

## 第10章

# 派生合同

为了完全理解本章，你必须先阅读9.1节和9.2节。派生金融交易[3]在交易中承担的角色日益显著。派生交易的价值依赖于另一个债券的价值。更简单的派生形式已经出现了相当长的时间；例如，股票期权于1973年第一次在有组织的交易所交易。从那时起，越来越多奇异的不同的派生出现。因为它们可以减少来自价格变化的风险，所以对投资者是很有价值的。然而，当它们不能完全被控制时，派生可能是危险的：最近，在几个著名的案例中，许多机构由于管理不良的派生而损失惨重。

对派生进行建模体现了建模的许多有用特征，因为派生体现了一个自然的泛化层次结构——这比一般的动植物的例子更有意思。因此，本章的目的是探索在这种以派生作为代表的泛化层次结构中存在的一些问题。

我们从介绍简单的派生开始：期货合同（参见10.1节）和期权（参见10.2节）。期货合同介绍期限的概念，这导致一个为什么日期计算比合计天数更复杂的讨论。期权提出了两个难于对付的建模领域：掌握交易人对于股票看涨和看跌的定义，以及期权和潜在合同之间的关系。

一个更复杂的派生类型——组合期权——可以被看作是一个更简单的期权的聚合。以组成模式从期权子类型化并不是总有效；这导致产品模式（参见10.3节）的产生。这种模式基于卖主和交易人对交易的观点的不同并且也适用于定期交易。这也可作为一个例子，说明泛化虽然总是我们首先想起使用的方法，但不一定是最好的方法。

随着子类型化，我们必须确保子类型的行为和超类型的行为一致。使用关卡期权作为一个例子，我们将研究子类型化和状态图如何同子类型状态机（参见10.4节）交互。

如果我们有一个期权的合同夹，就能够依据合同夹中期权的类型，选择一个相关细节的浏览器。这导致并行的应用和领域层次结构（参见10.5节）的出现。这两个层次很难结合。这个模式引起的问题有几种解决办法，但没有一种解决办法是全能的。

关键概念：期货合同、期限、期权、产品

## 10.1 期货合同

9.1节讨论的是简单的和直接交易有关的合同。大多数市场包括一个更复杂范围的交易。它们中最简单的就是期货合同。对于一个标准的合同(通常称为现货合同),交货发生在尽可能和合同交易的日期接近的时候。交货通常发生在几天之内。期货合同是对在将来某个时间达成交易的协议。例如,一个公司应该在两个月内接收到一油轮的石油。这个公司将不得不为这些石油支付数百万美元。然而,如果这个公司在德国,它的正常的资金筹措应该以马克来完成。如果美元/马克的汇率在未来两个月内显著改变,则这个公司就会不得不支付比预期更多的马克,这将是一个重大的问题。当然,这个公司也可能从汇率的有利改变中获利;可是不确定性对这个公司来说是不利的。为了减少不确定性,这个公司可以选择以一个期货合同汇率的交易来购买几百万美元,方法是现在为两个月后的美元交货支付约定数量的马克。这个价格由执行这个交易的银行提供,这个交易基于下两个月市场上美元/马克汇率变动的预测。这个交易的期限是两个月(与现货期限相反)。

通过持有分离的交易和合同的交付日期,就很容易描述一个期货合同,如图10-1所示。一个现货交易将会有适当紧密的交易日期和交付日期,而一个期货合同交易将会使这些日期之间相隔两个月。虽然我们可以增加一个子类型来清楚地显示,但是我们不需要一个子类型来显示它。

198

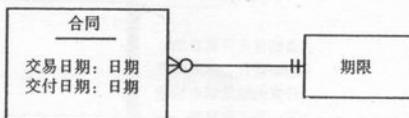


图10-1 一个能支持期货合同的合同

期限是基于交易日期和交付日期之间的不同。

**例:** Aroma Coffee Makers公司在1997年1月1日同意从巴西咖啡出口公司购买5 000吨的巴西咖啡。交货定在1997年10月20日,并且价格定为当日的价格。

**例:** 我为旅游购买了三个月内定期航线的机票,按当前的旅行报价支付票价。

当讨论期货合同时,一个重要的考虑因素是合同的期限。在我们的这个例子中,期限是交易日期和交付日期之间的时期。期货的报价一般都是要考虑到期货的具体期限,期限因而是期货合同中一个重要的组成部分。

然而，期限并不简单的是交易日期和交付日期之间的持续时间。如果我们的两个月的合同在5月4日完成交易，则交付日期将不会是7月4日，仅仅因为7月4日是美国的独立日。在这些日期计算时，节假日有很大的影响。假定7月4日不是周末，一个在5月4日交易的两个月合同将会在7月5日交付。注意：如果由于某种原因德国在7月5日有一个节日，交付日期还会向后再推移一天。即使这个合同的交付日期和一个带有两个月零一天的期限的合同一样，它还是有两个月的期限。注意：这个行为对于现货合同也是需要的：即使现货是两天，一个在星期四完成的交易也将在下星期一交付（前提是星期一不是假期）。因此，图10-1包括：交易日期、交付日期和期限。

在这种结构中，交付日期的计算并不是单单通过交易日期和期限就可以完成。如果没有考虑假日，我们可以通过一个简单的日期和期限之间的计算来确定交付日期。然而，市场假日必须考虑在内。这就意味着市场有一个日期计算的常规，它允许日期随着假日而调整，如图10-2所示。这种对于假日的考虑在很多领域都是重要的特征，这些领域中工作日的概念变得很重要。因为国家之间的假日是不同的，或者可能具有更大的粒度，所以通常不可能确定全球性的工作日。每个地方可能有本地习惯的假日，这会影响日期的计算。

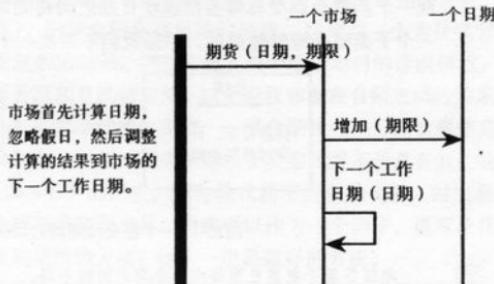


图10-2 市场的计算日期常规

当需要计算工作日时，需要将日期计算委托给另一个对象。

**例：**一个公司需要在从1997年6月30日开始的五个个工作日内给一个雇员支付薪水。如果这是一家美国公司，就是到7月8日为止（跳过周末和7月4日）；对于一家英国公司，就是7月7日。

**建模原则：**日期计算经常受到需要跳过的假日的影响。不同国家之间的假日通常是不同的，甚至不同商业机构也有自己的假日。

## 10.2 期权

对于我们在德国的石油公司，期货合同是一个很有价值的工具，可以用来降低汇率改变的风险，而汇率改变也许会导致他们必须为石油支付更多的钱。但是汇率可能按他们的希望改变，这样这家公司还是冒了损失的风险。财务主管本质上不得不在汇率浮动上打一个赌。如果他们认为马克将会升值，他们会在现货市场购买；如果他们认为马克会贬值，他们会购买期货合同。期权降低了风险。如果持有者愿意，期权给予购买者以预先安排的汇率购买美元的权利。因而，如果马克贬值，石油公司可以行使它的期权并且以预先安排的汇率购买美元；如果美元升值，他们可以忽略他们的期权（让它终止）并且在现货市场购买。银行向石油公司索要保险费而卖给他们期权，所以银行现在承担了风险。由于银行操作了许多这样的交易，因此他们可以依靠不同的交易相互抵消这种风险。图10-3和图10-4描述一个期权的行为。

[200]

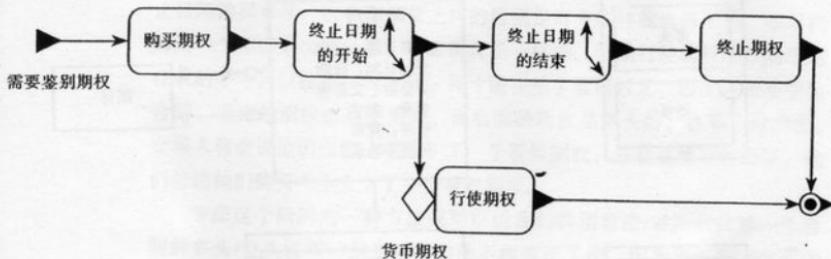


图10-3 期权使用过程的事件图

期权只有在终止日期开始之后才能行使，并且将只能在“货币”期权时才能行使，也就是说，如果行使期权比以当前的价格进行现货交易更好的话才能行使。

期权的许多特征和普通合同很相似。像一个普通合同，期权有对方团体和交易日期。期权的其它特征包括终止日期、保险费的数量以及交付保险费的日期。

因而我们可以认为期权是合同的子类型，如图10-5所示。期权结构的关键特征是多态操作“价值（场景）”。一个现货合同的估价很容易理解，因为把指定场景中的现汇交换率应用到这个合同的金额得到的结果就是现货合同的价值。说得婉转些，就是期权的估价更复杂。最普通的技术是Black-Scholes分析[3]。这个问题的解释超过了本书讨论的范围，但对于与这个操作有关的调用者来说，它也只是一个简单的操作。数学上的复杂性可以被安全地隐藏在这个操作中。

[201]

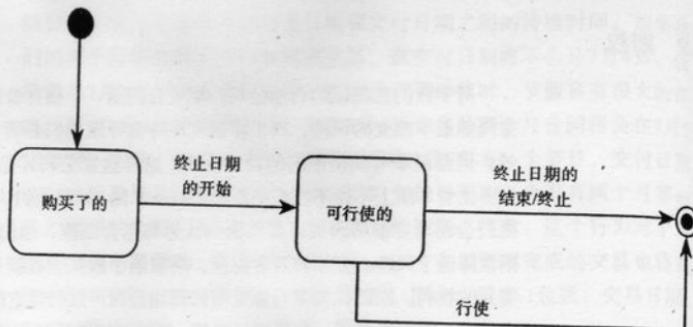


图10-4 说明期权如何运作的Harel状态图

期权只能在它的终止日期之前行使（“欧洲式的”期权）。

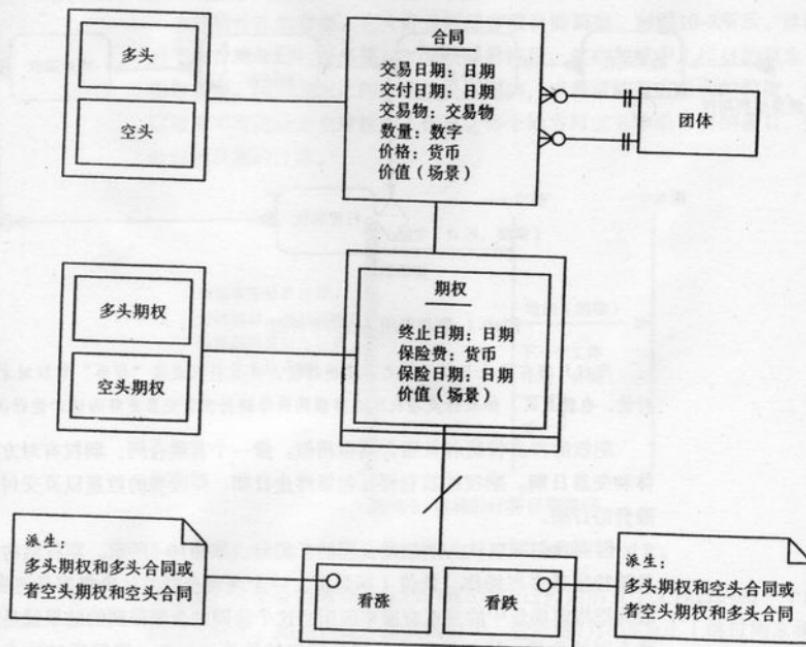


图10-5 期权的结构

看涨和看跌是从“多头”和“空头”派生的术语。

## 9.2.1 多头、空头、看涨和看跌：体现一种谋略的词汇

多头和空头的问题不需要再进行讨论。在9.1节中，我们解释了一个合同可以是多头合同（购买）也可以是空头合同（出售）。然而，对于期权，我们发现有四种可能的选择。我们可以出售一个期权来售出货币，出售一个期权来买进货币，购买一个期权来售出货币，或者购买一个期权来买进货币。多头/空头选择仍然存在于合同中，但是它通过期权中一个更进一步的多头/空头选择来补充。交易人的词汇表包括术语“看涨”和“看跌”。看涨期权是购买一个期权（也就是，一个多头合同），而看跌期权是出售一个期权（也就是，一个空头合同）。自然地，我们可以购买或者出售一个看涨期权，也可以购买或者出售一个看跌期权。描述这种语言有点难以掌握，也容易混淆。

[202]

如果我们出售一个期权来购买日元，那么对方团体可以从我这里在终止日期购买日元。这个和期货之间的区别是对方团体能选择不买。如果我购买一个期权来出售日元，那么情况是一样的，但是行使期权的控制却是在我的手中。以任何一种方式，我（潜在地）卖出日元，因此合同是空头合同。前面的期权也是空头的，而后面的期权是多头的。在第一种情况，交易人将会说他们出售（空头）了一个看涨期权，并且在第二种情况，他们会说他们购买（多头）了一个看跌期权。

考虑这个问题的一种方法是可以说我们将用看涨/看跌来代替一个合同的多头/空头说明。但是这个方法不能真正工作，因为我们在非期权的合同中不使用术语“看涨”和“看跌”。

另一个可能性是对期权使用术语“多头”和“空头”来指出期权而不是合同的状态。因而上面第一个例子将是一个空头的看涨期权，而第二个例子将是一个多头的看跌期权。这对一个交易人可能很有意义，但是将会搞乱所有的软件。当评估风险时，合同的金额额度是很重要的，并且上面的例子都是空头的。因此我们需要能询问合同的方向（这决定了额度）、期权的方向和看涨/看跌。所以这两个例子是（空头合同，空头期权，看涨）和（空头合同，多头期权，看跌）。很清楚，它们中的任一个可以从另外两个派生得到。图表暗示看涨/看跌是派生的。这只是暗示一种派生，而不是向实现者表明实际实现中的存储或者计算的任何方向。

描述像这样的语言总是带有一点挑战性，尤其当看起来毫无必要地不合逻辑的时候。重要的事情是以一个合理的风格描述基本术语。这些基本术语可能是领域专家的术语的一部分或者是在建模过程中的发明（但是如

果它们被发明出来，领域专家就必须适应它们）。术语中其余内容可以从这些基本术语派生。

**建模原则：**派生标志应该被用来定义从模型中其它的结构中派生的术语。

**建模原则：**标记一个特征为派生是接口上的约束。它并不影响基础的数据结构。

**例：**1997年6月1日，我得到一个期权，内容是在1999年1月1日以5美元每股的价格来购买200股Aroma Coffee Makers的股票。这是一个带有交易日期是1997年6月1日的期权，交易物是Aroma Coffee Makers的股票，额度是200，交付和终止日期是1999年1月1日，保险费是0美元，价格是5美元。我想得到股票，所以合同是多头合同（对于我来说），期权也是多头期权（因为我持有期权）；因而它是一个看涨期权。

