PCD2018 - 15

DATI THREAD-SAFE

PROBLEMA

Qual'è il problema che incontriamo ponendoci nel quadrante dati mutabili/stato condiviso?

pcd2018.safe.AdderTest

```
public class Adder {

  int target = 0;

  public void add() {
    ...
    t1.start();
    t2.start();
    t3.start();
  }
}
```

I thread sono uguali e così definiti:

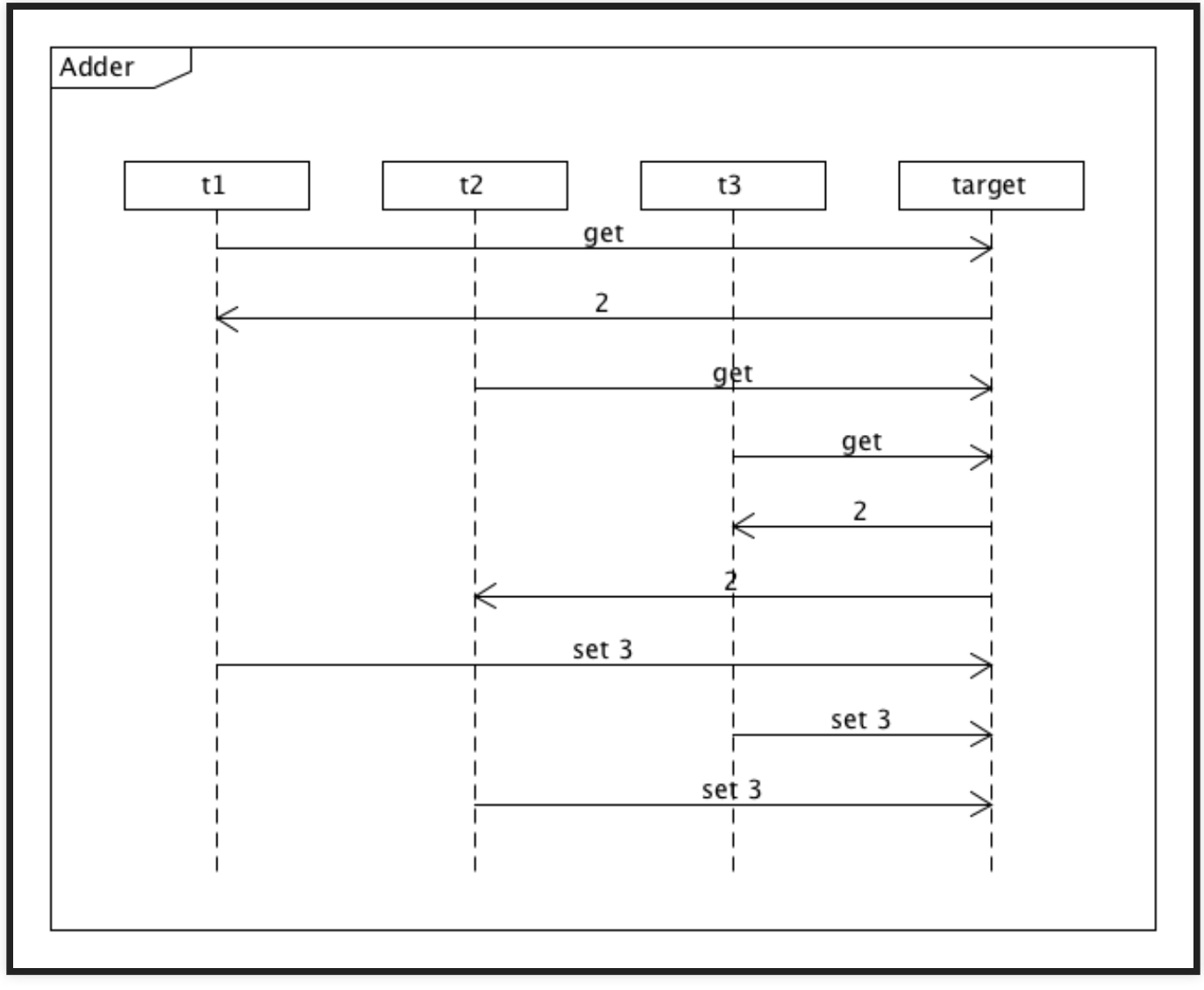
```
var t1 = new Thread(() -> {
  for (int i = 0; i < 100000; i++) {
    target += 1;
  }
});

var t2 = ...
var t3 = ...
```

Test:

```
@Test
void test() throws InterruptedException {
  var adder = new Adder();
  adder.add();
  Thread.sleep(1000);
  assertEquals(300000, adder.target);
}
```

Cosa fallisce?


