互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

第 5 章

对象间的关系

第 4 章曾介绍过，对象之间可以因互相调用对方的方法而存在短暂的关系，如同陌生人在街头擦肩而过，互相问候一样。本书将这种关系称为行为关系(behavior relationship)，因为它来自与对象 Y 相关的对象 X 做出的行为或动作。通过这种行为关系，对象 X 可以临时将对象 Y 的引用作为参数传入方法调用，也可以临时从对象 Z 请求 Y 的引用。然而，这里强调的是“临时”：当对象 X 完成与对象 Y 的通信后，对象 X 通常会丢弃 Y 的引用。

您需要和一些人保持更为重要和持久的关系(家人、朋友、同事等)，同样，对象之间也有建立更持久关系的需要。本书将这种关系称为结构关系(structural relationship)，因为要保持这样的关系，对象需要在其字段中维护其相关对象的持久引用，第 3 章已经介绍过这种技术。

本章主要介绍如下主题：

- 类之间或对象之间存在的各种结构关系，以及描述它们的方法
- 如何借助于继承(inheritance)这种强大的机制，仅仅通过描述与现存类的不同之处而派生新的类
- 通过继承派生类时要遵循的规则

5.1 关联和链接

类之间结构关系的正式名称是关联(association)。对于学生选课系统(SRS)来说，可能存在以下关联：

- Student 选修 Course
- Professor 讲授 Course
- DegreeProgram 要求选修 Course

关联是指类间的关系，而术语“链接(link)”是指两个特定对象(实例)间的结构关系。给定“学生选修课程”这个关联，则可有以下链接：

- Jackson Palmer(特定 Student 对象)选修 Math 101(特定 Course 对象)
- Helmut Schmidt(特定 Student 对象)选修 Basketweaving 972(特定 Course 对象)
- Mary Smith(特定 Student 对象)选修 Basketweaving 972(特定 Course 对象，Helmut

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

Schmidt 链接的同一对象)

给定 Student 对象 X 和 Course 对象 Y，它们之间就有可能存在“选修”链接关系，因为它们所属类之间就定义了这种关联关系。换句话说，关联使链接成为可能。

多数时候，我们会定义两个不同类之间的关联，这种关联称为二元关联(binary association)。例如，“选修”关系就是二元关联，因为它和两个不同类相关：Student 和 Course。而一元关联(unary association)或反身关联(reflexive association)，是同一个类的两个实例间的关系，例如：

- 一门课程是另一门课的先修课程
- 一位教授是其他教授的导师

即使反身关联两端的类完全一样，相应对象通常也是不同的实例：

- Math 101(Course 对象)是 Math 202(不同的 Course 对象)的先修课程。
- Gupta 教授(Professor 对象)是 Jones 教授和 Green 教授(其他 Professor 对象)的导师

等等。虽然这类情况较罕见，但的确也存在同一对象处于反身关联两端的情况。

多元关联可能存在，不过较少见。三元关联(ternary association)涉及 3 个类，例如，学生(Student)会选修特定教授(Professor)讲授的课程(Course)，如图 5-1 所示。

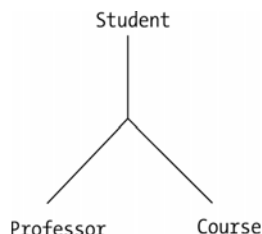


图 5-1　三元关联

然而，当描述关联时，通常会将多元关联分解为数个合适的二元关联。例如，可以把上例中的三元关联表示为以下 3 个二元关联(如图 5-2 所示)：

- 学生选修课程
- 教授讲授课程
- 教授指导学生

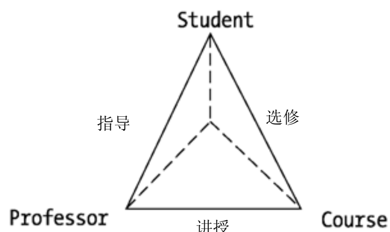


图 5-2　使用 3 个二元关系的等价表示

在给定关联中，每个参与类都有一个角色(role)。在“指导”关联(教授指导学生)中，教授的角色就是“指导者”，而学生的角色是“被指导者”。我们仅在有助于说明

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

模型时才会给关联的角色命名。在“选修”关联(学生选修课程)中，可能没有必要为两端的 Student 和 Course 类命名，因为这不能让关联所属的抽象模型更加清晰。

5.1.1 多样性

对于类 A 和 B 之间的关联类型 X，术语“多样性(multiplicity)”是指和类型 B 的实例相关联的类型 A 的对象数量。例如，一个学生可以选修多门课程，但只能有一位教授成为其导师。

多样性有 3 种类别：一对一、一对多、多对多，下面将分别介绍。

1. 一对一(1:1)

类 A 的一个实例只与类 B 的一个实例相关，不多不少，反之亦然，例如：

- 一个学生只有一份成绩单，一份成绩单只属于一个学生。
- 一个教授只负责管理一个院系，一个院系也只有一位教授作为负责人。

还可以通过指定关联两端是“可选”或“必须”来作出更多的约束。例如，可将上例的关联修改如下：

- 一位教授“可选地”负责管理一个院系，但一个院系“必须”有一位教授作为负责人。

这个版本的关联更切合现实世界的情况，因为大学中每个院系都会有一个负责人，但并非每位教授都负责管理院系。然而，毋庸置疑，如果一位教授是系主任，则他只负责管理这个院系。

2. 一对多(1:m)

给定一个类 A 的实例，可能会有多个类 B 的实例与之关联。但从类型 B 的对象角度来看，它只和一个类 A 的实例相关。例如：

- 一个院系聘用多位教授(多样性中的“多”)，但一位教授(通常)只为一个院系工作(多样性中的“一”)。
- 一位教授指导多个学生，但一个学生只有一位教授作为其导师。

注意，此处的“多”可理解为“0 或更多(可选)”或者“1 或更多(必须)”。为了更具体地进行说明，将上面的示例重新定义：

- 一个院系聘用 1 位或多位(“多”，必须)教授，但是一位教授(通常)只为一个院系工作。
- 一位教授指导 0 个或多个(“多”，可选)学生，但是一个学生只有一位教授成为其导师。

另外，和一对一关系一样，在一对多关联的“一”端，也可以指定是必须的或是可选的。例如，如果为“学生不必非有指导教授”这样的大学制度建模时，可能需要按如下方式修改上面的关联：

- 一位教授指导多个(0 个或多个，可选)学生，但一个学生最多(可选择)只有一位教授成为其导师。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

3. 多对多(m:m)

对于一个给定的类 A 的实例，可以有多个类 B 的实例与之关联，反之亦然。例如：

- 一个学生可选修多门课程，一门课程可有多个学生选修。
- 一门课程可有多门先修课程，一门课程也可能是其他多门课程的先修课程(这是多对多反身关联的示例)。

和一对多关联一样，这里的“多”可解释为“0 或更多”(可选)或“1 或更多”(必须)。例如：

- 一个学生可选修 0 门或多门(“多”，可选)课程，一门课程有 1 个或多个(“多”，必须)学生选修。

当然，对特定关联的验证——所涉及的类、多样性、关联各方是可选还是必须——完全取决于建模的真实世界的情况。如果您正在对每个院系可拥有多名系主任或每个学生可有多名导师的大学进行建模，那么选择的多样性可能就和上面的实例有所区别。

