## 第18章 点菜也需要翻译——解释器模式

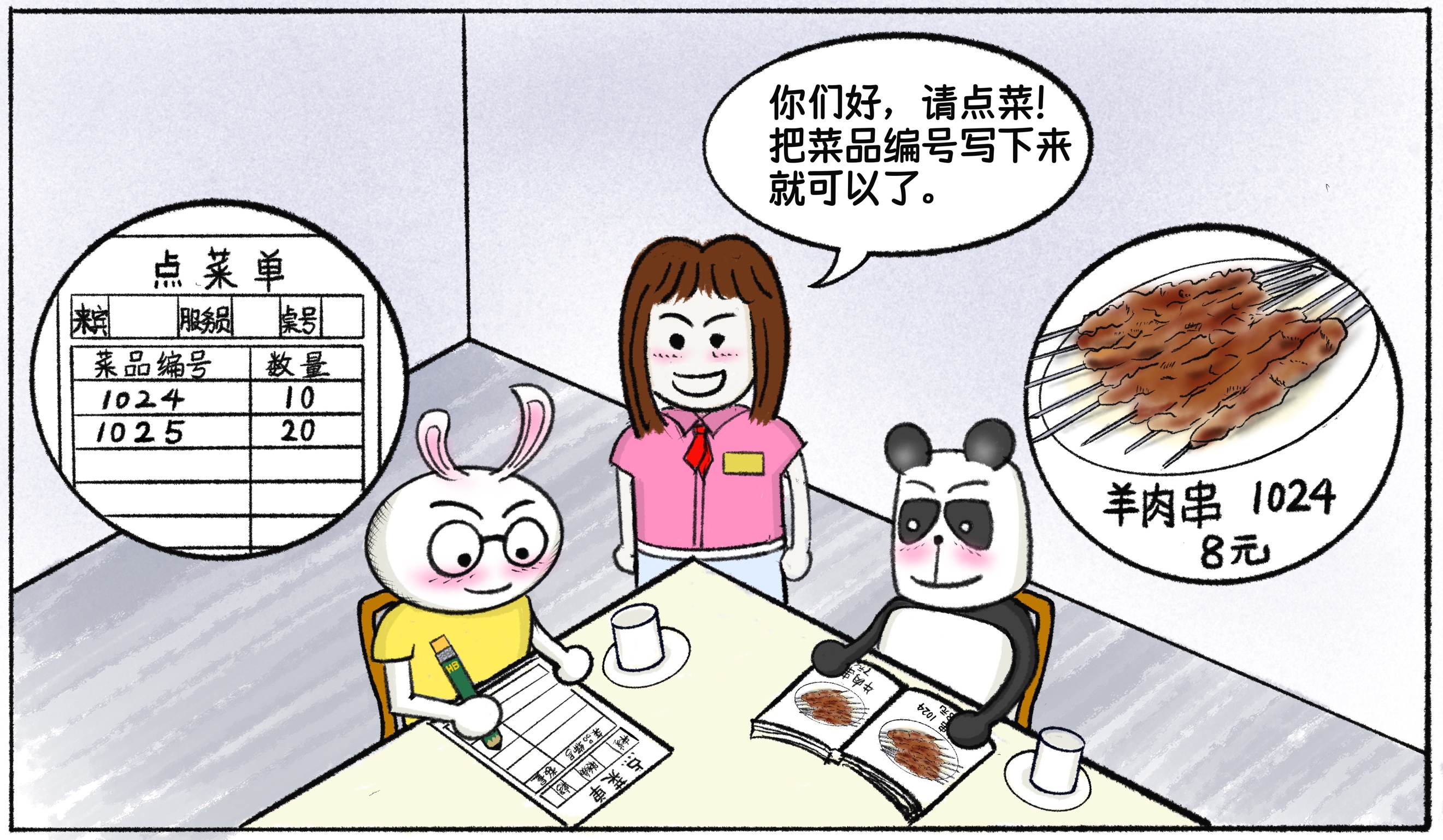
## 18.1 记忆力惊人的服务员

熊小猫：今天咱们去吃点好的，我请客，庆祝上线成功！

兔小白：这我必须得去，咱们去烧烤店撸串吧！

两人下班后来到了烧烤店庆祝项目顺利上线。

服务员：你们好，请点菜！把菜品编号写下来就可以了。



两人点完菜后，服务员进行核对。此时，兔小白突然想加一个菜。

兔小白：你好，我想加一个疙瘩汤，我查下编号。

服务员：不用查了，2032，我给你加上了。

熊小猫：不好意思，我还想再加个烤馒头片。

服务员：2043，也给你加好了。

兔小白：哇，你的记忆力也太好了吧！

服务员：每天都做同样的工作，想不记住都难呀！两位要是没有其他需要，我就下单了。

待服务员离开后，兔小白又开始赞叹服务员的记忆力。

兔小白：这位服务员太厉害了！这么多菜品的编码她都能记得！

