第 9 章 简化条件表达式

条件逻辑有可能十分复杂，因此本章提供一些重构手法，专门用来简化它们。其中一项核心重构就是 Decompose Conditional(238),可将一个复杂的条件逻辑分成若干小块。这项重构很重要，因为它使得“分支逻辑”和“操作细节”分离。

本章的其余重构手法可用以处理另一些重要问题: 如果你发现代码中的多处测试有相同结果, 应该实施 Consolidate Conditional Expression(240);如果条件代码中有任何重复，可以运用 Consolidate Duplicate Conditional Fragments(243)将重复成分去掉。

如果程序开发者坚持“单一出口”原则，那么为让条件表达式也遵循这一原则，他往往会在其中加入控制标记。我并不特别在意“一个函数一个出口”的教条，所以我使用 Replace Nested Conditional with Guard Clauses(250)标示出那些特殊情况，并使用 Remove Control Flag(245)去除那些讨厌的控制标记。

较之于过程化程序而言，面向对象程序的条件表达式通常比较少，这是因为很多条件行为都被多态机制处理掉了。多态之所以更好，是因为调用者无需了解条件行为的细节,因此条件的扩展更为容易。所以面向对象程序中很少出现 switch 语句。一旦出现，就应该考虑运用 Replace Conditional with Polymorphism(255)将它替换为多态。

多态还有一种十分有用但鲜为人知的用途: 通过 Introduce Null Object(260)去除对于 null 值的检验。

9.1 Decompose Conditional(分解条件表达式)

你有一个复杂的条件(if-then-else)语句。

从 if、then、else 三个段落中分别提炼出独立函数。

```
if (date.before(SUMMER_START) || date.after(SUMMER_END))
    charge = quantity * _winterRate + _winterServiceCharge;
else charge = quantity * _summerRate;
```

⬇↓

```
if (notSummer(date))
    charge = winterCharge(quantity);
else charge = summerCharge(quantity);
```

动机

程序之中，复杂的条件逻辑是最常导致复杂度上升的地点之一。你必须编写代码来检查不同的条件分支、根据不同的分支做不同的事，然后，你很快就会得到一个相当长的函数。大型函数自身就会使代码的可读性下降，而条件逻辑则会使代码更难阅读。在带有复杂条件逻辑的函数中，代码(包括检查条件分支的代码和真正实现功能的代码)会告诉你发生的事，但常常让你弄不清楚为什么会发生这样的事，这就说明代码的可读性的确大大降低了。

和任何大块头代码一样，你可以将它分解为多个独立函数，根据每个小块代码的用途,为分解而得的新函数命名,并将原函数中对应的代码改为调用新建函数,从而更清楚地表达自己的意图。对于条件逻辑，将每个分支条件分解成新函数还可以给你带来更多好处:可以突出条件逻辑，更清楚地表明每个分支的作用，并且突出每个分支的原因。

做法

- 将 if 段落提炼出来，构成一个独立函数。
- 将 then 段落和 else 段落都提炼出来,各自构成一个独立函数。
- 如果发现嵌套的条件逻辑，我通常会先观察是否可以使用 Replace Nested Conditional with Guard Clauses(250)。如果不行,才开始分解其中的每个条件。

范例

假设我要计算购买某样商品的总价(总价=数量 x 单价),而这个商品在冬季和夏季的单价是不同的:

```
if (date.before(SUMMER_START) || date.after(SUMMER_END))
    charge = quantity * _winterRate + _winterServiceCharge;
else charge = quantity * _summerRate;
```

我把每个分支的判断条件都提炼到一个独立函数中，如下所示:

```
if (notSummer(date))
    charge = winterCharge(quantity);
else charge = summerCharge(quantity);
private boolean notSummer(Date date) {
    return date.before(SUMMER_START) || date.after(SUMMER_END);
}
private double summerCharge(int quantity) {
    return quantity * _summerRate;
}
private double winterCharge(int quantity) {
    return quantity * _winterRate + _winterServiceCharge;
}
```

通过这段代码可以看出整个重构带来的清晰性。实际工作中,我会逐步进行每一次提炼，并在每次提炼之后编译并测试。

像这样的情况下，许多程序员都不会去提炼分支条件。因为这些分支条件往往非常短，看上去似乎没有提炼的必要。但是，尽管这些条件往往很短，在代码意图和代码自身之间往往存在不小的差距。哪怕在上面这样一个小小例子中，notSummer(date)这个语句也能够比原本的代码更好地表达自己的用途。对于原来的代码，我必须看着它，想一想，才能说出其作用。当然，在这个简单的例子中，这并不困难。不过,即使如此,提炼出来的函数可读性也更高一些一它看上去就像一段注释那样清楚而明白。

9.2 Consolidate Conditional Expression(合并条件表达式)

你有一系列条件测试，都得到相同结果。

将这些测试合并为一个条件表达式,并将这个条件表达式提炼成为一个独立函数。

```
double disabilityAmount() {
        if (_seniority < 2) return 0;
        if (_monthsDisabled > 12) return 0;
        if (_isPartTime) return 0;
        // compute the disability amount
```

⬇↓

```
double disabilityAmount() {
    if (isNotEligableForDisability()) return 0;
    // compute the disability amount
```

动机

有时你会发现这样一串条件检查:检查条件各不相同，最终行为却一致。如果发现这种情况，就应该使用“逻辑或”和“逻辑与”将它们合并为一个条件表达式。

之所以要合并条件代码,有两个重要原因。首先,合并后的条件代码会告诉你“实际上只有一次条件检查,只不过有多个并列条件需要检查而已”,从而使这一次检查的用意更清晰。当然,合并前和合并后的代码有着相同的效果,但原先代码传达出的信息却是“这里有一些各自独立的条件测试，它们只是恰好同时发生”。其次，这项重构往往可以为你使用 Extract Method(110)做好准备。将检查条件提炼成一个独立函数对于厘清代码意义非常有用，因为它把描述“做什么”的语句换成了“为什么这样做”。

条件语句的合并理由也同时指出了不要合并的理由:如果你认为这些检查的确彼此独立，的确不应该被视为同一次检查，那么就不要使用本项重构。因为在这种情况下，你的代码已经清楚表达出自己的意义。

做法

- 确定这些条件语句都没有副作用。

→ 如果条件表达式有副作用，你就不能使用本项重构。

- 使用适当的逻辑操作符,将一系列相关条件表达式合并为一个。
- 编译,测试。
- 对合并后的条件表达式实施 Extract Method(110)。

范例: 使用逻辑或 (||)

请看下列代码:

```
double disabilityAmount() {
    if (_seniority < 2) return 0;
    if (_monthsDisabled > 12) return 0;
```

```
    if (_isPartTime) return 0;
    // compute the disability amount
    ...
```

