Javaの理論と実践: 疑似typedefアンチパターン

エクステンションは型定義ではない

Brian Goetz
Principal Consultant
Quiotix

2006年 2月 21日

Java™言語にジェネリックスが追加されたことによって、型システムが複雑になり、また多くの場合、変数やメソッド宣言の冗長性が増しています。型に短縮名を定義するための「typedef」機能が用意されていないため、開発者によっては「溺れる者がつかむ藁」としてエクステンションに頼ろうとし、望ましくない結果に終わっているようです。今回の『Javaの理論と実践』では、JavaのエキスパートであるBrian Goetzが、この「アンチパターン（antipattern）」の限界について説明します。

このシリーズの他の記事を見る

Java 5.0での新しいジェネリックス機能に関して頻繁に聞かれる不満として、コードが冗長になりすぎる、という問題があります。今までは1ラインの中に問題なく収まっていた変数宣言が収まらなくなり、しかもパラメーター化された変数型の宣言にまつわる繰り返し作業は面倒です。この問題は、自動完了をサポートする高品質なIDEを使えない場合には特に顕著になります。例えば、鍵がSocketsで値がFuture<String>であるMapを宣言しようとする場合、

```
Map socketOwner = new HashMap();
```

という今までの方法の方が、

```
Map<Socket, Future<String>> socketOwner
  = new HashMap<Socket, Future<String>>();
```

という新しい方法よりも簡潔です。もちろん、新しい方法では、より多くの型情報を埋め込むことができるためプログラミングの間違いが減り、プログラムが読みやすくなりますが、とにかく型やメソッド・シグニチャーを宣言する、という初期作業が発生します。宣言の中での型パラメーターの繰り返しと、その初期化は、特に不必要に思えます。つまりSocketとFuture<String>を2回タイプせねばならず、「DRY（don't repeat yourself: 繰り返すな）」の原則から外れることを余儀なくされています。

typedefを合成する・・・つもり

ジェネリックスが追加されたことによって、型システムが少し複雑になりました。Java 5.0以前は、「型」と「クラス」はほとんど同じ意味でしたが、パラメーター化型（parameterized type）、特にバインドされたワイルドカード型を持つパラメーター化型によって、サブタイプとサブクラスの概念が非常に異なったものになっています。ArrayList<?>型やArrayList<? extends Number>型、ArrayList<Integer>型などは（どれもArrayListという同じクラスによって実装されていますが）、特殊タイプ（distinct type）です。こうしたタイプは、階層構造を構成しています。つまりArrayList<?>はArrayList<? extends Number>のスーパータイプであり、ArrayList<? extends Number>はArrayList<Integer>のスーパータイプです。

オリジナルの単純な型システムでは、Cのtypedefのような機能は意味をなしませんでした。しかし、もっと複雑な型システムでは、typedefのような機能にも何らかの利点があるかも知れません。ともあれ、ことの良し悪しはともかくジェネリックスがJava言語に追加された時には、typedefは追加されなかったのです。

一部の人達が「溺れる者がつかむ藁（typedef）」として使っている1つの（不正な）イディオムが、単純なエクステンションです。つまり、ジェネリック型を拡張しながら何ら機能は追加しないクラス、（例えばリスト1のSocketUserMapタイプのようなもの）を作成する方法です。

リスト1.　疑似typedefアンチパターン（pseudo-typedef antipattern）・・・こんなことをしてはいけません

```
public class SocketUserMap extends HashMap<Socket<Future<String>> { }
SocketUserMap socketOwner = new SocketUserMap();
```

このトリック（私は疑似typedefアンチパターン（pseudo-typedef antipattern）と呼んでいます）によって、socketOwner定義を1行にする、という（怪しげな）目標は一応達成されます。しかし、それ以上の効果はなく、やがて再利用や維持管理の障害になります。（引数を持たないコンストラクター以外にはコンストラクターを持たないクラスでは、コンストラクターは継承されないため、派生クラスも各コンストラクターを宣言しなければなりません。）

疑似型（pseudotype）の問題

Cでは、typedefを使って新しい型を定義することは、（型宣言の定義というよりも）むしろマクロの定義に近いものです。同じ型を定義しているtypedef同士は自由に交換でき、また、生の型とも自由に交換できます。リスト2は、コールバック機能の定義の例です。ここではシグニチャーの中でtypedefが使われていますが、呼び出し側が同じ型のコールバックを提供しており、コンパイラーにとってもランタイムにとっても、まったく問題ありません。

リスト2. Cでのtypedefの例

```
// Define a type called "callback" that is a function pointer
typedef void (*Callback)(int);

void doSomething(Callback callback) { }

// This function conforms to the type defined by Callback
void callbackFunction(int arg) { }

// So a caller can pass the address of callbackFunction to doSomething
void useCallback() {
  doSomething(&callbackFunction);
}
```

エクステンションは型定義ではない

これと等価なプログラム（つまり疑似typedefアンチパターンを使うプログラム）をJava言語で書こうとすると、問題が起きます。リスト3の中のStringList型とUserList型は、どちらも共通のスーパークラスを拡張していますが、両者の型は同じではありません。これはつまり、lookupAllを呼ぼうとする全てのコードは、（List<String> あるいはUserListではなく）StringListを渡さなければならない、ということを意味しています。

リスト3. 疑似型によって、クライアントは疑似型を使うようにロックされる

```
class StringList extends ArrayList<String> { }
class UserList extends ArrayList<String> { }
...
class SomeClass {
    public void validateUsers(UserList users) { ... }
    public UserList lookupAll(StringList names) { ... }
}
```

