

## 第1章 重构，第一个示例

我该从何说起呢？按照传统做法，一开始介绍某样东西时应该先大致讲讲它的历史、主要原理等。可是每当有人在会场上介绍这些东西，总是诱发我的瞌睡虫。我的思绪开始游荡，我的眼神开始迷离，直到主讲人秀出示例，我才能够提起精神。

示例之所以可以拯救我于太虚之中，因为它让我看见事情在真正进行。谈原理，很容易流于泛泛，又很难说明如何实际应用。给出一个示例，就可以帮助我把事情认识清楚。

因此，我决定从一个示例说起。在此过程中我会谈到很多重构的工作方式，并且让你对重构过程有一点点感觉。然后在下一章中我才能向你展开通常的原理介绍。

但是，面对这个介绍性示例，我遇到了一个大问题。如果我选择一个大型程序，那么对程序自身的描述和对整个重构过程的描述就太复杂了，任何读者都不忍卒读（我试了一下，哪怕稍微复杂一点的例子都会超过100页）。如果我选择一个容易理解的小程序，又恐怕看不出重构的价值。

和任何立志要介绍“应用于真实世界的程序中的有用技术”的人一样，我陷入了一个十分典型的两难困境。我只能带你看看如何在一个我选择的小程序中进行重构，然而坦白说，那个程序的规模根本不值得我们这么做。但是，如果我给你看的代码是大系统的一部分，重构技术很快就变得重要起来。所以请你一边观赏这个小例子，一边想象它身处于一个大得多的系统。

## 1.1 起点

在本书第1版中，我使用的示例程序是为影片出租店的顾客打印一张详单。放到今天，很多人可能要问了：“影片出租店是什么？”为了避免过多回答这个问题，我翻新了一下示例，将其包装成一个仍有古典韵味又尚未消亡的现代示例。

设想有一个戏剧演出团，演员们经常要去各种场合表演戏剧。通常客户（customer）会指定几出剧目，而剧团则根据观众（audience）人数及剧目类型来向客户收费。该团目前出演两种戏剧：悲剧（tragedy）和喜剧（comedy）。给客户发出账单时，剧团还会根据到场观众的数量给出“观众量积分”（volume credit）优惠，下次客户再请剧团表演时可以使用积分获得折扣——你可以把它看作一种提升客户忠诚度的方式。

该剧团将剧目的数据存储在简单的JSON文件中。

### plays.json...

```
{
  "hamlet": {"name": "Hamlet", "type": "tragedy"},
  "as-like": {"name": "As You Like It", "type": "comedy"},
  "othello": {"name": "Othello", "type": "tragedy"}
}
```

他们开出的账单也存储在一个JSON文件里。

### invoices.json...

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

下面这个简单的函数用于打印账单详情。

```
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}`);
    }

    // add volume credits
    volumeCredits += Math.max(perf.audience - 30, 0);
    // add extra credit for every ten comedy attendees
    if ("comedy" === play.type) volumeCredits += Math.floor(perf.audience / 5);

    // print line for this order
    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;
}
```

用上面的数据文件（`invoices.json`和`plays.json`）作为测试输入，运行这段代码，会得到如下输出：

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

## 1.2 对此起始程序的评价

你觉得这个程序设计得怎么样？我的第一感觉是，代码组织不甚清晰，但这还在可忍受的限度内。这样小的程序，不做任何深入的设计，也不会太难理解。但我前面讲过，这是因为要保证例子足够小的缘故。如果这段代码身处于一个更大规模——也许是几百行——的程序中，把所有代码放到一个函数里就很难理解了。

尽管如此，这个程序还是能正常工作。那么是不是说，对其结构“不甚清晰”的评价只是美学意义上的判断，只是对所谓丑陋代码的反感呢？毕竟编译器也不会在乎代码好不好看。但是，当我们需要修改系统时，就涉及了人，而人在乎这些。差劲的系统是很难修改的，因为很难找到修改点，难以了解做出的修改与现有代码如何协作实现我想要的行为。如果很难找到修改点，我就很有可能犯错，从而引入bug。

因此，如果我需要修改一个有几百行代码的程序，我会期望它有良好的结构，并且已经被分解成一系列函数和其他程序要素，这能帮我更易于清楚地了解这段代码在做什么。如果程序杂乱无章，先为它整理出结构来，再做需要的修改，通常来说更加简单。

💡 如果你要给程序添加一个特性，但发现代码因缺乏良好的结构而不易于进行更改，那就先重构那个程序，使其比较容易添加该特性，然后再添加该特性。

在这个例子里，我们的用户希望对系统做几个修改。首先，他们希望以HTML格式输出详单。现在请你想一想，这个变化会带来什么影响。对于每处追加字符串到`result`变量的地方我都得为它们添加分支逻辑。这会为函数引入更多复杂度。遇到这种需求时，很多人会选择直接复制整个方法，在其中修改输出HTML的部分。复制一遍代码似乎不算太难，但却给未来留下各种隐患：一旦计费逻辑发生变化，我就得同时修改两个地方，以保证它们逻辑相同。如果你编写的是一个永不需要修改的程序，这样剪剪贴贴就还好。但如果程序要保存很长时间，那么重复的逻辑就会造成潜在的威胁。

