Теория и практика Java: Обобщенные типы и подстановочные символы, часть 1

Разбираемся с вхождениями подстановочных символов

Brian Goetz
Java Language Architect
Sun Microsystems

09.09.2009

Одним из самых сложных аспектов обобщенных типов в языке Java™ являются подстановочные символы или маски (wildcards), и в особенности вхождения подстановочных символов (*wildcard capture*) и связанные с ними непонятные сообщения об ошибках. В этом эпизоде цикла *Теория и практика Java* , ветеран разработки на Java Брайан Гетц расшифровывает некоторые таинственные сообщения об ошибках, выдаваемые компилятором javac, и предлагает несколько приемов и трюков, упрощающих использование обобщенных типов.

Больше статей из этой серии

Обобщенные типы (generics) вызывают дискуссии с самого момента своего появления в JDK 5. Одни говорят, что они упрощают программирование, расширяя возможности работы с системой типов и тем самым увеличивая возможности компилятора по проверке безопасности типов; другие говорят, что сложности, с которыми сопряжено их использование, не оправдываются пользой, которую они приносят. Все мы сталкивались со сложностями при работе с обобщенными типами, но, несомненно, наиболее сложным их аспектом являются подстановочные символы (wildcards).

Введение в подстановочные символы

Обобщенные типы позволяют описывать ограничения, накладываемые на поведение класса или метода, в терминах неизвестных типов. Например, "Каких бы типов ни были параметры `x` и `y` этого метода, они должны быть одного типа", "необходимо передать параметр одного типа обоим этим методам" или "возвращаемое значение метода `foo()` имеет тот же тип, что и параметр метода `bar()`."

Подстановочные символы - знаки вопроса на том месте, где обычно указывается тип данных, - это способ выражения ограничений, накладываемых на тип, в терминах неизвестного типа. Изначально они не были частью обобщенных типов (появившихся из

© Copyright IBM Corporation 2009

проекта Generic Java или GJ); они были добавлены за пять лет последующего развития языка между зарождением JSR 14 и его окончательным релизом.

Подстановочные символы играют важную роль в системе типов; с их помощью вы можете описать тип, ограниченный некоторым семейством типов, описываемых обобщенным классом. Для обобщенного класса `ArrayList`, тип `ArrayList<?>` обозначает супертип `ArrayList<T>` для любого типа `T` (также как простой тип `ArrayList` и корневой тип `Object`, которые, однако, гораздо менее полезны для определения типа).

Подстановочный тип списка `List<?>` отличается как от простого типа `List`, так и от конкретного типа `List<Object>`. Когда говорят, что переменная `x` имеет тип `List<?>`, это значит, что существует некоторый тип `T`, для которого `x` имеет тип `List<T>`, что `x` является гомогенной последовательностью, хотя и неизвестно, элементы какого именно типа она содержит. Это не значит, что содержимым может быть все что угодно, это значит, что мы не знаем, какие именно ограничения типа имеются у содержимого, — но мы знаем, что ограничения *присутствуют*. С другой стороны, простой тип `List` является гетерогенным; мы не можем накладывать никаких ограничений на тип его содержимого, а конкретный тип `List<Object>` значит, что мы точно знаем, что он может содержать любой объект. (Конечно, система обобщенных типов не имеет понятия "содержимого списка", но обобщенные типы проще всего понять в терминах типов-коллекций, таких как `List`.)

Полезность подстановочных символов в системе типов частично объясняется тем фактом, что обобщенные типы не ковариантны. Массивы - ковариантны. Например, так как `Integer` является подтипом `Number` и тип массива `Integer[]` является подтипом для `Number[]`, то значение типа `Integer[]` можно использовать везде, где требуется значение типа `Number[]`. С другой стороны, обобщенные типы - не ковариантны; тип `List<Integer>` не является подтипом `List<Number>`, и попытка использовать `List<Integer>` там, где необходимо значение типа `List<Number>` является ошибкой типов. Это не случайность и не ошибка, которой это часто считают, но такое различие в поведении обобщенных типов и массивов зачастую вызывает путаницу.

Мне встретился подстановочный символ - что дальше?

В листинге 1 показан простой тип-контейнер `Box`, который поддерживает операции `put` и `get`. `Box` параметризован параметром типа `T`, который обозначает тип элементов, которые могут храниться в этом контейнере; например `Box<String>` может содержать только элементы типа `String`.

Листинг 1. Простой обобщенный тип Box

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

Одним из преимуществ подстановочных символов является то, что они позволяют писать код, который может работать с переменными обобщенных типов, не зная, к какому конкретно типу они принадлежат. Изучим листинг 2. Метод `unbox()` принимает в качестве

параметра переменную `box` типа `Box<?>`. Что может делать метод `unbox()` с переданным ему параметром box?

