リアルタイム Java での開発: 第 1 回 リアルタイム Java ならではの独特な機能を探る

独自のアプリケーションでリアルタイム Java のパフォーマンスを実現する

Sean C. Foley
Staff Software Developer
IBM

2009年 9月 01日

リアルタイム Java™ は、Java 言語でのプログラミングの容易さと、リアルタイム要件を満たさなければならないアプリケーションに必要なパフォーマンスとを組み合わせた、Java 言語の拡張機能一式です。これらの拡張機能は、従来の Java ランタイム環境には欠けていた、リアルタイム環境のための機能を提供します。3 回連載の第 1 回目となるこの記事では、これらの拡張機能をいくつか取り上げて説明し、さらに独自のアプリケーションに適用してリアルタイム・パフォーマンスを実現する方法を説明します。

このシリーズの他の記事を見る

リアルタイム Java とは、標準 Java 技術のパフォーマンスを超えたリアルタイム・パフォーマンスをアプリケーションにもたらす、Java 言語の拡張機能一式です。リアルタイム・パフォーマンスは従来のスループット・パフォーマンスとは異なります。スループット・パフォーマンスは通常、一定の期間に完了可能な命令やタスクの合計数、または作業量を測定したものですが、リアルタイム・パフォーマンスは、アプリケーションが外部からの刺激に応答しなければならない時間に焦点を当てたものであり、決められた時間制限を超えてはなりません。ハード・リアルタイム・システムの場合は、この時間要件を必ず満たさなければなりません。一方、ソフト・リアルタイム・システムでは時間制限に対する違反に比較的寛容です。リアルタイム・パフォーマンスを実現するには、アプリケーション自体がプロセッサーを制御して刺激に応答できること、そして刺激に対して応答している間、仮想マシン内のプロセスとの競合によってアプリケーション・コードの実行が阻止されないことが必要です。リアルタイム Java は、これまで Java アプリケーションでは満たすことのできなかった応答性を実現します。

リアルタイム JVM は、リアルタイム・オペレーティング・システム (RTOS) サービスを利用してハード・リアルタイム機能を実現しますが、課せられたリアルタイム要件がそれよりも緩いアプリケーションの場合には、通常の (RTOS でない) オペレーティング・システム上で稼働することもできます。リアルタイム Java で使用されている技術の一部は、リアルタイム JVM を使用するように切り替えることで、「無料」で使えるようになります。しかしリアルタイム Java の特定の機

能を利用するには、アプリケーションに変更を加える必要が出てきます。この記事では、これらの特定の機能に焦点を合わせます。

抑制しなければならないサブプロセス

JVM は、特定のアプリケーションが大まかにしか制御できない作業を行うことによって、そのアプリケーションにサービスを提供します。JVM 内部では以下を始めとするいくつかのランタイム・サブプロセスが動作しています。

- **ガーベッジ・コレクション**: アプリケーションが破棄したランタイム・メモリー・ブロックを回収する作業です。ガーベッジ・コレクションによって、アプリケーションの実行が、ある程度の時間遅延されることがあります。
- **クラス・ロード**: このプロセス (Java アプリケーションはクラスの粒度でロードされるため、クラス・ロードと呼ばれます) には、ファイルシステムまたはネットワークからのアプリケーションの構造、命令、その他リソースのロードが伴います。標準的な Java では、アプリケーションはクラスが初めて参照されたときに、そのクラスをロードします (遅延ロード)。
- **JIT (Just-In-Time) 動的コンパイル**: 多くの仮想マシンはアプリケーションの実行中に、メソッドの動的コンパイルにより、解釈された Java バイトコードからネイティブ・マシン命令にメソッドをコンパイルします。動的コンパイルによってパフォーマンスは向上しますが、コンパイル・アクティビティー自体が一時的な遅延を引き起こし、アプリケーション・コードの実行がブロックされることがあります。
- **スケジューリング**: 標準的な Java では、アプリケーション自体が実行するスレッドについてのスケジューリングにしても、同じオペレーティング・システム上で実行中の他のアプリケーションとの相対的なスケジューリングにしても、スケジュールを決定する最小限の権限しかアプリケーションには与えられません。

上記のサブプロセスのすべてが、外部からの刺激に対するアプリケーションの応答性を妨げることになります。これらのサブプロセスによって、アプリケーション・コードの実行が遅延される可能性があるためです。例えば、ネットワークやレーダー・システム、あるいはキーボードなどのデバイスからの信号に応答して一連の命令が実行されるようにスケジューリングされている場合もありますが、リアルタイム・アプリケーションでは、ガーベッジ・コレクションなどの関係のないプロセスが実行されることで生じる、一連の命令に対する応答の実行遅延としては、最小限の時間しか許容されません。

リアルタイム Java は、これらの内在するサブプロセスによるアプリケーションへの干渉を最小限に抑えることを目的とした各種の技術を提供します。リアルタイム JVM への切り替えに伴う「無料」の技術には、コレクションのための割り込みの期間および影響を制限するように特化されたガーベッジ・コレクション、最適化を遅延する代わりに起動時にパフォーマンスを最適化できるように特化されたクラス・ロード、特化されたロックおよび同期化、そして優先順位の逆転を防止するように特化されたスレッド・スケジュールなどがあります。ただし、RTSJ (Real-Time Specification for Java) で導入されている機能を利用するには、アプリケーションを変更する必要が出てきます。

RTSJ は、JVM 内部で数多くのリアルタイム機能を有効にする API を提供します。これらの機能には、仕様の実装で必須の機能もあれば、オプションの機能もあります。この仕様が対象範囲とする一般分野は以下のとおりです。

