第 18 章　专题：序列键生成器与单例及多例模式

本章的内容来自于一个真实的全球金融网站项目，本书去掉了所有与商业有关的内容，仅仅讨论技术观点。

18.1　问　　题

序列键

开发过数据库驱动信息系统的读者都知道，在一个关系数据库中，所有的数据都是存储在表里的；而现代的数据库设计要求每一个表都有一个主键（Primary Key）。对大多数的用户输入数据来讲，主键需要由系统以序列号方式产生，而不是由操作人员给出。

有一些关系数据库引擎提供某种序列键生成机制。比如，Microsoft SQL Server 提供一个 AutoNumber 的属性，允许每一个表内可以有一个 AutoNumber 列；Oracle 提供 Sequence 对象，可以提供序列键值。下面的 SQL 语句会建立一个名为 ISSUE_SEQ 的 Sequence 对象，其开始值为 1000，每次增加 1。

代码清单 1：用来创建一个 Oracle 序列的 SQL 语句源代码

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

其他一些数据库引擎则没有相应的机制，比如 Sybase 就没有类似的功能。这时候就只好使用一个表，并在其内部设有两个列，一个列存放键名，另一个列存放键值，客户端使用 SQL 语句自行管理键值。使用一个表来存储所有的序列键，如下表所示。

KeyName	KeyValue
PO_NUMBER	1045
SO_NUMBER	5069

（续表）

ITEM_NUMBER	10874
RMA_NUMBER	2065
…	

有些商业化的系统（如 Vignette Story Server）需要支持几种主要的数据库，包括 Oracle，Microsoft SQL Server 和 Sybase 等。这样的系统就不能使用 Microsoft SQL Server 或者 Oracle 特有的机制，而必须使用某种具有普遍性的机制。这时，上面提到的使用一个表来管理各种主键就变得较为合理。

预定式存储

不论是使用哪一个机制，最终系统必须要有一些数据库操作来提供这些序列值。比如一个餐馆的贩卖系统需要一个序列号给每天开出去的卖单编号，这个序列号码就应当存放到数据库里面。每当发出序列号码的时候，都应当从数据库读取这个号码，并更新这个号码。

为了保证在任何情况下键值都不会出现重复，应当使用预定式键值存储办法。在请求一个键值时，首先将数据库中的键值更新为下一个可用值，然后将旧值提供给客户端。这样万一出现运行中断的话，最多就是这个键值被浪费掉。

与此相对的是记录式键值存储办法。也就是说，键值首先被返还给客户端，然后记录到数据库中去。这样做的缺点是，一旦系统中断，就有可能出现客户端已经使用了一个键值，而这个键值却没有来得及存储到数据库中的情况。在系统重启之后，系统还会从这个已经被使用过的键值开始，从而导致错误。因此不要使用这种登记式的存储办法。

预定式的存储办法可以每一次预定多个键值（也即一个键值区间）；而不是每一次仅仅预定一个值。由于这些值都是一些序列数值，因此，所谓一次预定多个值，不过就是每次更新键值时将键值增加一个大于 1 的数目。在后面的设计方案中，首先考虑每次预定一个键值的做法，然后将之改进为每次预定 20 个值的情况。

单例模式的应用

上面讨论了序列的存储机制，另一个重要的机制是键的查询管理机制。与其将键值的查询工作交给各个模块，不如将之集中到一个对象身上。这个对象负责管理序列键的查询，称之为序列键管理器。

显然，不难看出，整个系统只需要一个序列键管理器对象。由于系统运行期间总是需要序列键，因此序列键管理器对象需要在系统运行期间存在。考虑到可以让一个序列键管理器负责管理分属于不同模块的多个序列键，因此这个序列键管理器需要让整个系统访问。

学习过单例模式的读者会意识到，这个系统设计应当使用到单例模式。是的，这个序列键管理器可以设计成一个单例类。

一个客户端系统往往需要管理不止一个键值，而是多个键值。这时候，可以将这个单

例对象的内部状态扩展成为一个聚集，从而可以存储任意多个键值。也就是说，这个序列键管理器是一个聚集对象，而此聚集本身是一个单例对象。

关于单例模式，请读者参考本书的“单例（Singleton）模式”一章。

多例模式的应用

多例模式往往持有一个内蕴状态；多例类的每一个实例都有独特的内蕴状态。一个多例类持有一个聚集对象，用来登记自身的实例，而其内蕴状态往往就是登记的键值。当客户端通过多例类的静态工厂方法请求多例类的实例时，这个工厂方法都会在聚集内查询是否已经有一个这样的实例。如果有，就直接返还给客户端；如果没有，就首先创建一个这样的实例，将之登记到聚集中，然后再向客户端提供。

