C语言（下）

王继纲 编

目　　录

第十五章 接 口

Windows2000 的推出是许多人盼望已久的事情它带来的多种新特性令人为之兴奋不已对于一名程序设计人员来说最关注的一些问题有在 Windows2000 操作系统中将组件对象模型 COM 与 Microsoft 事务服务器 MTS 合二为一命名为 COM+ 全新的应用程序编程接口 Application Programming Interface 特性等等那么这一切对于软件开发人员来说意味着什么呢如何能够在新的视窗系统下高效地编写可靠的桌面应用程序和分布式应用程序本章将向读者介绍有关的问题首先本章讲解了组件化程序设计的基本概念随后详细地论述了如何从组件编程的角度利用 C#定义和实现接口为我们设计组件级的应用程序接口是一种新的基于组件的编程概念如果读者有过一些 COM 方面的基础知识对阅读本章将有一定的帮助。

15.1 组件编程技术从软件业的发展历程来看程序设计方法经历了多次变革每当一种程序设计方法不能适应应用软件发展的需要时人们就会努力寻找一种新的方法来解决这种软件危机 组件化程序设计就是程序设计的一种新的变革它结合了对象技术和组件技术两种特性更为适合现代企业级应用程序的开发需要这一节我们将向读者简要地介绍组件和分布式应用程序设计的基础知识如果您希望了解更多组件化程序设计的知识请参考这方面论述的专著。

15.1.1 应用程序的体系结构一个应用程序的体系

结构是应用程序结构的一种概念性描述当前随着信息技术的飞速发展现代企业中大多采用了分布式计算机系统日益激烈的竞争要求应用程序尽量缩短开发周期并且具有高度的灵活性以适应变化多端的市场需要这一切都对分布式应用程序的开发人员在业务方面和技术方面提出了挑战传统的分布式应用程序大多是两层的客户机/服务器模式 Client/Server 客户机直接连接到服务器上在客户机上负责处理数据和执行客户端应用程序这种两层的应用程序体系结构存在着许多限制客户连接的开销服务器数据格式的限制可扩展性等尤其是当客户的数目未知或者客户数目可能非常庞大时两层的应用程序体系结构将无法处理这种情况为了提高分布式应用程序的灵活性和可重用性可以在两层的体系结构中再加入第三层这就是三层式应用程序体系结构表示层向用户提供数据展现用户接口商业层用以实施商业逻辑表示层使用商业层提供的服务数据访问层执行具体的数据访问服务包括检索和存储在三层式应用程序体系结构中商业层不固定地连接到任何客户也不关心数据的存储方式修改任何一层都不会对其它层产生不良影响每种服务都是独立的并且可用新的方式进行组合创建新的应用程序这种三层结构方便开发人员创建高伸缩性的应用程序注意三层式应用程序并不意味着三台独立的计算机三层体系结构是一个逻辑模型具体采用哪种物理模型依赖于提供服务的位置。

15.1.2 组件组件的概念和特点组件的英文名为 component 也称为元件实际上组件并不是一种新概念它在许多成熟的工程领域有着十分广泛的应用比如我们组装计算机自己并不一定要了解 CPU 主板光驱等配件的工

作原理而只需要知道如何将这些配件组装在一起软件行业的组件系统比其它许多行业发展的都要慢在计算机软件发展的早期一个应用系统往往是一个单独的应用程序随着人们对软硬件需求的不断增加应用更加复杂程序更加庞大系统开发的难度也越来越大从软件模型的角度考虑人们希望把庞大的应用程序分割成为多个模块每个模块完成独立的功能模块之间协同工作这样的模块我们称为组件这些组件可以进行单独开发单独编译单独测试把所有的组件组合在一起就得到了完整的系统许多人都认为未来的应用程序都将利用组件实现组件化的软件结构为我们带来了极大的好处但是为了能够通过组装现有的组件来创建应用程序系统我们必须解决几个技术上的关键问题.. 采用一个标准方式来规范组件的定位和使用这样将大大减少在人员培训上的开销提高了组件的通用性.. 提供与对象进行交互操作的标准方式组件和对象所处的具体位置不应该影响程序员的开发方式也不妨碍它们之间的交互操作即我们所说的位置透明性.. 要便于创建组件的版本对软件的升级应该具有灵活性组件的更新不会对现有的应用程序的运行造成不良影响.. 提供满足用户需要的安全性接口了解了组件的基本含义后我们还必须进一步理解接口 interface 的含义接口描述了组件对外提供的服务在组件和组件之间组件和客户之间都通过接口进行交互因此组件一旦发布它只能通过预先定义的接口来提供合理的一致的服务这种接口定义之间的稳定性使客户应用开发者能够构造出坚固的应用一个组件可以实现多个组件接口而一个特定的组件接口也可以被多个组件来实现组件接口必须是能够自我描述的这意味着组件接口应该不依赖于具体的实现将实现和接

