Project Lombok によるカスタム AST 変換

カスタム・コードを生成するように Lombok を拡張する場合とその方法

Alex Ruiz
Software engineer
Google

2011年 3月 01日

Alex Ruiz が Project Lombok について紹介するこの記事では、まず、Lombok を独特なものにしているプログラミングの簡略化手法のいくつかを取り上げます。そこでは、アノテーション駆動のコード生成方法や、その簡潔で短く、読みやすいコードについて説明します。次に、より実用的な Lombok の使い方の 1 つに目を向けます。それは、Lombok をカスタム AST (Abstract Syntax Tree: 抽象構文木) 変換で拡張することです。Lombok を拡張することで、独自のプロジェクトやドメインに固有のボイラープレート・コードを生成できるようになりますが、それにはかなりの作業が必要となります。Alex は記事の締めくくりとして、この Lombok を拡張するプロセスの重要なステージを楽に行うためのヒントと併せ、自由に使用できる JavaBeans のカスタム拡張について説明します。

筋金入りの Java™ 開発者にとっても、構文の冗長性は Java 言語でアプリケーションを作成する場合の欠点となるはずです。Groovy などの新しい言語を使用して冗長性を回避するという手段を使えることもありますが、それでも Java コードを使用したほうが好ましい場合や、さらにはそれが必須である場合もあります。そのような場合には、Java プラットフォームのためのオープンソースのコード生成ライブラリー、Project Lombok を試してみてください。

Lombok は、Java アプリケーションの行ごとに現れるボイラープレート・コードを除去することに見事な成果を上げるため、2 度と書くことのなくなる Java 構文はかなりの量になります。けれども Lombok の魅力は構文の簡略化だけではありません。独特なコード生成手法、そして Java 開発に開かれるあらゆる可能性も魅力のうちです。

Project Lombok について紹介するこの記事では、完璧とは言えないながらも、私が Lombok を Java 開発者のツールボックスにぜひ追加しておきたいツールであると考える理由を説明します。まず、Lombok が機能する仕組みとその最適な用途を概説し、Lombok が現在持っている利点と欠点を要約して列挙した後、最も強力であると同時に複雑でもある Lombok の使用例の 1 つに目を向けます。それは、Lombok をカスタム・コード・ベースに合わせて拡張することです。カスタム・コード・ベースは、独自のコードである場合も、Lombok のライブラリーにはまだ含まれていない既存の Java パターンである場合もあります。いずれの場合にしても、記事の残りでは Lombok を拡張するためのヒントと秘訣に焦点を絞り、Lombok API に時間をかけて取り組むだけ

の価値があるのか、それとも特定のプロジェクトにはボイラープレート・コードを作成したほうが良いのかを判断する上でのガイドラインを提供します。

記事に付属のサンプル・コード (「ダウンロード」を参照) は、JavaBeans ボイラープレート・コードを生成するように Lombok を拡張します。このコードは自由に使用することができ、Apache License Version 2.0 の下で提供されています。

Lombok を差別化する特徴

他のコード生成ツールを差し置いて Lombok を使用する最大の理由として考えられるのは、Lombok は単に Java ソースやバイト・コードを生成するだけではないからです。Lombok は AST (Abstract Syntax Tree: 抽象構文木) の構造をコンパイル時に変更することによって、AST の変換を行います。AST は、コンパイラーによって作成される構文解析後のソース・コードのツリー表現であり、XML ファイルの DOM ツリー・モデルに似ています。Lombok はこの AST を変更 (つまり変換) するという方法でソース・コードを簡潔に維持するため、プレーン・テキストのコード生成とは違ってコードが膨れ上がることがありません。Lombok が生成するコードは、CGLib や ASM などのライブラリーで直接バイト・コードを操作する場合とは異なり、同じコンパイル単位に含まれるクラスにも可視になります。

Lombok がコード生成をトリガーするためにサポートするメカニズムは 1 つだけではありません。サポートされるメカニズムの 1 つは、非常によく使われている Java アノテーションです。Java アノテーションを使用することにより、開発者がアノテーション付きのクラスを変更できるようになります。これは、通常の Java アノテーション処理では禁じられていることです。

