

## 使用财务模型

要全面理解本章，你需要先阅读第6章。这是本书中不寻常的一章。本章给出我们如何使用第6章的模式，而非继续描述一组模式。因为第6章的账务模式非常抽象，所以这是一个很艰巨的任务。要理解模式如何真正工作，我们需要一个完整的能工作的示例。

本章以一个Total Telecommunications集团（TT）的电话应用系统为例，讲述模型所使用的账目和记入规则。按照教材的惯例，需要把给出的示例简单化，但这些示例至少要让读者明白模型是如何工作的。我的目的就是说明账目模型的使用而非为电话公司建模。

既然在本章中我的目的就是举例，我在示例里面也就用了很多程序代码。我选用Digital Smalltalk语言而不是C++，因为对我来说Smalltalk更容易传递我的想法。这些概念可以很容易转换到C++语言里面。我用了一些在第14章中才详细说明的模型转换模式。我还使用Kent Beck的编码模式（可能略有变化）[1]。我要强调，我没有试图优化我的代码，也没有提供完整的代码，而只是给出重点部分。

TT公司的基本收费方案很简单。所有的电话被分为白天电话和晚上电话。白天电话的时间从早晨7:00到晚上7:00。这种区分以电话的开始时间为基础<sup>①</sup>。白天的电话费用是这样计算的：第1分钟98美分，接下来的每分钟30美分。晚上电话的计费方式是：第1分钟70美分，接下来的20分钟每分钟20美分，其余的每分钟12美分。政府对于日历月中第1个50美元的电话收取6%的税，其余的收取4%的税。

本章以结构模型（参见7.1节）的讨论开始，结构模型当然是基于第6章中给出的模式。然后我们看看这些结构在实现方面的一些有趣问题（参见7.2节）。为了设置对象，我们先要设置新的电话服务（参见7.3节），然后要建立通话（参见7.4节）。随后我们看看记入规则，同时检查用于实现基于账目的触发的代码（参见7.5节）。我们给出3个记入规则的示例：把

<sup>①</sup> 为了简单起见，我略过了电话跨时间分界的情况。

电话分成白天和夜晚两类（参见7.6节），按时间收费（参见7.7节），计算税款（参见7.8节）。每条规则都表明行为的一个特定方面。头两条规则以条目到条目的方式进行操作，一个公共的超类型——每个条目记入规则——处理公共的行为。由每个条目记入规则的一个简单的singleton子类型来处理白天和夜晚电话的分开收费。虽然我们需要不同的策略来区分白天和夜晚通话，但既然基本处理过程一样，我们就可以使用费率表作为参数化的策略对象。这允许我们处理任何依照通话时间长短而收费的记入规则。作为策略对象的费率表可以被用于这种基于通话时间的计算。事实上，费率表也被用于下面的计算税的规则。不像前一条规则，这条规则一定是按月计算的，但我们不能假定每月只运行它一次。

三个记入规则类应该能给我们带来好的启示，启示我们如何使用账目/库存模式来既表示货币的事务也表示非货币的事务。

在开发代码时，我喜欢一开始就建立结构模型的大体框架。然后使用原型，但对于结构模型的更新，我一向都很谨慎（要不然我自己都不知道进展到哪一步）。伴随着难于处理的行为的出现，在开始的时候或是在编程的时候可以使用事件图或交互图。如果我认为有必要写文档来记录我用这些行为做了什么（就像在本书里面），我会在完成代码后就画图。图形不是代码的替代品，它们帮助说明代码的作用。（使用合适的工具，事件图可以作为代码使用。）

## 7.1 结构模型

最好开始的时候就使用结构模型，因为它们给出最终的各种模型的概貌。图7-1显示模型内的包。我把模型分为两个包：电话服务和账目。使用账目框架的一个好处就是它可以被用于不同的行业，所以我们需要确保将账目模型从任何行业相关的概念中分离（就是账目模型独立于任何行业相关的概念）。

134

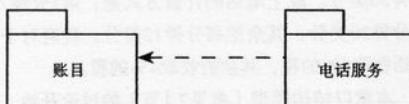


图7-1 TT模型的包

账目包持有抽象账务类型，在这个特定领域里，账务类型由电话服务包来扩展。

图7-2表示TT的账务模型，这是基于第6章的模式。模型中从记入规则到账目有3个关联。触发器和输出都与第6章相似，但带键值的输出是新的。

这可以给那些有需要的记入规则一些多重输出账目。这种需要在本章稍后的示例中会很明显。

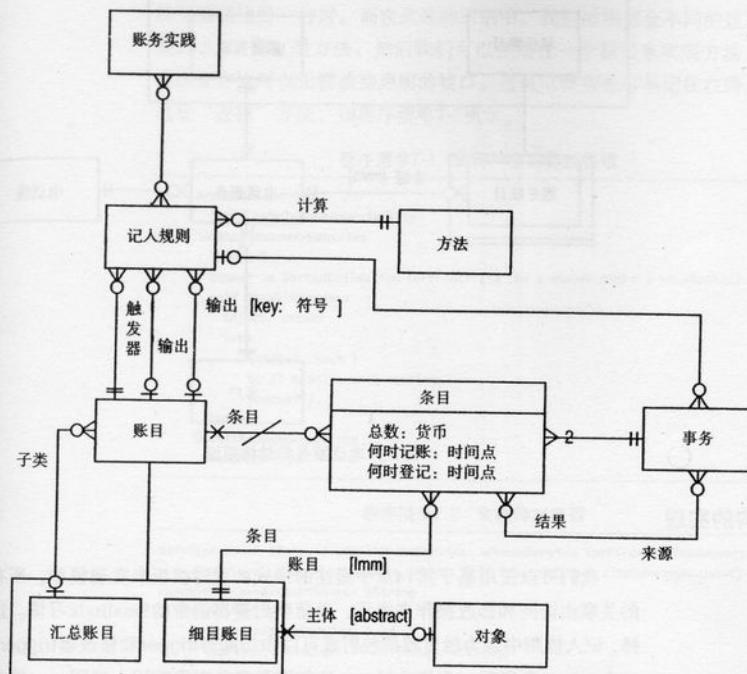


图7-2 TT的账目模型

135

图7-3展示电话服务的模型。客户被允许有多条电话线。电话服务是指每个客户被分配一条电话线。每个电话服务被绑定到描述如何付账的账务实践。此图说明为什么目标映射被加到图7-2中的细目账目。我们需要一种方式来找出细目账目到底做什么，但是我们不需要从账目包看到电话服务包，因为它会破坏重用。这样，我们形成细目账目的子类型。使用子类型化，只能从子类型看到父类型。服务账目了解电话服务是绝对允许的，因为它们都在电话服务包中。然而，我们在不知道某个细目账目是服务账目的情况下就可以对其进行引用。细目账目上的抽象映射告诉我们细目账目可以作为主题被连接到一个对象（类型不定）。这些可以由细目账目的子类型实现——这是多态现象的一个经典示例。

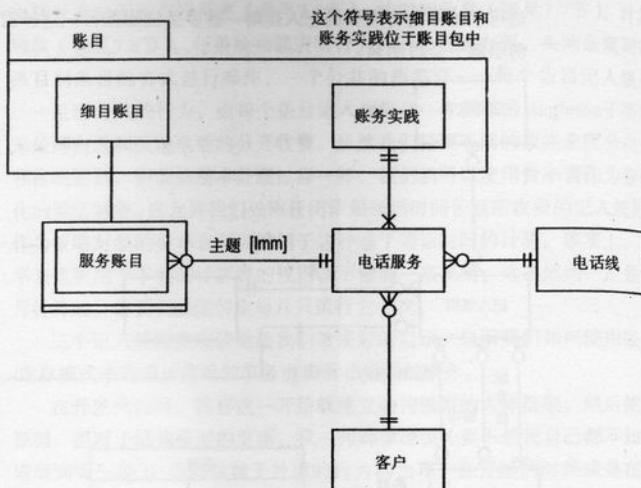


图7-3 电话服务的结构模型

## 7.2 结构的实现

我们可以使用基于第14章中描述的模式的设计模板来实现模型。所有的关联由访问和修改操作来表示。单值映射遵循通常的Smalltalk习惯。这样，记入规则中名为触发器的映射就可以由访问器trigger和修改器trigger:anAccount来实现。多值映射——比如是账务实践的记入规则——就有访问器postingRules和修改器addPostingRule:aPostingRule和removePostingRule:aPostingRule。

账目上的条目操作是多态操作——当汇总账目统计子节点的时候，细目账目返回一个变量实例（如程序清单7-1所示）。

这个模型没有账目类型。记入规则由汇总账目定义。在本章的例子中，我们可以使用汇总账目或是账目类型来定义记入规则。使用汇总账目有一点儿复杂，但作为示例更合适。依照14.5.1节的风格，已定义的高级别汇总账目被保存在账目类里面的某个类变量中，并可由类方法findWithName:aString来访问。

有时需要小段代码来为特定汇总账目的特定电话服务找出服务账目。不难想出很多种实现方式：请求电话服务在给定的汇总账目中查找账目，

或是请求汇总账目查找它连到电话服务的子节点。这两种方式都很合理，但很难决定哪种更好。另外，每种方式都应用某种导航路径，这样其中一种可能就比另一种好。而在此处的示例中，我们采用完全不同的技术，依照14.5.1节建立类方法。然后我们可以使用任一个路径来实现方法，而且在改变方法时也无需改变声明的接口。这可以使我更容易记住在哪里实现这些“查找”方法，如程序清单7-2所示。

#### 程序清单7-1 获得一个账目的条目

```

Account>>entries
^self subclassResponsibility
SummaryAccount>>entries
|answer|
answer := SortedCollection sortBlock:[:a :b| a whenBooked > b whenBooked].
self detailAccounts
inject: answer
into:
[:total :each |
total addAll: each entries;
yourself].
^answer
DetailAccount>>entries
^entries copy

```

#### 程序清单7-2 查找特定账目

```

ServiceAccount class>>findWithPhoneService: aPhoneService topParent: aTopSummaryAccount
^aPhoneService serviceAccounts detect: [:i| i parentTop = aTopSummaryAccount]
PhoneService>>accountNamed: aString
^ServiceAccount
findWithPhoneService: self
topParent: (Account findWithName: aString)

```

实际上，在电话服务中编写的一个方法（比如`accountNamed:aString`）经常很方便使用。这个方法调用`findWithPhoneService: topParent`并能提供全部两种方式的长处。

虽然模型支持多腿事务，但这里的示例还是全部使用双腿事务。程序清单7-3表示一个事务的特殊构造方法，使用该方法，我们可以生成双腿事务。一个方法传送所有的信息，包括源条目和生成它的记入规则。另一个方法被用于一开始就读进来的初始化的属性。

列表给出很多编码技术。一个构造器参数方法[1]（以`set`为前缀）用参数初始化新对象。在构造参数方法的内部，用`require`消息进行预处理检查。为了提高性能，可以重新定义`require`方法来去掉这个检查。按“契约式设计”的观点[3]，另一个设计因素就是使用不变式的检查。

## 程序清单7-3 构造双腿事务

```

Transaction class>>newWithAmount: aQuantity from: fromAccount to: toAccount
whenCharged: aTimepointOrDate
^self
    newWithAmount: aQuantity
    from: fromAccount
    to: toAccount
    whenCharged: aTimepointOrDate
    creator: nil
    sources: Set new
newWithAmount: aQuantity from: fromAccount to: toAccount whenCharged:
aTimepointOrDate creator: aPostingRule sources: aSetOfEntries
^self new
    setAmount: aQuantity
    from: fromAccount
    to: toAccount
    whenCharged: aTimepointOrDate
    creator: aPostingRule
    sources: aSetOfEntries
Transaction>>setAmount: aMoney from: aDebitAccount to: aCreditAccount whenCharged:
aTimepointOrDate creator: aPostingRule sources: aSetOfEntries
"private"
self require:
    [aMoney isKindOf: Money.
    aDebitAccount isKindOf: ServiceAccount.
    aCreditAccount isKindOf: ServiceAccount.
    (aTimepointOrDate isKindOf: Date) or: [aTimepointOrDate isKindOf: Timepoint].
    (creator = nil) or: [creator isKindOf: PostingRule]].
self initialize.
self addEntry: (Entry new
    setAccount: aCreditAccount
    amount: aMoney
    charged: aTimepointOrDate).
self addEntry: (Entry new
    setAccount: aDebitAccount
    amount: aMoney negated
    charged: aTimepointOrDate).
creator:= aPostingRule.
aSetOfEntries do: [:i| self sourcesAdd: i].
self checkInvariant.
Object>>require: aBooleanBlock
aBooleanBlock value ifFalse: [self error: 'Precondition Violation']
Transaction>>checkInvariant
|balance|
balance := entries
inject: Quantity zero
into: [:total :each | total := total + each amount].
self require: [balance = Quantity zero].

```

## 7.3 设置新的电话服务

构建一个新的电话服务不仅仅是一个实例化电话服务对象的问题。要使账务系统运行，也必须构造服务账目。如图7-4、图7-5和程序清单7-4所示，尽管这个示例没有包含多个账务实践，但也有足够的弹性来为任何账

务实践设置账目。

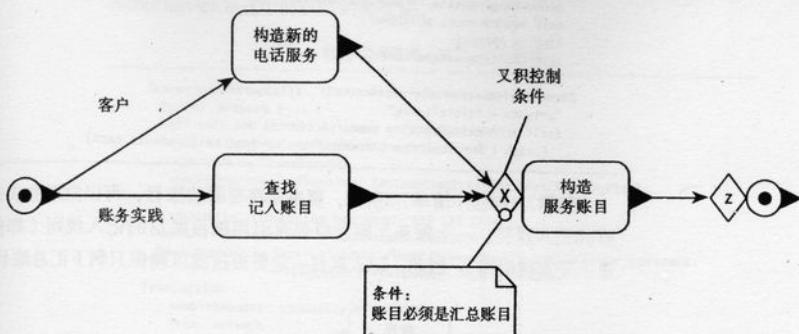


图7-4 构造新电话服务的事件图

本图里使用又积控制条件（对通常事件图的一个扩展）。控制条件要对输入触发器和各种组合进行估计，本例中就是每个新电话服务和记入账目的组合。它为账务实践中的每个电话服务和汇总记入账目调用构造服务账目的操作。

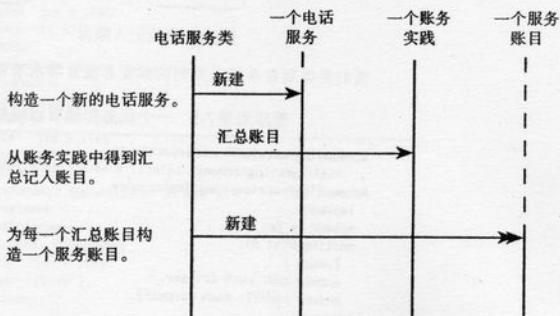


图7-5 构造新电话服务的交互图

#### 程序清单7-4 设置新的电话线

```

PhoneService class>>newWithAccountingPractice: anAccountingPractice customer:
aCustomer phoneLine: aPhoneLine
^self new
setAccountingPractice: anAccountingPractice
customer: aCustomer
phoneLine: aPhoneLine
PhoneService>>setAccountingPractice: anAccountingPractice customer: aCustomer
phoneLine: aString
!newObj summaryAccounts!
self require:
[(anAccountingPractice isKindOf: AccountingPractice) &

```

```

(aCustomer isKindOf: Customer]).
name := (aCustomer name), '#', (aCustomer phoneServices size + 1) printString.
accountingPractice := anAccountingPractice.
self setCustomer: aCustomer.
line := aString.
self createServiceAccounts.
^self
PhoneService>>createServiceAccounts
"private - initializing"
(self accountingPractice summaryAccounts) do:
[:each | ServiceAccount newWithPhoneService: self parent: each].

