第 7 章

委 托 模 式

面向对象编程的最强大功能之一是其拥有的动态特性。当今世界不断涌现出更多可用的功能、混合搭建结构以及持续发展的标准，动态代码具有了全新的含义。无论是新文件存储标准或流媒体标准，还是社会网站或某些现有 Internet 先锋 API 上的新事物，Web 编程总是在不断地发生突变。如今在面对大量的可用选项时，处理判决的传统方式不再有效。通过将智能化对象移动到适当的位置，委托设计模式能够远离复杂的判决。

名称：委托

通过分配或委托至其他对象，委托设计模式能够去除核心对象中的判决和复杂的功能性。

7.1 问题与解决方案

大多数 PHP 编程人员最初接触的几乎都是程序化的编程类型。这种编程样式极大地依赖于以条件语句为基础的流控制。面向对象的编程提供了某些不同于传统条件语句的手段，从而创建了更多态的代码流。这个功能的实现方法之一是创建基于委托设计模式的对象。

委托设计模式致力于从核心对象中去除复杂性。此时我们并不设计极大依赖于通过评估条件语句而执行特定功能性的对象，基于委托模式的对象能够将判决委托给不同的对象。委托既可以像使用中间对象处理判决树一样简单，也可以像使用动态实例化对象提供期望的功能一样复杂。

不要将委托设计模式视为条件语句的直接竞争者，这是非常重要的。相反，委托设计模式通过不需要条件语句就可以调用正确功能性的方式来帮助构成体系结构。条件语句最好驻留在实际方法中，并且在方法中完成对业务规则的处理。

委托设计模式的一个使用示例是为特定数据部分提供多种格式。假设在开放源代码库中存在一个归档。当访问者打算下载部分源代码时，他们可以选择两种不同格式的文件。

指定文件被压缩后将被发送至浏览器。在这个示例中，我打算采用 zip 和 tgz 压缩格式的文件。

通常，我们需要创建一个文件收集和下载对象。这个对象具有收集被请求文件以及再将对这些文件的引用存储在内部的方法。随后，专门用于指定压缩类型的方法会被调用。如果压缩类型为 zip，那么就会调用 generateZip( )方法。

我们应当使用基于委托设计模式的对象来取代上述约定的函数。generateZip( )方法的功能性应当被转移至针对基对象的文件列表执行该功能的 Delegate 类。这不仅可以减少基对象的复杂性，而且也可以为代码提供更好的可维护性。如果要采用新的压缩类型(如.dmg)，那么只需要创建新的 Delegate 对象。稳定的基对象并不需要被编辑。

当一个对象包含复杂但独立的、必须基于判决执行的功能性的若干部分时，最佳的做法是使用基于委托设计模式的对象。

7.2 UML

如图 7-1 所示，该 UML 图详细说明了一个使用委托设计模式的类设计。

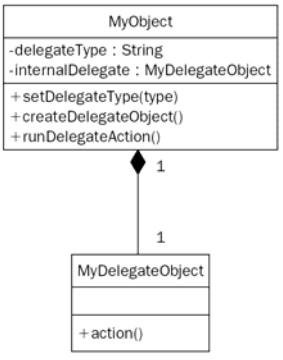

图 7-1

下面是对图 7-1 的说明：

- 基类 MyObject 知道会使用基于委托设计模式的对象。这个类包含私有字符字符串 delegateType 以及 MyDelegateObject 的私有实例 internalDelegate。
- setDelegateType( )方法接收一个名为 type 的参数，这个参数被存储在 delegateType 字符串中。
- createDelegateObject( )方法会创建委托对象的一个实例，并且根据 delegateType 变量为实例命名。随后，通过将该实例指派给 internalDelegate，这个方法将其存储在内部。

- runDelegateAction( )方法负责运行 internalDelegate 对象的 action( )方法。
- MyDelegateObject 包含负责特定动作的逻辑。MyObject 运行 action( )方法实现具体的功能。

7.3　代码示例

示例中的 Web 站点具有创建 MP3 文件播放列表的功能。MP3 文件可以来自访问者的硬盘驱动器，也可以来自 Internet。访问者可以选择以 M3U 或 PLS 格式下载播放列表。为了节省篇幅，具体的代码示例只会显示播放列表的创建部分。

如下所示，第一个步骤是创建 Playlist 类：

```
class Playlist
{
    private $__songs;

    public function __construct()
    {
        $this->__songs = array();
    }

    public function addSong($location, $title)
    {
        $song = array('location'= > $location, 'title'= > $title);
        $this->__songs[] = $song;
    }

    public function getM3U()
    {
        $m3u = "#EXTM3U\n\n";

        foreach ($this->__songs as $song) {
            $m3u .= "#EXTINF:-1,{$song['title']}\n";
            $m3u .= "{$song['location']}\n";
        }
            return $m3u;
        }

        public function getPLS()
        {
           $pls = "[playlist]\nNumberOfEntries=".count($this->__songs)."\n\n";
```

```
            foreach ($this->__songs as $songCount= > $song) {
                $counter = $songCount + 1;
                $pls .= "File{$counter}={$song['location']}\n";
                $pls .= "Title{$counter}={$song['title' ]}\n";
                $pls .= "Length{$counter}=-1\n\n";
            }
            return $pls;
        }
    }
```

