Paralelismo & Concurrencia

Guia practica

Mail: adrianmarino@gmail.com
Github: https://github.com/adrianmarino

Paralelismo

- Es la ejecución simultánea de dos o más tareas.
- Las tareas se asignan a cada núcleo.
- Si ejecutamos una proceso compuesto por varias tareas asignadas a distintas máquinas, se está distribuyendo su procesamiento.

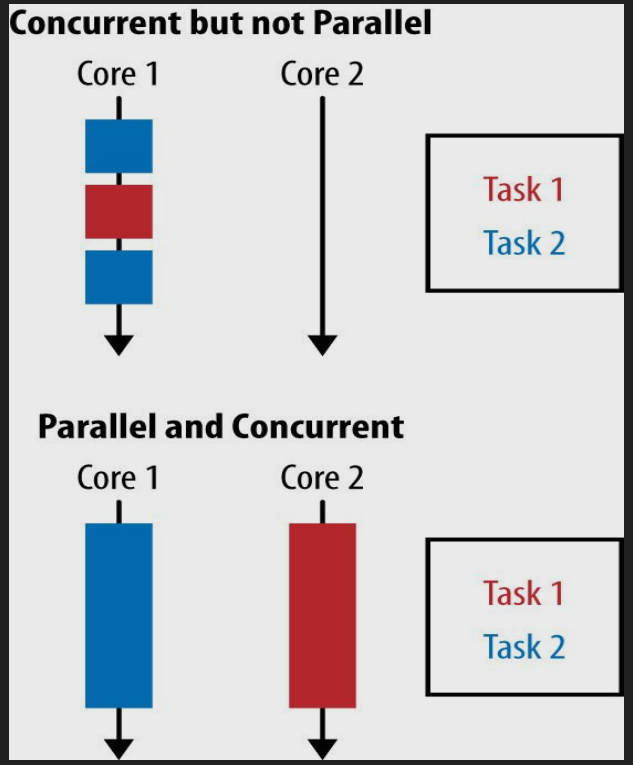
Concurrencia

- Es la ejecución simultánea de dos o más tareas.
- Las tareas se ejecutan en forma alternada en el tiempo.
- A cada tareas se le asigna un periodo de tiempo para utilizar el procesador de forma alternada.

Son iguales?... No!

- Concurrencia es **ejecutar varias tareas** utilizando el procesador **en forma alternada** (Una rodaja de tiempo cada tareas).

- Paralelizar es **ejecutar varias tareas en forma simultánea** (un núcleo por cada tarea).



Paralelismo vs. Concurrencia

- La concurrencia suele enfatizar la sincronización para manejar la lectura/escritura de recursos dentro de un programa.
- El paralelismo suele enfatizar el partir una tarea en varias subtareas aisladas y luego esperar la finalización de todas estas, para finalmente construir un resultado.

...Esto no está escrito en piedra...

Paralelismo vs. Concurrencia

Cuando se paraleliza un operación sobre una colección hay que tener en cuenta el tamaño de la misma y el costo de la operación.

...si la colección es demasiado pequeña o el costo computacional es bajo la paralelización perjudicará el rendimiento (Overhead)...

Sincronización

- Es necesaria cuando dos o más threads comparten un mismo recurso.
- En este caso, se dice que se **sincroniza** el acceso al recurso.
- Se llama **zona crítica** a la sección de código que accede al recurso.
- Una zona crítica solo puede ser accedida por un thread a la vez.
- De esta manera evitamos **race condition**.

Sincronización

3 estrategias

- Locking
- Confinamiento
- Inmutabilidad

Sincronización

Locking

- Solo un thread a la vez puede acceder a un recurso.
- Esto produce un **cuello de botella** a medida que crece el numero de threads.
- **Atenta a la paralelización**. Por mas que se corra un thread por procesador, no se gana tiempo. Cada thread se bloquea constantemente en espera de un recurso.
- **Se puede bajar el locking particionado el recurso compartido**. Es una estrategia muy usada por *parallel streams*.

Sincronización

Confinamiento

- Es preferible que cada thread acumule su resultado(parcial) y luego reagrupar; que sincronizar un recurso para ir agregando resultados a medida que son generados.
- Es la ausencia de locking.
- **La alternativa más efectiva**.

Sincronización

Inmutabilidad

- En vez de mantener una colección compartida entre varias threads, la idea es crear una nueva colección inmutable cuando necesito guardar un nuevo resultado.
- Es la ausencia de locking.
- **Es preferible al locking, en el caso en que no sea posible el confinamiento.**

Inmutabilidad

- Los lenguajes funcionales están empezando a convertirse en alternativas reales con la llegada de procesadores con múltiples unidades de ejecución.
- Uno de los conceptos que suele venir asociado a la **programación funcional** es la **inmutabilidad**.

Inmutabilidad

¿Pero que es inmutabilidad?

- Fácil, **algo inmutable no se puede modificar.**
- En OOP, **si un objeto es inmutable, no se puede modificar su estado.**

¿Hace falta sincronizar el acceso a un objeto/estructura inmutable?

- Si **solo puedo leer estado** y no modificarlo, **para que** lo voy a **sincronizar**!
- Por eso los lenguajes funcionales son felices en este sentido! ;)

Inmutabilidad

Bueno, ¿Y cuando aplico inmutabilidad?

Depende... todo depende…

Si todo es inmutable lo primero que uno piensa es:

> Ah... pero estoy asesinado al Garbage Collector!....

Inmutabilidad

Vamos por otro camino:

- ***Elixir*** es un lenguaje funcional que corre sobre una VM pensada para ejecutar aplicaciones distribuidas en realtime (*Beam VM*).
- *Beam* consume poca memoria a pesar de que todas las estructuras de datos son inmutables.

¿Como lo hace?

Reutiliza las estructuras de datos en forma parcial o completa al construir nuevas estructuras *(Programing Elixir book)*.

Inmutabilidad

Ejemplo:

```
iex(1)> list1 = [3, 2, 1]
[3, 2, 1]
iex(2)> list2 = [4 | list1]
[4, 3, 2, 1]
```

- En la mayoría de los lenguajes `list2` sería un nuevo objeto lista que contiene todos los elementos, copia (ya sea el valor o referencia) de los elementos de `list1`.
- En **elixir** ya sabemos que `list1` nunca va a cambiar, entonces simplemente se construye la lista `list2` con `4` como cabeza y `list1` como tail.

