第 2 章　创建和编辑基本图形对象

本章将以创建和编辑基本图形对象作为突破口，逐步深入，引导你进入 ObjectCRX 编程的乐园。

2.1 创建直线

2.1.1 说明

本实例运行的结果是在 电子图板 2011 中，创建一条直线，该直线的起点是（0，0，0），终点是（100，100，0）。除此之外，不准备再做更多的事情。麻雀虽小，五脏俱全。通过这个程序，你将要开始了解 电子图板数据库的基本结构。

2.1.2 思路

首先来看看，在电子图板中，使用 LINE 命令创建一条直线，需要哪些东西：

```
命令：line
第一点(切点或垂足)：0,0
下一点(切点或垂足)：100,100
```

从上面的命令提示可以看出，创建一条直线，需要用户指定起点和终点。

在继续之前，必须给大家介绍一点数据库最基础的几个名词：

- 表：表是数据库的组成单位，一个数据库至少包含一个表。
- 记录：记录是表的组成单位，一个表可能包含多条记录，也可能不包含任何记录。

图 2.1 用来描述电子图板数据库的基本结构再好不过了。从图中来看，实体包含在块表记录中，因此要创建一个图形对象，需要遵循下面的基本步骤：

（1）确定要创建对象的图形数据库；

（2）获得图形数据库的块表；

（3）获得一个存储实体的块表记录，所有模型空间的实体都存储在模型空间的特定记录中。

（4）创建实体类的一个对象，将该对象附加到特定的块表记录中。

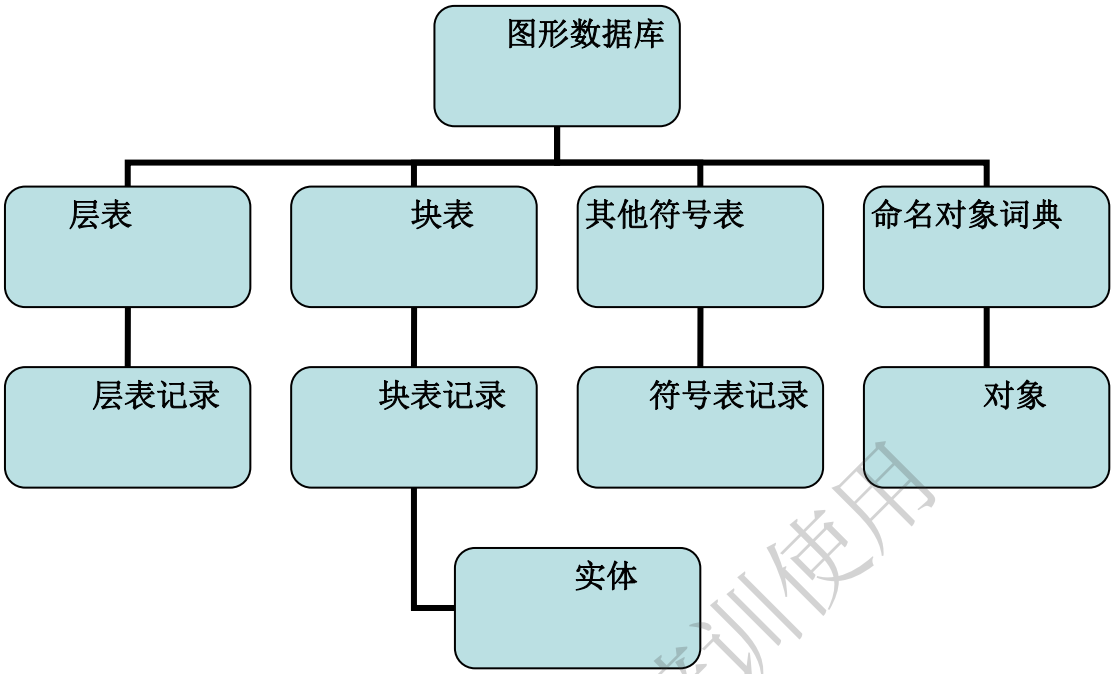


图2.1 图形数据库的结构

2.1.3 步骤

（1）在 Visual Studio 2010 中，使用向导创建一个新的 ObjectCRX 项目（名称为 CreateLine），其方法与 1.3 节完全一致。

