

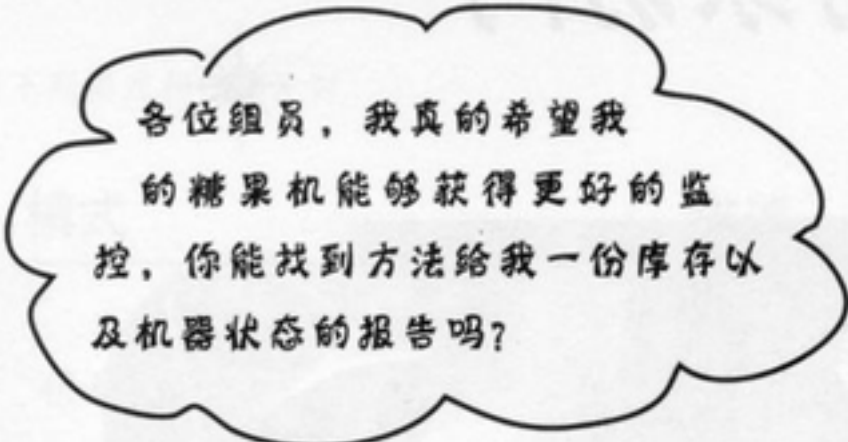
代理模式

控制对象访问

朱霆

2017.5.21

问题



各位组员，我真的希望我的糖果机能够获得更好的监控，你能找到方法给我一份库存以及机器状态的报告吗？

糖果公司CEO：希望对糖果机有更好的监控，能够获得库存以及机器状态的报告

已有取得糖果数量的getCount()方法和取得糖果机状态的getState()方法

看起来很简单，只需要调用这两个方法生成报告，交给CEO就可以了

```
public class GumballMonitor {
    GumballMachine machine;

    public GumballMonitor(GumballMachine machine) {
        this.machine = machine;
    }

    public void report() {
        System.out.println("Gumball Machine: " + machine.getLocation());
        System.out.println("Current inventory: " + machine.getCount() + " gumballs");
        System.out.println("Current state: " + machine.getState());
    }
}
```

The monitor takes the machine in its constructor and assigns it to the machine instance variable.

Our report method just prints a report with location, inventory and the machine's state.