4. 多样性和链接

注意，多样性的概念只适用于关联，不适用于链接。链接只存在于对象间两两对应的情况(或者，比较罕见的情况是，对象自己和自己对应)。因此，多样性从本质上定义了从给定对象可以产生的某种关联类型链接的数量。这可用一个示例来说明。

再次考虑多对多关联：

“一个学生选修 0 门或多门课程，一门课程有 1 个或多个学生选修。”

一个特定的 Student 对象 X 可以有 0 个、1 个或多个到 Course 对象的链接，但这些链接中的每一个都仅在两个对象间存在——Student X 和单个 Course 对象。例如，在图 5-3 中：

- Student X 有 1 个链接(到 Course A)
- Student Y 有 4 个链接(到 Course A、B、C 和 D)
- Student Z 没有到任何 Course 的链接(Z 本学期缺席！)

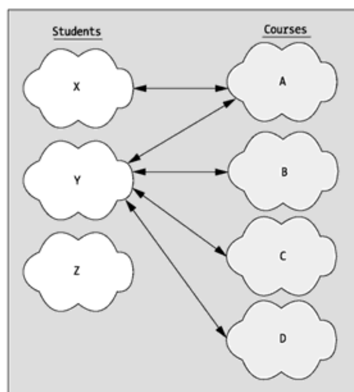


图 5-3 类之间的多对多关联；链接仅在对象间存在

反过来说，一个特定的 Course 对象 A 必须有到 Student 对象的 1 个或多个链接，这

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

样才能满足“必须”特性和关联的多样性。但是，这些链接中任意一个都只存在于两个对象之间：Course A 和单个 Student 对象。在图 5-3 中，例如：

- Course A 有 2 个链接(到 Student X 和 Y)
- Course B、C 和 D 各自有 1 个链接(到相同的 Student Y)

然而请注意，链接只存在于两个对象之间：一个 Student 对象和一个 Course 对象。本例的场景的确与 Student 和 Course 之间的多对多“选修”关联相关，但它只是类对象之间许多可能场景中的一个。

为了更清楚地说明这一点，现在将介绍另一个一对一的关联示例。

“一位教授(可选地)负责管理一个院系，但一个院系只能由一位教授担任系主任”，如图 5-4 所示：

- Professor 对象 1 和 4 分别有一个到 Department 对象 A 和 B 的链接。
- Professor 对象 2 和 3 没有这样的链接。

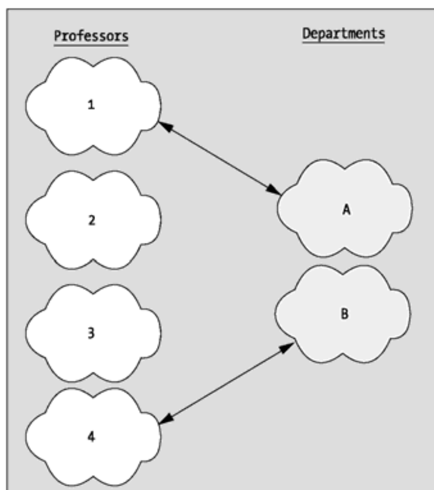


图 5-4 类之间的一对一关联；链接始终存在于两个对象之间

此外，从 Department 对象的角度来看，每个 Department 对象都只有一个到 Professor 对象的链接。因此，本例说明了 Professor 和 Department 之间的一对一“负责管理”的关系，也说明了在此链接中 Professor 类的“可选”特性。同样，这也是类之间许多可能的场景之一。

5.1.2 聚集

聚集(Aggregation)是一种特殊形式的关联，它说明了一种“包含”、“由……组成”或“有……”的关系。和关联一样，聚集用于描述类 A 和 B 之间的关系。不同的是，聚集表示的不仅是关系：某对象属于类 A(称为聚集类)，它由属于 B 的 1 个或多个对象组成。

例如，一辆汽车由引擎、变速器和 4 个轮子组成；如果 Car、Engine、Transmision 和 Wheel 都是类，则可得到以下聚集：

- 一辆汽车包括一个引擎

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

- 一辆汽车包括一个变速器
- 一辆汽车有多个(这里是 4 个)轮子

或者，以 SRS 为例，可以这样说：

- 一所大学由多个学院(工程学院、法学院)组成
- 一个学院由多个院系组成

但不能说一个院系由多位教授组成，应该说一个院系聘用多位教授。

注意，这些聚集语句和关联非常相似，只不过关联名称是“由……组成”或“包含……”。这是因为聚集本身就是关联！那么何必要区分聚集和关联呢？我们真的需要知道有所谓的聚集么？事实上，聚集和关联之间存在微妙的差别，这的确会影响到抽象在代码中的表示。因此，此处将不继续讨论下去，第 14 章还会继续讨论这些细微差别。

现在暂时简单地表示：当您在类 A 和 B 之间发现一种关系，而且倾向于用“包括”、“由……组成”等短语来命名这种关联时，则这种关联极有可能是一种聚集关系。

5.1.3 继承

假定我们已经正确而完整地通过 Student 类对所有“学生”的基本特征进行了建模，并且已经用 C#编写了这个类。Student 类的简单版本如下：

```
using System;

public class Student
{
  private string name;
  private string studentId;
  // 等等。

  public string Name {
    get {
      return name;
    }
    set {
      name = value;
    }
  }

  public string StudentId {
    get {
      return studentId;
    }
    set {
      studentId = value;
    }
  }

  // 等等。
}
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

事实上，进一步假设 Student 类的代码经过严格测试，没有任何缺陷，并且已经应用于一些应用程序中，如学生选课系统、学生收费系统和校友录。

此时出现了新需求：对作为特殊学生类型的研究生进行建模。研究生和本书已经建模的普通学生之间的区别在于：

- 在进入研究生学习阶段前，该生已获得本科学位
- 在哪个院系获得这个本科学位

所有用于描述研究生所需的其他特征——name、studentId 等字段以及访问这些字段的属性——与已经编写的 Student 类没有区别，因为研究生归根到底也是学生。

如何实现这个 GraduateStudent 类的新需求呢？如果您不精通面向对象概念，则可能会尝试以下方法：

- 修改 Student 类以实现双重用途
- “克隆”Student 类
- 使用继承(Inheritance)来扩展现有的 Student 类

下面将深入介绍这 3 种方法：

1. 方法 1：修改 Student 类以实现双重用途

可以在 Student 的定义中加入反映本科学位信息的字段，还有访问这些字段的属性，对于那些还没毕业的学生，就让这些字段为空：

```
public class Student
{
   private string name;
   private string studentId;
   private string undergraduateDegree;
   private string undergraduateInstitution;
   // 等等。
```

然后，为了跟踪这些字段是否包含 Student 对象的值，则可能还会添加一个 bool 字段来判断这个学生是否是研究生：

```
public class Student
{
  private string name;
  private string studentId;
  private string undergraduateDegree;
  private string undergraduateInstitution;
  private bool isGraduateStudent;
  // 等等。
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