Adder
t1
t2
t3
target
get
2
get
get
2
2
set 3
set 3
set 3

La concorrenza ci porta al non determinismo.

Il problema di condividere l'accesso a dati non è quindi solo quello del deadlock, ma anche quello della correttezza del risultato.

Una struttura dati non thread-safe non consente a più thread di operare contemporaneamente.

- nel migliore dei casi lancia una `java.util.ConcurrentModificationExcep`
- nel caso intermedio lo stato diventa inconsistente
- nel peggiore dei casi ottengo un'altra eccezione

pcd2018.safe.ListTraverser

```
list.iterator().forEachRemaining(el -> {
  System.out.println(el);
  try {
    Thread.sleep(250);
  } catch (InterruptedException e) {
    e.printStackTrace();
  }
});
```

pcd2018.safe.ListUpdater

```
try {
  Thread.sleep(300);
} catch (InterruptedException e) {
  e.printStackTrace();
}
list.add("d");
```

pcd2018.safe.ListTest

```
List<String> list = new ArrayList<String>();
list.add("a");
list.add("b");
list.add("c");
var t1 = new Thread(new ListTraverser(list));
var t2 = new Thread(new ListUpdater(list));
t1.start();
t2.start();
Thread.sleep(1000);
assertEquals(4, list.size());
```

Approfondimenti:

A Crash Course in Modern Hardware by Cliff Click
https://www.youtube.com/watch?v=OFgxAFdxYAQ

Adventures with concurrent programming in Java
https://www.youtube.com/watch?v=929OrIvbW18

In conclusione:

Se abbiamo la necessità di condividere dati fra più thread, abbiamo bisogno di strutture dati thread-safe.

ATOMIC VARIABLES

Se il nostro caso d'uso riguarda semplicemente l'incremento di un contatore, una possibile soluzione sono le classi del package `java.concurrent.atomic`

tipo	singolo
Integer	`AtomicInteger`
Long	`AtomicLong`
Object	`AtomicReference`

tipo	array
Integer	`AtomicIntegerArray`
Long	`AtomicLongArray`
Object	`AtomicReferenceArray`

Queste classi garantiscono:

- che la modifica del valore che contengono sia "atomica" e thread-safe
- che la modifica (quasi sempre) non blocchi il thread che la sta eseguendo

Quasi sempre: la funzionalità richiede la disponibilità del supporto dell'hardware attraverso istruzioni *CAS*: Compare-and-swap

In mancanza di queste l'implementazione ripiega su metodi più convenzionali (meno efficienti, che bloccano il thread).