熊小猫：确实厉害，不过这家饭店的点菜系统有些落后，还在用输入菜品编码的方式点菜。现在比较先进的点菜系统都通过用户界面操作了。

兔小白：通过用户界面点菜，需要逐一查找菜品，我觉得不如直接输入菜品编码快。当然，前提是足够熟悉菜品编码。

熊小猫：这就像Dos和Window的区别。熟悉命令行的电脑使用者，使用命令行效率更高。但命令行的输入是开放的，意味着使用者要熟悉操作系统命令的文法规则。

点菜终端也是一样，服务员需要按照一定格式输入，命令才能被正确解析。例如我们点羊肉串10串，服务员需要输入1024\*10，再输入#确认。

兔小白：这有点像通过手机办理银行业务时，让我从手机输入密码的操作。

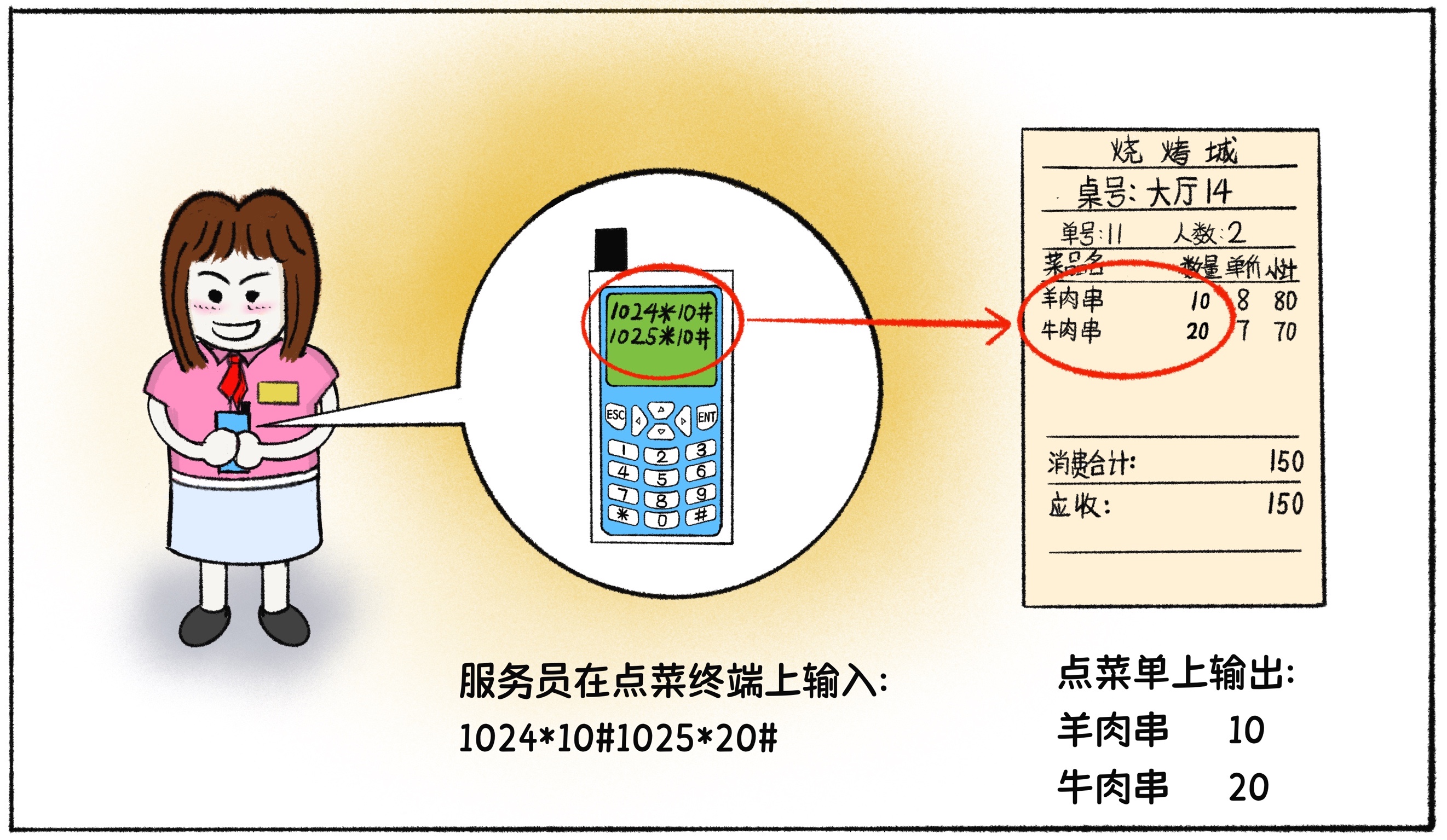
熊小猫：这两个场景是类似的。你输入一段命令后，后台服务器会按照文法规则解析你的输入。如果能够正确解析，将按照解析结果执行命令。这其实就是设计模式中的解释器模式。