- リアルタイム・スケジューリング
- 高度なメモリー管理
- 高分解能タイマー
- 非同期イベント処理
- スレッドの非同期割り込み

`Realtime` スレッド

RTSJ で定義している `javax.realtime.RealtimeThread` は、標準 `java.lang.Thread` クラスのサブクラスです。`RealtimeThread` は単独で、この仕様の高度な機能のいくつかを可能にします。例えば、リアルタイム・スレッドはリアルタイム・スレッド・スケジューラーに従います。リアルタイム・スレッド・スケジューラーでは、このスケジューラーに特有のスケジューリング優先度の範囲を指定し、ファーストイン・ファーストアウト・リアルタイム・スケジューリング・ポリシー (最高の優先度を持つスレッドが割り込みなしで実行されることを確実にするポリシー)、ならびに優先度の継承 (優先度の高いスレッドが妨げられずに実行される上で必要なロックを、優先度の低いスレッドが無期限に保持すること (優先順位の逆転として知られる状態) を防ぐアルゴリズム) を実装することができます。

`RealtimeThread` のインスタンスはコード内に明示的に作成することができます。その一方で、大々的な開発作業とそれに関連するコストが発生するのを避けるために、アプリケーションの変更を最小限に抑えてリアルタイム・スレッド化を実現することも可能です。この記事ではリアルタイム・スレッド化を最小限の作業で最も簡単に実現するさまざまな方法の例を紹介します (記事のすべての例のソース・コードはダウンロードすることができます)。ここで紹介する手法では、最小限の作業でアプリケーションがリアルタイム・スレッドを利用できるようにするだけでなく、アプリケーションが標準仮想マシンでも変わらず実行できるようにします。

優先度に従ってスレッド・タイプを割り当てる

リスト 1 に、優先度の値を基にリアルタイム・スレッドまたは通常のスレッドを割り当てるコード・ブロックを記載します。このコードをリアルタイム仮想マシンで実行すると、一部のスレッドをリアルタイム・スレッドにすることができます。

リスト 1. 優先度に従ってスレッド・クラスを割り当てる

```
import javax.realtime.PriorityScheduler;
import javax.realtime.RealtimeThread;
import javax.realtime.Scheduler;

public class ThreadLogic implements Runnable {
    static void startThread(int priority) {
        Thread thread = ThreadAssigner.assignThread(
                priority, new ThreadLogic());
        thread.start();
    }

    public void run() {
        System.out.println("Running " + Thread.currentThread());
    }
}

class ThreadAssigner {
    static Thread assignThread(int priority, Runnable runnable) {
        Thread thread = null;
        if(priority <= Thread.MAX_PRIORITY) {
            thread = new Thread(runnable);
```

```
        } else {
            try {
                thread = RTThreadAssigner.assignRTThread(priority, runnable);
            } catch(LinkageError e) {}
            if(thread == null) {
                priority = Thread.MAX_PRIORITY;
                thread = new Thread(runnable);
            }
        }
        thread.setPriority(priority);
        return thread;
    }
}

class RTThreadAssigner {
    static Thread assignRTThread(int priority, Runnable runnable) {
        Scheduler defScheduler = Scheduler.getDefaultScheduler();
        PriorityScheduler scheduler = (PriorityScheduler) defScheduler;
        if(priority >= scheduler.getMinPriority()) {
            return new RealtimeThread(
                    null, null, null, null, null, runnable);
        }
        return null;
    }
}
```

リスト 1 のコードは、RTSJ のクラスとともにコンパイルする必要があります。実行時にリアルタイム・クラスが見つからない場合、コードは仮想マシンによってスローされた `LinkageError` をキャッチし、リアルタイム・スレッドの代わりに通常の Java スレッドをインスタンス化します。そのためこのコードは、リアルタイム仮想マシンであるか否かに関わらず、どの仮想マシンでも実行することができます。

リスト 1 では、`RealtimeThread` オブジェクトを提供するメソッドが、それ専用のクラスに分離されています。そのためこのメソッドは、`assignRTThread` メソッドが初めてアクセスされたことよってクラスがロードされるまでは検証されません。クラスがロードされると、ランタイム仮想マシンのバイトコード・ベリファイヤーは、`RealtimeThread` クラスが `Thread` クラスのサブクラスであるかどうかを検証します。リアルタイム・クラスが見つからない場合、この検証は失敗し、`NoClassDefFoundError` がスローされされます。

リフレクションを使用してスレッドを割り当てる

リスト 2 に記載するのは、リスト 1 と同じ効果を持つもう 1 つの手法です。このコードはまず、優先度の値によって必要なスレッド・タイプを判別し、クラス名に基づいてリアルタイム・スレッドまたは通常のスレッドのいずれかをインスタンス化します。このリフレクションを使用したコードがこのクラスに求めるコンストラクターは、`java.lang.Runnable` のインスタンスを最後の引数として取り、その他すべての引数には null 値が渡されるようなコンストラクターです。

