



## 第IV部分

# 战略性设计

当系统变得太复杂以至于我们不能完全从对象级中理解系统时，就需要借助一些技术来操作并掌握这些大模型。本部分将介绍一些原则，将这种建模方法扩展到很复杂的领域中。大部分像这样的决策必须在团队级别甚至需要在团队之间协商后才能做出。而这些决策往往是在设计和策略的共同作用下产生的。

最完备的企业应用系统的目标是一个高度集成的系统，它包括了所有业务。然而任何一个把所有业务组织在一起的模型都太庞大、也太复杂，所以很难管理，甚至不能把它作为一个单独的单元来理解。因此我们必须从概念和实现两个方面，把整个系统分成较小的部分。面临的挑战是如何实现模块化而又不失集成的优势，允许系统不同部分之间互操作从而协调各种业务运作。领域模型设计成一个包括所有业务的单一模块是不实用的，而且在使用时会有一些重复和冲突。使用特别接口把一系列小而清晰的子系统结合在一起，这种系统将缺乏解决企业级问题的能力，还会在每一个集成点上引起一致性问题。用一个系统的、发展的设计策略能够避免以上两种极端情况下出现的问题。

即使在这样大规模的系统中，领域驱动设计也不能脱离实现去开发模型。每个决策要么必须直接对系统的开发产生影响，要么与开发无关。策略设计原则必须指导设计决策，减少系统各部分之间的相互依赖性，在无损关键的互用性和协同性条件下使系统结构更加清晰。这些原则必须要做到在不影响项目进展的情况下，创建的模型要抓住系统的概念核心，以及系统的“愿景”。为了帮助我们实现这些目标，第IV部分将论述3大主题：上下文、精练以及大比例结构。

上下文，在这3个原则中最不明显的，也是最基本的原则。一个成功的模型，不管大小，都必须自始自终在逻辑上保持一致，没有矛盾和重复的定义。企业应用系统有时候会集成不同来源的子系统，或者有截然不同的应用，以至于领域中的每一部分看起



来都不同。要求统一这些不同部分的模型可能会不太现实。通过明确地定义适用于一个模型的限界上下文，并且在必要时定义它与其他上下文环境的关系，这样建模者就能避免误用这个模型。

精练能够减少系统的混乱并且适当地集中我们开发的注意力。我们常常会耗费许多的精力来解决领域中细枝末节的问题。整个领域模型需要突出系统中最有价值和最特殊的方面，并且在构造模型时要尽可能地加强这部分的性能。当一些支持的组件非常关键时，它们必须要引起足够的重视。这种重视不仅能够帮助指导我们把精力放在处理系统的重要部分上，而且还能保持系统的完整愿景。策略精练能使庞大模型的结构更加清晰。因此，我们可以清楚地看到，核心领域(Core Domain)的设计可以得到更加广泛的应用。

大比例结构能够完成对整个系统的描述。在一个很复杂的模型中，可能很难看清它的全貌。通过把注意力集中在系统的核心部分并且以辅助作用的形式来表现其他元素，精练能够帮助我们获取系统的概貌，但是如果每一个主线，不应用一些系统级的设计元素和模式的话，系统内的关系仍然会显得很混乱。我将概述几种有关大比例结构的方法，然后深入介绍职责层(Responsibility Layer)结构来探讨使用这种大比例结构的含意。本书讨论的具体结构仅仅是一些示例，而且还有很多种结构在本书中没有介绍。经过一个渐进顺序(Evolving Order)的开发过程后，可以根据需要开发新结构，或者修改已有结构。一些像这样的结构能够统一设计，加速开发进程和改善集成效果。

这3个原则，每一个都可以单独使用，但是如果把它们结合起来使用，效果就更加显著了。甚至在一个没有人能完全弄懂的、毫无章法的系统里，这些原则也能帮助我们开发出好的设计。大比例结构给不同的部分带来了一致性，并帮助我们把这些部分联系起来。结构和精练能使各个部分之间的复杂关系易于理解，同时保持对系统全局的认知。限界上下文允许工作能在不同的部分继续进行，而不会破坏模型或者无意识中造成模型的碎片。把这些概念添加到团队使用的通用语言中，可以帮助开发人员设计出自己的解决方案。

## 第14章

# 维护模型完整性

我曾经参加过一个项目，由几个团队并行开发一个较大的新系统。一天，开发顾客发票模块的团队准备实现一个叫作 Charge(收费)的对象时，他们发现另外一个团队已经建立了一个这样的对象，于是打算重用这个现成的对象。他们发现这个对象缺少“expense code(费用代码)”属性，就加了一个进去。里面已有一个他们需要的“posted amount(过账金额)”属性，而他们原本打算把这个属性叫作“amount due(到期金额)”，但是他们想名称不同又有什么关系呢？于是就把原来想好的名称改成了 posted amount，并且添加了几种方法和关联。看起来，他们得到了想要的东西，根本没有想到这里面会有什么麻烦。除了不得不忽略许多他们不需要的关联外，他们的应用模块照样能够正常运行。

几天以后，在票据支付应用模块中，原先写好的 Charge 上发生了一些奇怪的问题，出现了一些奇怪的 Charge，没有人记得曾经输入过它们，而且这些记录毫无意义。一旦运行某些函数，尤其是周月的税务报表，程序就会崩溃。调查的结果表明，当调用一个用来合计当月付款的抵减额的函数时会导致崩溃。尽管数据项应用的验证需要用到该函数并会赋予相应的默认值，但是那些奇怪的记录在“percent deductible(抵减率)”字段中并没有赋值。

问题出在这两个团队都没有意识到他们有不同的模型，而且没有任何相应的过程来检查它们之间的差异。每个团队都对 Charge 的性质作了假设，而这个性质在他们的上下文环境(向顾客收费 vs. 向供货商付款)中是有用的。因此，他们在还没有解决这些矛盾时就把他们的代码结合在一起，当然会产生不可靠的软件。

只要他们意识到了这种情况，就会有意识地决定如何去处理。这样就可能意味着他们要么合力打造一个共同的模型，并编写一个自动化测试集来预防今后的意外；要么简单一点，独立开发各自的模型，并保证他们的代码不会互相产生影响。这两种方法，都需要从双方确定一个清楚的分界线，划定每个模型应用的范围开始进行。



一旦认识到问题所在，他们会怎么做呢？他们分别创建 Customer Charge(顾客收费)和 Supplier Charge(供货商收费)两个类，并且根据各自团队的需要来定义每一个类。这个临时问题解决后，他们又重新回到原来的方式下继续开发。

尽管我们很少会去明确地考虑这个问题，但是一个模型最基本的要求是内部一致性，即它的术语总是表达一样的意思，而且不包含任何互相矛盾的规则。模型内部的一致性，也就是说术语无歧义而且没有冲突的规则，我们谓之统一(unification)。除非一个模型在逻辑上一致，否则它没有任何意义。在一个理想世界里，我们会拥有涵盖全部企业领域的单个模型。该模型是统一的，术语没有任何冲突或者重叠定义，领域的每一个逻辑声明都是一致的。

然而大型系统的开发并没有这么理想。在整个企业系统中维持这种水平的统一，带来的麻烦比收获的好处要多。虽然允许系统的不同部分采用不同的模型进行开发是必要的，但是需要仔细地选择，确定系统的哪些部分可以分开以及它们之间有什么样的关系。我们需要一些维护模型关键部分高度统一的方法。统一不会自己产生或凭着良好愿望就能产生，它只能通过有意识的设计决策和制定特定过程来实现。实现大型系统领域模型的完全统一是不可行的或者代价太高。

有时人们会力图来解决这个问题。大多数人都看到，使用多个模型增加了统一的代价，因为它们使得(模型之间的)集成受到限制，而且妨碍了(开发人员之间)沟通。最不好的是，多于一个的模型看起来有些不够优雅。出于对多个模型的反感，有时候人们会野心勃勃，想把一个大项目中的所有软件全部统一到一个模型中来。我自己就曾经因此而弄巧成拙。但是请考虑下面的风险。

- 企图一次性进行太多的遗留替换老式系统。
- 由于需要协调大量的问题，这可能会使大型项目不堪重负，从而陷入困境。
- 有特殊需求的应用可能会发现模型不能完全满足它们的需要，从而迫使它们把这些行为放到其他地方去。
- 相反，如果想用一个模型来满足所有人的要求，可能会导致复杂的选择，使模型难以使用。

此外，除了技术上的考虑，制度和管理优先权上的差异也会导致模型的分化。团队组织和开发过程可能也会导致不同模型的出现，所以即使没有技术因素阻碍完全集成，项目可能还是会面对多个模型。

由于为整个企业维护一个统一的模型根本行不通，所以我们就不必再受这种想法的支配。通过一系列积极主动的决策来决定什么是需要统一的、什么是实际上不需要统一的，我们能够对目前的情况有一个清晰的、共同的认识。一旦掌握了这种情况，我们就要开始确保那些需要统一的部分保持一致性，而那些不需要统一的部分不会引起混淆或破坏。

我们需要一种在不同模型之间标记界限和关系的方法。我们必须有意识地选择策略



并且始终如一地按照这个策略进行开发。

本章介绍的技术包括识别、传递、选择一个模型的界限及其与其他模型的关系。所有技术都从确定项目当前的领域开始进行。限界上下文确定每个模型的适用范围，而上下文映射(Context Map)则给出了项目上下文的一个全局概貌以及它们之间的关系。不管在上下文中还是上下文本身，减少歧义都会明确事件在项目上的发生方式，但是光这样做还是不够。一旦建立了限界上下文，还需要使用持续集成来保证模型的统一。

从这种稳定的情形入手，我们就可以开始转向限界上下文更有效的策略，从紧密联合的共享内核(Shared Kernels)模式到松散关联的隔离方式(Separate Ways)模式，并把它们联系起来。

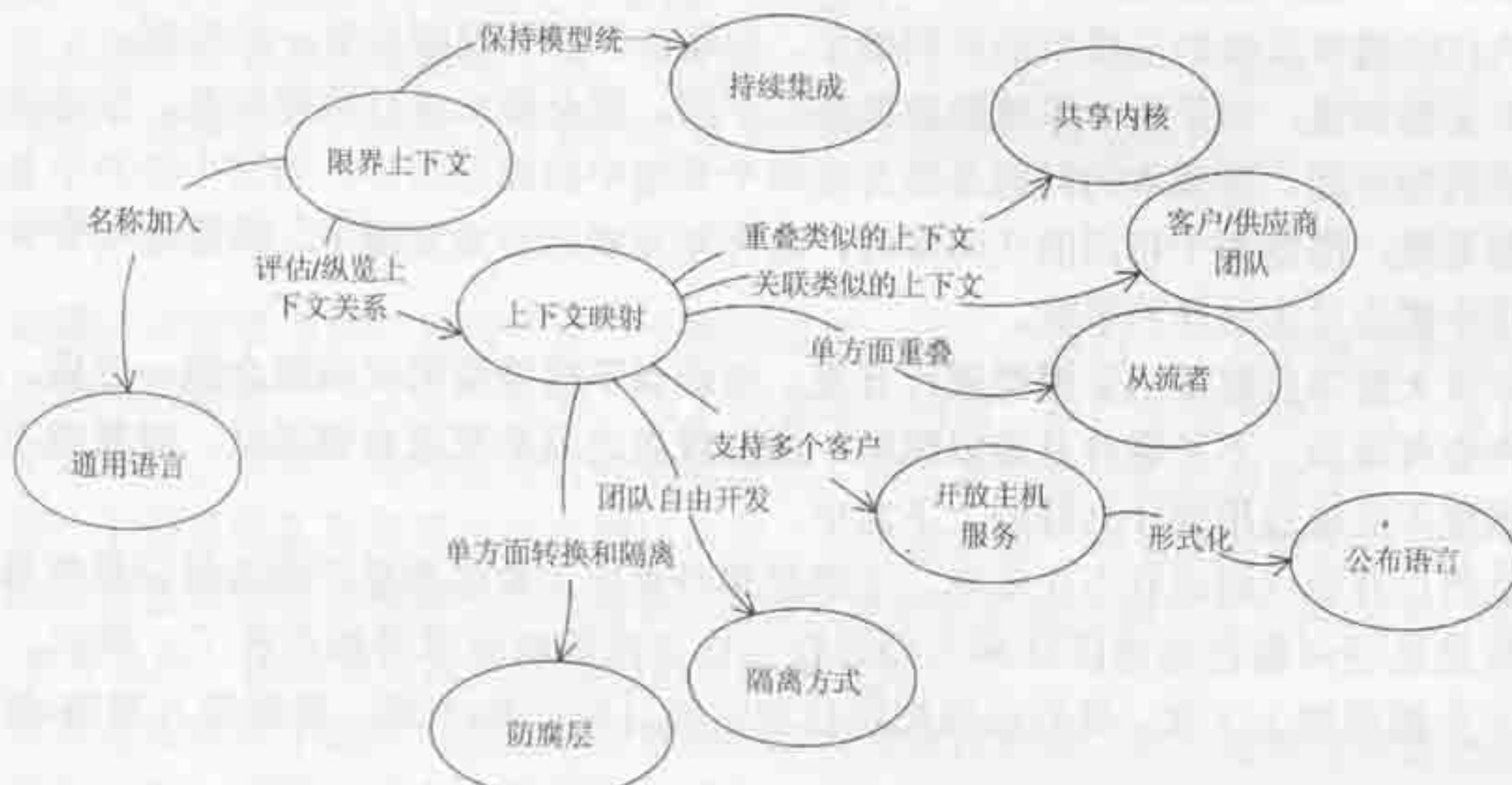
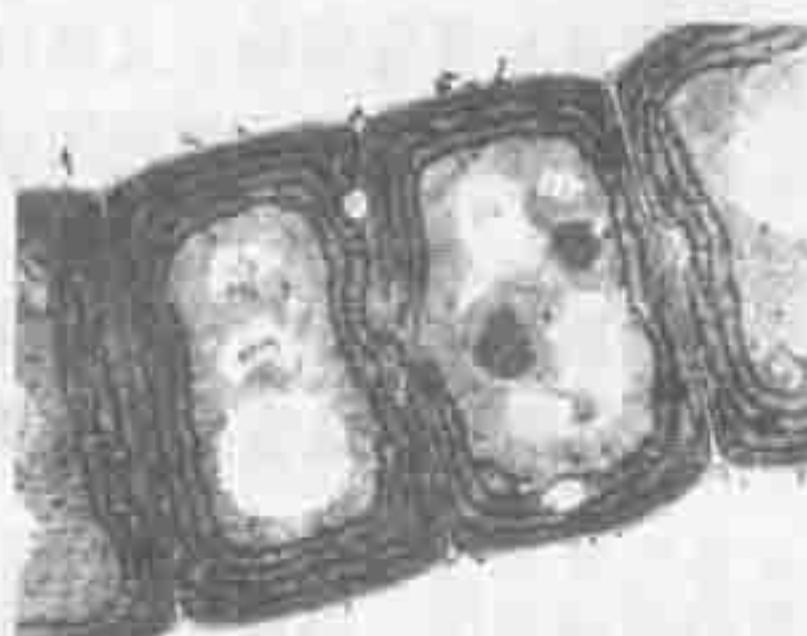


图 14-1 模型完整性模式的导航图

## 14.1 限界上下文



细胞膜不仅能把细胞内部和外部区分开来，而且还能决定通过的物质



有大型项目采用了多模型开发，而且都有不错的表现。不同模型应用于不同上下文中。例如，有时候，可能需要把一个团队无法控制的外部系统与新软件集成。这种情况对于每个人来说都非常清楚，在一个上下文当中开发模型是不合适的，但是在其他情况下，就不是那么容易弄清楚了。本章一开始我们提到了一个故事，两个团队同时开发一个新系统，分别负责不同的功能。他们用同一个模型进行开发的吗？他们打算至少能够共享他们开发的部分内容，但是却没有任何划分的方法来确定哪些是能共享的，哪些是不能共享的。他们也没有任何相应的过程来维护共享模型或者迅速检测出他们之间的不一致。只有等到系统出现了不可预料的情况时，才会意识到他们之间存在不一致性。

甚至一个团队也能使用多模型。由于缺乏交流，会对模型的理解有一些出入。并且先前的代码通常反映的是模型的早期概念，与现在开发的模型会有一些差别。

大家都知道，如果两个系统的数据格式不同，就会要求进行数据转换，但这还只是问题的机制方面。更基本的问题是隐含在两个系统中的模型差异。当这个差异不是来自于外部系统，而是源于相同的代码库时，这种差异就比较难发现了。但是在所有大型团队项目中都会发生这样的事情。

所有大型项目都采用多模型进行开发。然而当不同模型的代码组合到一起后，构成的软件会有缺陷、不可靠并且难以理解。团队成员之间的交流也很混乱，经常搞不清楚一个模型不应该应用到什么样的上下文中。

虽然只有当代码运行不正常时，才能把那些表面上看起来是正确的错误最终暴露出来，但是这些问题在组建团队和人们进行工作交流的时候就开始存在了。所以，为了阐明一个模型的上下文，我们必须把项目和它的目标产品(代码、数据库方案等)都考虑进去。

一个模型应该要与一个上下文相适应。上下文可能是指一段代码，也可能是指特定团队的工作。如果一个模型是在一次头脑风暴会议上诞生的，那么它的上下文就可能会限制在这些讨论的范围中。稍后将在示例部分讨论一个模型的上下文。在有特定意义的模型中，不管模型的上下文是什么，必须要说明模型中的术语是什么意思。

开始解决多模型问题之前，我们首先需要明确定义软件系统中每一个特定模型的范围，并且要尽可能保持模型统一。定义这些范围必须同团队组织相协调。

因此：

明确定义模型应用的上下文。根据团队组织、应用程序各个部分的使用率、物理显现(如代码库和数据库方案)明确设置界限。在这些界限中要保持模型严格的一致性，但不要被外界问题干扰和迷惑。

限界上下文限制了特定模型的适用范围，因而团队成员对需要保持一致的部分以及



如何联系到其他上下文，都要有一个清楚和共同的认识。在上下文内要保持模型的逻辑一致，而不要担心适用于界限外的情况。在其他上下文中的模型使用的术语、概念和规则、通用语言的“方言”都是不一样的。通过画出清楚的界限，能够保持模型在它适用范围内纯粹而有力。同时，也可避免把注意力转移到其他上下文时的混淆。跨越边界的集成必定会引起一些转换，但限界上下文能够帮助您对这些集成进行清楚的分析。

### 限界上下文不是模块

这两个词有时候会被混淆，但它们是有着不同动机的两种模式。实际上，如果两组对象被认为构成不同的模型，那么它们总是要被分到不同的模块中。这种做法提供了不同的命名空间(特别是对不同的上下文)和一些划分。

