Javaの理論と実践: ノンブロッキング・アルゴリズムの紹介

ママ、見て、鍵がかかってないよ！

Brian Goetz
Principal Consultant
Quiotix

2006年 4月 18日

Java™ 5.0は、Java言語におけるノンブロッキング・アルゴリズムの開発を初めて可能にしました。java.util.concurrentパッケージは、この機能を広範囲にわたって活用しています。ノンブロッキング・アルゴリズムは並行アルゴリズムです。並行アルゴリズムのスレッド安全性はロック処理に由来するのではなく、compare-and-swapなどの、マシン・レベルに近いアトミックなハードウェア・プリミティブに由来します。ノンブロッキング・アルゴリズムの設計と実装はきわめて困難になる場合がありますが、より高いスループットが期待でき、デッドロックや優先順位の逆転などのliveness（ライブネス）問題に対してより強固な対応が可能になります。今回の「Javaの理論と実践」の記事では、並行アルゴリズムの専門家であるBrian Goetzが、単純なノンブロッキング・アルゴリズムの仕組みを説明します。

このシリーズの他の記事を見る

複数のスレッドが変更可能な変数にアクセスする場合、すべてのスレッドが同期化を使用してアクセスする必要があります。同期化せずにアクセスすると、大変なことが発生してしまう場合があります。Java言語における同期化の主要な手段は、同期化キーワード（イントリンシック・ロッキングともいいます）です。この手段は、相互排他を強制し、同期化ブロックを実行しているスレッドのアクションが他のスレッドから認識できるようにします。このスレッドのアクションが終了すると、同じロックで保護された同期化ブロックが他のスレッドによって実行されます。イントリンシック・ロッキングを正しく使用した場合、プログラムは「スレッド安全」になりますが、スレッドによるロックの競合が頻繁に発生する場合に短いコード・パスをロックで保護するのは、かなり負荷のかかる操作になります。

「アトミックで行く」の記事ではアトミック変数を紹介しました。アトミック変数を使用することにより、ロック処理をすることなく、共有変数を安全に更新するためのアトミックなread-modify-write操作が可能になります。アトミック変数は揮発性変数と同様のメモリー・セマンティクスを持ちますが、アトミック変数の場合は原子的に変更できるため、ロック処理を必要としない並行アルゴリズムの基礎として使用することができます。

ノンブロッキング・カウンター

リスト1のCounterはスレッド安全になっていますが、ロックを使用する必要があるためにパフォーマンスが犠牲になり、開発者を悩ませます。incrementは単一操作のように見えますが、値をフェッチし、それに1を加え、結果の値を書き出すという3つの個別の操作をまとめて行っているため、ロックが必要になります（getValueメソッドでは、getValueを呼び出したスレッドが最新の値を参照できるように、同期処理も必要になります。ロック処理の必要性を単純に無視するのは望ましいことではありませんが、無視してもかまわないと自分を納得させてしまう開発者が驚くほど多いようです）。

複数のスレッドが同じロック処理を同時に要求した場合、ロックを取得するのは1つのスレッドだけで、その他のスレッドはブロックされます。JVMは通常、ブロックされたスレッドを一時中止し、後からスケジュールしなおすことによってブロッキングを実装します。その結果としてのコンテキスト・スイッチによる遅延は、ロックによって保護される命令が少ない割には、かなりの遅延となる場合があります。

リスト1. 同期を使用するスレッド安全なカウンター

```
public final class Counter {
    private long value = 0;

    public synchronized long getValue() {
        return value;
    }

    public synchronized long increment() {
        return ++value;
    }
}
```

リスト2のNonblockingCounterは、最も単純なノンブロッキング・アルゴリズムの1つを示しています。すなわち、AtomicIntegerのcompareAndSet()（CAS）メソッドを使用するカウンターです。compareAndSet()メソッドは、「この変数をこの新しい値に更新すること。ただし、このメソッドが最後に参照した後で他のスレッドによって値が変更された場合は、更新処理を失敗すること」という指示を出します（アトミック変数とcompare-and-setの詳しい説明については、「アトミックで行く」の記事を参照してください）。