在这段代码中，我们看到一连串的条件检查，它们都做同一件事。对于这样的代码，上述条件检查等价于一个以逻辑或连接起来的语句:

```
double disabilityAmount() {
    if ((_seniority < 2) || (_monthsDisabled > 12) || (_isPartTime)) re
turn 0;
    // compute the disability amount
    ...
```

现在，我可以观察这个新的条件表达式，并运用 Extract Method(110)将它提炼成一个独立函数,以函数名称表达该语句所检查的条件:

```
double disabilityAmount() {
    if (isNotEligibleForDisability()) return 0;
    // compute the disability amount
    ...
}

boolean isNotEligibleForDisability() {
    return ((_seniority < 2) || (_monthsDisabled > 12) || (_isPartTim
e));
}
```

其中 `isNotEligibleForDisability()` 是重点(加粗了的)。

范例: 使用逻辑与 (&&)

上述实例展示了逻辑或的用法。下列代码展示逻辑与的用法:

```
if (onVacation())
    if (lengthOfService() > 10)
        return 1;
return 0.5;
```

这段代码可以变成:

```
if (onVacation() && lengthOfService() > 10) return 1;
else return 0.5;
```

你可能还会发现，某些情况下需要同时使用逻辑或、逻辑与和逻辑非，最终得到的条件表达式可能很复杂，所以我会先使用 Extract Method(110)将表达式的一部分提炼出来，从而使整个表达式变得简单一些。

如果我所观察的部分只是对条件进行检查并返回一个值，就可以使用三元操作符将这一部分变成一条 return 语句。因此，下列代码:

```
if (onVacation() && lengthOfService() > 10) return 1;
else return 0.5;
```

就变成了:

```
return (onVacation() && lengthOfService() > 10) ? 1 : 0.5;
```

9.3 Consolidate Duplicate Conditional Fragments (合并重复的条件片段)

在条件表达式的每个分支上有着相同的一段代码。

```
if (isSpecialDeal()) {
    total = price * 0.95;
    send();
}
else {
    total = price * 0.98;
    send();
}
```

将这段重复代码搬移到条件表达式之外。

```
if (isSpecialDeal())
    total = price * 0.95;
else
```

```
    total = price * 0.98;
send();
```

动机

有时你会发现，一组条件表达式的所有分支都执行了相同的某段代码。如果是这样，你就应该将这段代码搬移到条件表达式外面。这样，代码才能更清楚地表明哪些东西随条件的变化而变化、哪些东西保持不变。

做法

- 鉴别出“执行方式不随条件变化而变化”的代码。
- 如果这些共通代码位于条件表达式起始处，就将它移到条件表达式之前。
- 如果这些共通代码位于条件表达式尾端,就将它移到条件表达式之后。
- 如果这些共通代码位于条件表达式中段,就需要观察共通代码之前或之后的代码是否改变了什么东西。如果的确有所改变,应该首先将共通代码向前或向后移动,移至条件表达式的起始处或尾端,再以前面所说的办法来处理。
- 如果共通代码不止一条语句,应该首先使用 Extract Method(110)将共通代码提炼到一个独立函数中，再以前面所说的办法来处理。

范例

你可能遇到这样的代码:

```
if (isSpecialDeal()) {
    total = price * 0.95;
    send();
}
else {
    total = price * 0.98;
    send();
}
```

由于条件表达式的两个分支都执行了 send()函数，所以我应该将 send()移到条件表达式的外围:

```
if (isSpecialDeal())
    total = price * 0.95;
else
    total = price * 0.98;
send();
```

我们也可以使用同样的手法来对待异常。如果在 try 区段内可能引发异常的语句之后，以及所有 catch 区段之内，都重复执行了同一段代码，就可以将这段重复代码移到 final 区段 (finally block)。

9.4 Remove Control Flag(移除控制标记)

在一系列布尔表达式中，某个变量带有“控制标记”(control flag)的作用。

以 break 语句或 return 语句取代控制标记。

动机

在一系列条件表达式中，你常常会看到用以判断何时停止条件检查的控制标记:

```
set done to false
while not done
  if (condition)
      do something
      set done to true
  next step of loop
```

这样的控制标记带来的麻烦超过了它所带来的便利。人们之所以会使用这样的控制标记，因为结构化编程原则告诉他们:每个子程序只能有一个入口和一个出口。我赞同“单一入口”原则(而且现代编程语言也强迫我们这样做)，但是“单一出口”原则会让你在代码中加入讨厌的控制标记，大大降低条件表达式的可读性。这就是编程语言提供 break 语句和 continue 语句的原因:用它们跳出复杂的条件语句。去掉控制标记所产生的效果往往让你大吃一惊:条件语句真正的用途会清晰得多。

做法

对控制标记的处理，最显而易见的办法就是使用 Java 提供的 break 语句或 continue 语句。

- 找出让你跳出这段逻辑的控制标记值。
- 找出对标记变量赋值的语句，代以恰当的 break 语句或 continue 语句。
- 每次替换后,编译并测试。

在未能提供 break 和 continue 语句的编程语言中，可以使用下述办法。

- 运用 Extract Method(110),将整段逻辑提炼到一个独立函数中。
- 找出让你跳出这段逻辑的控制标记值。
- 找出对标记变量赋值的语句,代以恰当的 return 语句。
- 每次替换后,编译并测试。

即使在支持 break 和 continue 语句的编程语言中,我通常也优先考虑上述第二种方案。因为 return 语句可以非常清楚地表示:不再执行该函数中的其他任何代码。如果还有这一类代码，你早晚需要将这段代码提炼出来。

请注意标记变量是否会影响这段逻辑的最后结果。如果有影响，使用 break 语句之后还得保留控制标记值。如果你已经将这段逻辑提炼成一个独立函数，也可以将控制标记值放在 return 语句中返回。

范例: 以 break 取代简单的控制标记

下列函数用来检查一系列人名之中是否包含两个可疑人物的名字(这两个人的名字硬编码于代码中):

```
void checkSecurity(String[] people) {
    boolean found = false;
    for (int i = 0; i < people.length; i++) {
        if (! found) {
```

```
            if (people[i].equals ("Don")){
              sendAlert();
              found = true;
            }
            if (people[i].equals ("John")){
              sendAlert();
              found = true;
            }
        }
    }
}
```

这种情况下很容易找出控制标记: 当变量 found 被赋值为 true 时，搜索就结束。我可以逐一引入 break 语句:

```
void checkSecurity(String[] people) {
    boolean found = false;
    for (int i = 0; i < people.length; i++) {
        if (! found) {
            if (people[i].equals ("Don")){
              sendAlert();
              break;
            }
            if (people[i].equals ("John")){
              sendAlert();
              found = true;
            }
        }
    }
```

注意: break 的地方 (加粗)

