侯捷觀點

JDK 1.5 的泛型實現

— Generics in JDK 1.5 —

北京《程序員》2004/09
台北《Run!PC》2004/09

作者簡介：侯捷，資訊教育、專欄執筆、大學教師。常著文章自娛，頗示己志。
侯捷網站：http://www.jjhou.com（繁體）
北京鏡站：http://jjhou.csdn.net（簡體）
永久郵箱：jjhou@jjhou.com

- 讀者基礎：有 Java 語言基礎，使用過 Java Collections。
- 本文適用工具：JDK1.5
- 本文程式源碼可至侯捷網站下載
  http://www.jjhou.com/javatwo-2004-reflection-and-generics-in-jdk15-sample.ZIP
- 本文是 JavaTwo-2004 技術研討會同名講題之部分內容書面整理。
- 關鍵術語：
  persistence（永續性、持久性）
  serialization（序列化、次第讀寫）
  generics（泛型）
  polymorphism（多型）

全文提要

泛型技術與 Sun JDK 的淵源可追溯自 JDK1.3。但無論 JDK 1.3 或 JDK1.4，都只是以編譯器外掛附件的方式來支援泛型語法，並且 Java 標準程式庫未曾針對泛型全

面改寫。而今 JDK1.5 正式納入泛型。本文討論 JDK1.5 的泛型實現，包括如何使用及自訂 generic classes and algorithms，其中若干語法異於 JDK 1.3 和 1.4。

我曾經在JavaTwo 2002大會上針對泛型技術給出一個講題，並將內容整理爲《Java 泛型技術之發展》一文（http://www.jjhou.com/javatwo-2002.htm）。該文所談的 Java 泛型語法以及 Java 泛型技術之內部實作技術，在今天（被 JDK 1.5 正式納入）依然適用。但由於有了若干小變化，並且由於 Java 標準程式庫的全面改寫，使我認爲有必要再整理這篇文章，讓讀者輕鬆地在 JDK 1.5 中繼續悠遊「泛型」技術。

閱讀本文之前，如果自覺基礎不夠，可以補充閱讀適才提到的《Java 泛型技術之發展》，那是一篇非常完整的文章，可助您完整認識泛型技術的來龍去脈。

Sun JDK 的泛型發展歷史要從 1.3 版說起。該版本配合 GJ，正式進入泛型殿堂。所謂 GJ 是 "Generic Java" 的縮寫，是一個支援泛型的 Java 編譯器補充件，可謂 Java 泛型技術的先趨。隨後，泛型議題正式成爲 JSR #14，其技術基礎便是源自 GJ。JDK1.4 搭配 JSR14 提供的外掛附件，使泛型技術在 Java 世界從妾身未明的身份扶正而爲眾所屬目的焦點。今天，JDK1.5 終於內建泛型特性，不僅編譯器不再需要任何外力（外掛附件）的幫助，整個 Java 標準程式庫也被翻新（retrofit），許多角落針對泛型做了改寫。

讓我們把帶有「參數化型別」（parameterized types）的 classes 稱爲 generic classes，把帶有「參數化型別」的 methods 稱爲 generic algorithms，那麼，對眾多 Java 程式員而言，泛型帶來的影響不外乎以下四點，稍後逐一說明。

- 如何使用 generic classes
- 如何使用 generic algorithms
- 如何自訂 generic classes
- 如何自訂 generic algorithms

在此先提醒您，運用泛型時，加上 `-Xlint:unchecked` 編譯選項，可讓編譯器幫助我們檢查潛在的型別轉換問題。

何謂 Generic Classes

Generic classes 的最大宗運用是 collections（群集），也就是實作各種資料結構（例如 list, map, set, hashtable）的那些 classes。也有人稱它們爲容器（containers）。這些容器被設計用來存放 `Object`-derived 元素。而由於 Java 擁有單根繼承體系，任何 Java classes 都繼承自 `java.lang.Object`，因此任何 Java objects 都可以被放進上述各種容器。換句話說 Java 容器是一種異質容器，從「泛型」的字面意義來說，其實這（原本的設計）才是「泛型」。

