第15章

构建者（Builder）模式

当你创建一个对象时，并不一定拥有创建该对象的全部信息。倘若需要逐步获得创建对象的信息，更方便的做法就是分步骤构建对象。这种情况通常发生在构建解释器和用户界面上。或者，当类的构造函数过于复杂，却对类的重点功能没有太大影响时，就可能需要让类变得更小一些。构建者模式的意图是将类的构建逻辑转移到类的实例化外部。

常规的构建者

一个使用构建者模式的常见场景是，希望对象的数据嵌套在一段文本中。随着数据解析过程的发生，你需要在查询出数据的同时进行存储。无论解析器是基于 XML 还是通过手工方式，在一开始时，可能没有足够的数据来创建完整的目标对象。引入构建者模式，就可以提供一个中间对象来存储数据，直到准备好所有数据后，再根据中间对象的数据来构造目标对象[译注1]。

假设 Oozinoz 公司除了制造焰火弹外，偶尔还提供焰火表演。旅行社会发送预约表演的请

译注1：在现在的设计方案中，构建者模式越来越倾向于以领域特定的方式，运用类似 Fluent Interface 模式，流畅而自然地表达构建的过程，并赋予客户端灵活组装的权利。例如要创建一个 SqlStatement，就可以通过如下代码来表示：new SqlStatement("select * from Order").where().orderBy()。这里的 where()与 orderBy()方法相当于后面提到的 build()方法，实现了 SqlStatement 的组装过程。

求，它的邮件如下所示：

```
Date, November 5, Headcount, 250, City, Springfield,
DollarsPerHead, 9.95, HasSite, False
```

或许，你会猜测这是一个出现在 XML 之前的协议，但是截至目前，它已经足够用了。

这个请求告诉我们潜在客户对焰火表演期望的时间和地点，以及能保证的最少人数及每位客人的服务费。在这个例子中，客户希望举办一场能容纳 250 人的焰火表演，并且会为每位客人支付 9.95 美元，或者一次性为我们的服务支付 2487.5 美元。旅行社同时还表示没有为演出提供地点。

现在的任务是将这个文本请求转换成它所代表的 Reservation 对象。可以创建一个空的 Reservation 对象，并且把它的参数设置成解析器转换的结果。这可能存在一个问题，转换后的 Reservation 对象不一定表示一个合法的请求。例如，可能在读完请求文本后才发现它丢失了日期。

可以使用一个 ReservationBuilder 类，来保证 Reservation 对象总是能代表合法的请求。ReservationBuilder 对象可以保存解析器解析出来的预订请求属性，在解析完毕后再去构建 Reservation 对象，并验证它的合法性。图 15.1 展示了该设计的类图。

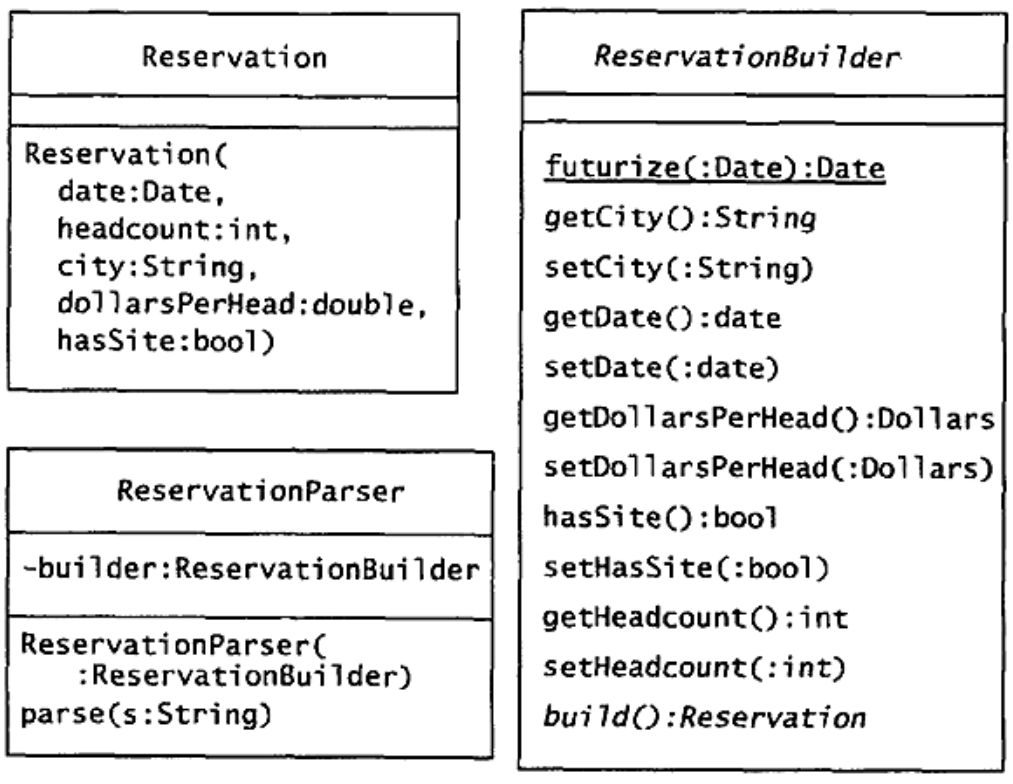

图 15.1　构建类 ReservationBuilder 将类的构建逻辑从业务类 Reservation 中移了出来，并且可以随着解析器的解析，逐步接受初始化参数来构建 Reservation 对象

