Javaの理論と実践: 割り込み例外の処理

キャッチした例外をどうするか

Brian Goetz
Principal Consultant
Quiotix

2006年 5月 23日

Thread.sleep() やObject.wait() など、JavaTM言語のメソッドの多くは割り込み例外をスローします。これはチェックされる例外なので無視することはできませんが、どう処理すればいいでしょうか。今月の「Javaの理論と実践」では、並行アルゴリズムの専門家であるBrian Goetzが割り込み例外とは何か、なぜスローされるのか、キャッチしたらどうすべきかについて説明します。

このシリーズの他の記事を見る

テスト・プログラムを書いていて、処理を一時的に停止したい場合に、Thread.sleep() を呼び出すとします。しかしこの場合、コンパイラーやIDEからは、チェックされる割り込み例外が処理されていないというメッセージが返ってきます。おそらく、誰にでも経験のあることでしょう。この割り込み例外とは何でしょうか。なぜ処理する必要があるのでしょうか。

割り込み例外に対して何も処理しないというのが、最も一般的な対応になっています。つまり、後でリスト4に示すように、キャッチして何もしないということです（またはログに記録する場合もありますが、この方法も大した違いはありません）。あいにくこの方法では、割り込みが発生したという事実に関する重要な情報を見逃すことになり、アクティビティーのキャンセルやサービスのシャットダウンを適切なタイミングで実行するというアプリケーションの機能を損なうことになりかねません。

ブロッキング・メソッド

メソッドが割り込み例外をスローする場合、このメソッドは特定のチェックされる例外をスローできるということの他にも、いくつかのことが分かります。つまり、このメソッドがブロッキング・メソッドであるということや、適切な要求を出した場合は、アンブロックを行って早期リターンを実行することなどが分かります。

ブロッキング・メソッドは、実行に時間がかかるだけの通常のメソッドとは異なります。通常のメソッドの完了を左右するのは、要求した作業の量と適切なマシン・リソース（CPUとメモ

リー) が使用可能かどうかということだけです。一方、ブロッキング・メソッドの完了は、タイマー切れ、I/Oの完了、別のスレッドのアクション (ロックの解放、フラグの設定、作業キューへのタスクの挿入) など、外部イベントにも依存します。通常のメソッドは作業の完了と同時に完了しますが、ブロッキング・メソッドは外部イベントに依存するため、完了を予測することは困難です。このように、ブロッキング・メソッドはいつ完了するか予測しにくいため、応答が遅くなる場合があります。

ブロッキング・メソッドは、待機しているイベントが発生しなかった場合、いつまでも終了しない可能性があるため、ブロッキング操作をキャンセル可能にする際に便利なことがあります (時間のかかる非ブロッキング・メソッドをキャンセル可能にする場合にも便利です)。キャンセル可能な操作とは、通常であれば自ら処理を終了する場合に、それに先立って外部からその処理を完了させることが可能な操作のことです。Threadによって提供され、Thread.sleep() およびObject.wait() によってサポートされる割り込みメカニズムは、キャンセル・メカニズムです。すなわち、スレッドは別のスレッドに対して、実行中の処理を停止するように要求することができます。メソッドが割り込み例外をスローした場合、そのメソッドを実行しているスレッドに対して割り込みが発生したときは、実行中の処理を停止して早期リターンを実行することになります。つまり、割り込み例外がスローされたときに早期リターンが実行されるということがわかります。適切に設計されたブロッキング・ライブラリー・メソッドとは、応答が遅れることなくキャンセル可能なアクティビティー内で使用できるように、割り込みに応答して割り込み例外をスローできるメソッドです。

スレッド割り込み

すべてのスレッドには割り込みステータスを表すBoolean属性が関連付けられています。割り込みステータスは、はじめはfalseです。あるスレッドがThread.interrupt() の呼び出しを通じて別のスレッドによって割り込まれると、次に挙げる2つの処理のうち、いずれかの処理が行われます。このスレッドがThread.sleep()、Thread.join()、Object.wait() などの低レベルの割り込み可能ブロッキング・メソッドを実行していた場合は、アンブロックを実行して割り込み例外をスローします。それ以外の場合は、interrupt() によってスレッドの割り込みステータスがセットされるだけです。割り込まれたスレッドで実行中のコードにより、後で割り込みステータスをポーリングして、実行中の処理の停止要求が出されたかどうかを確認することができます。割り込みステータスはThread.isInterrupted() で読み取ることができます。Thread.interrupted() という名前のメソッドを使用すると (メソッドの名前が機能に合っていませんが)、割り込みステータスの読み取りとクリアを1回の操作で行うことができます。

