第11章

代理（Proxy）模式

普通对象可以通过公共接口完成自己所需完成的工作。然而，有些对象却由于某些原因无法履行自己日常的职责。例如有的对象加载时间过长，有的对象运行在其他的计算机上，或者需要拦截发送到对象的消息等。对于这些场景，我们可以引入代理对象，通过它承担客户端需要的职责，并将相应的请求转发给底层的目标对象。

代理模式的意图是通过提供一个代理（Proxy）或者占位符来控制对该对象的访问。

经典范例：图像代理

代理对象通常拥有一个几乎和实际对象相同的接口。它常常会控制访问，并将请求合理地转发给底层的真实对象。代理模式的一个经典范例是如何避免将较大的图像加载到内存中。假设应用程序中的图像存在于页面或者容器中，并且在初始状态时并未显示。为避免在使用图像前将图像都加载进内存，就需要为这些图像创建一些代理，以便在真正需要使用图像时，才执行加载的操作。本节提供了一个图像代理的范例。但需要注意的是，使用代理模式的设计有时非常脆弱，因为它依赖于将方法调用转发给底层对象。这种转发机制可能会创建难以维护的脆弱设计。

假设 Oozinoz 公司的工程师在使用图像代理时，出于性能方面的考虑，在需要加载大图像

时，显示一个小的临时图片。工程师有一个原型版本，如图 11.1 所示。该程序的代码在 app.proxy 包的 ShowProxy 类中，支持该程序的底层代码在 com.oozinoz.imaging 包中。

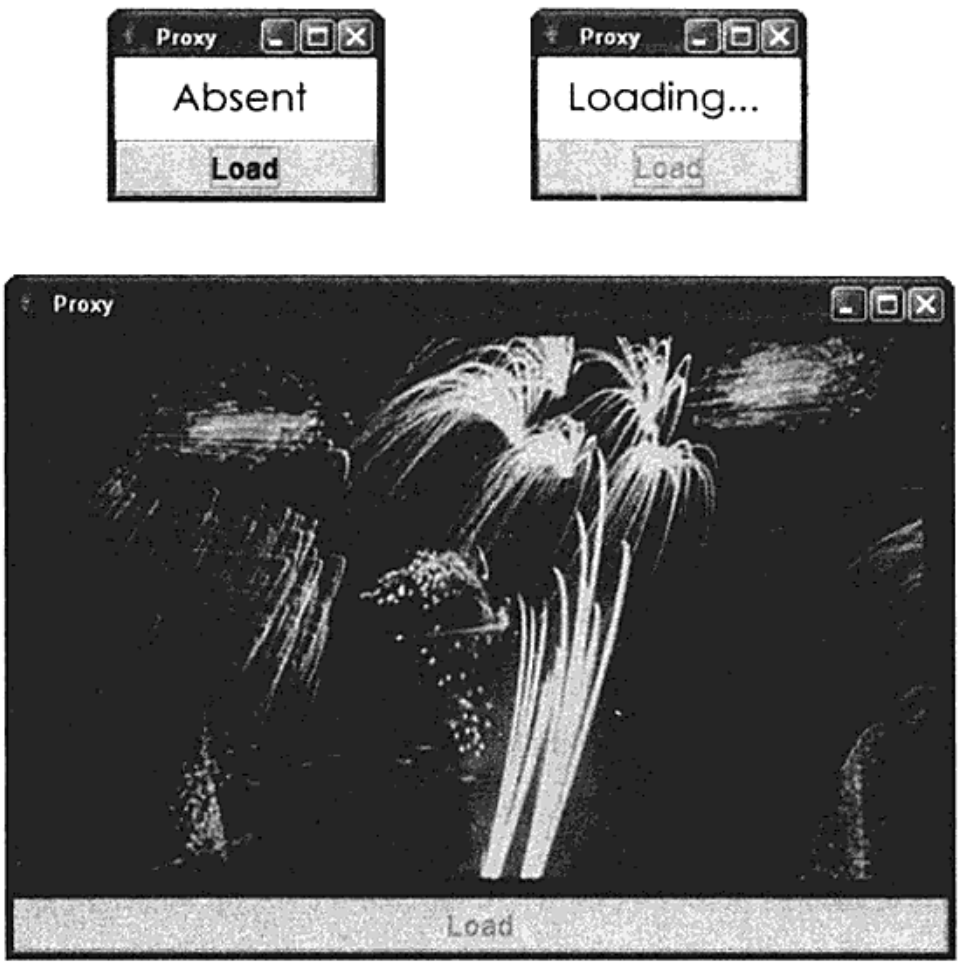

图 11.1　三张截图分别展示了应用程序中大图像在展示前、展示中，以及展示后的样子

用户界面包含三张图片：一张显示图片尚未加载，一张显示真实的图片正在加载，最后一张显示真实的图片。当应用程序启动时显示“Absent”图片，该图片需要提前用绘图工具做好。当用户单击 Load 按钮时，图片立刻就变成了预制好的“Loading...”。目标图像加载完毕后，显示该图片。

