

並列実行のための Java アクター・ライブラリー

軽量の Java アクター・ライブラリー `µjavaActors` を使用して、一般的な並行処理パターンを最新の状況に合うようにする

Barry A. Feigenbaum, Ph.D.
Software Engineer
Dell

2012年 6月 28日

現在のところ、Java プラットフォームはアクターによる並行処理をサポートしていませんが、それでもアクターを Java コードで使用することは可能です。この記事では、著者の Barry Feigenbaum が従来の Java アプリケーションで高度な並列実行を実現可能にする軽量の Java ベースのアクター・パッケージ、`µjavaActors` ライブラリーを紹介します。このチュートリアルには、`µjavaActors` ライブラリーの完全なソース・コードと、Command、Producer/Consumer、Map/Reduce などの Java スタンバイ・パターンにアクターを使用する実践例の完全なソース・コードが付属しています。

アクターを使うべきか、使うべきでないか、それが問題です！

Java 6 とそれに続く Java 7 で並行処理が更新されたとは言え、Java 言語での並列プログラミングはそれほど容易ではありません。Java スレッド、`synchronized` ブロック、`wait/notify` メソッド、そして `java.util.concurrent` パッケージはすべてその役割を果たすものの、マルチコア・システムの能力をフルに活用することを迫られている Java 開発者たちは、他の言語で新たに開発された手法へと転換しつつあります。アクター・モデルはそのような新しい手法の 1 つで、Erlang、Groovy、および Scala で実装されます。アクターを試してみたいけれど、コードの作成はやはり Java で行いたいという開発者のために、この記事では `µjavaActors` ライブラリーを紹介します。

JVM に対応した他の 3 つのアクター・ライブラリー

JVM でよく使われる 3 つのアクター・ライブラリーを `µjavaActors` と比較して簡単に紹介した表 1 を参照してください。

`µjavaActors` ライブラリーは、Java プラットフォームにアクター・ベースのシステムを実装するコンパクトなライブラリーです（「`µ`」は、「マイクロ」を意味するギリシャ文字の `Mμ` を表します）。この記事では `µjavaActors` を使用して、Producer/Consumer や Map/Reduce などの一般的なデザイン・パターンにおいてアクターがどのように機能するかを明らかにします。

µjavaActors ライブラリーのソース・コードは、いつでも[ダウンロード](#)することができます。

Java プラットフォームでのアクターによる並行処理

名前にいったい何の意味があるのでしょうか？呼び名はどうであれ、アクターの動作に変わりはありません！

アクター・ベースのシステムは、メッセージ・パッシング・スキームを実装することにより、並列処理のコーディングを容易にします。メッセージ・パッシング・スキームでは、システム内の各アクターがメッセージを受信し、メッセージで要求されているアクションを実行し、自身を含む他のアクターにメッセージを送信して、操作の複合シーケンスを実行することができます。アクターの間では、常に非同期でメッセージが受け渡されます。つまり、送信側は応答の受信を待たずに処理を続行できるため、アクターはその存続期間をひたすらメッセージを受信して処理する作業の繰り返しに費やすことができます。

複数のアクターが使用される場合には、独立したアクティビティを簡単に複数のスレッド（したがって複数のプロセッサ）に分散することができ、これらのスレッドでメッセージを並列に処理することができます。一般に、各アクターはそれぞれ別個のスレッドでメッセージを処理するため、複数のアクターが使用される場合には、アクターの数だけ並列に処理することができます。アクター・システムのなかには、スレッドを静的にアクターに割り当てるものもあれば、この記事で紹介するシステムのように、動的に割り当てるものもあります。

µjavaActors の紹介

µjavaActors は、アクター・システムの単純な Java 実装です。約1,200 行のコードに収まっているものの、µjavaActors は強力です。以降に記載する演習で、µjavaActors を使用してアクターを動的に作成および管理し、アクターにメッセージを配信する方法を学びます。

µjavaActors は、以下に記載する Message、ActorManager、Actor という 3 つのコア・インターフェースを中心に構築されています。

- **Message** は、アクター間で送信されるメッセージです。Message には、以下の 3 つの (オプションの) 値と何らかの動作が格納されます。
 - `source`: 送信側アクター
 - `subject`: メッセージの意味を定義するストリング (コマンドとも呼ばれます)。
 - `data`: メッセージの任意のパラメーター・データ (大抵は、マップ、リスト、または配列です)。パラメーターには、処理対象のデータや、相互作用する他のアクターを指定することができます。
 - `subjectMatches()`: メッセージ・サブジェクトがストリングまたは正規表現と一致するかどうかをチェックします。

µjavaActors パッケージのデフォルト・メッセージ・クラスは `DefaultMessage` です。

- **ActorManager** は、アクターのマネージャーです。その役割は、メッセージを処理するためにスレッド（したがってプロセッサ）をアクターに割り当てることです。ActorManager には以下の重要な動作または特性があります。
 - `createActor()`: アクターを作成し、そのアクターをこのマネージャーに関連付けます。

- `startActor()`: アクターを起動します。
- `detachActor()`: アクターを停止し、アクターとこのマネージャーとの関連付けを解除します。
- `send()/broadcast()`: メッセージをアクター、アクターのセット、カテゴリに対応する任意のアクター、またはすべてのアクターに送信します。

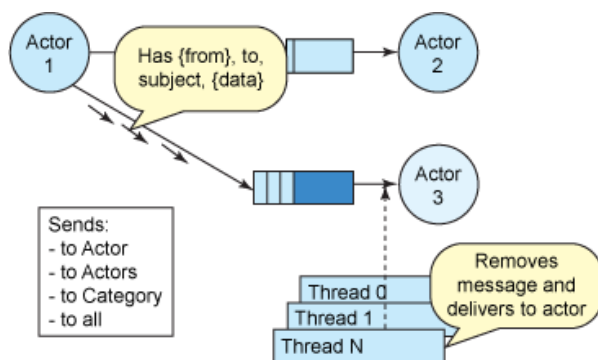
ほとんどのプログラムには1つの `ActorManager` しかありませんが、複数のスレッド・プールやアクター・プールを管理する必要がある場合には、複数の `ActorManager` を使用することができます。このインターフェースのデフォルト実装は `DefaultActorManager` です。

- `Actor` は実行単位であり、一度に1つのメッセージを処理します。`Actor` には以下の重要な動作または特性があります。
 - 各アクターの名前 (`name`) は、`ActorManager` のなかでは一意でなければなりません。
 - アクターは、それぞれが1つのカテゴリ (`category`) に属します。カテゴリとは、アクターのグループの1メンバーにメッセージを送信する手段です。アクターは同時に複数のカテゴリに属することはできません。
 - `receive()`: このメソッドが呼び出されるのは、このメソッドの親アクターを実行するスレッドを `ActorManager` が提供できる状態にあって、このアクター宛てのメッセージがある場合に限られます。最大限の効率を達成するためには、アクターが素早くメッセージを処理すると同時に、(例えば人間による入力を待機するなど) 長時間の待機状態に入らないようにする必要があります。
 - `willReceive()`: このメソッドを呼び出すことで、親アクターは受信するメッセージのサブジェクトを指定することができます。
 - `peek()`: このメソッドを呼び出すことで、親アクターなどは (例えば、指定したサブジェクトのメッセージで) 処理待ちのメッセージがあるかどうかを調べることができます。
 - `remove()`: このメソッドを呼び出すことで、親アクターなどは、まだ処理されていないメッセージを削除またはキャンセルすることができます。
 - `getMessageCount()`: このメソッドを呼び出すことで、親アクターなどは処理待ちのメッセージ数を取得することができます。
 - `getMaxMessageCount()`: このメソッドを呼び出して返される結果を、システムでサポートする処理待ちメッセージの最大数として使用することで、親アクターは処理待ちメッセージの数を制限することができます。

大抵のプログラムには多数のアクターがあり、異なる種類のアクターが使用されることも珍しくありません。アクターは、プログラムの開始時に作成することも、プログラムの実行中に作成 (または破棄) することもできます。この記事に付属の [アクター・パッケージ](#) には、`AbstractActor` という名前の抽象クラスが含まれています。これが、アクター実装の基底クラスです。

図1に、アクター間の関係を示します。それぞれのアクターが、他のアクターにメッセージを送信することができます。メッセージは、メッセージ・キュー (別名「メール・ボックス」。概念的には、アクターごとに1つあります) に保持されます。`ActorManager` は、メッセージを処理するために使用できるスレッドがあることを認知すると、このメッセージ・キューからメッセージを取り出して、そのスレッドで実行されているアクターにメッセージが配信され、処理されます。

図 1. アクター間の関係



µjavaActors を使用した並列処理

何よりも、やってみることで！

これから、µjavaActors を使用した並列処理の実践演習に取り掛かります。まずは、一連のアクターを作成するところから始めます。作成するアクターは、短時間の遅延を設けて他のアクターにメッセージを送信するだけの単純なものです。これらのアクターによって、始めは大量のメッセージが作成されますが、メッセージの数は徐々に減っていき、最終的にはメッセージが作成されなくなります。以降に紹介するデモでは、最初にアクターの作成方法を説明してから、アクターが徐々にディスパッチされてメッセージを処理する仕組みを明らかにします。

メッセージには、以下の 2 つのタイプがあります。

- `initialization (init)`: このメッセージによって、アクターが初期化されます。このメッセージは、1 つのアクターにつき一度だけ送信されます。
- `repeat`: このメッセージによって、アクターは N-1 個のメッセージを送信します。ここで、N は受信したメッセージで指定されているパラメーターです。

リスト 1 に記載する `TestActor` クラスは、`AbstractActor` クラスから継承した抽象メソッドを実装します。`activate` メソッドと `deactivate` メソッドは、この例ではアクターにその存続期間を通知するだけで、それ以外のことは行いません。`runBody` メソッドは、アクターが最初に作成された後、メッセージを受信する前に呼び出されます。このメソッドは通常、アクターへの最初のメッセージのブートストラップに使用されます。アクターがメッセージを受信しようとする時点で、`testMessage` メソッドが呼び出されます。ここで、アクターがメッセージを拒否するか、受け入れるかが決まります。この例では、アクターは基底クラスから継承した `testMessage` メソッドを使用して受け入れテストを行うことになるため、すべてのメッセージが受け入れられます。

リスト 1. TestActor

```
class TestActor extends AbstractActor {

    @Override
    public void activate() {
        super.activate();
    }

    @Override
    public void deactivate() {
        super.deactivate();
    }
}
```

```
@Override
protected void runBody() {
    sleeper(1); // delay up to 1 second
    DefaultMessage dm = new DefaultMessage("init", 8);
    getManager().send(dm, null, this);
}

@Override
protected Message testMessage() {
    return super.testMessage();
}
```

アクターがメッセージを受信すると、リスト 2 に記載する `loopBody` メソッドが呼び出されます。このメソッドでは、汎用処理をシミュレートするための短時間の遅延の後で、メッセージが処理されます。メッセージが「repeat」の場合、アクターは `count` パラメーターに基づいて、`count - 1` 個のメッセージの送信プロセスを開始します。アクター・マネージャーの `send` メソッドの呼び出しによって、メッセージは無作為にアクターに送信されます。

リスト 2. `loopBody()`

```
@Override
protected void loopBody(Message m) {
    sleeper(1);
    String subject = m.getSubject();
    if ("repeat".equals(subject)) {
        int count = (Integer) m.getData();
        if (count > 0) {
            DefaultMessage dm = new DefaultMessage("repeat", count - 1);
            String toName = "actor" + rand.nextInt(TEST_ACTOR_COUNT);
            Actor to = testActors.get(toName);
            getManager().send(dm, this, to);
        }
    }
}
```

メッセージが「init」の場合、アクターは `repeat` メッセージ・シーケンスを開始するために、無作為に選択したアクター、あるいは `common` カテゴリのアクターに、メッセージの 2 つのセットを送信します。一部のメッセージは即時に処理することができます (実際のところ、スレッドが使用可能であれば、アクターがメッセージを受信可能な状態になったと同時に処理されます)。それ以外のメッセージは、数秒間待機してからでないと実行できません。このような遅延メッセージ処理は、この例では重要ではありませんが、長時間実行されるプロセスのポーリングを実装するために使用することができます。そうしたプロセスの例としては、ユーザーからの入力を待機するプロセスや、ネットワーク要求に対する応答が到着するまで待機するプロセスなどが挙げられます。

リスト 3. 初期化シーケンス

```
else if ("init".equals(subject)) {
    int count = (Integer) m.getData();
    count = rand.nextInt(count) + 1;
    for (int i = 0; i < count; i++) {
        DefaultMessage dm = new DefaultMessage("repeat", count);
        String toName = "actor" + rand.nextInt(TEST_ACTOR_COUNT);
        Actor to = testActors.get(toName);
        getManager().send(dm, this, to);

        dm = new DefaultMessage("repeat", count);
        dm.setDelayUntil(new Date().getTime() + (rand.nextInt(5) + 1) * 1000);
        getManager().send(dm, this, "common");
    }
}
```

メッセージが適切なタイプでない場合には、エラーがレポートされます。

```
else {
    System.out.printf("TestActor:%s loopBody unknown subject: %s\n",
        getName(), subject);
}
}
```

main プログラムには、リスト 4 に記載するコードが格納されています。このコードは、common カテゴリの 2 つのアクターを作成し、default カテゴリの 5 つのアクターを作成した後、これらのアクターを起動します。その後、main は定期的に進行状況メッセージを表示しながら、最大 120 秒間待機します (sleeper は、渡された引数の値に 1000ms を掛けた時間まで待機します)。