糖果监视器：

构造器需要传入一个糖果机

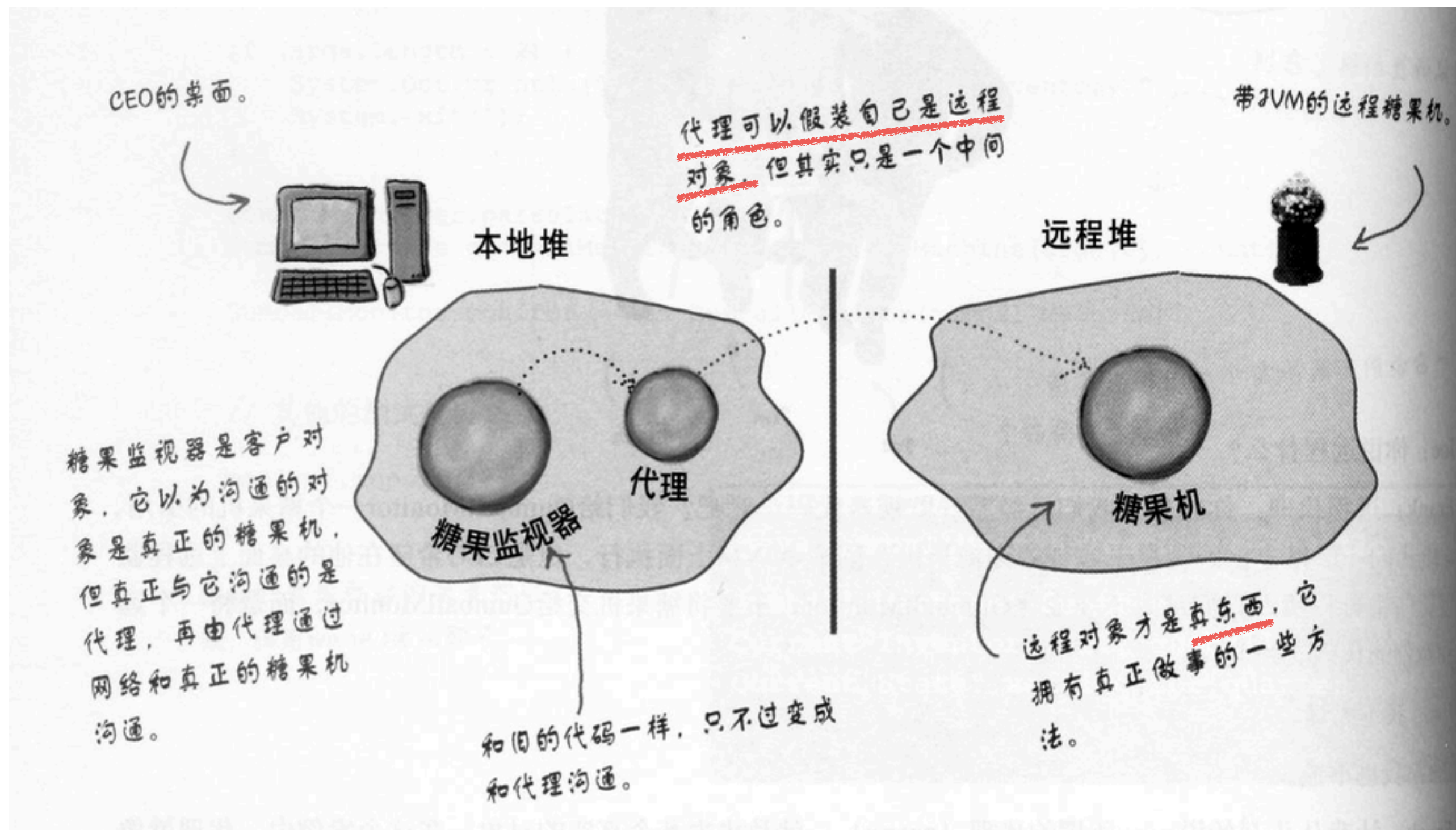
report方法负责将位置、库存等信息打印出来

```
File Edit Window Help FlyingFish
%java GumballMachineTestDrive Seattle 112

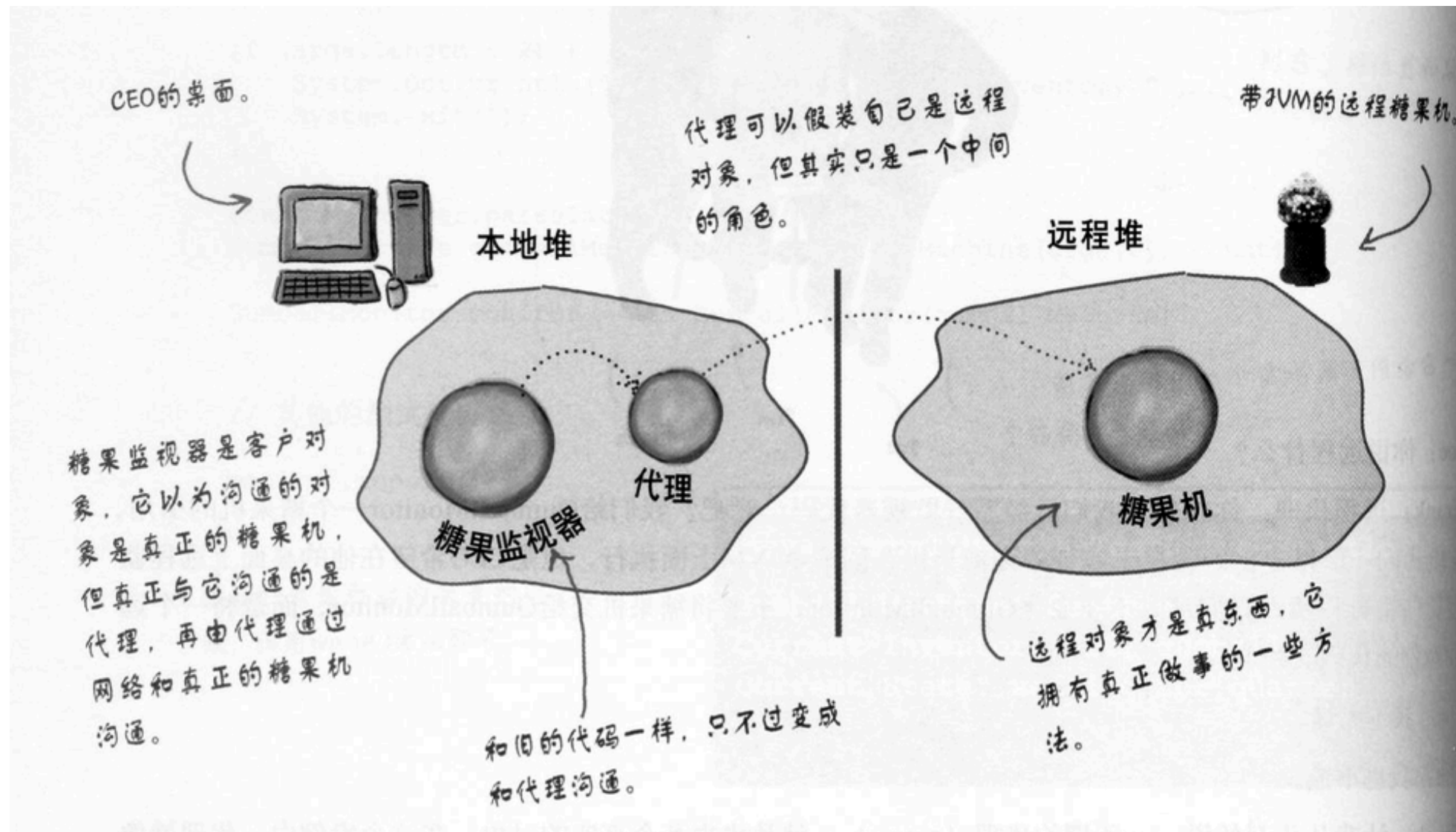
Gumball Machine: Seattle
Current Inventory: 112 gumballs
Current State: waiting for quarter
```

