第 1 章　OCC 体系结构和基本概念

OCC 是用面向对象方法设计的一个 CAD 基础平台（软件）。为了能从整体上把握 OCC 的组织情况，也为了方便后续章节的讨论，本章将介绍 OCC 体系结构和几个基本概念。

1.1　OCC 体系结构

1.1.1　面向对象方法和面向对象的软件工程

在介绍 OCC 体系结构之前，先介绍面向对象方法的概念和什么叫面向对象的软件工程。

在面向对象的方法出现以前，程序员都采用面向过程的程序设计方法，其中典型的是结构化程序设计。这种设计的思路是：自顶向下、逐步求精。其程序结构是按功能划分为若干个基本模块，这些模块形成一个树状结构。各模块间的关系尽可能简单，在功能上相对独立；每一模块内部均是由顺序、选择和循环三种基本结构组成。其模块化实现的具体方法是使用子程序。结构化程序设计由于采用了模块分解与功能抽象以及自顶向下、分而治之的方法，从而有效的将一个复杂的程序系统设计任务分解成许多易于控制和处理的子任务，便于开发和维护[2]。这种设计方法的致命缺点是：程序的可重用性差。因为它把数据和处理数据的过程分离为相互独立的实体，当数据结构改变时，所有相关的处理过程都要进行相应的修改。

而面向对象的方法将数据及对数据的操作放在一起，作为一个相互依存、不可分离的整体——对象。对同类型对象抽象出其共性，形成类。类中的大多数数据，只能用本类的方法进行处理。类通过一个简单的外部接口与外界发生关系，对象与对象之间通过消息进行通信[2]。这样，程序模块间的关系更为简单，程序模块的独立性、数据的安全性就有了良好的保障，实现了“高内聚”“低耦合”。另外，继承与多态性可以大大提高程序的可重用性，使得软件的开发和维护都更为方便。

面向对象的软件工程是面向对象方法在软件工程领域的全面应用。它包括面向对象的分析（OOA）、面向对象的设计（OOD）、面向对象的编程（OOP）、面向对象的测试（OOT）和面向对象的软件维护（OOSM）等主要内容[2]。

1.1.2 OCC 的体系结构

整个 OCC 就是用面向对象方法设计出来的一个对象库。之所以用面向对象方法而不是面向过程方法，是因为用面向对象方法有三个好处。第一，由面向对象方法抽象的系统结构能映射到数据库结构中,很容易实现程序与数据结构的封装。第二，面向对象方法从所处理的数据入手,以数据为中心来描述系统,数据相对于功能而言,具有更强的稳定性,这样设计出的系统模型往往能较好地映射问题域模型[3]。第三，对象、类、继承性、多态性的引入使用,令面向对象的设计方法能更好地生产可重用的软件构件和解决软件的复杂性问题。

不过，面向对象的设计方法要求开发人员必须花很大精力去分析对象是什么，每个对象应该承担什么责任，所有这些对象怎样很好地合作以完成预定的目标。这样做换来的好处是：提高了目标系统的可重用性，减少了生命周期后续阶段的工作量和可能犯的错误，提高了软件的可维护性[3]。

用面向对象方法和软件工程思想分析，整个 OCC 由五个模块组成，分别是基础类模块、建模数据模块、建模算法模块、可视化模块、数据交换模块和应用程序模块。其中，建模数据模块主要提供二维和三维几何模型的数据结构，也称数据结构模块。

一个模块主要由一个或几个工具箱构成。当然它也可以包含一些执行体和资源体等。就结构上看，一个工具箱就是一个共享库（如.so 或.dll 类型的文件）。每个工具箱由一个或几个包组成。而每个包则由许多类组成，例如，

一个几何包包含点类、线类和圆类等。在同一个包中，不能含有相同名字的两个类。使用类的时候，类名要以包名作前缀，如 Geom_Circle。

图 2.1 简要说明了包的内容。

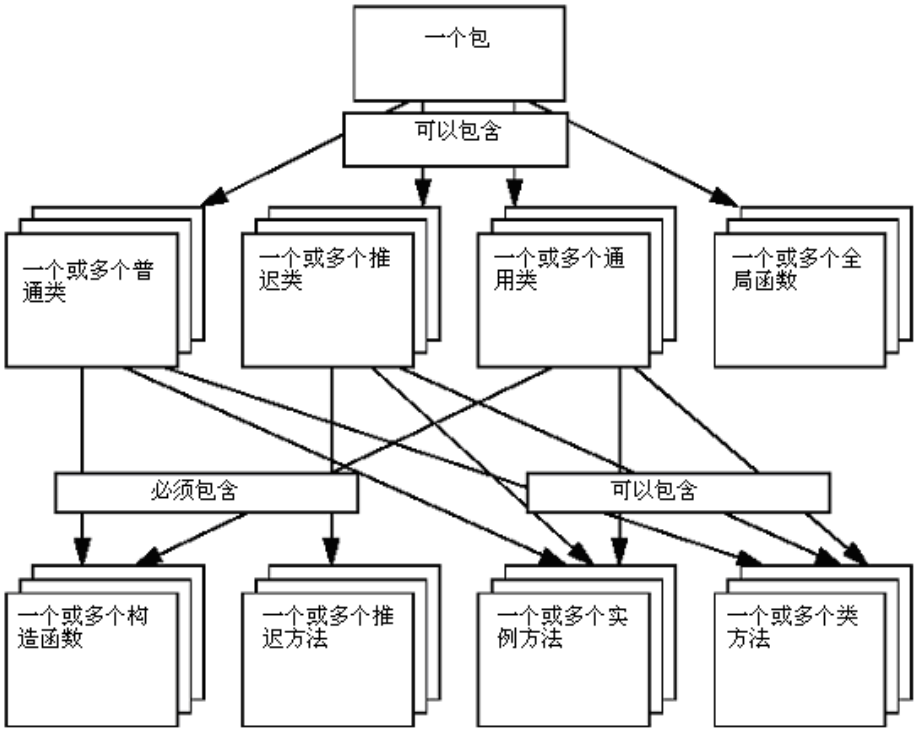


图 2.1 包的内容

1.2 基本概念

1.2.1 类和泛化

1、类

OCC 是一个面向对象的软件，与所有面向对象的软件一样，其最基本的软件成分是类。一个类就是一种数据类型的实现。类有自己的行为（由它的函数提供的服务）和结构（类的数据结构——用来存储其数据）。

OCC 中所有类按其实现方式可以分三种：普通类、推迟类和通用类。普通类含有实例方法，可以被直接实例化。而推迟类则不能被实例化。推迟类的作用在于使一层类共同拥有一种给定的行为，而这些行为的发生取决于普通类（推迟类的派生类）的实现。通过推迟类的创建，可以保证所有派生自同一推迟类的普通类拥有相同的继承行为。在 C++中，与推迟类等同的是抽象类。

至于通用类，它提供了一套处理其他数据类型的功能行为。通用类的实例化需要为它的参数指定类型。通用类的作用与 C++中模板类的作用一样。

2、泛化

这里所谓的泛化，主要是通过通用类的实现来获得的。

通用类分两步实现。首先，对一个通用类进行声明以建立模型。在 CDL（CASCADE 定义语言）中，通用类被声明为对不确定类型数据项的操作。这里

的数据项就是通用类中的形参。对通用类的形参进行限制，就可以使形参类型成为普通类的子类。要注意的是：声明一个通用类并没有创建一个新的类类型，而只是定义了一种适用于几个普通类的通用形式。然后，赋予通用类型信息对该通用类进行实例化。通用类被实例化时，它的形参类型由实参类型（基本类型或基本类）替代。实例化结束后，就创建出一个新的类。新类可以由用户在实例化声明中任意命名。按惯例，通用类的实例名通常由通用类名和实参类型名组成。至于通用类的名字，只需在其本身名字前添加一个前缀（该通用类所在包的名字）。

例 2.1：

```
class Array1OfReal instantiates Array1 from TCollection (Real);
```

这个声明位于 TColStd 包的一个 CDL 文件中。它定义了一个新的类 TColStd_Array1OfReal。该类是通用类 TCollection_Array1 的一个实例，并且参数类型指定为实型。

从具有相同形参类型的相同通用类中，可以实例化出不止一个类。这些实例化出来的类通过各自的实现来识别。其实在 C++中，它们已经不属于同一类了。另外，我们不能从通用类中派生类。

在由通用类获得的泛化中，我们经常发现许多类由一个共同的通用类型联系着。这种现象发生在一种基本结构供给迭代器的时候。在这种情况下，有必要弄清一件事，那就是由相关通用类构成的团体的确用于同一对象类型的实例化。为了使这个实例化过程成为一整体，可以将一些通用类声明为内嵌类。这样的通用类就叫内嵌通用类。

一旦主通用类被实例化，它的内嵌类也将被实例化。内嵌类的实例名由内嵌类名字和主通用类名字组成，通过“Of”连接。

例 2.2：

```
class MapOfReal instantiates Map from TCollection (Real,MapRealHasher);
```

这个声明位于 TColStd 中。它不仅定义了 TColStd_MapOfReal 类，也定义了 TColStd_MapIteratorOfMapOfReal 类（该类是通用类 TCollection_Map 的内嵌类 MapIterator 的一个实例）。内嵌类的实例独立于主类的实例，而决非绑定于它。作为内嵌类，即使它们本身不是通用类，但是内嵌于通用类，它们也是通用的。

1.2.2 数据类型的分类

数据类型是作为类被实现的。类不仅定义了它的数据结构和基于它的实例的方法，也声明了该实例的处理方式。

依据处理方式（见图 2.2）的不同， OCC 中所有数据类型可分为两大类：通过句柄（或引用）处理的数据类型和通过值处理的数据类型。一个通过值处理的类型变量包含自己的实例；而一个通过句柄处理的类型变量包含一个实例的引用。

通过值处理的类型首先有基本类型，如布尔类型、字符型、整型、实型等。通过句柄处理的类型变量，如果它不指向任何对象，那我们就说它是空的。要引用一个对象，我们就得用它的一个构造函数实例化该对象，如例 2.3。

例 2.3：

```
Handle(myClass) m = new myClass;
```

在 OCC 中，句柄是一些特殊类，它们以引用的方式对动态存储对象进行安全处理。句柄提供了一种引用计算机制，通过这种机制，当对象不被引用时，可以自动析构对象。

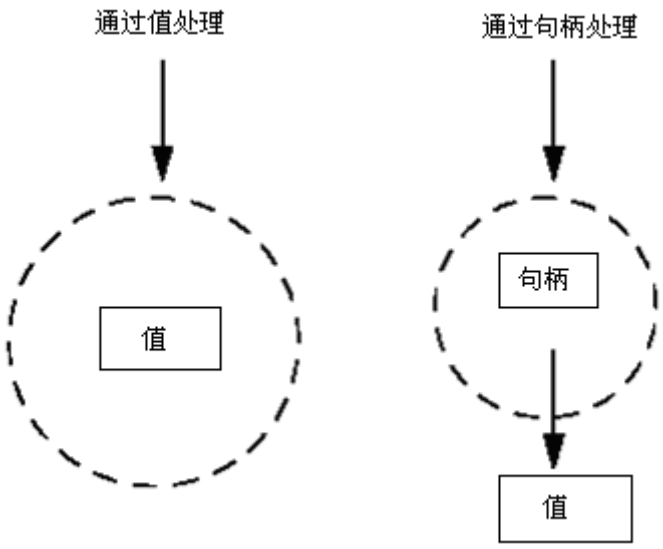


图 2.2 数据类型的两种处理方式

1.2.3 持久化和数据模式

数据模式是应用程序用来存储数据的一种结构，由一些持久类构成。

一个对象若可以被永久存储，则是持久的。持久对象可以被它的创建程序或其它程序在以后的时间里再次使用。

要想使一个对象在 CDL 中是持久的，必须声明它的类型继承自 Standard_Persistent 类或其派生类。所有继承自 Standard_Persistent 类的类都是通过引用处理的。

对于由 Standard_Storable 类派生出的所有类，它们的实例（对象）是不能被

单独存储的，但是可以作为持久对象的区域被存储。所有继承自 Standard_Storable 类的类，其对象都是通过值处理的。

1.3 本章小结

本章用面向对象方法和软件工程思想从整体上分析了 OCC 的体系结构。整个 OCC 包含五个模块；模块中包含工具箱；工具箱中包含包；包中包含类；类是 OCC 软件的最基本要素。

本章还介绍了 OCC 的几个基本概念：类、泛化、数据类型的分类、持久化和数据模式。与 C++类的命名不同，OCC 有自己的命名方法。OCC 中类分为普通类、推迟类和通用类三种，分别对应 C++中的具体类、抽象类和模板类。OCC 的数据可以分为句柄处理类型和值处理类型两种。OCC 的持久化和数据模式与一般软件的原理相同，不同的是：为了使对象持久化，需要声明该对象是由 Standard_Persistent 类或其派生类派生的。

第 2 章　基础类分析

顾名思义，基础类是 OCC 的基石。它提供了大量的通用服务，如自动动态内存管理（通过句柄对对象进行处理）、集合容器、异常处理、通过向下抛掷和创建插件程序而获得的泛化等。因此，本章将先对 OCC 的基础类模块进行概述，然后论述其中几个重点部分，如数据类型、集合容器等。

2.1　基础类概述

基础类包括根类组件、串类组件、集合容器组件、标准对象的集合容器组件、向量和矩阵类组件、基本几何类型组件、常用数学算法组件、异常类组件、数量类组件和应用程序服务组件。

1、根类组件

根类是基本的数据类型和类，其它所有类都是依此而建立的。它提供以下类型和类：

（1）基本类型，如 Boolean（布尔类型）、Character（字符型）、Integer（整型）或者 Real（实型）等。

（2）动态对象的安全处理，以确保那些不再被引用的对象能被及时删除（详见 Standard_Transient 类）。

（3）可设置的内存优化管理器。它能改善那些经常使用动态对象的程序性能。

（4）run-time 类型信息扩展机制。它使得复杂程序的创建变得更为简易。

（5）异常管理。

（6）C++各种流的封装。

根类主要在 Standard 和 MMgt 两个包中实现。

2、串类组件

串类用来处理动态大小的 ASCII 和 Unicode 字符序列，可以通过句柄处理，因此也可以被共享。串类在 TCollection 包中实现。

3、集合容器组件

集合容器是处理动态大小的数据集合的类。集合容器是通用的，即每一种集合容器定义了一种结构和一些算法，可持有许多对象——通常这些对象不必从根类继承。这与 C++模板相似。如果需要使用一个给定对象类型的集合容器，则必须对这个元素的指定类型进行实例化。一旦这个实例声明被编译，所有基于这个

通用集合容器的函数都可以在集合容器对象中实现。

集合容器包含许多通用类，如 run-time 大小的数组、列表、栈、队列、集（Set）和散列图（hash map）。

集合容器在 TCollection 和 NCollection 包中实现。

4、标准对象的集合容器组件

TColStd 包为 TCollection 包中通用类的一些经常使用的实例化提供对象（来自 Standard 包）或者串（来自 TCollection 包）。

5、向量和矩阵类组件

向量和矩阵类提供了有关向量和矩阵的常用数学算法和基本运算（加、乘、转置、求逆等）。

6、基本几何类型组件

基本几何类型提供了基本几何实体和代数实体的实现。这些实现符合 STEP（Standard Exchange of Product data model,即产品数据模型的交换标准）。它们提供基本几何 Shape 的描述（点、向量、直线、圆与圆锥、平面与基本曲面、通过坐标轴或坐标系使 Shape 在平面上或空间中定位）和 Shape 几何变换的定义与应用（平移、旋转、对称、缩放、复合变换、代数计算工具）。

7、常用数学算法组件

常用数学算法为那些经常使用的数学算法提供 C++实现。这些算法有：

（1）求解线性代数方程组的算法 ；

（2）求一元或多元函数最小值的算法；

（3）求解非线性方程或非线性方程组的算法；

（4）求矩阵特征值和特征向量的算法。

8、异常类组件

OCC 提供了一套异常类。所有异常类都是基于它们的根类—— Failure 类的。异常类描述了函数运行期间可能发生的异常情况。发生异常时，程序将不能正常运行。对这种情况的响应称为异常处理。

9、数量类组件

数量类为日期和时间信息提供支持，同时也为表示常用物理量的基本类型（如长度、面积、体积、质量、密度、重量、温度和压力等）提供支持。

10、应用服务组件

应用服务组件包括几种低级服务的实现。借助 OCC 平台，这些服务可以使

那些允许用户自定义和用户友好的应用程序的创建变得更容易。以下是该组件提供的四种服务：

（1）单位转换工具。它们为各种量和相应物理单位的处理提供统一机制。这种机制能够检查单位的兼容性，以及在两种不同的单位间进行数值转换等（详见 UnitsAPI 包）。

（2）有关表达的基本解释器。它使得用户脚本工具的建立和表达的通用定义等变得更容易（详见 ExprIntrp 包）。

（3）用于处理配置资源文件（见 Resource 包）和用户定制信息文件（见 Message 包）的工具。有了这些工具，为应用程序提供多语言支持就很容易了。

（4）进程提示和用户中断接口。它们甚至可能为低级算法提供一种综合便利的用户交流方式。

2.2 数据类型、句柄、内存管理器和异常类

一个软件首先要规定能处理的数据类型，其次要实现三项最基本的功能——引用管理、内存管理和异常管理。在 OCC 中，这三项功能分别对应基础类中的句柄、内存管理器和异常类。

2.2.1 数据类型

在第二章的基本概念里，已经介绍了 OCC 数据类型的分类，即依据处理方式的不同，分为值处理类型和句柄处理类型两类。在详细描述这两种类型之前，先描述 OCC 数据的基本类型。

1、基本类型

所有基本类型都是用 CDL 预定义的，并且只能通过值处理。依据不同的出自，或者依据各自的存储性能，它们可分为耐存的和非耐存的。耐存的基本类型由 Standard_Storable 类派生，能应用于持久对象的实现（要么包含于持久对象方法声明的实体中，要么作为持久对象内部结构的一部分）。

表 3.1 给出 OCC 提供的所有基本类型和相应的 C++基本类型。

表 3.1 OCC 基本类型和相应的 C++基本类型

OCC 基本类型	C++基本类型
Standard_Integer	int

Standard_Real	double
Standard_ShortReal	float
Standard_Boolean	unsigned int
Standard_False=0	
Standard_True=1	
Standard_Character	char
Standard_ExtCharacter	short
Standard_Cstring	char*（指针类型）
Standard_Address	void*（指针类型）
Standard_Extstring	short*（指针类型）

下面分别论述表中的类型。

（1）Standard_Integer（整型）。它是由 32 位二进制数表示的基本类型，包括正数、负数和零。Integer 类型与 C++ int 类型一样。因此，可以对 Integer 类型进行+、 -、 *、/四种代数运算，也可以对其进行<、<=、 ==、 !=、 >=、>六种关系运算。

（2）Standard_Real（实型）。它表示具有确定精度和确定范围的实数的基本类型。Real 类型与 C++ 中 double（双精度）类型一样。因此，+、 -、 *、/四种代数运算、-取反运算和<、<=、 ==、 !=、 >=、>六种关系运算同样适用于 Real 类型。

（3）Standard_ShortReal（短实型）。它表示具有确定精度和确定范围的实数。ShortReal 类型与 C++中 float 类型一样。因此，+、 -、 *、/四种代数运算、-取反运算和<、<=、 ==、 !=、 >=、>六种关系运算同样适用于 ShortReal 类型。

（4）Standard_Boolean（布尔类型）。 它是描述逻辑值的基本类型。它有两种值：false 和 true。Boolean 类型与 C++中 unsigned int 类型一样。因此，与、或、异或、非四种代数运算和==、!=两种关系运算同样适用于 Boolean 类型。

（5）Standard_Character（字符类型）。它是用来表示 ASCII 字符集的一种基本类型。它能被赋予的值有 128 个，对应 128 个 ASCII 字符。Character 类型与 C++ 中 char 类型一样。因此，<、<=、 ==、 !=、 >=、>六种关系运算同样适

用于 Character 类型（如：A<B）。

（6）Standard_ExtCharacter（扩展字符类型）。它是用来表示 Unicode 字符集的一种基本类型。由它表示的字符得用 16 位二进制数进行编码。ExtCharacter 类型与 C++ 中 short 类型一样。因此，<、<=、 ==、 !=、 >=、>六种关系运算同样适用于 ExtCharacter 类型（如：A<B）。

（7）Standard_CString（C 串类型）。它用来表示文字串。一个文字串就是由双引号括起来的一个 ASCII 字符序列。CString 类型与 C++中 char*类型是一样的。

（8）Standard_Address（地址类型）。它用来表示一个通用指针。Address 类型与 C++中 void*类型一样。

（9）Standard_ExtString（扩展串类型）。 它用来表示由 Unicode 字符序列构成的文字串。ExtString 类型与 C++中 short*类型一样。

2、值处理类型

值处理类型可分三大类：

（1）基本类型；

（2）枚举类型；

（3）由这样一些类（既不是由 Standard_Persistent 类派生，也不是由 Standard_Transient 类派生，无论是直接派生还是间接派生）定义的类型。

值处理类型的表现形式比句柄处理类型的表现形式更直接。因此，对值处理类型的操作也会更快。但是值处理类型对象不能单独存于文件中。图 3.1 表示了对一个值处理类型对象的处理过程。

需要注意的是：那些能被数据模式识别（包括基本类型和从 Storable 类继承过来的类型）的值处理类型可以作为持久对象的部分结构而存储在持久对象内部。这是值处理类型对象能够存进文件的唯一方式。

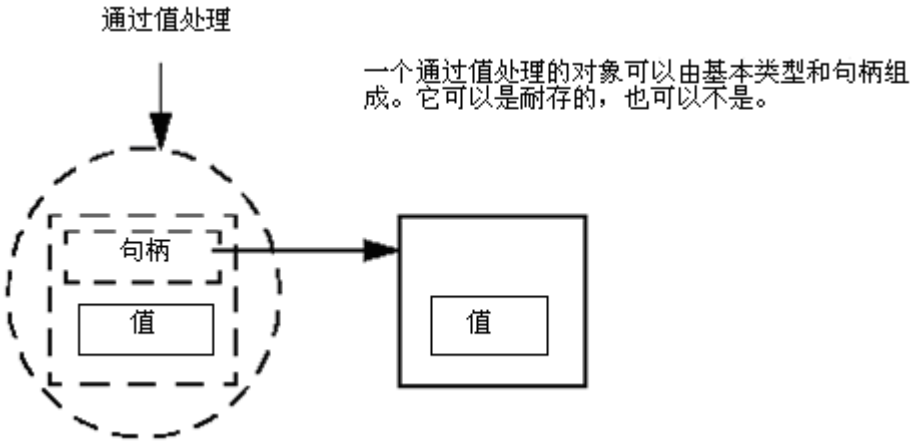


图 3.1 值处理类型对象的处理处理过程

3、句柄处理类型

句柄处理类型可以分为两大类：

（1）由 Persistent 类的派生类定义的类型。这些类型可以被长久地存在文件中。

（2）由 Transient 类的派生类定义的类型。

图 3.2 表示了对一个句柄处理类型对象的处理过程。

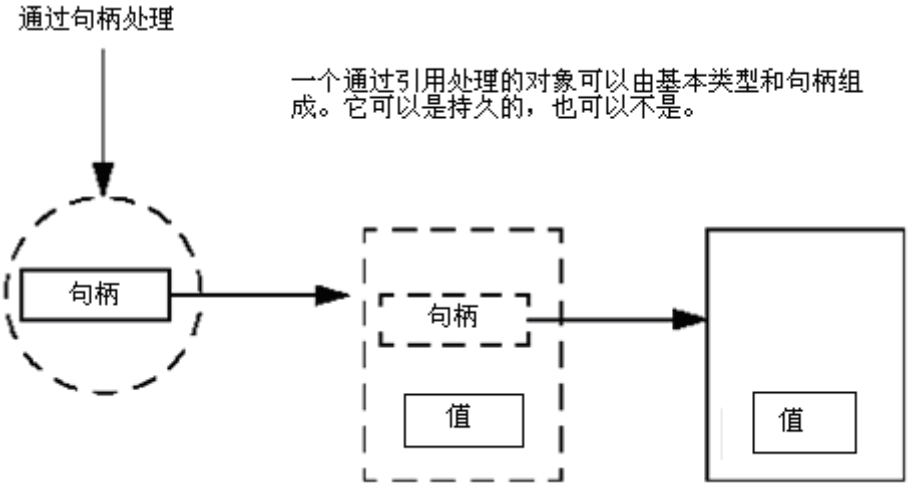


图 3.2 句柄处理类型对象的处理过程

4、特征总结

为了方便比较值处理类型和句柄处理类型的异同，也为了清楚地认识 OCC 各种数据类型的处理方式和各自的耐存性，下面给出图 3.3。

通过句柄处理的数据类型		通过值处理的数据类型
耐存的基本类型	持久类型	基本类型和耐存类型（如果内嵌在一个持久类中，则为耐存的）

非耐存的基本类型	短暂类型	其它类型

图 3.3 值处理类型和句柄处理类型的异同

2.2.2 句柄

1、句柄的定义

OCC 的引用管理采用的是一种句柄机制。这种机制的基本元素是句柄。

在 OCC 中，句柄是通过类实现的。句柄含有多个接口成员，其中一个包含一个引用。一般情况下，仅需要使用它的引用。正因为这样，习惯将句柄比作 C++指针。与 C++指针一样，几个句柄可以引用同一个对象；一个句柄也可以引用多个对象，但是每次只能引用一个；在句柄访问对象前，句柄必须被声明。

2、句柄处理类的组织

一般情况下，真正需要的是句柄引用的对象而非引用本身。在此，有必要介绍一下句柄处理类的组织。句柄处理类要么是持久的，要么是短暂的。如果由 Standard_Transient 类派生，则是短暂的；如果由 Standard _Persistent 类派生，则是持久的。不论短暂还是持久，它们的组织情况是一样的。故，下面一段文字将仅介绍短暂句柄处理类及其相关句柄的组织情况。

Standard_Transient 类是 OCC 中所有句柄处理类的一个根类（另一个根类是 Standard_Persistent 类）。它提供了一个引用计数器（被其所有后裔类继承）。该计数器被 Handle()类（也就是所谓的句柄）使用，用于计算指向对象实例的句柄数。对于每一个继承（直接或间接）自 Transient 类的类，CDL 提取器都创建了相应的 Handle()类（句柄）。该 Handle()类（句柄）的参数名字和由“Handle_”作前缀修饰的名字一样。OCC 专门提供了一个宏预处理函数 Handle()。它能够将一个 Handle()类（句柄）的参数名字提取出来作为指定短暂类的名字。

这里提三个注意事项：

（1）Transient 类和 Persistent 类不完全是通过句柄处理的，它们也可以通过值处理；

（2）持久对象不能含有非耐存句柄（那些引用非持久对象的句柄）；

（3）使用句柄的目的是共享对象（对于所有局部操作，建议使用值处理类）。

3、句柄的使用

句柄通过它引用的对象被特征化。在对一个短暂对象进行任何操作之前，必须对句柄进行声明。比如， Point 和 Line 是来自 Geom 包的两个短暂类，声明

得像例 3.1 这样写。

例 3.1：

```
Handle(Geom_Point) p1, p2;
Handle(Geom_Line) aLine;
```

对一个句柄进行声明，只是创建了一个空句柄，该句柄不指向任何对象。要初始化句柄，要么得创建一个新的对象，要么得将其它句柄值赋予该句柄（假定两种句柄类型是兼容的）。

4、句柄的类型管理

首先介绍句柄的通用管理。

OCC 能以通用方式对数据类型进行描述。这样，可以在程序运行时才去严格核对给定对象的类型。这与 C++ RTTI（运行时类型信息机制）类似。对于每一种由 Standard_Transient 类派生的类类型，CDL 提取器都创建了相应的代码段，用于对 Standard_Type 类进行实例化。通常 Standard _Type 类（也称类型描述器）持有类型信息：类型名和其祖先类型列表。

类型的实例（实际上是指向该类型的句柄）由虚函数 DynamicType()（该虚函数在 Standard_Transient 类的派生类中）返回。检查给定对象是否具有给定类型或给定类型的后裔类型，需调用另一个虚函数 IsKind()。

为给定类类型寻找相关的类型描述器，得用相关的宏 STANDARD_TYPE()。其中的宏参数就是给定类的名字。

接着介绍句柄的类型一致原则。

句柄声明中的对象类型是对象的静态类型，它能被编译器识别。句柄能够引用静态类型的子类对象。因此，对象的动态类型（也称对象的实际类型）可以是静态类型的后裔类型。这就是句柄的类型一致原则。

考虑到持久类 CartesianPoint 是 Point 类的一个子类，所以，类型一致原则可以用例 3.2 表示。

例 3.2：

```
Handle (Geom_Point) p1;
Handle (Geom_CartesianPoint) p2;
p2 = new Geom_CartesianPoint;
p1 = p2;  //可以，类型是兼容的。
```

例中，编译器将 p1 看做是指向 Point 类的句柄，尽管 p1 实际指向 CartesianPoint 类型对象。

最后介绍句柄的直接类型转换。

依据类型一致原则，我们总可以将低层句柄向上赋值给高层句柄。但是，反过来则不行。因此，我们需要一种直接类型转换机制。

如果一个句柄所指对象的实际类型是抛掷者（抛掷句柄的对象）的后裔类型，那么，该句柄可以直接向其子类型转换。这就是句柄的直接类型转换（见例 3.3）。

例 3.3：

```
Handle (Geom_Point) p1;
Handle (Geom_CartesianPoint) p2, p3;
p2 = new Geom_CartesianPoint;
p1 = p2;   //可以，标准的赋值。
p3 = Handle (Geom_CartesianPoint) : : DownCast (p1);
// 可以，p1 的实际类型是 CartesianPoint 句柄，尽管它的静态类型是 Point 句柄。
```

如果直接转换与句柄所指对象的实际类型不兼容，那么被抛掷的句柄会被清空，并且不会产生任何异常。因此，如果需要一些能在静态类型的子类型中实现的可靠服务，如例 3.4 编写程序：

例 3.4：

```
void MyFunction (const Handle(A) & a)
{
    Handle (B) b = Handle (B):: Downcast(a);
    if (! b.IsNull())
       {
          //如果 B 类由 A 类派生，我们就可以使用“b”。
       }
    else
       {
          // 类型不兼容。
       }
}
```

向下抛掷尤其被用于处理由不同类型对象组成的集合容器，但是有一个限制条件——这些对象必须继承自同一个根类。例如，有一个由多个短暂对象构成的 SequenceOfTransient 序列，同时还有两个继承自 Standard_T- ransient 类的类，那么例 3.5 的构造语句是有效的。