```

如图7-6和程序清单7-5所示，要决定需要哪些账目，可以向账务实践索要其记入账目。一个账务实践可以包含引用细目账目的记入规则（即便这里不是直接引用）。因此，记入账目一定要被过滤以确保只剩下汇总账目。

[140]

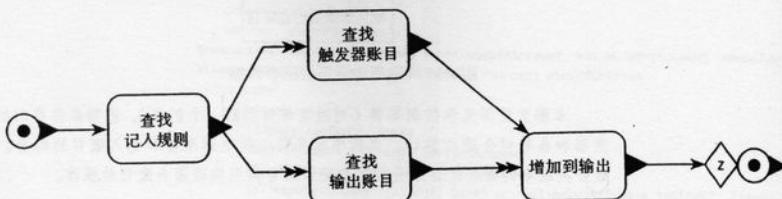


图7-6 查找记入账目

我们要得到每条记入规则的触发器账目和所有的输出账目。

#### 程序清单7-5 一个账务实践可以提供它的汇总账目

```

AccountingPractice>>summaryAccounts
^self postingAccounts select: [:each | each isSummary]
AccountingPractice>>postingAccounts
[answer]
answer := Set new.
postingRules do:
[:each |
answer add: each trigger.
answer addAll: each outputs].
^answer

```

[141]

## 7.4 建立通话

电话通话可以被建模成从网络账目到基本时间账目的事务。电话通话条目的单位是分钟。

下面的方法表示如何建立电话服务和设置一些示例通话。如程序清单7-6所示，注意：这个方法定义在一个名为Scenario1的类中。测试方法可能很复杂；这样把它们放在场景对象里面是很好的办法（这里使用“场景”(scenario)是“用例”(use-case)的意思，而不是9.4节里面定义的含义）。

这样即使在没有合适的测试框架的时候还能够控制它们。变量basicAccount和theService是这个测试类的类变量。

#### 程序清单7-6 建立测试通话

```

Scenario1>>setupCalls
|adams network|
self init.
adams := Customer new name: 'Adams'; persist.
theService := PhoneService
newWithAccountingPractice: (AccountingPractice basicBillingPlan)
customer: adams
phoneLine: (PhoneLine new name: '617 123 1234').
network := theService accountNamed: 'Network'.
basicAccount := ServiceAccount findWithPhoneService: theService topParent:
(Account findWithName: 'Basic Time').
Transaction
newWithAmount: (Quantity n:'10 min')
from: network
to: basicAccount
whenCharged: (Timepoint
date: 'jan 1 1995'
time: '13:15').
Transaction
newWithAmount: (Quantity n:'8 min')
from: network
to: basicAccount
whenCharged: (Timepoint
date: 'jan 1 1995'
time: '14:25').
Transaction
newWithAmount: (Quantity n:'6 min')
from: network
to: basicAccount
whenCharged: (Timepoint
date: 'jan 1 1995'
time: '19:05').
Transaction
newWithAmount: (Quantity n:'33 min')
from: network
to: basicAccount
whenCharged: (Timepoint
date: 'jan 1 1995'
time: '20:20').
AbasicAccount

```

142

使用用户自定义的基础类(比如数量)，可能使构造新对象变得困难。因此，数量有一个方法n:aString，用它可以从字符串生成数量。这是我使用的个人习惯，因为fromString:aString不太易于使用。

#### 7.5 实现基于账目的触发

我们这里使用基于账目的触发方案(参见6.7.3节)。每个账目有一个方法来处理自己，该方法引发所有以这个账目作为触发器的记入规则，如程序清单7-7所示。

## 程序清单7-7 一个账目引发以它为触发器的记入规则

```

DetailAccount->process
  self allOutboundRules do: [:j| j processAccount: self].
  lastProcessed := entries last
  allOutboundRules
    "private"
    |answer|
    answer := self triggerFor.
    self allParents do: [:i| answer addAll: i triggerFor].
    ^answer

```

条目被放在orderedCollection中，新加进来的都加在末尾。lastProcessed实例变量跟踪处理状态。

## 7.6 把电话分成白天和夜晚两类

为把电话通话分为白天和夜晚两类，我们检查每个条目，考虑条目上的时间，然后生成从基本时间账目到白天账目或到夜晚账目的事务。

按条目进行操作的记入规则是很常见的。我们可以构造名为“每条目记入规则”的记入规则的抽象子类型（EachEntryPR类）。如图7-7和程序清单7-8所示，这个子类型对触发账目中的每条未处理条目调用processEntry:anEntry操作。



图7-7 处理账目的每条目记入规则的方法

在每条未处理条目上调用处理条目的操作。

## 程序清单7-8 EachEntryPR是如何处理触发器账目的

```

EachEntryPR->processAccount: anAccount
  self currentInput: anAccount.
  anAccount unprocessedEntries do: [:each | self processEntry: each].
  self << clean.

EachEntryPR->processEntry: anEntry
  self subclassResponsibility.

DetailAccount->unprocessedEntries
  self isUnprocessed ifTrue: [^ entries copy].
  ^ entries
    copyFrom: self firstUnprocessedIndex
    to: entries size.

DetailAccount->isUnprocessed
  "private"
  ^ lastProcessed isNil.

DetailAccount->firstUnprocessedIndex
  "private"
  ^ (entries indexOf: lastProcessed) + 1

```

消息currentInput:读入实例变量来保存记入规则正在处理的服务账目，

143

如程序清单7-9所示。它由私有方法访问，并只能定义在processAccount的执行过程中。一个临时的私有实例变量经常被用于这种情况，因为通常的记入规则也作为实例变量来定义（虽然在本例中并非如此）。这样，我们不能用实例化的方法来激活这个规则。一个替代方法是把记入规则实例作为原型[2]来定义并通过克隆它来执行。

程序清单7-9 设置当前的输入和输出

```

PostingRule>>currentInput: anAccount
    "private"
    self require: [currentInput isNil].
    currentInput := anAccount.
    self setCurrentOutputs
PostingRule>>setCurrentOutputs
    "private"
    currentOutputs := Dictionary new.
    outputs associationsDo:
        [: each |
        currentOutputs
            at: each key
            put: (ServiceAccount
                findWithPhoneService: (currentInput phoneService)
                topParent: each value)]
PostingRule>>clean
    "private"
    currentInput := nil.
    currentOutputs := nil.

```

144

currentInput:消息也为与输入相同的电话服务设置了服务账目的当前输出。这个方法并不真正进行计算和记入。而是由processEntry:来完成，这个方法是抽象的，应该由子类来定义。在这里我们看到三层的类结构。PostingRule定义记入规则的基本接口和服务。EachEntryPR的处理账目方法是一个模板方法<sup>①</sup>，列出逐条处理条目的步骤，但如何具体地处理条目要在子类中实现。

对于这条记入规则，我们可以定义EachEntryPR的名为EveningDaySplitPR的新子类。这是 singleton类实现的一个实例（参见6.6.1节）。如程序清单7-10所示，类中包含一些账目代码，在初始化时进行设置。

程序清单7-10 初始化区分白天/夜晚的处理规则

```

EveningDaySplitPR>>initialize
super initialize.
outputs := Dictionary new .
outputs
    at: #evening
    put: (Account findWithName: 'Evening Time').
outputs
    at: #day
    put: (Account findWithName: 'Day Time')

```

<sup>①</sup> 模板方法是一个算法架构，其中有些步骤要推迟到子类中实现[2]。

通过重载processEntry方法来完成这个区分，如图7-8和程序清单7-11所示。

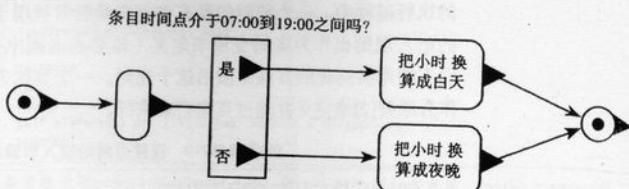


图7-8 白天/夜晚区分处理规则的方法(用于处理条目操作)

#### 程序清单7-11 白天/夜晚区分处理规则是如何处理条目的

```

EveningDaySplitPR>>processEntry: anEntry
Transaction
    newWithAmount: (anEntry amount)
    from: (anEntry account)
    to: (self outputFor: anEntry)
    whenCharged: (anEntry timepoint)
    creator: self
    sources: (Set with: anEntry)
EveningDaySplitPR>>outputFor: anEntry
^ (anEntry timepoint time > (Time fromString: '19:00')) | 
(anEntry timepoint time < (Time fromString: '07:00'))
    ifTrue: [self currentOutputs at: #evening]
    ifFalse: [self currentOutputs at: #day].

```

## 7.7 按时间收费

对白天和夜晚通话的收费可以使用同一个模式，如图7-9所示。收费是在一个条目接一个条目的基础上计算的，所以要使用EachEntryPR的一个子类。要使用两条记入规则：一条用于白天，一条用于夜晚。而同一个TransformPR类可用于两者。

这条记入规则的一个特性就是它由分钟的条目触发，但产生美元的条目。因此使用术语转换。实际的结果是产生两个事务。一个把分钟的账目再转换回账目网络，这样就完成了一个分钟的循环。第二个在财务方面产生一个事务：从网络收入账目到活动账目，如图7-10和程序清单7-12所示。

方法transformedAmount由方法对象计算（参见6.6.2节），明确地说是由费率表计算，比如像表7-1和表7-2所示的那样。方法类定义抽象的calculateFor方法。费率表是一个子类，存储一个两列的数量表以产生问题需要的阶梯状收费。这个可以使用字典来实现。字典里面的键值表示一些临界点，对应的值表示应用到临界点的费率。程序清单7-13表示了如何设置夜晚收费的费率。最高费率表明一旦超过最高门限值，我们该使用哪个费率。

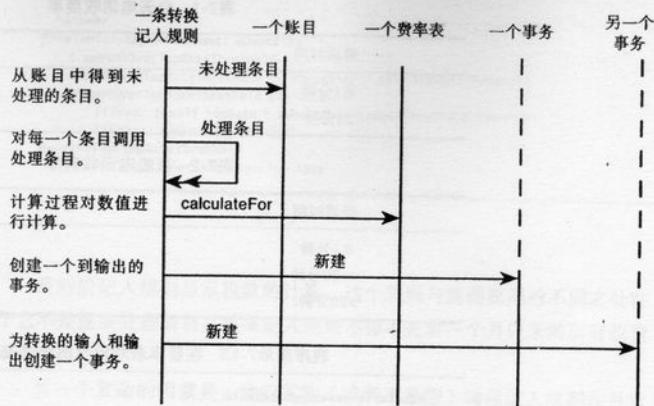


图7-9 使用转换记入规则处理账目的交互图

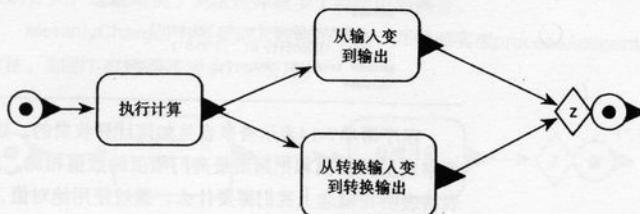


图7-10 针对转换记入规则方法（用于处理条目操作）的事件图

## 程序清单7-12 转换记入规则是如何处理条目的

```

TransformPR>>processEntry: anEntry
    Transaction
        newWithAmount: (anEntry amount)
        creator: self
        from: (anEntry account)
        timepoint: (anEntry timepoint)
        to: (self currentOutputs at: #out)
        sources: (Set with: anEntry).
        Transaction newWithAmount: (self transformedAmount: anEntry)
        creator: self
        from: (self currentOutputs at: #transformedFrom)
        timepoint: (anEntry timepoint)
        to: (self currentOutputs at: #transformedTo)
        sources: (Set with: anEntry).
TransformPR>>transformedAmount: anEntry
    "private"
    ^self calculationMethod calculateFor: anEntry amount
  
```

表7-1 白天电话收费率

通话时间	费用
≤1分钟	98美分
>1分钟	30美分

表7-2 夜晚电话收费率

通话时间	费用
≤1分钟	70美分
1~20分钟	20美分
>20分钟	10美分

程序清单7-13 在费率表对象里面设置夜晚费率

```

RateTable>>eveningRateTable
| answer |
answer := RateTable new.
answer
rateAt: (Quantity n: '1 min')
put: (Quantity n: '.7 USD').
answer
rateAt: (Quantity n: '21 min')
put: (Quantity n: '.2 USD').
answer topRate: (Quantity n: '.12 USD').
^answer

```

程序清单7-14表示费率表是如何计算收费的。这分两个步骤进行：在费率表里面扫描和把超出最高门限值的数值相加。我没有在此画出图示。表格表明在概念上我们需要什么。通过使用绝对值，它们可以一样地处理正数和负数，这是观察这些系统时比较特殊的事情。

程序清单7-14 费率表是如何依照输入数值进行计算的

```

RateTable>>calculateFor: aQuantity
| answer input|
self require: [aQuantity unit = self thresholdUnits].
input := aQuantity abs.
answer := (self tableAmount: input) + (self topRateAmount: input).
^aQuantity positive
  ifTrue: [answer]
  iffFalse: [answer negated]
RateTable>>tableAmount: aQuantity
"private"
|input sortedKeys lastKey thisRowKeyAmount answer|
sortedKeys := table keys asSortedCollection.
lastKey := Quantity zero.
answer := Quantity zero.
sortedKeys do:
[:thisKey |
thisRowKeyAmount := ((aQuantity min: thisKey) - lastKey) max: Quantity zero.
answer := answer + ((table at: thisKey) * thisRowKeyAmount amount).
lastKey := thisKey].

```

```

^answer
RateTable->topRateAmount: aQuantity
| amountOverTopRateThreshold |
amountOverTopRateThreshold := aQuantity - self topRateThreshold.
amountOverTopRateThreshold positive
  ifTrue: [^self topRate * amountOverTopRateThreshold amount]
  ifFalse: [^aQuantity zero].
RateTable->topRateThreshold
^table keys asSortedCollection last

```

## 7.8 计算税款

最后的记入规则显示税款的计算。这个规则与前面规则的不同之处在于它不是逐条处理条目。这条记入规则不得不查看一个月以来的所有收费来计算税款。

另一个复杂的因素是：我们不能（或者不希望）确保记入规则在月末仅运行一次。因此，记入规则就要考虑到这个月因前一次触发而已经进行收费的税。这就引出如下的定理，即记入规则的定义应当独立于它们被触发的方式。这就增加了灵活性并减少了模型中的耦合。[148]

`MonthlyChargePR`类是记入规则的子类，并因而实现`processAccount`方法，如图7-11和程序清单7-15所示。

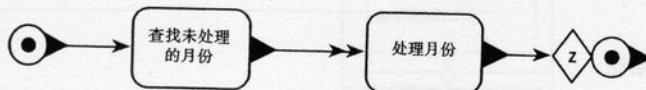


图7-11 处理账目的按月收费的记入规则的方法

这个处理是基于一段时间的结算，而非基于每个条目。

### 程序清单7-15 按月收费的记入规则的方法是如何处理账目的

```

MonthlyChargePR->processAccount: anAccount
self currentInput: anAccount.
(self monthsToProcess: anAccount) do: [:each | self processForMonth: each].
self clean
MonthlyChargePR->monthsToProcess: anAccount
^(anAccount unprocessedEntries collect:
  [:each | each whenCharged date firstDayOfMonth]) asSet.

