Programação Concorrente e Paralela

Capítulo 7 – Objeto Condição

Marcial Porto Fernández
marcial@larces.uece.br

Semestre 2018.2

Sumário

- Objetos Condição
  - Objeto Condição simples
  - O "Problema do Monitor Aninhado"
- Usando Permits
- Usando Semáforos

Sumário

- Objetos Condição
  - Objeto Condição simples
  - O "Problema do Monitor Aninhado"
- Usando Permits
- Usando Semáforos

Objetos Condição (1)

- Definição: *objeto condição* é uma **Interface** que consiste em encapsular as esperas e as notificações usadas em métodos guardados.
- Aplicabilidade
  - Para simplificar o projeto de classe, aliviar o processo de espera e os mecanismos de notificação.
  - **CUIDADO:** Por causa das características de objetos condição em Java, em alguns casos poderá aumentar ao invés de diminuir a complexidade!
    - Ex: quando um objeto tem várias condições de várias notificações.

Objetos Condição (2)

- Aplicabilidade
  - Ao isolar as condições, muitas vezes você pode evitar espera de notificações de threads que não podem prosseguir aguardando uma mudança de estado em particular.
  - Como uma forma de encapsular políticas específicas de programação em torno de notificações, por exemplo, impor políticas de justiça ou priorização.
  - Nos casos particulares, onde as condições tem a forma de "permissões" ou "bloqueios".
  - Como uma técnica para reduzir o tamanho do código.

Objetos Condição

- Um cliente que aguarda uma condição, bloqueia até que o signal() de outro objeto que atenda a essa condição mude de estado.

```
public interface Condition {
      public void await(); // wait for some condition
      public void await(long timeout, TimeUnit unit);
      public void await(Date timeout);
      // wait for some condition until timeout and return
      public void signal(); // signal that condition
}
```

java.util.concurrent.locks.Condition

Sumário

- Objetos Condição
  - Objeto Condição simples
  - O "Problema do Monitor Aninhado"
- Usando Permits
- Usando Semáforos

Um objeto condição simples

- Podemos encapsular as condições de guarda com essa classe:

```
public class SimpleConditionObject implements Condition
{
      public synchronized void await() {
            try { wait(); }
            catch (InterruptedException ex) {}
      }
      public synchronized void signal() {
            notifyAll ();
      }
}
```

Um objeto condição simples com timeout

- Exemplo de objeto condição com timeout:

```
public class TimeoutConditionObject implements Condition
{
      public synchronized void await(delay, TimeUnit.SECONDS) {
            try { wait(); }
            catch (InterruptedException ex) {}
      }
      public synchronized void signal() {
            notifyAll();
      }
}
```

Sumário

- Objetos Condição
  - Objeto Condição simples
  - O "Problema do Monitor Aninhado"
- Usando Permits
- Usando Semáforos

Problema do Monitor Aninhado (1)