リスト 4. createActor、startActor

```
DefaultActorManager am = DefaultActorManager.getDefaultInstance();
:
Map<String, Actor> testActors = new HashMap<String, Actor>();
for (int i = 0; i < 2; i++) {
    Actor a = am.createActor(TestActor.class, "common" + i);
    a.setCategory("common");
    testActors.put(a.getName(), a);
}
for (int i = 0; i < 5; i++) {
    Actor a = am.createActor(TestActor.class, "actor" + i);
    testActors.put(a.getName(), a);
}
for (String key : testActors.keySet()) {
    am.startActor(testActors.get(key));
}
for (int i = 120; i > 0; i--) {
    if (i < 10 || i % 10 == 0) {
        System.out.printf("main waiting: %d...\n", i);
    }
    sleeper(1);
}
:
am.terminateAndWait();
```

トレース出力

上記で実行したプロセスを理解するために、アクターからのトレース出力を見てみましょう (repeat のカウント値と遅延時間の値にはランダムな値が使用されているため、出力の内容は実行

ごとに異なることに注意してください)。リスト 5 には、プログラムが開始されたあたりで生成されたメッセージが示されています。左側の列 (角括弧内) に示されているのは、実行中のスレッドの名前です。このテスト・ランでは、メッセージを処理するために 25 のスレッドが使用可能になっていました。行の残りの部分は、(要約された) トレース出力です。ここに、アクターが受信した各メッセージが示されています。repeat カウント (つまり、パラメーター・データ) が次第に減っていることに注目してください (スレッドの名前は actor で始まっていますが、アクターの名前とは関係ないことにも注意してください)。

リスト 5. トレース出力: プログラムの開始

```
[main      ] - main waiting: 120...
[actor17   ] - TestActor:actor4 repeat(4)
[actor0    ] - TestActor:actor1 repeat(4)
[actor10   ] - TestActor:common1 repeat(4)
[actor1    ] - TestActor:actor2 repeat(4)
[actor3    ] - TestActor:actor0 init(8)
[actor22   ] - TestActor:actor3 repeat(4)
[actor17   ] - TestActor:actor4 init(7)
[actor20   ] - TestActor:common0 repeat(4)
[actor24   ] - TestActor:actor0 repeat(4)
[actor0    ] - TestActor:actor1 init(3)
[actor1    ] - TestActor:actor2 repeat(4)
[actor20   ] - TestActor:common0 repeat(4)
[actor17   ] - TestActor:actor4 repeat(4)
[actor17   ] - TestActor:actor4 repeat(3)
[actor0    ] - TestActor:actor1 repeat(8)
[actor10   ] - TestActor:common1 repeat(4)
[actor24   ] - TestActor:actor0 repeat(8)
[actor0    ] - TestActor:actor1 repeat(8)
[actor24   ] - TestActor:actor0 repeat(7)
[actor22   ] - TestActor:actor3 repeat(4)
[actor1    ] - TestActor:actor2 repeat(3)
[actor20   ] - TestActor:common0 repeat(4)
[actor22   ] - TestActor:actor3 init(5)
[actor24   ] - TestActor:actor0 repeat(7)
[actor10   ] - TestActor:common1 repeat(4)
[actor17   ] - TestActor:actor4 repeat(8)
[actor1    ] - TestActor:actor2 repeat(3)
[actor17   ] - TestActor:actor4 repeat(8)
[actor0    ] - TestActor:actor1 repeat(8)
[actor10   ] - TestActor:common1 repeat(4)
[actor22   ] - TestActor:actor3 repeat(8)
[actor0    ] - TestActor:actor1 repeat(7)
[actor1    ] - TestActor:actor2 repeat(3)
[actor0    ] - TestActor:actor1 repeat(3)
[actor20   ] - TestActor:common0 repeat(4)
[actor24   ] - TestActor:actor0 repeat(7)
[actor24   ] - TestActor:actor0 repeat(6)
[actor10   ] - TestActor:common1 repeat(8)
[actor17   ] - TestActor:actor4 repeat(7)
```

リスト 6 に、プログラムが終了するあたりで生成されたメッセージを示します。このあたりになると、repeat のカウント値はさらに少なくなっています。このプログラムの実行状況を観察していたとすると、行の生成速度が次第に遅くなっていくことに気付くはずです。これは、生成されるメッセージの数が徐々に減っていくためです。十分な待ち時間が与えられている場合、アクターに送信されるメッセージは完全になくなります (これは、リスト 6 に記載されている common アクターに示されているとおりです)。メッセージ処理は使用可能なスレッドの間で適度に分散されるため、特定のアクターが特定のスレッドにバインドされることはありません。

リスト 6. トレース出力: プログラムの終了

```
[main      ] - main waiting: 20...
[actor0    ] - TestActor:actor4 repeat(0)
[actor2    ] - TestActor:actor2 repeat(1)
[actor3    ] - TestActor:actor0 repeat(0)
[actor17   ] - TestActor:actor4 repeat(0)
[actor0    ] - TestActor:actor1 repeat(2)
[actor3    ] - TestActor:actor2 repeat(1)
[actor14   ] - TestActor:actor1 repeat(2)
[actor5    ] - TestActor:actor4 repeat(0)
[actor14   ] - TestActor:actor2 repeat(0)
[actor21   ] - TestActor:actor1 repeat(0)
[actor14   ] - TestActor:actor0 repeat(1)
[actor14   ] - TestActor:actor4 repeat(0)
[actor5    ] - TestActor:actor2 repeat(1)
[actor5    ] - TestActor:actor4 repeat(1)
[actor6    ] - TestActor:actor1 repeat(1)
[actor5    ] - TestActor:actor3 repeat(0)
[actor6    ] - TestActor:actor2 repeat(1)
[actor4    ] - TestActor:actor0 repeat(0)
[actor5    ] - TestActor:actor4 repeat(1)
[actor12   ] - TestActor:actor1 repeat(0)
[actor20   ] - TestActor:actor2 repeat(2)
[main      ] - main waiting: 10...
[actor7    ] - TestActor:actor4 repeat(2)
[actor23   ] - TestActor:actor1 repeat(0)
[actor13   ] - TestActor:actor2 repeat(1)
[actor8    ] - TestActor:actor0 repeat(0)
[main      ] - main waiting: 9...
[actor2    ] - TestActor:actor1 repeat(0)
[main      ] - main waiting: 8...
[actor7    ] - TestActor:actor2 repeat(0)
[actor13   ] - TestActor:actor1 repeat(0)
[main      ] - main waiting: 7...
[actor2    ] - TestActor:actor2 repeat(2)
[main      ] - main waiting: 6...
[main      ] - main waiting: 5...
[actor18   ] - TestActor:actor1 repeat(1)
[main      ] - main waiting: 4...
[actor15   ] - TestActor:actor2 repeat(0)
[actor16   ] - TestActor:actor1 repeat(1)
[main      ] - main waiting: 3...
[main      ] - main waiting: 2...
[main      ] - main waiting: 1...
[actor4    ] - TestActor:actor1 repeat(0)
[actor6    ] - TestActor:actor2 repeat(0)
```

シミュレーションのスクリーン・ショット


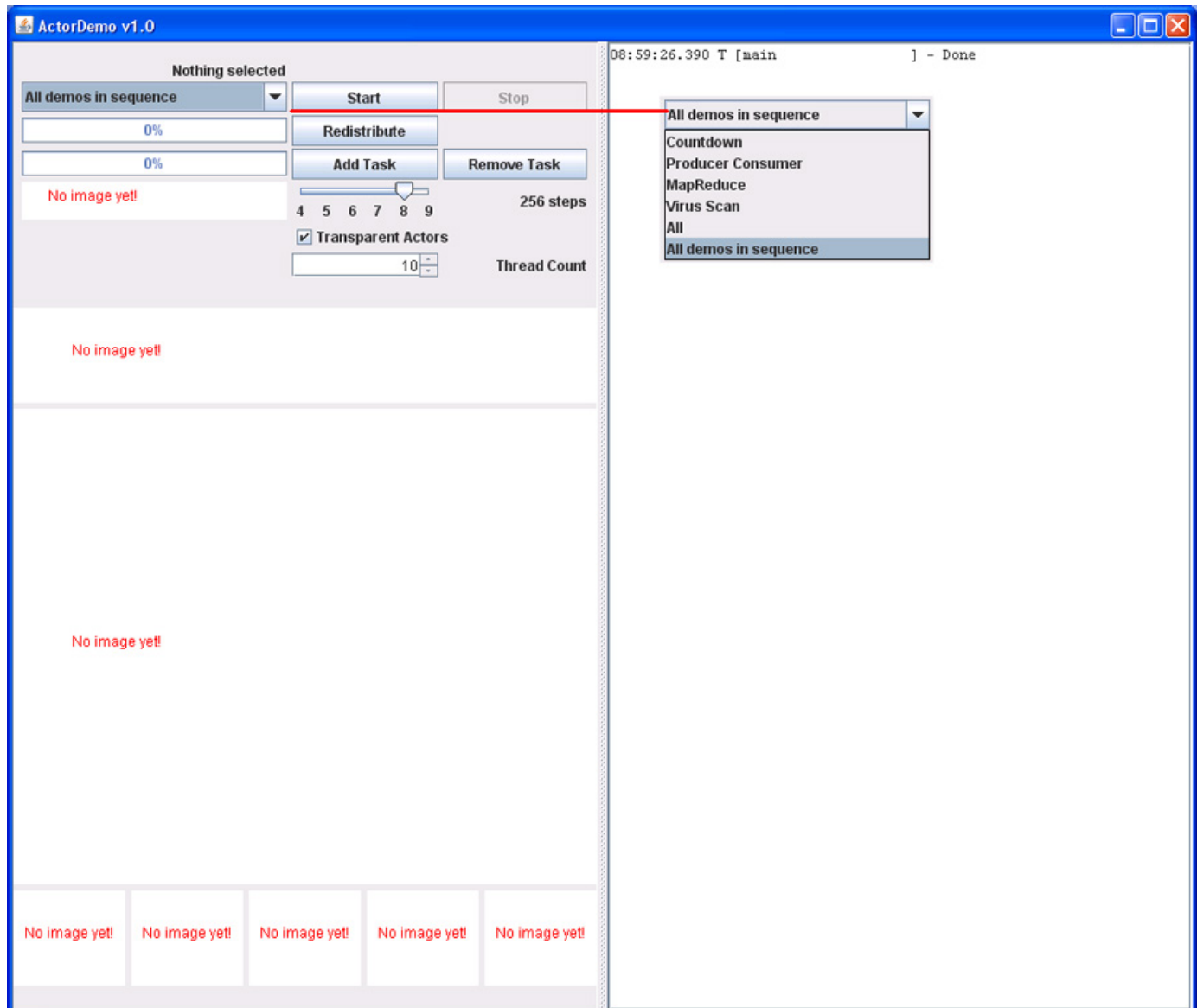
上記のトレース出力からアクター・システムがどのように動作するのかを完全に把握するのは、そう簡単なことではありません。それは一部に、トレース形式ではそれだけの情報が得られないためです。同様のアクター・シミュレーションを実行したときのスナップショット画像からは、同じ情報をグラフィック形式で確認することができます。スナップショットの各画像が、一定の時間が経過した後のシミュレーション結果を示します。以下の動画には、サンプル・コードおよびスクリーン・ショットでは取り込んでいない一部の Java アクター・プロセスが示されています。この動画は、以下のインラインで見えることも、[YouTube](#) で見えることもできます。YouTube には、インタラクティブな字幕を表示する機能が用意されているので、特定の時間を選択して、その部分の説明を読むことができます。この機能は、 動画画面の下にあるアイコンをクリックするだけで有効になります。

図 2 に、シミュレーションを実行する前のシミュレーションのユーザー・インターフェースを示します。右側に表示されているシミュレーション・メニューの内容に注目してください。

図 2. シミュレーションを実行する前のアクター・シミュレーター



画面の左側上部には、シミュレーション・メニューが表示されます。シミュレーションにはいくつかのバリエーションが考えられますが、特に断りのない限り、この記事のトレース出力および以降のスクリーン・ショットは以下のシミュレーションによるものです。

- 「Countdown (カウントダウン)」シミュレーション (動画では 0:15 あたりから開始): ゼロまでカウントダウンしてから次のリクエストを送信するアクターをシミュレーションします。
- 「Producer Consumer (プロデューサー/コンシューマー)」シミュレーション (動画では 2:40 あたりから開始): 従来の Producer/Consumer 並行性問題に基づき、そのバリエーションをシミュレーションします。

- 「MapReduce」シミュレーション (動画では 5:28 あたりから開始): 1000 個の整数の二乗和の並列実行をシミュレーションします。
- 「Virus Scan (ウィルス・スキャン)」シミュレーション (動画では 6:45 あたりから開始): ディスクのディレクトリー・ツリーで「.txt」ファイルをスキャンし (スキャンする数を制限するため)、疑わしいコンテンツ・パターンを検出します。この非 CPU バウンド・シミュレーションは、以降のスクリーン・ショットでは示されていませんが、これも動画デモの一部です。
- 「All (すべて)」のシミュレーション (動画では 8:18 あたりから開始): すべてのシミュレーションを並行して実行します。

動画デモでは、「All demos in sequence (すべてのデモを順番に実行)」シミュレーションを実行することで、各シミュレーションの間には短い間隔を空け、これらすべてのシミュレーションを順に実行しています。

