第 16 章　专题：单例模式与 MX 记录

本章的内容来自一个真实的华尔街金融网站项目。

本章假设读者对使用 JavaMail 库通过 SMTP（Simple Mail Transfer Protocol）服务器发送电子邮件有所了解，并且已经阅读过本书的“单例（Singleton）模式”一章。在运行本章提供的代码之前，请到 www.javasoft.com 下载最新版的 JavaMail 库。

16.1　问题与解决方案

问题

相信很多读者都接触过可以自动发送电子邮件的系统，大家想必知道可以利用 JavaMail 库通过一个 SMTP 服务器发送电子邮件的工作。一般来说，一个公司会有数个 SMTP 服务器，虽然每一台服务器都会定期停机维护检查，但是公司总会保持至少一台 SMTP 服务器正常运行，以便处理发送电子邮件的工作。

假设一个公司有一个 J2EE 系统需要自动发送电子邮件，而且这项工作必须是 24/7 运行的，也就是说是全天候的。换言之，不能因为某一台 SMTP 服务器停机就停止发送。与此同时，SMTP 服务器又不能不定时停机维护。这时候就需要系统架构设计师解决这个问题。

于是项目经理将几个设计师请到会议室，并说明问题所在，并要求设计师提供可能的设计方案。经过短暂的讨论，设计师们提出了两个可选方案。

第一个方案

安装一台自己的 SMTP 服务器，由系统维护人员维护这台服务器。这台服务器不一定要直接将电子邮件投递到客户那里，它只需要将电子邮件转投给公司里正在运行的 SMTP 服务器就可以了。由于 SMTP 服务器可以从 DNS 服务器那里拿到正在工作的 SMTP 服务器的名字，因此可以保证邮件总可以发送出去。

这当然是可以的，只是维护一台 SMTP 服务器等于添加了额外的维护工作，而且谁能保证这台服务器不需要停机检修呢？

第二个方案

在方案一中需要自己维护的那个 SMTP 服务器（以下称为源服务器）之所以能够找到正在运行的 SMTP 服务器（以下称为目标服务器），并将邮件转投给它，是因为源服务器能够利用 DNS 的目录服务进行目录查询。DNS 会向整个网络提供所有登记过的 SMTP 服务器的名字，这样源服务器可以一个一个地试验目标服务器清单上的所有名字，直到找到一台正常运行的目标服务器为止。

使用 Java 的 JNDI 功能，Java 程序可以做同样的事情。这也就是说，可以写一个 Java 程序自动从 DNS 服务器那里得到一个公司内登记过的所有的 SMTP 服务器的清单（称为 MX 记录），然后让它一个个地试验清单上所有的服务器，直到把邮件送出去为止。由于公司内总会有至少一台 SMTP 服务器是正常工作的，这样就可以保证总可以将邮件送出去。

第二个方案的进一步完善

如果只需要多写 20 行代码，就可以省去日复一日的服务器维护工作，这当然是再好不过的。于是，这个方案被列为首选方案加以讨论，以便能够最终得到一个完善的解决方案。

下面就是综合讨论中提出的各种意见达成的设计方案：

（1）既然需要 JNDI 和 DNS，那么不妨到 Sun Microsystem 的网站去下载最新的库。

（2）不应当每一次发送邮件时都做 DNS 查询，应当创建一个类负责查询和保存查询所得的结果。

（3）MX 记录是一个相当静态的表，所以整个系统只需要一份表。系统在任何时候需要发送电子邮件，就只需向这个对象调用这份列表即可。

所有这些都指向单例模式。

单例模式的使用

不能忘记本书引入这个例子是为了说明单例模式的使用，因此，有必要借此机会强调一下为什么这里应当使用单例模式，以及在这里使用单例模式应当注意的地方。

首先，正如设计师们在讨论中所指出的，在整个系统中只需要一份这样的 MX 记录表。这当然就是使用单例模式的最重要的理由。

其次，发送电子邮件的操作可能随时触发。也就是说，这个保存 MX 记录表的对象应当在全运行时期时间存在，而不应当被垃圾收集器所收集。了解垃圾收集器的读者可能会意识到，这就需要系统内保持至少一个对它的引用。那么由谁来保存对这个对象的引用，以保证它不被收集呢？当然是它自己最好。这样，只要这个单例对象一旦被创建，就会永远不会收集，直到服务器环境被重启为止。这也就是在这里使用单例模式的另一个重要的理由。

