Javaの理論と実践: アトミックで行く

新しいアトミック・クラスはjava.util.concurrentの秘宝

Brian Goetz
Principal Consultant
Quiotix

2004年 11月 23日

JDK 5.0までは、Java言語でwait-free、lock-freeなアルゴリズムを書くのはネイティブ・コードを使わない限り不可能でした。java.util.concurrentにアトミック変数クラスが追加されたことによって、それが変わるのです。この新しいクラスによって、いかに高度にスケーラブルな非ブロック・アルゴリズムがJavaで開発できるようになったかを、並行性のエキスパートであるBrian Goetzが解説して行きます。

このシリーズの他の記事を見る

15年前には、マルチプロセッサー・システムは非常に専門的なものであり、何十万ドルもするものでした (そして大部分は2つから4つのプロセッサーしか備えていませんでした) 。現在ではマルチプロセッサー・システムは安価で手軽なものであり、また主なマイクロプロセッサーはどれもマルチプロセシングのサポートを組み込んでいます。しかもその多くは、何十あるいは何百というプロセッサーをサポートしています。

マルチプロセッサー・システムの力を活用するために、アプリケーションは通常、複数スレッドを使って構成されます。ところが並行アプリケーションを書いたことのある人であれば分かると思いますが、単純に作業を複数スレッドに分割するだけでは、ハードウェアの有効利用には不十分なのです。つまり、スレッドが大部分の時間を実際の作業に費やすようにすべきであり、仕事待ちの状態になったり、共有データ構造に対してロックを待っていたりするようではいけません。

問題： スレッド間での協調

実際には、スレッド間で何ら協調(coordination)を必要としないほど、本当の意味で並行処理ができるタスクはほとんどありません。実行されるタスクが、通常はお互いに独立になっているスレッド・プールを考えてみてください。もしスレッド・プールが共通のワーク・キューを処理しているのであれば、ワーク・キューから要素を取り除いたり要素を追加したりするプロセスはスレッド・セーフである必要があり、それはつまり、先頭や最後のノード、ノード間などのリンク・ポインターへのアクセスの協調を意味することになります。そして、全てのトラブルの元凶となるのが、この協調なのです。

標準的な手法： ロッキング

Java言語において、共有フィールドへのアクセスを協調するための伝統的な方法は同期を使うことです。同期を使うことによって、共有フィールドへのアクセスは全て、適切なロックを保持して行われるようにします。同期では（クラスが適切に書かれていれば）、変数セットを保護するロックをどのスレッドが保持している場合でも、そのスレッドはその変数セットに排他アクセスができます。変数が変更された場合には、後で他のスレッドがロックを取得すると、そのスレッドには、加えられた変更が見えるようになります。この欠点としては、競合下での同期化は非常に高価につくので、ロックの競合が激しい場合（スレッドは、他のスレッドが既にロックを保持している時には、頻繁にロック取得を要求します）にはスループットが低下することです（お知らせ：非競合同期化(uncontended synchronization)は最近のJVMでは極めて安価です）。

ロック・ベースのアルゴリズムのもう一つの問題は、ロックを保持するスレッドが（ページ・フォールトやスケジュール遅延、その他予期しない遅延などによって）遅れると、そのロックを要求するスレッドがどれも、全く進行しなくなることです。

共用変数(shared variables)を保存するためとして、同期よりも低いコストで揮発性変数(Volatile variables)も使うことができるのですが、これには制限があります。揮発性変数に書き込むものが何なのかは、他のスレッドから直ちに見えることが保証されているのですが、read-modify-writeというシーケンスの操作をアトミックに表現する方法がありません。つまり例えば、揮発性変数は、mutex（mutual exclusion lock: 相互排他ロック）またはカウンターを、確実に実装するためには使うことができない、ということです。

ロック付きのカウンターとmutexを実装する

ここで、`get()`、`increment()`そして`decrement()`操作をエクスポーズするスレッド・セーフなカウンター・クラスを開発することを考えてみてください。リスト1は、ロック（同期）を使ってこのクラスを実装する方法の一例です。そのクラスがスレッド・セーフであるためには、どのアップデートも失われず、またすべてのスレッドがカウンターの最新値を見るようにするために、`get()`までも含めて全てのメソッドを同期する必要があることに注意してください。