例 3.5：

```
Handle (A) a;
Handle (B) b;
Handle (Standard_Transient) t;
SequenceOfTransient s;
a = new A;
s.Append (a);
b = new B;
```

```
s.Append (b);
t = s.Value (1);
//这里我们不能这样写：
a = t;
//这是错误的。
// 因此我们向下抛掷：
a = Handle (A) :: Downcast (t)
if (! a.IsNull())
   {
      //类型兼容的话，我们就可以使用 “a”。
   }
else
   {
      // 类型不兼容。
   }
```

5、用句柄创建对象

要创建一个由句柄处理的对象，得对句柄进行声明，并用 C++ new 操作符初始化句柄。这样的声明和初始化必须被构造函数的调用紧跟着，如例 3.6 所示。

例 3.6：

```
Handle (Geom_CartesianPoint) p;
p = new Geom_CartesianPoint (0, 0, 0);
```

与 C++指针不同，句柄并不支持 delete 函数。当句柄所指对象不再被使用时，该对象会被自动析构。

6、通过句柄调用对象方法

对于指向持久对象或短暂对象的句柄，可以像使用 C++指针那样使用它。通过句柄，可以调用对象方法。

调用对象方法有两种方式：一是使用操作符“->”；二是使用函数调用语句。但是当需要调用方法来测试或者修改句柄状态时，则必须用操作符“.”。

例 3.7 说明了怎样获取点的坐标。

例 3.7：

```
Handle (Geom_CartesianPoint) centre;
Standard_Real x, y, z;
if (centre.IsNull())
   {
      centre = new PGeom_CartesianPoint (0, 0, 0);
   }
centre->Coord(x, y, z);
```

例 3.8 说明了怎样获取一个笛卡尔点的类型。

例 3.8：

```
Handle(Standard_Transient) p = new Geom_CartesianPoint(0.,0.,0.);
if ( p->DynamicType() == STANDARD_TYPE(Geom_CartesianPoint) )
      cout << "Type check OK" << endl;
else
      cout << "Type check FAILED" << endl;
```

需要注意的是：如果指向对象方法或对象定义域的句柄是空的，那么将产生 NullObject 异常。

7、句柄的存储分配

在删除对象前，必须确保对象没有被引用。

为了减少与对象生存管理有关的编程，在每个句柄处理类里都含有一个删除函数。句柄能使引用计数管理自动进行，并且当对象不再被引用时，自动析构对象。通常，不能对 Standard_Transient 类的子类实例直接调用 delete 函数。

当对象有一个新句柄时，引用计数器的值加 1。当一个句柄被删除，或被清空，或被重新赋值而指向另一对象时，计数器的值减 1。一旦引用计数器的值为 0 时，对象将被自动析构。

句柄的分配规则可以通过例 3.9 体现出来。

例 3.9：

```
...
{
  Handle (TColStd_HSequenceOfInteger) H1 = new TColStd_HSequenceOfInteger;
  // H1  有一个引用，对应的内存空间是 48 字节。
  {
      Handle (TColStd_HSequenceOfInteger) H2;
      H2 = H1;
      // H1  有两个引用。
      if (argc == 3)
        {
          Handle (TColStd_HSequenceOfInteger) H3;
          H3 = H1;
          // H1 有三个引用。
          ...
        }
      // H1  有两个引用。
   }
  // H1  有一个引用。
}
// H1 没有引用。
// TColStd_HSequenceOfInteger  对象被析构。
```

如果两个或多个对象通过句柄（作为定义域）互相引用，就会出现循环。在这种情况下，对象不会被自动析构。

以图表（graph）为例，它的元素对象得知道它们所属的图表对象，也就是说，元素得有一个对图表对象的引用。如果元素和图表都是通过句柄处理的，并且都以句柄作为自己的定义域，则将出现循环。当最后一个句柄被删除时，图标对象不会被删除。这是因为图表里面还有许多指向该图表的句柄——这些句柄作为图表的数据结构（元素）被存储。

有两种方式可以避免循环的出现：

（1）用 C++指针代替每一个引用（比如由元素对图表的引用）。

（2）当图表对象需要被删除时，将整套句柄（如元素中指向图表的句柄）清空。

2.2.3 内存管理器

1、使用内存管理器的原因

标准的内存分配有三种方式：静态分配、栈分配和堆分配。静态分配是最简单的内存分配策略。程序中的所有名字在编译时绑定在某个存储位置上；这些绑定不会在运行时改变。块结构语言通过在栈上分配内存,克服了静态分配的一些限制。每次过程调用时,一个活动记录或是帧被压入系统栈,并在返回时弹出。堆分配与栈所遵循的后进先出的规律不同,堆中的数据结构能够以任意次序分配与释放[4]。

建模程序在运行期间，需要构造和析构大量的动态对象。在这种情况下，标准的内存分配函数可能无法胜任工作。因此，OCC 采用了特殊的内存管理器（在 Standard 包中实现）。

2、内存管理器的用法

在 C 代码中使用 OCC 内存管理器分配内存，只需用 Standard::Allocate()方法代替 malloc()函数，Standard::Free()方法代替 free()函数，以及 Standard::Reallocate()代替 realloc()函数。

在 C++中，可以将类的 new()操作定义为使用 Standard::Allocate()方法进行内存分配，而将类的 delete()操作定义为使用 Standard::Allocate()方法进行内存释放。这样就可以使用 OCC 内存管理器为所有对象和所有类分配内存。就是用这种方式，CDL 提取器为所有用 CDL 声明的类定义了 new()函数和 delete()函数。因此，除了异常类，所有 OCC 类都使用 OCC 内存管理器进行存储分配。

因为 new()函数和 delete()函数是被继承的，所以对于所有 OCC 类的派生类（比如，Standard_Transient 类的派生类），new()函数和 delete()函数同样适用。

3、内存管理器的配置

OCC 内存管理器可以适用于不同的内存优化技术（不同的内存块采用不同的优化技术，这主要依据内存块的大小而定）。或者，用 OCC 内存管理器，甚至可以不采用任何优化技术而直接使用 C 函数 malloc() 和 free()。

内存管理器的配置由下面几个环境变量值定义：

（1）MMGT_OPT。如果值设为 1（默认值），则内存管理器将如下面的描述那样对内存进行优化。如果值设为 0，则每个内存块将直接分配（通过 malloc()和 free()函数）在 C 内存堆里。在第二种情况下，所有异常（不包括 MMGT_CLEAR 异常）都将被忽略。

（2）MMGT_CLEAR。如果值设为 1（默认值），则每一个已分配的内存块都将被清零。如果值设为 0，则内存块正常返回。

（3）MMGT_CELLSIZE。它定义了大内存池中内存块的最大空间。默认值是 200 字节。

（4）MMGT_NBPAGES。它定义了页面中由小内存块构成的内存组件（内存池的大小（由操作系统决定）。默认值是 1000 字节。

（5）MMGT_THRESHOLD。它定义了内存块（能直接在 OCC 内部被循环使用）的最大空间。默认值是 40000 字节。

（6）MMGT_MMAP。当值设为 1（默认值）时，使用操作系统的映射函数对大内存块进行分配。当值设为 0 时，大内存块将被 malloc()分配在 C 内存堆里。

（7）MMGT_REENTRANT。当值设为 1 时，所有对内存优化管理器的调用都将被响应，以保证不同的线程能同时访问内存管理器。在多线程程序中，这个变量值应该设置为 1。这里所说的多线程程序是指那些使用 OCC 内存管理器，并且可能有不止一个调用 OCC 函数的线程的程序。默认值是 0。

在此提一个注意事项：当多线程程序使用 OCC 以达到最佳内存优化性能时，需要检查两组变量。其中一组是 MMGT_OPT=0；另一组则是 MMGT_OPT=1 和 MMGT_REENTRANT=1。

4、内存管理器的实现。

当且仅当 MMGT_OPT=1 时，才用到 OCC 的特殊的内存优化技术。这些技术有：

（1）小内存块（空间比由 MMGT_CELLSIZE 设定的值小）不能单独分配，而是分配在大内存池（大小由变量 MMGT_NBPAGES 决定）中。每一个内存块分配在当前内存池的空闲部分。若内存池被完全占据，则使用下一个内存池。在当前版本中，在进程结束前，内存池不能返回操作系统。然而，

那些由 Standard::Free()释放的内存块被记忆在释放列表中。当需要下一个内存块（与列表中某个块大小相同）时，相应的被释放的那个内存块所占空间可以被新的内存块占用，这也叫内存块的循环使用。

（2）对于中等大小的内存块（比 MMGT_CELLSIZE 大，但比 MMGT_THRESHOLD 小），它们是被直接分配（通过使用 malloc() 和 free()）在 C 内存堆里。这些块要是被 Standard::Free()方法释放的话，可以像小内存块那样被循环使用。然而，与小内存块不同，那些被记录在释放列表中的可循环使用的中内存块（由持有内存管理器的程序释放）可以被 Standard::Purge()方法返回到 C 内存堆中。

（3）大内存块（大小比 MMGT_THRESHOLD 大，包括用来分配小内存块的内存池）的分配取决于 MMGT_MMAP 的值。如果该值是 0，则这些大块被分配在 C 堆里。否则，它们被操作系统映射函数分配在内存映射文件中。当 Standard::Free()被调用时，大块立即被返回到操作系统中去。

5、内存管理器的优缺点。

OCC 内存管理器的优点主要体现在小块和中块的循环使用上。当程序需要连续分配和释放大小差不多的内存块时，这个优点能加速程序的执行。实际应用中，这种提升幅度可以高达 50%。

相应的，OCC 内存管理器的主要缺点是：程序运行时，被循环使用的内存块不能返回到操作系统中。这可能导致严重的内存消耗，甚至会被操作系统误认为内存泄露。为了减少这种影响，在频繁地对内存进行操作后，OCC 系统将调用 Standard::Purge()方法。

另外， OCC 内存管理器会带来额外的开销，它们有：

（1）舍入后，每个被分配的内存块的大小高达 8 字节。当 MMGT_OPT=0 时，舍入值由 CRT 决定；对 32 位平台而言，典型值是 4 字节。

（2）在每个内存块的开端需要额外的 4 字节以记录该内存块的大小（或者，当内存块被记录在释放列表时，这 4 字节用来记录下一个内存块的地址）。注意：

只有在 MMGT_OPT=1 时，才需要这 4 字节。

需要注意的是：由 OCC 内存管理器带来的额外开销可能比由 C 内存堆管理器带来的额外开销大，或者小。因此，整体而言，很难说到底是优化模式的内存消耗大还是标准模式的内存消耗大——这得视情况而定。

通常，编程人员自己也会采用一种优化技术——在内存里面划出一些重要的块。这样就可以将一些连续的数据存于这些块中，使内存页面管理器对这些块的处理变得更容易。

在多线程模式（MMGT_REENTRANT=1）中，OCC 内存管理器使用互斥机制以锁定程序对释放列表的访问。因此，当不同的线程经常同时调用内存管理器时，优化模式的性能不如标准模式的性能好。原因是：malloc() 函数和 free()函数在实现的过程中开辟了几个分配空间——这样就避免了由互斥机制带来的延迟。

2.2.4 异常类

1、异常类的定义

异常处理机制实现了正常程序逻辑与错误处理的分离，提高了程序的可阅读性和执行效率。为了转移程序运行的控制流,异常处理的模式通常有无条件转移模式、重试模式、恢复模式和终止模式[5]。与 C++一样，OCC 采用的是终止模式。

为了实现这种异常处理机制，OCC 提供了一套异常类。所有异常类都是基于它们的根类——Failure 类的。异常类描述了函数运行期间可能发生的异常情况。发生异常时，程序将不能正常运行。对这种情况的响应被称为异常处理。

2、异常类的使用

OCC 使用异常的语法与 C++使用异常的语法相似。要产生一个确定类型的异常，需用到相应异常类的 Raise()方法，如例 3.10 所示。

例 3.10：

```
DomainError::Raise(“Cannot cope with this condition”);
```

这样就产生了一个 DomainError 类型的异常，同时伴有相应的提示信息“Cannot cope with this condition”。这信息可以是任意的。

该异常可以被某种 DomainError 类型（DomainError 类型派生了好些类型）的句柄器捕获，如例 3.11 所示。

例 3.11：

```
try
```

```
    {
        OCC_CATCH_SIGNALS
        // try 块。
    }
catch (DomainError)
    {
        // 处理 DomainError 异常。
    }
```

不能把异常类的使用当作一种编程技巧，例如用异常类代替“goto”。应该把异常类的使用作为方法的一种保护方式（防止被错误使用），即保证方法调用者遇到的问题是方法能处理的。故，在程序正常运行期间，不该产生任何异常。

在使用异常类的时候，需要用一个方法来保护另外一个可能出现异常的方法。这样能通过外层方法来检查内层方法的调用是否有效。例如需要用三个方法（用于检查元素的 Value 函数、用于检查数组下边界的 Lower 函数和用于检查数组上边界的 Upper 函数）使用 TCollection_ Array1 类，那么，Value 函数可以如例 3.12 那样被实现：

例 3.12：

```
Item TCollection_Array1::Value (const Standard_Integer&index) const
{
    // 下面的 r1 和 r2 是数组的上下边界。
    if(index < r1 || index > r2)
        {
            OutOfRange::Raise(“Index out of range in Array1::Value”);
        }
    return contents[index];
}
```

在此，OutOfRange::Raise(“Index out of range in Array1::Value”)异常用 Lower 函数和 Upper 函数检查索引是否有效，以保护 Value 函数的调用。

一般地，在 Value()函数调用前，程序员已确定索引在有效区间内了。这样，上面 Value()函数的实现就不是最优的了，因为检查既费时又冗余。

在软件开发中有这样一种广泛的应用方式，即将一些保护措施置于反汇编构件而非优化构件中。为了支持这种应用，OCC 为每一个异常类提供了相应的宏 Raise_if()：

```
<ErrorTypeName>_Raise_if(condition, “Error message”)
```

这里 ErrorTypeName 是异常类型，condition 是产生异常的逻辑表达式，

而 Error message 则是相关的错误信息。

可以在编译的时候，通过 No_Exception 或者 No_<ErrorTypeName>两种预处理声明之一解除异常的调用，如例 3.13 所示：

例 3.13：

```
#define No_Exception /*解除所有的异常调用*/
```

使用这构造语句，Value 函数变为：

例 3.14：

```
Item TCollection_Array1::Value (const Standard_Integer&index) const
{
    OutOfRange_Raise_if(index < r1 || index > r2,
                        “index out of range in Array1::Value”);
    return contents[index];
}
```

3、异常处理

异常发生时，控制点将转移到调用堆栈中离当前执行点最近的指定类型的句柄器上。该句柄器具有如下特征：

（1）它的 try 块刚刚被进入还没有被退出；

（2）它的类型与异常类型匹配。

（3）T 类型异常句柄器与 E 类型异常匹配，即 T 类型和 E 类型相同，或者 T 类型是 E 类型的超类型。

OCC 的异常处理机制还可以将系统信号当作异常处理。为此，需要在相关代码的开端嵌入宏 OCC_CATCH_SIGNALS。建议将这个宏放在 try {}块中的第一位置。例如，有这样四个异常： NumericError 类型异常、Overflow 类型异常、Underflow 类型异常和 ZeroDivide 类型异常，其中 NumericError 类型是其它三种类型的超类型，那么，异常处理过程如例 3.15 所示。

例 3.15：

```
void f(1)
    {
        try
            {
                OCC_CATCH_SIGNALS
                // try 块
            }
        catch(Standard_Overflow)
            {   // 第一个句柄器
                // ...
            }
        catch(Standard_NumericError)
```

```
        {   //第二个句柄器
            // ...
        }
    }
```

在这个例子中，第一个句柄器将捕获 Overflow 类型异常；第二个句柄器将捕获 NumericError 类型异常及其派生异常，包括 Underflow 类型异常和 Zerodivide 类型异常。异常发生时，系统将从最近的 try 块到最远的 try 块逐一检查句柄器，直到找到一个在形式上与产生的异常相匹配的为止。

在 try 块中，如果将基类异常的句柄器置于派生类异常的句柄器之前，则将发生错误。因为那样会导致后者永远不会被调用，如例 3.16 所示。

例 3.16：

```
void f(1)
    {
        int i = 0;
        try
            {
                OCC_CATCH_SIGNALS
                g(i);
                // i 是可接受的。
            }
        // 在这放执行语句会导致编译错误!
        catch(Standard_NumericError)
            {
                // 依据 i 值处理异常。
            }
                // 在这放执行语句可能导致不可预料的影响。
    }
```

由异常类形成的树状体系与用户定义的类完全无关。该体系的根类是 Failure 异常。因此，Failure 异常句柄器可以捕获任何 OCC 异常。建议将 Failure 异常句柄器设置在主路径中，如例 3.17 所示。

例 3.17：

```
#include <Standard_ErrorHandler.hxx>
#include <Standard_Failure.hxx>
#include <iostream.h>

int main (int argc, char* argv[ ])
  {
     Try
        {
            OCC_CATCH_SIGNALS
```

```
            //主块
            return 0;
        }
    catch(Standard_Failure)
        {
            Handle(Standard_Failure) error = Failure::Caught ();
            cout << error << end1;
            return 1;
        }
}
```

这里的 Caught 函数是 Failure 类的一个静态成员，能返回一个含有异常错误信息的异常对象。这种接收对象的方法(通过 catch 的参数接收异常)代替了通常的 C++语句。

尽管标准的 C++处理法则和语法在 try 块和句柄器中同样适用，但在一些平台上，OCC 能以一种兼容模式被编译（此时异常支持长转移）。在这种模式中，要求句柄器前后没有执行语句。因此，强烈建议将 try 块置于{}中。此外，这种模式也要求 Standard_ErrorHandler.hxx 头文件包含在程序中（置于 try 块前），否则将不能处理 OCC 异常。再有，catch()语句不允许将一个异常对象作为参数来传递。

为了使程序能够像捕获其它异常那样捕获系统信号（如除零），在程序运行时要使用 OSD::SetSignal()方法安装相应的信号句柄器。通常，该方法在主函数开端处被调用。

为了能真正的将系统信号转换成 OCC 异常，OCC_CATCH_ SIGNALS 宏应该被嵌入到源代码中。典型的，将该宏置于捕获异常的 try{}块的开端处。

OCC 的异常处理机制依据不同的宏预处理 NO_CXX_EXCE- PTIONS 和 OCC_CONVERT_SIGNALS 有不同的实现。这些预处理将被 OCC 或者用户程序的编译程序连贯定义。在 Windows 和 DEC 平台上，这些宏不是以默认值被定义的，并且所有类都支持 C++异常，包括从句柄器中抛掷异常。因此，异常的处理与 C++异常处理一样。

2.3 集合容器和标准对象的集合容器

2.3.1 集合容器

1、概述

集合容器组件包含一些具有动态大小的数据集合类，如数组类、列表类和图

（map）类等。集合容器类是通用的，即它们可以持有许多不必从根类继承的对象。当需要使用一个给定对象类型的集合容器时，必须指定集合容器的元素类型。一旦这个声明被编译，所有适用于这个通用集合容器的函数，同样适用于这个集合容器实例。

然而，需要注意两点：

（1）在 OCC public 构造语句中，被当作参数直接使用的集合容器是在一个 OCC 组件中实例化的。

（2）TColStd 包为这些通用集合容器提供许多实例化；实例对象来自 Standard 包或者串类组件。

集合容器组件提供了一些通用集合容器：

（1）数组。通常用于快速访问项目。但是数组的大小是固定的（大小一旦被声明，将不能更改）。

（2）序列。其大小可变。使用序列可以避免使用大数组和类空数组。但是序列的项目通常比数组的项目长，故只能采用特殊的方法访问序列。另外，许多访问方法不适用于序列。数组和序列通常被用作复杂对象的一种结构。

（3）图（Map）。与序列不同，图的大小可变，而且对图的访问也快。图结构通常可以有多种访问方法。图通常作为复杂算法中的内部数据结构。集（Set）具有同样的用途，但是访问时间要长的多。

（4）列表、队列和栈。它们的结构与序列的结构相似。但是它们的算法却与序列的算法不同。

大部分集合容器遵循语意值，即一个集合容器实例就是实际的集合容器，而不是指向某个集合容器的句柄集合。只有数组和序列才可以通过句柄处理并被共享。

OCC 的集合容器有通用集合、通用图和迭代器。它们都在 TCollection 包中。

2、通用集合

通用集合有 Array1 、Array2 、HArray1 、HArray2 、Sequence 、HSequence 、List 、Queue 、Stack、Set 和 HSet。

TCollection_Array1 数组与 C 数组相似，即大小可以由用户在构造时指定，构造后不能更改。与 C 数组一样，访问 Array1 成员的时间是一个常量，而与数组的大小无关。Array1 数组通常被用作复杂对象的基本结构。Array1 数组是一个通用类，它的实际类型取决于它的项目，即数组元素的类型。Array1 的下标区

域由用户定义。因此，要访问 Array1 项目，必须保证下标在规定的区域内。

TCollection_Array2 是二维数组，它的情况与 TCollection_Array1 的类似。

TCollection_HArray1 数组与 TCollection_Array1 数组相似，不同的是 HArray1 的对象是指向数组的句柄数组。HArray1 数组可以被几个对象共享；可以用 TCollection_Array1 数组结构作为 HArray1 的实际结构。HArray1 是一个通用类，它的实际类型取决于两个参数：项目（数组的元素类型）和数组（通过 HArray1 处理的数组的实际类型，具有 TCollection_Array1 项目类型的一个实例）。

TCollection_HArray2 数组与 TCollection_HArray1 数组类似，只不过是二维的。

TCollection_Sequence 是通过整数来索引的序列。序列的用途与一维数组(TCollection_Array1)的一样，都是作为复杂对象的基本结构。但是序列的大小是可变的。使用序列可以避免使用大数组和类空数组。访问序列项目需要专门的方法，否则会比访问数组项目慢。另外也要注意，当需要支持多种访问方法时（图会更适合），序列不是一种有效的结构。序列是一个通用类，它的实际类型取决于项目，即它的元素类型。

TCollection_HSequence 序列 TCollection_Sequence 序列类似，不同的是 HSequence 对象是指向序列的句柄序列。HSequence 序列可以被几个对象共享；可以用 TCollection_Sequence 结构作为 HSequence 的实际结构。HSequence 是一个通用类，它的实际类型取决于两个参数：项目（序列元素的类型）和序列（通过 HSequence 处理的序列的实际类型，一个具有 TCollection_Sequence 项目类型的实例）。

TCollection_List 是允许项目重复的有序列表。列表可以使用迭代器（ListIterator 迭代器）进行线性迭代。可以快速地将一个项目插入列表的任何位置。但是，如果列表很长，那么通过值查找项目将会很慢，因为那样需要逐一查找。故当项目需要通过值来查找时，用序列结构会更好些。列表是一个通用类，它的实际类型取决于它的项目，即列表元素的类型。在列表被实例化时，系统将自动产生一个 TCollection_ListIterator 类的实例（迭代器）。

队列和栈也是列表的一种，但两者的数据访问方式不同。

对于队列（TCollection_Queue ），只能在其尾部插入元素，在其头部删除元素。这样，最先进入队列的元素将最先被清除。所以队列也叫先进先出列表。队列是一个通用类，它的实际类型取决于它的项目，即队列元素的类型。

对于栈（TCollection_Stack ），只允许在其顶部插入或删除项目。最后进栈的项目将最先出栈。所以栈也叫后进先出列表。栈是一个通用类，它的实际类型取决于它的项目，即它的元素类型。访问栈需要使用 StackIterator 迭代器。

TCollection_Set 集是一个项目无序且相异的集合容器。对集进行操作时，系统会检查其中是否含有相同的项目。集的效果和图的一样，但是图结构更有效。所以建议使用图来代替集。集是一个通用类，它的实际类型取决于它的项目，即它的元素类型。访问集，需要 SetIterator 迭代器。

TCollection_HSet 与 TCollection_Set 类似，不过 HSet 对象是指向集的句柄集合。HSet 是一个通用类，它的实际类型取决于两个参数：项目（Set 中元素的类型）和 Set（通过 HSet 处理的集的实际类型，一个具有 TCollection_Set 项目类型的实例）。

3、通用图

通用图有： TCollection_Map 、 TCollection_DataMap 、 TCollection_DoubleMap、TCollection_IndexedMap 和 TCollection_IndexedDataMap。这些图可以自动管理桶的数目：当键数超过桶数时，能重置桶数。如果知道图中项目的确切数目，就可以在构造的时候直接将这个值定义为图的初始大小，或者通过重设函数来设定。这是一种很好的内存优化方式。通用图有三个参数：键、项目和散列器。每一个实例自集合容器组件的图都要用到一个散列器对象。散列器对象所需函数来自 TCollection_MapHasher 散列器。如果一个图是通过迭代器来访问的，那么在图被实例化的同时，系统自动产生一个迭代器实例。注意：系统提供的通用图中，有些不是通过迭代器而是通过索引来访问的。

TCollection_Map 图是一个基本的散列图，用来线性存储和线性查找键。它的入口仅由键组成。它没有与键对应的数据。Map 通常被某个算法用来判断某个复杂结构的某个动作是否依然进行着。Map 是一个通用类，其实际类型取决于两个参数：键（键的类型）和散列器（基于这些键的散列器类型）。对 Map 进行访问，需要使用 MapIterator 迭代器。

TCollection_DataMap 图用来存储键和相应的项目。它的入口由键和项目组成。它可以看作是一种扩展数组，其中的键就是索引。它是一种通用类，实际类型取决于三个参数：键（入口键的类型）、项目（与键对应的元素的类型）和散列器（基于键的散列器类型）。其中散列器对象需要的函数由 TCollection_MapHasher 类描述。访问 DataMap 图需要使用 DataMapIterator 迭代器。

该迭代器（TCollection_DataMapIterator 通用类的一个实例）由系统在 DataMap 图的实例化过程中自动产生。

TCollection_DoubleMap 图用来将两组键进行绑定，并且可以对键进行线性访问。Key1 作为它的第一组键，而 Key2 作为第二组键。它的每一个入口由一对键组成，即第一个键和第二个键。它是一个通用类，实际类型取决于四个参数：Key1（每一个入口中第一个键的类型）、Key2（每一个入口中第二个键的类型）、Hasher1（基于第一组键的散列器类型）和 Hasher2（基于第二组键的散列器类型）。其中 Hasher1 和 Hasher2 需要的函数由 TCollection_ MapHasher 类描述。访问 DoubleMap 图需要使用 DoubleMapIterator 迭代器。该迭代器（TCollection_DoubleMapIterator 通用类的一个实例）由系统在 DoubleMap 图的实例化过程中自动产生。

TCollection_IndexedMap 图用来存储键，并将每一个键和一个索引进行绑定。每存入一个新键，都会给这个新建分配一个索引。索引随着键的增加而增加。键可以通过索引找到，而索引也可以通过键查找。只有最后一个键可以被删除。因此，索引在 1 到 Upper（map 中键的数目）之间取值。IndexedMap 图的每个入口都由键和索引组成。IndexedMap 图是有序的，允许线性迭代。但是它没有与键对应的数据。它通常被某个算法用来判断某个复杂结构上的动作是否依然进行着。IndexedMap 图是一个通用类，其实际类型取决于两个参数：键（每个入口的键的类型）和散列器（基于这些键的散列器类型）。

TCollection_IndexedDataMap 图用来存储键和相应的项目，并且将它们（键值对）与一个索引绑定。它的每一个键都赋有一个索引。随着键和项目的增加，索引也相应地增加。键可以通过索引来访问，而索引也可以通过键来访问。只有最后的键可以被删除。因此，索引在 1 到 Upper（IndexedDataMap 中键的数目）之间取值。每个项目与一个键存在一起。它的每一个入口都由一个键、一个项目和一个索引组成。IndexedDataMap 图是有序的，故允许线性迭代。它具有数组特点和图特点。IndexedDataMap 图是一个通用类，其实际类型取决于三个参数：键（每个入口的键的类型）、项目（与键相应的元素类型）和散列器（基于键的散列器类型）。

4、迭代器

迭代器有：TCollection_ListIterator、TCollection_StackIterator、TCollection_SetIterator、TCollection_BasicMapIterator、TCollection_MapIterator、TCollection

_DataMapIterator 和 TCollection_DoubleMapIterator。

TCollection_ListIterator 迭代器是用来对列表进行迭代函数。一个 List-Iterator 对象可以顺次遍历一个列表，并当作书签（不是索引）使用以记录列表的当前位置。每次迭代都给出了迭代器的当前位置（指向列表的当前项目）。如果列表为空，或者迭代已完成，则当前位置是未定义的。在列表被实例化的同时，系统自动产生一个 ListIterator 实例。

TCollection_StackIterator 迭代器是用来对一个栈进行迭代的函数。在栈被实例化的同时，系统自动产生一个实例自 StackIterator 通用类的迭代器。

TCollection_SetIterator 迭代器是用来对一个集进行迭代的函数。在一个集的实例化过程中，系统自动产生一个实例自 SetIterator 通用类的迭代器。

TCollection_BasicMapIterator 迭代器是其它所有图迭代器的根类。一个图迭代器可以对图的所有入口进行遍历。

TCollection_MapIterator 迭代器是用来对一个 Map 图进行迭代的函数。当一个 Map 图被实例化时，系统自动产生一个实例自 MapIterator 通用类的迭代器。

TCollection_DataMapIterator 迭代器是用来对 DataMap 图进行迭代的函数。因为 DataMap 图是无序的，故 TCollection_DataMapIterator 迭代器的迭代顺序也是不确定的，它取决于图的内容以及图的变化（当图被编辑时）。建议不要在迭代期间修改 DataMap 图的内容，因为那样会导致不可预料的结果。

TCollection_DoubleMapIterator 迭代器是用来对 DoubleMap 进行迭代的函数。

2.3.2 标准对象的集合容器

虽然描述通用集合容器的根类来自 TCollection 包，但是集合容器对象则来自 TColStd 包。TColStd 包 和 TShort 包为那些经常使用的通用类的实例化提供对象（来自 Standard 包）和串（来自 TCollection 包）。