割り込みは、協調メカニズムです。あるスレッドが別のスレッドに割り込んだとき、割り込まれたスレッドは実行中の処理を直ちに停止するとは限りません。割り込みとは、別のスレッドに対して、「できれば、実行中の処理を都合のよいときに停止してください」と、ていねいにお願いする手段です。Thread.sleep() のように、この要求にきちんと応えるメソッドもありますが、そもそもメソッドには割り込み要求に応える義務はありません。実行に時間がかかるが、ブロック化しないメソッドは、割り込みステータスをポーリングすることによって割り込み要求を優先し、割り込まれた場合は早期にリターンします。割り込み要求を無視するのは自由ですが、その場合は応答が遅くなる可能性があります。

割り込みが持つ協調性の利点の1つは、キャンセル可能なアクティビティーを安全に構築できる柔軟性が得られることです。アクティビティーを直ちに停止する必要がある場合はまれです。アク

ティビティーが更新中にキャンセルされた場合、プログラム・データ構造が不整合状態のままになってしまいます。割り込みによって、キャンセル可能なアクティビティーは進行中の作業をクリーンアップして不変条件を復元し、他のアクティビティーにキャンセルを通知してから終了することができます。

割り込み例外の処理

割り込み例外をスローするメソッドがブロッキング・メソッドであるとすれば、ブロッキング・メソッドを呼び出すメソッドもブロッキング・メソッドになります。この場合、割り込み例外の効果的な処理方法を用意しておく必要があります。よく使用される最も簡単な方法は、リスト1のputTask() とgetTask() メソッドに示されているように、割り込み例外を自分でスローすることです。こうすると、メソッドが割り込みに対して応答するようになります。通常の場合、throws節に割り込み例外を追加するだけで済みます。

リスト1. 割り込み例外をキャッチせずに呼び出し元に伝える

```
public class TaskQueue {
    private static final int MAX_TASKS = 1000;

    private BlockingQueue<Task> queue
        = new LinkedBlockingQueue<Task>(MAX_TASKS);

    public void putTask(Task r) throws InterruptedException {
        queue.put(r);
    }

    public Task getTask() throws InterruptedException {
        return queue.take();
    }
}
```

例外を伝える前に、ある程度のクリーンアップが必要な場合があります。この場合は、割り込み例外をキャッチしてクリーンアップを行ってから、例外を再びスローします。リスト2にこの技法を示します。これは、オンライン・ゲーム・サービスでプレイヤーをマッチングするためのメカニズムです。matchPlayers() メソッドは、2人のプレイヤーが到着するのを待って、新しいゲームを開始します。最初のプレイヤーが到着した後、2人目のプレイヤーが到着する前に割り込みが発生した場合は、1人目のプレイヤーのプレイ要求が失われないように、そのプレイヤーをキューに戻してから割り込み例外を再びスローします。

リスト2. タスク固有のクリーンアップを実行してから、割り込み例外を再スローする

```
public class PlayerMatcher {
    private PlayerSource players;

    public PlayerMatcher(PlayerSource players) {
        this.players = players;
    }

    public void matchPlayers() throws InterruptedException {
        try {
             Player playerOne, playerTwo;
             while (true) {
                 playerOne = playerTwo = null;
                 // Wait for two players to arrive and start a new game
```

```
                playerOne = players.waitForPlayer(); // could throw IE
                playerTwo = players.waitForPlayer(); // could throw IE
                startNewGame(playerOne, playerTwo);
            }
        }
        catch (InterruptedException e) {
            // If we got one player and were interrupted, put that player back
            if (playerOne != null)
                players.addFirst(playerOne);
            // Then propagate the exception
            throw e;
        }
    }
}
```

割り込みに対して処理を行う

Runnableによって定義されたタスクが割り込み可能メソッドを呼び出すときなど、割り込み例外をスローできない場合もあります。この場合、割り込み例外を再スローすることはできませんが、何もしないわけにもいきません。ブロッキング・メソッドが割り込みを検出して割り込み例外をスローするときには、割り込みステータスをクリアします。割り込み例外をキャッチしても再スローできない場合は、割り込みが発生した証拠を残す必要があります。この証拠により、呼び出しスタックの上位のコードが割り込みの発生を認識し、その割り込みに対する応答が可能になります。このタスクは、リスト3に示すように、interrupt() を呼び出して現在のスレッドに「再び割り込む」ことによって実行されます。少なくとも、割り込み例外をキャッチして再スローしないときには、リターンする前に常に現在のスレッドに対して再割り込みを行うようにします。

