger qui déplace les E/S vers un autre thread permet d'améliorer les performances mais introduit également un certain nombre de complications pour la conception, comme les interruptions (que se passera-t-il si un thread bloqué dans une opération avec le journal est interrompu ?), les garanties de service (le logger garantit-il qu'un message placé dans la file d'attente du journal sera inscrit dans celui-ci avant l'arrêt du service ?), la politique de saturation (que se passera-t-il lorsque les producteurs inscriront des messages dans le journal plus vite que le thread logger ne peut les traiter ?) et le cycle de vie du service (comment éteindre le logger et comment communiquer l'état du service aux producteurs ?).

l'accès au verrou du flux de sortie peut survenir si plusieurs threads envoient simultanément des messages au journal, auquel cas le résultat sera le même que pour les E/S bloquantes – les threads se bloquent sur le verrou et sont déprogrammés. Une journalisation en ligne implique des E/S et des verrous, ce qui peut conduire à une augmentation des changements de contexte et donc à un temps de service plus long.

L'augmentation du temps de service n'est pas souhaitable pour plusieurs raisons. La première est que le temps de service affecte la qualité du service – un temps plus long signifie que l'on attendra plus longtemps le résultat –, mais ce qui est plus grave est que des temps de service plus longs signifient ici plus de compétition pour les verrous. Le principe "rentre et sort" de la section 11.4.1 nous enseigne qu'il faut détenir les verrous le moins longtemps possible car plus le verrouillage dure, plus il y aura de compétition. Si un thread se bloque en attente d'une E/S pendant qu'il détient un verrou, un autre thread voudra sûrement le prendre pendant que le premier le détient. Les systèmes concurrents fonctionnent bien mieux lorsque les verrous ne sont pas disputés car l'acquisition d'un verrou très demandé implique plus de changements de contexte, or un style de programmation qui encourage plus de changements de contexte produit un débit global plus faible.

Déplacer les E/S en dehors du thread de traitement de la requête raccourcira le temps de service moyen. Les threads qui écrivent dans le journal ne se bloqueront plus en attente du verrou du flux de sortie ou de la fin de l'opération d'écriture ; ils n'ont besoin que de placer le message dans la file d'attente avant de revenir à leur traitement. Cela introduit évidemment une compétition possible pour la file d'attente, mais l'opération `put()` est plus légère que l'écriture dans le journal (qui peut nécessiter des appels systèmes) et risque donc moins de se bloquer (du moment que la file n'est pas pleine). Le thread de traitement de la requête se bloquera donc moins et il y aura moins de changements de contexte au beau milieu d'une requête. Nous avons donc transformé une portion de code compliquée et incertaine, impliquant des E/S et une possible compétition pour les verrous, en une portion de code en ligne droite.

Dans un certain sens, nous n'avons fait que contourner le problème en déplaçant l'E/S dans un thread dont le coût n'est pas perceptible par l'utilisateur (ce qui, en soi, est une victoire). Mais, en déplaçant dans un seul thread *toutes* les écritures dans le journal, nous avons également éliminé le risque de compétition pour le flux de sortie et donc une source de blocage, ce qui améliore le débit global puisque moins de ressources sont consommées en planification, changements de contexte et gestion des verrous.

Déplacer les E/S des nombreux threads de traitement des requêtes dans un seul thread de gestion du journal produit la même différence qu'entre une brigade de pompiers faisant une chaîne avec des seaux et un ensemble d'individus luttant séparément contre un feu. Dans l'approche "des centaines de personnes courant avec des seaux", il y a plus de risque de compétition pour la source d'eau et pour l'accès au feu (il y aura donc globalement moins d'eau déversée sur le feu). L'efficacité est également moindre puisque

chaque personne change continuellement de mode (remplissage, course, vidage, course, etc.). Dans l’approche de la brigade de pompiers, le flux de l’eau allant de la source vers l’incendie est constant, moins d’énergie est dépensée pour transporter l’eau vers le feu et chaque intervenant se dévoue continuellement à la même tâche. Tout comme les interruptions perturbent et réduisent la productivité humaine, les blocages et les changements de contexte perturbent les threads.

Résumé

L’une des raisons les plus fréquentes d’utiliser les threads est d’exploiter plusieurs processeurs. Lorsque l’on étudie les performances des applications concurrentes, nous nous intéressons donc généralement plus au débit et à l’adaptabilité qu’au temps de service brut. La loi d’Amdahl nous indique que l’adaptabilité d’une application est gouvernée par la proportion de code qui doit s’exécuter en série. La source essentielle de la sérialisation dans les programmes Java étant le verrouillage exclusif des ressources, nous pouvons souvent améliorer l’adaptabilité en détenant les verrous pendant moins longtemps, soit en réduisant la granularité du verrouillage, soit en remplaçant les verrous exclusifs par des alternatives non exclusives ou non bloquantes.

Tests des programmes concurrents

Les programmes concurrents emploient des principes et des patrons de conception semblables à ceux des programmes séquentiels. La différence est que, contrairement à ces derniers, leur léger non-déterminisme augmente le nombre d'interactions et de types d'erreurs possibles, qui doivent être prévues et analysées.

Les tests des programmes concurrents utilisent et étendent les idées des tests des programmes séquentiels. On utilise donc les mêmes techniques pour tester la justesse et les performances, mais l'espace de problèmes possibles est bien plus vaste avec les programmes concurrents qu'avec les programmes séquentiels. Le plus grand défi des tests des programmes concurrents est que certaines erreurs peuvent être rares et aléatoires au lieu d'être déterministes ; pour les trouver, les tests doivent donc être plus poussés et s'exécuter plus longtemps que les tests séquentiels classiques.

La plupart des tests des classes concurrentes appartiennent aux deux catégories classiques de la sécurité et de la vivacité. Au Chapitre 1, nous avons défini la première comme "rien de mauvais ne se passera jamais" et la seconde comme "quelque chose de bon finira par arriver".

Les tests de la sécurité, qui vérifient que le comportement d'une classe est conforme à sa spécification, consistent généralement à tester des invariants. Dans une implémentation de liste chaînée qui met en cache la taille de la liste à chaque fois qu'elle est modifiée, par exemple, un test de sécurité consisterait à comparer la taille en cache avec le nombre actuel des éléments de la liste. Dans un programme monothread, ce test est simple puisque le contenu de la liste ne change pas pendant que l'on teste ses propriétés. Dans un programme concurrent, en revanche, ce genre de test peut être perverti par des situations de compétition, sauf si la lecture de la valeur en cache et le comptage des éléments s'effectuent dans une seule opération atomique. Pour ce faire, on peut verrouiller la liste pour disposer d'un accès exclusif en se servant d'une sorte d'*instantané atomique* fourni par l'implémentation ou de "points de tests" permettant de vérifier les invariants

ou d'exécuter du code de test de façon atomique. Dans ce livre, nous avons utilisé des diagrammes temporels pour représenter les interactions "malheureuses" pouvant causer des erreurs avec les classes mal construites ; les programmes de tests tentent d'examiner suffisamment l'espace des états pour que ces situations malheureuses finissent par arriver. Malheureusement, le code de test peut introduire des artefacts de timing ou de synchronisation qui peuvent masquer des bogues qui auraient pu se manifester d'eux-mêmes¹.

Les tests de la vivacité doivent relever leurs propres défis car ils comprennent les tests de progression et de non-progression, qui sont difficiles à quantifier – comment vérifier, par exemple, qu'une méthode est bloquée et qu'elle ne s'exécute pas simplement lentement ? De même, comment tester qu'un algorithme ne provoquera pas un interblocage ? Pendant combien de temps faut-il attendre avant de déclarer qu'il a échoué ?

Les tests de performances sont liés aux tests de vivacité. Les performances peuvent se mesurer de diverses façons, notamment :

- **Le débit.** La vitesse à laquelle un ensemble de tâches concurrentes se termine.
- **La réactivité.** Le délai entre une requête et l'achèvement de l'action (également appelée latence).
- **L'adaptabilité.** L'amélioration du débit (ou non) lorsque l'on dispose de ressources supplémentaires (généralement des processeurs).

12.1 Tests de la justesse

Le développement de tests unitaires pour une classe concurrente commence par la même analyse que pour une classe séquentielle – on identifie les invariants et les post-conditions qui peuvent être testés mécaniquement. Avec un peu de chance, la plupart sont décrits dans la spécification ; le reste du temps, l'écriture des tests est une aventure dans la découverte itérative des spécifications.

À titre d'exemple, nous allons construire un ensemble de cas de tests pour un tampon borné. Le Listing 12.1 contient le code de la classe `BoundedBuffer`, qui utilise `Semaphore` pour borner le tampon et pour éviter les blocages. `BoundedBuffer` implémente une file reposant sur un tableau de taille fixe et fournissant des méthodes `put()` et `take()` contrôlées par une paire de sémaphores. Le sémaphore `availableItems` représente le nombre d'éléments pouvant être ôtés du tampon ; il vaut initialement zéro (puisque le tampon est vide au départ). De même, `availableSpaces` représente le nombre d'éléments pouvant être ajoutés au tampon ; il est initialisé avec la taille du tampon.

1. Les bogues qui disparaissent lorsque l'on débogue ou que l'on ajoute du code de test sont appelés *Heisen bogues*.

Listing 12.1 : Tampon borné utilisant la classe Semaphore.

```

@ThreadSafe
public class BoundedBuffer<E> {
    private final Semaphore availableItems, availableSpaces ;
    @GuardedBy("this") private final E[] items;
    @GuardedBy("this") private int putPosition = 0, takePosition = 0;

    public BoundedBuffer(int capacity) {
        availableItems = new Semaphore(0);
        availableSpaces = new Semaphore(capacity);
        items = (E[]) new Object[capacity];
    }
    public boolean isEmpty() {
        return availableItems.availablePermits() == 0;
    }
    public boolean isFull() {
        return availableSpaces.availablePermits() == 0;
    }

    public void put(E x) throws InterruptedException {
        availableSpaces.acquire();
        doInsert(x);
        availableItems.release();
    }
    public E take() throws InterruptedException {
        availableItems.acquire();
        E item = doExtract();
        availableSpaces.release();
        return item;
    }

    private synchronized void doInsert(E x) {
        int i = putPosition;
        items[i] = x;
        putPosition = (++i == items.length)? 0 : i;
    }
    private synchronized E doExtract() {
        int i = takePosition;
        E x = items[i];
        items[i] = null;
        takePosition = (++i == items.length)? 0 : i;
        return x;
    }
}

```

Une opération `take()` nécessite d'abord d'obtenir un permis de la part de `availableItems`, ce qui réussit tout de suite si le tampon n'est pas vide mais qui, sinon, bloque l'opération jusqu'à ce qu'il ne soit plus vide. Lorsque ce permis a été obtenu, l'élément suivant du tampon est ôté et on délivre un permis au sémaphore `availableSpaces`¹. L'opération `put()` a le fonctionnement inverse ; à la sortie des méthodes `put()` ou `take()`, la somme des compteurs des deux sémaphores est toujours égale à la taille du tableau (en pratique, il vaudrait mieux utiliser la classe `ArrayBlockingQueue` ou `LinkedBlockingQueue` pour implémenter un tampon borné plutôt qu'en construire un soi-même,

1. Dans un sémaphore, les permis ne sont pas représentés explicitement ni associés à un thread particulier ; une opération `release()` crée un permis et une opération `acquire()` en consomme un.

mais la technique utilisée ici illustre également comment contrôler les insertions et les suppressions dans d'autres structures de données).

12.1.1 Tests unitaires de base

Les tests unitaires les plus basiques pour `BoundedBuffer` ressemblent à ceux que l'on utiliserait dans un contexte séquentiel – création d'un tampon borné, appel à ses méthodes et assertions sur les postconditions et les invariants. Parmi ces derniers, on pense immédiatement au fait qu'un tampon venant d'être créé devrait savoir qu'il est vide et qu'il n'est pas plein. Un test de sécurité similaire mais un peu plus compliqué consiste à insérer N éléments dans un tampon de capacité N (ce qui devrait s'effectuer sans blocage) et à tester que le tampon reconnaissse qu'il est plein (et non vide). Les méthodes de tests JUnit pour ces propriétés sont présentées dans le Listing 12.2.

Listing 12.2 : Tests unitaires de base pour `BoundedBuffer`.

```
class BoundedBufferTest extends TestCase {  
    void testIsEmptyWhenConstructed() {  
        BoundedBuffer<Integer> bb = new BoundedBuffer<Integer>(10);  
        assertTrue(bb.isEmpty());  
        assertFalse(bb.isFull());  
    }  
  
    void testIsFullAfterPuts() throws InterruptedException {  
        BoundedBuffer<Integer> bb = new BoundedBuffer<Integer>(10);  
        for (int i = 0; i < 10; i++)  
            bb.put(i);  
        assertTrue(bb.isFull());  
        assertFalse(bb.isEmpty());  
    }  
}
```

Ces méthodes de tests simples sont entièrement séquentielles. Il est souvent utile d'inclure un ensemble de tests séquentiels dans une suite de tests car ils permettent de dévoiler les problèmes indépendants de la concurrence avant de commencer à rechercher les situations de compétition.

12.1.2 Tests des opérations bloquantes

Les tests des propriétés essentielles de la concurrence nécessitent d'introduire plusieurs threads. La plupart des frameworks de tests ne sont pas particulièrement conçus pour cela : ils facilitent rarement la création de threads ou leur surveillance pour vérifier qu'ils ne meurent pas de façon inattendue. Si un thread auxiliaire créé par un test découvre une erreur, le framework ne sait généralement pas à quel test ce thread est attaché et un travail supplémentaire est donc nécessaire pour relayer le succès ou l'échec au thread principal des tests, afin qu'il puisse le signaler.

Pour les tests de conformité de `java.util.concurrent`, il était important que les échecs soient clairement associés à un test spécifique. Le *JSR 166 Expert Group* a donc

créé une classe de base¹ qui fournissait des méthodes pour relayer et signaler les erreurs au cours de `tearDown()` en respectant la convention que chaque test devait attendre que tous les threads qu'il avait créés se terminent. Vous n'avez peut-être pas besoin d'aller jusqu'à de telles extrémités ; l'essentiel est de savoir clairement si les tests ont eu lieu et que les informations sur les échecs soient signalées quelque part, afin que l'on puisse les utiliser pour diagnostiquer le problème.

Si une méthode est censée se bloquer sous certaines conditions, un test de ce comportement ne devrait réussir que si le thread est bloqué : tester qu'une méthode se bloque revient à tester qu'elle lance une exception ; si son exécution se poursuit, c'est que le test a échoué.

Tester qu'une méthode se bloque introduit une complication supplémentaire : lorsque la méthode s'est correctement bloquée, vous devez la convaincre de se débloquer. La façon la plus évidente de le faire consiste à utiliser une interruption – lancer une activité bloquante dans un thread distinct, attendre que le thread se bloque, l'interrompre, puis tester que l'opération bloquante s'est terminée. Ceci nécessite évidemment que les méthodes bloquantes répondent aux interruptions en se terminant prématurément ou en lançant l'exception `InterruptedException`.

La partie "attendre que le thread se bloque" est plus facile à dire qu'à faire ; en pratique, il faut prendre une décision arbitraire sur la durée que peuvent avoir les quelques instructions exécutées et attendre plus longtemps. Vous devez vous préparer à augmenter ce délai si vous vous êtes trompé (auquel cas vous constaterez de faux échecs des tests).

Le Listing 12.3 montre une approche permettant de tester les opérations bloquantes. La méthode crée un thread `taker` qui tente de prendre un élément dans un tampon vide. Si cette opération réussit, il constate l'échec. Le thread qui exécute lance `taker` et attend un certain temps avant de l'interrompre. Si `taker` s'est correctement bloqué dans l'opération `take()`, il lancera une exception `InterruptedException` ; le bloc `catch` correspondant la traitera comme un succès et permettra au thread de se terminer. Le thread de test principal attend alors la fin de `taker` avec `join()` et vérifie que cet appel a réussi en appelant `Thread.isAlive()` ; si le thread `taker` a répondu à l'interruption, l'appel à `join()` devrait se terminer rapidement.

Listing 12.3 : Test du blocage et de la réponse à une interruption.

```
void testTakeBlocksWhenEmpty() {
    final BoundedBuffer<Integer> bb = new BoundedBuffer <Integer>(10);
    Thread taker = new Thread() {
        public void run() {
            try {
                int unused = bb.take();
                fail(); // if we get here, it's an error
            } catch (InterruptedException success) { }
        }
    };
    try {
```

1. Voir <http://gee.cs.oswego.edu/cgi-bin/viewcvs.cgi/jsr166/src/test/tck/JSR166TestCase.java>.

Listing 12.3 : Test du blocage et de la réponse à une interruption. (suite)

```
taker.start();
Thread.sleep(LOCKUP_DETECT_TIMEOUT );
taker.interrupt();
taker.join(LOCKUP_DETECT_TIMEOUT );
assertFalse(taker.isAlive());
} catch (Exception unexpected) {
    fail();
}
}
```

La jointure avec délai garantit que le test se terminera même si `take()` se fige de façon inattendue. Cette méthode teste plusieurs propriétés de `take()` – pas seulement qu'elle se bloque, mais également qu'elle lance `InterruptedException` lorsqu'elle est interrompue. C'est l'un des rares cas où il est justifié de sous-classer explicitement `Thread` au lieu d'utiliser un `Runnable` dans un pool puisque cela permet de tester la terminaison correcte avec `join()`. On peut utiliser la même approche pour tester que `taker` se débloque quand le thread principal ajoute un élément à la file.

Bien qu'il soit tentant d'utiliser `Thread.getState()` pour vérifier que le thread est bien bloqué sur l'attente d'une condition, cette approche n'est pas fiable. Rien n'exige en effet qu'un thread bloqué soit dans l'état `WAITING` ou `TIMED_WAITING` puisque la JVM peut choisir d'implémenter le blocage par une attente tournante. De même, les faux réveils de `Object.wait()` ou `Condition.await()` étant autorisés (voir Chapitre 14), un thread dans l'état `WAITING` ou `TIMED_WAITING` peut passer temporairement dans l'état `RUNNABLE`, même si la condition qu'il attend n'est pas encore vérifiée. Même en ignorant ces détails d'implémentation, le thread cible peut mettre un certain temps pour se stabiliser dans un état bloquant. *Le résultat de `Thread.getState()` ne devrait pas être utilisé pour contrôler la concurrence et son utilité est limitée pour les tests – son principal intérêt est de fournir des informations qui servent au débogage.*

12.1.3 Test de la sécurité vis-à-vis des threads

Les Listings 12.2 et 12.3 testent des propriétés importantes du tampon borné mais ne découvriront sûrement pas les erreurs dues aux situations de compétition. Pour tester qu'une classe concurrente se comporte correctement dans le cas d'accès concurrents imprévisibles, il faut mettre en place plusieurs threads effectuant des opérations `put()` et `take()` pendant un certain temps et vérifier que tout s'est bien passé.

La construction de tests pour découvrir les erreurs de sécurité vis-à-vis des threads dans les classes concurrentes est un problème identique à celui de l'œuf et de la poule car les programmes de tests sont souvent eux-mêmes des programmes concurrents. Développer de bons tests concurrents peut donc se révéler plus difficile que l'écriture des classes qu'ils testent.

Le défi de la construction de tests efficaces pour la sécurité des classes concurrentes vis-à-vis des threads consiste à identifier facilement les propriétés qui échoueront sûrement si quelque chose se passe mal, tout en ne limitant pas artificiellement la concurrence par le code d'analyse des erreurs. Il est préférable que le test d'une propriété ne nécessite pas de synchronisation.

Une approche qui fonctionne bien avec les classes utilisées dans les conceptions producteur-consommateur (comme `BoundedBuffer`) consiste à vérifier que tout ce qui est placé dans une file ou un tampon en ressort et rien d'autre. Une application naïve de cette méthode serait d'insérer l'élément dans une liste "fantôme" lorsqu'il est placé dans la file, de l'enlever de cette liste lorsqu'il est ôté de la file et de tester que la liste fantôme est vide quand le test s'est terminé. Cependant, cette approche déformerait la planification des threads de test car la modification de la liste fantôme nécessiterait une synchronisation et risquerait de provoquer des blocages.

Une meilleure technique consiste à calculer les sommes de contrôle des éléments mis dans la file et sortis de celle-ci en utilisant une fonction de calcul tenant compte de l'ordre, puis à les comparer. S'ils correspondent, le test a réussi. Cette approche fonctionne mieux lorsqu'il n'y a qu'un producteur qui place les éléments dans le tampon et un seul consommateur qui les en extrait car elle peut tester non seulement que ce sont (sûrement) les bons éléments qui sortent, mais également qu'ils sont extraits dans le bon ordre.

Pour étendre cette méthode à une situation multi-producteurs/multi-consommateurs, il faut utiliser une fonction de calcul de la somme de contrôle *qui ne tienne pas compte* de l'ordre dans lequel les éléments sont combinés, afin que les différentes sommes de contrôle puissent être combinées après le test. Sinon la synchronisation de l'accès à un champ de contrôle partagé deviendrait un goulet d'étranglement pour la concurrence ou perturberait le timing du test (toute opération commutative, comme une addition ou un "ou exclusif", correspond à ces critères). Pour vérifier que le test effectue bien les vérifications souhaitées, il est important que les sommes de contrôle ne puissent être anticipées par le compilateur. Ce serait donc une mauvaise idée d'utiliser des entiers consécutifs comme données du test car le résultat serait toujours le même et un compilateur évolué pourrait les calculer à l'avance.

Pour éviter ce problème, les données de test devraient être produites aléatoirement, mais un mauvais choix de générateur de nombre aléatoire (GNA) peut compromettre de nombreux autres tests. La génération des nombres aléatoires peut, en effet, créer des couplages entre les classes et les artefacts de timing car la plupart des classes de génération sont thread-safe et introduisent donc une synchronisation supplémentaire¹. Si

1. De nombreux tests de performances ne sont, à l'insu de leurs développeurs ou de leurs utilisateurs, que des tests qui évaluent la taille du goulet d'étranglement pour la concurrence que constitue le GNA.

chaque thread utilise son propre GNA, on peut utiliser un générateur de nombre aléatoire non thread-safe.

Les fonctions pseudo-aléatoires sont à tout faire préférables aux générateurs. Il n'est pas nécessaire d'obtenir une haute qualité d'aléa : il suffit qu'il puisse garantir que les nombres changeront d'une exécution à l'autre. La fonction `xorShift()` du Listing 12.4 (Marsaglia, 2003) fait partie de ces fonctions peu coûteuses qui fournissent des nombres aléatoires de qualité moyenne. En la lançant avec des valeurs reposant sur `hashCode()` et `nanoTime()`, les sommes seront suffisamment imprévisibles et presque toujours différentes à chaque exécution.

Listing 12.4 : Générateur de nombre aléatoire de qualité moyenne mais suffisante pour les tests.

```
static int xorShift(int y) {
    y ^= (y << 6);
    y ^= (y >> 21);
    y ^= (y << 7);
    return y;
}
```

La classe `PutTakeTest` des Listings 12.5 et 12.6 lance N threads producteurs qui génèrent des éléments et les placent dans une file et N threads consommateurs qui les extraient. Chaque thread met à jour la somme de contrôle des éléments à mesure qu'ils entrent ou sortent en utilisant une somme de contrôle par thread qui est combinée à la fin de l'exécution du test, afin de ne pas ajouter plus de synchronisation ou de compétition que nécessaire.

Listing 12.5 : Programme de test producteur-consommateur pour `BoundedBuffer`.

```
public class PutTakeTest {
    private static final ExecutorService pool
        = Executors.newCachedThreadPool();
    private final AtomicInteger putSum = new AtomicInteger(0);
    private final AtomicInteger takeSum = new AtomicInteger(0);
    private final CyclicBarrier barrier;
    private final BoundedBuffer <Integer> bb;
    private final int nTrials, nPairs;

    public static void main(String[] args) {
        new PutTakeTest(10, 10, 100000).test(); // params exemple
        pool.shutdown();
    }

    PutTakeTest(int capacity, int npairs, int ntrials) {
        this.bb = new BoundedBuffer<Integer>(capacity);
        this.nTrials = ntrials;
        this.nPairs = npairs;
        this.barrier = new CyclicBarrier(npairs * 2 + 1);
    }

    void test() {
        try {
            for (int i = 0; i < nPairs; i++) {
                pool.execute(new Producer());
            }
        }
```

```

        pool.execute(new Consumer());
    }
    barrier.await(); // Attend que tous les threads soient prêts
    barrier.await(); // Attend que tous les threads aient fini
    assertEquals(putSum.get(), takeSum.get());
} catch (Exception e) {
    throw new RuntimeException (e);
}
}

class Producer implements Runnable { /* Listing 12.6 */ }

class Consumer implements Runnable { /* Listing 12.6 */ }
}

```

Listing 12.6 : Classes producteur et consommateur utilisées dans PutTakeTest.

```

/* Classes internes de PutTakeTest (Listing 12.5) */
class Producer implements Runnable {
    public void run() {
        try {
            int seed = (this.hashCode() ^ (int)System.nanoTime());
            int sum = 0;
            barrier.await();
            for (int i = nTrials; i > 0; --i) {
                bb.put(seed);
                sum += seed;
                seed = xorShift(seed);
            }
            putSum.getAndAdd(sum);
            barrier.await();
        } catch (Exception e) {
            throw new RuntimeException (e);
        }
    }
}

class Consumer implements Runnable {
    public void run() {
        try {
            barrier.await();
            int sum = 0;
            for (int i = nTrials; i > 0; --i) {
                sum += bb.take();
            }
            takeSum.getAndAdd(sum);
            barrier.await();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}

```

En fonction de la plate-forme, la création et le lancement d'un thread peuvent être des opérations plus ou moins lourdes. Si les threads sont courts et qu'on en lance un certain nombre dans une boucle, ceux-ci s'exécuteront séquentiellement et non en parallèle dans le pire des cas. Même avant cette situation extrême, le fait que le premier thread ait une longueur d'avance sur les autres signifie qu'il peut y avoir moins d'entrelacements que prévu : le premier thread s'exécute seul pendant un moment, puis les deux premiers s'exécutent en parallèle pendant un certain temps et il n'est pas dit que tous les threads

finiront par s'exécuter en parallèle (le même phénomène a lieu à la fin de l'exécution : les threads qui ont eu une longueur d'avance se finissent également plus tôt).

Dans la section 5.5.1, nous avions présenté une technique permettant d'atténuer ce problème en utilisant un objet `CountDownLatch` comme porte d'entrée et un autre comme porte de sortie. Un autre moyen d'obtenir le même effet consiste à utiliser un objet `CyclicBarrier` initialisé avec le nombre de threads plus un et à faire en sorte que les threads et le pilote de test attendent à la barrière au début et à la fin de leurs exécutions. Ceci garantit que tous les threads seront lancés avant qu'un seul ne commence son travail. C'est la technique qu'utilise `PutTakeTest` pour coordonner le lancement et l'arrêt des threads, ce qui crée plus de possibilités d'entrelacement concurrent. Nous ne pouvons quand même pas garantir que le planificateur n'exécutera pas séquentiellement chaque thread mais, si ces exécutions sont suffisamment longues, on réduit les conséquences possibles de la planification sur les résultats.

La dernière astuce employée par `PutTakeTest` consiste à utiliser un critère de terminaison déterministe afin qu'il n'y ait pas besoin de coordination interthread supplémentaire pour savoir quand le test est fini. La méthode de test lance autant de producteurs que de consommateurs, chacun d'eux plaçant ou retirant le même nombre d'éléments afin que le nombre total des éléments ajoutés soit égal à celui des éléments retirés.

Les tests de `PutTakeTest` se révèlent assez efficaces pour trouver les violations de la thread safety. Une erreur classique lorsque l'on implémente des tampons contrôlés par des sémaphores, par exemple, est d'oublier que le code qui effectue l'insertion et l'extraction doit s'exécuter en exclusion mutuelle (à l'aide de `synchronized` ou de `ReentrantLock`). Ainsi, le lancement de `PutTakeTest` avec une version de `BoundedBuffer` ne précisant pas que `doInsert()` et `doExtract()` sont `synchronized` échouerait assez rapidement. Lancer `PutTakeTest` avec quelques dizaines de threads qui itèrent quelques millions de fois sur des tampons de capacités variées avec des systèmes différents augmentera votre confiance quant à l'absence de corruption des données avec `put()` et `take()`.

Pour augmenter la diversité des entrelacements possibles, les tests devraient s'effectuer sur des systèmes multiprocesseurs. Cependant, les tests ne seront pas nécessairement plus efficaces au-delà de quelques processeurs. Pour maximiser la chance de détecter les situations de compétition liées au timing, il devrait y avoir plus de threads actifs que de processeurs, afin qu'à un instant donné certains threads s'exécutent et d'autres soient sortis du processeur, ce qui réduit ainsi la possibilité de prévoir les interactions entre les threads.

Avec les tests qui exécutent un nombre fixé d'opérations, il est possible que le cas de test ne se termine jamais si le code testé rencontre une exception à cause d'un bug. La méthode la plus classique de traiter cette situation consiste à faire en sorte que le framework de test mette brutalement fin aux tests qui ne se terminent pas au bout d'un certain temps ; ce délai devrait être déterminé empiriquement et les erreurs doivent être

analysées pour vérifier que le problème n'est pas simplement dû au fait que l'on n'a pas attendu assez longtemps (ce n'est pas un problème propre aux tests des classes concurrentes car les tests séquentiels doivent également distinguer les boucles longues des boucles sans fin).

12.1.4 Test de la gestion des ressources

Pour l'instant, les tests se sont intéressés à la conformité d'une classe par rapport à sa spécification – à ce qu'elle fasse ce qu'elle est censée faire. Un deuxième point à vérifier est qu'elle ne fasse pas ce qu'elle n'est pas censée faire, comme provoquer une fuite des ressources. Tout objet qui contient ou gère d'autres objets ne devrait pas continuer à conserver des références à ces objets plus longtemps qu'il n'est nécessaire. Ces fuites de stockage empêchent en effet les ramasse-miettes de récupérer la mémoire (ou les threads, les descripteurs de fichiers, les sockets, les connexions aux bases de données ou toute autre ressource limitée) et peuvent provoquer un épuisement des ressources et un échec de l'application.

Les problèmes de gestion des ressources ont une importance toute particulière pour les classes comme `BoundedBuffer` – la seule raison de borner un tampon est d'empêcher que l'application n'échoue à cause d'un épuisement des ressources lorsque les producteurs sont trop en avance par rapport aux consommateurs. La limitation de la taille du tampon fera que les producteurs trop productifs se bloqueront au lieu de continuer à créer du travail qui consommera de plus en plus de mémoire ou d'autres ressources.

On peut facilement tester la rétention abusive de la mémoire à l'aide d'outils d'inspection du tas, qui permettent de mesurer l'utilisation mémoire de l'application ; il existe un grand nombre d'outils commerciaux et open-source qui permettent d'effectuer ce type d'analyse. La méthode `testLeak()` du Listing 12.7 contient des emplacements pour qu'un tel outil puisse faire un instantané du tas, forcer l'application du ramasse-miettes¹, puis enregistrer les informations sur la taille du tas et l'utilisation de la mémoire.

Listing 12.7 : Test des fuites de ressources.

```
class Big { double[] data = new double[100000]; }

void testLeak() throws InterruptedException {
    BoundedBuffer<Big> bb = new BoundedBuffer<Big>(CAPACITY);
    int heapSize1 = /* instantané du tas */;
    for (int i = 0; i < CAPACITY; i++)
        bb.put(new Big());
    for (int i = 0; i < CAPACITY; i++)
        bb.take();
    int heapSize2 = /* instantané du tas */;
    assertTrue(Math.abs(heapSize1 - heapSize2) < THRESHOLD);
}
```

1. Techniquement, il est impossible de forcer l'application du ramasse-miettes ; `System.gc()` ne fait que suggérer à la JVM qu'il pourrait être souhaitable de le faire. On peut demander à HotSpot d'ignorer les appels à `System.gc()` à l'aide de l'option `-XX:+DisableExplicitGC`.

La méthode `testLeak()` insère plusieurs gros objets dans un tampon borné puis les en extrait ; l'utilisation de la mémoire au moment du deuxième instantané du tas devrait être approximativement identique à celle relevée lors du premier. Si `doExtract()`, en revanche, a oublié de mettre à `null` la référence vers l'élément renvoyé (en faisant `items[i]=null`), l'utilisation de la mémoire au moment des deux instantanés ne sera plus du tout la même (c'est l'une des rares fois où il est nécessaire d'affecter explicitement `null` à une référence ; la plupart du temps, cela ne sert à rien et peut même poser des problèmes [EJ Item 5]).

12.1.5 Utilisation des fonctions de rappel

Les fonctions de rappel vers du code fourni par le client facilitent l'écriture des cas de tests ; elles sont souvent placées à des points précis du cycle de vie d'un objet où elles constituent une bonne opportunité de vérifier les invariants. `ThreadPoolExecutor`, par exemple, fait des appels à des tâches `Runnable` et à `ThreadFactory`.

Tester un pool de threads implique de tester un certain nombre d'éléments de sa politique d'exécution : que les threads supplémentaires sont créés quand ils doivent l'être, mais pas lorsqu'il ne le faut pas ; que les threads inactifs sont supprimés lorsqu'ils doivent l'être, etc. La construction d'une suite de tests complète qui couvre toutes les possibilités représente un gros travail, mais la plupart d'entre elles peuvent être testées séparément assez simplement.

Nous pouvons mettre en place la création des threads à l'aide d'une fabrique de threads personnalisée. La classe `TestingThreadFactory` du Listing 12.8, par exemple, gère un compteur des threads créés afin que les cas de tests puissent vérifier ce nombre au cours de leur exécution. Cette classe pourrait être étendue pour renvoyer un `Thread` personnalisé qui enregistre également sa fin, afin que les cas de tests puissent contrôler que les threads sont bien supprimés conformément à la politique d'exécution.

Listing 12.8 : Fabrique de threads pour tester `ThreadPoolExecutor`.

```
class TestingThreadFactory implements ThreadFactory {
    public final AtomicInteger numCreated = new AtomicInteger();
    private final ThreadFactory factory
        = Executors.defaultThreadFactory();

    public Thread newThread(Runnable r) {
        numCreated.incrementAndGet();
        return factory.newThread(r);
    }
}
```

Si la taille du pool est inférieure à la taille maximale, celui-ci devrait grandir à mesure que les besoins d'exécution augmentent. Soumettre des tâches longues rend constant le nombre de tâches qui s'exécutent pendant suffisamment longtemps pour faire quelques tests. Dans le Listing 12.9, on vérifie que le pool s'étend correctement.

Listing 12.9 : Méthode de test pour vérifier l'expansion du pool de threads.

```
public void testPoolExpansion() throws InterruptedException {
    int MAX_SIZE = 10;
    ExecutorService exec = Executors.newFixedThreadPool(MAX_SIZE, threadFactory);

    for (int i = 0; i < 10 * MAX_SIZE; i++) {
        exec.execute(new Runnable() {
            public void run() {
                try {
                    Thread.sleep(Long.MAX_VALUE);
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            }
        });
    }
    for (int i = 0;
         i < 20 && threadFactory.numCreated.get() < MAX_SIZE;
         i++)
        Thread.sleep(100);
    assertEquals(threadFactory.numCreated.get(), MAX_SIZE);
    exec.shutdownNow();
}
```

12.1.6 Production d'entrelacements supplémentaires

La plupart des erreurs potentielles des codes concurrents étant des événements peu probables, leurs tests sont un jeu de hasard, bien que l'on puisse améliorer leurs chances à l'aide de quelques techniques. Nous avons déjà mentionné comment l'utilisation de systèmes ayant moins de processeurs que de threads actifs permettait de produire plus d'entrelacement qu'un système monoprocesseur ou un système disposant de beaucoup de processeurs. De même, effectuer des tests sur plusieurs types de systèmes avec des nombres de processeurs, des systèmes d'exploitation différents et des architectures différentes permet de découvrir des problèmes qui pourraient ne pas apparaître sur tous ces systèmes.