ReservationBuilder 是一个抽象类，拥有一个 build()方法。我们将会根据创建 Reservation 对象的方式不同，来创建 ReservationBuilder 类的子类。ReservationParser 类的构造函数接收一个构建者传递的信息。parse()方法将信息从预订字符串中取出来，并传递给构建者。如下所示：

```
public void parse(String s) throws ParseException {
    String[] tokens = s.split(",");
    for (int i = 0; i <tokens.length; i += 2) {
        String type = tokens[i];
        String val = tokens[i + 1];

        if ("date".compareToIgnoreCase(type) == 0) {
            Calendar now = Calendar.getInstance();
            DateFormat formatter = DateFormat.getDateInstance();
            Date d = formatter.parse(
                val + ", " + now.get(Calendar.YEAR));
            builder.setDate(ReservationBuilder.futurize(d));
        } else if ("headcount".compareToIgnoreCase(type) == 0)
            builder.setHeadcount(Integer.parseInt(val));
        else if ("City".compareToIgnoreCase(type) == 0)
            builder.setCity(val.trim());
        else if ("DollarsPerHead".compareToIgnoreCase(type)==0)
            builder.setDollarsPerHead(
                new Dollars(Double.parseDouble(val)));
        else if ("HasSite".compareToIgnoreCase(type) == 0)
            builder.setHasSite(val.equalsIgnoreCase("true"));
    }
}
```

Parse()代码使用 String.split()方法来对输入字符串进行词法分析。代码中期望的预订字符串是由逗号进行分隔的，每一部分都包含了类型和值信息。String.compareToIgnoreCase()方法忽略待比较字符的大小写。当解析器读到“date”一词时，它会将随后出现的值存起来。futurize()方法将年份移到日期的前面，来确保类似“November 5”的日期会被解析为 11 月 5 日。在阅读代码的过程中，可能会注意到解析器需要多次重新遍历预订字符串。

挑战 15.1

`split(s)`调用使用正则表达式，将用逗号分隔的预订信息分隔成若干独立的字符串。请为这个正则表达式提供一个改进的方案，或者一个全新的方法，来使解析器更好地识别预订信息。

答案参见第 331 页

在约束条件下构建对象

你需要确保非法的 `Reservation` 对象永远不会被创建出来。针对这个例子，假设每个预订信息都必须有一个非空的日期和城市。假设 Oozinoz 公司的业务规则规定在人数低于 25 人，或者总费用少于 495.95 美元时，将不进行演出。我们需要将这些限制记录到数据库中。但是就目前而言，可以使用 Java 代码写出这些常量。如下所示：

```
public abstract class ReservationBuilder {
    public static final int MINHEAD = 25;
    public static final Dollars MINTOTAL = new Dollars(495.95);
        // ...
}
```

为了避免在请求不合法时创建 `Reservation` 实例，可能会在 `Reservation` 类的构造函数中进行业务逻辑检查和异常处理。但是 `Reservation` 对象一旦被创建，这些逻辑相对于 `Reservation` 本身的业务对象来讲就没用了。此时，引入一个构建者，并将 `Reservation` 的构建逻辑移植到构建者对象，将会使 `Reservation` 类本身变得简单，使其更加关注自身的业务逻辑。引入构建者还能使 `Reservation` 对象对传入的不同参数做出不同的反应。最终，将 `Reservation` 的构建逻辑移植到 `ReservationBuilder` 子类中，并在解析器解析预订字符串的过程中逐步创建 `Reservation` 实例。图 15.2 展示了 `ReservationBuilder` 的两个子类对不同的错误参数进行了不同的错误处理。

图 15.2 给出了构建者模式的一大优势：通过将 `Reservation` 类的构建逻辑抽取出来，就可以将这部分逻辑看做一个完整的任务，甚至可以为该逻辑创建一个独立的层级关系。不同的

