第10章

# 调停者（Mediator）模式

面向对象开发要求尽可能恰当地分配职责，要求对象能够独立地完成自己的任务。例如观察者模式，就是通过最小化对象与对象之间的职责交互，从而支持职责的合理分配。单例模式是将职责集中在某个对象中以便其他对象的访问与重用。与单例模式相似，调停者模式也是集中职责，但它是针对一组特殊的对象，而不是系统中全部的对象。

当对象间的交互趋向复杂，而每个对象都需要知道其他对象的情况时，提供一个集中的控制权是很有用的。当相关对象的交互逻辑独立于对象的其他行为时，职责的集中同样有用。

**调停者模式的意图是定义一个对象，封装一组对象的交互，从而降低对象间的耦合度，避免了对象间的显式引用，并且可以独立地改变对象的行为。**

## 经典范例：GUI 调停者（Mediator）

在开发 GUI 应用时，可能经常遇到调停者模式。应用程序常常会变得越来越庞大，类聚集了太多的代码，这些代码完全可以重构到多个类中。如第 4 章外观模式中的 `ShowFlight` 类，最初扮演了 3 个角色，在重构之前，它是一个显示面板，一个完整的 GUI 应用，一个轨迹计算器。重构完成后，飞行轨迹面板的应用变得很简单，仅仅包含少量代码行。然而，对于大型应用，即使这些代码仅仅包含创建部件与安排部件的交互逻辑，在经过重构后，仍然很复杂。思

考图 10.1 所示的应用。

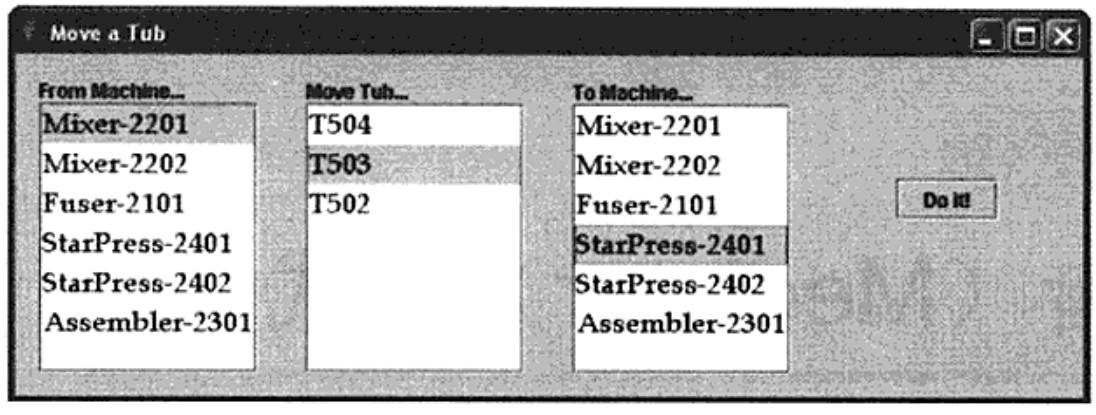


**图 10.1　该应用程序可以使用户手工更新药品位置**

Oozinoz 公司使用橡胶箱存放化学药品。机器会读取箱子上的条形码，以便记录箱子在工厂中的位置。有时，人工干预是必要的，特别是由人而非机器人去移动箱子时。图 10.1 展示了一个全新的、开发了部分功能的应用程序，可以使用户指定箱子在机器上的位置。

在包 `app.mediator.moveATub` 的 `MoveATub` 应用程序中，用户在左侧箱子列表菜单中选择其中一台机器，继而选择箱子，再选中目标机器，最后单击 Do it!按钮，系统就会更新箱子的位置。图 10.2 展示了程序中的部分类。