Lombok の実力を説明する例として、まずはリスト 1 に記載するクラスを見てください。

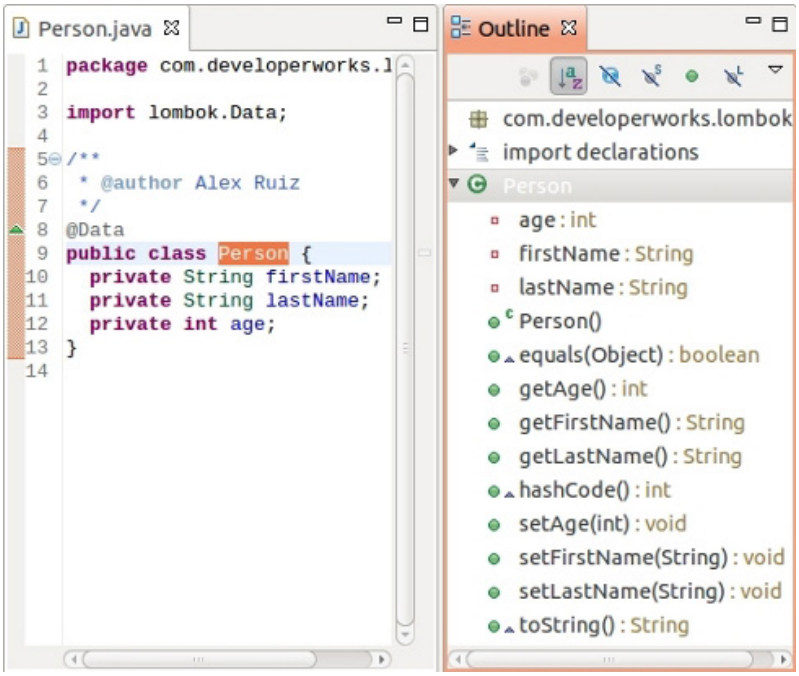
リスト 1. 単純な Java クラス

```
public class Person {
  private String firstName;
  private String lastName;
  private int age;
}
```

コードに `equals`、`hashCode`、および `toString` の実装を追加するのは難しいことではありません。ただ、面倒で、エラーの原因となりやすいだけです。Eclipse などの最近の Java IDE を使用すれば、ボイラープレート・コードのほとんどを自動的に生成することができますが、それでは部分的なソリューションにしかなりません。時間と作業は省けるものの、そこにはコードの読みやすさが損なわれ、理解しにくくなるという犠牲を伴います。通常、ボイラープレート・コードは、アプリケーションのソースにノイズを追加するからです。

その一方、Lombok はボイラープレート・コードの問題に対して賢いソリューションを提供します。リスト 1 の例では、`Person.java` クラスにアノテーション `@lombok.Data` を追加するだけで、必要なメソッドを簡単に生成することができます。図 1 に、Eclipse 内での Lombok のコード生成を示します。「Outline (概要)」ビューのコンパイル・クラスには生成されたメソッドが表示されている一方、ソース・ファイルにはボイラープレート・コードが追加されていません。

図 1. Lombok の動作



Lombok の仕組み

Lombok は、よく使用されている 2 つの Java コンパイラー、javac と ECJ (Eclipse Compiler for Java) をサポートします。この 2 つのコンパイラーは同じような出力を生成しますが、それぞれの実装はまったく異なります。そのため、Lombok にはアノテーション・ハンドラー (Lombok に組み込まれる、コード生成ロジックが含まれるコード) のセットが 2 つ付属しています (それぞれのコンパイラー用に 1 つのセット)。幸い、2 つのセットがあることはユーザーには見えないようになっているので、私たちがユーザーとして扱わなければならないのは、1 つの Java アノテーションのセットだけです。

Lombok は Eclipse とも密接に統合します。Java ファイルを保存すると (気付くほどの遅れもなく) 自動的に Lombok のコード生成がトリガーされるとともに、Eclipse の「Outline (概要)」ビューが更新されて、図 1 のように生成されたメンバーが表示されます。

内部で何が行われているかに興味がある開発者にとっては、Lombok の `delombok` (Maven または Ant コマンドラインからアクセス) がその詳細を明らかにするためのツールとなります。`delombok` は Lombok による変換後のコードを取得し、そこから通常の Java ソース・ファイルを生成します。「delombok された」コードには、Lombok によって行われたすべての変換がプレーン·テキストで含まれます。例えば delombok を図 1 のコードに適用したとすると、`equals`、`hashCode`、`toString` が実際にどのように実装されているかを確かめることができるはずです。

Lombok のセットアップ

Lombok サポートをプロジェクトに追加するには、`lombok.jar` をクラス・パスに追加するだけです。最終的に使用する Lombok 変換によっては、`lombok.jar` を実行時にクラス・パスに組み込む必要がある場合も、そうでない場合もあります (Lombok の次のバージョン 0.10 では、実行時に `lombok.jar` を組み込む必要はなくなります)。Lombok のインストー

ル方法についての詳細は、プロジェクトのホーム・ページ (「参考文献」を参照) にアクセスしてください。

何にでも欠点はあるものです: Lombok を使用する場合のマイナス面

その魅力に飛びついて Lombok をプロジェクトに追加しようとする前に、Lombok にはいくつかの欠点があることを知っておいてください。特に重要なのは、以下の 2 つの点です。

