

构建面向对象的应用软件系统框架

孙亚民

目录

第一部分	综述	4
第 1 章	本书会讨论什么内容	5
第 2 章	系统的分层结构	8
2.1.	简述	8
2.2.	设计的原则和评判标准	9
2.3.	应用服务层的内容	10
2.4.	数据实体的表示	11
2.5.	数据的存取方式	15
2.6.	业务逻辑的处理	18
2.7.	业务服务的提供	20
2.8.	层的部署和层间交互	21
2.9.	剪裁和取舍	21
2.10.	小结	22
第二部分	应用服务层的设计	23
第 3 章	数据和对象	24
3.1	数据的形态	24
3.2	对象/关系型映射	26
3.3	对象的状态	28
	Transient	29
	Persistent-new	29
	Persistent-dirty	29
	Persistent-clean	30
	Persistent-deleted	30
第 4 章	O/R Mapping 的一般做法	31
第 5 章	设计一个 O/R Mapping 框架	40
5.1	封装数据库访问层	40
5.2	设计映射	48
5.3	对继承的支持	56
5.4	设计对象操纵框架	62
5.5	实现对象操纵框架	67
第 6 章	面向方面编程	73
6.1	AOP 概念	73
6.2	Websharp AOP 的使用	75
6.2.1.	使用 AOP 实现松散耦合	75
6.2.2.	使用 AOP 组合两个业务逻辑	78
6.3	Websharp AOP 的实现	79
6.3.1	AspectObject 抽象类	80
6.3.2	IAspect 接口	81
6.3.3	AspectManagedAttribute	81

6.3.4 定义 AspectProxy 类	82
6.3.5 其他一些辅助类	83
6.3.6 配置文件	83
6.4 关于 AOP 和过滤器	84
6.5 小结	85
第 7 章 接口	86
第 8 章 事务处理	89
8.1 事务的基本概念	89
8.2 实际开发中可用的事务处理方式	91
第 9 章 性能优化	105
第三部分 用户界面层设计	106
第 10 章 界面层的功能划分	107
第 11 章 界面设计模式	108
11.1 MVC 模式	108
11.2 页面控制器	111
第 12 章 动态代码生成和编译技术	112
12.1 Emit	112
12.2 CodeDom	112
第 13 章 远程过程访问的客户端整合	116
Web Service	116
.Net Remoting	117
Websharp Service Locator 的主要接口	119
Websharp Service Locator 的配置文件	119
如何使用 Websharp Service Locator	121
LocalAssemblyLocator 的 Hello World 例子	121
Hello World 的 WebServiceLocator 例子	124
Websharp Service Locator 的实现	125
目前的进展	125
将来的目标	125
小结	126
第 14 章 智能客户端	127
小结	133
第四部分 系统建模过程	134
第 15 章 简述	135
第 16 章 用例模型——系统需求的获取	136
第 17 章 分析模型——开发者的视野	140
第 18 章 系统设计——实现方案	146

第一部分 综述

第1章 本书会讨论什么内容

从软件工程说起。提起这个概念，往往令人想起 CMM、RUP、印度模式等。管理的因素，在软件开发过程中起着非常重要的作用，然而，软件工程并非只指软件开发的管理工作，而是一个范围很广的综合性学科。在软件工程中，大约一半的内容是专业性很强的，涉及到软件分析、设计甚至编码的技术。所谓的结构化、面向对象，都在软件工程的范畴内。“软件工程范围极为广泛。软件工程的某些方面属于数学或计算机科学，其他方面可归入经济学、管理学或心理学中。”

软件业一直在探讨，如何使软件实现如同传统产业一样的大规模生产。软件工程的提出，便是为了实现这个愿望。然而，虽然软件工程至今已经有了很大的发展，软件的大规模工业化生产仍然没有实现。原因何在？

从软件的本质属性来说，软件的复杂性是软件的本质属性，在这个属性没有改变之前，软件便不会实现同传统产业一样的工厂化生产。

从软件生产的介质来说，传统产业生产都是有形的物质产品，人的生产活动都受制于生产资料这些物质介质；然而，软件生产的介质，却是无形的人类的思维。物质资料的生产，受制于物质本身的属性，不容易为人类的思维所左右，并且容易被大量复制，这使得工业化大生成为可能。而人类的思维，却是如此的容易变化，更关键的是不能被复制，甚至同一个人，不同时期思维的复制都不可能，这使得软件这个纯粹依赖人的思维活动的生产实现大规模工业化生产是如此的困难。实际上，不仅仅是软件产业，凡是主要生产介质是人本身的活动产业，都很难实现工业化生产，如咨询、演艺等。

从生产过程来看，对于传统产业来说，产品的设计和生产是分开的。在设计阶段，主要的工作是人的思维，因此，在这个阶段，同软件一样，不是批量生产的。而在生产阶段，主要的对象便是物质资料，并且一切标准已经制定，只需要在流水线上大量复制。对于传统产业来说，设计和生产的界限是如此的明确，并且，生产和设计的比重是如此的悬殊。然而，对于软件产业来说，软件的生产过程便是设计的过程，纯粹的生产过程几乎不存在（或许，光盘的复制算是），这使得软件的生产形态同传统产业必然存在区别。

对于软件的开发过程来说，从业务模型、需求分析、系统架构、系统分析和设计、到最后代码实现，越往前，抽象层次越高，可控性越小，越往后，越接近实际，可控性越大，因此，在软件开发中，核心团队的作用是如此巨大，一个软件产品的成败，核心团队的核心人员的作用在很大程度上是决定性的。对于软件开发来说，如果软件开发要实现工业化生产，必定是从后向前推进，从编码开始。印度模式或许给出了这么一个例子。

因此，我们在软件工程的路上，只是在不断的向工程化的目标迈进，但是，要达到这个目标，可能会花很长的时间。技术上的每一次进步，都使我们向这个目标迈进了一步。在软件工程的发展过程中，技术进步起了非常大，甚至可以说是决定性的作用。随着采用的技术的不同，所采用的管理方法也在不断变化。软件工程技术的很多方面，也是为管理做准备的。优秀的软件开发技术的采用，能够弥补我们在工程化方面的不足，从而使得软件开发更加可

控，软件质量更加有保障。

本书不准备讨论软件工程过程的问题，而只是对软件工程中软件技术的一个方面——系统框架设计，做一些探讨。

现在，很多开发人员都已经意识到这很重要的一点，那就是，在开发一个应用软件系统的时候，一个好的系统框架是非常重要的。从底层开始构建应用程序，是一件吃力不讨好的事情，而没有框架的应用程序，则很难想象会是一个好的应用程序。

除了对于开发的直接帮助，一个好的框架对于公司的知识管理也是非常有意义的。想象一下，我们经常在讨论，现在是一个知识经济的时代，尤其对于软件公司来说，知识（拥有这些知识的员工）就是公司最大的财富。那么，怎么来进行有效的知识管理呢？

首先，应当明确，知识管理，一个重要的目的，就是要把员工对公司最重要的知识沉淀下来。公司的每个员工头脑里都有很多的知识，这些知识对于员工来说是很重要的，但是其重要性同公司并不是完全一致的。某些知识，对于某个员工来说是最重要的，但是对于公司可能并不需要。知识管理需要做的，是把员工对公司最重要的知识累积起来。

其次，知识管理必须有一个载体。如果知识管理没有载体，那么，公司的知识就存在于员工的头脑之中，一旦这个员工离职，那么，知识也就离去了，没有办法沉淀。如果只是把公司做过的项目的文档作为载体，那么，这个载体就过于零碎了。实际上，如果公司有一个统一的框架，那么，这个框架就是一个很好的知识管理的载体。因为，这个框架，必定是集中了公司所有软件项目的共同点的，集中了对于公司最重要的知识的精华，能够为公司所有的项目服务。另外，随着框架的不断被使用，框架本身也会随之升级优化。对于一个新成员的加入，他只要理解掌握了这个框架，就可以很好的融入团队中来；而人员的离去，也已经把自己对公司最重要的知识留在了这个框架中。可以说，在这里，框架承担了一个知识管理平台的作用。一个最好的例子就是微软的 Windows。这是微软所有知识的最集中的平台。

软件，从本质上来说，就是现实世界在计算机中的模拟。在考虑应用软件系统架构的时候，实际上，考虑的问题主要在于：处理什么？怎么处理？如何使用？因此，应用软件系统，需要关注的方面，概括起来，主要包括以下三个大类：

- 1、 处理的对象，也就是数据。
- 2、 处理的方式，也就是我们的系统如何来处理系统的逻辑。
- 3、 如何进行交互，这个交互包括用户（使用者），以及外部系统。

在应用软件系统中，数据是处理的基本对象，程序总是以一定的数据结构来表现数据，并且，在使用面向对象语言开发的系统中，数据总是以类和对象的形式表现出来。另外一方面，数据总是需要存储，对于大部分应用软件系统来说，通常会采用关系型数据库来保存数据。这样，由于数据在程序和数据库中表现格式的不一致，就必然要求在两者之间进行映射。这个映射，在面向对象设计语言和关系型数据库之间，通常称为对象/关系型映射，即 O/R Mapping。

目前，在 O/R Mapping 部分，在 Java 平台下，已经有多种可以选择的方案，例如 J2EE 架构中的 Entity Bean，轻量级的 JDO，以及开源项目的 Hibernate 等，由于微软的 .Net 框架推出时间不长，成熟的 O/R Mapping 框架并不多见。O/R Mapping 框架的选择或者设计是

构建应用软件系统的最基本的工作。本书将讨论构建 O/R Mapping 框架的一些基本理论、概念和方法。

系统的业务逻辑处理，是应用软件系统的核心部分，如何合理的构建业务逻辑、如何提供业务逻辑层的服务，以及表现层如何访问业务逻辑提供的功能，也是应用软件系统需要重点关注的问题。在这个方面，业界已经发展了很多可供选择的范式，如契约式设计、SOA 架构(面向服务的架构)等。这些方法指明了设计的方向，同时也需要我们在实际开发中加以应用。

在业务逻辑确定后，随后而来的问题就是，如何向客户端来提供业务逻辑服务，或者说，客户端如何访问这些服务。在多层应用软件系统中，客户端和业务逻辑在物理上可能存在于不同的机器上，也可能存在于同一台机器，但至少，在逻辑上，是存在于两个不同部分，这就涉及到一个问题：这两个层之间如何进行通信？还会涉及到远程过程调用的问题。

当然，现在我们已经有多种技术来远程过程调用，包括 Webservice、.Net Remoting、Corba、甚至 EJB 等。如此多的实现技术，带来的很大的灵活性，但同时也带来了问题，其中一个就是，有多少种服务端技术，就得有多少种相应的客户端访问技术。甚至，在某些分布式应用系统中，应用逻辑使用不同的技术开发，存在于不同的机器上，有的存在于客户机本机，有的使用 .Net Remoting 开发，存在于局域网内，有的使用因特网上的 Web Service，有的时候，我们希望相同的业务逻辑能够支持不同的客户端。

在这种情况下，我们需要一个一致的服务访问编程模型，以统合不同的服务访问模式，简化系统的开发和部署。一个统一的远程过程调用框架的前景是如此的诱人，以至于每一种方法都试图一统天下，但出于种种原因，最终都没有一家能够做到，最新的 Web Service 就力图做到这一点。实际上，每一种方法的出现，最终都会带来一个副作用，那就是，可供选择的多了，混乱也就又多了一点。在实际的开发过程中，我们也需要一个统一的访问方式来解决这个问题。本书将讨论一些可用的方案。

为了更加清晰的进行表述，文章会附加一些程序代码。因为在讲到具体的技术的时候，本书会对各种可用的技术进行比较，因此，本书的代码可能会使用不同的语言，通常是 Java 和 C#，不过，在给出代码的时候，一般都会指明所用的语言。在大部分情况下，如果不说明具体的语言，那么就是 C#（因为我比较喜欢这门语言）。因为 Java 和 C#的语法是如此的相像，我想，对有经验的程序员来说，这应该不会造成阅读上的麻烦。

第2章 系统的分层结构

2.1 简述

我们在解决一个复杂的问题的时候,通常使用的一个技巧就是分解,把复杂的问题分解成为若干个简单的问题,逐步地、分别地解决这几个小问题,最后就把整个问题解决掉。在设计一个复杂的软件系统的时候,同样的,为了简化问题,我们也通常使用的一个技术就是分层,每个层完成自身的功能,最后,所有的层整合起来构成一个完整的系统。

分层是计算机技术中的常用方法,一个典型的例子就是 TCP/IP 技术的 OSI 七层模型。在应用软件开发中,典型的的就是 N 层应用软件模型。N 层的应用软件系统,由于其众多的优点,已经成为典型的软件系统架构,也已经为广大开发人员所熟知。

在一个典型的三层应用软件系统中,应用系统通常被划分成以下三个层次:数据库层、应用服务层和用户界面层。如下图(图 2.1)所示:



图 2.1

其中,应用服务层集中了系统的业务逻辑的处理,因此,可以说是应用软件系统中的核心部分。软件系统的健壮性、灵活性、可重用性、可升级性和可维护性,在很大程度上取决于应用服务层的设计。因此,如何构建一个良好架构的应用服务层,是应用软件开发者需要着重解决的问题。

为了使应用服务层的设计达到最好的效果,我们通常还需要对应用服务层作进一步的职能分析和层次细分。很多开发者在构建应用服务层的时候,把数据库操纵、业务逻辑处理甚至界面显示夹杂在一起,或者,把业务逻辑处理等同于数据库操纵,等等,这些,都是有缺陷的做法。我们将就在这个方面进行设计时可采用的方案进行一些探讨。

在一个分布式应用系统中,整个系统会部署在不同的物理设备上,如上面所示的三层体系,用户界面和应用服务器可能在不同的设备上,这就涉及到不同机器之间的通信问题,也就是层间的通信和交互问题。我们已经有了很多可以用于分布式远程访问的技术,如 CORBA,在 Java 平台上,我们还有 Java RMI、EJB,在 Windows 平台上,从 DCOM 到 COM+,再到 .Net 下的 Web Service 和 .Net Remoting 等。如何选用合适的远程访问技术,也是我们在系统框架中需要考虑的问题。^[6]

为了使讨论更具有针对性,本文也会讨论一些比较流行的系统架构,例如 J2EE 架构,以及 JDO,然后,我们会讨论 Websharp 在这个方面的一些设计理念。

2.2. 设计的原则和评判标准

同软件工程的原则一样,应用服务层的设计,必须遵循的最重要的原则就是高内聚和低耦合^[7]。软件分层的本来目的,就是提高软件的可维护性和可重用性,而高内聚和低耦合正是达成这一目标必须遵循的原则。尽量降低系统各个部分之间的耦合度,是应用服务层设计中需要重点考虑的问题。

内聚和耦合,包含了横向和纵向的关系。功能内聚和数据耦合,是我们需要达成的目标。横向的内聚和耦合,通常体现在系统的各个模块、类之间的关系,而纵向的耦合,体现在系统的各个层次之间的关系。

系统的框架,通常包含了一系列规范、约定和支撑类库、服务。

对于如何判断一个软件的系统框架的优劣,笔者认为,可以从以下几个方面来评判:

系统的内聚和耦合度

这是保证一个系统的架构是否符合软件工程原则的首要标准。

层次的清晰和简洁性

系统每个部分完成功能和目标必须是明确的,同样的功能,应该只在一个地方实现。如果某个功能可以在系统不同的地方实现,那么,将会给后来的开发和维护带来问题。

系统应该简单明了,过于复杂的系统架构,会带来不必要的成本和维护难度。在尽可能的情况下,一个部分应该完成一个单独并且完整的功能。

易于实现性

如果系统架构的实现非常困难,甚至超出团队现有的技术能力,那么,团队不得不花很多的精力用于架构的开发,这对于整个项目来说,可能会得不偿失。简单就是美。

可升级和可扩充性

一个系统框架,受设计时技术条件的限制,或者设计者本人对系统认识的局限,可能不会考虑到今后所有的变化。但是,系统必须为将来可能的变化做好准备,能够在今后,在目前已有的基础上进行演进,但不会影响原有的应用。接口技术,是在这个方面普遍应用的技巧。

是否有利于团队合作开发

一个好的系统架构,不仅仅只是从技术的角度来看,而且,它还应该适用于团队开发模型,可以方便一个开发团队中各个不同角色的互相协作。例如,将 Web 页面和业务逻辑组件分开,可是使页面设计人员和程序员的工作分开来同步进行而不会互相影响。

性能

性能对于软件系统来说是很重要的,但是,有的时候,为了能让系统得到更大的灵活性,可能不得不在性能和其他方面取得平衡。另外一个方面,由于硬件技术的飞速发展和价格的下降,性能的问题往往可以通过使用使用更好的硬件来获得提升。

2.3. 应用服务层的内容

应用服务层，通常也被称为业务逻辑层，因为这一层，是应用软件系统业务逻辑处理集中的部分。然而，我将这一层称为应用服务层，而不称业务逻辑层，因为，这一层需要处理的不仅仅是业务逻辑，还包含了其他方面的内容。

从完整的角度来说，应用服务层需要处理以下内容：

数据的表示方式

数据，是软件处理的对象。从某种程度上来说，“软件，就是数据结构加算法”的说法，是有一定意义的。在面向对象的系统中，数据是用类来表示的，代表了现实世界实体对象在软件系统中的抽象。考虑所谓的 MVC 模式，这个部分的类属于 M--实体类的范畴。由于应用软件通常会使用数据库，数据库中的数据，可以看成是对象的持久化保存。由于数据库一般是关系型的，因此，这个部分，还需要考虑类（对象）同关系型数据的映射，即通常所说的 O-R MAP 问题。

数据的存取方式

如同上述所说，软件系统处理的实体对象数据需要持久化保存数据库中，因此，我们必须处理系统同数据库的交互，以及数据的存取和转换方式的问题。

业务逻辑的组织方式

在面向对象的系统中，业务逻辑表现为对象之间的交互。有了上述的实体对象，以及对象的保存策略，就可以将这些对象组合起来，编写我们的业务逻辑处理程序。在业务逻辑的处理中，必须保证处理的正确性和完整性，这将会涉及到事务处理。通常，我们也会把业务逻辑封装成组件的形式，以得到最大的可重用性。

业务服务的提供方式

在我们完成系统的功能后，如何向客户提供服务，是我们需要考虑的问题。这里的客户，不仅仅是指软件的使用者，也包括调用的界面、其他程序等。例如，在一个基于 Web 的 ASP.Net 或 JSP 系统中，业务逻辑功能的客户便是这些 ASP.Net 页面或 JSP 页面。业务逻辑组件应该通过什么方式，直接的，或间接的，向这些客户提供服务，是这一层需要完成的任务。

层的部署和层间交互

对于一个多层的应用软件系统来说，尤其是大型的应用软件系统，通常需要把不同的部分部署在不同的逻辑或物理设备上。特别是一些基于 Web 的应用软件系统，其部署工作将涉及到 Web 服务器、组件服务器、数据库服务器等不同的服务设备。在进行应用软件架构的设计的时候，必须考虑各种不同的部署方案。当系统需要进行分布式访问的时候，如何统一和简化分布式系统的开发，便成了系统框架需要考虑的内容。

综上所述，一个完整的基于 Web 的应用软件系统，其架构可以用图 2.2 来表示（Websharp 的应用软件系统架构）：

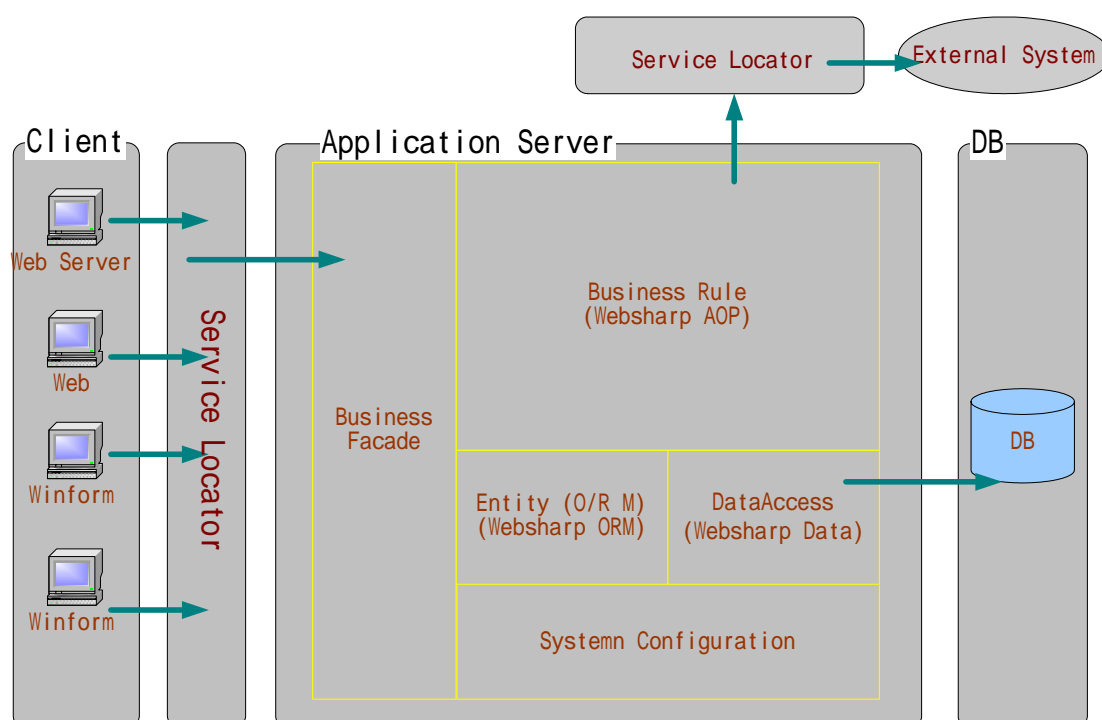


图 2.2

对于以上各个方面来说，每个问题都可以有很多种策略和方案，但是，在一个系统中，应该尽可能的统一这些策略和方案。也就是说，在一个系统，或者一个项目中，应该统一每个解决每个问题所采用的方法。软件的开发方法是灵活的，可以用不同的方法解决相同的问题，这会诱使开发人员采用他们认为能够表现自己的方法，但是，从整个系统来看，这将会是灾难性的。我们应该尽可能统一，就是，采用统一的数据表示方式、统一的数据存取方式、统一的业务逻辑处理方式等。

下面，将就这些部分的设计策略和可用方案进行一些比较详细的论述。

2.4. 数据实体的表示

应用软件系统，从本质上来说，是计算机对现实世界的模拟。现实世界中的实体对象，在软件系统中，表现为需要处理的数据。在面向对象的系统中，这是通过“类”和“对象”来表示的。

参考著名的“MVC”模式^[8]，类可以分成实体类(M)、控制类(C)、和边界类(V)，分别代表了实体对象、控制和界面显示。系统中需要处理的数据，在面向对象的系统中，属于实体类部分。

在考虑数据实体层的设计策略的时候，需要把握以下要点：

一致的数据表示方式。在一个系统中，数据的表示方式必须尽可能统一，同时，在处理单个数据和多个数据的时候，处理方式尽可能一致。

因为数据通常是需要存储到数据库中，因此，良好的映射方法是必需的。

处理好对象的粒度，即所谓的粗粒度对象、细粒度对象。

一般例子

考虑一个现实的例子，一个仓库中的产品（Product），在系统中可以使用如下定义：

```
public class Product
{
    public string Name; //名称
    public decimal Price;//价格
    public int Count;//数量
}
```

可以按照如下方法使用Product类：

```
Product p=new Product();
```

```
//.....处理 Product
```

这是一个包含了三个属性的 Product 类的定义。为了便于说明，在这里，我们尽量将问题简化了。

又例如，一张入库单可以使用如下定义：

```
public class Form
{
    public string ID; //入库单编号
    public DateTime AddTime; //入库时间
    public FormDetail[] FormDetails; //入库单明细
}
public class FormDetail
{
    public Product InProduct; //入库产品
    public int Count; //入库数量
}
```

对于处理单个对象，通常采用上述的方法，但是，当我们需要处理相同类的一组对象，也就是处理一个对象集合的时候，就会有一些小小的麻烦。

如前所述，我们希望在处理单个对象和对象集合的时候，处理的方式尽量统一，这对于软件开发的意义是很大的。常用的处理对象集合的方法有：

数组表示的方法

例如，上面的例子中当一张入库单包含多条入库单明细的时候采用的方法。为了灵活性，也可以使用容器来，如 Java 中的 Vector 或 C# 的 ArrayList(C#)。只是，在处理对象的时候，

需要一个类型转换的操作。这个问题，在支持泛型的语言中不会存在，如使用 C++ 的标准库的容器类。

ObjectCollection 方法。

这个方法同上面的方法类似，不同之处在于，为每个实体类设计一个 Collection 类。例如，可以为 FormDetail 设计一个 FormDetailsCollection 类(C#)：

```
public class FormDetailsCollection: ArrayList
{
    public void Add(FormDetail detail)
    {
        base.Add(detail);
    }
    public new FormDetail this[int nIndex]
    {
        get
        {
            return (FormDetail)base[nIndex];
        }
    }
}
```

这么做的好处在于，在操作集合中的对象时，不必进行类型转换的操作。

数据集的表示方法。

采用这种方法，通常是直接把从数据库查询中获取的数据集(Recordset)作为数据处理对象。这种方法在 ASP 应用程序中是非常常见的做法。这种做法简单，初学者很容易掌握，但是他不是一种面向对象的方法，弊病也很多。

EJB 的方法

在J2EE体系中，对实体对象的处理的典型方法是Entity Bean。J2EE中使用Entity Bean来表示数据，以及封装数据的持久化储存（同数据库的交互）。由于Entity Bean比较消耗资源，而且采用的是远程调用的方式来访问，因此，在需要传递大量数据，或者在不同的层次之间传递数据的时候，往往还会采用一些诸如“值对象”(Value Object)的设计模式来提升性能。关于J2EE中的设计模式的更多内容，可以参考《J2EE核心模式》一书。^[9]

JDO 的方法

相对于J2EE这个昂贵的方法来说，JDO提供了一个相对“轻量级”的方案。在JDO中，你可以采用一般的做法，编写实体类，然后，通过一些强化器对这些类进行强化，以使其符合JDO的规范，最后，你可以通过PersistenceManager来实现对象的持久化储存。^[10]

无论是 EJB 还是 JDO，在同数据库进行映射的时候，都选用了 XML 配置文件的方式。这是一种灵活的方式。由于 XML 强大的表达能力，我们可以很好的用它来描述代码中的实体类和数据库之间的映射关系，并且，不用在代码中进行硬编码，这样，在情况发生变化的时候，有可能只需要修改配置文件，而不用去修改程序的源代码。关于 EJB 和 JDO 的配置文件的更多的信息，各位可以参考相关的文档，这里不再赘述了。

然而，使用 XML 配置文件的方式并不是唯一的方法，在微软提供的一些案例中，如 Duwamish 示例^[11]，就没有采用这种方式。至于开发人员在开发过程中具体采用哪种方式，是需要根据具体情况进行权衡和取舍的。

Websharp 的方法

Websharp 在数据的表现上，充分利用了 .Net Framework 类库中 DataSet 和特性 (Attribute) 的功能。我们设计了一个 EntityData 类，这个类继承了 DataSet，并增加了一些属性和方法。

在 Websharp 中，当表示一个实体类的时候，需要定义一个抽象类，这个抽象类继承 PersistenceCapable。例如，一个 Schdule 类可以表示如下：

```
[TableMap("Schdule", "GUID")]
[WebsharpEntityInclude(typeof(Schdule))]
public abstract class Schdule : PersistenceCapable
{
    [ColumnMap("GUID", DbType.String, "")]
    public abstract string GUID {get;set;}

    [ColumnMap("UserID", DbType.String, "")]
    public abstract string UserID {get;set;}

    [ColumnMap("StartTime", DbType.DateTime)]
    public abstract DateTime StartTime {get;set;}

    [ColumnMap("EndTime", DbType.DateTime)]
    public abstract DateTime EndTime {get;set;}

    [ColumnMap("Title", DbType.String, "")]
    public abstract string Title {get;set;}

    [ColumnMap("Description", DbType.String, "")]
    public abstract string Description {get;set;}

    [ColumnMap("RemidTime", DbType.DateTime)]
    public abstract DateTime RemidTime {get;set;}

    [ColumnMap("AddTime", DbType.DateTime)]
```

```
public abstract DateTime AddTime {get;set;}

[ColumnMap("Status", DbType.Int16, 0)]
public abstract short Status {get;set;}

}
```

类的 TableMap 特性指明了同 Schdule 实体类相映射的数据库表，以及关键字，ColumnMap 特性指明了同某个属性相映射的数据库表字段，以及数据类型和默认值。

在实际的应用中，定义了这样一个 Schdule 抽象类后，要获取一个实体对象，因为 Schdule 类是抽象的，所以你不可以直接使用 new 操作来初始化 Schdule 对象，应当通过如下方式取得：

```
Schdule schdule = EntityManager.CreateObject(typeof(Schdule)) as Schdule;
```

EntityManager 会即时编译出一个 Schdule 的实现类，并且返回一个对象。

在这种方式下，实体类同数据库表的映射是通过 Attribute 来实现的。

可以使用另外一种方法来表示一个实体类。在这种方式下，需要编写一个 XML 映射文件，然后，可以使用如下方式取得一个实体对象：

```
EntityData schdule =EntityManager.GetEntityData("Schdule");
```

然后，可以通过如下方式来访问这个对象的属性：

```
string Title = schdule["Title"]
```

可以看到，这种方式同传统的方式有点不同。在这种方式下，数据的表现形式只有一个，那就是 EntityData。其好处是明显的，不用为每个实体都单独编写一个类，能够大大减少代码的编写量。其缺点也很明显，那就是不能利用编译器类型检测的功能，如果在调用对象的属性时，写错了属性的名称，就可能出错，这需要更加仔细的测试工作。但是，这个问题可以通过工具生成代码来解决。

2.5. 数据的存取方式

数据存取的目的，是持久化保存对象，以备后来的使用，如查询、修改、统计分析等。存取的对象，可以是数据库、普通文件、XML 甚至其他任何方式，只要保证数据能够长久保存，并且，不会受断电、系统重起等因素的影响。在这个部分，最理想的状况，自然是能够支持除了数据库以外的各种类型的存取方式，或者，至少留有接口，能够比较方便的扩充。

因为数据库是最常用，也是最有效的数据存储方法，因此，支持数据库存储是最首先必须支持的。在不同的平台下，有不同的数据库访问的手段。例如，在 Java 平台下，有 JDBC，在 Windows 平台下，可以使用 ADO、ADO.Net 等。但是，这些手段还比较接近底层，在

实际操纵数据库的时候，需要编写大量的代码，并且，我们还需要通过手工的方式来完成将程序中的面向对象的数据存储到关系型数据库的工作。这么做，自然编程的效率不高，并且非常容易出错。但是，不可否认，这也是一种可以选用的方式。

从另外一个方面来看，由于我们前面已经解决了数据的映射问题，因此，在数据的存取方面是非常有规律的，我们完全可以让这个工作通过框架来执行。这样，我们一方面可以简化很多同数据库交互方面的代码编写工作量，能够减少出现 Bug 的几率，另一方面，由于框架封装了不同数据库之间的差异，使得我们在编写程序的时候，不用考虑不同数据库之间的差异，而将这个工作交给框架去做，实现软件的后台数据库无关性。

在这个部分，以下两个部分的类会显得特别重要：

对象--关系映射的分析类，能够通过既定的方案完成对象--关系的映射，确定数据存取方案

数据库操纵类：根据映射关系，将数据准确的存储到数据库中，并且封装不同数据库之间的差异。

这个部分的操作过程，可以用图(图 2.3)大概的表示如下：

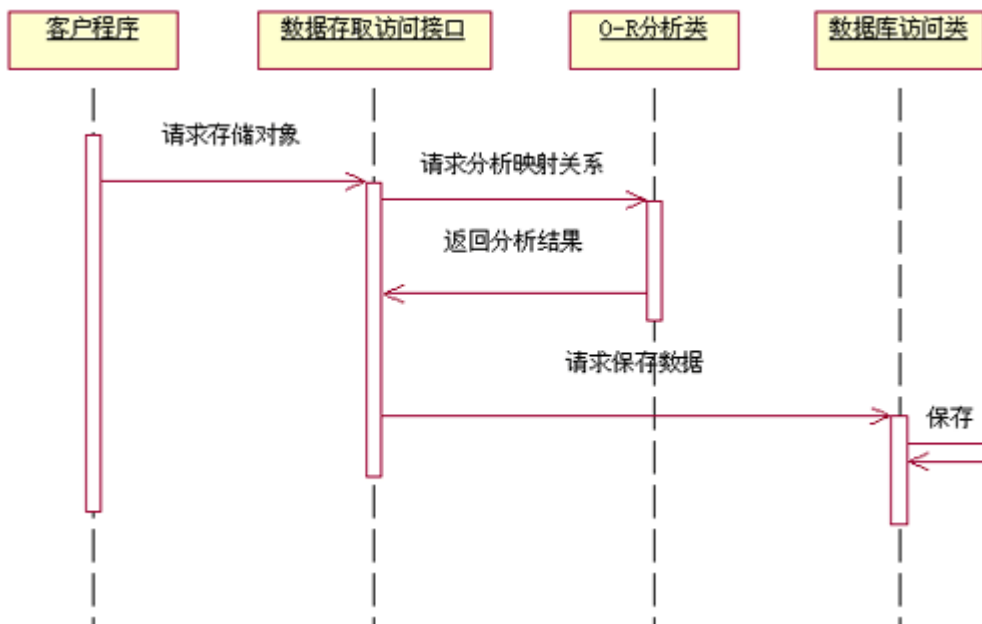


图 2.3

在 J2EE 中，这个部分比较典型的的就是 EntityBean 中的 CMP。由于在 BMP 中，同数据库的交互部分需要通过手工编写代码的方式来实现，因此，很难享受到容器带来的便利，只是由于 EJB2.0 以前的标准，CMP 的功能，包括映射能力、实体关系模式等方面的功能比较弱，所以，在很多时候，我们不得不使用 BMP。现在，EJB2.0，在这个方面的功能已经非常强大了，我们完全可以享受容器带来的便利，而将大部分精力放在实现更加复杂的业务逻辑方面了。

在 JDO 中，您同样可以通过 PersistenceManager 来实现同样的目标，例如，您想把一个 Customer 对象保存到数据库中，可以采用类似于下面的代码：

```
Schedule schdule=new Schedule(.....);  
PersistenceManager PM=PMFactory.initialize(.....);  
Pm.persist(schdule);
```

代码同样非常简明和直观，没有一大堆数据库操纵的代码，也不容易发生差错。

Websharp 的方案

同 JDO 类似，Websharp 定义了 PersistenceManager 接口，这个接口的定义在后面的章节中会给出，这里，我们先看看其使用方式。

当我们有了某个实体对象后，需要保存到数据库中的时候，我们可以使用下面的代码来实现：

```
public bool AddSchedule(Schedule schdule)  
{  
    PersistenceManager pm =  
        PersistenceManagerFactory.Instance().CreatePersistenceManager();  
    try  
    {  
        pm.PersistNewObject(schdule);  
        return true;  
    }  
    catch  
    {  
        return false;  
    }  
    finally  
    {  
        pm.Close();  
    }  
}
```

在这里，我们不需要关心具体的数据库版本，框架会封装不同数据库之间的差异，保证数据可以正确的存储到不同的数据库中。

在这个部分，另外需要注意的是，为了保证数据存储的完整性，应当考虑事务处理的功能。J2EE、JDO 和 Websharp 都支持在数据存储的时候使用事务处理。

在 Websharp 中，通过 Transaction 接口，提供了基本的事务处理能力。上面的代码，如果需要使用事务处理，则可以修正如下：

```
public bool AddSchdule(Schdule schdule)
{
    if(!CheckSchdule(schdule))
        return false;
    PersistenceManager pm =
        PersistenceManagerFactory.Instance().CreatePersistenceManager();
    Transaction trans = pm.CurrentTransaction;
    trans.Begin();
    try
    {
        pm.PersistNewObject(schdule);
        trans.Commit();
        return true;
    }
    catch
    {
        trans.Rollback();
        return false;
    }
    finally
    {
        pm.Close();
    }
}
```

