Javaアプリケーションで繰り返しタスクをスケジューリングする

Java言語のタイマー・クラスを一般化する

Tom White
Lead Java Developer
Kizoom

2003年 11月 04日

どんなJavaアプリケーションでも、タスクをスケジューリングして繰り返し実行できる必要があります。企業レベルでのアプリケーションでは日々のロギングや夜間のバッチ処理をスケジューリングする必要があります。J2SEやJ2MEのカレンダー・アプリケーションでは、ユーザーの行動予定に合わせてお知らせをする機能が必要になります。ところが普通のスケジューリング・クラスである、`Timer`や`TimerTask`は柔軟性に乏しく、よく要求されるようなスケジューリング・タスクには対応しきれません。この記事ではJava開発者であるTom Whiteが、簡単かつ一般的なタスクのスケジューリング・フレームワークを、任意で複雑なスケジュールに合わせて構築するにはどうすべきかを説明します。

`java.util.Timer`と`java.util.TimerTask`で簡単なタスクをスケジューリングできますが、この2つを合わせてJava タイマー・フレームワーク と呼ぶことにします。（これらのクラスはJ2MEにもあることに注意してください。）Java 2 SDKでこのフレームワークが導入される前、つまりStandard Edition, Version 1.3では、スレッドや微妙な`Object.wait()`メソッドと格闘しながら、開発者は自分でスケジューラーを書かなければなりませんでした。ただ、Javaのタイマー・フレームワークは多くのアプリケーションで要求されるスケジューリングの用途には十分とは言えません。毎日毎日、同じ時間に繰り返すタスクであっても、夏時間があるとその始まりと終わりで時間のずれが発生するので、`Timer`を使って直接スケジューリングすることはできません。

この記事では、より柔軟なスケジューリングができるように`Timer`と`TimerTask`を一般化したスケジューリング・フレームワークを説明します。非常に単純なもので、2つのクラスと1つのインターフェースからなり、簡単に覚えられます。Javaタイマー・フレームワークを使い慣れていれば、このスケジューリング・フレームワークもすぐに理解できるでしょう。（Javaタイマー・フレームワークについては参考文献を見てください。）

ワンショット・タスクのスケジューリング

このスケジューリング・フレームワークはJavaタイマー・フレームワーク・クラスの上に作られています。スケジューリング・フレームワークをどう使い、どう実装するかを説明する前にまず、これらのクラスを使ったスケジューリングを見てみましょう。

エッグ・タイマー（ゆで卵器用のタイマー）を考えてみてください。何分経過したか（つまり、どのくらい卵がゆだっているか）を音で知らせるものです。リスト1は簡単なエッグ・タイマーをJavaで書いたときの基本を示しています。

リスト1. EggTimerクラス

```
package org.tiling.scheduling.examples;
import java.util.Timer;
import java.util.TimerTask;
public class EggTimer {
    private final Timer timer = new Timer();
    private final int minutes;
    public EggTimer(int minutes) {
        this.minutes = minutes;
    }
    public void start() {
        timer.schedule(new TimerTask() {
            public void run() {
                playSound();
                timer.cancel();
            }
            private void playSound() {
                System.out.println("Your egg is ready!");
                // Start a new thread to play a sound...
            }
        }, minutes * 60 * 1000);
    }
    public static void main(String[] args) {
        EggTimer eggTimer = new EggTimer(2);
        eggTimer.start();
    }
}
```

`EggTimer`は`Timer`インスタンスを所有し必要なスケジューリングを行います。`start()`メソッドでエッグ・タイマーが動き始めると、タイマーは`TimerTask`を指定分後に実行するようにスケジュールします。時間が来ると、`Timer`が後ろに隠れている`TimerTask`の`run()`メソッドを呼び、`run()`メソッドに音を発生させます。タイマーが中止されるとアプリケーションは終了します。

繰り返しタスクをスケジュールする

`Timer`を使うと一定の実行間隔、または各実行間に一定の遅延を指定して、タスクを繰り返し実行するようにスケジュールすることができるのですが、多くのアプリケーションではもっと複雑なスケジューリングが要求されます。例えば毎朝同じ時間に鳴る目覚まし時計でも、単純に86400000ミリ秒（24時間）の実行間隔を設定するわけには行きません。(夏時間を採用している地域では）一定間隔の実行では夏時間が始まる日には鳴る時間が遅すぎ、終わる日には早すぎることになってしまうからです。これを解決するにはカレンダー計算式を使い、日々のイベントが次にいつ起きるかを計算するわけですが、これはまさにスケジューリング・フレームワークでできることなのです。リスト2の`AlarmClock`実装を考えてみてください。（スケジューリング・フレームワークのソースコードや、このフレームワークやその例を含んだJARファイルのダウンロードについては参考文献を見てください。）