```

每个月份都由`processForMonth:`来处理，如图7-12和程序清单7-16所示。

最后的事务是从输出账目到输入账目的，由于税款债务的原因，活动账目的生命周期将会增加。[149]

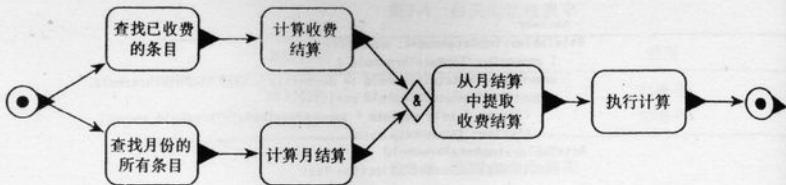


图7-12 处理月份的事件图

## 程序清单7-16 月收费记入规则是如何处理月份的

```

MonthlyChargePR>>processForMonth: aDate
  | inputToProcess totalToCharge |
  inputToProcess := (self inputBalance: aDate) - (self outputAlreadyCharged: aDate).
  totalToCharge := (self calculationMethod calculateFor: inputToProcess) -
    (self outputAlreadyCharged: aDate).
  Transaction
    newWithAmount: totalToCharge
    creator: self
    from: self currentOutput
    timepoint: aDate lastDayOfMonth
    to: self currentInput
    sources: (self currentInput entriesChargedInMonth: aDate).
MonthlyChargePR>>inputBalance: aDate
  ^self currentInput balanceChargedInMonth: aDate.
MonthlyChargePR>>outputAlreadyCharged: aDate
  ^(self currentOutput balanceChargedInMonth: aDate) negated
  
```

## 7.9 结论

150

这是一个非常简单的例子，所以很难从中总结出太多的结论。读者可能会很有说服力地提出异议，认为这个问题可以由相对简单的形式来解决，而不用我提出的这个框架。这个框架对于系统的可扩展性很有价值。实际的业务中可能有很多个实践，每一个实践又有很多个处理规则。使用这个结构，我们可以由账务实践给出新的收费计划。当我们建立新实践时，我们生成记入规则的新实例网络。我们这么做完全不必再编译或是重构系统，而它们还是好好地可以运行。当我们需要一个新的记入规则的子类型时，重构就不可避免，但这种情况很少见。

## 7.9.1 记入规则的结构

图7-13表示本章讨论的记入规则的泛化结构。抽象的记入规则类有一个抽象的processAccount方法。子类型分别实现processAccount。每个条目记入规则通过对条目调用另一个抽象方法processEntry来实现这个方

法。如需要，以后的子类型可以实现processEntry。对于白天/夜晚通话的区分离入规则的方法是硬编码的，而转换记入规则委托给费率表。示例表示：在一个有组织的结构中，抽象方法、多态性和委托的结合是如何提供能支持多种记入规则的结构。

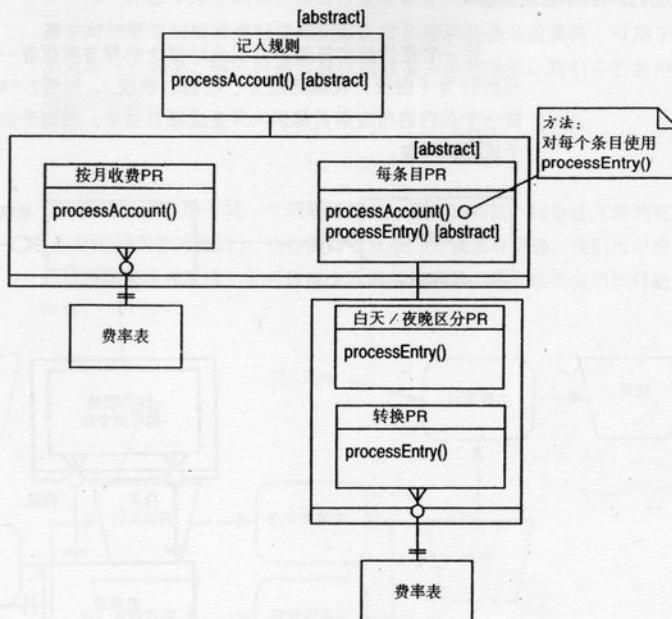


图7-13 记入规则的泛化结构

151

这也不是惟一一个我们能够使用的记入规则的结构。另一个可用方案是把计算费用的两个步骤合并为一个步骤。这样一条记入规则有两个费率表，一个是为白天通话的，一个是为夜晚通话的，这条记入规则既负责白天和夜晚的区分又负责费用的计算。

是否分解记入规则并没有一定之规。我们的基本目标就是在不必增加记入规则的子类型时能够建立新的实践。我们想要有尽可能小的记入规则的子集，因为这样可以容易理解和维护记入规则的类型。然而，我们也需要子类型来提供需要的所有功能，这样就可以用它们组成新的实践。当我们需要建立新的记入规则的子类型时，我们希望能尽快做出来。

更简单的记入规则将导致更复杂的实践，并且通常用途也更广泛。我乐意让记入规则在初始的时候很少的行为。如果我看不见有经常使用的记入规则的组合，我可能会给出一个功能更强的记入规则来替代这个组合。

### 7.9.2 什么时候不能使用框架

另一个使用这个框架的方案是对每个收费方案仅有一个类来处理所有的行为（白天／夜晚的区别、收费、收税），如图7-14所示。这个类将一个月内的所有条目都读入并生成账目清单。对每个收费方案都有一个这样的对象。

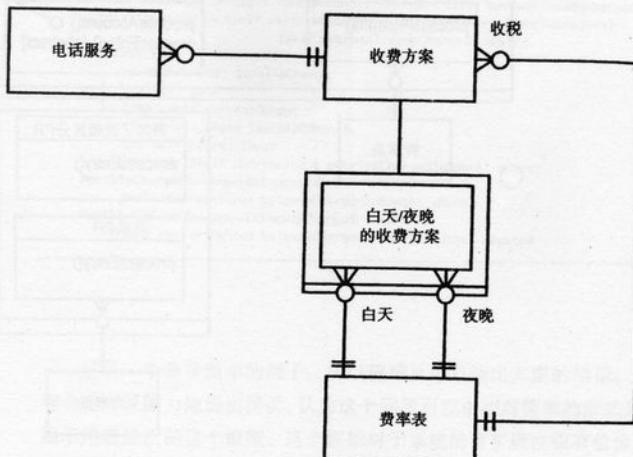


图7-14 使用收费方案

152 收费方案简单但是没什么灵活性。

对这种方式很有争议。虽然可能有很多收费方案，但是通常只会有几个简单的基本收费方法，它们可以实现参数化，就像转换记入规则因为很多不同的费率表而参数化一样。本问题中的这种类型可以由一个白天费率表、夜晚费率表和税率表来实现参数化。

关键的问题是收费方案中有多少个子类型。如果我们用十几个收费方案的子类型来表示所有的记账结构（即便可能有成百上千的实例），那

么使用收费方案类型就是有争议的。账目模型的长处在于：通过把记入规则和账目对象组合在一起，便能建立起收费方案的新的子类型的等价物。如果有一个大而经常改变的收费方案的子类型的集合，这就是一个很大的优点。

另一种思考的方式是把收费方案看作记入规则，它只需要一步就从基本时间账目到活动账目进行记账。使用账目仍是有价值的，可用于给出通话历史记录、输入时的收费以及收费方案的结果。我们将失去中间的汇总信息。

### 7.9.3 财务实践图

图形经常有助于将一个问题可视化。图7-15和图7-16是这方面的建议（当然多数是尝试性的）。这些图对于复杂的实践很有帮助。我们可以设想通过画图来建造系统，并可以减少需要的编程量，提高这类应用软件的生产率。

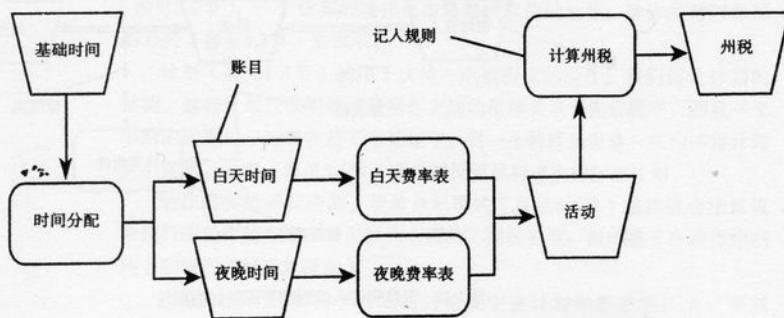
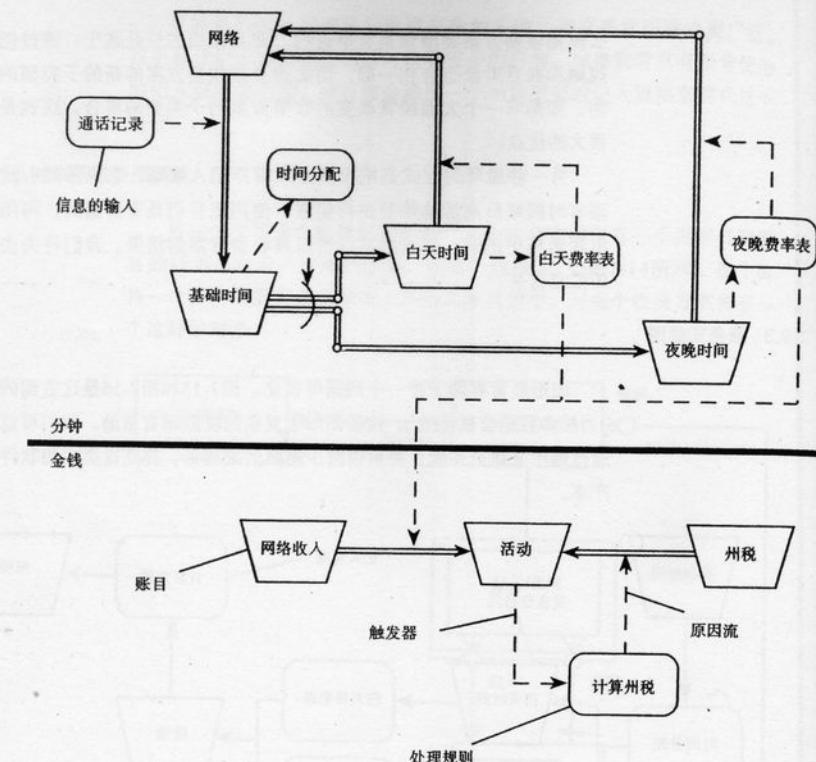


图7-15 处理规则和账目的简单图形化表示

对于每条记入规则，图中显示触发器和主要的输出账目，但隐藏全部事务流。

实践应该有一个图形形式的东西，简单但能表述关键信息。图7-15的长处就是很简单但表述了主要的触发和输出关系。但它不像图7-16那样表示全部统计条目的流向。如果你使用这些模式，我强烈建议你使用图形。从这里介绍的那些图开始，让图形标准进化成为最有用的标准（也让我知道它会变成什么）。



154

图7-16 账目和记入规则的更深刻的图形

本图显式地表达事务流。每条记入规则由单个的账目触发，并引发一系列流。流的方向表明条目被存取的方向。图形表示更多的信息，因而更为复杂。

## 参考文献

1. Beck, K. *Smalltalk Best Practice Patterns. Volume 1: Coding*. Englewood Cliffs, NJ: Prentice Hall, in press.
2. Gamma, E., R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.
3. Meyer, B. "Applying 'Design by Contract,'" *IEEE Computer*, 25, 10 (1992), pp. 40-51.

155  
156

## 第8章

# 计 划

计划是任何巨大尝试中至关重要的部分。许多管理者花费大部分时间来制定和跟踪各种计划。本章提供有关计划的一些基本模式。这些模式既描述单独的计划，又描述可以像标准过程一样被反复使用的方案。

可以记录一个领域内任何动作的实施。提议和执行的动作模式（参见8.1节）把一个动作的可能状态分为两个关键的子类型，这些子类型描述动作的意图和实际发生的情况。动作的结束同样被分为完成和放弃的动作（参见8.2节）。一个放弃的动作代表最终被取消的动作，动作的暂时保留由挂起（参见8.3节）来表示。

计划（参见8.4节）被用于支持一组被提议的动作。我们讨论计划的结构，这些计划应该能够记录动作之间的依赖关系和先后顺序，而且一个单独的动作可以出现在若干个计划中。后一个特性是安排一次性多重计划所必需的。方案（参见8.5节）就是被反复执行多次的标准计划。

动作的实施需要资源。资源分配模式（参见8.6节）描述适合于提议和执行的动作的方案。我们认为有两种不同的资源：消耗品（在动作中消耗）和资产（可反复使用）。

迄今为止，我们关于计划的讨论集中在计划和监控动作上，而忽视了动作的效果。我们最后讨论的模式处理输出和启动函数（参见8.7节），这个模式把本章的模式和第3章的观察和测量模式联系起来。这些函数允许我们表达：我们认为动作的成果（输出）是什么，方案可以完成的（输出函数）是什么，什么情况下我们需要开始一个方案（启动函数）。

计划是一个复杂的领域，本章的模式显然不能穷尽所有的计划模式。当然，其它章的模式也有类似情况。本章的模式产生于Cosmos临床过程模型[1]，这个模型的结构明显是针对医疗保健计划的。资源方面主要来源于Cosmos的开发者和使用者的一些私下讨论，以及NHS公共基本规约[2]的影响。

**关键概念：**提议的动作、执行的动作、计划、挂起、资源分配、资产、消耗品、临时资源、启动函数、输出函数

## 8.1 提议和执行的动作

任何计划的基础都是由人们所采取的基本动作组成。这里我们只能给出一个动作组成的概要描述。一个计划可以是粗粒度的，由大的动作组成，也可以是细粒度的，由小的动作组成。动作可以有一些基于什么人、什么时候和什么地方的属性。通过这些粗粒度的属性，一般只能提供最普遍的团体、时间参考和位置等方面的术语，如图8-1所示。

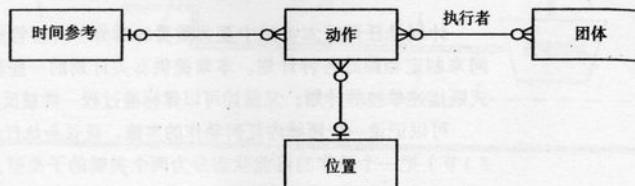


图8-1 动作属性

当制订和监控计划时，我们必须考虑一个动作可能会经历的许多状态。它可以被调度、分配资源、分配人力、启动和结束。一个状态转移图可以记录这些状态以及转移是如何发生的。很难制定有关状态转移的规则。调度一个动作并且给它分配资源可以清楚地按任何顺序进行。浅显的分析就可以推断出：一个计划不能在分配资源和调度之前开始。可有些动作在任何正式的决定来规定其时间之前就启动了，我们如何处理它们？我们可能会争论这样的动作是否会在启动之前的一刻调度，可这听起来更像是一个管理学原理的讨论而不是一个真正业务过程的反映。另一个可能出现的问题是部分资源分配。任何项目经理会告诉你，在现实世界中，任务经常会在所有需要的资源分配结束之前开始。在动作状态的描述中，我们如何来处理这种情况？

