Java の理論と実践: Generics のワイルドカードを使いこなす、第 2 回

get と put の原則

Brian Goetz
Senior Staff Engineer
Sun Microsystems

2008年 7月 01日

Java™ 言語での Generics のワイルドカードは非常にわかりにくく、最もよくある間違いは境界ワイルドカードが必要なときに、その 2 つの形式 (「? super T」 と「? extends T」) のうちの 1 つを使いそこねる、というものです。皆さんはこの間違いをしたことがあるでしょうか？もしあったとしても、それを恥じることはありません。エキスパートですら間違えることはあるのです。今月は、Brian Goetz がその間違いを避けるための方法を説明します。

このシリーズの他の記事を見る

Java 言語では、配列は共変 (covariant) です (なぜなら`Integer` は `Number` でもあり、`Integer` の配列も `Number` の配列だからです)。しかし、Generics は共変ではありません (`List<Integer>` は `List<Number>` ではありません)。`List<Integer>` と `List<Number>` のどちらを選択するのが「正し」く、どちらが「誤り」なのか議論することはできます (もちろん、どちらにも長所と短所があります)。しかしわずかに異なるセマンティクスを持つ派生型を作成するために似たような 2 つの仕組みがあることが、混乱と誤りの根本的な原因であることは間違いありません。

境界ワイルドカード (例の奇妙な「`? extends T`」というジェネリック型の指定子) は、共変でない場合を扱うために用意されたツールの 1 つです。境界ワイルドカードによって、クラスはメソッドの引数や戻り値が共変の場合 (あるいは逆に反変 (contravariant) の場合) の宣言をすることができます。境界ワイルドカードをいつ使うのかという問題は Generics の複雑な側面の 1 つですが、その問題の重荷は、ほとんどの場合ライブラリーを使う人ではなくライブラリーを作成する人の肩にかかっています。境界ワイルドカードに関して最もよくある間違いは、境界ワイルドカードを使うこと自体を忘れてしまい、クラスの有用性を制限してしまったり、あるいは既存のクラスを再利用するという困難をユーザーに強制したり、といった事態に陥ることです。

境界ワイルドカードの必要性

まず、`Box` という単純なジェネリック・クラス (値のコンテナー) から始めましょう。このクラスは既知の型の値を保持します。

© Copyright IBM Corporation 2008

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

Generics は共変ではないため、たとえ `Integer` は `Number` であっても `Box<Integer>` は `Box<Number>` ではありません。しかし `Box` のような単純なジェネリック・クラスの場合には、これは問題にはならず、実際私達はそれに気付かないかもしれません。なぜなら `Box<T>` のインターフェースは完全に T 型の変数の観点で規定されており、T に対するジェネリック型の観点で規定されているわけではないからです。型変数の観点で直接扱うことによって、必要なポリモーフィズムを苦労せずに手に入れることができます。リスト 1 は、この種のポリモーフィズムの 2 つの例を示しています (ここでは `Box<Integer>` の内容を `Number` として取得し、`Integer` を `Box<Number>` の中に入れています)。

リスト 1. ジェネリック・クラスに元々備わっているポリモーフィズムを活用する

```
Box<Integer> iBox = new BoxImpl<Integer>(3);
Number num = iBox.get();

Box<Number> nBox = new BoxImpl<Number>(3.2);
Integer i = 3;
nBox.put(i);
```

この単純な `Box` クラスでの経験から、共変は必要ないと納得させられるかもしれません。なぜなら、ポリモーフィズムを期待できる場所では、そのデータは既にコンパイラーが適切なサブタイプ・ルールを適用できる形式になっているからです。

しかし、T 型の変数だけではなく T に対するジェネリック型も API によって処理できるようにしたい場合には、話が複雑になります。例えば、`Box` に新しいメソッドを追加し、この `Box` に別の `Box` から取得した内容を入れられるようにしたいとしましょう (リスト 2)。