然而有時候，而且是大半時候，我們不希望容器元素如此異質化。我們多半希望使用同質容器。即使用於多型（polymorphism），我們也希望至少相當程度地規範容器，令其元素型別爲「帶有某種約束」的 base class。例如面對一個準備用來放置各種形狀（圓圈、橢圓、矩形、四方形、三角形…）的容器，如果我們能夠告知這個容器其每個元素都必須是 `Shape`-derived objects，將相當有助於程式的可讀性，並減少錯誤，容易除錯，甚至可避免一大堆轉型（cast）動作。

Java 同質容器的語法如下，其中角括號（<>）的用法和 C++完全相同，角括號之內的指定型別，就是同質容器的元素型別，如圖 1。

```
ArrayList<String> strList = new ArrayList<String>();
strList.add("zero");
strList.add("one");
strList.add("two");
strList.add("five");
System.out.println(strList);  // [zero, one, two, five]
```

圖 1 / 同質容器的用法。角括號（<>）內就是元素型別。

下面是另一個實例，程式員要求容器內的每一個元素都必須是「一種形狀」，這是一種「多型」應用，如圖 2。這些泛型語法自 JDK 1.3+GJ 以來不曾改變過。

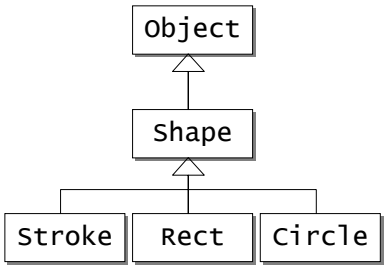


圖 2a / 典型的 "Shape" 多型繼承體系。

```
//假設 Stroke, Rect, Circle 皆繼承自 Shape
LinkedList<Shape> sList = new LinkedList<Shape>();
sList.add(new Stroke(...));
sList.add(new Rect(...));
sList.add(new Circle(...));
```

圖 2b / 令容器內含各種 Shape 元素，並加入一個 Stroke，一個 Rect 和一個 Circle。

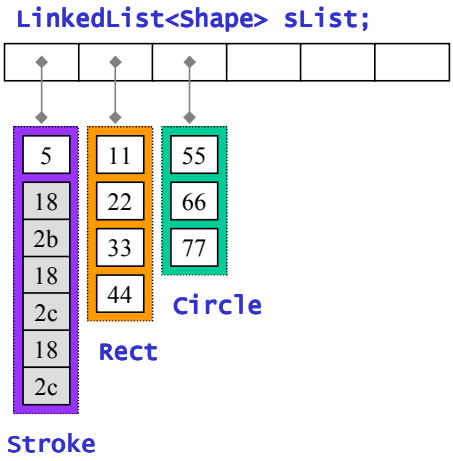


圖 2c / 圖 2b 程式碼所製造的結果。

Boxing 和 Un-boxing 帶來的影響

前面曾經說過，任何 Java objects 都可以被放進各種容器內。但是 Java 基本數值型別（primitive types，例如 `int`, `double`, `long`, `char`）並不是一種 class，而數值也談不上是個 object。如果要把這一類數值放進容器內，必須將容器元素宣告爲基本型別所對應的外覆類別（wrapper classes），例如圖 3。這實在是非常不方便。JDK1.5

新增自動 boxing（封箱）和 un-boxing（拆箱）特性，也就是在必要時刻自動將數值轉爲外覆物件，或將外覆物件轉爲數值。有了這項特性，我們可以將圖 3 改寫爲圖 4，那就方便多了。

```
LinkedList<Integer> iList = new LinkedList<Integer>();
iList.add(new Integer(0));
iList.add(new Integer(1));
iList.add(new Integer(5));
iList.add(new Integer(2));
```

圖 3 / 容器元素必須是 object，不可是數值，所以必須使用外覆型別（wrapper）。

```
LinkedList<Integer> iList = new LinkedList<Integer>();
iList.add(0);           //boxing
iList.add(1);
iList.add(5);
iList.add(2);
int i = iList.get(2);   //un-boxing
```

圖 4 / JDK1.5 新增的 boxing/un-boxing 特性，使得以方便地將數值放進容器。

使用 Generic Algorithms

在 Java 程式庫中，針對容器而設計的 algorithms 並不多（不像 C++ 標準程式庫所提供的那麼多），它們都被置於 `java.util.Collections` 內以 static methods 的形式呈現，例如 `sort()`, `max()`, `min()`, `copy()`, `fill()`。圖 5 是兩個運用實例，其語法和 C++完全相同：使用 generic algorithms 時並不需要以角括號（<>）爲「參數化型別」做任何具體指定。這種泛型語法自 JDK1.3+GJ 以來不曾改變過。

```
String str = Collections.max(strList);  //strList 見前例（圖 1）
Collections.sort(strList);
```

