偶合與內聚力

1. 耦合和內聚力，和 OO 設計的品質是大有關係的，一般來說好的 OO 設計會追求鬆散耦合(loose coupling)而避開緊密耦合，好的 OO 設計會追求高內聚力(high cohesion)而避開低內聚力
2. 希望一個軟體應用程式會達到

   <1>.容易建立

   <2>.容易維護

   <3>.容易加強

在物件導向的設計的實務上，最困難的工作之一就是「如何決定物件的責任以及物件之間的關係」。
我們可以由兩個不同的觀點來解釋這句話：從抽象的觀點來看，物件的責任分配應該易於理解與合理，不讓人困惑，並且讓物件之間的運作盡量不會互相干擾；從實做的觀點來看，讓物件之間相依性降到最低，不會因為特定物件的變更而引起了整個架構的漣漪，確保設計擴充上的彈性。

現代社會由於工作繁忙，所以許多上班族都成了外食族，但是在筆者還小的時候，全家人都會回家吃晚飯。到了晚上，爸爸的責任是在家翹著二郎腿看報紙。哥哥跟我的責任則是好好讀書，長大以後要成大事立大業，光宗耀祖，讓媽媽有面子。媽媽的責任是負責煮菜，晚上全家人吃晚餐時有一頓美味的餐點可以大快朵頤。每個人都有自己要負責的工作，彼此分工合作支撐起這個美滿家庭。
如果爸爸沒有報紙看的話，爸爸一個人會覺得無聊，但是對我們沒有影響。哥哥如果期中考沒考好的話，媽媽會覺得沒面子，但是我跟爸爸覺得沒關係，反正勝敗乃兵家常事。

但是晚上的時候每個人都要「吃飯」，而飯菜是靠媽媽「煮菜」得到的，所以每個人如果要吃飽，要依靠媽媽煮菜這件工作能夠順利完成，有一天媽媽生病了，結果全家人都餓到雙腿發軟，我們全家才體認到一個事實，媽媽能不能好好煮菜這件事情會影響到全家人，所以我們必須不惜一切讓媽媽可以每晚好好的「煮菜」。
要活下去就要吃飯是一件不可違逆的事情，就好像在軟體系統內，希望物件之間完全沒有關連是不可能的，所以當我們發現大部分的物件都需要相依於某幾項特定物件的工作時，我們唯一能做的，就是盡全力讓這幾個重要物件的介面能夠保持穩定，而實做的方式就是善用介面讓物件的相依性與實體類別脫勾。

回到我美滿的家庭，既然一定要人煮飯，如果媽媽生病了沒辦法煮飯時，一定要有人能夠替代媽媽把飯菜煮好，可能是請庸人煮 (可是請不起庸人)，也可以叫外賣讓餐廳的廚師來煮。
從實做的觀點來看，我們讓煮飯這件事情成為一個抽象介面，在正常的狀況下，我們讓媽媽負責實作它，但是一旦媽媽不能煮飯了，外面餐廳的廚師也可以實作煮飯的介面來幫我們準備好飯菜。這樣一來我們全家人就能從此過著幸福快樂美滿的日子了。

當然也可以讓我們全家人都學習如何煮菜，但是這樣一來大家都會搞的很累，而且平常白天要上班跟上課，根本沒辦法來得及晚上趕回家煮菜，媽媽也說不定會開始耍賴叫我們負責煮菜，自己卻跑去 SHOPPING。這樣就違背了中國五千年文化說分工合作團結力量大的道理。

撇開物件導向設計的高深理論，上面所談論的觀念其實是最基本的馬步功夫，盡量讓物件之間的工作能夠彼此脫勾，不會互相影響，這就叫做降低物件之間的耦合力。每個物件之間有各自負責的工作，各司其職，彼此合作下達成整體的目標，這樣就叫做提高物件的內聚力。

物件導向設計時不見得一定要應用什麼複雜的 Patterns，但是卻一定要堅守物件導向設計這兩個基本原則，事實上，任何 Patterns 的出發點也都是為了能夠符合這兩個基本原則，讓設計盡善盡美。

by iT 邦部落格 http://netgo2oo.blog.ithome.com.tw/post/224/1349

藕合

1. 耦合就是一個類別認識另一個類別的程度，假如 A 類別對 B 類別的認識程度，只是知道 B 類別所暴露的介面，則 A 類別和 B 類別就是鬆散耦合...這是好事，相反的，假如 A 類別依賴於 B 類別的某個部份，但不僅止於 B 類別的介面，則這兩個類別就是緊密耦合...這不是好事
2. 假如 A 對於 B 的認識多於它所應該知道的，例如 A 必須知道 B 是如何實作的，則 A 和 B 就是緊密耦合
3. 根據第二種情況，假設 B 類別被加強了，而且加強 B 類別的開發者有可能根本不知道 A 類別的存在，理論上，只要不破壞這個類別的介面，搞任何的加強應該都是安全的，所以他有可能修改一些 B 類別內和介面無關的部份，這就有可能導致 A 類別出問題了

```
//緊密耦合
class Doubler {

    public static int doubleMe(Holder h) {
        return h.getAmount() * 2;
    }
}
class Holder {

    int amount = 10;

    public void doubleAmount() {
        amount = Doubler.doubleMe(new Holder());
    }

    public int getAmount() {
        return amount;
    }
//more code here
}
```

```
//鬆散耦合
//public class Doubler {
//
//    public static int doubleMe(int h) {
//        return h * 2;
//    }
//}
//
//class Holder {
//
//    int amount = 10;
//
//    public void doubleAmount() {
//        amount = Doubler.doubleMe(amount);
//    }
//
//    public int getAmount() {
//        return amount;
//    }
//more code here
//}
```