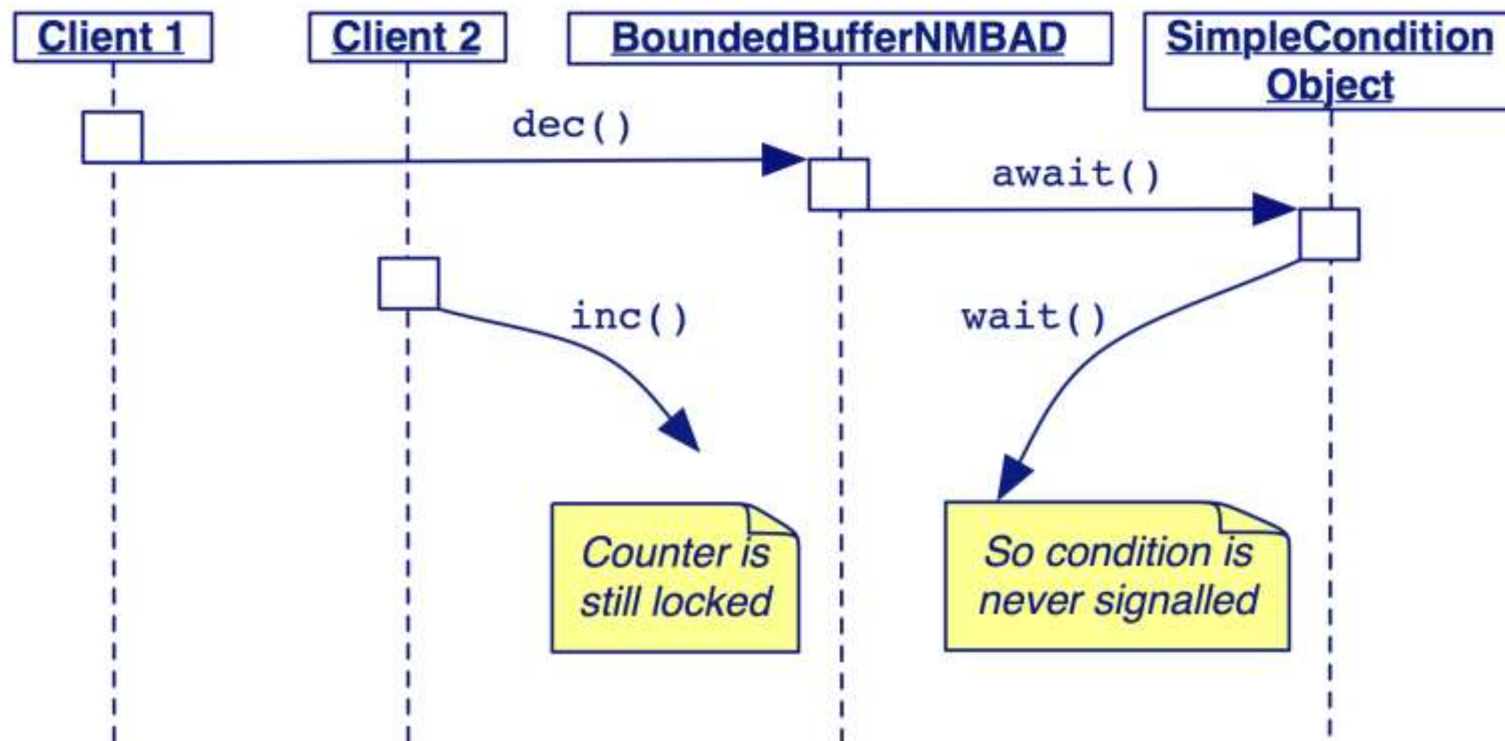
- Nested Monitor Problem
  - Queremos evitar acordar os threads errados, notificando separadamente as condições notMin e notMax.

```
public class BoundedCounterNestedMonitorBAD
 extends BoundedCounterAbstract {
     protected Condition notMin = new SimpleConditionObject();
     protected Condition notMax = new SimpleConditionObject();
     public synchronized long value() { return count; }
...
```

Problema do Monitor Aninhado (2)

```
public synchronized void dec() {
   while (count == MIN)
      notMin.await();          // wait till count not MIN
   if (count-- == MAX)
      notMax.signal();
}
public synchronized void inc() {  // can't get in!
   while (count == MAX)
      notMax.await();
   if (count++ == MIN)
      notMin.signal();               // we never get here!
}
```

Problema do Monitor Aninhado ...



O problema do monitor aninhado ocorre sempre que um thread bloqueado mantiver o bloqueio de um objeto que contém um método que execute uma notificação para desbloquear a espera.

Resolvendo o problema de monitores aninhados (1)

- Você deve garantir que:
  - Waits não podem ocorrer quando a sincronização é mantida sobre o objeto do programa principal.
    - Deve usar um loop de guarda que reverte uma sincronização com falha
  - Notificações nunca serão perdidas.
    - O loop de guarda de espera deve ser colocado dentro de blocos sincronizados dentro do objeto condição.

Resolvendo o problema de monitores aninhados (2)

- Você deve garantir que:
  - Notificações não entram em impasse.
    - Todas as notificações devem ser realizadas apenas após a liberação de toda a sincronização (exceto para a condição deste próprio objeto).
  - Objeto auxiliar e estado do host devem ser consistente.
    - Se o objeto auxiliar mantém algum estado, deve ser sempre coerentes com o estado do host.
    - Se ele compartilhar algum estado com o host, o acesso deve ser sincronizado (exclusão mútua).