**例：**当我预订了机票时，我开始被给予关于机票的买入期权。期权的终止日期是预订的必须拿票的日期。

另一个有关的问题是交付日期和终止日期之间的相互关系。对于一个期权，如果知道终止日期，是可以计算交付日期的（交付日期 = 终止日期 + 现货期限）。然而（由于假日的关系），反过来却不成立。这意味着，对于期权，交付日期是一个可计算映射。这里重要的一点是接口不能改变：这里还有交付日期的一个访问者；然而，信息存储了。有两个可选择的方法来描述这种情况：我们可以注意（通常在术语表中）对于期权，交付日期属性依照公式被覆盖并从终止日期进行计算。另一个选择是描述一个公式作为期权类型上的约束。两个方法都是完全合理的并且如何选择只是爱好问题。这完全取决于实现者使用什么样的代码和数据结构。

## 10.2.2 子类型化或者非子类型化

图10-5所示的结构不是处理期权的惟一方法；另一个选择如图10-6显示。两种结构之间的区别是如何把选择权加到合同中。在图10-5中，我们通过子类型化来加入它。在这个方案中，期权是一种带有附加属性和一些不同行为的合同。在图10-6中，我们可以说期权有一个基础合同，交易人通常把它称为期权基础。这里至少有一些包含的概念，尤其如果合同实际上是一个期权的基础，则我们不可能要求合同给出自己的价值。类似地，交付日期将依赖期权的终止日期。

在两种结构之间做出选择不容易。每一种都有优雅的品质。图10-6的模型用一个明确的基础概念分离了期权和合同的概念。这种方案的一个

缺点是一个单独的合同由两个对象来描述。更容易改变图10-6来处理复合期权（期权的基础成分又是一个期权）。由于在它们之间很少有选择的余地，因此我们可能很容易最终陷入泥沼。原型法有时可以阐明这种情形，但也并不总是如此。当像这样的多种方法需要选择时，先使用更简单的方法，并且如果需要以后再改变为更复杂的方法是一个好主意。然而，在这种情况下，哪一种方法更简单仍然需要讨论。当这种情况来临时，我相信领域专家的直觉，所以询问他们感觉那一个最好。我们将使用图10-5作为基础，进行更进一步的讨论。

**建模原则：**当面对可选择的方法时，首先要选择最简单的，并且当需要时改为更复杂的方法。

**建模原则：**当可选择模型之间的选择余地很小时，听从领域专家的直觉。

[204]

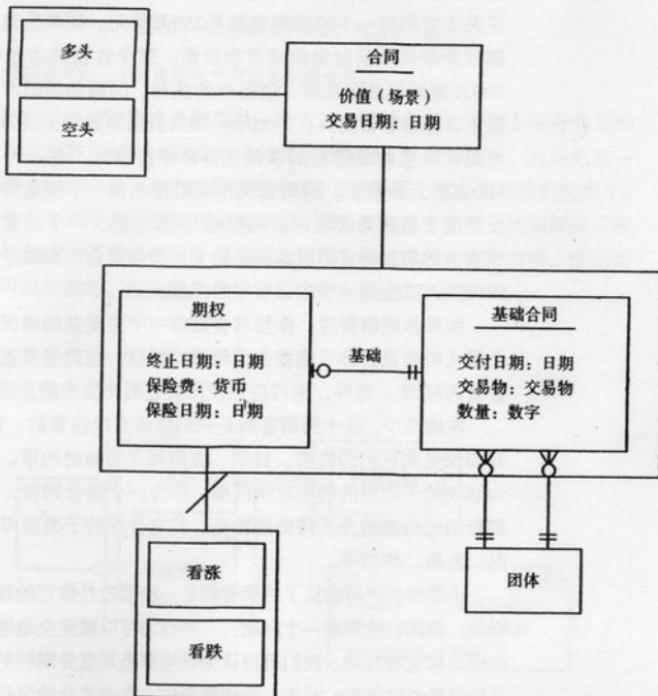


图10-6 对期权和合同的分离对象方法

本图和图10-5都是合理的可选择的方法，但是本章是基于图10-5。

### 10.3 产品

派生交易很长时间被认为有一定风险，主要是因为评估风险所需的复杂数学。作为求值过程基础的Black-Scholes方程[3]是一个二阶偏微分方程。即使我有工程背景，这些东西<sup>①</sup>仍然让我焦虑不安。

最引人注目的派生交易陷阱的例子是英国古老的巴林银行的倒闭。依照现在的报导，倒闭的主要原因是一种特殊的派生交易叫做约期套购——一个组合期权的例子。组合期权可以看作是其它期权的合成物。看起来本节以约期套购作为例子讨论是很合适的。

约期套购的概念实际上非常简单。你持有大约7000万美元的财产，依靠价格，你关心未来三个月内，它的价值的任何大的改变。或者增值或者贬值都会导致一个问题。为了避免这个问题，你可以购买一个看涨期权和看跌期权，都带有7000万美元的价格并且终止日期是三个月后。让我们假定关于它们每一个的保险费都是250万美元。如果价格增值，你行使看涨期权并获得按新价格的持有物价值。这个价值减去原有的7000万美元和500万美元的总保险费，就是你的获利。因而如果财产升值到7500万美元以上，你会非常高兴。类似地，如果价值跌落到6500万美元以下，你也很高兴。发生的最糟糕的事情是价格保持稳定，在这种情况下，你损失了500万美元保险费。约期套购的吸引性来自一个固定的风险，它补偿了相反情况下范围非常宽的价格变动。很自然，一个非常不稳定的交易物会导致为约期套购支付更高的保险费，但如果你试图减少在一个易变环境中的风险，这会是一个非常有用的产品。

如果你是销售者，你当然会面临一个更复杂的情况：如果价格移动一个很大的数量，你可能损失无限额度的钱。这就是真正导致一家银行损失惨重的原因。此外，银行应该使用其它的交易来防止这种风险。

在建模中，这个约期套购是我们应该直接注意的，它由两个期权组成，而期权受到它们的价格、日期、方向和交易物的约束。图10-7显示一个作为期权的子类型的约期套购模型。作为一个组合期权，它会有成分，对约期套购的约束定义了精确的特征。其它合并的子类型可用于其它常见的情况：差价、抑制等。

子类型的使用确认了约期套购是一种期权并像它的超类型一样有相同的行为。然而，这带来一个问题：一些行为可以被安全地继承，例如自我估价的能力和交易日期。我们可以认为保险费是其成分期权的保险费的总和。可是价格是怎样的呢？对于一个约期套购，它的成分的定价都是一样的，所以

<sup>①</sup> 意指Black-Scholes方程。——译者注

我们可以认为它是约期套购的价格。然而，另一种普通的合并是差价。像我们以前提到的，一个差价是两个期权，但是两个期权都是以不同价格的相同方向（也就是说，两个看涨期权或者两个看跌期权）。这种情况下的价格是什么？回到约期套购，它是一个看涨期权还是一个看跌期权？

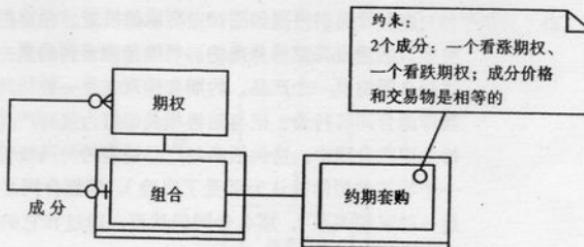


图10-7 作为期权子类型的约期套购建模

约期套购是一个看涨期权和看跌期权的合并。

图10-8显示解决这个问题的一个方法。那些在所有级别上都有意义的属性可以放在期权上，而难处理的属性被放到常规的期权上。这虽然在一定程度上有所帮助，但是如果我们考虑下面的情况，就会发现我们失败了：价格是在合同中而不是期权中确定的，而且有的组合期权（例如覆盖了看涨期权和保护了看跌期权）是期权和定期合同组成的。再次表明，泛化结构是可以处理的，但问题是什么可以被安全地放在超类型中。

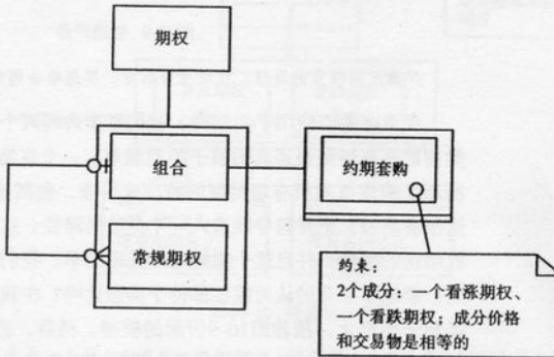


图10-8 子类型分为组合的和常规的

这些考虑足以对将组合和子类型化一起使用提出一个严重的问题。最

207

主要的问题是：当经营有风险时，交易人实际上并不关心组合。一个组合期权并不比其成分合同多什么。我们认为它的风险也是一样的，就像同一个合同夹中的合同被出售给不同的对方团体。正是客户和销售人员形成了组合，并且将合同看作是一个组合期权。一旦处理了组合，它的行为就和其它的合同没有什么区别。

这引导我们得到如图10-9所示的模型。在这里，销售人员的观点和风险管理的观点是截然分离的。风险管理看到的是一些合同，是销售人员把这些合同组成一个产品。约期套购现在是一种特殊的产品。这允许我们重新考虑合同的行为，把与销售相关的行为放到产品上，而把与风险相关的行为留在合同中。这包括参与产品销售的与风险管理不相关的团体（除非一个特殊的团体被认为承受了风险）。既然合同必须附带一个产品（由于是一种强制关系），那么合同仍然可以通过和它的产品的合作发现它的团体（但是请参见10.3.1节的讨论）。

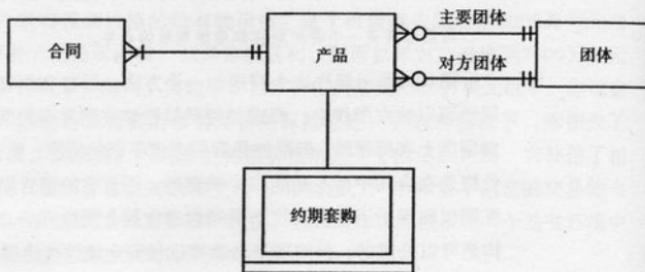


图10-9 引进产品

产品反映销售的展望。在风险分析中，要忽略合同如何被组合为产品。

在考虑是否使用子类型时，我们需要询问两个问题。第一个问题是超类型的所有特征是否真的被子类型继承。一个直接子类型化（例如图10-7所示）应该依靠所有超类型的特征来检查，包括超类型的超类型的特征。这很容易被忘记并且导致进入一个危险的路径。这种分析将会导致泛化层次结构的重构，并且这个重构可能并不简单。我们需要询问的第二个问题是：领域专家真的认为需要保持子类型化吗？在我们的例子中，领域专家抵制子类型化，推荐图10-9所示的模型。稍后，图10-8的风格再次出现，但是没有足够的说服力来改变这个模型（以及实现这个模型的框架）。

**建模原则：**只有当所有超类型的特征适合于超类型并且从概念上讲每一个子类型的实例是一个超类型的实例有意义时才应该使用子类型化。

208

这留下了一个关于是否值得在产品上赋予一些明显的泛化结构来描述不同种类组合的问题，如图10-10所示。很清楚，这不是用于风险计算的目的。然而，用于产生这种形式的新产品是很有用的。实际上，这种类型的泛化的最深刻的例子最可能存在于应用层和表示层（参见12.3节），此时需要特定的表示来描述定价和处理组合。在这种情况下，领域模型中一个共享的定义是很有价值的。即使这种定义当前只用于销售工作。更复杂的交易分析可能需要理解这些组合是如何定义的。

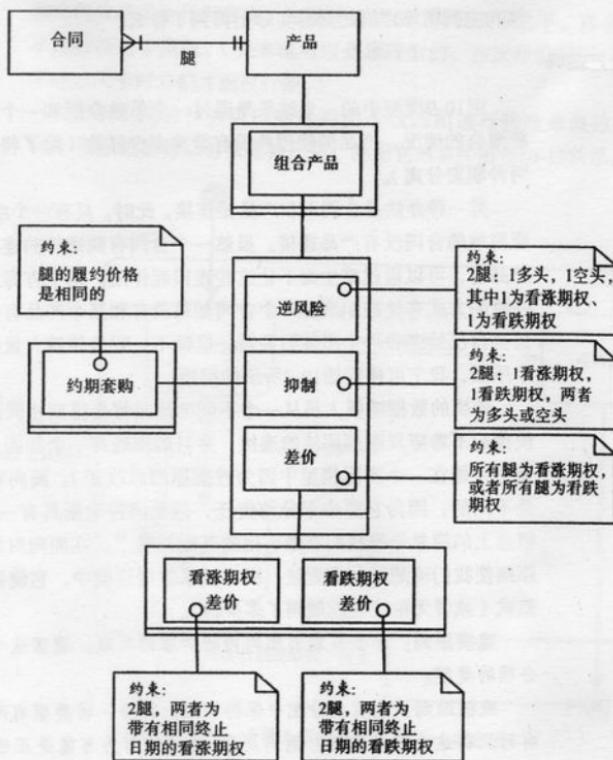


图10-10 普通组合产品

这是一个基于约束层次的好例子。连接到一个产品的合同称为产品的腿。

209

**例：**一个客户持有Aroma Coffee Makers的大量股票，他关心未来6个月在他可以出售这些股票之前的股票价格的浮动。他可能以现在大约5美元的

价格购买一个约期套购。对于交易人，这个产品被分解为两个分离的期权。

**例：**我希望购买7000股Aroma Coffee Makers的股票。一个交易人不能找到一个单独的希望出售相等数量股票的其他团体。他能找到一个出售2000股的团体和另一个出售5000股的团体。我有一个与这个交易人相关的产品：购买7000股。这个产品由两个合同组成：每个交易一个合同。

**建模原则：**当客户眼中的一个交易在交易人眼中是多个交易时，都可以使用产品/合同这种划分。

产品和合同之间的关键区别是产品描述客户的目的而合同涉及对方团体和主要团体之间的交易实际上得到了什么。

总需要有一个产品吗

图10-9模型中的一个结果是通过一个单独合同和一个单独产品来描述非组合的情况。产品的使用并没有带来多少好处（除了将销售和风险管理两种职责分离）。

另一种办法是合同不和产品相连接。此时，只有一个组合有一个产品。更简单的合同没有产品连接。虽然一个合同有到团体的连接，但是当存在产品时，可以通过派生而不让这些连接起作用。这个方案的缺点是它以不一致的方式来处理职责。一个合同如果没有和某个产品有代理关系，它就需要自己处理和一个团体的关系。这种不一致会导致大量的混乱。由于这个原因，我宁可使用图10-9所示的模型。

