Basic Thread Synchronization

Programación Concurrente

2017

Ing. Ventre, Luis O.

Running concurrent phased tasks

- PHASER es una de las primitivas mas potentes y complejas es la que nos brinda java, para ejecutar tareas concurrentes por fases.

- Este mecanismo es muy util cuando tenemos tareas concurrentes divididas en etapas.

- Esta primitiva provee el mecanismo de sincronizacion al final de cada etapa, para que ningun hilo comienze la siguiente etapa hasta que no lleguen todos.

Running concurrent phased tasks

- Como otras primitivas, PHASER debe ser inicializado con la cantidad de participantes a esperar para avanzar de etapa.

- Con la gran ventaja de poder modificar este valor DINAMICAMENTE DURANTE LA EJECUCION.

- Veremos un ejemplo de sincronizacion de 3 tareas, las 3 buscan archivos con extension .log modificados en las ultimas 24 hs.

Running concurrent phased tasks

- La tarea es dividia en 3 etapas:
  - Obtener una lista con los archivos extension .log en el directorio y subdirectorios asignados.
  - Filtrar los archivos, dejando solo los modificados las ultimas 24 horas.
  - Imprimir los resultados en la consola.
  - Las primeras dos etapas, si el hilo actuante no puede cumplirlas, avisa y se desregistra del phaser.

Running concurrent phased tasks

- Metodos de la clase PHASER:
  - arriveAndAwaitAdvance(): metodo que decrementa el contador interno y avisa que el hilo ha llegado al punto de sincronizacion y se duerme a la espera del resto.
  - arriveAndDeregister(): esto notifica al Phaser, que el hilo ha cumplido la etapa en cuestion, pero que no quiere participar en las proximas sincronizaciones. El Phaser no debera esperar mas por ese hilo.
  - isTerminated(): este metodo retorna True, solo si todos los hilos se han desregistrado del Phaser.
  - arrive(): avisa que llego, pero no espera a nadie, ojo con la implementacion.

Running concurrent phased tasks

- Métodos de la clase PHASER:
  - register(): este método, agrega un participante al phaser. Es considerado como no arrivado para la fase actual().
  - bulkRegister(int Parties): este método, agrega el número especificado de participantes al phaser.

Running concurrent phased tasks

- Clase Main

Main.java | FileSearch.java

```
1  package com.packtpub.java7.concurrency.chapter3.recipe5.core;
2  import java.util.concurrent.Phaser;
3  import com.packtpub.java7.concurrency.chapter3.recipe5.task.FileSearch;
5  * Main class of the example
8  public class Main {
9      public static void main(String[] args) {
10
11         // Creates a Phaser with three participants
12         Phaser phaser=new Phaser(3);
13         // Creates 3 FileSearch objects. Each of them search in different directory
14         FileSearch system=new FileSearch("C:\\Windows", "log", phaser);
15         FileSearch apps=new FileSearch("C:\\Program Files","log",phaser);
16         FileSearch documents=new FileSearch("C:\\Documents And Settings","log",phaser);
17
18         // Creates a thread to run the system FileSearch and starts it
19         Thread systemThread=new Thread(system,"System");
20         systemThread.start();
21
22         // Creates a thread to run the apps FileSearch and starts it
23         Thread appsThread=new Thread(apps,"Apps");
24         appsThread.start();
25
26         // Creates a thread to run the documents  FileSearch and starts it
27         Thread documentsThread=new Thread(documents,"Documents");
28         documentsThread.start();
29         try {
30             systemThread.join();
31             appsThread.join();
32             documentsThread.join();
33         } catch (InterruptedException e) {
34             e.printStackTrace();
35         }
36         System.out.printf("Terminated: %s\n",phaser.isTerminated());
```

Running concurrent phased tasks

- Clase

FileSearch

Main.java | FileSearch.java

```
1  package com.packtpub.java7.concurrency.chapter3.recipe5.task;
2  import java.io.File;
8
10 * This class search for files with an extension in a directory
12 public class FileSearch implements Runnable {
13
15     * Initial path for the search
17     private String initPath;
19     * Extension of the file we are searching for
21     private String end;
23     * List that stores the full path of the files that have the extension we are searching for
25     private List<String> results;
27     * Phaser to control the execution of the FileSearch objects. Their execution will be divided
33     private Phaser phaser;
34
36     * Constructor of the class. Initializes its attributes
41     public FileSearch(String initPath, String end, Phaser phaser) {
42         this.initPath = initPath;
43         this.end = end;
44         this.phaser=phaser;
45         results=new ArrayList<>();
46     }
```

