Programação concorrente em Java

Paulo Sérgio Almeida

Grupo de Sistemas Distribuídos
Departamento de Informática
Universidade do Minho

2007/2008

Concorrência em Java

- Java suporta a criação explícita de threads, com objectos passivos, e adopta uma variante simplificada de monitores.
- Java tem algum suporte directo na linguagem para concorrência, incluindo *keywords* como `synchronized`, para exclusão mutua.
- Tal oferece vantagens face a um suporte de biblioteca como a API POSIX Threads; e.g. pares de lock-unlock são balanceados, evitando alguns erros acidentais.
- O modelo oferecido na linguagem é demasiado restritivo; é útil recorrer a bibliotecas de concorrência, como a desenvolvida por Doug Lea, agora integrada no JDK1.5.

Concorrência em Java: criação de threads

- Threads podem ser criadas e manipuladas por operações da classe `java.lang.Thread`.
- Para definir o comportamento de uma thread, podemos herdar de `Thread` e redefinir o método `run`.
- Também podemos criar uma classe que implementa a interface `java.lang.Runnable`.

```
public interface java.lang.Runnable { void run() }
Runnable r = ...
Thread t = new Thread(r);
t.start();
```

- Tal é mais versátil do que extender `Thread`, pois permite ultrapassar a falta de herança múltipla em Java.

Concorrência em Java: ciclo de vida de uma thread

- Uma thread criada só começa a correr quando é executado o método `start()`, que leva à invocação de `run()` do objecto com que a thread é iniciada.
- A thread termina a execução quando `run()` retorna.
- Depois de começar a correr e antes de terminar, uma thread pode estar num dos estados: *runnable*, *running* ou *blocked*.
- Uma thread tendo terminado não pode recomeçar a execução. (Não pode ser invocado novamente `start`.)

Concorrência em Java: alguns métodos de Thread

- O método `isAlive()` é um predicado que devolve `true` se a thread começou a correr mas ainda não terminou.
- `t.join()` bloqueia a thread invocadora até a thread `t` terminar (até `t.isAlive()` devolver `false`).
- O método de classe (`static`) `currentThread()` devolve uma referência para a thread a executar.
- O método de classe `sleep(long msecs)` faz a thread invocadora suspender a execução pelo menos `msecs` milisegundos.

Concorrência em Java: exclusão mútua

- Em Java, existe um *lock* associado a cada objecto.
- Acesso ao objecto em exclusão mútua pode ser efectuado através do uso de `synchronized`, que pode ser utilizado:
  - em métodos:

```
class Contador {
  int i;
  synchronized void inc() {
    i++;
  }
}
```

  - em blocos de código, utilizando o *lock* de um objecto `obj`:

```
synchronized (obj) {
  ...
}
```

Concorrência em Java: *locking* recursivo ou reentrante

- Uma thread que adquiriu um *lock*, através de `synchronized`, pode em seguida invocar métodos ou código `synchronized` relativamente ao mesmo objecto sem ficar bloqueada.
- É diferente do comportamento por omissão em POSIX Threads.
- O *lock* conta quantas vezes foi adquirido, bloqueando outras threads até ser libertado o mesmo número de vezes.
- Isto pode ser designado de *locking* recursivo, pois permite métodos `synchronized` serem recursivos:

```
class Contador {
  int i;
  synchronized void soma(int n) {
    if (n>0) { ++i; soma(n-1); }
  }
}
```

Concorrência em Java: sincronização

- Java usa uma variante simplificada de monitores para a sincronização entre objectos.
- A cada objecto está associado um *lock* e uma única variável de condição (implícita) com a correspondente fila de espera.
- Os métodos relevantes, existentes na classe `Object`, são:

  `public final void wait() throws InterruptedException` a thread invocadora liberta o lock associado ao monitor e fica à espera de ser notificada. Quando notificada readquire o lock antes de recomeçar a execução.

  `public final void notify()` acorda uma thread bloqueada na fila de espera de `wait()` do objecto.

  `public final void notifyAll()` acorda todas as threads bloqueadas na fila de espera de `wait()` do objecto.

Concorrência em Java: uso de inner classes anónimas

- Muitas vezes é necessário criar uma thread que possa manipular objectos acessíveis no contexto da criação.
- Torna-se pouco prático ter que, para cada contexto: criar uma nova classe `Runnable`, declarar variáveis de instância, declarar um construtor com parâmetros apropriados.
- É frequente o uso de `inner classes` anónimas em programação concorrente. Exemplo: criação de uma thread que fica a invocar a operação `vender` de um `Artigo`:

```
final Artigo a = ...
final int quant = ...
Runnable r = new Runnable() {
  public void run() {
    a.vender(quant);
  }
};
(new Thread(r)).start();
```

Exemplo: semáforos

O conceito de semáforos pode ser implementado facilmente em Java:

```
public class Semaforo {
  protected int v;

  public Semaforo(int i) { v = i; }

  public synchronized void up() {
    ++v;
    notify();
  }

  public synchronized void down()
      throws InterruptedException {
    while (v == 0) wait();
    --v;
  }
}
```