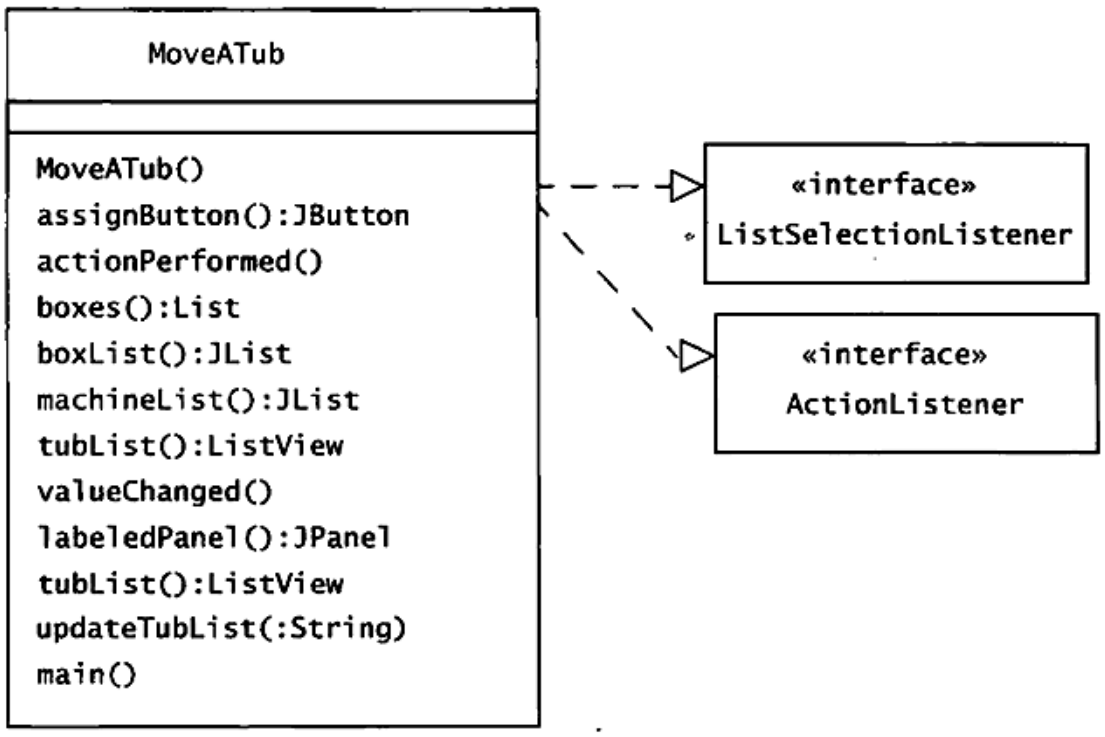


**图 10.2　MoveATub 类混合了创建组件、事件处理以及模拟数据库的方法**

该程序的开发人员最初使用向导进行开发，并且已经进行了重构。MoveATub 类的一半方法都包含了拥有 GUI 组件的变量，这些变量是可以延迟初始化的。assignButton()方法就是一个典型的例子：

```
private JButton assignButton() {
    if (assignButton == null) {
        assignButton = new JButton("Do it!");
        assignButton.setEnabled(false);
        assignButton.addActionListener(this);
    }
    return assignButton;
}
```

程序员已经清除了由向导自动生成的那些指定按钮位置和大小的硬编码。然而，现在的问题是，MoveATub 在类中具有的职责不够单一。

大多数静态方法都提供了包含箱子名称与机器名称的模拟数据库。开发人员最终放弃了仅仅使用名称的工作方式，并用 Tub 对象和 Machine 对象来升级程序。程序中剩下的方法主要包含处理事件的逻辑。例如，valueChanged()方法用来管理分配按钮是否可用：

```
public void valueChanged(ListSelectionEvent e) {
    // ...
    assignButton().setEnabled(
            ! tubList().isSelectionEmpty()
        && ! machineList().isSelectionEmpty());
}
```

我们可能会考虑将 valueChanged()方法与其他事件处理方法移到一个单独的协调类中。首先注意到，调停者模式已经应用到这个类中，即组件间不能直接更新对方。例如，无论是机器组件还是列表组件，都不能直接更新分配按钮。然而，MoveATub 程序注册了列表的选择事件，根据列表中的选项是否被选中来更新按钮。在移动箱子的应用程序中，MoveATub 对象扮演了调停者，接收事件并分配相应的动作。