构建逻辑对预订业务几乎没有影响。例如，在图 15.2 中，构建者之间的区别在于是否抛出 BuilderException 异常。使用构建者的代码如下所示。

```
package app.builder;
import com.oozinoz.reservation.*;

public class ShowUnforgiving {
    public static void main(String[] args) {
        String sample =
            "Date, November 5, Headcount, 250, "
            + "City, Springfield, DollarsPerHead, 9.95, "
            + "HasSite, False";
        ReservationBuilder builder = new UnforgivingBuilder();
        try {
            new ReservationParser(builder).parse(sample);
            Reservation res = builder.build();
            System.out.println("Unforgiving builder: " + res);
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```

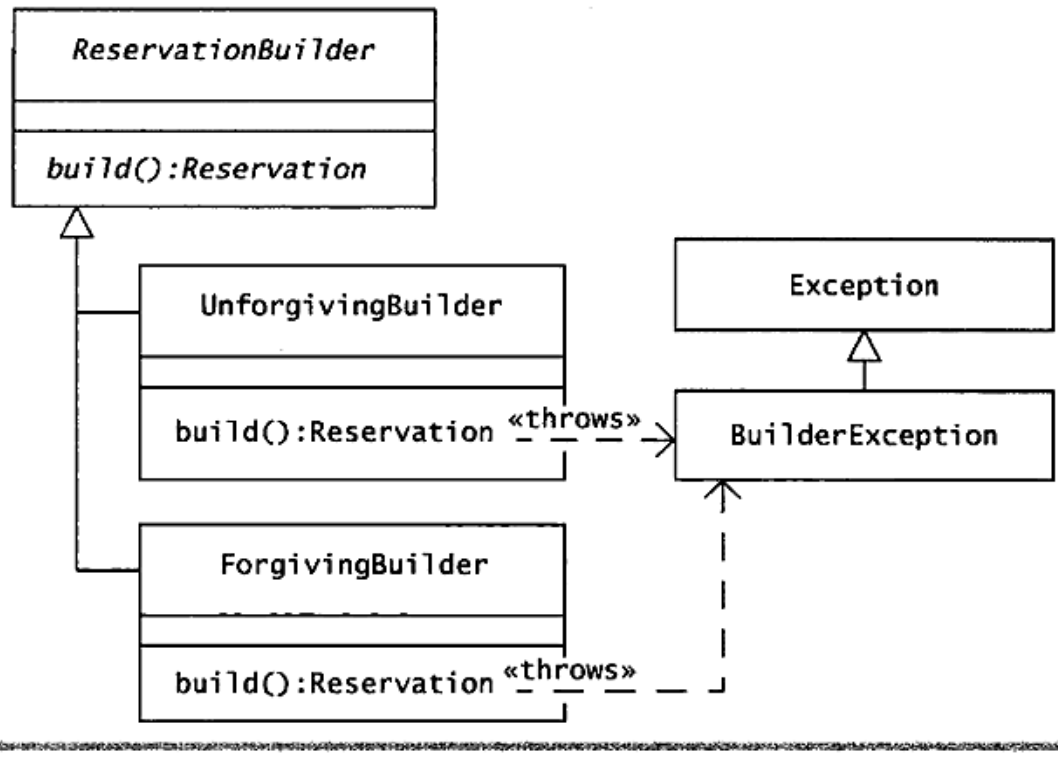

图 15.2　在面对不同的错误参数时，构建者对象会进行不同的错误处理

运行该程序将输出这个 Reservation 对象：

```
Date: Nov 5, 2001, Headcount: 250, City: Springfield,
Dollars/Head: 9.95, Has Site: false
```

当给定一个预订字符串时，代码会初始化一个构建者和一个解析器，并会让解析器去解析字符串。当解析器读到预订的属性值时，会使用构建者的 set 方法将该值传递给构建者。

转换结束后，代码会要求构建者构建一个合法的预订。当出现异常时，这个例子会简单地打印出异常信息。而在实际应用中，我们可能需要采取一些行动来处理异常。

挑战 15.2

当日期或时间为空时，或者总人数太少，再或者演出费太低时，UnforgivingBuilder 类的 build()方法都将抛出异常。请根据这些约束条件写出 build()方法的实现。

答案参见第 332 页

可容错的构建者

UnforgivingBuilder 类会拒绝任何信息不完整的请求。对请求中缺失的信息做出合理的变通方案，可能是一个更好的业务规则。

当预订请求缺少总人数信息时，Oozinoz 公司的分析员会要求为该请求提供一个最少人数。类似的，当缺少单人的金额值时，构建者应该为该属性设置一个合理的值。这些需求都很简单，但是这种设计却需要一些技巧。例如当预订字符串包含了单人的金额值，却未指定总人数时，构建者应该具有什么样的行为呢？

挑战 15.3

为 ForgivingBuilder.build()方法编写一个规则，重点是构建者如何处理缺失的人数或者单人的金额值。

答案参见第 333 页

挑战 15.4

请写出 ForgivingBuilder 类的 build()方法的实现。

答案参见第 333 页

ForgivingBuilder 与 UnforgivingBuilder 类保证你创建的 Reservation 对象都是合法的。这个设计还有一个好处，当构建一个预订对象时，如果出错，该设计可以让你有足够的弹性来处理错误。

小结

构建者模式将复杂对象的构建逻辑从对象本身抽离了出来，这样能够简化复杂的对象。构建者关注目标类的构建过程，目标类关注合法实例的业务本身。这样做的好处是在实例化目标类前，确保得到的是一个有效的对象，并且不会让构建逻辑出现在目标类本身。构建者构建类对象的过程通常是分步骤的，这使得该模式通常被应用于解析文本以创建对象的场景。