第 18 章　專題：序列鍵生成器與單例及多例樣式

本章的內容來自於一個真實的全球金融網站專案，本書去掉了所有與商業有關的內容，僅僅討論技術觀點。

18.1　問　　題

序列鍵

開發過資料庫驅動資訊系統的讀者都知道，在一個關聯資料庫中，所有的資料都是存儲在表格裏的；而現代的資料庫設計要求每一個表格都有一個主鍵（Primary Key）。對大多數的用戶輸入資料來講，主鍵需要由系統以序列號方式產生，而不是由操作人員給出。

有一些關聯資料庫引擎提供某種序列鍵生成機制。比如，Microsoft SQL Server 提供一個 AutoNumber 的屬性，允許每一個表格內可以有一個 AutoNumber 列；Oracle 提供 Sequence 物件，可以提供序列鍵值。下面的 SQL 語句會建立一個名爲 ISSUE_SEQ 的 Sequence 物件，其開始值爲 1000，每次增加 1。

程式碼清單 1：用來建構一個 Oracle 序列的 SQL 語句程式碼

```
CREATE SEQUENCE ISSUE_SEQ
 INCREMENT BY 1
 START WITH 1000
 MINVALUE 20
 MAXVALUE 9999999999999999999999999
 NOCYCLE
 NOORDER
NOCACHE
```

其他一些資料庫引擎則沒有相應的機制，比如 Sybase 就沒有類似的功能。這時候就只好使用一個表格，並在其內部設有兩個列，一個列存放鍵名，另一個列存放鍵值，用戶端使用 SQL 語句自行管理鍵值。使用一個表來存儲所有的序列鍵，如下表所示。

KeyName	KeyValue
PO_NUMBER	1045
SO_NUMBER	5069

（續表）

ITEM_NUMBER	10874
RMA_NUMBER	2065
…	

有些商業化的系統（如 Vignette Story Server）需要支援幾種主要的資料庫，包括 Oracle，Microsoft SQL Server 和 Sybase 等。這樣的系統就不能使用 Microsoft SQL Server 或者 Oracle 特有的機制，而必須使用某種具有普遍性的機制。這時，上面提到的使用一個表來管理各種主鍵就變得較爲合理。

預定式存儲

不論是使用哪一個機制，最終系統必須要有一些資料庫操作來提供這些序列值。比如一個餐館的販賣系統需要一個序列號給每天開出去的賣單編號，這個序列號碼就應當存放到資料庫裏面。每當發出序列號碼的時候，都應當從資料庫讀取這個號碼，並更新這個號碼。

爲了保證在任何情況下鍵值都不會出現重複，應當使用預定式鍵值存儲辦法。在請求一個鍵值時，首先將資料庫中的鍵值更新爲下一個可用值，然後將舊值提供給用戶端。這樣萬一出現運行中斷的話，最多就是這個鍵值被浪費掉。

與此相對的是記錄式鍵值存儲辦法。也就是說，鍵值首先被傳回給用戶端，然後記錄到資料庫中去。這樣做的缺點是，一旦系統中斷，就有可能出現用戶端已經使用了一個鍵值，而這個鍵值卻沒有來得及存儲到資料庫中的情況。在系統重啓之後，系統還會從這個已經被使用過的鍵值開始，從而導致錯誤。因此不要使用這種登記式的存儲辦法。

預定式的存儲辦法可以每一次預定多個鍵值（也即一個鍵值區間）；而不是每一次僅僅預定一個值。由於這些值都是一些序列數值，因此，所謂一次預定多個值，不過就是每次更新鍵值時將鍵值增加一個大於 1 的數目。在後面的設計方案中，首先考慮每次預定一個鍵值的做法，然後將之改進爲每次預定 20 個值的情況。

單例樣式的應用

上面討論了序列的存儲機制，另一個重要的機制是鍵的查詢管理機制。與其將鍵值的查詢工作交給各個模組，不如將之集中到一個物件身上。這個物件負責管理序列鍵的查詢，稱之爲序列鍵管理器。

顯然，不難看出，整個系統只需要一個序列鍵管理器物件。由於系統運行期間總是需要序列鍵，因此序列鍵管理器物件需要在系統運行期間存在。考慮到可以讓一個序列鍵管理器負責管理分屬於不同模組的多個序列鍵，因此這個序列鍵管理器需要讓整個系統訪問。