因此，单例模式最适合使用在这个系统设计中。

为了给不熟悉 JNDI，DNS 和 MX 记录等概念的读者一个熟悉这些概念的机会，在进行系统设计之前对这些概念进行一下复习。已经熟悉这些概念的读者可以跳过下面的几个小节，直接阅读“系统设计”一节。

16.2　目录服务与 MX 记录

命名–目录服务

一个生活中的命名–目录服务的例子就是电话台提供的电话目录查询服务。这个服务将把列在电话簿上的商家名字、地址和电话联系起来。

计算机的命名–目录服务是计算机系统的一个基本工具，而且比电话目录服务更加强大。命名–目录服务将一个对象与一个名字联系起来，使得客户可以通过对象的名字找到这个对象。目录服务允许对象有属性，这样客户端通过查询找到对象后，可以进一步得到对象的属性，或者反过来根据属性查询对象。

最常见的目录服务包括 LDAP（Lightweight Directory Access Protocol）和 DNS（Domain Name Service）。

什么是 DNS

DNS 即域名服务（Domain Name Service），电脑用户在网络上找到其他的电脑用户必须通过域名服务。在国际网络以及任何一个建立在 TCP/IP 基础之上的网络上，每一台电脑都有一个惟一的 IP 地址（Internet Protocol Address）。这些 IP 地址就像街道上的门牌号码，使得其他的电脑能找到某一台电脑。DNS 服务器提供 DNS 服务。

MX 记录（MX Record）

MX（Mail Exchange）记录就是邮件交换记录。电子邮件服务器记录指定一台服务器接收某个域名的邮件。也就是说，发往 jeffcorp.com 的邮件将发往 mail.jeffcorp.com 的服务器，完成此项任务的 MX 记录应当像下面这样

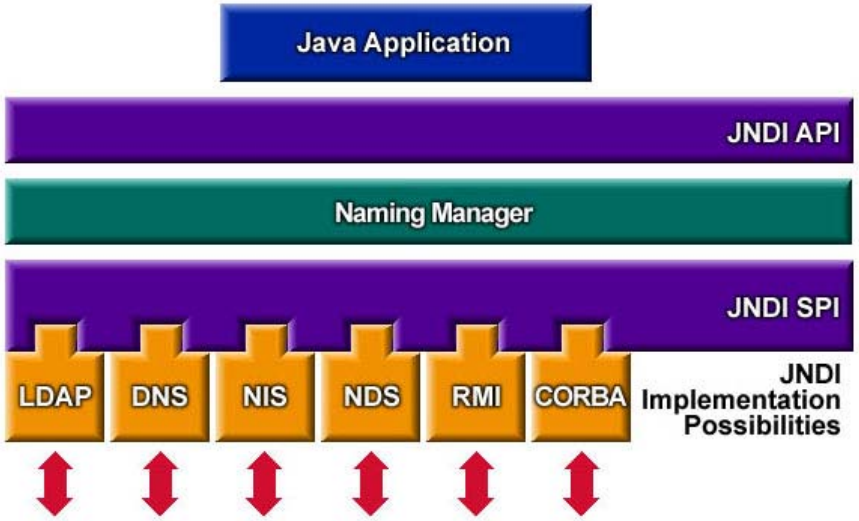
```
jeffcorp.com. IN MX 10 mail.jeffcorp.com
```