Inmutabilidad

Beam también tiene *GC* como *JVM*.

¿Pero cómo hace para no llenar el heap?

- Se favorece correr todo en muchos pero muchos procesos (OTP).
- Cada proceso tiene su propio heap.
- En cuanto más procesos tengo, más se fragmenta el heap en **heaps pequeños**.
- Esto hace que el ***GC* se ejecute muy rápido**.

Inmutabilidad

¿Y en la *JVM*?

- Tengo entendido que solo ciertos tipos básicos son inmutables (`String`).
- Solo hay un **heap compartido**.
- Si se llena seguido, hay que ejecutar el *GC* más seguido.
- **El GC demora mas a medida que crece el heap.**

```
java.lang.OutOfMemoryError: Java heap space error
```

Inmutabilidad

Conclusión: Usemos inmutabilidad siempre y cuando no generemos objetos a lo pavote.

¿Que es generar objetos a lo pavote?

Bueno, todo depende de cuan seguido cambia el estado de un objeto y de la cantidad de instancias:

- **Si un objeto cambia constantemente de estado**, tiene mucho estado y hay muchas instancias, **tal vez no sea una buena idea hacerlo inmutable**.
- **Si un objeto tiene pocos cambios de estado**, por más que tenga muchas instancias, **puede convenir pensarlo como inmutable**.

Inmutabilidad

Lo idea es que todos los objetos sean inmutables, pero bueno hay que encontrar un **tradeoff para no asesinar a la *JVM***.

Inmutabilidad: Algo mas...

¿Como se define un objeto inmutable en java?

- Declara todos los atributos con el modificador `final`.
- Inicializa el estado del objeto por constructor únicamente.
- Setters son mala palabra (En general. Causan efecto de lado).
- Componer objetos inmutable únicamente.

¿Y si hay atributos que son colecciones?
Usa ***unmodifiable collections*** del *SDK* o ***Guava*** que tiene colecciones inmutables.

```
Collections.unmodifiableList(list)
Collections.unmodifiableSet(set)
Collections.unmodifiableMap(map)
...
```

```
ImmutableList.copyOf(list)
ImmutableSet.copyOf(set)
ImmutableMap.copyOf(map)
...
```

Locking

- Asegura que sólo un thread a la vez pueda modifique una variable.
- Esta modificación se realiza dentro de una **zona crítica**.
- Una zona crítica es una **porción de código** que se ejecuta en forma atómica.
- Cada thread se encolan en espera de su turno.

Locking

Reentrant lock

Es la forma explícita de hacer locking en java.

```
Lock locker = new ReentrantLock();
...
locker.lock();
try {
   // Critical section
} finally {
   locker.unlock();
}
```

Locking

Reentrant lock

- El primer thread en entrar bloquea la zona.
- El siguiente thread queda bloqueado en espera.
- El primer thread puede ejecutar normalmente o arrojar una excepción, pero siempre se ejecuta el finally asegurando el unlock.

Locking

Syncronized keyword

- Es una forma más concisa y declarativa de hacer locking.
- En general no se usa reentrant locks, pero ayuda a entender como funciona syncronized.
- Hace uso del lock intrínseco que tiene todos los objetos en java.
- 2 sabores:
  - Syncronized block: Es una zona critica donde se locked el acceso a un objeto.
  - Syncronized method: El objeto receptor del mensaje es el recurso a lockear.

Locking

Syncronized block

```
synchronized (obj) {
   // Critical section
}
```

Se puede pensar como:

```
obj.getIntrinsicLock().lock(); // getIntrinsicLock() propiedad hipotetica
try {
   // Critical section
} finally {
   obj.getIntrinsicLock().unlock();
}
```

Locking

Syncronized method

```
public synchronized void method() {
   // Critical section
}
```

Se puede pensar como:

```
public void method() {
   this.getIntrinsicLock().lock(); // getIntrinsicLock() propiedad hipotetica
   try {
       // Critical section
   } finally {
       this.getIntrinsicLock().unlock();
   }
}
```

Locking

Wait & notify

- Son mensajes que pueden contestar objetos de la clase *`Object`*.
- En conjunto funcionan como una **cola de espera**.
- Se puede aplicar en problemas del tipo productor-consumidor.

Locking

Wait & notify

- Si N threads invocan *obj.wait()* quedan bloqueados en una **cola de espera** ordenada por orden de llegada.
- *obj.notify()* despierta al primer thread en la cola. Luego, cada thread se va despertando a medida que se vuelve a invocar *obj.notify()*
- Si hacemos N *obj.notify()* antes de que haya threads en la cola, los siguientes hilos que hagan *obj.wait()* no se quedarán bloqueados.
- Ambos métodos deben invocarse dentro de un block/método syncronized.

Locking

Wait & notify

Implementar una cola utilizando wait y notify que cumpla con los siguiente requisito: Si la cola está vacía, El próximo thread que invoque queue.pop(element) se bloquea.

Locking

Wait & notify

```
@Test(timeout = 2000L)
public void testAsyncPush() throws InterruptedException {
    // Prepare
    Integer value = 1;
    Queue<Integer> queue = new Queue<>();
    Thread asyncPush = new Thread(() -> { sleep(1000L); queue.push(value); });

    // Perform
    asyncPush.start();

    // Asserts
    assertThat(queue.pop(), is(equalTo(value)));
}
```

Locking

Wait & notify

```
public class Queue<T> {
   private Node<T> head, tail;
   public synchronized Queue<T> push(T value) {
       Node<T> node = new Node<>(value);
       if (head == null) { head = node; notify(); } else tail.next(node);
       tail = node;
       return this;
   }
   public synchronized T pop() throws InterruptedException {
       if (head == null) wait();
       Node<T> node = head;
       head = node.next();
       return node.value();
   }
}
```

Locking

Wait & notify

```
class Node<D> {
   private D value;
   private Node<D> next;

   public Node(D value) { this.value = value; }

   public D value() { return value; }
   public Node<D> next() { return next; }
   public void next(Node<D> next) { this.next = next; }
}
```

Locking

Hasta ahora todo muy lindo y mucho para practicar pero…

suena a que estamos reinventando la rueda!