直到替换掉所有对 found 变量赋值的语句:

```
void checkSecurity(String[] people) {
    boolean found = false;
    for (int i = 0; i < people.length; i++) {
        if (! found) {
            if (people[i].equals ("Don")){
              sendAlert();
              break;
            }
            if (people[i].equals ("John")){
```

```
                sendAlert();
                break;
            }
        }
    }
}
```

注意: 上面代码的第 2 个 break 的地方 (加粗)

然后就可以把所有对控制标记的引用都去掉:

```
void checkSecurity(String[] people) {
      for (int i = 0; i < people.length; i++) {
        if (people[i].equals ("Don")){
           sendAlert();
           break;
        }
        if (people[i].equals ("John")){
           sendAlert();
           break;
        }
    }
}
```

范例: 以 return 返回控制标记

本项重构的另一种形式将使用 return 语句。为了阐述这种用法, 我把前面的例子稍加修改，以控制标记记录搜索结果:

```
void checkSecurity(String[] people) {
    String found = "";
    for (int i = 0; i < people.length; i++) {
        if (found.equals("")) {
           if (people[i].equals ("Don")){
              sendAlert();
              found = "Don";
           }
           if (people[i].equals ("John")){
              sendAlert();
```

```
                found = "John";
            }
        }
    }
    someLaterCode(found);
}
```

在这里，变量 found 做了两件事：它既是控制标记，也是运算结果。遇到这种情况，我喜欢先把计算 found 变量的代码提炼到一个独立函数中：

```
void checkSecurity(String[] people) {
    String found = foundMiscreant(people);
    someLaterCode(found);
}

String foundMiscreant(String[] people){
    String found = "";
    for (int i = 0; i < people.length; i++) {
        if (found.equals("")) {
            if (people[i].equals ("Don")){
              sendAlert();
              found = "Don";
            }
            if (people[i].equals ("John")){
              sendAlert();
              found = "John";
            }
        }
    }
    return found;
}
```

然后以 return 语句取代控制标记：

```
String foundMiscreant(String[] people){
    String found = "";
    for (int i = 0; i < people.length; i++) {
        if (found.equals("")) {
            if (people[i].equals ("Don")){
               sendAlert();
               return "Don";
            }
            if (people[i].equals ("John")){
```

```
                sendAlert();
                found = "John";
            }
        }
    }
    return found;
}
```

注意: **return "Don";** 是加粗的部分。

最后完全去掉控制标记:

```
String foundMiscreant(String[] people){
    for (int i = 0; i < people.length; i++) {
      if (people[i].equals ("Don")){
            sendAlert();
            return "Don";
        }
        if (people[i].equals ("John")){
            sendAlert();
            return "John";
        }
    }
    return "";
}
```

即使不需要返回某值，也可以用 return 语句来取代控制标记。这时候你只需要一个空的 return 语句就行了。

当然,如果以此办法去处理带有副作用的函数,会有一些问题。所以我需要先以 Separate Query from Modifier(279)将函数副作用分离出去。稍后你会看到这方面的例子。

9.5 Replace Nested Conditional with Guard Clauses

(以卫语句取代嵌套条件表达式)

函数中的条件逻辑使人难以看清正常的执行路径。

使用卫语句表现所有特殊情况。

```
double getPayAmount() {
  double result;
  if (_isDead) result = deadAmount();
  else {
      if (_isSeparated) result = separatedAmount();
      else {
          if (_isRetired) result = retiredAmount();
          else result = normalPayAmount();
      };
  }
return result;
};
```

↓

```
double getPayAmount() {
  if (_isDead) return deadAmount();
  if (_isSeparated) return separatedAmount();
  if (_isRetired) return retiredAmount();
  return normalPayAmount();
};
```

动机

根据我的经验，条件表达式通常有两种表现形式。第一种形式是: 所有分支都属于正常行为。第二种形式则是:条件表达式提供的答案中只有一种是正常行为，其他都是不常见的情况。

这两类条件表达式有不同的用途，这一点应该通过代码表现出来。如果两条分支都是正常行为，就应该使用形如 if...else...的条件表达式;如果某个条件极其罕见，就应该单独检查该条件，并在该条件为真时立刻从函数中返回。这样的单独检查常常被称为“卫语” (guardclauses)[Beck]。

Replace Nested Conditional with Guard Clauses(250)的精髓就是: 给某一条分支以特别的重视。如果使用 if-then-else 结构，你对 if 分支和 else 分支的重视是同等的。这样的代码结构传递给阅读者的消息就是: 各个分支有同样的重要性。卫语句就不同了，它告诉阅读者:“这种情况很罕见，如果它真地发生了，请做一些必要的整理工作，然后退出。”

“每个函数只能有一个入口和一个出口”的观念，根深蒂固于某些程序员的脑海里。我发现，当我处理他们编写的代码时，经常需要使用 Replace Nested Conditional with Guard Clauses(250)。现今的编程语言都会强制保证每个函数只有一个入口，至于“单一出口”规则，其实不是那么有用。在我看来，保持代码清晰才是最关键的:如果单一出口能使这个函数更清楚易读，那么就使用单一出口;否则就不必这么做。

做法

- 对于每个检查，放进一个卫语句。

→ 卫语句要不就从函数中返回，要不就抛出一个异常。

- 每次将条件检查替换成卫语句后，编译并测试。

→ 如果所有卫语句都导致相同结果，请使用 Consolidate Conditional Expressions(240)。

范例

想象一个薪册系统，其中以特殊规则处理死亡员工、驻外员工、退休员工的薪资。这些情况不常有，但的确偶而会出现。

假设我在这个系统中看到下列代码:

```
double getPayAmount() {
    double result;
    if (_isDead) result = deadAmount();
      else {
          if (_isSeparated) result = separatedAmount();
          else {
              if (_isRetired) result = retiredAmount();
              else result = normalPayAmount();
          };
      }
    return result;
};
```

在这段代码中,非正常情况的检查掩盖了正常情况的检查,所以应该用卫语句来取代这些检查,以提高程序清晰度。我可以逐一引入卫语句。让我们从最上面的条件检查动作开始:

```
double getPayAmount() {
  double result;
  if (_isDead) return deadAmount();
  if (_isSeparated) result = separatedAmount();
  else {
      if (_isRetired) result = retiredAmount();
      else result = normalPayAmount();
  };
  return result;
};
```

注意: **if (_isDead) return deadAmount();** 是加粗的部分。

然后，继续下去，仍然一次替换一个检查动作:

```
double getPayAmount() {
  double result;
  if (_isDead) return deadAmount();
  if (_isSeparated) return separatedAmount();
  if (_isRetired) result = retiredAmount();
  else result = normalPayAmount();
  return result;
};
```

注意: **if (_isSeparated) return separatedAmount();** 是加粗的部分。

```
double getPayAmount() {
  double result;
  if (_isDead) return deadAmount();
  if (_isSeparated) return separatedAmount();
  if (_isRetired) return retiredAmount();
  result = normalPayAmount();
  return result;
};
```