但是模块也在模型里组织元素；它们不必划分上下文。在一个限界上下文里，由模块创建的分离命名空间实际上让我们更难发现偶然出现的模型碎片。

### 示例：预订的上下文

运输公司需要为预订货物开发一个新的应用。这个应用由一个对象模型来驱动。这个模型适用的限界上下文是什么呢？为了回答这个问题，我们必须考虑在这个项目中会发生什么事情。记住，是考虑项目的实际情况，而不是考虑它的理想情况。

有一个项目团队负责开发这个预订应用。在不修改这个模型的对象的前提下，他们创建的应用必须能显示和操作这些对象。该团队要用到的这个模型在应用(它的主要消费者)中是有效的，因此预订应用在模型的界限范围内。

因为完成的预订必须交给原来的货物跟踪系统进行处理，所以要预先决定把新模型和老模型分离，这样，原来的货物跟踪系统就在新模型的界限之外。新模型和老模型之间的必要转换由老模型的维护团队负责完成。转换机制不是由模型驱动的，而且不属于限界上下文(转换机制是边界本身的一部分，这将在上下文映射中讨论)。转换在上下文之外(不基于模型)，这种处理方式是正确的。要求老团队真的利用这个新开发的模型是不切实际的，因为他们的主要工作超出了上下文的范围。

负责开发模型的团队为每一个对象都制定了它的生命周期，包括持久性。因为这个团队控制数据库方案，所以他们已经仔细考虑过如何简单明了地映射相关对象之间的关系。换句话来说，由于数据库方案是由该模型驱动的，因此它在上下文界限内。

另外一个团队工作的模型及应用是负责安排货船航次。负责调度的团队和负责预订的团队，他们的工作是同时开始的，他们都打算创建一个唯一的统一系统。两个团队偶尔进行一些协调，偶尔也共享一些对象，但是从来没有系统地合作过。他们不是工作在同一个限界上下文中，这样会存在风险，因为他们不认为自己工作在不同的模型中。如



果最后把两个系统集成，如果没有适当的过程来管理这种情况，就会出现问题（在本章后面讨论的共享内核可能是处理这类问题的一种好选择）。无论如何都要首先认清楚实际的开发情况。两个团队不属于相同的上下文，除非作出相应的改变，否则不应共享代码。

系统中所有由这个特定模型驱动的方面组成了限界上下文：模型对象、保存这些对象的数据库方案以及预订应用。两个团队——建模团队和应用团队，主要工作在这个上下文中应用程序开发。他们需要和老的跟踪系统交换信息，原来的开发团队获得建模团队的协助，主要负责边界上的转换。在预订模型和航次调度模型之间没有明确定义的关系，而定义这种关系应该是两个团队首要完成的任务之一。同时，这两个团队还应该非常注意代码和数据的共享。

那么，通过定义限界上下文会得到什么呢？对在上下文中开发的团队来说：清晰。两个团队知道他们必须得与其中的一个模型保持一致。他们要根据这种认识做出设计决策，以防止割裂系统的一致性。对外部团队来说：自由。他们不必行走在灰色地带，而不使用同样的模型，但是不知道为什么，所以他们没有使用相同的模型。从这个特定例子中得到的最大的收获是，意识到在两个团队之间共享信息存在着风险。为了避免这些问题，他们确实需要对共享和使用共享的代价做出权衡，并放到流程之中让它运转起来。除非每个人都知道模型上下文的界限在哪里，才不会出现这种问题。

当然，边界是一些特殊的地方。限界上下文和相邻上下文之间的关系是要密切关注的。上下文映射勾勒出了上下文的疆界，并给出上下文的大图以及它们之间的连接，而几种模式则定义了上下文之间各种关系的性质。一个持续集成(Continuous Integration)的过程保证了模型在限界上下文内的统一。

但在进一步讨论这些技术之前，设想一下当一个模型的统一被破坏时，整个系统看起来会像什么呢？您又是怎么认识概念碎片的呢？

## 识别限界上下文中的碎片

许多症状可以显示出未知的模型差异，当编码的接口不匹配时最为明显。一些细微的、意外发生的行为也许就是一种模型差异的征兆。带有自动化测试的持续集成过程能够帮助我们发现这类问题，早期的警告信号通常是语言上存在混淆。

结合不同模型的元素会产生两类问题：重复概念(duplicate concept)和假同源(false cognate)。重复概念意味着有两个模型元素(及其实现)表达的实际上是同一个概念。当这个信息每次发生改变时，都不得不在两处进行转换更新。每次由于新知识导致其中一个对象发生改变时，也必须重新对另一个进行分析并相应作出改变。如果在实际操作时不重新分析，那么同一个概念会有两种版本，采用不同的规则，甚至具有不同的数据。最

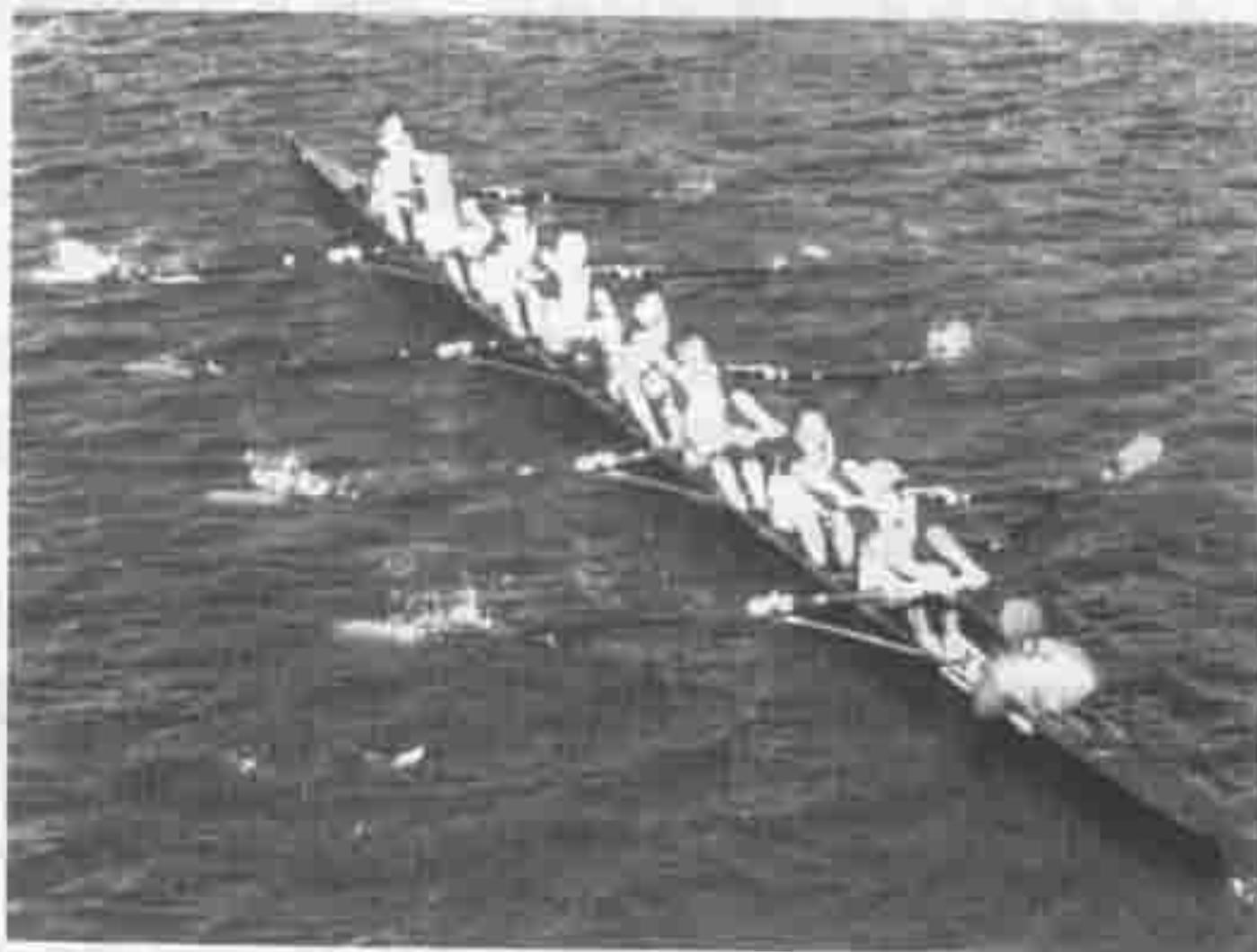


重要的是，团队成员必须认识到，是两种而不是一种方式在处理同一件事情，这两种方式在开发过程的各个方面都要保持同步。

假同源可能比较少见，但是具有更加难以察觉的危害性。这就是，当两个人在使用相同的术语(或者实现对象)时，他们会认为他们谈论的是同一件事情，但实际却并非如此。本章一开始所举的示例(两个不同的业务活动都被称为 Charge)就是这样一个典型示例，当两个定义确实关联领域中的同一个方面，只在概念上有很小的不同时，冲突可能就会更加难以察觉。假同源导致开发团队相互使用对方的代码、数据库，而这些代码和数据库却存在着出入，使两个团队之间的交流变得很混乱。术语“假同源”平常用于自然语言中。例如，说英语的人在学习西班牙语时常常会错用单词 *embarazada*。其实这个词的意思不是 *embarrassed*，而是 *pregnant*。

当您检测到这些问题时，您的团队将不得不作出应对决策。您可能想重新对这个模型进行开发，并仔细制定开发的过程以防止模型碎片出现。但是模型碎片也可能是团队分组开发的结果，出于某些好的想法，他们想以不同的方式进行开发，而且您可能也决定让他们这样做。本章后面介绍的模式将讨论如何来解决这些问题。

## 14.2 持续集成



定义好一个限界上下文后，我们还必须让它合理化。

当许多人在同一个限界上下文中工作时，很有可能会造成模型碎片。开发团队越大，出现的模型碎片就会越多，但是只有三四个人可能会碰到严重的问题。把系统分成更小的上下文最后会失去有效的集成和一致性。



有时候开发人员还没有完全领会某些对象的意图或者别人所建模的交互，就想着去改变它，导致它不象原来的目的那样使用。有时候他们没有意识到他们要实现的概念已经在模型的另一部分中实现了，却仍然(不正确地)去重复开发那些概念和行为。有时候他们明知道还有另外的表示方法，却害怕修改模型，生怕破坏了现有的功能，所以就去重复开发那些概念和功能。

不管开发的系统规模有多大，维持系统统一所要求的交流都很难达到。我们需要各种增进交流的方法并减少交流的复杂程度。同时还需要可靠的措施，防止过于谨慎的行为，比如开发人员因为害怕破坏原有代码而重复开发功能。

极限编程模式应运而生。许多 XP 实践就是针对如何在经常被很多人更改的情况下维护一个一致的设计这个特定问题而提出来的。在 XP 最纯粹的形式下，来维护模型在一个限界上下文中的完整性是非常适合的。然而不管是否使用 XP，使用一些持续集成的过程都是至关重要的。

持续集成意味着合并上下文里的所有工作，经常保持它们的一致，一旦产生碎片就能及时地被发现并纠正过来。正如领域驱动设计的所有情况一样，持续集成也分两个级别：模型概念集成和实现集成。

通过团队成员的不断交流可以集成概念。团队必须对经常改变的模型形成共同的理解。虽然许多实践为我们提供了很多有益的帮助，但是不断地锤炼出通用语言才是最基本的。同时，通过一个系统的合并/创建/测试过程，能够及早发现模型碎片。虽然有不少的集成流程在使用，但是大多数有效的方法都具有以下特点：

- 分步骤的、可重用的合并/创建技术；
- 自动化测试集
- 一些规则，用来对没有集成的变化设置稍高的生命期上限。

概念集成虽然很少被正式纳为一种有效的过程，但是它的作用也是不容忽视的。

- 在讨论模型和应用时不断练习通用语言

大多数敏捷项目至少会每天合并每个开发人员的代码变化。合并的频率可以根据改变的速度来调整，只要在其他团队成员做出大量不相容的工作前，把所有没有集成的变化进行合并就可以。

在模型驱动设计中，概念集成为实现集成扫清了道路，实现集成证明了模型的有效性和一致性，并且暴露模型碎片。

因此：

建立一个过程，负责经常合并所有的代码和其他实现工件，同时进行自动化测试，以便很快标记出碎片。必须不断地运用通用语言，锤炼出对模型的共同认识，从而在不

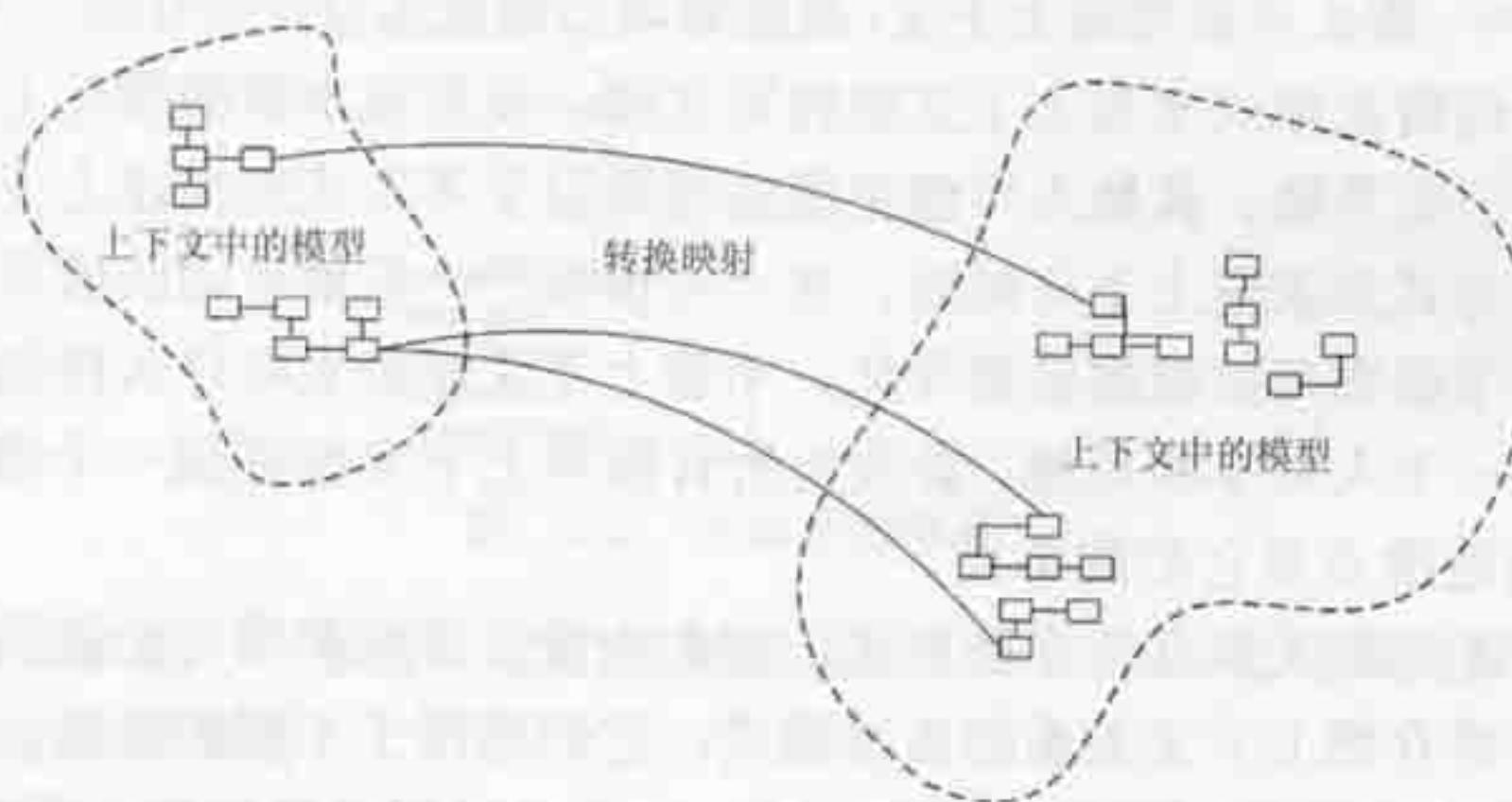


同人的脑子里形成统一的概念。

最后记住，千万不要把工作范围扩大。只有在一个限界上下文内，持续集成才是有用的。涉及相邻上下文的设计问题，包括转换，都不必以同样的速度处理。

在任何单个限界上下文中，对超过两个人的工作进行持续集成都是可行的。它维持该模型的完整性。如果同时存在多个限界上下文，您必须根据它们之间的关系做出决定并且设计所有可能需要的接口。

### 14.3 上下文映射



一个限界上下文不能提供全局视图，所以其他模型的上下文可能仍然不清楚而且变化不定。

其他团队中对上下文界限不太清楚的人，将会不知不觉地改变界限，使其边缘变得模糊或者使互相间的连接变得复杂。如果不同的上下文之间必须要进行连接，它们往往会相互融合。

在限界上下文之间重用代码是一种冒险，应该尽量避免。功能和数据的集成必须通过一个转换来实现。通过定义不同上下文之间的关系，创建一个包括项目中所有模型上下文的全局视图，能够减少混乱。

上下文映射处于项目管理和软件设计的重叠部位，通常是根据团队组织的轮廓来自然地划分界限。在一起工作的人自然共享一个模型上下文。不同团队的人，或者即使他们属于一个团队但不需要交流的人，都将分裂到不同的上下文中。物理办公空间也会产生影响，如果没有额外的整合工作，就应该把那些位于一幢建筑物两头的团队成员分到不同上下文，更不要说在不同城市的团队成员了。大多数项目经理直觉上认为这些因素是合理的，并且都大致围绕着开发的子系统来组织团队，但是团队组织结构以及软件模



型和设计之间的相互关系仍然不是很突出。经理和团队成员都必须对正在进行的软件模型和设计的概念细分有一个清晰的认识。

因此：

明确每一个模型在项目中的作用，明确定义它的限界上下文。其中包括非面向对象子系统的固有模型。为每一个限界上下文命名，使其成为通用语言的一部分。

描述模型间的接触点，为每次交流大致描述出显式的转换，并且突出所有共享的东西。

画出现有的疆界。稍后再进行转换。

在每一个限界上下文中，都会有一种一致的通用语言的方言。把限界上下文的名称加入到这种语言中，那么只要理清上下文，就能够明白地说出设计中任何一个部分的模型。

不需要用任何特定格式来为上下文映射写文档。我发现本章的图在上下文映射的可视化和沟通上有一定帮助。其他人可能更愿意选择以文本方式来描述上下文映射或者以另外的图形表现方式来表现上下文映射。在一些情况下，只要有团队成员之间的讨论就足够了，而且细节级别可以根据需要变化。不管上下文映射采取什么样的方式，必须能够被项目中的每一个人共享和理解，必须为所有限界上下文都提供一个明确的名称，必须清楚地表示出连接点及它们的本质。