圖 5 / 運用 `max()`和 `sort()`

自訂 Generic Classes

先前的 `LinkedList<T>` 運用實例中，我曾假設 `Stroke`, `Rect`, `Circle` 皆繼承自 `Shape`。如果我們希望這些 classes 有足夠的彈性，讓用戶得以在運用這些 classes 時才指定其內部數據（長、寬、半徑等等）的型別，那就得用上泛型語法，如圖 6，

而先前的運用實例也得對應地修改爲圖 7。

```
public abstract class Shape {
  public abstract void draw();
}
```

```
public class Rect<T> extends Shape
                    implements Serializable {
  T m_left, m_top, m_width, m_height;
  public Rect(T left, T top, T width, T height ) { ... }
  ...
}
```

```
public class Circle<T> extends Shape
                     implements Serializable {
  T m_x, m_y, m_r;
  public Circle(T x, T y, T r) { ... }
  ...
}
```

```
public class Stroke<W,T> extends Shape
                       implements Serializable {
  W m_width;
  ArrayList<T> m_ia;
  public Stroke(W width, ArrayList<T> ia) { ... }
  ...
}
```

圖 6 / 自訂 generic classes。本圖實現圖 2a 的繼承體系，並以「參數化型別」（圖中灰色的 `T,W` 等等）代表各 classes 內的數據型別。

```
    LinkedList<Shape> sList = new LinkedList<Shape>();
    sList.add(new Stroke<Integer,Integer>(...));
    sList.add(new Rect<Integer>(...));
    sList.add(new Circle<Integer>(...));
```

圖 7 / 容器的每個元素型別都是 generic classes，所以製造元素時必須使用泛型語法（角括號）。請與圖 2b 比較。

圖 6 和圖 7 的泛型語法自 JDK1.3+GJ 以來不曾改變過。它迥異於 C++，後者要求程式必須在 class 名稱前加上語彙單元 `template<>`，藉此告訴編譯器哪些符號是型別參數（type parameters），如圖 8。

```
template <typename T>
class Rect : public Shape
```

```
{
  private:
     T m_left, m_top, m_width, m_height;
  public:
     Rect(T left, T top, T width, T height ) { ... }
     ...
}
```

圖 8 / C++ class 必須以 `template<typename T>` 這種語彙單元型式，告訴編譯器 `T` 是個參數化型別。請與圖 6 之同名 Java class 比較。

現在讓我們看看 Java 程式庫源碼，從中學習更多的泛型語法。圖 9a 是 `java.util.ArrayList` 的 JDK1.5 源碼，圖 9b 是其 JDK 1.4 源碼，可資比較。

```
#001 public class ArrayList<E> extends AbstractList<E>
#002                          implements List<E>, RandomAccess,
#003                             Cloneable, java.io.Serializable
#004 {
#005     private transient E[] elementData;
#006     private int size;
#007     public ArrayList(int initialCapacity) {
#008          super();
#009          // check if (initialCapacity < 0)...
#010          this.elementData = (E[])new Object[initialCapacity];
#011     }
#012
#013     public ArrayList() {
#014          this(10);
#015     }
#016     ...
#017 }
```

圖 9a / JDK1.5 的 `java.util.ArrayList` 源碼

```
#001 public class ArrayList extends AbstractList
#002                        implements List, RandomAccess,
#003                               Cloneable, java.io.Serializable
#004 {
#005     private transient Object elementData[];
#006     private int size;
#007     public ArrayList(int initialCapacity) {
#008        super();
#009        // check if (initialCapacity < 0) ...
#010        this.elementData = new Object[initialCapacity];
#011     }
#012
```

```
#013     public ArrayList() {
#014        this(10);
#015     }
#016     ...
#017 }
```

圖 9b / JDK1.4 的 java.util.ArrayList 源碼

從圖 9a 可以看出，參數型別（圖中的 E）不但可以繼續被沿用做爲 base class 或 base interfaces 的參數型別，也可以出現在 class 定義區內「具體型別可以出現」的任何地方。不過，例外還是有的，例如這一行：

```
#010 this.elementData = (E[])new Object[initialCapacity];
```

不能寫成：

```
#010 this.elementData = new E[initialCapacity];
```

那會出現 generic array creation error.