リスト 2. リフレクションを使用してスレッドを割り当てる

```
import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;

public class ThreadLogic implements Runnable {
    static void startThread(int priority) {
        Thread thread = ThreadAssigner.assignThread(
                priority, new ThreadLogic());
        thread.start();
    }
```

```
    public void run() {
        System.out.println("Running " + Thread.currentThread());
    }
}

class ThreadAssigner {
    static Thread assignThread(int priority, Runnable runnable) {
        Thread thread = null;
        try {
            thread = assignThread(priority <= Thread.MAX_PRIORITY, runnable);
        } catch(InvocationTargetException e) {
        } catch(IllegalAccessException e) {
        } catch(InstantiationException e) {
        } catch(ClassNotFoundException e) {
        }
        if(thread == null) {
            thread = new Thread(runnable);
            priority = Math.min(priority, Thread.MAX_PRIORITY);
        }
        thread.setPriority(priority);
        return thread;
    }

    static Thread assignThread(boolean regular, Runnable runnable)
        throws InvocationTargetException, IllegalAccessException,
            InstantiationException, ClassNotFoundException {
        Thread thread = assignThread(
                regular ? "java.lang.Thread" :
                "javax.realtime.RealtimeThread", runnable);
        return thread;
    }

    static Thread assignThread(String className, Runnable runnable)
        throws InvocationTargetException, IllegalAccessException,
            InstantiationException, ClassNotFoundException {
        Class clazz = Class.forName(className);
        Constructor selectedConstructor = null;
        Constructor constructors[] = clazz.getConstructors();
        top:
        for(Constructor constructor : constructors) {
            Class parameterTypes[] =
                constructor.getParameterTypes();
            int parameterTypesLength = parameterTypes.length;
            if(parameterTypesLength == 0) {
                continue;
            }
            Class lastParameter =
                parameterTypes[parameterTypesLength - 1];
            if(lastParameter.equals(Runnable.class)) {
                for(Class parameter : parameterTypes) {
                    if(parameter.isPrimitive()) {
                        continue top;
                    }
                }
                if(selectedConstructor == null ||
                    selectedConstructor.getParameterTypes().length
                        > parameterTypesLength) {
                    selectedConstructor = constructor;
                }
            }
        }
        if(selectedConstructor == null) {
            throw new InstantiationException(
                    "no compatible constructor");
        }
        Class parameterTypes[] =
```

```
            selectedConstructor.getParameterTypes();
        int parameterTypesLength = parameterTypes.length;
        Object arguments[] = new Object[parameterTypesLength];
        arguments[parameterTypesLength - 1] = runnable;
        return (Thread) selectedConstructor.newInstance(arguments);
    }
}
```

リスト 2 のコードは、クラスパス上のリアルタイム・クラスとともにコンパイルする必要はありません。リアルタイム・スレッドは、Java リフレクションを使用してインスタンス化されるためです。

クラスの継承によってスレッド・タイプを割り当てる

次に説明するのは、特定クラスの継承を変更してリアルタイム・スレッドを利用する例です。特定のスレッド・クラスでは、`javax.realtime.RealtimeThread` を認識するバージョンと認識しないバージョンの 2 つを作成することができます。どちらのバージョンを選択するかは、ベースとなる JVM 次第です。対応するクラス・ファイルをディストリビューションに含めるだけで、どちらか一方を有効にすることができます。どちらのバージョンのクラスを選ぶにしてもコードは比較的単純で、以前の例とは違い、例外処理は必要ありません。ただし、アプリケーションを配布する際には、アプリケーションを実行することになる仮想マシンに応じて、クラスの 2 つのバージョンのどちらか一方を組み込む必要があります。

リスト 3 のコードでは、標準的な方法で通常の Java スレッドを作成しています。

リスト 3. クラスの継承を使用してスレッドを割り当てる

```
import javax.realtime.PriorityScheduler;
import javax.realtime.RealtimeThread;
import javax.realtime.Scheduler;

public class ThreadLogic implements Runnable {
    static void startThread(int priority) {
        ThreadContainerBase base = new ThreadContainer(priority, new ThreadLogic());
        Thread thread = base.thread;
        thread.start();
    }

    public void run() {
        System.out.println("Running " + Thread.currentThread());
    }
}

class ThreadContainer extends ThreadContainerBase {
    ThreadContainer(int priority, Runnable runnable) {
        super(new Thread(runnable));
        if(priority > Thread.MAX_PRIORITY) {
            priority = Thread.MAX_PRIORITY;
        }
        thread.setPriority(priority);
    }
}

class ThreadContainerBase {
    final Thread thread;

    ThreadContainerBase(Thread thread) {
        this.thread = thread;
    }
}
```

リアルタイム・スレッドを有効にするには、リスト 4 のように `ThreadContainer` コードを変更するという方法を使えます。

リスト 4. リアルタイムの動作を実現する代替スレッド・コンテナー・クラス

```
class ThreadContainer extends ThreadContainerBase {
    ThreadContainer(int priority, Runnable runnable) {
        super(assignRTThread(priority, runnable));
        thread.setPriority(priority);
    }

    static Thread assignRTThread(int priority, Runnable runnable) {
        Scheduler defScheduler = Scheduler.getDefaultScheduler();
        PriorityScheduler scheduler = (PriorityScheduler) defScheduler;
        if(priority >= scheduler.getMinPriority()) {
            return new RealtimeThread(
                    null, null, null, null, null, runnable);
        }
        return new Thread(runnable);
    }
}
```

リアルタイム JVM で実行するときには、古いクラス・ファイルの代わりに、この新しくコンパイルした `ThreadContainer` クラス・ファイルをアプリケーションに組み込むことができます。

分離メモリー領域

リアルタイム JVM も含めたすべての JVM で共通しているのは、ヒープでガーベッジ・コレクションが行われることです。JVM はガーベッジ・コレクションによってヒープからメモリーを回収します。リアルタイム JVM には、特に実行中アプリケーションへの干渉を回避、あるいは最小限に抑えるように設計されたガーベッジ・コレクション・アルゴリズムがあります。

スレッドごとの割り当てコンテキストという概念を導入している RTSJ では、この概念に伴って追加メモリー領域を導入しています。あるメモリー領域が 1 つのスレッドに対して割り当てコンテキストの役割を果たす場合、そのスレッドによってインスタンス化されたすべてのオブジェクトは、このメモリー領域から割り当てられます。RTSJ が指定する追加の分離メモリー領域には以下のものがあります。

