第 1 章

重构-入门示例

应该如何开始学习重构呢？通常情况下，我们会先了解重构的演变史、重构的核心原理等内容，但这样的讲述方式，难免会让人感觉枯燥乏味。不如直接给出重构示例，直奔主题。

一个恰当的示例，能够让我们快速地了解重构是如何进行的。仅谈原理，无疑是纸上谈兵，很难落地到实际的项目案例中。而恰当的示例，能够帮助我们快速入门。

因此，我们以示例进行开篇，并在对该示例进行拆解的过程中融入重构方式的相关内容，让读者逐步了解如何进行重构。等入门示例学习完成后，我们会在第 2 章进行重构原理的介绍。

很能理解作者所面临的困境，就好比我们在学习设计模式的过程中，很难用真实的、全面的项目案例去呈现设计模式的使用方式。但这并不影响此书的价值，我们仍然可以通过学习此书，掌握重构的技巧。

选择一个恰当的示例，并不是一件容易的事情。如果选择一个大型程序作为示例，必然会占用很大的篇幅对示例本身进行描述，随之而来的重构过程也会非常复杂，这种示例对读者并不友好（我尝试写了一个稍微复杂的示例，至少需要 100 页的篇幅）。但如果选择一个简单的小程序，又怕读者感受不到重构的意义所在。

和其他的作者一样，之所以会陷入这两难的境地，是因为我们的初衷是一致的，都想为读者提供真实的程序案例。可碍于篇幅问题以及真实的程序案例太过庞大，我只能选择在一个恰当的小示例中进行重构。坦白地说，小示例不值得大费周章地重构，但是它能够让我们理解并掌握重构的技巧。当我们面对真实的项目时，我们所掌握的重构技能就能发挥它重大的作用。所以，希望读者在学习本章示例的过程中，把所学到的重构技巧应用到真实的项目之中。

1.1 示例介绍

话剧表演在国内的流行程度不如西方国家，部分读者可能会对这个示例有所抵触。为了帮助读者更好地理解本章示例，我对英文原书的、本小节的讲述结构，进行了调整，以便读者更快地理解本章示例。

在本书的第一版中，我们以“影片租赁店”为示例背景，为顾客打印租赁票据。可能很多读者并没有接触过“影片租赁店”，为了避免读者在理解示例上花费过多的精力，我在本书中更新了示例，为大家提供了一个更加贴合现实生活的示例。

本章示例以“话剧表演团”为背景，表演类型及演出剧目如下：

- **喜剧类（Comedy）**：《皆大欢喜》（*As You Like It*）。
- **悲剧类（Tragedy）**：《哈姆雷特》（*Hamlet*），《奥赛罗》（*Othello*）。

该剧团的演出剧目数据，以 JSON 的形式存储在 plays.json 文件中，如下所示：

```
{
  "hamlet": {"name": "Hamlet", "type": "tragedy"},
  "as-like": {"name": "As You Like It", "type": "comedy"},
  "othello": {"name": "Othello", "type": "tragedy"}
}
{...}
   function playFor(aPerformance) {...}
   function amountFor(aPerformance) {...}
```

客户预约演出时，会提供预约的“剧目名称”（playID）和“观看人数”（Audience），客户预约单如下所示，我们将其以 JSON 的形式存储在 invoice.json 文件中：

```
[
  {
    "customer": "BigCo",
    "performances": [
      {
        "playID": "hamlet",
        "audience": 55
      },
      {
        "playID": "as-like",
        "audience": 35
      },
      {
        "playID": "othello",
        "audience": 40
      }
    ]
  }
]
```

我们所做的示例，就是根据客户的预约单，计算出“客户需要支付的演出费用”以及“剧团为客户提供的代金券”，此代金券可以在下次预约购票时使用，以提升客户的黏性。费用及代金券计算规则如下：

- 悲剧类和喜剧类的出场费用不同；
- 根据观众人数的不同，需要额外支付表演费用；
- 根据本次观看表演的人数，为客户发放代金券，可以在下次预约购票时使用。

我们用如下代码，进行费用计算和代金券计算：

```
function statement(invoice, plays) {
   let totalAmount = 0;
   let volumeCredits = 0;
   let result = `Statement for ${invoice.customer}\n`;
   const format = new Intl.NumberFormat("en-US", {
         style: "currency",
         currency: "USD",
         minimumFractionDigits: 2
      })
      .format;
   for (let perf of invoice.performances) {
      const play = plays[perf.playID];
      let thisAmount = 0;
      switch (play.type) {
         case "tragedy":
            thisAmount = 40000;
            if (perf.audience > 30) {
               thisAmount += 1000 * (perf.audience - 30);
            }
            break;
         case "comedy":
            thisAmount = 30000;
            if (perf.audience > 20) {
               thisAmount += 10000 + 500 * (perf.audience - 20);
            }
            thisAmount += 300 * perf.audience;
            break;
         default:
            throw new Error(`unknown type: ${play.type}`);
      }
      // 增加代金券金额
      volumeCredits += Math.max(perf.audience - 30, 0);
      // 每有十个观看喜剧表演的观众，增加额外代金券金额
```

```
34        if ("comedy" === play.type) volumeCredits +=
   Math.floor(perf.audience / 5);
35        // 打印详单
36        result += ` ${play.name}: ${format(thisAmount/100)}
   (${perf.audience} seats)\n`;
37        totalAmount += thisAmount;
38      }
39      result += `Amount owed is ${format(totalAmount/100)}\n`;
40      result += `You earned ${volumeCredits} credits\n`;
41      return result;
```

用 plays.json 文件和 invoice.json 文件作为入参，运行以上代码，会得到以下结果：

```
1 Statement for BigCo
2   Hamlet: $650.00 (55 seats)
3   As You Like It: $580.00 (35 seats)
4   Othello: $500.00 (40 seats)
5 Amount owed is $1,730.00
6 You earned 47 credits
```

1.2 对示例代码的评价

对于以上示例代码的设计，你觉得如何？给我的第一感觉是“代码结构模糊”，所有的代码逻辑堆叠在了一起。部分读者也可能认为这段代码的设计尚可，毕竟这是一个非常简单的功能，将代码写在一个方法里更直接。但是，作为开发行业的从业者，我们实际工作中所面对的程序规模会十分庞大，如果把所有的代码逻辑放到一个函数中……那我们距离“毕业”就不远了。

当然了，单从功能角度来说，这个程序是能正常运行的，毕竟代码编译器仅是将我们的代码翻译成机器可识别的语言，而不会在乎代码的结构是否清晰。笔者上文所说的“代码结构模糊”，仅是从代码的可读性、扩展性等方面进行的评价，实际工作中，当企业项目发生需求变更，需要修改代码逻辑时，就需要程序员出马了。如果在程序的设计初期，没有考虑程序的扩展性和可读性，那么后续修改起来就会非常麻烦。对于可读性差的代码，我们很难快速地定位到需要修改的点，而且，很可能会引入未知的 Bug。

所以说，当我们需要修改一个几百行甚至更大的程序时，如果这个程序有清晰的结构和良好的方法封装，就能够帮助我们快速地理解程序，进而快速地定位到需要修改的点。相反，如果“代码结构模糊”，会增加修改的复杂性和风险。因此，我们需要将现有的代码进行重构，使其结构清晰，然后进行代码的修改就会变得非常简单。

> 如果读者在实际工作中需要根据新需求对代码进行修改，而之前的代码结构设计又十分差劲，该怎么办？笔者建议先将程序进行有效的重构，然后根据业务需求进行修改。

可能部分读者认为我小题大做了，仍然认为上面代码的设计还是很不错的。之所以有这样的想法是因为“你还没有亲身体验过，基于新需求的代码修改工作”。一旦新的需求接踵而至，并且指派你负责相关的修改工作，你就会深切地体会到代码设计的重要性。接下来，模拟两个新的需求，让我们身临其境地思考一下。

第一个新需求：我们的用户希望新增一项功能——以 HTML 的格式打印订单详情。如果这个工作安排给你，你会怎么做？相信大部分人会直接复制一份代码，然后将数据打印的那部分代码修改为 HTML 格式的打印，这样我们的系统就能同时支

持两种格式输出（简易的“字符串输出 +HTML 输出”）。复制一份代码并不难，但是如果“费用计算的逻辑发生了变化”怎么办？那我们需要修改两处代码，来保证老方法和新复制的方法保持一致的费用计算逻辑。如果将来需要支持 PDF 格式输出和 Excel 格式输出，又当如何呢？重复的代码逻辑，会造成潜在的威胁。

第二个新需求：剧团会在未来一段时间尝试不同的表演类型，不局限于悲剧和喜剧类型。而针对不同的表演类型，费用计算逻辑也需要做出相应的改变。苍天何曾放过我，只怪我是程序员，该来的总会来……为了应对不同的表演类型及不同的计费逻辑变化，对 statement 函数的重构已经不可避免。如果我们坚持复制粘贴，将费用计算的逻辑在多处进行修改，随着业务逻辑越来越复杂，我们的代码维护成本也会越来越大，出错的可能也会越来越多。

笔者此处想特别说明：“重构，需要依照需求而定，因地制宜。”如果你的代码能够正常运行，并且将来很长一段时间不会修改，那么完全可以不进行重构，或者根据自己的兴趣进行重构。相反，如果你的代码随时会面临诸多的修改，那么笔者强烈建议对代码进行优化和重构。

在内容中，会通过不同的重构手法，分离费用计算部分以及订单详情打印部分的逻辑。如此一来，我们便可以很轻松地添加 HTML 格式打印的新功能，同时，还能够针对不同的表演类型，单独地维护费用计算逻辑。

1.3 重构的前提：单元测试用例

代码的重构，必然会对现有的代码结构进行调整，而在调整的过程中，可能会引入未知的 Bug。因此，重构的前提就是“拥有一组可靠的、全面的单元测试用例”。单元测试用例是必不可少的，程序越复杂，重构过程引入 Bug 的可能性就越大。在数字时代，程序本身，是十分脆弱的。