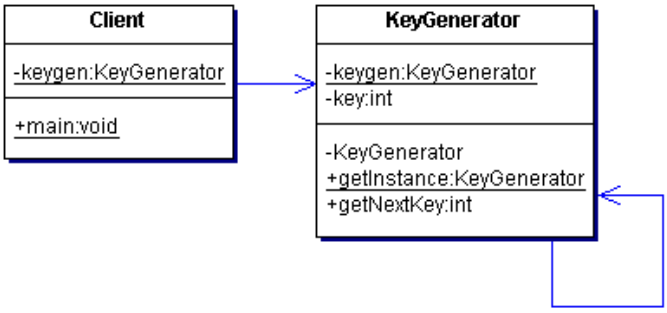
关于多例模式以及它与单例模式的关系，请读者参考本书的“专题：多例（Multiton）模式与多语言支持 ”一章。

18.2　将单例模式应用到系统设计中

下面从一个最简单的情况出发，逐渐将问题的复杂性提高，直到给出具有实用价值的解决方案为止。

方案一：没有数据库的情况

首先考虑一个没有数据库背景的方案，这个设计由一个单例类 KeyGenerator 组成。一个序列键生成器的类图如下图所示。



下面是这个键生成器 KeyGenerator 的源代码。

代码清单 2：KeyGenerator 类的源代码

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

```
            new KeyGenerator();
        private int key = 1000;
        /**
         *   私有构造子，保证外界无法直接实例化
         */
        private KeyGenerator() {}
        /**
         *   静态工厂方法，提供自己的实例
         */
        public static KeyGenerator getInstance()
        {
            return keygen;
        }
        /**
         *   取值方法，提供下一个合适的键值
         */
        public synchronized int getNextKey()
        {
            return key++;
        }
}
```

可以看出，上面的 KeyGenerator 类的构造子是私有的，因此，外界无法通过调用构造子将之实例化。同时，它提供了一个静态的工厂方法 getInstance()，自己向外界提供自己的实例。如果再考查一下这个工厂方法就会发现，这个方法永远仅提供同一个实例。换言之，这是一个单例类。

商业方法 getNextKey()返还一个整型数，这个数会自行递增，每次加 1。

代码清单 3：客户类的源代码

```
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());
    }
}
```

在运行时，客户对象会打印出得到的序列键的数值，这表明系统是正常工作的。

代码清单 4：运行结果

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

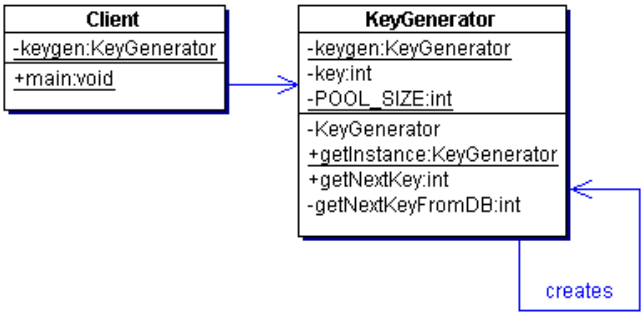
key = 1002

这一设计基本上实现了向客户端提供键值的功能，但是也有明显的缺点。由于没有数据库的存储，一旦系统重新启动，KeyGenerator 都会重新初始化，这就会造成键值的重复。

为了避免这一点，就必须将每次的键值存储起来，以便一旦系统中断和重启时，可以将这个键值取出，并在这个值的基础上重新开始。这就将设计师引向了下一个设计方案。

方案二：有数据库的情况

这个方案是对方案一的修正。与方案一一样，这个设计由一个单例类组成；而与方案一不同的是，这个单例类有数据库功能。它将键值存储在数据库的表中，每次客户端请求键值时，首先将这个表中的值增加 1，然后将这个值返还给客户端（当然，这两个数据库操作应当是一个完整的交易单位）。有数据库的键值生成器的结构图如下图所示。