在上面的这个记录的最左边是国际网络上使用的电子邮件域名；第三列是一个数字 10，它代表此服务器的优先权是 10。通常来说，一个大型的系统会有数台电子邮件服务器，这些服务器可以依照优先权作为候补服务器使用。优先权必须是一个正整数，这个数字越低，表明优先权越高。

16.3　JNDI 架构介绍

JNDI 的全称是 Java 命名和地址界面（Java Naming and Directory Interface），是在 1997 年初由 Sun Microsystem 引进的，其目的是为 Java 系统提供支持各种目录类型的一个一般性的访问界面。

JNDI 架构由 JNDI API 和 JNDI SPI 组成。JNDI API 使得一个 Java 应用程序可以使用一系列的命名（naming）和目录（directory）服务。JNDI SPI 是为服务提供商，包括目录服务提供商准备的，它可以让各种命名和目录服务能够以对应用程序透明的方式插入到系统里。在 JNDI 架构图中，给出了几个命名和目录服务作为例子，如下图所示。



JNDI API 由下面的四个库组成：

- javax.naming：包括了使用命名服务的类和接口。
- javax.naming.directory：扩展了核心库 javax.naming 的功能，提供目录访问功能。
- javax.naming.event：包括了命名和目录服务中所需要的事件通知机制的类和接口。
- javax.naming.ldap：包括了命名和目录服务中支持 LDAP（v3）所需要的类和接口。

构成了 JNDI SPI 的库为 javax.naming.spi 包括的类和接口允许各种的命名和目录服务提供商能将自己的软件服务构件加入到 JNDI 架构中去。

在 JNDI 里，所有的命名和目录操作都是相对于某一个 context（环境）而言的，JNDI 并没有任何的绝对根目录。JNDI 定义一个初始环境对象，称为 InitialContext，来提供命名和目录操作的起始点。一旦得到了初始环境，就可以使用初始环境查询其他环境和对象。

16.4　如何使用 JNDI 编程

本节提供一个非常简洁的介绍，讲解如何在 Java 中调用 JNDI 的功能。

建立开发环境

首先，如果读者没有最新版的 JNDI 和 DNS 库的话，请到 Sun Microsystem 的网站下载 JNDI 1.2.1 版和 DNS 1.2 版，然后将以下的三个 jar 文件放到 classpath 上面：

- jndi.jar
- dns.jar
- providerutil.jar

本章的代码需要以上三个包才能正常运行。

建立 Java 源文件

现在请读者建立一个 MXTest.java 文件，并加入以下的内容。

代码清单 1：需要导入的类

```
import java.util.Hashtable;
import javax.naming.NamingEnumeration;
import javax.naming.directory.DirContext;
import javax.naming.directory.InitialDirContext;
import javax.naming.directory.Attributes;
import javax.naming.directory.Attribute;
import javax.naming.NamingException;
```

这会把下面编程所需要的类导入进来。本程序大部分的代码都在本程序的 main()方法中。

创建环境对象

在创建初始环境之前，必须指明两个环境性质参数。第一个环境参数是服务提供商名，即 java.naming.factory.initial，这个性质的值应当是 com.sun.jndi.dns.DnsContextFactory，也就是 DNS 服务类的名字。有了这个类名，程序就可以动态地加载这个驱动类。第二个性质参数就是 DNS 服务器的 URL，即 java.naming.provider.url，这个性质的值应当是所在网络的一个合法 DNS 服务器的名字，对于本书作者来说就是 dns://dns01390.ny.jeffcorp.com。

代码清单 2：需要指明的系统性质参数

```
Hashtable env = new Hashtable();
```

```
env.put("java.naming.factory.initial",
        "com.sun.jndi.dns.DnsContextFactory");
env.put( "java.naming.provider.url",
         "dns://dns01390.ny.jeffcorp.com");
```

这时就可以创建初始环境了。

代码清单 3：创建环境对象

```
DirContext dirContext = new InitialDirContext(env);
```

读取环境对象的属性

有了环境对象，就可以进行目录的属性查询了。一个环境对象的 getAttributes()方法会返还这个环境对象的属性。由于这里需要的是某个域的所有 MX 记录，因此，可以使用下面的语句得到这些属性。