有一种简单的做法可以显示 JPEG 图片，创建一个 ImageIcon 类，将 JPEG 文件作为参数传递给该类，再用 label 来显示图像：

```
ImageIcon icon = new ImageIcon("images/fest.jpg");
JLabel label = new JLabel(icon);
```

在创建应用程序时，需要给 JLabel 传递一个代理对象，该对象将转发绘图请求给：（1）“absent”图像，（2）“loading”图像，（3）请求的图像。图 11.2 所示的顺序图大致给出了调用

的消息流。

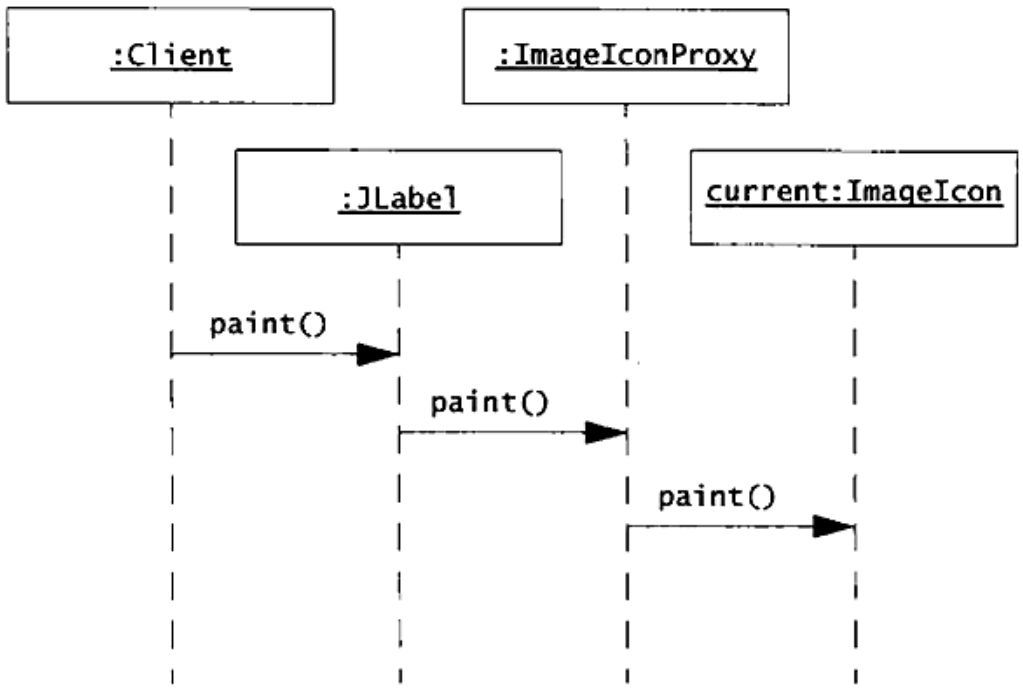

图 11.2　ImageIconProxy 对象将 paint()方法的请求转发给当前的 ImageIcon 对象

当用户单击 Load 按钮时，代码中的 ImageIconProxy 对象会将当前图片切换成 Loading...图片，此时，代理对象也会加载目标图片。当目标图片加载完毕后，ImageIconProxy 对象会将当前图片切换为目标图片。

如图 11.3 所示，可以通过创建 ImageIcon 类的子类获得一个代理类。ImageIconProxy 类定义了两个静态变量，分别包含 Absent 和 Loading...两张图片：

```
static final ImageIcon ABSENT = new ImageIcon(
    ClassLoader.getSystemResource("images/absent.jpg"));

static final ImageIcon LOADING = new ImageIcon(
    ClassLoader.getSystemResource("images/loading.jpg"));
```

ImageIconProxy 的构造函数需要图片文件的名称作为参数来加载图片。当调用 ImageIconProxy 对象的 load()方法时，会将当前图片切换到 LOADING 图片，并且新开启一个线程来加载目标图片。使用独立线程的目的是当图片加载时，不让应用程序一直等待而造成假死状态。load()方法接收一个 JFrame 对象，当目标图片加载完毕后，会回调 run()方法。以下是 ImageIconProxy.java 文件较为完整的代码：

