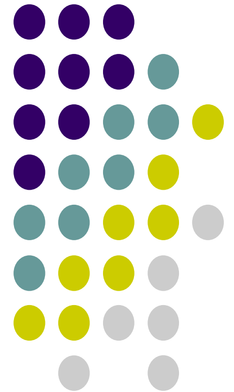


Caché 面向对象软件开发教程

第一版

作者：Jason, Jeffery, Robert, Ryan

序言

这本 Caché 中文教程的问世，使我们感到由衷的欣慰。它反映出 Caché 已得到我国更多软件开发企业的青睐和欢迎这一客观事实。因为他们的确需要迅速掌握和充分利用 Caché 这一不仅具有多维数据库架构、丰富的功能和自带有开发环境的新产品，来尽快满足最终用户迫切希望拥有能适应信息经济和 Internet 时代要求的新一代高性能的数据库应用系统的这一客观存在的渴望。

Caché 是新型的后关系型数据库(Post-Relational Database)，也是独树一帜的 e－dbms；而且它是一个先进而成熟的技术，它以 ANSI 和 ISO 标准 M 语言的独特多维数据模型为基石，为适应 Internet 迅猛发展对数据库信息技术的新需求，InterSystems 公司在多年来已得到国际上公认的以 M 技术开发和运行大型应用系统所积累的可靠经验的基础上，经过积极的开发和演变，推出了举世无双的 Caché 的数据库管理系统产品。在 Caché 中的对象模型具备了符合 ODMG 标准的完整的对象特性，而且融合了为 OLTP 专门优化过的 SQL 技术，并无缝集成了为实现快速开发 Web 应用的最新技术。这种数据库创新地摆脱了传统关系数据库的局限性，它在性能上全面超越了关系数据库，能够更好地满足 Internet 时代对于能处理复杂数据存储的数据库技术的迫切要求。

Caché 自 1997 年 9 月问世后，不仅得到了许多市场分析权威机构，例如 IDC、Bloor Research、Gartner Group 的 Dataquest、Aberdeen Group、Kervin Dick Associate、和 KLAS 等众多机构的好评,和包括 ComputerWorld、InfoWorld、Info Week、Health Informatics 和 Application Development Trends 等许多专业媒体的推介，而且在为包括医疗领域等各个行业创建高性能数据库应用信息系统的实践中呈现和发挥出了显著的作用和效益。

目前，Caché 数据库已在世界上 88 个国家得到广泛采用。同样她在我国也得到了日益重视和积极采用，例如从国外引进后经过汉化的 MedTrak 一体化医院信息系统，由于它是基于 Caché 数据库数据库平台的，因而它不仅在临床信息管理上成功集成了电子病历、检验科室信息处理、和 PACS 系统的功能，并且在性能上在系统的稳定可靠运行、节约数据库存储空间、和易于进行数据库的管理维护等方面体现出了众多优越性，因此已开始在北京和哈尔滨等地推广到多家医院使用。更为可喜的是国内一些重要软件开发厂商经过一段时间对 Caché 后关系型数据库技术进行了测试、熟悉、和试用之后，已经开始正式采用 Caché 数据库自主开发应用系统。近几年来 InterSystems 公司在各地，如在北京、上海、广州、深圳、江苏、浙江、辽宁和四川等省市为我国用户相继举办过的 Caché 技术培训班也从另一个侧面反映出 Caché 已日益得到中国用户的关注和喜爱。

现在，我们高兴地看到 InterSystems 公司最近已多方面加大了对它在中国市场推进的支持力度，其中包括在上海和北京都已建立了办事处、和建立了中文网站。而为帮助中国用户和读者编写的这本 Caché 数据库用户手册又是一个非常重要的举措。

这是一本以前难得一见的专门关于 Caché 数据库及其应用的中文工具书籍。为了便于了解和掌握后关系型数据库及其应用，本书撰稿者们在这本相当全面而深入浅出的手册中，不仅对 Caché 后关系型数据库的各个组成部份及其优点和特点作了综合的阐述，而且具体介绍了 Caché 数据库的安装和运行方法，着重在手册中重点介绍了 Caché 数据库服务器端开发、Caché ObjectScript 对象脚本语言、CSP 等关键组成部分及其技术特点和使用方法。并详细具体地提供了如何用 JAVA 开发基于 Caché 的应用、用 CSP 开发 Web 应用、又或是从 Delphi 客户端访问 Caché 数据库服务等应用示例。使读者能很快了解和掌握如何使用 Caché 提供的自带开发环境与便利、以及利用 Caché 与流行的应用开

发工具间的无缝链接，通过创建 Caché 对象类及其实例等步骤，来快速有效地开发和部署各种应用系统。同时在手册中还专门列出了数据迁移这部分内容，对如何将数据库应用从关系数据库迁移到 Caché 后关系型数据库，或者利用 Caché 的 SQL Gateway 实现不同类型的数据库之间的集成整合都作了详细的介绍。无疑，手册中的上述这些内容、以及提供的一些参考资料附件，都将可以在很大程度上帮助读者迅速入门。

总之，这本手册不仅可以用作了解和掌握 Caché 后关系型数据库及采用它进行应用开发的培训教材，而且也可以作为 Caché 的一本参考性手册使用，供程序员查阅和其他读者自学之用。

王 继 中

中国医药信息学会 M 技术专业委员会主任委员

2004 年 11 月于北京

目录

序言 I

第一章 Caché 基本概念 1

第二章 初尝 Caché 面向对象开发 22

第三章 Caché ObjectScript 语言及其语法 86

第四章 以 Caché 开发应用程序 160

第五章 以 Java 开发 Caché 应用程序 177

第六章 以 Delphi 开发 Caché 应用程序 219

第七章 以 CSP 开发 Web 应用程序 307

第八章 平行数据迁移和 Caché SQL 392

附录：Caché ObjectScript 参考资料 431

第一章 Caché 基本概念

1　引言

1970 年之前出现了关系型数据库理念和在 1980 年出现了第一个商品化的关系数据库产品之后，关系型数据库在它的技术演变和应用规模上有了很大的发展，曾被誉为数据库领域的重大的创新之一。关系型数据库用统一的数据结构取代了以前单一的、结构不一的数据库，将数据以表格形式进行存储；任何懂一些 SQL 查询语言的人，都可以访问数据，它在这些方面取得了成功，然而，在实践中人们发现关系型数据库系统虽然技术已相当成熟，但其局限性也是显而易见的：它能很好地处理所谓的“表格型数据”，却对技术界出现的越来越多的复杂类型的数据无能为力。作为一种旧技术，关系型技术有很多的局限性，使得它在当今世界的适应性大大降低，主要表现在它的效率性能、可扩展性、和使用的简洁性较差，以及难于适应和现代快速应用开发技术相匹配的新需求。

计算机应用程序的广泛使用、复杂性的快速升级和当今系统不断增长的处理需求已经超过了关系型技术的能力。许多要求高性能和高扩展能力的关键应用有许多是从来没有在关系数据库实现或迁移到关系型数据库上的，现在即使很简单的应用也开始触及到传统关系型技术的性能极限。

关系型数据库和当今开发技术的“阻抗不匹配”现象已经成为了一个严重的问题，它使得开发过程更加复杂，失败机率大为增高。尽管表格结构的简洁可以支持了强大的 SQL 查询语言的使用，但现实世界的复杂数据是很难分解为这种简单的行列结构的。其结果就是是数据库应用中产生了大量的表，表和表之间

的关系就变得很难记忆和表达。行列结构是简单，但是留给程序完成的外连接、存储过程、触发器就不是这么简单了。

现代的应用程序通常都是使用面向对象的技术编写的，这种技术更加简单、直观、可靠和有效，并且可以大大缩短开发周期和提高健壮可用性。所以，我们需要一种可以解决这个问题的数据库产品，这就是融合了对象技术和关系两大技术长处的 Caché。它既克服了以往关系型数据库和不成熟的“纯粹的对象数据库”存在的缺陷，又不会出现在所谓的“对象－关系数据库”中存在的难以实现高性能的弱点。在 Caché 数据库中具有与众不同的独特的多维数据结构，并且同时具备高速性能、高可伸缩性、面向对象和面向 Web 应用的特色；它不仅不会出现关系型数据库那种固有的局限性，而且保持了可以继续使用 SQL 查询语言的便利。

2 后关系型数据库 Caché

Caché 是新一代高性能数据库技术，被誉为创新的“后关系型”数据库（Post-relational Database）。作为后关系型数据库，它整合了对象数据库访问、高性能的 SQL 访问、强大的多维数据访问——这三种方法能够天衣无缝地同时访问相同的数据。数据只要在单一的整合 数据字典中描述一次，就可被这三种方法访问。Caché 提供了比关系型技术更加高效的性能，更大的扩展性，更快速的编程能力和更加便捷的使用性能。

Caché 提供不仅是一种单纯的数据库技术。在 Caché 中包括一个应用服务器，这个服务器提供高级对象编程，并且可以很容易地与很多技术集成。Caché 还提供高性能的运行环境，这一运行环境采用了独特的分布式数据缓存协议技术。

Caché 还在另一方面远远胜过了传统的数据库技术。Caché 为开发复杂的、基于网页应用程序提供了丰富的集成开发环境。Caché Service Page（CSP）技术可以进行快速开发，动态产生。上千个用户甚至在在比较差的硬件条件下也可以同时访问数据上的应用程序。

对于那些不基于浏览器的应用，用户接口可以用任何一个流行的程序设计语言来编写，例如可以任意选用程序员自己熟悉的 VB，Delphi，Java，或者 C++来编程。剩下的工作都交给 Caché 运行，这样可以得到最好的结果（最快的编程效率，最高的性能，最低的维护成本）。另一方面，Caché 也提供了与其他技术的交互，支持大多数通用的开发工具，所以开发方法的选择范围很大。

3 Caché 的数据存取模式

3.1 Caché 架构

后关系型数据库 Caché 的特点是高效率、很好的延展性、应用程序的快速开发能力、和低成本。这些特点可以从 Caché 基本架构得到反映。

Caché 使用的是一种高效的多维数组形式存储数据，即使在使用比其他数据库系统配置更低的硬件条件下高负荷运行也能保证高效率。此外， Caché 能运用各种技术存取数据，开发者可以选择使用自己熟悉的和易于得到的开发工具，这就大大提高了开放性和应用程序开发的能力。

3.2 多维数据引擎

与关系型数据库不同，Caché 以多维数组存储数据，而关系数据库以两维表存储数据。Caché 除了使现实数据建模成为可能，还因为减少了表连接等处理过程（这在关系型数据库中是非常频繁的），所以运用多维数组能更快地存取数据。Caché 从它的独特的数据机构中获得了许多强大功能，关键之处在于 Caché 的数据库引擎为我们提供了一套完整的服务：包括数据存储、并发管理、事务处理、和过程管理，这就为我们提供了强大的可用于建立复杂管理系统的功能和工具。

Caché 的另一个提高性能的特征就是 Caché 具有独特的分布式缓存协议，它大大减少了分布式系统中的网络通信吞吐量。在有的客户所进行的数据库应用程序性能的比较测试中，Caché 数据库的响应性能要比关系型数据库快 20 倍。

虽然在 Caché 中数据是以多维数据结构形式存储的，但 Caché 允许开发者用任何他们选择的方式进行数据建模：对象，表格，或者多维数组。Caché 拥有一个非常易用的图形界面开发环境来建立和开发 Caché 对象。 Caché 还可以接受从 Rational Rose 对象建模工具或以 DDL 文件（数据库定义语言）形式的导入。

Caché 统一的数据架构使所有数据都能以对象和表格形式被访问。既不需要为从一种数据形式到另一种数据形式进行映射，也不需要为不同数据形式的转换进行处理。统一的数据架构提高了编程效率和应用程序性能。

Caché 提供了可以用多种技术编写数据库和业务逻辑的能力。 Caché 的 ObjectScript 支持所有数据存取方法：对象， SQL ，多维和嵌入式 HTML 。 Caché Basic 与 Visual Basic 非常相似，只是做了很少的调整扩展，以便利用 Caché 独特的性能。

3.3 Web 存取

与 InterSystems 公司的核心价值一致， Caché 提供了与 Web 连接和其应用程序开发平台，这些都进一步提高了效率和延展性。在 Caché 独特的网络架构中， Caché 服务器页面（ Caché Sever Pages ）在数据服务器上运行，与他们要存取的数据放在一处。这样不但提高了效率而且通过降低网络服务器的负荷而大大提高了延展性，从而能处理更多的浏览器请求。

Caché 运用对象技术的快速开发能力来开发 Caché 服务器页面。每一个 Caché 服务器页面本身就是一个对象，它能从 InterSystems 提供的系统对象中继承会话管理方法和各层安全策略。这就使应用程序开发者不需要再开发大量

单调的系统代码来维护使用会话期（ Session ）对象的状态。利用对象继承也保证了能快速地浏览应用程序的所有页面。

此外，因为允许网页设计者和应用程序开发者可以平行地进行开发来完成任务，Caché 便简化和加速了 web 应用开发过程。网页设计者可以运用熟悉而易用的工具，通过添加
Caché 应用程序标记（ CAT ）来增加网页功能，这项工作类似于通常的添加标准的 HTML 标记的工作。 Caché 不仅提供了一些标准功能的 CAT ，而且还可以户自己定制 CAT 。当应用程序开发者编写 CAT 来完成实用的功能时，不需要考虑包含它们的网页样风格样式的设计。这样就能更快和更有效地开发 web 应用程序，缩短产品进入市场的时间

3.4 对象存取

当今，所有新应用程序的开发都是运用对象建模技术。用对象来进行数据建模使开发者能以一个自然而直观的方式思考数据。因为对象是模块化的，接口定义明确，所以它们可以重复使用，可以被多个应用程序共享，这样编程的效率就能大大提高。

Caché 支持多种对象建模技术，包括多重继承，封装，多态，引用，采集，关系和 BLOB 。 Caché 对象能通过 Caché Studio 图像界面和 Rational Rose（一种流行的对象建模工具， Caché 与之有双向接口）开发。与一些“对象－关系型”数据库系统不同， Caché 可以改进数据模型，使得对象定义能不断修改以适应应用程序变化的需要。 Caché 统一的数据架构使得所有 Caché 对象都自动兼容 ODBC 。

Caché 对象也和大量面向对象的工具和技术相兼容。使用 Java 和 C++ 的开发者和使用 COM 界面的工具（例如 Visual Basic 和 Delphi ）的开发者都可以使用它们。 Caché 也支持双向 CORBA 界面。

3.5 SQL 存取

便于使用的 SQL 查询语言已成为了访问数据库的一个通用标准，在关系型数据库的全盛时期，它们被广泛运用，即使是今天，许多软件应用程序，尤其是那些用于数据报表和分析的程序，都使用 SQL 作为它们的查询语言，同时需要支持 ODBC 或 JDBC 的数据库。Caché 允许将通过 SQL 数据存取作为访问方式之一， 使得 Caché 可以兼容所有这些使用 SQL 的应用程序。此外， Caché 的 SQL 网关（SQL Gateway）功能使得 Caché 应用程序能从关系型数据库中存取数据，这个功能在需要从不同来源整合数据时就十分有用。

一些开发者希望把已有的应用程序从关系型数据库上移植到 Caché 上以利用 Caché 的高效率和先进的对象技术。 这也是可以满足和实现的，Caché 能根据在 DDL 文件中的关系型表格定义来创建数据结构。运用 Caché 统一的数据架构，每个表格被定义成为一个简单对象或复杂结构的组成部分。运用 SQL 网关等技术，数据就能从关系型数据库转移到 Caché 上。并且利用 Caché 网关或者 InterSystems 的另一个重大新产品 Ensemble 软件也可以搭建出一个 Caché 数据库和关系数据库等其他类型数据库共存并用的集成整合工作环境，以满足不同用户的需要。

3.6 多维存取

多维数据存取还可以使 Caché 能兼容 InterSystems 以往开发的以 M 技术为基础开发的其它数据库产品，InterSystems 的 Open M 产品等都使用与 Caché 一样的多维数据结构。因此只要用户感觉需要升级就可以迁移到性能更强的 Caché 数据库新环境下工作。

4　Caché 应用服务器

4.1　丰富的功能

Caché 应用服务器使快速开发复杂数据库应用成为可能，并且能够以更高地性能运行他们，更容易支持这些应用程序。

Caché 应用服务器具体提供了以下功能：

- Caché 虚拟机可以运行两种内嵌脚本语言——Caché ObjectScript 和 Basic。
- 可以透明地访问同一台或者不同的计算机上的 Caché 多维数据服务器。
- 带客户端缓存的连接软件允许使用通用的技术（如：Java，C++，C＃，COM，.NET，Visual Basic，Delphi）快速访问 Caché 对象。Caché 能自动连接客户端和应用服务器。
- 能与 SOAP 和 XML 相兼容
- 使用 ODBC 和 JDBC 进行 SQL 访问，具有客户端复杂的缓存和高性能的应用服务
- 能够访问关系型数据库
- 提供高性能的 Caché 服务页技术，更便捷地开发 Web 应用程序。
- Caché 工作室（Caché Studio），它是一个集成的开发环境， 能够快速地开发和测试 Caché 应用。
- 脚本语言的代码存放在数据库中，并且能够在线更改，这些更改能够自动地传递到所有应用服务上面。

4.2 Caché 虚拟机和脚本语言

Caché 应用服务的核心是非常快速的 Caché 虚拟机，它能够支持两种脚本语言，即

- Caché ObjectScript
- Caché Basic

Caché ObjectScript 是一种强大而且易用的面向对象语言，并且有非常灵活的数据结构。

Caché Basic 为 Basic 编程人员提供了另一种方便使用 Caché 的方法。与 VBScript 相似， Caché Basic 已扩展成能够直接访问 Caché 多维数组。

在 Caché 虚拟机中的数据库访问已经经过高度优化。在虚拟机中的每个用户进程通过调用共享存储器直接访问多维数据结构，这个共享存储器可以访问共享的数据库缓存。所有其他的技术（Java，C++，ODBC，JDBC，等等）也通过虚拟机连接来访问数据库。

4.2.1 完全的交互性

既然 Caché ObjectScript 和 Caché Basic 都是运行在相同的 Caché 虚拟机上的，它们是完全可以交互的。用两种脚本语言都可以编写任何一个对象方法，同一个类还可以同时使用这两种脚本语言。 一种脚本语言可以调用另一种脚本语言编写的代码。两种脚本语言可以共享变量，数组和对象。

4.2.2 快速开发/灵活开发

多数成功案例的实践证明，编程人员能够更加快速地开发应用程序。使用脚本语言—Caché ObjectScript 和 Caché Basic——编写代码，使得应用程序运行时更加快速，更具有扩展性，因为这些代码是运行在 Caché 虚拟机上的。并且，这些代码对硬件和操作系统没有变更的要求。Caché 能够自动地处理操作系统和硬件的不同。

5 Caché 在分布式系统中的使用

5.1 分布系统的企业缓存协议

InterSystems 的企业缓存协议（ECP）是一项具有极高性能和高扩展性能的技术。通过该协议，能够使得分布式系统中的计算机共享彼此的数据库。使用 ECP 不需要重写或者改变应用程序，应用程序却能像使用本地数据库一样使用其他系统上的数据库。

下面介绍一下 ECP 是如何工作的。每个应用服务器都含有自己的数据服务器。数据服务器能够处理存储在本地系统的数据或者通过 ECP 协议从其他计算机上传输过来的数据块。当一个客户端发出一个要获取信息的请求，Caché 应用服务器尽量从本地缓存中获取数据满足请求。如果本地缓存不能满足这个请求，Caché 应用服务器就会从远程 Caché 数据服务器上获取必要的数据。响应这个请求的应答中包括数据块，所需的数据存储在这些块中。这些数据库块就缓存在应用服务器中，这样运行在这个服务器上的所有程序都能够使用这些数据。ECP 会自动通过网络维护缓冲数据的一致性，把变化的数据发布到数据服务器上。

从 ECP 中获得的高性能和高扩展性是激动人心的。因为客户端常常使用的是本地的缓存数据，所以客户端能够获得快速的响应。缓存大大降低了数据库和应用服务器之间的网络通信，所以在给定的网络环境下，能够支持更多的服务器和客户端。

5.1.1 易用性——无须进行应用程序的改变

使用 ECP 对于应用程序是透明的。在单个服务器上运行的应用程序不需要做任何改变就能运行在多服务器环境下。如果要使用 ECP，系统管理员只需要简单地指定一个或者多个数据服务器来对应一个应用服务器，然后使用命名空间应用来表明对某些或者所有 global 结构（或者 global 结构中的一部分）的应用优先选用哪个远程的数据服务器。

5.1.2 配置的灵活性

每个 Caché 系统都能够作为其他系统的应用服务器或者数据服务器。ECP 支持应用服务器和数据服务器的整合以及任何点对点的拓扑结构（最大可以达到 255 个系统）。

5.1.3 分布系统中的容错机制

使用企业缓存协议（ECP）的分布系统，一旦出现暂时的网络断连或者数据服务器死机并重启，服务器会自动重新连接。如果在规定的时间内重连成功，那么应用服务器会重发没有完成的请求，继续操作，这对使用远程应用服务器的用户而言没有明显的影响，只会出现一点延迟而已。但是，如果在规定的时间内重连没有成功，未完成的事务就会回滚，用户进程就会提示出错信息。

在某些配置中，数据服务器支持故障切换，可以转移到一个镜像服务器或者簇成员上，以进一步提高稳定性。新的服务器替代故障数据服务器继续进行处理，这样就不会中断操作。

5.1.4 簇

数据库簇支持自动故障切换功能。在一个数据库簇中，多个计算机共享相同的磁盘驱动器，使用簇功能可以协调共享访问或者排他地访问磁盘。如果一个计算机发生了故障，它的过程就丢失了，但其他的计算机继续执行。发生故障的计算机上正在处理的事务会自动回滚，然后这些用户可以登陆到其他计算机上。在动态分配用户到簇计算机上时，通常使用负载平衡机制。

虽然数据库簇提供操作的灵活性，也提高了稳定性，但他们通常需要比其它系统更多的系统管理，并且需要特殊的硬件和操作系统的支持。

5.2 镜像服务器

每个数据服务器都可以有一个备份服务器，称为镜像服务器，它能经常读取主服务器的数据，更新备份服务器上的数据库。

如果主服务器发生了故障，备份服务器能够立即使用，但所有未完成的事务处理都需要回滚。 备份服务器能够用来产生报表或者进行查询。当主服务器在工作的时候，一般不更新备份服务器。如果服务器之间的连接暂时中断了，镜像服务器就可以替代直到连接恢复。

6 Caché 的优势

6.1 Caché 的面向对象技术和多维结构的优势

- Caché 完全面向对象，为开发高效事务处理应用系统的程序员提供了对象技术的所有强大功能。
- 直观的数据建模：对象技术使得开发者能以简单和真实的方法思考和实现使用数据建模，甚至是十分复杂的数据模型，这样就提高了应用程序开发的效率。
- 快速应用开发：对象技术是提高编程效率的强而有力的工具。开发者能够用简单而实际的方法来思考和使用对象——甚至是极其复杂的对象，这样大大加快了应用程序开发的进程。同时，对象本身的模块化和交互性能使得应用程序的维护变得相对简单， 使程序开发员在多个项目中可以积累和利用他们的原有工作成果。封装，继承和多态性这些技术使得类可以在应用程序之间能够重用和共享，这使得开发人员可以在多个项目之间胜任他们的工作。
- 灵活性：Caché 的三种访问模式——对象，SQL，直接访问多维数据结构——能够对同一数据进行并发操作。 这种灵活性使得程序员能够充分自由地考虑数据，采用每个程序所需的最合适的数据访问方法。
- 减少工作量：Caché 的统一数据架构能够在一种定义下自动用对象和表两种方式描述数据。 这样就没有必要进行代码转换了，不仅应用程序的开发效率提高了，而且应用程序的维护性也变得简单易行。
- 继续延用现存的技术和应用程序：程序员能够继续延用现存的关系型技能，逐步引入对象技术。

- 通过使用高效多维数据模型以及稀疏数组存储技术来替代传统的两维表，只要少量的磁盘读写就能完成数据访问和更新。降低 I/O 意味着应用程序运行得更快。
- 延展性：多维数据模型允许基于 Caché 的应用程序扩展到数千个客户端同时使用，而不影响其高性能。这是因为不同于关系型模型，多维数据模型的数据库的大小和复杂性并不显著地影响数据的访问。 另一方面，事务处理访问数据时不需要进行表连接或者从这个表跳跃到那个表。
- Caché 更新数据时使用逻辑锁，而不是锁住整个物理页， 这对提高并发性能也是一个很大的贡献。
- 使用 Caché，开发过程变得更快，因为 Caché 的数据结构支持复杂数据的简单存储，并且不需要复杂的声明或者定义。 由于允许相同的语句来访问本地数组，直接对 global 的访问非常简单。
- 成本有效性： 与大小相同的关系型应用相比，基于 Caché 的应用对硬件要求很低，不需要配置专门的数据库管理员。系统管理和操作非常简单。

6.2 Caché 和 SQL

- 在 Caché 中有更快的 SQL 查询：通过把 Caché SQL 嵌入到 Caché 后关系型数据库中，和以往关系型数据库相比较，应用系统在用 SQL 查询时能大大提高其性能。
- 在 Caché 中能实现更快的开发还表现在 SQL 查询语句的书写更加直接，只需要用少量的代码行。
- 与现存的应用系统和报表编辑器兼容：Caché 自带的优化 ODBC 和 JDBC 驱动器提高了与传统应用的交互性能，包括最流行的数据分析

器和报表编辑器。

6.3 Caché 的事务处理

- 快速查询中， 通过使用事务型位图索引技术，使用者能够快速搜索大型数据库——通常查询成百万记录只需要不到一秒时间， 这种数据库上运行的主要是用于处理事务的系统。
- 实时数据分析：Caché 的事务型位图处理允许在实时更新的数据上进行实时数据分析。
- 低成本：即不需要第二台计算机来服务于数据仓库或者决策支持，也不需要每天把数据迁移到第二个系统，更不需要数据库管理员维护这项工作。
- 高扩展性：快速事务型位图索引提高了系统的性能，应用系统一般有大量的数据，需要定期维护和查找。

6.4 Caché 的分布式应用

- 高扩展性：当使用不断增加时，Caché 里的企业缓存协议（ECP）允许任意增加应用服务器。 每个增加的应用服务器都好像在使用本地数据库一样。 因此如果磁盘吞吐量成为瓶颈，可以采取增加数据服务器的措施，并且数据库可在逻辑上分区。
- 更高的可用性：因为用户分散在多个计算机上，一个应用服务器的失败只影响一小部分用户。 如果一个数据服务器死机并重启，或者出现暂时的网络断连的情况时，应用服务器仍可以继续进行处理，而用户除了能感受到一点延迟之外没有什么影响。在某些配置中，数据服务器还支持失败切换功能， 数据可以转移到镜像数据库或者

簇系统中，这样可以进一步提高稳定性。

- 更低的成本：大量低成本计算机能够联合成 一个非常强大的系统以支持大量的事务处理，称为“网格计算机”。
- 透明的使用： 应用程序不需要为了使用 ECP 而作特别的修改，Caché 应用可以不加改变，而会自动使用 ECP。

6.5 Caché 的日常管理

- 简单的系统管理：Caché 的日常管理非常简单，不仅可以自动化地完成特定的任务（例如备份工作）， 还可以免除掉一些对其他类型数据库而言是日常一定要做的管理任务。Caché 由于只需便捷的系统管理，还可以因而减少故障的产生和降低维护管理费用。
- 不需要很多的 DBA（数据库管理员）： 因为 Caché 很容易管理，多数应用甚至不需要 DBA。
- 重新配置：可以动态地进行 动态命名空间映射允许系统在动态环境下重新配置，不需要中断程序的正常运行。
- 成熟的 24 小时运行：在线备份，镜像服务器，数据库簇等技术支持了 24 小时连续运行、或无缝地从故障恢复。

6.6 Caché 服务器端的开发语言

- 使用 Caché ObjectScript 开发复杂的数据库应用时，程序的开发比用其他任何语言都要快，常常快 10 到 100 倍。快速就意味着项目有更多的成功机会，并且需要更少的编程人员，和能够按应用需求的变更更加快速地调整。
- 更容易的培训学习： Basic 可能是世界上最知名的计算机语言。使

用 Visual Basic 语言或其他 Basic 的开发员可以利用原来掌握的 Basic 语言知识立即进行程序的开发。Caché 对象模型也是很容易学习的。

- 更具伸缩扩展性： Caché 虚拟机能够直接访问数据库，这样使得应用程序能够运行得更加快速， 并且应用程序在低性能的硬件上也能够扩展到支持上万个用户同时使用。
- 移植灵活性: 运行在 Caché 数据库上的程序代码往往不需要改变就能直接运行到其他的硬件和操作系统上。 代码分别经过对象封装后存储在数据库中，便于理解和查找，并且能自动传播到所有应用服务器上。
- 当开发者在他们熟悉的环境下使用他们熟悉的工具进行工作时，效率也总是会要高一些。 并且由于具备能同时提供 SQL 访问、对象访问和直接访问方式，Caché 就能够支持很多的流行开发技术和工具。

6.7 Caché 与 JAVA

- 灵活性:当 Java 程序员要访问 Caché 对象的时候，他们能有很多种选择。他们可以使用 SQL 来访问，也可以通过使用 JDBC， 或更自然地把对象映射为 Java 类或者 Enterprise Java Beans 然后再进行访问。
- 高性能: 不管 Java 应用是如何链接到 Caché 数据库的，总能从 Caché 的高性能和高扩展性中受益。
- 与 J2EE 的自然兼容意味着更快的程序开发 Caché 类可以很容易地映射为 EJB，这就为 J2EE 开发员提供了一种链接到 Caché 后关系型数据库的简单方法。当使用 Bean 管理的永久性进行 Caché 类的

映射时，Caché 能自动产生 EJB 可用方法来访问 Caché 数据库。因为开发者不需要手动处理代码的永久性方法，应用就能够更加快速地完成。

6.8 Caché 的 Web 应用

- 与.NET 兼容：由于能够很好地融入.NET 的框架，这样使得开发者可以选择 Web 服务，XML，COM 或者 ODBC 来链接客户端。
- 与 XML 的便捷链接：Caché 充分利用了它的多继承功能，给每个 Caché 类提供了 与 XML 的双向接口。结果使得 Caché 类可以很容易并且快速地转化为 XML 文档和模板。 同样，XML 文档和模板页也可以很容易地转化为 Caché 类定义和对象。
- 更快的 XML 应用的快速开发：因为 Caché 本身的多维数据结构能够与 XML 文档很好地匹配，所以开发员不需要手动编写 XML 和 Caché 数据库之间的“映射”。因此，内嵌 XML 的应用能够更快地开发。 这样，处理和超负荷瓶颈现象也相应减少了，所以这样的应用程序也就能运行地更快。
- 及时的 Web 服务：任何 Caché 方法只需要点击几下鼠标就可以发布为 Web 服务。 当某项服务被触发时，Caché 能够自动产生 WSDL 描述符和 SOAP 响应。 已有的 Caché 应用能够很容易地打包为 Web 服务，这样新的 Web 服务应用能够很快地建立。
- 选择的多样性：Caché 允许程序员使用任何开发环境及工具来编写所需的用户接口和业务逻辑。 通过使用映射、对象服务器、以及网关， 基于 Caché 的应用程序可以和使用其他技术如 COM，.NET，C++，Java，EJB 和 SQL 的程序进行交互。
- 与流行的工具链接：Caché 提供了与 对象建模工具（如 Rational

Rose）和网页设计工具（如 Dreamweaver）的高效链接。这就使得使用这些工具的开发者能够非常快速的创建基于 Caché 后关系型数据库的应用。

- 快速应用开发：Caché Studio 工作室是一个创建、调试、测试 Caché 应用的高效环境和有力工具。通过提供的大量的向导，利用这个 Caché 工作室可以消除很多开发过程中的繁重工作。
- 面向对象的编程加上 Caché 向导使得快速开发基于浏览器的复杂数据库的应用成为可能。
- 通过自动在浏览器上保存状态信息，网络传输活动就会减少，服务器上的负载也会减轻，因为应用程序不必因为每页的请求而访问数据和保存文档。这样一来，应用程序的编程变得更为简单。
- 用动态服务器页面和 Caché 应用服务器的好处是响应请求的灵活性很大、Web 应用能在避免出现 Web 服务器上应用软件错误风险的情况下更快地执行，并且能够有一个更丰富和方便的编程环境。

第二章 初尝 Caché 面向对象开发

1　引言

我们将通过一个小型的案例逐步了解如何以 Caché 实现面向对象设计、服务器端开发所需要的技术和工具，为以后在 Caché 上进行开发建立基础。

在本章里，我们亦会逐步安装及配置 Caché 服务器。

2 安装 Caché

2.1 安装前准备

安装之前请查看 Caché 安装的系统要求，而我们将要安装的是 Caché 的 **5.0** 版本。

运行 Caché 的基本系统要求如下表：

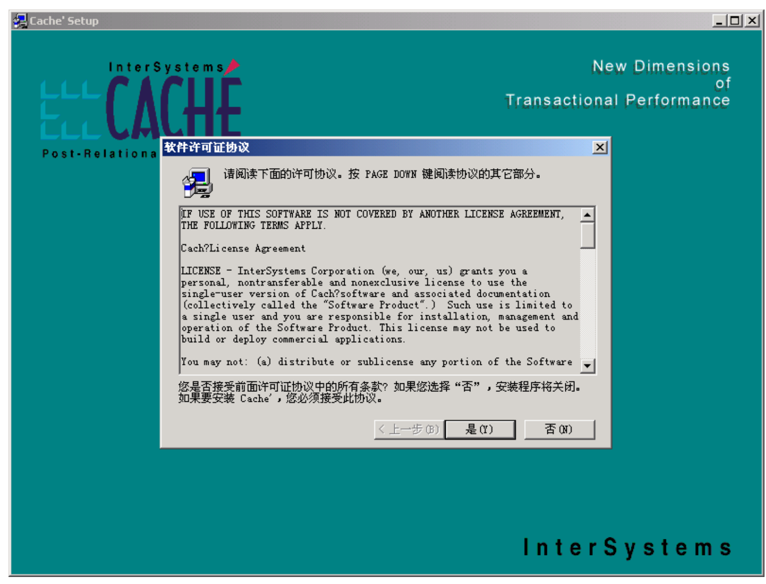
	Windows	Unix / Linux
硬盘空间	173MB – 270MB	140MB – 151MB
内存空间	128MB 或以上	128MB 或以上
支持版本	Windows Server 2003 Windows XP Windows 2000 (SP2 或更高)	**Alpha Tru64 UNIX** 5.1, 5.1A, 5.1B **HP-UX** 11, 11i **IBM AIX** 4.3.3, 5.1, 5.2 **Red Hat Linux** 7.3, 8.0, 9.0 **Sun Solaris** 8, 9 (64-bit only) SuSE Linux 8.0
其它	需要系统管理员(Administrator) 特权	需要 root 特权

* 本单元将只介绍在 Windows 上安装 Caché 的步骤。如果需要了解在其它平台上安装的方法，请参看联机文档：
http://127.0.0.1:1972/csp/docbook/DocBook.UI.Page.cls?KEY=GCI

* 我们可以通过光盘，本地硬盘或网络进行安装。要获得安装文件，可以通过以下网址下载最新版本的 Caché：
http://www.intersystems.com/cache/downloads/index.html

2.2 安装过程

准备好安装文件之后，我们便可以开始安装 Caché：

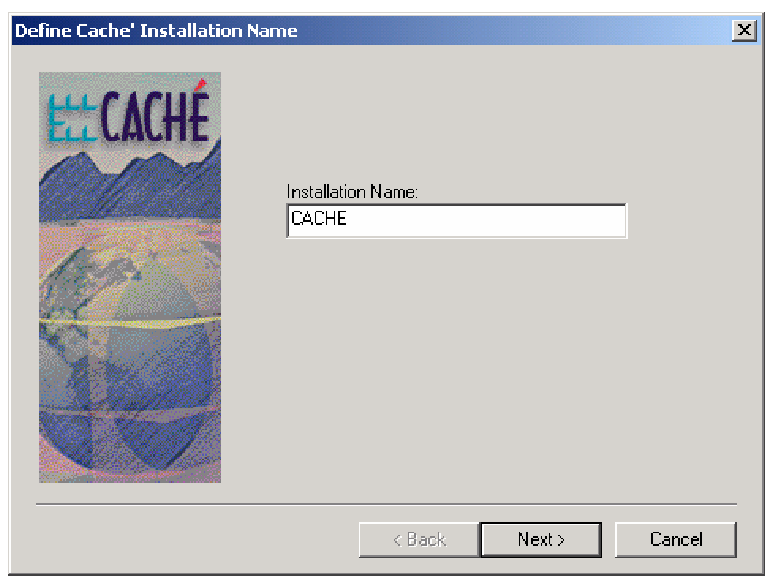


步骤 1：

在 Windows 中，运行 安装文件目录下的 SETUP.EXE 文件。

首先弹出的是 Caché 安装程序界面以及使用协议，阅读协议后，如果同意协议内容，

点击 按钮 "是"，进入下一页面。



步骤 2：

如果本机操作系统是第一次安装 Caché，将会提示输入安装名称，默认值为“CACHE”。

* 安装名称又称配置名称，当一台计算机安装了多个 Caché 时，此名称用来进行标识。

* 如果本机已安装了 Caché，安装程序将会显示出当前已安装的 Caché 列表，我们可以选择覆盖原有的 Caché 或添加一个新的 Caché。

点击 按钮 “Next>”，进入下一页面。

步骤 3：

在本页面中设定安装类型和安装路径。

Caché 安装程序中共有三种安装类型：

a. 标准(Standard，默认选项)。安装 Caché 服务器和客户端的所有工具。

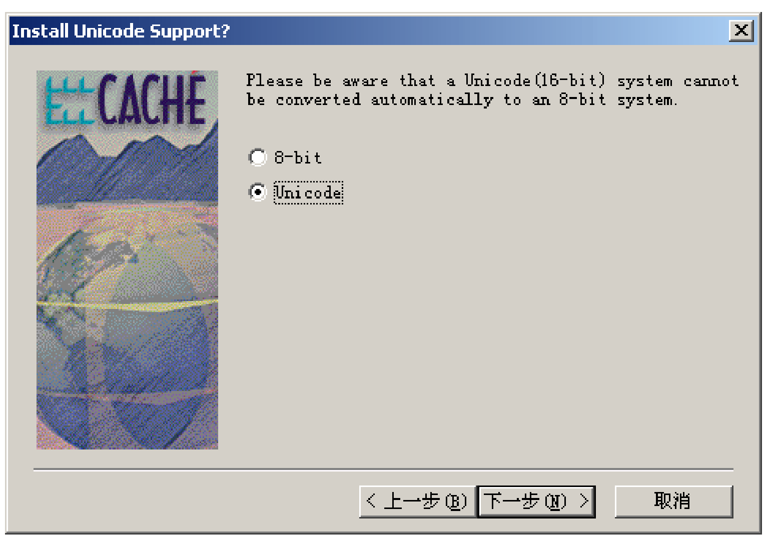
b. 客户端(Client)。只安装 Caché 的客户端的所有工具。

c. 自定义(Custom)。允许你自己定制要安装到你的计算机上的各个部分的清单，例如：只作 Web 服务器安装。

设置目标文件夹：

安装路径默认为“c:\CacheSys”。你可以点击“浏览”来更改希望安装在那个文件夹。

点击 按钮“下一步>”，进入下一页面。

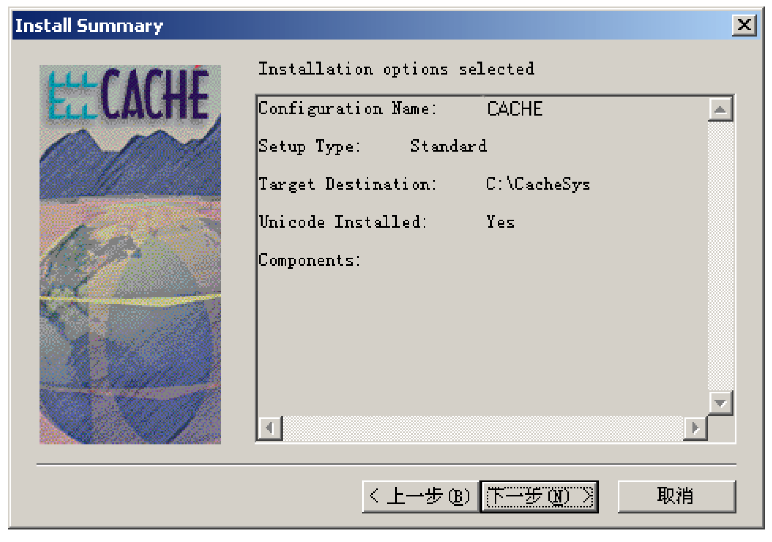


步骤 4：

在本页面中可选择设置成 Unicode 支持（16-bit）。(*否则系统默认将会是自动设置成为 8-bit 的)

选择 “Unicode”，这样 Caché 就可以支持中文显示。

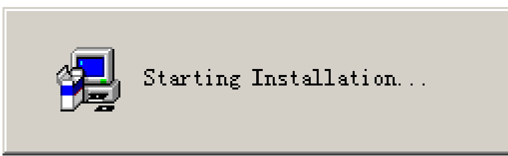
点击 按钮 “下一步>”，进入下一页面。



步骤 5：

本页面列出了前面步骤中所输入的信息以待最后检查。检查无误后，

点击 按钮 “下一步>”，便开始安装过程。



* 此时 Caché 将开始安装

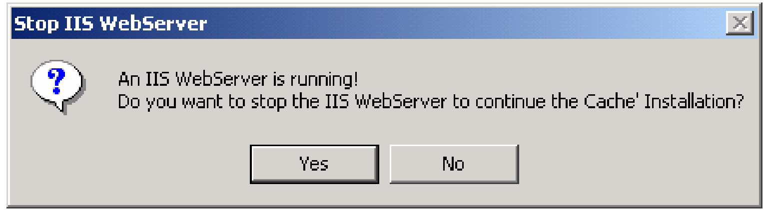
步骤 6：

安装过程将需要要持续一段时间。在这期间安装进程可能会暂停，进行提问：

a. 如果安装程序没有发现许可证(License Key)，将会提示是否现在安装许可证，

选择“否(N）”时。我们可以在 Caché 安装完成后再安装许可证。

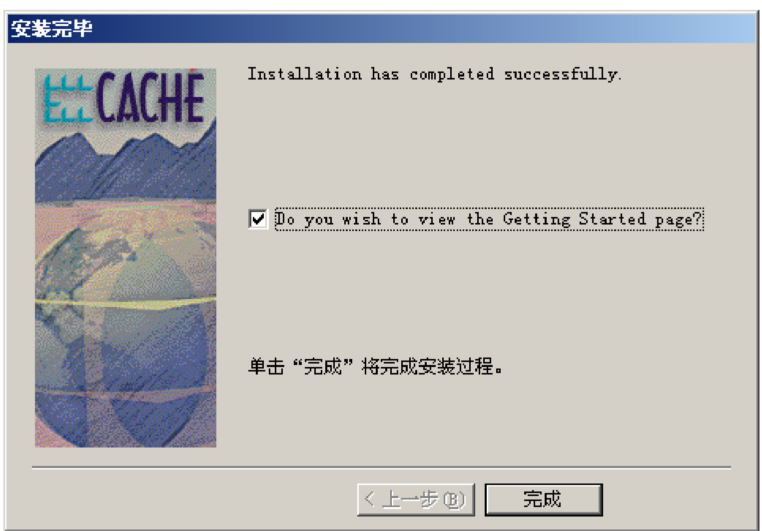
b. 如果安装程序发现 IIS(Internet Information Service)服务正在运行，将会进一步提示是否现在关闭 IIS 服务以配置 CSP 脚本驱动程序。



选择 “Yes”。安装程序会自动配置 CSP 到 IIS 中。

* 如果运行的是 Apache 服务器，安装程序也会检测到并自动配置。

* 安装完成之后，我们也可以再去修改 Web Server 的配置。



步骤 7：

安装过程顺利完成后，

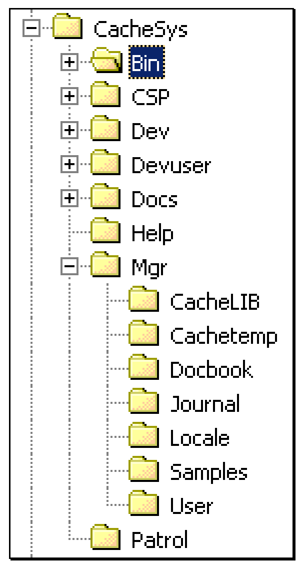
点击 按钮“完成”，退出安装程序。

2.3 安装过后

我们来看一下 Caché 安装之后是什么样子的

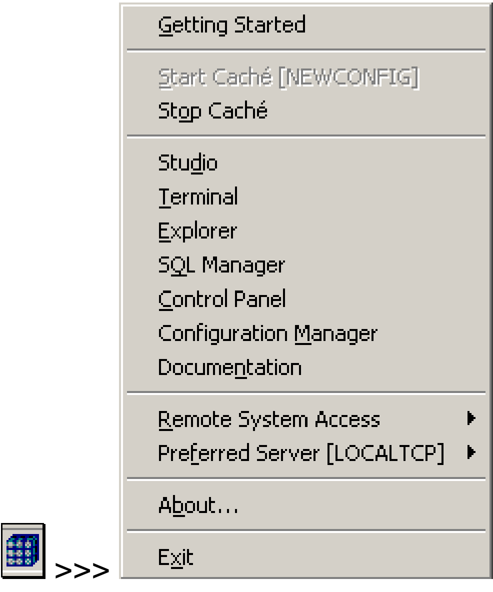
在 Caché 安装完毕后，在我们的计算机中多了如下项目：

a. 在文件系统中，新增了一个文件夹目录："c:\CacheSys" (该目录即为安装阶段默认的安装路径)。Caché 的系统文件和数据库文件等都位于这个目录中：



* 对于不同的安装配置，目录结构会略有不同。但主要的目录如\Bin、\CSP 和\Mgr 不会有变化。

b. 在 Windows 桌面右下方的任务栏中增加了一个立方体形状的图标，我们称之为 **Caché** 立方体(Caché Cube)。它是访问 Caché 的最常用的入口。点击 Caché 立方体，将会弹出一个菜单，通过此菜单我们可以选择各种工具来管理 Caché。



菜单中包含如下项目：

菜单项	中文名称	功能
Getting Started	入门文档	初学者可从中获取必要信息
Start Caché [xxx]	启动 Caché	启动 Caché 服务
Stop Caché	关闭 Caché	关闭 Caché 服务。点击后选择是要关闭还是要重新启动 Caché。
Studio	工作室	Caché 的集成开发环境。
Terminal	终端	通过类似于 DOS 的终端界面访问 Caché
Explorer	资源管理器	查看 Caché 中的数据资源，包括多维数组、类和例程
SQL Manager	SQL 管理器	通过关系型方式访问 Caché
Control Panel	控制面板	监视和管理 Caché 的运行

Configuration Manager	配置管理器	进行基本配置，包括数据库的配置
Documentation	联机文档	只有启动 Caché 后才可以阅读全部文档信息
Remote System Access >	远程系统访问	在本地控制远端的 Caché 服务器(任意平台)。UNIX 系统的管理要通过这种方式实现。子菜单中同样包含了工作室，终端，资源管理器，SQL 管理器，控制面板，配置管理器，联机文档等菜单项。
Preferred Server [xxx] >	当前使用的 Caché 服务器	子菜单中可以编辑 Caché 服务器列表。在这里配置远程服务器。我们可以在一台 Windows 平台的计算机上管理网络中所有的各种平台的 Caché 服务器。
About…	关于	查看版本
Exit	退出	关闭任务栏中的 Caché 立方体

* 对于这些菜单项的详细用法，我们将在后面章节中陆续介绍。

如果我们拥有许可证，可将许可证文件(CACHE.KEY)拷贝到目录\CacheSys\Mgr\ 下，再重新启动 Caché（在这里我们只为了简化步骤而选择重新启动；在真实的环境里，我们可通过别的步骤来更新许可证），许可证即可生效。

如果没有许可证，Caché 仍然是完整的，不过是按单机版状态运行。

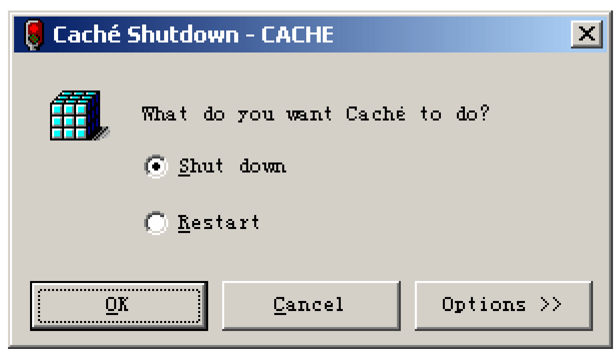
2.4　启动/停止 Caché 服务

Caché 在 Windows 中以 Windows 服务(Service)的形式存在，默认随 Windows 启动而启动。

现在 Caché 是启动的，我们来关闭 Caché：

选择 Caché 立方体->Stop Caché ，弹出窗口，提示：

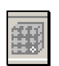
此时选择关闭(Shut down) Caché 或是重新启动(Restart) Caché。



选择后，按 “OK” 键，

关闭界面出现：

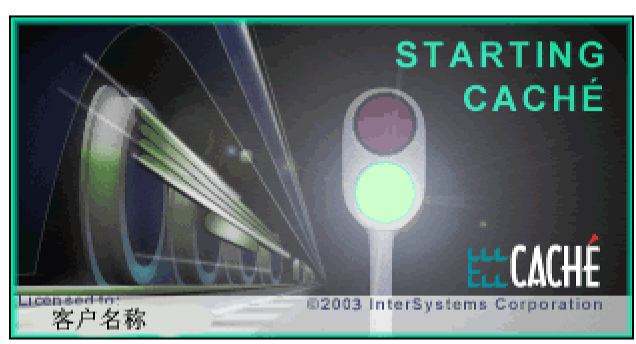
此界面将持续十几秒钟，消失时 Caché 便已关闭。

关闭后 Caché 立方体即变为灰色。

接着我们来启动 Caché：

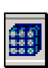
选择 Caché 立方体>Start Caché，

启动界面出现：(如果刚才选择了重新启动，则自动弹出此界面)



* 左下角将显示出许可证中的客户名称。

此界面将持续十几秒钟，消失时 Caché 即启动完毕。

启动后 Caché 立方体即变为蓝色。

现在我们已经安装了 Caché，并可以启动和关闭 Caché 服务。在下面单元中我们将对 Caché 进行基本的配置，为后面的开发工作做准备。

配置 Caché

Caché 安装完毕之后，我们首先会看到一个简单的案例，并为它在 Caché 中做一些基本配置，包括建立数据库等，为后面的开发作准备；

2.5 应用案例简介

整个服务器端的开发工作将围绕一个简单的应用案例进行。在后面的单元中我们将一步步地建立它。

本案例的模型为一个公司(Company)的框架，其中包含一些主要的类：

基类：Person 类。

嵌入类：Address 类，嵌入 Person 类当中。

继承类：Employee 类，Manager 类, 均继承于 Person 类。

这些类我们都将在 Caché 里创建，之后会产生实例并进行查询等操作。在此过程中，我们会逐渐接触到在 Caché 中进行服务器端开发的各种概念和技术。

下面我们首先为此案例在 Caché 中做基本的配置。

2.5.1 创建数据库

为了在磁盘中放置我们自己的应用案例，我们需要创建一个单独的数据库，名为“COMPANY”。

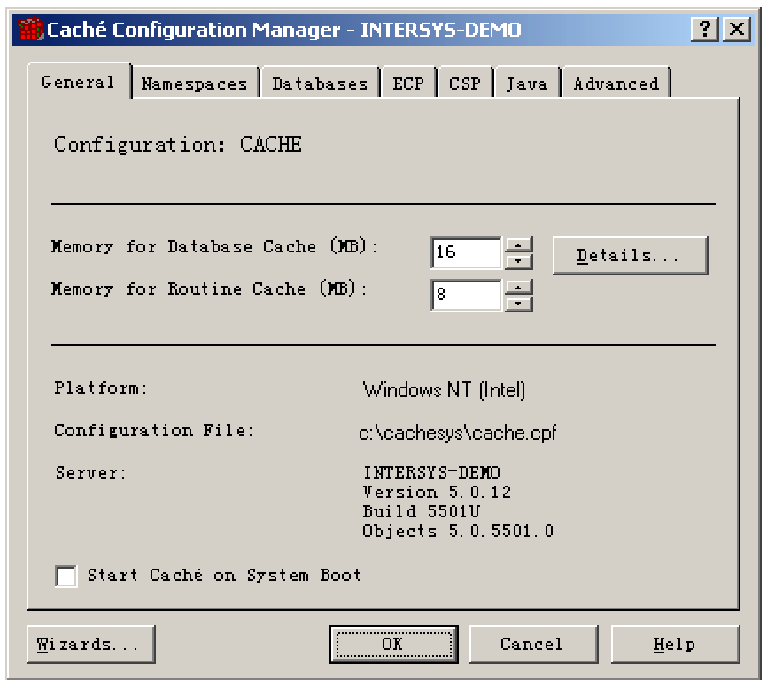
在 Caché 里，**数据库**(Database)指的是物理上存在于本地或远程计算机上的一个数据库文件，该文件的名称为 CACHE.DAT，所有数据和程序均保存在其中。

* Caché 可以有多个数据库，数据库文件名字均为 CACHE.DAT，只是存在于不同的目录中。例如，我们到文件目录 “CacheSys\Mgr\Samples\”下，可以看到一个 CACHE.DAT 文件，该文件保存着 Caché 的名为 Samples 的数据库里的所有数据。CACHE.DAT 文件最大可达 32TB（在不同的操作系统上，可能有区别）。

在新建数据库之前，我们要为它创建一个用来存放 CACHE.DAT 文件的目录：
在 Windows 中，新建一个目录：“c:\Company\” ，此目录将作为该案例的主目录。
(我们可以在认为合适的任意位置创建该目录)；

下面开始新建数据库：

在 Windows 中，选择 立方体->Configuration Manager ，打开 **配置管理器**(Configuration Manager)，以后我们可以通过配置管理器添加/删除数据库文件；



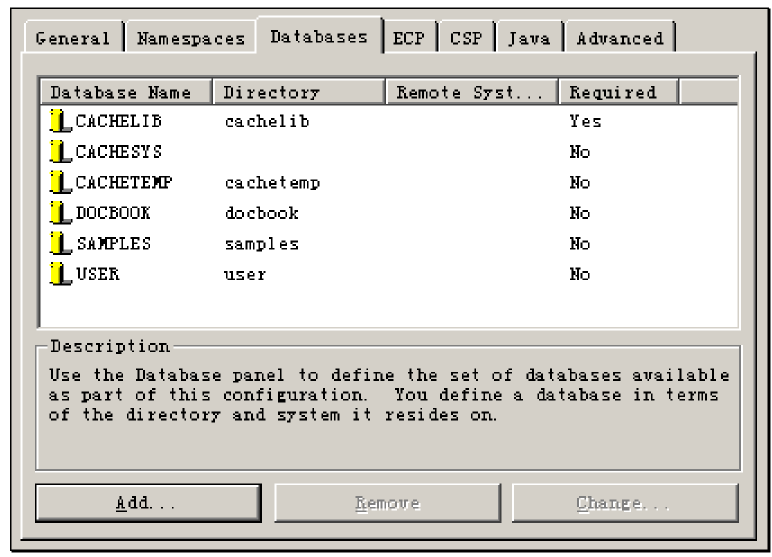
* 配置管理器是 Caché 基本的配置工具。Caché 的大多数配置都可以在此完成。

* 左下角的选项 “Start Caché on System Boot” 表示是否指定要在计算机系统启动时自动启动 Caché。

在配置管理器中，选择 “Databases” 标签，

进入 数据库(Databases)页面；

我们在数据库页面中可以看到系统中已配置的所有数据库文件：



可以看到，Caché 中已经存在了几个数据库，这是在安装时就已经自动默认配置好的。

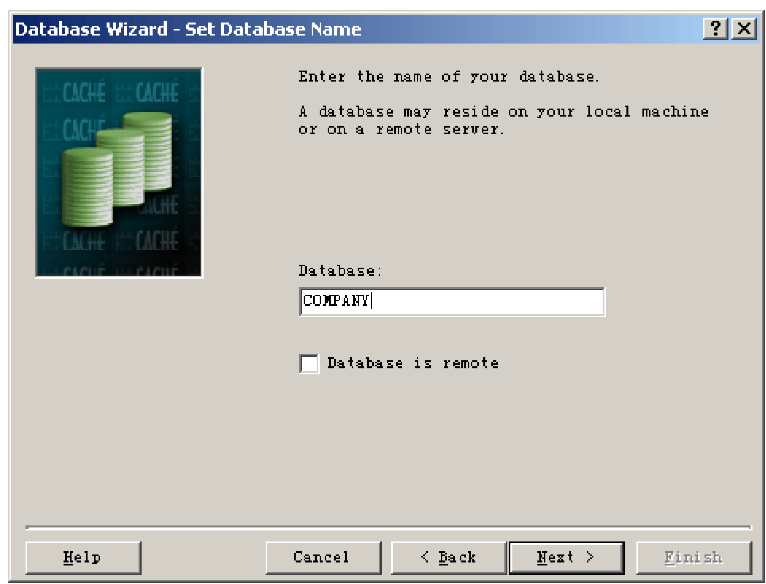
Caché 默认安装的数据库有以下几个：

名字	作用	文件所在目录
CACHELIB	系统类库	\Mgr\CacheLIB
CACHESYS	系统函数和工具	\Mgr
CACHETEMP	临时数据	\Mgr\Cachetemp
DOCBOOK	联机文档	\Mgr\Docbook
SAMPLES	所有示例	\Mgr\Samples
USER	用户自定义	\Mgr\User

下面我们将添加一个我们要用的名称为 COMPANY 的新数据库。

在数据库页面中，选择 “Add…” 按钮，

弹出 数据库向导(Database Wizard)：



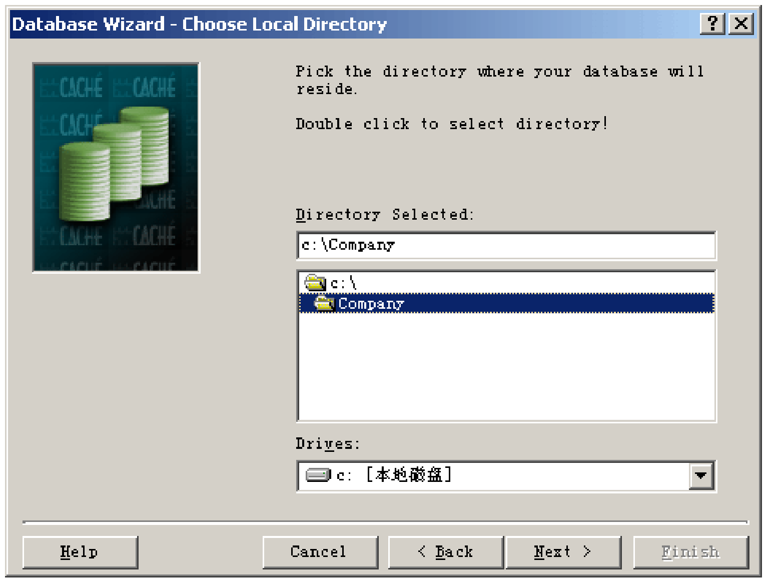
我们将通过数据库向导一步步地创建这个数据库。

步骤 1：

输入数据库名称：

我们在 Database 栏中 输入 数据库名称 “COMPANY”，

之后 点击 按钮 “Next>”；

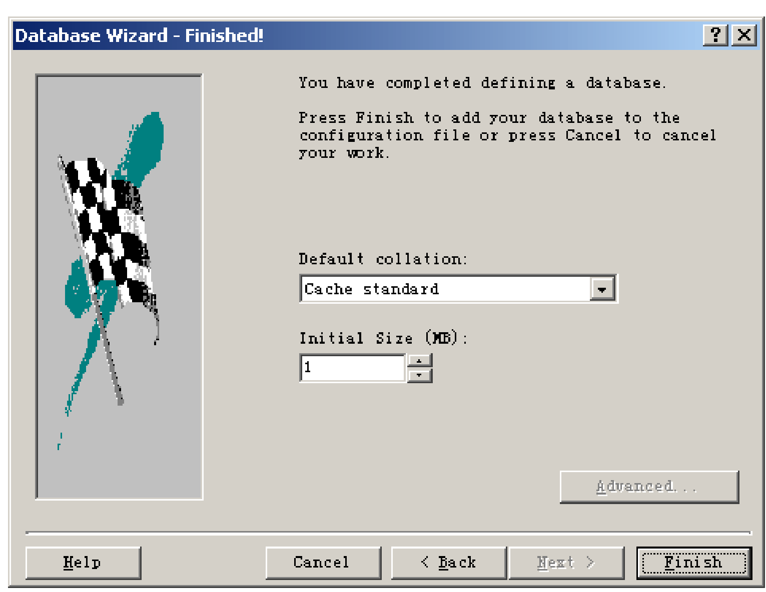


步骤 2：

选择数据库文件所在的目录：

在文件目录中选择文件夹 “c:\Company”，双击确认。

点击 按钮 “Next>”；

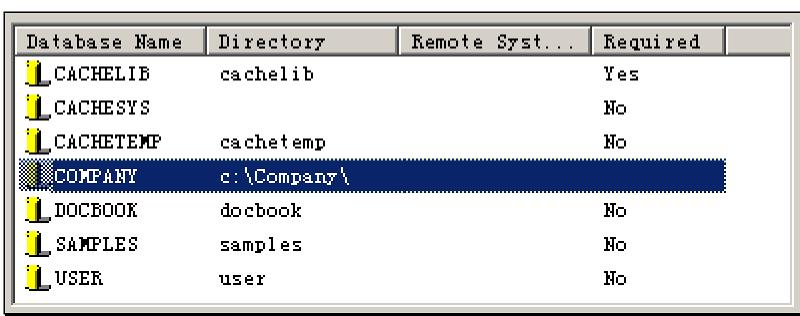


步骤 3：

使用默认值即可，点击 按钮 “Finish” 完成数据库的创建；

* 新建的数据库默认大小为 1MB，它的大小将会随数据量的增加而自动增加。

完成步骤 3 后，我们在数据库页面中将会看到新增的数据库 COMPANY；



Database Name	Directory	Remote Syst...	Required
CACHELIB	cachelib		Yes
CACHESYS			No
CACHETEMP	cachetemp		No
COMPANY	c:\Company\		
DOCBOOK	docbook		No
SAMPLES	samples		No
USER	user		No

接着，我们必须更新激活 Caché 以使刚才的操作生效：

具体做法是在配置管理器中，

点击 按钮 “OK” 确认，

此时，有对话框弹出询问：是否更新激活(Activate)，选择 “Activate” 按钮即可在不需要重新启动 Caché 的情况下完成新数据库的建立操作。

* 在配置管理器中所作的修改都需要最后按 OK 键确认，否则不会生效。按 OK 键之后，大部分操作均不需要重新启动 Caché (称为 Activate，像创建数据库这样的操作)，有少部分则需要重新启动(称为 Restart，如修订了缓存所用的内存量)。

此时我们也可以查看一下文件目录：c:\Company\ ，里面新增了数据库文件 CACHE.DAT。数据库已创建完成。

接下来，如果我们要真正能够在 Caché 程序中访问该数据库里的数据，就需要为它创建命名空间 (Namespace)。

创建命名空间

为了在我们的程序中能访问到数据库 COMPANY 中的数据，我们需要创建一个命名空间，指向数据库 COMPANY。

在 Caché 里，命名空间(Namespace),也称名字空间，是 Caché 中资源的逻辑表示方式。它是一个虚拟的、逻辑的工作空间。系统管理员可以在一个命名空间中定义不同的小组或个人所需的各项数据资源。

例如，财务部门需要使用某些已经存在于不同系统或不同目录中的数据，系统管理员就设置一个简单的引用在网上的所有数据库的命名空间。你的程序现在便可以根据名称来引用任何一部分 Caché 数据，而不需要同时指明数据的名称和位置。这就使得程序和系统两者都增加了灵活性。当要将数据移到一个不同的位置时，对于系统管理员来说全部需要做的事只是改动命名空间来包括更新的位置即可。

应用程序通过命名空间访问数据库里的数据和程序，因此，命名空间和数据库之间要建立映射。

* 命名空间和数据库之间的映射不一定是一对一的。一个数据库可以被多个命名空间访问；相反，一个命名空间可以访问多个数据库里的数据。命名空间也用来映射远程计算机中的数据库文件。

建立命名空间的主要工作就是建立与数据库的映射，这样做可以将程序逻辑与物理存在的数据独立开来，便于开发人员专注于系统功能的设计，不需要为未来实施时不同的系统架构而作出额外的工序，系统架构也因为这样变得更灵活。

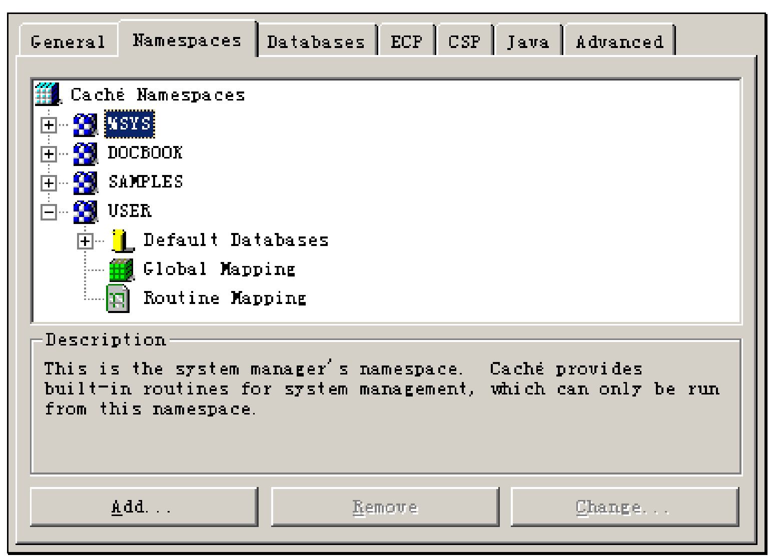
下面开始新建命名空间：

在 Windows 中，选择 立方体->Configuration Manager ，打开 配置管理器；

以后我们可以通过配置管理器添加/删除命名空间定义；

在配置管理器中，选择 Namespaces 标签，进入 命名空间(Namespaces)页面；

我们在命名空间页面中可以看到系统中所有的命名空间定义；



可以看到，同数据库一样，Caché 中已经存在了几个命名空间，这也是在安装时就已经配置好的。

Caché 默认安装的命名空间有以下几个：

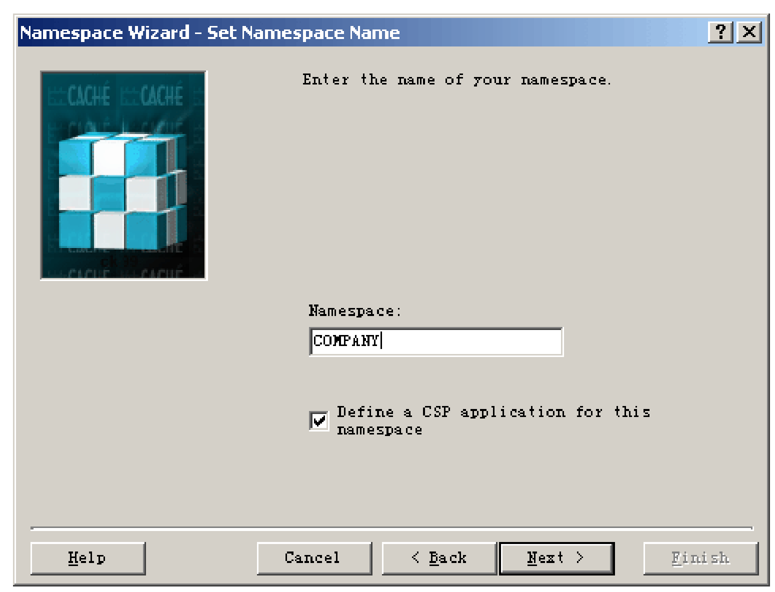
名字	作用	映射的主数据库
%SYS	系统用	CACHESYS
DOCBOOK	联机文档	DOCBOOK
SAMPLES	所有示例	SAMPLES
USER	用户自定义	USER

* 默认地，任何命名空间都会映射 CACHESYS 作为系统函数库，CACHELIB 作为系统类库，CACHETEMP 作为临时缓存库，以便此命名空间中的程序能够访问系统函数、类库以及临时缓存。

接著，我们添加一个新的命名空间。

在命名空间页面中，选择 “Add…” 按钮，

弹出命名空间向导(Namespace Wizard)；



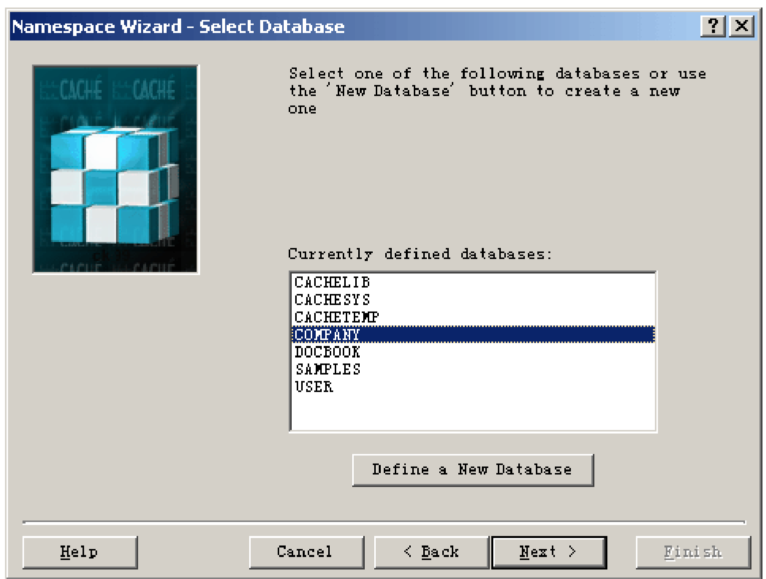
步骤 1：

输入命名空间名称：

在 Namespace 栏中输入命名空间名称 “COMPANY”，

* 命名空间和数据库名字可以相同，也可以不同

点击 按钮 “Next>”；



步骤 2：

选择所对应的数据库：

从列表框中选择 “COMPANY”，

点击 按钮 “Next>”；

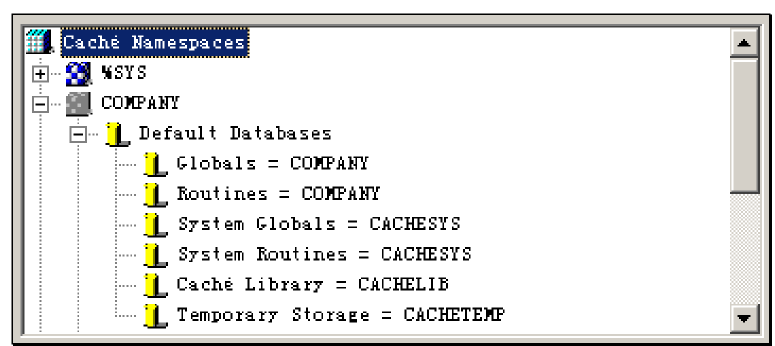
* 也可以在这里新建数据库。只要点击命名空间“Define a New Database” 按钮，就可以启动数据库向导。

步骤 3：

完成命名空间 COMPANY 的创建。

点击 按钮 “Finish>” 完成；

此时我们在命名空间页面中可以看到新增的命名空间 COMPANY，此命名空间对应数据库 COMPANY 里的数据。



* 可以看到命名空间 COMPANY 中的 Globals(数据)和 Routines(程序)都已指向数据库 COMPANY。

现在 COMPANY 的图标是灰色的，这表示它还未生效，因此我们必须更新激活 Caché 以使刚才的操作生效：

在配置管理器中，点击 按钮 “OK”；

弹出对话框询问是否更新激活(Activate)，选择 “Activate” 即可。

此时，命名空间创建完成。我们以后的开发工作将都会在 COMPANY 命名空间下进行。

经过以上工作，在 Caché 中的基本配置已完成。在下面单元中我们将开始开发，首先是创建第一个类。

2.5.2 创建类

我们将在 Caché 里创建第一个类：Person。

2.5.2.1 创建新类

Caché 中数据的定义可以表示为类或表，而且它们是同时存在的。我们以类的方式创建数据的定义和操作，因为它更符合我们的设计和开发习惯，功能也更强大。

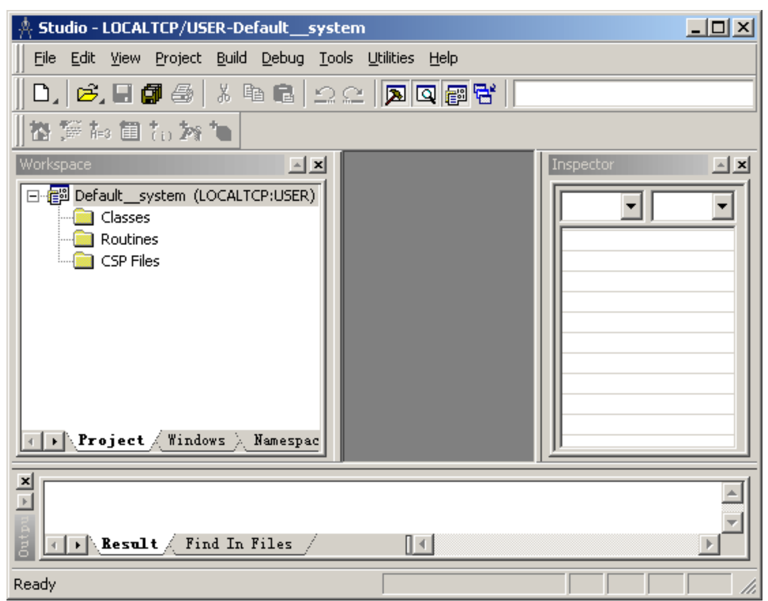
我们将要创建一个名为 Person 的类，根据设计，这个类有以下几个基本特点：

1. 基类。案例中的 Employee，Manager 类均为 Person 的子类；
2. 抽象类。Person 类只作为人的特征的抽象，作为基类，不应被实例化。
3. 持久性。Person 的子类不单能在内存中被实例化，还要拥有在磁盘中保存的能力，该能力应在 Person 类中定义，继承给子类。
4. XML 映射能力。可以将对象输出成 XML 格式，也可以输入 XML 文件。

1) 启动开发环境

要在 Caché 中创建类定义，首先要启动开发环境：

在 Windows 中，**选择** 立方体->Studio ，打开 **Caché** 工作室(Caché Studio)：



* Caché 工作室界面由以下部分组成：

在上方是菜单和工具按钮；

在左方 Workspace 中，可以看到当前命名空间下包含的类（Class）和程序（Routine）和 CSP 文件等；

在右方 Inspector 中，将显示类的各种属性和配置信息；

在下方 Output 中，将显示编译日志和搜索结果；

中间是文件编辑区，在这里编辑类文件。

Caché 工作室本身就是一个集成开发环境(IDE)，我们可以通过 Caché 工作室进行类的设计，或进行其它的开发。

Caché 工作室的标题栏中(“…LOCALTCP/USER…”)显示了它正工作在 LOCALTCP 服务器(本机)下的 USER 命名空间。

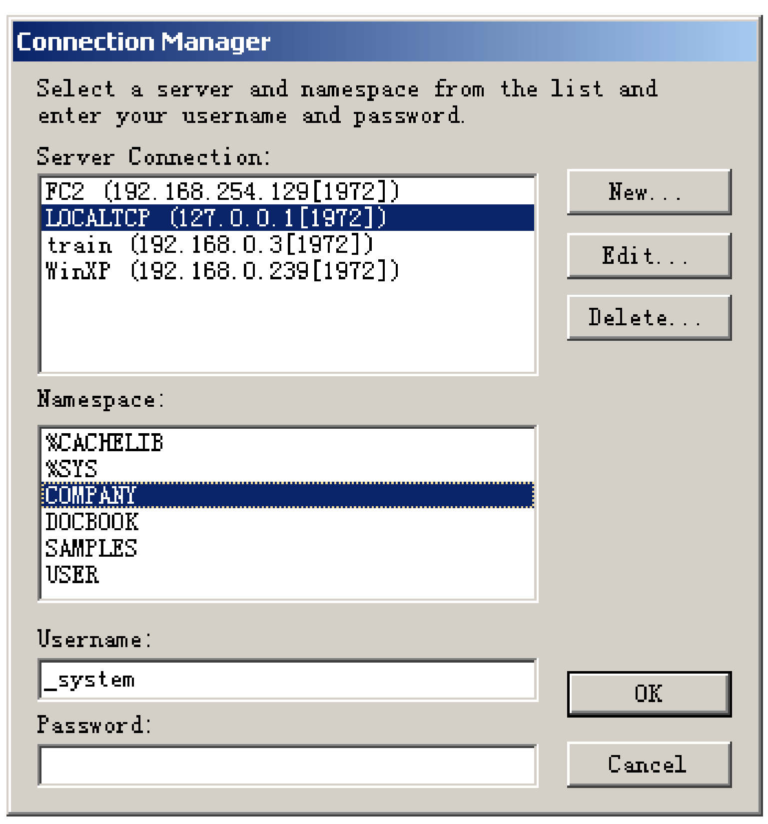
命名空间下面，我们要改变 Caché 工作室当前工作的命名空间为 COMPANY。

