7 Atomic-Variablen

Der lesende oder schreibende Zugriff auf Variablen eines primitiven Datentyps ist, außer bei `long` und `double`, in Java atomar, d. h. nicht unterbrechbar. Die Zugriffe auf Referenzvariablen sind dagegen immer atomar, unabhängig davon, ob es sich um eine 32- oder 64-bit-JVM handelt. Werden die Variablen mit `volatile` gekennzeichnet, ist der Zugriff garantiert immer atomar, unabhängig vom Datentyp[1].

Oft besteht aber eine Operation auf einer Variablen aus mehreren Schritten, obwohl im Code dafür nur eine Anweisung angegeben ist. Die Anweisung `counter++` besteht z. B. aus den Befehlen:

1. Laden des Inhalts von `counter` in ein Register (Lesen).
2. Registerinhalt wird inkrementiert.
3. Das Ergebnis wird in `counter` geschrieben (Schreiben).

Obwohl jeder einzelne Schritt atomar ist, kann der aufrufende Thread dazwischen unterbrochen werden. Ist ein anderer an der Reihe und führt er ebenfalls diese Anweisung aus, kann es zu einem inkonsistenten Zustand kommen.

Um einen Thread-sicheren Zähler zu bauen, kann man auf bekannte Mittel zurückgreifen, etwa wie:

```
class SynchronizedCounter
{
  private int counter = 0;

  public synchronized void increment() { counter++;  }
  public synchronized void decrement() { counter--; }
  public synchronized int get() {  return counter; }
}
```

Codebeispiel 7.1: Thread-sicherer Zähler

[1] Wenn ein Thread versucht, eine atomare Variable zu lesen, während der andere sie gerade ändert, wird der lesende Thread entweder den alten oder den neuen Wert lesen, nicht aber eine inkonsistente Mischung.

Die Methode `get` liest zwar nur den Wert der `int`-Variablen und ist somit atomar, sie muss aber trotzdem mit `synchronized` gekennzeichnet sein, damit parallel zugreifende Threads immer den aktuellen Wert von `counter` lesen[2].

Wenn einfache Daten mithilfe von Locks geschützt werden, bedeutet dies immer einen Performance-Verlust, weil das Betriebssystem angesprochen und eventuell ein Kontextwechsel stattfinden wird.

Moderne Prozessoren bieten zur Vermeidung dieses Overheads einen ununterbrechbaren *Compare-and-Set*-Befehl an[3], mit dessen Hilfe man in manchen Situationen auf eine Synchronisation und somit auf Locks verzichten kann.

7.1 Compare-and-Set-Operation

Mithilfe der Compare-and-Set-Operation kann eine Variable (Speicherzelle) atomar gelesen und verändert werden. Die Operation benötigt hierzu drei Angaben:

1. Eine Speicherstelle,
2. den erwarteten, alten Wert und
3. einen neuen Wert.

Wenn der Inhalt der Speicherzelle mit dem erwarteten, alten Wert übereinstimmt, wird der neue an die Speicherstelle geschrieben. Stimmt der erwartete, alte Wert nicht überein, da ihn z. B. ein anderer Thread zwischenzeitlich geändert hat, findet keine Modifikation statt. Eine boolesche Rückgabe signalisiert, ob eine Änderung stattgefunden hat.

Mit einer `compareAndSet`-Operation würde eine atomare Erhöhung des Zählers (`counter++`) etwa so aussehen:

```
int tmp = counter;
while (! compareAndSet( counter, tmp, tmp+1))
{
  tmp = counter;   // counter wurde verändert,
                   // neuen Wert einlesen
}
```

[2] Eine Kennzeichnung des `counter`-Attributs mit `volatile` hätte denselben Effekt. In diesem Fall kann bei der `get`-Methode auf das `synchronized` verzichtet werden. Die Methoden `increment` und `decrement` müssen aber nach wie vor mit `synchronized` deklariert werden.

[3] Eine analoge Möglichkeit ist der sogenannte *ExchangeWord*-Befehl, der garantiert, dass das Lesen und das unmittelbare Schreiben einer Speicherzelle atomar sind.