リスト1. 同期化したカウンター・クラス

```
public class SynchronizedCounter {
    private int value;
    public synchronized int getValue() { return value; }
    public synchronized int increment() { return ++value; }
    public synchronized int decrement() { return --value; }
}
```

`increment()`操作と`decrement()`操作は、アトミックなread-modify-write操作です。安全にカウンターを増加するには、現在値を取得し、それに1を加え、新しい値を書き込むという操作を、他のスレッドには妨害不能な、単一操作として行う必要があります。そうしないと、もし2つのスレッドが同時に増加を実行しようとした時に、差し込み操作（interleaving of operations）がうまく行われなければ、カウンターは2カウントの増加ではなく、1カウントの増加で終わってしまいます（値のインスタンス変数を揮発性にすることでは、この操作を確実に行うことはできないことに注意してください）。

アトミックなread-modify-write組み合わせは、多くの並行アルゴリズムに出現します。リスト2のコードは単純なmutexを実装しており、`acquire()`メソッドもまたアトミックなread-modify-write

操作です。mutexを取得するには、他はどれもmutexを保持していないことを確認し（`curOwner == null`）、その後で自分がそれを所持しているという事実を記録する必要があります。しかもその全てを、他のスレッドが途中で入ってきて`curOwner field`を修正する可能性が無い中で行う必要があるのです。

リスト2. 同期化したmutexクラス

```
public class SynchronizedMutex {
    private Thread curOwner = null;
    public synchronized void acquire() throws InterruptedException {
        if (Thread.interrupted()) throw new InterruptedException();
        while (curOwner != null) wait();
        curOwner = Thread.currentThread();
    }
    public synchronized void release() {
        if (curOwner == Thread.currentThread()) {
            curOwner = null;
            notify();
        } else
            throw new IllegalStateException("not owner of mutex");
    }
}
```

リスト1のカウンター・クラスは確実に動作します。そして競合がほとんどないか、全くない場合には問題なく動作します。ところが競合が激しい場合には、JVMはカウンターを増加するなどの実際の仕事をするよりも、スレッドのスケジューリングや、競合や待ち状態にあるスレッドのキュー管理に多くの時間を費やしてしまうので、劇的にパフォーマンスが低下してしまいます。前回の記事のグラフを思い出してもらうと分かると思いますが、一旦複数のスレッドが同期を使って組み込みのモニター・ロックに対する競合を始めると、スループットが劇的に低下してしまいます。その記事では同期の置き換えとして、新しい`ReentrantLock`クラスの方がスケーラブルであることを説明しましたが、一部の問題に対しては、さらに良い手法があるのです。

ロックにおける問題

ロックでは、あるスレッドが既に保持しているロックを別のスレッドが取得しようとすると、ロックが取得できるようになるまで、ロックを保持しているスレッドは他のスレッドをブロックしてしまいます。この手法には明らかに幾つかの欠点がありますが、その一つは、あるスレッドがロックを待ってブロックされていると、そのスレッドは他に何もできない、ということです。このシナリオは、ブロックされているスレッドが優先度の高いタスクであったりすると悲劇的なものになります（priority inversionとして知られる危険性です）。

ロックを使うと、デッドロック（複数のロックが不規則な順序で取得される時に起こります）のような、他の危険性もあります。こうした危険性が無い場合でも、ロックは単純に言って比較的キメの粗い協調機構なのです。従ってカウンターを増加したり、誰がmutexを所有するのかを更新したりといった単純な操作を管理するには、極めて「重量級」です。ですから、個々の変数に対する並行処理の更新を確実に管理する、キメの細かい機構があることが望ましいと言えます。そして、最近のほとんどのプロセッサーは、そうしたキメ細かい機構を備えているのです。

ハードウェア同期プリミティブ

先に述べた通り、最近のほとんどのプロセッサーは、マルチ・プロセシングをサポートしています。そのため当然ながら、複数のプロセッサーが周辺機器やメイン・メモリーを共有できるの