现在，第二个变化来了：演员们尝试在表演类型上做更多突破，无论是历史剧、田园剧、田园喜剧、田园史剧、历史悲剧还是历史田园悲喜剧，无论一成不变的正统戏，还是千变万幻的新派戏，他们都希望有所尝试，只是还没有决定试

哪种以及何时试演。这对戏剧场次的计费方式、积分的计算方式都有影响。作为一个经验丰富的开发者，我可以肯定：不论最终提出什么方案，他们一定会在6个月之内再次修改它。毕竟，需求通常不来则已，一来便会接踵而至。

为了应对分类规则和计费规则的变化，程序必须对statement函数做出修改。但如果我把statement内的代码复制到用以打印HTML详单的函数中，就必须确保将来的任何修改在这两个地方保持一致。随着各种规则变得越来越复杂，适当的修改点将越来越难找，不犯错的机会也越来越少。

我再强调一次，是需求的变化使重构变得必要。如果一段代码能正常工作，并且不会再被修改，那么完全可以不去重构它。能改进之当然很好，但若没人需要去理解它，它就不会真正妨碍什么。如果确实有人需要理解它的工作原理，并且觉得理解起来很费劲，那你就需要改进一下代码了。

## 1.3 重构的第一步

每当我要进行重构的时候，第一个步骤永远相同：我得确保即将修改的代码拥有一组可靠的测试。这些测试必不可少，因为尽管遵循重构手法可以使我避免绝大多数引入bug的情形，但我毕竟是人，毕竟有可能犯错。程序越大，我的修改不小心破坏其他代码的可能性就越大——在数字时代，软件的名字就是脆弱。

`statement`函数的返回值是一个字符串，我做的就是创建几张新的账单（`invoice`），假设每张账单收取了几出戏剧的费用，然后使用这几张账单作为输入调用`statement`函数，生成对应的对账单（`statement`）字符串。我会拿生成的字符串与我已经手工检查过的字符串做比对。我会借助一个测试框架来配置好这些测试，只要在开发环境中输入一行命令就可以把它们运行起来。运行这些测试只需几秒钟，所以你会看到我经常运行它们。

测试过程中很重要的一部分，就是测试程序对于结果的报告方式。它们要么变绿，表示所有新字符串都和参考字符串一样，要么就变红，然后列出失败清单，显示问题字符串的出现行号。这些测试都能够自我检验。使测试能自我检验至关重要，否则就得耗费大把时间来回比对，这会降低开发速度。现代的测试框架都提供了丰富的设施，支持编写和运行能够自我检验的测试。

💡 重构前，先检查自己是否有一套可靠的测试集。这些测试必须有自我检验能力。

进行重构时，我需要依赖测试。我将测试视为bug检测器，它们能保护我不被自己犯的错误所困扰。把我想要达成的目标写两遍——代码里写一遍，测试里再写一遍——我就得犯两遍同样的错误才能骗过检测器。这降低了我犯错的概率，因为我对工作进行了二次确认。尽管编写测试需要花费时间，但却为我节省下可观的调试时间。构筑测试体系对重构来说实在太重要了，因此我将用第4章一整章的笔墨来详细讨论它。

## 1.4 分解statement函数

每当看到这样长长的函数，我便下意识地想从整个函数中分离出不同的关注点。第一个引起我注意的就是中间那段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}`);
    }

    // add volume credits
    volumeCredits += Math.max(perf.audience - 30, 0);
    // add extra credit for every ten comedy attendees
    if ("comedy" === play.type) volumeCredits += Math.floor(perf.audience / 5);

    // print line for this order
    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;
}
```

看着这块代码，我就知道它在计算一场戏剧演出的费用。这是我的直觉。不过正如Ward Cunningham所说，这种理解只是我脑海中转瞬即逝的灵光。我需要梳理这些灵感，将它们从脑海中搬回到代码里去，以免忘记。这样当我回头看时，代码就能告诉我它在干什么，我不需要重新思考一遍。

要将我的理解转化到代码里，得先将这块代码抽取成一个独立的函数，按它



所干的事情给它命名，比如叫`amountFor(performance)`。每次想将一块代码抽取成一个函数时，我都会遵循一个标准流程，最大程度减少犯错的可能。我把这个流程记录了下来，并将它命名为提炼函数（106），以便日后可以方便地引用。

首先，我需要检查一下，如果我将这块代码提炼到自己的一个函数里，有哪些变量会离开原本的作用域。在此示例中，是`perf`、`play`和`thisAmount`这3个变量。前两个变量会被提炼后的函数使用，但不会被修改，那么我就可以将它们以参数方式传递进来。我更关心那些会被修改的变量。这里只有唯一一个——`thisAmount`，因此可以将它从函数中直接返回。我还可以将其初始化放到提炼后的函数里。修改后的代码如下所示。

### 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;
}
```

当我在代码块上方使用了斜体（中文对应为楷体）标记的题头“*function xxx*”时，表明该代码块位于题头所在函数、文件或类的作用域内。通常该作用域内还有其他的代码，但由于不是讨论重点，因此把它们隐去不展示。