Estructuras de datos Threadsafe

- Son estructuras de datos que incorporar locking para sincronizar su acceso/modificación.
- Particionan sus datos para disminuir el locking.
- Pobre consistencia con iteradores.
  - El iterador muestra el estado de una colección antes de ser creado.
  - No se arroja `ConcurrenModificationException` cuando se modifica una colección mientra es iterada.

Estructuras de datos Threadsafe

Concurrent Hash Maps

- Tiene locking incorporado.
- Es suficiente con esto?
  - Supongamos que queremos contar la frecuencia de un conjunto de palabras. Tengo un hash con palabras como key y un contador como valor. Cada thread incrementa al encontrar una palabra.

Estructuras de datos Threadsafe

Concurrent Hash Maps

Entonces actualizamos el contador como sigue:

```
Map<String, Long> wordsFrequency = new ConcurrentHashMap<>();
...
Long oldCount = wordsFrequency.get("word");
Long newCount = oldCount == null ? 1 : oldCount + 1;
wordsFrequency.put("word", newCount);
```

Estructuras de datos Threadsafe

Concurrent Hash Maps

- Mal!. No estamos sincronizando la modificación del valor. Otro thread puede estar incrementando al mismo tiempo.
- Entonces? bloqueamos a mano?. No, la colección lo maneja.

```
wordsFrequency.compute("word", (key, value) -> value == null ? 1 : value + 1);
```

- *Compute* es atómico (Lockea una partición). Ningún otro thread puede operar si se está ejecutando este lambda.
- *ComputeIfPresent*, *computeIfAbsent* y *merge* son variantes.

Estructuras de datos Threadsafe

Concurrent Hash Maps

```
wordsFrequency.merge("word", 1L, (count, increment) -> count + increment);
```

1L es el valor inicial y el incremento a la vez.

Importante

- Estos lambdas (atómicos) lockean la colección mientra se ejecuten.
- Agregar el mínimo codigo necesario para actualizar el valor.
- Lo mas performante possible.

Estructuras de datos Threadsafe

Examples

```
public class ExecutorServiceTest {

  private Map<String, Integer> map;

  @Before
  public void setUp() { map = new ConcurrentHashMap<>(); }


  @Test
  public void incrementKeyValueUsingCompute() {
     // Perform
     map.compute(KEY, (key, value) -> value == null ? 1 : value + 1);

     // Asserts
     assertThat(map.get(KEY), is(equalTo(1)));
  }
```

Estructuras de datos Threadsafe

Examples

```java
@Test
public void incrementKeyValueUsingMerge() {
    // Perform
    map.merge(KEY, 1, (currentValue, defaultValue) -> currentValue + defaultValue);
    map.merge(KEY, 1, (currentValue, defaultValue) -> currentValue + defaultValue);

    // Asserts
    assertThat(map.get(KEY), is(equalTo(2)));
}

@Test
public void incrementKeyValueUsingIfAbsentAndIfPresent() {
    // Perform
    map.computeIfAbsent(KEY, key -> { return 0; });
    map.computeIfPresent(KEY, (key, value) -> { return value + 1; });

    // Asserts
    assertThat(map.get(KEY), is(equalTo(1)));
}
```

Estructuras de datos Threadsafe

Examples

```
  @Test
  public void incrementKeyValueUsingPutIfAbsentAndIfPresent() {
     // Perform
     map.putIfAbsent(KEY, 0);
     map.computeIfPresent(KEY, (key, value) -> { return value + 1; });

     // Asserts
     assertThat(map.get(KEY), is(equalTo(1)));
  }

}
```

Estructuras de datos Threadsafe

Blocking queues

- Comúnmente usada para coordinar trabajo entre tareas.
- Las tareas que producen elementos, insertan en la cola.
- Las tareas que consumen, remueven elementos de la cola.
- Si agrego un elemento cuando la cola está llena o borro cuando está vacía, la operación se bloquea.
- Es una forma de balancear la carga de trabajo.
    - Si los productores son lentos, los consumidores quedan bloqueados a la espera de un nuevo elemento.
    - Si los productores más rápido de los consumidores, quedan bloqueados hasta que un consumidor remueve un elemento.

Estructuras de datos Threadsafe

Blocking queues

- Hay 3 categorías de métodos sincronizados según la acción que se realiza cuando la cola está llena o vacía.
  - Los que bloquean: *put* y *take*.
  - Los que arrojan una excepción: *add*, *remove* y *element*.
  - Los que retornan un fail indicator(retorna null), *offer*, *poll* y *peek*.

Estructuras de datos Threadsafe

Blocking queues

Method	Normal Action	Error Action
`put`	Adds an element to the tail	Blocks if the queue is full
`take`	Removes and returns the head element	Blocks if the queue is empty
`add`	Adds an element to the tail	Throws an `IllegalStateException` if the queue is full
`remove`	Removes and returns the head element	Throws a `NoSuchElementException` if the queue is empty
`element`	Returns the head element	Throws a `NoSuchElementException` if the queue is empty
`offer`	Adds an element and returns `true`	Returns `false` if the queue is full
`poll`	Removes and returns the head element	Returns `null` if the queue is empty
`peek`	Returns the head element	Returns `null` if the queue is empty

Estructuras de datos Threadsafe

Blocking queues

`ArrayBlockingQueue`

- Se fija su tamaño al momento de crearla.
- Su tamaña no se puede cambiar una vez fijada.
- El tamaño máximo posible es Integer.MAX_VALUE
- Crear un array de gran tamaño puede consumir mucho espacio en memoria.
- No se puede cambiar su tamaño una vez creada.

Estructuras de datos Threadsafe

Blocking queues

`LinkedBlockingQueue`

- Su tamaño se incrementa dinámicamente a medida que sea necesario (Se agregan nodos).
- Por defecto el tamaño máximo es Integer.MAX_VALUE.
- Crear un LinkedList de gran tamaño puede no consume mucho espacio en memoria.
- Se puede limitar su tamaño una vez creada.