- Lombok の強みは、弱みになる可能性もあります。Lombok に対する主な反対意見は、これが「あまりにも大きな魔法」をかけすぎるという意見です。第 1 に、Lombok が Java コードの冗長性をある程度取り除くことによって、多くのプログラマーが Java 言語で気に入っている「WYSIWYG: What You See Is What You Get (見たままのものが結果となって返ってくる)」という状況が変わってしまいます。Lombok を使用すると、.java ファイルには .class ファイルの内容が示されなくなるからです。
  第 2 に、特定の Lombok 変換は、私たちが知っている Java 構文を根本から変えます。その好例は、`@SneakyThrows` 変換です。この変換によって、チェック例外をメソッド定義で宣言することなく、まるで非チェック例外であるかのようにスローできるようになります。
  リスト 2. @SneakyThrows — かなり狡猾な変換
  ```
  // normally, we would need to declare that this method throws Exception
  @SneakyThrows
  public void doSomething() {
    throw new Exception();
  }
  ```
- Lombok のアノテーション命名規則では、アノテーションの目的が伝わりません。Lombok でのアノテーションは、単なるメタデータではなく、実際にコード生成を行うコマンドとしての役割を持ちます。私は `@GenerateGetter` というアノテーションのほうが、現在の `@Getter` よりもその目的が伝わると思います。

以上の Lombok 固有の問題に加え、Lombok と Eclipse との統合にもいくつかの問題があります。その大部分は、Eclipse が Lombok のコード生成について認識できないことによる結果です。

- Lombok でコード生成が行われている間、Eclipse は時折 `NullPointerExceptions` をスローします。この問題の原因はまだ不明ですが、通常は Eclipse を閉じてから再び開くと、この問題は解決されます。
- Lombok を使用すると、Eclipse でのリファクタリングが難しくなります。例えば、Lombok が生成したゲッターおよびセッターを持つフィールドの名前を Eclipse を使用して変更するには、その場でフィールド名を変更するのではなく、Alt+Shift+R を 2 回押して「Rename Field (フィールドの名前変更)」ダイアログを使用しなければなりません。「Preview (プレビュー)」ステップでは、リファクタリングしているタイプの getXXX とsetXXX のチェック・マークを外す必要があります。
- Lombok が生成したコードには Java ソースがないため、デバッグ作業が多少ややこしくなります。例えば Lombok が生成した `getName` というゲッターのコードを調べようとすると、Eclipse デバッガーは `name` フィールドの `@Getter` アノテーションにジャンプします。この点を抜かせば、Eclipse のデバッガーは Lombok が使用されていても、いつもどおりに機能します。

全体的に見て、これらの問題は対処可能なものであり、いずれは Lombok 開発チームと Eclipse 開発チームによってそのほとんどが解決されると思いますが、それでもやはり、自分が何に関わろ

うとしているのかを知ることは賢明です。これは、自分のツールボックスに新しいツールを追加するときには常に言えることです。

Lombok の拡張

Lombok は一般的な Java ボイラープレート・コードの大多数を生成します。ゲッター、セッター、`equals`、`hashCode` は、そのうちのほんの一例です。これは確かに便利な点ですが、独自のボイラープレート・コードを生成したいという場合もあります。例えば、Lombok ではまだ、JavaBeans などの一般的なコーディング・パターンをサポートしていません。さらに、場合によってはプロジェクトやドメインに固有のコードを生成しなければならないこともあります。

Lombok を拡張する最善の事例として私が発見したのは、プロジェクトの初期段階で新しいコード・パターンのプロトタイプを作成して実験するようなケースです。Lombok を使用すれば、コード・パターンの完成度が高くなるにつれて、その実装を簡単に変更または拡張することができます。具体的には、アノテーション・ハンドラー (コードを生成するために Lombok に組み込むコード) を変更してコンパイルするだけで、すべてのコード・ベースが自動的に更新されます (ただし、生成されたコードの公開規約が変更された場合には、コンパイル・エラーが発生します)。コード・パターンが定まれば、コードを `delombok` することができ、それ以降は通常の Java ソースで作業することができます。

AST 変換とは何か

AST 変換とは、コンパイル・プロセス中に抽象構文木 (Abstract Syntax Tree) の構造を変更するコードに付けられた、いくらか派手な名前です。コードを生成するには、バイト・コードに変換される前に、ノードを追加して AST を変更するほうが有効な方法となります (この方法よりもよく使われている、テキスト・コードの生成や直接バイト・コードを操作する方法と比べた場合の話です)。AST を直接操作するということは、開発者がコンパイラーの API にアクセスしなければならないことも意味します。コンパイラーの API にアクセスすることで、他のコード生成ツールでは提供していない Lombok の機能を利用できるようになり、カスタム・コンパイル・エラーおよび警告を定義することができます。

