今まで知らなかった 5 つの事項: java.util.concurrent 第 1 回

並行コレクションによるマルチスレッド・プログラミング

Ted Neward
Principal
Neward & Associates

Alex Theedom
Senior Java developer
Consultant

2017年 8月 31日
(初版 2010年 5月 18日)

優れたパフォーマンスを実現し、なおかつアプリケーションが破損しないように動作するマルチスレッド・コードを作成する作業は、ともかく非常に困難です。そこで java.util.concurrent が登場します。この記事では著者の Ted Neward が、CopyOnWriteArrayList、BlockingQueue、ConcurrentMap などの並行コレクション・クラスで標準的なコレクション・クラスを置き換え、並行プログラミングの要求に応える方法について説明します。

このシリーズの他の記事を見る

並行コレクションは Java に追加された非常に重要な機能ですが、多くの Java 開発者は、この並列コレクションのパッケージによって解決しようとする問題が複雑であるのと同様に、このパッケージ自体も複雑であるに違いないと考えているため、このパッケージを使おうとしていません。

実際には、`java.util.concurrent` には数多くのクラスが含まれており、これらのクラスを利用すれば苦労せずに並行処理の一般的な問題を効果的に解決することができます。この記事では、CopyOnWriteArrayList や `BlockingQueue` といった `java.util.concurrent` のクラスを使用することで、どのようにマルチスレッド・プログラミングの難題を解決するのかを説明します。

1. TimeUnit

この連載について

皆さんは自分が Java プログラミングについて知っていると思うかもしれません。しかし実際には、ほとんどの開発者は Java プラットフォームの表面的な部分しか扱っておらず、当面の作業を完了するために十分なことしか学んでいません。この連載では、Ted Neward が Java

プラットフォームのコア機能を深く掘り下げ、非常に厄介な Java プログラミングの難題の解決にも役立つ、ほとんど知られていない事実を紹介します。

いわゆるコレクション・クラスではありませんが、`java.util.concurrent.TimeUnit` による列挙を使うと、コードが非常に読みやすくなります。`TimeUnit` を使うことで、メソッドや API を使ってミリ秒単位の操作を行う大変さから解放されます。

`TimeUnit` は `MILLISECONDS` や `MICROSECONDS` から `DAYS` や `HOURS` に至るまで、あらゆる時間単位に対応することができます。これはつまり、必要な時間間隔のほとんど全種類を `TimeUnit` によって処理できるということです。しかも列挙に対して変換メソッドを宣言できるため、短い時間単位にする必要がある場合には `HOURS` を簡単に `MILLISECONDS` に変換することができます。

2. CopyOnWriteArrayList

配列をまったく新規にコピーする操作は、通常使用するには時間やメモリーがあまりにも余分に必要となる、コストのかかる操作です。開発者は多くの場合、代わりに `ArrayList` を同期型で使おうとします。しかし同期型の `ArrayList` も時間とメモリーを必要とし、コストがかかります。というのも、コレクションの内容全体に対して繰り返し処理を行うたびに、読み書きを含むすべての操作を同期させ、一貫性を保証する必要があるからです。

そのため、`ArrayList` の読み取りは大量に行われるけれども、`ArrayList` の変更はほとんど行われないような状況では、`ArrayList` を使う方が配列を新規にコピーするよりもコストが高くなってしまいます。

この問題を解決してくれる貴重なツールが `CopyOnWriteArrayList` です。Javadoc によると、`CopyOnWriteArrayList` は、「配列に対して変更を行うすべての操作 (add、set など) を、配列を新規コピーすることで実装しているスレッド・セーフな `ArrayList` である」と定義されています。

何らかの操作でコレクションが変更されると、そのコレクションの内容は内部処理によって新しい配列にコピーされます。そのため、その配列の内容に読み取りアクセスする際に同期のコストは発生しません (変更される可能性のあるデータを読み取るわけではないため)。

つまり `CopyOnWriteArrayList` は、`ArrayList` には不向きな状況 (JavaBean イベントに対する `Listener` など、頻繁に読み取られ、稀にしか書き込まれないコレクション) で使うには理想的なのです。

3. BlockingQueue

`BlockingQueue` インターフェースは、`Queue` であることを示す名前になっています。つまり BlockingQueue インターフェースの各項目は先入れ先出し (FIFO) の順序で保存されます。特定の順序で挿入された項目は挿入時と同じ順序で取り出されます。さらに、空のキューから項目を取得しようとすると、その項目を取得できるようになるまで、呼び出し側スレッドがブロックされることも保証されます。同様に、キューが一杯の場合に項目を挿入しようとすると、そのキューのストレージに空きができるまで呼び出し側スレッドはブロックされます。