在为该类编写的后续新方法中，都要考虑这个 bool 字段的值：

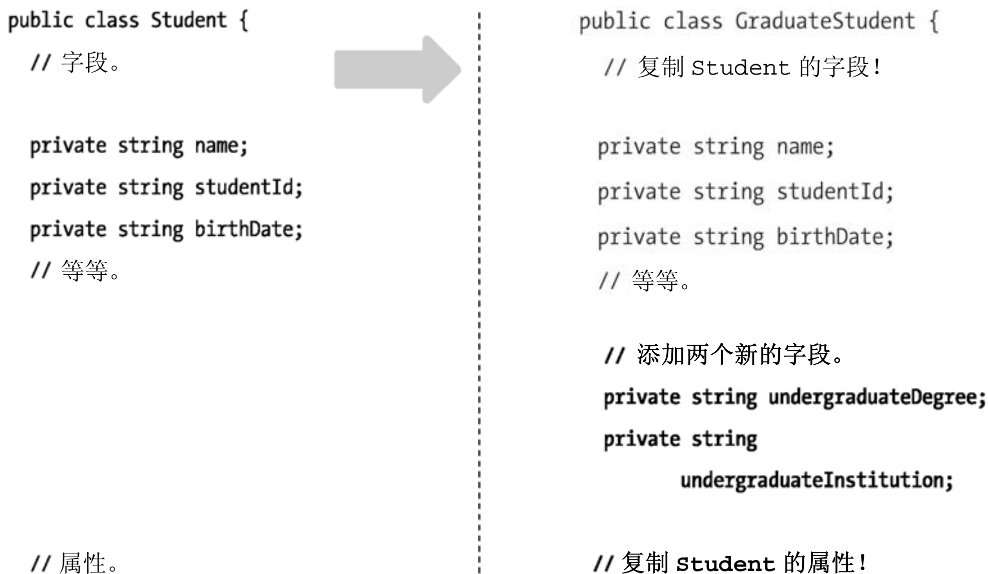
```
public void DisplayAllFields() {
   Console.WriteLine(name);
   Console.WriteLine(studentId);

   // 如果某个特定学生不是研究生，则不会定义 undergraduateDegree 和
   // undergraduateInstitution 字段，只在学生是研究生时才会输出这两个字段的值。
   if (isGraduateStudent) {
      Console.WriteLine(undergraduateDegree);
      Console.WriteLine(undergraduateInstitution);
   }
   // 等等。
}
```

这种方法会导致代码复杂，难以调试和维护。

2. 方法 2：克隆 Student 类

我们还可以创建一个新的 GraduateStudent 类，即通过(a)复制 Student 类，(b)将类的副本命名为 GraduateStudent 类，以及(c)向类的副本添加研究生所需的额外特征。



互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
public string Name {
  get {
    return name;
  }
  set {
    name = value;
  }
}

// 等等。
```

```
public string Name {
  get {
    return name;
  }
  set {
    name = value;
  }
}

// 等等

// 添加两个新字段的属性，
// 细节从略……
```

这样做效率非常低，因为两个类的代码大部分相同，而且假如稍后需要修改某个方法或字段的定义——例如，把 birthDate 字段的类型从 string 改为 DateTime，并且修改该字段对应的属性——则必须在两个类中做出相同的修改。

严格来讲，上述两种方法都可用，但产生的冗余代码却让应用程序难以维护。另外，如果要创建多种“特殊类型”的学生，这两种方法就不适用了。例如，如果要在第 3 种学生类中使用第 1 种方法的 DisplayAllFields 方法，那会变得相当复杂；例如，继续教育的学生，他们没有学位，只是为提升专业素养而听课。

- 最有可能需要添加另一个 bool 标志，以跟踪学生是否攻读学位：

```
public class Student
   {
   private string name;
   private string studentId;
   private string undergraduateDegree;
   private string undergraduateInstitution;
   private string degreeSought;
   private bool isGraduateStudent;
   private bool seekingDegree;
   // 等等。

   // 在 DisplayAllFields 方法中还必须考虑这个 bool 字段：
   public void DisplayAllFields() {
     Console.WriteLine(name);
     Console.WriteLine(studentId);

     if (isGraduateStudent) {
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
        Console.WriteLine(undergraduateDegree);
        Console.WriteLine(undergraduateInstitution);
    }

    // 如果学生不攻读学位，那么 degreeSought 的字段值未定义，我们应该只为攻读学位的
    // 学生输出这个值。
    if (seekingDegree) {
      Console.WriteLine(degreeSought);
    } else {
      Console.WriteLine("NONE");
    }

    // 等等。
}
```

这让情况变得更复杂！

如果要在这个方法逻辑里处理多种学生的情况，则会让代码愈发复杂。假设要实现几十种不同的学生类型，代码会变得乱七八糟！遗憾的是，如果使用非面向对象的语言，则只能通过这种复杂的方式来处理新对象类型的需求。当需求不可避免地更改时，应用程序无疑将变得复杂，并且维护成本极高。

幸运的是，我们有另外的选择！

3. 方法 3：使用继承扩展现有的 Student 类

使用面向对象编程语言(OOPL)，就能利用继承(inheritance)来解决问题。继承是一种强大的机制，它通过指出新旧类的不同之处，就能在旧类上定义一个新类。使用继承，就能声明一个继承 Student 类所有成员的新类，即 GraduateStudent。GraduateStudent 只需关注与研究生相关的两个字段：undergraduateDegree 和 undergraduateInstitution。在 C# 类声明中，继承可通过在基类名称后面跟冒号的方式来表示。

```
public class GraduateStudent : Student {
    // 声明 Student 类没有声明的两个新字段……

    private string undergraduateDegree;
    private string undergraduateInstitution;

