侯捷觀點

千秋偉業 物件永續

Java 的物件永續之道

— Object Persistence in Java —

北京《程序員》2003/11
台北《Run!PC》2003/11

作者簡介：侯捷，資訊工作者、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　jjhou@ccca.nctu.edu.tw

- 讀者基礎：有 Java 語言基礎，用過 Java Collections 及其 serialization 功能。
- 本文適用工具：JDK1.2+
- 本文程式源碼可至侯捷網站下載
- 本文同時也是 JavaTwo-2003 技術研討會之同名講題的部分內容書面整理。
- 關鍵術語：
  persistence（永續性、持久性）
  serialization（序列化、次第讀寫）
  reflection（反射）

全文提要

只要 objects 必須儲存於磁碟或傳輸於網絡，就需要用到 Serialization 技術。Java serialization 機制包裝得非常好，程式員只需 "implements Serializable"，幾乎不必再付出其他努力，便可充份享用 object persistence 能力。本文告訴您 "implements Serializable" 背後的故事，帶您認識 Serialization 的技術關鍵以及 Java 對它的實現手法，並具體讓您知道 object 被儲存起來的模樣。同時也告訴您如果不使用預設機制而自行實現 serialization，會帶來什麼隱微影響。

有時候我會在網絡論壇上看到「探索軟體的永恆之道」這一類題目。永恆，呵呵，多麼令人肅然起敬的字眼。以我的歷練，無法體會軟體或計算機技術爲什麼能夠和永恆扯上關係，那和「愛你億萬年」一樣地空洞。我常笑謂『什麼是永恆？塑膠即永恆』，所以戀人表達永恆愛意時互贈塑膠花即可，不必贈鑽石。

然而，在一個語不驚人死不休的話語時代中，我也樂意聲稱「物件永續，千秋萬載」。本文談永續，純粹從技術角度出發，和哲學的永恆大不相同。永續一詞雖然頗具學術味，說穿了其實是指「電源關閉後，計算機內的資料繼續存在」。因此就技術而言，意味（在現今技術水平下）如何將 objects 的完整狀態儲存起來，以及如何根據被儲存的資料重建原本的 objects（並恢復其原本狀態）。

任何一名程式員如果不知道如何存取檔案，沒資格說自己會編寫程式。既然存取檔案不成問題，objects 的永續又何問題之有？硬編硬幹的 "dirty work" 誰都會，問題在於高階化、復用化、彈性化。

讓我舉個例子。假設你有兩個繁複的資料結構，就說 tree 和 hashtable 好了，如果能夠這麼就把它們全寫入檔案中：

```
// 資料寫至儲存裝置
myFileStream << myContainer1;   // myContainer1 是個 tree.
myFileStream << myContainer2;   // myContainer2 是個 hash table.
```

下次執行時這麼就把它們完整讀取回來：

```
// 資料讀自儲存裝置
```

```
myFileStream >> myContainer1;
myFileStream >> myContainer2;
```

斯可謂高階化、復用化、彈性化。這就是本文所要探討的目標。上例用到的運算子重載（operator overloading）是 C++ 關鍵特性之一，Java 並不提供這種性質，但如果可以在 Java 中這麼做：

```
// 資料寫至儲存裝置
myFileStream.writeObject(myContainer1);
myFileStream.writeObject(myContainer2);
```

下次執行時再這麼就把它們完整讀取回來：

```
// 資料讀自儲存裝置
myContainer1 = (Type of myContainer1)myFileStream.readObject();
myContainer2 = (Type of myContainer2)myFileStream.readObject();
```

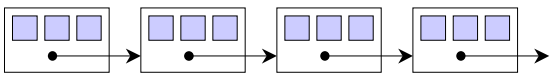
意思是一樣的。

Serialization 概觀

Persistence（永續）是一種概念。**Serialization**（序列化、次第讀寫）則是 Java Library 和 MFC（以及其他可能平台）上的一種具體實作技術。本文談的是 **Serialization** 在 Java 上的實現。Serialization 在 MFC 中泛指物件的輸出和讀取，Java 則把物件的讀取另稱爲 **deSerialization**。

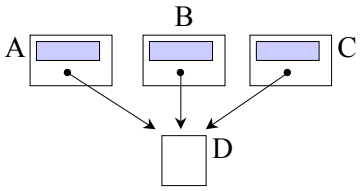
一個 object 的背後可能帶有一整個網絡，稱爲 a graph of object 或 a web of object。Java serialization 允許程式員將 object 內容一骨腦兒全寫出去而後又讀進來。這項技術的目標是爲這一整套「物件網絡」產生一連串的連續性表述（a serialized reprsentation），成爲一連串帶有次序性的 bytes（a sequence of bytes），並於日後完整回復，重新生成原物件。

所以，Serialization 是一種深層拷貝（deep copy），不僅儲存了物件的映像（image），還儲存其中所指涉（引用, *refer*）的所有物件，以及那些物件所指涉（引用, *refer*）的所有物件…。例如，你有一個 linked list 如下：

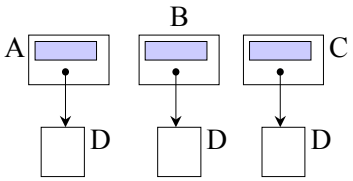


只要寫（或讀）這個 linked list（的頭），整個「物件網絡」便會全部被寫（或讀）。

這就遇上了一個麻煩的問題：如果物件之內出現 **reference semantices**（所謂「指涉語意」或「參考語意」或「引用語意」），該如何回復其原始狀態？例如：



如果 A,B,C 物件都有個 reference 指向同一個 D 物件，當你將這樣一整套 objects 寫入檔案，再回復它時，A,B,C 是否還是指向同一個 D 物件？抑或會變成這樣：



答案是當然應該回復原始情況。看似簡單並且理所當然，但這其實是一個很不容易的技術，需要繁複的演算法。Java 和 MFC 對此都有正確的表現，我們可以在 MFC 源碼中看到其實作上的複雜。

Java 程式庫中與 Serialization 相關的三個 packages 是：(1) **java.io**、(2) **java.util**、(3) **java.lang.reflect**，它們也是本文挖掘源碼的對象。其中 java.io 是物件輸入與輸出的最重要技術所在；java.util 本身與 Serialization 技術無關，而是大量運用了這項技術。java.lang.reflect 所支援的反射機制則是 deSerialization 的重要技術支撐。