现在原`statement`函数可以直接调用这个新函数来初始化`thisAmount`。

### 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);

  // add volume credits
  volumeCredits += Math.max(perf.audience - 30, 0);
  // add extra credit for every ten comedy attendees
  if ("comedy" === play.type) volumeCredits += Math.floor(perf.audience / 5);


  // print line for this order
  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;

```

做完这个改动后，我会马上编译并执行一遍测试，看看有无破坏了其他东西。无论每次重构多么简单，养成重构后即运行测试的习惯非常重要。犯错误是很容易的——至少我知道我是很容易犯错的。做完一次修改就运行测试，这样在我真的犯了错时，只需要考虑一个很小的改动范围，这使得查错与修复问题易如反掌。这就是重构过程的精髓所在：小步修改，每次修改后就运行测试。如果我改动了太多东西，犯错误就可能陷入麻烦的调试，并为此耗费大把时间。小步修改，以及它带来的频繁反馈，正是防止混乱的关键。

这里我使用的“编译”一词，指的是将JavaScript变为可执行代码之前的所有步骤。虽然JavaScript可以直接执行，有时可能不需任何步骤，但有时可能需要将代码移动到一个输出目录，或使用Babel这样的代码处理器等。

因为是JavaScript，我可以直接将amountFor提炼成为statement的一个内嵌函数。这个特性十分有用，因为我不需要再把外部作用域中的数据传给新提炼的函数。这个示例中可能区别不大，但也是少了一件要操心的事。

 重构技术就是以微小的步伐修改程序。如果你犯下错误，很容易便可发现它。

做完上面的修改，测试是通过的，因此下一步我要把代码提交到本地的版本控制系统。我会使用诸如git或mercurial这样的版本控制系统，因为它们可以支持本地提交。每次成功重构后我都会提交代码，如果待会不小心搞砸了，我便能轻松回滚到上一个可工作的状态。把代码推送（push）到远端仓库前，我会把零碎的修改压缩成一个更有意义的提交（commit）。

提炼函数（106）是一个常见的可自动完成的重构。如果我是用Java编程，我会本能地使用IDE的快捷键来完成这项重构。在我撰写本书时，JavaScript工具对此重构的支持仍不是很健壮，因此我必须手动重构。这不是很难，当然我还是需要小心处理那些局部作用域的变量。

完成提炼函数（106）手法后，我会看看提炼出来的函数，看是否能进一步提升其表达能力。一般我做的第一件事就是给一些变量改名，使它们更简洁，比如将`thisAmount`重命名为`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;
}
```

这是我个人的编码风格：永远将函数的返回值命名为“`result`”，这样我一眼就能知道它的作用。然后我再次编译、测试、提交代码。接着，我前往下一个目标——函数参数。

## 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;
}
```

这是我的另一个编码风格。使用一门动态类型语言（如JavaScript）时，跟踪变量的类型很有意义。因此，我为参数取名时都默认带上其类型名。一般我会使用不定冠词修饰它，除非命名中另有解释其角色的相关信息。这个习惯是从Kent Beck那里学的[Beck SBPP]，到现在我还一直觉得很有用。

💡 傻瓜都能写出计算机可以理解的代码。唯有能写出人类容易理解的代码的，才是优秀的程序员。

这次改名是否值得我大费周章呢？当然值得。好代码应能清楚地表明它在做什么，而变量命名是代码清晰的关键。只要改名能够提升代码的可读性，那就应该毫不犹豫去做。有好的查找替换工具在手，改名通常并不困难；此外，你的测试以及语言本身的静态类型支持，都可以帮你揪出漏改的地方。如今有了自动化的重构工具，即便要给一个被大量调用的函数改名，通常也不在话下。

本来下一个要改名的变量是play，但我对这个参数另有安排。

## 移除play变量

观察amountFor函数时，我会看看它的参数都从哪里来。aPerformance是从循环变量中来，所以自然每次循环都会改变，但play变量是由performance变量计算得到的，因此根本没必要将它作为参数传入，我可以在amountFor函数中重新计算得到它。当我分解一个长函数时，我喜欢将play这样的变量移除掉，因为它们创建了很多具有局部作用域的临时变量，这会使提炼函数更加复杂。这里我要使用的重构手法是以查询取代临时变量（178）。

我先从赋值表达式的右边部分提炼出一个函数来。

### function statement...

```
function playFor(aPerformance) {
```

```
    return plays[aPerformance.playID];  
  }  
}
```

## 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);  
  
    // add volume credits  
    volumeCredits += Math.max(perf.audience - 30, 0);  
    // add extra credit for every ten comedy attendees  
    if ("comedy" === play.type) volumeCredits += Math.floor(perf.audience / 5);  
  
    // print line for this order  
    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;  
}
```

编译、测试、提交，然后使用内联变量（123）手法内联play变量。

## 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));  
  
    // add volume credits  
    volumeCredits += Math.max(perf.audience - 30, 0);  
    // add extra credit for every ten comedy attendees  
    if ("comedy" === playFor(perf).type) volumeCredits += Math.floor(perf.audience / 5);  
  
    // print line for this order  
    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;  
}
```

编译、测试、提交。完成变量内联后，我可以对amountFor函数应用改变函数声明（124），移除play参数。我会分两步走。首先在amountFor函数内部使用新提炼的函数。

## 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;
}
```

