第6章

桥接（Bridge）模式

桥接模式的英文为 Bridge，桥接模式关注抽象的设计。抽象是指包含了一组抽象方法的类，这些抽象方法可能包含多个实现。

实现抽象的一般做法是创建类的层次结构，该层次的顶部是一个包含抽象方法的抽象类；该类的每个子类都提供这些抽象方法的不同实现。但是，当需要对该层次进行子类化时，这一做法就存在不足了。

你可以创建一个桥，然后把这些抽象方法移到接口中。这样，抽象就将依赖于接口的实现。

桥接模式的意图是将抽象与抽象方法的实现相互分离来实现解耦，以便二者可以相互独立地变化。

常规抽象：桥接模式的一种方法

从某种意义上说，如果认为每个类都是实体模型的近似化、理想化或者简单化，那么几乎每个类都是一个抽象。但在讨论桥接模式时，我们用“抽象”这个词专指依赖一组抽象操作的类。

假设 Oozinoz 公司有一些机器控制类，这些类控制物理机器来生产焰火弹。它们操作机器的方法各不相同。你可能需要创建一些抽象方法，使得所有机器达到同样的效果。图 6.1 展示

了 com.oozinoz.controller 包中当前的控制类。

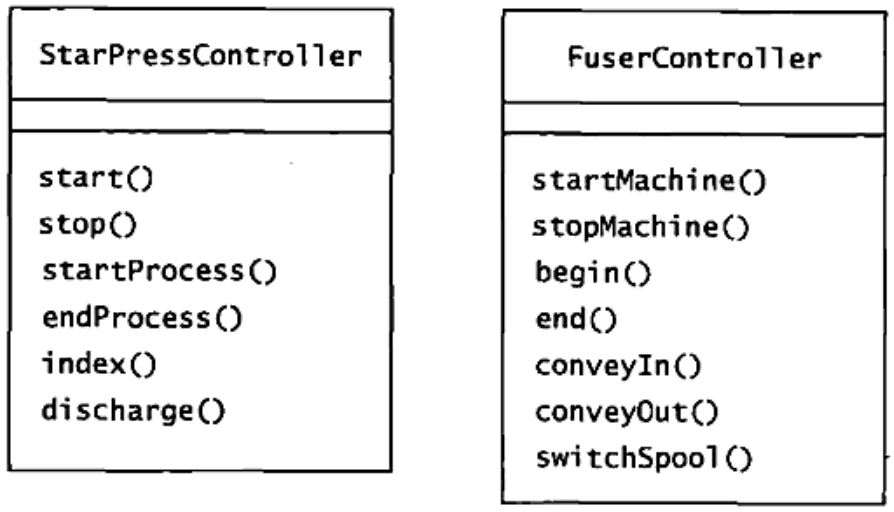

图 6.1　两个类展示了相似的方法，你可以将它们抽象到操作机器的通用模型中

在图 6.1 中，尽管 StarPressController 类将这两个方法命名为 start()和 stop()，而 FuserController 类则将它们命名为 startMachine()和 stopMachine()，但它们表达的意义都是启动和停止机器。两个控制类都包含了处理箱子的操作：将箱子移到加工区的方法（index()和 conveyIn()），开始处理和结束处理的方法，以及移除箱子（discharge()和 conveyOut()）的方法。FuserController 类还包含一个用于切换使用备用引线线圈的方法。

现在假设你要创建一个 shutdown()方法，在两台机器上执行相同的步骤，以确保有序地关机。为了简化 shutdown()方法的编写，需要为通用方法的命名标准化，例如 startMachine()、stopMachine()、startProcess()、stopProcess()、conveyIn()和 conveyOut()。但是，我们不能改变控制类，因为其中的某些方法是由机器自身提供的。

> **挑战 6.1**
>
> 请阐述如何运用一个设计模式，通过一个公共接口来控制不同的机器。
>
> 答案参见第 308 页

图 6.2 展示了一个含有子类的 MachineManager 抽象类，它可以转发机器控制指令，适配它们到 FuserController 和 StarPressController 支持的方法上。

如果机器的控制器包含一些只针对特定机器类型的操作，则没有问题。例如，FuserManager 类包含一个 switchSpool()方法（没有在图 6.2 中展现出来），该方法会转发到