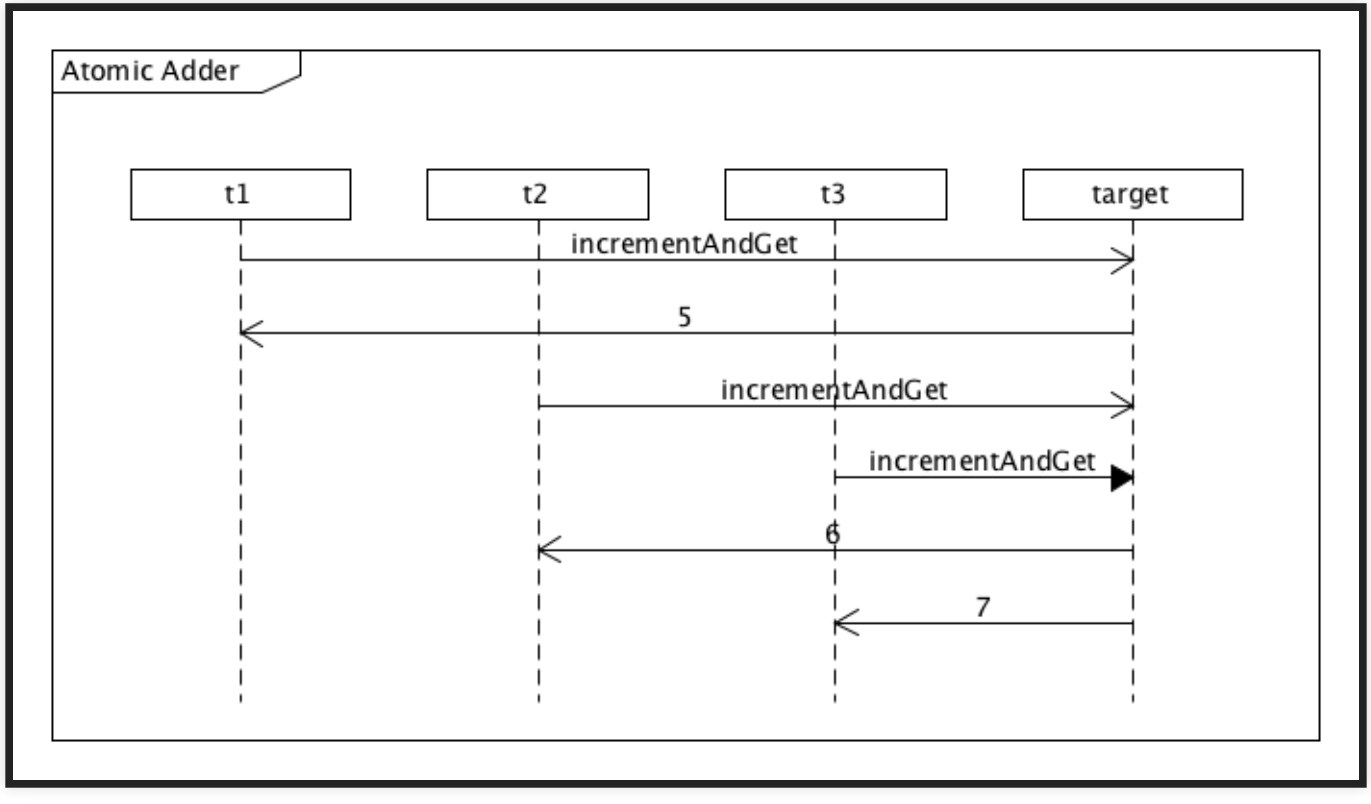
pcd2018.safe.AtomicAdder

```
public class AtomicAdder {

  public AtomicInteger target = new AtomicInteger(0);

  public void add() {
    ...
  }
}
```

```
var t1 = new Thread(() -> {
  for (int i = 0; i < 100000; i++) {
    target.incrementAndGet();
  }
});
```


Atomic Adder
t1
t2
t3
target
incrementAndGet
5
incrementAndGet
incrementAndGet
6
7

```
/**
 * Atomically sets the value to the given updated
 * value if the current value == the expected value.
 *
 * @param expect the expected value
 * @param update the new value
 * @return true if successful. False return indicates
 * that the actual value was not equal to the expected value.
 */
public final boolean compareAndSet(long expect, long update)
```

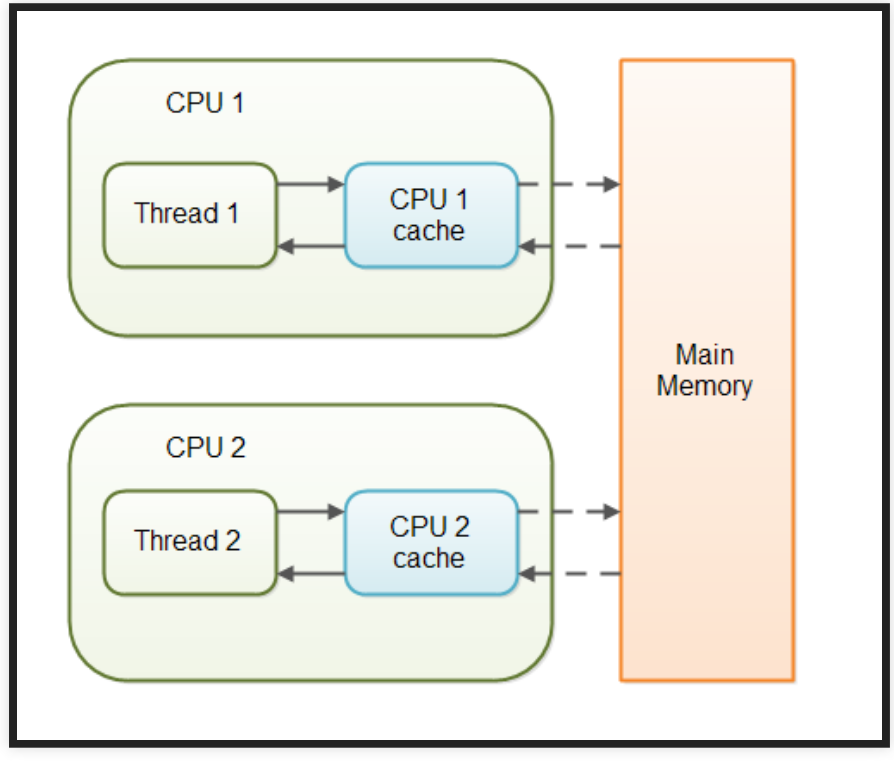
```
/**
 * An AtomicMarkableReference maintains an object
 * reference along with a mark bit, that can be
 * updated atomically.
 *
 * Implementation note : This implementation maintains
 * markable references by creating internal objects
 * representing "boxed" [reference, boolean] pairs.
 */
public class AtomicMarkableReference<V>;
```