口分离彻底消除了接口的使用者和接口的实现者之间的耦合关系增强了信息的封装程度同时这也要求组件接口必须使用一种与组件实现无关的语言目前组件接口的描述标准是 IDL 语言由于接口是组件之间的协议因此组件的接口一旦被发布组件生产者就应该尽可能地保持接口不变任何对接口语法或语义上的改变都有可能造成现有组件与客户之间的联系遭到破坏每个组件都是自主的有其独特的功能只能通过接口与外界通信当一个组件需要提供新的服务时可以通过增加新的接口来实现不会影响原接口已存在的客户而新的客户可以重新选择新的接口来获得服务。

15.1.3 组件化程序设计组件化程序设计方法继承并发展了面向对象的程序设计方法它把对象技术应用于系统设计对面向对象的程序设计的实现过程作了进一步的抽象我们可以把组件化程序设计方法用作构造系统的体系结构层次的方法并且可以使用面向对象的方法很方便地实现组件组件化程序设计强调真正的软件可重用性和高度的互操作性它侧重于组件的产生和装配这两方面一起构成了组件化程序设计的核心组件的产生过程不仅仅是应用系统的需求组件市场本身也推动了组件的发展促进了软件厂商的交流与合作组件的装配使得软件产品可以采用类似于搭积木的方法快速地建立起来不仅可以缩短软件产品的开发周期同时也提高了系统的稳定性和可靠性组件程序设计的方法有以下几个方面的特点.. 编程语言和开发环境的独立性.. 组件位置的透明性.. 组件的进程透明性.. 可扩充性.. 可重用性.. 具有强有力的基础设施.. 系统一级的公共服务 C#语言由于其许多优点十分适用于组件编程但这并不是说 C#是一门组

件编程语言也不是说 C#提供了组件编程的工具我们已经多次指出组件应该具有与编程语言无关的特性请读者记住这一点组件模型是一种规范不管采用何种程序语言设计组件都必须遵守这一规范比如组装计算机的例子只要各个厂商为我们提供的配件规格接口符合统一的标准这些配件组合起来就能协同工作组件编程也是一样我们只是说利用 C#语言进行组件编程将会给我们带来更大的方便。

15.2 接 口 定 义从技术上讲接口是一组包含了函数型方法的数据结构通过这组数据结构客户代码可以调用组件对象的功能。

15.2.1 声明接口声明实际上就是一种定义新的接口的类型声明声明的格式如下 attributes interface-modifiers interface identifier interfacebase interface-body ; 接口仅可使用下列修饰符.. new .. public .. protected .. internal .. private 在一个接口定义中同一修饰符不允许出现多次 new 修饰符只能出现在嵌套接口中表示覆盖了继承而来的同名成员 The public, protected, internal, and private 修饰符定义了对接口的访问权限在接口的声明体中可以定义接口的成员接口的成员可以是方法属性索引指示器和事件下面的例子定义了一个名为 IControl 的接口接口中包含一个成员方法 Paint interface IControl { void Paint(); } 。

15.2.2 接口的继承接口具有不变性但这并不意味着接口不再发展类似于类的继承性接口也可以继承和发展注意接口继承和类继承不同首先类继承不仅是说明继承而且也是实现继承而接口继承只是说明继承也就是说派生类可以继承基类的方法实现而派生的接口只继承了