FuserController 对象的 switchSpool()方法中。

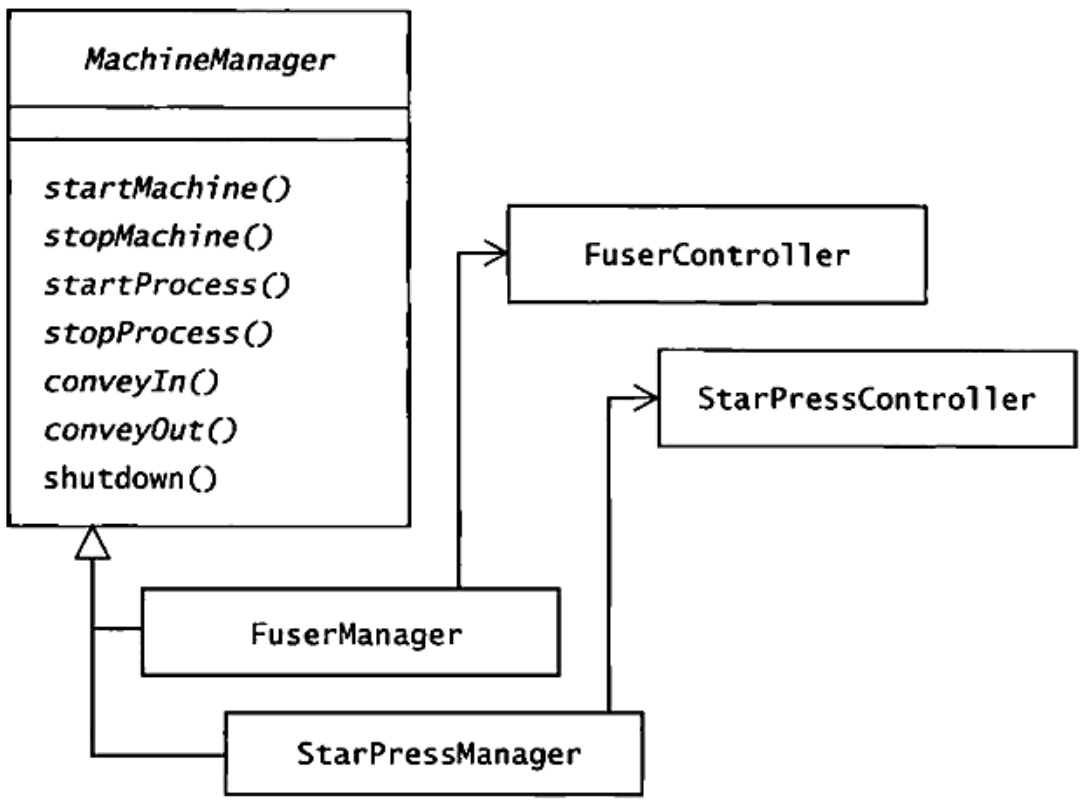

图 6.2 FuserManager 和 StarPressManager 类将调用传递给 FuserController 和 StarPressController 对象中相应的方法，以此实现 MachineManager 中的抽象方法

挑战 6.2

请写出 MachineManager 类的 shutdown()方法，该方法会停止处理过程。取出正在处理中的箱子，并且停止机器。

答案参见第 308 页

虽然 MachineManager 类的 shutdown()方法是具体方法，而非抽象方法，然而，我们仍可以说它是抽象的，因为关闭设备的步骤已经被泛化。

从抽象到桥接模式

根据不同设备的排列定义 MachineManager 的层次结构，则每个机器类型都需要

MachineManager 类的不同子类。如果需要增加新的排列方式，则类的层次结构会发生什么变化呢？例如，假设直接在机器上工作，每当机器完成一个步骤后都会提供一个反馈。与之对应，就需要创建 MachineManager 类的握手子类，允许设置我们与机器交互的参数，比如设置一个超时时间。然而，我们仍然需要不同的 MachineManager 对象管理火药球的压入与引线。如果不修改 MachineManager 类的层次结构，新的类层次结构就如图 6.3 所示。