    // 以及两个字段的属性。

    public string UndergraduateDegree {
      get {
        return undergraduateDegree;
      }
      set {
        undergraduateDegree = value;
      }
    }
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
        public string UndergraduateInstitution {
           get {
             return undergraduateInstitution;
           }
           set {
             undergraduateInstitution = value;
           }
        }
     }
```

这些就是在 GraduateStudent 新类中所要声明的内容：两个字段及其相关的属性。这里没有必要复制任何 Student 类的成员，因为 GraduateStudent 自动地继承了这些成员。这就像从 Student 类中“借”字段、属性和方法并添加到 GraduateStudent 类一样，不过免去了实际操作的麻烦。

在使用继承时，原来的类(即 Student 类)称为基类(base class)，新类 GraduateStudent 称为派生类(derived class)。我们说派生类扩展了基类，也可以说子类从超类继承。

继承通常是指两个类之间的“是……”关系，因为如果类 B(GraduateStudent)派生自类 A(Student)，则 B 正是 A 的特例。基类具有的特点对于其所有派生类都适用，即：

- Student 可以上课，所以 GraduateStudent 也可以上课
- Student 有导师，所以 GraduateStudent 也有导师
- Student 攻读学位，所以 GraduateStudent 也攻读学位

事实上，判断继承关系的方法是：如果类中有一些特性不适用于类 B，则类 B 不应该是类 A 的派生类。

然而要注意，上面的说法反之则不然：因为派生类是基类的特例，所以派生类的特性不一定适用于基类，例如：

- GraduateStudent 已经本科毕业，但普通意义上的 Student 还没有本科毕业。
- GraduateStudent 已经获得本科学位，但普通意义的 Student 则不一定。

因为派生类是基类的特例，则术语特殊化(specialization)用来说明从一个类派生新类的过程。另一方面，一般化(generalization)用来说明相反的过程：从几个现存类中找出共性并为它们创建一个新的共同基类。假设要创建 Professor 类。Student 和 Professor 有一些共同点：name、birthDate 等字段以及操作这些字段的方法。但它们又有各有独特的字段：Professor 类可能需要 title(string 类型)和 workFor(Department 的引用)字段，而 Student 类则需要 StudentID、degreeSought 和 majorField 等 Professor 类没有的字段。因为每个类都有一些对其他类毫无用处的字段，所以这两个类不能互相派生。然而，在两个类中重复同样的字段声明和方法代码是非常低效的。

在这种情况下，可能需要创建一个新的基类，称为 Person，它集成了 Student 和 Professor 类中的共有成员，然后让 Student 类和 Professor 类从 Person 类继承这些共有成员。以下是结果代码：

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
// 定义基类：
public class Person
{
   // Student 和 Professor 共有的字段。
   private string name; // 参见本例后面有关私有可访问性在继承中的使用说明。
   private string address;
   private string birthDate;
   // 共有属性。
   public string Name {
     get {
       return name;
     }
     set {
       name = value;
     }
   }

   // 等等。
}

// 从 Person 类派生 Student 类……
public class Student : Person
{
   // 学生特有的字段。
   private string studentId;
   private string majorField;
   private string degreeSought;

   // 学生特有的属性。
   public string StudentId {
     get {
       return studentId;
     }
     set {
       studentId = value;
     }
   }

   // etc.
}

// 从 Person 类派生 Professor 类！
public class Professor : Person
{
   // 教授特有的字段。
   private string title;
   private Department worksFor;

   // 教授特有的属性。
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
    public string Title {
      get {
        return title;
      }
      set {
          title = value;
      }
    }

    // 等等。
}
```

注意

第 13 章将介绍继承私有成员的复杂之处，以及由此引出的“受保护的”可访问性。但目前还不会介绍受保护的可访问性，因为您尚未具备足够的背景知识。

5.1.4 继承的优点

继承有可能是面向对象语言中最强大且独特的方面之一，因为：

- 派生类比非继承类更简洁。派生类只包含与基类不同的“要素”。例如，从 GraduateStudent 类的定义可以看出，研究生是“已经从教育机构获得本科学位”的学生。因此，派生类让应用程序的代码相对于传统非面向对象实现的代码量大大减少。
- 通过继承，能够重用和扩展已经被彻底测试过的代码，且无需修改之。如前所见，您可以创建一个新类 GraduateStudent，而无需扰乱 Student 类的代码。这样，就能确保任何依赖于 Student 对象实例化和调用方法的客户代码不受派生类 GraduateStudent 的影响，也避免了对现有应用程序的大量测试工作(如果使用非面向对象的方法向 Student 类添加研究生特有的字段，则必须重新测试整个应用程序，以确保一切正常)。
- 最妙的是，即使没有基类的源代码，也可以从中派生出新类！只要有编译好的类，继承机制便会起作用；扩展一个类不需要其源代码。这是面向对象语言提高生产力的最有效方式之一：找一个能满足您需求的类(别人写的或是语言内置的)，从该类派生一个类，添加您所需的成员；或者购买第三方的类库，如法炮制。
- 最后，如第 2 章所示，分类是人类组织信息的自然方式；所以，唯有按照这种方式组织软件，并使其更直观，方可达到易于维护和扩展的目的。

5.1.5 继承的缺点

继承是广泛应用于面向对象编程中的强大概念，但基类和派生类的结对关系会使对基类的任何修改都将影响到所有派生类。仔细地设计类层次结构能够避免或最小化可能发生的问题(本书第 II 部分将讨论这个概念)。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

5.1.6 类的层次结构

到目前为止，我们得到了一棵倒立的类树，它由通过继承而彼此关联的类组成：这样的树称为类层次结构(class hierarchy)。类层次结构的示例如图 5-5 所示。

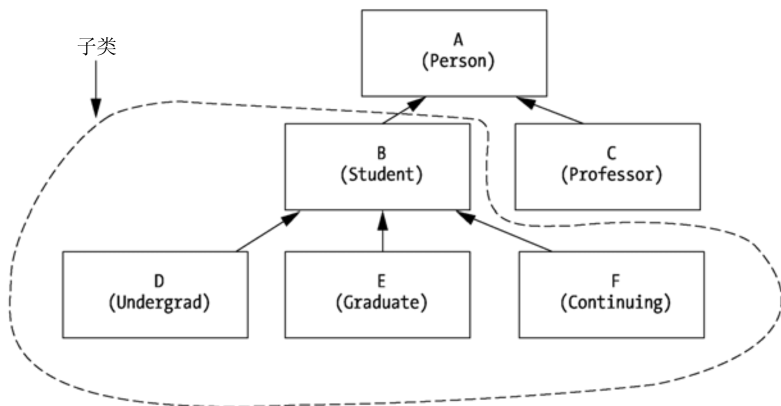


图 5-5 类层次结构示例

定义如下：

- 把层次结构中的每个类称为节点(node)。
- 任意一个节点(直接地或间接地)派生自位于其上方层次结构中的节点，这些节点称为祖先(ancestor)。
- 最接近给定节点其上的祖先节点称为该节点的直接基类(direct base class)。
- 反之，层次结构中位于给定节点下方的所有节点称为其子孙(descendant)。
- 层次结构顶端的节点称为根节点(root node)。
- 终端节点(terminal node)或叶节点(leaf node)没有子孙节点。
- 派生自同一基类的两个节点是兄弟(sibling)。
- 每个派生类用向上的箭头指向直接基类。

将上述术语应用到图 5-5 所示的层次结构示例中：

- 类 A(Person)是整个层次结构中的根节点。
- 类 B、C、D、E 和 F 派生自类 A，所以是 A 的子孙。
- 类 D、E 和 F 派生自类 B。
- 类 D、E 和 F 是兄弟，类 B 和类 C 也是兄弟。
- 类 D 有两个祖先：A 和 B。
- 类 C、D、E 和 F 是终端节点，它们没有派生类。

注意

在 C#语言中，Object 类(属于 System 名称空间)是所有其他类型的最终基类，无论是

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

用户自定义类还是语言内置的类。本书第Ⅲ部分将深入讨论 Object 类。

类层次结构可能会发生以下变化：

- 当产生新的兄弟或分支时，它会变宽。
- 当进行特殊化时，会向下扩展。
- 当进行一般化时，会向上扩展。

当有新的需求或对现有需求的理解有变化时，以上改变就会发生。例如，可能需要 MastersStudent 和 PhDStudent 类(GraduateStudent 的特殊化类)或者给 Student 和 Professor 增加兄弟类 Administrator，此时的类层次结构如图 5-6 所示。