Java 类库的机制促使你使用调停者，尽管 JCL 并不要求应用程序必须是自己的调停者。为了避免将组件创建方法、事件处理方法与模拟数据库方法全都混合到一个类中，可以按照职责的不同，将它们写到不同的类。

**挑战 10.1**

完成图 10.3 中重构 MoveATub 后的类图，引入一个独立的模拟数据库类与一个调停者类来接收 MoveATub GUI 的事件。

答案参见第 318 页

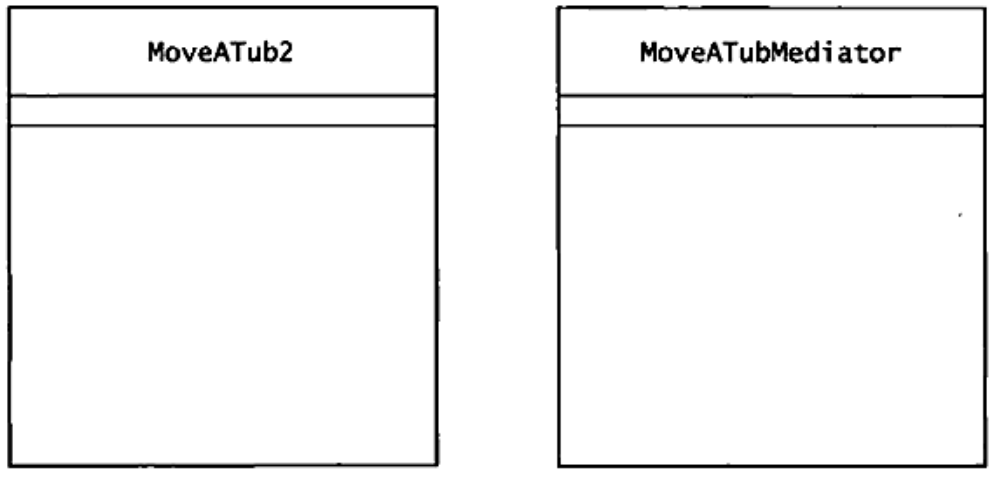


图 10.3　分离程序中的组件创建、事件处理以及模拟数据库的模块

通过重构，调停者变成一个独立的类，使得我们可以独立地开发并处理它。运行 MoveATub 程序后，组件将事件传递给 MoveATubMediator 对象。调停者可能会调用非 GUI 对象：例如，当分配完成后更新数据库。调停者对象可能也会回调 GUI 的组件：例如，当分配完成后禁用该按钮。

**挑战 10.2**

当用户单击 Do it!按钮后，绘出后面发生的事件图。标出你认为最重要的对象，并且标出在这些对象间传递的消息。

答案参见第 319 页

GUI 组件对调停者模式的运用可谓水到渠成，当事件发生时，它会通知调停者而不是直接更新其他组件。GUI 应用可能是最常见的使用调停者模式的程序了。但在其他场景下，可能也需要引入调停者模式。

无论何时，只要对象之间存在复杂的交互行为，就可以将这些交互职责集中到这些对象之外的一个调停者对象中。这就形成了松耦合——减少了对象间的直接交互。在一个独立类中，管理对象间的交互还能简化与标准化对象间的交互规则。比如，当需要管理关系的一致性时，可以使用调停者模式。

## 关系一致性中的调停者模式

面向对象的方法总是让人很容易地将现实世界中的对象与 Java 对象映射起来。然而，用 Java 对象来建模现实世界至少存在两个基本的缺陷。首先，现实世界中的对象可以随时间的变化而变化，而 Java 却没有这种内建的机制。例如，赋值语句清除了语句左边对象之前的值，却没有记住该值，但是人却可以。其次，在现实世界中，关系和对象一样重要，但是包括 Java 语言在内的面向对象语言中，关系却没有任何的支持。例如，没有内建机制支持这样的事实：如果压缩机 2402 在 1 号车间，则 1 号车间一定包含压缩机 2402。事实上，这种关系很可能出错，这促使我们引入调停者模式。