关于事务处理的 Transaction 接口的更多内容，在后面的章节中会详细说明。

2.6. 业务逻辑的处理

有了上面的工作，我们就可以把这些对象组合起来，编写我们的业务逻辑。在面向对象的系统中，业务逻辑表现为对象之间的交互。在一些简单的系统中，没有复杂的业务逻辑，只是一些数据的维护工作，那么，有了上面两个部分的工作，我们实际上可能已经忘成了大部分的工作。

在这个部分，由于不同系统之间业务逻辑千差万别，基本上没有办法提供统一的模式。但是，应当注意的是，在同一个系统中，采用基本一致的策略是非常必要的，这有助于消除项目内部的不一致性，使项目更加可控。甚至于，这些策略可以扩展成公司部分、甚至所有项目的策略。

值得指出的是，很多人在这个部分操纵数据库，把业务逻辑处理等同于数据库操作，这是不可取的。在业务逻辑处理中，处理的应该是对象，而不是直接同数据库打交道，这样，才能获得更好的系统结构。

在业务逻辑处理部分，由框架提供一些支撑的服务是非常必要的。这其中，最重要的一点就是事务的处理。业务逻辑的处理过程，会涉及到多个对象之间的交互，以及多次同数据库的交互。为了保证处理过程的完整性，必须使用事务处理的方法。框架必须支持事务处理。

事务处理的功能，基本上有两种选择：使用基于数据库连接的事务、使用外部事物处理服务^[12]。

使用基于数据库连接的事务，事务处理的性能相对比较高，但是，当系统涉及到多个数据库之间的交互时，基于数据库连接的事务便无能为力了。而使用专用的事务处理服务，能够适应更多的情况，并且，有测试表明，随着数据处理量的上升，两者之间的性能差异会逐渐减小。

在 J2EE 中，容器提供了事务处理的能力。在 .Net 平台上，事务处理是通过 Windows COM+ 服务来提供的。在 Websharp 中，如上面所讲，通过 Transaction 接口，提供了基本的事务处理能力，能够满足大部分事务处理的要求。当 Websharp 提供的事务处理能力不能满足需求的时候，可以使用 EnterpriseService。

下面是一个简单的例子：

```
public bool AddSchedule(Schedule schedule, string[] otherPeoples)
{
    if(!CheckSchedule(schedule))
        return false;
    PersistenceManager pm =
        PersistenceManagerFactory.Instance().CreatePersistenceManager();
    Transaction trans = pm.CurrentTransaction;
    trans.Begin();
    try
    {
        pm.PersistNewObject(schedule);
        foreach(string otherPeople in otherPeoples)
        {
            Schedule s = EntityManager.CreateObject(typeof(Schedule)) as Schedule;
            s.GUID = Guid.NewGuid().ToString();
            s.UserID = otherPeople;
            s.StartTime = schedule.StartTime;
            s.EndTime = schedule.StartTime;
            s.Title = schedule.Title;
            s.Description = schedule.Description;
            s.RemidTime = schedule.RemidTime;
            s.AddTime = DateTime.Now;
        }
    }
}
```

```
s.Status = 0;
pm.PersistNewObject(s);
}
trans.Commit();
return true;
}
catch
{
    trans.Rollback();
    return false;
}
finally
{
    pm.Close();
}
}
```

在业务逻辑这一层，另外一个需要关注的问题是所谓的 AOP。关于 AOP 的内容，我们会在下面的章节中再讨论。

2.7. 业务服务的提供

业务外观层 (Business Facade) 的目的，是隔离系统功能的提供者和使用者，更明确地说，是隔离业务逻辑的软件的用户界面 (可以参见 Facade 设计模式)。这一层没有任何需要处理的逻辑，只是作为后台逻辑处理和前端用户界面的缓冲区，以达到如下目的

将用户界面和系统业务逻辑处理分开，这样，当业务逻辑发生变化时，不用修改客户端程序，是一种支持变化的设计方法。

使同一个业务逻辑能够处理不同的客户端请求。例如，可以将 Facade 设计成 Web Service，这样，可以同时为传统的 WinForm 客户端程序、Web 程序以及其他外部系统提供服务，而使用相同的应用服务层，同时，也可以实现系统的分布式部署。关于如何做到这一点，可以参见 loffice 的 Demo 程序。

作为系统不同模块之间的调用接口。一个系统通常会包含很多模块，这些模块相对独立，又可能互相调用。为了减少各个不同部分之间的耦合度，必须采用一定的设计方法，Facade 设计模式就是非常有效的一种，也是业务外观层的基础。

有利于项目团队的分工协作。业务外观层作为一个访问接口，将界面设计人员和逻辑设计人员分开，使得系统的开发可以实现纵向的分工，不同的开发人员可以关注自己的领域而不会受到干扰。

业务外观层的代码框架，在系统分析和设计完成后就可以完成，他需要提供的方法，就相当于在界面设计人员和逻辑设计人员之间签订了一个协议，他虽然没有实现任何逻辑，但

是，他的引入，能使系统的开发更加有条理，更加简明。套用《设计模式》上的一句话，就是，“任何问题，都可以通过引入一个中间层来得到简化”。

2.8. 层的部署和层间交互

对于一个多层的应用软件系统来说，尤其是大型的应用软件系统，通常需要把不同的部分部署在不同的逻辑或物理设备上。特别是一些基于 Web 的应用软件系统，其部署工作将涉及到 Web 服务器、组件服务器、数据库服务器等不同的服务设备。在进行应用软件架构的设计的时候，必须考虑各种不同的部署方案。

已经有了很多可以用于远程访问的服务，如此多的实现技术，带来的很大的灵活性，但同时也带来了文题，其中一个就是，有多少种服务端技术，就得有多少种相应的客户端访问技术。甚至，在某些分布式应用系统中，应用逻辑使用不同的技术开发，存在于不同的机器上，有的存在于客户机本机，有的使用 .Net Remoting 开发，存在于局域网内，有的使用因特网上的 Web Service，有的时候，我们希望相同的业务逻辑能够支持不同的客户端。

在这种情况下，我们需要一个一致的服务访问编程模型，以统合不同的服务访问模式，简化系统的开发和部署。Websharp 中的 Service Locator 提供了这样一种能力，开发人员只需要定义服务访问接口，就可以使用一致的方式透明的访问这些服务，而不用理会这些服务之间的不同点。框架会自动生成访问远程服务需要的代理。

使用 WSL，你可以使用类似于如下的代码来访问远程服务，而不用关心远程服务的种类：

```
public interface ISecuritySystem
{
    bool Login(string userID, string password);
    void Logout();
    bool IsLogin();
    Suser CurrentUser();
}
.....
//在需要调用服务的客户端:
ISecuritySystem ss = ServiceLocator.FindService(
    "SecurityService", typeof(ISecuritySystem)) as ISecuritySystem;
```

关于 WSL 的更多内容，在后面会更加详细的讨论。

2.9. 剪裁和取舍

以上四个层次，对于大型的应用软件系统来说，是非常必要的。但是，对于一些小型的应用软件系统，如果完全按照以上的层次来做，可能反而会影响工作效率。因此，针对不同的系统，可以对架构进行一定的剪裁。

数据实体层和实体控制层，是每个应用软件系统所必需的，显然无法裁减。对于业务逻辑层和业务外观层，根据实体情况，可以进行如下裁减：

如果系统没有复杂的业务逻辑,而只是一些数据的操作,或者业务逻辑特别少,那么,可以省略业务逻辑层,而将相关的功能移至实体控制层。

如果不考虑多种客户端的情况,也不考虑分布式部署的问题,系统的模块又很少,不会产生模块间紧耦合的情况,那么,可以不使用业务外观层,而让用户界面程序直接访问业务功能。

在上面的论述中,对于每个层次,都说明了可以选择的多种方案,每一种方案都有他的优点和缺点,在具体开发的过程中,需要根据具体情况加以取舍。

2.10. 小结

应用软件系统架构,是软件工程的重要组成部分。设计一个好的框架,其目的很明确,那就是,在目前还没有"银弹"之前,尽最大的可能,提高软件开发的效率和软件质量,把不必要的工作和容易出错的工作,交给框架去处理。

应用服务层,在软件系统中,是一个非常复杂的部分,乍看之下,没有任何规律可行,给人无从下手的感觉。我们的目标,就是尽量化无规律为有规律,把有规律的东西提取出来,形成规范,从而减少今后的开发工作量。其方法,就是对系统进行合理的分层,这样,系统的层次清晰了,每个层次完成的功能就比较单一,就意味着每个层次的都相对更有规律可循,这样,我们就可以把这些有规律的东西交给框架去执行,或者,开发一个辅助工具,来完成这部分的代码编写工作。Websharp 就提供了这样一个代码自动生成的工具。这个工具被设计成 Visual Studio.Net 集成开发环境的插件,在实际开发过程中,能够提供很多便利。这是系统层次清晰带来的另外一个好处。

对于一个软件公司来说,统一的系统框架的意义不仅仅在于软件开发的本身。一个统一的系统框架,也是公司知识管理的重要组成部分。公司如果有一个或有限个数的明确的软件框架,那么,这些框架就可以成为凝结公司开发人员经验、智慧的载体,并且可以在不断的实践中加以充实和完善。由于公司的软件系统的框架比较统一,那么当某个项目更换或增加开发人员的时候,后来的人也能够比较容易接手,这对于公司的开发管理是具有非常重要的意义的。

第二部分 应用服务层的设计

第3章 数据和对象

3.1 数据的形态

在应用软件系统中，首先要处理的对一个对象就是数据。应用软件系统，主要目标就是采集数据、处理数据、分析数据、察看数据。对于软件，诚如有一句名言所说：“软件，就是数据结构加算法”。

在软件中，数据有多种表现形态。

首先，在程序中，数据总是以某种数据结构的方式被表示出来，这种表示，通常被编译成二进制文件存在于硬盘上，并且在运行时刻在内存中被实例化。

这种数据结构有多种表达的方式，简单的情况下，他可能只是一个数字，或者一个字符串，用某个变量来描述。例如，为了表述某种商品的价格，可能使用如下的申明来表述这个数据：

```
double price = 100 ;
```

现实中要处理的数据总是比较复杂的，为了描述一个完整的信息，通常要组合多项简单的数据，例如，为了描述某种商品的信息，通常需要描述他的名称、价格、重量等。在传统的 C 语言中，可以使用结构来描述：

```
struct product
{
    char* name;
    double price;
    double weight;
}
```

在面向对象的语言里，类似的数据结构，可以使用类来表述。上面的代码可以用 Java 语言表述如下：

```
public class Product
{
    public String name;
    public double price;
```

```
public double weight;  
}
```

可以看出来，实际上两者的差别是非常小的。

对于更加复杂的数据结构，一个类可能引用到其它的类，例如，上面的 Product，可能有一个 Size 属性，而这个 Size 属性，也有 height 和 width 构成，那么，整体的数据结构就可以描述如下：

```
public class Product  
{  
    public String name;  
    public double price;  
    public double weight;  
    public Size size;  
}  
public class Size  
{  
    public int height;  
    public int width;  
}
```

数据的另外一种表现形态，就是永久化保存的形态。上面描述的数据的形态，是一种“瞬时”的数据，只有在程序运行的时候才存在于内存中，一旦程序结束，或者数据处理结束，数据就从内存中清除。在很多情况下，需要把处理的数据保存到磁盘上，这时候，数据就进入永久化保存状态。

可以有多种保存数据的格式。可以把数据保存为普通文本文件存放在磁盘上，或者，也可把数据保存在 XML 文件中。在 Java 和 C# 中，也都提供了这样一种能力，就是可以把对象序列化后保存到磁盘上，然后，在需要的时候，可以反序列化对象。

虽然有多种持久化保存数据的方案，然而其中使用关系型数据库来保存数据，无疑是最常用的办法和最可靠的办法。这就引伸出一个在面向对象的系统设计中的常见问题：对象/关系型映射 (O/R Mapping)。

在考虑 O/R Mapping 的时候，有两个概念是经常会接触的，那就是 VO 和 PO

所谓的 VO，就是 Value Object，这种对象，只包含了对象的数据，而没有状态，或者说，处于瞬时状态。VO 可以用来在层之间传递数据

所谓 PO，就是 Persistent Object，就是持久化保存的对象，这种对象，一般是有状态的。O/R Mapping 框架需要根据 PO 的状态，来执行相应的同数据库的交互。关于 PO 的状态，

我们在后面再讨论

3.2 对象/关系型映射

对象关系型映射，最核心的要完成两个功能：对象和关系型之间的映射规则，以及两者之间的相互转换。

除了这两个基本的功能，一般的 O/R Mapping 产品还会加上一些额外的特性和功能，以加强产品的功能，为软件开发提供更多的方便和提高性能。一些常见的功能，例如缓存。

现在有一些典型的 O/R Mapping 框架可以参考和使用，比较著名的有 EJB 中的 Entity Bean，JDO，Hibernate 等，这些方案都是基于 Java 的。在 Microsoft .Net 平台下，相对来说可供选择的方案比较少，其中有一个开放源代码的方案 Websharp，可以从 www.websharp.org 下载。

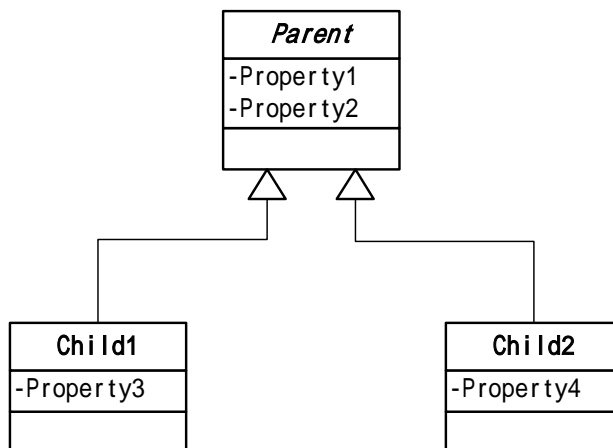
我们在实际开发中，可以选择使用已有的解决方案和产品，也可以自己设计自己的 O/R Mapping 框架。当然，无论采用何种方式，都需要我们对 O/R Mapping 的基本原理和方法有一个基本的了解。

在支持 OO 的语言中，继承是语言的基本特征。O/R Mapping 的框架，也需要对继承做出相应的支持。一般说来，有三种继承模式：ONE_INHERITANCE_TREE_ONE_TABLE、ONE_INHERITANCE_PATH_ONE_TABLE 和 ONE_CLASS_ONE_TABLE。

➤ ONE_INHERITANCE_TREE_ONE_TABLE：

一个继承树映射到一个表，即将具有相同父类的所有类都映射到一个数据库表中，这些类属性映射的集合组成这个表的列，在这种模式下，只需要对最底层的类进行映射。

如下面一个类结构：



在这个类结构中，父类 Parent 有两个子类 Child1 和 Child2，父类有属性 Property1 和 Property2，Child1 新增了一个属性 Property3，而 Child2 新增了另外一个属性

Property4, 当采用 ONE_INHERITANCE_TREE_ONE_TABLE 映射模式的时候, 数据库中只有一张表, 结构如下:

Table1	
PK	<u>Column1</u>
	Column2 Column3 Column4

其中, Column1 和 Column2 字段分别对应 Parent 类的 Property1 和 Property2 属性, Column3 字段对应 Child1 的 Property3 属性, 而 Column4 字段对应 Child2 的 Property4 属性。Column3 对于 Child2 和 Column4 对于 Child1 是没有意义的。

采用这种映射模式, 优点是比较简单, 缺点是数据的冗余比较大。这个表要容纳所有的子类的字段, 很多字段只是对某个类有意义, 而对于其他类则没有意义, 是纯粹多余的, 并且, 在继承树中新增一个继承节点的时候, 往往导致表的字段的重新设计, 这是一件非常麻烦的事情。

➤ ONE_INHERITANCE_PATH_ONE_TABLE :

一个继承路径映射到一个表, 即将一个类映射到一个数据库表中, 这个类的所有属性(包括从父类继承来的)映射的集合组成这个表的列。

在这种模式下, 对于上面的类结构, 数据库的结构就是:

Child1	
PK	<u>Column1</u>
	Column2 Column3

Child2	
PK	<u>Column1</u>
	Column2 Column4

其中, 表 Child1 和 Child2 分别对应于类 Child1 和 Child2。

这种模式是非常常用的, 也没有数据冗余, 缺点是在搜索的时候比较麻烦。例如, 当我要搜索符合某个条件的 Parent 对象的时候, 需要同时搜索 Child1 和 Child2 表, 并且, 当在继承树中新增一个继承节点的时候, 需要新增一个表, 搜索的范围也必须扩大, 原来的程序可能不得不重新设计。

➤ ONE_CLASS_ONE_TABLE :

一个类映射到一个表，即将每个类映射到对应的一个表，这个类的属性(不包括从父类继承来的非主键属性)映射组成这个表的列，在这种模式下，具有继承关系的类被映射到不同的表中，表之间通过主键关联。

在这种模式下，对于上面的类结构，数据库的结构就是：

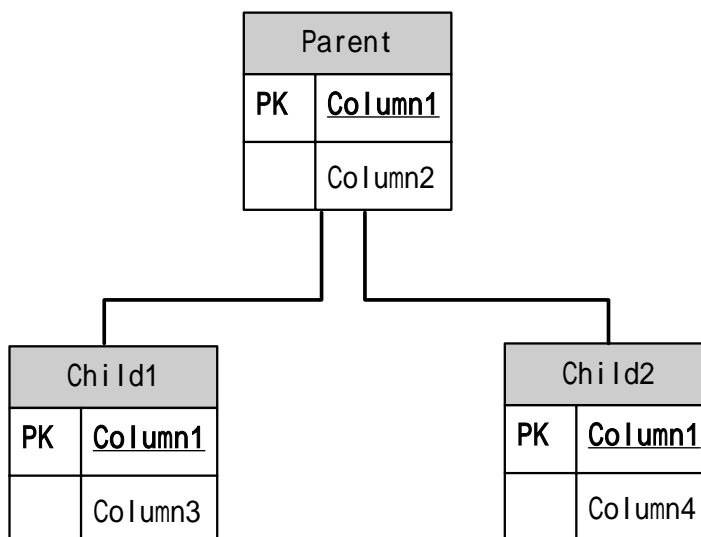


表 Parent 作为 Child1 和 Child2 的父表，父子表之间通过主键 Column1 关联。

这种模式是很容易理解的，因为和类图非常相像，缺点是在查询的时候，由于设计到表的关联，SQL 语句会比较复杂，效率也会比较低。这个情况当继承数的深度增加的时候，会体现的很明显。

如果一个类没有子类和父类，那么采用三种模式中的哪一种都是一样的效果。

3.3 对象的状态

为了很好的控制对象，以及在同后台存储交互时的行为，通常 O/R Mapping 框架需要维护 PO 对象的状态。在不同的框架中，对对象状态的定义不尽相同，不过，也都有一些共同点，某些方面可能只是名称的不同。通常的 O/R Mapping 框架都需要以各种方式来直接或间接的处理 PO 的这些状态。下面列出的一些状态是一些基本的，比较共通的一些状态

- Transient
- Persistent-new
- Persistent-dirty
- Persistent-clean
- Persistent-deleted

Transient

在这种状态下，对象处于一种“瞬时”的状态，还没有同任何数据库的数据相关联，对象也不受 O/R Mapping 框架的运行时控制，对象的表现就是一个普通的类的实例。

比较典型的，在 JDO 里面，对于 Transient 的状态的说明如下：

JDO instances created by using a developer-written or compiler-generated constructor that do not involve the persistence environment behave exactly like instances of the unenhanced class.

There is no JDO identity associated with a transient instance.

There is no intermediation to support fetching or storing values for fields. There is no support for demarcation of transaction boundaries. Indeed, there is no transactional behavior of these instances, unless they are referenced by transactional instances at commit time.

When a persistent instance is committed to the datastore, instances referenced by persistent fields of the flushed instance become persistent. This behavior propagates to all instances in the closure of instances through persistent fields. This behavior is called persistence by reachability.

No methods of transient instances throw exceptions except those defined by the class developer.

A transient instance transitions to persistent-new if it is the parameter of makePersistent, or if it is referenced by a persistent field of a persistent instance when that instance is committed or made persistent.

意思是说，

Persistent-new

这种状态，表示一个新的可持久化对象，该对象还没有被保存到存储介质中。在这种状态下，当事务结束被提交的时候，框架会执行一个插入的操作，将对象保存到存储设备中。

Persistent-dirty

这种状态，表示一个对象是可持久化的对象，已经对应于数据库中的某表记录，但是，在程序中，该对象已经被编辑过，与数据库中的数据并不同步。在这种情况下，当事务被提交的时候，框架会执行一个更新的操作，将对象和数据库同步。

Persistent-clean

这种状态，表示一个对象是可持久化的对象，并且与数据库中的数据是同步的。

Persistent-deleted

这种状态，表示一个对象是可持久化的对象，并且该对象已经被删除。在这种情况下，当事务被提交的时候，框架会执行一个删除的操作，将数据从数据库中删除。

第4章 O/R Mapping 的一般做法

对象和关系型数据库之间的映射,在一个框架中,需要定义映射的规范,在实际开发过程中,对于某个具体的映射,按照规范,使用一定的方法描述映射信息并保存下来,以供程序处理的时候使用。这种描述映射的数据,可以称之为元数据。

什么是元数据?元数据最本质,最抽象的定义为[14]:data about data (关于数据的数据)。它是一种广泛存在的现象,在许多领域有其具体的定义和应用。在软件开发领域,元数据被定义为:在程序中不是被加工的对象,而是通过其值的改变来改变程序的行为的数据。它在运行过程中起着以解释方式控制程序行为的作用。在程序的不同位置配置不同值的元数据,就可以得到与原来等价的程序行为。元数据描述数据的结构和意义,就象描述应用程序和进程的结构和意义一样。元数据是抽象概念,具有上下文,在开发环境中有多种用途。

元数据是抽象概念

当人们描述现实世界的现象时,就会产生抽象信息,这些抽象信息便可以看作是元数据。例如,在描述风、雨和阳光这些自然现象时,就需要使用“天气”这类抽象概念。还可以通过定义温度、降水量和湿度等概念对天气作进一步的抽象概括。

在数据设计过程中,也使用抽象术语描述现实世界的各种现象。人们把人物、地点、事物和数字组织或指定为职员、顾客或产品数据。

在软件设计过程中,代表数据或存储数据的应用程序和数据库结构可以概括为开发和设计人员能够理解的元数据分类方案。表或表单由对象派生出来,而对象又由类派生。

在元数据中有多个抽象概念级别。可以描述一个数据实例,然后对该描述本身进行描述,接着再对后一个描述进行描述,这样不断重复,直到达到某个实际限度而无法继续描述为止。通常情况下,软件开发中使用的元数据描述可扩展为二至三级的抽象概念。比如“loan table”数据实例可以描述为数据库表名。数据库表又可以描述为数据库表对象。最后,数据库表对象可以用一个抽象类描述,该抽象类确定所有派生对象都必须符合的固定特征集合。

元数据具有上下文

人们通常把数据和元数据的区别称为类型/实例区别。模型设计人员表述的是类型(如各种类或关系),而软件开发人员表述的是实例(如 Table 类或 Table Has Columns 关系)。

实例和类型的区别是上下文相关的。在一个方案中的元数据将在另一个方案中变为数据。例如,在典型的系统型 DBMS 中,系统目录将描述包含数据的表和列。这就意味着系统目录描述数据定义,因而可以认为其中的数据是元数据。但只要使用正确的软件工具,仍然可以象操作其它数据一样对这些元数据进行操作。操作元数据的示例包括:查看数据沿袭或表的版本控制信息,或通过搜索具有货币数据类型的列来识别所有表示财务数据的表。在此方案中,如系统目录这样的标准元数据变为可操作的数据。

元数据有多种用途

可以像使用任何类型的应用程序或数据设计元素一样使用元数据类型和实例信息。将设计信息表达为元数据，特别是标准元数据，可以为再次使用、共享和多工具支持提供更多的可能性。

例如，将数据对象定义为元数据使您得以看到它们是如何构造和进行版本控制的。版本控制支持提供一种查看、衍生或检索任何特定 DTS 包或数据仓库定义的历史版本的方法。开发基于元数据的代码时，可以一次性定义结构，然后重复使用该结构创建可作为特定工具和应用程序的不同版本的多个实例。还可以在现有元数据类型之间创建新关系，以支持新的应用程序设计。

从目前的实际应用来看，在已有的一些框架中，通常使用 XML 文件来保存 O/R 映射的元数据。例如 EJB、JDO、Hibernate 等，都无一例外的使用了 XML 文件，其描述的方法也是大同小异。

下面的例子是一个 Entity Bean 的部署描述文件的例子，在其中，包含了 O/R Mapping 的信息。

```
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN"
"http://java.sun.com/dtd/ejb-jar_2_0.dtd">
<ejb-jar>
  <description>Here is the description of the test's beans</description>
  <enterprise-beans>
    <entity>
      <description>... Bean example one ...</description>
      <display-name>Bean example two</display-name>
      <ejb-name>ExampleTwo</ejb-name>
      <home>tests.Ex2Home</home>
      <remote>tests.Ex2</remote>
      <ejb-class>tests.Ex2Bean</ejb-class>
      <persistence-type>Container</persistence-type>
      <prim-key-class>tests.Ex2PK</prim-key-class>
      <reentrant>False</reentrant>
      <cmp-version>1.x</cmp-version>
      <cmp-field>
        <field-name>field1</field-name>
      </cmp-field>
      <cmp-field>
        <field-name>field2</field-name>
      </cmp-field>
      <cmp-field>
        <field-name>field3</field-name>
      </cmp-field>
      <primkey-field>field3</primkey-field>
    </entity>
  </enterprise-beans>
</ejb-jar>
```

```

    <env-entry>
    <env-entry-name>name1</env-entry-name>
    <env-entry-type>java.lang.String</env-entry-type>
    <env-entry-value>value1</env-entry-value>
    </env-entry>
  </entity>
</enterprise-beans>
<assembly-descriptor>
  <container-transaction>
    <method>
    <ejb-name>ExampleTwo</ejb-name>
    <method-name>*</method-name>
    </method>
    <trans-attribute>Supports</trans-attribute>
  </container-transaction>
</assembly-descriptor>
</ejb-jar>

```

而下面，则是一个 JDO 的映射文件的例子：

```

<?xml version="1.0"?>

<jdo>

  <package name="org.lgd.test">
    <class name="Human" objectid-class="Human$ObjectId">
      <field name="ID" primary-key="true"/>
      <field name="name" primary-key="true"/>
      <field name="dress">
        <collection element-type="Dress"/>
      </field>
    </class>
    <class name="Dress">
      <field name="color">
        <map key-type="String" value-type="ColorTye"/>
      </field>
    </class>
    <class name="ColorTye">
      <field name="red" embedded="true"/>
      <field name="blue" embedded="true"/>
    </class>
  </package>

```

```
<package name="org.lgd.test">
  <class name="Human"/>
  <class name="Asian" persistence-capable-superclass="Human">
    <field name="country">
      <collection element-type="Asua$Chinese"/>
      <extension vendor-name="kodo" key="inverse-owner" value="Human"/>
    </field>
  </class>
  <class name="Human$Asian"/>
</package>
</jdo>
```

除了使用 XML 文件来描述元数据信息，现在还有另外一个趋势，就是使用语言本身对元数据的支持来描述映射信息。这个在 Microsoft .Net 中表现得比较明显。微软提供了一种称为 Attribute（特性）的语法来实现语言对元数据的支持（这一点，在 JDK1.5 中也有了类似的语法，并且 EJB3.0 就是采用这种描述方式）。

在 .Net 环境下，微软对元数据的描述是：元数据是一种二进制信息，用以对存储在公共语言运行库可移植可执行文件（PE）文件或存储在内存中的程序进行描述。我们将代码编译为 PE 文件时，便会将元数据插入到该文件的一部分中，而将代码转换为 Microsoft 中间语言（MSIL）并将其插入到该文件的另一部分中。在模块或程序集中定义和引用的每个类型和成员都将在元数据中进行说明。当执行代码时，运行库将元数据加载到内存中，并引用它来发现有关代码的类、成员、继承等信息。

元数据以非特定语言的方式描述在代码中定义的每一类型和成员。元数据存储以下信息：

程序集的说明。

标识（名称、版本、区域性、公钥）。

导出的类型。

该程序集所依赖的其他程序集。

运行所需的安全权限。

类型的说明。

名称、可见性、基类和实现的接口。

成员（方法、字段、属性、事件、嵌套的类型）。

特性。

修饰类型和成员的其他说明性元素。

对于一种更简单的编程模型来说,元数据是关键,该模型不再需要接口定义语言 (IDL) 文件、头文件或任何外部组件引用方法。元数据允许 .NET 语言自动以非特定语言的方式对其自身进行描述,而这是开发人员和用户都无法看见的。另外,通过使用属性,可以对元数据进行扩展。元数据具有以下主要优点:

自描述文件。

公共语言运行库模块和程序集是自描述的。模块的元数据包含与另一个模块进行交互所需的全部信息。元数据自动提供 COM 中 IDL 的功能,允许将一个文件同时用于定义和实现。运行库模块和程序集甚至不需要向操作系统注册。结果,运行库使用的说明始终反映编译文件中的实际代码,从而提高应用程序的可靠性。

语言互用性和更简单的基于组件的设计。

元数据提供所有必需的有关已编译代码的信息,以供您从用不同语言编写的 PE 文件中继承类。您可以创建用任何托管语言(任何面向公共语言运行库的语言)编写的任何类的实例,而不用担心显式封送处理或使用自定义的互用代码。

特性。

使用特性,.NET Framework 允许我们在编译文件中声明特定种类的元数据,来批注编程元素,如类型、字段、方法和属性。在整个 .NET Framework 中到处都可以发现特性的存在,特性用于更精确地控制运行时您的程序如何工作。另外,可以通过用户定义的自定义特性向 .NET Framework 文件发出自己的自定义元数据。

为运行库编译代码时,特性代码被转换为 Microsoft 中间语言 (MSIL),并同编译器生成的元数据一起被放到可移植可执行 (PE) 文件的内部。特性使我们得以向元数据中放置额外的描述性信息,并可使用运行库反射服务提取该信息。

.NET Framework 出于多种原因使用特性并通过它们解决若干问题。特性可以描述如何序列化数据,指定用于强制安全性的特性,以及限制实时 (JIT) 编译器的优化以使代码易于调试。特性还可以记录文件名或代码作者,或在窗体开发阶段控制控件和成员的可见性。

可使用特性以几乎所有可能的方式描述代码,并以富有创造性的新方式影响运行库行为。

在 .Net 中,特性是从 System.Attribute 派生的特殊的类的。

下面的例子是 Websharp 框架的一个使用 Attribute 来描述 O/R Mapping 的例子。

```
[TableMap("Schedule","GUID")]
[WebsharpEntityInclude(typeof(Schedule))]
public abstract class Schedule : PersistenceCapable
{
    [ColumnMap("GUID",DbType.String,"")]
    public abstract string GUID{get;set;}

    [ColumnMap("UserID",DbType.String,"")]
    public abstract string UserID{get;set;}
}
```

```
[ColumnMap("StartTime", DbType.DateTime)]
public abstract DateTime StartTime {get;set;}

[ColumnMap("EndTime", DbType.DateTime)]
public abstract DateTime EndTime {get;set;}

[ColumnMap("Title", DbType.String, "")]
public abstract string Title {get;set;}

[ColumnMap("Description", DbType.String, "")]
public abstract string Description {get;set;}

[ColumnMap("RemidTime", DbType.DateTime)]
public abstract DateTime RemidTime {get;set;}

[ColumnMap("AddTime", DbType.DateTime)]
public abstract DateTime AddTime {get;set;}

[ColumnMap("Status", DbType.Int16, 0)]
public abstract short Status {get;set;}

}
```

按照前面所讨论的内容，在一个应用系统中，如果我们能够确定数据的表现形式，并且能够有一个规则的对象/关系型映射方式，那么，我们就可以设计一个框架，来完成对象的操纵，完成对象和关系型数据之间的转换，用更加简单的话来说，就是完成数据的增、删、改和查询的操作。

管理对象和关系型数据之间的转换，是 O/R Mapping 框架的基本功能，在一些框架中，通常还会包含一些其他方面的功能，用以获取更好的性能或者更高的可靠性，一些比较高级的框架则以中间件的形式存在，例如一些 EJB 容器。这些常见的功能包括：

- 缓存
 - 事务处理
 - 数据库连接池
- 等等。

对象存储的一般过程是：

- 程序提交 PO 保存
- 状态管理器读取 PO 的状态，确定需要保存的内容
- 读取元数据信息，和数据库信息，生成相应的 SQL 语句
- 获取数据库连接
- 存储数据

➤ 修改 PO 的状态

为了方便框架对对象的操作，通常都需要实体类的编写符合某种规范。然后，可以使用统一的方法来操纵对象。

在数据操纵的设计方面，有两种方法倾向，一种是把对数据的增、删、改的方法和实体类设计在一起，另一种方法是把实体类和对实体类的操作分开，设计一个通用的接口来实现对对象的操纵。

第一种方法的典型是 EJB。在 EJB 中，O/R Mapping 通过 Entity Bean 来完成，在 Entity Bean 中，实体数据和对实体数据进行操纵的方法是在一起的。我们可以看一下 EJB 中关于 Entity Bean 的规范定义。

首先，一个 Entity Bean 必须定义 Home 接口，这个接口扩展 EJBHome 接口，并且在这个接口中定义 create、findByPrimaryKey 等方法，如，要定义一个 Customer 类的 Home 接口，可以如下来定义：

```
public interface CustomerHome extends EJBHome
{
    public Customer create(String CustomerID, String CustomerName)
        throws RemoteException , CreateException;
    public Customer findByPrimaryKey(String CustomerID)
        throws RemoteException , FinderException;
}
```

其次，定义远程访问接口，这个接口扩展 EJBObject，在这里，可以定义一些业务方法，例如，一个 Customer 实体类的远程访问接口可以如下定义：

```
public interface Customer extends EJBObject
{
    public void setCustomerName(String customerName) throws RemoteException;
    public String getCustomerName() throws RemoteException;
}
```

最后，需要定义 Bean 的实现类，这个类实现 EntityBean 接口，上述的 Customer 实现类，可以定义如下：

```
public class CustomerEJB implements EntityBean
{
    public String customerID;
    public String customerName;
    public void setCustomerName(String customerName)
    {
        this.customerName = customerName ;
    }
    public String getCustomerName ()
```

```
{  
    return this.customerName;  
}  
}
```

在 Bean 实现中，有一些相关的方法，是用来描述是实体类如何同数据库进行交互的，这些方法包括 `ejbCreate`、`ejbLoad`、`ejbStore`、`ejbRemove` 等，关于细节部分的内容，读者可以参考 EJB 的相关资料。从上面可以看出，在 EJB 中，对于实体类来说，实体类和对实体类的操作，都是定义在一起的，当然，对于 CMP 来说，具体实体类是如何同数据库打交道的细节，是由容器来管理的。

另外一种方式，是将实体数据同对对象的操作分开。这种方式，一般都会定义一个访问接口来供调用。例如，我们可以考察一下 JDO 的方法。

在 JDO 中，任何实体类都必须实现 `PersistenceCapable` 接口，当然，这个接口的实现，可以通过手工的编码的方式来实现，但是在更多的情况下，程序员只需要编写一个普通的 Java 类，然后，通过代码增强器来实现。

在数据操纵方面，主要的访问接口是 `PersistenceManager` 接口。所有对对象的保存、删除的操作都可以通过这个接口来进行。在对象查询方面，JDO 定义了 `Query` 接口，可以从这个接口，根据条件查询数据库，并返回对象的集合。