因此，我们需要给 statement 函数准备一组可靠的单元测试用例。测试用例的准备步骤如下：

（1）手动创建几张新的演出预约单（invoice），自定义预约单内容，尽量使预约单的内容覆盖所有的场景，以确保单元测试的全面性。

（2）将手动创建的演出预约单作为 statement 函数的入参，获取 statement 函数的返回值。

（3）将 statement 函数的返回值与期望的值进行 Assert 校验。

运行这些单元测试只需要几秒，借助单元测试的框架，在开发环境中输入一行命令就可以触发这组单元测试，我们会在后续的重构过程中多次运行这组测试，以确保重构过程的正确性。借助测试框架，可以提升我们的开发速度，可以清晰地了解到单元测试的运行结果：“如果结果为绿色，表示单元测试通过；如果结果为红色，会为我们显示具体出错位置的行号及内容。”避免手动进行结果的对比。

在重构时，我们需要依赖测试，测试可以为我们避免 Bug，对程序的健壮性起到了至关重要的作用。尽管编写单元测试需要花费很多的时间，但却为我们保障了程序的健壮性，节省了重构过程中的代码调试时间，构建测试体系对于重构而言，实在太重要了。因此，我们将在第 4 章中，详细地为大家介绍测试体系的构建。

本小节主要介绍单元测试的重要性，可总结为一句话：“拥有一组可靠的、全面的单元测试用例是重构的前提。”如果读者对这句话没有异议，可以选择直接跳过本节的内容。

重构前，请务必检查单元测试用例的完整性及正确性。

1.4 分解statement函数

对于代码量比较大的函数，我首先会进行快速的浏览，根据不同的代码逻辑（行为），进行初次的模块划分。首先引起我关注的是 statement 函数中的 switch 语句。

```
function statement(invoice, plays) {
   let totalAmount = 0;
   let volumeCredits = 0;
   let result = `Statement for ${invoice.customer}\n`;
   const format = new Intl.NumberFormat("en-US", {
         style: "currency",
         currency: "USD",
         minimumFractionDigits: 2
      })
      .format;
   for (let perf of invoice.performances) {
      const play = plays[perf.playID];
      let thisAmount = 0;

      switch (play.type) {
         case "tragedy":
            thisAmount = 40000;
            if (perf.audience > 30) {
               thisAmount += 1000 * (perf.audience - 30);
            }
            break;
         case "comedy":
            thisAmount = 30000;
            if (perf.audience > 20) {
               thisAmount += 10000 + 500 * (perf.audience - 20);
            }
            thisAmount += 300 * perf.audience;
            break;
         default:
            throw new Error(`unknown type: ${play.type}`);
      }

      // 增加代金券金额
      volumeCredits += Math.max(perf.audience - 30, 0);
      // 每有十个观看喜剧表演的观众，增加额外代金券金额
      if ("comedy" === play.type) volumeCredits +=
Math.floor(perf.audience / 5);
      // 打印详单
      result += ` ${play.name}: ${format(thisAmount/100)}
(${perf.audience} seats)\n`;
      totalAmount += thisAmount;
   }
   result += `Amount owed is ${format(totalAmount/100)}\n`;
   result += `You earned ${volumeCredits} credits\n`;
   return result;
}
```

基于对代码的第一印象，不难看出，这段 switch 代码是演出费用的计算逻辑。正如沃德·坎宁安（Ward Cunningham）所说的：“第一印象的直觉在我的脑海中转瞬即逝。我需要抓住这个瞬间，凭借这份直觉，书写优雅的代码。代码能够让转瞬即逝的直觉成为永恒，每当我再次回看代码时，便能重温那份直觉。”

都是文化人，不让沃德·坎宁安露个脸又觉得对不住本书的作者，硬着头皮翻译了这段“诗句”。

我们需要对这段 switch 代码进行封装，并且根据代码的功能，为封装后的新方法进行合理的命名（增加代码的可读性），此处我们将该方法命名为 amountFor(performance)。在进行代码封装时，为了尽可能地避免犯错，我通常会把整个流程记录下来，并且将所有的代码封装操作统一记录到“*提炼函数法*（参见 6.1 节）”的分类标签下，以便后续使用。

对于 switch 代码段的封装，我们首先需要检查这段代码所涉及的“变量的作用域”。在这段代码中，一共有 3 个变量：pref、play 和 thisAmount。对于 pref 和 play 变量，我们可以通过参数的形式传递给 amountFor 方法；但 thisAmount 变量则

不同，它会随着不同演出类型的变化而变化，因此，我们需要将 thisAmount 作为 amountFor 方法的返回值来适配这种变化，同时需要将 thisAmount 的初始化代码也放到 amountFor 函数中。封装后的 amountFor 方法如下所示（请读者注意以下代码中的“function statement...”，我们使用这种方式表示 amounFor 函数在 statement 函数的作用域中）：

```
function statement...
    function amountFor(perf, play) {
       let thisAmount = 0;
       switch (play.type) {
          case "tragedy":
             thisAmount = 40000;
             if (perf.audience > 30) {
                thisAmount += 1000 * (perf.audience - 30);
             }
             break;
          case "comedy":
             thisAmount = 30000;
             if (perf.audience > 20) {
                thisAmount += 10000 + 500 * (perf.audience - 20);
             }
             thisAmount += 300 * perf.audience;
             break;
          default:
             throw new Error(`unknown type: ${play.type}`);
       }
       return thisAmount;
    }
```

然后，我们在 statement 函数中，将 switch 代码块替换为 amountFor 函数的调用，代码如下所示：

```
//顶层作用域
    function statement(invoice, plays) {
       let totalAmount = 0;
       let volumeCredits = 0;
       let result = `Statement for ${invoice.customer}\n`;
       const format = new Intl.NumberFormat("en-US", {
             style: "currency",
             currency: "USD",
             minimumFractionDigits: 2
          })
          .format;
       for (let perf of invoice.performances) {
          const play = plays[perf.playID];
          let thisAmount = amountFor(perf, play);
          // 增加代金券金额
          volumeCredits += Math.max(perf.audience - 30, 0);
          // 每有十个观看喜剧表演的观众，增加额外代金券金额
          if ("comedy" === play.type) volumeCredits +=
Math.floor(perf.audience / 5);
          // 打印详单
          result += ` ${play.name}: ${format(thisAmount/100)}
(${perf.audience} seats)\n`;
          totalAmount += thisAmount;
       }
       result += `Amount owed is ${format(totalAmount/100)}\n`;
       result += `You earned ${volumeCredits} credits\n`;
       return result;

```

此时，我们需要立即执行单元测试，确保我们的重构没有引发未知的 Bug。

重构技术就是步步为营，多阶段修改并且伴随测试。如果重构过程引发 Bug，能够快速定位问题、修复问题。

可能部分读者亲身经历过“代码回滚到最近一个可运行版本”。回滚的时候我们肯定会想：“这得回滚多少代码？”如果回滚的代码量过大，多数人会后悔没有进行阶段性测试和提交。这也是作者建议大家“小步重构，步步为营”的原因所在。

虽然，对 amountFor 方法的封装是非常简单的改动，但养成重构后立即运行单元测试的习惯，非常重要。在重构的过程中，我们需要步步为营，每完成一小部分代码的重构，都需要进行单元测试的验证，如果发现了问题，便可以快速地定位到异常所在。相反，如果经过大量的重构改动后一次性地进行单元测试验证，一旦发生问题，将会耗费大量的 Bug 定位及调试时间。小步重构，步步为营，非常重要。

当然，JavaScript 语言是支持“内嵌函数”的，根据这个特性，我们可以直接将 amountFor 函数提炼成 statement 函数的一个“内嵌函数”。对内嵌函数而言，我们就不需要将 perf 和 play 参数传递给 amountFor 函数了，amountFor 函数可以直接使用 perf 和 play 参数。提炼为“内嵌函数”不是重构必需的步骤，读者可以自行决定。

当所有的测试用例都通过后，我们需要将这部分修改提交到代码版本控制系统。可以使用如 git 或 mercurial 作为代码的版本控制系统，进行代码的版本管理。还是那句话——步步为营，在每次成功重构后都需要进行代码提交，如果以后出现问题，可以很容易地将代码回滚到之前的版本。此处笔者建议大家，在每次提交代码时，尽量为本次提交（commit）的代码改动添加简要的说明。

提炼函数法（参见 6.1 节）是一种常见的重构方式，一些开发工具也支持自动重构。比如，大部分支持 Java 语言的 IDE，都能够通过快捷键将选中的代码自动进行函数的封装。在我写本书的时候，JavaScript 的相关开发工具还没有这种功能，所以我必须手动进行函数的提炼。当然，手动提炼也并不复杂，只不过在提炼的过程中，需要小心处理那些局部变量。

函数提炼完成后，需要再次对函数进行检查，看看能否对该函数做进一步的优化。针对 amountFor 函数，我要做的第一件事就是重命名一些变量，使它们更清晰，例如将 thisAmount 更改为 result。

```
function statement…
    function amountFor(perf, play) {
        let result = 0;
        switch (play.type) {
            case "tragedy":
                result = 40000;
                if (perf.audience > 30) {
                    result += 1000 * (perf.audience - 30);
                }
                break;
            case "comedy":
                result = 30000;
                if (perf.audience > 20) {
                    result += 10000 + 500 * (perf.audience - 20);
                }
                result += 300 * perf.audience;
                break;
            default:
                throw new Error(`unknown type: ${play.type}`);
        }
        return result;
    }
```

将程序的返回值命名为“result”是我个人的编码习惯，只要看到 result，就知道它代表了整个方法的返回值，读者可以根据自己的编码习惯进行变量的命名。修

改完成后，我会再次进行测试并提交代码。

接下来，我会重命名函数的入参：“将 perf 修改为 aPerformance。”之所以做这样的修改是因为当我们使用动态类型语言时（如 JavaScript），描述变量的类型对变量的可读性帮助很大。一般我会使用不定冠词 (此处我用的是 a) 修饰它，看到 aPerformance 后，我就知道，这里指的是“一个演出”。这个习惯是从 Kent Beck 那里学习的，我一直受用至今。修改后的代码如下：

此处作者将自己的命名习惯进行了分享，大家可以参考借鉴。“命名”是重构过程中不可或缺的，读者可以根据企业项目风格或自身偏好打造一套属于自己的命名方式。

```
function statement…
    function amountFor(aPerformance, play) {
        let result = 0;
        switch (play.type) {
            case "tragedy":
                result = 40000;
                if (aPerformance.audience > 30) {
                    result += 1000 * (aPerformance.audience - 30);
                }
                break;
            case "comedy":
                result = 30000;
                if (aPerformance.audience > 20) {
                    result += 10000 + 500 * (aPerformance.audience -
20);
                }
                result += 300 * aPerformance.audience;
                break;
            default:
                throw new Error(`unknown type: ${play.type}`);
        }
        return result;
    }
```