（2）在 CreateLine 函数中，添加创建直线对象（在 ObjectCRX 中，CRxDbLine 类代表直线）的代码：

```
// 在内存上创建一个新的CRxDbLine对象
    CRxGePoint3d ptStart(0, 0, 0);
    CRxGePoint3d ptEnd(100, 100, 0);
    CRxDbLine *pLine = new CRxDbLine(ptStart, ptEnd);
```

> 注意：基于电子图板内部的实现机制，必须在堆上创建对象，而不能用下面的语句创建直线的对象：CRxDbLine line(ptStart, ptEnd); 此时，直线对象仅被在内存上创建，并没有添加到图形数据库中，因此不可能会显示在图形窗口中。

（3）在 CreateLine 函数中，添加获得指向块表的指针的相关代码：

```
// 获得指向块表的指针
    CRxDbBlockTable *pBlockTable;
    crxdbHostApplicationServices()->workingDatabase()->getSymbolTable(pBlockTable,
    CRxDb::kForRead);
```

crxdbHostApplicationServices()->workingDatabase()能够获得一个指向当前活动的图形数据库的指针，这在后面还要经常遇到。getBlockTable 是 CRxDbDatabase 类的一个成员函数，用于获得指向图形数据库的块表的指针，其定义为：

```
inline CDraft::ErrorStatus getBlockTable(
CRxDbBlockTable*& pTable,
CRxDb::OpenMode mode);
```

该函数的返回值 CDraft::ErrorStatus 是 ObjectCRX 中定义的一个枚举类型，主要用于判断函数的返回状态，如果函数成功执行会返回 CDraft::eOk。第一个参数 pTable 返回指向块表的指针； 第二个参数同样是一个枚举类型的变量， 其类型 CRxDb::OpenMode 包含了 CRxDb::kForRead、CRxDb::kForWrite 和 CRxDb::kForNotify 三个可取的值，创建直线的时候不需要更改块表，因此这里打开的模式为 CRxDb::kForRead。

如果对 C++的概念理解不够深入，很可能会不了解 CRxDbBlockTable*& pTable 的含义。从左向右来读，就可知道该形式参数的类型是 CRxDbBlockTable*（指向块表的指针），由于引用运算符（&）的存在，那么形参 pTable 是一个指针的引用。

下面的函数相信在学习引用时大部分人都遇到过：

```
void swap(int &m, int &n);
```

没错，之所以在参数中使用引用运算符，是为了通过函数的参数实现返回值。在 getBlockTable 函数中是一样的情况，不过形参的类型变成了一个指针而已。

（4）在 CreateLine 函数中，添加获得指向特定块表记录的指针的相关代码：

```
// 获得指向特定的块表记录（模型空间）的指针
    CRxDbBlockTableRecord *pBlockTableRecord;
    pBlockTable->getAt(CRXDB_MODEL_SPACE,pBlockTableRecord,CRxDb::
    kForWrite);
```

getAt 函数是 AcDbBlockTable 类的一个成员函数，用于获得块表中特定的记录，其定义为：

```
CDraft::ErrorStatus getAt(
const CxCHAR* entryName,
AcDbBlockTableRecord*& pRec,
AcDb::OpenMode openMode,
bool openErasedRec = false) const;
```

第一个参数用于指定块表记录的名称，CRXDB_MODEL_SPACE 是 ObjectCRX 中定义的一个常量，其内容是“*Model_Space”；第二个参数用于返回指向块表记录的指针；第三个参数指定了块表记录打开的模式，下一步要向块表记录中添加实体，所以就用写的模式（CRxDb::kForWrite）打开；第四个参数指定是否查找已经被删除的记录，这里暂时不深入介绍，后面在合适的地方会谈到它，一般使用默认的参数值。

（5）在 CreateLine 函数中，添加向块表记录中附加实体的代码：

```
// 将CRxDbLine类的对象添加到块表记录中
    CRxDbObjectId lineId;
    pBlockTableRecord->appendAcDbEntity(lineId, pLine);
```

appendAcDbEntity 是AcDbBlockTableRecord 类的成员函数，用于将pEntity 指向的实体

添加到块表记录和图形数据库中，其定义为：

```
CDraft::ErrorStatus appendAcDbEntity(
CRxDbObjectId& pOutputId,
CRxDbEntity* pEntity);
```