Running concurrent phased tasks

- Clase FileSearch

```
51    @Override
52    public void run() {
53
54        // Waits for the creation of all the FileSearch objects
55        phaser.arriveAndAwaitAdvance();
56
57        System.out.printf("%s: Starting.\n",Thread.currentThread().getName());
58
59        // 1st Phase: Look for the files
60        File file = new File(initPath);//en file obtengo un path abstracto
61        if (file.isDirectory()) {
62            directoryProcess(file);
63        }
64
65        // If no results, deregister in the phaser and ends
66        if (!checkResults()){
67            return;
68        }
69
70        // 2nd Phase: Filter the results
71        filterResults();
72
73        // If no results after the filter, deregister in the phaser and ends
74        if (!checkResults()){
75            return;
76        }
77
78        // 3rd Phase: Show info
79        showInfo();
80        phaser.arriveAndDeregister();
81        System.out.printf("%s: Work completed.\n",Thread.currentThread().getName());
82
83    }
```

Running concurrent phased tasks

- Clase FileSearch

```
86⊕     * This method prints the final results of the search..
88⊖    private void showInfo() {
89         for (int i=0; i<results.size(); i++){
90             File file=new File(results.get(i));
91             System.out.printf("%s: %s\n",Thread.currentThread().getName(),file.getAbsolutePath());
92         }
93         // Waits for the end of all the FileSearch threads that are registered in the phaser
94         phaser.arriveAndAwaitAdvance();
95     }
96
98⊕     * This method checks if there are results after the execution of a phase. If there aren't..
102⊖    private boolean checkResults() {
103        if (results.isEmpty()) {
104            System.out.printf("%s: Phase %d: 0 results.\n",Thread.currentThread().getName(),phaser.getPhase());
105            System.out.printf("%s: Phase %d: End.\n",Thread.currentThread().getName(),phaser.getPhase());
106            // No results. Phase is completed but no more work to do. Deregister for the phaser
107            phaser.arriveAndDeregister();
108            return false;
109        } else {
110            // There are results. Phase is completed. Wait to continue with the next phase
111            System.out.printf("%s: Phase %d: %d results.\n",Thread.currentThread().getName(),
112                    phaser.getPhase(),results.size());
113            phaser.arriveAndAwaitAdvance();
114            return true;
115        }
116    }
```

Running concurrent phased tasks

- Clase FileSearch

Main.java | FileSearch.java

```
119     * Method that filter the results to delete the files modified more than a day before now
121    private void filterResults() {
122        List<String> newResults=new ArrayList<>();
123        long actualDate=new Date().getTime();
124        for (int i=0; i<results.size(); i++){
125            File file=new File(results.get(i));
126            long fileDate=file.lastModified();
127
128            if (actualDate-fileDate<TimeUnit.MILLISECONDS.convert(1,TimeUnit.DAYS)){
129                newResults.add(results.get(i));
130            }
131        }
132        results=newResults;
133    }
135     * Method that process a directory
140    private void directoryProcess(File file) {
141
142        // Get the content of the directory
143        File list[] = file.listFiles();
144        if (list != null) {
145            for (int i = 0; i < list.length; i++) {
146                if (list[i].isDirectory()) {
147                    // If is a directory, process it
148                    directoryProcess(list[i]);
149                } else {
150                    // If is a file, process it
151                    fileProcess(list[i]);
152                }
153            }
154        }
155    }
```

Running concurrent phased tasks

- Clase FileSearch

```
163     private void fileProcess(File file) {
164         if (file.getName().endsWith(end)) {
165             results.add(file.getAbsolutePath());
166         }
167     }
168
169 }
```

- Salida por pantalla:

Console

<terminated> Main (22) [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe

```
System: Starting.
Apps: Starting.
Documents: Starting.
Documents: Phase 1: 0 results.
Documents: Phase 1: End.
Apps: Phase 1: 10 results.
System: Phase 1: 29 results.
Apps: Phase 2: 0 results.
Apps: Phase 2: End.
System: Phase 2: 5 results.
System: C:\Windows\setupact.log
System: C:\Windows\SoftwareDistribution\DataStore\Logs\edb.log
System: C:\Windows\System32\catroot2\edb.log
System: C:\Windows\System32\wbem\Logs\wmiprov.log
System: C:\Windows\WindowsUpdate.log
System: Work completed.
Terminated: true
```

Changing data between concurrent tasks

- Java brinda un mecanismo que permite el intercambio de datos entre dos tareas concurrentes. Primitiva Exchanger.

- Esta primitiva permite la sincronizacion entre dos hilos en un determinado punto.

- Cuando los dos hilos llegan a esta punto, la estructura de datos de un hilo va al otro y viceversa.

Changing data between concurrent tasks

- Esta primitiva sincroniza solo dos hilos, por lo tanto puede ser utilizada en el caso productor consumidor con solo una instancia de cada uno.

- Veremos un ejemplo producer consumer.
  - El consumidor comienza con un buffer vacío, y llama al EXCHANGER para sincronizarse. Necesita datos para consumir.
  - El productor comienza con un buffer vacío, genera 10 strings, almacena en un buffer, y llama la primitiva para intercambiar.

Changing data between concurrent tasks

- En este punto ambos hilos estan en el EXCHANGER, e intercambian estructuras
- El primer hilo que ejecuta el EXCHANGER por cuestiones logicas se duerme para el intercambio en espera del arribo del otro.
- La clase EXCHANGER tiene otra version que permite indicar el tiempo maximo que estara el hilo esperando para la sincronización.

Changing data between concurrent tasks

- El proyecto:

Main.java | Consumer.java | Producer.java

```
1  package com.packtpub.java7.concurrency.chapter3.recipe7.core;
2
3  import java.util.ArrayList;
4  import java.util.List;
5  import java.util.concurrent.Exchanger;
6
7  import com.packtpub.java7.concurrency.chapter3.recipe7.task.Consumer;
8  import com.packtpub.java7.concurrency.chapter3.recipe7.task.Producer;
9
11 * Main class of the example
14 public class Main {
15
17      * Main method of the example
20     public static void main(String[] args) {
21         // Creates two buffers
22         List<String> buffer1=new ArrayList<>();
23         List<String> buffer2=new ArrayList<>();
24         // Creates the exchanger
25         Exchanger<List<String>> exchanger=new Exchanger<>();
26         // Creates the producer
27         Producer producer=new Producer(buffer1, exchanger);
28         // Creates the consumer
29         Consumer consumer=new Consumer(buffer2, exchanger);
30         // Creates and starts the threads
31         Thread threadProducer=new Thread(producer);
32         Thread threadConsumer=new Thread(consumer);
33
34         threadProducer.start();
35         threadConsumer.start();
36
37     }
```

Changing data between concurrent tasks

- El proyecto:

Main.java | Consumer.java | Producer.java

```
1  package com.packtpub.java7.concurrency.chapter3.recipe7.task;
2  import java.util.List;
5   * This class implements the consumer of the example
8  public class Consumer implements Runnable {
10     * Buffer to save the events produced
12    private List<String> buffer;
14     * Exchager to synchronize with the consumer
16    private final Exchanger<List<String>> exchanger;
18     * Constructor of the class. Initializes its attributes
22    public Consumer(List<String> buffer, Exchanger<List<String>> exchanger){
23        this.buffer=buffer;
24        this.exchanger=exchanger;
25    }
27     * Main method of the producer. It consumes all the events produced by the Producer.
31    @Override
32    public void run() {
33        int cycle=1;
34
35        for (int i=0; i<10; i++){
36            System.out.printf("Consumer: Cycle %d\n",cycle);
37
38            try {
39                // Wait for the produced data and send the empty buffer to the producer
40                buffer=exchanger.exchange(buffer);
41            } catch (InterruptedException e) {
42                e.printStackTrace();
43            }
44
45            System.out.printf("Consumer: %d\n",buffer.size());
46
47            for (int j=0; j<10; j++){
48                String message=buffer.get(0);
49                System.out.printf("Consumer: %s\n",message);
50                buffer.remove(0);
51            }
52
53            cycle++;
54        }
```