以上改动，真的值得吗？答案是肯定的。好的代码能够清晰地表达代码逻辑，而变量的命名就是其中重要的一环。任何修改，只要能够提升代码的可读性，我们都需要去做。而且，目前几乎所有的开发工具都支持变量名称的一键修改，函数名称的一键修改，这些工具极大地减少了我们的工作量。

优秀的程序员，能写出易于其他人理解的代码；普通的程序员，只能写出计算机可以理解的代码。

接下来，我们继续对 play 变量进行优化。

1. 移除 play 变量

我们可以看到，函数 amountFor 有两个参数。参数 aPerformance 是从外层循环变量中传入的，它的值会随着循环而改变；但 play 变量不同，它依赖 aPerformance 变量，我们可以通过 aPerformance 变量得到 play 变量的值。所以，无须传入 play 参数，我们可以在 amountFor 函数中，通过 aPerformance 变量得到 play 变量的值。

play 变量是一个局部临时变量，如果我们创建很多这样的变量，会使提炼函数变得复杂。因此，当重构一个函数时，强烈建议大家移除诸如 play 这样的变量，此处，我们使用另一种常用的重构方式——*函数替代临时变量法*（详见 7.4 节）。

首先，我们将 play 变量封装到函数 playFor 中。代码如下：

```
function statement…
function playFor(aPerformance) {
    return plays[aPerformance.playID];
}
```

其次，我们将 statement 函数中有关 play 变量的部分，替换为 playFor 函数的调

用。替换完成后，我们进行阶段性的编译、测试、代码提交，确保代码的正确性。代码如下：

```
//顶层作用域...
function statement(invoice, plays) {
   let totalAmount = 0;
   let volumeCredits = 0;
   let result = `Statement for ${invoice.customer}\n`;
   const format = new Intl.NumberFormat("en-US", {
         style: "currency",
         currency: "USD",
         minimumFractionDigits: 2
      })
      .format;
   for (let perf of invoice.performances) {
      const play = playFor(perf);
      let thisAmount = amountFor(perf, play);
      // 增加代金券金额
      volumeCredits += Math.max(perf.audience - 30, 0);
      // 每有十个观看喜剧表演的观众，增加额外代金券金额
      if ("comedy" === play.type) volumeCredits +=
Math.floor(perf.audience / 5);
      // 打印详单
      result += ` ${play.name}: ${format(thisAmount/100)}
(${perf.audience} seats)\n`;
      totalAmount += thisAmount;
   }
   result += `Amount owed is ${format(totalAmount/100)}\n`;
   result += `You earned ${volumeCredits} credits\n`;
   return result;
```

再次，我们使用内联变量法（详见 6.4 节）将 play 变量进行代码内联，内联完成后，我们再次进行阶段性的编译、测试、代码提交，确保代码的正确性。代码如下：

```
//顶层作用域...
function statement(invoice, plays) {
   let totalAmount = 0;
   let volumeCredits = 0;
   let result = `Statement for ${invoice.customer}\n`;
   const format = new Intl.NumberFormat("en-US", {
         style: "currency",
         currency: "USD",
         minimumFractionDigits: 2
      })
      .format;
   for (let perf of invoice.performances) {
      const play = playFor(perf);
      let thisAmount = amountFor(perf, playFor(perf));
      // 增加代金券金额
      volumeCredits += Math.max(perf.audience - 30, 0);
      // 每有十个观看喜剧表演的观众，增加额外代金券金额
      if ("comedy" === playFor(perf)
         .type) volumeCredits += Math.floor(perf.audience / 5);
      // 打印详单
      result += ` ${playFor(perf).name}: ${format(thisAmount/100)}
(${perf.audience} seats)\n`;
      totalAmount += thisAmount;
   }
   result += `Amount owed is ${format(totalAmount/100)}\n`;
   result += `You earned ${volumeCredits} credits\n`;
   return result;
```

最后，我们通过修改函数声明法（详见 6.5 节），将 amountFor 函数中的 play 变量都修改为对 playFor 函数的调用，最后移除 play 变量。详细步骤如下。

（1）将 amountFor 函数中所有使用 play 变量的地方，替换为对 playFor 函数的调用，代码如下：

```
function statement…
function amountFor(aPerformance, play) {
   let result = 0;
   switch (playFor(aPerformance).type) {
      case "tragedy":
         result = 40000;
         if (aPerformance.audience > 30) {
            result += 1000 * (aPerformance.audience - 30);
         }
         break;
      case "comedy":
         result = 30000;
         if (aPerformance.audience > 20) {
            result += 10000 + 500 * (aPerformance.audience - 20);
         }
         result += 300 * aPerformance.audience;
         break;
      default:
         throw new Error(`unknown type:
${playFor(aPerformance).type}`);
   }
   return result;
}
```

（2）移除 play 参数，并进行阶段性的编译、测试、代码提交，确保代码的正确性。代码如下：

```
//顶层作用域...
function statement(invoice, plays) {
   let totalAmount = 0;
   let volumeCredits = 0;
   let result = `Statement for ${invoice.customer}\n`;
   const format = new Intl.NumberFormat("en-US", {
         style: "currency",
         currency: "USD",
         minimumFractionDigits: 2
      })
      .format;
   for (let perf of invoice.performances) {
      let thisAmount = amountFor(perf, playFor(perf));
      // 增加代金券金额
      volumeCredits += Math.max(perf.audience - 30, 0);
      // 每有十个观看喜剧表演的观众，增加额外代金券金额
      if ("comedy" === playFor(perf).type) volumeCredits +=
Math.floor(perf.audience / 5);
      // 打印详单
      result += ` ${playFor(perf).name}: ${format(thisAmount/100)}
(${perf.audience} seats)\n`;
      totalAmount += thisAmount;
   }
   result += `Amount owed is ${format(totalAmount/100)}\n`;
   result += `You earned ${volumeCredits} credits\n`;
   return result;

function statement…
function amountFor(aPerformance, play) {
   let result = 0;
```

```
  switch (playFor(aPerformance).type) {
    case "tragedy":
      result = 40000;
      if (aPerformance.audience > 30) {
        result += 1000 * (aPerformance.audience - 30);
      }
      break;
    case "comedy":
      result = 30000;
      if (aPerformance.audience > 20) {
        result += 10000 + 500 * (aPerformance.audience - 20);
      }
      result += 300 * aPerformance.audience;
      break;
    default:
      throw new Error(`unknown type:
${playFor(aPerformance).type}`);
  }
  return result;
}
```

重构必然会影响代码的组织结构，因此在重构和性能之间我们需要进行权衡。

部分读者可能对这次重构过程存在疑问：“重构前的代码，为了获取 play 变量的值，我们只需要获取一次；而重构后却需要获取 3 次，性能会有问题吧？”请读者不要着急，我们会在后文探讨“重构与性能的平衡”。对于当前的重构示例，我认为不会有太大的性能影响。退一万步说，即便真的有影响，基于优秀代码结构的性能调优也会更加简单。

此处，再次强调一下关于移除局部变量的好处：减少局部变量，能够使函数的封装变得更加容易。做任何函数封装前，我会习惯性地把局部变量移除掉。

现在，我们看一下用 statement 函数对 amountFor 函数进行调用的代码。我们将 amountFor 函数的返回值赋值给了局部变量 thisAmount，之后不会有其他的修改。因此，我们可以使用*内联变量法*（详见 6.4 节）移除 thisAmount 变量。代码如下：

```
//顶层作用域...
function statement(invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Statement for ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("en-US", {
        style: "currency",
        currency: "USD",
        minimumFractionDigits: 2
    })
    .format;
  for (let perf of invoice.performances) {
    // 增加代金券金额
    volumeCredits += Math.max(perf.audience - 30, 0);
    // 每有十个观看喜剧表演的观众，增加额外代金券金额
    if ("comedy" === playFor(perf).type) volumeCredits +=
Math.floor(perf.audience / 5);
    // 打印详单
    result += ` ${playFor(perf).name}: ${format(amountFor(perf)/100)}
(${perf.audience} seats)\n`;
    totalAmount += amountFor(perf);
  }
  result += `Amount owed is ${format(totalAmount/100)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;
```

2. 封装代金券计算逻辑

目前，statement 函数的代码逻辑如下所示：

```
//顶层作用域...
function statement(invoice, plays) {
   let totalAmount = 0;
   let volumeCredits = 0;
   let result = `Statement for ${invoice.customer}\n`;
   const format = new Intl.NumberFormat("en-US", {
         style: "currency",
         currency: "USD",
         minimumFractionDigits: 2
      })
      .format;
   for (let perf of invoice.performances) {
      // 增加代金券金额
      volumeCredits += Math.max(perf.audience - 30, 0);
      // 每有十个观看喜剧表演的观众，增加额外代金券金额
      if ("comedy" === playFor(perf).type) volumeCredits +=
  Math.floor(perf.audience / 5);
      // 打印详单
      result += ` ${playFor(perf).name}: ${format(amountFor(perf)/100)}
  (${perf.audience} seats)\n`;
      totalAmount += amountFor(perf);
   }
   result += `Amount owed is ${format(totalAmount/100)}\n`;
   result += `You earned ${volumeCredits} credits\n`;
   return result;
```

接下来，我们需要对“代金券的计算逻辑”进行函数的提炼、封装。对于这部分代码逻辑，我们需要处理以下两个局部变量。

（1）perf 变量。我们可以将 perf 作为函数的入参。

（2）volumeCredits 变量。我们可以看到，每次循环遍历都会对 volumeCredits 的值进行累加更新。对于这样的变量，最简单的方法就是将整块逻辑封装到新的函数中，然后在新封装的函数中，返回 volumeCredits 的值。代码如下：

```
function statement…
function volumeCreditsFor(perf) {
   let volumeCredits = 0;
   volumeCredits += Math.max(perf.audience - 30, 0);
   if ("comedy" === playFor(perf)
      .type) volumeCredits += Math.floor(perf.audience / 5);
   return volumeCredits;
}
```

```
//顶层作用域...
function statement(invoice, plays) {
   let totalAmount = 0;
   let volumeCredits = 0;
   let result = `Statement for ${invoice.customer}\n`;
   const format = new Intl.NumberFormat("en-US", {
         style: "currency",
         currency: "USD",
         minimumFractionDigits: 2
      })
      .format;
   for (let perf of invoice.performances) {
      volumeCredits += volumeCreditsFor(perf);
      // 打印详单
      result += ` ${playFor(perf).name}: ${format(amountFor(perf)/100)}
  (${perf.audience} seats)\n`;
```

```
16        totalAmount += amountFor(perf);
17    }
18    result += `Amount owed is ${format(totalAmount/100)}\n`;
19    result += `You earned ${volumeCredits} credits\n`;
20    return result;
```

