Javaの理論と実践: それをドキュメント化しなければならないか

Javadocに準拠した統合ドキュメンテーションの利点と欠点

Brian Goetz
Principal Consultant
Quiotix

2002年 8月 01日

Java言語は、Javadocコメント規約によるAPIドキュメンテーションへの統合アプローチを採用しています。Javadocツールによって、見た目には美しいAPIドキュメンテーションを生成することはできますが、実際のJava APIドキュメンテーションの大半は実にひどいものです。APIのドキュメンテーションはソース・コードの一部であるため、これは完全に技術者の責任です。今回の記事で、Brian Goetz氏は、Javaドキュメンテーションの現状について苦言を呈し、もっと有益なJavadocを作成するためのいくつかのガイドラインを提示しています。

このシリーズの他の記事を見る

ほとんどのJavaクラス・ライブラリーに関して、Javadocが唯一のドキュメンテーションです。さらに、市販のソフトウェア・コンポーネントを除いて、多くのJavaクラスには話題に上げるほどのJavadocはありません。JavadocはAPI参照ツールとしては優れていますが、クラス・ライブラリーがどのように構成されているか、どのように使用すべきかを学ぶにはまったくおそまつな方法というよりほかありません。さらに、たとえ提供されたとしても、Javadocには、メソッドの動作についての最も基本的な情報しかない場合が多く、エラー処理、パラメーターや戻り値の定義域と範囲、スレッド・セーフティー、ロック動作、事前条件、事後条件、不変条件、副作用などの重要な特徴は無視されています。

Javadocから学ぶ

ほとんどのオープン・ソース・パッケージや社内開発コンポーネントなど、数多くのJava機能に関して、ほとんどのクラス・ライブラリーやコンポーネントにはJavadoc以外の意味のあるドキュメンテーションがないのが現状です。つまり、開発者はJavadocから機能の使用方法を学ぶことになるため、私たちはこの現実を踏まえてJavadocを構成する必要があります。私はよく冗談で、「今日のJavaプログラマーに必要な最も重要なスキルの1つは、GoogleとJavadocをうまく利用して、おそまつなドキュメンテーションしかないAPIをリバース・エンジニアリングする能力である」と言っていました。たとえこれが本当だとしても、まったく笑いごとではありません。

ほとんどのJavaパッケージには、その機能の他のオブジェクトを使用する前に作成しなければならない最初のオブジェクトである「ルート・オブジェクト」というものがあります。JNDIでは、このルート・オブジェクトは`Context`であり、JMSとJDBCでは`Connection`です。JDBCの基本オブジェクトが`Connection`であるということとそれを獲得する方法がわかれば、Javadocで利用可能なメソッドのリストを確認することで、次に`Statement`の作成と実行、さらに結果の`ResultSet`の繰り返しを行う方法もJavadocからわかるはずです。しかし、最初のステップが`Connection`の獲得であることは、どのようにしてわかったのでしょうか。Javadocは、パッケージ内でクラスを、またクラス内でメソッドをアルファベット順に並べています。残念ながらJavadocには、APIの検討を始める論理的な出発点を教えてくれるような魔法の「開始」サインはありません。

パッケージの説明

「開始」サインに最も近いのがパッケージの説明ですが、これは事実上めったに使用されません。package.htmlファイルとパッケージのソース・コードを一緒にしておくと、標準のdocletは、生成されたpackage-summary.htmlファイルにそのコンテンツとそのパッケージのクラスのリストを入れます。残念ながら、標準docletは私たちが見慣れたHTMLドキュメンテーションを作成するので、パッケージの説明は簡単には見つかりません。左上のペインのパッケージをクリックすると、左下のペインにメソッドのリストが現れますが、メイン・ペインにパッケージの要約は現れません。パッケージの要約を表示するには、左下のペインのパッケージ名をクリックしなければなりません。しかし、いずれにしろ、ほとんどのパッケージにはパッケージの説明はありません。

パッケージ・ドキュメンテーションは、パッケージがもつ機能、鍵となる抽象化、このパッケージのJavadocをどこから検討し始めればよいかについての概要として「開始」ドキュメンテーションを置くのにきわめて適切な場所です。