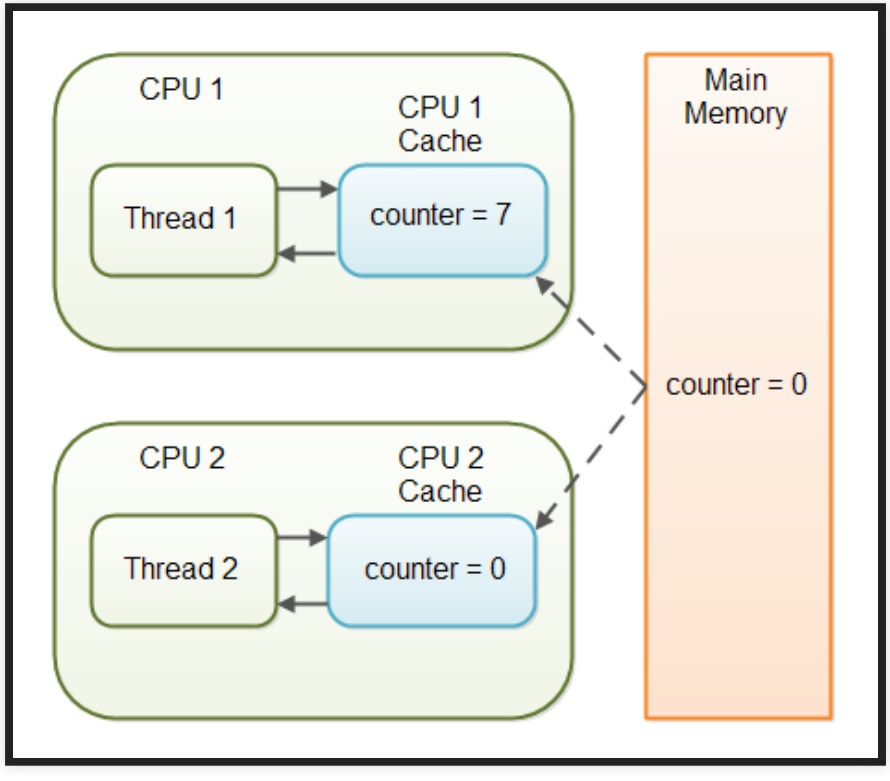
```
/**
 * An AtomicStampedReference maintains an object
 * reference along with an integer "stamp", that
 * can be updated atomically.
 *
 * Implementation note : This implementation maintains
 * stamped references by creating internal objects
 * representing "boxed" [reference, integer] pairs.
 */
public class AtomicStampedReference<V>;
```

VOLATILE

La parola chiave `volatile` indica che una variabile deve sempre essere letta "dalla memoria principale" e non da cache intermedie.

