Programação Concorrente

Sincronizadores: Locks, Latches e Barreiras

Prof. Rodrigo Campiolo

UTFPR - Universidade Tecnológica Federal do Paraná
DACOM - Departamento de Computação
BCC - Bacharelado Ciência da Computação

30 de outubro de 2018

Introdução

- Apresentar estruturas para sincronização entre threads.
- Estruturas:
  - Locks
  - Latches
  - CyclicBarrier
  - Phaser
  - Exchanger

Locks

- A sincronização de código usando locks reentrantes com **syncronized**, **wait** e **notify** têm limitações.
- O pacote **java.util.concurrent.locks** provê outros mecanismos com mais opções.
- A interface básica é a **Lock**.
- Objetos **Lock** se comportam como *locks* implícitos, logo somente uma thread pode manter o lock.
- Objetos **Lock** também suportam **wait** e **notify** por meio de objetos **Condition**.

interface **Lock**

- Provê três funcionalidades adicionais a blocos sincronizados.
  - **tryLock()**: tentativa de obter o *lock* sem bloquear.
  - **lockInterruptibly()**: tentativa de obter o *lock* que pode ser interrompida.
  - **tryLock(long, TimeUnit)**: tentativa de obter o *lock* até o limite do *timeout*.
- Podem prover acesso concorrente a um recurso (p. ex. read lock de **ReadWriteLock**).
- O acesso e liberação dos *locks* não precisa ser em cadeia.
- Outros comportamentos e semânticas oferecidas são garantia de ordem, uso não-reentrante e detecção de deadlock.

interface **Lock** - Uso padrão

```
Lock l = ...;
l.lock();
try {
   // access the resource protected by this lock
} finally {
   l.unlock();
}
```

interface **Lock** - Métodos

- `void lock()`
  Adquire o *lock*.
- `void lockInterruptibly()`
  Adquire o *lock* exceto se a thread corrente é interrompida.
- `Condition newCondition()`
  Devolve uma nova instância de *Condition* que é mapeada para a instância desse *Lock*.
- `boolean tryLock()`
  Devolve o *lock* somente se está livre no momento da invocação.
- `boolean tryLock(long time, TimeUnit unit)`
  Adquire o *lock* se está livre no período de tempo *time* e a thread corrente não foi interrompida.
- `void unlock()`
  Libera o *lock*.

Classe **ReentrantLock**

- Comportamento similar aos blocos sincronizados, mas com funcionalidades adicionais.
- Construtor aceita um parâmetro de *"fairness"*. O padrão é falso.
- Os principais métodos são os mesmos da interface **Lock**.

Classe **ReentrantLock** - Exemplo

```
class X {
   private final ReentrantLock lock = new ReentrantLock();
   // ...

   public void m() {
     lock.lock();  // block until condition holds
     try {
       // ... method body
     } finally {
       lock.unlock()
     }
   }
}
```

interface **Condition**

- *Condition* possibilita uma thread suspender a execução até ser notificada por outra sobre a mudança do estado da condição.
- *Conditions* também são denominadas de filas de condições ou variáveis de condições.
- Uma instância de *Condition* está sempre mapeada a um Lock.
- Possibilita associar múltiplas condições de espera para um mesmo objeto.

Interface **Condition** - Exemplo

```
class BoundedBuffer {
   final Lock lock = new ReentrantLock();
   final Condition notFull  = lock.newCondition();
   final Condition notEmpty = lock.newCondition();

   final Object[] items = new Object[100];
   int putptr, takeptr, count;

   public void put(Object x) throws InterruptedException {
     lock.lock();
     try {
       while (count == items.length)
         notFull.await();
       items[putptr] = x;
       if (++putptr == items.length) putptr = 0;
       ++count;
       notEmpty.signal();
     } finally {
       lock.unlock();
     }
   }
```

Interface **Condition** - Exemplo

```
  // ...
  public Object take() throws InterruptedException {
    lock.lock();
    try {
      while (count == 0)
        notEmpty.await();
      Object x = items[takeptr];
      if (++takeptr == items.length) takeptr = 0;
      --count;
      notFull.signal();
      return x;
    } finally {
      lock.unlock();
    }
  }
}
```

interface **ReadWriteLock**

- Mantém um par de *locks* associados: 1 somente para leitura e 1 somente para escrita.
- O *lock* de leitura pode ser mantido por várias threads leitoras.
- O *lock* de escrita pode ser mantido por uma única thread escritora.
- Discussão: *lock* de leitura aumenta o desempenho?.
- a classe ReentrantReadWriteLock implementa a interface **ReadWriteLock**

