Javaの理論と実践: 並行コレクション・クラス

ConcurrentHashMapとCopyOnWriteArrayListにより、スレッド・セーフとスケーラビリティーの改良が得られます

Brian Goetz
Principal Consultant
Quiotix

2003年 7月 23日

数多くの他の便利な並行性ビルディング・ブロックに加えて、Doug Lea氏のutil.concurrentパッケージには、コレクション・タイプのListやMapを利用した高機能でスレッド・セーフな処理系が含まれていますが、今回、Brian Goetz氏は、HashtableやsynchronizedMapをConcurrentHashMapに変えるだけで、並行プログラムはどれだけ恩恵を得ることができるかについて説明しています。

このシリーズの他の記事を見る

Javaクラス・ライブラリーの最初の結合的コレクション・クラスは、JDK 1.0の機能であるHashtableでした。Hashtableは使いやすくてスレッド・セーフな結合的マップ機能を保持しており、確かに便利なものでした。しかし、Hashtableのスレッド・セーフ機能はすべてのメソッドが同期化されるという点で、かなり不便なものでした。よってJDK1.0での競合のない同期化には、かなりのパフォーマンス・コストがかかっていました。Hashtableの後継であるHashMapはJDK 1.2のコレクション・フレームワークの機能として登場し、非同期の基底クラスおよび同期化ラッパーであるCollections.synchronizedMapで、スレッド・セーフに対処しました。つまり、スレッド・セーフなCollections.synchronizedMapから基本機能を分割することで、同期化が必要なユーザーは機能を有し、同期化が必要ではないユーザーは機能を有さないようにすることができるようになりました。

HashtableやsynchronizedMap(Hashtableあるいは同期化Mapラッパーオブジェクトの各メソッドを同期化します)による同期化のアプローチは、2つの重要な問題を抱えています。1つ目は、1度にハッシュ・テーブルにアクセスすることができるスレッドは1つだけであるというスケーラビリティーの障害です。2つ目は、多くのcommon複合オペレーションが追加の同期を要求するという点で、真のスレッド・セーフを提供しているとは言えないということです。get()やput()のようなシンプルなオペレーションは追加同期なしでも支障はないものの、イテレーションや、put-if-absentというようなオペレーションの共通シーケンスは、データ競合の回避を外部同期に要求します。

© Copyright IBM Corporation 2003

条件付きスレッド・セーフ

同期化コレクション・ラッパーのsynchronizedMapとsynchronizedListは、条件付きスレッド・セーフ と呼ばれることがあります。個別の操作はすべてスレッド・セーフであるものの、制御フローが前のオペレーションの結果に依存するようなオペレーション・シーケンスの場合は、データ競合が起こる可能性があるためです。リスト１の前半は、よくある「put-if-absent 技法」であり、エントリーがMapに存在しない場合は追加するというものですが、あいにく、リスト1に書かれているcontainsKey()のメソッドの戻りとput()メソッドが呼ばれるまでの間に別のスレッドが同じkeyに値を設定することが可能です。書き込みを一度だけにしたければ、Map mで同期化する同期化ブロックでステートメントをラップする必要があります。

リスト1では、イテレーションも扱われています。最初の例では、別のスレッドがリストからアイテムを削除することができたので、ループの実行中にList.size()の結果は無効になる可能性がありました。タイミング悪くループの最後のイテレーションを入力した直後に、アイテムが別のスレッドによって削除された場合、List.get()はnullを返し、その結果doSomething()はNullPointerExceptionを投げることになるでしょう。これを回避するためには何ができるでしょう？もしListを使って繰返し処理をしている間に、別のスレッドがListにアクセスする可能性があるとしたら、同期化ブロックでListをラップしList lを同期化して、繰返しの間はList全体をロックしなければなりません。これによりデータ競合を解決することはできますが、繰返しの間List全体をロックすると、他のスレッドは長い間Listにアクセスするすることができなくなり、並行性に深刻な影響を与えることになってしまいます。