簡介 Java Serialization 構件（constructs）

構件（constructs）一詞最近數年才大量見於語言書籍之中，意思是程式語言中用以支援某種特性（features）的一些關鍵字或設施（facility）。通常是指語言層次的東西，不過既然很多人都把 Java 程式庫視爲 Java 語言的一部分，以下我就將 Java 程式庫所提供的 APIs 也視爲一種構件。

與 Serialization 相關的 Java 構件包括：

- **Serializable** 介面。任何 class 只要實作它，就享有 default Serialization 的能力。所謂 "default" 意指它所寫出（和讀取）的是預設型式，簡言之就是把物件中的所有資料一字不漏地寫出和讀入。對程式員而言，這是通往 Serialization 最輕鬆的途徑了。但是如果你對安全性以及未來的彈性有更多考量，這條路對你就可能不很適合。
- 關鍵字 **transient**。上面說過，如果採用 default 模式，則物件內的所有資料（即使是 private）都會被寫出去。如果你希望某些資料不要被寫出（寫出去就可能受到駭客的侵犯），不要成爲「永續」，那就讓它成爲「暫態」吧，只要在宣告式中加上關鍵字 transient 即可。當然，暫態資料不被寫出，也就無法被讀取，所以你必須在創建物件時自行填寫暫態資料的內容。
- 函式 **readObject()**, **writeObject()**。這兩個 API 函式可讓客戶端自行決定物件的寫出和讀取動作。任何 classes 只要提供這兩個函式，Serialization 機制便會喚起它們。

這些構件將於文後詳細討論。除此之外還有其他 Serialization 構件，可以做更多更細微的操控，例如 `Externalizable`, `WriteExternal()`, `ReadExternal()`，限於篇幅本文並不討論它們。

實例示範 Java Serialization

爲了示範 Serialization 的用法、驗證其成果、並將輸出所得的資料做爲稍後分析之用，我設計了一個實例，採用 Java Collections 中的 `LinkedList` 和 `ArrayList` 容器，放置一些資料如圖 1。Java 物件其實都是 references，圖中以 pointer 的方式來畫它們，只是爲了方便。程式碼中我使用泛型表示法（角括號 `<>`）來表現容器的元素型別。不這麼做也沒有關係，但這可使程式的可讀性更高一些，轉型動作也少一些。Java 的泛型討論請參考本刊 2002/08《*Generics in Java*》一文。

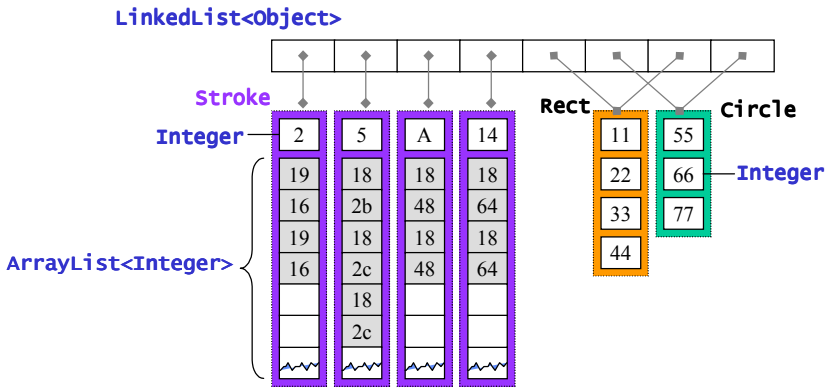


圖 1 / 以 Java Collections 所提供的容器安置若干資料

範例程式說明－：main()

讓我簡單介紹圖 1 整個物件的內容。這個資料結構模擬繪圖軟體的圖形儲存方式：以一個 `LinkedList<Object>` 放置 8 個 `Object` 物件，其中 4 個爲 `Stroke` 物件，每個 `Stroke` 物件內含線條粗細（整數）以及組成線條的所有點座標（x,y;x,y;…，以 `ArrayList<int>` 表現）：

```
import java.util.*;
import java.io.*;

    LinkedList<Object> sl = new LinkedList<Object>();

    ArrayList<Integer> ia = new ArrayList<Integer>();
    ia.add(new Integer(0x19));    ia.add(new Integer(0x16));
    ia.add(new Integer(0x19));    ia.add(new Integer(0x16));
    sl.add(new Stroke(new Integer(2), ia));
    //…依此類推，添加共 4 個 Stroke objects
```

隨後又放置 1 個 `Rect` 物件和 1 個 `Circle` 物件，各有自己的資料（左上寬高、圓心半徑等等）：

```
    Rect r = new Rect(new Integer(0x11),
                      new Integer(0x22),
                      new Integer(0x33),
                      new Integer(0x44));
    Circle c = new Circle(new Integer(0x55),
                          new Integer(0x66),
                          new Integer(0x77));
    sl.add(r);    sl.add(c);
```

最後，刻意讓 LinkedList 的最後兩個元素分別指向（代表）先前的 Rect 物件和 Circle 物件，用以表現出 reference semantics（而非 value semantics）：

```
    sl.add(r);    sl.add(c);
    System.out.println(sl);
/*
請注意，如果在 C++ 標準程式庫中，由於其所提供的容器只允許 "value semantics"，
因此上述作法會導致 list 中出現內容相同的兩份 r 物件和內容相同的兩份 c 物件，而
不是兩個元素共同指向同一份物件。
*/
```

如此所得的結果是（以下排列經過刻意整理）：

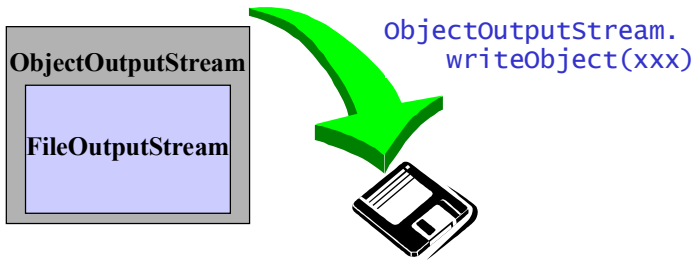
```
/*
[[width=2,points=[25, 22, 25, 22]],
 [width=5,points=[24, 43, 24, 44, 24, 44]],
 [width=10,points=[24, 72, 24, 72]],
 [width=20,points=[24, 100, 24, 100]],
 [L=17,T=34,W=51,H=68], [X=85,Y=102,R=119],
 [L=17,T=34,W=51,H=68], [X=85,Y=102,R=119]]
*/
```