クラス・ドキュメンテーション

パッケージ・ドキュメンテーション以外に、クラス固有のドキュメンテーションも、ユーザーを新しい機能に導くのに重要な役割を果たします。クラス・ドキュメンテーションには当然、その特定のクラスの機能に関する説明が含まれるべきですが、さらに、そのクラスがパッケージの他のクラスにどのように関係しているかについても説明し、特にそのクラスに関係のあるファクトリー・クラスを特定すべきです。たとえば、JDBCの`Statement`クラスのドキュメンテーションは、`Statement`が`Connection`クラスの`createStatement()`メソッドによって獲得されることを説明します。そうすれば、新しいユーザーは、`Statement`ページを見つけた場合に、まず`Connection`を獲得する必要があることがわかります。パッケージが各クラスに対してこの規約を使用していれば、ユーザーに迅速にルート・オブジェクトが示され、ユーザーはスムーズに先に進むことができます。

Javadocは特定のクラスをドキュメント化するよう設計されているため、多くの場合、Javadocには、いくつかの関連クラスの使用を示すコード例を置く明確な場所はありません。しかし、特定のクラスまたはメソッドのドキュメンテーションに的を絞ってしまうと、パッケージがどのように組み合わさっているのかを説明することはできなくなります。パッケージ・ドキュメンテーション内でもルート・オブジェクトのクラス・ドキュメンテーション内でも、いくつかの基本的な使用法を説明するシンプルなコード例があると、多くのユーザーにとって非常に役に立つはずです。たとえば、`Connection`クラスのドキュメンテーションには、接続の獲得、準備済みステートメントの作成、ステートメントの実行、結果セットの繰り返しなどのシンプルな例

を含めることが考えられます。これはパッケージの他のクラスも説明しているため、技術的には `Connection` ページにすべきはない可能性があります。しかし、特に、現在のクラスが依存するクラスを参照する上述のテクニックを組み合わせれば、ユーザーは、クラスの構成に関係なく、簡単な実施例にすばやく進むことができます。

悪いドキュメンテーション == 悪いコード

ほとんどのJavaクラス・ライブラリーには、パッケージ・コンポーネントとして市販されている製品を除くと、Javadocが存在しないか不十分なものしかありません。ほとんどのパッケージでは、Javadocが利用可能な唯一のドキュメンテーションであるという現実を考えると、ほとんどのコードが作成者以外利用できないという状況に甘んじていることになります – 少なくとも考古学じみた大変な努力を払わない限りは。

ドキュメンテーションは現在コードの一部とされているので、たとえコードが良くてもドキュメンテーションが悪い場合にはそれを悪いコードと見なすという風潮が、ソフトウェア・エンジニアリング・コミュニティーにそろそろ生まれても良いように思います。というのは、そうしたコードは事実上再利用できないからです。ユニット・テストは最近まで評判が悪く、ごく最近になってやっと多くの技術者の間で人気が出始めましたが、このユニット・テストと同じようにAPIドキュメンテーションも、作成するソフトウェアの信頼性と再利用可能性を向上させるために、開発プロセスの不可欠な部分とならなければなりません。

コード・レビューとしてのJavadoc作成

適切なJavadocを作成する副次効果は、結果的にある種のコード・レビューを行い、クラスのアーキテクチャーとクラスの相互関係を検討する必要に迫られることです。パッケージ、クラス、メソッドのドキュメント化が難しいとしたら、おそらく一度に2つ以上のことを行おうとしているためでしょう。これはリエンジニアリングが必要な証拠だといえます。

ただコードが完成するのを待って、(時間があれば) ドキュメンテーションを作成するというのではなく、Javadocの作成を開発プロセスの初期に行い、コードの進展に伴って定期的にレビューするということが、ドキュメンテーションの自己レビューという側面からますます重要であることがわかります。コードが完成するのを待ってドキュメンテーションを作成するという戦略はきわめて一般的に行われている方法ではありますが、それではプロジェクトが終了するまでドキュメンテーションの作成を待つことになり、その時点ではスケジュールに無理が生じ、スタッフはストレスに悩まされることになります。その結果、ほとんどの場合「ドキュメンテーションの幻影」にすぎないリスト1のような価値のないドキュメンテーションが作られることになります。これは、そのクラスがどのように機能するかについてユーザーが本当に知る必要のあることを何も伝えていません。

リスト1. 典型的な価値のないJavadoc

```
/**
  * Represents a command history
  */
public class CommandHistory {
    /**
     * Get the command history for a given user
     */
    public static CommandHistory getCommandHistory(String user) {
    . . .
    }
}
```

良いドキュメンテーションとは

クラスの説明で関連クラスやファクトリー・クラスを参照したり、パッケージの概要やコード例を含めるといった上記の構成手法は、良いドキュメンテーションのためのきわめて適切な出発点です。これによって、新規のユーザーは、Javadocを使用して新しい機能を学ぶことができます。

