今まで知らなかった 5 つの事項: Java 10

ローカル変数型推論が物議の的になるなか Java 10 は JVM 内のガーベッジ・コレクションとコンテナー対応機能に歓迎すべき変化をもたらす

Alex Theedom

2018年 9月 27日

Java 9 のすぐ後にリリースされた Java 10 は、プログラミングのプロ、コンテナー・マニア、ガーベッジ・コレクションの天才にとって嬉しい新機能の宝庫です。JDK 10 の新機能を発見するのに今ほどぴったりのタイミングはありません。

このシリーズについて

皆さんは自分が Java プログラミングについて知っていると思うかもしれません。しかし実際には、ほとんどの開発者は Java プラットフォームの表面的な部分しか扱っておらず、当面の作業を完了するために十分なことしか学んでいません。この連載では、Java テクノロジーを徹底的に追及する著者たちが Java プラットフォームのコア機能を深く掘り下げ、非常に厄介な Java プログラミングの難題の解決にも役立つ、アドバイスとコツを紹介します。

Java 開発者は新しい Java のリリースを長々と待たされることに慣れていますが、こうした状況は新しく採用された短期リリース・サイクルによって一変しています。Java 9 が登場してからわずか 6 か月後に、早くも Java 10 がリリースされました。その 6 か月には、Java 11 のリリースを迎えることになります。開発者の中には、ここまで頻繁なリリースはやりすぎだと思う人もいるかもしれませんが、新しいリリース・サイクルは、長い間必要とされていた変化を指し示しています。

その名にふさわしく、Java 10 が提供する新機能は 10 個あります。この記事では、私が最も重要だと思う 5 つの新機能を取り上げます (すべての新機能を確認するには、このリンク先の Open JDK 10 プロジェクト・ページを参照してください)。

コードを入手する

1. Java の新しいリリース・サイクル

これまでの JDK リリース・サイクルは、注目度の大きい新機能を中心に進められていました。最近の例で言うと、Java 8 ではラムダ式とストリームという形で関数型プログラミングを導入し、Java 9 ではモジュール式 Java システムを導入しています。いずれの新バーションも大いに期待されていたものですが、小さなフィックスについては、主役級のコンポーネントが完成するまで棚上げにされることが通常でした。このことから、Java の進化は他の言語より遅れをとっていたのです。

新しく採用された短期リリース・サイクルは、Java をより細かい増分単位で進化させます。リリース日までに準備できる機能は新しいリリースに盛り込まれ、そうでない機能でもわずか 6 か月後に控えた次のリリースを目標に準備を進めることができます。この新しいリリース・サイクルでの初めての Java バージョンは、2017 年 10 月にリリースされた Java 9 でした。2018 年 3 月には Java 10 がすでにリリースされ、Java 11 のリリースは 2018 年 9 月に予定されています。

新しいリリース・サイクルの一環として、Oracle は各メジャー・リリースのサポートを提供するのは、次のメジャー・リリースまでと発表しています。つまり、Java 11 がリリースされた時点で、Oracle は Java 10 をサポートしなくなります。したがって、開発者は自分が使っている Java バージョンがサポート対象となるよう、6 か月ごとにメジャー・リリースに移行しなければなりません。そこまで頻繁には移行したくない、あるいはその必要がないという開発者は、3 年ごとにアップデートされる LTS (Long Term Support、長期サポート) リリースを利用できます。最新の LTS リリースは Java 8 です。Java 8 は今年の秋に Java 11 がリリースされるまでサポートされます。

2. ローカル変数型推論

ローカル変数型推論は、Java 10 の新機能の中で群を抜いて目立っています。盛んな議論が重ねられた末に、ようやく JDK 10 で導入されたローカル変数型推論は、プログラマーが明示的にローカル変数の型を指定しなくても、その型をコンパイラーで推論できるようにする機能です。

リスト 1 に、Java 10 導入以前の `String` 変数型の定義方法を示します。

リスト 1. String 変数を宣言して割り当てる

```
String name = "Alex";
```

リスト 2 に、Java 10 導入後の `String` 変数型の定義方法を示します。

リスト 2. ローカル変数型推論を使用して定義された String 変数

```
var name = "Alex";
```

ご覧のように、上記の 2 つの方法の違いは、`var` 予約型名を使用するかどうかだけです。ローカル変数型推論では、コンパイラーが式の右辺を使用することで、変数名の型が `String` であると推論できます。

上記の例はあまりにも単純だと思うので、もう少し複雑な例を見ていきましょう。変数がメソッド呼び出しの戻り値に割り当てられているとしたら、どうなるでしょうか？その場合、コンパイラーはメソッドの戻り値の型から、変数の型を推論することができます。

リスト 3. 戻り値の型から推論された String 変数

```
var name = getName();

String getName(){
    return "Alex";
}
```

ローカル変数型の使い方

その名のとおり、ローカル変数型推論機能はローカル変数に対してのみ使用できます。この機能をインスタンス変数やクラス変数を定義するために使用することはできません。また、メソッド・パラメーター内や戻り値の型で使用することもできません。ただし、イテレーターから型を推論できる従来型および拡張 for ループ内で使用することは可能です (リスト 4 を参照)。