Une astuce utile pour augmenter le nombre d'entrelacements (et donc pour explorer efficacement l'espace d'état des programmes) consiste à utiliser `Thread.yield()` pour encourager les changements de contexte au cours des opérations qui accèdent à l'état partagé (l'efficacité de cette technique dépend de la plate-forme car la JVM peut très bien traiter `Thread.yield()` comme une "non-opération" [JLS 17.9] ; l'emploi d'un `sleep()` court mais non nul serait plus lent mais plus fiable). La méthode présentée dans le Listing 12.10 transfère des fonds d'un compte vers un autre ; les invariants comme "la somme de tous les comptes doit être égale à zéro" ne sont pas vérifiés entre les deux opérations de modification. En appelant `yield()` au milieu d'une opération, on peut parfois déclencher des bogues sensibles au timing dans du code qui ne synchronise pas correctement ses accès à l'état partagé. L'inconvénient de l'ajout de ces appels pour les tests et de leur suppression pour la version de production peut être atténué en utilisant des outils de "programmation orientée aspects".

Listing 12.10 : Utilisation de Thread.yield() pour produire plus d'entrelacements

```
public synchronized void transferCredits(Account from,
                                         Account to,
                                         int amount) {
    from.setBalance(from.getBalance() - amount);
    if (random.nextInt(1000) > THRESHOLD)
        Thread.yield();
    to.setBalance(to.getBalance() + amount);
}
```

12.2 Tests de performances

Les tests de performances sont souvent des versions améliorées des tests fonctionnels et il est presque toujours intéressant d'inclure quelques tests fonctionnels de base dans les tests de performances afin de s'assurer que l'on ne teste pas les performances d'un code incorrect.

Bien que les tests de performances et les tests fonctionnels se recouvrent, leurs buts sont différents. Les premiers tentent de mesurer les performances de bout en bout pour des cas d'utilisation représentatifs. Il n'est pas toujours aisément de choisir un jeu de scénarios d'utilisation raisonnable ; dans l'idéal, les tests devraient refléter la façon dont les objets testés sont réellement utilisés dans l'application.

Dans certains cas, un scénario de test approprié s'impose clairement ; les tampons bornés étant presque toujours utilisés dans des conceptions producteur-consommateur. Par exemple, il est raisonnable de mesurer le début des producteurs qui fournissent les données aux consommateurs. Nous pouvons aisément étendre `PutTakeTest` pour en faire un test de performances pour ce scénario.

Un second but classique des tests de performances est de choisir empiriquement des valeurs pour différentes mesures – le nombre de threads, la capacité d'un tampon, etc. Bien que ces valeurs puissent dépendre des caractéristiques de la plate-forme (le type, voire la version du processeur, le nombre de processeurs ou la taille de la mémoire) et donc nécessiter une configuration en conséquence, des choix raisonnables pour ces valeurs fonctionnent souvent correctement sur un grand nombre de systèmes.

12.2.1 Extension de PutTakeTest pour ajouter un timing

L'extension principale à `PutTakeTest` concerne la mesure du temps d'exécution d'un test. Au lieu d'essayer de le mesurer pour une seule opération, nous obtiendrons une valeur plus précise en calculant le temps du test complet et en le divisant par le nombre d'opérations afin d'obtenir un temps par opération. Nous avons déjà utilisé un objet `CyclicBarrier` pour lancer et stopper les threads, aussi pouvons-nous continuer et utiliser une action de barrière pour noter les temps de début et de fin, comme dans le Listing 12.11.

Nous modifions l'initialisation de la barrière pour qu'elle utilise cette action en nous servant du constructeur de CyclicBarrier, qui prend en paramètre une action de barrière.

Listing 12.11 : Mesure du temps à l'aide d'une barrière.

```
this.timer = new BarrierTimer();
this.barrier = new CyclicBarrier(npairs * 2 + 1, timer);
public class BarrierTimer implements Runnable {
    private boolean started;
    private long startTime, endTime;

    public synchronized void run() {
        long t = System.nanoTime();
        if (!started) {
            started = true;
            startTime = t;
        } else
            endTime = t;
    }
    public synchronized void clear() {
        started = false;
    }
    public synchronized long getTime() {
        return endTime - startTime;
    }
}
```

La méthode de test modifiée qui mesure le temps à l'aide de la barrière est présentée dans le Listing 12.12.

Listing 12.12 : Test avec mesure du temps à l'aide d'une barrière.

```
public void test() {
    try {
        timer.clear();
        for (int i = 0; i < nPairs; i++) {
            pool.execute(new Producer());
            pool.execute(new Consumer());
        }
        barrier.await();
        barrier.await();
        long nsPerItem = timer.getTime() / (nPairs * (long)nTrials);
        System.out.print("Throughput: " + nsPerItem + " ns/item");
        assertEquals(putSum.get(), takeSum.get());
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

Nous pouvons apprendre plusieurs choses de l'exécution de `TimedPutTakeTest`. La première concerne le débit de l'opération de passage du producteur au consommateur pour les différentes combinaisons de paramètres ; la deuxième est l'adaptation du tampon borné aux différents nombres de threads ; la troisième est la façon dont nous pourrions choisir la taille du tampon. Répondre à ces questions nécessitant de lancer le test avec différentes combinaisons de paramètres, nous avons donc besoin d'un pilote de test comme celui du Listing 12.13.

Listing 12.13 : Programme pilote pour TimedPutTakeTest.

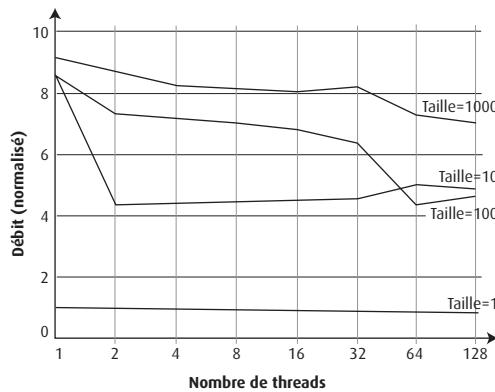
```

public static void main(String[] args) throws Exception {
    int tpt = 100000; // essais par thread
    for (int cap = 1; cap <= 1000; cap *= 10) {
        System.out.println("Capacity: " + cap);
        for (int pairs = 1; pairs <= 128; pairs *= 2) {
            TimedPutTakeTest t = new TimedPutTakeTest (cap, pairs, tpt);
            System.out.print("Pairs: " + pairs + "\t");
            t.test();
            System.out.print("\t");
            Thread.sleep(1000);
            t.test();
            System.out.println();
            Thread.sleep(1000);
        }
    }
    pool.shutdown();
}

```

La Figure 12.1 montre les résultats sur une machine à quatre processeurs, avec des capacités de tampon de 1, 10, 100 et 1 000. On constate immédiatement qu'on obtient un débit très faible avec un tampon à un seul élément car chaque thread ne peut progresser que très légèrement avant de se bloquer et d'attendre un autre thread. L'augmentation de la taille du tampon à 10 améliore énormément le débit, alors que les capacités supérieures produisent des résultats qui vont en diminuant.

Figure 12.1
TimedPutTakeTest avec différentes capacités de tampon.



Il peut sembler étonnant qu'ajouter beaucoup plus de threads ne dégrade que légèrement les performances. La raison de ce phénomène est difficile à comprendre uniquement à partir des données, mais bien plus compréhensible lorsque l'on utilise un outil comme perfbar pour mesurer les performances des processeurs pendant l'exécution du test : même avec de nombreux threads, il n'y a pas beaucoup de calcul et la plupart du temps est passé à bloquer et à débloquer les threads. Il y a donc beaucoup de processeurs inactifs pour plus de threads qui font la même chose sans pénaliser énormément les performances.

Cependant, ces résultats ne devraient pas vous amener à conclure que l'on peut toujours ajouter plus de threads à un programme producteur-consommateur utilisant un tampon borné. Ce test est assez artificiel dans sa simulation de l'application : les producteurs ne font quasiment rien pour produire l'élément placé dans la file et les consommateurs n'en font presque rien non plus. Dans une véritable application, les threads effectueraient des traitements plus complexes pour produire et consommer ces éléments et cette inactivité disparaîtrait, ce qui aurait un impact non négligeable sur les résultats. L'intérêt principal de ce test est de mesurer les contraintes que le passage producteur-consommateur *via* le tampon borné impose au débit global.

12.2.2 Comparaison de plusieurs algorithmes

Bien que `BoundedBuffer` soit une implémentation assez solide dont les performances sont relativement correctes, il n'y a aucune comparaison avec `ArrayBlockingQueue` ou `LinkedBlockingQueue` (ce qui explique pourquoi cet algorithme de tampon n'a pas été choisi pour les classes de la bibliothèque). Les algorithmes de `java.util.concurrent` ont été choisis et adaptés – en partie à l'aide de tests comme ceux que nous décrivons – pour être aussi efficaces que possible tout en offrant un grand nombre de fonctionnalités¹. La principale raison pour laquelle les performances de `BoundedBuffer` ne sont pas satisfaisantes est que `put()` et `take()` comprennent des opérations qui peuvent impliquer une compétition – prendre un sémaphore ou un verrou, libérer un sémaphore. Les autres approches contiennent moins de points susceptibles de donner lieu à une compétition avec un autre thread.

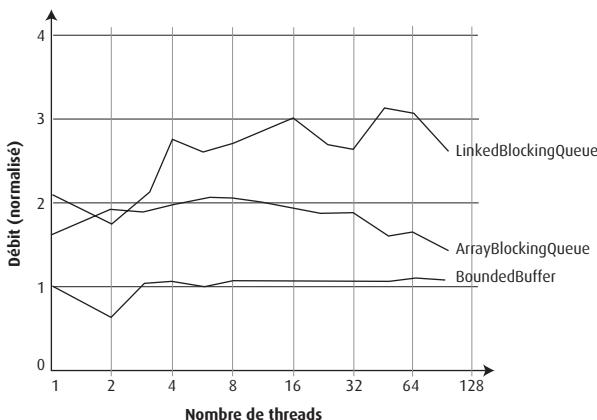
La Figure 12.2 compare les débits de trois classes utilisant des tampons de 256 éléments en utilisant une variante de `TimedPutTakeTest`. Les résultats semblent suggérer que `LinkedBlockingQueue` s'adapte mieux que `ArrayBlockingQueue`. Ceci peut sembler curieux au premier abord : une liste chaînée doit allouer un objet nœud pour chaque insertion et semble donc devoir faire plus de travail qu'une file reposant sur un tableau. Cependant, bien qu'il y ait plus d'allocations et de coûts dus au ramasse-miettes, une liste chaînée autorise plus d'accès concurrents par `puts()` et `take()` qu'une file reposant sur un tableau car les algorithmes de la liste chaînée permettent de modifier simultanément sa tête et sa queue.

L'allocation étant généralement locale au thread, les algorithmes qui permettent de réduire la compétition en faisant plus d'allocation s'adaptent généralement mieux (c'est un autre cas où l'intuition concernant les performances va à l'encontre des besoins de l'adaptabilité).

1. Vous devriez pouvoir faire mieux si vous êtes un expert en concurrence et que vous soyez prêt à abandonner quelques-unes des fonctionnalités offertes.

Figure 12.2

Comparaison des implémentations des files bloquantes.



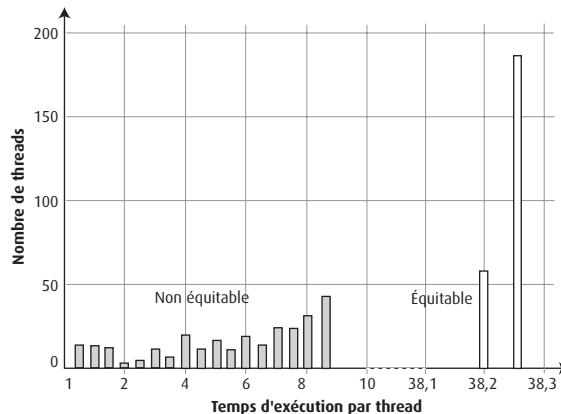
12.2.3 Mesure de la réactivité

Pour l'instant, nous nous sommes intéressés à la mesure du débit, car c'est généralement le critère de performance le plus important pour les programmes concurrents. Parfois, cependant, on préfère connaître le temps que mettra une action particulière à s'exécuter et, dans ce cas, nous voulons mesurer la variance du temps de service. Il est parfois plus intéressant d'autoriser un temps de service moyen plus long s'il permet d'obtenir une variance plus faible ; la prévisibilité est également une caractéristique importante de la performance. Mesurer la variance permet d'estimer les réponses aux questions concernant la qualité du service, comme "quel est le pourcentage d'opérations qui réussiront en 100 millisecondes ?". Les histogrammes des temps d'exécution des tâches sont généralement le meilleur moyen de visualiser les écarts des temps de service. Les variances sont juste un petit peu plus compliquées à mesurer que les moyennes car il faut mémoriser les temps d'exécution de chaque tâche, ainsi que le temps total. La granularité du timer pouvant être un facteur important lors de ces mesures (une tâche peut s'exécuter en un temps inférieur ou très proche du plus petit "intervalle de temps", ce qui déformerait la mesure de sa durée), nous pouvons plutôt évaluer le temps d'exécution de petits groupes d'opérations `put()` et `take()`, afin d'éviter ces artefacts.

La Figure 12.3 montre les temps d'exécution par tâche d'une variante de `TimedPutTakeTest` utilisant un tampon de 1 000 éléments, dans laquelle chacune des 256 tâches concurrentes ne parcourt que 1 000 éléments avec des sémaphores non équitables (barres grises) et équitables (barres blanches) – la différence entre les versions équitables et non équitables des verrous et des sémaphores sera expliquée dans la section 13.3. Les temps d'exécution avec des sémaphores non équitables varient de 104 à 8 714 millisecondes, soit d'un facteur quatre-vingts. Il peut être réduit en imposant plus d'équité dans le contrôle de la concurrence, ce qui est assez facile à faire dans `BoundedBuffer` en initialisant les sémaphores en mode équitable.

Figure 12.3

Histogramme des temps d'exécution de TimedPutTakeTest avec des sémaphores par défaut (non équitables) et des sémaphores équitables.

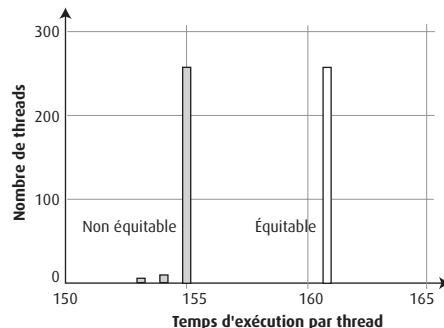


Comme le montre la Figure 12.3, cela réduit beaucoup la variance puisque l'intervalle ne va plus que de 38 194 à 38 207 millisecondes mais cela diminue malheureusement énormément le débit (un test plus long avec des types de tâches plus classiques montrerait sûrement une réduction encore plus importante).

Nous avons vu précédemment que des tailles de tampon très petites provoquaient de nombreux changements de contexte et un mauvais débit, même en mode non équitable, car quasiment chaque opération impliquait un changement de contexte. Pour se rendre compte que le coût de l'équité provient essentiellement du blocage des threads, nous pouvons relancer le test avec une taille de tampon de un et constater que les sémaphores non équitables ont maintenant des performances comparables à celles des sémaphores équitables. La Figure 12.4 montre que, dans ce cas, l'équité ne donne pas une moyenne beaucoup plus mauvaise ni une meilleure variance.

Figure 12.4

Histogrammes des temps d'exécution pour TimedPut - TakeTest avec des tampons d'un seul élément.



Par conséquent, à moins que les threads ne se bloquent de toute façon continuellement à cause d'une synchronisation trop stricte, les sémaphores non équitables fournissent un bien meilleur débit alors que les sémaphores équitables produisent une variance plus

faible. Ces résultats étant si différents, la classe `Semaphore` force ses clients à décider du facteur qu'ils souhaitent optimiser.

12.3 Pièges des tests de performance

En théorie, le développement des tests de performance est simple – il s'agit de trouver un scénario d'utilisation typique, d'écrire un programme qui exécute plusieurs fois ce scénario et de mesurer les temps d'exécution. En pratique, cependant, il faut prendre garde à un certain nombre de pièges de programmation qui empêchent ces tests de produire des résultats significatifs.

12.3.1 Ramasse-miettes

Le timing du ramasse-miettes étant imprévisible, il est tout à fait possible qu'il se lance pendant l'exécution d'un test chronométré. Si un programme de test effectue N itérations et ne déclenche pas le ramasse-miettes alors que l'itération $N + 1$ le déclenche, une petite variation de la taille du test peut avoir un effet important (mais fallacieux) sur le temps mesuré par itération.

Il existe deux stratégies pour empêcher le ramasse-miettes de biaiser vos résultats. La première consiste à s'assurer qu'il ne se lancera jamais au cours du test (en invoquant la JVM avec l'option `-verbose:gc` pour le vérifier) ; vous pouvez aussi vous assurer que le ramasse-miettes s'exécutera un certain nombre de fois au cours du test, afin que le programme de test puisse tenir compte de son coût. La seconde est souvent meilleure – elle nécessite un test plus long et elle reflète donc souvent mieux les performances réelles.

La plupart des applications producteur-consommateur impliquent un certain nombre d'allocations et de passage du ramasse-miettes – les producteurs allouent de nouveaux objets qui sont utilisés et supprimés par les consommateurs. En exécutant le test du tampon borné suffisamment longtemps pour attirer plusieurs fois le ramasse-miettes, on obtient des résultats plus précis.

12.3.2 Compilation dynamique

Écrire et interpréter les tests de performance pour des langages compilés dynamiquement comme Java est bien plus difficile qu'avec les langages compilés statiquement, comme C ou C++. La JVM Hotspot (et les autres JVM actuelles) utilise en effet une combinaison d'interprétation du pseudo-code et de compilation dynamique. La première fois qu'une classe est chargée, la JVM l'exécute en interprétant le pseudo-code. À un moment donné, si une méthode est exécutée suffisamment souvent, le compilateur dynamique l'extrait et la traduit en code machine ; lorsque la compilation se termine, il passe de l'interprétation à une exécution directe.

Le timing de la compilation est imprévisible. Vos tests ne devraient s'exécuter qu'après que tout le code eut été compilé : il n'y a aucun intérêt à mesurer la vitesse du code interprété puisque la plupart des programmes s'exécutent suffisamment longtemps pour que tout leur code finisse par être compilé. Autoriser le compilateur à s'exécuter pendant que l'on mesure du temps d'exécution peut donc fausser les résultats des tests de deux façons : la compilation consomme des ressources processeur et la mesure du temps d'exécution d'une combinaison de code compilé et interprété ne produit pas un résultat significatif. La Figure 12.5 montre comment ces résultats peuvent être faussés. Les trois lignes représentent l'exécution du même nombre d'instructions : la ligne A concerne une exécution entièrement interprétée, la ligne B, une exécution avec une compilation survenant en plein milieu et la ligne C, une exécution où la compilation a eu lieu plus tôt. On remarque donc que la compilation a une influence importante sur le temps d'exécution¹.

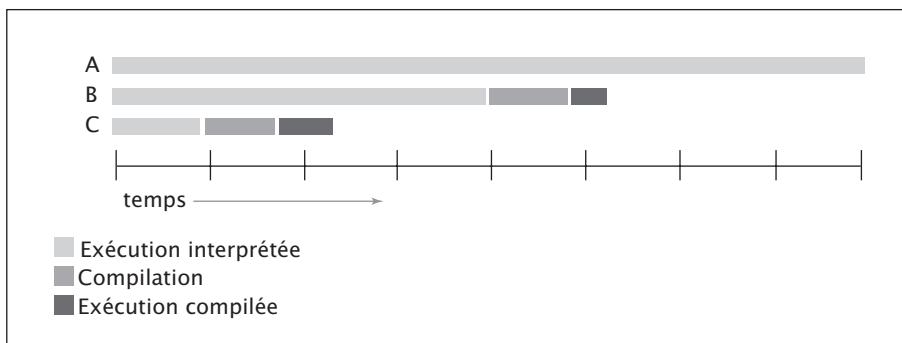


Figure 12.5
Résultats faussés par une compilation dynamique.

Le code peut également être décompilé (pour revenir à une exécution interprétée) et recompilé pour différentes raisons : pour, par exemple, charger une classe qui invalide des suppositions faites par des compilations antérieures ou afin de rassembler suffisamment de données de profilage pour décider qu'une portion de code devrait être recompilée avec des optimisations différentes.

Un moyen d'empêcher la compilation de fausser les résultats consiste à exécuter le programme pendant un certain temps (au moins plusieurs minutes) afin que la compilation et l'exécution interprétée ne représentent qu'une petite fraction du temps total. On peut également utiliser une exécution "de préchauffage" dans laquelle le code est suffisamment exécuté pour être compilé lorsqu'on lance la véritable mesure. Avec Hotspot,

1. La JVM peut choisir d'effectuer la compilation dans le thread de l'application ou dans un thread en arrière-plan ; les temps d'exécution seront affectés de façon différente.

l’option `-XX:+PrintCompilation` affiche un message lorsque la compilation dynamique intervient : on peut donc vérifier qu’elle a lieu avant les mesures et non pendant.

Lancer le même test plusieurs fois dans la même instance de la JVM permet de valider la méthodologie du test. Le premier groupe de résultat devrait être éliminé comme données “de préchauffage” et la présence de résultats incohérents dans les groupes restants signifie que le test devrait être réexaminé afin de comprendre pourquoi ces résultats ne sont pas identiques.

La JVM utilise différents threads en arrière-plan pour ses travaux de maintenance. Lorsque l’on mesure plusieurs activités de calcul intense non apparentées dans le même test, il est conseillé de placer des pauses explicites entre les mesures : cela permet à la JVM de combler son retard par rapport aux tâches de fond avec un minimum d’interférences de la part des tâches mesurées (lorsque l’on évalue plusieurs activités apparentées – plusieurs exécutions du même test, par exemple –, exclure les tâches en arrière-plan de la JVM peut donner des résultats exagérément optimistes).

12.3.3 Échantillonnage irréaliste de portions de code

Les compilateurs dynamiques utilisent des informations de profilage pour faciliter l’optimisation du code compilé. La JVM peut utiliser des informations spécifiques à l’exécution pour produire un meilleur code, ce qui signifie que la compilation de la méthode *M* dans un programme peut produire un code différent avec un autre programme. Dans certains cas, la JVM peut effectuer des optimisations en fonction de suppositions qui ne peuvent être que temporaires et les supprimer ensuite en invalidant le code compilé si ces suppositions ne sont plus vérifiées¹.

Il est donc important que les programmes de test se rapprochent de l’utilisation d’une application typique, mais également qu’ils utilisent le code approximativement comme le ferait une telle application. Sinon un compilateur dynamique pourrait effectuer des optimisations spécifiques à un programme de test purement monothread qui ne pourraient pas s’appliquer à des applications réelles utilisant un parallélisme au moins occasionnel. Les tests de performances multithreads ne devraient donc pas être mélangés aux tests monothreads, même si vous ne voulez mesurer que les performances monothreads (ce problème ne se pose pas avec `TimedPutTakeTest` car même le plus petit des cas testés utilise deux threads).

12.3.4 Degrés de compétition irréalistes

Les applications concurrentes ont tendance à entrelacer deux types de travaux très différents : l’accès aux données partagées – comme la récupération de la tâche suivante à partir

1. La JVM peut, par exemple, utiliser une transformation pour convertir un appel de méthode virtuelle en appel direct si aucune des classes actuellement chargées ne redéfinit cette méthode ; il invalidera ce code compilé lorsqu’une classe redéfinissant la méthode sera chargée par la suite.

d'une file d'attente partagée – et les calculs locaux aux threads – l'exécution de la tâche en supposant qu'elle n'accède pas elle-même aux données partagées. Selon les proportions relatives de ces deux types de travaux, l'application subira des niveaux de compétition différents et aura des performances et une adaptabilité différentes.

Il n'y aura quasiment pas de compétition si N threads récupèrent des tâches à partir d'une file d'attente partagée et les exécutent, et si ces tâches font des calculs intensifs qui durent longtemps (sans trop accéder à des données communes) ; le débit sera dominé par la disponibilité des ressources processeur. Si, en revanche, les tâches sont très courtes, la compétition pour la file d'attente sera très importante et le débit sera dominé par le coût de la synchronisation.

Pour obtenir des résultats réalistes, les tests de performances concurrents devraient essayer de se rapprocher du calcul local au thread effectué par une application typique tout en étudiant la coordination entre les threads ; sinon il est facile d'obtenir des conclusions hasardeuses sur les endroits qui gênent la concurrence. Dans la section 11.5 nous avons vu que, pour les classes qui utilisaient des verrous (les implémentations de Map synchronisées, par exemple), le fait que le verrou soit très ou peu disputé pouvait avoir des répercussions énormes sur le débit. Les tests de cette section ne font que maltraiter l'objet Map : même avec deux threads, toutes les tentatives d'y accéder donnent lieu à une compétition. Cependant, si une application effectuait un grand nombre de calculs locaux aux threads à chaque fois qu'elle accède à la structure de données partagée, le niveau de compétition pourrait être suffisamment faible pour autoriser de bonnes performances.

De ce point de vue, `TimedPutTakeTest` peut être un mauvais modèle pour certaines applications. Les threads ne faisant pas grand-chose, le débit est en effet dominé par le coût de leur coordination, ce qui n'est pas nécessairement représentatif de toutes les applications qui échangent des données entre producteurs et consommateurs *via* des tampons bornés.

12.3.5 Élimination du code mort

L'un des défis à relever pour écrire de bons programmes de tests des performances (quel que soit le langage) consiste à s'accommoder des compilateurs qui repèrent et éliminent le code mort – le code qui n'a aucun effet sur le résultat. Les tests ne calculant souvent rien, ils sont en effet une cible de choix pour l'optimiseur. La plupart du temps, il est souhaitable qu'un optimiseur supprime le code mort d'un programme mais cela pose problème dans le cas des tests puisque l'on mesure alors moins de code que prévu. Avec un peu de chance, l'optimiseur peut même supprimer tout le programme et il sera alors évident que vos données sont boguées. Si vous n'avez pas de chance, l'élimination du code mort ne fera qu'accélérer le programme d'un facteur qui pourrait s'expliquer par d'autres moyens.

Bien que l'élimination du code mort pose également problème dans les tests de performances compilés statiquement, il est bien plus facile de détecter que le compilateur a

éliminé une bonne partie du programme de test puisque l'on peut examiner le code machine et constater qu'il en manque un bout. Avec les langages compilés dynamiquement, cette vérification est moins facile.

De nombreux micro-tests de performances se comportent "mieux" lorsqu'ils sont exécutés avec l'option `-server` de Hotspot qu'avec `-client` : pas simplement parce que le compilateur peut alors produire du code plus efficace, mais également parce qu'il optimise mieux le code mort. Malheureusement, cette optimisation qui permet de réduire le code d'un test ne fera pas aussi bien avec un code qui effectue vraiment des traitements. Cependant, nous vous conseillons quand même d'utiliser `-server` plutôt que `-client` sur les systèmes multiprocesseurs (que ce soit pour les programmes de test ou de production) : il suffit simplement d'écrire les tests pour qu'ils ne soient pas sujets à l'élimination du code mort.

Écrire des tests de performances efficaces nécessite de tromper l'optimisateur afin qu'il ne considère pas le programme de test comme du code mort. Ceci implique d'utiliser quelque part dans le programme tous les résultats qui ont été calculés – d'une façon qui ne nécessite ni synchronisation ni calculs trop lourds.

Dans `PutTakeTest`, nous calculons la somme de contrôle des éléments ajoutés et supprimés de la file d'attente et nous les combinons à travers tous les threads : ce traitement pourrait être supprimé par l'optimisateur si nous n'utilisons pas vraiment cette somme de contrôle. Il se trouve que nous pensions l'utiliser pour vérifier la justesse de l'algorithme mais, pour s'assurer que n'importe quelle valeur est utilisée, il suffit de l'afficher. Cependant, il est préférable d'éviter les opérations d'E/S pendant l'exécution d'un test, afin de ne pas perturber la mesure du temps d'exécution. Une astuce simple pour éviter à moindre coût qu'un calcul soit supprimé par l'optimisateur consiste à calculer le code de hachage du champ d'un objet dérivé, de le comparer à une valeur quelconque (la valeur courante de `System.nanoTime()`, par exemple) et d'afficher un message inutile en cas d'égalité :

```
if (foo.x.hashCode() == System.nanoTime())
    System.out.print(" ");
```

Cette comparaison réussira rarement et, même dans ce cas, elle n'aura pour effet que d'insérer un espace inoffensif dans la sortie (la méthode `print()` plaçant ce caractère dans un tampon jusqu'à ce que `println()` soit appelée, il n'y aura donc pas de véritable E/S, même si les deux valeurs sont égales).

Non seulement chaque résultat calculé doit être utilisé, mais les résultats doivent également ne pas être prévisibles. Sinon un compilateur dynamique intelligent pourrait remplacer les opérations par les résultats précalculés. Nous avons réglé ce problème lors de la construction de `PutTakeTest`, mais tout programme de test dont les entrées sont des données statiques est vulnérable à ce type d'optimisation.

12.4 Approches de tests complémentaires

Même si nous aimerais croire qu'un programme de test efficace "trouvera tous les bogues", il s'agit d'un vœu pieux. La NASA consacre aux tests plus de ressources techniques (on considère qu'il y a vingt testeurs pour un programmeur) que n'importe quelle société commerciale ne pourrait se le permettre : pourtant, le code qu'elle produit n'est pas exempt de défauts. Pour des programmes complexes, aucun volume de test ne peut trouver toutes les erreurs. Le but des tests n'est pas tant de *trouver les erreurs* que d'*augmenter l'espoir que le code fonctionnera comme prévu*. Comme il est irréaliste de croire que l'on peut trouver tous les bogues, le but des *plans qualité* devrait être d'obtenir la plus grande confiance possible à partir des tests disponibles. Comme un programme concurrent peut contenir plus de problèmes qu'un programme séquentiel, il faut donc plus de tests pour obtenir le même niveau de confiance. Pour l'instant, nous nous sommes surtout intéressés aux techniques permettant de construire des tests unitaires et des performances efficaces. Les tests sont essentiels pour avoir confiance dans le comportement des classes concurrentes, mais ils ne constituent qu'une des méthodes des plans qualité que vous utilisez. D'autres méthodologies de la qualité sont plus efficaces pour trouver certains types de défaut et moins pour en trouver d'autres. En utilisant des méthodes de tests complémentaires, comme la relecture du code et l'analyse statique, vous pouvez obtenir une confiance supérieure à celle que vous auriez avec une approche simple, quelle qu'elle soit.

12.4.1 Relecture du code

Aussi efficaces et importants que soient les tests unitaires et les tests de stress pour trouver les bogues de concurrence, ils ne peuvent pas se substituer à une relecture rigoureuse du code par plusieurs personnes (inversement, la relecture du code ne peut pas non plus se substituer aux tests). Vous pouvez et devez concevoir des tests pour augmenter au maximum les chances de découvrir les erreurs de sécurité vis-à-vis des threads et les lancer régulièrement, mais vous ne devez pas négliger de faire relire soigneusement tout code concurrent par un autre développeur que son auteur. Même les experts de la programmation concurrente font des erreurs ; vous avez donc toujours intérêt à prendre le temps de faire relire votre code par quelqu'un d'autre. Les experts en concurrence sont meilleurs que la plupart des programmes de test lorsqu'il s'agit de découvrir des situations de compétition (en outre, les détails de l'implémentation de la JVM ou les modèles mémoire utilisés par le processeur peuvent empêcher des bogues d'apparaître sur certaines configurations logicielles ou matérielles). La relecture du code a également d'autres bénéfices : en plus de trouver des erreurs, elle améliore souvent la qualité des commentaires qui décrivent les détails d'implémentation, ce qui réduit les coûts et les risques dus à la maintenance du code.

12.4.2 Outils d'analyse statiques

Au moment où ce livre est écrit, de nouveaux *outils d'analyse statiques* voient le jour régulièrement pour compléter les tests formels et la relecture du code. L'analyse statique du code consiste à analyser le code sans l'exécuter et les outils d'audit du code peuvent analyser les classes pour rechercher des *types d'erreurs classiques* dans leurs instances. Des outils comme le programme open-source FindBugs¹ contiennent ainsi des *détecteurs de bogues* qui savent reconnaître de nombreuses erreurs de codage classiques, dont beaucoup peuvent aisément passer à travers les mailles des tests et de la relecture du code.

Les outils d'analyse statique produisent une liste d'avertissements qui doit être examinée manuellement pour déterminer s'il s'agit de véritables erreurs. Des utilitaires historiques comme lint produisaient tellement de faux avertissements qu'ils effrayaient les développeurs, mais les outils comme FindBugs ont été conçus pour en produire moins. Bien qu'ils soient parfois assez rudimentaires (notamment leur intégration dans les outils de développement et dans le cycle de vie), leur efficacité mérite qu'ils soient ajoutés aux processus de test.

Actuellement, FindBugs contient les détecteurs suivants pour trouver les erreurs liées à la concurrence, bien que d'autres soient sans cesse ajoutés :

- **Synchronisation incohérente.** De nombreux objets utilisent une politique de synchronisation consistant à protéger toutes les variables par le verrou interne de l'objet. Si l'on accède souvent à un champ alors que l'on ne détient pas toujours ce verrou, ceci peut indiquer que l'on ne respecte pas la politique de synchronisation.

Les outils d'analyse doivent supposer la politique de synchronisation car les classes Java ne disposent pas d'un moyen de spécifier formellement la concurrence. Dans le futur, si des annotations comme @GuardedBy sont standardisées, ces outils pourront les interpréter au lieu de deviner les relations entre les variables et les verrous, ce qui améliorera la qualité de l'analyse.

- **Appel de Thread.run().** Thread implémente Runnable et dispose donc d'une méthode run(). Il s'agit presque toujours d'une erreur de l'appeler directement : généralement, le programmeur voulait appeler Thread.start().
- **Verrou non relâché.** À la différence des verrous internes, les verrous explicites (voir Chapitre 13) ne sont pas automatiquement relâchés en sortant de la portée dans laquelle ils ont été pris. L'idiome classique consiste à relâcher le verrou dans un bloc finally ; sinon le verrou pourrait rester verrouillé en cas d'exception.
- **Bloc synchronized vide.** Bien que les blocs synchronized vides aient une signification pour le modèle mémoire de Java, ils sont souvent utilisés incorrectement et il existe généralement de meilleures solutions pour résoudre le problème.

1. Voir <http://findbugs.sourceforge.net>.

- **Verrouillage contrôlé deux fois.** Le verrouillage contrôlé deux fois est un idiomme erroné utilisé pour réduire le surcoût dû à la synchronisation dans les initialisations paresseuses (voir la section 16.2.4). Il implique de lire un champ modifiable partagé sans synchronisation appropriée.
- **Lancement d'un thread à partir d'un constructeur.** Lancer un thread à partir d'un constructeur risque de poser des problèmes d'héritage et peut permettre à la référence `this` de s'échapper du constructeur.
- **Erreurs de notification.** Les méthodes `notify()` et `notifyAll()` indiquent que l'état d'un objet a été modifié pour débloquer des threads qui attendent sur la file de condition appropriée. Ces méthodes ne doivent être appelées que lorsque l'état associé à cette file a été modifié. Un bloc `synchronized` qui les appelle sans modifier un état représente sûrement une erreur (voir Chapitre 14).
- **Erreurs d'attente sur une condition.** Lorsque l'on attend dans une file associée à une condition, `Object.wait()` ou `Condition.await()` devraient toujours être appelées dans une boucle en détenant le verrou approprié et après avoir testé un prédictat concernant l'état (voir Chapitre 14). Appeler ces méthodes sans détenir de verrou en dehors d'une boucle ou sans tester d'état est presque certainement une erreur.
- **Mauvaise utilisation de Lock et Condition.** L'utilisation d'un objet `Lock` comme paramètre verrou d'un bloc `synchronized` est probablement une erreur de frappe, tout comme appeler `Condition.wait()` au lieu de `await()` (même si cette dernière serait probablement détectée lors des tests puisqu'elle lancerait `IllegalMonitorStateException` lors de son premier appel).
- **Mise en sommeil ou en attente pendant la détention d'un verrou.** L'appel de `Thread.sleep()` alors que l'on détient un verrou peut empêcher pendant longtemps d'autres threads de progresser et constitue donc un risque sérieux pour la vivacité du programme. L'appel de `Object.wait()` ou `Condition.await()` avec deux verrous pose le même type de problème.
- **Boucles inutiles.** Le code qui ne fait rien à part tester si un champ a une valeur donnée (attente active) peut gaspiller du temps processeur et, si ce champ n'est pas volatile, n'est pas certain de se terminer. Les loquets ou les attentes de conditions sont généralement préférables pour attendre qu'une transition d'état ait lieu.

12.4.3 Techniques de tests orientées aspects

Actuellement, les techniques de programmation orientée aspects (POA) sont peu utilisées en programmation concurrente car la plupart des outils de POA ne gèrent pas encore les points de synchronisation. Cependant, la POA peut servir à tester les invariants ou certains aspects de la conformité aux politiques de synchronisation. (Laddad, 2003), par exemple, donne un exemple d'utilisation d'un aspect pour envelopper tous les appels aux méthodes non thread-safe de Swing avec l'assertion que l'appel a lieu dans le thread

des événements. Comme elle ne nécessite aucune modification du code, cette technique est simple à appliquer et peut mettre en évidence des erreurs subtiles de publication et de confinement aux threads.

12.4.4 Profileurs et outils de surveillance

La plupart des outils de profilage du commerce savent gérer les threads. Leurs fonctionnalités et leur efficacité varient mais ils offrent souvent une vision de ce que fait le programme (bien que les outils de profilage soient généralement indiscrets et puissent avoir une influence non négligeable sur le timing et l'exécution d'un programme). La plupart affichent une chronologie de chaque thread avec des couleurs différentes pour représenter leurs états (en cours d'exécution, bloqué en attente d'un verrou, bloqué en attente d'une E/S, etc.). Cet affichage permet de mettre en évidence l'utilisation des ressources processeur disponibles par le programme et, si elle est mauvaise, d'en constater la cause (de nombreux profileurs revendiquent également de pouvoir identifier les verrous qui provoquent les compétitions mais, en pratique, ces fonctionnalités ne suffisent pas à analyser le comportement du verrouillage). L'agent JMX intégré offre également quelques possibilités limitées pour surveiller le comportement des threads. La classe `ThreadInfo` contient l'état du thread courant et, s'il est bloqué, le verrou ou la condition qu'il attend. Si la fonctionnalité "surveillance de la compétition des threads" est activée (elle est désactivée par défaut à cause de son impact sur les performances), `ThreadInfo` contient également le nombre de fois où le thread s'est bloqué en attente d'un verrou ou d'une notification, ainsi que la somme cumulée du temps passé à attendre.

Résumé

Tester la justesse des programmes concurrents peut se révéler extrêmement complexe car la plupart des erreurs possibles de ces programmes sont des événements peu fréquents qui sont sensibles au timing, à la charge et à d'autres conditions difficiles à reproduire. En outre, l'infrastructure d'un test peut introduire une synchronisation supplémentaire ou des contraintes de timing qui peuvent masquer les problèmes de concurrence du code testé. Tester les performances des programmes concurrents est tout aussi difficile ; les programmes Java sont plus durs à tester que ceux écrits dans des langages compilés statiquement comme C car les mesures du temps peuvent être affectées par la compilation dynamique, le ramasse-miettes et l'optimisation adaptative.

Pour avoir les meilleures chances de trouver des bogues avant qu'ils ne surviennent en production, il faut combiner des techniques de tests traditionnelles (en évitant les pièges mentionnés ici) avec des relectures du code et des outils d'analyse automatisée. Chacune de ces techniques trouve en effet des problèmes qui n'apparaîtront probablement pas aux autres.

IV

Sujets avancés

Verrous explicites

Avant Java 5.0, les seuls mécanismes permettant de synchroniser l'accès aux données partagées étaient `synchronized` et `volatile`. Java 5.0 leur a ajouté une autre option, `ReentrantLock`. Contrairement à ce que certains ont prétendu, `ReentrantLock` ne remplace pas les verrous internes mais constitue plutôt une alternative disposant de fonctionnalités avancées lorsque ces derniers ont atteint leurs limites.

13.1 Lock et ReentrantLock

L'interface `Lock`, présentée dans le Listing 13.1, définit un certain nombre d'opérations abstraites de verrouillage. À la différence du verrouillage interne, `Lock` offre le choix entre une prise de verrou scrutable, avec délai et interruptible ; par ailleurs, toutes les opérations de verrouillage et de déverrouillage sont explicites. Les implémentations de `Lock` doivent fournir la même sémantique de visibilité de la mémoire que les verrous internes mais peuvent avoir des sémantiques de verrouillage, des algorithmes de planification, des garanties sur l'ordre des verrous et des performances différentes (`Lock.newCondition()` est présentée au Chapitre 14).

Listing 13.1 : Interface Lock.