図 2 のスクリーン・ショットには、「Start (開始)」ボタンと「Stop (停止)」ボタンの他に、以下のコントロールと設定が示されています (「Stop (停止)」はスレッドを停止するわけではありません。したがって、このボタンをクリックした後に何らかのアクションが行われる場合もあります)。

- 「Redistribute (再分配)」は、アクター・サークル内のアクターをほぼランダムに再分配します (デフォルトの順序は、作成順です)。これにより、アクターが再配置されることで、グループ化された近隣のアクター間でのメッセージを確認しやすくなります。また、アクターに新しい色を割り当てることもできます。
- 「Add Task (タスクの追加)」と「Remove Task (タスクの削除)」は、それぞれ開始プールのタスク (スレッド) を追加、削除します。「Remove Task (タスクの削除)」によって削除されるタスクは、追加された (当初はなかった) タスクだけです。
- 最大ステップ数 (設定した値に対して [log2](#) 演算をした結果) は、シミュレーションの長さを制限します。この値は、シミュレーションの開始前にのみ設定することができます。ステップの長さは、約 1 秒間です。
- 「Transparent Actors (透明アクター)」を選択すると、隣接するアクター間のメッセージを確認しやすくなります。一方で、アクターを不透明にしたほうが見やすい場合はよくあります。この設定は、シミュレーションの実行中に変更することができます。
- 「Thread Count (スレッド数)」スピナーは、シミュレーションの開始前にのみ有効です。スレッド数を増やすと、多数のシミュレーションを実行する時間が大幅に短縮されます。

デモを選択する左上のコントロールの下にある表示ブロックには、現在のスレッド使用状況 (過去 1 秒間の平均として) 示されます。シミュレーションは、左側中央の大きな部分を占めるエリアに表示されます。その下にあるエリアに表示されるのは、シミュレーション履歴です。右側のエリアにはシミュレーション・トレース全体が表示されます。実行中のシミュレーション・フレームは、以下のように構成されます。

- コントロール・エリア内の以下の測定値は、約 1 秒ごとに更新されます。
 - 1 秒あたりの受け入れメッセージ数
 - 1 秒あたりの処理完了メッセージ数
 - 1 秒あたりの受け入れメッセージ数と処理完了メッセージ数の差

この横長のバーの右側部分に活発な動きが見られる場合は、処理されているメッセージよりも到着するメッセージのほうが多いことを意味します。この場合、最終的にはメッセージ・バッファがオーバーフローします。バーの左側部分に動きが見られる場合は、到着するメッセージよりも処理されているメッセージのほうが多いということであり、システムは最終的にアイドル状態になります。バランスの取れたシステムでは、このような動きが見られないか、長期間にわたって緑色でのみレベルが示されます。

- 左側中央のエリアの上には、複数の縦長のバーで構成されるグリッドがあります。各バーが1つのスレッドを表します(スレッドについては、次の項目で説明してあります)。バーが緑色で塗りつぶされている場合は、スレッドが完全に使用されていることを意味し、黄色で塗りつぶされている場合はスレッドが完全にアイドル状態であることを意味します。
- 左側中央のエリアで環状に配置されている一連の四角形は、スレッドを表します(ここで説明しているシミュレーションでのスレッド数は10です。前のトレースでは25個でした)。これらの緑色のスレッドがアクターに接続されて、受信したメッセージが実行されます。四角形の内側中央にある点の色は、アクターのタイプを示します。四角形の近くに示されている数字は、アクターの番号です(左を起点に0から360度まで時計回りで番号が付けられます)。黄色のスレッドはアイドル状態です。
- 左側中央のエリアで環状に配置されている一連の円は、アクターを表します。アクターのタイプは色によって示されます(この最初の例では、アクターのタイプは1つだけです)。アクターがメッセージの処理でビジー状態になっている場合には、濃い影で示されます(透明でないアクターを使用している場合には、影がより目立ちます)。円(アクター)同士を結ぶ線は、メッセージを表します。明るい赤の線は、所定のリフレッシュ・サイクル(このシミュレーションでは1秒間に10回リフレッシュします)期間中に新しく送信されたメッセージです。その他の色は、バッファリングされたメッセージ(過去に送信され、まだ処理されていないメッセージ)を表します。バッファリングされたメッセージを表す線には、受信側に小さな円が示されています。この円は、バッファリングされたメッセージの数が増えるにつれ、大きくなっていきます。
- 右側のエリアには、トレース出力が表示されます。これは前に記載したトレースと同様ですが、その内容はさらに詳しくなっています。
- 画面の左側下部には、環状部を小さく表示した一連の画像が並んでいます。このそれぞれの画像は、過去の一定期間ごとの環状部の表示を縮小したバージョンです。これらによって、メッセージのこれまでの傾向が簡単に見て取れます。この履歴を観察すれば、未処理のメッセージ数が短時間のうちに急増した後、徐々に減っていくことがわかります。

図3に、「Countdown (カウントダウン)」シミュレーションを約10秒間実行した時点の結果を示します。この短い時間で多数の処理待ちメッセージが溜まっていることに注目してください。アクターの数は34であるのに対し、スレッドはわずか10個です。したがって、一部のアクターはどうしてもアイドル状態になってしまいます。この時点では、すべてのスレッドがメッセージ処理を実行している状態です。

図 3. 実行開始後の「Countdown (カウントダウン)」シミュレーション (動画では 0:15 あたりから開始)

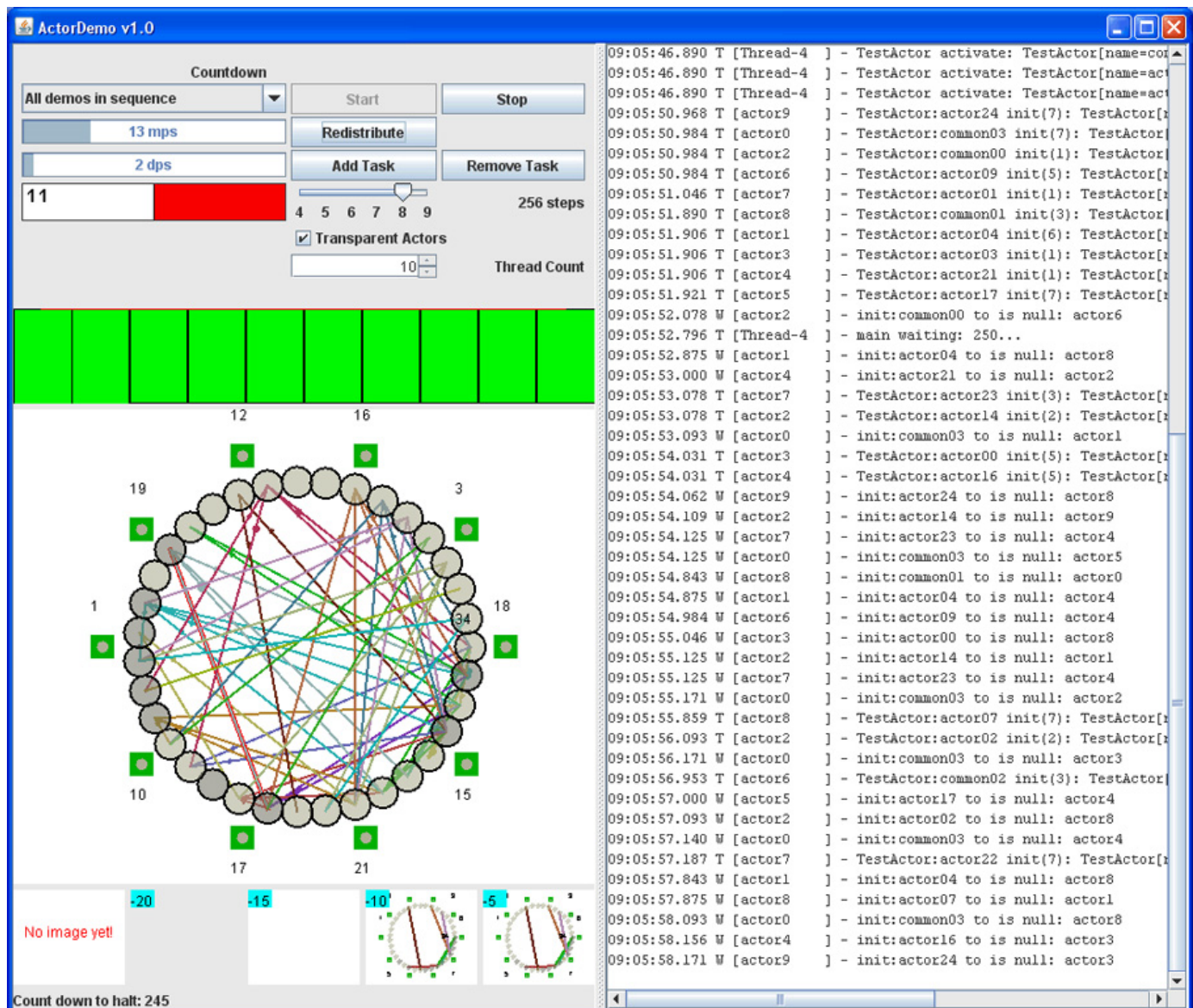


図 4 に、「Countdown (カウントダウン)」シミュレーションを約 30 秒間実行した時点の結果を示します。この時点で、処理待ちメッセージの数はかなり減っています。時間あたりに到着するメッセージ数が少なくなったことから、メッセージの処理で完全にビジー状態になっているスレッドは一部しかありません。

図 4. 「Countdown (カウントダウン)」 シミュレーション実行途中の結果

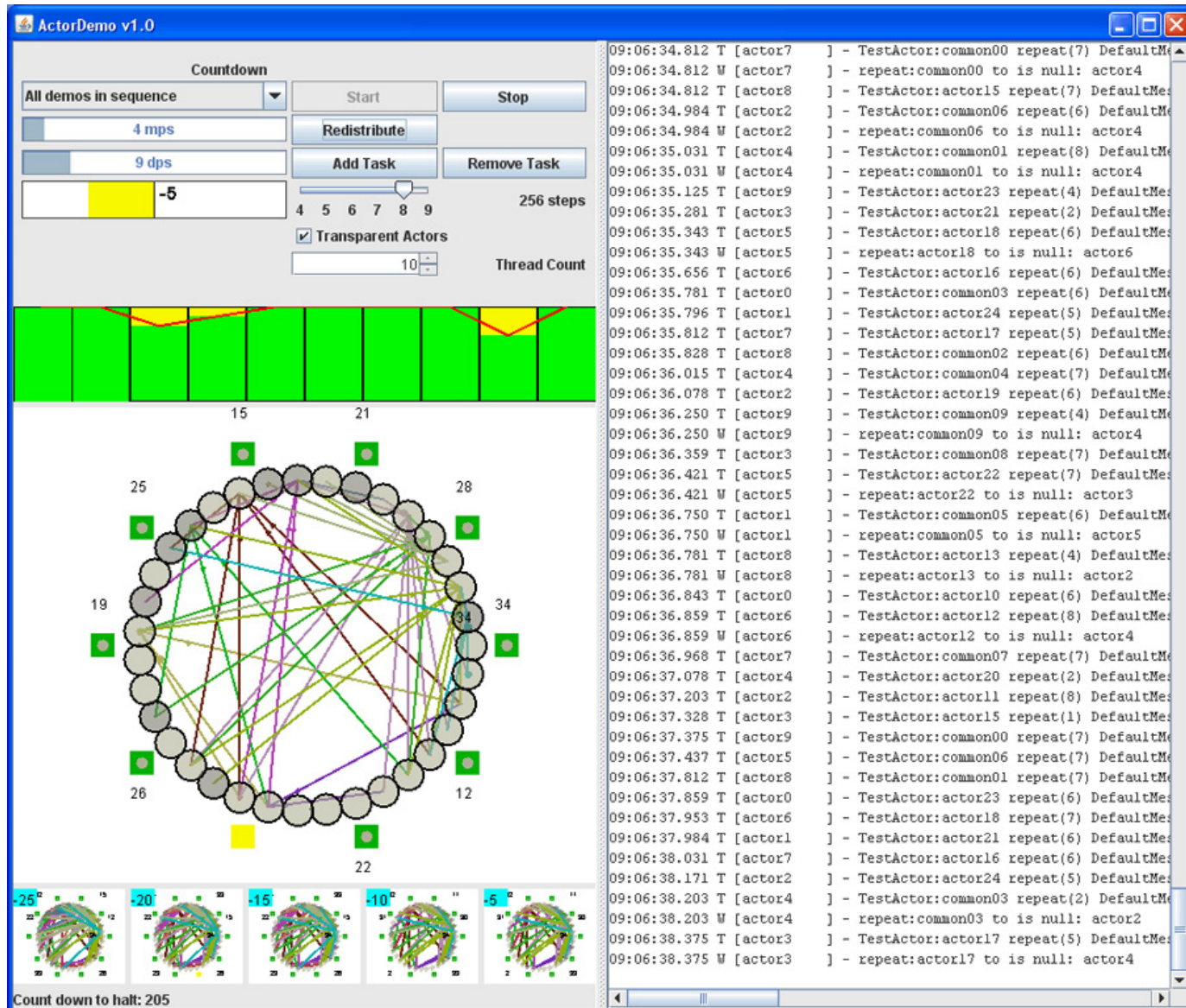
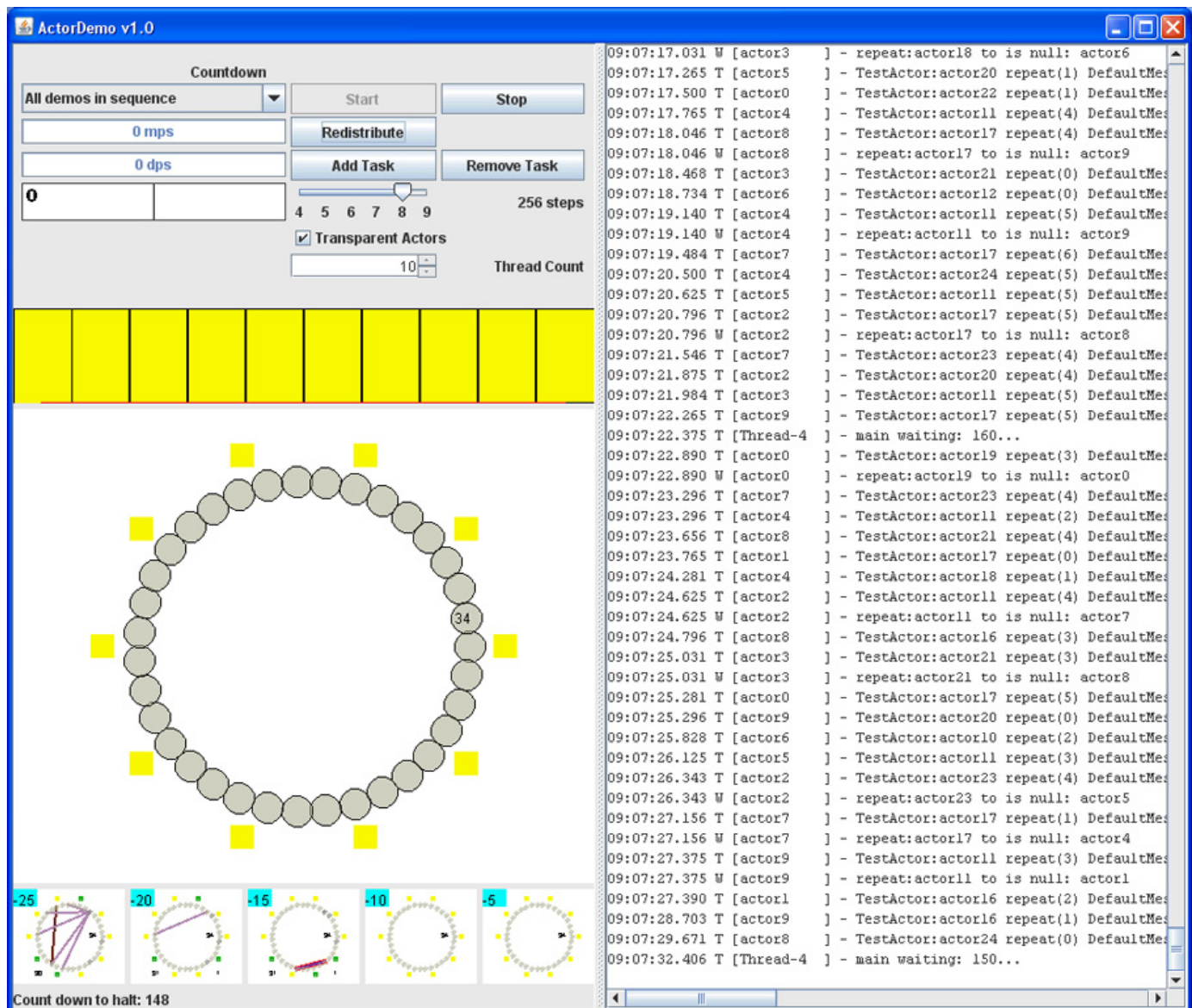


