Javaの理論と実践: 例外をめぐる議論

チェックすべきか、チェックせずにおくべきか

Brian Goetz
Principal Consultant
Quiotix

2004年 5月 25日

Java言語での例外の使用に関する多くの助言では、例外が捕捉される可能性のある場合にはどこでも、チェック例外が望ましいとされています。この助言はJavaの言語設計（コンパイラーが、投げられる可能性のあるチェック例外をメソッド・シグニチャーの中に強制的にリストアップさせる）においても、またスタイルや使い方に関して初期の頃に書かれたものの中でも推奨されています。最近何人かの著名なライターが、良質なJavaのクラス設計において非チェック例外というのは、以前思われていた以上に意味を持つのだ、という立場を取るようになっています。この記事ではBrian Goetzが非チェック例外を使う是非を見て行きます。

このシリーズの他の記事を見る

C++と同様Java言語でも例外を投げ、捉えることができるようになっています。ただしC++とは異なり、Java言語ではチェック例外と非チェック例外の両方をサポートしています。Javaクラスはそのクラスが投げるチェック例外をすべて、メソッド・シグニチャーの中で宣言する必要があり、またタイプEのチェック例外を投げるメソッドを呼ぶメソッドはどれも、Eを捉えるか、同じくE（またはEのスーパークラス）を投げるとして宣言する必要があります。こうすることでJava言語は、制御がメソッドから出る方法として想定できるものをすべて、強制的に文書化させるのです。

プログラミングの間違いの結果として発生する例外や、プログラムが捉えると想定できないような例外（null ポインターによる参照、配列の最後が見つからない、ゼロでの割り算など）に関して開発者が対処する必要がないように、一部の例外は非チェック例外として指名されており（`RuntimeException`に由来するようなもの）、宣言が不要になっています。

慣習的な知恵

以下はSunの「The Java Tutorial」（参考文献）の抜粋ですが、例外をチェック済みとして宣言すべきか未チェックとして宣言すべきかに関する慣習的な知恵を要約しています。

> Java言語ではメソッドがランタイム例外を捉えること、または規定することを要求してはいないので、プログラマーはランタイム例外のみを投げるようなコードを書い

> たり、全ての例外サブクラスがRuntimeExceptionから継承するコードを書いたりしがちです。こうしたプログラミングのショートカットによってプログラマーはコンパイラーからのうるさいエラーに煩わされず、また例外を規定したり捉えたりすることに頭を悩ませずにJavaコードが書けるようになります。これはプログラマーにとっては重宝ですが、Javaの趣旨である、捕捉するか、規定するか、という要求からは外れており、そのクラスを使うプログラマーが問題に陥ることになります。
> チェック済みの例外は、規則に従って規定されたリクエストの操作に関する有用な情報を表します。こうしたリクエストは呼び出し側が何の制御もできないものですが、呼び出し側が知る必要のあるもので、例えばファイルシステムが現在一杯であるとか、リモート・エンドが接続を終了した、またはこのアクセス権限ではこの操作はできない、といったような情報を表すのです。
> もし規定するのが面倒だからという理由だけでRuntimeExceptionを投げたりRuntimeExceptionのサブクラスを作ったりしたらどうなるでしょう？ 単純に、例外を投げると規定せずに例外を投げることができる、というだけです。言い換えれば、これはメソッドが投げる可能性のある例外を文書化せずに済ませる方法なのです。どういう場合にこれが良いことと言えるのでしょう？ はて、メソッドの振る舞いを文書化せずに済ませて良い場合と言うのは一体あるのでしょうか？ 答えは当然ながら、「そんな場合があるわけがない」のです。

言い換えれば、Sunは通常はチェック例外を使うべきだ、と言っているのです。このチュートリアルはさらに続けて色々な言い方で、（・・・JVMでもない限り）一般的にはExceptionを投げるべきであり、RuntimeExceptionを投げるべきではない、と言っています。

Josh Blochはその著書Effective Java: Programming Language Guide（参考文献）の中で、チェック例外と非チェック例外に関して、次のようなちょっとしたヒントを授けていますが、これは「The Java Tutorial」にも通ずるものです（ただし「The Java Tutorial」ほどには厳しくありません）。