リスト2. CASを使用するノンブロッキング・カウンター

```
public class NonblockingCounter {
    private AtomicInteger value;

    public int getValue() {
        return value.get();
    }

    public int increment() {
        int v;
        do {
            v = value.get();
        while (!value.compareAndSet(v, v + 1));
        return v + 1;
    }
}
```

アトミック変数クラスが「アトミック」と呼ばれるのは、数値参照とオブジェクト参照の詳細でアトミックな更新を提供するためですが、ノンブロッキング・アルゴリズムの基本的ビルディング・ブロックであるという意味でも「アトミック」です。ノンブロッキング・アルゴリズムは20年以上前から多くの調査・研究の対象でしたが、Java言語で初めて可能になったのはJava 5.0からです。

最近のプロセッサーには、共有データをアトミックに更新するための特殊な命令が用意されています。この命令によって他のスレッドからの干渉を検出することができ、compareAndSet() はこれらをロックの代わりに使用します（カウンターの値を増加するだけでよい場合は、AtomicIntegerが増加処理の手段を提供しますが、こうした手段はNonblockingCounter.increment() と同様にcompareAndSet() に基づいています）。

ノンブロッキング版にはロックベース版に比べて、パフォーマンス上の利点がいくつかあります。JVMロッキング・コード・パスの代わりにハードウェア・プリミティブを使用してより細かな粒度（個別のメモリー位置）で同期し、ロック処理の競合に負けたスレッドは一時中止されて再スケジュールされるのではなく、ただちに再試行することができます。粒度が細かいために競合の可能性が少なくなり、スケジュールし直すことなく再試行できるので競合のコストが小さくなります。CAS操作がいくつか失敗したとしても、ロック競合のために再スケジュールするより、このアプローチの方が処理は速くなります。

NonblockingCounterは単純な例かもしれませんが、すべてのノンブロッキング・アルゴリズムの基本的特性をよく表しています。すなわち、CASが成功しなかった場合はやり直す必要があるかもしれないという前提で、いくつかのアルゴリズム・ステップが推論的に実行されます。ノンブロッキング・アルゴリズムは干渉がないという前提で処理が行われるため、楽観的なアルゴリズムと呼ばれることがよくあります。干渉が検出された場合は、処理を後戻りして再び実行します。上述のカウンター処理の場合、推論的ステップはカウウンターの増加処理になります。更新の計算中に値が変わらないという前提で、元の値をフェッチして1を加えます。前提が間違っていた場合は、値を再びフェッチしてから増分計算をやり直す必要があります。

ノンブロッキング・スタック

もう少し複雑なノンブロッキング・アルゴリズムの例は、リスト3のConcurrentStackです。ConcurrentStackのpush() およびpop() 操作は、いずれもNonblockingCounterのincrement() と構造的には同じです。いくつかの処理を推論的に行い、その処理を「コミット」するときに基本前提が無効になっていないことを前提としています。push() メソッドは、現在の上位ノードを監視してスタックにプッシュする新しいノードを構築し、最上部のノードが監視を始めてから変化していない場合に、新しいノードをインストールします。CASが失敗した場合は別のスレッドがスタックを変更したことを意味するため、プロセスが再び開始されます。

リスト3. Treiberのアルゴリズムを使用するノンブロッキング・スタック

```
public class ConcurrentStack<E> {
    AtomicReference<Node<E>> head = new AtomicReference<Node<E>>();

    public void push(E item) {
        Node<E> newHead = new Node<E>(item);
        Node<E> oldHead;
        do {
            oldHead = head.get();
            newHead.next = oldHead;
```

```
        } while (!head.compareAndSet(oldHead, newHead));
    }

    public E pop() {
        Node<E> oldHead;
        Node<E> newHead;
        do {
            oldHead = head.get();
            if (oldHead == null)
                return null;
            newHead = oldHead.next;
        } while (!head.compareAndSet(oldHead,newHead));
        return oldHead.item;
    }

    static class Node<E> {
        final E item;
        Node<E> next;

        public Node(E item) { this.item = item; }
    }
}
```

