Javaコードの診断: 痛みを伴わないJava総称クラス

Java TigerバージョンとJSR-14プロトタイプ・コンパイラーにおける総称クラス

Eric Allen

2003年 2月 11日

今月のJavaコードの診断では、2003年後半にリリース予定のJavaバージョン1.5、通称Tigerに組み込まれる総称型（GenericTypes）と、それをサポートする特性について紹介します。総称型のさまざまな特徴を示すコード・サンプルを使って、EricAllen氏が、プリミティブ型に対する制限、制約付き総称クラス、多義的メソッドなど、Tigerの特性に重点を置きながら説明します(今後のコラムでは、Tigerでの実際の総称型の実現や、Tiger以降のバージョンにおける総称型の拡張の可能性について考察する予定です)。

J2SE 1.5 (コードネームTiger) は、2003年の終わりにリリースされる予定です。私は、このような新しく紹介されるテクノロジーについては、できる限り多くの情報を事前に収集しておくべきだと考えています。この記事は、バージョン1.5から使用可能となる新機能や変更内容に関して紹介するシリーズの第1回です。ここでは特に総称型について説明し、そのサポートを目的とするTigerの変更点と付加機能に重点を置いて話を進めていきます。

Tigerは、ソース言語の構文の大幅な拡張をはじめとして、多くの点でJavaプログラミング史上最大の飛躍となりそうです。Tigerで予定される最も注目すべき変更点は、JSR-14プロトタイプ・コンパイラー (今すぐ無料でダウンロードできます。詳しくは参考文献を参照してください) でも紹介しているように、総称型が追加されたことです。

まず始めに、総称型とは何か、その総称型をサポートするためにどのような機能が追加されているのかについて説明します。

キャストとエラー

総称型がなぜ便利なのかを理解するために、Java言語におけるバグの最も大きな原因、つまり、式を静的な型ではなく、より具体的な特定のデータ型に頻繁にダウンキャストする必要がある点に注目してみましょう (キャストによってトラブルが発生するいくつかのケースについては、参考文献の「二段たどりバグ・パターン」を参照してください)。

プログラム内のダウンキャストは、すべて`ClassCastException`の原因となる可能性があるため、できるだけ避けるべきです。しかしJava言語では、いくらデザインに優れたプログラムであっても、多くの場合、ダウンキャストを避けることは困難です。

Java言語でダウンキャストを実行する最も一般的な理由は、ほとんどのクラスが特定の用途で使用されるため、メソッドの呼び出しによって実行時に返される可能性のある引数の型が限定されることです。たとえば、`Hashtable` クラスで要素の追加や取得を行うとします。プログラムでキーとして使用する要素の型や、ハッシュテーブルに格納する値の型は、実行時に決まる任意のオブジェクトではありません。通常、すべてのキーは特定の型のインスタンスになります。同様に、格納される値も`Object` より具体的な特定の型を持ちます。

しかし、現在のJava言語の各バージョンでは、ハッシュテーブル内の特定のキーや要素に`Object` より具体的な型を宣言することはできません。ハッシュテーブルにオブジェクトを挿入したり、そこからオブジェクトを取得する処理の型シグネチャーからは、任意の型のオブジェクトが格納または削除されるということしかわかりません。たとえば、`put` および`get` のシグネチャーは次のようになります。

リスト1. 挿入/取得の型シグネチャーは任意の型のオブジェクトのみを示す

```
class Hashtable {
  Object put(Object key, Object value) {...}
  Object get(Object key) {...}
  ...
}
```

このように、`Hashtable` クラスのインスタンスから要素を取得する場合、`Hashtable` に`String` 型の要素以外に何も挿入されていないことがわかっていても、型システムでは取得する値が`Object` 型だということしかわかりません。取得した要素が同じコード・ブロックに追加されていたものであっても、その値に対して`String` 固有の操作を行うには、その値を事前に`String` へキャストしなければなりません。