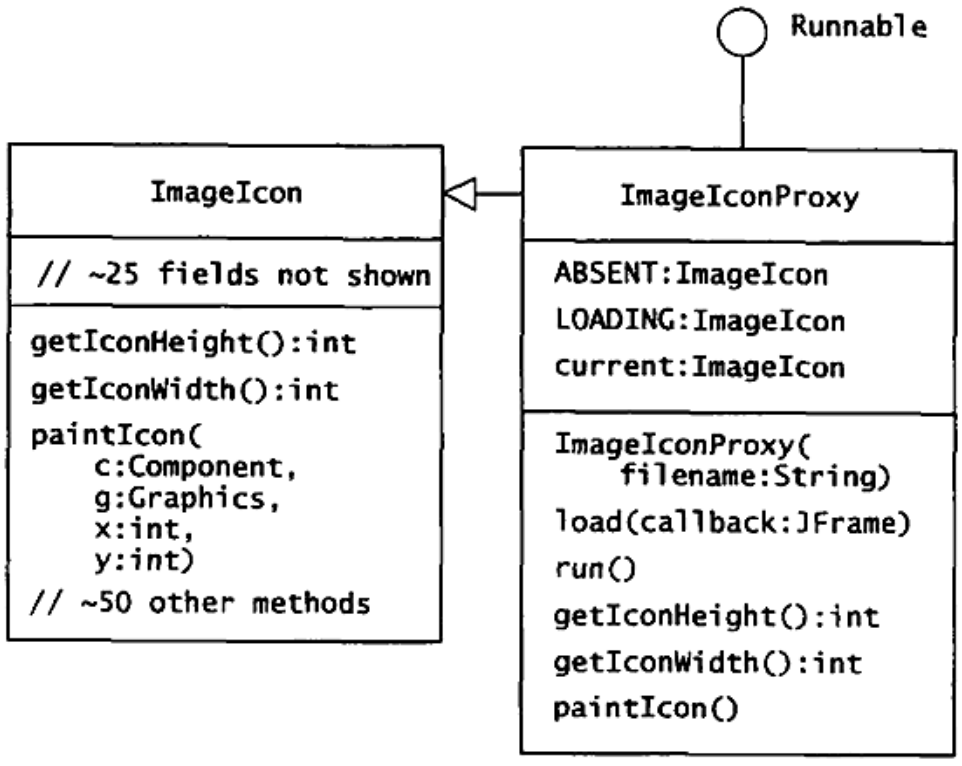

图 11.3 由于 ImageIconProxy 对象是一种 ImageIcon 对象，因此它可以替代 ImageIcon 对象

```
package com.oozinoz.imaging;
import java.awt.*;
import javax.swing.*;

public class ImageIconProxy
        extends ImageIcon implements Runnable {
    static final ImageIcon ABSENT = new ImageIcon(
        ClassLoader.getSystemResource("images/absent.jpg"));
    static final ImageIcon LOADING = new ImageIcon(
        ClassLoader.getSystemResource("images/loading.jpg"));
    ImageIcon current = ABSENT;
    protected String filename;
    protected JFrame callbackFrame;

    public ImageIconProxy(String filename) {
        super(ABSENT.getImage());
        this.filename = filename;
    }

    public void load(JFrame callbackFrame) {
        this.callbackFrame = callbackFrame;
        current = LOADING;
```

```
            callbackFrame.repaint();
            new Thread(this).start();
        }

        public void run() {
            current = new ImageIcon(
                        ClassLoader.getSystemResource(filename));
            callbackFrame.pack();
        }

        public int getIconHeight() { /* 挑战! */ }

        public int getIconWidth() { /*挑战! */ }

        public synchronized void paintIcon(
                Component c, Graphics g, int x, int y) {
            // 挑战!
        }
    }
```

挑战 11.1

ImageIconProxy 对象可以将对显示三个图片的调用转发给当前图片。请写出 ImageIconProxy 类的 getIconHeight()、getIconWidth()和 paintIcon()方法的实现代码。

答案参见第 321 页

假设你拿到了这个演示程序的代码，在创建真正的应用程序，而不仅仅是包含一个加载功能之前，需要进行设计评审，这样才能让设计中的问题显露出来。

挑战 11.2

ImageIconProxy 类并非一个设计良好的可重用组件。请指出设计中存在的两个问题。

答案参见第 322 页

当你在评审别人的设计时，既要理解别人的设计，又要持有自己的看法。开发者在运用特定设计模式时，常常会对模式的运用是否正确产生分歧。在该例子中，很明显使用了代理模式，但并不能说该模式的运用恰到好处。事实上，对于该例子有更好的设计。在使用代理模式时，使用必须得当。因为转发请求可能会造成一些问题，而这些问题在其他设计中可能没有。在阅读下一节时，你需要重新思考代理模式是否运用合理。

重新思考图片代理

对此，你可能会问设计模式是否有用。之前费尽心思地实现了一个模式，现在却需要将该模式去掉。事实上，这种现象在实际开发过程中十分常见。代码的作者在其他评审人员的帮助下，会重新思考，以改进原有设计。在实际生产过程中，设计模式可以帮助我们设计应用程序，也便于对我们的设计进行讨论。在 Oozinoz 公司的 ImageIconProxy 例子中，尽管不用模式会更简单，但模式也起到了自己的作用。