编译、测试、提交，最后将参数删除。

## 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));

    // add volume credits
    volumeCredits += Math.max(perf.audience - 30, 0);
    // add extra credit for every ten comedy attendees
    if ("comedy" === playFor(perf).type) volumeCredits += Math.floor(perf.audience / 5);

    // print line for this order
    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;
}
```

然后再一次编译、测试、提交。

这次重构可能在一些程序员心中敲响警钟：重构前，查找`play`变量的代码在每次循环中只执行了1次，而重构后却执行了3次。我会在后面探讨重构与性能之间的关系，但现在，我认为这次改动还不太可能对性能有严重影响，即便真的有所影响，后续再对一段结构良好的代码进行性能调优，也容易得多。

移除局部变量的好处就是做提炼时会简单得多，因为需要操心的局部作用域变少了。实际上，在做任何提炼前，我一般都会先移除局部变量。

处理完`amountFor`的参数后，我回过头来看一下它的调用点。它被赋值给一个临时变量，之后就不再被修改，因此我又采用内联变量（123）手法内联它。

## 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) {

    // add volume credits
    volumeCredits += Math.max(perf.audience - 30, 0);
    // add extra credit for every ten comedy attendees
    if ("comedy" === playFor(perf).type) volumeCredits += Math.floor(perf.audience / 5);
  }
}
```

```
    // print line for this order
    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;
}
```

## 提炼计算观众量积分的逻辑

现在statement函数的内部实现是这样的。

### 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) {

    // add volume credits
    volumeCredits += Math.max(perf.audience - 30, 0);
    // add extra credit for every ten comedy attendees
    if ("comedy" === playFor(perf).type) volumeCredits += Math.floor(perf.audience / 5);

    // print line for this order
    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;
}
```

这会儿我们就看到了移除play变量的好处，移除了一个局部作用域的变量，提炼观众量积分的计算逻辑又更简单一些。

我仍需要处理其他两个局部变量。perf同样可以轻易作为参数传入，但volumeCredits变量则有些棘手。它是一个累加变量，循环的每次迭代都会更新它的值。因此最简单的方式是，将整块逻辑提炼到新函数中，然后在新函数中直接返回volumeCredits。

### 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);

    // print line for this order
    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;
}
```

我还顺便删除了多余（并且会引起误解）的注释。

编译、测试、提交，然后对新函数里的变量改名。

## function statement...

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

这里我只展示了一步到位的改名结果，不过实际操作时，我还是一次只将一个变量改名，并在每次改名后执行编译、测试、提交。

## 移除format变量

我们再看一下statement这个主函数。

## 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);

    // print line for this order
    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;
}
```

正如我上面所指出的，临时变量往往会带来麻烦。它们只在对其进行处理的代码块中 useful，因此临时变量实质上是鼓励你写长而复杂的函数。因此，下一步我要替换掉一些临时变量，而最简单的莫过于从 `format` 变量入手。这是典型的“将函数赋值给临时变量”的场景，我更愿意将其替换为一个明确声明的函数。

## function statement...

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

## 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);

    // print line for this order
    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;
}
```

尽管将函数变量改变成函数声明也是一种重构手法，但我既未为此手法命名，也未将它纳入重构名录。还有很多的重构手法我都觉得没那么重要。我觉得上面这个函数改名的手法既十分简单又不太常用，不值得在重构名录中占有一席之地。

我对提炼得到的函数名称不很满意——`format`未能清晰地描述其作用。`formatAsUSD`很表意，但又太长，特别它仅是小范围地被用在一个字符串模板中。我认为这里真正需要强调的是，它格式化的是一个货币数字，因此我选取了一个能体现此意图的命名，并应用了改变函数声明（124）手法。

## 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);

    // print line for this order
    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);
}
```

好的命名十分重要，但往往并非唾手可得。只有恰如其分地命名，才能彰显出将大函数分解成小函数的价值。有了好的名称，我就不必通过阅读函数体来了解其行为。但要一次把名取好并不容易，因此我会使用当下能想到最好的那个。如果稍后想到更好的，我就会毫不犹豫地换掉它。通常你需要花几秒钟通读更多代码，才能发现最好的名称是什么。

重命名的同时，我还将重复的除以100的行为也搬移到函数里。将钱以美分为单位作为正整数存储是一种常见的做法，可以避免使用浮点数来存储货币的小

数部分，同时又不影响用数学运算符操作它。不过，对于这样一个以美分为单位的整数，我又需要以美元为单位进行展示，因此让格式化函数来处理整除的事宜再好不过。

## 移除观众量积分总和

我的下一个重构目标是`volumeCredits`。处理这个变量更加微妙，因为它是在循环的迭代过程中累加得到的。第一步，就是应用拆分循环（227）将`volumeCredits`的累加过程分离出来。

### 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) {

    // print line for this order
    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;
}
```

完成这一步，我就可以使用移动语句（223）手法将变量声明挪动到紧邻循环的位置。

### top level...

```
function statement (invoice, plays) {
  let totalAmount = 0;
  let result = `Statement for ${invoice.customer}\n`;
  for (let perf of invoice.performances) {

    // print line for this order
    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;
```

