第 3 章

适配器模式

如果世界是简单的，那么软件需求将会一成不变，大量应用和业务都不会得到革新。编程将会变得简单，不过同时也会令人厌烦。编程人员会遵循数年前的相同技术构建应用程序。他们不再需要引入不同的数据库，也不用实现新的最优方法或者使用不同的 API。然而，所有事物都是变化的。幸运的是，通过适配器设计模式(Adapter Design Pattern)，编程人员能够使用新的代码和功能性来帮助更新原有的系统。

名称：适配器

适配器设计模式只是将某个对象的接口适配为另一个对象所期望的接口。

3.1 问题与解决方案

在应用程序中，您也许会使用一个在体系结构上可靠稳定的工作代码库。不过，我们常常会添加新的功能，这些功能要求采用不同的方式使用现有的对象，而不是采用原先设计的方式。此时，障碍可能只是新功能需要一个不同的名字。在较为复杂的场景中，障碍也可能是新功能需要与原始对象稍有不同的行为。

针对上述问题，我们采用的解决方案是使用适配器模式构建另一个对象。这个 Adapter 对象充当了原始应用与新功能之间的中介。适配器模式为已有的对象定义了新的接口，从而能够匹配新对象的要求。

如果采用适配器设计模式，那么在绝大部分情况下，不会丢失现有的功能性，只是使用或消耗的方式不同。读者可以将这种情况视为一个将三相插座转换为两相插座的电源适配器，该适配器透明地发送来自插座管脚的交流电，不过为接地功能提供了不同的接口。在大多数常用的电源适配器中，接地功能并未丢失，不过改为通过与电源插座容器上螺丝钉相连接的接地线来提供这种功能。同样地，适配器设计模式的目标是有助于面向对象的

代码，该模式可以为对象接口创建对话。

虽然可以修改现有代码从而采用新功能所期望的方式运行，但我们最好还是创建一个适配器对象。大家常常认为，要完成这样的任务，快速调整现有对象是最快速和最划算的方式。我坚决主张下面的观点：在创建适配器对象时，速度和成本基本不会成为争论的问题。实际上，我们不会创建新的功能。到修改原始对象并进行回归测试时，我们可能不必回归就已创建了一个灵活的、具有若干代码行的适配器类。

创建 Adapter 对象仍然是最佳的解决方案，这可以提供并行开发新功能和现有代码库的可能性。如果您的工作是集成新功能性，并且要通过编辑基本代码完成该目标，那么就会发现自己与在初始类中开发新功能性的团队之间存在矛盾。开发团队可能添加另外的私有方法，同时期望稳定的最新版本原来所采用的公有方法调用这些私有方法。我们最后想要创建的是复杂的合并场景或叉状代码库。

适配器设计模式也是针对数据源改变的优秀解决方案。下面给出了与数据库引擎改变和数据文件格式改变相关的两个常见问题。

- 因为各种原因，具体项目需要改变数据库引擎。常见的场景涉及将使用 MySQL 创建的应用程序迁移至更大的数据库(如 Oracle)。平时，授权许可限制和成本要求使用不同的引擎，例如产品最终发布时要求使用 Postgres。如果没有使用过数据库抽象层，那么需要创建 Adapter 对象，从而截获对原有数据库功能性的调用并使其兼容于新的数据库。非常有趣的是，如果查看某些数据库抽象库的代码，那么会看到它们只是适配器的集合而已。
- 在使用第三方数据进行工作时，被提供的数据文件的格式可能发生变化。供应商可能在过去数年都提供 CSV 格式的数据，但是马上准备迁移至 XML 文档。为此，我们可以创建一个适配器，该适配器用于接收 XML 格式的数据，并且将这些数据以可用的格式交给稳定的 CSV 处理对象。