Estructuras de datos Threadsafe

Otras estructuras

- `ConcurrentSkipListMap`
- `ConcurrentSkipListSet`
- `CopyOnWriteArrayList`
- `CopyOnWriteArraySet`
- `AtomicInteger`
- `AtomicIntegerArray`
- `AtomicIntegerFieldUpdater`
- `AtomicReference`
- `AtomicReferenceArray`
- `AtomicReferenceFieldUpdater`

Paralelismo & Concurrencia

¿Con qué herramientas cuento para usa paralelismo/concurrencia en *Java*?

3 sabores:

- *`Executors`*
- *`For/Join`*
- *`Parallel streams`*

Executors

- La clase `ExecutorService` es parte de *`java.util.concurrent`*.
- Se introduce en Java 5.
- `ExecutorService` permite ejecutar tareas concurrentes en un `Executor`.
- Las tareas se ejecutan en un pool de threads, donde cada thread se reutiliza.
- Ya no es necesario crear y manejar thread manualmente!

Executors

La clase `Executors` tiene factory methods para construir distintas configuraciones de `ExecutorService`:

- `newSingleThreadExecutor`
- `newFixedThreadPool`
- `newScheduledThreadPool`
- `newCachedThreadPool`

Executors

`SingleThreadExecutor`

- Instancia un executor con un unico thread worker.

`FixedThreadPool`

- Instancia un executor con un conjunto fijo de thread workers.
- Estos corren instancias de Runnable.

`ScheduledThreadPool`

- Instancia un executor que puede correr runnables luego un delay dado.

Executors

`newCachedThreadPool`

- Instancia un executor con un thread pool que crea un nuevo thread solo cuando es necesario y reutiliza los thread que creó previamente.
- Este tipo de thread pool es usado para mejorar la performance cuando es necesario correr muchas tareas de corta duración.
- Cuando no hay un thread disponible para ejecutar se crea uno nuevo y se agrega al pool.

Executors

`newCachedThreadPool`

- Los thread que no son utilizados luego de 6 segundos se terminan y remueven del poll cache.
- De esta manera si el poll no se usa por un largo periodo no tengo thread instanciados consumiendo menos recursos del sistema.
- **Ojo!. Si no hay límite en el número y duración de las tareas podemos estar creando y borrando thread constantemente y consumir CPU a lo pavote!**

`Ver:` EfficientThreadPoolExecutor

Executors

Runnable

- Interfaz funcional usada por la clase `Thread/ExecutorService` para ejecutar una tarea.

```
@FunctionalInterface
public interface Runnable { void run(); }
```

- Su ejecución **NO** devuelve un resultado.

```
ExecutorService service = Executors.newSingleThreadExecutor();
service.execute(() -> System.out.println("Hello " + Thread.currentThread().getName()));
```

Executors

Callable

- Interfaz funcional usada por `ExecutorService` para correr una tarea.

```
@FunctionalInterface
public interface Callable<V> { V call() throws Exception; }
```

- Su ejecución **SI** devuelve un resultado.

```
ExecutorService service = Executors.newSingleThreadExecutor();
Future<String> result = service.submit(() -> "Hello " + Thread.currentThread().getName());
System.out.println(result.get());
```

Executors

Callable

- Al ejecutar submit de un `Callable,` como resultado obtenemos un `Future`

```
public interface Future<V> {
   boolean cancel(boolean var1);
   boolean isCancelled();
   boolean isDone();
   V get() throws InterruptedException, ExecutionException;
   V get(long var1, TimeUnit var3) throws InterruptedException, ExecutionException,
                                          TimeoutException;
}
```

- Al ejecutar submit, **el thread que lo ejecuto no queda en espera**.
- Para obtener el resultado de la operación hay que pedirselo al `Future`

Executors

Callable

- Un `Future` te permite realizar las siguientes acciones:
  - `V get()`
    - Se espera por el resultado de la ejecución del callable.
    - Si el callable continúa en ejecución se bloque el thread que ejecutó esta acción hasta que este finaliza.
  - `V get(long timeout, TimeUnit unit)`
    - Se espera por un tiempo x el resultado de la ejecución del callable.
    - Se arroja una `java.util.concurrent.TimeoutException` cuando se llega al timeout.
  - `boolean isDone()`
    - Retorna true si es callable terminó sus ejecucion.

Executors

Callable

- Un `Future` te permite realizar las siguientes acciones:
  - `void cancel(boolean mayInterruptIfRunning)`
    - Intenta cancelar la ejecución del callable.
    - Este intento puede fallar si la tarea ya se completo, ya fue cancelada o no es cancelable (?).
    - Si la tarea no comenzó se aborta su ejecución.
    - Si la tarea está en ejecución, el parámetro mayInterruptIfRunning determina si puedo abortar o no el thread que ejecuta la tareas.
    - Luego de la invocación de este método isDone() retorna true.
  - `boolean isCancelled()`
    - Retorna true si se canceló la ejecución del callable.

Executors

Callable

- Metodos de `ExecutorService`
  - `Future<T> submit(Runnable runnable)`
  - `void execute(Runnable runnable)`
  - `List<Future<T>> invokeAll(callables)`
  - `List<Future<T>> invokeAll(callables, long timeout, TimeUnit unit)`
    - Devuelve una list de Futures finalizados.
  - `T invokeAny(callables)`
  - `T invokeAny(callables, long timeout, TimeUnit unit)`
    - Devuelve el resultado del primer Callable finalizado.