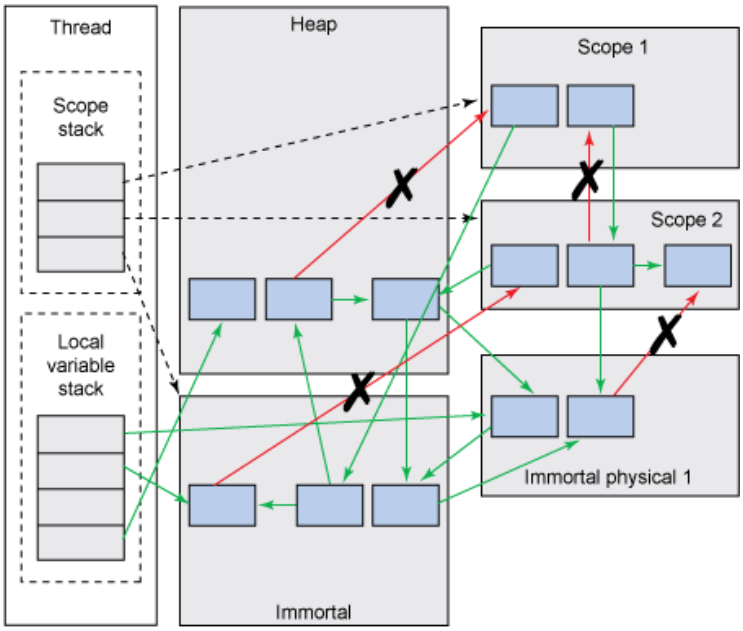
- シングルトン・ヒープ・メモリー領域
- シングルトン永続メモリー領域。この領域のメモリーが別の用途で再利用されることは決してありません。クラスを初期化するスレッドは、この領域を、静的イニシャライザーを実行するときの割り当てコンテキストとして使用します。永続メモリーをガーベッジ・コレクターの対象にする必要はありませんが、メモリーが回収されないことから、この領域のメモリーは無制限に使用できるわけではありません。
- スコープ・メモリー領域 (スコープ)。スコープでは、ガーベッジ・コレクションのアクティビティーは必要ありません。そして、この領域のメモリーは、再利用のために (一度にすべて) 回収可能です。仮想マシンは、ライブ・スレッドの割り当てコンテキスト領域からスコープが外れたと判断すると、そのスコープに割り当てられていたオブジェクトをファイナライズして消去し、オブジェクトに割り当てられていたメモリーを解放して再利用できるようにします。
- 物理メモリー領域。アドレスのタイプによって識別される、それぞれの物理メモリー領域は、再利用するためにスコープ領域として指定することも、1 度だけ使用するために永続領

域として指定することもできます。このようなメモリー領域では、特有の性質を持つメモリーにアクセスしたり、フラッシュ・メモリーや共有メモリーなどの特定のデバイスのメモリーにアクセスしたりすることができます。

スコープでは、オブジェクト参照に関する制約が強化されます。スコープ・メモリー・ブロックが解放されて、そこに含まれるオブジェクトが消去されると、解放されたメモリー・ブロック内を参照するオブジェクトは宙ぶらりんのポインターになってしまうため、存在できなくなります。こうしたことは、割り当てルールを強化することで、一部実現されています。このルールによると、非スコープ・メモリー領域から割り振られたオブジェクトはスコープ・オブジェクトを指すことができません。そのため、スコープ・オブジェクトが解放された後、他のメモリー領域のオブジェクトが存在しないオブジェクトを参照したままになるという事態がなくなります。

図 1 に、これらのメモリー領域と割り当てルールを図解します。

図 1. メモリー領域と、オブジェクト参照に対する割り当てルール



この割り当てルールでは、あるスコープ内のオブジェクトが別のスコープを指すことは許容しています。しかしその場合には、スレッドごとにスコープ・クリーンアップの強制シーケンスが必要となります。つまり、各スレッド内のスタックによって維持管理されるシーケンスが必要になるのです。スタックにはスコープだけでなく、スレッドが利用したことのある別のメモリー領域への参照も含まれています。メモリー領域がスレッドの割り当てコンテキストになると、そのメモリー領域は常にスレッドのスコープ・スタックの上に配置されます。最上位にあるスコープが最初にクリアされることから、割り当てルールでは、スタックの上位にあるスコープ内のオブジェクトは、スタックの下位にあるスコープ内のオブジェクトを参照できるとしています。下位にあるスコープが上位にあるスコープを参照することはできません。

スタックでのスコープの順序は、他のスレッドのスタックでのスコープの順序とも調整されます。スコープが任意のスレッドのスタックに配置されると、スタック上でそのスコープに一番近い下位スコープが親としてみなされます (スタックに他のスコープがない場合には、1 つしかない基本スコープがスコープの親とみなされます)。そのスコープがそのスタック上にある間、ス

コープを他のスレッドのスタック上に配置できるのは、親の一貫性が保たれる場合、つまりそのスコープの親がもう一方のスレッドのスタック上でも最上位のスコープである場合に限ります。言い換えると、使用中のスコープは 1 つの親しか持てないことになります。したがってスコープが解放されると、どのスレッドが各スコープのクリーンアップを実行するのかに関わらず、同じ順序でクリーンアップが行われ、割り当てルールがすべてのスレッドで一貫性を維持することになります。

分離メモリー領域を利用する方法

特定のメモリー領域を使用するには、その領域を (スレッド・オブジェクトが作成されるときに) スレッドを実行する初期メモリー領域として指定するか、その領域をデフォルト領域として実行する `Runnable` オブジェクトを指定した上で、その領域を明示的に指定します。

