使用通配符简化泛型使用

**简介：** 通配符是 Java™ 语言中最复杂的泛型之一，特别是围绕捕获通配符 的处理和令人困惑的错误消息。在这一期的 [*Java 理论与实践*](http://www.ibm.com/developerworks/cn/java/j-jtp/)中，资深 Java 开发人员 Brian Goetz 解释了一些由 javac 生成的怪异错误消息并提供了一些简化泛型使用的技巧和解决方法。

自从泛型被添加到 JDK 5 语言以来，它一直都是一个颇具争议的话题。一部分人认为泛型简化了编程，扩展了类型系统从而使编译器能够检验类型安全；另外一些人认为泛型添加了很多不必要的复杂性。对于泛型我们都经历过一些痛苦的回忆，但毫无疑问通配符是最棘手的部分。

**通配符基本介绍**

泛型是一种表示类或方法行为对于未知类型的类型约束的方法，比如 “不管这个方法的参数 x 和 y 是哪种类型，它们必须是相同的类型”，“必须为这些方法提供同一类型的参数” 或者 “foo() 的返回值和 bar() 的参数是同一类型的”。

通配符 — 使用一个奇怪的问号表示类型参数 — 是一种表示未知类型的类型约束的方法。通配符并不包含在最初的泛型设计中（起源于 Generic Java（GJ）项目），从形成 JSR 14 到发布其最终版本之间的五年多时间内完成设计过程并被添加到了泛型中。

通配符在类型系统中具有重要的意义，它们为一个泛型类所指定的类型集合提供了一个有用的类型范围。对泛型类 ArrayList 而言，对于任意（引用）类型 T，ArrayList<?> 类型是 ArrayList<T> 的超类型（类似原始类型 ArrayList 和根类型 Object，但是这些超类型在执行类型推断方面不是很有用）。

通配符类型 List<?> 与原始类型 List 和具体类型 List<Object> 都不相同。如果说变量 x 具有 List<?> 类型，这表示存在一些 T 类型，其中 x 是 List<T>类型，x 具有相同的结构，尽管我们不知道其元素的具体类型。这并不表示它可以具有任意内容，而是指我们并不了解内容的类型限制是什么 — 但我们知道*存在* 某种限制。另一方面，原始类型 List 是异构的，我们不能对其元素有任何类型限制，具体类型 List<Object> 表示我们明确地知道它能包含任何对象（当然，泛型的类型系统没有 “列表内容” 的概念，但可以从 List之类的集合类型轻松地理解泛型）。

通配符在类型系统中的作用部分来自其不会发生协变（covariant）这一特性。数组是协变的，因为 Integer 是 Number 的子类型，数组类型 Integer[] 是 Number[] 的子类型，因此在任何需要 Number[] 值的地方都可以提供一个 Integer[] 值。另一方面，泛型不是协变的， List<Integer> 不是 List<Number> 的子类型，试图在要求 List<Number> 的位置提供 List<Integer> 是一个类型错误。这不算很严重的问题 — 也不是所有人都认为的错误 — 但泛型和数组的不同行为的确引起了许多混乱。

**我已使用了一个通配符 — 接下来呢？**

清单 1 展示了一个简单的容器（container）类型 Box，它支持 put 和 get 操作。 Box 由类型参数 T 参数化，该参数表示 Box 内容的类型， Box<String> 只能包含 String 类型的元素。

**清单 1. 简单的泛型 Box 类型**

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

通配符的一个好处是允许编写可以操作泛型类型变量的代码，并且不需要了解其具体类型。例如，假设有一个 Box<?> 类型的变量，比如清单 2 unbox() 方法中的 box 参数。unbox() 如何处理已传递的 box？

**清单 2. 带有通配符参数的 Unbox 方法**

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

事实证明 Unbox 方法能做许多工作：它能调用 get() 方法，并且能调用任何从 Object 继承而来的方法（比如 hashCode()）。它惟一不能做的事是调用 put() 方法，这是因为在不知道该 Box 实例的类型参数 T 的情况下它不能检验这个操作的安全性。由于 box 是一个Box<?> 而不是一个原始的 Box，编译器知道存在一些 T 充当 box 的类型参数，但由于不知道 T 具体是什么，您不能调用 put() 因为不能检验这么做不会违反 Box 的类型安全限制（实际上，您可以在一个特殊的情况下调用 put()：当您传递 null 字母时。我们可能不知道 T 类型代表什么，但我们知道 null 字母对任何引用类型而言是一个空值）。