目訂 Generic Algorithms

定義於任何 classes 內的任何一個 static method，你都可以說它是個 algorithm。如果這個 method 帶有參數化型別，我們就稱它是 generic algorithm。例如：

```
//在某個 class 之內
public static <T> T gMethod (List<T> list) { ... }
```

這種語法和 generic classes 有相當程度的不同：泛型符號<T> 必須加在 class 名稱之後，卻必須加在 method 名稱（及回傳型別）之前。

JDK 1.5 比以前版本增加了更多彈性，允許所謂 **bounded type parameter**，意指「受到更多約束」的型別參數。下例表示 gMethod()所收到的引數不但必須是個 List，而且其元素型別必須實作 Comparable：

```
public static <T extends Comparable<T>> T gMethod (List<T> list)
{ ... }
```

這種「受到更多約束」的型別參數寫法，雖然不存在於 JDK1.4+JSR14，但其實原本存在於 JDK1.3+GJ 中，只不過用的是另一個關鍵字：

```
public static <T implements Comparable<T>> T gMethod (List<T> list)
```

JDK 1.5 還允許將「不被 method 實際用到」的型別參數以符號 '?' 表示，例如：

```
public static    List<?> gMethod (List<?> list)
{
  return list;    // 本例簡單地原封不動傳回
}
```

此例 gMethod()接受一個 List（無論其元素型別是什麼），傳回一個 List（無論其元素型別是什麼）。由於不存在（或說不在乎）型別參數（因爲 method 內根本不去用它），也就不必如平常一般在回傳型別之前寫出<T>來告知編譯器了。

上面這個例子無法真正表現出符號 '?' 的用途。真正的好例子請看 JDK1.5 的 java.util.Collections 源碼，見圖 10a。圖 10b 則是其 JDK1.4 源碼，可資比較。請注意，例中的 '?' 不能被替換爲任何其他符號。圖 10a 程式碼所描述的意義，請見圖 11 的細部解釋。

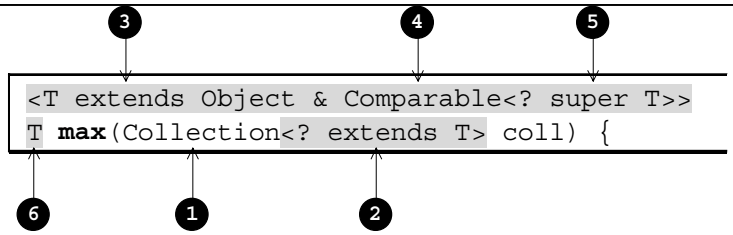
```
#001 public class Collections
#002 ...
#003   public static
#004   <T extends Object & Comparable<? super T>>
#005   T max(Collection<? extends T> coll) {
#006      Iterator<? extends T> i = coll.iterator();
#007      T candidate = i.next();
#008
#009      while(i.hasNext()) {
#010         T next = i.next();
#011         if (next.compareTo(candidate) > 0)
#012              candidate = next;
#013      }
#014      return candidate;
#015   }
#016   ...
#017 } // of Collections
```

圖 10a / JDK1.5 的 java.util.Collections 源碼。

```
#001 public class Collections
#002 ...
#003    public static
#004                          //這裡我刻意放空一行，以利與 JDK1.5 源碼比較
```

```
#005     Object max(Collection coll) {
#006    Iterator i = coll.iterator();
#007    Comparable candidate = (Comparable)(i.next());
#008
#009        while(i.hasNext()) {
#010           Comparable next = (Comparable)(i.next());
#011           if (next.compareTo(candidate) > 0)
#012                candidate = next;
#013        }
#014        return candidate;
#015     }
#016 ...
#017 } // of Collections
```

圖 10b / JDK1.4 的 `java.util.Collections` 源碼。



1. `max()` 接收一個 `Collection` object。
2. 該 `Collection` object 所含元素必須是 `T`-derived object。
3. `T` 必須繼承 `Object`（這倒不必明說，因爲 Java 必定如此）。
4. `T` 必須實作 `Comparable`。
5. `Comparable` 所比較的型別必須是 T 的 super type。
6. `max()` 傳回 `T` object。