第一个参数返回图形数据库为添加的实体分配的 ID 号；第二个参数指定了所要添加的实体。

> （CRxDbObjectId）是 ID（身份、标识）。管理者通过这个编号来管理。图形数据库也一样作为管理者，只能通过编号（CRxDbObjectId）来管理每一个实体。

（6）在 CreateLine 函数中，添加关闭图形数据库各种对象的代码：

```
// 关闭图形数据库的各种对象
    pBlockTable->close();
    pBlockTableRecord->close();
    pLine->close();
```

在操作图形数据库的各种对象时，必须遵守电子图板的打开和关闭对象的协议。该协议确保当对象被访问时在物理内存中，而未被访问时可以被分页存储在磁盘中。创建和打开数据库的对象之后，必须在不用的时候关闭它。

初学 ObjectCRX 的人肯定会有两点疑问：

- 各种数据库对象的关闭：在打开或创建数据库对象之后，必须尽可能早的关闭它。在初学者所犯的错误中，未及时关闭对象的错误至少占一半！
- 不要使用 **delete pLine** 的语句。对 C++比较熟悉的读者，习惯于配对使用 new 和 delete 运算符，这在 C++编程中是一个良好的编程习惯。但是在 ObjectCRX 的编程中，编程者使用 appendAcDbEntity 函数将对象添加到图形数据库之后，就需要由图形数据库来操作该对象。

（7）最后，来看一下完整的代码：

```
void CreateLine()
{
// 在内存上创建一个新的CRxDbLine对象
    CRxGePoint3d ptStart(0, 0, 0);
    CRxGePoint3d ptEnd(100, 100, 0);
    CRxDbLine *pLine = new CRxDbLine(ptStart, ptEnd);

    // 获得指向块表的指针
    CRxDbBlockTable *pBlockTable;
    crxdbHostApplicationServices()->workingDatabase()->getSymbolTable(pBlockTable,
CRxDb::kForRead);
```

```
    // 获得指向特定的块表记录（模型空间）的指针
    CRxDbBlockTableRecord *pBlockTableRecord;
    pBlockTable->getAt(CRXDB_MODEL_SPACE, pBlockTableRecord,CRxDb::kForWrite);

    // 将CRxDbLine类的对象添加到块表记录中
    CRxDbObjectId lineId;
    pBlockTableRecord->appendAcDbEntity(lineId, pLine);


    // 关闭图形数据库的各种对象
    pBlockTable->close();
    pBlockTableRecord->close();
    pLine->close();
}
```

（8）按下键盘上的快捷键 F7，对上面的代码进行编译，肯定会得到两个错误，如图 2.2 所示。这就引出另一个在 ObjectCRX 编程中常见的问题：包含适当的头文件。

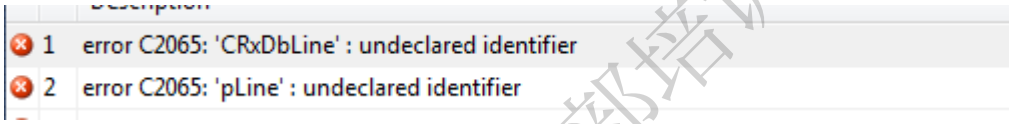


图2.2 编译出错的位置和错误提示

编译错误的提示为：CRxDbLine 是一个未定义的标识符。已经知道了 CRxDbLine 是 ObjectCRX 中的一个类，那么就要找到其定义的位置，将其所在的头文件包含到当前文件中来。如何确定 CRxDbLine 类需要包含哪个头文件呢？

打开 ObjectCRX 帮助文档中的 ObjectCRX 类参考文档，转到【索引】选项卡，在文本框中输入 CRxDbLine（如果你已经指定了 ObjectCRX 帮助文件的位置，并且为其定义了快捷键，就可以像这样，直接按下 Alt+F1，自动完成上面的手工操作），在列表中双击 CRxDbLine Class 选项，得到如图 2.3 所示的结果。