关于 box.get() 的返回类型，unbox() 了解哪些内容呢？它知道 box.get() 是某些未知 T 的 T，因此它可以推断出 get() 的返回类型是 T 的擦除（erasure），对于一个无上限的通配符就是 Object。因此清单 2 中的表达式 box.get() 具有 Object 类型。

**通配符捕获**

清单 3 展示了一些似乎*应该* 可以工作的代码，但实际上不能。它包含一个泛型 Box、提取它的值并试图将值放回同一个 Box。

**清单 3. 一旦将值从 box 中取出，则不能将其放回**

|  |
| --- |
| 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 |

这个代码看起来应该可以工作，因为取出值的类型符合放回值的类型，然而，编译器生成（令人困惑的）关于 “capture#337 of ?” 与Object 不兼容的错误消息。

“capture#337 of ?” 表示什么？当编译器遇到一个在其类型中带有通配符的变量，比如 rebox() 的 box 参数，它认识到必然有一些 T，对这些 T 而言 box 是 Box<T>。它不知道 T 代表什么类型，但它可以为该类型创建一个占位符来指代 T 的类型。占位符被称为这个特殊通配符的*捕获（capture）*。这种情况下，编译器将名称 “capture#337 of ?” 以 box 类型分配给通配符。每个变量声明中每出现一个通配符都将获得一个不同的捕获，因此在泛型声明 foo(Pair<?,?> x, Pair<?,?> y) 中，编译器将给每四个通配符的捕获分配一个不同的名称，因为任意未知的类型参数之间没有关系。

错误消息告诉我们不能调用 put()，因为它不能检验 put() 的实参类型与其形参类型是否兼容 — 因为形参的类型是未知的。 在这种情况下，由于 ? 实际表示 “？extends Object” ，编译器已经推断出 box.get() 的类型是 Object，而不是 “capture#337 of ?”。它不能静态地检验对由占位符 “capture#337 of ?” 所识别的类型而言 Object 是否是一个可接受的值。

**捕获助手**

虽然编译器似乎丢弃了一些有用的信息，我们可以使用一个技巧来使编译器重构这些信息，即对未知的通配符类型命名。清单 4 展示了 rebox() 的实现和一个实现这种技巧的泛型助手方法（helper）：

**清单 4. “捕获助手” 方法**

|  |
| --- |
| public void rebox(Box<?> box) {  reboxHelper(box);  }  private<V> void reboxHelper(Box<V> box) {  box.put(box.get());  } |

助手方法 reboxHelper() 是一个*泛型方法*， 泛型方法引入了额外的类型参数（位于返回类型之前的尖括号中），这些参数用于表示参数和/或方法的返回值之间的类型约束。然而就 reboxHelper() 来说，泛型方法并不使用类型参数指定类型约束，它允许编译器（通过类型接口）对 box 类型的类型参数命名。

捕获助手技巧允许我们在处理通配符时绕开编译器的限制。当 rebox() 调用 reboxHelper() 时，它知道这么做是安全的，因为它自身的 box 参数对一些未知的 T 而言一定是 Box<T>。因为类型参数 V 被引入到方法签名中并且没有绑定到其他任何类型参数，它也可以表示任何未知类型，因此，某些未知 T 的 Box<T> 也可能是某些未知 V 的 Box<V>（这和 lambda 积分中的 α 减法原则相似，允许重命名边界变量）。现在 reboxHelper() 中的表达式 box.get() 不再具有 Object 类型，它具有 V 类型 — 并允许将 V 传递给 Box<V>.put()。

我们本来可以将 rebox() 声明为一个泛型方法，类似 reboxHelper()，但这被认为是一种糟糕的 API 设计样式。此处的主要设计原则是 “如果以后绝不会按名称引用，则不要进行命名”。就泛型方法来说，如果一个类型参数在方法签名中只出现一次，它很有可能是一个通配符而不是一个命名的类型参数。一般来说，带有通配符的 API 比带有泛型方法的 API 更简单，在更复杂的方法声明中类型名称的增多会降低声明的可读性。因为在需要时始终可以通过专有的捕获助手恢复名称，这个方法让您能够保持 API 整洁，同时不会删除有用的信息。