Lombok を拡張するには、Lombok のコード生成をトリガーするアノテーションを識別または作成する必要があります。続いて、識別したアノテーションのそれぞれに対応するアノテーション・ハンドラーを作成します。アノテーション・ハンドラーとは、2、3 の Lombok インターフェース、そして AST 変換ロジック (別名、コード生成) を実装するクラスのことです。

以降のセクションでは、プロジェクトのセットアップからテストに至るまで、独自の AST 変換を作成する際に参考となる推奨案を提供します。また、JavaBeans をサポートするための Lombok の拡張を具体的に説明するサンプル・コードも記載します。次のセクションではまず、この実用的な Lombok の拡張について詳しく説明します。

Lombok で生成する JavaBeans コード

前述のとおり、Lombok は現在、一般的なコード・パターンをサポートしますが、そのすべてを網羅しているわけではありません。サポートされていないコード・パターンには、JavaBeans も含まれます。Lombok の拡張を具体的に説明するために、私は基本的な処理を行う JavaBeans コードを生成する簡単なサンプル・プロジェクトを作成しました。このプロジェクトは、javac と ECJ のそれぞれに対応するカスタム・アノテーション・ハンドラーで Lombok を拡張する方法を示すとともに、プロセスを大幅に簡潔かつ単純にする有用ないくつかのユーティリティー (例えば、両方のコンパイラー向けのフィールドおよびメソッド・ビルダーなど) のパッケージ化を行います。

サンプル・プロジェクトで使用したのは、Eclipse 3.6 (Helios)、そして Lombok `git` リポジトリーのバージョン 0.10-BETA2 のスナップショットです。コードは、JavaBean の Bound セッターを生成するアノテーション・ハンドラーからなります。記事に付属の zip ファイル (「ダウンロード」セクションを参照) の内容は以下のとおりです。

- Ant ビルド・ファイル
- `@GenerateBoundSetter` および `@GenerateJavaBean` アノテーション
- Bound セッターを生成するアノテーション・ハンドラー (javac 用と ECJ 用)
- 基本的な処理を行ういくつかの JavaBeans コード (`PropertyChangeSupport` フィールドの生成など)

付属のコードは完全に機能し、Apache 2.0 ライセンスの下でライセンスされています。このコードの更新バージョンは、GitHub (「参考文献」を参照) から入手することができます。大体の感じがつかめるように、このコードで実行できる内容を簡単に説明しておきます。

例えば、リスト 3 のコードを作成したとすると、Lombok はアノテーション・ハンドラーを使ってリスト 4 に記載するようなコードを生成します。

リスト 3. Lombok に JavaBean を生成させるコード

```
@GenerateJavaBean
public class Person {
  @GenerateBoundSetter private String firstName;
}
```

リスト 4. 生成される JavaBean サポート・コードの例

```
public class Person {

  public static final String PROP_FIRST_NAME = "firstName";

  private String firstName;

  private PropertyChangeSupport propertySupport = new PropertyChangeSupport(this);

  public void addPropertyChangeListener(PropertyChangeListener listener) {
    propertySupport.addPropertyChangeListener(listener);
  }

  public void removePropertyChangeListener(PropertyChangeListener listener) {
    propertySupport.removePropertyChangeListener(listener);
  }

  public void setFirstName(String value) {
    String oldValue = firstName;
    firstName = value;
    propertySupport.firePropertyChange(PROP_FIRST_NAME, oldValue, firstName);
  }
}
```

サンプル・コードのビルド・ファイルから Eclipse プロジェクトを生成する方法については、サンプル・コードに付属の readme.txt ファイルを参照してください。

拡張の基準

Lombok を拡張してどのコードを生成するかを判断するときには、何らかの基準が必要です。1 つの基準として、そのコードは極めて単純なコードでなければなりません。以降のセ

クションを読むとわかるように、独自の Lombok の拡張を作成するために使用する API は複雑であり、単純な要素を生成するにも、大量のコードが必要になります。Lombok を拡張するのは容易なことではありません。Lombok を保守するとなると、さらに難しく、コストも増えてきます (これは、あらゆるプロジェクトに当てはまります)。Lombok の拡張は、それによって確実に生産性が向上する場合に行ってください。