监视器的输出看起来虽然很不错，但可能是我之前说的不够清楚，我需要的是在远程监控糖果机！事实上，我们已经把网
络准备好了。拜托，你们这些人不是号称Internet一代吗？

需要远程代理
Remote Proxy



远程代理，好比远程对象的本地代表



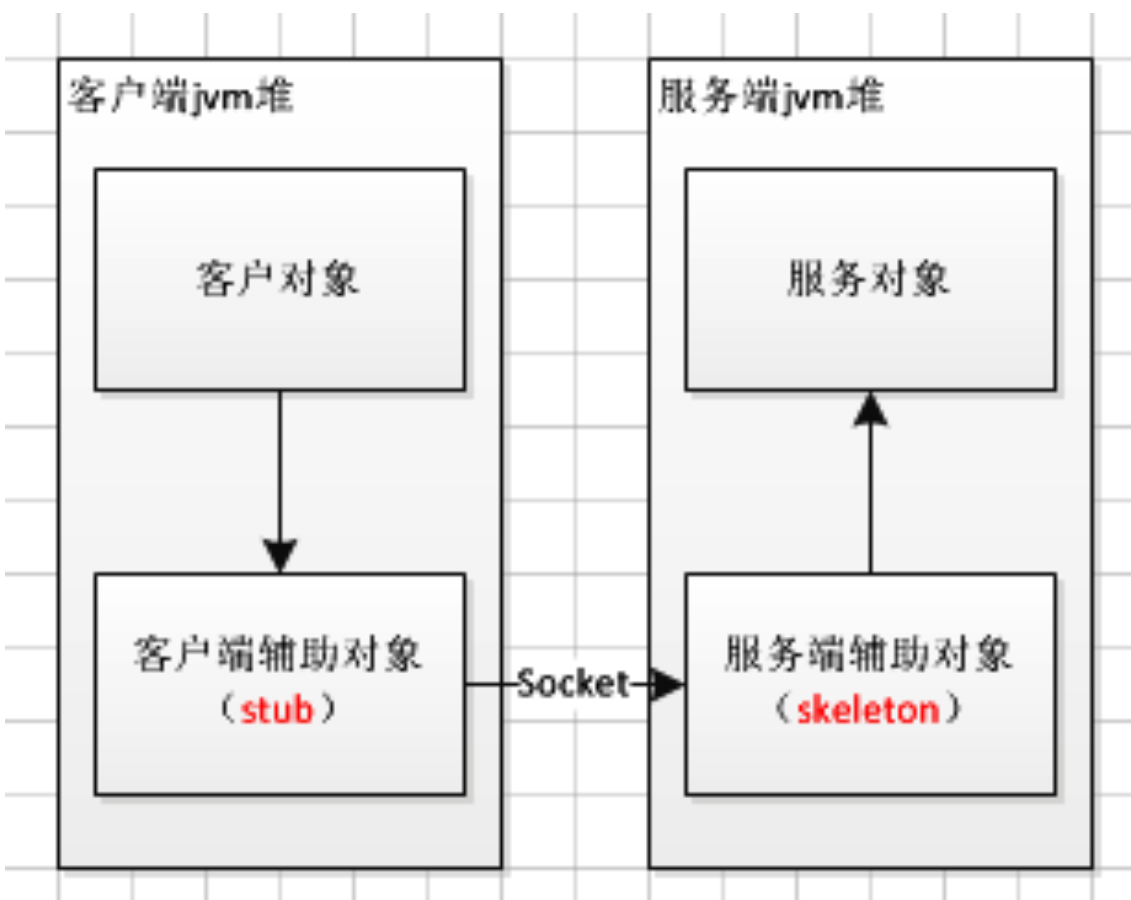
代理就像是糖果机对象一样，但其实幕后它利用网络和一个远程的**真正**的糖果机沟通

不需要修改代码，只需要将GumballMachine**代理**版本的**引用**交给监视器即可

监视器就像在做远程方法调用，但其实只是调用**本地堆**中的代理对象上的方法，再由代理处理所有网络通信的底层细节

Java RMI (Remote Method Invocation)