下面的例子演示了一个简单的网数据库中保存一个 `Customer` 对象的过程：

```
Customer customer = new Customer( "12345" );  
customer.setCustomerName( "My Customer" );  
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory(props);  
PersistenceManager pm = pmf.getPersistenceManager();  
pm.makePersistent(customer);  
pm.close();
```

比较上述两种方法，应该说各有特点，但是，现在比较倾向的，是把实体数据同对对象的操作分开的方式，主要理由在于：

- 1、把实体数据同对对象的操作分开，可以得到一个统一的操作接口，而不是在所有的实体对象中分别来实现，系统的结构更加清晰。
- 2、更加重要的是，在分布式的 N 层应用系统结构中，对对象的逻辑处理存在于应用服务器上，而应用服务层和表现层之间只传递数据。如果采用两者合一的方式，那么想象一下，在表现层调用 `Customer` 对象的 `Add` 方法的时候，会出现什么样的情况？

在组合一个业务逻辑的时候，我们通常的工作是处理一组对象，进行某些运算，然后，将结果显示或者保存起来。在这种情况下，我们通常所处理的对象是个别的。在新增、修改和删除对象的时候，通常都是一个个操作的。

数据保存，最终是要供检索的，有的时候，我们需要查询一组对象，有时候还会使用多个组合查询条件来检索对象。关系型数据库成功的一个重要方面，是得益于 SQL 查询功能的强大，使用 SQL 语句，可以很方便的操纵数据库。在 O/R Mapping 框架中，提供一个相当的查询机制，是非常有必要的。因为对象查询的复杂性，一般会提供单独的机制来实现对象的查询。

我们可以来考察一下 JDO。在 JDO 中，提供了 Query 接口，以及 JDOQL 语法，用来提供对象的查询功能。

下面是 Query 接口的定义，供参考：

```
public interface Query extends java.io.Serializable
{
    void setClass(Class cls);
    void setCandidates(Extent pcs);
    void setCandidates(Collection pcs);
    void setFilter(String filter);
    void declareImports(String imports);
    void declareParameters(String parameters);
    void declareVariables(String variables);
    void setOrdering(String ordering);
    void setIgnoreCache(boolean ignoreCache);
    boolean getIgnoreCache();
    void compile();
    Object execute();
    Object execute(Object p1);
    Object execute(Object p1, Object p2);
    Object execute(Object p1, Object p2, Object p3);
    Object executeWithMap (Map parameters);
    Object executeWithArray (Object[] parameters);
    void close (Object queryResult);
    void closeAll ();
}
```

下面的代码，是一个使用 JDO 的 Query 接口进行查询的例子：

```
Class empClass = Employee.class;
Extent clnEmployee = pm.getExtent (empClass, false);
String filter = "salary > sal";
Query q = pm.newQuery (clnEmployee, filter);
String param = "Float sal";
q.declareParameters (param);
Collection emps = (Collection) q.execute (new Float (30000.));
```

第5章 设计一个 O/R Mapping 框架

在本章中，我们将设计一个可用的 O/R Mapping 框架，来详细讨论一下在 O/R Mapping 中可能用到的一些技术，以及一些问题的处理对策。

整个框架，我们会使用 C# 语言来编写，并且，会以 Websharp 框架作为实际的例子，关于 Websharp 框架的信息和源代码，可以从 www.websharp.org 下载。

5.1 封装数据库访问层

一个好的 O/R Mapping 框架，应当做到数据库无关性，这就要求对数据库的访问做一个封装，能够屏蔽不同数据库之间的差异，这样，在更换后台数据库的时候，能够不用重新修改代码。

在 .Net 中，微软提供的基础数据库访问技术是 ADO.Net。ADO.NET 是基于 .NET 的应用程序的数据访问模型。可以使用它来访问关系数据库系统（如 SQL Server 2000、Oracle）和其他许多具有 OLE DB 或 ODBC 提供程序的数据源。在某种程度上，ADO.NET 代表 ADO 技术的最新进展。不过，ADO.NET 引入了一些重大变化和革新，旨在解决 Web 应用程序的松耦合特性以及在本质上互不关联的特性。

ADO.NET 依赖于 .NET 数据提供程序的服务。这些提供程序提供对基础数据源的访问，并且包括五个主要对象（Connection、Command、DataSet、DataReader 和 DataAdapter）。

目前 ADO.NET 随附了两类提供程序：Bridge 提供程序和 Native 提供程序。通过 Bridge 提供程序（如那些为 OLE DB 和 ODBC 提供的提供程序），可以使用为以前的数据访问技术设计的数据库。Native 提供程序（如 SQL Server 和 Oracle 提供程序）通常能够提供性能方面的改善，部分原因在于少了一个抽象层。

- SQL Server .NET 数据提供程序。这是一个用于 Microsoft SQL Server 7.0 和更高版本数据库的提供程序。它被进行了优化以便访问 SQL Server，并且它通过使用 SQL Server 的本机数据传输协议来直接与 SQL Server 进行通讯。当连接到 SQL Server 7.0 或 SQL Server 2000 时，应当始终使用该提供程序。
- Oracle.NET 数据提供程序。用于 Oracle 的 .NET 框架数据提供程序通过 Oracle 客户端连接软件支持对 Oracle 数据源的数据访问。该数据提供程序支持 Oracle 客户端软件版本 8.1.7 及更高版本。
- OLE DB .NET 数据提供程序。这是一个用于 OLE DB 数据源的托管提供程序。它的效率要比 SQL Server .NET 数据提供程序稍微低一些，因为它在与数据库通讯时通过 OLE DB 层进行调用。请注意，该提供程序不支持用于开放式数据库连接（ODBC）的 OLE DB 提供程序 MSDASQL。对于 ODBC 数据源，请改为使用 ODBC .NET 数据提供程序（稍后将加以介绍）。

- ODBC .NET 数据提供程序。用于 ODBC 的 .NET 框架数据提供程序使用本机 ODBC 驱动程序管理器 (DM) 来支持借助于 COM 互操作性进行的数据访问。还有其他一些目前正处于测试阶段的 .NET 数据提供程序。

与各个 .NET 数据提供程序相关联的类型 (类、结构、枚举等) 位于其各自的命名空间中:

- System.Data.SqlClient: 包含 SQL Server .NET 数据提供程序类型。
- System.Data.OracleClient: 包含 Oracle .NET 数据提供程序。
- System.Data.OleDb: 包含 OLE DB .NET 数据提供程序类型。
- System.Data.Odbc: 包含 ODBC .NET 数据提供程序类型。
- System.Data: 包含独立于提供程序的类型, 如 DataSet 和 DataTable。

在各自的关联命名空间内, 每个提供程序都提供了对 Connection、Command、DataReader 和 DataAdapter 对象的实现。SqlClient 实现的前缀为“Sql”, 而 OleDb 实现的前缀为“OleDb”。例如, Connection 对象的 SqlClient 实现是 SqlConnection, 而 OleDb 实现则为 OleDbConnection。同样, DataAdapter 对象的两个实现分别为 SqlDataAdapter 和 OleDbDataAdapter。

为了屏蔽不同数据库之间的差异, 我们首先要设计数据库访问的接口。把这个接口名为 DataAccess, 定义如下:

```
public interface DataAccess
{
    #region Support Property & Method
    DatabaseType DatabaseType {get;}
    IDbConnection DbConnection {get;}
    IDbTransaction BeginTransaction();
    void Open();
    void Close();
    bool IsClosed {get;}

    #endregion

    #region ExecuteNonQuery

    int ExecuteNonQuery(CommandType commandType, string commandText);
    int ExecuteNonQuery(string commandText);
    int ExecuteNonQuery(string commandText, QueryParameterCollection commandParameters);
    int ExecuteNonQuery(CommandType commandType, string commandText,
    QueryParameterCollection commandParameters);

    #endregion ExecuteNonQuery
}
```

//……因篇幅的原因，这里没有列出所有的方法，关于其他方法的定义请参见源代码。

}

在这个接口之下，定义 `AbstractDataAccess` 类，实现一些公用的数据方法，在 `AbstractDataAccess` 类之下，再扩展出各个具体的 `DataAccess` 实现类。整个结构可以用下面的图(图 3.1)来表示：

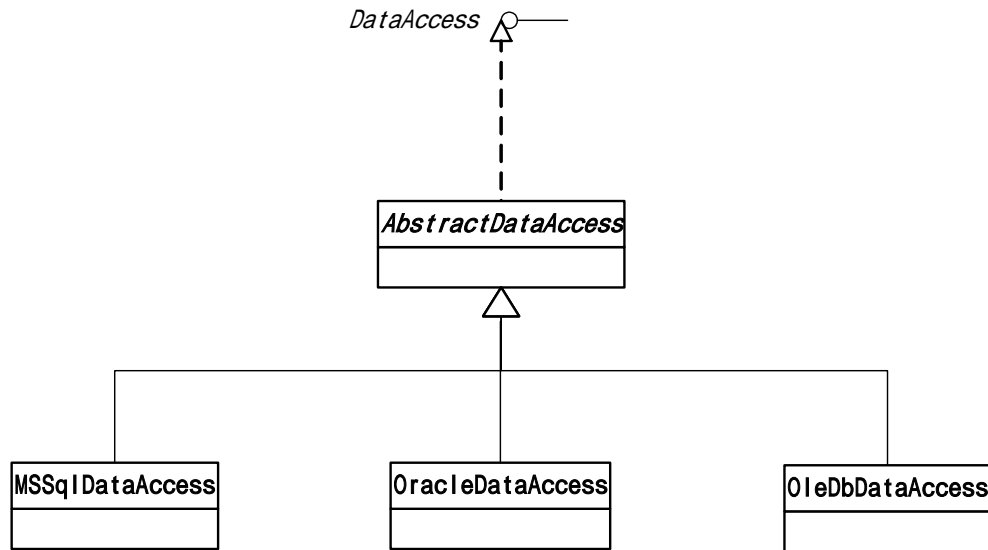


图 3.1

`DataAccess` 类的代码片断如下：

```

public abstract class AbstractDataAccess : DataAccess
{
    #region DataAccess

    #region Support Property & method
    public abstract DatabaseType DatabaseType {get;}
    public abstract IDbConnection DbConnection {get;}

    public void Close()
    {
        this.DbConnection.Close();
    }
    public void Open()
    {
        if(this.DbConnection.State.Equals(ConnectionState.Closed))
            this.DbConnection.Open();
    }
    .....
}
    
```

```
#endregion Support Property & method

#region ExecuteNonQuery
public int ExecuteNonQuery(CommandType commandType, string commandText)
{
    return this.ExecuteNonQuery(commandType, commandText, null);
}

public int ExecuteNonQuery(string commandText)
{
    return this.ExecuteNonQuery(CommandType.Text, commandText, null);
}

public int ExecuteNonQuery(string commandText, QueryParameterCollection commandParameters)
{
    return this.ExecuteNonQuery(CommandType.Text, commandText, commandParameters);
}

public abstract int ExecuteNonQuery(CommandType commandType, string commandText,
QueryParameterCollection commandParameters);
#endregion ExecuteNonQuery

protected void SyncParameter(QueryParameterCollection commandParameters)
{
    if((commandParameters!=null) && (commandParameters.Count>0) )
    {
        for(int i=0;i<commandParameters.Count;i++)
        {
            commandParameters[i].SyncParameter();
        }
    }
}
}
```

然后，我们可以实现具体的数据库访问的方法。例如，SQL Server 的数据库访问类可以实现如下：

```
public sealed class MSSqlDataAccess : AbstractDataAccess
{
    #region Constructor
    public MSSqlDataAccess(SqlConnection conn)
    {
        this.m_DbConnection=conn;
    }
}
```

```
public MSSqlDataAccess(string connectionString)
{
    this.m_DbConnection=new SqlConnection(connectionString);
}
#endregion

#region DataAccess

#region Support Property & method
public override DatabaseType DatabaseType
{
    get{return DatabaseType.MSSQLServer;}
}
private SqlConnection m_DbConnection;
public override IDbConnection DbConnection
{
    get
    {
        return m_DbConnection;
    }
}

private SqlTransaction trans=null;
public override IDbTransaction BeginTransaction()
{
    trans=m_DbConnection.BeginTransaction();
    return trans;
}

#endregion Support Property & method

#region ExecuteNonQuery
public override int ExecuteNonQuery(CommandType commandType, string commandText,
QueryParameterCollection commandParameters)
{
    SqlCommand cmd=new SqlCommand();
    PrepareCommand(cmd, commandType, commandText, commandParameters);
    int tmpValue=cmd.ExecuteNonQuery();
    SyncParameter(commandParameters);
    cmd.Parameters.Clear();
    return tmpValue;
}
}
```

```
#endregion ExecuteNonQuery

#region ExecuteDataSet

public override DataSet ExecuteDataset(CommandType commandType, string commandText,
QueryParameterCollection commandParameters, DataSet ds, string tableName)
{
    SqlCommand cmd=new SqlCommand();
    PrepareCommand(cmd, commandType, commandText, commandParameters);

    SqlDataAdapter da=new SqlDataAdapter(cmd);
    if(Object.Equals(tableName, null) || (tableName.Length<1))
        da.Fill(ds);
    else
        da.Fill(ds, tableName);

    SyncParameter(commandParameters);
    cmd.Parameters.Clear();
    return ds;
}

#endregion ExecuteDataSet

#region ExecuteReader

public override IDataReader ExecuteReader(CommandType commandType, string commandText,
QueryParameterCollection commandParameters)
{
    SqlCommand cmd=new SqlCommand();
    PrepareCommand(cmd, commandType, commandText, commandParameters);
    SqlDataReader dr=cmd.ExecuteReader();
    SyncParameter(commandParameters);
    cmd.Parameters.Clear();
    return dr;
}

#endregion ExecuteReader

#region ExecuteScalar

public override object ExecuteScalar(CommandType commandType, string commandText,
QueryParameterCollection commandParameters)
{
    SqlCommand cmd=new SqlCommand();
    PrepareCommand(cmd, commandType, commandText, commandParameters);
    object tmpValue=cmd.ExecuteScalar();
}
```

```
        SyncParameter(commandParameters);
        cmd.Parameters.Clear();
        return tmpValue;
    }
#endregion ExecuteScalar

#region ExecuteXmlReader
public override XmlReader ExecuteXmlReader(CommandType commandType, string commandText,
QueryParameterCollection commandParameters)
{
    SqlCommand cmd=new SqlCommand();
    PrepareCommand(cmd, commandType, commandText, commandParameters);
    XmlReader reader=cmd.ExecuteXmlReader();
    SyncParameter(commandParameters);
    cmd.Parameters.Clear();
    return reader;
}
#endregion ExecuteXmlReader

#endregion

private void PrepareCommand(SqlCommand cmd, CommandType commandType, string commandText,
QueryParameterCollection commandParameters)
{
    cmd.CommandType=commandType;
    cmd.CommandText=commandText;
    cmd.Connection=this.m_DbConnection;
    cmd.Transaction=trans;
    if((commandParameters!=null) && (commandParameters.Count>0) )
    {
        for(int i=0;i<commandParameters.Count;i++)
        {
            commandParameters[i].InitRealParameter(DatabaseType.MSSQLServer);
            cmd.Parameters.Add(commandParameters[i].RealParameter as SqlParameter);
        }
    }
}
}
```

现在，我们已经有了数据库访问的接口和具体的实现类，为了管理这些类，并且提供可扩展性，我们需要创建一个类来提供获取具体实现类的方法，这个类就是 Factory 类。这个类很简单，主要的功能就是根据参数，判断使用什么数据库，然后，返回适当的 DataAccess

类。这是典型的 Factory 设计模式。关于设计模式的更多资料，可以参考《设计模式——可复用面向对象设计基础》一书。

这个类的定义如下：

```
public sealed class DataAccessFactory
{
    private DataAccessFactory() {}
    private static DatabaseProperty defaultDatabaseProperty;
    public static DatabaseProperty DefaultDatabaseProperty
    {
        get{return defaultDatabaseProperty;}
        set{defaultDatabaseProperty=value;}
    }
    public static DataAccess CreateDataAccess(DatabaseProperty pp)
    {
        DataAccess dataAccess;
        switch(pp.DatabaseType)
        {
            case(DatabaseType.MSSQLServer):
                dataAccess = new MSSqlDataAccess(pp.ConnectionString);
                break;
            case(DatabaseType.Oracle):
                dataAccess = new OracleDataAccess(pp.ConnectionString);
                break;
            case(DatabaseType.OleDbSupported):
                dataAccess = new OleDbDataAccess(pp.ConnectionString);
                break;
            default:
                dataAccess=new MSSqlDataAccess(pp.ConnectionString);
                break;
        }
        return dataAccess;
    }
    public static DataAccess CreateDataAccess()
    {
        return CreateDataAccess(defaultDatabaseProperty);
    }
}
```

关于 DatabaseProperty 和 DatabaseType 的定义，可以参见相关源代码。

数据访问功能的调用形式如下：

```
.DataAccess dao=DataAccessFactory.CreateDataAccess(persistenceProperty);  
  
db.Open();  
  
db.需要的操作  
  
db.Close();
```

当数据库发生变化的时候，只需要修改相应的 DatabaseProperty 参数，DataAccessFactory 会根据参数的不同，自动调用相应的类，客户端不会感觉到变化，也不用去关心。这样，实现了良好的封装性。当然，前提是，你在编写程序的时候，没有用到特定数据库的特性，例如，Sql Server 的专用函数。

5.2 设计映射

映射部分，完成对象和关系型数据库之间映射关系的表达。前面探讨过，在 .Net 环境中，可以使用 Attribute 来描述。在 Websharp 框架中，我们设计了以下 Attribute 来描述对象和关系型数据库之间的映射。

➤ TableMapAttribute，

这个 Attribute 描述对象和数据库表的映射关系，这个类有两个属性，TableName 属性指明和某个类所对应的数据库表，PrimaryKeys 用来描述表的主关键字。这个类的定义如下：

```
[AttributeUsage(AttributeTargets.Class)]  
public class TableMapAttribute : Attribute  
{  
    private string tableName;  
    private string[] primaryKeys;  
    public TableMapAttribute(string tableName, params string[] primaryKeys)  
    {  
        this.tableName = tableName;  
        this.primaryKeys = primaryKeys;  
    }  
    public string TableName  
    {  
        get{return tableName;}  
        set{tableName = value;}  
    }  
  
    public string[] PrimaryKeys
```

```
{  
    get{return primaryKeys;}  
    set{primaryKeys = value;}  
}  
}
```

➤ ColumnMapAttribute

这个 Attribute 描述对象属性和数据库中表的字段之间的映射关系，这个类有三个属性，ColumnName 属性指明和某个属性所对应的字段，DbType 属性指明数据库字段的数据类型，DefaultValue 指明字段的默认值。这个类的定义如下：

```
[AttributeUsage(AttributeTargets.Property)]  
public class ColumnMapAttribute : Attribute  
{  
    private string columnName;  
    private DbType dbtype;  
    private object defaultValue;  
    public ColumnMapAttribute(string columnName, DbType dbtype)  
    {  
        this.columnName = columnName;  
        this.dbtype = dbtype;  
    }  
  
    public ColumnMapAttribute(string columnName, DbType dbtype, object defaultValue)  
    {  
        this.columnName = columnName;  
        this.dbtype = dbtype;  
        this.defaultValue = defaultValue;  
    }  
  
    public string ColumnName  
    {  
        get{return columnName;}  
        set{columnName = value;}  
    }  
  
    public DbType DbType  
    {  
        get{return dbtype;}  
        set{dbtype = value;}  
    }  
}
```

```
public object DefaultValue
{
    get{return defaultValue;}
    set{defaultValue = value;}
}
}
```

➤ ReferenceObjectAttribute

ReferenceObjectAttribute 指示该属性是引用的另外一个对象，因此，在执行持久化操作的时候，需要根据参数进行额外的处理。默认情况下，当持久化实体对象的时候，ReferenceObjectAttribute 指示的属性，不进行操作。这个类有三个属性，ReferenceType 指明所引用的对象的类型，PrimaryKey 和 ForeignKey 用来指明两个类之间进行关联的主键和外键。这个类的定义如下：

```
[AttributeUsage(AttributeTargets.Property)]
public class ReferenceObjectAttribute : Attribute
{
    private Type referenceType;
    private string primaryKey;
    private string foreignKey;
    public ReferenceObjectAttribute(Type referenceType, string primaryKey, string
foreignKey)
    {
        this.referenceType = referenceType;
        this.primaryKey = primaryKey;
        this.foreignKey = foreignKey;
    }

    public ReferenceObjectAttribute() {}

    public Type ReferenceType
    {
        get{return referenceType;}
        set{referenceType = value;}
    }

    public string PrimaryKey
    {
        get{return primaryKey;}
        set{primaryKey = value;}
    }
}
```

```
public string ForeignKey
{
    get{return ForeignKey;}
    set{ForeignKey = value;}
}
```

➤ SubObjectAttribute

SubObjectAttribute 指示该属性是引用的是子对象，因此，在执行持久化操作的时候，需要根据参数进行额外的处理。默认情况下，当持久化实体对象的时候，SubObjectAttribute 指示的属性，不进行操作。

这个类的定义如下：

```
[AttributeUsage(AttributeTargets.Property)]
public class SubObjectAttribute : Attribute
{
    private Type subObjectType;
    private string primaryKey;
    private string ForeignKey;
    public SubObjectAttribute(Type subObjectType, string primaryKey, string ForeignKey)
    {
        this.subObjectType = subObjectType;
        this.primaryKey = primaryKey;
        this.ForeignKey = ForeignKey;
    }

    public SubObjectAttribute() {}

    public Type SubObjectType
    {
        get { return subObjectType; }
        set { subObjectType = value; }
    }

    public string PrimaryKey
    {
        get{return primaryKey;}
        set{primaryKey = value;}
    }

    public string ForeignKey
```

```
{  
    get{return foreignKey;}  
    set{foreignKey = value;}  
}  
}
```

➤ AutoIncreaseAttribute

AutoIncreaseAttribute 指示该属性是自动增长的。自动增长默认种子为

这个类的定义如下：

```
[AttributeUsage(AttributeTargets.Property)]  
public class AutoIncreaseAttribute : Attribute  
{  
    private int step = 1;  
  
    public AutoIncreaseAttribute() {}  
  
    public AutoIncreaseAttribute(int step)  
    {  
        this.step = step;  
    }  
  
    public int Step  
    {  
        get{return step;}  
        set{step = value;}  
    }  
}
```

设计好映射的方法后，我们就可以来定义实体类以及同数据库之间的映射。下面是一个例子：

```
//订单类别  
[TableMap("OrderType", "ID")]  
public class OrderType  
{  
    private int m_ID;  
    private string m_Name;  
  
    [ColumnMap("ID", DbType.Int32)]  
    public int ID  
    {
```

```
        get { return m_ID; }
        set { m_ID = value; }
    }
    [ColumnMap("Name", DbType.String)]
    public string Name
    {
        get { return m_Name; }
        set { m_Name = value; }
    }
}

//订单
[TableMap("Order", "OrderID")]
public class Order
{
    private int m_OrderID;
    private OrderType m_OrderType;
    private string m_Title;
    private DateTime m_AddTime;
    private bool m_IsSigned;
    private List<OrderDetail> m_Details;

    [ColumnMap("OrderID", DbType.Int32)]
    [AutoIncrease]
    public int OrderID
    {
        get { return m_OrderID; }
        set { m_OrderID = value; }
    }
    [ReferenceObject(typeof(OrderType), "ID", "TypeID")]
    [ColumnMap("TypeID", DbType.String)]
    public OrderType OrderType
    {
        get { return m_OrderType; }
        set { m_OrderType = value; }
    }
    [ColumnMap("Title", DbType.String)]
    public string Title
    {
        get { return m_Title; }
        set { m_Title = value; }
    }
    [ColumnMap("AddTime", DbType.DateTime)]
```

```
public DateTime AddTime
{
    get { return m_AddTime; }
    set { m_AddTime = value; }
}
[ColumnMap("AddTime", DbType.Boolean)]
public bool IsDigned
{
    get { return m_IsSigned; }
    set { m_IsSigned = value; }
}
[SubObject(typeof(OrderDetail), "OrderID", "OrderID")]
public List<OrderDetail> Details
{
    get { return m_Details; }
    set { m_Details = value; }
}
}

//订单明细
public class OrderDetail
{
    private int m_DetailID;
    private int m_OrderID;
    private string m_ProductName;
    private int m_Amount;

    [ColumnMap("ID", DbType.Int32)]
    [AutoIncrease]
    public int DetailID
    {
        get { return m_DetailID; }
        set { m_DetailID = value; }
    }
    [ColumnMap("OrderID", DbType.Int32)]
    public int OrderID
    {
        get { return m_OrderID; }
        set { m_OrderID = value; }
    }
    [ColumnMap("ProductName", DbType.String)]
    public string ProductName
    {
```

```
    get { return m_ProductName; }
    set { m_ProductName = value; }
}
[ColumnMap("Amount", DbType.Int32)]
public int Amount
{
    get { return m_Amount; }
    set { m_Amount = value; }
}
}
```

Order 中

```
[ReferenceObject(typeof(OrderType), "ID", "TypeID")]
[ColumnMap("TypeID", DbType.String)]
public OrderType OrderType
{
    get { return m_OrderType; }
    set { m_OrderType = value; }
}
```

这段代码表明，OrderType 这个属性，引用了 OrderType 这个对象，同 OrderType 相关联的，是 OrderType 的主键 ID 和 Order 的外键 TypeID。

```
[SubObject(typeof(OrderDetail), "OrderID", "OrderID")]
public List<OrderDetail> Details
{
    get { return m_Details; }
    set { m_Details = value; }
}
```

这段代码表明，Details 这个属性，由子对象 OrderDetail 的集合组成，其中，两个对象通过 Order 类的 OrderID 主键和 OrderDetail 的外键 OrderID 相关联。

有了以上的类结构，我们可以为他们生成相应的数据库操作的 SQL 语句。在上面的三个对象中，分别对应的 SQL 语句是（以 SQL Server 为例）：

```
OrderType :
INSERT INTO OrderType (ID, Name) VALUES (@ID, @NAME)
UPDATE OrderType SET Name=@Name Where ID=@ID
DELETE FROM OrderType Where ID=@ID
```

```
Order :
```

```
INSERT INTO Order (TypeID, Title, AddTime, IsSigned) VALUES (@TypeID, @Title, @AddTime, @IsSigned) ; SELECT @OrderID=@@IDENTITY ‘其中@OrderID 为传出参数
```

```
UPDATE Order SET TypeID=@TypeID, Title=@Title, AddTime=@AddTime, IsSigned=@IsSigned WHERE OrderID=@OrderID
```

```
DELETE FROM Order WHERE OrderID=@OrderID
```

OederDetail :

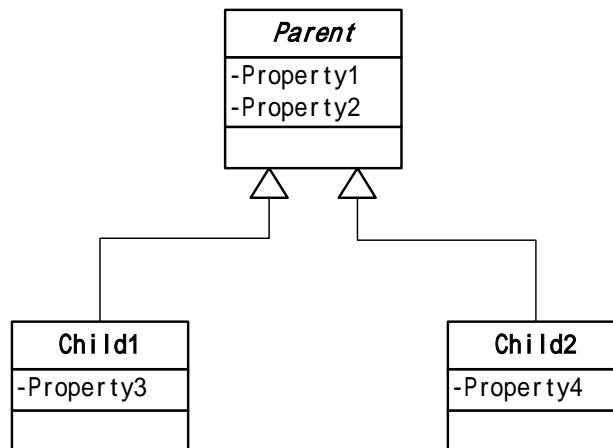
```
INSERT INTO OederDetail (OrderID, ProductName, Amount) VALUES (@OrderID, @ProductName, @Amount); SELECT @ID=@@IDENTITY ‘其中@ID 为传出参数
```

```
UPDATE OederDetail SET OrderID=@OrderID, ProductName=@ProductName, Amount=@Amount WHERE ID=@ID
```

```
DELETE FROM OederDetail WHERE ID=@ID
```

5.3 对继承的支持

Websharp 框架在设计的时候，要求能够支持面向对象语言中的继承。前面已经讨论过，在 O/R Mapping 框架中，一般说来，有三种继承模式：ONE_INHERITANCE_TREE_ONE_TABLE、ONE_INHERITANCE_PATH_ONE_TABLE 和 ONE_CLASS_ONE_TABLE。在 Websharp 框架中可以实现对这三种模式的支持。我们依然以前面的第三章第 3.2 节的例子为例：



ONE_INHERITANCE_TREE_ONE_TABLE

这种映射模式将具有相同父类的所有类都映射到一个数据库表中。数据库结构如下图：

Table1	
PK	Column1
	Column2
	Column3
	Column4

在 Websharp 中,只需要对每个类都指明具有相同值的 TableMap 特性就可以了。如下面的代码:

```
[TableMap("Table1", "Property1")]
public class Parent
{
    private string property1;
    private string property2;

    [ColumnMap("Column1", DbType.String)]
    public string Property1
    {
        get { return property1; }
        set { property1=value; }
    }

    [ColumnMap("Column2", DbType.String)]
    public string Property2
    {
        get { return property2; }
        set { property2 = value; }
    }
}

[TableMap("Table1", "Property1")]
public class Child1 : Parent
{
    private string property3;

    [ColumnMap("Column3", DbType.String)]
    public string Property3
    {
        get { return property3; }
        set { property3 = value; }
    }
}
```

```
[TableMap("Table1", "Property1")]
public class Child2 : Parent
{
    private string property4;

    [ColumnMap("Column4", DbType.String)]
    public string Property4
    {
        get { return property4; }
        set { property4 = value; }
    }
}
```

此时，当按照如下的代码初始化一个 Child1 对象，

```
Child1 c1 = new Child1();
c1.Property1 = "P11";
c1.Property2 = "P12";
c1.Property3 = "P13";
```

并保存到数据库中的时候，数据库中的记录应该是：

Column1	Column2	Column3	Column4
P11	P12	P13	NULL

如果按照如下的代码初始化一个 Child2 对象：

```
Child2 c1 = new Child2();
c2.Property1 = "P21";
c2.Property2 = "P22";
c2.Property4 = "P24";
```

并保存到数据库中的时候，数据库中的记录应该是：

Column1	Column2	Column3	Column4
P21	P22	NULL	P24

ONE_INHERITANCE_PATH_ONE_TABLE

这种映射模式将一个继承路径映射到一个表，这种情况下的数据库的结构是：

Child1	
PK	Column1
	Column2 Column3

Child2	
PK	Column1
	Column2 Column4

这种情况下，实际上 Parent 类并不映射到实际的表，Child1 和 Child2 类分别映射到 Child1 和 Child2 表。因此，在这种情况下，需要把 Parent 类的 TableMap 特性设置为 Null，而 Child1 和 Child2 类的 TableMap 特性分别设置为 Child1 和 Child2，代码如下面所示：

```
[TableMap(null, "Property1")]
public class Parent
{
    private string property1;
    private string property2;

    [ColumnMap("Column1", DbType.String)]
    public string Property1
    {
        get { return property1; }
        set { property1=value; }
    }

    [ColumnMap("Column2", DbType.String)]
    public string Property2
    {
        get { return property2; }
        set { property2 = value; }
    }
}

[TableMap("Child1", "Property1")]
public class Child1 : Parent
{
    private string property3;

    [ColumnMap("Column3", DbType.String)]
    public string Property3
    {
        get { return property3; }
        set { property3 = value; }
    }
}
```

```
}

[TableMap("Child2", "Property1")]
public class Child2 : Parent
{
    private string property4;

    [ColumnMap("Column4", DbType.String)]
    public string Property4
    {
        get { return property4; }
        set { property4 = value; }
    }
}
```

此时，当按照如下的代码初始化一个 Child1 对象，

```
Child1 c1 = new Child1();
c1.Property1 = "P11";
c1.Property2 = "P12";
c1.Property3 = "P13";
```

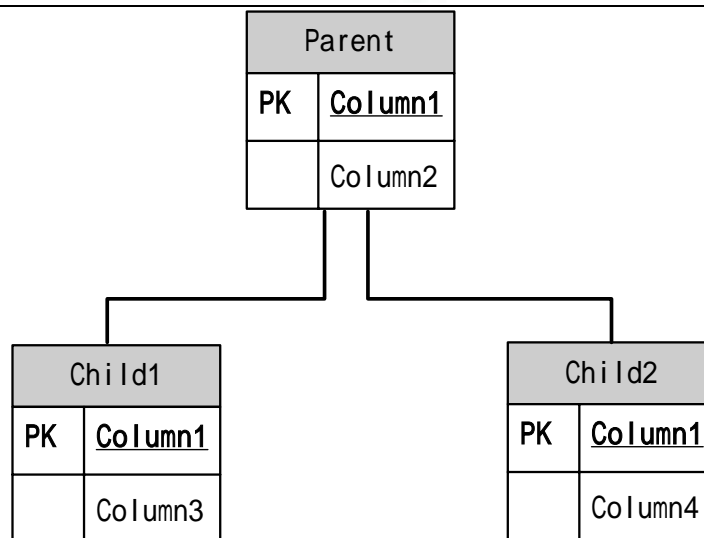
并保存到数据库中的时候，数据库中应该只在 Child1 表中添加下面的数据：

Column1	Column2	Column3
P11	P12	P13

如果保存的是一个 Child2 对象，那么，应该只在 Child2 表中添加数据。Child1 表和 Child2 表是互相独立的，并不会互相影响。

ONE_CLASS_ONE_TABLE

这种映射模式将每个类映射到对应的一个表，对于上面的类结构，数据库的结构是：



这种映射模式,我们只需要分别对每个类设定各自映射的表就可以了。代码如下面所示:

```
[TableMap("Parent", "Property1")]
public class Parent
{
    private string property1;
    private string property2;

    [ColumnMap("Column1", DbType.String)]
    public string Property1
    {
        get { return property1; }
        set { property1=value; }
    }

    [ColumnMap("Column2", DbType.String)]
    public string Property2
    {
        get { return property2; }
        set { property2 = value; }
    }
}

[TableMap("Child1", "Property1")]
public class Child1 : Parent
{
    private string property3;

    [ColumnMap("Column3", DbType.String)]
    public string Property3
```

```

    {
        get { return property3; }
        set { property3 = value; }
    }
}

[TableMap("Child2", "Property1")]
public class Child2 : Parent
{
    private string property4;

    [ColumnMap("Column4", DbType.String)]
    public string Property4
    {
        get { return property4; }
        set { property4 = value; }
    }
}

```

此时，当按照如下的代码初始化一个 Child1 对象，

```

Child1 c1 = new Child1();
c1.Property1 = "P11";
c1.Property2 = "P12";
c1.Property3 = "P13";

```

并保存到数据库中的时候，数据库中的记录应该是：

Parent 表：

Column1	Column2
P11	P12

Child1 表：

Column1	Column3
P11	P13

同样的，如果保存的是一个 Child2 对象，那么，将在 Parent 表和 Child2 表中添加记录。

5.4 设计对象操纵框架

由于我们采用了对象同操作分开的方式，因此，需要设计一个统一的接口来完成对对象的操纵。为了使用的方便性，我们尽量设计少的接口。我们设计以下三个主要接口：

- 1、PersistenceManager：这类完成所有对对象的增加、修改以及删除的操作，并提供

简单的查询功能。

- 2、Query：这个接口完成对对象的查询功能。
- 3、Transaction：这个接口负责处理事务。

另外，为了描述对象的状态，定义了 EntityState 枚举。为了简单化，这里只定义了四个状态，如下面的定义：

```
public enum EntityState {Transient, New, Persistent, Deleted}
```