此处，请读者思考这样一个问题：“为什么这次的函数封装，不需要考虑 play 变量呢？”相信所有读者都感受到了移除 play 变量的好处：移除一个局部变量，使得提炼代金券计算逻辑的工作更加简单。

紧接着，我们对新函数中的变量进行重命名，代码如下：

```
1 function statement…
2 function volumeCreditsFor(aPerformance) {
3    let result = 0;
4    result += Math.max(aPerformance.audience - 30, 0);
5    if ("comedy" === playFor(aPerformance).type)
6 result += Math.floor(aPerformance.audience / 5);
7    return result;
8 }
```

这里我只展示了重命名后的最终代码，在实际操作中，每完成一个变量的重命名，我都会对代码执行编译、测试和提交，确保重构的正确性。

3. 移除 format 变量

我们看一下当前阶段的 statement 函数：

```
1 //顶层作用域...
2 function statement(invoice, plays) {
3    let totalAmount = 0;
4    let volumeCredits = 0;
5    let result = `Statement for ${invoice.customer}\n`;
6    const format = new Intl.NumberFormat("en-US", {
7          style: "currency",
8          currency: "USD",
9          minimumFractionDigits: 2
10       }).format;
11   for (let perf of invoice.performances) {
12      volumeCredits += volumeCreditsFor(perf);
13      // 打印详单
14      result += ` ${playFor(perf).name}: ${format(amountFor(perf)/100)}
   (${perf.audience} seats)\n`;
15      totalAmount += amountFor(perf);
16   }
17   result += `Amount owed is ${format(totalAmount/100)}\n`;
18   result += `You earned ${volumeCredits} credits\n`;
19   return result;
```

正如前文所讲，临时变量仅在它的局部代码块中起作用，并且，临时变量会在重构过程中带来麻烦，因此我们需要对临时变量进行移除。观察当前的 statement 函数能够发现，移除 format 变量是最简单的，因为 format 变量仅是一个常规的赋值操作，我更倾向于将 format 变量的赋值操作封装到新的函数中，代码如下：

```
1 function statement…
2 function format(aNumber) {
3    return new Intl.NumberFormat("en-US", {
4          style: "currency",
5          currency: "USD",
6          minimumFractionDigits: 2
7       }).format(aNumber);
8 }
```

```
//顶层作用域...
function statement(invoice, plays) {
   let totalAmount = 0;
   let volumeCredits = 0;
   let result = `Statement for ${invoice.customer}\n`;
   for (let perf of invoice.performances) {
      volumeCredits += volumeCreditsFor(perf);
      // 打印详单
      result += ` ${playFor(perf).name}: ${format(amountFor(perf)/100)}
  (${perf.audience} seats)\n`;
      totalAmount += amountFor(perf);
   }
   result += `Amount owed is ${format(totalAmount/100)}\n`;
   result += `You earned ${volumeCredits} credits\n`;
   return result;
```

虽然将“变量的常规的赋值操作封装为函数”也是一种重构手法，但我并没有在本书中对这个手法进行单独章节的讲解。此外，还有很多其他重构手法，我都觉得不够重要，也没有收录在本书中。此处对 format 变量赋值的重构，手法简单但不常用，不值得在本书中占用太多的篇幅。

对于新封装的 format 函数，我们无法从方法名中清晰地了解到函数的作用。如果改名为“formatAsUSD”，确实能够清晰地表达函数的意义，但考虑到我们是在拼接 result 字符串的过程中使用的这个函数，如果函数名字太长，会导致 result 字符串拼接的代码变长，不方便阅读。最后，我选择 usd 作为该函数的名字，之所以选择 usd 作为函数名，是因为这个方法想要表达的关键点是“转化成什么格式”，usd 就是我们的目标转化格式。此处的重构，我们应用修改函数声明法（详见 6.5 节），代码如下：

“命名”因人而异，因“code reviewer”而异。函数名字简短精悍很好，全面明确也很好。请读者根据实际情况进行适当的命名，好的命名十分重要。

```
//顶层作用域...
function statement(invoice, plays) {
   let totalAmount = 0;
   let volumeCredits = 0;
   let result = `Statement for ${invoice.customer}\n`;
   for (let perf of invoice.performances) {
      volumeCredits += volumeCreditsFor(perf);
      // 打印详单
      result += ` ${playFor(perf).name}: ${usd(amountFor(perf))}
  (${perf.audience} seats)\n`;
      totalAmount += amountFor(perf);
   }
   result += `Amount owed is ${usd(totalAmount)}\n`;
   result += `You earned ${volumeCredits} credits\n`;
   return result;
```

```
function statement…
function usd(aNumber) {
   return new Intl.NumberFormat("en-US", {
         style: "currency",
         currency: "USD",
         minimumFractionDigits: 2
      }).format(aNumber / 100);
}
```

好的命名非常重要。将一个大函数重构分解为较小的函数，好的命名能够增加重构的价值。什么是好的命名？只看名字就能够读懂代码就是好的命名。我们很难第一次就找到好的命名，所以需要花时间通读代码，反复揣摩，一旦有最佳的命名，就立刻对代码进行重命名。

在重命名 format 函数时，我顺带将重复的 aNumber / 100 也移到了函数中，并将所有对美元数据的转化代码都统一封装到了该函数中。

4. 移除 volumeCredits 变量

移除 volumeCredits 会费一些周折，因为它是循环迭代过程中的累加值。我们分为以下 5 步移除 volumeCredits 变量。

（1）通过*拆分循环法*（详见 8.7 节），分离出 volumeCredits 的累加逻辑，代码如下：

```
//顶层作用域....
function statement(invoice, plays) {
   let totalAmount = 0;
   let volumeCredits = 0;
   let result = `Statement for ${invoice.customer}\n`;
   for (let perf of invoice.performances) {
      // 打印详单
      result += ` ${playFor(perf).name}: ${usd(amountFor(perf))}
  (${perf.audience} seats)\n`;
      totalAmount += amountFor(perf);
   }
   for (let perf of invoice.performances) {
      volumeCredits += volumeCreditsFor(perf);
   }
   result += `Amount owed is ${usd(totalAmount)}\n`;
   result += `You earned ${volumeCredits} credits\n`;
   return result;
```

（2）通过*移动语句法*（详见 8.6 节），把与 volumeCredits 有关的代码都集中到一起，代码如下：

```
//顶层作用域....
function statement(invoice, plays) {
   let totalAmount = 0;
   let result = `Statement for ${invoice.customer}\n`;
   for (let perf of invoice.performances) {
      // 打印详单
      result += ` ${playFor(perf).name}: ${usd(amountFor(perf))}
  (${perf.audience} seats)\n`;
      totalAmount += amountFor(perf);
   }
let volumeCredits = 0;
   for (let perf of invoice.performances) {
      volumeCredits += volumeCreditsFor(perf);
   }
   result += `Amount owed is ${usd(totalAmount)}\n`;
   result += `You earned ${volumeCredits} credits\n`;
   return result;
```

（3）通过*提炼函数法*（详见 6.1 节），将与 volumeCredits 有关的代码封装到新的函数 totalVolumeCredits 中，代码如下：

```
function statement…
function totalVolumeCredits() {
   let volumeCredits = 0;
   for (let perf of invoice.performances) {
      volumeCredits += volumeCreditsFor(perf);
   }
   return volumeCredits;
}
```

（4）通过*函数替代临时变量法*（详见 7.4 节），在 statement 函数中调用

totalVolumeCredits 函数，代码如下：

```
//顶层作用域...
function statement(invoice, plays) {
   let totalAmount = 0;
   let result = `Statement for ${invoice.customer}\n`;
   for (let perf of invoice.performances) {
      // 打印详单
      result += ` ${playFor(perf).name}: ${usd(amountFor(perf))}
  (${perf.audience} seats)\n`;
      totalAmount += amountFor(perf);
   }
   let volumeCredits = totalVolumeCredits();
   result += `Amount owed is ${usd(totalAmount)}\n`;
   result += `You earned ${volumeCredits} credits\n`;
   return result;
```

（5）通过内联变量法（详见 6.4 节），在 statement 函数中内联 totalVolumeCredits 函数。代码如下：

```
//顶层作用域...
function statement(invoice, plays) {
   let totalAmount = 0;
   let result = `Statement for ${invoice.customer}\n`;
   for (let perf of invoice.performances) {
      // 打印详单
      result += ` ${playFor(perf).name}: ${usd(amountFor(perf))}
  (${perf.audience} seats)\n`;
      totalAmount += amountFor(perf);
   }
   result += `Amount owed is ${usd(totalAmount)}\n`;
   result += `You earned ${totalVolumeCredits()} credits\n`;
   return result;
```

至此，相信部分读者会再次提出对于性能问题的疑问，作为程序开发的从业者，我们会本能地警惕 for 循环。重构前的代码仅需要执行一次 for 循环，而现在却需要执行两次 for 循环。我非常理解大家的担忧，但“大多数情况”下，多执行一次循环体对性能造成的影响可以忽略不计，随着编译器技术的发展，现代缓存技术的发展，我们很多对性能问题的主观直觉可能是不准确的。而大部分情况，影响程序性能的可能只是一小部分逻辑代码，修改其他代码逻辑，往往不能达到很好的性能调优效果。

当然了，“大多数情况”不能代表“所有情况”。重构的过程中，确实会存在影响性能的情况，但我会坚持继续重构，暂时不管性能的问题。等到重构完成后，我会拿出时间进行性能调优，拥有一份结构清晰的代码时，再进行代码调优工作就会容易得多。调优过程中，必要的话也可以回滚部分重构代码，但大多数情况下不需要回滚代码。最终，我们会得到一份结构清晰且性能高效的代码。

先重构，获取一份结构清晰且可读性高的代码；然后进行相应的性能测试；如果有性能问题，再进行性能调优。实际工作中，我们还需要根据接口并发量及业务要求对重构和性能进行权衡。