Zuerst wird der Wert von `counter` gelesen. Der Aufruf von `compareAndSet` prüft dann, ob der gerade gelesene Wert noch aktuell ist. Zwischen dem Lesen und dem Aufruf von `compareAndSet` könnte der Wert ja durch einen anderen Thread geändert worden sein. Falls `tmp` und `counter` denselben Wert besitzen, wird der Inhalt der Speicherstelle (`counter`) erhöht – falls nicht, wird im Schleifenblock der aktuelle Wert von `counter` gelesen. Danach erfolgt erneut die Überprüfung.

7.2 Umgang mit Atomic-Variablen

Java bietet im Paket `java.util.concurrent.atomic` Kapselungen für verschiedene Datentypen an, die intern auf dem `compareAndSet`-Mechanismus aufbauen. Die gängigen Klassen sind `AtomicBoolean`, `AtomicInteger`, `AtomicLong` und `AtomicReference`.

7.2.1 Atomic-Skalare

Die Klassen `AtomicInteger` und `AtomicLong` besitzen ähnliche Methoden, sodass wir hier nur auf `AtomicInteger` eingehen. Es stehen zwei Konstruktoren zur Verfügung: `AtomicInteger()`, der ein neues Objekt mit 0, und `AtomicInteger(int initialValue)`, der ein Objekt mit dem übergebenen Startwert erzeugt. Einige wichtige atomare Methoden der Klasse `AtomicInteger` sind in Tabelle 7-1 aufgelistet. Bei den meisten Methoden gibt es zwei Varianten, die sich nur dadurch unterscheiden, dass der alte (`getAndXXX`) oder der neue Wert (`XXXAndGet`) zurückgeliefert wird.

Die folgende Klasse realisiert einen Thread-sicheren Zähler, ohne dass dabei die schwergewichtige Synchronisation verwendet wird:

```
final class AtomicCounter
{
    private final AtomicInteger counter = new AtomicInteger(0);
    public void increment() { counter.incrementAndGet(); }
    public void decrement() { counter.decrementAndGet(); }
    public int get() { return counter.get(); }
}
```

Codebeispiel 7.2: Thread-sicherer Zähler mit `AtomicInteger`

Atomare Operationen finden häufig Anwendungen in nicht blockierenden Algorithmen und Datenstrukturen sowie bei Implementierungen der verschiedenen Synchronisationsmechanismen. In anderen Programmiersprachen wie C++ werden atomare Operationen benötigt, um das Konzept von Referenzzählern zu realisieren, was in Java nicht notwendig ist.

API	Erläuterung
`boolean compareAndSet(int expect, int update)`	Der *Compare-and-Set*-Befehl, angewandt auf den Wert des Objekts.
`int addAndGet(int delta)`	Der Wert wird atomar um `delta` erhöht. Der neue Wert wird zurückgegeben.
`int decrementAndGet()`	Der Inhalt wird atomar dekrementiert und der neue Wert wird zurückgegeben.
`int incrementAndGet()`	Der Inhalt wird atomar inkrementiert und der neue Wert wird zurückgegeben.
`int set(int newValue)`	Der Wert wird durch `newValue` ersetzt und der neue Wert wird zurückgegeben.
`int get()`	Liefert den aktuellen Wert.

Tabelle 7-1: Zugriffsmethoden auf `AtomicInteger`

Hinweis

1. Das Sperren kann im Prinzip nicht vermieden werden. Mit Atomic-Variablen wird diese Aufgabe auf die Hardware delegiert. Wenn zwei Threads auf verschiedenen Rechenkernen gleichzeitig auf eine Atomic-Variable zugreifen, wird ein Thread den Bus blockieren und der andere muss so lange auf die Freigabe warten. Das bedeutet ein mögliches (sehr kurzes) Blockieren.
2. Zugriffe auf Atomic-Variablen haben ähnliche Wirkung auf den Speicher wie die `volatile`-Variablen. Während `volatile`-Variablen je nach Zugriff einen sogenannten *acquire fence* oder *release fence* verwenden, kann ein atomarer Zugriff beide, sowohl einen *acquire* als auch einen *release fence*, benötigen. Die wesentlichen Methoden und deren Speichereffekte sind im Vergleich zu einem Zugriff auf eine `volatile`-Variable [29]:
   - `get` entspricht dem Lesen.
   - `set` entspricht dem Schreiben.
   - `compareAndSet` entspricht dem Lesen und Schreiben.