開始手順: javac と ECJ のどちらをサポートするか

個人的な意見として、Lombok を拡張するには javac と ECJ の両方をサポートする必要があります。少なくとも、現時点ではそう言えると思います。Ant や Maven などのビルド・ツールがデフォルトで使用するコンパイラーは javac ですが、この記事を執筆している時点で最も円滑なコード編集エクスペリエンスを提供するのは、Lombok を Eclipse で使用した場合です。開発者の生産性を考えると、この両方のコンパイラーをサポートすることが必須です。

javac と ECJ は同じような AST 構造を処理しますが、残念ながら、それぞれの実装はかなり異なるため、アノテーションごとに javac 用と ECJ 用の 2 つのアノテーション・ハンドラーを作成せざるを得ません。幸い、Lombok チームはすでに統一した AST API に取り組んでいるので (「参考文献」を参照)、最終的にはアノテーションごとに両方のコンパイラーで機能するアノテーション・ハンドラーを 1 つだけ開発すれば済むようになるはずです。

Lombok のソース・コードの調査

次に必要な作業は、自分が何に関わろうとしているのかを調べることです。それには、ソース・コードに勝る手段はありません。

Lombok はその賢いコード生成手法を実現するために、javac と ECJ の両方で非公開 API を使用します。コードは Lombok に組み込まれることになるため、Lombok と同じ API でないにしても、おそらく同じような API を使用する必要があります。

非公開 API の主な問題は、ドキュメントと安定性が欠けていることです。有難いことに、Lombok チームによると、Eclipse の新しいバージョンでは、移植性に関する問題は起きていません (Java 7 がリリースされた際には、この問題がどうなるかを確認することになります)。したがって、差し当たり対処しなければならない最も大きな問題は、ドキュメントの欠如です。さらに、詳しいドキュメントが作成されているとしても、2 つの異なるコンパイラーの API を習得するのは大変なことであり、時間もかかります。私たちに必要なのは、javac と ECJ の「すぐに使える実用的なガイド」ですが、これについてはこの記事では説明しません。

幸い、Lombok チームでは javac と ECJ を使用して AST ノードを作成する方法をかなり丁寧にドキュメント化しているので、ぜひ、彼らが作成したコードを一読してください。Lombok チームは、変数宣言、メソッド実装などの最も一般的な事例を取り上げています。Lombok のソース・コードを読むことが、javac のAPI と ECJ の API を学ぶ一番の近道となります。リスト 5 に一例として、Lombok チームが作成したソース・コードを記載します。

リスト 5. javac を使用したローカル変数の生成

```
/* final int PRIME = 31; */ {
  if (!fields.isEmpty() || callSuper) {
    statements.append(maker.VarDef(maker.Modifiers(Flags.FINAL),
        primeName, maker.TypeIdent(Javac.getCTCint(TypeTags.class, "INT")),
        maker.Literal(31)));
  }
}
```

ご覧のように、Lombok チームはどのコード・ブロックが何を生成するのかをドキュメント化しています。次回、ローカル変数の宣言を生成する必要が出てきたときは、このソース・コードをもう一度読んで参考にすることができます。

Lombok の .java ファイルを読むだけに終わらせないでください。Lombok の開発者たちも、プロジェクトのセットアップとビルド、そしてアノテーション・ハンドラーのテストに関する指針を提供しています。この後のセクションでは、これらのトピックについて詳しく説明します。

依存関係の管理

一度プロジェクトで依存関係の自動管理を試してみると、以前の手動による依存関係の管理には、なかなか戻れなくなります。Java の世界には、Ivy と Maven を含め、依存関係を管理するビルド・ツールが複数あります (「参考文献」を参照)。しかし、Lombok の拡張を作成する場合には、ツールの選択肢は 1 つに絞り込まれます。それは、Ivy です。

Ivy を選ぶ 1 つの理由としては、Maven では javac などの必要な依存関係がすべて中央リポジトリー内に集められることが挙げられます。そのため、Maven は選択肢から除外されます。もう 1 つの理由は、Ivy は Maven リポジトリーには含まれていない依存関係の管理をサポートしていることです。依存関係をダウンロード可能なリンクは、簡単に指定することができます。この構成にはカスタム ivysettings.xml 構成ファイルが必要になりますが、この点は大したことではありません。

Ant をベースとする Ivy は、ビルドの依存関係を管理します。Lombok チームで使用しているのは、独自に開発した、ivyplusplus という改良版の Ivy です (「参考文献」を参照)。この Ivy の拡張は、例えば依存関係のリストから Eclipse および IntelliJ のプロジェクト・ファイルを作成するなど、有用な Ant ターゲットを提供します。

依存関係を追加する

Lombok 拡張プロジェクトをセットアップするには、以下のファイルが必要です。

- build.xml ファイル: 以下の操作を行う Ant ビルド・ファイルです。
  - 初めてビルドが呼び出されると (指定された場所から) ivyplusplus をダウンロードします。
  - Ivy 構成ファイルの配置場所を指定します。
  - コードのコンパイル、テスト、パッケージ化を行います。
- buildScripts/ivy.xml ファイル: プロジェクトの依存関係を指定します。
- buildScripts/ivysettings.xml ファイル: 取得する依存関係が置かれているリポジトリー (Maven または URL 自体) を指定します。

