

Paralelismo & Concurrency

Guia practica (Java 8)

Mail: adrianmarino@gmail.com

Github: <https://github.com/adrianmarino>

Paralelismo

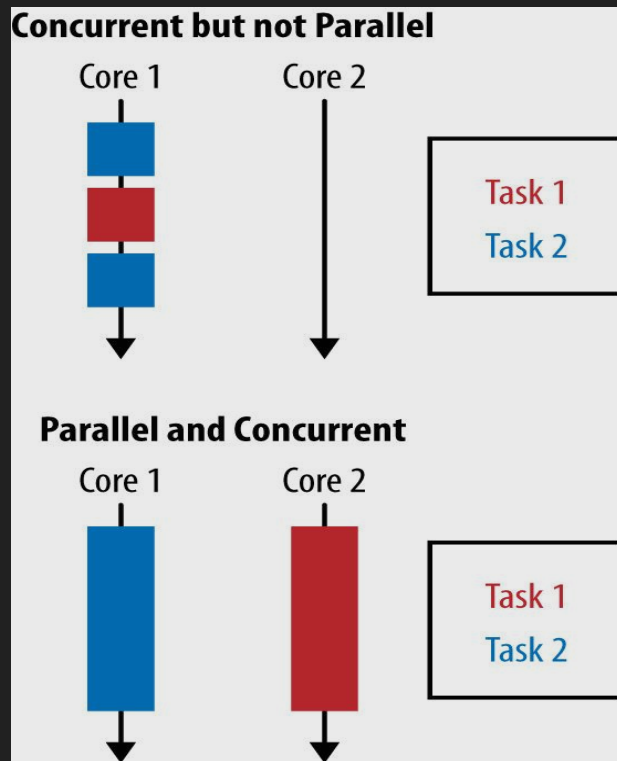
- Es la ejecución simultánea de dos o más tareas.
- Las tareas se asignan a cada núcleo.
- Si ejecutamos un proceso compuesto por varias tareas asignadas a distintas máquinas, se está distribuyendo su procesamiento.

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).



Concurrencia vs. Paralelismo

- 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 subtarear aisladas y luego esperar la finalización de todas estas, para finalmente construir un resultado.

...Esto no está escrito en piedra...

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 más que se corra un thread por procesador, no se gana tiempo. Cada thread se bloquea constantemente en espera del 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 necesite modificarla.
- 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 qué 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.

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 cola.

Inmutabilidad

Beam también tiene GC como la 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?

- Solo ciertos tipos básicos son inmutables (`String`).
- Solo hay un **heap compartido** (Con dos partes: Young y old Generation).
- Si se llena seguido, hay que ejecutar el GC más seguido.
- **El GC demora más 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 todos los objetos que necesitemos que sean inmutables lo sean, 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

Synchronized keyword

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

Locking

Synchronized block

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

Se puede pensar como:

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

Locking

Synchronized method

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

Se puede pensar como:

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

Locking

Wait & notify

- Son mensajes que pueden responder 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

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

Locking

Wait & notify

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

Locking

Wait & notify

```
@Test(timeout = 2500L)
public void testAsyncPush() throws InterruptedException {
    // Prepare
    Thread asyncPush = new Thread(() -> {
        sleep(1000L); queue.push(1);
        sleep(1000L); queue.push(2);
    });

    // Perform
    asyncPush.start();

    // Asserts
    assertThat(queue.pop(), is(equalTo(1)));
    assertThat(queue.pop(), is(equalTo(2)));
}
```

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; } else tail.next(node);
        tail = node;
        notify();
        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...

suenan a que estamos reinventando la rueda, no?

Estructuras de datos Threadsafe

- Son estructuras de datos que incorporan 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 `ConcurrentModificationException` cuando se modifica una colección mientras es iterada.

Estructuras de datos Threadsafe

ConcurrentHashMap

- Tiene locking incorporado.
- Pero esto es suficiente?
 - Supongamos que queremos contar la frecuencia de un conjunto de palabras.
 - Tenemos un hash donde la clave es una palabra y el valor un contador.
 - Cada thread incrementa el valor al encontrar una palabra.