図 5 は、「Countdown (カウントダウン)」 シミュレーションを約 90 秒間実行した時点の結果です。処理待ちメッセージはすべて処理されたため、すべてのスレッドがアイドル状態になっています。

図 5. メッセージ処理完了時の「Countdown (カウントダウン)」シミュレーション



Producer/Consumer システムでのアクター

次は、Producer/Consumer パターンでのアクターのデモを見ていきましょう。Producer/Consumer は、マルチプロセッサ・システムで最も一般的な同期パターンの 1 つです。以降に取り上げる `µJavaActors` デモでは、プロデューサー・アクターがコンシューマー・アクターに対し、各種アイテムの作成を要求します。コンシューマーは要求されたアイテムを作成してから（これにはある程度の時間がかかります）、完了メッセージを要求側プロデューサーに送信します。

図 6 に、「Producer Consumer (プロデューサー/コンシューマー)」シミュレーションを約 30 秒間実行した時点の結果を示します。アクターの 2 つのタイプは、色で区別されていることに注意してください。実行を始めると、最初にプロデューサー・アクターが画面右下あたりに表示され、その後プロデューサーによってコンシューマーが作成されて表示されます。ワークロードは徐々に低下していくことから、スレッドはほとんどの時間ビジー状態になります。プロデューサーは

そのタスクを短時間で完了するため、めったにアクティブ状態として表示されることがありません。

図 6. 実行開始後の「Producer Consumer (プロデューサー/コンシューマー)」シミュレーション (動画では 2:40 あたりから開始)

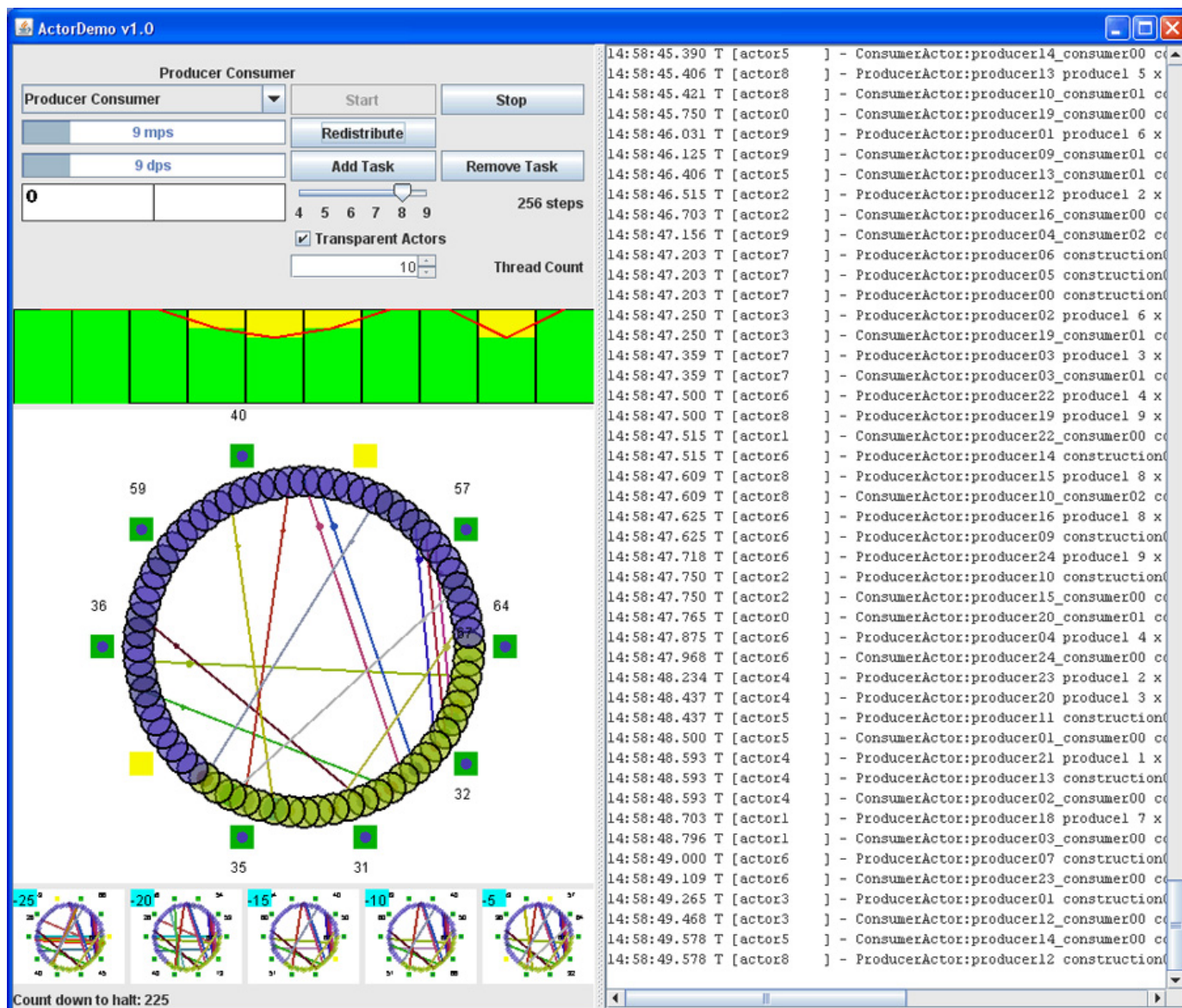
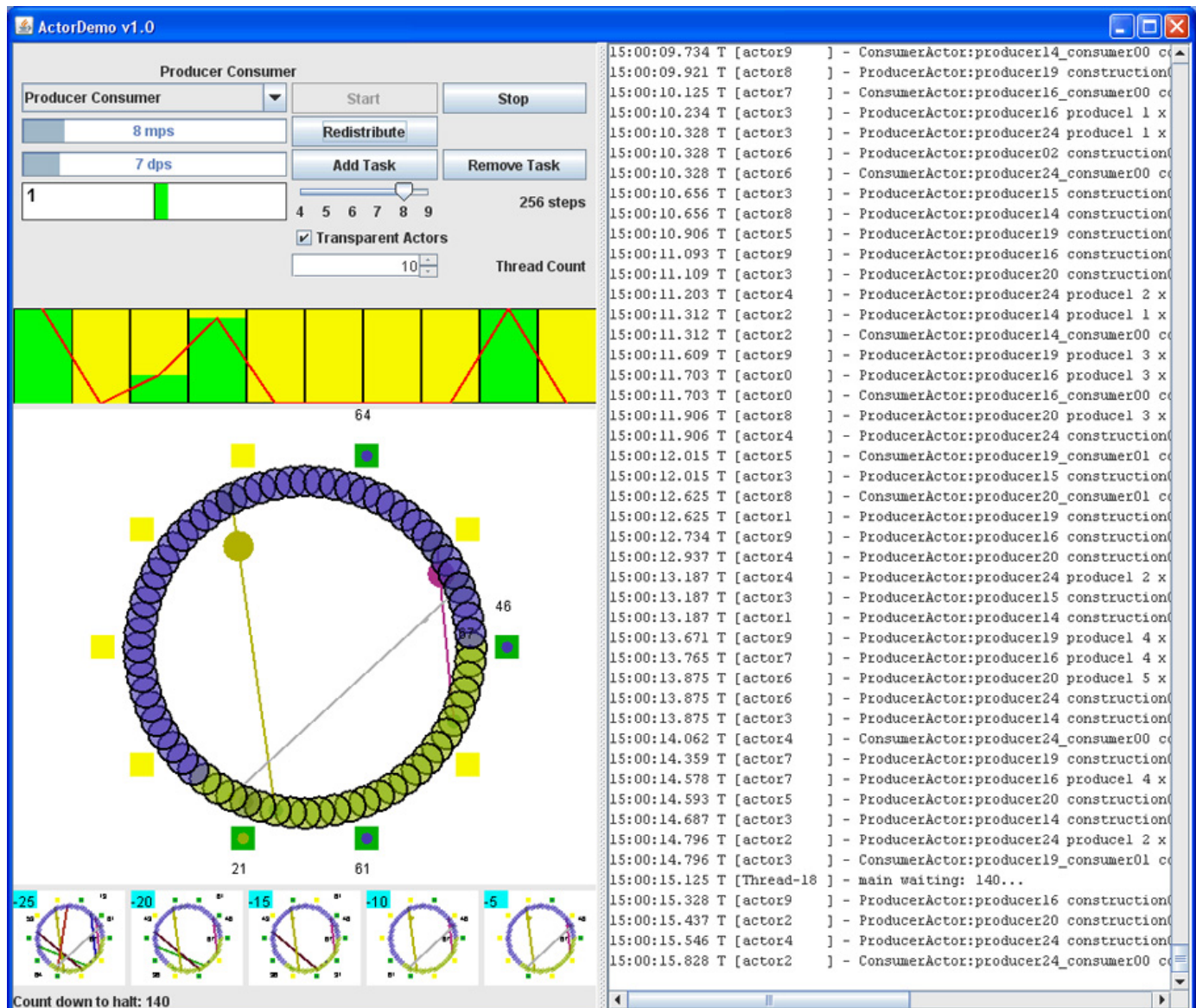


図 7 に、「Producer Consumer (プロデューサー/コンシューマー)」シミュレーションを約 115 秒間実行した時点の結果を示します。この時点で、このシミュレーション・プログラムの実行は完了に近づいています。新しい要求と処理待ちのメッセージの数は、すでに大幅に減っています。動画デモを見ると、一部のアクターが一瞬、色で塗りつぶされていない円として表示されることが気付くはずですが、これは、アクターが自分宛てに送信されたメッセージを処理していることを意味します。