ImageIcon 类操作 Image 对象。比起将绘图请求转发给独立的 ImageIcon 对象，用 ImageIcon 对象来包装操作 Image 对象会更简单。图 11.4 展示了 com.oozinoz.imaging 包中的 LoadingImageIcon 类，该类除了构造函数外，仅有 load()和 run()两个方法。

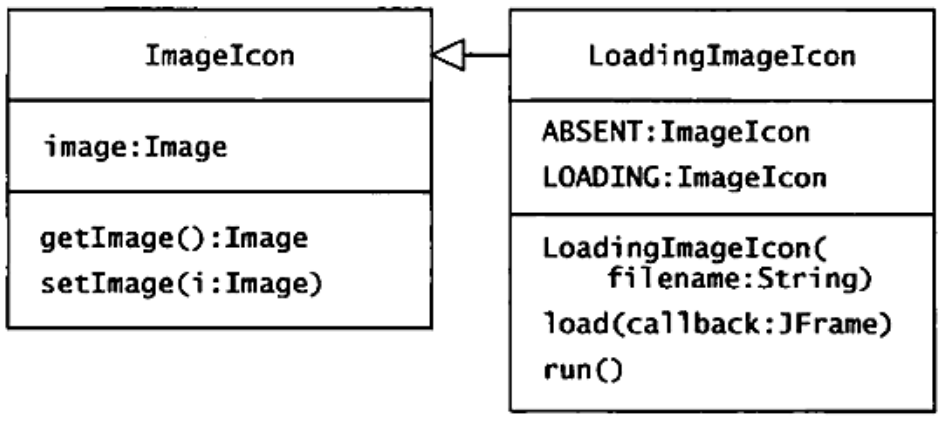

图 11.4 LoadingImageIcon 类的职责是切换 Image 对象

改进后的 load()方法依然接收一个 JFrame 对象作为参数，用于在指定图片加载后进行回调。当 load()方法执行时，会单独启动一个线程，用 LOADING 图片作为参数来调用 setImage()方法，重绘图片显示窗体。run()方法执行在一个单独的线程中，它会根据构造函数中的文件名创建一个新的 ImageIcon 对象，调用 setImage()方法将该对象传入进去。最后重新绘制该窗口。

LoadingImageIcon.java 的主要代码如下所示：

```
package com.oozinoz.imaging;
import javax.swing.ImageIcon;
import javax.swing.JFrame;
public class LoadingImageIcon
        extends ImageIcon implements Runnable {
    static final ImageIcon ABSENT = new ImageIcon(
            ClassLoader.getSystemResource("images/absent.jpg"));
    static final ImageIcon LOADING = new ImageIcon(
            ClassLoader.getSystemResource("images/loading.jpg"));
    protected String filename;
    protected JFrame callbackFrame;

    public LoadingImageIcon(String filename) {
        super(ABSENT.getImage());
        this.filename = filename;
    }

    public void load(JFrame callbackFrame) {
        // 挑战!
    }

    public void run() {
        // 挑战!
    }
}
```

挑战 11.3

将 LoadingImageIcon 类中 load()与 run()方法的代码补齐。

答案参见第 322 页

修正后的代码降低了与 ImageIcon 之间的耦合，依赖于 getImage()与 setImage()方法，而不是方法转发机制。事实上，根本不存在这种转发机制：LoadingImageIcon 类虽然结构上不是一个代理，但本质上却是。

依赖于转发的代理模式会造成维护上的负担。当底层对象改变时，Oozinoz 公司将不得不更新代理。为了避免这个负担，通常应考虑代理模式的替代方案。但是，代理模式依然可能是

正确的选择，尤其是当需要获取的消息对象在另一台机器上执行时。

远程代理

如果需要调用的对象方法运行在其他机器上，不能直接调用该方法，这就需要另辟蹊径。可以在远程机器上打开一个套接字（Socket），并设计一些协议来与远程对象之间进行消息传输。理想情况下，这种方案可以使你像在本地一样传输消息。可以调用代理对象的方法，将对真实对象的调用转发给远程机器。事实上，这种方案业已实现，例如在 CORBA（公共对象请求代理体系结构）、ASP.NET（.NET 下的动态页面）以及 Java 远程方法调用（RMI）上。

RMI 使客户类可以很容易地获取代理对象，并且转发给另一台电脑的目标对象。企业版 JavaBeans（EJB）规范是一个非常重要的行业标准。RMI 属于其中一部分，因此，很有必要了解它。无论行业标准如何改进，我们可以预见，代理模式始终会应用在分布式计算中。而 RMI 为这种模式的实践提供了一个很好的范例。