* 一个 Caché 工作室实例同时只能在一个命名空间下工作

在 Caché 工作室中，

选择菜单 File->Change Namespace ...，切换当前的命名空间；

此时弹出 连接管理器(Connection Manager)，我们可以选择命名空间：



* 在连接管理器中可以选择服务器和命名空间。可见，一台 Caché 工作室可以管理网络上的所有 Caché 实例。

在连接管理器中，

选择 Namespace 为 COMPANY，点击 按钮 “OK”；

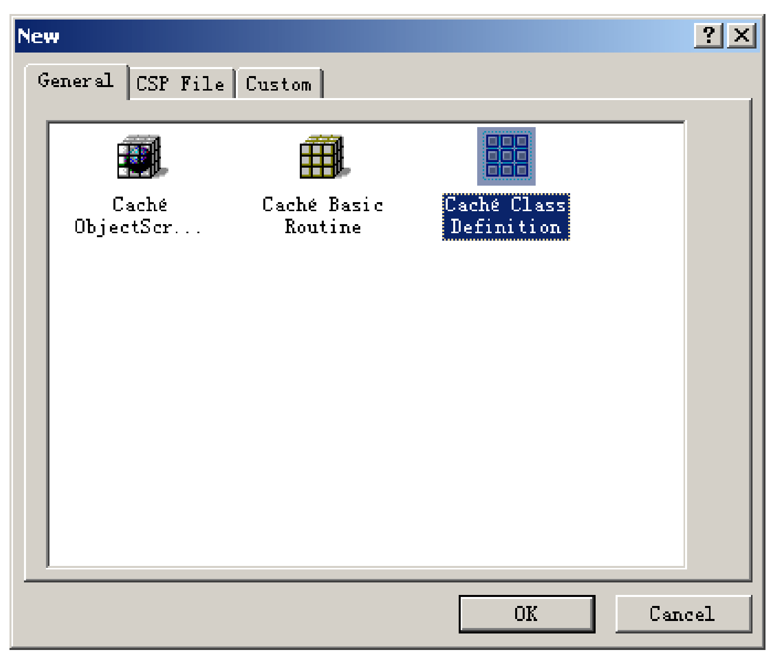
此时 Caché 工作室已连接到 COMPANY 命名空间。

2) 通过向导创建新类

下面，我们开始通过向导来创建新类：

在 Caché 工作室中，

选择菜单 File->New ，弹出 新建(New) 窗口；

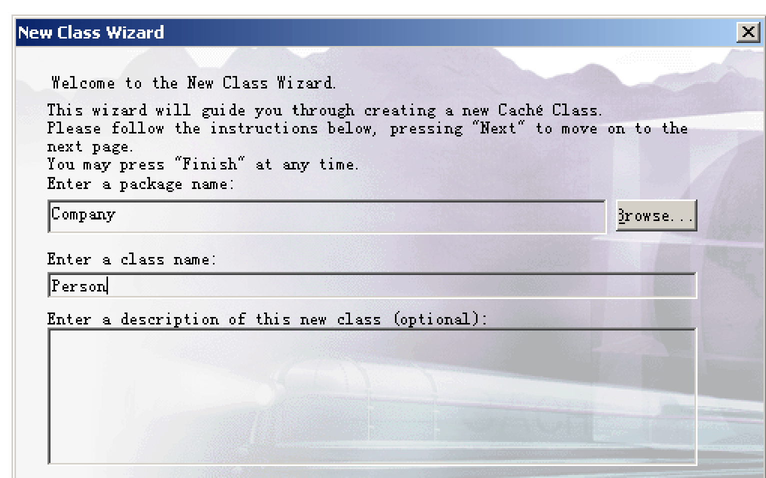


在新建窗口中，

选择 General 页面标签下的 Caché Class Definition (CDL，类定义文件)，

点击 按钮 “OK”；

弹出 新建类向导(New Class Wizard)：



我们将通过新建类向导一步步地建立类：

步骤 1：

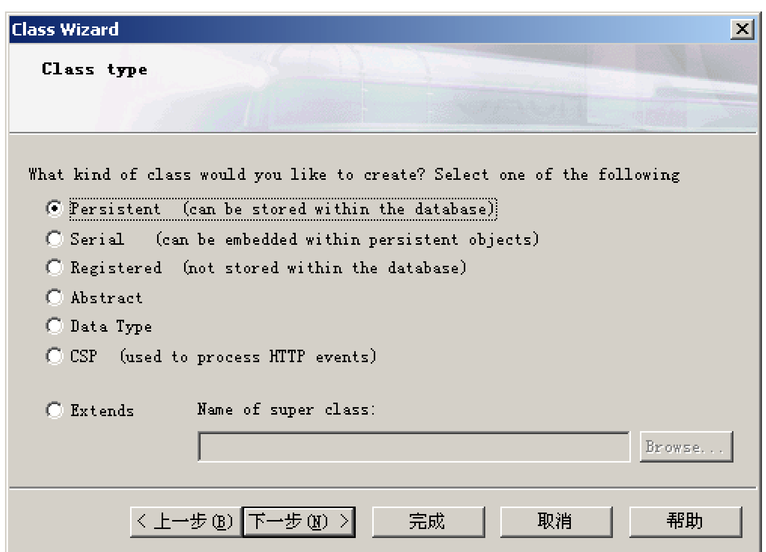
输入一个包(Package)和类(Class)名称：

包名输入 "Company"作为包的名字，

类名输入 "Person"作为类的名字，

点击 按钮 "下一步>"；

* 包的作用纯为逻辑上的，以便于管理各个类。另外，包名和类名中不能包含下划线 "_"



* 此界面在创建类定义时非常有用。我们可以在这里选择不同的类型：

a. Persistent：持久类，其实例可写到磁盘上保存；

b. Serial：可序列化的类(嵌入类)，可以被嵌入到其它类中；

c. Registered：内容不会写到磁盘上，实例只存在于内存中；

d. Abstract：抽象类，不创建实例的类；

e. Data Type：数据类型类；

f. CSP：Caché 服务器脚本语言类；

g. Extends：从某一类继承下来的类。

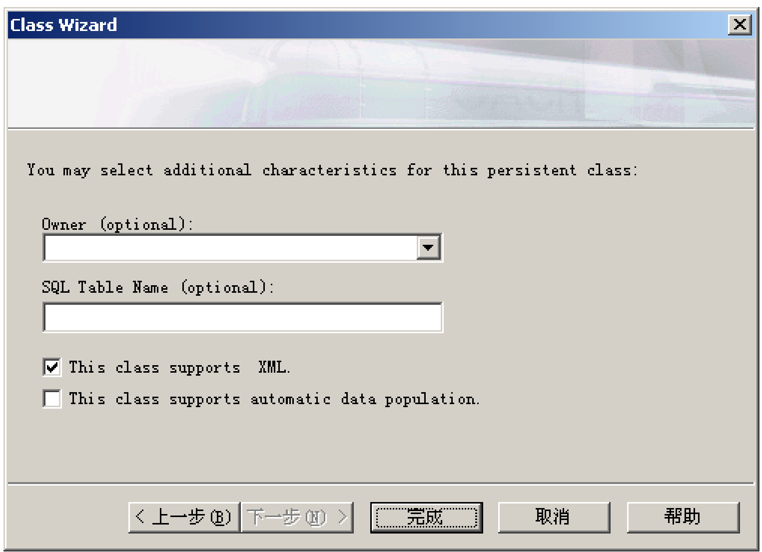
在后面我们将通过此界面创建不同类型的类。

步骤 2：

选择类的类型：

使用默认值 Persistent(持久类)，

点击 按钮 “下一步>”；



步骤 3：

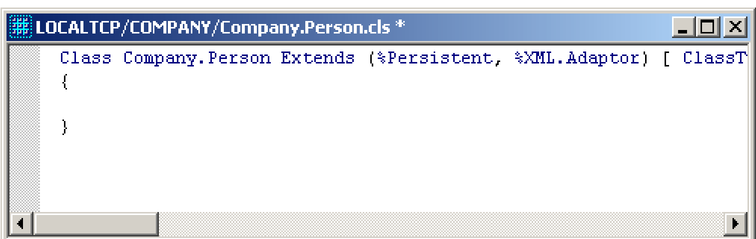
设置 XML 支持：

勾选上 “This class supports XML”，

点击“完成” 按钮；

新的 Person 类就已经被创建。

此时，在 Caché 工作室中增加了一个名为 Company.Person 的类，同时出现了其编辑器的界面：



新建的类文件内容为：

```
Class Company.Person Extends (%Persistent, %XML.Adaptor) [ ClassType = persistent, ProcedureBlock ]
{

}
```

从中我们可以看到 Caché 类文件头的格式为：

```
Class 包名.类名 Extends (父类 1, 父类 2…) [ 参数 1, 参数 2… ]
{
        其它定义…
}
```

其中，Class 关键字描述类的名称，Extends 关键字描述该类的父类。方括号中为类的参数。

Caché 是支持多重继承的，Person 类是从%Persistent (继承了持久性)和%XML.Adaptor(继承了 XML 连接器) 两个类继承而来。之后方括号里的参数是用来修饰这个类声明的。在这里，系统已为这个类预先设置了两个参数：ClassType 和 ProcedureBlock。

* 任何具有持久性的类都直接或间接地继承了%Persistent(全名为：%Library.Persistent)类，它提供了一些关于持久化的方法(如%OpenId, %Save 等)。其类名和方法名中开头的字符"%"说明了它是系统级的类和方法。关于%Persistent 类的详细信息可参看联机文档：http://127.0.0.1:1972/apps/documatic 中选择 %SYS 命名空间，%Library 包下的 Persistent 类。

* 注意：在 Caché 中，类名、属性名和方法名都区分大小写。

3) 添加 Abstract 参数

由于 Person 类是个抽象类，所以我们还需要为该类添加一个参数：Abstract。
在 编辑环境 中，
将类声明文件中第一行的方括号里的内容修改为：
… [Abstract, ClassType = persistent, ProcedureBlock]

此时，Person 类的框架已经创建，下一步我们要为它添加属性。

2.5.2.2 创建类属性

至此，新的 Person 类已被创建完成了，但它仍是一个空类。下面我们要为它添加属性。

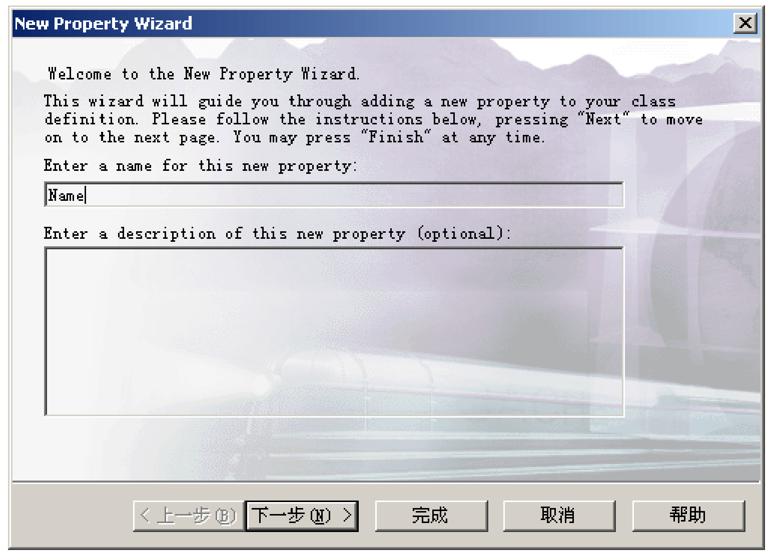
1) 利用向导创建类属性 Name

我们将要为 Person 类创建第一个名为 Name 的字符串型属性。

在 Caché 工作室中，

选择菜单 Class->Add->New Property，

弹出 新建属性向导(New Property Wizard) 窗口：



我们将通过新建属性向导一步步地创建属性：

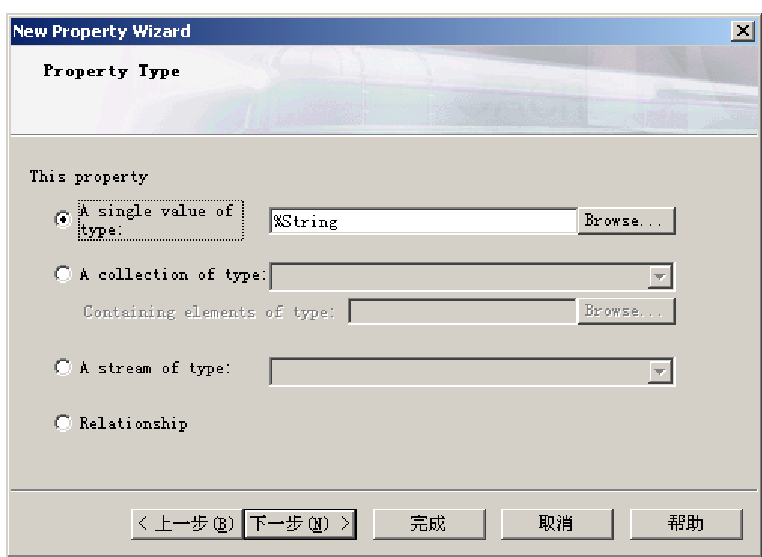
步骤 1：

输入属性的名称：

输入 “Name”，

点击 按钮 “下一步>”；

* 属性名不能包含下划线 ”_”。



* 此界面在创建类属性时是很常用的。我们可以在这里选择不同的属性类型：

a.单值类型：可以选择不同的类型(数据类型或类定义)；

b.集合类型：一组相同类型的集合，可以选择不同的基础类型和集合的方式(array 或 list 两种)；

c.流类型：可选择二进制流(图像或声音)和字符流(超长字符串)；

d.关系类型：可以同其它类建立相互引用的关系 (一对多和父子两种)；

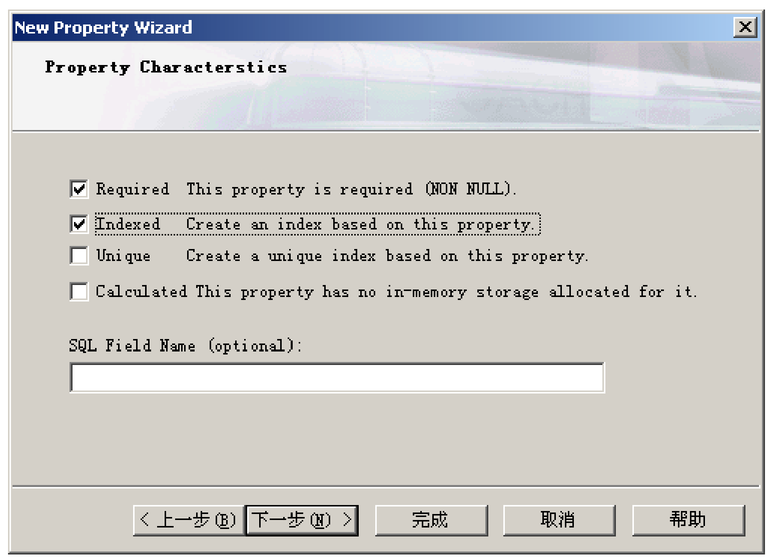
在后面我们将通过此界面创建不同类型的属性。

步骤 2：

选择属性的类型：

默认的即为字符串类型 “%String”，

直接点击 “下一步>” 按钮；



* 在这里可以为属性配置一些参数：

Required：表示该属性不能为空值；是必须有值输入的。

Indexed：表示要在该属性上创建索引(Index)，以提高查询的速度；

Unique：表示该属性的内容是唯一的，即不能有重复。并为之创建唯一索引；

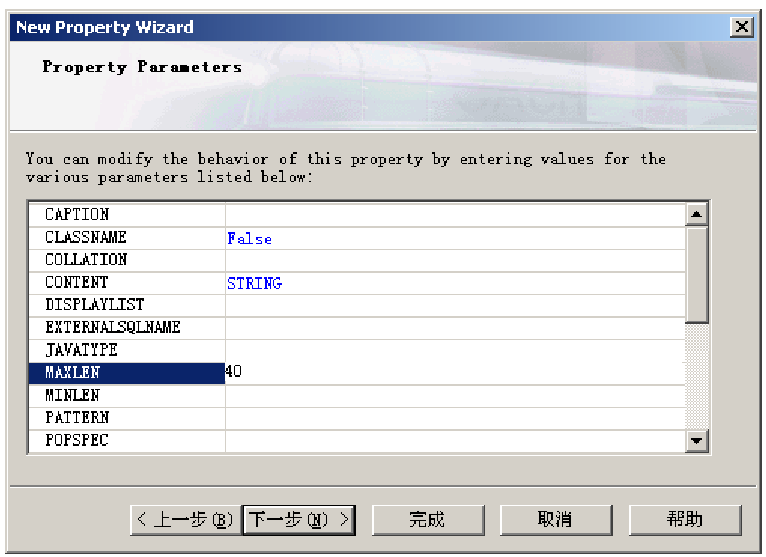
Calculated：表示该属性的值是由其它属性的值计算出来的。

步骤 3：

勾选上 “Required”，人的姓名不允许为空值 ,

勾选上 “Indexed”，为人名创建索引，

点击 按钮 “下一步>”；



*在这里可以为属性配置更详细的参数列表。这些列表内容也可以在 Caché 工作室中的 Inspector 中进行设置。

步骤 4：

将参数 “MAXLEN” 的值修改为 “40” , 即限制了该字串允许的最大长度。

再直接点击 “完成” 按钮；

属性 “Name” 及其索引 “NameIndex” 即被创建完成。

此时，在类定义文件中新增了如下信息：

```
Class Company.Person Extends (%Persistent, %XML.Adaptor) [ Abstract, ClassType = persistent, ProcedureBlock ]
{
Property Name As %String(MAXLEN = 40) [ Required ];
Index NameIndex On Name;
}
```

* 我们也可以完全不使用向导，直接在编辑器中输入上面的内容。

从中我们可以看到 Caché 类中属性声明的格式为：

Property 属性名 As 类型 (参数 1=值，参数 2=值…) [参数 1，参数 2…];

其中，Property 关键字描述属性名，As 关键字描述类型，后面的圆括号中描述了一部分的参数，方括号中描述了其它的参数。最后要以分号 “;” 结尾。

Name 属性在这里被定义描述为最大长度为 40 的非空字符串。

另外，Caché 类中索引声明的格式为：

Index 索引名 On 属性名 [参数 1，参数 2…];

其中，Index 关键字描述索引名，On 关键字描述属性名。方括号中为参数。最后要以分号 “;” 结尾。

* 我们可以根据程序的需求为属性创建不同类型的索引，以提高查询的性能。

2) 创建其它属性

我们不但需要创建字符串型的属性，还要创建其他类型的属性。这些属性和字符串属性根据需要的不同在类定义文件里的区别也只是需要将 “%String” 替换为相应的其他 Caché 数据类型。

以下是 Caché 里提供的一些数据类型：

描述	Caché 类型和参数	SQL 类型
变长字符串	%String	VARCHAR
定长字符串	%String (MAXLEN=n)	CHAR(n)
整数	%Integer	INTEGER
浮点数	%Float	FLOAT
日期	%Date	DATE
时间	%Time	TIME
二进制信息	%Stream [Collection = binarystream]	BLOB
长字符串	%Stream [Collection = charstream]	BLOB

布尔类型	%Boolean	

* 更多的类型信息请参看联机文档：http://127.0.0.1:1972/apps/documatic

下面，我们按照下表创建 Person 类的其它属性：

名称	说明	类型	特殊操作说明
Name	姓名	%String 非空	已完成
CardID	身份证号码	%String 非空	步骤 3：勾选 “Required” 步骤 4：设置参数 MAXLEN=18
DOB	生日	%Date 非空	步骤 2：类型框中输入：%Date 步骤 3：勾选 “Required”
Phone	电话号码	%String	步骤 4：设置参数 PATTERN = "3N1""-""8N"
Gender	性别	%String 非空	步骤 3：勾选 “Required” 步骤 4：设置参数 VALUELIST = ",男,女"
Spouse	配偶	Company.Person	步骤 2：类型框中输入：Company.Person
Picture	照片	%Stream 二进制	步骤 2：选择流属性(binary 类型)，如下图： A stream of type: binary
FavoriteColors	喜好颜色	%String 集合	步骤 2：选择集合属性(list 类型)，类型框中输入：%String。如下图： A collection of type: list Containing elements of type: %String Browse...

* PATTERN 可以定制参数字串的模式。Phone 属性字串的模式为：“3 个数字” 加 “-” 加 “8 个数字”，如 “010-58771912”。

* VALUELIST 定义了字串的可选值。如 Gender 属性的值可以为 “男” 或 “女”。

* Picture 为二进制的流

* FavoriteColors 为一组%String 属性的集合

完成后的 Person 类文件内容将如下所示：

```
Class Company.Person Extends (%Persistent, %XML.Adaptor) [ Abstract,
ClassType = persistent, ProcedureBlock ]
{
// 姓名
Property Name As %String(MAXLEN = 40) [ Required ];
// 姓名索引
Index NameIndex On Name;
// 身份证号码
Property CardID As %String(MAXLEN = 18) [ Required ];
// 生日
Property DOB As %Date [ Required ];
// 电话号码
Property Phone As %String (PATTERN = "3N1""-""8N");
// 性别
Property Gender As %String(VALUELIST = ",男,女") [ Required ];
// 配偶
Property Spouse As Company.Person;
// 照片
Property Picture As %Stream [ Collection = binarystream ];
// 喜好颜色
Property FavoriteColors As %String [ Collection = list ];
}
```

* 符号 “//”后面的内容为注释信息

Person 类的属性创建完毕之后，下一步我们将为该类添加所需的方法（Method）。

2.5.2.3 创建方法

类中不只是有数据定义，还有对数据的操作（在面向对象的领域里，这叫方法），下面我们将为 Person 类创建方法

在前面我们创建了生日属性 DOB，现在我们可以编写一个方法从这个属性得到这个人的年龄。

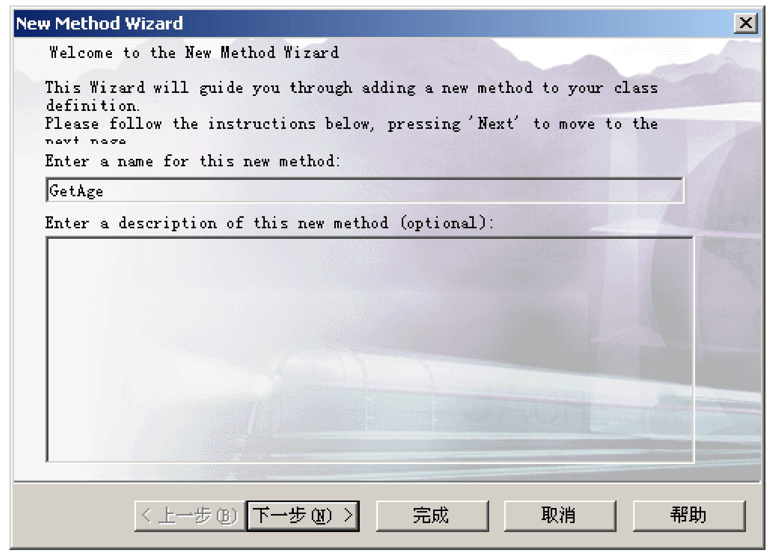
1) 用向导创建方法的框架

我们将要为 Person 类创建第一个方法，名为 GetAge()：

在 Caché 工作室中，

选择菜单 Class->Add->New Method ，

弹出 新建方法向导(New Method Wizard) 窗口：



我们将通过新建方法向导一步步创建方法：

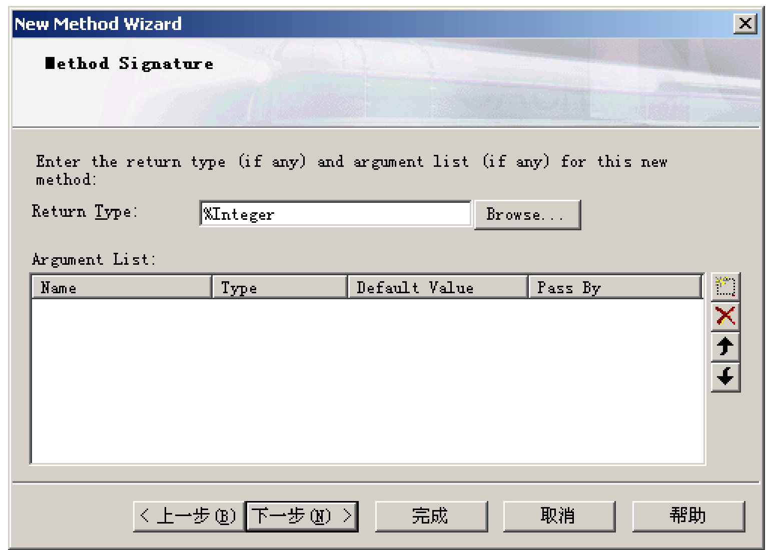
步骤 1：

输入方法的名称：

输入 “GetAge”，

* 方法名中不能包含下划线 “_” 或空格

点击 “下一步>” 按钮；

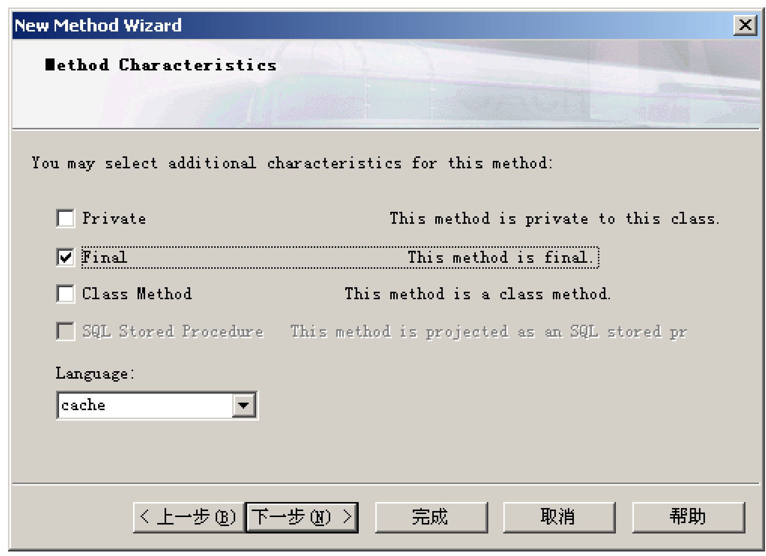


步骤 2：

选择方法的返回类型和参数定义：

将返回类型设为 "%Integer"，和这个方法没有参数定义，

点击 "下一步>" 按钮；



* 这里可以为方法配置一些附加的特性：

Private：表示该方法为私有；

Final：表示该方法不能被子类重载；

Class Method：类方法。表示该方法的调用不必依赖于对象实例，相当于静态方法；

SQL Stored Procedure：表示该方法可以作为存储过程，被关系型的访问方式所调用。

步骤 3：

设置方法的附加特性：

勾选 “Final”，说明该方法不能被子类重载，

直接点击 “完成” 按钮；

此时在类定义文件中就新增了一条方法的定义：

```
Method GetAge () As %Integer [ Final ]
{
}
```

从中我们可以看到 Caché 方法声明的格式为：

Method 方法名**(** 参数 1 **As** 类型 [= 默认值]，参数 2 **As** 类型 [= 默认值] …**) As** 返回类型 **[** 参数 1**,** 参数 2… **]**

{

　　实现代码…

}

其中，Method 关键字描述方法名，As 关键字描述参数和返回值的类型。可选地，通过 "=" 设置默认值。方括号中为参数。

如："Method Add(a As %Integer,b As %Integer=10) As %Integer […] {…}"

2) 添加方法的实现代码

将下列代码输入到方法 GetAge 的实现部分：

```
Method GetAge () As %Integer [ Final ]
{
 Quit ($Horolog-..DOB)\365.25 // 计算大概年龄
}
```

* 在 Caché 里每一行语句前面一定要至少空一格（在往后的章节里将会提到原因）。

* 以上语句将当前日期(从函数$Horolog 获得)与本人的出生日期(访问本对象的生日属性用"..DOB"形式)相减，再整除 365.25(考虑闰年因素，只计算大概的年龄值，整除用符号"\")，得到本人的年龄，并通过 "Quit "返回值。在这个语句的

最后是利用 // 符号隔开加上的备注，在此例中是“计算大概年龄”，只是为了备忘和便于读懂程序代码之用。

至此，即经过以上步骤后，，方法 GetAge()就已被创建完成。

现在，基类 Person 和其基本的属性和方法都已被创建完成。
接下来，要**保存**类定义文件 (选择菜单 File->Save)。
和**编译**此类定义文件 (选择菜单 Build->Compile)。

* 编译之后，我们可以在 工作室->Output 窗口 中看到编译器产生的信息。如果最后一行内容为 “ Compilation finished successfully.”，就表明已被编译成功。
* 我们可以在 工作室->Workspace 窗口->Project 标签->Classes 节点->Company 节点 中看到新生成的类，名称为 Person。

此时，类 Person 已经基本创建完成。在下面单元中我们将开始添加嵌入类。

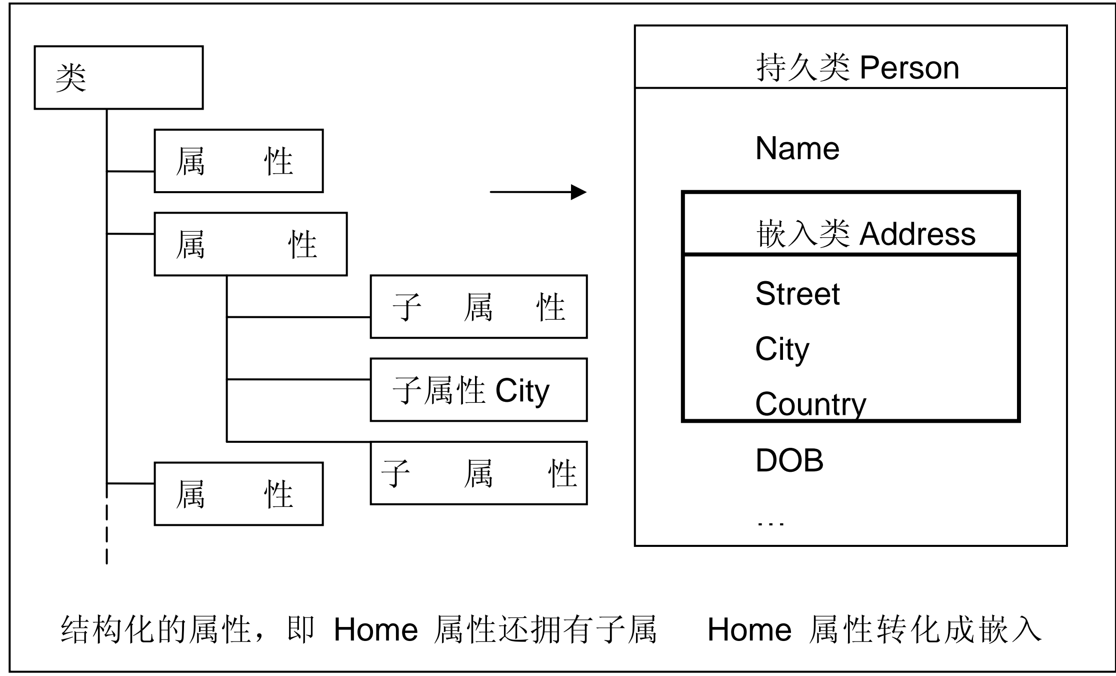
2.5.2.4 创建嵌入类

我们将建立另一个类 Address，他将被嵌入在 Person 类中，作为 Person 的一个属性，这样使得 Person 类的结构化更强。

新概念：%SerialObject 基类型，嵌入类；
新技术：创建嵌入类定义，将嵌入类嵌入到持久类中；

2.5.2.4.1 创建嵌入类定义

当持久类的属性具有结构化的特征时，我们可以为其创建嵌入类。



* 属性 Street，City 和 Country 都是 Person 的 Home 的一部分，因此可以将它们汇集起来组成新的类 Address。

嵌入类(Embedded Class)与持久类不同，持久类从基类型%Persistent 继承了持久性，而嵌入类则从另一基类型**%SerialObject** 继承嵌入类的特性。

* %SerialObject 类型具有序列化(Serialization)和反序列化(Unserialization)的能力。便于其主类进行管理。%SerialObject 类的详细信息可参看联机文档：http://127.0.0.1:1972/apps/documatic 中选择 %SYS 命名空间， %Library 包下的 SerialObject 类。

创建嵌入类的过程与创建持久类很相似，我们可以参照创建持久类的方法创建嵌入类。

1) 用向导创建嵌入类

在 Caché 工作室中，

选择菜单 File->New，

弹出 新建(New) 窗口；

在新建窗口中，

选择 General 标签下的 Caché Class Definition (类定义文件)，

点击 按钮 “OK”；

弹出 新建类向导(New Class Wizard)，

我们将通过新建类向导一步步创建类：

步骤 1：

输入包(Package)和类(Class)名称：

包名输入 “Company”，

类名输入 “Address”，

点击 “下一步>” 按钮；

What kind of class would you like to create? Select one of the following

- ○ Persistent (can be stored within the database)
- ◉ Serial (can be embedded within persistent objects)
- ○ Registered (not stored within the database)
- ○ Abstract
- ○ Data Type
- ○ CSP (used to process HTTP events)

步骤 2：

选择类的类型：

选择 “Serial”, 即可序列化的对象；

点击 “下一步>” 按钮；

- ☑ This class supports XML.
- ☐ This class supports automatic data population.

步骤 3：

选择 XML 支持：(Person 类支持 XML，同样 Address 也需要支持 XML)

勾选上 “This class supports XML”，

点击 “完成” 按钮；

此时在 Caché 工作室中增加了一个名为 Company.Address 的类，类文件内容为：

```
Class Company.Address Extends (%SerialObject, %XML.Adaptor)
[ ClassType = serial, ProcedureBlock ]
{
}
```

可以看到，Address 类是从%SerialObject(继承了嵌入类的特性)和%XML.Adaptor(继承了 XML 的映射能力)两个类继承而来。之后方括号里设置了两个参数：ClassType 和 ProcedureBlock。

2) 创建类属性

下面我们为 Address 类创建属性。嵌入类的类属性创建方法与持久类完全相同。具体方法可以参照 上一单元中的创建类属性内容。

我们按照下表创建 Address 类的属性：

名称	说明	类型
Country	国家	%String
City	城市	%String
Street	街	%String
Postalcode	邮编	%String

完成后的 Address 类文件内容如下：

```
Class Company.Address Extends (%SerialObject, %XML.Adaptor)
[ ClassType = serial, ProcedureBlock ]
{
// 国家
Property Country As %String;
// 城市
Property City As %String;
// 街
Property Street As %String;
// 邮编
```

```
Property Postalcode As %String;
}
```

至此，嵌入类 Address 和属性被创建完成。

接下来，要**保存**类定义文件　(选择菜单 File->Save)。

和**编译**此类定义文件 (选择菜单 Build->Compile)。

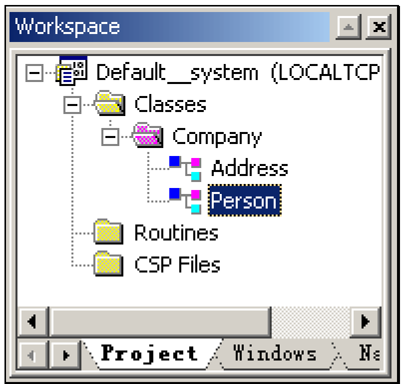
下面，我们将 Address 类嵌入到 Person 类中。

2.5.2.4.2 将嵌入类嵌入到持久类中

嵌入类将作为其主类(指包含嵌入类的持久类)的一个属性存在。嵌入类的实例由主类管理。

要嵌入一个嵌入类，我们只需在主类中声明一个新属性，将属性的类型定为嵌入类的类型即可。在这里，我们将为 Person 创建一个名为 Home 的属性，并将其类型设置为 Address。

首先，打开 Person 类的类文件：



打开 工作室->Workspace 窗口->Project 标签->Classes 节点->Company 节点，可以看到我们创建的类 Person 和 Address。

双击 Person 节点就可以打开其类文件。

将下面的代码添加到 Person 的类定义中：

```
// 家庭住址
Property Home As Company.Address;
```

此时，Address 类已经嵌入。

接着，保存 Person 的类定义文件 (选择菜单 File->Save)。

和编译 Person 的类定义文件 (选择菜单 Build->Compile)。

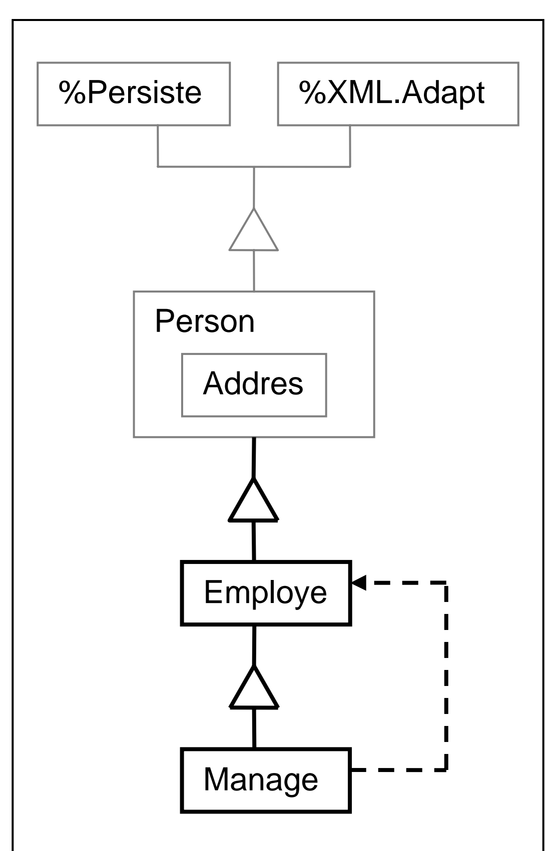
此时，嵌入类已经创建完成。在下面单元中我们将开始创建另一种继承类。

2.5.2.5　　创建继承类

继承是 Caché 的面向对象技术中最重要特征之一。下面我们要创建两个类 Employee 和 Manager，它们从 Person 类继承而来，并增加了一些自身特有的属性。

新概念：类的继承，关系；

新技术：创建继承类，创建关系；



* Employee 继承于 Person。而 Manager 继承于 Employee，并且可以“领导”多个 Employee 实例。

创建继承类的过程与创建持久类很相似，我们可以参照创建持久类的方法创建继承类。

1) 用向导创建新继承类

我们首先创建 Employee 类：

在 Caché 工作室中，

选择菜单 File->New ，

弹出 新建(New) 窗口；

在新建窗口中，选择 General 标签下的 Caché Class Definition (类定义文件)，

点击 “OK” 按钮；

弹出 新建类向导(New Class Wizard)；

我们将通过新建类向导一步步地创建该类：

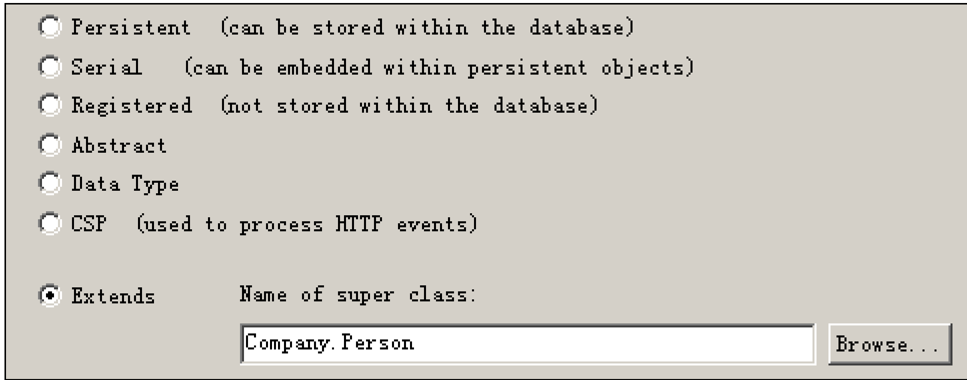
步骤 1：

输入包(Package)和类(Class)名称：

包名输入 “Company”，

类名输入 “Employee”，

点击 “下一步>” 按钮；



步骤 2：

选择类的类型：

选择 "Extends", 即继承类型；

输入 父类名称："Company.Person"；

点击"完成" 按钮；

此时在 Caché 工作室中增加了一个名为 Company.Employee 的类，类文件内容为：

```
Class Company.Employee Extends Company.Person [ ClassType = persistent, ProcedureBlock ]
{
}
```

可以看到，Employee 类是从 Company.Person 继承而来，同时也间接继承了%Persistent 类和%XML.Adaptor 类。

* 实际上，我们从类定义文件可以看到，持久类(如 Person)本身也是继承类，都是通过关键字 "Extends"继承了父类的特性。

下面我们将创建 Employee 类的子类：Manager 类。

可以完全仿照创建 Employee 类的方式去创建 Manager 类，只需改变以下几点：

步骤 1 中，类名输入：“Manager”；

步骤 2 中，父类名称输入：“Company.Employee”。

创建完成后，在 Caché 工作室中增加了一个名为 Company.Manager 的类，类文件内容为：

```
Class Company.Manager Extends Company. Employee [ ClassType = persistent, ProcedureBlock ]
{
}
```

2) 创建继承类特有的属性

下面我们为 Employee 类创建属性，属性的建立方法与持久类完全相同。具体可以参看第三单元中有关创建类的属性的内容。

我们按照下表创建 Employee 类的属性：

名称	说明	类型	特殊操作说明
Salary	薪水	%Integer	步骤 2：类型框中输入：%Integer
Title	头衔	%String	
Resume	照片	%Stream 二进制	步骤 2：选择流属性(character 类型)，如下图：
	A stream of type: character		

* Resume 为字符型的流，用来表示长字符串。

完成后的 Employee 类文件内容如下：

```
Class Company.Employee Extends Company.Person [ ClassType = persistent, ProcedureBlock ]
{
// 薪水
Property Salary As %Integer;
// 头衔
Property Title As %String;
// 简历
Property Resume As %Stream [ Collection = characterstream ];
}
```

3) 创建关系

下面，我们为类 Manager 和 Employee 创建一对多的关系。

关系**(Relationship)**是两个持久类之间相互的引用。在关系创建完成后，两个类都可以相互访问到对方的实例。一个 Manager 要领导多个 Employee，而它们又都是持久类，所以非常适合在这两个类之间建立关系。

* 关于关系的详细信息，请参看联机文档：
http://127.0.0.1:1972/csp/docbook/DocBook.UI.Page.cls?KEY=GOBJ_relationships

在 Caché 工作室中，

打开 类定义文件 Manager，

* 也可以在 Employee 中操作。因为关系是相互的，在哪一边创建都可以

选择 菜单 Class->Add->New Property ，

弹出 新建属性向导(New Property Wizard) 窗口；

我们将通过新建属性向导一步步创建关系：

步骤 1：

输入属性的名称：

输入 “Employees”，

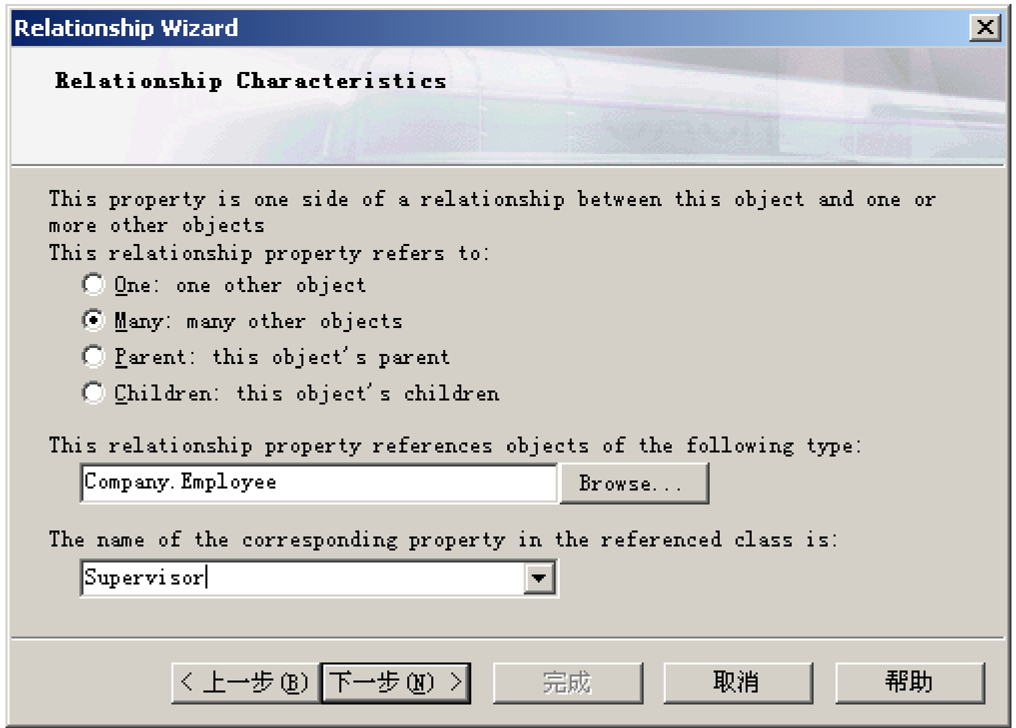
点击 “下一步>” 按钮；

步骤 2：

选择属性的类型：

选择 “Relationship”，即为关系类型；

点击 “下一步>” 按钮；



* 关系向导会自动在两个相关的类中创建关系。

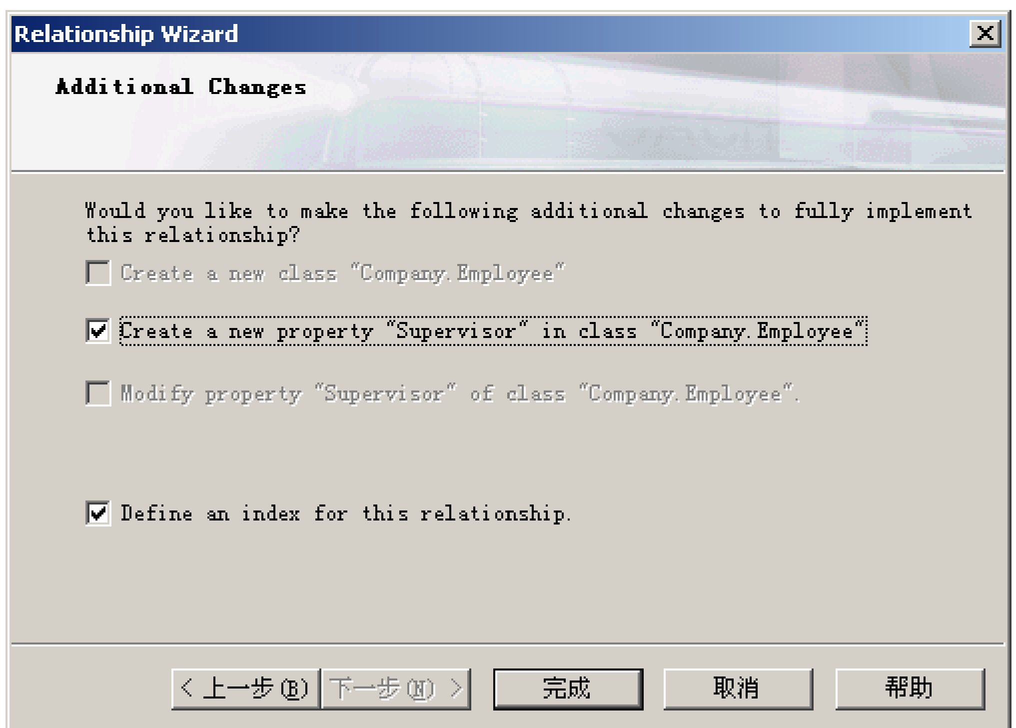
步骤 3：

选择 “Many”，即 Manager 的 Employees 属性代表一对多关系中的多的一边；

属性类型输入：“Company.Employee”，即关系中另一边的类型；

另一边的类对本类的引用名称输入："Supervisor"，这将会在另一边(Employee)的类定义中添加对 Manager 的引用，即 Employee 的 Supervisor 属性。

点击 "下一步>" 按钮；



* 此界面将询问和设置是否在关系的另一边中的已有 Company.Employee 类上添加新属性 "Supervisor 以及是否为此关系定义一个索引。在此例中它们是需要的，因此点击打上了钩表示需要添加。

步骤 3：

保持默认设置即可；

点击 "完成" 按钮。

完成后，Manager 类文件中增加了如下内容：

```
Class Company.Manager Extends Company.Employee [ ClassType = persistent, ProcedureBlock ]
{
```

```
Relationship Employees As Company.Employee [ Cardinality = many, Inverse = Supervisor ];
}
```

从中，我们可以看到 Caché 类定义中声明关系的格式为：

Relationship 属性名 As 属性类型 [Cardinality=基数类型，Inverse=相对属性名，其它参数 1…];

其中，Relationship 关键字描述属性名(对另一边的引用)，As 关键字描述属性类型(另一边的类型)。方括号中，Cardinality 参数表示另一边作为一对多(或父子)关系中的哪一边(many 或 one，父子关系中为 parent 和 children)，Inverse 参数表示另一边对本类的引用的名称。

在这里，Employees 属性作为多(many)的一边，因此 Cardinality 设为 many；在 Employee 类中也将会有 Supervisor 属性指向 Manager 类。

同时，Employee 类文件中增加了如下内容：

```
Class Company.Employee Extends Company.Person [ ClassType = persistent, ProcedureBlock ]
{
…
Relationship Supervisor As Company.Manager [ Cardinality = one, Inverse = Employees ];
Index SupervisorIndex On Supervisor;
}
```

在这里可以看到 Supervisor 属性，类型为 Manager。它与 Manager 类中的 Employees 属性之间是相互引用的关系。

另外，为了提高相互引用和查询时的性能，Caché 还为该关系建立了一个索引 SupervisorIndex。

此时，类 Employee 和 Manager 都已经创建完成。
接着，要**保存** 全部类定义文件 (选择菜单 File->Save All)。
和**编译** 全部类定义文件 (选择菜单 Build->Rebuild All)。

至此，继承类已经创建完成。在下面单元中我们将开始创建这些类的实例。

第三章 Caché ObjectScript 语言及其语法

1　引言

在这一部分中我们将介绍 Caché 为你提供的 Caché ObjectScript 语言，它是一种非常有用的对象脚本语言，针对 Caché 数据库应用的实际需要，重点增加了面向对象设计数据库的功能，不仅丰富了语言本身，而且极大便利了 Caché 数据库应用系统的设计。

目前，主流的编程语言都是以面向对象技术为基础的，当今数据库的发展也是以采用面向对象技术作为分水岭的。但是，过去我们常用的关系型数据库却都是基于关系型的思维建立数据库的。由于这样的情况造成了我们的设计者在用以往传统的关系数据库设计开发的过程中，需要首先试图把现实世界中的对象实体拆分成为若干个二维关系的表，并将程序中对象的属性与表里面的字段联系起来。这样的工作往往既困难又费时，占据了开发初期的 30%以上的时间，而且拆分的工作，会造成数据库也会造成相当一部分的磁盘空间的浪费。并且，在实施开发的过程中，也会因为原设计的不足而需要增加新的表或者字段，而最终造成更大的时间消耗和更多的空间浪费。关系数据库的缺点不仅使数据库的设计和优化复杂和难以驾驭。而且由于在数据库检索和输出时需要将所拆分的大量的表连接起来，这就必然会影响数据库应用系统的性能，使其难于实现用户所需的高速响应性能。甚至，在一些比较复杂的开发中，许多现实生活中的对象及其关系是根本不能很好用二维的关系表达或拆分的。

这样，我们迫切的需要一种新型的可以解决这个问题的数据库，而随着多媒体技术和 Internet 应用的迅猛发展，这种需求显得更为迫切，在适应当代的面向对象和面向 web 应用的新需求背景和推动下出现了 Caché。至此，我们可使用

新型的 Caché 数据库和它的 Caché ObjectScript 语言功能来开发出各种高性能、高伸缩性和高可用性的数据库应用系统，这种先进而成熟的技术使得我们的数据库的建立也实现了以面向对象技术为基础。

在使用 Caché 以前，您需要对 Caché 为你提供的数据库程序设计语言进行了解，那就是 Caché ObjectScript 语言。

Caché ObjectScript 对象脚本语言是一种以面向对象技术为基础的数据库库设计语言。Caché ObjectScript 的对象的建立和函数的编写，和目前通用的面向对象的编程语言是类似的。所以，熟悉面向对象编程的程序员使用 Caché 是很容易上手的。

2　Caché 对象及对象类型

2.1　Caché 对象的特性

2.1.1　继承

Caché 对象可以实现一重的继承和多重继承，也可以实现多层继承。

一重的继承：从一个父对象上继承。例如：B 可以继承于 A。

多重的继承：从多个父对象上继承。例如：B 可以继承于 A 和 C。

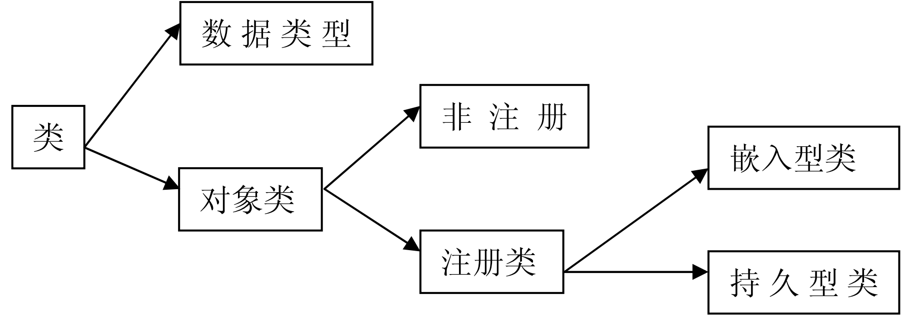
多层继承：B 可以继承于 A，C 可以继承于 B，D 可以再继承于 E，以此类推。

2.1.2　多态

一个函数方法可以以不同的形式存在于不同的类里面。也可以叫做重载。例如 B 和 C 都继承于 A。在 A 里面有一个函数叫 Count（）。在 B 和 C 里面都可以重新改写自己的 Count（）函数的代码，以达到自己的功能的实现。

2.2　Caché 对象的类型

类的分类情况和关系如下图。



2.2.1 数据类型类（Data Type Classes）

这种数据类型类是用作为对象类的属性（property）的定义的。它没有独立的身份，不能被实例化，也不能有任何的属性。
它提供一些特定的方法保证值的有效性和值之间的运算。

2.2.2 对象类

每个对象类都有独一无二的名字，对象类可以包含有自己属性、方法和关键字等等，可以被实例化。

2.2.3 非注册类

由于非注册类没有注册到 Caché 里面，它的 OIDs 和 OREFs 需要开发者自己提供和管理。同时，非注册类存在一些限制：

1) 系统不会为非注册类分配存储空间；
2) 不支持多态
3) 如果变量要引用非注册类，需要和相关的数据类型一起声明。

这里我们提到了 OID 和 OREF。OID（Object Identifier）是对象的 ID； 而 OREF（Object Reference）是已经放在内存中对象的实例。

2.2.4 注册类

注册类是完全从系统的%RegisteredObject 继承的，有完整的方法管理它们的存储。
注册型的类是只存在于内存中的。它的生成和管理都是 Caché 来管理的。它有 OREF，以用来在内存中访问。它也可以支持多态。

2.2.5 嵌入型类

嵌入型类是从系统的%SerialObject 继承的。它可以以独立的形式存在于内存中，但是它必须嵌入在别的对象中才能存在硬盘上。

2.2.6 持久型类

持久型类类是从%Persistent 继承迩来的。它可以独立地存在于内存和硬盘中。这种类也拥有独一无二的 OID，和可以包含自己的属性和方法。如果一个类的某一个属性是另一个类，那么这就称为是另一个类的引用。

2.3 Caché 类的元素

一个类的定义决定了这个类的类型和它的功能。这些定义通常包括了这个类的属性和方法。然而一个完整的类的定义，通常还包含有其它的一些元素：

- 一个独一无二的名字
- 类定义的名字推荐大写首字母，例如 Peron。还有如果一个类的名字是有意义的，可以把每一个有意义的部分使用大写的第一个字母，例如 HealthRecord。（并注意不要包含空格）
- 关键字
- 属性
- 方法
- 变量
- 查询
- 索引

2.3.1 Caché 类定义的关键字

一些关键字可以改变一个类的定义。而关键字主要是使用在利用 Caché 的类定义语言即 CDL（Caché Class Definition Language）开发一些类的时候。但是，您不需要记住这些关键字的名字。

除了类的定义有关键字以外，属性、方法、查询和索引都有关键字。

2.3.1.1 Caché 类的属性

2.3.1.1.1 属性表达出了一个对象的状态，也可以表达出一些对象之间的关系。

i. 属性可以是文字，数字，对象的引用，嵌入式对象，流（二进制流或者字符流），集合，多维属性或者是对象间的双向联系。
ii. 属性都有自己的一些方法来验证值的合法性以及保存这些值。
iii. 也可以对属性的格式进行控制
iv. 在访问到一个引用型或者嵌入式对象型的属性时，系统会自动把这个对象的内容读到内存中。

2.3.1.1.2 属性可以是 public 或者 private 的。Public 是指对这个属性的访问是不受限制的。而 private 是表示了这个属性只有类和它的子类内部可以访问。

2.3.1.1.3 属性都可以自动获得相关联的方法。它的这些方法要么是从系统提供的属性类里面继承得到，要么是从数据类型类继承得到。而所有的属性类都是系统类，用户是不可以更改它们的方法的。

2.3.1.2 数据类型

Caché 提供了许多的数据类型类，而 Caché 里面的每一个数据类型其实就是一个数据类型类。用户可以通过正确的定义数据类型类而定制自己的数据类型。

2.3.1.3 数据类型类提供的功能类似于 SQL 里面的表的字段。

2.3.1.3.1 数据类型类不同于其它的类在：

数据类型类不能形成实例；

数据类型类不能包含属性；

数据类型接口隐藏了数据类型类的方法。

2.3.1.3.2 基本的数据类型：

%Binary 二进制数据

%Boolean 布尔值

%Currency 货币

%Date 时间

%Float 浮点数

%Integer 整数

%List 列表

%Name 名字

%Numeric 实数

%Status 状态

%String 字符串

%Time 时间

%TimeStamp 日期+时间

2.3.1.3.3 数据格式和格式间的转化

i. 格式的种类

Display　用户界面上表示出的格式

Logical　在内存中存储的时候的格式

Storage　数据库里用来存储的格式

ODBC　用于 ODBC 和 SQL 访问的格式

ii. 格式转换的方法

DisplayToLogical()
LogicalToDisplay()
LogicalToOdbc()
OdbcToLogical()
LogicalToStorage()
StorageToLogical()

例如，如果一个类 Person 里的属性 DOB 是%Date 数据类型的。那么 Caché 会自动地产生 DOBDisplayToLogical()方法。

2.3.1.3.4 数据类型类的合法性判断方法

每个数据类型类除了含有格式转化函数以外，还有一个 IsValidDT()函数，专门检查它的有效性的。1 为有效。0 为无效。

2.3.1.3.5 用户数据类型，ODBC 类型，SQL 类型

用户类型指 JAVA 或者 ActiveX 的数据类型；
ODBC 类型表示在 ODBC 中使用的数据类型；
SQL 类型表示一些 SQL 工具所使用的数据类型；
下面是它们的比较：

Caché	用户类型	ODBC 类型	SQL 类型
%Binary	BINARY	BINARY	STRING
%Boolean	INTEGER	INTEGER	INTEGER
%Currency	CURRENCY	CURRENCY	CURRENCY
%Date	DATE	DATE	DATE
%Float	DOUBLE	DOUBLE	DOUBLE
%Integer	INTEGER	INTEGER	INTEGER
%List	LIST	VARCHAR	STRING
%Name	VARCHAR	VARCHAR	NAME
%Numeric	NUMERIC	NUMERIC	NUMERIC
%String	VARCHAR	VARCHAR	STRING
%Time	TIME	TIME	TIME
%TimeStamp	TIMESTAMP	TIMESTAMP	TIMESTAMP

2.3.1.3.6 枚举类型

枚举类型的变量有 VALUELIST 和 DISPLAYLIST，它们分别表示具体值列和显示值列。用户写的 LogicalToDisplay()、DisplayToLogical() 和 IsValidDT() 方法必须注意 VALUELIST 和 DISPLAYLIST 里面的具体值。

2.3.1.4 属性类型

2.3.1.4.1 属性可以是文字（包括数字），对象的引用，嵌入式对象，流（二进制流或者字符流），集合，多维属性或者是对象间的双向联系。

2.3.1.4.2 文字（包括数字）的属性类型

包括%Integer, %Date 和 %String。

例如：

```
Property Count As %Integer(MAXVAL = 100);
```

2.3.1.4.3 对象引用型的属性类型

一个属性可以是持久类（persistent），这样它就是一个对象的引用。

例如：

```
Property Manufacturer As User. Manufacturer（持久类）;
```

2.3.1.4.4 嵌入式对象型的属性类型

一个属性可以是嵌入式的类，这样它就是一个嵌入式对象型的属性。

例如：

```
Property Address As User.Address（serial 类）;
```

2.3.1.4.5 流（二进制流或者字符流）型的属性类型

Caché 的流属性有 CHARACTERSTREAM (包含字符流) and BINARYSTREAM (包含二进制流)两种。

2.3.1.4.6 集合型的属性类型

Caché 的集合型的属性有两种：数组型的和列表型的。前者通过关键字进行排序。后者按照一定顺序排列。集合里面可以包含文字、嵌入式对象、对象的引用。

类型的名称：

类型	数组型集合	列表型集合
文字	%ArrayOfDataTypes	%ListOfDataTypes
嵌入式对象	%ArrayOfObjects	%ListOfObjects
对象的引用	%ArrayOfObjects	%ListOfObjects

数组型集合的例子：

```
关键字  值
Hart  7/24/1960
Hoffmann  6/30/1963
Huber  8/7/1942
Meier  9/29/1959
Miller  2/13/1958
Smith  10/21/1958
Schulz  6/1/1950
...  ...
```

例如：

```
Property DOB As %Date [ Collection = array ];
Property Person As User.Person [ Collection = array ];
```

列表型集合的例子：

位置 值

1 Meier

2 Miller

3 Hoffmann

4 Smith

5 Hart

6 Schulz

... ...

n Huber

例如：

```
Property Name As %String [ Collection = list ];
Property Person As User.Person [ Collection = list ];
```

2.3.1.4.7 多维属性型的属性类型

一个属性可以是多维的，就象多维变量一样。在 Caché 中，凡是可以用在多维变量的方法就都可以应用在多维属性，例如$Order 等等。

例如定义了下面一个属性：

Property Children As %String [MultiDimensional];

那么它可以使用用在多维变量的方法，例如：

```
$Data(person.Children)
Set person.Children(2)="Sophie"
$Get(person.Children(1))
$Order(person.Children(""),-1)
Merge children=person.Children
Kill person.Children
```

但是由于 SQL 的表不能表示多维属性的字段，所以不能把它们存在一张 SQL 的表里面，也不能用表的方式来查看它们。

2.3.1.4.8 对象间的双向联系型的属性类型

在 Caché 中支持两种关系：一对多的关系（independent relationship）和父子关系（dependent relationship）。

2.3.1.4.9 属性的存储

不同的属性类型在 Caché 里面的存储情况是不同的，如下表所示：

属性类型	内存中	硬盘中
普通型	内部格式	保存格式
暂时型	内部格式	不存
计算型	不存	不存
多维型	多维数列	不存

暂时型（Transient）的属性，可以表示一个对象，但是一个变量则不行。这样在具体应用中，通常暂时型的属性可以用来表示一个持久型类对象的实例。

一个计算型（Calculated）的属性有 Get() 方法。用来定制得到它的值的规则。例如，年龄通常可以是这个属性类型，通过 AgeGet()得到。

2.3.1.5 Caché 类的方法

2.3.1.5.1 方法的参数和返回值

参数可以是值传递也可以是传递引用。

返回值可以是任何的类型。通常使用%Status 类型可以返回方法调用的成功与否的信息。

2.3.1.5.2 方法的可见性

方法可以是 public 或者 private 的。Public 是指这个属性的访问是不受限制的。而 private 是表示了这个属性只有类和它的子类内部可以访问。

2.3.1.5.3 类方法和实例方法

一般定义的”method”是指实例方法，ClassMethod 指的是类方法。

2.3.1.5.4 方法编写可以使用的语言：

- CachéObjectScript
- Caché Basic
- Java

2.3.1.6 Caché 类的参数

类的参数和类的属性不同的地方在于，类的参数主要是可以在类方法里面。而属性只在实例方法里面使用。

2.3.1.7 Caché 类的查询

查询提供了对对象实例的操作，通常查询可以被看作过滤器。

查询可以用 Caché ObjectScript、Caché Basic 或者 SQL 来编写。

查询的结果被作为结果集，它通过一个接口可以被 Caché ObjectScript、Caché Basic、ActiveX 或者 Java 所使用。

2.3.1.8 Caché 类的索引

所有的索引都是建立在一个或者多个属性的基础上的。下面列出了一些通用的索引的排序标准：

EXACT (完全按照分类的)

UPPER (字母大写化)

ALPHAUP (字母大写化并去掉符号)

SQLSTRING (去掉前面的空格，空的值为 SQL 的空串)

SQLUPPER (去掉前面的空格，字母大写化，空的值为 SQL 的空串)

STRING (去掉前面的空格，字母大写化并去掉符号，空的值为 SQL 的空串)

SPACE (强制按字符排序)

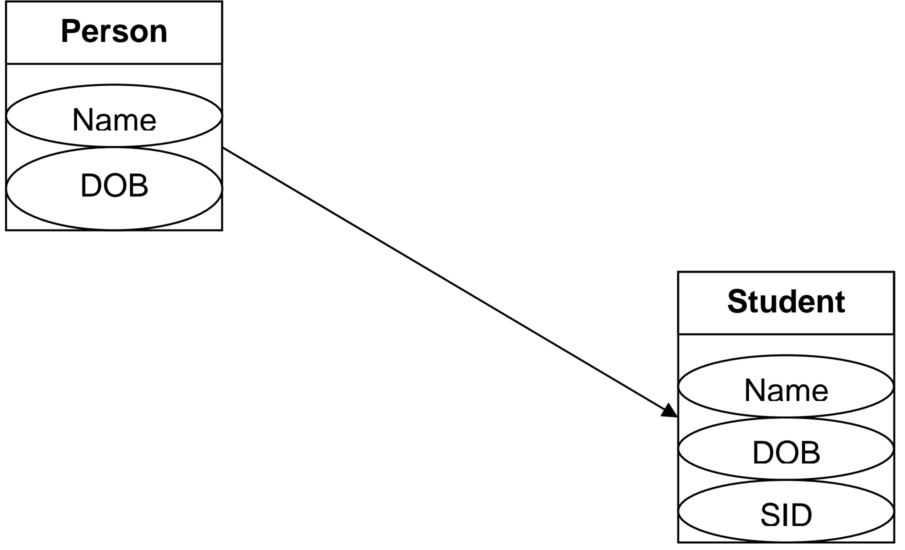
PLUS (强制按数字大小排序)

MINUS (数字的逆序排序)

2.3.1.9 Caché 类的继承

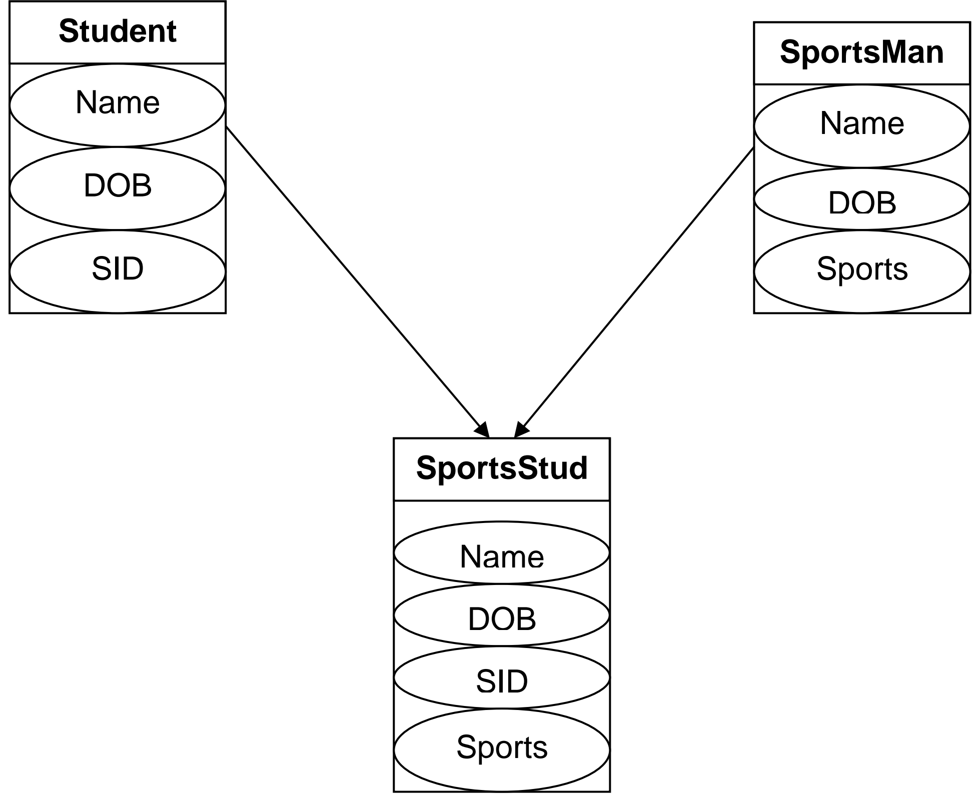
2.3.1.9.1 单一继承

单一继承就是从一个对象继承，关系如下图：



2.3.1.9.2 多重继承

多重继承就是从多个对象继承，关系如下图：



2.4 Caché ObjectScript 的基本结构

2.4.1 变量

按照存储的方式，可以分为：临时变量和持久变量。后者是以多维数组的形式存在于硬盘的存储中的。

按照访问的方式分，变量有两种：本地变量（Local Variable）和全局变量（Global Variable，即 Globals）。虽然 Globals 可以叫做全局变量，但是它在 Caché 中有不同于其它面向对象的编程语言中的全局变量的特点。在 Caché 中，我们可以直接用^Globals 的方式来访问它。而本地变量 Local Variable 是仅可以由本进程对其进行访问的数据。

2.4.1.1 变量的名称

Caché ObjectScript 不限定变量的长度。然而只有前 31 个字符是重要的，如果前 31 个字符一样的变量，是被看作为一个的。变量的名称必须是以大写或者小写的字符或者是用%符来开头。

2.4.1.2 变量的定义

Caché ObjectScript 的变量的定义一共有 3 个函数：Set、Read 和 For。Kill 命令是用来删除变量的。

例如：

```
SET ARTICLE1="Trousers"
```

这是一个直接赋值语句，它把 Article1 的值定义为"Trousers"。Caché ObjectScript 的变量的使用是不需要象 C 里面那样声明的。

Read "Article - ",Article2 语句指的是：

在屏幕上显示出 Article-字样时，用键盘输入一个值，即读一个值到变量 Article2 中。

Kill Article1,Article2 语句的含义是：

删除 Article1 和 Article2 这两个变量。

注意：kill 的使用需要谨慎，在 terminal 终端环境中如果只输入一个 kill 然后回车的话，会把所有的 global 都删除的。

2.4.1.3 一些特殊的变量

这些特殊的变量有指定的含义，它们一般以$开头。有个别特殊变量也同时支持缩写（注意下划线）

下面介绍一些常见的特殊变量：

$HALT

如果$HALT 指定出一个 routine 即程序模块的名字，当语句被执行的时候这个 routine 将被调用。

例如：

```
SET $HALT="MyTrap^CleanupRoutine"
WRITE !,"the halt trap is: ",$HALT
```

$HOROLOG

$Horolog 包含了时间和日期，它们由逗号分开。第一部分是日期的数字，它是从 1840 年 12 月 31 日开始计算日期的数字。第二部分是秒的数字，它是从午夜开始计算的。

例如：

```
>WRITE $HOROLOG
57713,36814
>WRITE $PIECE($H,",",2)
```

```
36815
>WRITE +$H
57713
```

$IO

$IO 表示正在使用的输入输出设备的名称。

例如：

```
>USE "TTA3:" WRITE $IO
TTA3:
```

$JOB

$JOB 包含一个正整数，表示系统所分配给每一个在执行中的进程独一无二的编号。当进程还在的时候，$JOB 的值是不变的。

例如：

```
>WRITE $JOB
1024
```

$KEY

$KEY 包含了上一个由 read 命令读的最后一个字符。

例如：

```
>WRITE $KEY
<Return>
```

$PRINCIPAL

和$IO 类似，$PRINCIPAL 包含了当前进程开始采用的设备的名称。

例如：

```
>WRITE $PRINCIPAL
/dev/tty05
```

$QUIT

$QUIT 在调用用户定义的函数中的值是 1，在其它的时候值为 0。

例如：

```
>WRITE $QUIT
0
```

$STORAGE

$STORAGE 以字节为单位表示空间的大小。

例如：

```
>WRITE $STORAGE
15845
```

$TEST

$TEST 用作测试某一个表达式的真假。它和"IF "搭配使用，也可以和 OPEN、LOCK、 READ 和 JOB 一起使用。

例如：

```
>IF A=5 WRITE $TEST
1 (如果 A=5 时)
>OPEN DEV::10 WRITE $TEST
1 (如果 OPEN 命令在 10 秒以内曾被成功执行时)
```

$X

$X 光标在输出设备的水平坐标的位置 。

例如：

```
>IF $X>79 WRITE !
```

$Y

$Y 光标在输出设备的垂直坐标的位置。

例如：

```
>IF $Y>59 WRITE #
```

$ZA

表示在当前的设备上的上一个 READ 命令的状态。

例如：

```
>WRITE $ZA#2
1
```

$ZB

$ZB 和$KEY 一样。

例如：

```
>WRITE $ZB
<Return>
```

$ZCHILD

用 JOB 命令执行的上一个进程的进程 ID。如果值是 0，那么就没有进程被执行。

例如：

```
>WRITE $ZCHILD
37
```

$ZEOF

$ZEOF 表示是不是已经到了文件的结尾了。如果值是-1 的话，就是已经到了结尾了。

例如：

```
>IF $ZEOF CLOSE FILE
```

$ZHOROLOG

$ZHOROLOG 返回的值包含了 Caché 开始以后计时的秒数和毫秒数两个部分。

例如：

```
>WRITE $ZHOROLOG
2365.632
```

$ZIO

$ZIO 表示现在的终端设备的连接类型。

例如：

```
>Write $ZIO
192.9.200.79/1260
```

$ZJOB

$ZJOB 以二进制数的形式表示当前的 JOB 的信息。

例如：

```
>WRITE $ZJOB
5
```

$ZMODE

$ZMODE 包含了刚被 OPEN 或者 USE 访问的设备变量的值。

例如：

```
>WRITE $ZMODE
RY\ISM\
```

$ZNAME

$ZNAME 表示当前被读出来的 ROUTINE 的名字。

```
>WRITE $ZNAME
Rec112
```

$ZNSPACE

$ZNSPACE 表示当前命名空间的名字。它可以用来切换命名空间，效果和 ZNSPACE 命令是一样的。

例如：

```
>WRITE $ZNSPACE
USER
```

$ZPARENT

$ZPARENT 包含正在运行的由 JOB 创建的进程 ID。如果值是 0 ，则没有这样的进程。

例如：

```
>WRITE $ZPARENT
0
```

$ZPI

$ZPI 包含 PI 的值：3.141592653589...