限界上下文之间的关系具有多种形式，它们由设计情况和项目的组织情况来共同决定。稍后，本章将介绍上下文关系的各种模式，它们适用于不同的情况，并且能够给出描述您在自己的图里发现的关系的术语。记住，上下文映射总是代表上下文所处的位置，您发现的关系最初可能并不适合这些模式。如果这些上下文关系跟某个模式很相似，您可能马上就会想到，用这个模式名来描述它们，但不要硬套模式名。使用模式名仅仅是为了很好地描述出在图中发现的关系而已，随后您就可以开始转向更加标准化的关系。

如果发现了一个碎片——一个完全缠绕的模型包含了不一致性，那么该怎么处理呢？此时一定要有所警觉，并停止描述任何事情。然后以一个准确的全局视图，标注出混淆点。小碎片是可以修补的，只需要提供相应的过程来处理就行了。如果一个关系不是很明确，可以选择一个最接近的模式来处理。首要的任务是得到一张清楚的上下文映射，这样就意味着已经真正解决了您所发现的问题。但是不要让这些必要的修补导致全局的重新组织。即使有一个明确的上下文映射，虽然图中的限界上下文包含了所有的工作，所有连接模型之间的关系都非常清楚，这也只不过是消除了明显的矛盾而已。

一旦有了一致的上下文映射，就会看到您打算要改变的东西。您可以慎重地更改团队的组织或者设计。记住，如果实际中没有作任何改变，千万不要去改变已有的上下文映射。



### 示例：运输应用中的两个上下文

再次回到运输系统。应用的一个主要特性是在预订的时候能够自动计算出货船的航线。模型看起来如图 14-2 所示：

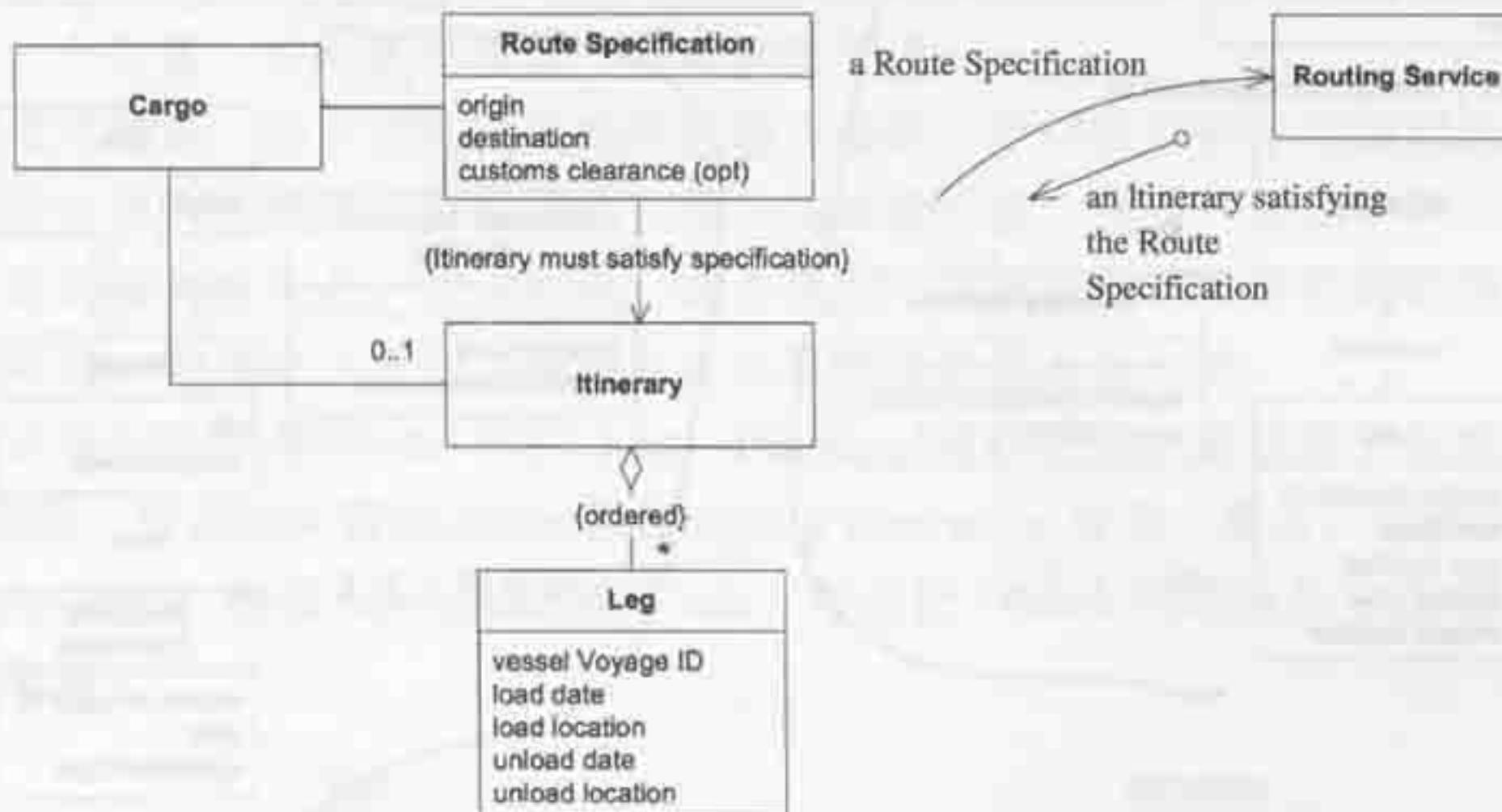


图 14-2 运输系统模型

**Routing Service** 是一个服务，它把机制封装在无副作用函数组成的释意接口上。这些函数的结果由断言来刻画。

- 当向接口中传入一个 **Route Specification** 时，将返回一个 **Itinerary**。
- 断言规定，返回的 **Itinerary** 要满足传入的 **Route Specification** 的需求。

这个模型还没有规定如何去完成这项艰巨的任务。现在就让我们来到幕后看看这个机制。

最初，我在项目上对 **Routing Service** 的内部处理显得非常的教条。我希望实际的航线安排要由扩展的领域模型来完成，这个模型代表船的航次并且直接与 **Itinerary** 的 **Leg** 相关联。但是负责航线安排问题的团队指出，为了提高安排航线的性能并且制定良好的选路算法，我们需要把它实现为一个优化网络，其中航次的每一个航段将被作为一个矩阵元素来对待。他们坚持要用一个单独的运输操作模型来达到这个目的。

对航线安排需要进行的计算以及后面的设计，毫无疑问他们是正确的，由于没有更好的做法，我屈服了。结果我们创建了两个分离的限界上下文，每个都有自己的对航运操作概念的组织，如见图 14-3 所示。

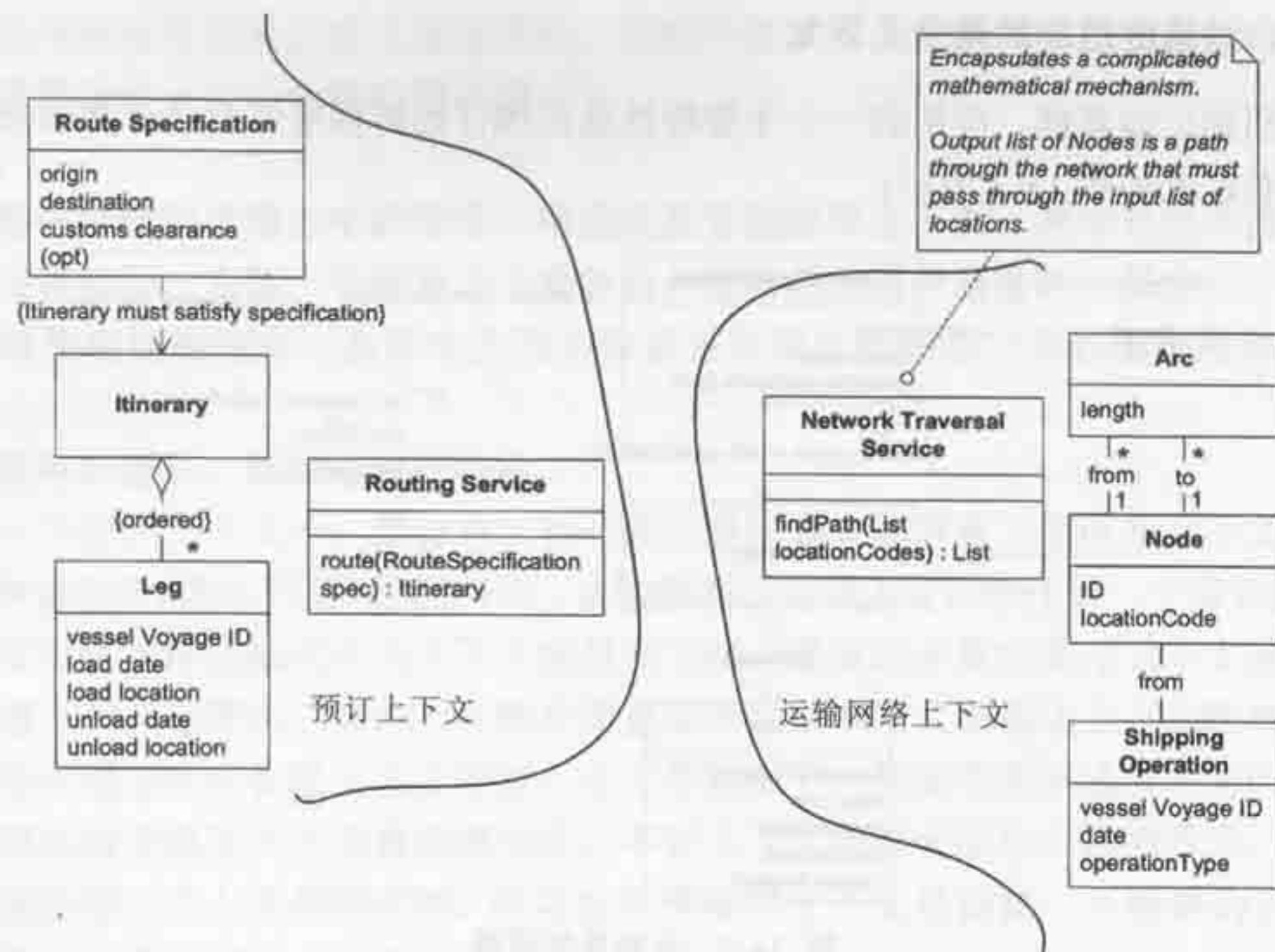


图 14-3 形成两个限界上下文，允许应用有效的航线算法

我们需要做的是，把 Routing Service 的请求转换成为 Network Traversal Service 能够理解的术语，然后把结果转换成为 Routing Service 希望的格式，如图 14-4 所示。

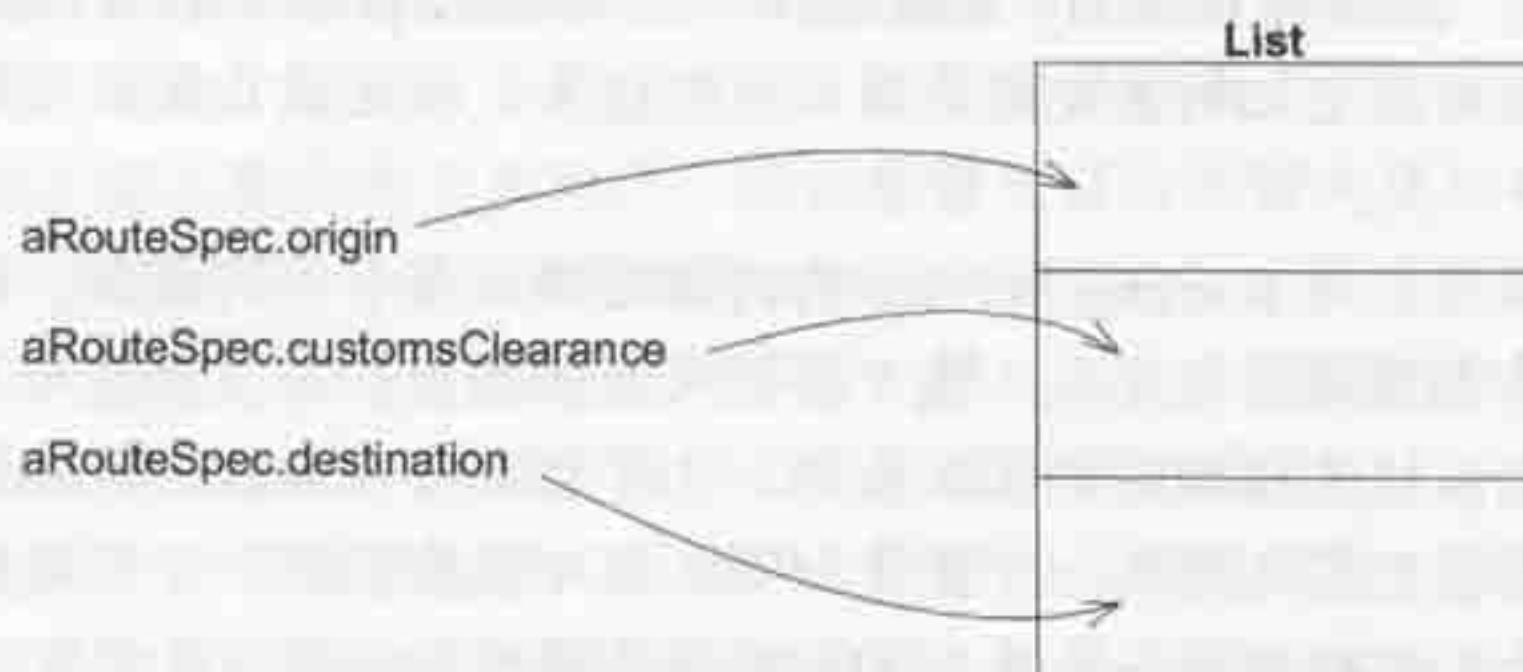


图 14-4 转换到 Network Traversal Service 的一个查询

这意味着不必映射这两个模型中的所有东西，只要能够完成两个特殊的转换：

- Route Specification → 地点编码的 List
- Node 标识的列表 → Itinerary

为实现这两个转换，我们必须考虑其中一个模型的元素表示的含义是如何在另一个



模型中表示的。

从第一个转换开始(Route Specification → 地点编码的 List)，我们要考虑列表中地点的顺序表示什么意思。列表中的第一个地点代表这条航线的起点，按照此航线依次到达每一个地点，直到到达列表中的最后一个地点。那么起始地和目的地就在列表中的第一个和最后一个位置，中间的是需要一些海关清关点。

值得庆幸的是，两个团队使用相同的地点编码，所以我们就不用处理这种转换了。

注意反过来的转换是没有意义的，因为网络遍历输入允许任意的中间点，而不是一个特别指定的海关清关点。幸运的是，因为我们不需要进行这个方向的转换，所以不会产生什么问题，但是这让我们了解到为什么有些转换是不可行的。

现在让我们对结果(Node 标识 List→Itinerary)进行转换。假设我们能够根据收到的 Node 标识用一个仓储来查询 Node 和 Shipping Operation 对象。那么，这些 Node 是怎么映射到 Leg 的呢？根据 operationTypeCode，可以把 Nodes 列表分为出发/到达对。每对代表一个航段。

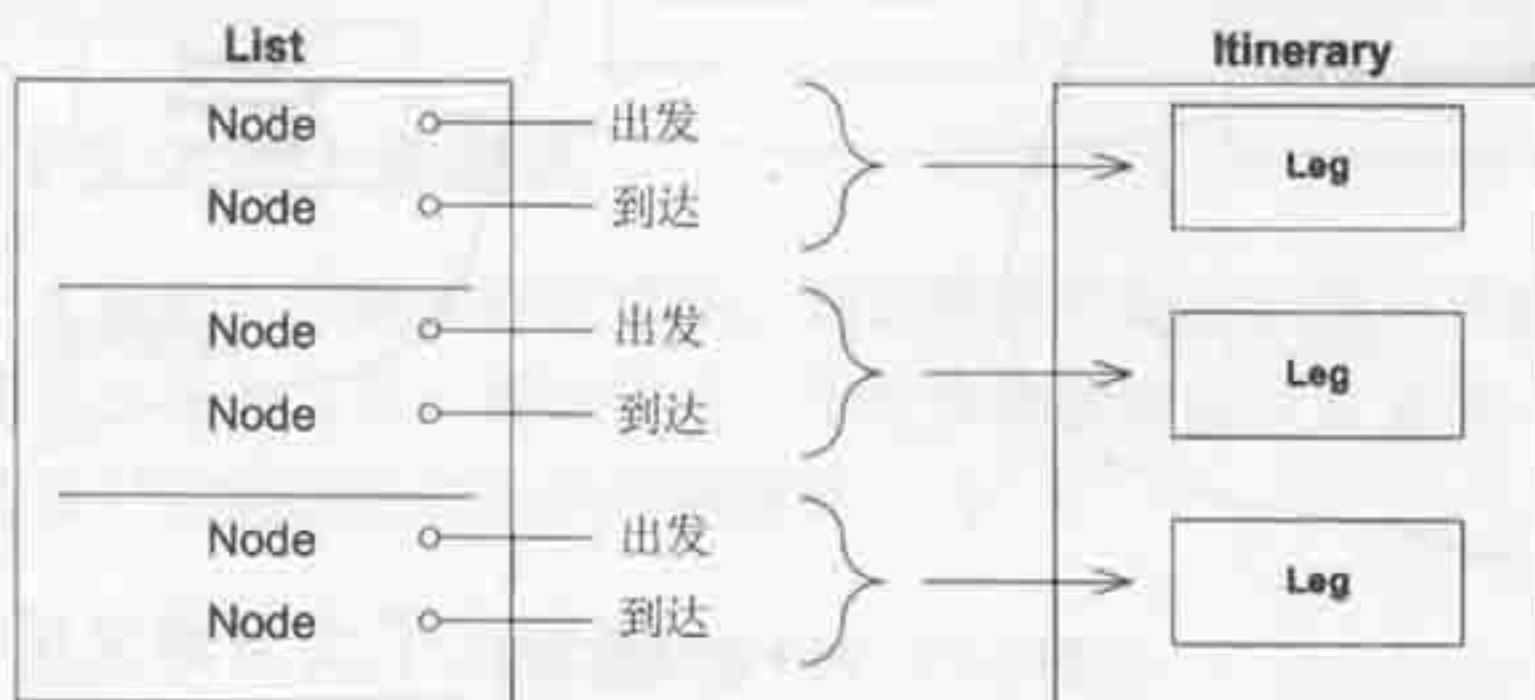


图 14-5 转换由 Network Traversal Service 发现的航线

每个 Node 对的属性被映射如下：

```
departureNode.shippingOperation.vesselVoyageId → leg.vesselVoyageId  
departureNode.shippingOperation.date → leg.loadDate  
departureNode.locationCode → leg.loadLocationCode  
arrivalNode.shippingOperation.date → leg.unloadDate  
arrivalNode.locationCode → leg.unloadLocationCode.
```