RMI: 能够让在某个Java虚拟机上的对象像调用本地对象一样调用**另一个**Java虚拟机中的对象上的方法



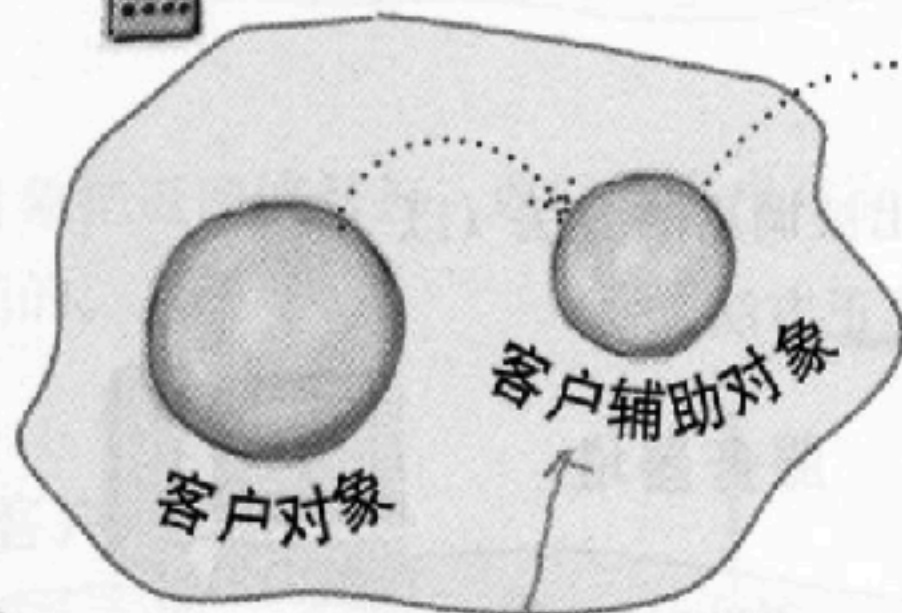
1. 客户对象调用客户端辅助对象上的方法
2. 客户端辅助对象**打包**调用信息（变量，方法名），通过网络发送给服务端辅助对象
3. 服务端辅助对象将客户端辅助对象发送来的信息**解包**，找出真正被调用的方法以及该方法所在对象
4. 调用**真正**服务对象上的**真正**方法，并将结果返回给服务端辅助对象
5. 服务端辅助对象将结果**打包**，发送给客户端辅助对象
6. 客户端辅助对象将返回值**解包**，返回给客户对象
7. 客户对象获得返回值

看起来应该很熟悉……

客户辅助对象假装自己就是服务，但其实它只是“真东西”的一个代理。



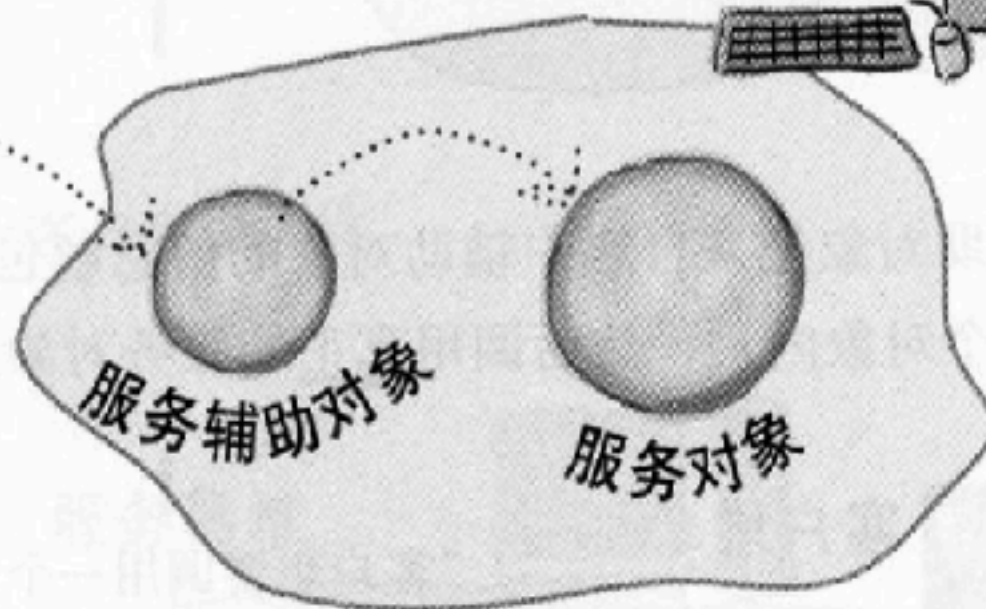
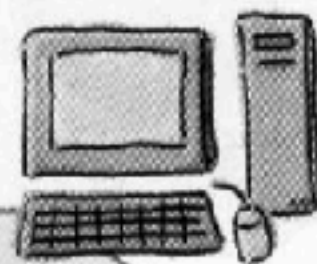
客户堆



象以为在和
服务沟通，以
辅助对象就
真正做事的

这是我们的
代理。

服务器堆



服务辅助对象从客户辅助
对象处取得请求，对它解
包，调用真正服务上的方
法。

服务对象是真正提供
服务的地方，它的方
法真正在做事情。

远程接口

别忘了 import java.rmi.*

```
import java.rmi.*;
```

这就是远程接口。

```
public interface GumballMachineRemote extends Remote {  
    public int getCount() throws RemoteException;  
    public String getLocation() throws RemoteException;  
    public State getState() throws RemoteException;  
}
```

↑
所有的返回类型都必须
是原语类型或可序列化类
型.....