學習過單例樣式的讀者會意識到，這個系統設計應當使用到單例樣式。是的，這個序列鍵管理器可以設計成一個單例類別。

一個用戶端系統往往需要管理不止一個鍵值，而是多個鍵值。這時候，可以將這個單

例物件的內部狀態擴展成爲一個聚集，從而可以存儲任意多個鍵值。也就是說，這個序列鍵管理器是一個聚集物件，而此聚集本身是一個單例物件。

關於單例樣式，請讀者參考本書的“單例（Singleton）樣式”一章。

多例樣式的應用

多例樣式往往持有一個內蘊狀態；多例類別的每一個實例都有獨特的內蘊狀態。一個多例類別持有一個聚集物件，用來登記自身的實例，而其內蘊狀態往往就是登記的鍵值。當用戶端通過多例類別的靜態工廠方法請求多例類別的實例時，這個工廠方法都會在聚集內查詢是否已經有一個這樣的實例。如果有，就直接傳回給用戶端；如果沒有，就首先建構一個這樣的實例，將之登記到聚集中，然後再向用戶端提供。

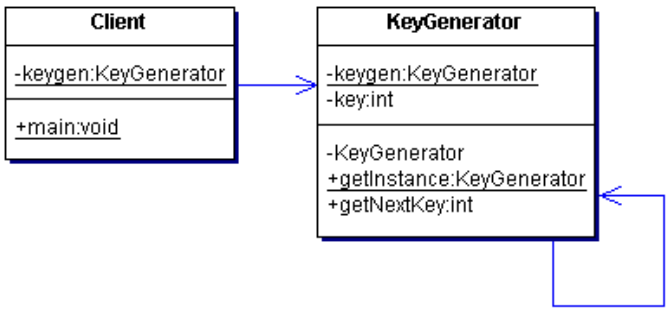
關於多例樣式以及它與單例樣式的關係，請讀者參考本書的“專題：多例（Multiton）樣式與多語言支援 ”一章。

18.2　將單例樣式應用到系統設計中

下面從一個最簡單的情況出發，逐漸將問題的複雜性提高，直到給出具有實用價值的解決方案爲止。

方案一：沒有資料庫的情況

首先考慮一個沒有資料庫背景的方案，這個設計由一個單例類別 KeyGenerator 組成。一個序列鍵生成器的類別圖如下圖所示。



下面是這個鍵生成器 KeyGenerator 的程式碼。

程式碼清單 2：KeyGenerator 類別的程式碼

```
package com.javapatterns.keygen.ver1;
public class KeyGenerator
{
    private static KeyGenerator keygen =
```

```
            new KeyGenerator();
        private int key = 1000;
        /**
         *   私有建構方法，保證外界無法直接實例化
         */
        private KeyGenerator() {}
        /**
         *   靜態工廠方法，提供自己的實例
         */
        public static KeyGenerator getInstance()
        {
            return keygen;
        }
        /**
         *   取值方法，提供下一個合適的鍵值
         */
        public synchronized int getNextKey()
        {
            return key++;
        }
}
```

可以看出，上面的 KeyGenerator 類別的建構方法是私有的，因此，外界無法通過呼叫建構方法將之實例化。同時，它提供了一個靜態的工廠方法 getInstance()，自己向外界提供自己的實例。如果再考查一下這個工廠方法就會發現，這個方法永遠僅提供同一個實例。換言之，這是一個單例類別。

商業方法 getNextKey()傳回一個整型數，這個數會自行遞增，每次加 1。

程式碼清單 3：客戶類別的程式碼

```
package com.javapatterns.keygen.ver1;
public class Client
{
    private static KeyGenerator keygen;
    public static void main(String[] args)
    {
        keygen = KeyGenerator.getInstance();
        System.out.println("key = " + keygen.getNextKey());
        System.out.println("key = " + keygen.getNextKey());
        System.out.println("key = " + keygen.getNextKey());
    }
}
```

在運行時，客戶物件會列印出得到的序列鍵的數值，這表明系統是正常工作的。

程式碼清單 4：運行結果

```
key = 1000
key = 1001
```

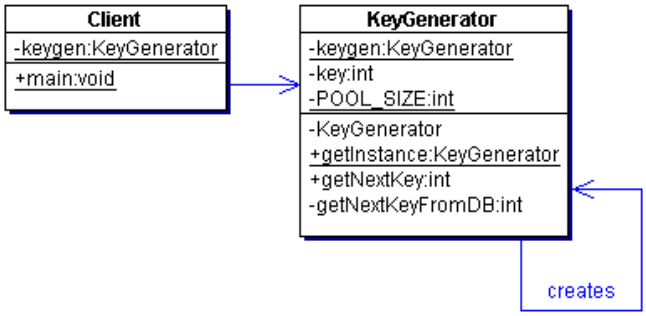
key = 1002

這一設計基本上實作了向用戶端提供鍵值的功能，但是也有明顯的缺點。由於沒有資料庫的存儲，一旦系統重新啓動，KeyGenerator 都會重新初始化，這就會造成鍵值的重複。