这是两个模型间的概念转换映射。现在必须来实现这些转换，如图 14-6 所示。在这个简单的例子中，为了实现这些转换，我一般创建一个对象，然后找出或者创建另一个对象，为剩下的这些子系统提供服务。

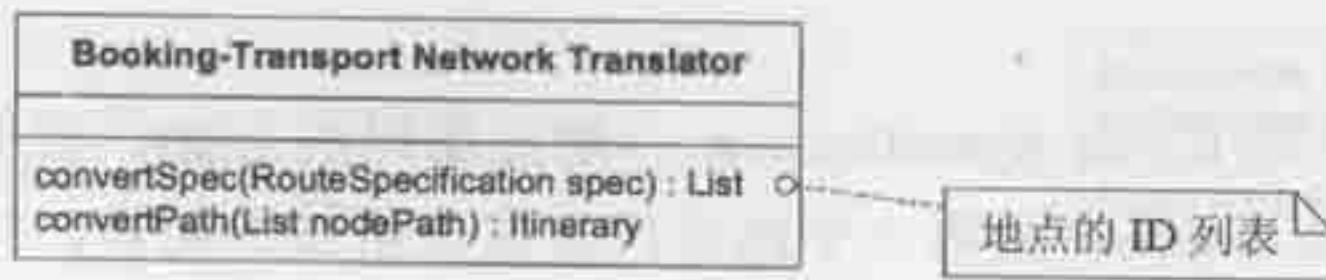


图 14-6 双向转换器

这个对象必须由两个团队来一起维护。设计应该使得单元测试很容易，因此最好由团队合作来为它编写一个测试集。除此之外，他们可以按各自的方式进行开发，如图 14-7 所示。

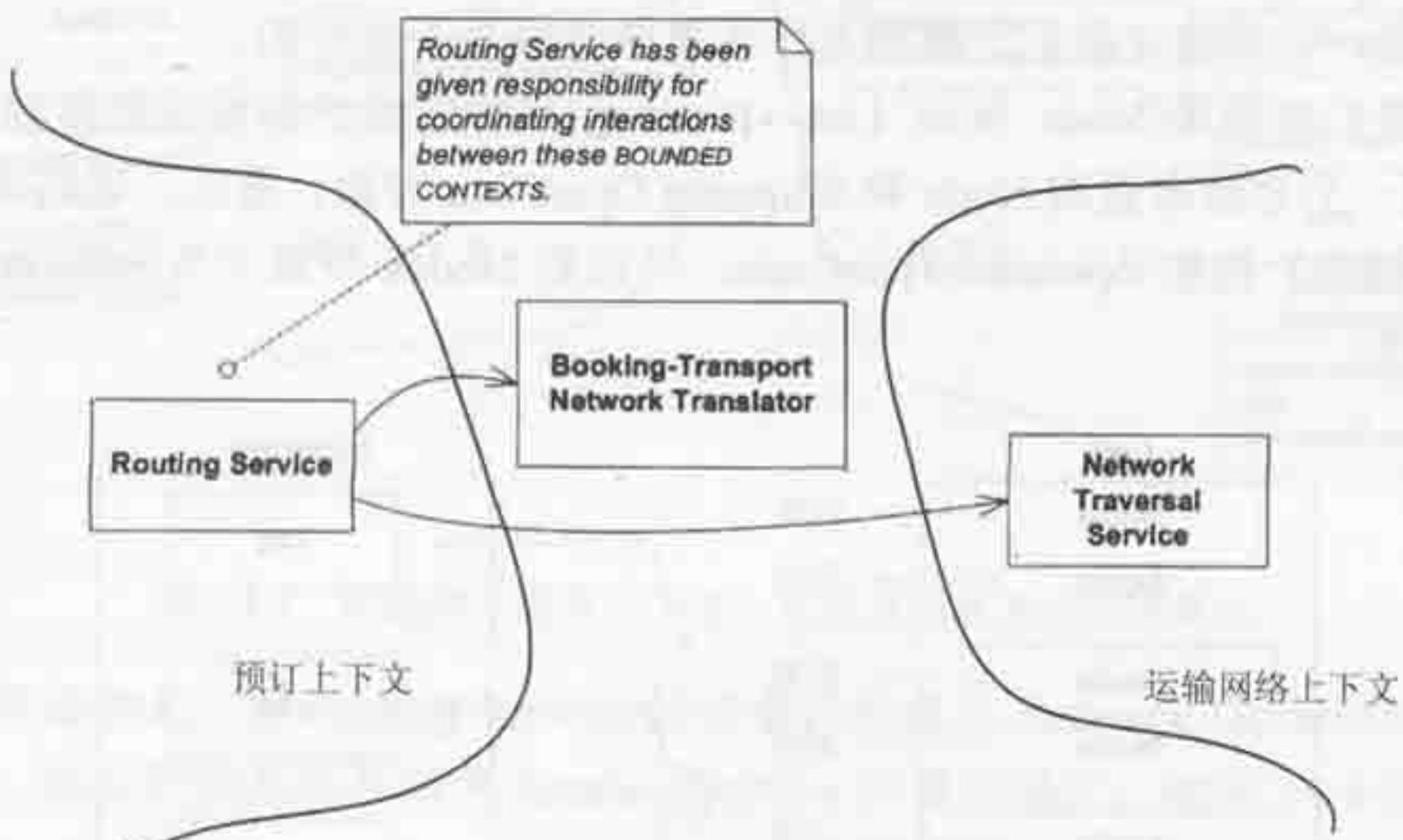


图 14-7 两个团队的工作

Routing Service 的实现现在委托到了 Translator 和 Network Traversal Service。它的一个操作看起来如下：

```
public Itinerary route(RouteSpecification spec) {  
    Booking_TransportNetwork_Translator translator =  
        new Booking_TransportNetwork_Translator();  
  
    List<ConstraintLocation> constraintLocations =  
        translator.convertConstraints(spec);  
  
    // Get access to the NetworkTraversalService  
    List<nodePath> pathNodes =  
        traversalService.findPath(constraintLocations);  
  
    Itinerary result = translator.convert(pathNodes);  
}
```

```
    return result;  
}
```

还行，限界上下文使得每一个模型都比较清楚，使得各个团队在很大程度上都能独立完成工作，如果最初的假定正确，或许会表现得更加出色(在本章后面将再次讨论这一点)。

这两个上下文的接口是相当小的。Routing Service 接口把预定上下文设计中的剩余部分与航线查找中的事件分离。因为接口是由无副作用函数组成的，所以很容易测试。要与其他上下文和谐共存，其中的一个秘诀是拥有一个有效的接口测试集。正如里根总统在进行裁减武器谈判<sup>1</sup>时曾经说过：“信任，但要核实(Trust, but verify)”。

把 Route Specifications 输入到 Routing Service 中，并检查返回的 Itinerary，设计这样的自动化测试集是一件很容易的事情。

模型的上下文总是存在的，但是没有下意识地去关注这个问题时，这些上下文可能会相互重叠并且经常变化。通过明确定义限界上下文和上下文映射，团队就可以开始考虑模型的统一和模型的连接了。

### 14.3.1 在上下文边界上的测试

对限界上下文接触点的测试特别重要。测试能够帮助我们弥补转换时出现的小疏忽以及在边界上需要交流的信息。测试作为一个有价值的提前报警系统，尤其能够对那些要依赖的却不由我们控制的模型提供可靠保证。

### 14.3.2 组织和文档化上下文映射

这里只有两个要点：

- 为了谈论它们，必须给限界上下文命名，这些名称要添加到团队的通用语言中。
- 每个人都必须知道边界的位置，并且能够识别任意一段代码或者任何情况的上下文。

有很多种方式可以满足第二个要求，这由团队文化决定。一旦限界上下文定义好，自然而然地会把不同上下文的代码放到不同的模块中，那么就会产生如何知道哪个模块属于哪个上下文的问题。这时可以用一个命名约定来指示，只要能避免混淆而且简单，采用任何其他机制也可以。

对概念边界的交流同样重要，团队里的每一个人必须以相同的方式了解这些边界。

<sup>1</sup> 里根在这里引用了一句俄罗斯谚语，从而增加了双方的信任——这是连接上下文的一种隐喻。



为了达到交流的目的，我喜欢使用像例子中的那些非正式图形来表达我的思想。当然也可以用更加严格的图形或者文本列表，沿着接触点以及连接和转换的机制，展现出每个上下文中的所有包。一些团队比较适合使用这种方法，而其他一些团队则喜欢通过口头协议和大量的讨论来交流。

无论如何，如果打算把上下文中的名称加到通用语言中，那么对上下文映射进行讨论是很重要的。不要说，“George 团队中的内容改变了，因此我们不得不改变我们对应的那些内容”。而是应该说，“Transport Network 模型改变了，因此我们不得不改变 Booking 上下文的转换器”。

## 14.4 限界上下文之间的关系

下面的模式包括了关于处理两个关联模型的一些策略，这两个模型的组合可以包含整个企业。这些模式有双重目的，一是成功地组织开发工作，一是提供描述现有组织的词汇。

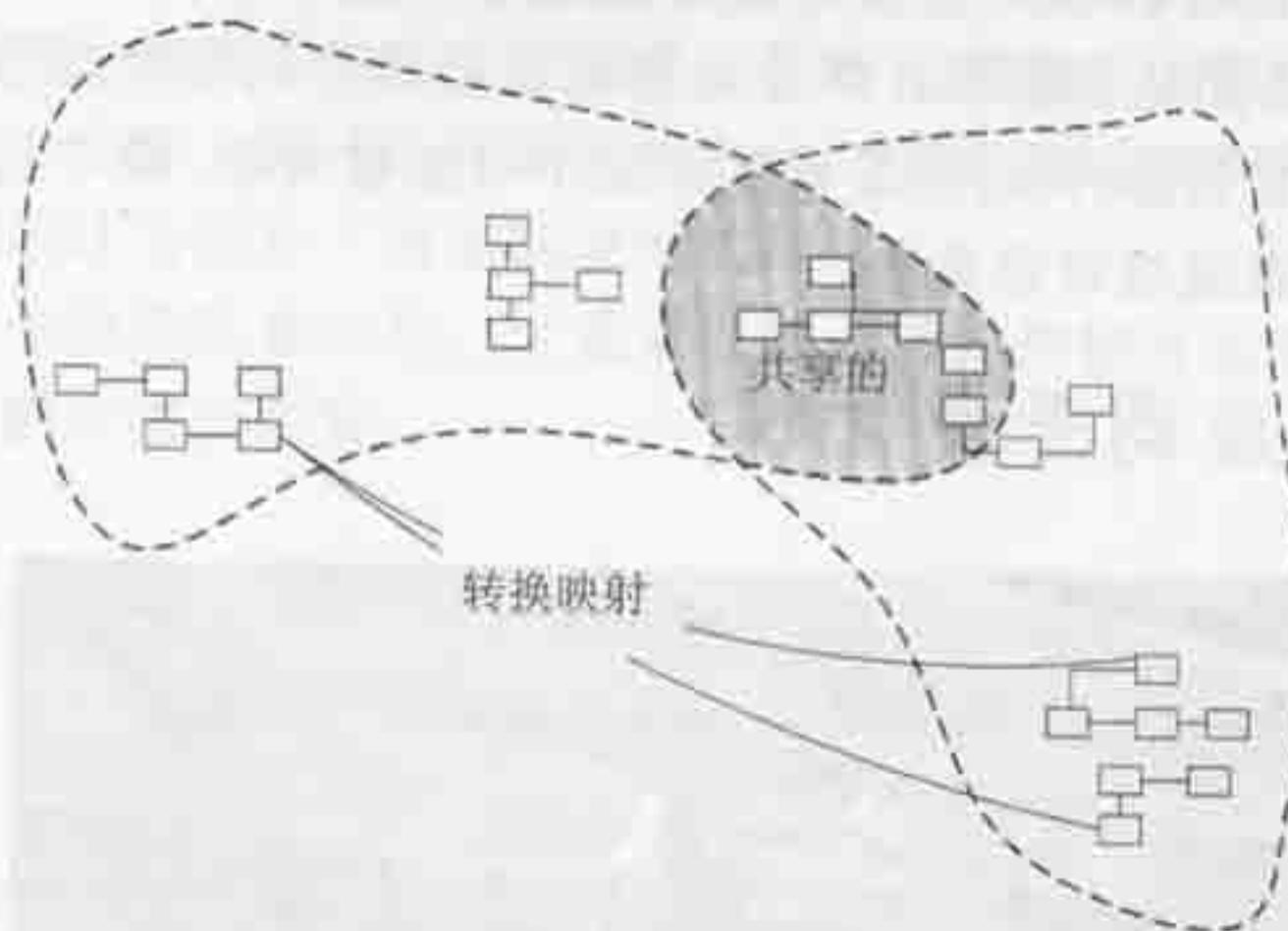
不管是由于碰巧还是由于设计的原因，现有的关系可能会与其中的某个模式中相似，在这种情况下，可以使用这些术语或者它们的变体来描述这种关系。然后，对它们稍微进行一点设计改动，使这个关系能够更接近于所选的模式。

另一方面，您可能会觉得现有的关系太乱或者太复杂。需要进行一些必要的重新组织，尽可能地使模糊的上下文映射变得清晰。在这种情况下，或者任何需要考虑重新组织的情况下，这些模式都给出了不同的选择。它们之间的不同地方主要包括对其他模型的控制程度、团队合作的程度和类型，以及特性和数据的集成程度。

下面的模式集包含了一些最常用的和最重要的案例，它们能够帮助您更好地知道如何动手处理其他的案例。紧密工作在一个高度集成产品之上的高明团队可以部署一个大型的统一模型。由于需要服务于不同的用户群体或者由于团队的协调能力有限，可能会产生共享内核模式(Shared Kernel)或者顾客/供应商模式(Customer/Supplier)。有时候如果确实认为集成并不重要，那么系统最好采用独立模式(Separate Way)进行设计。当然，很多项目在一定程度上必须与老式系统以及外部系统进行集成，这样就可能会导致开放主机服务模式(Open Host Service)或者防腐层模式(Anticorruption Layer)。



## 14.5 共享内核



如果功能集成有限，我们就会认为持续集成的费用太高。当团队缺乏技巧或者策略的组织来维持持续集成时，当团队太大而显得笨拙时，就完全没有必要持续集成。所以可以确定分离的限界上下文并且组成多个团队。

虽然没有协调的团队能够暂时地取得快速进展，但是他们开发出来的东西也许根本就不能结合。他们应该停止把更多的精力放在转换层上进行花样翻新，而应该把精力放在持续集成上面，与此同时，他们做了重复的工作并且失去使用通用语言所带来的好处。

在许多项目中，我曾经发现在团队间共享的基础结构层是非常独立的。在领域设计中采用类似的结构也是非常合适的。要想实现整个模型和代码库的完全同步，需要付出非常大的代价，但是同步一个仔细挑选的子集，却可以用比较少的代价换取同步整个模型的优势。

因此：

指定领域模型中两个团队同意共享的子集，当然，这里面还包括了相关的代码或者数据库设计。明确共享的内容有特殊的状态，在没有与其他团队磋商前是不能改变的。

经常对功能系统进行集成，但是集成的频率要比在团队内进行的持续集成的频率稍微少一点。在这些集成中，两个团队的测试都要运行。

这需要我们仔细地平衡。共享内核不能象设计中的其他部分那样，可以随意改变，只有与另一个团队磋商后才能作出决策。因为在作出更改后必须要通过两个团队的所有测试，所以必须把自动化测试集集成进去。通常各个团队只是在各自的内核备份上改动，每隔一定时间才会与其他团队集成。例如，一个团队中的持续集成每天都会进行，甚至会



更加频繁，但核心的合并却可能是一周才进行一次。不管规定什么时候集成代码，团队之间对所作的改动讨论得越早越好。

共享内核通常是核心领域，但也可能是通用子域的某些集合，或者两者都是(参见15章)，总之只要双方团队都需要，共享内核可以是模型的任何部分。设置共享内核的目的只是为了减少重复资源开发(但是重复开发不可能被消除，除非只有一个限界上下文)以及使两个子系统的集成相对容易些。

## 14.6 顾客/供应商开发团队



通常在这种模式中，一个子系统必须为另外一个子系统服务。“下游”的组件执行分析的功能或者其他的功能，这些功能很少会把结果反馈给“上游”的组件，所有的依赖关系都是单向的。一般两个子系统服务于不同的用户群，它们所做的工作不同，所以采用两种不同的模型可能会更好。使用的工具集也可能不一样，因而不可能共享编程代码。

上游和下游子系统自然会被分在两个限界上下文中。当两种组件在实现中需要采用不同的技巧或者使用不同的工具集时，将它们分在两个环境中会更加理想。因为仅仅需要单方向操作，所以转换更加容易。但是问题可能出在两个团队的策略关系上。

如果下游团队对改动有否决权，如果请求改变的过程太繁琐，那么就会限制上游团队随心所欲地开发。上游团队甚至可能会过于谨慎，担心破坏了下游的系统；同样，由于上游具有优先权，从而使下游团队感到无助。

下游需要从上游获取服务，但是上游却不负责处理下游提供的结果。要想预见下游团队会对上游团队产生什么样的影响，预见每个人的任务是什么，以及每个人的进度表是什么，得需要付出很大的努力。把两个团队之间的关系形式化，可以使每个人的开发工作更加简单。可以对这种过程进行组织，协调两个用户群的需要，并根据下游的特点



制定出工作计划。

在极限编程项目中，对此已经有了一个合适的机制：迭代规划的过程。所有需要我们做的就是根据规划过程确定两个团队的关系。下游团队的代表作为用户代表参与规划，直接与他们的“顾客”协商顾客需要的任务。这种机制最后得到的是为供应商团队制定的迭代规划，规划中包括下游团队最需要的任务，或者通过协商需要延期开发的任务，所以不要把这种规划认为这是下游团队交付的应用。

如果不用 XP 而使用其他过程，无论采取什么样的类似方法来协调不同用户的关注点，都应该包括下游应用的需要。

因此：

在两个团队之间建立一个清楚的顾客/供应商关系。在规划过程中，让下游团队扮演上游团队的顾客角色。根据下游的需求进行磋商和预算任务，让每个人都知道承担的义务和进度表。

联合开发自动化验收测试来检验期望的接口。把这些测试作为持续集成的一部分加入到上游团队的测试集中。这个测试能够使上游团队在对模型作出改动时不用担心会对下游产生副作用。