下面就是 KeyGenerator 类的源代码。这是一个单例类，它提供了私有的构造子，所以外界无法直接将其实例化，所有的实例化请求都必须通过静态工厂方法进行。

代码清单 5：KeyGenerator 的源代码

```
package com.javapatterns.keygen.ver2;
public class KeyGenerator
{
    private static KeyGenerator keygen =
        new KeyGenerator();
  /**
   *  私有构造子，保证外界无法直接实例化
   */
   private KeyGenerator() {}
  /**
   *  静态工厂方法，提供自己的实例
   */
   public static KeyGenerator getInstance()
   {
```

```
            return keygen;
        }
       /**
        *   取值方法，提供下一个合适的键值
        */
        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
            //示意性地返还一个数值
            return 1000;
        }
    }
```

在接到客户端的请求时，这个 KeyGenerator 每次都向数据库查询键值，将新的键值登记到表里，然后将查询的结果返还给客户端。在上面的源代码中，给出了两个 SQL 语句，但是并没有给出执行这两行语句的 JDBC 代码。相信读者在将这个设计应用到自己的系统中时，可以将必要的 JDBC 代码加进去。

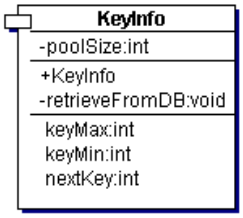
必须指出的是，为了将读者的注意力集中在系统设计上面，本书不想涉及到 JDBC 的细节，因此上面并没有给出这方面的源代码。相信读者在将这个设计应用到自己的系统中去时，可以自行实现这部分代码。

在这个设计方案里面，可以使用与第一个设计方案相同的客户端，因此就不在此重复了。

方案三：键值的缓存方案

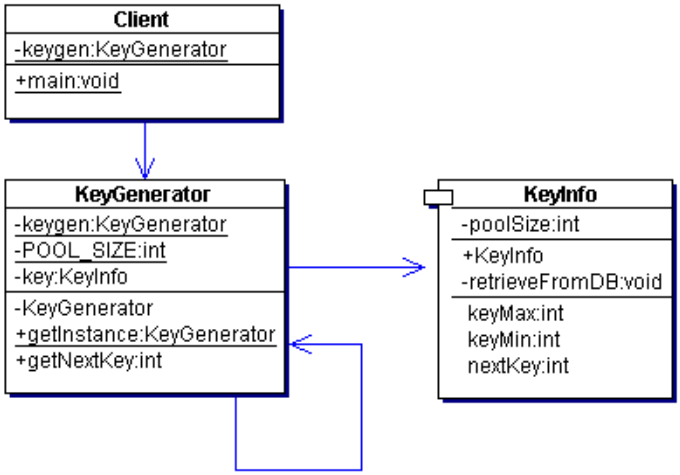
每一次都进行键值的查询有必要吗？毕竟一个键的值只是一些序列号码，与其每接到一次请求就查询一次，然后向客户端提供这一个值，不如在一次查询中一次性地预先登记多个键值，然后连续多次地向客户端提供这些预订的键值。这样一来，不是节省了大部分不必要的数据库查询操作吗？

是的，这就是键值的缓存机制。当 KeyGenerator 每次更新数据库中的键值时，它都将键值增加。与方案二不同之处是，键值的增加值不是 1 而是更多。在下面给出的例子中，键值的增加值是 20。为了存储所有的与键有关的信息，特地引进一个 KeyInfo 类，如下图所示。



这个 KeyInfo 除了存储与键有关的信息外，还提供了一个 retrieveFromDB()方法，向数据库查询键值。每次查询得到的 20 个键值会在随后提供给请求者，直到 20 个键值全部使用完毕，然后再向数据库预定后 20 个键值。

KeyGenerator 作为一个单例类，保持一个对 KeyInfo 对象的引用。客户端调用 getNextKey()方法以得到下一个键的键值。有缓存的序列键生成机制如下图所示。



下面就是 KeyGenerator 类的源代码。与第二个方案一样，KeyGenerator 类使用了私有的构造子，以及一个静态工厂方法向外界提供自己惟一的实例。

代码清单 6：KeyGenerator 的源代码

```
package com.javapatterns.keygen.ver3;
public class KeyGenerator
{
    private static KeyGenerator keygen =
          new KeyGenerator();
    private static final int POOL_SIZE = 20;
    private KeyInfo key ;
   /**
    *   私有构造子，保证外界无法直接实例化
    */
    private KeyGenerator()
    {
          key = new KeyInfo(POOL_SIZE);
```

```
    }
    /**
     *  静态工厂方法，提供自己的实例
     */
    public static KeyGenerator getInstance()
    {
        return keygen;
    }
    /**
     *  取值方法，提供下一个合适的键值
     */
    public synchronized int getNextKey()
    {
        return key.getNextKey();
    }
}
```