パフォーマンスに関する考慮事項

競合の緩和という点で、ノンブロッキング・アルゴリズムはブロッキング・アルゴリズムよりも優れています。ほとんどの場合、CASは最初の実行で成功するためです。競合が発生した場合でも、スレッドの一時中止とコンテキスト・スイッチングは不要です。この場合は、ループをさらに数回反復するだけで済みます。競合のないCASは競合のないロック取得より低コストで（競合のないロック取得にはCASのほかに追加の処理が必要になるため、これは正しいはずです）、競合のあるCASも競合のあるロック取得より遅延が短くなります。

競合が激しい場合、すなわち、多くのスレッドが1つのメモリー位置に殺到する場合、ロックベースのアルゴリズムの方がノンブロッキング・アルゴリズムよりもスループットが高くなります。これは、スレッドがブロックされると、無理にメモリーを獲得することをあきらめておとなしく順番を待つようになるため、それ以上の競合が発生しなくなるからです。ただし、これほど高い競合レベルが発生することはまれであり、ほとんどの場合、スレッド内で行う計算が共有データを争う操作にインターリーブされるため、他のスレッドが共有データへアクセスできないということはありません（これほど高い競合レベルが発生する場合は、共有データを少なくする方向でアルゴリズムを見直した方がいいでしょう）。「アトミックで行く」のグラフは、この点で少々紛らわしいものでした。測定対象のプログラムが現実にはありえないほどの高い競合レベルだったため、スレッド数がかなり少ない場合でもロック処理の方が有利であるように見えました。

ノンブロッキング・リンク・リスト

これまでの例（カウンターとスタック）は、非常に単純なノンブロッキング・アルゴリズムでした。ループ処理でCASを使用するパターンさえ把握すれば、容易に理解することができます。より高度なデータ構造の場合、ノンブロッキング・アルゴリズムはこうした単純な例に比べてはるかに複雑になります。リンク・リスト、ツリー、ハッシュ・テーブルの変更には、複数のポインターの更新処理が伴う場合があるためです。CASの場合、単一ポインターでのアトミックな条件付き更新が可能ですが、ポインターが2つの場合は不可能になります。したがって、ノンブロッキングなリンク・リスト、ツリー、ハッシュ・テーブルを構築するには、データ構造を不整合な状態にすることなく、CASを使用した複数のポインターを更新する手段を見つける必要があります。

リンク・リストの末尾に要素を挿入するには、一般に2つのポインターを更新する必要があります。すなわち、常にリストの最後の要素を参照する「tail」ポインターと、以前の最後の要素から新しく挿入された要素を参照する「next」ポインターです。2つのポインターを更新する必要があるため、2つのCASが必要になります。2つのポインターを個別のCAS操作で更新するには、考慮しなければならない2つの潜在的な問題があります。すなわち、最初のCASが成功して2番目のCASが失敗した場合にどうなるかという問題と、最初のCASと2番目のCASの間に別のスレッドがリストへのアクセスを試みた場合にどうなるかという問題です。

複雑なデータ構造体用のノンブロッキング・アルゴリズムを構築するコツは、スレッドがデータ構造体の変更を開始してから終了するまでの間も含めて、データ構造体が常に整合状態に保たれるようにすることです。また、最初のスレッドの状態（更新を終了したか、あるいは更新中なのか）だけでなく、最初のスレッドが無断で更新を停止した場合に、更新を完了するために必要な操作を他のスレッドに認識させることも必要になります。こうすることにより、あるスレッドが更新処理の現場に到着したときにデータ構造体が更新中であった場合、このスレッドは更新中のスレッドに協力して処理を終了させ、その後改めて自分自身の処理を続行することができるようになります。最初のスレッドが更新処理を再開しようとすると、更新処理に協力したスレッドからの干渉（この場合は、良い意味での干渉ですが）をCASが検出するため、このスレッドは更新作業がすでに終了したことを認識して元の位置に戻ります。