リスト 2. 拡張された Box インターフェース (見た目ほど柔軟ではありません)

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

この拡張された `Box` の問題は、`Box` の中に入れられる内容は型パラメーターが受信側の box とまったく同じもののみであるという点です。そのため、例えばリスト 3 のコードはコンパイルすることができません。

リスト 3. Generics は共変ではありません

```
Box<Number> nBox = new BoxImpl<Number>();
Box<Integer> iBox = new BoxImpl<Integer>();

nBox.put(iBox);     // ERROR
```

コンパイラーは、`Box<Number>` に対する `put(Box<Integer>)` メソッドが見つからない、というエラー・メッセージを表示します。このエラーは、Generics は共変ではないことを考えれば納得で

きます。つまり、たとえ Integer は Number であっても Box<Integer> は Box<Number> ではないのですが、そうするとこの Box クラスは私達が望んだほど「ジェネリック」ではなくなってしまいます。ジェネリックなコードをより有効なものにするためには、ジェネリック型のパラメーターの型を厳密に指定する代わりに、上限境界または下限境界を指定することができます。そうするためには、「? extends T」または「? super T」の形を取る、境界ワイルドカードを使います。(境界ワイルドカードは型パラメーターとしてのみ使うことができ、型そのものとして使うことはできません。型そのものとして使うためには、境界名型変数が必要です。) リスト 4 では put() のシグニチャーを変更し、上限境界ワイルドカード (Box<? extends T>) を使っていますが、これは Box の型パラメーターが T または T の任意のサブクラスだということです。

リスト 4. リスト 3 の Box クラスを改善したもの (共変に対応しています)

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

今度はリスト 3 のコードをコンパイルすることができ、想定どおりの動作をさせることができます。これは、T または T の任意のサブタイプを型パラメーターとして持つ Box を put() へのパラメーターに指定することができるように指示したためです。Integer は Number のサブタイプなので、コンパイラーは put(Box<Integer>) というメソッドの参照を解決することができます (これが可能なのは Box<Integer> は境界ワイルドカード Box<? extends Number> と一致するためです)。

リスト 3 における Box での「誤り」は、よくある誤りです。たとえエキスパートであっても、この誤りを犯すものです。プラットフォームのクラス・ライブラリーの中の数えられないほど多くの場所で、Collection<? extends T> ではなく Collection<T> が使われています。例えば java.util.concurrent パッケージの AbstractExecutorService で、invokeAll() の引数は元々 Collection<Callable<T>> でした。ただしこれでは invokeAll() を使うことが非常に面倒でした。なぜなら、Callable<T> によって厳密にパラメーター化されたコレクションを使って一連のタスクを保持する必要があり、Callable<T> を実装する何らかのクラスでパラメーター化されたコレクションを使ってタスクを保持することはできないからです。このシグニチャーは Java 6 では Collection<? extends Callable<T>> に変更されましたが、この誤りが非常に起こりやすいことを考えると、適切な修正としては invokeAll() が Collection<? extends Callable<? extends T>> という引数を取るようにする必要があったと言えるでしょう。後者の方が明らかに見た目は良くないですが、クライアントが boxing を行う必要がないというメリットがあります。

下限境界ワイルドカード

ほとんどの境界ワイルドカードは上限境界が設定されています (「? extends T」と記述すると型に上限境界が設定されます)。上限境界ほど一般的ではありませんが、「? super T」(「T または T の任意のスーパークラス」を意味します) と記述することで、型に下限境界を設定することもできます。下限境界ワイルドカードは、コンパレーターなどのコールバック・オブジェクトや、値を格納するためのデータ構造などを指定したい場合に登場します。

例えば Box を機能強化し、Box の内容を別の box の内容と比較できるようにしたいとしましょう。それには、containsSame() メソッドを使い、Comparator コールバック・オブジェクトを定義することで、Box を拡張します (リスト 5)。

