Javaの理論と実践: 優れたHashMapの構築

ConcurrentHashMapにより、スレッド・セーフを犠牲にすることなく高い並行性を提供する方法

Brian Goetz
Principal Consultant
Quiotix

2003年 8月 21日

Doug Lea氏による`util.concurrent`パッケージの`ConcurrentHashMap`機能は、`Hashtable`や`synchronizedMap`よりも高い並行性が備わっています。また、正常なget()オペレーションでは、完全にロックを回避することで、並行アプリケーションにおける優れたスループットを提供することができます。今回、Brian Goetz氏は`ConcurrentHashMap`のコードの分析に取り組み、この素晴らしい機能がスレッド・セーフを犠牲にすることなく、どのように遂行されているのかについて説明をします。

このシリーズの他の記事を見る

Javaの理論および実践 (「並行コレクション・クラス」)では、スケーラビリティーのボトルネックを調査し、共用データ構造での高い並行性やスループットを実現する方法について説明しました。学習の早道はエキスパートの成果を考察すること、という場合がよくあります。そこで今回は、Doug Lea氏の`util.concurrent`パッケージにおける`ConcurrentHashMap`の実装について考察していくことにしましょう。新しいJava Memory Model(JMM)向けに最適化された`ConcurrentHashMap`のバージョンは、JSR 133で仕様を定められて、JDK 1.5の`java.util.concurrent`パッケージに含まれる予定です。`util.concurrent`のバージョンは、新旧のMemory Model環境下でスレッド・セーフに行われてきました。

スループットの最適化

`ConcurrentHashMap`は、ハイレベルな並行性を実現しロックを回避するためにいくつかの手段を用いています。その手段には、異なるハッシュ・バケットに対する複数書き込みのロックの使用や、JMMの不確実さを利用してロック時間を最小限に止めることや、ロックを完全に回避することなどがあります。そして`ConcurrentHashMap`は最も一般的な利用向けに最適化され、マップに既に存在しているはずの値を検索します。実際、ほとんどの正常な`get()`オペレーションはロックなしで実行するでしょう。(注意: 気軽な気持ちで試さないでください! JMMを操ることは見た目よりも困難で、簡単なことではありません。`util.concurrent`クラスは並行性のエキスパートによって書かれており、JMMの安全性に関して十分に検証されているのです。)

複数書き込みのロック

Hashtable(もしくはCollections.synchronizedMap)のスケーラビリティーにとっての主な障害は、マップ全体にわたって一つのロックを使用しており、そのロックが書き込み、削除、検索オペレーションの全体で保持され、時にはiteratorの検索のためにさえ保持されているためである、ということを思い出してください。これにより、たとえアイドル・プロセッサーが利用可能となっても、ロックが保持されている間は他のスレッドがマップにアクセスできなくなり、並行性が著しく制限されています。

ConcurrentHashMapの場合は、一つのマップ全体にわたるロックではなく、32のロックのコレクションを使用し、そのそれぞれがハッシュ・バケットの一部を保護しています。ロックは、主として変更のオペレーション(put()やremove())で使用されます。32のロックがあるということは、最高32のスレッドが一度にマップを修正することができるということです。これは、同時にマップへ書き込むスレッドが32未満の場合は、他の書き込みオペレーションはブロックをされないというわけではありません。つまり、32は書き込みにおける理論上の並行性の限界であるものの、必ずしも実際に実行できる数字ではありません。それでも、32は1よりもずっとよく、現代のコンピュータ・システムで起動するほとんどのアプリケーションには十分すぎると言ってよいでしょう。

マップ全体に及ぶオペレーション

32の別々なロックがあり、それぞれがハッシュ・バケットの一部を保護しているという事は、マップへ排他アクセスを要求するオペレーションは32のロックすべてを取得しなければならないということです。size()やisEmpty()のようなマップ全体に及ぶオペレーションでも、(これらのオペレーションの意味合いを適切に限定することで)1度にマップ全体をロックすることなく終了することができるかもしれません。しかし、(マップが大きくなるにつれて、ハッシュ・バケットの数を拡張して要素を再分配する) map rehashingのようなオペレーションでは、必ず排他アクセスを利用しなければなりません。Java言語には、サイズがいろいろ異なるロックを簡単に取得する方法は提供されていません。これを行うべき稀少なケースでは、再帰呼出で取得を行います。

JMMの概念

put()、get()、remove()の実装の話へ進む前に、JMMを簡単に見ておきましょう。JMMは1つのスレッドによるメモリーへのアクション(読み取り、書き取り)が、他のスレッドによるメモリーへのアクションにどう影響するかを管理しています。レジスターやプロセッサーごとのキャッシュを使用すれば、メモリ・アクセス速度を上げることができるというパフォーマンスの利点のために、Java言語仕様(JLS)では、いくつかのメモリー操作は他のスレッドからは直接見えないという状況を許可しています。スレッドにまたがるメモリー操作の整合性を保証するためには、synchronizedおよびvolatileという2つの言語メカニズムがあります。