动作的两个重要状态是提议和执行，如图8-2所示。一个被提议的动作在某些计划中纯粹是一个建议。同样地，它可以通过增加一个时间参考来调度，增加团体来分配资源，寻找合适的位置来定位。这些变化可以在任何时间、以任何顺序发生。当计划开始，它就执行了。不但状态变化了，而且要创建一个单独的处于执行状态的动作对象。通过保留原始的提议动作，我们可以观察计划和现实的差异。举例来说，一个共同的差异是时间参考；然而，当计划文档最终变为动作的时候，任何属性都可能会改变。

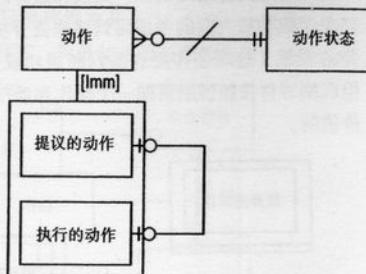


图8-2 计划和动作的基本结构

用独立的对象记录提议和执行，使得差异可以被跟踪。

**例：**在1997年7月1日，我决定为OOPSLA准备一个报告，可是我在3日之前不能着手做这件事。这些动作可以被描述为一个日期为7月1日的提议动作和一个日期为7月3日的执行动作。这个提议的所有其它属性都是一样的。

我们可以提供一个派生的动作状态属性，使我们更容易分辨一个动作正处于什么状态，而不需要遍历用来记录动作状态的各种结构。在这个阶段，也许并不一定需要这样。但是，当我们考虑以后附加的结构时，这就变得很有价值了。

在每日动作记录中，为了保持最大程度的灵活性，如图8-2所示，提议动作和执行动作之间的连接是可选择的。通常，如果没有执行，最好的计划也会逐渐变坏，而许多动作发生却没有预先的计划。我们应该抵制将最后一分钟计划变为合理化的诱惑。

**例：**Thursz医生为John Smith定制了一个全面的血液检查，可是没有找到患者不能完成这个检测。这就描述了一个被提议但没有执行的动作。如果这个患者在以后的日子重新启动这个检查，这就构造了一个新的动作提议。

**例：**Cairns医生被命令护理一个在火车上犯了急性病的妇女。这是一个执行的动作但是没有被提议。

## 8.2 完成和放弃的动作

迄今为止，我们考虑了动作如何被提议和开始，但没有考虑它们可以如何结束。很明显，动作要么成功，要么失败。问题是通常我们并不能很确定地判断动作是成功还是失败，尤其是在医疗保健方面。因而在本节里，

我们只考虑两种结局动作：完成和放弃。完成就是动作按照计划执行。任何关于成功或失败的考虑留到将来去分析（参见8.7节）。对于医疗保健以外的领域（这时往往更容易判断成功或失败），这个定义可能过于严格。但区别对待按照预期贯彻一个动作和动作达到其目的这两者，依然是很有价值的。

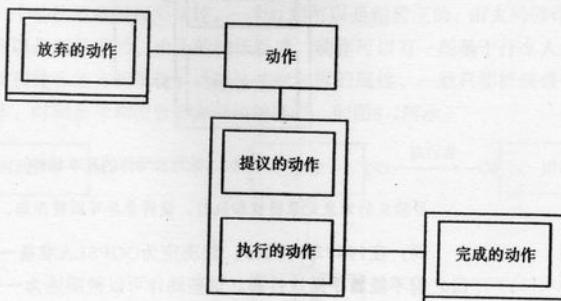


图8-3 完成的和放弃的动作

放弃是一个动作的完全的和最终的停止。它或者在动作开始执行之前，或者在动作开始执行之后。放弃一个提议的动作就是决定不执行它。

**例：**肾脏移植术通过使用一个捐赠的健康的肾来替换受到损害的肾，提供正常的肾脏功能。如果肾被安全地移植到患者体内，肾脏移植动作就可以被判断是成功的。如果以后这个肾脏出现排斥反应，并不能使移植过程的成功变为无效。这个移植过程依然完成了；只有在手术过程中发生了问题，才可能导致动作被放弃。

**例：**我决定从伦敦坐飞机到波士顿，期望下午2:00可以到达。飞机晚点了，所以直到晚上7:00我才抵达。这个动作仍然是完成了，因为我在那天到了波士顿。我所遭受的飞机晚点意味着这个动作没有成功。另一个提议的动作是在那个晚上吃晚餐，然而，却放弃了。

**例：**我的汽车不能启动，我确定问题是启动电机坏了。因而我提议并开始更换这个启动电机。就在刚开始，我发现问题的竟然是一段坏掉的线路，启动电机是好的。因而我放弃了更换启动电机的动作，尽管我对这样的结果并没有不满！

### 8.3 挂起

我们也可能推迟动作，以后再继续完成这个动作。当这种情况发生时，挂起就连接到某个动作，如图8-4所示。这个挂起在它的时间段里是正确

的（这个时间段也许没有截止）。如果一个动作在挂起的结束之后继续，这个挂起依然存在，但是却不再起阻塞作用，动作继续。

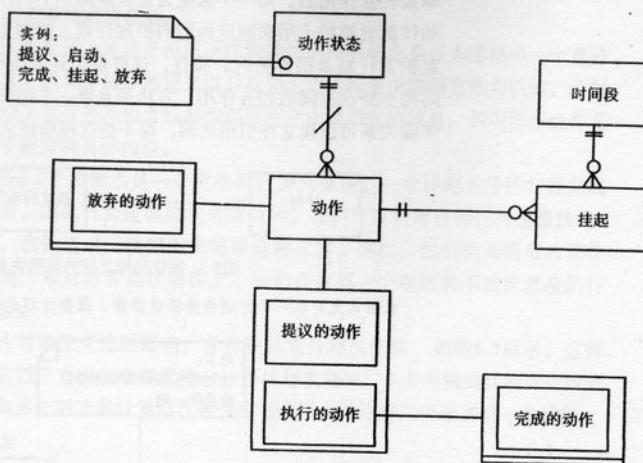


图8-4 动作挂起

挂起是一个动作的暂时停止。

[161]

因而一个动作如果与一个正在起作用的挂起连接，那么这个动作就被挂起。提议动作和执行动作都可以被挂起；挂起一个提议的动作和推迟一个动作的开始是等价的。

例：一个患者的名字在等候肾脏移植手术的名单上。这描述一个关于肾脏移植手术的提议动作。这个患者必须等到一个可用的肾脏。如果等候手术名单上的病人感冒了，医生就必须把这个患者挂起。但是移植手术并没有放弃，因为当感冒减轻时，患者还会回到等候手术的名单上来。挂起记录实质上是解释为什么在那段时间里，医生不能给那个患者更换一个合适的肾脏。

例：我有一个提议动作是洗盘子。这个动作经常被挂起很长一段时间，可是我从没有彻底放弃它！

## 8.4 计划

在最简单的意义上，计划是按某种顺序连接的提议动作的聚合。一个序列可以有许多种表达方式，可是一般大多数都表现为一种依赖，它表示一个动作在另一个动作结束之前不能开始。计划经常用依赖图来描述，比如在关键路径分析中。

图8-5是提议动作之间的一种依赖关系图。当动作总是作为一个单独计划的一部分被提议时，这个结构是很有用的。然而，在很多情况下，计划是相互作用的。当一个医生为患者提出一个治疗计划时，治疗计划中的动作就会被护士用来制定她们的护理计划。护理工作者这样为一个患者制定护理计划是很常见的。而且，这样制定计划是很重要的。图8-6显示的结构支持动作间的相互作用，方法是允许一个动作被多个计划引用，并且依赖关系可以建立在引用之间，而不是仅在动作之间。

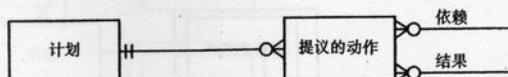


图8-5 提议动作之间的依赖关系

本图只允许在一个计划内的提议动作，调整计划是很困难的。

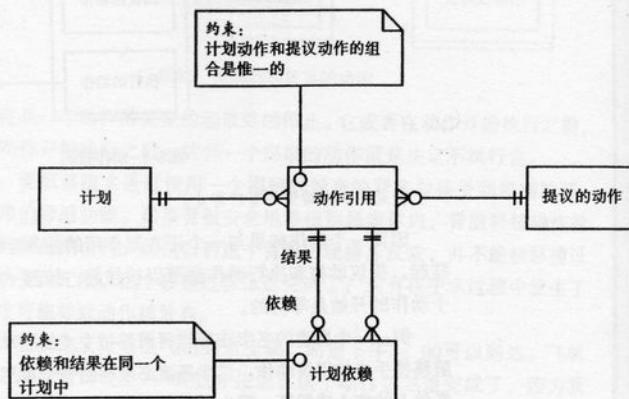


图8-6 一个由提议动作的引用组成的计划

这个结构允许动作被若干个计划引用。

**例：**一个医生为患者进行完整的血液检查。她核对了提议动作的清单，并且发现另一个医生已经提议了一个完整的血液检查作为他计划的一部分。这描述另一个医生的计划中有一个动作引用了完整血液检查的提议动作。可以创立一个新的计划，其中有一个新的动作引用这同一个提议动作。

**例：**我要去一个卖酒的商店为周六的晚餐买一些St. Emillion，为周日的聚会买一些Old Peculiar。去卖酒商店的动作在准备晚餐和准备聚会的

计划中都被引用。准备晚餐的引用有一个依赖关系，其中出席晚餐是结果，去卖酒商店是依赖。聚会计划的引用有一个依赖关系，其中开始聚会是结果，去卖酒商店是依赖。

这个在行为描述里使用动作和动作引用的概念是行为建模中一个普通的模式。这和子程序的定义以及在另一个子程序中调用它是类似的。子程序的定义部分并不包含如何在调用程序中使用它的信息。调用程序也并不知道子程序的具体内容。

图8-6中的模型是一个简单的行为元模型。一个计划是描述打算完成的行为，因此行为建模技术是适当的。我们可以使用任何行为建模技术。首先，我们通过元模型图来描述这种技术。其次，我们把元模型的动作联系到计划对象和提议动作上。我们会选择一个精细而不过度复杂的元模型。

计划常受变化的影响，并且被其它计划所代替，如图8-7所示。这种关联在两个方向上是多值的——当计划改变时，一个单独的计划可以被分裂并由多个独立的计划所代替，或者若干个计划可以被合并成一个计划。

[163]

代替



图8-7 替代计划

**例：**我有一个计划：在Garden of Eden买面包，在Bread and Circus买干酪。我用从Jac的快餐店买外卖的计划来代替这个计划。

我们可以认为计划是一个动作的子类型，如图8-8所示。因此我们可以提议一个计划（就是说，我们可以为计划做一个计划）并且监视一个计划，观察它是否完成。既然计划通常十分复杂，那么能调度并追踪一个计划是很有价值的。

我们可以认为一个计划是聚集动作的一种方法。例如，一个完全的血液检查可以被描述成一个计划，组成它的每一个测量都可以作为里面的提议动作。然而，这是一种非常笨手笨脚的表示法。图8-8所示的结构允许将一个动作分解为几个组成动作，但是它允许采用两种方法来描述动作成为一个更大动作的部分：使用组成关系适合于简单的情况，而使用一个计划适合于更复杂的情况。我们可以对由组成关系构成的层次结构进行约束，使得组成关系只用于简单的情况。

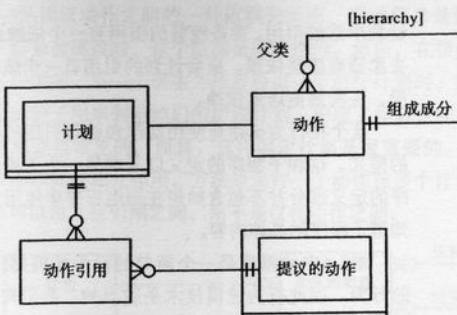


图8-8 计划作为动作和组成动作

[164]

我们给计划做计划，并且可以有复杂的动作而没有一个清楚的计划。

## 8.5 方案

一个组织的标准运转过程是普遍的动作，它要运行许多次，每次都采用同样的方式。我们可以用方案来描述这些普遍的动作，而方案的构造与我们为计划所使用的构造相似，如图8-9所示。计划模式，就像本书中其它的模式一样，可以被划分为知识级和操作级。操作级描述日常计划和动作。在知识级是方案，它描述支配操作级的标准过程。

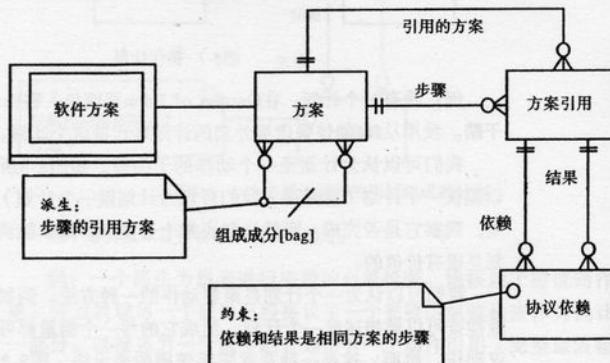


图8-9 方案的结构

本图的结构类似于前面描述计划所使用的结构——一个简单的行为元模型。

在这个结构中，知识级和操作级之间有一些很有意思的差别。在知识

级上使用层次结构是很少有用的。方案可以被许多其它的方案引用；很难想像对这种引用进行限制会起作用。在需要把动作按某种有规则的方式聚集的时候，我们经常能有效地把一个动作描述为另一个动作的一部分，例如测量可以作为一个完全血液检查的一部分。

在知识级上，提议动作和执行动作之间是没有差别的，在一个计划和另一组动作之间也没有有价值的差别。方案的成分总是一个bag（因为一个方案可以在另一个方案中被执行不止一次），但是一个计划的提议动作总是形成一个集合（因为你不能两次执行同一个动作，但是两个动作可能符合同一个方案）。

165

一个方案的描述不需要详细到其组成成分。一个方案可以仅仅是一个名字、一段描述文字、教科书中的章节、网页、甚至是某人进行的一次独特的复杂的外科手术过程的录像。方案引用可以只描述其组成成分，而不描述组成成分间的任何依赖关系。一些方案可以被完全编写为计算机代码，在这种情况下，方案变成了一个软件。（软件方案是一个以软件方式编码的方案，而不是通信协议中的协议。）

从一个复杂的方案中，我们可以按两种方式形成动作。最简单的方法是使用组成关系。当子动作都在一个适当限定的时期内发生，并且没有人需要共享这些成分动作时，这种技术的效果很好。我们首先为整个方案产生一个提议动作，如果我们不得不指出那些特殊的性质（例如定时和资源），我们就要简要地说明其中的子动作。（如果有许多这样特殊的性质，那么我们应该使用一个计划。）如果所有的动作在大约同一个时间被相同的团体完成，父动作就足够了。为每一个子方案的引用创立一个子动作；也就是说，一个方案在父方案中执行三次将会产生三个子动作；动作间的任何依赖都会与方案中的表示吻合。

当我们想监控什么时候和怎样执行单独的方案步骤时，计划能够提供更灵活和精确的跟踪，因此此时我们更倾向于使用计划。这些关系如图8-10所示。另外，一个计划允许它的子提议动作可见并被其它计划所共享。计划的一个重要特征是：当它们可以复制方案的依赖时，它们也可以定义新的依赖，从而忽略方案中的某些依赖。这种能力在技能性的职业（如医疗保健）中是很重要的，比如，我们经常不得不为考虑具体患者的需求而修改标准方案。我们经常需要一次性的计划，这些计划是基于方案的但不是简单的复制。