例如：

```
>Write $ZPI
3.141592653589793238
```

$ZREFERENCE

$ZREFERENCE 提供了一个对最近常访问的 GLOBAL 的引用。

例如：

```
>SET ^G(1)="HELLO"
>WRITE $ZREFERENCE
^G(1)
```

$ZSTORAGE

$ZSTORAGE 包含了可以给 Caché 进程使用的空间的大小（KB 为单位）。

例如：

```
>WRITE $ZSTORAGE
24
```

$ZTIMESTAMP

和$HOROLOG 类似，$ZTIMESTAMP 包含了时间的日期、时间、和毫秒数。

例如：

```
>WRITE $ZTS
57500,11608.52
```

$ZTIMEZONE

$ZTIMEZONE 包含了 GMT（格林威治平均时区）往西的相差时间的分钟数。

例如：

```
Boston 是+300，Berlin 是 -60。
>Write $ZTZ
300
```

$ZVERSION

$ZVERSION 返回的是当前 Caché 的版本号

例如：

```
>WRITE $ZVERSION
Cache for Windows NT (Intel) 5.0
```

2.4.2 运算符和表达式

Caché ObjectScript 提供的运算符有两种：一种是单元的运算符，另一种是二元的运算符。

二元的运算符包括了：算术运算符；字符串运算符；逻辑运算符；比较运算符。

2.4.2.1 基本的运算符

i. 单元算术运算符+、和－。这两个运算符在使用的过程中会强制的做类型转化的。、

例如下述转换：

```
字符串　值
________________
“--13”    13
“+12ab”   12
“-0”      0
```

ii. 二元算术运算符+、-、*、/、\、#，**。

其中，+、一、*、/ 是基本的 4 个算术运算符。

\ 整数的除法

** 幂函数

取模函数，a#b = a-(b*[a/b])

例如：

```
假定 a=5, b=7, c=10 ，下面就是这些运算的结果：
a+b = 12
a*b-c = 25
c/a = 2
2+a*b = 49
2+(a*b) = 37
3700#3600 = 100
c#a = 0
9.123\1 = 9
3700\3600 = 1
2**5 = 32
9**.5 =  3
16**.25 = 2
4**(-2) =0 .0625
```

iii. 算术比较运算符<、>

```
假定 y=12, z=15，则：
y<z 值为 1 （真，即 true）
z>"16abc" 值为 0 （伪，即 Fulse）
```

iv. 字符串比较运算符=、[、]、]]

= 比较运算符是比较两个字符串的相同与否。如果需要比较两个数字是否一样的话，需要在前面再加一个 + 号。例如，+number1=+number2。

[运算符检查右边的串是不是包含了左边的串；

] 运算符检查左边的串是不是在字典顺序上跟着右边的串；它是严格按照 ASCII 码来操作的。例如：2]19，返回是 1，因为 2 的 ASCII 值是 50，而 1 的 ASCII 值是 49。

]] 运算符检查左边的串是不是在顺序上跟着右边的串，它是按照复合的标准判断的。例如：2]19，返回是 0，因为按照数字的排序 2 不在 19 的后面。返回的判断值以 1 表示为真即成立，0 表示为伪即不成立。

例如：

```
如果 x=3、y="A"、 z="3.0" ，则：
x="a" = 0
x=z = 0
"ABC"[y = 1
x]y = 0
y]x =  1
z](x=y) = 1
2]]19 = 0
```

v. 逻辑运算符 &(与)、!（或）、&&（与）、||(或)

后面的&&运算符与前面的&运算符是有区别的，&&运算符当判断出结果的时候将不再计算所有的表达式。例如，左边的第一个表达式为 FALSE 的时候，&&将不再计算其它的表达式的值了，直接把结果定为 FALSE。

vi. 连接符_

它是用来连接两个字符串的

vii. 格式判断运算符 ?

? 运算符是专门用来检测是不是满足一个指定格式的。

例如：257？3N，这个是满足的。N 的含义是数字。下面是一个比照表：

```
－－－－－－－－－－－－－－－－
字符 含义    ASCII 码
－－－－－－－－－－－－－－－－
C  控制符    0-31, 127
P  标点符号   32-47, 58-64
N  数字     48-57
U  大写字母   65-90
L  小写字母   92-122
A  文字字符   65-90, 92-122
E  所有的字符  0-127
```

比如判断一个日期 date 是不是 mm/dd/yyyy 格式的，可以用 date?2N1P2N1P4N 来判断。

2.4.2.2 基本的表达式

Caché ObjectScript 里面的表达式是由一些表达式的元素和一些运算符连接构成的。表达式的元素可以是：本地或者全局变量、特殊变量、函数、用户定义的函数、用户定义的变量、数字、文字、系统变量、在括号中的表达式。

2.4.3 Caché ObjectScript 的语句命令与函数

2.4.3.1 命令的格式

i. 普通的命令格式

一般的命令后面就跟着一些相关的变量参数等等；但命令名后必须留有一个空格，以便和参数等其他部分区别隔开。

格式是： [命令名] [参数 1 ，参数 2……]————注释：“[]”没有实际意义。

例如：

```
Set ArtNo=12345678,ArtDes="light-colored trousers"
Do ^P1(ArtNo,ArtDes),P2(ArtNo)
```

但也有些命令例如：Do、For、If、Else、Quit，后面是可以没有参数的。

ii. 有条件的命令

有些命令是需要有条件的。如果这些条件成立，这个命令才被执行。

格式是：[命令名]：[逻辑表达式] [参数 1 ，参数 2……]

例如：

```
Do:Sales>100000 ^SoBu(Sales),^Warehouse(type)
Set:var=1 var=2
Quit:i>31
Kill:t=9 var1,var2,var3
```

但是下面几个命令是不能有条件的：If、Else、For。

iii. 设备设置命令

有些命令是用来对设备进行设置的，如 Open、Use、 Close。

例如：

```
Open dev:(/CRT:/MARGIN=80)
```

iv. 超时设置

有 4 个命令是可以设置超时条件的：Read、Open、Lock、Job。

例如：

```
Read "Article Number: ",ArtNo:10
Open dev::0 Else  Write "Device not available"
```

2.4.3.2　一些常见的命令

下面我们将扼要介绍 Caché ObjectScript 语言的一些最常用的命令，在附录中收录了更多命令，大家可以参考。另外，如果对命令的使用需要更详细的说明，可以参看 Caché 的在线帮助文档。如前，ObjectScript 命令亦支持缩写（请留意下划线）。

i.　变量操作命令：Set, Merge, Kill, New, Lock

SET

格式：　Set[:cond] variable=expression

　　　　Set[:cond] (variablelist)=expression

给一个或者多个变量赋值。

例如：

```
Set x=5,n(1)=4
Set ^FILE=3
Set:a>0 (i,j,k)=1
Set @a=@b+1
Set $Piece(v,"*",3)="A"
```

MERGE

格式：　Merge [:cond] target=source

拷贝变量树。

例如：

```
Merge a=b
Merge var(1)=^G(1,2)
```

KILL

格式：　Kill[:cond]

　　　　Kill[:cond] variable

　　　　Kill[:cond] (variable)

删除所有变量或一些指定的变量或所有变量除了指定的变量。

例如：

```
Kill
Kill a,b,^C
Kill a(1,3)
Kill (v1,v2,v3)
Kill:bed p1,p2,@var
```

NEW

格式：　New[:cond]

　　　　New[:cond] variable[,…]

　　　　New[:cond] (variable[,…])

初始化变量。

例如：

```
New
New a,b
New(x1,x2)
```

LOCK

格式：　Lock[:cond]

　　　　Lock[:cond][+/-] variable[,…] [:timeout]

　　　　Lock[:cond][+/-] (variable[,…]) [:timeout]

设置或者取消一些变量的锁定。

例如：

```
Lock
Lock (a,^G)
Lock ^A(1,2)
Lock (b,^H):10
Lock +^A,-^PER(name)
Lock +(^P1,^P2)
```

ii.　程序流控制命令：　If, Else, For, Quit, Do, Goto, Break

IF

格式：If

　　　If expression

　　　If expression {…}

条件判断语句。

例如：

```
If t=1
If age>30 If sex="m"
If  Set x=1
```

ELSE

格式：　Else {…}

当判断条件不通过的时候，执行后面的命令行。

例如：

```
If a<1 Do ^P1(t1)
Else  Do ^P2(t1)
```

格式：　ElseIf expression {…}

当上层判断条件不通过的时候作另外的判断，并在条件通过时执行后面的程序段。

例如：

```
If a<1 Do ^P1(t1)
Elseif a>10  Do ^P2(t1)
Else Do ^P3(t1)
```

FOR

格式：　For {…}

For var=expr[,…]

For var=expr[,…]{…}

For var=start:increment:[end]

For var=start:increment:[end] {⋯}

循环命令。

例如：

```
For j=-9,5,7,36,-100
For i=1:1:10
For k="A","B",1:2:11
For l=1:1
```

QUIT

格式：　Quit[:cond]

　　　　Quit[:cond] expression

结束执行一个函数并返回一个值。

例如：

```
Quit
Quit:t=10
Quit a*b
```

DO

格式：　Do[:cond]

　　　　Do[:cond]{…}While expression

执行命令。

例如：

```
Do LoPro,^R1(lv,lv1)
Do ^Routine(a1,.a2)
Do:A=1 PROG:T=1
Do @nam^@b
Do ^P1:C1
```

GOTO

格式：　Goto [:cond]

　　　　Goto [:cond] location[:cond]

跳转命令。

例如：

```
Goto loclabel
Goto:t A:s=1,B:s=2
Goto L1:C1,L2:C2
```

BREAK

格式：　Break[:cond]

在调试的时候中断一个 routine 的执行。

格式：　Break[:cond][status]

中断执行或者取消中断执行。

例如：

```
Break
Break 1
Break "S"
```

iii.　输入输出命令：Read, Write, Open, Use, Close

READ

格式：　Read[:cond][f,][string,][f,] variable[:timeout]

　　　　Read[:cond][f,][string,][f,] #variable[:timeout]

　　　　Read[:cond][f,][string,][f,] variable #n[:timeout]

从当前的设备读信息。

例如：

```
Read x,^G1(ind)
Read "City: ",city
Read *z
Read:$Data(g) !,"Input? ",input#10
Read @a:10
```

WRITE

格式：　Write[:cond][f,] expression

　　　　Write[:cond][f,] #expression

输出信息到当前设备上。

例如：

```
Write "HELLO"
Write #!?10,*7
Write:'t a,b,!!,c+t/5_s
Write @a,@@v
```

OPEN

格式：　Open[:cond] device[:(parameter)] [:timeout]

开启一个设备以备使用。

例如：

```
Open device
Open 3,prnt::time
Open:'closed @tape
Open term:(Param):20
```

USE

格式：　Use[:cond] device[:(parameter)] [:timeout][:”mnespace”]

启用指定设备作为当前的设备，并可以设置一些属性。

例如：

```
Use device
Use:status="OPEN" dev
Use 3:(parameter)
Use @print
```

CLOSE

格式： Close[:cond]device[:parameter]

关闭一个设备。

例如：

```
Close "DEV",3
Close:bed>3 line
Close tty:(/DELETE)
```

iv. 其它： Job, Hang, Halt

JOB

格式： Job[:cond] routine[:cond] [(routineparms)] [:[process parms][:timeout]]

背景执行一个新的 Caché 进程。

例如：

```
Job ^A
Job B,A1^PROG
Job:g=1 J1::10
Job @var:(Parameter)
Job ^P1(4,x1,$Extract(name))
```

HANG

格式： Hang[:cond]seconds

暂停当前的 routine 的执行一段时间。

例如：

```
Hang 10
Hang:t=1 b/4
Hang @i
```

HALT

格式：Halt[:cond]

停止当前 Caché 的工作，并退出 Terminal 窗口。

例如：

```
Halt
Halt:cancel
```

2.4.3.3　　Z 开头的命令

在 Caché ObjectScript 中还有很多的命令是以 Z 开头的，一般来说这些命令可以被看作系统命令。它们主要有 3 个方面的功能。

第一、在程序中之用来编辑程序模块即 routine 的；

第二、用于设定断点和捕获出错信息的；

第三、系统的低端命令，一般是用来设置一些系统属性。

这里我们简要地介绍几个比较常见的 Z 开头的命令。

在程序中用作编辑 routine 的命令：ZInsert、ZLoad、ZPrint、ZRemove、ZSave、ZWrite

ZINSERT

格式：　ZInsert “code” [:location]

插入一行代码到正在编写的 routine 里面。

ZLOAD

格式：　ZLoad[:cond]

　　　　ZLoad routine

读一个 routine 用作编写。

ZPRINT

格式：　ZPrint

　　　　ZPrint line1[:line2]

输出指定的 routine 的行到当前的设备。

ZREMOVE

删除当前正在编写的 routine 的一些行。

格式：ZRemove

　ZRemove line1[:line2]

ZSAVE

格式：　ZSave

　　　　ZSave routine

保存一个 routine

ZWRITE

把所有的或者指定的本地变量或者全局变量都写到当前的设备上。

格式：　ZWrite

　　　　ZWrite variable

i. 设定断点和错误信息产生的命令：ZBreak、ZQuit、ZTrap、ZSYNC

ZBREAK

格式：　ZBreak

设置一个断点。

ZQUIT

格式：　ZQuit

　　　　ZQuit [:cond] expression

移去所有的或者部分执行栈的执行层。

ZTRAP

格式：　ZTrap[:cond]

　　　　ZTrap[:cond] expression

产生一条错误信息。

ZSYNC

格式：　ZSYNC

保证所有的逻辑事务在物理上已经结束。

ii.　系统的低端命令：ZKill、ZNspace、ZZDUMP

ZKILL

格式：　ZKill variable

删除变量但不删除它的子节点。

ZNSPACE

格式：　ZNspace expression

更改当前的 namespace。

ZZDUMP

格式：　ZZDUMP (expression)

把字符串作为 16 进制写到当前的设备上。

2.4.3.4　间接运算符@的使用

根据它的使用不同，间接运算符@的类型可以被分为 4 种。

i. 名称间接运算

名称间接运算可以用在 Caché 需要一些变量名称或者是引用 routine 的地方，通过一个例子我们来了解它的作用：

```
Set pname="^Prog1" Do @pname
```

这个命令就间接调用了^Prog1 这个 routine。

ii. 参数间接运算

参数间接运算可以用在一些需要参数的命令中。例如，

```
Set isetarg="x=1",@isetarg
```

这个命令相当于执行 Set x=1。

```
Set ikill="(e,f,g)" Kill @ikill
```

这个命令相当于执行 Kill e,f,g。

```
Set inew="(a,b,c)" New @inew
```

这个命令相当于执行 New a,b,c

iii. 下标间接运算

下标间接运算用在一些需要给前面的表达式增加一层参数的地方。

例如：

```
>Set x(2,5,3)="SubInd" Set field="x(2,5)",d1=3
>Write @field@(d1)
SubInd
```

这个例子就说明了后面的这个@把 3 这个参数作为后面的下标加到前面的表达式中。

iv.　格式间接运算

格式间接运算可以用于间接表示一些格式的声明。

例如：

```
Set lvpattern="1.3N" If input'?@lvpattern Do Error
```

2.4.4 Caché ObjectScript 语言的内部函数

Caché ObjectScript 语言提供了很多内部函数，这些函数可以被作为一个表达式那样调用。它们也可以把对方作为自己的参数。

2.4.4.1　内部函数的格式

Caché ObjectScript 的内部函数以$符号开头。格式是：$函数名（参数 1，参数 2，……）。函数名的大小写是没有限制的。除了$ListBuild 函数以外，其它的函数都有一个或者一个以上的参数。

例如：

```
>Write $Extract("Summersmog",7,10)
smog
>Set dl=2*$Length("Summersmog") Write dl
20
>Set zk="Spring,Summer,Fall,Winter"
>Write $Extract(zk,8,$Length(zk))
Summer,Fall,Winter
```

2.4.4.2　内部函数的分类

Caché ObjectScript 语言的内部函数可以被大致分为下面一些类型：

2.4.4.2.1 普通的函数

$Ascii, $CASE, $Char, $Random, $Select, $STack, $Text, $View

$ISObject, $ZF, $ZHex, $ZISWide,$ZLAscii, $ZLChar, $ZName,$ZSEArch, $ZSEEK, $ZWAscii,$ZWBPack, $ZWBUnpack, $ZWChar, $ZWPack,$ZWUnpack

变量和数据的操作函数

$Data, $Get, $Order, $NAme, $Query, $QSubscript, $QLength,$SORTBEGIN, $SORTEND

2.4.4.2.2 字符串函数

$Extract, $Find, $Length,$Piece, $Reverse, $TRanslate
$ZCONVert, $ZSTRIP, $ZPosition, $ZWidth, $ZZENKAKU

2.4.4.2.3 数字函数

$FNumber, $Justify, $INumber, $NUMber

2.4.4.2.4 列表函数

$LIst, $ListBuild, $ListData, $ListFind, $ListGet, $ListLength

2.4.4.2.5 增量函数

$Increment

2.4.4.2.6 数学函数

$ZABS, $ZEXP, $ZLN, $ZSIN 等等。

2.4.4.2.7 时间函数

$ZDate, $ZDateTime, $ZDateH, $ZTime, $ZTimeH

2.4.4.2.8 位串函数

$BIT, $BITCOUNT, $BITFIND, $BITLOGIC, $ZBOOLEAN, $ZCyc

2.4.4.2.9 性能函数

$ZUtil(n)

以上这些函数的使用办法和详细说明可以参见附录，或者联机帮助文档。下面只简要介绍一些常见的函数。

2.4.4.3　常见的内部函数

i. $Ascii

格式：$Ascii(expression[,position])

说明：返回一个字符的 ASCII 值。

例如：

```
>Write $Ascii("A")
65
>Write $Ascii("ABC",3)
67
>Write $Ascii("")
-1
```

ii. $CASE

格式：$CASE(expression,result:value,…)

说明：判断表达式的值，返回相应的值。

例如：

```
>Write $CASE(A,1:"One",2:"Two",3:"Three",:"something else")
>Goto $CASE(Input,"*":End,:^Process(Input))
```

iii. $Char

格式：$Char(expression[,…])

说明：以一个整数型的串里面的数字作为 ASCII 码值来产生一个字符串。

例如：

```
>Write $Char(65)
A
```

```
>Write $Char(65,66)
AB
>Write $Char(-1)
(空字符串)
```

iv. $Random

格式：$Random(range)

说明：返回一个随机数。

例如：

```
>Write $Random(10)
5
```

v. $Select

格式：$Select(expression:value,…)

说明：返回第一个为真的表达式对应的值。

例如：

```
>Set a=1
>Write $Select(a=1:5,a>1:0)
5
>Write $Select(a=2:5,1:0)
0
>Set min=$Select(s<t:s,1:t)
```

vi. $Text

格式：$Text(location)

说明：返回一个 routine 的指定行的源文件。

例如：

```
>Write $Text(+3)
 Read "Input: ",x
>Write $Text(+0)
P1Spec
```

```
>Write $Text(Label)
Label Set a=1,b=2
```

vii. $View

格式：$View(address[,mode,length])

说明：返回内存地址里的内容。

viii. $Extract

格式：$Extract(expression,[,from[,to]])

说明：返回字符串的指定部分。

例如：

```
>Write $Extract("ABC")
A
>Write $Extract("XYZ",$L("XYZ"))
Z
>Write $Extract("AABB",2,3)
AB
>Write $Extract("Summer",3,255)
mmer
>Write $Extract("abc",5)
(空字符串)
```

ix. $Find

格式：$Find(string,substring[,position])

说明：返回子串的最后的位置。

例如：

```
>Write $Find("ABC","A")
2
>Write $Find("XYZ","T")
0
>Write $Find("ABABAB","AB",3)
```

```
5
```

x. $Justify

格式：$Justify(expression,width[,desimal])

说明：返回一个右对齐的值，并可以改变它的格式。

例如：

```
>Write $Justify(12,3)
 12
>Write $Justify("Text",10)
      Text
>Write $Justify(12,3,2)
12.00
>Write $Justify(3.14,1,0)
3
>Write $Justify(.414,6,3)
 0.414
```

xi. $Length

格式：$Length(expression[,delimiter])

说明：返回一个字符串的长度，或者是被分隔符的子串的个数。

例如：

```
>Write $Length("ABCD")
4
>Write $Length("")
0
>Write $Length("AB/CD/EF","/")
3
```

xii. $Piece

格式：$Piece(expression,delimiter [,from[,to]])

说明：返回一个或者多个被分隔符分开的子串。

例如：

```
>Set v="ABC/XYZ/123"
>Write $Piece(v,"/")
ABC
>Write $Piece(v,"/",2)
XYZ
>Write $Piece(v,"/",2,3)
XYZ/123
>Set $P(v,"/",2)="***" Write v
ABC/***/123
```

xiii. $Reverse

格式：$REverse(string)

说明：逆序返回一个字符串。

例如：

```
>Write $Reverse("Rail")
liaR
```

xiv. $TRanslate

格式：$TRanslate(string,replace[,by])

说明：替换一个字符串中的字符，并返回替换后的结果。

例如：

```
>Set u="EPUR",l="epur"
>Write $TRanslate("UPPER",u,l)
upper
>Write $TRanslate("train station","aeiou")
trn sttn
```

其它的函数的说明参见附录或者联机帮助（documentation）。

2.4.4.4　　Z 开头的内部函数

i.　数学函数

三角函数：$ZSIN, $ZCOS, $ZTAN, $ZCOT

反三角函数： $ZARCSIN, $ZARCCOS, $ZARCTAN

对数函数：$ZLN 、$ZLOG

指数函数： $ZEXP

ii.　时间日期的函数

$ZDate　把$Horolog 格式的日期值按照指定格式显示。

$ZDateH　$ZDate 的反函数。

$ZDateTime　把$Horolog 格式的日期时间值按照指定格式显示。

$ZDateTimeH　$ZDateTime 的反函数。

$ZTime　把$Horolog 格式的时间值按照指定格式显示。

$ZTimeH　$ZTime 的反函数。

iii.　位串函数

$BIT　产生一个位串（bit string）。

$BITCOUNT　数一个位串的位数。

$BITFIND　搜索一个位值在一个位串里的位置。

$BITLOGIC　对一个位串进行位运算。

$ZBITAND　返回两个位串的与结果。

$ZBITCOUNT　返回位串中值为 1 的个数。

$ZBITFIND　返回该值从指定位置开始第一次出现的位置。

$ZBITGET　返回位串中指定位置的值。

$ZBITLEN　返回位串的长度。

$ZBITNOT　返回位串非运算的结果。

$ZBITOR　返回位串或运算的结果。

$ZBITSET　对位串的指定位置进行赋值。

$ZBITXOR　　返回位串异或运算的结果。

iv.　一般的 Z 函数

$ZF　　在操作系统的环境下执行命令行。

$ZHex　　把一个 16 进制的数变为一个 10 进制的数，反之亦然。

$ZNAME　检查一个字符串是不是标准的名称格式。

v.　字符串函数

$ZConVerT　　返回一个按照指定模式表示的字符串。

$ZCyc　　与程序间的通信有关的函数。它产生的检查和可以检查数据的有效性的。产生方式是 XOR（异或）。

$ZSTRIP　　在一个给出的字符串中，删除或者保留一些字符。

2.5 Caché ObjectScript 语言对列表的操作

2.5.1 列表的定义

列表包含了一些独立的元素。列表在传统的关系型数据库和面向对象的数据库中都有使用。

下面用几个例子来说明一个列表的一些性质：

一个有三个元素的列表：

L1={甲,乙,丙}

一个空列表：

L2={}

第三个元素是未定义的列表：

L3={甲,乙,,丙}

第三个元素是空串的列表：

L4={甲,乙,"",丙}

以一个列表作为一个元素的列表：

L5={甲,{丁,子,丑},乙,丙}

定义一个列表有两种做法：

```
Set L1="red/green/blue"
Set L1=$ListBuild("red","green","blue")
```

我们推荐使用第二种。

2.5.2 列表的操作

$Length

带有两个参数的$Length 函数，可以计数一个列表的元素个数。

$Piece

$Piece 取出列表的某一个元素。

Set

Set 命令可以新增或者替换列表的某一个元素。

例如：

```
Set list="spring/summer/fall/winter"
Set address="Wagner^John^2712, Washington Street^Cedar Rapi ds/IA^52405^(319) 696-4521*(319) 694-6077"
>Write $Length(list,"/")
4
>Write $Piece(address,"^",3)
2712, Washington Street
>Set $Piece(address,"^",5)="52405-1966"
>Write address
Wagner^John^2712, Washington Street^Cedar Rapids/IA^52405-1 966^(319) 696-4521*(319) 694-6077
```

2.5.3 列表函数

Caché ObjectScript 定义了 6 种列表函数

i. $ListBuild ，缩写为$LB，用来产生列表。

$ListBuild("a","b")

$ListBuild("a","b")_$ListBuild()

$ListBuild("a","b")_$ListBuild("")

这 3 个命令，前两个中只有两个元素，而第三个中有一个空的元素。

例如：

```
Set L1=$ListBuild("甲","乙","丙")
L1={甲,乙,丙}
Set L2=$ListBuild()
empty list L2={}
Set L3=$ListBuild("甲","乙",,"丙")
L3={甲,乙,,丙}
Set L4=$ListBuild("甲","乙","","丙")
L4={甲,乙,"",丙}
Set L5=$ListBuild("甲",$ListBuild("丁","子","丑"),"乙","丙")
```

ii. $ListLength，缩写为$LL ，返回列表的元素个数

例如：

```
>Write $ListLength(L1)
3
>Write $ListLength(L2)
1
>Write $ListLength(L3)
4
>Write $ListLength(L4)
```

```
4
>Write $ListLength(L5)
4
```

iii. $LIst ，缩写为$LI， 取出一个或者多个列表的元素

例如：

```
>$LIst(L1,2)
乙
>$LIst(L1,2,2)
乙
>$LIst(L1,0)
出错信息
>$LIst(L1,0,2)
和$LIst(L1,1,2)一样
```

iv. $ListGet ，缩写为$LG， 和$Get 一样，但是取消了对空值引用的报错。

例如：

```
>Write $ListGet(L3,3)
      （空值，不报错）
>Write $ListGet(L3,3,"申")
申
```

v. $ListData， 缩写为$LD， 检查一个列表的元素是不是有值。

例如：

```
>Write $ListData(L3,2)
1
>Write $ListData(L3,3)
0
```

vi. $ListFind ，缩写为$LF ，从指定位置开始找列表中的某一个

值。

例如：

```
>Write $ListFind(L3,"乙")
2
```

3 Caché ObjectScript 语言对 Routine 的使用

我们可以把 Routine 理解为“小程序”或程序模块。Caché ObjectScript 对 Routine 的使用类似于传统关系型数据库的存储过程，但是和存储过程也有不一样的地方。例如，对一个类或者对对象实例的操作的存储过程，在 Caché 中通常是被封装为类方法或者是对象方法。而 Routine 实现的功能和应用也比存储过程要多一些。而且 Routine 可以嵌入很多其它语言的程序段，或者可以调用一些外部函数。

3.1 Routine 的分类

i. Macro code（宏代码），后缀名为.MAC。它可以包含 Caché ObjectScript 代码、macro directives、macros、嵌入式 SQL、HTML 和 JavaScript。它被编译成中间代码然后被写入到对象的代码中。
ii. Macro include（宏包含代码），后缀名为.INC。它可以包含和 Macro code 一样的代码。它是用来建立代码库的，然后在 macro routines 中，可以被包含（include）进去。
iii. Intermediate code（中间代码），后缀名为.INT。它可以是任何 Caché ObjectScript 代码，M 语言和嵌入式 SQL 语言都是先被转化成中间代码的。
iv. Object code（对象代码），后缀名为.OBJ。在中间代码被执行以前，先要转化为内部的对象代码。

3.2 Routine 的基本结构

Caché 的 Routine 是由一些代码片段组成的。在编译的时候，Routine 会被编译成特殊的类。这个类的代码不是机器码，而是一种跨平台的代码。

3.2.1 Routine 的命名

Routine 的名字可以是任何的字母数字式字符。但是有要求：一般不能以数字或百分号％开头命名，和句号不能在开始或者结束的位置。以％百分号开头命名的 Routine 是不局限在某一个名字空间（Namespace）应用的，％号它是 Caché 系统内部保留的，只是系统为我们提供的程序命名所使用。此外，对 Routine 名字的长度也没有限制，但是只有前 31 个字符是用来对不同的 Routine 加以区分的。

3.2.2 Routine 的程序行

程序行是 routine 最基本的单位之一。我们这里说的是一个逻辑上的行，因为有的逻辑上的行在物理上占据了可能超过一行的位置。Routine 的行有 3 种不同的类型，下面作一一的介绍。

3.2.3 标签行

一个标签要确定一行，它从当前行的第一个位置开始。它的命名条件和 routine 的命名条件类似。但是它的长度不能超过 31 个字符。通常一个标签后面有一些空格，后面可以跟一些 Caché 的可执行命令。

3.2.4 代码行

代码行的最前端至少要有一个空格，作为一行的开始部分。

3.2.5 注释行

有 4 种形式的注释符号：

；符号后面的本行内容将不被执行。

；；一些时候，注释的内容在执行的过程中是需要用到的。两个分号的注释内容是被编译进去的，而一个分号的注释在编译的时候就被去掉了。

// 和一个分号是一样的。

/* ……*/中间省略号的部分是被注释的内容。

注意：一个 Caché 行的大小不能超过 4k。

3.2.6 Routine 的代码片段

代码片段或者叫代码块是比行大一些的单位。代码块一共有 3 种形式：

a) “点”片段

通常是由 Do 命令和一些点组成的，它们的结构是：

```
Do   // Do 命令，不带参数
 .   // 第一层
 . <代码>
 . Do
 ..   // 第二层
 .. <代码>
 .   // 第一层的代码，第二层到这里结束。
. <代码>
    // 没有点，主块层
 Quit
```

块里面可以调用一些外面的 Routine，但是不能使用 goto 等命令从一层跳到另一层。

b) 花括号块｛｝

一队花括号可以包含很多 Routine 行。

c) 流控制命令块

在这些命令中的可执行部分也是用花括号括在一起的。

3.3 Routine 本地变量的使用

3.3.1 Routine 的组成

3.3.1.1 Routine 可以由一些标签开头，Quit 结尾的结构组成。

例如：

```
P1   //
 Set a=1
 Do P2
 Quit
P2   //
 Set x=2,y=3
 Quit
```

在这里面，可以直接采用 Do P2^P1 就可以调用 P1 的子 routine P2。

也可以采用 Do P1+3^P1 调用 P1，但是是从第三行开始执行的。

3.3.1.2 5.3.1.2 标签也可以带一些参数

例如：

```
Invert(x1,x2,x3)   //
```

3.3.1.3 标签也可以为 private 或者是 public 的。

例如：

```
Move(x) Private  //
Rotate(x) Public  //
Invert(x)          //
```

如果没有写后面的关键字，会被默认为 public 的。

3.3.1.4 在一个有参数的标签前面的片段如果没有写 Quit，也会自动退出执行的。

3.3.2 routine 的变量有效区

变量在 Caché 中，在当前的进程分区是可见的。所以，前面的例子中，a 在 P2 里面也是可见的。Caché 里面有一些可以释放本地变量的命令：

Kill 在一个 routine 的最后，可以释放不再使用的本地变量。

New 在一个 routine 的开始定义一些本地变量，这些变量将在 routine 结束的时候也被释放掉。

例如：

```
P1    //
    New a
    Set a=1
    Do P2
    Quit
P2    //
    New x,y
    Set x=a,y=3
    Quit
```

3.4 参数传递

在以下几个情况下，将用到参数的传递：调用一个 routine，调用一个用户定义的函数，调用一个过程。

3.4.1 值传递

调用的时候，直接写变量的名称，传递的将是变量的值。

例如：

```
    Set a=1,b=2,c=3
    Do P1(a,b,c)    //
    ……
P1(r,s,t,)          //
    Set Sum=r+s+t   // 它们得到的值是 a,b,c 的值。
    ……
    Quit
```

3.4.2 传递引用

调用的时候，在写变量的名称的前面写上一个点号，传递的将是变量的值。

例如：

```
    Kill var      //
    Do P1(.var)   //
    ……
P1(x)             //
    Set x=0,x(1)=1,x(2),x(3)=3
    ……
    Quit
```

在调用 P1 以后，var 就存在于 routine 中了，它的值就是在 P1 中被赋的值。

3.4.3 混合使用

值传递和引用的传递是可以一起使用的。

例如：

```
Do ^Sum(a,b,c,.sum)
```

3.5 过程

3.5.1 过程的结构：

<过程名> (<参数列表>) [<公共参数>] <访问方式>

{

代码

}

虽然参数列表部分必须有，但是可以是一个空的列表。

3.5.2 过程的代码：

3.5.2.1 一个过程必须以一个标签作为开始，而花括号和标签的位置是可以两者任何一个在前面的。

3.5.2.2 在一个过程里面的所有的标签都是 private 的，所以，在过程里面的标签的访问属性可以写明 private，不过没有这个必要。但是，如果写 public 是会出错的。

3.5.2.3 虽然在同一个过程里面的标签名是不能一样的。但是在不同的过程中，可以采用一样的标签名。

3.5.2.4 如果一个 do 命令或者一个调用外部函数的命令在没有写明 routine 的时候，会被认为就是调用本过程里面的子 routine 或者函数。

3.5.2.5 当一个 routine 或者函数带着它的 routine 的名称的时候，在被调用的过程中，（例如，LABEL1^ROU1）系统将在 ROU1 里面搜索

LABEL1 的。就算当前就是 ROU1 这个 routine 里面，它也会把这个调用作为外部调用处理。

3.5.2.6 Goto 只能跳转到内部的标签。

3.5.2.7 标签+偏移的使用方式在过程内部是不支持的。

3.5.2.8 在一个过程内部的$Test 的更改在外部是看不到的。

3.5.2.9 在花括号}后面不能再有代码了。

3.5.2.10 一个隐藏的 Quit 将在}前被执行

3.5.2.11 间接命令和 Xecute 在过程中的使用，和在外面的使用是一样的。在过程中 Xecute 的执行就象它是一个外部的子 routine 一样。

3.5.2.12 由于间接调用被当作 public 一样，而 goto 只能在过程内部跳转，所以在一个过程内部，Goto @A 是不支持的。

3.5.3 过程中的变量：

在公共参数列表中的变量都是公共的（public），其它的都是内部的（private）。

3.6 外部函数

调用外部函数的方法就是在函数名前面加上$$符号。调用可以是值传递也可以是引用传递。

格式是：

$$函数名（参数 1，参数 2，……）

例如：

```
Write $$Random(n,.m)
```

如果您需要完整的 routine 的代码的例子，可以在安装 Caché 后，查看里面的例子（SAMPLE），里面有许多的比较完整的 routine 的代码可以参看，就可以进一步了解上面叙述的模式和功能了。

4　Caché ObjectScript 的结构化编程

在 Caché ObjectScript 的结构化编程中有四种命令块：

If <表达式>[,<表达式> ……] { 代码 }

ElseIf <表达式>[,<表达式> ……] {代码}

Else {代码}

For <for 参数> {代码 e }

WHILE <表达式>[,<表达式> ……] {代码}

Do {代码} WHILE <表达式>[,<表达式>……]

4.1　流控制函数块

4.1.1　表达式列表

在 If、WHILE、Do/WHILE 命令块中有表达式列表，它由一个或者多个表达式组成的。这些表达式可以用逗号分开，而且是从左到右执行的。只有这里面所有的表达式都是真的话，命令块中的代码才被执行。

4.1.2　选择语句 If/ElseIf/Else

格式：

i. If <表达式>[,<表达式> ……] { 代码 }

ii. If <表达式>[,<表达式> ……] { 代码 }
    Else { 代码 }

iii. If <表达式>[,<表达式> ……] { 代码 }
    ElseIf <表达式>[,<表达式> ……] { 代码 }
    ……

例如：

```
If b=5 {
 Set a=1
} ElseIf b=6 {
 Set a=2
} Else {
 Set a=4
}
Goto x
```

4.1.3 循环函数 For

格式 1：

For for 变量=<表达式>[,<表达式> ……] { 代码 }

例如：

```
For i=1,3,5,7 { Write !,i," is odd number " }
```

这里 i 的值将是后面列出的 1、3、5、7 这些值。

格式 2：

For for 变量=<数字表达式 1>:<数字表达式 2>:<数字表达式 3> {代码 }

数字表达式 1 表示初始值，数字表达式 2 表示增量，数字表达式 3 表示最终值。

例如：

```
For i=1:1:10 { Write !,i," ",i**2 }
For delta=-2:.1:0 { Do ^function(delta) }
```

格式 3：

For for 变量=<数字表达式 1>:<数字表达式 2>{代码 }

数字表达式 1 表示初始值，数字表达式 2 表示增量，没有最终值。

例如：

```
Set pa=1
```

```
For p=3:2 {
 If $$^PrimeTest(p) {
  Write !,p," is prime" Set pa=pa+1
  If pa>100 {
   Quit
  }
 }
}
```

格式 4：

For {代码 }

没有参数，流的控制由代码块中的内容进行控制。

例如：

```
For {
 Write $Char(7)
 Hang 1
 Quit:$Piece($Horolog,",",2)>72000)
}
```

混合格式：

上面的格式可以混合使用，通过下面的两个例子可以说明两种不同的混合方式。

混合使用：

```
For lv=2,5,49,1:1:15,1:2 { ……}
```

嵌套使用：

```
For i=1:1:3 {
 For j=1:1:5 {
  Set m(i,j)=0
  If i=j {
   Set m(i,j)=1
```

```
   }
  }
 }
```

4.1.4 循环函数 WHILE 和 Do/WHILE

格式：

WHILE <表达式>[,<表达式> ……] { 代码 }

Do { 代码 } WHILE <表达式>[,<表达式> ……]

例如：

```
WHILE x>3,y>4 {
  Set a=5
  Do Label
 }
 Write "complete"
Do {
  Set a=5 Do Label
 } WHILE y>4,y<10
 Write "complete"
```

4.1.5 CONTINUE 命令

在循环函数的内部，可以使用 CONTINUE 命令。它的作用是跳到下一个循环层。

例如：

```
Set pa=1
For p=3:2 {
 CONTINUE:'$$^PrimeTest(p)
 Write !,p," is prime" Set pa=pa+1
 If pa>100 {
```

```
  Quit
 }
}
```

这里 CONTINUE 命令后面跟了一个条件，如果后面的返回值是 1（即为真）的话，将继续执行本层循环，否则将继续下一层的循环。

4.2 {代码}块的使用

4.2.1 规则

一个命令块由代码块的开始处开始输入，而不是使用一个标签作为开始。

一个代码块也是从开始处开始输入，也不使用标签。

4.2.2 代码块的例子

```
If x>3,y<4 { Set z=1 Do label1 } Else  { Set z=2 Do label2 }
```

```
If x>3,y<4 {
 Set z=1
 Do label1
} Else  {
 Set z=2
 Do label2
}
```

```
For y=1:1:10 {
 If abc(y)>0 {
   Set x=x+abc(y)
```

```
 } Else {
   Set x=x-abc(y)
 }
 Do tag
}
```

4.3 基于行的流转控制

基于行的流转控制指的是，当一些命令，例如 if 、else 或者是 for 被写在同一行里面的时候的情况。虽然，我们不推荐把许多的命令写在同一行，这样会降低可读性（最好不要把一些基于块的命令结构写在一行里）。但是，对于一些情况还是需要加以介绍说明的。

我们通过一些例子来说明这些问题。

例子一：

```
If x<1 { write "ok！ " Else  Write "Timeout" } Set x=1
```

由于花括号的存在，else 命令实际是只存在于花括号内部的代码块中产生作用。所以，后面的 set x=1 是会被执行的。

例子二：

```
If x<1,y<1 Set a=1 If z { Set b=2
Set c=3 } Else { Set k=5 }
If x<1,y<1 Set a=1 If z { Set b=2 Set c=3 }
Else { Set k=5 }
```

上面两段代码都会出错。因为在 if、else、for 基于行的结构中，是不能随便拆开成两行的。

在第一个片段中，第一个 if 是基于行的格式，尽管第二个 if 有了花括号形成的代码块格式，也不能拆开。它们必须都放在同一行。

在第二个片段中，else 是属于第二个 if 命令的，也由于行开头的时候的那个 if 是基于行的，所以第二个 if 的 else 部分也必须放在这一行里面。

例子三：

基于行的 for 的功能和基于块的 for 命令的功能是一样。下面两个程序段就是等效的：

```
If x>3 {
 Set a=""
 For  {
  Set a=$Order(abc(a)) Quit:a=""  Set x=x+abc(a)
 }
 Set y=y+x
}
```

```
If x>3 {
 Set a="" For  Set a=$Order(abc(a)) Quit:a=""  Set x=x+abc(a)
 Set y=y+x
}
```

4.4 $CASE 的使用

$CASE 是$Select 函数的一个变形。它属于分支命令的一种。

格式是：

$CASE(<表达式>,<表达式>:<值>[,<表达式>:<值> ……][,:<默认值>])

第一个表达式如果匹配某一个后面的表达式，那么将返回该表达式后面的值。如果找不到匹配的表达式，那么将返回最后的默认值。

例如：

```
Set x=1
```

```
Set y=$CASE(2*x+1,1:"A",2:"B",3:"C",:"Z")
```

$CASE 也可以作为 do 和 goto 的参数来使用。

例如：

```
Do $CASE(a,1:^P1(3),2:^P2(2),3:^P3(1),:Error)
```

如果 a=1，那么效果就相当于 do ^P1(3)。

在 goto 后面的使用也是一样的。

$Select 函数的使用和格式和$CASE 是一样的，但是前者在过程中的使用是有限制的，所以一般的情况下，在过程的内部，我们推荐使用$CASE 函数。

第四章 以 Caché 开发应用程序

1 引言

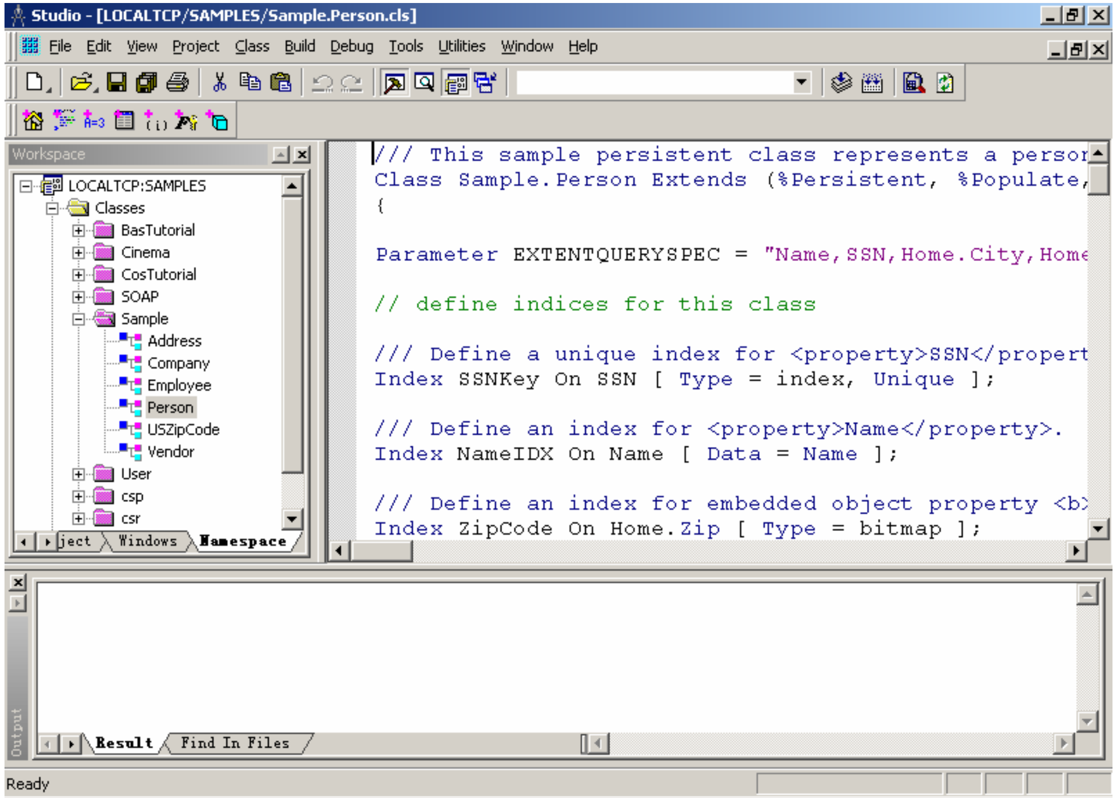
从这部分开始，我们将开始讨论如何用 Caché 开发应用程序。这里假设读者已经安装了 Caché5.0 或者更高的版本，并且看完了本书前面的章节。那么我们现在就有了一个由 Caché 安装时自动提供的叫做“SAMPLES”的命名空间，（你可以在启动 Caché 后使用 Explorer 来看到它及其内容），我们以后的例子都是在这个命名空间下完成的。我们将通过这些例子向大家讲解如何用各种开发工具连接 Caché 并且操作 Caché 中存储的数据。同时我们也将要向大家介绍如何设计应用程序的体系架构。

为了便于后边我们将给出的开发应用示例的的演示需要，我们先稍微修改一下“SAMPLES”里面的内容，以使它符合我们这个例子的开发的要求，另外我们还要为此做另一个准备工作即帮助大家配置一下 Caché 的 ODBC。让我们首先来介绍一下这些准备工作的具体做法：

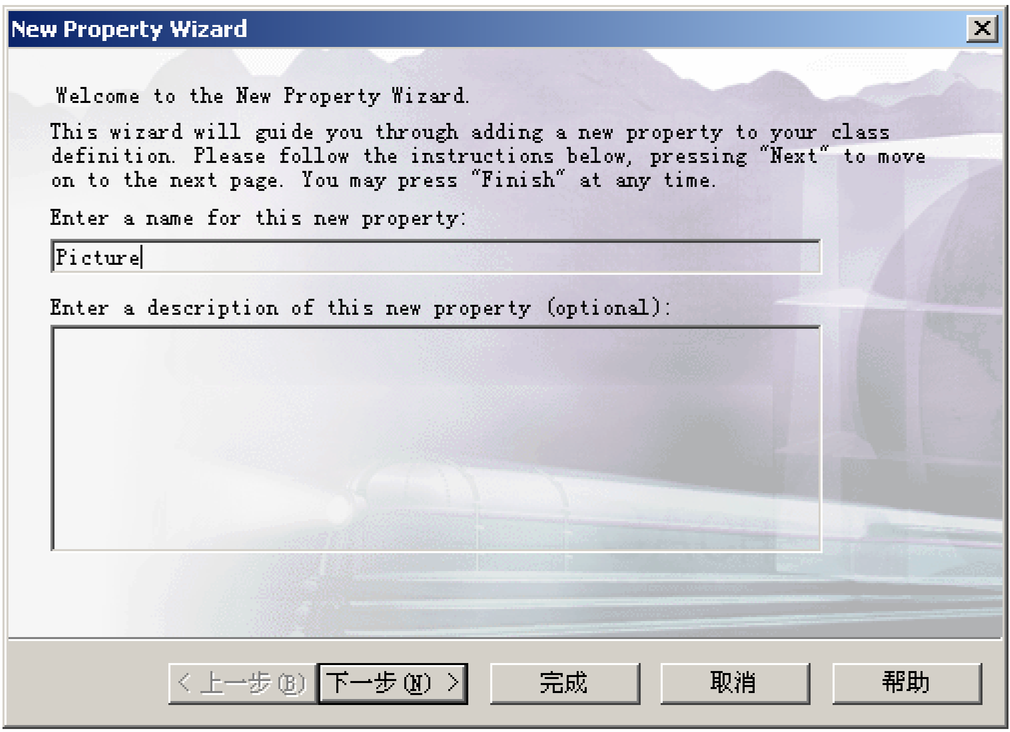
1.1 给 Sample.Person 增加一个 BinaryStream 属性

由于 Caché 默认安装的“SAMPLES”的命名空间中的 Sample.Person 类没有用于存放图片的属性，我们要为它添加一个这样的属性，来演示如何操作 BinaryStream。

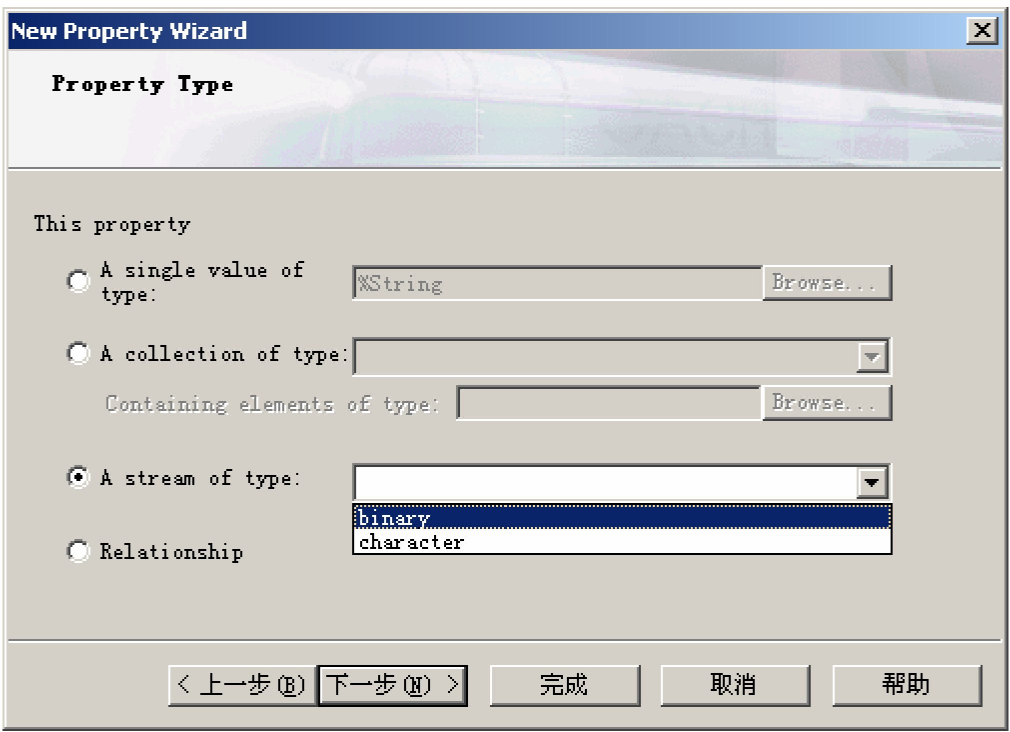
首先打开 Caché Studio，切换到“SAMPLES”的命名空间，再打开 Sample.Person 这个类。（*从 Caché 视窗的菜单条上 File 菜单中选用 Open Project 命令选择 Project Sample，即可在 Classes 中见到 Sample.Person 这个类）



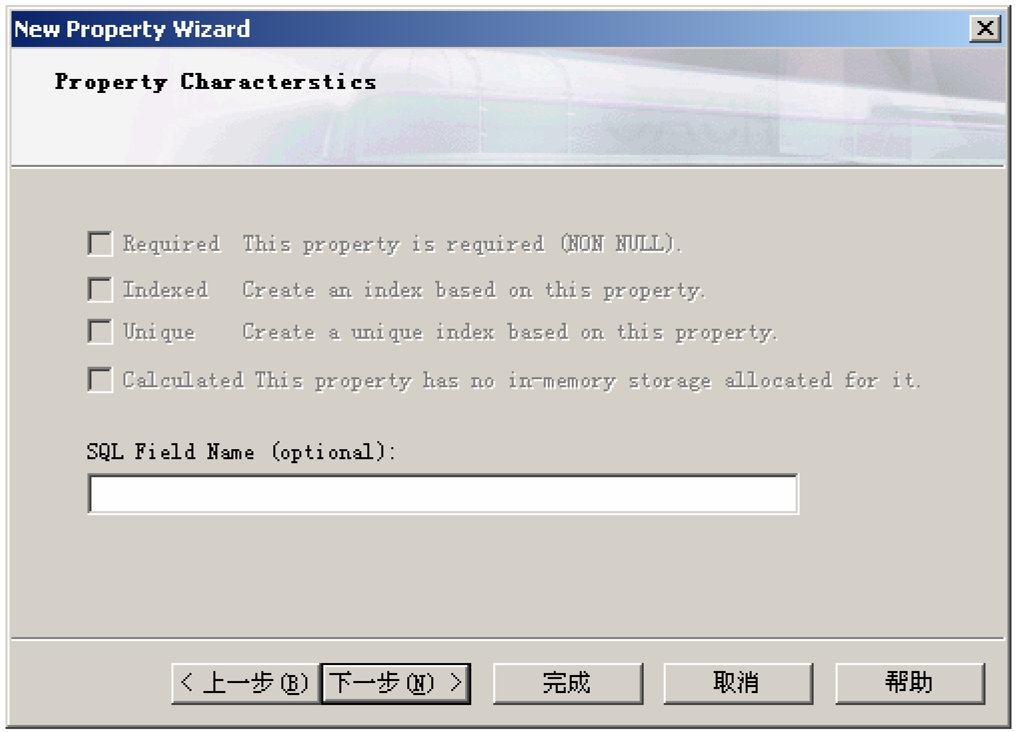
然后点添加属性用的按钮：(*即点击在工具条上的添加 New Property 的屋形图标按钮，将显示出下面的 New Property 向导窗口)



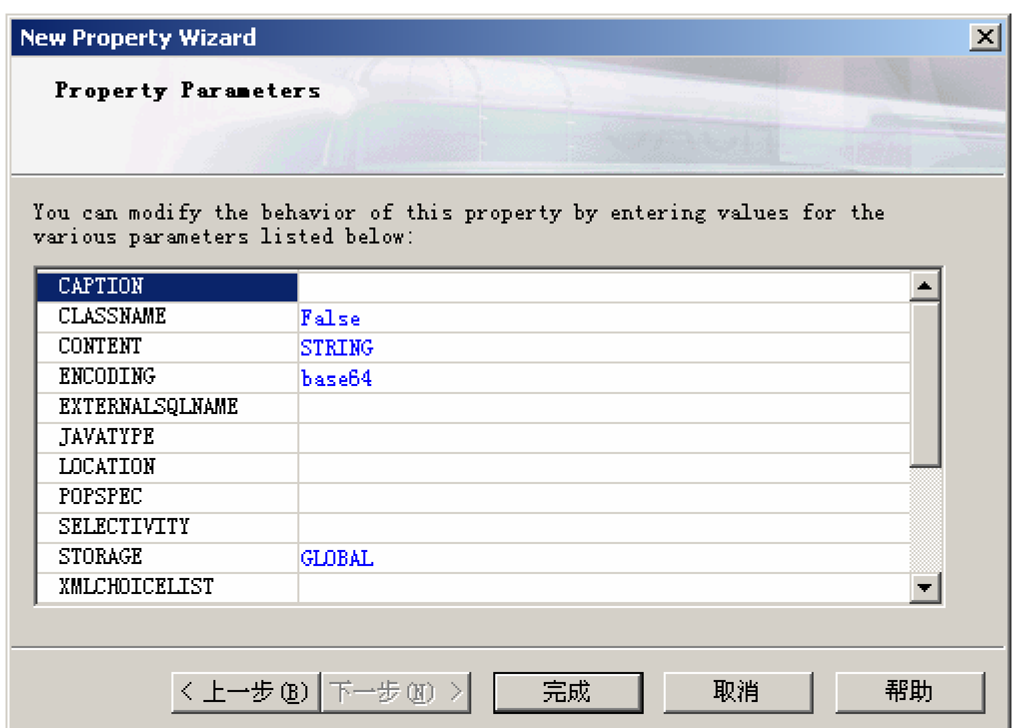
输入 Picture，然后点下一步按钮。



选择这个属性为 Binary Stream，然后点下一步按钮。



点下一步按钮。



点完成按钮。然后你会在 Sample.Person 的类定义里面看到类似：

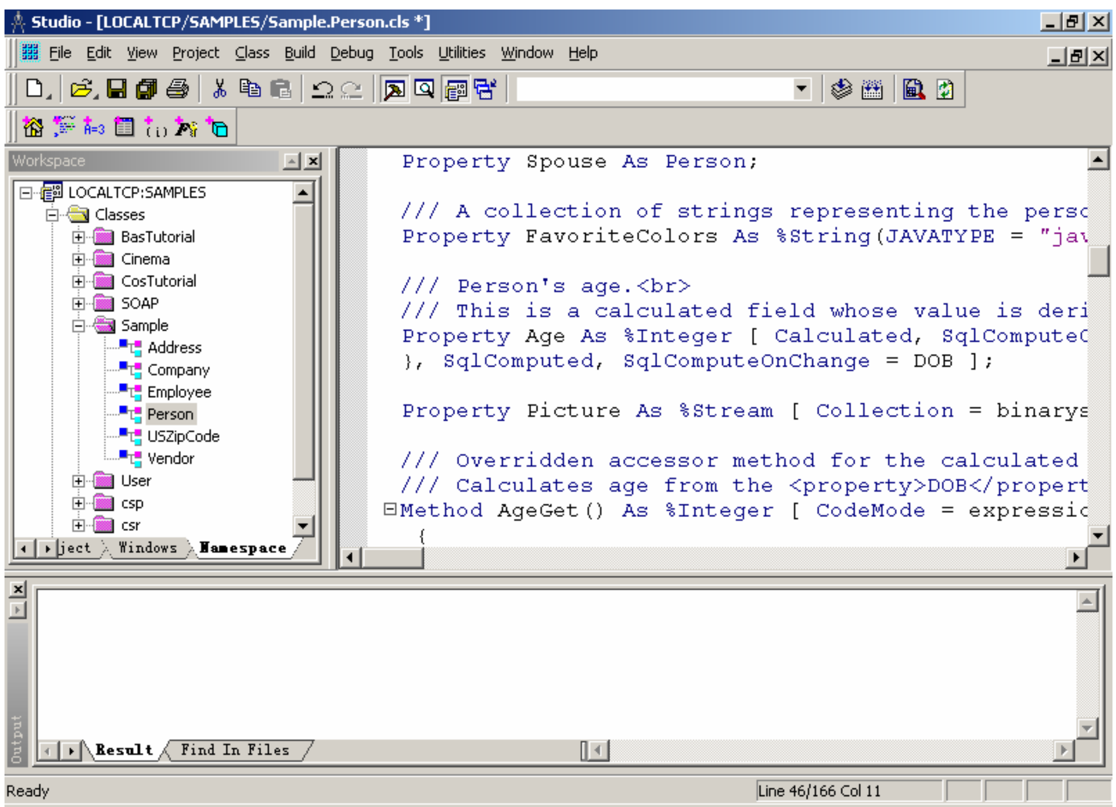
```
Property Picture As %Stream [ Collection = binarystream ];
```

的内容。

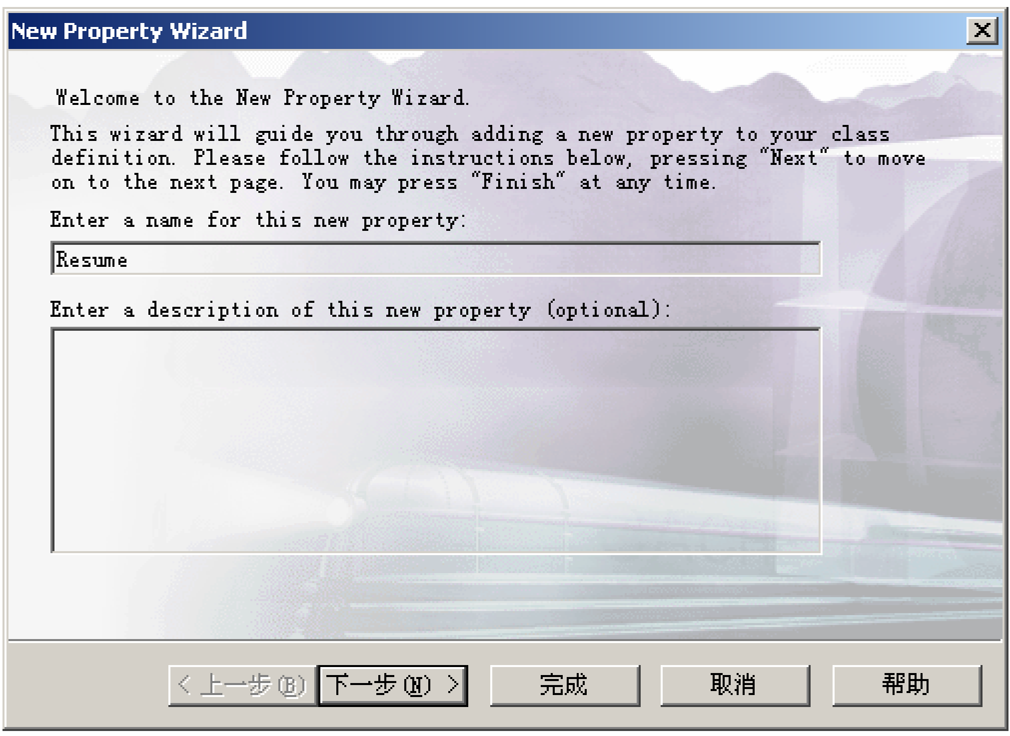
1.2 给 Sample.Person 增加一个 CharacterStream 属性

由于 Caché 默认安装的“SAMPLES”的命名空间中的 Sample.Person 类没有用于存放大文本的属性，我们要为它添加一个这样的属性，来演示如何操作 CharacterStream。

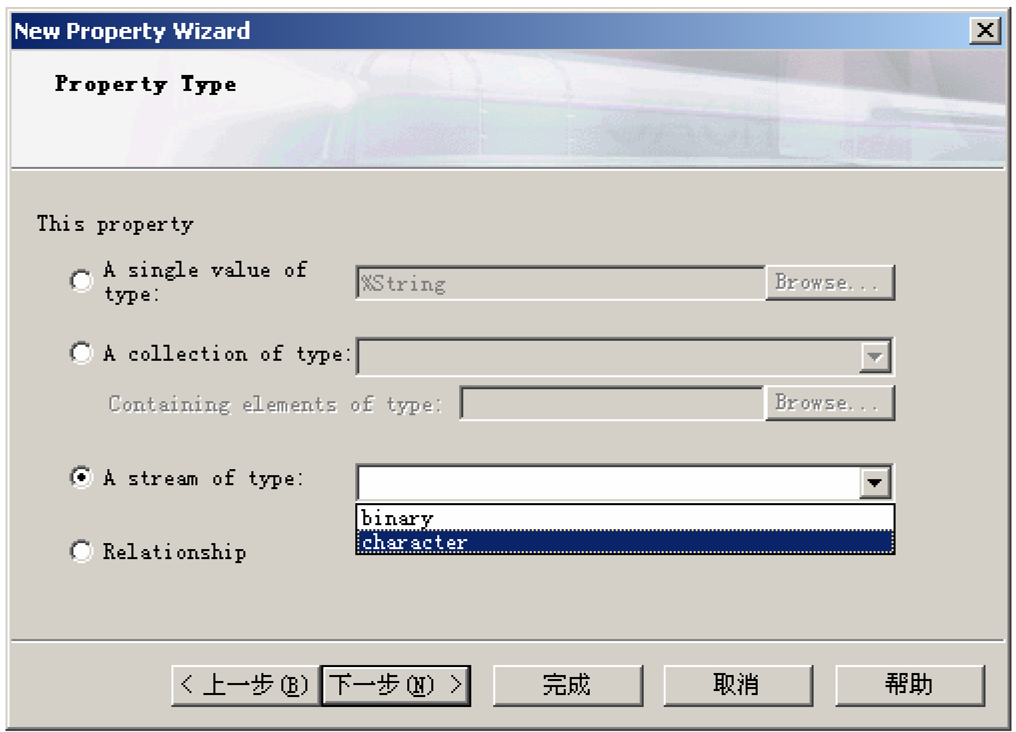
首先打开 Caché Studio，切换到"SAMPLES"的命名空间，再打开 Sample.Person 这个类。



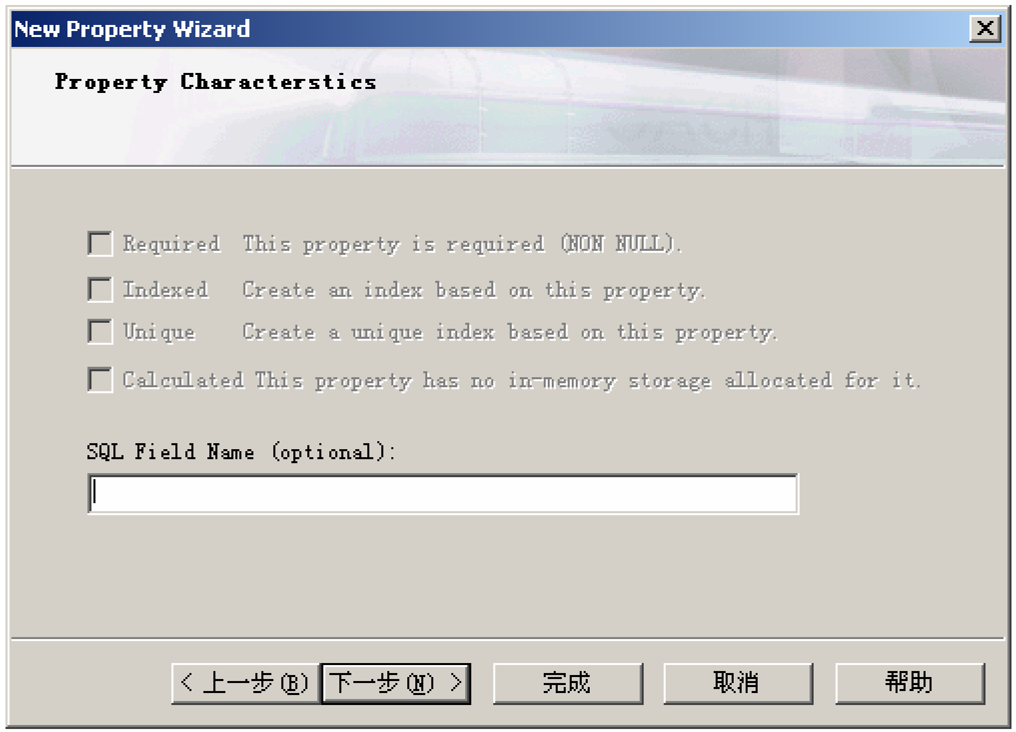
然后点添加属性用的按钮。



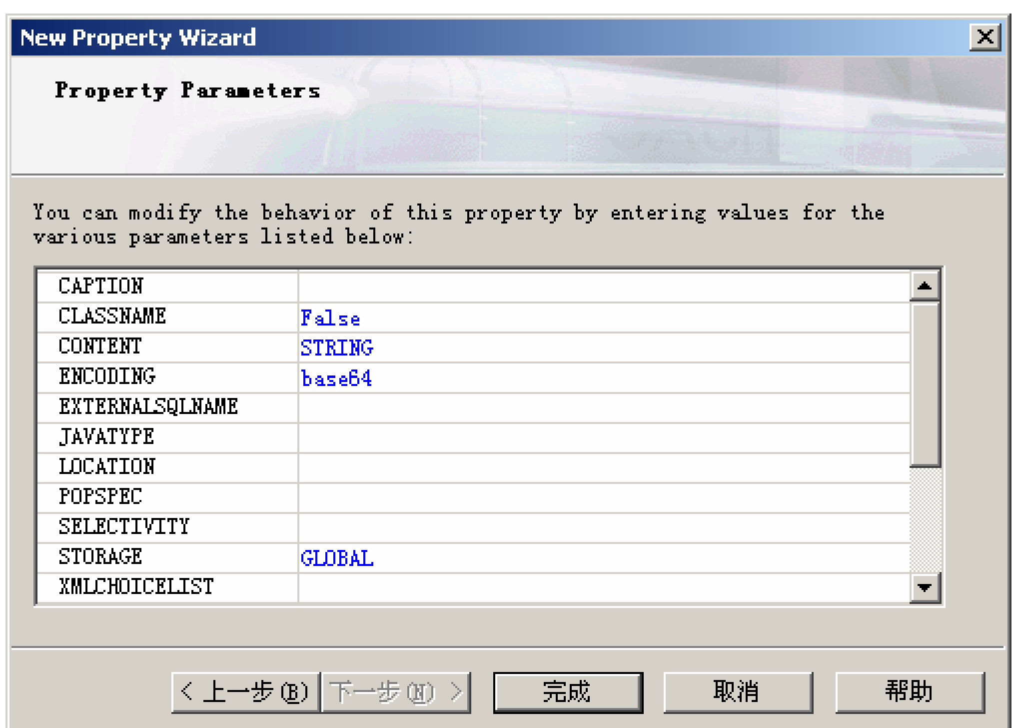
输入 Resume，然后点下一步按钮。



选择属性类型为 Character Stream，然后点下一步按钮。



点下一步按钮。



点完成按钮。你会在类定义中看到如下代码：

```
Property Resume As %Stream [ Collection = characterstream ];
```

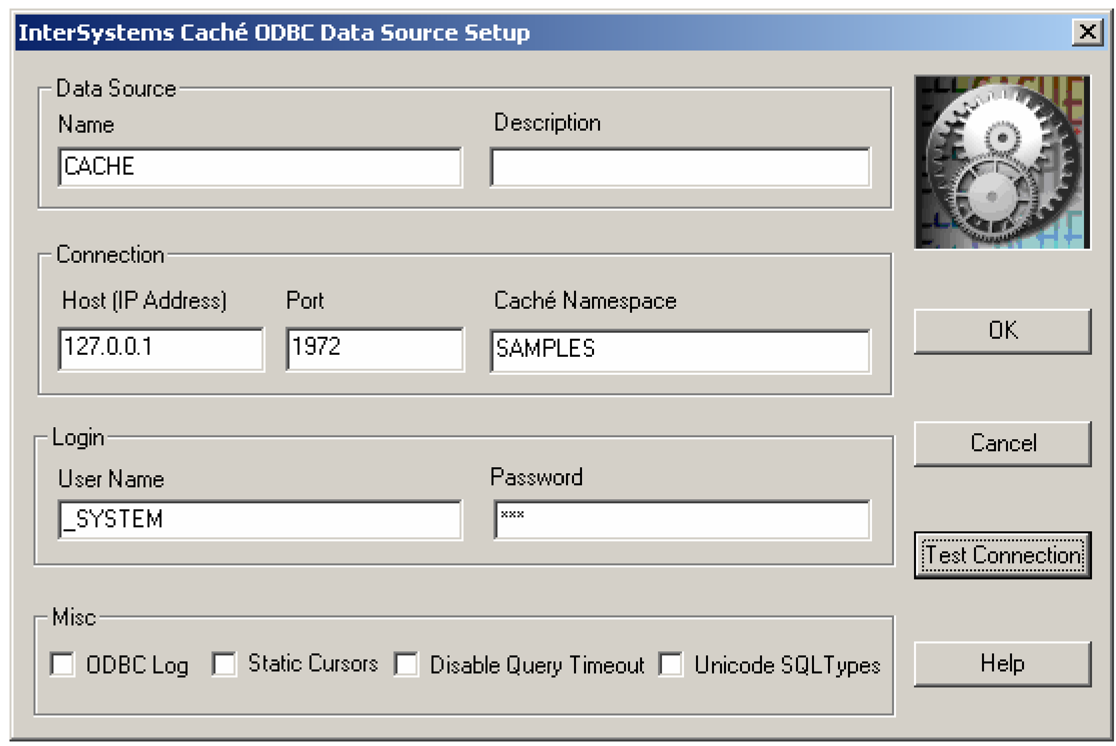
1.3　为客户端配置 ODBC

Caché 为 Win32 提供了 ODBC 支持，你可以在安装了 Caché 客户端或者 Caché 服务器的 Windows 机器上设置 ODBC 访问。注意：通过 **ODBC** 访问 **Caché** 是将使用 **SQL** 来访问的。

进入控制面板里面的管理工具，运行数据源（**ODBC**）。

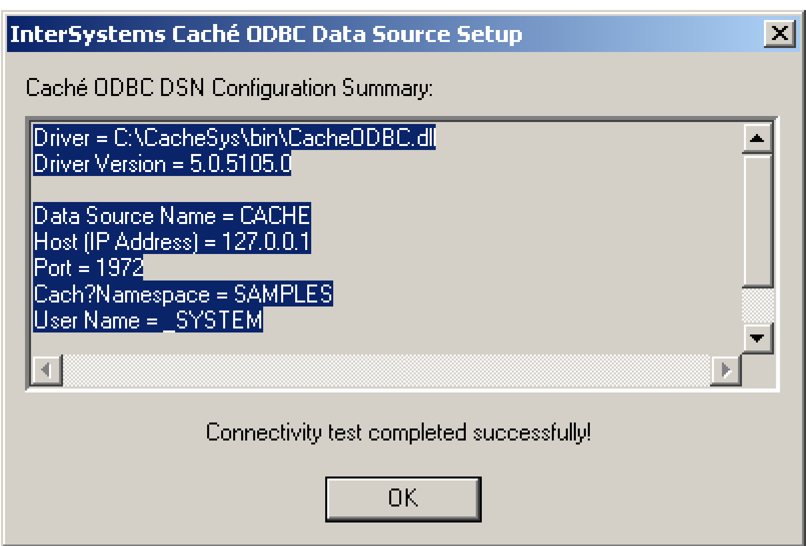
点添加按钮。

选择 **Intersystems ODBC**，点完成按钮。



数据源名称随意填写，这里我写的是“CACHE”,主机的 IP 地址根据实际情况填写，如果你没有多机版的许可，并且你是在 Caché 服务器上配置 ODBC 的话，这里就添“127.0.0.1”，端口号是“1972”，Caché 的命名空

间我们要连接到“SAMPLES”，用户名是“_SYSTEM”，密码是“sys”。点 Test Connection 按钮。



这意味着我们已经成功建立了到 Caché 的连接。

2 如何设计系统架构

当我们设计基于 Caché 数据库的系统时，首先要进行系统架构的设计。Caché 具备兼容各种主流技术的能力，因此选择合适的技术方案对开发人员来说尤其重要。

新概念：**Caché** 的开发观念，面向服务架构；

2.1 Caché 的开发观念

Caché 自身的特点改变了传统的开发观念，我们要了解这些特点，以便于选择最合适的技术方案。

1) Caché 后关系型数据库的服务器后端可以是数据库服务器和应用服务器的集合。

Caché 是面向对象的多维数据库，同时它也通过它独特的统一数据结构和 SQL 访问支持类似于关系型数据库的技术。在 Caché 中编写代码时不需要作对象-关系的转换或映射，开发人员可以专注于相关领域的业务逻辑开发。Caché 不仅能实现数据的持久化存储，而且可以封装业务逻辑，并通过面向对象技术将这两者结合在一起。而且，Caché 提供了优化措施(ECP 网络)来保证数据库和应用服务器之间的联系更加紧密；

** 通过 ECP(Enterprise Cache Protocol，企业缓存协议)可以在内存、应用服务器和数据库服务器上等地点自动设置高速缓存，并且在多个应用服务器和数据服务器之间形成一个可相互共享的网络，提高了整个系统的性能(Performance)和可伸缩性(Scalability)。相关的详细信息可参看联机文档 http://127.0.0.1:1972/csp/docbook/DocBook.UI.Page.cls?KEY=GDDM_intro*

2) Caché 支持面向服务的架构

由于 Caché 紧密结合了数据和逻辑，而这两者结合到一起就成为服务(Service)，可以说客户端程序访问的是 Caché 提供的服务，这样的架构称作面向服务的架构(Service-Oriented Architecture,SOA)。同样的服务可以被任何形式的客户所访问，无论它是基于 Windows Form，Java UI，浏览器或 Web Service 客户端。所以，最好的基于 Caché 的应用架构中将会用 Caché 构建整个服务(包括数据存储和业务逻辑)，而不仅仅是保存数据。这样不仅会带来性能的提升，还可使整个服务具备客户端无关性，即可以用各种客户端技术访问 Caché 服务，而服务器端不需要因为客户端类型变化而作改变。

** 我们通过 SQL 访问完全可以把 Caché 只当作像普通的关系型数据库那样来使用，但这样做显然并不能完全体现 Caché 性能的真正能力。*

3) 利用 COS 开发业务逻辑

Caché 提供有与建模工具(Rose)的接口，因而我们可以直接用 UML 设计具备持久化能力和包含业务逻辑的对象模型集合，而且可以在 Caché Studio 工作室中完成业务逻辑程序代码的编写。

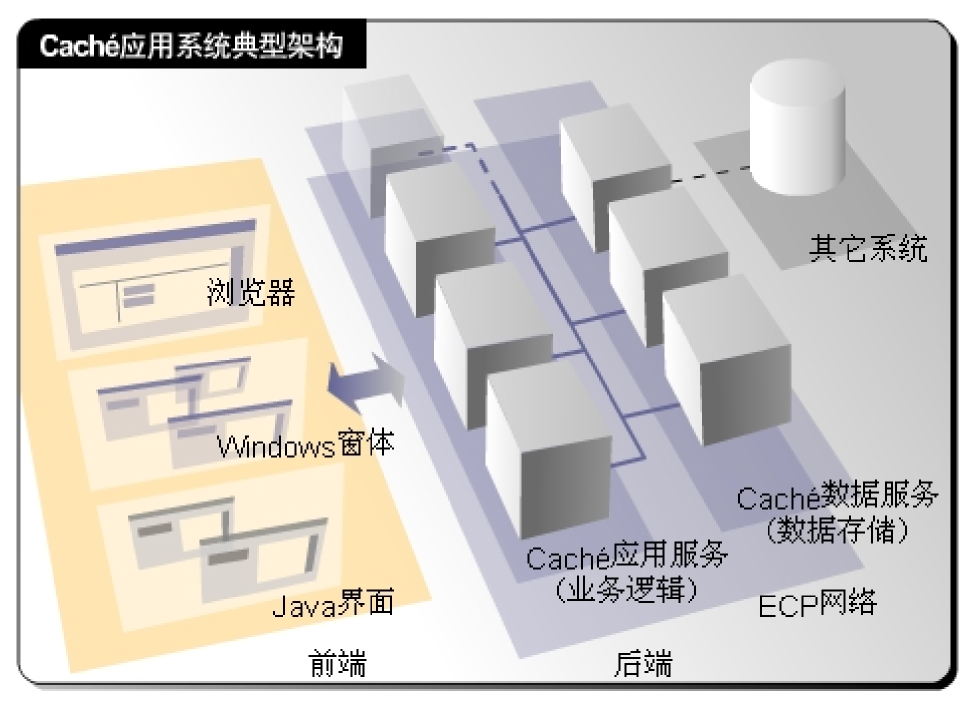
开发人员选择通过 Caché Object Script (COS)语言编写业务逻辑，因为 COS 可以同时拥有面向对象、SQL 和多维数组三种数据访问方式的优点：

面向对象	实现了结构化访问，信息封装，类的继承，多态
SQL	结合已优化过的和嵌入式的 SQL 能快速批量存取数据
多维数组	直接访问底层，树状结构，速度极快

在开发过程中，我们可以根据实际情况灵活掌握究竟选用哪种访问方式。

2.2 基于 Caché 的应用系统的典型架构

下图列出了在 Caché 上开发应用系统的典型的基础架构。



参照上图，基于 Caché 的应用系统应以如下方式实现：

2.2.1 系统整体架构方面：

在后端(服务器端)，通过 Caché 处理业务逻辑(通过 COS)和数据存储，并通过 ECP 网络紧密连接。后端设计人员不用考虑前端的实现方式。

在前端(客户端)，采用各种客户端技术来实现每个客户节点的用户界面显示和界面逻辑，设计人员不必考虑后端的设计，也不必考虑系统部署时的规模如何。

2.2.2 前端技术方面：

Caché 对于前端开发提供了很广泛的选择性，所有主流的前端开发技术都可以连接到 Caché 上来。不同的前端技术连接 Caché 的方式各不相同。目前有以下几种主要的方式：

2.2.2.1 基于浏览器的客户端。(B/S 架构)

CSP(Caché Server Pages)技术。

CSP 是一种服务器端的脚本技术，是 Caché 自身提供的前端实现方式。它工作在服务器端，使用它进行开发可以达到与 Caché 服务器端最紧密的配合，并提供了许多其它服务器端脚本技术没有的特性。

J2EE 和 **EJB** 技术。

Caché 的类可以映射成 BMP 型的 EJB，使得 EJB 在持久化时不必进行 O-R 转换。这样 Caché 可以为 J2EE 客户端提供服务。

其它服务器端脚本。

JSP、ASP、PHP 等可以通过关系型的方式访问 Caché 数据库，用法和其它关系型数据库相同。

2.2.2.2 基于 Windows 窗体的客户端。(C/S 架构)

COM 技术。

客户端程序通过 Caché 提供的 COM 组件连接到 Caché 服务，可以获得对象和关系型的访问能力。这种情况下客户端可以使用任何支持 COM 的开发工具进行开发，如 VB，

Delphi，PB，.Net 等，但客户端只能工作于 Windows 平台。

Web 服务。

Caché 服务可以轻松包装成为高效的 Web 服务，在各种支持 Web 服务的客户端中运行。这依赖于 Caché 天然的对象和 XML 支持能力。

.Net 技术。*(即将全面推出)*

Caché 的类文件可以映射成.Net 中的托管对象，这样，.Net 客户端程序将可以无缝地连接到 Caché 服务，获得对象和关系型的访问能力。

C++绑定。

客户端程序通过 Caché 提供的 C++库文件连接到 Caché 服务，可以获得对象(Caché 对象可映射成 C++的类文件和头文件)和关系型的访问能力。通过 VC 或 C++ Builder 进行开发。

关系型技术。

各种支持 ODBC 的客户端程序都可以通过 SQL 语句查询以类似于关系型的方式访问 Caché，用法和其它关系型数据库相同。

2.2.2.3　基于 Java 界面技术(SWING)的客户端。(C/S 架构)

Java 绑定。

客户端程序通过 Caché 提供的 Java 库文件连接到 Caché 服务，可以获得对象(Caché 对象可映射成 Java 的类文件)和关系型的访问能力。通过各种 Java 开发平台进行开发。

关系型技术。

各种支持 JDBC 的客户端程序都可以通过 SQL 语句查询以类似于关系型的方式访问 Caché，用法和其它关系型数据库相同。

2.2.3 系统交互和系统集成方面：

当面对与其它系统(旧有的或其它部门的基于关系型数据库的系统)的交互时，我们既可以通过 Caché 方便地将其它系统的数据和业务逻辑一次性导入到 Caché 中；也可以通过 Caché SQL Gateway 与之建立实时连接，此时 Caché 会将该系统的表结构和存储过程等信息自动转化成 Caché 类，数据仍存储在原来系统内，这样 Caché 的客户端就可以获得集成的服务；

** 详细内容请参看 Caché SQL Gateway，位于联机文档：*
http://127.0.0.1:1972/csp/docbook/DocBook.UI.Page.cls?KEY=GSQG

第五章 以 Java 开发 Caché 应用程序

1 引言

用 Java 开发 Caché 应用程序对于原来熟悉 Java 编程的程序员来说是一件十分轻松的事情，Caché 为 Java 提供了多种选择。你能够以下面几种方式创建 Java 在 Caché 数据库方面的应用：

- 绑定方式

  Caché 与 Java 的绑定使得 Java 对象直接同 Caché 服务器对话。Java 绑定自动为每个指定的 Caché 类创建一个在 Java 环境中运行的代理类。代理类是一个纯 Java 类，它只包含标准的 Java 代码，提供了对映射的 Caché 类的属性和方法的访问。

- **JDBC** 驱动方式

  Caché 包含了一个第四类的（纯 Java 的）JDBC 驱动，这个驱动支持 JDBC 2.0 API. Caché 的 JDBC 驱动提供了高性能的以关系的方式访问 Caché 的方式。

- **EJB** 绑定

  Caché 的 EJB 绑定使 EJB (Enterprise Java Bean) 直接和 Caché 服务器上的对象通话。Caché 的 EJB 绑定自动为每个指定的 Caché 类生成一个 EJB 实体 Bean，这个 Bean 可以在 Caché 数据库中使用一种有效的，自动生成的，对 Bean 管理的持久的接口来保存和载入它自己。

1.1 Caché 与 Java 的绑定

Caché 与 Java 的绑定提供了一个在 Java 应用中能以简单的、直接的方式使用 Caché 对象的方式。

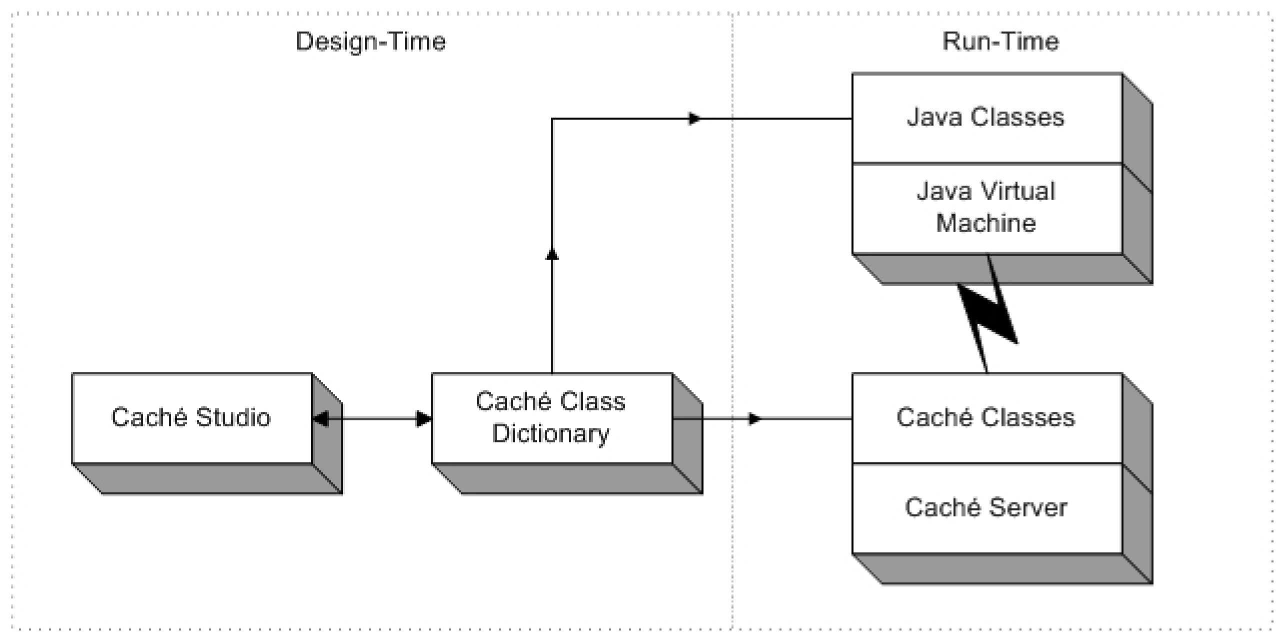
1.1.1 Caché 与 Java 的绑定的机制

Caché 与 Java 的绑定提供了一个访问 Caché 服务器上的对象的方法。这些对象可以是存储在 Caché 服务器上的持久对象，也可以是在 Caché 服务器上执行一些操作的临时对象。

Caché 的 Java 绑定由下列组件构成：

- Caché-Java 类生成器—一个扩展 Caché 类编译器从在 Caché 类字典中定义的类中生成纯 Java 类。
- Caché-Java 类包—这个包是纯 Java 类的包，它同由 Caché-Java 类生成器生成的 Java 类，并且向它们提供到 Caché 数据库中存储的对象的透明的连接的类协同工作。
- Caché 对象服务器—一个管理使用 TCP 协议的 Java 客户端和 Caché 数据库服务器之间通信的高性能的服务器进程。注意：Java 对象、EJB 和 JDBC 访问可以使用一个公共的服务器。

Caché 类编译器可以自动为 Caché 类目录中的任何类创建 Java 客户端类。这些生成的 Java 客户端的类在运行时同它们在 Caché 服务器上的对应的 Caché 类联系。它们只包含纯的 Java 代码，并且自动同主类的定义保持同步。如下图所示：



基本的工作机制如下：

1. 你在 Caché 中定义一个或者多个类。这些类可以表现为持久对象保存在 Caché 数据库中，或者是运行在 Caché 服务器上的临时对象。
2. Caché 生成对应于你的 Caché 类的 Java 类。这些 Java 类包含对应的服务器上的 Caché 方法（get 和 set）来访问对象的属性。
3. 在运行时，你的 Java 应用连接到 Caché 服务器。它然后创建一个对应于 Caché 服务器上的对象的 Java 对象实例。你可以象是用其它任何 Java 对象一样使用这些对象。Caché 自动管理所有的通信，就像客户端的数据缓冲一样。

运行时的架构由下列组成：

- Caché 服务器（一个或多个）
- 在你的应用运行的机器上有一个 Java 虚拟机。Caché 不提供 JVM，但是它需要和一个标准的 JVM 一同工作。

- Java 的应用程序（包括 servlet、applet 或者基于 Swing 的应用程序）。

在运行期的时候，Java 应用既可以使用对象连接借口也可以使用标准的 JDBC 接口连接 Caché。所有的 Java 应用程序和 Caché 服务器之间的通信都使用 TCP/IP 协议。

1.1.2 Caché 与 Java 的绑定的配置

所有的使用 Caché 与 Java 绑定的应用程序都分成了两个部分：Caché 服务器和 Java 客户端。Caché 服务器负责数据库操作和执行 Caché 对象方法。Java 客户端负责执行所有的 Java 代码（例如附加的业务逻辑或者用户界面）。当一个应用程序运行的时候，Java 客户端通过 TCP/IP 连接 Caché 服务器。实际的开发配置由应用程序开发者决定：Java 客户端和 Caché 服务器可以存在于同一台物理机器上也可以存在于不同的机器上。Java 客户端可是一个简单的 JVM,或者是一个 EJB 的服务器。

Caché 需要 Java 的客户端使用 Java SDK1.3 或者更高版本。

连接到 Caché 需要的 Java 组件包含在文件 *CacheDB.jar*。对于 Windows 系统，这个文件在：*<cache_root>\Dev\java\lib\CacheDB.jar*；如果你在 Windwos 的安装中接受了默认的安装，这个文件在：*C:\CacheSys\Dev\java\lib\CacheDB.jar*。在 UNIX 的版本中，默认的 *<cache_root>* 是 */usr/cachesys*.