Executors - **Examples**

```
public class ExecutorServiceTest {

    @Test
    public void testExecuteUsingARunnableBlock() throws InterruptedException {
        // Prepare
        List<String> threadNames = newArrayList();
        Runnable lambda = () -> threadNames.add(Thread.currentThread().getName());
        ExecutorService service = Executors.newSingleThreadExecutor();

        // Perform
        service.execute(lambda);

        // Asserts
        service.awaitTermination(1, SECONDS);
        assertThat(threadNames, hasItem("pool-5-thread-1"));
    }
```

Executors - **Examples**

```
@Test
public void testSubmitUsingACallableBlock() throws ExecutionException, InterruptedException {
    // Prepare
    Callable<String> lambda = () -> Thread.currentThread().getName();
    ExecutorService service = Executors.newSingleThreadExecutor();

    // Perform
    Future<String> threadName = service.submit(lambda);

    // Asserts
    assertThat(threadName.get(), is(equalTo("pool-8-thread-1")));
}
```

Executors - **Examples**

```java
@Test
@SuppressWarnings("unchecked")
public void testInvokeAll() throws InterruptedException, ExecutionException {
    // Prepare
    Callable<String> lambda = () -> Thread.currentThread().getName();
    List<Callable<String>> invocations = newArrayList(lambda, lambda);
    Function<Future<String>, String> getName = future -> {
        try { return future.get(); }
        catch (Exception exception) { fail(exception.getMessage()); return ""; }
    };
    ExecutorService service = Executors.newFixedThreadPool(2);

    // Perform
    List<String> threadNames = service.invokeAll(invocations).stream().map(getName).collect(toList());

    // Asserts
    assertThat(threadNames, hasItem("pool-3-thread-1"));
    assertThat(threadNames, hasItem("pool-3-thread-2"));
}
```

Executors - **Examples**

```java
@Test
@SuppressWarnings("unchecked")
public void testInvokeAny() throws ExecutionException, InterruptedException {
    // Prepare
    Callable<String> lambda = () -> Thread.currentThread().getName();
    List<Callable<String>> invocations = newArrayList(lambda, lambda);
    ExecutorService service = Executors.newFixedThreadPool(2);

    // Perform
    String threadName = service.invokeAny(invocations);

    // Asserts
    assertThat(newArrayList("pool-4-thread-1", "pool-4-thread-2"), hasItem(threadName));
}
```

Executors - **Examples**

```
@Test(expected = CancellationException.class)
public void testCancelAndInterruptCallableWhileRun() throws ExecutionException, InterruptedException {
    // Prepare
    final boolean mayInterruptIfRunning = true;
    final Boolean[] wasInterrupted = {false};
    Callable<Integer> lambda = () -> {
        try { Thread.sleep(100L); } catch (InterruptedException e) { wasInterrupted[0] = true; }
        return 1;
    };
    ExecutorService service = Executors.newSingleThreadExecutor();
    Future<Integer> future = service.submit(lambda);

    // Perform
    Thread.sleep(50L);
    future.cancel(mayInterruptIfRunning);

    // Asserts
    Thread.sleep(150L);
    assertThat(wasInterrupted[0], is(equalTo(true)));
    future.get();
}
```

Executors - **Examples**

```
@Test(expected = CancellationException.class)
public void testCancelCallableWhileRun() throws ExecutionException, InterruptedException {
    // Prepare
    final boolean mayInterruptIfRunning = false;
    final Boolean[] wasInterrupted = {false};
    Callable<Integer> lambda = () -> {
        try { Thread.sleep(100L); } catch (InterruptedException e) { wasInterrupted[0] = true; }
        return 1;
    };
    ExecutorService service = Executors.newSingleThreadExecutor();
    Future<Integer> future = service.submit(lambda);
    Thread.sleep(50L);

    // Perform
    future.cancel(mayInterruptIfRunning);

    // Asserts
    assertThat(future.isCancelled(), is(equalTo(true)));
    Thread.sleep(100L); assertThat(wasInterrupted[0], is(equalTo(false)));
    future.get();
}
```

Executors - **Examples**

```
@Test(expected = CancellationException.class)
public void testCancelCallableBeforeRun() throws ExecutionException, InterruptedException {
    // Prepare
    final boolean mayInterruptIfRunning = false;
    Callable<Integer> lambda = () -> {
        Thread.sleep(9999999999L);
        return 1;
    };
    ExecutorService service = Executors.newSingleThreadExecutor();
    Future<Integer> future = service.submit(lambda);

    // Perform
    future.cancel(mayInterruptIfRunning);

    // Asserts
    assertThat(future.isCancelled(), is(equalTo(true)));
    future.get();
}
```

Executors - **Examples**

```
@Test
public void testCancelCallableAfterRun() throws ExecutionException, InterruptedException {
    // Prepare
    Callable<Integer> lambda = () -> {
        Thread.sleep(50L);
        return 1;
    };
    ExecutorService service = Executors.newSingleThreadExecutor();
    Future<Integer> future = service.submit(lambda);
    Thread.sleep(100L);
    final boolean mayInterruptIfRunning = false;

    // Perform
    future.cancel(mayInterruptIfRunning);

    // Asserts
    assertThat(future.isCancelled(), is(equalTo(false)));
    assertThat(future.get(), is(equalTo(1)));
}
```

Fork/Join

- Se introdujo en Java 7 para extender el package *concurrency* para soportar paralelismo por hardware.
- Cuando el problema a resolver es del tipo divide y vencerás *Fork/Join* le pasa el trapo a *ThreadPool*.

Parallel streams

- Se introduce en Java 8.
- Es una implementación de iteradores internos a diferencias de los externos en versiones anteriores del JDK.
- Esta implementado con el framework *`Fork/Join`*.
- Para crear un stream paralelo desde una collecion se invoca metodo `collection.streamParallel()`
- En cualquier momento se puede paralelizar un stream secuencia con `stream.parallel()`

Parallel streams

- Cada vez que se usa `streamParallel()/parallel()` se invoca al metodo `ForkJoinPool.commonPool()`.
- El tamaño de pool depende del número de procesadores físicos.
- En un dual core con hyper threading

```
ForkJoinPool commonPool = ForkJoinPool.commonPool();
System.out.printf("Pool size: %s\n", commonPool.getParallelism());
```

```
> Pool size: 3
```

- Se puede cambiar el tamaño con la siguiente propiedad:

```
-Djava.util.concurrent.ForkJoinPool.common.parallelism=10
```

```
> Pool size: 10
```