在实践 RMI 前，你需要阅读几篇相关的文章，例如，*Java™ Enterprise in a Nutshell*（由 Flanagan 等人在 2002 年编写）。下面的例子并不会介绍 RMI 的用法，而是通过 RMI 应用程序展现代理模式及其价值所在。RMI 和 EJB 带来了许多新的设计理念。当然，简单地将所有对象都变成远程对象，是不能得到一个合理的系统的。我们无法深入讨论这些内容，只能浮光掠影地看看为何 RMI 是运用代理模式的好例子。

倘若你决定开始 RMI 的实践：用 Java 代码使一个对象方法可以在另一台机器上被调用。首先需要为远程访问类创建一个接口。作为一个实验性项目，假设你创建了一个独立于 Oozinoz 公司的 Rocket 接口：

```
package com.oozinoz.remote;
import java.rmi.*;
public interface Rocket extends Remote {
    void boost(double factor) throws RemoteException;
    double getApogee() throws RemoteException;
    double getPrice() throws RemoteException;
}
```

Rocket 接口继承自 Remote，该接口中的所有方法都声明会抛出 RemoteException 异常。这种做法的原因超出了本书的讨论范围，但是任何一本关于 RMI 的书都会涉及这一点。作为一台服务器，你的远程接口应该继承自 UnicastRemoteObject 类，如图 11.5 所示。

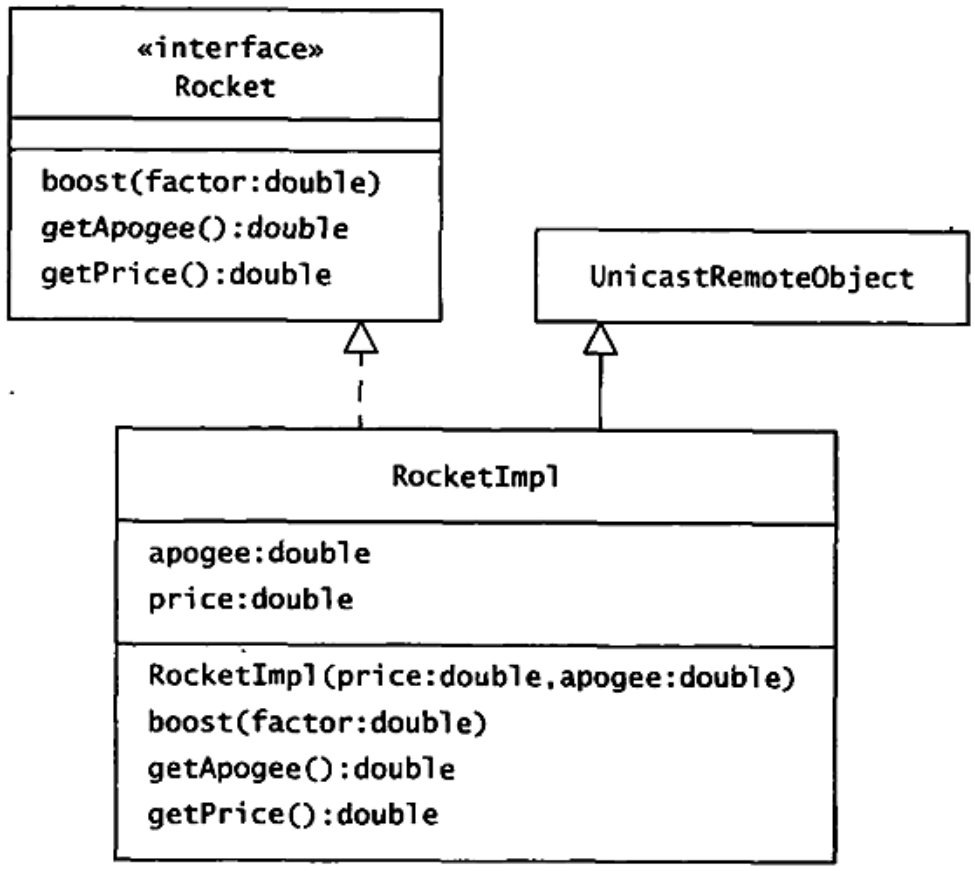

图 11.5　要使用 RMI，首先需要定义一个在计算机间传递消息的接口，然后创建 UnicastRemoteObject 类的子类来实现它

我们希望 RocketImpl 对象运行在服务器上，客户端可以通过代理访问服务器端的 RocketImpl 对象。RocketImpl 类的代码很简单。

```
package com.oozinoz.remote;
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
public class RocketImpl
        extends UnicastRemoteObject
        implements Rocket {
    protected double price;
    protected double apogee;

    public RocketImpl(double price, double apogee)
```

```
        throws RemoteException {
        this.price = price;
        this.apogee = apogee;
    }

    public void boost(double factor) {
        apogee *= factor;
    }

    public double getApogee() {
        return apogee;
    }

    public double getPrice() {
        return price;
    }
}
```