複数の異なるメモリー領域を使用する場合、複雑になり、場合によってはリスクも生じるため、特別の配慮が必要です。領域のサイズと数は慎重に選んでください。スコープを使用している場合、スレッドのスコープ・スタックの順序を慎重に設計するとともに、割り当てルールを常に意識していなければなりません。

時間制約の厳しいコードのスケジューリング・オプション

ヒープ以外のメモリー領域を使用する場合には、`javax.realtime.RealtimeThread` のサブクラスである `javax.realtime.NoHeapRealtimeThread` (NHRT) を使用する方法を選ぶことができます。この方法では、スレッドがガーベッジ・コレクターから干渉されずに実行することが保証されます。スレッドが干渉されずに実行できる理由は、これらのスレッドは、ヒープから割り当てられたオブジェクトにはアクセスすることができないからです。このアクセス制限に違反する試みが行われると、`javax.realtime.MemoryAccessError` がスローされます。

別のスケジューリング方法として、非同期イベント・ハンドラーを使用する方法もあります。この方法では、コードが非同期イベントまたは周期イベントに応答して実行されるようにスケジューリングする場合に使用することができます (イベントがタイマーによって起動される場合は、周期イベントになります)。非同期イベント・ハンドラーを使用することで、非同期または周期イベントに対してスレッドを明示的にスケジューリングする必要はなくなります。代わりに仮想マシンが、イベントの発生時に非同期イベント・ハンドラーのコードを実行するためにディスパッチされる、共有スレッドのプールを維持管理します。この場合、スレッドとメモリー領域を管理する必要がなくなるため、リアルタイム・アプリケーションが単純になります。

図 2 のクラス図に、コードをスケジューリングするために選択可能な方法を示します。

図 2. コードをスケジューリングするための方法を示すクラス図

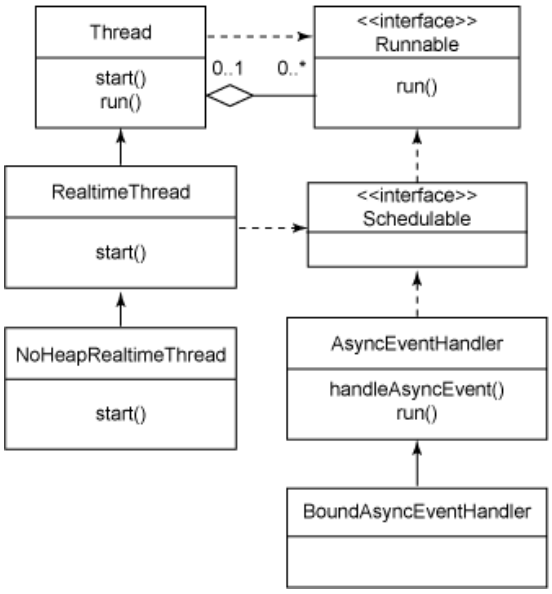
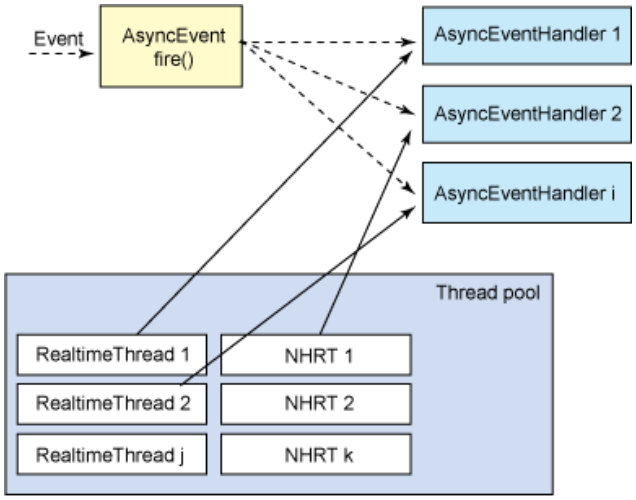


図 3 に、非同期イベント・ハンドラーがどのようにディスパッチされるかを示します。

図 3. 非同期イベント・ハンドラーのディスパッチ方法



一般に移植性とモジュール性に関して言えば、イベントに応答するコードと、ハンドラーを有効にしてそのハンドラーをディスパッチするコードとは分離しておくのが有益です。コードを `java.lang.Runnable` の実装にカプセル化すれば、そのコードをディスパッチする際に多数のオプションから選択することが可能になります。例えば、コードを実行するスレッドを作成するか、スレッドのプールを使用する非同期イベント・ハンドラーによってオンデマンドでコードを実行するか、あるいはこの 2 つを組み合わせて使用するかを選べます。

表 1 に、考えられるさまざまな選択肢の特性を大まかに分類します。

表 1. リアルタイム Java におけるコードのディスパッチ方法の比較

	コードを実行するスレッドの共有	定期的なディスパッチ	ヒープ・メモリー内での実行	永続メモリー内での実行	スコープ・メモリー内での実行	期限の割り当て	実行時のガーベッジ・コ

							レクション による干渉
通常の`Thread`	なし	不可	可	可	不可	不可	あり
`RealtimeThread`	なし	可	可	可	可	可	あり
`NoHeapRealtimeThread`	なし	可	不可	可	可	可	なし
`AsyncEventHandler`	あり	可 (周期タイマーを使用している場合)	可	可	可	可	あり
`BoundAsyncEventHandler`	なし	可 (周期タイマーを使用している場合)	可	可	可	可	あり
ヒープを使用しない `AsyncEventHandler`	あり	可 (周期タイマーを使用している場合)	不可	可	可	可	なし
ヒープを使用しない `BoundAsyncEventHandler`	なし	可 (周期タイマーを使用している場合)	不可	可	可	可	なし