下面是 KeyInfo 类的源代码。正如同上面所谈到的，这个类提供了向数据库查询的功能，并且存储一定数目的键值。

代码清单 7：KeyInfo 的源代码

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

```
    {
        return keyMin;
    }
    /**
     *  取值方法，提供键的当前值
     */
    public int getNextKey()
    {
        if (nextKey > keyMax)
        {
            retrieveFromDB();
        }
        return nextKey++;
    }
    /**
     *  内部方法，从数据库提取键的当前值
     */
    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
        //示意性地返还一个数值
        int keyFromDB = 1000;
        keyMax = keyFromDB;
        keyMin = keyFromDB - poolSize + 1;
        nextKey = keyMin;
    }
}
```

下面就是一个示意性的客户端的源代码。

代码清单 8：一个示意性的客户端的源代码

```
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());
        }
```

```
        }
    }
```

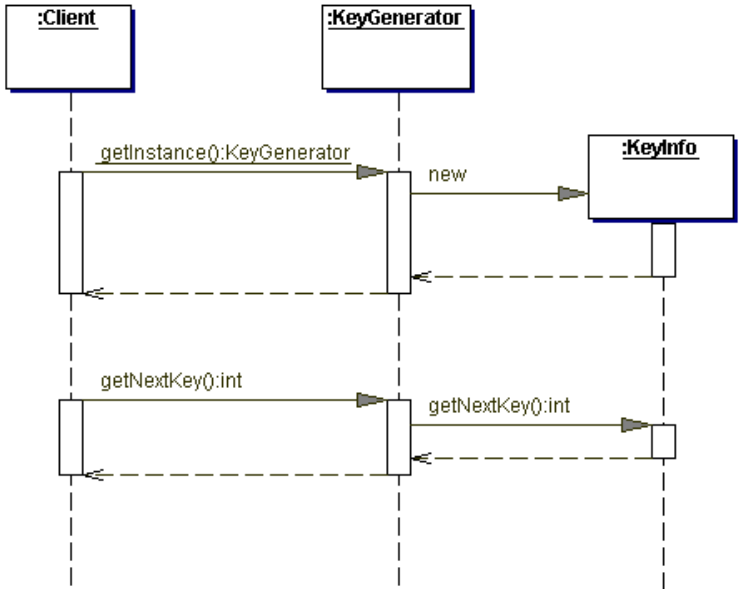
可以看出，这个示意性的客户端首先通过调用 KeyGenerator 的静态工厂方法得到 KeyGenerator 的实例，然后调用 KeyGenerator 对象的取值方法，以得到一个个的键值。运行的结果如下所示。

代码清单 9：系统的运行结果

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

为了说明系统的活动时序，这里特地给出系统的活动序列图，如下图所示。从图中可以看出，客户端首先通过调用 KeyGenerator 的静态工厂方法 getInstance()得到 KeyGenerator 的单例实例，与此同时 KeyInfo 对象被创建。

然后客户端调用 KeyGenerator 对象的 getNextKey()方法，而 KeyGenerator 对象则将调用委派给 KeyInfo 对象的 getNextKey()方法。KeyInfo 对象则通过数据库调用，将查询所得的键值返还给客户端。