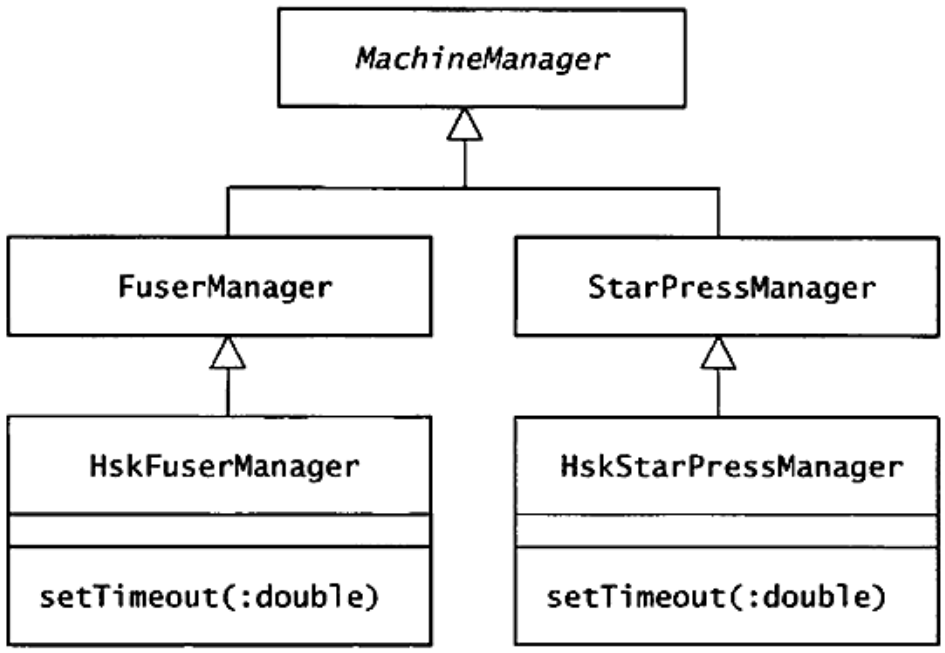

图 6.3　握手（Hsk）子类增加了一个用于表示等待实际机器反馈需要多长时间的参数

图 6.3 中的类层次结构有两种不同的排列：一种依据机器类型，另一种依据是否支持握手协议[译注1]。这种对偶原则（dual principle）会带来诸多问题。特别如 setTimeout()这样的方法可能会重复出现在两个地方，却不能将该方法提升到超类中，因为它的超类并不支持握手协议。

支持握手协议的类通常无法共享代码，因为并不存在支持握手协议的超类。随着我们向该层次结构中添加更多的类，问题会变得更加糟糕。假如，最终需要 5 台机器的控制器，从而要求修改 setTimeout()方法，就必须同时更改 5 个地方。

此时，就是桥接模式粉墨登场的时候了。通过将 MachineManager 的抽象方法分离到单独的类层次中，可以完成抽象方法的实现与抽象的解耦。MachineManager 类依然是抽象类，调

译注1：即常说的一个对象，承担了两个及以上具有不同变化方向的职责。它事实上违背了单一职责原则，即一个对象只能有一个引起它变化的原因。此时，我们常常需要分离职责，并对分离出去的职责进行抽象，再以组合的方式与原有对象进行协作。

用其方法则取决于我们是要控制火药球的压入还是引线。

将抽象从抽象方法的实现中分离出来，使得两个类层次结构能够独立变化。在不影响 MachineManager 类层次结构的情况下，我们能够添加新机器；也可以在不改变机器控制器的情况下，对 MachineManager 的类层次结构进行扩展。图 6.4 展示了我们所期望的分离效果。