RocketImpl 的实例可以运行在一台机器上，运行在其他机器上的 Java 程序可以访问它。因此，客户端需要 RocketImpl 对象的代理。它必须实现 Rocket 接口，并且包含远程对象通信的额外特性。RMI 的一个重要好处是能自动创建这个代理类。为了产生代理类，需要把 RocketImpl.java 文件和 Rocket.java 接口文件替换到 RMI 注册目录下：

```
c:\rmi>dir /b com\oozinoz\remote
RegisterRocket.class
RegisterRocket.java
Rocket.class
Rocket.java
RocketImpl.class
RocketImpl.java
ShowRocketClient.class
ShowRocketClient.java
```

为了创建用于简化远程通信的 RocketImpl 存根，需要运行 JDK 提供的 RMI 编译器。

```
c:\rmi>rmic com. oozinoz.remote.RocketImpl
```

注意，rmic 的执行需要一个类名作为参数，而不是文件名。JDK 的早期版本将客户端的代码和服务器端的代码存放在不同的文件中。从版本 1.2 起，RMI 编译器将客户端与服务器端的

代码都创建在一个桩文件中。rmic 命令会创建一个 RocketImpl_Stub 类：

```
c:\rmi>dir /b com\oozinoz\remote
RegisterRocket.class
RegisterRocket.java
Rocket.class
Rocket.java
RocketImpl.class
RocketImpl.java
RocketImpl_Stub.class
ShowRocketClient.class
ShowRocketClient.java
```

为了运行该对象，必须使用运行在服务器上的 RMI 注册程序注册该对象。rmiregistry 可执行程序是 JDK 的一部分。当运行该注册程序时，需要指定该注册程序侦听的端口号：

```
c:\rmi> rmiregistry 5000
```

在服务器上运行该注册程序后，可以创建并注册一个 RocketImpl 对象：

```
package com.oozinoz.remote;
import java.rmi.*;
public class RegisterRocket {
    public static void main(String[] args) {
        try {
            // 挑战!
            Naming.rebind(
                "rmi://localhost:5000/Biggie", biggie);
            System.out.println("Registered biggie");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

编译并运行该代码，程序会显示出注册的确认信息。

```
Registered biggie
```

你需要替换 RegisterRocket 类的“//挑战!”代码，该类会创建 biggie 对象，对火箭进行建模。main()方法中的其余代码注册了这个对象。关于 Naming 类机制的描述超过了本章的

讨论范围。当然，你必须拥有足够的信息来创建 biggie 对象。

挑战 11.4

用声明和实例化 biggie 对象的代码来替换“//挑战!”的内容。假定 biggie 建模火箭的价格是$29.95，最高 820 米。

答案参见第 323 页

运行 RegisterRocket 程序，可以在服务器上创建一个 RocketImpl 对象 biggie。运行在另一台机器上的客户端只要有权限访问 Rocket 接口和 RocketImpl_Stub 类，就可以访问 biggie 对象。如果只有一台机器，仍然可以测试 RMI，可以通过 localhost，而不是另一台主机访问服务器。

```
package com.oozinoz.remote;

import java.rmi.*;
public class ShowRocketClient {
    public static void main(String[] args) {
        try {
            Object obj = Naming.lookup(
                "rmi://localhost:5000/Biggie");
            Rocket biggie = (Rocket) obj;
            System.out.println(
                "Apogee is " + biggie.getApogee());
        } catch (Exception e) {
            System.out.println(
                "Exception while looking up a rocket:");
            e.printStackTrace();
        }
    }
}
```

程序运行时，会通过已经注册的“Biggie”名字来查找对象。该对象代表着 RocketImpl，而 obj 对象的 lookup()方法将返回 RocketImpl_Stub 类的实例。RocketImpl_Stub 类实现了 Rocket 接口，因此可以将 obj 对象转换为 Rocket 接口的实例。RocketImpl_Stub 类继承自 RemoteStub 类，并且可以让该对象与服务器进行通信。

运行 ShowRocketClient 程序时，它将打印出“Biggie”火箭的高度。