最后总结下，对于如何权衡重构与性能，我的建议是：大多数情况可以忽略性能问题；如果真的引入了性能损耗，先重构，再调优。

另外，在移除 volumeCredits 的过程中，我们划分了五小步，每一步都需要进行一次编译、测试和代码提交。坦白地说，现实中我并不总是小步重构，如此频繁的操作会增加不少工作量。但是，一旦引发了大面积的代码问题，我们的第一反应就是回滚代码，回滚到上一个稳定运行的版本，回滚的代码越少越好，频繁提交的好处就在于此。因此，在进行复杂代码的重构时，强烈建议大家小步重构，步步

为营。

接下来，我们以同样的步骤移除 totalAmount 变量，最终代码如下：

```
function statement…
function totalAmount() {
   let totalAmount = 0;
   for (let perf of invoice.performances) {
      totalAmount += amountFor(perf);
   }
   return totalAmount;
}
```

```
//顶层作用域...
function statement(invoice, plays) {
   let result = `Statement for ${invoice.customer}\n`;
   for (let perf of invoice.performances) {
      result += ` ${playFor(perf).name}: ${usd(amountFor(perf))}
 (${perf.audience} seats)\n`;
   }
   result += `Amount owed is ${usd(totalAmount())}\n`;
   result += `You earned ${totalVolumeCredits()} credits\n`;
   return result;
```

最后，我们对新函数内部的变量进行重命名，保持一致的代码风格，代码如下：

```
function totalAmount() {
   let result = 0;
   for (let perf of invoice.performances) {
      result += amountFor(perf);
   }
   return result;
}

function totalVolumeCredits() {
   let result = 0;
   for (let perf of invoice.performances) {
      result += volumeCreditsFor(perf);
   }
   return result;
}
```

1.5 阶段性展示：大量函数嵌套

现在，我们对代码的全貌进行阶段性查看和总结。

```
function statement(invoice, plays) {
   let result = `Statement for ${invoice.customer}\n`;
   for (let perf of invoice.performances) {
      result += ` ${playFor(perf).name}: ${usd(amountFor(perf))}
 (${perf.audience} seats)\n`;
   }
   result += `Amount owed is ${usd(totalAmount())}\n`;
   result += `You earned ${totalVolumeCredits()} credits\n`;
   return result;

   function totalAmount() {
      let result = 0;
      for (let perf of invoice.performances) {
         result += amountFor(perf);
      }
      return result;
   }

```

```
18    function totalVolumeCredits() {
19       let result = 0;
20       for (let perf of invoice.performances) {
21          result += volumeCreditsFor(perf);
22       }
23       return result;
24    }
25
26    function usd(aNumber) {
27       return new Intl.NumberFormat("en-US", {
28             style: "currency",
29             currency: "USD",
30             minimumFractionDigits: 2
31          })
32          .format(aNumber / 100);
33    }
34
35    function volumeCreditsFor(aPerformance) {
36       let result = 0;
37       result += Math.max(aPerformance.audience - 30, 0);
38       if ("comedy" === playFor(aPerformance)
39          .type) result += Math.floor(aPerformance.audience / 5);
40       return result;
41    }
42
43    function playFor(aPerformance) {
44       return plays[aPerformance.playID];
45    }
46
47    function amountFor(aPerformance) {
48       let result = 0;
49       switch (playFor(aPerformance)
50          .type) {
51          case "tragedy":
52             result = 40000;
53             if (aPerformance.audience > 30) {
54                result += 1000 * (aPerformance.audience - 30);
55             }
56             break;
57          case "comedy":
58             result = 30000;
59             if (aPerformance.audience > 20) {
60                result += 10000 + 500 * (aPerformance.audience - 20);
61             }
62             result += 300 * aPerformance.audience;
63             break;
64          default:
65             throw new Error(`unknown type:
   ${playFor(aPerformance).type}`);
66       }
67       return result;
68    }
69 }
```

现在代码结构清晰了很多。顶层的主函数 statement 只有 7 行代码，专门负责订单详情的打印。其次，费用计算的相关逻辑也进行了单独封装，提高了代码的可读性。

1.6 模块化拆分：计算模块与输出模块

到目前为止，我们已经将部分代码片段进行了合理封装，将复杂的代码拆分成了更小的函数单元，使代码结构变得非常清晰，同时，我们还对函数进行了命名优

还记得我们在 1.2 节讨论的 HTML 格式输出的新需求吗？经过 1.3 节 ~1.5 节的重构，我们将“费用计算”的逻辑进行了封装，作为计算相关的模块进行单独管理。本节，为了新增 HTML 版本输出的新功能，我们需要先将“输出打印逻辑”进行单独的封装，作为输出模块进行单独的管理。

化。一般情况下，这些都是在开始重构时，首先需要完成的工作。现在，我们拥有了清晰的代码结构，那接下来，就可以尝试添加新的功能——以 HTML 格式输出订单详情。顶层的主函数 statement 只有 7 行代码，专门负责订单详情的打印输出，我们只需要为这 7 行代码实现一个 HTML 版本即可。而费用计算的相关逻辑，我们早已完成封装，这些费用计算的函数，可以被当前的文本格式输出版本和即将新增的 HTML 输出版本共用。

实现模块拆分、公共代码复用的方式很多，我最常用的方式是*阶段拆分法*（详见 6.1 1 节）。此处，我们可以将代码拆分为以下两个阶段。

阶段一：创建一个中转数据结构，里边包含打印订单详情所需要的数据，该阶段的代码统一封装为计算模块进行单独管理。

阶段二：创建一个新的函数，专门负责订单详情的打印，通过*提炼函数法*（参见 6.1 节），封装 statement 函数中的 7 行代码。并且，把**阶段一**的中转数据结构作为该函数的入参。该阶段的代码统封装为输出模块进行单独管理。

在应用*阶段拆分法*（详见 6.11 节）之前，我会先对 statement 函数中的 7 行代码应用*提炼函数法*（参见 6.1 节），这部分代码就是专门负责打印输出订单详情的。我们把 statement 函数中的所有代码提取到一个新的顶层函数 renderPlainText 中，并进行编译、测试和代码提交。

```
function statement(invoice, plays) {
   return renderPlainText(invoice, plays);
}

function renderPlainText(invoice, plays) {
   let result = `Statement for ${invoice.customer}\n`;
   for (let perf of invoice.performances) {
      result += ` ${playFor(perf).name}: ${usd(amountFor(perf))} (${perf.audience} seats)\n`;
   }
   result += `Amount owed is ${usd(totalAmount())}\n`;
   result += `You earned ${totalVolumeCredits()} credits\n`;
   return result;

   function totalAmount(){...}
   function totalVolumeCredits() {...}
   function usd(aNumber) {...}
   function volumeCreditsFor(aPerformance) {...}
   function playFor(aPerformance) {...}
   function amountFor(aPerformance) {...}
```

接下来，我们就可以开始阶段一和阶段二的操作了，创建一个中转数据结构，并将该数据结构作为 renderPlainText 的入参。代码如下：

```
function statement(invoice, plays) {
const statementData = {};
   return renderPlainText(statementData, invoice, plays);
}

function renderPlainText(data, invoice, plays) {
   let result = `Statement for ${invoice.customer}\n`;
   for (let perf of invoice.performances) {
      result += ` ${playFor(perf).name}: ${usd(amountFor(perf))} (${perf.audience} seats)\n`;
   }
   result += `Amount owed is ${usd(totalAmount())}\n`;
```

```
  result += `You earned ${totalVolumeCredits()} credits\n`;
  return result;

  function totalAmount(){...}
  function totalVolumeCredits() {...}
  function usd(aNumber) {...}
  function volumeCreditsFor(aPerformance) {...}
  function playFor(aPerformance) {...}
  function amountFor(aPerformance) {...}
```

接着，我们需要处理 renderPlainText 方法中的其他参数（invoice 和 plays 参数）。如果我们能够将 invoice 和 plays 参数都挪到这个中转数据结构中，那么 renderPlainText 方法就只需要关心参数 data 即可。

首先，我们将顾客（customer）字段挪到中转数据结构 data 中（编译、测试并提交代码），代码如下：

想要移除 invoice 变量，我们需要观察下哪里使用了 invoice 变量。发现代码中使用了 invoice.customer 和 invoice.performances。因此，想要移除 invoice 变量，我们就需要将 customer 和 performances 挪到中转数据结构中。

```
function statement(invoice, plays) {
  const statementData = {};
  statementData.customer = invoice.customer;
  return renderPlainText(statementData, invoice, plays);
}

function renderPlainText(data, invoice, plays) {
  let result = `Statement for ${data.customer}\n`;
  for (let perf of invoice.performances) {
    result += ` ${playFor(perf).name}: ${usd(amountFor(perf))} (${perf.audience} seats)\n`;
  }
  result += `Amount owed is ${usd(totalAmount())}\n`;
  result += `You earned ${totalVolumeCredits()} credits\n`;
  return result;
```

其次，我们用同样的方法，将 performances 字段也挪到中转数据结构 data 中，这样我们就可以完全移除 invoice 参数了。代码如下：

```
//顶层作用域...
function statement(invoice, plays) {
  const statementData = {};
  statementData.customer = invoice.customer;
  statementData.performances = invoice.performances;
  return renderPlainText(statementData, invoice, plays);
}

function renderPlainText(data, plays) {
  let result = `Statement for ${data.customer}\n`;
  for (let perf of data.performances) {
    result += ` ${playFor(perf).name}: ${usd(amountFor(perf))} (${perf.audience} seats)\n`;
  }
  result += `Amount owed is ${usd(totalAmount())}\n`;
  result += `You earned ${totalVolumeCredits()} credits\n`;
  return result;
```

```
function renderPlainText...
function totalAmount() {
  let result = 0;
  for (let perf of data.performances) {
    result += amountFor(perf);
  }
  return result;
}

```

```
function totalVolumeCredits() {
   let result = 0;
   for (let perf of data.performances) {
      result += volumeCreditsFor(perf);
   }
   return result;
}
```

再次，我们需要将 plays 参数也挪到中转数据结构中。此处，我们将 plays 中的数据一一对应地填充到 performances 中（编译、测试并提交代码）。

```
function statement(invoice, plays) {
   const statementData = {};
   statementData.customer = invoice.customer;
   statementData.performances =
invoice.performances.map(enrichPerformance);
   return renderPlainText(statementData, plays);

   function enrichPerformance(aPerformance) {
      const result = Object.assign({}, aPerformance);
      return result;
   }
```

enrichPerformance 函数的入参是 aPerformance，我们需要向 aPerformance 中填充数据。是直接向 aPerformance 中添加，还是创建一个 aPerformance 的浅副本然后向该副本中填充数据？选择前者会对入参的原始值造成影响，但是不需要创建新的对象；选择后者不会对入参的原始值造成影响，但会额外在内存创建一个新的对象。两种选择各有利弊，请读者根据实际情况进行选择。