- **39項: 例外は例外的な条件にのみ使う。** つまり、例えばIterator.next()を呼ぶ時に、最初にIterator.hasNext()をチェックする代わりにNoSuchElementExceptionを捉えるなどの制御フローには例外を使わない。
- **40項: 回復可能な条件にはチェック例外を、プログラミング・エラーにはランタイム例外を使う。** ここでBlochはSunが以前から言っているヒント・・ラインタイム例外は前提条件違反などプログラミング・エラーを示すためだけに使うべきである・・というヒントを改めて強調しています。
- **41項: チェック例外を不必要に使わない。** 言い換えれば、呼び出し側がその条件から回復する可能性のない場合、またはその条件から予測できるレスポンスはプログラムが終了することのみの場合には、そうした条件に対してチェック例外を使うなということです。
- **43項: その抽象化に適切な例外を投げる。** 言い換えるとメソッドが投げる例外は、メソッドが何を行うのかと一貫した形で、抽象化レベルで定義すべきであり、必ずしもそのメソッドがどう実装されるかという低レベルの詳細で定義する必要はありません。例えばファイルやデータベース、あるいはJNDIからリソースをロードするメソッドは、リソースが見つからない場合には、何らかのResourceNotFound例外を投げるべき（一般的には下にある原因を保存するために例外チェーンを使います）であって、低レベルのIOExceptionやSQLException、NamingExceptionを投げるべきではありません。

チェック例外の正当性を再検証する

最近、多くの人の尊敬を集めているBruce EckelやRod Johnsonといった何人かのエキスパートが次のように公式に述べています。つまり、自分たちも当初はチェック例外に関する正統的な立場を完全に支持したのですが、結論として分かったのは、チェック例外のみを使うのは当初思われていた程には良い考えではないこと、またチェック例外が、多くの大プロジェクトで重大な問題の元になる、ということなのです。Eckelはさらに極端な立場を取り、全ての例外は非チェック例外であるべきだと言っています。Johnsonはもう少し穏健ですが、やはりチェック例外のみを重視する正統的な手法は行き過ぎだと言っています。（言っておいて良いかと思いますが、Java技術を使っての経験が絶対的に豊富なはずのC#の設計者達はC#の言語設計からチェック例外を無くすという選択をし、全ての例外を非チェック例外にしています。ただしそういう彼らも、チェック例外が後で実装できるような余地は残しています。）

チェック例外に対するいくつかの批判

EckelもJohnsonのどちらも、チェック例外に関する問題点リストとして似たようなものを挙げています。一部はチェック例外固有の特性であり、別の一部はチェック例外をJava言語で特定な実装を行うことによる特性であり、また一部は単なる観察であって、チェック例外の間違った使い方がいかに広まってしまったか、その結果、おそらく例外の機構を考え直す必要がありそうだという状況を述べています。

チェック例外が、実装の詳細を不用意にさらけ出している

データベースやファイルとは何の関係も無さそうなのに、メソッドが`SQLException`や`IOException`を投げるのを何度も見たこと（あるいは書いたこと）がありませんか？ あるメソッドの初期実装で投げられる可能性のある例外を単純に全て集め、そのメソッドのthrows文に加えてしまうのは、開発者にとってはごく普通のことです（多くのIDEではこの作業をするように促しさえします）。このパス・スルー手法の問題はBlochの43項に違反してしまうのです。つまり投げられる例外は抽象化レベルですが、その例外を投げるメソッドとは一貫性がありません。

ユーザー・プロファイルをロードするのが仕事のメソッドは、ユーザーが見つからない時には`NoSuchUserException`を投げるべきであって、`SQLException`を投げるべきではありません。`SQLException`を投げてしまうと、呼び出し側はユーザーが見つからないことは分かるかも知れませんが、`SQLException`をどうすべきかが分かりません。下にある失敗の詳細（例えばスタック・トレースなど）を投げ出すことなく、より適切な例外を投げるためには例外チェーンを使います。それによって抽象化レイヤーがデバッグに有用な情報を保ちつつ、抽象化レイヤーより上の層と抽象化レイヤーよりも下にある詳細を分離できるようになります。

とは言うもののJDBCのようなパッケージでは、この問題を避けにくいように作られてしまっています。JDBCインターフェースにあるどのメソッドも`SQLException`を投げますが、データベースをアクセスする中で様々な問題が起きる可能性があり、様々なメソッドが様々なエラー・モードに陥りやすくなります。`SQLException`はシステムレベルの問題（データベースに接続できない）を意味するかも知れず、論理的な問題（result set にもう行が無い）かも知れず、あるいは特定のデータによるものかも知れません（挿入しようとした行に対する主キーが既に存在する、または実体整合性制約(entity integrity constraint)に違反する）。呼び出し側はメッセージ・テキストを構

文解析するという、許し難い罪悪を犯さない限り、こうした様々に異なるSQLExceptionを区別することができません。（SQLExceptionでもデータベース特有のエラー・コードやSQL状態変数を取り出すためのメソッドを公開しているのですが、現実的にはそうしたメソッドをデータベースのエラー条件を区別するために使うことは稀です。）

