在本章中，我们将要学习Bridge模式。

【将类的功能层次结构与实现层次结构分离】

Bridge的意思是“桥梁”。就像在现实世界中，桥梁的功能是将河流的两侧连接起来一样，Bridge模式的作用也是将两样东西连接起来，它们分别是类的功能层次结构和类的实现层次结构。

Bridge模式的作用是在“类的功能层次结构”和“类的实现层次结构”之间搭建桥梁。话虽如此，当大家读到这里的时候，脑海中还是很难想象出大概的模样吧。

在开始阅读Bridge模式的示例代码之前，我们需要先来理解一下这两种层次结构。这是因为如果不能理解河流两边的土地，也就无法理解桥梁存在的意义了。

·类的功能层次结构·类的实现层次结构

|类的层次结构的两个作用

◆希望增加新功能时

假设现在有一个类Something。当我们想在Something中增加新功能时（想增加一个具体方法时），会编写一个Something类的子类（派生类），即SomethingGood类。这样就构成了一个小小的类层次结构。

Something

—somethingGood

这就是为了增加新功能而产生的层次结构。

·父类具有基本功能

·在子类中增加新的功能

以上这种层次结构被称为“类的功能层次结构”。

如果我们要继续在SomethingGood类的基础上增加新的功能，该怎么办呢？这时，我们可以同样地编写一个SomethingGood类的子类，即SomethingBetter类。这样，类的层次结构就加深了。

Something

—SomethingGood

—somethingBetter

当要增加新的功能时，我们可以从各个层次的类中找出最符合自己需求的类，然后以它为父类编写子类，并在子类中增加新的功能。这就是“类的功能层次结构”。

注意通常来说，类的层次结构关系不应当过深。

◆希望增加新的实现时·

在Template Method模式（第3章）中，我们学习了抽象类的作用。抽象类声明了一些抽象方法，定义了接口（API），然后子类负责去实现这些抽象方法。父类的任务是通过声明抽象方法的方式定义接口（API），而子类的任务是实现抽象方法。正是由于父类和子类的这种任务分担，我们才可以编写出具有高可替换性的类。

这里其实也存在层次结构。例如，当子类Concreteclass实现了父类Abstractclass类的抽象方法时，它们之间就构成了一个小小的层次结构。

AbstractClass

—ConcreteClass

但是，这里的类的层次结构并非用于增加功能，也就是说，这种层次结构并非用于方便我们增加新的方法。它的真正作用是帮助我们实现下面这样的任务分担。

·父类通过声明抽象方法来定义接口（API）·子类通过实现具体方法来实现接口（API）这种层次结构被称为“类的实现层次结构”。

当我们以其他方式实现AbstractClass时，例如要实现一个AnotherConcreteClass时，类的层次结构会稍微发生一些变化。

AbstractClass

一ConcreteClass

一AnotherConcreteClass

为了一种新的实现方式，我们继承了Abstractclass的子类，并实现了其中的抽象方法。

这就是类的实现层次结构。

◆类的层次结构的混杂与分离

通过前面的学习，大家应该理解了类的功能层次结构与类的实现层次结构。那么，当我们想要编写子类时，就需要像这样先确认自己的意图：“我是要增加功能呢？还是要增加实现呢？”当类的层次结构只有一层时，功能层次结构与实现层次结构是混杂在一个层次结构中的。这样很容易使类的层次结构变得复杂，也难以透彻地理解类的层次结构。因为自己难以确定究竟应该在类的哪一个层次结构中去增加子类。

因此，我们需要将“类的功能层次结构”与“类的实现层次结构”分离为两个独立的类层次结构。当然，如果只是简单地将它们分开，两者之间必然会缺少联系。所以我们还需要在它们之间搭建一座桥梁。本章中要学习的Bridge模式的作用就是搭建这座桥梁。

哎呀，引言说得太多了。下面我们赶紧看看Bridge模式的示例程序吧。请在阅读示例程序时着重注意上面学习的“类的两个层次结构”。

|9.3Bridge模式中的登场角色

在Bridge模式中有以下登场角色。

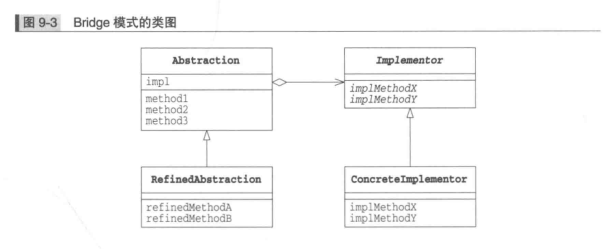
◆Abstraction（抽象化）该角色位于“类的功能层次结构”的最上层。它使用Implementor角色的方法定义了基本的功能。该角色中保存了Implementor角色的实例。在示例程序中，由Display类扮演此角色。