观察 enrichPerformance 方法，我们通过代码 const result = Object.assign({}, aPerformance) 为参数 aPerformance 创建了一个浅副本，请不要着急，我们马上就会往浅副本中添加 palys 中的数据。之所以创建对象的浅副本，是因为不建议大家直接对入参 aPerformance 进行修改，否则，可能会带来一些未知的问题。

const result = Object.assign({}, aPerformance) 这行代码是 JavaScript 中创建浅副本的写法，不熟悉 JavaScript 的读者可能会对此感到陌生。这是 JavaScript 中约定俗称的写法，不需要额外对这行代码进行函数的封装。

最后，我们将 plays 的数据填充到创建的浅副本中。此处，我们使用重构的函数迁移法（详见 8.1 节），代码如下：

```
function statement…
    function enrichPerformance(aPerformance) {
        const result = Object.assign({}, aPerformance);
        result.play = playFor(result);
        return result;
    }

    function playFor(aPerformance) {
        return plays[aPerformance.playID];
    }
```

至此，我们将 plays 中的数据，都挪到中转数据结构中的 performances 中。接下来，我们把 renderPlainText 方法中，所有对 playFor 方法引用的地方，都修改为从中转数据结构中的 performances 里获取（测试、编译并提交代码），代码如下：

```
function renderPlainText…
    let result = `Statement for ${data.customer}\n`;
    for (let perf of data.performances) {
        result += ` ${perf.play.name}: ${usd(amountFor(perf))}
(${perf.audience} seats)\n`;
    }
    result += `Amount owed is ${usd(totalAmount())}\n`;
    result += `You earned ${totalVolumeCredits()} credits\n`;
    return result;
```

```
 9
10      function volumeCreditsFor(aPerformance) {
11          let result = 0;
12          result += Math.max(aPerformance.audience - 30, 0);
13          if ("comedy" === aPerformance.play.type) result +=
    Math.floor(aPerformance.audience / 5);
14          return result;
15      }
16
17      function amountFor(aPerformance) {
18          let result = 0;
19          switch (aPerformance.play.type) {
20              case "tragedy":
21                  result = 40000;
22                  if (aPerformance.audience > 30) {
23                      result += 1000 * (aPerformance.audience - 30);
24                  }
25                  break;
26              case "comedy":
27                  result = 30000;
28                  if (aPerformance.audience > 20) {
29                      result += 10000 + 500 * (aPerformance.audience -
    20);
30                  }
31                  result += 300 * aPerformance.audience;
32                  break;
33              default:
34                  throw new Error(`unknown type:
    ${aPerformance.play.type}`);
35          }
36          return result;
37  }
```

紧接着，我们使用同样方式对 amountFor 函数进行迁移。代码如下：

```
 1  function statement…
 2      function enrichPerformance(aPerformance) {
 3          const result = Object.assign({}, aPerformance);
 4          result.play = playFor(result);
 5          result.amount = amountFor(result);
 6          return result;
 7      }
 8
 9      function amountFor(aPerformance) {
10          ...
11      }
```

```
 1  function renderPlainText…
 2      let result = `Statement for ${data.customer}\n`;
 3      for (let perf of data.performances) {
 4          result += ` ${perf.play.name}: ${usd(perf.amount)}
    (${perf.audience} seats)\n`;
 5      }
 6      result += `Amount owed is ${usd(totalAmount())}\n`;
 7      result += `You earned ${totalVolumeCredits()} credits\n`;
 8      return result;
 9
10      function totalAmount() {
11          let result = 0;
12          for (let perf of data.performances) {
13              result += perf.amount;
14          }
15          return result;
16      }
```

趁热打铁，我们依然使用同样的方式，对 volumeCreditsFor 函数进行迁移。代码如下：

```
function statement…
    function enrichPerformance(aPerformance) {
        const result = Object.assign({}, aPerformance);
        result.play = playFor(result);
        result.amount = amountFor(result);
        result.volumeCredits = volumeCreditsFor(result);
        return result;
    }

    function volumeCreditsFor(aPerformance) {
        ...
    }
```

```
function renderPlainText…
    function totalVolumeCredits() {
        let result = 0;
        for (let perf of data.performances) {
            result += perf.volumeCredits;
        }
        return result;
    }
```

最后，我们将 totalAmount 函数和 totalVolumeCredits 函数迁移到 statement 函数中，并将相关数据存放到中转数据结构中，代码如下：

```
function statement…
    const statementData = {};
    statementData.customer = invoice.customer;
    statementData.performances =
invoice.performances.map(enrichPerformance);
    statementData.totalAmount = totalAmount(statementData);
    statementData.totalVolumeCredits =
totalVolumeCredits(statementData);
    return renderPlainText(statementData, plays);

    function totalAmount(data) {
        ...
    }

    function totalVolumeCredits(data) {
        ...
    }
```

```
function renderPlainText…
    let result = `Statement for ${data.customer}\n`;
    for (let perf of data.performances) {
        result += ` ${perf.play.name}: ${usd(perf.amount)}
(${perf.audience} seats)\n`;
    }
    result += `Amount owed is ${usd(data.totalAmount)}\n`;
    result += `You earned ${data.totalVolumeCredits} credits\n`;
    return result;
```

我们可以看到，totalAmount 函数和 totalVolumeCredits 函数都传入了 statementData 参数，其实无须进行参数传入，因为这两个函数都在 statementData 的作用域范围内。此处读者可以根据自己的喜好进行调整。

经过对以上函数的迁移，请务必进行编译、测试并提交代码，确保重构的正确性。然后，我们应用*管道替代循环法*（详见 8.8 节），继续对代码进行重构。代码如下：

```
function renderPlainText…
    function totalAmount(data) {
        return data.performances
            .reduce((total, p) => total + p.amount, 0);
    }

    function totalVolumeCredits(data) {
        return data.performances
            .reduce((total, p) => total + p.volumeCredits, 0);
    }
```

现在，我可以把**阶段一**（创建一个中转数据结构，里边包含订单详情所需要的数据）的代码提取到一个新的独立函数中。代码如下：

```
//顶层作用域...
function statement(invoice, plays) {
    return renderPlainText(createStatementData(invoice, plays));
}

function createStatementData(invoice, plays) {
    const statementData = {};
    statementData.customer = invoice.customer;
    statementData.performances =
  invoice.performances.map(enrichPerformance);
    statementData.totalAmount = totalAmount(statementData);
    statementData.totalVolumeCredits =
  totalVolumeCredits(statementData);
    return statementData;
}
```

考虑到需要将**阶段一（计算模块）**和**阶段二（输出模块）**的代码逻辑进行彻底分离，我们将**阶段一（计算模块）**的代码统一移动到一个 js 文件中进行单独管理（顺便将返回结果的变量名称修改为 result，代码风格保持一致）。代码如下：

```
statement.js…
   import createStatementData from './createStatementData.js';
```

```
createStatementData.js…
export default function createStatementData(invoice, plays) {
    const result = {};
    result.customer = invoice.customer;
    result.performances = invoice.performances.map(enrichPerformance);
    result.totalAmount = totalAmount(result);
    result.totalVolumeCredits = totalVolumeCredits(result);
    return result;

    function enrichPerformance(aPerformance) {
        ...
    }

    function playFor(aPerformance) {
        ...
    }

    function amountFor(aPerformance) {
        ...
    }

    function volumeCreditsFor(aPerformance) {
        ...
    }

```

计算模块，统一封装到 createStatement.js 中；输出模块，统一封装到 statement.js 中。由于输出模块需要依赖计算模块的数据，因此，在 statement.js 中，需要 import 计算模块中的相关数据。

```
    function totalAmount(data) {
        ...
    }

    function totalVolumeCredits(data) {
        ...
    }
```

最后，请读者进行编译、测试并提交代码。基于目前的代码结构，添加一个HTML 版本打印的方法非常简单（我顺手把 usd 函数也迁移到了顶层作用域中，这样 HTML 版本打印的方法也能直接调用它）。代码如下：

```
statement.js…
    function htmlStatement(invoice, plays) {
        return renderHtml(createStatementData(invoice, plays));
    }

    function renderHtml(data) {
        let result = `<h1>Statement for ${data.customer}</h1>\n`;
        result += "<table>\n";
        result += "<tr><th>play</th><th>seats</th><th>cost</th></tr>";
        for (let perf of data.performances) {
            result += `  <tr><td>${perf.play.name}</td>
<td>${perf.audience}</td>`;
            result += `<td>${usd(perf.amount)}</td></tr>\n`;
        }
        result += "</table>\n";
        result += `<p>Amount owed is <em>${usd(data.totalAmount)}</em>
</p>\n`;
        result += `<p>You earned <em>${data.totalVolumeCredits}</em>
credits</p>\n`;
        return result;
    }

    function usd(aNumber) {
        ...
    }
```

1.7 阶段性展示：两阶段文件隔离

至此，我们再次停下来浏览代码的全貌，并思考下一步的重构计划。目前的代码结构如下。

statement.js 文件：

```
import createStatementData from './createStatementData.js';

function statement(invoice, plays) {
   return renderPlainText(createStatementData(invoice, plays));
}

function renderPlainText(data) {
   let result = `Statement for ${data.customer}\n`;
   for (let perf of data.performances) {
      result += `  ${perf.play.name}: ${usd(perf.amount)}
(${perf.audience} seats)\n`;
   }
   result += `Amount owed is ${usd(data.totalAmount)}\n`;
   result += `You earned ${data.totalVolumeCredits} credits\n`;
   return result;
}