內聚力

1. 當耦合度講的是類別間如何互動，內聚力的重點全在於單一類別如何被設計，內聚力這個詞是被用來說明:一個類別擁有單一明確目標的程度，一個類別的目標越明確，它的內聚力就越高-這是好事
2. 高內聚力的關鍵好處是，這樣的類別通常比低內聚力的類別更容易維護(而且更少改變)
3. 另一個高內聚力的好處是 ， 一個目標明確的類別，通常會比其他類別更容易再利用

```java
//低內聚力
class 報表A {

    public void 印表機() {
        System.out.println("甲印表機");
    }
    public void 資料庫() {
        System.out.println("access");
    }
    public void 存檔() {
        System.out.println("存到excel");
    }
}
//低內聚力
class 報表B {

    public void 印表機() {
        System.out.println("Z印表機");
    }
    public void 資料庫() {
        System.out.println("SQLSERVER");
    }
    public void 存檔() {
        System.out.println("存到excel");
    }
}
//低內聚力
class 報表C {

    public void 印表機() {
        System.out.println("甲印表機");
    }
    public void 資料庫() {
        System.out.println("SQLSERVER");
    }
    public void 存檔() {
        System.out.println("存到WORD");
    }
}
```

```java
//低內聚力
public static void 印報表A1() {
    報表A p = new 報表A();
    p.印表機();
    p.資料庫();
    p.存檔();
}

public static void 印報表B1() {
    報表B p = new 報表B();
    p.印表機();
    p.資料庫();
    p.存檔();
}

public static void 印報表C1() {
    報表C p = new 報表C();
    p.印表機();
    p.資料庫();
    p.存檔();
}
```

```java
//高內聚力
class 印表機 {

    public void 甲印表機() {
        System.out.println("甲印表機");
    }
    public void Z印表機() {
        System.out.println("Z印表機");
    }
    public void 丙印表機() {
        System.out.println("丙印表機");
    }
}
//高內聚力
class 存檔 {

    public void 甲存檔() {
        System.out.println("甲存檔");
    }
    public void Z存檔() {
        System.out.println("Z存檔");
    }
    public void 丙存檔() {
        System.out.println("丙存檔");
    }
}
//高內聚力
class 資料庫 {

    public void 甲資料庫() {
        System.out.println("甲資料庫");
    }
    public void Z資料庫() {
        System.out.println("Z資料庫");
    }
    public void 丙資料庫() {
        System.out.println("丙資料庫");
    }
}
```

```java
//高內聚力
public static void 印報表A2() {
    印表機 x = new 印表機();
    資料庫 y = new 資料庫();
    存檔 z = new 存檔();

    x.丙印表機();
    y.甲資料庫();
    z.甲存檔();
}
public static void 印報表B2() {
    印表機 x = new 印表機();
    資料庫 y = new 資料庫();
    存檔 z = new 存檔();

    x.Z印表機();
    y.甲資料庫();
    z.Z存檔();
}
public static void 印報表C2() {
    印表機 x = new 印表機();
    資料庫 y = new 資料庫();
    存檔 z = new 存檔();

    x.甲印表機();
    y.Z資料庫();
    z.Z存檔();
}
```

序列化

1. 假如你想要儲存一個或多個物件的狀態 · 如果 Java 沒有序列化的功能(古老的 Java 並沒有這個功能) · 你就必須使用 I/O 類別的其中一個 · 來將每個你想要儲存的物件的實體變數的狀態寫出去 · 最糟糕的部份是 · 還必須從寫出的資料重新建立相同的物件 · 你將需要 · 為你如何寫出物件的狀態 · 以及如何從檔案回復成物件 · 建立好自己的物件寫出讀入協定 · 否則你將有可能把變數回復成不相同的值
2. 序列化的功能可以幫你儲存物件 · 和它的所有實質變數➔除非被標示為 "transient" 不想被存
3. 序列化的摩法類別➔**ObjectOutputStream and ObjectInputStream➔java.io 套件的高階類別**

序列化範本

```
public class SerialCat {

    public static void main(String[] args) {
        Cat c = new Cat(); // 2

        try {
            FileOutputStream fs = new FileOutputStream("testSer.ser");
            ObjectOutputStream os = new ObjectOutputStream(fs);
            os.writeObject(c); // 3

            os.close();
        } catch (Exception e) {
            e.printStackTrace();
        }

        try {
            FileInputStream fis = new FileInputStream("testSer.ser");
            ObjectInputStream ois = new ObjectInputStream(fis);
            c = (Cat) ois.readObject(); // 4

            ois.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

//序列化範本
class Cat implements Serializable {

}
```