代码清单 4：读取环境对象的属性

```
Attributes attrs = dirContext.getAttributes(
    "jeffcorp.com", new String[]{"MX"});
```

在得到这些属性后，可以通过属性对象的遍历方法读出每一条 MX 记录。

完整的程序

下面给出这个程序的完整源代码。

代码清单 5：MXTest 类的源代码

```
import java.util.Hashtable;
import javax.naming.NamingEnumeration;
import javax.naming.directory.DirContext;
import javax.naming.directory.InitialDirContext;
import javax.naming.directory.Attributes;
import javax.naming.directory.Attribute;
import javax.naming.NamingException;
public class MXTest
{
    public static void main(String[] args)
        throws NamingException
    {
        Hashtable env = new Hashtable();
        env.put("java.naming.factory.initial",
            "com.sun.jndi.dns.DnsContextFactory");
        env.put( "java.naming.provider.url",
             "dns://dns01390.ny.jeffcorp.com");
        // 创建环境对象
        DirContext dirContext = new
```

```
                InitialDirContext(env);
            // 读取环境对象的属性
            Attributes attrs = dirContext.getAttributes(
                "jeffcorp.com", new String[]{"MX"});
            for(NamingEnumeration ae = attrs.getAll();
                ae != null && ae.hasMoreElements();)
            {
                Attribute attr = (Attribute)ae.next();
                NamingEnumeration e = attr.getAll();
                while(e.hasMoreElements())
                {
                    String element = e.nextElement().toString();
                    System.out.println(element);
                }
            }
        }
    }
```

运行结果

在运行程序后，打印出一列 SMTP 服务器的名字及优先权。下面就是一个典型的运行结果。

代码清单 6：系统的运行结果

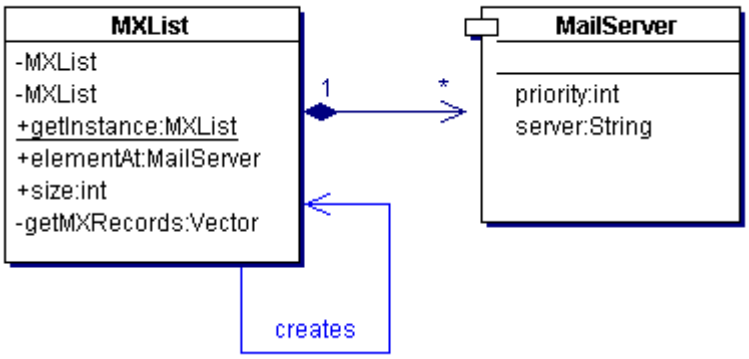
```
10 bird2-a.jeffcorp.com
10 bird2-b.jeffcorp.com
10 horse1-a.jeffcorp.com
10 horse1-b.jeffcorp.com
10 horse2-a.jeffcorp.com
10 horse2-b.jeffcorp.com
10 bird1-a.jeffcorp.com
10 bird1-b.jeffcorp.com
```

16.5　系 统 设 计

现在可以进行系统设计了。首先，系统需要一个单例类负责查询和保存列表。当然由于 SMTP 服务器的信息包括优先权和服务器名字，因此还需要一个 MailServer 类，用于存储关于一个 SMTP 服务器的信息。这样就有了以下所示的两个类。

系统的静态结构

MX 信息查询系统的设计图如下所示。



源代码

下面就是 MailServer 类的源代码，这个类用于存储 SMTP 服务器的信息，包括服务器的名字和优先权。

代码清单 7：MailServer 类的源代码

```
package com.javapatterns.singleton.mxrecord;
public class MailServer
{
    private int priority;
    private String server;
   /**
    * 优先权的赋值方法
    */
    public void setPriority(int priority)
    {
        this.priority = priority;
    }
   /**
    * 服务器名的赋值方法
    */
    public void setServer(String server)
    {
        this.server = server;
    }
   /**
    * 优先权的取值方法
    */
    public int getPriority()
    {
        return priority;
    }
```

```
    /**
     * 服务器名的取值方法
     */
     public String getServer()
     {
          return server;
     }
}
```