对上面几个接口的说明分别如下：

➤ PersistenceManager

这个接口的定义如下：

```
public enum PersistOptions {SelfOnly, IncludeChildren, IncludeReference, Full}
public interface PersistenceManager : IDisposable
{
    void Close();
    bool IsClosed {get;}
    Transaction CurrentTransaction { get;}
    bool IgnoreCache {get;set;}

    void PersistNew(object entity);
    void PersistNew(object entity, PersistOptions options);

    void Persist(object entity);
    void Persist(object entity, PersistOptions options);
    void Persist(object entity, params string[] properties);
    void Persist(object entity, PersistOptions options, params string[] properties);

    void Delete(object entity);
    void Delete(object entity, PersistOptions options);

    void Attach(object entity);
    void Attach(object entity, PersistOptions options);

    void Reload(object entity);
    void Reload(object entity, PersistOptions options);

    void Evict (object entity);
    void EvictAll (object[] pcs);
    void EvictAll (ICollection pcs);
```

```
void EvictAll ();

object FindByPrimaryKey(Type entityType, object id);
object FindByPrimaryKey(Type entityType, object id, PersistOptions options);
T FindByPrimaryKey<T>(object id);
T FindByPrimaryKey<T>(object id, PersistOptions options);

object GetReference(object entity);
object GetReference(object entity, Type[] parents);
object GetChildren(object entity);
object GetChildren(object entity, Type[] children);

EntityState GetState(object entity);
ICollection GetManagedEntities();
bool Flush();

Query NewQuery();
Query NewQuery(Type entityType);
Query NewQuery(Type entityType, string filter);
Query NewQuery(Type entityType, string filter, QueryParameterCollection
paramCollection);

Query<T> NewQuery<T>();
Query<T> NewQuery<T>(string filter);
Query<T> NewQuery<T>(string filter, QueryParameterCollection paramCollection);
}
```

对于这个接口的几个主要方法说明如下：

`PersistNew` 方法将一个新的实体对象转换成可持续对象，这个对象在事务结束的时候，会被 `Insert` 到数据库中。调用这个方法后，该对象的状态为 `EntityState.New`。如果一个对象的状态为 `EntityState.Persistent`，那么，这个方法将抛出一个 `EntityIsPersistentException` 异常。

`Persist` 方法将一个实体对象保存到数据库中。如果一个对象是 `Transient` 的，则将其转换为 `EntityState.New` 状态。在事务结束的时候，会被 `Insert` 到数据库中；否则，其状态就是 `EntityState.Persist`，就更新到数据库中。如果一个 `Transient` 对象实际上已经存在于数据库中，由于 `Persist` 方法并不检查实际的数据库，因此，调用这个方法，将会抛出异常。这个时候，应该使用先使用 `Attach` 方法，然后调用 `Persist`。`Persist` 方法主要用于已受管的对象的更新。

`Delete` 方法删除一个对象。一个对象被删除后，其状态变成 `EntityState.Deleted`，在事务结束的时候，会被从数据库中删除。如果一个对象不是持久的，那么，这个方法将抛出异常。

Attach 方法将一个对象标记为可持续的。如果这个对象已经存在于实际的数据库中，那么，这个对象的状态就是 EntityState.Persistent，否则，这个对象的状态就是 EntityState.New。

Reload 方法重新从数据库中载入这个对象，这意味着重新给对象的各个属性赋值。

Evict 方法从缓存中把某个对象移除。

FindByPrimaryKey 方法根据主键查找某个对象，如果主键是多个字段的，主键必须是 PrimaryKey 数组，否则抛出异常。

➤ Query :

这个接口的定义如下：

```
public interface Query
{
    Type EntityType {get;set;}
    string EntityTypeName {get;set;}
    string Filter {get;set;}
    QueryParameterCollection Parameters {get;set;}
    string Ordering {get;set;}
    bool IgnoreCache {get;set;}
    PersistOptions Options { get;set;}

    ICollection QueryObjects();
    DataSet QueryDataSet();
    object GetChildren(object entity);
    object GetChildren(DataSet dst);
    object GetChildren(object entity, Type[] children);
    object GetChildren(DataSet entity, Type[] children);

    object GetReference(object entity);
    object GetReference(DataSet entity);
    object GetReference(object entity, Type[] parents);
    object GetReference(DataSet entity, Type[] parents);

    PersistenceManager PersistenceManager {get;}

    bool IsClosed {get;}
    void Close ();
    void Open();
}
```

```
public interface Query<T> : Query
{
    new ICollection<T> QueryObjects();
}
```

Query 接口的主要使用方法，是设定需要查询的对象的类型，以及过滤条件，然后执行 QueryObjects 方法，就可以得到相应的复合条件的对象。

➤ Transaction

这个接口主要用于处理事务，提供的功能比较简单，包括事务的开始、提交以及回滚三个主要功能。这个接口的定义如下：

```
public interface Transaction
{
    void Begin();
    void Commit();
    void Rollback();
    PersistenceManager PersistenceManager {get;}
}
```

定义好了接口，下面准备实现。这将在下面的小节中描述。

下面的例子展示了一个利用 Websharp 框架保存一个 Order 对象的过程：

```
DatabaseProperty dbp = new DatabaseProperty();
dbp.DatabaseType = DatabaseType.MSSQLServer;
dbp.ConnectionString = "Server=127.0.0.1;UID=sa;PWD=sa;Database=WebsharpTest;";
PersistenceManager pm = PersistenceManagerFactory.Instance().Create(dbp);

Order o = new Order();
o.OrderType = new OrderType(3, "音响");
o.OrderID = 3;
o.Title = "SecondOrder";
o.IsDigned = false;
o.AddTime = DateTime.Now;
o.Details = new List<OrderDetail>(2);

for (int j = 1; j < 3; j++)
{
    OrderDetail od= new OrderDetail();
    od.OrderID = 3;
    od.ProductID = j;
    od.Amount = j;
}
```

```
o.Details.Add(od);  
}  
  
pm.PersistNew(o, PersistOptions.IncludeChildren);  
  
pm.Flush();  
  
pm.Close();
```

5.5 实现对象操纵框架

前面，我们已经定义好了 O/R Mapping 的基本框架，下面，我们来具体讨论实现这个框架需要的一些主要工作。

在实现中，以下几个方面是比较主要的：

- MetaData
- StateManager
- SqlGenerator
- IEntityOperator

MetaData 用来记录对象和数据库之间映射的元数据信息，包括两个部分的元数据：

- 1、对象和表映射的信息
- 2、对象属性和字段映射的信息。

关于 MetaData 的定义可以参见 Websharp 的源代码。

MetaData 数据，由专门的类来进行解析，并进行缓存处理。在 Websharp 中，由 MetaDataManager 来完成这个任务。MetaDataManager 通过反射，读取实体类的信息，来得到 MetaData 数据。下面的代码片断演示了这个过程的主要内容。

首先，ParseMetaData 方法读取类的信息：

```
private static MetaData ParseMetaData(Type t, DatabaseType dbType)  
{  
    MetaData m = new MetaData();  
    m.ClassName = t.Name;           //类名  
    m.EntityType = t;              //实体类的类型  
    m.MapTable = GetMappedTableName(t); //实体类映射的表  
    m.PrimaryKeys = GetKeyColumns(t); //主键
```

```
if (m.PrimaryKeys.Length > 1)
    m.IsMultiPrimarykey = true;
else
    m.IsMultiPrimarykey = false;
.....
}
```

然后，读取每个字段的信息。这个部分的代码比较长，下面，只列出部分代码：

```
PropertyInfo[] pinfos = t.GetProperties();
m.FieldMetaDatas = new Dictionary<string, FieldMetadata>(pinfos.Length);

foreach (PropertyInfo pinfo in pinfos)
{
    FieldMetadata fd = new FieldMetadata();

    fd.PropertyName = pinfo.Name;
    ColumnMapAttribute cattr = Attribute.GetCustomAttribute(pinfo,
typeof(ColumnMapAttribute)) as ColumnMapAttribute;
    if(!Object.Equals(null, cattr))
    {
        fd.ColumnName = cattr.ColumnName;
        fd.DbType = cattr.DbType;
        fd.DefaultValue = cattr.DefaultValue;
    }
    else
    {
        fd.ColumnName = fd.PropertyName ;
        fd.DbType = DbType.String;
        fd.DefaultValue = String.Empty;
    }
}
.....
}
```

最后，根据映射信息，构建同数据库进行交互时候的 SQL 语句。O/R Mapping 框架的最后操作，还是回归到根据 SQL 语句来进行对数据库的操作。

```
SqlGenerator sg = SqlGenerator.Instance(dbType);

m.SelectSql = sg.GenerateSql(t, OperationType.SelectByKey);
```

SQL 语句的具体构建，由 SqlGenerator 来完成。

SqlGenerator 是一个抽象类，定义了构建同数据库进行交互的方法接口。在这个抽象类的下面，根据不同的数据库，扩展出针对不同数据库的 SQL 语句生成器。例如，一个 SQL Server 的 SqlGenerator 可以这样来生成插入一条记录需要的 SQL 语句：

```
private SqlStruct GenerateInsertSql(Type entityType)
{
    string autoP;
    bool autoInscreease = MetaDataManager.GetAutoInscreeaseProperty(entityType, out autoP);
    List<string> lcolumns = MetaDataManager.GetDbColumns(entityType);
    string[] parameters = new string[lcolumns.Count];
    ParamField[] paramField = new ParamField[lcolumns.Count];

    if (autoInscreease)
    {
        lcolumns.Remove(autoP);
    }
    string[] columns = lcolumns.ToArray();

    for (int i = 0; i < columns.Length; i++)
    {
        parameters[i] = "@" + columns[i];
        paramField[i] = new ParamField(parameters[i], columns[i]);
    }
    if (autoInscreease)
    {
        parameters[parameters.Length-1] = "@" + autoP;
        paramField[parameters.Length-1] = new ParamField(parameters[parameters.Length -
1], autoP);
    }

    string tableName = MetaDataManager.GetMappedTableName(entityType);
    StringBuilder strSql = new StringBuilder("INSERT INTO
").Append(tableName).Append("(").Append(string.Join(", ", columns)).Append("
VALUES(").Append(string.Join(", ", parameters)).Append(")");
    if (autoInscreease)
    {
        strSql.Append(";SELECT @").Append(autoP).Append("=@@IDENTITY");
    }
    return new SqlStruct(strSql.ToString(), paramField);
}
```

前面的章节讨论过对象的状态问题。在 Websharp 中，因为采用了普通的类就可以持久化的操作的方式，因此，需要另外的机制来管理对象的状态。

在 Websharp 中，为了简化，只定义了四种对象的状态，分别是 Transient, New, Persistent 和 Deleted，定义如下：

```
public enum EntityState {Transient, New, Persistent, Deleted}
```

在实现中，定义了 StateManager 类来管理对象的状态，这个类的定义如下：

```
public class StateManager
{
    public StateManager(object entity)
    {
        this.m_Entity = entity;
    }
    public StateManager(object entity, EntityState state)
    {
        this.m_Entity = entity;
        this.m_State = state;
    }
    private object m_Entity;
    public object Entity
    {
        get { return m_Entity; }
        set { m_Entity = value; }
    }

    private EntityState m_State;
    public EntityState State
    {
        get { return m_State; }
        set { m_State = value; }
    }
}
```

在 PersistenceManager 里面，持久化一个对象的时候，如果这个对象不是受管理的，则 PersistenceManager 会给这个对象分配一个 StateManager。例如，当对一个对象执行 PersistNew 操作的时候，PersistenceManager 将首先检查这个对象是否是受管理的，如果不是，则为这个对象分配一个 StateManager，并且其状态为 EntityState.New，然后，将这个对象添加到待操作列表中，在执行 Flush 方法的时候，会对这个对象执行一个新增的操作。代码如下：

```
public void PersistNew(object entity, PersistOptions options)
{
    //首先, 检查这个对象是否已经是受管理的对象
    StateManager smanager;
    if (IsManagedBySelf(entity, out smanager))
    {
        throw new EntityIsPersistentException();
    }

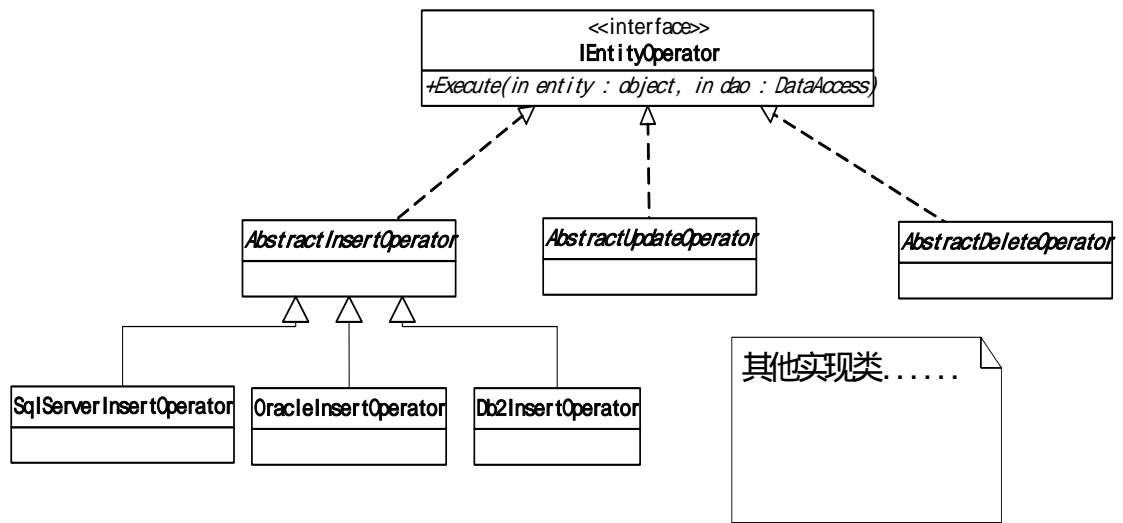
    //将对象标记为受管理的, 并且状态是 EntityState.New
    smanager = new StateManager(entity, EntityState.New);
    stateManagers.Add(smanager);

    //添加到操作列表中
    opEntityList.Add(new OperationInfo(smanager, options));
}
```

最后, 在执行 Flush 方法的时候, PersistenceManager 会把所有的对象的变化反应到数据库中。

```
foreach (OperationInfo opInfo in opEntityList)
{
    IEntityOperator op = EntityOperatorFactory.CreateEntityOperator(dao,
opInfo.StateManager.State);
    op.Execute(opInfo.StateManager.Entity, dao);
    CacheProxy.CacheEntity(opInfo.StateManager.Entity);
}
```

可以看到, 具体的对数据库的操作, 通过 IEntityOperator 接口来完成。IEntityOperator 接口定义了执行某个具体的对象同数据库进行交互的接口, 在这个接口的下面, 扩展出针对各个数据库的具体的实现类。这个部分的结构可以用下面的图来表示:



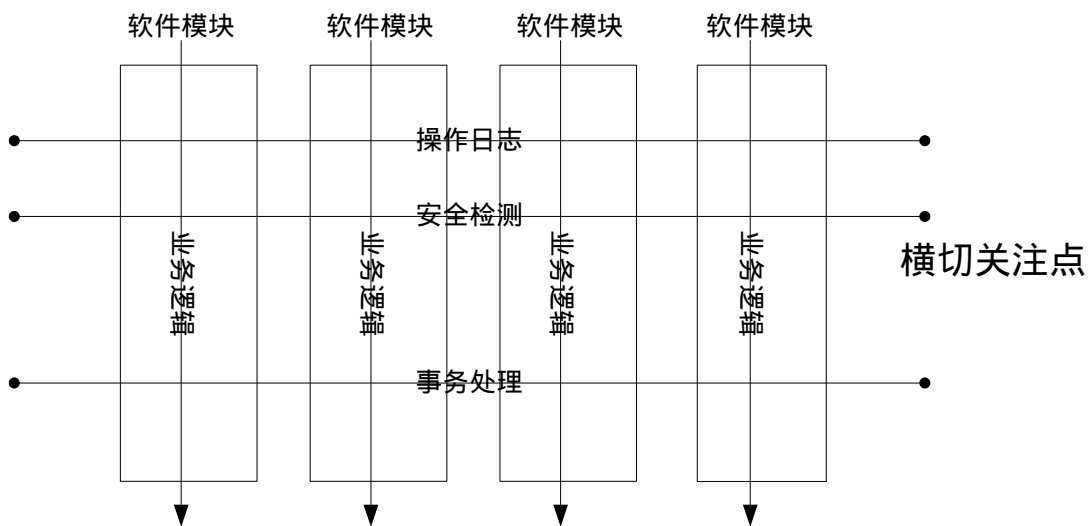
第6章 面向方面编程

6.1 AOP 概念

AOP 是 Aspect Oriented Programming 的简写，中文通常译作面向方面编程，其核心内容就是所谓的“横切关注点”。[17]

我们知道，使用面向对象方法构建软件系统，我们可以利用 OO 的特性，很好的解决纵向的问题，因为，OO 的核心概念，如继承等，都是纵向结构的。但是，在软件系统中，往往有很多模块，或者很多类共享某个行为，或者说，某个行为存在于软件的各个部分中，这个行为可以看作是“横向”存在于软件之中，他所关注的是软件的各个部分的一些共有的行为，而且，在很多情况下，这种行为不属于业务逻辑的一部分。例如，操作日志的记录，这种操作并不是业务逻辑调用的必须部分，但是，我们却往往不得在代码中显式进行调用，并承担由此带来的后果（例如，当日志记录的接口发生变化时，不得不对调用代码进行修改）。这种问题，使用传统的 OO 方法是很难解决的。AOP 的目标，便是要将这些“横切关注点”与业务逻辑代码相分离，从而得到更好的软件结构以及性能、稳定性等方面的好处。

纵向关注点



软件的纵向和横向结构

图 5.1

AOP 包含以下主要概念[18]：

- Aspect 方面：一个关注点的模块化，这个关注点实现可能另外横切多个对象。事务管理是 J2EE 应用中横切关注点中一个很好的例子。

- Joinpoint 连接点：程序执行过程中明确的点，如方法的调用或特定的异常被抛出。
- Advice 通知：在特定的连接点 AOP 框架执行的动作。各种类型的通知包括“around”、“before”和“throws”通知。通知类型将在下面讨论。许多 AOP 框架都是以拦截器做通知模型，维护一个“围绕”连接点的拦截器链。
- Pointcut 切入点：指定一个通知将被引发的一系列连接点的集合。AOP 框架必须允许开发者指定切入点：例如，使用正则表达式。
- Introduction 引入：添加方法或字段到通知化类。
- IsModified 接口，来简化缓存。
- Target object 目标对象：包含连接点的对象。也被用来引用通知化或代理化对象。
- AOP proxy AOP 代理：AOP 框架创建的对象，包含通知。
- Weaving 织入：组装方面创建通知化对象。这可以在编译时完成（例如使用 AspectJ 编译器），也可以在运行时完成。Spring 和其他一些纯 Java AOP 框架，使用运行时织入。

各种通知类型包括：

- Around 通知：包围一个连接点的通知，如方法调用。这是最强大的通知。Around 通知在方法调用前后完成自定义的行为。它们负责选择继续执行连接点或直接返回它们自己的返回值或抛出异常来短路执行。
- Before 通知：在一个连接点之前执行的通知，但这个通知不能阻止流程继续执行到连接点（除非它抛出一个异常）。
- Throws 通知：在方法抛出异常时执行的通知。
- After returning 通知：在连接点正常完成后执行的通知，例如，如果一个方法正常返回，没有抛出异常。

Around 通知是最通用的通知类型。大部分基于拦截器的 AOP 框架如 Nanning 和 JBoss4 只提供 Around 通知。

AOP，给我们的软件设计带来了一个新的视角和软件架构方法。使用 AOP，我们可以专注于业务逻辑代码的编写，而将诸如日志记录、安全检测等系统功能交由 AOP 框架，在运行时刻自动耦合进来。

通常，我们可以在如下情景中使用 AOP 技术：

- Authentication 权限
- Caching 缓存
- Context passing 内容传递

- Error handling 错误处理
- Lazy loading 懒加载
- Debugging 调试
- logging, tracing, profiling and monitoring 记录跟踪 优化 校准
- Performance optimization 性能优化
- Persistence 持久化
- Resource pooling 资源池
- Synchronization 同步
- Transactions 事务

Websharp 实现了一个基于 .Net 的轻量级的 AOP 框架。

6.2 Websharp AOP 的使用

6.2.1 使用 AOP 实现松散耦合

下面，我们通过一个例子来具体讨论 AOP 技术的应用。为了更好的说明这个问题，我们会给出部分代码，因此，需要选用一个具体的 AOP 框架。在这里，我们依然选用 Websharp 框架来进行说明。在 Websharo 框架中，我们也实现了一个 Aspect 的框架。

考虑如下情况：对于应用软件系统来说，权限控制是一个常见的例子。为了得到好的程序结构，通常使用 OO 的方法，将权限校验过程封装在一个类中，这个类包含了一个校验权限的代码，例如：

```
public class Security
{
    public bool CheckRight(User currentUser, Model accessModel, OperationType operation)
    {
        .....//校验权限
    }
}
```

然后，在业务逻辑过程中进行如下调用：

```
public class BusinessClass
{
    public void BusinessMethod()
    {
        Security s = new Security();

        if (!s. CheckRight(……))
        {
            return ;
        }

        ……//执行业务逻辑
    }
}
```

这种做法在 OO 设计中，是常见的做法。但是，这种做法会带来以下一些问题：

- 1、 不清晰的业务逻辑：从某种意义上来说，权限校验过程并不是业务逻辑执行的一部分，这个工作是属于系统的，但是，在这种情况下，我们不得不把系统的权限校验过程和业务逻辑执行过程掺杂在一起，造成代码的混乱。
- 2、 代码浪费：使用这种方法，我们必须所有的业务逻辑代码中用 Security 类，使得同样校验的代码充斥在整个软件中，显然不是很好的现象。
- 3、 紧耦合：使用这种方法，我们必须在业务逻辑代码中显式引用 Security 类，这就造成了业务逻辑代码同 Security 类的紧耦合，这意味着，当 Security 发生变化时，例如，当系统进化时，需要对 CheckRight 的方法进行改动时，可能会影响到所有引用代码。下面所有的问题都是因此而来。
- 4、 不易扩展：在这里，我们只是在业务逻辑中添加了权限校验，哪一天，当我们需要添加额外的功能，例如日志记录功能的时候，我们不得不同样在所有的业务逻辑代码中添加这个功能。
- 5、 不灵活：有的时候，由于某些特定的需要，我们需要暂时禁止，或者添加某项功能，采用传统的如上述的做法，我们不得不采用修改源代码的方式来实现。

为了解决这些问题，我们通常会采用诸如设计模式等方式，对上面的方案进行改进，这往往需要很高的技巧。利用 AOP，我们可以很方便的解决上述问题。

我们以 Websharp Aspect 为例，看看如何来对上面的代码进行改动，以获得一个更好的系统结构。

首先，Security 并不需要做任何修改。

然后，我们对 BusinessClass 做一个小小的改动：为 BusinessClass 添加一个名为 AspectManaged 的 Attribute，并使得 BusinessClass 继承 AspectObject，然后，删除代码中对 Security 的调用，这样，我们的代码就变成了如下的样子：

```
[AspectManaged(true)]  
  
public class BusinessClass : AspectObject  
{  
    public void BusinessMethod()  
    {  
        .....//执行业务逻辑  
    }  
}  
  
.....//执行业务逻辑
```

然后，我们为系统增加一个 SecurityAspect。

```
public class SecurityAspect : IAspect  
{  
    public void Execute(object[] paramList)  
    {  
        if(!Security.CheckRight(.....))  
        {  
            throw new SecurityException("你没有权限!");  
        }  
    }  
}
```

最后，我们在系统配置文件中添加必要的信息：

```
<Websharp.Aspects>  
  
    <Aspect type="MyAPP.SecurityAspect, MyAPP" deploy-model="Singleton"  
        pointcut-type="Method" action-position="before" match="*,*" />
```

```
</Websharp.Aspects>
```

这样，我们就完成了代码的重构。当 BusinessClass 被调用的时候，AOP 框架会自动拦截 BusinessClass 的 BusinessMethod 方法，并调用相应的权限校验方法。

采用这种方式，我们在 BusinessClass 中没有显式引用 Security 类及其相应方法，并且，在所有业务逻辑代码中，都没有必要引用 Security 类。这样，借助 AOP 机制，我们就实现了 BusinessClass 和 Security 类的松散耦合，上面列出的所有问题都迎刃而解了。同时，这也是一种易于扩展的机制，例如，当我们需要添加日志记录功能的时候，只需要添加相应的 Aspect 类，然后在配置文件中进行配置即可，而无需对业务逻辑代码进行任何改动。

6.2.2. 使用 AOP 组合两个业务逻辑

使用 AOP，我们不仅仅可以用来分离系统功能和业务逻辑，就象上面我们做的那样，也可以用来耦合不同的业务逻辑，得到更加灵活的软件结构。下面，我们通过一个具体的案例，来看看怎么通过 AOP，组合两个业务逻辑过程。

我们假设有如下一个场景：

我们设计了一个 ERP 系统，其中，库存管理系统需要同财务系统相交互，例如，当某个库存商品报废的时候，需要有相应的财务处理过程。因此，我们通常需要在库存商品报废的业务逻辑中引用相关的财务处理逻辑。这必然会造成两个部分的耦合。当然，为了使两个部分尽量耦合程度降低，我们通常会使用 Façade 等设计模式来进行解耦。

由于某些原因，我们需要将库存管理系统单独出售，这就需要我们将从库存商品报废的业务逻辑中将引用的相关的财务处理逻辑去除，这意味着我们需要修改原有的代码。为了解决这个问题，即可以随时将财务处理逻辑添加或者从库存商品报废的业务逻辑中删除，我们可以采用一些方法，例如，设置一些开关参数，在库存商品报废的业务逻辑中，根据这些开关参数的值，来判断是否需要执行财务处理逻辑。

问题是，这仍旧不是理想的解决方案。采用这种方式，你必须事先知道所有需要设置的开关参数，并且，在业务逻辑代码中添加相应的判断。当为系统增加一个类似的需要灵活处理的部分时，开发人员不得不添加相应的参数，并且修改相应的代码（添加相应的判断代码）。修改代码总是不好的事情，因为按照软件工程的要求，当有新的需求是，尽量不要修改原来的代码，而是新增相应的代码。但是，在这种情况下，你做不到。

使用 AOP，我们可以通过一种更加自然的方式来实现这个目标。基本方法如下：

首先，编写相关的库存商品报废业务逻辑，不需要添加任何其他的内容，并且，把这个逻辑的代码设置为可 AOP 的。

其次，按照正常的方式，编写财务处理逻辑。

添加一个把库存商品报废业务逻辑和财务处理逻辑组合起来的 Aspect，这个 Aspect 可以拦截库存商品报废业务逻辑的执行，动态的加入财务处理逻辑的过程，并且，在配置文件中进行配置。

这样，我们就通过一个 Aspect，组合了这两个业务逻辑。并且，我们随时可以通过修

改配置文件的方式把财务处理从库存商品报废业务逻辑中去除，而不用修改任何代码。

从上面的例子可以看出，采用 AOP 的好处是，我们可以独立的编写各个业务逻辑，使得系统各个部分之间的耦合度降到最低，然后，可以在系统中根据需要随时组合两个逻辑，而不用修改原来的任何代码。

6.3 Websharp AOP 的实现

应该认识到，完全的AOP实现，需要开发语言的支持。因为对于AOP的研究，还正在进行之中，目前的开发语言，都还没有完全支持AOP的，但是，我们可以利用现有的一些语言功能，来实现AOP的部分功能。

实现 AOP 的关键，是拦截正常的方法调用，将我们需要额外附加的功能透明的“织入”到这些方法中，以完成一些额外的要求。从总体方法上来说，织入的方法有两大类：静态织入和动态织入。

静态织入方法，一般都是需要扩展编译器的功能，将需要织入的代码，通过修改字节码（Java）或者IL代码（.Net）的方法，直接添加到相应的被织入点；或者，我们需要为原来语言添加新的语法结构，从语法上支持AOP。AspectJ^[19]就是采用的这种方式。使用这种方式来实现AOP，其优点是代码执行的效率高，缺点是实现者需要对虚拟机有很深的了解，才能够做到对字节码修改。由于织入方法是静态的，当需要添加新的织入方法时，往往需要重新编译，或者说运行字节码增强器重新执行静态织入的方法。当然，在.Net平台上，我们也可以使用Emit提供的强大功能来实现这一点。另外，字节码增强器带来了很大的不透明性，程序员很难直观的调试增强后的字节码，因此很多程序员总是在心理上抵制这种字节码增强器。

动态织入的方法，具体实现方式就有很多选择了。在 Java 平台上，可以使用 Proxy 模式，或者定制 ClassLoader 来实现 AOP 功能。在 .Net 平台上，要实现 AOP 的动态织入，归纳起来，可以采用以下几种方法：

- 使用 ContextAttribute 和 ContextBoundObject 来对对象的方法进行拦截。关于 ContextAttribute 的具体使用方法，读者可以参考 MSDN 等相关资料。
- 使用 Emit 来，在运行时刻动态构建被织入代码后的类，当程序调用被织入类时，实际上调用的是被修改后的类。LOOM 使用的就是这种方式，但是，个人认为，LOOM 目前的实现非常生硬，其可扩展性和灵活性都不是很好。
- 使用 Proxy 模式。这也是 Websharp 的实现方法。

Websharp的实现，是利用了对象代理(Proxy)机制。所谓Proxy，就是“为其他对象提供一种代理以控制对这个对象的访问”。可以用下面的图（图5.2）来表示Proxy模式：

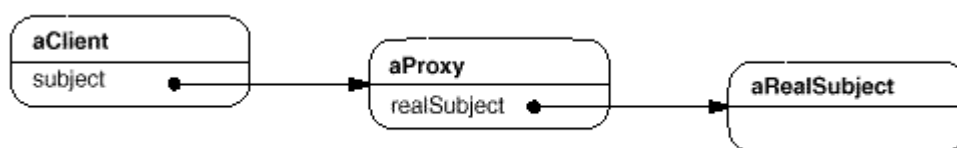


图5.2

关于Proxy模式的更多信息和资料，可以参见注解。

在WebsharpAspect中，当一个对象被标记为AspectManaged后，这个类的实例的创建过程，以及方法的调用会被WebsharpAspect控制。因此，当你在调用如下语句：

```
BusinessClass bc = new BusinessClass();
```

的时候，你得到的实际上并不是BusinessClass类的一个实例，而是他的一个代理（关于其中的实现机理，可以参见相关的源代码）。因此，当调用这个“实例”的方法的时候，所有的调用都会被代理所捕获，代理会在实际的方法调用之前，透明的执行一些预定义的操作，然后再执行实际的方法，最后，在实际的方法调用之后，再执行一些预定义的操作。这样，就实现了AOP的功能。

注意，AOP并不仅仅等同于方法调用拦截，当然，这也是最常用的和非常有效的AOP功能。

在Websharp AOP中，定义的主要类和接口如下：

6.3.1 AspectObject 抽象类

首先定义了抽象类AspectObject，所有需要AOP管理的类，都必须从这个类继承下来。这个类的定义如下：

```
public abstract class AspectObject : ContextBoundObject
```

之所以定义这个类，并且让它继承ContextBoundObject，其原因是因为，ContextBoundObject的子类，.Net运行环境会为其建立一个绑定的上下文，我们据此可以对其在运行时刻的行为做出一些自定义的控制。AspectObject继承ContextBoundObject，除此之外，没有给AspectObject添加其他属性和方法。实际上，WebsharpAspect能够拦截任何直接从ContextBoundObject派生的类，之所以定义AspectObject，目的是为了将来可能的扩充性。

当某个业务逻辑类需要接受Aspect管理的时候，必须继承AspectObject，并且加上AspectManaged特性。例如：

```
[AspectManaged(true)]  
public class BusinessClass : AspectObject  
{  
    .....  
}
```

关于AspectManaged特性，将来后面说明

6.3.2 IAspect 接口

IAspect 定义一个方面，这个方面可以在被拦截类的方法的执行之前或之后截获方法的执行，然后，执行相应的 Advice 通知的方法。Websharp AOP 的 IAspect 方法定义了 PreProcess 和 PostProcess 方法，以支持 Before 和 After 通知。IAspect 的定义如下：

```
public interface IAspect
{
    void PreProcess(IMessage msg);
    void PostProcess(IMessage msg);
}
```

6.3.3 AspectManagedAttribute

这是一个非常关键的类，其作用是拦截类的构造函数。如前所述，当你在执行诸如：

```
BusinessClass cls=new BusinessClass()
```

的时候，你实际上得到的不是 BusinessClass，而是 BusinessClass 的一个代理，正因为如此，我们才能够在执行这个对象的方法的时候，把他们拦截下来，插入我们自己的代码。

这个类的定义如下：

```
[AttributeUsage(AttributeTargets.Class)]
[SecurityPermissionAttribute(SecurityAction.Demand,
Flags=SecurityPermissionFlag.Infrastructure)]
public class AspectManagedAttribute : ProxyAttribute
{
    private bool aspectManaged;
    public AspectManagedAttribute()
    {
        aspectManaged=true;
    }
    public AspectManagedAttribute(bool AspectManaged)
    {
        aspectManaged=AspectManaged;
    }
}
```

在 AspectManagedAttribute 中，最重要的方法是 MarshalByRefObject 方法，你必须重载他，当我们拦截构造函数的时候，就会执行这个方法，在这里，我们可以对被拦截的构造函数的类进行一些处理，生成被实例化的类的代理，原理代码如下：

```
public override MarshalByRefObject CreateInstance(Type serverType)
{
```

```
MarshalByRefObject mobj= base.CreateInstance(serverType);
if(aspectManaged)
{
    RealProxy realProxy = new AspectProxy(serverType, mobj);
    MarshalByRefObject retobj = realProxy.GetTransparentProxy() as MarshalByRefObject;
    return retobj;
}
else
{
    return mobj;
}
}
```

在这里，我们为被拦截的类添加了一个名为 AspectProxy 的代理。这个代理的定义在下面讨论。

6.3.4 定义 AspectProxy 类

这个类的定义如下：

```
public class AspectProxy : RealProxy
```

这个类就是我们定义的 Proxy 类，对于方法的织入就是在这里进行的。当被代理的某个对象的方法执行时，就会被代理所拦截，代理会执行 Invoke 方法。我们所需要额外执行的方面代码就是在这里织入的。这部分的代码如下：

```
public override IMessage Invoke(IMessage msg)
{
    PreProcess(msg);
    IMessage retMsg;
    if (msg is IConstructionCallMessage)
    {
        IConstructionCallMessage ccm = (IConstructionCallMessage)msg;
        RemotingServices.GetRealProxy(target).InitializeServerObject(ccm);
        ObjRef oRef = RemotingServices.Marshal(target);
        RemotingServices.Unmarshal(oRef);
        retMsg = EnterpriseServicesHelper.CreateConstructionReturnMessage
            (ccm, (MarshalByRefObject) this.GetTransparentProxy());
    }
    else
    {
        IMethodCallMessage mcm = (IMethodCallMessage) msg;
        retMsg = RemotingServices.ExecuteMessage(target, mcm);
    }
}
```

```
}  
PostProcess(msg);  
return retMsg;  
}
```

可以看到，我们在这里，分别在方法执行前后执行后，进行了一些我们的处理。这些处理，就是根据配置文件，查找匹配的我们在前面定义的 `IAспект` 的派生类，并执行相应的 `Advice` 通知方法。

6.3.5 其他一些辅助类

上面的一些类完成了我们的 AOP 框架的主要功能，当然，我们还需要一些辅助来完成一些其他工作，例如，如何查找匹配的 `Aspect` 织入类，如何读取配置文件等，这些类的方法在这里就不一一列举了，可以参见源代码。

6.3.6 配置文件

配置文件的格式定义如下：

```
<?xml version="1.0" encoding="utf-8" ?>  
<configuration>  
  <configSections>  
    <section name="Websharp.Aspects"  
      type="Websharp.Aspect.AspectConfigHandler, Websharp.Aspect" />  
  </configSections>  
  
  <Websharp.Aspects>  
    <Aspect type="WeaveTest.FirstAspect, WeaveTest" deploy-model="None"  
      pointcut-type="Method|Construction|Property" action-position="Both"  
      match="*, Get*" />  
  </Websharp.Aspects>  
</configuration>
```