兔小白：趁着还没上菜，正好给我讲讲解释器模式！

熊小猫：哈哈，你可真是走到哪学到哪！没问题，我现在就给你讲讲解释器模式。

## 18.2 何为解释器模式

熊小猫：咱们回忆一下刚才点菜的过程。服务员按照我们点的菜品输入编号和数量，确认后，再重复刚才的操作点下一个菜。点餐系统会把服务员的输入转化为点菜单打印出来。为了方便写程序练习，假设在输入菜品编号后，需要先按“\*”键再输入数量，最后按“#”键确认。然后继续输入下一个菜品。

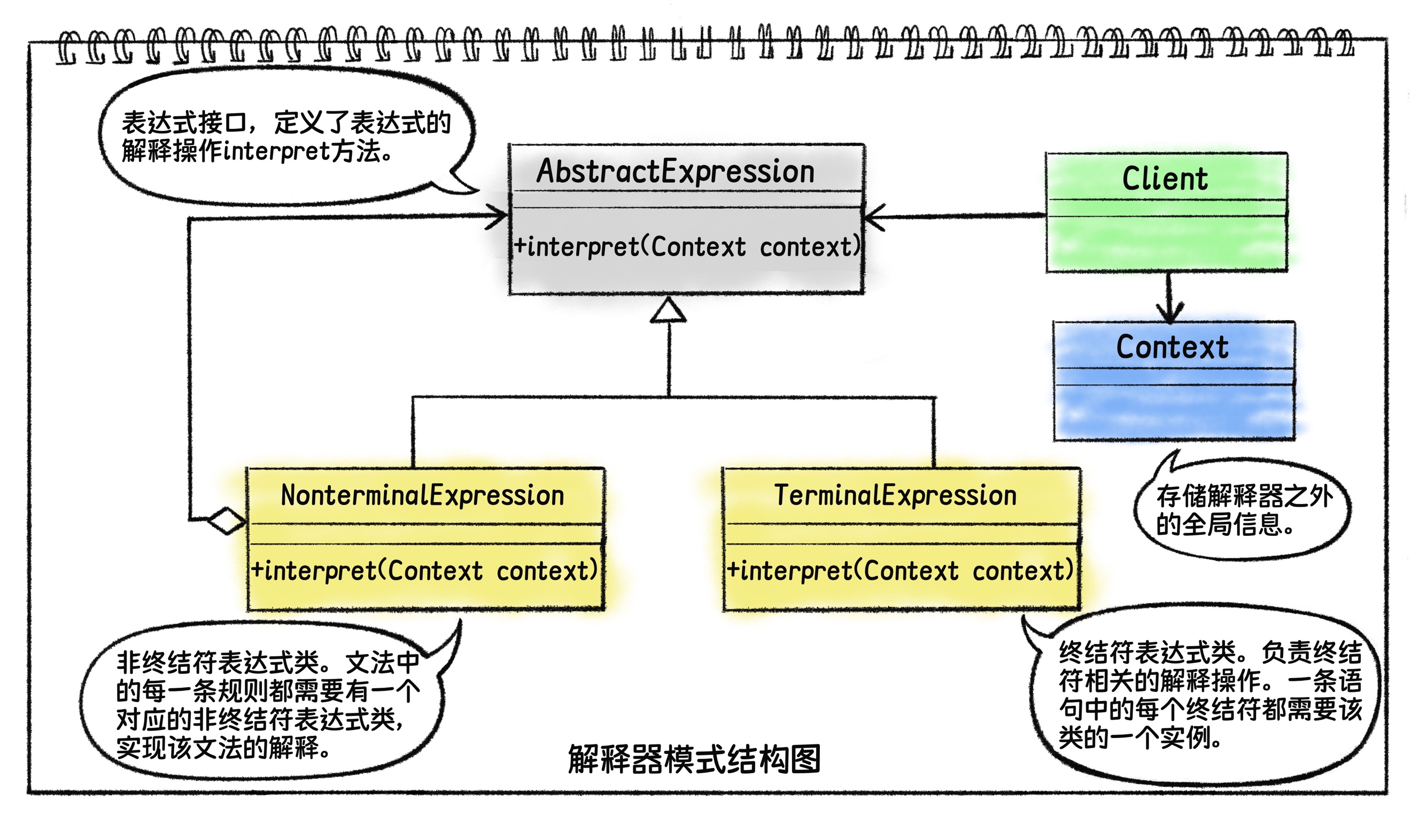


程序的目标是将输入的字符串转换为点菜单上菜品格式的输出。今天要讲的解释器模式，正是用来解决此类问题。服务员在点菜终端上的输入内容，是按照固定文法规则组织的一段文本。解释器对这段文本使用到的每一条文法规则进行解释后，这段文本最终被转化为点菜单上的文本。

兔小白：什么是文法规则呢？

熊小猫：以“\*”举例。“\*”用来连接菜品编码和数量，这就是一条文法规则。在解释器模式中，这类文法规则被抽象为非终结符表达式。还有另外一类表达式，比如具体的菜品编号和数量，会被抽象为终结符表达式。

按照固定文法规则编写的语句，可以被表示为一棵由非终结符表达式和终结符表达式对象所组成的抽象语法树。显然，终结符表达式对象是叶子节点。我们来看一下解释器的结构图。