把与更新`volumeCredits`变量相关的代码都集中到一起，有利于以查询取代临时变量（178）手法的施展。第一步同样是先对变量的计算过程应用提炼函数（106）手法。

## function statement...

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

## 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) {

    // print line for this order
    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;
}
```

完成函数提炼后，我再应用内联变量（123）手法内联`totalVolumeCredits`函数。

## 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) {

    // print line for this order
    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;
```

重构至此，让我先暂停一下，谈谈刚刚完成的修改。首先，我知道有些读者会再次对此修改可能带来的性能问题感到担忧，我知道很多人本能地警惕重复的循环。但大多数时候，重复一次这样的循环对性能的影响都可忽略不计。如果你在重构前后进行计时，很可能甚至都注意不到运行速度的变化——通常也确实没什么变化。许多程序员对代码实际的运行路径都所知不足，甚至经验丰富的程序员有时也未能避免。在聪明的编译器、现代的缓存技术面前，我们很多直觉都是不准确的。软件的性能通常只与代码的一小部分相关，改变其他的部分往往对总体性能贡献甚微。

当然，“大多数时候”不等同于“所有时候”。有时，一些重构手法也会显著地影响性能。但即便如此，我通常也不去管它，继续重构，因为有了一份结构良好的代码，回头调优其性能也容易得多。如果我在重构时引入了明显的性能损耗，我后面会花时间进行性能调优。进行调优时，可能会回退我早先做的一些重构——但更多时候，因为重构我可以使用更高效的调优方案。最后我得到的是既整洁又高效的代码。

因此对于重构过程的性能问题，我总体的建议是：大多数情况下可以忽略它。如果重构引入了性能损耗，先完成重构，再做性能优化。

其次，我希望你能注意到：我们移除`volumeCredits`的过程是多么小步。整个过程一共有4步，每一步都伴随着一次编译、测试以及向本地代码库的提交：

- 使用拆分循环（227）分离出累加过程；
- 使用移动语句（223）将累加变量的声明与累加过程集中到一起；
- 使用提炼函数（106）提炼出计算总数的函数；
- 使用内联变量（123）完全移除中间变量。

我得坦白，我并非总是如此小步——但在事情变复杂时，我的第一反应就是采用更小的步子。怎样算变复杂呢，就是当重构过程有测试失败而我又无法马上看清问题所在并立即修复时，我就会回滚到最后一次可工作的提交，然后以更小的步子重做。这得益于我如此频繁地提交。特别是与复杂代码打交道时，细小的步子是快速前进的关键。

接着我要重复同样的步骤来移除`totalAmount`。我以拆解循环开始（编译、测试、提交），然后下移累加变量的声明语句（编译、测试、提交），最后再提炼函数。这里令我有点头疼的是：最好的函数名应该是`totalAmount`，但它已经被变

量名占用，我无法起两个同样的名字。因此，我在提炼函数时先给它随便取了一个名字（然后编译、测试、提交）。

## function statement...

```
function appleSauce() {
  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`;
  }
  let totalAmount = appleSauce();

  result += `Amount owed is ${usd(totalAmount)}\n`;
  result += `You earned ${totalVolumeCredits()} credits\n`;
  return result;
}
```

接着我将变量内联（编译、测试、提交），然后将函数名改回 `totalAmount`（编译、测试、提交）。

## 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;
}
```

## function statement...

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



```
}
```

趁着给新提炼的函数改名的机会，我顺手一并修改了函数内部的变量名，以便保持我一贯的编码风格。

### function statement...

```
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 进展：大量嵌套函数

重构至此，是时候停下来欣赏一下代码的全貌了。

```
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;
  }
  function totalVolumeCredits() {
    let result = 0;
    for (let perf of invoice.performances) {
      result += volumeCreditsFor(perf);
    }
    return result;
  }
  function usd(aNumber) {
    return new Intl.NumberFormat("en-US",
      { style: "currency", currency: "USD",
        minimumFractionDigits: 2 }).format(aNumber/100);
  }
  function volumeCreditsFor(aPerformance) {
    let result = 0;
    result += Math.max(aPerformance.audience - 30, 0);
    if ("comedy" === playFor(aPerformance).type) result += Math.floor(aPerformance.audience /
5);
    return result;
  }
  function playFor(aPerformance) {
    return plays[aPerformance.playID];
  }
  function amountFor(aPerformance) {
    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;  
  }  
}
```

现在代码结构已经好多了。顶层的`statement`函数现在只剩7行代码，而且它处理的都是与打印详单相关的逻辑。与计算相关的逻辑从主函数中被移走，改由一组函数来支持。每个单独的计算过程和详单的整体结构，都因此变得更易理解了。

## 1.6 拆分计算阶段与格式化阶段

到目前为止，我的重构主要是为原函数添加足够的结构，以便我能更好地理解它，看清它的逻辑结构。这也是重构早期的一般步骤。把复杂的代码块分解为更小的单元，与好的命名一样都很重要。现在，我可以更多关注我要修改的功能部分了，也就是为这张详单提供一个HTML版本。不管怎么说，现在改起来更加简单了。因为计算代码已经被分离出来，我只需要为顶部的7行代码实现一个HTML的版本。问题是，这些分解出来的函数嵌套在打印文本详单的函数中。无论嵌套函数组织得多么良好，我总不想将它们全复制粘贴到另一个新函数中。我希望同样的计算函数可以被文本版详单和HTML版详单共用。