```
public interface Lock {  
    void lock();  
    void lockInterruptibly() throws InterruptedException ;  
    boolean tryLock();  
    boolean tryLock(long timeout, TimeUnit unit)  
        throws InterruptedException ;  
    void unlock();  
    Condition newCondition();  
}
```

La classe `ReentrantLock` implémente `Lock` en fournissant les mêmes garanties d'exclusion mutuelle et de visibilité mémoire que `synchronized`. Prendre un `ReentrantLock` a la même sémantique mémoire qu'entrer dans un bloc `synchronized` et libérer un `ReentrantLock`

a la même sémantique mémoire que sortir d'un bloc `synchronized` (la visibilité mémoire est présentée dans la section 3.1 et au Chapitre 16). Comme `synchronized`, `ReentrantLock` offre une sémantique de verrous réentrants (voir la section 2.3.2). `ReentrantLock` dispose de tous les modes d'acquisition des verrous définis par `Lock` et permet de gérer de façon plus souple que `synchronized` l'indisponibilité des verrous.

Pourquoi créer un nouveau mécanisme de verrouillage si proche des verrous internes ? Parce que ces derniers fonctionnent très bien dans la plupart des situations mais souffrent de quelques limitations fonctionnelles – il est impossible d'interrompre un thread qui attend de prendre un verrou ou de tenter de prendre un verrou sans être prêt à attendre indéfiniment. En outre, les verrous internes doivent être relâchés dans le même bloc de code où ils ont été pris ; cela simplifie le codage et s'accorde bien avec le traitement des exceptions, mais cela empêche aussi de créer des disciplines de verrouillage qui ne sont pas structurées en blocs. Aucune de ces limitations ne constitue de raison d'abandonner `synchronized` mais, dans certains cas, un mécanisme de verrouillage plus souple permet d'obtenir une meilleure vivacité ou des performances supérieures.

Le Listing 13.2 présente l'utilisation canonique d'un objet `Lock`. Cet idiome est un peu plus compliqué que l'utilisation des verrous internes puisque le verrou doit être relâché dans un bloc `finally` ; sinon le verrou ne serait jamais libéré si le code protégé levait une exception. Lorsque l'on utilise des verrous, il faut aussi tenir compte de ce qui se passera si une exception survient en dehors du bloc `try` ; si l'objet peut rester dans un état incohérent, il peut être nécessaire d'ajouter des blocs `try-catch` ou `try-finally` supplémentaires (il faut toujours tenir compte des effets des exceptions lorsque l'on utilise n'importe quelle forme de verrouillage, y compris les verrous internes).

Listing 13.2 : Protection de l'état d'un objet avec `ReentrantLock`.

```
Lock lock = new ReentrantLock();
...
lock.lock();
try {
    // Modification de l'état de l'objet
    // capture les exceptions et restaure les invariants si nécessaire
} finally {
    lock.unlock();
}
```

Oublier d'utiliser `finally` pour relâcher un `Lock` est une bombe à retardement. Lorsqu'elle explosera, vous aurez beaucoup de mal à retrouver son origine car il n'y aura aucune information sur où et quand le `Lock` aurait dû être relâché. C'est l'une des raisons pour lesquelles il ne faut pas utiliser `ReentrantLock` comme un remplacement systématique de `synchronized` : il est plus "dangereux" car il ne libère pas automatiquement le verrou lorsque le contrôle quitte le bloc protégé. Bien qu'il ne soit pas si difficile que cela de se rappeler de libérer le verrou dans un bloc `finally`, il reste possible de l'oublier¹.

1. FindBugs dispose d'un détecteur "verrou non libéré" permettant d'identifier les cas où un `Lock` n'est pas relâché dans toutes les parties du code en dehors du bloc dans lequel il a été pris.

13.1.1 Prise de verrou scrutable et avec délai

Les modes d’acquisition de verrou scrutable et avec délai offerts par `tryLock()` autorisent une gestion des erreurs plus sophistiquée qu’avec une acquisition inconditionnelle. Avec les verrous internes, un interblocage est fatal – la seule façon de s’en sortir est de relancer l’application et le seul moyen de s’en prémunir consiste à construire le programme pour empêcher un ordre d’acquisition incohérent. Le verrouillage scrutable et avec délai offre une autre possibilité : l’évitement probabiliste des interblocages.

L’utilisation d’une acquisition scrutable ou avec délai permet de reprendre le contrôle lorsque l’on ne peut pas prendre tous les verrous demandés, de relâcher ceux qui ont déjà été pris et de réessayer (ou, au moins, d’enregistrer l’échec et de passer à autre chose). Le Listing 13.3 montre un autre moyen de résoudre l’interblocage dû à l’ordre dynamique de la section 10.1.2 : il appelle `tryLock()` pour tenter de prendre les deux verrous mais revient en arrière et réessaie s’il n’a pas pu prendre les deux. Pour réduire la probabilité d’un livelock, le délai d’attente entre deux essais est formé d’une composante constante et d’une autre aléatoire. Si les verrous n’ont pas pu être pris dans le temps imparti, `transferMoney()` renvoie un code d’erreur afin que l’application puisse échouer en douceur (voir [CPJ 2.5.1.2] et [CPJ 2.5.1.3] pour plus d’exemples sur l’utilisation de la prise éventuelle de verrous afin d’éviter les interblocages).

Listing 13.3 : Utilisation de `tryLock()` pour éviter les interblocages dus à l’ordre des verrouillages.

```
public boolean transferMoney(Account fromAcct,
                             Account toAcct,
                             DollarAmount amount,
                             long timeout,
                             TimeUnit unit)
        throws InsufficientFundsException , InterruptedException {
    long fixedDelay = getFixedDelayComponentNanos(timeout, unit);
    long randMod = getRandomDelayModulusNanos(timeout, unit);
    long stopTime = System.nanoTime() + unit.toNanos(timeout);

    while (true) {
        if (fromAcct.lock.tryLock()) {
            try {
                if (toAcct.lock.tryLock()) {
                    try {
                        if (fromAcct.getBalance().compareTo(amount) < 0)
                            throw new InsufficientFundsException ();
                        else {
                            fromAcct.debit(amount);
                            toAcct.credit(amount);
                            return true;
                        }
                    } finally {
                        toAcct.lock.unlock();
                    }
                }
            } finally {
                fromAcct.lock.unlock();
            }
        }
    }
}
```

Listing 13.3 : Utilisation de `tryLock()` pour éviter les interblocages dus à l'ordre des verrouillages. (suite)

```
        fromAcct.lock.unlock();
    }
}
if (System.nanoTime() >= stopTime)
    return false;
NANOSECONDS.sleep(fixedDelay + rnd.nextLong() % randMod);
}
}
```

Les verrous avec délai facilitent également l'implémentation des activités qui doivent respecter des délais (voir la section 6.3.7). Une activité dont le temps est compté appellant une méthode bloquante peut, en effet, fournir un délai d'expiration correspondant au temps restant dans celui qui lui est alloué ; cela lui permet de se terminer plus tôt si elle ne peut pas délivrer de résultat dans le temps voulu. Avec les verrous internes, en revanche, il est impossible d'annuler la prise d'un verrou une fois qu'elle a été lancée : ces verrous considèrent donc les activités avec temps imparti comme des activités à risque.

L'exemple du portail du Listing 6.17 crée une tâche distincte pour chaque société de location de véhicule qui lui demande de passer une annonce. Cette demande d'annonce nécessite sûrement un mécanisme réseau, comme un service web, mais peut également exiger un accès exclusif à une ressource rare, comme une ligne de communication directe vers la société.

Dans la section 9.5, nous avons déjà vu comment garantir un accès sérialisé à une ressource : il suffit d'utiliser un exécuteur monothread. Une autre approche consiste à servir d'un verrou exclusif pour protéger l'accès à la ressource. Le code du Listing 13.4 tente d'envoyer un message sur une ligne de communication protégée par un Lock mais échoue en douceur s'il ne peut pas le faire dans le temps qui lui est imparti. L'appel temporisé à `tryLock()` permet en effet d'ajouter un verrou exclusif dans une activité limitée par le temps.

Listing 13.4 : Verrouillage avec temps imparti.

```
public boolean trySendOnSharedLine(String message,
                                    long timeout, TimeUnit unit)
    throws InterruptedException {
long nanosToLock = unit.toNanos(timeout) - estimatedNanosToSend (message);
if (!lock.tryLock(nanosToLock, NANOSECONDS))
    return false;
try {
    return sendOnSharedLine(message);
} finally {
    lock.unlock();
}
}
```

13.1.2 Prise de verrou interruptible

Tout comme l'acquisition temporisée d'un verrou permet d'utiliser un verrouillage exclusif avec des activités limitées dans le temps, la prise de verrou interruptible permet de faire appel au verrouillage dans des activités annulables. La section 7.1.6 avait identifié plusieurs mécanismes qui ne répondent pas aux interruptions et la prise d'un verrou interne en fait partie, or ces mécanismes bloquants non interruptibles compliquent l'implémentation des tâches annulables. La méthode `lockInterruptibly()` permet de tenter de prendre un verrou tout en restant réactif aux interruptions et son inclusion dans `Lock` évite de devoir créer une autre catégorie de mécanismes bloquants non interruptibles.

La structure canonique de l'acquisition interruptible d'un verrou est un peu plus compliquée que celle d'une prise de verrou classique car on a besoin de deux blocs `try` (si l'acquisition interruptible peut lancer `InterruptedException`, l'idiome standard `try-finally` du verrouillage fonctionne). Le Listing 13.5 utilise `lockInterruptibly()` pour implémenter la méthode `sendOnSharedLine()` du Listing 13.4 afin qu'elle puisse être appelée à partir d'une tâche annulable. La méthode `tryLock()` temporisée répond également aux interruptions et peut donc être utilisée quand on a besoin à la fois d'une acquisition de verrou interruptible et temporisée.

Listing 13.5 : Prise de verrou interruptible.

```
public boolean sendOnSharedLine(String message)
    throws InterruptedException {
    lock.lockInterruptibly ();
    try {
        return cancellableSendOnSharedLine(message);
    } finally {
        lock.unlock();
    }
}

private boolean cancellableSendOnSharedLine(String message)
    throws InterruptedException { ... }
```

13.1.3 Verrouillage non structuré en bloc

Avec les verrous internes, les paires prise et relâchement des verrous sont structurées en bloc – un verrou est toujours relâché dans le même bloc que celui où il a été pris, quelle que soit la façon dont on sort du bloc. Ce verrouillage automatique simplifie l'analyse du code et empêche d'éventuelles erreurs de codage mais on a parfois besoin d'une discipline plus souple.

Au Chapitre 11, nous avons vu comment une granularité plus fine du verrouillage permettait d'améliorer l'adaptabilité. Le découpage des verrous permet, par exemple, aux différentes chaînes de hachage d'une collection d'utiliser des verrous différents. Nous pouvons appliquer un principe similaire pour réduire la granularité du verrouillage dans une liste chaînée en utilisant un verrou différent *pour chaque nœud*, afin que plusieurs threads puissent agir séparément sur différentes parties de la liste. Le verrou pour un nœud

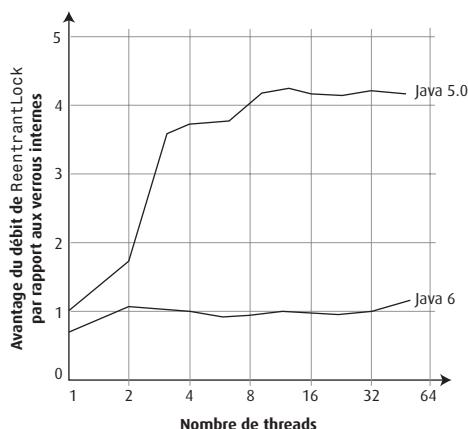
donné protège les pointeurs des liens et les données stockées dans ce nœud ; lorsque l'on parcourt ou que l'on modifie la liste, nous devons posséder le verrou sur un nœud jusqu'à obtenir celui du nœud suivant – ce n'est qu'alors que nous pouvons relâcher le verrou sur le premier nœud. Un exemple de cette technique, appelée *verrouillage de la main à la main* ou *couplage du verrouillage*, est décrit dans [CPJ 2.5.1.4].

13.2 Remarques sur les performances

Lorsque ReentrantLock a été ajouté à Java 5.0, il offrait des performances en termes de compétition bien meilleures que celles des verrous internes. Pour les primitives de synchronisation, ces performances sont la clé de l'adaptabilité : plus il y a de ressources monopolisées pour la gestion et la planification des verrous, moins il en reste pour l'application. Une meilleure implémentation des verrous fait moins d'appels systèmes, impose moins de changements de contexte et provoque moins de trafic de synchronisation sur le bus de la mémoire partagée – toutes ces opérations prennent du temps et détournent du programme les ressources de calcul disponibles. Java 6 utilise un algorithme amélioré pour gérer les verrous internes, ressemblant à celui de ReentrantLock, qui réduit considérablement l'écart d'adaptabilité. La Figure 13.1 montre les différences de performances entre les verrous internes et ReentrantLock avec Java 5.0 et sur une préversion de Java 6 s'exécutant sur un système à quatre processeurs Opteron avec Solaris. Les courbes représentent l'"accélération" de ReentrantLock par rapport aux verrous internes sur une version de la JVM. Avec Java 5.0, ReentrantLock offre un débit considérablement meilleur alors qu'avec Java 6 les deux sont relativement proches¹. Le programme de test est le même que celui de la section 11.5, mais compare cette fois-ci les débits de HashMap protégés par un verrou interne et par un ReentrantLock.

Figure 13.1

Performances du verrouillage interne et de ReentrantLock avec Java 5.0 et Java 6.



1. Bien que ce graphique ne le montre pas, la différence d'adaptabilité entre Java 5.0 et Java 6 vient vraiment de l'amélioration du verrouillage interne plutôt que de la régression dans ReentrantLock.

Avec Java 5.0, les performances des verrous internes chutent énormément lorsque l'on passe d'un seul thread (pas de compétition) à plusieurs ; celles de `ReentrantLock` baissent moins, ce qui montre la meilleure adaptabilité de ce type de verrou. Avec Java 6.0, c'est une autre histoire – les résultats des verrous internes ne se détériorent plus avec la compétition et les deux types de verrous ont une adaptabilité assez similaire.

Des graphiques comme ceux de la Figure 13.1 nous rappellent que des affirmations comme "X est plus rapide que Y" sont, au mieux, éphémères. Les performances et l'adaptabilité tiennent compte de facteurs propres à une plate-forme, comme le type et le nombre de processeurs, la taille du cache et les caractéristiques de la JVM, qui peuvent tous évoluer au cours du temps¹.

Les performances sont une cible mouvante ; un test montrant hier que X était plus rapide que Y peut ne plus être vrai aujourd'hui.

13.3 Équité

Le constructeur `ReentrantLock` permet de choisir entre deux politiques : on peut créer un verrou *non équitable* (ce qui est le comportement par défaut) ou un verrou *équitable*. Les threads prennent un verrou équitable dans l'ordre où ils l'ont demandé alors qu'un verrou non équitable autorise le *bousculement* : les threads qui demandent un verrou peuvent sortir de la file des threads en attente si le verrou est disponible au moment où il est demandé (`Semaphore` offre aussi le choix entre une acquisition équitable et non équitable). Les verrous `ReentrantLock` non équitables ne sortent pas de la file pour encourager le bousculement – ils n'empêchent tout simplement pas un thread de passer devant tout le monde s'il se présente au bon moment. Avec un verrou équitable, un thread qui vient de le demander est mis dans la file d'attente si le verrou est détenu par un autre thread ou si d'autres threads sont déjà en attente du verrou ; avec un verrou non équitable, le thread n'est placé dans la file d'attente que si le verrou est déjà détenu par un autre thread².

Pourquoi ne pas utiliser que des verrous équitables ? Après tout, l'équité, c'est bien, et l'injustice, c'est mal, n'est-ce pas (il suffit de demander à vos enfants) ? Cependant, concernant le verrouillage, l'équité a un impact non négligeable sur les performances à cause des coûts de suspension et de réveil des threads. En pratique, une garantie d'équité statistique – la promesse qu'un thread bloqué finira par acquérir le verrou – suffit

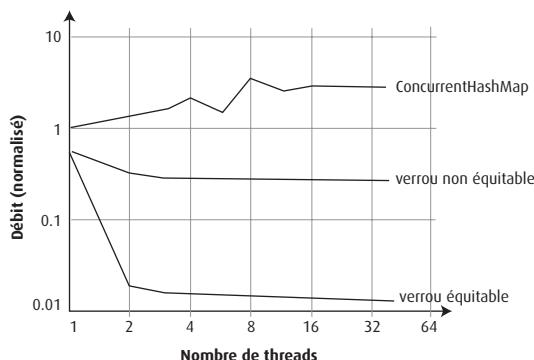
1. Quand nous avons commencé à écrire ce livre, `ReentrantLock` semblait être le dernier cri en terme d'adaptabilité des verrous. Moins de un an plus tard, les verrous internes lui ont rendu la monnaie de sa pièce. Les performances ne sont pas simplement une cible mouvante, elles peuvent être une cible très rapide.

2. La méthode `tryLock()` essaie toujours de bousculer l'ordre, même avec des verrous équitables.

souvent et est bien moins coûteuse à délivrer. Les algorithmes qui reposent sur une file d'attente équitable pour garantir leur fiabilité sont rares ; dans la plupart des cas, les bénéfices en termes de performances des verrous non équitables sont bien supérieurs à ceux d'une attente équitable.

Figure 13.2

Performances des verrous équitables et non équitables.



La Figure 13.2 montre un autre test des performances de Map, mais compare cette fois-ci un HashMap enveloppé avec des ReentrantLock équitables et non équitables sur un système à quatre processeurs Opteron avec Solaris ; les résultats sont représentés selon une échelle logarithmique¹. On constate que la pénalité de l'équité est d'environ deux ordres de grandeur. *Ne payez pas pour l'équité si cela n'est pas nécessaire.*

Une des raisons pour lesquelles les verrous non équitables ont de meilleures performances que les verrous équitables en cas de forte compétition est qu'il peut s'écouler un certain délai entre le moment où un thread suspendu est réveillé et celui où il s'exécute vraiment. Supposons que A détienne un verrou et que le thread B demande ce verrou. Celui-ci étant déjà pris, B est suspendu. Lorsque A libère le verrou, B est réveillé et peut donc tenter à nouveau de le prendre. Entre-temps, si un thread C a demandé le verrou, il y a de fortes chances qu'il l'obtienne, l'utilise et le libère avant même que B ait terminé de se réveiller. Dans ce cas, tout le monde est gagnant : B n'a pas été retardé dans sa prise du verrou, C l'a obtenu bien plus tôt et le débit est donc amélioré.

1. La courbe de ConcurrentHashMap est assez perturbée dans la région comprise entre quatre et huit threads. Ces variations sont presque certainement dues à un bruit lors de la mesure qui pourrait avoir été produit par des interactions entre les codes de hachage des éléments, la planification des threads, le changement de taille du Map, le ramasse-miettes ou d'autres actions sur la mémoire, voire par le SE, qui aurait pu effectuer une tâche périodique au moment de l'exécution du test. La réalité est qu'il y a toutes sortes de variations dans les tests de performances et qu'il ne sert généralement à rien d'essayer de les contrôler. Nous n'avons pas voulu nettoyer artificiellement le graphique car les mesures des performances dans le monde réel sont également perturbées par le bruit.

Les verrous équitables fonctionnent mieux lorsqu'ils sont détenus pendant relativement longtemps ou que le délai entre deux demandes de verrou est assez important. Dans ces situations, la condition qui fait que les verrous non équitables fournissent un meilleur débit – lorsque le verrou est libre mais qu'un thread est en train de se réveiller pour le réclamer – a moins de chance d'être vérifiée. Comme les verrous `ReentrantLock` par défaut, le verrouillage interne n'offre aucune garantie d'équité déterministe, mais les garanties d'équité statistique de la plupart des implémentations de verrous suffisent dans quasiment toutes les situations. La spécification du langage n'exige pas que la JVM implémente les verrous internes pour qu'ils soient équitables et aucune JVM actuelle ne le fait. `ReentrantLock` ne réduit donc pas l'équité des verrous à un nouveau minimum – elle rend simplement explicite quelque chose qui existait déjà.

13.4 synchronized vs. ReentrantLock

`ReentrantLock` fournit la même sémantique de verrouillage et de mémoire que les verrous internes, ainsi que des fonctionnalités supplémentaires comme les attentes de verrous avec délai, interruptibles, l'équité et la possibilité d'implémenter un verrouillage non structuré en bloc. Les performances de `ReentrantLock` semblent dominer celles des verrous internes, légèrement avec Java 6 mais énormément avec Java 5.0. Pourquoi alors ne pas déprécier `synchronized` et inciter tous les nouveaux codes à utiliser `ReentrantLock` ?

Certains auteurs ont, en réalité, fait cette proposition, en traitant `synchronized` comme une construction "historique" ; mais c'est remiser une bonne chose bien trop loin. Les verrous internes ont des avantages non négligeables sur les verrous explicites. Leur notation est familière et compacte et de nombreux programmes existants les utilisent – mélanger les deux ne ferait qu'apporter de la confusion et provoquerait sûrement des erreurs. `ReentrantLock` est vraiment un outil plus dangereux que la synchronisation ; si l'on oublie d'envelopper l'appel `unlock()` dans un bloc `finally`, le code semblera fonctionner correctement alors que l'on aura créé une bombe à retardement qui blessera d'innocents spectateurs.

Réservez `ReentrantLock` pour les situations dans lesquelles vous avez besoin d'une de ses fonctionnalités qui n'existe pas avec les verrous internes.

`ReentrantLock` est un outil avancé pour les situations dans lesquelles les verrous internes ne sont pas appropriés. Utilisez-le si vous avez besoin de ses fonctionnalités avancées : pour l'acquisition de verrous avec délais, `scrutable` ou `interruptible`, pour les mises en attente équitables ou pour un verrouillage non structuré en bloc. Dans les autres cas, préférez `synchronized`.

Avec Java 5.0, le verrouillage interne a un autre avantage sur `ReentrantLock` : les traces de threads montrent quels appels ont pris quels verrous et peuvent détecter et identifier les threads en interblocage. La JVM, en revanche, ne sait rien des threads qui ont pris des `ReentrantLock` et ne sera donc d'aucun secours pour déboguer les problèmes des threads qui les utilisent. Cette différence de traitement a été résolue par Java 6, qui fournit une interface de gestion et de surveillance auprès de laquelle les threads peuvent s'enregistrer afin que les informations sur les verrous externes apparaissent dans les traces et dans les autres interfaces de gestion et de débogage. Ces informations constituent un avantage (temporaire) pour `synchronized` ; les informations de verrouillage dans les traces de threads ont épargné bien des soucis à de nombreux programmeurs. Le fait que, par nature, `ReentrantLock` ne soit pas structuré en bloc signifie quand même que les prises de verrous ne peuvent pas être liées à des cadres de pile spécifiques, comme c'est le cas avec les verrous internes.

Les améliorations de performances futures favoriseront sûrement `synchronized` par rapport à `ReentrantLock`. Les verrous internes sont, en effet, intégrés à la JVM, qui peut effectuer des optimisations comme l'élation de verrou pour les verrous confinés à un thread et l'épaississement de verrou pour éliminer la synchronisation avec les verrous internes (voir la section 11.3.2) ; faire la même chose avec des verrous externes semble bien moins évident. À moins de déployer des applications Java 5.0 dans un avenir prévisible et d'avoir réellement besoin des bénéfices de l'adaptabilité fournie par `ReentrantLock` sur votre plate-forme, il n'est pas conseillé de l'utiliser à la place de `synchronized` pour des raisons de performances.

13.5 Verrous en lecture-écriture

`ReentrantLock` implémente un verrou d'exclusion mutuelle standard : à un instant donné, un `ReentrantLock` ne peut être détenu que par un et un seul thread. Cependant, l'exclusion mutuelle est souvent une discipline de verrouillage plus forte qu'il n'est nécessaire pour préserver l'intégrité des données, et elle limite donc trop la concurrence. Il s'agit en effet d'une stratégie de verrouillage classique qui empêche les conflits entre écrivain et écrivain ou entre écrivain et lecteur mais empêche également la coexistence d'un lecteur avec un autre lecteur. Dans de nombreux cas, pourtant, les structures de données sont "principalement en lecture" – elles sont modifiables et parfois modifiées, mais la plupart des accès ne sont qu'en lecture. Dans ces situations, il serait pratique de relâcher les exigences du verrouillage pour permettre à plusieurs lecteurs d'accéder en même temps à la structure. Tant que l'on est sûr que chaque thread verra une version à jour des données et qu'aucun autre thread ne les modifiera pendant que les lecteurs les consultent, il n'y aura aucun problème. C'est ce que permettent de faire les verrous en lecture-écriture : plusieurs lecteurs simultanément ou un seul écrivain peuvent accéder à une ressource, mais pas les deux.

L'interface `ReadWriteLock`, présentée dans le Listing 13.6, expose deux objets `Lock` – un pour la lecture, l'autre pour l'écriture. Pour lire les données protégées par un `ReadWriteLock` il faut d'abord prendre le verrou de lecture puis détenir le verrou d'écriture pour modifier ces données. Ces deux verrous sont simplement des vues différentes d'un objet verrou de lecture-écriture.

Listing 13.6 : Interface `ReadWriteLock`.

```
public interface ReadWriteLock {  
    Lock readLock();  
    Lock writeLock();  
}
```

La stratégie de verrouillage implémentée par les verrous de lecture-écriture autorise plusieurs lecteurs simultanés mais un seul écrivain. Comme `Lock`, `ReadWriteLock` a plusieurs implémentations dont les performances, les garanties de planification, les préférences d'acquisition, l'équité ou la sémantique de verrouillage peuvent varier.

Les verrous de lecture-écriture constituent une optimisation des performances et permettent une plus grande concurrence dans certaines situations. En pratique, ils améliorent les performances pour les structures de données auxquelles on accède le plus souvent en lecture sur des systèmes multiprocesseurs ; dans d'autres situations, leurs performances sont légèrement inférieures à celles des verrous exclusifs car ils sont plus complexes. Le meilleur moyen de savoir s'ils sont avantageux pour une application consiste donc à analyser l'exécution du code à l'aide d'un profileur ; `ReadWriteLock` utilisant `Lock` pour définir les verrous de lecture et d'écriture, il est relativement simple de remplacer un verrou de lecture-écriture par un verrou exclusif lorsque le profileur estime qu'un verrou de lecture-écriture n'apporte rien.

L'interaction entre les verrous de lecture et d'écriture autorise un certain nombre d'implémentations. Voici quelques-unes des options possibles pour une implémentation de `ReadWriteLock` :

- **Préférence lors de la libération.** Lorsqu'un écrivain relâche le verrou d'écriture et que des lecteurs et des écrivains sont en attente, qui doit avoir la préférence : les lecteurs, les écrivains, celui qui a demandé le premier ?
- **Priorité aux lecteurs.** Si le verrou est détenu par des lecteurs et que des écrivains soient en attente, les nouveaux lecteurs doivent-ils avoir un accès immédiat ou attendre derrière les écrivains ? Autoriser les lecteurs à passer devant les écrivains améliore l'adaptabilité mais court le risque d'affamer les écrivains.
- **Réentrance.** Les verrous de lecture et d'écriture sont-ils réentrant ?
- **Déclassement.** Si un thread détient le verrou d'écriture, peut-il prendre un verrou de lecture sans libérer celui d'écriture ? Cela permettrait à un écrivain de se "contenter" d'un verrou de lecture sans laisser les autres écrivains modifier entre-temps la ressource protégée.

- **Promotion.** Un verrou de lecture peut-il être promu en verrou d'écriture au détriment des autres lecteurs ou écrivains en attente ? La plupart des implémentations des verrous de lecture-écriture n'autorisent pas cette promotion car, sans opération explicite, elle risque de provoquer des interblocages (si deux lecteurs tentent simultanément de promouvoir leur verrou en verrou d'écriture, aucun d'eux ne relâchera le verrou de lecture).

`ReentrantReadWriteLock` fournit une sémantique de verrouillage réentrant pour les deux verrous. Comme `ReentrantLock`, un `ReentrantReadWriteLock` peut être non équitable (comportement par défaut) ou équitable. Dans ce dernier cas, la préférence est donnée au thread qui attend le verrou depuis le plus longtemps ; si le verrou est détenu par des lecteurs et qu'un thread demande le verrou d'écriture, aucun autre lecteur n'est autorisé à prendre le verrou de lecture tant que l'écrivain n'a pas été servi et qu'il n'a pas relâché le verrou d'écriture. Avec un verrou non équitable, l'ordre d'accès des threads n'est pas spécifié. Le déclassement d'un écrivain en lecteur est autorisé, mais pas la promotion d'un lecteur en écrivain (cette tentative produit un interblocage).

Comme `ReentrantLock`, le verrou d'écriture de `ReentrantReadWriteLock` n'a qu'un seul propriétaire et ne peut être libéré que par le thread qui l'a pris. Avec Java 5.0, le verrou de lecture se comporte plus comme un `Semaphore` que comme un verrou ; il ne mémorise que le nombre de lecteurs actifs, pas leurs identités. Ce comportement a été modifié en Java 6 pour garder également la trace des threads qui ont reçu le verrou de lecture¹.

Le verrouillage en lecture-écriture peut améliorer la concurrence lorsque les verrous sont le plus souvent détenus pendant un temps moyennement long et que la plupart des opérations ne modifient pas les ressources qu'ils protègent. La classe `ReadWriteMap` du Listing 13.7 utilise ainsi un `ReentrantReadWriteLock` pour envelopper un `Map` afin qu'il puisse être partagé sans problème par plusieurs lecteurs tout en empêchant les conflits lecteur-écrivain et écrivain-écrivain². En réalité, les performances de `ConcurrentHashMap` sont si bonnes que vous l'utiliserez sûrement à la place de cette approche si vous aviez simplement besoin d'un `Map` concurrent reposant sur un hachage ; cependant, cette technique reste utile pour fournir plus d'accès concurrents à une autre implémentation de `Map`, comme `LinkedHashMap`.

1. L'une des raisons de cette modification est qu'avec Java 5.0 l'implémentation du verrou ne peut pas faire la différence entre un thread qui demande le verrou de lecture pour la première fois et une demande de verrou réentrant, ce qui ferait des verrous de lecture-écriture équitables des sources possibles d'interblocage.

2. `ReadWriteMap` n'implémente pas `Map` car implémenter des méthodes de consultation comme `entrySet()` et `values()` serait compliqué alors que les méthodes "faciles" suffisent généralement.

Listing 13.7 : Enveloppe d'un Map avec un verrou de lecture-écriture.