ですが、一般的には命令セットも強化されており、マルチプロセシングに対する特別な要求をサポートするようになっています。特に最近ではほとんど全てのプロセッサーが、他のプロセッサーからの並行アクセスを検出あるいは防止するような方法で、共用変数を更新する命令を持っています。

Compare and swap (CAS)

並行処理をサポートした初期のプロセッサーは、アトミックなtest-and-set操作を提供していましたが、これは普通、単一ビットに対して操作を行うものでした。IntelやSparcプロセッサーを含めて、現在のプロセッサーで最も一般的な手法は、compare-and-swapまたはCASと呼ばれるプリミティブを実装することです。（Intelプロセッサーでは、compare-and-swapはcmpxchg命令ファミリーによって実装されており、PowerPCプロセッサーには同じことを実現する「load and reserve」と「store conditional」という対の命令があります。MIPSの場合も、最初の方が「load linked」と呼ばれることを除いて似ています。）

CAS操作には、メモリー位置（V）、期待される古い値（A）そして新しい値（B）という3つのオペランドがあります。プロセッサーは、そこにある値と期待される古い値が一致する場合には、メモリー位置を新しい値にアトミックに更新します。一致しない場合には何もしません。いずれの場合でもプロセッサーは、CAS命令の前にその位置にあった値を返します。（一部のCASでは、現在の値をフェッチせずに、単純にCASが成功したかどうかのみを返します。）つまりCASは実質的に次のように言うのです。「メモリー位置Vは値Aを持っているはずです。もし持っていたら、そこにBを入れなさい。そうでなかったら、変更してはいけません。でもそこの、今の値を言いなさい。」

同期化に対するCASの使い方として自然なのは、まずアドレスVから値Aを読み、新しい値Bを得るために複数ステップの演算を行い、それからCASを使ってVの値をAからBに変更する、というものです。その間、Vにある値が変化しなければCASは成功します。

CASのような命令を使うと、もし他のスレッドが変数を変更するとCASがそれを検出（または失敗）し、アルゴリズムがその操作を繰り返すことができるので、他のスレッドが途中で変数を変更してしまう、という心配をせずにread-modify-writeシーケンスを実行するアルゴリズムが可能になります。リスト3はCAS操作の振る舞いを示しています（パフォーマンス特性は示していません）。しかしCASの真価は、ハードウェアで実装でき、しかも（大部分のプロセッサーでは）非常に軽量だということです。

リスト3. compare-and-swapの振る舞いを示すコード（パフォーマンスを示すわけではない）

```
public class SimulatedCAS {
     private int value;
     public synchronized int getValue() { return value; }
 public synchronized int compareAndSwap(int expectedValue, int newValue) {
          int oldValue = value;
          if (value == expectedValue)
              value = newValue;
          return oldValue;
     }
}
```

CASでカウンターを実装する

CASベースの並行アルゴリズムは、スレッドがロックを待つ必要が全くないので、lock-freeと呼ばれます（使用するスレッド・プラットフォームの用語によっては、mutexまたは致命的セクション（critical section）と呼ばれることもあります）。CAS操作が成功するにせよ失敗するにせよ、どちらの場合であっても、操作は予測できる時間内に完了します。もしCASが失敗すると、呼び出し側はCAS操作をリトライするか、あるいは適切と思われる別のアクションを起こします。リスト4は、ロックの代わりにCASを使うように書き直したカウンター・クラスを示しています。

リスト4. compare-and-swapでカウンターを実装する

```
public class CasCounter {
    private SimulatedCAS value;
    public int getValue() {
        return value.getValue();
    }
    public int increment() {
        int oldValue = value.getValue();
        while (value.compareAndSwap(oldValue, oldValue + 1) != oldValue)
            oldValue = value.getValue();
        return oldValue + 1;
    }
}
```