Playlist 对象存储一个歌曲数组，这个数组是由构造数组初始化的。

addSong( )公共方法接受两个参数：MP3 文件的位置和该文件的标题。这两个参数组成一个关联数组，随后被添加至内部的歌曲数组。

具体的需求规定播放列表必须可以任意使用 M3U 和 PLS 格式。为此，Playlist 类具有 getM3U( )和 getPLS( )两个方法。这两个方法都负责创建适当的播放列表文件头并遍历内部的歌曲数组以完成播放列表。之后，每个方法都会返回采用字符字符串格式的播放列表。

如下所示，执行上述功能性的当前代码包含了常见的 if/else 子句：

```
$playlist = new Playlist();
$playlist->addSong('/home/aaron/music/brr.mp3', 'Brr');
$playlist->addSong('/home/aaron/music/goodbye.mp3', 'Goodbye');

if ($externalRetrievedType == 'pls') {
    $playlistContent = $playlist->getPLS();
}
else {
    $playlistContent = $playlist->getM3U();
}
```

上面的代码创建了 Playlist 对象的一个新实例，添加了两个歌曲位置和标题，接着创建了一个 if/else 子句。如果格式类型为 pls，那么就执行 getPLS( )方法，其输出被置入变量$playlist Content。否则，$externalRetrievedType 可能包含 m3u，这将进入 if/else 子句的 else 部分。

这个 Web 站点的销售团队发现至少可以使用 5 种以上播放列表格式。因此，他们在开发之前就开始推销相应的软件功能。那时，编程人员仍然不知道哪种新播放列表格式会被出售。

在这一时期，编程人员可以使用委托设计模式来更改代码。这种做法的目的是消除潜在的、难以控制的 if/else 语句。此外，随着添加更多的代码，最初的 Playlist 类会变得极为庞大。

创建 newPlaylist 类是因为意识到要使用委托设计模式的事实。PHP 动态创建基于某个变量的类实例的能力也十分有用。

```
class newPlaylist
{
    private $__songs;
    private $__typeObject;

    public function __construct($type)
    {
        $this->__songs = array();
        $object = "{$type}Playlist";
        $this->__typeObject = new $object;
    }

    public function addSong($location, $title)
    {
        $song = array('location'= > $location, 'title'= > $title);
        $this->__songs[] = $song;
    }

    public function getPlaylist()
    {
        $playlist = $this->__typeObject->getPlaylist($this->__songs);
        return $playlist;
    }
}
```

newPlaylist 对象的构造函数现在接受$type 参数。除了初始化内部的歌曲数组之外，构造函数还可以根据$type 动态地创建指定委托的新实例并将该实例内部存储在$__type Object 变量中。

addSongs( )方法与最初 Playlist 对象所包含的方法相同。getM3U( )和 getPLS( )方法被替换为 getPlaylist( )方法。这个方法执行内部存储的委托对象的 getPlaylist( )方法，它将歌曲数组传递指定对象，从而使该对象能够创建和返回正确的播放列表。

如下所示，作为 Playlist 对象原有的上述两个方法被移动至这些对象自己的委托对象：

```
class m3uPlaylistDelegate
{
    public function getPlaylist($songs)
    {
        $m3u = "#EXTM3U\n\n";
```

```
        foreach ($songs as $song) {
            $m3u .= "#EXTINF:-1,{$song['title']}\n";
            $m3u .= "{$song['location']}\n";
        }

    return $m3u;
    }
}

class plsPlaylistDelegate
{
    public function getPlaylist($songs)
    {
        $pls = "[playlist]\nNumberOfEntries=" . count($songs) . "\n\n";

        foreach ($songs as $songCount=>$song) {
            $counter = $songCount + 1;
            $pls .= "File{$counter}={$song['location']}\n";
            $pls .= "Title{$counter}={$song['title']}\n";
            $pls .= "Length{$counter}=-1\n\n";
        }

        return $pls;
    }
}
```

每个委托类本质上都只是重新包装基类 Playlist 中的原有方法。每个委托对象都具有完全相同的指定公共方法 getPlaylist( )，该方法接受 songs 参数。这种方式使基对象能够简单、动态地创建和访问任何委托者。

如下所示，执行这个基于委托的新系统的代码更为简单：

```
$externalRetrievedType = 'pls';

$playlist = new newPlaylist($externalRetrievedType);
$playlistContent = $playlist->getPlaylist();
```

当通告其他播放列表格式时，开发人员不必修改上面的代码就能够创建基于委托设计模式的新类。

为了去除核心对象的复杂性并且能够动态添加新的功能，就应当使用委托设计模式。