Estructuras de datos Threadsafe

ConcurrentHashMap

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);
```

Esto es correcto? Que opinan?

Estructuras de datos Threadsafe

ConcurrentHashMap

- 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

ConcurrentHashMap

```
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 mientras se ejecuten.
- Agregar el mínimo código necesario para actualizar el valor.
- Lo más performante posible.

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

```
@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

BlockingQueue

- 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

BlockingQueue

- 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, true/false), `offer`, `poll` y `peek`.

Estructuras de datos Threadsafe

BlockingQueue

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 <code>IllegalStateException</code> if the queue is full
remove	Removes and returns the head element	Throws a <code>NoSuchElementException</code> if the queue is empty
element	Returns the head element	Throws a <code>NoSuchElementException</code> if the queue is empty
offer	Adds an element and returns <code>true</code>	Returns <code>false</code> if the queue is full
poll	Removes and returns the head element	Returns <code>null</code> if the queue is empty
peek	Returns the head element	Returns <code>null</code> if the queue is empty

Estructuras de datos Threadsafe

BlockingQueue

ArrayBlockingQueue

- Se fija su tamaño al momento de crearla.
- Su tamaño 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

BlockingQueue

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 no consume mucho espacio en memoria.
- Se puede modificar su tamaño una vez creada.

Estructuras de datos Threadsafe

Otras estructuras

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

ThreadLocal

- Se utiliza para evitar compartir objetos **non-threadsafe** entre threads.
- Evita la sincronización(confinamiento).
- Cuando un thread invoca al método `threadlocal.get()` obtiene una instancia del recurso.
- Cada thread tiene su propia copia del recurso.

ThreadLocal

```
@Test
public void testFormantNumbersUsingExecutors() {
    // Prepare
    List<Double> totals = newArrayList(1500.15D, 2500.15D, 3500.15D);
    ThreadLocal<NumberFormat> formatter = ThreadLocal.withInitial(NumberFormat::getCurrencyInstance);
    ExecutorService service = newFixedThreadPool(3);
    Function<Future<String>, String> getFutureResult = future -> {
        try { return future.get(); }
        catch (Exception exception) { fail(exception.getMessage()); return null; }
    };

    // Perform
    List<String> stringTotals = totals.stream()
        .map(total -> service.submit(() -> formatter.get().format(total)))
        .map(getFutureResult)
        .collect(toList());

    // Asserts
    assertThat(stringTotals, hasItems("$1,500.15", "$2,500.15", "$3,500.15"));
}
```

ThreadLocal

```
@Test
public void testFormantNumbersUsingParallelStream() {
    // Prepare
    List<Double> totals = newArrayList(1500.15D, 2500.15D, 3500.15D);
    ThreadLocal<NumberFormat> formatter = ThreadLocal.withInitial(NumberFormat::getCurrencyInstance);

    // Perform
    List<String> stringTotals = totals.parallelStream()
        .map(total -> formatter.get().format(total))
        .collect(toList());

    // Asserts
    assertThat(stringTotals, hasItems("$1,500.15", "$2,500.15", "$3,500.15"));
}
```

Paralelismo & Concurrency

¿Con qué herramientas cuento para usar paralelismo/concurrency 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 pool cache.
- De esta manera, si el pool no se usa por un largo periodo, se consume 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 invocar el método `submit` de un `Callable`, este responde con un `Future`.

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

- Al invocar `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 bloquea el thread que ejecutó esta acción hasta que el `Callable` finaliza.
 - `V get(long timeout, TimeUnit unit)`
 - Se espera por un tiempo el resultado de la ejecución del `Callable`.
 - Se arroja una `java.util.concurrent.TimeoutException` cuando se llega al timeout.
 - `boolean isDone()`
 - Retorna true si el `Callable` terminó su ejecución.

Executors

Callable

- Un `Future` te permite realizar las siguientes acciones:
 - `void cancel(boolean mayInterruptIfRunning)`
 - 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 tarea, arrojando una `InterruptedException` dentro del `Callable`.
 - Luego, la invocación de este método `isDone` retornara 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 = new ArrayList();  
        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

```
@Test
@SuppressWarnings("unchecked")
public void testInvokeAll() throws InterruptedException, ExecutionException {
    // Prepare
    Callable<String> lambda = () -> Thread.currentThread().getName();
    List<Callable<String>> invocations = new ArrayList(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