リスト2. AlarmClockクラス

```
package org.tiling.scheduling.examples;
import java.text.SimpleDateFormat;
import java.util.Date;
import org.tiling.scheduling.Scheduler;
import org.tiling.scheduling.SchedulerTask;
import org.tiling.scheduling.examples.iterators.DailyIterator;
public class AlarmClock {
    private final Scheduler scheduler = new Scheduler();
    private final SimpleDateFormat dateFormat =
        new SimpleDateFormat("dd MMM yyyy HH:mm:ss.SSS");
    private final int hourOfDay, minute, second;
    public AlarmClock(int hourOfDay, int minute, int second) {
        this.hourOfDay = hourOfDay;
        this.minute = minute;
        this.second = second;
    }
    public void start() {
        scheduler.schedule(new SchedulerTask() {
            public void run() {
                soundAlarm();
            }
            private void soundAlarm() {
                System.out.println("Wake up! " +
                    "It's " + dateFormat.format(new Date()));
                // Start a new thread to sound an alarm...
            }
        }, new DailyIterator(hourOfDay, minute, second));
    }
    public static void main(String[] args) {
        AlarmClock alarmClock = new AlarmClock(7, 0, 0);
        alarmClock.start();
    }
}
```

このコードがエッグ・タイマーと非常に似ていることに気がついたかと思います。`AlarmClock`は、（`Timer`ではなく）`Scheduler`を所有し必要なスケジューリングを行います。目覚まし時計は動き始めると（`TimerTask`ではなく）`SchedulerTask`をスケジュールし、アラームを鳴らします。そして一定の遅延後に実行するようにスケジュールする代わりに、`DailyIterator`クラスを使ってスケジュールを記述します。この場合では、毎日午前7時にタスクをスケジュールします。実行すると出力はこんな風になります。

```
Wake up! It's 24 Aug 2003 07:00:00.023
Wake up! It's 25 Aug 2003 07:00:00.001
Wake up! It's 26 Aug 2003 07:00:00.058
Wake up! It's 27 Aug 2003 07:00:00.015
Wake up! It's 28 Aug 2003 07:00:00.002
...
```

`DailyIterator`は`ScheduleIterator`を実行します。`ScheduleIterator`は、一連の`java.util.Date`オブジェクトの一つとしての`SchedulerTask`が、スケジュールに従って実行する時間を規定するインターフェースです。`next()`メソッドは次に、`Date`オブジェクトに従って時間の順に繰り返します。戻り値`null`でタスクは中止されます（つまりタスクは再度走ることはありません。実際、再度スケジュールしようとすると例外がスローされます）。リスト3には`ScheduleIterator`インターフェースがあります。

リスト3. ScheduleIteratorインターフェース

```
package org.tiling.scheduling;
import java.util.Date;
public interface ScheduleIterator {
    public Date next();
}
```

DailyIteratorのnext()メソッドはリスト4に示すように、毎日同じ時間（午前7時）を表すDateオブジェクトを返します。ですから新しく作られたDailyIteratorクラス上でnext()を呼ぶと、コンストラクターに渡された日またはその後の日の午前7時が返ります。その後next()を呼ぶとその後の日の午前7時が返り、これが永遠に続きます。これを実現するためにDailyIteratorはjava.util.Calendarインスタンスを使います。コンストラクターはカレンダーを設定し、最初にnext()が起動されると、そのカレンダーに単純に一日を加えることで正しいDateを返せるようにします。ここでのコードには夏時間補正に関して具体的には何も含まれていないことに注意してください。これはCalendarを実装することで（この場合ではGregorianCalendar）夏時間補正がされるからです。