下面就是系统的核心类 MXList 的源代码。可以看到，这个类的构造子是私有的，因此，外界不能够直接将此类实例化。但是，这个类提供了一个静态工厂方法 getInstance() 以提供自己的实例。

由于这个工厂方法实际上仅会返还惟一的一个实例，因此这个类是一个单例类。熟悉单例模式的读者可以发现，这个单例类采用的是懒汉式初始化方法。关于单例类及其实现方法，请参见本书的“单例（Singleton）模式”一章。

代码清单 8：MXList 类的源代码

```
package com.javapatterns.singleton.mxrecord;
import java.util.Enumeration;
import java.util.Hashtable;
import java.util.Vector;
import java.util.StringTokenizer;
import javax.naming.NamingEnumeration;
import javax.naming.directory.DirContext;
import javax.naming.directory.InitialDirContext;
import javax.naming.directory.Attributes;
import javax.naming.directory.Attribute;
public class MXList
{
    private static MXList mxl = null;
    private Vector list = null;
    private static final String FACTORY_ENTRY =
        "java.naming.factory.initial";
    private static final String FACTORY_CLASS =
        "com.sun.jndi.dns.DnsContextFactory";
    private static final String PROVIDER_ENTRY =
        "java.naming.provider.url";
    private static final String MX_TYPE = "MX";
    private String dnsUrl = null;
    private String domainName = null;
    /**
     * 默认构造子是私有的，
     * 保证本类不能被外部直接实例化
     */
      private MXList() {}
    /**
```

```
    * 构造子是私有的，
    * 保证本类不能被外部直接实例化
    */
    private MXList(String dnsUrl,
            String domainName) throws Exception
    {
            this.dnsUrl = dnsUrl;
            this.domainName = domainName;

            this.list = getMXRecords(dnsUrl, domainName);
    }
    /**
    * 静态工厂方法，提供本类惟一的实例
    */
    public static synchronized MXList getInstance(
            String dnsUrl,
            String domainName) throws Exception
    {
            if (mxl == null)
            {
                    mxl = new MXList(dnsUrl, domainName);
            }
            return mxl;
    }
    /**
    * 聚集方法，提供聚集元素
    */
    public MailServer elementAt(int index)
            throws Exception
    {
            return (MailServer) list.elementAt(index);
    }
    /**
    * 聚集方法，提供聚集大小
    */
    public int size()
    {
            return list.size();
    }
    /**
    * 辅助方法，向 DNS 服务器查询 MX 记录
    */
    private Vector getMXRecords(
            String providerUrl,
            String domainName) throws Exception
    {
```

```
        // 设置环境性质
        Hashtable env = new Hashtable();

        env.put(FACTORY_ENTRY, FACTORY_CLASS);
        env.put(PROVIDER_ENTRY, providerUrl);

        // 创建环境对象
        DirContext dirContext = new InitialDirContext(env);

        Vector records = new Vector(10);

        // 读取环境对象的属性
        Attributes attrs = dirContext.getAttributes(
            domainName,
            new String[] {MX_TYPE});

        for(NamingEnumeration ne = attrs.getAll();
            ne != null && ne.hasMoreElements();)
        {
            Attribute attr = (Attribute)ne.next();
            NamingEnumeration e = attr.getAll();
            while(e.hasMoreElements())
            {
                String element = e.nextElement().toString();
                StringTokenizer tokenizer =
                    new StringTokenizer(element, " ");

                MailServer mailServer = new MailServer();

                String token1 = tokenizer.nextToken();
                String token2 = tokenizer.nextToken();

                if(token1 != null && token2 != null)
                {
                    mailServer.setPriority(
                        Integer.valueOf(token1).intValue());
                    mailServer.setServer(token2);
                    records.addElement(mailServer);
                }
            }
        }
        System.out.println("List created.");
        return records;
    }
}
```