Java 环境使用系统环境变量 CLASSPATH 来查找它需要的文件。它必须包含文件 *CacheDB.jar* 和 *jdbc2_0-stdext.jar* 所在的路径。

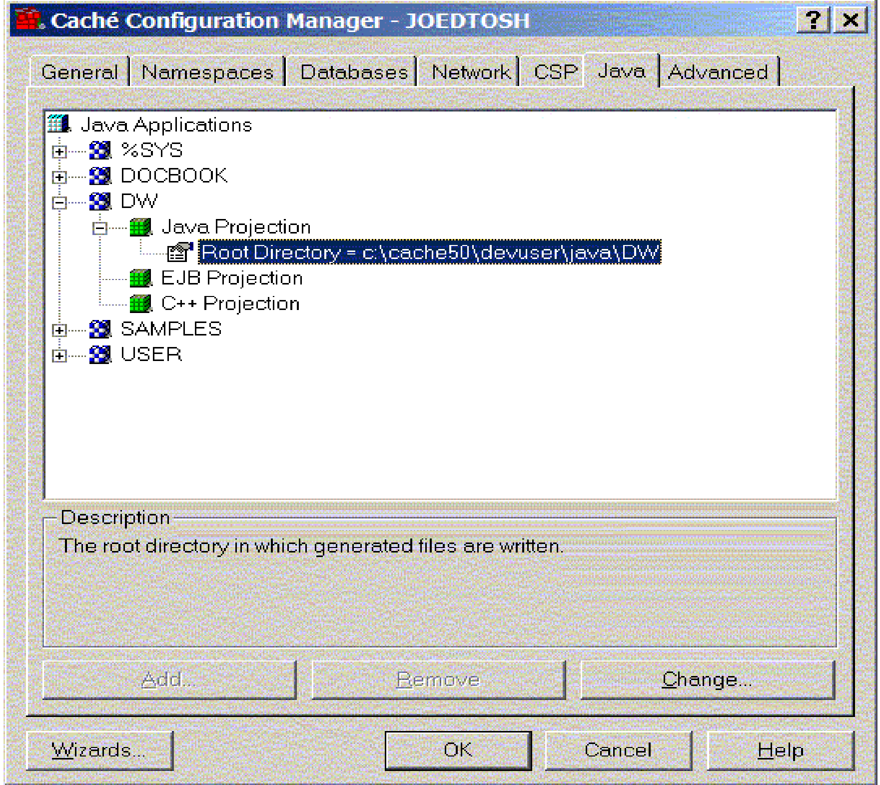
如果你使用的 Java SDK 版本是 1.3.x，*jdbc2_0-stdext.jar* 必须从 Sun 的网站上下载，http://java.sun.com/products/jdbc/download.html。选择连接 "JDBC 2.0 Optional Package API" 然后点 **Continue** 按钮连接到"JDBC 2.0 Optional Package Binary").

如果你使用 Java SDK 1.4.x，这个包已经在标准安装中了，不用再下载了。

1.1.3 配置服务器

Caché 与 Java 的绑定自动生成 Java 类对应于 Caché 中定义的类。你能通过使用 Caché Configuration Manager 中的 Java 活页夹控制生成的 Java 类的位置。

Java Application Configuration



对于每一个命名空间（namespace），你都能够指定创建 Java 类的一个根目录。默认地，每个 namespace 都在*/cachesys/devuser/java/namespace* 目录下创建 Java 类，(“cachesys” 是 Caché 的安装所在的目录）。使用 Java Applications 树控制，你能为每个 namespace 改变根目录。

1.1.4 从 Caché 类生成 Java 文件

要从 Java 客户端使用 Caché 类，就要生成 Java 类来在客户端运行。

要生成 Java 类，如下做：

1. 在 Caché Studio 中，建立一个到类的 namespace 的连接然后打开类。例如，要使用 **Person** 类，连接到 SAMPLES namespace，在打开 Sample package 中的 **Person** 类。
2. 使用 Caché Studio 的 New Projection 向导给类添加一个 Java 映射定义：从 **Class** 菜单中的 **Add** 子菜单调用 **New Projection**。
3. 选择映射的名字以后，指定它的类型为 **%Projection.Java** 。
4. *ROOTDIR* 参数是用来指定保存生成的 Java 类的目录的。
5. 在 New Projection 向导的最后一个屏幕，点 **Finish**。在这一步，向导向你的类定义添加了类似如下的代码：

```
Projection MyProjection As %Projection.Java(ROOTDIR="C:\temp\");
```

注意，如果你忽略了 *ROOTDIR* 参数，你的 Java 类将被创建在 Caché 配置的默认目录下（你可以使用 Configuration Manager 中的 Java 活页夹来设置它）。

6. 编译 Caché 类。这将生成类的 Java 映射，不论编译发生在 Studio 中还是在 Caché 命令行下（终端模式）。 这时你就可以得到这个 Caché 类的 Java 的代理类。这个 Java 的类被放在了 Caché 配置的默认目录下或

者你指定的目录下，当然，还要考虑到你的 Caché 类的包的信息，我们将在后面详细讨论这个内容。

编译 Java 类。你既可以使用 **javac** 命令，也可以使用任何其它的工具来编译 Java 类。

1.1.4.1 例子

Caché 提供了两个 Java 应用的例子来示范使用 Caché 与 Java 的绑定。它们位于*<cache_root>/dev/java/samples/* 目录。

第一个例子，*SampleApplication.java*，是一个简单的连接到 Caché SAMPLES 数据库，打开和修改保存在数据库中的 **Sample.Person** 对象的实例，执行一个预定义的数据查询的例子。这个程序通过命令行来调用，从命令行读取对象 id 的值，然后使用 Java *system.out* 对象输出。这个例子假设 Caché 和 Java 运行在 Windows 上。

第二个例子，*JDBCQuery.java*，使用 JDBC 建立从客户端到服务器的连接。然后它执行一个 SQL 查询。同第一个例子一样，它是从命令行调用的，假定 Caché 和 Java 都运行在 Windows 上。

这些示例程序都将使用 SAMPLES 命名空间中的 **Sample.Person** 类。这个 Caché 类已经映射到 Java 上，*<cache_root>/dev/java/samples/Sample/Person.java*。对于你的程序，你需要映射 Caché 类到 Java 类。

这里是示例程序的已完成的 Java 源码：

```
import java.io.*;
import java.util.*;
import com.intersys.objects.*;
public class SampleApplication {
  public static void main(String[] args){
    Database    dbconnection = null;
    String       url="jdbc:Cache://localhost:1972/SAMPLES";
```

```
String       username="_SYSTEM";
String       password="sys";
ObjectServerInfo    info = null;
Sample.Person    person = null;
    try {
  //通过 CacheDatabase 的静态方法 getDatabase 取得连接
      //url 是连接字符串，username 是用户名，password 是密码
  dbconnection = CacheDatabase.getDatabase (url, username, password);
  //打开一个 Sample.Person 类的实例,
  //ID 是从控制台读取的
  InputStreamReader isr = new InputStreamReader(System.in);
  BufferedReader br = new BufferedReader(isr);
  System.out.print("Enter ID of Person object to be opened:");
  String strID = br.readLine();

  // Use the entered strID as an Id and use that Id to
  // to open a Person object with the _open() method inherited
  // from the Persistent class. Since the _open() method returns
  // an instance of Persistent, narrow it to a Person by casting.
  person = (Sample.Person)Sample.Person._open(dbconnection, new Id(strID));

  // Fetch some properties of this object
  System.out.println("Name: " + person.getName());
  System.out.println("City: " + person.getHome().getCity());

  // Modify some properties
  person.getHome().setCity("Ulan Bator");

  // Save the object to the database
  person._save();

  // Report the new residence of this person */
  System.out.println( "New City: " + person.getHome().getCity());
```

```
    /* de-assign the person object */
    dbconnection.closeObject(person.getOref());
    person = null;

    // Close the connection
    dbconnection.close();

    } catch (Exception ex) {
    System.out.println("Caught exception: "
      + ex.getClass().getName()
      + ": " + ex.getMessage());
   }
  }
}
```

编译和执行

在*<cache_root>/dev/Java/Samples* 这个目录中，*SampleApplication.java* 文件有这个程序的拷贝。

在 *Samples* 目录中，编译程序：

```
javac SampleApplication.java
```

然后运行它：

```
java SampleApplication
```

执行完毕，程序将产生如下结果：

```
C:\java> java SampleApplication
Enter ID of Sample.Person object to be opened: 1
Name: Isaacs,Sophia R.
City: Tampa
New City: Ulan Bator
```

程序的动作

- 建立到 Caché 服务器的连接
- 询问要打开的 **Person** 对象的 ID。
- 从 Caché 数据库中打开对象。
- 显示 *Name* 和 *City* 属性的值。
- 为 *City* 属性设置一个新值。
- 把编辑过的对象存入数据库。

在 Java 的 import 语句和声明以后，有一句尝试连接到本地 Caché 服务器的语句：

```
String url="jdbc:Cache://localhost:1972/SAMPLES";
String username="_SYSTEM";
String password="sys";
//...
dbconnection = CacheDatabase.getDatabase (url, username, password);
```

这段代码建立了到 Caché 数据库的连接。**CacheDatabase** 是一个含有一个静态方法—**getDatabase** 的类，它建立从 Java 客户端到 Caché 服务器端的 TCP/IP 连接。它有三个参数，第一个包含了一个 IP 地址（此处是本地地址 127.0.0.1）以及一个端口号（此处是默认的 1972）还有一个 namespace（此处是 SAMPLES），第二个和第三个参数分别是用于登录 Caché 服务器的用户名和密码。它返回 **Database** 对象，这里是 *dbconnection*。

下一步，程序使用标准的 Java 功能来提示输入用户的 ID 以打开对象和取值。一旦得到 ID，下一步就可以打开指定的对象了：

```
person = (Sample.Person)Sample.Person._open(dbconnection, new Id(strID));
```

这段代码调用 Sample 包里面的 **Person** 对象的**_open** 方法，这个方法有两个参数：第一个是包含要打开的对象的数据库，第二个是要打开的对象的 ID。返回值和 **Sample.Person** 的实例吻合，因为**_open** 继承自 **Persistent** 类，因此返回的就这这个类的一个实例。

一旦这个对象打开了，程序就显示这个对象 *Name* 属性的值。

```
System.out.println("Name: " + person.getName());
```

注意，与 Caché 服务器上直接引用属性不同，Java 中引用对象的属性是通过 get 和 set 方法的。

下一步，显示属性 *City* 的值，然后给它一个新的值：

```
System.out.println("City: " + person.getHome().getCity());

person.getHome().setCity("Ulan Bator");
```

这几行操作 *City* 属性的代码演示了查看和修改嵌入对象的属性。如果一个属性是对象的话（就像 *Home* 属性），那么它就有它自己的属性（比如 *City* 属性）和附属的方法。你可以使用点句法调用这些方法。

设置 *City* 属性新值以后，程序就进行保存对象，显示值，关闭对象，取消定义等操作然后关闭自己。

```
person._save();
// Report the new residence of this person */
System.out.println( "New City: " + person.getHome().getCity());
// * de-assign the person object */
dbconnection.closeObject(person.getOref());
person = null;
// Close the connection
factory.close();
```

注意，**Java** 执行垃圾回收，所以不需要你关闭任何对象。然而，**InterSystems** 建议你关闭所有的对象，以确保完整性。

1.1.5 映射的细节

下面我们讨论 Caché 到 Java 映射的一些细节问题，比如包名，类名，属性和方法等信息。

1.1.5.1 包

通常地，Caché 包的名字也作为 Java 包的名字。如果 Caché 类定义了一个类参数——*JAVAPACKAGE* 那么 Java 生成器就会使用参数的值来作为包的名字。在包中的"%"被翻译成 "_"。

注意，在你的代码中，Caché 和 Java 包的名字必须彼此保持一致。

唯一的例外是**%Library** 包，它变成了"com.intersys.objects"。因此 **%Library.Persistent** 类就成为"com.intersys.objects.Persistent"了。这就是为什么导出的 Java 文件包含语句：

```
import com.intersys.objects.*;
```

它包含了 Caché %Library 包里面的所有的类。

1.1.5.2 类

Caché 类的名称被原样地映射到 Java 里面，除非 Caché 的类名是 Java 的保留字，如果真的是这样的话，Caché 类名将被映射为"_"加上类名。

1.1.5.3 属性

你可以通过为属性生成的两个方法来引用 Caché 属性：**getProp** 和 **setProp**， "Prop"是属性的名字。如果属性名字以 "%"开头，它将被 "sys_"替换。因此，Color 属性的映射将包含 **getColor** 和 **setColor** 方法；%Concurrency 属性的映射将包含 **get_Concurrency** 和 **set_Concurrency** 方法。

例如，假设你定义了一个持久类，它包含了两个属性，一个是字符串类型的另外一个是对象类型的：

```
Class MyApp.Student Extends %Persistent [ClassType = persistent]
{

/// Student's name
Property Name As %String;
```

```
/// Reference to a school object
Property School As School;
}
```

生成的 Java 类 Name 和 School 两个属性都包含了 get 和 set 两个附属方法。另外，它还提供引用 School 对象的 ID 的附属方法。

```
public class Student extends Persistent {
  // ...
  public String getName() throws CacheException {
    // implementation...
  }
  public void setName(String value) throws CacheException {
    // implementation...
  }
  public School getSchool() throws CacheException {
    // implementation...
  }
  public void setSchool(School value) throws CacheException {
    // implementation...
  }
  public Id idgetSchool() throws CacheException {
    // implementation...
  }
  public void idsetSchool(Id value) throws CacheException {
    // implementation...
    }
}
```

当一个映射的 Java 对象在 Java 中被实例化的时候，它将从 Caché 服务器获取它的属性的副本，并且复制他们到本地的 Java 端的缓存中去。后来访问这个对象的属性的操作都是对这个缓存进行的，这将减少和服务器之间的消息传递。Caché 自动管理本地的缓存来保证它是和 Caché 服务器上的对象是同步的。

注意：你在你的 Caché 类中定义的 get 或者 set 方法（例如一个属性的值依赖于其它的属性）存取值的属性，它们不会存储在本地的缓存中。取而代之的是，当你操作这样的属性的时候，对应的附属方法在 Caché 服务器段被调用。由于这将增加网络的交通负担，所以请慎重使用这种方法。

1.1.5.4 方法

一般情况下，方法的名称是直接被映射的，不用修改，例外的情况是：

- 如果方法名以"%"开头，它将被替换为"sys_"。
- 如果方法的名字是 Java 的保留字， "_"将被加到名字的前面。
- %Library 包的部分类方法，开始的"%"被 "_"替换，并且首字母被转换成小写的。

Caché 类里面的方法作为对应的 Java 类的桩方法被映射到 Java。实例方法作为 Java 的实例方法被映射，类方法作为 Java 的静态方法被映射。当调用一个客户端方法的时候，实际上是调用在 Caché 服务器上实现的真正的方法。

如果一个方法的声明中包含了带有默认值的参数，Caché 就用不同的参数产生多个方法的版本来模拟在 Java 中的默认的参数值。

例如，假设你定义了一个简单的 Caché 类，它有一个下面的方法：

```
Class MyApp.Simple Extends %RegisteredObject
{
Method LookupName(id As %String) As %String
{
   // lookup a name using embedded SQL
   Set name = ""
   &sql(SELECT Name INTO :name FROM Person WHERE ID = :id)
   Quit name
   }
}
```

产生的 Java 类将看起来像：

```
public class Simple extends Object {
   //...
   public String LookupName(String id) throws CacheException {
      // generated code to invoke method remotely...
      // ...
      return typedvalue;
   }
}
```

当从 Java 中调用一个映射了的方法，Java 客户端首先同步服务器的对象缓存，然后调用 Caché 服务器上的方法，最后返回结果（如果有的话）。如果指定的任何的参数为引用的方式，那么它们的值也同时更新。

除了 Caché 类定义的方法外，映射的 Java 类还包括一些自动生成的系统方法来执行许多操作：

_close	关闭服务器上的一个对象（通过调用对象的%Close 方法）。
_open	对于持久的对象，通过实例的 OID 作为句柄打开一个存储在数据库中的对象的实例
_openId	对于持久对象，使用实例的类 ID 的值作为句柄打开存储在数据库中的对象的实例。

1.1.5.5　　数据类型

Caché 使用了多种数据类型 (例如 strings 或者 numbers)，这些类型可以用作属性、方法的返回类型和反法的参数。每个 Caché 的数据类型都有一个对应的 Java 客户端的数据类型；这个客户端的数据类型指定了 Java 类哪个变量被映射。因此，使用客户端数据类型，每个 Caché 的数据类型都被指定了使用一个相应的 Java 对象，例如 **Integer** 或者 **String** 。

不管属性是什么类型，如果它没有被设值，Java 都用 null 关键字来设置它的值。例如，假设你创建了一个新的对象，这个对象有个 **Integer** 的属性 age，那么先前的调用 **getAge** 方法返回的就是 null。

对于对象值的类型（引用其它对象实例）表现为使用相应的 Java 类。某些特定的 Caché 对象要特殊对待，例如，流对应为 Java 的流对象，而集合类型对应于 Java 的集合对象。

1.1.5.6 形式变量名

如果方法参数中有以"%"开头的，"%"将被替换为"_"，变量的名字是 Java 的保留字，"_"将被加到名字的前面。

1.1.5.7 集合

Caché 支持两种类型的集合：列表和数组。这是两种不同的分组方式。列表维持的是有序的元素，而数组是一些无序的元素组成的成对的名字和值的条目。

- 列表是有序的信息的集合，每个列表的元素通过它在列表中的位置来识别。你可以设置某个位置的数据的值或者在一个位置插入一个数据。如果你给一个位置设置了新的值，这个值就保存在列表中了。如果你给一个已经存在的位置设置值，那么新的值就会覆盖原先的值，但是位置没有变化。如果你在一个已经存在的位置插入数据，新的列表条目数量将增加，在第二的位置插入数据，那么原来第二的位置的数据就变成了第三位的，原来第三的对象就成了第四个，以此类推。这里有列表的数据类型和列表的数据对象。
- 数组是无序的信息的集合（不像列表那样是有序的）。一个数组包含了一个或者多个名字-值的对子，这里元素的名字作为主键，值是和

主键关联的数据。添加一个新元素和更改已经存在的元素的数据在这里没有语法上的差别。这里有数组的数据类型和数组的对象。

Java 客户端的操作通常都假定集合是已经存在的。

1.1.5.8 列表方法

列表由一连串的独立的条目组成，每个条目由一个主键（一个表示该条目在列表中的位置的整数）和一个元素的值组成。列表维护它的主键所以最大的主键的值就是列表中条目的数量。你可以在列表的尾部添加条目也可以在中间插入条目（将导致这个条目往后的所有的条目的主键的值都增加）。

支持的操作包括：

- 在指定的位置插入一个元素: **_insertAt(key, element)**
- 在列表的尾部添加一个元素: **_insert(element)**
- 取得一个元素的值: **_getAt(key)**
- 修改一个元素的值: **_setAt(key, element)**
- 删除一个元素: **_removeAt(key)**
- 清除列表的数据: **_clear()**
- 计算元素的数量: **_count()**

导航的动作包括：

- 取得下一个元素：**_next(key)**
- 取得上一个元素：**_previous(key)**
- 转到下一个元素：**_getNext(keyHolder)**
- 转到上一个元素：**_getPrevious(keyHolder)**

1.1.5.9　数组方法

数组由一组无序的条目组成，每个条目包含一个主键（字符类型）和一个元素的值（自定义的值）。

注意：

数组不同于列表，因为它们没有固定的顺序。因此，如果你试图用一个已经使用的主键名字添加一个条目，_setAt 方法将覆盖这个条目原来的值。这点和列表不同，列表将为已经存在的条目增加主键的值（和下面的条目）。

支持的对数组元素的操作包括：

- 添加一个元素：**_setAt(key, element)**
- 取得一个元素的值：**_getAt(key)**
- 修改一个元素的值：**_setAt(key, element)**
- 删除一个元素：**_removeAt(key)**
- 检查一个元素是否存在：**_isDefined(key)**
- 清除数组的数据：**_clear()**
- 计算元素的数量：**_count()**

导航的操作包括：

- 取得下一个元素：**_next(key)**
- 取得上一个元素：**_previous(key)**
- 转到下一个元素：**_getNext(keyHolder)**
- 取得上一个元素：**_getPrevious(keyHolder)**

1.1.5.10　流（Stream）

Caché 允许你创建大字符序列的属性，即可以是字符也可以是二进制的格式；这些序列叫做“流”。字符流是长的文本序列，例如：在数据录入界面中

的自由格式的文本框中的内容；二进制流通常是图片或者是声音文件，它类似其他数据库中的 BLOB 字段。

当你要向流中写或从流中读的时候，Caché 监控你在流中的位置，于是你可以向前或者向后移动。

- 添加内容：**_writeLine** (character streams only), **_write**
- 读取内容：**_readLine** (character streams only), **_read**
- 转到尾部：**_moveToEnd**
- 转到头部：**_rewind**
- 检查当前的位置是否在流的尾部：**atEnd**
- 取得流的大小：**_sizeGet**
- 检查流是否为空：**isNull**
- 清除流的内容：**_clear**

1.1.5.11　类查询

Caché 允许你定义查询为类的一部分。这些查询被编译并且能够在运行期被调用。

要运行预定义的查询，需要使用 CacheQuery 类：

1. 建立到 Caché 服务器的连接。
2. 使用下面的代码创建 CacheQuery 对象的实例：

   ```
   myQuery = new CacheQuery(factory, classname, queryname);
   ```

   factory 指定已经存在的连接， *classname* 指定要运行查询的类，*queryname* 指定预定义的查询的名字。

3. 一旦你实例化了 CacheQuery 对象，你就能调用预定的查询：

   ```
   java.sql.Result ResultSet = myQuery.execute(parm1);
   ```

这个方法接受 3 个参数，它直接传递给查询；如果查询接受 4 个或者更多的参数，你可以在参数数组中传递。这个方法返回标准的 JDBC ResultSet 对象的实例。

1.2 Caché 的 JDBC 驱动

很多 Java 程序员对采用 JDBC 在传统的关系数据库环境下编写应用程序都已经很熟悉了，Caché 也提供了这种方式。区别于 Caché 与 Java 绑定的方式，JDBC 驱动提供了对 Caché 数据库的一种关系的访问方法。程序员不用对 Caché 有更多的了解就可以使用 JDBC 编写应用程序了。这时候 Caché 和其它的数据库的访问方式没有差别。

使用 JDBC 来查询数据，我们只要按照下面的做就可以了：

1. 使用 JDBC 连接对象连接到 Caché，代码如下：

```
2. String user = "_SYSTEM";
3. String password = "SYS";
4. String url = "jdbc:Cache://127.0.0.1:1972/SAMPLES";
5. //...
6. Class.forName ("com.intersys.jdbc.CacheDriver");
     Connection conn = DriverManager.getConnection(url, user, password);
```

这段代码初始了变量，载入了 JDBC 驱动，并且建立了到 Caché 的连接。

7. 指定你要运行的查询。这包括创建 statement 对象（使用新创建的 Connection 对象）并建立 statement 的内容：

```
8. Statement stmt = conn.createStatement();
```

```
9. java.sql.ResultSet rs = stmt.executeQuery(stQuery);
```

在上面的代码，stQuery 变量包含了一个合法的 SQL 查询

10. 执行 statement。

```
ResultSetMetaData rsmd = rs.getMetaData();
```

11. 使用 Java **ResultSet** 对象管理返回的数据。

```
12.         int colnum = rsmd.getColumnCount();
```

1.3 Caché 的 EJB 绑定

Caché 的 EJB 绑定允许你用 EJB 应用使用 Caché。Caché 的 EJB 绑定提供了下面的好处：

- 提高手写持久化代码的效率而不用一边写一边维护它——Caché 使用存储在 Caché 类目录里面的元数据为 EJB 实体 Bean 自动生成持久化代码。
- 没有对象-关系配合的错误——因为 Caché 数据库是按照对象定义的，所以不需要在 EJB 服务器中自定义对象关系影射。
- 高性能——除了 Caché 数据库提供的高性能，Caché EJB 提供了高效率的缓存以最大限度地降低 EJB 服务器和数据库服务器之间的网络负担。
- 快速地应用开发——Caché 避免了大量的 EJB 需要的工作。除了自动生成 EJB 实体 Bean，Caché 生成 EJB 部署描述符、要在 EJB 服务器上部署 Bean 的脚本和测试你的实体 Bean 的 Java 代码。
- 高适应性——Caché 让你在你的 EJB 应用中混合和匹配对数据库有效的关系和对象访问，让你不论选择哪个对手边的技术来说都是合适的技术。

Caché 的 EJB 服务器在下面的小节中有更详细的描述。本章假定你已经熟悉 Java 和 EJB 了。

1.3.1 EJB 是如何工作的

EJB 是一组基于 Java 的技术，它提供了使用表示层和业务逻辑层的 Java 组件开发 n 层、分布式应用的框架。EJB 提供了一个标准框架。这个框架管理所有的安全、客户端连接、
对象的生命周期和对应用的事务控制。

1.3.1.1　EJB 应用的架构

普通的 EJB 应用的架构包括三个部分（或层），即表示（presentation）层、业务逻辑（Business Logic）层、和数据库（Database）层，如下图所示。这些部分可以按照需要部署在不同的机器上，：

表示层负责用户界面。业务层负责业务逻辑。数据库层负责所有的数据存储和数据库操作。EJB 应用的中间层（业务逻辑层）被分成了两个不同的部分：

- EJB 服务器——市场上有许多现成的 EJB 服务器可以用。
- 用户定义的 Bean——一组用户提供的组件（Beans），它们既提供业务逻辑或者在应用的数据层中的表示条目。

1.3.1.2　Bean 的类型和持久性

在 EJB 里面有两种类型的用户定义的 Bean 或者组件：

- 实体 Bean

- 会话 Bean

对于每种类型的 Bean，EJB 服务器提供一个叫做容器的对象。容器对象封装了对实体 Bean 和会话 Bean 的操作以及其它的服务。EJB 不定义应用使用什么类型的数据库，它只定义实体 Bean 的接口。每个实体 Bean 都知道如何在数据库中维护它的内容。EJB 提供了两种形式的实体 Bean 的持久化：

- 容器管理的持久化（CMP）
- Bean 管理的持久化（BMP）

Caché 的 EJB 绑定提供了两者最好的：Caché 用 BMP 创建实体 Bean，它是自动生成的。你得到高性能的不用对象-关系映射和 SQL 查询的基于对象访问数据库中的对象。（注意，如果你喜欢，你也可以通过 Caché JDBC 驱动和 CMP 使用 Caché）。

1.3.1.3　支持的 EJB 服务器

Caché 支持下列的 EJB 服务器：

EJB Server	Versions
BEA WebLogic	7.0 (SP1)
JBoss + Tomcat	3.0.4, 3.0.6 + 4.1.12
Pramati	3.0 (SP4)

1.3.1.4　找出更多的关于 EJB 的信息

有许多关于 EJB 的网站和书，可以访问和参阅：

http://java.sun.com/products/ejb/index.html

Enterprise Java Beans by Richard Monson-Haefel, 2001, O'Reilly & Associates.

1.3.2 Caché EJB 绑定的架构

Caché 的 EJB 绑定和 Caché 的 Java 绑定很相似。主要的不同之处是 Caché 的类被作为 EJB 实体类生成出来而不是正常的 Java 类。另外，Caché 的 EJB 绑定还要额外生成 EJB 需要的一些类。

Caché 的 EJB 绑定由下面的组件组成：

- Caché 的 Java 类生成器——一个 Caché 类变异器的扩展，它从 Caché 类目录生成 EJB 类以及其它相关的文件。
- Caché 的 Java 包——一个纯 Java 类的包，它和 Caché 的 Java 类生成器生成的 Java 类一起工作，并为它们提供访问存储在 Caché 数据库中的对象的连接。这和使用 Caché 的 Java 绑定是相同的。
- Caché 的对象服务器——一个管理使用 TCP 协议的 Java 客户端和 Caché 数据库服务器至今通信的高性能的服务器进程。这个服务器和 Caché 的 Java 绑定一样。

Caché 类编译器能够自动地为任何 Caché 类目录中的类创建 EJB 实体 Bean。这些生成的 Bean 在运行的时候和在 Caché 服务器上对应的 Caché 类通信。生成的 Java 类只包含纯 Java 的代码，并且自动和主类定义保持同步。这可以用下图说明：

基本的工作机制如下：

1.你在 Caché 中定义一个或者多个类。典型地，这些是存储在 Caché 数据库中的持久的对象。

2. Caché 生成对应你的 Caché 类的 EJB 实体 Bean。这些 Bean 包含附属的方法（get 和 set）来访问所有对象的属性。

3. Caché 也生成附加的辅助类（例如主键类）、部署描述符以及部署 Bean 的脚本和测试 Bean 的代码。

4.在运行的时候，EJB 服务器连接到 Caché 服务器。当它需要创建一个实体 Bean 的实例的时候，它调用一个生成的方法来从 Caché 服务器载入数据。相似地，有生成的用来把修改的 Bean 保存到数据库中的方法。

运行时的架构包含下列内容：

- Caché 数据库服务器。
- EJB 服务器。
- 连接到 EJB 服务器的 Java 客户端应用。

运行的时候，EJB 服务器连接到 Caché，就好像它是任何的标准的 JDBC 数据源一样。由于 Caché 同时提供了 JDBC 和对象访问，EJB 服务器使用标准的 JDBC 接口来控制连接和事务。

1.3.3 安装和配置

本节表述如何设置 WebLogic 来运行 EJB 和 Caché。首先假设你默认安装了 Caché 和 WebLogic，并且 Caché 已经运行在你的系统上了。

1.3.3.1 安装和配置 EJB 服务器

要使用 EJB 和 Caché，首先开始安装和配置 EJB 服务器（本例中使用 WebLogic）。

1.安装 WebLogic。InterSystems 建议你采用全部默认的安装，包括你作为单机安装 WebLogic，不是一个服务。

2.安装完 WebLogic 以后，按照下面来编辑文件 startWebLogic.cmd，默认的，它位于 c:\Bean\wlserver6.1\config\mydomain 目录下。

- 修改 set CLASSPATH 环境变量那一行

```
set CLASSPATH=.;.\lib\weblogic_sp.jar;.\lib\weblogic.jar
```

为

```
set CACHEPATH=c:\cachesys5\dev\java\lib\CacheDB.jar
set CLASSPATH=.;.\lib\weblogic_sp.jar;.\lib\weblogic.jar;%CACHEPATH%
```

- 在关于产品和开发模式的那些行里，确定 STARTMODE 变量被设置为 false，如下：

```
@rem Set Production Mode.  When set to true, the server starts up in
@rem production mode.  When set to false, the server starts up in development
@rem mode.  If not set, it is defaulted to false.
set STARTMODE=false
```

这个变量指定 WebLogic 已“开发者模式”启动。在这个模式下，WebLogic 被设置成自动部署 Bean，在产品模式下，部署需要手工干预并且需要你提供密码。

1.3.3.2　创建 EJB 映射和 JDBC 驱动配置信息

一旦你安装并配置了 EJB 服务器，下一步就是为已经存在的类创建 EJB 映射；当你编译这个类的时候，Caché 就生成了配置 Caché JDBC 驱动需要的内容。

1. 从载入你想映射的类开始，例如在 Studio 中的 **Sample.Person**。
2. 还是在 Studio 中，从菜单 **Class** 的 **Add** 子菜单选择 **New Projection**；这将显示 **New Class Projection Wizard**。
3. 在 **New Class Projection Wizard**，指定映射的名字，例如“PersonEJBProjection”

4. 在向导的第二个界面，在 **Projection Type** 之下，点 **...** 按钮并从可用的列表中选择 **EJB**；它显示一个 EJB 映射的参数的列表。对于这个列表，至少要输入下面的条目：
   - *APPSERVERHOME*—WebLogic 安装的路径，默认是 *c:\bea\wlserver6.1*。 这个参数为了编译类而必须有一个指定的值。
   - *BEANNAME*—为映射的类生成的 Bean 的名字。如果在 CLASSLIST 中有超过一个的类，每个类都必须有它自己的 Bean 被指定，而在 CLASSLIST 中，列表必须以逗号分开。如果这个参数没有指定值，Caché 使用字符串“Package_Class_”为 Bean 相关的名字，它包含了包和简单的“类”的名字。因此，这儿有一个 *Sample_Person_* 目录包含 Bean 的信息，包括 *EJBPerson.java* 文件和 *TestSample_Person_Client* 目录，包含 *TestSample_Person__Bean.java* 文件。
   - *CLASSLIST*—当前的类，按照 **Package.Class** 的形式，并且可以用逗号分开的列表列出任何没有自动包括进来的其它类，它自动包括超类和关联类——对象类型的属性。如果这个参数没有指定值，Caché 使用当前类。
   - *JAVAHOME*—Java 安装的位置。这个参数必须有一个指定的值来为了编译类。如果你默认安装了 WebLogic，这就是 *C:\bea\jdk131*。
   - *ROOTDIR*—Caché 放置生成的 EJB 的内容的路径。如果没有指定这个参数的值，Caché 把生成的东西放在 *<cache-install-dir>\Devuser\ejb\<NS>*， *<NS>* 是包含要映射的类的命名空间。你可以在 Configuration Manager 设置这个目录。
5. 一旦你在这个向导中完成了所有必须的参数的设置，点 **Finish** 按钮。

6. 变异这个类。这将为 EJB 使用 Caché 类创建所有的必须的文件，参看下面的关于测试 Bean 的小节以获得使用这些文件的信息。为了配置的目的，这些里面最有意义的是*<BeanName>ConnectionPool.xml* 文件，*<BeanName>* 是上面 *BEANNAME* 参数指定的值，这个文件位于 *ROOTDIR* 指定的目录。

1.3.3.3 配置 Caché JDBC 驱动

一旦你映射了一个类，你就能从映射里面配置 Caché JDBC 驱动来使用数据：

1.当 WebLogic 没有运行的时候，拷贝<BeanName>ConnectionPool.xml 的内容并且立刻把它粘贴到 config.xml 中去（默认情况下，这个文件在 c:\bea\wlserver6.1\config\mydomain\），要在最后“</Domain>”标签前面。你粘贴的内容可能会像这样:

```
<JDBCConnectionPool
        CapacityIncrement="2"
        DriverName="com.intersys.jdbc.CacheDriver"
        InitialCapacity="2"
        MaxCapacity="255"
        Name="cachePool<NS>"
      Properties="user=_SYSTEM;password=sys;TCP_NODELAY=true"
        RefreshMinutes="10"
      Targets="myserver"
    TestConnectionsOnReserve="false"
    TestTableName="ISC.TestTable"
    URL="jdbc:Cache://127.0.0.1:1972/<NS>/ejbjdbc.log"/>

<JDBCTxDataSource JNDIName="Weblogic.jdbc.jts.cachePool<NS>"
   Name="cachePool<NS>" PoolName="cachePool<NS>" Targets="myserver"/>
```

每个<NS>的地方都是包含映射的类的命名空间，真正的值不会包括<>，这些元素的内容都是标准的 WebLogic 格式，唯一 Caché 特点的信息是

“JDBCConnectionPool”元素的 URL 属性。要知道这些元素的详细信息请参考 WebLogic 的文档。config.xml 也许包含多套入口——例如多个命名空间或者服务器。如果这儿有多个入口，确定这儿没有冲突，例如两个同样的命名空间有不同的密码或测试参数。

2.下一步，你可以确认 WebLogic 被正确配置和 Caché 一起工作。要做到这点，打开 Caché 的 Control Panel 并且注意多少个进程正在运行。然后，开始 WebLogic。一旦服务器开始运行，检查多少个进程在运行，这个数字应该增加了 InitialCapacity 属性定义的数字。

至此，你就配置好了 WebLogic 和 Caché 一起工作了。

1.3.4 测试 Bean

在配置 WebLogic 的过程中，你创建了一个 Bean。特别的，当你编译这个类的时候，Caché 创建了一套 Bean 和 Bean 相关的文件。例如，对于叫做 MyClass 的类和一个叫做 MyBean 的 Bean，这些是：

- MyBean——包含 Bean 相关文件的目录
- MyBeanObj——包含 Bean 相关文件的第二个目录
- testMyBeanServlet——使用 Bean 的一个基于 Web 的简单应用
- runtestclient.bat——一个调用 testMyClassServlet 的批处理
- MyBeanConnectionPool.xml——包含 EJB 连接信息的文件
- deployear.bat——创建 Web service EJB JAR 和 EAR 的批处理，于是你就可以使用新生成的 Bean 运行应用。

你可以使用示例提供的应用测试 Bean，程序是：

1.运行 deployear.bat。因为 WebLogic 运行在开发者模式下，调用这个文件连续执行一些操作而不会询问密码。deployear.bat 编译已经创建的 Bean 为可执行的类和编译 Bean 所需要的更多的文件（MyBeanDeploy.jar 盒 MyBean.jar）。

2.启动 WebLogic，你会看到类似下面这样：

<Notice> <WebLogicServer> <Started WebLogic Admin Server "myserver" for domain "mydomain" running in Development Mode>.

3.你可以通过两种方式调用测试程序：

命令行，运行 runtestclient.bat。Web 方式:

http://localhost:7001/Test<BeanName>Servlet/Test<BeanName>.html

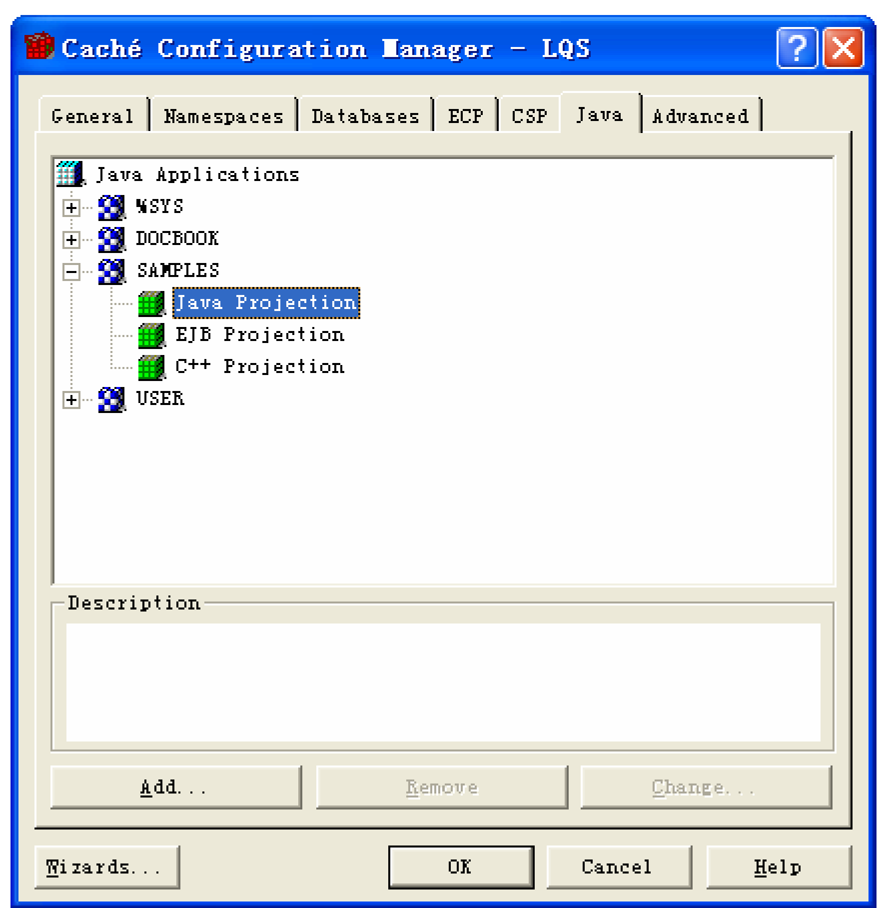
1.4 例程说明

现在我们开始做几个关于用 Java 开发 Caché 应用程序的练习。练习中的全部源代码可以在 Caché 产品的光盘中找到。

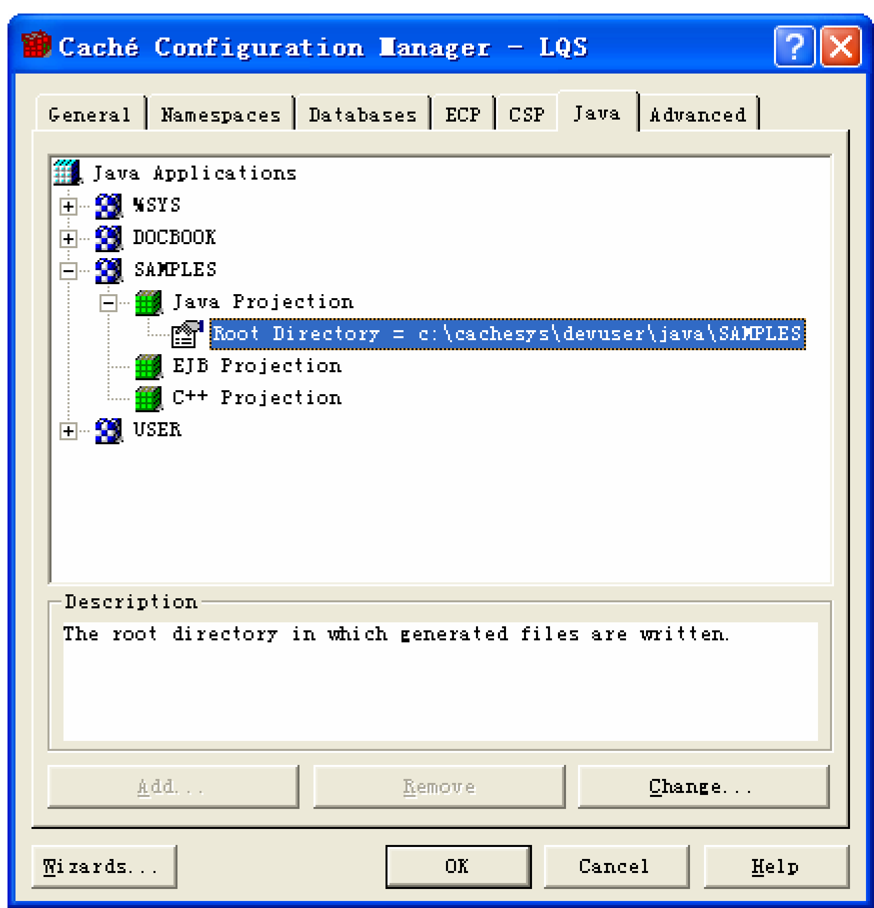
1.4.1 设置绑定的方式访问 Caché 服务器

采用绑定的方式访问 Caché，你首先需要配置你的 Caché 服务器，告诉它把 Caché 类映射到哪个目录下。

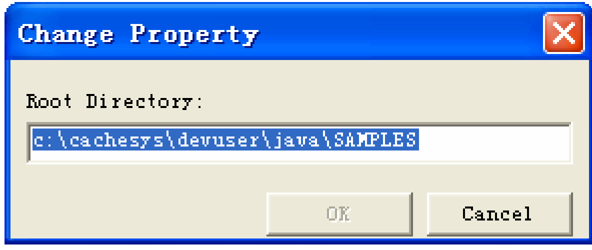
点 Caché Cube，运行 Configuration Manager，

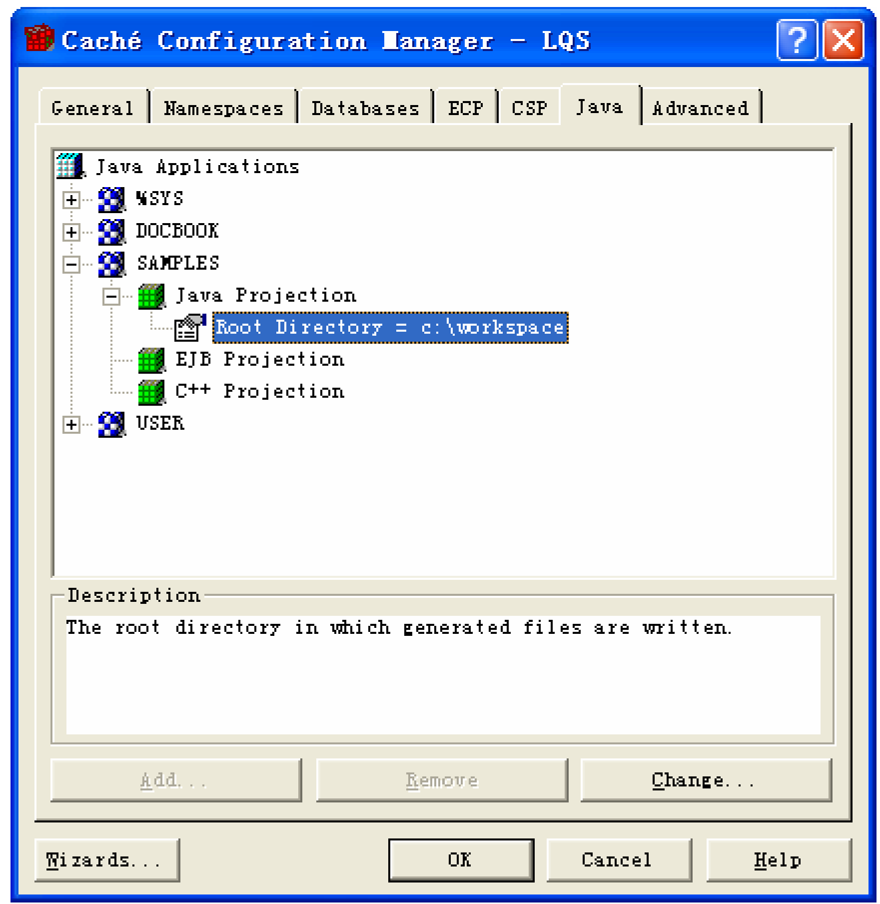


点 Add 按钮

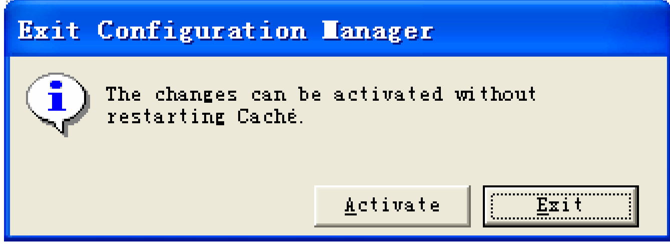


点 Change 按钮，修改路径为“c:\workspace”



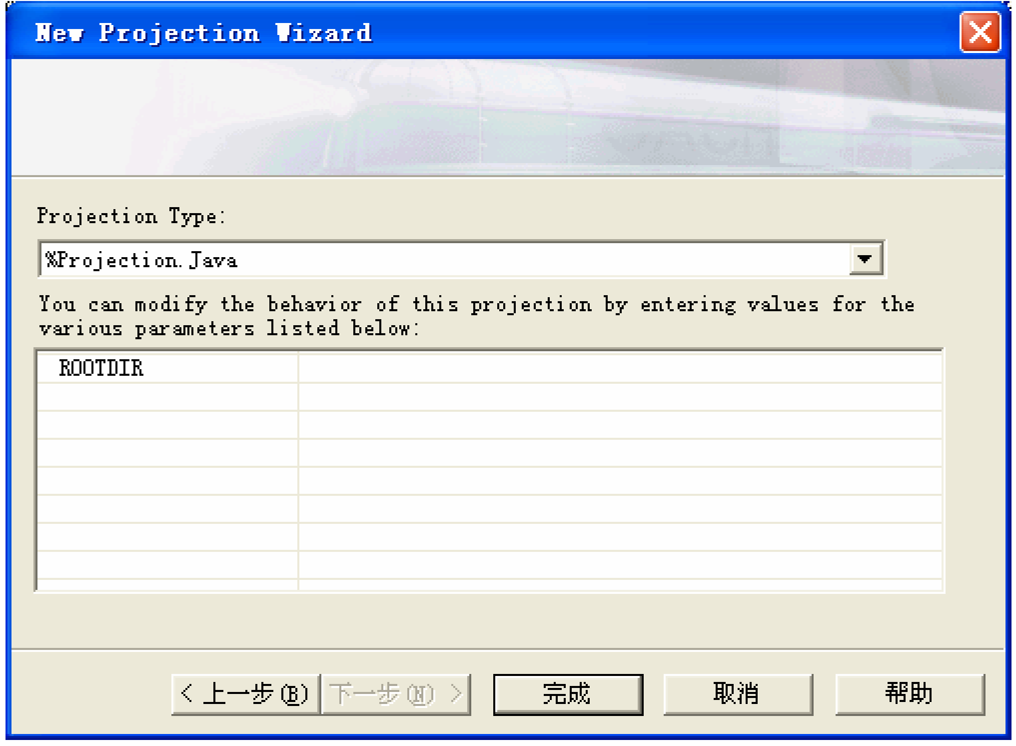


点 OK 按钮



点 Activate 按钮，可以在不必重新启动 Caché 的情况下把所做过的修改自动激活；然后再次运行 Caché Studio，打开 Samples 命名空间下面的 Sample.Person 类，点菜单 Class->Add->New Projection。

更改 Projection 名字更改为“PersonProjection”，点下一步按钮



如果这时候你想更改生成 Java 类的路径，可以修改 ROOTDIR 的值。点完成按钮。

得到如下的脚本：

“Projection PersonProjection As %Projection.Java;”

这时你编译 Sample.Person 类，将会看到：

```
Compilation started on 10/15/2004 14:02:12
Compiling class Sample.Person .................
Compiling table Sample.Person ...
Compiling routine Sample.Person.1
Compiling routine Sample.Person.2
Generating Java Binding: c:\workspace\Sample\Address.java
Generating Java Binding: c:\workspace\Sample\Person.java
Compilation finished successfully.
```

你会发现编译器自动在 c:\workspace 目录下创建了两个 Java 文件。这两个文件就是

Caché 编译器自动为 Caché 类创建的 Java 代理类。使用生成的代理类，用户就可以象使用本地的 Java 类一样来使用 Caché 类了。

1.4.2 使用绑定的方式访问 Caché 服务器

你可以通过 com.intersys.objects.CacheDatabase 类提供的一个静态方法 getDatabase 来获得一个到 Caché 的连接。getDatabase 有三个参数，分别是：

url：连接字符串，符合 JDBC 连接字符串的规范，例如

```
jdbc:Cache://127.0.0.1:1972/SAMPLES
```

username：连接 Caché 的用户名

password：连接 Caché 的用户的密码

该方法会产生一个 CacheException 异常。

1.4.3 使用 JDBC 的方式访问 Caché 服务器

用 JDBC 连接 Caché 采用标准的 JDBC 方式，首先：

Class.forName("com.intersys.jdbc.CacheDriver");

然后使用 DriverManager 类的 getConnection 方法获得连接。

结合 1.4.2，我们编写一个类 Utility.java，用它来获得到 Caché 的连接，请看下面的源代码：

```
/*
 * Utility.java
 *
 * Created on 2004年9月27日, 下午12:49
 */

package training;

/**
 * * @author  Administrator
 */
public class Utility {
   static String url = "jdbc:Cache://127.0.0.1:1972/SAMPLES";
   static String username = "_SYSTEM";
   static String password = "sys";

   public static com.intersys.objects.Database getConnection() {
      try{
         return com.intersys.objects.CacheDatabase.getDatabase(
                    url,
                    username,
```

```
                    password);
        }catch(Exception e){
            e.printStackTrace();
            return null;
        }
      }
    public static java.sql.Connection getSQLConnection(){
        try{
            Class.forName("com.intersys.jdbc.CacheDriver");
            java.sql.Connection conn = java.sql.DriverManager.getConnection(
                                                        url,username,password);
            return conn;
        }catch(Exception e){
            e.printStackTrace();
            return null;
        }
    }
}
```

其中 getConnection 方法返回一个绑定的方式的连接，getSQLConnection 返回的是一个 SQL 方式的连接。本例作者把连接字符串固定写死在程序里了，读者可以稍微修改这个例子以适合任意的场合。

1.4.4 如何使用 Person.Java

Person.Java 是从 Caché 的 Sample.Person 来的，作为 Sample.Person 的代理类，你可以用它在你的 Java 应用中直接定义对象，如：

```
Person person = null;
```

那么如何创建一个新的 Person 并把它保存在数据库中呢，很简单，请看如下代码：

```
person = new Person();
```

```
……
person._save();
```

如果我们想打开一个 Person 类的实例，可以按照如下的方法做：

```
person = (Sample.Person)Sample.Person._open(db,new Id(id));
```

其中，db 是到 Caché 数据库的连接，Id 属于 com.intersys.objects 包，id 是一个整数，表示对象的 id。_open 方法返回的实际上是一个 com.intersys.classes.RegisteredObject 对象，所有的 Caché 持久类生成的代理类都是它的子类，所以我们可以通过 Java 的强制类型转换把它转换一下。

如果你想操作对象的属性，可以采用如下的方式：

```
String name = person.getName());
String dob = person.getDOB().toString();
int age = person.getAge().intValue();
person.setName(“Zhang san”);
person.setDOB(new java.sql.Date());
person.setAge(new Integer(24));
person._save();
```

如果你想删除一个对象，可以采用如下语法：

```
Person._deleteId(db,1);
```

其中，db 是到 Caché 数据库的连接， id 是一个整数，表示对象的 id。

1.4.5 如何使用 BinaryStream

我们可以使用 BinaryStream 来用于图片、声音等的数据的操作。Sample.Person 类的 Picture 属性被我们设置成了 BinaryStream 类型，我们用这个字段保存图片数据。在编译器生成 Java 类的时候，这样的属性不生成其它类型的属性的方法，而是声称 get**Prop**In()和 get**Prop**Out()这样的方法用于读取流和写入流。请看下面的例子：

```
try{
        com.intersys.objects.CacheInputStream cin = person.getPictureIn();
        byte[] imagebyte = new byte[cin.available()];
        cin.read(imagebyte);
        image = new javax.swing.ImageIcon(imagebyte);
        this.lblPicture.setIcon(image);
        this.lblPicture.repaint();
}catch(Exception e){
        e.printStackTrace();
}
```

通过 Sample.Person 类的实例 person 的 getPictureIn()方法，我们获得一个 com.intersys.objects.CacheInputStream 对象。由于 CacheInputStream 类继承了 java.io.InputStream，所以我们用 java.io.InputStream 提供的 read 方法取出二进制流并放入一个 byte 数组。

关于如何写入二进制流，请看下面的例子：

```
try{
        java.io.FileInputStream fin = new java.io.FileInputStream(file);
    byte[] imagebyte = new byte[fin.available()];
    fin.read(imagebyte);
    image = new javax.swing.ImageIcon(imagebyte);
    this.lblPicture.setIcon(image);
    this.lblPicture.repaint();
    com.intersys.objects.CacheOutputStream cou = person.getPictureOut();
    cou.write(imagebyte);
    cou.close();
}catch(Exception e){
    e.printStackTrace();
}
```

通过 Sample.Person 类的实例 person 的 getPictureOut()方法，我们获得一个 com.intersys.objects.CacheOutputStream 对象。由于 CacheOutputStream 类继承了 java.io.OutputStream，所以我们用 java.io.OutputStream 提供的 write 方法把一个二进制数组写入这个流。

1.4.6 如何使用 CharacterStream

我们可以使用 CharacterStream 来用于很长的字符串数据的操作。Sample.Person 类的 Resume 属性被我们设置成了 CharacterStream 类型，我们用这个字段保存个人的简历数据。在编译器生成 Java 类的时候，这样的属性不生成其它类型的属性的方法，而是声称 get**Prop**In()和 get**Prop**Out()这样的方法用于读取流和写入流。请看下面的例子：

```
try{
     com.intersys.objects.CacheReader reader = this.person.getResumeIn();
   this.txtResume.setText(reader.read(99999));
   this.txtResume.repaint();
}catch(Exception e){
   e.printStackTrace();
}
```

通过 Sample.Person 类的实例 person 的 getResumeIn()方法，我们获得一个 com.intersys.objects.CacheReader 对象。由于 CacheReader 类继承了 java.io.Reader，所以我们用 java.io.Reader 提供的 read 方法取出流。

关于如何写入字符流，请看下面的例子：

```
try{
     com.intersys.objects.CacheWriter cw = this.person.getResumeOut();
   cw.write(this.txtResume.getText().toCharArray());
   cw.close();
```

```
}catch(Exception e){
   e.printStackTrace();
}
```

通过 Sample.Person 类的实例 person 的 getResumeOut()方法，我们获得一个 com.intersys.objects.CacheWriter 对象。由于 CacheWriter 类继承了 java.io.Writer，所以我们用 java.io.Writer 提供的 write 方法把一个字符数组写入这个流。

第六章 以 Delphi 开发 Caché 应用程序

1 引言

利用 Borland 公司的快速应用开发(RAD)工具 Delphi 可以用来开发 C/S 结构的客户端应用。将 Delphi 强大的 Windows 界面表现能力和 Caché 稳定、高效、便捷的服务功能相结合，将会创造出优秀的应用系统。在 Delphi 中，可以通过 COM 访问 Caché 的对象和关系服务，也可以通过传统的关系型方式访问 Caché，还可以将 Delphi 作为 Web Service 的客户端访问 Caché。这里将介绍基于 COM 技术的访问方式。

在这一节中我们将用一个简单的应用示例来具体介绍如何从 Delphi 客户端访问 Caché 数据库服务，包括实现 Delphi 客户端与 Caché 的连接、实现对 Caché 对象的访问及操作、执行预定义的查询和直接的 SQL 查询、和实现访问 Caché 的字符流、二进制流及列表等一系列工作，来逐步地完成一个我们所希望的有如下图所示的客户端用户界面的应用开发示例：

让我们来一步一步地实现它：

1.1　配置 Delphi 工程

为了在 Delphi 程序中连接 Caché，我们需要新建一个 Delphi 工程，并为之配置 CacheObject 组件。

1.2 新建 Delphi 工程

本案例中使用的 Delphi 版本为 Delphi 7。

** 我们可以在以下地址下载 Delphi 7 和 Personal 许可证：*

http://www.borland.com/products/downloads/download_delphi.html

** 执行下面操作前，请保证 Delphi 和 Caché 已被正确安装并在运行状态。*

我们首先在文件系统中建立一个新的文件夹，用来存放此工程。例如 “c:\DelphiCache\”。

下面我们将创建一个新的 Delphi 工程：

运行 Delphi，在 Delphi 中，
选择 菜单 File->New->Application。
出现了一个新的 Delphi 工程(Project1)，包含一个单元文件(Unit1)和一个窗体(Form1)。我们将为它们起新的名字：

在 Object Inspector 中，
将 Form1 的 “Name” 属性修改为 “Form_Main”，
将 “Caption” 属性修改为 “Delphi 访问 Cache”。

之后保存工程：

选择 菜单 File->Save All。
弹出窗口，要求选择单元 Unit1 保存位置，我们将它重命名为 Unit_Main，并保存在 c:\DelphiCache 中；

再次弹出窗口，要求选择工程 Project1 保存位置，我们将它重命名为 DelphiCache，并保存在 c:\DelphiCache 中；

新 Delphi 工程创建完毕，其中包含名为 Unit_Main 的主单元文件和名为 Form_Main 的窗体。

1.3 为 Delphi 工程添加 CacheObject 组件

新工程创建完后，我们要为它配置好 CacheObject 组件

** 关于通过 COM 访问 Caché 的原理，请参照单元：* *应用开发->COM->通过 COM 访问 Caché 的基础知识*

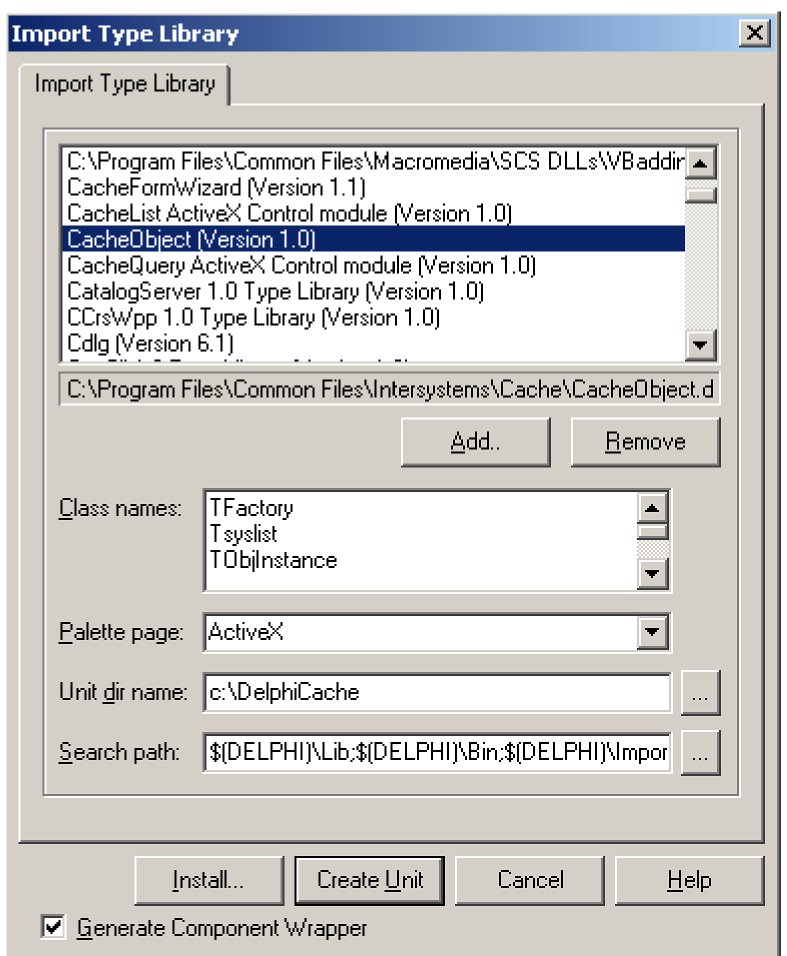
我们将要导入 CacheObject 组件的类型库到 Delphi 工程中。Delphi 会自动创建该类型库的前期绑定文件。Delphi 程序通过它提供的接口访问工厂对象。

下面我们将导入 CacheObject 组件的类型库：

在 Delphi 中，

选择 菜单 Project->Import Type Library…。

弹出 “Import Type Library”(导入类型库) 窗口。



在列表中选择名为 “CacheObject [Version 1.0]” 的类型库。

可以看到 “Class names” 栏中显示了其中包含的接口列表。

将 “Unit dir name” 栏的内容修改为 “c:\ DelphiCache”。

点击 按钮 “Create Unit”。

此时，Delphi 将产生一个名为 “CacheObject_TLB.pas” 的新文件(单元)并添加到工程中，这就是前期绑定文件。里面包含了 CacheObject 类型库中的接口（Ixxx），辅助类（Coxxx）以及 OLE 服务代理类（Txxx）的声明。

保存工程：

选择 菜单 File->Save All。

以下内容为文件 CacheObject_TLB.pas 中关于工厂对象接口(IFactory)的代码片断：

```
…
// *********************************************************************//
// DispIntf:  IFactory
// Flags:     (4096) Dispatchable
// GUID:      {A98252EA-17D9-11D1-A181-0000F8773CDC}
// *********************************************************************//
  IFactory = dispinterface
    ['{A98252EA-17D9-11D1-A181-0000F8773CDC}']
    function Connect(const ConnectString: WideString): WordBool; dispid 1;
    function Disconnect: WordBool; dispid 2;
    procedure SetLogMask(LogMask: Integer); dispid 3;
    function New(const ClassName: WideString; vtInit: OleVariant): IDispatch;
dispid 4;
    function Open(const ClassName: WideString; const OID: WideString;
Concurency: OleVariant): IDispatch; dispid 5;
    function OpenEx(const ClassName: WideString; const ID: WideString;
Concurency: OleVariant): IDispatch; dispid 6;
    function OpenId(const ClassName: WideString; const ID: WideString;
Concurency: OleVariant): IDispatch; dispid 7;
    function Static(const ClassName: WideString): IDispatch; dispid 8;
    function ConnectDlg(Title: OleVariant): WideString; dispid 9;
    function IsConnected: WordBool; dispid 10;
    function ResultSet(const ClassName: WideString; const QueryName:
WideString): IDispatch; dispid 11;
```

```
    function GetClassName(var OID: WideString): WideString; dispid 12;
    function GetId(var OID: WideString): WideString; dispid 13;
    function GetErrorText(ErrorCode: Integer; Param1: OleVariant; Param2:
OleVariant;
                         Param3: OleVariant; Param4: OleVariant; Param5:
OleVariant;
                         Param6: OleVariant; Param7: OleVariant; Param8:
OleVariant;
                         Param9: OleVariant; Param10: OleVariant): WideString;
dispid 14;
    procedure SetOutput(pOut: OleVariant); dispid 15;
    function GetLastErrorCount: Smallint; dispid 16;
    function GetLastErrorNumber(ErrNo: OleVariant): Integer; dispid 17;
    function GetLastErrorParamCount(ErrNo: OleVariant): Smallint; dispid 18;
    function GetLastErrorParam(ErrNo: OleVariant; ParamNo: OleVariant):
WideString; dispid 19;
    function DynamicSQL(const Statement: WideString): IDispatch; dispid 20;
    procedure TransactionStart; dispid 21;
    procedure TransactionRollBack; dispid 22;
    procedure TransactionCommit; dispid 23;
    function TransactionLevel: Smallint; dispid 24;
    procedure SetCacheLog(LogMask: Integer); dispid 25;
    function GetConnectionList: WideString; dispid 26;
    function IsMultibyte: WordBool; dispid 27;
  end;
…
```

可见，工厂对象中的方法定义已被绑定为 Object Pascal 语言的形式。我们随后将通过 IFactory 接口操作工厂对象。类型库中的其它接口也是这样实现的。

1.4 添加日志记录功能

为了方便地记录执行过的操作，我们将在工程中添加简单的日志记录的功能，以替代弹出对话框的方式。

添加日志控件：

在 Form_Main 中添加一个 ListBox 类型的控件，命名为 “lb_Log”，将 Align 属性设为“alBottom”。

添加日志方法：

为类 TForm_Main 添加一个公共方法声明，名为 AddLog，代码如下：

```
…
public
{ Public declarations }
  procedure AddLog(Log:String);
end;
…
```

在实现部分输入以下代码：

```
procedure TForm_Main.AddLog(Log:String);
begin
   lb_Log.Items.Add(Log);
        lb_Log.Selected[lb_Log.Count-1]:=True;
```

```
end;
```

保存工程：

选择 菜单 File->Save All。

现在，我们已经配置好了 Delphi 工程。下面单元我们将创建工厂对象的实例，并通过它与 Caché 建立连接。

1.5 与 Caché 建立连接

在 Delphi 中访问 Caché，首先要创建工厂对象，并通过它与 Caché 连接。

1.5.1 添加对 CacheObject_TLB 的引用

在工程的主单元 Unit_Main 的代码中，为 uses 列表添加对 CacheObject_TLB 单元的引用。完成后的代码如下：

```
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, CacheObject_TLB;
```