Lock-freeアルゴリズムとwait-freeアルゴリズム

他のスレッドに任意の遅れが生じても（あるいは失敗が起きても）、どのスレッドも継続して進行する場合には、そのアルゴリズムはwait-freeと言われます。対照的にlock-freeアルゴリズムでは、一部のスレッドのみが継続して進行すればよいのです。（別の仕方で定義すると、wait-freeでは他のスレッドのアクションやタイミング、インターリーブ、またはスピードによらず、各スレッドはそのスレッドのステップとして指定された数の演算操作を正しく行うことが保証されている、ということです。この指定数は、システム中にあるスレッド数の関数になっている場合があります。例えば10のスレッドがそれぞれ`CasCounter.increment()`操作を一度行うと、最悪の場合、各スレッドは増加操作を完了するまでに最大9回のリトライを行う必要があることになります。）

過去15年の間、wait-freeアルゴリズムとlock-freeアルゴリズムには広範な研究が行われ、数多くの一般的なデータ構造に対して非ブロック・アルゴリズム（nonblocking algorithms）が発見されています。非ブロック・アルゴリズムは、OSやJVMのレベルで、スレッドやプロセスのスケジューリングなどのタスクに対して広く使われています。非ブロック・アルゴリズムは実装がより複雑ですが、ロック・ベースのアルゴリズムよりも有利な点が幾つかあります。つまりpriority inversionやデッドロックのような危険性が回避され、競合のコストは低くなり、協調はよりキメ細かなレベルで起こるので、より高度な並行処理が実現するのです。

アトミックな変数クラス

JDK 5.0までは、ネイティブ・コードを使わない限りJavaでwait-freeやlock-freeのアルゴリズムを書くことは不可能でした。それが、`java.util.concurrent.atomic`パッケージにアトミック変数クラスが追加されたことによって変わりました。アトミック変数クラスはすべてcompare-and-setプリミティブ（compare-and-swapと似ています）をエクスポーズしますが、これはそのプラットフォームで使用できる中で最も速いネイティブ構成体（compare-

and-swap、load linked/store conditional、または最悪の場合spin locks ）を使って実装されます。java.util.concurrent.atomicパッケージには、それぞれ少しずつ異なる9種類のアトミック変数が用意されています（ AtomicInteger、AtomicLong、AtomicReference、AtomicBoolean、配列形式のアトミック整数、long、参照、そして一対の値をアトミックに更新する、アトミックなマーク付き参照とスタンプ付き参照クラス ）。

アトミック変数クラスは、アトミックな、条件付きcompare-and-set更新をサポートするように揮発性変数の概念を拡張したvolatile変数の汎用化と考えることができます。アトミック変数の読み取りと書き込みに関するメモリー意味体系は、揮発性変数への読み書きアクセスと同じです。

アトミック変数クラスは表面的にはリスト1のSynchronizedCounterの例と同じように見えるかも知れませんが、あくまでも表面的にそう見えるだけです。アトミック変数に対する操作は、裏ではcompare-and-swapなど、そのプラットフォームが並行アクセスに対して提供するハードウェア・プリミティブになります。

キメが細かいということは軽量であるということ

競合が起きている並行アプリケーションのスケーラビリティーを調整するために一般的に使われる手法は、より多くのロック取得が競合状態から非競合状態になることを祈って、使われているロック・オブジェクトのキメを粗くすることです。ロックからアトミック変数に変換することによっても同じ結果が得られます。つまり、よりキメの細かい協調機構に切り換えることによって、競合する操作は少なくなり、スループットが改善します。

ABA問題

CASは基本的に、Vを変更する前に「Vの値は今でもAなのか」と尋ねるので、最初にVが読み取られてからVに対してCASが実行されるまでの間に値がAからBへそして再びAへと変わると、CASベースのアルゴリズムが混乱してしまうことがあり得ます。このような場合、CAS操作は成功するのですが、ある状況では望ましくない結果になる可能性があります。（ リスト1とリスト2での、カウンターとmutexの例ではこの問題は起きませんが、全てのアルゴリズムで起きないわけではないことに注意してください。 ）この問題はABA問題と呼ばれています。これに対処するには普通、compare-and-swapされる値に対してタグやバージョン番号を関連付け、値とタグの両方をアトミックに更新するようにします。AtomicStampedReferenceクラスでは、この手法に対するサポートを提供しています。

java.util.concurrentでのアトミック変数

java.util.concurrentパッケージのクラスはほとんど全て、直接または間接的に、同期ではなくアトミック変数を使います。ConcurrentLinkedQueueのようなクラスは、wait-freeアルゴリズムを直接実装するためにアトミック変数を使います。またConcurrentHashMapのようなクラスは、ロックが必要なところでReentrantLockを使います。ReentrantLockは今度は、ロックを待っているスレッドのキューを維持するためにアトミック変数を使います。