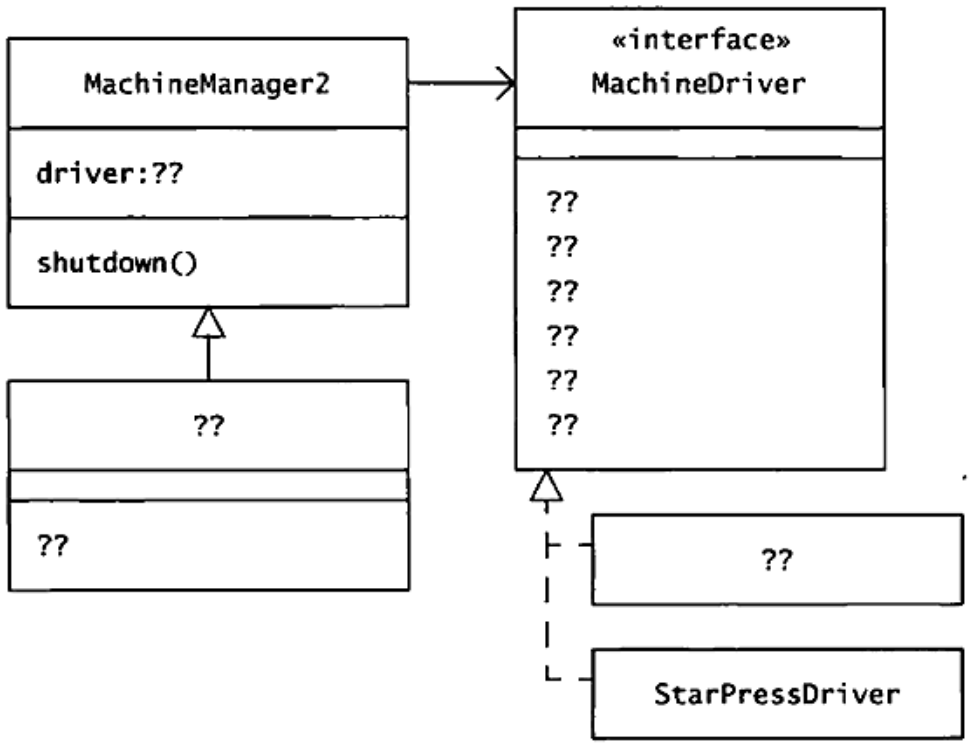

图 6.4 完成此图后，将展现 MachineManager 抽象类与其抽象方法的实现相分离

新的设计目标就是将 MachineManager 类层次结构与其抽象方法的实现分离。

挑战 6.3

图 6.4 描述了将 MachineManager 类层次结构重构为桥接模式，请填写缺失的部分。

答案参见第 308 页

注意在图 6.4 中，尽管 MachineManager2 类是抽象类，但是它却是具体功能的体现。MachineManager2 类所依赖的抽象方法现在被放到 MachineDriver 接口中[译注2]。该接口的名称

译注2：这一说法似有不妥。直观上看，MachineManager2 仍然体现了抽象层面的接口，相反，相关的 Driver 类则代表了实现，而 MachineDriver 则是对实现的抽象。

就代表它成为了驱动器（driver），用以将 MachineManager 的请求转到指定的机器上。驱动器（driver）对象能够根据定义好的接口操作计算机系统或外部设备。驱动器是最为常见的桥接模式的实践。

使用桥接模式的驱动器

驱动器是抽象的。应用程序的执行结果取决于当前执行的是哪个驱动器。每个驱动器都是适配器（Adapter）模式的一个实例。它们通过调用具有不同接口的类提供的服务，向客户提供期望的接口。使用驱动器的整体设计就是一个桥接模式的实例。这样的设计将应用程序的开发从驱动器的开发中分离出来，驱动器实现了应用程序所依赖的抽象方法。

基于驱动器的设计迫使你为被驱动出来的机器或系统设计出一个通用、抽象的模型。这样做的好处是，抽象端的代码可以被应用到任何可能被执行的驱动器中。为驱动器定义一系列通用方法可能会引发一些问题，例如某个驱动实体支持的行为可能会消失。重新来看图 6.1，fuser 控制器包含了 switchSpool()方法。再看修改后的设计图 6.4，这一方法到哪里去了？答案是我们将它抽象出去了。可以在新的 FuserDriver 类中定义 switchSpool()方法。然而，这可能导致抽象端的代码必须检查当前驱动器对象是否为 FuserDriver 的实例。

为了避免丢失 switchSpool()方法，我们必须让每个驱动器都实现此方法，并让某些驱动程序简单地忽略该调用[译注3]。当选择一个驱动器支持的方法的抽象模型时，常常会面临这样的抉择。我们既可以包含一些驱动程序不支持的方法，也可以将这些方法排除在外。后者可能缩小驱动程序抽象的范围，或者强制要求抽象包含一些特殊情况下的代码。

