システム負荷を軽減したスレッド化: 常に共用が最善とは限らない

ThreadLocalを活用してスケーラビリティーを向上させる

Brian Goetz

2001年 10月 01日

Javaプラットフォームのバージョン1.2で`ThreadLocal`クラスが登場しましたが、あまり話題になりませんでした。スレッド・ローカル変数が、Posixの`pthreads`機能など、多くのスレッド化機能に組み込まれたのはずいぶん前のことですが、Java Threads APIの初期デザインにはこの便利な機能がありませんでした。その上、その後の最初の実装はかなり効率の悪いものでした。こうした理由から、`ThreadLocal`はあまり注目されていませんが、スレッド・セーフ並行プログラムの開発を単純化する場合には非常に便利です。システム負荷を軽減したスレッド化 の第3回では、Javaソフトウェア・コンサルタントのBrian Goetz氏が`ThreadLocal`を紹介し、その実力を活用するためのヒントをお教えします。

スレッド・セーフ・クラスの作成は簡単なことではありません。変数を読み取り/書き込みする場合の条件だけでなく、そのクラスを他のクラスがどのように使用するのかも慎重に分析することが要求されます。機能性や使いやすさ、パフォーマンスに悪影響を与えずに、クラスをスレッド・セーフにすることに、非常に苦労する場合もあります。中には、メソッド呼び出し間で状態情報を保持するクラスもあるので、このようなクラスをスレッド・セーフにするのは、どのような方法でも困難なことです。

クラスをスレッド・セーフにするよりも、非スレッド・セーフのクラスを使用した方が管理しやすいかもしれません。スレッド・セーフではないクラスは、マルチスレッド化プログラムにおいても、あるスレッドが使用するそのクラスのインスタンスを別のスレッドが使用することがない限り安全に使用できます。たとえば、JDBC`Connection`クラスはスレッド・セーフではありません。2つのスレッドは、細かい粒度レベルで`Connection`を安心して共用することはできませんが、各スレッドが独自の`Connection`を持っていれば、複数のスレッドが同時にデータベース操作を安全に実行することができます。

`ThreadLocal`を使用せずに、スレッドごとに別々のJDBC接続 (あるいは他のオブジェクト) を保持できることは確かです。Thread APIは、オブジェクトとスレッドとの関連付けに必要なツールをすべて含んでいます。しかし、`ThreadLocal`クラスを活用すれば、スレッドとそのスレッドあたりのデータとの関連付けは、より一層簡単になります。

スレッド・ローカル変数とは何か？

スレッド・ローカル変数 は、その変数を使用するスレッドごとに別々の値を効果的に提供します。各スレッドは、自分に関連する値しか認識できないので、他のスレッドが値を使用していたり変更していたりしても気付きません。コンパイラーの中には (Microsoft Visual C++ コンパイラーやIBM XL FORTRANコンパイラーなど)、ストレージ・クラス修飾子 (`static` や`volatile` など) を使用して、スレッド・ローカル変数のサポートを言語に組み込んでいるものもあります。Javaコンパイラーは、特に言語でスレッド・ローカル変数をサポートすることはありませんが、その代わりに、この変数は、`Thread` コア・クラスで特殊なサポートを提供する、`ThreadLocal` クラスで実装されます。

スレッド・ローカル変数は、Java言語そのものに組み込まれているわけではなく、クラスを介して実装されるため、スレッド・ローカル変数を使用する場合の構文は、スレッド・ローカル変数が組み込まれている言語の構文よりも少し扱いにくくなります。スレッド・ローカル変数を作成するには、クラス`ThreadLocal` のオブジェクトのインスタンスを生成します。`ThreadLocal` クラスの振る舞いは、`java.lang.ref` のさまざまな`Reference` クラスとほとんど同じです。値を保管したり検索したりする場合の間接ハンドルの役割を果たします。リスト1に`ThreadLocal` インターフェースを示します。

リスト1. ThreadLocalインターフェース

```
public class ThreadLocal { public Object get();
  public void set(Object newValue);
  public Object initialValue();
}
```