こうしたクラスは、JDK 5.0でのJVMの改善がなければ構成することはできませんでした。JDK 5.0では、（ ユーザー・クラスに対してではなく、クラス・ライブラリーに対して ）ハードウェア・レベルでの同期プリミティブに対してアクセスするインターフェースをエクスポーズしています。こうした機能をユーザー・クラスに対してエクスポーズするのは、アトミック変数クラスや、java.util.concurrentにあるその他のクラスが行います。

アトミック変数で高いスループットを達成する

前回の記事では、同期よりもReentrantLockクラスがスケーラビリティーの点でいかに有利かを説明しました。そしてサイコロを振るのを疑似乱数生成でシミュレートした、単純な、高競合状態のベンチマーク例を構成しました。そこでは同期を使っての協調、ReentrantLockを使った協調、フェアなReentrantLockを使った協調のそれぞれの実装と、その結果を示しました。今回はそのベンチマークにもう一つの実装を追加し、AtomicLongを使ってPRNG状態を更新するものを追加します。

リスト5は同期を使ったPRNG実装と、CASを使ったもう一つの実装を示しています。CASは（CASを使うコードでは必ずそうですが）成功するまでに一度ならず失敗する可能性があるので、ループで実行する必要があることに注意してください。

リスト5. 同期とアトミック変数を使って、スレッドセーフなPRNGを実装する

```
public class PseudoRandomUsingSynch implements PseudoRandom {
    private int seed;
    public PseudoRandomUsingSynch(int s) { seed = s; }
    public synchronized int nextInt(int n) {
        int s = seed;
        seed = Util.calculateNext(seed);
        return s % n;
    }
}
public class PseudoRandomUsingAtomic implements PseudoRandom {
    private final AtomicInteger seed;
    public PseudoRandomUsingAtomic(int s) {
        seed = new AtomicInteger(s);
    }
    public int nextInt(int n) {
        for (;;) {
            int s = seed.get();
            int nexts = Util.calculateNext(s);
            if (seed.compareAndSet(s, nexts))
                return s % n;
        }
    }
}
```

下記の図1と図2は、アトミック・ベースの手法を示す折れ線が一本追加されただけで、前回示したものと似ています。このグラフでは、8つのプロセッサーを持つUltrasparc3マシンと単一プロセッサーのPentium 4マシンで様々な数のスレッドを使った場合の、乱数生成に対するスループット（毎秒何回サイコロを振ったか）を示しています。ただしテストで使われたスレッドの数は勘違いしやすいかも知れません。これらのスレッドは通常よりはるかに多くの競合を示しているため、現実的なプログラムの場合よりもずっと少ないスレッド数で、既にReentrantLockとアトミック変数が優劣をつけにくくなっています。これを見ると、同期よりもずっと有利なReentrantLockよりも、アトミック変数の方がさらに改善を示していることが分かるでしょう。（それぞれの作業単位で行われることがほとんど無いので、下記のグラフではReentrantLockに比較して、アトミック変数のスケーラビリティーでの利点が控えめなものになっています。）