注意: **if (_isRetired) return retiredAmount();** 是加粗的部分。

此时, result 变量已经没有价值了, 所以我把它删掉:

```
double getPayAmount() {
  if (_isDead) return deadAmount();
  if (_isSeparated) return separatedAmount();
  if (_isRetired) return retiredAmount();
  return normalPayAmount();
};
```

注意: **return;** 是加粗的部分。

嵌套条件代码往往由那些深信“每个函数只能有一个出口”的程序员写出。我发现那条规则实在有点太简单粗暴了。如果对函数剩余部分不再有兴趣，当然应该立刻退出。引导阅读者去看一个没有用的 else 区段，只会妨碍他们的理解。

范例: 将条件反转

Invert If

审阅本书初稿时，Joshua Kerievsky 指出: 你常常可以将条件表达式反转，从而实现 Replace Nested Conditional with Guard Clauses(250)。为了拯救我可怜的想象力，他还好心帮我想了个例子:

```
public double getAdjustedCapital() {
  double result = 0.0;
  if (_capital > 0.0) {
    if (_intRate > 0.0 && _duration > 0.0) {
     result = (_income / _duration) * ADJ_FACTOR;
    }
  }
  return result;
}
```

同样地，我逐一进行替换。不过这次在插入卫语句时，我需要将相应的条件反转过来:

```
public double getAdjustedCapital() {
    double result = 0.0;
```

```
    if (_capital <= 0.0) return result;
    if (_intRate > 0.0 && _duration > 0.0) {
      result = (_income / _duration) * ADJ_FACTOR;
    }
    return result;
}
```

注意: **if (_capital <= 0.0) return result;** 是加粗的部分。

下一个条件稍微复杂一点，所以我分两步进行逆反。首先加入一个逻辑非操作:

```
public double getAdjustedCapital() {
    double result = 0.0;
    if (_capital <= 0.0) return result;
    if (!(_intRate > 0.0 && _duration > 0.0)) return result;
    result = (_income / _duration) * ADJ_FACTOR;
    return result;
}
```

注意: **if (!(_intRate > 0.0 && _duration > 0.0)) return result;** 是加粗的部分。

但是在这样的条件表达式中留下一个逻辑非，会把我的脑袋拧成一团乱麻，所以我把它简化成下面这样:

```
public double getAdjustedCapital() {
    double result = 0.0;
    if (_capital <= 0.0) return result;
    if (_intRate <= 0.0 || _duration <= 0.0) return result;
    result = (_income / _duration) * ADJ_FACTOR;
    return result;
}
```

注意: **if (_intRate <= 0.0 || _duration <= 0.0) return result;** 是加粗的部分。

这时候，我比较喜欢在卫语句内返回一个明确值，因为这样我可以一目了然地看到卫语句返回的失败结果。此外，这种时候我也会考虑使用 Replace Magic Number with Symbolic Constant(204)。

```
public double getAdjustedCapital() {
   double result = 0.0;
    if (_capital <= 0.0) return 0.0;
    if (_intRate <= 0.0 || _duration <= 0.0) return 0.0;
    result = (_income / _duration) * ADJ_FACTOR;
    return result;
}
```

注意: 两处 **return 0.0;** 是加粗的部分。

完成替换之后，我同样可以将临时变量移除:

```
public double getAdjustedCapital() {
    if (_capital <= 0.0) return 0.0;
    if (_intRate <= 0.0 || _duration <= 0.0) return 0.0;
    return (_income / _duration) * ADJ_FACTOR;
}
```

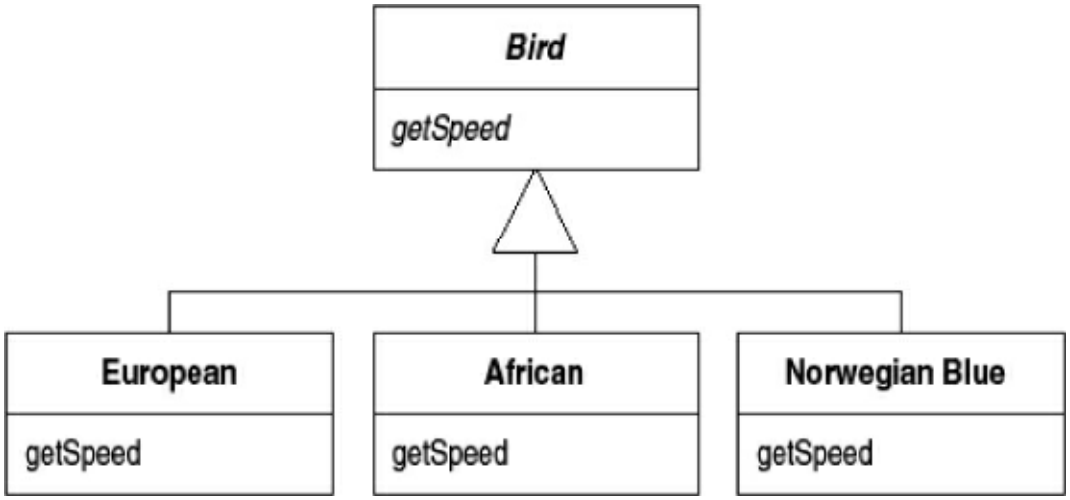
9.6 Replace Conditional with Polymorphism(以多态取代条件表达式)

你手上有个条件表达式，它根据对象类型的不同而选择不同的行为。

将这个条件表达式的每个分支放进一个子类内的覆写函数中，然后将原始函数声明为抽象函数。

```
double getSpeed() {
    switch (_type) {
        case EUROPEAN:
           return getBaseSpeed();
        case AFRICAN:
           return getBaseSpeed() - getLoadFactor() * _numberOfCoconuts;
        case NORWEGIAN_BLUE:
           return (_isNailed) ? 0 : getBaseSpeed(_voltage);
    }
    throw new RuntimeException ("Should be unreachable");
}
```

↓



动机

在面向对象术语中，听上去最高贵的词非“多态”莫属。多态最根本的好处就是:如果你需要根据对象的不同类型而采取不同的行为，多态使你不必编写明显的条件表达式。

正因为有了多态,所以你会发现:“类型码的 switch 语句”以及“基于类型名称的 if-then-else 语句”在面向对象程序中很少出现。

多态能够给你带来很多好处。如果同一组条件表达式在程序许多地点出现,那么使用多态的收益是最大的。使用条件表达式时,如果你想添加一种新类型,就必须查找并更新所有条件表达式。但如果改用多态,只需建立一个新的子类,并在其中提供适当的函数就行了。类的用户不需要了解这个子类,这就大大降低了系统各部分之间的依赖,使系统升级更加容易。