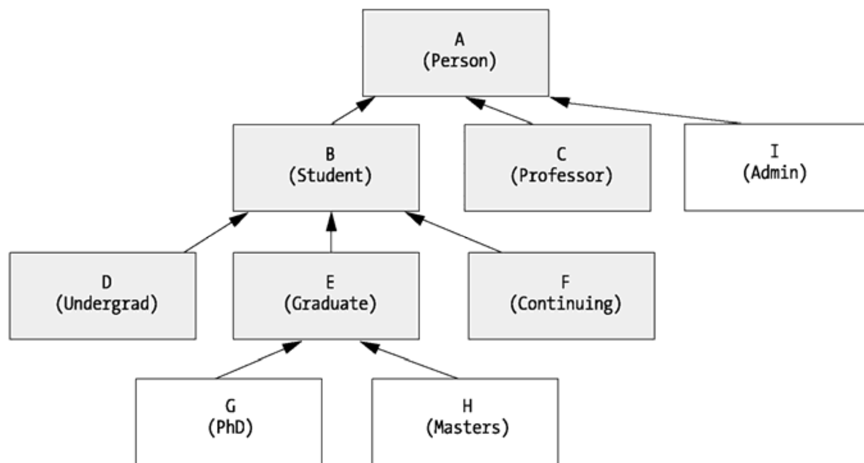


图 5-6 类层次结构几乎总是在扩展

5.1.7 继承是否真是一种关系

如本章前面所述，关联和聚集(关联的特殊形式)是对象关系，因为它们被实现为对象的引用。两个不同对象通过它们各自类之间的关联而联系在了一起。另一方面，继承是类的关系，因为它在类定义中被静态地(在编译程序前)定义。通过继承，对象同时是派生类和所有其基类的实例，GraduateStudent 是 Student，也是 Person。

重新回顾图 5-6 中的层次结构，可以发现：

- Person 类层次结构中的所有类(Student、Professor、Graduate 等)都共享 Person 类的特征并与之兼容。例如，如果声明了保存 Person 引用的数组，它可用来保存 Professor、Continuing、Student 以及 Person 的引用。
- 子层次结构的工作方式相同，如 Undergrad、GraduateStudent 和 PhDStudent 类共享 Student 类的特征并与之兼容。

对象共享基类的特征是一个重要的概念，本书后面还会多次提到。

5.1.8 避免“连锁反应”

一旦类层次结构已经建立，且应用程序编码已经完成，则对非叶节点类(即有子孙类的节点)的修改将导致层次结构中向下的“连锁反应”。例如，如果在建立 GraduateStudent

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

类之后又向 Student 类添加一个 minorField 字段，则 GraduateStudent 会在下次编译时继承这个新的字段，这可能正是我们想要的；但另一方面，如果当初在设计 Student 类时没有预料到会派生出 GraduateStudent 类，则这可能并非我们所愿！

在理想情况下，Student 类的开发者要告诉所有派生类(GraduateStudent、MastersStudent 和 PhDStudent)的开发者，让它们同意对 Student 所做的任何修改。但事实可能并不理想，我们甚至不知道自己的类被别人扩展了，例如，代码已经发布并在其他项目中被重用，或是卖给了客户。这就引出了一个通用规则：

在应用程序建立好非叶节点后，除非万不得已，应避免向其添加新的公有可访问成员，从而避免在整个继承层次结构中造成连锁反应。

这条规则知易行难。然而，它强调了花费尽可能多的时间在需求分析上、而不是急于编码的重要性。这虽然不能阻止日后新需求的产生，但却能在考虑当前需求时不致失察。

5.1.9　派生类的规则：可为

当派生新类时，可采取以下操作对基类进行特殊化处理。

- 可通过添加成员来扩展基类。在 GraduateStudent 示例中，添加了 4 个成员：2 个字段(undergraduateDegree 和 undergraduateInstitution)和 2 个属性(UndergraduateDegree 和 UndergraduateInstitution)
- 可对派生类执行从基类继承而来的方法进行特殊化。例如，当学生选修课程时，首先要求该学生
  - 完成所需的先修课程。
  - 该课程是学生获得学位所必需的课程。
  - 另一方面，当研究生选修课程时，除了上面这两个要求，还要让学位委员会确信这门课适合该生选修。

与基类所实现的方法相比，特殊化派生类执行方法的方式可通过重写技术来完成。

重写

重写(overriding)是指不通过修改方法在基类中声明的接口/签名而在派生类内部重写方法或属性的工作原理。例如，假设在 Student 类声明了一个 Print 方法，用来输出所有学生的字段值：

```
public class Student
{
  // 字段。
  private string name;
  private string studentId;
  private string majorField;
  private double gpa;
  // 等等。
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
    // 还会提供每个字段的属性，细节从略。

    public void Print() {
      // 输出 Student 类的所有字段值，注意 get 存取器的用法。
      Console.WriteLine("Student Name: " + Name + "\n" +
                        "Student No.: " + StudentId + "\n" +
                        "Major Field: " + MajorField + "\n" +
                        "GPA: " + Gpa);
    }
}
```

上例的 Print 方法假设都已经为 Student 类的所有字段值编写了属性。Print 方法使用属性来访问相关的字段值，而不是直接访问字段：

```
// 该示例直接通过名称访问字段，而不是通过相关属性；本书不鼓励这种方法。
Console.WriteLine("Student Name: " + name + "\n" +
                  "Student No.: " + studentId + "\n" +
                  "Major Field: " + majorField + "\n" +
                  "GPA: " + gpa);
```

在类方法内部使用 get 存取器是最佳实践，在第 4 章曾讨论过；它能充分利用值的检查功能或 get 存取器所能提供的其他操作。

通过继承，所有 Student 的派生类都将继承该方法。然而，还有一个问题：我们向 GraduateStudent 派生类添加了两个新字段：undergraduateDegree 和 undergraduateInstitution。如果只是采取懒办法让 GraduateStudent 类继承 Student 类的 Print 方法，则在调用 GraduateStudent 的 Print 方法时，总会输出从 Student 类继承的 4 个字段——name、studentId、majorField 和 gpa，因为 Print 方法的代码只能输出这些字段值。理想情况是，GraduateStudent 的 Print 方法不仅能输出这 4 个字段，还能输出 undergraduateDegree 和 undergraduateInstitution 这两个添加的字段。

通过面向对象语言，我们能够重写或者说替换 GraduateStudent 类从 Student 类继承而来的 Print 方法。要在 C#中重写基类的方法，则必须先在所属类的基类中，使用 virtual 关键字将该方法声明为虚方法。声明一个方法为虚方法，则说明该方法可被(而非必需)派生类重写。