1.5.2 声明工厂对象

在 Unit_Main 中，为 TForm_Main 添加新的私有属性 CacheFactory，类型为 IFactory。完成后的代码如下：

```
type
  TForm_Main = class(TForm)
  …
  private
    { Private declarations }
    CacheFactory:IFactory;
  public
{ Public declarations }
…
  end;
```

** 一个 Delphi 应用程序中只需要建立一个 IFactory 的实例*

1.5.3 创建工厂对象实例

为简单起见，我们将会在窗体 Form_Main 创建时建立连接，窗体关闭时关闭连接。

在 Unit_Main 中，为窗体类 TForm_Main 产生 OnCreate 和 OnClose 事件的处理代码框架。

在 OnCreate 的实现部分输入如下代码：

```
procedure TForm_Main.FormCreate(Sender: TObject);
begin
// 创建 IFactory 的实例
CacheFactory := CoFactory.Create;
end;
```

这里用到了 CoFactory 类的 Create 类方法。可见，CoFactory 类的作用就是协助创建接口 IFactory 的实例(即工厂对象)。

1.5.4 连接 Caché 服务器

下面我们将通过 CacheFactory 连接服务器

我们首先要得到连接字串，再通过连接字串连接服务器。
在 OnCreate 的实现部分继续输入如下代码：

```
procedure TForm_Main.FormCreate(Sender: TObject);
// 在这里声明两个局部变量
var
  ConnectString: String; // 用来保存连接字串
  Success: Boolean; // 用来检查是否连接成功
Begin
  // 创建 IFactory 的实例
CacheFactory := CoFactory.Create;

// 这里定义了编译标记。
// 在默认情况下，程序将通过直接输入的连接字串进行连接；
// 如果希望由用户自己来选择服务器，只需将下面一行代码去掉
{$DEFINE USE_EXPLICT_CONNECTSTRING}

{$IFDEF USE_EXPLICT_CONNECTSTRING}
// 在这里直接输入连接字串。将会连接到本地服务器的 SAMPLES 命名空间上
ConnectString := 'cn_iptcp:127.0.0.1[1972]:SAMPLES';
{$ELSE}
```

```
// 否则，由用户从弹出的对话框中自行选择所要连接的服务器，并返回连接字串
ConnectString := CacheFactory.ConnectDlg('请将我连接到 SAMPLES 命名空间...');
{$ENDIF}

// 连接 Caché 服务器
Success := CacheFactory.Connect(ConnectString);
if not Success then AddLog('不能建立连接!')
else AddLog('已成功连接到 '+ConnectString);
end;
```

这里用到了工厂对象的以下方法：

ConnectDlg*(Title:OleVariant):WideString*，弹出服务器选择对话框，参数 为该对话框的标题，返回 连接字串；

Connect*(const ConnectString:WideString):WordBool*，与服务器建立连接，参数 为连接字串，返回 是否连接成功；

在 OnClose 的实现部分输入如下代码：

```
procedure TForm_Main.FormClose(Sender: TObject; var Action: TCloseAction);
begin
 // 如果已建立连接，则关闭它
 if CacheFactory .IsConnected then CacheFactory.Disconnect;
end;
```

这里用到了工厂对象的以下方法：

IsConnected*: WordBool*，返回 是否已连接到服务器上；

Disconnect，与服务器断开连接；

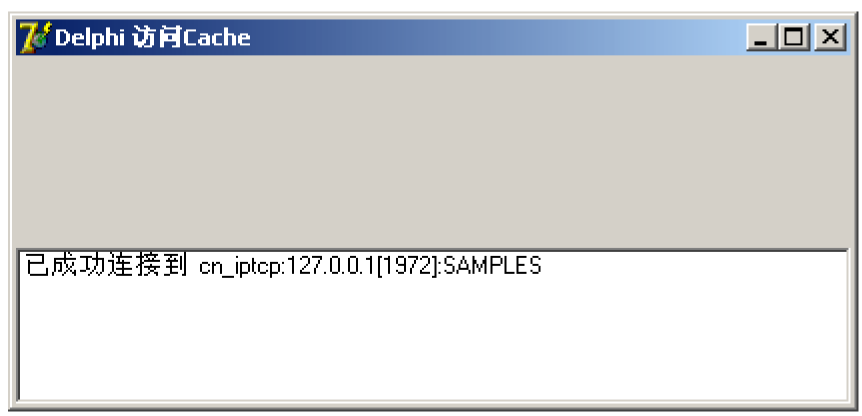
保存工程：

选择菜单 File->Save All。

测试运行：

选择菜单 Run->Run。

运行后的主窗体如图所示：



1.5.5 服务器选择对话框

当我们希望让用户自己选择服务器时，可以将代码中的编译标记定义一行注释掉：

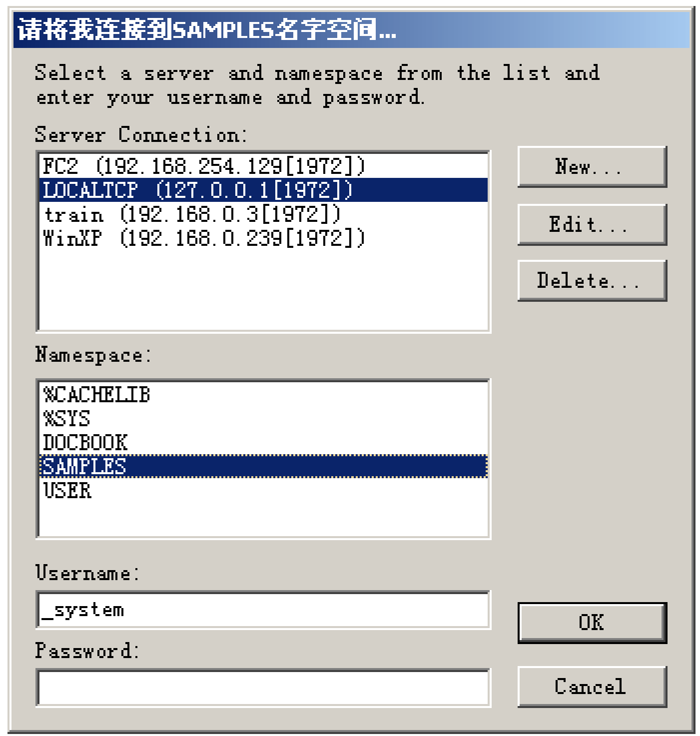
```
…
// 如果希望由用户自己来选择服务器，只需将下面一行代码注释掉
//{$DEFINE USE_EXPLICT_CONNECTSTRING}
…
```

、

此时，测试运行：

选择菜单 Run->Run。

程序会先弹出 **服务器选择对话框**。



在这里，我们可以选择选择服务器和命名空间，也可以通过它管理服务器列表。

在 Server Connection 列表中，

选择 “LOCALTCP”。

** LOCALTCP (127.0.0.1[1972])代表本地服务器。*

在 Namespace 列表中，

选择 “SAMPLES”。

点击 “OK” 返回。

** 此时服务器选择对话框返回的连接字串为 “cn_iptcp:127.0.0.1[1972]:SAMPLES”，和直接输入的完全一样。*

主窗体出现，内容和上次运行时完全相同。

关闭窗体，将编译标记**反注释**，恢复到前面的状态。

现在，我们已经可以成功地连接到 Caché 了。下面单元我们将在 Delphi 工程中访问 Caché 对象实例。

1.6 访问 Caché 对象实例

连接到 Caché 之后，我们可以灵活地访问 Caché 服务器中的各种对象。

1.6.1 关于对象实例变量

Caché 服务器端对象的本地版本将保存在对象实例变量中。

在 Delphi 中，我们可以将 Caché 中的各种对象映射到**对象实例变量**里，之后便能够调用该变量的属性和方法，调用方式和访问 Delphi 中的其它对象一样。

由于 Delphi 的 Object Pascal 属于类型强制的语言，又由于通过 COM 的调用方式决定了只能使用后期绑定方式，因此我们必须将对象实例变量保存到一种特殊类型：**Variant 类型**中。

** 关于前期类型绑定和后期类型绑定的信息，请参照 应用开发>COM>通过 COM 访问 Caché 的基础知识*

在 Unit_Main 中，为 TForm_Main 添加新的私有属性 aPerson (对象实例变量)，类型为 Variant。完成后的代码如下：

```
  …
  private
{ Private declarations }
CacheFactory:IFactory;
```

```
aPerson: Variant;
  public
{ Public declarations }
…
```

1.6.2 创建新对象实例

我们将在 Delphi 中创建第一个 Caché 对象

由于连接到了 SAMPLES 命名空间，所以我们将创建一个 Sample.Person 类的实例。

** 关于 Sample.Person 类的结构信息，可以参看联机文档：http://127.0.0.1:1972/apps/documatic 中选择 SAMPLES 命名空间， Sample 包下的 Person 类。*

1) 添加创建对象的功能

在 Form_Main 中添加一个按钮，命名为 btn_NewPerson，标题设为 “创建 Person”。

为按钮 btn_NewPerson 的 OnClick 事件创建处理代码框架。

在 OnClick 事件的实现部分输入以下代码：

```
procedure TForm_Main.btn_NewPersonClick(Sender: TObject);
begin
  // 关闭存在的 aPerson 实例
ClosePerson;
try
   // 新建类'Sample. Person'的实例
aPerson := CacheFactory.New('Sample.Person', True);
```

```
AddLog('Sample.Person 新实例已经建立');

except
   // 错误处理
   on E: Exception do begin
     AddLog('错误: '+E.Message);
     ClosePerson;
   end;
  end;
end;
```

这里用到了工厂对象的以下方法：

New*(const ClassName: WideString; vtInit: OleVariant): IDispatch*，创建对象实例并返回，参数 ClassName 为类的全名(中间不能有空格)，vtInit 设为 True，返回 对象实例引用；

New 方法与服务器端 COS 中的以下语句相当：

Set aPerson=##class(Sample.Person).%New()

ClosePerson 方法将在后面添加。

错误信息可以通过异常处理机制获得。

2) 添加关闭对象的相关功能

为类 TForm_Main 添加一个私有方法声明，名为 ClosePerson，代码如下：

```
…
  aPerson: Variant;
  procedure ClosePerson;
public
…
```

在实现部分输入以下代码：

```
procedure TForm_Main. ClosePerson;
begin
 // 关闭已存在的 aPerson 实例
if VarIsNull(aPerson) then Exit;
    if VarIsClear(aPerson) then begin
      aPerson := NULL;
      Exit;
end;
aPerson.sys_Close;
         aPerson := NULL;
 AddLog('Sample.Person 的实例已经关闭');;
end;
```

函数 ***VarIsNull*** 和 ***VarIsClear*** 可以判断 Variant 类型的变量(aPerson)是否为空值。

通过调用 aPerson 的 sys_Close 方法关闭内存中的实例引用 (即关闭对象实例)。(对象实例方法的调用将在后面介绍)

在 Form_Main 中添加一个按钮，命名为 btn_ClosePerson，标题设为 “关闭 Person”。

为按钮 btn_ClosePerson 的 OnClick 事件创建处理代码框架。

在 OnClick 事件的实现部分输入以下代码：

```
procedure TForm_Main.btn_ClosePersonClick(Sender: TObject);
begin
 ClosePerson;
end;
```

在 TForm_Main 类的 FormClose 方法实现部分中最开始处输入以下代码：

```
procedure TForm_Main.FormClose(Sender: TObject; var Action:
TCloseAction);
begin
   ClosePerson;
   // 如果已建立连接，则关闭它
   if CacheFactory .IsConnected then CacheFactory.Disconnect;
end;
```

这样，程序退出之前会自动关闭对象实例。

保存工程：

选择菜单 File->Save All。

测试运行：

选择菜单 Run->Run。

运行后，点击按钮 “创建 Person”，

点击按钮 “关闭 Person”，

查看日志，显示内容如下：

已成功连接到 cn_iptcp:127.0.0.1[1972]:SAMPLES

Sample.Person 新实例已经建立

Sample.Person 的实例已经关闭

运行成功。

** 上面的操作只是在内存中创建了新的 Sample.Person 的实例，之后又关闭了它，该实例并没有被保存到磁盘中。*

1.6.3 打开对象实例

我们将在 Delphi 中打开数据库中保存的某个 Caché 对象

在 Form_Main 中添加一个按钮，命名为 btn_OpenPerson，标题设为 “打开 Person”。

我们将通过用户界面得到要打开的对象的 Id 号码：

在 Form_Main 中添加一个输入框，命名为 ed_Id，内容设为 “1”。

在 ed_Id 旁边添加相应文字控件，命名为 lb_Id，标题设为 “ID: ”。

为按钮 btn_OpenPerson 的 OnClick 事件创建处理代码框架。

在 OnClick 事件的实现部分输入以下代码：

```
procedure TForm_Main.btn_OpenPersonClick(Sender: TObject);
begin
  // 关闭存在的 aPerson 实例
  ClosePerson;

  try
    // 打开类'Sample. Person'的实例
    aPerson := CacheFactory.OpenId('Sample.Person', ed_Id.Text ,1);
    AddLog('Sample.Person 的第'+ed_Id.Text+'个实例已经打开');

  except
    // 错误处理
    on E: Exception do begin
      AddLog('错误: '+E.Message);
      ClosePerson;
    end;
  end;
end;
```

这里用到了工厂对象的以下方法：

OpenId*(const ClassName: WideString; const ID: WideString; Concurency: OleVariant): IDispatch*，从数据库打开对象实例并返回，参数 ClassName 为类的全名(中间不能有空格)，ID 为对象的 Id 号，Concurrency 为数据同步访问(相当于锁)的设置，返回 对象实例引用；

Concurrency 值有以下可选项：

0	No Locking
1	Atomic (默认使用)
2	Shared
3	Shared/Retained
4	Exclusive

* *详细信息可以参看联机文档：*

http://127.0.0.1:1972/csp/docbook/DocBook.UI.Page.cls?KEY=GOBJ_concurrency

保存工程：

选择菜单 File->Save All。

测试运行：

选择菜单 Run->Run。

运行后，在输入框 ed_Id 中输入要打开的 Id 号码 (默认为 1)，

* *新安装的 Caché 中保存了 200 个 Sample.Person 的实例，其 ID 为 1 到 200。我们可以通过 Caché 立方体->Explorer->SAMPLES 命名空间->Globals->Sample.PersonD 的根节点查看数据库中的实例数量。*

点击按钮“打开 Person”，

点击按钮“关闭 Person”，

查看日志，显示内容如下：

已成功连接到 cn_iptcp:127.0.0.1[1972]:SAMPLES

Sample.Person 的第 1 个实例已经打开

Sample.Person 的实例已经关闭

运行成功。

1.6.4 查看对象实例

我们将在 Delphi 中查看对象的各种属性。

在下面的内容中，我们将了解到怎样在 Delphi 中调用 Caché 对象实例的方法、属性和进行其它复杂的操作。

我们要在窗体 Form_Main 中添加一些控件用来显示和修改其属性。查看了 Sample.Person 的定义后，我们挑选了以下几个典型的属性用来显示在界面中：

			前端 - 界面控件			
			名称	类型	只读	备注
第一阶段						
Name	姓名	是	ed_Name	输入框	否	
SSN	号码	是	ed_SSN	输入框	否	NNN-NN-NNNN
DOB	生日	否	ed_DOB	输入框	否	
Age	年龄	否	ed_Age	输入框	是	随 DOB 变化

<table>
<tr><td colspan="10">第二阶段</td></tr>
<tr><td rowspan="4">Home or Office</td><td>State</td><td rowspan="4">家庭或办公室地址</td><td>州</td><td>否</td><td>ed_State</td><td>输入框</td><td>否</td><td rowspan="4">两个属性共用界面</td><td>2 个字母</td></tr>
<tr><td>City</td><td>城市</td><td>否</td><td>ed_City</td><td>输入框</td><td>否</td><td></td></tr>
<tr><td>Street</td><td>街</td><td>否</td><td>ed_Street</td><td>输入框</td><td>否</td><td></td></tr>
<tr><td>Zip</td><td>邮编</td><td>否</td><td>ed_Zip</td><td>输入框</td><td>否</td><td>5 个数字</td></tr>
<tr><td colspan="10">第三阶段</td></tr>
<tr><td colspan="2">FavoriteColors</td><td colspan="2">喜好颜色集合</td><td>否</td><td>lb_Colors</td><td>列表框</td><td>是</td><td colspan="2">一组元素</td></tr>
<tr><td colspan="10">第四阶段</td></tr>
<tr><td colspan="2">Spouse</td><td colspan="2">配偶</td><td>否</td><td>ed_Spouse</td><td>输入框</td><td>否</td><td colspan="2">显示配偶姓名</td></tr>
</table>

我们将工作分为四个阶段进行。

第一阶段：

处理普通属性。

需要在界面中按照上表添加四个输入框和相关文字控件。

其中，年龄控件 ed_Age 的：

ReadOnly 属性应设为 True。

号码输入框 ed_SSN 的：

MaxLength 属性设为 11。(格式为 NNN-NN-NNNN，共 11 位)

添加这些控件之后，窗体外观如下图所示：

下面我们要在 TForm_Main 中专门建立一个方法，用来将 aPerson 中的信息反映到界面控件中。

为类 TForm_Main 添加一个私有方法声明，名为 DisplayPerson，代码如下：

```
…
  aPerson: Variant;
  procedure ClosePerson;
  procedure DisplayPerson;
public
…
```

在实现部分输入以下代码：

```
procedure TForm_Main.DisplayPerson;
begin
  ed_Id.Text := aPerson.sys_Id;
     ed_Name.Text := aPerson.Name;
     ed_SSN.Text := aPerson.SSN;
```

```
    ed_DOB.Text := aPerson.DOB;
    ed_Age.Text := aPerson.Age;
end;
```

以上语句刷新了属性显示控件的相应内容。

其中 Caché 对象实例(aPerson)的**属性调用**用到了以下语法：

… := 对象实例.属性;

我们将可以用这样的语法访问 Caché 对象实例的属性。

但 aPerson.Sys_Id 是方法调用，相当于 COS 中的：

Do aPerson.%Id()

其中，COS 中系统方法的百分号 “%” 在 Object Pascal 中用 “sys_” 替代，如 “%Close” 变为 “sys_Close”，“%Save” 变为 “sys_Save”。

可见，在 Delphi 中 Caché 对象实例**方法调用**的语法为：

对象实例.方法(参数 1，参数 2，…);

** 无参数时省略括号*

** 可以调用实例方法和类方法*

然后，在 btn_NewPersonClick 方法中的 except 语句之前添加一句代码：

```
        …
AddLog('Sample.Person 新实例已经建立');
        DisplayPerson;
except
…
```

这样，新建对象的内容将会显示到窗体中。

然后，在 btn_OpenPersonClick 方法中的 except 语句之前添加一句代码：

```
        …
AddLog('Sample.Person 的第'+ed_Id.Text+'个实例已经打开');
        DisplayPerson;
```

except
…
这样，打开的对象的内容将会显示到窗体中。

保存工程：选择菜单 File->Save All。
测试运行：选择菜单 Run->Run。
运行后，选择新建或打开对象，即可看到窗体中显示出了相应的内容。

第二阶段：
处理嵌入对象的属性。
由于 Home 和 Office 属性都是 Sample.Address 嵌入类型，因此我们采取两个属性共用同一组界面的设计，可以使我们更深刻地理解嵌入类型。

首先，在界面中添加第二阶段的四个属性输入框和相关文字控件。
其中，需要将州名称输入框 ed_State 的：
MaxLength 属性设为 2，
CharCase 属性设为 ecUpperCase；
将邮编输入框 ed_Zip 的：
MaxLength 属性设为 5。
然后，在界面中添加一个 ComboBox 控件，命名为 cb_Address，它用来在 Home 属性和 Office 属性之间切换。
将控件 cb_Address 的：
Style 属性设为 csDropDownList，
ItemIndex 属性设为 0。
编辑 cb_Address 的 Items 属性内容为：
家庭住址
办公室地址

这样，cb_Address 将会有两个选项。

添加这些控件之后，窗体外观如下图所示：

下面我们添加和补充一些属性和方法定义：

在 Unit_Main 中，

为 TForm_Main 添加新的私有属性 aPersonAddress，类型为 Variant，用来保存当前正在使用的 Address 引用。

为 TForm_Main 添加新的私有方法 AssignAddress，用来设置 aPersonAddress 的值。

为 TForm_Main 添加新的私有方法 DisplayAddress，用来显示 aPersonAddress 的内容。

完成后的代码如下：

```
 …
 private
   { Private declarations }
   CacheFactory:IFactory;
   aPerson: Variant;
   aPersonAddress: Variant;
   procedure ClosePerson;
procedure DisplayPerson;
```

```
procedure AssignAddress;
procedure DisplayAddress;
  public
{ Public declarations }
…
```

为方法 AssignAddress 的实现部分输入以下代码：

```
procedure TForm_Main.AssignAddress;
begin
  // 根据 cb_Address 的选择，决定 aPersonAddress 代表 Home 还是 Office
if cb_Address.ItemIndex=0 then
    aPersonAddress:=aPerson.Home
  else
    aPersonAddress:=aPerson.Office;
end;
```

在这里，aPerson 的 Home 和 Office 都不是简单的字串类型，而是另一个 Caché 对象。这个对象的生存周期完全由 aPerson 管理，不需要写代码手动地创建和关闭。我们可以将此对象引用保存在另一个 Variant 类型的变量(aPersonAddress)中。

为方法 DisplayAddress 的实现部分输入以下代码：

```
        procedure TForm_Main.DisplayAddress;
begin
  AssignAddress;
    ed_State.Text := aPersonAddress.State;
    ed_City.Text := aPersonAddress.City;
    ed_Street.Text := aPersonAddress.Street;
    ed_Zip.Text := aPersonAddress.Zip;
end;
```

为方法 DisplayPerson 的实现部分补充输入对 DisplayAddress 的调用：

```
      …
  ed_Age.Text := aPerson.Age;
  DisplayAddress;
end;
 …
```

下面为控件 cb_Address 的 OnChange 事件创建处理代码框架。

在 OnChange 事件的实现部分输入以下代码：

```
procedure TForm_Main.cb_AddressChange(Sender: TObject);
begin
  if VarIsNull(aPerson) or VarIsClear(aPerson) then Exit;
  try
    DisplayAddress;
    AddLog('已切换到 '+cb_Address.Text);
  except
    // 错误处理
    on E: Exception do begin
      AddLog('错误: '+E.Message);
      ClosePerson;
    end;
  end;
end;
```

在切换 Home/Office 时，重新显示 Address 的内容。

保存工程：选择菜单 File->Save All。

测试运行：选择菜单 Run->Run。

运行后，选择打开对象，

即可看到窗体中显示出了相应的地址信息。

在下拉框中选择 "家庭住址" 或 "办公室地址"，

窗体中的地址信息将随选择而变。

第三阶段：

处理集合对象的属性。

由于 FavoriteColors 属性是 List 集合类型(Collection)，子类型为字串。因此我们采取列表框的方式来显示其内容。

在界面中添加一个 ListBox 控件，命名为 lb_Colors，它用来显示 FavoriteColors 的内容。

窗体外观如下图所示：

下面我们添加和补充一些属性和方法定义：

在 Unit_Main 中，

为方法 DisplayPerson 的实现部分补充输入以下代码：

```
procedure TForm_Main.DisplayPerson;
var
  i:Integer;
  Colors:Variant;
begin
  …
  DisplayAddress;
  lb_Colors.Clear;
  Colors := aPerson.FavoriteColors;
  for i:=1 to Colors.Count do
    lb_Colors.Items.Add(Colors.GetAt(i));
end;
```

以上代码将 aPerson 的 FavoriteColors 属性传给 Colors 变量，它将被绑定成 Caché 的%Library.ListOfDateTypes 类。后面用到了该类的 Count 和 GetAt 方法遍历其内容。

** 关于%ListOfDateTypes 类定义请参看联机文档：http://127.0.0.1:1972/apps/documatic 中选择 %SYS 命名空间， %Library 包下的 ListOfDateTypes 类*

** 在程序中也可以直接用 aPerson.FavoriteColors.GetAt(i)这样的连续引用的语法。*

** ListOfDateTypes 类的元素序号从 1 开始*

保存工程：选择菜单 File->Save All。

测试运行：选择菜单 Run->Run。

运行后，选择打开对象，

即可看到列表框中显示出了 Person 的喜好颜色。

第四阶段：

处理持久对象引用属性。

由于表示配偶的 Spouse 属性也是 Sample.Person 类型，在磁盘中有自己的存储方式，不同于嵌入类对象。因此我们在读出该属性后，只显示出该对象的名字即可。我们还将创建一个功能，将配偶的信息直接显示到界面中。

在界面中添加一个输入框，命名为 ed_Spouse。它将用来显示配偶的名字。

在界面中添加一个按钮，命名为 btn_OpenSpouse，将其 Caption 属性设为“切换到配偶”。

完成后，窗体外观如下图所示：

1) 显示配偶的名字

在 Unit_Main 中，

为方法 DisplayPerson 的实现部分补充输入以下代码：

```
procedure TForm_Main.DisplayPerson;
var
   i:Integer;
   Colors:Variant;
   Spouse:Variant;
begin
…
```

```
  for i:=1 to Colors.Count do
    lb_Colors.Items.Add(Colors.GetAt(i));

  // 下面的语句将利用 Swizzle 技术隐式地打开 Spouse 的实例
  Spouse := aPerson.Spouse;
  if not (VarIsNull(Spouse) or VarIsClear(Spouse)) then begin
    ed_Spouse.Text := Spouse.sys_Id+': '+Spouse.Name;
    // 用完之后关闭 Spouse 的实例
    Spouse.sys_Close;
  end else
    ed_Spouse.Text := ' ';
end;
```

以上代码声明了 Variant 类型的变量 Spouse。

其中，语句 “Spouse := aPerson.Spouse;” 执行时，将通过本地内存中 Spouse 属性的 Id 号码隐式地打开 Caché 数据库中保存的 Sample.Person 类型的相应实例，这个过程称作 ***Swizzling***。代码中看不到类似 OpenId 的方法调用，一切都将会自动完成。产生的 Spouse 实例将被返回到 Spouse 变量中独立存在，当 aPerson 实例关闭时不会自动关闭 Spouse 实例。因此，我们在读取了 Spouse 的 Name 属性后，要手动地关闭它。

2) 显示配偶的全部信息

要显示配偶的全部信息，只需将当前 aPerson 关闭，然后将提前读出的 Spouse 传给 aPerson 变量，再刷新界面即可。

为按钮 btn_OpenSpouse 的 OnClick 事件创建处理代码框架。

在 OnClick 事件的实现部分输入以下代码：

```
procedure TForm_Main.btn_OpenSpouseClick(Sender: TObject);
```

```
var
   Spouse:Variant;
begin
   if VarIsNull(aPerson) or VarIsClear(aPerson) then Exit;
   // 下面的语句将利用 Swizzle 技术隐式地打开 Spouse 的实例
   Spouse := aPerson.Spouse;
   if VarIsNull(Spouse) or VarIsClear(Spouse) then begin
      AddLog('无法切换');
      Exit;
   end;
        // 关闭当前 aPerson 实例
   ClosePerson;
        // 将 aPerson 指向 Spouse
   aPerson := Spouse;
   AddLog('已切换到配偶，第'+Spouse.sys_Id+'个实例已经打开');
        // 刷新界面
   DisplayPerson;
end;
```

保存工程：选择菜单 File->Save All。

测试运行：选择菜单 Run->Run。

运行后，选择打开对象，

** 请打开 Id 为 101-200 的 Person 实例，因为它们的 Spouse 属性不为空。*

即可看到配偶的 Id 和名字。(没有配偶则显示为空) 运行结果如下图：

点击“切换到配偶”，界面将刷新为配偶的相关信息。查看日志。

下面我们将尝试修改 aPerson 对象的信息。

1.6.5 修改并保存对象实例

我们将在 Delphi 中修改对象的属性，并保存回数据库中。

下面我们将继续添加代码，让界面信息的变化返回到 Caché 对象中去，还要在窗体 Form_Main 中添加一些控件用来完成修改的操作。

1) 添加保存界面信息的方法

我们知道，DisplayPerson 和 DisplayAddress 方法的作用是将 aPerson 中的信息显示到界面上；我们将要创建相反功能的方法：UpdatePerson 和 UpdateAddress，将界面的内容返回到 aPerson 中。

在 Unit_Main 中，为类 TForm_Main 添加两个私有方法声明，名为 UpdatePerson 和 UpdateAddress，代码如下：

```
…
  procedure DisplayAddress;
  procedure UpdatePerson;
  procedure UpdateAddress;
public
…
```

在相应实现部分输入以下代码：

```
procedure TForm_Main.UpdatePerson;
begin
   aPerson.Name := ed_Name.Text;
   aPerson.SSN := ed_SSN.Text;
   aPerson.DOB := ed_DOB.Text;
   UpdateAddress;
end;

procedure TForm_Main.UpdateAddress;
begin
   aPersonAddress.State := ed_State.Text;
   aPersonAddress.City := ed_City.Text;
   aPersonAddress.Street := ed_Street.Text;
   aPersonAddress.Zip := ed_Zip.Text;
end;
```

以上语句刷新了 aPerson 的属性。

为方法 cb_AddressChange 的实现部分补充输入以下代码：

```
procedure TForm_Main.cb_AddressChange(Sender: TObject);
begin
   if VarIsNull(aPerson) or VarIsClear(aPerson) then Exit;
   try
      UpdateAddress;
      DisplayAddress;
      AddLog('已切换到 '+cb_Address.Text);
   except
        …
begin
```

在切换 Home/Office 时，先将界面信息存入切换前的 Address 对象中，再进行切换并显示切换后的 Address 对象的属性。

在 Form_Main 中添加一个按钮，命名为 btn_SavePerson，标题设为“保存 Person”。

为按钮 btn_SavePerson 的 OnClick 事件创建处理代码框架。

在 OnClick 事件的实现部分输入以下代码：

```
procedure TForm_Main.btn_SavePersonClick(Sender: TObject);
begin
   if VarIsNull(aPerson) or VarIsClear(aPerson) then Exit;
   try
      UpdatePerson;
      aPerson.sys_Save;
      DisplayPerson;
      AddLog('Sample.Person 的第'+aPerson.sys_Id+'个实例已经成功保存');
```

```
  except
    // 错误处理
    on E: Exception do
      AddLog('错误: '+E.Message);
  end;
end;
```

以上代码通过 sys_Save 方法保存 aPerson 中的信息到 Caché 数据库中。相当于执行了 COS 语句：

Do aPerson.%Save()

在 sys_Save 执行保存的过程中，会检查 aPerson 中属性的内容是否合法(Valid)，如果出现不合法的情况(如 SSN 属性中出现了字母，或未填写非空属性等)，将会抛出异常，传出错误信息，并将之前保存过的内容回滚(Rollback)，使数据库内部返回到执行 sys_Save 之前的状态。

保存成功后，要重新刷新界面。因为如果保存的是新建对象的话，在保存过后将会被分配新的 Id 值，可以通过刷新界面显示出来。

** 对于修改过的对象，只有经过保存，其变化才能被永久地记录，否则对象实例只是存在于内存中，一旦程序关闭(或断开连接)后，前面所作的各种修改将完全消失，不会影响到数据库中的数据。*

保存工程：选择菜单 File->Save All。

测试运行：选择菜单 Run->Run。

运行后，选择新建或打开对象，

修改属性，(期间可以切换 Home/Office，或尝试忽略非空属性，或填入不合法的信息等)

选择保存，查看日志。(之后可以再重起程序打开相应实例进行查看)

** 也可以到 Caché 中查看相应数据的变化。*

2) 通过调用类方法得到年龄 Age

Age (年龄)是通过 DOB(生日)计算出来的属性，所以我们不能直接设置 Age。在这里，我们可以尝试调用 Sample.Person 的类方法 CurrentAge 从某一 DOB 值得到 Age。

为输入框 ed_DOB 的 OnExit 事件创建处理代码框架。
在 OnExit 事件的实现部分输入以下代码：

```
procedure TForm_Main.ed_DOBExit(Sender: TObject);
var
  PersonStatic:Variant;
begin
  try
    PersonStatic := CacheFactory.Static('Sample.Person');
    ed_Age.Text := PersonStatic.CurrentAge(ed_DOB.Text);
  except
    on E: Exception do
      AddLog('错误: '+E.Message);
  end;
end;
```

上面语句中展示了怎样调用类方法。类方法可以通过工厂对象(CacheFactory)产生代理对象进行调用，可以不依赖于具体实例(如 aPerson)。我们用 Sample.Person 类的代理对象 PersonStatic 调用了类方法 CurrentAge。

这里用到了工厂对象的以下方法：

Static*(const ClassName: WideString): IDispatch;* 参数 ClassName 为类的全名(中间不能有空格)，返回 代理对象的引用。

代理对象只能用来调用类方法，语法为：

代理对象.方法(参数 1，参数 2，…);

** 无参数时省略括号*

保存工程：选择菜单 File->Save All。

测试运行：选择菜单 Run->Run。

修改生日的值，(如 “1949-10-01”，此时不需要有对象实例存在)

切换该控件的焦点，

查看年龄栏中的值。

3) 修改喜好颜色集合

我们将为 FavoriteColors 属性提供添加和删除元素的功能。

在 Form_Main 中添加一个 ComboBox 控件，命名为 cb_Colors，它用来选择和编辑颜色名称。

将控件 cb_Colors 的：

Text 属性设为 “”。

编辑 cb_Colors 的 Items 属性内容为：

Red

Orange

Yellow

Green

Blue

Purple

Black

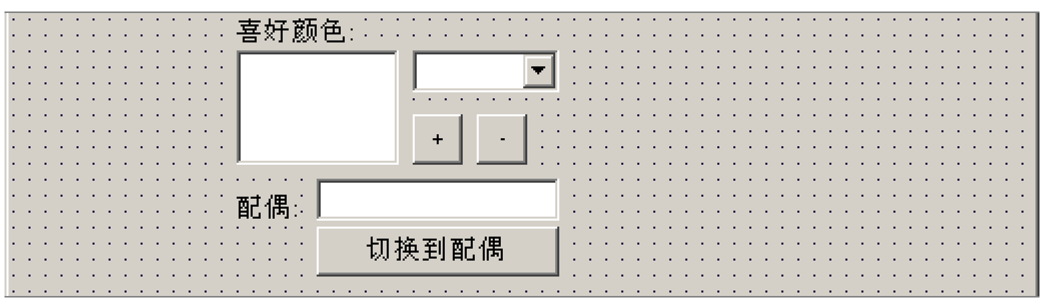
White

这样，cb_Colors 将会有八个颜色选项。

在 Form_Main 中添加一个按钮，命名为 btn_Add，标题设为 “+”。

在 Form_Main 中添加一个按钮，命名为 btn_Remove，标题设为 “-”。

窗体外观如下图所示：



然后，为按钮 btn_ Add 和 btn_Remove 的 OnClick 事件创建处理代码框架。

在 OnClick 事件的实现部分输入以下代码：

```
procedure TForm_Main.btn_AddClick(Sender: TObject);
begin
  if VarIsNull(aPerson) or VarIsClear(aPerson)
    or (cb_Colors.Text='') then Exit;
  try
    aPerson.FavoriteColors.Insert(cb_Colors.Text);
    lb_Colors.Items.Add(cb_Colors.Text);
  except
    on E: Exception do
      AddLog('错误: '+E.Message);
  end;
end;

procedure TForm_Main.btn_RemoveClick(Sender: TObject);
begin
  if VarIsNull(aPerson) or VarIsClear(aPerson)
    or (lb_Colors.ItemIndex=-1) then Exit;
  try
    aPerson.FavoriteColors.RemoveAt(lb_Colors.ItemIndex+1);
```

```
      lb_Colors.Items.Delete(lb_Colors.ItemIndex);
    except
      on E: Exception do
        AddLog('错误: '+E.Message);
    end;
end;
```

以上语句用到了 Caché 类%Library.ListOfDataTypes 的 Insert 和 RemoveAt 方法，可以和 ListBox 的相应方法(Add 和 Delete)作比较。

** RemoveAt 方法和前面介绍的 GetAt 方法一样，参数(为元素序号)是从 1 开始计数的。*

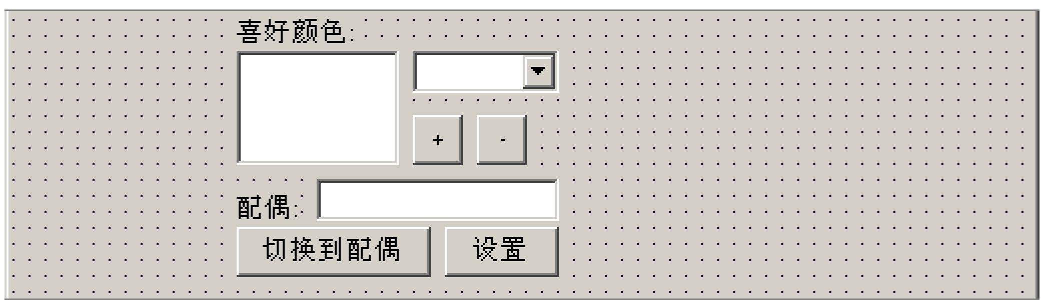
保存工程：选择菜单 File->Save All。
测试运行：选择菜单 Run->Run。
运行后，选择新建或打开对象，
在下拉框中选择一种颜色，(或直接输入其它字符)
点击 “+”，列表中将会添加一个元素，
在列表中选择一个元素，
点击 “-”，列表中将会删除该元素。
点击保存使改变生效。

4) 添加配偶

下面我们通过为 Person 实例添加配偶(Spouse)项来了解怎样实现对另一持久对象的引用。

在 Form_Main 中添加一个按钮，命名为 btn_AssignSpouse，标题设为 “设置”。

窗体外观如下图所示：



输入框 ed_Spouse 将用来输入一个 Id 值，程序通过此 Id 值找到相应对象实例作为当前对象的配偶。

下面，为按钮 btn_ AssignSpouse 的 OnClick 事件创建处理代码框架。

在 OnClick 事件的实现部分输入以下代码：

```
procedure TForm_Main.btn_AssignSpouseClick(Sender: TObject);
var
  id:Integer;
  Spouse:Variant;
begin
  if VarIsNull(aPerson) or VarIsClear(aPerson) then Exit;
  try
    id := StrToInt(ed_Spouse.Text);
    if not Variant(CacheFactory.Static('Sample.Person')).sys_ExistsId(id)
then
      AddLog('错误: 配偶 Id: '+IntToStr(id)+' 不存在')
    else begin
      Spouse := CacheFactory.OpenId('Sample.Person',IntToStr(id),1);
      aPerson.Spouse := Spouse;
      AddLog('第'+aPerson.sys_Id+'个实例的配偶已设置,Id 为
'+Spouse.sys_Id);
```

```
        Spouse.sys_Close;
        DisplayPerson;
      end;
    except
      on E: Exception do
        AddLog('错误: '+E.Message);
    end;
end;
```

从以上语句可以看到，我们首先用 CacheFactory 调用了 Sample.Person 的 sys_ExistsId 类方法判断 Id 值是否存在，之后按照 Id 打开实例保存在 Spouse 局部变量中。

下面语句将 aPerson 的 Spouse 属性指向实例 Spouse：

aPerson.Spouse := Spouse;

这样，两个持久对象的引用关系就建立了。

之后，关闭 Spouse 实例，刷新界面。

** 关闭 Spouse 实例之后，如果再通过 aPerson.Spouse 语法进行引用，系统将会自动将硬盘中存储的 Spouse 的实例调用到内存中，即所谓的 Swizzle 操作。另外，只有保存 aPerson 实例之后才会使该引用关系永久生效。*

保存工程：选择菜单 File->Save All。

测试运行：选择菜单 Run->Run。

运行后，选择新建或打开对象，

在配偶输入框中输入 Id 号，

点击 “设置”，查看日志，

配偶输入框中将显示配偶的 Id 和名字信息，

点击保存使改变生效。

程序执行后的界面如下图所示：

在本单元中，我们已经尝试了在 Delphi 客户端操作 Caché 对象的基本方法，包括创建、读取、修改和保存对象，以及嵌入、集合和引用属性的访问方法。下面单元我们将在 Delphi 工程中执行 Caché 的查询功能。

1.7 执行 Caché 查询

下面我们将通过工厂对象执行预定义的查询或 SQL 语句。

开发人员对于 SQL 语言都是比较熟悉的，而且在一些情况下进行 SQL 查询也是更方便的做法。Caché 同时提供了对象和关系的访问方式，这使得我们可以不拘泥于单一的数据访问方式，从而增加了我们在编写应用程序时的灵活性。

我们可以直接通过工厂对象进行 SQL 查询，而不必通过 ODBC 接口。

1．执行预定义查询

下面我们将为程序添加新的功能，其中用到了在 Caché 类中预先定义并封装的查询语句。该新功能将从人名的片断信息查找到可能的 Person：

通过输入人名的片断信息(如 “Ja”)，程序将会自动匹配出符合的名单供选择。

匹配过程中执行了 Sample.Person 类的预定义查询 ByName。

在类定义中，它以如下形式声明：

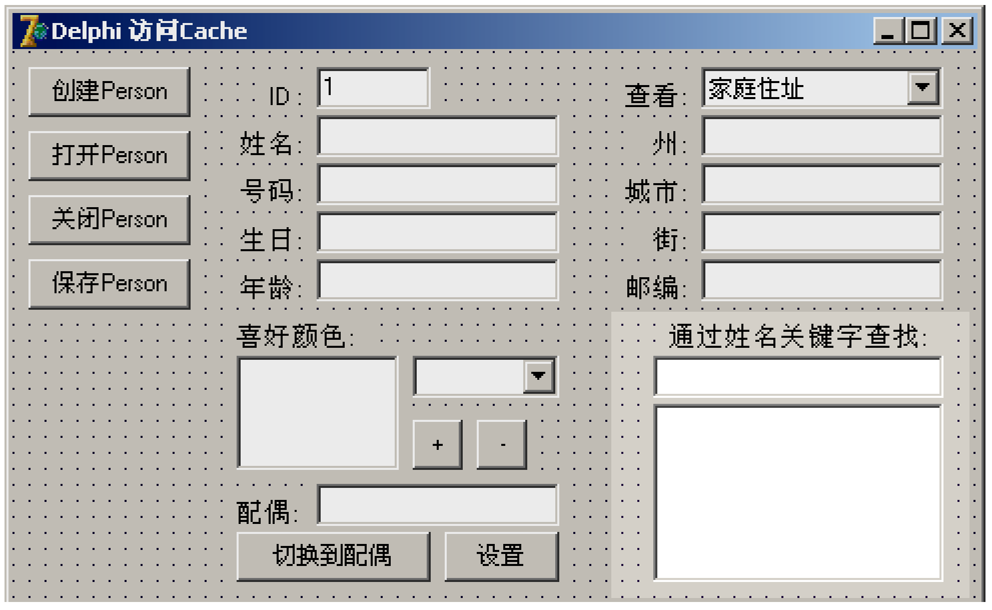
```
Query ByName(name As %String = "") As %SQLQuery(CONTAINID = 1,
ROWSPEC = "ID:%Integer,Name:%String(MAXLEN=30),DOB,SSN",…) [ … ]
{
 SELECT ID, Name, DOB, SSN
 FROM Sample.Person
 WHERE (Name %STARTSWITH :name)
 ORDER BY Name
}
```

该查询通过预定义的 SQL 语句在 Sample.Person 表(类的实例)中查找 Name 字段(属性)以参数 name 为开头(%STARTWITH 谓词)的记录，并添加该记录的 ID，Name，DOB，SSN 字段到结果集的新记录中，最后返回结果集。如果 name 为空值，则返回所有记录的相关信息。
例如：name 为 “Ja” 时，名为 “Jason”、“Jack”和“Jackson” 的记录都将被选中。

** 查询总是伴随着索引存在。由于在“WHERE”中用到了 Name 属性作比较，因此必须要为 Name 属性建立索引。Sample.Person 的作者已经为该属性建立了索引：NameIDX。*

然后，随着用户增加片断信息的内容(例如从“Ja”变为 “Jac”)，程序将会重新进行匹配，可选名单范围将越来越小，最后用户选择到需要找到的人(“Jackson”)，并显示出详细信息。

下面我们开始添加界面组件和代码：
在界面中添加一个输入框，命名为 ed_Key。它将用来输入人名的信息。
在界面中添加一个列表框，命名为 lb_NameList。它将用来显示候选的名单。
完成后的界面如下图所示：



为类 TForm_Main 添加一个私有方法声明，名为 UpdateNameList，用来从 ed_Key 中的信息匹配出符合的名单；

添加一个名为 FireUpdate 的 Boolean 型私有属性，用来做 ed_Key 的刷新标志；

添加一个名为 IdList 的 TStringList 型私有属性，用来保存候选名单的 Id 值。

代码如下：

```
…
 private
  { Private declarations }
  CacheFactory:IFactory;
  aPerson: Variant;
  aPersonAddress: Variant;
  FireUpdate: Boolean;
  IdList: TStringList;
  procedure ClosePerson;
  …
  procedure UpdateAddress;
```

```
    procedure UpdateNameList;
  public…
```

候选人的名字保存在 lb_NameList 中，便于用户查阅，由于是界面控件对象，系统会自动管理其生存期；候选人的 Id 号码保存在 IdList 中，用于选中后打开实例，由于是自定义对象，要手写代码管理其生存期。

在 FormCreate 方法实现部分的最后添加初始化的代码：

```
…
  else AddLog('已成功连接到 '+ConnectString);

  IdList := TStringList.Create;
  UpdateNameList;
  FireUpdate := True;
end;
```

在这里，生成了 IdList 的实例，之后便开始匹配(由于 ed_Key 为空，将返回所有实例的相关信息)，并设置更新标记为 True。

在 FormClose 方法实现部分的最开始处添加销毁 IdList 实例的代码：

```
procedure TForm_Main.FormClose(Sender: TObject; var Action:
TCloseAction);
begin
  IdList.Destroy;
  ClosePerson;
      …
end;
```

在 UpdateNameList 的实现部分输入以下代码：

```
procedure TForm_Main.UpdateNameList;
var
```

```
  ResultSet:IResultSet; // 声明结果集对象为 IResultSet 类型(前期绑定)
begin
  try
        // 从 Sample.Person 的预定义查询 ByName 产生结果集对象，并绑定到 IResultSet 类型
    ResultSet := CacheFactory.ResultSet('Sample.Person','ByName') as IResultSet;
        // 执行查询，输入唯一的参数：待匹配的关键字
    Variant(ResultSet).Execute(ed_Key.Text);
        // 清空列表
    lb_NameList.Clear;
    IdList.Clear;
        // 遍历结果集对象，并保存其中的姓名和 ID 信息到列表中
    while ResultSet.Next do begin
      lb_NameList.Items.Add(ResultSet.GetDataByName('Name'));
      IdList.Add(ResultSet.GetDataByName('ID'));
    end;
        ResultSet.Close;
  except
    on E: Exception do
      AddLog('错误: '+E.Message);
  end;
end;
```

这里用到了工厂对象的以下方法：

ResultSet *(const ClassName: WideString; const QueryName: WideString): IDispatch;*，将数据库中的查询返回成结果集，参数 ClassName 为类的全名(中间不能有空格)，QueryName 为查询的名称，返回 结果集对象实例引用；

由于在类型库中已经绑定了结果集的类型结构到 **IResultSet** 接口中(可以查看 CacheObject_TLB.pas 文件)，因此我们可以声明 ResultSet 变量为 IResultSet 类型，而不必用 Variant 类型。

工厂对象的 ResultSet 方法返回值为 IDispatch 类型，必须将其类型转换为 IResultSet 传给 ResultSet 变量。

调用 ResultSet 的 Execute 方法时，将在服务器端执行查询，并将结果返回输入结果集中。

** 在 IResultSet 定义中，Execute 方法有 16 个参数代表查询的相应参数集合(0 号到 15 号)。为了使程序代码简明(仅限于本句代码)，将其转换为 Variant 类型后再调用 Execute 方法，这样可以按照真实情况的需要(ByName 只需要一个参数)输入参数，不会输入多余的信息。*

用 ResultSet 的 Next 方法和 GetDataByName 方法分别去进行遍历和读取当前数据。最后，将 Name 和 Id 分别存入了相应列表中。

** GetDataByName 方法的参数将参照查询 ByName 定义时的 ROWSPEC 参数*

** 到达记录末尾时，Next 方法返回 False 值，否则为 True*

为输入框 ed_Key 的 OnChange 和 OnKeyUp 事件创建处理代码框架。

在 OnChange 事件的实现部分输入以下代码：

```
procedure TForm_Main.ed_KeyChange(Sender: TObject);
begin
  if FireUpdate then begin
    UpdateNameList;
    if lb_NameList.Count=1 then begin
      FireUpdate := False;
      ed_Key.Text := lb_NameList.Items.Strings[0];
      lb_NameList.ItemIndex := 0;
      Exit;
```

```
    end;
  end;
  FireUpdate := True;
end;
```

在 ed_Key 内容变化且允许更新时，去重新匹配名单。

如果只有一个候选人，则直接将其名字填入 ed_Key 中。这样会再次引发 ed_KeyChange 事件，所以将更新标志 FireUpdate 设为 False，即忽略下一个事件。

在 OnKeyUp 事件的实现部分输入以下代码：

```
procedure TForm_Main.ed_KeyKeyUp(Sender: TObject; var Key: Word;
 Shift: TShiftState);
begin
  if lb_NameList.Count=0 then begin
    if Key=VK_RETURN then AddLog('查无此人: '+ed_Key.Text);
    Exit;
  end;
  if Key=VK_DOWN then
  begin
    if (lb_NameList.ItemIndex < lb_NameList.Count-1) then
      lb_NameList.ItemIndex := lb_NameList.ItemIndex+1
    else
      lb_NameList.ItemIndex := 0;
    FireUpdate := False;
    ed_Key.Text := lb_NameList.Items.Strings[lb_NameList.ItemIndex];
  end else
  if Key=VK_UP then
  begin
    if (lb_NameList.ItemIndex > 0) then
```

```
        lb_NameList.ItemIndex := lb_NameList.ItemIndex-1
      else
        lb_NameList.ItemIndex := lb_NameList.Count-1;
      FireUpdate := False;
      ed_Key.Text := lb_NameList.Items.Strings[lb_NameList.ItemIndex];
    end else
    if Key=VK_RETURN then
    begin
      if lb_NameList.ItemIndex = -1 then begin
        FireUpdate := False;
        ed_Key.Text := lb_NameList.Items.Strings[0];
        lb_NameList.ItemIndex := 0;
        Exit;
      end;
      ed_Id.Text := IdList.Strings[lb_NameList.ItemIndex];
      btn_OpenPersonClick(Sender);
    end;
end;
```

当用户按下上箭头或下箭头时：将候选名单中相应的人名填入 ed_Key 中，并忽略下一次 ed_KeyChange 事件；

当用户按下回车键时：如果候选名单为空，则显示 “查无此人”；否则将显示出选中的用户信息；如果没有做选择，则将第一名候选人的名字填入 ed_Key 中，并忽略下一次 ed_KeyChange 事件。

为列表框 lb_NameList 的 OnClick 事件创建处理代码框架。

在 OnClick 事件的实现部分输入以下代码：

```
procedure TForm_Main.lb_NameListClick(Sender: TObject);
begin
  if lb_NameList.ItemIndex = -1 then Exit;
```

```
    FireUpdate := False;
    ed_Key.Text := lb_NameList.Items.Strings[lb_NameList.ItemIndex];
    ed_Id.Text := IdList.Strings[lb_NameList.ItemIndex];
    btn_OpenPersonClick(Sender);
end;
```

这样，在 lb_NameList 中在名单中做出选择时，通过取得其 Id，相应 Person 的信息就会显示出来。

保存工程：选择菜单 File->Save All。

测试运行：选择菜单 Run->Run。

程序执行后的界面如下图所示：

运行后，在关键字输入框中输入关键字(如 “Ja”)，

出现候选名单后，在输入框中按上或下键进行选择，

选中后在输入框中按回车键(或直接用鼠标在列表框中选取)，查看此人信息。

2．执行 SQL 语句

下面我们将可查询的范围扩大，采用直接调用 SQL 语句的方式。该新功能将从雇员(Employee)所在公司名称和 Person 本人或配偶的家庭住址邮编查找到可能的 Person：
通过输入公司名称的片断信息(如 “Int”)，以及邮编片断信息(如 “10”),程序将会自动匹配出符合的名单供选择。匹配过程中执行了相应的 SQL 语句。

** Employee 为 Person 的继承类(可以查看该类的类定义信息)。Employee 在 Person 基础上添加了一些属性，其中包括联系属性 Company (类似于引用属性 Spouse)，该属性指向了 Company 类型的持久类实例。在新安装的 Caché 上，Id 号码为 101-200 的 Person 实例是 Employee 类型的。之前，我们一直把它们当作 Person 实例来引用，这是符合面向对象的原则的。*

下面，为类 TForm_Main 添加一个私有方法声明，名为 UpdateNameList2，功能和 UpdateNameList 相似。它将用来从 ed_Key_Company 和 ed_Key_Zip 中的信息匹配出符合的名单；
代码如下：

```
   …
   procedure UpdateAddress;
   procedure UpdateNameList;
   procedure UpdateNameList2;
  public…
```

在 UpdateNameList2 的实现部分输入以下代码：

```
procedure TForm_Main.UpdateNameList2;
```

```
var
   ResultSet:IResultSet;
   SQL:String;
begin
   try
      if ed_Key_Company.Text='' then
         // 没有公司信息，则扩大范围：在 Person 表寻找
         // 自己地址的邮编或配偶的均可
         SQL:='SELECT ID,Name FROM Sample.Person WHERE '
            +'Home_Zip %STARTSWITH ? OR Spouse->Home_Zip
%STARTSWITH ?'
      else
         // 有公司信息，则直接在 Employee 表寻找
         SQL:='SELECT ID,Name FROM Sample.Employee WHERE '
            +'(Home_Zip %STARTSWITH ? OR Spouse->Home_Zip
%STARTSWITH ?) '
            +'AND Company->Name %STARTSWITH ?';
      // 从 SQL 语句得到结果集
           ResultSet := CacheFactory.DynamicSQL(SQL) as IResultSet;
                // 执行查询，输入参数
      Variant(ResultSet).Execute(ed_Key_Zip.Text,ed_Key_Zip.Text,
         ed_Key_Company.Text);
                // 清空列表
      lb_NameList.Clear;
      IdList.Clear;
                // 遍历结果集对象，并保存其中的姓名和 ID 信息到列表中
      while ResultSet.Next do begin
         lb_NameList.Items.Add(ResultSet.GetDataByName('Name'));
         IdList.Add(ResultSet.GetDataByName('ID'));
```

```
    end;
             ResultSet.Close;
  except
    on E: Exception do
      AddLog('错误: '+E.Message);
  end;
end;
```

这里用到了工厂对象的以下方法：

DynamicSQL *(const Statement: WideString): IDispatch;*，通过 SQL 语句访问数据库，返回结果集。参数 Statement 为 SQL 语句字串，返回 结果集对象实例引用；

这里的 SQL 语句用到了 Caché 扩展的语法(“->”)，这样就可以直接访问子属性，使语法更加简洁了。

** 传统的 SQL 语法同样可以使用。Caché 的 SQL 完全兼容 SQL92 标准。上面第二句 SQL 语句与以下语句作用完全相同：*

SELECT e.ID,e.Name FROM Sample.Employee e, Sample.Person s, Sample.Company c WHERE e.Spouse=s.ID AND e.Company=c.ID AND (e.Home_Zip %STARTSWITH ? OR s.Home_Zip %STARTSWITH ?) AND c.Name %STARTSWITH ?

在界面中添加一个输入框，命名为 ed_Key_Company。它将用来输入公司名称。

在界面中添加一个输入框，命名为 ed_Key_Zip。它将用来输入邮编。

完成后的界面如下图所示：

为输入框 ed_Key_Company 的 OnChange 事件创建处理代码框架。

在 OnChange 事件的实现部分输入以下代码：

```
procedure TForm_Main.ed_Key_CompanyChange(Sender: TObject);
begin
   UpdateNameList2;
end;
```

在 ed_Key_Company 内容变化时，重新匹配名单，通过新的匹配方法：UpdateNameList2。

同样，为输入框 ed_Key_Zip 的 OnChange 事件创建处理代码框架。

在 OnChange 事件的实现部分输入以下代码：

```
procedure TForm_Main.ed_Key_ZipChange(Sender: TObject);
begin
   UpdateNameList2;
```

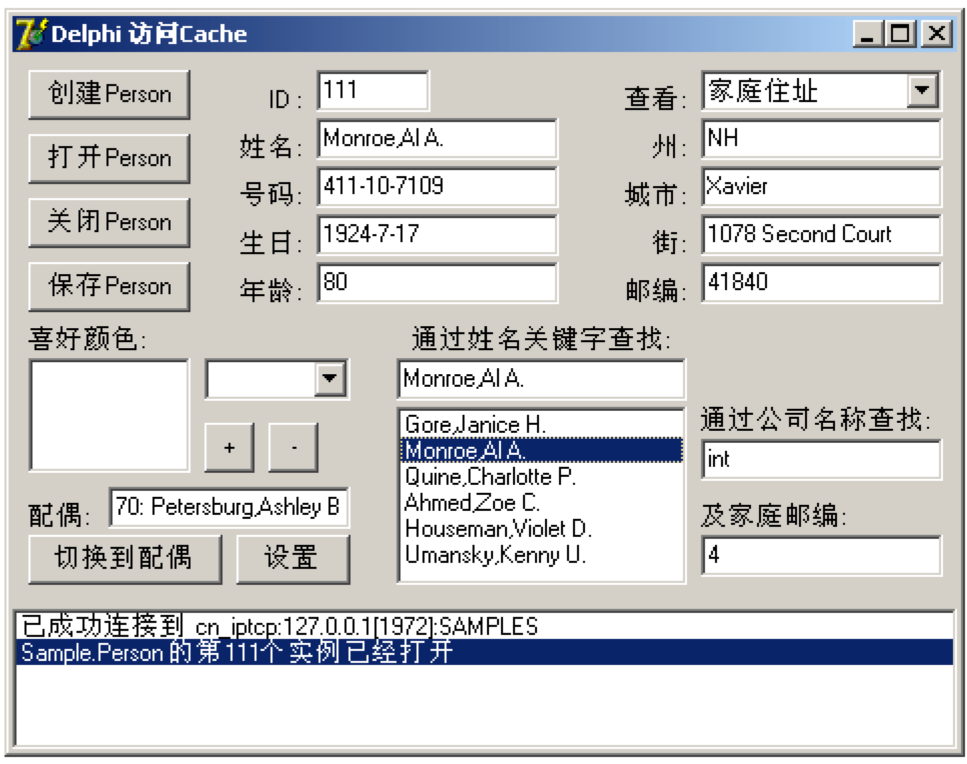
end;

在 ed_Key_Zip 内容变化时，重新匹配名单。

保存工程：选择菜单 File->Save All。

测试运行：选择菜单 Run->Run。

程序执行后的界面如下图所示：



运行后，在公司名称输入框中输入关键字(如 “Int”)，

也可以在邮编输入框中输入关键字(如 “4”)，

出现候选名单后，

在列表框中选取人名，查看此人信息。

测试完成。

在本单元中，我们尝试了在 Delphi 客户端执行 Caché 预定义查询和直接运行 SQL 语句两种查询方式。通过以上示例，我们已经可以在 Delphi 和 Caché 的基础上，通过片断信息查找到相应人员及其结构化的详细信息，并可以修改、、保存该对象实例，也可以建立和访问它和其它对象之间的关系。
可见，在开发过程中，通过将查询操作与面向对象操作结合起来，可以显著地提高客户端程序的开发效率。

1.8 执行 Caché 查询

下面我们将通过工厂对象执行预定义的查询或 SQL 语句。

开发人员对于 SQL 语言都是比较熟悉的，而且在一些情况下进行 SQL 查询也是更方便的做法。Caché 同时提供了对象和关系的访问方式，这使得我们可以不拘泥于单一的数据访问方式，从而增加了我们在编写应用程序时的灵活性。
我们可以直接通过工厂对象进行 SQL 查询，而不必通过 ODBC 接口。

1．执行预定义查询

下面我们将为程序添加新的功能，其中用到了在 Caché 类中预先定义并封装的查询语句。该新功能将从人名的片断信息查找到可能的 Person：
通过输入人名的片断信息(如 “Ja”)，程序将会自动匹配出符合的名单供选择。
匹配过程中执行了 Sample.Person 类的预定义查询 ByName。
在类定义中，它以如下形式声明：
*Query **ByName**(name As %String = "") As %SQLQuery(CONTAINID = 1, ROWSPEC = "ID:%Integer,Name:%String(MAXLEN=30),DOB,SSN",…) […]*

```
{
 SELECT ID, Name, DOB, SSN
 FROM Sample.Person
 WHERE (Name %STARTSWITH :name)
 ORDER BY Name
}
```

该查询通过预定义的 SQL 语句在 Sample.Person 表(类的实例)中查找 Name 字段(属性)以参数 name 为开头(%STARTWITH 谓词)的记录，并添加该记录的 ID，Name，DOB，SSN 字段到结果集的新记录中，最后返回结果集。如果 name 为空值，则返回所有记录的相关信息。

例如：name 为 “Ja” 时，名为 “Jason”、“Jack”和“Jackson” 的记录都将被选中。

** 查询总是伴随着索引存在。由于在“WHERE”中用到了 Name 属性作比较，因此必须要为 Name 属性建立索引。Sample.Person 的作者已经为该属性建立了索引：NameIDX。*

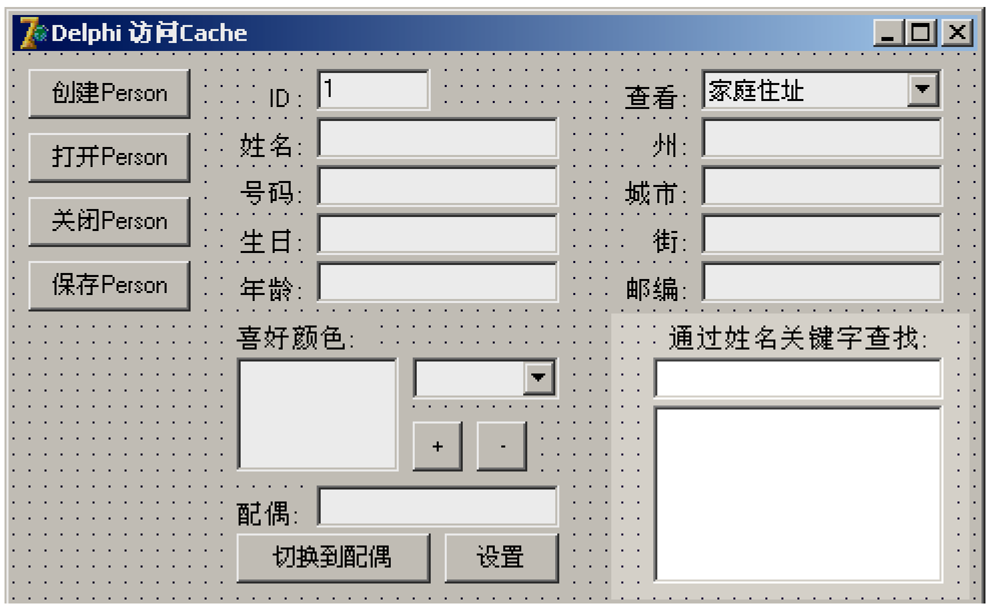
然后，随着用户增加片断信息的内容(例如从“Ja”变为 “Jac”)，程序将会重新进行匹配，可选名单范围将越来越小，最后用户选择到需要找到的人(“Jackson”)，并显示出详细信息。

下面我们开始添加界面组件和代码：

在界面中添加一个输入框，命名为 ed_Key。它将用来输入人名的信息。

在界面中添加一个列表框，命名为 lb_NameList。它将用来显示候选的名单。

完成后的界面如下图所示：



为类 TForm_Main 添加一个私有方法声明，名为 UpdateNameList，用来从 ed_Key 中的信息匹配出符合的名单；

添加一个名为 FireUpdate 的 Boolean 型私有属性，用来做 ed_Key 的刷新标志；

添加一个名为 IdList 的 TStringList 型私有属性，用来保存候选名单的 Id 值。

代码如下：

```
…
 private
   { Private declarations }
   CacheFactory:IFactory;
   aPerson: Variant;
   aPersonAddress: Variant;
   FireUpdate: Boolean;
   IdList: TStringList;
   procedure ClosePerson;
   …
   procedure UpdateAddress;
```

```
    procedure UpdateNameList;
  public…
```

候选人的名字保存在 lb_NameList 中，便于用户查阅，由于是界面控件对象，系统会自动管理其生存期；候选人的 Id 号码保存在 IdList 中，用于选中后打开实例，由于是自定义对象，要手写代码管理其生存期。

在 FormCreate 方法实现部分的最后添加初始化的代码：

```
…
  else AddLog('已成功连接到 '+ConnectString);

  IdList := TStringList.Create;
  UpdateNameList;
  FireUpdate := True;
end;
```

在这里，生成了 IdList 的实例，之后便开始匹配(由于 ed_Key 为空，将返回所有实例的相关信息)，并设置更新标记为 True。

在 FormClose 方法实现部分的最开始处添加销毁 IdList 实例的代码：

```
procedure TForm_Main.FormClose(Sender: TObject; var Action:
TCloseAction);
begin
  IdList.Destroy;
  ClosePerson;
       …
end;
```

在 UpdateNameList 的实现部分输入以下代码：

```
procedure TForm_Main.UpdateNameList;
var
```

```
  ResultSet:IResultSet; // 声明结果集对象为 IResultSet 类型(前期绑定)
begin
  try
          // 从 Sample.Person 的预定义查询 ByName 产生结果集对象，并绑定到 IResultSet 类型
    ResultSet := CacheFactory.ResultSet('Sample.Person','ByName') as IResultSet;
          // 执行查询，输入唯一的参数：待匹配的关键字
    Variant(ResultSet).Execute(ed_Key.Text);
          // 清空列表
    lb_NameList.Clear;
    IdList.Clear;
          // 遍历结果集对象，并保存其中的姓名和 ID 信息到列表中
    while ResultSet.Next do begin
      lb_NameList.Items.Add(ResultSet.GetDataByName('Name'));
      IdList.Add(ResultSet.GetDataByName('ID'));
    end;
          ResultSet.Close;
  except
    on E: Exception do
      AddLog('错误: '+E.Message);
  end;
end;
```

这里用到了工厂对象的以下方法：

ResultSet *(const ClassName: WideString; const QueryName: WideString): IDispatch;*，将数据库中的查询返回成结果集，参数 ClassName 为类的全名(中间不能有空格)，QueryName 为查询的名称，返回 结果集对象实例引用；

由于在类型库中已经绑定了结果集的类型结构到 **IResultSet** 接口中(可以查看 CacheObject_TLB.pas 文件)，因此我们可以声明 ResultSet 变量为 IResultSet 类型，而不必用 Variant 类型。

工厂对象的 ResultSet 方法返回值为 IDispatch 类型，必须将其类型转换为 IResultSet 传给 ResultSet 变量。

调用 ResultSet 的 Execute 方法时，将在服务器端执行查询，并将结果返回输入结果集中。

** 在 IResultSet 定义中，Execute 方法有 16 个参数代表查询的相应参数集合(0 号到 15 号)。为了使程序代码简明(仅限于本句代码)，将其转换为 Variant 类型后再调用 Execute 方法，这样可以按照真实情况的需要(ByName 只需要一个参数)输入参数，不会输入多余的信息。*

用 ResultSet 的 Next 方法和 GetDataByName 方法分别去进行遍历和读取当前数据。最后，将 Name 和 Id 分别存入了相应列表中。

** GetDataByName 方法的参数将参照查询 ByName 定义时的 ROWSPEC 参数*

** 到达记录末尾时，Next 方法返回 False 值，否则为 True*

为输入框 ed_Key 的 OnChange 和 OnKeyUp 事件创建处理代码框架。

在 OnChange 事件的实现部分输入以下代码：

```
procedure TForm_Main.ed_KeyChange(Sender: TObject);
begin
   if FireUpdate then begin
      UpdateNameList;
      if lb_NameList.Count=1 then begin
         FireUpdate := False;
         ed_Key.Text := lb_NameList.Items.Strings[0];
         lb_NameList.ItemIndex := 0;
         Exit;
```

```
    end;
  end;
  FireUpdate := True;
end;
```

在 ed_Key 内容变化且允许更新时，去重新匹配名单。

如果只有一个候选人，则直接将其名字填入 ed_Key 中。这样会再次引发 ed_KeyChange 事件，所以将更新标志 FireUpdate 设为 False，即忽略下一个事件。

在 OnKeyUp 事件的实现部分输入以下代码：

```
procedure TForm_Main.ed_KeyKeyUp(Sender: TObject; var Key: Word;
 Shift: TShiftState);
begin
  if lb_NameList.Count=0 then begin
    if Key=VK_RETURN then AddLog('查无此人: '+ed_Key.Text);
    Exit;
  end;
  if Key=VK_DOWN then
  begin
    if (lb_NameList.ItemIndex < lb_NameList.Count-1) then
      lb_NameList.ItemIndex := lb_NameList.ItemIndex+1
    else
      lb_NameList.ItemIndex := 0;
    FireUpdate := False;
    ed_Key.Text := lb_NameList.Items.Strings[lb_NameList.ItemIndex];
  end else
  if Key=VK_UP then
  begin
    if (lb_NameList.ItemIndex > 0) then
```

```
      lb_NameList.ItemIndex := lb_NameList.ItemIndex-1
    else
      lb_NameList.ItemIndex := lb_NameList.Count-1;
    FireUpdate := False;
    ed_Key.Text := lb_NameList.Items.Strings[lb_NameList.ItemIndex];
  end else
  if Key=VK_RETURN then
  begin
    if lb_NameList.ItemIndex = -1 then begin
      FireUpdate := False;
      ed_Key.Text := lb_NameList.Items.Strings[0];
      lb_NameList.ItemIndex := 0;
      Exit;
    end;
    ed_Id.Text := IdList.Strings[lb_NameList.ItemIndex];
    btn_OpenPersonClick(Sender);
  end;
end;
```

当用户按下上箭头或下箭头时：将候选名单中相应的人名填入 ed_Key 中，并忽略下一次 ed_KeyChange 事件；

当用户按下回车键时：如果候选名单为空，则显示 “查无此人”；否则将显示出选中的用户信息；如果没有做选择，则将第一名候选人的名字填入 ed_Key 中，并忽略下一次 ed_KeyChange 事件。

为列表框 lb_NameList 的 OnClick 事件创建处理代码框架。

在 OnClick 事件的实现部分输入以下代码：

```
procedure TForm_Main.lb_NameListClick(Sender: TObject);
begin
  if lb_NameList.ItemIndex = -1 then Exit;
```

```
    FireUpdate := False;
    ed_Key.Text := lb_NameList.Items.Strings[lb_NameList.ItemIndex];
    ed_Id.Text := IdList.Strings[lb_NameList.ItemIndex];
    btn_OpenPersonClick(Sender);
end;
```

这样，在 lb_NameList 中在名单中做出选择时，通过取得其 Id，相应 Person 的信息就会显示出来。

保存工程：选择菜单 File->Save All。

测试运行：选择菜单 Run->Run。

程序执行后的界面如下图所示：

运行后，在关键字输入框中输入关键字(如 “Ja”)，

出现候选名单后，在输入框中按上或下键进行选择，

选中后在输入框中按回车键(或直接用鼠标在列表框中选取)，查看此人信息。

2．执行 SQL 语句

下面我们将可查询的范围扩大，采用直接调用 SQL 语句的方式。该新功能将从雇员(Employee)所在公司名称和 Person 本人或配偶的家庭住址邮编查找到可能的 Person：
通过输入公司名称的片断信息(如 “Int”)，以及邮编片断信息(如 “10”),程序将会自动匹配出符合的名单供选择。匹配过程中执行了相应的 SQL 语句。

** Employee 为 Person 的继承类(可以查看该类的类定义信息)。Employee 在 Person 基础上添加了一些属性，其中包括联系属性 Company (类似于引用属性 Spouse)，该属性指向了 Company 类型的持久类实例。在新安装的 Caché 上，Id 号码为 101-200 的 Person 实例是 Employee 类型的。之前，我们一直把它们当作 Person 实例来引用，这是符合面向对象的原则的。*

下面，为类 TForm_Main 添加一个私有方法声明，名为 UpdateNameList2，功能和 UpdateNameList 相似。它将用来从 ed_Key_Company 和 ed_Key_Zip 中的信息匹配出符合的名单；
代码如下：

```
   …
   procedure UpdateAddress;
   procedure UpdateNameList;
   procedure UpdateNameList2;
  public…
```

在 UpdateNameList2 的实现部分输入以下代码：

```
procedure TForm_Main.UpdateNameList2;
```

```
var
   ResultSet:IResultSet;
   SQL:String;
begin
   try
      if ed_Key_Company.Text='' then
         // 没有公司信息，则扩大范围：在 Person 表寻找
         // 自己地址的邮编或配偶的均可
         SQL:='SELECT ID,Name FROM Sample.Person WHERE '
            +'Home_Zip %STARTSWITH ? OR Spouse->Home_Zip
%STARTSWITH ?'
      else
         // 有公司信息，则直接在 Employee 表寻找
         SQL:='SELECT ID,Name FROM Sample.Employee WHERE '
            +'(Home_Zip %STARTSWITH ? OR Spouse->Home_Zip
%STARTSWITH ?) '
            +'AND Company->Name %STARTSWITH ?';
      // 从 SQL 语句得到结果集
           ResultSet := CacheFactory.DynamicSQL(SQL) as IResultSet;
                // 执行查询，输入参数
      Variant(ResultSet).Execute(ed_Key_Zip.Text,ed_Key_Zip.Text,
         ed_Key_Company.Text);
                // 清空列表
      lb_NameList.Clear;
      IdList.Clear;
                // 遍历结果集对象，并保存其中的姓名和 ID 信息到列表中
      while ResultSet.Next do begin
         lb_NameList.Items.Add(ResultSet.GetDataByName('Name'));
         IdList.Add(ResultSet.GetDataByName('ID'));
```

```
    end;
              ResultSet.Close;
  except
    on E: Exception do
      AddLog('错误: '+E.Message);
  end;
end;
```

这里用到了工厂对象的以下方法：

DynamicSQL *(const Statement: WideString): IDispatch;*，通过 SQL 语句访问数据库，返回结果集。参数 Statement 为 SQL 语句字串，返回 结果集对象实例引用；

这里的 SQL 语句用到了 Caché 扩展的语法(“->”)，这样就可以直接访问子属性，使语法更加简洁了。

** 传统的 SQL 语法同样可以使用。Caché 的 SQL 完全兼容 SQL92 标准。上面第二句 SQL 语句与以下语句作用完全相同：*

SELECT e.ID,e.Name FROM Sample.Employee e, Sample.Person s, Sample.Company c WHERE e.Spouse=s.ID AND e.Company=c.ID AND (e.Home_Zip %STARTSWITH ? OR s.Home_Zip %STARTSWITH ?) AND c.Name %STARTSWITH ?