リスト 5. Box に比較メソッドを追加しようとして制限が厳しすぎる場合

```
public interface Box<T> {
    public T get();
    public void put(T element);
    public void put(Box<? extends T> box);

    boolean containsSame(Box<? extends T> other,
                         EqualityComparator<T> comparator);

    public interface EqualityComparator<T> {
        public boolean compare(T first, T second);
    }
}
```

私達は、もう一方の box の型の定義を `containsSame()` の中でワイルドカードを使って行う方法を覚えました。そうすることによって、先ほど見た問題を避けることができます。しかし相変わらず同じような問題も抱えており、comparator パラメーターは厳密に `EqualityComparator<T>` でなければなりません。つまりリスト 6 のようなコードを作成することができないということです。

リスト 6. リスト 5 の比較メソッドを使って失敗する場合

```
public static EqualityComparator<Object> sameObject
    = new EqualityComparator<Object>() {
        public boolean compare(Object o1, Object o2) {
            return o1 == o2;
        }
};

...

BoxImpl<Integer> iBox = ...;
BoxImpl<Number> nBox = ...;

boolean b = nBox.containsSame(iBox, sameObject);
```

ここで `EqualityComparator<Object>` を使うことは、まったく妥当なことのように見えます。ジェネリックに指定できるにもかかわらず、なぜクライアント・コードが `Box` のすべての型に対してそれぞれ別のコンパレーターを作成する必要があるのでしょう。これに対するソリューションは、(「`? super T`」で表現される) 下限境界ワイルドカードを使うことです。`compareTo()` メソッドによって拡張された、適切なバージョンの `Box` クラスをリスト 7 に示します。

リスト 7. 下限境界ワイルドカードを使ってリスト 5 の比較操作を柔軟にしたもの

```
public interface Box<T> {
    public T get();
    public void put(T element);
    public void put(Box<? extends T> box);

    boolean containsSame(Box<? extends T> other,
                         EqualityComparator<? super T> comparator);

    public interface EqualityComparator<T> {
        public boolean compare(T first, T second);
    }
}
```

下限境界ワイルドカードを使うことによって、`containsSame()` メソッドは T または T の任意のスーパータイプと比較できる何かが必要であることを示しており、このことか

ら、`EqualityComparator<Number>` の中にラップせずに Object を比較する方法を知っているコンパレーターを提供すればよいことになります。

get と put の原則

「腕時計を 1 つ持っている人は常に正しい時刻を知っているが、2 つ持っている人は時刻を知ることができない」という古い冗談があります。Java 言語は上限境界ワイルドカードと下限境界ワイルドカードの両方をサポートしているため、どちらを使うのか、そしてそれをいつ使うのかを、どのようにして知ればよいのでしょう。

これに関しては get と put の原則 (get-put principle) という単純なルールがあり、このルールによって、どちらの種類のワイルドカードを使うのかを判断することができます。この get と put の原則は、Naftalin と Wadler による Generics に関する優れた著書『Java Generics and Collections』(「参考文献」を参照) の中で、次のように表現されています。

> 構造から値を取得する (get) だけの場合には `extends` ワイルドカードを使い、構造の中に値を格納する (put) だけの場合には `super` ワイルドカードを使い、そして両方を行う場合にはワイルドカードを使ってはなりません。

get と put の原則を最も容易に理解できるのは、`Box` のようなコンテナー・クラスや Collections クラスにこの原則を適用する場合です。その理由は、これらのクラスが行うこと (何かを保存する) を get や put という概念に自然に結びつけられるからです。そのため、例えば 1 つの `Box` から別の `Box` にコピーするメソッドを get と put の原則を適用して作成する場合、最も一般的な形式はリスト 8 のようになります。リスト 8 では上限境界ワイルドカードがコピー元に使われ、下限境界ワイルドカードがコピー先に使われています。

リスト 8. Box に対する Copy メソッド (上限境界ワイルドカードと下限境界ワイルドカードの両方を使用しています)

```
public static<T> void copy(Box<? extends T> from, Box<? super T> to) {
    to.put(from.get());
}
```