```

public class ReadWriteMap<K,V> {
    private final Map<K,V> map;
    private final ReadWriteLock lock = new ReentrantReadWriteLock ();
    private final Lock r = lock.readLock();
    private final Lock w = lock.writeLock();

    public ReadWriteMap(Map<K,V> map) {
        this.map = map;
    }

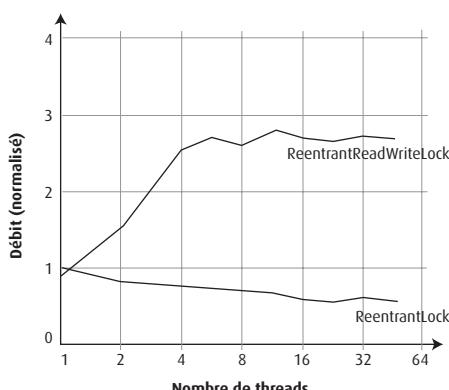
    public V put(K key, V value) {
        w.lock();
        try {
            return map.put(key, value);
        } finally {
            w.unlock();
        }
    }
    // Idem pour remove(), putAll(), clear()

    public V get(Object key) {
        r.lock();
        try {
            return map.get(key);
        } finally {
            r.unlock();
        }
    }
    // Idem pour les autres méthodes en lecture seule de Map
}

```

La Figure 13.3 compare les débits d'une `ArrayList` enveloppée avec un `ReentrantLock` et avec un `ReadWriteLock` sur un système à quatre processeurs Opteron tournant sous Solaris. Ce programme de test est comparable à celui que nous avons utilisé pour tester les performances de `Map` tout au long du livre – chaque opération choisit aléatoirement une valeur et la recherche dans la collection, seul un petit pourcentage d'actions modifie le contenu du `Map`.

Figure 13.3
Performances
du verrouillage
en lecture-écriture.



Résumé

Les Lock explicites disposent d'un ensemble de fonctionnalités supérieur à celui des verrous internes ; ils offrent notamment une plus grande souplesse dans la gestion de l'indisponibilité du verrou et un plus grand contrôle sur le comportement de la mise en attente. Cependant, ReentrantLock n'est pas un remplaçant systématique de synchronized ; il ne faut l'utiliser que lorsque l'on a besoin des caractéristiques absentes de synchronized. Les verrous de lecture-écriture permettent à plusieurs lecteurs d'accéder simultanément à un objet protégé, ce qui permet d'améliorer l'adaptabilité lorsque l'on utilise des structures de données qui sont plus souvent lues que modifiées.

Construction de synchronisateurs personnalisés

Les bibliothèques de Java contiennent un certain nombre de classes *dépendantes de l'état* – c'est-à-dire des classes avec des *préconditions reposant sur un état* – comme `FutureTask`, `Semaphore` et `BlockingQueue`. On ne peut pas, par exemple, supprimer un élément d'une file vide ou récupérer le résultat d'une tâche qui ne s'est pas encore terminée ; avant de pouvoir effectuer ces opérations, il faut attendre que la file passe dans l'état "non vide" ou que la tâche entre dans l'état "terminée".

Le moyen le plus simple de construire une classe dépendante de l'état consiste généralement à la créer à partir d'une classe de la bibliothèque, elle-même dépendante de l'état : c'est d'ailleurs ce que nous avons fait pour `ValueLatch` au Listing 8.17 puisque nous avions utilisé la classe `CountDownLatch` pour fournir les blocages nécessaires. Si les classes existantes ne fournissent pas la fonctionnalité requise, il est également possible d'écrire ses propres synchronisateurs à l'aide des mécanismes de bas niveau offerts par le langage et ses bibliothèques, notamment les *files d'attentes de conditions internes*, les objets `Condition` explicites et le framework `AbstractQueuedSynchronizer`. Ce chapitre explore les différents moyens de réaliser cette implémentation, ainsi que les règles qui permettent d'utiliser les mécanismes fournis par la plate-forme pour gérer cette dépendance.

14.1 Gestion de la dépendance par rapport à l'état

Dans un programme monothread, une précondition sur l'état (comme "le pool de connexion n'est pas vide") non vérifiée lorsqu'une méthode est appelée ne deviendra jamais vraie. Les classes dans les programmes séquentiels peuvent donc être écrites pour échouer lorsque leurs préconditions ne sont pas vérifiées. Dans un programme concurrent, en revanche, les conditions sur l'état peuvent évoluer grâce aux actions d'autres threads : un pool qui était vide il y a quelques instructions peut ne plus l'être parce qu'un autre thread

a renvoyé un élément. Avec les objets concurrents, les méthodes qui dépendent de l'état peuvent parfois être refusées lorsque leurs préconditions ne sont pas vérifiées, mais il y a souvent une meilleure alternative : attendre que les préconditions deviennent vraies.

Les opérations dépendantes de l'état qui se bloquent jusqu'à ce qu'elles puissent s'exécuter sont plus pratiques et moins sujettes aux erreurs que celles qui se contentent d'échouer. Le mécanisme interne de file d'attente de condition permet aux threads de se bloquer jusqu'à ce qu'un objet passe dans un état qui permette d'avancer et réveille les threads bloqués lorsqu'ils peuvent poursuivre leur progression. Nous décrirons les files d'attente de condition dans la section 14.2 mais, pour insister sur l'intérêt d'un mécanisme efficace d'attente d'une condition, nous montrerons d'abord comment la dépendance par rapport à l'état pourrait être (difficilement) mise en place à l'aide de tests et de mises en sommeil.

Le Listing 14.1 présente la structure générale d'une action bloquante en fonction de l'état. Le motif du verrouillage est un peu inhabituel puisque le verrou est relâché puis repris au milieu de l'opération. Les variables d'état qui composent la précondition doivent être protégées par le verrou de l'objet afin de rester constantes pendant le test de la précondition. Si celle-ci n'est pas vérifiée, le verrou doit être libéré pour qu'un autre thread puisse modifier l'état de l'objet – sinon cette précondition ne pourrait jamais devenir vraie. Le verrou doit ensuite être repris avant de tester à nouveau la précondition.

Listing 14.1 : Structure des actions bloquantes en fonction de l'état.

```
prendre le verrou sur l'état de l'objet
tant que (précondition non vérifiée) {
    libérer le verrou
    attendre que la précondition puisse être vérifiée
    échouer éventuellement en cas d'interruption ou d'expiration d'un délai
    reprise du verrou
}
effectuer l'action
libérer le verrou
```

Les conceptions de type producteur-consommateur utilisent souvent des tampons bornés comme `ArrayBlockingQueue`, qui fournit les opérations `put()` et `take()` ayant, chacune, leurs propres préconditions : on ne peut pas placer un élément dans un tampon plein ni en prendre un dans un tampon vide. Les opérations dépendantes de l'état peuvent traiter les échecs des préconditions en lançant une exception ou en renvoyant un code d'erreur (ce qui transfère le problème à l'appelant), voire en se bloquant jusqu'à ce que l'objet passe dans l'état approprié.

Nous allons développer plusieurs implémentations d'un tampon borné en utilisant différentes approches pour traiter l'échec des préconditions. Chacune étend la classe `BaseBoundedBuffer` du Listing 14.2, qui implémente un tampon circulaire à l'aide d'un tableau et dans laquelle les variables d'état (`buf`, `head`, `tail` et `count`) sont protégées par le verrou interne du tampon. Cette classe fournit les méthodes synchronisées `doPut()` et

`doTake()`, qui seront utilisées par les sous-classes pour implémenter leurs opérations `put()` et `take()` ; l'état sous-jacent est caché aux sous-classes.

Listing 14.2 : Classe de base pour les implémentations de tampons bornés.

```
@ThreadSafe
public abstract class BaseBoundedBuffer <V> {
    @GuardedBy("this") private final V[] buf;
    @GuardedBy("this") private int tail;
    @GuardedBy("this") private int head;
    @GuardedBy("this") private int count;

    protected BaseBoundedBuffer (int capacity) {
        this.buf = (V[]) new Object[capacity];
    }

    protected synchronized final void doPut(V v) {
        buf[tail] = v;
        if (++tail == buf.length)
            tail = 0;
        ++count;
    }

    protected synchronized final V doTake() {
        V v = buf[head];
        buf[head] = null;
        if (++head == buf.length)
            head = 0;
        --count;
        return v;
    }

    public synchronized final boolean isFull() {
        return count == buf.length;
    }

    public synchronized final boolean isEmpty() {
        return count == 0;
    }
}
```

14.1.1 Exemple : propagation de l'échec de la précondition aux appels

La classe `GrumpyBoundedBuffer` du Listing 14.3 est une première tentative brutale d'implémentation d'un tampon borné. Les méthodes `put()` et `take()` sont synchronisées pour garantir un accès exclusif à l'état du tampon, car elles font toutes les deux appel à une logique de type *tester-puis-agir* lors de l'accès.

Listing 14.3 : Tampon borné qui se dérobe lorsque les préconditions ne sont pas vérifiées.

```
@ThreadSafe
public class GrumpyBoundedBuffer <V> extends BaseBoundedBuffer <V> {
    public GrumpyBoundedBuffer (int size) { super(size); }

    public synchronized void put(V v) throws BufferFullException {
        if (isFull())
            throw new BufferFullException();
        doPut(v);
    }
```



Listing 14.3 : Tampon borné qui se dérobe lorsque les préconditions ne sont pas vérifiées. (suite)

```
public synchronized V take() throws BufferEmptyException {
    if (isEmpty())
        throw new BufferEmptyException();
    return doTake();
}
```

Bien que simple à mettre en œuvre, cette approche est pénible à utiliser. Les exceptions sont en effet censées être réservées aux conditions exceptionnelles [EJ Item 39] ; or "le tampon est plein" n'est pas une condition exceptionnelle pour un tampon borné, pas plus que "rouge" ne l'est pour un feu tricolore. La simplification lors de cette implémentation (qui force l'appelant à gérer la dépendance par rapport à l'état) est plus que noyée par les complications de son utilisation puisque l'appelant doit être préparé à capturer les exceptions et, éventuellement, à retenter chaque opération sur le tampon¹. Le Listing 14.4 montre un appel à `take()` correctement structuré – ce n'est pas très joli, notamment si `put()` et `take()` sont souvent appelées au cours du programme.

Listing 14.4 : Code client pour l'appel de `GrumpyBoundedBuffer`.

```
while (true) {
    try {
        V item = buffer.take();
        // Utilisation de item
        break;
    } catch (BufferEmptyException e) {
        Thread.sleep(SLEEP_GRANULARITY );
    }
}
```

Une variante de cette approche consiste à renvoyer un code d'erreur lorsque le tampon n'est pas dans le bon état. Il s'agit d'une petite amélioration puisqu'elle n'abuse plus du mécanisme des exceptions uniquement pour signifier "désolé, réessayez", mais elle ne règle pas le problème fondamental, qui est que ce sont les appelants qui doivent gérer eux-mêmes les échecs des préconditions².

Le code client du Listing 14.4 n'est pas le seul moyen d'implémenter le réessai. L'appelant pourrait retenter immédiatement `take()`, sans se mettre en sommeil – c'est ce que l'on appelle une *attente active*. Cette approche risque de consommer beaucoup de ressources processeur si l'état du tampon ne change pas pendant un certain temps. En revanche, si l'appelant décide de se mettre en sommeil pour ne pas gaspiller trop de temps processeur, il risque de dormir trop longtemps si le tampon change d'état juste après l'appel à `sleep()`.

1. Repousser vers l'appelant la dépendance par rapport à l'état empêche également la préservation de l'ordre FIFO ; en forçant l'appelant à réessayer, on ne sait plus qui est arrivé le premier.

2. Queue offre ces deux options – `poll()` renvoie `null` si la file est vide et `remove()` lance une exception – mais elle n'a pas été conçue pour être utilisée dans un contexte producteur-consommateur. Lorsque des producteurs et des consommateurs doivent s'exécuter en parallèle, il est préférable d'utiliser `BlockingQueue` car ses opérations se bloquent jusqu'à ce que la file soit dans le bon état.

C'est donc le code du client qui doit choisir entre une mauvaise utilisation du processeur par l'attente active et une mauvaise réactivité due à sa mise en sommeil (un intermédiaire entre l'attente active et le sommeil consiste à faire appel à `Thread.yield()` dans chaque itération pour indiquer au planificateur qu'il serait raisonnable de laisser un autre thread s'exécuter. Si l'on attend le résultat d'un autre thread, ce résultat pourrait arriver plus vite si on libère le processeur plutôt que consommer tout le quantum de temps qui nous a été imparti).

14.1.2 Exemple : blocage brutal par essai et mise en sommeil

La classe `SleepyBoundedBuffer` du Listing 14.5 tente d'épargner aux appelants l'inconvénient d'implémenter le code de réessay à chaque appel ; pour cela, elle encapsule le même mécanisme brutal de *essayer-et-dormir* à l'intérieur des opérations `put()` et `take()`. Si le tampon est vide, `take()` s'endort jusqu'à ce qu'un autre thread y ajoute des données ; s'il est plein, `put()` s'endort jusqu'à ce qu'un autre thread fasse de la place en supprimant des données. Cette approche encapsule donc le traitement des préconditions et simplifie l'utilisation du tampon – c'est donc un pas dans la bonne direction.

Listing 14.5 : Tampon borné avec blocage brutal.

```
@ThreadSafe
public class SleepyBoundedBuffer <V> extends BaseBoundedBuffer <V> {
    public SleepyBoundedBuffer (int size) { super(size); }

    public void put(V v) throws InterruptedException {
        while (true) {
            synchronized (this) {
                if (!isFull()) {
                    doPut(v);
                    return;
                }
            }
            Thread.sleep( SLEEP_GRANULARITY );
        }
    }

    public V take() throws InterruptedException {
        while (true) {
            synchronized (this) {
                if (!isEmpty())
                    return doTake();
            }
            Thread.sleep( SLEEP_GRANULARITY );
        }
    }
}
```



L'implémentation de `SleepyBoundedBuffer` est plus compliquée que celle de la tentative précédente¹. Le code du tampon doit tester la condition appropriée en détenant le verrou du tampon car les variables qui représentent cette condition sont protégées par

1. Nous passerons sous silence les cinq autres implémentations de tampons bornés de Snow White, notamment `SneezyBoundedBuffer`.

ce verrou. Si le test échoue, le thread en cours s'endort pendant un moment en relâchant d'abord le verrou afin que d'autres threads puissent accéder au tampon¹. Lorsque le thread se réveille, il reprend le verrou et réessaie en alternant sommeil et test de la condition jusqu'à ce que l'opération puisse se dérouler.

Du point de vue de l'appelant, tout marche à merveille – l'opération s'exécute si elle peut avoir lieu immédiatement ou se bloque sinon – et il n'a pas besoin de s'occuper de la mécanique des échecs et des réessais. Choisir la granularité du sommeil résulte d'un compromis entre réactivité et utilisation du processeur ; plus elle est fine, plus on est réactif mais plus on consomme de ressources processeur. La Figure 14.1 montre l'influence de cette granularité sur la réactivité : il peut s'écouler un délai entre le moment où de l'espace se libère dans le tampon et celui où le thread se réveille et teste à nouveau la condition.

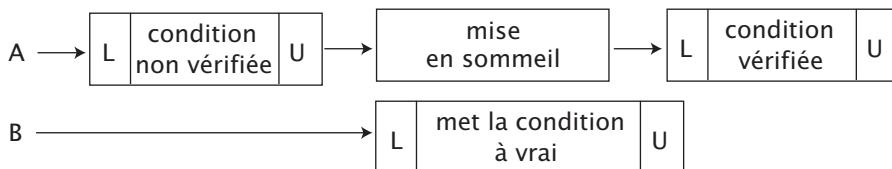


Figure 14.1

Sommeil trop profond car la condition devient vraie juste après l'endormissement du thread.

SleepyBoundedBuffer impose aussi une autre exigence pour l'appelant – celui-ci doit traiter InterruptedException. Lorsqu'une méthode se bloque en attendant qu'une condition devienne vraie, le comportement adéquat consiste à fournir un mécanisme d'annulation (voir Chapitre 7). Pour respecter cette règle, SleepyBoundedBuffer gère les annulations grâce à une interruption en se terminant précocement et en levant InterruptedException.

Ces tentatives de synthétiser une opération bloquante à partir d'essais et de mises en sommeil sont, en réalité, assez lourdes. Il serait préférable de disposer d'un moyen de suspendre un thread tout en garantissant qu'il sera vite réveillé lorsqu'une certaine condition (quand le tampon n'est plus plein, par exemple) devient vraie. C'est exactement ce que font les files d'attentes de conditions.

1. Il est généralement fortement déconseillé qu'un thread s'endorme ou se bloque en déttenant un verrou mais, ici, c'est encore pire puisque la condition voulue (tampon plein ou vide) ne pourra jamais devenir vraie si le verrou n'est pas relâché !

14.1.3 Les files d'attente de condition

Les files d'attente de condition ressemblent au signal d'un grille-pain indiquant que le toast est prêt. Si on y est attentif, on est rapidement prévenu que l'on peut faire sa tartine et abandonner ce que l'on était en train de faire (mais on peut également vouloir finir de lire son journal) pour aller chercher le toast. Si on ne l'écoute pas (parce qu'on est sorti chercher le journal, par exemple), on peut observer au retour dans la cuisine l'état du grille-pain et récupérer le toast s'il est prêt ou recommencer à attendre le signal s'il ne l'est pas.

Une *file d'attente de condition* tire son nom du fait qu'elle permet à un groupe de threads – appelé *ensemble en attente* – d'attendre qu'une condition spécifique devienne vraie. À la différence des files d'attente classiques, dans lesquelles les éléments sont des données, les éléments d'une file d'attente de condition sont les threads qui attendent que la condition soit vérifiée.

Tout comme chaque objet Java peut agir comme un verrou, tout objet peut également se comporter comme une file d'attente de condition dont l'API est définie par les méthodes `wait()`, `notify()` et `notifyAll()` de la classe `Object`. Le verrou et la file d'attente de condition internes d'un objet sont liés : pour appeler l'une des méthodes de la file sur un objet *X*, il faut détenir le verrou sur *X*. En effet, le mécanisme pour attendre des conditions reposant sur l'état est nécessairement étroitement lié à celui qui préserve la cohérence de cet état : on ne peut pas attendre une condition si l'on ne peut pas examiner l'état et on ne peut pas libérer un autre thread de son attente d'une condition sans modifier l'état.

`Object.wait()` libère le verrou de façon atomique et demande au SE de suspendre le thread courant afin de permettre aux autres threads de le prendre et donc de modifier l'état de l'objet. Lorsqu'il est réveillé, le thread reprend le verrou avant de continuer. Intuitivement, appeler `wait()` signifie "je vais dormir, mais réveille-moi quand il se passe quelque chose d'intéressant" et appeler l'une des méthodes de notification signifie "il s'est passé quelque chose d'intéressant".

La classe `BoundedBuffer` du Listing 14.6 implémente un tampon borné à l'aide de `wait()` et `notifyAll()`. Son code est à la fois plus simple que les versions précédentes, plus efficace (le thread se réveille moins souvent si l'état du tampon n'est pas modifié) et plus réactif (le thread se réveille plus vite en cas de modification de l'état qui le concerne). C'est une grosse amélioration, mais vous remarquerez que l'introduction des files d'attente de condition n'a pas modifié la sémantique par rapport aux versions précédentes. Il s'agit simplement d'une optimisation à plusieurs dimensions : efficacité processeur, coût des changements de contexte et réactivité. Les files d'attente de condition ne permettent rien de plus que la mise en sommeil avec réessai¹, mais elles facilitent beaucoup l'expression et la gestion de la dépendance par rapport à l'état tout en la rendant plus efficace.

1. Ce n'est pas tout à fait vrai ; une file d'attente de condition équitable peut garantir l'ordre relatif dans lequel les threads seront sortis de l'ensemble en attente. Les files d'attente de condition internes, comme les verrous internes, ne permettent pas cette équité : les `Condition` explicites offrent ce choix.

Listing 14.6 : Tampon borné utilisant des files d'attente de condition.

```

@ThreadSafe
public class BoundedBuffer<V> extends BaseBoundedBuffer <V> {
    // PRÉDICAT DE CONDITION : non-plein (!isFull())
    // PRÉDICAT DE CONDITION : non-vide (!isEmpty())

    public BoundedBuffer(int size) { super(size); }

    // BLOQUE JUSQU'À : non-plein
    public synchronized void put(V v) throws InterruptedException {
        while (isFull())
            wait();
        doPut(v);
        notifyAll();
    }

    // BLOQUE JUSQU'À : non-vide
    public synchronized V take() throws InterruptedException {
        while (isEmpty())
            wait();
        V v = doTake();
        notifyAll();
        return v;
    }
}

```

La classe `BoundedBuffer` est finalement assez bonne pour être utilisée – elle est simple d’emploi et gère correctement la dépendance par rapport à l’état¹. Une version de production devrait également inclure une version avec délai de `put()` et `take()` afin que les opérations bloquantes puissent expirer si elles n’arrivent pas à se terminer dans le temps imparti. La version avec délai de `Object.wait()` permet d’implémenter facilement cette fonctionnalité.

14.2 Utilisation des files d'attente de condition

Les files d’attente de condition facilitent la construction de classes dépendantes de l’état efficaces et réactives mais on peut quand même les utiliser incorrectement ; beaucoup de règles sur leur bon emploi ne sont pas imposées par le compilateur ou la plate-forme (c’est l’une des raisons de s’appuyer sur des classes comme `LinkedBlockingQueue`, `CountDownLatch`, `Semaphore` et `FutureTask` lorsque cela est possible, car c’est bien plus simple).

14.2.1 Le prédictat de condition

La clé pour utiliser correctement les files d’attente de condition est d’identifier les *prédictats de condition* que l’objet peut attendre. Ce sont ces prédictats qui causent la plupart des confusions à propos de `wait()` et `notify()` car il n’y a rien dans l’API ni dans la spécification du langage ou l’implémentation de la JVM qui garantisse leur

1. La classe `ConditionBoundedBuffer` de la section 14.3 est encore meilleure : elle est plus efficace car elle peut utiliser une seule notification au lieu de `notifyAll()`.

emploi correct. En fait, ils ne sont jamais mentionnés directement dans la spécification du langage ou dans Javadoc. Pourtant, sans eux, les attentes de condition ne pourraient pas fonctionner.

Un prédictat de condition est la précondition qui rend une opération dépendante de l'état. Dans un tampon borné, `take()` ne peut se dérouler que si le tampon n'est pas vide ; sinon elle doit attendre. Pour cette opération, le prédictat de condition est donc "le tampon n'est pas vide" et `take()` doit le tester avant de faire quoi que ce soit. De même, le prédictat de condition de `put()` est "le tampon n'est pas plein". Les prédictats de condition sont des expressions construites à partir des variables d'état de la classe ; `BaseBoundedBuffer` teste que "le tampon n'est pas vide" en comparant `count` avec zéro et que "le tampon n'est pas plein" en comparant `count` avec la taille du tampon.

Documentez le ou les prédictats de condition associés à une file d'attente de condition et les opérations qui attendent qu'ils soient vérifiés.

Une attente de condition contient une relation tripartite importante qui implique le verrouillage, la méthode `wait()` et un prédictat de condition. Ce dernier implique les variables d'état, qui sont protégées par un verrou ; avant de tester le prédictat de condition, il faut donc détenir ce verrou, qui doit être le même que l'objet file d'attente de condition (celui sur lequel `wait()` et `notify()` sont invoquées).

Dans `BoundedBuffer`, l'état du tampon est protégé par le verrou du tampon et l'objet tampon est utilisé comme file d'attente de condition. La méthode `take()` prend le verrou du tampon, puis teste le prédictat de condition ("le tampon n'est pas vide"). Si le tampon n'est pas vide, la méthode supprime le premier élément, ce qu'elle peut faire puisqu'elle détient toujours le verrou qui protège l'état du tampon.

Si le prédictat de condition n'est pas vérifié (le tampon est vide), `take()` doit attendre qu'un autre thread y place un objet. Pour cela, elle appelle `wait()` sur la file d'attente de condition interne du tampon, ce qui nécessite de détenir le verrou sur cet objet file. Avec une conception soigneuse, `take()` détient déjà le verrou, qui lui est nécessaire pour tester le prédictat de condition (et, si ce dernier est vérifié, pour modifier l'état du tampon dans la même opération atomique). La méthode `wait()` libère le verrou, bloque le thread courant et attend jusqu'à ce que le délai imparti se soit écoulé, que le thread soit interrompu ou qu'il soit réveillé par une notification. Après le réveil du thread, `wait()` reprend le verrou avant de se terminer. Un thread qui se réveille de `wait()` n'a aucune priorité particulière sur la reprise du verrou ; il lutte pour son acquisition au même titre que n'importe quel autre thread désirant entrer dans un bloc `synchronized`.

Chaque appel à `wait()` est implicitement associé à un prédictat de condition spécifique. Lorsqu'il appelle `wait()` par rapport à un prédictat particulier, l'appelant doit déjà détenir

le verrou associé à la file d'attente de condition et ce verrou doit également protéger les variables d'état qui composent le prédicat de la condition.

14.2.2 Réveil trop précoce

Comme si la relation triangulaire entre le verrou, le prédicat de condition et la file d'attente de condition n'était pas assez compliquée, le fait que `wait()` se termine ne signifie pas nécessairement que le prédicat de condition qu'attend le thread soit devenu vrai.

Une même file d'attente de condition interne peut être utilisée avec plusieurs prédicats de condition. Lorsque le thread est réveillé parce que quelqu'un a appelé `notifyAll()`, cela ne signifie pas nécessairement que le prédicat attendu soit désormais vrai (c'est comme si le grille-pain et la machine à café partageaient la même sonnerie ; lorsqu'elle retentit, vous devez aller vérifier quelle machine a déclenché le signal¹). En outre, `wait()` peut même se terminer "sauvagement" – pas en réponse à un thread qui appelle `notify()`².

Lorsque le contrôle repasse à nouveau dans le code qui a appelé `wait()`, il a repris le verrou associé à la file d'attente de condition. Est-ce que le prédicat de condition est maintenant vérifié ? Peut-être. Il a pu être vrai au moment où le thread notificateur a appelé `notifyAll()` mais redevenir faux pendant le temps où vous avez repris le verrou. D'autres threads ont pu prendre le verrou et modifier l'état de l'objet entre le moment où votre thread s'est réveillé et celui où `wait()` a repris le verrou. Peut-être même n'a-t-il jamais été vrai depuis que vous avez appelé `wait()`. Vous ne savez pas pourquoi un autre thread a appelé `notify()` ou `notifyAll()` ; peut-être l'a-t-il fait parce qu'un autre prédicat de condition associé à la même file d'attente de condition est devenu vrai. Il est assez fréquent que plusieurs prédicats soient associés à la même file – `BoundedBuffer`, par exemple, utilise la même file d'attente de condition pour les deux prédicats "le tampon n'est pas plein" et "le tampon n'est pas vide"³.

Pour toutes ces raisons, il faut tester à nouveau le prédicat de condition lorsque l'on se réveille d'un appel à `wait()` et recommencer à attendre (ou échouer) s'il n'est pas vrai. Comme on peut se réveiller plusieurs fois sans que le prédicat ne soit vrai, il faut toujours appeler `wait()` dans une boucle en testant le prédicat de condition à chaque itération. Le Listing 14.7 montre la forme canonique d'une attente de condition.

1. Cette situation décrit très bien la cuisine de l'auteur ; il y a tant de machines qui sonnent que, lorsque l'on entend un signal, il faut vérifier le grille-pain, le micro-ondes, la machine à café et bien d'autres encore pour déterminer la cause de la sonnerie.

2. Pour aller encore plus loin dans l'analogie avec le petit déjeuner, c'est comme si le grille-pain sonnait lorsque le toast est prêt mais également parfois quand il ne l'est pas.

3. En fait, il est possible que les threads attendent en même temps "non plein" et "non vide" ! Cette situation peut arriver lorsque le nombre de producteurs-consommateurs dépasse la capacité du tampon.

Listing 14.7 : Forme canonique des méthodes dépendantes de l'état.

```
void stateDependentMethod() throws InterruptedException {
    // Le prédicat de condition doit être protégé par un verrou.
    synchronized(lock) {
        while (!conditionPredicate())
            lock.wait();
        // L'objet est désormais dans l'état souhaité
    }
}
```

Lorsque vous attendez une condition avec `Object.wait()` ou `Condition.await()` :

- Ayez toujours un prédicat de condition – un test de l'état de l'objet qui doit être vérifié avant de continuer.
- Testez toujours le prédicat de condition avant d'appeler `wait()` et à nouveau après le retour de `wait()`.
- Appelez toujours `wait()` dans une boucle.
- Vérifiez que les variables d'état qui composent le prédicat de condition sont protégées par le verrou associé à la file d'attente de condition.
- Prenez le verrou associé à la file d'attente de condition lorsque vous appelez `wait()`, `notify()` ou `notifyAll()`.
- Relâchez le verrou non pas après avoir testé le prédicat de condition, mais avant d'agir sur ce prédicat.

14.2.3 Signaux manqués

Le Chapitre 10 a présenté les échecs de vivacité comme les interblocages (*deadlocks*) et les livelocks. Les *signaux manqués* sont une autre forme d'échec de la vivacité. Un signal manqué a lieu lorsqu'un thread doit attendre une condition qui est déjà vraie mais qu'il oublie de vérifier le prédicat de la condition avant de se mettre en attente. Le thread attend alors une notification d'un événement qui a déjà eu lieu. C'est comme lancer le grille-pain, partir chercher le journal en laissant la sonnerie retentir pendant que l'on est dehors puis se rasseoir à la table en attendant que le grille-pain sonne : vous pourriez attendre longtemps – sûrement éternellement¹. À la différence de la confiture sur le toast, la notification ne "colle" pas – si un thread *A* envoie une notification à une file d'attente de condition et qu'un thread *B* se mette ensuite en attente sur cette file, *B* ne se réveillera *pas* immédiatement : il faudra une autre notification pour le réveiller. Les signaux manqués sont le résultat d'erreurs provenant, par exemple, du non-respect des conseils de la liste ci-dessus – ne pas tester le prédicat de condition avant d'appeler `wait()`,

1. Pour sortir de cette attente, quelqu'un d'autre devrait faire griller un toast, mais cela ne ferait qu'empirer le problème : lorsque la sonnerie retentirait, vous auriez alors un désaccord sur la propriété de la tartine.

notamment. Si vous structurez vos attentes de condition comme dans le Listing 14.7, vous n'aurez pas de problème avec les signaux manqués.

14.2.4 Notification

Pour l'instant, nous n'avons décrit que la moitié de ce qui se passe lors d'une attente de condition : l'attente. L'autre moitié est la notification. Dans un tampon borné, `take()` se bloque si elle est appelée alors que le tampon est vide. Pour que `take()` se *débloque* quand le tampon devient non vide, on doit s'assurer que chaque partie du code dans laquelle le tampon pourrait devenir non vide envoie une notification. Dans `BoundedBuffer`, il n'y a qu'un seul endroit – après un `put()`. Par conséquent, `put()` appelle `notifyAll()` après avoir réussi à ajouter un objet dans le tampon. De même, `take()` appelle `notifyAll()` après avoir supprimé un élément pour indiquer que le tampon peut ne plus être plein, au cas où des threads attendraient sur la condition "non plein".

À chaque fois que l'on attend sur une condition, il faut s'assurer que quelqu'un enverra une notification lorsque le prédictat de cette condition devient vrai.

Il y a deux méthodes de notification dans l'API des files d'attente de condition – `notify()` et `notifyAll()`. Pour appeler l'une ou l'autre, il faut détenir le verrou associé à l'objet file de condition. L'appel de `notify()` demande à la JVM de choisir *un* thread parmi ceux qui attendent dans cette file et de le réveiller, tandis qu'un appel à `notifyAll()` réveille *tous* les threads en attente dans la file. Comme il faut détenir le verrou sur l'objet file d'attente de condition lorsque l'on appelle `notify()` ou `notifyAll()` et que les threads en attente ne peuvent pas revenir du `wait()` sans reprendre le verrou, le thread qui envoie la notification doit relâcher ce verrou rapidement pour que les threads en attente soient débloqués le plus vite possible. Plusieurs threads pouvant attendre dans la même file d'attente de condition pour des prédictats de condition différents, il peut être dangereux d'utiliser `notify()` au lieu de `notifyAll()`, essentiellement parce qu'une unique notification ouvre la voie à un problème semblable aux signaux manqués.

`BoundedBuffer` offre une bonne illustration de la raison pour laquelle il vaut mieux utiliser `notifyAll()` plutôt qu'une notification simple dans la plupart des cas. La file d'attente de condition sert à deux prédictats de condition différents : "non plein" et "non vide". Supposons que le thread *A* attende dans une file la précondition *PA* alors que le thread *B* attend dans la même file le prédictat *PB*. Supposons maintenant que *PB* devienne vrai et que le thread *C* envoie une notification simple : la JVM réveillera donc un thread de son choix. Si elle choisit *A*, celui-ci se réveillera, constatera que *PA* n'est pas vérifiée et se remettra en attente. Pendant ce temps, *B*, qui aurait pu progresser, ne s'est pas réveillé. Il ne s'agit donc pas exactement d'un signal manqué – c'est plutôt un "signal volé" –, mais le problème est identique : un thread attend un signal qui s'est déjà produit (ou aurait dû se produire).

Les notifications simples ne peuvent être utilisées à la place de `notifyAll()` que lorsque les deux conditions suivantes sont réunies :

- **Attentes uniformes.** Un seul prédictat de condition est associé à la file d'attente de condition et chaque thread exécute le même code après l'appel à `wait()`.
- **Un seul dedans.** Une notification sur la variable condition autorise au plus un seul thread à continuer son exécution.

`BoundedBuffer` satisfait l'exigence *un seul dedans*, mais pas celle d'*attente uniforme* car les threads en attente peuvent attendre sur la condition "non plein" et "non vide". En revanche, un loquet "porte d'entrée" comme celui utilisé par la classe `TestHarness` du Listing 5.11, dans laquelle un unique événement libérait un ensemble de threads, ne satisfait pas cette exigence puisque l'ouverture de la porte permet à plusieurs threads de s'exécuter.

La plupart des classes ne satisfaisant pas ces exigences, la sagesse recommande d'utiliser `notifyAll()` plutôt que `notify()`. Bien que cela puisse ne pas être efficace, cela permet de vérifier bien plus facilement que les classes se comportent correctement.

Cette "sagesse" met certaines personnes mal à l'aise, et pour de bonnes raisons. L'utilisation de `notifyAll()` quand un seul thread peut progresser n'est pas efficace – cette pénalité est parfois faible, parfois assez nette. Si dix threads attendent dans une file de condition, l'appel de `notifyAll()` les forcera tous à se réveiller et à combattre pour l'acquisition du verrou, puis la plupart d'entre eux (voire tous) retourneront se mettre en sommeil. Ceci implique donc beaucoup de changements de contexte et des prises de verrou très disputées pour chaque événement qui autorise (éventuellement) un seul thread à progresser (dans le pire des cas, l'utilisation de `notifyAll()` produit $O(n^2)$ alors que n suffirait). C'est un autre cas où les performances demandent une certaine approche et la sécurité vis-à-vis des threads, une autre.

La notification de `put()` et `take()` dans `BoundedBuffer` est classique : elle est effectuée à chaque fois qu'un objet est placé ou supprimé du tampon. On pourrait optimiser en observant qu'un thread ne peut être libéré d'un `wait()` que si le tampon passe de vide à non vide ou de plein à non plein et n'envoyer de notification que si un `put()` ou un `take()` effectue l'une de ces transitions d'état : c'est ce que l'on appelle une *notification conditionnelle*. Bien qu'elle puisse améliorer les performances, la notification conditionnelle est difficile à mettre en place correctement (en outre, elle complique l'implémentation des sous-classes) et doit être utilisée avec prudence. Le Listing 14.8 illustre son utilisation avec la méthode `put()` de `BoundedBuffer`.

Listing 14.8 : Utilisation d'une notification conditionnelle dans `BoundedBuffer.put()`.

