Javaの理論と実践: Generics のワイルドカードを使いこなす、第 1 回

ワイルドカードのキャプチャーを理解する

Brian Goetz
Senior Staff Engineer
Sun Microsystems

2008年 5月 06日

Java™ 言語における Generics で最も複雑な部分の 1 つはワイルドカードであり、なかでもワイルドカードのキャプチャーの扱い、そしてワイルドカードのキャプチャーに関するわかりにくいエラー・メッセージは特に厄介です。「Java の理論と実践」シリーズの今回は、ベテランの Java 開発者である Brian Goetz が、javac から出力される奇妙なエラー・メッセージのいくつかを解読し、さらに Generics の使い方を単純化するための手法と回避策をいくつか提供します。

このシリーズの他の記事を見る

Generics が JDK 5 で Java 言語に追加されて以来、Generics に関する議論は続いています。一部の人は、Generics によって型システムの到達範囲を広げることができ、従ってコンパイラーがタイプセーフを検証できるため、プログラミングを単純化できると言いますが、その一方で、Generics を使うことの価値以上にプログラミングが複雑になると言う人もいます。私達は誰しも、Generics に関して頭を悩ますような経験を何度かしていますが、Generics の中でも特に理解しにくい部分はワイルドカードです。

ワイルドカードの基礎

Generics は、未知の型に関してクラスやメソッドの振る舞いに対する型の制約を表現するための方法です。型の制約とは、例えば「このメソッドの x パラメーターと y パラメーターの型が何であれ、`x` と `y` は同じ型でなければならない」とか、「この両方のメソッドに対して同じ型のパラメーターを提供する必要がある」、あるいは「`foo()` の戻り値は `bar()` のパラメーターと同じ型である」などです。

ワイルドカード (つまり型パラメーターがあるはずの変な場所にある疑問符) は、未知の型に関する型の制約を表現するための方法です。ワイルドカードは Generics の元々の設計 (GJ (Generic Java) プロジェクトが元になっています) にあったわけではなく、JSR 14 の策定が始められてから最終リリースまでの 5 年間にわたる設計プロセスの間に追加されたものです。

ワイルドカードは型システムにおいて重要な役割を果たします。つまりワイルドカードを利用することで、ジェネリック・クラスによって規定される型のファミリーに対して効果的に型を制約することができます。ジェネリック・クラスである `ArrayList` の場合、`ArrayList<?>` という型は、任意の (参照) 型 `T` に対する `ArrayList<T>`;のスーパータイプです (`ArrayList` 型そのものとルート型の`Object` もスーパータイプですが、これらのスーパータイプは型推論を行う上ではあまり使い道がありません)。

ワイルドカード型の `List<?>` は List 型そのものや具象型の `List<Object>` のどちらとも異なります。変数 `x` が `List<?>` 型を持つということは、`x` を `List<T>` 型とするような何らかの型 `T` が存在するということであり、また (`x` の要素が具体的にどんな型を持つかはわからないものの) `x` は同じ型であるということです。`x` の内容は任意であるという意味ではなく、その内容に対する型の制約が何かはわからないものの、制約が「ある」ことがわかっているという意味なのです。一方、`List` 型そのものは異なる型を取ることができ、`List` の要素に対しては何も制約を課すことができません。また具象型の `List<Object>` の意味は、ここには任意のオブジェクトを含むことができる、と明確にわかっているということです。(もちろん、Generics の型システムには「リストの内容」という概念はありませんが、`List` のようなコレクション型の観点で見ると Generics を非常に理解しやすくなります)。

型システムにおけるワイルドカードの有用性の一部は、ジェネリック型は共変 (covariant) ではないという事実に由来しています。配列は共変です。なぜなら、`Integer` は `Number` のサブタイプであり、また配列型 `Integer[]` は `Number[]` のサブタイプであるため、`Number[]` の値が要求される場合には必ず `Integer[]` の値が提供されるからです。一方、Generics は共変ではありません。`List<Integer> # List<Number>` のサブタイプではなく、`List<Number>` が要求される場所で `List<Integer>` を提供しようとすると型エラーになります。これは偶然のエラーではなく (また、誰もが起きて当然と思うようなエラーでもありません)、Generics と配列との振る舞いの違いによって大きな混乱が生じるのです。