リスト2. 取得した値をStringにキャストする

```
import java.util.Hashtable;

class Test {
  public static void main(String[] args) {
    Hashtable h = new Hashtable();
    h.put(new Integer(0), "value");
    String s = (String)h.get(new Integer(0));
    System.out.println(s);
  }
}
```

`main` メソッド本文の3行目でキャストが必要になっていることに注目してください。Javaの型システムは型付けが非常に弱いため、コードには上記のようなキャストが多くなりがちです。このようなキャストは、Javaのコードを冗長にするだけでなく、静的型チェックの意義を小さくしてしまいます(各キャストは、選択的に静的型チェックを無視するディレクティブであるため)。では、どうすれば型システムを拡張し、静的型チェックを回避せずに済むのでしょうか。

総称型の登場

上記の例のようなキャストをなくす無理のない方法は、Javaの型システムにいわゆる総称型を追加することです。総称型は、型の「関数」と考えることができます。総称型は、型変数によってパ

ラメーター化され、コンテキストに応じたさまざまな型引数でインスタンス化することができます。

たとえば、単にHashtable クラスを定義するかわりに、Key およびValue という型パラメーターを持つHashtable<Key, Value> という総称クラスを定義できます。Tigerでこのような総称クラスを定義する構文は通常のクラス定義と同じですが、クラス名に続けて型パラメーター宣言のシーケンスを角括弧で囲って指定する点が異なります。たとえば、次のように独自の総称クラスHashtableを定義することができます。

リスト3. 総称クラスHashtableの定義

```
class Hashtable<Key, Value> { ... }
```

これらの型パラメーターは、クラス定義の本文で従来の型に対して行うのと同じように参照できます。次のコードをご覧下さい。

リスト4. 従来の型と同様の方法による型パラメーターの参照

```
class Hashtable<Key, Value> {
  ...
  Value put(Key k, Value v) {...}
  Value get(Key k) {...}
}
```

型パラメーターのスコープは、静的メンバーを除き、対応するクラス定義の本文です。(次回の記事では、Tigerの実装でなぜ静的メンバーにこのような制限が必要であったのかについて説明します。どうぞお楽しみに。)

Hashtable の新しいインスタンスを作成するには、型引数を渡してKey とValue の型を指定する必要があります。どのような型にするかは、Hashtable の使用目的によって異なります。前の例では、Integer をString にマッピングするだけのHashtable のインスタンスを作成することが目的でした。以下の新しいHashtable クラスを使用すればそれが可能です。

リスト5. IntegerをStringにマッピングするインスタンスの作成

```
import java.util.Hashtable;

class Test {
  public static void main(String[] args) {
    Hashtable<Integer, String> h = new Hashtable<Integer, String>();
    h.put(new Integer(0), "value");
    ...
  }
}
```

これでキャストは必要なくなりました。総称クラスHashtable のインスタンス化に使用した構文に注目してください。総称クラスの型パラメーターを角括弧で囲むのと同様に、総称型を使用するときにも引数を角括弧で囲みます。

リスト6. 不要なキャストの排除

```
...
String s = h.get("key");
System.out.println(s);
```

総称型を使用できるように、`Hashtable` や`List` のような標準的ユーティリティー・クラスをすべて再定義しなければならないとすれば、当然、プログラマーの作業量は膨大なものになります。幸いTigerでは、すべてのJavaコレクション・クラスについて総称クラス・バージョンが提供されるため、ユーザーが自分で再定義を行う必要はありません。さらに、これらのクラスは、従来のコードと新しい総称クラスのコードの両方でシームレスに機能します (なぜこれが可能なのかについては来月説明します)。

Tigerの「プリミティブ」な限界

Tigerの型変数の1つの制限は、型変数を参照型を使ってインスタンス化する必要があるために、プリミティブ型を使用できないという点です。このため、上記の例では、代わりに`int` を`String` にマッピングする`Hashtable` を作成しようとしても、これは不可能です。