Wenn komplexere Änderung durchgeführt werden müssen, kann die Methode `compareAndSet` sehr nützlich sein. Betrachten wir das folgende Beispiel: Wir wollen in einer `AtomicLong`-Variablen einen Maximalwert speichern, der von verschiedenen Threads durch Aufruf einer `update`-Methode geändert werden kann. Die folgende, naiv implementierte `update`-Methode ist aber falsch:

```
private static final AtomicLong maxVal = new AtomicLong();

public static void update(long newVal)
{
  maxVal.set( Math.max(maxVal.get(), newVal) ); // FALSCH !!!
}
```

Der Fehler besteht darin, dass `update` aus mehreren Schritten besteht. Korrekterweise muss die Methode wie folgt implementiert werden:

```
public static void update(long newVal)
{
  long alt, neu;
  do{
     alt = maxVal.get(); // Lesen
     neu = Math.max(alt, newVal);
  } while( maxVal.compareAndSet(alt, neu) == false );
}
```

Um solche typischerweise notwendigen Schleifen zu vermeiden, stehen ab Java 8 weitere Methoden in den Atomic-Klassen zur Verfügung. Dies sind zum einen die `accumulateAndGet`- und `getAndAccumulate`-Methoden, die einen `LongBinaryOperator`-Ausdruck (*Functional Interface*) und einen Operanden als Parameter erhalten. Mit ihrer Hilfe kann die `update`-Methode elegant implementiert werden:

```
public static void update(long newVal)
{
  maxVal.accumulateAndGet( newVal, x-> Math.max(x, newVal) );
}
```

bzw. durch die Verwendung einer Methodenreferenz:

```
public static void update(long newVal)
{
  maxVal.accumulateAndGet(newVal, Math::max);
}
```

Zum anderen sind es die `updateAndGet`- und `getAndUpdate`-Methoden, die einen `LongUnaryOperator` erhalten. Hierdurch kann z. B. eine atomare Multiplikation realisiert werden:

```
private static final AtomicLong val = new AtomicLong(5);

public static void mult(long x)
{
  val.updateAndGet( a -> a*x );
}
```

7.2.2 Atomic-Referenzen

Wie für primitive Datentypen gibt es auch Atomic-Kapselungen für Referenzen. Für atomare Änderungen von Referenzen vom Typ `V` stehen neben der Klasse `AtomicReference` noch die beiden Hilfsklassen `AtomicMarkableReference` und `AtomicStampedReference` zur Verfügung.

Während ein `AtomicReference`-Objekt lediglich nur die Referenz speichert, besitzt ein `AtomicMarkableReference` einen zusätzlichen booleschen Wert, der typischerweise die Gültigkeit des referenzierten Objekts signalisiert (vgl. Abb. 7-1). Die `AtomicStampedReference`-Klasse verwendet im Vergleich zu `AtomicMarkableReference` einen `Integer`-Wert, der eine Versionsnummer für das referenzierte Objekt darstellen kann.

Schauen wir stellvertretend die Klasse `AtomicMarkableReference` etwas näher an. Der Konstruktor `AtomicMarkableReference(V initialRef, boolean initialMark)` hat neben der zu kapselnden Referenz noch einen booleschen Parameter. Wichtige Methoden sind:

- `boolean attemptMark(V expectedRef, boolean newMark)`: Atomares Setzen der Markierung auf `newMark`, wenn die aktuelle Referenz `expectedRef` ist. Im Erfolgsfall wird `true` zurückgeliefert, ansonsten `false`.
- `boolean compareAndSet(V expectedRef, V newRef, boolean expectedMark, boolean newMark)`: Atomare Änderung der Referenz und der Markierung, nur wenn die aktuelle Referenz `expectedRef` und die aktuelle Markierung `expectedMark` ist. Im Fall einer Änderung wird `true` zurückgeliefert.
- `V get(boolean[] markHolder)`: Diese Methode gibt die aktuelle Referenz und die aktuelle Markierung zurück. Da nur ein Return-Wert erlaubt ist, wird der boolesche Wert in das erste Element des übergebenen Arrays geschrieben. Das Array hat üblicherweise die Länge 1.
- `V getReference()`: Diese Methode gibt die aktuelle Referenz zurück.