このスレッド間の協力という要件は、個々のスレッドの失敗に対してデータ構造体がより強固に対応するために必要な要件です。スレッドが更新処理の現場に到着したときに、データ構造体が別のスレッドによって更新中で、そのスレッドの更新処理が終了するのをただ待つだけだとしたら、更新処理の途中で失敗した場合、後から来たスレッドは永遠に待つことになってしまいます。失敗しなかったとしても、この方法ではパフォーマンスが落ちてしまいます。後から到着したスレッドはプロセッサーを譲らなければならず、その結果コンテキスト・スイッチが発生するためです。さらに悪いことには、自分の持ち時間がなくなるのを待たなければなりません。

リスト4のLinkedQueueは、Michael-Scottノンブロッキング・キュー・アルゴリズムの挿入操作を示しています。これは、ConcurrentLinkedQueueによって実装されています。

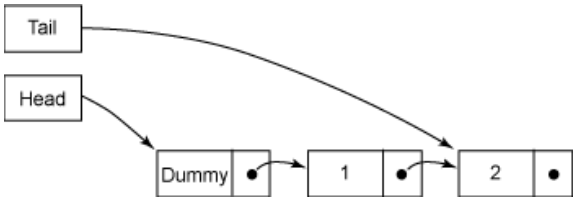
リスト4. Michael-Scottノンブロッキング・キュー・アルゴリズムにおける挿入

```
public class LinkedQueue <E> {
    private static class Node <E> {
        final E item;
        final AtomicReference<Node<E>> next;

        Node(E item, Node<E> next) {
            this.item = item;
            this.next = new AtomicReference<Node<E>>(next);
        }
    }

    private AtomicReference<Node<E>> head
        = new AtomicReference<Node<E>>(new Node<E>(null, null));
    private AtomicReference<Node<E>> tail = head;

    public boolean put(E item) {
        Node<E> newNode = new Node<E>(item, null);
        while (true) {
            Node<E> curTail = tail.get();
            Node<E> residue = curTail.next.get();
            if (curTail == tail.get()) {
```

```
                if (residue == null) /* A */ {
                    if (curTail.next.compareAndSet(null, newNode)) /* C */ {
                        tail.compareAndSet(curTail, newNode) /* D */ ;
                        return true;
                    }
                } else {
                    tail.compareAndSet(curTail, residue) /* B */;
                }
            }
        }
    }
}
```

多くのキュー・アルゴリズムと同様、空のキューは単一のダミー・ノードから構成されています。headポインターは常にダミー・ノードを指します。tailポインターは常に最後のノードか最後から2番目のノードを指します。図1は、正常な状態で2つの要素を持つキューを示しています。

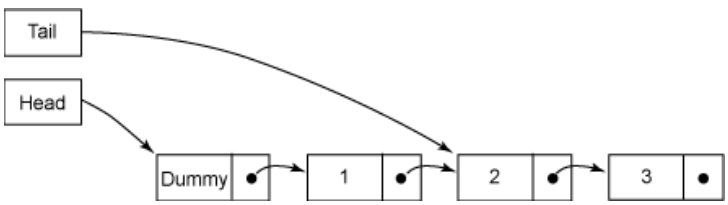
図1. 静止状態で2つの要素を持つキュー



リスト4に示されているように、要素の挿入には2つのポインターの更新が伴い、いずれの更新もCASで行われます。すなわち、キューにある現在の最後のノードから新しいノードにリンクして（C）、tailポインターを新しい最後のノードを指すように変更します（D）。このうち最初の操作が失敗した場合、キューの状態は変わらずにスレッドの挿入が成功するまで操作が実行されます。この操作が成功すると挿入は有効になったとみなされて、他のスレッドによって変更が認識されます。新しいノードを指すようにtailポインターを変更する操作が残っていますが、この操作は「後片付け」と考えることができます。更新処理の現場に到着したスレッドは、こうした後処理が必要かどうかを自分自身で判断することができ、その方法もすでにわかっています。