図 7. 処理完了が近い「Producer Consumer (プロデューサー/コンシューマー)」シミュレーション



ProducerActor

リスト 7 に、デモで使用されているプロデューサー・アクターのコードを記載します。このコードで処理されているのは、「produceN」メッセージです。アクターはこのメッセージを「produce1」メッセージに変換してから、自身に送信します。想定される応答は、後で確認するために処理待ち応答のカウントとして記録されます。

リスト 7. プロデューサー・アクター

```
public class ProducerActor extends AbstractActor {
    Map<String, Integer> expected = new ConcurrentHashMap<String, Integer>();

    @Override
    protected void loopBody(Message m) {
```

```
String subject = m.getSubject();
if ("produceN".equals(subject)) {
    Object[] input = (Object[]) m.getData();
    int count = (Integer) input[0];
    if (count > 0) {
        DefaultActorTest.sleeper(1); // this takes some time
        String type = (String) input[1];
        // request the consumers to consume work (i.e., produce)
        Integer mcount = expected.get(type);
        if (mcount == null) {
            mcount = new Integer(0);
        }
        mcount += count;
        expected.put(type, mcount);

        DefaultMessage dm = new DefaultMessage("produce1",
            new Object[] { count, type });
        getManager().send(dm, this, this);
    }
}
```

リスト 8 では、「produce1」メッセージが処理されます。残りのカウントがゼロより大きい場合、このメッセージは「construct」メッセージに変換されてコンシューマーに送信されます。このロジックは、「produce1」メッセージを再送信する代わりにカウント値での for ループとして実行することもできますが、メッセージを再送信したほうがスレッドの負荷にとって望ましい結果となることはよくあります。これは、ループ本体の処理にかなりの時間がかかる場合には、特に言えることです。

リスト 8. プロデューサーの要求の処理

```
} else if ("produce1".equals(subject)) {
    Object[] input = (Object[]) m.getData();
    int count = (Integer) input[0];
    if (count > 0) {
        sleep(100); // take a little time
        String type = (String) input[1];
        m = new DefaultMessage("construct", type);
        getManager().send(m, this, getConsumerCategory());

        m = new DefaultMessage("produce1", new Object[] { count - 1, type });
        getManager().send(m, this, this);
    }
}
```

リスト 9 では、コンシューマーから送信された「constructionComplete」メッセージが処理されます。処理待ち応答のカウントは、このコードによってデクリメントされます。すべてが正常に機能していれば、シミュレーションが完了した時点で、アクターおよびタイプ値のすべてに対して、処理待ち応答のカウントはゼロになります。

リスト 9. constructionComplete

```
} else if ("constructionComplete".equals(subject)) {
    String type = (String) m.getData();
    Integer mcount = expected.get(type);
    if (mcount != null) {
        mcount--;
        expected.put(type, mcount);
    }
}
```

リスト 10 では、「init」メッセージが処理されます。プロデューサーはコンシューマー・アクターを作成した後、複数の produceN 要求を自身に送信します。

リスト 10. 初期化

```

} else if ("init".equals(subject)) {
    // create some consumers; 1 to 3 x consumers per producer
    for (int i = 0; i < DefaultActorTest.nextInt(3) + 1; i++) {
        Actor a = getManager().createAndStartActor(ConsumerActor.class,
            String.format("%s_consumer%02d", getName(), i));
        a.setCategory(getConsumerCategory());
        if (actorTest != null) {
            actorTest.getTestActors().put(a.getName(), a);
        }
    }
    // request myself create some work items
    for (int i = 0; i < DefaultActorTest.nextInt(10) + 1; i++) {
        m = new DefaultMessage("produceN", new Object[]
            { DefaultActorTest.nextInt(10) + 1,
              DefaultActorTest.getItemTypes()[
                DefaultActorTest.nextInt(DefaultActorTest.getItemTypes().length)] });
        getManager().send(m, this, this);
    }
}

```

リスト 11 は、無効なメッセージを処理するコードです。

リスト 11. 無効なメッセージの処理

```

} else {
    System.out.printf("ProducerActor:%s loopBody unknown subject: %s\n",
        getName(), subject);
}
}

protected String getConsumerCategory() {
    return getName() + "_consumer";
}
}

```

ConsumerActor

コンシューマー・アクターは単純なもので、「construct」メッセージを処理して応答メッセージを要求側に送信するだけに過ぎません。リスト 12 にコンシューマー・アクターのコードを記載します。

リスト 12. コンシューマー・アクター

```

public class ConsumerActor extends AbstractActor {

    @Override
    protected void loopBody(Message m) {
        String subject = m.getSubject();
        if ("construct".equals(subject)) {
            String type = (String) m.getData();
            delay(type); // takes ~ 1 to N seconds

            DefaultMessage dm = new
                DefaultMessage("constructionComplete", type);
            getManager().send(dm, this, m.getSource());
        } else if ("init".equals(subject)) {
            // nothing to do
        } else {
            System.out.printf("ConsumerActor:%s loopBody unknown subject: %s\n",
                getName(), subject);
        }
    }
}

```

リスト 13 で処理している生成遅延は、作成するアイテムのタイプに基づきます。トレースに示されていたように、サポートされるアイテムのタイプは、`widget`、`framit`、`frizzle`、`gothca`、および `splat` です。タイプによって、作成時間は異なります。

リスト 13. 生成遅延

```
protected void delay(String type) {
    int delay = 1;
    for (int i = 0; i < DefaultActorTest.getItemTypes().length; i++) {
        if (DefaultActorTest.getItemTypes()[i].equals(type)) {
            break;
        }
        delay++;
    }
    DefaultActorTest.sleeper(DefaultActorTest.nextInt(delay) + 1);
}
```

Producer/Consumer パターンでのアクター

「Producer Consumer (プロデューサー/コンシューマー)」シミュレーションのデモから、アクター実装を簡単に作成できることは明らかです。典型的なアクターは、case 文での処理のように、受信したメッセージをデコードして処理します。この単なる時間遅延の例で行う実際の処理はありきたりなものです。実際のアプリケーションでは、これよりも複雑な処理になるはずですが、それでも標準的な Java 同期手法を使用した実装より複雑になることはありません。通常は、それよりも遥かに単純です。

このデモでもう 1 つ注目してもらいたい点として、複雑なアルゴリズム (特に繰り返されるアルゴリズム) は別々の (そして多くの場合は再利用可能な) ステップに分割することができます。各ステップには異なるサブジェクト名を割り当てることができるため、サブジェクトごとのケースを作成するのは非常に単純です。状態 (前に説明したカウントダウンの値など) がメッセージ・パラメーターで渡される場合でも、多くのアクターはステートレスになることができます。このようなプログラムは、定義するにも、(スレッド数の増加に合わせてアクターを追加することで) スケーリングするにも簡単です。しかも、マルチスレッド環境で安全に実行することができます。これは、関数型プログラミングで不変の値を使用する手法と似ています。

アクターのその他のパターン

「Producer Consumer (プロデューサー/コンシューマー)」シミュレーションのデモでのアクターは、特定の目的に沿ってハード・コーディングされていますが、これがアクターをコーディングする際の唯一の選択肢というわけではありません。このセクションでは、より汎用的なパターンでアクターを使用する方法について学びます。まず初めに取り上げるのは、[Gang of Four の Command パターン](#)を適応させたアクターです。

リスト 14 のアクターが実装するのは、ほとんどの Java 開発者にとってお馴染みの Command パターンのバリエーションです。この `CommandActor` は、「`execute`」と「`executeStatic`」という 2 つのメッセージをサポートします。

リスト 14. CommandActor

```
public class CommandActor extends AbstractActor {

    @Override
    protected void loopBody(Message m) {
        String subject = m.getSubject();
        if ("execute".equals(subject)) {
            excuteMethod(m, false);
        } else if ("executeStatic".equals(subject)) {
            excuteMethod(m, true);
        } else if ("init".equals(subject)) {
            // nothing to do
        } else {
            System.out.printf("CommandActor:%s loopBody unknown subject: %s",
                              getName(), subject);
        }
    }
}
```

executeMethod メソッド (リスト 15 を参照) は、パラメーター化されたクラスをロードし、そのクラスまたはそのクラスのインスタンスでメソッドを呼び出して、結果 (例外が発生した場合にはその例外) を返します。これらのリストから、この単純な CommandActor アクターを使用すれば、クラスパスに存在していて、適切な実行メソッドを持つあらゆるサービス・クラスを実行できることがわかるはずです。id パラメーターがクライアントから送信されるため、アクターはこの id パラメーターを使用して、応答とそれに対応する要求とを関連付けることができます。要求が送信されたのとは異なる順序で、応答が返されることはよくあることです。

リスト 15. パラメーター化されたメソッドの実行

```
private void excuteMethod(Message m, boolean fstatic) {
    Object res = null;
    Object id = null;
    try {
        Object[] params = (Object[]) m.getData();
        id = params[0];
        String className = (String) params[1];
        params = params.length > 2 ? (Object[]) params[2] : null;
        Class<?> clazz = Class.forName(className);
        Method method = clazz.getMethod(fstatic ? "executeStatic"
                                         : "execute", new Class[] { Object.class });
        if (Modifier.isStatic(method.getModifiers()) == fstatic) {
            Object target = fstatic ? null : clazz.newInstance();
            res = method.invoke(target, params);
        }
    } catch (Exception e) {
        res = e;
    }

    DefaultMessage dm = new DefaultMessage("executeComplete", new Object[] {
        id, res });
    getManager().send(dm, this, m.getSource());
}
}
```

Event Listener パターンでのアクター

リスト 16 の DelegatingActor は、お馴染みの Java Event Listener (または Callback) パターンをベースにした同様の汎用的な手法を実装します。このアクターは、到着したメッセージのそれぞれを登録済みの各リスナーの onMessage コールバックにマッピングするという動作を、1 つのコール

バックがイベントを使用 (つまり処理) するまで続けます。この委任手法によって、アクター・システムとそのメッセージ・プロセッサとの間の結合を大幅に緩めることができます。

リスト 16. DelegatingActor

```
public class DelegatingActor extends AbstractActor {
    private List<MessageListener> listeners = new LinkedList<MessageListener>();

    public void addMessageListener(MessageListener ml) {
        if (!listeners.contains(ml)) {
            listeners.add(ml);
        }
    }

    public void removeMessageListener(MessageListener ml) {
        listeners.remove(ml);
    }

    protected void fireMessageListeners(MessageEvent me) {
        for (MessageListener ml : listeners) {
            if (me.isConsumed()) {
                break;
            }
            ml.onMessage(me);
        }
    }

    @Override
    protected void loopBody(Message m) {
        fireMessageListeners(new MessageEvent(this, m));
    }
}
```

`DelegatingActor` クラス (リスト 17 を参照) は、`MessageEvent` クラスと `MessageListener` クラスに依存します。

リスト 17. DelegatingActor

```
/** Defines a message arrival event. */
public static class MessageEvent extends EventObject {
    private Message message;

    public Message getMessage() {
        return message;
    }

    public void setMessage(Message message) {
        this.message = message;
    }

    private boolean consumed;

    public boolean isConsumed() {
        return consumed;
    }

    public void setConsumed(boolean consumed) {
        this.consumed = consumed;
    }

    public MessageEvent(Object source, Message msg) {
        super(source);
        setMessage(msg);
    }
}
```

```
}

/** Defines the message arrival call back. */
public interface MessageListener {
    void onMessage(MessageEvent me);
}
```

リスト 18 に、`DelegatingActor` の使用例を記載します。

リスト 18. `DelegatingActor` の使用例

```
public static void addDelegate(DelegatingActor da) {
    MessageListener ml = new Echo("Hello world!");
    da.addMessageListener(ml);
}

public class Echo implements MessageListener {
    protected String message;

    public Echo(String message) {
        this.message = message;
    }

    @Override
    public void onMessage(MessageEvent me) {
        if ("echo".equals(me.getMessage().getSubject())) {
            System.out.printf("%s says \"%s\".%n",
                me.getMessage().getSource(), message);
            me.setConsumed(true);
        }
    }
}
```