AbstractExpression为表达式接口，定义了表达式的解释操作interpret方法。NonterminalExpression为非终结符表达式类。文法中的每一条规则，都需要有一个对应的非终结符表达式类实现该文法的解释。TerminalExpression为终结符表达式类，负责终结符相关的解释操作。一条语句中的每个终结符都需要该类的一个对象。

一条语句首先被解析成由表达式对象组成的抽象语法树，然后调用根节点对象的interpret方法，递归触发树上所有节点对象的interpret方法被调用，从而完成对整条语句的解释。

兔小白：咱们还是做个练习吧，这样会理解的更清楚。

熊小猫：没问题，理论基础已经讲完了，咱们实际操作一下。

## 18.3 使用解释器模式实现点菜系统

熊小猫：咱们就拿点菜系统做练习，使用解释器模式来实现它。你先分析一下，点菜系统有哪些文法规则。

兔小白：用来练习的点菜系统涉及到这几种文法规则。

（1）“\*”，连接菜品和数量，构成一条完整点菜单项目。

（2）“#”，代表当前菜品项输入完成。

熊小猫：这几条规则属于非终结符规则，需要有对应的非终结符规则类，实现相应的文法解释。此外还有菜品编号和数量，这两个是终结符，需要定义终结符表达式类。分析工作已经做好了，你来尝试代码实现？