考虑 Oozinoz 公司的药品箱。箱子总是被分配给某台特定机器。可以用表格来对这种关系进行建模，如表 10.1 所示。

表 10.1　在表格中记录关系信息以保持关系一致性

| 箱　子 | 机　器 |
|---|---|
| T305 | StarPress-2402 |
| T308 | StarPress-2402 |
| T377 | ShellAssembler-2301 |
| T379 | ShellAssembler-2301 |
| T389 | ShellAssembler-2301 |
| T001 | Fuser-2101 |
| T002 | Fuser-2101 |

表 10.1 展示了箱子与机器间的关系——两者间是如何相互作用的。从数学的角度来说，关系是所有有序对象序列对的子集。因此，箱子与机器间存在一种关系，反过来，机器与箱子间也存在一种关系。保持表中箱子列的名称唯一，可以保证一个箱子不会同时分配给两台机器。

下面给出了对象模型中关系一致性的严格定义。

### 关系一致性

在对象模型中，如果在对象 a 指向对象 b 的同时，对象 *b* 也指向对象 *a*，则称对象 a 与对象 b 关系一致。

在更严格的定义中，假定有 Alpha 和 Beta 两个类，A 代表 Alpha 类的一组对象集合，B 代表 Beta 类的一组对象集合，让 a 和 b 分别代表 A 和 B 的成员，让有序对(a,b)代表 a∈ A，同时 b∈ B。这种引用既可以是直接引用，也可以是集合引用，比如对象 a 拥有一个集合对象，而该集合包含了 b。

A×B 的笛卡儿积是所有 a∈ A 和 b∈ B 的有序对(a,b)的集合。集合 A 和 B 可以产生两个笛卡儿积 A×B 和 B×A。A 和 B 的对象模型关系是存在于 A×B 对象模型的子集。让 AB 代表这个子集，BA 代表该模型中 B×A 的子集。

任意二元关系 R⊆A×B 都有一个逆关系 R⊆B×A，定义如下：

$(b, a) \in R^{-1}$ 当且仅当$(a, b) \in R$

如果对象模型是一致的，则 AB 的逆关系提供一组 B 实例到 A 实例的引用。换句话说，当且仅当 BA 与 AB 互逆时，Alpha 类与 Beta 类才是关系一致的。

当在表格中记录箱子和机器之间的关系信息时，可以限制每一个箱子仅在箱子列中出现一次，以此保证箱子与机器一一对应。在关系型数据库中，一种做法是用箱子列作为表的主键。在现实中存在这样的模型，当且仅当$(a, b) \in R$，$(b, a) \in R^{-1}$ 时，一个箱子就不会同时出现在两个机器中。

对象模型无法保证关系一致性如关系模型一般简单。考虑 MoveATub 应用程序，开发者会逐渐修改设计，不再使用名称，而是使用 Tub 对象与 Machine 对象来做设计。当现实世界中箱子在机器附近时，代表箱子的对象将会引用代表机器的对象。每一个机器对象都会有一组箱子对象集合，该集合表示在机器附近的箱子。图 10.4 展示了一个典型的对象模型。