しかし構成上の概要が占めるのは全体の半分だけです。残りの半分は、メソッドが何を行い何を行わないのか、どのような状況でメソッドが動作するのか、メソッドがどのようにエラー条件を処理するのかについての具体的な説明です。ほとんどのJavadocは、たとえ、メソッドが望ましいケースにおいてどのように動作するのかを適切に説明しているものでも、以下のものをはじめとして、必要なすべての情報を提供しているわけではありません。

- メソッドがエラー条件や不適切な入力をどのように処理するか
- エラー条件が呼び出し側にどのように戻されるか
- 例外のどの特定のサブクラスがスローされるか
- どの値が入力に有効であるか
- クラスの不変条件、メソッドの事前条件、またはメソッドの事後条件
- 副作用
- メソッド間に重要なリンケージがあるかどうか
- 複数のスレッドから同時にアクセスされるインスタンスをクラスがどのように処理するか

Javadoc規約では`@param`タグが与えられ、これによって、その名前や型だけでなく、パラメーターの意味をドキュメント化することができます。しかし、すべてのメソッドがパラメーターのどの値でもすんなりと受け入れるわけではありません。たとえば、型チェックのルールに抵触することなくオブジェクト・パラメーターを使用するメソッドにヌル値を渡すのは正当ですが、すべてのメソッドがヌル値をすんなりと受け入れるわけではありません。Javadocは、パラメーターの有効な範囲を明示的に説明する必要があります。パラメーターがヌル値以外であることを求める場合はそれを示し、特定の長さの文字列や1以上の整数など特定の範囲の値を求める場合はそれを示す必要があります。すべてのメソッドが妥当性に関して注意深く引数を確認するわけではありません。入力の許容範囲に関する妥当性チェックもドキュメンテーションもなければ、大きな災いを招くことになります。

戻りコード

Javadocによって戻り値の意味の説明が簡単になりますが、メソッド・パラメーターと同様、`@return`タグには戻される値の範囲の詳細な説明を含める必要があります。オブジェクト値の戻り型の場合、ヌル値を戻しますか。整数値の戻り型の場合、結果は、既

知の値のセットまたは負ではない値に制限されていますか。ファイルの終わりを示すために`java.io.InputStream.read()` から -1を戻すなど、特別な意味を持つ戻りコードがありますか。テーブル・エントリーが見つからない場合にヌル値を戻すなど、エラー条件を示す戻りコードが使用されていますか。

例外

標準のdocletはメソッドの`throws` 節を複製しますが、Javadocの`@throws` タグはそれよりはるかに具体的にする必要があります。たとえば、`NoSuchFileException` は`IOException` のサブクラスですが、`java.io` のほとんどのメソッドでは`IOException` をスローすることだけが宣言されています。しかし、メソッドが他の`IOException` とは別に`NoSuchFileException` をスローするという事実は呼び出し側に役立つものなので、Javadocに含める必要があります。さらに、特定の例外がスローされたときに取るべき修正アクションが呼び出し側にわかるように、さまざまな例外クラスがスローされる実際のエラー条件も指定する必要があります。メソッドが`@throws` タグによってスローするすべてのチェックされる例外とチェックされない例外をドキュメント化し、その例外がスローされる条件をドキュメント化する必要があります。

事前条件、事後条件、不変条件

当然ながら、オブジェクトの状態に対するメソッドの影響もドキュメント化しなければなりません。しかしそれだけではなく、メソッドの事前条件、事後条件、クラスの不変条件を説明することも必要です。事前条件とは、メソッドが呼び出される前のオブジェクトの状態への制約です。たとえば、`Iterator.next()` を呼び出す事前条件は、`hasMore()` が真であることです。事後条件は、メソッドの呼び出しの完了後のオブジェクトの状態への制約で、たとえば、`add()` の呼び出し後に`List` が空ではないといったことです。不変条件は、`Collection.size() == Collection.toArray().length()` などのように、常に真であることが保証されるオブジェクトの状態への制約です。

jContractなどの契約による設計（ design-by-contract ）ツールによって、特別なコメントを使用して事前条件、事後条件、クラスの不変条件を指定することができ、さらにこれらの制約を強制するための追加のコードをツールが生成します。このような制約を強制するツールを使用してもしなくても、こうした制約をドキュメント化することによって、ユーザーはクラスによって何を安全に行えるかがわかります。