做法

使用 *Replace Conditional with Polymorphism(255)*之前，首先必须有一个继承结构。你可能已经通过先前的重构得到了这一结构。如果还没有,现在就需要建立它要建立继承结构,有两种选择: Replace Type Code with Subclasses(223) 和 Replace Type Code with State/Strategy(227)。

前一种做法比较简单,因此应该尽可能使用它。但如果你需要在对象创建好之后修改类型码,就不能使用继承手法,只能使用 State Strategy 模式。此外,如果由于其他原因,要重构的类已经有了子类,那么也得使用 State/Strategy。记住，如果若干 switch 语句针对的是同一个类型码，你只需针对这个类型码建立一个继承结构就行了。

现在,可以向条件表达式开战了。你的目标可能是 switch 语句, 也可能是 if 语句。

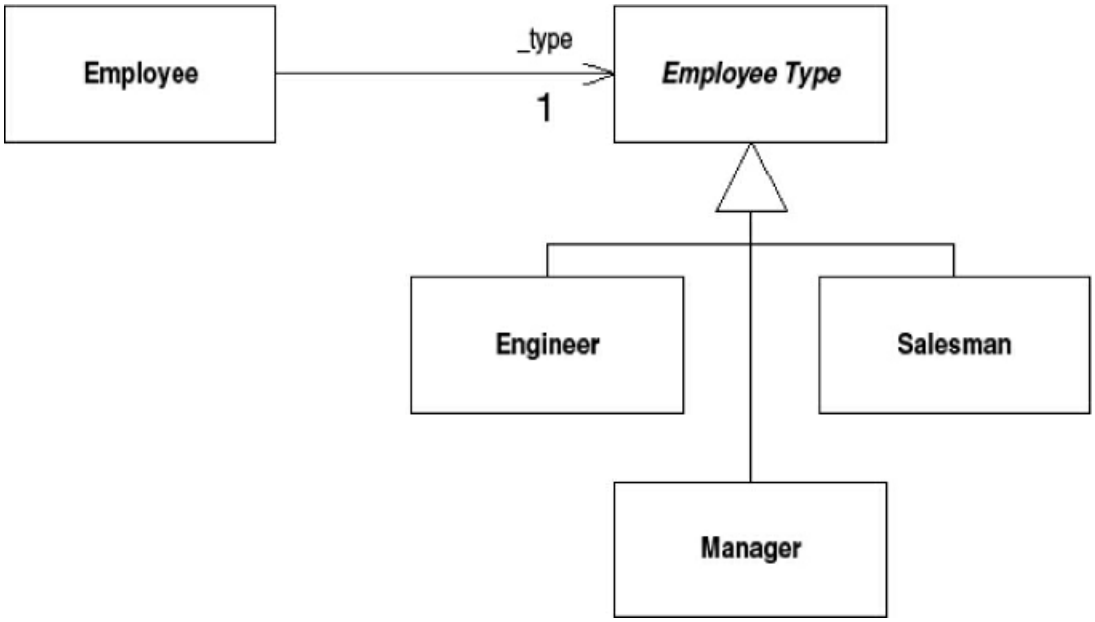
- 如果要处理的条件表达式是一个更大函数中的一部分,首先对条件表达式进行分析,然后使用 Extract Method(110)将它提炼到一个独立函数去。
- 如果有必要,使用 MoveMethod(142)将条件表达式放置到继承结构的顶端。
- 任选一个子类,在其中建立一个函数,使之覆写超类中容纳条件表达式的那个函数。将与该子类相关的条件表达式分支复制到新建函数中,并对它进行适当调整。

→ 为了顺利进行这一步骤,你可能需要将超类中的某些 private 字段声明为 protected。

- 编译，测试。
- 在超类中删掉条件表达式内被复制了的分支。
- 编译, 测试。
- 针对条件表达式的每个分支，重复上述过程，直到所有分支都被移到子类内的函数为止。
- 将超类之中容纳条件表达式的函数声明为抽象函数。

范例

请允许我继续使用“员工与薪资”这个简单而又乏味的例子。我的类是从 Replace Type Code with State/Strategy(227)那个例子中拿来的，因此示意图就如图 9-1 所示(如果想知道这个图是怎么得到的，请看第 8 章的范例)。




```
class Employee...
  int payAmount() {
      switch (getType()) {
              case EmployeeType.ENGINEER:
             return _monthlySalary;
          case EmployeeType.SALESMAN:
             return _monthlySalary + _commission;
          case EmployeeType.MANAGER:
             return _monthlySalary + _bonus;
          default:
             throw new RuntimeException("Incorrect Employee");
      }
  }

  int getType() {
      return _type.getTypeCode();
  }

  private EmployeeType _type;

  abstract class EmployeeType...
  abstract int getTypeCode();

  class Engineer extends EmployeeType...
      int getTypeCode() {
      return EmployeeType.ENGINEER;
  }
```

```
// ... and other subclasses
```

switch 语句已经被很好地提炼出来，因此我不必费劲再做一遍。不过我需要将它移到 EmployeeType 类，因为 EmployeeType 才是要被继承的类。

```
class EmployeeType...
  int payAmount(Employee emp) {
      switch (getTypeCode()) {
          case ENGINEER:
             return emp.getMonthlySalary();
          case SALESMAN:
             return emp.getMonthlySalary() + emp.getCommission();
          case MANAGER:
             return emp.getMonthlySalary() + emp.getBonus();
          default:
                  throw new RuntimeException("Incorrect Employee");
      }
  }
```

由于我需要 Employee 的数据，所以需要将 Employee 对象作为参数传递给 payAmount()。这些数据中的一部分也许可以移到 EmployeeType 来，但那是另一项重构需要关心的问题了。

调整代码，使之通过编译，然后我修改 Employee 中的 payAmount()函数，令它委托 EmployeeType:

```
class Employee...
  int payAmount() {
      return _type.payAmount(this);
  }
```

现在，我可以处理 switch 语句了。这个过程有点像淘气小男孩折磨一只昆虫每次掰掉它一条腿。首先我把 switch 语句中的 Engineer 这一分支复制到 Engineer 类:

```
class Engineer...
  int payAmount(Employee emp) {
      return emp.getMonthlySalary();
  }
```

这个新函数覆写了超类中的 switch 语句内专门处理 Engineer 的分支。我是个偏执狂，有时我会故意在 case 子句中放一个陷阱，检查 Engineer 子类是否正常工作:

```
class EmployeeType...
  int payAmount(Employee emp) {
      switch (getTypeCode()) {
          case ENGINEER:
             throw new RuntimeException ("Should be being overridden");
          case SALESMAN:
             return emp.getMonthlySalary() + emp.getCommission();
          case MANAGER:
             return emp.getMonthlySalary() + emp.getBonus();
          default:
             throw new RuntimeException("Incorrect Employee");
      }
  }
```