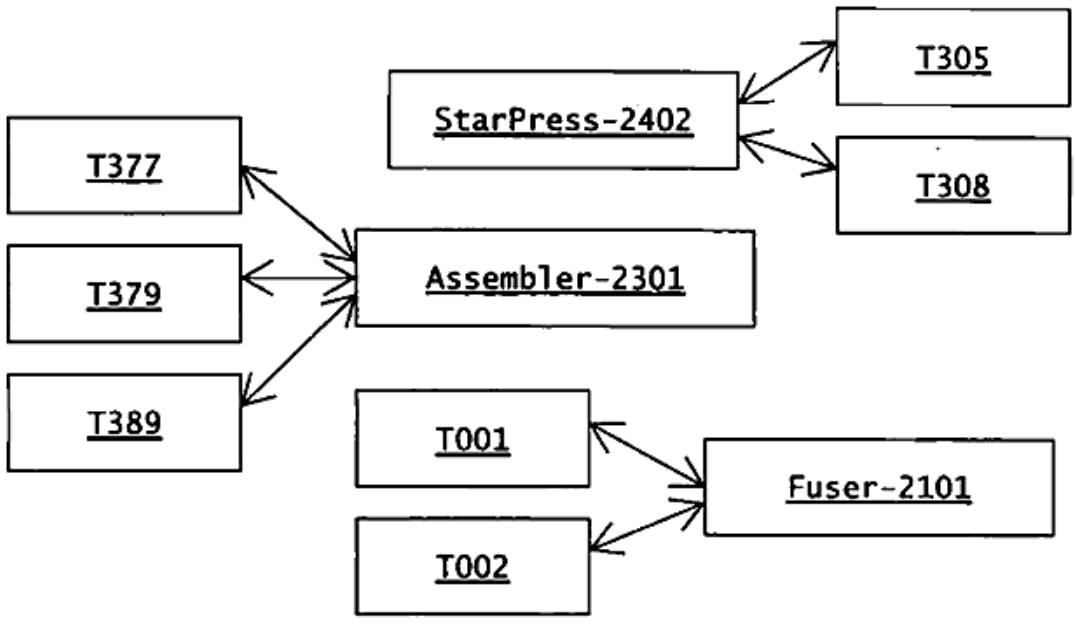


**图 10.4　对象模型中分布的关系信息**

图 10.4 中的箭头表示箱子知道所属机器，机器也知道包含了哪些箱子。箱子与机器的关系信息现在分布在多个对象中，而不是集中在一个表里。箱子与机器关系的分布化，增加了管理的难度。这时就需要考虑调停者模式。

思考 Oozinoz 公司的一个问题：一名开发人员开始对一台新机器建模，该机器包含了读取箱子条形码的功能。通过扫描箱子的 ID，开发人员通过如下代码将箱子位置 t 的信息通知给机器 m。

```
// 在 tub 设置机器，而在机器中添加 tub
t.setMachine(m);
m.addTub(t);
```

**挑战 10.3**

假设对象的初始状态如图 10.4 所示，对象 t 代表箱子 T308，对象 m 代表机器 Fuser-2101。完成图 10.5 中的对象图，以展示执行更新箱子位置代码的效果。看看出现了什么问题？

答案参见第 319 页

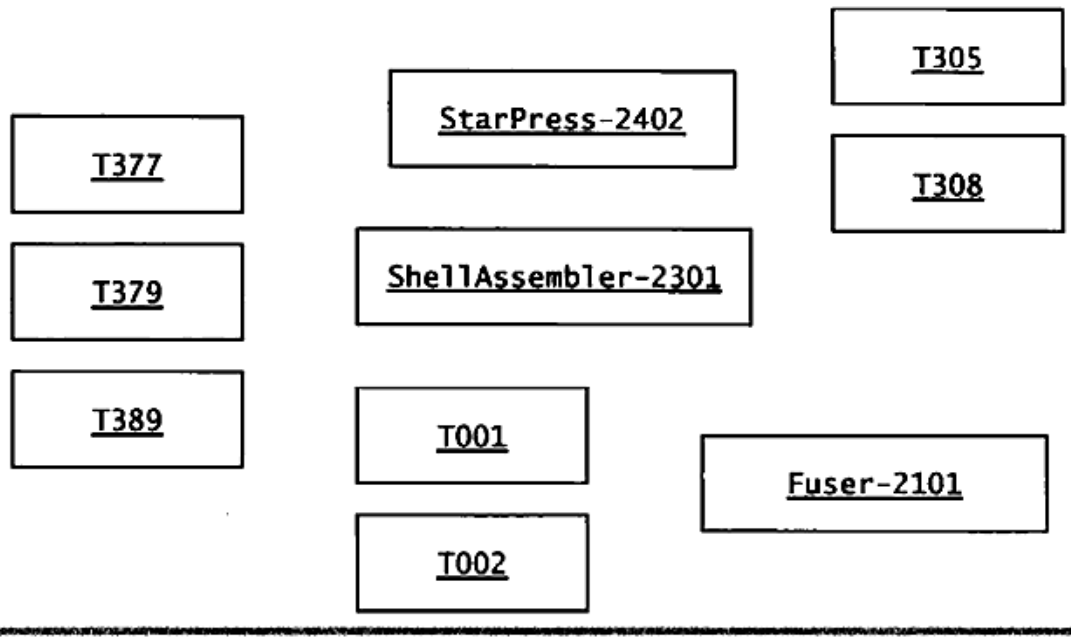


**图 10.5　完成此图后，在更新箱子位置时，将看到出现的问题**

管理关系一致性最简单的做法是：将关系信息拉回到到单个表中，并通过调停者对象进行管理，而不让机器与箱子相互知道对方。所有对象只需要维护一个到调停者对象的引用。这个“表”可以是 Map 类（Java.util 包中）的一个实例。图 10.6 展示了类图中的调停者对象。