```
public synchronized void put(V v) throws InterruptedException {
    while (isFull())
        wait();
    boolean wasEmpty = isEmpty();
    doPut(v);
    if (wasEmpty)
        notifyAll();
}
```

Les notifications simples et conditionnelles sont des optimisations. Comme toujours, respectez le principe "Faites d'abord en sorte que cela fonctionne et faites ensuite que cela aille vite – si cela n'est pas déjà assez rapide" : il est très facile d'introduire d'étranges problèmes de vivacité lorsqu'on utilise incorrectement ce type de notification.

14.2.5 Exemple : une classe "porte d'entrée"

Le loquet "porte d'entrée" de `TestHarness` (voir Listing 5.11) est construit à l'aide d'un compteur initialisé à un, ce qui crée un *loquet binaire* à deux états : un état initial et un état final. Le loquet empêche les threads de passer la porte d'entrée tant qu'elle n'est pas ouverte, à la suite de quoi tous les threads peuvent passer. Bien que ce mécanisme soit parfois exactement ce dont on a besoin, le fait que cette sorte de porte ne puisse pas être refermée une fois ouverte a quelquefois des inconvénients.

Comme le montre le Listing 14.9, les attentes de condition permettent d'écrire assez facilement une classe `ThreadGate` qui peut être refermée. `ThreadGate` autorise l'ouverture et la fermeture de la porte en fournissant une méthode `await()` qui se bloque jusqu'à ce que la porte soit ouverte. La méthode `open()` utilise `notifyAll()` car la sémantique de cette classe ne respecterait pas le test "un seul dedans" avec une notification simple.

Listing 14.9 : Porte refermable à l'aide de `wait()` et `notifyAll()`.

```
@ThreadSafe
public class ThreadGate {
    // PRÉDICAT DE CONDITION : ouverte-depuis(n) (isOpen || generation > n)
    @GuardedBy("this") private boolean isOpen;
    @GuardedBy("this") private int generation;

    public synchronized void close() {
        isOpen = false;
    }

    public synchronized void open() {
        ++generation;
        isOpen = true;
        notifyAll();
    }

    // BLOQUE-JUSQU'À : ouverte-depuis(génération à l'entrée)
    public synchronized void await() throws InterruptedException {
        int arrivalGeneration = generation;
        while (!isOpen && arrivalGeneration == generation)
            wait();
    }
}
```

Le prédictat de condition utilisé par `await()` est plus compliqué qu'un simple test de `isOpen()`. En effet, si N threads attendent à la porte lorsqu'elle est ouverte, ils doivent tous être autorisés à entrer. Or, si la porte est ouverte puis refermée rapidement, tous les threads pourraient ne pas être libérés si `await()` examinait simplement `isOpen()` : au moment où ils reçoivent tous la notification, reprennent le verrou et sortent de `wait()`, la porte peut déjà s'être refermée. `ThreadGate` emploie donc un prédictat de condition plus compliqué : à chaque fois que la porte est fermée, un compteur "génération" est incrémenté et un thread peut passer `await()` si la porte est ouverte ou a été ouverte depuis qu'il est arrivé sur la porte.

Comme `ThreadGate` ne permet que d'attendre que la porte soit ouverte, elle n'envoie une notification que dans `open()` ; pour supporter à la fois les opérations "attente d'ouverture" et "attente de fermeture", elle devrait produire cette notification dans `open()` et dans `close()`. Ceci illustre la raison pour laquelle les classes dépendantes de l'état peuvent être délicates à maintenir – l'ajout d'une nouvelle opération dépendante de l'état peut nécessiter de modifier de nombreuses parties du code pour que les notifications appropriées puissent être envoyées.

14.2.6 Problèmes de safety des sous-classes

L'utilisation d'une notification simple ou conditionnelle introduit des contraintes qui peuvent compliquer l'héritage [CPJ 3.3.3.3]. Si l'on veut pouvoir hériter, il faut en effet structurer les classes pour que les sous-classes puissent ajouter la notification appropriée pour le compte de la classe de base lorsqu'elle est héritée d'une façon qui viole l'une des exigences des notifications simples ou conditionnelles.

Une classe dépendante de l'état doit exposer (et documenter) ses protocoles d'attente et de notification aux sous-classes ou empêcher celles-ci d'y participer (c'est une extension du principe "concevez et documentez vos classes pour l'héritage ou empêchez-le" [EJ Item 15]). Au minimum, la conception d'une classe dépendante de l'état pour qu'elle puisse être héritée exige d'exposer les files d'attente de condition et les verrous et de documenter les prédictats de condition et la politique de synchronisation ; il peut également être nécessaire d'exposer les variables d'état sous-jacentes (le pire qu'une classe dépendante de l'état puisse faire est d'exposer son état aux sous-classes sans documenter ses invariants). Une possibilité reste effectivement d'interdire l'héritage, soit en marquant la classe comme `final`, soit en cachant aux sous-classes ses files d'attente de condition, ses verrous et ses variables d'état. Dans le cas contraire, si la sous-classe fait quelque chose qui détériore l'utilisation de `notify()` par la classe de base, elle doit pouvoir réparer les dégâts. Considérons une pile non bornée bloquante dans laquelle l'opération `pop()` se bloque lorsque la pile est vide alors que l'opération `push()` réussit toujours : cette classe correspond aux exigences d'une notification simple. Si elle utilise une notification simple et qu'une sous-classe ajoute une méthode bloquante "dépile deux éléments consécutifs", on a maintenant deux types d'attentes : ceux qui attendent de dépiler un seul

élément et ceux qui attendent d'en dépiler deux. Si la classe de base expose la file d'attente de condition et documente les protocoles permettant de l'utiliser, la sous-classe peut redéfinir la méthode `push()` pour qu'elle effectue un `notifyAll()` afin de restaurer la safety.

14.2.7 Encapsulation des files d'attente de condition

Il est généralement préférable d'encapsuler la file d'attente de condition pour qu'elle ne soit pas accessible à l'extérieur de l'arborescence de classes dans laquelle elle est utilisée. Sinon les appelants pourraient être tentés de penser qu'ils comprennent vos protocoles d'attente et de notification et de les utiliser de façon incohérente par rapport à votre conception (il est impossible d'imposer l'exigence d'attente uniforme pour une notification simple si la file d'attente de condition est accessible à un code que vous ne contrôlez pas ; si un code étranger attend par erreur sur votre file d'attente, cela pourrait perturber votre protocole de notification et provoquer un vol de signal).

Malheureusement, ce conseil – encapsuler les objets utilisés comme files d'attente de condition – n'est pas cohérent avec la plupart des patrons de conception classiques des classes thread-safe, où le verrou interne d'un objet sert à protéger son état. `BoundedBuffer` illustre cet idiom classique, où l'objet tampon lui-même est le verrou et la file d'attente de condition. Cependant, cette classe pourrait aisément être restructurée pour utiliser un objet verrou et une file d'attente de condition privés ; la seule différence serait qu'elle ne permettrait plus aucune forme de verrouillage côté client.

14.2.8 Protocoles d'entrée et de sortie

Wellings (Wellings, 2004) caractérise l'utilisation correcte de `wait()` et `notify()` en termes de *protocoles d'entrée* et de *sortie*. Pour chaque opération dépendante de l'état et pour chaque opération modifiant l'état dont dépend une autre opération, il faudrait définir et documenter un protocole d'entrée et de sortie. Le protocole d'entrée est le prédictat de condition de l'opération, celui de sortie consiste à examiner toutes les variables d'état qui ont été modifiées par l'opération, afin de savoir si elles pourraient impliquer qu'un autre prédictat de condition devienne vrai et, dans ce cas, notifier la file d'attente de condition concernée.

La classe `AbstractQueuedSynchronizer`, sur laquelle la plupart des classes de `java.util.concurrent` dépendantes de l'état sont construites (voir la section 14.4), exploite ce concept de protocole de sortie. Au lieu de laisser les classes synchroniseurs effectuer leur propre notification, elle exige que les méthodes synchroniseurs renvoient une valeur indiquant si l'action aurait pu débloquer un ou plusieurs threads en attente. Cette exigence explicite de l'API rend plus difficile l'*oubli* d'une notification concernant certaines transitions d'état.

14.3 Objets conditions explicites

Comme nous l'avons vu au Chapitre 13, les Lock explicites peuvent être utiles dans les situations où les verrous internes ne sont pas assez souples. Tout comme Lock est une généralisation des verrous internes, l'interface Condition (décrise dans le Listing 14.10) est une généralisation des files d'attente de condition internes.

Listing 14.10 : Interface Condition.

```
public interface Condition {  
    void await() throws InterruptedException ;  
    boolean await(long time, TimeUnit unit) throws InterruptedException ;  
    long awaitNanos(long nanosTimeout) throws InterruptedException ;  
    void awaitUninterruptibly ();  
    boolean awaitUntil(Date deadline) throws InterruptedException ;  
    void signal();  
    void signalAll();  
}
```

Ces files ont, en effet, plusieurs inconvénients. Chaque verrou interne ne peut être associé qu'à une seule file d'attente de condition, ce qui signifie, dans des classes comme Bounded Buffer, que plusieurs threads peuvent attendre dans la même file pour des prédictats de condition différents et que le patron de verrouillage le plus classique implique d'exposer l'objet file d'attente de condition. Ces deux facteurs empêchent d'imposer l'exigence d'attente uniforme pour utiliser `notify()`. Pour créer un objet concurrent avec plusieurs prédictats de condition ou pour avoir plus de contrôle sur la visibilité de la file d'attente de condition, il est préférable d'utiliser les classes Lock et Condition explicites, car elles offrent une alternative plus souple que leurs équivalents internes.

Une Condition est associée à un seul Lock, tout comme une file d'attente de condition est associée à un seul verrou interne ; on crée une Condition, en appelant `Lock.newCondition()` sur le verrou concerné. Tout comme Lock offre plus de fonctionnalités que les verrous internes, Condition en fournit plus que les files d'attente de condition : plusieurs attentes par verrou, attentes de condition interruptibles ou non interruptibles, attentes avec délai et choix entre une attente équitable ou non équitable.

À la différence des files d'attente de condition internes, il est possible d'avoir autant d'objets Condition que l'on souhaite avec un même Lock. Les objets Condition héritent leur équité du Lock auquel ils sont associés ; avec un verrou équitable, les threads sont libérés de `Condition.await()` selon l'ordre FIFO.

Attention : les équivalents de `wait()`, `notify()` et `notifyAll()` pour les objets Condition sont, respectivement, `await()`, `signal()` et `signalAll()`. Cependant, Condition hérite de Object et dispose donc également de `wait()` et des deux méthodes de notification. Assurez-vous par conséquent d'utiliser les versions correctes – `await()` et les deux méthodes `signal` !

Le Listing 14.11 montre encore une autre implémentation d'un tampon borné, cette fois-ci à l'aide de deux Condition, notFull et notEmpty, pour représenter exactement les prédictats de condition "non plein" et "non vide". Lorsque take() se bloque parce que le tampon est vide, elle attend notEmpty et put() débloque tous les threads bloqués dans take() en envoyant un signal sur notEmpty.

Listing 14.11 : Tampon borné utilisant des variables conditions explicites.

```

@ThreadSafe
public class ConditionBoundedBuffer <T> {
    protected final Lock lock = new ReentrantLock ();
    // PRÉDICAT DE CONDITION : notFull (count < items.length)
    private final Condition notFull = lock.newCondition();
    // PRÉDICAT DE CONDITION : notEmpty (count > 0)
    private final Condition notEmpty = lock.newCondition();
    @GuardedBy("lock")
    private final T[] items = (T[]) new Object[BUFFER_SIZE];
    @GuardedBy("lock") private int tail, head, count;

    // BLOQUE-JUSQU'À : notFull
    public void put(T x) throws InterruptedException {
        lock.lock();
        try {
            while (count == items.length)
                notFull.await();
            items[tail] = x;
            if (++tail == items.length)
                tail = 0;
            ++count;
            notEmpty.signal();
        } finally {
            lock.unlock();
        }
    }

    // BLOQUE-JUSQU'À : notEmpty
    public T take() throws InterruptedException {
        lock.lock();
        try {
            while (count == 0)
                notEmpty.await();
            T x = items[head];
            items[head] = null;
            if (++head == items.length)
                head = 0;
            --count;
            notFull.signal();
            return x;
        } finally {
            lock.unlock();
        }
    }
}

```

Le comportement de ConditionBoundedBuffer est identique à celui de BoundedBuffer, mais l'utilisation de deux files d'attente de condition le rend plus lisible – il est plus facile d'analyser une classe qui utilise plusieurs Condition qu'une classe qui n'utilise qu'une seule file d'attente de condition avec plusieurs prédictats de condition. En séparant

les deux prédictats sur des attentes différentes, `Condition` facilite l'application des exigences pour les notifications simples. Utiliser `signal()`, qui est plus efficace que `signalAll()`, réduit le nombre de changements de contexte et de prises de verrous déclenchés par chaque opération du tampon.

Comme les verrous et les files d'attente de condition internes, la relation triangulaire entre le verrou, le prédictat de condition et la variable de condition doit également être vérifiée lorsque l'on utilise des `Lock` et des `Condition` explicites. Les variables impliquées dans le prédictat de condition doivent être protégées par le `Lock` et lui-même doit être détenu lorsque l'on teste le prédictat de condition et que l'on appelle `await()` et `signal()`¹. Vous devez choisir entre les `Condition` explicites et les files d'attente de condition internes comme vous le feriez pour `ReentrantLock` et `synchronized` : utilisez `Condition` si vous avez besoin de fonctionnalités supplémentaires, comme l'attente équitable ou plusieurs types d'attentes par verrou, mais préférez les files internes dans les autres cas (si vous utilisez déjà `ReentrantLock` parce que vous avez besoin de ses fonctionnalités étendues, votre choix est déjà fait).

14.4 Anatomie d'un synchronisateur

Les interfaces de `ReentrantLock` et `Semaphore` partagent beaucoup de points communs. Ces deux classes agissent comme des "portes" en ne laissant passer à la fois qu'un nombre limité de threads ; ceux-ci arrivent à la porte et sont autorisés à la franchir (si les appels `lock()` ou `acquire()` réussissent), doivent attendre (si `lock()` ou `acquire()` se bloquent) ou sont détournés (si `tryLock()` ou `tryAcquire()` renvoient `false`, ce qui indique que le verrou ou le permis n'est pas disponible dans le temps imparti). En outre, toutes les deux autorisent des tentatives interruptibles, non interruptibles et avec délai d'expiration, ainsi que le choix entre une mise en attente des threads équitable ou non équitable.

Avec tous ces points communs, on pourrait penser que `Semaphore` a été implémentée à partir de `ReentrantLock` ou que `ReentrantLock` l'a été à partir d'un `Semaphore` avec un seul permis. Ce serait tout à fait possible et il s'agit d'un exercice classique pour prouver qu'un sémaphore peut être implémenté à l'aide d'un verrou (comme `SemaphoreOnLock`, dans le Listing 14.12) et qu'un verrou peut l'être à partir d'un sémaphore.

Listing 14.12 : `Semaphore` implémenté à partir de `Lock`.

```
// java.util.concurrent.Semaphore n'est pas implémentée de cette façon
@ThreadSafe
public class SemaphoreOnLock {
    private final Lock lock = new ReentrantLock();
    // PRÉDICAT DE CONDITION : permitsAvailable (permits > 0)
```

1. `ReentrantLock` exige que le `Lock` soit détenu lorsque l'on appelle `signal()` ou `signalAll()`, mais les implémentations de `Lock` peuvent construire des `Condition` qui n'ont pas cette exigence.

Listing 14.12 : Semaphore implémenté à partir de Lock. (suite)

```

private final Condition permitsAvailable = lock.newCondition();
@GuardedBy("lock") private int permits;

SemaphoreOnLock(int initialPermits ) {
    lock.lock();
    try {
        permits = initialPermits;
    } finally {
        lock.unlock();
    }
}

// BLOQUE-JUSQU'À : permitsAvailable
public void acquire() throws InterruptedException {
    lock.lock();
    try {
        while (permits <= 0)
            permitsAvailable.await();
        --permits;
    } finally {
        lock.unlock();
    }
}

public void release() {
    lock.lock();
    try {
        ++permits;
        permitsAvailable.signal();
    } finally {
        lock.unlock();
    }
}
}

```

En réalité, ces deux classes sont implémentées à partir de la même classe de base, `AbstractQueuedSynchronizer` (AQS) – comme de nombreux autres synchronisateurs. AQS est un framework permettant de construire des verrous et un nombre impressionnant de synchronisateurs. `ReentrantLock` et `Semaphore` ne sont pas les seules classes construites à partir d’AQS : c’est également le cas de `CountDownLatch`, `ReentrantReadWriteLock`, `SynchronousQueue`¹ et `FutureTask`.

AQS gère la plupart des détails d’implémentation d’un synchronisateur, notamment le placement des threads en attente dans une file FIFO. Les différents synchronisateurs peuvent également définir des critères permettant de déterminer si un thread doit être autorisé à passer ou forcé d’attendre.

L’utilisation d’AQS pour construire des synchronisateurs offre plusieurs avantages : non seulement on réduit l’effort d’implémentation, mais on n’a pas à payer le prix de plusieurs points de compétition – ce qui serait le cas si l’on construisait un synchronisateur à partir d’un autre. Dans `SemaphoreOnLock`, par exemple, l’acquisition d’un permis

1. Java 6 a remplacé la classe `SynchronousQueue` qui reposait sur AQS par une version non bloquante (plus adaptable).

comprend deux endroits potentiellement bloquants – lorsque le verrou protège l'état du sémaphore et lorsqu'il n'y a plus de permis disponible. Les synchroniseurs construits avec AQS, en revanche, n'ont qu'un seul point de blocage potentiel, ce qui réduit le coût des changements de contexte et améliore le débit. AQS a été conçu pour être adaptable et tous les synchroniseurs de `java.util.concurrent` qui reposent sur ce framework bénéficient de cette propriété.

14.5 AbstractQueuedSynchronizer

La plupart des développeurs n'utiliseront sûrement jamais AQS directement car l'ensemble standard des synchroniseurs disponibles couvre un assez grand nombre de situations. Cependant, savoir comment ces synchroniseurs sont implémentés permet de mieux comprendre leur fonctionnement.

Les opérations de base d'un synchronisateur reposant sur AQS sont des variantes de `acquire()` et `release()` : l'acquisition est l'opération qui dépend de l'état et peut donc se bloquer. Avec un verrou ou un sémaphore, la signification de `acquire()` est évidente – prendre le verrou ou un permis – et l'appelant peut devoir attendre que le synchroniseur soit dans un état où cette opération peut réussir. Avec `CountDownLatch`, `acquire()` signifie "attend jusqu'à ce que le loquet ait atteint son état final" alors que, pour `FutureTask`, elle signifie "attend que la tâche se soit terminée". La méthode `release()` n'est pas une opération bloquante ; elle permet aux threads bloqués par `acquire()` de poursuivre leur exécution.

Pour qu'une classe dépende de l'état, elle doit avoir un état. AQS s'occupe d'en gérer une partie pour la classe synchroniseur : elle gère un unique entier faisant partie de l'état, qui peut être manipulé *via* les méthodes protégées `getState()`, `setState()` et `compareAndSetState()`. Cet entier peut servir à représenter un état quelconque ; `ReentrantLock` l'utilise par exemple pour représenter le nombre de fois où le thread propriétaire a pris le verrou, `Semaphore` s'en sert pour représenter le nombre de permis restants et `FutureTask`, pour indiquer l'état de la tâche (pas encore démarrée, en cours d'exécution, terminée, annulée). Les synchroniseurs peuvent également gérer eux-mêmes d'autres variables d'état ; `ReentrantLock` mémorise par exemple le propriétaire courant du verrou afin de pouvoir faire la différence entre les requêtes de verrou qui sont réentrant et celles qui sont en compétition.

Avec AQS, l'acquisition et la libération ont la forme des méthodes présentées dans le Listing 14.13. En fonction du synchroniseur, l'acquisition peut être exclusive, comme dans `ReentrantLock`, ou non exclusive, comme dans `Semaphore` et `CountDownLatch`. Une opération `acquire()` est formée de deux parties : le synchroniseur décide d'abord si l'état courant permet l'acquisition, auquel cas le thread est autorisé à poursuivre ; sinon l'opération bloque ou échoue. Cette décision est déterminée par la sémantique du

synchronisateur : l'acquisition d'un verrou, par exemple, peut réussir s'il n'est pas déjà détenu et l'acquisition d'un loquet peut réussir s'il est dans son état final.

Listing 14.13 : Formes canoniques de l'acquisition et de la libération avec AQS.

```
boolean acquire() throws InterruptedException {
    while (l'état n'autorise pas l'acquisition) {
        if (blocage de l'acquisition demandé) {
            mettre le thread courant dans la file s'il n'y est pas déjà
            bloquer le thread courant
        }
        else
            return échec
    }
    mettre éventuellement à jour l'état de la synchronisation
    sortir le thread de la file s'il était dans la file
    return succès
}

void release() {
    mettre à jour l'état de la synchronisation
    if (le nouvel état peut permettre l'acquisition à un thread bloqué)
        débloquer un ou plusieurs threads de la file
}
```

La seconde partie implique de modifier éventuellement l'état du synchronisateur ; un thread prenant le synchronisateur peut influer sur le fait que d'autres threads puissent le prendre. Acquérir un verrou, par exemple, fait passer l'état du verrou de "libre" à "détenue" et prendre un permis d'un Semaphore réduit le nombre de permis restants. En revanche, l'acquisition d'un loquet par un thread n'affecte pas le fait que d'autres threads puissent le prendre : la prise d'un loquet ne modifie donc pas son état.

Un synchronisateur qui autorise une acquisition exclusive devrait implémenter les méthodes protégées `tryAcquire()`, `tryRelease()` et `isHeldExclusively()` ; ceux qui reconnaissent l'acquisition partagée devraient implémenter `tryAcquireShared()` et `tryReleaseShared()`. Les méthodes `acquire()`, `acquireShared()`, `release()` et `releaseShared()` d'AQS appellent les formes `try` de ces méthodes dans la sous-classe synchronisateur afin de déterminer si l'opération peut avoir lieu. Cette sous-classe peut utiliser `getState()`, `setState()` et `compareAndSetState()` pour examiner et modifier l'état en fonction de la sémantique de ses méthodes `acquire()` et `release()` et se sert du code de retour pour informer la classe de base de la réussite ou non de la tentative d'acquérir ou de libérer le synchronisateur. Si `tryAcquireShared()` renvoie une valeur négative, par exemple, cela signifie que l'acquisition a échoué ; une valeur nulle indique que le synchronisateur a été pris de façon exclusive et une valeur positive, qu'il a été pris de façon non exclusive. Les méthodes `tryRelease()` et `tryReleaseShared()` devraient renvoyer `true` si la libération a pu débloquer des threads attendant de prendre le synchronisateur.

Pour simplifier l'implémentation de verrous supportant les files d'attente de condition (comme `ReentrantLock`), AQS fournit également toute la mécanique permettant de construire des variables conditions associées à des synchronisateurs.

14.5.1 Un loquet simple

La classe `OneShotLatch` du Listing 14.14 est un loquet binaire implémenté à l'aide d'AQS. Il dispose de deux méthodes publiques, `await()` et `signal()`, qui correspondent respectivement à l'acquisition et à la libération. Initialement, le loquet est fermé ; tout thread appelant `await()` se bloque jusqu'à ce que le loquet s'ouvre. Lorsque le loquet est ouvert par un appel à `signal()`, les threads en attente sont libérés et ceux qui arrivent ensuite sont autorisés à passer.

Listing 14.14 : Loquet binaire utilisant `AbstractQueuedSynchronizer`.

```
@ThreadSafe
public class OneShotLatch {
    private final Sync sync = new Sync();

    public void signal() { sync.releaseShared(0); }

    public void await() throws InterruptedException {
        sync.acquireSharedInterruptibly (0);
    }

    private class Sync extends AbstractQueuedSynchronizer {
        protected int tryAcquireShared(int ignored) {
            // Réussit si le loquet est ouvert (state == 1), sinon échoue
            return (getState() == 1) ? 1 : -1;
        }

        protected boolean tryReleaseShared(int ignored) {
            setState(1); // Le loquet est maintenant ouvert
            return true; // Les autres threads peuvent maintenant passer
        }
    }
}
```

Dans `OneShotLatch`, l'état de l'AQS contient l'état du loquet – fermé (zéro) ou ouvert (un). La méthode `await()` appelle `acquireSharedInterruptibly()` d'AQS, qui, à son tour, consulte la méthode `tryAcquireShared()` de `OneShotLatch`. L'implémentation de `tryAcquireShared()` doit renvoyer une valeur indiquant si l'acquisition peut ou non avoir lieu. Si le loquet a déjà été ouvert, `tryAcquireShared()` renvoie une valeur indiquant le succès de l'opération, ce qui autorise le thread à passer ; sinon elle renvoie une valeur signalant que la tentative d'acquisition a échoué. Dans ce dernier cas, `acquireShared Interruptibly()` place le thread dans la file des threads en attente. De même, `signal()` appelle `releaseShared()`, qui provoque la consultation de `tryReleaseShared()`. L'implémentation de `tryReleaseShared()` fixe inconditionnellement l'état du loquet à "ouvert" et indique (*via* sa valeur de retour) que le synchronisateur est dans un état totalement libre. Ceci force AQS à laisser tous les threads en attente tenter de reprendre

le synchronisateur, et l'acquisition réussira alors car `tryAcquireShared()` renvoie une valeur indiquant le succès de l'opération.

`OneShotLatch` est un synchronisateur totalement fonctionnel et performant, implémenté en une vingtaine de lignes de code seulement. Il manque bien sûr quelques fonctionnalités utiles, comme une acquisition avec délai d'expiration ou la possibilité d'inspecter l'état du loquet, mais elles sont relativement faciles à implémenter car AQS fournit des versions avec délais des méthodes d'acquisition, ainsi que des méthodes utilitaires pour les opérations d'inspection classiques.

`OneShotLatch` aurait pu être réalisée en étendant AQS plutôt qu'en lui déléguant des opérations, mais ce n'est pas souhaitable pour plusieurs raisons [EJ Item 14]. Notamment, cette pratique détériorerait l'interface simple (deux méthodes seulement) de `OneShotLatch` et, bien que les méthodes publiques d'AQS ne permettent pas aux appelants de perturber l'état du loquet, ceux-ci pourraient aisément les utiliser indirectement. Aucun des synchronisateurs de `java.util.concurrent` ne dérive directement d'AQS – ils délèguent tous leurs opérations à des sous-classes internes privées d'AQS.

14.6 AQS dans les classes de `java.util.concurrent`

De nombreuses classes bloquantes de `java.util.concurrent`, comme `ReentrantLock`, `Semaphore`, `ReentrantReadWriteLock`, `CountDownLatch`, `SynchronousQueue` et `FutureTask`, sont construites à partir d'AQS. Sans aller trop loin dans les détails (le code source est inclus dans le téléchargement¹ du JDK), étudions rapidement comment chacune de ces classes tire parti d'AQS.

14.6.1 `ReentrantLock`

`ReentrantLock` ne supporte que l'acquisition exclusive et implémente donc `tryAcquire()`, `tryRelease()` et `isHeldExclusively()` ; le Listing 14.15 montre la version non équitable de `tryAcquire()`. `ReentrantLock` utilise l'état de la synchronisation pour mémoriser le nombre d'acquisitions de verrous et gère une variable `owner` contenant l'identité du thread propriétaire, qui n'est modifiée que lorsque le thread courant vient de prendre le verrou ou qu'il va le libérer². Dans `tryRelease()`, on vérifie le champ `owner` pour s'assurer que le thread courant possède le verrou avant d'autoriser une libération ; dans

1. Ou, avec moins de contraintes de licence, à partir de <http://gee.cs.oswego.edu/dl/concurrency-interest>.

2. Les méthodes protégées de manipulation de l'état ont la sémantique mémoire d'une lecture ou d'une écriture de volatile et `ReentrantLock` prend soin de ne lire le champ `owner` qu'après avoir appelé `getState()` et de l'écrire avant l'appel de `setState()` ; `ReentrantLock` peut englober la sémantique mémoire de l'état de synchronisation et ainsi éviter une synchronisation supplémentaire – voir la section 16.1.4.

tryAcquire(), ce champ sert à différencier une acquisition réentrant d'une tentative de prise de verrou avec compétition.

Listing 14.15 : Implémentation de tryAcquire() pour un ReentrantLock non équitable.

```
protected boolean tryAcquire(int ignored) {
    final Thread current = Thread.currentThread();
    int c = getState();
    if (c == 0) {
        if (compareAndSetState(0, 1)) {
            owner = current;
            return true;
        }
    } else if (current == owner) {
        setState(c+1);
        return true;
    }
    return false;
}
```

Lorsqu'un thread tente de prendre un verrou, tryAcquire() consulte d'abord l'état de ce verrou. S'il est disponible, la méthode tente de mettre à jour cet état pour indiquer qu'il est pris. Comme il pourrait avoir été modifié depuis sa consultation, tryAcquire() utilise compareAndSetState() pour essayer d'effectuer de façon atomique la modification signalant que le verrou est désormais détenu et pour confirmer que l'état n'a pas été modifié depuis sa dernière consultation (voir la description de compareAndSet() dans la section 15.3). Si l'état du verrou indique qu'il est déjà pris et qu'il appartient au thread courant, le compteur d'acquisition est incrémenté ; si le thread courant n'est pas son propriétaire, la tentative échoue.

ReentrantLock tire également parti du fait qu'AQS sait gérer les variables conditions et les ensembles en attente. Lock.newCondition() renvoie une nouvelle instance de ConditionObject, une classe interne d'AQS.

14.6.2 Semaphore et CountDownLatch

Semaphore utilise l'état de synchronisation d'AQS pour stocker le nombre de permis disponibles. La méthode tryAcquireShared() (voir Listing 14.16) commence par calculer le nombre de permis restants ; s'il n'y en a plus assez, elle renvoie une valeur indiquant que l'acquisition a échoué. S'il reste assez de permis, elle tente de diminuer de façon atomique leur nombre à l'aide de compareAndSetState(). Si l'opération réussit (ce qui signifie que le nombre de permis n'a pas été modifié depuis sa dernière consultation), elle renvoie une valeur indiquant que l'acquisition a réussi. Cette valeur de retour précise également si une autre tentative d'acquisition partagée pourrait réussir, auquel cas les autres threads en attente seront également débloqués.

Listing 14.16 : Les méthodes `tryAcquireShared()` et `tryReleaseShared()` de `Semaphore`.