爲了避免這一點，就必須將每次的鍵值存儲起來，以便一旦系統中斷和重啓時，可以將這個鍵值取出，並在這個值的基礎上重新開始。這就將設計師引向了下一個設計方案。

方案二：有資料庫的情況

這個方案是對方案一的修正。與方案一一樣，這個設計由一個單例類別組成；而與方案一不同的是，這個單例類別有資料庫功能。它將鍵值存儲在資料庫的表中，每次用戶端請求鍵值時，首先將這個表中的值增加 1，然後將這個值傳回給用戶端（當然，這兩個資料庫操作應當是一個完整的交易單位）。有資料庫的鍵值生成器的結構圖如下圖所示。



下面就是 KeyGenerator 類別的程式碼。這是一個單例類別，它提供了私有的建構方法，所以外界無法直接將其實例化，所有的實例化請求都必須通過靜態工廠方法進行。

程式碼清單 5：KeyGenerator 的程式碼

```
package com.javapatterns.keygen.ver2;
public class KeyGenerator
{
    private static KeyGenerator keygen =
          new KeyGenerator();
  /**
   *  私有建構方法，保證外界無法直接實例化
   */
   private KeyGenerator() {}
  /**
   *  靜態工廠方法，提供自己的實例
   */
   public static KeyGenerator getInstance()
   {
```

```
            return keygen;
        }
       /**
        *   取值方法，提供下一個合適的鍵值
        */
        public synchronized int getNextKey()
        {
            return getNextKeyFromDB();
        }
        private int getNextKeyFromDB()
        {
            String sql1 = "UPDATE KeyTable SET keyValue = keyValue + 1 ";
            String sql2 = "SELECT keyValue FROM KeyTable";
            //execute the update SQL
            //run the SELECT query
            //示意性地傳回一個數值
            return 1000;
        }
    }
```

在接到用戶端的請求時，這個 KeyGenerator 每次都向資料庫查詢鍵值，將新的鍵值登記到表格裏，然後將查詢的結果傳回給用戶端。在上面的程式碼中，給出了兩個 SQL 語句，但是並沒有給出執行這兩行語句的 JDBC 程式碼。相信讀者在將這個設計應用到自己的系統中時，可以將必要的 JDBC 程式碼加進去。

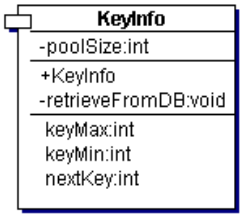
必須指出的是，爲了將讀者的注意力集中在系統設計上面，本書不想涉及到 JDBC 的細節，因此上面並沒有給出這方面的程式碼。相信讀者在將這個設計應用到自己的系統中去時，可以自行實作這部分程式碼。

在這個設計方案裏面，可以使用與第一個設計方案相同的用戶端，因此就不在此重複了。

方案三：鍵值的暫存方案

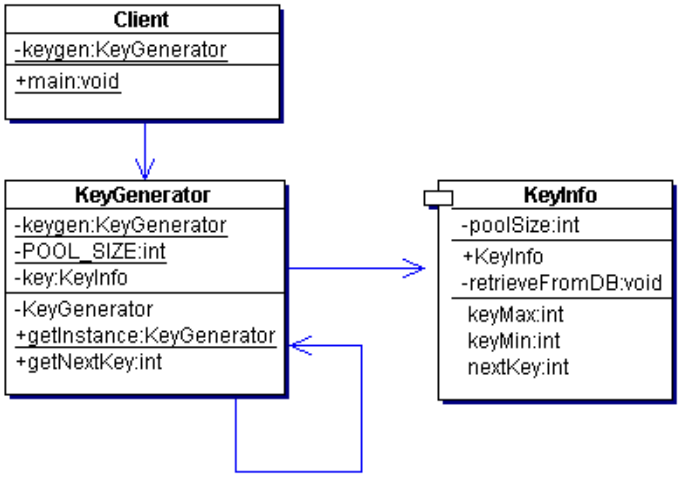
每一次都進行鍵值的查詢有必要嗎？畢竟一個鍵的值只是一些序列號碼，與其每接到一次請求就查詢一次，然後向用戶端提供這一個值，不如在一次查詢中一次性地預先登記多個鍵值，然後連續多次地向用戶端提供這些預訂的鍵值。這樣一來，不是節省了大部分不必要的資料庫查詢操作嗎？

是的，這就是鍵值的暫存機制。當 KeyGenerator 每次更新資料庫中的鍵值時，它都將鍵值增加。與方案二不同之處是，鍵值的增加值不是 1 而是更多。在下面給出的例子中，鍵值的增加值是 20。爲了存儲所有的與鍵有關的資訊，特地引進一個 KeyInfo 類別，如下圖所示。