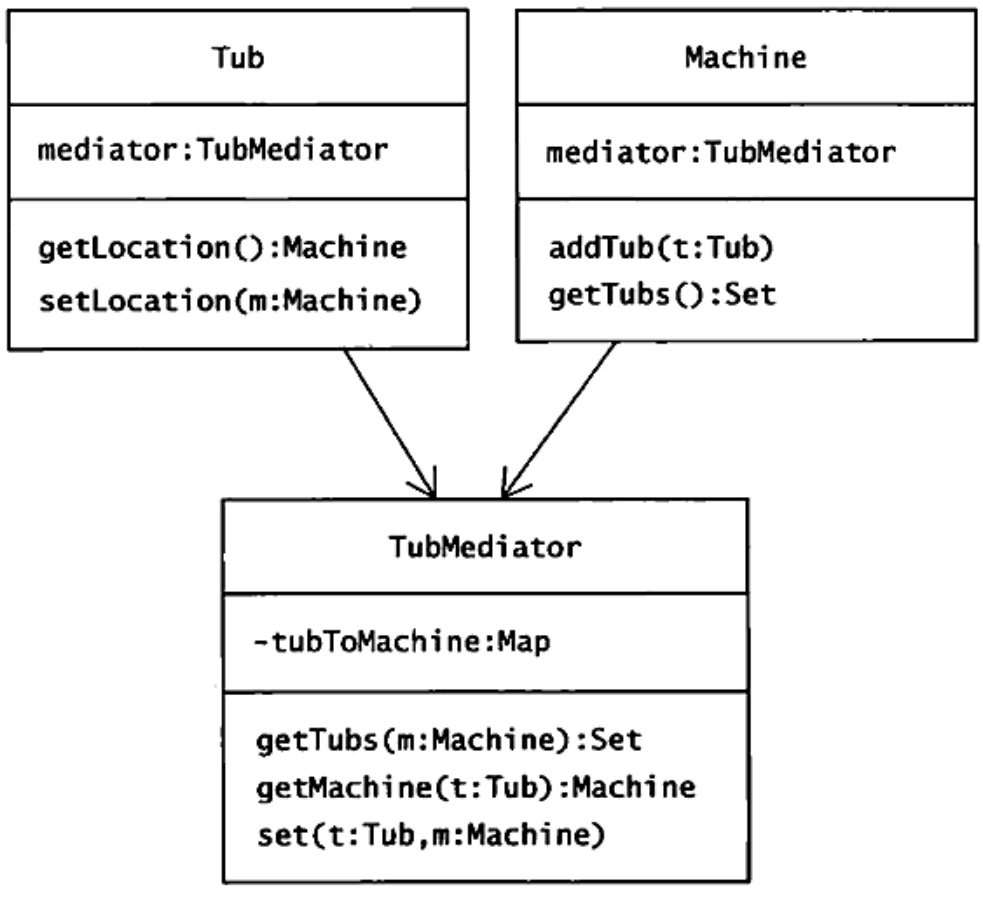


**图 10.6　箱子和机器依赖于一个调停者，通过该调停者来控制箱子与机器的关系**

Tub 类中包含一个位置信息，允许记录箱子离哪台机器比较近。代码保证了一个箱子在同一时间只能在一个地方，并且使用 TubMediator 对象来管理箱子与机器的关系：

```
package com.oozinoz.machine;
public class Tub {
    private String id;
    private TubMediator mediator = null;

    public Tub(String id, TubMediator mediator) {
        this.id = id;
        this.mediator = mediator;
    }



    public Machine getLocation() {
        return mediator.getMachine(this);
    }

    public void setLocation(Machine value) {
        mediator.set(this, value);
    }

    public String toString() {
        return id;
    }

    public int hashCode() {
        // ...
    }

    public boolean equals(Object obj) {
              // ...
    }
}
```