物件圖

1. 儲存一個物件的真實意義是什麼 ? 如果實體變數都是基本型別 · 就非常直覺 · 但假如實體變數本身是物件的參考會儲存什麼 ? java 儲存任何的參考變數的真實的值是沒有意義的 · 因為 java 的參考變數的值只對當次正在執行的 JVM 有意義 · 換句話說 · 假如你嘗試在另一次執行中的 JVM 回復該物件 · 即使是在同一台電腦上跑 · 這個參考值也是無意義
2. 以下的程式 ➔當要儲存 Dog 時會發生什麼事 ? 假如主要目標是儲存然後再回復 · 那所回復的 Dog 必須和儲存的 Dog 一模一樣 · 因此回復的 Dog 的 Collar 也必須和儲存的 Dog 的 Collar 一模一樣 · 也就是說 Dog 和 Collar 必須同時被儲存 · 假如 Collar 也指涉到其它物件呢 ? 例如 Color 物件 · 事情很快就複雜起來 · 假如程式設計師必須負責知道每個 Dog 所指涉到的物件內部結構 · 然後再寫程式把它儲存起來 · 這將會是一場惡夢
3. 很幸運的 · java 的序列化將負責處理這些事情 · 當你對物件進行序列化時 · Java 的序列化就會儲存整個 "物件圖" · 就是說對於要序列化的物件進行深度複製
4. 記得➔必須藉由實作 Serializable 介面 · 來明確地選擇那些類別的物件可以被序列化

沒有序列化的當掉

```
public class SerialDog {

    public static void main(String[] args) {
        Collar c = new Collar(3);
        Dog d = new Dog(c, 5);
        try {
            FileOutputStream fs = new FileOutputStream("testSer.ser");
            ObjectOutputStream os = new ObjectOutputStream(fs);
            os.writeObject(d);
            os.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
class Dog implements Serializable {

    private Collar theCollar;
    private int dogSize;

    public Dog(Collar collar, int size) {
        theCollar = collar;
        dogSize = size;
    }

    public Collar getCollar() {
        return theCollar;
    }
}

class Collar {

    private int collarSize;

    public Collar(int size) {
        collarSize = size;
    }

    public int getCollarSize() {
        return collarSize;
    }
}
```

```
java.io.NotSerializableException: java22_序列化.Collar
        at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1180)
        at java.io.ObjectOutputStream.defaultWriteFields(ObjectOutputStream.java:1528)
        at java.io.ObjectOutputStream.writeSerialData(ObjectOutputStream.java:1493)
        at java.io.ObjectOutputStream.writeOrdinaryObject(ObjectOutputStream.java:1416)
        at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1174)
        at java.io.ObjectOutputStream.writeObject(ObjectOutputStream.java:346)
        at java22_序列化.Test22.serializeDog(Test22.java:41)
        at java22_序列化.Main.main(Main.java:7)
```

物件圖裡的每一個類別都要自行加上序列化

```
public class SerialDog1 {

    public static void main(String[] args) {
        Collar1 c = new Collar1(3);
        Dog1 d = new Dog1(c, 5);

        System.out.println("before: collar size is " + d.getCollar().getCollarSize());  //3

        try {
            FileOutputStream fs = new FileOutputStream("testSer.ser");
            ObjectOutputStream os = new ObjectOutputStream(fs);
            os.writeObject(d);
            os.close();
        } catch (Exception e) {
            e.printStackTrace();
        }

        try {
            FileInputStream fis = new FileInputStream("testSer.ser");
            ObjectInputStream ois = new ObjectInputStream(fis);
            d = (Dog1) ois.readObject();
            ois.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
        System.out.println("after: collar size is " + d.getCollar().getCollarSize());  //3
    }
}
//頸圈加上序列化
class Dog1 implements Serializable {

    private Collar1 theCollar;
    private int dogSize;

    public Dog1(Collar1 collar, int size) {
        theCollar = collar;
        dogSize = size;
    }

    public Collar1 getCollar() {
        return theCollar;
    }
}

class Collar1 implements Serializable {

    private int collarSize;

    public Collar1(int size) {
        collarSize = size;
    }

    public int getCollarSize() {
        return collarSize;
    }
}
```

```
before: collar size is 3
after: collar size is 3
```

transient 修飾字➔不想序列化的修飾字

1. 為何會有 transient 修飾字 ? 以上的例子 · 若我們沒辦法修改 Collar 類別呢 ? 也就是 · 假如我們沒辦法讓 Collar 類別實作 Serializable 要怎麼辦 · 是否就沒辦法序列化 Dog · 很明顯的 · 我們可以建立一個 繼承自 Collar 的子類別 · 並對這個子類別進行序列化 · 但這不總是可行