リスト 4. ループ内で var を使用する

```
for(var book : books){}
for(var i = 0; i < 10; i++){}
```

ローカル変数型を使用する最も明らかな理由は、コードの冗長さを軽減することです。リスト 5 の例を見てください。

リスト 5. 長々とした型名によって長くなったコード

```
String message = "Incy wincy spider...";
StringReader reader = new StringReader(message);
StreamTokenizer tokenizer = new StreamTokenizer(reader);
```

`var` 予約型名を使用してリスト 5 を作成し直すとどうなるか見てください。

リスト 6. var 型によって冗長さが軽減されたコード

```
var message = "Incy wincy spider...";
var reader = new StringReader(message);
var tokenizer = new StreamTokenizer(reader);
```

リスト 6 に示されている 3 つの型宣言は縦に整列されていて、コンストラクター呼び出しの右側の各ステートメント内では、型が 1 回しか言及されていません。一部の Java フレームワーク内では、長々としたクラス名が一般的に使われています。そのようなクラス名に対してこの型を使用した場合のメリットを想像してみてください。

ローカル変数型に伴う問題

1. var によって型が曖昧になる

上記の例に示されているように、`var` を使用するとコードが読みやすくなりますが、その一方で、コードが曖昧になる可能性があります。リスト 7 の例を見てください。

リスト 7. 不明瞭な戻り値の型

```
var result = searchService.retrieveTopResult();
```

リスト 7 では、戻り値の型を推測しなければなりません。読む側が処理内容を推測しなければならないようなコードは、保守するのが難しくなります。

2. var はラムダ式と相性が悪い

ラムダ式と同時に使用すると、型推論は上手く機能しません。その主な理由は、コンパイラーで利用できる型情報がなくなるためです。リスト 8 に示すラムダ式はコンパイルされません。

リスト 8. 不十分な型情報

```
Function<String, String> quotify = m -> "'" + message + "'";
var quotify = m -> "'" + message + "'";
```

リスト 8 に示されている式の右辺には、コンパイラーが変数型を推論できるだけの十分な情報がありません。ラムダ式を使用するステートメントでは、常に明示的な型を宣言する必要があります。

3. var はダイヤモンド演算子と相性が悪い

ダイヤモンド演算子と同時に使用する場合も、型推論は上手く機能しません。リスト 9 の例を見てください。

リスト 9. ダイヤモンド演算子を使用した var

```
var books = new ArrayList<>();
```

> **自分で試してみてください**
> ローカル変数型推論を試すには、このリンク先のページから JDK 10 をダウンロードしてください。さらに、JDK 10 をサポートする IDE をダウンロードする必要もあります。このリンク先のページからダウンロードできる IntelliJ の EAP (Early Access Program) バージョンは、JDK 10 をサポートします。JDK 10 と IDE をダウンロードしてインストールした後は、この記事に付属の GitHub リポジトリーをチェックアウトしてください。そこに、ローカル変数型推論の例が含まれています。

リスト 9 で、`books` で参照されている `ArrayList` のパラメーターの型が何であるかわかりますか？`ArrayList` に本のリストを格納したいということはわかると思いますが、コンパイラーはそれを推論できません。コンパイラーは唯一できることとして、`Object` 型によってパラメーター化される `ArrayList` を推論します (つまり、`ArrayList<Object>()`)。

別の方法としては、式の右辺に含まれているダイヤモンド演算子内に型を指定します。こうすれば、コンパイラーはそこから変数型を推論することができます (リスト 10 を参照)。このようにしないのであれば、従来の方法で明示的に `List<Book> books` として変数を定義しなければなりません。実のところ、この方法のほうが優先的かもしれません。それは、抽象型を指定して `List` インターフェースに対してプログラムできるためです。

リスト 10. 型を指定する

```
var books = new ArrayList<Book>();
```

3 追加、削除、非推奨

削除

Java 10 では以下のツールが削除されています。

- コマンド・ライン・ツール `javah`。ただし、`javac -h` で代替できます。
- コマンド・ライン・オプション `-X:prof`、ただし、`jmap` ツールを使用してプロファイリング情報にアクセスできます。

- `policytool`

Java 1.2 以降、非推奨として示されていたいくつかの API も、完全に削除されました。それらの API には `java.lang.SecurityManager.inCheck` と以下のメソッドが含まれます。

- `java.lang.SecurityManager.classDepth(java.lang.String)`
- `java.lang.SecurityManager.classLoaderDepth()`
- `java.lang.SecurityManager.currentClassLoader()`
- `java.lang.SecurityManager.currentLoadedClass()`
- `java.lang.SecurityManager.getInCheck()`
- `java.lang.SecurityManager.inClass(java.lang.String)`
- `java.lang.SecurityManager.inClassLoader()`
- `java.lang.Runtime.getLocalizedInputStream(java.io.InputStream)`
- `java.lang.Runtime.getLocalizedOutputStream(java.io.OutputStream)`