リスト4. DailyIteratorクラス

```
package org.tiling.scheduling.examples.iterators;
import org.tiling.scheduling.ScheduleIterator;
import java.util.Calendar;
import java.util.Date;
/**
 * A DailyIterator class returns a sequence of dates on subsequent days
 * representing the same time each day.
 */
public class DailyIterator implements ScheduleIterator {
    private final int hourOfDay, minute, second;
    private final Calendar calendar = Calendar.getInstance();
    public DailyIterator(int hourOfDay, int minute, int second) {
        this(hourOfDay, minute, second, new Date());
    }
    public DailyIterator(int hourOfDay, int minute, int second, Date date) {
        this.hourOfDay = hourOfDay;
        this.minute = minute;
        this.second = second;
        calendar.setTime(date);
        calendar.set(Calendar.HOUR_OF_DAY, hourOfDay);
        calendar.set(Calendar.MINUTE, minute);
        calendar.set(Calendar.SECOND, second);
        calendar.set(Calendar.MILLISECOND, 0);
        if (!calendar.getTime().before(date)) {
            calendar.add(Calendar.DATE, -1);
        }
    }
    public Date next() {
        calendar.add(Calendar.DATE, 1);
        return calendar.getTime();
    }
}
```

スケジューリング・フレームワークを実装する

前の節ではスケジューリング・フレームワークの使い方を学び、Javaタイマー・フレームワークとの比較をしました。次は実装方法を説明します。スケジューリング・フレームワークには、リスト3に示すScheduleIteratorインターフェースに加え、これとは別にSchedulerとSchedulerTaskという2つのクラスがあります。これらのクラスは、実際には裏でTimerやTimerTaskを使っています

が、これはスケジューラーがワンショット・タイマーの連続にすぎないからです。リスト5と6はこれら2つのクラスのソースコードです。

リスト5. Schedulerのソースコード

```
package org.tiling.scheduling;
import java.util.Date;
import java.util.Timer;
import java.util.TimerTask;
public class Scheduler {
    class SchedulerTimerTask extends TimerTask {
        private SchedulerTask schedulerTask;
        private ScheduleIterator iterator;
        public SchedulerTimerTask(SchedulerTask schedulerTask,
                ScheduleIterator iterator) {
            this.schedulerTask = schedulerTask;
            this.iterator = iterator;
        }
        public void run() {
            schedulerTask.run();
            reschedule(schedulerTask, iterator);
        }
    }
    private final Timer timer = new Timer();
    public Scheduler() {
    }
    public void cancel() {
        timer.cancel();
    }
    public void schedule(SchedulerTask schedulerTask,
            ScheduleIterator iterator) {
        Date time = iterator.next();
        if (time == null) {
            schedulerTask.cancel();
        } else {
            synchronized(schedulerTask.lock) {
                if (schedulerTask.state != SchedulerTask.VIRGIN) {
                  throw new IllegalStateException("Task already scheduled " + "or cancelled");
                }
                schedulerTask.state = SchedulerTask.SCHEDULED;
                schedulerTask.timerTask =
                    new SchedulerTimerTask(schedulerTask, iterator);
                timer.schedule(schedulerTask.timerTask, time);
            }
        }
    }
    private void reschedule(SchedulerTask schedulerTask,
            ScheduleIterator iterator) {
        Date time = iterator.next();
        if (time == null) {
            schedulerTask.cancel();
        } else {
            synchronized(schedulerTask.lock) {
                if (schedulerTask.state != SchedulerTask.CANCELLED) {
                    schedulerTask.timerTask =
                        new SchedulerTimerTask(schedulerTask, iterator);
                    timer.schedule(schedulerTask.timerTask, time);
                }
            }
        }
    }
}
```

リスト6は SchedulerTask クラスのソースコードです。

リスト6.SchedulerTaskのソースコード

```
package org.tiling.scheduling;
import java.util.TimerTask;
public abstract class SchedulerTask implements Runnable {
    final Object lock = new Object();
    int state = VIRGIN;
    static final int VIRGIN = 0;
    static final int SCHEDULED = 1;
    static final int CANCELLED = 2;
    TimerTask timerTask;
    protected SchedulerTask() {
    }
    public abstract void run();
    public boolean cancel() {
        synchronized(lock) {
            if (timerTask != null) {
                timerTask.cancel();
            }
            boolean result = (state == SCHEDULED);
            state = CANCELLED;
            return result;
        }
    }
    public long scheduledExecutionTime() {
        synchronized(lock) {
         return timerTask == null ? 0 : timerTask.scheduledExecutionTime();
        }
    }
}
```