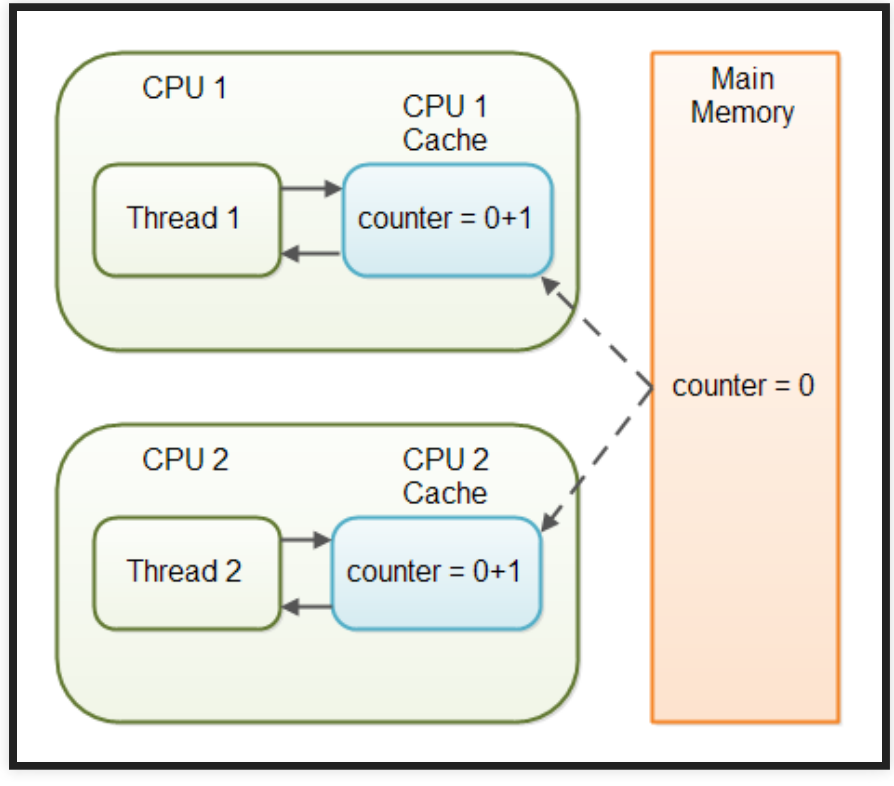
In un'architettura hardware ormai comune, più CPU colloquiano con la stessa memoria principale. Ognuna, ha una cache locale per velocizzare l'esecuzione del codice e l'interazione con i dati.





Questo significa però che thread diversi che sono stati smistati su CPU diverse vedono valori differenti, provenienti dalla cache, della stessa variabile. E' il problema della "visibilità del valore scritto".

Una variabile dichiarata "volatile" viene sempre letta dalla memoria principale in modo da garantire la visibilità dell'ultima scrittura.



Secondo la specifica della JVM, `volatile` stabilisce una relazione di `happens-before` in determinati casi di accesso alla variabile.

Una *relazione di* happens-before è una garanzia forte fornita dal compilatore riguardo l'ordinamento dell'esecuzione delle istruzioni espresse dal codice.

Per approfondire *happens-before*

What Came First: The Ordering of Events in Systems
https://www.infoq.com/presentations/events-riak-go

Cliff Click's Blog
http://cliffc.org/blog/
https://itunes.apple.com/us/podcast/id1286422919

Stabilire una relazione di `happens-before` è, ovviamente, costoso: richiede il supporto dell'hardware e limita la capacità del compilatore di ottimizzare il codice riordinandone le istruzioni.

La Java Language Specification 8 al capitolo 17.4.4 "Happens-before Order" definisce che:

> *A write to a volatile field (§8.3.1.4) happens-before every subsequent read of that field.*

Tuttavia...

pcd2018.safe.VolatileTest

```
class VolatileHolder {
  volatile int counter = 0;
}
```

```java
@Test
public void volatileCounter() {
  VolatileHolder holder = new VolatileHolder();

  ExecutorService executor = Executors.newFixedThreadPool(4);
  IntStream.range(0, 10000).forEach(i ->
    executor.submit(() -> holder.counter++));
  awaitDone(executor);

  assertEquals(10000, holder.counter);
}
```

La garanzia fornita da `volatile` è utile alla correttezza del programma solo se

nessun thread scrive nella variabile volatile un valore dipendente dal valore che ha appena letto dalla stessa variabile

In un certo senso le classi `atomic` sono una generalizzazione e semplificazione di alcuni usi di `volatile`.

Usate con cautela questo costrutto.

Trovate le immagini ed un approfondimento in questo articolo:

http://tutorials.jenkov.com/java-concurrency/volatile.html

CONCURRENT DATA STRUCTURES

Nel package `java.util.concurrent` si trovano le versioni ottimizzate per la concorrenza di molte delle collezioni più comuni.

In generale, sono pensate per essere più efficienti della versione sincronizzata delle loro controparti, per es. `Collections.syncronizedMap(new HashMap())`

CONCURRENTMAP

L'interfaccia `ConcurrentMap` aggiunge a `Map` garanzie di atomicità ed ordinamento delle operazioni.

```
/**
 * A Map providing thread safety and atomicity
 * guarantees.
 */
public interface ConcurrentMap<K,V>
  extends Map<K,V>
```

```
/**
 * If the specified key is not already associated with a
 * value, associate it with the given value.
 * The action is performed atomically.
 */
V putIfAbsent(K key, V Value)
```

```
/**
 * Replaces the entry for a key only if currently mapped
 * to some value.
 * The action is performed atomically.
 */
V replace(K key, V Value)
```

```
/**
 * Replaces the entry for a key only if currently mapped
 * to a given value.
 * The action is performed atomically.
 */
V replace(K key, V oldValue, V newValue)
```

Analogamente `ConcurrentNavigableMap` con `NavigableMap`.