传统的数据建模人员从一个不同的途径将会得到相同的结论。标准化使他们不希望复制到团体的连接，并且因而选择一个如图10-9的模型（尽管它可能在一个实际模型中因为性能原因而改变）。面向对象方面的论据是不同的，因为它集中于分清责任，但是两种论据具有一个共同的主题：概念上的简单导致我们有最小化的基础关联<sup>①</sup>。在面向对象开发中，这个原则使我们清楚地分离责任；而在关系数据建模中，它使我们进入到第14范式（或者无论现在发展到了多少种）。

**建模原则：**不要复制有相同内涵的基础关联。遵循这个原则导致责任分明的类型。

**建模原则：**在责任分配中保持一致。提防一种类型有时对某事负责而有时又将这个责任委托给别的类型。（这种行为可能是正确的，但是它总是值得怀疑。）

## 10.4 子类型状态机

尽管许多普通的派生都可以被刻画为期权的组合，但是并不一律是这

① 我们可以有任意多的派生关联。

种情况。当在某个达成一致的市场定价（例如路透社的一个专栏）中被报价的交易物的价格达到一个特殊的极限时，一个关卡期权可以出现或者消失。因而，一个期权可以被买来以90 JPY/USD的价格来购买（看涨期权）1000万日元，这个期权在汇率突破85 JPY/USD时才可能被行使。这个期权的行为与一个标准的期权不同。实际上，这个期权不能被行使，除非汇率在终止日期之前降到85 JPY/USD。如果价格确实降到这个关卡，那么这个期权就被插入并且将会保留可行使的权利，不论在那个日期和终止日期之间价格发生任何改变。如果价格从没有降到关卡之下，那么购买者就不能行使这个期权。（关卡也可以是剔除型的，在这种情况下，只有汇率不超过关卡时它们才能被行使。）

这个不同的行为可以通过对关卡状态图进行修改来表达，使用图10-11所示的图可以有效地替代它。使用它的事件图如10-12所示。

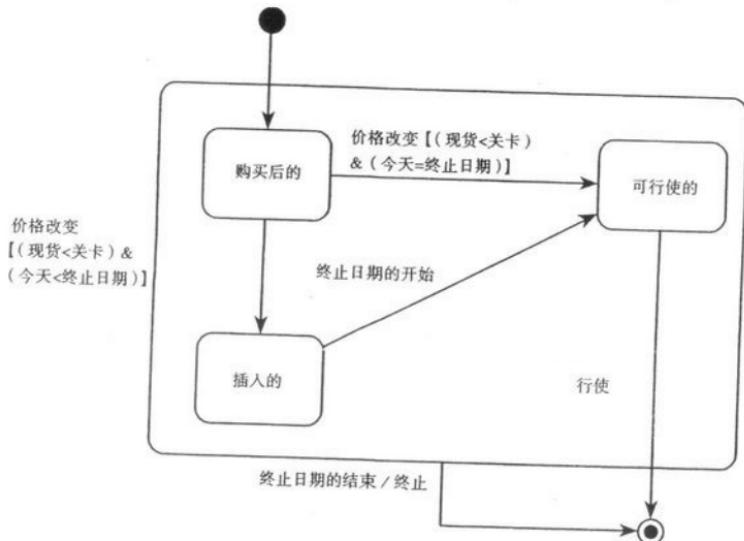


图10-11 关于一个插入型看涨期权的Harel状态图

如果某种交易物的价格从没有越过关卡，则期权不能被行使。而价格一旦越过关卡以后，则发生其它什么改变都是无所谓的。

惟一的结构改变是增加了这个期权的关卡标准，这作为一个期权的子类型确实工作良好，因为它在行为上提供了改变并且增加了一个新的特征（关卡标准）。

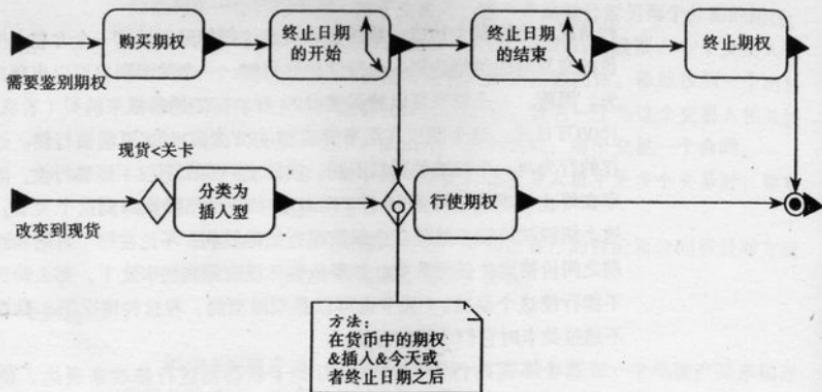


图10-12 使用一个插入型期权的过程的事件图

#### 10.4.1 确保状态图的一致

这种状态图本身引出一个有趣的问题。倘若关卡子类型有不同的行为，我们可以用图10-11所示的状态图来替代图10-4的状态图。然而，这带来一个问题：允许我们那样做吗？大多数方法强调支持用子类型替代超类型的重要性；这在对象图中通过只允许我们增加关联来反映，而不允许移动它们。许多教科书中没有提到可以用什么规则来管理带有子类型的状态图。Shlaer和Mellor[6]指出状态图只能出现子类型或者超类型。然而，如果所有的子类型共享一个共有的部分，那么把这个共有部分放在超类型中可能提高可维护性。Rumbaugh[5]指出子类型（通常）只可以增加正交的状态图。

关于子类型化与状态图之间的关系，Cook和Daniels[1]给出了最好的探讨，他们在子类型和状态图上花费了整整一章。他们强调“契约式设计”的原则[4]：这可以总结为一个超类型的状态图可以以两种方式进行扩展：或者通过增加一个正交的状态图，或者通过把超类型状态图的一个状态分割成子状态。超类型的变迁只能通过把它们重定向到它们超类型状态的子状态来更改。

应用这些方针到期权状态模型，我们会看到许多问题。第一个问题存在于对终止日期开始的事件的处理。在图10-4（期权图）中，这引起从购买到行使的变迁，但是在图10-11（关卡图）中，变迁起自新的插入状态。终止日期结束的事件有一个类似的问题：图10-4显示仅仅从可行使状态的变迁，而图10-11显示从任何状态的变迁。

第一个问题来自于要考虑一个对象如果处于的状态不能对收到的事件做出反应时应该做什么。这个对象或者可以默默地忽略这个事件，或者可以提示一个错误。应该提出一些普遍的策略解释如何处理这种情况；例如，Cook和Daniels[1]建议显式地列出与一个对象有关的事件。如果没有已定义的转移作为允许的事件列出，任何事件一般都应该被默默地忽略。这解决了当购买开始之后，图10-11（关卡图）中图表收到终止日期开始事件时会发生什么事情。如果终止日期开始是一个被允许的事件，它就正好可以被忽略。

然而，这还没有完全和超类型一致。图10-11显示当收到终止日期开始事件之后，一个购买了的期权变成可执行的期权。用合同中的术语来考虑，购买后的改变是终止日期开始的后置条件的一部分。在子类型中，我们不能削弱这个后置条件，只能加强它。对于一个插入型看涨期权作为一个期权的子类型，我们必须用图10-13和图10-14所示的那些模型替换这两者的状态图。

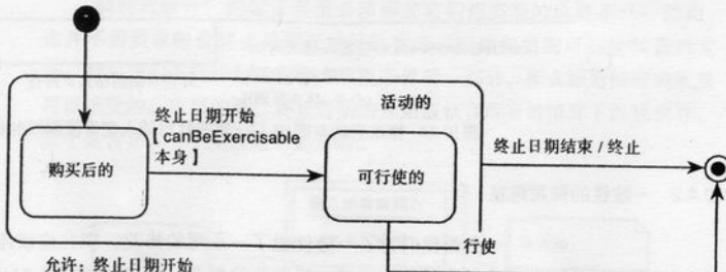


图10-13 为期权而更改状态图以便Cook和Daniels的一致性条件适用于插入型看涨期权

213

为了提供一致性，这两幅图反映两个改变。第一个改变是把购买后的和可执行的归纳到一个活动状态中。我们可以从这里重定向终止日期结束事件。第二个改变是增加canBeExercisable作为终止日期开始事件的守护操作。这个操作是表示终止日期的开始不能总是导致可执行状态的一个方法。对于定期期权，canBeExercisable总为真。期权的子类型可以由其它行为来覆盖它。

图10-14显示对于插入型关卡，这个覆盖如何发生。我们引入购买后的状态的子状态来指明关卡是否被插入。接下来我们分离终止日期开始变迁的来源，并且减弱守护条件以显示无守护的变迁。既然我们在超类型中允许终止日期的开始，那么当未插入时，关卡可以忽略终止日期的开始。

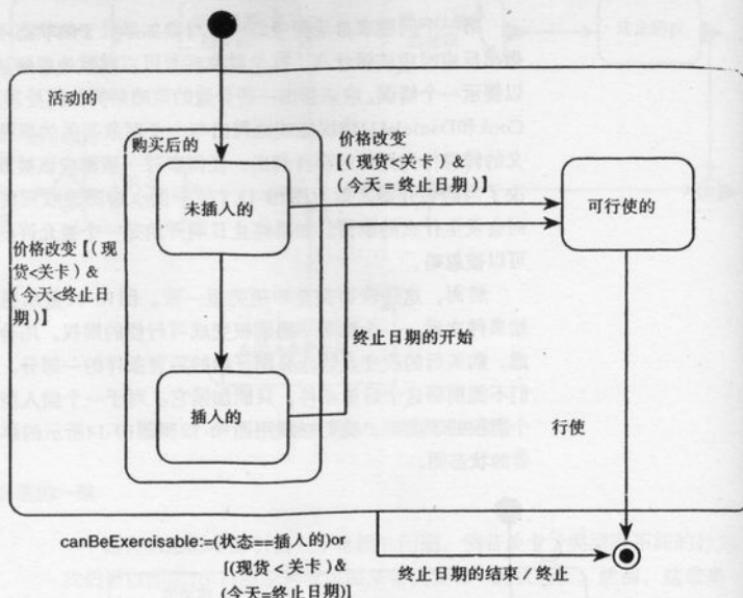


图10-14 修改后的与图10-13一致的针对插入型看涨期权的状态图

#### 10.4.2 一致性的使用问题

上面我们为了一致性做了一系列的修改，现在应该停下来问自己关于这个过程的一些问题。根据我的判断，图10-4和图10-11比图10-13和图10-14描述了更简单和更清楚的行为表达。因而，尽管我们得到了一致性（至少根据Cook和Daniels的定义），但是我们失去了易理解性。另外，对插入型看涨期权的建模导致我们改变超类型图表。这已经是我们所能达到的最好结果——因为我们需要一个不同的状态图来强制我们构造一个一致的子类型状态图，所以我们仅仅为此而改变了它。这意味着一个新的子类型可以强制我们来改变超类型状态图，否则我们必须足够聪明来产生一个非常灵活的超类型状态图。不幸的是，我不认为我足够聪明，所以子类型化将充满困难。

这些困难的一个解决办法是重做泛化层次结构来避免需要对一致性产生的担心。我们假定一个插入型看涨期权将是一个期权的子类型，每一个都有它们自己的状态图，如图10-15所示。另一个方法是在没有自己的状态图的情况下将一个期权作为一个抽象类型来对待，并且产生一个

常规的期权子类型来支持图10-4的状态图，如10-16所示。这避免了对状态图一致性的担忧，允许更自然的状态图，但是的确引入了一个新的类型。这也和Rumbaugh和Shlaer以及Mellor的原则更一致，但他们没有讨论状态模型之间的一致性。

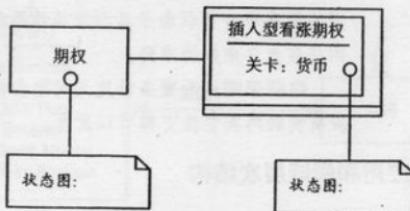


图10-15 插入型看涨期权作为一个期权的子类型

这是一个自然的方法，但是状态模型之间如何联系呢？

“契约式设计”约定子类型必须满足它们超类型的后置条件。然而，这并不需要意味着终止日期开始的后置条件应该包括到可行使状态的变迁。如果我们不选择包括它作为后置条件的一部分，那么原始的图表就是可以接受的。重要的是，终止日期的开始应该在所有的情况下都被允许；至于是否导致一个变迁是不确定的。215

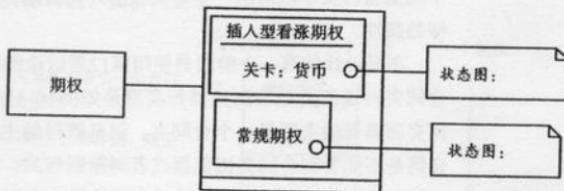


图10-16 产生一个常规期权类型

这可以更简单地处理状态模型，但是它并不自然。

实际上，这是一个关于“契约式设计”的更广问题的例子。通常，人们说一个操作的后置条件应该定义到一个对象的所有可观察到的状态的改变。形式化方法的研究团体通常很推崇这个原则，但是对“契约式设计”却兴趣不大。后置条件仅仅指定在操作结束时必须支持的状态。我们能经常声明除了指定的东西之外没有东西必须被改变，但是在这个方法中并没有假设必须是这样。

事实上，子类型化使这样一个限制性的后置条件很危险。子类型化的

一个总的特点是超类型不能预知子类型可能进行的所有扩展。使用一个过度限制性的后置条件会削弱子类型化提供的灵活性。后置条件定义了这个对象的可观察到的状态的各方面都必须为真。因而，如果它们不违反后置条件的直接条款，则任何其它的改变都可以发生。

**建模原则：**关于状态图的泛化结果不能被很好地理解。确保关于超类型的所有事件可以由子类型来操作是很重要的。任何可被子类型化的状态图必须允许未知的事件。

**建模原则：**后置条件定义对象在操作之后必须为真的条件。后置条件没有提到的其它改变都可以发生。

## 10.5 并行的应用和领域层次结构

面对包含不同合同的一个合同夹，一个交易人往往喜欢看到把合同和有关合同的重要信息放在一起的列表。这样的一个列表应当把每个合同显示在一行上。并且这行上的信息应当随合同的种类而变化。列可能是多头/空头、交易日期、履约价格、看涨/看跌（只针对期权）、终止日期（只针对期权）、关卡层次（只限于关卡）、插入型还是剔除型（只限于关卡）。

