

Concurrence

Opérations atomiques

Rémi Forax

Opérations atomiques

La plupart des processeurs possède des instructions assembleurs dites atomiques

- Effectuer un calcul en 1 seule opération

Par exemple, pour intel,

`ladd rax 1` est équivalent à `i++`

Il est possible d'accéder en Java à ces opérations grâce au package `java.util.concurrent.atomic`

Atomique implique volatile

En Java, les opérations atomiques sont vues comme des méthodes que l'on peut exécuter sur un champ volatile

Deux façons de faire:

- la classe `java.util.concurrent.atomic.AtomicInteger`
 - Encapsule un champ volatile (ici integer) et fournit des méthodes atomiques
- la classe `java.lang.invoke.VarHandle`
 - Référence un champ volatile d'une classe externe et fournit des méthodes atomiques plus bas niveau

Exemple, un compteur thread-safe

Version avec synchronized:

```
public class ThreadSafeCounter {  
    private final Object lock = new Object();  
    private int counter;  
  
    public int nextValue() {  
        synchronized (lock) {  
            return counter++;  
        }  
    }  
}
```

le code est thread-safe mais pas très rapide

Avec un AtomicInteger

Un AtomicInteger contient un champ volatile:

```
public class ThreadSafeCounter {  
    private final AtomicInteger counter = new AtomicInteger();  
    public int nextValue() {  
        return counter.getAndIncrement();  
    }  
}
```

getAndIncrement() est vu

- comme un appel de méthode par le compilateur et
- comme une instruction assembleur par le JIT

Atomic* est une famille de classes

Il existe une classe spécifique pour chaque type

- boolean -> AtomicBoolean
- int -> AtomicInteger
- long -> AtomicLong
- ...
- reference -> AtomicReference<T>

et aussi, pour les tableaux

- AtomicIntegerArray, AtomicLongArray, AtomicReferenceArray<T>

Compare and Set

En fait, toutes les opérations atomiques ne sont pas toutes disponibles sur tous les processeurs mais il existe une primitive de base

Le compareAndSet ou CAS

Signature en C:

CAS(&field, expectedValue, newValue) -> boolean

Si le champ est == à expectedValue, alors la valeur du champ devient newValue et on renvoie vrai sinon on renvoie faux

Avec un AtomicInteger

AtomicInteger a une méthode compareAndSet:

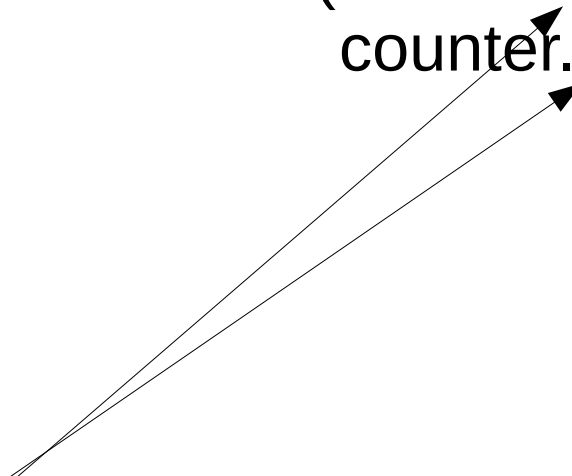
```
public class ThreadSafeCounter {  
    private final AtomicInteger counter = new AtomicInteger();  
    public int nextValue() {  
        for(;;) {  
            int current = counter.get(); // volatile read  
            int newValue = current + 1;  
            if (counter.compareAndSet(current, newValue)) { // volatile write  
                return current;  
            } // otherwise retry  
        }  
    }  
}
```

Si le CAS marche, c'est équivalent à une écriture volatile

Attention, exemple de CAS mal écrit !

```
public class BadCounter {  
    private final AtomicInteger counter = new AtomicInteger();  
    public int nextValue() {  
        for(;;) {  
            if (counter.compareAndSet(counter.get(),  
                                     counter.get() + 1) { // ahhhh  
                return current;  
            }  
        }  
    }  
}
```

On fait 2 appels à get() sans passer par une variable temporaire



CAS + boucle avec une lambda

La méthode `getAndUpdate(UnaryOperator)` permet de faire un CAS + boucle en indiquant uniquement la fonction qui permet d'aller à la valeur suivante

```
public class ThreadSafeCounter {  
    private final AtomicInteger counter = new AtomicInteger();  
    public int nextValue() {  
        return counter.getAndUpdate(x -> x + 1);  
    }  
}
```

Problème de AtomicInteger

AtomicInteger est simple à utiliser mais

- Il est gourmand en mémoire
 - il faut créer un autre objet pour chaque champ volatile
- Il nécessite une indirection en mémoire pour chercher la valeur (donc un potentiel *cache-miss*)

La classe `java.lang.invoke.VarHandle` permet de palier ces problèmes avec une API moins simple

VarHandle

Un VarHandle correspond à 'un pointeur' (un handle) sur un champ volatile ou une case d'un tableau (aussi volatile)

```
Lookup lookup = MethodHandles.lookup();  
VarHandle handle = lookup.findVarHandle(  
    ThreadSafeCounter.class, // classe contenant le champ  
    "counter",                // nom du champ  
    int.class);               // type du champ
```

Lookup.findVarHandle() permet de créer un VarHandle à partir d'un objet Lookup correspondant à un contexte de sécurité.