圖 11 / 本圖詳細說明圖 10a 的怪異內容（#4, #5 兩行）

面對圖 11 如此「怪異而罕見」的語法，給個實際用例就清楚多了：

```
LinkedList<Shape> sList = new LinkedList<Shape>();
...
Shape s = Collections.max(sList);
```

我們讓 `Collections.max()` 接受一個先前曾經說過的 `LinkedList<Shape>`（見圖 2a,b），那是個 `Collections` object（符合圖 11 條件 1），其中每個元素都是 `Shape`-derived objects（符合圖 11 條件 2），因此本例中的 `T` 就是 `Shape`。`Shape` 的確繼承自 `Object`（符合圖 11 條件 3），並且必須實作 `Comparable`（才能符合圖 11

條件 4），而被比較物的型別必須是 Shape 的 super class（才能符合圖 11 條件 5）。max()比較所得之最大值以 Shape 表示（符合圖 11 條件 6）——這是合理的，因爲不知道比較出來的結果會是 Rect 或 Circle 或 Stroke，但無論如何它們都可以向上轉型爲 Shape。

爲了完成上述的條件 4 和條件 5，先前的 Shape 必須修改，使得以被比較。也就是說 Shape 必須實作 Comparable 介面，如圖 12，其中針對 compareTo()用上了典型的 **Template Method** 設計範式（design pattern），再令每一個 Shape-derived classes 都實作 L()如圖 13，這就大功告成了。

```
public abstract class Shape implements Comparable<Shape>  {
  ...
  public abstract double L();         //計算周長
  public int compareTo(Shape o) {     //假設「以周長爲比較依據」合理!
    return (this.L() < o.L() ? -1 : (this.L() == o.L() ? 0 : 1));
  }
}
```

圖 12 / 修改圖 7 的 Shape class

```
  public double L() {
    ... 計算周長     //這裡有點學問，見最後「擦拭法帶來的遺憾」討論。
    return xxx;     //傳回周長
  }
```

圖 13 / 每個 Shape-derived classes 都必須實作出如此型式的 L()。

參數化型別（Parameterized type）存在多久？

究竟 generic classes 所帶的參數化型別，在編譯後是否還保留？或者說，假設我們將一個元素型別爲 Integer 的 LinkedList 容器寫入檔案，而後讀出並恢復「先前被 serialized（序列化）至檔案」的容器，如圖 14。此時我們的第一個考慮作法或許如下（和最初的宣告完全相同）：

```
LinkedList<Integer> iList2 = (LinkedList<Integer>)in.readObject();
```

但 JDK1.5 編譯器發出警告，告訴我們 "unchecked cast"（JDK1.4 則直接抱怨它是錯誤的）。改成這樣情況亦同：

```
LinkedList iList2 = (LinkedList<Integer>)in.readObject();
```

看來編譯器面對即將被 deSerialized（反序列化）讀得的型別資訊，似乎無法判別是否可以成功轉型爲 `LinkedList<Integer>`。另一種寫法是：

```
LinkedList<Integer> iList2 = (LinkedList)in.readObject();
```

這一次 JDK1.5 編譯器發出的警告訊息是："unchecked conversion"。改爲這樣更非警告可以善了：

```
LinkedList<Integer> iList2 = in.readObject();
```

對此 JDK1.5 編譯器會直接報錯："incompatible types"。如果改成這樣：

```
LinkedList iList2 = (LinkedList)in.readObject();
```

這才是既無錯誤又無警告的完美寫法。

```
LinkedList<Integer> iList = new LinkedList<Integer>();
...
ObjectOutputStream out =
    new ObjectOutputStream(
      new FileOutputStream("out"));

out.writeObject(iList);    //寫入
out.close();

ObjectInputStream in =
  new ObjectInputStream(
    new FileInputStream("out"));

//...這裡準備進行讀取動作 readObject()
```

圖 14 / 將元素型別爲 `Integer` 的容器寫入檔案，而後準備讀出。

由以上測試結果可以預期，似乎存在這一事實：當 object 被寫入檔案，即失去其泛型型別參數（如果有的話）。因此讀回的只是「非泛型」的 class 資訊。如果上述最後那個「完美」寫法改成這樣：

```
ArrayList iList2 = (ArrayList)in.readObject();
```