interface **ReadWriteLock** - Exemplo 1

```
class CachedData {
   Object data;
   volatile boolean cacheValid;
   final ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();

   void processCachedData() {
     rwl.readLock().lock();
     if (!cacheValid) {
       // Must release read lock before acquiring write lock
       rwl.readLock().unlock();
       rwl.writeLock().lock();
       try {
         // Recheck state because another thread might have
         // acquired write lock and changed state before we did.
         if (!cacheValid) {
           data = ...
           cacheValid = true;
         }
         // Downgrade by acquiring read lock before releasing write lock
         rwl.readLock().lock();
       } finally {
         rwl.writeLock().unlock(); // Unlock write, still hold read
       }
     }

     try {
       use(data);
     } finally {
       rwl.readLock().unlock();
     }
   }
 }
```

interface **ReadWriteLock** - Exemplo 2

```
class RWDictionary {
   private final Map<String, Data> m = new TreeMap<String, Data>();
   private final ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
   private final Lock r = rwl.readLock();
   private final Lock w = rwl.writeLock();

   public Data get(String key) {
     r.lock();
     try { return m.get(key); }
     finally { r.unlock(); }
   }
   public String[] allKeys() {
     r.lock();
     try { return m.keySet().toArray(); }
     finally { r.unlock(); }
   }
   public Data put(String key, Data value) {
     w.lock();
     try { return m.put(key, value); }
     finally { w.unlock(); }
   }
   public void clear() {
     w.lock();
     try { m.clear(); }
     finally { w.unlock(); }
   }
 }
```

Atividades

1. Faça um programa usando Lock para simular a atualização de um contador que é acessado por múltiplas threads. O contador pode diminuir e aumentar.
2. Crie uma classe SharedFifoQueue e use Conditions para controlar se a fila está vazia ou cheia. Teste usando threads produtoras e consumidoras.
3. Faça uma classe ArrayListThreadSafe usando ReadWriteLock. Teste usando threads que realizam leitura e escrita para essa estrutura.

Latches

- Latch é um sincronizador que pode atrasar o progresso de threads até que atinjam o estado final.
- Latch permite a uma ou mais threads aguardarem até um conjunto de operações serem executados em outras threads.
- Latch opera como um portão: enquanto N threads não chegarem ao portão, as threads não podem prosseguir.
- Uma vez que o portão é aberto, não pode ser mais fechado.
- A classe **CountDownLatch** do pacote **java.util.concurrent** implementa um latch com contador.

Latches - Casos de Uso

- Garantir que a execução não continue até que os recursos necessários estejam disponíveis.
  Exemplo: Latch binário poderia ser usado para indicar que um recurso R foi inicializado.
- Garantir que um serviço não inicie até outros serviços estejam inicializados.
  Exemplo: Latch poderia ser usado para aguardar a inicialização de N serviços por outras threads que são básicos para a thread atual continuar.
- Esperar até que todas as partes envolvidas em uma atividade estejam prontas e, só assim, proceder.
  Exemplo: Esperar um conjunto de jogadores de um jogo multiplayer realizarem a inicialização para dar início ao jogo.

Classe **CountDownLatch**

- Inicializado com um contador. Esse contador não pode ser reiniciado.
- O método **await()** bloqueia até o contador atingir 0.
- O método **countDown()** decrementa o contador.
- **CountDownLatch** inicializado com N pode ser usado para:
  - uma thread esperar até N threads terem finalizado uma ação.
  - alguma ação ser completada N vezes.

Classe **CountDownLatch** - Exemplo 1

```
class Driver { // ...
   void main() throws InterruptedException {
     CountDownLatch startSignal = new CountDownLatch(1);
     CountDownLatch doneSignal = new CountDownLatch(N);

     for (int i = 0; i < N; ++i) // create and start threads
       new Thread(new Worker(startSignal, doneSignal)).start();

     doSomethingElse();            // don't let run yet
     startSignal.countDown();      // let all threads proceed
     doSomethingElse();
     doneSignal.await();           // wait for all to finish
   }
 }
```

```
class Worker implements Runnable {
   private final CountDownLatch startSignal;
   private final CountDownLatch doneSignal;
   Worker(CountDownLatch startSignal, CountDownLatch doneSignal) {
     this.startSignal = startSignal;
     this.doneSignal = doneSignal;
   }
   public void run() {
     try {
       startSignal.await();
       doWork();
       doneSignal.countDown();
     } catch (InterruptedException ex) {} // return;
   }

   void doWork() { ... }
 }
```

Barreiras

- Barreiras bloqueiam um grupo de threads até todas chegarem no ponto da barreira, só assim para procederem.
- Latches esperam por eventos e barreiras por outras threads.
- A classe **CyclicBarrier** do pacote **java.util.concurrent** implementa uma barreira cíclica.
- Uma barreira cíclica possibilita reaproveitar a barreira em vários ciclos da execução.