Alcune implementazioni, come `ConcurrentHashMap`, offrono metodi come `reduce`, `search` e `foreach` che possono operare su tutte le chiavi suddividendo autonomamente il lavoro in più thread.

```
/**
 * Returns the result of accumulating the given
 * transformation of all (key, value) pairs using the
 * given reducer to combine values, or null if none.
 *
 * @param the elements needed to switch to parallel
 * @param the transformation for an element
 * @param a commutative associative combining function
 */
public <U> U reduce(long parallelismThreshold,
  BiFunction<? super K,? super V,? extends U> transformer,
  BiFunction<? super U,? super U,? extends U> reducer)
```

pcd2018.safe.ReducePerf

```
Random rnd = new Random();
ConcurrentHashMap<String, Long> map =
  new ConcurrentHashMap<String, Long>();
IntStream.range(0, 10000)
  .forEach(i -> map.put("k" + i, new Long(rnd.nextInt(1000))))
```

```
long start = System.currentTimeMillis();
Long parres = map.reduceEntries(500,
  entry -> entry.getValue(), (a, b) -> a + b);
long partime = System.currentTimeMillis() - start;
```

```
long start = System.currentTimeMillis();
Long parres = map.reduceEntries(10000001,
  entry -> entry.getValue(), (a, b) -> a + b);
long partime = System.currentTimeMillis() - start;
```

Quiz: costruire un esempio in cui è l'implementazione parallela la più efficace.

Suggerimento: il lavoro di riduzione deve essere superiore all'overhead introdotto dal parallelismo...

```
/**
 * Returns a non-null result from applying the given
 * search function on each (key, value), or null if none.
 * Upon success, further element processing is suppressed.
 *
 * @param the elements needed to switch to parallel
 * @param a search function, that returns non-null on
 *    success
 */
public <U> U search(long parallelismThreshold,
  BiFunction<? super K,? super V,? extends U> searchFunction)
```

```
/**
 * Performs the given action for each (key, value).
 *
 * @param the elements needed to switch to parallel
 * @param the action (can have side-effects)
 */
public void forEach(long parallelismThreshold,
  BiConsumer<? super K,? super V> action)
```

Le funzioni usate nei metodi di trasformazione delle mappe devono:

- non dipendere dall'ordinamento
- non dipendere da uno stato condiviso durante il calcolo

Inoltre, per i metodi diversi da `forEach` *non devono avere effetti collaterali*

Per ogni algoritmo vi sono varie versioni:

- con una trasformazione opzionale prima dell'uso del valore
- iterazione solo sulle chiavi o solo sui valori
- con risultati primitivi (`int`, `double` ecc.)

Le operazioni di riduzione, ricerca ed esecuzione di effetti non sono atomiche nel loro complesso, ma ogni coppia chiave-valore non nulla ha una garanzia di `happens-before` con il suo uso nell'iterazione.

BLOCKINGQUEUE

L'interfaccia `BlockingQueue` aggiunge alla classica `Queue` metodi con cui è possibile scegliere la semantica dell'operazione di accodamento e prelievo.

Diventa così possibile richiedere il comportamento desiderato all'interno di una esecuzione concorrente.

Accodamento

metodo	**risultato negativo**
`add(e)`	eccezione
`offer(e)`	`false`
`put(e)`	attesa
`offer(e, time, unit)`	attesa limitata