`get()` アクセサーは、変数の現行スレッドの値を取得します。`set()` アクセサーは、現行スレッドの値を変更します。`initialValue()` メソッドは、変数がまだこのスレッドで使用されていない場合に、その変数の初期値を設定できるオプション・メソッドです。このメソッドを使用すると、遅延初期化を考慮できます。`ThreadLocal` の振る舞いがよく分かるように、実装例を紹介します。リスト2に、`ThreadLocal` を実装する1つの方法を示します。これはおそらくパフォーマンスも良くなく、特に優れた実装というわけではありませんが (これは、初期の実装とよく似ています)、`ThreadLocal`がどのように振る舞うかは、的確に示されています。

リスト2. ThreadLocalの悪い実装

```
public class ThreadLocal { private Map values = Collections.synchronizedMap(new HashMap());
  public Object get() {
    Thread curThread = Thread.currentThread();
    Object o = values.get(curThread);
    if (o == null && !values.containsKey(curThread)) {
      o = initialValue();
      values.put(curThread, o);
    }
    return o;
  }
  public void set(Object newValue) {
    values.put(Thread.currentThread(), newValue);
  }
  public Object initialValue() {
    return null;
  }
}
```

この実装では、`get()` 操作と`set()` 操作ごとに`values` マップの同期化が必要であり、また複数のスレッドが同時に同じ`ThreadLocal`にアクセスすると競合が発生するため、パフォーマンスは良くありません。さらに、この実装では、`Thread` オブジェクトを`values` マップのキーとして使用することで、スレッドの終了後に`Thread`がガーベッジ・コレクションされないようになっており、終了したスレッドの`ThreadLocal`のスレッド固有の値もガーベッジ・コレクションされないため、実用的ではありません。

ThreadLocalを使用してスレッドごとのSingletonを実装する

スレッド・ローカル変数は一般に、`ThreadLocal` にスレッド・セーフでないオブジェクト全体をカプセル化するか、`ThreadLocal` にオブジェクトのスレッド固有の状態をカプセル化することで、ステートフル（変化する状態を持つこと）なSingletonや共用オブジェクトをスレッド・セーフにする場合に使用します。たとえば、データベースに密接に結合しているアプリケーションでは、多くのメソッドがデータベースにアクセスする必要があるので、アプリケーションのすべてのメソッドに`Connection` を引数として組み込むのは大変です。そこで、Singletonを使って接続にアクセスすれば、時間はかかりますが、著しく楽になります。複数のスレッドが1つのJDBC`Connection`を共用することは安全ではありません。そこで、リスト3に示すように、Singletonで`ThreadLocal` を使用すれば、プログラム内のどのクラスでも、スレッドごとの`Connection`のリファレンスを簡単に取得できるようになります。このように、`ThreadLocal` は、スレッドごとのSingleton を作成できるものと考えることができます。

リスト3. スレッドごとのSingletonにJDBC Connectionを格納する

```
public class ConnectionDispenser { private static class ThreadLocalConnection extends ThreadLocal {
    public Object initialValue() {
      return DriverManager.getConnection(ConfigurationSingleton.getDbUrl());
    }
  }
  private ThreadLocalConnection conn = new ThreadLocalConnection();
  public static Connection getConnection() {
    return (Connection) conn.get();
  }
}
```

JDBC`Connection` や正規表現マッチャーなど、使用するよりも作成する方が比較的にコストがかかるステートフル・オブジェクトやスレッド・セーフでないオブジェクトには、スレッドごとのSingleton方法の使用をお勧めします。もちろん、このような場合でも、プーリングなど、共用アクセスを安全に管理するための他の方法を使用することもできますが、プーリングにおいても、スケーラビリティーが低下する可能性があります。プールの実装は、プール・データ構造の整合性を保持するように同期化しなければならないため、すべてのスレッドが同じプールを使用している場合に、プールへのアクセス頻度が高いスレッドが多いとシステムで競合が発生するため、プログラムのパフォーマンスが影響を受ける可能性があります。

ThreadLocalを使用してデバッグ・ロギングを単純化する

プーリングが有効な代替策にならない`ThreadLocal` のその他の用途としては、後で検索できるようにスレッドごとのコンテキスト情報を格納したり集計したりすることなどがあります。たとえば、マルチスレッド化アプリケーションでデバッグ情報を管理する機能を作成するとしましょ

う。リスト4のDebugLogger クラスに示すように、デバッグ情報はスレッド・ローカル・コンテナーに集計することができます。ある作業単位の始めにコンテナーを空にし、エラーが発生したらコンテナーに問い合わせて、これまでにこの作業単位で生成されたデバッグ情報をすべて検索します。