从一个方案形成动作通常意味着使用比方案层次更高的计划，而使用组成关系则意味着层次更低。

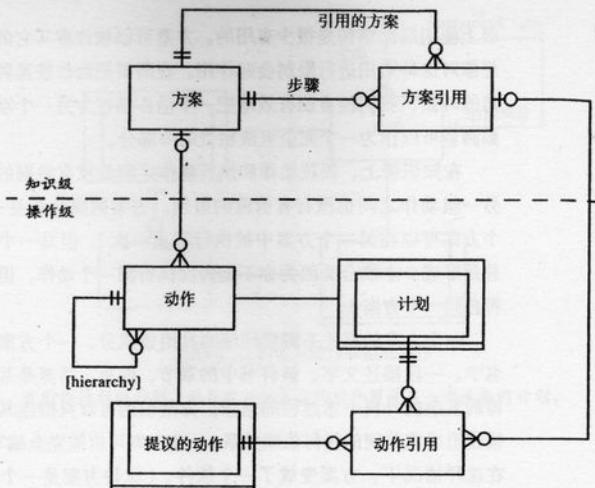


图8-10 动作、计划和方案间的关系

### 计划和方案图

我们也可以把一个计划描述为提议动作的有向非循环图（DAG）。图中的弧对应动作引用的依赖关系。每个计划有自己单独的图结构。我们可以如图8-11所示那样简洁表示。在本质上，这是另一种关联模式，属于第15章描述的风格。

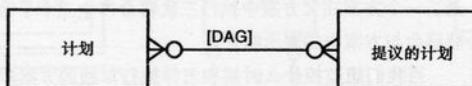


图8-11 计划由提议动作的有向非循环图表达

然而，要将这个概念应用到方案，我们不能为子方案形成DAG图，而要为方案引用形成DAG图，如图8-12所示，因为一个方案在另一个父方案中可以出现不止一次。对于图8-6所示的唯一的约束，与这里考虑的不是同一个问题。因此，DAG关联模式的基本形式包括依赖类型（以及约束）和DAG图中的元素在图中只能作为一个节点出现的要求。

如果我们为计划结构使用一个图，那么我们就失去在计划引用和方案引用之间建造关联的能力，如图8-10所示。自然地，我们可能还要有DAG

的版本作为一个派生的映射；派生将会包括如何派生图的弧。

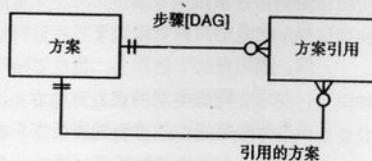


图8-12 使用DAG的方案

## 8.6 资源分配

计划的第二个主要部分就是分配资源。提议动作和执行动作的一个主要差别在于它们如何使用资源。一个执行的动作实际上将会使用分配给它的资源。一个提议的动作将会预定一些资源。图8-13显示作为一定数量某种资源的资源分配。资源只能被一个动作预定和使用。

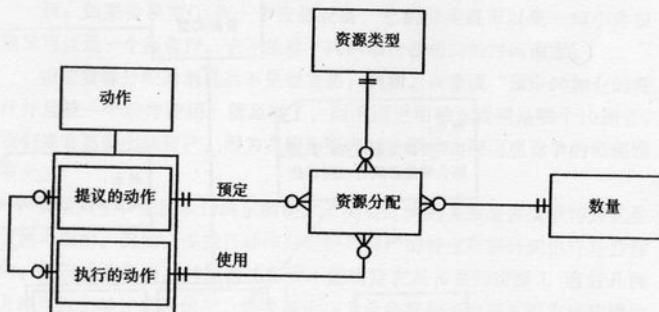


图8-13 使用资源的动作

提议的动作预定资源，执行的动作使用资源。

资源是多种多样的。最开始、也是最明显的资源是消耗品。消耗品是如药品、针和原材料这样的一类东西。消耗品只能被使用一次并且由动作来使用它们。通常消耗品要按数量来请求。

例：10加仑的橙汁的分配是10加仑的数量和橙汁这种资源类型。

例：有一个特殊的髋骨替换手术，要预定4个单位的袋装红血球（血液），可是只会使用两个单位的袋装红血球（血液）。这样替换会表现出两次关于袋装红血球类型的资源分配。一个是关联到提议髋骨替换的4个单位数量的资源分配；另一个是关联到执行髋骨替换的两个单

[168]

位数量的资源分配。

一些资源是不能消费的，如装备、房间和人。人是不能被一个动作所消费的（虽然我在写完本书之后仍对这种说法感到疑惑）。然而，我可以说一个人的时间被用完了。在这种情况下，资源类型是人，并且数量是时间。因而在这个动作上，我花了5个小时是一次我的5个小时的资源分配。

这个资源类型的观点可能有点太具体化。存在于知识级的资源类型，更典型的是指一类事物而不是某个事物本身。具体举个例子，我进行的工程需要一个有经验的面向对象建模员5个小时，而不是具体我这个人5个小时的工作。尽管有些人单独凭本身的头衔作为资源类型就足够了，但是我们这些大多数的凡人只是许多人中的一个。

因此，在计划编制中，需求的描述形如“我们需要面向对象建模员的5个工作小时”。在编制计划的某些阶段，实际的结果是预定了我的5个小时，这是一个特定的资源类型的实例。这意味着资源分配的两种级别：一种是通用的指定类型，而另一种是具体指定的资源分配。

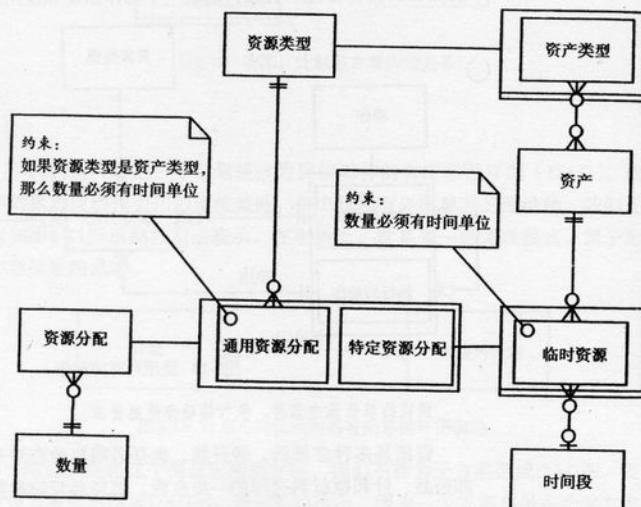


图 8-14 资产的资源分配

特定分配要指定具体使用或预定的个人资产。通用分配只是指定资产的类型。

在图8-14中，具体的资源称为资产，资产又分为多种资产类型，这也是一种资源类型。特殊资源类型和普通资源类型之间的差别是前者分配的

[169]

是资产而后者分配的是资源类型，而资产的资源类型就是指资产的类型。临时资源是资产的特定的资源分配。此时不仅要分配时间的数量，而且要分配一个特定的时间段。这个时间段可以从使用或预定临时资源的动作派生，也可以与动作相对分离。

**例：**一个建模会议被安排在一间小会议室中进行，用时两个小时。最初，这描述一个预定通用资源分配的动作。这个通用资源分配的资源类型是小会议室这种资产类型。通用资源分配的数量是两个小时。后来，实际的会议室是预定了Q9。这重新分配（或者说替换）了通用资源分配到一个临时资源是两个小时的资产Q9。如果这个提议动作是会议预定在星期四下午2:00到5:00之间进行，那么时间段就是从Q9的分配中派生的。如果会议的最后一个小时要在酒店中举行，那么星期四下午2:00到4:00的时间段就被连接到了临时资源。

一个资产允许有几个资产类型。这种资产的多重分类在描述那些可以做几件事情的资产中是很重要的，尽管那时不一定需要同时做这几件事。

**例：**如果会议室Q9有一个投影设备，它的分类既可以是一间小会议室又可以是一个报告厅。它不能被不同的动作在相同的时间预定。

特定资源分配对消耗品不是很重要。例如，通常说“预定10加仑的橙汁并且被一个动作使用”就足够了，而不用更明确地说明是哪个10加仑。我们通常需要明确资产，因为在使用资产时，团体之间出现竞争的可能性很大。

在这点上，如图8-13所示的动作子类型之间的关系是否应该特殊化是值得考虑的。例如：说执行动作只能使用资产的特定资源分配也许是合理的。假设这儿需要某种东西（但我不能确定它是否真的需要），有好几种方法可以来描述这种情况。这是展示一个业务规则可以用不同方法建模的好例子。

首先，最显而易见的办法就是引进结构性的约束。既然这样，我们可以使用一个像“执行动作不能使用通用资源分配，这种方法分配的资源类型是资产类型”这样的规则。这种急切检查是一种强迫实施业务规则的强制性方法。也就是说，你不能记录一个破坏政策的情况。

然而，这是一个太强硬的方法，以至于不能适应很多情况。有时候，允许记录一个破坏政策的情况是有意义的，而稍后还有一个独立的检查阶段。这种消极检查可能由一些执行动作（例如isConsistent（））上的操作来完成，并且如果遵循了业务规则，操作返回“真”。在一个充分的约束可能不会从头开始有效时，这就在控制状态方面提供了更大的灵活性。这个不完善的信息被记录下来，并且提供一个检测的手段。

消极检查的巨大优势在于它分离了问题的解决和信息的记录。记录这个信息的人们可以在那个时候尽力记录，然后或者由他们或者由一个有资格的人在日后整理这些信息。如果在捕获信息时就可以容易地解决问题，那么急切检查就更好。

是否允许资产的通用资源分配来进行执行动作取决于具体的问题。如果领域的具体要求满意于知道要花费面向对象建模员的两个工作小时，而不需要知道具体是哪一个建模员，那么就允许资产类型的普通分配。这个问题依赖于资产类型。例如，医院的政策可能规定必须明确指明顾问的具体分配，但护理员可能只需要进行一般分配。

如果我们关注从需要跟踪的有限储备中移走消耗品的情况，就可以对消耗品使用具体的资源分配。在这种情况下，我们需要说消耗品从一个特定的可消费的财产中减少了，如图8-15所示。依靠资源追踪过程，财产可以按照多种方法来组织，这些方法不是我要在这儿考虑的。然而，值得一提的是可以将财产看作一个账目，可以将资源看作其中一个条目，这样就可以用6.14节的方法描述。

171

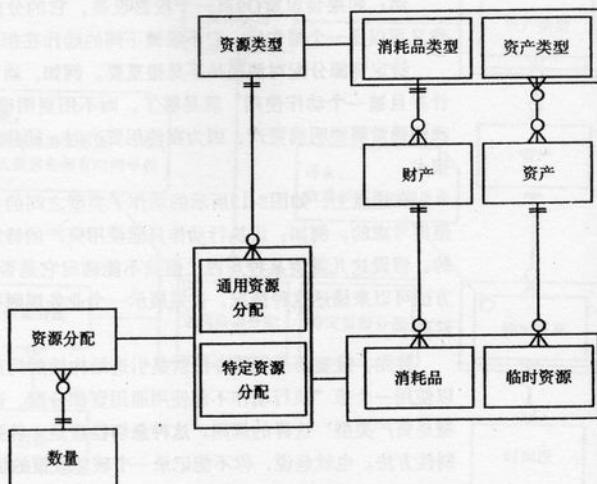


图8-15 允许特定的消耗品分配

我们也可以使用资源分配来描述一个方案执行所需要的资源。此时，我们使用通用资源分配。

例：为了制作薄煎饼（印度面包），你需要1/4杯的面粉，1/8杯的水，1/4大汤勺的油和一小撮盐。这可以描述为4个通用资源分配。

## 8.7 输出和启动函数

在本节里，我们使用第3章中发展出的概念来考虑我们形成一个计划的原因和如何判定计划的成功。

计划由观察开始，当然，观察可以是假设或者推理。类似地，它们的输出是连接到计划中动作的观察，如图8-16所示。如同观察的许多其它方面，不同的执行者看到的输出连接是不同的。因而一些团体可能看不到作为一个动作输出的观察，而另外的团体却可以。我们将通过不同的执行者的不止一个的观察来记录这种情况。

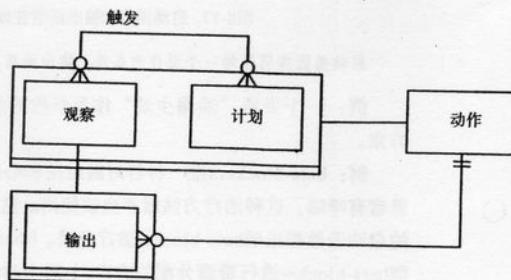


图8-16 观察、计划和动作之间的连接

**例：**John Smith 因为有了典型的糖尿病症状：体重减轻、口渴、尿频，所以来到他的医生这里看病。由于这些观察的触发，医生制定了一个计划。这个计划包括一个提议动作，来进行患者的血糖测量。

**例：**在经历了不好的销售状况之后，一个公司决定通过加强代理和降低价格来提高销售额。一些分析专家认为销售额的增长是销售代理增加的结果，另一些分析专家则认为主要是降低价格的结果。每一个团体将进行分离的观察，每个观察都连接到不同的动作。[172]

**注意：**观察是动作的子类型。它们可以被调度、定时，它们可以指定执行者并且成为计划的一部分。它们的额外行为是确定一个观察的概念或者测量一个现象的类型。

一个类似的连接集合出现在知识级，使用启动函数和输出函数，如图8-17所示。一个启动函数包含信息，这些信息应该可能触发一个方案的使用。在下面这个联合功能的例子中，这个模型记录观察的概念和方案，它们用作启动函数的参数，但是却不指明它们是如何结合的。这样做的意图是说明不同类的启动函数有不同的方法来结合两者。

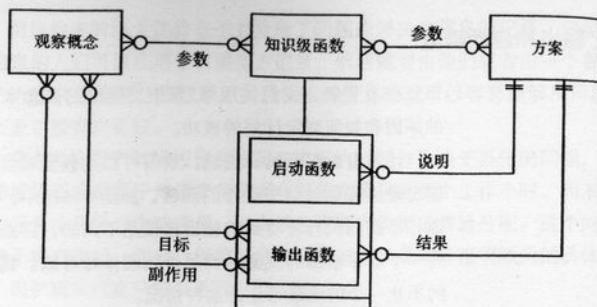


图8-17 启动函数和输出函数在知识级的使用

启动函数指明开始一个动作的条件，输出函数指明目标和副作用。

**例：**一个带着“油箱少油”作为参数的启动函数指明了这个加油的方案。

**例：**beta-blockers是一种针对高血压和心绞痛的治疗方法，但是如果患者有哮喘，这种治疗方法就不应该使用。这将导致三个启动函数，所有的启动函数都指明beta-blocker治疗方法。（beta-blocker疗法是关于资源类型beta-blocker进行资源分配的方案。）两个启动函数，一个的参数是高血压，另一个的参数是心绞痛，并且没有其它症状需要处理。第三个参数是哮喘并且是一个逻辑上的否定。（我们可以有一个与指明相反的启动函数的子类型，但是它实际上是完全依赖于处理参数的方法。）

输出函数的操作类似。同样，输入是方案和观察概念的组合。结果是两个观察概念的集合。一些观察概念描述使用方案的目标，就是描述方案的意图的输出。另一些观察概念是副作用。一个方案可能有许多结果。这可能会反映出在那个时候患者在其它方案或者观察概念中的情况。这些作为参数从知识函数继承而来。

**例：**降价是一个以增加市场占有率为目地的输出函数，它的副作用是减少了每一单位商品的销售收入。

**例：**肝脏移植的方案有一个以得到健康的肝脏为目标的输出函数，副作用是器官排斥和输送胆汁的困难（因为胆管变窄了）。启动函数也可能包括这些产生条件的可能性的信息。独立的输出函数可能有相同的目标和副作用作为参数，但是影响程序的还有描述疾病的参数。由于疾病参数的存在，这些独立的输出函数指明目标和副作用的不同可能性。

## 参考文献

1. Cairns, T., A. Casey, M. Fowler, M. Thursz, and H. Timimi. *The Cosmos Clinical Process Model*. National Health Service, Information Management Centre, 15 Frederick Rd, Birmingham, B15 1JD, England. Report ECBS20A & ECBS20B <<http://www.sm.ic.ac.uk/medicine/cpm>>, 1992.
2. IMC. *Common Basic Specification Generic Model*. National Health Service, Information Management Centre, 1992.