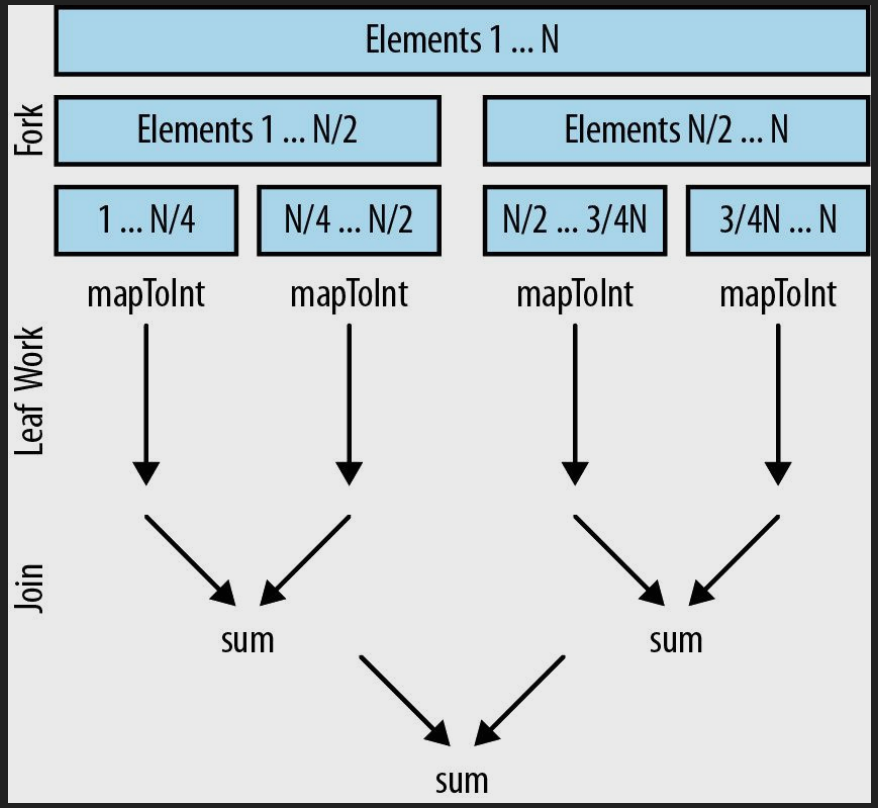
Parallel streams

Habíamos dicho que *Parallel streams* está implementado con *Fork/Join* pero cómo funciona?

Entonces, si tenemos un procesador con 4 nucleos fisicos...

```
public static int addIntegers(List<Integer> values) {
   return values
       .parallelStream()
       .mapToInt(i -> i)
       .sum();
}
```

Parallel streams



Parallel streams

Pasos

1. Se descompone la colección en 4 partes.
2. Se computa cada parte (leaf work) en paralelo, cada una en un thread distinto. (La tarea hace un unboxing del objeto Integer al tipo valor `int`).
3. Finalmente, merge de los resultados (`sum`).

```
public static int addIntegers(List<Integer> values) {
    return values // → Data Source
        .parallelStream() // → Split(4 cores)
        .mapToInt(i -> i) // → Compute each
        .sum(); // → Join
}
```

Parallel streams

Las estructuras de datos utilizadas para realizar el paso de descomposición, se pueden clasificar en 3 grupos según su desempeño:

- **Buenas:** *`ArrayList`*, *`IntStream`* y *`IntStream.range`* son estructuras de acceso aleatorio. Se puede splitear relativamente fácil.
- **Ok:** *`HashSet`* y *`TreeSet`* no son fáciles de descomponer, pero se puede.
- **Malas:** Algunas estructuras no se pueden desacoplar (Por ejemplo 0(N)).
    - Splitear *`LinkedList`* tiene un costo computacional alto.
    - No se puede conocer la cantidad de elementos de *`Stream.iterate`* y *`BufferReader.lines`*.

Parallel streams

Stateless operations

Es la forma de obtener el mayor rendimiento.

Ej.: *`map`*, *`flatMap`*, *`filter`*.

Statefull operations

Tiene el overhead de construir(fullscan) y sincronizar estructuras.

Ej.: *`sort`*, *`distinct`*, *`limit`*.

Parallel streams

To be or not to be?

Existe un grupo de factores que influencia directamente sobre el tiempo de ejecución de un proceso:

Para mi problema particular, ¿Voy a lograr mejores tiempos con *Parallel streams* o es mejor *Sequencial Streams*?

Parallel streams

Factores

- Cantidad de datos
- Estructura de datos de origen
- Boxing
- Numero de nucleos of cores
- Costo por elemento

Parallel streams

Cantidad de datos

Para poder paralelizar un problema es necesario invertir tiempo en **descomponer** los datos, luego procesarlos en paralelo y finalmente reagruparlos.

Si tengo suficientes datos, y procesar cada dato demora un tiempo considerable → Tiene sentido tomarse el tiempo para descomponer y reagrupar.

Parallel streams

Estructura de datos de origen

Cada pipeline de operaciones procesa un parte de los datos iniciales, que en general es una colección de elementos.

Ojo, es muy fácil caer en subdividir esta colección en otros procesos paralelos, esto tiene el costo de aumentar el tiempo de ejecución.

→ **subdividi tu proceso lo menos posible**

Parallel streams

Boxing

Es más eficiente operar tipos primitivos que sus pares en Objetos.

- *int → Integer*
- *long → Long*
- *double → Double*
- *etc...*

Parallel streams

Numero de nucleos

- Si tengo un solo núcleo, no tiene sentido paralelizar.
- A medida que crece el numero, paralelizar mejora mas los tiempos.
- En realidad no importa el numero de nucleos que tenga la máquina **sino los que tenga asignados mi proceso**.

→ **Pensemos que hay otros procesos corriendo en la misma máquina afectando indirectamente la performance de mi proceso.**

Parallel streams

Costo por elemento

Así como, cuanto más datos tenemos mas costo al descomponer y reagrupar tenemos…

Cuanto más demore el cómputo de un elemento de la colección →mejores tiempos voy a lograr al paralelizando mi proceso.

Conclusiones

¿Entonces cuando uso sincronización y cuando no?

- Evitemos usarla es un cuello de botella. El confinamiento es lo mejor.
- Si hay que sincronizar, usa estructuras Threadsafe(No reinventar la rueda). Sincronizar a mano es propenso a errores difíciles de detectar.
- Cuando sea necesario implementar una estructura Threadsafe, seguro vas a usar synchronized, wait, notify, fork/join o capas ni hace falta si compones otras estructuras Threadsafe.