↑
这是准备支持的方法，每个都要抛出
RemoteException。

```
public class GumballMachine
    extends UnicastRemoteObject implements GumballMachineRemote
{
    // 这里有实例变量

    public GumballMachine(String location, int numberGumballs) throws RemoteException {
        // 这里有代码
    }

    public int getCount() {
        return count;
    }

    public State getState() {
        return state;
    }

    public String getLocation() {
        return location;
    }

    // 这里有其他的方法
}
```

↓

……构造器需要抛出
RemoteException, 因为超
类是这么做的。

不要怀疑, 这里完全不
需要改!

需要实现**远程端口**, 确定它可以当成服务使用

```
import java.rmi.*;

public class GumballMonitor {
    GumballMachineRemote machine;

    public GumballMonitor(GumballMachineRemote machine) {
        this.machine = machine;
    }

    public void report() {
        try {
            System.out.println("Gumball Machine: " + machine.getLocation());
            System.out.println("Current inventory: " + machine.getCount() + " gumballs");
            System.out.println("Current state: " + machine.getState());
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }
}
```

我们需要import RMI的包，因为下面将用到RemoteException类……

现在我们准备依赖此远程接口，而不是具体的GumballMachine类。

当我们试图调用那些最终要通过网络发生的方法时，我们需要捕获所有可能发生的远程异常。

Monitor现在依赖**远程接口**，而不是具体的GumballMachine类

Monitor认为这个远程接口就是**真正**的糖果机


```
import java.rmi.*;

public class GumballMonitorTestDrive {

    public static void main(String[] args) {
        String[] location = {"rmi://santafe.mightygumball.com/gumballmachine",
                             "rmi://boulder.mightygumball.com/gumballmachine",
                             "rmi://seattle.mightygumball.com/gumballmachine"};

        GumballMonitor[] monitor = new GumballMonitor[location.length];

        for (int i=0; i < location.length; i++) {
            try {
                GumballMachineRemote machine =
                    (GumballMachineRemote) Naming.lookup(location[i]);
                monitor[i] = new GumballMonitor(machine);
                System.out.println(monitor[i]);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }

        for(int i=0; i < monitor.length; i++) {
            monitor[i].report();
        }
    }
}
```