JLSでは「明示的な同期化がないと、実装はメイン・メモリーを自由に更新することができ、驚くような仕方で行われ得る。」と定義しています。つまり、同期化がなければ、あるスレッドにある順で生じる書き込みが、異なるスレッドでは異なる順で生じるように見えるかもしれず、またメモリー変数を更新する書き込みを、あるスレッドから別のスレッドへ伝達するに要する時間は特定できない、ということです。

同期化を使用する最も一般的な理由は、コード内の重要なセクションへのアトミック・アクセスを保証することですが、実際、同期化はアトミシティ、可視性、順序という3つの機能を提供し

ています。アトミシティは非常に簡単です。つまり同期化は、モニターによって保護されているコード・ブロックが1つ以上のスレッドを同時実行しないようにすることで、再入可能な相互排除を実行しています。残念なことに、JLSの説明の大部分は、他の側面を無視して同期化のアトミシティに焦点をあてています。しかし、同期化はJMMにおいても重要な役割を果たしており、モニターを取得・解放する際、JVMにメモリーバリアを実行させるのです。

スレッドがモニターを取得するときに、読み込みバリアが実行されます。つまり、(on-processor cacheまたはプロセッサー・レジスターprocessor registerのような)スレッド・ローカルメモリーにキャッシュされた変数を無効にして、メイン・メモリーから同期化ブロックで使用される変数をプロセッサーに再読み込みさせます。同様に、モニターの解放に際しては、メイン・メモリーに修正された変数を消去して、スレッドは書き込みバリアを実行します。相互排除とメモリーバリアの組合せにより、プログラムが正確な同期化規則(次に他のスレッドに読み込まれる可能性のある変数に書き込む時、もしくは、他のスレッドが前に書き込んだ可能性のある変数を読み込むときは同期化を行う)に従う限り、それぞれのスレッドは自分が使用するどんな共用変数の正確な値も知ることができるのです。

共用変数にアクセスする場合に同期化を行わないと、非常に奇妙なことが起こります。ある変更がスレッド全体に瞬時に反映されてしまう一方で、ある変更はしばらく時間がかかったりするのです。メモリーのviewが整合性のあるものか（関連した変数が関連できていないかも知れません）、現在のviewをきちんと見ているのか（値によっては整合性がないかも知れません）確実でなくなってしまうのです。これらの危険要素を回避する一般的な(そしてお勧めの)方法は、もちろん適切に同期化を行うことです。しかし、`ConcurrentHashMap`のような非常に広く使用されているライブラリー・クラスなどのケースでは、優れたパフォーマンスを実現しようとすると特別の専門知識や努力が必要になってきます。

ConcurrentHashMapの実装

先に述べた通り、`ConcurrentHashMap`が使用するデータ構造は、（ハッシュ・バケットのサイズ変更可能な配列を保持する）`Hashtable`または`HashMap`のデータ構造の実装と似ており、それぞれはリスト1のように、`Map.Entry`要素の連鎖から成りたっています。一つのコレクション・ロックの代わりに、`ConcurrentHashMap`はバケットのコレクション上のパーティションを形成するロックのプールを使用しています。

リスト1. ConcurrentHashMapが使用するMap.Entry要素

```
protected static class Entry implements Map.Entry {
    protected final Object key;
    protected volatile Object value;
    protected final int hash;
    protected final Entry next;
    ...
}
```

ロックせずにデータ構造を調べる

`Hashtable`あるいは代表的なロックのプールでのマップ実装と違い、`ConcurrentHashMap.get()`オペレーションは必ずしも対象のバケットに関連したロックを取得する必要はありません。ロックがない場合は、実装は、list head pointerや(各ハッシュ・バケットのエントリーのリンクリストか

ら成るリンク・ポインターなどの)Map.Entry要素のフィールドといった、使用する変数の不整合な値に対処する準備ができていなければなりません。

ほとんどの並行クラスは、データ構造への排他アクセス、および整合性のあるviewを保証するために同期化を使用しています。排他性と整合性を期待する代わりに、ConcurrentHashMapが使用するリンクリストは、リストのviewが不整合であることを実装が検知できるように注意深く設計されています。そのviewの不整合を検知した場合や、捜しているエントリーが見つからない場合は、適切なバケット・ロックで同期化を行い再び連鎖を探索します。これは、ほとんどの検索が成功し、検索の回数が挿入と削除を上回る、というような一般的なケースの探索を最適化します。これによって一般的な場合の検索、つまりほとんどの検索は成功し、検索の回数が挿入と削除を上回る場合、が最適化されるのです。