先ほど示した `containsSame()` メソッドの場合には、この get と put の原則をどのように適用するのでしょう (このメソッドでは、上限境界ワイルドカードを box に対して使い、下限境界ワイルドカードをコンパレーターに対して使いました)。get と put の原則の最初の部分は簡単です。もう一方の box から値を取得 (get) するので、`extends` ワイルドカードを使う必要があります。しかし 2 番目の部分は明確ではありません。コンパレーターはコンテナーではないため、データ構造との間では get も put も行わないように思われます。

データ型が、明らかにコレクションなどのコンテナー・クラスではない場合、get と put の原則の考え方としては、`EqualityComparator` はデータ構造ではないとはいえ、(そのメソッドの 1 つに値を渡せるという意味で) 相変わらずその中に値を「put」できると考えることです。`containsSame()` メソッドの中では、`Box` を使って値を生成し (`Box` から値を取得し)、コンパレーターを使って値を利用し (コンパレーターに値を渡し) ています。そこで `Box` に対しては `extends` ワイルドカードを使うことが適切であっても、コンパレーターには `super` ワイルドカードを使うことが適切ということになります。

`Collections.sort()` の宣言の中で get と put の原則が使われている様子をリスト 9 に示します。

リスト 9. 下限境界ワイルドカードの別の例

```
public static <T extends Comparable<? super T>> void sort(List<T>list) { ... }
```

ここでは、`Comparable` を実装した任意の型でパラメーター化された `List` をソートする、ということを言っています。しかし `sort()` の対象範囲を自分自身と同等な要素を持つリストのみに制限するのではなく、さらに一歩進め、自分のスーパータイプと自分自身を比較する方法を知っている要素のリストもソートすることができるのです。

ここではコンパレーターの中に値を置く (put) ことで 2 つの要素の相対的な順序を判定しているため、get と put の原則から、ここでは super ワイルドカードを使う必要があることがわかります。

`T` でパラメーター化されたものを `T` が継承するのは一見すると循環参照のようですが、実は決して循環参照ではなく、`List<T>` をソートできるためには T がインターフェース `Comparable<X>` を実装する必要がある、という制約を単に表現しているにすぎません (`X` は `T` または `T` のスーパータイプの 1 つです)。

get と put の原則の最後の部分、つまり get と put を両方行う場合にはワイルドカードを使ってはならない、という部分は、最初の 2 つの部分によって決まります。もし、`T` または `T` の任意のサブタイプを put することができ、`T` また `T` の任意のスーパータイプを get できるとすると、get も put も行える対象となるのは `T` そのもの、ということになります。

境界ワイルドカードを戻り値に入れない

境界ワイルドカードを、あるメソッドの戻り値の型として使いたい誘惑にかられる場合があります。しかしこの誘惑は避けた方が賢明です。なぜなら、境界ワイルドカードを返すことによってクライアント・コードが「汚染」されることがよくあるからです。あるメソッドが `Box<? extends T>` を返すとすると、その戻り値を受信する変数の型は `Box<? extends T>` でなければなりませんが、そうすると境界ワイルドカードの処理を呼び出し側に負担させることになります。境界ワイルドカードが最も有効に機能するのは API の中に使われた場合であり、クライアント・コードの中に使われた場合ではありません。

まとめ

境界ワイルドカードはジェネリックな API の柔軟性を高める上で非常に便利です。境界ワイルドカードを適切に使う上での最大の障害は、境界ワイルドカードを使う必要はない、という考え方そのものです。ある状況では下限境界ワイルドカードが必要であり、またある状況では上限境界ワイルドカードが必要であり、そのどちらを使うのかを判断するためには get と put の原則を使うことができます。

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。

© Copyright IBM Corporation 2008
(www.ibm.com/legal/copytrade.shtml)
商標
(www.ibm.com/developerworks/jp/ibm/trademarks/)