   <1>、 Collar 類別可能是 final 不能被繼承

   <2>、 Collar 類別本身指涉到不能被序列化的物件

   <3>、 繼承 Collar 並不相容於你的設計

2. 假如在 Dog 的 Collar 上加上 transient 修飾字能讓你在序列化 Dog 時略過 Collar
3. 但 · 還是會當掉 ➔ 因為沒被序列化的 Collar 物件會得到一個 null 值➔屬性加了 transient 的修飾字 · 因為序列化時會略過 · 反序列化時 初始化也不會做 · 所以會得到 null 值)

```java
public class SerialDog2 {

    public static void main(String[] args) {
        Collar2 c = new Collar2(3);
        Dog2 d = new Dog2(c, 5);
        System.out.println("before: collar size is " + d.getCollar().getCollarSize());
        try {
            FileOutputStream fs = new FileOutputStream("testSer.ser");
            ObjectOutputStream os = new ObjectOutputStream(fs);
            os.writeObject(d);
            os.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
        try {
            FileInputStream fis = new FileInputStream("testSer.ser");
            ObjectInputStream ois = new ObjectInputStream(fis);
            d = (Dog2) ois.readObject();
            ois.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
        System.out.println("after: collar size is " + d.getCollar().getCollarSize());
    }
}

//項圈不想序列化
class Dog2 implements Serializable {

    private transient Collar2 theCollar;
    private int dogSize;

    public Dog2(Collar2 collar, int size) {
        theCollar = collar;
        dogSize = size;
    }

    public Collar2 getCollar() {
        return theCollar;
    }
}
```

```
before: collar size is 3
Exception in thread "main" java.lang.NullPointerException
        at java22_序列化.Test22.serializeDog2(Test22.java:100)
        at java22_序列化.Main.main(Main.java:9)
Java Result: 1
```

自製的寫入和讀出

1. 使用 writeObject 與 readObject ➔序列化對於這樣的狀況擁有一個特殊的機制 · 一組 private 函式 · 你可以在你自己的類別實作它們 · 假如存在的話 · 將會在序列化和反序列化的時候自動被呼叫 · 是屬於序列化系統內的特殊回呼機制的一部份
2. 這個函式可以讓你介入序列化和反序列化的過程 · 當 Dog 被儲存時 · 你就可以在序列化的時候說 "順便提醒一下 · 在 Dog 被序列化時 · 我想將 Collar 變數 (一個 int) 的狀態也加入 Dog 資料流內
3. 在反序列化時 · 介入反序列化的過程 · 並且告訴 JVM · 我要讀取額外加在 Dog 資料流後面的 int · 並且使用它來建立一個新的 Collar · 再將新的 Collar 指派給已經被序列化完成的 Dog

```
public class SerialDog3 {

    public static void main(String[] args) {
        Collar3 c = new Collar3(3);
        Dog3 d = new Dog3(c, 5);

        System.out.println("before: collar size is " + d.getCollar().getCollarSize());

        try {
            FileOutputStream fs = new FileOutputStream("testSer.ser");
            ObjectOutputStream os = new ObjectOutputStream(fs);
            os.writeObject(d);
            os.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
        try {
            FileInputStream fis = new FileInputStream("testSer.ser");
            ObjectInputStream ois = new ObjectInputStream(fis);
            d = (Dog3) ois.readObject();
            ois.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
        System.out.println("after: collar size is " + d.getCollar().getCollarSize());
    }
}
```

```
//自製的讀出和寫入
class Dog3 implements Serializable {

    //不想序列化的變數 , 會恢復成 null 或 0 或 0.0
    private transient Collar3 theCollar;
    private int dogSize;

    public Dog3(Collar3 collar, int size) {
        theCollar = collar;
        dogSize = size;
    }
    public Collar3 getCollar() {
        return theCollar;
    }
    private void writeObject(ObjectOutputStream os) {

        try {
            // os.defaultWriteObject(); // 2
            os.writeInt(theCollar.getCollarSize()); // 3

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    private void readObject(ObjectInputStream is) {

        try {
            // is.defaultReadObject(); // 5
            theCollar = new Collar3(is.readInt()); // 6

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
class Collar3 {

    private int collarSize;

    public Collar3(int size) {
        collarSize = size;
    }

    public int getCollarSize() {
        return collarSize;
    }
}
```

```
before: collar size is 3
after: collar size is 3
```

繼承關係裡的序列化

1. 繼承如何影響序列化 ? 假如父類別是 Serializible 那根據 Java 介面的相關規則 · 該類別的所有子類別都自動地暗中實作了 Serializible
2. 但假如父類別不是 Serializible · 但子類別是➔會暗藏著嚴重的影響
3. 當一個物件 用 new 來建構時 (相對於反序列化)