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

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

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

Executors - Examples

```
@Test(expected = CancellationException.class)
public void testCancelCallableBeforeRun() throws ExecutionException, InterruptedException {
    // Prepare
    final boolean mayInterruptIfRunning = false;
    Callable<Integer> lambda = () -> {
        Thread.sleep(999999999L);
        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.get(), is(equalTo(1)));
    assertThat(future.isCancelled(), is(equalTo(false)));
}
```

Fork/Join

- Se introdujo en Java 7 para extender el package `concurrency` para soportar paralelismo por hardware.
- Para problemas donde se puede aplicar el principio **divide y vencerás**.
- Está pensado para realizar tareas que pueden dividirse en subtareas más pequeñas, las cuales se ejecutan en paralelo y finalmente se combinan en un resultado único.
- Las tareas son **stateless** y cada una procesa una parte del problema.

Parallel streams

- Se introduce en Java 8.
- Es una implementación de iteradores internos a diferencias de los externos de versiones anteriores del JDK (`collection.iterator()`).
- Otros lenguajes que implementar iteración interna en sus colecciones: Smalltalk, Ruby, Groovy, etc...
- Esta implementado con el framework *Fork/Join*.
- Para crear un stream paralelo desde una colección se invoca `collection.streamParallel()`
- En cualquier momento se puede paralelizar un stream secuencial 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
```

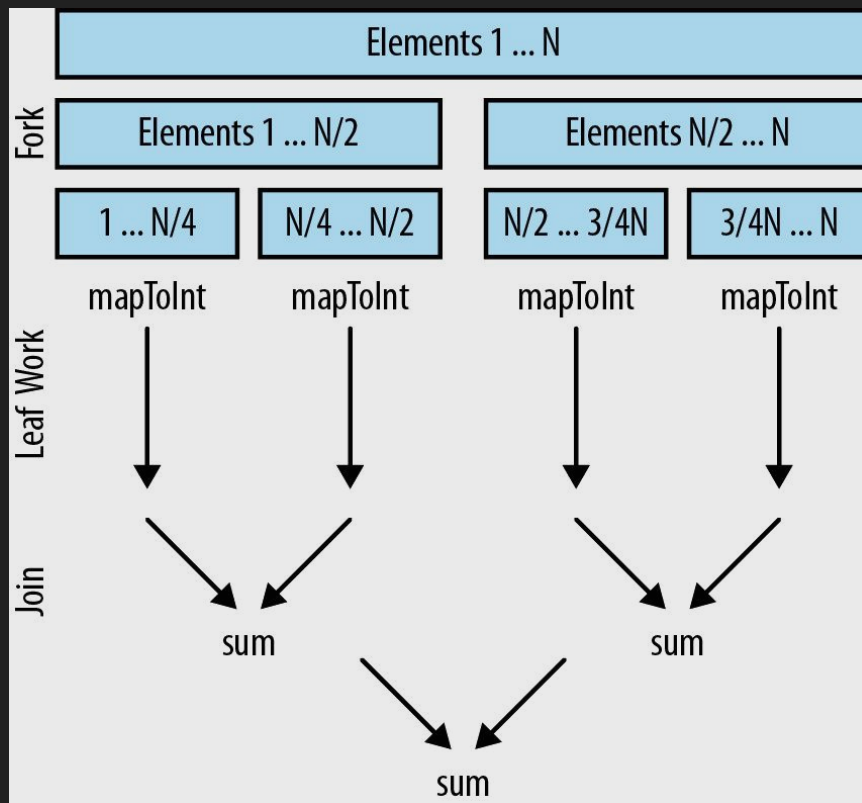
Parallel streams

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

Supongamos que tenemos un procesador con 4 núcleos físicos...

```
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.
- **It's ok:** `HashSet` y `TreeSet` no son fáciles de descomponer, pero se puede.
- **Malas:** Algunas estructuras no se pueden desacoplar (Por ejemplo $O(N)$).
 - Splitear `LinkedList` tiene un costo computacional alto.
 - No se puede conocer la cantidad de elementos de `Stream.iterate` y `BufferedReader.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

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 *Sequential Streams*?

Parallel streams

Factores

- Cantidad de datos
- Estructura de datos de origen
- Boxing
- Numero de nucleos
- 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 una 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 su 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.**

Parallel streams

Examples

Imaginemos que necesitamos paralelizar una tareas CPU-bound, como contar primos. Veamos qué diferencias tenemos entre un proceso secuencia y uno paralelo.