首先，需要在配置文件中添加一个配置节 (`configSections`)，在配置节中指明使用读取配置文件的类的细节。在 `Windows Form` 程序中，配置文件一般可以是 `app.config`，对于 `Web` 项目，可以在 `Web.config` 文件中添加配置信息。关于配置文件的其他详细信息，可以参考 `MSDN` 中相关的

在 `<Websharp.Aspects>` 节中，具体描述 `Aspect` 的信息。可以为一个系统添加多个 `Aspect`。

在 `Aspect` 配置中，各个属性的说明如下：

- ◆ `type` 属性说明 `Aspect` 类的类型，采用“`Aspect` 类全称, `Assembly`”的格式，分别说明 `Aspect` 类的类型，以及所在的 `Assembly`。

- ◆ `deploy-model` 属性指明 Aspect 类在运行时刻的行为，可以有 Singleton 和 None 两种属性值。当属性值是 Singleton 的时候，在系统中只有该 Aspect 类的一个实例，当属性值是 None 的时候，对于该类的每次调用，都会生成该类的实例。使用 Singleton 模式，可以得到性能上的好处
- ◆ `pointcut-type` 属性，指明该 Aspect 类拦截点的类型，可以是 Method、Construction、Property 三种，分别表示拦截方法，构造器和属性。可以使用“|”符号来指明多种类型的拦截点，例如：“Method|Construction”。
- ◆ `action-position` 指明拦截方相对于拦截点的位置，可以有 Before, After, Both 三个值，分别表示相对于拦截点的前面、后面还是前后都来执行 Aspect 类的方法。
- ◆ `match` 指明被拦截类的匹配方式，格式是“名称空间，类名”，例如，“MyNamespace, GetString”指明拦截 MyNamespace 名称空间下，名称为 GetString 的方法；又如“*, Get*”指明拦截所有名称空间下以 Get 开头的方法，他可以拦截诸如 GetString、GetName 等以 Get 开头的方法。

6.4 关于 AOP 和过滤器

在某些开发中，我们可能使用过滤器来完成某些 AOP 功能。例如，当用户访问某些资源时，我们可以对访问信息进行一些过滤处理。一个常见的场景是，在 JSP 开发中，为了实现对中文的正确处理，我们通常需要对浏览器同服务器之间传递的数据进行转码处理，以获得正确的文字编码。在每个 Request 中手工进行转码肯定不是一个好的解决方案。一个比较好的例子，是为应用程序编写一个 Filter，自动进行转码处理。例如，我们可以为 TOMCAT 写如下一个过滤器来实现转码：

```
public class SetCharacterEncodingFilter implements Filter
{
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
        throws IOException, ServletException
    {
        request.setCharacterEncoding( "gb2312" );
        chain.doFilter(request, response);
    }
}
```

这样，我们就不必在具体业务处理中进行手工的转码。实现业务逻辑同转码这样的系统功能的分离。

目前，常见的 Web 服务器都提供类似的机制。例如，在 IIS 中，也可以使用过滤器功能。传统的开发方式是使用 VC 开发一个 ISAPI Filter。在 .Net 发布后，可以使用 `HttpHandler` 和

HttpModule实现相同的功能，但是，开发难度要低很多^[20]。

使用过滤器的另外一个场景，可以是权限控制。例如，在客户请求某个 Web 页面的时候 (这个 Web 页面通常同某个业务功能相关联)，可以使用过滤器截获这个请求，然后，判断这个用户是否具备对请求资源的访问权限。如果是，那么，过滤器可以把这个请求放过去，什么都不做，否则，过滤器可以重定向到某个页面，告诉用户不能访问的原因，或者，直接抛出异常，交由前面的处理者处理。通过这种方式，我们可以同样的分离诸如身份验证这样的系统功能和业务逻辑，实现更好的系统结构。

通过象 Web 服务器这样的应用程序环境提供的功能，我们还可以实现其他一些 AOP 的功能，构建更好的系统框架。

6.5 小结

AOP 给了我们一个新的视角来看待软件的架构，有的时候，即使不使用 AOP 技术，只使用 AOP 的某些观念和现有的技术来搭建系统架构，分离某些本来是紧耦合的关注点，对我们也是非常有益的。

第7章 接口

软件由各个不同的功能模块所组成，各个不同的部分既相互独立，又互相影响。所以，在软件系统中，保证各个部分的独立性和互相无错调用就成了一对矛盾。

为了保证系统各个部分的独立性，高内聚、低耦合是系统设计必须遵循的基本准则。高内聚、低耦合的要求，不仅仅是在横向的各个模块之间的要求，也是系统纵向的各个层次之间设计所必须遵循的准则。关于系统层次设计的问题，在后面的应用软件系统架构部分会给与论述。

同时，为了保证各个部分的互相无错调用，必须对调用制定一系列规则，这些规则便是接口。因此，这里所指的接口，不是程序语言上的 `interface` 语法的含义，而是一个广泛的概念。

从本质上来说，接口就是一个契约，他规定了接口的实现者(服务提供者)和调用者(客户)之间的交互规范。接口的引入给系统设计带来了很大的灵活性，在对象、模块之间解耦、设计模式的运用等地方，都可以看到妙用接口的身影。可以比较夸张的说，接口技术是面向对象设计的灵魂所在。

“面向接口编程，而不是面向实现编程”²。为了使软件架构更加合理，同时，也为了团队之间的合作，以及后续软件开发工作的顺利展开，软件设计的第一个工作，便是为软件的各个横向和纵向的部分制定调用接口。

接口在服务的提供者和使用者之间签订了一个契约，将服务的具体实现封装起来，在保证双方各自独立的情况下，提供了功能调用的稳定性。当软件修改、升级或进行其他变动时，双方都可以各自独立的变化，但是，这个契约不能够变，这是软件稳定性的保证。

“面向接口编程，而不是面向实现编程”²。为了使软件架构更加合理，同时，也为了团队之间的合作，以及后续软件开发工作的顺利展开，软件设计的第一个工作，便是为软件的各个横向和纵向的部分制定调用接口。

特别注意的是，我们在设计基于数据库的应用系统的时候，当两个模块发生交互的需求时，应当避免使用直接在数据库中搬运数据的方法，而应当在程序中调用模块定义的接口方法，否则，对数据库的任何修改，都将带来噩梦般的问题。调用接口方法，而不是直接操纵数据库。

下面来看一个具体的例子：

这是一个 BBS 的例子。为了清楚的说明问题，让我们把问题尽量简化。现在设这个系统共有两个组成部分：用户登录信息维护和论坛发帖。

我们可以把这个系统划分成两个模块：UserSystem 和 ArticleSystem。

UserSystem 将会包含以下功能：

- 新用户注册
- 修改用户信息
- 用户登录

因此，UserSystem 的功能可以用代码表示如下：

```
public class UserSystem
{
    public void CreateNewAccount(Account account)
    public void UpdateAccount(Account account)
    public bool Login(Account account)
}
```

而 ArticleSystem 的功能比较简单，就是发贴，代码可以如下：

```
public class ArticleSystem
{
    public void CreateNewArticle(Article article)
}
```

因为在发送帖子的时候，需要知道当前发贴的人，会用到 UserSystem 中的用户信息，因此，UserSystem 必须提供这个访问接口，我们在 UserSystem 中添加一个属性：

```
public static Account GetCurrentLogAccount {}
```

这个 GetCurrentLogAccount 便是 UserSystem 为 ArticleSystem 提供的访问接口。

ArticleSystem 不需要知道 GetCurrentLogAccount 的具体实现方法，他只需要知道，当他需要使用用户登录信息的时候，调用这个方法就可以了。以后，无论 UserSystem 进行如何的变化，甚至改变数据库中 Account 表的设计，只要 GetCurrentLogAccount 这个调用接口不变，都不会影响 ArticleSystem。注意：千万不要在 ArticleSystem 中直接 SelectUserSystem 的 Account 表。

以上的接口，是一个广义上的接口的概念。至于语言层面的接口的妙用，在《设计模式》中，几乎到处都是。

上面所述，是应用系统横向的接口，下面，我们来看系统纵向接口的问题。

关于系统层次的问题，在后面会详细论述，在这里，为了说明问题，我们先做一个简要说明。

第8章 事务处理

8.1 事务的基本概念

事务处理在应用系统开发中起着重要的作用。简单地来说,事务就是由若干步处理组成的工作单元,这些步骤之间具有一定的逻辑关系,作为一个整体的操作过程,每个步骤必须同时成功或者失败。当所有的步骤都成功的时候,事务就必须“提交”,而当其中有一个步骤失败的时候,整个事务都必须“回滚”,也就是,其中的每一个步骤所做的改动都必须撤销。

举一个例子来说明事务处理的情形。我们考察一个仓库入库的过程。仓库的入库要经过以下步骤:

- 填写一张入库单,这张入库单上可能包含多种商品,每一种商品入库一定的数量
- 根据入库单上面的商品的数量,需要修改被入库的商品的现有库存量
- 保存这张入库单。

上面的操作都必须完整地执行,也就是说,必须在一个“事务”里面完成,否则,就可能出现下面的情形:

- 商品 1 在仓库里面原来有 100 件,这个时候,有一张入库单准备入库 50 件商品 1
- 系统根据入库单的信息,修改商品 1 的库存量,这个时候商品 1 的库存量便为 150 件
- 在保存入库单的时候,由于网络故障,入库单保存失败了!

此时的系统中,商品 1 的库存量已经改变了,已经变成了 150 件,而实际上,没有任何商品入库!所以,在这种情况下,系统的数据就出现了不一致性。显然,这种情况是不允许发生的。事务处理提供了防止发生这种情况的方法。

事务的主要特性包括原子性、一致性、隔离性和永久性,即所谓的 ACID 属性。

原子性

原子性表示事务作为一个工作单元处理。事务可能包括若干个步骤,通过把这些步骤作为一个工作单元处理,并在进程周围放上逻辑边界,就可以要求每一个步骤都完全成功,否则,就不能进行下一步的操作。

每一格步骤都依赖于前一步的顺利完成。如果一步失败,则不能完成任何其余的步骤,前面已经完成的步骤也必须全部回滚。如前面的例子,如果在保存入库单的时候出现问题,则所有的对商品现有库存量的修改,都必须恢复到初始的状态。

一致性

一致性将保证事务使系统或数据处于一致状态。数据处于一致状态就是符合数据库的限制或规则。限制就是事务完成时要成立的数据库条件，已在定义数据库结构时被定义，用以指定主键、数字字段有效范围和可否包含 null 值之类的事项。

如果事务开始时系统处于一致状态，则事务结束时系统应处于一致状态，不管事务成功还是失败，提交还是撤销。

一致性是由同一组业务规则或完整性限制确定的。为了保证一致性，需要事务管理者和应用程序开发人员共同努力。

事务管理者应保证事务的原子性、隔离性和持久性，应用程序开发人员应保证事务的一致性，指定主键、引用完整性限制和其他条件。当准备提交事务的时候，数据库根据这些规则验证其一致性。如果发现事务结果与系统确定的规则一致，则事务提交。如果结果不符合要求，则事务撤销。

一致性保证了事务所做的任何改变不会使系统处于无效状态。这样能使事务更加可靠。如果事务将钱从一个帐户转到另一个帐户，则只要系统中的总金额保持相同，事务就是一致的。

隔离性

事务的隔离性提供了事务处理中最强大的特性之一。简单地说，隔离性能保证事务访问的任何数据不会受其他事务所做的任何改变的影响，直到第一个事务完成。

这等于让事务象系统中的唯一事务一样执行。其他请求的数据库操作只能在不破坏当前使用的数据时才能进行。这对于支持数据的并发访问至关重要。

由于并发系统中很可能出现错误或搞乱数据的情况，因此保证隔离性和管理并发执行的责任不能由开发人员完成，而应该由系统来提供一个事务管理器。通常的数据库都支持事务管理，另外还有一些独立的事务管理服务器，如 COM+、J2EE 中的 JTA 等。

事务可以采用几种不同的隔离级，使用哪种隔离级取决于几种不同的因素。

永久性

永久性指定了事务提交时，对数据所做的任何改变都要记录到永久存储器中。这通常通过事务日志来实现。

事务日志将数据存放在数据库之类的资源中，可以在遇到故障时重新采用丢失的事务，也可以在遇到错误时撤销事务。简单的说，事务日志负责跟踪数据库中发生的每个数据操作，使数据可以返回到搞乱前的已知状态。一旦系统恢复到已知状态后，便可以利用事务日志更新构造或重新采用从这个状态开始的改变。

永久性在提交的数据改变是个关联合约时也很重要。事务提交表示数据源和应用程序之间的协议。事务日志就是这个协议的书面记录。更为重要的是，改变本身不是永久的，另一事务可能在此后改变数据。事务日志能提供可追查的依据。

8.2 实际开发中可用的事务处理方式

在实际的开发过程中，通常可以采用以下一些事务处理的方式：

- 使用数据库的事务处理
- 基于数据库连接的事务处理
- 使用事务处理服务器

使用数据库的事务处理

一般的关系型数据库都提供了处理事务的能力。在数据库中，调用事务处理的一般流程是：

```
BEGIN TRANSACTION
--进行处理
如果在处理的过程中发生错误，ROLLBACK TRANSACTION
如果一切正常，COMMIT TRANSACTION
END TRANSACTION
```

下面的代码演示了一个在 SQL Server 中在触发器中使用 T-SQL 进行事务处理的例子：

```
CREATE TRIGGER TestTrig ON TestTab FOR UPDATE AS
BEGIN TRANSACTION
INSERT INTO TrigTarget
    SELECT * FROM inserted
IF (@@error <> 0)
BEGIN
    ROLLBACK TRANSACTION
END
ELSE BEGIN
    COMMIT TRANSACTION
END
```

基于数据库连接的事务处理

常用的数据库编程接口，如 JDBC，ADO、ADO.Net，都提供了基于数据连接进行事务处理的功能。

下面，我们通过一个例子，来看看在 JDBC 中是怎么使用事务处理的。我们模拟一个银行转帐的过程。

有如下一个数据库表(Account)，表示银行的账户：

字段名称	数据类型
AccountID	int
Balance	Double

同时，我们在数据库中设定了一个约束条件:Balance>=0。这样，当任何对数据库的操作导致 Balance<0 的时候，都会引起异常。

可以通过下面的 SQL 语句来创建这个表：

```
CREATE TABLE Account(
    AccountID int,
    Balance double,
    Check (Balance>=0)
)
```

往数据库中添加如下的记录。

AccountID	Balance
1	300
2	1000

下面，我们来编写一个转帐的方法。

```
public boolean transferFunds(int fromAccount, int toAccount, double amount) throws SQLException
{
    const String withdrawSql = "UPDATE Account SET Balance=Balance-"+amount+" WHERE
AccountID="+fromAccount;
    const String depositSql = "UPDATE Account SET Balance=Balance"+amount+" WHERE
AccountID="+toAccount;
```

```
//首先获取数据库连接
Connection conn;
try
{
    Context initCtx = new InitialContext();
    javax.sql.DataSource ds = (javax.sql.DataSource) initCtx.lookup(dataSourceName);
    Connection cn = ds.getConnection();
}
catch(Exception e)
{
    return false;
}
boolean success = false;
try
{
    conn.setAutoCommit(false); //开始启动事务
    Statement stmt = conn.createStatement();
    stmt.executeUpdate(withdrawSql); //取钱
    stmt.executeUpdate(depositSql); //存钱
    conn.commit();
    success = true;
}
catch (SQLException e)
{
    conn.rollback();
    success = false;
}
finally
{
    conn.close();
}
return success;
}
```

上面的代码，如果我们调用 `transferFunds(1, 2, 200)`，表示从账户 1 转帐到账户 2，那么，数据库的结果就是：

AccountID	Balance
1	100
2	1200

这个时候，如果在调用一次 `transferFunds(1, 2, 200)` 的方法，那么，由于账户 1 的数据违反了 `Balance >= 0` 的约束条件，就会引发一个异常，所有的数据都不会被修改。

类似的，也可以使用 .Net 下面的编程语言来完成这项工作。下面的代码是用 C# 写的，可以看到，代码是非常类似的。

```
public bool TransferFunds(int fromAccount, int toAccount, double amount)
{
    const string connString = "....."; // 设定数据库连接字符串
    const string withdrawSql = "UPDATE Account SET Balance=Balance-" + amount + " WHERE
AccountID=" + fromAccount;
    const string depositSql = "UPDATE Account SET Balance=Balance+" + amount + " WHERE
AccountID=" + toAccount;
    // 首先获取数据库连接
    IDbConnection conn = null;
    IDbTransaction trans = null;
    try
    {
        conn = new GetConnection(connString); // 得到数据库连接
        conn.Open();
    }
    catch (Exception e)
    {
        return false;
    }
    boolean success = false;
    try
    {
        trans = conn.BeginTransaction();
        IDbCommand cmd = conn.CreateCommand();
        cmd.ExecuteNonQuery(withdrawSql); // 取钱
        cmd.ExecuteNonQuery(depositSql); // 存钱
        trans.Commit();
        success = true;
    }
    catch (SQLException e)
    {
        trans.Rollback();
        success = false;
    }
    finally
    {
        conn.Close();
    }
}
```

```
return success;

}

}
```

可以看出，使用基于连接的事务是很简单的，只要指定事务边界，执行数据库代码，然后提交或者撤销事务就可以了。

使用事务处理服务器

前面介绍了使用数据库本身的事务处理，以及使用基于数据库连接的事务。但是，当涉及到多个数据库事务的时候，上面的方式就力不从心了，特别是涉及到一些分布式处理的事务的时候，例如，在多个银行之间转帐。在这种情况下，我们通常需要一个事务处理服务器来协调处理这些事务。

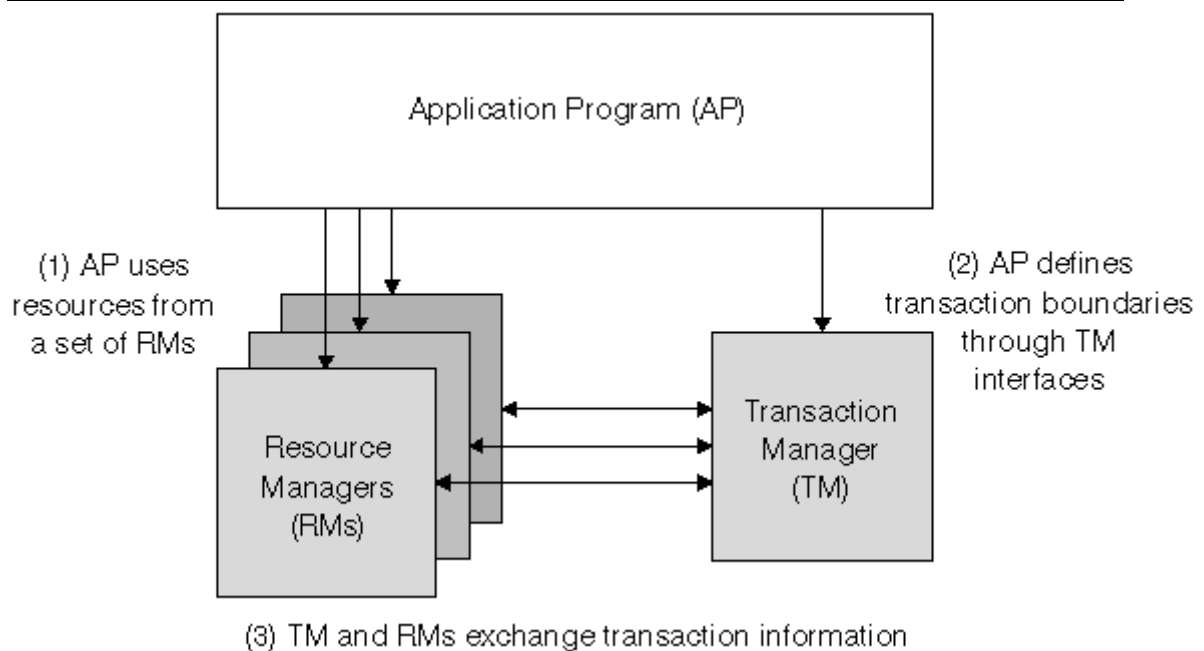
什么是分布式事务

分布式事务就是事务分布在多个资源上、由多个组件共享的事务。分布式事务具有如下的特征：

- 组件要在同一原子操作中与多个资源通信。在一个银行的系统中，资金可能从工商银行转帐到中国银行，这就涉及到两个银行系统。
- 多个组件要在同一原子操作中操作。资金从工商银行转帐到中国银行的时候，要么转帐成功，双方的帐户的资金都发生变化，要么失败，双方的帐户资金都没有变化，只有一方发生变化的情况是绝对不允许的。

分布式事务需要多个不同的事务管理器的合作。主事务管理器称为分布式事务管理器，与其他事务管理器协调。分布式事务管理器负责控制在若干本地事务管理器之间传递、分界和解析单个事务。本地事务管理器是参与分布式事务的事务管理器。

X/Open 组织(即现在的 Open Group)定义了分布式事务处理模型。X/Open DTP 模型(1994)包括应用程序(AP)、事务管理器(TM)、资源管理器(RM)、通信资源管理器(CRM)四部分。一般，常见的事务管理器(TM)是交易中间件，常见的资源管理器(RM)是数据库，常见的通信资源管理器(CRM)是消息中间件。



X/Open 分布式事务 (DTP) 模型

XA 与两阶段提交协议

XA 就是 X/Open DTP 定义的交易中间件与数据库之间的接口规范 (即接口函数), 事务管理服务器用它来通知数据库事务的开始、结束以及提交、回滚等。XA 接口函数由数据库厂商提供。

通常情况下, 事务管理服务器与数据库通过 XA 接口规范, 使用两阶段提交来完成一个全局事务, XA 规范的基础是两阶段提交协议。

在第一阶段, 事务管理服务器请求所有相关数据库准备提交 (预提交) 各自的事务分支, 以确认是否所有相关数据库都可以提交各自的事务分支。当某一数据库收到预提交后, 如果可以提交属于自己的事务分支, 则将自己在该事务分支中所做的操作固定记录下来, 并给事务管理服务器一个同意提交的应答, 此时数据库将不能再在该事务分支中加入任何操作, 但此时数据库并没有真正提交该事务, 数据库对共享资源的操作还未释放 (处于上锁状态)。如果由于某种原因数据库无法提交属于自己的事务分支, 它将回滚自己的所有操作, 释放对共享资源上的锁, 并返回给事务管理服务器失败应答。

在第二阶段, 事务管理服务器审查所有数据库返回的预提交结果, 如所有数据库都可以提交, 事务管理服务器将要求所有数据库做正式提交, 这样该全局事务被提交。而如果有任何一数据库预提交返回失败, 事务管理服务器将要求所有其它数据库回滚其操作, 这样该全局事务被回滚。

以一个全局事务为例, AP 首先通知事务管理服务器开始一个全局事务, 事务管理服务器通过 XA 接口函数通知数据库开始事务, 然后 AP 可以对数据库管理的资源进行操作, 数据库系统记录事务对本地资源的所有操作。操作完成后事务管理服务器通过 XA 接口函数通知数据库操作完成。事务管理服务器负责记录 AP 操作过哪些数据库 (事务分支)。AP 根据情

况通知事务管理服务器提交该全局事务，事务管理服务器会通过 XA 接口函数要求各个数据库做预提交，所有数据库返回成功后要求各个数据库做正式提交，此时一笔全局事务结束。

XA 规范对应用来说，最大好处在于事务的完整性由事务管理服务器和数据库通过 XA 接口控制，AP 只需要关注与数据库的应用逻辑的处理，而无需过多关心事务的完整性，应用设计开发会简化很多。

在 EJB 中使用分布式事务

在 EJB 中，容器提供了事务服务的功能，可以使用 Java 事务 API (JTA) 来访问事务服务，它提供对 Java 事务服务 (Java Transaction Service, JTS) 的简单访问。对于任何遵循 XA 规范的资源，JTA 的 XA 部分都有能力协调用这些资源的事务 (通过接口 `javax.transaction.xa.XAResource`)。J2EE 中支持 XA 的两个资源类型是 Java 数据库连接 (Java Database Connectivity, JDBC) API (通过接口 `javax.sql.XAConnection`) 和 Java 消息服务 (Java Message Service, JMS) API (通过接口 `javax.jms.XAConnection`)。

下面我们通过一个简单的例子来看看在 EJB 中是如何使用 JTA 的。

首先，我们需要设置两个数据源，分别针对 Oracle 和 DB2 数据库。JNDI 分别为 “jdbc/OracleXADS” 和 “jdbc/DB2XADS”。另外，配置一个 JMS 提供者，JNDI 名为 “jms/XAExampleQ”。编写 XAExampleSessionBean 并添加下面的代码：

```
public void persistAndSend(String data) throws Exception {
    try {
        DataSource oracleDS = getDataSource("java:comp/env/jdbc/OracleXADS");
        persist(data, oracleDS);

        DataSource db2DS = getDataSource("java:comp/env/jdbc/DB2XADS");
        persist(data, db2DS);

        QueueConnectionFactory factory =
            getQueueConnectionFactory("java:comp/env/jms/XAExampleQCF");
        Queue queue = getQueue("java:comp/env/jms/XAExampleQ");
        send(data, factory, queue);
    }
    catch (Exception e) {
        e.printStackTrace();
        this.getSessionContext().setRollbackOnly();
        throw e;
    }
}
```

该方法执行以下三个步骤：

1. 使用名为 jdbc/OracleXADS 的数据源保持数据。

2. 使用名为 jdbc/DB2XADS 的数据源保持数据。
3. 将 JMS 消息中的数据发送到名为 jms/XAExampleQ 的队列。

如果发生任何错误，批处理块将捕捉该异常，将事务标记为回滚，并重新抛出该异常。为了代码的完整性，以下列出了实现该会话 bean 的其余代码：

```
private void persist(String data, DataSource datasource) throws SQLException {
    System.out.println("Adding a new database row containing: " + data);
    Connection connection = null;
    try {
        connection = datasource.getConnection();
        PreparedStatement statement = connection.prepareStatement(
            "INSERT INTO XA_EXAMPLE (TEXT) VALUES (?)");
        statement.setString(1, data);
        statement.execute();

        System.out.println("Successfully added row: " + data);
    }
    finally {
        if (connection != null)
            connection.close();
    }
}

private void send(String data, QueueConnectionFactory factory, Queue queue)
    throws JMSEException {
    System.out.println("Sending a message containing: " + data);
    QueueConnection connection = null;
    try{
        connection = factory.createQueueConnection();
        QueueSession session = connection.createQueueSession(
            false, Session.AUTO_ACKNOWLEDGE);
        QueueSender sender = session.createSender(queue);
        TextMessage message = session.createTextMessage(data);
        sender.send(message);
        System.out.println("Successfully sent message: " + data);
    }
    finally {
        if (connection != null)
            connection.close();
    }
}
```

```
private DataSource getDataSource(String jndiName) throws NamingException {
    return (DataSource) this.getJNDIObject(jndiName);
}

private QueueConnectionFactory getQueueConnectionFactory(String jndiName)
throws NamingException {
    return (QueueConnectionFactory) this.getJNDIObject(jndiName);
}

private Queue getQueue(String jndiName) throws NamingException {
    return (Queue) this.getJNDIObject(jndiName);
}

private Object getJNDIObject(String jndiName) throws NamingException {
    Context root = new InitialContext();
    return root.lookup(jndiName);
}
```

这是用于使用 JDBC 和 JMS 的标准代码。甚至没有任何代码用于定义事务或者将事务变为 XA。但由于 EJB 容器将这些代码作为单独的公共 EJB 方法来调用，所以容器会在事务中自动运行这些代码。容器将正常使用简单(单阶段)事务，但当第二个事务资源更新时，容器将自动检测，并将简单事务转换为 XA 事务。作为该工作的一部分，容器将通知资源并为其协调 XA 事务，让这些资源参与该事务。所有这些行为都是自动完成的，不需要开发人员编写任何额外的代码，而仅仅通过实现 EJB 中的代码就可以完成。由此我们也可以看出使用事务管理服务器可以给我们的开发带来的便利。

Websharp 中的事务处理

Websharp 中提供了基本的事务处理功能，这个通过 Transaction 接口来完成。这个接口前面已经给出了，在这里再描述一遍。

```
public interface Transaction
{
    void Begin();
    void Commit();
    void Rollback();
    PersistenceManager PersistenceManager {get;}
}
```

在设计上，一个 Transaction 依附于一个 PersistenceManager，因此，他只能处理简

单的事务，而不能处理分布式事务。这也是由于 Websharp 框架式一个轻量级的框架，只提供一一些基本的服务。

下面的代码描述了使用 Websharp 进行事务处理的过程：

```
DatabaseProperty dbp = new DatabaseProperty();
dbp.DatabaseType = DatabaseType.MSSQLServer;
dbp.ConnectionString = "Server=127.0.0.1;UID=sa;PWD=sa;Database=WebsharpTest;";
PersistenceManager pm = PersistenceManagerFactory.Instance().Create(dbp);

Order o = new Order();
o.OrderType = new OrderType(3, "音响");
o.OrderID = 3;
o.Title = "SecondOrder";
o.IsDigned = false;
o.AddTime = DateTime.Now;
o.Details = new List<OrderDetail>(2);
for (int j = 1; j < 3; j++)
{
    OrderDetail od = new OrderDetail();
    od.OrderID = 3;
    od.ProductID = j;
    od.Amount = j;
    o.Details.Add(od);
}

Transaction t = pm.CurrentTransaction;
t.Begin(); //开始一个事务
try
{
    pm.PersistNew(o, PersistOptions.IncludeChildren);
    pm.Flush();
    t.Commit();
}
catch (Exception e)
{
    Console.WriteLine(e.ToString());
    t.Rollback();
}
finally
{
    pm.Close();
}
```

前面说过，Websharp 的 Transaction 接口只能完成简单的事务处理功能，复杂的事务，例如，涉及到两个数据库的事务，可以利用 .Net Framework 的 EnterpriseService (COM+) 来完成。

COM+事务

COM+ 是一个对象运行时环境，提供了一套服务旨在简化可伸缩分布式系统的创建过程。 .NET 企业服务是基于 COM+提供的服务的，或者说，COM+现在是 .NET 企业服务的一部分。

所有的 COM+ 服务都是依据上下文概念实现的。上下文是指一个进程中的空间，可以为驻留在其中的一个或者多个对象提供运行时服务。当位于一个上下文中的对象（或者线程）调用另一个上下文中的对象时，方法将被一个代理侦听，如图 所示。代理为 COM+ 运行时提供了机会，可以对调用进行预处理和后处理，并执行必要的服务代码以满足目标对象的需要，例如调用序列化、声明性事务管理等等。

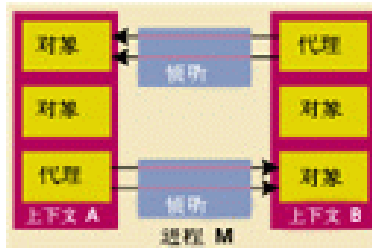
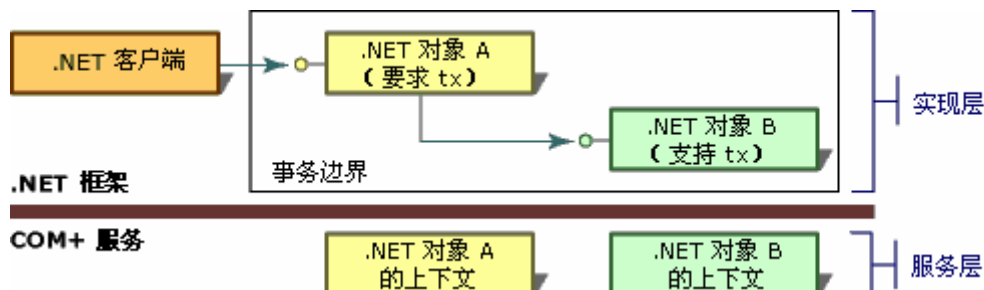


图 上下文与侦听

COM+ 服务（例如自动事务处理或排队组件）都是可以声明方式配置的。在设计时应用与服务相关的属性，并创建使用这些服务的类的实例。配置某些服务的途径是在与服务相关的类或接口上调用方法。某些服务可以从一个对象流到另一个对象。例如，配置为要求事务的对象可以在第二个对象也支持或要求事务的情况下，将事务扩展到第二个对象。

COM+ 目录中包含配置信息，可以将该配置信息应用于类的实现。在运行时，COM+ 会根据您为代码提供的属性来创建上下文服务层。下面的插图显示了在 COM+ 承载的两个托管对象之间流动的自动事务处理。



承载服务组件的 COM+ 应用程序

服务还可以在 COM+ 和 .NET Framework 对象之间流动。每一个环境都控制其本机代码的实现和执行；COM+ 总是提供对象上下文。

如同 COM 一样，CLR 依赖 COM+ 提供了对构建可伸缩应用程序的开发人员有用的运行时服务。用 CLR 来实现 COM+ 已配置类比使用 COM 来实现它们更容易，并且在某些情况下更有效。这两种技术的集成并不只是通过与 COM 的互操作性才能达到，理解这一点非常重要。也就是说，在可以使用 CLR 实现使用 COM+ 的传统 COM 组件时，CLR 与 COM+ 之间的集成程度实际上已经深入多了，这产生了一种可以与其他 CLR 技术（如远程处理和代码访问安全性）更好地集成的编程模型。COM+ 的 CLR 托管 API 通过 System.EnterpriseServices 命名空间的类型定义。依赖于 COM+ 的 CLR 类使用这些类型定义它们的声明性属性，同对象和调用上下文交互。

System.EnterpriseService 命名空间中最重要类型是 ServicedComponent。所有使用 COM+ 运行时服务的 CLR 类必须扩展 ServicedComponent，如下所示：

```
using System.EnterpriseServices;
namespace ESExample
{
    [Transaction(TransactionOption.Required)]
    public class MyCfgClass : ServicedComponent
    {
        [AutoComplete]
        static void Main() {}
    }
}
```

在这段代码中，Transaction 属性的存在指示了 MyTxCfgClass 需要使用一个 COM+ 托管的分布式事务。

当已配置类实现后，它必须被编译。编译代码是容易的，但需要牢记两件事情。首先，COM+ 集成基础结构要求被编译的程序集具有强名称。为了创建一个具有强名称的程序集，必须通过运行强名称实用工具 sn.exe，生成一个密钥。接着必须在您的组件代码中使用一个来自于 System.Reflection 命名空间称为 AssemblyKeyFileAttribute 的程序集级别属性来引用该密钥，它被存储在一个文件中，如下面的代码所示：

```
using System.EnterpriseServices;

using System.Reflection;

[assembly:ApplicationName("MyApp")]

[assembly:ApplicationActivation(ActivationOption.Library)]

// AssemblyKeyFile attribute references keyfile generated
// by sn.exe - assembly will have strong name

[assembly:AssemblyKeyFile("keyfile")]

namespace ESExample
```

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

其次，在编译具有强名称的程序集时，必须引用导出 System.EnterpriseServices 命名空间中类型的程序集 System.EnterpriseServices.dll。下面给出的是生成一个密钥以及编译一个已配置类所需要的命令：

```
sn -k keyfile csc /out:ESEExample.dll  
  
/t:library/r:System.EnterpriseServices.dll MyCfgClass.cs
```

在一个基于 CLR 的已配置类已经编译后，就需要部署它的程序集了。可以通过从命令行运行服务安装实用工具 regsvcs.exe 来完成，如下所示：

```
regsvcs ESEExample.dll
```

该工具完成三件事情。首先，它将 CLR 程序集作为一个 COM 组件注册（如同已经运行了程序集注册实用工具 regasm.exe）。其次，它生成一个 COM 类型库（如同已经运行了程序集到类型库转换器 tlbexp.exe）并且使用它来部署在 COM+ 编录中程序集实现的已配置类。Regsvcs.exe 在默认情况下将创建程序集的 ApplicationName 与 ApplicationActivation 属性所描述的目标应用程序。（也可以使用命令行开关重写这种行为。）第三，它使用 .NET 反射 API 来询问程序集实现的已配置类的元数据，并使用该信息编程更新 COM+ 编录，使每个类都将有相应的声明性属性设置