被监视的位置有这些。

我们创建一个数组，数组内的元素是每台机器的位置。

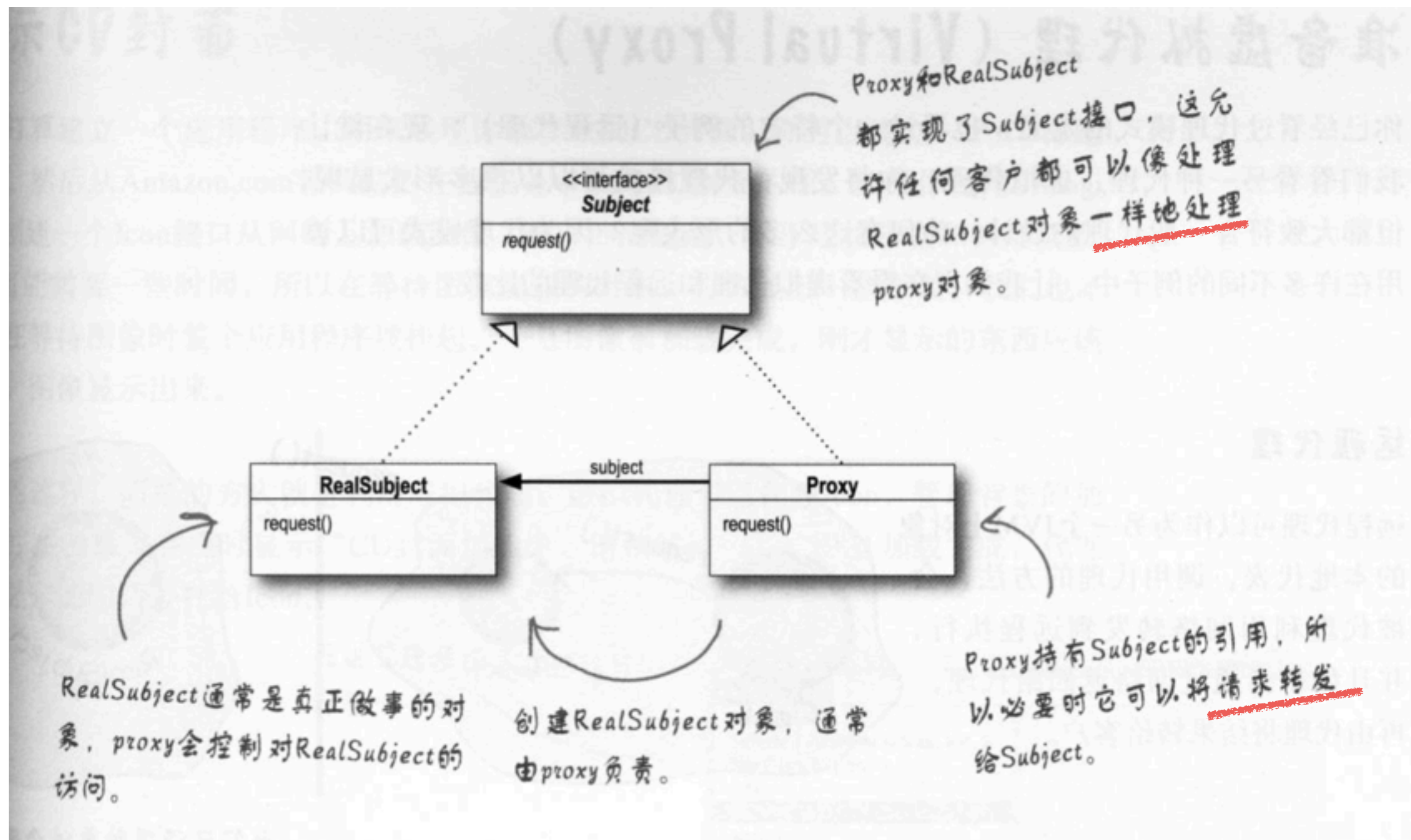
我们也创建监视器的数组。

现在，需要为每个远程机器创建一个代理。

实际传给Monitor的只是代理

代理模式定义

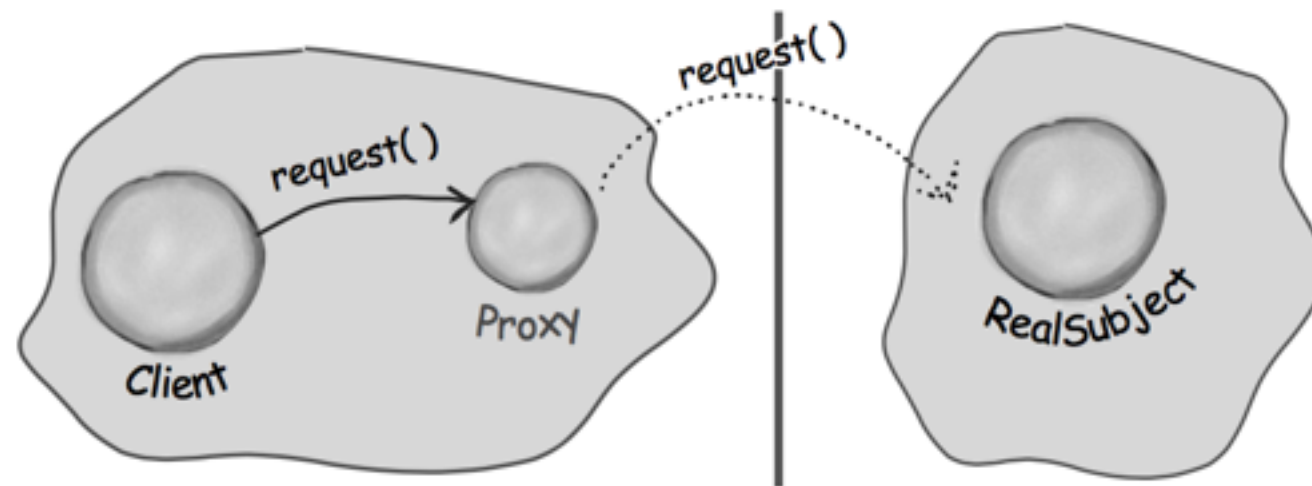
- 代理模式为另一个对象提供一个**替身或占位符**以控制对这个对象的**访问**
- 使用代理模式创建**代表(representative)**对象，让代表对象控制某对象的访问
- 被代理的对象可以是**远程**的对象、创建**开销**大的对象或需要**安全**控制的对象