注意:

throw new RuntimeException ("Should be being overridden"); 是加粗的部分。

接下来，我重复上述过程，直到所有分支都被去除为止:

```
class Salesman...
  int payAmount(Employee emp) {
      return emp.getMonthlySalary() + emp.getCommission();
  }
class Manager...
  int payAmount(Employee emp) {
      return emp.getMonthlySalary() + emp.getBonus();
  }
```

然后，将超类的 payAmount()函数声明为抽象函数:

```
class EmployeeType...
  abstract int payAmount(Employee emp);
```

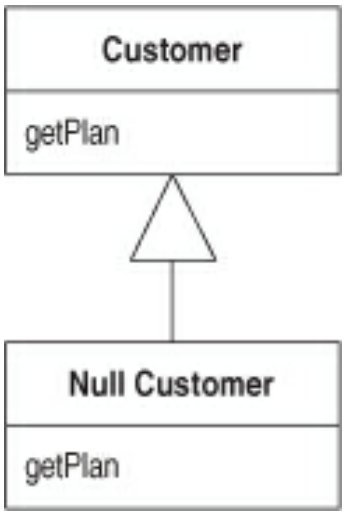
9.7 Introduce Null Object(引入 Null 对象)

你需要再三检查某对象是否为 null。

将 null 值替换为 null 对象。

```
if (customer == null) plan = BillingPlan.basic();
else plan = customer.getPlan();
```

↓



动机

多态的最根本好处在于:你不必再向对象询问“你是什么类型”而后根据得到的答案调用对象的某个行为一你只管调用该行为就是了,其他的一切多态机制会为你安排妥当。当某个字段内容是 null 时,多态可扮演另一个较不直观(亦较不为人所知)的用途。让我们先听听 Ron Jeffries 的故事。

- Ron Jeffries

  我们第一次使用 Null Object 模式,是因为 Rich Garzaniti 发现, 系统在向对象发送一个消息之前,总要检查对象是否存在,这样的检查出现很多次。我们可能会向一个对象索求它所相关的 Person 对象，然后再问那个对象是否为 null。如果对象的确存在，我们才能调用它的 rate()函数以查询这个

人的薪资级别。我们在好些地方都是这样做的，造成的重复代码让我们很烦心。

所以，我们编写了一个 MissingPerson 类，让它返回‘0’薪资等级[我们也把空对象(null object)称为虚拟对象(missing object)]。很快地，MissingPerson 就有了很多函数，rate()自然是其中之一。如今我们的系统有超过 80 个空对象类。

我们常常在显示信息的时候使用空对象。例如我们想要显示一个 Person 对象信息，它大约有 20 个实例变量。如果这些变量可被设为 null，那么打印一个 Person 对象的工作将非常复杂。所以我们不让实例变量被设为 null，而是插入各式各样的空对象它们都知道如何正确地显示自己。这样，我们就可以摆脱大量过程化的代码。

我们对空对象的最聪明运用，就是拿它来表示不存在的 Gemstone 会话:我们使用 Gemstone 数据库来保存成品(程序代码),但我们更愿意在没有数据库的情况下进行开发,每过一周左右再把新代码放进 Gemstone 数据库。然而在代码的某些地方,我们必须登录一个 Gemstone 会话。当没有 Gemstone 数据库时,我们就仅仅安插一个“虚构的 Gemstone 会话”,其接口和真正的 Gemstone 会话一模一样,使我们无需判断数据库是否存在,就可以进行开发和测试。

空对象的另一个用途是表现出“虚构的箱仓”(missing bin)。所谓“箱仓”，这里是指集合,用来保存某些薪资值,并常常需要对各个薪资值进行加和或遍历。如果某个箱仓不存在,我们就给出一个虚构的箱仓对象,其行为和一个空箱仓一样。这个虚构箱仓知道自己其实不带任何数据，总值为 0。通过这种做法，我们就不必为上千位员工每人产生数十来个空箱对象了。

使用空对象时有个非常有趣的性质:系统几乎从来不会因为空对象而被破坏。由于空对象对所有外界请求的响应都和真实对象一样,所以系统行为总是正常的。但这并非总是好事，有时会造成问题的侦测和查找上的困难，因为从来没有任何东西被破坏。当然，只要认真检查一下，你就会发现空对象有时出现在不该出现的地方。

请记住: 空对象一定是常量，它们的任何成分都不会发生变化。因此我们可以使用 Singleton 模式[Gang of Four]来实现它们。例如不管任何时候,只要你索求一个 MissingPerson 对象,得到的一定是 MissingPerson 的唯一实例。

关于 Null Object 模式，你可以在 Woolf[Woolf]中找到更详细的介绍。

做法

- 为源类建立一个子类,使其行为就像是源类的 null 版本。在源类和 null 子类中都加上 isNull()函数，前者的 isNull()应该返回 false，后者的 isNull()应该返回 true。

→ 下面这个办法也可能对你有所帮助:建立一个 nullable 接口，将 isNull()函数放在其中，让源类实现这个接口。

→ 另外，你也可以创建一个测试接口，专门用来检查对象是否为 null。

- 编译。
- 找出所有“索求源对象却获得一个 null”的地方。修改这些地方，使它们改而获得一个空对象。
- 找出所有“将源对象与 null 做比较”的地方。修改这些地方，使它们调用 isNull()函数。

→你可以每次只处理一个源对象及其客户程序,编译并测试后,再处理另一个源对象。

→你可以在“不该再出现 null”的地方放上一些断言，确保 null 的确不再出现。这可能对你有所帮助。

- 编译，测试。
- 找出这样的程序点:如果对象不是 null,做 A 动作,否则做 B 动作。
- 对于每一个上述地点，在 null 类中覆写 A 动作，使其行为和 B 动作相同。
- 使用上述被覆写的动作,然后删除“对象是否等于 null”的条件测试。编译并测试。

范例

一家公用事业公司的系统以 site 表示地点(场所)。庭院宅第(house)和集体公寓(apartment)都使用该公司的服务。任何时候每个地点都拥有(或说都对应于)一个顾客，顾客信息以 Customer 表示:

```
class Site...
  Customer getCustomer() {
      return _customer;
  }
  Customer _customer;
```

Customer 有很多特性，我们只看其中三项:

```
class Customer...
  public String getName() {...}
  public BillingPlan getPlan() {...}
  public PaymentHistory getHistory() {...}
```

本系统又以 PaymentHistory 表示顾客的付款记录，它也有其自己的特性:

```
public class PaymentHistory...
  int getWeeksDelinquentInLastYear()
```