現在將整個物件輸至檔案：

```
    ObjectOutputStream out =
         new ObjectOutputStream(
           new FileOutputStream("collect.out"));

    out.writeObject(sl);
    out.close();  // also flush output stream
```

上述動作的意思是，先將檔案 collect.out 包裝爲一個低階的 FileOutputStream，再以此爲基礎包裝爲一個高階的 ObjectOutputStream，而後利用後者的 writeObject()函式將物件輸出。圖示如下：

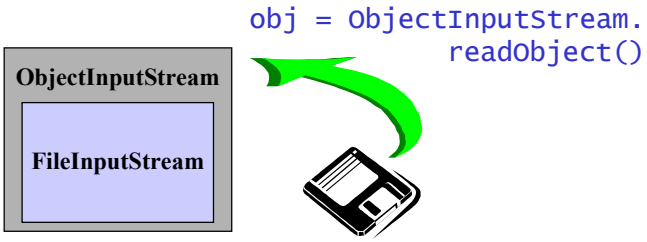


接下來再將這個檔案包裝爲資料串流後讀入，並再次列印出來：

```
ObjectInputStream in =
  new ObjectInputStream(
    new FileInputStream("collect.out"));

LinkedList sl2  = (LinkedList)in.readObject();
in.close();
System.out.println(sl2);
```

動作圖示如下：



所得結果和先前完全相同，可見 Java Serialization 是一種深層拷貝（deep copy），而且原物件的確被重新建構且回復原狀態。

但是，且慢，我們怎麼知道重建出來的物件，不是如圖 2 呢？畢竟圖 2 的列印結果和圖 1 的列印結果相同：

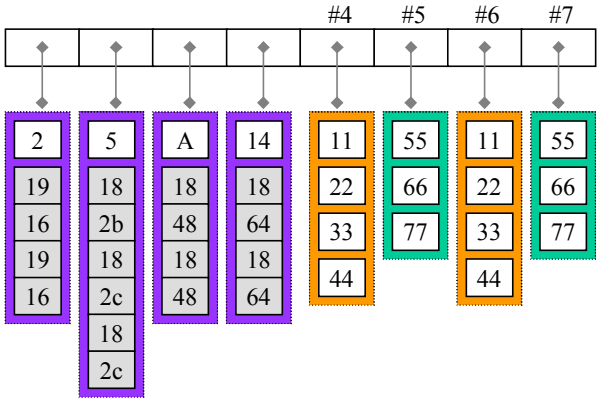


圖 2 / 圖 1 的物件經過 Serialization/deSerialization 之後，會不會變成這樣呢？

要證明確實回復了 reference semantics，一個簡單的辦法就是修改#4 和#5 的元素內容，看看#6 和#7 有沒有隨之更動。實驗證明，#4 和#6 元素以及#5 和#7 元素的確

是彼此連動的。

範例程式說明二：classes 的對應設計

如何才能讓上述程式獲得那樣的 serialization 能力？這個程式用到了 `LinkedList`, `ArrayList`, `Stroke`, `Rect`, `Circle`, `Integer` 等 classes，讓我們看看各個 classes 的宣告和定義，如圖 3。

```
public class Stroke implements Serializable
{
  Integer m_width;
  ArrayList<Integer> m_ia;

  ...//ctor()
  public String toString() {
    return "[width=" + m_width +
      ",points=" + m_ia + "]";
  }
}

public class Rect implements Serializable
{
  Integer m_left, m_top, m_width, m_height;

  ...//ctor()
  public String toString() {
    return "[L=" + m_left + ",T=" + m_top +
          ",W=" + m_width + ",H=" + m_height + "]";
  }
}

public class Circle implements Serializable
{
  Integer m_x, m_y, m_r;

  ...//ctor()
  public String toString() {
    return "[X=" + m_x + ",Y=" + m_y + ",R=" + m_r + "]";
  }
}
//上述三個 classes 都實作出 toString()，這才能使 System.out.println()
//起作用。
```

圖 3 / 三個客戶端設計的 classes

這三個 classes 惟一和 Serialization 有關連的就是出現在每一個 class 名稱之後的 "implements Serializable"。這將使得這些 classes 全部採用預設型式來完成 Serialization。稍後當我們觀察輸出檔的內容時，可以看到這些 classes 的所有資料成員都被輸出到檔案去（這正是預設型式）。

上述 classes 所做出來的物件被放在各種容器之中，因此是否容器也有相似的設計呢？的確如此，請見圖 4。

```
public class LinkedList extends AbstractSequentialList
                        implements List, Cloneable,
                                   java.io.Serializable;
public class ArrayList extends AbstractList
                       implements List, RandomAccess, Cloneable,
                                  java.io.Serializable;
```

圖 4 / `LinkedList` 和 `ArrayList` 的宣告

此外，本例用到了 `Integer` class，它也有類似的設計，見圖 5。

```
public final class Integer extends Number implements Comparable {
    ...
    private int value;
}
public abstract class Number implements java.io.Serializable {
   ...
   private static final long serialVersionUID =
                                          -8742448824652078965L;
}
```