Exemplo: bounded buffer

Um *bounded buffer* clássico com as operações `get` e `put`:

```
class Buffer {
  final int N = 10;
  int i = 0;

  public synchronized Object get() throws InterruptedException {
    while (i == 0) wait();
    i--;
    ...
    notifyAll();
    return ...
  }
  public synchronized void put(Object o) throws InterruptedException {
    while (i == N) wait();
    i++;
    ... = o
    notifyAll();
  }
}
```

Exemplo: bounded buffer—produtor

Produtor que invoca num ciclo infinito a operação `put`:

```
class Producer implements Runnable {
  Buffer b;
  Producer(Buffer b1) {
    b = b1;
  }
  int i;
  public void run() {
   try {
    while(true) {
      i++;
      b.put(...);
      Thread.sleep(1000);
    }
   } catch (InterruptedException e) { }
  }
}
```

Exemplo: bounded buffer—consumidor

Consumidor que invoca num ciclo infinito a operação `get`:

```
class Consumer implements Runnable {
  Buffer b;
  Consumer(Buffer b1) {
    b = b1;
  }
  int i;
  public void run() {
    try {
    while(true) {
      i++;
      b.get();
      Thread.sleep(2000);
    }
    } catch (InterruptedException e) { }
  }
}
```

Exemplo: bounded buffer—Main

Classe main que instancia um buffer, um produtor e um consumidor:

```
class Main {
  public static void main(String[] args) {
    Buffer b = new Buffer();
    Producer p = new Producer(b);
    Consumer c = new Consumer(b);
    Thread t1 = new Thread(p);
    Thread t2 = new Thread(c);
    t1.start();
    t2.start();
  }
}
```

Monitores aninhados

- A combinação de monitores pode dar origem a *deadlock* se não forem tomadas precauções.
- Um *bounded buffer* implementado descuidadamente via semáforos:

```
class Buffer {
  protected Semaforo items, slots;
  ...
  public synchronized void put(Object O) {
    slots.down();
    ...
    items.up();
  }
  public synchronized Object get() {
    items.down();
    ...
    slots.up();
  }
}
```

Monitores aninhados

- O código anterior dá origem a *deadlock* pois o *lock* de um `buffer` não é libertado ao ser feito um `wait()` num semáforo, impedindo outras threads de entrarem nesse objecto `buffer`.
- O problema pode ser corrigido com `synchronized` em blocos de código:

```
class Buffer {
  protected Semaforo items, slots;
  ...
  public void put(Object O) {
    slots.down();
    synchronized (this) { ... }
    items.up();
  }
  public Object get() {
    items.down();
    synchronized (this) { ... }
    slots.up();
  }
}
```

Limitações dos monitores nativos de Java

- Nativamente, um objecto só tem uma variável de condição.
- Tal pode levar ao uso de `notifyAll` e a muita ineficiência.
- Exemplo:
  - no bounded-buffer, usamos `notifyAll`;
  - poderiamos usar `notify` para melhorar eficiência?
- Em geral pode ser perigoso ou impossível usar `notify`, tendo apenas uma variável de condição.

Monitores genéricos em Java 5

- Temos agora uma biblioteca genérica de concorrência em java.util.concurrent.

```
interface java.util.concurrent.locks Lock;
class java.util.concurrent.locks ReentrantLock;
interface java.util.concurrent.locks Condition;
```

- Podemos obter locks com a mesma semântica dos nativos com:

```
ReentrantLock lock = new ReentrantLock();
lock.lock();
lock.unlock();
```

- Podemos ter variáveis de condição associadas a um lock com:

```
Condition cond = lock.newCondition();
cond.await();
cond.signal();
cond.signalAll();
```

Bounded-buffer com monitores genéricos

- Vamos distinguir espera por buffer estar cheio ou vazio.
- Declaramos um lock e duas variáveis de condição.

```
import java.util.concurrent.locks.*;

class Buffer {
  private final ReentrantLock lock;
  private final Condition notFull, notEmpty;
  private final int N = 5;
  private int i = 0;

  public Buffer() {
    lock = new ReentrantLock();
    notFull = lock.newCondition();
    notEmpty = lock.newCondition();
  }
```

Bounded-buffer com monitores genéricos

- Os métodos não são `synchronized`.
- O lock é explicitamente adquirido e libertado.
- Usamos `try`/`finally` para libertar lock mesmo que haja excepção.

```
public void get() throws InterruptedException {
  lock.lock();
  try {
    while (i == 0) notEmpty.await();
    i--;
    notFull.signal();
  } finally {
   lock.unlock();
  }
}
```

Bounded-buffer com monitores genéricos

```
  public void put() throws InterruptedException {
    lock.lock();
    try {
      while (i == N) notFull.await();
      i++;
      notEmpty.signal();
    } finally {
      lock.unlock();
    }
  }
}
```