在这个方案中，表格中的一些列只是和某些期权的子类型相关。这增加了这个问题的一定复杂性。我们不能做的是假定某个代表浏览器行的类向每一个合同请求每一种相关的属性。这样一个方法将不会工作，因为这个浏览器行类不能请求一个非期权的终止日期，根据定义，非期权没有这样的属性。

布局设计的第一个阶段是使用第12章讨论的层次结构。在这个结构中，合同夹浏览器类型和浏览器行类型是如图10-17所示的应用外观模式。合同夹浏览器的主题是一个合同夹，浏览器行的主题是一个合同。合同夹和合同是不能看到合同夹浏览器或者浏览器行的，原因是合同夹和合同位于应用层次，并且领域类型应该对应用类型没有可见性（参见图12-6）。

这个结构允许一个浏览器行有接口需要的所有列的属性。根据一个表示层的程序员的考虑，每一列都有那些可能会为空的属性。如果一个属性为空，那么意味着在浏览器表中的一个空白的空间。问题存在于浏览器行和领域模型之间的连接中。

浏览器行知道它处理的是合同的聚合。但是，它需要请求只在某种合同的子类型上定义的信息。如果一个浏览器行请求一个非期权的终止日期，它将会得到一个错误。有几个策略可以用来解决这种交互：应用外观的类型检查，给超类型一个包装性的接口，使用一个运行时属性，使应用外观对领域模型可见，以及使用异常处理。

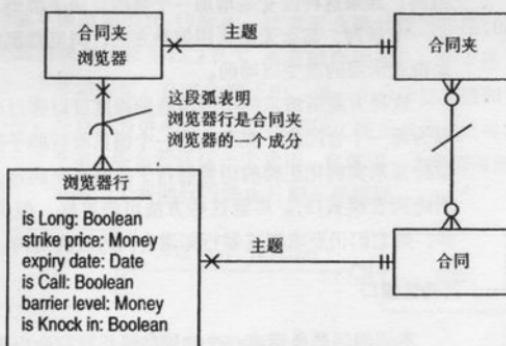


图10-17 一个合同夹浏览器及其与领域模型的关系

一个合同夹浏览器和浏览器行是应用外观。

217

## 5.1 应用外观的类型检查

在这个策略中，由浏览器行负责处理这个问题。在每一个对合同的请求之前，都要做对合同的类型检查来确保这个请求可以被安全地发送，如图10-18所示。在C++中，这采取某种形式的类型检查，接着转向子类型的强制类型转换，接着发请求。

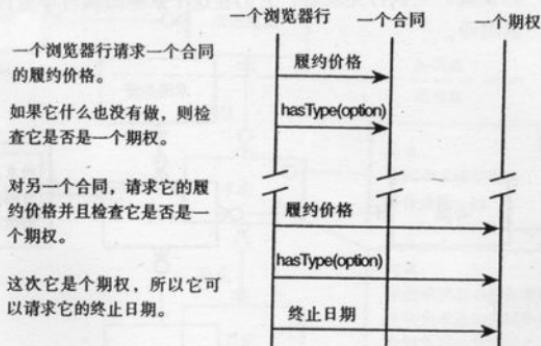


图10-18 在浏览器行中类型检查的交互图

在调用一个只在子类型中定义的操作之前进行类型检查。

这个策略有许多缺点。浏览器类在有多种合同的子类型的情况下会变得十分复杂。此外，合同层次结构的任何改变都会引起浏览器类的改变。

当然，如果这种改变是增加一个新的合同子类型，而导致要在浏览器中引进一个新列，那么无论采用何种方法，浏览器都需要改变，因为这种改变是由表示层的改变驱动的。

这种方案所暗示的类型检查的程度可以通过两种方法来简化。我们可以为每一个合同的子类型使用一个浏览器行的子类。我们可以使用一个类型检查来实例化正确的浏览器行子类型来完成这个工作。另一个方法是使用访问者模式[2]。尽管这些方法可能更好，但是如果类型检查的次数过多，则它们仍要求浏览器行知道合同的层次结构。

#### 10.5.2 给超类型一个包装性接口

本质的问题是请求一个合同的终止日期会出现错误。一个解决办法是在合同中增加对合同的所有子类型操作。合同将会自然地用空值来回复所有这些操作，但是相关的子类型可以覆盖这些操作以便提供它们的数值。

这个方法有许多问题。几乎不可能分辨出对合同的一个真正合法的操作和什么是一个真正的错误。编译时的类型检查也不能用了，因为它不能分辨出合同的子类型是什么。每一次一个子类型被引入，合同的接口就必须被改变。因而我不是这种方法的支持者。

#### 10.5.3 使用一个运行时属性

运行时属性提供一个为类型增加属性而不改变概念模型的非常灵活的系统。当执行完成后，它们在这个系统的执行中允许属性改变而不重新编译。

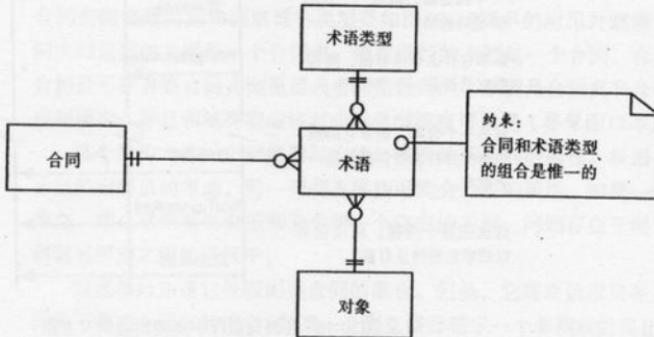


图10-19 合同的一个运行时属性

在使用这种方法的情况下，向一个非期权请求一个只在期权上定义的属性将不会导致错误。

合同的基本模型如图10-19所示，或者更合适的模型如图10-20所示，它使用带键值的映射（参见15.2节）。所有的合同都有一些术语，并且每一个都对应一个术语类型。在这个例子中，每一个合同的属性和它的子类型（履约价格，是否是看涨期权，关卡层次等）将会成为术语类型。如果向合同请求一个术语，那么若存在一个术语，它就用一个数值对象来回复。在这种方法中，请求一个非期权的终止日期不是错误。

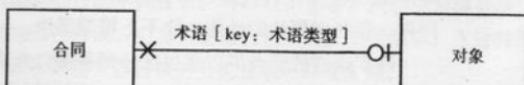


图10-20 图10-19使用一个带键值的映射

219

当然，这个模型允许一个非期权偶然被给予一个终止日期。这可以由两种方法来防止。第一种方法是使用一个知识级（参见2.5节），如图10-21所示。另一种方法是产生这个术语类型作为一个派生接口。模型属性（那些关于合同和其子类型的属性）和术语类型接口都被提供。更新只能通过模型属性来提供。

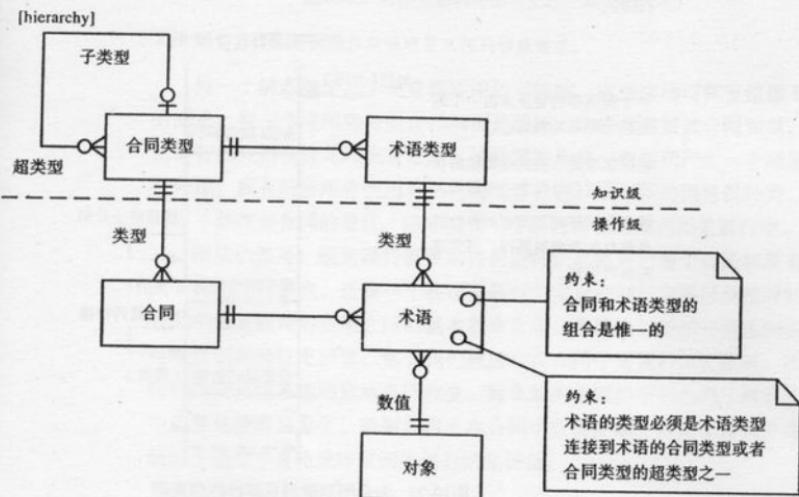


图10-21 使用一个知识级来控制将术语放置到合同上

这将会阻止术语被不正确地放置到合同上，但是只能在运行时被检查。

使用运行时属性确实提供了灵活性但是它也带来了重大的缺点。第一，

使用术语类型增加了合同的接口并且它的子类型更难以理解。一个合同的使用者在注意类型上定义的操作的同时，还必须注意术语类型的实例，而实际上该术语类型的实例是有效的。第二，属性类型不能在编译时进行类型检查，失去了编译检查的这样一个重要的优点。这对于浏览器行无关紧要，因为根本的目的就是放松任何的编译时检查，但是对系统其它部分，它的影响却很大。第三个缺点是搅乱了基本的语言机制。编译器不清楚将继续进行的是什么，并且语言特征(例如多态)必须由程序员进行硬编码。同样，运行时属性的性能也比不上模型属性。

许多这样的缺点可以通过区分两种接口来减轻。那些在编译时间知道属性的软件部分可以使用模型属性，并且浏览器可以使用运行时属性。

#### 10.5.4 使应用外观对领域模型可见

在这种方法中，加载一个浏览器行的责任被委托给需要这个浏览器行显示的合同，如图10-22所示。因为控制目前在合同中或者在它的子类型中，所以它可以用子类型的正确值来加载浏览器行。浏览器行支持所有必需的应用信息，并且合同或者它的子类型知道对于那个子类型什么是可应用的。

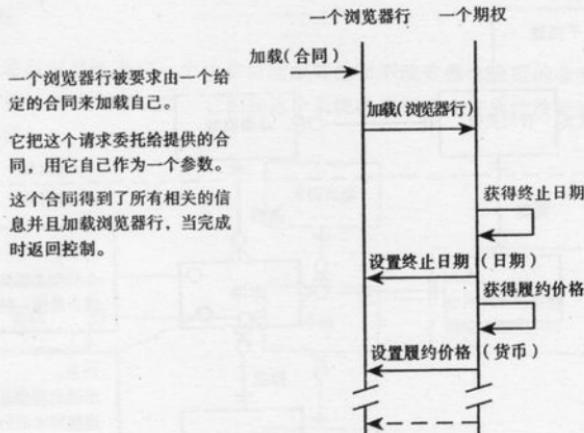


图10-22 为合同加载浏览器行的交互图

浏览器行必须对合同是可见的，这违反领域层和应用层之间通常的可见性规则。

这个方法的优点包括更简单的交互，因为不需要类型检查，并且合同不需要一个更复杂的接口。另外，增加一个新的合同不需要浏览器行的改

变，除非在表达中有一个相应的改变。全部所需只是一个加载浏览器行的覆盖操作。

最大的缺点是破坏了在第12章讨论的应用层和领域层之间的可见性。通过放置浏览器行在它自己的包中可以避免这个缺点，如图10-23所示。在这种方法中，来自领域模型对显示的依赖只被限制在浏览器行类型上。通过将浏览器行类型一分为二，可见性可以进一步减少。浏览器表示和合同使用浏览器行的不同接口。可以在浏览器外观包中为浏览器行提供它自己的浏览器行外观，它与浏览器行接口的交互很简单。在这种情况下，从浏览器表示到浏览器行包的可见性就可以被去掉。

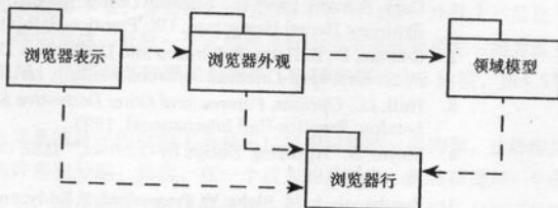


图10-23 对浏览器行包的可见性（基于图12-6）

浏览器行包是应用层和领域层之间的特殊情况。

另一个缺点源于几个浏览器应用的可能性，这些应用可能有稍微不同的需求。每一个应用将需要自己的浏览器行，这些都需要被合同知道。浏览器行的分割在这儿可能有帮助。这样需要为每一个应用产生一个浏览器外观，所有的应用将使用单一的浏览器行包。增加新的浏览器种类，从而将不能改变合同的责任，除非要将一个新的特征增加到浏览器行中。

事实仍然是：浏览器行需要的任何新特征都需要对整个合同和所有它的子类型进行修改。这是一个在浏览器行中安置控制的方案以及相对的在合同中安置控制的方案之间的基本折衷方案。如果增加新的合同比浏览器行增加新的特征更频繁，那么我们就应该在合同中安置控制。然而，标准模式的可见性不能随意地进行改变。除非新的合同类型的出现比浏览器行中的变化更容易发生，否则我将不在合同中安置控制，因为在合同中许多新的子类型本身将意味着浏览器行的新特征。

#### 10.5.5 使用异常处理

当然，上面的所有想法都是基于这样一个想法：即向一个非关卡请求它的关卡级别是一件不好的事情。然而，如果有合适的环境，这就不是一个问题。如果进行一个对象的请求导致一个运行时错误并且错误通过一个

异常表现出来，那么浏览器行就可以简单地捕获异常并且把它看作一个空值。浏览器行应该检查出：异常实际上是由于接收者不理解这个请求引起的，而不是其它的更令人担忧的错误。这还假定了编程语言允许发送一条消息到没有接口的接收者对象。在这种情况下，如果和异常处理功能（现在很多新的实现都支持这种功能）相结合，缺乏类型的安全性反而成了一个优点。由于是没有类型的，因此Smalltalk语言就可以这样使用。类型安全在C++中可以通过使用一个向下类型转换来绕过。

## 参考文献

1. Cook, S. and J. Daniels. *Designing Object Systems: Object-Oriented Modelling with Syntropy*. Hemel Hempstead, UK: Prentice-Hall International, 1994.
2. Gamma, E., R. Helm, R. Johnson and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.
3. Hull, J.C. *Options, Futures, and Other Derivative Securities* (Second Edition). London: Prentice-Hall International, 1993.
4. Meyer, B. "Applying 'Design by Contract,'" *IEEE Computer*, 25, 10 (1992), pp. 40-51.
5. Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Englewood Cliffs, NJ: Prentice-Hall, 1991.
6. Shlaer, S. and S. J. Mellor. *Object Life Cycles: Modeling the World in States*. Englewood Cliffs, NJ: Prentice-Hall, 1991.