基本上，只要存在要求主平台持续稳定并且不使现有应用程序流程混乱的问题，在开发解决方案时就可以使用适配器设计模式。

3.2 UML

如图 3-1 所示，该 UML 图详细说明了一个使用适配器设计模式的类设计。

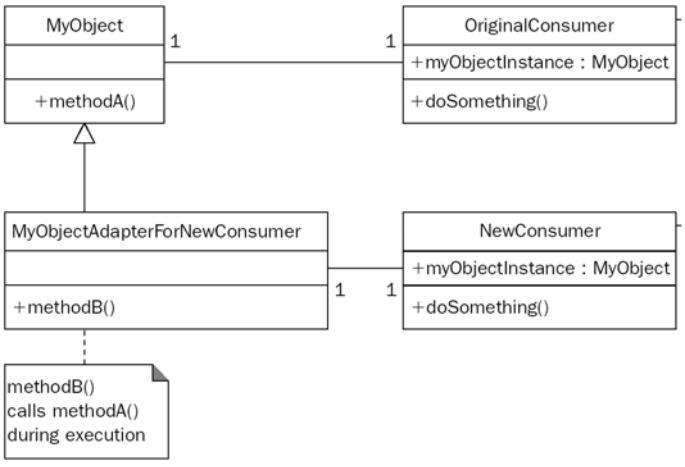

图 3-1

下面是对图 3-1 的说明：

- MyObject 类包含一个名为 methodA( )的公共方法。OriginalConsumer 类获取 MyObject 的一个实例，并且在执行其 doSomthing( )函数期间调用 methodA( )。
- 引入 NewConsumer 类。这个类的 doSomthing( )函数在执行期间期望调用 MyObject 实例的一个公共方法：methodB( )。
- 通过扩展 MyObject 类创建 MyObjectAdapterForNewConsumer 类。像 NewConsumer 期望的一样，新创建的类会提供名为 methodB( )的公共方法。在这个简单的示例中，所有 methodB( )都会调用 methodA( )。

3.3　代码示例

在项目最初的代码库中，名为 errorObject 的对象能够处理所有错误的消息和代码。最初的编程人员并不认为自己的代码会产生任何错误，因此他们设计系统将 errorObject 对象的错误消息直接输出至控制台。

下面的示例会生成一个“404: Not Found”错误。我们假定错误消息的内容和代码可能变化，但是文本始终采用相同的格式。

```
class errorObject
{
    private $__error;
```

```
        public function __construct($error)
        {
            $this->__error = $error;
        }

    public function getError()
    {
            return $this->__error;
    }
}

class logToConsole
{
    private $__errorObject;

    public function __construct($errorObject)
    {
        $this->__errorObject = $errorObject;
    }

    public function write()
    {
        fwrite(STDERR,$this->__errorObject-> getError());
    }
}

    /** create the new 404 error object **/
    $error = new errorObject("404:Not Found");

    /** write the error to the console **/
    $log = new logToConsole($error);
    $log-> write();
```

在这个场景中，项目中加入了新的网络管理员。建议的最佳做法是安装用于监控软件的网络日志。网络管理员选择的软件包要求将错误记录至一个多列 CSV 文件。具体的 CSV 格式要求第一列是数值错误代码，第二列应当是错误文本。

选择的新软件包精通于 errorObject 类，并且软件供应商已提供了实现适当日志记录格式的代码！遗憾的是，这些代码是根据 errorObject 的另一版本编写的，该版本与当前项目使用的版本不同。新的 errorObject 类具有另外两个名为 getErrorNumber( )和 getErrorText( ) 的公共方法，logToCSV 类会使用到这两个方法。

```
    class logToCSV
```

```
{
    const CSV_LOCATION = 'log.csv';

    private $__errorObject;

    public function__construct($errorObject)
    {
        $this->__errorObject = $errorObject;
    }

    public function write()
    {
        $line = $this->__errorObject->getErrorNumber();
        $line .= ',';
        $line .= $this->__errorObject->getErrorText();
        $line .= "\n";

        file_put_contents(self::CSV_LOCATION, $line, FILE_APPEND);
    }
}
```

针对这个问题，我们可以采用下面两种解决方案：

- 更改现有代码库的 errorObject 类。
- 创建一个 Adapter 对象。

考虑到保持这些公共接口标准性的需求，因此创建一个 Adapter 对象是最佳的解决方案。

新创建的适配器对象中必须存在现有 errorObject 的功能性。此外，getErrorNumber( )和 getErrorText( )公共方法必须有效。在传统的 logToConsole 类中，getError( )方法用于获取错误消息。适配器对象应当利用该方法从父类中获取错误消息，然后再转化输出供给两个新公共方法使用。

```
class logToCSVAdapter extends errorObject
{
    private $__errorNumber, $__errorText;

    public function __construct($error)
    {
        parent::__construct($error);

        $parts = explode(':', $this->getError());

        $this->__errorNumber = $parts[0];
        $this->__errorText = $parts[1];
```

```
    }

    public function getErrorNumber()
    {
        return $this->__errorNumber;
    }

    public function getErrorText()
    {
        return $this->__errorText;
    }
}
```

最后，为了实现这个适配器，必须通过使用该适配器替代原来的 errorObject 来更新代码。这样，logToCSV 类就能够接收被适配的类(而不是原来的 errorObject 类)，从而使原有的代码能够像 logToCSV 类期望的那样运行。

```
/** create the new 404 error object adapted for csv **/
$error = new logToCSVAdapter("404:Not Found");

/** write the error to the csv file **/
$log = new logToCSV($error);
$log->write();
```

需要记住的是，在需要转化一个对象的接口用于另一个对象时，实现 Adapter 对象不仅是最佳做法，而且也能减少很多麻烦。