Map/Reduce パターンでのアクター

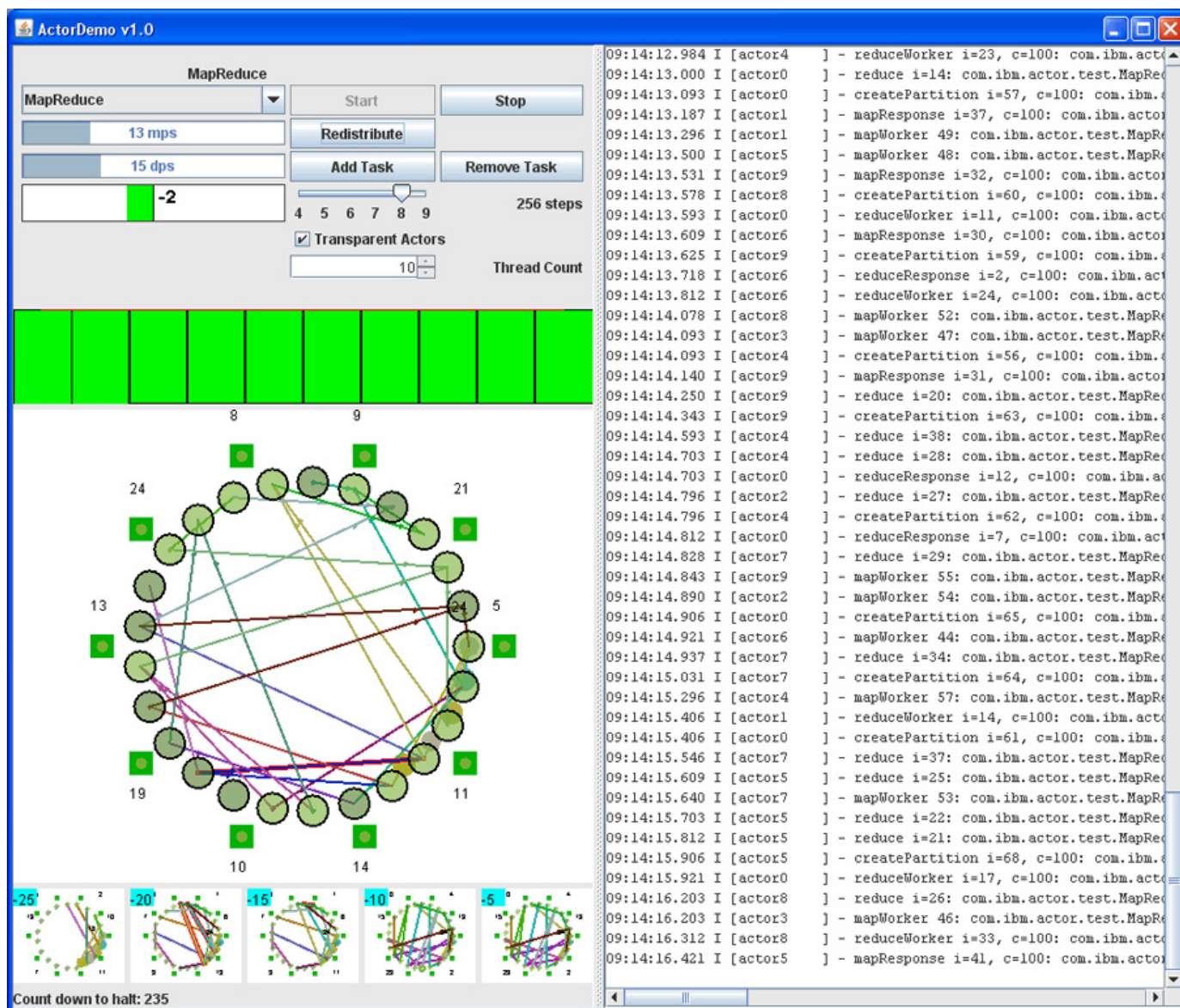
リスト 14 からリスト 18 に記載したサンプル・アクターは、メッセージを一方向にしか送信しないことから単純明快なものになっていますが、(例えば、これまでのメッセージの処理がすべて完了するまでは、プロセスを続行できない場合など) アクターの動作にフィードバックが必要な場合には、事態は複雑になってきます。Map/Reduce 実装を例として考えてみると、`map` フェーズが完了するまでは、`reduce` フェーズを開始することはできません。

大量のデータを処理するプログラムでは、並列処理を行うために Map/Reduce が使用されます。次に記載する例では、大量のアイテムが含まれるリストを `map` 関数が受け取り、それをパーティションに分割した後、メッセージを送信して各パーティションをマッピングします。この例では、マップ要求ごとにメッセージ・カウントをインクリメントし、パーティション化されたマップ・プロセッサに応答を送信させてカウントを減らすことにしました。カウントがゼロに達した時点ですべてのマッピングが完了したことになるので、その時点で `reduce` フェーズを開始することができます。`reduce` フェーズでも同様に、リストを (同じく並列処理するために) パーティション化し、メッセージを送信してパーティションの `reduce` 処理を行います。`map` フェーズと同じように、`reduce` 処理でもそのメッセージをカウントするので、`reduce` フェーズの完了を検出することができます。処理対象の値のリストおよびカウントは、各メッセージでパラメーターとして渡されます。

この例には、多くのサブジェクトに対し、1つのアクター・タイプだけを使用することにしましたが、複数のアクター・タイプを使用して、アクターあたりのサブジェクト数を(最小1まで)減らすこともできます。

図8に、「MapReduce」シミュレーションを約20秒間実行した時点の結果を示します。このフェーズではアクティブに処理が行われるので、すべてのスレッドはメッセージの処理に追われてビジー状態になっています。

図8. 実行開始後の「MapReduce」シミュレーション (動画では 5:28 あたりから開始)



MapReducer を使用した map と reduce

この実装はプラグラブルなので、MapReducer インターフェースの任意の実装を実行することができます (リスト 19 を参照)。

リスト 19. MapReduceer

```
public interface MapReduceer {
    /**
     * Map (in place) the elements of an array.
     *
     * @param values elements to map
     * @param start start position in values
     * @param end end position in values
     */
    void map(Object[] values, int start, int end);

    /**
     * Reduce the elements of an array.
     *
     * @param values elements to reduce
     * @param start start position in values
     * @param end end position in values
     * @param target place to set reduced value
     * @param posn position in target to place the value
     */
    void reduce(Object[] values, int start, int end, Object[] target, int posn);
}
```

`MapReduceer` は、例えば一連の整数の二乗和を計算するために使用することができます (リスト 20 を参照)。

リスト 20. MapReduceer による計算

```
public class SumOfSquaresReducer implements MapReduceer {
    @Override
    public void map(Object[] values, int start, int end) {
        for (int i = start; i <= end; i++) {
            values[i] = ((BigInteger) values[i]).multiply((BigInteger) values[i]);
            sleep(200); // fake taking time
        }
    }

    @Override
    public void reduce(Object[] values, int start, int end, Object[] target, int posn) {
        BigInteger res = new BigInteger("0");
        for (int i = start; i <= end; i++) {
            res = res.add((BigInteger) values[i]);
            sleep(100); // fake taking time
        }
        target[posn] = res;
    }
}
```

MapReduceActor

Map/Reduce アクターは、それぞれに単純なタスクを定義するいくつかのサブジェクトに分けられます。これらのサブジェクトのそれぞれを以降のサンプル・コードで見えていきます。サンプル・コードだけでなく、動画デモで [Map/Reduce 処理を見る](#) ことをお勧めします。シミュレーションの動画を見てからサンプル・コードを調べることで、Map/Reduce がアクターによってどのように実装されるかを極めて明確に理解できるようになります (以下のリストでのサブジェクトの順序は、いくらでも変更できることに注意してください。また、動画デモをより興味深いものにするために、このサンプル・コードは多数の送信処理を行うように設計してあります)。

リスト 21 に記載する `mapReduce` サブジェクトは、`createPartition` メッセージを送信することにより入力配列をパーティション化し、Map/Reduce を開始します。map と reduce のパラメーターは、`MapReduceParameters` インスタンス内に指定されます。このインスタンスが必要に応じて複製および変更されてから渡されます。時間遅延はこの処理には必要ありませんが、この例ではユーザー・インターフェースで確実にシミュレーションを確認できるようにするために追加しました。

リスト 21. `mapReduce`

```
@Override
protected void loopBody(Message m) {
    ActorManager manager = getManager();
    String subject = m.getSubject();
    if ("mapReduce".equals(subject)) {
        try {
            MapReduceParameters p = (MapReduceParameters) m.getData();
            int index = 0;
            int count = (p.end - p.start + 1 + partitionSize - 1) / partitionSize;
            sleep(1000);
            // split up into partition size chunks
            while (p.end - p.start + 1 >= partitionSize) {
                MapReduceParameters xp = new MapReduceParameters(p);
                xp.end = xp.start + partitionSize - 1;
                DefaultMessage lm = new DefaultMessage("createPartition",
                    new Object[] { xp, index, count });
                manager.send(lm, this, getCategory());
                p.start += partitionSize;
                index++;
            }
            if (p.end - p.start + 1 > 0) {
                DefaultMessage lm = new DefaultMessage("createPartition",
                    new Object[] { p, index, count });
                manager.send(lm, this, getCategory());
            }
        } catch (Exception e) {
            triageException("mapFailed", m, e);
        }
    }
}
```

`createPartition` サブジェクトはアクターをさらに作成して、要求をワーカーに転送します (リスト 22 を参照)。`createMapReduceActor` メソッドが作成するアクター数には、上限 (現在は 25) が設けられていることに注意してください。

リスト 22. `createPartition`

```
} else if ("createPartition".equals(subject)) {
    try {
        Object[] oa = (Object[]) m.getData();
        MapReduceParameters p = (MapReduceParameters) oa[0];
        int index = (Integer) oa[1];
        int count = (Integer) oa[2];
        sleep(500);
        createMapReduceActor(this);
        DefaultMessage lm = new DefaultMessage("mapWorker",
            new Object[] { p, index, count });
        manager.send(lm, this, getCategory());
    } catch (Exception e) {
        triageException("createPartitionFailed", m, e);
    }
}
```

リスト 23 の `mapWorker` サブジェクトは、提供された `MapReduceer` を介してパーティションで `map` 処理を呼び出し、`map` 処理が完了した時点でその旨を応答します。

リスト 23. `mapWorker`

```
} else if ("mapWorker".equals(subject)) {
    try {
        Object[] oa = (Object[]) m.getData();
        MapReduceParameters p = (MapReduceParameters) oa[0];
        int index = (Integer) oa[1];
        int count = (Integer) oa[2];
        sleep(100);
        p.mr.map(p.values, p.start, p.end);
        DefaultMessage rm = new DefaultMessage("mapResponse",
            new Object[] { p, index, count });
        manager.send(rm, this, getCategoryName());
    } catch (Exception e) {
        triageException("mapWorkerFailed", m, e);
    }
}
```

続いて、リスト 24 の `mapResponse` サブジェクトが (カウントを保持している) `MapReduceParameters` インスタンスを完了し、`reduce` プロセスを開始します。

リスト 24. `mapResponse`

```
} else if ("mapResponse".equals(subject)) {
    try {
        Object[] oa = (Object[]) m.getData();
        MapReduceParameters p = (MapReduceParameters) oa[0];
        int index = (Integer) oa[1];
        int count = (Integer) oa[2];
        sleep(100);
        p.complete();
        DefaultMessage rm = new DefaultMessage("reduce",
            new Object[] { p, index, count });
        manager.send(rm, this, getCategoryName());
    } catch (Exception e) {
        triageException("mapResponseFailed", m, e);
    }
}
```

次に、`reduce` メッセージによって要求がワーカーに転送されます (リスト 25 を参照)。

リスト 25. `reduce`

```
} else if ("reduce".equals(subject)) {
    try {
        MapReduceParameters p = null;
        int index = 0, count = 0;
        Object o = m.getData();
        if (o instanceof MapReduceParameters) {
            p = (MapReduceParameters) o;
        } else {
            Object[] oa = (Object[]) o;
            p = (MapReduceParameters) oa[0];
            index = (Integer) oa[1];
            count = (Integer) oa[2];
        }
        sleep(100);
        if (p.end - p.start + 1 > 0) {
            createMapReduceActor(this);
            MapReduceParameters xp = new MapReduceParameters(p);
```

```

        DefaultMessage lm = new DefaultMessage("reduceWorker",
            new Object[] { xp, index, count });
        manager.send(lm, this, getCategory());
    }
} catch (Exception e) {
    triageException("reduceFailed", m, e);
}
}

```

リスト 26 の `reduceWorker` サブジェクトは、提供された `MapReduceer` を介してパーティションで `reduce` 処理を呼び出し、`reduce` 処理が完了した時点でその旨を応答します。すべての `reduce` 処理が完了した場合には、`Map/Reduce` 処理が完了したことを応答します。

リスト 26. `reduceWorker`

```

} else if ("reduceWorker".equals(subject)) {
    try {
        Object[] oa = (Object[]) m.getData();
        MapReduceParameters p = (MapReduceParameters) oa[0];
        int index = (Integer) oa[1];
        int count = (Integer) oa[2];
        sleep(100);
        if (index >= 0) {
            p.mr.reduce(p.values, p.start, p.end, p.target, index);
            DefaultMessage rm = new DefaultMessage("reduceResponse",
                new Object[] { p, index, count });
            manager.send(rm, this, getCategory());
        } else {
            Object[] res = new Object[1];
            p.mr.reduce(p.target, 0, count - 1, res, 0);
            DefaultMessage rm = new DefaultMessage("done",
                new Object[] { p, res[0] });
            manager.send(rm, this, getCategory());
        }
    } catch (Exception e) {
        triageException("reduceWorkerFailed", m, e);
    }
}
}

```

次に、リスト 27 の `reduceResponse` サブジェクトがパーティションを完了し、すべてのパーティションが完了したかどうかをテストして、その結果を通知します。

リスト 27. `reduceResponse`

```

} else if ("reduceResponse".equals(subject)) {
    try {
        Object[] oa = (Object[]) m.getData();
        MapReduceParameters p = (MapReduceParameters) oa[0];
        int index = (Integer) oa[1];
        int count = (Integer) oa[2];
        sleep(100);
        p.complete();
        if (p.isSetComplete()) {
            if (count > 0) {
                createMapReduceActor(this);
                MapReduceParameters xp = new MapReduceParameters(p);
                DefaultMessage lm = new DefaultMessage("reduceWorker",
                    new Object[] { xp, -1, count });
                manager.send(lm, this, getCategory());
            }
        }
    } catch (Exception e) {
        triageException("mapResponseFailed", m, e);
    }
}
}

```

```
    }  
}
```

最後に、リスト 28 の done サブジェクトが結果をレポートします。

リスト 28. done

```
    } else if ("done".equals(subject)) {  
        try {  
            Object[] oa = (Object[]) m.getData();  
            MapReduceParameters p = (MapReduceParameters) oa[0];  
            Object res = oa[1];  
            sleep(100);  
            System.out.printf("**** mapReduce done with result %s", res);  
        } catch (Exception e) {  
            triageException("mapResponseFailed", m, e);  
        }  
    }  
}
```