·RefinedAbstraction（改善后的抽象化）在Abstraction角色的基础上增加了新功能的角色。在示例程序中，由CountDisplay类扮演此角色。

Implementor（实现者）该角色位于“类的实现层次结构”的最上层。它定义了用于实现Abstraction角色的接口（API）的方法。在示例程序中，由DisplayImp1类扮演此角色。

·Concretelmplementor（具体实现者）该角色负责实现在lmplementor角色中定义的接口（API）。在示例程序中，由 stringpisplayImpl类扮演此角色。

Bridge模式的类图如图9-3所示。左侧的两个类构成了“类的功能层次结构”，右侧两个类构成了“类的实现层次结构”。类的两个层次结构之间的桥梁是imp1字段。



|9.4拓展思路的要点

|分开后更容易扩展

Bridge模式的特征是将“类的功能层次结构”与“类的实现层次结构”分离开了。将类的这两个层次结构分离开有利于独立地对它们进行扩展（具体的扩展示例请参见习题）。

当想要增加功能时，只需要在“类的功能层次结构”一侧增加类即可，不必对“类的实现层次结构”做任何修改。而且，增加后的功能可以被“所有的实现”使用。

例如，我们可以将“类的功能层次结构”应用于软件所运行的操作系统上。如果我们将某个程序中依赖于操作系统的部分划分为Windows版、Macintosh版、Unix版，那么我们就可以用Bridge模式中的“类的实现层次结构”来表现这些依赖于操作系统的部分。也就是说，我们需要编写一个定义这些操作系统的共同接口（API）的Implementor角色，然后编写Windows版、Macintosh版、Unix版的3个Concretelmplementor角色。这样一来，无论在“类的功能层次结构”中增加多少个功能，它们都可以工作于这3个操作系统上。

|继承是强关联，委托是弱关联

虽然使用“继承”很容易扩展类，但是类之间也形成了一种强关联关系。例如，在下面的代码中，SomethingGood类是Something的子类，但只要不修改代码，就无法改变这种关系，因此可以说它们之间形成了一种强关联关系。

class SomethingGood extends Something{

….

}

如果想要很轻松地改变类之间的关系，使用继承就不适合了，因为每次改变类之间关系时都需要修改程序。这时，我们可以使用“委托”来代替“继承”关系。

示例程序的Display类中使用了“委托”。Display类的imp1字段保存了实现的实例。这样，类的任务就发生了转移。

·调用open 方法会调用impl.rawopen（）方法

·调用print方法会调用impl.rawPrint（）方法·调用close方法会调用impl.rawclose（）方法也就是说，当其他类要求Display类“工作”的时候，Display类并非自己工作，而是将工作“交给imp1”。这就是“委托”。

继承是强关联关系，但委托是弱关联关系。这是因为只有Display类的实例生成时，才与作为参数被传入的类构成关联。

例如，在示例程序中，当Main类生成Display类和CountDisplay类的实例时，才将stringDisplayImp1的实例作为参数传递给Display类和CountDisplay类。如果我们不传递stringDisplayImp1类的实例，而是将其他Concretelmplementor角色的实例传递给Display类和CountDisplay类，就能很容易地改变实现。这时，发生变化的代码只有Main类，Display类和DisplayImp1类则不需要做任何修改。继承是强关联关系，委托是弱关联关系。在设计类的时候，我们必须充分理解这一点。在Template Method模式（第3章）中，我们也讨论了继承和委托的关系，大家可以再回顾一下相关部分的内容。

|9.5相关的设计模式

◆Template Method模式（第3章）在Template Method模式中使用了“类的实现层次结构”。父类调用抽象方法，而子类实现抽象方法。

◆Abstract Factory模式（第8章）为了能够根据需求设计出良好的Concretelmplementor角色，有时我们会使用Abstract Factory模式。

◆Adapter模式（第2章）使用Bridge模式可以达到类的功能层次结构与类的实现层次结构分离的目的，并在此基础上使这些层次结构结合起来。

而使用Adapter模式则可以结合那些功能上相似但是接口（API）不同的类。