MethodHandles.lookup() permet de demander le contexte de sécurité à l'endroit de l'appel

Avec un VarHandle

Un compteur thread-safe avec un VarHandle:

```
public class ThreadSafeCounter {
    private volatile int counter;

    private final static VarHandle COUNTER_REF;
    static {
        Lookup lookup = MethodHandles.lookup();
        try {
            COUNTER_REF = lookup.findVarHandle(ThreadSafeCounter.class,
                                                "counter", int.class);
        } catch (NoSuchFieldException | IllegalAccessException e) {
            throw new AssertionError(e);
        }
    }

    public int nextValue() {
        ...
    }
}
```

Avec un VarHandle (2)

VarHandle possède une méthode compareAndSet:

```
public class ThreadSafeCounter {  
    private volatile int counter;  
  
    private final static VarHandle COUNTER_REF;  
    static {  
        COUNTER_REF = ...  
    }  
  
    public int nextValue() {  
        for(;;) {  
            int current = this.counter; // volatile read  
            if (COUNTER_REF.compareAndSet(this, current, current + 1)) {  
                return current;  
            } // otherwise retry  
        }  
    }  
}
```

Avec un getAndAdd

VarHandle.getAndAdd() prend en paramètre l'incrément, c'est l'équivalent de AtomicInteger.getAndIncrement().

```
public class ThreadSafeCounter {  
    private volatile int counter;  
    private final static VarHandle COUNTER_REF;  
    static {  
        COUNTER_REF = ...  
    }  
    public int nextValue() {  
        return (int) COUNTER_REF.getAndAdd(this, 1);  
    }  
}
```

Signature Polymorphique

Les méthodes de VarHandles sont 'magiques' dans le sens où elles acceptent différents paramètres sans faire de surcharge

Par contre, il faut faire un (faux) cast pour indiquer le type de retour

```
public class ThreadSafeCounter {  
    private volatile long counter;  
  
    private final static VarHandle COUNTER_REF = ...  
        .findVarHandle(ThreadSafeCounter.class, "counter", long.class);  
  
    public long nextValue() {  
        return (long) COUNTER_REF.getAndAdd(this, 1L);  
    }  
}
```

VarHandle n'utilise pas les types paramétrés

withInvokeExactBehavior()

Par défaut, l'API des VarHandle permet aussi de faire du boxing (int → Integer, etc) mais ce n'est pas performant

- On va demander un VarHandle qui ne fait pas le boxing

Exemple

```
public class ThreadSafeCounter {  
    private volatile long counter;  
    private final static VarHandle COUNTER_REF = ...  
        .findVarHandle(ThreadSafeCounter.class, "counter", long.class)  
        .withInvokeExactBehavior();  
    ...  
}
```

Algorithmique concurrente

Algorithmique concurrente

En fait, quasiment toutes les implantations des structures de données présentes dans le package `java.util.concurrent` n'utilisent pas de bloc `synchronized`

On utilise

- soit des `ReentrantLock`
- soit pour les implantations “lock-free”
 - Volatile et les opérations atomiques (souvent CAS)

Exemple

On veut une liste chaînée concurrente lock-free avec insertion en tête

```
public final class NotConcLinkedList {  
    private record Entry(Object value, Entry entry) {}  
    private Entry head;  
    public void add(Object value) {  
        this.head = new Entry(value, this.head);  
    }  
    ...  
}
```

Liste chaînée Lock-free

```
public class LockFreeLinkedList {  
    private record Entry(Object value, Entry next) {}  
  
    private final AtomicReference<Entry> reference =  
        new AtomicReference<>();  
  
    public void add(Object value) {  
        for(;;) {  
            Entry head = this.reference.get();  
            Entry entry = new Entry(value, head);  
            if (this.reference.compareAndSet(head, entry)) {  
                return;  
            }  
        }  
    }  
}
```

Liste chaînée Lock-free (2)

```
public class LockFreeLinkedList {  
    ...  
    private final static VarHandle HEADER_REF;  
    static {  
        ...  
        HEADER_REF = lookup.findVarHandle(LockFreeLinkedList.class,  
                                            "head", Entry.class)  
                                .withInvokeExactBehavior();  
        ...  
    }  
    private volatile Entry head;  
    public void add(Object value) {  
        for(;;) {  
            Entry head = this.head; // volatile read  
            Entry entry = new Entry(value, head);  
            if (HEADER_REF.compareAndSet(this, head, entry)) { // volatile write  
                return;  
            }  
        }  
    }  
}
```

Legacy API

Atomic*FieldUpdater

L'API des VarHandle a été introduit dans la version 9, précédemment, on utilisait des `j.u.c.AtomicReferenceFieldUpdater`

- Cette API est moins efficace car elle fait des casts dynamiques
- `ARFU.weakCompareAndSet` a un gros bug, l'écriture n'est pas volatile (aaaah)

Sémantiques exotiques

Autres sémantiques

VarHandle possède d'autres méthodes permettant d'avoir une gestion fine des effets

Fences:

- loadLoadFence, storeStoreFence, acquireFence, releaseFence, fullFence

Autres instructions supportées par les CPUs

- getAndAdd, getAndBitwiseOr, getAndBitwiseAnd

CAS 'weak' sémantique

Un CAS 'weak' peut renvoyer false pour 2 raisons

- La valeur en RAM a été changée par une autre thread
- Le cœur courant ne possède pas la cache line contenant la valeur (car un autre la possède)

Le weak CAS est plus rapide sauf en cas de forte contention

Avec un 'weak' CAS

VarHandle possède une méthode weakCompareAndSet:

```
public class ThreadSafeCounter {  
    private volatile int counter;  
  
    private final static VarHandle COUNTER_REF;  
    static {  
        COUNTER_REF = ...  
    }  
  
    public int nextValue() {  
        for(;;) {  
            int current = this.counter; // volatile read  
            if (COUNTER_REF.weakCompareAndSet(this, current, current + 1)) {  
                return current;  
            } // otherwise retry  
        }  
    }  
}
```