Exemplo de solução

```
public class BoundedCounterCondition extends BoundedCounterAbstract {
public void dec() {                     // not synched!
    boolean wasMax = false;             // record notification condition
    synchronized(notMin) {          // synch on condition object
        while (true) {                  // new guard loop
            synchronized(this) {
                if (count > MIN) {  // check and act
                    wasMax = (count == MAX);
                    count--;
                    break;
                }
            }
            notMin.await();         // release host synch before wait
        }
    }
    if (wasMax) notMax.signal(); // release all syncs!
} ...
}
```

Sumário

- Objetos Condição
  - Objeto Condição simples
  - O "Problema do Monitor Aninhado"
- Usando Permits
- Usando Semáforos

Permits

- Objetivo: Pacote de sincronização em um objeto condição quando a sincronização depende do valor de um contador.
- Aplicabilidade:
  - Quando uma determinada await pode prosseguir somente se tiver havido mais signals do que await.
    - Ou seja, quando await diminui e signal incrementa o número disponível em "permits".
  - Você precisa garantir a ausência de signals perdidos.
    - Ao contrário dos objetos *simple condition*, funcionam mesmo se uma thread entra em seu await após outro thread já tenha sinalizado que pode continuar (!)
  - As classes host pode invocar métodos Condition fora do código sincronizado.

Permits - etapas do projeto

- Defina uma classe que implementa a Condition que mantém um contador de permissão e imediatamente libera o await se há possibilidade de continuar o permit.
  - Por exemplo, BoundedCounter

Exemplo

```
public class Permit implements Condition {
	private int count;
	Permit(int init) { count = init; }
	public synchronized void await() {
		while (count == 0) {
			try { wait(); }
			catch(InterruptedException ex) { };
		}
		count --;
	}
	public synchronized void signal() {
		count ++;
		notifyAll();
	}
}
```

Projeto ...

- Tal como acontece com todos os tipos de objetos condição, os clientes devem evitar invocar await (espera) dentro do código sincronizado.

```
class Host {
      Condition aCondition; ...
      public method m1() {
            aCondition.await();                    // not synced
            doM1();                                // synced
            for each Condition c enabled by m1()
                  c.signal();                      // not synced
      }
      protected synchronized doM1() { ... }
}
```

Usando permit

```
public class Building{
      Permit permit;
      Building(int n) {
            permit = new Permit(n);
      }
      void enter(String person) {      // NB: unsynchronized
            permit.await();
            System.out.println(person + " has entered the building");
      }
      void leave(String person) {
            System.out.println(person + " has left the building");
            permit.signal();
      }
}
```

Usando permit

```
bob has entered the building
carol has entered the building
ted has entered the building
bob has left the building
alice has entered the building
ted has left the building
carol has left the building
elvis has entered the building
alice has left the building
elvis has left the building
```

```
public static void main(String[] args) {
      Building building = new Building(3);
      enterAndLeave(building, "bob");
      enterAndLeave(building, "carol");
      ...
}

private static void enterAndLeave(final Building building,
             final String person) {
      new Thread() {
            public void run() {
                  building.enter(person);
                  pause();
                  building.leave(person);
            }
      }.start();
}
```

Sumário

- Objetos Condição
  - Objeto Condição simples
  - O "Problema do Monitor Aninhado"
- Usando Permits
- Usando Semáforos

Semáforos em Java

```
public class Semaphore {                    // simple version
private int value;
public Semaphore (int initial) { value = initial; }
synchronized public void up() {             // AKA V
   ++value;
   notify();                        // wake up just one thread!
}
synchronized public void down() {         // AKA P
   while (value== 0) {
      try { wait(); }
      catch(InterruptedException ex) { };
   }
   --value;
}
}
```

Veja: java.util.concurrent.Semaphore

Usando Semáforos

```
public class BoundedCounterSem extends BoundedCounterAbstract { …
      protected Semaphore mutex, full, empty;
      BoundedCounterVSem() {
            mutex = new Semaphore(1);
            full = new Semaphore(0);           // number of items
            empty = new Semaphore(MAX-MIN);    // number of slots
      }
      public long value() {
            mutex.down();                      // grab the resource
            long val = count;
            mutex.up();                        // release it
            return val;
      }
      public void inc() {
            empty.down();                      // grab a slot
            mutex.down();
            count ++;
            mutex.up();
            full.up();                         // release an item
      }
…
```

Fim do Capítulo 7

- Esse capítulo apresentou o Objeto Condição, Permit e Semaphore.
- Revise os conceitos apresentados.
- Resolver os exercícios da Lista.