コレクション・フレームワークはリストあるいは他のコレクションの検索に#####を導入しました。これにより、コレクションの要素を通じて繰返し処理を最大限に利用することができます。しかし、java.utilのCollectionsクラスで実装された#####はfail-fastであるので、あるスレッドがIteratorを通じて検索している間に別のスレッドがコレクションを変更すれば、この後のIterator.hasNext()もしくはIterator.next()呼出しはConcurrentModificationExceptionを投げるということになります。上の例のように、ConcurrentModificationExceptionを回避したければ、List lで同期化する同期化ブロックでList全体をラップし、繰返しの間List全体をロックしなければなりません。(または、同期化を必要としない配列のList.toArray()やiterateを起動することもできますが、リストが大きい場合はパフォーマンス・コストがかかってしまいます。)

リスト1. synchronized mapのよくある競合状態

```
Map m = Collections.synchronizedMap(new HashMap());
List l = Collections.synchronizedList(new ArrayList());
// put-if-absent idiom -- contains a race condition
// may require external synchronization
if (!map.containsKey(key))
  map.put(key, value);
// ad-hoc iteration -- contains race conditions
// may require external synchronization
for (int i=0; i<list.size(); i++) {
  doSomething(list.get(i));
}
// normal iteration -- can throw ConcurrentModificationException
// may require external synchronization
for (Iterator i=list.iterator(); i.hasNext(); ) {
  doSomething(i.next());
}
```

信頼に関する誤った意識

synchronizedListとsynchronizedMapが提供している条件付きスレッド・セーフには、隠された危険性があります。開発者はこれらのコレクションは同期化しており、完全にスレッド・セーフであると考え、複合的なオペレーションを適切に同期化させることを怠たるようになるのです。その結果、これらのプログラムは簡単に機能しているように見えて、実際は高い負荷により、NullPointerExceptionやConcurrentModificationExceptionを投げることになるかもしれないのです。

スケーラビリティーの問題

スケーラビリティーとは、アプリケーションのワークロードや利用できるコンピュータ・リソースの増加に応じて、スループットがどのようになるかということです。スケーラブルなプログラムは、プロセッサー、メモリー、I/O帯域幅に比例して、より大きなワークロードを扱うことができます。排他アクセス用の共有リソースをロックすることは、スケーラビリティーのボトルネックです。というのも、たとえアイドル・プロセッサーをスレッド使用に組み入れることができたとしても、その他のスレッドはリソースにアクセスすることができないからです。スケーラビリティーを実現するためには、排他的なリソース・ロックへの依存を排除するか縮小しなければなりません。

同期化コレクション・ラッパーに関する重要な問題は、HashtableやVectorクラスが一回のロックで同期化するということです。つまり、1度に1つのスレッドだけがコレクションにアクセスでき、1つのスレッドがMapから読みこむ途中である場合、読み込みや書き込みを行いたい他のすべてのスレッドは待機しなければなりません。最も一般的なMapオペレーションのget()とput()は、明白となっている処理よりも多くの処理をしている可能性があります。それは特定のキーを見つけるためにハッシュ・バケットを検索する際、get()は多くの候補でObject.equals()を呼び出さなければならないかもしれないためです。keyクラスに使用されるhashCode()関数が値をハッシュ範囲に均一に広げなかったり、ハッシュ衝突がしばしば起こる場合、あるバケットの連鎖は他のバケットより長くなるかもしれません。その結果、長いハッシュ連鎖の検索や、何%かの要素はequals()呼び出しが遅くなる可能性があります。これらの条件下でのget()とput()に起こりうる問題は、アクセスが単に遅くなるというだけではなく、ハッシュ連鎖が検索されている間は他のすべてのスレッドはMapをアクセスできなくなるということです。

get()の実行にかなりの時間が必要となるケースがあるという事実は、上で説明した条件付きスレッド・セーフ問題において顕著です。リスト1で説明した競合状況では、1回のオペレーションの実行より長い間1つのコレクションをロックしていなければなりません。もし全てのイテレーションの間コレクションをロックするならば、他のスレッドはコレクションのロックを長い間待っていなければならない可能性があります。