这些实例化有：

（1）对于一维数组，使用标准对象和 TCollection 串对 TCollection_Array1 通用类进行实例化。

（2）对于二维数组，使用标准对象对 TCollection_Array2 通用类进行实例化。

（3）对于通过句柄处理的一维数组，使用标准对象和 TCollection 串对 TCollection_HArray1 通用类进行实例化。

（4）对于通过句柄处理的二维数组，使用标准对象对 TCollection _H- Array2

通用类进行实例化。

（5）对于序列，使用标准对象和 TCollection 串对 TCollection _Sequence 通用类进行实例化。

（6）对于通过句柄处理的序列，使用标准对象和 TCollection 串对 TCollection_HSequence 通用类进行实例化。

（7）对于列表，使用标准对象对 TCollection_List 通用类进行实例化。

（8）对于队列，使用标准对象对 TCollection_Queue 通用类进行实例化。

（9）对于集，使用标准对象对 TCollection_Set 通用类进行实例化。

（10）对于通过句柄处理的集，使用标准对象对 TCollection_HSet 通用类进行实例化。

（11）对于栈，使用标准对象对 TCollection_Stack 通用类进行实例化。

（12）对于基于图键的散列器，使用标准对象对 TCollection_MapHasher 通用类进行实例化。

（13）对于基本散列图：使用标准对象对 TCollection_Map 通用类进行实例化。

（14）对于有一组项目的散列图，使用标准对象对 TCollection_DataMap 通用类进行实例化。

（15）对于基本索引图，使用标准对象对 TCollection_IndexedMap 通用类进行实例化。

（16）对于有一组项目的索引图，使用标准对象对 TCollection_IndexedDataMap 通用类进行实例化。

（17）对于整数图，使用 TColStd_PackedMapOfInteger 类进行实例化。这种实例化在性能和内存使用上都得到了优化（它使用位标记对整数编码；这样的优化结果是每 32 个整数只需 24 个字节空间）。就像对整数集一样，这个类也为整数图提供了布尔操作（求并运算、求交运算、求补运算、求异运算、相等和包含检查）。

2.4 数学基本类型和数学算法

在处理现实问题时，经常将问题抽象成一个数学模型，接着对模型求解，然后将解提取出来以解决现实问题。其实在 CAD 软件中，主要解决的就是数学模型。因此，本节将描述 OCC 的数学基本类型和数学算法。它们包括向量和矩阵类、基本几何类型和常用数学算法。

2.4.1 向量和矩阵类

向量和矩阵组件为向量和矩阵提供了 C++实现。这个组件通常用来定义更复杂的数据结构。向量和矩阵类支持由实数组成的向量和矩阵的标准操作，如加、乘、转置、求逆等。

向量和矩阵的范围是任意的，但必须在声明的时候就定义好，并且定义后不能更改，如例 3.18 所示。

例 3.18：

```
math_Vector v(1, 3);
//一个三维向量，索引区间为（1,3）。
math_Matrix m(0, 2, 0, 2);
//一个 3x3 的矩阵，行区间和列区间都为（0,2）。
math_Vector v(N1, N2);
//一个（N2-N1+1）维的向量，索引区间为(N1,N2)。
```

向量和矩阵对象遵循语意值，即它们不能被共享，也不能通过赋值来拷贝，如例 3.19 所示。

例 3.19：

```
math_Vector v1(1, 3), v2(0, 2);
v2 = v1;
// 将 v1 的值拷贝给 v2；对 v1 的改变不会影响 v2。
```

可以使用索引（必须在定义的范围内）来初始化或包含向量和矩阵值，如例 3.20 所示。

例 3.20：

```
math_Vector v(1, 3);
math_Matrix m(1, 3, 1, 3);
Standard_Real value;
v(2) = 1.0;
value = v(1);
m(1, 3) = 1.0;
value = m(2, 2);
```

在向量和矩阵对象上的有些操作可能是不合法的。这种情况下，系统会产生异常，如例 3.21 所示。可能用到的两种标准异常有：

（1）Standard_DimensionError 异常。当两个矩阵或者向量的维数不同时，系统产生 Standard_DimensionError 异常。

（2）Standard_RangeError 异常。如果向量或矩阵的索引在定义的范围外，系统将产生 Standard_RangeError 异常。

例 3.21：

```
math_Vector v1(1, 3), v2(1, 2), v3(0, 2);
v1 = v2;
//错误，将产生 Standard_DimensionError 异常。
v1 = v3;
//可以，尽管索引区间不相等，但维数是一样的。
v1(0) = 2.0;
//错误，将产生 Standard_RangeError 异常。
```

2.4.2 基本几何类型

在创建一个几何对象前，必须知道这个对象是 2D 的还是 3D 的，以及将如何使用这个对象。

gp 包为二维和三维对象提供了一些通过值处理的类。它定义了一些基本的非持久几何实体；这些实体在二维和三维的代数计算和基本几何结构分析中用到。它也提供一些基本的几何转换，如等价、旋转、平移、镜像、缩放、复合变换等。注意：gp 包中的实体是通过值处理的。

gp 包中可实现的几何实体有：二维和三维直角坐标(x, y, z)、矩阵、笛卡尔点、向量、方向、轴、直线、圆、椭圆、双曲线、抛物线、平面、无穷圆柱曲面、球面、螺旋面和圆锥面。

在创建一个几何对象前，必须知道它是二维的还是三维的，以及将如何使用它。如果需要的不是某种基本几何类型的单个实例，而是某种几何类型的一系列实例，那么 TColgp 包能够处理这样的集合容器，并且提供一些必要的功能。特别地，这个包为通用类中那些标准的和经常使用的实例化提供几何对象。TColgp 包为 TCollection 类的实例化提供类（来自 gp 包，如 XY、 XYZ、 Pnt、 Pnt2d、Vec、Vec2d、 Lin、Lin2d、 Circ、 Circ2d 等； 这些类是非持久的）。

2.4.3 常用数学算法

常用数学算法组件为一些经常使用的数学算法提供 C++实现。它们包括：

（1）求解线性方程组的算法；

（2）寻找一元或多元函数最小值的算法；

（3）求解非线性方程或非线性方程组的算法；

（4）寻找矩阵特征值和特征向量的算法。

所有的数学算法都是采用相同的规则来实现的。这些规则包括：

（1）构造函数。在给定合适的参数后，构造函数完成算法需要实现的所有或者大部分算法。所有相关信息存储在构造的对象中。因此后发计算或后发问题

将以最有效的方式解决。

（2）IsDone 函数。如果计算成功，IsDone 函数将返回布尔真值。

（3）对每一种算法的都提供了一套函数；每套函数能够包含多个结果。只有 IsDone 函数返回真值时，才可以调用函数。否则将产生 StdFail_NotDone 异常。

例 3.22 说明了 Gauss 类（用来实现线性方程组的高斯解法）的用法。例中的声明是从 math_Gauss 类的头文件中提取出来的。

例 3.22：

```
class Gauss
{
    public:
    Gauss (const math_Matrix& A);
    Standard_Boolean IsDone() const;
    void Solve (const math_Vector& B,
                math_Vector& X) const;
};
```

假设要用高斯类解方程 a*x1=b1 和 a*x2=b2，那么程序实现如例 3.23 所示。

例 3.23：

```
#include <math_Vector.hxx>
#include <math_Matrix.hxx>

main ()
{
    math_ Matrix a(1, 3, 1, 3);
    math_Vector b1(1, 3), b2(1, 3);
    math_Vector x1(1, 3), x2(1, 3);
    // a, b1 和 b2 设有相应的值。
    math_Gauss sol(a);
    //计算系数矩阵 A 的 LU 分解。
    if(sol.IsDone())
      {   //是否分解成功?
          sol.Solve(b1, x1);
          //分解成功，则计算第一组解 x1。
          sol.Solve(b2, x2);
          //接着计算第二组解 x2。
          ...
      }
   else
      {  //分解不成功。
         //分析错误原因。
         sol.Solve(b1, x1);
         //错误，产生 StdFail_NotDone 异常。
```

```
    }
}
```

例 3.24 说明了 BissecNewton 类（实现了 Newton 和 Bissection 算法的结合，用来解一个具有指定边界的函数）的用法。

例 3.24：

```
class BissecNewton
{
    public:
    BissecNewton (math_FunctionWithDerivative& f,
                  const Standard_Real bound1,
    const Standard_Real bound2,
    const Standard_Real tolx);
    Standard_Boolean IsDone() const;
    Standard_Real Root();
};
```

抽象类 math_FunctionWithDerivative 描述了这样一些服务，这些服务在 BissecNewton 算法用到的 f 函数中必须被实现。例 3.25 中的声明来自抽象类 math_FunctionWithDerivative 的头文件。

例 3.25：

```
class math_FunctionWithDerivative
{
    public:
    virtual Standard_Boolean Value (const Standard_Real x, Standard_Real& f) = 0;
    virtual Standard_Boolean Derivative (const Standard_Real x,
                                         Standard_Real& d) = 0;
    virtual Standard_Boolean Values (const Standard_Real x,
                                     Standard_Real& f,
                                     Standard_Real& d) = 0;
};
```

下面的测试例子（例 3.26）用 BissecNewton 类来解方程 f(x)=x**2-4，其中 x 在区间[1.5, 2.5]取值。这个待解函数在 myFunction 类中实现，而 myFunc- tion 类是 math_FunctionWithDerivative 的派生类。Main 函数将找到所需的根。

例 3.26：

```
#include <math_BissecNewton.hxx>
#include <math_FunctionWithDerivative.hxx>

class myFunction : public math_FunctionWithDerivative
{
        Standard_Real coefa, coefb, coefc;
    public:
```

```
        myFunction (const Standard_Real a, const Standard_Real b,
                    const Standard_Real c) : coefa(a), coefb(b), coefc(c)
        {
        }
        virtual Standard_Boolean Value (const Standard_Real x,
                                        Standard_Real& f)
        {
            f = coefa * x * x + coefb * x + coefc;
        }
        virtual Standard_Boolean Derivative (const Standard_Real x,
                                             Standard_Real& d)
        {
            d = coefa * x * 2.0 + coefb;
        }
        virtual Standard_Boolean Values (const Standard_Real x,
                                         Standard_Real& f, Standard_Real& d)
        {
            f = coefa * x * x + coefb * x + coefc;
            d = coefa * x * 2.0 + coefb;
        }
};

main()
{
    myFunction f(1.0, 0.0, 4.0);
    math_BissecNewton sol(F, 1.5, 2.5, 0.000001);
    if(Sol.IsDone())
      {  //条件是否为真?
         Standard_Real x = sol.Root();
         //条件为真则执行该语句。
      }
    else
      {  //条件是假的。
         //这里需要一些代码，用来尝试别的方法或者产生异常。
      }
    . . .
}
```

2.5 本章小结

本章分析了 OCC 的基础类模块。

首先，概述了整个基础类。基础类包括根类组件、串类组件、集合容器组件、标准对象的集合容器组件、向量和矩阵类组件、基本几何类型组件、常用数学算

法组件、异常类组件、数量类组件和应用程序服务组件。对于每个组件的功能文中已作了相应的简要的说明。

接着，分析了数据类型、句柄、内存管理器和异常类。数据类型的理解主要在于基本类型的理解。句柄的引入是为了实现引用管理。使用内存管理器的目的是为了能够灵活的分配内存。异常类的使用则是为了保护方法的正常调用。

然后，分析了集合容器和标准对象的集合容器。集合容器都是些通用类，而标准对象的集合容器是用标准对象或串对象对通用集合容器进行实例化的结果。标准对象的集合容器也称集合容器对象。

最后，分析了数学基本类型和数学算法。CAD 软件要分析的是一些几何类型，而分析过程中要用到向量和矩阵这一个代数工具和一些常用数学算法。

第 3 章　数据结构分析

数据结构,指的是数据元素之间的相互关系,尤其是数据的逻辑结构。选择数据结构的主要依据是数据的逻辑结构[6]。因此，本章将主要描述三种数据的逻辑结构。这三种数据包括：二维几何数据、三维几何数据和拓扑数据。

3.1　数据结构模块的整体框架

OCC 的第二个模块是建模数据，也称数据结构。它主要为二维和三维几何模型提供数据结构。数据结构模块由四个工具箱组成：几何工具、二维几何、三维几何和拓扑。各个工具箱提供的服务如表 4.1 所示。

表 4.1 数据结构模块的组成及其各工具箱提供的服务

数据结构模块			
几何工具	二维几何	三维几何	拓扑
1.插值和逼近	1.二维几何类型	1.三维几何类型	1.保持 Shape 定位轨迹
2.Shape 的直接构造	2.二维几何类型的集合容器	2.三维几何类型的集合容器	2.Shape 和 SubShape 的处理
3.B 样条转换	3.二维适配器	3.三维适配器	3.拓扑数据结构的访问
4.极值计算	4.拓扑类型和拓扑方向	4.曲线和曲面的局部特征	4.Shape 列表和 Shape 图

3.2　二维几何数据结构

3.2.1　概述

二维几何数据结构定义了二维空间上几何对象的数据结构，主要由 Geom2d 包提供。

Geom2d 包提供了比 gp 包更大范围的对象。这些对象是非持久的，并且是通过引用而不是通过数值处理的。复制一个实例，只是将句柄拷贝而已，对象没有被拷贝。因此，对一个实例（对象）的改变，将在所有引用它的地方发生改变。

如果需要的对象不是单一的，而是一系列的，那么 TColGeom2d 包（用来处

理这类对象的集合容器）将提供必要的功能。特别地，该包为通用类中那些标准的和经常使用的实例化提供几何对象。

TColGeom2d 包为来自 Geom2d 包的曲线提供一维数组、二维数组和序列的实现。所有这些对象能以两种不同的方式处理：句柄处理和数值处理。

3.2.2 TopAbs 包

TopAbs 包提供通用枚举类，用来描述拓扑学基本概念和处理枚举类的方法，不包含具体类。该包已经从拓扑包中分离出来了，因为对于所有拓扑工具而言，它的概念已经足够通用了，可以通过保留独立的模型资源，避免枚举类的重复定义。TopAbs 定义了三个概念：拓扑类型（由 TopAbs_ShapeEnum 类描述）、拓扑方向(由 TopAbs_Orientation 类描述)和拓扑状态(由 TopAbs_State 类描述)。

1、拓扑类型

TopAbs_ShapeEnum 枚举类列出了不同的拓扑类型。一个拓扑模型可以看作一个由多个彼此相邻的对象组成的图表。当在二维或三维空间上建立模型的一部分的时候，该部分的类型必须是 ShapeEnum 枚举类列出的类型之一。对于任何一个模型，里面可以找到的所有对象，TopAbs 包都列出了。Top- Abs 包不能再扩展，但是可以用到它的一部分，例如，实心体的概念在二维中是没有意义的。

表 4.2 给出 ShapeEnum 枚举类的各元素及其意义。

表 4.2 中出现的枚举类术语是从复杂到简单依次排列的，因为在描述对

表 4.2 ShapeEnum 枚举类的各元素及其意义

元素		意　义
TopAbs_COMPOUND	复合体	由不同拓扑类型对象组成的整体。
TopAbs_COMPSOLID	组合实心体	由几个通过面连接的实心体组成。它将线框和壳的概念延伸到实心体上。
TopAbs_SOLID	实心体	由壳限制的空间的一部分。它是三维的。
TopAbs_SHELL	壳	由几个面（通过边连接）组成。壳可以是开放的，也可以是封闭的。
TopAbs_FACE	面	在二维中，它是平面的一部分。 在三维中，它是曲面的一部分；它的几何结构由构造函数决定；它是二维的。

TopAbs_WIRE	线框	由一组边（通过定点连接）组成。它可以是开放的，也可以是封闭的，这取决于边是否彼此相连。
TopAbs_EDGE	边	与约束曲线对应的拓扑元素。边通常由顶点限制。它是一维的。
TopAbs_VERTEX	顶点	与点对应的拓扑元素。它没有维度。
TopAbs_SHAPE	Shape	包含以上所有类型的通用术语。

象时，可以说对象包含更简单的对象。例如，一个面引用了它的线框、边和顶点。为了进一步理解 Shape 枚举类型，给出一个示意图（图 4.1），图中表示了部分 Shape 枚举类型。

2、拓扑方向

拓扑方向的概念由 TopAbs_Orientation 枚举类描述。在许多建模器里都用到了方向；拓扑方向就是这种方向感的通用概念。当一个 Shape 限制了一

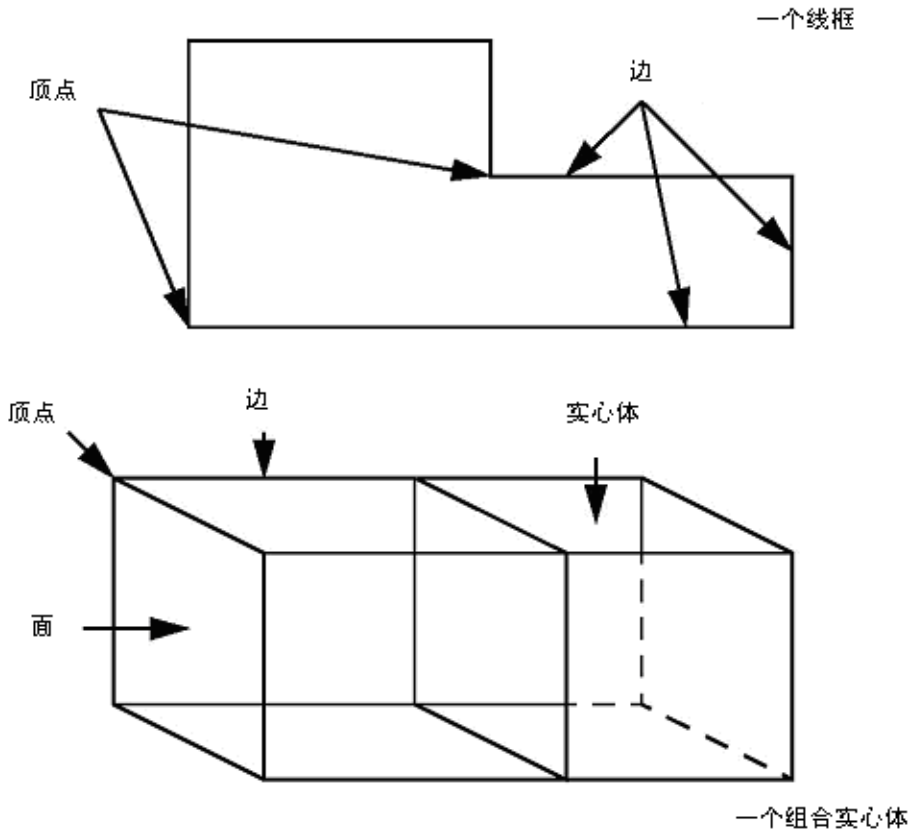


图 4.1 几个 Shape 枚举类型

个几何域时，就说这个 Shape 具有拓扑方向。拓扑方向的概念与边界的概念紧密联系。需要用到拓扑方向的三种 Shape 有：由顶点限制的曲线、由边限制的曲面

和由面限制的空间。

无论哪种 Shape（作为维数更高的空间几何域的边界）都定义了两个局部段，其中一个被指定为默认段。

对于一条由顶点限制的曲线，默认段就是一系列点（这些点的参数比顶点的大）。也就是说，沿着曲线方向，除去顶点，其它所有点就是该曲线的默认段。对于一个由边限制的曲面，它的默认段位于边的左边（沿着边的自然方向看，即沿着逆时针方向看）。确切地说，默认段由曲面法向量和曲线切向量指出。对于一个由面限制的空间，它的默认段位于与曲面法向相反一边。

基于这个默认段，拓扑方向允许对保留段（称为 Shape 的内部或 Shape 的材质）进行定义。定义 Shape 的内部有四种拓扑方向，如表 4.3 所示。

表 4.3 四种拓扑方向

拓扑方向	对 Shape 内部的定义
前面	内部是默认段。
后面	内部是与默认段完全相反的段。
里面	内部包含两个段。边界在材质的里面，例如，实心体的内曲面。
外面	两个段都不在内部。边界在材质外面。例如，线框模型的边。

拓扑方向是一个非常通用的概念，只要有段或边界出现的地方就可以用到它。例如，当需要描述一条边与一个轮廓的交叉时，不但可描述出交叉段的定点，而且可以描述出这条边是如何穿越该轮廓的——将边当成轮廓的一个边界，如图 4.2 所示。

表 4.4 给出了这四个拓扑方向的文字描述。

与 Orientation 枚举类对应，TopAbs 包定义了四种方法，如表 4.5 所示。

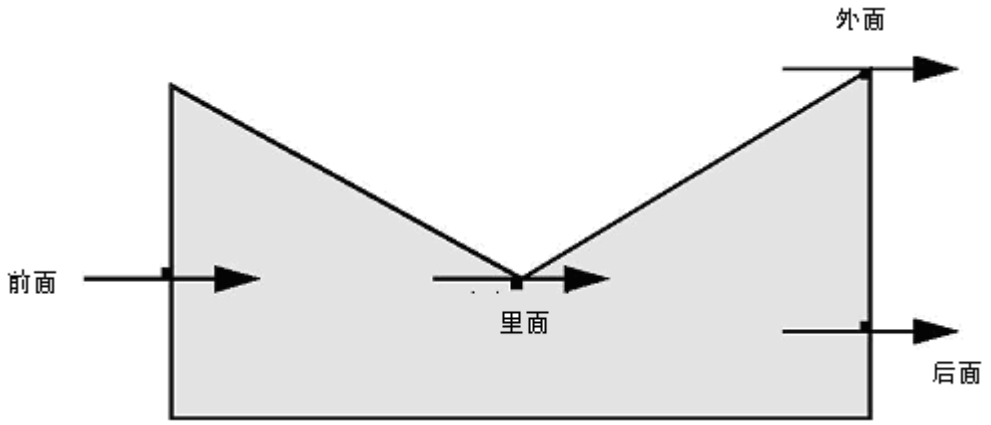


图 4.2 交叉顶点的四个拓扑方向

3、拓扑状态

TopAbs_State 枚举类描述了段上一个或一组顶点的位置。有四种位置，如表 4.6 所示。

UNKNOWN 术语之所以被引入，是因为这个枚举类经常被用于表达一个可能失败的计算结果。当无法判断点在内部还是外部的时候（通常这种情况发生在线框或面是开放的时候），就可以用到该术语，如图 4.3 所示。

表 4.4 交叉顶点四个拓扑方向的描述

交叉顶点的四个拓扑方向	描述
前面	由形状的外部进入内部。
后面	从形状的内部退出。
里面	从形状的内部接触。
外面	从形状的外部接触。

表 4.5 处理拓扑方向的四种方法及描述

方法	描述
组合	将两个方向组合在一起以包含第三个拓扑方向。例如，将面上线框的拓扑方向与其中一条边的拓扑方向组合，以包含该面上该边的拓扑方向。
倒置	在两边将内部拓扑状态和外部拓扑状态对换。这发生在方向感被倒置的时候。前面和后面被对换了。里面和外面没有变化。

求补	在每边将内部拓扑状态和外部拓扑状态对换。对一个对象求补意味着要在对象的边界上实现该操作。前面和后面被对换了。里面和外面也被对换了。
输出	将拓扑方向的名字送到输出流。

表 4.6 四种位置（或拓扑状态）及其描述

位置	描述
IN	点在内部。
OUT	点在外部。
ON	点在边界上（在容差内）。
UNKNOWN	拓扑状态不明。

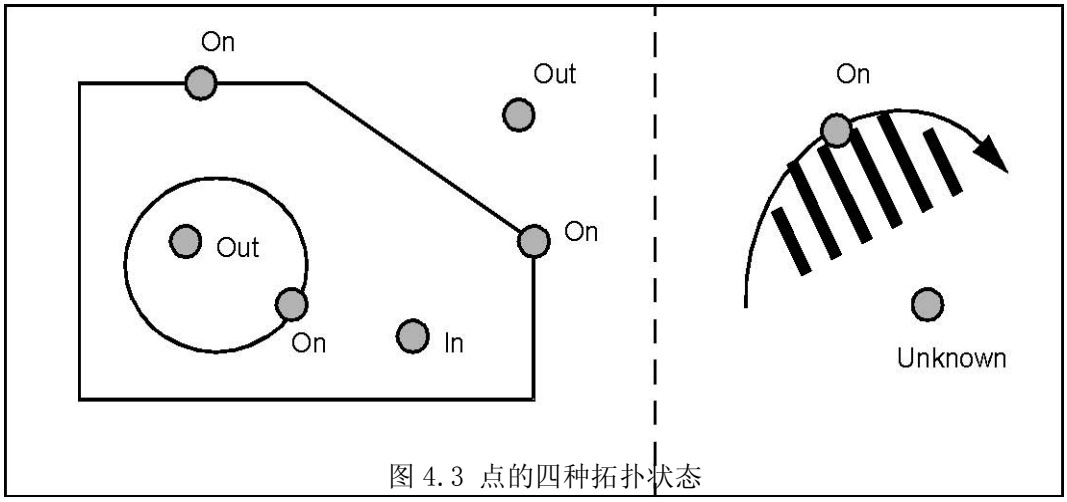


图 4.3 点的四种拓扑状态

State 枚举类也适用于对象的各个部分。图 4.4 表示了一条边（与一个面交叉）的各部分拓扑状态。

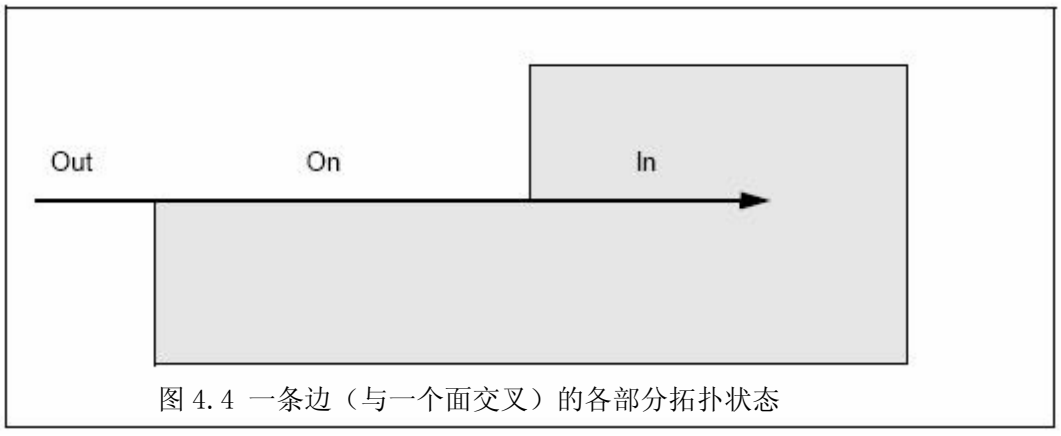


图 4.4 一条边（与一个面交叉）的各部分拓扑状态

3.3 三维几何数据结构

三维几何对象的数据结构主要由 Geom 包提供。

Geom 包包含了所有的基本几何转换（如等价、旋转、平移、镜像、缩放和复合变换等）。另外，Geom 包也提供了一些基于几何对象引用定义的特殊函数（如在 B 样条曲线上增加一个控制点，对曲线进行改善等）。

Geom 包中所有几何实体都是以 STEP 方式处理的。包中可实现的非持久的通过句柄处理的对象有：点、笛卡尔点、向量、方向具有幅值的向量、轴、曲线、直线、圆锥、圆、椭圆、双曲线与抛物线、基本曲面、平面、边界曲线与边界曲面、裁剪曲线与裁剪曲面、非均匀有理化 B 样条曲线与曲面、Bezier 曲线与曲面、圆柱面、球面与螺旋面、扫描曲面、线性挤压曲面、旋转曲面、偏移曲面。

对于每一条曲线（具有一个参数），其局部特征有：点、导数、正切、法向量、曲率和曲率中心。

对于每一个参数，曲面（具有两个参数）的局部特征有：导数、切线、曲率中心、点、法向量、最大曲率、最小曲率、曲率的主方向、中间曲率和高斯曲率。

另外，GeomLProp 和 Geom2dLProp 包提供了一些描述算法的类，如表 4.7 所示。

TColGeom 为那些来自 Geom 包的曲线和曲面提供一维数组、二维数组和序列的实现。所有这些可实现的对象有两种处理方式：通过引用处理和通过数值处理。

3.4 拓扑数据结构

拓扑数据结构定义了参数空间中对象的数据结构，由拓扑工具箱提供。该工具箱提供五种服务：保持 Shape 位置轨迹，Shape、SubShape 以及它们的拓扑方向和拓扑状态的命名，Shape 和 SubShape 的处理，访问拓扑数据结构，以及使用 Shape 列表和 Shape 图。本节将论述第一、第三和第四种服务。

表 4.7 GeomLProp 和 Geom2dLProp 包提供的算法及其描述

类	描述
CLProps	这个函数计算了曲线的局部特征（正切、曲率、法向量和扭矩）。它在二维上计算曲率极值和变形点。

SLProps	这个函数计算了曲面的局部特征（切线、放向量和曲率）。
Continuity	这个全局函数计算了两曲线连接处的平滑性。

注 1：这些算法也适用于特殊情况（如不可导，平行直线等）。

注 2：B 样条曲线和曲面作为算法的参数被接受；这些曲线和曲面不是被切成一段段或一块块具有理想连续性的碎片，而是具有全局连续性的。

3.4.1 保持 Shape 位置轨迹

可以将 OCC 的局部坐标系可以看作以下两者之一：

（1）具有一个原点和三个标准正交向量的右手三面体。这个定义由 gp _Ax2 类提供。

（2）由 a+1 决定的转换，它允许我们在局部引用框架和全局引用框架间进行坐标转换。这个定义由 gp_Trsf 类提供。

在此，需要认识两个概念（在 TopLoc 包中）：基本引用坐标和复合引用坐标。基本引用坐标是用在右手直角坐标系或者右手线性转换中的坐标；而复合引用坐标则由几个基本引用坐标组成。

如果两个引用坐标系是由相同的基本坐标系以相同的顺序组成的，那么这两个坐标系是等价的。例如，有三个基本坐标系：R1, R2, R3 。由上述三者复合而成的坐标系有：

C1 = R1 * R2

C2 = R2 * R3

C3 = C1 * R3

C4 = R1 * C2

这里 C3 和 C4 是等价的，因为它们都是由 R1 * R2 * R3 复合而成的。

3.4.2 Shape 和 SubShape 的处理

1、概述

Shape 和 SubShape 的处理由 TopoDS 包提供。

TopoDS 包描述了具有以下特征的拓扑数据结构：

（1）对没有拓扑方向也没有位置的抽象 Shape 的引用

（2）通过工具类对数据结构的访问。

TopoDS 包是基于 TopoDS_Shape 类和那个定义了自己下层 Shape 的类的。

这样做有不少好处，但也有一个缺点：这些类太通用了，不同的 Shape 的不一定属于不同的类型。因此不可能通过检查来防止冲突（如将一个面插入到一条边）。

TopoDS 包提供了两套类：一套由下层 Shape（既没有拓扑方向也没有位置）派生；另一套则由 TopoDS_Shape（与 TopoAbs 包中列出的标准拓扑 Shape 一致）派生。

2、TopoDS_Shape 类

如前面所言，OCC 描述了参数空间对象的数据结构。这些描述用到了定位和限制。可以由这些术语描述的 Shape 的类型有：顶点、面和 Shape。顶点是依据参数空间位置来定义的，而面和 Shape 则依据空间的限制来定义。