Conclusiones

Si puedo usar tanto ExecutorService como Parallel Streams para PARALELIZAR trabajo, ¿Cuál es la principal diferencia?

- `Parallel streams` es un método super declarativo y podes hacer lo mismo que con `ExecutorService` en muy poco código.
- `ExecutorService` es adecuado cuando se requiere lanzar N tareas y no importa si se ejecutan en paralelo o no.
- Si mis tareas se bloquea en espera de un recurso, no tiene mucho sentido paralelizar ya que dedicas procesador a tareas que no lo usan.
- Si son tareas CPU-bound usar `Parallel streams` es una buena opción.

Conclusiones

¿Que diferencias en tiempos de ejecución hay entre `ExecutorService`, `Fork/join` y `Parallel streams`?

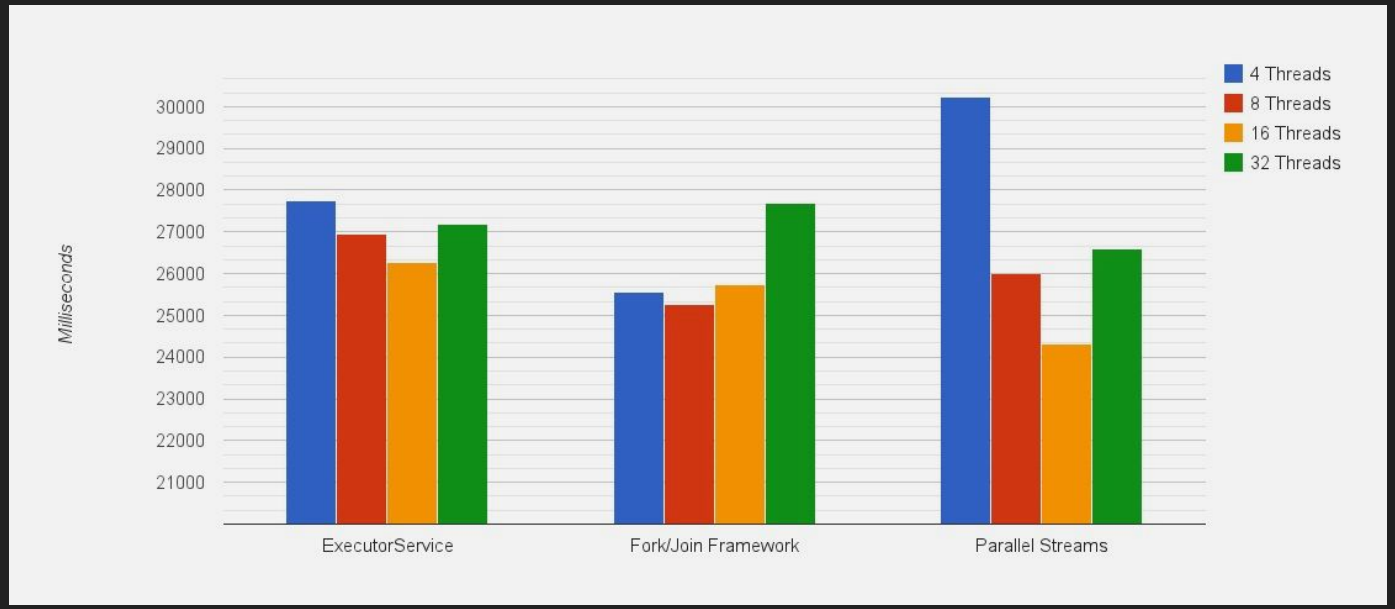
Blog.takipi.com

- Tomaron 2 tareas, una de mucho consumo de CPU y otra de E/S intensiva.
- Todo sobre una máquina 8 núcleos.
- Realizaron variaciones entre 4, 8, 16 y 32 threads.
- 260 pruebas.

Ver: Framework Fork/Join vs. flujos paralelos vs. ExecutorService: el benchmark definitivo al Fork/Join

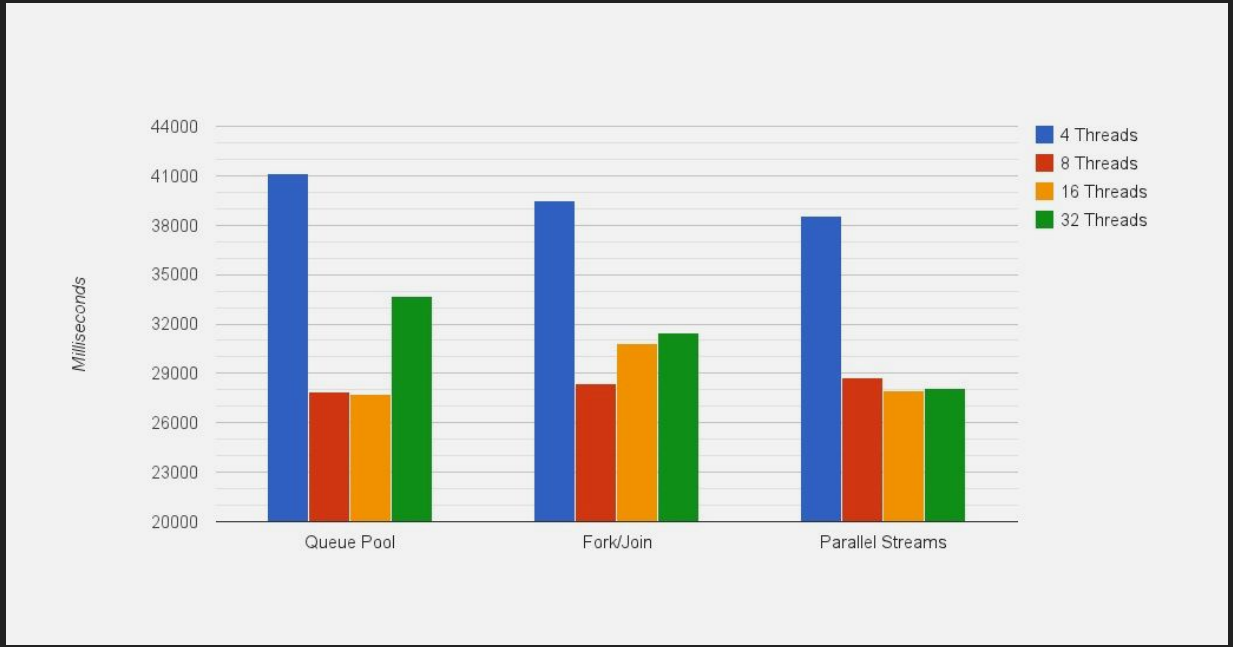
Conclusiones

I/O Bound: Indexado de un archivo de 6GB con 5.8M de líneas de texto



Conclusiones

CPU Bound: 1.530.692.068.127.007.263 es primo?