兔小白：没问题，我来试试。

20分钟后，兔小白写完了代码。

兔小白：我先定义了表达式接口Expression，声明interpret方法。

public interface Expression {  
    String interpret(Context context);  
}

Context类维护了菜品编号和菜品名称的映射关系。用于编码到名城的转换。

public class Context {  
    public Map<String,String> dishMap = new HashMap<>();  
​  
    public Context(){  
        dishMap.put("1024","羊肉串");  
        dishMap.put("1025","牛肉串");  
   }  
    public String getDishName(String dishNumber){  
        return dishMap.get(dishNumber);  
   }  
}

程序中有两个非终结符表达式类，第一个为ConnectExpression，对应“\*”的文法规则，连接菜品名称和数量。

public class ConnectExpression implements Expression {  
    private final Expression dishName;  
    private final Expression amount;  
​  
    protected ConnectExpression(Expression dishName, Expression amount) {  
        this.dishName = dishName;  
        this.amount = amount;  
   }  
​  
    public String interpret(Context context) {  
        return dishName.interpret(context) + " " + amount.interpret(context);  
   }  
}

FinishDishItemExpression类对应“#”的文法规则，表示一个菜品项输入完成，负责菜品项之间换行。

public class FinishDishItemExpression implements Expression {  
    private final Expression dishOrder;  
    private final Expression nextDishOrder;  
​  
    public FinishDishItemExpression(Expression dishOrder, Expression nextDishOrder) {  
        this.dishOrder = dishOrder;  
        this.nextDishOrder = nextDishOrder;  
   }  
​  
    public String interpret(Context context) {  
        return dishOrder.interpret(context) + "\n" +  
                nextDishOrder.interpret(context);  
   }  
}

DishNumberExpression类为菜品编号终结符表达式。通过Context找到编号对应的名称，对菜品编号进行解释。

public class DishNumberExpression implements Expression {  
    private final String dishNumber;  
​  
    public DishNumberExpression(String dishNumber) {  
        this.dishNumber = dishNumber;  
   }  
​  
    public String interpret(Context context) {  
        return context.getDishName(dishNumber);  
   }  
}

DishAmountExpression为菜品数量终结符表达式，负责解释菜品数量。

public class DishAmountExpression implements Expression {  
    private final String amount;  
​  
    public DishAmountExpression(String amount) {  
        this.amount = amount;  
   }  
​  
    public String interpret(Context context) {  
        return amount;  
   }  
}

我在客户端中构造了“1024\*10#1025\*20#”语句的语法树。调用根节点joinExp对象的interpret方法.

Expression connectExp1 = new ConnectExpression(new DishNumberExpression("1024"),  
                new DishAmountExpression("10"));  
Expression connectExp2 = new ConnectExpression(new DishNumberExpression("1025"),  
                new DishAmountExpression("20"));  
Expression joinExp = new JoinDishItemExpression(connectExp1, connectExp2);  
​  
System.out.println(joinExp.interpret(new Context()));