在迭代中，下游团队的成员就像传统的顾客一样，要能够回答并帮助解决上游开发人员提出的问题。

验收测试自动化是顾客关系的一个重要部分。即使项目中的大部分工作是合作完成的，尽管顾客能够确定他所依赖的部分并且能与之交流，同样供应商也可以不停地交流改变的信息，但是如果缺少测试，就会发生奇怪的事情。它们会打断下游团队的工作，迫使上游团队采取不在规划内的紧急修复工作，反而会让顾客团队与供应商团队协作来开发自动化验收测试来验证期望的接口。上游团队将把这些测试作为它标准测试集的一部分。因为改变测试就意味着改变接口，所以对这些测试的任何改变都需要与另外那个团队交流。

顾客/供应商关系也会出现在不同公司的项目之间，在这种情况下，一个顾客对供应商的业务来说是非常重要的。局部能够影响全局：一个有影响力的顾客提出的要求，对上游项目的成功可以说是至关重要的，但是这些要求也可能会破坏上游项目的开发。因为在权衡成本/收益上，外部关系比起 IT 环境内部的关系更难，所以双方都受益于响应需求的形式化过程。

该模式有两个关键元素。

- 必须是顾客和供应商的关系，并且顾客的需要极为重要。因为下游团队不是唯一的顾客，所以必须通过协商来协调不同顾客的需要，但是要对顾客设置优先权。该情况与常出现的“穷亲戚(poor-cousin)”关系作对照，在这种关系中，下游团队必须向上游恳求它所需要的服务。



- 必须有一个自动化测试集允许上游团队改变它的代码而无需害怕破坏下游的开发，让下游团队把精力集中在自己的工作上而不需要经常去留意上游团队所作的变化。

在一场接力赛中，跑在前面的运动员不能老是回头看，他必须信任后面的运动员能够把接力棒正好交到他手上，否则整个团队的速度必定会受到影响而减慢下来。

### 示例：收益分析与预订

回到我们的运输系统中来。组建一个非常专业的团队来分析公司中的所有预订，看看如何获得最大收入。团队成员可能会发现船有空余空间，就会推荐接受更多的超额预订；而船只一旦装满了散货时，公司就不得不放弃运输那些可获更多利益的货物了。如果那样的话，他们会建议要么为那些有利可图的货物预留空间，要么提高散货的运输价格。

他们使用自己他们的一个复杂模型来完成这种分析。在实现上，用一个带工具支持的数据仓库来创建分析模型，而且他们需要从预订应用上获取很多的信息。

因为他们使用的实现工具不同，更重要的是，创建了不同的领域模型，所以一开始就很清楚地认识到这是两个限界上下文。它们之间的关系是什么呢？

共享内核模式可能看起来会比较合乎逻辑，因为收益分析只对“预订”模型的一个子集感兴趣，而且这两个模型在货物、价格等概念上有一些重复。但是如果使用了不同的实现技术，共享内核模式就难以运用了。除此之外，收益分析团队的建模需要是十分专用的，他们不断地修改他们的模型，并且尝试着其他的方法。他们可能很容易就能把“预订”上下文中有用的部分转换成自己的东西(另一方面，如果他们能够使用共享内核的话，那么转换的负担将会更轻。他们仍然要重建模型并把数据转换到新的实现中去，但是如果模型是一样的，转换就简单多了)。

因为不打算制定自动协调策略，所以“预订”应用根本不依赖于收益分析。专家们将作出决策并传达给必要的开发人员和系统。因而我们有一个上游/下游的关系。下游的需要是：

- 一些任何预订操作都不需要的数据
- 稳定的数据库方案(至少能可靠地通告改变)或者导出实用程序

幸运的是，预订应用程序开发团队的项目经理打算帮助收益分析团队。可是存在着这样一个问题，因为运作部门每天实际做的预订报表都是呈送给一个副总经理而非真正做收益分析的那个人。但是高层管理者非常关心收益的管理，而且他曾见过两个部门在过去合作时出现的问题，也曾经亲自组织过软件开发的项目，所以他决定两个团队的项目经理都向同一个人汇报。

由此可见，所有需求都适合采用顾客/供应商开发团队模式。

我曾经在很多个地方都看到过这种场景，分析软件的开发人员和操作软件的开发人



员之间有一个顾客/供应商的关系。如果上游团队的成员认为自己是在为顾客服务，采用这种模式就非常顺利了。但是这种关系的组织几乎总是不正式的，所以每次得到的结果都像是两个项目经理之间的私人关系一样。

在一个 XP 项目上，我曾见过这种正式的关系，在每一次迭代时，下游团队代表用顾客的身份来“计划游戏”，与更普通的顾客代表(应用功能)一起协商是哪些任务使它进入迭代计划。这个项目属于一个小公司，所以最近的共同上司在这条关系链上所处的位置不会太高，这个模式在该项目使用得非常合适。

如果两个团队在同一个管理部门中，所以他们基本上拥有共同的目标，或者他们在不同的公司里，但实际上符合这种模式，那么采用顾客/供应商团队模式就更加成功。但是当没有条件来激发上游团队时，情况又会非常不一样了。

## 14.7 同流者



当具有上游/下游关系的两个团队不是由一个源头指挥的话，就不能采用象顾客/供应商这种合作开发的模式。如果硬是要采用这种模式，那么下游团队就会碰到很多麻烦。这种情况经常会在大公司中，当两个团队所在的管理层相隔太远，或者共同的管理者与两个团队的关系无关紧要时，就不适合使用顾客/供应商模式。如果顾客业务对供应商来说不是特别重要，那么处于不同公司的两个团队也会产生这种情况。也许供应商有许多小顾客；也许供应商打算改变市场方向并不再重视老顾客；可能是由于供应商的运



行状况不好；可能这项业务已经被停止了；但不管出于什么原因，事实上，下游都得依靠自身发展。

两个具有上下游关系的开发团队，如果上游不打算为下游团队提供服务的话，那么下游团队就没有如何办法了。虽然利他主义可以驱使上游开发人员作出一些承诺，但是这些承诺未必能够兑现。如果下游团队对这种美好的承诺充满了信任，就会基于这些根本不可能实现的承诺去制定计划。下游项目将会被耽搁，直到它最终学会只能依靠从上游那里获得的东西去进行开发。所以根据下游的需要裁剪接口是不可能的。

在这种情况下，可以有3种选择。一种选择是完全放弃使用上游。这种选择应该根据实际情况来评估，因为它假定了上游不会提供下游的需要，但有时候我们高估或者低估了这种依赖的代价。如果下游团队决定自由开发，那么他们就是打算采用“隔离方式”模式(参考本章后面对此模式的描述)。

有时候下游使用上游软件的价值是如此巨大，因而不得不维持这种依赖(或者有策略规定团队不能改变这种关系)。在这种情况下，仍然要保持两条开发路线，但如何选择开发的方式将由上游的设计质量和风格来决定。如果是因为缺乏封装、不合适的抽象，或者用团队不能使用的范型来建模，那么下游团队仍需要开发自己的模型，而且必须由他们完全负责实现的转换层大概会很复杂(参见本章后面介绍的防腐层模式)。

### 追随并不总是坏事

当使用一个现成的大型接口组件时，一定得遵照该组件隐含的规则。因为组件和应用无疑是两个不同的限界上下文，所以基于团队的组织和控制，可能需要用适配器来减少格式的变化，但是它们采用的模型应该是相同的，要不然，就应该怀疑使用这个组件的价值。如果认为这个组件有足够的使用价值的话，就说明在它的设计中可能包含了消化的知识。从有限的角度看，现成的组件里包含的思想可能会比您自己理解的东西更加先进。您的模型大概会扩展这个组件的范围，并且您自己的概念将会在这些扩展的部分中得到发展。但是不管它们连接到哪里，您的模型只是个同流者，得服从这个组件模型的领导。如果这种开发方法有效的话，您就能够开发出一个更好的设计。

当您的模型与组件的接口非常小时，共享一个统一的模型并不太重要，而且转换是一种可行的选择方式。如果它是一个大型接口并且集成更加重要时，那么追随这个组件的模型进行开发通常会更有意义些。

另一方面，如果设计质量不是很差，风格又相当一致，那么最好能完全放弃开发独立模型的想法。这种情况我们称之为“同流者(conformist)”模式。



因此：

通过忠实地采用上游团队的模型，消除限界上下文之间转换的复杂性。虽然这种做法限制了下游设计者的风格，而且可能也不会为应用建立一个理想的模型，但是选择同流者模式使集成得到了极大的简化，同时将与供应商团队共享一种通用语言。这就好比供应商就坐在司机的位置上，最好能选择一种能让他们易于交流的简单方式。而且利他主义还能够让他们把信息与您共同分享。

这种决策加深了您对上游的依赖性，并且限制了您对上游模型能力的运用，以及附加的改进。就情绪上来说，这种选择其实一点都不吸引人，这就是为什么我们应该这样做却又不太愿意选择这种模式的原因。

如果对这些权衡您都不能接受，但上游的依赖又是不可缺少的，那么还有第二种方法可供选择：通过创建“防腐层”尽可能地隔离自己，这是一种实现转换映射的积极方法，我们将在后面讨论。

同流者类似于共享内核，因为它们都有一个重叠的区域。在该区域内，用的是同一种模型，您的模型通过附加而得到扩展，并且不会受到另外一个模型的影响。两个模式之间的不同地方在于决策和开发的过程。共享内核模式是两个团队紧密协调合作的结果，而同流者模式是跟一个对合作不感兴趣的团队进行集成。

我们已经介绍了在限界上下文之间集成的一些合作模式，从紧密合作的共享内核或顾客/供应商开发团队，到单方面顺从的同流者模式。最后我们将介绍一种更加没有依赖的关系，假定对方即没有合作的可能，也没有一个可用的设计。

## 14.8 防腐层





新系统差不多总是要与老式系统或者其他系统集成，那些系统都有自己的模型。当把设计好的限界上下文与合作团队的上下文连接起来时，其转换层可能会显得很简单甚至是雅致。但是如果在边界的另外一端开始出现漏洞时，转换层就得担当起更多的防御任务。

如果正在创建的新系统与其他系统必须依靠一个大型的接口来连接，那么关联两个模型的难度可能最终会推翻新模型打算表现的意图，而是以一种特别的风格被修改成类似于另外那个系统的模型。老式系统的模型通常很弱，即使是精心开发的例外，也可能会不适合当前项目的需要。然而在集成中可能会有许多有价值的东西，而且有时候是绝对需要集成的。

答案是不要一概拒绝与其他系统的集成。我曾经参加过一个项目，人们都有着狂热的开发劲头，打算把原来的所有东西都替换掉，但是需要做的工作实在太多，不可能马上实施。另外，与现有系统集成是重用的一种有用的形式。在大型项目里，一个子系统会经常与另外几个独立开发的子系统有接口。这些都会从不同的方面来反映这个问题领域。当把基于不同模型的系统结合起来时，要使新系统适应于另外那个系统的语义可能会破坏新系统自己的模型。即使那个系统的设计非常优秀，它也不能以同样的模型作为基础，更何况通常那个系统的设计并不是那么好。

与外部系统进行接口连接会有许多障碍。例如，基础结构层必须提供与其他系统交流的手段，而这些系统可能属于不同的平台或者使用不同的协议，所以必须把另外那个系统的数据类型转换成您的系统中的类型，但是通常忽视了那个系统没有使用同样的概念领域模型。

很明显，如果您从一个系统中获取数据，而在另一个系统中却曲解了它的含义，错误就会发生。这样做甚至可能会把数据库搞乱。虽然如此，这种问题还是会常常不经意地发生在我们身上。因为我们总是认为在系统之间传递的内容是原生数据，它们意思明确并且对双方都相同，但这种假设常常是错误的。数据与每个系统关联的方式引起了它们在意思上出现细微但重要的差别。即使原生数据元素确实具有相同的意思，但是让另外那个系统的接口进行如此底层的操作通常是错误的。尽管底层接口不根据系统自己的模型来承担解释原始数据的任务，它还是取得了那个系统模型解释数据以及约束其值和关系的能力。

我们必须在附在不同模型上的部件提供一种转换，因而这两个模型就不会被外来模型中未充分理解的元素破坏。

因此：

根据两个领域模型创建一个隔离层，为客户提供功能。该层可以通过现有接口与另



外一个系统会话，而无需修改那个系统。在内部，该层进行两个模型之间必要的双向转换。

讨论连接两个系统的机制时，可能会让我们想到数据从一个程序传递到另一个程序，或者从一个服务器传到另一个服务器的问题。我将从技术方面简要讨论一下通信机制的结合，但是千万不要把这样的细节同防腐层混淆，因为防腐层不是一个向其他系统传递消息的机制，它更像是一种转换机制，只是将概念上的对象和动作从一个模型和协议转换到另一个上去。

防腐层本身就可以变成一个复杂的软件。下一步我将概述在创建一个防腐层时需要考虑的一些设计事项。

#### 14.8.1 设计防腐层的接口

防腐层的公共接口通常像是一组服务，尽管有时候它能够以实体的形式出现。构建整个新的层面来负责两个系统的语义转换给我们提供了一个机会，即重新抽象其他系统的行为并且把它的服务和信息提供给我们的系统，并与我们的模型保持一致。在我们的模型里，把外部系统当作一个组件对待可能甚至会说不通，所以最好是使用多重服务(或者有时候使用实体)，以我们的模型来说，每种服务都有相应的职责。

#### 14.8.2 实现防腐层

一种组织防腐层设计的方法是，连同经常在系统间会话的通信和传递机制一起作为外观(Facade)、适配器(Adapter)(两个词都源自[Gamma et al. 1995])和转换器的组合。

我们需要集成的系统通常都有巨大、复杂、杂乱的接口。是实现上而非概念模型差异上的问题促使我们使用防腐层的，这是您试图创建这些接口时将会碰到的问题。就算不同时处理一个难以沟通的子系统接口，把一个模型转换到另一个模型的工作也已经够困难了。幸运的是，外观就是解决这样的问题。

外观是子系统的一个可选接口，它能够简化客户的访问并且使子系统能够更容易被使用。因为我们确切知道在另外一个系统中我们想用的功能，所以我们能够创建一个外观，方便并有效地访问那些特性，同时把其他的特性隐藏起来。外观不改变所包裹系统的模型，它应该严格按照另一个系统的模型来编写。否则，您顶多是把转换的职责分配给多个对象，但这会加重外观的负担，而且最坏的情况是创建出另外一种不属于现在这两个系统限界上下文的模型。外观在另一个系统的限界上下文中。它只不过是专门为了满足您的需要提供更加友好的界面而已。

适配器是允许客户使用不同协议的一个包装，并非如我们理解的那样是行为的实施者。



当客户发送一个消息给适配器时，这个消息被转换成语义对等的消息，然后发送给“被适配者(adaptee)”。响应在被转换后传回来。我现在所用的“适配器”这个术语有点不确切，因为Gamma et al. 1995强调，适配器是一个包装对象，它遵从客户期望的标准接口，然而我们却选择“适配的接口”，而且“被适配者”甚至可能不是一个对象。虽然我们的重点是两个模型之间的转换，但是我认为这与适配器的目的是一致的。

对于我们定义的每种服务，需要一个适配器来支持服务接口，并了解如何产生针对其他系统或系统外观的等价请求。

接下来将介绍转换器。适配器的工作是要懂得如何产生请求。概念对象或数据的实际转换是一个明确而复杂的任务，这种任务可以放在它自己的对象中，使数据和转换的数据能够更容易被理解。在需要的时候，转换器可以是一个轻量级对象，在需要的时候实例化。因为它属于它服务的适配器，所以不需要任何状态并且不需要被分配。

这些就是我用来创建防腐层的几个基本要素。还有几个需要考虑的其他事项。

- 一般情况下，如图 14-8 所示，由正在设计的系统(您的子系统)发起动作。不过，也有这样的情况，即另一个子系统可能需要向您的子系统发出请求或者通报它的一些事件。防腐层可以是双向的，如果用它们自己的适配器在两边的接口上定义服务，则可能会使用同一个对称转换的转换器。尽管防腐层的实现通常不需要对另一个子系统进行任何改变，但是为了使那个系统能访问防腐层的服务，做一点改变是必要的。

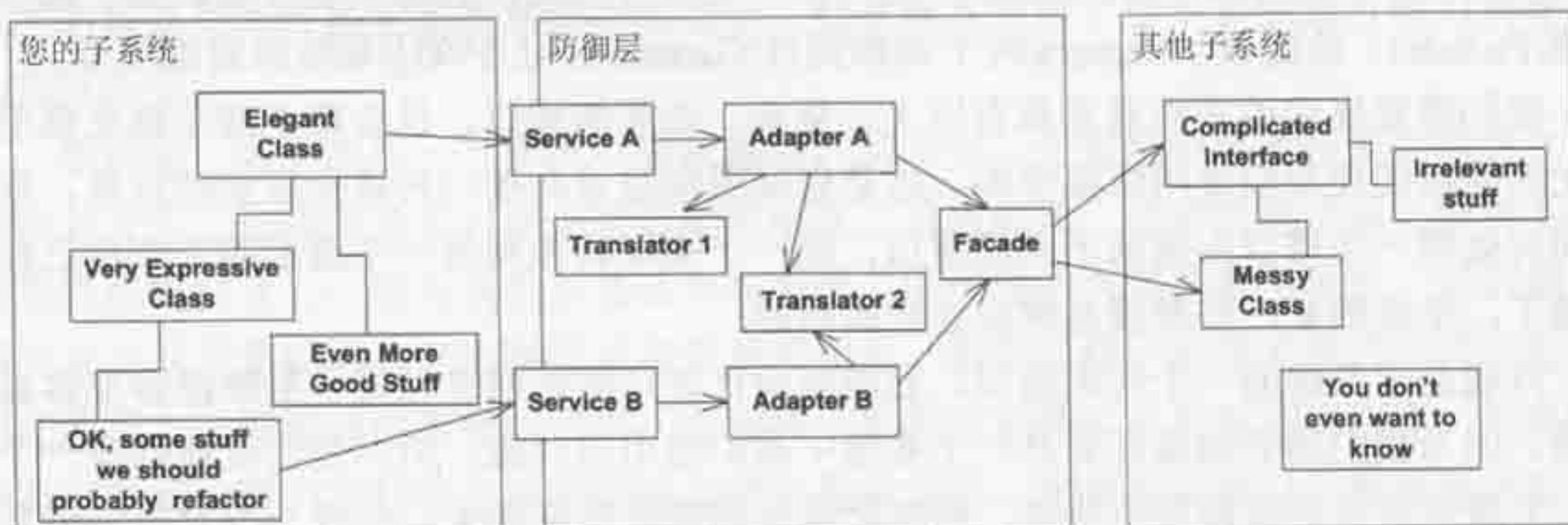


图 14-8 防腐层的结构

- 通常需要一些通信机制来连接两个子系统，它们可以位于不同的服务器上。在这种情况下，必须决定在哪里放置这些通信链接。如果根本无法访问到另一个子系统，可能需要在外观和那个子系统之间安放链接。可是，如果外观可以与那个子系统直接集成，那么把通信链接放在适配器和外观之间会是一种好的选择，因为