圖 5 / `Integer` 和 `Number` 的宣告

Java Serialization 格式實例分析

以二進制傾印工具（我用的是 TDUMP）傾印本例所得的輸出檔 "collect.out，結果如圖 6。以下將分析圖 6 的數據。我的用意不是要完全搞清楚每一個 byte 代表的意義（如果打算做爲這方面的駭客，也許就有必要），而是希望從先前各 classes 的資料成員及其對 Serialization 的實作手法中，映照這一份傾印內容，從中獲得 Serialization 儲存格式的一個梗概認識。

```
Turbo Dump  Version 5.0.16.12 Copyright (c) 1988, 2000 Inprise Corporation
                    Display of File COLLECT.OUT

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 08 73 J`."...xpw.....s
000030: 72 00 06 53 74 72 6F 6B  65 F2 D8 79 18 90 CC 00 r..Stroke蔥y.  .
000040: C1 02 00 02 4C 00 04 6D  5F 69 61 74 00 15 4C 6A ....L..m_iat..Lj
000050: 61 76 61 2F 75 74 69 6C  2F 41 72 72 61 79 4C 69 ava/util/ArrayLi
000060: 73 74 3B 4C 00 07 6D 5F  77 69 64 74 68 74 00 13 st;L..m_widtht..
000070: 4C 6A 61 76 61 2F 6C 61  6E 67 2F 49 6E 74 65 67 Ljava/lang/Integ
000080: 65 72 3B 78 70 73 72 00  13 6A 61 76 61 2E 75 74 er;xpsr..java.ut
000090: 69 6C 2E 41 72 72 61 79  4C 69 73 74 78 81 D2 1D il.ArrayListx  .
0000A0: 99 C7 61 9D 03 00 01 49  00 04 73 69 7A 65 78 70   a....I..sizexp
0000B0: 00 00 00 04 77 04 00 00  00 0A 73 72 00 11 6A 61 ....w.....sr..ja
0000C0: 76 61 2E 6C 61 6E 67 2E  49 6E 74 65 67 65 72 12 va.lang.Integer.
0000D0: E2 A0 A4 F7 81 87 38 02  00 01 49 00 05 76 61 6C ?父?8...I..val
0000E0: 75 65 78 72 00 10 6A 61  76 61 2E 6C 61 6E 67 2E uexr..java.lang.
0000F0: 4E 75 6D 62 65 72 86 AC  95 1D 0B 94 E0 8B 02 00 Number  ...  ...
000100: 00 78 70 00 00 00 19 73  71 00 7E 00 08 00 00 00 .xp....sq.~.....
000110: 16 73 71 00 7E 00 08 00  00 00 19 73 71 00 7E 00 .sq.~......sq.~.
000120: 08 00 00 00 16 78 73 71  00 7E 00 08 00 00 00 02 .....xsq.~......
000130: 73 71 00 7E 00 02 73 71  00 7E 00 06 00 00 00 06 sq.~..sq.~......
000140: 77 04 00 00 00 0A 73 71  00 7E 00 08 00 00 00 18 w.....sq.~......
000150: 73 71 00 7E 00 08 00 00  00 2B 73 71 00 7E 00 08 sq.~.....+sq.~..
000160: 00 00 00 18 73 71 00 7E  00 08 00 00 00 2C 73 71 ....sq.~.....,sq
000170: 00 7E 00 08 00 00 00 18  73 71 00 7E 00 08 00 00 .~......sq.~....
000180: 00 2C 78 73 71 00 7E 00  08 00 00 00 05 73 71 00 .,xsq.~......sq.
000190: 7E 00 02 73 71 00 7E 00  06 00 00 00 04 77 04 00 ~..sq.~......w..

0001A0: 00 00 0A 73 71 00 7E 00  08 00 00 00 18 73 71 00 ...sq.~......sq.
0001B0: 7E 00 08 00 00 00 48 73  71 00 7E 00 08 00 00 00 ~.....Hsq.~.....
0001C0: 18 73 71 00 7E 00 08 00  00 00 48 78 73 71 00 7E .sq.~.....Hxsq.~
0001D0: 00 08 00 00 00 0A 73 71  00 7E 00 02 73 71 00 7E ......sq.~..sq.~
0001E0: 00 06 00 00 00 04 77 04  00 00 00 0A 73 71 00 7E ......w.....sq.~
0001F0: 00 08 00 00 00 18 73 71  00 7E 00 08 00 00 00 64 ......sq.~.....d
000200: 73 71 00 7E 00 08 00 00  00 18 73 71 00 7E 00 08 sq.~......sq.~..
000210: 00 00 00 64 78 73 71 00  7E 00 08 00 00 00 14 73 ...dxsq.~......s
000220: 72 00 04 52 65 63 74 85  E1 2D 3E 3D A4 30 AA 02 r..Rect  ->=.0..
000230: 00 04 4C 00 08 6D 5F 68  65 69 67 68 74 71 00 7E ..L..m_heightq.~
000240: 00 04 4C 00 06 6D 5F 6C  65 66 74 71 00 7E 00 04 ..L..m_leftq.~..
000250: 4C 00 05 6D 5F 74 6F 70  71 00 7E 00 04 4C 00 07 L..m_topq.~..L..
000260: 6D 5F 77 69 64 74 68 71  00 7E 00 04 78 70 73 71 m_widthq.~..xpsq
000270: 00 7E 00 08 00 00 00 44  73 71 00 7E 00 08 00 00 .~.....Dsq.~....
000280: 00 11 73 71 00 7E 00 08  00 00 00 22 73 71 00 7E ..sq.~......"sq.~
000290: 00 08 00 00 00 33 73 72  00 06 43 69 72 63 6C 65 .....3sr..Circle
0002A0: C3 76 A9 B7 F7 E4 F7 FF  02 00 03 4C 00 03 6D 5F 瀚弧瀾.....L..m_
0002B0: 72 71 00 7E 00 04 4C 00  03 6D 5F 78 71 00 7E 00 rq.~..L..m_xq.~.
0002C0: 04 4C 00 03 6D 5F 79 71  00 7E 00 04 78 70 73 71 .L..m_yq.~..xpsq
0002D0: 00 7E 00 08 00 00 00 77  73 71 00 7E 00 08 00 00 .~.....wsq.~....
0002E0: 00 55 73 71 00 7E 00 08  00 00 00 66 71 00 7E 00 .Usq.~.....fq.~.
0002F0: 27 71 00 7E 00 2D 78 00  00 00 00 00 00 00 00 00 'q.~.-x.........
```

圖 6 / 本文程式例之 Serialization 檔案傾印結果，可用來剖析 Serialization 格式

正式剖析格式之前，這裡先提綱挈領地點出最後結論：Serialization 的格式要領是「遞迴而深度優先地處理每一個 object 及其內的 referenced objects；如遇先前已記錄之 type 或 object，便只儲存其 handle」。當 Serialization 機制遇上本例，其記錄次序如圖 7。圖中所謂的 type 資訊是指能夠表現 type 的所有信息，可以想像將包括 class 名稱、資料成員的個數、型別和名稱。

表格 1-16 便是圖 6 的整個二進制傾印碼的剖析。其中標示 N/A（Not Analysis）的欄位並非無法分析，只要深研 Sun 公司的 Serialization Spec.，沒有解不開的謎題。這裡之所以沒有分析它們，一方面是篇幅因素，一方面因爲它們並不是我的觀察重點。