リスト3. 割り込み例外をキャッチした後の割り込みステータスの復元

```
public class TaskRunner implements Runnable {
    private BlockingQueue<Task> queue;

    public TaskRunner(BlockingQueue<Task> queue) {
        this.queue = queue;
    }

    public void run() {
        try {
             while (true) {
                 Task task = queue.take(10, TimeUnit.SECONDS);
                 task.execute();
             }
         }
         catch (InterruptedException e) {
             // Restore the interrupted status
             Thread.currentThread().interrupt();
         }
    }
}
```

割り込み例外の最悪の処理方法は、何も処理を行わないことです。つまり、キャッチした後、再スローもせず、スレッドの割り込みステータスの再アサートもしないことです。予定外の例外に対する標準的な処理方法、すなわち、キャッチしてログに記録するという方法も、大した違いはありません。この方法では、呼び出しスタックの上位のコードは割り込みに気づかないためです（割り込み例外をログに記録するのは、無意味ですらあります。人間がログを読むときには、すでに手遅れになっているからです）。リスト4は、割り込みに対して何も処理を行わない、よく見られるパターンを示しています。

リスト4. 割り込みに対して何も処理を行わない悪い例

```
// Don't do this
public class TaskRunner implements Runnable {
    private BlockingQueue<Task> queue;

    public TaskRunner(BlockingQueue<Task> queue) {
        this.queue = queue;
    }

    public void run() {
        try {
             while (true) {
                 Task task = queue.take(10, TimeUnit.SECONDS);
                 task.execute();
             }
         }
         catch (InterruptedException swallowed) {
             /* DON'T DO THIS - RESTORE THE INTERRUPTED STATUS INSTEAD */
         }
    }
}
```

割り込み例外を再スローできない場合、割り込み要求に対して応答するにしろしないにしろ、現在のスレッドに対して再割り込みを行ってください。単一の割り込み要求に対して、複数の「受取人」がいる可能性があるためです。標準のスレッド・プール（ThreadPoolExecutor）のワーカー・スレッド実装は割り込みに対して応答するため、スレッド・プールで実行中のタスクに対して割り込みを行うと、タスクをキャンセルする効果と、スレッド・プールのシャットダウンを実行スレッドに通知する効果があります。タスクが割り込み要求に対して何も処理を行わない場合、ワーカー・スレッドは割り込み要求があったことを認識できず、アプリケーションまたはサービスのシャットダウンを遅延させる可能性があります。

キャンセル可能タスクの実装

言語仕様では、割り込みに具体的な意味合い（セマンティクス）は何も与えられていませんが、大規模なプログラムでは、キャンセル以外に割り込みのセマンティクスを維持するのは困難です。アクティビティーによっては、ユーザーはGUIを通じて、またはJMXやWebサービスなどのネットワーク・メカニズムを通じてキャンセルを要求することができます。プログラム・ロジックからも要求することができます。たとえば、Webクローラーによってディスクがいっぱいであることが検出された場合は、Webクローラーを自動的にシャットダウンすることができます。また、並行アルゴリズムによって複数のスレッドを開始してソリューション空間の複数の領域を検索し、いずれかのスレッドがソリューションを検出した時点で、開始されたスレッドをキャンセルすることもできます。

タスクがキャンセル可能だというだけで、割り込み要求に直ちに応答しなければならないわけではありません。コードをループ実行するタスクの場合、ループ内で1回だけ割り込みをチェックするのが一般的です。ループの実行時間の長さによっては、（Thread.isInterrupted() で割り込みステータスをポーリングすることによって、またはブロッキング・メソッドを呼び出すことによって）タスク・コードがスレッドの割り込みを認識するまで時間がかかることがあります。タスクの応答時間を短縮する必要がある場合は、割り込みステータスのポーリングの頻度を高めるようにします。ブロッキング・メソッドは通常の場合、エントリー時に直ちに割り込みステータスをポーリングします。その際、応答時間を短縮するように設定されている場合は、ブロッキング・メソッドによって割り込み例外がスローされます。