在界面中添加一个输入框，命名为 ed_Key_Company。它将用来输入公司名称。

在界面中添加一个输入框，命名为 ed_Key_Zip。它将用来输入邮编。

完成后的界面如下图所示：

为输入框 ed_Key_Company 的 OnChange 事件创建处理代码框架。

在 OnChange 事件的实现部分输入以下代码：

```
procedure TForm_Main.ed_Key_CompanyChange(Sender: TObject);
begin
   UpdateNameList2;
end;
```

在 ed_Key_Company 内容变化时，重新匹配名单，通过新的匹配方法：UpdateNameList2。

同样，为输入框 ed_Key_Zip 的 OnChange 事件创建处理代码框架。

在 OnChange 事件的实现部分输入以下代码：

```
procedure TForm_Main.ed_Key_ZipChange(Sender: TObject);
begin
   UpdateNameList2;
```

end;

在 ed_Key_Zip 内容变化时，重新匹配名单。

保存工程：选择菜单 File->Save All。

测试运行：选择菜单 Run->Run。

程序执行后的界面如下图所示：

运行后，在公司名称输入框中输入关键字(如 “Int”)，

也可以在邮编输入框中输入关键字(如 “4”)，

出现候选名单后，

在列表框中选取人名，查看此人信息。

测试完成。

在本单元中，我们尝试了在 Delphi 客户端执行 Caché 预定义查询和直接运行 SQL 语句两种查询方式。通过以上示例，我们已经可以在 Delphi 和 Caché 的基础上，通过片断信息查找到相应人员及其结构化的详细信息，并可以修改、保存该对象实例，也可以建立和访问它和其它对象之间的关系。
可见，在开发过程中，通过将查询操作与面向对象操作结合起来，可以显著地提高客户端程序的开发效率。

1.9 访问 Caché 对象的特殊属性

下面我们将通过前期绑定的接口访问 Caché 中的流和列表。

在 Caché 中，大的二进制数据(Binary Large Object, BLOB)和长的字符型数据都是以%Stream 类型表示。在客户端，可以用流的相关标准函数(如 Read,Write)来访问其中的数据。Caché 为对象的流属性提供了前期绑定的支持(专门的接口用来处理两种流)，这样可以更加方便地进行操作。同样，对于 Caché 中的%List 类型，在客户端也提供了前期绑定的支持。

** 在 SAMPLES 命名空间的 Sample.Person 类中，没有定义这几种类型的属性。我们可以参照单元：应用开发->开发前准备 来为 Person 添加这些属性。否则程序会出现运行时错误。*

下面我们将为程序添加如下新功能：
1.查看和编写 Person 的简历(Resume)：
　　访问 Caché 中保存着简历信息的字符流。
2.查看和更改 Person 的照片(Picture)：

访问 Caché 中保存着照片图像的二进制流。

3.查看 Person 的目录列表内容(DirTree)：

访问 Caché 中创建的可结构化的%List 类型。

下面，我们要为这些新功能分别创建相应的控件：

1.在界面中添加一个 Memo 控件，命名为 “mm_CharStream”， 它用来显示简历内容。

其中，需要将它的 ScrollBars 属性设为 ssBoth，

并清空 Lines 属性的内容。

2.在界面中添加一个 Image 控件，命名为 “Im_BinaryStream”，它用来显示照片。

其中，需要将它的：

Proportional 属性设为 True，保持图像纵横比例；

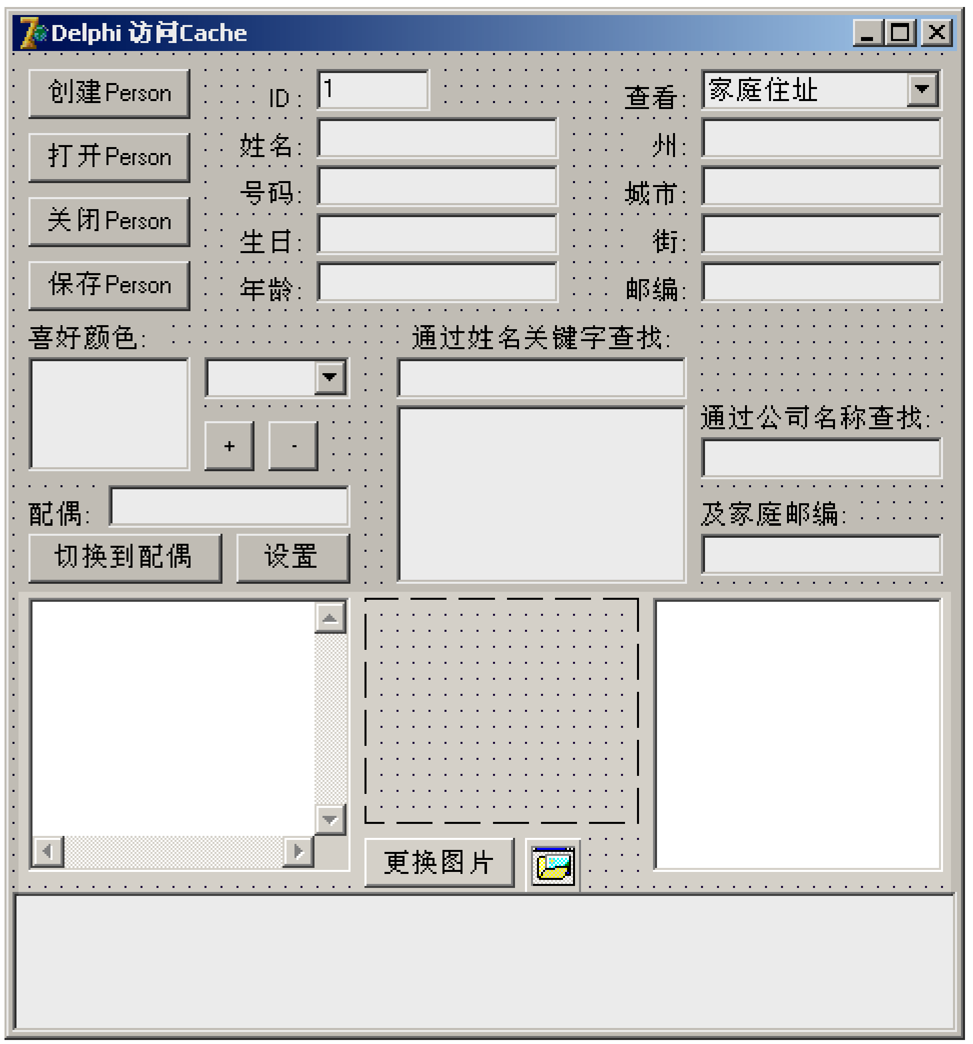
Stretch 属性设为 True，自动调整尺寸。

添加一个按钮，命名为 “btn_Picture”。

添加一个 OpenPictureDialog 类型的对话框控件，命名为 “OPD”。

3.在界面中添加一个 TreeView 控件，命名为 “tv_Syslist”，它用来显示目录列表的内容。

添加这些控件之后，窗体外观如下图所示：



下面开始添加代码：

为 Unit_Main 添加新的单元引用：AxCtrls 和 ActiveX，这两个单元都是只和显示照片的功能相关。

完成后的代码如下：

```
uses
  …
  Dialogs, CacheObject_TLB, StdCtrls, Buttons, ExtCtrls, ExtDlgs, ComCtrls,
  AxCtrls, ActiveX;
```

```
type
…
```

为 TForm_Main 添加新的私有方法 DisplaySpecial，用来显示这些特殊属性的内容到界面中。

为 TForm_Main 添加新的私有方法 UpdateSpecial，用来将界面信息返回到这些特殊属性中。

完成后的代码如下：

```
      …
  procedure UpdateNameList;
  procedure UpdateNameList2;
  procedure DisplaySpecial;
  procedure UpdateSpecial;
 public{ Public declarations }
…
```

为方法 DisplayPerson 实现部分的最后补充输入以下代码：

```
  end else
    ed_Spouse.Text := '';

  DisplaySpecial;
end;
```

为方法 UpdatePerson 实现部分的最后补充输入以下代码：

```
  aPerson.DOB := ed_DOB.Text;
  UpdateAddress;
  UpdateSpecial;
end;
```

为方法 DisplaySpecial 的实现部分输入以下代码：

```
procedure TForm_Main.DisplaySpecial;
var
  Resume: ICharStream;
  Picture: IBinaryStream;
  PictureDisp: IPictureDisp;
  DirTree: ISyslist;

      // 内嵌递归函数
  procedure AssignSyslist(Syslist:ISyslist;Parent:TTreeNode);
  var
    i:Integer;
    ItemList:ISyslist;
  begin
    For i:=1 to Syslist.Count do begin
      ItemList := Syslist.ItemList[i] as ISyslist;
      if ItemList.Count>0 then
        AssignSyslist(ItemList,tv_Syslist.Items.AddChild(Parent,'_子节点'))
      else tv_Syslist.Items.AddChild(Parent,Syslist.Item[i]);
    end;
  end;

begin
  // 将 Resume 属性绑定成 ICharStream 类型
  Resume := IDispatch(aPerson.Resume) as ICharStream;
  // 清空文字控件
  mm_CharStream.Clear;
  // 设置文字
  mm_CharStream.Lines.Add(Resume.Data);
```

```
  // 清空图片控件
  Im_BinStream.Picture.Assign(nil);
  // 将 Picture 属性绑定成 IBinaryStream 类型
  Picture := IDispatch(aPerson.Picture) as IBinaryStream;
  // 设置图片：先得到 IPictureDisp 类型的指针，再设置到 TPicture 类型中
  PictureDisp := Picture.GetPicture;
  SetOlePicture(Im_BinStream.Picture,PictureDisp);

      // 清空 TreeView
  tv_Syslist.Items.Clear;
  // 将 DirTree 属性绑定成 ISyslist 类型
  DirTree := IDispatch(aPerson.DirTree) as ISyslist;
      // 递归地将 Syslist 中的结构化信息显示到 TreeView 控件中
  AssignSyslist(DirTree,tv_Syslist.Items.AddChild(nil,'_根节点'));
      // 展开节点
  tv_Syslist.Items.GetFirstNode.Expand(True);
end;
```

为方法 UpdateSpecial 的实现部分输入以下代码：

```
procedure TForm_Main.UpdateSpecial;
var
  Resume: ICharStream;
  Picture: IBinaryStream;
  PictureDisp: IPictureDisp;
begin
  // 将 Resume 属性绑定成 ICharStream 类型
  Resume := IDispatch(aPerson.Resume) as ICharStream;
```

```
  // 从控件读取文字
  Resume.Data := mm_CharStream.Lines.Text;

  // 将 Picture 属性绑定成 IBinaryStream 类型
  Picture := IDispatch(aPerson.Picture) as IBinaryStream;
  // 读取图片：先得到 IPictureDisp 类型的指针，再设置到 IBinaryStream 类
型中
  GetOlePicture(Im_BinStream.Picture,PictureDisp);
  Picture.SetPicture(PictureDisp);
end;
```

下面，为按钮 btn_Picture 的 OnClick 事件创建处理代码框架。

在 OnClick 事件的实现部分输入以下代码：

```
procedure TForm_Main.btn_PictureClick(Sender: TObject);
begin
  if OPD.Execute then
    Im_BinStream.Picture.LoadFromFile(OPD.FileName);
end;
```

保存工程：选择菜单 File->Save All。

测试运行：选择菜单 Run->Run。

运行后，选择新建或打开对象，

查看目录列表(DirTree)信息，

(要使目录列表属性中存在数据，可提前在 Terminal 中为该对象的 DirTree 属性进行设置，例如：Set aPerson.DirTree=$lb(“a”,”b”,$lb(“c1”,”c2”,$lb(“p”,”q”)),”d”)，并保存：do aPerson.%Save()。下一单元我们将能够直接在程序中修改该列表属性。)

输入简历信息，

点击按钮“更换图片”，从文件目录中选择图片作为照片。

保存对象。

尝试再次打开此对象，并检查各项数据读取的结果。

程序执行后的界面如下图所示：

在本单元中，我们已经尝试了在 Delphi 客户端访问特殊属性：二进制流、字符流和列表的方法。这样，在遇到存储二进制文件、长字符串或简单结构化列表信息的情况下，我们都可以采用相应的办法去处理。下一单元我们将在 Delphi 中以终端命令的方式访问 Caché。

1.10 访问 Caché Direct

下面我们将通过另一个组件来以终端命令方式直接控制 Caché。

前面几个单元的操作都是应用在 CacheObject 这个组件上的，下面我们将使用 Caché 提供的另外一个组件：VisM。VisM 是一个 ActiveX 组件，可以让 Delphi 客户端程序以 Caché 终端的方式工作，即可以运行命令行、调用例程并返回结果等。这就是 **Caché Direct** 技术。

1.安装 **VisM** 组件

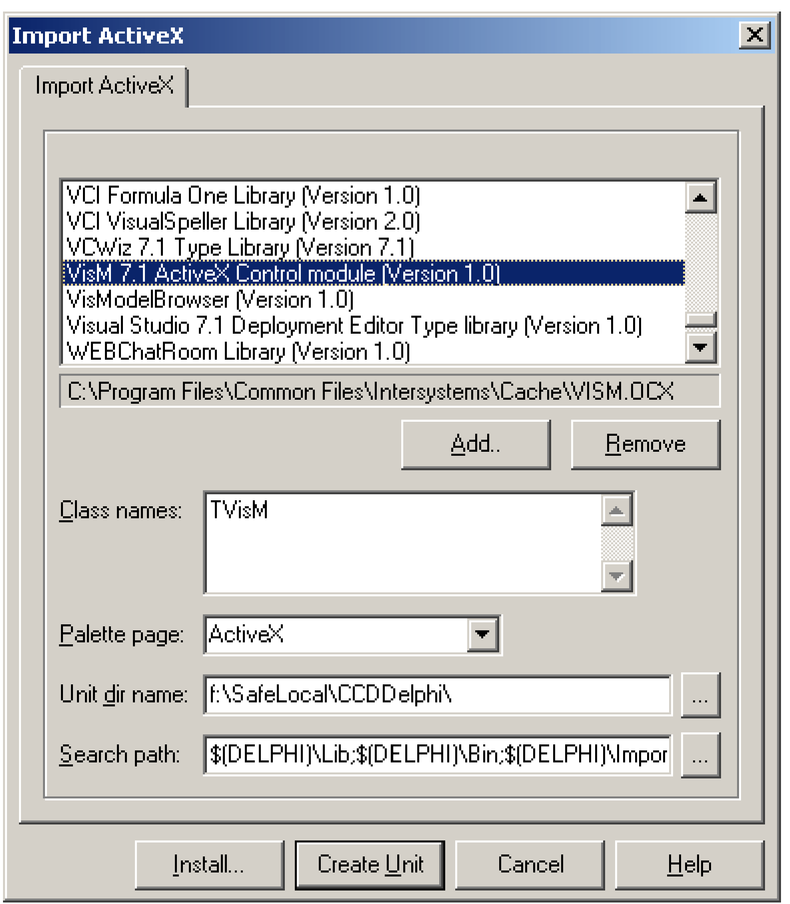
由于 VisM 是 ActiveX 组件，因此在使用它之前要在 Delphi 工程中进行安装。

我们将要导入 VisM 组件的类型库到 Delphi 中。Delphi 会自动创建该组件类型库的前期绑定文件。

在 Delphi 中，

选择菜单 Component->Import ActiveX Control…。

弹出 ” Import ActiveX Control”(导入 ActiveX 控件) 窗口。



在列表中选择名为 “VisM 7.1 ActiveX Control module [Version 1.0]” 的控件。

可以看到 “Class names” 栏中显示了其中包含的类：TVisM。

将 ”Unit dir name” 栏的内容修改为 “c:\ DelphiCache”(工程文件所在目录)。

点击按钮 “Install…”。

此时，在 Delphi 组件栏的 ActiveX 标签中将新增一个名为 ”VisM” 的新组件，

图标为一个立方体的形状。

点选它，并将它放置到窗体 Form_Main 上，

并重新命名为 “VisM”，

修改属性 NameSpace 为 “SAMPLES”。

2.在程序中添加 Caché Direct

我们将在 Delphi 工程中添加 VisM 组件的功能

我们将为程序添加如下新功能：

修改 Person 的目录列表内容(DirTree)：

通过 VisM 输入 COS 命令直接操作服务器进行修改。

下面开始添加界面控件：

在界面中添加一个输入框，命名为 “ed_DirTree”，它用来用 COS 的$LB 函数编辑列表内容。其中，需要将它的 Text 属性预先设为 “$lb()”。

在界面中添加一个按钮，命名为 “btn_DirTree” ，并将 Caption 属性设为 “设置”。

添加这些控件之后，窗体外观如下图所示：

下面开始添加代码：

为按钮 btn_DirTree 的 OnClick 事件创建处理代码框架。

在 OnClick 事件的实现部分输入以下代码：

```
procedure TForm_Main.btn_DirTreeClick(Sender: TObject);
begin
  if VarIsNull(aPerson) or VarIsClear(aPerson) then Exit;
  if ed_Id.Text='' then Exit;
  try
    // 执行 COS 改变 DirTree 属性
    VisM.Execute('Kill');
    VisM.Execute('Set p=##class(Sample.Person).%OpenId('+ed_Id.Text+')');
    VisM.Execute('Set p.DirTree='+ed_DirTree.Text);
    VisM.Execute('Set r=p.%Save()');
    VisM.Execute('$Get(r)');
```

```
    if VisM.VALUE='1' then AddLog('属性 DirTree 设置成功')
    else AddLog('属性 DirTree 设置失败');
    VisM.Execute('Kill');
    // 刷新界面
    btn_OpenPersonClick(Sender);
  except
    on E: Exception do
      AddLog('错误: '+E.Message);
  end;
end;
```

以上代码相当于在客户端运行了服务器端脚本：

Kill

// 打开实例

*Set p=##class(Sample.Person).%OpenId(**ID**)*

// 设置列表属性

*Set p.DirTree=**LIST***

// 保存实例

Set r=p.%Save()

返回$Get(r) // 用于客户端查看返回值 r

Kill

该脚本的作用上是将 ed_DirTree 中用于创建列表的字串("$lb(…)")输入到相应 Person 实例(ed_Id 中保存着 Id 号码)的 DirTree 属性中去。

可见，通过 VisM 的 Execute 方法，我们既可以执行终端命令，也可以调用例程和函数并取得返回值。

保存工程：选择菜单 File->Save All。

测试运行：选择菜单 Run->Run。

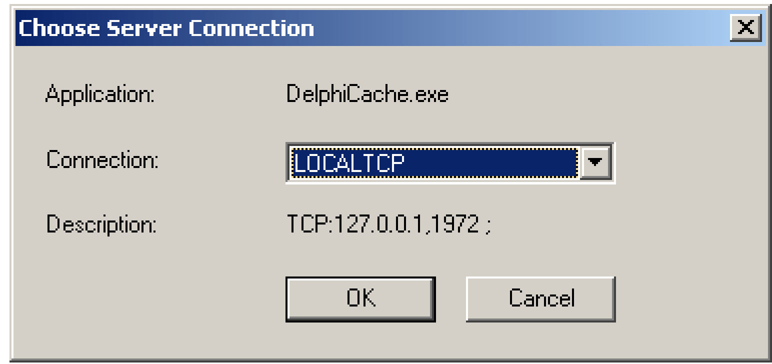
运行后，选择打开对象，

在 ed_DirTree 中输入新的目录列表属性信息，

(如$lb(“a”,”b”,$lb(“c1”,”c2”,$lb(“p”,”q”)),”d”))

点击按钮“设置”，

弹出“Choose Server Connection”对话框，



保持默认的选择“LOCALTCP”，按“OK”。

完成后，即可查看目录列表(DirTree)信息和日志。

程序执行后的界面如下图所示：

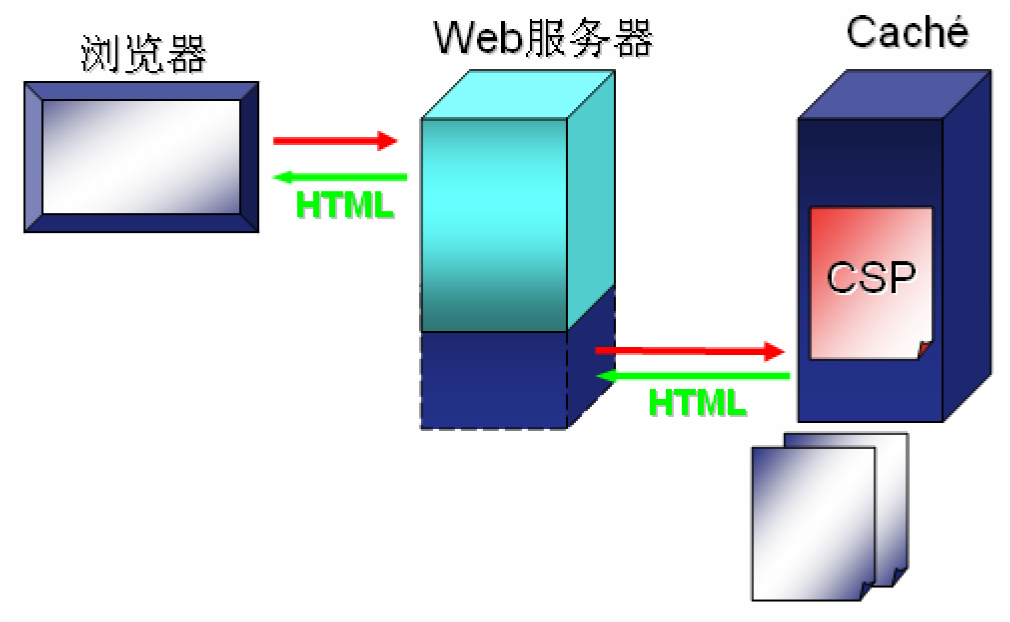
在本单元中，我们已经尝试了在 Delphi 客户端直接用终端命令控制 Caché 的方法，这一特性又大大提高了我们在开发时的灵活性。利用它可以直接调用例程，进而可以进行系统级别的操作；同时，这也难免会带来一些风险。因此，Caché Direct 技术一定要谨慎使用。

第七章 以 CSP 开发 Web 应用程序

1 引言

为了适应现在越来越多的 Web 应用程序的需要，Caché 也提供了自己的 Caché Server Page（CSP）技术。CSP 让你能够建立和部署高性能的、高可伸缩性的 Web 应用。开发人员可以通过两种方式来开发 Web 应用，一种是使用 Caché 类建立对象框架来创建动态网页，一种是通过基于 HTML 的标记语言把 Caché 脚本嵌入到 HTML 中去。当然，开发人员可以在一个应用中同时使用者两种方法。

CSP 的各个请求都是利用标准的 Web 服务器和 HTML 协议处理的。当一个 HTML 的客户端，通常是一个 Web 浏览器，通过 HTML 向 Web 服务器发出请求的时候，如果 Web 服务器认为这个请求是 CSP 的请求，那么 Web 服务器会把这个请求转送到 Caché，在 Caché 中正在运行的 CSP 服务便处理这个请求并且送回一个页面给 Web 服务器，再由这个 Web 服务器把该页面送给发出请求的浏览器。CSP 不仅管理着 Web 服务器和 Caché 之间的通讯，并且引用应用程序代码来生成页面。



Web 服务器和 Caché 服务器都是抽象的组件，它们可以被部署在一台或者多台计算机上。在开发期间，所有的组件都可以在同一台计算机上。在大规模的系统部署时则可以根据需要部署成为多台 Web 服务器和 Caché 服务器位于两层或者三层的体系结构中。

说明：为了使本教程的例子简化一些，我们把所有的组件都当成在一台计算机上来处理。

1.1 起步

为了有效使用 CSP，你应当对下列内容有一定的了解：

- Caché 对象和 Caché ObjectScript
- HTML
- JavaScript
- SQL

1.1.1 CSP 示例

Caché 带有一些原先已建立好的 CSP 示例页面，你可以通过下面的手段看到这些页面：

1. 启动 Caché
2. 启动浏览器并且访问 http://localhost:1972/csp/samples/menu.csp

，应当显示出一个带有简短说明的各种示例页面。

1.1.2 CSP 培训教材

Caché（仅限于 Windows 版本）带有一个详细的联机培训教材。这个教材包括有一个使用 Caché 和 CSP 的应用程序。要运行这个教材，点击 Caché “立方体”，点弹出菜单中的“Documentation”，然后进入“Caché Tutorials”里面的“Caché Web Applications Tutorial”。

1.1.3 你的第一个 CSP 应用程序

用建立“Hello World”网页作为例子是常见的，如果没有“Hello World”这个例子，对于任何一个介绍建立网页的技术文档来说可能都不是完整的。本小节包含一个如何利用两种不同的办法来建立一个“Hello World”网

页的示例。我们首先介绍通过建立一个对象来输出页面的方法，然后介绍用标记过的 HTML 建立相同页面的办法：

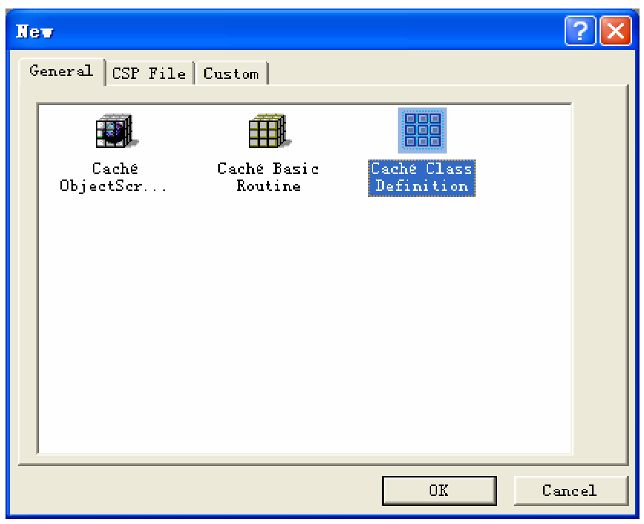
1.1.3.1　　基于类的方法

你可以借助于建立一个继承自%CSP.Page 类的类，并且覆盖它的 OnPage 方法，来建立动态的页面。任何通过这个方法写到主要设备上的输出都被自动地送到 Web 浏览器并被作为 Web 页面显示出来。

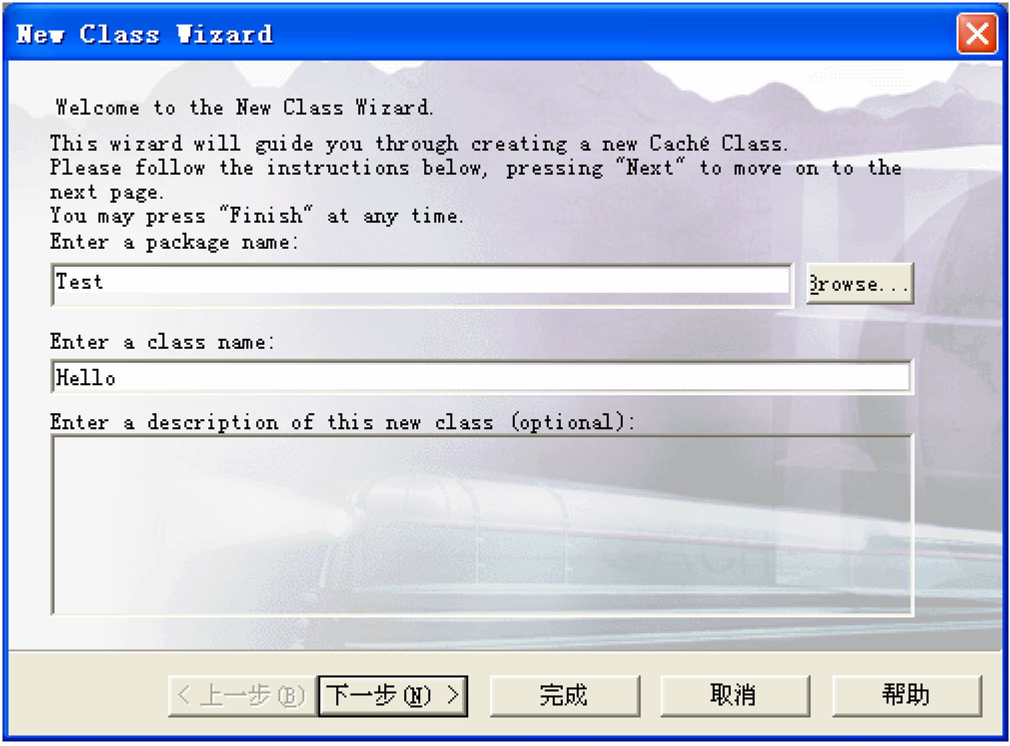
请按照如下的方法来做：

1.启动 Caché Studio，切换到 User 命名空间。

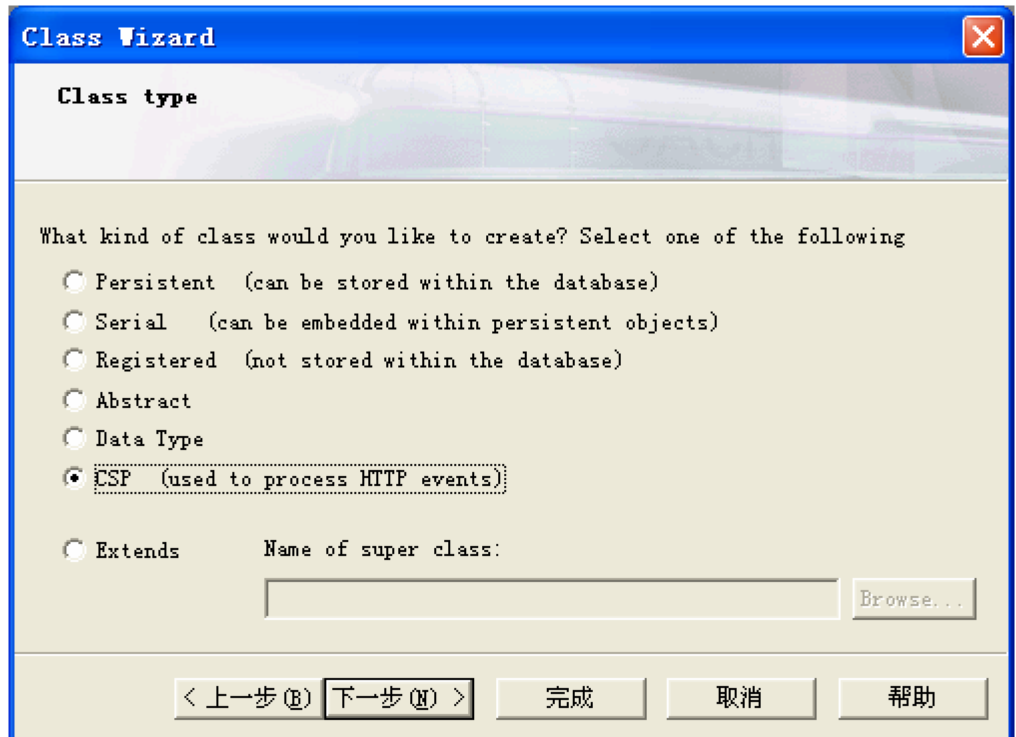
2.点击新建按钮或者菜单。然后选择 Caché Classes Definition，然后点 OK 按钮。



3.输入“Test”为包的名字，“Hello”为类的名字。然后点下一步按钮。



4.选择 CSP 作为类的类型，点完成按钮



5.得到如下代码：

```
Class Test.Hello Extends %CSP.Page [ ProcedureBlock ]
{
	ClassMethod OnPage() As %Status
	{
	&html<<html>
	<head>
	</head>
	<body>>
	;To do...
	&html<</body>
	</html>>
	Quit $$$OK
	}
}
```

6.在生成了的 OnPage 方法中，替换注释：

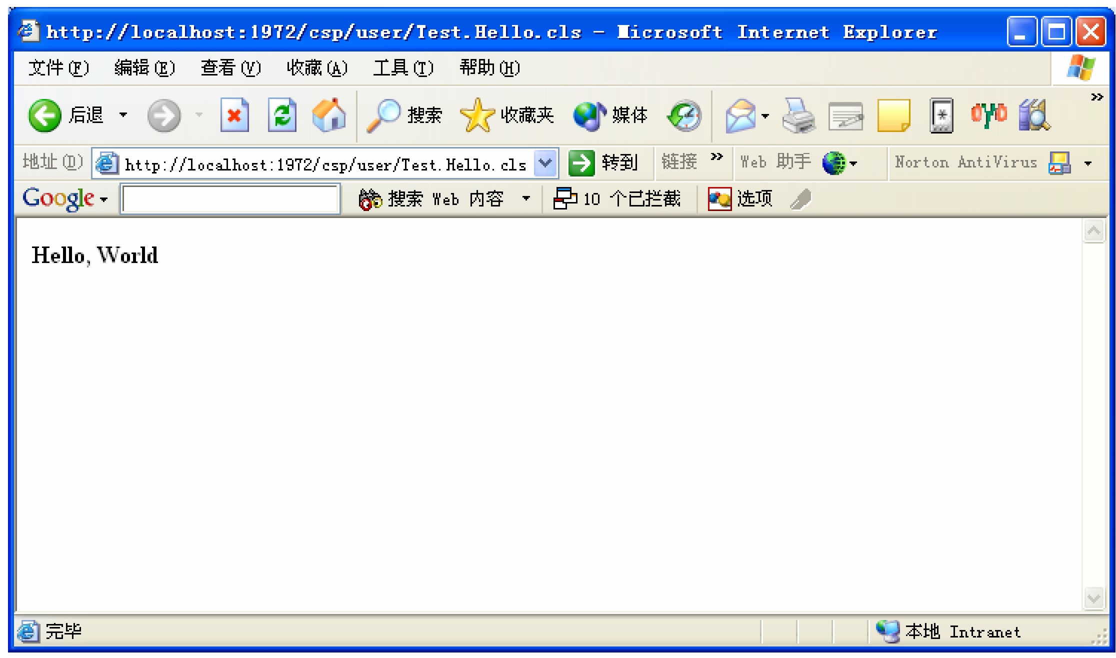
```
; To do...
```

为：

```
Write "<b>Hello, World</b>",!
```

7.保存并编译这个类。

8. 打 开 浏 览 器 并 输 入 如 下 地 址 ： http://localhost:1972/csp/user/Test.Hello.cls



这个程序是这样工作的：

1.浏览器向本地的 Web 服务器送出一个要求访问 Test.Hello.cls 的请求。

2.Web 服务器将这个请求传递给 CSP Gateway（已经连接到 Web 服务器），接着由它将该请求传递给 CSP 服务器。在我们这个简单的环境中，浏览器、Web 服务器和 Caché 应用服务器都在同一台计算机上。而在一个真实的系统部署环境中，它们或许是位于分开的机器上的。

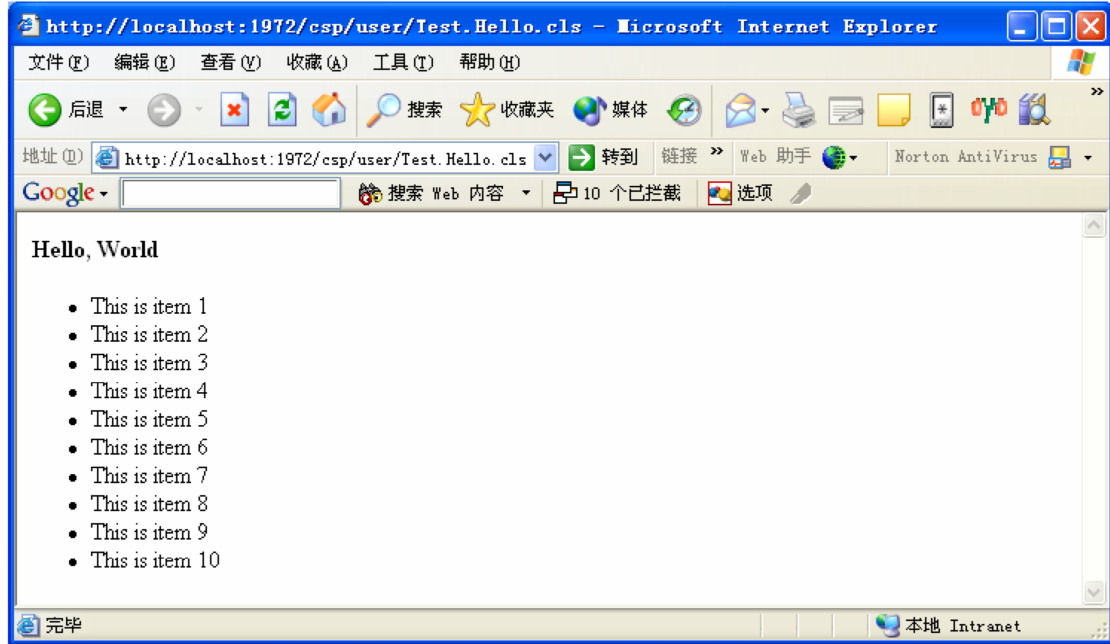
3. Caché CSP 服务器查找一个叫做 Test.Class 的类并且调用它的 OnPage 方法（这个方法是我们已经在 Caché Studio 中编辑过的）。

4.任何由 OnPage 方法写到主要设备上的输出都被送回到浏览器上。

如果你愿意的话，你可以向 OnPage 方法中添加更多的代码：

```
Write "<ul>",!
 For i = 1:1:10 {
   Write "<LI> This is item ", i,!
 }
 Write "</ul>",!
```

运行这段代码你将得到如下输出：

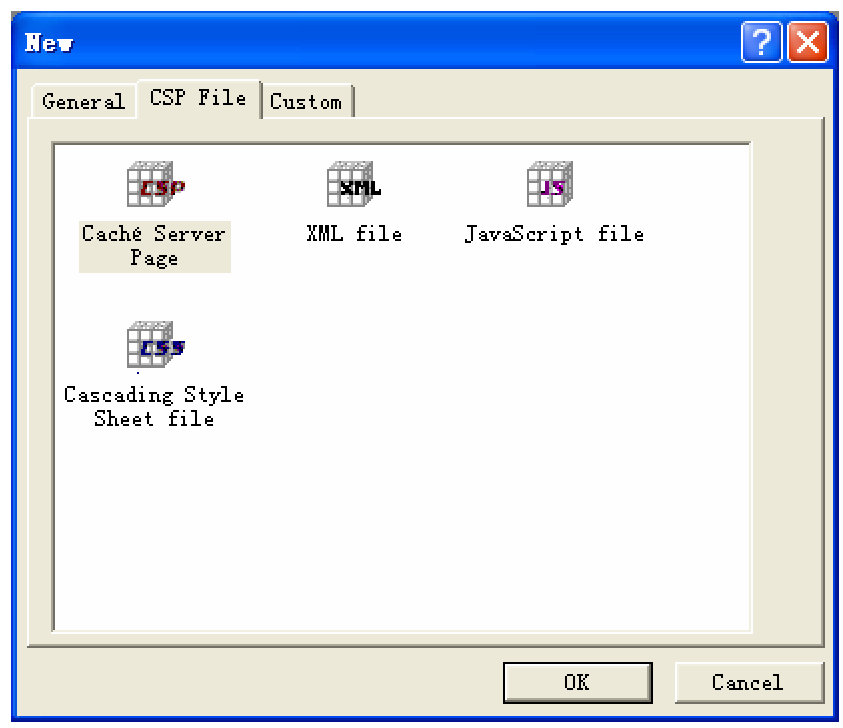


1.1.3.2　　基于 HTML 文件的方法

另外一个使用 CSP 的方法是创建一个 HTML 文件，然后让 CSP 编译器把它翻译成类。

要使用标记过的 HTML 创建“Hello World”页面，请按照如下方法做：

1.启动 Caché Studio，点新建按钮或者菜单



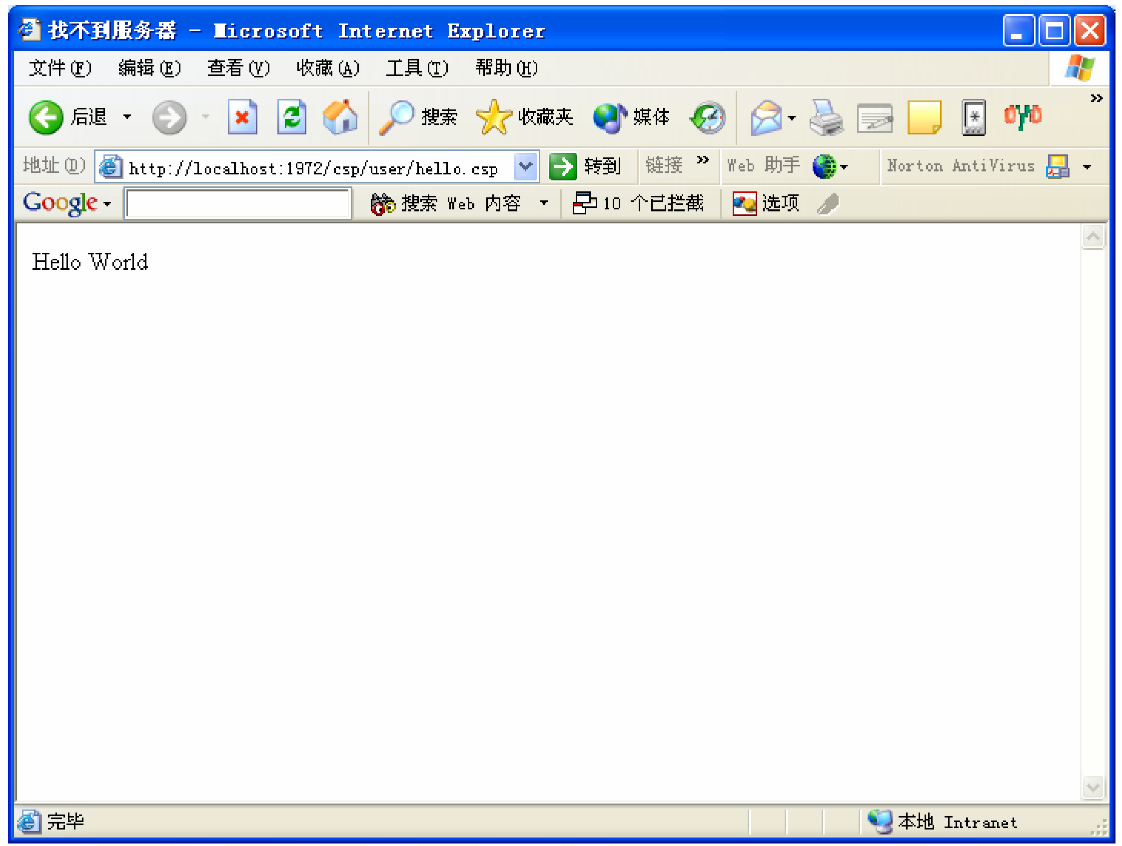
选择“Caché Server Page”，然后点 OK 按钮，你将看到：

```
<HTML>
<HEAD>
<!-- Put your page Title here -->
<TITLE>	Cache Server Page </TITLE>
</HEAD>
<BODY>
		<!-- Put your page code here -->
		My page body
</BODY>
</HTML>
```

2.修改“Mypage body”为“Hello World”。

3.保存，把它存为 csp/user 下的 Hello.csp

4.打开浏览器，输入地址：http://localhost:1972/csp/user/hello.csp，你将看到



注意：

你也可以用文本编辑软件或者其它的 HTML 编辑器创建标记过的 HTML 文件。只要把这个文件作为 *Hello.csp* 保存到本地的 */cachesys/csp/user* 目录下即可（“cachesys”是 Caché 的安装路径）。

程序是这样工作的：

1.浏览器把对 Hello.csp 的请求送到本地的 Web 服务器。

2.Web 服务器把这个请求传递给 CSP Gateway（已经连接到 Web 服务器），然后 CSP Gateway 把请求传递给 Caché CSP 服务器。

3.Caché CSP 服务器查找 Hello.csp 这个文件，然后把它交给 CSP 编译器。

4.CSP 编译器创建一个叫做 csp.Hello 的类，这个类包含了 OnPage 方法，用于输出 Hello.csp 的内容。实际上，它会生成一套方法，供在 OnPage

里面调用。编译这一步只有 csp 文件比生成的类新的时候才会发生，否则请求直接被转给相应的类。

5.CSP 服务器将调用新生成的 OnPage 方法并且它的输出如同前面的例子那样被发送给浏览器。

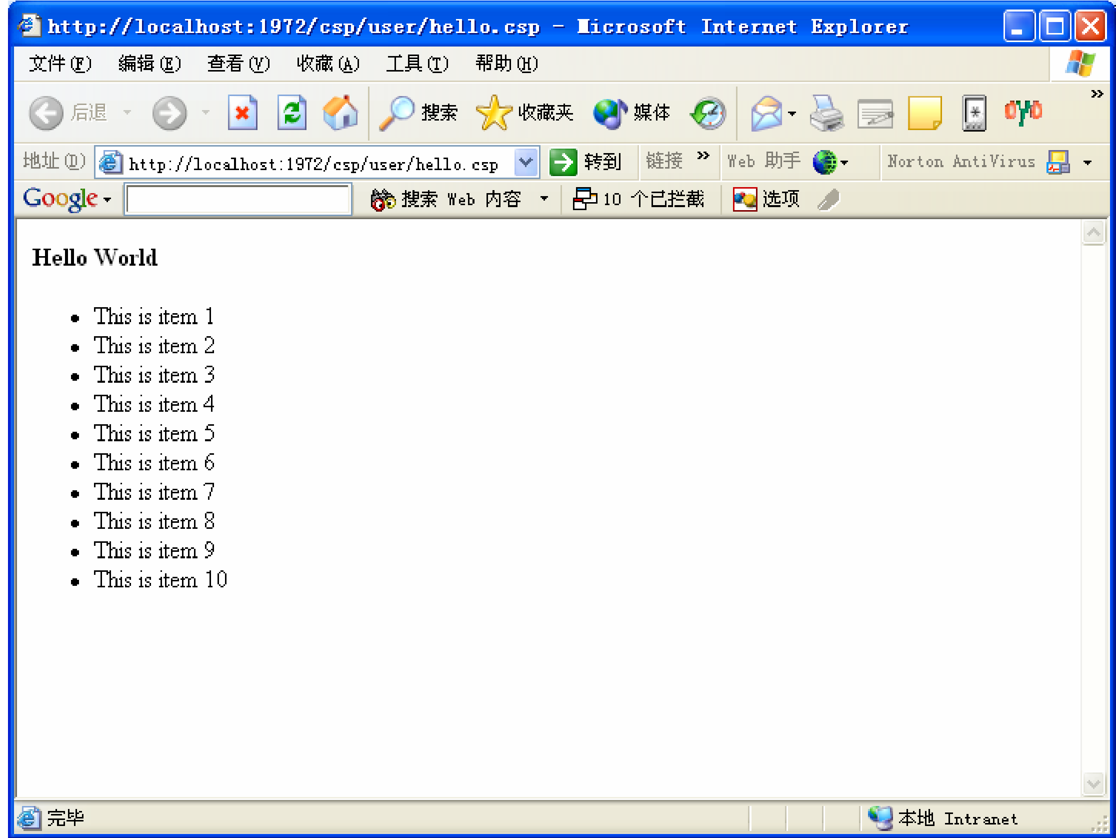
像前面说过的例子一样，这是一个为了教学的目的而有意简化的示例。CSP 编译器在实际上是一个专门的 XML/HTML 处理引擎，它能够：

- 在一个 HTML 页面中处理服务器一侧的脚本和表达式
- 当某些 HTML 标记被认出时可以完成服务器一侧的动作。

像前面的例子一样，你可以增加程序逻辑来使它更加有趣，例如：

```
<html>
<body>
<b>Hello, World!</b>
<script language="CACHE" runat="server">
 // this code is executed on the server
 Write "<ul>",!
 For i = 1:1:10 {
   Write "<li> This is item ", i,!
 }
 Write "</ul>",!
</script>
</body>
</html>
```

访问的时候看起来像：


http://localhost:1972/csp/user/hello.csp - Microsoft Internet Explorer
文件(F) 编辑(E) 查看(V) 收藏(A) 工具(T) 帮助(H)
后退 搜索 收藏夹 媒体
地址(D) http://localhost:1972/csp/user/hello.csp 转到 链接 Web 助手 Norton AntiVirus
Google 搜索 Web 内容 10 个已拦截 选项
Hello World
This is item 1
This is item 2
This is item 3
This is item 4
This is item 5
This is item 6
This is item 7
This is item 8
This is item 9
This is item 10
完毕
本地 Intranet

1.2 CSP 配置

本章涵盖了如下主题：

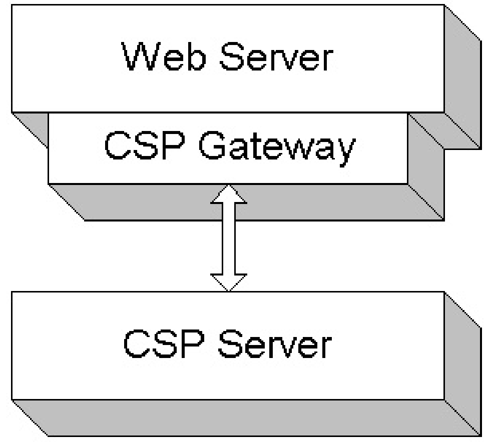
- 主要的 CSP 组件
- 为使用 CSP 配置 Web 服务器，包括 IIS、Apache 和 NetScape 的 Web 服务器
- 对 Caché 内建的 Web 服务器的描述
- 配置 CSP Gatway
- 如何配置一个新的 CSP 应用
- CSP 服务器的配置

要得到更多的安装和配置 CSP Gateway 和你的 Web 服务器的信息，请参考：http://localhost:1972/csp/docbook/DocBook.UI.Page.cls?KEY=GCSP_gateway

1.2.1 CSP 组件

CSP 的操作需要三个独立的组件的互相配合，它们是：Web 服务器、CSP Gateway 和 CSP 服务器：

CSP 组件



你应当了解这些组件以及它们所起的作用和它们是如何配置的。

CSP 组件

组件	作用	配置
Web 服务器	访问授权，静态访问服务（非 CSP），例如 html。	虚拟目录，访问控制，静态内容的位置。
CSP Gateway	传递 CSP 给选定的 CSP 服务器。	要连接的 CSP 服务器，超时和灾难恢复。
CSP 服务器	处理 CSP 请求，从 CSP 文件生成 CSP 类。	URI 目录到“application”的映射，CSP 文件的位置，CSP 应用的默认设置。

这些组件将在后面的章节中仔细讨论。

1.2.2 CSP 应用

CSP 的配置是基于“CSP 应用”这个概念的。CSP 应用被定义为通过给定的 URL 路径访问的一组页面或者类。

例如，所有的 CSP 例子页面都被看作是“/csp/samples”应用的一部分。一个应用可以包含许多子程序，例如“/csp/samples/cinema”。

下面的各节含有如何配置一个新的 CSP 应用，即“/myapp”

1.2.3 Web 服务器的配置

在一个 CSP 应用中，Web 服务器的作用是接受正在提交的 HTTP 请求，实施访问控制（检察许可）、静态访问服务以及传送任何 CSP 的内容（即任何带有一个.csp 或者.cls 扩展名的文件）给已经被安装在 Web 服务器上的 CSP Gateway。

配置的细节，不同的 Web 服务器上是不一样的。某些 Web 服务器，例如 Microsoft IIS，采用的是一个基于目录访问许可的方式，而另外一些，例如 Apache，依靠的是由用来提供数据服务的可执行程序定义的许可。

1.2.3.1 Apache

Apache 被配置成所有的对.csp 和.cls 文件的请求都被重定向到 CSP Gateway，它已经被 Apache 的配置文件做好了。

即使你添加一个新的 CSP 应用你也不需要重新配置 Apache，默认的配置就足够了。

1.2.3.2 IIS

Microsoft IIS 以及 PWS，是根据一系列“虚拟目录”配置的。每一个虚拟目录包含一个名字（它相当于一个 URL 的目录部分），一个目录实际的目录（就是其中保存静态内容，例如.html 或者.jpg 等文件的本地目录）和一组许可。

要求获得 CSP 内容的任何请求（URL）包括有一个目录的名字。这个目录的名字必须相应于一个由 Web 服务器定义的虚拟目录或者是一个虚拟目录的子目录。为了使 CSP 的内容得以存储，这个虚拟目录必须至少有所定义的“读”和“执行”的许可。

为虚拟目录定义的“真实的”目录，是为 Web 服务器用来找出正在运行着 Web 服务器的机器上的文件系统中的任何静态的内容（例如.html 或者.jpg 文件）所使用的。注意这个实际的目录是不用于放置各个.csp 文件的，那些.csp 文件是存储在运行着 Caché 的服务器的机器上的文件系统中的。如果 Web 服务器和 Caché 服务器两者都运行在同一台机器上的话（例如在开发期间就可能出现这种情况），那么两者可能使用相同的位置用于静态内容和.csp 文件。这就是 Caché 在安装时如何为它自己和本地 Web 服务器进行配置的。

安装的时候，Caché 检测 IIS 服务是否运行，并且试图配置它以定义一个名称为“/csp”的虚拟目录。这样，对“/csp/samples”和“/csp/user”（它们都是“/csp”的子目录）两者的访问请求都被转移给了本地安装的 Caché。

当你决定增加一个新的 csp 应用时，如果用于新的应用的 URL 路径也是以“/csp”开始的，你就不需要再做任何 IIS 的配置。例如，“/csp/myapp”，它使用的是为“/csp”定义过的虚拟目录。如果你不希望你的 URL 路径以“/csp”开始，那么你需要定义一个相应于你的 URL 路径的用于 IIS 的新的虚拟目录。

例如，要定义一个使用 URL 路径“/myapp”的 CSP 应用，需要按照如下步骤做：

1.启动 IIS 管理器

2.定义一个叫做“/myapp”的虚拟目录

3.为这个目录指定“读”和“执行”的权限

4.指定你计划在其中存储静态内容的实际目录。

你还需要像下面几节中描述的那样来完成额外的 CSP Gateway 和 CSP 服务器的配置。

1.2.3.3 NetScape

Netscape Web 服务器是配置成使所有的.csp 和.cls 文件的请求都重新指向到 CSP Gatway 的。这是在 Netscape 的配置文件“<Webserverdir>/httpd/config/obj.conf”中完成的。

当你增加一个新的 CSP 应用时，你不需要重新配置 Netscape Web 服务器，默认的配置就足够了。

1.2.3.4　　Caché HTTP Server

Caché 包含了一个轻量级的内置的 HTTP 服务器。这个 HTTP 服务器负责监听和直接响应默认的 Caché TCP 端口（1972）上的 HTTP 事件（而不通过 Web 服务器或者 CSP Gateway）。由于有内建的 HTTP 服务器，所以即使你的机器上没有安装别的 Web 服务器 CSP 也可以工作。这就是为什么前面的例子里面的访问地址都带有 1972 这个端口号。实际上，如果指定这个端口，你的请求将直接访问到 CSP。这些服务包含了 Caché 的联机文档、类参考和 Studio 模板。另外，你也可以使用内建的 HTTP 服务器运行 CSP 的例子和在开发期测试 CSP 页面。你不需要做任何配置来启用内建的 HTTP 服务器，只要通过默认的 Caché 的 TCP 端口（1972）提交 HTTP 请求就可以了，例如: http://localhost:1972/csp/samples/menu.csp

内建的 HTTP 服务器只支持大部分的基本 HTTP 服务，并没有为多用户优化。

注意：

内建的 Caché HTTP 服务器不是用来当作 Web 服务器的。如果你正在开发基于 Web 的应用程序或者 Web 服务，你应该使用一个真正的 Web 服务器，例如 Apache 等等。在真正的部署阶段，你应该把 Caché 服务器放在防火墙后面，这将导致内建的 HTTP 服务器对 Internet 客户端来说是不可用的。

1.2.4 CSP Gateway 的配置

CSP Gateway 是一个安装并运行在 Web 服务器上的 DLL 文件或者共享库。

CSP Gateway 探测任何对.csp 或者.cls 扩展名的文件的请求，然后发送它们到定义的 Caché 服务器上处理。

1.2.4.1　CSP Gateway Manager

你可以使用 CSP Gateway Manager 配置 CSP Gateway 或者直接边界它的配置文件。CSP Gateway Manager 是一个小的 Web 应用程序，你可以从浏览器中来使用它。使用它的 URL 取决于你正在使用的 Web 服务器的种类：

- Apache
  http://localhost/cgi-bin/nph-CSPcgiSys
- IIS
  http://localhost/csp/bin/cspmssys.dll
- Netscape Web Server
  http://localhost/cspsys/

注意你必须用你的 Web 服务器的真正的 IP 地址更换“localhost”。

注意：

上面给出的到 Gateway Manager 的链接只有当你在你的系统中安装符合的 Web 服务器才能工作，并且你的 Web 服务器是安装在你的本地机器上的。如果上面的链接不能工作，可以在你的浏览器的地址栏中输入正确的 Gateway Manager 的 URL。

1.2.4.2　服务器访问

CSP Gateway 配置是你能够定义一个 Caché 各个服务器的列表。每个服务器被指定一个逻辑名、一个 IP 地址、一个 TCP/IP 的端口（默认是 1972），和一个用于指出该服务器是可用的还是禁用的标志。此外，你可以配置和这个 Caché 服务器连接的最小和最大连接数以及超时和日志的值。

通过给每个服务器一个逻辑名，CSP Gateway 使应用程序连接到特定的服务器上变得很容易，并且以后如果要改动一个服务器的特性无需对使用该服务器的应用都重新进行配置。

在初始化安装的时候，CSP Gateway 会定义一个逻辑服务器，它就是“LOCAL”，它被定义为连接到本地的 Caché 的拷贝上。

1.2.4.3　　应用程序访问

CSP Gateway 使你能够对一个特定的 CSP 应用将如何连接到一个特定的 Caché 服务器进行配置。CSP 将认为在一个特定的 URL 目录（或者它的子目录）中的所有文件都是同一个应用的一部分。

在初始化安装时，CSP Gateway 会有一个定义好的应用即“/csp”，它是被定义用来传送全部 CSP 申请给逻辑服务器“LOCAL”的。这就是如何将“/csp/samples”和“/csp/user”两者都传送给本地的 Caché 的。

当你决定增加一个新的 CSP 应用时，如果用于新应用的 URL 路径也是以“/csp”开始的，你就不需要再做任何 CSP Gateway 的配置。例如“/csp/myapp”，它使用的是为“/csp”定义过 CSP 的应用。如果你不希望你的 URL 路径以“/csp”开始，那么你需要定义相应于你的 URL 路径的在 CSP Gateway 中的一个新 CSP 应用。

例如，要定义一个使用 URL 路径“/myapp”的 CSP 应用，请按照如下步骤来做：

1.调用 CSP Manager 页面

2.从菜单选择“Application Access”来显示“Configure Application Access”页面。

3.点“Add Application”按钮来显示“Configure Application”页面

4.在“Application Path”里面输入“/myapp”

5.点 Submit 按钮来保存这个新应用的配置。

要了解在“Configure Application”的其他字段的详细信息，和参看它的 Help 页面。

你也必须按照相应的章节的说明来完成其他的 Web 服务器和 CSP 服务器的配置。

1.2.4.4　　CSP Gateway 参数

CSP Gateway 有许多你可以调节的参数。其中包括有超时、灾难恢复以及负载平衡等特性以及你的应用可能要用到的 CGI 环境变量。

要了解这些参数的细节可以参考 CSP Gateway Manager 提供的各个有关的 help 页面。

1.2.5 CSP 服务器的配置

当一个 CSP 服务器接受一个发来的 HTTP 请求时，它使用本地的 Caché CSP 配置来决定这个请求将如何处理。这些配置都存储在 Caché 的配置文件（.cpf 文件）中，并且可以在 Caché Configuration Manager 中利用 CSP 标签页来修改。

CSP 服务器的配置显示了一个由应用的名子（URL 目录）组织的 CSP 应用的树。这个树被分割成“Applications”（关于用户的应用的信息）和“System”（关于 Caché 里面包含的基于 CSP 的工具的信息）。

对于每一个应用，该配置表保留以下信息：

CSP 服务器配置选项

项目	说明
Namespace	这个应用页面所运行于的 Caché 命名空间。
Caché Physical Path	在 Caché 服务器上存放的 CSP 源文件的物理路径（目录）。这个路径在 Caché 服务器上一般是*<cachesys>/csp/* 这个目录。
h 是 URL 路径 Path 是物理路径，那么 Re*curse*	

打开，UPat*h/xxx/*yyy 在 PPa*th/xxx/* yyy *中查找 CSP 文件。/yyy* 在 *PPath/xxx/ yyy* 中查找 CSP 文件。如果 *Recurse* 是关闭的，只有包含在 UPath 中的文件录才会被使用。	
Auto Compile	指定 CSP 服务器是否自动编译 CSP 源文件。如果打开这个功能的话，当一个 CSP 文件比编译过的类新的话，这个源文件将被重新编译。你可以在开发期间打开这个功能而在真正的运行环境中关闭这个功能。
Event Class	指定 CSP 类（**%CSP.Page** 的子类）的默认的名字，这个 CSP 的类的方法被 CSP 应用的事件调用，例如超时或者会话终止。你可以使用**%CSP.Session** 对象的 *EventClass* 属性覆盖这个值。
Default Timeout	默认的会话超时，按秒计。你可以使用%CSP.Session 对象的 *AppTimeout* 属性覆盖它的值。
Default Super Class	指定当 CSP 编译器从 CSP 文件创建类的时候默认的超类的名字。如果这个字段是空的，默认的值是“%CSP.Page”。
Use Cookies for Session	指定 CSP 是如何保持对浏览器所在的那一个会话的跟踪。CSP 通过使用会话的 cookie

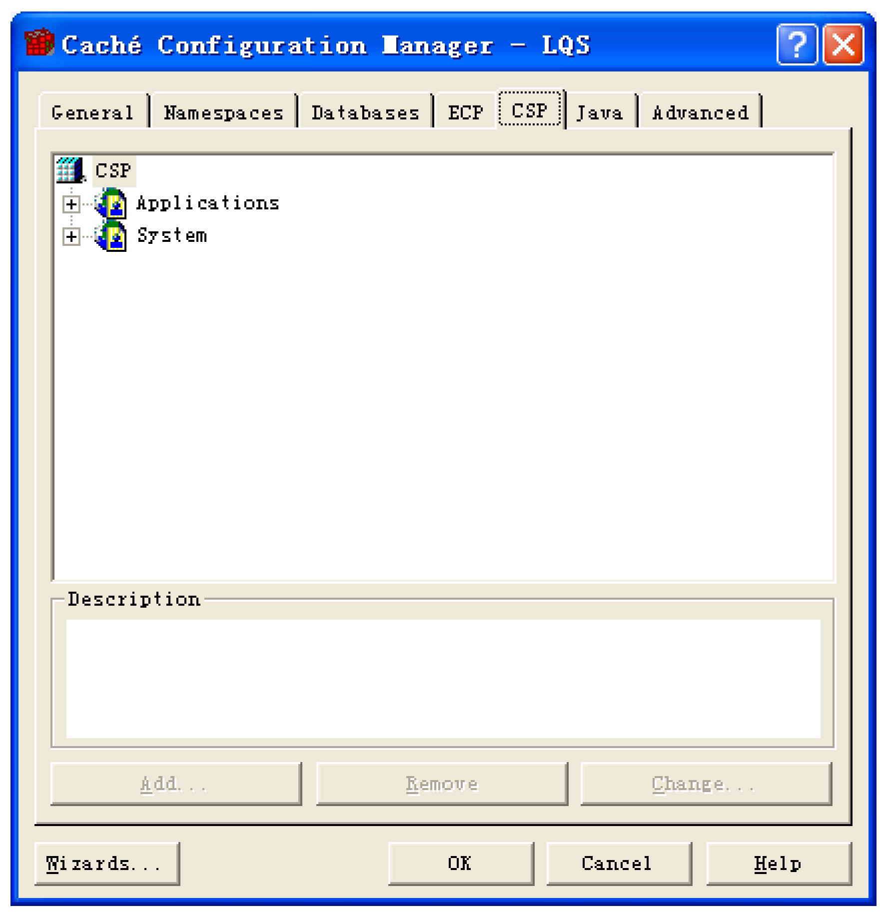
	或者重写发送到客户端的 URL 跟踪会话。该选项让你选择：a) 总使用 cookies， b) 从不使用 cookies， c) 使用 cookies 除非客户端浏览器禁用 cookies (自动检测)。 注意：该选项不管应用是否使用 cookies 它都不做什么；它仅仅控制 CSP 是如何控制会话的。
Session Cookie Path	会话 cookie 的范围。这决定了浏览器要使用哪个 URL 来向 Caché 送回会话的 cookie。如果你的应用叫做“myapp”， 它默认只有为 */myapp/* 下面的页面发送 cookie。你应当限制这个使得只有你的应用能使用，这样就可以阻止这个会话的 cookie 被这台机器上的别的 CSP 应用，或者这个 Web 服务器上的别的应用使用。
Serve Files	只适用于内建的 Web 服务器。允许 Caché 内建的 Web 服务器从应用的路径提供静态内容的服务。这也允许 CSP 流服务器从这个路径来提供文件服务。
URL 路径的默认的页面。	•Custom Error Page •当这个应用产生页面的时候发生错误时显示的.csp 或者.c
包的名字。如果这个 Package N me •是被 CSP 编译器使用的一个可选用*的包的名字*。这 *个名字*	

是用来指定从 CS
件建立的类的包的名字。如果个被 CSP 编译器使用的一个可选用的包的名字。这个名字是用来指定从 CSP 文件建立的类的包的名字。如果这个字段为空，默认值为“csp”。

1.2.6 在 CSP 服务器上定义一个新应用

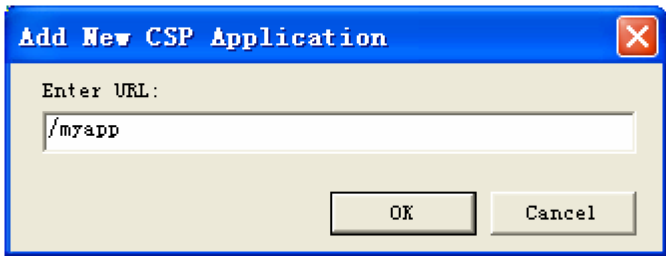
要在一个 CSP 服务器上定义一个名为“/myapp”的新 CSP 应用，请按照如下方式做：

1.运行 Caché Configuration Manager 并且选定 CSP 标签页。

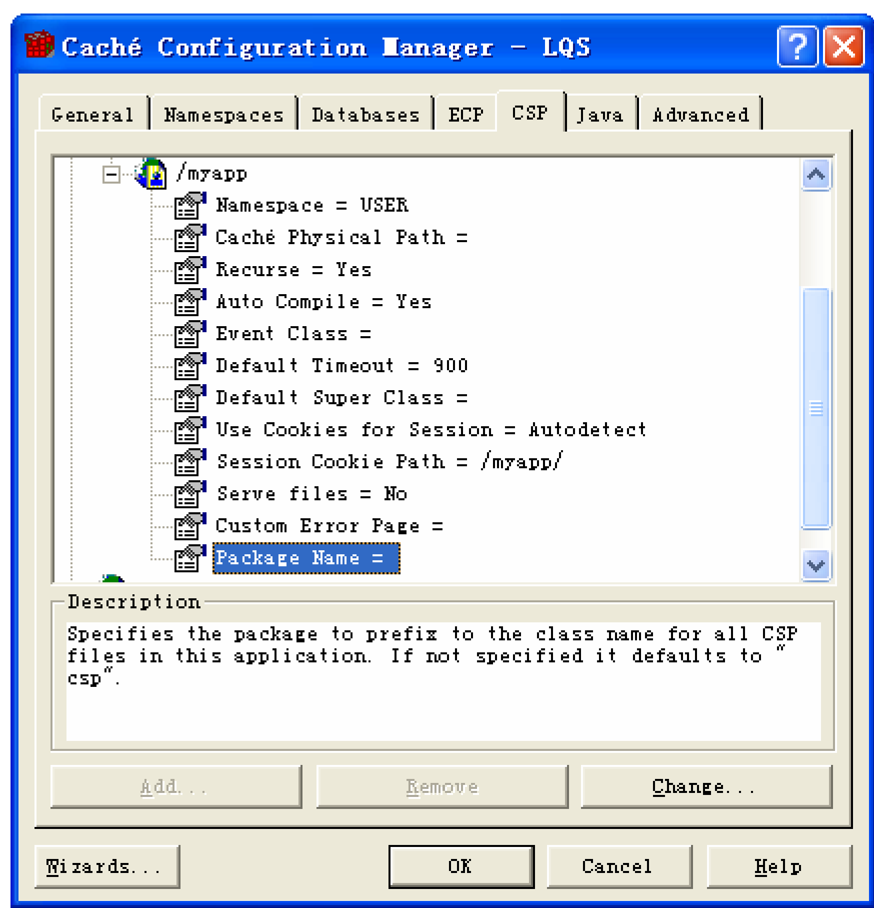


2.点击“Applications”项并且点 Add 按钮。

3.输入“/myapp”，点 OK 按钮。



4.填入所希望的各个应用属性（大部分是可选的）。最重要的是应用要在其中运行的 Caché 命名空间和 CSP 各个文件的实际位置（如果你正在使用以 HTML 为基础开发）。



5.点 OK 按钮，然后点 Activate Changes 按钮。

你也必须要像前几节所说明的那样来完成额外的 CSP Gateway 和 Web 服务器的配置工作。

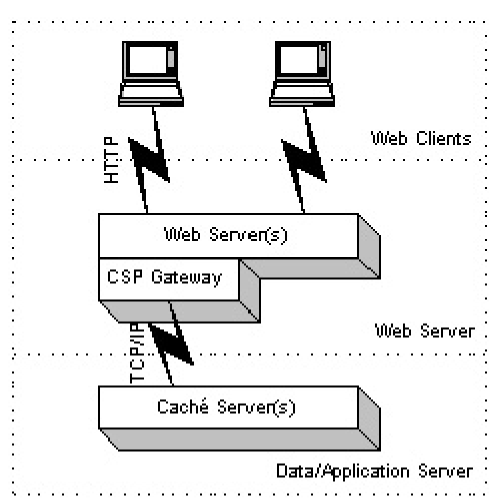
1.3 HTTP 请求

CSP 的主要任务是提供动态的内容来响应发来的 HTTP 请求。你并不需要了解 HTTP 来使用 CSP 的最内部的详细情况。本节涵盖了一些基本概念和 CSP 如何处理这些请求的概述。

HTTP 是一个简单的协议，一个客户机在它里面产生一个发给服务器的申请。HTTP 是一个无状态的协议——在一个客户机和一个服务器之间的连接只是在它为该请求服务时才存在。每一个 HTTP 请求都还有一个“请求标题”，由它指定该请求的一个类型（例如 GET 或者 POST）、一个 URL 和一个版本号。一个请求中也可以包括其它的附属信息。CSP 自动判断哪一个请求应当处理，然后将它们发送给在运行在 Caché 服务器上的适当的类和包装各种不同的申请信息使其成为易于使用的各个对象（例如%CSP.Request 对象）。

1.3.1 CSP 体系结构

下图显示了 CSP 的架构：



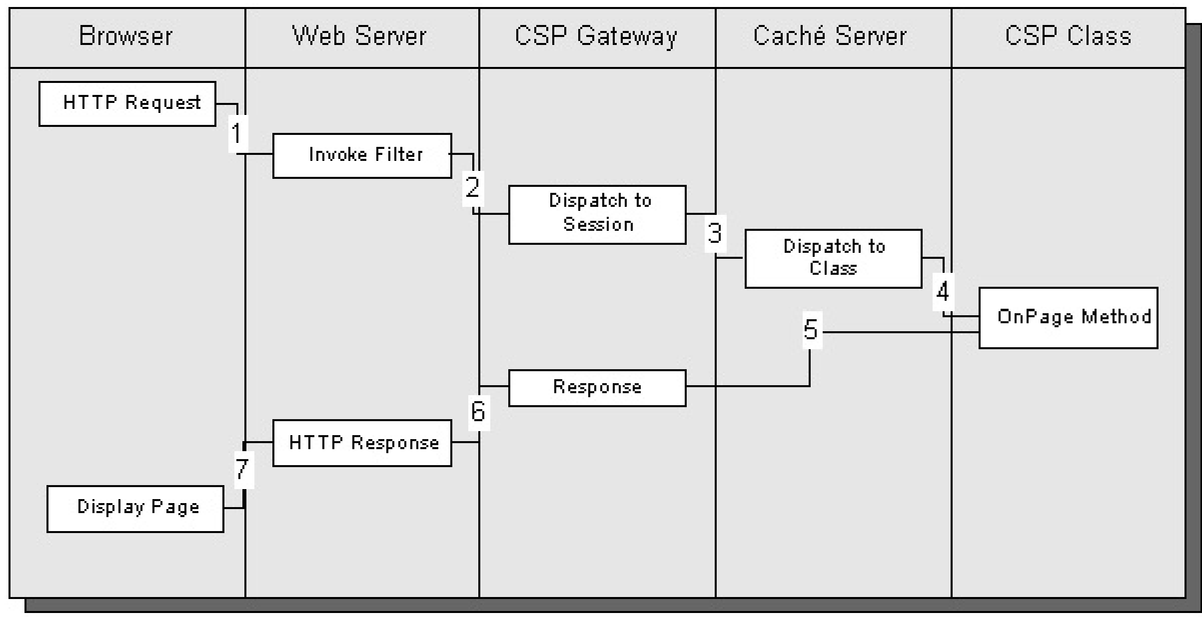
CSP 的运行时的环境包括：

- HTTP 客户端（例如 Web 浏览器）
- Web 服务器

- CSP Gateway
- CSP 服务器

1.3.2 HTTP 请求的处理

下面的图标解释了 CSP 处理 HTTP 请求时的事件流程：



1.浏览器（或者类似的 Web 客户端）提交 HTTP 请求。

2.Web 服务器判断是否是一个 CSP 请求，如果是的话转交给 CSP Gateway 处理，CSP Gateway 是安装在 Web 服务器上的。

3.CSP Gateway 重新打包并发送请求给正确的 Caché 服务器。

4.Caché 服务器解码消息，判断哪个类（你的应用的一部分）应该处理这个事件，并调用类的 **OnPage** 方法。

5.**OnPage** 方法的输出结果被作为一个 HTTP 响应送回到 CSP Gateway。

6.CSP Gateway 把 HTTP 响应交给 Web 服务器（实际上，响应被通过 CSP Gateway 变成流送回到 Web 服务器）。

7.Web 服务器把响应返还给 Web 浏览器。

这些细节将在后面的部分继续讨论。

1.3.2.1 Web 服务器和 CSP Gateway

HTTP 请求始于一个 HTTP 客户端向 Web 服务器发出一个消息。如果这是用来请求访问一个静态文件（例如一个 HTML 或者 JPG 文件），Web 服务器就申请在本地的文件系统中找出该文件，并将它的内容送到客户机。CSP 的各种请求都是利用 CSP Gateway 进行不同的处理的。CSP Gateway 是一个 DLL 或者共享的库文件，它是被 Web 服务器（例如 IIS、Apache 等等）用来处理某些类型的事件的。CSP Gateway 如果已经安装到 Web 服务器上，当 URL 的目录路径有在 Web 服务器中定义的访问权限的话并且 URL 的文件名是.csp 或者.cls 为扩展名的，那么 CSP Gateway 就处理一个 HTTP 申请。

例如，Caché 的默认安装下，下列 URL 是被 CSP 处理的：

http://localhost/samples/menu.csp

CSP Gateway 提供了以下功能：

1.它是轻量级的；它提供最少的处理而把大部分的工作丢给 Caché CSP 服务器，这样就为 Web 服务器保留了更多的资源。这一点是和其它的 Web 技术不同的，其它的 Web 技术中 Web 服务器会被用来处理各种需求而导致不堪重负。

2.它保持一个到指定的 CSP 服务器的连接池。

3.它提供灾难恢复的可选功能来允许使用多台内部已经连接好的 CSP 服务器。

1.3.2.2　　CSP 服务器

CSP 服务器是一个运行在 Caché 应用服务器上的一个进程，它是专门用于为从 CSP Gateway 来的请求提供服务的。每一个应用服务器可以运行多少个 CSP 服务器进程取决于机器的类型和 Caché 许可证上所加的限制。

当处理“无状态”申请时，每一个 CSP 服务器进程可以支持从许多不同的客户机上来的请求。在“保留状态”的方式中，一个进程专门用于处理从一个客户机来的请求并且一直延续到“保留状态”模式被关闭。

注意：

Caché 的最主要的优点之一就是应用服务器和数据服务器之间并没有真正的区别，你可以根据实际的需求把你的应用配置成使用多个或者单个的机器。这和应用逻辑和数据模式是无关的。无论一个特定的系统是一个应用服务器还是一个数据服务器（或者两者皆是），这都只是配置上的事情。

1.3.2.3　　服务器上的事件流

当 CSP 服务器接收到来自 CSP Gateway 的请求时，它会做以下事情：

1.判断这个请求属于哪个会话，如果没有，它就开始一个新的会话。

2.确保该请求将在正确的 Caché 命名空间中被处理。

3.确定正确的%CSP.Session 对象是可用的，并且创建基于 HTTP 请求中的信息的%CSP.Request 对象的实例。如果需要任何译码操作，它也会做这件事。

4.判断应该用哪个类来处理这个请求并且调用它的 Page 方法（并且接着调用它的 OnPage 方法）。

1.3.2.4　　URL/类名称的解析

CSP 服务器靠解析它的 URL 来判断要将 HTTP 请求发送给哪个类。CSP 把一个 URL 解析成以下部分：

URL: *http://127.0.0.1/csp/samples/object.csp?OBJID=2*

URL 组成部分

Component	Purpose
http://	协议
127.0.0.1	服务器地址
/csp/samples/	目录
object.csp	文件名及扩展名
?OBJID=2	查询

协议和服务器地址是由 Web 服务器处理的，而这和 CSP 服务器是无关的。目录是用于判断一个 URL 引用的是哪个“CSP 应用”。每一个 Caché 配置可以定义许多 CSP 应用，它是利用一个 URL 的目录部分来识别的。每一个 CSP 应用，可以在 Caché Configuration Manager 中建立和修改，并指定了为全部的请求所使用的一些设置和一个给定的 URL 目录。最重要的是 Caché 的命名空间要在请求要执行的命名空间内。

- 如果该文件的扩展名是“.cls”，那么使用该文件名作为类的名字。
- 如果该文件的扩展名是“.csp”，那么建立一个类是用“csp”作为一个包的名字，文件的名字作为类的名字。如果这个类是不存在的或者是已经过时的，那么 CSP 编译器就从一个 CSP 源文件建立一个类。这个源文件有和在 URL 中的那个文件相同的名字和扩展名。

注意：和 CSP 一起使用的 URL 文件名有许多限制：

- 他们必须是有效的 Caché 类名称（它们不可以办函空格或者标点符号并且不能以数字开头）。
- 它们必须和已经存在的其它的类的名字不冲突。

注意

如果一个*.csp* 文件放在一个已经定义的目录的子目录下，那么该子目录的名字就成为使用该页面的%CSP.Page 类的包的名称的一部分。例如，如果 URL 目录*/csp/samples* 定义为一个 CSP 应用，那么*/csp/ samples/myapp/page.csp* 将去引用名为 **csp.myapp.page** 的类。

1.3.3 %CSP.Page 类

在 CSP 服务器上，所有的 HTTP 请求都是被所调用的由%CSP.Page 类定义过的各种方法来处理的。%CSP.Page 类自己不会直接处理各个请求，它只是定义需要处理 HTTP 的各个请求的接口。实际事件的处理是由%CSP.Page 的子类来做的。（这些子类有可能是手工创建的也有可能是从 CSP 源文件创建的）。