這些完整的格式分析，乍見之下雖然十分瑣屑，但不實際走過這麼一遭，我們很難對於 Java Serialization 機制有了然於胸的體會。

- 首先面對 LinkedList。於是記錄其 type 資訊，然後記錄 LinkedList 物件內容…
  - 前 4 個元素都是 Stroke 物件，於是逐一…
    - 記錄 Stroke 的 type 資訊（如已有則只記錄 handle），然後記錄其內容…
      - Stroke 內容是一個 Integer 物件和一個 ArrayList 物件，於是…
        - 記錄 Integer 的 type 資訊，然後記錄其內容
        - 記錄 ArrayList 的 type 資訊，然後記錄其內容…
          - ArrayList 的內容是數個 Integer 物件，於是逐一…
            - 記錄 Integer 的 type 資訊。此前已記錄過，所以只記錄 handle
            - 記錄 Integer 的內容
  - LinkedList 的第 5 個元素是 Rect 物件，於是…
    - 記錄 Rect 的 type 資訊，然後記錄其內容…
      - Rect 的內容是 4 個 Integer 物件，於是逐一…
        - 記錄 Integer 的 type 資訊。此前已記錄過，所以只記錄 handle
        - 記錄 Integer 的內容
  - LinkedList 的第 6 個元素是 Circle 物件，於是…
    - 記錄 Circle 的 type 資訊，然後記錄其內容…
      - Circle 的內容是 3 個 Integer 物件，於是逐一…
        - 記錄 Integer 的 type 資訊。此前已記錄過，所以只記錄 handle
        - 記錄 Integer 的內容
  - LinkedList 的第 7 個元素是 Rect 物件，此前其 type 資訊和物件內容都已有記錄（後者乃因 reference semantics 之故），所以兩者都只需記錄 handle。
  - LinkedList 的第 8 個元素是 Circle 物件，此前其 type 資訊和物件內容都已記錄過（後者乃因 reference semantics 之故），所以兩者都只需記錄 handle。

圖 7 / 本文程式例之 Serialization 格式。圖中以縮排表示進一步的 method call。

表 1 / LinkedList 的 type 資訊和部分狀態

二進制碼	意義
AC ED	STREAM_MAGIC
00 05	STREAM_VERSION
73, 72	N/A
00 14 6A 61 76 61 2E 75 74 69 6C 2E 4C 69 6E 6B 65 64 4C 69 73 74	"java.util.LinkedList"
0C 29 53 5D 4A 60 88 22 03	SerialVersionUID(long)+flag
00 00	後有 0 筆 data member 記錄
78,70,77,04	N/A
00 00 00 08	共有 8 個元素

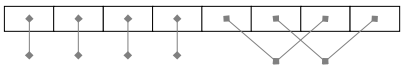


表 2 / 第一個 Stroke 物件的 type 資訊和部分狀態

二進制碼	意義
73, 72	N/A
00 06 53 74 72 6F 6B 65	"Stroke"
F2 D8 79 18 90 CC 00 C1 02	SerialVersionUID(long)+flag
00 02	後有 2 筆 data members 記錄
4C	'L'，代表 class 或 interface
00 04 6D 5F 69 61	"m_ia"
74	TC_STRING
00 15 4C 6A 61 76 61 2F 75 74 69 6C 2F 41 72 72 61 79 4C 69 73 74 3B	"Ljava/util/ArrayList;"
4C	'L'，代表 class 或 interface
00 07 6D 5F 77 69 64 74 68	"m_width"
74	TC_STRING
00 13 4C 6A 61 76 61 2F 6C 61 6E 67 2F 49 6E 74 65 67 65 72 3B	"Ljava/lang/Integer;"
78,70	N/A

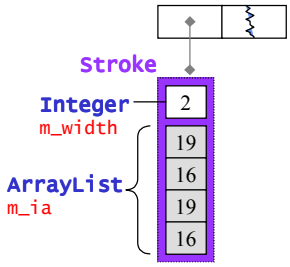


表 3 / ArrayList 的 type 資訊和部分狀態

二進制碼	意義
73, 72	N/A
00 13 6A 61 76 61 2E 75 74 69 6C 2E 41 72 72 61 79 4C 69 73 74	"java.util.ArrayList"
78 81 D2 1D 99 C7 61 9D 03	SerialVersionUID(long)+flag
00 01	後有 1 筆 data member 記錄
49	'I'，代表 int
00 04 73 69 7A 65	"size"
78,70	N/A
00 00 00 04	共有 4 個元素
77,04	N/A
00 00 00 0A	array 的目前長度（容量）

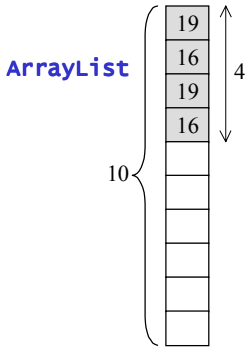


表 4 / ArrayList 的第一個元素的 type 資訊和狀態

二進制碼	意義
73, 72	N/A
00 11 6A 61 76 61 2E 6C 61 6E 67 2E 49 6E 74 65 67 65 72	"java.lang.Integer"
12 E2 A0 A4 F7 81 87 38 02	SerialVersionUID(long)+flag
00 01	後有 1 筆 data member 記錄
49	'I'，代表 int
00 05 76 61 6C 75 65	"value"
78,72	N/A
00 10 6A 61 76 61 2E 6C 61 6E 67 2E 4E 75 6D 62 65 72	"java.lang.Number"
86 AC 95 1D 0B 94 E0 8B 02	SerialVersionUID(long)+flag
00 00	後有 0 筆 data member 記錄
78,70	N/A
00 00 00 **19**	Integer 數值

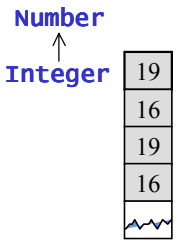


表 5 / Stroke 物件的其他狀態（包括其他點座標及筆寬）

二進制碼	意義
73, 71	N/A
00 7E 00 08	handle (for Integer type)
00 00 00 **16**	ArrayList 的第二個元素值
73, 71	N/A
00 7E 00 08	handle (for Integer type)
00 00 00 **19**	ArrayList 的第三個元素值
73, 71	N/A
00 7E 00 08	handle (for Integer type)
00 00 00 **16**	ArrayList 的第四個元素值
78, 73, 71	N/A
00 7E 00 08	handle (for Integer type)
00 00 00 **02**	筆寬