Changing data between concurrent tasks

- El proyecto:

Main.java | Consumer.java | Producer.java

```
1  package com.packtpub.java7.concurrency.chapter3.recipe7.task;
2  import java.util.List;
5   * This class implements the producer
8  public class Producer implements Runnable {
10     * Buffer to save the events produced
12    private List<String> buffer;
14     * Exchager to synchronize with the consumer
16    private final Exchanger<List<String>> exchanger;
18     * Constructor of the class. Initializes its attributes
22    public Producer (List<String> buffer, Exchanger<List<String>> exchanger){
23        this.buffer=buffer;
24        this.exchanger=exchanger;
25    }
27     * Main method of the producer. It produces 100 events. 10 cicles of 10 events.
32    @Override
33    public void run() {
34        int cycle=1;
35
36        for (int i=0; i<10; i++){
37            System.out.printf("Producer: Cycle %d\n",cycle);
38
39            for (int j=0; j<10; j++){
40                String message="Event "+((i*10)+j);
41                System.out.printf("Producer: %s\n",message);
42                buffer.add(message);
43            }
44            try {
45                /*
46                 * Change the data buffer with the consumer
47                 */
48                buffer=exchanger.exchange(buffer);
49            } catch (InterruptedException e) {
50                e.printStackTrace();
51            }
52
53            System.out.printf("Producer: %d\n",buffer.size());
54
55            cycle++;
56        }
```

Changing data between concurrent tasks

- La salida:

Console

<terminated> Main (23) [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe

```
Consumer: Cycle 1
Producer: Cycle 1
Producer: Event 0
Producer: Event 1
Producer: Event 2
Producer: Event 3
Producer: Event 4
Producer: Event 5
Producer: Event 6
Producer: Event 7
Producer: Event 8
Producer: Event 9
Producer: 0
Producer: Cycle 2
Consumer: 10
Producer: Event 10
Consumer: Event 0
Producer: Event 11
Consumer: Event 1
Consumer: Event 2
Consumer: Event 3
Consumer: Event 4
Consumer: Event 5
Producer: Event 12
Consumer: Event 6
Producer: Event 13
Consumer: Event 7
```

Creating a thread executor

- Generalmente en una aplicacion concurrente uno debe implementar:
  - Debe implementar todo el código relacionado con la creación y el manejo de Threads.
  - Se crea un Thread por cada tarea, en caso de aplicaciones con muchas tareas esto puede afectar el Throughput.
- Java brinda un mecanismo para resolver esto:
  - Executor Framework
  - Clase ThreadPoolExecutor

Creating a thread executor

- Con este mecanismo, solo debemos preocuparnos de la creación de los objetos runnables. Y enviarlos al ThreadPoolExecutor.
  - El executor es el responsable de la ejecucion, instanciacion, y uso de los threads necesarios. Pero ademas de esto, tiene como objetivo mejorar la performance con un pool de Threads.
  - Cuando uno envia una tarea un Executor, intenta ejecutarla con un Thread del pool ocioso

Creating a thread executor

- Otra ventaja del ThreadPoolExecutor es la interfaz Callable.
  - El metodo principal de esta interfaz es el metodo call() puede retornar un resultado.
  - Cuando uno envia una tarea callable() a un Executor, obtenemos un objeto que implementa la interfaz FUTURE, el cual nos permite controlar el estado y los resultados del objeto enviado.

Creating a thread executor

- El primer paso para trabajar con un Executor es crear un objeto de la clase ThreadPoolExecutor.
- Hay varios metodos para la creacion del Executor.
- Una vez creado el Executor podemos enviarle tareas runnables y callables.

- La clave de este ejemplo es la clase Server. Esta clase crea y usa el ThreadPoolExecutor para ejecutar tareas.

Creating a thread executor

- La clase ThreadPoolExecutor tiene varios constructores y tipos.
- El ejemplo crea un newCachedThreadPool
- Este ThreadPool crea hilos a medida que necesita. La ventaja de reutilizar es el ahorro el tiempo de creación de los hilos.
- Una vez creado se pueden enviar tareas para ejecucion con el método excecute().