上面的各种取值函数允许客户取得各种数据。但有时候一个地点的顾客搬走了新顾客还没搬进来，此时这个地点就没有顾客。由于这种情况有可能发生，所以我们必须保证 Customer 的所有用户都能够处理“Customer 对象等于 null”的情况。下面是一些示例片段:

```
Customer customer = site.getCustomer();
BillingPlan plan;
if (customer == null) plan = BillingPlan.basic();
else plan = customer.getPlan();
// ...
String customerName;
if (customer == null) customerName = "occupant";
else customerName = customer.getName();
// ...
int weeksDelinquent;
if (customer == null) weeksDelinquent = 0;
else weeksDelinquent = customer.getHistory().getWeeksDelinquentInLastYe
ar();
```

这个系统中可能有许多地方使用 Site 和 Customer 对象，它们都必须检查 Customer 对象是否等于 null，而这样的检查完全是重复的。看来是使用空对象的时候了。

首先新建一个 NullCustomer，并修改 Customer，使其支持“对象是否为 null”的检查:

```
class NullCustomer extends Customer {
  public boolean isNull() {
      return true;
  }
}

class Customer...
  public boolean isNull() {
      return false;
  }

  protected Customer() {} //needed by the NullCustomer
```

如果你无法修改 Customer，可以使用第 266 页的做法:建立一个新的测试接口。

如果你喜欢，也可以新建一个接口，昭告大家“这里使用了空对象”:

```
interface Nullable {
  boolean isNull();
}
class Customer implements Nullable
```

我还喜欢加入一个工厂函数，专门用来创建 NullCtomer 对象。这样一来，用户就不必知道空对象的存在了:

```
class Customer...
  static Customer newNull() {
       return new NullCustomer();
  }
```

接下来的部分稍微有点麻烦。对于所有“返回 null”的地方,我都要将它改为“返回空对象”。此外,我还要把 foo==null 这样的检查替换成 foo.isNull()。我发现下列办法很有用:查找所有提供 Customer 对象的地方,将它们都加以修改,使它们不能返回 null，改而返回一个 NullCustomer 对象。

```
class Site...
  Customer getCustomer() {
      return (_customer == null) ?
          Customer.newNull():
          _customer;
  }
```

另外,我还要修改所有使用 Customer 对象的地方,让它们以 isNull()函数进行检查，不再使用==null 检查方式。

```
Customer customer = site.getCustomer();
BillingPlan plan;
if (customer.isNull()) plan = BillingPlan.basic();
else plan = customer.getPlan();
// ...
String customerName;
if (customer.isNull()) customerName = "occupant";
else customerName = customer.getName();
// ...
int weeksDelinquent;
if (customer.isNull()) weeksDelinquent = 0;
else weeksDelinquent = customer.getHistory().getWeeksDelinquentInLastYe
ar();
```

毫无疑问，这是本项重构中最需要技巧的部分。对于每一个需要替换的可能等于 null 的对象，我都必须找到所有检查它是否等于 null 的地方，并逐一替换。如果这个对象被传播到很多地方，追踪起来就很困难。上述范例中，我必须找出每一个类型为 Customer 的变量，以及它们被使用的地点。很难将这个过程分成更小的步骤。有时候我发现可能等于 null 的对象只在某几处被用到，那么替换工作比较简单;但是大多数时候我必须做大量替换工作。还好，撤销这些替换并不困难，因为我可以不太困难地找出对 isNull()的调用动作，但这毕竟也是很零乱很恼人的。

这个步骤完成之后，如果编译和测试都顺利通过，我就可以宽心地露出笑容了。接下来的动作比较有趣。到目前为止，使用 isNull()函数尚未带来任何好处。只有把相关行为移到 NullCustomer 中并去除条件表达式之后，我才能得到切实的利益。我可以逐一将各种行为移过去。首先从“取得顾客名称”这个函数开始。此时的客户端代码大约如下:

String customerName;if(customer.isNull())customerName="occupant";else customerName= customer.getName();

首先为 NullCustomer 加入一个合适的函数，通过这个函数来取得顾客名称:

```
String customerName;
if (customer.isNull()) customerName = "occupant";
else customerName = customer.getName();
```

现在，我可以去掉条件代码了：

```
String customerName = customer.getName();
```

接下来我以相同手法处理其他函数，使它们对相应查询做出合适的响应。此外我还可以对修改函数做适当的处理。于是下面这样的客户端程序：

```
if (!customer.isNull())
    customer.setPlan(BillingPlan.special());
```

就变成了这样：

```
customer.setPlan(BillingPlan.special());

class NullCustomer...
  public void setPlan (BillingPlan arg) {}
```

请记住：只有当大多数客户代码都要求空对象做出相同响应时，这样的行为搬移才有意义。注意，我说的是“大多数”而不是“所有”。任何用户如果需要空对象做出不同响应，他们仍然可以使用 isNull()函数来测试。只要大多数客户端都要求空对象做出相同响应，他们就可以调用默认的 null 行为，而你也就受益匪浅了。

上述范例略带差异的某种情况是，某些客户端使用 Customer 函数的运算结果：

```
if (customer.isNull()) weeksDelinquent = 0;
      else weeksDelinquent = customer.getHistory().getWeeksDelinquentIn
LastYear();
```

我可以新建一个 NullPaymentHistory 类，用以处理这种情况：

```
class NullPaymentHistory extends PaymentHistory...
  int getWeeksDelinquentInLastYear() {
      return 0;
  }
```

并修改 NullCustomer，让它返回一个 NullPaymentHistory 对象：

```
class NullCustomer...
  public PaymentHistory getHistory() {
      return PaymentHistory.newNull();
  }
```

然后，我同样可以删除这一行条件代码：

```
int weeksDelinquent = customer.getHistory().getWeeksDelinquentInLastYear();
```

你常常可以看到这样的情况：空对象会返回其他空对象。

范例：测试接口

除了定义 isNull()之外，你也可以建立一个用以检查“对象是否为 null”的接口。

使用这种办法，需要新建一个 Null 接口，其中不定义任何函数：

```
interface Null {}
```

然后，让空对象实现 Null 接口：

```
class NullCustomer extends Customer implements Null...
```

然后，我就可以用 instanceof 操作符检查对象是否为 null：

```
aCustomer instanceof Null
```

通常我尽量避免使用 instanceof 操作符，但在这种情况下，使用它是没问题的。而且这种做法还有另一个好处：不需要修改 Customer。这么一来即使无法修改 Customer 源码，我也可以使用空对象。

其他特殊情况

使用本项重构时，你可以有几种不同的空对象，例如你可以说“没有顾客”（新建的房子和暂时没人住的房子）和“不知名顾客”（有人住，但我们不知道是谁）这两种情况是不同的。果真如此，你可以针对不同的情况建立不同的空对象类。有时候空对象也可以携带数据，例如不知名顾客的使用记录等，于是我们可以在查出顾客姓名之后将账单寄给他。