[174]

## 第9章

# 交 易

本章着眼于货物的购买和销售，这些货物的价值与变化中的市场条件有关。根据为一个银行构建交易系统的经验，本章从银行买和卖相同货物这两个角度来看购买和销售。银行必须了解这些交易方式在不同情况下实际效果的价值。

每一次交易都用一个合同（参见9.1节）来描述。合同能购买或者销售货物，并且它对于需要约束交易双方的商业活动来说是很有用的。我们可以通过使用一个合同夹（参见9.2节）来表示多个合同的实际效果。我们以这样一种方式来设计合同夹，使之易于汇集，并且以不同的方法选择合同。我们给合同夹一个独立的对象——合同夹过滤器——来定义选择标准。合同夹过滤器定义一个可以被不同的子类型实现的接口。这个结构提供更灵活的简单和复杂的选择标准。这是一种用灵活方式确定汇集的有用技术。

为了了解合同的价值，我们需要了解被交易的货物的价格。货物经常依据它们是要被买还是被卖来定不同的价格。这两种不同的定价行为可以由一个报价（参见9.3节）来捕捉。

在易变的市场中，价格可能会迅速变化。交易人员需要依据可能的改变范围来给货物确定价格。场景（参见9.4节）把表现市场状态的相关条件组合起来构成市场的估价环境。以便场景可能很复杂，并且需要一种方法来定义它们的结构，以便可以在不同的时间，按照一致的方式，使用相同的场景结构。场景在价格变化复杂的任何领域中都是很有用的。

本章基于这样一个项目：为一家主要银行开发外汇派生交易系统。

**关键概念：**合同、合同夹、报价、场景

### 9.1 合同

最简单的金融交易种类就是从另一个团体购买一些交易物。交易物可以是股票、日用品、外汇或者其它普通的交易项目。一个基本的起点是如图9-1所示的模型。这个模型有一个合同的交易与另一个团体有关，该团体我们称之为对方团体，合同涉及一些交易物的数量。只显示一个简单的

交易物，尽管严格地说所有的交易都包括两种交易物——一种是为了和另一种交易。对于大多数的市场，一种交易物总是在当前的一个市场中使用的货币。价格因而被描述成一个货币对象。货币是一种单位为币种的数量（参见3.1节）的子类型。

在外汇市场中，交易物是汇率。这看起来可能有点奇怪，可是所有交易物确实都是汇率。一个以道指出售股票的合同事实上是一个用股票兑换美元的合同。在大多数情况下，通过说这个交易物被兑换成一定数量的货币，更容易描述，可是对于汇率来说，更好的做法是使两个货币都依附于交易物上并让价格成为一个简单的数字。

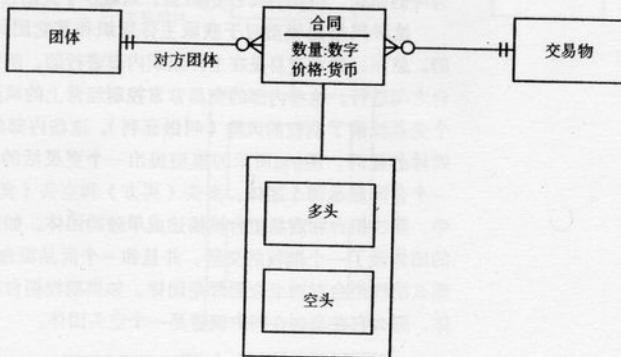


图9-1 合同的简单模型

交易物的总数是和对方团体进行交易的具体内容。多头和空头是分别表示买入和卖出含义的术语。惟一的对方团体限制了可以被描述的合同。

[176]

术语“多头”(long)和“空头”(short)是供交易人使用分别表达“买入”和“卖出”含义的术语，图9-1用子类型符号显示“多头”和“空头”的区别。另一个替代方法是采用布尔属性isLong。两种方法都是可以接受的，可是我更喜欢图9-1在概念建模方面具有的明确性。子类型化和布尔属性在概念模型上是等价的；子类型化(Subtyping)并不意味着子类化(Subclassing)。但是，在可执行建模技术中(当子类型化确实意味着子类化时)，除非多头和空头的行为确实不一致(有可能仍然没有区别)，否则图9-1就是不合适的。一个接口模型可以以任一种方法实现。14.2节描述无论是使用子类还是标记，如何进行这种变换才能够保持相同的接口。

**例：**Megabank以每股30美元的价格出售了1000股Aroma Coffee Markers的股票给Martin Fowler。这是一个空头合同，它的对方团体是

Martin Fowler, 交易物是Aroma Coffee Makers的股票, 数量是1000, 价格是30美元。

例: Megabank用200万美元(USD)从British Railways买了100万英镑(GBP), 这是一个多头合同, 对方团体是British Railways。数量是100万, 价格是2, 交易物是GBP/USD。此外, 它也可以是一个空头合同, 数量是200万, 价格是0.5, 并且交易物是USD/GBP。

例: Northeast Steel出售了10 000吨钢材给Chrysler。对于Chrysler, 这是一个多头合同, 对方团体是Northeast Steel。交易物是钢材, 在这种情况下, 合同额10 000吨用数量表示。(替代方法是: 允许交易物是单位为吨的钢材, 可是对其它 的数量, 这就少了灵活性。)

这种风格的模型对于获取主体组织和其它团体之间的交易是有好处的。然而, 通常交易是在主体组织内部进行的, 例如在期权柜台和商品柜台之间进行。这些内部的交易常常控制经营上的风险。一个普通例子是一个交易抵消了期权的风险(叫做套利)。这些内部的交易带来了谁是内部团体的疑问。图9-2所示的模型提出一个更灵活的方法来回答这个问题。一个合同显示两个团体: 多头(买方)和空头(卖方)。在这种描述方法中, 期权柜台和商品柜台被描述成单独的团体。如果期权柜台和一个外部的团体做了一个期权的交易, 并且和一个商品柜台做了一个套利的交易, 那么期权柜台对两个合同都是团体。如果期权柜台在期权交易中是多头团体, 那么它在套利合同中就将是一个空头团体。

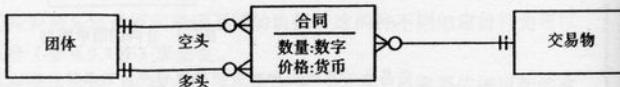


图9-2 通过分离的关系来显示买方和卖方

有两个团体支持内部交易, 完全的外部交易, 以及在主体组织内部进行的不同团体的交易。

图9-3用稍有不同的方法描述一个类似情况。此外, 允许使用两种关系描述内部交易。然而, 这儿有一个主要团体和对方团体的概念, 这比多头和空头要好。当主银行团体和一个外部组织做交易时, 主银行团体一直是主要团体。在内部交易中, 主要团体和对方团体之间的选择是随意的, 虽然按惯例通常是主要团体发起交易。多头和空头的子类型是主要团体所看到的交易的本质。

依照最初的分析, 图9-3所示的模型看起来不比图9-2所示的模型更有价值, 因为它增加了一对子类型却没有任何大的优势。当然, 如果数据结

178

构变得更复杂，数据建模时就会拒绝这种做法。在OO建模术语中，重要的问题是接口。是提供一些操作请求主要团体、对方团体、多头或空头合同的业务更有用呢，还是拥有一个多个头和空头的团体更有用？可能图9-4所示的模型是最好的，它从本质上来说提供所有两个接口。决定因素是从用户的观念来看，什么是最有用的。对于我们的实例系统，图9-3的模型对于交易人比图9-2的模型更有意义，它在构造软件方面更有用，尽管图9-4的接口是最后提供的。

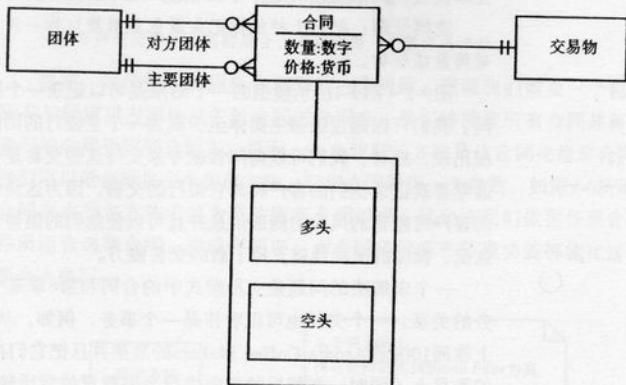


图9-3 对方团体和主要团体

这是一个比图9-2更简洁的图，但是它能更好地支持交易人的观点。

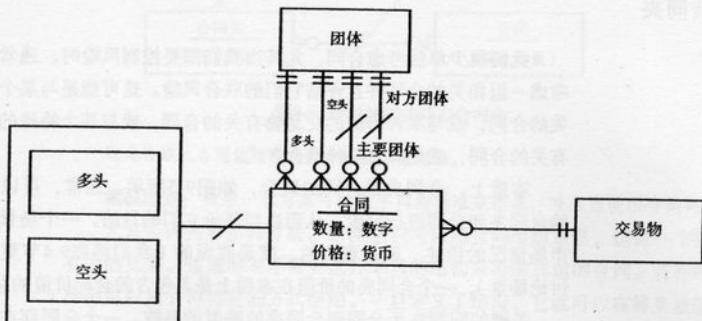


图9-4 使用四个团体的映射

本图通过派生双重的元素反映各种观点。

**建模原则：**当可以提供不止一个的等价特征集合时，就挑选领域专家最满意的一种方法。如果领域专家感到两种方法都有价值，那么就两种都显示并且把其中一个标记为派生的。

在图9-4中选择什么来进行派生是十分随意的。我们可以同时进行多头和空头的派生映射。这个模型不能限制实现者使用任一种实现方式。可以讨论的是你可以不派生任何东西，而只使用简单的规则（例如，如果合同是空头，那么空头团体和主要团体是相同的对象）。我更喜欢表示出派生来使交互关系更加清楚，可最终这只是对建模方法的个人喜好问题。

**建模原则：**把一个特征标记为派生是对接口的一种约束，但不会影响基础的数据结构。

图9-2~图9-4所示模型的一个结果是可以记录一个跟主银行无关的合同。我们可以通过强制主团体至少成为一个主银行的团体来避免这样的情况出现。或者，我们可以询问领域专家支持这些交易是否有价值。销售商通常喜欢记录他们的客户和其它银行的交易，因为这提供给他们关于他们的客户可能冒的风险的侧面信息并且可以使他们的销售合同得到改善。在这里，模型的灵活性就支持了新的交易能力。

一个未解决的问题是交易模式中的合同和第6章某个账目模式中的事务的关系。一个交易也可以看作是一个事务，例如，从Megabank的账目上收回1000股Aroma Coffee Makers的股票并且把它们存入Martin Fowler的账目上（同时，在相反的方向进行合适数量的货币转账）。交易和事务都是有用的，可是目的不同。建模时，需要更多的需求才能进一步探究这两者之间的关系。

## 9.2 合同夹

我们很少单独考虑合同，尤其当我们需要控制风险时。通常，银行会考虑一组相关的合同并且评估它们的联合风险。这可能是与某个交易人有关的合同，或与某种特殊的交易物有关的合同，或与某个特殊的对方团体有关的合同，或是其它某种组合方式。

本质上，合同夹是合同的聚合，如图9-5所示。通常，可以通过场景给合同夹和合同进行定价，从而达到评价它们的目的。一个场景就是某种市场情况的描述，或是现实的，或是假设的（我们将在9.4节更加详细地讨论场景）。一个合同夹的价值在本质上是其包含的合同价值的总和。

关键的问题在于合同到合同夹的映射的基数。一个合同存在于多个合同夹中是否有意义，依赖于我们怎样产生和使用合同夹。如果合同夹是一个交易人的工作簿，那么合同存在于管理此交易的交易人的合同夹中。然

而，这并不允许将和一个特定对方团体进行的所有交易都放在一起考虑。因而允许一个合同存在于许多合同夹中看起来是有优势的。因此，可以按照不同的观点构建合同夹来控制风险。

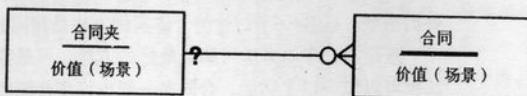


图9-5 合同夹介绍

一个合同夹是若干合同的聚合，可以作为一个整体来评估。

然而，这样使用合同夹会产生另一个问题。假设我们需要一个包含所有与特定对方团体有关的合同的合同夹。我们将搜索所有合同并将搜索到的合同指定到合同夹。然而，一个较好的方法是让合同夹指定合同。我们可以给合同夹一个布尔方法，它把合同看作一个参数，如图9-6所示。[180]

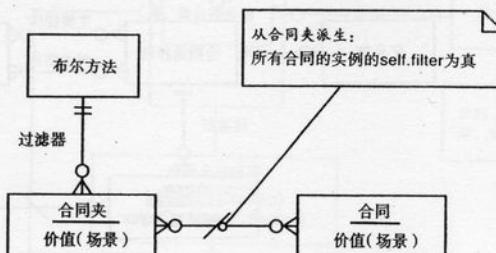


图9-6 带过滤器的动态合同夹

这允许通过合同的性质来隐含地描述合同夹。

**建模原则：**如果一组对象可以用不同的标准来组织，就应当使用合同夹。

允许合同夹有多种方法，以便它们可以用合同组织自己，这是一个很有力的见解。这意味着不需要选择单独的结构来考虑成组的合同。各种结构都可以用一种特定的方式使用。一旦定义了结构，它就可以在将来被想起并被使用，它的内容也可以被定期更新。可以在任何时候定义结构，甚至在将合同放在一起很久以后再定义这个结构。事实上，我们是在进行一个查询，并且对象聚合的结果本身又是一个对象。

这个布尔方法怎样实现？一般来说，当合同作为一个参数给出时，方法可以是任何一块返回真或假的代码。Smalltalk程序员可以认为把一个单独的参数块赋值给合同夹的一个实例变量就会得到想要的能力。C++程序员也可以使用相同的原则，尽管这更加难以掌握，因为C++需要一个编译好的函数。这和6.6节讨论的个体实例方法是相同的问题。

[181]

理论上，布尔方法可能是最好的方法，可是实际上有一个更简单的方法也可以完成以上功能。合同夹一般由许多合同组成，这些合同具有某些属性，包括对方团体、交易人（主要团体）、交易物和交易的日期等方面。我们可以合并这些属性到一个特殊的合同选择器对象，如图9-7所示。合同选择器不像布尔方法那样通用，它只能处理限定范围的过滤条件。然而，它却很容易建立；使用者可以通过一个合适的使用者界面轻松地配置它。如果我们使用合同选择器来处理用户需要的大多数合同夹，我们就能减少相当数量的编程工作量。