要实现复用有许多种方法，而我最喜欢的技术是拆分阶段（154）。这里我的目标是将逻辑分成两部分：一部分计算详单所需的数据，另一部分将数据渲染成文本或HTML。第一阶段会创建一个中转数据结构，再把它传递给第二阶段。

要开始拆分阶段（154），我会先对组成第二阶段的代码应用提炼函数（106）。在这个例子中，这部分代码就是打印详单的代码，其实也就是statement函数的全部内容。我要把它们与所有嵌套的函数一起抽取到一个新的顶层函数中，并将其命名为renderPlainText。

```
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) {...}
```

编译、测试、提交，接着创建一个对象，作为在两个阶段间传递的中转数据结构，然后将它作为第一个参数传递给renderPlainText（然后编译、测试、提交）。

```
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) {...}

```

现在我要检查一下renderPlainText用到的其他参数。我希望将它们挪到这个中转数据结构里，这样所有计算代码都可以被挪到statement函数中，让renderPlainText只操作通过data参数传进来的数据。

第一步是将顾客（customer）字段添加到中转对象里（编译、测试、提交）。

```

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;
}

```

我将performances字段也搬移过去，这样我就可以移除掉renderPlainText的invoice参数（编译、测试、提交）。

## 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;
}
```

现在，我希望“剧目名称”信息也从中转数据中获得。为此，需要使用`play`中的数据填充`aPerformance`对象（记得编译、测试、提交）。

```
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;
  }
}
```

现在我只是简单地返回了一个`aPerformance`对象的副本，但马上我就会往这条记录中添加新的数据。返回副本的原因是，我不想修改传给函数的参数，我总是尽量保持数据不可变（`immutable`）——可变的会很快变成烫手的山芋。

在不熟悉 JavaScript 的人看来，`result = Object.assign({}, aPerformance)`的写法可能十分奇怪。它返回的是一个浅副本。虽然我更希望有个函数来完成此功能，但这个用法已经约定俗成，如果我自己写个函数，在 JavaScript 程序员看来反而会格格不入。

现在我们已经有了安放`play`字段的地方，可以把数据放进去。我需要为`playFor`和`statement`函数应用搬移函数（198）（然后编译、测试、提交）。

## function statement...

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

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

然后替换renderPlainText中对playFor的所有引用点，让它们使用新数据（编译、测试、提交）。

## 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;

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 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;
}
```

接着我使用类似的手法搬移amountFor函数（编译、测试、提交）。



## function statement...

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

## 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(totalAmount())}\n`;  
result += `You earned ${totalVolumeCredits()} credits\n`;  
return result;  
  
function totalAmount() {  
  let result = 0;  
  for (let perf of data.performances) {  
    result += perf.amount;  
  }  
  return result;  
}
```

接下来搬移观众量积分的计算（编译、测试、提交）。

## 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，我将两个计算总数的函数搬移到`statement`函数中。

## 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;
```

尽管我可以修改函数体，让这些计算总数的函数直接使用`statementData`变量（反正它在作用域内），但我更喜欢显式地传入函数参数。

等到搬移完成，编译、测试、提交也做完，我便忍不住以管道取代循环（231）对几个地方进行重构。

## 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);  
}
```

现在我可以把第一阶段的代码提炼到一个独立的函数里了（编译、测试、提交）。

## 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;
}
```

由于两个阶段已经彻底分离，我干脆把它搬移到另一个文件里去（并且修改了返回结果的变量名，与我一贯的编码风格保持一致）。

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

最后再做一次编译、测试、提交，接下来，要编写一个HTML版本的对账单就很简单了。

## 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) {...}
```

（我把usd函数也搬移到顶层作用域中，以便renderHtml也能访问它。）

## 1.7 进展：分离到两个文件（和两个阶段）

现在正是停下来重新回顾一下代码的好时机，思考一下重构的进展。现在我有两个代码文件。

### statement.js

```
import createStatementData from './createStatementData.js';
function statement (invoice, plays) {
  return renderPlainText(createStatementData(invoice, plays));
}
function renderPlainText(data, plays) {
  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 = `

# 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


```
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);
}

```

代码行数由我开始重构时的44行增加到了70行（不算htmlStatement），这主要是将代码抽取到函数里带来的额外包装成本。虽然代码的行数增加了，但重构也带来了代码可读性的提高。额外的包装将混杂的逻辑分解成可辨别的部分，分离了详单的计算逻辑与样式。这种模块化使我更容易辨别代码的不同部分，了解它们的协作关系。虽说言以简为贵，但可演化的软件却以明确为贵。通过增强代码的模块化，我可以轻易地添加HTML版本的代码，而无须重复计算部分的逻辑。

 编程时，需要遵循营地法则：保证你离开时的代码库一定比来时更健康。

---

其实打印逻辑还可以进一步简化，但当前的代码也够用了。我经常需要在所有可做的重构与添加新特性之间寻找平衡。在当今业界，大多数人面临同样的选择时，似乎多以延缓重构而告终——当然这也是一种选择。我的观点则与营地法则无异：保证离开时的代码库一定比你来时更加健康。完美的境界很难达到，但应该时时都勤加拂拭。