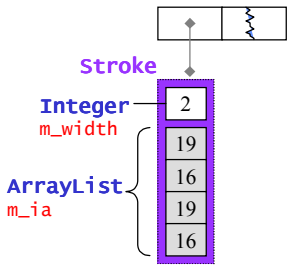


表 6 / 第二個 Stroke 物件的 type 資訊

二進制碼	意義
73, 71	N/A
00 7E 00 02	handle (for Stroke type)
73, 71	N/A
00 7E 00 06	handle (for ArrayList type)

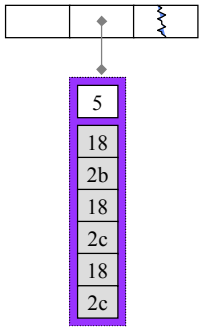


表 7 / 第二個 Stroke 物件的內容（狀態）

二進制碼	意義
00 00 00 06	ArrayList 共有 6 個元素
77 04	N/A
00 00 00 0A	array 的目前長度（容量）
73, 71	N/A
00 7E 00 08	handle (for Integer type)
00 00 00 **18**	ArrayList 的第一個元素值
73, 71	N/A
00 7E 00 08	handle (for Integer type)
00 00 00 **2B**	ArrayList 的第二個元素值
73, 71	N/A
00 7E 00 08	handle (for Integer type)
00 00 00 **18**	ArrayList 的第三個元素值
73, 71	N/A
00 7E 00 08	handle (for Integer type)
00 00 00 **2C**	ArrayList 的第四個元素值
73, 71	N/A
00 7E 00 08	handle (for Integer type)
00 00 00 **18**	ArrayList 的第五個元素值
73, 71	N/A
00 7E 00 08	handle (for Integer type)
00 00 00 **2C**	ArrayList 的第六個元素值
78, 73, 71	N/A
00 7E 00 08	handle (for Integer type)
00 00 00 **05**	筆寬（十進制 5）

表 8 / 第三個 Stroke 物件的 type 資訊

二進制碼	意義
73, 71	N/A
00 7E 00 02	handle (for Stroke type)
73, 71	N/A
00 7E 00 06	handle (for ArrayList type)

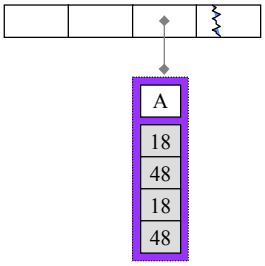


表 9 / 第三個 Stroke 物件的內容（狀態）

二進制碼	意義
00 00 00 04	ArrayList 共有 4 個元素
77 04	N/A
00 00 00 0A	array 的目前長度（容量）
73, 71	N/A
00 7E 00 08	handle (for Integer type)
00 00 00 **18**	ArrayList 的第一個元素值
73, 71	N/A
00 7E 00 08	handle (for Integer type)
00 00 00 **48**	ArrayList 的第二個元素值
73, 71	N/A
00 7E 00 08	handle (for Integer type)
00 00 00 **18**	ArrayList 的第三個元素值
73, 71	N/A
00 7E 00 08	handle (for Integer type)
00 00 00 **48**	ArrayList 的第四個元素值
78, 73, 71	N/A
00 7E 00 08	handle (for Integer type)
00 00 00 **0A**	筆寬（十進制 10）

表 10 / 第四個 Stroke 物件的 type 資訊

二進制碼	意義
73, 71	N/A
00 7E 00 02	handle (for Stroke type)
73, 71	N/A
00 7E 00 06	handle (for ArrayList type)

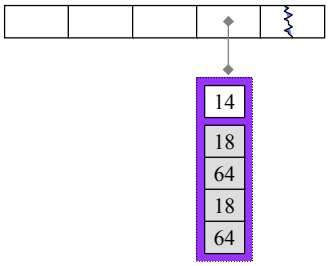


表 11 / 第四個 Stroke 物件的內容（狀態）

二進制碼	意義
00 00 00 04	ArrayList 共有 4 個元素
77 04	N/A
00 00 00 0A	array 的目前長度（容量）
73, 71	N/A
00 7E 00 08	handle (for Integer type)
00 00 00 **18**	ArrayList 的第一個元素值
73, 71	N/A
00 7E 00 08	handle (for Integer type)
00 00 00 **64**	ArrayList 的第二個元素值
73, 71	N/A
00 7E 00 08	handle (for Integer type)
00 00 00 **18**	ArrayList 的第三個元素值
73, 71	N/A
00 7E 00 08	handle (for Integer type)
00 00 00 **64**	ArrayList 的第四個元素值
78, 73, 71	N/A
00 7E 00 08	handle (for Integer type)
00 00 00 **14**	筆寬

表 12 / 第五個 LinkedList 元素（Rect）的 type 資訊

二進制碼	意義
73, 72	N/A
00 04 52 65 63 74	"Rect"
85 E1 2D 3E 3D A4 30 AA 02	SerialVersionUID(long)+flag
00 04	後有 4 筆 data member 記錄
4C	N/A
00 08 6D 5F 68 65 69 67 68 74	"m_height"
71 00 7E 00 04 4C	N/A
00 06 6D 5F 6C 65 66 74	"m_left"
71 00 7E 00 04 4C	N/A
00 05 6D 5F 74 6F 70	"m_top"
71 00 7E 00 04 4C	N/A
00 07 6D 5F 77 69 64 74 68	"m_width"
71 00 7E 00 04 78 70	N/A

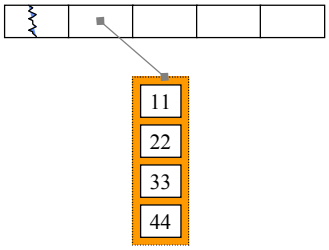


表 13 / 第五個 LinkedList 元素（Rect）的內容（狀態）

二進制碼	意義
73, 71	N/A
00 7E 00 08	handle (for Integer type)
00 00 00 **44**	m_high 數值
73, 71	N/A
00 7E 00 08	handle (for Integer type)
00 00 00 **11**	m_left 數值
73, 71	N/A
00 7E 00 08	handle (for Integer type)
00 00 00 **22**	m_top 數值
73, 71	N/A
00 7E 00 08	handle (for Integer type)
00 00 00 **33**	m_width 數值