```

```
function htmlStatement(invoice, plays) {
   return renderHtml(createStatementData(invoice, plays));
}

function renderHtml(data) {
   let result = `<h1>Statement for ${data.customer}</h1>\n`;
   result += "<table>\n";
   result += "<tr><th>play</th><th>seats</th><th>cost</th></tr>";
   for (let perf of data.performances) {
      result += ` <tr><td>${perf.play.name}</td><td>${perf.audience}
</td>`;
      result += `<td>${usd(perf.amount)}</td></tr>\n`;
   }
   result += "</table>\n";
   result += `<p>Amount owed is <em>${usd(data.totalAmount)}</em>
</p>\n`;
   result += `<p>You earned <em>${data.totalVolumeCredits}</em>
credits</p>\n`;
   return result;
}

function usd(aNumber) {
   return new Intl.NumberFormat("en-US", {
         style: "currency",
         currency: "USD",
         minimumFractionDigits: 2
      })
      .format(aNumber / 100);
}
```

createStatementData.js 文件：

```
export default function createStatementData(invoice, plays) {
   const result = {};
   result.customer = invoice.customer;
   result.performances = invoice.performances.map(enrichPerformance);
   result.totalAmount = totalAmount(result);
   result.totalVolumeCredits = totalVolumeCredits(result);
   return result;

   function enrichPerformance(aPerformance) {
      const result = Object.assign({}, aPerformance);
      result.play = playFor(result);
      result.amount = amountFor(result);
      result.volumeCredits = volumeCreditsFor(result);
      return result;
   }

   function playFor(aPerformance) {
      return plays[aPerformance.playID]
   }

   function amountFor(aPerformance) {
      let result = 0;
      switch (aPerformance.play.type) {
         case "tragedy":
            result = 40000;
            if (aPerformance.audience > 30) {
               result += 1000 * (aPerformance.audience - 30);
            }
            break;
         case "comedy":
            result = 30000;
            if (aPerformance.audience > 20) {
               result += 10000 + 500 * (aPerformance.audience - 20);
```

```
            }
            result += 300 * aPerformance.audience;
            break;
         default:
            throw new Error(`unknown type: ${aPerformance.play.type}`);
      }
      return result;
   }

   function volumeCreditsFor(aPerformance) {
      let result = 0;
      result += Math.max(aPerformance.audience - 30, 0);
      if ("comedy" === aPerformance.play.type) result +=
Math.floor(aPerformance.audience / 5);
      return result;
   }

   function totalAmount(data) {
      return data.performances
         .reduce((total, p) => total + p.amount, 0);
   }

   function totalVolumeCredits(data) {
      return data.performances
         .reduce((total, p) => total + p.volumeCredits, 0);
   }
```

编程时，我们需要遵循露营规则（不要破坏环境，相反，要更好地呵护环境）：确保由你经手的代码比以前更加健壮。

与重构之前相比，代码行数从 44 行增加到 70 行（新增的 htmlStatement 函数除外），这主要是因为重构过程中我们进行了函数的封装。虽然代码的行数增加了，但是得到了一份结构清晰的、高可读性的、高扩展性的代码。我们通过对不同功能模块的封装，将混杂的逻辑重构成为更加清晰的结构。简洁是语言的灵魂，但清晰的结构是软件的灵魂。将代码结构清晰化、模块化，我们就可以很轻松地添加 HTML 版本的代码，并且复用计算模块的逻辑。

其实，我们还可以进一步地简化输出模块，但又感觉当前的代码已经足够令人满意了。我们经常会在进一步重构与添加新代码功能之间进行取舍，大多数人都会选择延缓重构，毕竟目前的代码结构足够优秀了。

没有完美的代码，但我们可以时刻鼓励自己，确保我们经手过的代码比以前更加健壮。

1.8 多态：按类型封装计算逻辑

接下来，我们继续重构之旅。我们需要单独封装不同表演类型的费用计算逻辑。请读者观察下 amountFor 函数，目前我们支持“喜剧类”（comedy）和“悲剧类”（tragedy）的费用计算，通过 switch-case 进行表演类型的判断。如果有新的表演类型，我们只需要在 switch-case 中添加新的分支逻辑即可。但是，随着表演类型的增加，分支逻辑也会相应增加，长此以往，必然会造成代码的堆叠。

目前，我们想要达到的效果是：“不同的表演类型使用不同的费用计算逻辑，同时保证清晰的代码结构及高可读性”。想要达到这一效果的方式很多，最常用的方式就是使用面向对象特性中的多态特性，非常自然。由于传统的面向对象特性在 JavaScript 语言中备受争议，因此，在 ECMAScript 2015 版本规范中，特意为多态提供了健全且实用的语法糖。所以，我们可以在合适的场景下使用面向对象的多态特性，当然了，需要因地制宜。

关于这部分代码整体的重构计划，首先创建表演类型的继承体系：“创建表演类型的父类（超类），并且将喜剧类和悲剧类作为两个子类，子类负责维护各自的费用计算逻辑。”调用端通过调用一个多态的 amount 函数，JS 语言会根据多态特性将调用分派给喜剧类或悲剧类的不同计算逻辑中。volumeCredits 函数也会使用相同的方式进行重构。在重构过程中，我们需要使用最核心的重构手法是*多态替代条件表达式法*（*详见 10.4 节*），使用多态特性替换掉条件表达式的多个分支。在应用*多态替代条件表达式法*之前，我们首先需要创建一个父类，并将费用计算逻辑和代金券计算逻辑放到该父类中。

创建父类之前，我们先检查当前代码（之前的重构工作对代码带来了极大的优化，能够大幅度缩小我们对代码的检查范围。我们可以忽略一些代码的检查，比如打印订单详情的代码。我们只需要关注中转数据结构的数据不被篡改即可。当然了，也可以添加一些测试用例，进一步保证中转数据结构不被篡改）。

此处应用多态特性，创建集成体系，与设计模式中的“策略模式”如出一辙。

当前 createStatementData.js 的代码：

```
export default function createStatementData(invoice, plays) {
  const result = {};
  result.customer = invoice.customer;
  result.performances = invoice.performances.map(enrichPerformance);
  result.totalAmount = totalAmount(result);
  result.totalVolumeCredits = totalVolumeCredits(result);
  return result;

  function enrichPerformance(aPerformance) {
    const result = Object.assign({}, aPerformance);
    result.play = playFor(result);
    result.amount = amountFor(result);
    result.volumeCredits = volumeCreditsFor(result);
    return result;
  }

  function playFor(aPerformance) {
    return plays[aPerformance.playID]
  }

  function amountFor(aPerformance) {
    let result = 0;
    switch (aPerformance.play.type) {
      case "tragedy":
        result = 40000;
        if (aPerformance.audience > 30) {
          result += 1000 * (aPerformance.audience - 30);
        }
        break;
      case "comedy":
        result = 30000;
        if (aPerformance.audience > 20) {
          result += 10000 + 500 * (aPerformance.audience - 20);
        }
        result += 300 * aPerformance.audience;
        break;
      default:
        throw new Error(`unknown type: ${aPerformance.play.type}`);
    }
    return result;
  }

  function volumeCreditsFor(aPerformance) {
    let result = 0;
```

```
45        result += Math.max(aPerformance.audience - 30, 0);
46        if ("comedy" === aPerformance.play.type) result +=
   Math.floor(aPerformance.audience / 5);
47        return result;
48     }
49
50     function totalAmount(data) {
51        return data.performances
52           .reduce((total, p) => total + p.amount, 0);
53     }
54
55     function totalVolumeCredits(data) {
56        return data.performances
57           .reduce((total, p) => total + p.volumeCredits, 0);
58     }
```

1. 创建父类——演出计算器（PerformanceCalculator）

观察 enrichPerformance 函数，它包含了费用计算函数（amountFor）及代金券计算函数（volumeCreditsFor）的调用，并将最终的结果填充到中转数据结构中。前文提到，此处重构的目的，就是借助多态，重构费用计算的相关逻辑，因此，enrichPerformance 函数是重构的关键所在。我们创建父类 PerformanceCalculator，并在 enrichPerformance 函数中调用新创建的父类。代码如下：

```
1 function createStatementData…
2     function enrichPerformance(aPerformance) {
3         const calculator = new PerformanceCalculator(aPerformance);
4         const result = Object.assign({}, aPerformance);
5         result.play = playFor(result);
6         result.amount = amountFor(result);
7         result.volumeCredits = volumeCreditsFor(result);
8         return result;
9     }
```

```
1 //顶层作用域...
2 class PerformanceCalculator {
3    constructor(aPerformance) {
4       this.performance = aPerformance;
5    }
6 }
```

目前，我们完成了父类 PerformanceCalculator 的创建，接下来，我们需要将相关的函数迁移到 PerformanceCalculator 中，我们可以从最简单的 play 字段开始迁移。严格来说，play 字段并没有多态的特性，不需要进行迁移。之所以对 play 字段进行迁移，是为了把相关的所有数据迁移到一起，这种集中性管理可以使代码更加清晰。我们使用*修改函数声明法*（详见 6.5 节）将 play 字段迁移到 PerformanceCalculator 中。代码如下：

```
1 function createStatementData…
2     function enrichPerformance(aPerformance) {
3         const calculator = new PerformanceCalculator(aPerformance,
  playFor(aPerformance));
4         const result = Object.assign({}, aPerformance);
5         result.play = calculator.play;
6         result.amount = amountFor(result);
7         result.volumeCredits = volumeCreditsFor(result);
8         return result;
9     }
```

```
class PerformanceCalculator…
    class PerformanceCalculator {
        constructor(aPerformance, aPlay) {
            this.performance = aPerformance;
            this.play = aPlay;
        }
    }
```

可能部分读者早已厌烦“编译、测试及提交代码”的字眼，所以从现在开始，不会再提及。希望读者能够在重构过程中真正地做到阶段性测试及代码提交，小步前行，步步为营。

2. 迁移计算函数

接下来，我们将费用计算函数（amountFor）及代金券计算函数（volumeCreditsFor）相关的逻辑迁移到父类 PerformanceCalculator 中。

首先，我们使用*函数迁移法*（详见 8.1 节），将 amountFor 函数的逻辑迁移到 PerformanceCalculator 中。考虑到 PerformanceCalculator 中已经包含 performance 属性和 play 属性，因此，amountFor 函数可以直接使用 this.perfomance 和 this.play。代码如下：

```
class PerformanceCalculator…
    get amount() {
        let result = 0;
        switch (this.play.type) {
            case "tragedy":
                result = 40000;
                if (this.performance.audience > 30) {
                    result += 1000 * (this.performance.audience - 30);
                }
                break;
            case "comedy":
                result = 30000;
                if (this.performance.audience > 20) {
                    result += 10000 + 500 * (this.performance.audience
- 20);
                }
                result += 300 * this.performance.audience;
                break;
            default:
                throw new Error(`unknown type: ${this.play.type}`);
        }
        return result;
    }
```