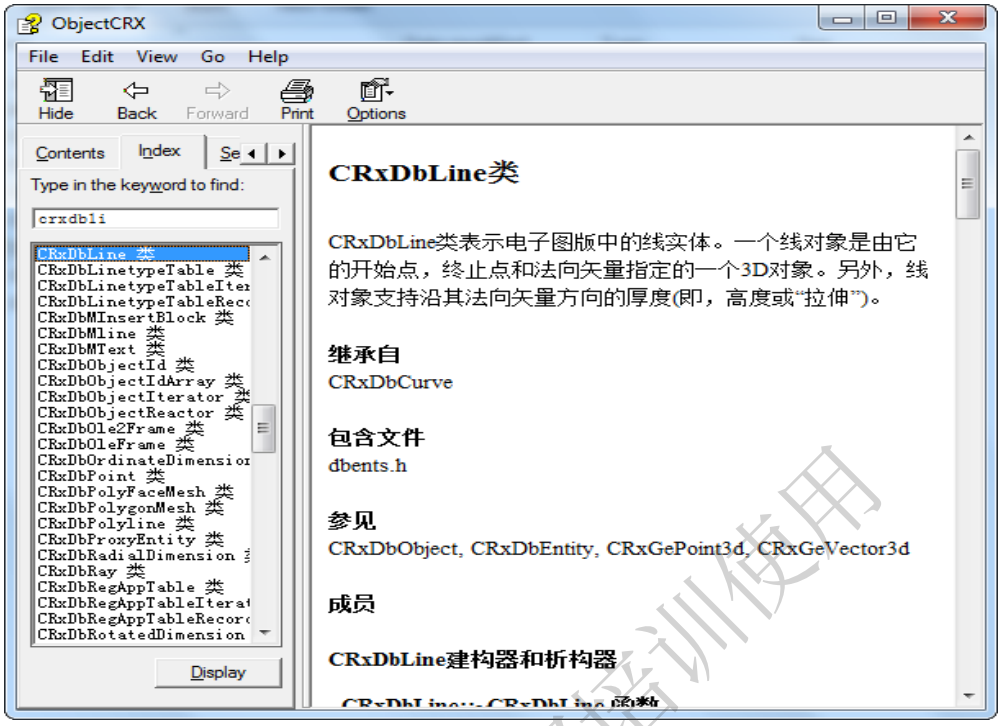


图2.3 确定所要包含的头文件

从图中的框内你可以发现一个秘密，没错，要确定一个 ObjectCRX 关键字、类、全局函数需要包含的头文件，都可以如法炮制。

（9）添加包含头文件的语句。已经知道 CRxDbLine 类需要包含什么头文件，就可以在 CreateLineCommands.cpp 文件的开头添加下面的语句：

```
#include "dbents.h"
```

然后按照 1.3 节中介绍的那样附加 ObjectCRX 开发包中的头文件和库文件目录，再次按下 F7 键编译程序， 不会再出现错误和警告， 并生成了目标文件 CrxCreateLine.crx。

> 提示：关于包含头文件语句放置的位置，还值得讨论一下。如果使用向导生成 ObjectCRX 项目，想到会自动创建一个名为 StdAfx.h 的文件，由于该文件自动被所有的源文件包含，因此其中可以放置一些经常要被包含的头文件（放在注释语句“//----- CAXA editor API”之后即可）；如果某个头文件可能仅会在一个文件内使用，那么可以像本节实例那样，直接放在源文件的开头。希望从上面的过程中，你可以了解到创建图形实体的一般过程。Ok，如果有足够的信心，可以试写一下创建圆的代码。
>
> 重要提示：在 **ObjectCRX** 程序中会大量地涉及到包含头文件的语句，如果每次都在书中出现这些语句无疑会浪费大量的篇幅，因此后面的很多程序都不会贴出需要包含的头文件，读者可以自行根据上面的方法确定需要包含的头文件，或者从配套光盘中获得相应的包含语句。