在上面 getMXRecords()方法的源代码中使用了 Tokenizer，将 MX 记录按照空格将优先权和服务器的名字分开。

下面是一个示意性的客户端的源代码。读者可以看到，这个客户端调用单例类 MXList 的静态工厂方法，将网络的 DNS 服务器名“dns01390.ny.jeffcorp.com”和所在的网域名“jeffcorp.com”以参量形式传入，便可以得到一个 MXList 类的实例。然后，客户端就可以使用这个实例的聚集方法一个个地读出所有的 SMTP 服务器信息。

代码清单 9：客户类的源代码

```
package com.javapatterns.singleton.mxrecord;
public class Client
{
    private static MXList mxl;
    public static void main(String[] args)
        throws Exception
    {
        mxl = MXList.getInstance(
            "dns://dns01390.ny.jeffcorp.com",
            "jeffcorp.com");
        for(int i=0; i < mxl.size(); i++)
        {
            System.out.println(
                (1 + i)
                + ") priority = "
                + ((MailServer)
                mxl.elementAt(i)).getPriority()
                + ", Name = "
                + ((MailServer)
                mxl.elementAt(i)).getServer()
                );
        }
    }
}
```

由于这仅是一个示意性的客户端，因此，仅仅将查询的结果打印出来而已。在实际的应用中，读者可以将结果传给发送电子邮件的对象，从而控制向特定的 SMTP 服务器发送邮件的目的。

运行结果

程序在运行时会打印出一列 SMTP 服务器的名字以及它们的优先权。下面就是系统在本书作者的网络环境下运行的结果。

代码清单 10：运行结果

```
1) priority = 10, Name = bird2-a.jeffcorp.com
2) priority = 10, Name = bird2-b.jeffcorp.com
```

3) priority = 10, Name = horse1-a.jeffcorp.com
4) priority = 10, Name = horse1-b.jeffcorp.com
5) priority = 10, Name = horse2-a.jeffcorp.com
6) priority = 10, Name = horse2-b.jeffcorp.com
7) priority = 10, Name = bird1-a.jeffcorp.com
8) priority = 10, Name = bird1-b.jeffcorp.com

16.6　讨　　论

发送邮件时的注意事项

在使用 JavaMail 包所提供的功能发送邮件时，要循环使用所有 MXList 给出的 SMTP 服务器进行邮件发送，同时注意捕捉被抛出的 javax.mail.SendFailedException 异常。如果没有捕捉到这个异常，就表明发送工作是成功的，应当终止循环；如果捕捉到这个异常，就表明发送工作失败，应当继续循环，尝试下一个 SMTP 服务器。

与多例模式的关系

最后值得指出的是，MXList 实际上是一个将单例模式应用到一个聚集上的例子。换言之，MXList 本身是一个聚集，向外界提供聚集管理方法，如 elementAt()和 size()等，而此聚集本身是一个单例。

熟悉多例模式的读者可能会问，这是不是与多例模式有相似之处？多例模式难道不是有一个聚集吗？

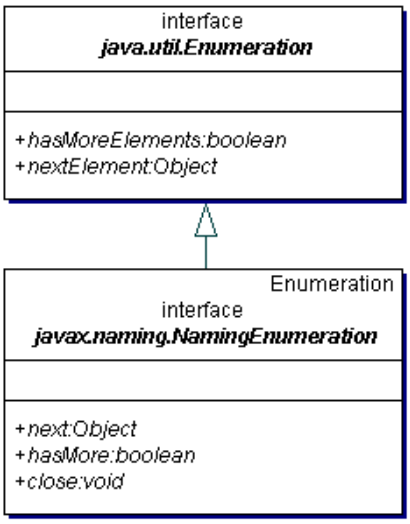
在多例模式中，聚集用来登记和管理多例类自身的实例，MXList 作为一个聚集，包含的元素是 MailServer 对象，而不是它自己。它们之间的关系就是一个多例对象与一个单例聚集对象的关系，两者虽然相像，但本质不同。