ワイルドカードを使ってできること

リスト 1 は、`put` 操作と `get` 操作をサポートする単純なコンテナー型、`Box` を示しています。`Box` は `box` の内容の型を表す型パラメーター `T` によってパラメーター化されています。また `Box<String>` は String 型の要素しか含むことはできません。

リスト 1. 単純なジェネリック Box 型

```
public interface Box<T> {
    public T get();
    public void put(T element);
}
```

ワイルドカードの 1 つの利点は、ワイルドカードを利用することによって、変数の型の制約を正確に知らなくてもジェネリック型の変数を操作するコードを作成できることです。例えば `Box<?>` 型の変数があるとします (例えばリスト 2 の `unbox()` メソッドの `box` パラメーターなど)。`unbox()` は渡された `box` に対して何ができるのでしょう。

リスト 2. ワイルドカードのパラメーターを持つ unbox メソッド

```
public void unbox(Box<?> box) {
    System.out.println(box.get());
}
```

unbox() は非常にたくさんのことができることがわかります。get() メソッドを呼び出すことができ、また Object (例えば hashCode() など) から継承した任意のメソッドを呼び出すことができます。unbox() にできないことは、put() メソッドを呼び出すことだけです。なぜなら、この Box インスタンスの型パラメーター T がわからないと put 操作が安全かどうかを検証できないからです。box は Box<?> であり、Box そのものではないため、コンパイラーは box の型パラメーターの役割を果たす T があることはわかりますが、その T が何であるかはわからないため、put() を呼び出すことは許可しません。なぜなら、put() を呼び出しても Box のタイプセーフに関する制約に違反しないかどうかコンパイラーは検証できないからです。(実際には put() を呼び出せる特別なケースが 1 つあり、それは null リテラルを渡す場合です。型 T が何を表すのかはわかりませんが、null リテラルはすべての参照型に対して有効な値であることはわかっています。)

unbox() は box.get() の戻り型について何を知っているのでしょう。unbox() は box.get() の戻り型が、ある未知の T という型であることを知っています。そのため unbox() 型が下せる最善の結論は、get() の戻り型は未知の型 T のイレイジャー (erasure) であり、それは制約なしのワイルドカードの場合には Object です。つまりリスト 2 の式 box.get() は Object 型を持ちます。

ワイルドカードのキャプチャー

リスト 3 は、動作するはずに見えながら実際には動作しないコードを示しています。このコードはジェネリック型の Box を取り上げ、Box の値を抽出し、そしてその値を同じ Box の中に戻そうとしています。

リスト 3. Box から出してしまうと、Box に戻すことはできません

```
public void rebox(Box<?> box) {
    box.put(box.get());
}

Rebox.java:8: put(capture#337 of ?) in Box<capture#337 of ?> cannot be applied
   to (java.lang.Object)
    box.put(box.get());
       ^
1 error
```

抽出される値は元に戻すための型としては正しいので、このコードは動作しそうに思えます。しかし実際には動作せず、代わりにコンパイラーが、「capture#337 of ?」は Object と互換性がありません、という (非常にわかりにくい) エラー・メッセージを生成します。

いったい「capture#337 of ?」は何を意味するのでしょう。コンパイラーは、型の中にワイルドカードを持つ変数 (例えば rebox() の box パラメーターなど) に遭遇すると、box # Box<T> とするような T があったに違いない、と理解します。コンパイラーには T がどの型を表すのかはわかりませんが、T の型であるはずの型を表すためのプレースホルダーを作成することはできます。このプレースホルダーは、その特定のワイルドカードのキャプチャーと呼ばれます。この例の場合では、コンパイラーは box の型の中にあるワイルドカードに対して「capture#337 of ?」という名前を割り当てており、各変数宣言の中でワイルドカードがあるごとに、それぞれ異なるキャプチャーが与えられます。そのため、例えばジェネリック宣言 foo(Pair<?,?> x, Pair<?,?> y) の場合、これらの未知の型パラメーターそれぞれの間には何も関係がないため、コンパイラーは 4 つのワイルドカードそれぞれのキャプチャーに対して別々の名前を割り当てます。