OCC 的拓扑描述允许我们将由上述术语定义的简单 Shape 组成集。例如，一个边集形成一个线框；一个面集形成一个壳；一个实心体集形成一个组合实心体。也可以将不同类型的 Shape 组成一个复合体，并为一个 Shape 指定拓扑方向和位置。

依据 Shape 的复杂程度，从顶点到组合实心体依次将 Shape 列出来，这样能方便的知道一个 Shape 是由哪些简单 Shape 组成的。事实上，这就是 TopoDS 包的意图。

一个 Shape 模型是一个可共享的数据结构，因为它可以被其它 Shape 使用（如一条边可以被一个实心体的多个面使用）。可共享的数据结构是通过引用来处理的。若一个简单引用不够充分时，需要增加两个信息：一个拓扑方向和一个局部坐标引用。拓扑方向 描述被引用的形状如何在一个边界使用。一个局部引用坐标(Location from TopLoc)允许程序员在 Shape 的定义位置之外引用该 Shape。

TopoDS_TShape 类是所有 Shape 描述的根类，包含一个 Shape 列表。如果有必要，由 TopoDS_TShape 派生的类可携带几何域信息（例如，一个 TVertex 类型的几何点）。一个 TopoDS_TShape 就是一个 Shape 在自己的引用框架上的描述。该类是通过引用处理的。

TopoDS_Shape 类描述了 Shape 的引用。它包含一个指向下层抽象 Shape 的引用、一个拓扑方向和一个引用坐标。该类是通过数值处理的，因此不能被共享。

描绘下层 Shape 的类从不被直接引用，而是通过 TopoDS_Shape 类被间接引用。

Shape 的特殊信息总是由 TopoDS_TShape 的派生类的派生类添加。图 4.5 表示了一个由两个面组成（通过一条边连接）的壳。

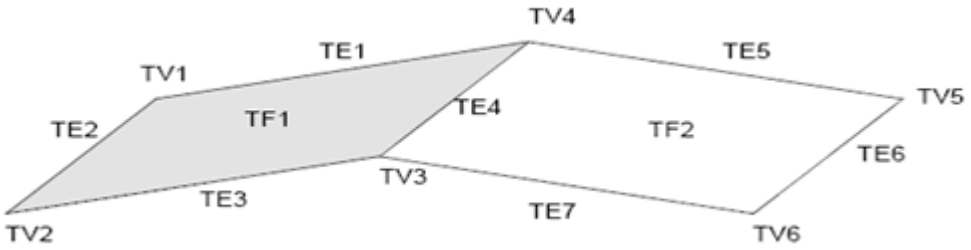


图 4.5 以两个面作为下层 Shape 的壳结构

而图 4.6 给出了图 4.5 中的壳的数据结构。

在图 4.6 中，壳由下层 Shape TS 描述，面则由下层面 TF1 和 TF2 描述。整个壳包含七条边（TE1~TE7）和六个顶点（TV1~TV6）。线框 TW1 引用了 TE1~TE4 四条边；TW2 则引用了 TE4~TE7 四条边。顶点被这些边引用：TE1(TV1,TV2), TE2(TV2,TV3), TE3(TV3,TV4), TE4(TV4,TV5), TE5(TV5,TV6), TE6(T6,TV7), TE7(TV7,TV8)。

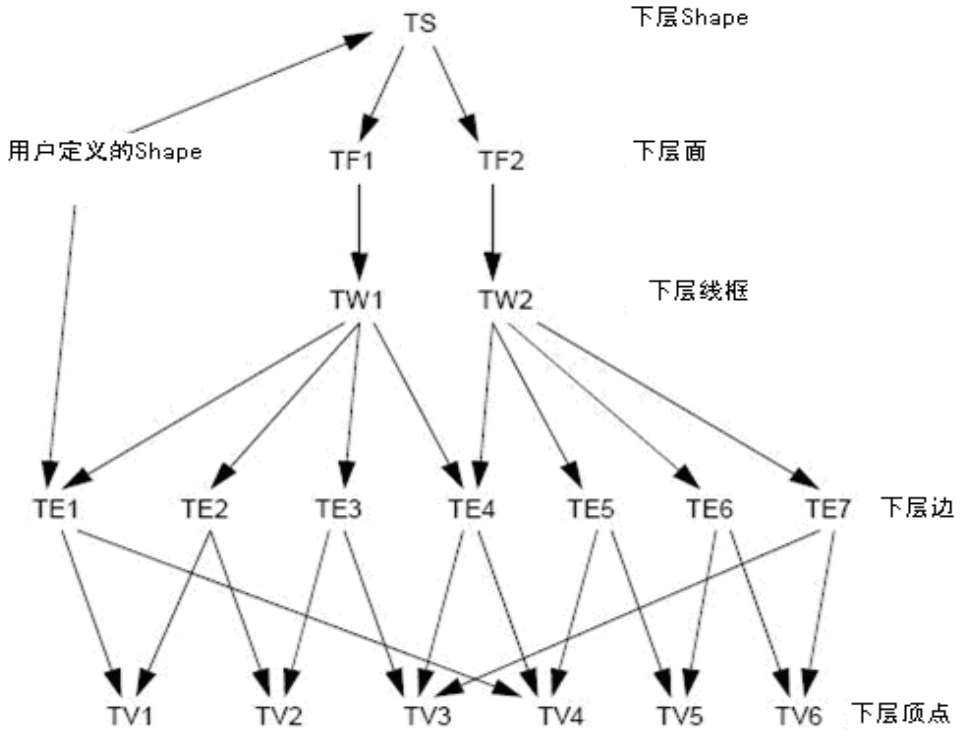


图 4.6 由两个面通过一条边连接组成的壳的数据结构

图 4.6 中的数据结构不包含任何中断引用。所有的引用都是从复杂的下层 Shape 到简单的下层 Shape 的。另外，这个数据结构要尽可能紧凑，并且子对象是能够被不同对象共享的。

两个非常相似的对象（也许是同一个对象的两个版本）可以共享相同的子对

象。该数据结构中，局部坐标的使用允许那些重复子结构的描述被共享。当创建一个新的对象版本或者对象需要适应坐标变化时，通常会用到复制这项操作；紧凑的数据结构有效避免了因复制而带来的信息缺失。

图 4.7 描绘的数据结构包含一个实心体的两种版本。其中第二个版本描绘了一系列处于不同位置但相同的孔。这样的结构不但紧凑，而且包含了子元素的所有信息。从 TSh2 到它的下层面 TFa 和 TFb 的三个引用，都有相应的基于孔心位置的局部坐标系。

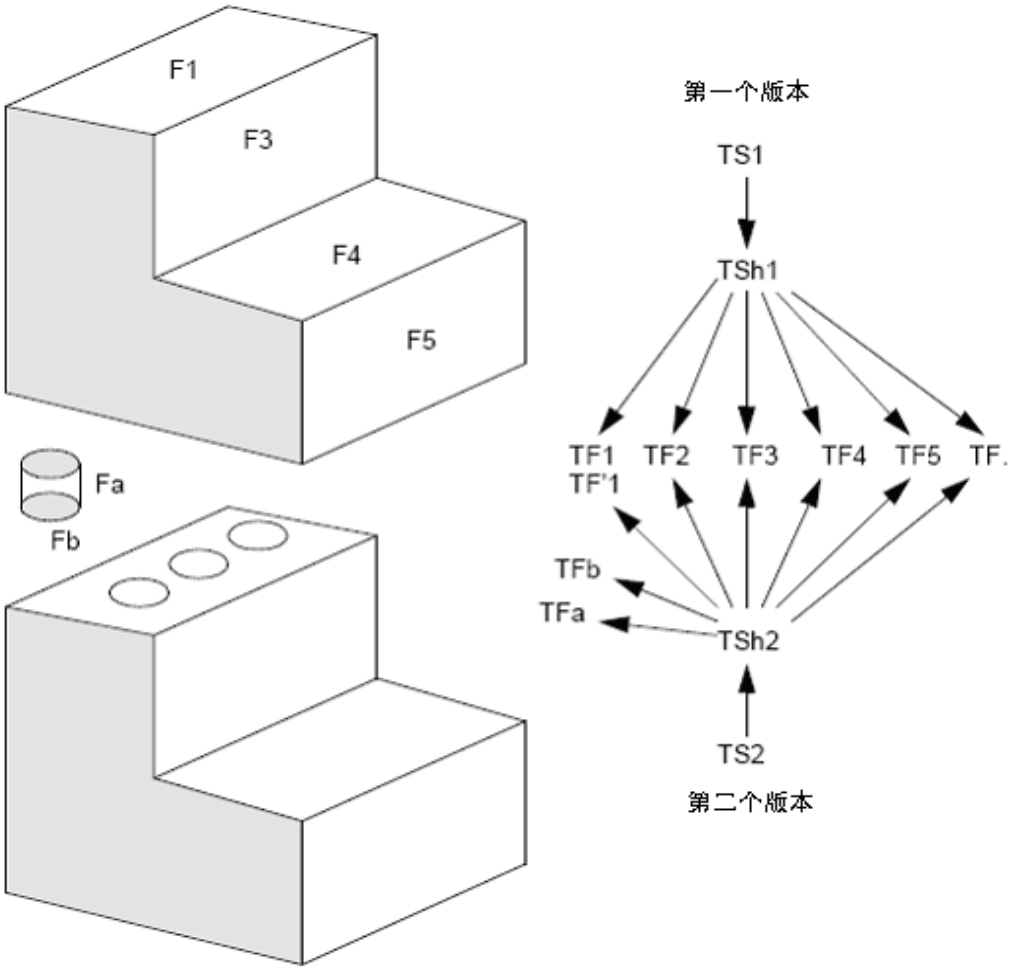


图 4.7 包含一个实心体的两种版本的数据结构

TopoDS_Shape 类的方法有：

（1）IsNull 和 Nullify。若一个 Shape 不指向任何下层 Shape（既没有拓扑方向也没有位置），那么该 Shape 是空的（Shape 的默认拓扑状态）。

（2）Location、Move 和 Moved：用来访问 Shape 的局部引用坐标。

（3）Orientation、Oriented、Reverse 和 Reversed：用来访问 Shape 的拓扑方向。

（4）ShapeType：用来访问基于下层 Shape（没有拓扑方向也没有位置）的相等方法。

（5）IsPartner。如果两个 Shape 拥有同一个下层 Shape（没有拓扑方向也没有位置），则称这两个 Shape 是伙伴关系。这两伙伴的拓扑方向和局部引用可以不同。

（6）IsSame。如果两个 Shape 是伙伴关系，并且它们拥有相同的局部引用(TopLoc_Location)，则称这两个 Shape 是等价的。例如，通过一条边连接的两个面，它们有两条相同的边。

（7)IsEqual。如果两个 Shape 等价并且有同样的拓扑方向，则称这两个 Shape 相等，用“=”表示相等关系。

3、TopoDS_Shape 的派生类

TopoDS_Shape 类及其派生类是处理拓扑对象的自然方式。Shape 的派生类有：TopoDS_Vertex、 TopoDS_Edge、TopoDS_Wire、TopoDS_Face、TopoDS_Shell、TopoDS_Solid、TopoDS_CompSolid 和 TopoDS_Compound。尽管它们的名字与 TopoDS_TShape 派生类的名字相似，但彼此的用法相差很大。

TopoDS_TShape 类是不可见的。它在自己的没有拓扑方向的原局部坐标系中定义了一个类。TopoDS_Shape 是一个引用；这个引用包含在 TopoDS_ TShape（具有拓扑方向和局部引用）中。

TopoDS_TShape 类是推迟类，而 TopoDS_Shape 类不是。有了 TopoDS_ Shape 类，就可以在不知道拓扑对象具体类型的情况下处理拓扑对象了。这是一种通用形式。纯拓扑算法经常用到 TopoDS_Shape 类。

TopoDS_TShape 类是通过引用处理的；而 TopoDS_Shape 类通过数值处理。TopoDS_Shape 类只是一个具有拓扑方向和局部坐标的引用。共享 TopoDS_Shapes 类是没有意义的。重要的是共享 TopoDS_TShapes。形参的赋值和传递并没有将数据结构拷贝，而只是创建了一些新的 TopoDS_Shape 类（这些类赋予同一个 TopoDS_TShape 类）。

尽管 TopoDS_TShape 的派生类用来添加额外信息，TopoDS_Shape 的派生类却不能添加额外的域。TopoDS_Shape 的派生类仅仅用来将一个引用特殊化，以便通过静态类型来控制对象（由编译器实现）。例如，对编译器而言，以 TopoDS_Face 类作参数要比以 TopoDS_Shape 类参数更精确。编译器能够接收的只有 TopoDS_TopoDS 类；其它的类都是没有意义的。拓扑数据结构的所有引用

都是由 Shape 类和它的继承类（在 TopoDS_TopoDS 类中定义）制造的。

TopoDS_Shape 的派生类没有构造函数，否则会因为“同名覆盖”（C++的一个特点）而失去类型控制权。TopoDS 包提供了这样一些类方法，这些类方法能够通过类型匹配，将 TopoDS_Shape 类的一个对象抛掷给 TopoDS_Shape 派生类的一个对象。

例 4.1 说明了如何接收一个 TopoDS_Shape 类型的参数，然后又是怎样将它传递给变量 V（如果该参数是一个顶点的话）的，或者如何调用 ProcessEdge 方法（如果该参数是一条边的话）。

例 4.1：

```
#include <TopoDS_Vertex.hxx>
#include <TopoDS_Edge.hxx>
#include <TopoDS_Shape.hxx>

void ProcessEdge(const TopoDS_Edge&);
void Process(const TopoDS_Shape& aShape)
{
    if (aShape.Shapetype() == TopAbs_VERTEX)
     {
        TopoDS_Vertex V;
        V = TopoDS::Vertex(aShape);
        //该语句是正确的。
        TopoDS_Vertex V2 = aShape;
        //该语句不能被编译器处理
        TopoDS_Vertex V3 = TopoDS::Vertex(aShape);
        //正确
     }
  else if (aShape.ShapeType() == TopAbs_EDGE)
         {
            ProcessEdge(aShape) ;
            //编译器将无法处理该语句。
            ProcessEdge(TopoDS::Edge(aShape)) ;
            //正确
         }
       else
          {
            cout << “Neither a vertex nor an edge ?” ;
            ProcessEdge(TopoDS::Edge(aShape)) ;
            //编译没有问题，但是运行的时候将产生一个异常。
          }
}
```

3.4.3 拓扑数据结构的访问

TopExp 包提供了一些工具，用来访问由 TopoDS 包描述的数据结构。访问一个拓扑结构意味着找到指定类型的所有子对象，例如，找到实心体的所有面。

TopExp 包提供了一个 TopExp_Explorer 类（一个工具，用来找到指定类型的所有子对象。）和一些包方法。这些包方法有：

（1）MapShapes ，用来收集一个 Shape 的子图对象。

（2）MapShapesAndAncestors，用来收集子对象和包含子对象的对象；例如，由一个实心体的所有边组成的集合容器和由一个面（包含某条边）的所有边组成的集合容器。

（3）FirstVertex、LastVertex 和 Vertices 这三者用来寻找边的顶点。

TopExp_Explorer 是一个工具，用来访问来自 TopoDS 包的拓扑数据结构。一个访问器与下面三者同时建立：

（1）要访问的 Shape。

（2）要查找的 Shape 的类型，例如顶点、边和 Shape 异常（这是不允许的）。

（3）Shape 要避免的类型，例如壳、边。默认的，这个类型是 SHAPE。这个默认值意味着访问不受限制。

访问器要访问整个结构，以找到要求的类型（不包括要避免的类型）。例 4.2 说明了如何找到 Shape S 的所有面。

例 4.2：

```
void test()
{
    TopoDS_Shape S;
    TopExp_Explorer Ex;
    for (Ex.Init(S,TopAbs_FACE); Ex.More(); Ex.Next())
      {
        ProcessFace(Ex.Current());
      }
}
```

找到不在指定边上的所有顶点，如例 4.3。

例 4.3：

```
for (Ex.Init(S,TopAbs_VERTEX,TopAbs_EDGE); ...)
```

先找到一个壳的所有面，然后找到不在壳中的所有面，如例 4.4。

例 4.4：

```
void test()
{
```

```
    TopExp_Explorer Ex1, Ex2;
    TopoDS_Shape S;
    for (Ex1.Init(S,TopAbs_SHELL);Ex1.More(); Ex1.Next())
    {
        //访问所有壳
        for (Ex2.Init(Ex1.Current(),TopAbs_FACE);Ex2.More(); Ex2.Next())
        {
            //访问当前壳的所有面
            ProcessFaceinAshell(Ex2.Current());
            ...
        }
    }
    for(Ex1.Init(S,TopAbs_FACE,TopAbs_SHELL);Ex1.More(); Ex1.Next())
    {
        //访问不在壳中的所有面。
        ProcessFace(Ex1.Current());
    }
}
```

访问器假定对象只包含相同类型或者下属类型。例如，要查找面，访问器不会在轮廓、边、或顶点上查找。

下面通过两个例子（例 4.5 和例 4.6）来说明 TopExp 包的使用方法。其中例 4.5 是 TopExp 包中 MapShapes 方法的源代码，说明了在填充图的时候如何使用访问器。

如果一个对象被多次引用，那么访问器就可以多次访问该对象。例如，实心体的一条边通常被两个面引用。要保证只对一个对象访问一次，必须将这些对象置于一个图中，如例 4.5。这就是 MapShapes 方法的意图。

例 4.5：

```
void TopExp::MapShapes (const TopoDS_Shape& S,
                        const TopAbs_ShapeEnum T,
                        TopTools_IndexedMapOfShape& M)
{
    TopExp_Explorer Ex(S,T);
    while (Ex.More())
    {
        M.Add(Ex.Value());
        Ex.Next();
    }
}
```

在例 4.6 中，一个对象的所有面和所有边都是依据以下规则画出来的：

（1）面是由具有 FaceIsoColor 颜色的 NbIso isoparametric 直线网格描绘的。

（2）边用一种颜色（标记着有多少个面共享这条边）画出。

（3）FreeEdgeColor 用来标记不属于面的边，即线框元素。

（4）BorderEdgeColor 用来标记只属于某一个面的边。

（5）SharedEdgeColor 用来标记属于多个面的边。

（6）要显示单条边和单个面可以使用 DrawEdge 和 DrawFaceIso 方法。

例 4.6 要经历的步骤如下：

（1）将边存到图中，并创建一个相应的整数数组（一个元素记录了一条边被几个面共享）。该数组初始值是零。

（2）访问面，并画出每一个面。

（3）访问边，并且每访问一条边都相应的在数组中增加面计数器的值。

（4）形成边图，用相应的颜色（记录着面数）画出每一条边。

例 4.6：

```
void DrawShape ( const TopoDS_Shape& aShape,
const Standard_Integer nbIsos,
const Color FaceIsocolor,
const Color FreeEdgeColor,
const Color BorderEdgeColor,
const Color SharedEdgeColor)
{
    //将边存入图中。
    TopTools_IndexedMapOfShape edgemap;
    TopExp::MapShapes(aShape,TopAbs_EDGE,edgeMap);
    //创建一个数组，数组的初始值为 0。
    TColStd_Array1OfInteger faceCount(1,edgeMap.Extent());
    faceCount.Init (0);
    //访问面。
    TopExp_Explorer expFace(aShape,TopAbs_FACE);
    while (expFace.More())
    {
        //画出当前的面。
        DrawFaceIsos(TopoDS::Face(expFace.Value()),nbIsos, FaceIsoColor);
        //访问该面的所有边。
        TopExp_Explorer expEdge(expFace.Value(),TopAbs_EDGE);
        while (expEdge.More())
        {
            //为该边增加面数。
            faceCount(edgeMap.Index(expEdge.value()))++;
            expEdge.Next();
        }
```

```
        expFace.Next();
        }
        //画出图中所有的边。
        Standard_Integer i;
        for (i=1;i<=edgemap.Extent();i++)
        {
            switch (faceCount(i))
            {
                case 0 :
                        DrawEdge(TopoDS::Edge(edgeMap(i)),FreeEdgeColor);
                        break;
                case 1 :
                        DrawEdge(TopoDS::Edge(edgeMap(i)),BorderEdgeColor);
                        break;
                default :
                        DrawEdge(TopoDS::Edge(edgeMap(i)),SharedEdgeColor);
                        break;
            }
        }
    }
```

3.5 本章小结

本章分析了 OCC 的数据结构模块。

首先，描绘了数据结构模块的整体框架。数据结构模块包含四个工具箱，它们分别是：几何工具工具箱、二维几何工具箱、三维几何工具箱和拓扑工具箱。每个工具箱提供了不同的服务。

接着，分析了二维几何数据结构。二维几何数据结构定义了二维空间上几何对象的数据结构，主要由 Geom2d 包提供。二维几何数据结构的理解难点在于拓扑类型、拓扑方向和拓扑状态的理解。

然后，分析了三维几何数据结构。三维几何数据结构定义了三维几何对象的数据结构，主要由 Geom 包提供。

最后，分析了拓扑数据结构。拓扑数据结构定义了参数空间中对象的数据结构，由拓扑工具箱提供。

第五章 OCC CAD 系统总体框架

5.1.1 基础类

OpenCASCADE 提供的基础类包括：内核、数学工具两部分。

在内核中提供：根类、各种变量、异常处理、串、集合、标准集合对象等功能。数学工具类提供：向量和矩阵的基本算法、方阵、线性非线性方程的相关算法。还提供几何图元，并具有支持坐标系的平移、旋转、对称、缩放转换和自由变换等功能。

5.1.2 可视化

OpenCASCADE 提供的可视化的功能是对图形对象进行显示以及对图形对象进行动态拾取。对于数据结构的可视化， OpenCASCADE 提供了“即用”的算法。将数据结构提供给视图，应用交互服务（AIS） 提供图像的显示，搜索管理服务。

5.1.3 图形工具

OCC 提供的 GUIF（Graphical User Interface Framework）是基于 OCAF 的用于开发图形用户界面的工具，为用户提供了“即用”的图形操作功能。这些图形操作包括：

1、二维：放大、局部放大、缩放、平移、选择新视图中心、格栅操作。

2、三维：放大、局部放大、缩放、平移、选择新视图中心、旋转、重设视图、隐藏线的显隐。

5.2 二维 Geometry 框架程序建立

5.2.1 建立类

本程序是通过使用 MDI[15]多文本模板生成的，使用 APPWizard 生成一个 MDI 工程，在此工程中程序框架提供了：头文件（.h） 、资源文件（.rc）、实现文件（.cpp）。框架为我们定义了如下类：应用程序类 CWinApp、文档类 CDocument、视图类 CView、主框架类 CMainFrame、子框架类 CChildFrame。我们建立的是和它们对应的派生类：CGeometryApp、CGeometryDoc、CGeo-metryView2D 、CMainFram、CChildFrame2D。在此基础上为完善应用程序功能又添加了如下类：GeomSources、ISession2D_Curve、ISession2D_Intera-ctiveContext、ISession2D_SensitiveCurve。

1、类功能

（1）CGeometryApp：派生自 MFC 提供的程序基类 CWinApp，从整体上对程序进行管理。例如，初始化程序、最后程序清除。管理从程序开始到程序结束的全过程。MFC 的文档/视图结构、程序主窗口都是在此类中定义和创建的。

（2）CGeometryDoc：派生自文档基类 CDocument 类，存放应用程序数据和文件间的存储和读取，文档类把数据处理从界面中分离出来，并且提供与其它对象的接口。

（3）CGeometryView2D：派生自基类 CView，显示文档中的数据，处理交互输入 ：鼠标、键盘、菜单、工具条等命令。通过 GetDocument()函数获取文档指针，用来读取和操作文档中的数据，实现对文档数据的修改。通过调用 CDocument::UpdateAllView()函数更新视图的显示内容。

（4）CMainFram：派生自 CFrameWnd，管理应用程序主框架窗口—主窗口。管理用户界面对象，菜单、工具条、状态栏、对话框、标题栏、控制按钮等。管理并跟踪视图窗口，包容子框架窗口。

（5） CChildFrame2D：派生自 CFrameWnd 类，管理应用程序窗口，与主框架共享菜单栏、工具栏、状态栏。

以上五类是基本类，程序通过调用类的公共成员函数和发送消息进行通信，协同工作交换数据，实现程序功能。

（6）GeomSources：设定 2D 点、曲线等的显示函数。二维图形绘制功能函数存放在此类中，包括：点、直线、圆、弧、B 样条线、自由曲线等。

（7）ISession2D_Curve：设计线形、线宽、颜色，提供计算曲线的相关算法。

（8）ISession2D_InteractiveContext：对交互环境进行初始化、显示、清理、移动视图、拾取，显示区域、清理区域，擦除所有操作。

（9）ISession2D_ SensitiveCurve：对自由曲线进行计算，包括：切线偏差、初始化、边界包容盒。

2、类关系

建立的基本类有五个，分别是：CGeometryDoc 、CMainFrame 、CChildFrame2D 、CGeometryView2D 和 CGeometryApp。它们之间的关系如上图 3.2 所示。

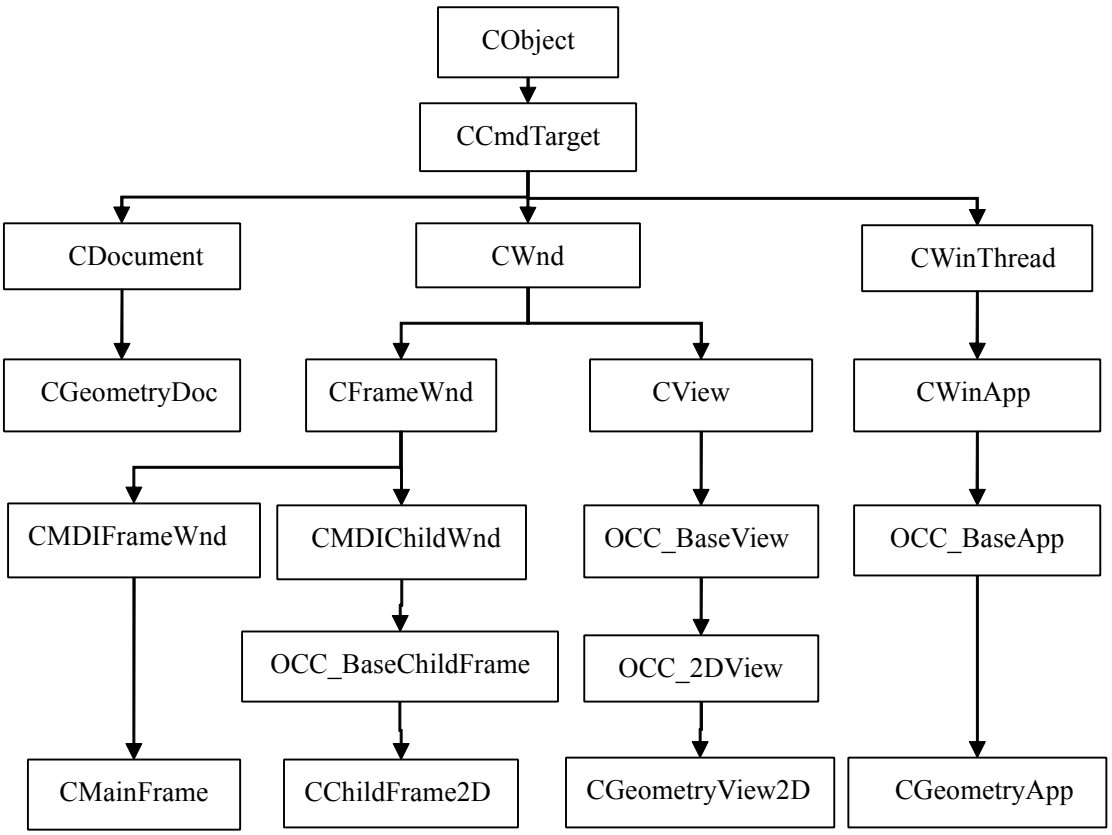


图 3.2　Geometry 类图

在类关系图中，可以发现所有类都派生自基类 CObject，组成一个整体。在派生出 CCmdTarget 类之后，产生三个子类：CDocument 文档类、 CWnd 窗口类、CWinThread 类。其中 CDocument 类是应用程序文档的基类。CWinThread 类经过一系列派生出应用程序类 CGeometryApp。CWnd 窗口类是 CFrameWnd 类和 CView 类的父类。主框架和子框架都派生自主框架窗口类，只是使用的模板类不同。

5.2.2 二维框架

使用 MFC 框架搭建应用程序的轮廓。应用 APPWizard 生成初步的框架文件（代码、资源等）。利用资源编辑器设计用户接口。使用 ClassWizard 添加代码到框架文件。编译，通过类库实现应用程序。使用 MFC 建立的框架如图 3.3 所示。

（1）声明全局变量 theApp，初始化应用程序类，在 CGeometryApp::Ini-tlns tance()进行应用程序的初始化。

（2）初始化的过程中，引用 CMultiDocTemPlate 类来建立文档模板类，利用文档模板类把 CGeometryDoc、CGeometryView2D 和 CChildFrame2D 联系起来。