数据库驱动

数据库访问对驱动程序的使用，我们已经司空见惯。在 Java 程序中，数据库连接依赖于 JDBC，*JDBC™ API Tutorial and Reference (2/e)*（由 White 等人在 1999 年编写）很好地解释了

译注3：只有如此才能让所有驱动器保持统一的抽象。

如何运用 JDBC 的资源。简而言之，JDBC 是执行结构化 SQL 语句的应用程序接口（API）。该接口的实现类就是 JDBC 驱动程序，依赖于该接口的应用程序是抽象的，它可以工作在任何提供了 JDBC 驱动的数据库中。JDBC 架构将抽象与实现相互分离，因而两者可以独立地变化：这是一个非常好的桥接模式案例。

要使用 JDBC 驱动程序，首先需要加载它，连接数据库，然后创建一个 Statement 对象：

```
Class.forName(driverName);
Connection c = DriverManager.getConnection(url, user, pwd);
Statement stmt = c.createStatement();
```

有关 DriverManager 类如何工作的内容不在我们讨论的范围之列。这里，只需知道 stmt 是一个 Statement 对象，它能够执行 SQL 查询并返回结果集。

```
ResultSet result = stmt.executeQuery(
            "SELECT name, apogee FROM firework");
while (result.next()) {
    String name = result.getString("name");
    int apogee = result.getInt("apogee");
    System.out.println(name + ", " + apogee);
}
```

挑战 6.4

图 6.5 是一个 UML 顺序图，它展现了一个典型的 JDBC 应用程序中的消息流。请在图中填充缺失的类型名称以及消息名称。

答案参见第 309 页

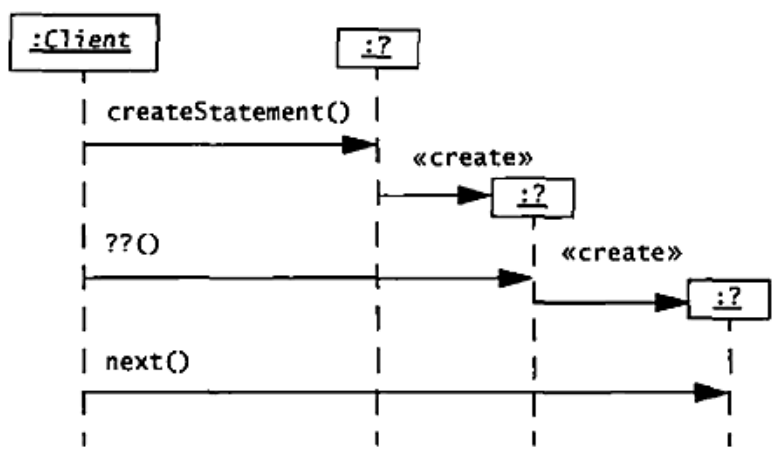

图 6.5　该图展现了 JDBC 应用程序中最为典型的消息流

挑战 6.5

假设在 Oozinoz 公司，当前仅有 SQL Server 数据库。请讨论是否应该为数据库做一套通用或专用的数据访问器或适配器。

答案参见第 309 页

JDBC 架构清晰地区分了驱动程序员与应用程序员两种角色。我们可以为抽象的超类建立驱动程序子类，每个子类负责驱动不同的子系统。在这种情形下，一旦需要更好的灵活性，就可以使用桥接模式。

小结

抽象是指依赖于抽象方法的类。最简单的抽象例子是一个抽象的类层次结构，在该结构下，超类的具体方法依赖于其他抽象方法。如果想将最初的类层次结构分解为另一种类层次结构，就必须将这些抽象方法移到另一个类层次结构中。此时，就应该运用桥接模式，完成抽象方法实现与抽象的分离。

驱动程序是最为常见的桥接模式范例，例如数据库驱动。数据库驱动程序很好地体现了在运用桥接模式时，需要对结构进行权衡的本质。一个驱动程序可能会调用一些实现者不支持的方法。另一方面，驱动程序可能会忽略一些用于特定数据库的方法。这将要求你写一些针对实现而非抽象的代码。抽象是否优于具体，或许并无定论，但有意识地去做出这些决定却是极为重要的。

第 2 部分

职责型模式