Creating a thread executor

- Se utilizan varios metodos:
  - getPoolSize(): devuelve el numero actual de hilos en el pool.
  - getActiveCount(): devuelve el numero de hilos que estan efectivamente en ejecucion.
  - getCompleteTaskAccount(): devuelve el numero de tareas finalizadas por el executor.
- Es importante FINALIZAR los executors. Si no tiene tareas, seguira esperando el envio de nuevas.

Creating a thread executor

- Si queremos finalizar el executor podemos utilizar los metodos:
  - shutdown(): cuando el executor recibe esta primitiva, no acepta nueva tareas y una vez que finaliza las que esta ejecutando se cierra.
  - shutdownNow(): este metodo finaliza el executor de inmediato. No ejecuta las tareas pendientes. Devuelve una lista de tareas pendientes. Pero las tareas que estaban en ejecucion se finalizan.

Creating a thread executor

- El proyecto:

Main.java | Server.java | Task.java

```
1  package com.packtpub.java7.concurrency.chapter4.recipe1.core;
2
3  import com.packtpub.java7.concurrency.chapter4.recipe1.task.Server;
5
7   * Main class of the example. Creates a server and 100 request of the Task class
11 public class Main {
12
14      * Main method of the example
17     public static void main(String[] args) {
18         // Create the server
19         Server server=new Server();
20
21         // Send 100 request to the server and finish
22         for (int i=0; i<100; i++){
23             Task task=new Task("Task "+i);
24             server.executeTask(task);
25         }
26
27         server.endServer();
28
29     }
30
31 }
```

Creating a thread executor

- Clase server

Main.java | Server.java | Task.java

```
1  package com.packtpub.java7.concurrency.chapter4.recipe1.task;
2  import java.util.concurrent.Executors;
4
6   * This class simulates a server, for example, a web server, that receives
10 public class Server {
11
13     * ThreadPoolExecutors to manage the execution of the request
15    private ThreadPoolExecutor executor;
16
18     * Constructor of the class. Creates the executor object
20    public Server(){
21        executor=(ThreadPoolExecutor)Executors.newCachedThreadPool();
22    }
23
25     * This method is called when a request to the server is made. The
29    public void executeTask(Task task){
30        System.out.printf("Server: A new task has arrived\n");
31        executor.execute(task);
32        System.out.printf("Server: Pool Size: %d\n",executor.getPoolSize());
33        System.out.printf("Server: Active Count: %d\n",executor.getActiveCount());
34        System.out.printf("Server: Completed Tasks: %d\n",executor.getCompletedTaskCount());
35    }
36
37    /**
38     * This method shuts down the executor
39     */
40    public void endServer() {
41        executor.shutdown();
42    }
43
44 }
```

Creating a thread executor

- Clase task

Main.java | Server.java | Task.java

```
1  package com.packtpub.java7.concurrency.chapter4.recipe1.task;
2  import java.util.Date;
4
6   * This class implements a concurrent task
9  public class Task implements Runnable {
10
12      * The start date of the task
14     private Date initDate;
16     * The name of the task
18     private String name;
19
21     * Constructor of the class. Initializes the name of the task
24     public Task(String name){
25         initDate=new Date();
26         this.name=name;
27     }
29     * This method implements the execution of the task. Waits a random period of time and finish
31     @Override
32     public void run() {
33         System.out.printf("%s: Task %s: Created on: %s\n",
34                 Thread.currentThread().getName(),name,initDate);
35         System.out.printf("%s: Task %s: Started on: %s\n",
36                 Thread.currentThread().getName(),name,new Date());
37
38         try {
39             Long duration=(long)(Math.random()*10);
40             System.out.printf("%s: Task %s: Doing a task during %d seconds\n",
41                     Thread.currentThread().getName(),name,duration);
42             TimeUnit.SECONDS.sleep(duration);
43         } catch (InterruptedException e) {
44             e.printStackTrace();
45         }
46
47         System.out.printf("%s: Task %s: Finished on: %s\n",
48                 Thread.currentThread().getName(),name,new Date());
49     }
50
51 }
```

Creating a thread executor

- Salida por consola