表 14 / 第六個 LinkedList 元素（Circle）的 type 資訊

二進制碼	意義
73, 72	N/A
00 06 43 69 72 63 6C 65	"Circle"
C3 76 A9 B7 F7 E4 F7 FF 02	SerialVersionUID(long)+flag
00 03	後有 3 筆 data member 記錄
4C	N/A
00 06 6D 5F 6C 65 66 74	"m_r"
71 00 7E 00 04 4C	N/A
00 03 6D 5F 78	"m_x"
71 00 7E 00 04 4C	N/A
00 03 6D 5F 79	"m_y"
71 00 7E 00 04 78 70	N/A

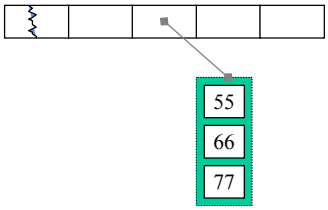
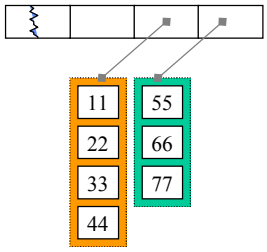


表 15 / 第六個 LinkedList 元素（Circle）的內容（狀態）

二進制碼	意義
73, 71	N/A
00 7E 00 08	handle (for Integer type)
00 00 00 **77**	m_r 數值
73, 71	N/A
00 7E 00 08	handle (for Integer type)
00 00 00 **55**	m_x 數值
73, 71	N/A
00 7E 00 08	handle (for Integer type)
00 00 00 **66**	m_y 數值

表 16 / 第七、第八個 LinkedList 元素（Rect, Circle）的 type 資訊和內容

二進制碼	意義
71	TC_REFERENCE
00 7E 00 27	handle (for Rect object)
71	TC_REFERENCE
00 7E 00 2D	handle (for Circle object)
78	TC_ENDBLOCKDATA



非預設的 Serialization 行為

從表 2、表 12、表 14 可以看出來，Stroke、Rect、Circle 三個 classes 輸出時記錄了每一個資料成員（包括其名稱和型別）。這很直觀易懂。從表 1 和表 3 又可以看出來，LinkedList 的輸出只記錄了當時的元素個數，ArrayList 的輸出只記錄了當時的元素個數和 array 長度（容量）。這兩個容器的設計難道如此簡單嗎？

事實不然。下面是 LinketList 的相關源碼：

```
private transient Entry header = new Entry(null, null, null);
private transient int size = 0;

private synchronized void writeObject(java.io.ObjectOutputStream s)
    throws java.io.IOException
     s.defaultWriteObject(); //這個動作一定要做

     // Write out size
     s.writeInt(size);      //這個動作製造了表 1 的最後一個欄位

     // Write out all elements in the proper order.
     for (Entry e = header.next; e != header; e = e.next)
          s.writeObject(e.element);  //寫出每一個元素
}

private static class Entry {
   Object element;
   Entry next;
   Entry previous;
      ...
}
```

LinkedList 將兩個資料成員設爲暫態（於是 default Serialization 機制不去處理它們），在 writeObject()中才顯式將 size 值寫出去，並令所有元素自己負責將自己寫出去。如果不是這麼做，而是採用 default Serialization，那麼 header 和 size 都會被完整寫出，於是整個雙向鏈結結構都被寫到檔案去，不僅浪費空間，對 LinkedList 未來的彈性也有比較大的傷害（因爲被固定死了）。

下面再看看 ArrayList 的相關源碼：

```
private static final long serialVersionUID = 8683452581122892189L;
private transient Object elementData[];
private int size;
```

```
private synchronized void writeObject(java.io.ObjectOutputStream s)
    throws java.io.IOException{
    s.defaultWriteObject();      //這個動作一定要做（其中會寫出 size）

    // Write out array length
    s.writeInt(elementData.length); //這個動作製造了表 3 的最後一個欄位

    // Write out all elements in the proper order.
    for (int i=0; i<size; i++)
        s.writeObject(elementData[i]); //寫出每一個元素
}
```

其中將 `elementData[]` 設爲暫態（於是 default Serialization 機制不去處理它），而在 `writeObject()`中，首先以 default 型式輸出（本例也就是輸出 `size`；至於 static 成員並不會被輸出），然後才顯式將 array 長度（`length`）寫出去，並令所有元素自己負責將自己寫出去。這麼做同樣是爲了保持格式的彈性：只需寫出最必要的資料（以備將來得以回復狀態）就好了，不必寫出容器的內部結構。換句話說，將來如果 `ArrayList` 的內部結構有變，依然可以讀取目前輸出的檔案（回溯相容），因爲讀入的只是元素個數、array 容量和每個元素的內容，並不涉及內部結構本身。

deSerialization

物件的寫出和讀入最大的不同是，前者只是將 type 資訊（多半就是 class 名稱）和 object 內容寫出去，比較簡單；後者不僅僅是讀取 class 名稱，還得根據它來創建 object（所謂 dynamic creation），然後才能將 object 內容讀進來放到 object 體內。

關鍵技術就在於如何根據一個讀進來的、代表 class 名稱的字串，創建出一個 object。我們知道不論 C++ 或 Java 都以 `new` 來創建物件，`new` 後面緊跟的 class 名稱必須是個字面常數（string literal），不能是個字串變數（string variable）。在 C++ 中，爲了不必將讀入的字串拿來和既有名單逐一比對然後才呼叫 `new`（那將會是很笨拙的動作），於是如 MFC 之流的程式庫便將 `"new XXX"` 函式佈樁在每一個（需此功能之）classes 都必然擁有的一塊結構中。程式執行期間一旦讀入 class 名稱字串，便到結構網中搜尋有無與此名稱吻合的一塊結構，找到的話便可呼叫該 `"new XXX"` 函式。爲了讓上述許多工作自動化（例如自動爲 classes 完成 `"new XXX"` 函式、自動爲 classes 產生上述的結構及結構網），MFC 創造出一些奇特的巨集（像

是 DECLARE_DYNCREATE, IMPLEMENT_DYNCREATE...等等）。

Java 的作法完全不是這樣，省力多了。主要因爲 Java 有一個功能十分驚人的「動態型別系統」：程式所用的 classes 不必在編譯期出現，只需於執行期存在即可。這個系統反映於 Reflection 機制身上。有了它 deSerialization 就好辦多了。

簡言之，Reflection 機制允許你在執行期根據一個 class 名稱，取得其相關的所謂 `Class` object（注意 'C' 大寫）。這個 `Class` object 內含「該 class 名稱所代表之 class」的一切資訊，例如資料成員、成員函式、建構式…等等。有了這個東西，我們可以在任意時刻從一個 class 名稱反求其所有 class 屬性，進而創建 objects。