不変性の利用

エントリー要素を不変にすることで、不整合の大きな要因を回避できます。volatileである値フィールドを除いたすべてのフィールドをfinalにするのです。これにより、ハッシュ連鎖の中間もしくは最後に要因の追加削除をすることができなくなります。つまり、要素を追加できるのは先頭のみになり、削除するには連鎖の全てまたは一部をクローン化したり、list head pointerを更新したりすることになります。したがって、ハッシュ連鎖を参照すると、そのリストの先頭への参照があるかどうかは分からなくても、リストの残り部分の構造は不変である事は分かるのです。また、値フィールドはvolatileであるので、値フィールドへの更新を即座に把握することができ、潜在的に不整合なメモリーのviewに対処できるマップ実装を書くというプロセスを大幅に単純化できるのです。

新しいJMMはfinal変数用に初期化の安全性を用意していますが、古いJMMは用意していません。これはつまり、別スレッドがfinalフィールドで見るのは、そのオブジェクトのコンストラクターが置いた値ではなく、デフォルト値である可能性があることを意味します。実装もこれを検出するように作られている必要があるのですが、実際そうなっており、Entryの各フィールドのデフォルトは有効な値ではないようになっています。Entryフィールドのどれかがデフォルト値（ゼロまたはnull）を持っているようならば検索は失敗し、get()実装を連鎖に同期させ、連鎖を全部調べさせるようにして、リストが構成されます。

検索オペレーション

検索オペレーションはまず希望のバケット(ロックなしで行われたため、不整合かもしれません)のヘッド・ポインターを見つけ、そのバケットのロックを取得せずにバケット連鎖を調べることで進められます。捜している値を見つけられない場合は、リスト2のように、同期化して再びエントリーを見つけようとします。

リスト2. ConcurrentHashMap.get ()実装

```
public Object get(Object key) {
  int hash = hash(key);     // throws null pointer exception if key is null
  // Try first without locking...
  Entry[] tab = table;
  int index = hash & (tab.length - 1);
  Entry first = tab[index];
  Entry e;
  for (e = first; e != null; e = e.next) {
```

```
    if (e.hash == hash & eq(key, e.key)) {
      Object value = e.value;
      // null values means that the element has been removed
      if (value != null) return value;
      else
        break;
    }
  }
  // Recheck under synch if key apparently not there or interference
  Segment seg = segments[hash & SEGMENT_MASK];
  synchronized(seg) { tab = table;
    index = hash & (tab.length - 1);
    Entry newFirst = tab[index];
    if (e != null || first != newFirst) {
      for (e = newFirst; e != null; e = e.next) {
        if (e.hash == hash & eq(key, e.key)) return e.value;
      }
    }
    return null;
  }
}
```

削除オペレーション

スレッドはハッシュ連鎖のリンク・ポインターに対する不整合な値を見る場合もあるので、単純に要素を連鎖から削除するだけでは、他のスレッドが探索を行う際に、その削除した値を絶対見る事はないとは言い切れません。実はリスト3のように、削除は2ステップのプロセスで行われます。まず、適切な`Entry`オブジェクトが見つかり、その値フィールドに`null`がセットされます。そして、先頭から、削除された要素までの連鎖部分がクローン化され、削除された要素以降の連鎖の残りにつながれます。値フィールドはvolatileなので、別のスレッドが削除された要素を捜して不整合な連鎖を検索すると、`null`値フィールドをすぐに見つけて、同期化を行いながら再検索を試みるでしょう。結局、元々のハッシュ連鎖のうち、削除される要素となる部分は、後でガーベジ・コレクションされることになります。

リスト3. ConcurrentHashMap.remove()実装

```
protected Object remove(Object key, Object value) {
  /*
    Find the entry, then 1. Set value field to null, to force get() to retry
      2. Rebuild the list without this entry.
         All entries following removed node can stay in list, but
         all preceding ones need to be cloned.  Traversals rely
         on this strategy to ensure that elements will not be
        repeated during iteration.
  */
  int hash = hash(key);
  Segment seg = segments[hash & SEGMENT_MASK];
  synchronized(seg) {
    Entry[] tab = table;
    int index = hash & (tab.length-1);
    Entry first = tab[index];
    Entry e = first;
    for (;;) {
      if (e == null)
        return null;
      if (e.hash == hash & eq(key, e.key)) break;
      e = e.next;
    }
    Object oldValue = e.value;
    if (value != null & !value.equals(oldValue))
      return null;
```

```
    e.value = null;
     Entry head = e.next;
     for (Entry p = first; p != e; p = p.next) head = new Entry(p.hash, p.key, p.value, head);
     tab[index] = head;
     seg.count--;
     return oldValue;
   }
 }
```