唯一、割り込みに対して何も処理を行う必要がないのは、スレッドがまもなく終了するとわかっている場合です。このような状況になるのは、リスト5に示されているように、割り込み可能メソッドを呼び出しているクラスがRunnableまたは汎用ライブラリー・コードではなく、Threadの一部である場合だけです。リスト5のコードは、割り込みが発生するまで素数を列挙するスレッドを作成し、割り込みが発生した時点でこのスレッドを終了させるコードです。この素数検索ループは、whileループのヘッダーでisInterrupted() メソッドをポーリングする部分と、ブロッキング・メソッドのBlockingQueue.put() を呼び出す部分の2か所で割り込みをチェックします。

リスト5. スレッドがまもなく終了するとわかっている場合は、割り込みに対して何も処理を行う必要はない

```
public class PrimeProducer extends Thread {
    private final BlockingQueue<BigInteger> queue;

    PrimeProducer(BlockingQueue<BigInteger> queue) {
        this.queue = queue;
    }

    public void run() {
        try {
            BigInteger p = BigInteger.ONE;
            while (!Thread.currentThread().isInterrupted())
                queue.put(p = p.nextProbablePrime());
        } catch (InterruptedException consumed) {
            /* Allow thread to exit */
        }
    }

    public void cancel() { interrupt(); }
}
```

割り込み不能ブロッキング

すべてのブロッキング・メソッドが割り込み例外をスローするわけではありません。入力および出力ストリーム・クラスは、I/Oの完了を待ってブロックすることがありますが、この場合は割り込み例外をスローせず、割り込みが発生した場合でも早期リターンはしません。しかしソケットI/Oの場合、あるスレッドがソケットをクローズすると、このクローズされたソケットでのブロッキングI/O操作が他のスレッドでも行われている場合、SocketExceptionによってこのブロッキングI/O操作の早期完了が実行されます。java.nioの非ブロッキングI/Oクラスでも割り込み可能I/Oはサポートされていませんが、チャンネルをクローズするかSelectorでのウエイクアップを要求することによって、同様にブロッキング操作をキャンセルすることができます。同様に、組み込みロックの取得中に (同期化ブロックへ入る際に) 割り込むことはできませんが、ReentrantLockによって割り込み可能取得モードがサポートされています。

キャンセル不能タスク

割り込みを単に拒否するキャンセル不能タスクもあります。しかし、キャンセル不能タスクの完了後、呼び出しスタックの上位のコードが割り込みに対して処理を行う必要がある場合には、キャンセル不能タスクであっても割り込みステータスの保存が必要になります。リスト6は、割り込みが発生しているかどうかに関係なく、アイテムが使用可能になるまでブロッキング・キューで待機するメソッドを示しています。この場合、メソッドの終了後にfinallyブロックで割り込みステータスを復元することで、呼び出し元から割り込み要求を奪わないように

しています (無限ループになるため、早期に割り込みステータスを復元することはできません。BlockingQueue.take() はエントリー時に直ちに割り込みステータスをポーリングし、割り込みステータスが設定されていることを検出すると、割り込み例外をスローします)。

リスト6. 割り込みステータスを復元してからリターンするキャンセル不能タスク

```
public Task getNextTask(BlockingQueue<Task> queue) {
    boolean interrupted = false;
    try {
        while (true) {
            try {
                return queue.take();
            } catch (InterruptedException e) {
                interrupted = true;
                // fall through and retry
            }
        }
    } finally {
        if (interrupted)
            Thread.currentThread().interrupt();
    }
}
```

まとめ

Javaプラットフォームによって提供される協調的な割り込みメカニズムを使用して、柔軟性のあるキャンセル・ポリシーを構築することができます。アクティビティーによって、こうしたポリシーがキャンセル可能かどうか、割り込みに対する応答の速さはどの程度必要かが決まります。直ちにリターンするとアプリケーションの整合性が取れなくなる場合は、割り込みを遅延してタスク固有のクリーンアップを行うことができます。コード内で割り込みを完全に無視したい場合でも、割り込み例外をキャッチして再スローしない場合には、必ず割り込みステータスを復元して、割り込みが発生したことが呼び出し元のコードにわかるようにしてください。

著者について

Brian Goetz

Brian Goetz は18 年間以上に渡って、専門的ソフトウェア開発者として働いています。彼はカリフォルニア州ロスアルトスにあるソフトウェア開発コンサルティング会社、Quiotixの主席コンサルタントであり、またいくつかのJCP Expert Groupの一員でもあります。2005年の末にはAddison-Wesleyから、Brianによる著、Java Concurrency In Practiceが出版される予定です。Brian著による有力業界紙に掲載済みおよび掲載予定の記事のリストを参照してください。