其次，将之前的 amountFor 函数简化为对 PerformanceCalculator 中新方法的调用。代码如下：

```
function createStatementData…
    function amountFor(aPerformance) {
        return new PerformanceCalculator(aPerformance,
playFor(aPerformance)).amount;
    }
```

最后，使用*内联函数法*（详见 6.2 节），将 amountFor 函数的调用点修改为对新方法的调用。代码如下：

```
function createStatementData…
    function enrichPerformance(aPerformance) {
```

```
        const calculator = new PerformanceCalculator(aPerformance,
playFor(aPerformance));
        const result = Object.assign({}, aPerformance);
        result.play = calculator.play;
        result.amount = calculator.amount;
        result.volumeCredits = volumeCreditsFor(result);
        return result;
    }
```

我们使用同样的步骤，对 volumeCreditsFor 函数进行迁移。代码如下：

```
function createStatementData…
    function enrichPerformance(aPerformance) {
        const calculator = new PerformanceCalculator(aPerformance,
playFor(aPerformance));
        const result = Object.assign({}, aPerformance);
        result.play = calculator.play;
        result.amount = calculator.amount;
        result.volumeCredits = calculator.volumeCredits;
        return result;
    }
```

```
class PerformanceCalculator…
    get volumeCredits() {
        let result = 0;
        result += Math.max(this.performance.audience - 30, 0);
        if ("comedy" === this.play.type) result +=
Math.floor(this.performance.audience / 5);
        return result;
    }
```

3. 应用多态特性

至此，我们已经将相关逻辑迁移到 PerformanceCalculator 中，是时候融入多态特性了。第一步我们需要使用子类替代类型法（详见 12.6 节），用子类替换相关的类型判断代码。因此，我们需要为 PerformanceCalculator 创建子类，并在 createStatementData 中获取对应的子类。由于 JavaScript 的构造函数中无法返回子类类型，因此，我们需要将构造函数的调用修改为普通函数的调用。此处，我们使用的重构手法是工厂函数替代构造函数法（详见 11.8 节），代码如下。

```
function createStatementData…
    function enrichPerformance(aPerformance) {
        const calculator = createPerformanceCalculator(aPerformance,
playFor(aPerformance));
        const result = Object.assign({}, aPerformance);
        result.play = calculator.play;
        result.amount = calculator.amount;
        result.volumeCredits = calculator.volumeCredits;
        return result;
    }
```

```
//顶层作用域…
    function createPerformanceCalculator(aPerformance, aPlay) {
        return new PerformanceCalculator(aPerformance, aPlay);
    }
```

将构造函数修改为普通函数调用后，我们就可以根据不同的表演类型，创建出不同的子类。代码如下：

```
//顶层作用域...
    function createPerformanceCalculator(aPerformance, aPlay) {
        switch (aPlay.type) {
```

```
            case "tragedy":
                return new TragedyCalculator(aPerformance, aPlay);
            case "comedy":
                return new ComedyCalculator(aPerformance, aPlay);
            default:
                throw new Error(`unknown type: ${aPlay.type}`);
        }
    }
    class TragedyCalculator extends PerformanceCalculator {}
    class ComedyCalculator extends PerformanceCalculator {}
```

至此，继承体系搭建完毕了，接下来，我们就可以应用多态替代条件表达式法（详见 10.4 节）了。

我们先对悲剧类的相关计算逻辑进行迁移。代码如下：

```
class TragedyCalculator…
    get amount() {
        let result = 40000;
        if (this.performance.audience > 30) {
            result += 1000 * (this.performance.audience - 30);
        }
        return result;
    }
```

然后，对喜剧类进行逻辑迁移。代码如下：

```
class ComedyCalculator…
    get amount() {
        let result = 30000;
        if (this.performance.audience > 20) {
            result += 10000 + 500 * (this.performance.audience - 20);
        }
        result += 300 * this.performance.audience;
        return result;
    }
```

目前来看，我们可以直接删除 PerformanceCalculator 中的 amount 方法，并不会对程序的运行产生影响。但此处，我们并不会删除这个方法，便于提醒我们子类需要实现这个方法。我们将 amount 方法做如下处理：

```
class PerformanceCalculator…
    get amount() {
        throw new Error('subclass responsibility');
    }
```

紧接着，我们对代金券计算的函数 volumeCredits 也进行同样的操作。通过对演出数据的检查，发现大多数代金券的计算逻辑都会检查观众人数是否超过 30 人。因此，我将该逻辑作为默认逻辑放到父类的方法中，其他不一样的需求，覆写代码即可。代码如下：

```
class PerformanceCalculator…
    get volumeCredits() {
        return Math.max(this.performance.audience - 30, 0);
    }
```

```
class ComedyCalculator…
    get volumeCredits() {
        return super.volumeCredits +
  Math.floor(this.performance.audience / 5);
    }
```

1.9 阶段性展示：多态下的计算模块

本章示例的重构基本完结，让我们来看看引入面向对象的多态特性后，代码带给我们的直观感受。代码如下：

```
createStatementData.js
    export default function createStatementData(invoice, plays) {
        const result = {};
        result.customer = invoice.customer;
        result.performances =
invoice.performances.map(enrichPerformance);
        result.totalAmount = totalAmount(result);
        result.totalVolumeCredits = totalVolumeCredits(result);
        return result;

        function enrichPerformance(aPerformance) {
            const calculator =
createPerformanceCalculator(aPerformance, playFor(aPerformance));
            const result = Object.assign({}, aPerformance);
            result.play = calculator.play;
            result.amount = calculator.amount;
            result.volumeCredits = calculator.volumeCredits;
            return result;
        }

        function playFor(aPerformance) {
            return plays[aPerformance.playID]
        }

        function totalAmount(data) {
            return data.performances
                .reduce((total, p) => total + p.amount, 0);
        }

        function totalVolumeCredits(data) {
            return data.performances
                .reduce((total, p) => total + p.volumeCredits, 0);
        }
    }

    function createPerformanceCalculator(aPerformance, aPlay) {
        switch (aPlay.type) {
            case "tragedy":
                return new TragedyCalculator(aPerformance, aPlay);
            case "comedy":
                return new ComedyCalculator(aPerformance, aPlay);
            default:
                throw new Error(`unknown type: ${aPlay.type}`);
        }
    }
    class PerformanceCalculator {
        constructor(aPerformance, aPlay) {
            this.performance = aPerformance;
            this.play = aPlay;
        }
        get amount() {
            throw new Error('subclass responsibility');
        }
        get volumeCredits() {
            return Math.max(this.performance.audience - 30, 0);
        }
    }
    class TragedyCalculator extends PerformanceCalculator {
        get amount() {
```

```
            let result = 40000;
            if (this.performance.audience > 30) {
                result += 1000 * (this.performance.audience - 30);
            }
            return result;
        }
    }
    class ComedyCalculator extends PerformanceCalculator {
        get amount() {
            let result = 30000;
            if (this.performance.audience > 20) {
                result += 10000 + 500 * (this.performance.audience -
20);
            }
            result += 300 * this.performance.audience;
            return result;
        }
        get volumeCredits() {
            return super.volumeCredits +
Math.floor(this.performance.audience / 5);
        }
    }
```

与上一阶段相比，代码量又有所增加，毕竟我们进行了新的封装。新代码利用了面向对象的多态特性，将计算逻辑进行了新的重构。基于当前的代码结构，如果我们需要修改某种表演类型的计费逻辑，会非常容易找到修改点，并且不会影响其他的代码逻辑；如果我们需要添加新的表演类型及计费逻辑，只需要创建一个新的子类，并在 createPerformanceCalculator 方法中添加返回该子类的逻辑即可。

通过对该示例的重构，希望读者能够体会到“新增继承结构并融入多态特性在什么场景下最为有效”。在本章示例中，我们将包含两个函数（amountFor 和 volumeCreditsFor）的多个条件分支迁移到 createPerformanceCalculator 函数中进行统一管理，条件分支越多，就越能够发挥出这种方案的价值。

除了上边的方式，我们还可以选择让 createStatementData 直接返回费用计算实例本身，不需要自己进行中转数据的填充，当然，重构的方式非常多。我在选择是“直接返回费用计算实例”还是“获取单独的费用数据并组装中转数据结构”时，主要考虑数据的使用者，在当前示例中，我更推荐“获取单独的费用数据并组装中转数据结构”，以此来隐藏应用了多态特性的子类计算实例。

设计模式中的迪米特法则，又称最少知道原则。作者此处选择了“获取单独的费用数据并组装中转数据结构”，就是为了隐藏子类的实现类。与设计模式中的迪米特法则如出一辙。

1.10 本章小结

本章的示例虽然简单，但它使我们对“如何重构”有了最初的认识。重构过程中，我们已经领略了多种重构手法，比如*提炼函数法*（*参见 6.1 节*）、*内联变量法*（*详见 6.4 节*）、*函数迁移法*（*详见 8.1 节*）、*多态替代条件表达式法*（*详见 10.4 节*）等。

整个重构过程中有三个重要阶段：大量函数嵌套阶段（详见 1.5 节）、两阶段文件隔离（详见 1.7 节）以及多态下的费用计算阶段（详见 1.9 节）。每个阶段都进行了新的封装，最终使整个代码结构更加清晰。

通常情况下，重构的念头出现在初期代码阅读的过程中。一个常见的规律是：初期阅读代码，获得一些想法；然后对代码进行重构，将这些想法应用到代码中。随着代码结构越来越清晰，你的重构设计理念也会随之变化，最终形成一个良性循环。

什么样的代码才是好的代码？不同的开发从业者可能会有不同的评判标准。而对于我来说，结构清晰、容易修改、容易扩展的代码才是好代码。好的代码应该是结构清晰的，当我们需要修改代码时，能够轻松、快速地找到需要修改的点，并且在修改时，不会影响其他的代码模块。

好代码的检验标准——结构清晰，易修改，易扩展，易维护。

另外，重构的节奏非常重要。每当我向其他人展示我如何重构时，他们都会对我“小步重构，频繁测试，步步为营”的做法感到惊讶。我很理解他们的感受，就像 20 年前，在底特律的一家旅馆中，Kent Beck 向我展示同样的手法时，我也对此感到惊讶。希望读者能够牢记重构的关键心得：小步重构，步步为营，时刻保证代码处于可运行的状态，稳健且小的步子可以更快地完成重构。