- buildScripts/ivy-repoフォルダー: ivy.xml に指定された依存関係それぞれの XML ファイルが格納されます。これらの XML ファイルには、依存関係の成果物が記述されます (例えば、成果物のダウンロード元となる場所、ホーム・ページなど)。

一からやり直す必要はありません。時間と作業を節約するには、Lombok によるビルド・ファイルや、この記事に付属のソースに含まれるビルド・ファイルから、必要な部分をコピー・アンド・ペーストしてください。

アノテーションの命名

前にも言いましたが、Lombok のアノテーションは、それ自身が単なるメタデータ以上の役割を果たすものであることをもっと上手に伝えることができるはずです。本来ならば、これらのアノテーションは、ある種のコード生成をトリガーする役目を担っていることを示すべきです。そこで私が強く勧めるのは、Lombok 関連のアノテーションにはすべて、”Generate” という接頭辞を付けるという案です。この記事のソース・コードでは、JavaBeans 関連のソース・コードをトリガーするアノテーションに、`@GenerateBoundSetter` および `@GenerateJavaBean` という名前を付けました。この命名規則であれば、コード・ベースにそれほど詳しくない開発者であっても、少なくともビルド環境のどこかでコード生成プロセスが行われることがわかります。

AST 変換のドキュメント化

Lombok を拡張するときには、ドキュメントの作成が不可欠です。アノテーション・ハンドラーをドキュメント化しておくと、AST 変換を保守する人々の助けになる一方、アノテーションをドキュメント化しておけば、それを使用する人々の役に立ちます。

アノテーション・ハンドラーをドキュメント化する

javac または ECJ の API を使用したコードは、簡単に読んで理解できるものではありません。最も単純な Java コードを生成するコードでも、複雑で長々と続きます。アノテーション・ハンドラーをドキュメント化しておけば、自分にとってもチームにとっても、アノテーション・ハンドラーの保守が遥かに容易になります。ドキュメント化に際して、以下の内容を含めておくと役立つと思います。

- アノテーション・ハンドラーが生成するコードの概要を説明する、クラス・レベルの Javadoc コメント。生成されるコードを説明するのに最も簡単な方法は、コメント内にサンプル・コードを含めることだと思います (リスト 6 を参照)。

  リスト 6. アノテーション・ハンドラーのクラス・レベルの Javadoc

```
/**
 * Instructs lombok to generate the necessary code to make an annotated Java
 * class a JavaBean.
 * <p>
 * For example, given this class:
 *
 * <pre>
 * @GenerateJavaBean
 * public class Person {
 *
 * }
 * </pre>
 * our lombok annotation handler (for both javac and eclipse) will generate
 * the AST nodes that correspond to this code:
 *
```

```
* <pre>
* public class Person {
*
*   private PropertyChangeSupport propertySupport
*       = new PropertyChangeSupport(this);
*
*   public void addPropertyChangeListener(PropertyChangeListener l) {
*     propertySupport.addPropertyChangeListener(l);
*   }
*
*   public void removePropertyChangeListener(PropertyChangeListener l) {
*     propertySupport.removePropertyChangeListener(l);
*   }
* }
* </pre>
* </p>
*
* @author Alex Ruiz
*/
```

- 通常の Javadoc ではないコメント。コード・ベース全体で、それぞれのコード・ブロックが生成する内容を説明します (リスト 7 を参照)。

リスト 7. コード・ブロックが生成する内容のドキュメント化

```
   // public void setFirstName(String value) {
   //   final String oldValue = firstName;
   //   firstName = value;
   //   propertySupport.firePropertyChange(PROP_FIRST_NAME, oldValue,
   //       firstName);
   // }
   JCVariableDecl fieldDecl = (JCVariableDecl) fieldNode.get();
   long mods = toJavacModifier(accessLevel) | (fieldDecl.mods.flags & STATIC);
   TreeMaker treeMaker = fieldNode.getTreeMaker();
   List<JCAnnotation> nonNulls = findAnnotations(fieldNode, NON_NULL_PATTERN);
   return newMethod().withModifiers(mods)
                     .withName(setterName)
                     .withReturnType(treeMaker.Type(voidType()))
                     .withParameters(parameters(nonNulls, fieldNode))
                     .withBody(body(propertyNameFieldName, fieldNode))
                     .buildWith(fieldNode);
```