- Proxy和RealSubject都实现Subject接口
- Proxy持有RealSubject的引用

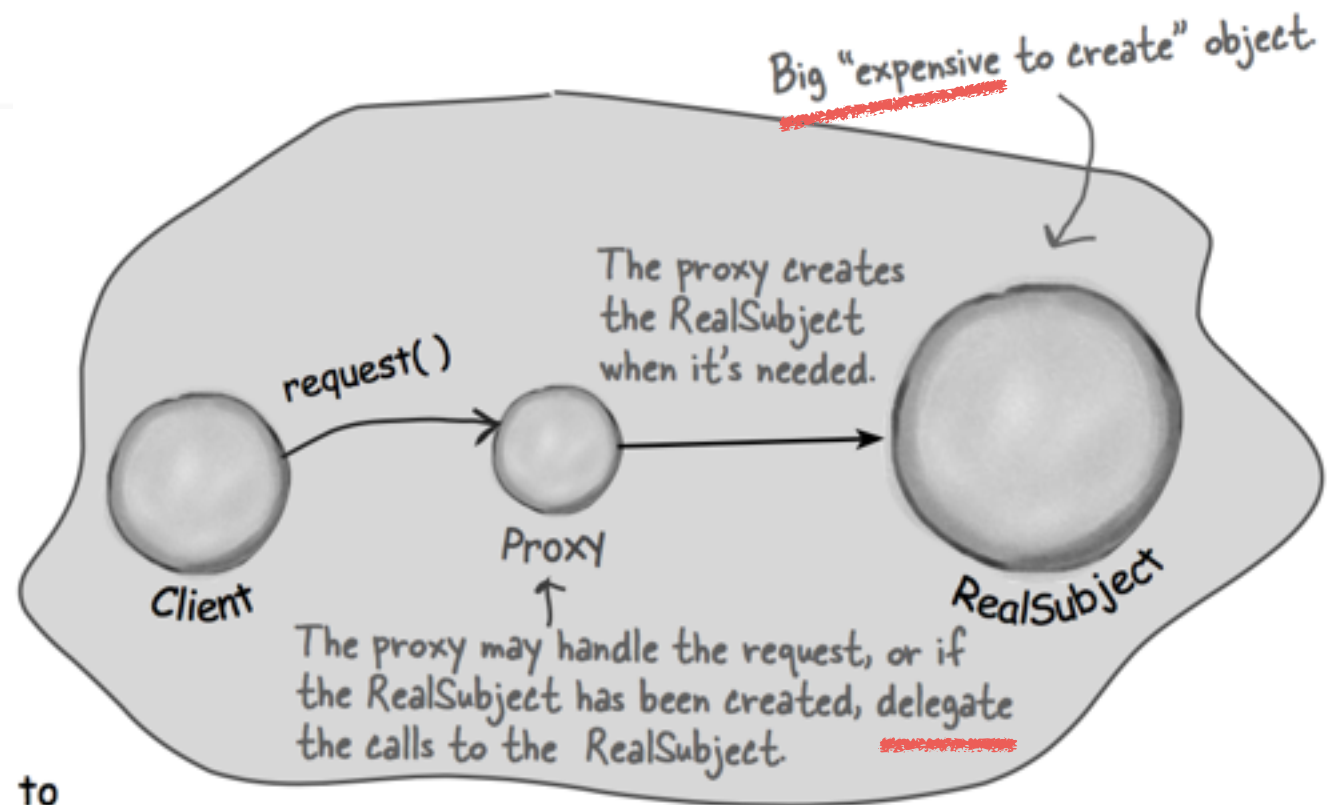
虚拟代理(virtual proxy)

远程代理:



虚拟代理:

当对象尚未创建时，
虚拟代理扮演对象的
替身。对象创建后，
代理将请求**直接委托**
给对象



Choose the album cover of your liking here.



While the CD cover is loading, the proxy displays a message.