Prelievo

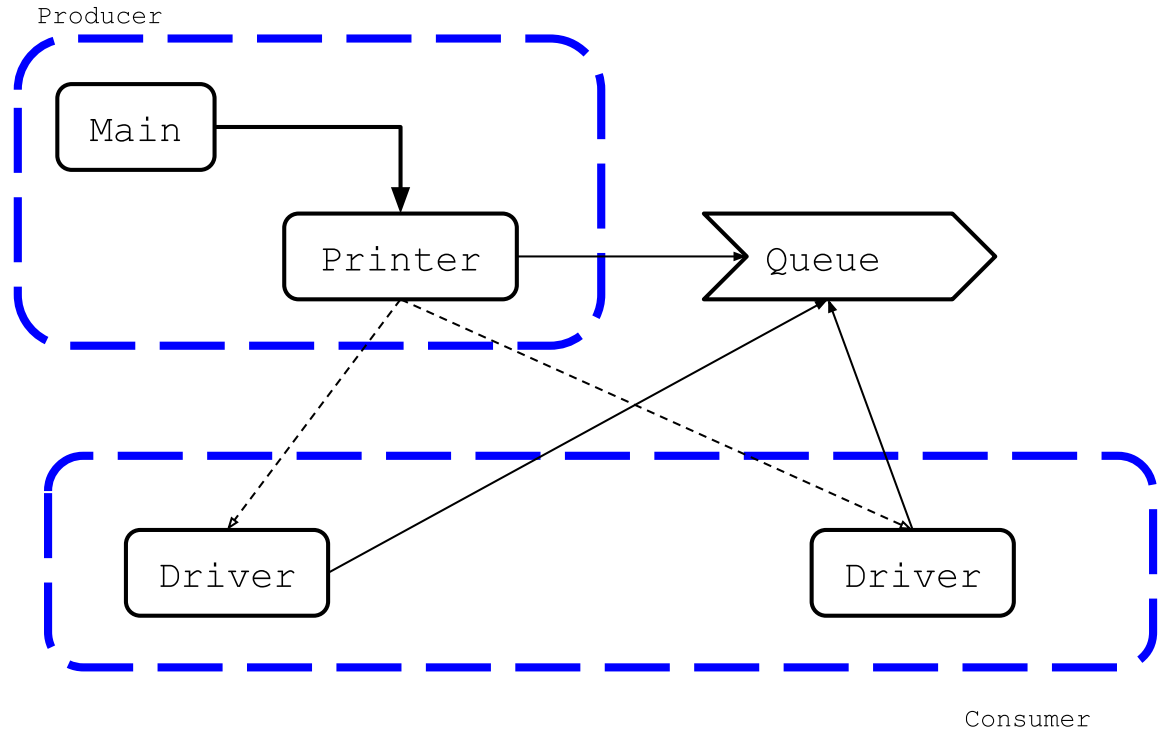
metodo	**risultato negativo**
`remove()`	eccezione
`poll()`	`null`
`take()`	attesa
`poll(time, unit)`	attesa limitata

Lettura

metodo	**risultato negativo**
`element()`	eccezione
`peek()`	`null`

```
/**
 * Removes all available elements from this
 * queue and adds them to the given collection.
 * This operation may be more efficient than
 * repeatedly polling this queue.
 *
 * @param c the collection to transfer elements into
 * @return the number of elements transferred
 */
int drainTo(Collection<? super E> c)
```

Le varie implementazioni di `BlockingQueue`, ciascuna con le sue specifiche caratteristiche, sono la scelta naturale per implementare sistemi Produttore-Consumatore.


Producer
Main
Printer
Queue
Driver
Driver
Consumer

pcd2018.safe.PrinterOperator

```
Printer concurrent = new ConcurrentPrinter(8);
Thread thread[] = new Thread[10];
System.out.println("Preparing...");
  IntStream.range(0, 10).forEach((i) -> thread[i] =
    new Thread(() -> {
      System.out.println("Queueing job " + i);
      concurrent.printJob(new Object());
      System.out.println("Job " + i + " queued");
    }));
System.out.println("Starting.");
for (int i = 0; i < 10; i++) { thread[i].start(); }
```

pcd2018.safe.ConcurrentPrinter

```
ConcurrentPrinter(int printers) {
  // limit printers to effective cores
  size = printers < cores ? printers : cores;
  // size and build the queue
  queue = new LinkedBlockingQueue<PrintJob>(QUEUE_SIZE);
  // start the executor
  executor = Executors.newFixedThreadPool(size);
  // start drivers
  IntStream.range(0, size).forEach((a) ->
    executor.execute(new PrinterDriver(queue)));
}
```

pcd2018.safe.ConcurrentPrinter

```
@Override
public void printJob(Object document) {
  try {
    queue.put(new PrintJob(document));
  } catch (InterruptedException e) {
    e.printStackTrace();
  }
}
```

pcd2018.safe.PrinterDriver

```
public void run() {
  try {
    while (true) {
      PrintJob job = queue.take();
      System.out.printf(...);
      int duration = rnd.nextInt(2500);
      Thread.sleep(duration);
      System.out.printf(...);
    }
  } catch (InterruptedException ex) {
    System.out.println("Printer shutting down.");
  }
}
```

Quiz: usare `TimeUnit.X.wait` al posto di `Thread.sleep()` nella classe precedente genera un'eccezione: quale e perché?

