CÉDRIC BEUST

アノテーションの内側に迫る

動作原理と効果的な使用法、独自のアノテーションの作り方

Javaプラットフォームにアノテーションが登場したのは、10年以上前のJava 5でした。それ以降、アノテーションはエコシステムにとって欠かせない要素となっています。本記事では、アノテーションが登場した経緯や理由に関する歴史を簡単に説明した後、アノテーションが動作する仕組み、効果的な使用法、独自のアノテーションの作成といった技術面の詳細について解説します。

起源

ソース・コードにメタデータを追加するという考え方は古くから存在します。そう考えられるようになったのは、かなり多くの場合、記述したコードにはツールの動作に必要な情報がすべて含まれているわけではないという事実が単純に明らかになったためです。Javaプラットフォームでは、初期の形式のアノテーションがJava 5よりも前に姿を現し始めていました。しかし、言語で公式にサポートされていなかったため、あり得ないような話ですが、開発者はJavadocツールを使用してコードにアノテーションを追加していました。2000年代初頭には、次の2つのツールによってアノテーションが注目されるようになりました。

- EJBGen：ソース・コードへのJavadocアノテーションの追加と、複雑なEJB XMLディスクリプタの生成が可能
- XDoclet：EJBGenの考え方を進化させ、JavadocタグをEJBだけでなく任意のドメイン用のアノテーションとして使用する汎用フレームワーク

この2つのツールが急速に人気を集めたことで、JVMがアノテーションを公式にサポートする道が開かれ、JSR 250「Common Annotations for the Java Platform（Javaプラットフォーム向け共通アノテーション）」がこの目的のために策定されて、Java 5とともにリリースされることが決まりました。その際に検討されたのは、アノテーションを型保証にし、開発者が独自のアノテーションを簡単に作成できるよう拡張可能にすることです。

興味深いことに、2004年以降、JDKのアノテーションに関して大きなアップデートが行われたのは、アノテーションを記述できる場所が増えたJSR 308のみです。大規模な変更が加えられたのはそれくらいです。現在使われているアノテーションは、2005年の仕様からほとんど変更されていません（JSR 308で追加された細かなユーティリティについては、2014年3月/4月号で取り上げています）。これから説明する、多くのイノベーションを可能にしたJSR 250が揺るぎない成功を収めていることは、否定しがたい事実でしょう。

アノテーションを使う場面

どのツールにも言えることですが、アノテーションも慎重に使う必要があります。ある種の問題に対しては非常に相性が良い反面、主要な条件が満たされないと、あまり意味のないものになってしまいます。アノテーションの主な代替となるのは、構成ファイルです。構成ファイルには、アノテーションと同じく、コードに含めることができないメタデータを含めることができます。それでは、特定のメタデータをアノテーションとして記述するか、外部ファイルに格納するかはどのように判断すればよいのでしょうか。

一般的な経験則では、そのメタデータがJava要素（メソッド、フィールド、変数、クラス、パッケージなど）に紐付けられる場合、アノテーションに記述するのが望ましいと言われています。そうでない場合は、構成ファイルに格納します。

前述のように、最初にアノテーションの対象となったのはEJBでした。コード内のメソッドを参照して情報を追加する、EJBのデプロイメント・ディスクリプタは、複雑なXMLファイルであるためです。デプロイメント・ディスクリプタを使用するアプローチは、エラーが起こりやすいものでした。コードのリファクタリング（メソッドの名前の変更など）がデプロイメント・ディスクリプタに反映されず、アプリケーションが動作しなくなる可能性があるからです。メタデータの対象となるメソッドにアノテーション

を付加する方がはるかに安全です。メリットとして最初に挙げられるのは、メソッドの名前を別の場所に記述しなくなることで、DRY原則（Don't Repeat Yourself、「重複の排除」）違反がなくなることです。