（3）由文档模板类建立应用程序的文档类，用于记录、保存和调用程序产生的数据。

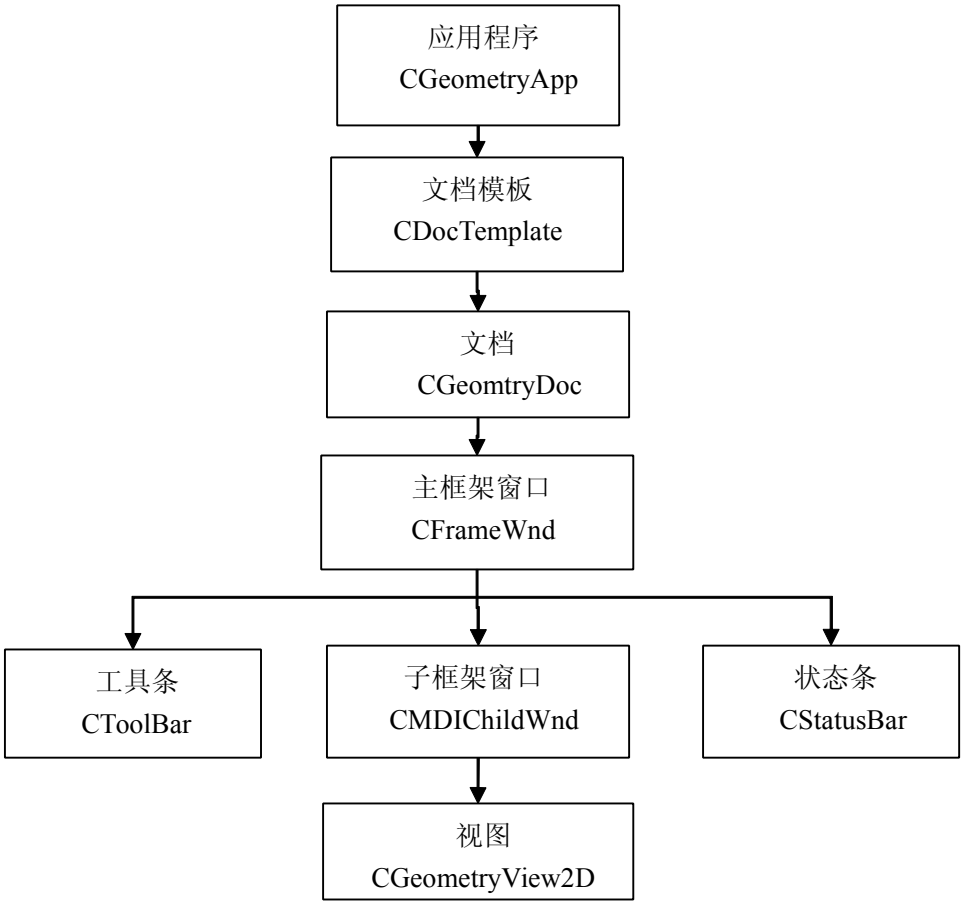


图 3.3 MDI 框架

（4）调用 CMainFrame 类来建立应用程序主框架，包括：菜单栏、工具栏和状态栏。

（5）调用 CChildFrame2D 类建立应用程序子框架。

（6）调用 CGeometryView2D 类来建立应用程序客户区，作为图像输出窗口。

5.2.3 Geometry 框架程序的实现

（1）初始化应用程序 ，在 CGeometryApp 函数中添加代码完成初始化。

```
CGeometryApp::CGeometryApp()
{
        SampleName = "Geometry";
}
BOOL CGeometryApp::InitInstance()
{    //设置注册表
     SetRegistryKey(_T("Local AppWizard-Generated Applications"));
     LoadStdProfileSettings();
     //添加多文档模板
     //用模板将文档类、子框架类、视图类关联
     pDocTemplateForView2d = new CMultiDocTemplate(
          IDR_2DTYPE,
          RUNTIME_CLASS(CGeometryDoc),
          RUNTIME_CLASS(CChildFrame2D),
          RUNTIME_CLASS(CGeometryView2D));
     //添加模板
     AddDocTemplate(pDocTemplateForView2d);
     //生成主框架窗口
     CMainFrame* pMainFrame = new CMainFrame;
     if (!pMainFrame->LoadFrame(IDR_MAINFRAME))
          return FALSE;
     m_pMainWnd = pMainFrame;
……
}
```

（2）创建视图器和交互环境。在 CGeometryDoc 的函数中，通过应用程序初始化得到的显示设备生成视图器并设置相关参数。

```
CGeometryDoc::CGeometryDoc()
{    //获得图形设备
     Handle(Graphic3d_WNTGraphicDevice)  theGraphicDevice  =
     ((CGeometryApp*)AfxGetApp())->GetGraphicDevice();
     ……
     //新 2D 视图器
     myViewer2D=newV2d_Viewer(theGraphicDevice,a2DName.ToExtString());
     ……
     //环境设置
     myISessionContext  =  new ISession2D_InteractiveContext(myViewer2D);
     myCResultDialog.Create(CResultDialog::IDD,NULL);
}
```

（3）创建文档。文档用于保存模型数据，在 CGeometryDoc::OnNew-Document()中，创建文档。

```
BOOL CGeometryDoc::OnNewDocument()
{
    if (!CDocument::OnNewDocument())
        return FALSE;
        return TRUE;
}
```

（4）初始化 View2D 视图。在 CGeometryView2D::OnlnitialUpdate()中，初始化视图。

```
void CGeometryView2D::OnInitialUpdate()
{
    Handle(WNT_Window)aWNTWindow=new WNT_Window(((CGeometryApp*)-Afx
    GetApp()) - > GetGraphicDevice(),GetSafeHwnd());
    //设置背景
    aWNTWindow->SetBackground(Quantity_NOC_BLACK);
    //窗口驱动
    Handle(WNT_WDriver) aDriver= new WNT_WDriver(aWNTWindow);
    //新 2D 视图
    myV2dView=newV2d_View(aDriver,GetDocument()->GetViewer2D(),0,0,50);
    ……
}
```

（5）完成初始化。在 CGeometryView2D 类的 OnSize 和 OnDraw 函数中，完成重绘的初始化。

```
void CGeometryView2D::OnSize(UINT nType, int cx, int cy)
{   if (!myV2dView.IsNull())
    {
      myV2dView->MustBeResized(V2d_TOWRE_ENLARGE_SPACE);
    }
}
void CGeometryView2D::OnDraw(CDC* pDC)
{
    CGeometryDoc* pDoc = GetDocument();
}
```

5.2.4 接口设计

在使用 MFC 建立的应用程序框架中，提供标准用户接口实现方法。利用资源编辑器设计用户接口，使用 ClassWizard 向框架中添加代码，经过编译利用类库就可以实现要求的功能[18]。

在所建立的 MFC 框架中，接口可以设在类 CGeometryDoc、CMainFrame 、

CGeometryView2D 中任意一个类中。本课题中，把接口设计在 CMainFrame 类中，并利用 Classwizard 来设计接口。在 CMainFrame 类的 OnCreate()中，把菜单栏、状态栏和工具条添加到主框架中，其中工具条分成两个工具条 ToolBar、ToolBar2。

ToolBar 关键代码：m_wndToolBar.SetBarStyle(m_wndToolBar.GetBarStyle() |CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC);

ToolBar2 关键代码：m_wndToolBar2.SetBarStyle(m_wndToolBar2.GetBarStyle() |CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC);

5.3 三维 Viewer3D 框架程序建立

353.1 建立类

本程序是通过调用 MDI 多文本模板生成的，使用 APPWizard 生成一个 MDI 工程，在此工程中程序框架提供了：头文件（.h） 、资源文件（.rc）、实现文件（.cpp）。建立了如下类：应用程序类 CWinApp、文档类 CDocument、视图类 CView，CAboutDlg。我们建立了从他们对应派生出的类：CViewer3dApp、CViewer3dDoc、CViewer3dView、ScaleDlg、CModleClipping、ShadingModel。

1、类功能

（1）CViewer3dApp：派生自 MFC 提供的程序基类 CWinApp，从整体上对程序进行管理。例如初始化程序、最后程序清除。管理从程序开始到程序结束的全过程。MFC 的文档/视图结构、程序主窗口都是在此类中定义和创建的。

（2）CViewer3dDoc：派生自文档基类 CDocument 类，存放应用程序数据和文件间的存储和读取，文档类把数据处理从界面中分离出来，并且提供与其它对象的接口。

（3）CViewer3dView：派生自基类 CView，显示文档中的数据，处理交互输入 ：鼠标、键盘、菜单、工具条等命令。通过 GetDocument()函数获取文档指针，用来读取和操作文档中的数据，实现对文档数据的修改。

（4）ScaleDlg：派生自 CDialog 类，用于对实体进行沿坐标方向进行比例变化。

（5）ShadingModel：派生自 CDialog 类，用于实体的隐藏、用三角面近似生成实体、近似实体恢复原状。

（6）CModleClipping：派生自 CDialog 类，沿 Z 轴对实体进行剪切，观察 BRep 实体内部。

2、类间关系

通过类关系图说明它们之间关系，如上图 3.4 所示。

5.3.2 三维框架

利用 MFC 框架定义的应用程序轮廓，通过预定义的接口把具体应用程序代码填入轮廓。使用 APPWizard 生成初步的框架文件（代码、资源等）；利用资源编辑器设计用户接口；使用 Classwizard 协助添加代码到框架文件；编译，通过类库实现应用程序。使用 MFC 建立的框架如图 3.5 所示。

（1）声明全局变量 theApp，初始化应用程序类，在 CViewer3dApp::Init-lnst

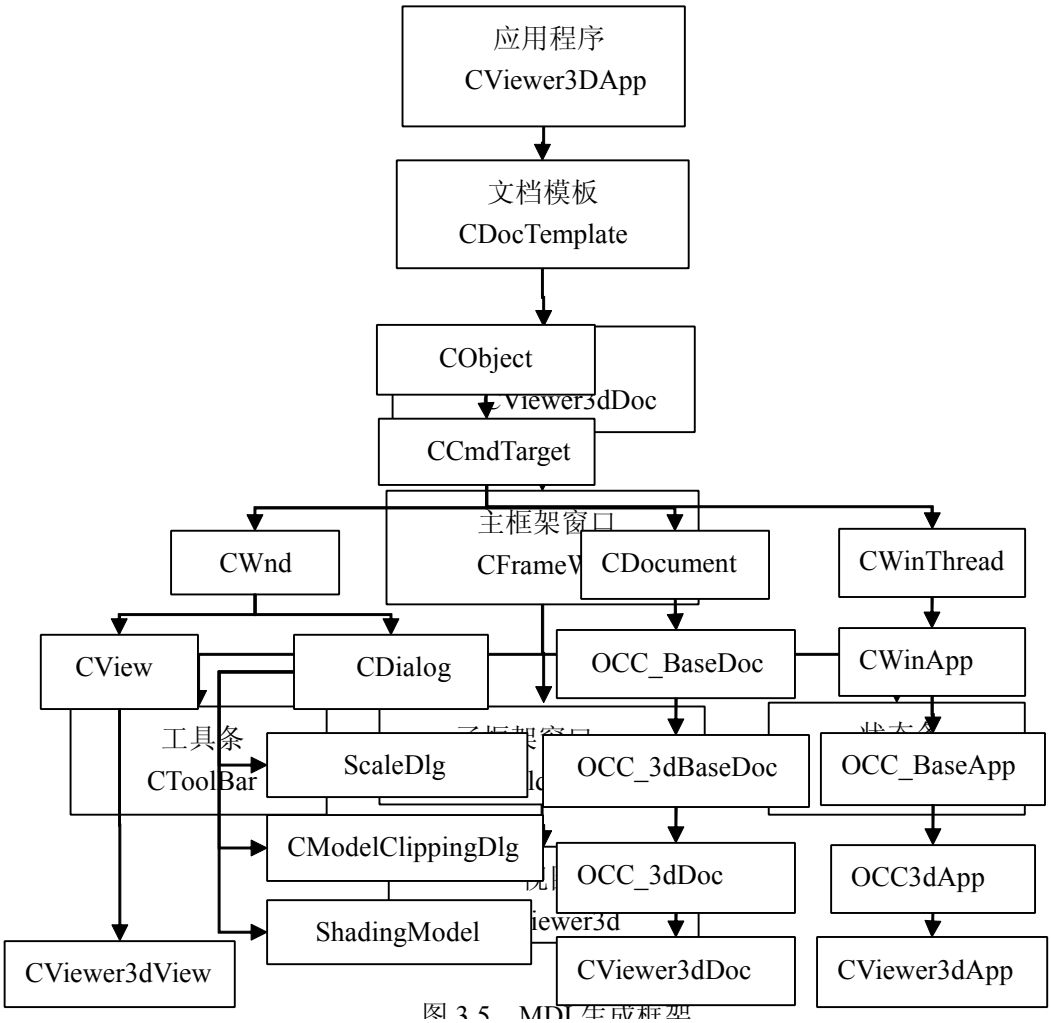


图 3.4 Viewer3d 类关系图

ance()进行应用程序的初始化。

（2）初始化的过程中，引用 CMultiDocTemPlate 类来建立文档模板类，利用文档模板类把 CViewer3dDoc、CViewer3dView、OCC_3dChildFrame 联系起来。

（3）由文档模板类建立应用程序的文档类，用于记录、保存和调用程序产生的数据。

（4）调用 OCC_MainFrame 类来建立应用程序主框架，包括：菜单栏、工具栏和状态栏，包容子框架。

（5）调用 CChildFrame 类建立应用程序子框架，客户区。

（6）调用 CViewer3dView 类来建立应用程序客户区，作为图像输出窗口。

5.3.3 Viewer3d 框架程序的实现

（1）初始化应用程序 ，在 CViewer3dApp 函数中添加代码完成初始化。

```
BOOL CViewer3dApp::InitInstance()
{    //选择注册表项
     SetRegistryKey(_T("Local AppWizard-Generated Applications"));
     LoadStdProfileSettings();
     //注册多模板
     //用模板将文档、子框架、视图类关联
     CMultiDocTemplate* pDocTemplate;
     pDocTemplate = new CMultiDocTemplate(
          IDR_3DTYPE,
          RUNTIME_CLASS(CViewer3dDoc),
          RUNTIME_CLASS(OCC_3dChildFrame),
     RUNTIME_CLASS(CViewer3dView));
     //添加模板
     AddDocTemplate(pDocTemplate);
     //建立 MDI 主框架窗口
     OCC_MainFrame* pMainFrame = new OCC_MainFrame(with_AIS_TB);
     if (!pMainFrame->LoadFrame(IDR_MAINFRAME))
          return FALSE;
     m_pMainWnd = pMainFrame;
}
```

（2）创建三维视图器和交互环境。此功能是通过继承基类 OCC_MainFr-ame 属性完成的。

（3）创建文档。通过继承基类 OCC_MainFrame 属性完成此功能。

（4）初始化 Viewer3d 视图。在 CViewer3dView::OnlnitialUpdate()中，初始化视图。

```
{    //获得图形设备
```

```
    Handle(Graphic3d_WNTGraphicDevice) theGraphicDevice =
    ((CViewer3dApp*)AfxGetApp())->GetGraphicDevice();
    //建立窗口
    Handle(WNT_Window)aWNTWindow = new
    WNT_Window(theGraphicDevice,GetSafeHwnd ());
    myView->SetWindow(aWNTWindow);
}
```

（5）在 CViewer3dView 类的 OnSize 和 OnDraw 函数中，完成重绘的初始化。

```
void CViewer3dView::OnSize(UINT nType, int cx, int cy)
{   if (!myView.IsNull())
    myView->MustBeResized();
 }
void CViewer3dView::OnDraw(CDC* pDC)
{   CViewer3dDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    CRect aRect;
    myView->Redraw();
 }
```

5.3.4 接口设计

在 MFC 框架中，接口可以设在类 CViewer3dDoc、OCC_CMainFrame 、CViewer3dView 中任意一个类中。本题目中，把接口设计在 OCC_MainFrame 类中，并利用 ClassWizard 来设计接口。在 OCC_MainFrame 类中，对菜单栏、状态栏和工具条进行添加。

5.4 本章小结

本章分析了 OCC 结构及类库的内容，结合第二章的分析，进行了框架的搭建。包括：类的添加及功能分析，类的层次结构分析，并对建立的框架进行分析。在此基础上完成了应用程序框架。包括二维 Geometry 框架和三维 Viewer3d 框架的建立。并对接口进行了设计。

第六章 二维草图的读取（不会）

6.1 综述

DXF 是 Drawing eXchange File 的缩写，意思为图形交换文件，在工程制图中有广泛的应用，掌握了 DXF 文件的读写对编写 CAD 软件时的图形信息的交换有重要意义。它有两种格式：一种是 ASCII DXF 格式；一种是二进制 DXF 格式。ASCII DXF 文件格式是 ASCII 文字格式的 AutoCAD 图形的完整表示，这种文件格式易于被其它程序处理。二进制格式的

DXF 文件与 ASCII 格式的 DXF 文件包含的信息相同，但格式上二进制格式比 ASCII 格式更精简，能够节省百分之二十五的文件空间。AutoCAD 能够更快地对其执行读写操作（通常能快五倍）。这可能是对 ASCII 格式的 DXF 文件操作时有 ASCII 与二进制形式的转换，因而花费时间较多。本文主要讨论 ASCII 格式的 DXF 文件，因为它可读性强。

6.2 ASCII 格式的 DXF 文件的组成

先来介绍一下 ASCII 格式的 DXF 文件的组成。(小提示：打开 AutoCAD，新建一个空的文件，然后再输出为 DXF 文件，并用记事本打开 DXF 文件，就可以看到它的所有代码了，这样有助于你更好地理解 DXF 文件的组成。

用记事本打开一个 DXF 文件，你可以发现它里面有这样一些代码：

0、SECTION、 2、HEADER、 9、$ACADVER、1、AC1015……

即里面总是数字和字符串/数字在交替的出现。数字就叫做代码（通常称为组码），紧跟组码数字的称为关联值对。（以下内容来自 DXF 参考）DXF 文件本质上由代码及关联值对组成。代码（通常称为组码）表明其后的值的类型。使用这些组码和值对，可以将 DXF 文件组织到由记录组成的区域中，这些记录由组码和数据项目组成。在 DXF 文件，每个组码和值各占一行。

一个完整的 ASCII 格式的 DXF 文件结构如下：

HEADER 段。它包含图形的基本信息。它由 AutoCAD 数据库版本号和一些系统变量组成。每个参数都包含一个变量名称及其关联的值。

CLASSES 段。包含应用程序定义的类的信息，这些类的实例出现在数据库的 BLOCKS、ENTITIES 和 OBJECTS 段中。类定义在类的层次结构中是固定不变的。

TABLES 段。包含以下符号表的定义：

APPID（应用程序标识表）

BLOCK_RECORD(块参照表)

DIMSTYLE(标注样式表)

LAYER(图层表)

LTYPE(线型表)

STYLE(文字样式表)

UCS(用户坐标系表)

VIEW(视图表)

VPORT(视口配置表)

BLOCKS 段。包含构成图形中每个块参照的块定义和图形图元。

ENTITIES 段。包含图形中的图形对象（图元），其中包括块参照（插入图元）。这里的信息很重要。

OBJECTS 段。包含图形中的非图形对象。除图元、符号表记录以及符号表以外的所有对象都存储在此段。OBJECTS 段中的条目样例是包含多线样式和组的词典。

THUMBNAILIMAGE 段。包含图形的预览图像数据。此段为可选。

每个段都以一个后跟字符串 SECTION 的组码 0 开始，其后是组码 2 和表示该段名称的字符串（例如，HEADER）。每个段都由定义其元素的组码和值组成。每个段都以一个后跟字符串 ENDSEC 的组码 0 结束。（第五章肯定不行，）

第七章 草图消隐

第八章 三维拉伸建模

Open CASCADE 里包含数量庞大的图形绘制包，拉伸用到了其中的一些类，例如：BRepBuilderAPI_MakeFace,BRepPrimAPI_MakePrism 等，在二维图形的拉伸过程中，从点的创建开始，形成边，线框，面，最后通过指定路径或指定高度得到三维图，里面用到了很多 Open CASCADE 里面的 BRep 代码，对相关的代码进行详细分析后，才能正确运用到程序设计中。

8.1 Open CASCADE 中拉伸的框架分析

下面介绍如何创建二维图形，并通过拉伸二维图形生成三维物体。

首先得由它的坐标系创建特征点，这些点将支持几何轮廓的定义。除此之外，还要确定它的轮廓被定位在笛卡尔坐标系的原点。

下面有两个属于 OPENCASCADE 中的类，它们用来描述 3D 坐标点：

（1） 初等几何 gp_Pnt 类

（2） 瞬时的 Geom_CartesianPoint 类，它由一个 handle 来进行操作

Handle 是智能指针的一种类型，提供自动化的记忆管理。

可以根据我们的实际需要来使用这两个类：

gp_Pnt 是由值来操作的，它定义一个非持续作用的 3D 笛卡尔的点；Geom_CartesianPoint 是由 gp_Pnt 类型的点与它的直角坐标系来定义的，handle 对它进行操作，它可以有多种参数值，并有较长的作用时间。

如果定义的所有点只能用来创建一个短暂作用时间的轮廓的曲线对象。则可以选择 gp_Pnt 类，建立一个 gp_Pnt 对象，首先得在笛卡尔坐标系下确定 X,Y,Z 的坐标点。

当使用 Geom_CartesianPoint 类时，方法有一点不同。所有的对象通过 handle 用标准的 C++来管理。接着用精确定义的点，计算出圆，矩形，椭圆等的几何轮廓的一部份。

创建一个实体，首先需要一个指定的，能实现 3D 几何对象的数据结构。Open CASCADE 里定义了一系列的类，它们有相同的属性，并同属于相同的结构。GEOM 包能实现 3D，而曲线和表面要素是为更复杂的对象提供的（例如贝赛尔曲线和 B 样条曲线）。

但是 GEOM 包只提供几何实体的数据结构，可以直接使用属于它的类来创建一个实例，但是如果要更容易的计算曲线和表面要素，最好使用 GC 包。这是因为 GC 提供两种运算类，对于建立轮廓的操作来说更为精确。

（1） GC_MakeSegment 类在 3D 空间里对一个线段执行构造运算。运算结果为

Geom_TrimmedCurve 类型的曲线。

（2） GC_MakeArcOfCircle 类在 3D 空间里对一段弧执行构造运算。运算结果为 Geom_TrimmedCurve 类型的曲线。

所有的类都由 handle 管理，并返回 Geom_TrimmedCurve。实体表现为基础的曲线，由它的两个参数值限制。例如圆 C 是在 0 和 2PI 之间参数化的。

创建完轮廓的一部份几何结构，但此时这些曲线之间没有任何关联。为了简化这个模型，可以把这些曲线当作几个单一的实体来处理。可以用 TOPODS 包里的 OPENCASCADE 拓扑数据结构，它的作用是定义能形成复杂外形的几何实体间的关系。

TOPODS 包内的每个对象，都是从 TOPODS_SHAPE 继承而来的。描述一个拓扑结构外形如表 8. 1 所示。

由表格可以看出，建立一个轮廓，首先要创建精确计算的曲线，然后是由这些曲线组成的线框。但是，TOPODS 包只为拓扑实体提供数据结构。运算类可以有效计算建立在 BRepBuilderAPI 内的标准的拓扑对象。

创建一条边，可以用 BRepBuilderAPI_MakeEdge 类。连接这些边，需要用 BRepBuilderAPI_MakeWire 类建立一个线框。下面有两种方法：

（1）直接从一条边开始，创建四条边

（2）对一个存在的线框用添加线框和边的方法。从少于四条边的图形上建立一个线框。可以用以下方法：

TopoDS_Wire aWire = BRepBuilderAPI_MakeWire();

一旦线框的第一部分创建了，就需要计算完整的轮廓。可以用一种简单的方法对已存在的一个进行反射，计算一个新的线框；为原本的对象添加线框。

对外形运用一个转化，首先得用类 gp_Trsf 定义 3D 几何变换的特性。这种转化关系可以是一个变换，旋转，比例，反射或者是组合。

表 8.1 几何元素描述

Shape	Open CASCADE Class	Description
Vertex	TopoDS_Vertex	Zero dimensional shape corresponding to a point in geometry.
Edge	TopoDS_Edge	Single dimensional shape corresponding to a curve and bounded by a vertex at each extremity.
Wire	TopoDS_Wire	Sequence of edges connected by vertices.
Face	TopoDS_Face	Part of a surface bounded by a closed wire.
Shell	TopoDS_Shell	Set of faces connected by edges.
Solid	TopoDS_Solid	Part of 3D space bounded by Shells.
CompSolid	TopoDS_CompSolid	Set of solids connected by their faces.
Compound	TopoDS_Compound	Set of any other shapes described above.

在考虑坐标系统中的 X 轴的同时需要定义一个反射。轴是由 gp_Ax1 类定义的，由一个点和一个方向进行创建的。有两种方式：

第一种是在它的草图上定义，用它的几何定义：

（1） X 轴在（0，0，0）定位—用 gp_Pnt。

（2） X 轴在（1，0，0）定位—用 gp_Dir。

第二种是最简单的，几何常数在 gp 包里定义，用 gp::OX 可以得到 X 轴。

3D 几何转换特征是由 gp_Trsf 类定义的，当遇到以下两种情况可以使用这个类：

在草图上定义一个转换矩阵或用适合的方法来调整所需的转换，矩阵是自动计算的。因为最简单的方法是最好的，可以使用关于轴镜像的方法。

当用 BRepBuilderAPI_Transform 类去进行转换时，应符合下面两个条件：

（1） 转换的外形必须是可实用的

（2） 必须是几何转换

BRepBuilderAPI_Transform 不会修改原始外形：反射线框的结果保持着线框的外形，当需要一个将反射结果转换为线框的方法时，TOPODS 的功能将提供这方面的服务，计算转换的线框应该用 TopoDS::Wire：

轮廓基本完成后，这时已经创建了两个线框：一个线框和一个镜像的线框。需要把它们连接起来用以计算一个单独的外形。为完成这个动作，选择使用 BRepBuilderAPI_MakeWire。

用 ADD 函数在这两个线框上添加所有的边

接着是计算主要部份，首先需要创建实体的形状。最简单的方法是用最精确的创建方法并沿着一个方向扫描：

表 8.2 几何元素的关系

Shape	Generates
Vertex	Edge
Edge	Face
Wire	Shell
Face	Solid
Shell	Compound of Solids

当前的轮廓是个线框，根据上面的表，需要计算线框之外的表面来生成一个实体。首先创建一个表面，用类 BRepBuilderAPI_MakeFace。面是由封闭线框限制的表面的一部分。通常，类 BRepBuilderAPI_MakeFace 用来计算表面之外的一个面不止一个的线框。

BRepPrimAPI 包提供所有的类来创建拓扑初等结构，立方体，圆锥体，圆柱体，球体等。它们是用类 BRepPrimAPI_MakePrism 创建的。

我们所需要的是一个有限的实体，可以沿着 Z 轴拉伸到我所需要的高度，高度参数 myHeight。顶点是由 X,Y,Z 轴上的 gp_Vec 来定义的，当所有创建瓶身的必需数据都有效时，则运用类 BRepPrimAPI_MakePrism 来计算我们所需要的实体：

通过以上分析可以总结出下面的一个流程图

gp_Pnt 定义一个非持续的三维笛卡尔点。

GC_MakeArcOfCircle 在三维空间里对一段圆弧执行运算，得到一个 Geom_TrimmedCurve 类型曲线。

GC_MakeSegment 在三维空间对线段执行运算，得到一个 Geom_TrimmedCurve 类型曲线。

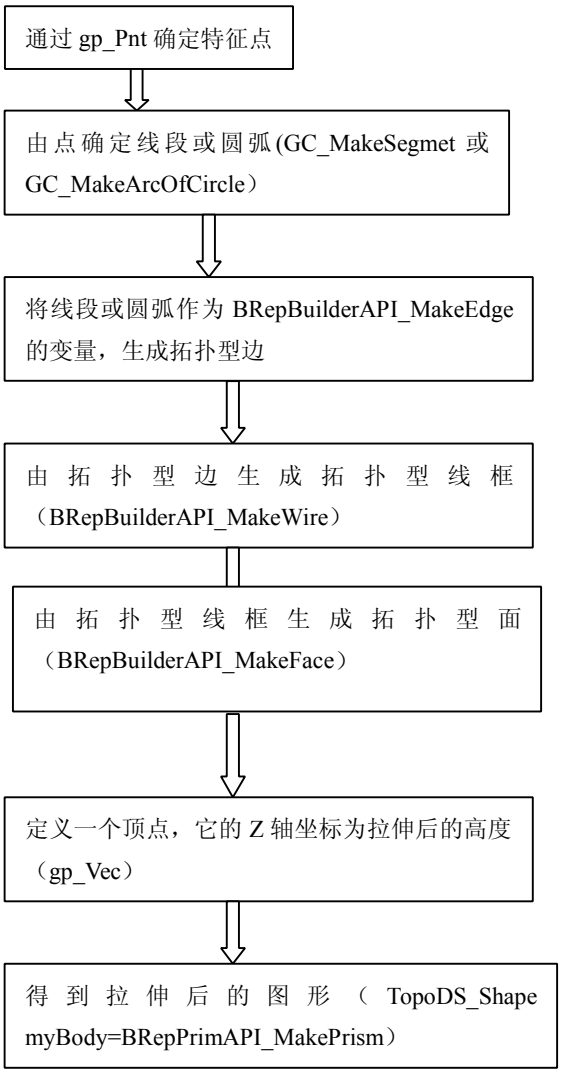
BRepBuilderAPI_MakeEdge 提供创建边的方法。

BRepBuilderAPI_MakeWire 描述从边创建线框的函数。一个线框可以由任何数量的边创建。创建一个首次进行初始化的线框，然后顺次添加边，对添加的边无数量限制。

BRepBuilderAPI_MakeFace 提供创建面的方法

gp_Vec 在三维空间定义一个非持续点

BRepPrimAPI_MakePrism 为创建线性扫描定义函数。



8.1 流程图

第九章 布尔运算（缺理论）

OCC 提供这样的类库。这个类库应用交、差、并、补计算来演示布尔操作创建新的实体：

BRepAlgoAPI 包含各种布尔操作的功能。在 BRepAlgoAPI 包中新的计算工具代替旧的布尔操作计算。

新的计算方法比原来的更有优势，它更加的柔性化并且能创建一个在原来的操作中不能创建的自变数。新的计算方法是基于图形操作接口。其优势是实体共享的能力。它能处理两个实体面的交接等问题。新的布尔操作计算能描述更多的图形。

共享同一个域的两个面按照同一个规则被处理，尽管这个基础曲面是不同的类型。在同一个域的面中可以执行布尔操作。两个面共享同一个面是很常见的 。

布尔操作可以计算不同类型的布尔操作间的冲突。在给定的交、差、并和补操作中，它可能只计算一次冲突。因此不需要重复计算，这节省了更多的时间。不支持 COMPSOLID 图形类型的布尔操作。

第十章 OCC 可视化基本概念

OCC 平台中，可视化是建立在建模数据及描述图形的数据结构之上的,主要实现对实体对象的显示和选取操作。本章对可视化的一些基础知识进行分析。首先介绍 OCC 提供的各种包类以及相关概念；其次，对选取操作的原理、感知图元、实体所有者等概念进行分析讨论。显示在 OCC 中，显示服务是独立于数据结构的，显示服务的表示由合适的算法产生。这样来安排的好处是当需要对显示对象进行修改时，不用修改服务只需对描述对象的几何或拓扑算法进行修改就可以实现了。

10.1 显示结构

在屏幕上显示一个对象需要由三种实体协作完成，即：可显像的交互对象、浏览器和交互环境，在 OCC 中分别由三种类实现，AIS_InteractiveObject、Viewer 和 AIS_InteractiveContext。以下是较为详细地说明这些类及相关包的用法。

1、可显像的交互对象类：可显像的交互对象类的用途就是提供给要显示对象的 Graphic2d 或 Graphic3d 图形结构。首先，根据显示请求，可显像的交互对象类会调用合适的算法来产生这种结构并保持结构框架，为显示做好准备。

在 StdPrs 和 Prs3d 包中已提供了标准的显示算法。然而，我们也可以自定义一些具体的显示算法，然后在 Graphic2d 或 Graphic3d 包中创建对象的结构。我们也可以为单个可显像的交互对象创建一些显示算法，但对于每一个可视化模式的算法是一定要支持应用程序的。当然，显示的对象要具有可显像的特性或与可显像的交互对象相关。