這個 KeyInfo 除了存儲與鍵有關的資訊外，還提供了一個 retrieveFromDB()方法，向資料庫查詢鍵值。每次查詢得到的 20 個鍵值會在隨後提供給請求者，直到 20 個鍵值全部使用完畢，然後再向資料庫預定後 20 個鍵值。

KeyGenerator 作爲一個單例類別，保持一個對 KeyInfo 物件的引用。用戶端呼叫 getNextKey()方法以得到下一個鍵的鍵值。有暫存的序列鍵生成機制如下圖所示。



下面就是 KeyGenerator 類別的程式碼。與第二個方案一樣，KeyGenerator 類別使用了私有的建構方法，以及一個靜態工廠方法向外界提供自己惟一的實例。

程式碼清單 6：KeyGenerator 的程式碼

```
package com.javapatterns.keygen.ver3;
public class KeyGenerator
{
    private static KeyGenerator keygen =
          new KeyGenerator();
    private static final int POOL_SIZE = 20;
    private KeyInfo key ;
   /**
    *   私有建構方法，保證外界無法直接實例化
    */
    private KeyGenerator()
    {
          key = new KeyInfo(POOL_SIZE);
```

```
    }
    /**
     *  靜態工廠方法，提供自己的實例
     */
    public static KeyGenerator getInstance()
    {
        return keygen;
    }
    /**
     *  取值方法，提供下一個合適的鍵值
     */
    public synchronized int getNextKey()
    {
        return key.getNextKey();
    }
}
```

下面是 KeyInfo 類別的程式碼。正如同上面所談到的，這個類別提供了向資料庫查詢的功能，並且存儲一定數目的鍵值。

程式碼清單 7：KeyInfo 的程式碼

```
package com.javapatterns.keygen.ver3;
class KeyInfo
{
    private int keyMax;
    private int keyMin;
    private int nextKey;
    private int poolSize;
    /**
     *  建構方法
     */
    public KeyInfo(int poolSize)
    {
        this.poolSize = poolSize;
        retrieveFromDB();
    }
    /**
     *  取值方法，提供鍵的最大值
     */
    public int getKeyMax()
    {
        return keyMax;
    }
    /**
     *  取值方法，提供鍵的最小值
     */
    public int getKeyMin()
```

```
        {
            return keyMin;
        }
    /**
     *  取值方法，提供鍵的當前值
     */
    public int getNextKey()
    {
        if (nextKey > keyMax)
        {
            retrieveFromDB();
        }
        return nextKey++;
    }
    /**
     *  內部方法，從資料庫提取鍵的當前值
     */
    private void retrieveFromDB()
    {
        String sql1 = "UPDATE KeyTable SET keyValue = keyValue + "
            + poolSize + " WHERE keyName = 'PO_NUMBER'";
        String sql2 = "SELECT keyValue FROM KeyTable WHERE KeyName = 'PO_NUMBER'";
        // execute the above queries in a transaction and commit it
        // assume the value returned is 1000
        //示意性地傳回一個數值
        int keyFromDB = 1000;
        keyMax = keyFromDB;
        keyMin = keyFromDB - poolSize + 1;
        nextKey = keyMin;
    }
}
```

下面就是一個示意性的用戶端的程式碼。

程式碼清單 8：一個示意性的用戶端的程式碼

```
package com.javapatterns.keygen.ver3;
public class Client
{
    private static KeyGenerator keygen;
    public static void main(String[] args)
    {
        keygen = KeyGenerator.getInstance();
        for (int i = 0 ; i < 20 ; i++)
        {
            System.out.println("key(" + (i+1)
                + ")= " + keygen.getNextKey());
        }
```

```
        }
    }
```

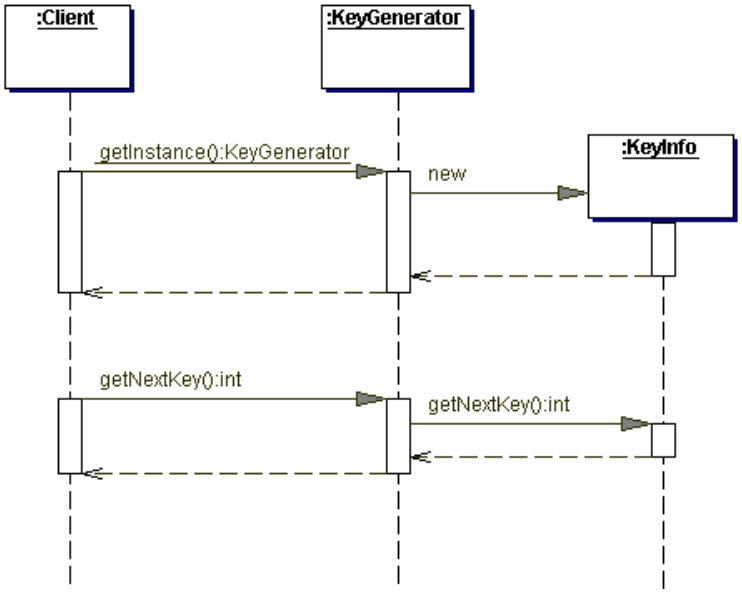
可以看出，這個示意性的用戶端首先通過呼叫 KeyGenerator 的靜態工廠方法得到 KeyGenerator 的實例，然後呼叫 KeyGenerator 物件的取值方法，以得到一個個的鍵值。運行的結果如下所示。