一旦一个基于 CLR 的已配置类编译和部署完成，就可以使用它了。从客户端的视角看，已配置类并没有什么特殊的地方；它使用 COM+ 运行时服务这一事实是无关紧要的。下面的代码显示了一个使用前面所定义的 MyTxCfgClass 类的简单客户端：

```
using ESEExample;  
  
public class Client  
{  
  
    public static void Main(string[] args)  
    {  
  
        MyTxCfgClass tcc = new MyTxCfgClass();  
  
        ... // use object as desired  
  
    }  
  
}
```

当然，对于所有的 CLR 代码，当编译客户端时必须提供一个指向已配置类程序集的引用。

此时可以看出，用 .NET CLR 实现 COM+ 已配置类是相当简单的。

System.EnterpriseServices 命名空间中的类型提供了 COM+ 的一个托管 API ,从而简化了运行时服务的使用。

以上只是简单的介绍了在 .Net 中使用 COM+事务的基本概念 ,更多的关于 COM+事务的知识 ,可以参考微软的相关资料。

第9章 性能优化

第三部分 用户界面层设计

第10章 界面层的功能划分

用户界面，承担着向用户显示问题模型和与用户进行操作和 I/O 交互的作用。用户界面的设计，包含技术和非技术的问题。

非技术方面，程序的界面，涉及到美工、心理学甚至社会学的内容，向客户提供方便的使用界面，是系统界面设计的一个重要内容。在某些宗教国家，甚至还可能涉及到一些宗教的敏感内容。这些都超出了本书要讨论的范围，在此就不再赘述了。

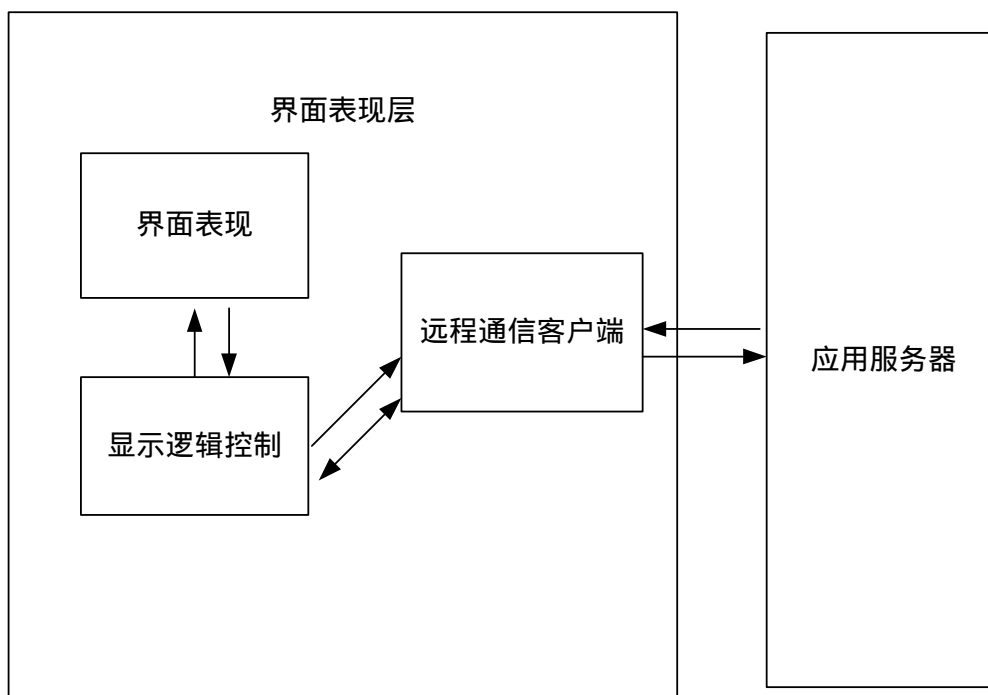
从技术的角度来看，同应用服务层一样，界面层的结构也应该是一个可维护的、可扩展的结构，在很多方面，界面层的设计原则和应用服务层的设计原则是一样的。

我们可以对用户界面这一层所要完成的功能来做一个细分。

首先，用户界面层，要完成界面表示，并同用户进行交互，接受输入和输出。

其次，根据某种条件，或者某个流程，用户界面之间在进行切换的时候，有一定的逻辑，我们不妨称之为显示逻辑。用户界面层的框架应当能够管理这种显示逻辑。尤其是在基于 Web 的应用系统的开发中，由于浏览器和服务器之间的无状态连接，以及页面增删的灵活性（非编译的应用程序），页面之间在进行切换的时候，涉及到系统的稳定性，以及页面间数据传递的问题，用户界面层的框架应当能够在这个方面进行有效的管理。

第三，对于一个 N 层的应用，业务逻辑的处理和计算，都在中间层，即应用服务层完成。应用服务层可能通过多种方式向界面层提供服务，这种方式表现为各种远程通信协议（如果是分布式应用系统的话）。因此，在用户界面层，必须有相应的模块，负责同应用服务器的通信。这个层面作为用户界面层和应用服务层的联系纽带。



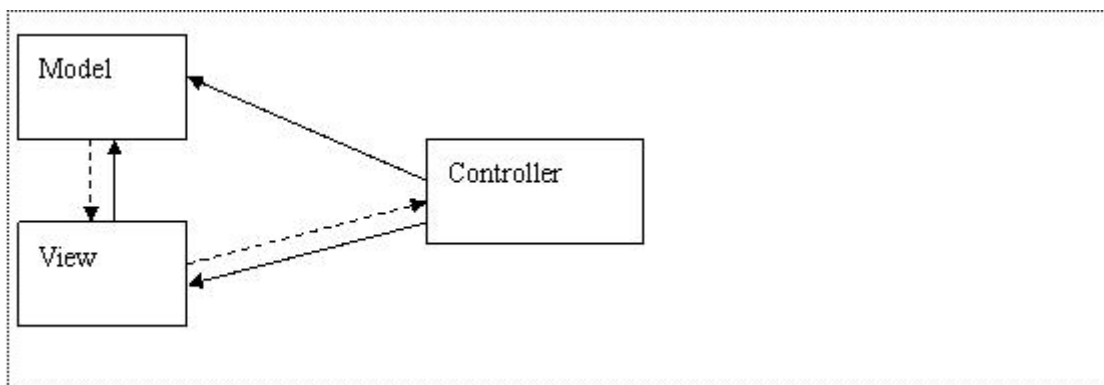
第11章 界面设计模式

在设计应用软件系统的界面层的时候，我们同样要考虑界面的可升级性和可维护性。为了达成这个目标，我们同样会使用一些设计模式来使得应用程序界面层的设计更为合理。

11.1 MVC 模式

第一个最著名的设计模式就是 MVC 模式。MVC 是三个单词的缩写，这三个单词分别为：模型 (Model)、视图 (View) 和控制 (Controller)。

模型 - 视图 - 控制器 (Model-View-Controller, MVC) 模式是一种非常常用的模式，他主要为那些需要为同样的数据提供多个视图的应用程序而设计的。它很好地实现了数据层与表示层的分离，特别适用于开发与用户图形界面有关的应用程序，其示意图见图。



模式中基本结构定义为：

- 控制器：用来处理用户命令以及程序事件的；
- 模型：维护数据并提供数据访问方法；表示领域信息；
- 视图：数据的显示。

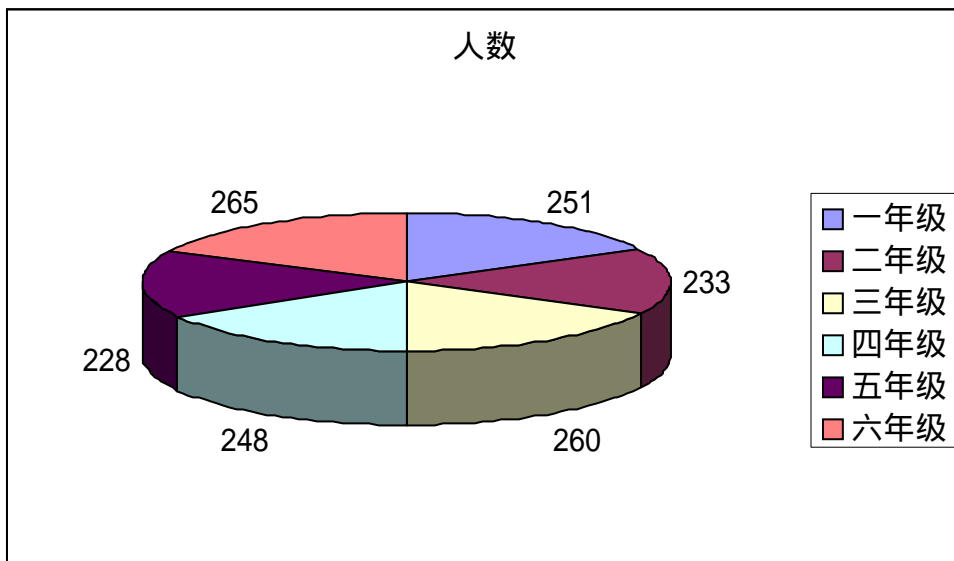
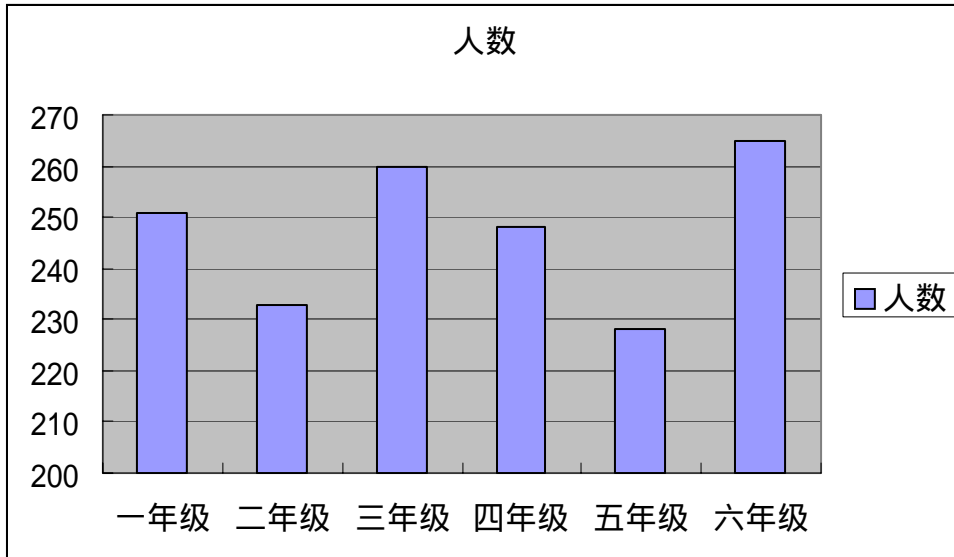
在 MVC 模式的时候，Martin Fowler 认为，主要关注两个分离：从模型中分离表现和从视图中分离控制器。

考察模型和表现的关系，可以得到的结论是，表现依赖于模型，而模型并不依赖于表现。模型的设计，应当能够做到显示独立，也就是说，同样的模型，应当能够支持不同的表现形式。当表现形式发生变化的时候，模型应该无需做改动。

考察一个 Excel 的例子。同样的一组数据，可以用不同的表现形式来表现。我们看下面的这组数据，这是某个学校的各个年级的人数统计表：

年级	一年级	二年级	三年级	四年级	五年级	六年级
人数	251	233	260	248	228	265

我们可以用简单的表格形式来表现这组数据，同时，我们也可以使用其他表现形式来表现这组数据，例如，柱状图、饼图，或者其他图形。



在这个例子中，同样的一组数据（模型），可以有多个表现的形式（视图），当用户通过其中的某个视图改变了模型，其他的模型应当也可以随着发生变化。例如，修改了表格中的数字，下面的柱状图和饼状图应该可以同步发生变化。在《设计模式》一书中，提供了一个称为“观察者”（Observer）的设计模式，可以达到这个目的。

同样的，从更大的范围来说，对于相同的业务逻辑，可以使用不同的界面形式，例如，可以分别使用 EXE 的 Windows 应用程序客户端，也可以使用 Web 客户端，甚至手机客户端，而后面连接相同的应用服务器。

我们在设计表现和模型的时候，考虑的关注点是不同的。在设计模型的时候，设计人员

更加关注的是业务领域的问题，例如数据的结构、逻辑的处理等等，这需要设计人员对领域有更深入的理解，对实现逻辑的技术有更深入的了解。而在设计表现的时候，设计人员更多的考虑的是界面的布局、界面的友好性以及美观程度等。双方考虑的角度是不同的，对人的技能的要求也是有差别的。

在测试方面，我们可以对模型进行小范围的单元测试，然后，组合进行整个逻辑的测试，这些工作可以在集成开发环境中利用很多测试工具来轻松完成，而界面的测试，则需要更加复杂的工具。

因此，从上面的这些角度来说，从模型中分离表现，是非常重要的。

相比之下，第二个分离，即视图和控制器的分离，就不那么重要了。很多情况下，可以把视图和控制器放在一起，尤其是在一些富客户端的程序中。当然，在基于 Web 的应用系统中，将视图和控制器分离的做法，是非常常见的。

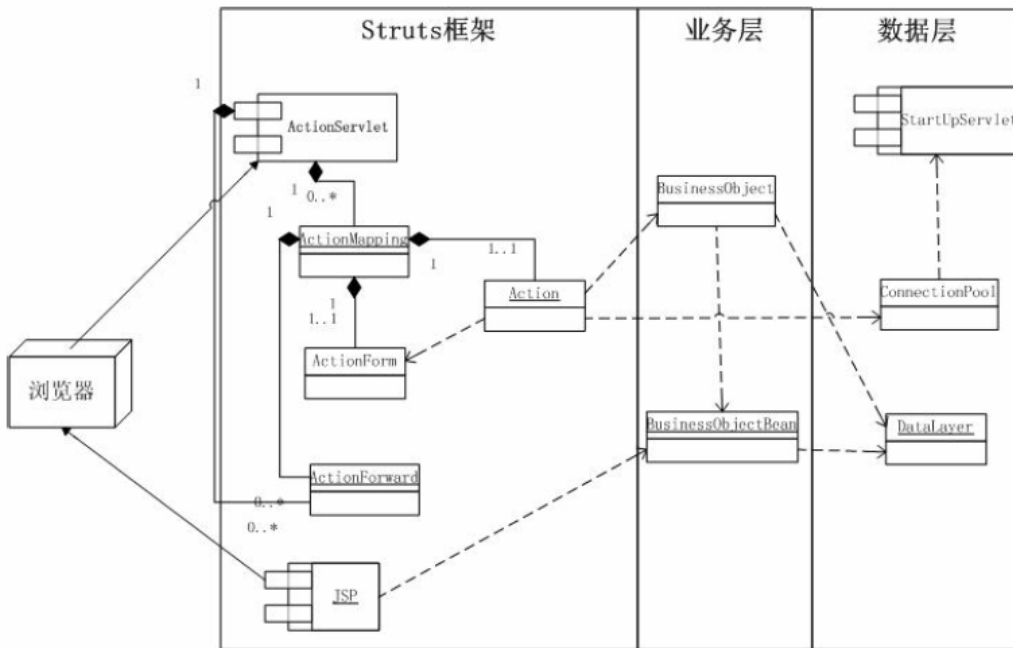
我们可以通过一个实例来理解 MVC 模式在开发中的应用。下面我们通过 Struts 框架来简要的探讨 MVC 模式在 Web 开发中的应用。

Struts 框架是一个典型的基于 JSP 的 Web 层应用框架。在 Struts 框架中，Model 通过 Action 来实现。在 Action 中，开发人员添加同逻辑处理相关的代码。如果是一个分布式的系统，实际的业务逻辑代码存在于应用服务层中，那么，Action 可以作为一个 Web 表现层和应用服务层之间的桥梁，一方面将页面的输入转发给应用服务层，另一方面，根据应用服务层处理的结果，请求控制器调用不同的界面进行显示。在数据封装方面，Struts 提供了 ActionForm，以实现对 View 和 Model 之间交互的支持。

View 就是 JSP 页面，页面上的数据封装在 FormBean 中。在这个方面，Struts 框架提供了一整套的标签库来支持更好的编程方式。通过使用这么标签库，可以方便的实现 ActionForm 和 JSP 表单之间的映射，完成对数据的封装。

Control 部分用于控制页面之间的转换逻辑。这个部分，Struts 通过一个 ActionServlet 来实现。ActionServlet 的核心就是 Struts-config.xml，Struts-config.xml 集中了所有页面的导航定义。对于大型的 WEB 项目，通过此配置文件即可迅速把握其脉络，这不管是对于前期的开发，还是后期的维护或升级都是大有裨益的。

下面的图显示了 Struts 框架的整体结构



使用 MVC 的模式来开发系统，能够很好的划分表现层的功能，这对于带给系统一个更好的结构，使系统更加易于维护有着很好的意义。

除了 MVC 模式，在界面设计方面，还有一种类似的模式，就是微软的 Document - View 模式。Document - View 模式在 MFC 中使用，Document 负责保存业务数据，处理业务逻辑，View 负责用户界面的显示、用户输入的收集和画面的跳转控制。通过前面的介绍我们可以发现，Document - View 模式实际上，几乎就是一个未将视图和控制器分离的 MVC 模型。这也再次说明一个问题，在某些情况下，尤其是富客户端的程序中，视图和控制器的分离是不那么重要的。

11.2 页面控制器

第12章 动态代码生成和编译技术

在后面的讨论中会涉及到一些关于动态代码生成的技术，因此，有必要先在这里对这个技术做一个介绍。

关于动态代码生成和编译的技术，在 Java 和 .Net 中都有所支持。在 Java 平台之上，有 JDK 自己的 tools.jar 提供的功能，也可以通过一些第三方的字节码增强器来实现。但就这方面的技术来说，.Net 提供的解决方案是最完整和成体系的。在这里，主要介绍 .Net 平台下的动态代码生成和编译技术。

在 .Net 平台下，有两种技术来实现动态代码生成和编译，分别是 Emit 和 CodeDom，下面，我们就这两种技术来做一些简单的介绍。

12.1 Emit

12.2 CodeDom

CodeDOM 的中文译名就是“代码文档对象模型”，使用这套模型，可以使编写源代码的程序的开发人员可以在运行时，根据表示所呈现代码的单一模型，用多种编程语言生成源代码，并且可以动态编译和运行所生成的代码。

为表示源代码，CodeDOM 元素相互链接以形成一个数据结构（称为 CodeDOM 图），它以某种源代码的结构为模型。System.CodeDom 名称空间定义可以表示源代码的逻辑结构（与具体的编程语言无关）的类型。System.CodeDom.Compiler 名称空间定义从 CodeDOM 图生成源代码的类型，和在受支持的语言中管理源代码编译的类型。编译器供应商或开发人员可以扩展受支持语言的集合。

.NET Framework 中包含了 C#、JScript 和 Visual Basic 的代码生成器和代码编译器。开发人员也可以通过扩展 System.CodeDom.Compiler 名称空间来实现自己的代码生成器和代码编译器。

使用 CodeDom 来动态生成代码和编译的过程一般是：

1. 使用 CodeCompileUnit 定义一个可编译的单元
2. 使用 CodeNamespace 定义一个名称空间，并把这个名称空间加入上面定义的可编译单元
3. 使用 CodeTypeDeclaration 定义一个类，并把这个类加入上面定义的名称空间
4. 使用 CodeTypeMember 的具体子类，例如 CodeMemberField 或者 CodeMemberMethod 为上面的类定义成员变量或者方法

-
5. 使用 CodeExpression 定义某个方法中具体代码的调用

6. 如果需要, 可以通过某个具体的 CodeDomProvider 来生成源代码, 例如, 可以使用 CSharpCodeProvider 为上面的结构生成具体的 C#代码。
7. 通过 CodeCompiler 将上面的代码树编译成可执行的文件。

下面的例子展示了使用 CodeDom 生成一个 HelloWorld 程序的例子。生成以后的代码应该是这个样子的：

```
using System;

namespace Sample
{
    public class DemoClass
    {
        static void Main(string[] args)
        {
            System.Console.WriteLine("Hello World!");
        }
    }
}
```

生成程序的代码如下：

```
public void GenerateCode()
{
    //生成一个可编译的单元, 这是最根部的东西
    CodeCompileUnit compunit = new CodeCompileUnit();
    //定义一个名为Sample的命名空间
    CodeNamespace sample = new CodeNamespace("Sample");
    compunit.Namespaces.Add(sample);
    sample.Imports.Add(new CodeNamespaceImport("System")); //导入System命名空间
    //定义一个名为DemoClass的类
    CodeTypeDeclaration MyClass = new CodeTypeDeclaration("DemoClass");
    sample.Types.Add(MyClass);
    //定义程序入口点, 就是Main()
    CodeEntryPointMethod Start = new CodeEntryPointMethod();
    MyClass.Members.Add(Start);
    //下面产生调用方法的语句, //这句会产生如下的C#代码System.Console.WriteLine("Hello
World!");
    CodeMethodInvokeExpression cs = new CodeMethodInvokeExpression
        (new CodeTypeReferenceExpression("System.Console"),
```

```
        "WriteLine", new CodePrimitiveExpression("Hello World!"));
Start.Statements.Add(cs);

//根据CodeDOM产生程序代码，代码文件就是DemoClass.cs，这里生成C#代码
CSharpCodeProvider cprovider = new CSharpCodeProvider();
ICodeGenerator gen = cprovider.CreateGenerator();
StreamWriter sw = new StreamWriter("DemoClass.cs", false);
gen.GenerateCodeFromCompileUnit(compunit, sw, new CodeGeneratorOptions());
sw.Close();

//编译源代码
ICodeCompiler compiler = cprovider.CreateCompiler();
//编译参数
CompilerParameters cp = new CompilerParameters(new string[] { "System.dll" },
        filepath.Substring(0, filepath.LastIndexOf(".") + 1) + "exe", false);
cp.GenerateExecutable = true;//生成EXE,不是DLL
CompilerResults cr = compiler.CompileAssemblyFromDom(cp, compunit);
}
```

上面只是一个很简单的例子，通过 CodeDom，可以生成任何复杂的程序。

第13章 远程过程访问的客户端整合

当今大部分的企业应用都是分布式的，单机版的软件虽然仍旧有很多，但是，在考虑一个完整的应用软件系统框架的时候，总是需要考虑完整的情况。多层分布式应用软件开发原则和技术通常也是适用于单机版软件的。

对于多层的应用系统来说，我们通常把它们划分成客户端、应用服务层和数据库。在应用服务层，我们需要考虑至少两个方面的问题：

- ✓ 如何实现业务逻辑
- ✓ 如何向客户端提供服务。

我们可能使用多种技术来实现服务的提供：Webservice、.Net Remoting、甚至 EJB 等。如此多的实现技术，带来的很大的灵活性，但同时也带来了问题，其中一个就是，有多少种服务端技术，就得有多少种相应的客户端访问技术。甚至，在某些分布式应用系统中，应用逻辑使用不同的技术开发，存在于不同的机器上，有的存在于客户机本机，有的使用 .Net Remoting 开发，存在于局域网内，有的使用因特网上的 Web Service，有的时候，我们希望相同的业务逻辑能够支持不同的客户端。

在这种情况下，我们需要一个一致的服务访问编程模型，以统合不同的服务访问模式，简化系统的开发和部署。Websharp Service Locator（以下简称 WSL）提供了这样一种能力，开发人员只需要定义服务访问接口，就可以使用一致的方式透明的访问这些服务，而不用理会这些服务之间的不同点。框架会自动生成访问远程服务需要的代理。

下面简单介绍一下 .Net 环境下的两种主要分布式访问技术：

Web Service

Web Service 是基于网络的、分布式的模块化组件，它执行特定的任务，遵守具体的技术规范，这些规范使得 Web Service 能与其他兼容的组件进行互操作^[21]。它可以使用标准的互联网协议，像超文本传输协议 HTTP 和 XML，将功能体现在互联网和企业内部网上。Web Service 平台是一套标准，它定义了应用程序如何在 Web 上实现互操作性。可以使用任何语言，在任何平台上写 Web Service。

Web Service 平台需要一套协议来实现分布式应用程序的创建。任何平台都有它的数据表示方法和类型系统。要实现互操作性，Web Service 平台必须提供一套标准的类型系统，用于沟通不同平台、编程语言和组件模型中的不同类型系统。目前这些协议有：

1. XML 和 XSD

可扩展的标记语言 XML 是 Web Service 平台中表示数据的基本格式。除了易于建立和易于分析外，XML 主要的优点在于它既与平台无关，又与厂商无关。XML 是由万维网协会 (W3C) 创建，W3C 制定的 XML Schema XSD 定义了一套标准的数据类型，并给出了一种语言来扩展这套数据类型。

Web Service 平台是用 XSD 来作为数据类型系统的。当你用某种语言如 VB.NET 或 C# 来构造一个 Web Service 时，为了符合 Web Service 标准，所有你使用的数据类型都必须被转换为 XSD 类型。如想让它使用在不同平台和不同软件的不同组织间传递，还需要用某种东西

将它包装起来。这种东西就是一种协议，如 SOAP。

2. SOAP

SOAP 即简单对象访问协议 (Simple Object Access Protocol)，它是用于交换 XML 编码信息的轻量级协议。它有三个主要方面：XML-envelope 为描述信息内容和如何处理内容定义了框架，将程序对象编码成为 XML 对象的规则，执行远程过程调用 (RPC) 的约定。SOAP 可以运行在任何其他传输协议上。例如，你可以使用 SMTP，即因特网电子邮件协议来传递 SOAP 消息，这可是很有诱惑力的。在传输层之间的头是不同的，但 XML 有效负载保持相同。

Web Service 希望实现不同的系统之间能够用“软件-软件对话”的方式相互调用，打破了软件应用、网站和各种设备之间的格格不入的状态，实现“基于 Web 无缝集成”的目标。

3. WSDL

Web Service 描述语言 WSDL 就是用机器能阅读的方式提供的一个正式描述文档而基于 XML 的语言，用于描述 Web Service 及其函数、参数和返回值。因为是基于 XML 的，所以 WSDL 既是机器可阅读的，又是人可阅读的。

4. UDDI

UDDI 的目的是为电子商务建立标准，UDDI 是一套基于 Web 的、分布式的、为 Web Service 提供的、信息注册中心的实现标准规范，同时也包含一组使企业能将自身提供的 Web Service 注册，以使别的企业能够发现的访问协议的实现标准。

5. 远程过程调用 RPC 与消息传递

Web Service 本身其实是在实现应用程序间的通信。我们现在有两种应用程序通信的方法：RPC 远程过程调用 和消息传递。使用 RPC 的时候，客户端的概念是调用服务器上的远程过程，通常方式为实例化一个远程对象并调用其方法和属性。RPC 系统试图达到一种位置上的透明性：服务器暴露出远程对象的接口，而客户端就好像在本地使用的这些对象的接口一样，这样就隐藏了底层的信息，客户端也就根本不需要知道对象是在哪台机器上。

微软的 .NET 技术应该算是时下最好的 Web Service 开发技术。.NET 平台不仅延续了微软一贯的编程风格，而且还增加了许多支持 Web 服务的关键性技术，使得 .NET 在操作的简单性和执行的稳定性，高效性上达到了一个非常好的结合。微软的 Visual Studio .NET 便是一个便于 Web 服务的开发工具。微软的目标是，将其新编程语言——C#作为 Web Service 的首选语言。

.Net Remoting

.Net Remoting 是 .Net 环境下的另外一种分布式处理方式。从某种意义上来说，Remoting 就是 DCOM 的一种升级，它改善了很多功能，并极好的融合到 .Net 平台下。Microsoft® .NET Remoting 提供了一种允许对象通过应用程序域与另一对象进行交互的框架^[22]。

在 Remoting 中是通过通道 (channel) 来实现两个应用程序域之间对象的通信的。如图所示：

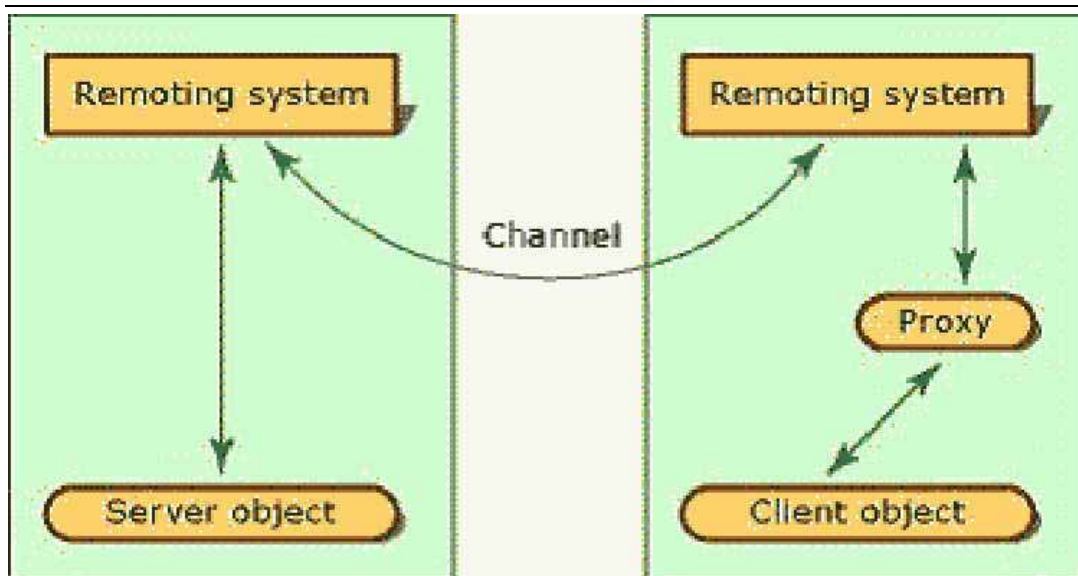


图 6.1

客户端通过 Remoting，访问通道以获得服务端对象，再通过代理解析为客户端对象。这就提供一种可能性，即以服务的方式来发布服务器对象。远程对象代码可以运行在服务器上（如服务器激活的对象和客户端激活的对象），然后客户端再通过 Remoting 连接服务器，获得该服务对象并通过序列化在客户端运行。

在 Remoting 中，对于要传递的对象，设计者除了需要了解通道的类型和端口号之外，无需再了解数据包的格式。但必须注意的是，客户端在获取服务器端对象时，并不是获得实际的服务端对象，而是获得它的引用。这既保证了客户端和服务端有关对象的松散耦合，同时也优化了通信的性能。

Remoting 的两种通道

Remoting 的通道主要有两种：Tcp 和 Http。在 .Net 中，System.Runtime.Remoting.Channel 中定义了 IChannel 接口。IChannel 接口包括了 TcpChannel 通道类型和 Http 通道类型。它们分别对应 Remoting 通道的这两种类型。

TcpChannel 类型放在名字空间 System.Runtime.Remoting.Channel.Tcp 中。Tcp 通道提供了基于 Socket 的传输工具，使用 Tcp 协议来跨越 Remoting 边界传输序列化的消息流。TcpChannel 类型默认使用二进制格式序列化消息对象，因此它具有更高的传输性能。HttpChannel 类型放在名字空间 System.Runtime.Remoting.Channel.Http 中。它提供了一种使用 Http 协议，使其能在 Internet 上穿越防火墙传输序列化消息流。默认情况下，HttpChannel 类型使用 Soap 格式序列化消息对象，因此它具有更好的互操作性。

远程对象的激活方式

在访问远程类型的一个对象实例之前，必须通过一个名为 Activation 的进程创建它并进行初始化。这种客户端通过通道来创建远程对象，称为对象的激活。在 Remoting 中，远程对象的激活分为两大类：服务器端激活和客户端激活。

- (1) 服务器端激活，又叫做 WellKnown 方式，在这种方式下，服务器应用程序在激活对

象实例之前会在一个众所周知的统一资源标识符 (URI) 上来发布这个类型。然后该服务器进程会为此类型配置一个 WellKnown 对象，并根据指定的端口或地址来发布对象。 .Net Remoting 把服务器端激活又分为 SingleTon 模式和 SingleCall 模式两种。

SingleTon 模式：此为有状态模式。如果设置为 SingleTon 激活方式，则 Remoting 将为所有客户端建立同一个对象实例。当对象处于活动状态时，SingleTon 实例会处理所有后来的客户端访问请求，而不管它们是同一个客户端，还是其他客户端。SingleTon 实例将在方法调用中一直维持其状态。

SingleCall 模式：SingleCall 是一种无状态模式。一旦设置为 SingleCall 模式，则当客户端调用远程对象的方法时，Remoting 会为每一个客户端建立一个远程对象实例，至于对象实例的销毁则是由 GC 自动管理的。

(2) 客户端激活。与 WellKnown 模式不同，Remoting 在激活每个对象实例的时候，会给每个客户端激活的类型指派一个 URI。客户端激活模式一旦获得客户端的请求，将为每一个客户端都建立一个实例引用。

Websharp Service Locator 的主要接口

WSL 是一个轻量级的框架，非常易于使用和扩展。如果想使用 WSL，那么只有一个类需要打交道：ServiceLocator，它的定义如下：

```
public abstract class ServiceLocator
{
    public static object FindService(string serviceName, Type clientInterface)
}
```

如果你想用自己的定位器扩展这个框架，那么，只有一个接口需要扩展：IServiceLocator。这个接口非常简单，只有一个方法：

```
public interface IServiceLocator
{
    object FindService(string serviceName, Type clientInterface);
}
```

Websharp Service Locator 的配置文件

需要在三个地方配置 WSL。

首先，在 configSections 节中，注册 WSL 配置文件处理类的相关信息，配置方法如下：

```
<configSections>

  <section name="Websharp.Enterprise"

    type="Websharp.Enterprise.EnterpriseConfigHandler, Websharp" />

</configSections>
```

然后，在 Websharp.Enterprise 节中，注册不同的服务定位器。如果你自己扩展了这个框架，添加了新的服务定位器，也在这里注册。其中，locator 属性的格式是：“类全名, Assembly 名”。服务定位器都是 Singleton 的。下面是目前 WSL 支持的服务定位器的注册的信息：

```
<Websharp.Enterprise>

  <ServiceTypes>

    <ServiceType name="LocalAssembly"

      locator="Websharp.Enterprise.LocalAssemblyLocator, Websharp" />

    <ServiceType name="WebService"

      locator="Websharp.Enterprise.WebServiceLocator, Websharp" />

    <ServiceType name="DotNetRemoting"

      locator="Websharp.Enterprise.DotNetRemotingLocator, Websharp" />

  </ServiceTypes>

</Websharp.Enterprise>
```

最后，在 Websharp.Enterprise 下的 Services 节中，注册每个服务。每个 Service 需要的属性取决于不同的 Locator 的实现，但是，name、service-type 和 deploy-model 是必须的。对于 deploy-model，可以有两种属性值：Singleton 和 MultiInstance。

下面是一个例子：

```
<Websharp.Enterprise>

  <ServiceTypes>

    <ServiceType name="LocalAssembly"

      locator="Websharp.Enterprise.LocalAssemblyLocator, Websharp" />

    <ServiceType name="WebService"

      locator="Websharp.Enterprise.WebServiceLocator, Websharp" />

  </ServiceTypes>

</Websharp.Enterprise>
```

```
<ServiceType name="DotNetRemoting"
    locator="Websharp. Enterprise. DotNetRemotingLocator, Websharp" />
</ServiceTypes>
<Services>
    <Service name="HelloWorld" service-type="LocalAssembly" deploy-model="Singleton"
        type="EnterpriseClient. HelloWorld, EnterpriseClient" />
    <Service name="HelloWorldWebService" service-type="WebService"
        deploy-model="Singleton"
        url="http://localhost/webservicetest/hello. asmx"
        namespace="http://www. websharp. org/webservices/" />
</Services>
</Websharp. Enterprise>
```