不安定なメソッド・シグニチャー

不安定なメソッド・シグニチャーの問題は先の問題に関連しています。単純にメソッドを通して例外を渡していると、メソッドの実装を変える度にそのメソッド・シグニチャーを変える必要があるだけでなく、そのメソッドを呼ぶ全てのコードも変える必要があります。一旦クラスが実稼働状態に展開されてしまうと、繊細なメソッド・シグニチャーを管理するのは高くつくものになります。ところがこの問題は基本的に、Blochの43項のヒントに従わないことから起きる別の症状なのです。メソッドは失敗があった時には例外を投げるべきですが、その例外が反映すべきなのはそのメソッドが何をするかであって、どのようにするか、ではないのです。

実装変更によるメソッド・シグニチャーへの例外の追加削除にプログラマーが疲れて、対象のレイヤーが投げる例外タイプの定義に抽象化を使わず、単純に全てのメソッドがExceptionを投げるように宣言してしまうことが時々あります。別の言い方をすれば、例外はあまりにも面倒すぎると結論し、例外という電源スイッチを切ってしまうのです。当然ですがこの手法は一般的に、どうでも良いようなコード以外では良いエラー処理とは言えません。

読みとれないコード

多くのメソッドが様々な例外を投げるため、実際に何かをするためのコード行数に対してエラー処理のためのコード行数の占める割合は高くなり、メソッドの中で実際に何かをするコードが見つけにくくなります。例外は本来エラー処理を集中化することでコードを小さくできるはずなのですが、3行のコードと6つのcatch ブロックを持つようなメソッドは（それぞれが単に例外をログするだけ、あるいは例外をラップして再度投げるだけならば）、本来単純なコードを水増しし、混乱を招くものに思えます。

例外の飲み込み

例外を捉えてみたらcatchブロックの中に何もない、というコードを何度も見たことがあります。このプログラミング流儀は明らかに悪いものですが、なぜそんなものが出てきたかは簡単に分かります。プロトタイピング中に誰かがそのコードをtry...catchブロックでラップしておいて、後から戻ってそのcatchブロックに中身を入れるのを忘れたのです。このエラーは一般的なものですが、良いツールで防止できるものでもあります。エディターやコンパイラー、あるいは静的検査ツールにとっては、例外が飲み込まれる部分を検出して警告を発するのは簡単なのです。

一般的すぎるtry...catchブロックも、また別の形の例外の飲み込みで、検出がより難しいものですが、これはJavaのクラス・ライブラリにおける例外のクラス階層の（疑わしい）構造によるものです。例えば、あるメソッドが4つ別々のタイプの例外を投げ、そうした例外のどれかに遭遇した呼び出し側はその例外を捉え、ログをとり、戻すものとしましょう。これを実装する方法の一つとしては、try...catchブロックに、それぞれの例外に対応して4つのcatch文を持つようにします。読めないコードの問題を避けるために、一部の開発者はこのコードをリスト1に示すようにリファクターします。

リスト1. 誤ってRuntimeExceptionを飲み込む

```
try { doSomething();
}
catch (Exception e) { log(e);
}
```

このコードは4つのcatchブロックよりはコンパクトですが、問題もあります。このコードではdoSomethingが投げる、どのRuntimeExceptionも捉えてしまい、伝達を妨げてしまうかも知れないのです。

例外ラッピングが多すぎる

例外が低レベルで発生し、何層ものコードを伝達されて上がってくると、最終的に処理されるまでに捉えられ、ラップされ、また何度か投げられることになります。その例外を最終的にログする時にはスタック・トレースは（各レイヤーのラッピング毎に何度も複製されるため）何ページにも渡ってしまいます（JDK 1.4以降の例外チェーンの実装ではこの問題をいくらか軽減しています）。

代替手法

Thinking in Java（参考文献）の著者であるBruce Eckelは、長年Java言語を使っての結論として、チェック例外は間違い、つまり失敗と宣言すべき実験であった、と言っています。Eckelは全ての例外を非チェック例外にすべきだと主張しており、チェック例外を非チェック例外に変換する方法として（特定タイプの例外は伝達されてスタックを上がるうちに捉える、という機能を保持したままの）、リスト2のクラスを挙げています（これをどう使うかについての説明は参考文献にある彼の記事を見てください）。

リスト2. Eckelによる例外アダプター・クラス

```
class ExceptionAdapter extends RuntimeException {
  private final String stackTrace;
  public Exception originalException;
  public ExceptionAdapter(Exception e) {
    super(e.toString());
    originalException = e;
    StringWriter sw = new StringWriter();
    e.printStackTrace(new PrintWriter(sw));
    stackTrace = sw.toString();
  }
  public void printStackTrace() { printStackTrace(System.err);
  }
  public void printStackTrace(java.io.PrintStream s) { synchronized(s) {
      s.print(getClass().getName() + ": ");
      s.print(stackTrace);
    }
  }
  public void printStackTrace(java.io.PrintWriter s) { synchronized(s) {
      s.print(getClass().getName() + ": ");
      s.print(stackTrace);
    }
  }
  public void rethrow() { throw originalException; }
}
```