   <1>、 預設初始化

   <2>、 先呼叫父親建構函數 · 再呼叫自己建構函數

   <3>、 明顯初始化

   <4>、 建構函數初始化
4. 這些事情在反序列化時都不會發生
5. 但 Dog 類別的不可序列化的 Animal 的部份將會被重新初始化 · 也就是所有發生在物件被建構時候的事情都會發生 · 但只發生在 Dog 繼承自 Animal 的部份 · Dog 繼承自不可序列化的 Animal 的部份都會恢復成預設的初始值 · 而不是他們被序列化時候的值

```
public class SuperNoSerial1 {

    public static void main(String[] args) {
        Dog4 d = new Dog4(35, "Fido");
        System.out.println("before: " + d.name + " " + d.weight);
        try {
            FileOutputStream fs = new FileOutputStream("testSer.ser");
            ObjectOutputStream os = new ObjectOutputStream(fs);
            os.writeObject(d);
            os.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
        try {
            FileInputStream fis = new FileInputStream("testSer.ser");
            ObjectInputStream ois = new ObjectInputStream(fis);
            d = (Dog4) ois.readObject();
            ois.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
        System.out.println("after: " + d.name + " " + d.weight);
    }
}

class Animal4 { // not serializable !

    public int weight = 42;
}

class Dog4 extends Animal4 implements Serializable {

    public String name;

    public Dog4(int w, String n) {
        weight = w; // inherited
        name = n; // not inherited
    }
}
```

```
before: Fido 35
after: Fido 42
```

```
public class SuperNoSerial2 {

    public static void main(String[] args) {
        CardPlayer c1 = new CardPlayer();
        try {
            FileOutputStream fos = new FileOutputStream("play.txt");
            ObjectOutputStream os = new ObjectOutputStream(fos);
            os.writeObject(c1);
            os.close();
            FileInputStream fis = new FileInputStream("play.txt");
            ObjectInputStream is = new ObjectInputStream(fis);
            CardPlayer c2 = (CardPlayer) is.readObject();
            is.close();
        } catch (Exception x) {
        }
    }
}

class Player {

    public Player() {
        System.out.print("p");
    }
}

class CardPlayer extends Player implements Serializable {

    public CardPlayer() {
        System.out.print("c");
    }

    public static void main(String[] args) {
        CardPlayer c1 = new CardPlayer();
        try {
            FileOutputStream fos = new FileOutputStream("play.txt");
            ObjectOutputStream os = new ObjectOutputStream(fos);
            os.writeObject(c1);
            os.close();
            FileInputStream fis = new FileInputStream("play.txt");
            ObjectInputStream is = new ObjectInputStream(fis);
            CardPlayer c2 = (CardPlayer) is.readObject();
            is.close();
        } catch (Exception x) {
        }
    }
}
```

pCp

屬性加了 transient 的修飾字 · 因為序列化時會略過 · 所以反序列化時 初始化不會做 · 因此 · 值就會是 0

```
public class NoSerial {

    public static void main(String[] args) {
        Bar d = new Bar();

        System.out.println("before: x is " + d.x);

        try {
            FileOutputStream fs = new FileOutputStream("testSer.ser");
            ObjectOutputStream os = new ObjectOutputStream(fs);
            os.writeObject(d);
            os.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
        try {
            FileInputStream fis = new FileInputStream("testSer.ser");
            ObjectInputStream ois = new ObjectInputStream(fis);
            d = (Bar) ois.readObject();
            ois.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
        System.out.println("after: x is " + d.x);
    }
}

class Bar implements Serializable {

    public transient int x = 42;
}
```

```
before: x is 42
after: x is 0
```

序列化不會影響靜態變數➔靜態變數是單純的類別變數 · 他們和個別物件沒有關係 · 序列化只應用在物件上面

```
public class StaticVar {

    public static void main(String[] args) {

        SpecialSerial s = new SpecialSerial();

        try {
            ObjectOutputStream os = new ObjectOutputStream(new FileOutputStream("myFile"));
            os.writeObject(s);
            os.close();
            System.out.print(++s.z + " ");
            ObjectInputStream is = new ObjectInputStream(new FileInputStream("myFile"));
            SpecialSerial s2 = (SpecialSerial) is.readObject();

            is.close();
            System.out.println(s2.y + " " + s2.z);

        } catch (Exception e) {
            System.out.println("exc");
        }
    }

}

class SpecialSerial implements Serializable {

    transient int y = 7;
    static int z = 9;
}
```

```
10 0 10
```

java 的資源回收

1. 資源回收是用來描述 Java 的自動記憶體管理 · 記憶體可以拿來建立堆疊、堆積、Java 的常數儲存池、函數區
2. Java 將物件儲存在記憶體的堆積內 · 而它也是 java 的資源回收機制唯一會處理到的記憶體

資源回收器什麼時候才會執行 ?