注：对于配置文件，在 Web 项目中，可以是 web.config 文件，对于 Windows 项目，可以自己为项目添加一个 app.config 配置文件。关于 .net 项目配置文件的更多内容，请参考 MSDN 的相关文档。

如何使用 Websharp Service Locator

使用 WSL，一般的方法是这样的：

1. 定义一个同你需要访问的服务一致的接口(当然,如果你的服务是实现某个接口的,可以直接使用该接口)。接口的方法名和参数必须同服务类的方法名和参数一致。如果你的方法名和服务的方法名不一致,那么,可以使用 ServiceMethodNameAttribute 来指明服务的方法名。
2. 在配置文件按中注册你需要访问的服务。
3. 调用 ServiceLocator 的 FindService 方法。
4. 调用接口的方法。

下面是一些例子,这些例子使用 visual studio.net 2003 开发,同样可以从 sourceforge 下载。

LocalAssemblyLocator 的 Hello World 例子

按照以下步骤进行：

1. 创建一个名为“EnterpriseClient”的 windows console 项目,加入 Websharp.dll

的引用。

2. 添加一个类，名为“HelloWorld”，然后添加一个名为“GetHello”的方法，代码如下：

```
public class HelloWorld
{
    public string GetHello(string hello)
    {
        return hello;
    }
}
```

3. 添加一个名为“IHelloworld”的接口，代码如下：

```
public interface IHelloworld
{
    string GetHello(string hello);
    [ServiceMethodName("GetHello")]
    string GetHello2(string hello);
}
```

4. 填写配置文件

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <configSections>
        <section name="Websharp.Enterprise"
            type="Websharp.Enterprise.EnterpriseConfigHandler, Websharp" />
    </configSections>

    <Websharp.Enterprise>
        <ServiceTypes>
```

```
<ServiceType name="LocalAssembly"
    locator="Websharp.Enterprise.LocalAssemblyLocator, Websharp" />
<ServiceType name="WebService"
    locator="Websharp.Enterprise.WebServiceLocator, Websharp" />
</ServiceTypes>
<Services>
    <Service name="HelloWorld" service-type="LocalAssembly"
        deploy-model="SSingleton"
        type="EnterpriseClient.HelloWorld, EnterpriseClient" />
</Services>
</Websharp.Enterprise>
</configuration>
```

5. 在 Main 方法中添加如下代码：

```
public static void Main(string[] args)
{
    IHelloWorld hello= ServiceLocator.FindService("HelloWorld", typeof(IHelloWorld)) as IHelloWorld;
    Console.WriteLine(hello.GetHello("Hello World"));
    Console.WriteLine(hello.GetHello2("Hello again"));
    Console.ReadLine();
}
```

6. 运行程序，就能够得到下面的结果（图 6.2）：



```
C:\E:\Websharp\Websharp2004\EnterpriseClient\bin\De
Hello World
Hello again
_
```

图 6.2

Hello World 的 WebServiceLocator 例子

按照以下步骤进行：

1. 新建一个 webservice 项目，名为“WebServiceTest”。
2. 新建一个 webservice 类，名为“Hello”，并添加一个“HelloWorld”方法，代码如下：

```
[WebService(Namespace="http://www.websharp.org/webservices/")]
public class Hello : System.Web.Services.WebService
{
    [WebMethod]
    public string HelloWorld()
    {
        return "Hello World";
    }
}
```

3. 使用上面我们创建的“EnterpriseClient”项目，添加一个接口“IHello”，代码如下：

```
public interface IHello
{
    string HelloWorld();
}
```

4. 填写配置文件

```
<Service name="HelloWorldWebService" service-type="WebService" deploy-model="SSingleton"
        url="http://localhost/webservicetest/hello.asmx"
        namespace="http://www.websharp.org/webservices/" />
```

5. 在 Main 方法中添加下面的代码：

```
public static void Main(string[] args)
```

```
{  
  
    IHello hello1= ServiceLocator.FindService  
  
        ("HelloWorldWebService",typeof(IHello)) as IHello;  
  
    Console.WriteLine(hello1.HelloWorld());  
  
    Console.ReadLine();  
  