**类型推断**

捕获助手技巧涉及多个因素：类型推断和捕获转换。Java 编译器在很多情况下都不能执行类型推断，但是可以为泛型方法推断类型参数（其他语言更加依赖类型推断，将来我们可以看到 Java 语言中会添加更多的类型推断特性）。如果愿意，您可以指定类型参数的值，但只有当您能够命名该类型时才可以这样做 — 并且不能够表示捕获类型。因此要使用这种技巧，要求编译器能够为您推断类型。捕获转换允许编译器为已捕获的通配符产生一个占位符类型名，以便对它进行类型推断。

当解析一个泛型方法的调用时，编译器将设法推断类型参数它能达到的最具体类型。 例如，对于下面这个泛型方法：

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

和它的调用：

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

编译器能够推断 T 是 Integer、Number、 Serializable 或 Object，但它选择 Integer 作为满足约束的最具体类型。

当构造泛型实例时，可以使用类型推断减少冗余。例如，使用 Box 类创建 Box<String> 要求您指定两次类型参数 String：

|  |
| --- |
| Box<String> box = new BoxImpl<String>(); |

即使可以使用 IDE 执行一些工作，也不要违背 DRY（Don't Repeat Yourself）原则。然而，如果实现类 BoxImpl 提供一个类似清单 5 的泛型工厂方法（这始终是个好主意），则可以减少客户机代码的冗余：

**清单 5. 一个泛型工厂方法，可以避免不必要地指定类型参数**

|  |
| --- |
| public class BoxImpl<T> implements Box<T> {  public static<V> Box<V> make() {  return new BoxImpl<V>();  }  ...  } |

如果使用 BoxImpl.make() 工厂实例化一个 Box，您只需要指定一次类型参数：

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

泛型 make() 方法为一些类型 V 返回一个 Box<V>，返回值被用于需要 Box<String> 的上下文中。编译器确定 String 是 V 能接受的满足类型约束的最具体类型，因此此处将 V 推断为 String。您还可以手动地指定 V 的值：

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

除了减少一些键盘操作以外，此处演示的工厂方法技巧还提供了优于构造函数的其他优势：您能够为它们提高更具描述性的名称，它们能够返回命名返回类型的子类型，它们不需要为每次调用创建新的实例，从而能够共享不可变的实例（参见 [参考资料](http://www.ibm.com/developerworks/cn/java/j-jtp04298.html#resources) 中的 Effective Java, Item #1，了解有关静态工厂的更多优点）。

**简介：** 在使用 Java™ 语言的泛型时，通配符非常令人困惑，并且最常见的一个错误就是在使用有界通配符的两种形式的其中之一（“? super T” 和 “? extends T”）时出现错误。您出错了吗？别沮丧，即使是专家也会犯这种错误，本月 Brian Goetz 将展示如何避免这个错误。

在 Java 语言中，数组是协变的（因为一个 Integer 同时也是一个 Number，一个 Integer 数组同时也是一个 Number 数组），但是泛型不是这样的（List<Integer> *并不等于* List<Number>）。人们会争论哪些选择是 “正确的”，哪些选择是 “错误的” — 当然，每种选择都各有优缺点 — 但有一点毫无疑问，存在两种使用差别很小的语义构造派生类型的类似机制，这将导致大量错误和误解。

有界通配符（一些有趣的 “? extends T” 通用类型说明符）是语言提供的一种工具，用来处理协变性缺乏 — 有界通配符允许类声明方法参数或返回值何时具有协变性（或相反，声明方法参数或返回值何时具有*逆变性（contravariant）*）。虽然了解何时使用有界通配符是泛型较为复杂的方面，但是，使用有界通配符的压力通常都落在库作者的身上，而非库用户。最常见的有界通配符错误就是忘记使用它们，这就限制了类的使用，或是强制用户不得不重用现有的类。