アノテーションを使用すると、他にも次のようなメリットが得られます。

- **コードの正確性：**@Nullable、@Deprecated、@Overrideなどのアノテーションにより、メソッドやフィールドに重要な意味情報を追加し、コンパイラによってその情報が強制されるようにすることができます。
- **テスト・メソッド：**TestNGとアノテーションが登場するまで、JUnitはリフレクションを使用してメソッドがテスト・メソッドであることを判別していました。この判別には、特別なネーミング規則が必要でした。アノテーションの採用によって、ネーミング規則を使用する必要がなくなりました。
- **永続性**（Hibernateなど）：アノテーションを付加することで、フィールドやメソッドをデータベースに保存されているデータと紐付けることができます。
- **依存性注入：**インジェクションする必要があるクラスにアノテーションを付加し、その旨を示すことができます。フィールドやパラメータについても同様です。
- **グラフィック・ツールキット：**たとえば、Androidではグラフィック・レイアウトをXMLで記述します。アノテーションを使用すると、グラフィック要素（テキスト・ビュー、ボタンなど）とその参照を格納するフィールドを直接紐付けることができます。

上記の例のすべてで、情報とJava要素を紐付けるという同じ特性が使われていることに注目してください。一方で、Javaアノテーションには適さないメタデータもあります。たとえば、次のようなものです。

- ホスト名、ポート、認証情報（たとえばパスワード）などのデプロイメント情報
- アプリケーションにデータベースとの接続方法を伝えるコネクション・プール
- アプリケーションの起動方法や実行時にアクセス可能な情報の種類を記述したパラメータ

重要なアノテーション

最新のJavaでは、アノテーションを避けて通ることはほぼ不可能であり、多くのライブラリで独自のアノテーションが提供されています。しかし、恒常的に使用すべき重要なアノテーションはさほど多くはありません。

@Nullableと@Nonnull（javax.annotation）：フィールドおよびメソッド・パラメータに付加でき、変数がnull値を取ることができるかどうかを指定します。このアノテーションは非常に有用であり、Javaコンパイラ（javac）を使用する多くのツール（主要なIDEが代表的な例です）がこの2つを認識します。可能な限り、常にこのアノテーションを使用するようにします。コードベースにおけるnullポインタ例外の数が急激に減ることにすぐ気付くはずです。

@Override：すでにおなじみでしょう。Java 6以降必須になっており、それには正当な理由があります。このアノテーションは、親インタフェースまたは親クラスのメソッドをオーバーライドするすべてのメソッドに付加する必要があります。この付加によって、意図せずにメソッドをオーバーライドしてしまうことや、逆に、タイプ・ミスのためにオーバーライドしていなかったメソッドを「オーバーライドした」と思ってしまう事態を防ぎます。

@FunctionalInterface：Java 8で追加されました。このアノテーションが付加されたインタフェースが実際に関数型インタフェース（1つの抽象メソッドのみが含まれるインタフェース）であることを保証するためのものです。この背景には、そのインタフェースに意図せず抽象メソッドが追加された際に、コンパイラにエラーを発行させるという意図があります。

@SuppressWarnings：このアノテーションは文字どおり警告を抑制します。一般的に、警告は非常に役立つものであり、グローバル・レベルで警告を止めるべきではありません。しかし、コードが安全であることがわかっていてもコンパイラが警告を出力する場合など、特定の文や式で警告を止めることができると便利な場合もあります。

アノテーションの記述

次に、一般的なアノテーション@Testについて見ていきます。このアノテーションは次のように定義されています。

```
@Retention(RUNTIME)
@Target({METHOD})
```

```
public @interface Test {
```

上記のスニペットの各行に重要な意味があります。下の行から順に確認します。

```
public @interface Test {
```

この構文はアノテーション特有のものです。アノテーションはJavaクラスのように見えますが、いくつかの制限があります。この制限については、後ほど説明します。

```
@Target({METHOD})
```

アノテーションを付加できるJava要素を指定します。METHODは、java.lang.annotation.ElementType列挙型の一部で、他にもCLASS、CONSTRUCTORなどの場所を定義できます。

```
@Retention(RUNTIME)
```

アノテーションをクラス・ファイルに保存するか、コンパイラで破棄するかを指定します。
リフレクション経由でアノテーションを検索できるようにする場合、RUNTIMEを指定します。その他のオプションは、java.lang.annotation.RetentionPolicyクラスで定義されています。