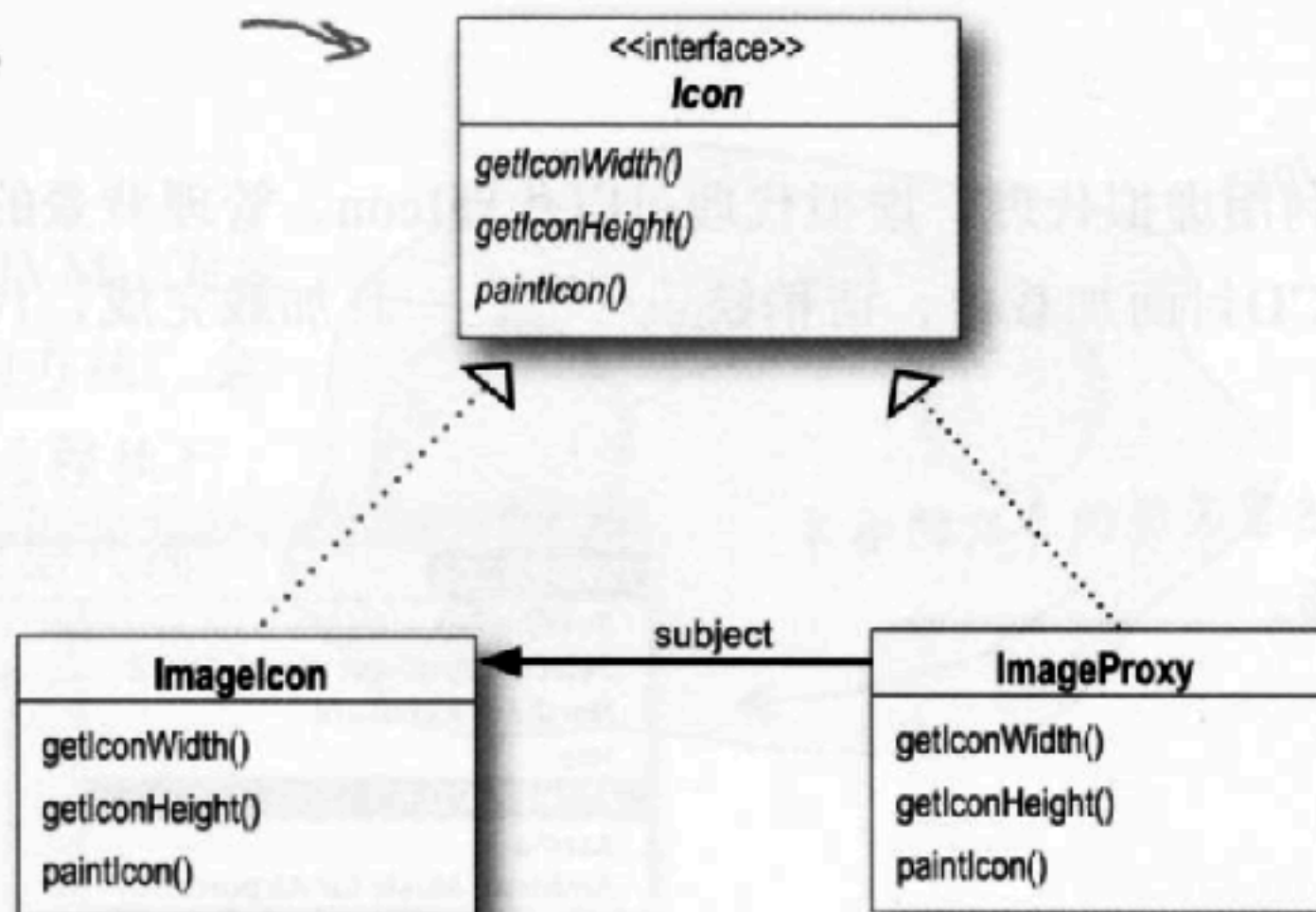


When the CD cover is fully loaded, the proxy displays the image.



虚拟代理可以代理Icon，管理背景的加载在加载未完成时显示waiting，一旦加载完成，代理就把显示的职责委托给Icon

这是Swing的Icon接口，在用
户界面上显示图像。



这是javax.swing.
ImageIcon，一个显
示图像的类

这是我们的代理，首先显示消
息，当图像加载完成后，委托
ImageIcon显示图像。

问答(1/3)

- **问：**代理与装饰者与很像。我们基本上都是用用一个对象把另一个包起来，然后把调用委托给真实对象。这样说有什么问题？
- **答：**它们的目的是不一样的。装饰者为对象**增加行为**，而代理是**控制对象的访问**
ImageProxy是控制ImageIcon的访问。代理将客户从ImageIcon解耦了，如果它们之间没有解耦，客户就必须等到每幅图像都被取回，才能绘制在界面上

问答(2/3)

- **问：** 如何让客户使用代理，而不是真正的对象？
- **答：** 一个常用的技巧是提供一个**工厂**，实例化并返回主题
因为这是在工厂方法内发生的，我们可以用代理包装主题再返回，而客户不知道也不在乎他使用的是代理还是真东西

问答(3/3)

- **问：**代理和**适配器**之间是什么关系？
- **答：**代理和适配器都是挡在其他对象前面，并负责将请求转发给它们。
适配器会改变对象适配的**接口**，而代理则实现相同的接口

其他代理

- **保护代理：** 根据客户的角色，决定是否允许客户访问特定的方法
- **防火墙代理：** 控制网络资源的访问，保护主题免于坏客户的侵害
- **智能引用代理：** 当主题被引用时，进行额外的动作，例如计算一个对象被引用的次数

其他代理

- **缓存代理：**为开销大的运算结果提供暂时存储；它也允许多个客户共享结果，以减少计算或网络延迟
- **同步代理：**在多线程的情况下，为主题提供安全访问
- **复杂隐藏代理：**用来隐藏一个类的复杂集合的复杂度，并进行访问控制
- **写入时复制代理：**用来控制对象的复制，方法是延迟对象的复制，直到客户真的需要为止

Thank You!