仍可順利編譯，在編譯器的眼裡看來也很完美，但執行期會出現異常，告知「讀入的 class 資訊」和「程式預備接收的 class 資訊」不相符合。異常訊息如下：

```
Exception in thread "main" java.lang.ClassCastException:
```

```
java.util.LinkedList
```

我們可以觀察 serialization 的輸出檔獲得證據。從圖 15 可以看出來，檔案內記錄的class名稱是`java.util.LinkedList`，並一併記錄了元素型別`java.lang.Integer`（及其 base class `java.lang.Number`）。但元素型別其實是針對每一個元素都會記錄的（當然啦，如果遇上相同型別的元素，這些資訊並不會又被傻傻地完整記錄一遍，那太浪費時間和空間，而是只記錄一個 handle，詳見《Java 的物件永續之道》，網址列於文後）。這些記錄對於 deSerialization 過程中恢復容器原型和內容有其必要，但無法讓編譯器推論當初使用的是 `LinkedList` 容器或是`LinkedList<Integer>`容器。

```
000000: AC ED 00 05 73 72 00 14  6A 61 76 61 2E 75 74 69 秒..sr..java.uti
000010: 6C 2E 4C 69 6E 6B 65 64  4C 69 73 74 0C 29 53 5D l.LinkedList.)S]
000020: 4A 60 88 22 03 00 00 78  70 77 04 00 00 00 04 73 J`."...xpw.....s
000030: 72 00 11 6A 61 76 61 2E  6C 61 6E 67 2E 49 6E 74 r..java.lang.Int
000040: 65 67 65 72 12 E2 A0 A4  F7 81 87 38 02 00 01 49 eger.?艾?8...I
000050: 00 05 76 61 6C 75 65 78  72 00 10 6A 61 76 61 2E ..valuexr..java.
000060: 6C 61 6E 67 2E 4E 75 6D  62 65 72 86 AC 95 1D 0B lang.Number  ...
```

圖 15 / 將元素型別爲 `Integer` 的容器寫入檔案，而後準備讀出。

Java 擦拭法 vs. C++膨脹法

爲什麼 Java 容器的參數化型別無法永續存在於檔案（或其他資料流）內？本文一開始已經說過，這些容器被設計用來存放 `Object`-derived 元素，而 Java 擁有單根繼承體系，所有 Java classes 都繼承自 `java.lang.Object`，因此任何 Java objects 都可以被放進各種容器，換句話說 Java 容器本來就是一種「泛型」的異質容器。今天加上參數化型別反而是把它「窄化」了。「泛型」之於 Java，只是一個角括號面具（當然這個面具帶給了程式開發過程某些好處）；摘下面具，原貌即足夠應付一切。因此 Java 使用所謂「擦拭法」來對待角括號內的參數化型別，如圖 16a。下面是「擦拭法」的四大要點（另有其他枝節，本文不談）：

- 一個參數化型別經過擦拭後應該去除參數（於是 *List<T>* 被擦拭成爲 *List*）
- 一個未被參數化的型別經過擦拭後應該獲得型別本身（於是 *Byte* 被擦拭成爲 *Byte*）
- 一個型別參數經過擦拭後的結果爲 *Object*（於是 *T* 被擦拭後變成 *Object*）

- 如果某個 method call 的回傳型別是個型別參數，編譯器會爲它安插適當的轉型動作。

這種觀念和 C++的泛型容器完全不同。C++容器是以同一套程式碼，由編譯器根據其被使用時所被指定的「不同的參數化型別」建立出不同版本。換句話說一份 template（範本、模板）被膨脹爲多份程式碼，如圖 16b。