# 交 易 包

为了充分理解本章，你需要先阅读第9章和第12章。开发大型信息系统提出了特殊的挑战。处理一个大规模系统的基本方法是把它分解为若干个更小的系统。这需要某种形式的构架（软件体系结构）建模，如A.5节中所述。

任何信息系统首先的组织工具是第12章中讨论的分层构架。这种构架确定系统的许多包分割。然而，在一个较大的系统中，领域模型对一个单独的包太大了。本章着眼于如何能分割一个巨大的领域模型。包和可见性的概念（参见A.5节）再一次作为分割的基本工具而被采纳。第9章的交易概念提供了例子。

第一个模式着眼于如何组织场景和合同夹的模型。主要问题是关于对一个包的多重访问级别（参见11.1节）。一个风险管理应用使用场景来得到评价合同夹所需要的信息。另一个应用需要建立并管理场景。两个应用都需要访问场景类型，但是需要非常不同的访问级别。不同的客户需要不同的接口是一个很普遍的问题。解决办法包括允许一个包有多重协议以及使用不同的包。

合同和团体之间的关系提出了相互可见性（参见11.2节）的问题。显然有三种解决办法：合同和团体之间单向可见，把它们放置到一个相同的包中，或者把它们放置到不同但相互可见的包中。所有三种解决办法都有严重的缺点。

最后的模式是通过考虑如何把第10章讨论的派生安置到包结构中来探究包的子类型化（参见11.3节）。这个模式阐明可以将子类型放置到一个和它们的超类型分离的包中，这个包具有从子类型到超类型的可见性。

[225]

## 11.1 对一个包的多重访问级别

从合同构造合同夹要使用市场指示器作为描述。独立地使用场景，以便为市场指示器开发价格。合同夹和合同需要使用场景来进行自我评估，但是场景不需要任何有关合同夹和合同的知识，如图11-1所示。

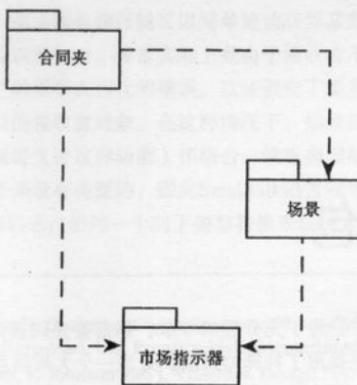


图11-1 包可见性的初始图

为了展开评估，一个合同夹只需要市场指示器的价格。合同夹包不需要知道如何建立场景。因而，虽然要使getQuote消息可以被发送，场景元素类型需要对合同夹可见，但是不需要看到带有有关报价如何形成的定义的子类型。当然，我们可以更深一步并且说甚至场景元素对合同夹也是没有用的。一个更好的方法是让合同夹看到如图11-2所示的接口。这个接口有一个在场景上的带键值的映射（参见15.2节），它把市场指示器作为一个参数。既然场景元素的其它属性都是不重要的，那么合同夹包的接口就简单了。

[226]

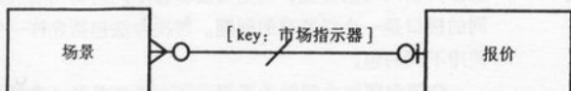


图11-2 隐藏了场景元素的场景包的一个接口

这是不需要知道场景元素的合同夹包的最好接口。

这个方法需要两个不同类型的场景包：一个用于合同夹的接口，另一个用于建立场景。因此，除了把类型简单地分配到包中还需要其它的东西。直接和明显的方法是把场景包中的类型划分为包中的公有类型和私有类型。公有类型是那些对其它包（这些包可看见场景包（如合同夹））可见的类型。私有类型只是对场景包内的类型可见。在这种情况下，场景是公有类型而场景元素是私有类型。这个逻辑可以扩展到操作。公有操作在一个包内是公有的并且对其它包也是公有的。虽然这说明了一个精细等级的

控制，但是它还是太难维护。要设计出良好的可见性的诀窍是选择一个足够精细被使用的可见性等级，但是不要精细到使管理合同夹成为一个恶梦。（很难管理的事物往往不被管理，这会导致过时的、不可用的模型。）

用这种方法的一个问题是使用者需要软件来建立并且使用场景。这需要在应用逻辑层和表示层上的构件，如第12章所述。因而，这个模型必须包括一个从场景包分离的场景管理应用包。图11-3显示新增一个场景管理应用包和一个风险管理应用包的情况。然而，这种方法用上面描述的公有/私有方法将不能工作，因为场景管理应用需要场景包的私有类型。虽然合同夹和场景管理都需要场景包的可见性，但是它们需要不同的可见性类型。

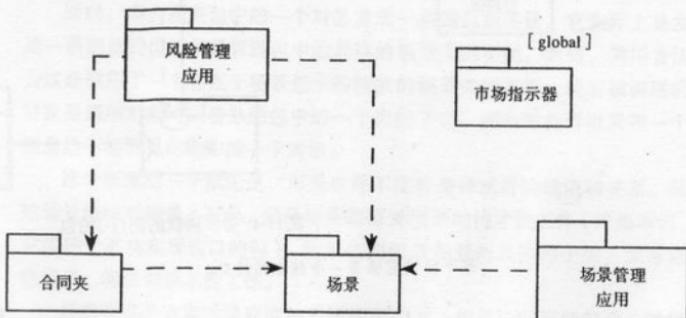


图11-3 在图11-1中增加应用包

这里的问题在于：风险管理应用需要的场景包接口比合同夹包需要的场景包接口大得多。

根据Wirfs-Brock[1]建议，这个问题的一个解决办法是一个包拥有一个以上的协议<sup>①</sup>。在我们原始的模式中，我们把一组操作作为协议；然而，完全有理由只把一组能够对可见性进行更好控制的类型作为协议。使用分离的协议导致如图11-4所示的图，图中的场景有两个协议：一个是只允许合同夹使用的小协议，而场景管理应用使用更深层次的协议。协议通过包的外框中的半圆端口显示。（我只在具有不止一个协议的包中显示端口。）

使用分离的协议是处理多重可见性问题的一种方法。另一种方法是引入一个附加的包，如图11-5所示。场景元素和它的类型从场景包中移动到

<sup>①</sup> Wirfs-Brock用的术语是“合同”，但在这个例子中用“合同”会产生歧义，所以我用了术语“协议”。

228

场景结构包中。场景包只包含场景类型和它的简单关联。合同夹包只有场景包的可见性，而场景管理应用可见场景包和场景结构包。我们可以定义具有额外可见性的新场景。

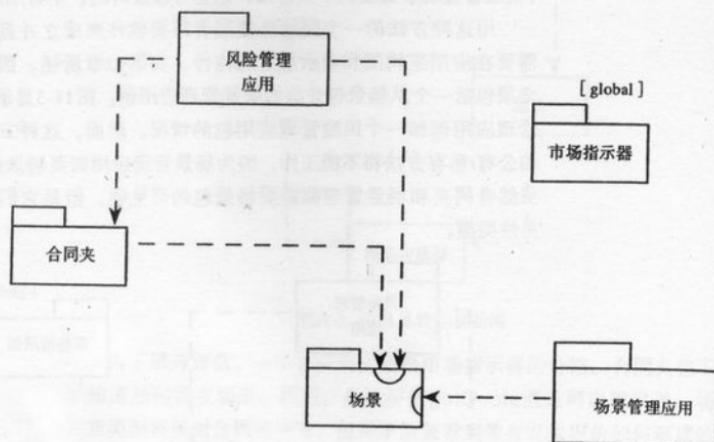


图11-4 带多协议的图11-3的包

每个协议意味着一个独立的接口。

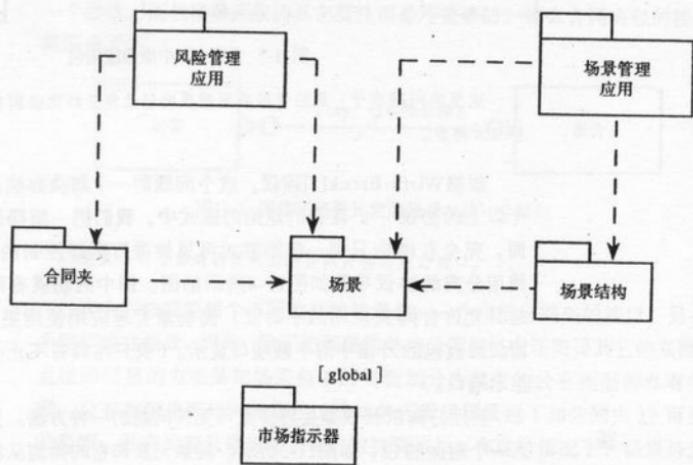


图11-5 使用一个针对场景结构的附加包

细心的读者可能会发现一个问题：场景是否需要对场景结构的可见性。回答一个报价的请求需要使用内部结构。这些可见性体现继承和多态引人入胜的方面。场景包可以包含一个场景类，这个类定义场景可见的所有包需要的接口。然而，这个场景类不需要所有接口的实现（因而是抽象的）。我们可以在执行接口的场景结构包中放置第二场景类。这个第二场景类对场景结构中的所有内容有完全的可见性。另一个包使用的任何场景对象都是场景结构中的场景类的一个实例，但是这个类的那些客户并不认识这一点。所有他们看到的都是和场景包中场景类的接口一致的对象。为了显示这种子类化出现在越过包边界的什么地方，可能值得提供一个符号，尽管我并不使用这个符号。

所以，当合同夹包中的一个对象发送一条消息到场景，它实际上是发送一条消息到位于场景管理包中的具体的场景类的实例。然而，调用者认为这是调用了一个存在于场景包中的抽象的场景类的实例。倘若被调用的对象是调用对象可以看见的包中的一个类的子类，那么对象可以发送一个消息给不能看见的包中的一个对象。[229]

这个原理的一个推论是：可见性并不反映编译或者加载依赖关系。虽然场景结构对场景不可见，但是场景却需要场景结构才能工作（严格地讲，它依赖于某些实现接口的包）。场景结构包含场景的具体的子类，没有这些场景，场景包就不能工作。

虽然在这个方案中需要两种不同的场景类，但是它们可能符合一种场景类型。在本例中，需要提供一个新的类型，使得像场景管理这样的应用可以访问场景的内部结构。然而，当其它的类型不需要调用特殊的只在子类型中出现的特征时，就有可能只拥有一个类型。

## 11.2 相互可见性

为合同和团体增加包会出现更复杂的问题。把场景和合同夹分别放在独立的包中有两个原因。第一，场景和合同夹看起来是构成模型的单独小块。它们本身是一些复杂的部分（看起来构成一个操作单元）。第二，我们不需要任何合同夹的知识来构造一个场景的模型。第二个原因是主要的，因为它导致图11-1所示的可见性关系。

完全有理由断言：合同可以被放置在一起，而且无需合同夹的知识就可以对合同建模。可以用独立于合同夹的动态结构来记录合同，虽然合同夹是为了风险评估目的而用来组织合同的，如图11-6所示。[230]

团体和合同之间的关系是一个更大的问题。我们有理由为团体设立一个单独的包。许多应用可能寻找关于团体的信息而不想知道任何被他们执

行的交易的事情。一个通用的团体包可能保存关于被许多交易系统使用的团体的通用信息，这比使用一个合同数据库要好。因而我们可以推断一个团体包是有价值的。

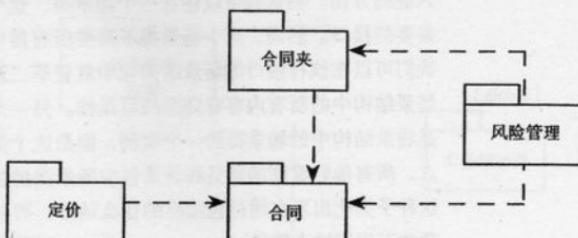


图11-6 合同夹和合同的包

风险管理应用需要所有两个包，但是定价应用只需要了解合同。

团体和合同包之间将会是什么关系？对于一个团体，分辨哪一个合同来处理它，对于一个合同，分辨谁是合同适合的团体，都将是很有价值的。这意味着团体和合同之间的相互可见性，如图11-7所示。但是相互可见性在一个包模型中可能会导致问题。从整体来看，我们设法用一个分层的构架和单纯的可见性线段来设计包模型。许多人相信这样一个构架在可见性关系中绝不应该有回路，因为回路打破了清楚的层次规则。相互可见性是回路的一种最简单的情况。

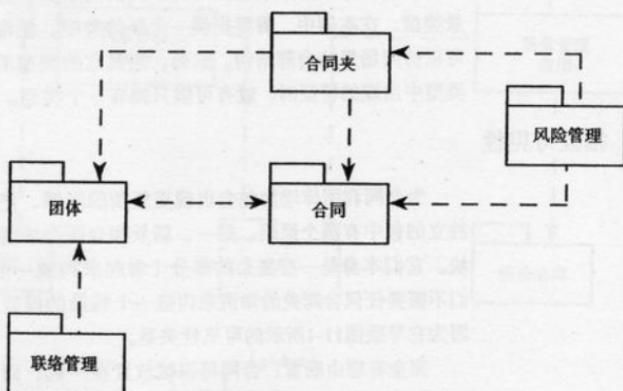


图11-7 独立的团体和合同包

一些应用只需要团体包或者合同包，但是这两种包是互相需要的。这就意味着相互可见性。如果相互可见性是不可接受的，我们可以选择一个方向或者合并这些包。

为了消除相互可见性，我们必须改变团体的属性或者合同的属性，使得只有一方知道另一方，或者合并它们到一个单独的包。每一种替代方法都要做出一些权衡。

[231]

限制团体和合同类型之间的可见性为单向的，其好处是减少了两种类型（和它们各自的包）之间的耦合。如果我们消除了从团体到它的合同的映射（使关联是单向的），我们就可以在团体包上工作而不需要知道任何关于合同的事情。这就减少了耦合（团体不再和合同耦合），这是一个优点。然而，一个想知道某个团体是哪些合同的对方团体的用户必须查看每一个合同，并且使用到团体的映射来形成集合。因而，我们虽然减少了团体包开发者的复杂性，但是却增加了对需要使用两种类型的任何应用的开发者的复杂性。这里没有绝对正确的答案：我们不得不考虑在每一个方向上的权衡并决定哪一个选择有较少的负担。

**建模原则：**单向关联和双向关联之间的决定是减少相关类型的开发者的工作（通过减少类型间的耦合）和为类型的使用者提供方便之间的折衷。

假定我们决定赞同双向关联，我们消除相互可见性的惟一办法是合并团体和合同包。然而，这个方案并非没有缺点。在图11-7中，我们可以看到联络管理包只需要知道团体，而不是合同。合并这两个包将会消除这个信息。联络管理将被强加比它需要的可见性更大的可见性。

这种情形使我不禁禁止相互可见性或者其它回路。当然，回路数目应该被减少到最小。然而，完全消除它们会导致：或者强加单向关联和双向关联之间的折衷方案，或者客户不需要的包含所有可见性的大包。

**建模原则：**如果一个包只需要对另一个包的一部分的可见性，那么考虑将后面的包分成两个相互可见的包。

图11-8所示的是这种情形的另一个例子。将产品（参见10.3节）加到它自己的包中。前面的论述导致产品、团体和合同之间的相互可见性。这导致领域模型包的完全耦合。然而，应用包只需要了解整个模型的一部分，并且每一个应用包都有稍微不同的需求。这三个相互可见的包使我们清楚地了解这些需求。