外观的协议通常要比外观所掩盖的协议更加简单。也有这样的情况，如果整个防腐层能够与那个子系统结合在一起，那么可以在您的子系统和构成防腐层接口的服务之间放置通信链接或者分配机制。这些实现和部署的决策很武断，它们与防腐层的概念角色无关。

- 如果有权使用另一个子系统，那么会发现一些重构能够使您的工作变得更加轻松，特别是设法为正在使用的功能编写更加显式的接口。如果可能的话，从自动化测试开始编写。
- 如果需要进行大量集成，会大大增加转换的代价。为了使转换更加容易实现，在模型的设计中应该选择更靠近外部系统的地方进行转换。这样做要非常小心，注意不能破坏了模型的完整性。当有些转换不能解决时，可以有选择地进行转换。如果这种方法看起来是解决问题最重要部分的最自然的方案，那么就考虑让您的子系统使用同流者模式，从而消除转换。
- 如果另外那个子系统比较简单或者有一个干净的接口，可以不需要外观。
- 如果功能对两个子系统的关系是特定的，就可以把它添加到防腐层中。要记住，跟踪外部系统的使用情况或者跟踪对其他接口调用的调试，是两种有用的特性。

记住，防腐层是一种连接两个限界上下文的方法。一般对于别人创建的系统，我们不可能完全理解，而且对它很少有控制能力。但是，这不是惟一需要在子系统之间补充内容的情况。如果您自己设计的两个子系统基于不同的模型，也可以使用防腐层来连接，可能是使用一个简单的转换层来完全控制两边。可是，如果两个限界上下文采用隔离方式模式，那么仍然有一些功能集成的需要，但采用防腐层可以减少它们之间的摩擦。

### 示例：遗留的预订应用程序

为了得到短小、快速的第一个发布，我们将编写一个能够建立装运的最小应用，然后通过一个转换层把它传递给老式系统进行预订和支持的操作。因为我们创建的转换层是为了保护我们开发的模型不会受到遗留设计的影响，所以这种转换就是一个防腐层。

最初，防腐层将接受表示装运的对象，对它们进行转换并传递给老式系统，请求一个预订，然后捕获确认信息并把它转换成新设计的确认对象。虽然我们必须在转换方面投入相当大的精力，但是这种隔离允许我们几乎完全独立于旧系统开发自己的新应用。

在后续发布中，新系统可以替换掉老式系统的更多功能，或者只是简单地添加新价值，不管怎么做都取决于后面的决定。这种灵活性，以及当系统逐步转变时能够不断操作结合系统的能力，都说明创建防腐层的价值所在。



### 14.8.3 一个关于警戒的故事

为了保护边境不被周边游牧民族侵犯，古代中国人修建了长城。长城并非是一个难以突破的屏障，只不过它的存在控制了同邻国之间的交往，对别国的入侵和有害影响设置了障碍。两千年来，中国的农业文明在长城的保护下免受了来自外界的干扰。

如果中国没有长城，可能不会形成如此独特的文明，但是修建长城耗费巨大，至少导致了一个朝代因此而衰亡。隔离策略带来的优点必须要同它们的成本平衡。要讲求实际的功用，对模型进行有规则的修改，这样就能够更加自如地与外界交流。

从单个限界上下文内的完全持续集成，经过较少限制的共享内核或顾客/供应商开发团队，到单方面的同流者以及防御状态的防腐层，在任何集成里都有基本开销。虽然集成可能很有价值，但是代价却非常高，所以我们应该确定集成是否真正需要。

## 14.9 隔离方式



我们必须严格地确定需求范围。两套没有必然联系的功能集完全可以脱离对方而独立存在。

**集成始终是昂贵的，而且有时收效甚微。**

除了协调两个团队的正常开销外，集成还迫使我们在设计时作出某些让步：本来使用一个简单专用的模型就能够满足特殊的需要，但是为了集成却必须选择能够处理所有情况的更抽象模型来代替；虽然一些完全不同的技术可以很容易地提供某些特性，但是它们却难以集成；也许某个开发团队很难进行交流，虽然与它们合作很难办好任何一件事情，但集成却要求团队之间进行合作。



在很多情况下，集成并不会带来很大的好处。如果两个功能部件不互相调用对方的功能，不要求双方接触的对象之间有交互，不要求在它们的操作中共享数据，那么不要说集成了，即使是转换层都可能是没有必要的。不能只是因为特性在一个用例中有联系就一定要集成它们。

因此：

声明一个与其他上下文根本没有任何连接的限界上下文，让开发人员在这个小范围内找出简单、专门的解决方案。

这种特性仍然可以组织在中间件或者 UI 层中，但是不存在逻辑共享，并且通过转换层的数据传输绝对最少，甚至没有。

### 示例：一个保险项目的精简过程

一个项目开发团队准备为保险理赔开发新软件，打算将客户服务代理或者理赔人所需的东西全都集成到一个系统中。通过一年的努力之后，团队成员进退维谷。分析瘫痪以及前期大量基础结构投入已经显示出他们日益急躁的开发方式。更严重的是，他们尝试界定的范围使他们陷入了窘境。

新来的项目经理者把所有人都关在一间房子里，要求他们一星期内制定出一个新的计划。刚开始，他们列出需求清单，并试着去估计它们的难点及其重要性。他们无情地砍掉那些有困难和不重要的部分，然后安排清单上剩下部分的顺序。那个礼拜，他们在那间房子里作出了很多巧妙的决策，但是到最后表明只有一个重要的。在某种程度上，他们认为集成不会给某些特性带来任何附加价值。例如，理赔人需要访问一些现有数据库，并且当前对它们的访问很不方便，尽管用户需要使用该数据，但这个软件系统的其他特性都不会用到它。

团队成员建议了各种方式来使访问变得容易些。一种情况是，可以把一份关键报告用 HTML 形式导出并放到内部网上。另一种情况是，可以向理赔人提供一个专门用标准软件包写成的查询。所有这些功能可以通过组织内部网页面上的链接，或者在用户的桌面上放置按钮来集成。

团队开发了一系列小项目，当时只是把它们摆在同一个菜单上，没有考虑更多的集成。几个有用的功能几乎在一夜之间就完成了。抛弃额外特性的包袱，留下精练的需求集，暂时看起来，有希望完成主要应用的移交。

本来可以照这样的方式进行下去，但不幸得很，团队又犯了老毛病，使开发重新陷入瘫痪。最后，他们只留下那些以自己的隔离方式开发的小应用。

采用隔离方式需要进行一些预先处理。虽然持续的重构最终能够撤销任何先前所作



的决定，但是却很难合并完全独立开发的模型。如果最终需要集成的话，转换层是必要的而且也可能会比较复杂。当然，无论如何这都是您将要面对的。

好了，回到更多的合作关系上来，让我们考虑提高集成度的方法。

## 14.10 开放主机服务

一般而言，对于每个限界上下文，您都要给上下文外的每个组件定义一个必须集成的转换层。这里的集成是一次性的，这种为每个外部系统插入转换层的方法，能够以最小的代价避免破坏模型。但是当您发现您的子系统需要大量集成时，就可能需要寻求另外一种更加灵活的方法了。

当子系统必须与大量的其他系统集成时，为每个系统定制一个转换器可能会让团队陷入困境。转换器越多，那么在作出改变时，需要担心的事情也越多。

团队可能正在反复地做着相同的事情。无论子系统存在什么样的一致性，或许都可以把它作为一组服务来描述，因为这些服务包含了其他子系统的共同需要。

很难设计出一种能被多个团队完全正确理解和使用的协议，所以，如果子系统需要多次集成，那么只有当子系统的资源可以用一个内聚的服务集来描述时，使用协议才会有效。在这种情况下，它可以区分模型维护和持续发展之间的差别。

因此：

定义一个协议，把您的子系统作为一组服务来使用。开放这个协议，让所有需要与您集成的人都能使用该协议。除非当一个团队有特殊需要时，否则，应该改进和扩充协议来满足新的集成需求。而为那种特殊情况使用一次性转换器来扩充该协议，从而可以保持公用协议的简单性和一致性。

这种形式化的通信包含了一些共享的模型词汇，即服务接口的基础。结果，其他的子系统都被关联到开放主机(Open Host)的模型中，并且其他的开发团队都被迫去学习主体团队使用的专门用语。所以，在一些情况下，使用众所周知的公布语言(Published Language)作为交换模型可以减少关联，而且也易于理解。

## 14.11 公布语言

两个限界上下文的模型之间进行转换需要一种共同语言。

当两个领域模型必须共存，而且必须在它们之间传递信息时，转换过程自身很可能变得很复杂并难以证明和理解。如果我们正在创建一个新系统，普遍都会认为新模型



是目前最好的，于是就会把旧系统直接转换成该模型来思考。但有时却需要改进一些旧系统，并尝试集成它们。此时就应该选择它们当中结构最差的模型来修改，这样就可以避免因丢失更多的信息而带来的麻烦。

另外一种情况：在各种业务之间相互交换信息，它们会怎样做呢？期望其中一个采用另一个的领域模型不但是不切实际的，而且也是不符合双方要求的。开发领域模型的目的是为了给它的用户解决问题，但这种模型可能包含了不必要的复杂通信。同样，如果某个应用的模型被用作通信媒介，它就不可能为了满足新的需求而被随意地改变，但它必须非常稳定地支持正在进行的通信任务。

直接转换现有的系统模型可能不是一个好办法。这些模型可能非常复杂也可能构造极差，或许它们的可行性还没有被证明。如果把其中一个拿来作为数据交换的语言，实际上是行不通的，而且也不能响应新的开发需要。

开放主机服务使用一种标准协议来实现多方集成。它使用领域中的一个模型在系统间进行交换，即使该模型可能还没有被那些系统所利用。我们在这里深入了解并公布那种模型所使用的语言，或者找出一种已经公布的语言。我说的“公布”就是简单地指这是一种容易获得的语言，对它感兴趣的人们会用它来进行讨论交流；并且这种语言已经被充分地证明，对它的独立解释来说都是一致的。

最近，在电子商务领域里出现了一个令人兴奋的新技术：可扩展标记语言(XML)。这种新技术让数据交换变得更加容易。XML 的一个非常有价值的特点是，通过文档类型定义(DTD)或者 XML 模式，XML 允许对专门领域的语言进行正式定义，数据能被转换成该领域的语言。为了给工厂制定出一个标准的 DTD，现在已经开始形成了相应的行业团体。由此可以说，化学公式或者遗传代码可以在许多团体之间沟通。实际上，这些团体正在用语言定义的形式创建一种共享的领域模型。

因此：

使用一种充分证明了的共享语言作为共同的通信方法，通过对该语言进行必要的转换，就能够表达出必要的领域信息。

该语言不需要从头开始创建。许多年前，一个公司要我用 Smalltalk 做一个软件产品，数据用 DB2 保存。而该公司想让没有 DB2 许可的用户也能使用这个软件，就让我构建一个 Btrieve 的接口，这是一种免费的轻量级数据库引擎。Btrieve 不完全是关系型的，但是我的客户端程序只用到 DB2 的一小部分功能。因此，我决定将这个作为我的 Btrieve 组件的接口。

该方法确实行之有效。这个软件与我的客户端系统平稳集成。可是由于缺乏一种正式的规范或文档来提取客户端设计中的持久对象，所以计算新组件需要的许多工作都得



由我来完成。并且，把其他一些应用从 DB2 移植到 Btrieve，不会有太多重用组件的机会。新软件对公司固有模型保护得越深，那么重新构造模型的持久对象就会变得越困难。

一种较好的方式是，确定公司所用 DB2 接口的子集，然后提供必要的支持。DB2 接口由 SQL 和一些私有协议构成。尽管接口很复杂，但是必须要对其进行严格的定义和详尽的说明。因为只用到接口的一小部分，所以减轻了它的复杂程度。如果已开发一个组件仿效 DB2 接口上必要的子集，那么开发人员只需要简单地确认该子集，这种方法已经得到了有效的证明。正如前面所作的改进一样，将来对持久层的重新设计只会受到所用 DB2 子集的限制。

DB2 接口就是公布语言的一个例子。在这种情况下，两种模型都不属业务领域，但是所有应用的原则确是一模一样的。因为合作的模型中有一个已经是公布语言了，所以就没有必要引入第三方的语言。

### 示例：化学的公布语言

在工业和学术界里，有着数不清的程序对化学公式分类、分析和处理。因为几乎每个程序都用一种不同的领域模型来表示化学结构，所以数据交换总是比较困难的。大多数程序是用某种语言编写而成的，例如 FORTRAN，无论如何都不能充分表示出领域模型。如果有人想共享数据，就必须得先弄清楚其他系统的数据库，然后再制定出相应的转换方案。

化学标记语言(CML)是 XML 的一种专用语言，作为这个领域的共同交换语言，它由学术界和工业界代表组成的一个小组进行开发和管理(Murray-Rust et al. 1995)。

化学的信息不仅复杂多变，而且总是随着新的发现而改变。于是他们开发了一种能够描述化学基本内容的语言，例如有机分子和无机分子的化学公式、蛋白质序列、光谱或者物理量。

语言被公布后就可以开发工具了，如果它们只对一个数据库有用，那么在此之前根本不需要去考虑工具的编写。例如，一个称作 JUMBO Browser 的 Java 应用程序，对保存在 CML 中的化学结构创建图形化视图。所以，如果您的数据以 CML 格式输入，就可以使用这种可视化的工具了。

实际上通过使用 XML，CML 就可以获得双重优势，即一种“公布的元语(published meta-language)”。人们对 XML 的熟悉程度缩短了 CML 的学习时间；各种现成的工具简化了实现，例如解析器；许多介绍处理 XML 的书籍可以帮助我们编写文件。

这里有一个 CML 的小例子。对于像我这样的非专业人士来说，只是对它有个模糊的理解就可以了，但对它的使用原则还是很清楚的。

```

<CML.ARR ID="array3" EL-TYPE=FLOAT NAME="ATOMIC ORBITAL ELECTRON
POPULATIONS" SIZE=30 GLO-ENT=CML.THE.AOEPOPS>
  1.17947  0.95091  0.97175  1.00000  1.17947  0.95090  0.97174  1.00000
  1.17946  0.98215  0.94049  1.00000  1.17946  0.95091  0.97174  1.00000
  1.17946  0.95091  0.97174  1.00000  1.17946  0.98215  0.94049  1.00000
  0.89789  0.89790  0.89789  0.89789  0.89790  0.89788
</CML.ARR>

```

## 14.12 盲人摸象

有 6 个印度人，  
很想知道大象是什么样子。  
(尽管他们都是瞎子)  
他们都想通过各自的“观察”，  
来达到认识大象的愿望。  
第 1 个人走近大象，  
碰巧撞到大象宽阔而强健的身体。  
立马大叫：“我的天哪！大象就像一堵墙！”

.....  
第 3 个人靠近大象，  
碰巧抓住扭动的象鼻，  
大声说到：“我知道了，大象就像一条蛇。”  
第 4 个人急忙伸出他的手去摸大象，  
结果，摸到了大象的腿。  
“很明显，这个神奇的动物像一棵树！”

.....  
第 6 个人迫不及待，  
去摸大象，他抓住了大象摆动的尾巴。  
“我知道了，大象就像一根绳子！”  
这几个印度人，  
不停地大声争论着。



每人都非常固执地坚持自己的看法，  
尽管每个人所说的那部分都是正确的，  
但是从整体上来说都错了！

——摘自 John Godfrey Saxe(1816~1887)编写的 *The Blind Men and the Elephant*

即使他们不能对大象的本质达成完全共识，每个盲人仍然可以通过他们所触摸大象的身体部分，对其作进一步的认识。如果不需要作任何的集成，那么模型不统一也是没有什么关系的。但是如果需要进行一些集成时，实际上他们可以不需要对“大象”是什么达成一致的意见，但他们可以从这些不同的意见中获得更多有用的东西。这样，至少在不知不觉中达到了交换想法的目的。

图 14-9 是盲人对大象形成的认识的 UML 表示图。各自的限界上下文已经建立好，对他们来说，情况很明确，就是如何交流他们共同关心的方面，这里可能只有一个：大象所在的位置。