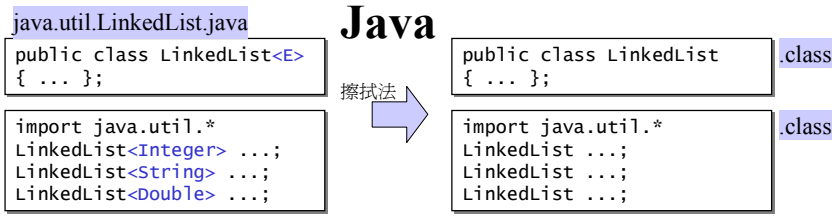


圖 16a / Java 以擦拭法成就泛型

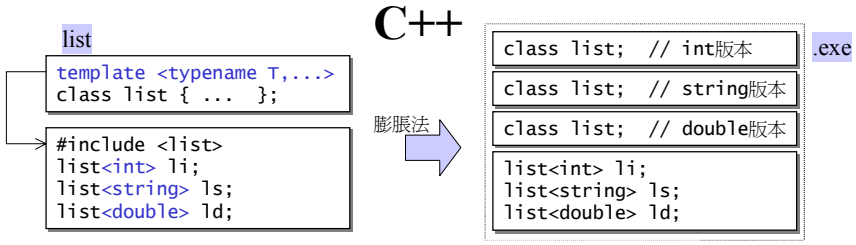


圖 16b / C++以膨脹法成就泛型

擦拭法帶來的遺憾

先前談到的 "Shape" 多型實例（圖 2a），其中的 class `Rect`：

```
public class Rect<T> extends Shape
                    implements Serializable {
  T m_left, m_top, m_width, m_height;
  public Rect(T left, T top, T width, T height ) { ... }
  ...
}
```

經過擦拭後變成了：

```
public class Rect extends Shape
```

```
                    implements Serializable {
  Object m_left, m_top, m_width, m_height;
  public Rect(Object left, Object top, Object width, Object height )
{ ... }
  ...
}
```

這麼一來，任何數值運算，例如先前提過的「周長計算」L()將無法編譯，如圖 17：

```
//class Rect<T> 內
public double L() {
   return (double)((m_width + m_height) * 2);
}
```

圖 17 / L()發生錯誤

錯誤訊息是："operator + cannot be applied to T,T"。是的，兩個 Object object 如何相加呢？Java 並沒有提供像 C++ 那樣的運算子重載（operator overloading）功能！可以說，圖 2a 的 Shape 繼承體系只是介面正確，一旦面臨某些情況，卻無實用性。我的結論是，將參數化型別用於 Java non-collection classes 身上，恐怕會面臨許多束縛。（註：讀者來函提供了此問題的一個解答，見本文末尾添加之補充）

更多資訊

以下是與本文主題相關的更多討論。這些資訊可以彌補本文篇幅限制而造成的不足，並帶給您更多視野。

- 《Java 泛型技術之發展》,by 侯捷。http://www.jjhou.com/javatwo-2002-generics-in-jdk14.pdf
- 《Java 的物件永續之道》,by 侯捷。http://www.jjhou.com/javatwo-2003-serialization-doc.pdf

■補充

讀者 AutoWay 針對無法計算周長這個問題，來信如下：

> From: AutoWay
>
> Sent: Monday, January 17, 2005 7:57 PM
>
> Subject: 《JDK 1.5 的泛型實現》讀者回應
>
> 侯捷兄：隨函寄上 Rect.java 程式之修訂，俾可以進行「周長計算」。修訂內容如下：在設定 type parameter 時，同時宣告其 type bound，例如本例修改爲 `<T extends Number>`。系統進行編譯時，就知道 `T` 是 `Number` 或其 subclass；因此內部程式就可運用 `Number` 提供的 methods 進行運算了。
>
> 我想，這就是 type parameters 之所以提供 type bounds 機制的主要原因。如果 type bounds 光用來限制 type arguments 之傳遞，實在沒啥意思！感謝本文揭示的例子，讓我對 type bounds 有進一步的省思與認識；若有謬誤，亦請來信指教。

侯捷回覆：非常感謝 AutoWay 兄的指正，解除了我的盲點。整理於下。圖 6 之程式碼應改爲：

```
public class Rect<T extends Number> extends Shape
                    implements Serializable {
  ...
}
```

```
public class Circle<T extends Number> extends Shape
                      implements Serializable {
  ...
}
```

```
public class Stroke<W extends Number,T extends Number> extends Shape
                         implements Serializable {
  ...
}
```

圖 17 程式碼應改爲：

```
//class Rect<T extends Number> 內
public double L() {
   return (m_width.doubleValue() + m_height.doubleValue()) * 2;
}
```

另兩個 classes（`Circle` 和 `Stroke`）同理修改。

侯捷觀點