父接口的成员方法说明而没有继承父接口的实现其次 C #中类继承只允许单继承但是接口继承允许多继承一个子接口可以有多个父接口接口可以从零或多个接口中继承从多个接口中继承时用后跟被继承的接口名字多个接口名之间用分割被继承的接口应该是可以访问得到的比如从 private 类型或 internal 类型的接口中继承就是不允许的接口不允许直接或间接地从自身继承和类的继承相似接口的继承也形成接口之间的层次结构请看下面的例子程序清单 15-1 using System; interface IControl { void Paint(); } interface ITextBox: IControl { void Set Text(string text); } interface IListBox: IControl { void SetItems(string[] items); }interface IComboBox: IText Box, IListBox {} 对一个接口的继承也就继承了接口的所有成员上面的例子中接口 ITextBox 和 IListBox 都从接口 IControl 中继承也就继承了接口 IControl 的 Paint 方法接口 IComboBox 从接口 ITextBox 和 IListBox 中继承因此它应该继承了接口 ITextBox 的 SetText 方法和 IListBox 的 SetItems 方法还有 IControl 的 Paint 方法。

15.3 接口的成员。

15.3.1 接口成员的定义接口可以包含一个和多个成员这些成员可以是方法属性索引指示器和事件但不能是常量域操作符构造函数或析构函数而且不能包含任何静态成员下面例子中接口 IExample 包含了索引指示器事件 E 方法 F 属性 P 这些成员 interface IExample { s tring this[int index] { get; set; } event EventHandler E; void F(int value); string P { get; set; } }public del egate void EventHandler(object sender, EventArgs e);

接口成员默认访问方式是 public 接口成员声明不能包含任何修饰符比如成员声明前不能加 abstract, public, protected, internal, private, virtual, override 或 static 修饰符接口的成员之间不能相互同名继承而来的成员不用再声明但接口可以定义与继承而来的成员同名的成员这时我们说接口成员覆盖了继承而来的成员这不会导致错误但编译器会给出一个警告关闭警告提示的方式是在成员声明前加上一个 new 关键字但如果没有覆盖父接口中的成员使用 new 关键字会导致编译器发出警告。

15.3.2 对接口成员的访问对接口方法的调用和采用索引指示器访问的规则与类中的情况也是相同的如果底层成员的命名与继承而来的高层成员一致那么底层成员将覆盖同名的高层成员但由于接口支持多继承在多继承中如果两个父接口含有同名的成员这就产生了二义性这也正是 C#中取消了类的多继承机制的原因之一这时需要进行显式的声明程序清单 15-2 using System; interface ISequence { int Count { get; set; } } interface IRing { void Count(int i); }interface IRingSequence: ISequence, IRing {} class C { void Test(IRingSequence rs) { //rs.Count(1); 错误, Count 有二义性//rs.Count = 1; 错误, Count 有二义性((ISequence)rs).Count = 1; // 正确 ((IRing)rs).Count(1); // 正确调用 IRing.Count } } 上面的例子中前两条语句 x.Count(1)和 x.Count = 1 会产生二义性从而导致编译时错误因此必须显式地给 X 指派父接口类型这种指派在运行时不会带来额外的开销再看下面的例子程序清单 15-3 using System; interface IInteger { void Add(int i); } interface IDouble { void Add(double d); }interface INumber: IInteger, IDouble

{} class C { void Test(INumber n) { //n.Add(1); 错误 n.Add(1.0); // 正确((IInteger)n).Add(1); // 正确((IDouble)n).Add(1); // 正确} } 调用 n.Add(1) 会导致二义性因为候选的重载方法的参数类型均适用但是调用 n.Add(1.0) 是允许的因为 1.0 是浮点数参数类型与方法 IInteger.Add 的参数类型不一致这时只有 IDouble.Add 才是适用的不过只要加入了显式的指派就决不会产生二义性接口的多重继承的问题也会带来成员访问上的问题 interface IBase { void F(int i); }interface ILeft: IBase { new void F(int i); } interface IRight: IBase { void G(); }interface IDerived: ILeft, IRight {} class A { void Test(IDerived d) { d.F(1); // 调用 ILeft.F ((IBase)d).F(1); // 调用 IBase.F ((ILeft)d).F(1); // 调用 ILeft.F ((IRight)d).F(1); // 调用 IBase.F } } 上例中方法 IBase.F 在派生的接口 ILeft 中被 Ileft 的成员方法 F 覆盖了所以对 d.F(1) 的调用实际上调用了虽然从 IBase IRight IDerived 这条继承路径上来看 ILeft.F 方法是没有被覆盖的我们只要记住这一点一旦成员被覆盖以后所有对其的访问都被覆盖以后的成员拦截了接口多继承中底层成员对高层成员的覆盖。