1. 資源回收器是被 JVM 所控制的 · JVM 決定什麼時候啟動資源回收器 · 在你的 Java 程式內 · 你可以要求 JVM 執行資源回收器 · 但在任何情況下 · 並不保證 JVM 會服從這個指令 · J VM 通常偵測到記憶體不夠用時 · 才會執行資源回收
2. 經驗告訴我們 · 當你的 Java 程式要求進行資源回收的時候 · JVM 通常會很快地同意你的請求 · 但這並不是保證 · 所以在你認為可以依賴它的時後 · JVM 就有可能決定要忽略你的請求

如何讓物件有資格被資源回收

1. 將參考設為 null
2. 重新指派參考變數
3. 特殊情況
4. 獨立參考

```
public class GarbageTruck1 {

    public static void main(String[] args) {
        StringBuffer sb = new StringBuffer("hello");
        System.out.println(sb);
        // The StringBuffer object is not eligible for collection
        sb = null;
        // Now the StringBuffer object is eligible for collection
    }
}
```

```
public class GarbageTruck2 {

    public static void main(String[] args) {
        StringBuffer s1 = new StringBuffer("hello");
        StringBuffer s2 = new StringBuffer("goodbye");
        System.out.println(s1);

        s1 = s2; // Redirects s1 to refer to the "goodbye" object

    }
}
```

```
public class GarbageTruck3 {

    public static void main(String[] args) {
        Date d = getDate();
        System.out.println("d = " + d);
    }

    public static Date getDate() {
        Date d2 = new Date();
        StringBuffer now = new StringBuffer(d2.toString());
        System.out.println(now);
        return d2;
    }
}
```

```
public class GarbageTruck4 {

    GarbageTruck4 i;

    public static void main(String[] args) {
        GarbageTruck4 i2 = new GarbageTruck4();
        GarbageTruck4 i3 = new GarbageTruck4();
        GarbageTruck4 i4 = new GarbageTruck4();
        i2.i = i3; // i2 refers to i3
        i3.i = i4; // i3 refers to i4
        i4.i = i2; // i4 refers to i2
        i2 = null;
        i3 = null;
        i4 = null;
    }
}
```

強迫資源回收

1. 資源回收是不能被強迫的 · 但 Java 提供某種函式讓你可以要求 JVM 去進行資源回收 · Java 所提供的函式只能請求而不能命令 JVM · 所以 JVM 將會盡力完成你的請求 · 但並不保證它會服從
2. 可以透過 Rumtime 類別的成員函式來要求 JVM 進行資源回收 · Runtime 類別是一個特殊的類別 · 每一個 Java 程式都有此類別的物件 · 而且只有唯一的一個 · 為了拿到 RunTime 的實體 · 你可以使用 Runtime.getRuntime()這個函式 ·它會回傳 Runtime 的單一物件 · 就可以使用 gc()來呼叫資源回收

   <1>.可以呼叫 System.gc()➔是一個靜態函式，也是要求資源回收的最簡單方法

   <2>.理論上在呼叫 System.gc()之後，JVM 就會盡量擠出可用的記憶體給你，" 理論上" 是因為這個程序不一定是這樣運作
3. 你的 JVM 或許沒有實作這個程序 · Java 規格允許資源回收不做任何事情
4. 另一個執行緒或許在你執行完資源回收就已經占據了大量的記憶體

```
public class CheckGC {

    public static void main(String[] args) {

        Runtime rt = Runtime.getRuntime();

        System.out.println("Total JVM memory: " + rt.totalMemory());
        System.out.println("Before Memory = " + rt.freeMemory());

        Date d = null;
        for (int i = 0; i < 10000; i++) {
            d = new Date();
            d = null;
        }
        System.out.println("After Memory = " + rt.freeMemory());
     // rt.gc();
        System.gc();
        System.out.println("After GC Memory = " + rt.freeMemory());

    }
}
```

```
Total JVM memory: 16252928
Before Memory = 15630200
After Memory = 15354608
After GC Memory = 15834184
```

finalize 函式➔資源回收器

1. Java 提供你一個機制 · 在你的物件被資源回收器刪除之前會先執行某些程式碼 · 這些程式碼就放在 finalize()這個函式 · 所有的類別都從 Object 類別那裡繼承了這個函式
2. 或許你的物件開啟了某些資源 · 而你想要在此物件被刪除之前關掉他們 · 但問題就是出在 · 因為某些原因你不能依賴資源回收器來刪除物件 · 所以任何你放在覆寫的 finalize()的程式碼 · 並不一定會被執行 · 所以不要將任何必要的程式碼放在 finalize()內
3. finalize()的小把戲

<1>、 對於任何物件 · finalize()只會被資源回收器呼叫一次(最多)

<2>、 呼叫 finalize()真的有可能導致物件不會被刪除

```
public class CheckFinalize1 {

    CheckFinalize1 i;

    public static void main(String[] args) {
        CheckFinalize1 i2 = new CheckFinalize1();
        CheckFinalize1 i3 = new CheckFinalize1();
        CheckFinalize1 i4 = new CheckFinalize1();
        i2.i = i3;
        i3.i = i4;
        i4.i = i2;
        i2 = null;
        i3 = null;
        i4 = null;

        System.gc();
    }

    public void finalize() {
        System.out.println("有發生資源回收喔....");
    }
}
```

```
有發生資源回收喔....
有發生資源回收喔....
有發生資源回收喔....
```

```java
public class CheckFinalize2 {

    public static Runtime rt = Runtime.getRuntime();

    public static void main(String[] args) {
        Date d = getDate();
        System.out.println("d = " + d);
        d=null; //再一次有資格被回收 , 但 finalize 已不會被執行
        System.gc();
    }
    public static Date getDate() {
        Date d2 = new Date();
        StringBuffer now = new StringBuffer(d2.toString());
        System.out.println(now);
        return d2;  //已經喪失回收的機會
    }
    public void finalize() {
        System.out.println("有發生資源回收喔....");
    }
}
```

```java
interface Animal {

    void makeNoise();
}

class Horse implements Animal {

    Long weight = 1200L;

    public void makeNoise() {
        System.out.println("whinny");
    }
}
public class Icelandic extends Horse {

    public void makeNoise() {
        System.out.println("vinny");
    }

    public static void main(String[] args) {
        Icelandic i1 = new Icelandic();
        Icelandic i2 = new Icelandic();
        Icelandic i3 = new Icelandic();
        i3 = i1;
        i1 = i2;
        i2 = null;
        i3 = i1;
        System.gc();
    }
    public void finalize() {
        System.out.println("有發生資源回收喔....");
    }
}
```

```
有發生資源回收喔....
有發生資源回收喔....
```