}
```

6. 运行程序，能够得到下面的结果（图 6.3）：

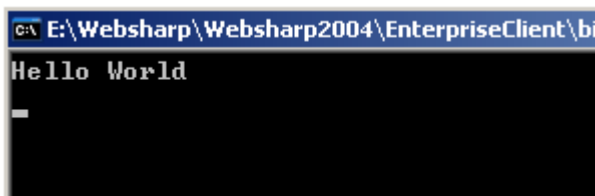


图 6.3

Websharp Service Locator 的实现

Websharp Service Locator 的实现原理是：

Websharp 根据给定的 Service 名和接口，查询配置文件，获取服务的类型以及相关代理类生成类，代理类生成类会根据服务类型和接口，在内存中生成调用服务需要的相关代理类，然后，由代理类调用远程服务。当然，为了减少即时编译代理类的开销，这里也做了缓存处理，代理类都只需要生成一次，然后，就缓存起来，以后就可以很快的调用。在这里，使用了动态代码生成技术，这个技术的使用同前面已经讨论的类似，只是需要动态生成的类，依据不同的服务类型而不同。因此，具体的技术，可以参见前面的讨论，相关代码，请参见源代码，这里就不列出来了。

目前的进展

目前，我们已经完成了以下服务定位器的设计和开发：

- Local Assembly Locator
- Web Service Locator
- .Net Remoting Locator

使用 Websharp Service Locator，我们已经可以很好的支持在 .Net 平台上的分布式系统的开发。

将来的目标

我们将来的目标如下：

-
1. 继续完善目前已经完成的定位器，包括
 - a) WebService 的异步调用
 - b) 性能的提高和 Bug 的清除
 2. 完成其他定位器的设计和实现，包括
 - a) Corba
 - b) 与 Java 的互操作，尤其是 J2EE

小结

使用 WSL，我们可以使用一致的编程模型访问不同类型的服务，从而简化软件的开发和部署。例如，我们可以在开始的时候，使用本地 Assembly 的方式开发软件，然后，能够很容易的改成使用 Webservice 来发布服务，将软件变成多层应用。我们也可以使用 WSL 来让相同的服务能够支持不同的客户端，而所有的客户端都使用相同的编程模型。

Websharp 是一个还处于开发阶段的框架，但是，因为他是开放源代码的，我们可以直接使用他来进行进一步的开发。目前 WSL 支持的服务还不是很多，实现也还比较简单，但是，他提供了一个很好的框架和构建分布式应用的方案，将来，他将提供越来越多的功能。

第14章 智能客户端

软件从主机系统向 C/S 结构的转变,除了带来了系统资源的合理分配,也带来了客户端的新体验,尤其是图形界面系统的大量使用(主要是 Windows 操作系统),提供了高质量、响应迅速的用户体验,并且具有良好的开发人员和平台支持。但是,C/S 结构下的胖客户端,非常难于部署和维护。随着应用程序和客户端平台的复杂性不断增加,以可靠且安全的方式将应用程序部署到客户计算机的难度也将不断增加。如果部署了不兼容的共享组件或软件库,则一个应用程序可以很容易地破坏另一个应用程序,这种现象称为应用程序脆弱性。新版本的应用程序通常通过重新部署整个应用程序来提供,这可能使应用程序脆弱性问题变得更加严重。

为了解决这个问题,同时也伴随着 Internet 的高速发展,出现了基于 Web 的 B/S 结构,客户端也就成了“瘦客户端”。这种结构,它解决了许多与应用程序部署和维护相关联的问题。瘦客户端应用程序是在中央 Web 服务器上部署和更新的,因此,它们消除了将应用程序的任何部分显式部署到客户计算机并加以管理的必要性。

然而,瘦客户端应用程序也具有一些缺点。首先一个缺点是瘦客户端应用程序的表现能力同胖客户端不能相比。虽然使用 HTML 能够做出非常漂亮的页面,使用 JavaScript 也能够客户端进行一些处理,但是,JavaScript 这样的脚本语言毕竟不能和编译语言的功能相比。从理论上来说,使用 C++ 等编译型语言,你可以做出任何形式的客户端界面。使用瘦客户端,常用的应用程序功能(如拖放、撤消-重复以及上下文相关帮助等)可能不可用,这些都可能降低应用程序的可用性。

其次,使用基于 Web 的瘦客户端,就意味着必须总是具有网络连接,一旦网络故障,整个应用系统就不可用。并且因为应用程序的大部分逻辑和状态位于服务器上,所以瘦客户端会频繁地向服务器发回数据和处理请求。浏览器必须等待响应到达,然后用户才能继续使用该应用程序;因此,该应用程序的响应速度通常要比胖客户端应用程序慢得多。该问题在低带宽或高延迟的情况下被恶化了,并且产生的性能问题可能导致应用程序可用性和用户效率大幅度下降。要求输入大量数据以及/或者在多个窗口中频繁导航的应用程序尤其会受到这一问题的影响。

最后,由于基于 Web 的瘦客户端都使用 HTTP 协议,而 HTTP 协议是一个无状态的协议,因此,为了保留客户端的状态,必须使用一些特殊的技术,例如 Session 等。这样,会给服务器带来额外的负担,同时,程序员也不得不处理因 Session 过期而带来的种种问题。

随着 Java 和 .Net 等新一代编程平台的出现和逐步成熟,并且由于这些编程语言和平台从一开始就考虑了网络应用,出现了智能客户端技术,分别以 Java Web Start 和 Microsoft Smart Client 为代表。智能客户端应用程序可以将胖客户端应用程序的优点与瘦客户端应用程序的部署和可管理性优点结合起来。

智能客户端具有以下一些特点:

丰富的表现能力

智能客户端使用诸如 Java 和 C# 这样的语言来编写,能够充分利用编程语言的功能(当

然前提是通过安全检查),最大限度地利用了代码和数据部署在客户端上并且在本地图行和访问这一事实。它为应用程序提供了内容丰富且响应迅速的用户界面,以及强大的客户端处理能力。例如,它可能使用户能够执行复杂的数据操作、可视化、搜索或排序操作。

智能客户端可以利用客户端硬件资源(如电话或条码读取器)以及其他软件和应用程序,可以最大限度地使用本地资源以及将本地资源集成到您的智能客户端应用程序,使应用程序更好、更有效地使用已经提供给您的硬件。这使它们非常适合于解决瘦客户端应用程序无法很好解决的问题,并且可以使用多文档界面,使用户可以更为有效地工作,并减少数据输入错误。此类解决方案还可以使您的应用程序更加紧密地与用户的工作环境集成(例如,通过采用自定义的或熟悉的用户界面),从而降低培训成本。使用客户计算机上的资源还可以减少服务器端资源要求。

可以通过智能客户端应用程序集成或协调其他客户端应用程序,以便提供一致且高效的总体解决方案。这些应用程序还应该了解正在使用应用程序的上下文,并且应该适应该上下文以尽可能地帮助用户;例如,通过根据用户的使用模式或角色抢先缓存适当且有用的数据。

零接触部署和动态加载

同瘦客户端一样,智能客户端可以只部署在服务器上面,当需要的时候,才从服务器上下载相应的模块,并动态加载进应用系统。当更新应用系统的时候,也只需要更新服务器,客户端会自动从服务器上下载最新的程序版本,这免去了胖客户端更新系统的麻烦。智能客户端应用程序可以在其运行时或位于后台时对自身进行自动更新。这一功能使其可以逐个角色地进行更新;以分阶段的方式更新,从而可以将应用程序推介给先导小组或受限的用户组;或者按照制定的时间表更新。

智能客户端可以部署在 Web 服务器上面,通过 80 端口来加载和更新程序,并同样使用 80 端口通过访问 Web Service 的方式来同服务器交互,这样,就可以解决防火墙的问题,保证系统的安全性。

离线客户端

智能客户端应用程序使用网络加载程序集,一旦加载之后,程序集便被缓存到了本地。当用户至少启动了一次应用程序后,程序集就被下载和缓存到本地内存中了,所以用户就可以离线运行的智能客户端程序了,并且只有在必要的时候才同服务器通信,调用服务器端的资源。

尽管网络已经成了大多数应用软件必不可少的环境,但我们仍不能给企业应用程序提供始终连续的连接。在某些情况下,我们还需要在断开网络的条件下工作,例如在某些情况使使用手持终端工作的时候。离线式工作方式可以把数据先缓存再本地,然后,在你重新在线时,自动进行数据和应用程序的更新,这种特征是人们很想得到的,但在使使用智能终端之前,这是很难实现的。

同胖客户端一样,智能客户端给客户端分布大量的处理,这就为服务器免除了它在一个基于 Web 的应用程序中需要承担的负荷。智能客户端采取一种用户希望应用程序采取的工作方式——允许快速数据存取和管理,而不需要不必要的屏幕更新。

提供客户端设备灵活性

智能客户端还可以提供灵活且可自定义的客户端环境,从而使用户可以将应用程序配置为支持他或她喜欢的工作方式。智能客户端应用程序没有被限制到桌面计算机或膝上型计算机。随着小规模设备的连接性和能力的增加,愈发需要能够提供对多个设备上重要数据和服务的访问的有用客户端应用程序。

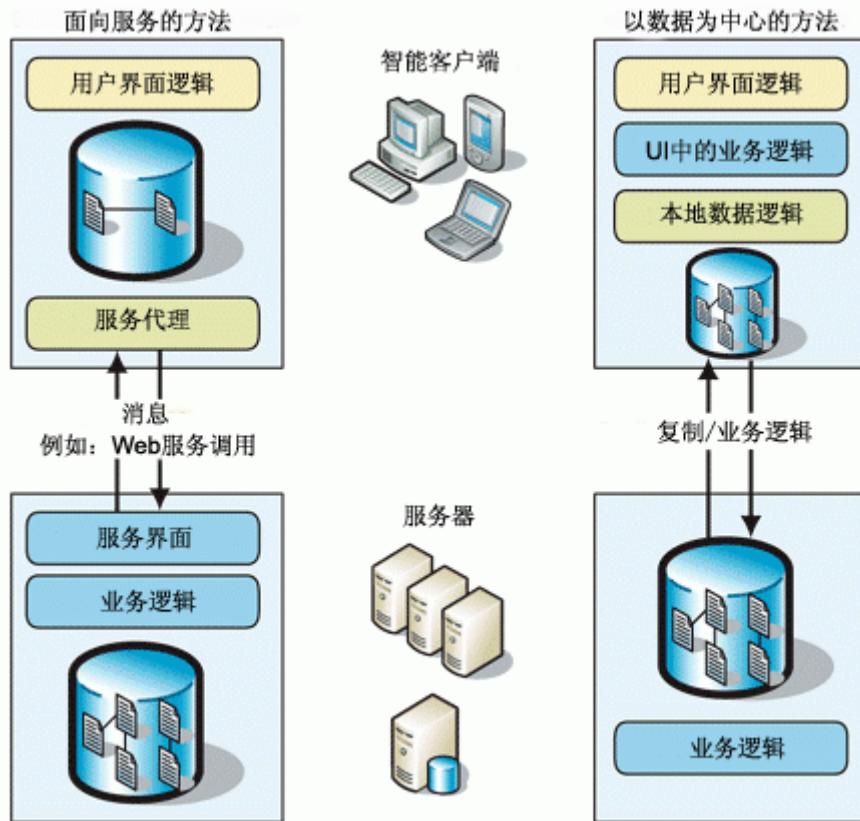
可以对智能客户端进行设计以使其适应宿主环境,并且为它们运行时所在的设备提供适当的功能。例如,适合在 Pocket PC 上运行的智能客户端应用程序应该提供相应的用户界面,该用户界面在较小的屏幕区域上被调整为使用笔针。

在许多情况下,您需要设计多个版本的智能客户端应用程序,每个版本都面向特定的设备类型,以便充分利用该设备所支持的特定功能。因为小规模设备通常在提供完整范围的智能客户端应用程序功能方面受到限制,所以它们可能只提供对功能完善的智能客户端应用程序所提供的数据和服务子集的移动访问,或者它们可用于在用户移动时收集和整合数据。最后,可以由功能更加完善的智能客户端应用程序或服务器端应用程序来分析或处理这些数据。

能够感知目标设备的功能和使用环境(无论它是桌面、膝上型、平板还是移动设备),以及能够定制应用程序以提供最适当的功能,这些都是许多智能客户端应用程序的基本特点。

两种类型的智能客户端方案

在考虑智能客户端和服务器的交互时,可以有两种类型的方法:以数据为中心的方法和面向服务的方法。使用以数据为中心的方法,客户端可以使用本地数据库和复制机制,以便在脱机模式下管理对数据的更改。使用面向服务的方法,客户端可以通过服务请求与许多服务进行交互。如果应用程序处于脱机模式,它可以推迟服务请求,直到重新连接至 Web 服务。每种方法都有其优点和缺点,且适用于不同类型的应用程序。下面的图显示了这两种方法的结构:



Microsoft Smart Client

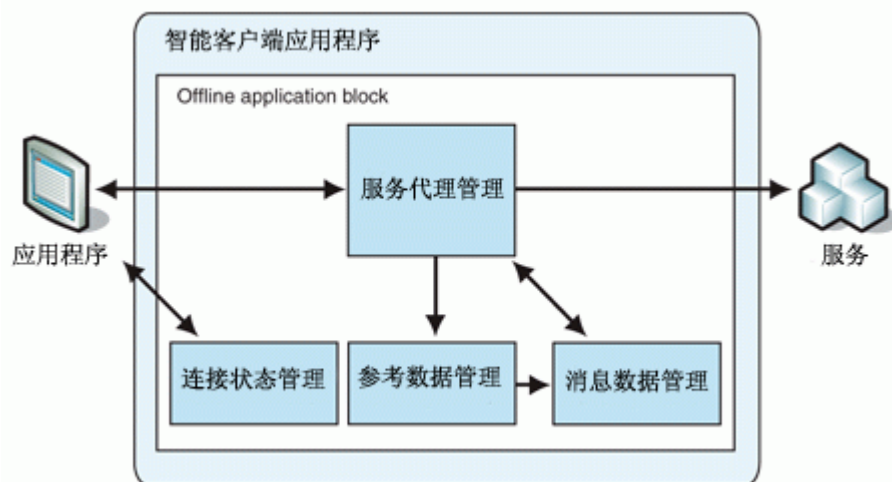
微软通过 Offline Application Block 提供了智能客户端技术。Offline Application Block 是一套在 Microsoft .Net 平台上进行智能客户端开发的类库，可以从微软的网站上下载。

Offline Application Block 提供如下的将脱机模式功能构建到应用程序中的能力：

- 检测网络连接是否存在，这样可以使应用程序能够根据其联机或脱机状态来执行
- 缓存必要的的数据，这样即使在网络连接不可用时，应用程序也可以继续运行
- 当网络连接可用时，将客户端应用程序的状态和/或数据与服务器同步化

它还可以利用其他块组件。例如，它可以使用 Caching Application Block 来利用缓存，并将 Caching Application Block 作为其参考数据缓存的基础。

因为 Offline Application Block 使用面向服务的方法，所以它所包含的功能（或类）与上图中所示的功能相对应。下面的图展示了 Offline Application Block 的相应子系统，它们是松散耦合的组件。



下面的表说明了图中显示的每个子系统。

子系统	说明
连接状态管理	“连接状态管理”检测应用程序是处于联机状态还是脱机状态。有两种方法可以判断连接状态：手动判断或通过自动过程判断。如果选择自动判断，则连接状态表示为包括网络层和应用程序连接的多层连接性（请注意，应用程序连接检测并不是在该应用程序块中实现的）。应用程序的行为会根据连接状态而变化。
服务代理管理	“服务代理管理”与 Offline Application Block 的这些元素（“消息数据管理”、“参考数据管理”）以及服务器进行交互。它会进行协调，以便将任务完成通知返回到应用程序。
参考数据管理	“参考数据管理”与“服务代理管理”和“消息数据管理”配合工作，以下载存储在本地计算机上的参考数据。在大多数情况下，参考数据是用于完成工作流的只读数据。“参考数据管理”可使参考数据与服务器上的数据保持一致。它将消息存储在“队列”中以下载参考数据。然后，“执行程序”将使用消息服务请求与服务连接，以下载参考数据。
消息数据管理	消息数据是在工作流过程中创建的数据。当应用程序处于脱机状态时，该数据将存储在一个本地队列中。当应用程序联机后，“执行程序”会从“队列”中删除消息，发出与服务器同步数据的“服务请求”，然后数据就会与服务器进行同步。

在安全性方面，Offline Application Block 并不提供安全架构。但是，它支持缓存和

队列中存储的所有数据的加密和签名。加密和签名的使用不是强制性的,但强烈建议您使用。通过使用这些安全措施,应用程序可以在将数据临时存储在硬盘上时对数据进行保护。

小结

不管采用什么开发模式,什么框架来构建我们的界面系统,我们都必须明确,在界面层,主要的工作就是三个:一是界面的表现,二是页面迁移模式,三是同应用服务层的交互。合理的将这三个功能进行划分,以及有机的进行组合,是构建一个优秀的界面系统的基础。

第四部分 系统建模过程

第15章 简述

面向对象的软件工程，同传统的面向过程的软件工程相比，在需求的获取、系统分析、设计和实现方面都有着很大的区别。UML 是 OOA 和 OOD 的常用工具。使用 UML 来构建软件的面向对象的软件工程的过程，就是一个对系统进行不断精化的建模的过程。这些模型包括用例模型、分析模型、设计模型，然后，我们需要使用具体的计算机语言来建立系统的实现模型。当然，在整个软件工程中，我们还需要建立系统的测试模型，以保证软件产品的质量。

使用面向对象的工具来构建系统，就应该使用面向对象的软件工程方法。然而，我们经常会发现，在实际的开发过程中，很多开发人员虽然能够理解 UML 的所有图形，却仍然不能得心应手地使用 UML 来构建整个项目，其很大的原因，是仍然在使用原有的软件工程方法，而不清楚如何使用 UML 来建立系统的这些模型，不清楚分析和设计的区别，以及他们之间的转化。

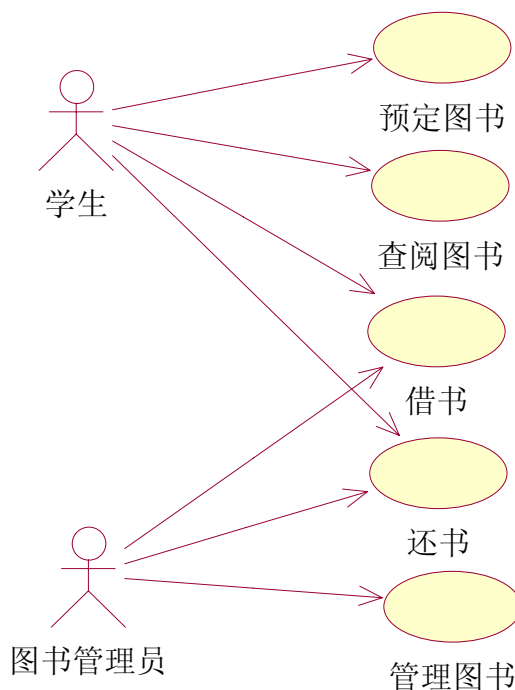
应用软件系统，就其本质来说，是使用计算机对现实世界进行的数字化模拟。应用软件的制造过程，按照 UML 的方法，就是建立这一些列模型的过程。本文将就一个图书馆系统，说明如何使用 UML 来对系统进行这一系列的建模。

关于这个图书馆系统，基本的需求比较简单，就是允许学生可以在图书馆借阅和归还图书，另外，也可以通过网络或者图书馆的终端来查阅和预订书。当然，图书馆管理员也可以对图书进行管理。为了简化系统，我们没有把图书馆中的人员作细分。

之所以采用这个相对简单案例，是因为很多人都对图书馆系统有很强的感性认识，这样，读者不需要花很多的时间来理解系统包含的业务知识。同时，也因为本文只是对使用 UML 的过程做一个探讨，着眼于使用 UML 进行建模的过程，说明各个层次的模型之间的区别和联系，展示系统演进的过程，而不会深入 UML 的细节方面。对于更加复杂的系统，其分析和设计的方法是相通的，可以举一反三。

第16章 用例模型——系统需求的获取

用例模型定义系统做什么，是用来获取系统需求的有效手段。用例模型由“角色”和“用例”组成。我们在构建一个用例的时候，通常要做的第一件事情是识别角色，或者说，参与者。然后我们需要识别系统为参与者提供的服务，或者说，参与者的行为，也就是用例。最后，我们确定角色和用例之间的关系。在这个图书馆系统中，我们可以识别出的角色有学生和图书管理员。整个用例模型包含的用例有：借书、还书、查阅图书、预订图书，以及图书维护。用例模型可以用用例图表示如下：



确定有效用例的关键是，检查用例是否包含了一个完整的功能。用例不能定的过细，不能把一个完整的功能的一个部分作为一个用例，也不能在一个用例中包含过多的功能。例如，用户的登录。学生在预定图书的时候，可能会需要首先登录系统，这是系统的一个功能。但是，这个功能只是预定图书这个完整的功能中的一个步骤，或者说一个子功能，就不适于做成一个用例。另一方面，借书和还书，都是相对完整的功能，如果把这两个用例合并成一个类似于“处理图书”的用例，显然是不能明确的表达用例需要表达的含义的。

描述用例

需要了解的是，使用 UML 进行系统建模，并非只是意味着画 UML 图形，对 UML 图的文档说明是同样重要的。学习 UML，不仅仅要学习 UML 图形，相应的文档描述方法也同样要学习，也同样重要。

在描述用例时，我们可以用文字来描述，也可以用其他图形来描述，例如，顺序图或者活动图等等。下面给出了一个 RUP 中推荐的描述用例的完整的结构：

- ◆ **名称**。名称无疑应该表明用户的意图或用例的用途，如“研究班招生”。
- ◆ **标识符 [可选]**。唯一标识符，如“UC1701”，在项目的其他元素（如类模型）中可用它来引用这个用例。
- ◆ **说明**。概述用例的几句话。
- ◆ **参与者 [可选]**。与此用例相关的参与者列表。尽管这则信息包含在用例本身中，但在没有用例图时，它有助于增加对该用例的理解。
- ◆ **状态 [可选]**。指示用例的状态，通常为以下几种之一：进行中、等待审查、通过审查或未通过审查。
- ◆ **频率**。参与者 *访问* 此用例的频率。这是一个自由式问题，如用户每次录访问一次或每月一次。
- ◆ **前置条件**。一个条件列表，如果其中包含条件，则这些条件必须在访问用例之前得到满足。
- ◆ **后置条件**。一个条件列表，如果其中包含条件，则这些条件将在用例成功完成以后得到满足。
- ◆ **被扩展的用例 [可选]**。此用例所扩展的用例（如果存在）。扩展关联是一种广义关系，其中扩展用例接续基用例的行为。这是通过扩展用例向基用例的操作序列中插入附加的操作序列来实现的。这总是使用带有 <<extend>> 的用例关联来建模的。
- ◆ **被包含的用例 [可选]**。此用例所包含用例的列表。包含关联是一种广义关系，它表明对处于另一个用例之中的用例所描述的行为的包含关系。这总是使用带有 <<include>> 的用例关联来建模的。也称为 *使用或具有 (has-a)* 关系。
- ◆ **假设 [可选]**。对编写此用例时所创建的域的任何重要假设。您应该在一定的时候检验这些假设，或者将它们变为决策的一部分，或者将它们添加到操作的基本流程或可选流程中。
- ◆ **基本操作流程**。参与者在用例中所遵循的主逻辑路径。因为它描述了当各项工作都正常进行用例的工作方式，所以通常称其为 *适当路径 (happy path)* 或 *主路径 (main path)*。
- ◆ **可选操作流程**。用例中很少使用的逻辑路径，那些在变更工作方式、出现异常或发生错误的情况下所遵循的路径。
- ◆ **修改历史记录 [可选]**。关于用例的修改时间、修改原因和修改人的详细信息。
- ◆ **问题 [可选]**。如果存在，则为与此用例的开发相关的问题或操作项目的列表。
- ◆ **决策**。关键决策的列表，这些决策通常由您的 SME 作出，并属于用例的内容。将这些决策记录下来对于维护 *团体记忆库 (group memory)* 是相当重要的。

下面，我们对借书这个用例来做描述：

名称：借书”。

说明：学生在图书馆挑选好需要的图书后，通过图书管理员把书借回去。

参与者：学生，图书管理员

频率：每天可能会有很多次。最繁忙的情况是，借书的人非常多，按照现在的速度，大约每分钟完成一个人的结束工作。

前置条件：无

后置条件：修改所借出的图书的剩余数量。

假设：借书者总是从图书馆找到书，然后才能拿书办理借书手续，因此，总是有足够的书可以出借。

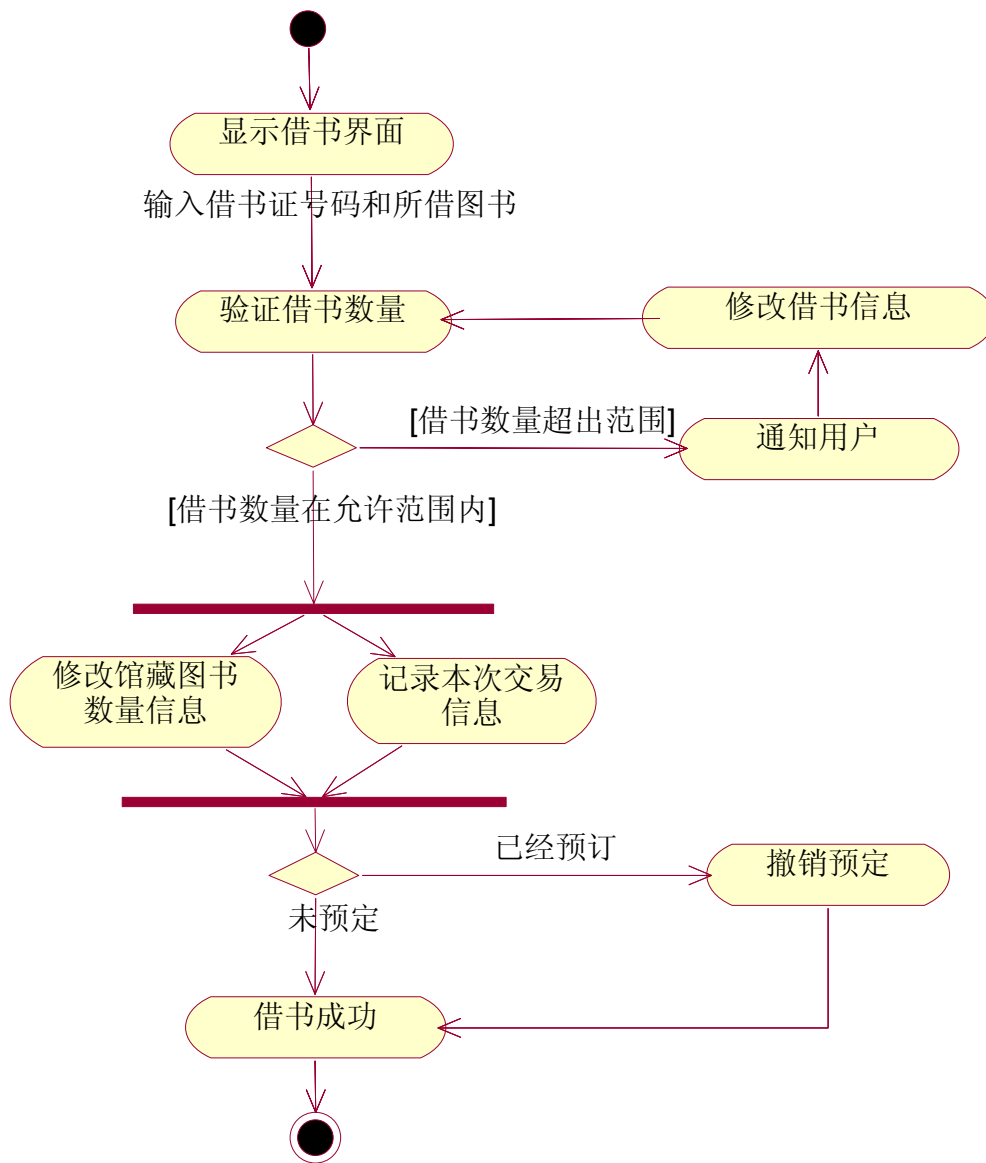
基本操作流程：借书成功。

- 1) 学生将所借图书和借书证交给图书管理员
- 2) 图书管理员将学生借书证号码和所借图书输入系统
- 3) 系统校对借书信息，比对该学生以往借书情况和当前借书情况，如果不存在不允许借书的情况，则记录借书交易的信息，并且修改相应的馆藏图书的数量信息。
- 4) 如果该学生已经预订了这本图书，则撤销该预定。
- 5) 报告交易成功。

可选操作流程：所借图书超出最大借书数量。

- 1) 学生将所借图书和借书证交给图书管理员
- 2) 图书管理员将学生借书证号码和所借图书输入系统
- 3) 系统校对借书信息，比对该学生以往借书情况和当前借书情况，发现已超出最大借书数量，则停止当前交易，并且提示用户错误原因。
- 4) 图书管理员可以应学生的意见，减少借书数量，并重新提交系统。

流程活动图：见图一。



图一：借书活动图

问题：暂无。

决策：略。

上面，我们就这个用例做了一个比较详细的描述。按部就班，我们就可以逐步把整个系统的用例模型完成。

再次强调一点，使用 UML 建模，文档说明和图形同样重要，并且，要使用 UML 的方法来编写文档。

第17章 分析模型——开发者的视野

在系统分析的过程中，我们所关注的依然是问题，但是，同用例模型不同的是，用例模型是从最终用户的角度来看待问题，而分析模型是从开发者的角度来描述问题。用例模型的主要工作是描述现实世界的业务流程，而很少会涉及系统的概念。分析，则是从系统的角度来看待软件应该为用户提供的服务。同样，同设计不同的是，分析仍然停留在“做什么”的层次。而设计，则需要解决“怎么做的问题”。

开发语言等技术选择通常不会在分析模型中考虑。分析模型是独立于实现的，这样，可以提供最大的复用，并且，可以帮助开发人员方知过早的陷入技术的细节中去，从而能够从一个更加一般的角度去理清思路。

静态模型的建立

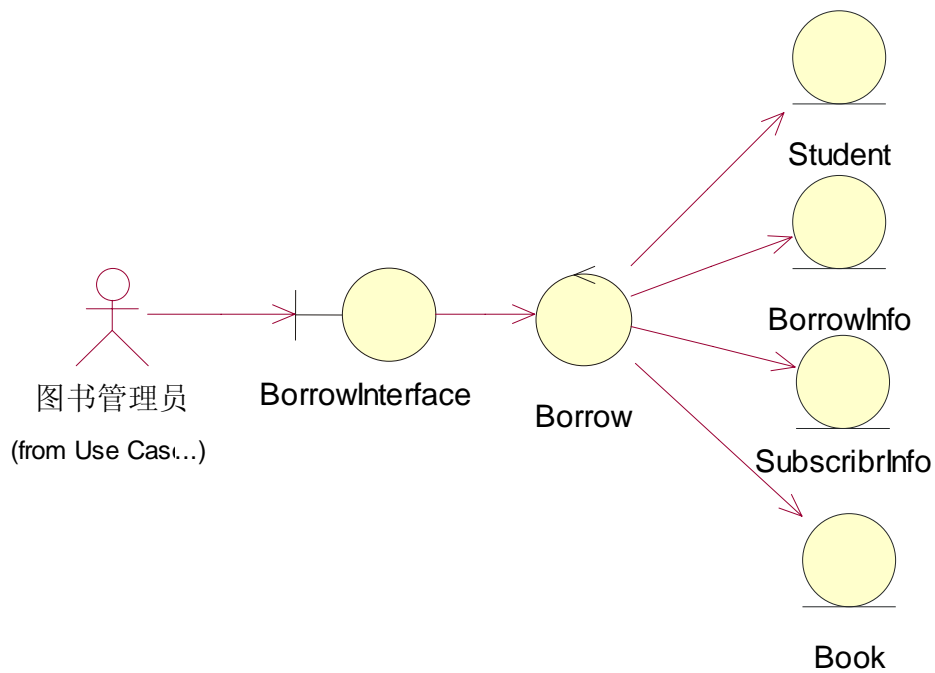
进行分析建模的第一步，通常是识别对象，然后提取出类。考虑著名的 MVC 模式，我们需要识别实体、控制和边界三种对象。按照 MVC 模式来为识别对象做指导，是非常好的做法。对象识别的结果，就是我们所需要的静态模型，通常表现为类图。

我们首先识别出实体对象，这些对象通常来说是比较明显的，例如系统中的角色，系统需要处理的资料，如本系统中需要处理的图书资料等；有些实体对象需要稍微分析一下才能得到，例如，在本系统中，为了记录图书借还的信息，我们可能需要一个对象来专门记录这一信息。这些对象就是所谓的 Modal（实体类）。

然后我们需要识别为了完成系统业务逻辑而需要的业务逻辑对象，以及同用户进行交互的界面类，在 MVC 模式中，他们分别对应于 Control（控制类）和 View（边界类）。在分析阶段，这些对象通常都按照比较自然的方式来组织，例如，为了完成一个业务功能，我们通常需要一个控制类和一个边界类，控制类执行业务逻辑，边界类同客户进行交互。当然，这不是绝对的，在进行进一步深入的分析后，这些类可能会被分解和合并。

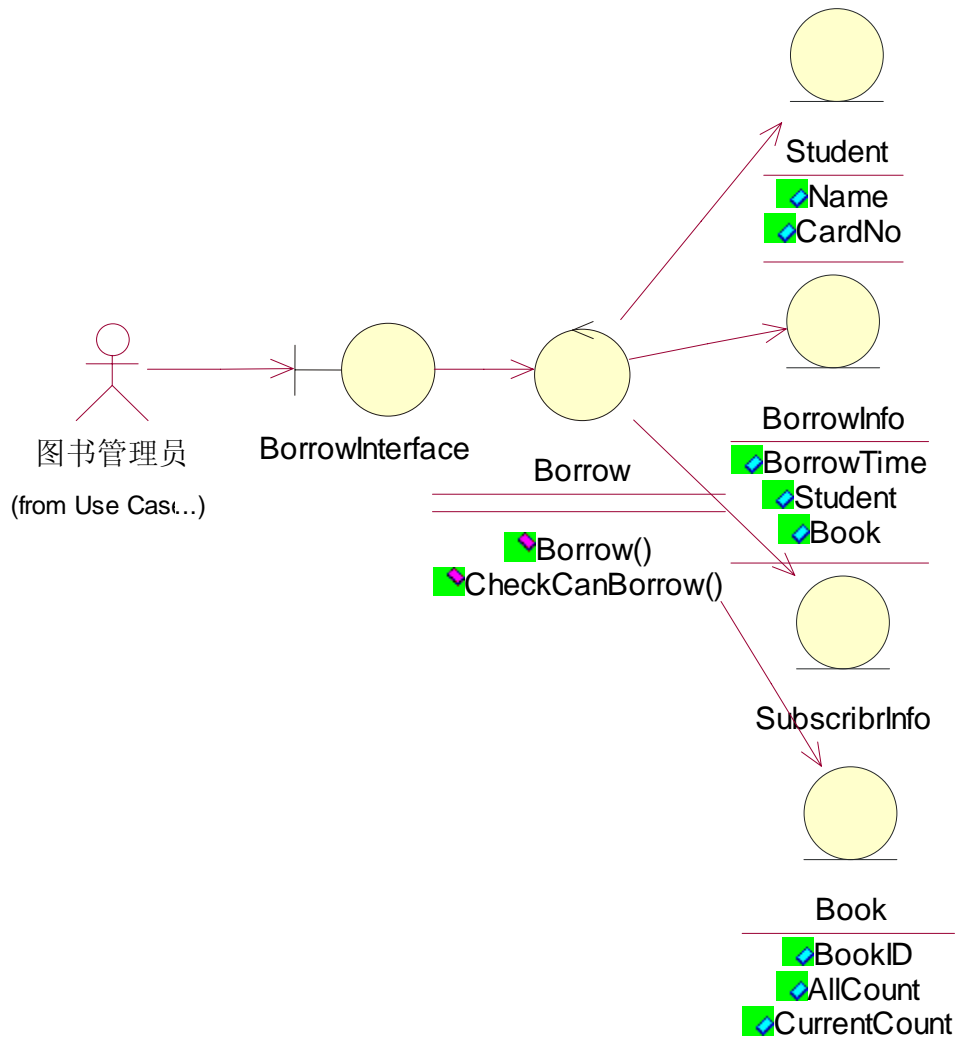
一口不能把所有的饭都吃掉，系统分析也是这样。我们需要一个一个用例的来进行分析。现在，我们首先来分析借书这个用例。

在这个用例中，我们首先可以识别出一些直接的对象，包括图书管理员 (BookAdmin)、学生 (Student)、图书 (Book)，然后，稍作分析，我们会发现我们需要一个实体对象来记录图书的借还信息 (BorrowInfo)。最后，我们发现，在借书的过程中，我们会使用到预定图书的信息 (SubscribrInfo)。到这一步，我们基本完成了实体对象的识别。然后，我们发现我们需要一个借书的控制类 (Borrow) 来执行借书的动作，以及一个用户界面 (BorrowInterface) 来接受用户的输入。这样，初步的模型我们就可以建立了。



图二：借书类图

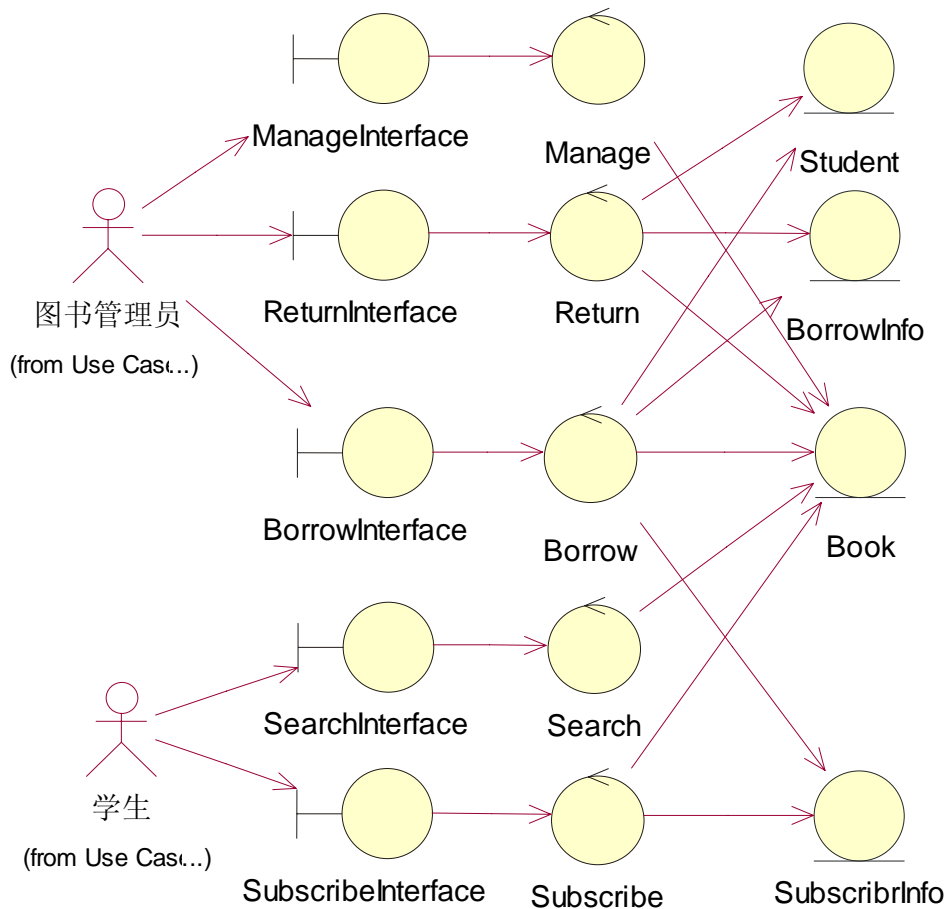
在分析模型中，我们也需要识别出类的一些属性和方法。同样的，为了避免过早的陷入细节中，以及适应将来在设计时类的变化，在分析模型中，我们一般只把一些主要的属性和方法标识出来。例如，对于 Student 类，我们只需要 Name 和 CardNo（借书证号）属性，对于 Book，我们只需要 BookID、AllCount（书的总数）、CurrentCount（当前数量）等属性。现在，我们为我们的类图添加上述属性，就可以得到下面的结果：



图三：借书类图

不过，把这些属性都标上后，图形就比较难布置。因为文章版面的问题，在以后的文字中，除非必要，一般会把属性都隐藏起来，这样，看起来会整洁一些。

依样画葫芦，我们把还书、查阅图书、预定图书、管理图书这些用例一并分析后，我们就能够得到整个系统的静态分析模型。

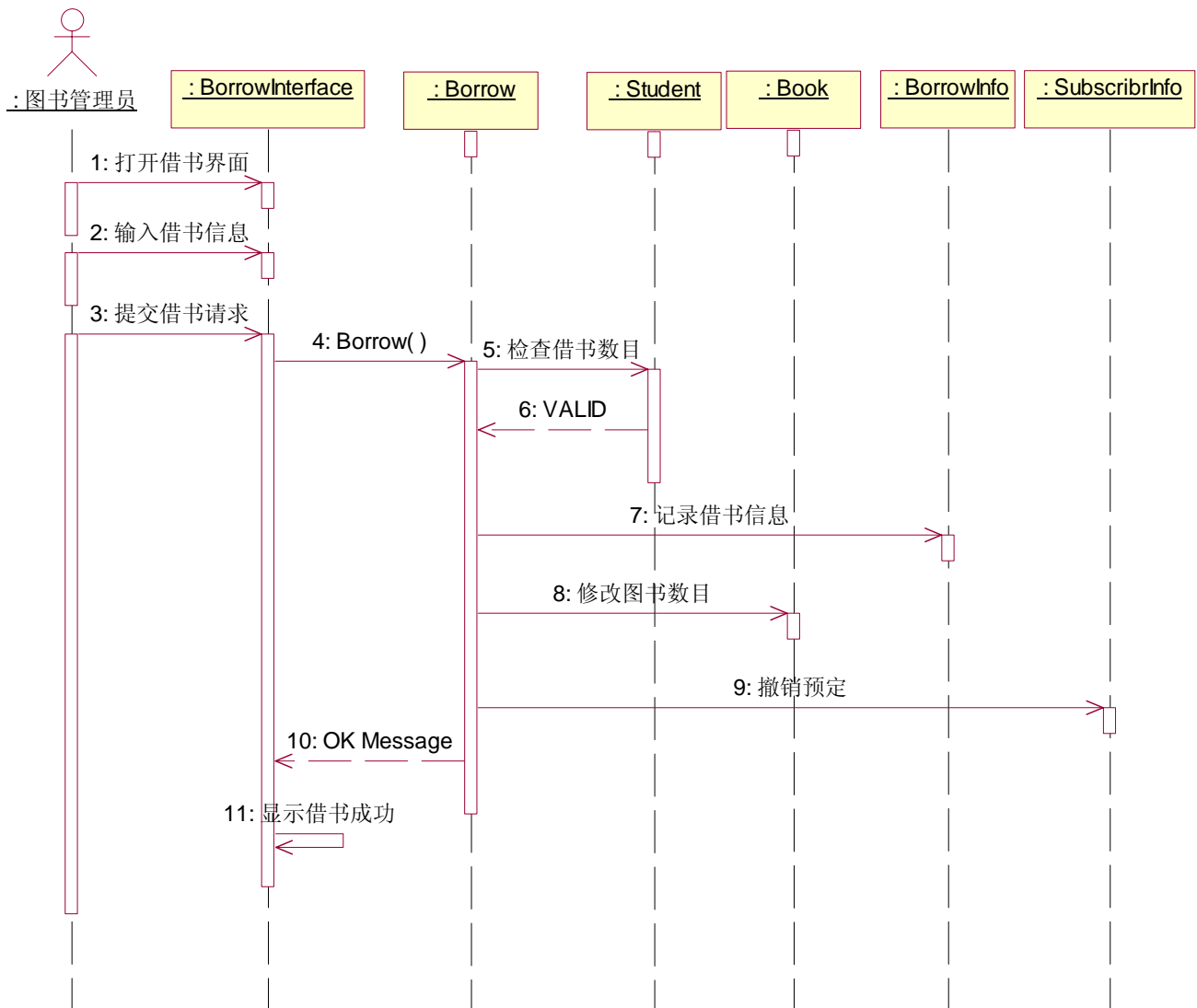


当然，我们随后需要对这个初步的模型作进一步的整理和分析，类图会发生变化，各个类之间会产生一些联系。我们也可能会使用一些分析模式，来进一步优化我们的分析结果。关于分析模型的知识，可以参见 Martin Fowler 所著的《Analysis Patterns——Reusable Object Models》一书。在这里，我们就不进一步做探讨了。

动态模型的建立

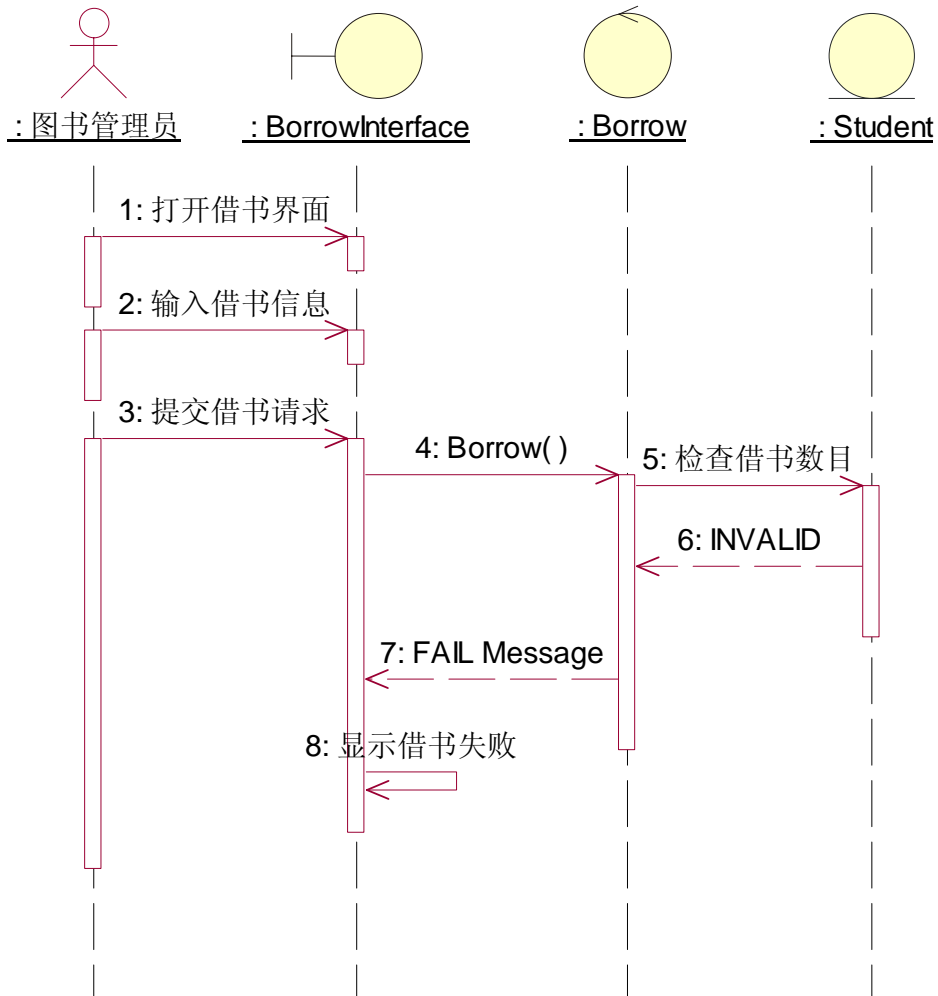
在面向对象的系统中，业务流程表现为对象之间的交互。我们有了上面分析的得到的对象后，就可以来描述他们是怎么进行交互和协作的了。在 UML 中，我们可以使用顺序图、活动图或者状态图来建模这些动态的过程。同样的，我们首先来看借书这个用例。

在借书这个用例中，有两个事件流：借书成功（正常事件流）和所借图书超出最大借书数量（非正常事件流）。我们首先来做“借书成功”这个事件流，下面是这个事件流的顺序图：



分析过程中，消息的传递同后面的设计模型和实现模型相比，可能会有区别，也不太严密，例如，Borrow 的过程中，记录 BorrowInfo 的动作，到最后，可能不会是 BorrowInfo 这个实体类自己来记录（往往会有一个实体访问类来完成这个功能，如 JDO 中的 PersistenceManager，这取决于你采用的具体的技术框架），但是，在这里，我们可以这样来传递消息，表示需要记录这个信息。

同样的，我们也可以为“所借图书超出最大借书数量”来作他的顺序图，这个图相对简单一些：



同样的,我们也可以为其他用例的其他事件流创建动态模型,在这里就不一一画出来了。动态模型和静态模型的建立是一个交互的过程。在建立动态模型的过程中,我们会发现一些新的类,也会为已有的类找到一些新的属性和方法,这样,我们会需要去修改我们的类图。反之亦然。

当分析模型完成后,我们就对系统需要完成的功能有了一个比较完整和清晰的认识,下面,就可以开始我们的设计工作了。

第18章 系统设计——实现方案

设计是对系统的详细描述。我们需要在这里提供详细的解决方案。设计同分析所使用的工具一样，也需要建立静态和动态的模型，也同样使用类图、顺序图、协作图、活动图、状态图等来表示。

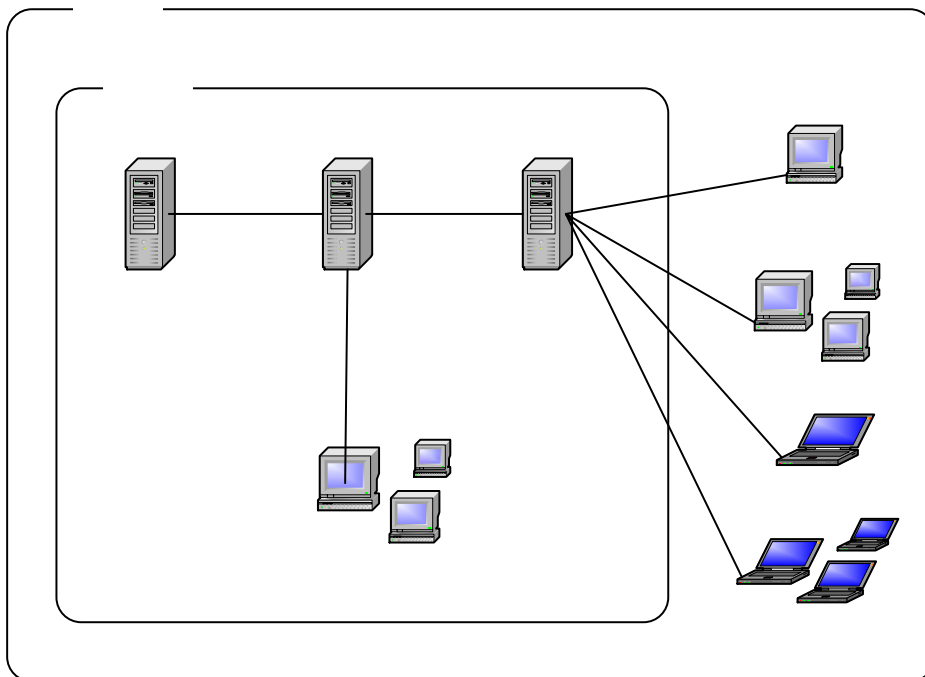
在正式进行设计之前，我们还有一些工作需要完成，那就是选择我们的技术方案，这会影响到我们设计。

技术选择——设计前的工作

设计是为实现服务的，实现时准备采用的技术会影响设计方案的采用。以下这些技术问题是需要考虑的：

- ◆ 准备使用什么样的客户端？
- ◆ 准备采用什么编程语言？
- ◆ 准备采用什么框架技术？
- ◆ 如果是分布式系统，那么，准备采用什么通信机制？

在这个图书馆系统中，我们发现，对于借书和还书来说，总是在图书馆内部发生，并且客户端的数量是有限的数个，其使用的频率比较高，效率和使用的方便性是需要注重考虑的，而客户端软件的维护工作量相对比较少，可以不用考虑太多，因此我们准备采用传统的 Windows Form 的客户端。但是，对于图书的查阅以及预定来说，我们希望在整个校园网内提供这个功能，使得学生无论在什么地方都能够使用这个功能，所以，我们会考虑采用 Web 浏览器的客户端，这样会方便系统的部署。也就是说，我们的系统需要同时支持两种不同的客户，显然，采用 N 层系统结构，把系统逻辑集中在应用服务器上是一个比较好的方案。最后，为了系统的安全，我们希望把 Web 服务器和应用服务器分开。这样，我们的系统的架构的拓扑图就基本上如下所示：



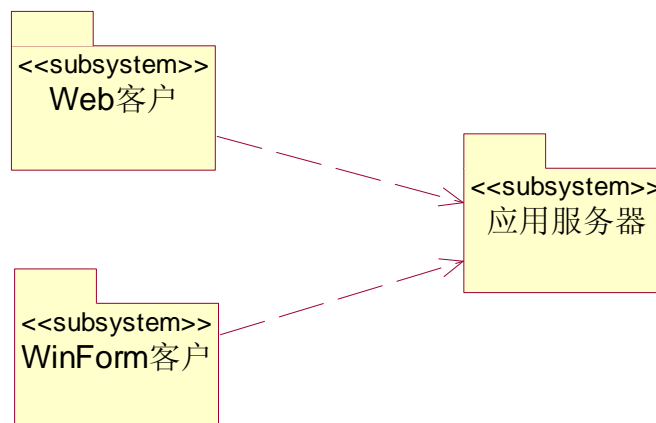
这是一个典型的分布式系统，在考虑了各种平台和技术之后，我们决定采用 EJB 技术来构建这个系统，他已经为我们提供了构建应用系统所需要的优秀的技术框架，同时，我希望在客户端和应用服务器的调用中，采用 Web Service 的方式（试验一下新技术：）。Windows Form 的客户端，我们使用 Java 来创建一个 Windows 应用程序，Web 客户，则采用 JSP 技术。

当然，你也可以采用微软的 .Net 平台以及 C# 语言来完成这个工作，但是由于微软在 .Net 平台上尚未提供象 J2EE，或者 JDO 一样成熟的应用系统框架，因此，你必须自己来设计这个框架。在这个方面，鄙人也设计了一个自己的 Websharp 框架，或许可以帮助你省略这方面的工作。关于 Websharp 的内容，可以参见拙作 [《面向对象的应用服务层设计》](#)。

这样，我们在具体技术方面的决策基本上就完成了，可以开始进行具体设计了。当然，在实际项目中，可能还有很多细节的工作需要去做，例如系统的约定，设计规范等，但这不是本文讨论的内容。

设计包或者子系统

首先我们需要来对系统进行一个划分。因为我们的系统是一个 N 层的分布式系统，包含了应用服务器和客户端，而客户端又包含了 Windows Form 客户端和 Web 客户端，所以，我们首先把系统分成 3 个包：应用服务器、Winform 客户和 Web 客户：



因为我们的系统包含了 5 个用例，每个用例都是一个完整的子系统，因此，我们可以在上述 3 个包的下面，又各自划分成 5 个子包。但是，在这里，因为系统比较简单，每个用例需要完成的功能都比较简单，因此，在这里我们就不错进一步的划分了。

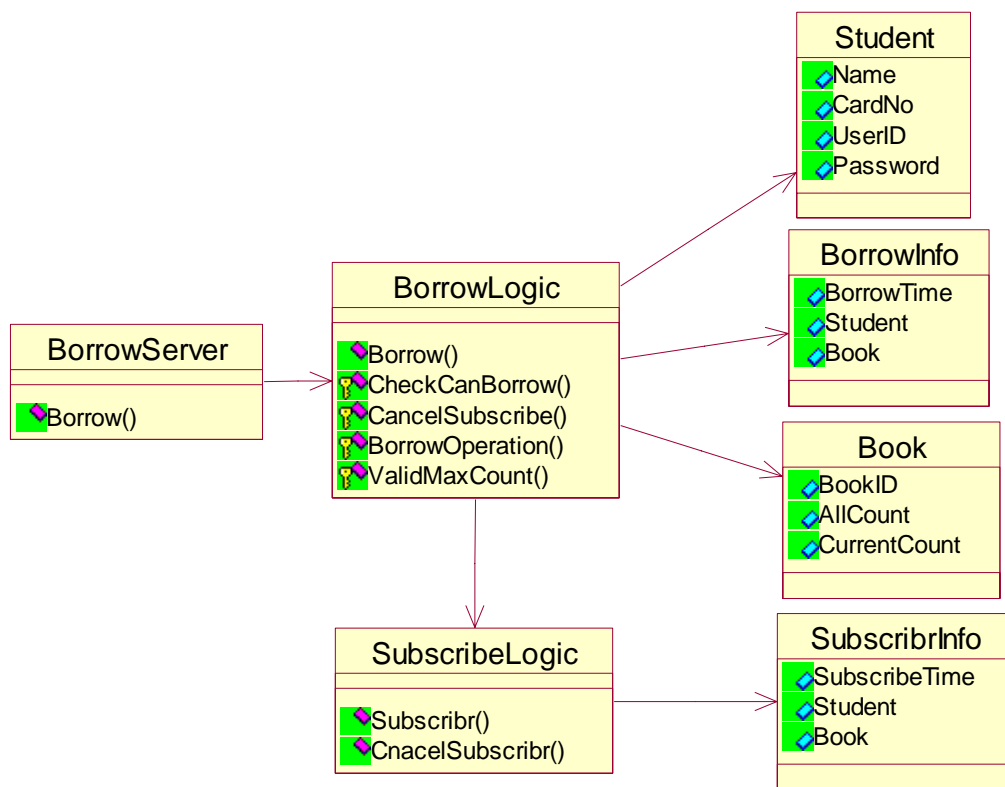
设计应用服务器

下面，我们首先来设计应用服务器部分。我们还是从借书这个用例开始设计。

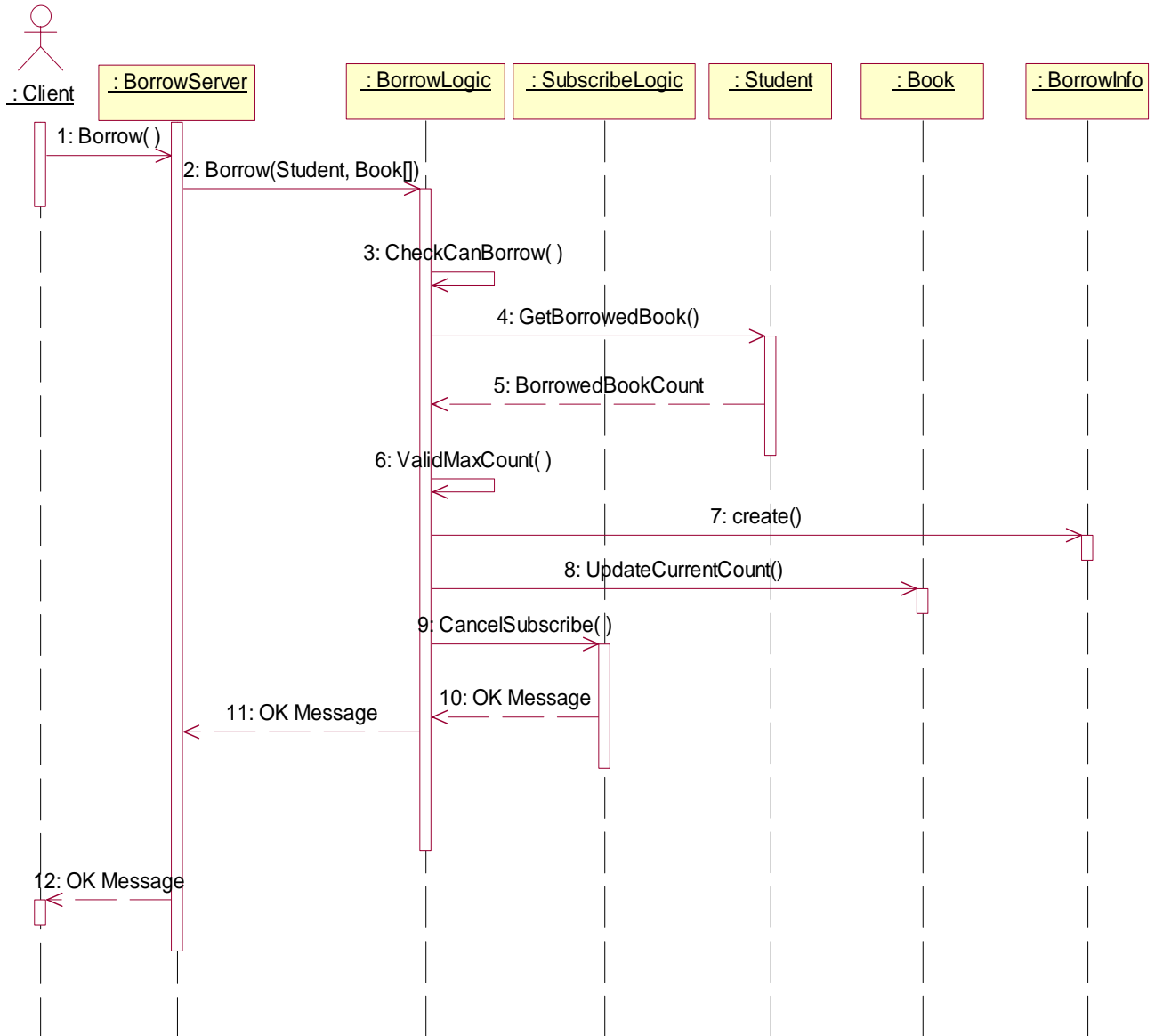
从分析模型中，我们知道，需要一个控制类来完成借书的业务逻辑，在这里，我们设计一个 BorrowLogic 类来完成这个功能，这个类被设计成 SessionBean；同时，因为我们需要向客户端提供服务，而我们又不希望直接把业务逻辑暴露给客户端，所以，设计 BorrowServer 这个 Web Service 来向客户端提供服务。这样，实际上，在分析模型中的 Borrow 类，在这里，被映射成了 BorrowLogic 和 BorrowServer 两个类。

同样的，我们需要 Book 类来记录图书的信息，Student 来标识学生，我们也需要 BorrowInfo 和 SubscribeInfo 类来分别记录借书和图书预定的情况。这样，建立设计模型中的静态模型所需要的类基本上已经齐全了。当然，到最后阶段，我们需要为这些类补充完整属性和方法。这些实体类，最后都被设计成 EntityBean。

这个部分的整个设计模型看上去就基本上会是这个样子：

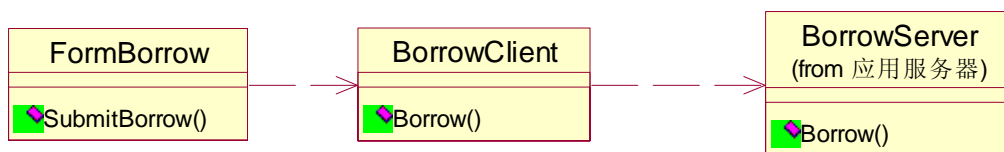


然后 ,我们来为借书成功这个事件流设计他的动态模型。我们还是使用顺序图来表示之。

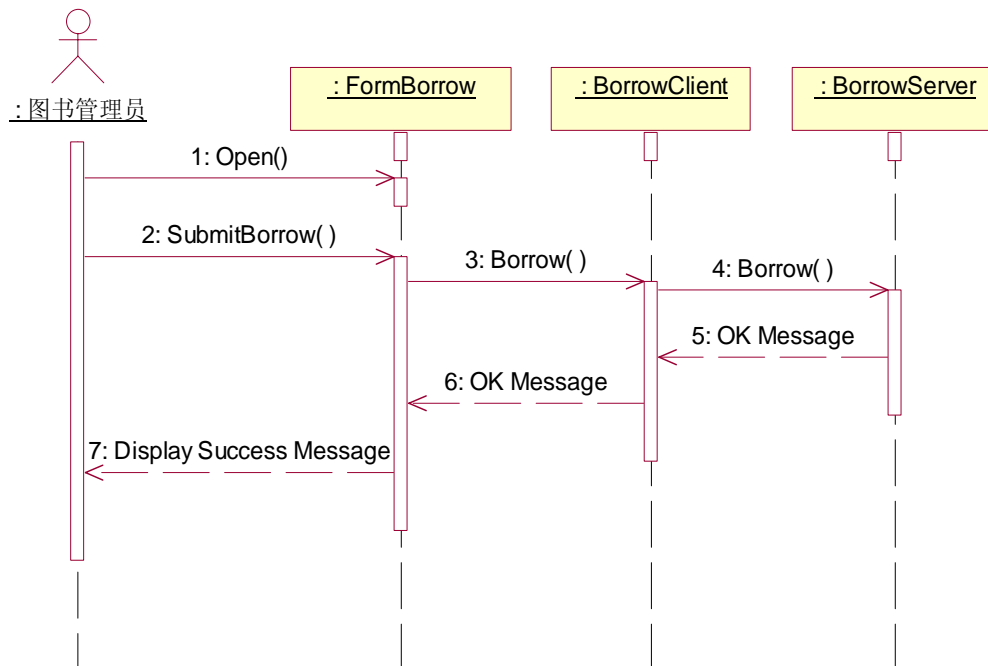


设计客户端

现在，我们为借书成功这个事件流来设计客户端。客户端在这里比较简单，他需要一个界面来接受输入，然后通过一个 BorrowServer 的客户端把借书的请求发送给 BorrowServer，我们把这个类设计成 BorrowClient。类图如下所示：



其 Sequence 图可以表示如下：



这样，对于借书这个用例，我们就完成了他的设计。依葫芦画瓢，我们就能够完成其他用例的设计。当然，在设计的过程中，我们可能采用一些技巧，来使得我们的设计更加有弹性，更加合理，这是对系统设计的优化。关于这个内容，最好的书籍莫过于著名的“四人帮”所著的《设计模式》一书了。

设计的最后工作，是根据我们设计的对象模型，把数据库的工作完成，并且，给出对象同数据库的映射关系。这个，就不是本文所准备讨论的问题了。另外，从某种意义上来说，数据库的设计，实际上是实现模型所需要完成的工作，他是实现我们的设计的一个部分的工作。这个观念，同原来的设计方法中，设计工作的主要任务就是做数据库设计的观念，不能不说是一个挑战。

可以看出，对于同样一个用例，在分析模型中的一个模型，在设计模型中被拆分成了两个部分。这时因为，在分析中，我们关注的还是系统的逻辑问题，而在设计中，我们必须给出实现这些逻辑的解决方案，包括系统的架构、系统的部署、应用的分布等。在分析模型中的一个类，在设计模型中可能会被映射成两个类，当然，也可能在分析中的多各类，会在设计中被映射成一个类。对于分析和设计的这些不同的侧重点和区别，是我们需要注意的。

实现模型——构造我们的系统

对于实现模型来说，其工作是非常清晰的，就是设计模型，转换成能够运行程序代码，这个工作，是我们所有的程序员每天都在做的事情，因此本文就不准备做过多的讨论了。提醒的一点是，同分析模型到设计模型的转换一样，在从设计模型向实现模型转换的时候，也会发生一些变化，这是正常的事情。

结束语

使用 UML 来为系统建模实际上是一个非常自然的过程，只要我们按照既定的合理步骤，采用

合理的方法，一步一步的进行深入的分析，就能够很好的完成我们的任务。在本文中，我们通过一个简单的系统，演示了这样一个基本的步骤和方法，展示了各个模型的不同点和相互之间的联系，以及他们的转换和演化过程，希望能够对大家在设计更加复杂的系统时有所启发和帮助。