アノテーションの内容

次のようにすると、アノテーションに追加パラメータを渡すことができます。

```
@Test(description = "Verify that bug #121 is fixed")
@Table(name = "ACCOUNTS")
```

この追加パラメータは属性と呼ばれ、アノテーションの宣言内でメソッドとして定義します。

```
public @interface Test {
    String description() default "";
}
```

属性は本体のないメソッドで、オプションでデフォルト値を割り当てることもできます。defaultキーワードを使用しない場合は、アノテーションを使用する際に属性の指定が必須になります。属性を指定しない場合、コンパイラがエラーを発行します。属性には、プリミティブ型または文字列の定数（かつnull以外）でなければならないという重要な制限があります。

アノテーションには、アノテーションを使用するコードの冗長性を減らすための2つの特徴的な仕様が定義されています。

アノテーションでvalueという特別な名前の属性を定義すると、valueを省略してその属性を指定できます。たとえば、次のアノテーション

```
public @interface Person {
    String value();
}
```

は、以下のように書くことができます。

```
@Person("John")
```

一般的な記述方法に従うと、次のようになります。

```
@Person(value = "John")
```

同じように、Array型の属性は、配列の要素が1つだけの場合、省略して記述できます。

```
public @interface Languages {
    String[] value();
}
```

上記のアノテーションは、次のように記述できます。

```
@Languages("English")
```

冗長的に書くと、次のようになります。

```
@Languages(value = { "English" })
```

このような省略構文は、アノテーションを使用する際に必要となる定型的なコードを減らすことを意図しています。

アノテーションの動作

ここまでで、シンプルなアノテーションを定義できるようになりました。しかし、実際にはどのように使えばよいのでしょうか。設計上、コンパイラはサード・パーティの開発者が定義したアノテーションを完全に無視します。コンパイラに影響を及ぼす一部の例外（@Deprecatedなど）もありますが、アノテーションが付加されることによってコードの意味が変わることは基本的にありません。したがって、アノテーションを活用するためには、アノテーションに対して何らかの動作を行うツールを作成する必要があります。

作成できるツールには、外部ツールとアノテーション・プロセッサの2つがあります。

外部ツール

外部ツールは、例外を処理するもっともシンプルなアプローチです。具体的には、main()メソッドを持つ別のアプリケーションを実装します。アノテーションのユーザーに必要なのは、自分が作成したクラスに対してそのツールを実行することのみです。このアプローチは、TestNG、JUnit、Guiceなどの有名なツールで使用されています。このようなツールは、一般的にビルドの一部として実行されます。ツールが出力する内容は、ソース・ファイル、ドキュメント・ファイル、XMLなど非常に多様です。実際、このツールは何でもできます。

JDKのリフレクション・パッケージには、クラス・ファイル内に散在するアノテーションを検索するAPIが含まれています。たとえば、あるクラスに含まれるすべてのアノテーションの取得、または特定のアノテーションのみの取得のいずれかが可能です。次のコードについて考えてみます。

```
@Languages({ "English", "French"})
class MyClass { … }
```

次のようにすると、アノテーションを検索できます。

```
Languages[] languages =
    getClass().getAnnotationsByType(Languages.class);
for (Languages language : languages) {
  System.out.println(
      "Languages spoken: " + language.value());
}
```

アノテーション・プロセッサ

比較的最近のJDKで追加されたアノテーション・プロセッサによって、アノテーションの分野でまったく新しいレベルのイノベーションが実現できるようになりました。アノテーション・プロセッサが生まれた背景には、アノテーションを処理するツールの大多数がJavaソース・ファイルを生成していることがわかったという要因があります。ソース・ファイルはその後コンパイルする必要があるため、アノテーションの処理をJavaコンパイラ内に統合して効率化できれば便利と考えられました。

背景にある発想は、コンパイラに対してアノテーション・プロセッサを宣言するというものです。するとコンパイラは、プロセッサを最初に起動し、出力された結果を自動的にコンパイルします。そして、コンパイルしたクラスを自身のクラスパスに追加した後、通常の処理に戻ります。