关于多例模式的详细讨论请参见本书的“专题：多例（Multiton）模式与多语言支持”一章。

迭代子模式的应用

关于迭代子模式的一般性讨论，请参见本书的“迭代子（Iterator）模式”一章。

Attributes 是一个聚集对象，因此它像所有的聚集一样，通过一个工厂方法提供一个迭代子对象。在这里工厂方法是 getAll()方法，而这个迭代子就是实现了 NamingEnumeration 接口的对象。NamingEnumeration 的类图如下所示。



从上图可以看出，NamingEnumeration 接口是 Enumeration 接口的子接口，因为 NamingEnumeration 规定出一个新的 close()方法。

由于针对命名–目录服务的信息查询会占用大量的资源，因此，在一个 Naming-Enumeration 类型的对象被创建之后，一旦不再需要，就应当调用 close()方法将资源释放掉。如果一个 NamingEnumeration 类型的迭代子对象迭代到了最后的元素之后，也就是说当 hasMore()方法返还 false 时，就不必再调用 close()方法释放资源，因为迭代子对象会自动将占用的资源释放掉。反过来，如果迭代在没有到达最后一个元素之前就中止了，那么客户端应当明显地调用 close()方法以便释放资源。

需要注意的是，在调用了 close()方法后，这个迭代子对象就不能再使用了。如果客户端再次需要 NamingEnumeration 类型的对象，就必须调用工厂方法 getAll()，以便重新产生这个对象。

问答题

1. DNS 服务器的 MX 记录并不是真正静态的，能否修改一下上面的设计，让 XMList 类能够每隔 24 小时进行一次查询，而不是仅在初始化时做一次查询？

2. 可不可以使用 JNDI 查询 javax.sql.DataSource？请给出源代码的关键部分。

3. 在上面的题目中使用了什么模式？

4. 请给出一个 EmailManager 类的源代码。这个类允许客户端传入收件人地址、发件人地址、邮件主题、邮件内容、附件等，并通过一个 SMTP 服务器发送电子邮件。

问答题答案

1. 这是可以的，只需要加入一个新的状态用于记录每一次 DNS 查询的时间即可。在用户调用 MX 记录表的时候，将实时与记录中的时间加以比较，如果时间差大于某一个数