残念なことに、プリミティブ型を総称型への引数として使用する場合には、必ずプリミティブ型をラップする必要があります。しかし、もともとすべてのキーは`Object` 型でなければならず、`int`をキーとして`Hashtable` に渡すことはできなかったわけですから、現在の状況と比べて悪いことだとはいえません。

私たちが本当に期待するものは、C# で実現されているような、プリミティブ型の自動的なボックス化とアンボックス化でしょう (もちろん、それ以上であれば言うことはありません)。残念ですが、Tigerではプリミティブの自動的なボックス化を組み込む予定はありません (Java 1.6には大いに期待したいものです)。

制約付き総称クラス

総称クラスのインスタンス化で使用できる型を制限したい場合があります。上記の例でいえば、`Hashtable` クラスの型パラメーターは任意の型引数でインスタンス化できますが、別のクラスでは、使用可能な型引数を、制約 (bound) として指定した型パラメータのサブタイプに限定したい場合があります。

たとえば、従来の`Pane` クラスへの参照を保持し、それにスクロール機能を付加する総称クラス`ScrollPane` を定義するとします。この総称クラスに含まれる`Pane` の実行時の型は、多くの場合`Pane` クラスのサブタイプになりますが、静的な型は単なる`Pane` です。

getterでこの`Pane` を取り出す際、その戻り型をできるだけ具体的に指定したい場合があります。その場合、`ScrollPane` に型パラメーター`MyPane` を追加して、`Pane` の任意のサブクラスでインスタンス化できるようにします。`MyPane` の宣言に "`extends##`" の形式で注釈を付けることにより、`MyPane` に制約を設定することができます。

リスト7. extends節を使用してMyPaneの宣言に注釈を付ける

```
class ScrollPane<MyPane extends Pane> { ... }
```

もちろん、明示的に制約を付けなくても、ただ型パラメーターを適切な型でインスタンス化するよう注意すればよいのです。

なぜわざわざ型パラメーターに制約を付ける必要があるのでしょうか。理由はいくつかあります。第1に、制約を設定することで静的型チェックが実行されます。これにより、その総称型がインスタンス化されるたびに、設定された制約が必ず厳守されるようになります。

次に、型パラメーターをインスタンス化するたびに、そのインスタンスが制約で指定されたクラスのサブクラスとなることがわかっているため、制約内の型パラメーターのインスタンスであれば、そのインスタンスの任意のメソッドを安心して呼び出すことができます。制約のないパラメーターはデフォルトで`Object` と見なされ、そのインスタンスでは`Object` クラス内に存在しないメソッドを呼び出すことができなくなります。

多義的メソッド

型パラメーターによるクラスのパラメーター化に加えて、同様の型パラメーターを使ったメソッドのパラメーター化も便利です。総称的なJavaプログラミング表現では、型によってパラメーター化されるメソッドのことを多義的 (polymorphic) メソッドと呼びます。

多義的メソッドは、引数の型と戻り値の型の依存関係が本質的に総称的で、その総称的な性質がクラス・レベルの型情報に依存せず、メソッド呼び出しのたびに変化するような状況で操作を実行する場合に便利です。

たとえば、`List` クラスに`factory` メソッドを追加するとします。この静的メソッドは、Listのただ1つの要素 (他の要素が追加されるまで) となる1つの引数を取ります。`List` には総称型の要素を含めたいので、静的な`factory` メソッドが型変数`T` を引数として取り、`List<T>` のインスタンスを返すようにします。

ただし、この型変数`T` は、メソッド呼び出しのたびに変化するため、メソッドのレベルで宣言する必要があります (これは次の記事でも説明しますが、Tigerのデザインでは、静的メンバーはクラス・レベルの型パラメーターのスコープ外になります)。Tigerでは、型パラメーターをメソッド宣言の前に追加することによって、個々のメソッド・レベルで型パラメーターを宣言できます。たとえば、この例の`factory` メソッド、`make` は次のように記述できます。