Листинг 2. Метод unbox, параметризованный подстановочным символом

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

Оказывается, он может делать достаточно много: он может вызвать метод `get()`, и не только его, а любой метод, унаследованный от класса `Object` (например, `hashCode()`). Единственное, что он не может вызвать – это метод `put()`. Метод unbox() не может вызвать put() потому, что не может проверить типобезопасность такой операции, не зная тип параметра `T` для этого экземпляра `Box`. Так как `box` является переменной обобщенного типа `Box<?>`, а не обыкновенного типа `Box`, компилятор знает, что имеется некоторый тип `T`, являющийся параметром типа для `box`, но он не знает, какой именно этот тип. Значит, компилятор не может проверить, что вызов метода `put()` не будет нарушать ограничения типобезопасности класса `Box`, и поэтому он не разрешает его вызывать. (На самом деле метод `put()` можно вызвать в одном особом случае: когда вы передаете ему значение `null`. Мы можем не знать, какой именно тип представляет параметр `T`, но мы знаем, что `null` является допустимым значением для любых ссылочных типов).

Что знает метод `unbox()` о типе значения, возвращаемого методом `box.get()`? Он знает, что оно имеет тип `T` для некоторого неизвестного `T`. А, значит, наилучшее, что он может из этого заключить, - что возвращаемое значение метода `get()` является результатом операции стирания (erasure) для некоторого неизвестного типа `T`, который, для случая неограниченного подстановочного символа имеет тип `Object`. Поэтому выражение `box.get()` в листинге 2 имеет тип `Object`.

Вхождения подстановочных символов

В листинге 3 показан код, который, казалось бы, *должен* работать, но не работает. Он принимает параметр обобщенного типа `Box`, извлекает его значение и пытается поместить это значение обратно в тот же самый объект `box`.

Листинг 3. Взяв значение переменной box, мы не можем поместить его обратно

```
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
```

Казалось бы, этот код должен работать, так как извлекаемое значение подходит по типу для того, чтобы поместить его обратно, но вместо этого компилятор выдает (очень странное) сообщение об ошибке, говорящее, что "capture#337 of ?" несовместимо с типом `Object`.

Что же значит сообщение "capture#337 of ?"? Когда компилятор встречает переменную с подстановочным символом в типе, такую как параметр `box` метода `rebox()`, он знает, что есть некоторый тип `T`, для которого переменная `box` является `Box<T>`. Компилятор не знает, какой именно тип представляет `T`, но он может создать заглушку (placeholder) для этого типа для обозначения того, какой тип имеет `T`. Такая заглушка называется переменной вхождения (*capture*) В данном случае компилятор назначил подстановочному символу в типе переменной box имя "capture#337 of ?". Для каждого появления подстановочного символа в каждом объявлении переменной создается своя переменная вхождения, т.е. в декларации вида `foo(Pair<?,?> x, Pair<?,?> y)`, компилятор назначает переменные с различными именами для каждого из четырех подстановочных символов, так как никаких взаимосвязей между этими неизвестными параметрами типа нет.

Это сообщение об ошибке говорит нам, что мы не можем вызвать метод `put()`, так как компилятор не может проверить, что тип значения, переданного в `put()` совместим с типом формального параметра этого метода — потому что тип его формального параметра неизвестен. В данном случае, так как `?` фактически означает "? extends Object," компилятор уже решил, что `box.get()` возвращает значение типа `Object`, а не "capture#337 of ?", и он не может статически убедиться в том, что значение типа `Object` является допустимым значением для типа, обозначаемого переменной "capture#337 of ?."

Helper-методы

Кажется, что компилятор упускает из вида некоторую полезную информацию. Однако существует трюк, позволяющий заставить компилятор восстановить эту информацию и согласиться с нами, - он заключается в том, чтобы вместо использования подстановочного символа дать имя этому неизвестному типу. В листинге 4 показана реализация метода `rebox()`и вспомогательного обобщенного метода, который и осуществляет этот трюк:

Листинг 4. Идиома "helper-метода" для переменной вхождения

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

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

Вспомогательный метод `reboxHelper()` является *обобщенным методом*; обобщенные методы имеют дополнительные параметры типа (помещаемые в угловые скобки перед значением), обычно используемые для описания ограничений, накладываемых на тип(ы) параметров и/или возвращаемого значения. Однако в случае `reboxHelper()` обобщенной метод не использует параметр типа для описания ограничений; он позволяет компилятору (с помощью техники выведения типов) дать параметру типа имя типа переменной box.

Такой прием позволяет нам обойти ограничения компилятора, проявляющиеся в работе с подстановочными символами. Когда метод `rebox()` вызывает метод `reboxHelper()`, он знает, что делать это безопасно, так как его собственный параметр `box` должен иметь тип `Box<T>` для некоторого неизвестного `T`. Так как параметр `V` вводится в сигнатуре метода и не

привязан ни к какому другому параметру типа, он может, так же как и T, обозначать любой неизвестный тип данных, а, значит, `Box<T>` для некоторого неизвестного `T` также может быть и `Box<V>` для некоторого неизвестного `V`. (Это похоже на принцип альфа-редукции в лямбда-вычислениях, который позволяет переименовывать связанные переменные). Теперь выражение `box.get()` в методе `reboxHelper()` уже не имеет тип `Object`, оно имеет тип `V`— и теперь уже можно передать переменную типа `V` в метод `Box<V>.put()`.