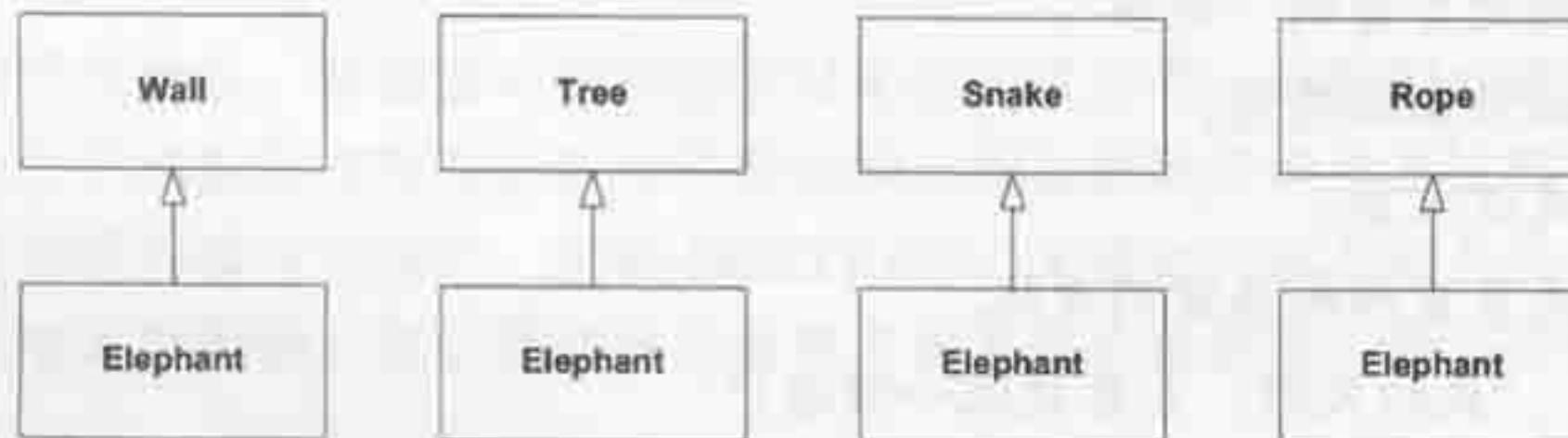
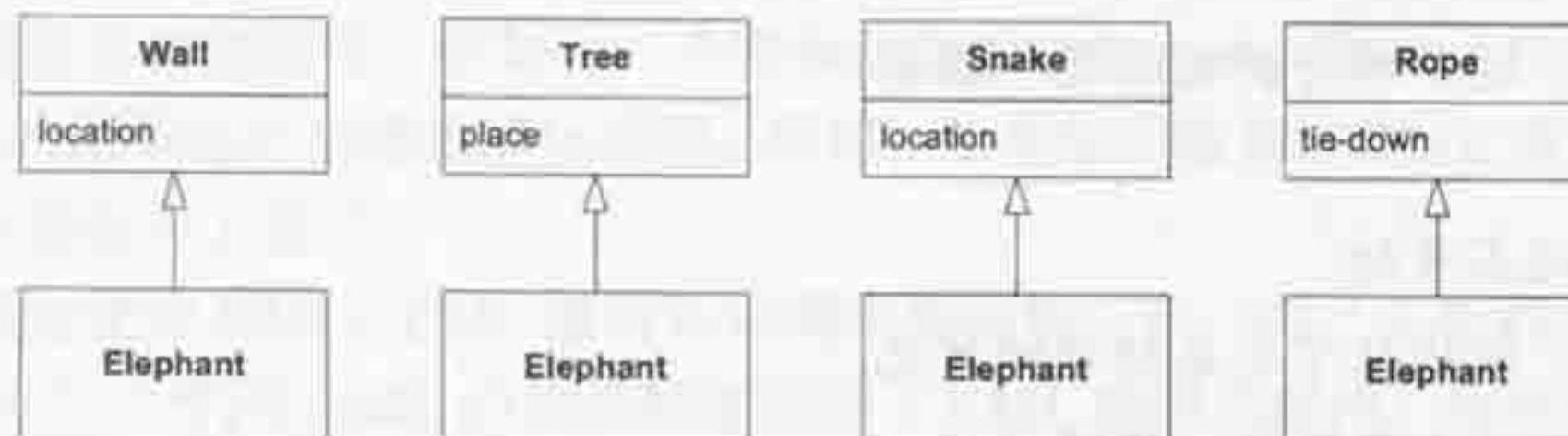


图 14-9 4个上下文：无集成

盲人们想要分享大象的更多信息时，就需要共享一个限界上下文，如图 14-10 所示。但是如何统一不同的模型是个难题，没有人会放弃自己的模型而采用别人的。毕竟，摸到尾巴的人知道大象不会像一棵树，因此，那个模型对他来说根本没有意义，换句话来说基本上就是无用的。统一多个模型几乎总是意味着创建一种新的模型。



Translations:{Wall.location↔Tree.place↔Snake.location↔Rope.tie-down}

图 14-10 四个上下文：最小集成



通过一些想象和不断讨论(可能很激烈)，盲人们最终会承认他们描述和建模的东西只是一个庞大整体中不同的小部分而已。出于多种目的，部分和整体的统一可能不需要太多的附加工作。至少集成的第一步只需要指出各部分是如何相连的就可以了。对于某些需要，确实可以把大象看作是一种这样的构成：一堵墙，下面由树干支撑着，一头是绳子，另一头是条蛇，如图 14-11 所示。

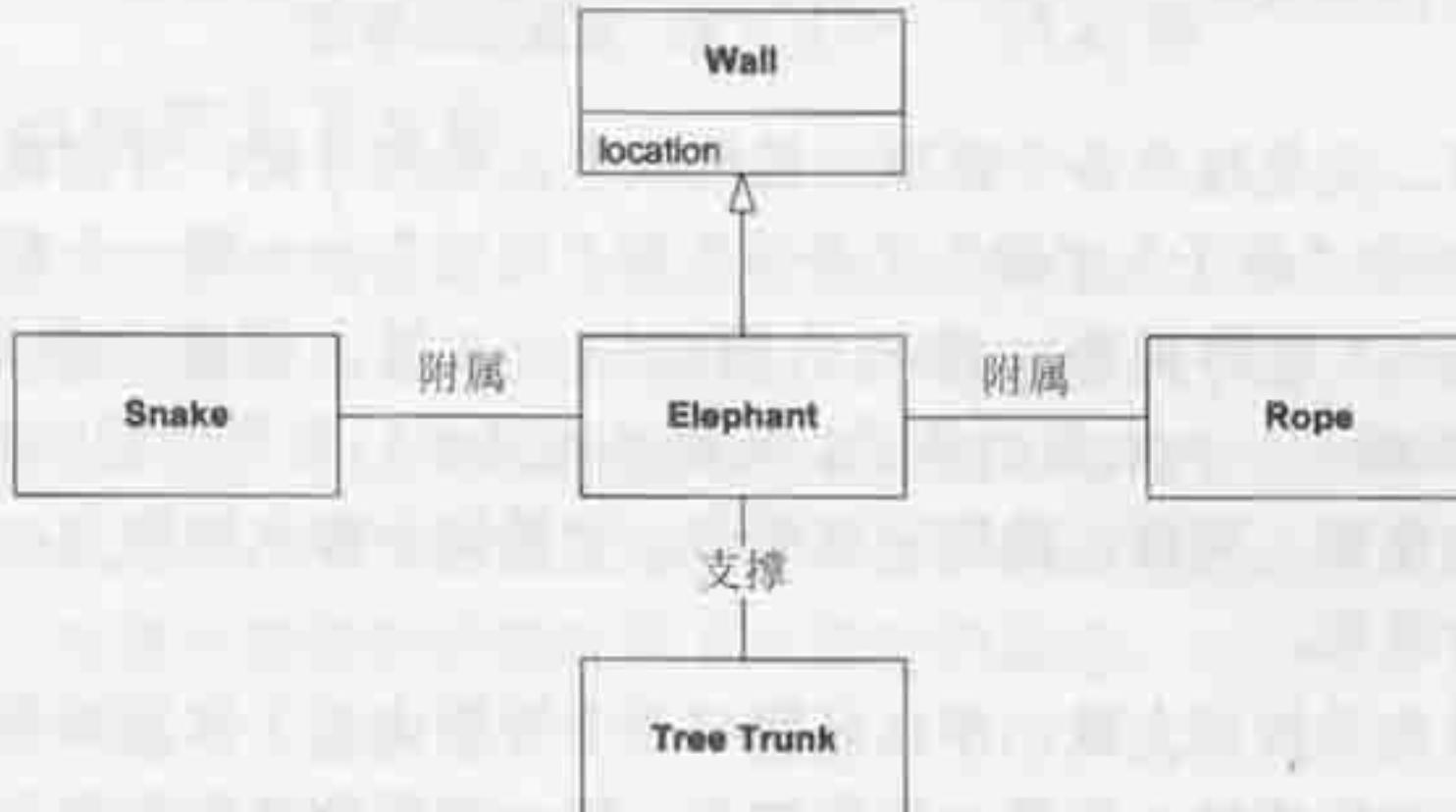


图 14-11 一个上下文：简单的集成

各种大象模型的统一比这种合并要来得简单的多。不幸地是，当两个模型完全是在描述整体的不同部分时就会成为一种例外，尽管这经常是差别的一个方面。但当两个模型以不同方式考虑同一部分时，统一的问题就变得更加困难了。如果有两个人都摸到了大象的鼻子，其中一个人认为它像一条蛇，而另一个人它当作是一条灭火水龙带的话，那么他们就会变得更加难以统一了。他们都不愿意接受对方的模型，因为那都与自己的体验发生了冲突。实际上，他们需要把“活的”蛇和喷水的水龙带合并起来得出一个新的抽象，并且还要忽略初始模型中不适当的性质，例如可能有毒的牙，或可以从车上分离并能够卷起来放进消防车厢里。

即使我们已经把各个部分组合成了一个整体，最后得到的模型还是很粗糙，与实际的东西一点都不相干，对大象的轮廓也没有任何的感觉。在持续精练的过程中，新的见识能够带来更成熟的模型，新的应用需求也可以产生更深层的模型。如果大象开始动了，那它像一棵“树”的说法就不成立了，我们的盲人建模者就能获得突破，得出“腿”的概念，如图 14-12 所示。

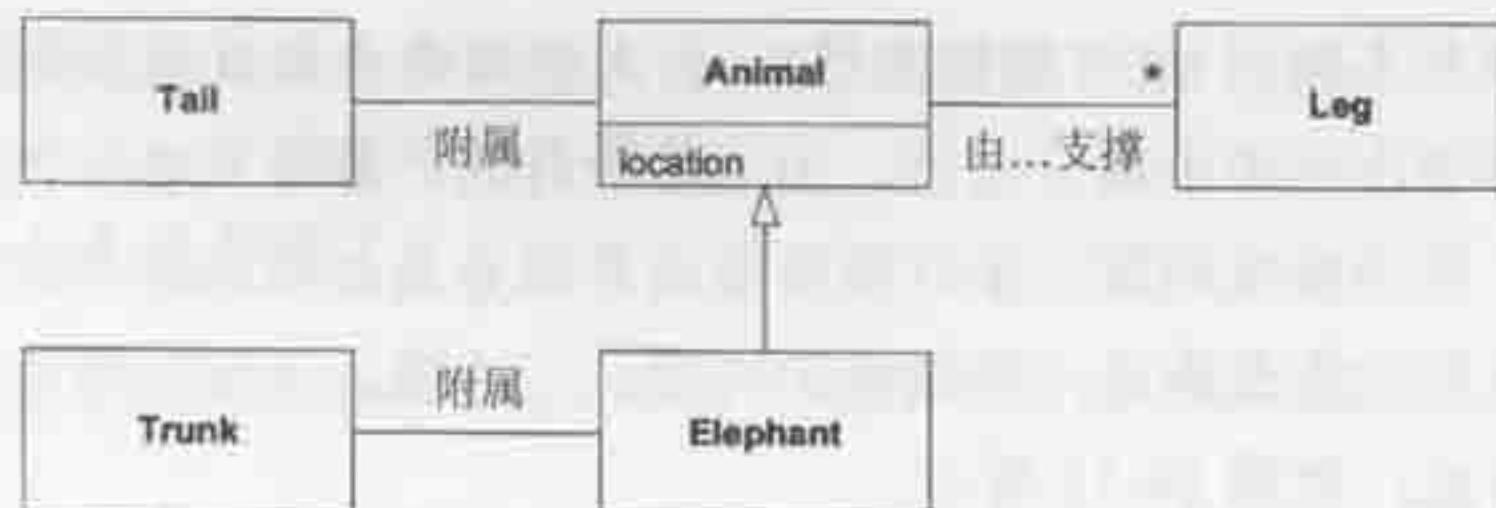


图 14-12 一个上下文：更深层的模型

模型集成的第二关是抛弃各个模型中偶然或不正确的方面，并创建新的概念。既然如此，“动物”应该有“鼻子”、“腿”、“身体”和“尾巴”——每一个都应该有自己的特点以及与其他部分清楚的关系。成功的模型统一，很大程度上都是以“极简主义(minimalism)”为准绳。一头大象的鼻子，可能比蛇还长，也有可能比蛇短，但是“短”可能比“长”更加重要。同样，我们应该明白，宁愿缺少喷水的能力，也不要将毒牙这个不正确特性包含进来。

如果目标只是为了找到大象，那么只要对各个模型中关于位置的表达进行转换就可以了。当需要更多的集成时，在第一次集成后，统一的模型并没有完全成熟。对于某种需要，完全可以把大象看作是由一堵墙，下面由树干支撑着，一头是绳子，另一头是蛇构成的。后面，通过新需求、更好的理解以及交流进行驱动，从而使模型得到深化和精炼。

承认多个有冲突的领域模型，是真正地面对现实。通过明确定义每个模型应用的上下文，您可以维护每一个模型的完整性，并且能够清楚地看出任何特定接口的含义。没有办法能让盲人看见整头大象，但只要他们认识到他们对大象的理解是不全面的，那么他们的问题就是可控制的。

## 14.13 选择模型上下文的策略

无论任何时候，画出上下文映射来反映当前的状态总是十分重要的。如果上下文映射反映出了当前的状态，根据实际情况，您很可能会非常想改变当前的状态。现在您可以有意识地选择上下文的边界及其关系。这里有一些指导方针。

### 14.13.1 团队或更高层的决策

首先，团队必须对在哪里定义限界上下文以及它们之间有什么样的关系都要作出决策。这些决策必须由团队作出，或者至少要保证团队的每一个人都被通知到并且能够理



解决策的全部意图。事实上，像这样的决策通常要在自己的团队内达成一致的意见。有可能会在独立的团队作用和直接、充分的集成之间权衡，然后作出是否扩展或者分割限界上下文的决策。在实践中，团队之间的地位关系通常决定了系统应该如何集成。由于团队的等级结构，可能做不到技术上的有利统一，管理层可能会要求作出不实用的合并。您不可能总会获得您想要的东西，但至少可以估计出因此而需要承担的代价，并设法减轻这些开销，从一个实际的上下文映射开始分析，同时在选择转换时更要注重实效。

### 14.13.2 把自己放在上下文中

当我们在开发一个软件项目时，主要是关注系统中我们团队正在改变的那些部分内容，其次才会关注那些需要与其通信的系统。一个典型的例子是，打算把设计系统划分成一个或两个限界上下文，由主开发团队来负责开发，可能让另外的一两个上下文来担当辅助的角色。除此之外，剩下的就是这些上下文和外部系统之间的关系了。这是一个简单的典型视图，对您可能碰到的情况作了大致性的描述。

我们只不过是在这个主要的上下文的一部分中工作，这在我们的上下文映射中能够反映出来。如果我们意识到偏好的存在，并且在我们离开了图的适用范围时能意识到，那么就不会有什么问题了。

### 14.13.3 转换边界

在划定限界上下文的边界时，会有无数种变化情况和无数种选择。但通常是对下面各个方面进行权衡：

#### 选择较大的限界上下文

- 当用一个统一模型来处理更多的用户任务时，它们之间的流动更加平稳。
- 一个内聚的模型比两个模型加上它们的映射更容易理解。
- 两个模型之间的转换可能很难(有时是不可能的)。
- 共享术语使团队之间的交流更清楚。

#### 选择较小的限界上下文

- 开发人员之间的交流开销降低。
- 规模小的团队和代码库使持续集成更加容易。
- 更大的上下文可能要求更加通用的抽象模型，所需要的技巧很少有人能掌握。
- 不同的模型可以满足特别的需要，或者包括特殊用户群的行话以及通用语言的专门术语。



在两个不同限界上下文之间进行功能的深度集成是不切实际的。集成被限制在模型中那些已经严格按照另一种模型规定的部分进行，而且这种集成能够起到相当大的作用。当两个系统之间存在小接口时进行深度集成才会有意义。

#### 14.13.4 接受我们不能改变的东西：描绘外部系统

最好从最容易的决策开始入手。一些子系统明显不属于在开发的系统的任何限界上下文。例如，您不可能马上替换掉的旧系统和提供您需要服务的外部系统。您可以马上确定这些系统，然后准备把它们从您的设计当中分离出去。

这里要注意我们所作的假设。虽然对这些系统都建立起自己的限界上下文，想起来可能应该很方便，但是大多数外部系统是很难满足定义的。最重要的是，限界上下文是试图在某些边界内统一模型。您可能负责管理老式系统的维护工作，在这种情况下，您可以声明这种目的，要不然就协助原来的开发团队进行非正式的持续集成，但是不要想当然地认为对每一个系统都建立一个限界上下文。检查一下，如果开发不能很好地被集成，就要特别小心了。在这样的系统的不同部分中，发现语义矛盾并不稀奇。

#### 14.13.5 与外部系统的关系

在这里可以应用 3 个模式。首先，考虑隔离方式。是的，如果真的不需要集成，就不要把它们包括进来。但是让用户轻松地使用这两个系统就足够了吗？集成的代价不仅昂贵而且还分散了开发的注意力，因此要尽可能多地解除项目中的负担。

如果集成真的很重要，可以在两个极端之间选择：同流者或者防腐层。作为一名追随者并不好玩。您的创造力和选择新功能的权利将受到限制。创建一个主要的新系统，按照遗留或者外部系统来设计未必管用(否则为什么要创建一个新系统？)。可是，对旧系统进行扩充，当然它仍然还是主要系统，在这种情况下，采用原有模型可能比较合适。这样的例子包括经常用 Excel 编写的轻量级决策支持工具或者其他的一些简单工具。如果您的应用真的是现有系统的扩展，并且与那个系统的接口增大了，那么，上下文之间的转换工作比实现应用程序本身的功能将还要大。即使您已经把自己放到另外那个系统的限界上下文中，仍然可以采用一些优秀的设计。如果在另外那个系统中有一个可辨别的领域模型，那么，只要严格遵照这个老模型，使得该模型更加清楚，就可以改进您的实现。如果决定采用同流者模式来设计的话，就必须尽心尽力地使用老模式。这样，把自己限定在只对现有的模型进行扩充，而不是修改它。

如果设计的系统与另外那个系统的接口较小，或者那个系统设计得很糟糕，那么当实现它的功能要比扩充现有系统更加棘手时，就需要有自己的限界上下文，这意味着需



要创建一个转换层，甚至防腐层。

#### 14.13.6 在设计系统

您的项目团队实际上开发的软件是在设计系统(system under design)。您可以在这个范围内确定限界上下文，并在每个上下文中应用持续集成，从而保持它们统一。但是需要确定多少个限界上下文呢？它们彼此有什么样的关系呢？因为在这个范围内，我们有更多的自由开发和管理的能力，所以比起使用外部系统的情况来说，答案就变得更加不确定。

解决的方法应该非常简单：为整个在设计系统创建单个限界上下文。例如，团队成员应少于 10 个人，而且他们开发的功能都紧密相关。

如果开发团队变大，持续集成可能会变得困难(尽管我曾经看到过在比较大的团队中持续集成)。您可能会考虑采用共享内核，把相对独立的功能集划分到不同的限界上下文中，每个上下文环境的开发人员都不超过 10 个人。如果这两个上下文中的所有依赖关系都是单向的，那么您就可以采用顾客/供应商开发团队。

您可能会认识到由于两个开发小组的思路相差太大，以至于他们的建模努力经常会发生冲突。可能是由于他们的实际需要差别很大，也可能是因为不同的背景知识，再或者可能是由于项目中嵌入的管理结构，这些可能性都将会造成了冲突。如果不能改变造成冲突的原因，或者不想改变时，可以让模型选择隔离方式。可以在需要集成的地方开发一个转换层，作为持续集成的单个点，由两个团队共同维护。同集成外部系统作对比，不管有没有另外一边的大力支持，防腐层通常必须要适应另外一边的那个系统。