アノテーションをドキュメント化する

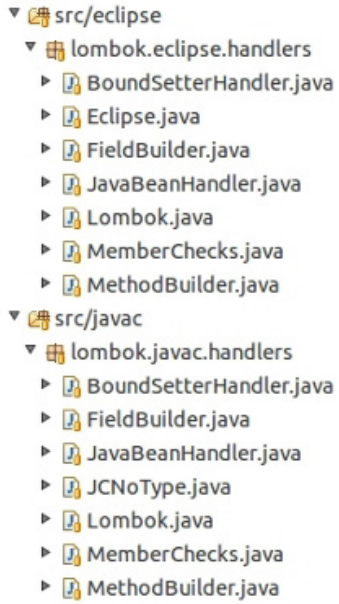
アノテーション・ハンドラーで使用したような (リスト 6 を参照)、クラス・レベルの Javadoc コメントを追加すると、アノテーションのユーザーがそれぞれのアノテーションを使用するとどうなるのかを知り、理解するのに役立ちます。

コンパイラー間の整合性

このヒントが役立つのは、javac と ECJ の両方をサポートすることにした場合のみです。アノテーション・ハンドラーのセットが 2 つある場合には、あらゆるバグの修正、変更、内容の追加は、両方のセット (または枝) に適用しなければなりません。これらの変更は、両方の枝が似ていれば似ているほど、より素早く、より安全に行うことができます。この類似性は、パッケージ・レベルとファイル・レベルの両方に必要です。

パッケージ・レベルの整合性: できる限り、それぞれの枝 (javac と ECJ) のクラス数は同じになるようにすること、そしてクラスには同じ名前を使用することが重要です (図 2 を参照)。

図 2. javac の枝と ECJ の枝とのパッケージの類似性



ファイル・レベルの整合性: 両方の枝には同様の名前が付けられた、ほぼ同じ数のクラスがあることから、同じ名前の 2 つのファイルの内容は、できるだけ同じになるようにする必要があります。フィールド、メソッド数、メソッド名などはすべて、ほぼ同じでなければなりません。リスト 8 に、javac と ECJ 両方の `generatePropertySupportField` メソッドを記載します。AST API は異なりますが、この 2 つのメソッドは非常に良く似ていることに注目してください。

リスト 8. javac と ECJ のアノテーション・ハンドラーの比較

```
// javac
  private void generatePropertyChangeSupportField(JavacNode typeNode) {
    if (fieldAlreadyExists(PROPERTY_SUPPORT_FIELD_NAME, typeNode)) return;
    JCExpression exprForThis = chainDots(typeNode.getTreeMaker(), typeNode, "this");
    JCVariableDecl fieldDecl = newField().ofType(PropertyChangeSupport.class)
                                         .withName(PROPERTY_SUPPORT_FIELD_NAME)
                                         .withModifiers(PRIVATE | FINAL)
                                         .withArgs(exprForThis)
                                         .buildWith(typeNode);
    injectField(typeNode, fieldDecl);
  }

// ECJ
  private void generatePropertyChangeSupportField(EclipseNode typeNode) {
    if (fieldAlreadyExists(PROPERTY_SUPPORT_FIELD_NAME, typeNode)) return;
    Expression exprForThis = referenceForThis(typeNode.get());
    FieldDeclaration fieldDecl = newField().ofType(PropertyChangeSupport.class)
                                           .withName(PROPERTY_SUPPORT_FIELD_NAME)
                                           .withModifiers(PRIVATE | FINAL)
                                           .withArgs(exprForThis)
                                           .buildWith(typeNode);
    injectField(typeNode, fieldDecl);
  }
```

AST 変換のテスト

Lombok が用意しているテスト・インフラストラクチャーのおかげで、カスタム AST 変換のテストは、皆さんが想像するよりも遥かに簡単に行えます。AST 変換のテストがいかに簡単であるかを実証するため、リスト 9 に記載する JUnit テスト・ケースを試してみます。

リスト 9. すべての ECJ アノテーション・ハンドラーのユニット・テスト

```
import static lombok.DirectoryRunner.Compiler.ECJ;

import java.io.File;

import lombok.*;
import lombok.DirectoryRunner.Compiler;
import lombok.DirectoryRunner.TestParams;

import org.junit.runner.RunWith;

/**
 * @author Alex Ruiz
 */
@RunWith(DirectoryRunner.class)
public class TestWithEcj implements TestParams {

  @Override public Compiler getCompiler() {
    return ECJ;
  }

  @Override public boolean printErrors() {
    return true;
  }

  @Override public File getBeforeDirectory() {
    return new File("test/transform/resource/before");
  }

  @Override public File getAfterDirectory() {
    return new File("test/transform/resource/after-ecj");
  }

  @Override public File getMessagesDirectory() {
    return new File("test/transform/resource/messages-ecj");
  }
}
```