リスト4. ThreadLocalを使用してスレッドごとのデバッグ・ログを管理する

```
public class DebugLogger {
  private static class ThreadLocalList extends ThreadLocal {
    public Object initialValue() {
      return new ArrayList();
    }
    public List getList() { return (List) super.get(); }
  }
  private ThreadLocalList list = new ThreadLocalList();
  private static String[] stringArray = new String[0];
  public void clear() {
    list.getList().clear();
  }
  public void put(String text) {
    list.getList().add(text);
  }
  public String[] get() {
    return list.getList().toArray(stringArray);
  }
}
```

コード内で、DebugLogger.put() を呼び出してプログラムの実行作業に関する情報を保存しておけば、後で必要なときに特定のスレッドに関するデバッグ情報 (エラーの発生時刻など)を簡単に取得することができます。この方法は、単にすべての情報をログ・ファイルに書き込んでから、どのログにどのスレッドのレコードが記録されているかを整理する(そして、スレッド間でログ・オブジェクトについて競合が発生するかどうかを心配する) よりも、はるかに便利で効率的です。

また、ThreadLocal は、ひとつの要求全体が作業単位であるサーブレット・ベースのアプリケーションや他のマルチスレッド化サーバー・アプリケーションでも役立ちます。なぜなら、そのような場合は要求の処理全体を通して使用するスレッドが1つだからです。ThreadLocal 変数は、前述のスレッドごとのSingleton方法を使用して、要求ごとのあらゆる種類のコンテキスト情報を格納する場合にも使用できます。

ThreadLocalのあまりスレッド・セーフではない親類、InheritableThreadLocal

ThreadLocal クラスには、親類のInheritableThreadLocal があります。機能はほとんど同じですが、全く違う種類のアプリケーションに適しています。スレッドを作成したときに、そのスレッドがInheritableThreadLocalオブジェクトの値を保持していると、これらの値は子プロセスに自動的に渡されます。子プロセスは、InheritableThreadLocal のget()を呼び出した場合、親プロセスと同じオブジェクトを見ることになります。オブジェクトは複数のスレッド間で共用されるので、スレッド・セーフという性質を保持するために、InheritableThreadLocal は不変オブジェクト (いったん作成したら状態が二度と変化しないオブジェクト)にのみ使用してください。InheritableThreadLocal は、親スレッドから子スレッドにユーザーIDやトランザクションIDなどのデータを渡す場合には有効ですが、JDBCConnection などのステートフル・オブジェクトには役に立ちません。

ThreadLocalのパフォーマンス

スレッド・ローカル変数の概念はかなり以前からあり、Posix`pthreads` 仕様など、多くのスレッド化フレームワークでサポートされていますが、スレッド・ローカルのサポートは、初期のJavaスレッド・デザインでは取り入れられず、Javaプラットフォームのバージョン1.2でようやく追加されました。多くの意味で、`ThreadLocal` はまだ発展途上にあります。バージョン1.3、そしてバージョン1.4でも、パフォーマンス問題を改善するために作り直されました。

JDK 1.2での`ThreadLocal` の実装は、`HashMap`ではなく同期化された`WeakHashMap` を使用して値を保管している点を除き、リスト2とほとんど同じです (`WeakHashMap` を使用すれば、`Thread` オブジェクトがガーベッジ・コレクションされない問題は解決されますがパフォーマンス・コストが増大します)。言うまでもなく、`ThreadLocal` のパフォーマンスはかなりお粗末でした。

Javaプラットフォームのバージョン1.3の`ThreadLocal` のバージョンは、実質的に改善されています。同期化を使用していないので、スケーラビリティーの問題も発生せず、また弱いリファレンスも使用していません。その代わりに、スレッド・ローカル変数をその現行スレッドでの値にマッピングする`HashMap` を保持するインスタンス変数を`Thread` に追加することで、`ThreadLocal` をサポートするように`Thread` クラスが変更されました。スレッド・ローカル変数を取得したり設定したりするプロセスでは、別のスレッドが読み取り/書き込みする可能性があるデータを読み取り/書き込みする必要がないので、同期化することなく`ThreadLocal.get()` と`set()` を実装することができます。また、スレッドごとの値へのリファレンスは、それを所有している`Thread` オブジェクトに保管されるため、`Thread` がガーベッジ・コレクションされると、そのスレッドごとの値もガーベッジ・コレクションすることができます。