シンプルなキャッシュの例

サーバー・アプリケーションにおけるMapを使用する最も一般的なアプリケーションの1つは、キャッシュの実装です。サーバー・アプリケーションはファイルコンテンツ、生成されたページ、データベース・クエリーの結果、解析されたXMLファイルに関連したDOMツリー、および他の多くのデータ型をキャッシュしています。キャッシュの主な目的は、サービス時間を縮小し、以前の計算結果を再使用することにより、スループットを増加させることです。キャッシュ・

ワークロードの典型的な特徴は、検索が更新よりもはるかに一般的であるということです、したがって、キャッシュは素晴らしいget()パフォーマンスを提供しています。アプリケーションのパフォーマンスを低下させるキャッシュは最悪です。

キャッシュの実装にsynchronizedMapを使用することは、アプリケーションへ潜在的なスケーラビリティーのボトルネックを導入することになります。それは、Mapへ新しいkeyやvalueを設定したいスレッドだけでなく、Mapから値を検索しているスレッドも含めて、1度に１つのスレッドしかMapにアクセスすることができないためです。

ロックの粒度を小さくする

スレッド・セーフを提供しつつHashMapの並行性を改善するためのアプローチは、テーブル全体用の1つのロックはやめて、各ハッシュ・バケット用のロック(一般的には、それぞれのロックがいくつかのバケットを保護するロックのpool)を使用することです。これにより、複数のスレッドは、コレクション全体に渡る1つのロックを使用するのではなく、同時に異なるMapにアクセスすることができるようになります。このアプローチを使えば、簡単に設定、検索、削除オペレーションのスケーラビリティーを改善することができます。ただし、この並行性がうまくいかない機能しない場合もあります。たとえば、size()やisEmpty()のようなコレクション全体で機能するメソッドに関しては、1度に多くのロックが必要になったり、不正確な結果を返すリスクがあるために、実装しづらくなるのです。しかし、キャッシュを実装するような状況にはこのアプローチは非常に適しています。なぜなら、キャッシュでは検索と設定のオペレーションは頻繁に行われますが、size()やisEmpty()はそれほど頻繁に行わないためです。

ConcurrentHashMap

util.concurrentのConcurrentHashMapクラス(JDK 1.5のjava.util.concurrentパッケージで登場します)は、synchronizedMapよりはるかに素晴らしい並行性を提供しているMapのスレッド・セーフな処理系です。複数読み取りが常にほぼ同時に実行することができ、同時読み書きも通常ほぼ同時に実行することができ、複数書き込みも多くの場合同時に実行することができるのです(関連するConcurrentReaderHashMapクラスもまた、同じような複数読み取りという並行性を提供していますが、アクティブな書き込みに関しては並行ではありません)。ConcurrentHashMapは、検索オペレーションを最適化するように設計されています。実際、get()オペレーションは、通常ロックなしでうまくいきます。ただし、ロックのないスレッド・セーフティーには注意が必要で、Java Memory Modelの詳細についてよく理解していなければなりません。残りのutil.concurrentと同様に、ConcurrentHashMapの実装は正確さおよびスレッド・セーフティーに関して並行性のエキスパートに広く評価されてきました。ConcurrentHashMapの実装の詳細については、次回の記事で見ていくことにします。

ConcurrentHashMapは、呼び出し側との決め事を少し緩めることで高い並行性を得ることができます。検索オペレーションは、直近の設定オペレーションによって設定された値を返したり、同時進行中である設定オペレーションに追加された値を返したりする可能性があります(決して無意味な結果を返すわけではありません)。ConcurrentHashMap.iterator()によって返された#####は各要素を一度に返して、ConcurrentModificationExceptionを投げることはありませんが、#####が構築されたことで生じた設定あるいは削除に関しては反映するかもしれないし、反映しないかもしれません。コレクションの繰返しに際して、スレッド・セーフティーを提供するためにテーブル全体をロックする必要はありません(不可能でさえあります)。ConcurrentHashMapは、更新を防

ぐためにテーブル全体をロックする必要がないすべてのアプリケーションで`synchronizedMap`または`Hashtable`の代わりに使用することができます。

`ConcurrentHashMap`は、共有キャッシュのような様々な一般的な利用の有効性を失うことなく、上記の歩み寄りによって`Hashtable`よりもはるかに優れたスケーラビリティーを提供することができます。

