第7章

职责型模式介绍

对象的职责好似 Oozinoz 公司呼叫中心所代表的职责。当要从 Oozinoz 公司购买焰火弹时，你需要联系公司代表或者代理机构。他/她需要完成一些既定任务，通常，会将这些任务委托给其他系统或者其他人去完成。有时，这些代表会将这些任务请求委托给唯一的中心机构，该机构会协调各种请求，或者将任务请求抛给职责链（chain of responsibility）去完成。

就像呼叫中心代表一样，普通的对象也需要一些独立操作的信息和方法。然而，有时你却需要将对象从一般的独立性操作中分离出来，以便集中职责。很多设计模式都能满足这一需求。有的模式则通过引入对象来封装这些请求，并将该对象从依赖于它的其他对象中分离出来。面向职责的模式提供了用于集中、加强以及限制普通对象责任的技术。

常规的职责型模式

你或许对那种设计良好的类抱有强烈的意识：属性与职责应该统一在一起，不过这种理解你却只可意会不可言传。

挑战 7.1

图 7.1 所示的类结构图中至少包含 10 处职责分配的问题。请尽可能地圈出你所发现的问题，并写出其中 4 处错误的原因。

答案参见第 310 页

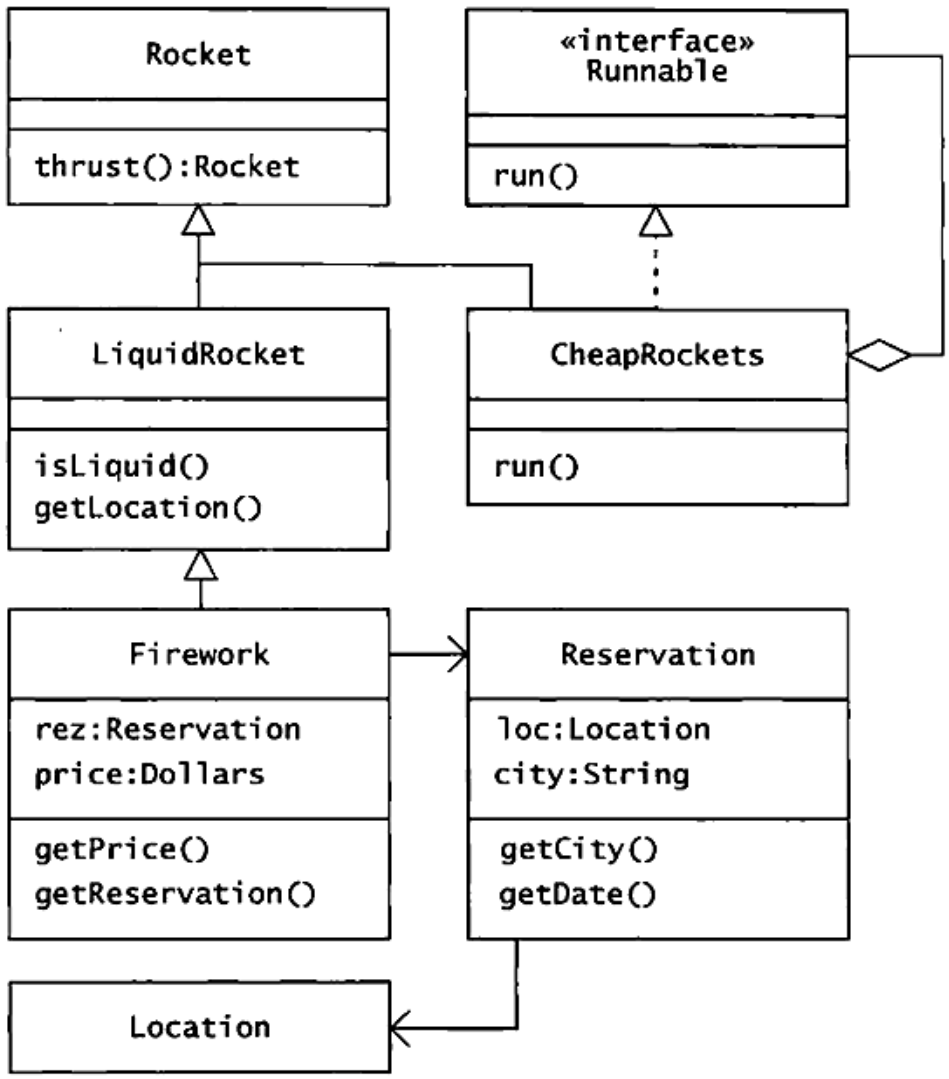

图 7.1　这幅图有什么错误

图 7.1 中所有奇怪的地方，都可能会引发你对如何正确进行对象建模的思考。当你开始定义像“类”一样的术语时，这种思考非常好。术语的价值随着人们交流的增加而增加，但如果只是为了定义术语而去定义术语，并会引起冲突，就没有必要定义术语。基于这一观念，让我们迎接下面这个困难的挑战。