```

protected int tryAcquireShared (int acquires) {
    while (true) {
        int available = getState();
        int remaining = available - acquires;
        if (remaining < 0 || compareAndSetState (available, remaining))
            return remaining;
    }
}

protected boolean tryReleaseShared (int releases) {
    while (true) {
        int p = getState();
        if (compareAndSetState (p, p + releases))
            return true;
    }
}

```

La boucle while se termine lorsqu'il n'y a plus assez de permis ou lorsque `tryAcquireShared()` peut modifier de façon atomique le nombre de permis pour refléter l'acquisition. Bien que tout appel à `compareAndSetState()` puisse échouer à cause d'une compétition avec un autre thread (voir la section 15.3), ce qui provoquera un réessai, l'un de ces deux critères de sortie deviendra vrai après un nombre raisonnable de tentatives. De même, `tryReleaseShared()` augmente le nombre de permis – ce qui débloquera éventuellement des threads en attente – en réessayant jusqu'à ce que cette mise à jour réussisse. La valeur renvoyée par `tryReleaseShared()` indique si d'autres threads pourraient avoir été débloqués par cette libération.

`CountDownLatch` utilise AQS un peu comme `Semaphore` : l'état de synchronisation contient le compte courant. La méthode `countDown()` appelle `release()`, qui décrémente ce compteur et débloque les threads en attente s'il a atteint zéro ; `await()` appelle `acquire()`, qui se termine immédiatement si le compteur vaut zéro ou se bloque dans le cas contraire.

14.6.3 FutureTask

Au premier abord, `FutureTask` ne ressemble pas à un synchronisateur. Cependant, la sémantique de `Future.get()` ressemble fortement à celle d'un loquet – si un événement (la terminaison ou l'annulation de la tâche représentée par la `FutureTask`) survient, les threads peuvent poursuivre leur exécution ; sinon ils sont mis en attente de cet événement.

`FutureTask` utilise l'état de synchronisation d'AQS pour stocker l'état de la tâche – en cours d'exécution, terminée ou annulée. Elle gère également des variables d'état supplémentaires pour mémoriser le résultat du calcul ou l'exception qu'elle a lancée. Elle utilise en outre une référence au thread qui exécute le calcul (s'il est dans l'état "en cours d'exécution") afin de pouvoir l'interrompre si la tâche est annulée.

14.6.4 ReentrantReadWriteLock

L'interface de `ReadWriteLock` suggère qu'il y a deux verrous – un verrou de lecture et un d'écriture – mais, dans son implémentation à partir d'AQS, une seule sous-classe

d’AQS gère ces deux verrouillages. Pour ce faire, `ReentrantReadWriteLock` utilise 16 bits de l’état pour stocker le compteur du verrou d’écriture et les 16 autres pour le verrou de lecture. Les opérations sur le verrou de lecture utilisent les méthodes d’acquisition et de libération partagées ; celles sur le verrou d’écriture, les versions exclusives de ces méthodes.

En interne, AQS gère une file des threads en attente et mémorise si un thread a demandé un accès exclusif ou partagé. Avec `ReentrantReadWriteLock`, lorsque le verrou devient disponible, si le verrou en tête de la file veut un accès en écriture il l’obtient ; s’il veut un accès en lecture tous les threads de la file jusqu’au premier thread écrivain l’obtiendront¹.

Résumé

Si vous devez implémenter une classe dépendante de l’état – une classe dont les méthodes doivent se bloquer si une précondition reposant sur l’état n’est pas vérifiée –, la meilleure stratégie consiste généralement à partir d’une classe existante de la bibliothèque, comme `Semaphore`, `BlockingQueue` ou `CountDownLatch`, comme on le fait pour `ValueLatch` au Listing 8.17. Cependant, ces classes ne fournissent pas toujours un point de départ suffisant ; dans ce cas, vous pouvez construire vos propres synchroniseurs en utilisant les files d’attente de condition internes, des objets `Condition` explicites ou `AbstractQueuedSynchronizer`. Les files d’attente de condition internes sont intimement liées au verrouillage interne puisque le mécanisme de gestion des dépendances vis-à-vis de l’état est nécessairement lié à celui qui garantit la cohérence de cet état. De même, les `Condition` explicites sont étroitement liées aux `Lock` explicites et offrent un ensemble de fonctionnalités plus étendu que celui de leurs homologues internes, notamment des ensembles d’attentes par verrou, des attentes de condition interruptibles ou non interruptibles, des attentes équitables ou non équitables et des attentes avec délai d’expiration.

1. Contrairement à certaines implémentations des verrous de lecture et d’écriture, ce mécanisme ne permet pas de choisir entre une politique donnant la préférence aux lecteurs et une autre donnant la préférence aux écrivains. Pour cela, la file d’attente d’AQS ne devrait pas être une file FIFO ou il faudrait deux files. Cependant, une politique d’ordonnancement aussi stricte est rarement nécessaire en pratique ; si la vivacité de la version non équitable de `ReentrantReadWriteLock` ne suffit pas, la version équitable fournit généralement un ordre satisfaisant et garantit que les lecteurs et les écrivains ne souffriront pas de famine.

Variab les atomiques et synchronisation non bloquante

De nombreuses classes de `java.util.concurrent`, comme `Semaphore` et `ConcurrentLinkedQueue`, ont de meilleures performances et une plus grande adaptabilité que les alternatives qui utilisent `synchronized`. Dans ce chapitre, nous étudierons la source principale de ce gain de performances : les variables atomiques et la synchronisation non bloquante.

L'essentiel des recherches récentes sur les algorithmes concurrents est consacré aux *algorithmes non bloquants*, qui assurent l'intégrité des données lors d'accès concurrents à l'aide d'instructions machines atomiques de bas niveau, comme *comparer-et-échanger*, au lieu de faire appel à des verrous. Ces algorithmes sont très fréquemment utilisés dans les systèmes d'exploitation et les JVM pour planifier l'exécution des threads et des processus, pour le ramasse-miettes et pour implémenter les verrous et autres structures de données concurrentes.

Ces algorithmes sont beaucoup plus compliqués à concevoir et à implémenter que ceux qui utilisent des verrous, mais ils offrent une adaptabilité et des avantages en termes de vivacité non négligeables. Leur synchronisation a une granularité plus fine et permet de réduire énormément le coût de la planification car ils ne se bloquent pas lorsque plusieurs threads concourent pour les mêmes données. En outre, ils sont immunisés contre les interblocages et autres problèmes de vivacité. Avec les algorithmes qui reposent sur les verrous, les autres threads ne peuvent pas progresser si un thread se met en sommeil ou en boucle pendant qu'il détient le verrou, alors que les algorithmes non bloquants sont imperméables aux échecs des différents threads. À partir de Java 5.0, il est possible de construire ces algorithmes efficacement en Java à l'aide des *classes de variables atomiques*, comme `AtomicInteger` et `AtomicReference`.

Les variables atomiques peuvent également servir de "meilleures variables volatiles", même si vous n'écrivez pas d'algorithme non bloquant. Elles offrent en effet la même sémantique mémoire que les variables volatiles mais lui ajoutent les modifications atomiques, ce qui en fait les solutions idéales pour les compteurs, les générateurs de séquence et les collectes de statistiques tout en offrant une meilleure adaptabilité que leurs alternatives avec verrou.

15.1 Inconvénients du verrouillage

Une synchronisation de l'accès à l'état partagé à l'aide d'un protocole de verrouillage cohérent garantit que le thread qui détient le verrou qui protège un ensemble de variables dispose d'un accès exclusif à celles-ci et que toutes les modifications qui leur sont apportées seront visibles aux threads qui prendront ensuite le verrou.

Les JVM actuelles peuvent optimiser assez efficacement les acquisitions de verrou qui ne donnent pas lieu à compétition, ainsi que leur libération, mais, si plusieurs threads demandent le verrou en même temps, la JVM a besoin de l'aide du système d'exploitation. En ce cas, le thread qui a perdu la bataille sera suspendu et devra reprendre plus tard¹. Lorsque ce thread reprend son exécution, il peut devoir attendre que les autres threads terminent leurs quanta de temps avant de pouvoir lui-même être planifié, or la suspension et la reprise d'un thread coûte cher et implique généralement une longue interruption. Pour les classes qui utilisent des verrous avec des opérations courtes (comme les classes collections synchronisées, dont la plupart des méthodes ne contiennent que quelques opérations), le rapport entre le coût de la planification et le travail utile peut être assez élevé *si le verrou est souvent disputé*.

Les variables volatiles constituent un mécanisme de synchronisation plus léger que le verrouillage car elles n'impliquent pas de changement de contexte ni de planification des threads. Cependant, elles présentent quelques inconvénients par rapport aux verrous : bien qu'elles fournissent les mêmes garanties de visibilité, elles ne peuvent pas servir à construire des actions composées atomiques. Ceci signifie que l'on ne peut pas utiliser les variables volatiles lorsqu'une variable dépend d'une autre ou que sa nouvelle valeur dépend de l'ancienne. Cet inconvénient limite l'utilisation des variables volatiles puisqu'elles ne peuvent pas servir à implémenter correctement des outils classiques comme les compteurs ou les mutex².

Bien que l'opération d'incrémentation (`++i`) puisse, par exemple, ressembler à une opération atomique, il s'agit en fait de trois opérations distinctes – récupérer la valeur

1. Une JVM intelligente n'a pas nécessairement besoin de suspendre un thread qui lutte pour obtenir un verrou : elle pourrait utiliser des données de profilage pour choisir entre une suspension et une boucle en fonction du temps pendant lequel le verrou a été détenu au cours des acquisitions précédentes.

2. Il est théoriquement possible, bien que totalement impraticable, d'utiliser la sémantique de volatile pour construire des mutex ou d'autres synchronisateurs ; voir (Raynal, 1986).

courante de la variable, lui ajouter un, puis écrire le résultat dans la variable. Pour ne pas perdre une mise à jour, toute l'opération lecture-modification-écriture doit être atomique. Pour l'instant, le seul moyen d'y parvenir que nous avons étudié consiste à utiliser un verrou, comme dans la classe `Counter` au Listing 4.1.

`Counter` est thread-safe et fonctionne bien en cas de compétition faible ou inexistante. Dans le cas contraire, cependant, les performances souffrent du coût des changements de contexte et des délais induits par la planification. Lorsque les verrous sont utilisés aussi brièvement, être mis en sommeil est une punition sévère pour avoir demandé le verrou au mauvais moment.

Le verrouillage a d'autres inconvénients. Notamment, un thread qui attend un verrou ne peut rien faire d'autre. Si un thread qui détient un verrou est retardé (à cause d'une faute de page, d'un délai de planification, etc.), aucun autre thread ayant besoin du verrou ne peut progresser. Ce problème peut être sérieux si le thread bloqué a une priorité forte alors que celui qui détient le verrou a une priorité faible – cette situation est appelée *inversion des priorités*. Même si le thread de forte priorité devrait être prioritaire, il doit attendre que le verrou soit relâché, ce qui revient à ramener sa priorité à celle du thread de priorité plus basse. Si un thread qui détient un verrou est définitivement bloqué (à cause d'une boucle sans fin, d'un interblocage, d'un livelock ou de tout autre échec de vivacité), tous les threads qui attendent le verrou ne pourront plus jamais progresser.

Même en ignorant ces problèmes, le verrouillage est tout simplement un mécanisme lourd pour des opérations courtes comme l'incrémentation d'un compteur. Il serait plus pratique de disposer d'une technique plus légère pour gérer la compétition entre les threads – quelque chose comme les variables volatiles, mais avec la possibilité de modification atomique en plus. Heureusement, les processeurs modernes nous offrent précisément ce mécanisme.

15.2 Support matériel de la concurrence

Le verrouillage exclusif est une technique *pessimiste* – elle suppose le pire (si vous ne fermez pas votre porte, les gremlins viendront et mettront tout sens dessus dessous) et ne progresse pas tant que vous ne pouvez pas lui garantir (en prenant les verrous adéquats) que d'autres threads n'interféreront pas.

Pour les opérations courtes, il existe une autre approche qui est souvent plus efficace – l'approche *optimiste*, dans laquelle on effectue une mise à jour en espérant qu'elle pourra se terminer sans interférence. Cette approche repose sur la *détection des collisions* pour déterminer s'il y a eu interférence avec d'autres parties au cours de la mise à jour, auquel cas l'opération échoue et peut (ou non) être tentée à nouveau. L'approche optimiste ressemble donc au vieux dicton "il est plus facile de se faire pardonner que de demander la permission", où "plus facile" signifie ici "plus efficace".

Les processeurs conçus pour les opérations en parallèle disposent d'instructions spéciales pour gérer les accès concurrents aux variables partagées. Les anciens processeurs disposaient des instructions *test-and-set*, *fetch-and-increment* ou *swap*, qui suffisaient à implémenter des mutex pouvant, à leur tour, servir à créer des objets concurrents plus sophistiqués. Aujourd'hui, quasiment tous les processeurs modernes ont une instruction de lecture-modification-écriture atomique, comme *compare-and-swap* ou *load-linked/store-conditional*. Les systèmes d'exploitation et les JVM les utilisent pour implémenter les verrous et les structures de données concurrentes mais, jusqu'à Java 5.0, elles n'étaient pas directement accessibles aux classes Java.

15.2.1 L'instruction *Compare-and-swap*

L'approche choisie par la plupart des architectures de processeurs, notamment IA32 et Sparc, consiste à implémenter l'instruction *compare-and-swap* (CAS) ; les autres, comme le PowerPC, obtiennent la même fonctionnalité avec une paire d'instructions, *load-linked* et *store-conditional*. L'instruction CAS porte sur trois opérandes – un emplacement mémoire *V*, l'ancienne valeur *A* et la nouvelle valeur *B*. CAS modifie de façon atomique *V* pour qu'elle reçoive la valeur *B* uniquement si la valeur contenue dans *V* correspond à l'ancienne valeur *A* ; sinon elle ne fait rien. Dans les deux cas, elle renvoie la valeur courante de *V* (la variante *compare-and-set* renvoie une valeur indiquant si l'opération a réussi). CAS signifie donc "je pense que *V* vaut *A* ; si c'est le cas, j'y place *B*, sinon je ne modifie rien car j'avais tort". CAS est donc une technique optimiste – elle effectue la mise à jour en pensant réussir et sait détecter l'échec si un autre thread a modifié la variable depuis son dernier examen. La classe `SimulatedCAS` du Listing 15.1 illustre la sémantique (mais ni l'implémentation ni les performances) de CAS.

Listing 15.1 : Simulation de l'opération CAS.

```
@ThreadSafe
public class SimulatedCAS {
    @GuardedBy("this") private int value;

    public synchronized int get() { return value; }

    public synchronized int compareAndSwap(int expectedValue ,
                                         int newValue) {
        int oldValue = value;
        if (oldValue == expectedValue )
            value = newValue;
        return oldValue;
    }

    public synchronized boolean compareAndSet(int expectedValue ,
                                              int newValue) {
        return (expectedValue == compareAndSwap(expectedValue, newValue));
    }
}
```

Lorsque plusieurs threads tentent de mettre à jour simultanément la même variable avec CAS, l'un gagne et modifie la valeur de la variable, les autres perdent. Cependant, les

perdants ne sont pas punis par une suspension, comme cela aurait été le cas s'ils avaient échoué dans la prise d'un verrou : on leur indique simplement qu'ils ont perdu la course cette fois-ci mais qu'ils peuvent réessayer. Un thread ayant échoué avec CAS n'étant pas bloqué, il peut décider quand recommencer sa tentative, effectuer une action de repli ou ne rien faire¹. Cette souplesse élimine la plupart des échecs de vivacité liés au verrouillage (bien que, dans certains cas rares, elle puisse introduire le risque de livelocks – voir la section 10.3.3).

Le patron d'utilisation typique de CAS consiste à d'abord lire la valeur *A* qui est dans *V*, à construire la nouvelle valeur *B* à partir de *A* puis à utiliser CAS pour échanger de façon atomique la valeur de *A* par *B* dans *V* si aucun autre thread n'a changé la valeur de *V* entre-temps. L'instruction CAS gère le problème de l'implémentation de la séquence lecture-modification-écriture atomique sans verrou car elle peut détecter les interférences provenant des autres threads.

15.2.2 Compteur non bloquant

La classe CasCounter du Listing 15.2 implémente un compteur thread-safe qui utilise CAS. L'opération d'incrémentation respecte la forme canonique – récupérer l'ancienne valeur, la transformer en nouvelle valeur (en lui ajoutant un) et utiliser CAS pour mettre la nouvelle valeur. Si CAS échoue, l'opération est immédiatement retentée. Les tentatives à répétition sont une stratégie raisonnable bien que, en cas de compétition très forte, il puisse être préférable d'attendre pour éviter un livelock.

Listing 15.2 : Compteur non bloquant utilisant l'instruction CAS.

```
@ThreadSafe
public class CasCounter {
    private SimulatedCAS value;

    public int getValue() {
        return value.get();
    }

    public int increment() {
        int v;
        do {
            v = value.get();
        }
        while (v != value.compareAndSwap (v, v + 1));
        return v + 1;
    }
}
```

1. Ne rien faire est une réponse tout à fait sensée à l'échec de CAS ; dans certains algorithmes non bloquants, comme celui de la file chaînée de la section 15.4.2, l'échec de CAS signifie que quelqu'un d'autre a déjà fait le travail que l'on comptait faire.

CasCounter ne se bloque pas, bien qu'il puisse devoir essayer plusieurs¹ fois si d'autres threads modifient le compteur au même moment (en pratique, si vous avez simplement besoin d'un compteur ou d'un générateur de séquence, il suffit d'utiliser `AtomicInteger` ou `AtomicLong`, qui fournissent l'incrémentation automatique et d'autres méthodes arithmétiques).

Au premier abord, le compteur qui utilise CAS semble être moins performant que le compteur utilisant le verrouillage : il y a plus d'opérations, son flux de contrôle est plus complexe et dépend d'une opération CAS apparemment compliquée. Cependant, en réalité, les compteurs qui utilisent CAS ont des performances bien supérieures dès l'apparition d'une compétition, même faible – et souvent meilleures lorsqu'il n'y en a pas. Le chemin le plus rapide pour une acquisition d'un verrou non disputé nécessite généralement au moins une opération CAS, plus d'autres opérations de maintenance liées au verrou : il faut donc faire plus de travail pour les compteurs à base de verrou dans le meilleur des cas que dans le cas normal pour les compteurs utilisant CAS. L'opération CAS réussissant la plupart du temps (avec une compétition moyenne), le processeur prédira correctement le branchement implicite dans la boucle `while`, ce qui minimisera le coût de la logique de contrôle plus complexe.

Bien que la syntaxe du langage pour le verrouillage soit compacte, ce n'est pas le cas du travail effectué par la JVM et le système d'exploitation pour gérer les verrous. Le verrouillage implique en effet l'exécution d'un code relativement complexe dans la JVM et peut également impliquer un verrouillage, une suspension de thread et des échanges de contextes au niveau du système d'exploitation. Dans le meilleur des cas, le verrouillage exige au moins une instruction CAS : si les verrous cachent CAS, ils n'économisent donc pas son coût. L'exécution de CAS à partir d'un programme, en revanche, n'implique aucun code supplémentaire dans la JVM, aucun autre appel système ni aucune activité de planification. Ce qui ressemble à un code plus long au niveau de l'application est donc en réalité un code beaucoup plus court si l'on prend en compte les activités de la JVM et du SE. L'inconvénient principal de CAS est qu'elle force l'appelant à gérer la compétition (en réessayant, en revenant en arrière ou en abandonnant) alors que le verrouillage la traite automatiquement en bloquant jusqu'à ce que le verrou devienne disponible².

Les performances de CAS varient en fonction des processeurs. Sur un système monoprocesseur, une instruction CAS s'exécute généralement en quelques cycles d'horloge puisqu'il n'y a pas besoin de synchronisation entre les processeurs. Actuellement, le coût d'une instruction CAS non disputée sur un système multiprocesseur prend de 10 à 150 cycles ; les performances de CAS évoluent rapidement et dépendent non seulement

1. Théoriquement, il pourrait devoir réessayer un nombre quelconque de fois si les autres threads continuent de gagner la course de CAS ; en pratique, ce type de famine arrive rarement.

2. En fait, le plus gros inconvénient de CAS est la difficulté à construire correctement les algorithmes qui l'utilisent.

des architectures, mais également des versions du même processeur. Ces performances continueront sûrement de s'améliorer au cours des prochaines années. Une bonne estimation est que le coût d'une acquisition "rapide" d'un verrou *non disputé* et de sa libération est environ le double de celui de CAS sur la plupart des processeurs.

15.2.3 Support de CAS dans la JVM

Comment Java fait-il pour convaincre le processeur d'exécuter une instruction CAS pour lui ? Avant Java 5.0, il n'y avait aucun moyen de le faire, à part écrire du code natif. Avec Java 5.0, un accès de bas niveau a été ajouté afin de pouvoir lancer des opérations CAS sur les `int`, les `long`, les références d'objet, que la JVM compile afin d'obtenir le plus d'efficacité possible avec le matériel sous-jacent. Sur les plates-formes qui disposent de CAS, ces instructions sont traduites dans les instructions machine appropriées ; dans le pire des cas, la JVM utilisera un verrou avec attente active si une instruction de type CAS n'est pas disponible. Ce support de bas niveau est utilisé par les classes de variables atomiques (`AtomicXxx` dans `java.util.concurrent.atomic`) afin de fournir une opération CAS efficace sur les types numériques et référence ; ces classes sont utilisées directement ou indirectement pour implémenter la plupart des classes de `java.util.concurrent`.

15.3 Classes de variables atomiques

Les variables atomiques sont des mécanismes plus fins et plus légers que les verrous et sont essentiels pour implémenter du code concurrent très performant sur les systèmes multiprocesseurs. Les variables atomiques limitent la portée de la compétition à une unique variable ; la granularité est donc la plus fine qu'il est possible d'obtenir (en supposant que votre algorithme puisse être implémenté en utilisant une granularité aussi fine). Le chemin rapide (sans compétition) pour modifier une variable atomique n'est pas plus lent que le chemin rapide pour acquérir un verrou, et il est même généralement plus rapide ; le chemin lent est toujours plus rapide que le chemin lent des verrous car il n'implique pas de suspendre et de replanifier les threads. Avec des algorithmes utilisant les variables atomiques à la place des verrous, les threads peuvent plus facilement s'exécuter sans délai et revenir plus vite dans la course s'ils ont été pris dans une compétition.

Les classes de variables atomiques sont une généralisation des variables volatiles pour qu'elles disposent d'opérations atomiques conditionnelles de type lecture-modification-écriture. `AtomicInteger` représente une valeur de type `int` et fournit les méthodes `get()` et `set()`, qui ont les mêmes sémantiques que les lectures et les écritures d'une variable `int` volatile. Elle fournit également une méthode `compareAndSet()` atomique (qui, si elle réussit, a les effets mémoire d'une lecture et d'une écriture dans une variable volatile) et les méthodes utilitaires `add()`, `increment()` et `decrement()`, toutes atomiques. Cette classe ressemble donc un peu à une classe `Counter` étendue mais offre une bien meilleure

adaptabilité en cas de compétition car elle peut exploiter directement les fonctionnalités qu'utilise le matériel sous-jacent en cas de concurrence.

Il existe douze classes de variables atomiques divisées en quatre groupes : les scalaires, les modificateurs de champs, les tableaux et les variables composées. Les variables atomiques les plus fréquemment utilisées appartiennent à la catégorie des scalaires : `AtomicInteger`, `AtomicLong`, `AtomicBoolean` et `AtomicReference`. Toutes supportent CAS ; les versions `Integer` et `Long` disposent également des opérations arithmétiques (pour simuler des variables atomiques pour les autres types primitifs, vous pouvez transtyper les valeurs `short` et `byte` en `int` ou inversement et utiliser `floatToIntBits()` ou `doubleToLongBits()` pour les nombres à virgule flottante).

Les classes de tableaux atomiques (disponibles en versions `Integer`, `Long` et `Reference`) représentent des tableaux dont les éléments peuvent être modifiés de façon atomique. Elles fournissent la sémantique des accès volatiles aux éléments du tableau, ce qui n'est pas le cas des tableaux ordinaires – un tableau `volatile` n'a une sémantique `volatile` que pour sa référence, pas pour ses éléments (les autres types de variables atomiques seront présentés dans les sections 15.4.3 et 15.4.4).

Bien que les classes atomiques scalaires héritent de `Number`, elles n'héritent pas des classes enveloppes des types primitifs comme `Integer` ou `Long`. En fait, elles ne le peuvent pas : ces classes enveloppes ne sont pas modifiables alors que les classes de variables atomiques le sont. En outre, ces dernières ne redéfinissent pas les méthodes `hashCode()` et `equals()` ; chaque instance est distincte. Comme avec la plupart des objets modifiables, il n'est donc pas conseillé d'utiliser une variable atomique comme clé d'une collection de type hachage.

15.3.1 Variables atomiques comme "volatiles améliorées"

Dans la section 3.4.2, nous avons utilisé une référence volatile vers un objet non modifiable pour mettre à jour plusieurs variables d'état de façon atomique. Cet exemple utilisait une opération composée de type *tester-puis-agir* mais, dans ce cas particulier, la situation de compétition ne posait pas de problème puisque nous ne nous soucions pas de perdre une modification de temps en temps. La plupart du temps, cependant, une telle opération poserait problème car elle pourrait compromettre l'intégrité des données. La classe `NumberRange` du Listing 4.10, par exemple, ne pourrait pas être implémentée de façon thread-safe avec une référence volatile vers un objet non modifiable pour les limites inférieure et supérieure, ni avec des entiers atomiques pour stocker ces limites. Un invariant contraignant les deux nombres qui ne peuvent pas être modifiés simultanément sans violer cet invariant, les séquences *tester-puis-agir* d'une classe d'intervalle de nombres utilisant des références volatiles ou plusieurs entiers atomiques ne seront pas thread-safe.

Nous pouvons combiner la technique utilisée pour `OneValueCache` et les références atomiques pour mettre fin à la situation de compétition en modifiant de façon atomique la référence vers un objet non modifiable contenant les limites inférieure et supérieure.

La classe CasNumberRange du Listing 15.3 utilise une `AtomicReference` vers un objet `IntPair` contenant l'état ; grâce à `compareAndSet()`, elle peut mettre à jour la limite inférieure ou supérieure sans risquer la situation de compétition de `NumberRange`.

Listing 15.3 : Préservation des invariants multivariables avec CAS.

```
public class CasNumberRange {  
    @Immutable  
    private static class IntPair {  
        final int lower; // Invariant : lower <= upper  
        final int upper;  
        ...  
    }  
    private final AtomicReference <IntPair> values =  
        new AtomicReference <IntPair>(new IntPair(0, 0));  
  
    public int getLower() { return values.get().lower; }  
    public int getUpper() { return values.get().upper; }  
  
    public void setLower(int i) {  
        while (true) {  
            IntPair oldv = values.get();  
            if (i > oldv.upper)  
                throw new IllegalArgumentException("Can't set lower to " +  
                    i + " > upper");  
            IntPair newv = new IntPair(i, oldv.upper);  
            if (values.compareAndSet(oldv, newv))  
                return;  
        }  
    }  
    // idem pour setUpper()  
}
```

15.3.2 Comparaison des performances des verrous et des variables atomiques

Pour démontrer les différences d'adaptabilité entre les verrous et les variables atomiques, nous avons construit un programme de test comparant plusieurs implémentations d'un générateur de nombres pseudo-aléatoires (GNPA). Dans un GNPA, le nombre "aléatoire" suivant étant une fonction déterministe du nombre précédent, un GNPA doit mémoriser ce nombre précédent comme faisant partie de son état.

Les Listings 15.4 et 15.5 montrent deux implémentations d'un GNPA thread-safe, l'une utilisant `ReentrantLock`, l'autre, `AtomicInteger`. Le pilote de test appelle chacune d'elles de façon répétée ; chaque itération génère un nombre aléatoire (qui lit et modifie l'état partagé `seed`) et effectue un certain nombre d'itérations "de travail" qui agissent uniquement sur des données locales au thread. Ce programme simule des opérations typiques qui incluent une partie consistant à agir sur l'état partagé et une autre qui opère sur l'état local au thread.

Listing 15.4 : Générateur de nombres pseudo-aléatoires avec ReentrantLock.

```
@ThreadSafe
public class ReentrantLockPseudoRandom extends PseudoRandom {
    private final Lock lock = new ReentrantLock(false);
    private int seed;

    ReentrantLockPseudoRandom(int seed) {
        this.seed = seed;
    }

    public int nextInt(int n) {
        lock.lock();
        try {
            int s = seed;
            seed = calculateNext(s);
            int remainder = s % n;
            return remainder > 0 ? remainder : remainder + n;
        } finally {
            lock.unlock();
        }
    }
}
```

Listing 15.5 : Générateur de nombres pseudo-aléatoires avec AtomicInteger.

```
@ThreadSafe
public class AtomicPseudoRandom extends PseudoRandom {
    private AtomicInteger seed;

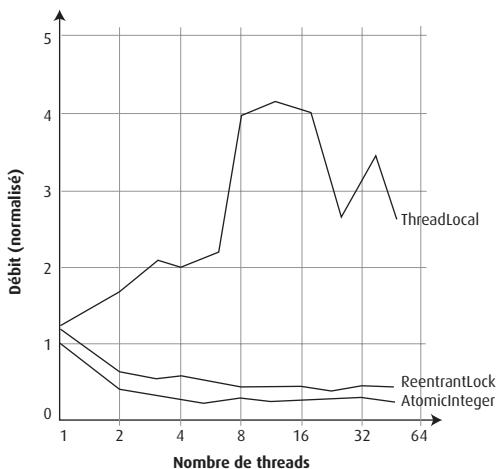
    AtomicPseudoRandom(int seed) {
        this.seed = new AtomicInteger(seed);
    }

    public int nextInt(int n) {
        while (true) {
            int s = seed.get();
            int nextSeed = calculateNext(s);
            if (seed.compareAndSet(s, nextSeed)) {
                int remainder = s % n;
                return remainder > 0 ? remainder : remainder + n;
            }
        }
    }
}
```

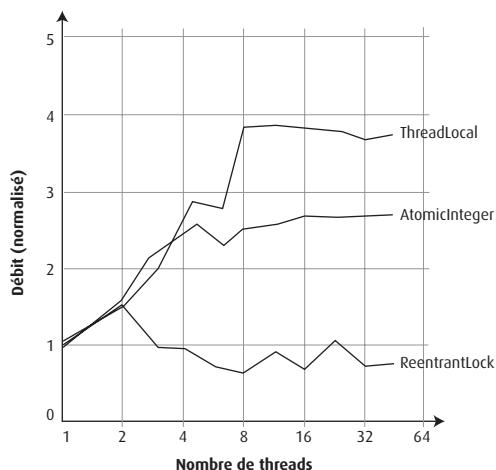
Les Figures 15.1 et 15.2 montrent le débit avec des niveaux faible et moyen de travail à chaque itération. Avec un peu de calcul local au thread, le verrou ou les variables atomiques affrontent une forte compétition ; avec plus de calcul local au thread, le verrou ou les variables atomiques rencontrent moins de compétition puisque chaque thread y accède moins souvent.

Figure 15.1

Performances de Lock et Atomic-Integer en cas de forte compétition.

**Figure 15.2**

Performances de Lock et Atomic-Integer en cas de compétition modérée.



Comme le montrent ces graphiques, à des niveaux de forte compétition le verrouillage a tendance à être plus performant que les variables atomiques mais, à des niveaux de compétition plus réalistes, ce sont ces dernières qui prennent l'avantage¹. En effet, un verrou réagit à la compétition en suspendant les threads, ce qui réduit l'utilisation processeur et le trafic de synchronisation sur le bus de la mémoire partagée (de la même façon que le blocage des producteurs dans une architecture producteur-consommateur réduit la charge sur les consommateurs et leur permet ainsi de récupérer). Avec les

1. C'est également vrai dans d'autres domaines : les feux de circulation ont un meilleur rendement en cas de fort trafic mais ce sont les ronds-points qui l'emportent en cas de trafic faible ; le schéma de compétition utilisé par les réseaux Ethernet se comporte mieux en cas de faible trafic alors que c'est le schéma du passage de jeton utilisé par Token Ring qui est le plus efficace pour un trafic réseau élevé.

variables atomiques, en revanche, la gestion de la compétition est repoussée dans la classe appelante. Comme la plupart des algorithmes qui utilisent CAS, `AtomicPseudoRandom` réagit à la compétition en réessayant immédiatement, ce qui est généralement la bonne approche mais qui, dans un environnement très disputé, ne fait qu'augmenter la compétition.

Avant de considérer que `AtomicPseudoRandom` est mal écrite ou que les variables atomiques sont un mauvais choix par rapport aux verrous, il faut se rendre compte que le niveau de compétition de la Figure 5.1 est exagérément haut : aucun programme ne se consacre exclusivement à lutter pour la prise d'un verrou ou pour l'accès à une variable atomique. En pratique, les variables atomiques s'adaptent mieux que les verrous parce qu'elles gèrent plus efficacement les niveaux typiques de compétition.

Le renversement de performances entre les verrous et les variables atomiques aux différents niveaux de compétition illustre les forces et les faiblesses de ces deux approches. Avec une compétition faible à modérée, les variables atomiques offrent une meilleure adaptabilité ; avec une forte compétition, les verrous offrent souvent un meilleur comportement (les algorithmes qui utilisent CAS sont également plus performants que les verrous sur les systèmes monoprocesseurs puisque CAS réussit toujours, sauf dans le cas peu probable où un thread est préempté au milieu d'une opération de lecture-modification-écriture).

Les Figures 15.1 et 15.2 contiennent une troisième courbe qui représente le débit d'une implémentation de `PseudoRandom` utilisant un objet `ThreadLocal` pour l'état du GNPA. Cette approche modifie le comportement de la classe – au lieu que tous les threads partagent la même séquence, chaque thread dispose de sa propre séquence privée de nombres pseudo-aléatoires – mais illustre le fait qu'il est souvent plus économique de ne pas partager du tout l'état si on peut l'éviter. Nous pouvons améliorer l'adaptabilité en gérant plus efficacement la compétition, mais une véritable adaptabilité ne peut s'obtenir qu'en éliminant totalement cette compétition.

15.4 Algorithmes non bloquants

Les algorithmes qui utilisent des verrous peuvent souffrir d'un certain nombre de problèmes de vivacité. Si un thread qui détient un verrou est retardé à cause d'une opération d'E/S bloquante, d'une faute de page ou de tout autre délai, il est possible qu'aucun autre thread ne puisse plus progresser. Un algorithme est dit *non bloquant* si l'échec ou la suspension de n'importe quel thread ne peut pas causer l'échec ou la suspension d'un autre ; un algorithme est dit *sans verrouillage* si, à chaque instant, un thread peut progresser. S'ils sont écrits correctement, les algorithmes qui synchronisent les threads en utilisant exclusivement CAS peuvent être à la fois non bloquants et sans verrouillage. Une instruction CAS non disputée réussit toujours et, si plusieurs threads luttent pour elle, il y en aura toujours un qui gagnera et pourra donc progresser. Les algorithmes non

bloquants sont également immunisés contre les interblocages ou l'inversion de priorité (bien qu'ils puissent provoquer une famine ou un livelock car ils impliquent des essais à répétition). Nous avons déjà étudié un algorithme non bloquant : CasCounter. Il existe de bons algorithmes non bloquants connus pour de nombreuses structures de données classiques comme les piles, les files, les files à priorités et les tables de hachage – leur conception est une tâche qu'il est préférable de laisser aux experts.

15.4.1 Pile non bloquante

Les algorithmes non bloquants sont considérablement plus compliqués que leurs équivalents avec verrous. La clé de leur conception consiste à trouver comment limiter la portée des modifications atomiques à une variable unique tout en maintenant la cohérence des données. Avec les classes collections comme les files, on peut parfois s'en tirer en exprimant les transformations de l'état en termes de modifications des différents liens et en utilisant un objet `AtomicReference` pour représenter chaque lien devant être modifié de façon atomique.

Les piles sont les structures de données chaînées les plus simples : chaque élément n'est lié qu'à un seul autre et chaque élément n'est désigné que par une seule référence. La classe `ConcurrentStack` du Listing 15.6 montre comment construire une pile à l'aide de références atomiques. La pile est une liste chaînée d'éléments `Node` partant de `top`, chacun d'eux contient une valeur et un lien vers l'élément suivant. La méthode `push()` prépare un nouveau `Node` dont le champ `next` pointe vers le sommet actuel de la pile (désigné par `top`), puis utilise CAS pour essayer de l'installer au sommet de la pile. Si le noeud au sommet de la pile est le même qu'au début de la méthode, l'instruction CAS réussit ; si ce noeud a changé (parce qu'un autre thread a ajouté ou ôté des éléments depuis le lancement de la méthode), CAS échoue : `push()` met à jour le nouveau noeud en fonction de l'état actuel de la pile puis réessaie. Dans les deux cas, la pile est toujours dans un état cohérent après l'exécution de CAS.

Listing 15.6 : Pile non bloquante utilisant l'algorithme de Treiber (Treiber, 1986).