现在，这个键值生成器已经具有如下的功能：在整个系统中是惟一的，能将生成过的键值存储到数据库中，以便在系统重新启动时也能够继续键值的生成，而不会造成键值上的重复。

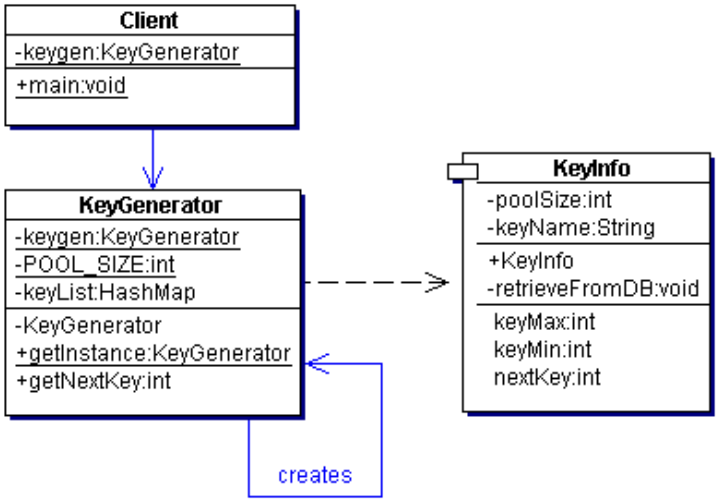
这本来已经足够好了，但是还有一点值得设计师考虑改进的是，一般的系统都不会只有一个键值，而是有多个键值需要生成。怎么让上面的设计适用于任意多个键值的情况呢？

首先，由于 KeyGenerator 是单例类，因此，给出多个 KeyGenerator 的实例并无可能，除非将之推广为多例类。关于这种可能性，请读者见本章后面的“多例模式的应用”一节。

其次，虽然 KeyGenerator 是单例类，但是 KeyGenerator 仍然可以在内部使用一个聚集管理多个键值。换言之，可以使用一个本身是单例对象的聚集对象，配合上合适的接口达到目的。下面本书就首先考虑这一方案。

方案四：有缓存的多序列键生成器

方案四是对方案三的改进。在本方案中，同样使用 KeyInfo 对象存储某一个键的信息。与方案三相比，本方案引进了一个聚集用来存储不同序列键信息的 KeyInfo 对象，如下图所示。



可以看出，KeyGenerator 类仍然是一个单例类。它提供了私有的构造子和一个静态工厂方法，向外界提供自己惟一的实例。

下面就是 KeyGenerator 类的源代码。

代码清单 10：KeyGenerator 的源代码

```
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);
   /**
    *   私有构造子，保证外界无法直接实例化
    */
```

```
    private KeyGenerator()
    {
    }
   /**
    *  静态工厂方法，提供自己的实例
    */
    public static KeyGenerator getInstance()
    {
        return keygen;
    }
   /**
    *  取值方法，提供下一个合适的键值
    */
    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 类的源代码。这个类存储了键名、缓冲的大小及在这个缓冲区内的最小键值、最大键值、当前键值等信息。这个类还提供了一个 retrieveFromDB()方法，向数据库查询键值。

代码清单 11：eyInfo 的源代码

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

```
 public KeyInfo(int poolSize, String keyName)
 {
       this.poolSize = poolSize;
       this.keyName = keyName;
       retrieveFromDB();
 }
/**
 *   取值方法，提供键的最大值
 */
 public int getKeyMax()
 {
       return keyMax;
 }
/**
 *   取值方法，提供键的最小值
 */
 public int getKeyMin()
 {
       return keyMin;
 }
/**
 *   取值方法，提供键的当前值
 */
 public int getNextKey()
 {
       if (nextKey > keyMax)
       {
             retrieveFromDB();
       }
       return nextKey++;
 }
/**
 *   内部方法，从数据库提取键的当前值
 */
 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;
        }
    }
```

从上面的源代码可以看出，每当 getNextKey()被调用时，这个方法都会根据缓冲区的大小和已经用过的键值来判断是否需要更新缓冲区。当缓冲区被更新后，KeyInfo 会持有已经向数据库预定过的 20 个序列号码，并不断向调用者顺序提供这 20 个号码。等这 20 个序列号码用完之后，KeyInfo 对象就会向数据库预定后 20 个新号码。

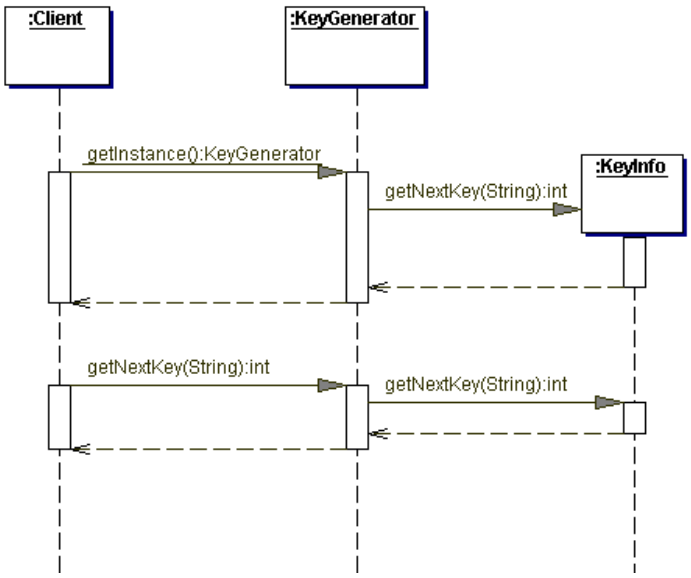
当然，如果系统被重新启动，而缓冲区中的号码并没有用完的话，这些没有用完的号码就不会再次被使用了。系统重新启动之后，KeyInfo 对象会重新向数据库预定下面的 20 个号码，并向外界提供这 20 个号码。

下面就是一个示意性的客户端 Client 类的源代码。

代码清单 12：客户端的源代码