派生类可在方法声明中使用 override 关键字来重新实现基类的虚方法。派生类中重写的方法必须有与基类方法相同的可访问性、返回类型、名称和参数列表。

下面介绍 GraduateStudent 类如何重写 Student 类的 Print 方法：

```
public class Student
{
    // 字段。
    private string name;
    private string studentId;
    private string majorField;
    private double gpa;
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
    // 等等。

    // 将为每个字段提供属性，细节从略。

    // 将 Student 类的 Print 方法声明为虚方法。
    public virtual void Print() {
      // 输出 Student 类的所有字段值，再次注意 get 存取器的用法。
      Console.WriteLine("Student Name: " + Name + "\n" +
                        "Student No.: " + StudentId + "\n" +
                        "Major Field: " + MajorField + "\n" +
                        "GPA: " + Gpa);
      }
  }

  public class GraduateStudent : Student
  {
    private string undergraduateDegree;
    private string undergraduateInstitution;

    // 也要为每个新添加的字段提供属性，细节从略。
    // 重写 Student 类的 Print 方法。
    public override void Print() {
      // 输出 GraduateStudent 类的所有字段值，即从 Student 类中继承的字段加上自己
     // 显式声明的字段。
      Console.WriteLine("Student Name: " + Name + "\n" +
                        "Student No.: " + StudentId + "\n" +
                        "Major Field: " + MajorField + "\n" +
                        "GPA: " + Gpa + "\n" +
                        "Undergrad. Deg.: " + UndergraduateDegree + "\n"+
                        "Undergrad. Inst.: " + UndergraduateInstitution);
    }
}
```

这样，GraduateStudent 类的 Print 方法就重写/替代了从 Student 类继承的版本。

上例并不完美，因为 GraduateStudent 类 Print 方法的前 4 行重复了 Student 类 Print 方法中的代码。您应该避免代码冗余，因为它会让维护变成一场噩梦：如果想在应用程序的一个位置修改代码，就得记住还要在无数个位置修改相同的代码，或者更糟的是，会引起代码逻辑矛盾。我们应该尽量避免代码冗余，但却鼓励代码重用，所以 GraduateStudent 类的 Print 方法可重写为：

```
public class GraduateStudent : Student
{
  // 细节从略……

  public override void Print() {
    // 通过调用由基类 Student 定义的 Print 方法来重用代码……
    base.Print();
    // 然后打印派生类的特有字段。
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
        Console.WriteLine("Undergrad. Deg.: " + UndergraduateDegree + "\n" +
                          "Undergrad. Inst.: " + UndergraduateInstitution);
    }
}
```

当需要调用在基类中定义的方法时，将使用 C#关键字 base 作为方法的限定符：

```
base.methodName(arguments);
```

在复杂的继承层次结构中，有时需要多次地重写方法。在如图 5-6 所示的层次结构中

- 根类 A(Person)声明以下方法头，该方法头将输出所有 Person 类声明的字段：

  ```
  public virtual void Print()
  ```

- 派生类 B(Student)重写该方法，修改方法体的内部逻辑，不仅输出从 Person 类继承的字段，还会输出在 Student 类中增加的字段。重写的方法头如下：

  ```
  public override void Print()
  ```

- 派生类 E(GraduateStudent)再次重写该方法，不但输出从 Student 类继承的字段(其中也有从 Person 类继承的)，还会输出 GraduateStudent 自己添加的字段。GraduateStudent 类的 Print 方法也要使用 override 关键字：

  ```
  public override void Print()
  ```

注意，在所有情况下，可访问性、返回类型和方法签名都必须保持一致——public void Print( )——这样重写才有效。

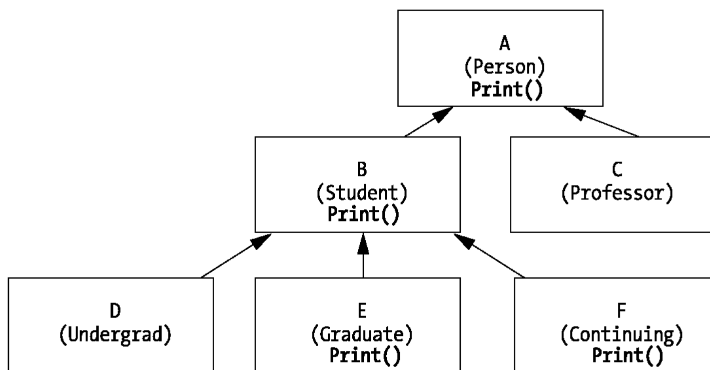


图 5-7 一个方法可在类层次结构中多次重写

在某些情况下，如果一个类没有特别重写某个方法，则将继承从其直接祖先中定义的方法。

5.1.10 派生类的规则：不可为

派生新类时，有些事不应该去做(而且面向对象编程语言也不允许编译通过，以阻止此类事件的发生)。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

1. 不要更改成员的语义

不应该更改语义——即成员的目的或含义。例如：

- 如果基类(Student)的 Print 方法用来在计算机屏幕上显示对象所有的字段值，则派生类(GraduateStudent)的 Print 重写方法就不该把输出方式改为写文件。
- 如果基类(Person)的 name 字段用来存储以“姓，名”作为格式的人名，则派生类(Student)的 name 字段也应该有相同的作用。

2. 不要删除成员

我们不能通过在派生类中忽略某些成员而删除从基类继承而来的成员。这样做会破坏“是……”的层次结构。继承的定义要求基类的所有成员都能体现在派生类 A 上，这样才能保证 A 是真正的派生类。如果 GraduateStudent 类删除了从 Student 类继承而来的 degressSought 字段，那么 GraduateStudent 还是 Student 吗？

3. 不要更改属性类型

派生类可重写基类属性，但是属性类型必须保持和基类的类型一致。例如，如果 Person 类将 Birthdate 属性类型声明为 string：

```
public class Person
{
  // 细节从略。

  // 基类引入一个属性。
  public virtual string Birthdate {
    get {
        // 细节从略。
    }
  }
}
```

则 Person 类的派生类 Student 不能在重写 Birthdate 属性时更改其类型，例如，修改为 DateTime：

```
public class Student : Person
{
    // 细节从略。

    // 派生类重写属性，试图更改其类型。
    public override DateTime Birthdate { // 无法编译
      get {
          // 细节从略。
      }
    }
}
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

如果试图编译 Student 类，将会得到以下编译错误：

```
error: 'Student.Birthdate' type must be 'string' to match overridden
member 'Person.Birthdate'
```

注意

派生类可通过隐藏(hiding)技术来更改基类属性的类型。第 13 章将讨论属性和方法隐藏。

4. 不要更改方法头

如果 Student 类从 Person 类继承的 Print 方法头为不带参数的 public void Print( )，那么 Student 类就不能更改方法头，例如，让它接受 1 个实参：public void Print(int noOfCopies)。这样做会创建一个完全不同的方法，在 C#语言中这种特性称为重载(overloading)。

5.1.11 重载

重载允许同一个类中的两个或多个不同方法拥有相同的方法名和不同的参数签名(见第 4 章)。重载方法和重写方法不一样，在重写方法中，派生类实现的是和基类相同的方法头。例如，Student 类可能会合法地声明以下 5 种不同的 Print 方法：

```
void Print(string fileName) ——一个参数
void Print(int detailLevel) ——和上例不同的一个参数类型
void Print(int detailLevel, string fileName) ——两个参数
int Print(string reportTitle, int maxPages) ——不同的参数类型
bool Print()——无参数
```

因此，可以说 Print 方法被重载了。注意以上 5 个方法签名在其参数签名上均不同：

- 第一个方法带 1 个 string 参数。
- 第二个方法带 1 个 int 参数。
- 第三个方法带 int 和 string 两个参数
- 第四个方法带 string 和 int 两个参数(虽然和上例的参数类型相同，但顺序不同)。
- 第五个方法不带参数。

因此这 5 个方法头代表了不同的有效方法，而且可以和其他方法共存，编译器也不会有任何抱怨！您可以根据在 Student 对象上调用的方法形式来决定执行哪个 Print 方法版本：