Можно было бы сразу объявить метод `rebox()` обобщенным методом, как `reboxHelper()`, но подобный подход считается плохим стилем проектирования API. Руководящий принцип здесь - "не давайте имя тому, к чему вы никогда не будете обращаться по имени". В случае обобщенных методов, если параметр типа появляется только однажды в сигнатуре метода, ему скорее следует быть подстановочным символом, а не именованным параметром. В общем случае интерфейсы API с подстановочными символами проще, чем API с обобщенными методами, а увеличение количества имен типов в декларациях сложных методов, вероятнее всего, сделает декларации трудно читаемыми. Так как при необходимости имя всегда может быть восстановлено из частного вспомогательного метода, подобный подход позволяет сохранять API чистым, не теряя при этом полезной информации.

Выведение типов

Работа рассмотренного выше приема основывается на двух вещах: выведении типов (type inference) и преобразовании переменных вхождения (capture conversion). Хотя компилятор Java в очень многих случаях не производит выведения типов, он все же производит его при определении параметров типа для обобщенных методов. (Другие языки гораздо активнее используют выведение типов и, возможно, в будущем в Java для него будут добавлены дополнительные возможности.) При желании вы можете указать значение параметра типа, но только если вы можете назвать тип, — но мы не можем назвать типы, хранящиеся в переменных вхождения. Поэтому этот трюк может сработать только в единственном случае - если компилятор сделает выведение типов за нас. Именно преобразование переменных вхождения позволяет компилятору сгенерировать имя типа переменной вхождения для данного подстановочного символа, чтобы операция выведения типа могла определить, что это именно этот тип.

При обработке вызова обобщенного метода компилятор, основываясь на переданных параметрах типа, пытается определить тип метода наиболее конкретным образом. Например, для обобщенного метода:

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

и такого вызова:

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

компилятор мог бы определить тип `T` как `Integer`, `Number`, Serializable или `Object`, но он определяет его как `Integer`, поскольку это наиболее конкретный тип, подходящий по условиям.

Выведение типов можно использовать, чтобы уменьшить избыточность при конструировании параметризованных объектов. Например, чтобы с помощью нашего класса `Box` создать объект `Box<String>`, необходимо дважды указать параметр типа `String`:

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

Это является нарушением принципа DRY (Don't Repeat Yourself - не повторяйтесь) и может надоедать, даже если среда IDE способна сделать часть работы за вас. Однако если класс-реализация `BoxImpl` имеет обобщенный метод-фабрику, как показано в листинге 5 (что в любом случае является хорошей идеей), вы можете уменьшить эту избыточность в клиентском коде:

Листинг 5. Обобщенный метод-фабрика, позволяющий избегать избыточного указания параметров типа

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

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

    ...
}
```

Теперь при создании объекта `Box` с помощью метода-фабрики `BoxImpl.make()` параметр типа нужно указывать только один раз:

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

Обобщенный метод `make()` возвращает значение типа `Box<V>`для некоторого типа `V`, и возвращенное значение используется в контексте, в котором требуется значение типа `Box<String>`. Компилятор определяет, что `String` является наиболее подходящим типом для `V` с учетом имеющихся ограничений, и работает с ним дальше как с объектом типа `String`. Также у вас по-прежнему есть возможность указать значение `V` вручную:

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

Помимо экономии в нажатии клавиш, показанная здесь техника методов-фабрик имеет и другие преимущества над конструкторами: вы можете давать им более точные имена, они могут возвращать подтипы объявленного типа возвращаемого значения. Также не обязательно создавать новый экземпляр при каждом вызове фабрики, - с помощью фабрик можно реализовать совместный доступ к неизменяемым экземплярам класса. (Подробнее о преимуществах статических фабрик см. Эффективная Java, Книга #1 в разделе Ресурсы).

Заключение

Подстановочные символы – несомненно сложная тема. С ними связаны многие из самых непонятных сообщений об ошибках, выдаваемых Java компилятором и несколько наиболее сложных разделов спецификации языка Java. Однако при правильном использовании они являются чрезвычайно мощным инструментом. Два показанных в этой статье приема — helper-метод для работы с переменными вхождения и обобщенный метод-фабрика — основаны на преимуществах обобщенных методов и выведения типов, которые при правильном применении позволяют в значительной мере скрывать сложность программы.

Об авторе

Brian Goetz

Brian Goetz is the Java Language Architect at Oracle and a veteran contributor to developerWorks. Brian's writings include the *Java theory and practice* column series published here from 2002 to 2008, and the definitive work on Java concurrency, *Java Concurrency in Practice* (Addison-Wesley, 2006).

© Copyright IBM Corporation 2009
(www.ibm.com/legal/copytrade.shtml)
Торговые марки
(www.ibm.com/developerworks/ru/ibm/trademarks/)