另一个办法是在包上提供多个协议。这样，团体包、产品包和合同包合并在一起，并为原来的三个包提供三个独立的协议。应用然后选择协议，选择方式与其在图11-8中选择包的方式相同。

[232]

总而言之，当类型自然地紧密耦合时，我们有三种选择。我们可以通过采取单向关联来降低耦合（但是这种方法给类型的用户造成更多的困难）。我们可以把它们放置到一个单独的大包中（但是这意味着包的任何

用户对整个包都有可见性，即使只需要包的一部分）。我们可以有两个相互可见的包（但是这在包结构中引入了回路）。如果你有多个包的多个协议，你可以拥有一个使用独立协议的大包。

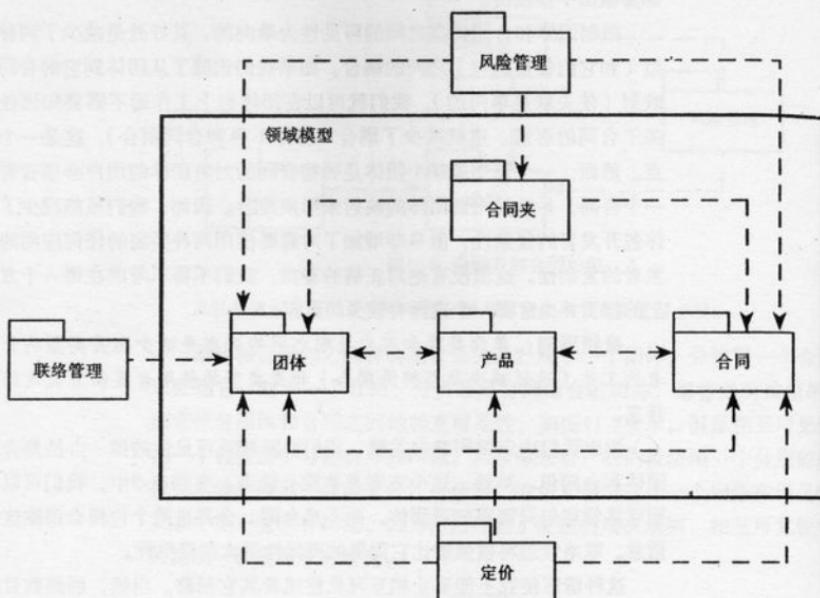


图11-8 在包中增加产品

再次看到，相互可见的包可以为不同的应用需求提供服务。

### 11.3 包的子类型化

在进行子类型化时，可见性是最容易考虑的。子类型永远需要看到超类型，但是我们应该避免相反的情况。因此，我们增加了组合、期权和卡片（在第10章描述），如图11-9所示。

我们还应该避免子类型和它的超类型之间的相互可见性。子类型化的全部要点是允许扩展一个类型而不让超类型知道它。如果我们设计类型时让超类型知道它们的子类型，那么将来的特殊化可能变得更困难，因为我们要把关于子类型化的假设构造到超类型中。任何消除这种依赖的努力在以后的增加中将得到加倍的补偿。正确地设计超类型通常需要在首先设计少许子类型时积累的经验，所以最好不要专注于超类型，直到把少许子类型组合在一起后再转向超类型。

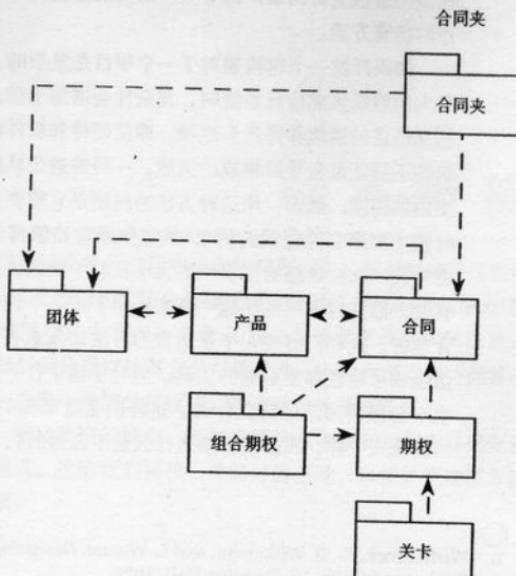


图11-9 增加不同种类的期权

子类型需要对它们的超类型可见，但是不能反过来。

## 结论

可见性总是意味着折衷方案。对可见性的限制减少了模型浏览的方便性。如果有很多单向的可见性，浏览整个模型就像是在一个有很多单行线街道的城市里游览。双向可见性使浏览更加容易，这意味着可以维护更少的代码。然而，这样的可见性也有代价。系统的越多部分互相可见，在模型中控制变化的后果就越困难。限制可见性削减了这种相互依赖性。[234]

不同的面向对象建模人员采取不同的折衷方案。一些折衷方案在很大程度上限制可见性，使用如单向关联和类型可见性图表的技术。我发现这种限制性太大。我考虑的是包的可见性而不是类型级别的可见性。第12章描述的构架把一个系统分割为基本的层次。在领域层次中，可以使用更进一步的可见性限制，但是这常常不简单。然而，因为我的信息系统经验，我更喜欢这种方法。开发的其它种类会有不同的折衷方案。

大多数项目不会很仔细地考虑包构架。通常只有构架的基本层次，甚至什么也没有。这导致项目相关的缺点，并且使评估一个适当实施的构架

模型的价值变得困难。只有更多的实践才能允许我们更进一步理解这个讨论的折衷方案。

如果开发一个包构架对于一个项目是复杂的，那么当我们试图为一个巨大的组织集成信息系统时，复杂性会增加十倍。大型组织会被多个不能相互通信的系统弄得焦头烂额。即使硬件和软件已经成形，系统的基础概念的不同还是会导至集成的失败。一种普遍公认的解决办法是进行企业范围内的建模。然而，用这种方法的问题是它需要太长的时间。到它完成的时候（如果它能完成的话），其工作通常是值得怀疑的和延期的。我认为能够被一次性处理的建模模块的大小是有上限的，并且这与提交有用的系统有关，而交付有用系统这一事实证明了在一个正常的期限内建模的花费是合理的。这样，需要一个更适宜的方法来集成它们。对于这个任务，我相信包和可见性都是必需的工具。但对于这个任务光有它们是不够的，而我也不会假装知道还需要什么。这样的企业范围的集成我们仍然知之甚少，和其他人一样，我也仅仅知道什么是不该做的！

## 参考文献

1. Wirsfs-Brock, R., B. Wilkerson, and L. Wiener. *Designing Object-Oriented Software*. Englewood Cliffs, NJ: Prentice-Hall, 1990.

# 第二部分

## 支持模式

分析模式讨论分析过程中遇到的问题以及一些可以处理这些问题的模型。支持模式处理那些围绕分析模式建立计算机系统的过程中出现的问题。在第12章和第13章中考虑一个客户/服务器信息系统的构架以及这样一个系统能够被如何分层以改善它的可维护性。第14章考虑如何能够实现一个概念模型，提出一些把分析模式转换成软件的通用模式。

最后，第15章更加抽象，考察建模技术本身和如何能够把先进的建模结构看作模式。这给我们提供一个更好的基础，以便扩展建模方法来支持特殊的需求。

237  
238



## 信息系统的分层构架

本书中的分析模式将对企业信息系统的开发者具有极大的价值。然而，信息系统（IS）开发所包括的不止是对一个领域的理解。我们必须要适应一个由大量用户、数据库和遗留系统组成的世界。本章讨论信息系统的构架模式。构架模式描述的是把一个系统分成主要子系统的高层次分割和子系统之间的依赖性。信息系统构架模式把系统分成几个层次。构架模式本身是有用的，但是它们也显示分析模式如何适合一个更广泛的范围。第13章描述使用本章中模式的一种技术。

对象技术的早期并不太关注IS开发。主要问题是大量的通常复杂的信息必须被很多人共享。即使这个信息被共享，不同的用户还会有不同的需求。提供能够被局部裁剪的公共信息是大型信息系统的一个主要目标。此外，需要大量的适应性来满足不断变化的信息需求。大多数信息系统的的主要任务是维护，维护主要包括处理变化的信息需求。在这些环境中，对象技术的主要优点不是在于建立新系统的速度而是在于减轻了维护的负担[3]。

在开发一个现代信息系统中，最基础的问题是理解潜在的软件构架。对适合信息系统的软件构架的一个明朗的描绘必须在关于使用哪种技术或者考虑什么过程的任何讨论之前。

239

大多数的IS开发都默许地采取一个两层构架（参见12.1节），它源自大型机交互系统并且现在通用与客户/服务器的开发。尽管两层构架应用广泛，但是它还是有很多缺点，这是因为用户界面紧密地耦合于物理数据设计。三层构架（参见12.2节），也称为三模式构架，通过在用户界面和物理数据之间放置一个中间层来处理这个问题。这个领域层精确地对问题领域的概念结构进行建模。对象技术特别适合于三层方法，而且领域层既可以放置在客户端的机器上，也可以放置在服务器端的机器上。

接下来我们把注意力转移到应用上，它操纵领域层的对象并且在用户界面上显示信息。这两个责任能够被用来把应用分成表示层和应用逻辑层（参见12.3节）。应用逻辑能够被组织成领域层上的一系列外观，每种表示

有一个外观。这个分割具有很多优点，并且应用外观能够被用来简化客户/服务器交互。

数据库交互（参见12.4节）能够用两种方法来处理。领域层能够负责访问数据库，数据库处理它本身的连续性。这对于面向对象系统或者简单关系系统来说工作得很好。当存在复杂的数据格式或者多个数据源时，可能需要一个额外的数据接口层。

本章是基于不同的实践，尤其是英国国家健康服务部门的Cosmos项目和伦敦一家银行的派生交易系统。

## 12.1 两层构架

大多数的IS开发都是（至少是粗略地）依据两层原则进行组织，如图12-1所示。一个两层构架把系统分成一个共享数据库和几个应用。共享数据库位于服务器上，这个服务器具有硬盘空间和应付繁重命令所需要的处理。数据库包含的数据是企业的一个重要部分所需要的，并且这些数据的组织结构是为了支持这个部分的所有需求。（一个单独的企业范围的数据库对大型公司来说是不可行的，因此一个数据库只能负责一个部分。）数据库由一个数据库组进行设计和维护。虽然这里使用术语数据库，但是应当记住的是：数据经常被存储在平面文件中（最商业化的数据仍然位于平面文件（例如VSAM）上）。像这样的数据库可以指任何数据源。

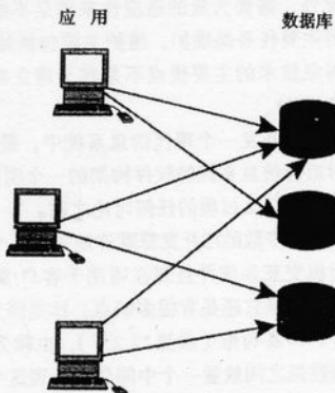


图12-1 两层构架

应用直接访问数据库。

应用是为专门的本地使用开发的。传统上使用CICS/COBOL，但是最

近的工作已经使用4GL和流行的应用开发工具Powerbuilder和Visual Basic。这些工具为开发GUI系统提供完善的特征，并且一个好的Windows界面也是PC用户通常所要求的，他们已经习惯于电子表格和文字处理器上的这种功能。应用通常是针对逐个的用况逐渐建立起来的。任何需要的新数据能力都由数据库组进行请求。

[240]

两层构架具有一些优点。大部分组织具有的数据需要集中控制和一致维护。解释这些数据的应用不需要太多的集中控制。很多IS需要把一些存在的数据以一种新的且有意义的方式显示。

两层构架也存在许多缺点，其中大多数都是现存的技术所固有的。所有数据是共享的而所有处理是本地的，这种思想大致上是正确的但却是一种粗糙的简化。一个企业的很多处理方面的问题是共享的。数据库（不管是SQL还是以前的那些数据库）都不能提供一种计算上完整的语言。数据仍是未封装的，因此大量完整性控制需要应用程序员来做。这使我们难以改变一个已经有大量应用在其上运行的数据库结构。存储过程减少了这些问题，它们能够为处理和封装数据提供支持。

数据库通常不能提供对企业的一个真实描述。这是由于缺乏建模结构，这些建模结构在建模技术中很普通，但是要在日常数据库中支持它们还有很长一段路。平面文件和层次数据库在数据结构上具有众所周知的限制。新的开发工作的现存标准、关系数据库还承受着高额的连接费用。适用于隐含业务语义的数据模型通常被高度地标准化并且为了达到合理的性能需要重新组织。

[241]

一个应用的数据不太可能都在一个数据库中。数据库即使在创建时进行了巧妙的组织，经过一些年的业务改变和企业重组之后通常也不连贯了。两层构架要求应用知道哪个数据库保存哪些数据，以及每个数据库中的数据结构，这个数据结构可能与这些数据的语义有很大的不同。

## 12.2 三层构架

一个更好的构架实际上已经出现了很长一段时间。早在20世纪70年代三模式构架就被提出[4]。它提供一种三层方法，如图12-2所示：外部模式、概念模式、存储（内部）模式。存储模式是数据库设计，外部模式是应用；新的层次是概念模式，我喜欢称其为领域层。这描述了企业的真实语义。它应当忽略数据存储结构和数据位置的限制。

[242]

三层方法的主要优点是它允许完全在领域的语义上描述应用。数据管理员不必关心数据的物理位置和结构，而是能够看到一个消除了这些依赖性的逻辑图像。这也使数据管理员能够自由地改变物理的结构和位置而不

会破坏现存的应用。

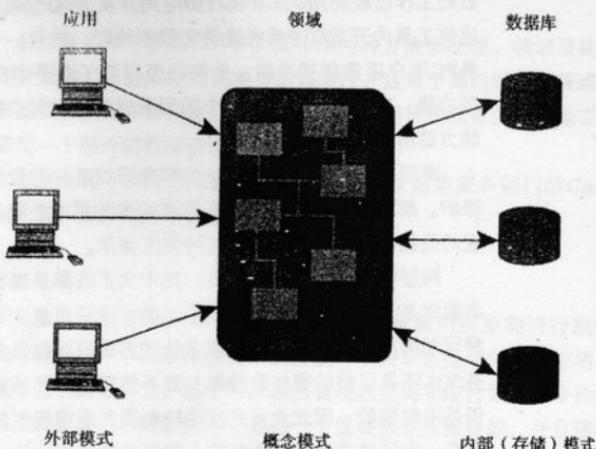


图12-2 三层构架

三层构架得到广泛的赞同但是却很少被实现。这种情况的主要原因是很难用现有的技术来应用它。现在有适合于数据存储和应用开发的工具但是没有实现一个领域层的工具。最有用的开发是逻辑数据模型，它通常被看作数据库设计中一个必需的开始步骤。这允许设计者在提交一个物理设计之前考虑企业语义。这样一来，对物理设计的修改就可以作为企业语义修改的自然映射。