2、浏览器类：允许用户交互地管理对象的视图。当在一个视图中进行缩放、转换或旋转等操作时，浏览器不关心所应用的数据模型，只考虑可显像对象的 Graphic2d 和 Graphic3d 图形结构。在 OCC 中，2D 和 3D 浏览器就是用来对显示算法产生的 Graphic2d 和 Graphic3d 图形结构进行操作的。

3、交互环境类：通常，交互环境类会接受一个高级的 API 信号，然后对整个显示过程进行控制。当应用程序请求显示一个对象时，交互环境的作用就是请求将可显像的交互对象的图形结构发送给浏览器，用以对象的显示。

4、显示包：显示包有 AIS、PrsMgr、StdPrs、V3d 和 V2d。另外，如果需要执行自定义的显示算法，还需要 Prs3d、Graphic3d 和 Graphic2d 等包。

5、选取包：选取包有 SelectBasics、Select2D、Select3D、SelectMgr 和 StdSelect。

SelectBasics 包含了选取的基础类，其中，SensitiveEntity 类定义感知图元，EntityOwner 类定义感知图元的所有者，SortAlgo 类中定义用于分类包围框的算法。

Select2D 包包含了 2D 中感知图元的基础类，例如：点、线段和圆，它是从 SensitiveEntity 类中继承来的，从动态选取的角度来看，它用来显示 2D 中可选取的对象。

Select3D 包含了所有 3D 中标准的感知图元，例如：点、曲线、面等，所有这些类是从 3D 的 SensitiveEntry 类中派生来的，如果需要的话，可以用于对 2D 图形选取空间中包围框的恢复，这个包还包含 3D-2D 放映机。

SelectMgr 包用来管理整个动态选取的过程，包含 SelectableObject、Selection、SelectionManager 和 ViewSelector 等类。StdSelect 包提供了以上所述所有类的标准用法和主要工具，以此来防止开发者重定义选取的对象。ViewSelector2D 类定义了 V2d 包中的一个视图选择器，用来对视图操作。

6、应用程序交互服务（AIS）包，提供了显示和选取 3D 对象时所需的各种类。

7、PrsMgr 包，提供显示过程中需要的所有类，即 Presentation 和 PresentableObject 的抽象类，以及 2D 和 3D 显示的具体类。

8、StdPrs 包，提供实时使用的标准显示算法，这些算法是对几何和拓扑工具箱中点、曲线、形状的描述。

9、V2d 和 V3d 包，提供支持 2D 和 3D 浏览器的所有服务。

10、Prs3d 包，提供一些通用的显示算法，例如：线框、阴影和隐藏线消除与绘图器类的关系。绘图器类控制显示过程中需要创建的一些特性，例如：色彩、线型、线宽等。

11、Graphic2d 和 Graphic3d 包，提供创建 2D 和 3D 图形结构的所有资源。

12、Aspect 包，用来对图形对象的属性设置进行管理。Aspect 包提供用于图形元件的所有类，包括所有通用的 2D 和 3D 浏览器 —— 屏幕背景、窗口、边界、图形属性组等的内容，常用来描述 2D 和 3D 对象。

10.1.2 显示一个 3D 对象的例子

下面是使用 OCC 显示一个 3D 对象的典型程序：

```
Void Standard_Real dx = ...;  //x 坐标
Void Standard_Real dy = ...;  //y 坐标
Void Standard_Real dz = ...;        //z 坐标
Handle(V3d_Viewer)aViewer=...;                    //创建浏览器对象
Handle(AIS_InteractiveContext)aContext;
aContext = new AIS_InteractiveContext(aViewer);    //创建交互环境
BRepAPI_MakeWedge w(dx, dy, dz, Ctx);
TopoDS_Solid &S = w.Solid();                      //创建楔形实体对象
Handle(AIS_Shape) anAis = new AIS_Shape(S);       //创建可显像的交互对象
aContext -> Display(anAis);                       //在 3D 浏览器中显示可显像的交互对象
```

用 BRepAPI_MakeWedge 命令来创建对象的形状，然后从形状中创建出 AIS_Shape。调用显示命令 PresentableObject 时，计算机就会响应交互环境的请求，调用算法计算需要的显示数据，将计算结果传递给浏览器。处理过程如图 2-1 所示。

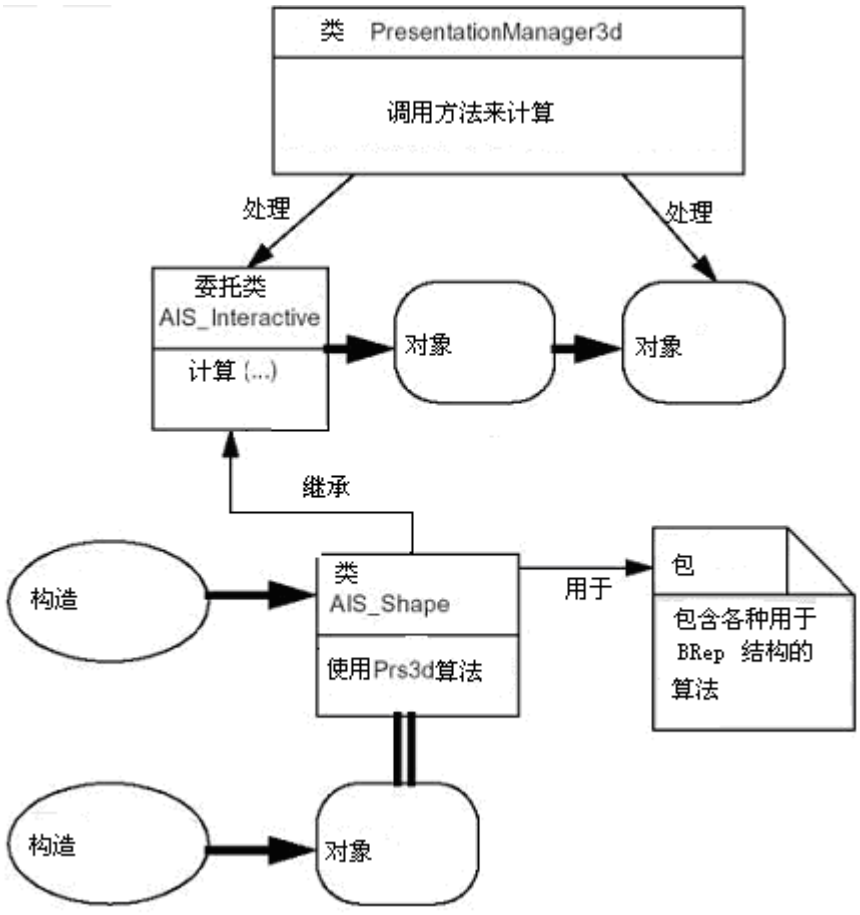


图 2　1 可显像形状的显示过程

10.2 选取操作

10.2.1 基本原理

我们能灵活地对对象进行选取和显示，是因为事先就设定好了一组感知图元。它给 2D 图形空间提供了感知区域。当我们执行选取操作的时候，这些区域会根据鼠标在屏幕上移动的位置又被划分成不同的区域，从而实现精确选取。

鼠标的位置与感知区域是紧密相关的。当在窗口内移动鼠标时，计算机会分析两者接触的程度，然后感知区域内的实体所有者就会处于高亮状态或对象的相关信息被高亮显示在一个清单里。这样，就实现了对元件身份的检测。选取操作的具体实现过程如图 2-2、2-3、2-4 和 2-5 所示。

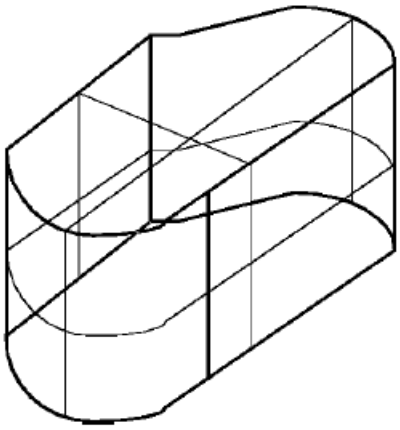


图 2　2 几何模型

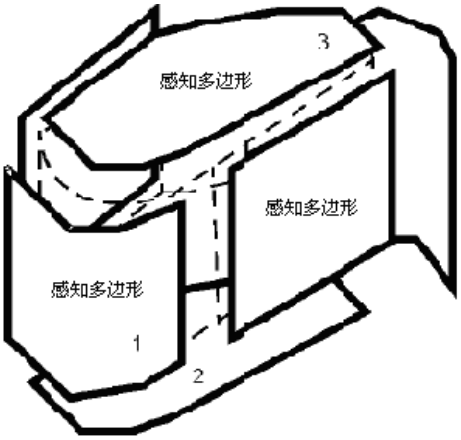


图 2　3 感知图元建模面

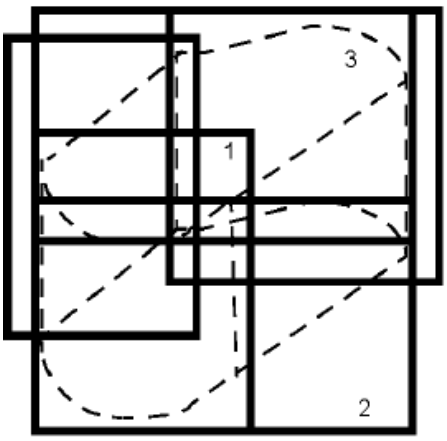


图 2　4 在动态选取中，

用包围矩形来描述感知多边形

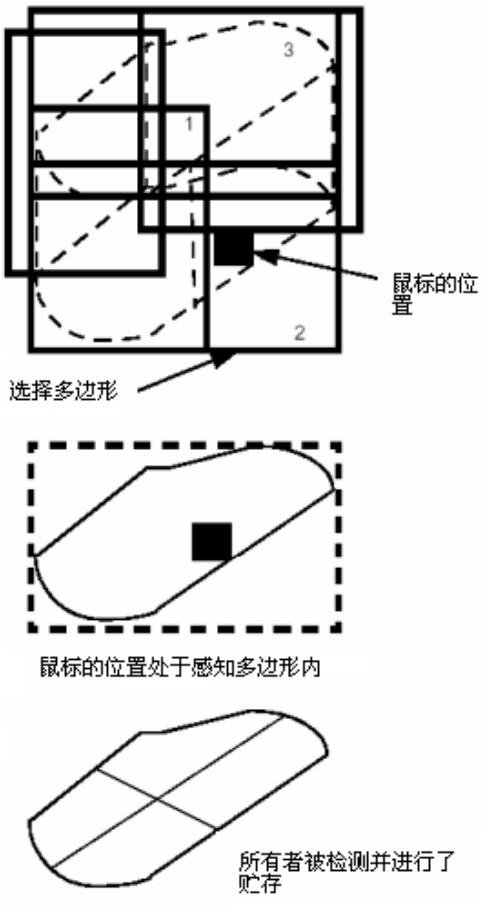


图 2　5 参照感知图元和所有者

10.2.2 感知图元

感知图元与实体所有者是一体的，这样就可以根据可选取对象的组成情况来

对感知图元进行定义了。当然感知区域是需要定义到 2D 包围框中的，对于对象的动态选取，它必须描述成感知图元的特性或是对象本身的某种身份。至于它们如何被限定在这个 2D 框里，是由一套算法来实现的。

使用 2D 包围框可以对检测到的实体进行预选取。预选取之后，算法会核对实际被检测到的感知图元，一旦被检测，图元就会自动提供所有者的身份。 如图 2-6 所示，这条感知线段已被限定到选择器的包围框里。选取操作时，包围框能检测到鼠标的位置 1 和 2，但进一步识别之后，算法才能给出最佳的选择位置，也就是说，位置 2 的线段才最合乎选择。

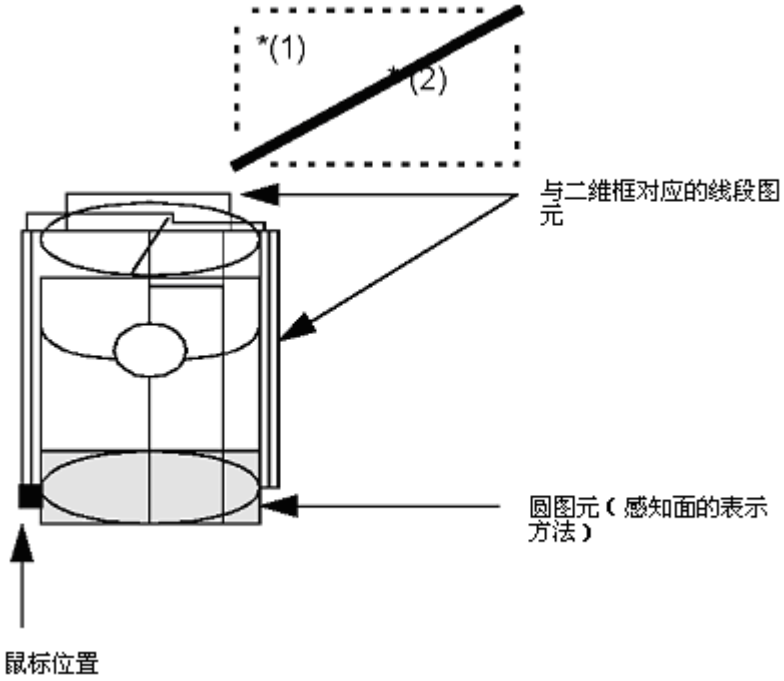


图 2　6 感知图元的例子

当 2D 包围框检测到感知图元与鼠标的位置相一致时，感知图元就会被激活并高亮显示需要显示的对象。感知图元的概念对于开发者来讲是至关重要的，这主要体现在为感知图元定义选取模式类的时候，这些类必须包括区域和匹配的函数，在作预选取的时候，前者将提供给 2D 包围框对象的感知图元，后者则定义在 2D 框内被检测到的感知图元是否是有效。

10.2.3 动态选取

动态选取时，以下操作必须提前执行。

1、在执行特定的感知图元时，如果定义在 SELECT2D 和 SELECT3D 中的图元不足够充分，那么需要的图元就必须从 SelectBasics 包的 SensitiveEntity 类

或 Select3D 包的 SensitiveEntity 类中继承过来，从 3D 投影到 2D 的过程中这是必需的。

2、定义所有者的类型及可选取对象的类用来描述感知交互的图元，以及将其计算的方法进行分解的方式可能有许多种。但如果对象仅有一种唯一的可选取的方法，那么系统只会默认一种方式。安装的过程就是在循环选取的时候提供给用户被检测实体的所有者身份。

当以上这些步骤完成之后，接着就要执行以下的过程：

1、创建一个交互环境。

2、创建可选取的对象和计算他们的各种选取参量。

3、将这些可选取的对象导入到交互环境中。对象对于所有的选择器来说是公有的，因此可以被选择器访问。

4、对象选取模式中对象处于活动或不活动状态是这样来实现的：处于活动状态时，管理器发出命令让选择器提取感知图元，如果是从 3D 投影到 2D，选择器就不得不请求用具体的方法来执行这些操作。这一阶段，选择器中选取的实体是可以利用的。循环选取会不停的通知选择器当前鼠标的位置并随时对实体进行检测。

动态选取促使在视图空间内的对象随着鼠标指针的移动自动处于高亮状态。这样，在一定程度上我们就可以正确的选取对象了。动态选取是建立在以下两个概念的基础之上的：可选取的对象和交互环境。可选取的对象描述了一些已给定的选取模式，这些模式可以在选择管理器中被重定义，它可以是活动的或是不活动的；交互环境是用来管理可选取的对象以及选取过程的，对可选取的对象而言，选取模式可能是活动的或是不活动的，这由选取方式中的选择器来通知。活动的对象选取之前是需要计算的，一旦处于非活动状态时选择器就会对其进行选取操作。

动态选取的思想就是在实际选取视图的 2D 空间通过包围框显示我们想选取的实体。这些区域的设置是由功能强大的类几何运算实现的，然后通过检测鼠标的位置找到适合的实体。这一过程一定要做的是读取视图中鼠标的位置（X,Y）处的矩形框，同时又要聪明地消除实体本身具有的矩形框。

可选取的对象被导入到选择管理器中，而选择管理器可以有多个，最好为每个浏览器分配一个选择器，这样一来之后我们需要做的就是让可选取的对象的选取模式变为有效或无效。SelectionManager 关心的是调用 ComputeSelection 函数

来计算不同的对象。对于每个选取模式，SelectMgr_Selection 对象类已经被包含在可选取的对象中了（每个选取模式为每个被定义的可选取对象类建立选取优先级）。

选取的概念和显示的概念相似，显示包含了一组图形元件允许在具体的显示模式中显示实体，而选取则包含了一组感知图元允许对相关联的实体进行检测。

显示过程中有一个中间阶段，那就是要让选择器知道某些高级别实体以及其用于感知交互的集合图元的可选取方式。

感知图元自身具有的功能有：把 2D 包围框传给选择器；通过匹配函数断然地或否定地应答拒绝标准；如果需要可以从 3D 中投影到 2D 视图空间；返回根据选取表示出来的所有者。

一组标准的感知图元存在在 Select3D 包中用于 3D 的感知，同样存在在 Select2D 中用于 2D 的感知。所有者是个实体，它是我们真正想检测的。所有者需要与感知图元连接起来，它本身就拥有很多信息，通过这些信息就可以很容易地找到相关对象。对于一个正在使用的所有者而言它有无以伦比的可选取权。从图 2-7 中可以看出实体、所有者、图元三者间的关系。

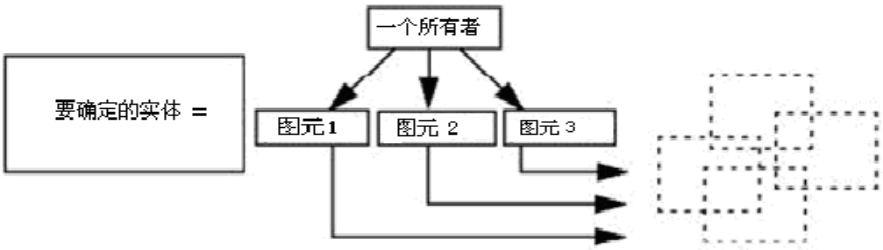


图 10　7 在交互的/可选取的对象中需要实现的过程

通过每个活动的选取模式来定义我们想确定的选取模式的数目，例如对一个交互对象用一个拓扑类型来显示，首先要给定如下选取模式：

模式 0：交互对象自身的选取

模式 1：顶点的选取

模式 2：边的选取

模式 3：线的选取

模式 4：可检测面的选取

对于交互对象的选取模式，实体的模型是通过感知图元来确定的。SelectMgr_ EntityOwner 是实体的根类，它包含了一个引用的可选取的对象。如果想要储存它的信息，就一定得创建派生自这个根类的新类，例如，对于形状来说就有 StdSelect_BRepOwner 类，它可保存在 TopoDS 形状中，和交互对象一样

占据了一个场所。

为一个已给的模式设置的感知图元已经被计算并保存在 SelectMgr_Selection.中了。对于交互对象，其建模过程是由 ComputeSelection 虚函数来执行。让一个交互对象显示在一个框内，模式 0（整个框的位置）和模式 1（包围框的位置）是我们需要关注的。

在第一个模式下，所有的感知图元都将用来显示本身的交互对象，而在第二个模式下，我们必须为每个边创建一个所有者，并且包含了用于显示边的标记。我们也可以创建一个从 SelectMgr_EntityOwner 派生的所有者类。用函数 Compute- Selection 来计算交互框的代码如下：

```
void InteractiveBox::ComputeSelection(const Handle(SelectMgr_Selection)& Sel,
    const Standard_Integer Mode)
{
    switch(Mode)
    {
        case 0:   //通过创建感知面来定位整个框
            Handle(SelectMgr_EntityOwner) Ownr = new SelectMgr_EntityOwner(this,5);
            for(Standard_Integer I=1;I<=Nbfaces;I++)
            {
                //数组是一个 TColgp_Array1OfPnt 类型的: 用它来描述顶点的排列。感知
                  值是 Select3D_TypeOfSensitivity 类型值。
                Sel->Add(new Select3D_SensitiveFace(Ownr,Array,Sensitivity));
            }
            break;
        case 1:   //定位边
            for(Standard_Integer i=1;i<=12;i++)    //给每个边创建一个所有者
            {
            Handle(mypk_EdgeOwner) Ownr = new mypk_EdgeOwner(this,i,6);  //优先考虑
            6
            Sel->Add(new Select3D_SensitiveSegment(Ownr,firstpt(i),lastpt(i)));
            }
            break;
    }
}
```

需要注意的是，这一过程如果使用 AIS 的交互环境，是完全不可见的，以下代码说明了这样的交互过程：

```
                                                    //有三个交互框 box1, box2, box3
    Handle(SelectMgr_SelectionManager) SM = new SelectMgr_SelectionManager();
    Handle(StdSelect_ViewerSelector3d) VS = new StdSelect_ViewerSelector3d();
                                                    //将三个选择框加载到选择器中
    SM -> Add(VS);
```

```
SM -> Load(box1);
SM -> Load(box2);
SM -> Load(box3);
                                                    //在选择器 VS 中激活框 1 的模式，
                                                    选模式为 0
SM->Activate(box1,0,VS);
SM->Activate(box2,1,VS);
SM->Activate(box3,1,VS);
VS -> Pick(xpix,ypix,vue3d) ;                       //根据鼠标的位置检测图元

Handle(EntityOwner) POwnr = VS->OnePicked();        //拾取检测到的最佳所有者
for(VS->Init();VS->More();VS->Next())
      {
        VS->Picked();                               //拾取检测到的所有所有者
      }
SM->Deactivate(box1);                               //取消框 1 中所有活动的模式
```

10.3 应用示例

假定我们要创建一个在 V3d 包的浏览器中显示房子（House）的应用程序，需要在图形窗口中选取房子或房子的部件，房间（Room）、墙（Wall）和门（Door）等，则操作步骤如下：

首先，定义一个可选取的对象 House，并提供给这个对象以下四种选取模式之一：选取 house 自身、选取 room、选取 wall、选取 door。

其次，写出计算这四种选取模式的方法，并申明在这些模式下，感知图元是活动的。当然，还必须定义详细的所有者类，这个类继承于 selectMgr 包中的 EntityOwner 类，它应包含显示房子组件的参数：house、wall、door、room。

如图 2-8 所示是我们创建的房子，图 2-9、2-10、2-11、2-12 是执行不同选取操作的结果。

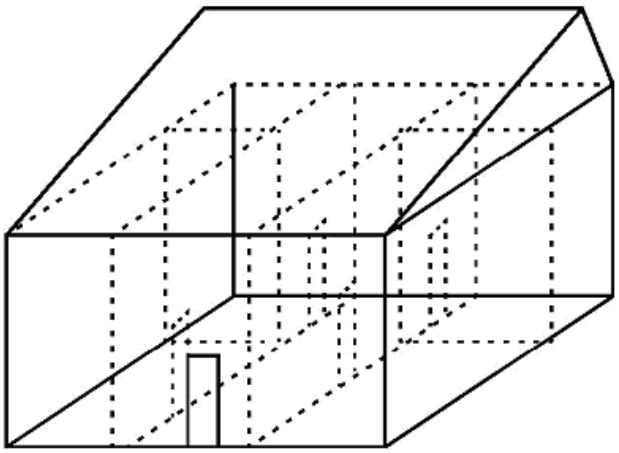


图 2　8 选取房间

创建选取的时候一定要与其选取模式保持一致。如果我们想创建模式 2 下的选取，即对 room 的选取。程序如下：

```
Void House::ComputeSelection(
    const Handle(SelectMgr_Selection)& Sel,
    const Standard_Integer mode)
{
    switch(mode)
    {
        case 0: //房间的选取
            for(Standard_Integer i = 1; i <= myNbRooms;i++)
            {
                //对每个房间, 创建一个所有者
                //要包含给定的房间和它的名字
                Handle(RoomOwner) aRoomOwner = new RoomOwner (Room(i),
                NameRoom(i)); //Room()返回房间序号，NameRoom()返回房间的名称
                Handle(Select3d_SensitiveBox) aSensitiveBox;
                aSensitiveBox = new Select3d_SensitiveBox(aRoomOwner, Xmin, Ymin, Zmin,
                Xmax, Ymax, Zmax);
                Sel -> Add(aSensitiveBox);
            }
            break;
        case 1: ... //门的选取
      }//Switch
  } // ComputeSelection
```

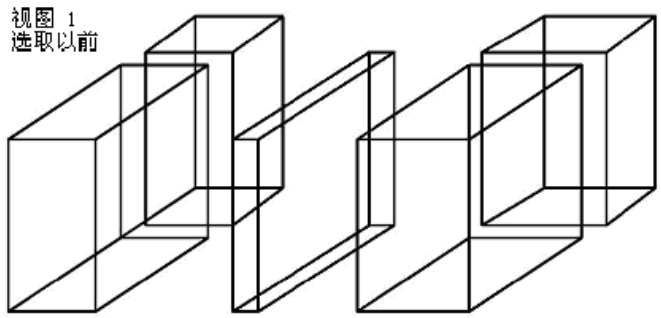


图 2　9 活动的感知框与选取模式 0 相一致（选取房间）

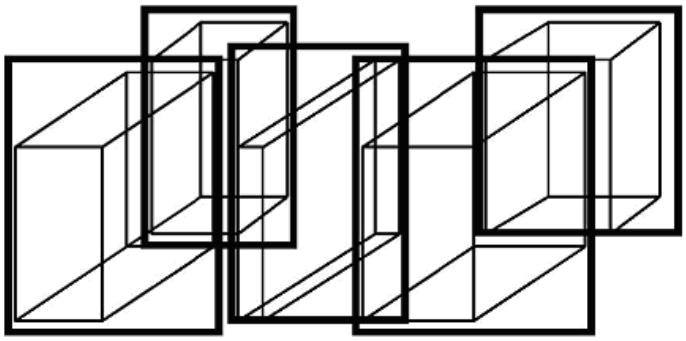


图 2　10 在视图 1 中，作动态选取时选择器中活动的感知矩形

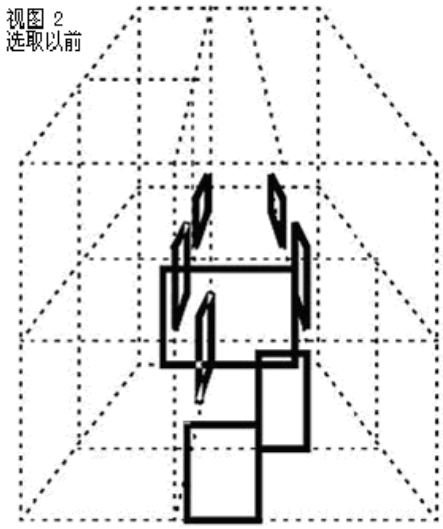


图 2　11 活动的感知多边形与选取模式 1 相一致（选取门）

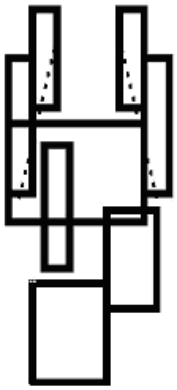


图 2　12 在视图 2 中，作动态选取时选择器中的感知矩形

10.4 本章小结

本章对 OCC 中可视化的基础知识进行了详细分析。包括显示对象所必需的三种实体：可显像的交互对象、浏览器、交互环境；显示包、选取包、标准显示算法包、AIS 包以及 2D、3D 图形资源包 Graphic2d、Graphic3d 等，并对这些包的用法进行了说明。然后讨论了选取操作，其原理是：在创建对象之初将对象的图形结构定义为感知图元，显然感知图元是一组用于感知交互的图形集合元素，它的创建为 2D 图形空间提供了感知区域。当在视图窗口内移动鼠标时，算法会分析鼠标位置与感知区域的接触程度，从而使感知区域内的实体所有者处于高亮状态。实体所有者与图形对象是一体，这样就实现了对对象的选取。

动态选取原理是建立在可选取的对象和交互环境基础上的。其思想是在实际选取视图的 2D 空间通过包围框显示想选取的实体。选取的概念和显示的概念是类似的。这好比显示包含了一组图形元件允许在具体的显示模式中显示实体，而选取则包含了一组感知图元允许对相联系的实体进行检测。

第十一章

第 4 章　OCC 可视化的应用程序交互服务 AIS

应用交互服务 AIS 包提供的服务是 OCC 可视化的核心内容，这些服务与数据结构和交互对象是紧密相关的，实现对图形对象的显示、检测及选择等过程的管理。为此，本章对 AIS 提供的服务进行全面分析。

11.1AIS 包

AIS 包是一个高级的界面，可以访问底层的显示和选择服务。AIS 扩展了在 GUI 浏览器中自定义的标准三维的选择属性、显示处理及显示属性等优越的功能并对它们进行管理。为了实现这些服务，AIS 包包含了交互对象、交互环境、图形属性管理器、选择过滤器类。

1、交互对象（AIS_InteractiveObject）。所谓的实体就是被选择和可视的交互对象。在 AIS 中，已经定义好了标准交互对象类的功能函数，我们只需要遵守一定的规定和协议就可以利用它来执行自定义的交互对象类。

2、交互环境/当前环境（AIS_InteractiveContext）。引导实体的选取和可视化的中央控制单元就是交互环境。它能连接到主要的浏览器实现以下两种操作模式：Neutral Point、当前可视化和选取的环境。Neutral Point 是默认方式，允许对交互对象的选取和可视化，并将对象导入到环境中。打开当前环境的方式是为临时选取作准备的，这样做就不会干扰在 Neutral Point 中的选取了。其中提供的函数就是为了执行选取的交互对象或是选取模式或是临时的可视化等。当所有的这些操作都就完成以后，我们需要做的工作就是关闭当前正在使用的环境使其回到打开以前的状态。

3、图形属性管理器或绘图器。如图 11-1 所示，Prs3d_Drawer 类派生了 AIS_Drawer 类。交互对象本身来说好像就是具体的图形属性（如可视化模式、色彩、材质等），交互环境中有一个绘图器在默认状态下是有效的，用它来对对象的控制。当交互对象被可视化时，图形属性就会要求自身的绘图器来检查在当前环境下是否有效。

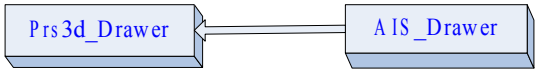


图 11　1 Prs3d_Drawer 是 AIS_Drawer 的父类

4、选择过滤器。在选取操作时对需要选取的实体进行过滤是非常必要的，因此，实体过滤器允许对动态检测环境以便精确定位，这样才能达到目的。有些

过滤器可能会用在 Neutral Point 中使用，而其他的只会用在打开的当前环境中。用户也可以自定义过滤器并将这些过滤器导入到交互环境中。图 11-2 列出了 SelectMgr_Filter 的派生类。