可以看到每个Expression节点的interpret方法被递归调用，输出结果也符合我们的期望。

羊肉串  10  
牛肉串  20

熊小猫：非常不错，这版代码将解释器模式很好的应用到了点菜系统中。。

兔小白：不过有一点我觉得做的还不够好。现在的客户端程序，直接构造出了抽象语法树，不是通过解析点菜语句生成的抽象语法树。

熊小猫：这里不能说你做的不够好，其实解释器模式原本就不包括语句分析和构建抽象语法树。但是为了让例子更完整，你也尝试加上将语句分析，构建抽象语法树加上吧！

15分钟后，兔小白完成了语法分析程序的编写。

兔小白：我在客户端中增加了抽象语法树分析方法getExpressionTree。

private static Expression getExpressionTree(String orderSentence) {  
  char[] chars = orderSentence.toCharArray();  
  String buffer = "";  
  Stack<Expression> expressionStack = new Stack<>();  
​  
  for (char character : chars) {  
    switch (character) {  
      case '\_':  
        //读取到 \_ ，意味着当前菜品项已经读取完成。Buffer中的值为菜品数量。  
        //创建连接表达式对象。  
        ConnectExpression connectExpression =  
          new ConnectExpression(expressionStack.pop(),  
                                new DishAmountExpression(buffer));  
        //如果Stack不为空，说明已经生成过连接表达式对象，需要和当前连接表达式对象继续连接。  
        //如果Stack为空，说明当前连接表达式对象是分析出的第一个，直接存储到Stack中。  
        if (expressionStack.size() > 0) {  
          JoinDishItemExpression item =  
            new JoinDishItemExpression(expressionStack.pop(),connectExpression);  
          expressionStack.push(item);  
       } else {  
          expressionStack.push(connectExpression);  
       }  
​  
        buffer = "";  
        break;  
      case '\*':  
        //\*为连接字符，意味着之前读到buffer的值为菜品编号。保存菜品编号表达式对象到Stack中。  
        expressionStack.push(new DishNumberExpression(buffer));  
        buffer = "";  
        break;  
      default:  
        //读取字符到缓冲区  
        buffer = buffer + character;  
   }  
 }  
  //完成语句分析后，弹出栈中表达式。  
  return expressionStack.pop();  
}

兔小白：程序逐字符对点菜语句进行分析，在遇到“\*”或者“#”之前，会把字符拼接起来存放在buffer中。一旦遇到“\*”或者“#”则会做相应的解析处理，生成表达式节点。

客户端程序直接调用该方法，对点菜语句进行分析，然后解释执行抽象语法树。

String orderSentence = "1024\*10\_1025\*20\_";  
Expression expression = getExpressionTree(orderSentence);  
System.out.println(expression.interpret(new Context()));

## 18.4 解释器模式的适用场景

熊小猫：解释器模式的适用场景比较固定。当你需要解释执行自定义文法结构编写的语句，并且可以将该文法结构的语句解析为抽象语法树时，可以使用解释器模式。

兔小白：自定义文法结构的语句？听起来像是一门微型的编程语言。

熊小猫：没错，可以认为这就是一门微型编程语言，解释器程序做的事情就是读懂这门语言写出的语句并执行。

兔小白：我们使用的编程语言已经足够强大了，为什么还需要再自定义一种微型编程语言呢？

熊小猫：自定义文法结构，主要用来解决特定类型问题。编程语言虽然强大，但文法规则和特定类型问题不一定匹配，从而面临难以描述的问题。另外，自定义文法结构编写的语句作为文本数据，可以存储和传输，在编程语言内部使用非常方便、灵活。

一个非常熟悉的例子，就是开发时经常用到的正则表达式。正则表达式用来解决字符串匹配的特定问题。它能够描述字符串的结构，这有赖于它所定义的各种文法规则。而正则表达式作为规则数据，在程序中的存在形式只是一段文本，可以被方便的存储、传输。

解释器模式适合文法简单的场景。如果文法过于复杂，那么文法规则类会变得十分庞大，难以管理和维护。而解析语法树的工作也会变得困难，想要将程序写好，有一定门槛。

解释器模式最大的优势是文法规则易于扩展。无论是终结符还是非终结符表达式，都支持扩展。另外，不同表达式类的实现大体一致，易于直接编写。

兔小白：我发现解释器模式和组合模式的结构几乎一模一样。组合模式中的对象结构为树形，解释器模式中的规则对象也会构成语法树。这两者是不是存在联系？

熊小猫：你观察的很仔细嘛！从结构上看，这两种设计模式的相似度非常高。解释器模式其实可以看作是组合模式的一种特殊形式。当将组合模式应用于构造和解释抽象语法树时，便特化为了解释器模式。

兔小白：看来起名字很重要呀，稍作改变，起个高大上的名字，就成为了一种新的设计模式。

熊小猫：其实你仔细研究不同设计模式的结构图，就会发现有多种设计模式的结构只存在细微的差别。万变不离其宗，设计模式的结构都是基于几种基本的类关系，因此差异不会太大。

在解决一类新问题的时候，如果已经存在比较匹配的设计模式，当然可以拿来直接修改使用，从而构成一种新的设计模式。任何解决方案，都是以问题为出发点。起一个和解决的问题相匹配的名称，以后遇到类似的问题时，就能快速匹配到对应的设计模式。

兔小白：没想到吃饭前又学了一种设计模式！已经上菜了，咱们开动吧！咦，咱们没点烤板筋呀？是不是点菜系统的解释器出问题了？

熊小猫：系统哪儿那么容易出问题？一定是你抄错菜品编号啦！

兔小白：哎呀！我还是更喜欢用户界面的点菜系统！！