程式碼清單 9：系統的運行結果

```
key(1)= 981
key(2)= 982
key(3)= 983
.....
key(19)= 999
key(20)= 1000
```

爲了說明系統的活動時序，這裏特地給出系統的活動循序圖，如下圖所示。從圖中可以看出，用戶端首先通過呼叫 KeyGenerator 的靜態工廠方法 getInstance()得到 KeyGenerator 的單例實例，與此同時 KeyInfo 物件被建構。

然後用戶端呼叫 KeyGenerator 物件的 getNextKey()方法，而 KeyGenerator 物件則將呼叫委派給 KeyInfo 物件的 getNextKey()方法。KeyInfo 物件則通過資料庫呼叫，將查詢所得的鍵值傳回給用戶端。



現在，這個鍵值生成器已經具有如下的功能：在整個系統中是惟一的，能將生成過的鍵值存儲到資料庫中，以便在系統重新啓動時也能夠繼續鍵值的生成，而不會造成鍵值上的重複。

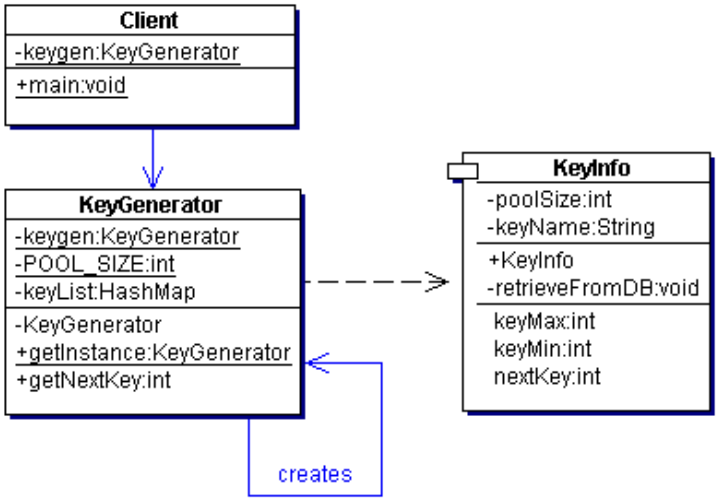
這本來已經足夠好了，但是還有一點值得設計師考慮改進的是，一般的系統都不會只有一個鍵值，而是有多個鍵值需要生成。怎麼讓上面的設計適用於任意多個鍵值的情況呢？

首先，由於 KeyGenerator 是單例類別，因此，給出多個 KeyGenerator 的實例並無可能，除非將之推廣爲多例類別。關於這種可能性，請讀者見本章後面的“多例樣式的應用”一節。

其次，雖然 KeyGenerator 是單例類別，但是 KeyGenerator 仍然可以在內部使用一個聚集管理多個鍵值。換言之，可以使用一個本身是單例物件的聚集物件，配合上合適的介面達到目的。下面本書就首先考慮這一方案。

方案四：有暫存的多序列鍵生成器

方案四是對方案三的改進。在本方案中，同樣使用 KeyInfo 物件存儲某一個鍵的資訊。與方案三相比，本方案引進了一個聚集用來存儲不同序列鍵資訊的 KeyInfo 物件，如下圖所示。



可以看出，KeyGenerator 類別仍然是一個單例類別。它提供了私有的建構方法和一個靜態工廠方法，向外界提供自己惟一的實例。

下面就是 KeyGenerator 類別的程式碼。

程式碼清單 10：KeyGenerator 的程式碼

```
package com.javapatterns.keygen.ver4;
import java.util.HashMap;
public class KeyGenerator
{
    private static KeyGenerator keygen =
          new KeyGenerator();
    private static final int POOL_SIZE = 20;
    private HashMap keyList = new HashMap(10);
   /**
    *   私有建構方法，保證外界無法直接實例化
```

```
        */
        private KeyGenerator()
        {
        }
       /**
        *   靜態工廠方法，提供自己的實例
        */
        public static KeyGenerator getInstance()
        {
            return keygen;
        }
       /**
        *   取值方法，提供下一個合適的鍵值
        */
        public synchronized int getNextKey(String keyName)
        {
            KeyInfo keyinfo ;
            if ( keyList.containsKey(keyName) )
            {
                keyinfo = (KeyInfo) keyList.get(keyName);
                System.out.println("key found");
            }
            else
            {
                keyinfo = new KeyInfo(POOL_SIZE, keyName);
                keyList.put(keyName, keyinfo);
                System.out.println("new key created");
            }
            return keyinfo.getNextKey(keyName);
        }
    }
```