Tub 类的 setLocation()方法使用一个调停者来更新箱子的位置，并委托其维护与调停者之间的关系一致性。Tub 类实现了 hashCode()与 equals()方法，以便在散列表中适当存储 Tub

对象。代码细节如下所示：

```
public int hashCode() {
    return id.hashCode();
}
public boolean equals(Object obj) {
    if (obj == this) return true;

    if (obj.getClass() != Tub.class)
        return false;
    Tub that = (Tub) obj;
    return id.equals(that.id);
}
```

TubMediator 类使用 Map 对象来存储箱子与机器之间的关系。通过将关系保存在单个表中，调停者可以保证对象模型永远不会存在两台机器同时处理一个箱子的情况。

```
public class TubMediator {
    protected Map tubToMachine = new HashMap();

    public Machine getMachine(Tub t) {
        // 挑战!
    }

    public Set getTubs(Machine m) {
        Set set = new HashSet();
        Iterator i = tubToMachine.entrySet().iterator();
        while (i.hasNext()) {
            Map.Entry e = (Map.Entry) i.next();
            if (e.getValue().equals(m))
                set.add(e.getKey());
        }
        return set;
    }

    public void set(Tub t, Machine m) {
        // 挑战!
    }
}
```

**挑战 10.4**

写出 TubMediator 类中 getMachine()和 set()方法的代码。

答案参见第 320 页

如果没有引入调停者类，而是通过在 Tub 类与 Machine 类中增加逻辑，你可能无法保证两台机器不会同时处理一个箱子。不过，对于化学箱子以及机器而言，这种关系一致性的逻辑几乎无效，也容易出错。特别是容易导致箱子被移到新机器中，更新了箱子与新机器，却忘记更新箱子在原机器中的位置。将关系管理的代码逻辑转移到调停者，由这样一个独立的类来封装对象间的交互逻辑。

将关系管理的代码逻辑写到一个调停者中，让一个独立的类来封装对象间的交互逻辑。对调停者而言，很容易保证在更改 Tub 对象的位置时，自动将箱子从原机器上移除。下面来自 TubTest.java 的 JUnit 测试代码体现了这种行为：

```
public void testLocationChange() {
    TubMediator mediator = new TubMediator();
    Tub t = new Tub("T403", mediator);
    Machine m1 = new Fuser(1001, mediator);
    Machine m2 = new Fuser(1002, mediator);

    t.setLocation(m1);
    assertTrue(m1.getTubs().contains(t));
    assertTrue(!m2.getTubs().contains(t));

    t.setLocation(m2);
    assertFalse(m1.getTubs().contains(t));
    assertTrue(m2.getTubs().contains(t));
}
```

如果存在一个不涉及关系型数据库的对象模型，则可以使用调停者来维持模型的关系一致性。将关系管理逻辑转移给这些调停者，让它们专门维护模型的关系一致性。

**挑战 10.5**

与其他模式一样，调停者模式也将逻辑从一个类转移到另一个新类中。请列出另外两种模式，它们都是通过重构将现有类或者继承关系中的行为移出的。

答案参见第 321 页

# 小结

调停者模式提供了松耦合的对象关系，避免了关联对象的显式引用。该模式经常应用在 GUI 程序开发中，使用它可以让你不用关心对象间复杂的更新关系。Java 架构会指导你使用该模式，鼓励你定义一些对象来监听 GUI 事件。如果你用 Java 开发用户界面，很可能需要使用调停者模式。

尽管在创建 GUI 程序时，Java 会要求使用调停者模式，但它并不要求将这个调停者移出应用类。不过，这种做法其实可以简化代码。调停者类可以集中处理 GUI 组件的交互，应用程序类则可以集中处理组件的构建。

还有其他一些例子可以引入调停者对象。例如，可能需要一个调停者来集中管理对象模型的关系一致性，甚至可以在任何需要封装对象交互的地方使用调停者模式。