## 1.8 按类型重组计算过程

接下来我将注意力集中到下一个特性改动：支持更多类型的戏剧，以及支持它们各自的价格计算和观众量积分计算。对于现在的结构，我只需要在计算函数里添加分支逻辑即可。`amountFor`函数清楚地体现了，戏剧类型在计算分支的选择上起着关键的作用——但这样的分支逻辑很容易随代码堆积而腐坏，除非编程语言提供了更基础的编程语言元素来防止代码堆积。

要为程序引入结构、显式地表达出“计算逻辑的差异是由类型代码确定”有许多途径，不过最自然的解决办法还是使用面向对象世界里的一个经典特性——类型多态。传统的面向对象特性在JavaScript世界一直备受争议，但新的ECMAScript 2015规范有意为类和多态引入了一个相当实用的语法糖。这说明，在合适的场景下使用面向对象是合理的——显然我们这个就是一个合适的使用场景。

我的设想是先建立一个继承体系，它有“喜剧”（`comedy`）和“悲剧”（`tragedy`）两个子类，子类各自包含独立的计算逻辑。调用者通过调用一个多态的`amount`函数，让语言帮你分发到不同的子类的计算过程中。`volumeCredits`函数的处理也是如法炮制。为此我需要用多种重构方法，其中最核心的一招是以多态取代条件表达式（272），将多个同样的类型码分支用多态取代。但在施展以多态取代条件表达式（272）之前，我得先创建一个基本的继承结构。我需要先创建一个类，并将价格计算函数和观众量积分计算函数放进去。

我先从检查计算代码开始。（之前的重构带来的一大好处是，现在我大可以忽略那些格式化代码，只要不改变中转数据结构就行。我可以进一步添加测试来保证中转数据结构不会被意外修改。）

### `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 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);
}

```

## 创建演出计算器

`enrichPerformance`函数是关键所在，因为正是它用每场演出的数据来填充中转数据结构。目前它直接调用了计算价格和观众量积分的函数，我需要创建一个类，通过这个类来调用这些函数。由于这个类存放了与每场演出相关数据的计算函数，于是我把它称为演出计算器（`performance calculator`）。

### function createStatementData...

```

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

```

```
}
```

## 顶层作用域...

```
class PerformanceCalculator {  
  constructor(aPerformance) {  
    this.performance = aPerformance;  
  }  
}
```

到目前为止，这个新对象还没做什么事。我希望将函数行为搬移进来，这可以从最容易搬移的东西——`play`字段开始。严格来讲，我不需要搬移这个字段，因为它并未体现出多态性，但这样可以把所有数据转换集中到一处地方，保证了代码的一致性和清晰度。

为此，我将使用改变函数声明（124）手法将`performance`的`play`字段传给计算器。

## function createStatementData...

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

## class PerformanceCalculator...

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

（以下行文中我将不再特别提及“编译、测试、提交”循环，我猜你也已经读得有些厌烦了。但我仍会不断重复这个循环。的确，有时我也会厌烦，直到错误又跳出来咬我一下，我才又学会进入小步的节奏。）

## 将函数搬移进计算器

我要搬移的下一块逻辑，对计算一场演出的价格（`amount`）来说就尤为重要了。在调整嵌套函数的层级时，我经常将函数挪来挪去，但接下来需要改动到更深入的函数上下文，因此我将小心使用搬移函数（198）来重构它。首先，将`amount`函数的逻辑复制一份到新的上下文中，也就是`PerformanceCalculator`类中。然后微调一下代码，将`aPerformance`改为`this.performance`，将`playFor(aPerformance)`改为`this.play`，使代码适应这个新家。

## 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;
}
```

搬移完成后可以编译一下，看看是否有编译错误。我在本地开发环境运行代码时，编译会自动发生，我实际需要做的只是运行一下Babel。编译能帮我发现新函数中潜在的语法错误，语法之外的就帮不上什么忙了。尽管如此，这一步还是很有用。

使新函数适应新家后，我会将原来的函数改造成一个委托函数，让它直接调用新函数。

## function createStatementData...

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

现在，我可以执行一次编译、测试、提交，确保代码搬到新家后也能如常工作。之后，我应用内联函数（115），让引用点直接调用新函数（然后编译、测

试、提交）。

## 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;
}
```

搬移观众量积分计算也遵循同样的流程。

## 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;
}
```

## 使演出计算器表现出多态性

我已将全部计算逻辑搬移到一个类中，是时候将它多态化了。第一步是应用以子类取代类型码（362）引入子类，弃用类型代码。为此，我需要为演出计算器创建子类，并在createStatementData中获取对应的子类。要得到正确的子类，我需要将构造函数调用替换为一个普通的函数调用，因为JavaScript的构造函数里无法返回子类。于是我使用以工厂函数取代构造函数（334）。

## 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);
}
```

改造成普通函数后，我就可以在里面创建演出计算器的子类，然后由创建函数决定返回哪一个子类的实例。

## 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 {
}
```