どれ程よいのか？

表1は、`Hashtable`と`ConcurrentHashMap`のスケーラビリティーの違いについての大まかな見解です。それぞれの実行において、nスレッドは`Hashtable`または`ConcurrentHashMap`でランダムなキー値を検索するようなタイトなループを同時に実行しました。検索結果は`put()`オペレーションの実行では80%失敗し、`remove()`オペレーションの実行は1%成功しました。テストは、Linuxが起動しているデュアルプロセッサーXeonシステムで行なわれました。データは`ConcurrentHashMap`の1スレッドで正規化されて、10,000,000回繰返した実行時間をミリ秒で示しています。`ConcurrentHashMap`のパフォーマンスはスレッドが多くなってもスケーラブルなままであるのに対して、`Hashtable`のパフォーマンスはロック競合が発生して、すぐに値が悪化していることがお分かりになるでしょう。

このテストでのスレッド数は代表的なサーバー・アプリケーションと比較して、少なく見えるかもしれません。しかし、それぞれのスレッドはテーブルを繰り返しhitしているので、実際に若干のコンテキストがテーブルを使用するスレッドよりもはるかに多くの競合をシミュレートしています。

表1. Hashtable対ConcurrentHashMapのスケーラビリティー

Threads	ConcurrentHashMap	Hashtable
1	1.00	1.03
2	2.59	32.40
4	5.58	78.23
8	13.21	163.48
16	27.58	341.21
32	57.27	778.41

CopyOnWriteArrayList

`CopyOnWriteArrayList`クラスは、挿入や削除よりも圧倒的に検索が多い並行アプリケーションで`ArrayList`に代わるものとしてみなされます。`ArrayList`が、AWTやSwingアプリケーションあるいは一般的なJavaBeanクラスのように、リスナーのリストを格納するために使用される場合は、`CopyOnWriteArrayList`と全く同じです。(関連する`CopyOnWriteArraySet`は、`Set`インタフェースを実装するために`CopyOnWriteArrayList`を使用しています。)

リストが可変の状態で、複数のスレッドにアクセスされる可能性があるにも関わらず、リスナーのリストの格納に通常の`ArrayList`を使用するならば、繰返しの間全リストをロックするか、繰返しの前にリストのクローンを作らなければなりません。それらはいずれもかなり手間のかかる

ことです。CopyOnWriteArrayListはその代りに、変更のオペレーションを行なう場合は常にリストの新規コピーを作成します。また、CopyOnWriteArrayListの#####は、#####が構築された時点で必ずリストの状態を返して、ConcurrentModificationExceptionを投げません。#####が見るリストのコピーは変わらないので、繰返しの前にリストのクローンを作ったり、繰返しの間ロックする必要はありません。言いかえれば、CopyOnWriteArrayListは、不変配列への可変の参照を含んでいます。したがって、その参照が固定されている限り、ロックの必要なく不変のスレッド・セーフの利点を得ることができます。

要約

synchronized collectionsクラス、Hashtable、Vector、および同期化ラッパー・クラスのCollections.synchronizedMapとCollections.synchronizedListは、MapとListの基本的な条件付きスレッド・セーフの実装を提供しています。しかし、これらの使用は次のような要因により、高度な並行アプリケーションには適していません。１つはコレクション全体に渡る1つのロックはスケーラビリティーの障害であること、2つ目は繰返しの間ConcurrentModificationExceptionsを回避するためには、相当な時間コレクションをロックする必要がある、とい理由です。しかし、ConcurrentHashMapおよびCopyOnWriteArrayListの実装では、呼び出し側が若干の歩み寄りをすることでスレッド・セーフを維持しつつ高い並行性を提供することができます。ConcurrentHashMapおよびCopyOnWriteArrayListは、HashMapまたはArrayListを使用してきた全てにおいて必ずしも役立つとは限りませんが、特定のcommon状況を最適化するように設計されています。多くの並行アプリケーションはConcurrentHashMapおよびCopyOnWriteArrayListを使用することで、利益を得られるようになるのです。

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。

© Copyright IBM Corporation 2003
(www.ibm.com/legal/copytrade.shtml)
商標
(www.ibm.com/developerworks/jp/ibm/trademarks/)