Parallel streams - Examples

```
@Test
public void testCountPrimesWithSequentialStream() {
    // Prepare
    final Integer MAX = 10000000; // En Elixir/Ruby 10_000_000
    Stopwatch stopwatch = Stopwatch.createStarted();

    // Perform
    long count = range(1, MAX).asLongStream().filter(this::isPrime).count();

    // Asserts
    stopwatch.stop();
    assertThat(count, is(equalTo(664579L)));
    assertThat(stopwatch.elapsed(MILLISECONDS), is(greaterThan(15000L)));
    System.out.printf("Sequential stream time: %s\n", stopwatch);
}
```

> Sequential stream time: 29.61 s

Parallel streams - Examples

```
@Test
public void testCountPrimesWithParallelStream() {
    // Prepare
    final Integer MAX = 10000000; // En Elixir/Ruby 10_000_000
    Stopwatch stopwatch = Stopwatch.createStarted();

    // Perform
    long count = range(1, MAX).parallel().filter(this::isPrime).count();

    // Asserts
    stopwatch.stop();
    assertThat(count, is(equalTo(664579L)));
    assertThat(stopwatch.elapsed(MILLISECONDS), is(lessThan(15000L)));
    System.out.printf("Parallel stream time: %s\n", stopwatch);
}
```

> Parallel stream time: 13.22 s

Parallel streams - Examples

Ahora, imaginemos que tenemos que consultar un número de usuarios por sus ids a un recurso REST, ¿Paralelizamos?

```
public List<User> findUserBy(Stream<Long> ids) {  
    return ids.parallel()  
        .map(this::getUser) // Rest endpoint: GET users/ID  
        .collect(toList());  
}
```

Parallel streams - Examples

- Lo primero que se puede pensar es: Si obvio! Si demoro 500ms en traerme cada usuario, con 20 usuario son 10s contra 500ms en paralelo.
- El problema reside en que **parallel streams usa un ÚNICO thread pool común** y si los thread se toman por un periodo largo de tiempo estas bloqueando los thread del pool y nadie más va a poder usarlos en ese periodo de tiempo.
- Hay alternativas para evitarlo:
 - Asegurate que las tareas que ejecuten en un `Parallel stream` terminen en un tiempo razonable.
 - Aumentar el tamaño del pool con la propiedad:
`-Djava.util.concurrent.ForkJoinPool.common.parallelism=50`
 - Otra alternativa es usar una nueva instancia de `ForkJoinPool`.

Parallel streams - Examples

```
@Test
public void testCountPrimesUsingParallelStreamFromNewPool()
    throws ExecutionException, InterruptedException {
    // Prepare
    final Integer MAX = 10000000, POOL_SIZE = 10;
    ForkJoinPool pool = new ForkJoinPool(POOL_SIZE);

    // Perform
    long count = pool.submit(() -> range(1, MAX).parallel().filter(this::isPrime).count()).get();

    // Asserts
    assertThat(pool.getParallelism(), is(equalTo(POOL_SIZE)));
    assertThat(stopwatch.elapsed(MILLISECONDS), is(lessThan(15000L)));
}
```

> Parallel stream from new pool(10) time: 13.42 s

Parallel streams - Examples

Con un `Util` podemos simplificar la invocacion:

```
public class ForkJoinUtils {  
    public static <T> T submit(int poolSize, Callable<T> callable)  
        throws ExecutionException, InterruptedException {  
        return new ForkJoinPool(poolSize).submit(callable).get();  
    }  
}  
  
import static com.nonosoft.ForkJoinUtils.submit;  
long count = submit(PPOOL_SIZE, () -> range(1, MAX).parallel().filter(this::isPrime).count());
```

Seria genial poder hacer esto:

```
ForkJoinPool pool = new ForkJoinPool(PPOOL_SIZE);  
long count = range(1, MAX).parallel(pool).filter(this::isPrime).count());
```

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 tal vez no haga 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 puedes 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 bloquean 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 hay en tiempos de ejecución entre ExecutorService, Fork/join y Parallel streams?