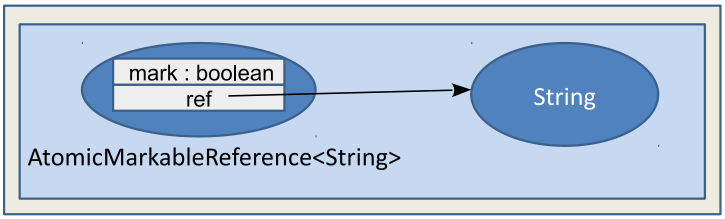

Abbildung 7-1: Schematischer Aufbau einer `AtomicMarkableReference<String>`

Die Verwendung von `AtomicMarkableReference` findet man z. B. bei Implementierungen von Thread-sicheren verketteten Listen (vgl. Kapitel 17). In diesem Abschnitt begnügen wir uns mit einem einfachen Beispiel, in dem zwei Threads über eine gemeinsam benutzte Datenstruktur Objekte austauschen. Thread A produziert dabei die Objekte und schreibt sie in eine Datenstruktur, während sie Thread B ausliest. Dabei muss A eventuell warten, wenn das Objekt noch nicht abgeholt wurde. Es handelt sich hierbei um das lockfreie Erzeuger-Verbraucher-Muster mit einer Queue der Länge 1.

Zur Realisierung der gemeinsam benutzten Datenstruktur wird eine `AtomicMarkableReference` benutzt. Codebeispiel 7.3 zeigt eine mögliche Implementierung.

```
class AtomicOneValueBuffer<V>
{
  private final AtomicMarkableReference<V> buffer                    ❶
          = new AtomicMarkableReference<V>(null, false);

 public void put(V value)                                            ❷
 {
   do
   {
     // Kein Inhalt
   }while( !buffer.compareAndSet(null, value, false, true) );
 }

 public V take()                                                     ❸
 {
   V value = null;

   do
   {
     value = buffer.getReference();
   }while( !buffer.compareAndSet(value, null ,true, false) );
   return value;
 }
}
```

Codebeispiel 7.3: Eine lockfreie Datenstruktur für den Austausch eines Objekts

Zur Verwaltung wird intern ein `AtomicMarkableReference`-Attribut benutzt (❶). Der boolesche Parameter dient zur Kennzeichnung, ob das Objekt bereits ausgelesen wurde (`false`) oder nicht (`true`).

Über die Methode `put` kann ein neues Objekt hinzugefügt werden (❷). Hierbei wird in der `do-while`-Schleife mithilfe von `compareAndSet` geprüft, ob die aktuelle Referenz `null` und die Markierung `false` ist. Treffen beide Bedingungen zu, werden gleichzeitig die beiden Daten geändert. Die `take`-Methode (❸) funktioniert im Prinzip analog. Hier wird in der `do-while`-Schleife die Referenz ausgelesen. Mit `compareAndSet` wird dann überprüft, ob die gelesene Referenz gültig ist. Ist dies der Fall, wird die Datenstruktur zurückgesetzt (mit `null` und `false`).

Die Verwendung von `AtomicMarkableReference` ist hier deshalb erforderlich, da `null` als ein gültiger Wert zugelassen ist. Ist `null` nicht erlaubt, reicht für die Realisierung `AtomicReference` aus. Der `null`-Wert kann dann als implizite Marke benutzt werden.

7.3 Accumulator und Adder in Java 8

Mit Atomic-Klassen kann der Overhead der Synchronisation nicht gänzlich eliminiert, sondern nur reduziert werden. Konkurrieren viele Threads um dieselbe Variable, kann es zu einem Performance-Problem kommen. Mit einem Lock ist der Verwaltungsaufwand zwar hoch, die Threads können aber schlafen gelegt werden und die Rechenressourcen werden an andere Aktivitäten verteilt. Beim Atomic-Konzept müssen dagegen die Threads aktiv auf die Busfreigabe warten, was unter Umständen die Systemleistung stark beeinflusst.