図1. 8 CPUのUltrasparc3での、同期、ReentrantLock、フェア・ロック、AtomicLongのスループット比較

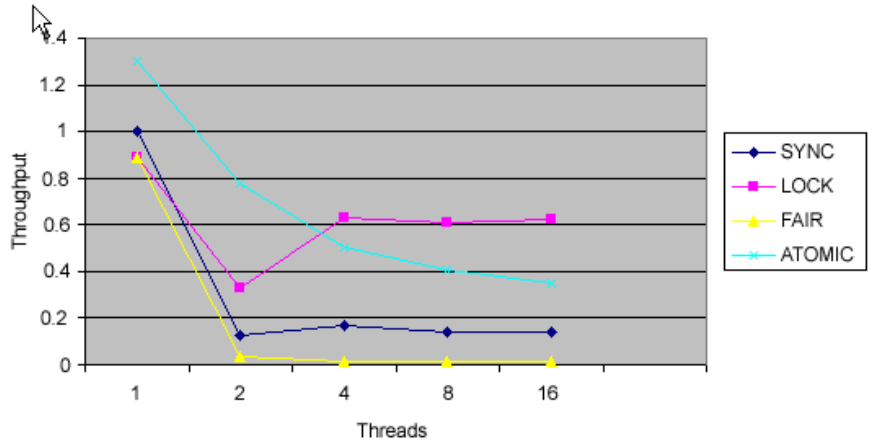
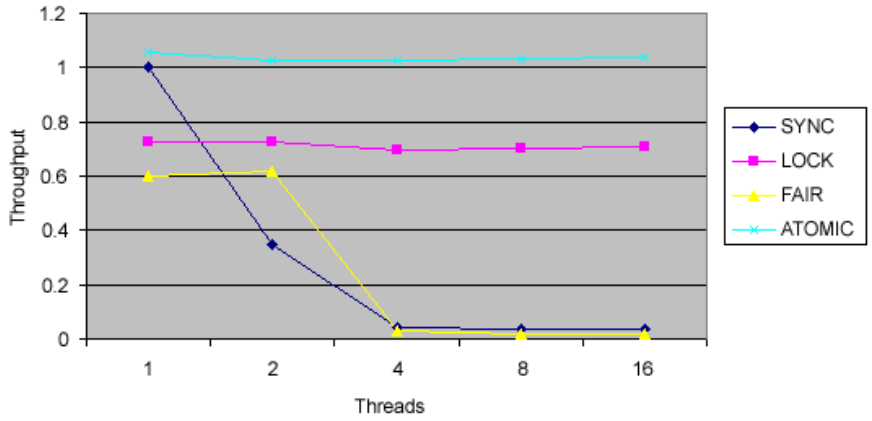


図2. Pentium 4単一プロセッサーでの、同期、ReentrantLock、フェア・ロック、AtomicLongのスループット比較



自分独自のアトミック変数を使って非ブロック・アルゴリズムを開発するユーザーは少ないと思います。ほとんどのユーザーは恐らく、`ConcurrentLinkedQueue`など、`java.util.concurrent`で提供されているものを使うでしょう。こうしたクラスがパフォーマンス強化された秘密はどこにあるのかを不思議に思う人のために念のため説明すると、以前のJDKにあるものに比べ、アトミック変数クラスを通してエクスポーズされる、キメの細かい、ハードウェア・レベルの並行性プリミティブが使われているのです。

開発者の中にはアトミック変数を、共有カウンターや連続番号生成その他独立の共用変数に対するハイ・パフォーマンスな直接的置き換え、つまりアトミック変数が無ければ同期で保護すべき変数の直接的な置き換えと考える人もいるかも知れません。

まとめ

JDK 5.0はハイ・パフォーマンスな並行クラス開発のためには非常に大きな前進です。新しい、下位レベルでの協調プリミティブを内部的にエクスポーズし、そしてパブリックなアトミック変数クラス・セットを提供することによって、wait-free、lock-freeのアルゴリズムをJava言語で開発することが、これで初めて現実的なものになったのです。そうすると`java.util.concurrent`のクラスはこうした下位レベルのアトミック変数機構の上に構成され、似たような機能を実行するこれまでのクラスに比べて、スケーラビリティーの面で大幅に有利になります。アトミック変数をクラスの中で直接使うことはないかも知れませんが、アトミック変数があることを喜ぶだけの理由は十分あるのです。

著者について

Brian Goetz

Brian Goetz は18 年間以上に渡って、専門的ソフトウェア開発者として働いています。彼はカリフォルニア州ロスアルトスにあるソフトウェア開発コンサルティング会社、Quiotixの主席コンサルタントであり、またいくつかのJCP Expert Groupの一員でもあります。2005年の末にはAddison-Wesleyから、Brianによる著、Java Concurrency In Practiceが出版される予定です。Brian著による有力業界紙に掲載済みおよび掲載予定の記事のリストを参照してください。