`BlockingQueue` を利用すると、あるスレッドが収集した項目を、どのようにして別のスレッドに渡して処理させるかという問題をスマートに解決することができ、同期の問題を特に気にする必

要がありません。Java Tutorial の Guarded Blocks のトレールは、その好例です。このトレールでは、手動による同期操作と `wait()`/`notifyAll()` とを使って単一スロット境界のバッファーを作成し、新しい項目が利用可能になると、また新しい項目をスロットに追加できる状態になると、スレッド間でシグナルがやりとりされます。(詳細は Guarded Blocks の実装の説明を参照してください。)

Guarded Blocks チュートリアルのコードは確かに動作しますが、長くて複雑であり、必ずしも直感的ではありません。Java プラットフォームの初期の頃には、確かに Java 開発者はこうしたコードと格闘する必要がありました。しかし、本当に事態は改善されたのでしょうか。

リスト 1 は Guarded Blocks のコードを書き直したバージョンであり、ここでは Drop を手動で作成する代わりに `ArrayBlockingQueue` を使っています。

リスト 1. BlockingQueue

```
import java.util.*;
import java.util.concurrent.*;

class Producer
    implements Runnable
{
    private BlockingQueue<String> drop;
    List<String> messages = Arrays.asList(
        "Mares eat oats",
        "Does eat oats",
        "Little lambs eat ivy",
        "Wouldn't you eat ivy too?");

    public Producer(BlockingQueue<String> d) { this.drop = d; }

    public void run()
    {
        try
        {
            for (String s : messages)
                drop.put(s);
            drop.put("DONE");
        }
        catch (InterruptedException intEx)
        {
            System.out.println("Interrupted! " +
                "Last one out, turn out the lights!");
        }
    }
}

class Consumer
    implements Runnable
{
    private BlockingQueue<String> drop;
    public Consumer(BlockingQueue<String> d) { this.drop = d; }

    public void run()
    {
        try
        {
            String msg = null;
            while (!((msg = drop.take()).equals("DONE")))
                System.out.println(msg);
        }
        catch (InterruptedException intEx)
```

```
        {
            System.out.println("Interrupted! " +
                "Last one out, turn out the lights!");
        }
    }
}

public class ABQApp
{
    public static void main(String[] args)
    {
        BlockingQueue<String> drop = new ArrayBlockingQueue(1, true);
        (new Thread(new Producer(drop))).start();
        (new Thread(new Consumer(drop))).start();
    }
}
```

また ArrayBlockingQueue は「公平さ」も重視します。つまり ArrayBlockingQueue を使う場合にはリーダー・スレッドもライター・スレッドも先入れ先出しでキューにアクセスすることができます。公平さを重視せず、もっと効率的なポリシーを使えるかもしれません。しかしそうすると、一部のスレッドがなかなか実行されない危険性があります。(つまり、他のリーダー・スレッドがロックを保持している間もリーダー・スレッドを実行できるようにした方が効率的かもしれませんが、そうするとリーダー・スレッドが連続的に読み取りを行い、ライター・スレッドが書き込みを実行できないリスクが発生します。)

バグに注意

ところで、Guarded Blocks に大きなバグがあることに気付いた人がいるかもしれませんが、まさにそのとおりです。もし、main() の中で Drop インスタンスに対して同期化したら一体どうなるでしょう。

また BlockingQueue は、時間を引数に取るメソッドもサポートしています。この引数により、そのスレッドがどの程度の期間ブロックされたら、対象項目の挿入または取得の失敗の通知に戻るかを指定します。ブロック期間を指定することで、待ち時間が無限になってしまうことがなくなります。待ち時間が無限になるとあまりにも簡単にシステムがハングアップし、リブートする必要が出てくるため、無限の待ち時間は本番システムにとって致命的です。

4. ConcurrentMap

Map には並行処理に関する小さなバグがあり、それを知らない多くの Java 開発者が途方に暮れる原因になっています。その問題を ConcurrentMap によって簡単に解決することができます。