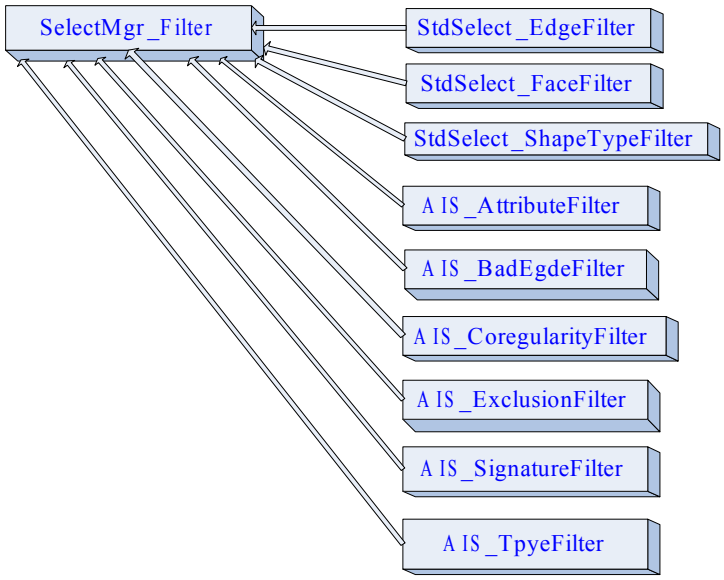


图 11　2 SelectMgr_Filter 的派生类

11.2 交互对象的管理

交互对象是个虚的实体，但可以对它进行显示和选取操作，因此它也有自己的可视化特性，例如色彩、材质和可视化模式。为了灵活的创建和执行交互对象，必须理解相关规范和协议，同时，要想让交互对象按照要求的方式活动，还需要执行一些虚函数。要执行一些已定义好的标准交互对象，就必须遵守 AIS 中的规范和协议，有关显示、选取和图像属性等操作的服务应当单独来考虑。

在 OCC 中，通过从一些主要的类派生出子类来实现不同的功能，图 11-3 列出了 PrsMgr_PresentableObject 类的派生类。

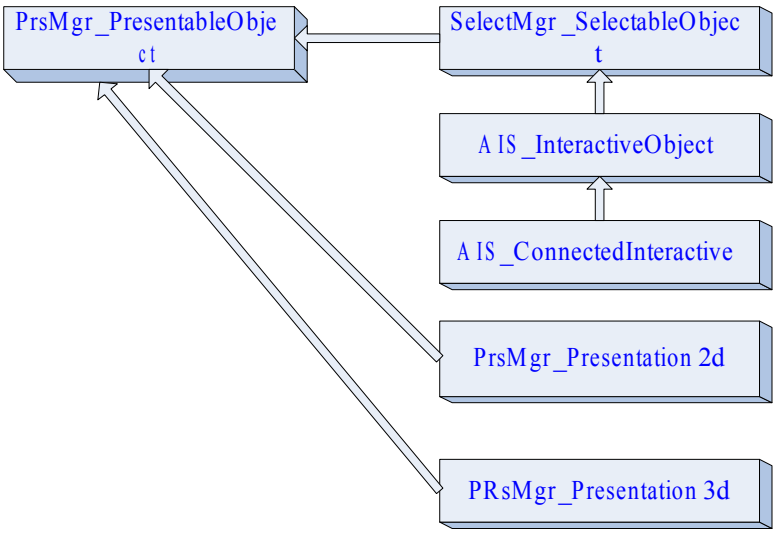


图 11　3 一些派生类

11.2.1 交互服务协定

交互服务协定如下：

1、在 2D 或 3D 中，一个交互对象可能被创建了许多次，而创建的时候就赋予了不同的意义。

2、3D 显示由 PresentationManager3D 管理；2D 显示是由 PresentationManager2D 管理。在 AIS 中这是显而易见的，用户根本不用为此担心。

3、显示管理器中，一种显示已经由一种标志和参数确定好了。在协定中已经规定，交互对象的默认显示模式是标志 0。

4、虚函数：交互对象的不同显示计算是由 Compute 函数来实现的。它继承于 PrsMgr_ PresentableObject::Compute 函数。如果我们要创建自定义的交互对象，必须按下面所述的方式执行 Compute 函数。

（1）对于 2D，例如：

```
void PackageName_ClassName::Compute (
const Handle( PrsMgr_PresentationManager2d) & aPresentationManager,
const Handle(Graphic2d_)& aGraphicObject,
const Standard_Integer aMode = 0 );
```

（2）对于 3D，例如：

```
void PackageName_ClassName::Compute (
const Handle(PrsMgr_PresentationManager3d)&aPresentationManager,
const Handle(Prs3d_Presentation)& aPresentation,
  const Standard_Integer aMode = 0 );
```

（3）对于 3D 中隐藏线模式，例如：

```
void PackageName_ClassName::Compute (
const Handle(Prs3d_Projector)& aProjector,
```

const Handle(Prs3d_Presentation)& aPresentation);

11.2.2 基本算法

由于遵循这些协定的使用在视图中是由系统自动安排的，因此，这种功用需要说明一下。视图有两种状态：衰退模式（通常的模式）和不衰退模式（隐藏线模式）。当后者起作用的时候，视图会寻找所有可显像的交互对象并将其显示在通常的模式下。如果接受了隐藏线模式的显示，那么会有信息来提示用户。有个内在的机制就是允许我们请求调用交互对象的计算，放映机、函数等。也许我们还不知道在隐藏线模式中这种机制是否还是有效的呢？协定中规定了交互对象可以接受也可以拒绝隐藏线模式下的显示。所以，我们可以随便申明两种方法中的一种，或者一开始就使用 PrsMgr_TypeOfPresentation 中列举的一种定义。如：

PrsMgr_TOP_AllView

PrsMgr_TOP_ProjectorDependant

或者使用以下的函数：

PrsMgr_PresentableObject::SetTypeOfPresentation

AIS 中交互对象有四种类型：构建的元件或数据、关系（尺寸和约束）、对象、无类型的对象。在这些种类中，也是可以采用额外的一种标记来描述。在默认状态下，交互对象是无类型的，这和给予一个 0 标记的效果是一样。如果想给交互对象一个特殊的类型或标记，需要重定义如下两个虚函数：

AIS_InteractiveObject::Type

AIS_InteractiveObject::Signature

一些标记已经被 AIS 中的标准对象使用。交互环境可能默认下接受了交互对象的显示模式，但这种模式是不会被一个已给定的对象类接受的。因此，需要执行以下的虚函数来获取对象类的信息。

AIS_InteractiveObject::AcceptDisplayMode

显示模式：一个对象可能已经有自己的临时显示模式，但这与交互环境提供的模式不同。要想给对象设置显示模式，可以用以下的函数来实现：

AIS_InteractiveContext::SetDisplayMode

AIS_InteractiveContext::UnsetDisplayMode.

要改变默认的标志，则使用这个虚函数：

AIS_InteractiveObject::DefaultDisplayMode.

动态检测时，在交互环境的作用下，对象会在默认方式下就可以显示到屏幕

上。我们可以不考虑对象默认的显示模式而将其指定在高亮显示模式下，那么使用以下的函数：

AIS_InteractiveObject::SetHilightMode

AIS_InteractiveObject::UnSetHilightMode

如果一种形状已经显示在线框区或阴影区，而我们想让它在线框区处于高亮状态，那么可以在交互对象的构造器里设置模式为 0，此时务必忘记计算函数对这种显示模式造成的影响。

无限身份：如果不想让对象受到 FitAll 视图的影响，一定要申明它的无限性。当然，也可以取消它的无限身份，实现函数如下：

AIS_InteractiveObject::SetInfiniteState

AIS_InteractiveObject::IsInfinite

例：用一个叫 IShape 的类来描述一个交互对象

```
myPk_IShape::myPK_Ishape (
    const TopoDS_Shape& SH,
    PrsMgr_TypeOfPresentation aType): AIS_InteractiveObject(aType),
    myShape(SH),
    myDrwr(new AIS_Drawer() )
    {
      SetHilightMode(0);
    }

  void myPk_IShape::Compute(
    const Handle(PrsMgr_PresentationManager3d) & PM,
    const Handle(Prs3d_Presentation)& P,
    const Standard_Integer TheMode )
    {
       switch (TheMode)
       {
            case 0: StdPrs_WFDeflectionShape::Add (P,myShape,myDrwr);  // 计算线框
                    break;
             case 1: StdPrs_ShadedShape::Add (P,myShape,myDrwr);  //计算阴影显示
                    break;
        }
    }

  void myPk_IsShape::Compute (
    const Handle(Prs3d_Projector)& Prj,
    const Handle(Prs3d_Presentation) P )
    {
```

```
    StdPrs_HLRPolyShape::Add(P,myShape,myDrwr);   //隐藏线模式的计算方法
}
```

11.3 选取算法

11.3.1 选取协定

1、交互对象可以有不确定的选取模式，它们都被保存在 SelectMgr_Selection 类中。在 AIS 中，用感知图元来描述对象，每个图元都有自身的所有者（SelectMgr_EntityOwner）；用标志来指定选取模式，使感知图元与给定选取模式相呼应。这种一对一的映射方式提高了检测实体时的准确性。协定规定，默认选取模式是 0。

2、感知图元的选取计算由虚函数来实现，即 ComputeSelection.。以下函数是对每个交互对象类执行不同的选取操作时需要使用。

AIS_ConnectedInteractive::ComputeSelection

3、正如使用最频繁的实体是 TopoDS_Shape 类型一样，频繁使用的交互对象是 AIS_Shape 类型。为了创建与 AIS_Shape 类成员行为一样的交互对象的新类如顶点、边等，需重定义虚函数：

AIS_ConnectedInteractive::AcceptShapeDecomposition

11.3.2 基本算法

交互对象的默认选取模式标志是可以改变的。通常我们会根据需要使用不同的函数来实现，比如需要：

1、检查是否有已存在一种选取模式。

2、检查当前的选取模式。

3、设置一种选取模式。

4、不设置选取模式。

执行以下函数：

AIS_InteractiveObject::HasSelectionMode

AIS_InteractiveObject::SelectionMode

AIS_InteractiveContext::SetSelectionMode

AIS_InteractiveContext::UnsetSelectionMode

同样的方法，可以临时改变某些已设定了选取模式 0 的交互对象的优先级，比如需要：

1、检查交互对象本身是否有选取的优先级。

2、检查当前的优先级。

3、设置优先级。

4、不设置优先级。

执行以下函数：

AIS_InteractiveObject::HasSelectionPriority

AIS_InteractiveObject::SelectionPriority

AIS_InteractiveObject::SetSelectionPriority

AIS_InteractiveObject::UnsetSelectionPriority

动态检测和选取的实现方式是比较直接的，我们需要熟知相关协定和实现函数。在 Neutral Point 和当前环境中以下函数所起的效果是相同的。

AIS_InteractiveContext::MoveTo —— 将鼠标的位置传递给选择器。

AIS_InteractiveContext::Select —— 保存最后检测到的对象，替换之前选取的对象。如果最后仍未检测到对象就保持空状态。

AIS_InteractiveContext::ShiftSelect —— 如果最后检测到的对象仍未选取，那么将它添加到选取列表中。有时检测到的对象不符合要求，那么即使我们在一个空的区域不断点击鼠标，也不会有任何事件发生。

AIS_InteractiveContext::Select —— 选取在周围能找到的每一个对象。

AIS_InteractiveContext::ShiftSelect —— 选取当前仍未列入已选取、未选取列表的对象。

在 Neutral Point 和当前环境中检测和选取实体时的高亮显示是自动由交互环境管理，高亮显示的颜色也由它来处理。当然，用户想通过自己的方式来管理高亮显示，也可以取消自动模式，实现函数是：

AIS_InteractiveContext::SetAutomaticHilight

AIS_InteractiveContext::AutomaticHilight

需要注意的是，当打开一个交互环境时，可以选取那些除了从标准模式中继承过来的交互对象（顶点、边）和在具体模式中活动的实体。只有交互对象才被储存在已选取对象列表中的，因此可以通过移动鼠标来请求交互环境执行相关操作。比如需要：

1、通知是否已经检测到实体。

2、通知实体的形状。

3、获得实体的形状。

4、如果检测到实体，那么就获取交互对象。

执行以下函数：

AIS_InteractiveContext::HasDetected

AIS_InteractiveContext::HasDetectedShape

AIS_InteractiveContext::DetectedShape

AIS_InteractiveContext::DetectedInteractive

在 Neutral Point 中使用 Select 和 ShiftSelect 函数以后就可以查看到选取的列表了，列表中的对象就作为当前对象来使用。比如需要：

1、开始扫描这个列表。

2、广泛扫描。

3、重新扫描。

4、获取扫描中检测到的当前对象的名字。

执行以下函数：

AIS_InteractiveContext::InitCurrent

AIS_InteractiveContext::MoreCurrent

AIS_InteractiveContext::NextCurrent

AIS_InteractiveContext::Current

如果需要：

1、获得当前的第一个交互对象。

2、高亮显示当前对象。

3、从当前对象上移除高亮显示。

4、为了更新，可以清空当前对象的列表。

5、找到当前的对象。

那么，执行以下函数：

AIS_InteractiveContext::FirstCurrentObject

AIS_InteractiveContext::HilightCurrents

AIS_InteractiveContext::UnhilightCurrents

AIS_InteractiveContext::ClearCurrents

AIS_InteractiveContext::IsCurrent.

在当前环境中，可以查看已选取的、可利用的对象的列表。如果需要：

1、开始扫描这个列表。

2、广泛扫描。

3、重新扫描。

4、获取可选取对象的名字。

那么，执行以下函数：

AIS_InteractiveContext::InitSelected

AIS_InteractiveContext::MoreSelected

AIS_InteractiveContext::NextSelected

AIS_InteractiveContext::SelectedShape.

还可以：

1、核实是否有一个已选取的形状。

2、获得已选取的交互对象。

3、核实适合的对象在交互中是否存在一个所有者。

4、获取检测到的实体的合适所有者。

5、获取已选取对象的名字。

执行以下函数：

AIS_InteractiveContext::HasSelectedShape

AIS_InteractiveContext::Interactive

AIS_InteractiveContext::HasApplicative

AIS_InteractiveContext::Applicative

AIS_InteractiveContext::IsSelected.

使用虚函数的例子如下：

```
myAISCtx->InitSelected();
while (myAISCtx->MoreSelected())
      {
       if (myAISCtx->HasSelectedShape)
          {
           TopoDS_Shape ashape = myAISCtx->SelectedShape();
           //可以使用已拾取到的形状
           }
       else
           {
              Handle_AIS_InteractiveObject aniobj= myAISCtx->Interactive();
            //可以使用已拾取到的交互对象
           }
```

```
myAISCtx->NextSelected();
}
```

11.3.3 交互对象的图形属性

每个交互对象可能有自身的可视化属性。在默认方式下，交互对象的图形属性是从交互环境的图形管理器中获取的，用来设置可视化模式、显示计算的偏差值、参量的数目、色彩、线型、材质等。在 AIS_InteractiveObject 抽象类中，一些标准的属性已经被给予了特殊的定义。这些定义包括色彩、线宽、材质和透明度等，设置这些属性，可以使用一些虚函数来实现。如果是交互对象的新类，那么必须重定义这些虚函数以便让它能产生一些需要的行为。如图 11-4、11-5 所示是函数被重定义的情况。

图 11　4 重定义 AISPoint:SetColor 函数

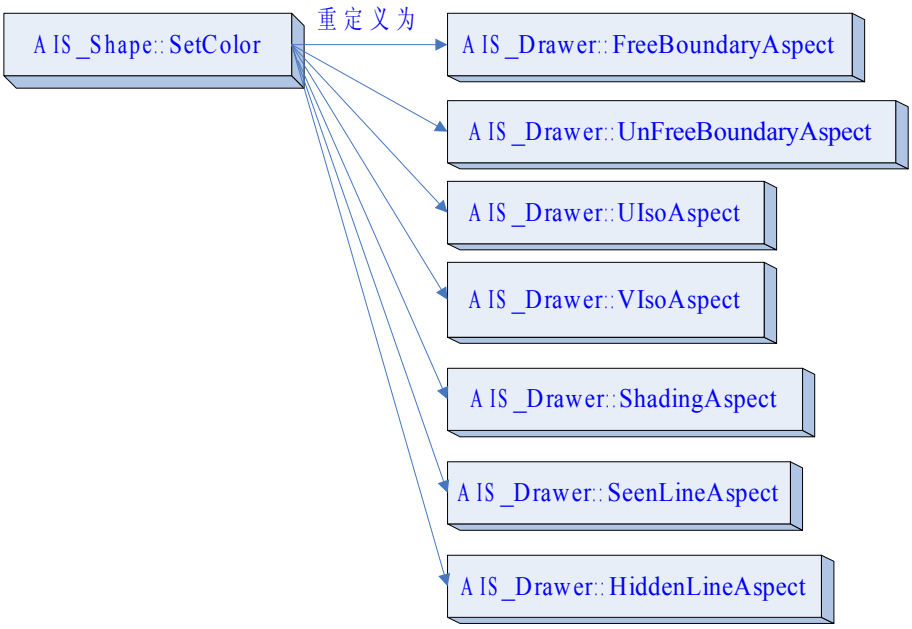


图 11　5 重定义 AIS_Shape::SetColor 函数

这些虚函数允许我们对色彩、线宽、材质及透明度进行设置。可能用到的函数还有：

AIS_InteractiveObject::UnsetColor

AIS_InteractiveObject::SetWidth

AIS_InteractiveObject::UnsetWidth

AIS_InteractiveObject::SetMaterial

(const Graphic3d_NameOfPhysicalMaterial & aName)

AIS_InteractiveObject::SetMaterial

(const Graphic3d_MaterialAspect & aMat)

AIS_InteractiveObject::UnsetMaterial

AIS_InteractiveObject::SetTransparency

AIS_InteractiveObject::UnsetTransparency

对于其它的属性类型设置，若需要改变交互对象的绘图器可以直接使用以下的函数：

AIS_InteractiveObject::SetAttributes

AIS_InteractiveObject::UnsetAttributes

11.3.4 属性管理

一些虚函数的功能可能已经暗指了显示对象的循环算法，我们需要它的用法。要想更新交互对象的显示模式，需要指明一个包括在PrsMgr_PresentableObject类中的标记。更新模式，直接使用AIS_InteractiveContext中提供的函数 Display 和 Redisplay。虚函数允许我们在视图环境中不用重复计算就可以临时转化到对交互对象的选取和显示操作。合适的实体与交互对象联系时执行以下的函数：

AIS_InteractiveContext::SetLocation

AIS_InteractiveContext::ResetLocation

AIS_InteractiveContext::HasLocation

AIS_InteractiveContext::Location

定义每一个交互对象都需要由相应的函数来实现，这样临时与对象的所有者取得联系也变得很容易，交互对象能否与合适的实体所有者联系到一起，需要以下的函数来执行它的行为。

AIS_InteractiveObject::SetOwner

AIS_InteractiveObject::HasOwner

AIS_InteractiveObject::Owner

11.4 交互环境

11.4.1 基本规定

交互环境以一种清晰的方式允许用户以一种或多种浏览器来管理交互对象的图像或可选取的属性。大多数的函数也允许修改交互对象的属性，这将在以后的章节中作进一步介绍。

在交互环境中已经介绍了如何对交互对象进行修改，但值得注意的是执行这些功能一定得用定义交互环境的函数来实现。当然，如果还没有导入到交互环境中的交互对象，可以调用其它的函数来实现。

以下的例子是用来说明以上所述内容的。

```
Handle (AIS_Shape) TheAISShape = new AIS_Shape (ashape);
myIntContext->Display(TheAISShape);
myIntContext->SetDisplayMode(TheAISShape ,1);
myIntContext->SetColor(TheAISShape,Quantity_NOC_RED);
                                                    //也可以这样来写
Handle (AIS_Shape) TheAISShape = new AIS_Shape (ashape);
TheAISShape->SetColor(Quantity_NOC_RED);
TheAISShape->SetDisplayMode(1);
myIntContext->Display(TheAISShape);
```

在交互环境中一定要区分以下两种情况：

1、没有打开当前环境；就要作为在 Neutral Point 中来使用。

2、虽然打开了一个或几个环境，但每个环境表示的只是选取和显示的一个临时状态而已。一些函数只能在打开的当前环境中使用；一些只能在关闭的当天环境中使用；另一些在同一种状态下就与其它的有不同的行为。交互环境由许多的函数组成，这些函数是根据某种需要才组合在一起。

11.4.2 交互环境的管理

交互环境由一个主要的浏览器、一个随意的 trash bin 或一个“Collector”浏览器组成。它有一组可调整的设置允许对选取和显示的行为个性化设置，具体说明如下：

1、默认的绘图器包括了所有被用在交互对象上的色彩和线的属性，本身并没有其他的属性。

2、对于交互对象默认的可视化模式是：0。

3、实体高亮状态的色彩是通过鼠标的移动来检测的。默认值为：

Quantity_NOC_CYAN1。

4、预选取色彩的默认值为: Quantity_NOC_GREEN。

5、选取色彩 (当用鼠标点击一个检测到的对象时)，默认值为：Quantity_NOC_GRAY80。

6、辅助的亮度色彩，默认值为：Quantity_NOC_GRAY40。

所有这些设置也可以被适合交互环境使用的函数修改。当我们需改变一个附属于交互环境的图形属性（比如可视化模式）时，所有交互对象在这种模式下的属性是不会被更正的。交互环境由一个主要的浏览器、一个随意的 trash bin 或一个“Collector”浏览器组成。它也有一组可调整的设置允许我们对选取和显示的行为个性化设置。

以下的例子是对以上内容的说明的：

```
                                        //obj1 和 obj2 是两个交互对象
TheCtx->Display(obj1,Standard_False);
                                        //False 等价于不更新浏览器
TheCtx->Display(obj2,Standard_True);
                                        //True 等价于更新浏览器
TheCtx->SetDisplayMode(obj1,3,Standard_False); TheCtx->SetDisplayMode(2);
                                        // 在模式 2 中，obj2 被可视化(如果 obj2 接
                                        受了模式 2)
                                        // obj1 只能被可视化在模式 3 中
```

主要的浏览器是与 PresentationManager3D 和 Selector3D 包有关，允许对当前可交互对象的选取和显示进行管理。

11.5 当前环境的管理

11.5.1 管理协定

1、打开一个当前的环境就是允许我们为临时的显示和选取操作作准备的，一旦关闭了当前环境，这些操作就会消失。

2、打开几个当前环境是可以的，但只有最后打开的环境才有效。当关闭了一个当前环境的时，在这之前打开的环境就开始起作用。如果已经没有打开的环境，那么就回到初始的 Neutral Point 环境模式。

3、每打开一个当前环境都会产生一个标志。这就是为什么要关闭当前环境的原因。

4、交互对象大多数情况下作为 AIS_Shape 类型来使用。因此，标准函数的用途就是容易地在一个打开的当前环境中对形状组元的选取操作好准备（如顶

点、边、面等的选取），

通常把给予形状类型对象的选取模式叫做标准活动模式。这些模式只有在打开的当前环境中才会被考虑，也只有在重定义虚函数 AcceptShapeDecomposition() 时才会对交互对象起作用，以便可以返回真值。

5、在一个当前环境中的临时对象是不会被其他的当前环境认可的，只有在 Neutral Point 中被可视化的对象才会被所有的当前环境认可。

6、在一个当前的环境中，一个临时的交互对象的状态只有在其他打开的当前环境中才能被修改。

需要注意的是，选取的具体模式针对的只是交互对象，这在主要的浏览器中定义。在收集器中，我们只能在当前打开的环境中把选取到的交互对象成功的传送给位置过滤器。

11.5.2 基本算法

打开或关闭一个当前环境用以下的函数来实现。

AIS_InteractiveContext::OpenLocalContext

AIS_InteractiveContext::CloseLocalContext

AIS_InteractiveContext::CloseAllContexts

在 Neutral Point 中的要被可视化的对象导入或移除当前环境则使用以下函数：

AIS_InteractiveContext::UseDisplayedObjects

AIS_InteractiveContext::NotUseDisplayedObjects

获得当前环境的标记，使用以下函数：

AIS_InteractiveContext::IndexOfCurrentLocal

当我们关闭一个当前的环境的时候，所有临时的交互对象可能已经被移出或删除，所有与交换环境有关的选取模式被取消，所有的过滤器也处于空的状态。

11.5.3 在 Neutral Point 中的显示算法

虽然大多数可视化函数可能被使用在 Neutral Point 和打开的当前环境两种状态中，但一定要区分开这两者间的关系。毕竟它们的行为方式是不同的。

操作模式 Neutral Point 被用在可视化交互对象的过程中，用以显示和选取一个合适的实体，实现移除、清除等的功能函数如下：

AIS_InteractiveContext::Display

AIS_InteractiveContext::Display

AIS_InteractiveContext::Erase

AIS_InteractiveContext::EraseMode

AIS_InteractiveContext::ClearPrs

AIS_InteractiveContext::Redisplay

AIS_InteractiveContext::Remove

AIS_InteractiveContext::EraseAll

AIS_InteractiveContext::Hilight

AIS_InteractiveContext::HilightWithColor

11.5.4 在当前环境中的显示算法

可以让具体选取的模式在当前环境中以不同的方式处于活动或不活动状态，在合适的模式下使用显示函数。

1、使标准模式活动，则使用此函数：

AIS_InteractiveContext::ActivateStandardMode

2、如果只打开了一个当前环境，则使用以下函数：

AIS_InteractiveContext::DeactivateStandardMode

AIS_InteractiveContext::ActivatedStandardModes

AIS_InteractiveContext::SetShapeDecomposition

在当前环境中，这种活动的效果对所有的对象与选取模式相一致。它负责将对象分解成 sub-shapes。每个被导入到交互环境中的新对象会根据这些模式自动响应标准的分解活动。

如果在默认的选项(AllowShapeDecomposition = Standard_True)下将一个对象导入到已经打开的一个当前环境中,那么所有 Shape 类型的对象也会以同样的模式被激活。可以通过使用 SetShapeDecomposition(Status)使这些标准的对象起作用。导入一个交互对象的函数如下：

AIS_InteractiveContext::Load

在一个对象上直接启动或不启动选取模式用以下的函数来实现：

AIS_InteractiveContext::Activate

AIS_InteractiveContext::Deactivate

11.5.5 过滤器的用法

当交互对象在当前环境中已经准备好的时候，就可以添加选择滤波器。滤波器的根类是 SelectMgr_Filter，添加后就可以实现其功能了。过滤器会在当前环境中检查鼠标所处的位置是否已经检测到所有者（SelectMgr_EntityOwner）。如果检测到就保持，反之就拒绝。可以执行延迟函数来创建自己的过滤器对象类。

以下是创建过滤器的例子：

```
class MyFilter : public SelectMgr_Filter { };
  virtual Standard_Boolean MyFilter::IsOk
  (const Handle(SelectMgr_EntityOwner)&anObj) const = 0;
```

在 SelectMgr 中有过滤器的组件，可以用它来组建一些过滤器。在交互环境中，所有添加了的过滤器都会储存在一个叫 OR 的过滤器中。在一些包中有标准过滤器，它们都可以使用。这些过滤器使用功能如下：

StdSelect_EdgeFilter —— 作用到像线、圆这样的边缘上。

StdSelect_FaceFilter —— 作用到像柱体、球这样的面上。

StdSelect_ShapeTypeFilter —— 针对复合体、体、壳、线等。

AIS_TypeFilter —— 作用到交互对象的类型上的。

AIS_SignatureFilter —— 作用到交互对象的标记类型上的。

AIS_AttributeFilter —— 作用到交互对象的属性如色彩、线宽等。

由于对象的形状上有一些具体的行为要实现，因此需要给每个新的过滤器类重定义一个函数，以此来通知当前环境是否执行 sub-shapes 类型。

AIS_LocalContext::ActsOn —— 默认下，这个函数的响应是错误的。

需要注意的是，在 Neutral Point 中只有类型滤波器是活动的，这样就可以识别出可视化对象的具体类型。要使滤波器起作用，必须启动一个或更多的对象选取模式。

AIS_InteractiveContext::AddFilter —— 添加过滤器

AIS_InteractiveContext::RemoveFilter —— 移除所有过滤器

以下是一个使用过滤器的例子：

```
myContext->OpenLocalContext(Standard_False);
                                                    //在 neutral point 中没有导入对象
myContext->ActivateStandardMode(TopAbs_Face);
                                                    //活动状态下将形状分解成面
Handle (AIS_Shape) myAIShape = new AIS_Shape ( ATopoShape);
myContext->Display(myAIShape,1,-1,Standard_True,Standard_True);
                                                    //阴影可视化模式, 是个不具体的模
```

式。这个模式下可将形状分解成sub-shapes。
//在这个阶段，myAIShape 被分解成面。

```
Handle(StdSelect_FaceFilter) Fil1= new StdSelect_FaceFilter(StdSelect_Revol);
Handle(StdSelect_FaceFilter) Fil2= newStdSelect_FaceFilter(StdSelect_Plane);
myContext->AddFilter(Fil1);
myContext->AddFilter(Fil2);
                                        //只有旋转面或平面才会被选取
myContext->MoveTo( xpix,ypix,Vue);
                                        //检测鼠标的位置
```

11.5.6 当前环境的使用

在当前环境可以恢复已选取的实体。但是，我们在想恢复实体之前不得不询问已经选取的实体类型或是一个交互对象。如果已经选取了一个在标准模式中被分解的TopoDS类型，那么交互函数就必须返回给交互对象已提供给的选取类型。否则函数只会允许我们执行已选取的或当前存在的对象的类型。如果需要以下操作：

1、清除已选取的对象。

2、显示已选取的对象。

3、将已选取的对象放入选取列表中。

则，执行以下函数：

AIS_InteractiveContext::EraseSelected

AIS_InteractiveContext::DisplaySelected

AIS_InteractiveContext::SetSelected

如果需要以下操作：

1、从一个当前环境中获得已选取的对象的清单。将目前的对象的清单放进Neutral Point 中。

2、在已选取的实体的清单中添加或移除一个对象。

3、高亮显示。

4、移除一个已选取对象的高亮显示。

5、清空已选取对象的列表。

则，执行以下函数：

AIS_InteractiveContext::SetSelectedCurrent

AIS_InteractiveContext::AddOrRemoveSelected

AIS_InteractiveContext::HilightSelected

AIS_InteractiveContext::UnhilightSelected

AIS_InteractiveContext::ClearSelected

如果对当前的对象进行高亮显示、移除高亮显示以及清空当前对象的列表，则执行以下函数：

AIS_InteractiveContext::HilightCurrents

AIS_InteractiveContext::UnhilightCurrents

AIS_InteractiveContext::ClearCurrents

当在一个打开的当前环境中要保持临时的交互对象时，需要执行以下函数：

AIS_InteractiveContext::KeepTemporary

AIS_InteractiveContext::SetSelectedCurrent

第一个函数的功能就是将当前环境（可视化模式）下可见的交互对象特征传递到 Neutral Point 中。这样，当当前环境关闭时对象就不会消失。第二个函数的功用是当关闭当前环境时允许把已选取的对象转变为当前对象。