エッグ・タイマーの場合と同じように、Schedulerの各インスタンスはどれもTimerのインスタンスを所有し、その下にあるスケジューリングを行います。エッグ・タイマーで使ったワンショット・タイマーの代わりに、Schedulerは一連のワンショット・タイマーをつなげ、ScheduleIteratorに指定された時間にSchedulerTaskクラスを実行します。

Schedulerのpublicschedule()メソッド（クライアントが呼ぶメソッドなのでスケジューリングのエントリー・ポイントです）を考えてみてください。（他のパブリック・メソッドとして唯一のcancel()はタスクを中止するで説明しています。）SchedulerTaskを最初に実行する時間はScheduleIteratorインターフェースのnext()を呼ぶことで分かります。するとこの時間に実行すべく下のTimerクラスにあるワンショットのschedule()メソッドが呼ばれ、スケジューリングがスタートします。ワンショットの実行に使われるTimerTaskオブジェクトはネストされたSchedulerTimerTaskクラスのインスタンスですが、このインスタンスはタスクとイテレーターをパッケージしています。割り当てられた時間になるとネストされたクラスのrun()メソッドが呼ばれますが、これはパッケージされたタスクとイテレーターへの参照を使って次のタスクの実行をスケジュールします。reschedule()メソッドはschedule()メソッドとよく似ていますが、reschedule()はプライベート・メソッドで、SchedulerTaskに対する状態チェックの仕方が少し違います。このリスケジューリング・プロセスは永遠に繰り返されますが、この繰り返しは各スケジュール実行の度にネストされたクラス・インスタンスを生成しながら、タスクまたはスケジューラーが中止されるまで（またはJVMが停止されるまで）行われます。

TimerTaskと対応するSchedulerTaskはTimerTaskと同じように、その存続期間中に一連の状態を経ます。生成されたときにはVIRGIN状態ですが、これは単にまだスケジュールされたことがないことを意味します。一旦スケジュールされるとSCHEDULED状態に移り、その後タスクが下記のメソッドどれかにより中止されると、CANCELLED状態に移ります。状態遷移を正しく管理しよう

とすると、例えば非VIRGIN状態が2回スケジュールされたりしないよう確実に管理しようとすると、SchedulerやSchedulerTaskクラスがより複雑なものになります。タスクの状態を変化させるような操作が行われるときには常に、コードはタスクのロック・オブジェクトに同期している必要があります。

タスクを中止する

スケジュールされたタスクを中止するには3つの方法があります。1番目はSchedulerTaskのcancel()メソッドを呼ぶ方法です。これはTimerTaskのcancel()を呼ぶのと似ています。つまりタスクは再度走ることはありませんが、既に走っていれば終わりまでは走ります。cancel()メソッドの戻り値はBooleanで、cancel()メソッドが呼ばれずにいたらスケジュールされたタスクがそのまま走っていたかどうかを示します。もっと正確に言うと、cancel()が呼ばれる直前にそのタスクがSCHEDULED状態であった場合にはtrueを返します。中止したタスク（スケジュールされたタスクであっても）を再スケジュールしようとする時には、SchedulerはIllegalStateExceptionをスローします。

2番目の方法はScheduleIteratorがnullを返すものです。これはSchedulerクラスがSchedulerTaskクラスのcancel()を呼ぶので、最初の方法の単純なショートカットです。この方法は、（タスクではなく）イテレーターにいつスケジューリングを停止させるかを制御させたい場合に有効です。

3番目はcancel()メソッドを呼ぶことでScheduler全体を中止するというものです。これはスケジューラーの全タスクを中止し、スケジューラーにそれ以上タスクがスケジュールされないようにします。

cron機能を拡張する

スケジューリング・フレームワークはUNIXのcron機能のようなものです。ただし時間をスケジューリングする仕様が宣言ではなく、命令として制御されると言う点がちょっと違いますが。例えばAlarmClock実装に使われているDailyIteratorクラスは、0 7 * * *で始まるcrontabエントリーで規定されるcronジョブと同じスケジューリングです。（ここで各フィールドはそれぞれ、分、時間、日、月、曜日を規定します。）