15.3.3 接口成员的全权名使用接口成员也可采用全权名 fully qualified name 接口的全权名称是这样构成的接口名加小圆点. 再跟成员名比如对于下面两个接口 IBase ILeft(覆盖) IRight(未覆盖) IDerivedinterface IControl { void Paint(); } interface ITextBox: IControl { void SetText(string text); } 其中 Paint 的全权名是 IControl.Paint SetText 的全权名是 ITextBox.SetText 当然全权名中的成员名称必须是在接口中已经声明过的比

如使用 ITextBox.Paint.就是不合理的如果接口是名字空间的成员全权名还必须包含名字空间的名称 namespace System { public interface ICloneable { object Clone(); } } 那么 Clone 方法的全权名是 System.ICloneable.Clone 。

15.4 接口的实现。

15.4.1 类对接口的实现前面我们已经说过接口定义不包括方法的实现部分接口可以通过类或结构来实现我们主要讲述通过类来实现接口用类来实现接口时接口的名称必须包含在类声明中的基类列表中下面的例子给出了由类来实现接口的例子其中 ISequence 为一个队列接口提供了向队列尾部添加对象的成员方法 Add IRing 为一个循环表接口提供了向环中插入对象的方法 Insert (object obj) 方法返回插入的位置类 RingSquence 实现了接口 ISequence 和接口 IRing 程序清单 15-4 using System;interface ISequence { object Add(); } interface IRing { int Insert(object obj); } class RingSequence: ISequence, IRing { public object Add() {…} public int Insert(object obj) {…} } 如果类实现了某个接口类也隐式地继承了该接口的所有父接口不管这些父接口有没有在类声明的基类表中列出 using System; interface IControl { void Paint(); } interface ITextBox: IControl { void SetText(string text); } class TextBox: ITextBox { public void Paint() {…} public void SetText(string text) {…} } 这里类 TextBox 不仅实现了接口 ITextBox 还实现了接口 ITextBox 的父接口 IControl 前面我们已经看到一个类可以实现多个接口再看下面的例子 using System; interface IControl { void Paint(); }interface

IDataBound { void Bind(Binder b); }public class Control: IControl {public void Paint() {…} }public class EditBox: Control, IControl, IDataBound { public void Paint() {…} public void Bind(Binder b) {…} } **上例中类** EditBox **从** Control **类继承并同时实现了** IControl and IDataBound **接口** EditBox **中的** Paint **方法来自** IControl **接口** Bind **方法来自** IDataBound **接口二者在** EditBox **类中都作为公有成员实现 当然在** C# **中我们也可以选择不作为公有成员实现接口如果每个成员都明显地指出了被实现的接口通过这种途径被实现的接口我们称之为显式接口成员** explicit interface member **用这种方式我们改写上面的例子** public class EditBox: IControl, IDataBound { void IControl.Paint() {…} void IDataBound.Bind(Binder b) {…} } **显式接口成员只能通过接口调用例如** class Test { static void Main() { EditBox editbox = new EditBox(); editbox.Paint(); // error: no such method IControl control = editbox; control.Paint(); // calls EditBox's Paint implementation } } **上述代码中对** editbox.Paint()**的调用是错误的因为** editbox **本身并没有提供这一方法** control.Paint()**是正确的调用方式注意接口本身不提供所定义的成员的实现它仅仅说明这些成员这些成员必须依靠实现接口的类或其它接口的支持。**

15.4.2 **显式接口成员执行体为了实现接口类可以声明显式接口成员执行体** Explicit interface member implementations **显式接口成员执行体可以是一个方法一个属性一个事件或者是一个索引指示器的声明声明与该成员对应的全权名应保持一致** using System; interface I