下面就是 KeyInfo 類別的程式碼。這個類別存儲了鍵名、緩衝的大小及在這個緩衝區內的最小鍵值、最大鍵值、當前鍵值等資訊。這個類別還提供了一個 retrieveFromDB()方法，向資料庫查詢鍵值。

程式碼清單 11：KeyInfo 的程式碼

```
package com.javapatterns.keygen.ver4;
class KeyInfo
{
    private int keyMax;
    private int keyMin;
    private int nextKey;
    private int poolSize;
    private String keyName;
   /**
    *   建構方法
```

```
 */
 public KeyInfo(int poolSize, String keyName)
 {
       this.poolSize = poolSize;
       this.keyName = keyName;
       retrieveFromDB();
 }
/**
 *   取值方法，提供鍵的最大值
 */
 public int getKeyMax()
 {
       return keyMax;
 }
/**
 *   取值方法，提供鍵的最小值
 */
 public int getKeyMin()
 {
       return keyMin;
 }
/**
 *   取值方法，提供鍵的當前值
 */
 public int getNextKey()
 {
       if (nextKey > keyMax)
       {
             retrieveFromDB();
       }
       return nextKey++;
 }
/**
 *   內部方法，從資料庫提取鍵的當前值
 */
 private void retrieveFromDB()
 {
       String sql1 = "UPDATE KeyTable SET keyValue = keyValue + "
             + poolSize + " WHERE keyName = '"
             + keyName + "'";
       String sql2 = "SELECT keyValue FROM KeyTable WHERE KeyName = '"
             + keyName + "'";
       // execute the above queries in a transaction and commit it
       // assume the value returned is 1000
       int keyFromDB = 1000;
       keyMax = keyFromDB;
```

```
            keyMin = keyFromDB - poolSize + 1;
            nextKey = keyMin;
        }
    }
```

從上面的程式碼可以看出，每當 getNextKey()被呼叫時，這個方法都會根據緩衝區的大小和已經用過的鍵值來判斷是否需要更新緩衝區。當緩衝區被更新後，KeyInfo 會持有已經向資料庫預定過的 20 個序列號碼，並不斷向呼叫者順序提供這 20 個號碼。等這 20 個序列號碼用完之後，KeyInfo 物件就會向資料庫預定後 20 個新號碼。

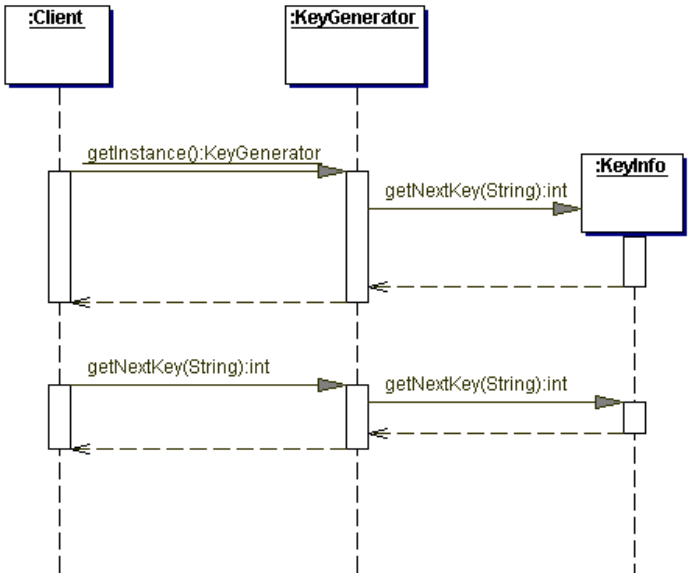
當然，如果系統被重新啓動，而緩衝區中的號碼並沒有用完的話，這些沒有用完的號碼就不會再次被使用了。系統重新啓動之後，KeyInfo 物件會重新向資料庫預定下面的 20 個號碼，並向外界提供這 20 個號碼。

下面就是一個示意性的用戶端 Client 類別的程式碼。

程式碼清單 12：用戶端的程式碼

```
package com.javapatterns.keygen.ver4;
public class Client
{
    private static KeyGenerator keygen;
    public static void main(String[] args)
    {
        keygen = KeyGenerator.getInstance();
        for (int i = 0 ; i < 20 ; i++)
        {
            System.out.println("key(" + (i+1)
                + ")= " + keygen.getNextKey("PO_NUMBER"));
        }
    }
}
```