この機能のAPIは、先ほど説明したリフレクションのコードとは少し違い、いくつかの変更が行われています。たとえば、プログラムを記述してアノテーションを検索するのではなく、コンパイラがアノテーションを発見した場合にその都度通知するようになっています。このアプローチははるかに効率的です。

アノテーション・プロセッサの記述は複雑なものです。これを説明するためには丸1本の記事が必要になるでしょう。そのため、ここではアノテーション・プロセッサの価値についてのみ説明します。

ソース・コードを生成するという考え方は、JVMで始まったことではありません。しかし、アノテーション・プロセッサはコンパイラの処理を直接フックするため、出力結果を組み立てて処理する労力が大幅に軽減されます。アノテーション・プロセッサは、複数のメカニズムによって完全にシームレスに実行できるようになっています。たとえば、プロセッサを含むJARファイルがクラスパス上にありさえすれば、javacはアノテーション・プロセッサを検知して実行します。

アノテーション・プロセッサを使うと、ライブラリ開発者はリフレクションの呼出しを直接呼出しに置き換えることができます。Androidでは、リフレクションの呼出しが遅いことが大きな懸念事項になっているため、アノテーション・プロセッサがとりわけ重要になっています。さらに、ソー

ス・コードが生成されるため、コードが正しいことを静的に検証できます。この検証は、リフレクションでは実現できないことです。

まとめ

アノテーションはJavaプラットフォームの不可欠な要素で、言語やエコシステムが革新的かつ生産的な方向に進化する原動力となってきました。これはアノテーションだけが実現できたことです。コードベースでアノテーションを使用することについて適切な意思決定を下すためには、アノテーションの動作原理やアノテーションで何ができるかを理解しておくことが重要です。**</article>**

Cédric Beust（@cbeust）：1996年よりJavaコードを書き続けており、長年にわたって言語やライブラリの発展に積極的に貢献。フランスのニース大学でコンピュータ・サイエンスの博士号を取得。JVMのアノテーションの設計を行った専門家グループの一員。

learn more

- オラクルのJavaアノテーション・チュートリアル
- Checker Framework（コードのチェックにアノテーションを多用）

Java開発者向けのOracle Cloud Services

Oracleでは、Javaアプリケーションの開発やデプロイに使用できるクラウド・サービスを展開しています。これらのサービスは、Oracleが提供している既存のサービスを補完するもので、すべての主要なクラウド層、すなわちInfrastructure as a Service（IaaS）、Platform as a Service（PaaS）、Software as a Service（SaaS）に対して提供されています。これらのサービスの展開によって、Java開発者が興味を持つ複数のクラウド・サービスが生まれています。3つのサービスの技術的な側面については、次号以降で詳しく取り上げる予定です。

Oracle Application Container Cloudを活用すると、専用の独立したJava SEおよびNodeランタイム・アプリケーション・コンテナをセルフサービスによって短時間でクラウドにプロビジョニングできます。コンテナはOracle JDK（バージョン7または8）を実行しており、標準のJDKでは利用できないOracle Java Flight Recorderツールも含まれています。また、サーバー・サイドのJavaScript環境であるNode.jsも提供されています。使用料は、月ごとまたは時間ごとにRAMのギガバイト単位で課金されます。

Oracle Developer Cloud Serviceは、「企業向けPaaS環境」と説明されており、Oracle Java Cloud Service（次の項目を参照）を使うと無償で利用できます。Oracle Developer Cloud Serviceには、Git、Maven、およびHudson（継続的インテグレーション・ツール）のインスタンス、タスク・ツール、Wikiが含まれています。Hudsonインスタンスは、3つの同時ビルドが可能です。

Oracle Java Cloud Serviceでは、クラスタ内または専用仮想マシン上で実行されるOracle WebLogic Server（11gまたは12c）が提供されます。オプションでOracle Coherenceキャッシュとインメモリ・データ・グリッドも提供されています。追加のSaaS Extensionによって、Oracle Software as a Service（Oracle Sales Cloud、Oracle Service Cloud、Oracle Marketing Cloudなど）との統合も可能です。