このテストは大体のところ、以下のように機能します。

1. `getBeforeDirectory` で指定されたフォルダーに含まれるすべての Java ファイルを、`getCompiler()` で指定されたコンパイラーと Lombok を使用してコンパイルします。
2. コンパイルが完了した後、`delombok` によって、コンパイルされたクラスのテキスト表現を作成します。
3. `getAfterDirectory` に指定されたフォルダーからファイルを読み取ります。これらのファイルには、コンパイルされたクラスに期待される内容が含まれます。テストでは、これらのファイルの内容を、ステップ 2 で取得したソースと比較します。比較する 2 つのファイルには、同じ名前が付いていなければなりません。
4. 今度は、`getMessagesDirectory` に指定されたフォルダーからファイルを読み取ります。これらのファイルには、想定されるコンパイラー・メッセージ (警告およびエラー) が含まれます。テストでは、これらのファイルの内容を、コンパイル中に表示された実際のメッセージ

と比較します (メッセージが表示された場合)。Java ファイルをコンパイルする場合、想定されるメッセージがなければ、メッセージ・ファイルは必要ありません。ファイルの突き合わせは、名前を基準にして行われます。例えば、`CompleteJavaBean.java` をコンパイルする際に想定されるコンパイラー・メッセージがある場合、これらのメッセージが含まれるファイルには、`CompleteJavaBean.java.messages` という名前が付けられます。
5. 想定される出力のすべてが実際の出力と一致すれば、テストは合格です。そうでなければテストは失敗します。

このように、アノテーション・ハンドラーをテストする方法はかなり変わっていますが、以下の点で効果的です。

- アノテーション・ハンドラーごとに 1 つの JUnit テストがあるのではなく、コンパイラー (javac および ECJ) ごとに 1 つの JUnit テストがあります。
- 事例ごとにテスト・メソッドを用意するのではなく、生成されるものとして想定されるコードが含まれるテキスト・ファイルと、想定されるコンパイラー・メッセージが含まれるオプションのテキスト・ファイルを用意します。
- テストは、javac API と ECJ API がどのように使用されているかに関知しません。テストで検証するのは、生成されたコードが正しいかどうかです。

生成されたコードを検証する

想定されるコードをアノテーション・ハンドラーが生成することは、上記で説明したテストで見事に検証することができます。しかし期待している処理を、生成されたコードが実際に行うかどうかをテストする必要がまだ残っています。生成されたコードの振る舞いの正しさを検証するには、AST 変換を使用する Java クラスを作成した上で、生成されたコードの振る舞いをチェックするためのテストを作成する必要があります。基本的には、生成されたコードを、自分で作成したコードであるかのようにテストすることになります。

このようなテストをコンパイルして実行するのに最も簡単な方法は、Ant を使用することです。つまり、javac でコンパイルすることを意味します。コードはテスト済みで、ECJ を使用して生成されたコードが正しいものであることはわかっているため、コードの振る舞いをチェックするテストを Eclipse 内で実行する必要はないと思います (Eclipse 内で実行すると、セットアップが非常に複雑になります)。

この記事のサンプル・コード (「ダウンロード」を参照) に、javac アノテーション・ハンドラーと ECJ アノテーション・ハンドラーのテストを両方とも含めておきました。

まとめ

Project Lombok は、Java コードの冗長性を見事に排除する強力なツールです。Lombok が冗長性を排除する手段は、Java アノテーションとコンパイラー API の賢い独特な使い方です。けれども、あらゆるツールの例に洩れず、Lombok も完璧ではありません。その利点 (短く簡潔なコード) には、犠牲が伴います。それは、Java コードが WYSIWYG 的でなくなること、そして開発者が慣れ親しんでいる IDE 機能の一部が使えなくなることです。Lombok を皆さんのツールボックスに加える前に、必ずその利点と欠点を考慮して、失われるものよりも利点が勝るかどうかを判断してください。

Lombok を使用することにした場合は、独自のボイラープレート・コードを生成するように Lombok を拡張できる可能性があります。現在のところ、Lombok を拡張するのは、簡単なタスクでも、万人向けのタスクでもありませんが、実行するのは不可能ではありません。この記事では、Lombok を拡張するのに適している場合に関するガイドラインを紹介し、その拡張方法を説明しました。Lombok を拡張するために費やす時間と労力が、手作業でボイラープレート・コードを作成するよりも安上がりになるかどうかは、皆さん自身の判断にお任せします。

ダウンロード

内容	ファイル名	サイズ
Source code for this article[1]	j-lombok.zip	63KB

注

1. An updated version of the source code can be found at github.

著者について

Alex Ruiz

Alex Ruiz の楽しみは、Java 開発、オブジェクト指向プログラミング、API 設定、そしてテストに関するあらゆる読み物を読むことです。プログラミングは彼の初恋でした。Alex は、UI テストを作成し、全体的にテストを容易にすることを目指す革新的 Java ライブラリー、FEST の作成者で、Google に勤めています。彼のブログをチェックしてください。