**有界通配符的作用**

让我们从一个简单的泛型类开始（一个称为 Box 的值容器），它持有一个具有已知类型的值：

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

由于泛型不具备协变性，Box<Integer> 并不等同于 Box<Number>，尽管 Integer 属于 Number。但是对于 Box 这样的简单泛型类来说，这不成问题，并且常常被忽略，因为 Box<T> 的接口完全指定为 T 类型的变量 — 而不是通过 T 泛型化的类型。直接处理类型变量允许实现多态性。清单 1 展示了这种多态性的两个示例：获取 Box<Integer> 的内容，并将它作为一个 Number，然后将一个 Integer 放入Box<Number> 中：

**清单 1. 通过泛型类利用固有的多态性**

|  |
| --- |
| 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); |

通过使用简单的 Box 类，使我们确信可以没有协变性，因为在需要实现多态的位置，数据已经具有某种形式，使编译器能够应用适当的子类型规则。

然而，如果希望 API 不仅能够处理 T 类型的变量，还能处理通过 T 泛型化的类型，事情将变得更加复杂。假设希望将一个新的方法添加到 Box，该方法允许获得另一个 Box 的内容并其放到清单 2 所示的 Box 中：

**清单 2. 扩展的 Box 接口并不灵活**

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

这个扩展 Box 的问题是，只能将内容放到类型参数与原 box 完全相同的 Box 中。因此，清单 3 中的代码就不能进行编译：

**清单 3. 泛型不具备协变性**

|  |
| --- |
| Box<Number> nBox = new BoxImpl<Number>();  Box<Integer> iBox = new BoxImpl<Integer>();  nBox.put(iBox); // ERROR |

显示一条错误消息，表示无法在 Box<Number> 中找到方法 put(Box<Integer>)。如果认为泛型是不具有协变性的，这条错误还讲得通；一个 Box<Integer> 不是 Box<Number>，尽管 Integer 是 Number，但是这使得 Box 类的 “泛型性” 比我们期望的要弱。要提高泛型代码的有效性，可以指定一个上限（或下限），而不是指定某个泛型类型参数的精确类型。这可以使用有界通配符来实现，它的形式为 “? extends T” 或 “? super T”。（有界通配符只能用作类型参数，而不能作为类型本身 — 因此，需要一个有界的命名的类型变量）。在清单 4 中，修改了 put() 的签名以使用一个上限通配符 — Box<? extends T>，这表示 Box 的类型参数可以是 T 或 T 的任何子类。

**清单 4. 对清单 3 的 Box 类的改进解释了协变性**

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

现在，[清单 3](http://www.ibm.com/developerworks/cn/java/j-jtp07018.html#listing3) 中的代码可以进行编译并执行，因为 put() 的参数现在可以是参数类型为 T 或 T 的子类型的 Box。由于 Integer 是Number 的子类型，编译器能够解析方法引用 put(Box<Integer>)，因为 Box<Integer> 匹配有界通配符 Box<? extends Number>。

很容易犯清单 3 中的 Box 错误，即使是专家也难以避免 — 在平台类库中，许多地方都使用 Collection<T>，而不是 Collection<? extends T>。例如，在 java.util.concurrent 包的 AbstractExecutorService 中，invokeAll() 的参数最初是一个Collection<Callable<T>>。但是，这样使用 invokeAll() 非常麻烦，因为这要求必须由 Callable<T> 参数化的集合持有任务集，而不是由实现 Callable<T> 的类参数化的集合。在 Java 6 中，这种签名被修改为 Collection<? extends Callable<T>> — 这只是为了演示非常容易犯这个错误，正确的修复应该是使 invokeAll() 包含一个 Collection<? extends Callable<? extends T>> 参数。这个参数无疑更加难看，但不会给客户机带来麻烦。

**下限通配符**

上面的大多数有界通配符都进行了限定；“? extends T” 符号为类型添加了一个上限。但是，虽然比较少见，仍然可以使用 “? super T” 符号为类型添加一个下限，表示 “类型 T 以及它的任何超类”。当您希望指定一个回调对象（例如一个比较器）或存放某个值的数据结构，可以使用下限通配符。

假设我们希望增强 Box，使它能够与另一个 box 的内容进行比较。可以通过 containsSame() 方法和 Comparator 回调对象的定义扩展Box，如清单 5 所示：

**清单 5. 尝试向 Box 添加一个比较方法**

|  |
| --- |
| 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);  }  } |

可以使用一个通配符定义 containsSame() 中另一个 box 的类型，这将避免前面遇到的问题。但是仍然会遇到一个类似的问题；比较器参数必须是 EqualityComparator<T>。这意味着我们不能编写如清单 6 所示的代码：