> **挑战 7.2**
>
> 请定义一个高效且易于使用的类的特性。
>
> 答案参见第 310 页

一个易于使用的类的特征在于，它的方法名是有意义的，并能准确地表述方法要做的事情。然而通常所见的是，所要调用的方法，其名称并没有准确表达它内部实现的信息。

挑战 7.3

请举出一个方法名未能准确表示内部实现的例子，并给出充分的理由。

答案参见第 311 页

在面向对象的系统中，职责的合理分配所建立的原则，似乎促进了计算机科学更进一步的成熟。在一个系统中，如果每个类和方法都清晰定义了它的职责，并能正确使用它们，这个系统就是迄今为止我们所能见到最为健壮的系统。

根据可见性控制职责

我们习惯去说类和方法承担着各种各样的职责。事实上，这通常意味着你有责任提供健壮的设计，并让代码承担合适的功能。幸运的是，Java 语言分担了一部分责任。我们可以限制类、字段和方法的可见性，从而去限制其他开发人员对你开发的代码的调用。可见性向读者展示了该如何暴露类的部分内容。表 7.1 给出了访问修饰符的非正式定义。

表 7.1 访问修饰符的非正式定义

访　问	非正式定义
public	访问不受限制
(none)	只允许包内的访问
protected	只允许所在包的类，或者继承该类的子类访问
private	只允许包含它的类访问

在实践过程中，会有一些微妙的问题出现，这就需要我们更多地思考这些修饰符的正式定义。一个需要思考的问题就是，可见性是否影响了类或者对象。

挑战 7.4

一个对象是否能引用该类其他实例的私有成员？特别的，如下代码能通过编译吗？

```
public class Firework {
    private double weight = 0;
    /// ...
    private double compare(Firework f) {
        return weight - f.weight;
    }
}
```

答案参见第 311 页

修饰符的可见性能帮助我们通过限制提供给其他开发者的服务，来限制你的职责。例如，如果不希望开发者操作类的字段，就可以把该字段设置为 private。另一方面，如果想给将来的开发者提供更多弹性，则可以把字段设置为 protected，尽管这样会有子类与父类耦合过紧的风险。我们可以做出刻意为之的决策，甚至建立一套策略，既能够支持未来的可扩展性，又能通过限制访问来约束当前的职责范围。

小结

作为一名 Java 开发者，你有责任去创建拥有一组逻辑相关的属性与行为的类。设计良好的类是一门艺术，它们所拥有的基本特征更需要我们去总结。对于自己编写的类，有责任确保方法名是方法实现的准确表达。可以通过合理运用访问修饰符去限制相应的职责，但同时还需要权衡代码的安全性和灵活性。

超越普通职责

无论一个类如何限制它的成员，面向对象开发通常都会将职责分散到各个独立的对象中。换句话说，面向对象开发促进了封装，封装是指对象基于自己的数据进行操作。

职责分离是一种规范的做法，但一些设计模式却反对这种规范，并且将职责转移到中间对象或者中心对象。例如，单例模式将职责集中到一个单独的对象中，并提供对该对象的全局访

问。识别单例和其他模式意图的一种办法，就是将它们作为普通职责分离原则的反例。表 7.2 列出了不同场景下适用的模式。

表 7.2 不同场景下适用的模式

如果你的意图是	适用的模式
将责任集中到某个类的单个实例中	单例模式
将对象从依赖于它的对象中解耦	观察者模式
将职责集中在某个类，该类可以监督其他对象的交互	调停者模式
让一个对象扮演其他对象的行为	代理模式
允许将请求传递给职责链的其他对象，直到这个请求被某个对象处理	职责链模式
将共享的、细粒度的对象职责集中管理	享元模式

每种设计模式的意图都是为了解决特定场景下的问题。如果需要违背通常的原则，尽可能早地分离职责，那么就是面向职责模式粉墨登场的时候了。