どのスケジューリング・オプションとメモリー領域を使用するかを検討するときには、リアルタイム Java に固有の設計問題が関与してきます。概して、リアルタイム環境向けのプログラミングは、通常の単純なアプリケーションのプログラミングよりも難易度の高い作業となり、リアルタイム Java 固有の難題が関わってくるものです。表 2 に、追加のメモリー領域や、NHRT、その他のリアルタイム機能を使用するのに伴う厄介な問題をいくつか抜粋します。

表 2. リアルタイム・スレッド化およびメモリー領域の問題と落とし穴

考慮事項	詳細
メモリー領域に割り当てるメモリー	アプリケーションによって作成されるメモリー領域のそれぞれには、要求されるサイズが割り当てられます。大きすぎるサイズを選択するとメモリーを非効率的に使用することになる一方、サイズが小さすぎると、アプリケーションで `OutOfMemoryError` が発生しやすくなります。開発中には、アプリケーションが変更されないとしても、ベースとなるライブラリーが変更される可能性があります。その結果、予想外のメモリーを追加で使用する羽目になり、メモリー領域の制限を超過するという事態になりかねません。
共有スコープのタイミングに関する考慮事項	複数のスレッドが共有するスコープ・メモリー領域は、この領域を使用しているスレッドがなくなるとクリアされることから、十分なサイズがあるように思えるかもしれません。しかし、スコープを使用するスレッドのタイミングが微妙に変更されると、スレッドの割り当てコンテキストとしてスコープが常に使用されている状況になる可能性があります。この場合、スコープ・メモリー領域が決してクリアされずに、`OutOfMemoryError` が発生するという不測の事態となる可能性が生まれます。 共有スコープ領域をスレッドが利用した後、その共有スコープ領域がクリアされると、スレッド間で一時ロックの競合が発生する可能性があります。
ランタイム例外 `IllegalAssignmentError`、`MemoryAccessError`、および `IllegalThreadStateException`	これらの例外は、コード設計に十分注意を払っていないと発生することがあります。実際、プログラムの振る舞いとタイミングを微妙に変更しただけでも、思いがけずこれらの例外が発生する原因となります。以下はその例です。 • スレッド間のタイミングと同期を変更したために、通常は NHRT には使用できないヒープのオブジェクトが使用可能になる場合があります。 • オブジェクトがどのメモリー領域から割り当てられているのか、あるいはスコープ・スタックのどこに

	特定のスコープが配置されているのかがわからないと、`IllegalAssignmentError` が発生する可能性があります。 • スコープ・メモリー領域に入ったコードが通常のスレッドによって実行されると、`IllegalThreadStateException` がスローされます。 • 静的フィールドやその他のデータ・キャッシング手段を一般的に使用するコードは、割り当てルールがあるため、スコープに使うには安全ではなく、`IllegalAssignmentError` が発生する可能性があります。
クラスの初期化	通常のスレッドやリアルタイム・スレッドは、どんなスレッドでも、NHRT を始めとするクラスを初期化することができますが、それによって不測の `MemoryAccessError` が発生する場合があります。
`finalize` メソッドによるオブジェクトのファイナライズ	スコープを出る最後のスレッドを使用して、スコープ内のすべてのオブジェクトがファイナイズされることから、以下の問題が考えられます。 • `finalize` メソッドがスレッドを作成した場合、スコープが正常にクリアされない場合があります。 • ファイナライズはデッドロックを発生させることもあります。メモリー領域をファイナライズする前に、ファイナライズ・スレッドがロックを取得済みの場合があります。このような他のスレッドによるロック、さらにファイナライズ中に取得されるロックが競合することにより、デッドロックが発生する可能性があります。
予期せぬ NHRT の遅延	NHRT はガーベッジ・コレクションに直接干渉されずに実行されることが保証されているものの、ガーベッジ・コレクションによるプリエンプト可能な他のタイプのスレッドとロックを共有することができます。そのため、ロックを所有するスレッドがガーベッジ・コレクションによって遅延されている場合、そのロックを取得しようとしている NHRT も遅延されることになり、結局、NHRT はガーベッジ・コレクションによって間接的に遅延されることになります。

包括的な例

次の例では、これまで説明したリアルタイム機能のいくつかを盛り込みます。まずはリスト 5 に、イベント・データのプロデューサーとコンシューマーを記述する 2 つのクラスを記載します。どちらのクラスも `Runnable` の実装なので、どの `Schedulable` オブジェクトでもこのクラスを簡単に実行することができます。

リスト 5. イベント・オブジェクトのプロデューサーおよびコンシューマー・クラス

```
class Producer implements Runnable {
    volatile int eventIdentifier;
    final Thread listener;

    Producer(Thread listener) {
        this.listener = listener;
    }

    public void run() {
        LinkedList<Integer> events = getEvents();
        synchronized(listener) {
            listener.notify();
            events.add(++eventIdentifier); //autoboxing creates an Integer object here
        }
    }

    static LinkedList<Integer> getEvents() {
        ScopedMemory memoryArea = (ScopedMemory) RealtimeThread.getCurrentMemoryArea();
        LinkedList<Integer> events =
            (LinkedList<Integer>) memoryArea.getPortal();
        if(events == null) {
```

```
            synchronized(memoryArea) {
                if(events == null) {
                    events = new LinkedList<Integer>();
                    memoryArea.setPortal(events);
                }
            }
        }
        return events;
    }
}

class Consumer implements Runnable {
    boolean setConsuming = true;
    volatile boolean isConsuming;

    public void run() {
        Thread currentThread = Thread.currentThread();
        isConsuming = true;
        try {
            LinkedList<Integer> events = Producer.getEvents();
            int lastEventConsumed = 0;
            synchronized(currentThread) {
                while(setConsuming) {
                    while(lastEventConsumed < events.size()) {
                        System.out.print(events.get(lastEventConsumed++) + " ");
                    }
                    currentThread.wait();
                }
            }
        } catch(InterruptedException e) {
        } finally {
            isConsuming = false;
        }
    }
}
```