一般而言，每个限界上下文对应一个开发团队。虽然一个开发团队可以维护多个限界上下文，但是很难(尽管不是不可能)让多个开发团队工作在一个限界上下文中。

#### 14.13.7 满足不同模型的特别需要

开发相同业务的不同小组经常会使用自己的专门术语，但它们的意思可能互不相同。这些行话可能很精确并且能够符合他们的需要。改变它们(例如，强加一种标准的、企业范围的术语)需要进行广泛训练和分析后才能够解决它们之间的差异问题。尽管那样，新术语可能还是不如原有术语表达的意思精确。

您可能会决定在不同的限界上下文中满足这些特别的需要，除了转换层要持续集成外，允许模型使用隔离方式。通用语言的不同方言会围绕这些模型和模型基于的行话演化。如果两种方言有重叠，那么共享内核模式可以提供必要的特殊化处理，将转换代价减到最小。



如果不需要集成，或者集成相对有限，那么就继续使用惯用术语以免破坏模型。这种做法仍然存在相应的代价和危险。

- 缺少共享术语会减少交流
- 增加集成的费用
- 随着同一个业务活动和实体的不同模型的演化，会有一些重复开发工作。

但可能出现的最大危险是，它可能会变成反对变化的一场争论或者一次为所有不常见、有局限性模型作的一次辩护。究竟需要对系统的这个部分作多少修改才能满足特殊需要呢？最重要的是，这个用户群里使用的特殊行话到底有多少价值呢？因此必须针对转换的风险权衡是否采取更加独立的开发方式，注意合理地处理术语的变化。

有时会出现一种深度模型，能够统一这些不同的术语并且满足双方需要。只有在经历了大量的开发和知识消化后，深度模型才会在生命周期的后面出现。所以您不可能去设计一个深度模型，只能在它出现时，您才能抓住这个机会，改变策略，重新构造。

记住，如果集成的需求很大，那么转换的代价就会攀升。从对有复杂转换的对象进行很小的修改，到对共享内核的修改，尽管不需要完全统一，但团队之间的协调仍可以使转换变得更加容易。

#### 14.13.8 部署

协调打包和复杂系统的部署是一项麻烦的任务，做起来总是比看起来复杂得多。选择限界上下文的策略对部署是有影响的。例如，如果顾客/供应商模式要重新部署开发系统，那么只有在他们互相协调好后，才能发布共同测试过的版本。代码和数据的迁移必须在这些合并中进行。在一个分布式系统里，我们可以在一个过程里保留上下文之间的转换层，这样就不会出现多种版本共存的现象了。

如果数据迁移很耗时，或者分布式系统不能即时更新，会导致两种版本的代码和数据共存，也就是说，即使是部署一个限界上下文里的组件也可能具有挑战性。

根据部署的环境和技术，有许多技术上需要考虑的事项。限界上下文的关系可以把问题指向关注的热点，并且转换接口都已经规划好了。

部署计划的可行性将反馈到上下文界限的绘制上来。当两个上下文由一个转换层连接时，其中一个环境可能会被更新，那么会有一个新的转换层提供相同的接口到另一个上下文。不仅仅在开发阶段，而且在部署阶段，共享内核模式都会更加注重协调性。而隔离方式模式就会简单许多。



### 14.13.9 权衡

总结这些指导方针可以发现，有不少是关于统一模型或集成模型的策略。概括地讲，要把无缝集成的优势与附加的协调和通信代价进行权衡：在选择更多独立行动与更平稳的通信之间进行权衡。进行更大的统一需要控制有关子系统的设计。

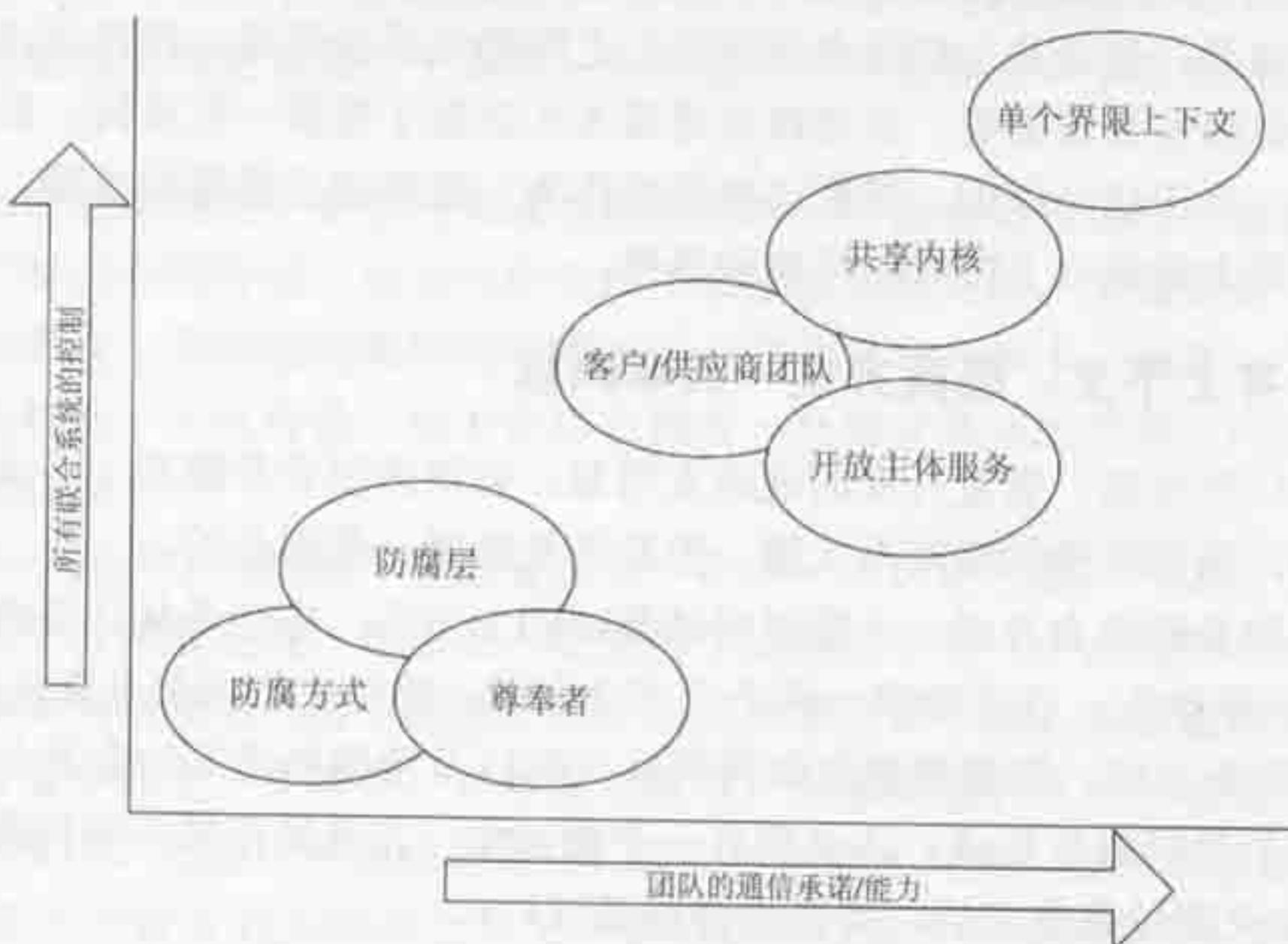


图 14-13 上下文关系模式的相对需求

### 14.13.10 考虑项目已经进行的情况

很有可能，您不是开始去开发一个项目而是考虑改进一个已经在进行的开发项目。在这种情况下，第一步是根据目前的方式来确定限界上下文，这是至关重要的。为了达到有效的目的，上下文映射必须能够反映出团队的真实习惯，而不是反映根据刚才描述的指导方针得到的理想结构图。

一旦描绘出目前存在的限界上下文并且描述了它们目前的关系后，下一步就是围绕着目前的结构，加强团队的开发。在上下文中改进持续集成，把所有杂乱的转换代码重构到防腐层中，为现有的限界上下文命名并且还要保证把它们放入项目的通用语言里。

现在考虑一下边界和关系自身的变化。在开发一个新项目时，我已经描述了一些适用的法则，而这些变化会很自然地由这些相同的法则来驱动，但是这些变化必须被分成很小的部分，并且被合理地选择，从而用最小的代价以及对项目造成的最小破坏来换取最大的好处。

下一节将讨论，决定做出后，如何动手改变上下文边界。

## 14.14 转换

像建模和设计的其他方面一样，关于限界上下文的决定也是不能改变的。不可避免地，在很多情况下，您必须改变对边界和限界上下文关系作出的最初决定。一般而言，划分上下文很容易，但合并它们或者改变它们之间的关系却很难。我将讲述几个典型的变化，它们很麻烦但却很重要。这些转换通常太大以至于需要一次重构，甚至可能是一次项目的迭代。由于这个原因，我把这些转换作为一连串便于管理的步骤。当然，您必须要使这些指导方针适合于特殊的环境和事件。

### 14.14.1 合并上下文：隔离方式→共享内核

这种转换代价太高，重复开发的痕迹太明显。有很多要合并限界上下文的动机，而这种转换很难，虽然转换的时间不太慢，但是还是需要一些耐心的。

即使最终的目标是合并成一个能够持续集成的上下文，也还是从共享内核开始吧。

- (1) 估计初期状况。在开始统一两个上下文之前，确信它们内部是真正统一的。
- (2) 设置转换过程。您需要确定如何共享代码以及模型的命名约定是什么。至少每周对共享内核的代码进行集成。还必须有一个测试集。在开发任何共享代码前，要设置好转换过程(如果测试集是空的，将会很容易通过！)。

(3) 从选择一些小的子领域——也就是说从两个上下文中存在的一些重复部分开始，而不是从核心领域的某个部分开始。第一次合并将建立转换过程，所以最好使用一些简单的和相对普遍或非关键的东西。检查已有的集成和转换。选择一些已转换的东西比从一个已证明的转换开始要好得多，另外这样会减少转换层的任务。

现在，有两个模型应用于相同的子领域。主要有3种合并的方法。可以选择一个模型并且重构另外的那个上下文，并保持它们的一致性。可以从全局的方式来作这个决定，有系统地更换一个上下文的模型，并保持作为单元开发的模型的内聚性。或者可以每次选择一个，最后选择两者最好的那个(但是注意不要以混乱告终)。

第三种选择是找出一种新模型，可能这个要比任意一个原始模型来得更优秀些，并且能够承担双方的职责。

(4) 从两个团队中抽出2~4个开发人员，组织成一个小组，为子域设计出一个共享的模型。不管是如何得到该模型的，都需要有一个详细的设计。这包括了确定同义词和映射所有没有被转换的术语等一些艰巨的任务。这个联合小组概述模型的基本测试集。

(5) 来自两个团队的开发人员承担模型实现的任务(或者把现有的代码修改为共享)，对模型进行详细设计并使其运行起来。如果这些开发人员在该模型上碰到了麻烦，那么



从步骤 3 开始，重新组队并参与任何需要的概念修正。

- (6) 每个团队的开发人员都承担集成新的共享内核的任务。
- (7) 移走那些不再需要的转换。

现在，您已经拥有一个很小的共享内核，而且还有了一个适当的过程来维护它。在后面的项目迭代设计过程中，重复步骤 3~步骤 7。当这些过程稳定下来并且团队有了信心以后，您就能够处理更复杂的子系统，并且能够同时处理多个子系统或者核心领域的子系统了。

注意，当在处理模型中更多的领域专用部分时，可能会碰到这样的情况，两个模型采用不同用户群的专用行话。除非已经发现了深层模型的突破，为您提供一种能够替代两种专用语的语言，否则推迟将这些专用术语合并到共享内核中，这是一个十分明智的做法。共享内核的一个优势是，您不仅可以拥有一些持续集成的优点，同时还能保留一些隔离方式的优点。

这些是合并到共享内核的一些指导方针。在进一步讨论之前，考虑从中选择一个模型，能满足这个转换提出的一些需要。如果两个模型之中有一个直接符合首选的条件，那么考虑不集成而直接采用这个模型。这并不是共享公共子域，而是通过重构应用程序使用那个符合条件的上下文模型，并增强该模型，有系统地把这些子域的全部职责都转移到那个上下文中。无需任何集成的费用，就已经消除了冗余。最终可能会(但不是必要的)完全采用那个比较有利的限界上下文，并且达到与合并相同的效果。转换时(可能很长或者不确定)，必须对隔离方式模式和共享内核模式进行权衡。

#### 14.14.2 合并上下文：共享内核→持续集成

如果共享内核有扩展的趋势，就有可能吸引您对两个限界上下文进行完全的统一，但这不是解决模型差异问题的方法。这样做将会改变团队结构和项目最终使用的语言。

从准备开发人员和团队开始。

- (1) 确保持续集成(共享代码所有者、频繁集成等)的所有过程都已经在各个团队中准备好了。协调两个团队之间的集成，以保证所有人以同样的方式处理事务。
- (2) 成员在团队之间流动。这样做将培养出很多了解双方模型的开发人员，并且开始让两个团队的成员有了联系。
- (3) 阐明各自模型的精华(参见第 15 章)。
- (4) 好了，现在有足够的信心把核心领域合并到共享内核里去了。这可能需要进行几次迭代，有时候在新的共享和尚未共享的部分之间还会需要暂时的转换层。一旦要合并核心领域时，转换越迅速越好。这个阶段不仅代价高而且风险大，转换时间尽可能短，

优先权比大多数最新的开发高，但是不要超出您能处理的范围。

要合并核心模型，有几种选择。可以坚持一个模型并修改另外的模型来与它保持一致，或者创建子域的新模型，然后修改两个上下文去使用它。注意，如果需要修改两个模型来满足不同用户的需要，可能需要有处理原始模型的专门能力，这就要求开发出一个能替代两个原始模型的深层模型。开发深层的统一模型是非常困难的，但如果坚持要完全合并这两个上下文的话，就不再需要使用多种专用语言了。根据最终模型和代码集成的清晰程度，会获得相应的好处，注意，并不是因为您的能力满足了用户的特殊需要。

(5) 当共享内核变大时，使集成的频率达到日集成并最终达到持续集成的目标。

(6) 当共享内核包含了先前两个限界上下文的边界时，您会发现自己拥有一个大的和两个较小的开发团队，他们共享着通过不断集成而形成的代码库，而且经常在团队之间来回交换其开发人员。

#### 14.14.3 逐步淘汰原有系统

所有的东西都会被淘汰，原有的计算机软件也不例外。这些旧系统可能已在商业和其他系统中得到了广泛的应用，那么替换它们的工作可能要花上几年的功夫。但是幸运的是，不需要马上把所有东西都淘汰掉。

可能出现的情况太多，而我在此所作的也只不过是掠其皮毛而已，但我将会讨论一个普遍的情况：最近，一个在商业上每天使用的旧系统添加了几个更先进的新系统，它们与旧系统之间通过防腐层通信。

第一步应该决定测试策略。在新系统中为新添功能编写自动的单元测试，但是淘汰旧系统还需要一些特殊的测试。一些组织在一段时间内会同时使用新老两种系统。

对于迭代过程：

- (1) 在一次迭代中，确定旧系统中的功能，并把它们都添加到合适的新系统中去。
- (2) 确定防腐层需要添加的东西。
- (3) 实现
- (4) 部署

有时需要几次迭代才能写出替代旧系统某个组件的功能，但仍以小的、迭代尺寸的单元来计划新功能，只在部署时再进行多次的迭代。

另一个关键的地方是对模型进行部署。如果这些小的、逐步增加的变化能够大批进行，会给开发带来好处。但通常需要组织成大型的版本，还必须培训用户使用新的软件，有时候还必须要顺利地与开发并行完成，而且还会有许多后勤的问题必须去解决。

部署一旦被决定下来后：



(5) 确定防腐层不需要的部分并去除掉。

(6) 考虑切除旧系统中现在没有用到的模型，尽管这样做可能不切实际。但有意思的是，设计得越好的软件越容易被淘汰。而设计糟糕的软件却很难被除去。所以可以这样做，忽略未被使用的部分，直到剩余部分已经被淘汰后，再停止使用。

不断重复上面的循环。让旧系统逐渐退出其在商业上的应用，从而最终完全抛弃掉老的系统。同时，当结合增加或减少系统之间的相互依赖时，防腐层也同时随之收缩和膨胀。如果其他地方都相同的话，当然首先迁移的那些功能应该使形成的防腐层变得较小才对。但可能会受到其他因素的影响，在一些过渡阶段，您可能必须忍受一些危险的转换。

#### 14.14.4 开放主机服务→公布语言

您可能曾经用过一组特殊协议与其他系统集成，但是使用的系统越多，需要维护的代价就会越高，否则它们之间的相互作用就会难以理解了。这时，需要用公布语言来规范系统之间的关系。

(1) 尽量使用工业标准语言。

(2) 如果没有标准的或已公布的术语，则对系统的核心领域作适当地整理，并把它作为主体(参见第 15 章)。

(3) 把核心领域作为交流语言的基础，如果有可能的话，使用像 XML 那样的标准交换范例。

(4) 发布新术语给所有合作的系统。

(5) 如果涉及到新系统的架构，那么就发布这个架构。

(6) 为每个合作系统建立转换层。

(7) 进行转换。

这时，新加入的合作者给系统带来的破坏已变得很少了。

记住，公布术语必须是稳定的，但当继续重构时，仍需要有改变主体模型的自由。因此，不要把交流的语言和主体模型等同起来。把它们紧密联系在一起只能降低转换的开销，可以让主体选择同流者模式，但是如果加强转换层更加有利于集成时，应该能够从同流者模式中摆脱出来。

项目领导会基于功能的集成需要和开发团队的关系来定义限界上下文。一旦限界上下文和上下文映射被明确定义后，就得保证它们在逻辑上的一致性。至少要提出相关的通信问题，并能够得到妥善地解决。

然而，不管是意识上的还是事实存在的模型上下文，有时候被误用来解决系统里的



问题而非逻辑矛盾。开发团队可能会发现，一个具有大的上下文的模型看起来可能太复杂，以至于不能从总体上获得更多的认识并对其进行彻底的分析。出于选择或者意外，经常会把大型的上下文分成易于处理的小片段，这种片段容易丧失保持一致性的机会。现在，仔细观察在一个很大的上下文中建立一个大型模型的决策是值得的，如果没有可能的结构或策略来维护这个大模型，而这个模型实际上是分割的，那么应该重新绘制和确定能维持的边界。但是如果一个大型限界上下文能够满足必要的集成需要时，除了模型本身的复杂性，它看起来基本上是可行的，那么分解上下文可能并不是最好的选择。

在划分大的上下文之前，应该考虑是否还有其他的方法来处理大型模型。后面两章将介绍两种关于处理大模型复杂性的主要原则：精练原则和大比例结构原则。