本质上来说，这是一个比 Null Object 模式更大的模式: Special Case 模式。所谓特例类(special case), 也就是某个类的特殊情况, 有着特殊的行为。因此表示“不知名顾客”的 UnknownCustomer 和表示“没有顾客”的 NoCustomer 都是 Customer 的特例。你经常可以在表示数量的类中看到这样的“特例类”,例如 Java 浮点数有“正无穷大”、“负无穷大”和“非数量”(NaN)等特例。特例类的价值是:它们可以降低你的“错误处理”开销，例如浮点运算决不会抛出异常。如果你对 NaN 做浮点运算，结果也会是个 NaN。这和“空对象的访问函数通常返回另一个空对象”是一样的道理。

9.8 Introduce Assertion(引入断言)

某一段代码需要对程序状态做出某种假设。

以断言明确表现这种假设。

```
double getExpenseLimit() {
    // should have either expense limit or a primary project
    return (_expenseLimit != NULL_EXPENSE) ?
        _expenseLimit:
            _primaryProject.getMemberExpenseLimit();
}
```

↓

```
double getExpenseLimit() {
    Assert.isTrue (_expenseLimit != NULL_EXPENSE || _primaryProject !=
null);
    return (_expenseLimit != NULL_EXPENSE) ?
        _expenseLimit:
        _primaryProject.getMemberExpenseLimit();
}
```

动机

常常会有这样一段代码:只有当某个条件为真时，该段代码才能正常运行。例如平方根计算只对正值才能进行,又例如某个对象可能假设其字段至少有一个不等于 null。

这样的假设通常并没有在代码中明确表现出来,你必须阅读整个算法才能看出。有时程序员会以注释写出这样的假设。而我要介绍的是一种更好的技术: 使用断言明确标明这些假设。明确标明这些假设。

断言是一个条件表达式，应该总是为真。如果它失败，表示程序员犯了错误。因此断言的失败应该导致一个非受控异常(unchecked exception)。断言绝对不能被系统的其他部分使用。实际上，程序最后的成品往往将断言统统删除。因此，标记“某些东西是个断言”是很重要的。

断言可以作为交流与调试的辅助。在交流的角度上,断言可以帮助程序阅读者理解代码所做的假设;在调试的角度上,断言可以在距离 bug 最近的地方抓住它们。当我编写自我测试代码的时候发现,断言在调试方面的帮助变得不那么重要了,但我仍然非常看重它们在交流方面的价值。

做法

如果程序员不犯错,断言就应该不会对系统运行造成任何影响,所以加入断言永远不会影响程序的行为。

- 如果你发现代码假设某个条件始终为真,就加入一个断言明确说明这种情况。

→ 你可以新建一个 Assert 类,用于处理各种情况下的断言。

注意,不要滥用断言。请不要使用它来检查“你认为应该为真”的条件,请只使用它来检查“一定必须为真”的条件。滥用断言可能会造成难以维护的重复逻辑。在一段逻辑中加入断言是有好处的,因为它迫使你重新考虑这段代码的约束条件。如果不满足这些约束条件,程序也可以正常运行,断言就不会带给你任何帮助,只会把代码变得混乱,并且有可能妨碍以后的修改。

你应该常常问自己: 如果断言所指示的约束条件不能满足, 代码是否仍能正常运行? 如果可以，就把断言拿掉。

另外，还需要注意断言中的重复代码。它们和其他任何地方的重复代码一样不好闻。你可以大胆使用 Extract Method(110)去掉那些重复代码。

范例

下面是一个简单例子:开支限制。后勤部门的员工每个月有固定的开支限额;业务部门的员工则按照项目的开支限额来控制自己的开支。一个员工可能没有开支额度可

用,也可能没有参与项目,但两者总要有一个(否则就没有经费可用了)。在开支限额相关程序中,上述假设总是成立的,因此:

```
class Employee...
  private static final double NULL_EXPENSE = -1.0;
  private double _expenseLimit = NULL_EXPENSE;
  private Project _primaryProject;
  double getExpenseLimit() {
      return (_expenseLimit != NULL_EXPENSE) ?
          _expenseLimit:
          _primaryProject.getMemberExpenseLimit();
  }

  boolean withinLimit (double expenseAmount) {
      return (expenseAmount <= getExpenseLimit());
  }
```

这段代码包含了一个明显假设:任何员工要么参与某个项目，要么有个人开支限额。我们可以使用断言在代码中更明确地指出这一点:

```
double getExpenseLimit() {
    Assert.isTrue (_expenseLimit != NULL_EXPENSE || _primaryProject !=
null);
    return (_expenseLimit != NULL_EXPENSE) ?
        _expenseLimit:
        _primaryProject.getMemberExpenseLimit();
}
```

注意: **Assert.isTrue (*expenseLimit != NULL*EXPENSE || _primaryProject != null);** 是加粗的部分。

这条断言不会改变程序的任何行为。另外，如果断言中的条件不为真，我就会收到一个运行期异常:也许是在 withinLimit()函数中抛出一个空指针异常，也许是在 Assert.isTrue()函数中抛出一个运行期异常。有时断言可以帮助程序员找到 bug，因为它离出错地点很近。但是，更多时候，断言的价值在于:帮助程序员理解代码正确运行的必要条件。

我常对断言中的条件表达式使用 Extract Method(110)，也许是为了将若干地方的重复码提炼到同一个函数中，也许只是为了更清楚说明条件表达式的用途。

在 Java 中使用断言有点麻烦: 没有一种简单机制可以协助我们插入这东西 ①。断言可被轻松拿掉，所以它们不可能影响最终成品的性能。编写一个辅助类(例如 Assert 类)当然有所帮助,可惜的是断言参数中的任何表达式不论什么情况都一定会被执行一遍。阻止它的唯一办法就是使用类似下面的手法:

```
double getExpenseLimit() {
    Assert.isTrue (Assert.ON &&
        (_expenseLimit != NULL_EXPENSE || _primaryProject != null));
    return (_expenseLimit != NULL_EXPENSE) ?
        _expenseLimit:
        _primaryProject.getMemberExpenseLimit();
}
```

① J2SE 1.4 已经支持断言语句。- 译者注

或者是这种手法:

```
double getExpenseLimit() {
    if (Assert.ON)
        Assert.isTrue (_expenseLimit != NULL_EXPENSE || _primaryProject
 != null);
    return (_expenseLimit != NULL_EXPENSE) ?
        _expenseLimit:
        _primaryProject.getMemberExpenseLimit();
}
```

如果 Assert.ON 是个常量，编译器就会对它进行检查;如果它等于 false，就不再执行条件表达式后半段代码。但是，加上这条语句实在有点丑陋，所以很多程序员宁可仅仅使用 Assert.isTrue()函数，然后在项目结束前以过滤程序滤掉使用断言的每一行代码(可以使用 Perl 之类的语言来编写这样的过滤程序)。

Assert 类应该有多个函数，函数名称应该帮助程序员理解其功用。除了 isTrue()之外，你还可以为它加上 equals()和 shouldNeverReachHere() 等函数。