このエラー・メッセージからわかることは、`put()` の実際のパラメーターの型が正式なパラメーターの型と互換性があることを検証できないため、put() を呼び出すことはできないということです。なぜなら `put()` の正式なパラメーターの型は未知だからです。この場合の `?` は基本的に「? は Object を継承する」という意味なので、コンパイラーは `box.get()` の型が `Object` であって「capture#337 of ?」ではないこと、そして「capture#337 of ?」というプレースホルダーで特定される型の値として `Object` が受け入れ可能なことを静的に検証することはできない、と既に結論づけています。

キャプチャー・ヘルパー

コンパイラーが有用な情報を捨てているように思えるかもしれませんが、コンパイラーにこの情報を再活用させ、ここで必要としているような目的に役立てるための手法があります。その手法というのは、未知のワイルドカード型に名前を付けることです。リスト 4 はこの手法の一例として、`rebox()` の実装と、ジェネリックのヘルパー・メソッドを示しています。

リスト 4. 「キャプチャー・ヘルパー」イディオム

```
public void rebox(Box<?> box) {
    reboxHelper(box);
}

private<V> void reboxHelper(Box<V> box) {
    box.put(box.get());
}
```

ヘルパー・メソッド `reboxHelper()` はジェネリック・メソッドです。ジェネリック・メソッドによって、さらに型パラメーターが導入されます (戻り型の前の不等号括弧の中に置かれます)。これらの型パラメーターは通常、メソッドのパラメーターや戻り型の間に型制約を形成するために使われます。しかし `reboxHelper()` の場合には、ジェネリック・メソッドは型パラメーターを使って型制約を指定することはせず、コンパイラーが (型推論を使って) `box` の型の型パラメーターに名前を付けるようにするのです。

キャプチャー・ヘルパーによる手法によって、コンパイラーがワイルドカードを処理する際の制約を回避することができます。`rebox()` が `reboxHelper()` を呼び出す場合、`rebox()` はその呼び出しが安全であることを知っています。`rebox()` 自身の `box` パラメーターは未知の `T` に対する `Box<T>` でなければならないからです。型パラメーター `V` はメソッド・シグニチャーの中に導入されており、他のどの型パラメーターとも結合されていないため、`V` によって任意の未知の型も表すことができます。つまり未知の `T` に対する `Box<T>` は未知の `V` に対する `Box<V>` としても同じです。(これは、束縛変数のリネームを許容する、ラムダ計算でのアルファ変換の原則と似ています)。そうすると `reboxHelper()` の中の式 `box.get()` はもはや `Object` 型を持つのではなく、`V` 型を持つことになり、`V` を `Box<V>.put()` に渡せることになります。

`rebox()` を (`reboxHelper()` のように) 最初からジェネリック・メソッドとして宣言することもできたのですが、それは API の設計スタイルとしては不適切です。ここで重視される設計原則は、「名前で参照しないものに対して名前を付けてはならない」というものです。ジェネリック・メソッドの場合について言えば、ある型パラメーターがメソッド・シグニチャーの中で 1 度しか現れないのであれば、その型パラメーターはおそらく、名前付きの型パラメーターではなくワイルドカードであるべきなのです。一般的に、ワイルドカードを持つ API の方がジェネリック・メソッドを持つ API よりも単純であり、また複雑なメソッド宣言の中に型の名前が散乱している

と、その宣言が読みにくくなります。必要であれば private のキャプチャー・ヘルパーを使っていつでも名前を復活させることができるため、この手法を利用すれば有用な情報を捨てることなく API を簡潔に保つことができます。

型推論

キャプチャー・ヘルパーによる手法は、いくつかのもの、つまり型推論とキャプチャー変換に依存しています。Java コンパイラーが型推論を行う場所はあまり多くありませんが、推論を行う 1 つの場所としてジェネリック・メソッドの型パラメーターを推論する場合を挙げることができます。(他の言語はもっと大きく型推論に依存しており、将来は Java 言語に型推論機能がさらに追加されるかもしれません。) 必要であれば型パラメーターの値を指定することはできますが、それは型に名前を付けられる場合に限られ、またキャプチャー型に名前を付けることはできません。そのため、この手法が使えるのはコンパイラーが型を推論してくれる場合のみです。キャプチャー変換は、キャプチャーされたワイルドカードに対するプレースホルダーとしての型名をコンパイラーが作成できるようにするので、型推論によって型を推論できるようになります。