残念ながら、このような改善がなされても、Java 1.3の`ThreadLocal` のパフォーマンスはまだ驚くほど遅いものです。2プロセッサーのLinuxシステム環境でのSun 1.3 JDKで個人的に行った実行ベンチマークによると、`ThreadLocal.get()` の操作は、競合のない同期の約2倍の時間がかかります。このようにパフォーマンスが低い理由は、`Thread.currentThread()` メソッドのコストが非常に高く、`ThreadLocal.get()` ランタイムの2/3以上に相当する点にあります。これらの弱点があっても、JDK 1.3の`ThreadLocal.get()` は、競合のある同期よりはかなり高速ですので、競合が発生する可能性がある場合 (スレッドの数が多い場合や、同期化ブロックを実行する頻度が高い場合、同期化ブロックが大きい場合など) は、`ThreadLocal` を使用する方が全体的に効率が良いでしょう。

Javaプラットフォームの最新バージョンであるバージョン1.4b2では、`ThreadLocal` と`Thread.currentThread()` のパフォーマンスが大幅に改善されました。これらの新たな改善により、`ThreadLocal` は、プーリングなどの他の方法よりも高速になったはずです。他の方法よりも簡単で、しかもエラーが発生する可能性が低いため、将来的にはスレッド間の不要な相互作用を避ける効率的な方法として認められるようになるでしょう。

ThreadLocalの長所

`ThreadLocal` には、いくつものメリットがあります。ステートフルなクラスをスレッド・セーフにしたり、マルチスレッド化環境で安全に使用できるようにスレッド・セーフでないクラスをカプセル化したりする一番簡単な方法です。`ThreadLocal` を使用すれば、スレッド・セーフな性質を実現するために同期化する時期を判別する煩わしさがなくなり、また同期化の必要がないのでスケーラビリティーが向上します。簡潔さに加え、`ThreadLocal` を使用して、スレッドごとの

Singletonやスレッドごとのコンテキスト情報を格納すれば、貴重なドキュメンテーション源となります。`ThreadLocal` を使用すれば、`ThreadLocal` に保管されているオブジェクトがスレッド間で共用されない ことは明らかなので、クラスがスレッド・セーフかどうかを判別する作業も簡単になります。

関連トピック

- このシリーズの第1回「同期化を敵視することはありません」(developerWorks、2001年7月)では、競合しない同期は、広く信じられているほどパフォーマンスへの影響がないことを解説しています。第2回「競合を低減させる」(developerWorks、2001年9月)では、競合する同期のプログラムのパフォーマンスへの影響を軽減するテクニックについて説明しています。
- Java Performance Tuning (Jack Shirazi著、O'Reilly & Associates社、2000年)には、Javaプラットフォーム におけるパフォーマンス上の問題を解決するための方法が示されています。
- Java Platform Performance: Strategies and Tactics (Steve Wilson、Jeff Kesselman著、Addison-Wesley社、2000年)には、迅速かつ効果的なJavaコードを構築するために必要となる、熟達したJavaプログラマーのための技法が示されています。
- Java Performance and Scalability, Volume 1: Server-Side Programming Techniques (Dov Bulka著、Addison-Wesley社、2000年)には、アプリケーションのパフォーマンスを向上させるヒントが数多く示されています。
- Brian Goetzの記事 "Double-checked locking: Clever, but broken"(JavaWorld、2001年2月)は、JMMの詳細の検討、および特定の状況下での同期化の失敗による驚くべき結果を示しています。
- Doug Lea著のConcurrent Programming in Java, Second Edition (Addison-Wesley社、1999年)は、Javaにおけるマルチスレッド・プログラミングに伴う難解な問題に関する優れた著作です。
- Alex Roetter氏は、同氏の記事「マルチスレッド化Javaアプリケーションの作成」(developerWorks、2001年2月)で、Java Thread APIを紹介し、マルチスレッド化に伴う問題を概説し、一般的な問題解決策を提示しています。
- 「接続プール」(developerWorks、2000年10月、Siva Visverwaran著)では、J2EE環境でのデータベース・リソースと非データベース・リソース両方の接続プーリングのサポートに焦点を当てています。
- IBM Thomas J. Watson Research Centerのperformance modeling and analysis team は、パフォーマンスおよびパフォーマンス管理の分野で幾つかのプロジェクトを研究しています。
- その他のJava参考文献に関しては、developerWorks Javaテクノロジー・ゾーン を参照してください。