図1は、要素が削除される前のハッシュ連鎖です。

図1.ハッシュ連鎖

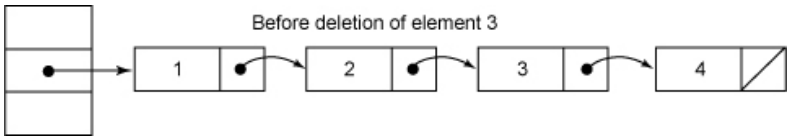
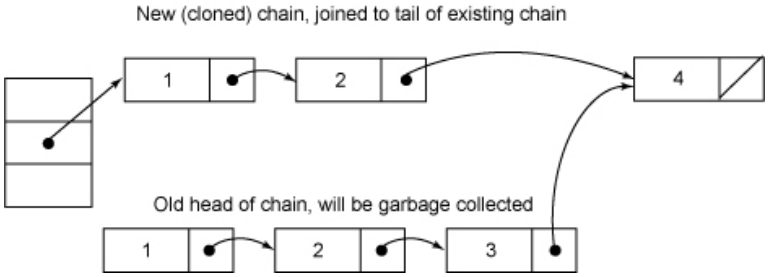


図2は要素3が削除された連鎖です。

図2. 要素の削除



挿入と更新のオペレーション

`put()`の実装は簡単です。`remove()`と同様、`put()`はその実行の間バケット・ロックを保持しますが、`get()`は必ずしもロックを取得する必要がないので、readerの実行を必ずしもを妨害しません(また、他のwriterが他のバケットにアクセスするのも妨害しません)。`put()`はまず、求めるキーを探して適切なハッシュ連鎖を探索します。キーが見つかれば、(volatileである)値フィールドは単純に更新されます。キーが見つからない場合は新しい`Entry`オブジェクトが作成され、新しいマッピングが記述されて、そのバケット用のリストのヘッドに挿入されることになります。

弱い整合性のiterator

`ConcurrentHashMap`が返すiteratorのセマンティックは`java.util`コレクションのものとは異なります。Fail-fast (iteratorが使用されている間に基本的なコレクションが修正されると、例外をスルーする)となるのではなく、弱い整合性を持つのです。ユーザーがハッシュ・キーのセットのiteratorを検索するために、`keySet().iterator()`を呼びだすと、実装は各連鎖のヘッド・ポインターが確実に最新になるように簡単な同期化を行います。`next()`や`hasNext()`オペレーションは明確な定義、つまり各ハッシュ連鎖を調べてから次の連鎖に移り、これを全ての連鎖を調べ終わるまで続ける、という定義がされています。弱い整合性を持つiteratorでは、繰り返し期間中の挿入は必ずしも反映しませんが、iteratorがまだ操作していないキーに対する更新あるいは削除は明確に反映しており、一度以上値を返す事はありません。`ConcurrentHashMap`が返すiteratorは`ConcurrentModificationException`をスローしません。

動的なサイズ変更

マップの要素数が増大するとともに、ハッシュ連鎖は長くなり、検索時間は増加します。ある時点で、バケットの数を増加させて、値を再ハッシュするのが賢明です。`Hashtable`のようなクラスでは、マップ全体で排他ロックを保持することが可能なので、これは簡単です。`ConcurrentHashMap`では、連鎖の長さがある限界を越えると、エントリーが挿入されるごとに、連鎖にはサイズ変更が必要だという記号が付けられます。。サイズ変更の必要な連鎖が十分な数に達すると、`ConcurrentHashMap`は再帰的手段を使って各バケットのロックを取得し、各バケットの要素を再ハッシュして新しい、より大きなハッシュ表にします。通常、これは自動的に、呼び出し側には透過的に行われます。

ロックなし？

`Entry`の`value`フィールドはvolatileであり、これが更新や削除の検出に使用されるので、正常な`get()`オペレーションはロックなしでよいと言うのは、少し言い過ぎです。マシン・レベルでは、volatileとsynchronizationは、同じキャッシュ整合性のプリミティブであるとしばしば解釈されます。ですから、ずっとキメが細かく、スケジューリングや、モニターを取得解放するJVMのオーバーヘッドはないのですが、ここには実質的に一定のロッキングは行われているのです。しかし、セマンティックは別として、(検索が挿入や削除よりも多い)多くの一般的な状況で`ConcurrentHashMap`が実現する並行性には圧倒されます。

結論

`ConcurrentHashMap`は、多くの並行アプリケーションで非常に役に立つクラスであると同時に、JMMの微妙な差異を理解して活用し、より高いパフォーマンスを実現するクラスの良い例です。`ConcurrentHashMap`は、コーディングの成果として素晴らしいものであり、並行性およびJMMについての深い理解無しにはできるものではありません。`ConcurrentHashMap`を使用して、習得して、楽しんでください。ただし、もしあなたがJavaの並行性エキスパートでないならば、一人で試すのは止めた方が良いでしょう。

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。