```
Apogee is 820.0
```

通过代理，对 getApogee()的调用会转发给运行在服务器上的 Rocket 接口的实现。

挑战 11.5

图 11.6 展示了 getApogee()调用被转发的过程。最右边的对象用加粗标识，表明它和 ShowRocketClient 程序不在一起运行。请填上图中缺失的类名。

答案参见第 323 页

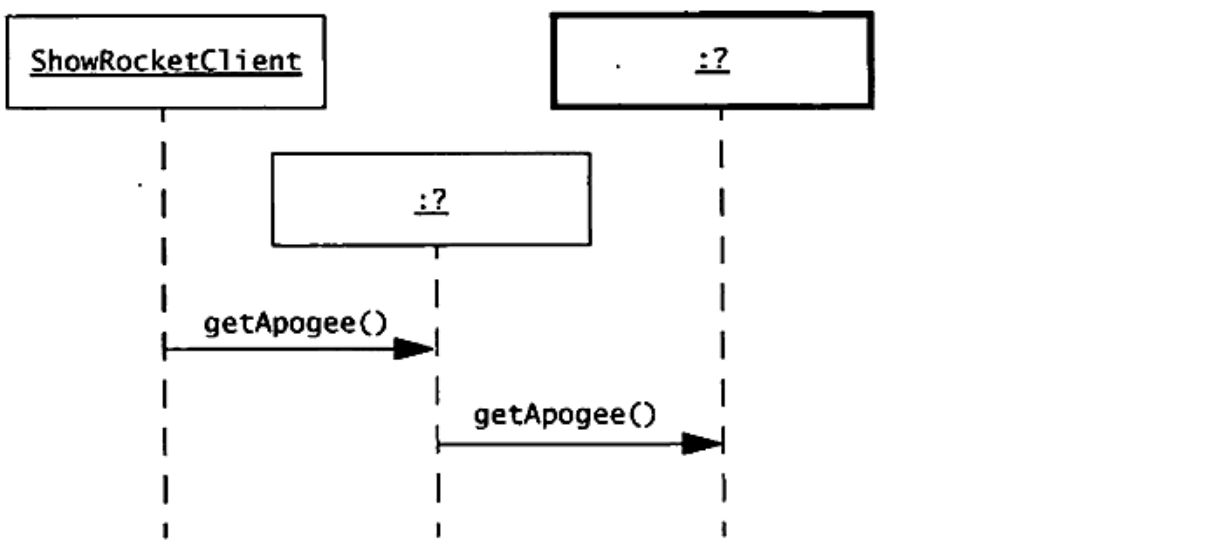

图 11.6　当你完成这幅图后，该图会展示基于 RMI 分布应用的消息流

使用 RMI 的好处在于，让客户端像与本地对象通信那样与远程对象通信。你为该对象定义了一个接口，并且让客户端和服务器端都共用这个接口。RMI 提供了客户端与服务器端的通信机制，并将它们相互隔离。两端相互协作，提供了进程间的无缝通信。

动态代理

Oozinoz 公司的工程师们偶尔会遇到性能问题，但他们却并不希望就此而对代码进行大的改动。

Java 的动态代理特性可以轻易实现这一点。它允许我们使用代理对象来包装其他对象。你可以使用代理对象来拦截对被包装对象的请求，然后用代理再转发给这些被包装对象。你还可以在拦截调用的前后增加自己的代码。对动态代理施加限制可以防止包装任意对象。在正常条件下，动态代理使你能完整地控制被包装对象的操作。

动态代理依赖于对象类所实现的接口。接口中定义什么调用，代理就能拦截什么调用。如果你想拦截一个接口实现类的方法，需要使用动态代理去包装这个类的实例。

为创建动态代理，必须列出所要拦截的接口列表。幸运的是，可以通过询问所要包装的对象，获得这一列表，代码如下：

```
Class[] classes = obj.getClass().getInterfaces();
```

这行代码可以获得所要拦截的方法实现的所有接口。为了建立动态代理，还需要考虑两个因素：类加载器与当代理拦截调用时需要执行的行为类。对于接口列表，可以使用需要包装的相关对象来获取一个合适的类加载器：

```
ClassLoader loader = obj.getClass().getClassLoader();
```

最后一个需要的元素是代理对象自身。这个对象的类型必须实现 java.lang.reflect 包中的 InvocationHandler 接口。该接口声明了如下操作：

```
public Object invoke(Object proxy, Method m, Object[] args)
    throws Throwable;
```

在对动态代理进行包装时，对包装对象的调用会转发给你所提供类的 invoke()方法。invoke()方法会继续将方法调用转发给被包装对象。可以通过如下代码转发调用：

```
result = m.invoke(obj, args);
```

这行代码通过反射将目标调用转发给被包装对象。动态代理的美妙之处在于可以在转发调用之前或之后执行任何行为。

假设某个方法需要执行较长的时间，你可能希望记录一个报警日志，就可以用如下代码创建一个 ImpatientProxy 类：

```
package app.proxy.dynamic;

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

```
public class ImpatientProxy implements InvocationHandler {
    private Object obj;

    private ImpatientProxy(Object obj) {
        this.obj = obj;
    }

    public Object invoke(
            Object proxy, Method m, Object[] args)
            throws Throwable {
        Object result;
        long t1 = System.currentTimeMillis();
        result = m.invoke(obj, args);
        long t2 = System.currentTimeMillis();
        if (t2 - t1 > 10) {
            System.out.println(
                "> It takes " + (t2 - t1)
                + " millis to invoke " + m.getName()
                + "() with");
            for (int i = 0; i < args.length; i++)
                System.out.println(
                    "> arg[" + i + "]: " + args[i]);
        }
        return result;
    }
}
```