```
Student s = new Student();

// 调用带 1 个 string 参数的方法版本。
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
s.Print("output.rpt");

// 调用带 1 个 int 参数的方法版本。
s.Print(2);

// 调用带 int 和 string 两个参数的方法版本。
s.Print(2, "output.rpt");

// 等等。
```

编译器能通过参数签名无歧义地判断在每个实例上调用的 Print 方法版本。

上例暗示了只有参数类型和顺序才能决定是否能添加新方法，因为参数名和方法的返回类型无法在方法调用中说明。例如：

● 您已经知道，不能向 Student 类再添加第 6 个 Print 方法：

```
bool Print(int levelOfDetail)
```

因为其参数签名(1 个 int 参数)和已有方法的参数签名重复，虽然这两个方法头中有不同的返回类型(bool 和 int)和不同的参数名称：

```
int Print(int detailLevel)
```

● 假设允许我们给 Student 类添加这第 6 个 Print 方法 bool Print(int levelOfDetail)。如果编译器在客户代码中看见以下方法调用，则它不知道该调用哪个方法，因为您只能从方法调用中看到方法名和参数类型(本例中是 int 类型)：

```
s.Print(3);
```

所以编译器将在第一时间防止类声明具有相同签名的方法。

构造函数(第 4 章曾介绍过，它是用来初始化对象的特殊函数)经常会被重载。以下是有多个重载构造函数的类示例：

```
public class Student
{
  private string name;
  private string id;
  private int age;
  // 等等。

   // 第一个构造函数。
   public Student() {
     // 如有需要，给选中的字段赋默认值。
     id = "?";
     // 如果没有在构造函数中显式地初始化某些字段，则编译器会自动假设为其提供与零等
价
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
    // 的值。
  }

  // 第二个构造函数。
  public Student(string s) {
    id = s;
  }

  // 第三个构造函数。
  public Student(string s, string n, int i) {
    id = s;
    name = n;
    age = i;
  }

  // 等等，其他方法从略。
}
```

通过向客户代码提供一系列构造函数，类具有了更多的灵活性。

重载让我们能够创建具有相似方法名称的方法家族，它们完成相同的工作，但接受不同的参数类型。回顾第 1 章，本书介绍了 Write 方法，它用来将输出显示在控制台上。事实上，Write 方法不止一个，而是有很多个，每个 Write 方法有不同的参数类型(Write (int)、Write (string)、Write(double)等)。使用 Write 重载方法，比用不同的方法名称(WriteString、WriteInt、WriteDouble 等)来区分显得简单灵活。

注意，没有“字段重载”这个概念，即不可以在类中声明具有相同名称的两个字段：

```
public class SomeClass
{
    private string foo;
    private int foo;
    // 等等。
```

编译器会报错：

```
  SomeClass.cs(5,15): error: The type 'SomeClass' already contains
a definition for 'foo'
```

5.2 略谈多重继承

到现在为止，本书介绍的继承层次结构还只是单重继承(single inheritance)层次结构，因为层次结构中任何一个类都只有一个直接的基类(直接祖先)。例如，在图 5-8 所示的层次结构中，B、C 和 I 都只有一个直接基类 A；D、E 的直接基类是 B，而 G 和 H 的直接基类是 E。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

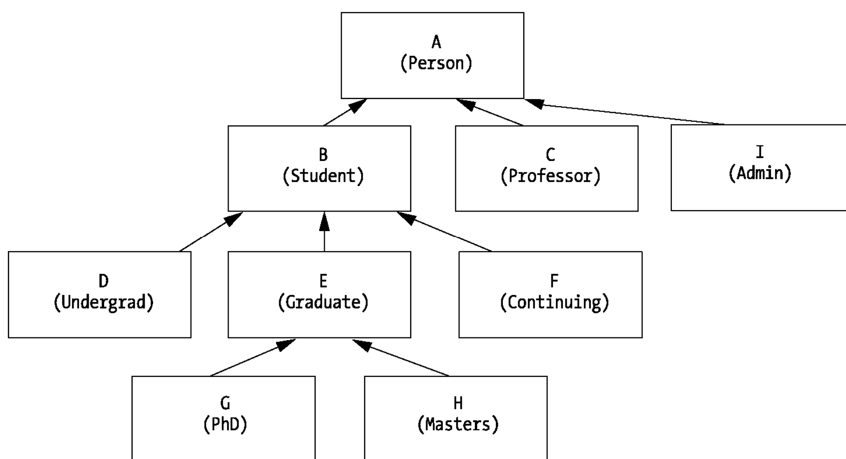


图 5-8　单重继承层次结构的示例

如果出于某种原因，需要把两个不同基类的特征融合到一起，创建一个混合类，则需要使用多重继承(multiple inheritance)。虽然 C#不允许多重继承，但多重继承允许类层次结构中的任何一个类有两个或两个以上的类作为其直接祖先。

例如，Professor 类表示教课的人，Student 类表示听课的人。怎样才能让教授通过 SRS 选课或者让学生(最可能是研究生)教本科课程呢？为了能将这两种人表示为对象，需要组合 Professor 类和 Student 类的成员，使其成为 ProfessorStudent 类。类层次结构如图 5-9 所示。

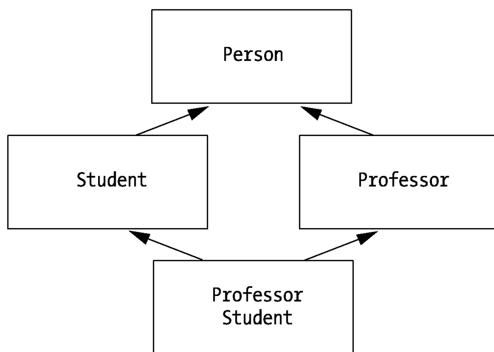


图 5-9　多重继承允许派生类有多个直接祖先类

表面上看，这易如反掌。然而，多重继承有很多的复杂性，所以 C#语言的设计者选择不支持多重继承。他们提供另一种替代机制来创建“分裂性格(split personality)”的对象(即行为表现为两个或多个实体的对象)。这种机制涉及接口(interface)概念，第 7 章将详细介绍这一概念。因此，如果您只对 C#语言中的对象概念感兴趣，请略过后面一节。如果您想了解更多有关多重继承的内容，请继续阅读。

上例存在的问题是：通过继承，派生类自动继承其基类的字段和方法。那有两个或多个基类时情况会如何呢？如果这些基类没有相同的字段名或方法签名，则万事大吉。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

如果直接基类具有下列情况，会怎样呢？

- 有相同签名的方法，但方法体的实现却不同
- 有相同的字段(名称和类型相同)
- 有同名的字段，但类型不同

下面将介绍一个示例。

首先，创建一个简单的 Person 类并声明 1 个字段和 1 个方法(GetDescription)，如下所示：