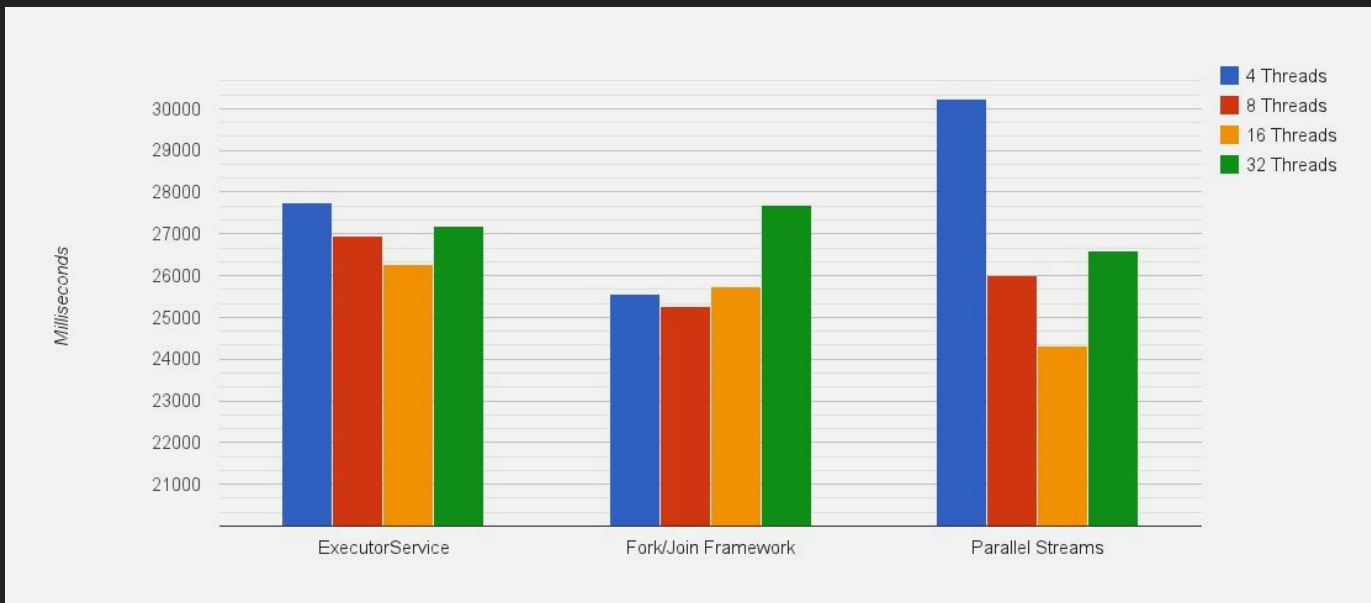
Blog.takipi.com

- Tomaron 2 tareas, una de mucho consumo de CPU y otra de E/S intensiva.
- Todo sobre una máquina 8 núcleos físicos.
- 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](#)

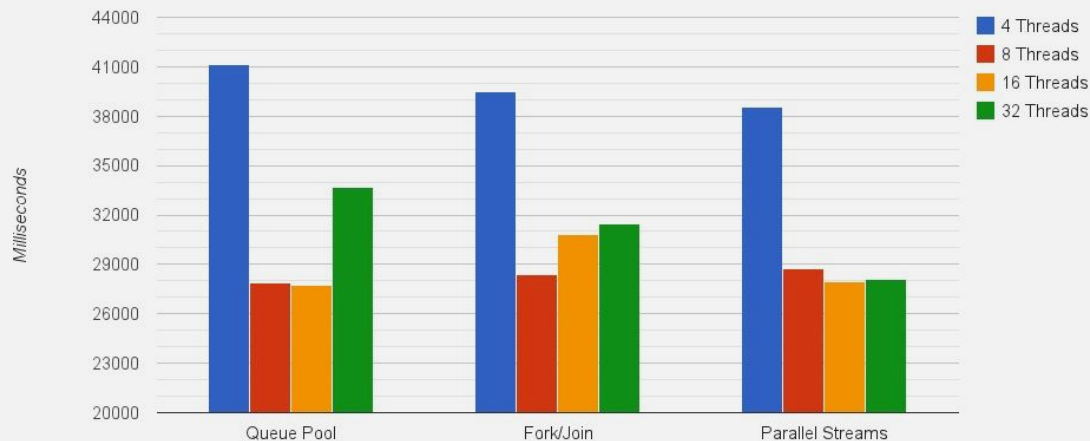
Conclusiones

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

Caso I/O Bound

- Los mejores tiempos se lograron con un `Parallel streams` con un pool size de 16 threads.