Classe **CyclicBarrier** - Construtor e Métodos

- `CyclicBarrier(int parties, Runnable barrierAction)`
  Constrói uma barreira e associa uma ação a ser executada.
- `int await()`
  Aguarda na barreira até todas as threads a alcançarem. Devolve o índice de chegada.
- `getNumberWaiting()`
  Devolve o número de threads (parties) que estão aguardando.
- `getParties()`
  Devolve número de threads (parties) para atravessar a barreira.
- `isBroken()`
  Verifica se a barreira está em um estado quebrado (exceção ou timeout).
- `reset()`
  Reinicializa a barreira e gera exceção *BrokenBarrierException* para quem já estiver esperando.

Classe **CyclicBarrier** - Exemplo: Decomposição paralela

```
class Solver {
  final int N;
  final float[][] data;
  final CyclicBarrier barrier;

  class Worker implements Runnable {
    int myRow;
    Worker(int row) { myRow = row; }
    public void run() {
      while (!done()) {
        processRow(myRow);

        try {
          barrier.await();
        } catch (InterruptedException ex) {
          return;
        } catch (BrokenBarrierException ex) {
          return;
        }
      }
    }
  }
```

```
  public Solver(float[][] matrix) {
    data = matrix;
    N = matrix.length;
    Runnable barrierAction =
      new Runnable() { public void run() { mergeRows(...); }};
    barrier = new CyclicBarrier(N, barrierAction);

    List<Thread> threads = new ArrayList<Thread>(N);
    for (int i = 0; i < N; i++) {
      Thread thread = new Thread(new Worker(i));
      threads.add(thread);
      thread.start();
    }

    // wait until done
    for (Thread thread : threads)
      thread.join();
  }
}
```

Classe **Phaser**

- Barreira de sincronização reusável.
- Similar a barreiras cíclicas e latches, mas mais flexível.
- Possibilita configurar dinamicamente o número de threads que aguardam na barreira.
- Mais informações em:
  `https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Phaser.html`.

Classe Phaser - Métodos

- `Phaser(int parties)`
  Construtor com um número inicial de partes que não chegaram.
- `public int register()`
  Adiciona uma nova parte (não chegou) no phaser.
- `public int arriveAndAwaitAdvance()`
  Chega no phaser e espera as outras partes.
- `public final int getPhase()`
  Devolve o número da fase atual.
- `public int arriveAndDeregister()`
  Chega no phaser e cancela o registro sem esperar outras partes.

Phaser - Exemplo 1 - Iniciar threads juntas

```
void runTasks(List<Runnable> tasks) {
   final Phaser phaser = new Phaser(1); // "1" to register self

   // create and start threads
   for (final Runnable task : tasks) {
     phaser.register();
     new Thread() {
       public void run() {
         phaser.arriveAndAwaitAdvance(); // await all creation
         task.run();
       }
     }.start();
   }

   // allow threads to start and deregister self
   phaser.arriveAndDeregister();
 }
```

Classe **Exchanger**

- Provê um ponto de sincronização para duas threads parearem e trocarem objetos.
- Pode ser usado em cenários com Produtores e Consumidores para a troca de estruturas preenchidas e vazias.
- A sincronização ocorre como método **exchange()** invocado em ambas threads.

Classe **Exchanger** - Exemplo

```
import java.util.concurrent.Exchanger;

public class ExchangerDemo {

    public static void main(String[] args) {
        Exchanger<String> ex = new Exchanger<String>();
        // Starting two threads
        new Thread(new ProducerThread(ex)).start();
        new Thread(new ConsumerThread(ex)).start();
    }
}

class ProducerThread implements Runnable {
    String str;
    Exchanger<String> ex;
    ProducerThread(Exchanger<String> ex){
        this.ex = ex;
        str = new String();
    }
    @Override
    public void run() {
        for(int i = 0; i < 3; i ++){
            str = "Producer" + i;
            try {
                // exchanging with an empty String
                str = ex.exchange(str);
            } catch (InterruptedException e) {
                System.out.println(e);
            }
        }
    }
}
```

```
class ConsumerThread implements Runnable {
    String str;
    Exchanger<String> ex;
    ConsumerThread(Exchanger<String> ex){
        this.ex = ex;
    }
    @Override
    public void run() {
        for(int i = 0; i < 3; i ++){
            try {
                // Getting string from producer thread
                // giving empty string in return
                str = ex.exchange(new String());
                System.out.println("Got from Producer " + str);
            } catch (InterruptedException e) {
                System.out.println(e);
            }
        }
    }
}
```

Referências

- GOETZ, Brian. **Java concurrency in practice**. Upper Saddle River, NJ.: Addison-Wesley, 2006.
- Jenkov. **Thread Safety and Shared Resources**. Disponível em `http://tutorials.jenkov.com/java-concurrency/thread-safety.html`
- Oracle. **The Java Tutorials - Concurrency**. Disponível em: `https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html`