```
public class Person
{
  private string name;

  // 不显示公有的 Name 属性语法。

  public virtual string GetDescription() {
    return name;
    // 例如， "John Doe"
  }
}
```

稍后，我们决定特殊化 Person 类，创建两个派生类——Professor 和 Student——它们都添加了一些新的字段并利用这些新字段重写了 GetDescription 方法，如下所示：

```
   public class Student : Person
   {
       // 添加两个字段。
       string major;
       int id; // 不重复的学生编号。

       // 重写从 Person 继承的方法。
       public override string GetDescription() {
         return Name + " [" + major + "; " + id + "]";
         // 例如，"Mary Smith [Math; 10273]"
     }
}

public class Professor : Person
{
    // 添加两个字段，注意这里的 id 与 Student 类的 id 字段相同，但类型不同。
    string title;
    string id; // 不重复的学生编号

    // 重写从 Person 继承的方法。
    public override string GetDescription() {
      return Name + " [" + title + "; " + id + "]";
      // 例如，"Harry Henderson [Chairman; A723]"
    }
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
}
```

注意，这两个派生类都碰巧有 id 字段，但在 Student 类中它被声明为 int 类型，而在 Professor 中被声明为 string 类型。同样要注意，这两个类都重写了 GetDescription 方法，从而利用自己类中的字段。

随着系统的演化，我们决定需要一种“既是教授又是学生”的对象，所以创建了混合类 ProfessorStudent，它是 Student 和 Professor 的派生类。我们不想添加任何字段或方法，而只是把两个基类的特征融合在一起，所以理想的 ProfessorStudent 声明如下：

```
// * * *重要提醒：C#不允许多重继承!!! * * *
class ProfessorStudent : Professor and Student
{
    // 允许类体为空，类本身并不“空”，因为它继承了基类的成员。
}
```

但是这样做马上会碰到问题。

首先，字段名会冲突。如果只是简单地继承 Professor 和 Student 类的所有字段，则会陷入如表 5-1 所示的困境。

表 5-1 多重继承会给派生类的成员带来很多歧义性

字　段	说　明
string name;	从 Student 继承，也就是从 Person 类继承
string major;	从 Student 继承
int id;	从 Student 继承，与从 Professor 类继承的 string id 字段冲突(编译器不允许两者共存)
string name;	从 Professor 继承，也就是从 Person 继承，字段重复，编译器不允许这种情况发生
string title;	从 Professor 继承
string id;	从 Professor 继承，与从 Student 类继承的 int id 字段冲突(编译器不能允许两者共存)

让编译器足够智能，从而解决并删除重复的 name 字段，这不是件很难的工作，但对于 int id 和 string id 该怎样处理呢？编译器无法知道该删除哪个字段，而且也不应该删除任何一个字段，因为它们表示了不同的信息。唯一的选择是回到 Student 或 Professor 类(或两者皆是)，将其 id 字段重名为 StudentId 和(或)ProfessorId，从而明确两个字段表示了不同的信息。这样，ProfessorStudent 就能毫无问题地继承这两个字段。但是，如果我们无法控制这两个基类的源代码，则会陷入麻烦。

另一个问题是编译器不会清楚要继承哪个 GetDescription 方法。可能这两个方法我们都不需要，因为它们不能访问另一个类中的字段；但即使想要使用其中一个，也必须

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

告诉编译器要继承哪个方法，或者强制在 ProfessorStudent 类中重写 GetDescription 方法。

这虽然只是个简单示例，但它也道出了面向对象语言实现多重继承之痛。

5.3 回顾面向对象编程语言的 3 个显著特点

第 3 章介绍了判断一种编程语言是否是面向对象语言的 3 个关键机制，本章中定义了其中两个：

- (程序员创建的)用户自定义类型，第 3 章介绍过
- 本章讨论的继承
- 多态(Polymorphism)

现在，只剩下多态没有介绍了，我们将其放到了后面的章节中(第 7 章)。这里先不介绍多态，而是讨论如何将一组对象组织为特殊的对象类型(称为集合)。

5.4 本章小结

本章介绍的内容如下：

- 关联描述了类之间的关系——即两个特定类型/类的对象之间的潜在关系。
- 本书还定义了类 X 和类 Y 之间的关联多样性，即有多少个类型为 X 的对象可链接到类型为 Y 的对象，反之亦然。多样性包含一对一(1:1)、一对多(1:m)和多对多(m:m)。在所有这些情况中，关系两端的对象可以是可选的或必须的。
- 聚集是一种特殊的关联，它实现了包含关系。
- 如何通过继承根据已有类派生新类，以及在派生新类时可为和不可为的事情。我们可以通过添加成员来扩展基类或通过重写方法来特殊化基类。
- 类层次结构如何随着时间推移而发展，在需求变化时，如何避免给应用程序带来连锁反应。
- 重载可用来创建名称相同但参数签名不同的多个方法。
- 多重继承在面向对象语言中的麻烦之处。

5.5 练习

(1) 回顾第 2 章练习(3)的解决方案。对于您提出的所有类，请列出这些类之间可能存在的关联。

(2) 如果 FeatureFilm 类定义了以下方法：

```
public void Update(Actor a, string title);
public void Update(Actor a, Actor b, string title);
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
public void Update(string topic, string title);
```

则编译器可允许通过以下哪些方法头？

```
public bool Update(string category, string theater);
public bool Update(string title, Actor a);
public void Update(Actor b, Actor a, string title);
public void Update(Actor a, Actor b);
```

(3) 以下代码说明了方法的重载、重写和直接继承：

```
class Vehicle
{
  string name;

  public virtual void Fuel(string fuelType) {
    // 细节从略……
  }

public virtual bool Fuel(string fuelType, int amount) {
    //细节从略……
  }
}


class Automobile : Vehicle
{
    public virtual void Fuel(string fuelType, string timeFueled) {
      // 细节从略……
    }

    public override bool Fuel(string fuelType, int amount) {
      //...
    }
  }
  class Truck : Vehicle
  {
    public override void Fuel(string fuelType) {
      //...
    }
}

class SportsCar : Automobile
{
    public override void Fuel(string fuelType) {
      //...
    }
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
        public override void Fuel(string fuelType, string timeFueled) {
          //...
        }
    }

    // 客户代码:

    Truck t = new Truck();
    SportsCar sc = new SportsCar();
```

这 4 个类分别能识别多少个不同的 Fuel 方法签名？

(4) 给定以下简单代码：

```
class FarmAnimal
{
  string name;

  public virtual string Name {
    get {
      return name;
    }
    set {
      name = value;
    }
  }

  public virtual void MakeSound() {
    Console.WriteLine(Name + " makes a sound...");
  }
      }

  class Cow : FarmAnimal
  {
    public override void MakeSound() {
      Console.WriteLine(Name + " goes Moooooo...");
    }
  }
  class Horse : FarmAnimal
  {
    public override string Name {
      set {
          base.Name = value + " [a Horse]";
      }
  }
}
```

则以下客户代码的输出是什么？

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
Cow c = new Cow();
Horse h = new Horse();
c.Name = "Elsie";
h.Name = "Mr. Ed";
c.MakeSound();
h.MakeSound();
```