ところがスケジューリング・フレームワークはcronよりもずっと柔軟です。毎朝、給湯器のスイッチを入れるHeatingControllerというアプリケーションを考えてみてください。この給湯器を「平日は午前8時に、週末は午前9時にスイッチを入れる」ようにしたいと思います。cronを使うと、2つのcrontabエントリーが必要です（0 8 * * 1,2,3,4,5と0 9 * * 6,7）。これがScheduleIteratorを使うと、コンポジションで一つのイテレーターを定義すればよいだけなので、ずっとスマートに解決できるのです。リスト7はこの一例です。

リスト7. コンポジションでイテレーターを定義する

```
int[] weekdays = new int[] {
        Calendar.MONDAY,
        Calendar.TUESDAY,
        Calendar.WEDNESDAY,
        Calendar.THURSDAY,
        Calendar.FRIDAY
    };
    int[] weekend = new int[] {
        Calendar.SATURDAY,
        Calendar.SUNDAY
    };
    ScheduleIterator i = new CompositeIterator(
        new ScheduleIterator[] {
            new RestrictedDailyIterator(8, 0, 0, weekdays),
            new RestrictedDailyIterator(9, 0, 0, weekend)
        }
    );
```

RestrictedDailyIteratorクラスはDailyIteratorと似ていますが、RestrictedDailyIteratorクラスは週のある特定の何日かしか走らないようになっています。また、CompositeIteratorクラスはScheduleIterators一式を日にちの順に正しく並べ、一つのスケジュールとしてまとめます。これらのクラスのソースコードについては参考文献を見てください。

cronでは生成できないけれども、ScheduleIteratorを実装することでならスケジュールできることが他にもたくさんあります。例えば「月の最終日」と記述されたスケジュールは標準のJavaのカレンダー計算（Calendarクラスを使って）で実現できますが、これをcronで表現するのは不可能です。アプリケーションではCalendarクラスを使う必要さえありません。この記事のソースコード（参考文献）に、「日没の15分前点灯する」ようにスケジュールした防犯ライトのコントローラーの例を入れておきました。これには（経緯度から）その地域での日没時間の計算にCalendrical Calculations Software Package（カレンダー計算ソフトウェア・パッケージ:参考文献）を使っています。

実時間保証

スケジューリングを使ったアプリケーションを書く場合、スケジューリングのフレームワークが指定時間通りの動作についてどういう規則で動くか理解しておくことが重要です。このタスクは早め、または遅めに実行される可能性があるのか？そうだとしたら最大どれだけの時間ずれが生じるのか？残念ながら答えは簡単ではありません。ただ現実的に、大きなアプリケーションのクラスでのタスクの振る舞いは問題の無いレベルと言えます。以下の説明ではシステム・クロックは正しいと言う前提です。（Network Time Protocol：ネットワーク時間プロトコル については参考文献を見てください）

SchedulerはスケジューリングをTimerクラスに任せるので、Schedulerによる実時間保証はTimerの実時間保証と同じです。TimerはObject.wait(long)メソッドを使ってタスクをスケジューリングします。次のどれかによってウェイクアップされるまで現在のスレッドは待たされます。

1. そのオブジェクトのnotify()またはnotifyAll()メソッドが別スレッドに呼ばれる。
2. 現在のスレッドが別スレッドにインタラプトされる。

3. 通知が無いので現在のスレッドがウェイクアップされる（ 参考文献のJoshua Bloch著Effective Java Programming Language Guide の50項に説明されているように、スプリアス・ウエィクアップとして知られています ）。
4. 指定されただけの時間が経過した。

1番目の可能性は`Timer`クラスには起こりえません。これは`wait()`が呼ばれるオブジェクトはプライベートなためです。それでも`Timer`実装では初めから3番目までの原因で早めにウェイクアップされることがないようにしてあり、その結果スレッドがウェィクアップされるのは時間が過ぎてからになるよう確実になっています。ところで`Object.wait(long)`のコメントの記述には「大体において」時間が過ぎてからウェイクアップする、とあるので、このスレッドが早めにウェイクアップすることもあり得ます。その場合には`Timer`はもう一つの`wait()`を発行して`#scheduledExecutionTime - System.currentTimeMillis()#`ミリ秒待つようにし、タスクが早めに実行されることが無いように保証しています 。