值时，系统就重新对 DNS 查询并重建 MX 记录表，然后再回应客户端的请求。

修改后的 XMList 类的源代码如下所示。

代码清单 11：XMList 类的源代码

```
package com.javapatterns.singleton.mxrecord1;
import java.util.Enumeration;
import java.util.Hashtable;
import java.util.Vector;
import java.util.StringTokenizer;
import java.util.Calendar;
import java.util.GregorianCalendar;
import java.util.Date;
import javax.naming.NamingEnumeration;
import javax.naming.directory.DirContext;
import javax.naming.directory.InitialDirContext;
import javax.naming.directory.Attributes;
import javax.naming.directory.Attribute;
public class MXList
{
    private static MXList mxl = null;
    private Vector list = null;
    private static final String FACTORY_ENTRY =
        "java.naming.factory.initial";
    private static final String FACTORY_NAME =
        "com.sun.jndi.dns.DnsContextFactory";
    private static final String PROVIDER_ENTRY =
        "java.naming.provider.url";
    private static final String MX_TYPE = "MX";
    private String dnsUrl = null;
    private String domainName = null;
    private static Calendar updateTime ;
   /**
    * 默认构造子是私有的，
    * 保证本类不能被外部直接实例化
    */
    private MXList() {}
   /**
    * 构造子是私有的，
    * 保证本类不能被外部直接实例化
    */
    private MXList(String providerUrl,
        String domainName) throws Exception
    {
        this.dnsUrl = dnsUrl;
        this.domainName = domainName;
```

```
        this.list = getMXRecords(providerUrl,
                domainName);
    }
    /**
     * 静态工厂方法，提供本类惟一的实例
     */
    public static synchronized MXList getInstance(
        String providerUrl,
        String domainName) throws Exception
    {
        if (mxl == null)
        {
            mxl = new MXList(providerUrl, domainName);
        }
        else
        {
            // 计算现在时间与 MX 记录生成时间的间隔
            Calendar now = new GregorianCalendar();
            long TimeDifference = now.getTime().getTime()
                - updateTime.getTime().getTime() ;
            System.out.println("TimeDifference"
                + TimeDifference);

            // 如果距离 MX 记录生成的时间超过
            // 24 小时，则重新生成 MX 记录表
            if (TimeDifference    > 1000 * 60 * 60 * 24)
            {
                mxl.list = mxl.getMXRecords(
                    providerUrl, domainName);
            }
        }
        return mxl;
    }
    /**
     * 聚集方法，提供聚集元素
     */
      public MailServer elementAt(int index)
        throws Exception
    {
        return (MailServer) list.elementAt(index);
    }
    /**
     * 聚集方法，提供聚集大小
     */
    public int size()
    {
```

```
        return list.size();
    }
    /**
     * 辅助方法，向 DNS 服务器查询 MX 记录
     */
    private Vector getMXRecords(
        String dnsUrl,
        String domainName) throws Exception
    {
        // 设置环境性质
        Hashtable env = new Hashtable();

        env.put(FACTORY_ENTRY, FACTORY_NAME);
        env.put(PROVIDER_ENTRY, dnsUrl);

        // 创建环境对象
        DirContext dirContext = new InitialDirContext(env);

        Vector records = new Vector(10);

        // 取得环境对象的属性
        Attributes attrs = dirContext.getAttributes(
            domainName,
            new String[] {MX_TYPE});

        for(NamingEnumeration ae = attrs.getAll();
            ae != null && ae.hasMoreElements();)
        {
            Attribute attr = (Attribute)ae.next();
            NamingEnumeration e = attr.getAll();
            while(e.hasMoreElements())
            {
                String element = e.nextElement().toString();
                StringTokenizer tokenizer =
                    new StringTokenizer(element, " ");

                MailServer mailServer = new MailServer();

                String token1 = tokenizer.nextToken();
                String token2 = tokenizer.nextToken();

                if(token1 != null && token2 != null)
                {
                    mailServer.setPriority(
                        Integer.valueOf(token1).intValue());
                    mailServer.setServer(token2);
```

```
                        records.addElement(mailServer);
                    }
                }
            }
            // 存储实时
            updateTime = new GregorianCalendar();
            System.out.println("List created.");
            return records;
        }
    }
```

2. 这是可以的，下面的代码给出了查询 DataSource 的办法。

代码清单 12：查询 DataSource 对象的源代码

```
javax.naming.InitialContext ctx =
    new InitialContext();
Object resource = ctx.lookup("datasource");
javax.sql.DataSource hal =
    (javax.sql.DataSource) resource;
```

一旦 JNDI 返还了类型为 DataSource 的对象，就可以将这个对象当做正常的 DataSource 对象使用了。

3. 在上面的查询 DataSource 的代码中使用了工厂方法模式。lookup()方法就是工厂方法，resource 对象就是产品，而初始环境对象就是工厂角色。

相对于 java.sql.Connection 对象来说，DataSource 对象又是工厂角色，Connection 对象则是产品角色。

关于工厂方法模式，请参见本书的“工厂方法（Factory Method）模式”一章。

4. 由于篇幅限制，EmailManager 类的源代码不在这里给出。有兴趣的读者可以从本书的源代码库中看到这个类的源代码。

参考文献

[JNDI00] Sun Microsystem, Getting Started: The JNDI Overview, http://java.sun.com /products /jndi/tutorial/getStarted/overview/index.html.