Cloneable { object Clone(); } interface IComparable { int CompareTo(object other); } class ListEntry: ICloneable, IComparable { object ICloneable.Clone() {…} int IComparable.CompareTo(object other) {…} } 上面的代码中 ICloneable.Clone 和 IComparable.CompareTo 就是显式接口成员执行体不能在方法调用属性访问以及索引指示器访问中通过全权名访问显式接口成员执行体事实上显式接口成员执行体只能通过接口的实例仅仅引用接口的成员名称来访问显式接口成员执行体不能使用任何访问限制符也不能加上 abstract, virtual, override 或 static 修饰符显式接口成员执行体和其他成员有着不同的访问方式因为不能在方法调用属性访问以及索引指示器访问中通过全权名访问显式接口成员执行体在某种意义上是私有的但它们又可以通过接口的实例访问也具有一定的公有性质使用显式接口成员执行体通常有两个目的.. 因为显式接口成员执行体不能通过类的实例进行访问这就可以从公有接口中把接口的实现部分单独分离开如果一个类只在内部使用该接口而类的使用者不会直接使用到该接口这种显式接口成员执行体就可以起到作用.. 显式接口成员执行体避免了接口成员之间因为同名而发生混淆如果一个类希望对名称和返回类型相同的接口成员采用不同的实现方式这就必须要使用到显式接口成员执行体如果没有显式接口成员执行体那么对于名称和返回类型不同的接口成员类也无法进行实现只有类在声明时把接口名写在了基类列表中而且类中声明的全权名类型和返回类型都与显式接口成员执行体完全一致时显式接口成员执行体才是有效的例如 class Shape: ICloneable { object ICloneable.Clone() {…} int ICo

mparable.CompareTo(object other) {…} } 这是一个无效的声明因为 Shape 声明时基类列表中没有出现接口 IComparable 下面的代码同样也有错误 class Shape: ICloneable { object ICloneable.Clone() {…} } class Ellipse: Shape { object ICloneable.Clone() {…} } 在 Ellipse 中声明 ICloneable.Clone 是错误的因为 Ellipse 即使隐式地实现了接口 ICloneable ICloneable 仍然没有显式地出现在 Ellipse 声明的基类列表中接口成员的全权名必须对应在接口中声明的成员如下面的例子中 Paint 的显式接口成员执行体必须写成 IControl.Paint using System; interface IControl { void Paint(); } interface ITextBox: IControl { void SetText(string text); } class TextBox: ITextBox { void IControl.Paint() {…} void ITextBox.SetText(string text) {…} } 。

15.4.3 接口映射类必须为在基类表中列出的所有接口的成员提供具体的实现在类中定位接口成员的实现称之为接口映射 interface mapping 映射数学上表示一一对应的函数关系接口映射的含义也是一样接口通过类来实现那么对于在接口中声明的每一个成员都应该对应着类的一个成员来为它提供具体的实现类的成员及其所映射的接口成员之间必须满足下列条件.. 如果 A 和 B 都是成员方法那么 A 和 B 的名称类型形参表包括参数个数和每一个参数的类型都应该是一致的.. 如果 A 和 B 都是属性那么 A 和 B 的名称类型应当一致而且 A 和 B 的访问器也是类似的但如果 A 不是显式接口成员执行体 A 允许增加自己的访问器.. 如果 A 和 B 都是时间那么 A 和 B 的名称类型应当一致.. 如果 A 和 B 都是索引指示器那么 A 和 B 的类型形参表包括参数个数和每

一个参数的类型应当一致而且 A 和 B 的访问器也是类似的但如果 A 不是显式接口成员执行体 A 允许增加自己的访问器那么对于一个接口成员怎样确定由哪一个类的成员来实现呢即一个接口成员映射的是哪一个类的成员在这里我们叙述一下接口映射的过程假设类 C 实现了一个接口 IInterface Member 是接口 IInterface 中的一个成员在定位由谁来实现接口成员 Member 即 Member 的映射过程是这样的 1 如果 C 中存在着一个显式接口成员执行体该执行体与接口 IInterface 及其成员 Member 相对应则由它来实现 Member 成员 2 如果条件 1 不满足且 C 中存在着一个非静态的公有成员该成员与接口成员 Member 相对应则由它来实现 Member 成员 3 如果上述条件仍不满足则在类 C 声明的基类列表中寻找一个 C 的基类 D 用 D 来代替 C 4 重复步骤 1 3 遍历 C 的所有直接基类和非直接基类直到找到一个满足条件的类的成员 5 如果仍然没有找到则报告错误下面是一个调用基类方法来实现接口成员的例子类 Class2 实现了接口 Interface1 类 Class2 的基类 Class1 的成员也参与了接口的映射也就是说类 Class2 在对接口 Interface1 进行实现时使用了类 Class1 提供的成员方法 F 来实现接口 Interface1 的成员方法 Finterface Interface1 { void F(); } class Class1 { public void F() {} public void G() {} }class Class2: Class1, Interface1 { new public void G() {} } 注意接口的成员包括它自己声明的成员而且包括该接口所有父接口声明的成员在接口映射时不仅要对接口声明体中显式声明的所有成员进行映射而且要对隐式地从父接口那里继承来的所有接口成员进行映射在进行接口映射时还要注意下面两点.. 在决