如果想在继续一个操作（清空对象、移除过滤器、标准活动模式）之前以通常的方式修改当前环境的状态，那么需要执行以下函数：

AIS_InteractiveContext::ClearLocalContext

对于当前环境的使用，需要具体问题具体分析，这就取决于我们究竟想用它来执行怎样的操作，比如：

1、作用到所有想可视化的交互对象上。

2、仅仅作用于一些对象上。

3、只作用于一个单独的对象。

当需要作用到实体的类型上时，应该使用 UseDisplayedObjects 选项设置成 FALSE，然后再打开一个当前环境。以下是一些从 Neutral Point 中已给定了类型和标记的函数，可以恢复已可视化的交互对象。

AIS_InteractiveContext::DisplayedObjects

(AIS_ListOfInteractive& aListOfIO) const;

AIS_InteractiveContext::DisplayedObjects

(const AIS_KindOfInteractive WhichKind,

const Standard_Integer WhichSignature,

AIS_ListOfInteractive& aListOfIO) const;

在这个阶段，必须调用一个又一个的函数，并执行它们。其次，当需要以默认的方式打开一个当前环境的时，须记住下面的要点：

默认选取模式下，在 Neutral Point 中要被可视化的交互对象是活动的，必须把不想使用的对象处于不活动状态。当标准活动模式启动的时候，共享的交互对象的类型自动会被分解成 sub-shapes。在当前环境中，临时出现的交互对象是不会被自动考虑使用。因此，要想使用就必须手动将它们导入到当前环境中。这个阶段可能要做下面的工作：

1、以正确的方式打开一个当前环境

2、用想要的活动模式导入/可视需求的补充对象

3、如果必要就启动标准模式

4、创建属于自己的过滤器并把它们添加到当前环境中

5、检测/选取/恢复想要的实体

6、用适当的标记关闭当前环境

最后，还需要创建一个交互编辑器，它是来传递交互环境的，这根据实际需要来设置选取/显示环境。以下用一个实例来说明当前环境的管理。

如果已经可视化了一些类型的交互对象，比如：AIS_Points, AIS_Axes, AIS_Trihedrons 和 AIS_Shapes。使用合适的函数，还需要一根轴来创建一个回转对象。为获得这根轴，可以这样来说明：一根轴已经被可视化为 2 个点；形状是一条直线边，一个圆柱面（因为需要使用这根轴的面）。程序执行代码如下：

```
myIHMEditor::myIHMEditor(
const Handle(AIS_InteractiveContext)& Ctx, ....) : myCtx(Ctx), ...{}
myIHMEditor::PrepareContext()
{
  myIndex =myCtx->OpenLocalContext();
                                                    //处理过滤器
  Handle(AIS_SignatureFilter) F1 = new
  AIS_SignatureFilter(AIS_KOI_Datum,AIS_SD_Point);//在点处的过滤器
  Handle(AIS_SignatureFilter) F2 = new
  AIS_SignatureFilter(AIS_KOI_Datum,AIS_SD_Axis);//在轴线处的过滤器
  Handle(StdSelect_FaceFilter) F3 = new
  StdSelect_FaceFilter(AIS_Cylinder);                //圆柱面过滤器//...
                                                    //激活了形状的标准模式
  myCtx->ActivateStandardMode(TopAbs_FACE);
  myCtx->ActivateStandardMode(TopAbs_VERTEX);
  myCTX->Add(F1);
  myCTX->Add(F2);
  myCTX->Add(F3);
```

```
                                                    //在这个点，我们可以调用选取/检测函数
}
void myIHMEditor::MoveTo(xpix,ypix,Vue)
{
  myCTX->MoveTo(xpix,ypix,vue);                  //检测对象并高亮显示是自动进行的
}
Standard_Boolean myIHMEditor::Select()
{
                                                    //如果继续选取就会返回一个真值
  myCTX->Select();
  myCTX->InitSelected();
  if(myCTX->MoreSelected())
    {
      if(myCTX->HasSelectedShape())
        { const TopoDS_Shape& sh = myCTX->SelectedShape();
            if( vertex)
              {
                 if(myFirstV...)
                            {
                            //如果它是第一个顶点,我们就保存，然后取消面的活动，仅仅
                            保持过滤器
                              mypoint1 = ....;
                              myCtx->RemoveFilters();
                              myCTX->DeactivateStandardMode(TopAbs_FACE);
                              myCtx->Add(F1);
                                                    //在 AIS_Points 处的过滤器
                             myFirstV = Standard_False;
                             return Standard_True;
                              }
                   else
                              {
                               mypoint2 =...;
                                                    //构建返回给 Standard_False 的轴线
                                }
              else
                {
                                                  //它是一个圆柱面：我们就要重新获得轴线；
                                                  可视化它，然后进行贮存
                        return Standard_False;
                 }
                                                  }
                                              //虽然不是一个形状，但我们不用怀疑，它是一个点
       else
         { Handle(AIS_InteractiveObject)SelObj = myCTX->SelectedInteractive();
```

```
                    if(SelObj->Type()==AIS_KOI_Datum)
                      {
                        if(SelObj->Signature()==1)
                          {
                           if (firstPoint)
                             {
                                mypoint1 =...
                                return Standard_True;
                             }
                           else
                             {
                              mypoint2 = ...;
                              //构建轴线，可视化后进行贮存
                              return Standard_False;
                             }
                          }
                        else
                          {
                                  //我们已经选择了一根轴线，务必要贮存它
                           return Standard_False;
                           }
                        }
            }
          }
      }
}
void myIHMEditor::Terminate()
   {
   myCtx->CloseLocalContext(myIndex); ...
   }
```

11.6AIS 标准交互对象类

1、基本类：包括 AIS_Point、AIS_Axis、AIS_Line、AIS_Circle、AIS_Plane 、AIS_Trihedron ，这些类包含于 AIS_Shape 类。

2、AIS_ConnectedInteractive 类：功能是使一个交互对象与另一个引用的交互对象联系起来，并在浏览器中确定它们的位置。这样做就不用再对引用的对象进行显示和选取计算，但需要知道所引用对象的出处。

3、AIS_ConnectedShape 类：用来对与交互对象联系的对象指定一个形状，这个类和 AIS_Shape 一样可以被分解来使用。除此之外，它还允许对对象的隐藏部分进行显示，这系统会根据形状的参数自动计算它。

4、AIS_MultipleConnectedInteractive 类：功用是将对象与交互对象的列表联系起来（也可以是对象之间联系）。实现这一过程不需要占用内存空间就可以进行显示计算）。

5、AIS_MultipleConnectedShape 类：用来对与交互对象列表联系的交互对象给定一个形状（可以是 AIS_Shape,AIS_ConnectedShape, AIS_MultipleConnectedShape）,隐藏部分的显示计算由系统自动完成。

6、关系类

(1)AIS_ConcentricRelation、AIS_FixRelation

(2)AIS_IdenticRelation、AIS_ParallelRelation

(3)AIS_PerpendicularRelation、AIS_Relation

(4)AIS_ SymmetricRelation、AIS_TangentRelation

注：这只是其中的一部分。

7、尺寸类

(1)AIS_AngleDimension、AIS_Chamf2dDimension

(2)AIS_Chamf3dDimension、AIS_DiameterDimension

(3)AIS_DimensionOwner、AIS_LengthDimension

(4)AIS_OffsetDimension、AIS_RadiusDimensio

11.7 本章小结

本章分析了 AIS 包的使用方法。从文中内容可以看出，AIS 包提供的算法是对交互对象、交互对象的图形属性、交互环境、选择过滤器进行全面管理，执行管理过程还需要遵循相关的规定和协定。在 AIS 中，已经定义好了标准交互对象类和一些虚函数，可以根据需要来调用。由于本章的算法之多，在此就不一一列举，只想对一些要点作如下总结：

交互环境有两中操作模式：Neutral Point 模式、当前可视化和选取的环境模式。前者为默认模式，允许对交互对象进行选取和可视化；后者即为打开的当前环境，是为临时选取和显示作准备的；两种模式下执行对象的显示算法也各有不同，有时需要各调用各自的算法。使用选择过滤器时要注意的是，不同的操作模式下需要使用不同的过滤器。

视图有两种状态：衰退模式（通常的模式）和不衰退模式（隐藏线模式）。两种模式下可显像的交互对象显示方式不同。交互对象也有不确定的选取模式，每个选取模式由一个标志来确定。

第十二章 2D 显示

2D 术语

1、属性：将材质的一些特性赋予到一个实体上。图元有属性，如线图元可以有这样的类型 —— 线段、直线、点划线；线还有一个线宽属性 —— 粗的、细的；文本图元有一个字型属性如 Helvetica。所有的图元都有一个色彩属性。

2、属性映像：程序中要使用的图元属性均保存在映像器里。一个映像就是一个由标记和属性组成的数组。

3、驱动器：驱动器是为绘制视图做准备的。驱动器或与一个窗口或与一个绘图仪文件有关。使用的最终目的是为了视图的需要。

4、图形设备：默认下，图形设备会与当前的工作站联系，从而来监控屏幕和色彩映像的预定状态。

5、图形对象：用来管理一组图元。一个图形对象可以对图元进行编辑、添加或移除等操作。默认下，图形对象是空的、可以拖动的、可绘制的、可拾取的，它既不会被显示也不会被突出显示。

6、图元：图元是可拽的图形元件，它已经在模型空间中给予了定义。图元可以是线、文本，也可以是图像。对图元进行显示时图像和文本必须保持同样的尺寸。线除标记以外其他的属性都可以被修改。图元一定要保存在图形对象中。

7、模型空间：整个 2D 空间就是模型空间的一部分，它由一个初始中心点和一个尺寸来定义。

8、更新：这是个搜索视图中显示列表的过程，目的是查找每一个可视的图形对象，更新是在视图映射中实现的。然而，对于每一个图形对象的更新过程是需要请求每个图形对象自身的绘制方法来实现。同时每个图形对象又需调用其图元的绘制方法。

9、视图：视图是由一组排列在显示列表中的图形对象构成。但它能够为拾取和绘制图形对象时提供方法。

10、视图映射：是从模型空间的中心和尺寸范围内定义的一个平方区域，其目的是用来选择要被显示的图元。

11、窗口：顾名思义是个程序应用界面，它由窗口管理器提供，例如：X Window。

12、工作空间：窗口驱动的工作场所就是窗口的工作空间。对于一个绘图仪驱动，它的工作空间就是图纸的页面尺寸。

12.1 创建 2D 显示

要创建一个 2D 图形对象并将它们显示到屏幕上，其步骤如下：

1、创建标记映像

2、创建属性映像

3、定义连接图形设备

4、创建一个窗口

5、创建窗口驱动

6、安装映像

7、创建一个视图

8、创建视图映射

9、创建一个或更多与视图有关的图形对象

10、创建与图形对象有关的图元

11、获取驱动的工作空间

12、更新在驱动中的视图

12.1.1 创建标记映像

标记映像定义了一组可以在程序中使用的标记。标记可以预定义，例如 Aspect_Tom_X，或用户自定义。标记是由索引来管理，标记映像的定义如图 5-1 所示。

图 5　1 标记

示例代码如下：

```
Handle(Aspect_MarkMap) mkrmap = new Aspect_MarkMap;
Aspect_MarkMapEntry mkrmapentry1 (1, Aspect_TOM_X)
Aspect_MarkMapEntry mkrmapentry2 (2, Aspect_TOM_PLUS)
Aspect_MarkMapEntry mkrmapentry3 (3, Aspect_O_PLUS)
mkrmap->AddEntry (mkrmapentry1);
```

```
mkrmap->AddEntry (mkrmapentry2);
mkrmap->AddEntry (mkrmapentry3);
```

12.1.2 创建属性映像

映像是为了实现色彩、线型、线宽以及文本字型而创建的，同时映像又是用来引用已给定一个整值的属性才定义的，图 5-2 所示是一些属性。

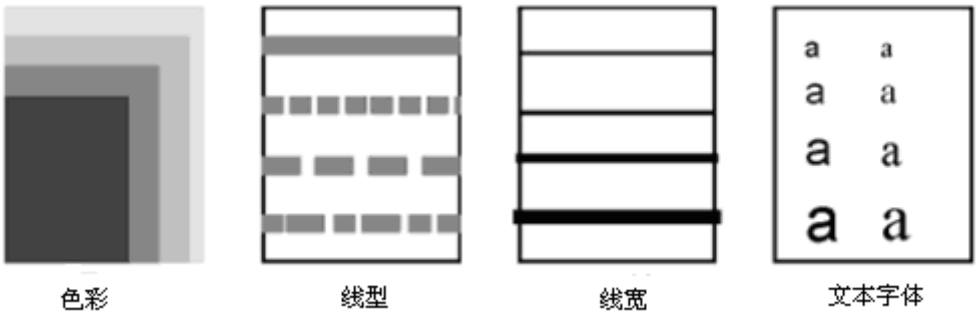


图 5　2 属性

1、色彩映像：为了使程序具有可移植性和永久性的特点，硬件系统一定要有一个默认的可应用到程序上的色彩。因此，它必须定义一组自身可以使用的色彩从而与外界的环境因素隔绝。色彩映像可以这样来定义：

```
Handle(Aspect_GenericColorMap) colmap =new Aspect_GenericColorMap;
Aspect_ColorMapEntry colmapentry;
Quantity_Color YELLOW (Quantity_NOC_YELLOW);
colmapentry.SetValue (1, YELLOW);
colmap->AddEntry (colmapentry);
Quantity_Color RED (Quantity_NOC_RED);
colmapentry.SetValue (2, RED);
colmap->AddEntry (colmapentry);
Quantity_Color GREEN (Quantity_NOC_GREEN);
colmapentry.SetValue (3, GREEN);
colmap->AddEntry (colmapentry);
```

当然可以定义包含更多我们想得到的色彩。如此，需要制定一些与硬件有关的约束。

2、类型映像：线可以是实线、点划线、虚线或者用户自定义的线。对于用户自定义的线的类型，实段和空段部分的线型结构需要列举出来。类型映像可以这样来定义：

```
Handle(Aspect_TypeMap) typmap = new Aspect_TypeMap;
{TColQuantity_Array1OfLength myLineStyle(1,2);
```

```
myLineStyle.SetValue(1, 2);
                                                    //实体部分是 2 毫米
myLineStyle.SetValue(2, 3);
                                                    //空白部分是 3 毫米
Aspect_LineStyle linestyle1 (Aspect_TOL_SOLID);
Aspect_LineStyle linestyle2 (Aspect_TOL_DASH);
Aspect_LineStyle linestyle3 (myLineStyle);
Aspect_LineStyle linestyle4 (Aspect_TOL_DOTDASH);
Aspect_TypeMapEntry typmapentry1 (1, linestyle1);
Aspect_TypeMapEntry typmapentry2 (2, linestyle2);
Aspect_TypeMapEntry typmapentry3 (3, linestyle3);
Aspect_TypeMapEntry typmapentry4 (4, linestyle4);
typmap->AddEntry (typmapentry1);
typmap->AddEntry (typmapentry2);
typmap->AddEntry (typmapentry3);
typmap->AddEntry (typmapentry4);
```

注：在 Aspect 包中，线型的枚举和所有其它的枚举都可以使用。

3、线宽映像：线宽映像定义了使用在程序中的线的宽厚属性。线宽和所有其它的尺寸都归类到 mm 类中或作为一个枚举型的成员来使用。线宽映像可以这样来定义：

```
Handle(Aspect_WidthMap) widmap = new Aspect_WidthMap;
Aspect_WidthMapEntry widmapentry1 (1,Aspect_WOL_THIN);
Aspect_WidthMapEntry widmapentry2 (2,Aspect_WOL_MEDIUM);
Aspect_WidthMapEntry widmapentry3 (3, 3);
Aspect_WidthMapEntry widmapentry4 (4, 40);
widmap->AddEntry (widmapentry1);
widmap->AddEntry (widmapentry2);
widmap->AddEntry (widmapentry3);
widmap->AddEntry (widmapentry4);
```

4、字体映像：字体映像定义了一组可以使用在程序中的文本字型。默认下，字型已经枚举在 Aspect 包中，它可以使用在由 X driver 获取到的任何其它字型中，X driver 中已经规定了字型的尺寸和期望的倾斜角度。字映像可以这样来定义：

```
Handle(Aspect_FontMap) fntmap = new Aspect_FontMap;
Aspect_FontStyle fontstyle1 (”Courier-Bold”, 3, 0.0);
Aspect_FontStyle fontstyle2 (”Helvetica-Bold”, 3, 0.0);
Aspect_FontStyle fontstyle3 (Aspect_TOF_DEFAULT);
Aspect_FontMapEntry fntmapentry1 (1, fontstyle1);
Aspect_FontMapEntry fntmapentry2 (2, fontstyle2);
```

```
Aspect_FontMapEntry fntmapentry3 (3, fontstyle3);
fntmap->AddEntry (fntmapentry1);
fntmap->AddEntry (fntmapentry2);
fntmap->AddEntry (fntmapentry3);
```

12.1.3 创建 2D 驱动

在 Windows 平台下创建 2D 浏览器的实现示例如下：

```
Handle(WNT_GraphicDevice) TheGraphicDevice = ...;
TCollection_ExtendedString aName("2DV");
my2DViewer = new V2d_Viewer(TheGraphicDevice,aName.ToExtString());
```

12.1.4 安装映像

当 2D 浏览器已经被创建的时可以提早对映像进行安装。 例：

```
my2DViewer->SetColorMap(colormap);
my2DViewer->SetTypeMap(typmap);
my2DViewer->SetWidthMap(widthmap);
my2DViewer->SetFontMap(fntmap);
```

12.1.5 创建视图

假定一个有效的 Windows 窗口可以被通用接口适配器（via）来访问，那么可以使用 GetSafeHwnd()方法来创建。在 Windows 下创建视图示例如下：

```
Handle(WNT_Window) aWNTWindow;
aWNTWindow = new WNT_Window(TheGraphicDevice, GetSafeHwnd());
aWNTWindow->SetBackground(Quantity_NOC_MATRAGRAY);
Handle(WNT_WDriver) aDriver = new WNT_WDriver(aWNT_Window);
myV2dView = new V2d_View(aDriver, my2dViewer, 0,0,50);
                                                    // 0,0：视图中心；50：视图尺寸
```

12.1.6 创建可显示的对象

计算可显示对象的过程如下：

```
Void myPresentableObject::Compute (
const Handle(Prs_Mgr_PresentationManager2D)&aPresentationManager,
const Handle(Graphic2d_GraphicObject)& aGrObj,
const Standard_Integer aMode)
{
...
}
```

12.1.7 创建图元

使用 Graphic2d 包的资源来创建图元。以下是一个数组的示例，目的是用不同的线宽和色彩来填充三个有不同半径的圆。这三个圆的圆心都在已给定的起始坐标（4.0，1.0）处，填充以后将它们传递给已定义好的图形对象。实现代码示例如下：

```
Handle(Graphic2d_Circle) tcircle[4];
Quantity_Length radius; for (i=1; i<=4; i++)
    {
     radius = Quantity_Length (i);
     tcircle[i-1] = new Graphic2d_Circle (aGrObj, 4.0, 1.0, radius);
     tcircle[i-1]->SetColorIndex (i);
     tcircle[i-1]->SetWidthIndex (1);
     }
```

添加一个填充好的矩形到图形对象中，它会从视图映像的外部添加进去。实现代码示例如下：

```
TColStd_Array1OfReal aListX (1, 5);
TColStd_Array1OfReal aListY (1, 5);
aListX (1) = -7.0; aListY (1) = -1.0;
aListX (2) = -7.0; aListY (2) = 1.0;
aListX (3) = -5.0; aListY (3) = 1.0;
aListX (4) = -5.0; aListY (4) = -1.0;
aListX (5) = -7.0; aListY (5) = -1.0;
Handle(Graphic2d_Polyline) rectangle =new Graphic2d_Polyline (go, 0., 0., aListX, aListY);
rectangle->SetColorIndex (6);
rectangle->SetWidthIndex (1);
rectangle->SetTypeOfPolygonFilling(Graphic2d_TOPF_FILLED);
rectangle->SetDrawEdge(Standard_True);
```

注：一个已给定的图元可能仅仅被指派给了一个单独的图形对象。如图 5-3 所示是视图映射的情况。

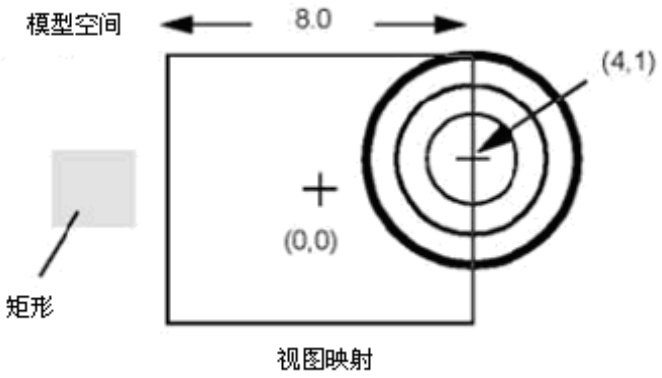


图 5　3 在模型空间的图形对象和视图映射

4.1　图像处理

4.1.1　一般情况

图像也是图元，图形资源在当前可能会接受所有在 AlienImage 包中描述的图像类型。下面的例子仅仅是.xwd 格式下接受的情况。在 GraphicObject 中定义图元图像。示例代码如下：

```
Handle(Image_Image) anImage;if (XwdImage || RgbImage)
{
anImage = AlienUser->ToImage ();
Handle(Graphic2d_Image) gImage = new Graphic2d_Image
(aGrObj, anImage, 0., 0., 0., 0., Aspect_CP_CENTER);
}
```

注：以上的图像构造器只是作为图形对象的一个依据；图形对象包含图像和图像本身的一些属性，如中心的 XY 坐标，在设备空间的 XY 偏移量以及一个给了显示方向的关键点。现在就可以在驱动中对视图进行更新，换句话说，就是绘制图像。代码如下：

```
Standard_Boolean clear = Standard_True
view->Update (driver, viewmapping, W/2., H/2., scale, clear);
```

4.1.2　特殊情况：.xwd 格式

当被管理的图像用.xwd 格式保存时，Graphic2d_ImageFile 这个特殊的类可能被用来增加其执行能力。

代码如下：

```
OSD_Path aPath (“C:\test.xwd”);
OSD_File aFile (aPath);
```

```
Handle(Graphic2d_ImageFile)gImageFile=new
Graphic2d_ImageFile (aGrObj,aFile,0.,0.,0.,0.,
Aspect_CP_Center, 1);
gImageFile->SetZoomable(Standard_True);
```

现在图形包含了一个图像，图像作为一个图元来处理。

4.2 文本处理

Graphic2d_Text 的构造器会从模型空间获得一个参考点和一个角度来作为一个依据，当然它也会指派给图形对象。需要注意的是除非要使用角度文本，否则角度是要被忽略的。例：

```
TCollection_ExtendedString str1 (”yellow Courier-bold”);
TCollection_ExtendedString str2 (”red Helevetica-bold”);
TCollection_ExtendedString str3 (”green Aspect_TOF_DEFAULT”);
Handle(Graphic2d_Text) t1 = new Graphic2d_Text(aGrObj, str1, 0.3, 0.3, 0.0);
Handle(Graphic2d_Text) t2 = new Graphic2d_Text(aGrObj, str2, 0.0, 0.0, 0.0);
Handle(Graphic2d_Text) t3 = new Graphic2d_Text(aGrObj, str3, -0.3, -0.3, 0.0);
t1->SetFontIndex (1); t1->SetColorIndex (1);
t2->SetFontIndex (2); t2->SetColorIndex (2);
t3->SetFontIndex (3); t3->SetColorIndex (3);
```

4.3 标记处理

标记是一种图元，当视图被缩放时需要保持它的原始尺寸。标记是可以使用的，例如作为对尺寸的参考时。

4.3.1 向量标记

每个标记取得一个 XY 坐标点作为它的参考点。构造器也会从 XY 坐标中取得一对坐标作为这些参考点的偏移量。对于 CircleMarker 和 EllipsMarker 这两个偏移点就是它们的中心。而对于 PolylineMarker 这个偏移点就是它的原点。下面的例子是用 Graphic2d_Polyline 创建一个矩形：

```
TColStd_Array1OfReal rListX (1, 5);
TColStd_Array1OfReal rListY (1, 5);
rListX (1) = -0.3; rListY (1) = -0.3;
rListX (2) = -0.3; rListY (2) = 0.3;
rListX (3) = 0.3; rListY (3) = 0.3;
rListX (4) = 0.3; rListY (4) = -0.3;
rListX (5) = -0.3; rListY (5) = -0.3;
Handle(Graphic2d_Polyline) rp =new Graphic2d_Polyline (aGrObj, rListX, rListY);
```

可以看到创建了两个 Graphic2d_CircleMarkers。第一个从它的中心没有给偏移量。而第二个则从参考点强行给定了一个偏移量，这可以从图 5-4 中看出。例：

```
Handle(Graphic2d_CircleMarker) rc1 = newGraphic2d_CircleMarker
(aGrObj, 0.04, 0.03, 0.0, 0.0, 0.01);
Handle(Graphic2d_CircleMarker) rc2 = newGraphic2d_CircleMarker
(aGrObj, 0.03, -0.03, 0.01, 0.0, 0.01);
window->Clear ();
```

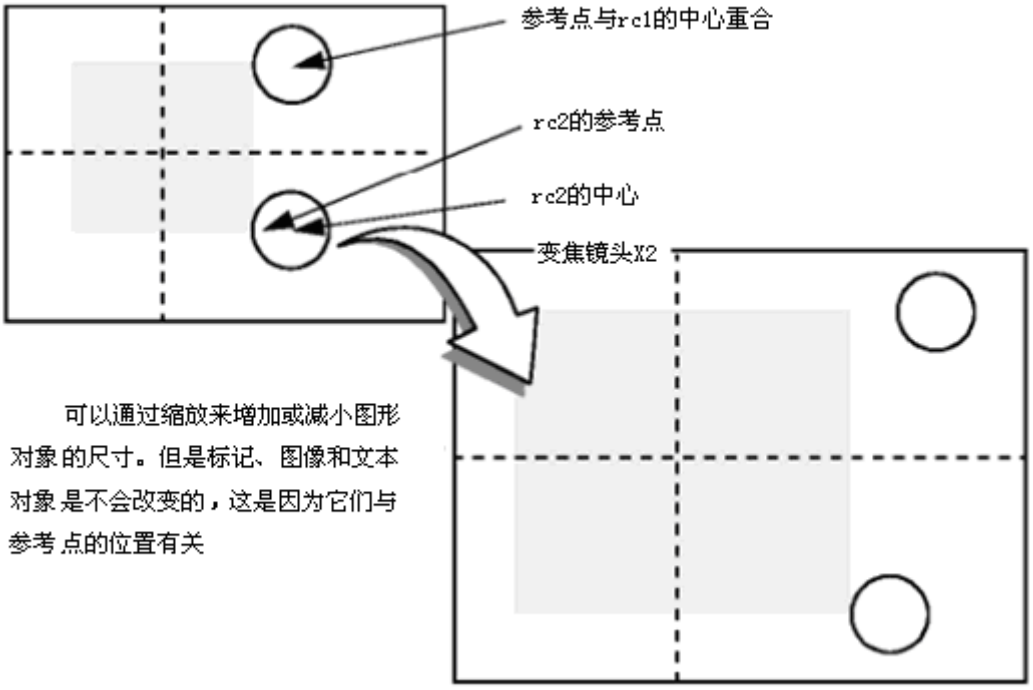


图 5　4 标记联系点和图的缩放

4.3.2　索引标记

一旦标记映像被创建，索引标记就会被添加到一个图形对象上了。例：

```
Handle (Graphic2d_Marker) xmkr = new Graphic2d_Marker
(aGrObj, 1, 0.04, 0.03, 0.0, 0.0, 0.0);
Handle (Graphic2d_Marker) plusmkr = new Graphic2d_Marker
(aGrObj, 2, 0.04, 0.0, 0.0, 0.0, 0.0);
Handle (Graphic2d_Marker) oplusmkr = new Graphic2d_Marker
(aGrObj, 3, 0.04, -0.03, 0.0, 0.0, 0.0);
```

4.4　使用缓冲器进行绘制

缓冲器的用途就是在不删除背景环境的情况下快速绘制场景的一部分区域。缓冲器包含一组要被移动的、旋转的或在视图的前向平面内绘制场景所需的图形

对象或图元。例如：在视图中绘制一个非常复杂的场景。创建一个有图元色彩 index 10 和字型 index 4 的图元缓冲器：

```
buffer = new Graphic2d_Buffer (view, 0., 0., 10, 4);
                                                    //添加图形对象或图元
buffer->Add (go);
buffer->Add (tcircle[1]);
buffer->Add (t1);
                                                    //在视图中加速缓冲器
buffer->Post ();
                                          //在上面的视图中移动、旋转或测量缓冲器
buffer->Move (x,y); buffer->Rotate (alpha);
buffer->Scale (zoom_factor);
                                                    //在视图中不加速缓冲器
buffer->Unpost ();
```

4.5 本章小结

本章对 2D 图形对象的创建和如何显示的过程进行了全面分析，从中可以看出 3D 显示和 2D 显示的区别是很大的。3D 显示直接应用 OCC 提供的方法即可实现，而 2D 显示更多是建立在计算机图形学算法基础上的，一些概念和术语也都引用自图形学的知识，因此，实现过程较 3D 显示更为复杂。

在创建 2D 图形对象之前，要理解几个重要的概念：映像器、驱动器、映射、工作空间、模型空间。映像器用来管理创建的图形属性；驱动器是为绘制视图做准备的，最终目的还是为了视图的需要；创建的过程还需要对标记和图元进行处理。

总之，2D 显示与 3D 显示的过程是截然不同的。所以，对本章内容的学习我们可以参照图形学的相关知识来理解。

第十二章 数据交换模块分析（缺理论）

应用 OpenCASCADE 软件诸如 CAD 系统时，数据交换是一个关键因素。在多层软件环境中，它应用 OpenCASCADE 开源的技术，处理外部数据并提供了一个良好的融合程度。这种情况所涉及的数据交换模块是有组织的模块化方式。

OpenCASCADE 数据交换是提供工具，诸如数据转换模量，允许软件基于

OpenCASCADE 将数据转换成其它的 CAD 软件，从而保证了良好的互通性。

标准的数据包有 STEP 、IGES 、STL 和 VRML 等。