```
@ThreadSafe
public class ConcurrentStack <E> {
    AtomicReference <Node<E>> top = new AtomicReference <Node<E>>();

    public void push(E item) {
        Node<E> newHead = new Node<E>(item);
        Node<E> oldHead;
        do {
            oldHead = top.get();
            newHead.next = oldHead;
        } while (!top.compareAndSet(oldHead, newHead));
    }

    public E pop() {
        Node<E> oldHead;
        Node<E> newHead;
        do {
```

Listing 15.6 : Pile non bloquante utilisant l'algorithme de Treiber (Treiber, 1986). (suite)

```
oldHead = top.get();
if (oldHead == null)
    return null;
newHead = oldHead.next;
} while (!top.compareAndSet(oldHead, newHead));
return oldHead.item;
}

private static class Node <E> {
    public final E item;
    public Node<E> next;

    public Node(E item) {
        this.item = item;
    }
}
```

CasCounter et ConcurrentStack illustrent les caractéristiques de tous les algorithmes non bloquants : une partie du traitement repose sur des suppositions et peut devoir être refait. Lorsque l'on construit l'objet Node représentant le nouvel élément dans Concurrent Stack, on espère que la valeur de la référence next sera correcte lorsqu'on l'installera sur la pile, mais on est prêt à réessayer en cas de compétition.

La sécurité vis-à-vis des threads des algorithmes non bloquants comme Concurrent Stack provient du fait que, comme le verrouillage, compareAndSet() fournit à la fois des garanties d'atomicité et de visibilité. Lorsqu'un thread modifie l'état de la pile, il le fait par un appel à compareAndSet(), qui a les effets mémoire d'une écriture volatile. Un thread voulant examiner la pile appellera get() sur le même objet AtomicReference, ce qui aura les effets mémoire d'une lecture volatile. Toutes les modifications effectuées par un thread sont donc publiées correctement vers tout autre thread qui examine l'état. La liste est modifiée par un appel à compareAndSet() qui, de façon atomique, change la référence vers le sommet ou échoue si elle détecte une interférence avec un autre thread.

15.4.2 File chaînée non bloquante

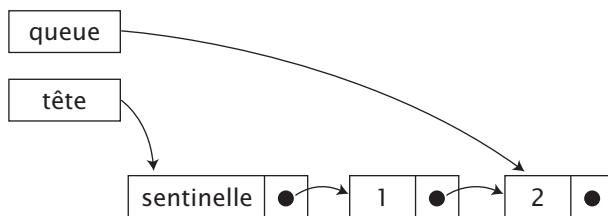
Les deux algorithmes non bloquants que nous venons d'étudier, le compteur et la pile, illustrent le modèle d'utilisation classique de l'instruction CAS pour modifier une valeur de façon spéculative, en réessayant en cas d'échec. L'astuce pour construire des algorithmes non bloquants consiste à limiter la portée des modifications atomiques à une seule variable ; c'est évidemment très simple pour les compteurs et assez facile pour les piles. En revanche, cela peut devenir bien plus compliqué pour les structures de données complexes, comme les files, les tables de hachage ou les arbres.

Une file chaînée est plus complexe qu'une pile puisqu'elle doit permettre un accès rapide à la fois à sa tête et à sa queue. Pour ce faire, elle gère deux pointeurs distincts, l'un vers sa tête, l'autre vers sa queue. Il y a donc deux pointeurs qui désignent la queue d'une file : le pointeur next de l'avant-dernier élément et le pointeur vers la queue. Pour insérer correctement un nouvel élément, ces deux pointeurs doivent donc être modifiés

simultanément, de façon atomique. Au premier abord, ceci ne peut pas être réalisé à l'aide de variables atomiques ; on a besoin d'opérations CAS distinctes pour modifier les deux pointeurs et, si la première réussit alors que la seconde échoue, la file sera dans un état incohérent. Même si elles réussissaient toutes les deux, un autre thread pourrait tenter d'accéder à la file entre le premier et le deuxième appel. La construction d'un algorithme non bloquant pour une file chaînée nécessite donc de prévoir ces deux situations.

Figure 15.3

File avec deux éléments, dans un état stable.



Nous avons pour cela besoin de plusieurs astuces. La première consiste à s'assurer que la structure de données est toujours dans un état cohérent, même au milieu d'une mise à jour formée de plusieurs étapes. De cette façon, si le thread *A* est au milieu d'une mise à jour lorsque le thread *B* arrive, *B* peut savoir qu'une opération est en cours et qu'il ne doit pas essayer immédiatement d'appliquer ses propres modifications. *B* peut alors attendre (en examinant régulièrement l'état de la file) que *A* finisse afin que chacun d'eux ne se mette pas en travers du chemin de l'autre.

Bien que cette astuce suffise pour que les threads "prennent des détours" afin d'accéder à la structure de données sans la perturber, aucun thread ne pourra accéder à la file si un thread échoue au milieu d'une modification. Pour que l'algorithme soit non bloquant, il faut donc garantir que l'échec d'un thread n'empêchera pas les autres de progresser. La seconde astuce consiste donc à s'assurer que, si *B* arrive au milieu d'une mise à jour de *A*, la structure de données contiendra suffisamment d'informations pour que *B* finisse la modification de *A*. Si *B* "aide" *A* en finissant l'opération commencée par ce dernier, *B* pourra continuer la sienne sans devoir attendre *A*. Lorsque *A* reviendra pour finir son traitement, il constatera que *B* a déjà fait le travail pour lui.

La classe `LinkedQueue` du Listing 15.7 montre la partie consacrée à l'insertion dans l'algorithme non bloquant de Michael-Scott (Michael et Scott, 1996) pour les files chaînées, qui est celui utilisé par `ConcurrentLinkedQueue`. Comme dans de nombreux algorithmes sur les files, une file vide est formée d'un nœud "sentinelle" ou "bidon" et les pointeurs de tête et de queue pointent initialement vers cette sentinelle. Le pointeur de queue désigne toujours la sentinelle (si la file est vide), le dernier élément ou (lorsqu'une opération est en cours) l'avant-dernier. La Figure 15.3 montre une file avec deux éléments, dans un état normal ou *stable*.

L'insertion d'un nouvel élément implique deux modifications de pointeurs. La première lie le nouveau nœud à la fin de la liste en modifiant le pointeur `next` du dernier

élément courant ; la seconde déplace le pointeur de queue pour qu'il désigne le nouvel élément ajouté. Entre ces deux opérations, la file est dans l'état *intermédiaire* représenté à la Figure 15.4. Après la seconde modification, elle est à nouveau dans l'état stable de la Figure 15.5.

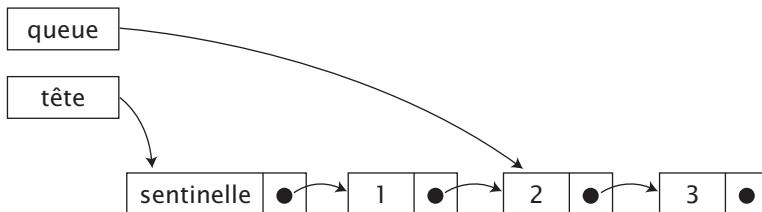


Figure 15.4

File dans un état intermédiaire pendant l'insertion.

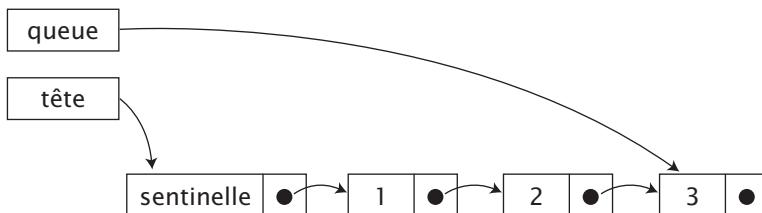


Figure 15.5

File à nouveau dans un état stable après l'insertion.

Le point essentiel qui permet à ces deux astuces de fonctionner est que, si la file est dans un état stable, le champ `next` du nœud pointé par `tail` vaut `null` alors qu'il est différent de `null` si elle est dans un état intermédiaire. Un thread peut donc immédiatement connaître l'état de la file en consultant `tail.next`. En outre, lorsqu'elle est dans un état intermédiaire, la file peut être restaurée dans un état stable en avançant le pointeur `tail` d'un nœud vers l'avant, ce qui termine l'insertion d'un élément pour un thread qui se trouve au milieu de l'opération¹.

Listing 15.7 : Insertion dans l'algorithme non bloquant de Michael-Scott (Michael et Scott, 1996).

```
@ThreadSafe
public class LinkedQueue <E> {
    private static class Node <E> {
        final E item;
        final AtomicReference <Node<E>> next;
```

1. Pour une étude complète de cet algorithme, voir (Michael et Scott, 1996) ou (Herlihy et Shavit, 2006).

```

public Node<E> item, Node<E> next) {
    this.item = item;
    this.next = new AtomicReference <Node<E>>(next);
}
}

private final Node<E> sentinelle = new Node<E>(null, null);
private final AtomicReference <Node<E>> head
    = new AtomicReference <Node<E>>(sentinelle);
private final AtomicReference <Node<E>> tail
    = new AtomicReference <Node<E>>(sentinelle);

public boolean put(E item) {
    Node<E> newNode = new Node<E>(item, null);
    while (true) {
        Node<E> curTail = tail.get();
        Node<E> tailNext = curTail.next.get();
        if (curTail == tail.get()) {
            if (tailNext != null) { Ⓐ
                // La file est dans un état intermédiaire, on avance tail
                tail.compareAndSet(curTail, tailNext); Ⓑ
            } else {
                // Dans l'état stable, on tente d'insérer le nouveau noeud
                if (curTail.next.compareAndSet(null, newNode)) { Ⓒ
                    // Insertion réussie, on tente d'avancer tail
                    tail.compareAndSet(curTail, newNode); Ⓓ
                    return true;
                }
            }
        }
    }
}
}

```

Avant de tenter d'insérer un nouvel élément, `LinkedQueue.put()` teste d'abord si la file est dans l'état intermédiaire (étape A). Si c'est le cas, cela signifie qu'un autre thread est déjà en train d'insérer un élément (il est entre les étapes C et D). Au lieu d'attendre qu'il finisse, le thread courant l'aide en finissant l'opération pour lui, ce qui consiste à avancer le pointeur `tail` (étape B). Puis il revérifie au cas où un autre thread aurait commencé à ajouter un nouvel élément, en avançant le pointeur `tail` jusqu'à trouver la file dans un état stable, afin de pouvoir commencer sa propre insertion.

L'instruction CAS de l'étape C, qui lie le nouveau nœud à la queue de la file, pourrait échouer si deux threads tentent d'ajouter un élément simultanément. Dans ce cas, il n'y aura aucun problème : aucune modification n'a été faite et il suffit que le thread courant recharge le pointeur `tail` et réessaie. Lorsque C a réussi, l'insertion est considérée comme effectuée ; la seconde instruction CAS (étape D) est du "nettoyage" puisqu'elle peut être effectuée soit par le thread qui insère, soit par un autre. Si elle échoue, le thread qui insère repart plutôt que retenter d'exécuter CAS : il n'y a pas besoin de réessayer puisqu'un autre thread a déjà fini le travail lors de son étape B ! Ceci fonctionne parce qu'avant de tenter de lier un nouveau nœud dans la file tout thread vérifie que la file nécessite un nettoyage en testant que `tail.next` ne vaut pas `null`. Dans ce cas, il avance d'abord le pointeur `tail` (éventuellement plusieurs fois) jusqu'à ce que la file soit dans l'état stable.

15.4.3 Modificateurs atomiques de champs

Bien que le Listing 15.7 illustre l'algorithme utilisé par `ConcurrentLinkedQueue`, la véritable implémentation est un peu différente. Comme le montre le Listing 15.8, au lieu de représenter chaque `Node` par une référence atomique, `ConcurrentLinkedQueue` utilise une référence volatile ordinaire et la met à jour grâce à la classe `AtomicReferenceFieldUpdater`, qui utilise l'introspection.

Listing 15.8 : Utilisation de modificateurs atomiques de champs dans `ConcurrentLinkedQueue`.

```
private class Node<E> {
    private final E item;
    private volatile Node<E> next;
    public Node(E item) {
        this.item = item;
    }
}

private static AtomicReferenceFieldUpdater<Node, Node> nextUpdater
    = AtomicReferenceFieldUpdater.newUpdater(Node.class, Node.class, "next");
```

Les classes modificateurs atomiques de champs (disponibles en versions `Integer`, `Long` et `Reference`) présentent une vue "introspective" d'un champ volatile afin que CAS puisse être utilisée sur des champs volatiles. Elles n'ont pas de constructeur : pour créer un modificateur, il faut appeler la méthode fabrique `newUpdater()`. Ces classes ne sont pas liées à une instance spécifique ; on peut utiliser un modificateur pour mettre à jour le champ de n'importe quelle instance de la classe cible. Les garanties d'atomicité pour les classes modificateurs sont plus faibles que pour les classes atomiques normales car on ne peut pas garantir que les champs sous-jacents ne seront pas modifiés directement – les méthodes `compareAndSet()` et arithmétiques ne garantissent l'atomicité que par rapport aux autres threads qui utilisent les méthodes du modificateur atomique de champ.

Dans `ConcurrentLinkedQueue`, les mises à jour du champ `next` d'un `Node` sont appliquées à l'aide de la méthode `compareAndSet()` de `nextUpdater`. On utilise cette approche quelque peu tortueuse uniquement pour des raisons de performances. Pour les objets souvent alloués, ayant une durée de vie courte (comme les nœuds d'une liste chaînée), l'élimination de la création d'un objet `AtomicReference` pour `Node` permet de réduire significativement le coût des opérations d'insertion. Toutefois, dans quasiment tous les cas, les variables atomiques ordinaires suffisent – les modificateurs atomiques ne sont nécessaires que dans quelques situations (ils sont également utiles pour effectuer des mises à jour atomiques tout en préservant la forme sérialisée d'une classe existante).

15.4.4 Le problème ABA

Le problème ABA est une anomalie pouvant résulter d'une utilisation naïve de *comparer-et-échanger* dans des algorithmes où les noeuds peuvent être recyclés (essentiellement dans des environnements sans ramasse-miettes). En réalité, une opération CAS pose la

question "est-ce que la valeur de *V* est toujours *A* ?" et effectue la mise à jour si c'est le cas. Dans la plupart des situations, dont celles des exemples de ce chapitre, c'est entièrement suffisant. Cependant, on veut parfois plutôt demander "est-ce que la valeur de *V* a été modifiée depuis que j'ai constaté qu'elle valait *A* ?". Pour certains algorithmes, changer la valeur de *V* de *A* en *B* puis en *A* compte quand même comme une modification qui nécessite de réessayer une étape de l'algorithme.

Ce *problème ABA* peut se poser dans les algorithmes qui effectuent leur propre gestion mémoire pour les nœuds de la liste chaînée. Dans ce cas, que la tête d'une liste désigne toujours un nœud déjà observé ne suffit pas à impliquer que le contenu de cette liste n'a pas été modifié. Si vous ne pouvez pas éviter le problème ABA en laissant le ramasse-miettes gérer les nœuds pour vous, il reste une solution relativement simple : au lieu de modifier la valeur d'une référence, modifiez deux valeurs – une référence et un numéro de version. Même si la valeur passe de *A* à *B*, puis de nouveau à *A*, les numéros de version seront différents. `AtomicStampedReference` et sa cousine `AtomicMarkableReference` fournissent une mise à jour atomique conditionnelle de deux variables. `AtomicStampedReference` modifie une paire référence d'objet-entier, ce qui permet d'avoir des références "avec numéros de versions", immunisées¹ contre le problème ABA. De même, `AtomicMarkableReference` modifie une paire référence d'objet-booléen, ce qui permet à certains algorithmes de laisser un nœud dans une liste bien qu'il soit marqué comme supprimé².

Résumé

Les algorithmes non bloquants gèrent la sécurité par rapport aux threads en utilisant des primitives de bas niveau, comme *comparer-et-échanger*, à la place des verrous. Ces primitives sont exposées via les classes de variables atomiques, qui peuvent également servir de "meilleures variables volatiles" en fournissant des opérations de mises à jour atomiques pour les entiers et les références d'objets.

Ces algorithmes sont difficiles à concevoir et à implémenter, mais peuvent offrir une meilleure adaptabilité dans des situations typiques et une plus grande résistance contre les échecs de vivacité. La plupart des avancées d'une version à l'autre de la JVM en termes de performances de la concurrence proviennent de l'utilisation de ces algorithmes, à la fois dans la JVM et dans la bibliothèque standard.

1. En pratique, en tout cas... car, théoriquement, le compteur pourrait reboucler.

2. De nombreux processeurs disposent d'une instruction CAS double (CAS2 ou CASX) qui peut porter sur une paire pointeur-entier, ce qui rendrait cette opération assez efficace. À partir de Java 6, `AtomicStampedReference` n'utilise pas ce CAS double, même sur les plates-formes qui en disposent (le CAS double est différent de DCAS, qui agit sur deux emplacements mémoire distincts – cette dernière n'existe actuellement sur aucun processeur).

Le modèle mémoire de Java

Tout au long de ce livre, nous avons évité les détails de bas niveau du modèle mémoire de Java (MMJ) pour nous intéresser aux problèmes de conception qui se situent à un niveau supérieur, comme la publication correcte et la spécification et la conformité aux politiques de synchronisation. Pour utiliser efficacement tous ces mécanismes, il peut être intéressant de comprendre comment ils fonctionnent, sachant que toutes leurs fonctionnalités prennent racine dans le MMJ. Ce chapitre lève donc le voile sur les exigences et les garanties de bas niveau du modèle mémoire de Java et révèle le raisonnement qui se cache derrière certaines des règles de conception que nous avons utilisées dans cet ouvrage.

16.1 Qu'est-ce qu'un modèle mémoire et pourquoi en a-t-on besoin ?

Supposons qu'un thread affecte une valeur à la variable `aVariable` :

```
aVariable = 3;
```

Un modèle mémoire répond à la question suivante : "Sous quelles conditions un thread qui lit `aVariable` verra la valeur 3 ? " Cette question peut sembler un peu idiote mais, en l'absence de synchronisation, il existe un certain nombre de raisons pour lesquelles un thread pourrait ne pas voir immédiatement (ou ne jamais voir) le résultat d'une opération effectuée par un autre thread. Les compilateurs peuvent, en effet, produire des instructions dans un ordre qui est différent de celui "évident" suggéré par le code source ou stocker des variables dans des registres au lieu de les mettre en mémoire ; les processeurs peuvent exécuter des instructions en parallèle ou dans n'importe quel ordre, les caches peuvent modifier l'ordre dans lequel les écritures dans les variables sont transmises à la mémoire principale, et les valeurs stockées dans les caches locaux d'un processeur peuvent ne pas être visibles par les autres processeurs. Tous ces facteurs peuvent donc empêcher un thread de voir la dernière valeur en date d'une variable et provoquer des

actions mémoire dans d'autres threads qui sembleront totalement mélangées – si l'on n'utilise pas une synchronisation adéquate.

Dans un contexte monothread, tous les tours que joue l'environnement à votre programme sont cachés et n'ont d'autres effets que d'accélérer son exécution. La spécification du langage Java exige que la JVM ait une sémantique *apparemment séquentielle au sein d'un thread* : tant que le programme a le même résultat que s'il était exécuté dans un environnement strictement séquentiel, toutes les astuces sont permises. C'est une bonne chose car ces réarrangements sont responsables de l'essentiel des améliorations des performances de calcul de ces dernières années. L'accroissement des fréquences d'horloge a certainement contribué à augmenter les performances, mais le parallélisme accru – les unités d'exécution superscalaires en pipelines, la planification dynamique des instructions, l'exécution spéculative et les caches mémoire multiniveaux – y a également sa part. À mesure que les processeurs deviennent plus sophistiqués, les compilateurs doivent aussi évoluer en réarrangeant les instructions pour faciliter une exécution optimale et en utilisant des algorithmes évolués d'allocation des registres. Les constructeurs de processeurs se tournent désormais vers les processeurs multicœurs, en grande partie parce qu'il devient difficile d'augmenter encore les fréquences d'horloge, le parallélisme matériel ne fera qu'augmenter.

Dans un environnement multithread, l'illusion de la séquentialité ne peut être entretenue sans que cela ait un coût significatif pour les performances. La plupart du temps, les threads d'une application concurrente font, chacun, "leurs propres affaires" et une synchronisation excessive entre eux ne ferait que ralentir l'application. Ce n'est que lorsque plusieurs threads partagent des données qu'il faut coordonner leurs activités, et la JVM se fie au programme pour identifier cette situation.

Le MMJ précise les garanties minimales que la JVM doit prendre sur le moment où les écritures dans les variables deviennent visibles aux autres threads. Il a été conçu pour équilibrer le besoin de prévisibilité et de facilité de développement des programmes par rapport aux réalités de l'implémentation de JVM performantes sur un grand nombre d'architectures processeurs connues. Cela étant dit, lorsque l'on n'est pas habitué aux astuces employées par les processeurs et les compilateurs modernes pour améliorer au maximum les performances des programmes, certains aspects du MMJ peuvent sembler troublants à première vue.

16.1.1 Modèles mémoire des plates-formes

Dans une architecture multiprocesseur à mémoire partagée, chaque processeur a son propre cache, qui est périodiquement synchronisé avec la mémoire principale. Les architectures des processeurs fournissent différents degrés de *cohérence des caches* ; certaines offrent des garanties minimales qui autorisent plusieurs processeurs à voir, quasiment en permanence, des valeurs différentes pour le même emplacement mémoire. Le système d'exploitation, le compilateur et l'environnement d'exécution (et parfois également le

programme) doivent combler les différences entre ce qui est fourni par le matériel et ce qu'exige la sécurité par rapport aux threads.

S'assurer que chaque processeur sache ce que font en permanence les autres est une opération coûteuse. La plupart du temps, cette information n'étant pas nécessaire, les processeurs assouplissent leurs garanties sur la cohérence mémoire afin d'améliorer les performances. Un *modèle mémoire* d'architecture indique aux programmes les garanties qu'ils peuvent attendre du système de mémoire et précise les instructions spéciales (*barrières mémoires*) qui sont nécessaires pour disposer de garanties supplémentaires de coordination de la mémoire lors du partage des données. Pour protéger le développeur Java des différences existant entre ces modèles mémoire en fonction des architectures, Java fournit son propre modèle mémoire et la JVM traite les différences entre le MMJ et le modèle mémoire de la plate-forme sous-jacente en insérant des barrières mémoires aux endroits adéquats.

Une vision pratique de l'exécution d'un programme consiste à imaginer que ses opérations ne peuvent se dérouler que dans un seul ordre, quel que soit le processeur sur lequel il s'exécute, et que chaque lecture de variable verra la dernière valeur qui y a été écrite par n'importe quel processeur. Ce modèle idéal, bien qu'irréaliste, est appelé *cohérence séquentielle*. Les développeurs la supposent souvent à tort car aucun système multiprocesseur actuel n'offre la cohérence séquentielle, pas plus que le MMJ. Le modèle de traitement séquentiel classique, celui de Von Neumann, n'est qu'une vague approximation du comportement des multiprocesseurs modernes.

Il en résulte que les systèmes multiprocesseurs à mémoire partagée actuels (et les compilateurs) peuvent faire des choses surprenantes lorsque des données sont partagées entre les threads, sauf si on leur précise de ne pas utiliser directement les barrières mémoires. Heureusement, les programmes Java n'ont pas besoin d'indiquer les emplacements de ces barrières : ils doivent simplement savoir à quels moments on accède à l'état partagé et utiliser correctement la synchronisation.

16.1.2 Réorganisation

Lorsque nous avons décrit les situations de compétition (*data race*) et les échecs d'atomicité au Chapitre 2, nous avons utilisé des diagrammes d'actions décrivant les "timings malheureux", où l'entrelacement des opérations produisait des résultats incorrects dans les programmes mal synchronisés. Mais il y a pire : le MMJ peut permettre à des actions de sembler s'exécuter dans un ordre différent en fonction des différents threads, ce qui complique d'autant plus le raisonnement sur leur enchaînement en l'absence de synchronisation. Les diverses raisons pour lesquelles les opérations pourraient être retardées ou sembler s'exécuter dans n'importe quel ordre peuvent toutes être rassemblées dans la catégorie générale de la *réorganisation*.

La classe `PossibleReordering` du Listing 16.1 illustre la difficulté de comprendre les programmes concurrents les plus simples lorsqu'ils ne sont pas correctement synchronisés. Il est assez facile d'imaginer comment `PossibleReordering` pourrait afficher (1, 0), (0, 1) ou (1, 1) : le thread A pourrait s'exécuter jusqu'à la fin avant que le thread B ne commence, B pourrait s'exécuter jusqu'à la fin avant que A ne démarre ou leurs actions pourraient s'entrelacer. Mais, étrangement, `PossibleReordering` peut également afficher (0, 0) ! Les actions de chaque thread n'ayant aucune dépendance entre elles peuvent donc s'exécuter dans n'importe quel ordre (même si elles étaient exécutées dans l'ordre, le timing avec lequel les caches sont vidés en mémoire peut faire, du point de vue de B, que les affectations qui ont lieu dans A apparaissent dans l'ordre opposé). La Figure 16.1 montre un entrelacement possible avec réorganisation qui produit l'affichage de (0, 0).

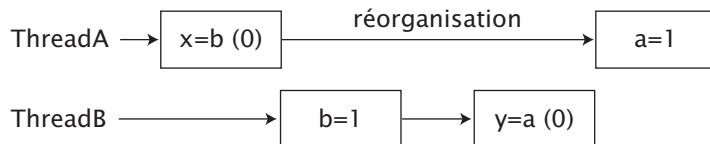
Listing 16.1 : Programme mal synchronisé pouvant produire des résultats surprenants. Ne le faites pas.

```
public class PossibleReordering {
    static int x = 0, y = 0;
    static int a = 0, b = 0;

    public static void main(String[] args) throws InterruptedException {
        Thread one = new Thread(new Runnable() {
            public void run() {
                a = 1;
                x = b;
            }
        });
        Thread other = new Thread(new Runnable() {
            public void run() {
                b = 1;
                y = a;
            }
        });
        one.start(); other.start();
        one.join(); other.join();
        System.out.println("(" + x + "," + y + ")");
    }
}
```



Figure 16.1
Entrelacement montrant une réorganisation dans `PossibleReordering`.



Bien que `PossibleReordering` soit un programme trivial, il est étonnamment difficile d'énumérer tous ses résultats possibles. La réorganisation au niveau de la mémoire peut provoquer des comportements inattendus des programmes, et il est excessivement difficile de comprendre cette réorganisation en l'absence de synchronisation ; il est bien plus facile de s'assurer que le programme est correctement synchronisé. La synchronisation empêche le compilateur, l'environnement d'exécution et le matériel de réorganiser

les opérations mémoire d'une façon qui violerait les garanties de visibilité assurées par le MMJ¹.

16.1.3 Le modèle mémoire de Java en moins de cinq cents mots

Le modèle mémoire de Java est spécifié en termes d'actions, qui incluent les lectures et les écritures de variables, les verrouillages et déverrouillages des moniteurs et le lancement et l'attente des threads. Le MMJ définit un ordre partiel² appelé *arrive-avant* sur toutes les actions d'un programme. Pour garantir que le thread qui exécute l'action *B* puisse voir le résultat de l'action *A* (que *A* et *B* s'exécutent ou non dans des threads différents), il doit exister une relation *arrive-avant* entre *A* et *B*. En son absence, la JVM est libre de les réorganiser selon son bon vouloir.

Une *situation de compétition (data race)* intervient lorsqu'une variable est lue par plusieurs threads et écrite par au moins un thread alors que les lectures et les écritures ne sont pas ordonnées par *arrive-avant*. Un programme *correctement synchronisé* n'a pas de situation de compétition et présente une cohérence séquentielle, ce qui signifie que toutes les actions du programme sembleront s'exécuter dans un ordre fixe et global.

- Les règles de *arrive-avant* sont : **Règle de l'ordre des programmes.** Toute action d'un thread *arrive-avant* toute autre action de ce thread placée après dans l'ordre du programme.
- **Règle des verrous de moniteurs.** Le déverrouillage d'un verrou de moniteur *arrive-avant* tout verrouillage ultérieur sur ce même verrou³.
- **Règle des variables volatiles.** Une écriture dans un champ volatile *arrive-avant* toute lecture de ce champ⁴.
- **Règle de lancement des threads.** Un appel à `Thread.start()` sur un thread *arrive-avant* toute action dans le thread lancé.
- **Règle de terminaison des threads.** Toute action d'un thread *arrive-avant* qu'un autre thread ne détecte que ce thread s'est terminé, soit en revenant avec succès de `Thread.join`, soit par un appel à `Thread.isAlive()` qui renvoie `false`.

1. Sur la plupart des architectures de processeurs connues, le modèle mémoire est suffisamment fort pour que le coût en termes de performances d'une lecture volatile soit équivalent à celui d'une lecture non volatile.

2. Un ordre partiel $<$ est une relation antisymétrique, réflexive et transitive sur un ensemble, mais deux éléments quelconques x et y ne doivent pas nécessairement vérifier $x < y$ ou $y < x$. On utilise tous les jours un ordre partiel pour exprimer nos préférences : on peut préférer les sushis aux cheeseburgers et Mozart à Mahler, bien que nous n'ayions pas nécessairement une préférence marquée entre les cheeseburgers et Mozart.

3. Les verrouillages et déverrouillages des objets Lock explicites ont la même sémantique mémoire que les verrous internes.

4. Les lectures et écritures des variables atomiques ont la même sémantique mémoire que les variables volatiles.

- **Règle des interruptions.** Un thread appelant `interrupt()` sur un autre thread *arrive-avant* que le thread interrompu ne détecte l'interruption (soit par le lancement de l'exception `InterruptedException`, soit en appelant `isInterrupted()` ou `interrupted()`).
- **Règle des finalisateurs.** La fin d'un constructeur d'objet *arrive-avant* le lancement de son finalisateur.
- **Transitivité.** Si *A arrive-avant B* et que *B arrive-avant C*, alors, *A arrive-avant C*.

Bien que les actions ne soient que partiellement ordonnées, les actions de synchronisation – acquisition et libération des verrous, lectures et écritures des variables volatiles – le sont totalement. On peut donc décrire *arrive-avant* en termes de prises de verrous et de lectures de variables volatiles "ultérieures".

La Figure 16.2 illustre la relation *arrive-avant* lorsque deux threads se synchronisent à l'aide d'un verrou commun. Toutes les actions dans le thread A sont ordonnées par le programme selon la règle de l'ordre des programmes, tout comme les actions du thread B.

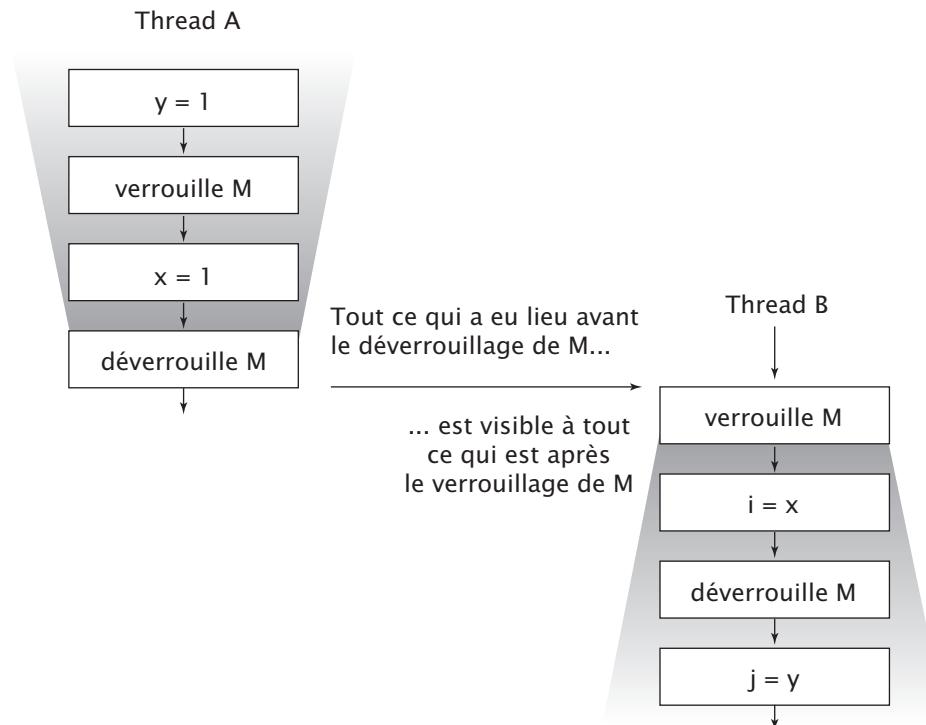


Figure 16.2

Illustration de *arrive-avant* dans le modèle mémoire de Java.

Le thread *A* relâchant le verrou *M* et *B* le prenant ensuite, toutes les actions de *A* avant sa libération du verrou sont donc ordonnées avant les actions de *B* après qu'il a pris le verrou. Lorsque deux threads se synchronisent sur des verrous *differents*, on ne peut rien dire de l'ordre des actions entre eux – il n'y a aucune relation *arrive-avant* entre les actions des deux threads.

16.1.4 Tirer parti de la synchronisation

Grâce à la force de l'ordre *arrive-avant*, on peut parfois profiter des propriétés de la visibilité d'une synchronisation existante. Ceci suppose de combiner la règle de l'ordre des programmes de *arrive-avant* avec une de ses autres règles (généralement celle concernant le verrouillage des moniteurs ou celle des variables volatiles) afin d'ordonner les accès à une variable qui n'est pas protégée par un verrou. Cette approche est très sensible à l'ordre d'apparition des instructions et est donc assez fragile ; il s'agit d'une technique avancée qui devrait être réservée à l'extraction de la dernière goutte de performance des classes comme `ReentrantLock`.

L'implémentation des méthodes protégées de `AbstractQueuedSynchronizer` dans `FutureTask` illustre ce mécanisme. AQS gère un entier de l'état de la tâche : en cours d'exécution, terminée ou annulée. Mais `FutureTask` gère également d'autres variables, comme le résultat de son calcul. Lorsqu'un thread appelle `set()` pour sauvegarder le résultat et qu'un autre thread appelle `get()` pour le récupérer, il vaut mieux qu'ils soient tous les deux ordonnés selon *arrive-avant*. Pour ce faire, on pourrait rendre `volatile` la référence au résultat, mais il est possible d'exploiter la synchronisation existante pour obtenir le même effet à moindre coût.

`FutureTask` a été soigneusement conçue pour garantir qu'un appel réussi à `tryReleaseShared()` *arrive-avant* un appel ultérieur à `tryAcquireShared()` ; en effet, `tryReleaseShared()` écrit toujours dans une variable volatile qui est lue par `tryAcquireShared()`. Le Listing 16.2 présente les méthodes `innerSet()` et `innerGet()`, qui sont appelées lorsque le résultat est sauvegardé ou relu ; comme `innerSet()` écrit le résultat avant d'appeler `releaseShared()` (qui appelle `tryReleaseShared()`) et que `innerGet()` lit le résultat après avoir appelé `acquireShared()` (qui appelle `tryAcquireShared()`), la règle de l'ordre des programmes se combine avec celle des variables volatiles pour garantir que l'écriture du résultat dans `innerSet()` *arrive-avant* sa lecture dans `innerGet()`.

Listing 16.2 : Classe interne de FutureTask illustrant une mise à profit de la synchronisation.

```
// Classe interne de FutureTask
private final class Sync extends AbstractQueuedSynchronizer {
    private static final int RUNNING = 1, RAN = 2, CANCELLED = 4;
    private V result;
    private Exception exception;

    void innerSet(V v) {
        while (true) {
```



Listing 16.2 : Classe interne de FutureTask illustrant une mise à profit de la synchronisation. (suite)

```
int s = getState();
if (ranOrCancelled(s))
    return;
if (compareAndSetState (s, RAN))
    break;
}
result = v;
releaseShared(0);
done();
}

V innerGet() throws InterruptedException , ExecutionException {
acquireSharedInterruptibly(0);
if (getState() == CANCELLED)
    throw new CancellationException ();
if (exception != null)
    throw new ExecutionException (exception);
return result;
}
}
```

On appelle cette technique *piggybacking* car elle utilise un ordre *arrive-avant* existant qui a été créé pour garantir la visibilité de l'objet *X* au lieu de créer cet ordre uniquement pour publier *X*.

Le piggybacking comme celui utilisé par FutureTask est assez fragile et ne devrait pas être utilisé inconsidérément. Cependant, c'est une technique parfaitement justifiable quand, par exemple, une classe s'en remet à un ordre *arrive-avant* entre les méthodes car cela fait partie de sa spécification. Une publication correcte utilisant une BlockingQueue, par exemple, est une forme de piggybacking. Un thread plaçant un objet dans une file et un autre le récupérant ultérieurement constituent une publication correcte car on est sûr qu'il y a une synchronisation interne suffisante dans l'implémentation de BlockingQueue pour garantir que l'ajout *arrive-avant* la suppression.

Les autres ordres *arrive-avant* garantis par la bibliothèque standard incluent les suivants :

- Le placement d'un élément dans une collection thread-safe *arrive-avant* qu'un autre thread ne récupère cet élément à partir de la collection.
- Le décomptage sur un CountDownLatch *arrive-avant* qu'un thread sorte de await() sur ce loquet.
- La libération d'un permis de Semaphore *arrive-avant* d'acquérir un permis de ce même semaphore.
- Les actions entreprises par la tâche représentée par un objet Future *arrivent-avant* qu'un autre thread revienne avec succès d'un appel à Future.get().
- La soumission d'un Runnable ou d'un Callable à un Executor *arrive-avant* que la tâche ne commence son exécution.

- Un thread arrivant sur un objet `CyclicBarrier` ou `Exchanger` *arrive-avant* que les autres threads ne soient libérés de cette barrière ou de ce point d'échange. Si `CyclicBarrier` utilise une action de barrière, l'arrivée à la barrière *arrive-avant* l'action de barrière, qui, elle-même, *arrive-avant* que les threads ne soient libérés de la barrière.

16.2 Publication

Le Chapitre 3 a montré comment un objet pouvait être publié correctement ou incorrectement. La thread safety des techniques de publication correctes décrites ici provient des garanties fournies par le MMJ ; les risques de publication incorrectes sont des conséquences de l'absence d'ordre *arrive-avant* entre la publication d'un objet partagé et son accès à partir d'un autre thread.

16.2.1 Publication incorrecte

L'éventualité d'une réorganisation en l'absence de relation *arrive-avant* explique pourquoi la publication d'un objet sans synchronisation appropriée peut faire qu'un autre thread voit un *objet partiellement construit* (voir la section 3.5). L'initialisation d'un nouvel objet implique d'écrire dans des variables – les champs du nouvel objet. De même, la publication d'une référence implique d'écrire dans une autre variable – la référence au nouvel objet. Si vous ne vous assurez pas que la publication de la référence partagée *arrive-avant* qu'un autre thread la charge, l'écriture de cette référence au nouvel objet peut être réorganisée (du point de vue du thread qui consomme l'objet) et se mélanger aux écritures dans ses champs. En ce cas, un autre thread pourrait voir une valeur à jour pour la référence à l'objet mais *des valeurs obsolètes pour une partie de l'état (voire pour tout l'état) de cet objet* – il verrait donc un objet partiellement construit.

La publication incorrecte peut être le résultat d'une initialisation paresseuse incorrecte, comme celle du Listing 16.3. Au premier abord, le seul problème ici semble être la situation de compétition décrite dans la section 2.2.2. Dans certaines circonstances – lorsque toutes les instances de `Resource` sont identiques, par exemple –, on peut vouloir ne pas en tenir compte ; malheureusement, même si ces défauts sont ignorés, `UnsafeLazyInitialization` n'est quand même pas thread-safe car un autre thread pourrait voir une référence à un objet `Resource` partiellement construit.

Listing 16.3 : Initialisation paresseuse incorrecte. Ne le faites pas.