非推奨

JDK 10 で非推奨となった API もいくつかあります。`java.security.acl` パッケージは非推奨となりました。そのため、`java.security` パッケージに含まれる各種の関連クラス (`Certificate`、`Identity`、`IdentityScope`、`Singer`、`Policy`) も非推奨となっています。また、`javax.management.remote.rmi.RMIConnectorServer` クラスに含まれる `CREDENTIAL_TYPES` も非推奨となりました。`java.io.FileInputStream` と `java.io.FileOutputStream` 内の `finalize()` メソッドは非推奨となりました。`java.util.zip.Deflater/Inflater/ZipFile` クラスに含まれる `finalize()` メソッドも同じく非推奨となりました。

追加と組み込み

現在進行中の Oracle JDK と Open JDK の調整の一環として、Open JDK に、Oracle JDK 内で有効となっているルート認証局のサブセットが組み込まれるようになりました。これには、Java Flight Recorder と Java Mission Control が含まれます。さらに JDK 10 では、`java.text`、`java.time`、および `java.util` パッケージ内で必要に応じて BCP 47 言語タグの Unicode 拡張サポートが追加されています。グローバル VM セーフポイントを実行せずに、スレッドのコールバックを実行するための新機能も追加されています。これにより、すべてのスレッドを停止するかどうかに関わらず、スレッドを個別に低コストで停止することが可能になっています。

4. コンテナー対応機能の強化

Docker などのコンテナーをデプロイする場合にとりわけ役立つ機能として、JVM がコンテナー内で稼働中であることを認識するようになりました。したがって、JVM は使用可能なプロセッサーの数を、ホスト・オペレーティング・システムではなくコンテナーに照会します。この機能により、コンテナー内で実行される Java プロセスに外部から接続することも可能になるため、JVM プロセスのモニタリングが容易になります。

これまでは、JVM は自身のコンテナーを認識しなかったことから、アクティブな PCU の数をホスト・オペレーティング・システムに照会していました。そのため、場合によっては実際よりも多いリソースが JVM に報告される結果となり、同じオペレーティング・システム上で複数のコ

ンテナーが稼働している場合に問題を引き起こしていました。Java 10 では、ホスト・オペレーティング・システムの CPU のサブセットを使用するようにコンテナーを構成することができるため、JVM は使用中の CPU の数を判断できます。また、`-XX:ActiveProcessorCount` フラグを使用して、コンテナー化された JVM に対して可視にするプロセッサーの数を明示的に指定することもできます。

5. アプリケーションのクラス・データ共有

この機能の目的は、複数の JVM で同じコードを実行していて、始動とシャットダウンを繰り返すときに JVM の実行間の起動時間を短縮するとともに、メモリー使用量を削減することです。この目的を達成するために、JVM 間でクラスに関するメタデータを共有するという方法が取られています。具体的には、JVM の最初の実行で、JVM がロードしたクラスに関するデータを収集して保管します。そのデータ・ファイルを他の JVM や同じ JVM の以降の実行で利用できるようにして、JVM 初期化プロセスの時間とリソースを節約するという仕組みです。クラス・データの共有は、実際にはこれまでも使用できましたが、その使用はシステム・クラスに限られていました。現在、この機能が拡張されて、すべてのアプリケーション・クラスで使用できるようになっています。

まとめ

Java 10 で最も話題を呼んでいる新機能は、明らかに var 予約型名です。`var` 予約型名には、コードを明確かつ単純にする力がありますが、慎重に使わなければコードの意味と目的を曖昧にしてしまう可能性があります。コードが明確でない場合、型を識別するために IDE を利用することもできますが、すべてのコードを IDE 内で読むわけではありません。GitHub リポジトリー、デバッガー、あるいはコード・レビュー・ツール内のコードをオンラインで読むこともよくあります。この新機能を使用する開発者は、今後の読み手や保守担当者にとってコードが読みやすくなるよう配慮してください。

新たに採用された Java の短期リリース・サイクルは、歓迎すべき変化です。リリース日に準備ができている機能はリリースされ、完成が遅れている機能については、短縮されたターンアラウンドで次のリリースを目標に準備を進めることができます。この新しいサイクルは Java の進化を早め、すでに開発されて棚上げされている機能がリリースされるのを開発者が何年も待ち焦がれるといった事態をなくします。メジャー・リリースのサポート有効期間が次のメジャー・リリースにまで短縮されたことについては合理的な懸念があるにせよ、その懸念は LTS によって和らげることができます。開発者はサポートを確実にするために絶えず移行を繰り返さなければならないことから、リリース疲れという別のリスクもありますが、それでも全体的に見ると、短期リリース・サイクルは今後しばらくの間は Java の進化に役立つ前向きな動きだと思います。

関連トピック

- JDK 10 のダウンロード
- JDK 10 での変更の概要
- JDK 10 のソース・コード
- AdoptOpenJDK: ビルド済み OpenJDK バイナリー