```
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"));
        }
    }
}
```

为了说明系统的活动时序，下这里特地给出系统的活动序列图，如下图所示。从图中可以看出，客户端首先通过调用 KeyGenerator 的静态工厂方法 getInstance()得到 KeyGenerator 的单例实例，与此同时 KeyInfo 对象被创建。与方案三的情况不同的是，这里的 KeyInfo 的构造子接收键名作为参量。

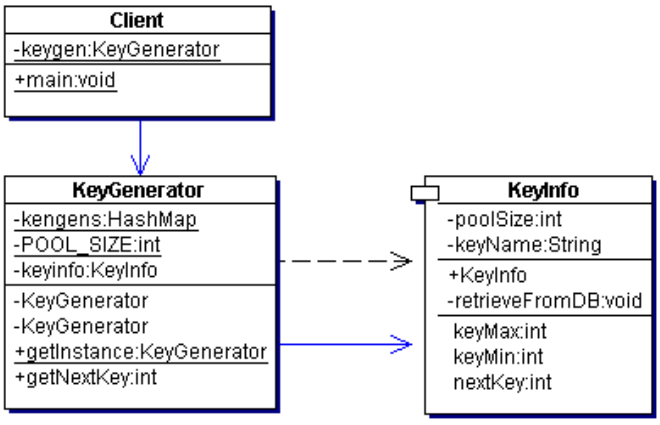


然后客户端调用 KeyGenerator 对象的 getNextKey()方法，而 KeyGenerator 对象则将调用委派给 KeyInfo 对象的 getNextKey()方法。KeyInfo 对象则通过数据库调用，将查询所得的键值返还给客户端。与方案三的情况不同的是，这里的两个 getNextKey()方法都接收键名作为参量。

运行的结果与上一个设计方案类似，所以不再重复。

18.3　将多例模式应用到系统设计中

正如前面所谈到的，为了能够处理多系列键值的情况，除了可以将单例模式所封装的单一状态改为聚集状态之外，还可以采用多例模式。多例模式允许一个类有多个实例，这些实例有各自不同的内蕴状态。这就是本书给出的第五个方案，应用多例模式的设计方案。下图所示就是这个设计方案的类图结构。



下面是 KeyGenerator 类的源代码。可以看出，这是一个多例类。每一个 KeyGenerator 对象都持有一个特定的 KeyInfo 对象作为内蕴状态。客户端可以使用这个类的静态工厂方法得到所需要的实例，而这个工厂方法会首先查看做登记用的 keygens 聚集。如果所要求的键名在聚集里面，就直接将这个键名所对应的实例返还给客户端；如果所要求的键名不在聚集里面，就需要创建一个新的实例，对应于这个键名，然后将这个事例登记到聚集里面，再返还给客户端。

代码清单 13：客户端的源代码