1 つの Map が複数のスレッドからアクセスされる場合、containsKey() または get() を使って指定されたキーがあるかどうかを確認してからキーと値のペアを保存するのが普通です。しかし Map を同期化したとしても、キーの確認や保存処理の間にスレッドが入り込み、Map の制御を奪う可能性があります。問題は、ロックは get() の開始時点で取得され、次にそのロックが解放され、put() を呼び出す際に再度そのロックが取得される、という点にあります。その結果、競合条件が発生します。これは 2 つのスレッド間の競合であり、どのスレッドが最初に実行されるかによって結果は異なります。

2 つのスレッドがまったく同じ瞬間に 1 つのメソッドを呼び出すと、それぞれのスレッドがテストと put 操作を実行するため、そのプロセスの間に最初のスレッドの値が失われてしまいます。

幸いなことに、`ConcurrentMap` インターフェースには 1 つのロックで 2 つのことを実行するように設計されたメソッドがいくつか追加でサポートされています。例えば `putIfAbsent()` は最初にテストを実行し、キーが `Map` の中に保存されていない場合にのみ put 操作を行います。

5. SynchronousQueue

`SynchronousQueue` は興味深いクラスです。Javadoc には、SynchronousQueue について以下のような記述があります。

> SynchronousQueue はブロッキング・キューであり、各挿入操作は、その挿入操作に対応して別のスレッドが削除操作を行うまで待つ必要があり、その逆の削除操作の場合も、別のスレッドが挿入操作を行うまで待つ必要があります。同期キューには内部容量がなく、そもそも容量自体がゼロです。

要するに `SynchronousQueue` は、先ほど触れた `BlockingQueue` の別の実装なのです。`SynchronousQueue` を使うと極めて軽量な手段で、あるスレッドから別のスレッドに 1 つの要素を渡すことができ、そのために `ArrayBlockingQueue` で使われるブロッキングの動作を利用することができます。リスト 2 ではリスト 1 のコードを書き直し、ArrayBlockingQueue の代わりに `SynchronousQueue` を使っています。

リスト 2. SynchronousQueue

```
import java.util.*;
import java.util.concurrent.*;

class Producer
    implements Runnable
{
    private BlockingQueue<String> drop;
    List<String> messages = Arrays.asList(
        "Mares eat oats",
        "Does eat oats",
        "Little lambs eat ivy",
        "Wouldn't you eat ivy too?");

    public Producer(BlockingQueue<String> d) { this.drop = d; }

    public void run()
    {
        try
        {
            for (String s : messages)
                drop.put(s);
            drop.put("DONE");
        }
        catch (InterruptedException intEx)
        {
            System.out.println("Interrupted! " +
                "Last one out, turn out the lights!");
        }
    }
}

class Consumer
    implements Runnable
{
    private BlockingQueue<String> drop;
    public Consumer(BlockingQueue<String> d) { this.drop = d; }
```

```
    public void run()
    {
        try
        {
            String msg = null;
            while (!((msg = drop.take()).equals("DONE")))
                System.out.println(msg);
        }
        catch (InterruptedException intEx)
        {
            System.out.println("Interrupted! " +
                "Last one out, turn out the lights!");
        }
    }
}

public class SynQApp
{
    public static void main(String[] args)
    {
        BlockingQueue<String> drop = new SynchronousQueue<String>();
        (new Thread(new Producer(drop))).start();
        (new Thread(new Consumer(drop))).start();
    }
}
```

この実装コードはほとんど同じに見えますが、このアプリケーションにはさらなるメリットがあります。つまり `SynchronousQueue` のおかげで、キューに挿入された項目を利用しようと待機しているスレッドがない限りキューに挿入することはできません。

実際には、`SynchronousQueue` は Ada や CSP などの言語にある「ランデブー・チャネル」と似ています。ランデブー・チャネルは .NET など他の環境では「joins」と呼ばれることもあります (「関連トピック」を参照)。

まとめ

Java ランタイム・ライブラリーに手軽で作成済みの等価なものが用意されているにもかかわらず、なぜコレクション・クラスへの並行処理の導入に苦労する必要があるのでしょう。この連載の次回の記事では、`java.util.concurrent` 名前空間について、さらに説明します。

ダウンロード

内容	ファイル名	サイズ
Sample code for this article	j-5things4-src.zip	23KB

著者について

Ted Neward

Ted Neward has written over 250 articles and a dozen books across many different technologies, including .NET, iOS, Java, Android, and JavaScript. He resides in Seattle with his wife, two kids, nine laptops, fourteen mobile devices, and two cats. Email him if you're interested in having him or his company work with you.

Alex Theedom

May 2017