%CSP.Page 的各个子类也从来不会被初始化，也就是说从来不会建立%CSP.Page 的对象。被%CSP.Page 定义的各种方法全都是类方法，因而为了调用它们是不需要一个对象的。如我们将要看到的那样，这些方法所需要的状态信息是由其它对象提供的（例如由%CSP.Request 和%CSP.Session 对象提供），他们都是由 CSP 服务器来管理的。

1.3.3.1 Page 方法

在 CSP 服务器判断了应当由哪个类来处理一个请求后，它调用该类的 Page 方法。在调用前，CSP 将确定为该请求要处理的内容是已经正确地设置好的。这包括重新指向标准的输出设备（$IO）以便使所有的输出（利用 Write 命令）是送回给 HTTP 客户机的并且建立处理内容所需的任何对象的实例或变量。

Page 方法处理对 HTTP 请求的完整响应。它是靠按照顺序调用 OnPreHTTP、OnPage 和 OnPostHTTP 各个回叫方法实现的。这些都被称为回叫方法是因为一个子类为了提供定制行为可以覆盖它们。

OnPreHTTP 方法是负责写出用于 HTTP 响应的标题的。这包括例如内容类型和 cookies 等信息。默认的行为是设置内容类型为“text/html”。在出现你需要更为直接控制该响应标题的情况时，你通常只需要覆盖 OnPreHTTP 方法。

OnPage 方法完成的是为响应一个 HTTP 请求要做的大量工作。它是负责来写出该申请的全部内容的，例如一个 HTML 或者 XML 文档。

例如，这里是含有很简单 OnPage 方法的 CSP 类的示例：

```
Class MyApp.Page Extends %CSP.Page
{
ClassMethod OnPage() As %Status
{
   Write "<html>",!
   Write "<body>",!
   Write "My page",!
   Write "</body>",!
   Write "</html>",!
   Quit $$$OK
}
}
```

OnPostHTTP 方法是用来提供一个地方来完成你希望在 HTTP 请求已经被处理完以后要完成的任何操作的。

1.3.3.2 %CSP.Page 类参数

%CSP.Page 类含有许多类参数，你可以覆盖它们来提供定制的行为而不需要编写代码。

要知道可以使用的各种参数的列表，可以参看%CSP.Page 的文档。

如果你是系统地开发应用的话，可以在你（例如利用 Studio）所建立的%CSP.Page 各个子类中覆盖这些参数。

如果你使用.csp 文件来创建各个页面，你可以使用 csp:CLASS 标记类来为这些参数提供值：

```
<csp:CLASS PRIVATE="1">
```

1.3.3.3 %CSP.Request 对象

当 CSP 服务器响应一个 HTTP 请求时，它将有关发送来的请求的各种信息都包装进入%CSP.Request 的一个对象之中，你可以使用%request 变量来引用这个对象。可参看%CSP.Request 类的文档来得到它的各种属性和方法的完整列表。

1.3.3.4 URL

要找出接收的 HTTP 请求的 URL，可使用%CSP.Request 对象的 URL 属性：

```
Write "URL: ", %request.URL
```

1.3.3.5 URL 参数

一个 URL 可能包括一个各个参数的列表。%CSP.Request 对象使这些可以通过他的 Data 属性来获得。

例如，假设发来的 URL 含有：

```
/csp/user/MyPage.csp?A=10&a=20&B=30&B=40
```

你可以在服务器一端得到这些参数：

```
Write %request.Data("A",1)    // this should be 10
 Write %request.Data("a",1)    // this should be 20
 Write %request.Data("B",1)    // this should be 30
 Write %request.Data("B",2)    // this should be 40
```

此处 Data 是一个多维的属性并且在它之中存储的每一个值都有两个下标：参数的名称和参数的索引值（各个参数在一个 URL 中可以出现多次，如上面的“B”）。注意到参数的名称都是大小写敏感的。还要注意一个 HTTP 请

求是一个 GET 请求还是一个 POST 请求对于 Data 属性表示的参数的值是没有关系的，该属性表示的参数值用的是完全相同的方法。

你可以使用 Caché ObjectScript 的**$Data** ($D)命令来检验一个给定的参数值是否已经被定义：

```
If ($Data(%request.Data("parm",1))) {
}
```

如果你想引用一个参数，但是无法肯定它是否被定义，你可以使用 Caché ObjectScript 的**$Get** 命令：

```
Write $Get(%request.Data("parm",1))
```

你可以使用%CSP.Request 对象的 **Count** 方法来统计出为特定的参数定义了多少个值。

```
For i = 1:1:%request.Count("parm") {
   Write %request.Data("parm",i)
 }
```

1.3.3.6　CGI 环境变量

Web 服务器提供一组被当作 CGI 环境变量的值，它们含有有关 HTTP 客户机和 Web 服务器的信息。你可以使用多维的 CgiEnvs 属性来访问这些 CGI 的环境变量的值。你可以像使用 Data 属性那样的方式来使用它。

例如，要查看是什么类型的浏览器发出的 HTTP 请求，使用 CGI 环境变量“HTTP_USER_AGENT”来查看即可：

```
Write %request.CgiEnvs("HTTP_USER_AGENT")
```

1.3.3.7　Cookies

如果 HTTP 的请求包含任何的 cookies，你可以使用多维的 Cookies 属性来取得它们的值。你可以像使用 Data 属性那样的方式来使用它。

1.3.3.8 MIME 数据

如果一个发来的请求中含有 MIMI（Multipurpose Internet Mail Extensions）数据（通常是用于比较大的信息，例如文件等），你可以使用%CSP.Request 对象的 GetMimeData 方法来得到它。这个方法发取得一个 Caché stream 对象的实例，它使你能够用来读取 MIME 数据。

要查看使用 MIME 数据的例子，可以参看在 CSP samples 中的 upload.csp 页面：

```
http://127.0.0.1:1972/csp/samples/upload.csp
```

1.3.4 %CSP.Response 对象

你可以使用%CSP.Response 对象来控制什么样的响应头信息要被送回给 HTTP 客户机。CSP 服务器自动创建一个这个类的一个实例并且在变量%response 中放置一个对它的引用。

当用%response 对象控制 HTTP 头信息时，你通常需要在%CSP.Page 类的 OnPreHTTP 方法中设置它的属性。例如要重定向一个发来的 HTTP 请求时，定义下面的 OnPreHTTP 方法：

```
Class MyApp.Page Extends %CSP.Page
{
// ...

ClassMethod OnPreHTTP() As %Boolean
{
   Set %response.ServerSideRedirect = "redirect.csp"
   Quit 1
}
}
```

1.3.4.1 为 Cookies 服务

你可以利用%response 将一些 cookies 送给 HTTP 客户机。参看“在 Cookies 中保存数据”一节。

1.3.4.2 为不同的 Content Type 服务

通常个一个 CSP 页面提供“text/html”的内容。你可以有几种方法来指定一个不同的内容类型：

- 靠设置%CSP.Page 类参数 CONTENTTYPE 的制。
- 靠在 OnPreHTTP 方法中设置%response 对象的 ContentType 属性的值。

1.4 会话管理

HTTP 是一种无状态的协议，每一个请求都不知道以前的请求。当这对于那些只是为用户提供简单的静态内容的网站来说是可行的，但是这却为要开发互动的、动态的各种应用带来了麻烦。为了帮助解决这个问题，CSP 提供了“会话管理”。

1.4.1 会话

一个会话指的是在一定时间内，从一个特定的客户机向一个特定的应用发出的一系列请求。

CSP 自动提供会话跟踪，你不需要做任何特别的事情来实现它。CSP 应用可以借助于%CSP.Session 对象来查询和修改它们的会话有关内容。CSP 服务器通过 Caché ObjectScript 的%session 变量使这个对象能为你所用。

1.4.1.1 创建会话

当一个 HTTP 客户机发出它的第一个请求给一个指定的 CSP 应用时，一个会话就开始了。

当一个新的会话被建立时，CSP 服务器做了以下的事情

1. 建立一个新的会话 ID 号。
2. 检查许可
3. 创建一个%CSP.Session 对象的实例（它是一个持久对象）
4. 调用当前会话的事件类（如果存在的话）的 OnSessionStart 方法
5. 建立一个 session-cookie 以便在会话期中跟踪 HTTP 客户端发来的后续的请求。如果客户端的浏览器禁用了 cookie，那么 CSP 将自动使用 URL 重写技术（在 URL 中放置一些特殊的值）来跟踪会话。

对于一个会话的第一个请求，%CSP.Session 的 NewSession 属性的值被设置为一，对于以后的请求，这个值被设置为 0。

```
If (%session.NewSession = 1) {
   // this is a new session
 }
```

1.4.1.2　　会话 ID

会话的 ID 是一个用于识别一个特定的会话的特殊的值。会话 ID 是用于持久地%CSP.Session 对象的标示符。CSP 服务器使用会话 ID 来保证 HTTP 的请求在被处理时能够使用正确的%CSP.Session 类的实例。

CSP 应用能够通过%CSP.Session 对象的 SessionId 属性来找到它的特定会话的 ID：

```
Write "Session ID is: ", %session.SessionId
```

1.4.1.3　　会话的终止和清除

会话将在以下情况之一终止：

1. 客户机停止或转向到另外一个网站。
2. 会话超时
3. 会话在服务器端被指示超时（借助于设置%CSP.Session 对象的 EndSession 属性为 1）

当会话终止的时候，CSP 服务器将删除持久对象%CSP.Session 并且减少占用的许可数。如果存在的话，它还调用会话事件类的 OnSessionEnd 方法。

1.4.2 %CSP.Session 对象

%CSP.Session 对象包含当前会话的信息和控制会话的一些方法。

1.4.2.1　　用户会话数据

你可以在%CSP.Session 对象的 Data 属性中存储指定的信息。Data 是一个多维的属性，它使你能够在一个多维数组中关联许多指定的信息。这个数组的内容在会话的生存期内是自动被维护的。

你可以像使用别的 Caché ObjectScript 多维数组的同样的方式来使用%CSP.Session 对象的 Data 属性。

例如，如果下面的方法在一个 OnPage 方法中执行：

```
Set %session.Data("MyData") = 22
```

那么以后的一个对于同一个会话的请求将会在%CSP.Session 对象之中看见这个值（不论由哪个类处理该请求）。

```
Write $Get(%session.Data("MyData")) // this should print 22
```

这个在%CSP.Session 中能存储指定的数据的能力是一个非常有用的特性，但是在使用时应当被正确使用，请参考“状态管理”一节的进一步讨论。

1.4.2.2　设置用户会话数据

要在%CSP.Session 中存储数据，可使用 **Set** 命令：

```
Set %session.Data("MyData") = "hello"
Set %session.Data("MyData",1) = 42
```

在 Data 数组中的一个节点上能够含有最大长度为 32K 的字符串。

1.4.2.3　取得用户会话数据

你可以当作 Caché ObjectScript 表达式的一部分来从 Data 属性取得数据：

```
Write %session.Data("MyData")
Write %session.Data("MyData",1) * 5
```

如果你引用了 Data 数组中一个未定义过的节点，运行时将会出现一个<UNDEFINED>未定义的错误。要避免这样，可以使用 Caché ObjectScript 的$Get 命令。

```
Write $Get(%session.Data(1,1,1)) // return a value or ""
```

1.4.2.4　删除用户会话数据

要从 Data 属性中删除数据，可以使用 Caché ObjectScript 中的 Kill 命令：

```
Kill %session.Data("MyData")
```

1.4.2.5　会话超时

CSP 各个会话自动跟踪自从他们接收到来自一个客户机的请求以来，已经经过多长时间。如果经过的时间超过了一定的阀值那么该会话会自动地超时。

1.4.2.6　超时设置

根据默认值，会话超时的阀值设置在 900 秒（15 分钟）。你可以在 Caché Configuration Manager 的 CSP 部分来改动这个默认值。你也可以从一个应用中来设置它，即借助于设置%CSP.Session 对象的 AppTimeout 属性的值：

```
Set %session.AppTimeout = 3600 // 设置超时为 1 小时
```

要禁用超时，可以设置该属性的值为 0。

1.4.2.7　超时通知

当一个 CSP 超时发生时，CSP 服务器靠调用一个所指定的%CSP.Page 类的 OnTimeout 方法通知该应用。你可以通过%CSP.Session 对象的 EventClass 属性来指定这个类的名称。

默认地，是没有事件类被定义的，并且超时只是终止当前的会话期。

1.4.3 状态管理

因为 HTTP 是一个无状态的协议，为 Web 写的各种应用不得不适用特殊的技术来管理应用的上下文关系，即“状态”。CSP 提供了用于管理状态的许多机制。这些机制中的每一个可能是适用于某些特定的情况的。

1.4.3.1 在请求之间跟踪数据

在一个 Web 应用中状态管理的基本问题是要对连续的各个 HTTP 请求的信息保持跟踪。有许多技术可以用于这方面，包括：

- 采用被隐藏的窗体字段，或者利用 URL 的参数在各自的页面中存储数据。
- 在客户机的 cookie 中存储数据。
- 在%CSP.Session 对象中存储数据。
- 在 Caché 数据库中存储数据。

1.4.3.2 在页面中存储数据

要在一个页面中存储状态信息，你必须放置好该信息，以便后续的请求从这个页面中可以获取信息。

如果该页面经过一个超链接产生了一个申请，那么该数据一定被放置在用于该超链接的 URL 中。例如，此处有一个包含在一个.csp 文件中被定义过的状态信息：

```
<a href="page2.csp?DATA=#(data)#">Page 2</A>
```

当该 CSP 为包含这个连接的页面服务时。表达式#(data)#被 CSP 服务器上变量 data 的值代替的。当使用者选定这个链接到 page2.csp 时，CSP 服务器已经通过%request 对象访问了“DATA”的值。如果需要，CSP 可以对上述数据进行编码。可参考“认证和加密”一节的内容来了解更详细的信息。

如果该页面含有一个窗体，你可以将状态信息放在隐藏的各个字段之中：

```
<form>
<input type="HIDDEN" name="DATA" value="#(data)#">
<input type="SUBMIT">
</form>
```

如同超链接的例子，当这个窗体被送给客户机时，表达式#(data)#被CSP 服务器上的变量 data 的值替代。当使用者提交了这个窗体，“DATA”的值经过%request 对象便可以获得。

1.4.3.3　在 Cookie 中存储数据

用于存储状态信息的另外一种技术是将它放在一个 cookie 之中。一个cookie 是一个存储在客户机上的名称/数值对。每一个从客户机发来的后续的申请都包括有以前全部的 cookie 的值。

要设置一个 cookie 值，需覆盖该页面的 OnPreHTTP 方法并且将该cookie 值放进%CSP.Response 对象的 SetCookie 方法之中：

```
Class MyApp.Page Extends %CSP.Page
{
//...

ClassMethod OnPreHTTP() As %Boolean
{
   Do %response.SetCookie("UserName",name)
    Quit 1
}
}
```

服务器可以利用%CSP.Request 对象的 Cookies 属性来获得该数据。

在一个 cookie 中存储的信息对于在会话结束后你想记住过去的信息是有用的。例如，你可以记住在一个 cookie 中的一个用户的姓名。这样在以后的会话中就不需要重新输入这个信息。

1.4.3.4　在会话中存储数据

如同在前面章节中已经讨论过的那样，你可以在%CSP.Session 对象中利用 Data 属性来存储状态信息。放置在%session 对象中的任何信息是可以用

于当前会话的剩余时间的（或者一直可以使用到它被从%session 对象中去掉为止）。

%session 对象是一个用于存储那些只在一个会话中有用的数据的好地方。例如一个用户的姓名等等。%session 对象对于存储那些必须在查过当前会话范围以外的地方也生存的信息是不适用的。它也不是存储那些在应用中取决于使用者采取的导航途径的信息的好地方。使用者通常都是自由地按照自己的意愿在 Web 的各种应用之间跳来跳去。并且如果一个应用由用户指定路径会带来麻烦的。

1.4.3.5 在数据库中存储数据

如果你有和一个使用者联系在一起的更复杂的信息，那么多半最好的办法是将这些信息存储在 Caché 数据库中。做这件事的一个办法是在数据库中定义一个或者多个持久的对象类，并且在%session 对象中存储他们的对象 ID 识别值来使以后的存取访问得到便利。

1.4.3.6 服务器上下文关系的保存

通常 CSP 服务器只是把从一个请求到下一个请求处理中的上下文关系保存在%session 对象中。CSP 服务器提供了一个机制可以用于保存在请求之间的完整的处理上下文关系——包括各种变量、实例化的各个对象、数据库的锁、打开的各个设备。这被称作为“上下文关系的保存”模式。你可以在一个应用中在任何时候借助于设置%CSP.Session 对象的 Preserve 属性的值来开通或者关闭上下文保存。

1.4.4 认证和加密

将状态信息放置在页面上传递给 HTTP 客户机是相当常见的方法。当后续的各个请求从这些页面发出时状态信息就会被回送给服务器。许多时候，状态信息以这样的方式放置在一个 Web 页面上是重要的。HTTP 的观察者并不能

确定状态信息的值，而服务器确能够确认这是同一个服务器或者会话发出的信息。通过加密服务，CSP 提供了一种易于使用的机制来完成这件事。

1.4.4.1 会话密匙

CSP 利用一个密匙能够对在服务器上的数据进行加密以及解密。每一个 CSP 会话都有一个独特的密匙（通过%CSP.Session 对象的 Key 属性可以存取它），它是被用于为一个会话加密数据的。这个机制是安全的，因为会话期的密匙是绝对不会送给一个 HTTP 客户机的。它是作为%CSP.Session 对象的一部分留在 CSP 服务器上的。

你可以认为地使用%CSP.Page 类的 Encrypt 方法来加密在服务器上的各个值。你能够在以后使用它的 Decrypt 方法来解密这个值。

1.4.4.2 加密的 URL 和 CSP 代码

在某些情况下，从一个.csp 文件生成的一个类能自动地对送给客户机的 URL 各个值进行加密。（对于自己建立的各个类，你必须调用 Link 方法来完成这个动作）。

例如，假设一个.csp 文件含有一个用以定义连接到其它页面的链接标记：

```
<a href="page2.csp?PI=314159">Page 2</a>
```

如果这个 URL 是被加密的，发送给客户机的 URL 可能是：

```
<a href="page2.csp?CSPToken=8762KJH987JLJ">Page 2</a>
```

当用户选择这个链接时，已经被加密的参数“CSPToken”将被送给 CSP 服务器。该服务器接着解密它并将已解密的内容放进%request 对象之中。如果加密的值已被修改或者是从不同的会话送出的那么服务器将发出一个错误信息。

你可以使用%CSP.Request 类的 IsEncrypted 方法来弄清一个参数的值是否被加密过。

CSP 编译器将自动地在一个 HTML 文档中检查 URL 可以出现的所有地点并且完成所需的加密（基于该目标页面的类参数），参看后面的说明。如果你证件按计划系统地建立一个页面，你可以使用%CSP.Page 类的 Link 方法来获得同样的行为。

如果你需要为一个在未被 CSP 编译器监测到的地点的.csp 文件中提供一个加密的 URL，可以使用#url()#指令。例如在一个客户机一侧的 JavaScript 函数中：

```
<script language=JavaScript>
function NextPage()
{
   // jump to next page
   self.document.location = '#url(nextpage.csp)#';
}
</script>
```

1.4.4.3　私有的页面

CSP 提供了一种“私有”的页面的概念。一个私有页面是只能在同一个 CSP 会话中被导航到别的页面或者从别的页面转来的。私有页面对于你想限制对某些页面的存取访问时是有用的。

例如，假设有一个叫做 private.csp 的私有页面（也是在 CSP samples 中的一个示例）。用户不能直接导航到 private.csp（例如靠键入它的 URL），它只能从另外一个 CSP 页面中含有的一个链接来导航到 private.csp。用户也不能为以后的使用作出一个私有页面的书签。因为用于保护私有页面的加密标示只是对于当前页面是有效的。

私有页面是这样工作的：

1. 把负责用于该页面的%CSP.Page 的子类的参数 PRIVATE 设置为 1。
2. 一个申请该页面的 URL 必须在它的字符串中包含一个有效的、已加密的“CSPToken”标记值。

3. 任何到这个被 CSP 处理的页面的链接将自动地有一个“CSPToken”值。

1.4.4.4　编码过的 URL 参数

类似于私有页面，一个 CSP 页面可以借助于设置%CSP.Page 的类参数 ENCODED 的值来指定它的 URL 参数必须编码。ENCODED 可以设置为 0、1、2。任何到它的 ENCODED 类参数的值是 1 或 2 的一个页面的链接，会自动地在被加密的“CSPToken”之内具有已编码的 URL 参数。如果 ENCODED 设置为 2，那么所有的值都要编码，如果设置为 1 则很用编码的和不编码的值是可能的。

例如，假设你有两个.csp 页面，其中一个（list.csp）显示银行各个帐户的一个列表作为超链接而另一个（account.csp）显示一个特定帐户的信息。account.csp 期待一个名称为“ACCOUNTID”的 URL 参数来判断哪个帐户要被显示。我们不希望在客户机上发现帐号，而且也不希望未经授权的人访问到 account.csp 或者有能力去显示任何别德帐号。我们可以借助于设置 account.csp 的 PRIVATE 类参数为 1 和设置它的 ENCODED 参数为 2 来实现这个需求。这里给出的是有关的.csp 各个文件：

list.csp 的源文件：

```
<html>
<body>
Select an account:<br>
<a href="account.csp?ACCOUNTID=100">Checking</a>
<a href="account.csp?ACCOUNTID=105">Saving</a>
</body>
</html>
```

account.csp 的源文件

```
<html>
<csp:class private=1 encoded=2>
<body>
```

```
Account Balance: <b>$#(..GetBalance())#</b>
</body>

<script language="CACHE" method="GetBalance" arguments=""
      returntype="%Integer">
   // server-side method to lookup account balance
   New id
   Set id = $Get(%request.Data("ACCOUNTID",1))
   If (id = 100) {
      Quit 157
   }
   ElseIf (id = 105) {
      Quit 11987
   }
   Quit 0
</script>

</html>
```

当 list.csp 被访问的时候，CSP 服务器将送出以下的 HTML 给客户端：

```
<html>
<body>

Select an account:<br>
<a href="account.csp?CSPToken=fSVnWw0jKIs">Checking</a>
<a href="account.csp?CSPToken=1tLL6NKgysXi">Saving</a>
</body>
</html>
```

注意：只是用于 **ACCOUNTID** 的加密过的值被送给客户机。

当 account.csp 被访问时，它看得见用于 ACCOUNTID 的已经被解密的值。（在它的 GetBalance 方法中被引用）。

1.5 用 CSP 进行基于标记的开发

为了使 HTML 开发者易于工作，CSP 提供了一种利用标准 HTML 开发 CSP 应用的机制。这个机制就是 CSP 编译器可以将标记好的 HTML（还有 XML）文档转换为能够影响 HTTP 各种申请的%CSP.Page 的类。由 CSP 编译器生成的各种类和手工建立的类没有什么不同的，而且是完全可以互相操作的。这使得开发者可以在基于 HTML 和在应用中系统地开发之间进行选择。而且，检验所生成的 CSP 类在调试时通常是有用的。

由 CSP 编译器处理的 HTML 文档中可以包括被编译器使用的特殊标记，来控制如何生成各种类、控制流程、管理数据访问和控制服务器一侧的行为。这些特殊的标记合在一起被称为 CSP 标记语言。重要的一点是要指出这些特殊的标记只是在开发时用在 CSP 服务器上的，而由 CSP 送给 HTTP 客户端的 HTML 是完全标准的并不含特殊的标记。

在 CSP 文件中，你可以使用正常的 HTML 标记，加上以下内容：

- Caché 数据表达式，#(…)#，它在页面生成的时候替换相应的值。
- Caché CSP 标记，<csp:XXX…>，提供内建的和自定义的功能。
- Caché 脚本，<script language=cache runat=server/complier>，它在页面生成或者页面编译时生成 Caché 代码。
- 服务器一侧的子程序调用，#server(…)#，它从客户机一侧的代码（Hyper-Events）调用服务器一侧的子程序。

此外，CSP 也为开发者提供了用增加自己定义的标记来扩展 CSP 标记语言的方法。如果需要详细的了解，可参考“开发自定义标记”一节。

1.5.1 CSP 编译器

CSP 编译器是一组运行在服务器一侧的 Caché 类和程序。它阅读和分析一个标记过的文档（利用 CSP 标记语言），使用匹配逻辑模式，生成一个 Caché 类，然后把这个类编译成为可执行的代码。

加入你有以下的简单 CSP 文档，hello.csp：

```
<html>
<body>
Hello!
</body>
</html>
```

CSP 编译器将把它转换成为一个 Caché 类，类似于：

```
Class csp.hello extends %CSP.Page
{

ClassMethod OnPage() As %Status
{
   Write "<html>"
   Write "<body>"
   Write "Hello!"
   Write "</body>"
   Write "</html>"
   Quit $$$OK
}
}
```

当从一个浏览器请求访问 hello.csp，CSP 服务器会调用所生成的 OnPage 方法，并且将原来的 CSP 文档的文本内容送给浏览器显示。

1.5.1.1 页面的自动和手动编译

在“自动编译”模式下（默认的），CSP 服务器会自动地要求 CSP 编译器将 CSP 源文件编译成为所需要的类。CSP 服务器能够比较各个源文件形成时间表示和类形成时间的标识，并且将重新编译那些源文件比它的类还要新

的页面。通常这个方式在已实施部署的应用中应当关闭掉，以避免不必要地再次核对时间标识（可以在 Caché Configuration Manager 中做到）。

你可以明确地指示将一个 CSP 源文件编译成一个类。这对于发现错误是很有帮助的。要这样做时你可以把 CSP 源文件装入 Caché Studio 并且编译它，当然你也可以通过 Caché Terminal 来做：

```
Do $system.CSP.LoadPage("/csp/user/mypage.csp","ck")
```

这个方法用 URL 路径（而不是实际的物理路径）装载和编译 CSP 文件“/csp/user/mypage.csp”，“c”标志指示出所生成的类应该被编译，而“k”标志指示出该生成的中间代码应该被保存。（这对于以后的纠错很有帮助）

1.5.2 CSP 标记语言

CSP 标记语言是一组指令和特殊标记，使得开发者能够控制由 CSP 编译器生成的类。

重要的是要了解处理一个 CSP 文档得到的结果将是一个执行 Caché ObjectScript 代码的 Caché 类。如果你不始终想着这一点，你或许会发现你在开发正确的应用逻辑和完成纠错时会感到很困难。事实上，你可以发现察看由 CSP 所生成的代码将是你更多地学习 CSP 和 CSP 标记语言是一个有用的方法。

另一方面，跟踪 CSP 服务器准备相应一个 HTTP 请求时在服务器上执行的代码和在 HTTP 端执行的代码（例如 HTML 和 Javascript）也是很重要的。

1.5.2.1 CSP 页面语言

默认情况下，CSP 编译器使用 Caché ObjectScript 生成运行时的表达式和代码，你可以通过在 CSP 文档的顶端加上下面这句来使编译器使用 Basic 语言生成代码：

```
<%@ PAGE LANGUAGE="BASIC" %>
```

你可以参考 http://127.0.0.1:1972/csp/samples/basic.csp 这个例子。

在 CSP 文档中，所有的运行时的表达式和任何服务器端的脚本标记的内容一样都必须使用默认的页面语言（否则编译时会出错）。注意，你可以在页面中通过定义不同语言而从默认的语言中调用它们的方法混合使用服务器端的语言。

1.5.2.2 文本

由于在 CSP 文档（HTML 或者 XML）中的任何文本信息不是 CSP 的指令或者特殊标记，它们将不被变动地传递给提出请求的 HTTP 客户端。

例如，某个 CSP 文档有以下内容：

```
<b>Hello!</b>
```

在所生成的类中将有以下内容：

```
Write "<b>Hello!</b>",!
```

接下来它会把下列内容发送给 HTTP 客户端：

```
<b>Hello!</b>
```

1.5.2.3 运行时的表达式

CSP 文档中可以包含一定的 Caché 表达式，这个表达式在页面被调用的时候在 CSP 服务器上运行。这样的表达式都是用“#(expr)#”来界定的，此处的 expr 是一个合法的 Caché ObjectScript 表达式。

例如，某个 CSP 文档包括：

```
Two plus two equals <b>#(2 + 2)#</b>
```

在所生成的类中有如下代码：

```
Write "Two plus two equals <b>", (2 + 2), "</b>",!
```

接着它会将以下内容送到 HTTP 客户端：

```
Two plus two equals <b>4</b>
```

你可以将运行时表达式应用于许多目的，例如饮用较早时候设置在页面上的一个变量的值：

```
The answer is <b>#(answer)#</b>.
```

引用对象的属性或者方法：

```
Your current balance is: <b>#(account.Balance)#</b>.
```

引用一个属于生成的类的方法（例如从%CSP.Page 继承来的一个方法）：

```
This page was processed by: <b>#(..ClassName())#</b>.
```

引用一个在%ResultSet 对象中的一个字段：

```
<table>
<csp:WHILE CONDITION="result.Next()">
<tr><td>#(result.Get("BookTitle"))#</td></tr>
</csp:WHILE>
</table>
```

利用%request 对象引用一个 URL 参数：

```
<table bgcolor='#(%request.Data("TABLECOLOR",1))#'></table>
```

运行时的表达式可以用在 CSP 文档的任何地方，在那里“#()#”可以被作为合法的 HTML 使用。这包括在 HTML 的文本中、作为一个 HTML 元素属性的值、或者在客户机一侧的 Javascript 定义中都可以使用。

如果一个运行时表达式是在一个 HTML 属性中被调用，那么在它被转换为可执行的代码以前是被转码的，例如：

```
<font size=#(1 &gt; 0)#>
```

在生成的类中有如下代码：

```
Write "<font size=",(1 > 0),">",!
```

如果表达式含有任何特殊的字符（例如“<”或者“>”），你必须对它们进行转码（利用由%CSP.Page 类提供的转码方法中的一种）来确保正确的转码序列能被传递给 HTTP 客户端，例如：

```
Description: <b>#(..EscapeHTML(object.Description))#</b>.
```

请参考“转码和引用 HTTP 输出”一节以获得详细信息。

1.5.2.4　运行时的代码

如果你在一个页面中需要比一个简单的表达式更多的东西，你可以利用 **<SCRIPT RUNAT=SERVER>**标记放上若干行服务器端的代码：

例如：某个 CSP 文档含有下列内容：

```
<ul>
<script language="cache" runat=server>
   For i = 1:1:4 {
      Write "<li>Item ",i,!
   }
</script>
</ul>
```

在生成的类里面将含有以下代码：

```
Write "<ul>",!
 For i = 1:1:4 {
   Write "<li>Item ",i,!
 }
 Write "</ul>",!
```

它将向客户机发送以下内容：

```
<ul>
<li>Item 1
<li>Item 2
<li>Item 3
<li>Item 4
</ul>
```

如同运行时的表达式一样，你可以将运行时代码用于各种不同的目的。

1.5.2.5　定义服务器端的代码

在 CSP 文档中，你可以为该文档定义一个属于所生成的类的方法。这是利用一个标准的 SCRIPT 标记的一个特殊变形来做到的。

你可以指定方法的名称以及指定它的参数列表和返回类型。

例如，下面定义了一个叫做“MakeList”的方法：

```
<script language="CACHE" method="MakeList"
arguments="count:%Integer" returntype="%String">
   New i
   Write "<ol>",!

   For i = 1:1:count {
      Write "<li> Item",i,!
   }
   Write "</ol>",!
   Quit ""

</script>
```

你可以在 CSP 文档中的任何地方引用这个方法：

```
<hr>
#(..MakeList(100))#
```

注意：你可以使用继承的特性（利用 **csp:CLASS** 标签）来继承以前定义过的方法到你的页面类中，或者你用其它别的类的各种类方法：

```
<hr>
#(##class(MyApp.Utilities).MakeList(100))#
```

1.5.2.6　SQL

你可以使用 SQL 来定义一个在 CSP 页面中的 Caché ResultSet 对象，其做法是利用标准 SCRIPT 标记的一个特殊版本。

下面的标记：

```
<script language="SQL" name="query">
SELECT Name FROM MyApp.Employee ORDER BY Name
</script>
```

可以为你建立一个动态 SQL %ResultSet 对象的一个名称为 query 的实例，准备好这个所指定的 SQL query，并且执行它（为了能多次重复调用它二做好准备）。

通常你可以结合利用 SQL SCRIPT 标记以及 csp:WHILE 标记来使用一个已建立过的%ResultSet 对象，来为你显示出 query 查询所得到的结果。

当页面完成执行后，SQL SCRIPT 标记会自动关闭建过实例的%ResultSet 对象。

你可以借助于在 SQL 文本中利用“?”字符来指定用于 SQL 查询的各个参数。你可以利用 SQL SCRIPT 标记的 P1，P2，…Pn 属性来为各个参数提供值。

下面是一个使用 SQL SCRIPT 标记来显示当前客户所购买的内容（假设该客户的 User ID 已经在存储在%session 对象之中了）：

```
<script language=SQL name=query P1='%session.Data("UserID")'>
SELECT DateOfPurchase,ItemName,Price
FROM MyApp.Purchases
WHERE UserID = ?
ORDER BY DateOfPurchase
</script>
<hr>
Items purchased by: <b>#(%session.Data("UserID"))#</b>
<br>
<table>
<tr><th>Date</th><th>Item</th><th>Price</th></tr>
<csp:WHILE CONDITION="query.Next()">
<tr>
<td>#(..EscapeHTML(query.GetData(1)))#</td>
<td>#(..EscapeHTML(query.GetData(2)))#</td>
<td>#(..EscapeHTML(query.GetData(3)))#</td>
</tr>
</csp:WHILE>
</table>
```

利用 csp:QUERY 标记，你可以使用一个已定义作为一个 Caché 类的一个组成部分的查询来建立一个%ResultSet 对象：

```
<csp:QUERY classname="MyApp.Employee" queryname="ByName">
```

你可以在你想用 SQL SCRIPT 时用同样的办法来使用这个能获得结果的%ResultSet 对象。

1.5.2.7　编译时的表达式和代码

你可以指定当某个 CSP 页面被编译时应当被计算的一个表达式的值（不同于运行时的）。这样的一些表达式通常在 CSP 规则定义中被使用。虽然有时也会在更一般的情况下使用它们。

编译时的表达式也是用#(expr)#来界定的，此处 expr 是一个合法的 Caché ObjectScript 表达式。

例如，某个 CSP 文档包含下列内容：

```
This page was compiled on: <b>##($ZDATETIME($H,3))##</b>
```

在所生成的类中有下列代码：

```
Write "This page was compiled on <b>2000-08-10 10:22:22</b>",!
```

类似地你可以定义在页面被编译时要执行的若干代码，这可以利用下列写法：

```
<script language="CACHE" runat="COMPILER">
```

注意：

你必须使用 Caché ObjectScript 编写编译时的表达式和代码。

1.5.2.8　控制生成的类

利用 csp:CLASS 标记，你可以对由 CSP 编译器所生成的类加以控制：这种控制包括选择一个或多个基类（super class）用于这个类，和定义一些值用作%CSP.Page 类的许多参数。

例如，假设你除了正常的%CSP.Page 类以外，你想要有一个生成的类来从另外的类来实现多重继承。你可以用 csp:CLASS 标记的 SUPER 属性来做这件事：

```
<csp:CLASS SUPER="%CSP.Page,MyApp.Utilities">
```

这将取得一个以逗号分隔的各个类的列表并且用它们作为各个基类用于所生成的类。

为了重新定义类参数 PRIVATE 的值（为了定义一个“private”私有的页面）：

```
<csp:CLASS PRIVATE=1>
```

1.5.3 流程控制

CSP 标记语言提供有一些特殊的标记来便于对各个页面的执行进行控制。这些标记可以使某些任务更加易于完成。

1.5.3.1 csp:IF 标记

<csp:IF>标记，连同<csp:ELSE>和<csp:ELSEIF>标记一同使用，提供了一种办法来定义在一个 CSP 页面中的有条件的输出。

<csp:IF>标记有一个简单的属性，CONDITION，它的值是一个由 CSP 服务器在运行时判断的 Caché ObjectScript 的表达式。如果它的值为真（True），那么该标记的内容将被执行。

举例来说：

```
<csp:IF CONDITION='user="Jack"'>
Welcome Jack!
<csp:ELSEIF CONDITION='user="Jill"'>
Welcome Jill!
<csp:ELSE>
Welcome!
</csp:IF>
```

1.5.3.2 csp:WHILE 标记

<csp:WHILE>标记提供的是一个办法，即只要一个给定的服务器一侧的条件为真时就来重复处理一个 CSP 文档中的一段。

<csp:WHILE>标记的 CONDITION 属性包含有一个 Caché ObjectScript 表达式，只要它的值为真，<csp:WHILE>标记中的内容就被执行。

<csp:WHILE>是通常和一个 Caché ResultSet 对象一起使用来显示一个用 HTML 书写的 SQL 查询的各个结果的，例如：

```
<script language=SQL name=query>
SELECT Name
FROM MyApp.Employee
ORDER BY Name
</script>

<csp:WHILE CONDITION="query.Next()">
#(..EscapeHTML(query.Get("Name")))#<BR>
</csp:WHILE>
```

在这个例子中，为输出该查询的“Name”列的值而使用的 csp:WHILE 标记的内容是重复执行，一直到%ResultSet 对象的 Next 方法在到达该结果集的末尾时将会回送出一个表示已到达末尾的 false 值时为止。

利用 csp:WHILE 标记的 COUNTER 属性你可以定义一个计数器变量，它会被初始化为零（0）并且在每次循环开始时自动地增加一个 1。

例如，这里的例子是一个利用 csp:WHILE 标记来输出一个 5 行 HTML 表格的例子：

```
<table>
<csp:WHILE COUNTER="row" CONDITION="row<5">
<tr><td>This is row #(row)#.</td></tr>
</csp:WHILE>
</table>
```

1.5.3.3　　csp:LOOP 标记

csp:LOOP 标记提供了另外一种重复执行 CSP 文档中的一个代码片断的方法：

csp:LOOP 标记让你能够定义一个计数器变量以及它的起始值、终了值和增量值（其默认增量值为 1）。

例如，你可以用 csp:LOOP 标记来举例说明一个含有 5 条项目的列表：

```
<ul>
<csp:LOOP COUNTER="x" FROM="1" TO="5">
<li>Item #(x)#
</csp:LOOP>
</ul>
```

1.5.4 转码和引用 HTTP 输出

当发出输出，特别是 HTML 的内容给一个 HTTP 客户端时有许多次程序不得不使用转码序列来避免和某些字符的特殊意义相冲突。例如为在 HTML 中显示“>”字符你不得不以“>”来对它进行转码。

一个文档或许在不同的部分会使用不同的转码规则（例如 HTML 和 JavaScript），这使得事情变得更加复杂。

为了有助于解决这个问题，%CSP.Page 类提供了许多转码和引用方法。一个转码（“escaping”）方法将一个字符串作为输入并且送回一个全部特殊字符由相应的转码序列替代的字符串。一个引用（“quoting”）方法将一个字符串作为输入并且送回一个被引用字符串（他是用相应的引号扩起来的）。该被引用的字符串也是全部特殊的字符由转码序列替换过的。

对于每一个转码方法，有一个相应的“不转码”方法（“unescape”），在其中则用普通的文本替代了转码序列。

1.5.4.1 给 HTML 转码

%CSP.Page 能用相应的 HTML 转码序列替换字符。例如，如果要某个.csp 文件在一个浏览器上显示出一个在服务器一侧的变量 x 的值，可以这样来对它进行转码：

```
#(..EscapeHTML(x))#
```

如果 x 的值是“<mytag>”，这将会导致下列文本被送到 HTTP 客户端：

```
&lt;mytag&gt;
```

类似地，在写出 HTML 各个属性的时候转码是必须的：

```
<input type="BUTTON" value="#(..EscapeHTML(value))#">
```

如果 value 的值是“<ABC>”，这将导致下列文本被送到 HTTP 的客户端：

```
<input type="BUTTON" value="&lt;ABC&gt;">
```

注意：在#()#两边的引号是为了确保所得的 HTML 属性值也是被引号括起来的。

当从一个数据库向一个客户机传送输出时，对它进行转码总是个好主意。例如，考虑下述用来将一个 user 的姓名送给一个 Web 页面的表达式（假设 user 是对于一个带有 Name 属性的对象的一个引用）：

```
User name: #(user.Name)#
```

如果你的应用让使用者输入他们的姓名给数据库，你可能会发现一个恶作剧的使用者也许输入了一个含有 HTML 命令的姓名，例如：

```
Set user.Name = "<input type=button onclick=alert('Ha!');>"
```

如果这被写出到一个 HTTP 客户端而没有进行转码的话，该页面会出现意料不到的情况。

1.5.4.2 URL 的参数

在 URL 字符串中的各个参数值也可以被转码。不幸的是 URL 用的是一个和 HTML 不同的转码序列的集合。但%CSP.Page 类的 EscapeURL 方法能将全部特殊的 URL 参数值用和它们相应的转码序列来替换。例如，如果某个.csp 文件使用一个服务器一端的变量 x 作为一个 URL 的参数，它可以这样转码：

```
<a href="page2?ZOOM=#(..EscapeURL(x))#">Link</A>
```

如果 x 的值是“100%”，那么下列文本将被发送给 HTTP 客户端：

```
<a href="page2?ZOOM=100%25">Link</A>
```

注意：“%”已经被转码成为“%25”。

1.5.4.3　　JavaScript

%CSP.Page 类的 QuoteJS 方法会转换一个字符串成为一个用引号括起来的 JavaScript 字符串并且把全部特殊的字符用和它们相应的 JavaScript 转码序列替换。例如，假设一个.csp 文件定义了一个客户机一侧的用于显示一个消息的 JavaScript 函数，该消息是由服务器一侧的在一个警示框中的一个变量 x 的值来指定的，这个 x 的值可以这样转换成为一个用引号括起来的 JavaScript 字符串：

```
<script language="JavaScript">
function showMessage()
{
   alert(#(..QuoteJS(x))#);
}

</script>
```

如果 x 的值是“Don’t press this button！”那么下列文本将被送给 HTTP 客户端：

```
<script language="JavaScript">
function showMessage()
{
   alert('Don\'t press this button!');
}
</script>
```

1.5.5 服务器端的方法

CSP 提供了一些技术来实现从 HTML 客户端调用服务器端的方法，这些技术包括：

- 使用 HTTP 提交的机制
- 使用**#server** hyper-event 超事件机制
- 使用**#call** hyper-event 超事件机制

所有这些都是基于行业标准的 HTTP 和浏览器的能力。每种技术都有它自己的优点和缺点，正确的选择取决于你的应用的特性。下表总结了各个技术的优缺点：

服务器端的方法

技术	优点	缺点
HTTP submit	客户端的编程很简单，不需要客户端的组件。	调用一个方法后整个页面都需要被刷新，而且对服务器端的编程带来困难。
#server hyper-event	允许通过 Java applet 直接从客户端调用服务器端的方法。服务器端的编程很简单。	客户端需要一个 Java 虚拟机，不是所有的浏览器都支持 applet。
#call hyper-event	允许从客户端直接调用服务器端的方法而不需要任何客户端的组件。服务器端的编程也很简单。	Asynchronous. Methods do not return values. Client-side of application has to be sensitive to timing issues.不同步，方法不能返回值。应用的客户端必须定时

1.5.5.1 通过 HTTP 的提交调用服务器端的方法

通过 HTTP 的提交来调用服务器端的方法是非常简单的。因为它对浏览器的要求很低，所以它对有广泛的用户和必须支持多种浏览器的应用来说是一项很好的技术。你可以通过下面的步骤掌握 HTTP 的提交：

1. 提供一个含有 SUBMIT 按钮的 HTML 表单（form）。

```
<form name="MyForm" action="MyPage.csp" method="GET">
User Name: <input type="TEXT" name="USERNAME"><br>
<input type="SUBMIT" name="BUTTON1" value="OK">
</form>
```

它定义了一个简单的表单，里面有一个叫做“USERNAME”的文本框和一个叫做“BUTTON1”的提交按钮。表单的 ACTION 属性被指定给要处理表单的 URL。METHOD 属性指定哪一个 HTTP 协议用来提交表单：“POST”和“GET”

2. 当用户点了 BUTTON1 这个按钮，浏览器就收集表单中所有控件的值并把它们提交给表单的 ACTION 属性指定的 URL。**注意：页面可以提交给**它自己。不管表单是通过 POST 还是 GET 提交的，CSP 把提交的值按照 URL 参数来对待。在这种情况下，提交表单等同于通过下面的 URL 发出请求：

```
MyPage.csp?USERNAME=Elvis&BUTTON1=OK
```

注意：SUBMIT 按钮的名字和值也包括在内。如果表单中有多个 SUBMIT 按钮，只有被按下的那个按钮才被包括在请求中。这是检查 SUBMIT 何时发生的关键。

3. 在这个例子中的 MyPage.csp 中的服务器端的代码，包含检查 submit 发生的逻辑。它是通过测试%request 对象中的“BUTTON1”来做到这点的：

```
<script language="CACHE" runat="SERVER">
  // test for a submit button
  If ($Data(%request.Data("BUTTON1",1))) {
```

```
        // this is a submit; call our method
        Do ..MyMethod($Get(%request.Data("USERNAME",1)))
    }
</script>
```

4. 调用完期望的服务器端的代码，服务器上的代码继续运行并且返回给浏览器显示的 HTML。这就可以重新显示当前的表单或者一个完全不同的页面。

1.5.5.2　通过#server 指令调用服务器端的方法

CSP 提供了一种能够调用 Caché 应用服务器上的类的方法以响应客户端浏览器上的事件而不用在客户端上重新装载页面的能力。这点在一些情况下很有用。在任何地方你都能使用 JavaScript 来处理一个在客户端上的事件，这样你就可以调用 Caché 服务器上的方法。CSP 通过下面的动作执行：

1.自动在定义了服务器端事件的页面上放置一个小的 Java applet（CSP Event Broker）。

2.使用 Event Broker 来制作一个加密的 HTTP 请求通过 Web 服务器访问 Caché 服务器。Caché 服务器通过执行代码相应请求并且把适当 JavaScript 送回客户机，然后在客户机上执行这些语句。

这个技术是以行业标准机制为基础的并且可以在任何支持 Java 的浏览器上工作。和 CSP 相集成使得易于使用和管理。

在 CSP 文件中，你可以使用**#server** 指令调用服务器端的方法。你可以在 JavaScript 允许的任何地方使用这个指令。**#server** 指令的语法是：

```
#server(classname.methodname(args,...))#
```

classname 是服务器端 Caché 类的名字，而 methodname 是这个类里面方法的名字，args 是客户端 JavaScript 传递给服务器端方法的参数列表。例如，要调用 Caché 服务器上面的 MyPackage 类里面的 Test 方法，使用：

```
<script language="JavaScript">
function test(value)
```

```
{
   // invoke server-side method Test
   #server(MyPackage.Test(value))#;
}
</script>
```

CSP 编译器使用调用服务器端方法的 JavaScript 代码替换每一个 **#server** 指令。从给出的 CSP 页面，你可以使用“..MethodName”这样的语法调用为它生成的类的方法，例如：

```
#server(..MyMethod(arg))#
```

4.5.5.3 通过#call 指令调用服务器端的方法

#call 指令提供了不同于**#server** 的另外一种调用 Caché 应用服务器上的类的方法以响应客户端事件的机制。

#call 除了下面的不同以外和**#server** 的用处是一样的：

1.**#call** 不使用客户端上的 Java Applet，取而代之的是它使用纯 HTML 来向服务器提交请求。取决于浏览器，这可能是通过 HTML IFRAME 标记或者 ILAYER 标记（CSP 自动判断浏览器类型并使用正确的标记）。

2.**#call** 是异步的：当你调用服务器端的代码时，**#call** 不等待返回值。实际上，你不能通过使用#call 取得返回值。取而代之的是你的应用依赖服务器送回到客户端上的 JavaScript 代码执行一些操作。因为异步的特征，当遇到使用多个连续的#call 的时候你要小心了：**如果你在上一个方法完成前通过#call 调用一个方法，Web 服务器也许会取消你上一个方法的调用。**

通常情况下，如果你的应用不需要同步的行为，你最好使用**#call** 以避免在客户端使用附加的 Java Applet。和**#server** 一样，你可以在 JavaScript 允许的任何地方使用**#call** 指令。使用**#call** 指令的语法是：

```
#call(classname.methodname(args,...))#
```

classname 是服务器端 Caché 类的名字，而 methodname 是这个类里面方法的名字，args 是客户端 JavaScript 传递给服务器端方法的参数列表。例如，要调用 Caché 服务器上面的 MyPackage 类里面的 Test 方法，使用：

```
<script language="JavaScript">
function test(value)
{
  // invoke server-side method Test
  #call(MyPackage.Test(value))#;
}
</script>
```

CSP 编译器使用调用服务器端方法的 JavaScript 代码替换每一个**#call** 指令。从给出的 CSP 页面，你可以使用“..MethodName”这样的语法调用为它生成的类的方法，例如：

```
#call(..MyMethod(arg))#
```

1.5.5.3　　从服务器端响应客户端事件

你可以使用**#server** 和**#call** 指令来执行服务器动作以响应客户端的事件。例如，假设你有一个窗体用来增加一个新客户到数据库中。一旦输入了客户的姓名，应用程序就会进行核对来弄清楚该客户是否在数据库中存在。当输入内容改变时，下述窗体定义将调用一个服务器一侧的 Find 方法。

```
<form name="Customer" method="POST">
Customer Name:
<input type="Text" name="CName"
onChange=#server(..Find(document.Customer.CName.value))# >
</form>
```

此例中，Find 方法可以在同一个 CSP 文件中这样定义：

```
<script language="CACHE" method="Find" arguments="name:%String">
 // test if customer with name exists
 // use embedded SQL query

 New id,SQLCODE
 &sql(SELECT ID INTO :id FROM MyApp.Customer WHERE Name = :name)

 If (SQLCODE = 0) {
   // customer was found
```

```
  // send JavaScript back to client
  &js<alert('Customer with name: #(name)# already exists.');>
 }
</script>
```

这个方法是靠执行送回的 JavaScript 脚本和客户端通信的。不论什么时候一个服务器端的方法被调用，任何写到主要设备的输出都被送回到客户端，在那里被转换成一个 JavaScript 函数，并且在客户端的浏览器中的该页面中执行。例如，如果一个服务器端的方法如下：

```
Write "self.document.title = 'New Title';"
```

那么下列的的 JavaScript 代码被送回到客户端并被执行：

```
self.document.title = 'New Title';
```

在这个例子里，我们要更换客户端页面的主题为“New Title”。任何合法的 JavaScript 都能被以这种方式送回到客户端。注意：你必须在 JavaScript 的每行后面使用“!”来替换回车，否则浏览器将不能执行它。为了使从服务器方法中返回 JavaScript 更加容易，Caché ObjectScript 支持使用“&js<>”指令“嵌入 JavaScript”。这是一种特殊的语言结构，使你可以在一个 Caché ObjectScript 方法中包含多行 JavaScript 代码，&js<>指令的内容被转换成相应的 Write 命令语句。嵌入的 JavaScript 可以使用#()#指令引用 Caché ObjectScript 表达式。例如，一个 Caché 方法包含下列代码：

```
Set count = 10
 &js<
    for (var i = 0; i &lt; #(count)#; i++) {
        alert('This is annoying!');
       }
 >
```

它等同于：

```
Set count = 10
 Write "for (var i = 0; i < ", count, "; i++) {",!
 Write "    alert('This is annoying!');",!
 Write "}",!
```

当从客户端调用的时候，这个方法显示 10 次警告信息。

1.5.5.4 使用服务器端的方法的技巧

从 Web 页面调用服务器端的方法是一个强劲的特性。但是，你应当记住什么时候在应用中应该使用服务器端的方法。

注意：

在这一节，任何提到**#server** 的如果不特别声明也适用于**#call**。

CSP 特别提供的**#server** 和**#call** 允许你在 Web 浏览器中的 JavaScript 里面调用 Caché 服务器上的方法。这使得 CSP 能够做到你从一个校验框离开的时候就做一些操作，而不用等待提交表单，从而立刻得到反馈。这儿有一些使用**#server** 的要素要注意，否则它可能导致营养能够变得非常慢甚至有时根本不能工作。

你要记住这儿有两个基本的原则来使用**#server** 和**#call**：

1. 不要在 Web 页面的 onload 事件中使用**#server**。这有时会失败，并且当在 Caché 中生成页面的时候很快也很容易会产生数据。
2. 不要在 Web 页面的 onunload 事件中使用**#server**。尽量少的使用**#server**，并且尽量每次使用的时候做尽量多的事情，因为这个操作的代价是非常高的，它要执行从客户端到服务器的往返操作。

在 onload 事件中使用**#server** 不是一个好注意的第一个原因，因为 onload 事件是只有当 Web 浏览器读完 HTML 页面触发的事件，它不等待这个页面上的任何 applet 的装入，或者这个页面的任何文件的引用，只是完成页面的主体部分。**#server** 调用的工作要使用浏览器装入的 Java applet，所以它是否能正常工作的前提依赖于 Java Applet 是否已被完全装入。当 Java applet 还没有被装入时试图使用**#server** 就会出错。这个问题当你处于一个低速的网络下因此需要更长的时间下载 Java applet 的时候经常变得更加糟糕。第二个原因是因为任何你需要在 onload 事件中运行的代码可以在页面生成以后运行的更快和更加

容易。例如，假设你希望为 JavaScript 变量设置一个初始值以便于以后使用，所以你现在可能这样做的：

```
<html>
<head>
<script language="JavaScript">
function LoadEvent()
{
   var value=#server(..GetValue())#;
}
</script>
</head>
<body onload=LoadEvent();>
</body>
</html>
<script language="CACHE" method="GetValue" returntype="%String">
   Quit %session.Data("value")
</script>
```

然而这儿绝对没必要使用#server，因为 JavaScript 变量的值在你产生页面时的%session.Data(“value”)中是已知的，所以最好写成：

```
<html>
<head>
<script language="JavaScript">
function LoadEvent()
{
   var value='#(%session.Data("value"))#';
}
</script>
</head>
<body onload=LoadEvent();>
</body>
</html>
```

同样的窍门可用在很多地方，例如：

```
<input type="text" name="TextBox" value='#(%request.Get("Value"))#'>
```

这儿没有任何必要在页面的 onload 事件中使用**#server**。

类似的，在 onunload 事件中使用#server 或者#call 也可能引起问题，没有办法保证在 onunload 事件发生的时候 Java Applet 仍然是装入的状态。因为页面在关闭的时候很难知道浏览器的行为，更别说用户直接关闭机器时你根本无法得到 onunload 事件了。

尽量少使用**#server** 和**#call**

#server 和**#call** 都通过浏览器用一个特别加密的符号为页面提交 HTTP 请求以告诉 Caché 方法的名字以运行。Caché 运行这个方法，任何的输出都被作为 JavaScript 送回并在浏览器上执行，另外**#server** 调用能返回一个值。因为这些调用都使用 HTTP 请求，所以就带来了网络和服务器上 CPU 的开销，并且这些调用是作为正常的 CSP 页面请求的。如果你使用大量的**#server** 请求，那么这将戏剧性地引起应用程序可用性的降低。因为每个**#server** 都从 Caché 服务器要一个新的 CSP 页面。这意味着你通过 URL 产生一次页面得到一个常规的页面，而一个有 10 个**#server** 调用的 CSP 页面将产生 10 个 CSP 页面，所以如果你减少**#server** 调用你就能增加应用能够支持的用户数。

减少使用**#server** 的方法是确定应用程序是否真正需要使用**#server**。例如这个例子，一些 JavaScript 用一些从服务器来的数据更新表单：

```
<script language="JavaScript">
function UpdateForm()
{
   self.document.form.Name.value = #server(..workGet("Name",objid))#;
   self.document.form.Address.value = #server(..workGet("Address",objid))#;
   self.document.form.DOB.value = #server(..workGet("DOB",objid))#;
}
</script>
```

服务器代码是（正常情况下它也许是使用对象或者 SQL，但是这里我们使用 global 来使代码更短一些）：

```
<script language="CACHE"
        method="workGet"
        arguments="type:%String,id:%String"
        returntype="%String">
    Quit $get(^work(id,type))
</script>
```

这个更新使用了 3 个调用！这其实可以转换成使用单个**#server** 调用一次更新所有的值，JavaScript 代码如下：

```
<script language="JavaScript">
function UpdateForm()
{
    #server(..workGet(objid))#;
}
</script>
```

方法定义如下：

```
<script language="CACHE"
            method="workGet"
            arguments="id:%String"
            returntype="%String">
    &js<self.document.form.Name.value = #($get(^work("Name",objid)))#;
        self.document.form.Address.value = #($get(^work("Address",objid)))#;
        self.document.form.DOB.value = #($get(^work("DOB",objid)))#;>
</script>
```

所以取代多个调用你只要传递一次数据然后让 Caché 做全部的工作就可以了。如果你有更为复杂的 JavaScript 例子，例如：

```
<script language="JavaScript">
function UpdateForm()
{
    self.document.form.Name.value = #server(..workGet("Name",objid))#;
    if (condition) {
        self.document.form.DOB.value = #server(..workGet("DOB",objid))#;
    }
    else {
```

```
      self.document.form.DOB.value = '';
   }
}
</script>
```

那么这将仍旧需要一个#server 调用，你只要把全部的 if 条件嵌入到#server 返回的 JavaScript 中去就可以，所以 workGet 方法的代码最后看起来像：

```
<script language="CACHE"
            method="workGet"
            arguments="id:%String"
            returntype="%String">
   &js<self.document.form.Name.value = #(^work("Name",objid))#;
      if (condition) {
         self.document.form.DOB.value = #(^work("DOB",objid))#;
      }
      else {
         self.document.form.DOB.value = '';
      }
   >
</script>
```

注意：

你还要注意的是由于活动的连接和 Java 安全的问题，**#server** 语法不能在 Netscape 6.0 和 6.1 或者 IE 在 Macintosh 计算机上的版本上工作。**#server** 需要浏览器已经安装了 Java 虚拟机。在**#call** 语法中论述了这些情况。

1.6 使用 Caché Server Page 建立数据库应用

CSP 的最强大功能之一就是可以让你建立能直接和内建的对象数据库打交道的动态的 Web 页面。 这意味着你可以快速地建数据库各种应用。而能：

- 避免了把关系数据映射成对象的复杂工作。
- 不需要使用复杂的中间件软件。
- 在运行时可在结构上进行从单一的服务器到多层、多服务器的重新配置，以获得真正灵活的可伸缩能力。

注意：利用 Caché SQL Gateway，你可以建立以对象为基础的 CSP 应用，它能在第三方的关系数据库中存取数据。Caché 管理这些事是以一个应用透明的方式进行的。所有在本章中描述的技术不论你是选择将数据存储在内建的 Caché 数据库里还是存储在一个第三方的数据库中都是可以工作的。

CSP 是灵活的，你可以用不同的技术建立数据库应用， 这些技术的范围可以从采用能自动地将数据结合进入 HTML 窗体的高级的标记；或书写服务器一侧的脚本来利用对象去直接访问数据。这些技术在概述如下：

1.6.1 使用页面上的对象

Caché 使得为一个应用建立表示数据的持久对象成为容易的事情。你可以在一个 Web 应用中以多种上方式来使用这些持久对象。在一个页面上显示对象数据的最直截了当的方式是用服务器一册的脚本代码来打开该对象和写出它的内容。下面的例子使用的是已包括在 Caché 数据库里 Project Sample 中的 Sample.Person 类。这些例子采用 CSP 页面，但所描述的的技术也适用于用 %CSP.Page 类的子类建立的应用。

1.6.1.1 在表格中使用对象数据

下面的 CSP 页面打开一个持久对象的实例，将该对象的一些属性显示在一个 HTML 表中。并且接着关闭该对象。

```
<html>
<body>
<script language="CACHE" runat="SERVER">
 // open an instance of Sample.Person
 Set id = 1
 Set person = ##class(Sample.Person).%OpenId(1)
</script>
<table border="1">
<tr><td>Name:</td><td>#(person.Name)#</td></tr>
<tr><td>SSN:</td><td>#(person.SSN)#</td></tr>
<tr><td>City:</td><td>#(person.Home.City)#</td></tr>
<tr><td>State:</td><td>#(person.Home.State)#</td></tr>
<tr><td>Zip:</td><td>#(person.Home.Zip)#</td></tr>
</table>
<script language="CACHE" runat="SERVER">
 // close the object
Set person = ""
</script>
</body>
</html>
```

如果你想要尝试做这件事，可能将上述代码拷贝到一个文本文件之中，将它作为“mytable.csp”文件存储在你的“/cachesys/csp/samples”目录之中（cachesys 是用于 Caché 的安装目录），并且将你的浏览器指向：

```
http://localhost:1972/csp/samples/mytable.csp
```

这时你应当能看见数据已被显示在一个简单的 HTML 表中，

注意：

要小心不要在“/csp/samples”上做建立你的应用的任何实际工作。否则如果你以后安装一个 Caché 的未来更新版本时，它会重新安装各个 sample 例子并且删除掉你的工作的。

1.6.1.2　　在表单中使用对象数据

所使用的代码类似于上述代码，你可以在一个 HTML 表单中显示数据：

```
<html>
<body>
<script language="CACHE" runat="SERVER">
 // open an instance of Sample.Person
 Set id = 1
 Set person = ##class(Sample.Person).%OpenId(1)
 If ($Data(%request.Data("SAVE",1))) {
   // If "SUBMIT" is defined, then this is a submit
   // Write the posted data into the object and save it
   Set person.Name = $Get(%request.Data("Name",1))
   Set person.SSN = $Get(%request.Data("SSN",1))
   Set person.Home.City = $Get(%request.Data("City",1))
   Do person.%Save()
 }
</script>
<form method="POST">
<br>Name:
<input type="TEXT" name="Name" value="#(..EscapeHTML(person.Name))#">
<br>SSN:
<input type="TEXT" name="SSN" value="#(..EscapeHTML(person.SSN))#">
<br>City:
<input type="TEXT" name="City" value="#(..EscapeHTML(person.Home.City))#">
<br>
<input type="SUBMIT" name="SAVE" value="SAVE">
</form>
<script language="CACHE" runat="SERVER">
 // close the object
Set person = ""
</script>
</body>
</html>
```

这个例子打开一个持久对象的一个实例，将它的一些属性显示在一个 HTML 表单中，并且接着关闭该对象。

1.6.1.3　处理一个表单提交的请求

除了能在一个表单内显示一个对象的内容以外，前述示例也会在使用者按“Save”按钮来提交该表单，能将所做作出的改动存储到该对象之中。这将像下面这样工作：

当一个表单被提交时，各种控制件（包括初始化这个提交的按钮）的值都有被送回给服务器。在这种情况下，该表单被提交给同一个即原来为该页面提供服务的 CSP 页。你可以借助于设置表单的 ACTION 属性来提交 给另外一个不同的页。

CSP 服务器将所提交的各个值放入%request 对象的 Data 属性。在服务器一侧的脚本程序会在该页开始时测试判断该正在被服务的页是否是响应一个提交申请的，即靠判断申请参数“Save”（它是提交按钮的名称）是否是已定义过的。这应当只是作为一个提交申请的结果而被定义的。如果判断出它的确是一个提交的申请，那么脚本将接着从表单拷贝所提交的各个值到该对象的相应的属性之中。并且去调用该对象的%Save 方法：

```
If ($Data(%request.Data("SAVE",1))) {
   // If "SUBMIT" is defined, then this is a submit
   // Write the posted data into the object and save it
   Set person.Name = $Get(%request.Data("Name",1))
   Set person.SSN = $Get(%request.Data("SSN",1))
   Set person.Home.City = $Get(%request.Data("City",1))
   Do person.%Save()
}
```

1.6.1.4　CSP:OBJECT 标记

在上述例子中的一些行为是自动地由 csp:OBJECT 标记提供的。根据 csp:OBJECT 标记会自动生成在服务器一侧为建立或打开一个用于 CSP 页面上

的对象实例所需的代码，以及用于关闭它的代码。例如，将一个对象类的实例和一个页面联系起来：

```
<CSP:OBJECT NAME="person" CLASSNAME="Sample.Person" OBJID="1">
<!-- Now use the object -->
Name: #(person.Name)# <br>
Home Address: #(person.Home.Street)#, #(person.Home.City)# <br>
```

在此例中，the csp:OBJECT 标记打开一个 Objest Id 为 1 的 CLASSNAME 类的对象并且将它赋值给 person 变量。更为贴近真实情况的是， object ID 是由%request 对象提供的：

```
<CSP:OBJECT NAME="person" CLASSNAME="Sample.Person"
OBJID='#($Get(%request.Data("PersonID",1))#'>
Name: #(person.Name)# <br>
Home Address: #(person.Home.Street)#, #(person.Home.City)# <br>
```

表达式，

```
$Get(%request.Data("PersonID",1))
```

引用的是 URL 参数 PersonID。

带有一个空白的 OBJID 属性的 csp:OBJECT 标记将建立所指定类的一个新的对象：

```
<CSP:OBJECT NAME="person" CLASSNAME="Sample.Person" ObjID="">
```

使用 csp:OBJECT 标记是相当于包括服务器一侧明显建立一个对象实例的脚本一样。可参考作为 CSP 例子的 object.csp，它是一个使用 csp:OBJECT 标记的例子（http://127.0.0.1:1972/csp/samples/object.csp）。

1.6.1.5　　给表单绑定数据

CSP 为把对象的数据绑定到表单上提供了一个机制。该绑定适用标准的 HTML 表单和输入控件标记来定义表单，该表单允许你很容易地使用 HTML 编辑器或者设计工具。<CSP:OBJECT>标记指定一个对象实例，并把属性“CSPBIND”增加到各个表单和输入控件标记上来指定它们应当如何被绑定。

CSP 便一起能识别含有“CSPBIND”属性的各个表单并且自动生成代码来：

- 在合适的控件上显示指定的对象的属性的值。
- 生成客户端的 JavaScript 函数来执行简单的确认（例如是否必填字段）。
- 生成服务器端的能和对以及能把数据保存进表单的方法。这些方法可以从页面使用 CSP Event Broker 直接调用，或者你可以作为表单提交操作的结果调用它们。
- 在表单中生成一个隐藏的 OBJID 字段，它存储该绑定窗体的对象的 ID 值。

这儿是一个绑定 **Sample.Person** 类的实例到一个表单的简单的例子：

```
<html>
<head>
</head>
<body>
<CSP:OBJECT NAME="person" CLASSNAME="Sample.Person" OBJID="1">
<form NAME="MyForm" cspbind="person">
<br>Name:
<input type="TEXT" name="Name" cspbind="Name" csprequired>
<br>SSN:
<input type="TEXT" name="SSN" cspbind="SSN">
<br>City:
<input type="TEXT" name="City" cspbind="Home.City">
<br>
<input type="BUTTON" name="SAVE" value="SAVE" OnClick="MyForm_save();">
</form>
</body>
</html>
```

这个例子使用了<CSP:OBJECT>标记来打开一个 **Sample.Person** 类的实例（在本例中，对象 ID 是 1）。这个对象实例被称作 *person*。这个例子然后

通过给 form 标记里面的叫做 CSPBIND 的属性赋值为“person”来绑定这个对象实例到一个 HTML 表单。该表单包含三个文本输入控件：“Name”，“SSN”和“City”，它们通过给每个 input 标记添加叫做“CSPBIND”的属性来分别绑定到对象的 Name、SSN 和 Home.City 属性上。

注意在绑定的表单中使用的控件的名字必须是合法的 JavaScript 标示符。

“Name”控件还有一个叫做“CSPREQUIRED”的属性。这表示这是一个必填字段。CSP 编译器会生成客户机一侧的 JavaScript 来判断使用者是否为这个字段提供了一个值。

在表单上的最后一个控件是一个按钮，它被定义成当该按钮被点击时便调用客户机一侧的 JavaScript 函数 MyForm_Save。MyForm_Save 方法是自动地由 CSP 编译器生成的。这个方法收集表单中各个控件的值并且把它们传递给服务器一侧的一个方法（这个方法也是 CSP 自动生成的），该方法能重新打开这个对象的实例，将已经做出的改动修改到不同的属性上去，然后存储在数据库中，然后转送 JavaScript 到客户机来更新表单里的各个值使其反映出保存的结果。

注意：我们已经在这个文档中包括了 HEAD 一段。当使用一个绑定的表单用到了 CSP 编译器处理一个绑定的表单时生成的客户端的任何 JavaScript 代码时这便是必须的。

根据约定，在绑定的表单中使用的对象 ID 被 URL 参数“OBJID”指定的时候。这使得绑定的表单和前面建立的页面交互成为可能，例如那些被 CSP 搜索功能使用的页面。要使用 URL 参数的值作为对象 ID，可以在 CSP:OBJECT 标记中使用一个表达式引用它：

```
<CSP:OBJECT NAME="person"
CLASSNAME="Sample.Person" OBJID=#($G(%request.Data("OBJID",1))#>
```

1.6.1.6　绑定一个属性

要绑定一个指定的 HTML 输入控件到一个对象的属性上去，你必须做以下几件事：

- 定义一个使用 CSP:OBJECT 标记引用一个对象实例的变量。
- 使用 form 标记创建一个 HTML 表单。通过给 form 标记增加“CSPBIND”属性绑定表单到对象的实例。“CSPBIND”属性的值必须是 CSP:OBJECT 标记的名字。
- 在表单重创建一个 HTML 的输入控件，然后把“CSPBIND”属性添加给它。“CSPBIND”属性的值必须是你希望绑定的对象属性的名字。

CSPBIND 属性使你可以绑定不同类型的对象属性。这个细节请看下表：

CSPBIND 的属性的作用

属性	例子	作用
Literal	CSPBIND=“Name”	绑定控件到一个字面属性。显示属性 DISPLAY 的值。
Property of Embedded Object	CSPBIND=“Home.City”	绑定控件到一个嵌入的对象属性。显示这个属性的 DISPLAY 的值。
Referenced Object	CSPBIND=“Company”	绑定控件到对象的 ID。显示的是引用属性的对象的 ID 值。
Property of Referenced Object	CSPBIND=“Company.Name”	绑定控件到一个引用的对象的属性。显示的是这个属性的 DISPLAY 的值。
Instance	CSPBIND=“%Id()”	绑定控件到一个有返回值的

Method		实例方法。显示的是方法的返回值。这个字段是只读的。

绑定的机制可以被大多数 HTML 输入控件使用，这个细节请看下表：

CSPBIND 支持的 HTML 输入元素

控件	作用
INPUT TYPE="TEXT"	在文本控件中显示属性的值。
INPUT TYPE="PASSWORD"	在密码框中显示属性的值
INPUT TYPE="CHECKBOX"	在 checkbox 控件上显示属性的值（布尔值）
INPUT TYPE="RADIO"	通过选择 radio 按钮显示属性的值——符合属性的值。
INPUT TYPE="HIDDEN"	在隐藏的控件中显示属性的值。
SELECT	通过选择在 SELECT 列表里的符合属性值的选择项显示一个属性的值。 你可通过指定 CLASSNAME、QUERY 和可选的 FIELD 属性使用一个类查询更新 SELECT 列表中的选择项。请参考 CSP 的例子 *form.csp*。
IMAGE	用 IMAGE 标记显示一个二进制流的属性。
TEXTAREA	在 TEXTAREA 里面显示属性的值。

1.6.2 CSP 的搜索页面

CSP:SEARCH 标记提供了对一个通用搜索页面的访问，利用这个搜索页面你可以结合绑定的表单来执行查找的操作。应用程序的用户可以从一个包

含绑定的表单的页面调用 CSP 搜索页面并使用它在数据库中查找符合特定标准的对象。用户可以选择这些对象之一然后编辑它。

CSP:SEARCH 标记生成一个客户端的 JavaScript 函数来显示搜索页面。搜索页面通过 CSP 类库提供的**%CSP.PageLookup** 类显示。
CSP:SEARCH 也许包含一定数量的属性使你控制搜索页面的全部操作，这些属性包括：

CSP:SEARCH Tag Attributes

属性	描述
CAPTION	在标准的搜索页面中显示的可选标题。
CLASSNAME	必须的。搜索根据这个类名执行。
FEATURES	可选字符串，包含当一个弹出搜索窗口时要传送给 JavaScript 的 window.open 方法的特征参数。这给了你更大的灵活性来控制弹出窗口是如何显示的。
MAXROWS	指定搜索结果表中的最大行数，默认值是 500。
NAME	必须的，是生成的客户端一侧的调用搜索页面的 JavaScript 函数的名字。
OBJID	当调用搜索页面的时候显示的对象 ID 的值。这还被用于当用户取消搜索以后重新显示旧的页面。
ONSELECT	在弹出搜索页面的情况下，当用户选择一个指定的搜索结果时调用的 JavaScript 函数的名字。这个函数和选定的对象的 ID 一起被调用。
OPTIONS	是在一个搜索页面中的用逗号分开的可选的列表。这些可选择的功能包括用于建立一个弹出窗口的“popup”和用于显

	示预期结果的下拉列表“predicates”。
ORDER	搜索结果根据它进行排序的字段，可选的。
SELECT	在搜索结果中显示的各个字段，用逗号分开。如果没有指定，WHERE 列表就作为 SELECT 列表使用。
STARTVALUES	调用搜索页面的表单的控件的名字的列表（用逗号分隔开的），这些控件的内容被作为了搜索页面的种子。空间名字的顺序符合条件字段（WHERE 属性指定的）。
TARGET	在非探出搜索页面的情况下，指定要和搜索页面窗口地点做链接的页面的名字。就是当用户作出选择后显示的页面。默认的是搜索调用的页面。
WHERE	必须的。是以逗号分开的各个字段用于搜索页面判断搜索条件的列表。这些字段也被显示在搜索结果页面中，除非 SELECT 属性被指定了。

例如，下面定义了一个 JavaScript 函数，MySearch，这个函数显示一个弹出搜索窗口来通过名字搜索 **Sample.Person** 的对象：

```
<CSP:SEARCH NAME="MySearch" WHERE="Name" CLASSNAME="Sample.Person"
 OPTIONS="popup" STARTVALUES="Name" ONSELECT="MySearchSelect">
```

这个搜索页面的 ONSELECT 回调函数看起来像：

```
<script language="JavaScript">
function MySearchSelect(id)
{
   #server(..MyFormLoad(id))#;
   return true;
}
</script>
```

这个函数使用 CSP 的#server()#指令调用服务器端的方法——MyFormLoad。MyFormLoad 方法是作为使用 CSPBIND 绑定的 HTML 表单——MyForm 的结果自动生成的。这个方法用对象 ID 为 id 的对象属性的值更换表单的内容。

要获得更多的例子，可以参考 CSP 示例页面 *form.csp* 和 *popform.csp*。

第八章 平行数据迁移和 Caché SQL

1 引言

应用开发商或最终用户们除了选择直接从一开始就采用 Caché 开发数据库应用系统以外，有的可能希望将原有的已在传统的关系数据库（例如 Oracle 或 SQL Server）上的应用系统程序和数据迁移到 Caché 上运行。这也是有可能实现的。为此，Caché 提供有一些数据迁移（Data Migration）用的工具，例如 SQL Gateway 网关、ODBC 数据源管理器、和 Data Migration Wizard 即数据迁移向导等工具，可以方便地将传统关系型数据库迁移到 Caché 数据库。通过这种平行迁移，不仅可作数据的迁移，而且，原有 SQL 访问方式的应用程序可以不改动或较少改动就可以在 Caché 上运行，同时你可以用面向对象的方式在 Caché 上开发新的应用模块。

通过平行数据迁移，原有关系型的表在 Caché 中编译出一个对应的类，可以通过继承让它拥有你期望的功能，如使其继承于 XML.Adaptor，这样它就拥有了 XML 导入导出功能

2 迁移方式

数据迁移工作可通过数据库网关将其他关系型数据库迁移到 Caché，表可以直接迁移、视图通过 DDL 在 Caché 中重建、存储过程和触发器需要小量改写。

如果只是通过 DDL 在 Caché 中重建数据库模型，可以使用 Caché 提供的系统类 **%SYSTEM.SQL**，该类提供 **MSSQLServer()**、**Oracle()**、**Informix()**、**Sybase()** 等方法可以导入各种常用关系型数据库产生的 DDL。