この制約は、ちょっと見た印象よりも、ずっと深刻です。小さなプログラムであれば、ほとんど差がないかも知れません。しかしプログラムが大きくなると、常に疑似型を使うという要求によって問題が起きる可能性があります。もし変数がStringList型であった場合、List<String> はStringListのスーパータイプであり、従ってStringListではないため、その変数に通常のList<String>を割り当てることができません。String型の変数にはObjectを割り当てられないのと同様、StringList型の変数にList<String>を割り当てることはできないのです。（ただし、逆は可能です。つまりList<String>はStringListのスーパータイプなので、例えばList<String>型の変数にStringListを割り当てることはできます。）

メソッド・パラメーターにも同じことが言えます。メソッド・パラメーターがStringList型である場合には、そのメソッド・パラメーターに通常のList<String>を渡すことはできません。これはつまり、そのメソッドを使う場合には必ず疑似型を使うのでない限り、メソッド引数として疑似型がまったく使えないということを意味します。現実的な意味合いとしては、ライブラリーAPIでは疑似型がまったく使えない、ということです。通常、ライブラリーAPIというものは、ライブラリー・コードとして書かれたものではないコードが成長してできあがるものです。ですから、「このコードは私専用です。他の人は誰も使えません」という言い訳は適切ではありません（ただしこれは、そのコードが良質である場合の話です。使い物にならないようなコードであれば、そうした言い訳も極めて適切でしょう。）

疑似型は伝染する

こうした「伝染性」が、Cコードを再利用する場合に問題が起きやすい理由の一つです。ほとんどのCパッケージには、ユーティリティー・マクロや、int32、boolean、true、falseなどの型を定義するヘッダー・ファイルがあります。こうした共通アイテムに対して同じ定義を使っていない幾つかのパッケージを1つのアプリケーションの中で使おうとする場合には、全てのヘッダー・ファイルを含んだ空プログラムもコンパイルしないうちから、しばらくの間「ヘッダー・ファイルの地獄」に時間を取られる羽目になります。別々の人が書いた十幾つものパッケージを使うCアプリケーションを書く場合には、多かれ少なかれ、この痛みを伴います。一方Javaアプリケーションでは、十幾つ、あるいはそれ以上のパッケージを使用しても、そうした痛みは無いことが普通です。しかし、もしパッケージがそのAPIの中で疑似型を使っている場合には、問題が再発し、痛ましい記憶として残ることになるでしょう。

一例として、2つの異なるパッケージを考えてください。それぞれのパッケージは、リスト4に示すように疑似typedefアンチパターンを使ってStringListを定義し、また、StringListを操作するユーティリティー・メソッドを定義しています。両方のパッケージで同じ識別子を定義していることが、既に少しばかり問題が起こる元凶となっています。つまりクライアント・プログラムは、インポートするために一方の定義を選択し、もう一方に対して完全修飾名を使わなければならないのです。しかしもっと大きな問題は、こうしたパッケージのクライアントが、sortListとreverseListの両方に渡せるオブジェクトを作成できなくなってしまうことです（2つのStringList型は特殊タイプであり、お互いに互換性がないため）。クライアントは、どちらか一方のパッケージのみを使うように選択するか、あるいは別々な種類のStringListの間での変換のために大量の作業を行わなければなりません。パッケージを書いた人にとっては手軽な手段のつもりが大きな障害となり、最も制限された状況でしか、そのパッケージを使えなくなってしまうのです。

リスト4. 疑似型を使うことによって再利用が制限される

```
package a;

class StringList extends ArrayList<String> { }
class ListUtilities {
    public static void sortList(StringList list) { }
}

package b;

class StringList extends ArrayList<String> { }
class SomeOtherUtilityClass {
    public static void reverseList(StringList list) { }
}

...

class Client {
    public void someMethod() {
        StringList list = ...;
        // Can't do this
        ListUtilities.sortList(list);
        SomeOtherUtilityClass.reverseList(list);
    }
}
```

通常は、疑似型では具象的すぎる

疑似typedefアンチパターンには、他にも問題があります。このパターンでは、変数型やメソッド引数の定義にインターフェースを使うという利点を無視しがちなのです。List<String>を拡張するインターフェースとしてStringListを定義し、またArrayList<String>を拡張しStringListを実装する具象型StringArrayListを定義することはできますが、疑似typedefアンチパターン手法の主な目的は型名を単純化し、短縮することなので、ほとんどの（疑似typedefアンチパターンの）ユーザーは、そんなことまではしません。その結果、つまりListのような抽象型ではなくArrayListのような具象型を使う結果として、APIは使いにくくなり、柔軟性に欠けるものになります。

安全なトリック

ジェネリックな集合を宣言するために必要なタイプ打ちの量を減らすための安全策としては、『型推論（type inference）』を使用することです。プログラムの中に埋め込まれた型情報を使って型引数を割り当てることに関して、コンパイラーは非常に利口です。例えば次のようなユーティリティー・メソッドを定義することを考えてみてください。

```
public static <K,V> Map<K,V> newHashMap() {
    return new HashMap<K,V>();
}
```

これを使えば、型パラメーターを2度入力することを安全に回避することができます。

```
Map<Socket, Future<String>> socketOwner = Util.newHashMap();
```

コンパイラーは、ジェネリック・メソッドnewHashMap() が呼ばれるコンテキストからKとVの値を推測できるため、この手法でうまく行くのです。

まとめ

開発者が疑似typtedefアンチパターンを使う動機は非常に単純です。彼らは、もっと簡潔に型識別子を定義できる方法を求めているのです。特にジェネリックによって型識別子が一層冗長になったため、この要求は切実です。しかしこのイディオムを使うことによって、このイディオムを使うコードと、そのコードのクライアントとが密結合となり、再利用が妨げられてしまいます。ジェネリック型識別子の冗長さは不満かも知れませんが、それを避けようとして、このイディオムを使うのは誤りなのです。

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。