爲了說明系統的活動時序，這裏特地給出系統的活動循序圖，如下圖所示。從圖中可以看出，用戶端首先通過呼叫 KeyGenerator 的靜態工廠方法 getInstance()得到 KeyGenerator 的單例實例，與此同時 KeyInfo 物件被建構。與方案三的情況不同的是，這裏的 KeyInfo 的建構方法接收鍵名作爲參數。

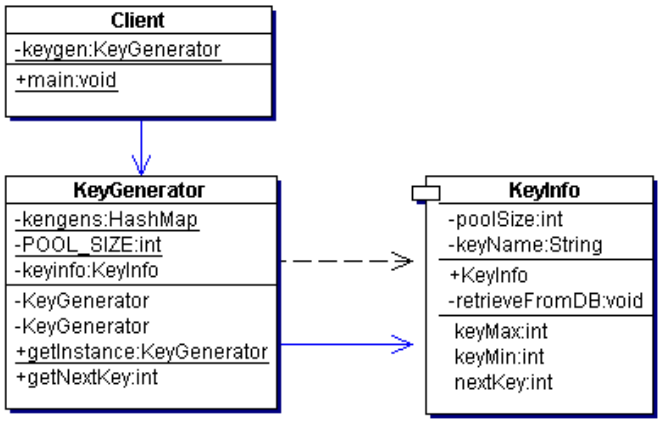


然後用戶端呼叫 KeyGenerator 物件的 getNextKey()方法，而 KeyGenerator 物件則將呼叫委派給 KeyInfo 物件的 getNextKey()方法。KeyInfo 物件則通過資料庫呼叫，將查詢所得的鍵值傳回給用戶端。與方案三的情況不同的是，這裏的兩個 getNextKey()方法都接收鍵名作爲參數。

運行的結果與上一個設計方案類似，所以不再重複。

18.3　將多例樣式應用到系統設計中

正如前面所談到的，爲了能夠處理多系列鍵值的情況，除了可以將單例樣式所封裝的單一狀態改爲聚集狀態之外，還可以採用多例樣式。多例樣式允許一個類別有多個實例，這些實例有各自不同的內蘊狀態。這就是本書給出的第五個方案，應用多例樣式的設計方案。下圖所示就是這個設計方案的類別圖結構。



下面是 KeyGenerator 類別的程式碼。可以看出，這是一個多例類別。每一個 KeyGenerator 物件都持有一個特定的 KeyInfo 物件作爲內蘊狀態。用戶端可以使用這個類別的靜態工廠方法得到所需要的實例，而這個工廠方法會首先查看做登記用的 keygens 聚集。如果所要求的鍵名在聚集裏面，就直接將這個鍵名所對應的實例傳回給用戶端；如果所要求的鍵名不在聚集裏面，就需要建構一個新的實例，對應於這個鍵名，然後將這個事例登記到聚集裏面，再傳回給用戶端。

程式碼清單 13：用戶端的程式碼

```
package com.javapatterns.keygen.ver5;
import java.util.HashMap;
public class KeyGenerator
{
    private static HashMap kengens
        = new HashMap(10);
    private static final int POOL_SIZE = 20;
    private KeyInfo keyinfo;
   /**
    *   私有建構方法，保證外界無法直接實例化
    */
    private KeyGenerator()
    {
    }
   /**
    *   私有建構方法，保證外界無法直接實例化
    */
    private KeyGenerator(String keyName)
    {
        keyinfo = new KeyInfo(POOL_SIZE, keyName);
    }
   /**
    *      靜態工廠方法，提供自己的實例
    */
    public static synchronized KeyGenerator
        getInstance(String keyName)
    {
        KeyGenerator keygen;
        if (kengens.containsKey(keyName))
        {
            keygen = (KeyGenerator)
                kengens.get(keyName);
        }
        else
        {
            keygen = new KeyGenerator(keyName);
        }
        return keygen;
```

```
    }
    /**
     *      取值方法，提供下一個合適的鍵值
     */
    public synchronized int getNextKey()
    {
        return keyinfo.getNextKey();
    }
}
```