リスト 5 では、プロデューサーとコンシューマーの両方のオブジェクトが、`java.lang.Integer` オブジェクトのシーケンスとしてエンコードされたイベントのキューにアクセスします。このコードは現行の割り当てコンテキストがスコープ・メモリー領域であること、そしてイベントのキューがスコープのポータル・オブジェクトとして保存されることを期待します (ポータル・オブジェクトとは、スコープ・メモリー領域オブジェクト自体に保存可能なスコープから割り当てられるオブジェクトのことです。スコープ・オブジェクトは静的フィールドにも、親スコープから割り当てられたオブジェクトにも保存できないため、このポータル・オブジェクトが役に立ちます)。キューが見つからない場合には、キューが作成されます。対象となるスレッドにイベントの作成状況および使用状況について通知するために、いくつかの volatile フィールドが使用されています。

リスト 6 の 2 つのクラスは、リスト 5 のコードを実行する方法を示しています。

リスト 6. スケジュール可能なクラス

```
class NoHeapHandler extends AsyncEventHandler {
    final MemoryArea sharedArea;
    final Producer producer;

    NoHeapHandler(
            PriorityScheduler scheduler,
            ScopedMemory sharedArea,
            Producer producer) {
        super(new PriorityParameters(scheduler.getMaxPriority()),
                null, null, null, null, true);
```

```
        this.sharedArea = sharedArea;
        this.producer = producer;
    }

    public void handleAsyncEvent() {
        sharedArea.enter(producer);
    }
}

class NoHeapThread extends NoHeapRealtimeThread {
    boolean terminate;
    final MemoryArea sharedArea;
    final Consumer consumer;

    NoHeapThread(
            PriorityScheduler scheduler,
            ScopedMemory sharedArea,
            Consumer consumer) {
        super(new PriorityParameters(scheduler.getNormPriority()),
            RealtimeThread.getCurrentMemoryArea());
        this.sharedArea = sharedArea;
        this.consumer = consumer;
    }

    public synchronized void run() {
        try {
            while(true) {
                if(consumer.setConsuming) {
                    sharedArea.enter(consumer);
                } else {
                    synchronized(this) {
                        if(!terminate) {
                            if(!consumer.setConsuming) {
                                wait();
                            }
                        } else {
                            break;
                        }
                    }
                }
            }
        } catch(InterruptedException e) {}
    }
}
```

リスト 6 では、データ・プロデューサー・コードが非同期イベント・ハンドラーに割り当てられ、使用可能な優先度のうち、最も高い優先度で実行されます。ハンドラーは単純にスコープ・メモリー領域に入ってプロデューサー・コードを実行します。このスコープ・メモリー領域が、データのコンシューマーとして機能する NHRT に対するパラメーターにもなります。スレッド・クラスも同じく単純明快で、振る舞いを決定する `terminate` フィールドと `setConsuming` フィールドへの同期アクセスを可能にしているのはこのクラスです。コンシューマー・スレッドがイベントを使用するときには、共有メモリー領域に入り、プロデューサーよりも低い優先度で動作してコンシューマー・コードを実行します (この例でのコンシューマーの振る舞いは平凡なもので、単にイベント ID をコンソールに出力するだけです)。

リスト 7 に記載するコードがシステムを初期化し、システムの振る舞いを明らかにします。

リスト 7. システムの振る舞い

```
public class EventSystem implements Runnable {
    public static void main(String args[]) throws InterruptedException {
```

```
        RealtimeThread systemThread = new RealtimeThread(
                null, null, null, new VTMemory(20000L), null, null) {
            public void run() {
                VTMemory systemArea = new VTMemory(20000L, new EventSystem());
                systemArea.enter();
            }
        };
        systemThread.start();
    }

    public void run() {
        try {
            PriorityScheduler scheduler =
                (PriorityScheduler) Scheduler.getDefaultScheduler();
            VTMemory scopedArea = new VTMemory(20000L);
            Consumer consumer = new Consumer();
            NoHeapThread thread = new NoHeapThread(scheduler, scopedArea, consumer);
            Producer producer = new Producer(thread);
            NoHeapHandler handler = new NoHeapHandler(scheduler, scopedArea, producer);
            AsyncEvent event = new AsyncEvent();
            event.addHandler(handler);

            int handlerPriority =
                ((PriorityParameters) handler.getSchedulingParameters()).getPriority();
            RealtimeThread.currentRealtimeThread().setPriority(handlerPriority - 1);

            thread.start();
            waitForConsumer(consumer);

            //fire several events while there is a consumer
            event.fire();
            event.fire();
            event.fire();
            waitForEvent(producer, 3);

            setConsuming(thread, false);

            //fire a couple of events while there is no consumer
            event.fire();
            event.fire();

            waitForEvent(producer, 5);

            setConsuming(thread, true);
            waitForConsumer(consumer);

            //fire another event while there is a consumer
            event.fire();
            waitForEvent(producer, 6);

            synchronized(thread) {
                thread.terminate = true;
                setConsuming(thread, false);
            }

        } catch(InterruptedException e) {}
    }

    private void setConsuming(NoHeapThread thread, boolean enabled) {
        synchronized(thread) {
            thread.consumer.setConsuming = enabled;
            thread.notify();
        }
    }

    private void waitForEvent(Producer producer, int eventNumber)
            throws InterruptedException {
```

```
        while(producer.eventIdentifier < eventNumber) {
            Thread.sleep(100);
        }
    }

    private void waitForConsumer(Consumer consumer)
            throws InterruptedException {
        while(!consumer.isConsuming) {
            Thread.sleep(100);
        }
    }
}
```