它实现了 invoke()方法，便于检查方法调用所需要的完整时间。如果执行时间过长，ImpatientProxy 类会打印出警告信息。

若要使用 ImpatientProxy 对象，就要用到 java.lang.reflect 包中的 Proxy 类。Proxy 类需要一组接口和一个类加载器，以及 ImpatientProxy 类的实例。为了简化动态代理的创建，可以为 ImpatientProxy 类增加如下方法：

```
public static Object newInstance(Object obj) {
    ClassLoader loader = obj.getClass().getClassLoader();
    Class[] classes = obj.getClass().getInterfaces();
    return Proxy.newProxyInstance(
        loader, classes, new ImpatientProxy(obj));
}
```

这个静态方法为我们创建了一个动态代理。为了包装这个对象，newInstance()方法会获取对象的接口列表以及类加载器。该方法实例化了 ImpatientProxy 类，传递要包装的对象。所有这些准备的对象都会传递给 Proxy 类的 newProxyInstance()方法。

返回的对象会实现被包装对象的所有接口。我们可以将其转换成它所实现的任意一种类型。

假设你正在维护一组对象，其中一些对象的某些方法运行得比较慢。为了找到这些运行缓慢的方法，可以包装一个 ImpatientProxy 对象。下面的代码展示了这个例子：

```
package app.proxy.dynamic;

import java.util.HashSet;
import java.util.Set;
import com.oozinoz.firework.Firecracker;
import com.oozinoz.firework.Sparkler;
import com.oozinoz.utility.Dollars;

public class ShowDynamicProxy {
    public static void main(String[] args) {
        Set s = new HashSet();
        s = (Set)ImpatientProxy.newInstance(s);
        s.add(new Sparkler(
            "Mr. Twinkle", new Dollars(0.05)));
        s.add(new BadApple("Lemon"));
        s.add(new Firecracker(
            "Mr. Boomy", new Dollars(0.25)));
        System.out.println(
            "The set contains " + s.size() + " things.");
    }
}
```

这段代码创建了一个 Set 对象来维护一些项目，然后代码调用了 ImpatientProxy 对象的 newInstance()方法，通过强制转换的方式对集合进行包装。这样做的结果是使 s 对象的行为看起来像一个集合，唯一的例外是当 ImpatientProxy 对象的方法执行时间过长时，导致结果不能正常返回。例如，当程序调用集合的 add()方法时，ImpatientProxy 对象会拦截该调用，并将该调用转发给实际集合对象，然后记录每个调用的耗时。

运行 ShowDynamicProxy 程序会产生如下的输出：

```
> It takes 1204 millis to invoke add() with
>     arg[0]: Lemon
The set contains 3 things.
```

ImpatientProxy 能够帮助我们找出集合中有哪些对象执行时间过长。目前是 BadApple 类的“Lemon”实例。BadApple 类的代码如下：

```
package app.proxy.dynamic;

public class BadApple {
    public String name;

    public BadApple(String name) {
        this.name = name;
    }

    public boolean equals(Object o) {
        if (!(o instanceof BadApple))
            return false;
        BadApple f = (BadApple) o;
        return name.equals(f.name);
    }

    public int hashCode() {
        try {
            Thread.sleep(1200);
        } catch (InterruptedException ignored) {
        }
        return name.hashCode();
    }
    public String toString() {
        return name;
    }
}
```

ShowDynamicProxy 代码使用 ImpatientProxy 对象去监视集合的调用，而集合和 ImpatientProxy 对象间没有任何连接。一旦你写了一个动态代理类，只要对象实现了你想拦截的方法所属的接口，就可以使用该代理类去包装该对象。

在方法执行前后对调用进行拦截，并创建自己行为的思想属于面向切面编程（Aspect

Oriented Programming，AOP）。在 AOP 中，切面就是所谓的“建议”（advice，你要插入的代码）和“切入点”（point cut，对插入代码执行点的定义）的组合。AOP 的知识足够写一本书了，但你可以使用动态代理来尝试在各种对象间复用行为。

Java 中的动态代理可以让你使用代理来包装对象，还可以拦截对象方法的调用，以便在调用方法前后增加自己的行为。这种给任意对象增加可复用行为的方式，与面向切面编程十分相似。

小结

代理模式的实现为对象建立了一个占位符，用来管理对目标对象的访问。代理对象可以隔离目标对象的状态迁移，例如图片加载的时间变化。然而，运用代理模式会使得代理对象与被代理对象造成紧耦合。Java 中的动态代理提供了一种增加可复用功能的机制。倘若某个对象实现了你想要拦截的接口方法，就可以使用动态代理去包装这个对象，增加自己的逻辑或者替换被包装对象的代码。