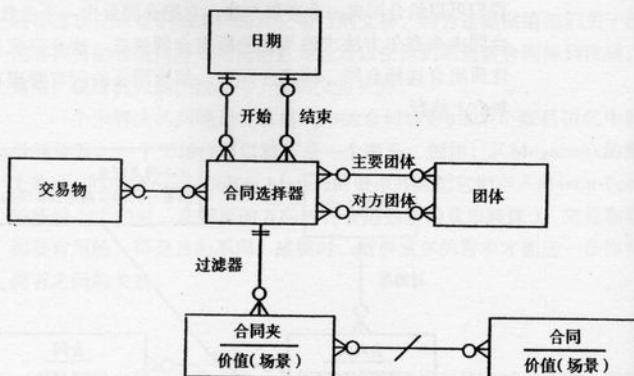


图9-7 合同选择器

注意：这是一个参数化方法的例子（参见6.6.4节）。它不能实现所有可能的合同夹，但是与更通用的实现方法相比，它可以更容易地实现实际中使用的大多数合同夹。

**例：**一个合同夹由所有的交易组成，包括出售给John Smith的Aroma Coffee Makers股票。这个合同夹有一个过滤器，Aroma Coffee Makers股票作为交易物，而John Smith作为对方团体。

我们不必一定要在合同选择器和布尔方法之间为我们的过滤器做出选择。我们可以有通过使用如图9-8所示的模型来表示的最好方法。这个模型把布尔方法和合同选择器都抽象成一个单独的、抽象的类型——合同夹

UML

182

过滤器。这允许我们对简单情况使用合同选择器，而对更复杂情况使用一定范围的硬编码得到的过滤器。我们可以轻松增加其它的合同夹过滤器。这是一个策略模式[1]的例子。

**建模原则：**当把一个过程看作类型的一个特征时，应该给这个过程提供一个抽象的接口，使得实现能轻松地通过子类化而改变。一个通过硬编码得到的实现是一个子类，不同的参数驱动方法是其它的子类。

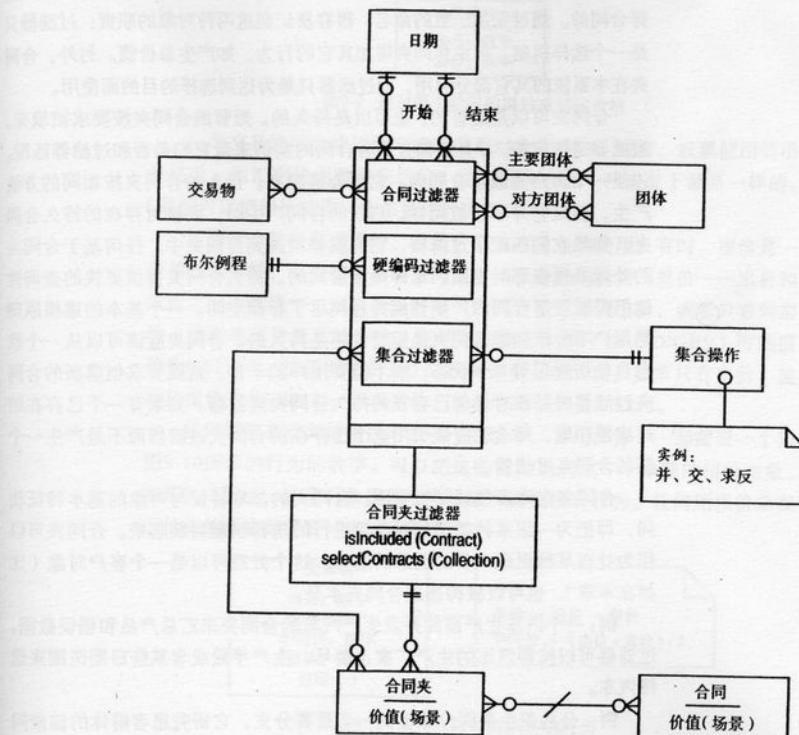


图9-8 提供多种合同夹过滤器

这个模型提供灵活性来处理复杂情况并提供简单的参数化方法来处理简单情况。这是一个策略和参数化实现的组合（参见6.6节）。

合同夹过滤器上的选择操作处理一个合同的聚合并且返回另一个合同的聚合。对于每一个输入聚合中的合同，选择操作求出isIncluded的值，

183

如果为真，把它加入结果集。合同夹过滤器的子类覆盖`isIncluded`函数来提供特殊的行为。一个合同夹可能使用`isIncluded`函数来检查单个的合同。

还有一点要说明的是：关于“合同夹过滤器”和“合同选择器”的命名。和我一起工作的同事发现这两个术语之间的区别在实践中十分有价值。一个选择器选择的应该是其前面出现的事物；因而一个合同选择器是用来选择合同的，它返回一个合同的聚合。一个过滤器选择的是一些其它类型的事物，而其前面出现的事物是其服务的对象；因此合同夹过滤器是为一个合同夹选择合同的。通过采用一致的命名，很容易记起这两种对象的职责：过滤器只是一个选择机制，而由合同夹增加其它的行为，如产生总价值。另外，合同夹在本系统的其它部分引用，而过滤器只是为达到选择的目的而使用。

合同夹可以是短暂的，也可以是持久的。短暂的合同夹按要求被填充。过滤器是指定的，并且要检查所有合同的实例来看它们是否和过滤器匹配。一旦一个客户完成了合同夹，它就会被抛弃。持久的合同夹按相同的方法产生，但是它并不会被抛弃。当新的合同产生时，要核对存在的持久合同夹。如果它们匹配了过滤器，它们就被增加到合同夹中。任何基于合同夹的处理必须在那时更新，最好是增量式的。持久合同夹提供更快的查询性能但是减缓了合同的产生性能并且耗尽了存储空间。一个基本的建模原则是用户不应该知道合同夹是短暂的还是持久的。合同夹应该可以从一个状态自动切换到另一个状态，而不需要用户的干预。这就要求创建新的合同夹过滤器时要核对任何已存在的持久合同夹过滤器。如果有一个已存在的过滤器匹配，那么就应该引用这个已存在的合同夹过滤器而不是产生一个新的合同夹过滤器。

合同夹在许多领域都很有用。合同夹的基本特征与对象的基本特征相同，即把一组某种类型的对象而进行的选择机制封装起来。合同夹可以作为处理某种更进一步的汇总的基础。这个处理可以是一个客户对象（比如在本章），也可以被构造成合同夹本身。

例：一个汽车生产商能开发生产汽车的合同夹来汇总产品和错误数据。过滤器可以按照汽车的生产厂家、型号、生产手段或者某些日期范围来选择汽车。

例：公共卫生是医疗保健的一个重要分支，它研究患者群体的健康问题。我们可以按照一种特征域来选择人群：年龄、居住地、适用的观察概念等。这些人群可以被过滤器定义，并且可以对他们进行观察，例如一天吸烟超过20支的人的平均峰值流量率。（人群是这样一个人的合同夹，其过滤器是一天吸烟超过20支的人。<sup>⑨</sup>）

<sup>⑨</sup> 香烟上的过滤嘴是另外一回事。

184

### 9.3 报价

任何在金融市场上交易的东西都有一个价格。然而，那个价格并不是通常单一的数字。被报价的有两个数字：购买的价格（出价）和销售的价格（索价）。我们可以通过给这对数字的使用进行建模来描述价格，如图9-9所示。

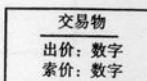


图9-9 通过两个数字的属性来描述价格

一个交易物可以使用数字或者货币对象来估价。通常，股票使用货币来估价，而汇率使用数字来估价。报价行为在任一种情况下都是一样的。（我们可以把报价看作一个参数化的类型。）

尽管两个数字是通用的，但是它们并不总是被使用。有时，报价是一个单独的价格，它描述价格的中间值。一个单独的价格用差价——出价和索价之间的差值——来报价。有时，我们可能只看到出价，或者只看到索价。这影响报价显示的方法。在外汇市场，汇率（如USD/GBP）可能报价为0.6712/5，这说明出价是0.6712而索价是0.6715。如果只有出价，显示的报价就是0.6712/；只有索价，显示的报价就是/0.6715。

任何可能有双向价格的对象——如汇率、日用品等——都需要一个如图9-10所示的行为的数字。可以把这些行为组织成一个单独的报价对象，如图9-11所示，这个对象提供双向价格所需的所有行为。任何用报价来表示价格的事物都需要一个报价属性。



图9-10 支持双向价格所需的行为

报价已经成为一个基本类型，并且最好能被描述成那些区别属性和对象类型的建模方法中的一个属性。注意：属性不能描述数据结构这一点是很重要的，仅仅是表示存在适当的操作。

185

例：汇率USD/GBP是0.6712/6。交易物是USD/GBP。这个交易物有一个报价，它的出价0.6712，索价是0.6716，中间价是0.6714，而差价是0.0004。

例：一个CD交易出售已使用过的全价CD12美元，而购买它们花了8美元。出价是12美元，索价是8美元，中间价是10美元，而差价是4美元。交易物是全价的古典CD（正好是肖邦的夜曲）。

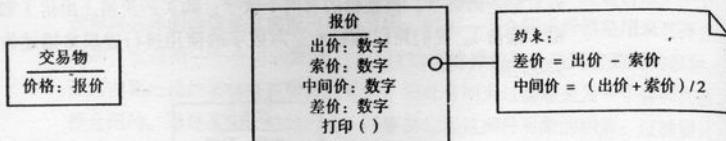


图9-11 使用独立的报价对象

这是一个好方法，因为它把特殊的职责放到一起成为一个简单的可重用概念。

**建模原则：**当多重属性和一个可能在几个类型中使用的功能进行交互时，属性应该被结合成一个新的基本类型。

双向价格是普遍的，可是有时单向价格也被使用。单向价格的建模有点难以掌握。一个替代办法是允许价格是一个报价或是一个数字。这在强类型语言（如C++）中几乎是不可能的。甚至在Smalltalk中，股票的客户在用它做任何事情之前，就被强制检查价格返回什么类型的对象。

一个替代办法是让报价成为数字的一个子类型。这个方法可行是因为报价可以响应算术操作，可是它还是要求客户在处理股票价格时（打印除外）要知道两者的差别。在C++中，数字不是一个内置类型，但实型和整型是内置类型，除非提供一个数字类，否则不应该使用这种方法。

另一个替代办法是让数字成为报价的一个子类型。在概念上，这种方法有一个明确的要求。数字只是简单的报价，并且不难考虑，每一个数字的实例都是一个报价的实例——出价和索价相同。（类似地，我们可以说数字是复数的一个子类型。）尽管这种考虑有概念上的优点，但是它在一个接口模型中却是失败的。为了使数字可以成为报价的子类型，它必须完全继承报价的接口。报价只是在很少的领域使用，而数字却几乎在每一个领域中使用。从报价子类型化意味着报价可以在所有的领域使用，包括许多报价行为没有用的领域。报价必须被设计为对数字可见，而不是反过来。

**建模原则：**如果超类的适用领域狭窄而子类的适用领域广泛，就不应该使用泛化。

现在，我们应该考虑报价的双向形式和单向形式之间存在着怎样的共性。存在两种替代方案：或者将单向报价作为一种报价，出价等于索价；或者向单向报价请求出价或索价是错误的。第一种方案有一个抽象

报价的存在，如图9-12所示，而第二种方案避免了任何这样的泛化。第一种替代方案，客户可以把单向和双向报价作为相同的行为而不涉及它们的差异。然而，这可能导致不准确，因为客户不能肯定是在处理双向报价的出价。需要一个类型测试操作（`isTwoWay`或者`hasType`（‘`TwoWayQuote`’）），以便客户可以进行类型测试。如果没有抽象的报价，则这些不准确就不会发生，可是客户每一次必须使用类型测试来了解调用的操作其使用是否安全。

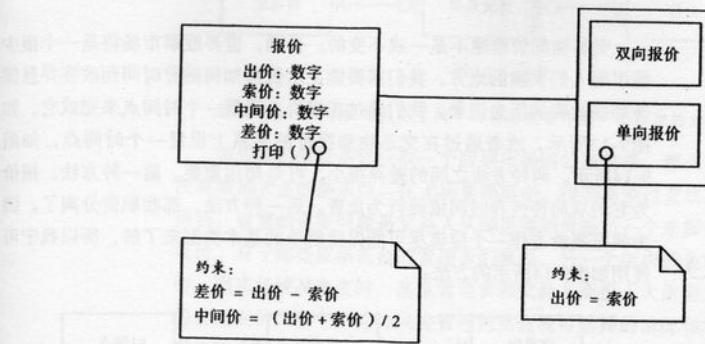


图9-12 带子类型的抽象报价

单向价格被看作是双向价格的一种特殊情况。

决定的关键在于是否可以接受经常忽略双向报价和单向报价之间的差异。如果这几乎不能被接受，那么最好不要有抽象的报价类型。然而，如果它经常可以被接受（这要看具体情况），那么我将强烈鼓励使用抽象的报价类型。记住，对于客户来讲，使用抽象的报价决不比不使用需要更多的努力。当不需要考虑两者的差异时，使用抽象的报价还能节省工作量。

**建模原则：**如果两个相似类型的差异经常被忽略，那么一个抽象超类型就可以被使用。如果它们之间的差异通常是很重要的，那么一个抽象的超类型就不应该被使用。

**建模原则：**如果一个抽象类型从不需要客户花费更多的努力来使用，那么它就应该被提供。

这个抽象的报价类型包含子类型的所有行为，因为在子类型上没有附加的操作或者关联。通常，我们不使用子类来实现一个抽象报价的子类型化，尤其是因为这样一个基本的对象经常使用C++中的容器。更好的实现方式是报价类中的一个内部标志，尤其是因为我们经常需要折合一个双向报价（也

就是，把它转化成一个单向报价），或者反过来，这时需要动态的分类。

一个隐含的报价可以是购买或者销售，在每种情况下都不需要双向价格。只有当购买和销售两者都需要时，才需要双向报价。

有时，我们需要描述作为报价的合同的价格。通常，当对方团体询问合同的价格时，它们并没有指定方向；在那种情况下，交易人用报价来答复。通过保存报价，当合同被报价时，交易人记起差价是多少。实际费用的数量就可以轻易地从合同的方向和报价中得到。

## 9.4 场景

交易物的价格绝不是一成不变的；否则，世界股票市场将是一个很少能引起人们兴趣的地方。我们需要能显示价格如何随着时间而改变并且保持那些改变的历史记录。我们通过在报价上设置一个时间点来完成它，如图9-13所示，或者通过在交易物和报价的关系上设置一个时间点，如图9-14所示。两种方法之间的差异很小，可是却很重要。前一种方法，报价为它的双向特性和时间依赖行为负责。后一种方法，那些职责分离了。因为我把报价当作一个应该尽可能保持简单的基本类型来了解，所以我宁可使用如图9-14所示的方法。

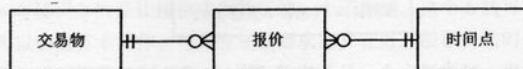


图9-13 给报价增加一个时间点

时间点指明在什么时间报价对于某种交易物来说是相符的。

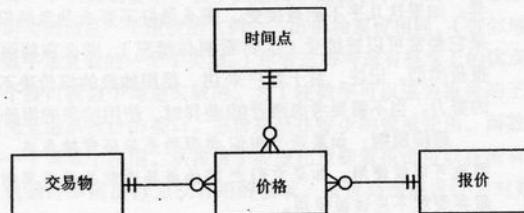


图9-14 在一个特别时间点上的股票价格

这从在某个时间点适于一种交易物价值的概念（价格）中分离了双向行为（报价）。

在这些模型中，查找市场的收盘价时，需要收集那个市场的所有股票，并寻找每支股票的最后报价。另一种替代方法是把这些报价的聚合本身看

作一个对象——即一个场景，如图9-15所示。这个场景描述在特定的时间点上的市场状态，而场景中的元素描述在那个时间点的价格。[188]