2.1 数据迁移步骤

以作 SQLServer 的迁移为例

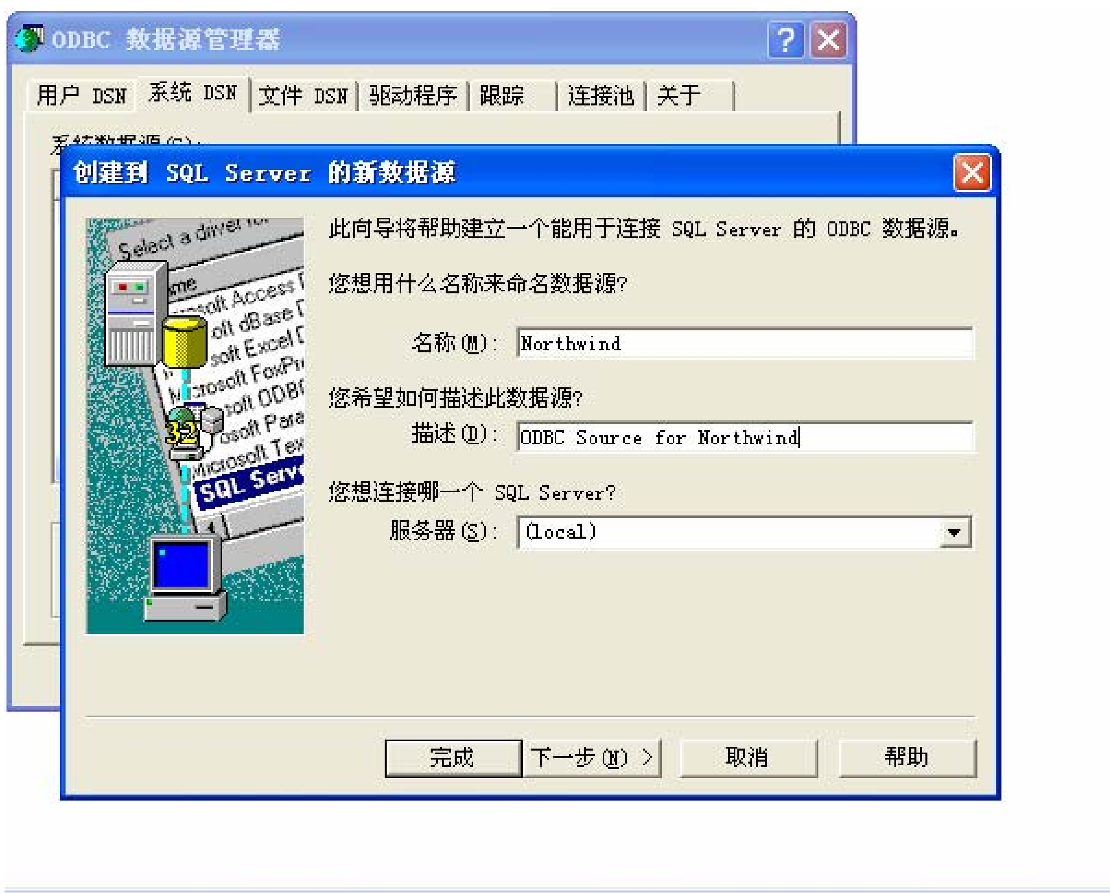
2.2 建立源数据库的 ODBC 数据源

在控制面板->管理工具->数据源(ODBC)中新建“系统 DSN”

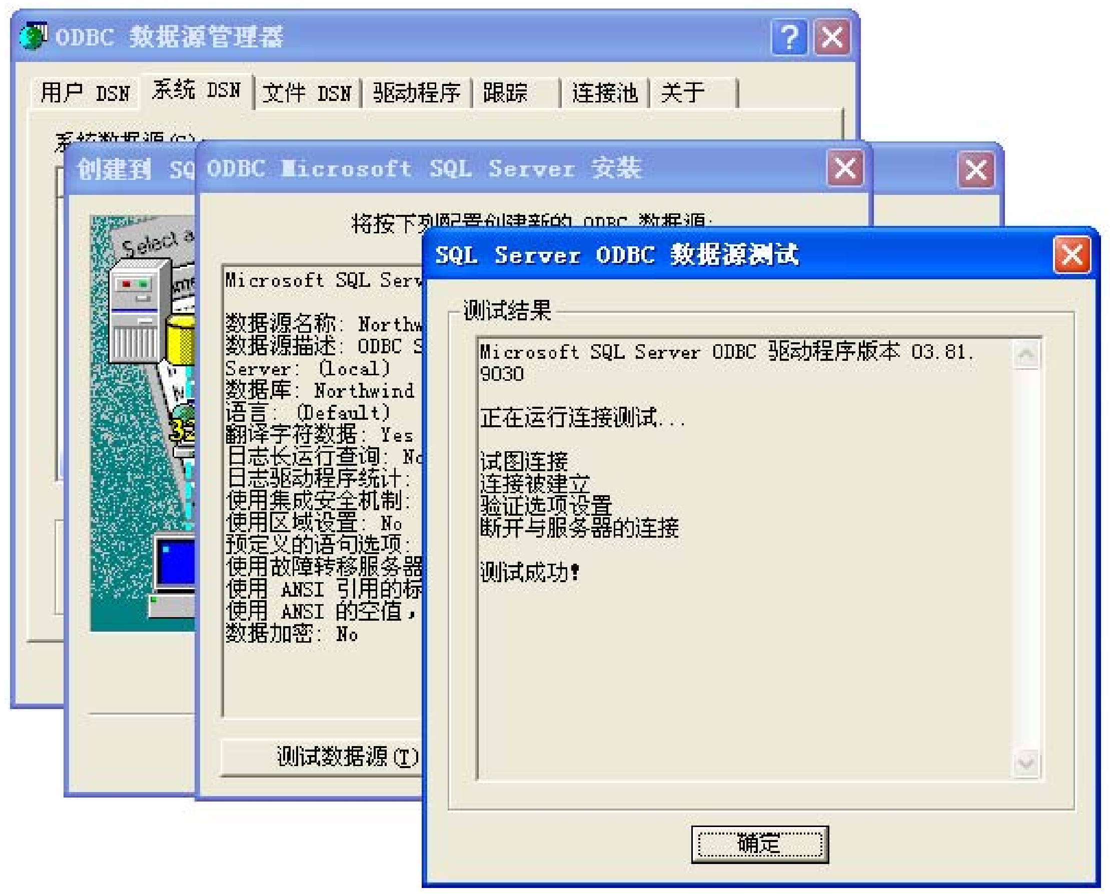
针对不同数据库选择 ODBC 驱动程序

在进行 Oracle 数据库迁移时，如果使用 Oracle 提供的 ODBC 驱动建立的数据源导入时出现异常，可以换用“Microsoft ODBC for Oracle”驱动建立 ODBC 数据源

选择源数据库



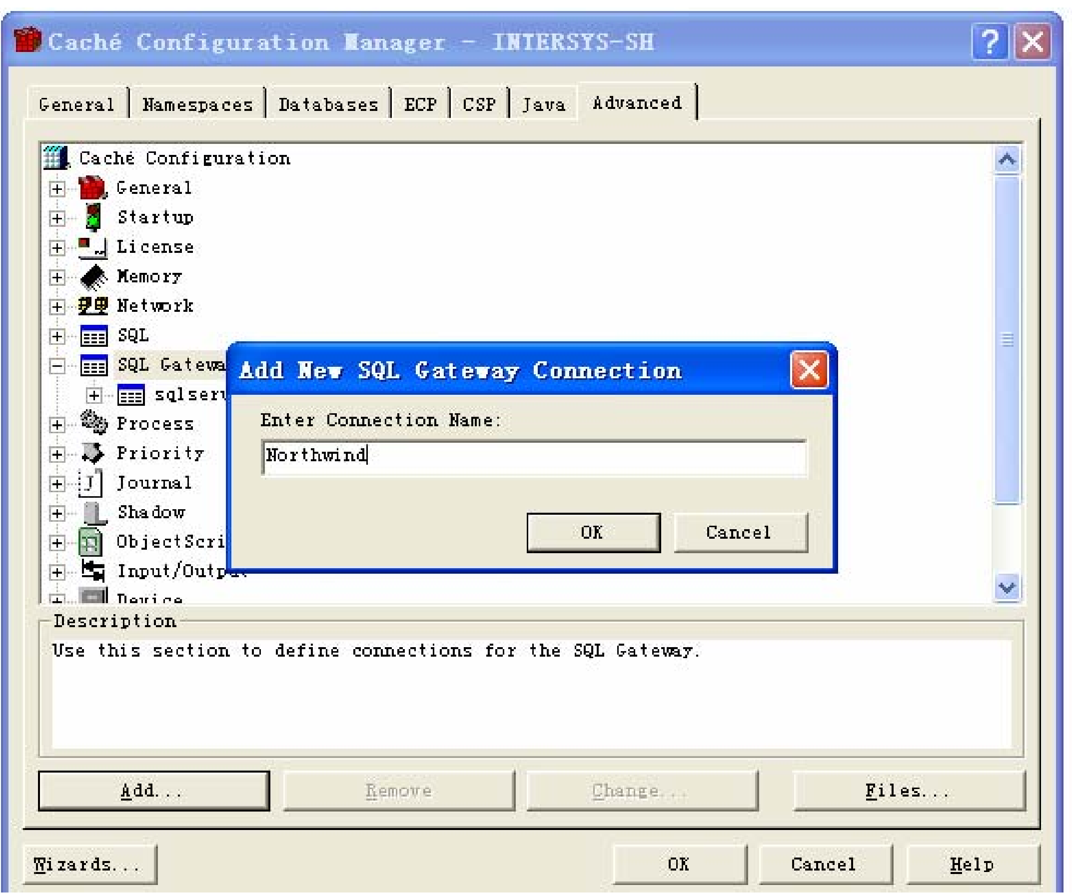
建立并测试 ODBC 数据源



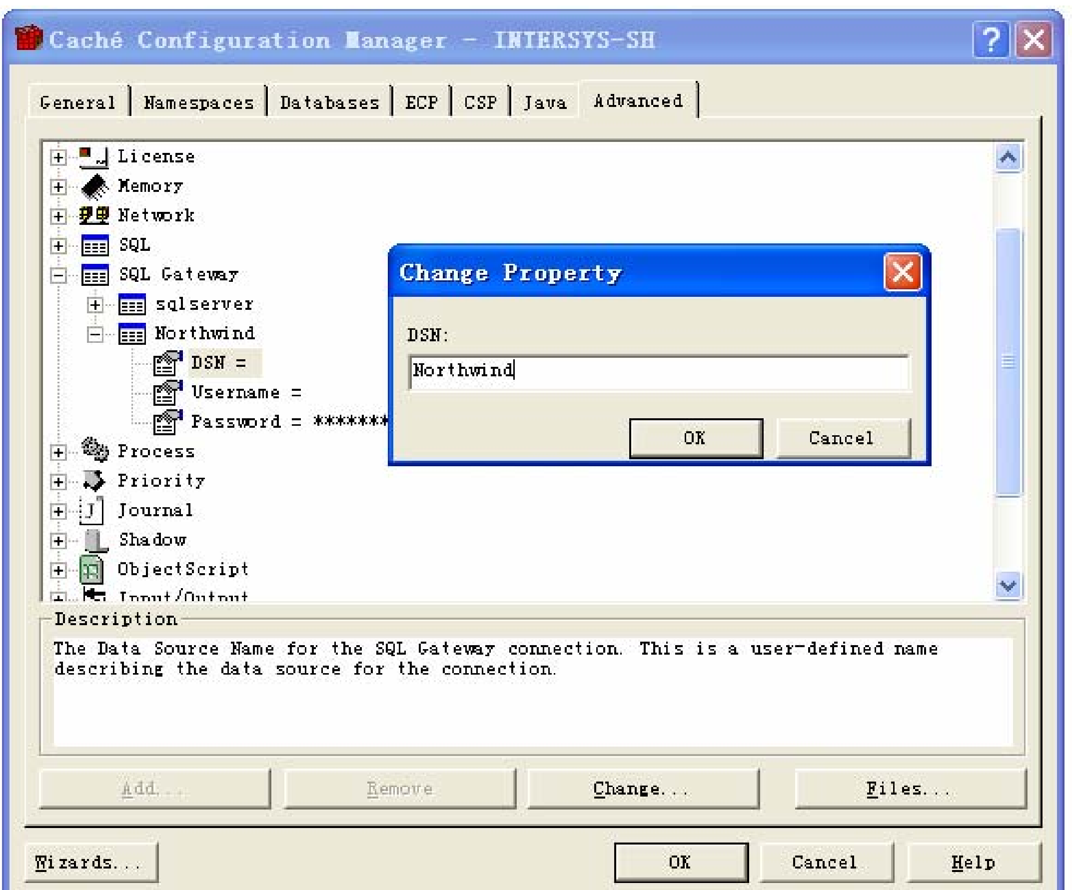
2.3 在 Caché 中建立 SQL Gateway

在 Caché Configuration Manager->Advanced->SQL Gateway 中新建对 ODBC 数据源的关系网关

新建 SQL Gateway 时，要填写新的 SQL 网关连接的名称：

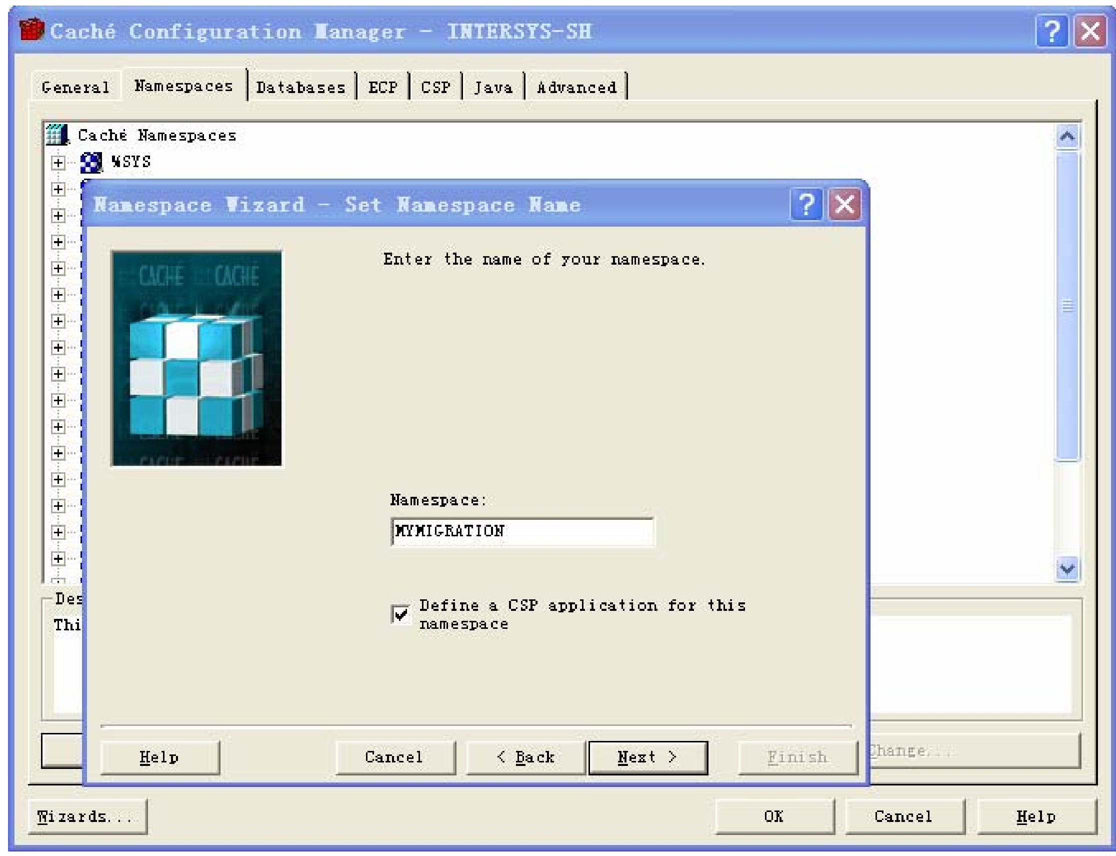


填写 ODBC 数据源名和数据源登陆用户名、密码

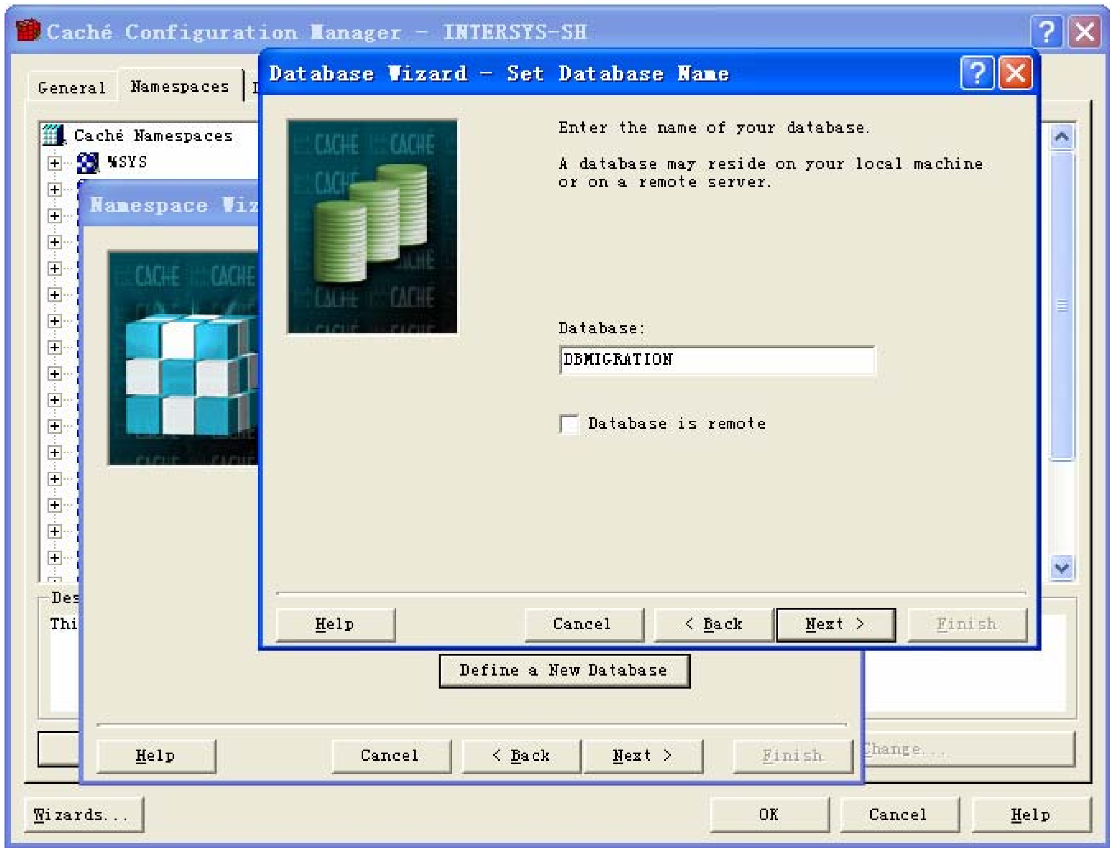


2.4　使用 Caché 数据迁移向导进行迁移

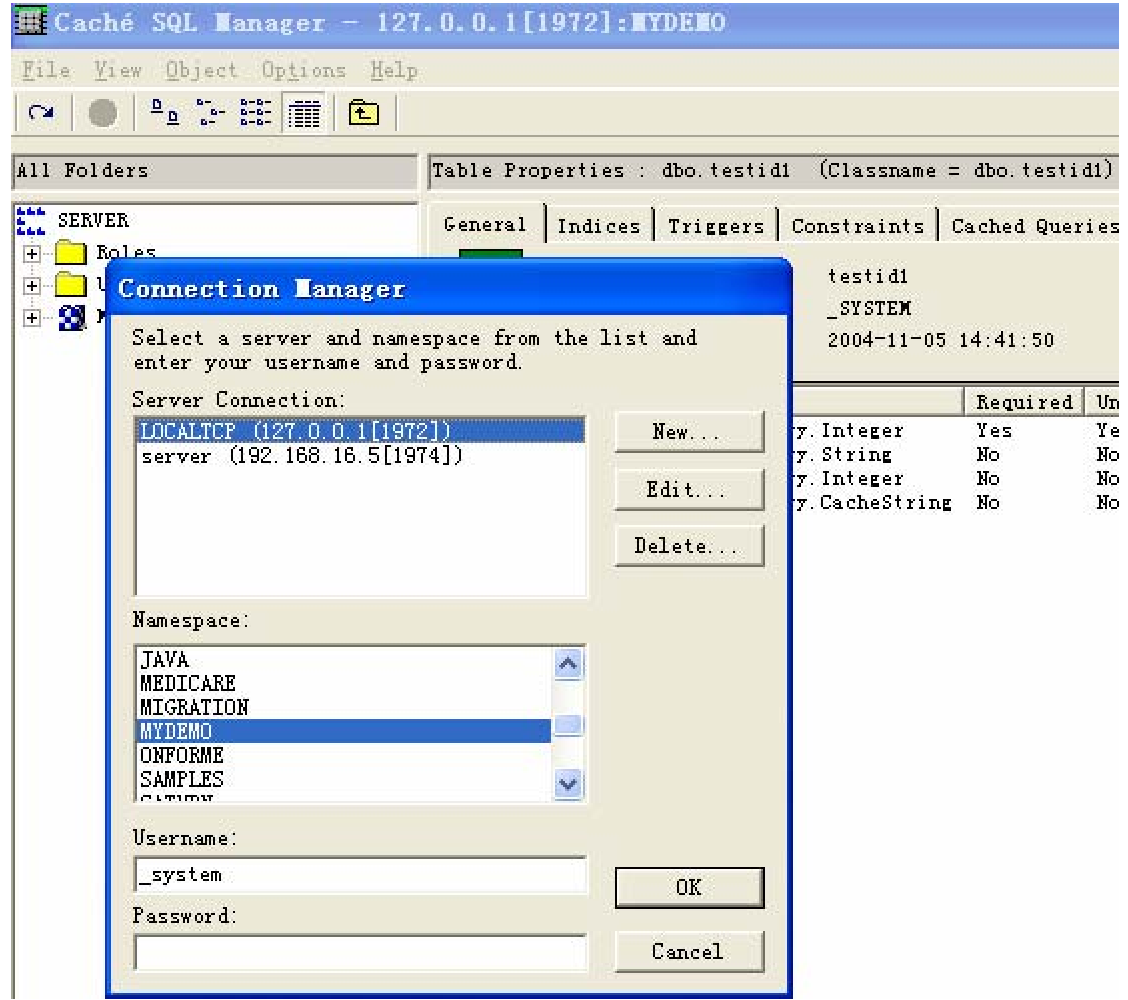
首先在 Configration Manager 中新建一个命名空间，作为目标命名空间；



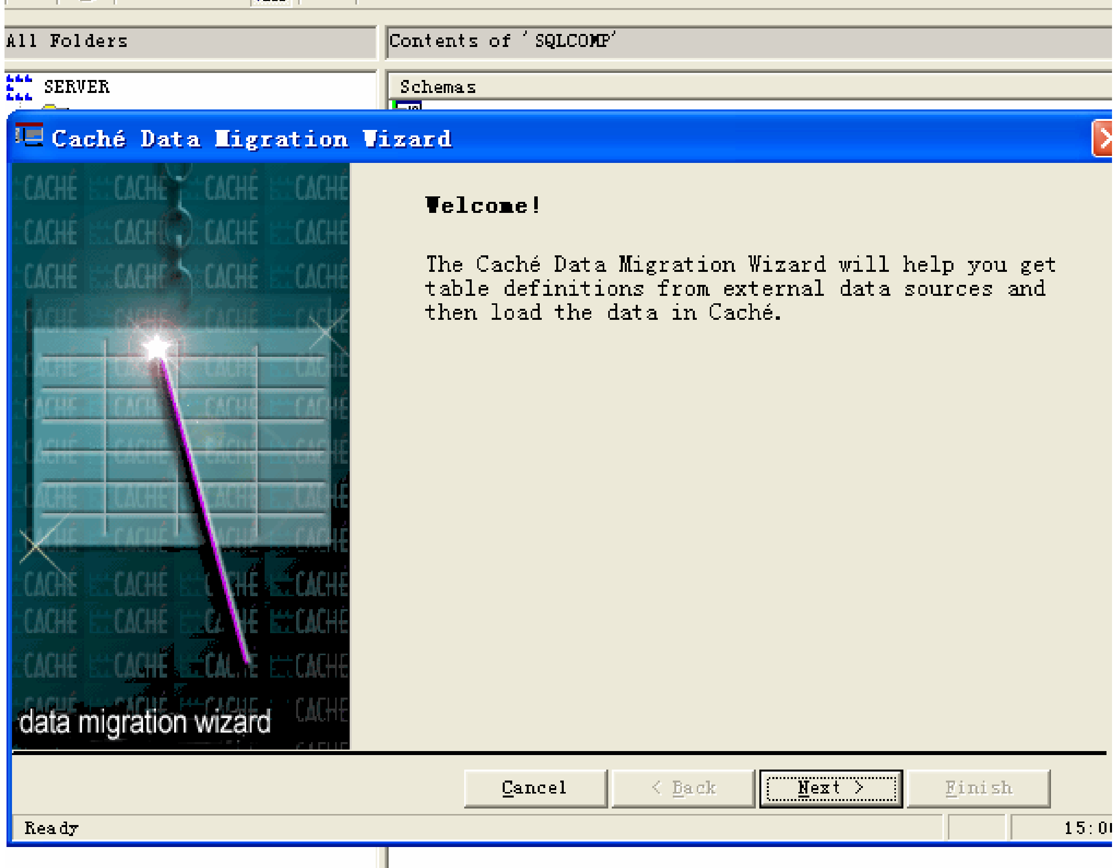
为目标命名空间选择一个在 Caché 已存在的数据库，如果在实际存在的各数据库中没有所想用的，可另外新建一个数据库用于数据迁移：



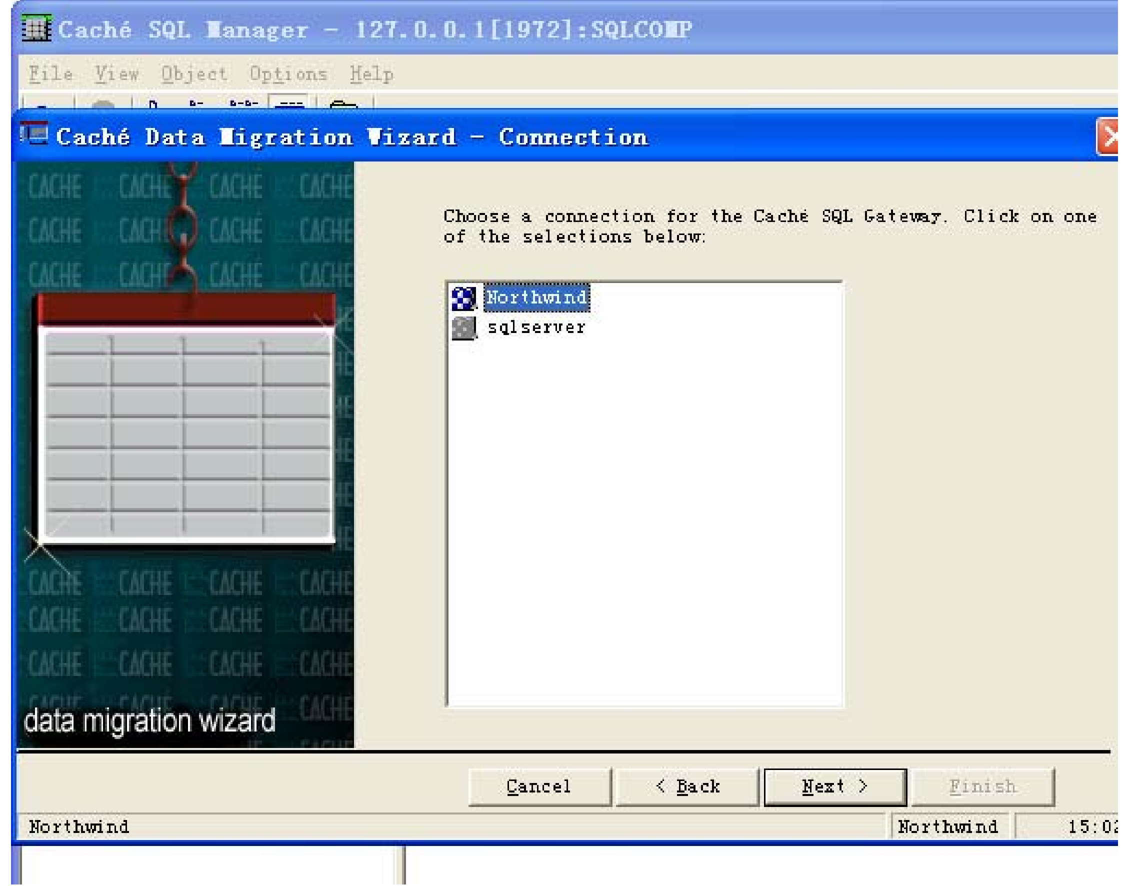
上述步骤完成并激活新命名空间后，打开 Caché SQL Manager->File->Change Namespace 选择要导入命名空间：



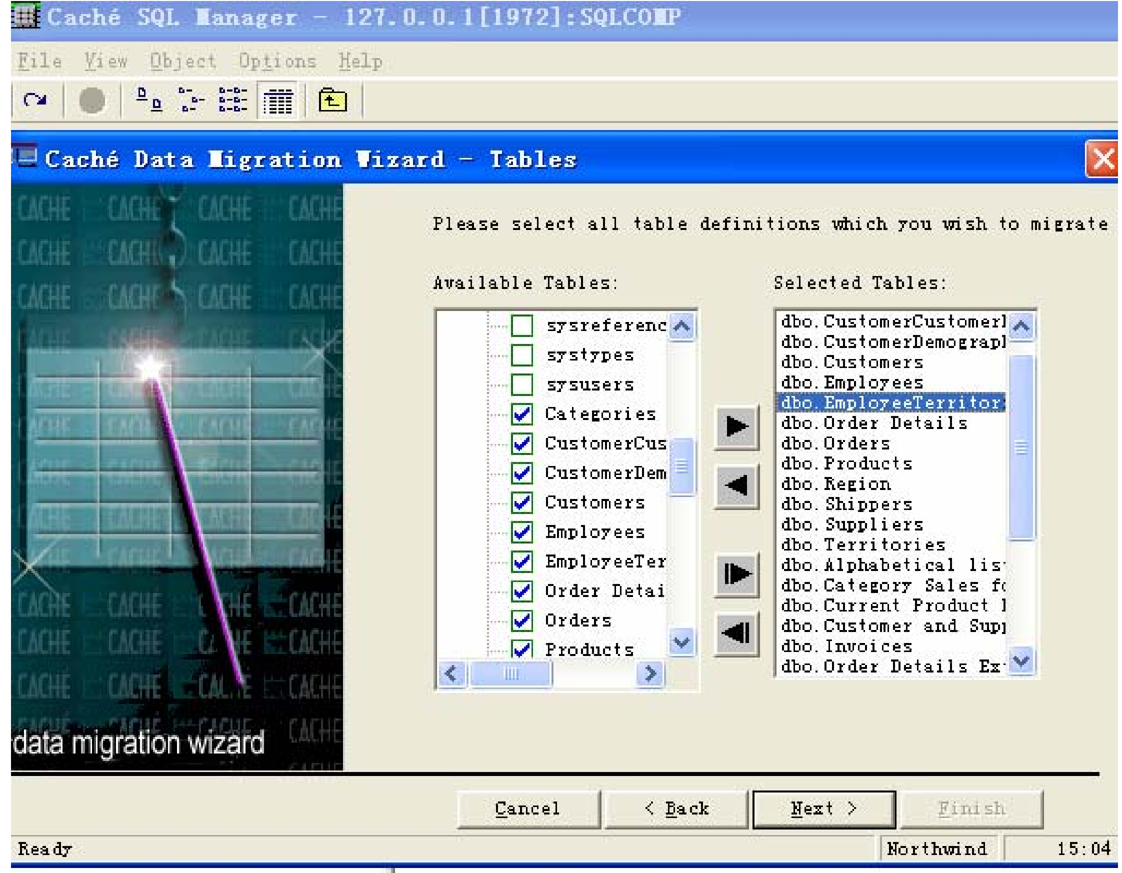
然后打开 Caché SQL Manager->File ->Data Migration，利用出现的 Caché Data MigrationWizard 即数据迁移向导开始迁移工作：



首先为 SQL Gateway 选择一个连接：



选择要导入的表：

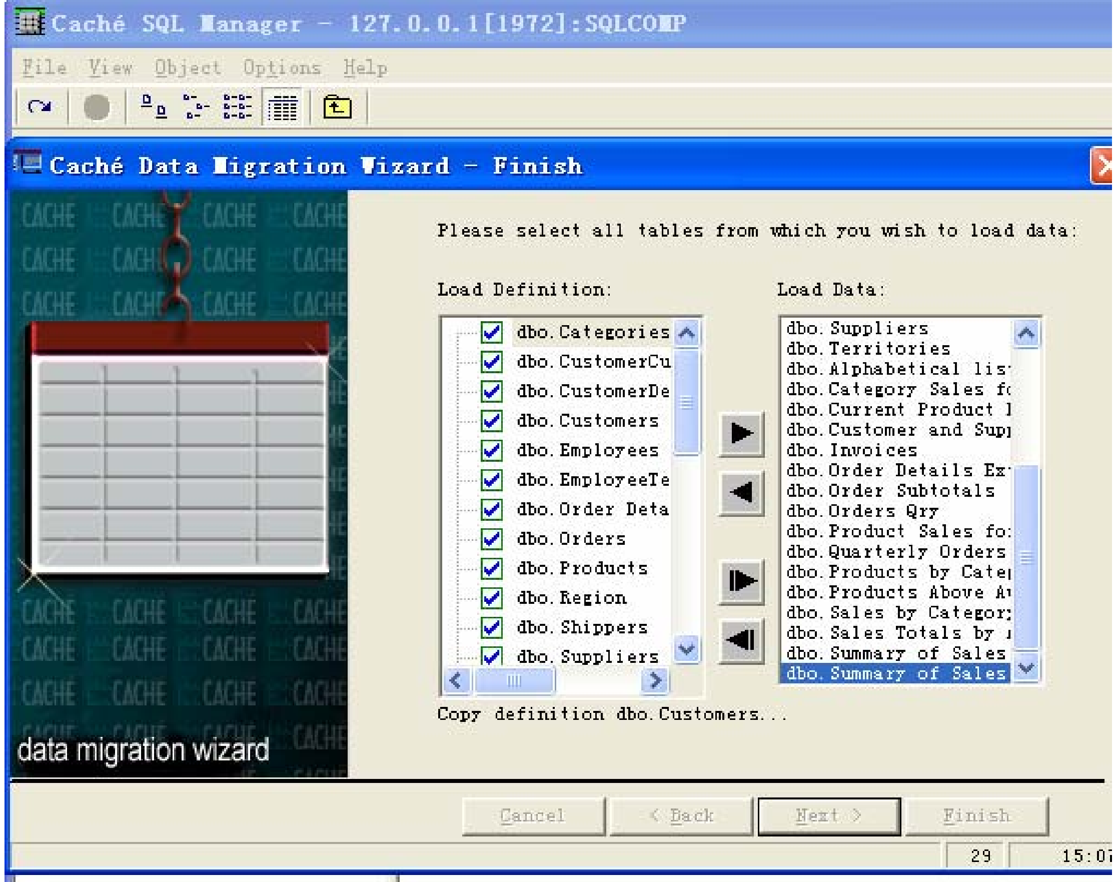


注意：

不要导入关系型数据库中的系统表

如果导入的目标机使用 FAT32 文件系统，注意 FAT32 上单个文件最大只能到 4Gb，导入大于 4Gb 的数据库可能会报错

如果导入的表有 Caché SQL 关键字，可将 Configuration Manager->Advanced->SQL->"Support Delimited Identifiers"设置为 Yes。Caché SQL 关键字见 reserved word.



备注：

除了用 Caché Data Migration 向导进行迁移外，也可以用其他工具，如 SQLServer 的 DTS 进行迁移，这种情况下，你需要先建立 Caché 数据库的 ODBC 数据源。

2.5 视图的迁移

将产生 View 的 DDL 在 Caché SQL Manager 中执行即可

2.6 存储过程和触发器的迁移

由于存储过程和触发器可能使用各个数据库非标准的专有语法，因此不能自动迁移。但 Caché 的 COS 非常强大，你可以很容易改写原有存储过程和触发器，下面是一个例子，COS 在 SQL 中的使用请参考第 3 节

```
CREATE PROCEDURE Demo(@NAME CHAR(20),@NUM INT=1) AS
  DECLARE @MAXNO INT
  IF @NUM < 1 SELECT @NUM=1
  BEGIN TRAN
    UPDATE dbo.SequenceTable SET SeriesNumber=SeriesNumber+@NUM WHERE Name=@NAME
    IF @@ROWCOUNT=0 AND @@ERROR = 0
    BEGIN
      INSERT INTO dbo.SequenceTable VALUES(@NAME,@NUM)
      IF @@ERROR <> 0 GOTO ERR
    END
    ELSE IF @@ERROR <> 0
      GOTO ERR
    SELECT @MAXNO=ISNULL(MAX(SeriesNumber),1) FROM dbo.SequenceTable WHERE Name=@NAME
    IF @@ERROR <> 0 GOTO ERR
  COMMIT TRAN
  select (@MAXNO-@NUM+1)
  RETURN(@MAXNO-@NUM+1)
ERR:
  ROLLBACK TRAN
  RAISERROR('fail to ceate SeriesNumber! ',1,2) WITH NOWAIT
 select 0
  RETURN(0)
```

SQLServer 存储过程

```
CREATE PROCEDURE Demo(IN sNAME VARCHAR(20), IN nNUM INT default 1)
 RETURNS INT
 LANGUAGE OBJECTSCRIPT
 {
 NEW SQLCODE,%ROWCOUNT
 TSTART
 IF nNUM < 1
 {
 set nNUM = 1
 }
 &sql( UPDATE dbo.SequenceTable SET SeriesNumber= SeriesNumber+ :nNUM WHERE Name=:sNAME)
 If (SQLCODE=100) && (%ROWCOUNT=0)
    {
    &sql( INSERT INTO dbo. SequenceTable VALUES(:sNAME,:nNUM))
    If SQLCODE < 0
      {
       TROLLBACK
       QUIT nNUM
      }
    }
    Else
   {
    If SQLCODE <0
     {
    TROLLBACK
    QUIT
     }
   }
 &sql( SELECT ISNULL(MAX(SeriesNumber),1) into :nMAXNO FROM dbo. SequenceTable WHERE Name=:sNAME)
 If SQLCODE < 0
  {
   TROLLBACK
   QUIT nNUM
  }
 TCOMMIT
 QUIT (nMAXNO-nNUM+1)
 }
```

改写后的 Caché 存储过程

3 Caché SQL

Caché SQL 支持 ANSI SQL 92 标准，在 Oracle、MS SQLServer 中的 SQL 语句符合 ANSI SQL 92 标准的，可在 Caché SQL 中执行。

下面简要介绍 Caché SQL：

3.1 基本 SQL 语法

3.1.1 表

可以使用 ANSI92 SQL 建立、修改表。

3.1.1.1 DDL

为了保证其他关系型数据库的表 DDL 能在 Caché 中执行，Caché 也支持下列选项，但不产生任何实际的作用。

```
{ON | IN} dbspace-name
LOCK MODE [ROW | PAGE]
[CLUSTERED | NONCLUSTERED]
WITH FILLFACTOR = literal
MATCH [FULL | PARTIAL]
```

3.1.1.2 字段数据类型

Caché SQL 支持绝大多数标准的 SQL 数据类型，可以在 ConfigurationManager->Advanced->SQL->”System DDL Datatype Mappings” 察看修改 SQL 数据类型和 Caché 数据类型的转换。如果有用户自定义的数据类型或特殊数据类型，可在 ConfigurationManager->Advanced->SQL->”User DDL Datatype Mappings”中添加。

3.1.1.3　RowID

在 SQL 中每行记录都有唯一标识符，即 RowID。Caché 默认会建立一个 RowID 列，一般名为“ID”。如果你希望用表中的某个字段作为 RowID，则要保证该字段是 PRIMARY KEY，并设置 ConfigurationManager->Advanced->SQL->”**Are Primary Keys Created through DDL also ID Keys**”为 Yes。例如下面的 DDL 将字段“Name”作为 RowID：

```
CREATE table dbo.testid(name varchar(100), age int, PRIMARY KEY (name))
```

通过 Caché SQL Manager 察看该表对应的 Global，会发现第一维度是 Name。其实在 Caché 中，可以自由定义底层多维数组的维度结构，这体现出 Caché 的多维数据库的巨大优势。

3.1.1.4　临时表

Caché 不支持临时表，但可以通过其它途径实现类似临时表的功能。例如，可以建立一张永久表，将$job 作为一个表字段，只存取本进程的行即可。

3.1.2 约束

Caché 支持各类约束，而且可以通过 ObjectScript 建立更为复杂的约束。

3.1.2.1　NULL 和 NOT NULL

Caché 中区分 NULL 和空字符串（""），空字符串不是 NULL。

3.1.2.2　UNIQUE

同其它关系型数据库

3.1.2.3　DEFAULT

Default 约束可以是字符串常量或数字常量，也可以是下列关键字：NULL, USER, CURRENT_USER, SESSION_USER, SYSTEM_USER, CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP.

Default 约束也可以使用 OBJECTSCRIPT 表达式，OBJECTSCRIPT 表达式写在单引号内作为 Default 约束值即可，例如：

```
CREATE TABLE mytest
(ID NUMBER(12,0) NOT NULL,
CREATE_DATE DATE DEFAULT OBJECTSCRIPT '+$HOROLOG' NOT NULL,
LOGNUM NUMBER(12,0) DEFAULT OBJECTSCRIPT '$INCREMENT(^LogNumber)')
```

3.1.2.4　Primary Key

同其它关系型数据库

3.1.2.5　Foreign Key

Caché 中外键定义可以使用参考动作子句，语法如下：

```
ON {DELETE|UPDATE} ref-action
```

其中 ref-action 可以是

NO ACTION：当外键父表的行被更新或删除时，外键子表检查是否有关联被更新/删除行的记录，如果有，则外键父表的行被更新或删除失败。这是默认选项。

SET NULL: 当外键父表的行被更新或删除时，外键子表检查是否有关联被更新/删除行的记录，如果有，则外键子表的关联字段被更新为 NULL。外键子表的关联字段要允许 NULL。

SET DEFAULT: 当外键父表的行被更新或删除时，外键子表检查是否有关联被更新/删除行的记录，如果有，则外键子表的关联字段被更新为默认值，如果没设默认值外键子表的关联字段被更新为 NULL。

CASCADE：当外键父表的行被更新或删除时，外键子表检查是否有关联被更新/删除行的记录，如果有，则外键子表的关联行跟随外键父表的行被更新或删除。

例如：

```
CREATE TABLE MyPatients (
    PatNum VARCHAR(16),
    Name VARCHAR(30),
    DOB DATE,
    Primary_Physician VARCHAR(16) DEFAULT 'A10001982321',
    CONSTRAINT Patient_PK PRIMARY KEY (PatNum),
    CONSTRAINT Patient_Physician_FK FOREIGN KEY
        Primary_Physician REFERENCES Physician (PatNum)
        ON UPDATE CASCADE
        ON DELETE SET NULL)
```

3.1.3 视图

3.1.3.1 可更新视图

可更新视图必须满足如下条件：

1. From 子句只能包含一个可被更新的表或视图
2. Select 子句必须包括表或视图的所有字段
3. 不能是 GROUP BY、 HAVING 或 SELECT DISTINCT 查询
4. 不是被映射为 View 的类查询（Query）
5. 对应得 View 类不能指定参数 READONLY=1

对于可更新视图，为了只更新可以更新的数据，可加入 WITH CHECK OPTION 进行限定：

```
CREATE VIEW GoodStudent AS
   SELECT Name, GPA
     FROM Student
```

```
    WHERE GPA > 3.0
  WITH CHECK OPTION
```

上面的视图只允许更新 GPA>3.0 的学生纪录，对此外的纪录执行 Insert 或 Update 将返回 SQLCODE -136（执行 Insert）、SQLCODE -137（执行 Update）。

3.1.3.2　只读视图

不满足可更新视图条件或被加上 WITH READ ONLY 子句的 View 是只读视图。

试图对只读视图进行更新将返回 SQLCODE -35。

3.1.4 索引

Caché 不仅支持传统索引，而且支持动态位图索引。因为 Caché 的 SQL 引擎会有效的用 AND、OR 进行索引合并，通常情况下，应为单个字段建立索引以减小所需索引的总的大小，除非你确实需要为一组字段建立索引。当字段有 10,000-20,000 以上的不同值时，使用常规索引；如果字段的值分布非常不均衡，即只有少数的可能值，这时可以使用位图索引。

建立位图索引的语法与常规索引类似：

```
CREATE BITMAP INDEX AgeIDX ON TABLE Person (Age)
```

3.1.5 游标

Caché 的游标语法如下：

```
&sql(DECLARE C1 CURSOR FOR
  SELECT %ID,Name
  INTO :id, :name
  FROM Sample.Person
```

```
  ORDER BY Name
)

&sql(OPEN C1)
&sql(FETCH C1)

While (SQLCODE = 0) {
   Write id, ":  ", name,!
  &sql(FETCH C1)
}

&sql(CLOSE C1)
```

3.1.6 存储过程

存储过程可以在类中以方法或查询形式实现，也可以用 DDL 编写。用 DDL 编写 Caché SQL 存储过程可以用 SQL 语法或 Caché ObjectScript 脚本语法编写代码段。

SQL 语法编写 SQL 存储过程：

关键字 LANGUAGE SQL 说明代码段为 SQL 语法（Create Procedure 默认代码段为 SQL 语法，所以关键字 LANGUAGE SQL 可省略），代码段要写在“BEGIN”、“END”之间，例

```
CREATE PROCEDURE UpdatePay
 (IN Salary FLOAT DEFAULT '0',
  IN Name VARCHAR(50),
  INOUT PayBracket VARCHAR(50) DEFAULT 'NULL')
BEGIN
  UPDATE Sample.Person SET Salary = :Salary
  WHERE Name=:Name ;
END
```

用 **Caché ObjectScript** 脚本语法编写 **SQL** 存储过程：

关键字 LANGUAGE OBJECTSCRIPT 说明代码段为 OBJECTSCRIPT 语法，代码段写在花括号中。用 ObjectScript 脚本可以编写更为复杂的存储过程，ObjectScript 脚本内可以使用嵌入 SQL，例

```
CREATE PROCEDURE Sample_Employee.GetTitle(
  INOUT pHandle %SQLProcContext,
  IN SSN VARCHAR(11),
  INOUT Title VARCHAR(50) )
   RETURNS VARCHAR(30)
   FOR Sample.Employee
   LANGUAGE OBJECTSCRIPT
   {
      NEW SQLCODE,%ROWCOUNT
      &sql(SELECT Title INTO :Title FROM Sample.Employee
          WHERE SSN = :SSN)
      IF $GET(pHandle)'=$$$NULLOREF {
        SET pHandle.SQLCode=SQLCODE
        SET pHandle.RowCount=%ROWCOUNT }
        QUIT
   }
```

Caché ObjectScript 功能强大，可以方便实现复杂逻辑，用 ObjectScript 改写要迁移的关系型数据库存储过程和触发器更加简单快捷。

3.1.7 触发器

用 DDL 编写触发器的语法为：

```
CREATE TRIGGER name {BEFORE|AFTER} event [ORDER integer] ON table
[REFERENCING alias] action
```

ORDER integer：指定有多个针对同一事件的触发器的触发顺序，默认为 0，随数字增大优先级降低；

REFERENCING alias：指定事件前后的值，如 REFERENCING OLD ROW AS oldalias NEW ROW AS newalias；

Action：为触发器代码段，由[FOR EACH ROW [WHEN (<trigger_condition>)]] [LANGUAGE SQL| LANGUAGE OBJECTSCRIPT]<trigger_body>组成。同存储过程一样可以用 SQL 语法或 OBJECTSCRIPT 语法实现。注意：如果用 OBJECTSCRIPT 语法则不能使用 WHEN (<trigger_condition>)。

例

```
CREATE TRIGGER Trigger_1 AFTER INSERT ON Table_1
  REFERENCING NEW ROW AS new_row
  BEGIN
    INSERT INTO Log_Table VALUES ('INSERT into Table_1');
    INSERT INTO New_Log_Table VALUES
      ('INSERT into Table_1', new_row.ID);
  END
```

3.1.8 用户、角色和权限

Caché SQL 可以用 ANSI92 标准 SQL 语句建立、修改、删除用户、角色和权限。

> Caché SQL 对 ODBC、JDBC、动态 SQL 进行权限检查，但对于嵌入式 SQL 不进行权限检查，因为 Caché 假设应用程序在使用嵌入式 SQL 前会进行权限检查。

SQL 对象权限

权限	描述
•插入行的权 .••REFEREN	

表或视图取数 的权限.••U	
RESHOL 的权限.••%	
RESHOLD 更新行 权限.••%T	
ESHOLD	从表或视图取数据的权限.
UPDATE	更新行的权限.
%THRESHOLD	限制用户所使用的运行资源.

SQL 对象定义权限

权限	描述
%CREATE_FUNCTION	定义函数的权限
%DROP_FUNCTION	删除函数的权限

%CREATE_METHOD	定义方法的权限
%DROP_METHOD	删除方法的权限
%CREATE_PROCEDURE	定义存储过程的权限
%DROP_PROCEDURE	删除存储过程的权限
%CREATE_QUERY	定义查询的权限
%DROP_QUERY	删除查询的权限
%CREATE_TABLE	定义表的权限
%ALTER_TABLE	修改表的权限
%DROP_TABLE	删除表的权限
%CREATE_VIEW	定义视图的权限
%ALTER_VIEW	修改视图的权限
%DROP_VIEW	删除视图的权限
%CREATE_TRIGGER	定义触发器的权限

%DROP_TRIGGER	删除触发器的权限

SQL 管理权限

权限	描述
%ALTER_USER	修改用户定义的权限.
%CREATE_ROLE	新建角色的权限.
%CREATE_USER	新建用户的权限.
%DROP_ANY_ROLE	删除角色的权限.
%DROP_USER	删除用户的权限.
%GRANT_ANY_PRIVILEGE	
%GRANT_ANY_ROLE	

赋予用户访问命名空间的权限：

```
GRANT ACCESS ON namespace-list TO grantee [WITH GRANT OPTION]
```

namespace-list 是以“，”分隔的命名空间列表

grantee 是以“，”分隔的用户（或角色）列表

> Caché “命名空间”是逻辑层数据库，可以是跨多个物理数据库。
>
> 将权限赋予现有所有用户时，以“*”代替 grantee。例“GRANT ACCESS ON namespace-list TO *”。

将权限赋予所有用户（包括以后要建立的用户）时，以“_PUBLIC”代替 grantee。例“GRANT ACCESS ON namespace-list TO _PUBLIC”。

赋予用户管理权限：

```
GRANT admin-privilege TO grantee [WITH ADMIN OPTION]
```

将角色赋予用户：

```
GRANT role TO grantee [WITH ADMIN OPTION]
```

赋予用户对象权限：

```
GRANT object-privilege ON table-list TO grantee [WITH GRANT OPTION]
```

可以使用“*”代表所有对象，如“grant select, delete on * to role ”赋予角色所有表（视图）的选取和删除权限。

限制用户使用的运行资源：

```
GRANT %THRESHOLD literal TO grantee [WITH ADMIN OPTION]
```

Literal 是限制运行时资源限制的数字常量

3.2 Caché SQL 变量

%ROWCOUNT：*被 SQL 语句影响的行数*

```
&sql(UPDATE MyApp.Employee
    Set Salary = (Salary * 1.1)
    WHERE Salary < 50000)

 Write "Employees: ", %ROWCOUNT,!
```

%ROWID：*新产生行的 ID 号*

```
&sql(INSERT INTO Sample.Person
   (Name,SSN)
   VALUES ('Swift,Jonathan','111-22-3333'))

 If (SQLCODE = 0) {
    Write "New Person inserted with ID: ", %ROWID,!
 }
```

3.3 BLOB、CLOB、Stream

Caché SQL 支持二进制大对象（BLOB）和字符型大对象（CLOB）的存储，BLOB 和 CLOB 可以存储多达 4Gb 的数据（这并不是受 Caché 的限制，而是因为受 ODBC、JDBC 的限制）。

3.3.1 BLOB 和 CLOB 字段的限制

BLOB 和 CLOB 字段不能建立索引

不能在 Where 子句中使用 BLOB 和 CLOB 字段

不能一次 Insert 或 Update 多行含有 BLOB 和 CLOB 字段的纪录，只能一行一行地完成

3.3.2 BLOB 和 CLOB 字段的创建和使用

3.3.2.1　创建 BLOB 和 CLOB 字段的 DDL

```
CREATE TABLE MyApp.Person (
   Name VARCHAR(50) not null,
   Notes LONGVARCHAR,
   Photo LONGVARBINARY
)
```

3.3.2.2　在 Caché 对象类的方法内使用 BLOB 和 CLOB

在 Caché 对象类的方法内，不能使用嵌入式 SQL 和动态 SQL 直接操作 BLOB 和 CLOB 字段，要使用 SQL 找到 BLOB 和 CLOB 的流标识符，然后创建一个**%AbstractStream** 对象来进行存取。例如：

```
/// Display the memos for all Persons with a given city
/// within an HTML table
ClassMethod DisplayMemo(city As %String = "") [language = basic]
{
   ' Define a query to find all the Stream Id values for memo
   rs = New %Library.ResultSet()
   rs.Prepare("SELECT Name,Memo FROM MyApp.Person WHERE Home_City = ?")
   rs.Execute(city)

   ' iterate over the results
   PrintLn "<TABLE>"
   While (rs.Next())
      PrintLn "<TR>"
      ' display the person's name
      PrintLn "<TD>" & rs.Data("Name") & "</TD>"

      ' Now open the stream object containing the memo
      stream = OpenId %Stream(rs.Data("Memo"))
      Print "<TD>"

      ' Write the contents of the stream to the current device
      stream.OutputToDevice()
      PrintLn "</TD></TR>"
   Wend
   PrintLn "</TABLE>"
}
```

3.3.2.3　通过 ODBC 使用 BLOB 和 CLOB

ODBC 驱动器（服务器）使用特殊的协议存取 BLOB 和 CLOB 字段，一般 ODBC 应用需要写特殊的代码以访问 BLOB 和 CLOB 字段。

3.3.2.4　通过 JDBC 使用 BLOB 和 CLOB

可使用 JDBC 标准的 BLOB 和 CLOB 接口进行访问

```
  Statement st = conn.createStatement();
ResultSet rs = st.executeQuery("SELECT MyCLOB,MyBLOB FROM MyTable");
rs.next();     // fetch the Blob/Clob

java.sql.Clob clob = rs.getClob(1);
java.sql.Blob blob = rs.getBlob(1);

// Length
System.out.println("Clob length = " + clob.length());
System.out.println("Blob length = " + blob.length());
```

3.4 SQL 错误信息

Caché SQL 的错误信息可以在 SQL Error Code 中查到

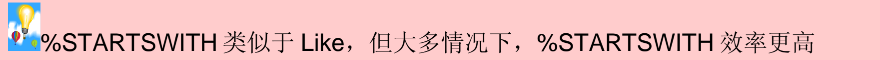
3.5 特殊 SQL 语法

%ID：%ID 是代表当前行 ID 的虚拟字段

```
SELECT %ID FROM Sample.Person ORDER BY %ID
```

%STARTSWITH： %STARTSWITH 操作符检查字符型量（字段）是否以特定字符（串）开始

```
SELECT %ID, Name FROM Sample.Person WHERE Name %STARTSWITH 'A'
```


%STARTSWITH 类似于 Like，但大多情况下，%STARTSWITH 效率更高

x__classname： x__classname 是 Caché 表中的隐含字段，存储当前行的类名

```
SELECT %ID,LastName,FirstName,x__classname
   FROM MyStart.Person
WHERE x__classname = '~Doctor~'
ORDER BY LastName
```

如果基类是%Persistent 类，则基类表中有各个子类的实例，由此隐含字段就可以区分当前行是哪个子类的实例

隐式连接： Caché SQL 提供特殊的“->”操作符可以从相关表直接取值，而不需要显式地写表连接。

例如你有如下两个类定义: **Company**:

```
Class Sample.Company Extends %Persistent [ClassType = persistent ]
{
/// The Company name
Property Name As %String;
}
```

和 **Employee**:

```
Class Sample.Employee Extends %Persistent [ClassType = persistent ]
{
/// The Employee name
Property Name As %String;
```

```
/// The Company this Employee works for
Property Company As Company;
}
```

有如下 SQL 可得到所有职员和其所在公司：

```
SELECT Sample.Employee.Name AS EmpName, Sample.Company.Name AS
CompName
FROM Sample.Employee LEFT OUTER JOIN Sample.Company
ON Sample.Employee.Company = Sample.Company.ID
```

用隐式连接可方便得改写为：

```
SELECT Name AS EmpName, Company->Name AS CompName
FROM Sample.Employee
```

用户定义函数：Caché SQL 允许在 SQL 中调用类方法

例如在类 MyApp.Person 中定义 Cube 类方法，且声明为 SQL 存储过程（SqlProc）：

```
Class MyApp.Person Extends %Persistent [ClassType = persistent, language = basic]
{
    /// Find the Cube of a number
    ClassMethod Cube(val As %Integer) As %Integer [SqlProc]
    {
    Return val * val * val
    }
}
```

则在 SQL 查询中，可以调用该方法就像它是内建的 SQL 方法:

```
SELECT %ID, Age, MyApp.Person_Cube(Age) FROM MyApp.Person
```

利用“用户定义函数”可以实现很多功能，例如类似于 SQLServer 中的 HOST_NAME()函数，可以在 Utility 类中建立 HostName 类方法如下:

```
ClassMethod HostName() As %String [ SqlProc ]
{
```

```
Set HostName = $System.Server.HostName()
Quit HostName
}
```

则可使用如下类似于 MS SQL 的 SQL 语句：

```
SELECT Name, Utility_HostName()FROM LogonHistory
```

4 ODBC&JDBC

在 Caché 安装时，会自动安装 ODBC 驱动，应用程序可以通过 ODBC 驱动访问 Caché ODBC 数据源。

4.1 ODBC 变量

Caché ODBC 环境变量包括：

CACHEODBCDEFTIMEOUT：登陆超时，单位为秒。

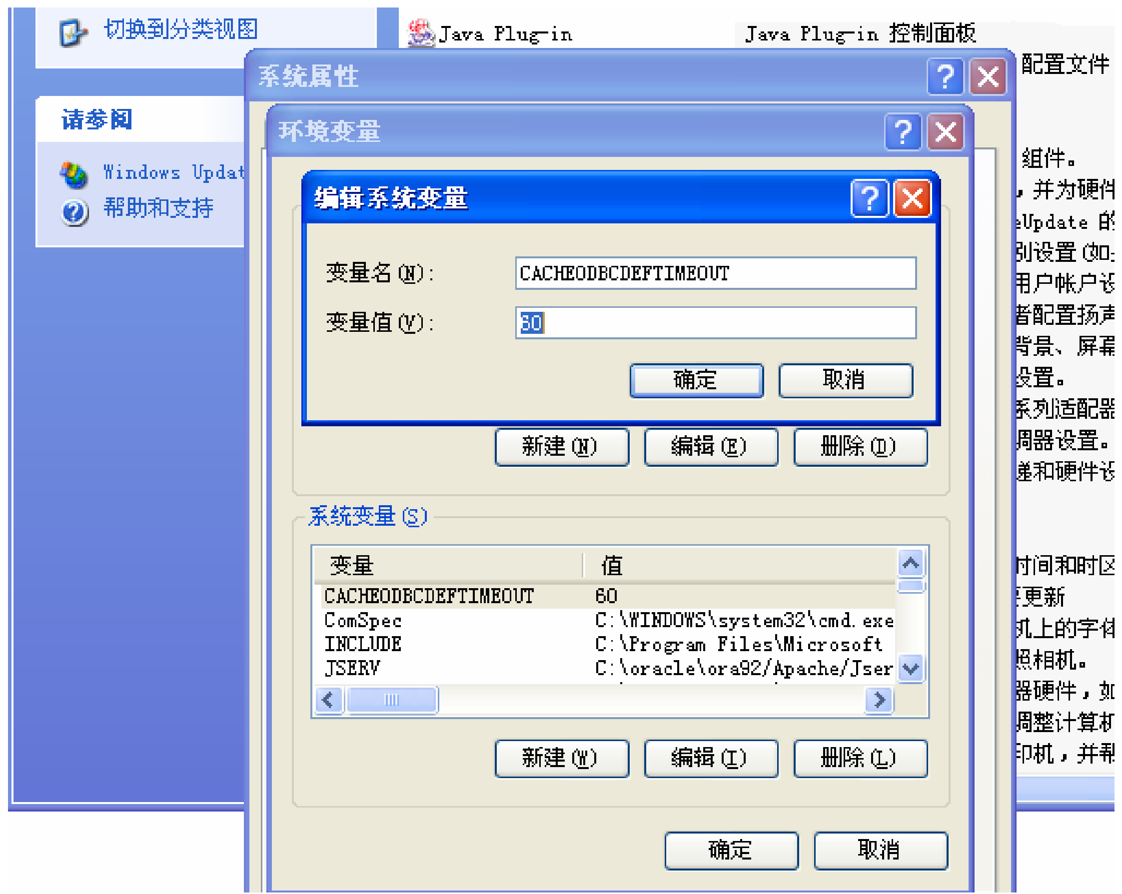
CACHEODBCHEADER：此布尔型量设置驱动是否写头信息到日志文件，头信息包括版本号、日期等信息，1 为写，默认为 0。

CACHEODBCPID：此布尔型量设置是否将连接进程号作为日志文件名扩展，例如进程号为 21933 的进程将产生名为 CacheODBC.log.21933 的日志文件名。1 为扩展，默认为 0。

CACHEODBCTRACE (UNIX only)：略

CACHEODBCTRACEFILE：设置 trace 文件的文件名和物理位置

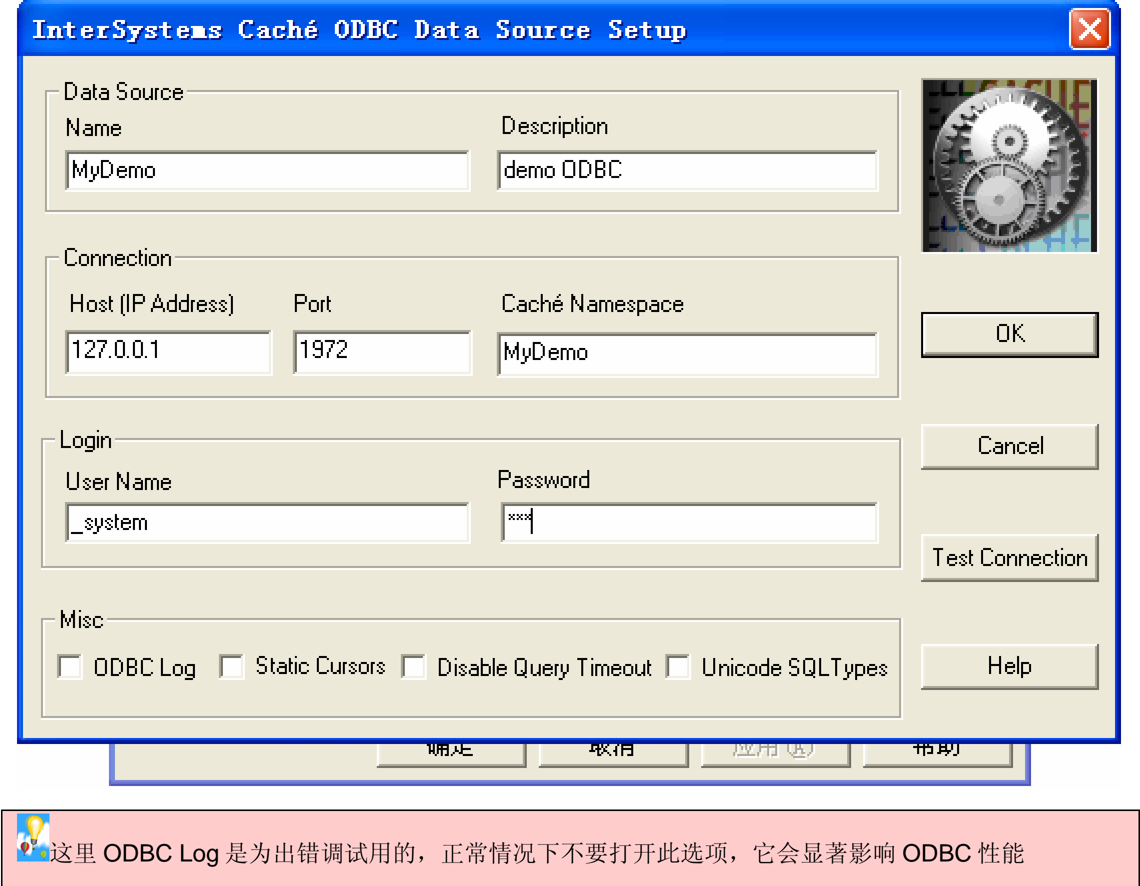
在 Windows 中，打开控制面板->系统->高级->环境变量 设置 Caché ODBC 环境变量：



4.2 Caché ODBC 数据源

建立 Caché ODBC 数据源

ODBC 数据源管理器
用户 DSN
系统 DSN
文件 DSN
驱动程序
跟踪
连接池
关于
系统数据源(S):
名称
驱动程序
添加(D)...
创建新数据源
选择您想为其安装数据源的驱动程序(S)。
名称
Driver da Microsoft para arquivos texto (*.
Driver do Microsoft Access (*.mdb)
Driver do Microsoft dBase (*.dbf)
Driver do Microsoft Excel(*.xls)
Driver do Microsoft Paradox (*.db)
Driver para o Microsoft Visual FoxPro
InterSystems ODBC
Microsoft Access Driver (*.mdb)
Microsoft Access-Treiber (*.mdb)
Microsoft dBase Driver (*.dbf)
< 上一步(B)
完成
取消



4.3 Caché JDBC 数据源

配置 JDBC 客户端需要以下两步：

安装 Caché JDBC 驱动（在*<cachesys>/dev/java/lib/CacheDB.jar 中会自动安装*）

设置你的 CLASSPATH 环境变量指向这个文件

具体配置详见《Java 配置》

5　SQL 性能优化

本节主要介绍软件方面的 SQL 性能优化

5.1　影响性能的系统参数

由于 Caché 会将数据和 Routine 缓存在 Database Cache 和 Routine Cache 中，因此影响性能的系统参数主要是 Database Cache 和 Routine Cache。这两个参数的大小对系统性能至关重要。建议尽量调大这两个参数。

Database Cache 和 Routine Cache 在 Caché Configuration Manager 的 General 中设置。

5.2　索引优化

要使 Caché 的性能发挥得更好，索引至关重要。在 Caché 中有 B-Tree 索引和 Bitmap（位图）索引。

确保所有外键字段都有索引。

所有在查询中使用=, <, >, LIKE, BETWEEN, %STARTSWITH 的字段都应该考虑建立索引；字段有超过 10,000 的不同值，使用常规索引，否则考虑位图索引。

如果字段的值分布非常不均衡，即只有少数的可能值，这时可以使用位图索引。

5.3　收集表统计信息（TuneTable）

在 Caché SQL Manager 中执行 TuneTable，以保证 SQL 引擎给出最优的执行计划。

Caché ObjectScript 参考资料

常用的命令：

需要做更详细的了解可以查阅在线帮助，即 documentation。

说明：[]内部的为可选的参数。

命令名称及格式	描述
Break[:cond]	在调试的时候中断一个 routine 的执行。
Break[:cond][status]	中断执行或者取消中断执行。
Close[:cond]device[:parameter]	关闭一个设备。
CONTINUE[:cond]	结束执行当前一轮的循环程序模块，并判断是否进行下一轮的循环。
Do[:cond] Do[:cond]{…}While expression	循环命令。
Do [:cond] entryref [(parameter)] [:cond]	调用一个函数，并把值传进去。
Else	当$Test 的值是 0 的时候，执行后面的命令行。
Else {…}	当$Test 的值是 0 的时候，执行后面的命令段。
ElseIf expression {…}	当$Test 的值是 0 而且后面的条件成立时，执行后面的程序段。
For For {…} For var=expr[,…] For var=expr[,…]{…} For var=start:increment:[end] For var=start:increment:[end] {…}	循环命令。
Goto [:cond] Goto [:cond] location[:cond]	跳转命令。

Halt[:cond]	停止当前 Caché 的工作，并退出 Terminal 窗口。
Hang[:cond]seconds	暂停当前的 routine 的执行一段时间。
If If expression If expression {…}	条件判断语句。
Job[:cond] routine[:cond] [(routineparms)] [:[process-parms][:timeout]]	执行一个新的 Caché 进程。
Kill[:cond] Kill[:cond] variable Kill[:cond] (variable)	删除所有变量或一些指定的变量或所有变量除了指定的变量。
Lock[:cond] Lock[:cond][+/-] variable[,…] [:timeout] Lock[:cond][+/-] (variable[,…]) [:timeout]	设置或者取消一些变量的锁定。
Merge [:cond] target=source	拷贝变量树。
New[:cond] New[:cond] variable[,…] New[:cond] (variable[,…])	初始化变量。
Open[:cond] device[:(parameter)] [:timeout]	开启一个设备以备使用。
Print [line1][:line2]	把指定的 routine 的一些行输出到当前的设备上。
Quit[:cond] Quit[:cond] expression	结束执行一个函数并返回一个值。
Read[:cond][f,][string,][f,] variable[:timeout] Read[:cond][f,][string,][f,] #variable[:timeout] Read[:cond][f,][string,][f,] variable #n[:timeout]	从当前的设备读信息。
Set[:cond] variable=expression Set[:cond]	给一个或者多个变量赋值。

(variablelist)=expression	
TCommit[:cond]	提交一个事务。
TROllback[:cond]	回滚一个事务。
TStart[:cond]	开始一个事务。
Use[:cond] device[:(parameter)] [:timeout][:"mnespace"]	把刚才启用的设备作为当前的设备，并可以设置一些属性。
View block	返回内存块的信息。
While expression {…}	循环函数。
Do {…} while expression	
Write[:cond][f,] expression	输出信息到当前设备上。
Write[:cond][f,] #expression	
Xecute[:cond]expression[:cond]	执行 Caché ObjectScript 代码。
ZBreak	设置一个断点。
ZInsert "code" [:location]	插入一行代码到正在编写的 routine 里面。
ZKill variable	删除变量但不删除他的子节点。
ZLoad[:cond]	读一个 routine 用作编写。
ZLoad routine	
ZNspace expression	更改当前的 namespace。
ZPrint	输出指定的 routine 的行到当前的设备。
ZPrint line1[:line2]	
ZQuit	移去所有的或者部分执行栈的执行层。
ZQuit [:cond] expression	
ZRemove	删除当前正在编写的 routine 的一些行。
ZRemove line1[:line2]	
ZSave	保存一个 routine
ZSave routine	
ZSYNC	保证所有的逻辑事务在物理上已经结束了。
ZTrap[:cond]	产生一条错误信息。
ZTrap[:cond] expression	
ZWrite	把所有的或者指定的本地变量或者全局变量都写到当前的设备上。
ZWrite variable	
ZZDUMP (expression)	把字符串作为 16 进制写到当前的设备上。

常用的函数：

需要做更详细的了解可以查阅在线帮助，即 documentation。

说明：[]内部的为可选的参数。命令中大写的部分表示该命令的缩写。

函数名及格式	描述
$Ascii(expression[,position])	返回一个字符的 ASCII 值。
$BIT(bitstring,position[,bitvalue])	产生一个位串（bit string）。
$BITCOUNT(bitstring[,bitvalue])	数一个位串的位数。
$BITFIND(bitstring,bitvalue[,startpos])	搜索一个位值在一个位串里的位置。
$BITLOGIC(bitstring_expression [,length])	对一个位串进行位运算。
$CASE(expression,result:value,…)	判断表达式的值，返回相应的值。
$Char(expression[,…])	以一个整数型的串里面的数字作为 ASCII 码值来产生一个字符串。
$Data(variable)	判断一个变量的引用是不是已经定义了。
$Extract(expression,[,from[,to]])	返回字符串的指定部分。
$Find(string,substring[,position])	返回子串的最后的位置
$FNumber(number,format[,decimal])	按照用户定义的格式返回表达式的值。
$Get(variable[,default])	返回一个变量的值。
$Increment(number[,num])	将变量增加 1 或者指定的值。
$INumber(number,format[,erropt])	按照指定格式检查一个数，并转化为内部表示法。
$ISOBJECT(variable)	判断该变量是否是合法的 OREF。
$Justify(expression,width[,desimal])	返回一个右对齐的值，并可以改变它的格式。
$Length(expression[,delimiter])	返回一个字符串的长度，或者是被分隔符的子串的个数。
$LIst(list,from[,to])	返回列表中指定范围的元素。
$ListBuild(element,…)	使用括号中的参数产生一个列表。

$ListData(list[,position])	检查一个列表的元素是不是有值。
$ListFind(list[,value[,start]])	从指定位置开始找列表中的某一个值。
$ListGet(list[,position[,default]])	返回一个存在的列表元素的值。
$ListLength(list)	返回列表的元素的数目。
$NAme(variable[,number])	返回一个规范格式的名字，且把它的元素控制在指定的数目内。
$Next(variable)	返回索引的下一个元素。
$NUMber(number[,format[,min [,max]]])	验证一个数字，并把它转化为 Caché 的规范格式。
$Order(variable[,direction])	按照指定的方向返回一个变量的上一个或者下一个索引。
$Piece(expression,delimiter [,from[,to]])	返回一个或者多个被分隔符分开的子串。
$QLength(reference)	返回一个建立了索引的变量的元素的数目。
$QSubscript(reference,number)	取出一个建立了索引的变量指定数目的的元素。
$Query(reference[,direction])	按照指定的方向返回一个有索引的变量的上一个或者下一个定义的引用。
$Random(range)	返回一个随机数。
$REverse(string)	逆序返回一个字符串。
$Select(expression:value,…)	返回第一个为真的表达式对应的值。
$SORTBEGIN(global)	对一个 global 开始一个指定模式的分类。
$SORTEND(global[,save])	结束对一个 global 的分类，并保存已更新的值。
$STack(context_level[,code_string])	返回在命令堆中正在执行的命令的信息。
$Text(location)	返回一个 routine 的指定行的源文件。

$TRanslate(string,replace[,by])	替换一个字符串中的字符，并返回替换后的结果。
$View(address[,mode,length])	返回内存地址里的内容。
$ZABS(n)	返回一个数字的绝对值。
$ZARCCOS(n)	反三角余弦函数
$ZARCSIN(n)	反三角正弦函数
$ZARCTAN(n)	反三角正切函数
$ZBITAND(bitstring1,bitstring2)	返回 2 个位串的与结果。
$ZBITCOUNT(bitstring)	返回位串中值为 1 的个数。
$ZBITFIND(bitstring,value[,position])	返回该值从指定位置开始第一次出现的位置。
$ZBITGET(bitstring,position)	返回位串中指定位置的值。
$ZBITLEN(bitstring)	返回位串的长度。
$ZBITNOT(bitstring)	返回位串非运算的结果。
$ZBITOR(bitstring1,bitstring2)	返回位串或运算的结果。
$ZBITSET(bitstring,position,value)	对位串的指定位置进行赋值。
$ZBITXOR(bitstring1,bitstring2)	返回位串异或运算的结果。
$ZBOOLEAN(arg1,arg2,bit-op)	返回 2 个位串按照指定的逻辑运算产生的结果。
$ZConVerT(string,mode)	返回一个按照指定模式表示的字符串。
$ZCOS(n)	三角余弦函数
$ZCOT(n)	三角余切函数
$ZCSC(n)	三角余割函数
$ZCrc(string)	返回循环冗余码校验的结果。
$ZCyc(<expr>)	与程序间的通信有关的函数。他产生的检查和可以检查数据的有效性的。产生方式是 XOR（异或）。
$ZDate(Hdate [,format [,monthlist [,yearopt [,start [,end [,mindate [,maxdate [,erropt]]]]]]]])	把$Horolog 格式的日期值按照指定格式显示。
$ZDateH(date [,format [,monthlist	$ZDate 的反函数。

[yearopt [start [end [,mindate [,maxdate [,erropt]]]]]]])	
$ZDateTime(Hdatetime,dformat [,zformat [,precision[,monthlist [,start[,end[,mindate[,maxdate [,erropt]]]]]]]])	把$Horolog 格式的日期时间值按照指定格式显示。
$ZDateTimeH(datetime,dformat [,zformat [,monthlist [,start [,end [,mindate[,maxdate [,erropt]]]]]]])	$ZDateTime 的反函数。
$ZEXP(n)	n 的指数函数
$ZF(-1,”commandline”[,args])	在操作系统的环境下执行命令行。
$ZHex(parm)	把一个 16 进制的数变为一个 10 进制的数，反之亦然。
$ZIncrement(variable[,num])	与 $Increment(variable[,num]) 是一样的。
$ZLASCII(string[,position])	计算从指定位置开始 4 位的数值。
$ZLCHAR(n)	把一个指定的数变位一个 4 位的字符串。
$ZLN(n)	自然对数函数。
$ZLOG(n)	十对数函数。
$ZNAME (str,n)	检查一个字符串是不是标准的名称格式。后面的 n 表示需要判断的格式的类型。
$ZNext(reference)	索引中的变量的下一个引用。
$ZOrder(reference)	索引中的变量的下一个引用。
$ZPOWER(num,n)	求幂函数。
$ZPrevious(reference)	索引中的变量的前一个引用。
$ZSEarch(target)	返回操作系统中文件或者文件集的名称和位置。
$ZSEC(n)	三角正割函数。
$ZSEEK(offset[,mode])	设定当前的设备的偏移量。
$ZSIN(n)	三角正弦函数。

$ZSort(variable[,direction])	按照指定的方向返回一个有索引的变量的上一个或者下一个定义的引用。
$ZSQR(n)	平方根函数。
$ZSTRIP(<expr>,action,remove,retain)	在一个给出的字符串中，删除或者保留一些字符。
$ZTAN(n)	三角正切函数。
$ZTime(Htime[,format[,precision [,erropt]]])	把$Horolog 格式的时间值按照指定格式显示。
$ZTimeH(time[,format[,erropt]])	$ZTime 的反函数。
$ZTSRT(tlist[,delimeter])	把一个 tlist 格式的列表转换为一个 plist 格式的列表。
$ZWAscii(string[,position])	从指定或者默认的位置开始，计算头 2 个字节的数值。
$ZWChar(n)	$ZWAscii 的反函数。