タスクが遅めに実行されることはあり得るのか？あり得ます。遅めに実行される要因には2つあります。スレッド・スケジューリングとガーベジ・コレクションです。

Java言語仕様はスレッド・スケジューリングに関して意図的にあいまいになっています。これはJavaプラットフォームというのが汎用で、対象とするハードウェアやオペレーティング・システムが多岐にわたるためです。大部分のJVM実装にはそれなりのスレッド・スケジューラーがありますが、そのスケジューラーが保証されているわけではありません。これはJVM実装によってプロセッサー時間の割り付けに異なる方針があるので当然とも言えます。ですから`Timer`スレッドが指定時間経過後にウェイクアップした時、実際にそのタスクを実行する時間はJVMのスレッド・スケジューリングの仕方、またプロセッサー時間が競合しているスレッドがいくつあるかに依存することになります。ですからタスク実行の遅れを減らすには、そのアプリケーションで走る可能性のあるスレッドの数を最小限にする必要があります。それを実現するのに別のJVMでスケジューラーを走らせることを検討するのも無駄ではありません。

オブジェクトをいくつも生成する大きなアプリケーションでは、JVMがガーベジ・コレクション（ GC ）に費やす時間もばかになりません。デフォルトで、GCが起きるとそのアプリケーション全体がGCの終了まで待たされますが、これには数秒またはそれ以上かかります。（ `java`アプリケーション・ランチャーのコマンドラインのオプション`-verbose:gc`で各GCイベントをコンソールにレポートするようにできます。 ）GCによる中断で迅速なタスクの実行が妨げられますが、この中断を最小にするにはアプリケーションが生成するオブジェクトの数を最小限にする必要があります。繰り返しますが、スケジューリングのコードを別のJVMで実行するのも一つの方法です。また、GCによる中断を最小限にする調整オプションもいくつかあるので、それらを試すのも一つです。例えばインクリメンタルGCでは、大がかりなコレクションをいくつかの小規模なコレクションに分散することで影響を小さくしています。その代わりGCの効率は下がりますが、タイマー・スケジューリングのことを考えれば代償としては高くはないと言えるかもしれません。（ GC調整のヒントについては参考文献を見てください。 ）

いつすることになっていたっけ？

タスクが時間通り動いているかどうか判断できるように、タスクが自身の実行を監視し、遅れた場合には記録するようになっていると便利です。`SchedulerTask`には`TimerTask`と同じく`scheduledExecutionTime()`メソッドがあり、このメソッドがタスクが実行されるはずであった

一番最近の時間を返します。式`System.currentTimeMillis() - scheduledExecutionTime()`をタスクの`run()`メソッドの初めに評価することで、タスクの実行がどのくらい遅れたかがミリ秒単位で分かります。この値のログをとることで実行遅れの統計的な分布を知ることもできます。またこの値によって、タスクが何をすべきなのかを決めることもできます。例えば、タスクが遅れすぎた場合には何もしない、など。こうした指針に従った上で、アプリケーションがより厳密に時間通り動作することが必要な場合には、Javaの実時間仕様を検討してみてください（詳しい情報については参考文献を見てください）。

まとめ

この記事ではJavaタイマー・フレームワークを簡単に機能向上させ、非常に柔軟なスケジュールができるようにする方法を紹介しました。この新しいフレームワークは実は`cron`の一般化そのものです。実際`cron`を`ScheduleIterator`インターフェースとして実装し、純然たるJava`cron`の置き換えにできれば便利です。このフレームワークには厳密な実時間保証はありませんが、定常的にタスクをスケジュールする必要のある、一般的なJavaアプリケーションに適用できます。

著者について

Tom White

Tom WhiteはイギリスKizoomのLead Java Developer。Kizoomはモバイル・デバイスに対し、各ユーザーに合わせたトラベル情報を提供するソフトウェア会社として最先端を行っており、顧客にはイギリスの国有鉄道、ロンドンの公衆運輸システム、国有バス会社などがあります。Kizoomは1999年の設立以来、極限プログラミングのあらゆる規律を使いこなしています。Tom Whiteは1996年からフルタイムでJavaプログラムを書いていますが、スタンダード及びエンタープライズJava APIの大部分を使い、クライアントのSwing GUIやグラフィックスからバックエンドのメッセージング・システムまでを書いています。ケンブリッジ大学で数学の一級名誉号を取得。プログラミングをしているとき以外には幼い娘を笑わせ、1930年代のハリウッド映画を楽しんでいます。連絡先はtom-at-tiling.org。