定由类中的哪个成员来实现接口成员时类中显式说明的接口成员比其它成员优先实现.. 使用 Private protected 和 static 修饰符的成员不能参与实现接口映射例如 interface ICloneable { object Clone(); } class C: ICloneable { object ICloneable.Clone() {…} public object Clone() {…} } 例子中成员 ICloneable.Clone 称为接口 ICloneable 的成员 Clone 的实现者因为它是显式说明的接口成员比其它成员有着更高的优先权如果一个类实现了两个或两个以上名字类型和参数类型都相同的接口那么类中的一个成员就可能实现所有这些接口成员 interface IControl { void Paint(); } interface IForm { void Paint(); } class Page: IControl, IForm { public void Paint() {…} } 这里接口 IControl 和 IForm 的方法 Paint 都映射到了类 Page 中的 Paint 方法当然也可以分别用显式的接口成员分别实现这两个方法 interface IControl { void Paint(); } interface IForm { void Paint(); } class Page: IControl, IForm { public void IControl.Paint() { //具体的接口实现代码}public void IForm.Paint() { //具体的接口实现代码} } 上面的两种写法都是正确的但是如果接口成员在继承中覆盖了父接口的成员那么对该接口成员的实现就可能必须映射到显式接口成员执行体看下面的例子 interface IBase { int P { get; } } interface IDerived: IBase { new int P(); } 接口 IDerived 从接口 IBase 中继承这时接口 IDerived 的成员方法覆盖了父接口的成员方法因为这时存在着同名的两个接口成员那么对这两个接口成员的实现如果不采用显式接口成员执行体编译器将无法分辨接口映射所以如果某个类要实现接口 IDerived 在类中必须至少声明一个显式接

口成员执行体采用下面这些写法都是合理的//一对两个接口成员都采用显式接口成员执行体来实现 class C: IDerived { int IBase.P get { //具体的接口实现代码}} int IDerived.P(){ //具体的接口实现代码} } //二对 Ibase 的接口成员采用显式接口成员执行体来实现 class C: IDerived { int IBase.P get { //具体的接口实现代码} } public int P(){ //具体的接口实现代码} }//三对 IDerived 的接口成员采用显式接口成员执行体来实现 class C: IDerived { public int P get { //具体的接口实现代码} }int IDerived.P(){//具体的接口实现代码} } 另一种情况是如果一个类实现了多个接口这些接口又拥有同一个父接口这个父接口只允许被实现一次 using System; interface IControl { void Paint(); }interface ITextBox: IControl { void SetText(string text); } interface IListBox: IControl { void SetItems(string[] items); } class ComboBox: IControl, ITextBox, IListBox { void IControl.Paint() {…} void ITextBox.SetText(string text) {…} void IListBox.SetItems(string[] items) {…} } 上面的例子中类 ComboBox 实现了三个接口 IControl, ITextBox 和 IListBox 如果认为 ComboBox 不仅实现了 IControl 接口而且在实现 ITextBox 和 IListBox 的同时又分别实现了它们的父接口 IControl 实际上对接口 ITextBox 和 IListBox 的实现分享了对接口 IControl 的实现 15.4.4 接口实现的继承机制一个类继承了它的基类提供的所有接口的实现如果不显式地重新实现接口派生类就无法改变从基类中继承来的接口映射 interface IControl { void Paint(); } class Control: IControl { public void Paint() {…} } class TextBox: Control { new public void Pain