EckelのWebサイトにある議論を見ていると反応が大きく分かれていることが分かります。一部の人は彼の考えが馬鹿げていると思っており、一部の人は素晴らしい考えだと思っています（私

の意見としては適切に例外を使うのは確かに難しく、また間違った使い方をされる例は多いものの、彼に同意する人たちは間違った理由から同意していると思います。ちょうど政治家が、誰でもチョコレートがもらえるような補助を主張すると10歳の子供から絶大な支持を受けるのと似た話のように思います。 ）

J2EE Design and Development（ 参考文献 ）は私が読んだJava開発やJ2EEなどの本の中で最高のものの一つですが、その著者のRod Johnsonはもう少し穏健な立場を取っています。彼は例外をいくつかの範疇に分け、それぞれに対して方策を特定しています。一部の例外は基本的に二次的な戻りコード（ 通常はビジネスルール違反を知らせます ）であり、一部は「何かがえらくおかしくなった」の類（ 例えばデータベース・コネクションをしようとして失敗、など ）です。Johnsonは最初の範疇にはチェック例外（ 代替戻りコード ）を、後の範疇にはランタイム例外を使うように主張しています。「何かがえらくおかしくなった」範疇では単純に認識するのみで、どの呼び出し側でもこの例外を効果的に処理することはできない、従って途中にあるコードにはほとんど影響を与えずに（ そして例外の飲み込みの可能性もほとんどなく ）伝達されてスタックを一番上まで上がって行くかもしれない、と認識するのです。

Johnsonは中間の立場も指摘しており、「ごく一部の呼び出し側だけがこの問題を処理したいと思っているのだろうか」という疑問を投げかけています。こうしたシナリオでは彼も非チェック例外を使うように言っています。この範疇の例としてJDO例外を挙げています。大部分の場合、JDO例外は呼び出し側が処理したがらないような条件を示すのですが、場合によっては特定なタイプの例外は有用として捉えられ、処理されることもあるのです。こうした例外を捉え、再度投げるという形での一部の可能性のコストを、JDOを使用する他のクラスにも負担させる代わりに、非チェック例外を使ってはどうかと彼は言っています。

非チェック例外を使う

どういう場合に非チェック例外を使うべきか決めるのは難しく、明確な答えがないことも確かです。Sunはどれにも使うな、と助言しており、C#の流儀（ これにはEckelらも同意しています ）ではどれにでも使え、となります。その他の人は「その中間がある」と言っています。

例外はすべてチェック無し、というC++で例外を使った私の経験から言えることは、非チェック例外の一番大きな問題は、チェック例外とは違ってそれ自体に文書化の仕組みが無いことです。APIを作る人が、投げる例外に関して明示的に文書化しない限り、呼び出し側ではそのコードで捉える例外がどんなものなのかを知るすべがないのです。残念ながら私の経験では、大部分のC++ APIは文書化が非常にお粗末であり、良く文書化されたものであっても、与えられたメソッドがどんな例外を投げるかに関する情報は十分ではありません。この問題が、非チェック例外に大きく頼るJavaのクラス・ライブラリでは状況が異なるとも思えません。あなた自身の、あるいは同僚のプログラミング能力に頼るのは非常に難しいのです。自分の主要なエラー処理機構として使おうとしているコール・スタックの、16フレーム下の方にあるコードを書いた誰かの文書化能力に頼らざるを得ないとしたら恐ろしいことです。

文書化の問題はさらに、楽をしようとして非チェック例外を使うのは間違いだということを裏付けています。非チェック例外の文書化はチェック例外の場合よりもずっと重要になるので、非チェック例外を使うパッケージに関する文書化の負担は、チェック例外の場合よりもはるかに高いはずだからです。

文書化、文書化、そして文書化すべし

非チェック例外を使うことを決めたのであれば、その選択を完全に文書化する必要があります。これには、あるメソッドが投げる非チェック例外の全てをJavadocの中に文書化することも含まれます。Johnsonはパッケージ毎にチェック例外を使うか非チェック例外を使うかという選択をしてはどうかと助言しています。非チェック例外を使う時には、（データベース・コネクションを閉じるようなクリーンアップ動作を実行できるように）何も例外を捉えることがないような場合でも`try...finally`をブロックとして使う必要があることも覚えておいてください。チェック例外では`try...catch`が`finally`文を追加するように言ってくれます。非チェック例外では、そうした頼るべき松葉杖のようなものはありません。

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。