La soluzione richiede il materiale della prossima lezione.

Esercizio 1: implementare un `SerialPrinter` che crea un solo driver ed usa un solo thread.

Esercizio 2: Casualmente, il numero di job che vengono accodati è pari a `ConcurrentPrinter.QUEUE_SIZE`.

E' facile immaginare cosa succede se sono di meno. Cosa avviene se sono di più?

Esercizio 3: La classe `PrinterOperator` accoda i job tramite threads separati.

Cosa succede, nelle condizioni dell'esercizio 2, se invece la classe accodasse direttamente i job?

Esercizio 4: La classe `PrinterOperator` così come è scritta non termina: la JVM non si chiude perché l'`Executor` del `ConcurrentPrinter` non viene chiuso.

Come si può fare per permettergli di chiudere correttamente, *dopo* aver eseguito tutti i job?

Nota: questo è un esercizio di design; non è detto che sia risolvibile nel design attuale.

Altre varianti:

- `TransferQueue`: interfaccia per una coda in cui i produttori aspettano i consumatori
- `BlockingDeque`: interfaccia che permette di prendere un elemento dalla coda o dalla testa

- `ArrayBlockingQueue`: implementazione basata su array, con possibilità di *fairness*
- `LinkedBlockingDeque`, `LinkedBlockingQueue`, `LinkedTransferQueue`: implementazioni basate su liste collegate

- `PriorityBlockingQueue`: coda ordinata per priorità
- `DelayQueue`: un elemento non può essere preso prima di un ritardo impostato
- `SynchronousQueue`: ogni produttore deve attendere un consumatore (capacità nulla)

Altre strutture dati interessanti:

Disruptor

http://lmax-exchange.github.io/disruptor/

Altri Esempi:

http://winterbe.com/posts/2015/05/22/java8-concurrency-tutorial-atomic-concurrent-map-examples/

THREAD LOCAL VARIABLES

Finora abbiamo visto come condividere la stessa variabile fra più thread.

Un'approccio alternativo è invece garantire che la stessa variabile abbia un valore indipendente e separato per ciascun Thread.

```
/**
 * These variables differ from their normal counterparts
 * in that each thread that accesses one (via its get
 * or set method) has its own, independently initialized
 * copy of the variable.
 */
public class ThreadLocal<T>
```

Una variabile `ThreadLocal` esiste in una copia differente ed indipendente per ciascun Thread che attraversa la sua dichiarazione.

```
/**
 * Creates a thread local variable. The initial value of
 * the variable is determined by invoking the get method
 * on the Supplier.
 *
 */
static <S> ThreadLocal<S>
  withInitial(Supplier<? extends S> supplier)
```

```
/**
 * Returns the value in the current thread's copy of this
 * thread-local variable. If the variable has no value
 * for the current thread, it is first initialized to the
 * value returned by an invocation of the initialValue()
 * method.
 */
public T get()
```

```
/**
 * Removes the current thread's value for this thread-local
 * variable.
 */
public void remove()
```

```java
/**
 * Sets the current thread's copy of this thread-local
 * variable to the specified value.
 */
public void set(T value)
```

```
/**
 * Returns the current thread's "initial value" for
 * this thread-local variable.
 *
 */
protected T initialValue()
```

pcd2018.safe.LocalVar

```
class LocalVar {
  private static final var nextId = new AtomicInteger(0);

  ThreadLocal<Integer> counter;

  LocalVar() {
    counter = ThreadLocal.withInitial(() ->
      nextId.incrementAndGet());
  }

  Integer get() { return counter.get(); }
}
```

pcd2018.safe.LocalReader

```
class LocalReader implements Runnable {
  private final LocalVar var;
  private final int item;

  @Override
  public void run() {
    System.out.println(Thread.currentThread().getName() +
      ", item " + item + ": read " + var.get());
  }
}
```

pcd2018.safe.LocalMain

```
ExecutorService executor = Executors.newFixedThreadPool(4);
LocalVar var = new LocalVar();
IntStream.range(0, 20).forEach((a) ->
  executor.execute(new LocalReader(var, a)));
executor.shutdown();
```

Le variabili `ThreadLocal` hanno la cattiva abitudine di assomigliare molto a delle variabili globali.

Usare con cautela.

PUBBLICITÀ

Amazon Corretto

https://www.infoq.com/news/2018/11/amazon-corretto-java