Betrachten wir ein Beispiel: In einer Multithreaded-Anwendung sollen die bearbeiteten Bytes und die empfangenen Nachrichten gezählt werden. Für diese Aufgabe ist es naheliegend, für jeden Zähler ein Objekt vom Typ `AtomicLong` zu benutzen, wobei die beiden Zähler selten gelesen werden. Dagegen sprechen aber einige Argumente:

1. Bei einer großen Anzahl von Threads kann theoretisch ein Thread aufgrund eines *Spinlocks*[4] endlos in der `compareAndSet`-Schleife laufen.
2. Das Erhöhen des Atomic-Zählers verstärkt den sogenannten Cache-Kohärenz-Verkehr, da alle Threads auf dieselbe Variable zugreifen. Die Caches müssen ständig in einen konsistenten Zustand gebracht werden.

[4] In unserem Fall ein aktives Warten auf der Hardwareebene auf eine erfolgreiche Änderung der Speicherstelle. Ein Spinlock kann die Performance des Programms stark negativ beeinflussen, insbesondere wenn viele Threads um dieselbe Speicherstelle konkurrieren.

Es wäre effizienter, wenn jeder Thread einen eigenen Zähler besitzt und wir einen Mechanismus haben, mit dem die gesamte Anzahl bei Bedarf berechnet wird.

Zu diesem Zweck wurden die Klassen `LongAccumulator` und `DoubleAccumulator` eingeführt. Abgesehen vom Datentyp ist der Umgang mit den beiden identisch. Jeder Thread erhält einen eigenen Eintrag in einer Tabelle. Erst beim Lesen werden die Einträge zusammengerechnet. Dabei wird die Cache-Problematik durch die Verwendung von `volatile` und Padding (unbenutzte Variablen, um eine Cache-Line zu füllen) berücksichtigt.

Der Konstruktor `LongAccumulator(LongBinaryOperator f, long identity)` erwartet ein Objekt mit dem Interface

```
public interface LongBinaryOperator
{
   public long applyAsLong(long left, long right);
}
```

und einen `long`-Wert für den Reset (der Reset-Wert entspricht dem neutralen Element des Operators. Er wird verwendet, um alle Zähler zu initialisieren bzw. zurückzusetzen. Bei einer Addition wäre das die Null, bei einer Multiplikation die Eins).

Eine spezielle Klasse ist `LongAdder`, bei der der Operator mit der gewöhnlichen Addition übereinstimmt. Ein `new LongAdder()` entspricht somit:

```
new LongAccumulator((x,y)-> x + y, 0L);
```

Einige wichtige Methoden sind:

- `public void accumulate(long x)`: Der Zähler soll um `x` geändert werden.
- `public long get()`: Der aktuelle Wert wird zurückgegeben.
- `public void reset()`: Alle internen Zähler werden mit dem angegebenen neutralen Element zurückgesetzt.

Praxistipp

Wenn die Performance eine sehr große Rolle spielt und man den akkumulierten Wert erst nach dem Ende der Berechnung benötigt, dann ist es sogar

effizienter, dass jeder Thread seinen eigenen (nicht `volatile`-) Zähler hat und man die Werte erst nach dem Ende der Threads (`join`) einsammelt.

7.4 Zusammenfassung

Mit dem Paket `java.util.concurrent.atomic` werden lockfreie, Thread-sichere Zugriffe auf einzelne Variablen unterstützt. Für Wahrheitswerte steht die Klasse `AtomicBoolean` zur Verfügung. Für ganzzahlige Typen sind Objekte der Klassen `AtomicInteger`, `AtomicLong` zu wählen. Um gemeinsam referenzierte Objekte atomar auszutauschen, gibt es die Klassen `AtomicReference`, `AtomicMarkableReference` und `AtomicStampedReference`.

Werden gemeinsame Variablen häufig durch mehrere Threads modifiziert und selten gelesen, dann sollte man Objekte der Klasse `LongAccumulator` bzw. `DoubleAccumulator` oder die speziellen Versionen `LongAdder` und `DoubleAdder` verwenden.

In diesem Paket sind außerdem einige hier nicht besprochene Hilfsklassen definiert, mit denen man Arrays von Atomic-Daten manipulieren kann: `AtomicIntegerArray`, `AtomicLongArray` und `AtomicReferenceArray`. Für den Umgang mit unabhängigen Atomic-Attributen eines Objekts sind die Klassen `AtomicReferenceFieldUpdater`, `AtomicIntegerFieldUpdater` und `AtomicLongFieldUpdater` nützlich.