舉個例子，下面是在執行期間取得 `Stroke`（前例出現過）的所有 class 屬性：

```
import java.lang.reflect.*;

Class c2 = Class.forName("Stroke");
System.out.println(c2);
Method[] md2 = c2.getDeclaredMethods();
Constructor[] ctord2 = c2.getDeclaredConstructors();
Field[] fd2 = c2.getDeclaredFields();
for (int i = 0; i < md2.length; i++)
     System.out.println(md2[i]);
for (int i = 0; i < ctord2.length; i++)
     System.out.println(ctord2[i]);
for (int i = 0; i < fd2.length; i++)
     System.out.println(fd2[i]);

/* output:
class Stroke
public java.lang.String Stroke.toString()
public Stroke(java.lang.Integer,java.util.ArrayList)
java.lang.Integer Stroke.m_width
java.util.ArrayList Stroke.m_ia
*/
```

Stroke 的確有一個 method 和一個 constructor

Stroke 的確有兩個 fields

程式進行 Serialization 時，會用到一個後援 class（客戶端程式不會接觸到）：`java.io.ObjectStreamClass`。它其實只是用來提供「儲存於 stream 內」的 class 的資訊（包括 class 完整名稱及其 serialization version UID），有些文件稱它爲 "class descriptor"。這個 `ObjectStreamClass` 內含許多與 Reflection 相關的欄位，像是：

```
private static final ObjectStreamField[] serialPersistentFields;
private Class cl;
private ObjectStreamField[] fields;
private Constructor cons;
private Method writeObjectMethod;
private Method readObjectMethod;
private Method readObjectNoDataMethod;
private Method writeReplaceMethod;
private Method readResolveMethod;
private ObjectStreamClass localDesc;
private ObjectStreamClass superDesc;
```

其中出現的 `ObjectStreamField`，也是一個後援 class（客戶端程式更不會接觸到它了），其中也有若干與 Reflection 相關的欄位，像是：

```
   /** field name */
    private final String name;
    /** canonical JVM signature of field type */
    private final String signature;
    /** field type (Object.class if unknown non-primitive type) */
    private final Class type;
    /** whether or not to (de)serialize field values as unshared */
    private final boolean unshared;
    /** corresponding reflective field object, if any */
    private final Field field;
    /** offset of field value in enclosing field group */
    private int offset = 0;
```

這便明白顯示 Java deSerialization 和 Reflection 分不開關係。關於 Reflection 的技術討論，需要另一大塊篇幅，也許下次我再來介紹它 ☺

非預設的 deSerialization 行為

如果輸出時做爲客戶端程式員的你設計了自己的 `writeObject()`（稍早曾列出 `LinkedList` 和 `ArrayList` 的作法），讀取時也必須對應地撰寫自己的 `readObject()`。下面是 `LinkedList` 的作法(請對照先前所列之 `writeObject()`)：

```
/**
 * Reconstitute LinkedList instance from a stream
 * (that is deserialize it).
 */
private synchronized void readObject(java.io.ObjectInputStream s)
    throws java.io.IOException, ClassNotFoundException {
    s.defaultReadObject();    //這個動作一定要做
```

```
    int size = s.readInt();      //這個動作讀取了表 1 的最後一個欄位

    // Initialize header
    header = new Entry(null, null, null);
    header.next = header.previous = header;

    // Read in all elements in the proper order.
    for (int i=0; i<size; i++)
        add(s.readObject());     //讀取每一個元素
}
```

下面是 ArrayList 的作法（請對照先前所列的 writeObject()）：

```
/**
* Reconstitute ArrayList instance from a stream
* (that is, deserialize it).
*/
private synchronized void readObject(java.io.ObjectInputStream s)
    throws java.io.IOException, ClassNotFoundException {
     s.defaultReadObject();  //這個動作一定要做（其中會讀取 size）

    // Read in array length and allocate array
    int arrayLength = s.readInt(); //這個動作讀取了表 3 的最後一個欄位
    elementData = new Object[arrayLength];

     // Read in all elements in the proper order.
     for (int i=0; i<size; i++)
         elementData[i] = s.readObject();   //讀取每一個元素
}
```

編程注意事項

本文的主要目的是讓大家對 Serialization 的動作和其格式有所了解。至於編程方面的技術，也就是先前曾提過的 Serialization 相關構件（constructs），並非本文重點。我在這兒只簡單列出一些心得整理，並不全面：

- 必須實現 java.io.Serializable（或 Externalizable，較罕見）
- 確定哪些欄位應當設爲暫態（使用關鍵字 transient）
- 只要是 non-transient, non-static 欄位，都會被 default serialization 機制處理。
- 第一個 non-serializable superclass 的 no-argument constructor（無引數建構式）會被喚起。因此你必須備妥這個東西。
- 選擇性地定義以下函式：writeObject(), readObject(), writeReplace(),

`readResolve()`。一旦 class 實作出這些函式，就有機會在 serializable field values 被寫出之前或被讀取之後加以修改。通常我們會在這些函式之中呼叫 API 函式 `defaultWriteObject()` 和 `defaultReadObject()`，以求喚起 default serialization 機制。

- 請爲 serializable 欄位（fields）提供說明文件。可使用以下三種標籤：`@serial`, `@serialField`, `@serialData`。

《*Effective Java*》第 10 章專門談論 Serialization 的編程技術與應用，很值得參考。

更多資訊

以下是我視野所及，與本文主題相關的更多討論。這些資訊可以彌補因文章篇幅限制而帶來的不足，或帶給你更多視野。

- 《*Java Object Serialization Specification*》by Sun Microsystems Inc.
  這是研究 Java Serialization 的終極文件。當然，規格書往往不易讀 ☺。
- Java source code（以下是我爲研究 Serialization 而研讀的 Java 源碼）
  io.FileOutputStream, io.ObjectOutputStream, io.ObjectStreamClass,
  util.LinkedList, util.ArrayList,
  lang.Integer, lang.Number, lang.reflect.Constructor, lang.reflect.Method...
- 《*Effective Java*》by Joshua Bloch, chap10: *Serialization*.
- 《*Thinking in Java*》by Bruce Eckel, chap11: *Java I/O System*; chap12:*RTTI*.
- 《*Java Collections*》