```
@NotThreadSafe
public class UnsafeLazyInitialization {
    private static Resource resource;

    public static Resource getInstance() {
        if (resource == null)
            resource = new Resource(); // publication incorrecte
        return resource;
    }
}
```



Supposons que le thread *A* invoque en premier `getInstance()`. Il constate que `resource` vaut `null`, crée une nouvelle instance de `Resource`, qu'il désigne par `resource`. Lorsque *B* appelle ensuite `getInstance()`, il peut constater que cette `resource` a déjà une valeur non `null` et utilise simplement la `Resource` déjà construite. À première vue, tout cela peut sembler sans danger mais *il n'y a pas d'ordre arrive-avant entre l'écriture de resource dans A et sa lecture dans B* : on a utilisé une situation de compétition dans *A* pour publier l'objet, et *B* ne peut donc pas être certain de voir l'état correct de la `Resource`.

Le constructeur `Resource` modifie les champs de l'objet `Resource` qui vient d'être alloué pour qu'ils passent de leurs valeurs par défaut (écrites par le constructeur de `Object`) à leurs valeurs initiales. Comme aucun thread n'utilise de synchronisation, *B* pourrait éventuellement voir les actions de *A* dans un ordre différent de celui utilisé par ce dernier. Par conséquent, même si *A* a initialisé l'objet `Resource` avant d'affecter sa référence à `resource`, *B* pourrait voir l'écriture dans `resource` *avant* l'écriture des champs de l'objet `Resource` et donc voir un objet partiellement construit qui peut très bien être dans un état incohérent – et dont l'état peut changer plus tard de façon inattendue.

L'utilisation d'un objet modifiable initialisé dans un autre thread n'est pas sûre, sauf si la publication *arrive-avant* que le thread consommateur ne l'utilise.

16.2.2 Publication correcte

Les idiomes de publication correcte décrits au Chapitre 3 garantissent que l'objet publié est visible aux autres threads car ils assurent que la publication *arrive-avant* qu'un thread consommateur ne charge une référence à l'objet publié. Si le thread *A* place *X* dans une `BlockingQueue` (et qu'aucun thread ne le modifie ensuite) et que le thread *B* le récupère, *B* est assuré de voir *X* comme *A* l'a laissé. Cela est dû au fait que les implémentations de `BlockingQueue` ont une synchronisation interne suffisante pour garantir que le `put()` *arrive-avant* le `take()`. De même, l'utilisation d'une variable partagée protégée par un verrou ou d'une variable volatile partagée garantit que les lectures et les écritures seront ordonnées selon *arrive-avant*.

Cette garantie est, en réalité, une promesse de visibilité et elle est d'ordre plus fort que celle de la publication correcte. Lorsque *X* est correctement publié de *A* vers *B*, cette publication garantit la visibilité de l'état de *X*, mais pas celle des autres variables que *A* a pu modifier. En revanche, si le placement de *X* par *A* dans une file *arrive-avant* que *B* ne récupère *X* à partir de cette file, *B* voit non seulement *X* dans l'état où *A* l'a laissé (en supposant que *X* n'ait pas été ensuite modifié par *A* ou un autre thread) mais également

tout ce qu'a fait A avant ce transfert (en supposant encore qu'il n'y ait pas eu de modifications ultérieures)¹.

Pourquoi insistons-nous si lourdement sur @GuardedBy et la publication correcte alors que le MMJ nous fournit déjà *arrive-avant*, qui est plus puissant ? Parce que penser en terme de transmission de propriété et de publication d'objet convient mieux à la plupart des conceptions de programmes que raisonner en terme de visibilité des différentes écritures en mémoire. L'ordre *arrive-avant* intervient au niveau des accès mémoire individuels – il s'agit d'une sorte de "langage assebleur de la concurrence". La publication correcte, en revanche, agit à un niveau plus proche de celui du programme.

16.2.3 Idiomes de publication correcte

S'il est parfois raisonnable de différer l'initialisation des objets coûteux jusqu'au moment où l'on en a vraiment besoin, on vient aussi de voir qu'une mauvaise utilisation de l'initialisation paresseuse peut poser des problèmes. UnsafeLazyInitialization peut être corrigée en synchronisant la méthode getInstance(), comme dans le Listing 16.4. Le code de cette méthode étant assez court (un test et un branchement prévisible), les performances de cette approche seront suffisantes si getInstance() n'est pas constamment appelée par de nombreux threads.

Listing 16.4 : Initialisation paresseuse thread-safe.

```
@ThreadSafe
public class SafeLazyInitialization {
    private static Resource resource;

    public synchronized static Resource getInstance() {
        if (resource == null)
            resource = new Resource();
        return resource;
    }
}
```

Le traitement des champs statiques par les initialisateurs (ou des champs dont la valeur est fixée dans un bloc d'initialisation statique [JPL 2.2.1 et 2.5.3]) est un peu spécial et offre des garanties de thread safety supplémentaires. Les initialisateurs statiques sont lancés par la JVM au moment de l'initialisation de la classe, après son chargement mais avant qu'elle ne soit utilisée par un thread. La JVM prenant un verrou au cours de l'initialisation [JLS 12.4.2] et ce verrou étant pris au moins une fois par chaque thread pour garantir que la classe a été chargée, les écritures en mémoire au cours de l'initialisation statique sont automatiquement visibles par tous les threads. Par conséquent, les objets initialisés statiquement n'exigent aucune synchronisation explicite au cours de leur construction ni lorsqu'ils sont référencés. Cependant, ceci ne s'applique qu'à l'état *au moment de la construction* – si l'objet est modifiable, la synchronisation reste nécessaire

1. Le MMJ garantit que B voit une valeur au moins aussi à jour que celle que A a écrite ; les écritures ultérieures peuvent être visibles ou non.

pour les lecteurs et les écrivains, afin que les modifications ultérieures soient visibles et pour éviter de perturber les données.

Listing 16.5 : Initialisation impatiente.

```
@ThreadSafe
public class EagerInitialization {
    private static Resource resource = new Resource();

    public static Resource getResource() { return resource; }
}
```

L'utilisation d'une initialisation impatiente, présentée dans le Listing 16.5, élimine le coût de la synchronisation subi par chaque appel à `getInstance()` dans `SafeLazyInitialization`. Cette technique peut être combinée avec le chargement de classe paresseux de la JVM afin de créer une technique d'initialisation paresseuse qui n'exige pas de synchronisation du code commun. L'idiome de la *classe conteneur d'initialisation paresseuse* [EJ Item 48] présenté dans le Listing 16.6 utilise une classe dans le seul but d'initialiser l'objet `Resource`. La JVM diffère de l'initialisation de la classe `ResourceHolder` jusqu'à ce qu'elle soit vraiment utilisée [JLS 12.4.1] et aucune synchronisation supplémentaire n'est nécessaire puisque l'objet est initialisé par un initialisateur statique. Le premier appel à `getResource()` par n'importe quel thread provoque le chargement et l'initialisation de `ResourceHolder`, et l'initialisation de l'objet ressource a lieu au moyen de l'initialisateur statique.

Listing 16.6 : Idiome de la classe conteneur de l'initialisation paresseuse.

```
@ThreadSafe
public class ResourceFactory {
    private static class ResourceHolder {
        public static Resource resource = new Resource();
    }

    public static Resource getResource() {
        return ResourceHolder.resource;
    }
}
```

16.2.4 Verrouillage contrôlé deux fois

Aucun livre sur la concurrence ne serait complet sans une présentation de l'infâme anti-patron du verrouillage contrôlé deux fois (DCL, pour *Double Checked Locking*), présenté dans le Listing 16.7. Dans les toutes premières JVM, la synchronisation, même sans compétition, avait un impact non négligeable sur les performances. De nombreuses astuces rusées (ou, en tout cas, qui semblaient l'être) ont donc été inventées pour réduire ce coût – certaines étaient bonnes, certaines étaient mauvaises et certaines étaient horribles. DCL fait partie des "horribles".

Les performances des premières JVM laissant à désirer, on utilisait souvent l'initialisation paresseuse pour éviter des opérations onéreuses potentiellement inutiles ou pour réduire

le temps de démarrage des applications. Une initialisation correctement écrite exige une synchronisation. Or, à l'époque, cette synchronisation était lente et, surtout, pas totalement maîtrisée : les aspects de l'exclusion étaient bien compris, mais ce n'était pas le cas de ceux liés à la visibilité.

DCL semblait alors offrir le meilleur des deux mondes – une initialisation paresseuse qui ne pénalisait pas le code classique à cause de la synchronisation. Cette astuce fonctionnait en testant d'abord sans synchronisation s'il y avait besoin d'une initialisation et utilisait la référence de la ressource si elle n'était pas `null`. Sinon on synchronisait et on vérifiait à nouveau si la ressource était initialisée afin de s'assurer qu'un seul thread puisse initialiser la ressource partagée. Le code classique – récupération d'une référence à une ressource déjà construite – n'utilisait pas de synchronisation. On se retrouvait donc avec le problème décrit dans la section 16.2.1 : un thread pouvait voir une ressource partiellement construite.

Le véritable problème de DCL est que cette technique suppose que le pire qui puisse arriver lorsqu'on lit une référence d'objet partagé sans synchronisation est de voir par erreur une valeur obsolète (`null`, ici) ; en ce cas, l'idiome DCL compense ce risque en réessayant avec le verrou. Cependant, le pire des cas est, en fait, encore pire – on peut voir une valeur à jour de la référence tout en voyant des valeurs obsolètes pour l'état de l'objet, ce qui signifie que celui-ci peut être vu dans un état incorrect ou incohérent.

Les modifications apportées au MMJ (depuis Java 5.0) permettent à DCL de fonctionner si la ressource est déclarée `volatile` ; cependant, l'impact sur les performances est minime puisque les lectures volatiles ne sont généralement pas beaucoup plus coûteuses que les lectures non volatiles.

Listing 16.7 : Antipatron du verrouillage vérifié deux fois. *Ne le faites pas.*

```
@NotThreadSafe
public class DoubleCheckedLocking {
    private static Resource resource;

    public static Resource getInstance() {
        if (resource == null) {
            synchronized (DoubleCheckedLocking .class) {
                if (resource == null)
                    resource = new Resource();
            }
        }
        return resource;
    }
}
```



Il s'agit donc d'un idiome aujourd'hui totalement dépassé – les raisons qui l'ont motivé (synchronisation lente en cas d'absence de compétition, lenteur du lancement de la JVM) ne sont plus d'actualité, ce qui le rend moins efficace en tant qu'optimisation. L'idiome du conteneur d'initialisation paresseuse offre les mêmes bénéfices et est plus simple à comprendre.

16.3 Initialisation sûre

La garantie d'une initialisation sûre permet aux objets non modifiables correctement construits d'être partagés en toute sécurité sans synchronisation entre les threads, quelle que soit la façon dont ils sont publiés – même si cette publication utilise une situation de compétition (ce qui signifie que `UnsafeLazyInitialization` est sûre si `Resource` n'est pas modifiable).

Sans cette sécurité d'initialisation, des objets censés être non modifiables, comme `String`, peuvent sembler changer de valeur si le thread qui publie et ceux qui consomment n'utilisent pas de synchronisation. L'architecture de sécurité reposant sur l'immutabilité de `String`, une initialisation non sûre pourrait créer des failles permettant à un code malicieux de contourner les tests de sécurité.

Une initialisation sûre garantit que les threads verront les valeurs correctes des champs `final` de tout objet correctement construit tels qu'ils ont été initialisés par le constructeur et quelle que soit la façon dont a été publié l'objet. En outre, toute variable pouvant être atteinte via un champ `final` d'un objet correctement construit (comme les éléments d'un tableau `final` ou le contenu d'un `HashMap` référencé par un champ `final`) est également assurée d'être visible par les autres threads¹.

Avec les objets ayant des champs `final`, l'initialisation sûre empêche de réorganiser la construction avec un chargement initial d'une référence à cet objet. Toutes les écritures dans les champs `final` effectuées par le constructeur, ainsi que toutes les variables accessibles au moyen de ces champs, deviennent "figées" lorsque le constructeur se termine, et tout thread obtenant une référence à cet objet est assuré de voir une valeur qui est au moins aussi à jour que la valeur figée. Les écritures qui initialisent les variables accessibles via des champs `final` ne sont pas réorganisées pour être mélangées avec les opérations qui suivent le gel postconstruction.

Une initialisation sûre signifie qu'un objet de la classe `SafeStates` présentée dans le Listing 16.8 pourra être publié en toute sécurité, même via une initialisation paresseuse non sûre ou en dissimulant une référence à un `SafeStates` dans un champ statique public sans synchronisation, bien que cette classe n'utilise pas de synchronisation et qu'elle repose sur un `HashSet` non thread-safe.

Listing 16.8 : Initialisation sûre pour les objets non modifiables.

```
@ThreadSafe
public class SafeStates {
    private final Map<String, String> states;
```

1. Ceci ne s'applique qu'aux objets qui ne sont accessibles que par des champs `final` de l'objet en construction.

```
public SafeStates() {
    states = new HashMap<String, String>();
    states.put("alaska", "AK");
    states.put("alabama", "AL");
    ...
    states.put("wyoming", "WY");
}

public String getAbbreviation (String s) {
    return states.get(s);
}
}
```

Cependant, il suffirait de quelques modifications pour que `SafeStates` ne soit plus thread-safe. Si `states` n'est pas `final` ou si une autre méthode que le constructeur modifie son contenu, l'initialisation n'est plus suffisamment sûre pour que l'on puisse accéder à `SafeStates` en toute sécurité sans synchronisation. Si cette classe a d'autres champs non `final`, les autres threads peuvent quand même voir des valeurs incorrectes pour ces champs. En outre, autoriser l'objet à s'échapper au cours de la construction invalide la garantie de sécurité offerte par l'initialisation.

L'initialisation sûre ne donne des garanties de visibilité que pour les valeurs accessibles à partir des champs `final` au moment où le constructeur se termine. Pour les valeurs accessibles via des champs non `final` ou celles qui peuvent être modifiées après la construction, la visibilité ne peut être garantie qu'au moyen de la synchronisation.

Résumé

Le modèle mémoire de Java précise les conditions dans lesquelles les actions d'un thread sur la mémoire seront visibles aux autres threads. Il faut notamment s'assurer que les actions se déroulent selon un ordre partiel appelé *arrive-avant*, qui est précisé au niveau des différentes opérations mémoire et de synchronisation. En l'absence d'une synchronisation suffisante, il peut se passer des choses très étranges lorsque les threads accèdent aux données partagées. Cependant, les règles de haut niveau décrites aux Chapitres 2 et 3, comme `@GuardedBy` et la publication correcte, permettent de garantir une sécurité vis-à-vis des threads sans devoir faire appel aux détails de bas niveau de *arrive-avant*.

Annexe

Annotations pour la concurrence

Nous avons utilisé des annotations comme `@GuardedBy` et `@ThreadSafe` pour montrer comment il était possible de préciser les promesses de thread safety et les politiques de synchronisation. Cette annexe documente ces annotations ; leur code source peut être téléchargé à partir du site web consacré à cet ouvrage (d'autres promesses de thread safety et des détails d'implémentation supplémentaires devraient, bien sûr, être également documentés, mais ils ne sont pas capturés par cet ensemble minimal d'annotations).

A.1 Annotations de classes

Nous utilisons trois annotations au niveau des classes pour décrire leurs promesses de thread safety : `@Immutable`, `@ThreadSafe` et `@NotThreadSafe`. `@Immutable` signifie, bien sûr, que la classe n'est pas modifiable et implique `@ThreadSafe`. L'annotation `@NotThreadSafe` est facultative – si une classe n'est pas annotée comme étant thread-safe, on doit supposer qu'elle ne l'est pas ; toutefois, `@NotThreadSafe` permet de l'indiquer explicitement.

Ces annotations sont relativement discrètes et bénéficient à la fois aux utilisateurs et aux développeurs. Les premiers peuvent ainsi savoir immédiatement si une classe est thread-safe et les seconds, s'il faut préserver les garanties de thread safety. En outre, les outils d'analyse statique du code peuvent vérifier que le code est conforme au contrat indiqué par l'annotation en vérifiant, par exemple, qu'une classe annotée par `@Immutable` est bien non modifiable.

A.2 Annotations de champs et de méthodes

Les annotations de classes font partie de la documentation publique d'une classe. D'autres aspects de la stratégie d'une classe vis-à-vis des threads sont entièrement destinés aux développeurs et n'apparaissent donc pas dans sa documentation publique.

Les classes qui utilisent des verrous devraient indiquer quelles sont les variables d'état qui sont protégées et par quels verrous. Une source classique de non-respect de thread safety dû à un oubli est lorsqu'une classe thread-safe qui utilise correctement le verrouillage pour protéger son état est ensuite modifiée pour lui ajouter une nouvelle variable d'état qui n'est pas correctement protégée par un verrou ou de nouvelles méthodes qui n'utilisent pas le verrouillage adéquat pour protéger les variables d'état existantes. En précisant les variables protégées et les verrous qui les protègent, on peut empêcher ces deux types d'oublis.

`@GuardedBy(verrou)` précise qu'on ne devrait accéder à un champ ou à une méthode qu'à condition de détenir le verrou indiqué. Le paramètre *verrou* identifie le verrou qui doit avoir été pris avant d'accéder au champ ou à la méthode annotée. Les valeurs possibles de *verrou* sont :

- `@GuardedBy("this")`, qui indique un verrouillage interne sur l'objet contenant (celui auquel appartient la méthode ou le champ).
- `@GuardedBy("nomChamp")`, qui précise le verrou associé à l'objet référencé par le champ indiqué ; il peut s'agir d'un verrou interne (pour les champs qui ne réfèrent pas un Lock) ou d'un verrou explicite (pour les champs qui font référence à un objet Lock).
- `@GuardedBy("NomClasse.nomChamp")`, identique à `@GuardedBy("nomChamp")`, mais qui désigne un objet verrou contenu dans un champ statique d'une autre classe.
- `@GuardedBy("nomMéthode()")`, qui précise que l'objet verrou est renvoyé par l'appel à la méthode indiquée.
- `@GuardedBy("NomClasse.class")`, qui désigne l'objet classe littéral de la classe indiquée.

L'utilisation de `@GuardedBy` pour identifier chaque variable d'état qui a besoin d'un verrouillage et pour préciser les verrous qui les protègent permet de faciliter la maintenance, la relecture du code et l'utilisation d'outils d'analyses automatiques pour déceler les éventuelles erreurs de thread safety.

Bibliographie

Ken Arnold, James Gosling et David Holmes, *The Java Programming Language*, Fourth Edition, Addison-Wesley, 2005.

David F. Bacon, Ravi B. Konuru, Chet Murthy et Mauricio J. Serrano, "Thin Locks: Featherweight Synchronization for Java", dans *SIGPLAN Conference on Programming Language Design and Implementation*, pages 258-268, 1998 (<http://citeseer.ist.psu.edu/bacon98thin.html>).

Joshua Bloch, *Effective Java Programming Language Guide*, Addison-Wesley, 2001.

Joshua Bloch et Neal Gafter, *Java Puzzlers*, Addison-Wesley, 2005.

Hans Boehm, "Destructors, Finalizers, and Synchronization", dans *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 262-272, ACM Press, 2003 (<http://doi.acm.org/10.1145/604131.604153>).

Hans Boehm, "Finalization, Threads, and the Java Memory Model", JavaOne presentation, 2005 (<http://developers.sun.com/learning/javaoneonline/2005/coreplatform/TS-3281.pdf>).

Joseph Bowbeer, "The Last Word in Swing Threads", 2005 (<http://java.sun.com/products/jfc/tsc/articles/threads/threads3.html>).

Cliff Click, "Performance Myths Exposed", JavaOne presentation, 2003.

Cliff Click, "Performance Myths Revisited", JavaOne presentation, 2005 (<http://developers.sun.com/learning/javaoneonline/2005/coreplatform/TS-3268.pdf>).

Martin Fowler, "Presentation Model", 2005 (<http://www.martinfowler.com/eaaDev/PresentationModel.html>).

Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides, *Design Patterns*, Addison-Wesley, 1995.

Martin Gardner, "The fantastic combinations of John Conway's new solitaire game 'Life'", *Scientific American*, octobre 1970.

James Gosling, Bill Joy, Guy Steele et Gilad Bracha, *The Java Language Specification*, Third Edition, Addison-Wesley, 2005.

Tim Harris et Keir Fraser, "Language Support for Lightweight Transactions", dans *OOPSLA '03: Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 388-402, ACM Press, 2003 (<http://doi.acm.org/10.1145/949305.949340>).

- Tim Harris, Simon Marlow, Simon Peyton-Jones et Maurice Herlihy, "Composable Memory Transactions", dans *PPoPP '05: Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 48-60, ACM Press, 2005 (<http://doi.acm.org/10.1145/1065944.1065952>).
- Maurice Herlihy, "Wait-Free Synchronization", *ACM Transactions on Programming Languages and Systems*, 13(1):124-149, 1991 (<http://doi.acm.org/10.1145/114005.102808>).
- Maurice Herlihy et Nir Shavit, *Multiprocessor Synchronization and Concurrent Data Structures*, Morgan-Kaufman, 2006.
- C. A. R. Hoare, "Monitors: An Operating System Structuring Concept", *Communications of the ACM*, 17(10):549-557, 1974 (<http://doi.acm.org/10.1145/355620.361161>).
- David Hovemeyer et William Pugh, "Finding Bugs is Easy", *SIGPLAN Notices*, 39 (12) : 92-106, 2004 (<http://doi.acm.org/10.1145/1052883.1052895>).
- Ramnivas Laddad, *AspectJ in Action*, Manning, 2003.
- Doug Lea, *Concurrent Programming in Java*, Second Edition, Addison-Wesley, 2000.
- Doug Lea, "JSR-133 Cookbook for Compiler Writers" (<http://gee.cs.oswego.edu/dl/jmm/cookbook.html>).
- J. D. C. Little, "A proof of the Queueing Formula $L = _W^n$ ", *Operations Research*, 9 : 383-387, 1961.
- Jeremy Manson, William Pugh et Sarita V. Adve, "The Java Memory Model", dans *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 378-391, ACM Press, 2005 (<http://doi.acm.org/10.1145/1040305.1040336>).
- George Marsaglia, "XorShift RNGs", *Journal of Statistical Software*, 8(13), 2003 (<http://www.jstatsoft.org/v08/i14>).
- Maged M. Michael et Michael L. Scott, "Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms", dans *Symposium on Principles of Distributed Computing*, pages 267-275, 1996 (<http://citeseer.ist.psu.edu/michael96simple.html>).
- Mark Moir et Nir Shavit, "Concurrent Data Structures", dans *Handbook of Data Structures and Applications*, Chapitre 47, CRC Press, 2004.
- William Pugh et Jeremy Manson, "Java Memory Model and Thread Specification", 2004 (<http://www.cs.umd.edu/~pugh/java/memoryModel/jsr133.pdf>).
- M. Raynal, *Algorithms for Mutual Exclusion*, MIT Press, 1986.
- William N. Scherer, Doug Lea et Michael L. Scott, "Scalable Synchronous Queues", dans *11th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPoPP)*, 2006.
- R. K. Treiber, "Systems Programming: Coping with Parallelism", Technical Report RJ 5118, IBM Almaden Research Center, avril 1986.
- Andrew Wellings, *Concurrent and Real-Time Programming in Java*, John Wiley & Sons, 2004.

Index

A

AbortPolicy, politique de saturation 179
AbstractQueuedSynchronizer, classe 351
acquire(), méthode de
 AbstractQueuedSynchronizer 318
acquireShared(), méthode de
 AbstractQueuedSynchronizer 318
afterExecute(), méthode de
 ThreadPoolExecutor 166, 183
Analyse des échappements 236
Annotations 7
Annulable, activité 140
Annulation
 points d'annulation 143
 politique 141
 tâches 140
Antisèche sur la concurrence 113
ArrayBlockingQueue, classe 94, 269
ArrayDeque, classe 97
ArrayIndexOutOfBoundsException, exception 84
 itérations sur un Vector 85
AsynchronousCloseException, exception 152
AtomicBoolean, classe 332
AtomicInteger, classe 331, 333
AtomicLong, classe 332
AtomicMarkableReference, classe 343
AtomicReference, classe 25, 332, 333
AtomicReferenceFieldUpdater, classe 342
AtomicStampedReference, classe 343
Atomique, opération 20

Attente tournante 237

availableProcessors(), méthode de Runtime
 174
await()
 méthode de Condition 313
 méthode de CountDownLatch 100, 174
 méthode de CyclicBarrier 105

AWT, toolkit graphique 12

B

Barrière mémoire 235
beforeExecute(), méthode de
 ThreadPoolExecutors 183
BlockingQueue
 classe 354
 garanties de publication 55
 interface 92

C

Cache
 éviction 112
 pollution 112
CallerRunsPolicy, politique de saturation 179
cancel(), méthode de Future 150, 153
CancellationException, exception 130
Classes
 internes, publication des objets 43
 thread-safe 18
ClosedByInterruptException, exception 152
Cohérence
 des caches 346
 séquentielle 347

Collections thread-safe, garanties de publication 55

compareAndSet()
méthode de AtomicInteger 331
méthode de AtomicReference 333, 338

compareAndSetState(), méthode de AbstractQueuedSynchronizer 317

ConcurrentHashMap, classe 109, 247

ConcurrentLinkedQueue
classe 232, 339
garanties de publication 55

ConcurrentMap, classe, garanties de publication 55

ConcurrentModificationException, exception, itérations 86

ConcurrentSkipListMap, classe 248

Confinement
au thread 195
des objets
classe ThreadLocal 46
types primitifs 47
variables locales 47
variables volatiles 46

Constructeurs
itérations cachées 88
publication des objets 44

CopyOnWriteArrayList, classe, garanties de publication 55

CopyOnWriteArraySet, classe, garanties de publication 55

countDown(), méthode de CountDownLatch 100

CountDownLatch, classe 100

CyclicBarrier, classe 105, 266

D

Data Race, définition 21

Date, classe, garanties de publication 57

DelayQueue, classe 128

Diagrammes d'entrelacement 7

DiscardOldestPolicy, politique de saturation 179

DiscardPolicy, politique de saturation 179

done(), méthode de FutureTask 202

E

Élision de verrou 236

Encapsulation
et thread safety 59
état d'un objet 16

Épaississement de verrou 236

equals(), méthode des collections, itérations cachées 88

État d'un objet 15, 59
accès concurrents 15, 16
encapsulation 16, 29
espace d'état 60
objets sans état 19
opérations dépendantes de l'état 61
protection par verrous internes 30
publication 73

Éviction d'un cache 112

Exchanger, classe 107

ExecutionException, exception 102, 130

F

FindBugs9, analyse statique du code 278

Frontières entre les tâches 117

Fuite de threads 163

Future
classe 201
interface 101

FutureTask, classe 109, 351

G

Garanties de publication 55
des objets, exigences 57

Générateur de nombres pseudo-aléatoires 333

`get()`, méthode de Future 101, 109, 131, 133, 151, 322

`getState()`, méthode de AbstractQueuedSynchronizer 317

H

`hashCode()`

 méthode de Object 212

 méthode des collections, itérations cachées 88

`HashMap`, classe 247

`Hashtable`, classe, garanties de publication 55

I

`identityHashCode()`, méthode de System 212

`Initialisateurs statiques`, garanties de publication 56

`interrupt()`, méthode de Thread 98

`InterruptedException`, exception 98

`InterruptedException`

 d'un thread 98

 mécanisme coopératif 139

`Invariant`

 de classe 25

 variables impliquées 25

 Violation 25

`Inversion des priorités` 327

`invokeAll()`, méthode de ExecutorService 136

`invokeLater()`, méthode Swing 45

`iostat`, commande Unix 246

`isCancelled()`, méthode de Future 203

`isEmpty()`, méthode de Map 244

`isHeldExclusively()`, méthode de AbstractQueuedSynchronizer 318

`Itérations`

 cachées

 méthode des collections equals() 88

 méthode des collections toString() 87

 exception

`ConcurrentModificationException` 86

J

`java.util.concurrent.atomic`, paquetage 24

`JavaServer Pages (JSP)` 11

`join()`, méthode de Thread 174

`JSP (JavaServer Pages)` 11

L

`LinkedBlockingDeque`, classe 97

`LinkedBlockingQueue`, classe 94, 178, 232, 269

`LinkedList`, classe 232

`Lire-modifier-écrire`, opérations composées 20

`Little, loi` 238

`Lock striping` 89

`lockInterruptibly()`, méthode de Lock 152, 287

M

`Mémoïzation` 107

`Mettre-si-absent`, opération composée 30

`mpstat`, commande Unix 245

`Mutex`, implémenté par un sémaphore binaire 103

`MVC (Modèle-Vue-Contrôleur)` 194

N

`newCachedThreadPool()`

 méthode de ThreadPoolExecutor 176

 méthode fabrique 124

`newCondition()`, méthode de Lock 313

`newFixedThreadPool()`

 méthode de ThreadPoolExecutor 176

 méthode fabrique 124

`newScheduledThreadPool()`, méthode de ThreadPoolExecutor 176

`newScheduledThreadPool()`, méthode fabrique 125, 127

newSingleThreadExecutor(), méthode fabrique 124
newThread(), méthode de ThreadFactory 181
newUpdater(), méthode de AtomicReferenceFieldUpdater 342
Nombres sur 64 bits, safety magique 38
Notification conditionnelle 309
notify(), méthode de Object 304, 308
notifyAll(), méthode de Object 303, 308

O

offer(), méthode de BlockingQueue 92

Opérations

atomiques 20
composées
Lire-modifier-écrire 20
Mettre-si-absent 30

Optimisation du code, précautions 17

Option -server, Hotspot 276

P

perfmon, outil Windows 235, 245

Point d'annulation 143

Politique

d'exécution 117
d'interruption 145
de saturation 179
de synchronisation 60, 79
distribution 76

poll(), méthode de BlockingQueue 92

Pollution d'un cache 112

prestartAllCoreThreads(), méthode de ThreadPoolExecutor 176

Priorité des threads vs. priorité du système d'exploitation 222

PriorityBlockingQueue, classe 94

privilegedThreadFactory(), méthode de Executors 183

Publication des objets

conditions de publication correcte 55
problèmes de visibilité 54
valeurs obsolètes 54

put(), méthode de BlockingQueue 92, 174

putIfAbsent(), méthode de ConcurrentMap 111

Q

Queue, interface 232

R

Race condition 8

ReadWriteLock, interface 293

ReentrantLock, classe 333

ReentrantReadWriteLock, classe 294

RejectedExecutionException, exception 179

release(), méthode de AbstractQueuedSynchronizer 318

releaseShared(), méthode de AbstractQueuedSynchronizer 318

Remote Method Invocation (RMI) 11

RMI (Remote Method Invocation) 11

RuntimeException, exception 163

S

Safety magique, nombres sur 64 bits 38

Semaphore, classe 103, 180

Sérialisation, sources 232

Service de terminaison 133

Servlets, framework 11

setRejectedExecutionHandler(), méthode de ThreadPoolExecutor 179

setState(), méthode de AbstractQueuedSynchronizer 317

shutdownNow(), méthode de ExecutorService 161

signal(), méthode de Condition 313

-
- s**
- signalAll(), méthode de Condition 313
 - Situation de compétition**
 - lire-modifier-écrire 22
 - tester-puis-agir 22
 - size(), méthode de Map** 244
 - SocketException, exception** 152
 - submit(), méthode de ExecutorService** 150
 - Swing**
 - confinement des objets 45
 - toolkit graphique 12
 - SwingUtilities, classe** 196
 - SwingWorker, classe** 204
 - Synchronisation, politiques** 29, 79
 - synchronized**
 - blocs 26
 - visibilité mémoire 39
 - mot-clé 16
 - synchronizedList, classe, garanties de publication** 55
 - synchronizedMap, classe, garanties de publication** 55
 - synchronizedSet, classe, garanties de publication** 55
 - SynchronousQueue, classe** 94
- T**
- take(), méthode de BlockingQueue** 92
 - Terminaison, service** 133
 - terminated(), méthode de ThreadPoolExecutors** 183
 - this, référence, publication des objets** 44
 - ThreadFactory, interface** 181
 - ThreadInfo, classe** 280
 - ThreadLocal, classe** 336
 - confinement des objets 46
 - Thread-safe**
 - classe 17
 - programme 17
 - TimeoutException, exception** 135
- T**
- TimerTask, classe** 11, 127
 - toString(), méthode des collections, itérations cachées** 87
 - Transition d'état** 60
 - TreeMap, classe** 248
 - TreeModel, modèle de données de Swing** 199
 - tryAcquire(), méthode de AbstractQueuedSynchronizer** 318
 - tryAcquireShared(), méthode de AbstractQueuedSynchronizer** 318
 - tryLock(), méthode de Lock** 285
 - tryRelease(), méthode de AbstractQueuedSynchronizer** 318
 - tryReleaseShared(), méthode de AbstractQueuedSynchronizer** 318
 - Types primitifs, confinement des objets** 47
- U**
- unconfigurableExecutorService () , méthode de Executors** 183
- V**
- Variables**
 - locales, confinement des objets 47
 - volatiles *Voir Volatiles, variables*
 - Vector, classe**
 - garanties de publication 55
 - synchronisation 29
 - Verrou**
 - côté client 77
 - externe 77
 - moniteur 66
 - privé 66
 - public 66
 - visibilité mémoire 39
 - Verrouillage**
 - partitionné 89
 - protocoles 29
 - Visibilité mémoire** 35
 - données obsolètes 37
 - verrous 39

Vivacité, panne [9](#)

vmstat, commande Unix [235, 245](#)

Volatiles, variables

confinement des objets [46](#)

critères d'utilisation [41](#)

synchronisation [40](#)

W

wait(), méthode de Object [303](#)

Y

yield(), méthode de Thread [222, 265](#)

Programmation concurrente en Java

La programmation concurrente permet l'exécution de programmes en parallèle. À l'heure où les processeurs multicœurs sont devenus un standard, elle est désormais incontournable, et concerne tous les développeurs Java. Mais l'écriture d'un code qui exploite efficacement la puissance des nouveaux processeurs et supporte les environnements concurrents représente un défi à la fois en termes d'architecture, de programmation et de tests.

Le développement, le test et le débogage d'applications multithreads s'avèrent en effet très ardu斯 car, évidemment, les problèmes de concurrence se manifestent de façon imprévisible. Ils apparaissent généralement au pire moment – en production, sous une lourde charge de travail.

Le but de ce livre est de répondre à ces défis en offrant des techniques, des patrons et des outils pour analyser les programmes et pour encapsuler la complexité des interactions concurrentes. Il fournit à la fois les bases théoriques et les techniques concrètes pour construire des applications concurrentes fiables et adaptées aux systèmes actuels – et futurs.

Programmation

Niveau : Intermédiaire / Avancé
Configuration : Multiplate-forme



Pearson Education France
47 bis, rue des Vinaigriers
75010 Paris
Tél. : 01 72 74 90 00
Fax : 01 42 05 22 17
www.pearson.fr

TABLE DES MATIÈRES

- Thread Safety
- Partage des objets
- Composition d'objets
- Briques de base
- Exécution des tâches
- Annulation et arrêt
- Pools de threads
- Applications graphiques
- Éviter les problèmes de vivacité
- Performances et adaptabilité
- Tests des programmes concurrents
- Verrous explicites
- Construction de synchroniseurs personnalisés
- Variables atomiques et synchronisation non bloquante
- Le modèle mémoire de Java
- Annotations pour la concurrence

À propos de l'auteur :

Brian Goetz, consultant informatique, a vingt ans d'expérience dans l'industrie du logiciel et a écrit plus de 75 articles sur le développement en Java. Il s'est entouré des principaux membres du *Java Community Process JSR 166 Expert Group* pour la rédaction de cet ouvrage.

ISBN : 978-2-7440-4109-9

9 782744 041099