リスト 7 ではスコープのペアを、ヒープを使用しないスレッドおよびハンドラー用の、スコープ・スタックのベースとして使用しています。こうする必要があるのは、これらの `Schedulable` はヒープが割り振られたオブジェクトにはアクセスできないためです。非同期イベント・オブジェクトはイベントを表し、イベントの起動時にディスパッチされるハンドラーが接続されています。システムが初期化されると、このコードがコンシューマー・スレッドを開始し、イベントを何回か起動して、イベント・ハンドラーの優先度より 1 つ下の優先度で実行します。このコードはまた、追加イベントの起動中にコンシューマー・スレッドのオン/オフを切り替えます。

リスト 8 は、`EventSystem` をリアルタイム JVM で実行した場合の出力です。

リスト 8. コンソールの出力

```
1 2 3 6
```

この例で興味深い点は、イベント 4 とイベント 5 がレポートされていない理由です。リッスンしている側のスレッドは、キュー内のイベントをレポートするたびにキューの先頭から末尾へと進みます。つまり、6 つのイベントがいずれも最低 1 回はレポートされるということを意味します。

しかしこの設計では、イベントの保存に使用するメモリーは、すべてのスレッドがメモリーを使用していないと自動的に破棄されるようになっています。コンシューマー・スレッドはキューからの読み取りを停止すると、スコープ・メモリー領域から出ます。この時点で、このスコープ・メモリー領域を割り当てコンテキストとして使用している `Schedulable` オブジェクトはなくなります。

この領域を使用する `Schedulable` オブジェクトがないということは、スコープ・メモリー領域のオブジェクトがクリアされてリセットされるということです。これにはポータル・オブジェクトも含まれるため、スレッドがリッスン動作を停止すると、キューとそのキュー内のすべてのイベントが破棄されます。後続のイベントが起動されるたびに、キューが再作成され、そのキューにイベントが入れられますが、リッスンしているスレッドがないことから、メモリーはその後すぐに破棄されます。

メモリー管理は自動であり、ガーベッジ・コレクターがアクティブでも、コレクターによって干渉されることなく実行されます (ハンドラーとスレッドはいずれもヒープを使用しないため)。イベントはオブジェクトのキューとしてメモリーの中に保存され、リッスンする側のスレッドが有効になってイベントを使用するようになると、メモリー内のキューは大きくなり続けます。そうでない場合は、キューおよび関連付けられたイベントは自動的に破棄されます。

一般的な使用シナリオ

スケジューリング・フレームワークやメモリー管理フレームワークでは、スレッドにさまざまな優先度を設定して、リアルタイム仮想マシンで最適に (場合によっては他の仮想マシンでも適切に) 実行されるアプリケーションを設計することができます。アプリケーションに最高の優先度を持つイベント処理スレッドを組み込み、外部入力からのデータを収集して処理用にデータを保存することもできます。イベント処理スレッドは、その過渡的で非同期であるという特質により、メモリー管理の代替手段としてふさわしいだけでなく、リアルタイム要件にも極めて確実に準拠するはずです。中間の優先度としては、データを使用して計算を行ったり、データを配布したりする処理スレッドが考えられます。このような中間スレッドには、そのワークロードの管理に十分な CPU 使用時間を割り当てる必要があるかもしれません。最も低い優先度のスレッドとしては、保守およびロギング用のスレッドが考えられます。リアルタイム仮想マシンを使用して、アプリケーションでのこのような各種のタスクのスケジューリングとメモリー使用量を管理することで、アプリケーションは最大限効率的に実行できるようになります。

RTSJ の目的は、開発者が、必要なリアルタイム要件のなかで実行するアプリケーションを作成できるようにすることです。この目的を果たすには、リアルタイム・スケジューラーとスレッドを使うだけで十分です。それでも目的を果たせないようであれば、さらに高度な開発によって、仮想マシンが実装するさらに高度な 1 つ以上の機能を利用する必要があります。

第 1 回のまとめ

この記事では、Java アプリケーションにリアルタイム Java の要素を統合する上でのヒントをいくつか紹介しました。そこでは、リアルタイム・パフォーマンスを実現するために使用できるスケジューリングおよびメモリー管理機能を取り上げました。これは、相互運用性や安全性などといった従来の Java 言語の利点を活用しながらも、さらにアプリケーションに必要なリアルタイム要件を満たす新機能を加えるための出発点です。

連載の次回の記事では、既存のアプリケーションをリアルタイム Java に移植する手法を学びます。今回と次回の記事をベースに、最終回ではリアルタイム Java を統合したリアルタイム・システムを設計、検証、デバッグする方法を紹介します。

ダウンロード

内容	ファイル名	サイズ
Source code for the article examples	j-devrtj1.zip	5KB

著者について

Sean C. Foley

Sean Foley は、IBM Java Technology Centre の IBM Ottawa Lab に在籍するソフトウェア開発者です。Queen's University で数学の理学士号を取得し、University of Toronto で数学の修士号を取得した彼は、大学院で設計理論とグラフ理論の組み合わせにおける問題を焦点とした研究を行っていました。その後、移動体通信および組み込みプロセッサー業界で企業用ソフトウェアの開発に経験を積み、2002年、IBM Software Group に入社しました。IBM では、組み込み JVM、そしてコンパイル済み Java プログラムの静的分析および最適化を行うツールなどのサポート製品の開発に携わっています。最近では、IBM WebSphere Real Time 製品のリアルタイム・クラス・ライブラリー実装で主要な貢献を果たしました。現在、リアルタイム Java 技術の開発および改善に取り組むチームで技術主任を務めています。