Conclusiones

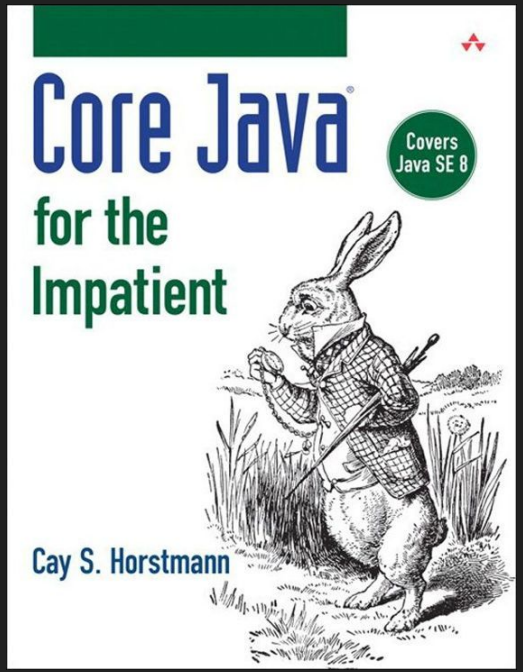
Finalmente, estas son reglas a seguir para lograr procesos más eficientes, pero **siempre la mejor forma de asegurar una mejora es medir**.

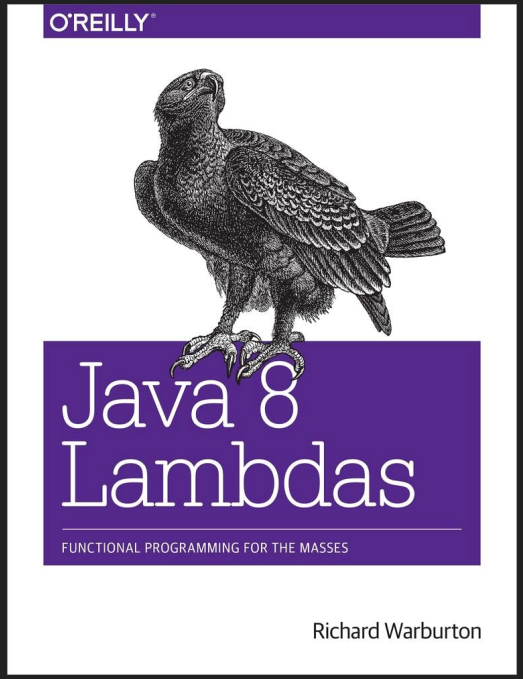
Github

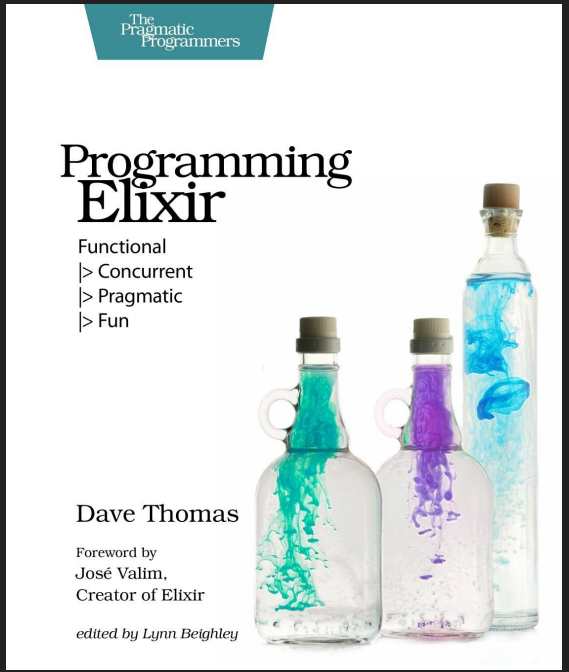
github.com/adrianmarino/parallelismAndConcurrence

Referencias

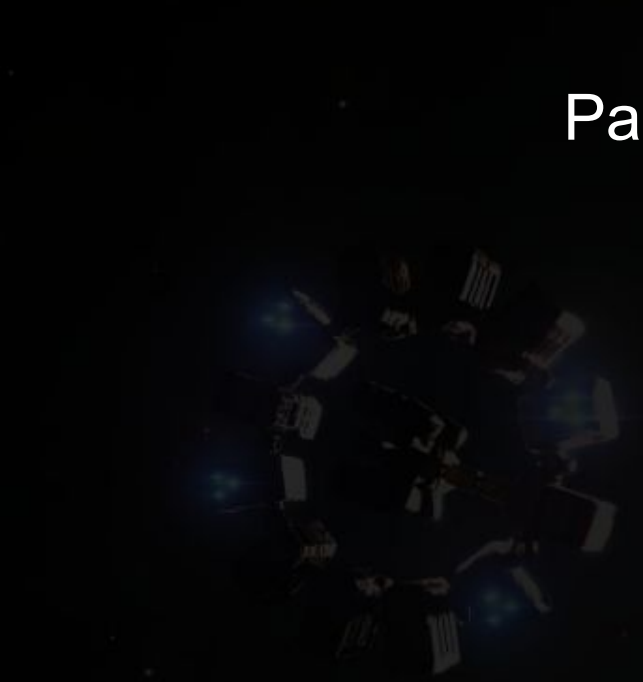
Google







Paralelismo & Concurrencia



¿Preguntas?