Caso CPU Bound

- `ExecutorService` y `Parallel Streams` tiene prácticamente los mismos tiempos cuando el pool size es de 16 threads.
- En todas la variaciones `Parallel Streams` tuvo prácticamente los mejores tiempos.

Conclusiones

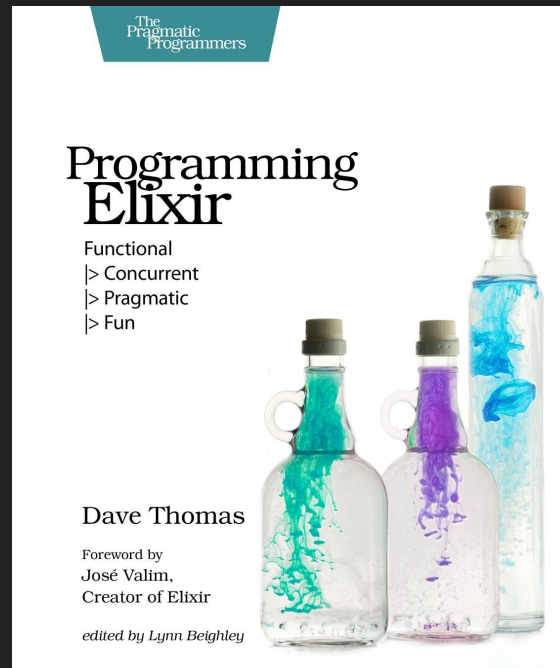
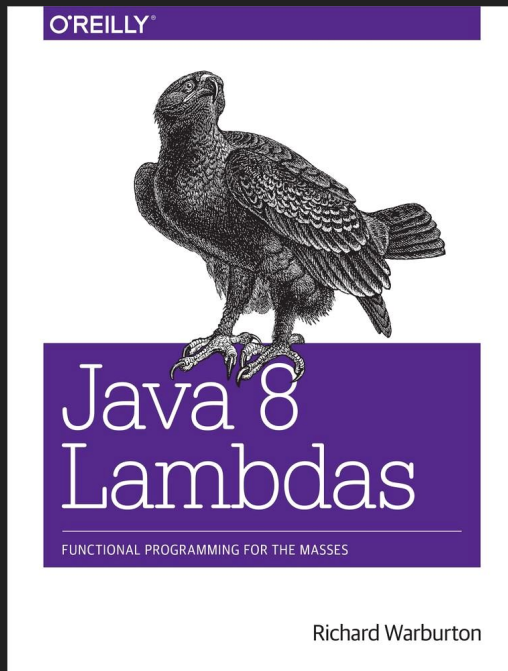
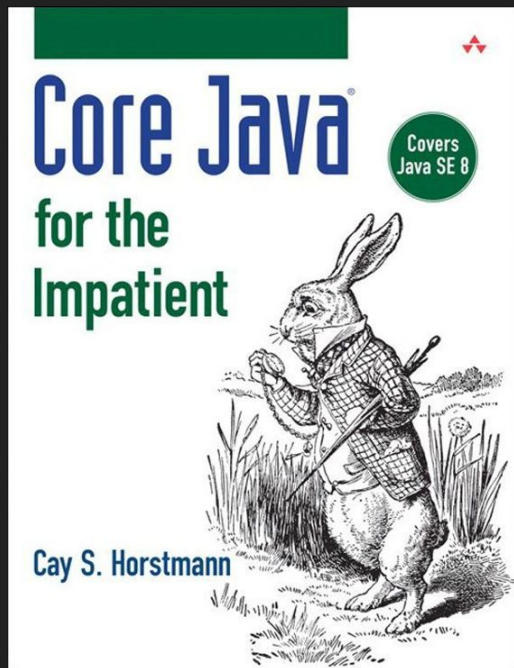
Finalmente, estas son reglas a seguir para lograr procesos más eficientes, pero **siempre la mejor forma de asegurar una mejora es medir.**



Github

github.com/adrianmarino/parallelismAndConcurrency

Referencias



Paralelismo & Concurrency

¿Preguntas?