リスト8. メソッド宣言の前に型パラメーターを追加する

```
class Utilities {
   <T extends Object> public static List<T> make(T first) {
     return new List<T>(first);
   }
}
```

Tigerでは、多義的メソッドによって柔軟性が向上していますが、それ以外にもメリットがあります。Tigerでは、型インターフェース・メカニズムの使用により、多義的メソッドの型が引数の型に基づいて自動的に推測されます。そのため、メソッド呼び出しの冗長さや複雑さが大幅に軽減します。たとえば、`make` メソッドを呼び出して、`new Integer(0)` を含む`List<Integer>` の新しいインスタンスを作成するには、次のような簡単なコードを記述するだけで済みます。

リスト9. makeで新しいインスタンスを作成する

```
                        Utilities.make(Integer(0))
```

これにより、型パラメーターのインスタンス化は、メソッドの引数から自動的に推測されます。

総称のまとめ

これまで見てきたように、Java言語に総称型が追加されることによって、これまでよりも有効に静的型システムを活用できるようになるはずです。総称型の使用方法を学ぶのはとても簡単ですが、避けなければならない落とし穴もあります。今後の記事では、Tigerで実際に提供される総称型を有効活用する方法や、それに関連するいくつかの落とし穴について考察します。また、現在設計段階にある今後のJavaプラットフォームのバージョンに期待できるJava総称型の機能拡張についても検証する予定です。

関連トピック

- IntelliJのIDEA development environment (J2EEのWebアプリケーション高速開発機能、強力なコード検査ツール、およびサード・パーティーのプラグインをサポートするOpen APIを含む) を考察し、さらに「アイデア」を得てください。
- 効果的なリファクタリングの詳細については、Martin Fowler氏のWebサイトを参照してください。
- developerWorksの 『Javaコードの診断』 コラムの総括には、バグ・パターン、テスト可能性、デザイン計画などに関するEric Allen氏のその他のコラムが掲載されています。
  - バグ・パターン： Javaプログラムで頻発しがちなバグの分析と修正
  - 「宙ぶらりん複合型」バグ・パターン： ヌル・ポインター例外の最もよくある原因を鎮圧する
  - 「ヌル・フラグ」バグ・パターン： 例外状況を表すフラグとしてヌル・ポインターを使うことを避ける
  - Javaコードのパフォーマンスを向上させる： 末尾再帰変換はアプリケーションの速度を向上させる可能性はあるが、すべてのJVMで可能な操作ではない
  - 虚偽の実装というバグ・パターン: 第2回： 表明とユニット・テスト - バグを除去するための実行可能なドキュメンテーション -
  - 「みなし子スレッド」バグ・パターン： マスター・スレッドが自滅し、その他のスレッドが生き残っていると、どうなるか?
  - 拡張可能アプリケーションの設計 第1回： ブラック・ボックス、オープン・ボックス、またはガラス・ボックス: どんな場合にどれがふさわしいか?
  - 深さ優先visitorと、破綻したディスパッチ： このVisitorパターンの変形を使えば、コードをより簡潔にできます
  - replによる対話式評価： ソフトウェアを効率的かつ対話式に診断するためのテクニックとツール
  - 「付け足し初期化コード」バグ・パターン： 引数の足りないコンストラクターを避ければ、このバグを撃退できる
  - プラットフォーム依存を引き起こす"犯人"： プラットフォーム依存のバグ・パターンにスポットライトを当てる
  - 単体テストと自動コード分析の連携： テストを工夫して、ツールによるコード分析を手助けする
  - パッケージの依存関係の分離： コンポーネント・ベース・プログラミングにより、テスト可能なコードをより柔軟に接続
  - 保守が容易なコードの設計： 不必要な変異やアクセスを回避することにより、堅固で保守が容易なコードを作成する
- Javaコードへの総称型の追加については、Javaコミュニティー・プロセスの提案JSR-14 を参照してください。
- developerWorks Java technologyゾーン に含まれている、Javaテクノロジーに関する他の記事やチュートリアルをお読みください。