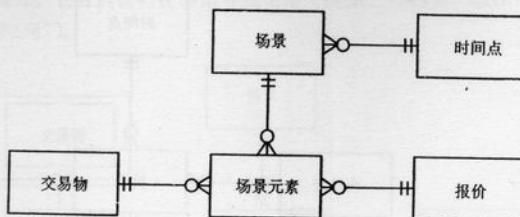


图9-15 场景

这允许在一个单独时间点上的一组价格被看作是一个单独的对象。

如果我们仅仅是想获取一个股票市场的公布价格，那么场景看起来并没有增加太多的新东西。在无场景模型中，很容易通过查找某个时间点来得到。这里的重要问题是交易人从哪里得到价格。一个来源是交易的公共报价。对于那些股票基金的管理者们来说，另一个考虑就是可能的将来价格。当市场情况改变时，基金管理者和交易人要投入大量努力到管理他们的合同夹的风险中。这种风险管理包括查找可选择的情况和考虑它们在资产价格方面的影响。

**例：**一个基金管理者正在管理股票的一个合同夹。她要考虑可能的油价下跌，这将促进许多股票价格的上升但是会降低其它一些股票（例如石油公司股票）的价格。这个管理者想看看几种不同程度的下跌并且考虑它们如何影响一个合同夹。这些下跌股票中的每一个股票都导致一个不同的场景。[189]

**例：**一位生产管理人员正在评定汽车可能的生产成本。原材料成本和人工成本是和价格有关的交易物。对于这些交易物不同的可能价格，可以构建多个场景。

以上的几个例子是假想的情况，显示了采用场景的好处。在假想情况下，场景对象提供把所有因素综合在一起的基础，这样不同的情况能很容易相互比较。

我们也要考虑市场，市场中的价格没有一个单一的发布者，例如外汇市场。在这种情况下，我们需要增加发布价格的团体到模型中。图9-16和图9-17显示在初期的模型上增加了发布团体。有场景和无场景的方法都是有效的。显然，风险管理是否需要场景的决定因素。[190]

**例：**一个进/出口商人考虑在许多欧洲国家的货物的价格。我们可以通过形成一个场景来描述价格，在场景中，交易物是他有兴趣交易的货物。

通过着眼于这些市场之间的差异，使用双向价格，他可以寻找价格差别比运输这些货物的成本大的地方（这个过程我们通常称之为套利）。

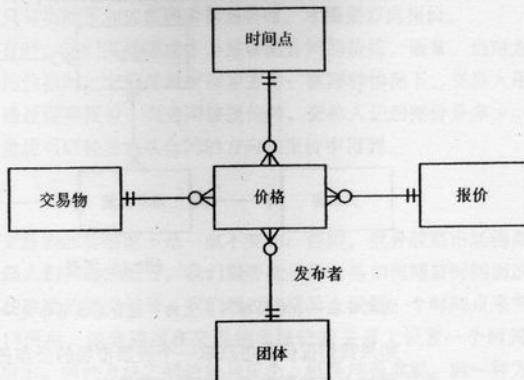


图9-16 带发布团体的图9-14中的模型

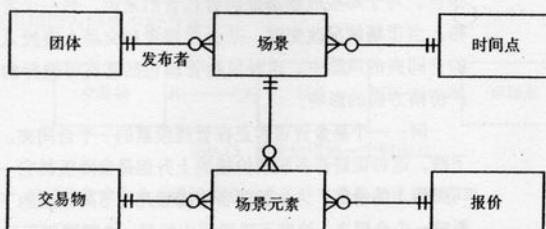


图9-17 带发布团体的图9-15中的模型

使用发布者是使用场景的另一个原因。

**建模原则：**场景应该在价格或者汇率的组合需要被看作一个整体的时候使用。

#### 定义如何构造场景

价格到底从哪里来？在有些情况下，这可能是一个简单的问题，例如，当价格由一个交易所发布时。在其它的情况下，特别是当假想的场景存在时，就可能要使用更复杂的方案。

在宽泛的条件下，我们可以看到价格的三个起源：由某些机关发布的市场中的普遍报价，从其它的价格或者市场特征计算，或者是某个单

独的交易者或者分析家们的意见。第一种情况，如图9-18所示，是最直接的，通过特定的操作得到相关信息。通常，这些信息来自一个信息源，例如路透社，它告诉我们在哪里寻找信息（例如，“第3页，以IBM开始的行的第2列”）。

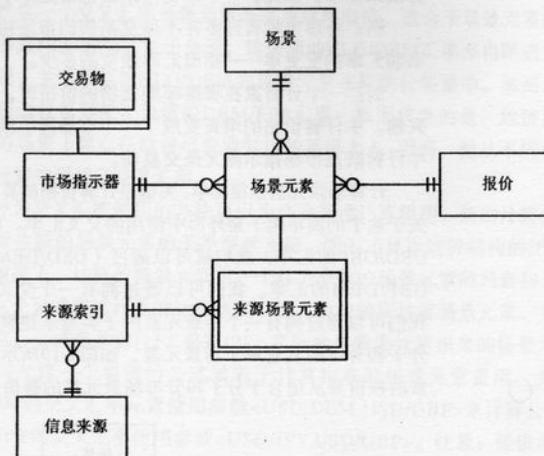


图9-18 场景元素的来源

这个模型描述一个特定的元素是从哪里来的。[91]

实际上，使用一个发布的价格，需要派生一个来源场景元素报价。我们从来源索引派生得到这个报价，而不是断言这个报价是针对一个来源场景元素。因此，可以考虑将这个到报价的连接做成一个派生连接。如果连接从没有被断言，这就可以安全地实现（比如记录交易人的一个推测）。对报价进行断言可能会导致问题，因为有时报价是断言的，有时报价是派生的。排除这个问题的一种方法是为混合的和随意的派生关系使用一个表示法（符号）（参见Odell[2]）。这看起来把这个派生的问题带得太远了。我倾向于做上标记，按照大多数普遍情况并且在支持的文档中精确地描述发生了什么。

例：一个分析家在观察邮购货物的价格时，可以把每个公司看作一个信息来源。来源索引可以是目录里的一个页码。这对每一个零售商来说，都可以设立一个场景，或者把所有零售商联合起来，建立全局的场景。这样，不仅能支持寻找交易物的价格，还可以支持某个交易物的最低价格和平均价格。

图9-18提出市场指示器作为一种交易物的超类型。这反映出场景可以包含交易物以外的事物。对于派生的事物来说，定价方法的一个重要部分就是交易物的变更率——一个数字指明某种交易物价值改变的多少。变更率不是可以交易的交易物，可是在一个场景中，它和一个交易物以相同的方法来记录。因此，一个市场指示器包括变更率，也包括所有的交易物。

例：外币市场有许多并不是交易物的市场指示器，包括各种货币的利率和汇率的变更率——指出汇率改变的多少。

例：一个分析家在观察邮购货物的价格时，对牛仔裤价格的上涨很感兴趣。牛仔裤价格的增长变成了一个市场指示器，可是并不是交易物。而牛仔裤既是市场指示器又是交易物。

计算场景元素也很简单。关键是计算价格的算法本身可以成为一个对象。关于这个的简单例子是外币中使用的交叉汇率。如果我们知道USD/DEM和USD/GBP的汇率，我们就可以通过 $(\text{USD}/\text{DEM}) / (\text{USD}/\text{GBP})$ 来计算GBP/DEM的汇率。我们可以通过拥有一个交叉汇率场景元素来描述它，我们可以通过拥有一个场景元素的子类型来建模，这个场景元素引用作为分子和分母的其它两个场景元素，如图9-19所示。适合于交叉汇率场景元素的报价要从适合于分子和分母场景元素的报价派生而来。

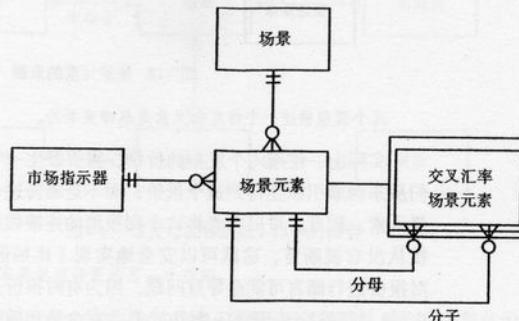


图9-19 通过交叉汇率来计算场景元素

这可以用来从两个已知元素的比率来测定第三个元素。

注意：分子和分母被表示为场景元素而不是市场指示器。如果我们仅仅通过上述的描述来表达交叉汇率，那么引用市场指示器看起来是最切合实际的（USD/GBP是一个市场指示器）。然而，提供场景的根本目的在于：对于相同的市场指示器，应该允许我们在不同的假设下公布几种不同的价格。引用市场场景元素允许我们把注意力集中在这些我们将要使用的价格

上。这些可能是两个USD/DEM数字：一个来自路透社，一个来自LIBOR。通过引用这些场景元素，我们能指明我们需要的是哪一个。

**例：**有一个交易人是法国法郎的专家。她通过使用德国马克（DEM）作为交叉汇率，测定了荷兰盾（NLG）和法国法郎（FFR）之间的汇率。她通过产生一个交叉汇率场景元素来完成这件事情。适合于场景元素的市场指示器是NLG/FFR。关于分子，她使用的NLG/DEM汇率来自路透社；也就是说，关于交易物NLG/DEM的场景元素在路透社场景中。然而，她没有从路透社得到关于分母的DEM/FFR汇率；取而代之的是，她使用了她自己的场景（建立在她自己专业知识的基础上）。因而，她从不同场景的场景元素中形成了交叉汇率。

这种为交叉汇率使用的方法可以为许多普通计算使用，新的计算的种类通过派生新的场景元素的子类型被支持。图9-20显示这种结构的泛化。在这种情况下，计算的场景元素有一个作为参数的场景元素的列表和一个公式。这个公式描述计算的运算法则，这种方法使用计算场景元素。关于交叉汇率，公式是参数[1] / 参数[2]。实际的参数由计算出来的场景元素提供。这允许一个单独的公式被若干计算出来的场景元素重用。关于GBP/DEM的交叉汇率元素使用参数<USD/DEM,USD/GBP>来计算公式，而GBP/JPY的交叉汇率使用参数<USD/JPY,USD/GBP>。注意：提供的参数应该作为一个列表，而不是通常的多值映射的集合。参数在列表中的位置对于正确书写公式是很重要的。[193]

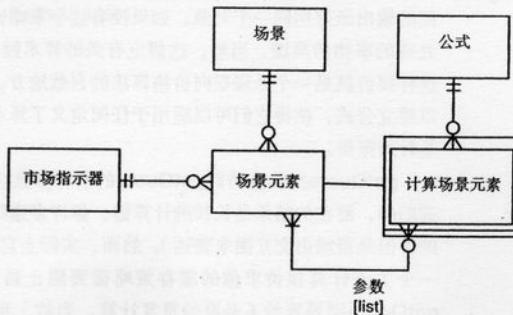


图9-20 计算场景元素的更一般方法

这个公式可以是基于参数的电子数据表样式的公式。它支持广泛的场景元素的算术组合。

**例：**牛仔裤的价格改变通过采纳了今年的场景和去年的场景之间的牛

仔裤价格的差价场景元素来计算。

我们可以用多种方法来实现这个公式。一种方法是用实现语言对公式进行硬编码。因为普通公式（例如交叉汇率）可以被很广泛地重用，所以在这种情况下硬编码不是一个缺点。如果公式的数量很小并且不能太频繁改变，那么这就是一个最好的方法。即使每个月增加一个公式，这种方法也能够相当简单地进行控制，甚至对一个复杂的系统也是如此。如果我们想给使用者增加公式的能力，那么我们可以使用一个更成熟的方法。我们可以构造一个解释器[1]，它认可一个简单范围的公式表达式。这种技术对于使用电子数据表的用户是很熟悉的。我们可以提供一个交互式的公式生成器，可是任何生成公式的人大概都可以打出一个类似电子数据表的公式。这个解释器[1]因而没有必要识别所有的公式。最完美的方案可能是一些公式用解释器而另一些公式用硬编码。使用场景元素的软件并不关心如何生成一个公式，可是它却需要关注提供参数给产生计算报价的公式。这遵循了最好的面向对象的原则——接口和实现分离。（更深的讨论请参见6.6节。）

**建模原则：**为了把一个过程看作类型的一个特征，这个过程应该采用一个抽象的接口，以便实现能轻松地通过子类化而改变。一个通过硬编码得到的实现是一个子类，不同的参数驱动方法是其它的子类。

如图9-21所示的交互图揭示关于这个行为可能如何工作的一些有用要点。第一个要点是如何给出公式作为输入的一个报价列表，而不是一个场景元素列表。虽然这是适度的随意，但通常有一个策略：输入的事件和返回的输出最好用同一个对象。如果没有这个策略，程序员会很快搞混它们处理的事物的类型。当然，这假定有关的算术操作都已经在报价中定义，这样报价就是一个处理双向价格算法的自然地方。在这种情况下，我们可以建立公式，使得它们可以适用于任何定义了算术操作的对象，而不仅仅是针对报价。

**getQuote**操作在访问**getQuote**的所有参数这一点上，其行为自然是递归的，潜在的结果是长长的计算链。像许多递归结构一样，这是很优雅的（但是很难由交互图来表达）。然而，实际上它会导致大量冗余的计算。一个关于计算报价求值的缓存策略需要阻止通过多次对相同对象调用**getQuote**而导致的不必要的重复计算。当然，和所有的缓存一样，我们必须确保当一个源值改变时，缓存要适时更新。我们可以在相反的方向使用参数映射来重置所有依赖于它的场景元素。

**建模原则：**当信息可以从一个信息源检索或者可以从其它可用的数据计算时，应当提供一个抽象的接口，而源和计算各为一个子类。

194

195

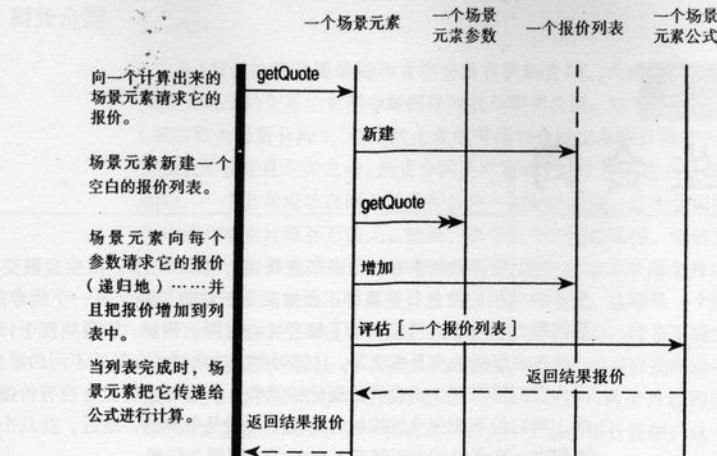


图9-21 计算场景元素的交互图

## 参考文献

- Gamma, E., R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.
- Martin, J., and J. Odell. *Object-Oriented Methods: A Foundation*, Englewood Cliffs, NJ: Prentice-Hall, 1995.

[196]