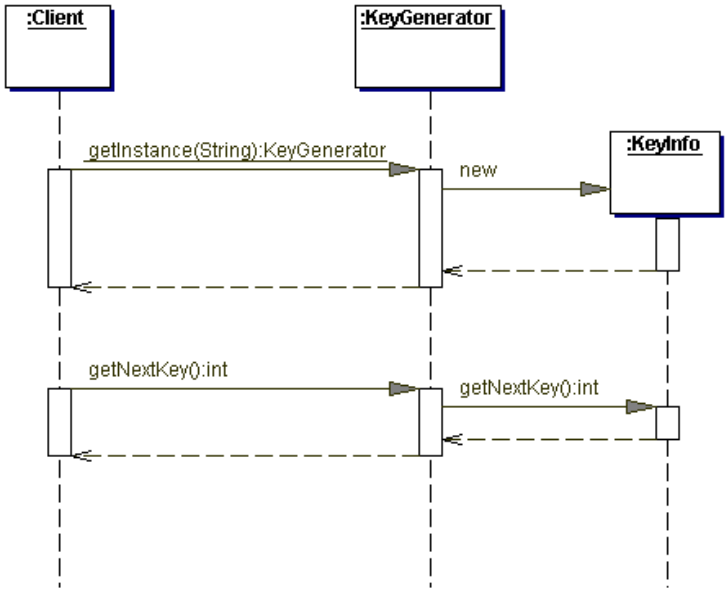
在這個設計裏面 KeyInfo 類別與設計方案四中的 KeyInfo 類別並無區別，因而省略。

下面是一個示意性的用戶端的程式碼。與方案四相比，這裏在呼叫 KeyGenerator 的工廠方法時，傳入序列鍵的名字作爲參數；而在呼叫 getNextKey()方法時，則不需要傳入序列鍵的名字作爲參數。

程式碼清單 14：用戶端的程式碼

```
package com.javapatterns.keygen.ver5;
public class Client
{
    private static KeyGenerator keygen;
    public static void main(String[] args)
    {
        keygen = KeyGenerator.getInstance("PO_NUMBER");
        for (int i = 0 ; i < 20 ; i++)
        {
            System.out.println("key(" + (i+1)
                + ")= " + keygen.getNextKey());
        }
    }
}
```

可以看出，上面這個用戶端物件首先建構一個多例類別的實例，這個實例是以鍵名爲內蘊狀態的。然後就可以通過呼叫這個實例的 getNextKey()方法，得到所需的鍵值。這個系統的循序圖如下所示。



從圖中可以看出，用戶端首先通過呼叫多例類別 KeyGenerator 的靜態工廠方法 getInstance()得到一個多例類別實例，與此同時 KeyGenerator 會建構一個 KeyInfo 物件。與方案四的情況不同的是，這裏的靜態工廠方法接收鍵名作爲參數。

由於每一個多例物件都僅持有一個鍵名的 KeyInfo 物件，所以用戶端可以向方案三一樣呼叫 KeyGenerator 物件的 getNextKey()方法，得到所需的鍵值並傳回給用戶端。

運行的結果與上一個設計方案並無不同，所以不再重複。

18.4 討　　論

在上面給出的方案中，第四個和第五個方案都是具有實用價值的設計方案。讀者可以嘗試著將這兩個方案應用到自己的設計中去，並根據具體的要求，將設計方案進一步完善。

如果一個單例樣式是一個聚集物件的話，那麼這個聚集中所保存的是對其他物件的引用。一個多例樣式則不同，多例物件使用一個聚集物件登記和保存自身的實例。由於這兩種設計樣式的相似之處，在很多情況下它們可以互換使用。本章所給出的設計方案四和設計方案五就是建立在單例聚集物件和多例物件的基礎之上的實作了相同功能的兩種不同設計。

在方案四裏面，KeyGenerator 物件是一個單例物件，同時也是一個聚集，而聚集中所存儲的是 KeyInfo 物件。在方案五中，KeyGenerator 物件是一個多例物件，當然也是一個聚集物件，這個聚集中存儲的是這個多例物件自身。

對於用戶端來說，兩種設計的區別並不大。使用方案四時，用戶端建構一個單例物件，然後根據鍵名呼叫這個物件的 getNextKey()方法從聚集中取出這個鍵名所對應的 KeyInfo 物件；如果聚集中沒有這個 KeyInfo 物件，就建構一個新的物件，先將其存儲到聚集中，然後傳回給呼叫者。具體地講，KeyInfo 會將鍵名和 KeyInfo 物件作爲 HashMap 的鍵和物

件存儲在 HashMap 裏面。

而當使用方案五時，用戶端根據鍵名建構一個多例物件，這個物件以鍵名爲內蘊狀態。多例物件會將鍵名和自身的實例當做 HashMap 的鍵和物件存儲在內部的 HashMap 物件裏面。當用戶端通過靜態工廠方法請求 KeyGenerator 的實例時，會將所要求的鍵名傳入；而在接到請求之後，KeyGenerator 首先會在自己的登記聚集中查找是否已經有這樣一個滿足要求的 KeyGenerator 物件。如果有，就將之提供給用戶端；如果沒有，就立即建構一個滿足要求的 KeyGenerator 物件，將之登記到聚集裏面，然後傳回給用戶端。