DAO(Data Access Object)介绍

1.、DAO(Data Access Object)介绍

DAO 應用在数据层那块，用于访问数据库，对数据庫进行操作的類。

2.、DAO 设计模式的结構

DAO 设计模式一般分为几个类：

1.VO(Value Object)：一個用於存放網頁的一行數據即一條記錄的類，比如网页要显示一个用户的信息，则这个类就是用户的类。

2.DatabaseConnection：用于打开和关闭数据库。

3.DAO 介面：用于声明对于数据库的操作。

4.DAOImpl：必须实现 DAO 介面，真实实现 DAO 接口的函数，但是不包括数据库的打开和关闭。

5.DAOProxy：也是实现 DAO 介面，但是只需要借助 DAOImpl 即可，但是包括数据库的打开和关闭。

6.DAOFactory：工廠類，含有 getInstance()创建一个 Proxy 類。

3.、DAO 的好处

DAO 的好处就是提供给用户的接口只有 DAO 的接口，所以如果用户想添加数据，只需要调用 create 函数即可，不需要数据库的操作。

4.、DAO 包命名

对于 DAO，包的命名和类的命名一定要有层次。

5.、实例解析

1.Emp.java

```
package org.vo，
import java.util.*，
public class Emp{
    private int empno，
    private String ename，
    private String job，
    private Date hireDate，
    private float sal，
    public Emp(){

    }
    public int getEmpno(){
        return empno，
    }
```

```
    public void setEmpno(int empno){
        this.empno = empno，
    }
    public String getEname(){
        return ename，
    }
    public void setEname(String ename){
        this.ename = ename，
    }
    public Date getHireDate(){
        return hireDate，
    }
    public void setHireDate(Date hireDate){
        this.hireDate = hireDate，
    }
    public float getSal(){
        return sal，
    }
    public void setSal(float sal){
        this.sal = sal，
    }
    public String getJob(){
        return job，
    }
    public void setJob(String job){
        this.job = job，
    }
}
```

2.DatabaseConnection.java

```
package org.dbc，
import java.sql.*，
public class DatabaseConnection{
    private Connection con = null，
    private static final String DRIVER = "com.mysql.jdbc.Driver"，
    private static final String USER = "root"，
    private static final String URL = "jdbc:mysql://localhost:3306/mldn"，
    private static final String PASS = "12345"，
    public DatabaseConnection()throws Exception{
        Class.forName(DRIVER)，
        con = DriverManager.getConnection(URL，USER，PASS)，
    }
    public Connection getConnection()throws Exception{
```

```
        return con，
    }
    public void close()throws Exception{
        if(con!=null){
            con.close()，
        }
    }
}
```

3.IEmpDAO.java

```
package org.dao，
import java.util.List，
import org.vo.*，
public interface IEmpDAO{
    public boolean doCreate(Emp emp)throws Exception，
    public List<Emp> findAll()throws Exception，
    public Emp findById(int empno)throws Exception，
}
```