**清单 6. 使用清单 5 中的比较方法会导致失败**

|  |
| --- |
| 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); |

在这里使用一个 EqualityComparator<Object> 似乎非常合理。既然可以使用泛型指定，客户机就不必为每一个可能的 Box 类型创建独立的比较器了！解决方法是使用一个下限通配符 “? super T”。使用 compareTo() 方法扩展的正确 Box 类如清单 7 所示：

**清单 7. 清单 5 中的比较操作在使用有界通配符后更加灵活**

|  |
| --- |
| 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);  }  } |

通过使用一个下限通配符，containsSame() 方法表示需要能够比较 T *或它的任何超类型* 的工具，这就允许我们提供一个能够比较对象的比较器，并且不需要使用 EqualityComparator<Number> 封装它。

**get-put 原则**

有一个流传已久的笑话：“佩戴一只手表的人常常知道时间，而佩戴两只手表后反而难以确定了”。由于 Java 语言同时支持上限和下限通配符，那么如何判断何时使用哪一种呢？

这里有一条简单的规则，称为 *get-put 原则*，它解释了应该使用哪一种通配符。get-put 原则首次出现在 Naftalin 和 Wadler 所著的有关泛型的 *Java Generics and Collections* 一书中（参见参考资料），它是这样描述的：

仅从某个结构中获取值时使用 extends 通配符；仅将值放入某个结构时使用 super 通配符；同时执行以上两种操作时不要使用通配符。

在应用到 Box 等容器类或 Collections 类时，get-put 原则很好理解，因为 get 和 put 概念和这些类的作用有着自然的联系：存储内容。因此，如果希望应用 get-put 原则来创建一个可以在 Box 之间进行复制的方法，最常见的形式如清单 8 所示，其中复制源使用上限通配符，目标使用下限通配符：

**清单 8. 同时使用上限和下限通配符的 Box 复制方法**

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

如果对前面的 containsSame() 方法（对 box 使用了上限通配符而对比较器使用了下限通配符）应用 get-put 原则？第一步很简单：需要从其他 box 获取一个值，因此使用一个 extends 通配符。但第二步有点复杂 — 因为比较器并不是容器，因此与从一个数据结构获得或存入值有所不同。

当数据类型并不是一个明显的容器类（例如集合）时，应该这样考虑 get-put 原则：尽管 EqualityComparator 不是一个数据结构，仍然可以向它 “存入” 值 — 即将值传递给它的一个方法。在 containsSame() 方法中，使用 Box 作为值的生成器（从 Box 获取值）并使用比较器作为值的使用者（将值传递给比较器）。因此可以对 Box 使用 extends 通配符，而对比较器使用 super 通配符。

我们可以看到 get-put 应用到了 Collections.sort() 的声明中，如清单 9 所示：

**清单 9. 使用下限通配符的另一个示例**

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

在这里，可以对 List 排序，它由实现 Comparable 的任何类型参数化。但是没有将 sort() 的域限制为具有可相互比较的元素的列表，而是更进一步 — 我们还对能够与其超类型相比较的列表元素进行排序。由于将值放入到比较器来决定两个元素的相对顺序，get-put 原则告诉我们在这里需要使用一个超通配符。

表面上看似循环引用 — T 扩展经过 T 参数化的内容 — 实际上并不是真正的循环。它只是表示对 List<T> 排序的限制，T 需要实现接口 Comparable<X>，其中 X 是 T 或是 T 的一个超类型。

接着是 get-put 原则的最后一部分 — 当同时执行 get 和 put 操作时不要使用通配符。如果可以存入 T 或它的任意子类型，那么就可以获得 T 或它的任意超类型，而惟一可以同时获得或存入的是 T 本身。

**不要对返回值使用通配符**

有时希望对方法的返回类型使用有界通配符。但最好不要这样做，因为返回的有界通配符往往会 “弄脏” 客户机代码。如果某方法返回一个 Box<? extends T>，那么接收返回类型的变量的类型必须是 Box<? extends T>，这将处理有界通配符的工作推给了调用者。有界通配符最适合用于 API，而不是客户机代码。