对数据的强调是很重要的。大部分专业人员认为领域层是一个逻辑数据模型。他们可能进行过程建模，但是应用开发者通常单独考虑它。然而，这种观点不是所有数据建模者所共有的。一个语义数据建模者的强大学派认为数据建模完全可以与面向对象建模相比，因为它包含子类型化和派生数据、把过程连接到对象、把过程看作数据并且把过程嵌入到语义模型中。

随着面向对象技术的发展，领域层开始能够走到前面来。对象描述一个很好的方法来实现领域层。它们支持封装、复杂的结构关系、规则、过程以及由高级语义建模者考虑的所有东西。可重用的类库（或者仍然很好的框架）也处于领域层的核心。一个企业的关键重用对象是那些描述领域的对象——那些实现领域层的框架（因此用术语领域框架）。因而，对象建模和领域层开发非常有效地一致。

实现问题多少复杂一些，但是基本的原则仍然工作得很好：如果领域

层被表达成一个面向对象模型并且被作为一个领域框架来实现，那么就能够针对这个领域框架来编写应用。这提供了我们非常需要的应用和数据库之间的分离。

### 领域层的位置

在一个客户/服务器世界中，一个重要问题是这个领域层应当位于什么位置。一个两层方法把应用软件放在客户端（台式机器）而把数据放在不同的数据服务器上。若存在领域层，我们就有两个基本的选择：我们能够把领域层放在客户端，或者我们能够引入一个新的处理器层次，它作为领域服务器并且由一个或者多个联网的机器组成。[243]

基于客户的领域框架使我们把开发集中于客户机器，简化了我们的系统支持。引入机器的一个新层次很可能成为许多用户的一个新的头痛问题，而且它还带来新的需要维护的机器和系统。领域层被作为一个库的集合提供给客户系统的应用开发者，然后由他们编写必需的代码。

一个基于客户的领域层的问题是：我们可能需要在客户端进行大量的数据选择和处理。这强迫我们使用强大的客户机器。随着台式机器变得越来越强大，这不再成为一个大问题，但是我们不能把这种强大作为前提。技术把我们推向比以前更加小型的机器；一些用户想要使用掌上电脑和PDA，这会对处理造成限制。当需要更多的处理能力时，更新服务器通常相对简单。

可用的软件非常适合于基于客户的方法。作为通常的对IS应用最有用的语言，Smalltalk需要一个连接到领域层内部的用户界面，虽然运行在服务器上的没有用户界面的“无头”Smalltalk已经开始出现。

在一个基于服务器的领域层中，领域层更加容易控制和更新。如果领域层位于客户端，那么任何修改都需要被发送到每个客户。一个服务器上的软件更新能够以一种更加简单明了的方式进行处理。这种控制也扩展到对固定数据（尤其是那些包含数据如何被访问的数据项）的支持。

我们需要考虑并发问题。有意思的是，IS应用很可能比任何其它风格的软件都使用更多的并发但是却对此考虑得最少。这是因为强大的事务模型通常由一个数据库很好地操纵，它将应用程序员从大部分令人头痛的并发问题中解放出来。随着领域层被引入，我们不得不问自己事务的边界在哪里。我们可以把它放在数据服务器中或者放在领域层本身中。逻辑的位置是领域层，但是这需要我们在领域层提供事务控制特征——一个错综复杂的业务。这样的放置也鼓励基于服务器的领域层，因为一个跨越大量客户的提交实际上超越了现有的技术。我从来不鼓励客户建立他们自己的事

务控制系统；这个任务处在大多数IS开发的范围之外。

OO数据库为这个问题提供了一个解决方法。IS社团对OO数据库的主要顾虑是这需要把企业数据托付给一种新技术。OO数据库对此做出的响应是已经提供了到传统数据库产品的接口。用这种方法，一个OO数据库能够充当事务控制机制而不需自己存储任何数据。随着时间的过去，一些数据，尤其是（一个OO数据库能够很好管理的）复杂并相互关联的数据，就能够被转移到OO数据库中。然而，只要开发者喜欢，关键的企业数据总是能够被放在一个更传统的地方。这里有一个重要的警告是：很少有关于OO数据库的多用户性能的信息。对OO数据库而言，很多被引用的引人注目的性能改进都是基于小型的单用户的数据库。任何使用OO数据库的人，即使只是为了事务控制，也应当在决定使用OO数据库之前进行基准测试。

如果只使用一个单独的OO数据库，那么数据存储层就被有效地压缩到领域层中。这是可容许的，只要这是一种有效的构架并且为了支持其它数据库而对系统进行的扩展用以下的方式提供，即那些其它的数据库被隐含在领域层之后以便使它们从应用是不可见的。

### 12.3 表示层和应用逻辑层

三层构架提供一些非常重要的好处。目前，太多的注意力被放在如何能够构造领域层上，而且OO建模的很大一个部分被直接应用到这个关键的层次上。然而，很少谈到关于应用的内容。应用是通过在领域层内装配可重用的构件来建立的，并且还存在针对这个任务的指导方针，虽然通常没有对它们的细节描述。

在今天的环境中，典型的情况是一个程序员在一个（建立在领域层上的）GUI环境中开发一个应用。这就需要关于GUI环境和领域层的知识，而一个复杂的领域层能够使学习的曲线非常陡峭。在很多图形环境（例如Visual C++）中，编程也是使人畏惧的。

考虑关于金融机构的一个相对简单的例子，这个机构有一份合同夹包含多个美元（USD）和日元（JPY）之间的派生合同。这样的一个组织关心的是管理与这份合同夹有关的风险。有几个因素能够影响这个风险，包括即期汇率、汇率的易变性和这两种货币的利率。为了考虑这个风险，分析员想要看到在这几种不同因素的各种组合的情况下合同夹的价钱。实现的一个方法是使用图12-3中显示的表格。分析员挑选两个要分析的变量，为它们设置不同的值，然后观察矩阵中显示的合同夹在数值的不同组合情况下的价值。

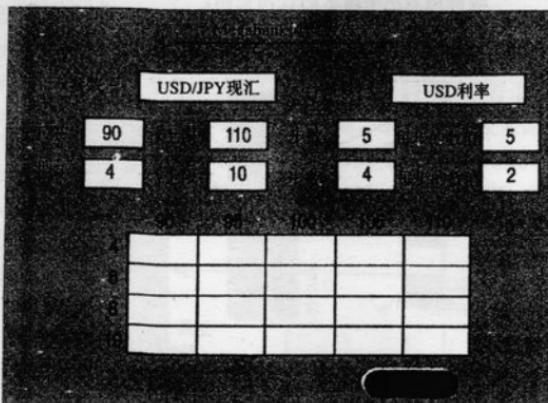


图12-3 一个管理派生风险的应用示例

处理任务是什么，我们应当如何在应用和领域层之间划分它们？一个基本的任务是确定派生合同的价值，典型的情况下这是由Black-Scholes分析[2]处理的一个复杂过程。这个过程会被一个派生交易环境中的任何系统广泛使用，因此它应当被放置在领域层。另一个通用的任务是对一个合同夹中的大量合同进行评价，它通常被放置在领域层。下一个任务是从表格中的参数（上限，下限，步距，步数）建立整个表格中的数值。这个任务是这个风险报告屏幕所特有的，因此在逻辑上应当是应用层的一部分，与建立和控制GUI的代码一起。

245

建立矩阵的任务也应当包括在内并且该任务需要更详细的观察。它包括设置不同的参数，使它们保持一致，然后用参数建立数值的表格。这个过程能够并且应当从一个GUI屏幕的显示中分离出来。因此我建议把应用层分成两个层次：一个表示层和一个应用逻辑层，如图12-4所示。

这两个层次的职责非常容易区分。表示层只负责用户界面。它处理窗口、菜单、字体、颜色和在屏幕或者页面上的所有定位。典型的情况下，它使用一个用户界面框架，例如MFC或者MacApp。它不执行任何计算、查询或者到领域层的更新。实际上它不需要具有到领域层的任何可见性。应用逻辑层不执行任何用户界面处理。它负责所有到领域层的访问和除了用户界面处理外的任何处理。它从隐藏的领域层中选取信息并且把它简化成表示层需要的确切形式。领域层的复杂的相互关系因而对表示层是不可见的。除此之外，应用逻辑层还执行类型转换。表示层一般只处理公共类型的一个小集合（整型，实型，字符串，日期，加上软件中使用的聚合类）。应用逻辑层提供这些类型，而且负责把隐藏的领域类型转换成这些类型并

246

解释由表示层请求的任何更新。

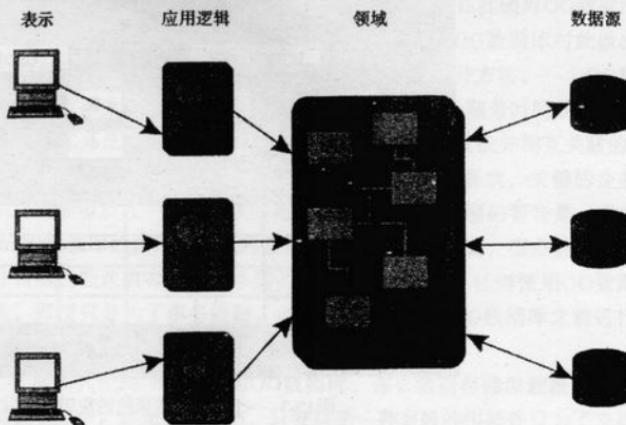


图12-4 把应用层分成表示层和应用逻辑层

组织应用逻辑层的一个有用方法是开发一系列的外观。外观[1]是为复杂模型提供简化接口的一种类型。我们能够为每种表示准备一个外观。在外观中，相应用户界面上的每个元素都有一个特征。因此每种表示都有一个到领域模型的简单接口，这个接口把对表示而不是对用户界面的所有处理进行最小化（第13章讨论设计这些外观的一种技术）。

图12-5显示这种组织是如何处理上面提到的风险报告屏幕。我们需要两个类：一个风险报告表示类和一个风险报告外观类。表示类创建屏幕的布局并且管理用户和它的交互。外观类提供一个模拟表示类的隐藏结构。它有操作来得到和设置参数、上限、下限、步数、表格的x和y坐标的步距。外观类还包含在这些数值之间确保正确的一致性所必需的规则（例如不变式 $xUpper - xLower == xNumberOfSteps * xStepSize$ ）。它也提供一个方法来返回结果表格。理想的情况下，这使用一个通用的矩阵类返回一个单独的矩形。（如果因为某种原因这既不是可行的也不是所期望的，那么外观类提供操作来得到特殊的单元，但是一个可重用的矩阵类（本质上是一种新的聚合）通常是最好的解决方案。）

外观上的getResultSetMatrix方法查看：表示对象是否提供了足够的信息（如果没有，它能够增加默认值）并且接下来请求领域层用参数的不同组合来评价合同夹对象。领域层把结果填入矩阵然后把它返回给表示对象。

设置参数是一个使用类型转换的例子。不同的对象可以作为参数被放

置在这个列表中，包括USD/JPY现汇、USD/JPY的易变性、USD的利率和JPY的利率。（这个列表依赖于合同夹中合同的货币。）外观对象向表示对象提供从领域层（参见13.5节）中的类型转换得到的字符串。外观对象一般提供这种字符串的一个列表给表示对象，后者把它放置在弹出式菜单中。表示对象因此可以选择一个字符串。外观对象把选中的字符串关联到下层的领域对象（一个字典能够很好地对此进行处理）。这样一来，用户界面就完全和领域模型隔离开来。

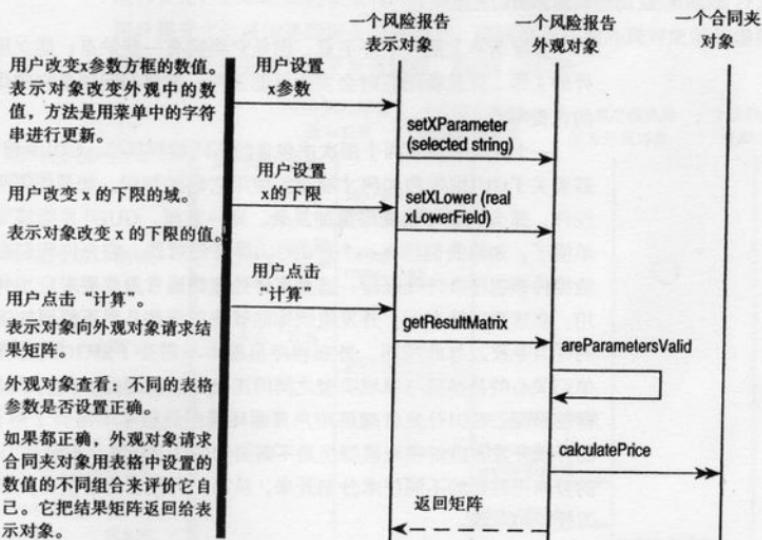


图12-5 总结表示、外观和领域层之间协作的交互图

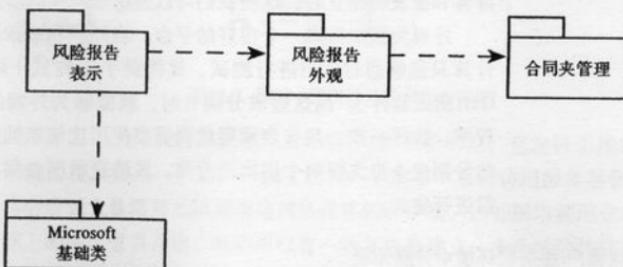


图12-6 表示、外观和领域分类之间的可见性

在这种情况下，两个领域之间可见性的定义如图12-6所示。可见性只能从表示层流动到应用逻辑层再到领域层。这条可见性的路线是很有价值的，因为它把领域层和依赖于它的应用完全隔离开来。然而，如果当领域模型中发生一个变化时表示层需要被自动地更新，就会发生问题。一个选择是让表示层定期进行更新，但是这可能会变得非常凌乱。一个更好的选择是使用观察者模式[1]。这就使外观对象和表示对象在不破坏可见性规则的情况下被自动地更新。