4.EmpDAOImpl.java

```
package org.dao.impl，
import org.dao.*，
import java.sql.*，
import org.vo.*，
import java.util.*，
public class EmpDAOImpl implements IEmpDAO{
    private Connection con，
    private PreparedStatement stat = null，
    public EmpDAOImpl(Connection con){
        this.con = con，
    }
    public boolean doCreate(Emp emp)throws Exception{
        String sql = "INSERT INTO emp(empno，ename，job，hiredate，sal) VALUES(?，?，?，?，?)"，
        stat = con.prepareStatement(sql)，
        stat.setInt(1，emp.getEmpno())，
        stat.setString(2，emp.getEname())，
        stat.setString(3，emp.getJob())，
        stat.setDate(4，new java.sql.Date(emp.getHireDate().getTime()))，
        stat.setFloat(5，emp.getSal())，
        int update = stat.executeUpdate()，
        if(update>0){
            return true，
        }
```

```
        else{
            return false，
        }
    }
    public List<Emp> findAll()throws Exception{
        String sql = "SELECT empno，ename，job，hiredate，sal FROM emp"，
        stat = con.prepareStatement(sql)，
        ResultSet rs = stat.executeQuery()，
        Emp emp = null，
        List<Emp> list = new ArrayList<Emp>()，
        while(rs.next()){
            int empno = rs.getInt(1)，
            String ename = rs.getString(2)，
            String job = rs.getString(3)，
            float sal = rs.getFloat(5)，
            emp = new Emp()，
            emp.setEmpno(empno)，
            emp.setEname(ename)，
            emp.setJob(job)，
            emp.setHireDate(rs.getDate(4))，
            emp.setSal(sal)，
            list.add(emp)，
        }
        return list，
    }
    public Emp findById(int empno)throws Exception{
        String sql = "SELECT empno，ename，job，hiredate，sal FROM emp WHERE empno=?"，
        stat = con.prepareStatement(sql)，
        stat.setInt(1，empno)，
        ResultSet rs = stat.executeQuery()，
        Emp emp = null，
        if(rs.next()){
            String ename = rs.getString(2)，
            String job = rs.getString(3)，
            float sal = rs.getFloat(5)，
            emp = new Emp()，
            emp.setEmpno(empno)，
            emp.setEname(ename)，
            emp.setJob(job)，
            emp.setHireDate(rs.getDate(4))，
            emp.setSal(sal)，
        }
        return emp，
```

```
    }
}
```

5.EmpDAOProxy.java

```
package org.dao.impl，
import org.dao.*，
import java.sql.*，
import org.vo.*，
import java.util.*，
import org.dbc.*，
public class EmpDAOProxy implements IEmpDAO{
    private DatabaseConnection dbc，
    private IEmpDAO dao = null，
    public EmpDAOProxy()throws Exception{
        dbc = new DatabaseConnection()，
        dao = new EmpDAOImpl(dbc.getConnection())，
    }
    public boolean doCreate(Emp emp)throws Exception{
        boolean flag = false，
        if(dao.findById(emp.getEmpno())==null){
            flag = dao.doCreate(emp)，
        }
        dbc.close()，
        return flag，
    }
    public List<Emp> findAll()throws Exception{
        List<Emp>list = dao.findAll()，
        dbc.close()，
        return list，
    }
    public Emp findById(int empno)throws Exception{
        Emp emp = dao.findById(empno)，
        dbc.close()，
        return emp，
    }
}
```

6.DAOFactory.java

```
package org.dao.factory，
import org.dao.*，
import java.sql.*，
import org.vo.*，
import java.util.*，
```

```
import org.dbc.*，
import org.dao.impl.*，
public class DAOFactory{
    public static IEmpDAO getInstance(){
        IEmpDAO dao = null，
        try{
            dao = new EmpDAOProxy()，
        }
        catch(Exception e){
            e.printStackTrace()，
        }
        return dao，
    }
}
```

7.TestDAO.java

```
package org.dao.test，
import org.dao.factory.*，
import org.vo.*，
import org.dao.*，
public class TestDAO{
    public static void main(String args[])throws Exception{
        Emp emp = null，
        for(int i=0，i<5，i++){
            emp = new Emp()，
            emp.setEmpno(i)，
            emp.setEname("xiazdong-"+i)，
            emp.setJob("stu-"+i)，
            emp.setHireDate(new java.util.Date())，
            emp.setSal(500*i)，
            DAOFactory.getInstance().doCreate(emp)，
        }
    }
}
```

[OO 設計模式] Singleton Patterns：確保同一時間只有一個實例或物件進行服務

前言：

在某些時候， 為了管控資源或是避免多個物件分享相同資源造成的 Race condition， 我們希望在同一時間， 某個類別只會存在一個唯一的物件， 這時你便可以利用這個設計模式.

範例一：

代碼說明：透過下面代碼， 我們透過統一的 API 來獲得類別 Singleton 的物件， 只要該類別被建立過在接下來的 null 判斷就不在產生新的物件.

view plaincopy to clipboardprint?

```
1. public class Singleton{
2.     private static Singleton uniqueInstance；
3.     private Singleton(){} // 使用 Private 建構子， 確保類別 Singleton 的物件化只能透過
API:getInstance()
4.     public static Singleton getInstance() {
5.         if(uniqueInstance == null ) {uniqueInstance = new Singleton()；}
6.         return uniqueInstance；
7.     }
8. }
```

問題點：

乍看好像可以滿足 Singleton 的要求， 但如果是在多線程環境下呢? 可能某支 Thread 正在產生物件同時， 另一支 Thread 在 if(uniqueInstance==null) 判斷時也進入迴圈， 造成同一時間產生了一個以上的物件， 為了解決這個問題可以使用 synchronized 關鍵字：

view plaincopy to clipboardprint?

```
1. public class Singleton{
2.     private static Singleton uniqueInstance；
3.     private Singleton(){} // 使用 Private 建構子， 確保類別 Singleton 的物件化只能透過
API:getInstance()
4.     public static synchronized Singleton getInstance(){ // 使用 synchronized 關鍵字避免同時兩支 Thread
進入函數
5.         if(uniqueInstance == null ) {uniqueInstance = new Singleton()；}
6.         return uniqueInstance；
7.     }
```