```
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;
   /**
    *   私有构造子，保证外界无法直接实例化
    */
    private KeyGenerator()
    {
    }
   /**
    *   私有构造子，保证外界无法直接实例化
    */
    private KeyGenerator(String keyName)
    {
        keyinfo = new KeyInfo(POOL_SIZE, keyName);
    }
   /**
    *      静态工厂方法，提供自己的实例
    */
    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，提供下一个合适的键值
     */
    public synchronized int getNextKey()
    {
        return keyinfo.getNextKey();
    }
}
```

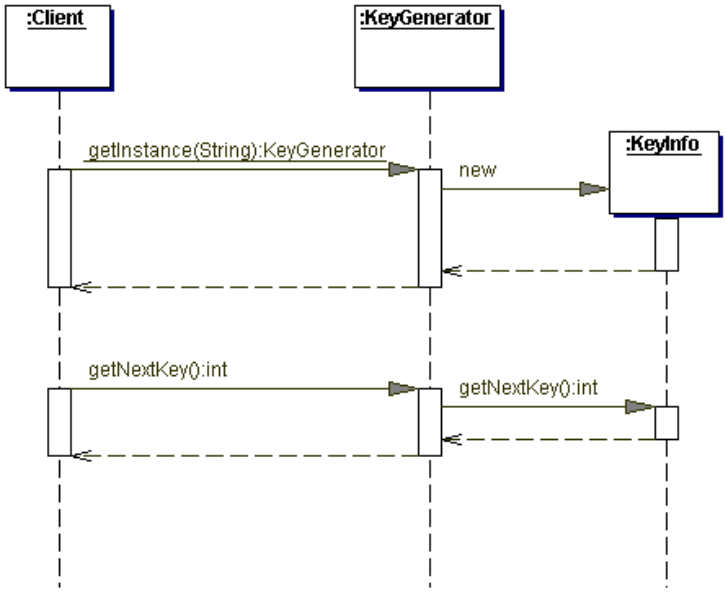
在这个设计里面 KeyInfo 类与设计方案四中的 KeyInfo 类并无区别，因而省略。

下面是一个示意性的客户端的源代码。与方案四相比，这里在调用 KeyGenerator 的工厂方法时，传入序列键的名字作为参数；而在调用 getNextKey()方法时，则不需要传入序列键的名字作为参数。

代码清单 14：客户端的源代码

```
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，上面这个客户端对象首先创建一个多例类的实例，这个实例是以键名为内蕴状态的。然后就可以通过调用这个实例的 getNextKey()方法，得到所需的键值。这个系统的时序图如下所示。



从图中可以看出，客户端首先通过调用多例类 KeyGenerator 的静态工厂方法 getInstance()得到一个多例类实例，与此同时 KeyGenerator 会创建一个 KeyInfo 对象。与方案四的情况不同的是，这里的静态工厂方法接收键名作为参量。

由于每一个多例对象都仅持有一个键名的 KeyInfo 对象，所以客户端可以向方案三一样调用 KeyGenerator 对象的 getNextKey()方法，得到所需的键值并返还给客户端。

运行的结果与上一个设计方案并无不同，所以不再重复。

18.4　讨　　论

在上面给出的方案中，第四个和第五个方案都是具有实用价值的设计方案。读者可以尝试着将这两个方案应用到自己的设计中去，并根据具体的要求，将设计方案进一步完善。

如果一个单例模式是一个聚集对象的话，那么这个聚集中所保存的是对其他对象的引用。一个多例模式则不同，多例对象使用一个聚集对象登记和保存自身的实例。由于这两种设计模式的相似之处，在很多情况下它们可以互换使用。本章所给出的设计方案四和设计方案五就是建立在单例聚集对象和多例对象的基础之上的实现了相同功能的两种不同设计。

在方案四里面，KeyGenerator 对象是一个单例对象，同时也是一个聚集，而聚集中所存储的是 KeyInfo 对象。在方案五中，KeyGenerator 对象是一个多例对象，当然也是一个聚集对象，这个聚集中存储的是这个多例对象自身。

对于客户端来说，两种设计的区别并不大。使用方案四时，客户端创建一个单例对象，然后根据键名调用这个对象的 getNextKey()方法从聚集中取出这个键名所对应的 KeyInfo 对象；如果聚集中没有这个 KeyInfo 对象，就创建一个新的对象，先将其存储到聚集中，然后返还给调用者。具体地讲，KeyInfo 会将键名和 KeyInfo 对象作为 HashMap 的键和对象存储在 HashMap 里面。

而当使用方案五时，客户端根据键名创建一个多例对象，这个对象以键名为内蕴状态。多例对象会将键名和自身的实例当做 HashMap 的键和对象存储在内部的 HashMap 对象里面。当客户端通过静态工厂方法请求 KeyGenerator 的实例时，会将所要求的键名传入；而在接到请求之后，KeyGenerator 首先会在自己的登记聚集中查找是否已经有这样一个满足要求的 KeyGenerator 对象。如果有，就将之提供给客户端；如果没有，就立即创建一个满足要求的 KeyGenerator 对象，将之登记到聚集里面，然后返还给客户端。