以上のループは、init サブジェクトが別の Map/Reduce プロセスを開始することによって繰り返されます (リスト 29 を参照)。それぞれの Map/Reduce には別の「セット」名が設定されるため、複数の Map/Reduce を同時に実行することができます。

リスト 29. 別の Map/Reduce の開始

```
    } else if ("init".equals(subject)) {  
        try {  
            Object[] params = (Object[]) m.getData();  
            if (params != null) {  
                Object[] values = (Object[]) params[0];  
                Object[] targets = (Object[]) params[1];  
                Class clazz = (Class) params[2];  
                MapReduceer mr = (MapReduceer) clazz.newInstance();  
                sleep(2 * 1000);  
                MapReduceParameters p = new MapReduceParameters("mrSet_" + setCount++,  
                    values, targets, mr, this);  
                DefaultMessage rm = new DefaultMessage("mapReduce", p);  
                manager.send(rm, this, getCategoryName());  
            }  
        } catch (Exception e) {  
            triageException("initFailed", m, e);  
        }  
    } else {  
        System.out.printf("**** MapReduceActor:%s loopBody unexpected subject: %s",  
            getName(), subject);  
    }  
}
```

Map/Reduce の main

リスト 30 に記載する MapReduceActor 実装は、データ値を作成して、そのデータで Map/Reduce を実行します。この実装では、パーティション・サイズを 10 に設定しています。

リスト 30. Map/Reduce の main

```

BigInteger[] values = new BigInteger[1000];
for (int i = 0; i < values.length; i++) {
    values[i] = new BigInteger(Long.toString((long)rand.nextInt(values.length)));
}
BigInteger[] targets = new BigInteger[Math.max(1, values.length / 10)];

// start at least 5 actors
DefaultActorManager am = new DefaultActorManager();
MapReduceActor.createMapReduceActor(am, 10);
MapReduceActor.createMapReduceActor(am, 10);
MapReduceActor.createMapReduceActor(am, 10);
MapReduceActor.createMapReduceActor(am, 10);
MapReduceActor.createMapReduceActor(am, 10);

DefaultMessage dm = new DefaultMessage("init", new Object[]
    { values, targets, SumOfSquaresReducer.class });
am.send(dm, null, MapReduceActor.getCategoryName());

```

Map/Reduce は最も広く使われている分割統治法デザイン・パターンの 1 つであり、基本的な関数型プログラミングのアルゴリズムから、大規模な並列処理 (Google がその Web 検索エンジンの索引を作成するために行っているような並列処理) に至るあらゆる分野で使用されています。µJavaActors ライブラリーがこのような高度なパターンを簡単に実装できることは、µJavaActors ライブラリーの能力を示しているだけでなく、潜在的な使用法も示しています。

µJavaActors ライブラリーの詳細

アクターがマネージャーを呼び出すのではありません。マネージャーがアクターを呼び出すのです。

ここまでで、アクターを使用して、よく使われているいくつかのオブジェクト指向パターンを別の目的で使用方法を見てきました。ここからは、µJavaActors システムの実装の詳細、つまり `AbstractActor` クラスと `DefaultActorManager` クラスを取り上げます。ここで取り上げるのは、各クラスの重要なメソッドだけです。その他の実装の詳細については、µJavaActors の [ソース・コード](#) を調べてください。

AbstractActor

すべてのアクターは、自身を管理する `ActorManager` を認識しています。アクターは、マネージャーの支援によって、他のアクターにメッセージを送信します。

リスト 31 では、`receive` メソッドが条件付きでメッセージを処理します。`testMessage` メソッドが `null` を返す場合、メッセージは使用されません。そうでない場合は、`loopBody` メソッドが呼び出されて、メッセージがアクターのメッセージ・キューから取り出され、処理されます。アクターのすべての具象サブクラスは、このメソッドを提供しなければなりません。メッセージが処理される場合もそうでない場合も、アクターはマネージャーの `awaitMessage` メソッドを呼び出すことによって、以降のメッセージが到着するのを待機します。

リスト 31. AbstractActor が実装する DefaultActorManager

```
public abstract class AbstractActor implements Actor {
    protected DefaultActorManager manager;

    @Override
    public boolean receive() {
        Message m = testMessage();
        boolean res = m != null;
        if (res) {
            remove(m);
            try {
                loopBody(m);
            } catch (Exception e) {
                System.out.printf("loop exception: %s\n", e);
            }
        }
        manager.awaitMessage(this);
        return res;
    }

    abstract protected void loopBody(Message m);
}
```

各アクターは、どのメッセージ・サブジェクトを受け入れるか（つまり、メッセージ・リストに入れられるメッセージ）を制御するために `willReceive` メソッドを実装することができます。デフォルトでは、`null` 以外のサブジェクトを持つすべてのメッセージが受け入れられます。処理可能なメッセージがあるかどうか（つまり、メッセージ・リストにメッセージがあるかどうか）をチェックするために、各アクターは `testMessage` メソッドを実装することもできます。デフォルトでは、このメッセージの監視の実装には `peekNext` メソッドが使用されます。

リスト 32. willReceive()、testMessage()、および peekNext()

```
@Override
public boolean willReceive(String subject) {
    return !isEmpty(subject);
}

protected Message testMessage() {
    return getMatch(null, false);
}

protected Message getMatch(String subject, boolean isRegExpr) {
    Message res = null;
    synchronized (messages) {
        res = peekNext(subject, isRegExpr);
    }
    return res;
}
```

メッセージ容量

アクターのメッセージ容量は、無制限にすることも、制限することもできます。一般には、容量を制限したほうが、短期間に大量のメッセージを送信する送信側の検出に役立つので、望ましいです。あらゆるクライアント（ただし一般的には `ActorManager`）がメッセージを、選別せずにアクターに追加する可能性があります。`messages` のリストへのアクセスは、常に同期アクセスであることに注意してください。

リスト 33. メッセージの処理

```
public static final int DEFAULT_MAX_MESSAGES = 100;
protected List<DefaultMessage> messages = new LinkedList<DefaultMessage>();
```

```
@Override
public int getMessageCount() {
    synchronized (messages) {
        return messages.size();
    }
}

@Override
public int getMaxMessageCount() {
    return DEFAULT_MAX_MESSAGES;
}

public void addMessage(Message message) {
    synchronized (messages) {
        if (messages.size() < getMaxMessageCount()) {
            messages.add(message);
        } else {
            throw new IllegalStateException("too many messages, cannot add");
        }
    }
}

@Override
public boolean remove(Message message) {
    synchronized (messages) {
        return messages.remove(message);
    }
}
```

メッセージ・マッチング

クライアント (特にアクター自体) は、アクターに処理待ちのメッセージがあるかどうかをチェックすることができます。このチェックを利用して、メッセージを送信順に処理したり、特定のサブジェクトを優先的に処理したりすることができます。メッセージ・マッチングは、あるストリング値とメッセージ・サブジェクトが等しいかどうかをテストするか、パラメーターの値に対して正規表現による突合せをすることによって行われます。`null` サブジェクトは、あらゆるメッセージとマッチします。この場合も、メッセージ・リストへのアクセスは必ず同期アクセスによって行われることに注意してください。

リスト 34. peekNext()

```
@Override
public Message peekNext() {
    return peekNext(null);
}

@Override
public Message peekNext(String subject) {
    return peekNext(subject, false);
}

@Override
public Message peekNext(String subject, boolean isRegExpr) {
    long now = new Date().getTime();
    Message res = null;
    Pattern p = subject != null ? (isRegExpr ? Pattern.compile(subject) : null) : null;
    synchronized (messages) {
        for (DefaultMessage m : messages) {
            if (m.getDelayUntil() <= now) {
                boolean match = subject == null ||
                    (isRegExpr ? m.subjectMatches(p) : m.subjectMatches(subject));
                if (match) {
```

```
        res = m;
        break;
    }
}
}
return res;
}
```

ライフサイクル関連のメソッド

各アクターに、ライフサイクル関連のメソッドがあります。activate メソッドと deactivate メソッドは、特定の ActorManager との関連付けごとに 1 回呼び出されます。run メソッドが呼び出されるのも、特定の ActorManager との関連付けごとに 1 回です。このメソッドは通常、起動メッセージをアクター自身に送信することによってアクターのブートストラップを行います。run メッセージによって、メッセージ処理が開始されます。

リスト 35. ライフサイクル関連のメソッド

```
@Override
public void activate() {
    // defaults to no action
}

@Override
public void deactivate() {
    // defaults to no action
}

/** Do startup processing. */
protected abstract void runBody();

@Override
public void run() {
    runBody();
    ((DefaultActorManager) getManager()).awaitMessage(this);
}
}
```

DefaultActorManager

アクター・マネージャーの状態は、以下のフィールドに格納されます。

- `actors`: マネージャーに登録済みのすべてのアクターを格納します。
- `runnables`: 作成されたアクターのうち、その run メソッドがまだ呼び出されていないすべてのアクターを格納します。
- `waiters`: メッセージを待機中のすべてのアクターを格納します。
- `threads`: マネージャーによって開始されたすべてのスレッドを格納します。

LinkedHashMap を使用することが不可欠であることに注意してください (特に waiters リストには極めて重要です)。これを使用しないと、一部のアクターでスレッドの枯渇が発生する可能性があります。

リスト 36. DefaultActorManager クラスとその状態

```
public class DefaultActorManager implements ActorManager {

    public static final int DEFAULT_ACTOR_THREAD_COUNT = 25;
```

```
protected static DefaultActorManager instance;
public static DefaultActorManager getDefaultInstance() {
    if (instance == null) {
        instance = new DefaultActorManager();
    }
    return instance;
}

protected Map<String , AbstractActor> actors =
    new LinkedHashMap<String , AbstractActor>();

protected Map<String , AbstractActor> runnables =
    new LinkedHashMap<String , AbstractActor>();

protected Map<String , AbstractActor> waiters =
    new LinkedHashMap<String , AbstractActor>();

protected List<Thread> threads = new LinkedList<Thread>();
```

`detachActor` メソッドは、アクターとそのマネージャーとの関連付けを解除します。

リスト 37. アクターの終了

```
@Override
public void detachActor(Actor actor) {
    synchronized (actors) {
        actor.deactivate();
        ((AbstractActor)actor).setManager(null);
        String name = actor.getName();
        actors.remove(name);
        runnables.remove(name);
        waiters.remove(name);
    }
}
```

send メソッド

`send` メソッド・ファミリーは、メッセージを 1 つ以上のアクターに送信します。送信されたメッセージごとに、最初にアクターがそのメッセージを受け入れるかどうかチェックされます。メッセージがキューに入れられると、`notify` を使用して、メッセージを処理するスレッドをウェイクアップします。カテゴリーにメッセージを送信する場合、実際にはそのカテゴリー内の 1 つのアクター (その時点でメッセージ数が最も少ないアクター) だけにメッセージが送信されます。`awaitMessage` メソッドは、単に `waiters` リストのアクターをキューに入れるだけです。

リスト 38. DefaultActorManager クラスによる send の処理

```
@Override
public int send(Message message, Actor from, Actor to) {
    int count = 0;
    AbstractActor aa = (AbstractActor) to;
    if (aa != null) {
        if (aa.willReceive(message.getSubject())) {
            DefaultMessage xmessage = (DefaultMessage)
                ((DefaultMessage) message).assignSender(from);
            aa.addMessage(xmessage);
            count++;
            synchronized (actors) {
                actors.notifyAll();
            }
        }
    }
    return count;
}
```

```
}

@Override
public int send(Message message, Actor from, Actor[] to) {
    int count = 0;
    for (Actor a : to) {
        count += send(message, from, a);
    }
    return count;
}

@Override
public int send(Message message, Actor from, Collection<Actor> to) {
    int count = 0;
    for (Actor a : to) {
        count += send(message, from, a);
    }
    return count;
}

@Override
public int send(Message message, Actor from, String category) {
    int count = 0;
    Map<String, Actor> xactors = cloneActors();
    List<Actor> catMembers = new LinkedList<Actor>();
    for (String key : xactors.keySet()) {
        Actor to = xactors.get(key);
        if (category.equals(to.getCategory()) &&
            (to.getMessageCount() < to.getMaxMessageCount())) {
            catMembers.add(to);
        }
    }
    // find an actor with lowest message count
    int min = Integer.MAX_VALUE;
    Actor amin = null;
    for (Actor a : catMembers) {
        int mcount = a.getMessageCount();
        if (mcount < min) {
            min = mcount;
            amin = a;
        }
    }
    if (amin != null) {
        count += send(message, from, amin);
    }
    return count;
}

@Override
public int broadcast(Message message, Actor from) {
    int count = 0;
    Map<String, Actor> xactors = cloneActors();
    for (String key : xactors.keySet()) {
        Actor to = xactors.get(key);
        count += send(message, from, to);
    }
    return count;
}

public void awaitMessage(AbstractActor a) {
    synchronized (actors) {
        waiters.put(a.getName(), a);
    }
}
}
```

スレッド・プールの初期化

マネージャーは、受信メッセージを処理するためにアクターに割り当てる、優先度の低いデーモン・スレッドのプールを提供します (簡潔にするために、以下のリストではオプションの処理が省略されていることに注意してください。オプションの処理は、付属のソース・コードに含まれています)。