コンパイラーはジェネリック・メソッドへの呼び出しを解決する際に、型パラメーターに対して最適と思われる型を試み、そして推論します。例えば次のようなジェネリック・メソッドを考えてみましょう。

```
public static<T> T identity(T arg) { return arg };
```

そして次のような呼び出しを行う場合、

```
Integer i = 3;
System.out.println(identity(i));
```

コンパイラーは、`T` が `Integer` か `Number` かシリアライズ可能、または `Object` であると推論できますが、いくつかの制約に対して最適な型は `Integer` なので、`Integer` を選択します。

ジェネリック・インスタンスを作成する際には、型推論を使うことで冗長性をいくらか削減することができます。例えば `Box` クラスを使って `Box<String>` を作成するためには型パラメーター `String` を 2 回指定する必要があります。

```
Box<String> box = new BoxImpl<String>();
```

ここに見られるような、DRY (Don't Repeat Yourself) の原則に対する違反は、たとえ IDE が多少のことをしてくれるとしても面倒です。しかし、リスト 5 のように実装クラス `BoxImpl` がジェネリックなファクトリー・メソッドを提供する場合には (これはいずれにせよ良い考えです)、クライアント・コードではこの冗長性を削減することができます。

リスト 5. 型パラメーターを指定する際の冗長性を避けられるジェネリックなファクトリー・メソッド

```
public class BoxImpl<T> implements Box<T> {

    public static<V> Box<V> make() {
        return new BoxImpl<V>();
    }

    ...
}
```

`BoxImpl.make()` ファクトリーを使って `Box` をインスタンス化すると、型パラメーターの指定を 1 度ですますことができます。

```
Box<String> myBox = BoxImpl.make();
```

ジェネリックな `make()` メソッドは、ある `V` 型の `Box<V>` を返し、この戻り値は `Box<String>` を必要とするコンテキストで使われます。コンパイラーは型制約を満足するものの中で `V` に最適な型は `String` であると判断し、そのためここでは `V` が `String` であると推論しています。ただし `V` の値を手動で指定することは相変わらず可能です (下記)。

```
Box<String> myBox = BoxImpl.<String>make();
```

ここに示したファクトリー・メソッドによる手法には、キー入力を少し節約できること以外に、コンストラクターに対する利点が他にもあります。コンストラクターの内容を記述するような名前をコンストラクターに付けることができ、コンストラクターは指定された名前の戻り型のサブタイプを返すことができ、またコンストラクターは呼び出しごとに必ずしも新しいインスタンスを作成する必要がないため、不変インスタンスを共有することができます。(「参考文献」に挙げた『Effective Java プログラミング言語ガイド』の項目 1 には静的ファクトリーの利点が詳細に説明されています。)

まとめ

ワイルドカードは確かに理解しにくいものです。Java コンパイラーから出力される非常にわかりにくいエラー・メッセージのいくつかはワイルドカードに関係しており、Java 言語仕様の最も複雑なセクションのいくつかはワイルドカードに関係しています。しかし適切に使用すれば、ワイルドカードは非常に強力です。ここで説明した 2 つの手法、つまりキャプチャー・ヘルパーによる手法とジェネリック・ファクトリーによる手法 (どちらもジェネリック・メソッドと型推論を利用しています) を適切に使用すれば、ワイルドカードの複雑さの大部分を隠すことができます。

著者について

Brian Goetz

Brian Goetz はこれまで 20 年間、プロのソフトウェア開発者として活躍してきました。現在は Sun Microsystems のシニア・スタッフ・エンジニアであり、複数の JCP Expert Group の一員でもあります。2006年5月に Addison-Wesley から彼の著書『Java 並行処理プログラミング — その「基盤」と「最新 API」を究める —』が出版されています。人気の業界紙に掲載された、Brian のこれまでの記事、そして今後の記事を参照してください。