准备好实现多态的类结构后，我就可以继续使用以多态取代条件表达式（272）手法了。

我先从悲剧的价格计算逻辑开始搬移。

## class TragedyCalculator...

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

虽说子类有了这个方法已足以覆盖超类对应的条件分支，但要是你也和我一样偏执，你也许还想在超类的分支上抛一个异常。

### class PerformanceCalculator...

```
get amount() {
  let result = 0;
  switch (this.play.type) {
    case "tragedy":
      throw 'bad thing';
    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;
}
```

虽然我也可以直接删掉处理悲剧的分支，将错误留给默认分支去抛出，但我更喜欢显式地抛出异常——何况这行代码只能再活个几分钟了（这也是我直接抛出一个字符串而不用更好的错误对象的原因）。

再次进行编译、测试、提交。之后，将处理喜剧类型的分支也下移到子类中去。

### 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;
}
```

理论上讲，我可以将超类的`amount`方法一并移除了，反正它也不应再被调用到。但不删它，给未来的自己留点纪念品也是极好的，顺便可以提醒后来者记得实现这个函数。

### class PerformanceCalculator...

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

下一个要替换的条件表达式是观众量积分的计算。我回顾了一下前面关于未来戏剧类型的讨论，发现大多数剧类在计算积分时都会检查观众数是否达到30，仅一小部分品类有所不同。因此，将更为通用的逻辑放到超类作为默认条件，出现特殊场景时按需覆盖它，听起来十分合理。于是我将一部分喜剧的逻辑下移到子类。

### 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 进展：使用多态计算器来提供数据

又到了观摩代码的时刻，让我们来看看，为计算器引入多态会对代码库有什么影响。

### 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);
  }
}

```

代码量仍然有所增加，因为我再次整理了代码结构。新结构带来的好处是，不同戏剧种类的计算各自集中到了一处地方。如果大多数修改都涉及特定类型的计算，像这样按类型进行分离就很有意义。当添加新剧种时，只需要添加一个子类，并在创建函数中返回它。

这个示例还揭示了一些关于此类继承方案何时适用的洞见。上面我将条件分支的查找从两个不同的函数（`amountFor`和`volumeCreditsFor`）搬移到一个集中的构造函数`createPerformanceCalculator`中。有越多的函数依赖于同一套类型进行多态，这种继承方案就越有益处。

除了这样设计，还有另一种可能的方案，那就是让`createStatementData`返回计算器实例本身，而非自己拿到计算器来填充中转数据结构。JavaScript的类设计有不少好特性，例如，取值函数用起来就像普通的数据存取。我在考量是“直接返回实例本身”还是“返回计算好的中转数据”时，主要看数据的使用者是谁。在这个例子中，我更想通过中转数据结构来展示如何以此隐藏计算器背后的多态设计。

## 1.10 结语

这是一个简单的例子，但我希望它能让你对“重构怎么做”有一点感觉。例中我已经示范了数种重构手法，包括提炼函数（106）、内联变量（123）、搬移函数（198）和以多态取代条件表达式（272）等。

本章的重构有3个较为重要的节点，分别是：将原函数分解成一组嵌套的函数、应用拆分阶段（154）分离计算逻辑与输出格式化逻辑，以及为计算器引入多态性来处理计算逻辑。每一步都给代码添加了更多的结构，以便我能更好地表达代码的意图。

一般来说，重构早期的主要动力是尝试理解代码如何工作。通常你需要先通读代码，找到一些感觉，然后再通过重构将这些感觉从脑海里搬回到代码中。清晰的代码更容易理解，使你能够发现更深层次的设计问题，从而形成积极正向的反馈环。当然，这个示例仍有值得改进的地方，但现在测试仍能全部通过，代码相比初见时已经有了巨大的改善，所以我已经可以满足了。

我谈论的是如何改善代码，但什么样的代码才算好代码，程序员们有很多争论。我偏爱小的、命名良好的函数，也知道有些人反对这个观点。如果我们说这只关乎美学，只是各花入各眼，没有好坏高低之分，那除了诉诸个人品味，就没有任何客观事实依据了。但我坚信，这不仅关乎个人品味，而且是有客观标准的。我认为，好代码的检验标准就是人们是否能轻而易举地修改它。好代码应该直截了当：有人需要修改代码时，他们应能轻易找到修改点，应该能快速做出更改，而不易引入其他错误。一个健康的代码库能够最大限度地提升我们的生产力，支持我们更快、更低成本地为用户添加新特性。为了保持代码库的健康，就需要时刻留意现状与理想之间的差距，然后通过重构不断接近这个理想。



好代码的检验标准就是人们是否能轻而易举地修改它。

这个示例告诉我们最重要的一点就是重构的节奏感。无论何时，当我向人们展示我如何重构时，无人不讶异于我的步子之小，并且每一步都保证代码处于编译通过和测试通过的可工作状态。20年前，当Kent Beck在底特律的一家宾馆里向我展示同样的手法时，我也报以同样的震撼。开展高效有序的重构，关键的心得是：小的步子可以更快前进，请保持代码永远处于可工作状态，小步修改累积起来也能大大改善系统的设计。这几点请君牢记，其余的我已无需多言。