リスト 39. DefaultActorManager クラスの初期化

```
protected static int groupCount;

@Override
public void initialize(Map<String, Object> options) {
    int count = getThreadCount(options);
    ThreadGroup tg = new ThreadGroup("ActorManager" + groupCount++);
    for (int i = 0; i < count; i++) {
        Thread t = new Thread(tg, new ActorRunnable(), "actor" + i);
        threads.add(t);
        t.setDaemon(true);
        t.setPriority(Math.max(Thread.MIN_PRIORITY,
            Thread.currentThread().getPriority() - 1));
    }
    running = true;
    for (Thread t : threads) {
        t.start();
    }
}
```

各アクターは、リスト 40 に記載する Runnable 実装によってディスパッチされます。使用可能なアクター (処理待ちのメッセージを持つアクター) がある限り、アクターのディスパッチが行われます。使用可能なアクターがなければ、スレッドはメッセージの到着を待機します (待機時間には可変のタイムアウトが設定されます)。

リスト 40. Runnable によるメッセージの処理

```
public class ActorRunnable implements Runnable {
    public void run() {
        int delay = 1;
        while (running) {
            try {
                if (!procesNextActor()) {
                    synchronized (actors) {
                        actors.wait(delay * 1000);
                    }
                    delay = Math.max(5, delay + 1);
                } else {
                    delay = 1;
                }
            } catch (InterruptedException e) {
            } catch (Exception e) {
                System.out.printf("procesNextActor exception %s\n", e);
            }
        }
    }
}
```

`procesNextActor` メソッドは、まず、新しく作成されたアクターがあるかどうかを調べ、該当するアクターがあれば、そのアクターを実行します。新しく作成されたアクターがなければ、待機中のアクターがあるかどうかを調べます。待機中のアクターがあると、そのアクターを次のメッセージを処理するためにディスパッチします。1 回の呼び出しで処理されるメッセージは 1 つだ

けです。注意する点として、同期はすべて `actors` フィールドで行われます。そのため、デッドロックが発生する可能性が低くなっています。

リスト 41. 次のアクターの選択とディスパッチ

```
protected boolean procesNextActor() {
    boolean run = false, wait = false, res = false;
    AbstractActor a = null;
    synchronized (actors) {
        for (String key : runnables.keySet()) {
            a = runnables.remove(key);
            break;
        }
    }
    if (a != null) {
        run = true;
        a.run();
    } else {
        synchronized (actors) {
            for (String key : waiters.keySet()) {
                a = waiters.remove(key);
                break;
            }
        }
    }
    if (a != null) {
        // then waiting for responses
        wait = true;
        res = a.receive();
    }
}
return run || res;
}
```

終了メソッド

マネージャーの終了は、`terminate` メソッドまたは `terminateAndWait` メソッドのいずれかの呼び出しによって要求されます。`terminate` メソッドは、すべてのスレッドにできるだけ早く処理を停止するように指示します。`terminateAndWait` メソッドは、スレッドが完了するまで待ってから停止します。

リスト 42. DefaultActorManager クラスの終了

```
@Override
public void terminateAndWait() {
    terminate();
    for (Thread t : threads) {
        try {
            t.join();
        } catch (InterruptedException e) {
        }
    }
}

boolean running;

@Override
public void terminate() {
    running = false;
    for (Thread t : threads) {
        t.interrupt();
    }
    synchronized (actors) {
        for (String key : actors.keySet()) {
            actors.get(key).deactivate();
        }
    }
}
```



```

    }
  }
}

```

作成メソッド

create メソッド・ファミリーは、アクターを作成し、そのアクターをこのマネージャーに関連付けます。create には、アクターのクラスが指定されますが、このクラスはデフォルト・コンストラクターを持つクラスでなければなりません。さらに、アクターはその作成時、あるいは後で起動することができます。この実装では、すべてのアクターが `AbstractActor` を継承する必要があります。ことに注意してください。

リスト 43. アクターの作成と起動

```

@Override
public Actor createAndStartActor(Class<? extends Actor> clazz, String name,
    Map<String, Object> options) {
    Actor res = createActor(clazz, name, options);
    startActor(res);
    return res;
}

@Override
public Actor createActor(Class<? extends Actor> clazz, String name,
    Map<String, Object> options) {
    AbstractActor a = null;
    synchronized (actors) {
        if (!actors.containsKey(name)) {
            try {
                a = (AbstractActor) clazz.newInstance();
                a.setName(name);
                a.setManager(this);
            } catch (Exception e) {
                throw e instanceof RuntimeException ?
                    (RuntimeException) e : new RuntimeException(
                        "mapped exception: " + e, e);
            }
        } else {
            throw new IllegalArgumentException("name already in use: " + name);
        }
    }
    return a;
}

@Override
public void startActor(Actor a) {
    a.activate();
    synchronized (actors) {
        String name = a.getName();
        actors.put(name, (AbstractActor) a);
        runnables.put(name, (AbstractActor) a);
    }
}

```

まとめ

別れとは、なんと甘い悲しみなのでしょう！

この記事では、一般的な各種の Java プログラミング・シナリオとパターンに比較的単純なアクター・システムを適用する方法を学びました。µjavaActors ライブラリーの柔軟かつ動的な動作は、Akka などの重たいアクター・ライブラリーに代わる Java ベースの手段になります。

サンプル・コードとシミュレーションの動画から、`µjavaActors` がアクター・メッセージ処理を効率的に実行スレッド・プールの全体に分配できることは、はっきりと見て取れます。さらに、追加のスレッドが必要な場合には、ユーザー・インターフェースがすぐにそのことを明らかにします。このインターフェースでは、ワークが不足しているアクターや、過負荷状態のアクターがあるかどうか簡単に判断することができます。

`ActorManager` インターフェースのデフォルト実装である `DefaultActorManager` は、どのアクターも一度に1つのメッセージだけ処理することを保証します。したがって、アクターの作成者は、リエントラントな処理 (再入可能処理) について懸念する必要はありません。さらに、(1) アクターがプライベート・データ (つまり、インスタンスまたはメソッドのローカル・データ) だけを使用し、(2) メッセージ・パラメーターがメッセージ送信側によってのみ書き込まれ、(3) メッセージ受信側によってのみ読み取られる限り、この実装にはアクターによる同期が不要です。

`DefaultActorManager` の重要な設計パラメーターには、スレッドとアクターの比率、そして使用するスレッドの合計数の2つが挙げられます。コンピューター上のプロセッサの一部が他の用途に予約されている場合を除き、スレッドはプロセッサと同じ数だけ存在しなければなりません。スレッドは頻繁にアイドル状態になる可能性があるため (例えば I/O 処理の待機中など)、大抵の場合、スレッドの適切な比率はプロセッサ数の2倍以上となります。アクターの数 (実際には、一定時間内のアクター間でのメッセージ数) は一般に、スレッド・プールの大部分をほとんどの時間ビジー状態に維持するのに十分なものでなければなりません (最適な応答のためには、一部の予約スレッドを用意しておく必要があります。通常、負荷状態でのアクティブ・スレッドが平均 75 パーセントから 80 パーセントとなるのが最適です)。これはつまり、アクターの処理対象である処理待ちメッセージがないことも考えれば、通常はスレッドの数よりもアクターの数の方が遥かに多いことを意味します。もちろん、これらの基準は場合に応じて異なります。例えば人間の応答を待機するなど、待機状態に関わるアクションをアクターが実行する場合には、さらに多くのスレッドが必要になります (スレッドは待機中のアクター専用となり、他のメッセージを処理できないためです)。

`DefaultActorManager` は、Java スレッドを有効に利用します。それは、アクターがメッセージを処理している間だけ、その特定のアクターにスレッドを関連付けるためです。それ以外の場合、スレッドは他のアクターが自由に使用できるようになります。このように、一定サイズのスレッド・プールで無数のアクターに対応できることから、所定のワークロードを作成するために必要なスレッド数が少なくなります。これは重要な点です。スレッドは非常に重たいオブジェクトであり、ほとんどの場合はホスト・オペレーティング・システムによって比較的少数のインスタンスに制限されるためです。`µjavaActors` ライブラリーはこの点で、アクターごとに1つのスレッドを割り当てるアクター・システムとの差別化が図られています。アクターごとに1つのスレッドを割り当てる場合、アクターに処理対象のメッセージがなければ、スレッドは事実上アイドル状態となり、存在可能なアクター・インスタンスの数が制限される可能性があります。

`µjavaActors` 実装は、スレッドの切り替えに関しては極めて効率的です。メッセージの処理が完了した時点で、次に処理するメッセージがある場合には、スレッドの切り替えは行われません。単純なループが繰り返されて、新しいメッセージが処理されます。したがって、待機中のメッセージがスレッドの数と同じか、それ以上ある限り、スレッドはアイドル状態にならず、切り替えの必要は発生しないというわけです。十分な数のプロセッサ (少なくともスレッドごとに1つのプロセッサ) があれば、各スレッドを効果的にプロセッサに割り当てられるので、スレッドの切

り替えが起こることは決してありません。バッファリングされたメッセージの数が十分でない場合、スレッドはスリープ状態になります。けれども、オーバーヘッドは処理待ちのワークがないときにしか発生しないため、これは大した問題ではありません。

JVM に対応した他のアクター・ライブラリー

JVM に対応したアクター・ソリューションは他にも存在します。表 1 に、そのうちの 3 つを `µJavaActors` ライブラリーと比較して簡単に紹介します。

表 1. `µJavaActors` と他の JVM アクター・ライブラリーとの比較

名前	参照先	説明	<code>µJavaActors</code> との比較
Kilim	http://www.malhar.net/sriram/kilim/	軽量のスレッドをベースに、複数プロデューサーと単一コンシューマーのメール・ボックス・モデルをサポートする Java ライブラリー。	Kilim にはバイト・コード調整が必要です。 <code>µJavaActors</code> では、各アクターがメール・ボックスを持っているため、別個のメール・ボックス・オブジェクトは必要ありません。
Akka	http://akka.io/	アクターのパターン・マッチングを関数型言語でエミュレーションしようとするライブラリー。通常は、 <code>instanceof</code> による型チェックを使用します（一方、 <code>µJavaActors</code> はストリングの比較または正規表現によるマッチングを使用するのが通常です）。	Akka は関数型に近い（例えば、分散アクターをサポートしているなど）、 <code>µJavaActors</code> よりもサイズが大きく、ほぼ確実に複雑になります。
GPars	http://gpars.codehaus.org/Actor	Groovy のアクター・ライブラリー。	<code>µJavaActors</code> と似ていますが、より Groovy 開発者向けに作られています。

表 1 に記載した JVM アクター・ソリューションの一部では、同期送信が追加されることに注意してください（つまり、送信側が応答を待機します）。これは便利な一方、メッセージ処理の公平性が損なわれたり、場合によってはアクターに対してリエントラント（再入可能）な呼び出しが行われたりする可能性があります。`µJavaActors` では従来からの実装である POJT (Plain Old Java Thread) や標準的なスレッド・モニターを使用しますが、表 1 に記載した JVM アクター・ソリューションのなかには、特殊なサポートによって独自のスレッド・モデルを提供するものもあります。`µJavaActors` は Pure Java ライブラリーであるため、その JAR がクラスパスに設定されていることを確認するだけで、`µJavaActors` を使用することができます。バイト・コードの操作やその他の特殊なアクションは必要ありません。

`µJavaActors` の機能強化

`µJavaActors` ライブラリーにも当然、改善あるいは拡張の余地があります。参考いくつかの可能性を示して、この記事締めくくります。

- ・ カテゴリー内で処理待ちメッセージを再分配する: 現在、送信されたメッセージはラウンドロビン方式で割り当てられますが、それ以降にバランスを取るために再分配されることはありません。
- ・ 優先度を考慮してアクターを実行する: 現在、すべてのアクターは同じ優先度のスレッドで実行されます。異なる優先度のスレッド（またはスレッド・プール）があり、条件の変更に応じ

て、アクターをこれらのスレッドに割り当てることができたとすれば、より柔軟なシステムになります。

- メッセージの優先度を考慮する: 現在、メッセージは一般に送信順に処理されます。優先度を考慮して処理することができると、より柔軟性のある処理を実現できます。
 - 複数のカテゴリーのメッセージをアクターが処理できるようにする: 現在アクターは、一度に1つのカテゴリーにしか属することができません。
 - スレッドの切り替えを減らすように実装を最適化することによって、受信するメッセージの処理速度を改善する: この場合、複雑さが増すという犠牲が伴います。
 - アクターを分散させる: 現在のところ、すべてのアクターを単一の JVM で実行しなければなりません。複数の JVM 間での実行は、強力な拡張になります。
-

ダウンロード

内容	ファイル名	サイズ
Actor runtime and actor demo source	j-javaactors.jar	104KB
Java source files	j-javaactors.zip	47KB

著者について

Barry A. Feigenbaum, Ph.D.



Barry Feigenbaum は、現在 Dell に勤務するソフトウェア・エンジニアです。以前は、IBM そして Amazon で働いていました。Sun (現在は Oracle) Certified Java Programmer, Developer and Architect の肩書きを持つ彼は、他にも developerWorks 記事を投稿している他、JavaOne などのコンファレンスでの発表、技術書の著作活動を行っています。彼はコンピューター・エンジニアリングの博士号を持っています。

© Copyright IBM Corporation 2012

(www.ibm.com/legal/copytrade.shtml)

商標

(www.ibm.com/developerworks/jp/ibm/trademarks/)