### 12.3.1 表示层/应用逻辑层分离的优点

分层基本上是一个好主意，但是它确实有一些缺点：建立层次需要额外的工作，并且使用它时会发生性能损失。重要的问题是它的优点值得它的花费吗？

一个优点来自于两个层次中包含的不同编程风格。GUI编程非常复杂，需要关于GUI框架和如何才能很好使用它们的知识。如果还需要新的GUI控件，那么编程可能变得更加复杂。另一方面，GUI开发能够变得非常简单明了，如果我们拥有一个好的GUI屏幕创建器，它允许我们在屏幕上描绘控件和制作事件处理器，这些事件处理器通常最终要对应用外观进行调用。在这两种情况下，开发组织都能够使用那些几乎不需要知道领域模型的GUI专家。与此相同，外观程序员根本不需要了解GUI系统如何工作，他们关心的是得到与领域类型之间的正确交互。因此，我们看到会存在这样的情况：GUI开发者理解用户界面环境但是根本不需要了解领域模型，而外观开发者理解领域模型但是不需要知道GUI开发。表示层/应用逻辑层的分离把需要的不同技术分割开来，从而使开发者为了做出贡献只需要学习较少的东西。

这个分离允许从一个单独的外观开发出多种表示；当我们需要包含相同信息的由用户定制的屏幕和页面布局时，这尤其有用。当工具用来进行屏幕和报表的建立时，这使我们可以迅速地转变到新的表示风格。

外观为测试提供一个很好的平台。当外观和表示层被合并时，基础的计算只能够通过GUI进行测试，这需要手工测试（或者针对回归测试的GUI测试软件）。当这些被分隔开时，就能够为外观的接口编写一个测试程序。这样一来，只有表示层代码需要使用比较笨拙的工具来测试。测试的分割进一步支持两个层次的分离，虽然表示层必须在外观能够被建立之前进行定义。

### 12.3.2 在客户/服务器环境中伸展外观

如果领域层是基于服务器的，那么外观对客户/服务器交互的价值相

当于一个焦点。在这种情况下，一种有用的技术是把外观“伸展”到客户和服务端，即把一个外观类既放在客户端上又放在服务器端上。当用户打开一个表示对象时就在客户端打开相应的外观。客户外观把请求传递到服务器外观上。服务器外观检查创建过程，从领域类中找出信息。当客户外观需要的所有信息都完成时，服务器外观把客户外观需要的所有信息都发送到客户端。由于服务器外观和客户外观可以是不同的对象空间，因此在两个外观类之间就会发生一系列的私有通信。因而，用户能够和表示对象进行交互，表示对象将根据每一个修改来更新客户外观。这些修改一直到用户提交才会被传递到服务器外观上。那时修改过的外观对象被传递回服务器，然后服务器外观更新领域层，如图12-7所示。

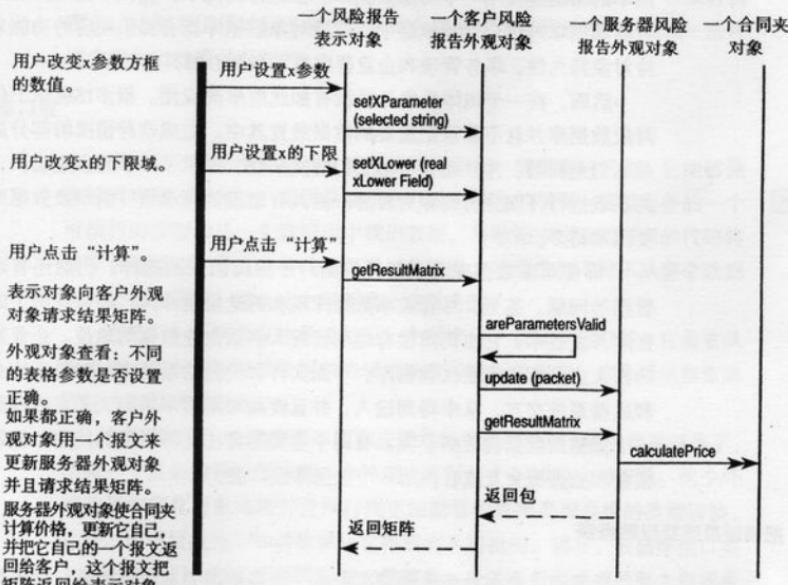


图12-7 图12-3使用伸展外观的交互图

伸展一个外观的要点是：对于客户/服务器交互，它允许引用的一个单独的点。如果一个客户外观（或者一个表示）直接访问服务器领域类，将会看到大量需要的调用通过网络转移到客户端。这些网络调用会成为性能上的一个重要开销。外观可以有一些方法来建立一个单独的转移报文并且把这样的一个报文解释成外观的数据。然后就能够在一个单独的网络调用中传递所有的信息。

外观的不同职责能够在客户类和服务器类之间进行分割。只有服务器外观需要具有与领域模型交互的职责。两种类都需要能够向其它类发送信息和从其它类接受信息。理想上，只有客户外观需要支持表示层的操作。然而在实践中，我发现：为了使测试更加简单，给这两个类提供相同的接口是值得的（也就是说，它们是相同的类型）。两个方面都需要加载和保存操作。客户外观通过与服务器外观的通信来实现这些操作，而服务器则通过与领域模型的通信来实现它们。

## 2.4 数据库交互

我们需要仔细地考虑如何把数据库和遗留应用结合到这个结构中。最简单的情况是使用一个对象数据库。在这种情况下，直截了当的方法是把数据库简单地结合到领域层中。然后对象数据库就会提供相应的功能来支持对象持久性、事务管理和企业程序员不必担心的其它特征。

然而，在一个IS体系中几乎没有如此简单的应用。很多IS组织不信任对象数据库并且不愿意把重要的数据放置其中。造成这种情况的部分原因是它们是新的，并且还要考虑它们的复杂性。如果某个事物发生错误，对关系表进行仔细分析要相对容易。而具有繁茂的硬盘指针的对象数据库就要困难得多。

即使对象数据库对新的开发是一个值得信任的选择，仍然还有现存数据的问题。甚至，尽管关系数据库现在的地位是作为已经被证明了的数据库开发技术，它也仍然没有达到管理大多数企业数据的地位。企业数据的绝大多数都位于层次数据库、平面文件和类似的系统中。对象系统必须和这些系统交互，从中得到输入，并且要面对不得不访问大量系统以获得一个完整的数据视图的事实。有两个主要的方法能够供我们使用：让领域模型和数据源交互或者使用一个数据库接口层。

### 2.4.1 把领域层连接到数据源

让我们考虑一种简单的情况：一个需要使用关系数据库进行数据存储的独立系统。我们应当为了专门支持这个领域模型而设计关系数据库。我们应当首先设计领域层并且把数据库模式建立在这个基础之上。除了最简单的系统以外，不可能简单地获得领域模型中的每个对象类型并且把它变成一个关系表。不管它们的名称，关系数据库还有一个关联数据的问题，因为计算连接是耗费时间的。因此，一个好的关系化的设计应当非常标准化，以便获得好的性能。领域模型为数据库设计提供一个起点，但是数据库设计要做得好需要时间。最后得出的数据库模式可能看起来与原始的对

象图区别很大。

把领域层连接到数据库的显而易见的方法是使领域类知道如何从数据库建立它们自己。类能够具有从数据库中获得数据的装载程序并使用它来创建和组装框架。重要的是：应用不被包含在这个行为中。当一个应用请求一个对象时，领域层应当查看它是否在内存中。如果没有，它应当使该对象脱离数据库创建它自己。应用应该不需要知道这个交互是如何发生的。

这个过程会发生一个例外，就是应用需要一个特殊的数据配置发生作用，并且在开始时只用一个步骤就能够从数据库中获得该数据，以便改善性能。在这种情况下，一个有用的做法是领域层提供针对具体应用的装载请求，这种请求给应用一个机会让领域层知道它想要请求什么。在某种程度上，这损害了领域层不应当知道什么应用使用它的原则，但是在一些环境中性能的改善是强制的。

## 4.2 数据库接口层

领域层和数据库之间的直接连接确实存在一些重要的问题。它能够使领域类变得过分复杂，原因是赋予它们两个独立的职责：提供业务的一个可执行的模型和从一个数据库中找出数据。与数据库交互所需要的代码将是非常重要的，并且这些代码使类过分膨胀。如果数据不得不从多个数据库和供给中获得，那么这个问题就会变得关键。

[252]

当然，一个解决方法是增加另一个层次——数据库接口层，它负责从数据库中为领域层装载数据和当领域改变时更新数据库。这个层也负责处理输入以及其它遗留交互。

在许多方面，数据库接口层和应用逻辑层非常相似。在这两种情况下，都向一个复杂的领域层提供一个外观来应付一个不够强大的表示。这个外观选择和简化对象结构并且执行到更加简单的外部类型系统的类型转换。此外，领域层应当不知道能够从它得到的不同视图。通常，数据库接口类是基于和它们一起工作的数据源。能够为一个关系数据库中的每个表或者一个供给中的每个记录类型构造一个数据库接口类。支持数据库交互的类库通常都支持这种通信。

这一层和应用逻辑层之间最大的区别在于活动的启动。通过用户界面，用户的动作导致表示层对活动进行初始化。由于表示层具有到应用逻辑层的可见性，因此它调用应用逻辑层是很简单明了的。活动的初始化遵循可见性的线路。然而这不是数据库接口的情况。领域层通过想要保存它自己来启动过程，但是我们不想让领域模型看到数据库。因此活动的初始化与

期望的可见性是对立的。一个解决方法是再次使用观察者模式[1]，但是那很可能会导致很大的消息通信量。

一种替代方案是用一个对领域层可见的接口代理来扩展构架。这个代理提供一个非常小的接口，它只允许初始化数据库接口的消息。通常，这些可能是像loadMe(anObject)和saveMe(anObject)一样通用的调用，它们所有的职责就是处理到数据库接口层的请求。因此接口代理的职责是把这个请求传递到数据库接口中能够最好地处理这个请求的一个类上。所以，如果我们拥有保存在一个数据库表中的现汇合同和保存在另一个表中的常规期权，那么接口代理首先询问这个对象以便找出它是哪一个，然后把请求传递到适当的数据库接口类上，如图12-8和图12-9所示。

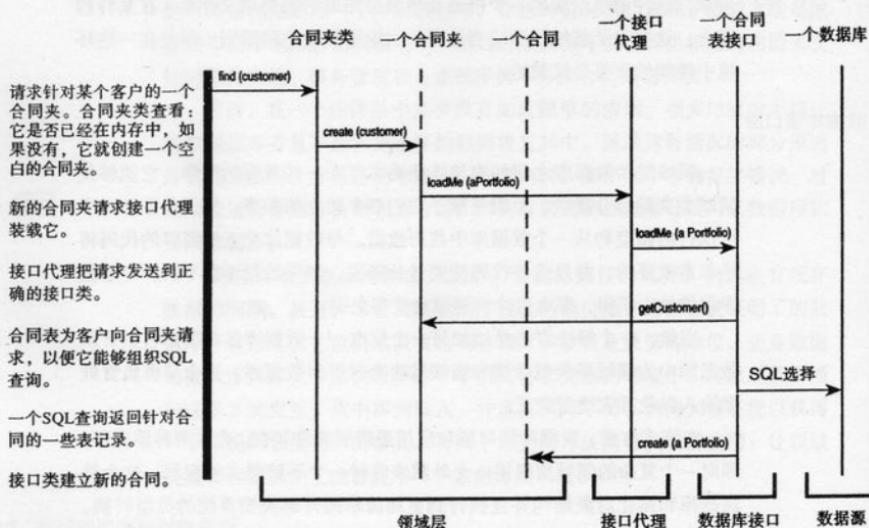


图12-8 解释一个典型领域层和一个数据源之间交互的交互图

这个分层的优点和在其它地方分层的优点是一样的。职责再次被以一种有用的方式进行分割，数据接口从企业模型中被分离出来。表格式或输入的改变不用改变领域模型就能够进行。当表格式不在项目小组的控制之中时或者当数据结构很可能为了促进性能而改变时，这是尤其重要的。这些数据来源的易变性越大，使用一个中间层就越重要。

访问不同数据库需要不同的工具和技术。为了连接到数据库产品的接

口，有专门的类库。还可能需要关于SQL和特殊数据库格式的知识。其它的数据库（多维的，层次的）有它们自己的接口和结构要学习。把这种交互分离出来就能够允许小组成员集中精力于他们技术最强的领域，尤其当存在很多不同的数据源时，这是非常有效的。

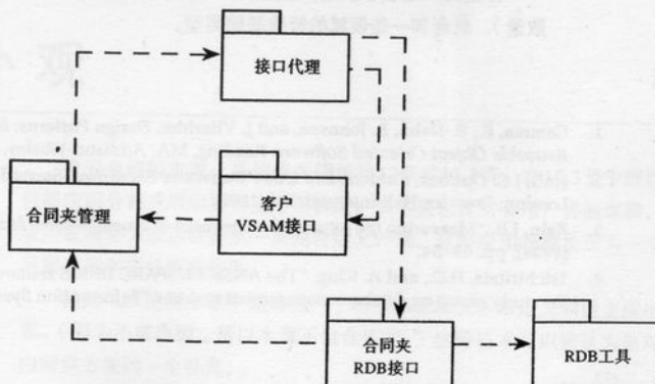


图12-9 数据库接口层的分类

## 结论

在一个客户/服务器环境中建立大型IS系统仍然是一个有着很多缺陷的困难的活动。这些缺陷的大部分都存在于一个两层构架中，两层构架对小型系统工作得很好但是不能很好地扩展到大型系统。一个三层构架大大改善了这些问题并且被对象技术很好地支持。表12-1提供对三层构架的主要描述。

表12-1 层次和其目标的总结

层 次	描 述
领域	关于业务对象的适用于整个领域的一个直接模型。独立于单个的应用和数据源
应用逻辑	针对一个应用的，对一个领域模型的选择和简化。不包含用户界面代码，但是为用户界面提供一系列领域层的外观。把大量领域层类型转换成一个表示层所需要的类型
表示	执行从应用外观到一个GUI或者页面报表的信息格式化。只关心用户界面，并且没有关于隐藏的领域层的知识
数据接口	负责在数据源和领域层之间移动信息。将为领域层提供一个简单的接口代理以便发布请求。具有对领域层和数据源的可见性。将依据所用数据源的类型，把数据分配到子系统中

分割应用层以便把应用逻辑从用户界面中分离出来是一种有价值的技术。它的优点包括：为不同的GUI重用应用逻辑、易于测试、为客户/服务器进行性能管理和支持更加专业的开发人员。一个中间层对数据访问也是有用的，尤其是当存在很多复杂的数据源时。

有些类一定会被所有的层使用。这包括公共基础类型（整型，日期，数量）、聚合和一些领域的特殊基础类型。

## 参考文献

1. Gamma, E., R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.
2. Hull, J.C. *Options, Futures, and Other Derivative Securities* (Second Edition). London: Prentice-Hall International, 1993.
3. Kain, J.B. "Measuring the return on investment of reuse." *Object Magazine*, 4, 3 (1994), pp. 49-54.
4. Tsichritzis, D.C., and A. Klug. "The ANSI/X3/SPARC DBMS framework: report of the study group on database management systems." *Information Systems*, 3 (1978).