キューの状態は、常に正常（静止）状態（図1と図3）か中間状態（図2）のいずれかです。キューは、挿入操作の前と2番目のCAS（D）の成功後は静止状態になり、最初のCAS（sC）が成功した後は中間状態になります。静止状態では、tailによって示されるリンク・ノードの次のフィールドは常にNULLになります。中間状態では、常に非NULLになります。スレッドは、tail.nextをNULLと比較することによってキューの状態を知ることができます。これが、スレッド間で協力を行う場合の重要な要素になります。

図2. 挿入時の中間状態のキュー（新しい要素が追加された後、tailポインターが更新される前）

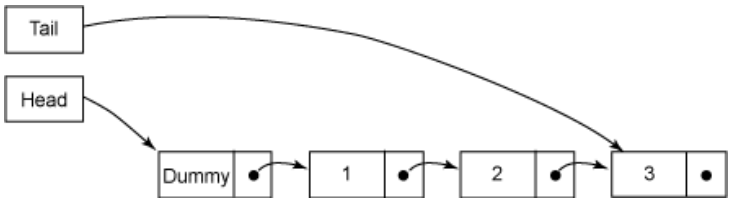


挿入操作では、リスト4に示されているように、新しい要素を挿入する前に（A）、まず、キューが中間状態かどうかをチェックします。中間状態の場合は、ステップ（C）と（D）の間で、すで

に他のスレッドが要素の挿入処理を実行中のはずです。このスレッドの終了を待つ代わりに、現在のスレッドはtailポインターを前に移動して（B）処理を終了することによって、このスレッドに協力することができます。現在のスレッドは、キューが静止状態になるまでtailポインターをチェックし、必要であればtailポインターを前に進めます。キューが静止状態になった時点で、現在のキューは自分の挿入を開始することができます。

2つのスレッドによって、キューにある現在の最後の要素へのアクセスに対する競合が発生するため、最初のCAS（C）は失敗する場合があります。その場合、変更は有効にならず、CASを失ったスレッドはtailポインターをリロードしてもう一度実行します。2番目のCAS（D）が失敗した場合、スレッドによる挿入操作を再度実行する必要はありません。もう1つのスレッドがステップ（B）で挿入操作を完了しているためです。

図3. tailポインターの更新後、再び静止状態になったキュー



舞台裏のノンブロッキング・アルゴリズム

JVMとOSの世界を覗いてみると、あらゆる所でノンブロッキング・アルゴリズムを目にすることになります。Garbage Collectorは同時並行のパラレル・ガベージ・コレクションの処理速度を上げるため、スケジューラーはスレッドとプロセスを効率的にスケジュールし、イントリンシック・ロッキングを実装するためにノンブロッキング・アルゴリズムを使用しています。Mustang（Java 6.0）では、ロックベースのSynchronousQueueアルゴリズムが新しいノンブロッキング・バージョンに置き換えられています。SynchronousQueueを直接使用する開発者はほとんどいませんが、Executors.newCachedThreadPool() ファクトリーで構築されるスレッド・プールの作業用キューとして使用されています。キャッシュ付きスレッド・プールのパフォーマンスを比較したベンチマーク・テストによると、新しいノンブロッキング同期キュー実装は、現行の実装の3倍近い速度を記録しています。また、Mustangに続くリリース（コード名Dolphin）では、さらなる改良が予定されています。

まとめ

ノンブロッキング・アルゴリズムは、ロックベースのアルゴリズムよりはるかに複雑になる傾向があります。ノンブロッキング・アルゴリズムの開発はかなり特殊な分野であり、アルゴリズムの正しさを証明するのが極めて困難な場合があります。しかし、Javaのバージョン全般に見られる並行処理の利点の多くは、ノンブロッキング・アルゴリズムの使用によるものです。並行処理の重要性が増すにつれて、Javaプラットフォームの将来のリリースでは、さらに多くのノンブロッキング・アルゴリズムが使用されるものと予想されています。

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。