副作用

場合によってメソッドには、そのオブジェクトの状態への変更以外にも、関連オブジェクトの状態やJVM、基本となるコンピューター・プラットフォームの変更などの副作用があります。たとえば、I/Oを実行するすべてのメソッドには副作用があります。クラスによって処理される要求数のカウントなど、害のない副作用もありますが、メソッドに渡されるオブジェクトの状態を変更したり、そのオブジェクトに対する参照のコピーを格納するなど、プログラムのパフォーマンスや正確さに重大な影響を与えるものもあります。関連オブジェクトの状態の変更やメソッド・パラメーターとして渡されるオブジェクトの参照の格納などの副作用はドキュメント化する必要があります。

メソッドのリンケージ

メソッドのリンケージとは、クラス内の2つのメソッドが互いに依存し、それぞれの振る舞いについて仮定することです。メソッドのリンケージが起こる良くある状況は、メソッドが内部で同じ

クラスの`toString`メソッドを使用し、`toString`が特定の方法でオブジェクトの状態をフォーマットすると仮定した場合です。この状況では、クラスがサブクラス化され`toString`メソッドがオーバーライドされた場合に問題が起こります。つまり、もう1つのほうのメソッドもオーバーライドされない限り、正常な機能が突然停止してしまいます。メソッドが他のメソッドの実装の振る舞いに依存している場合は、そうした依存関係をドキュメント化する必要があります。そうすれば、クラスがサブクラス化されていても、そのサブクラスが適切に機能するよう、両方のメソッドを一貫してオーバーライドすることができます。

スレッド・セーフティー

ドキュメント化すべき最も重要な振る舞いの1つ (ただし、ほとんどの場合ドキュメント化されない) は、スレッド・セーフティーです。そのクラスはスレッド・セーフですか。そうでない場合は、呼び出しを同期でラップすることによってクラスをスレッド・セーフにすることができますか。こうした同期は特定のモニターに関連していなければなりませんか。あるいは、使用されているモニターは一貫して適切なものですか。メソッドは、クラスの外から見えるオブジェクトのロックを獲得していますか。

実は、スレッド・セーフティーは二値属性ではありません。スレッド・セーフティーにはいくつかの識別可能な程度があります。スレッド・セーフティーのドキュメント化 (あるいはスレッド・セーフティーの程度の決定も) は、必ずしも簡単ではありません。しかし、それを行わないと重大な問題が発生します。同時アプリケーションでスレッド・セーフではないクラスを使用すると、デプロイメント (アプリケーションが公開されてロードされるとき) で初めて露呈する障害がたびたび発生します。また、すでにスレッド・セーフであるクラスで追加のロックをラップすると、パフォーマンスの低下やデッドロックの原因となります。

Josh Bloch氏は、著書「Effective Java Programming Language Guide」 (参考文献を参照) で、クラスのスレッド・セーフティーの程度をドキュメント化するのに役立つ分類法を提示しています。クラスは、スレッド・セーフティーの程度の高い順から、不変、スレッド・セーフ、条件付きスレッド・セーフ、スレッド互換、反スレッドのグループの1つに分類することができます。

この分類は、同時アクセスでのクラスの振る舞いに関する重要な情報を伝えるための優れたフレームワークです。この分類法そのものを使用するかどうかは重要ではありませんが、クラスで示すようにしたスレッド・セーフティーの程度を必ず特定すべきです。また、メソッドが、クラス自体のコードの外から見えるオブジェクトに対して、ロックを獲得する場合は、それが単なる「実装の詳細」であったとしても、グローバルなロック順序を決定しデッドロックを回避できるように、それをドキュメント化することもお勧めします。

結論

クラスの振る舞いのドキュメント化は、単に各メソッドの機能を1行で説明するだけではありません。有効なJavadocには以下の説明を盛り込む必要があります。

- クラスが互いにどのように関係しているか
- メソッドがオブジェクトの状態にどのような影響を与えるか
- メソッドが呼び出し側にエラー条件をどのように伝え、どのエラーの場合に信号を送るか
- クラスはマルチスレッド化アプリケーションでの使用にどのように対応しているか

- メソッドの引数の定義域と戻り値の範囲

ドキュメンテーションが不適切であったり、さらに悪いことに、ドキュメンテーションが全くない場合には、良いコードの利用や再利用が不可能になります。ドキュメンテーションにもう少し時間を費やすことで、みなさんのユーザー (そしておそらくみなさん自身) の無限ともいえるフラストレーションが解消されるでしょう。

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。