2.1.4 效果

编译应用程序之后，在电子图板 2011 中加载并运行程序中注册的 CreateLine 命令，能够得到如图 2.4 所示的结果。

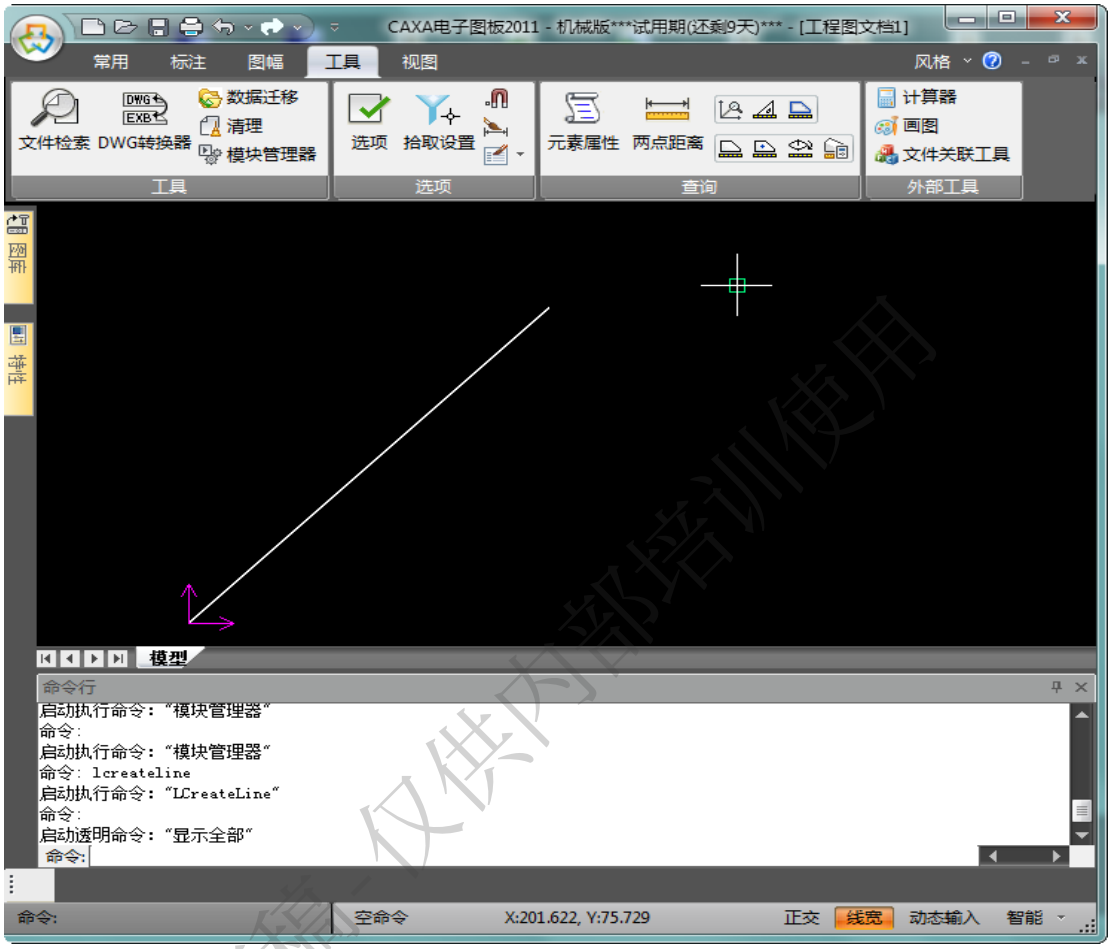


图2.4 创建直线的结果

2.1.5 小结

步骤（3）使用 getAt 函数获得了指向模型空间块表记录的指针，因此创建的直线是在模型空间，如果你想在图纸空间创建一条直线，那么你可以用 CRXDB_PAPER_SPACE 作为 getAt 函数的第一个参数，其他的代码完全不用改变。

本节用了大量的篇幅来介绍在 ObjectCRX 中创建一条直线的过程，目的是让你了解图形数据库中实体存储的结构，一个实例可能还太少了，于是，后面的内容跟着来了。

2.2 修改图形对象的属性

2.2.1 说明

上一节的学习，你已经能创建一条直线了，本节介绍的例子则会改变直线的颜色。所要实现的效果非常简单：创建一条直线之后，将它的颜色变为红色。

2.2.2 思路

如果是在创建时修改直线的颜色，就可以直接在上节的 CreateLine 函数中加入下面的代码（放在关闭图形数据库各种对象之前）：

```
pLine->setColorIndex(1);
```

运行程序中注册的 CreateLine 命令，创建的直线颜色变为红色。

在实际编程中，并不是每一次都可以在创建对象时将其特性设置到合适的状态，相反，更多的时候可能在创建对象之后才修改其特性，本节正要解决这个问题。

1．打开和关闭图形数据库的对象

访问图形数据库中对象的特性，必须在该对象被打开（对象创建时也会被打开）的状态下，用对象的指针进行访问，并且在访问结束后要及时关闭该对象，不然就会引起电子图板的错误终止。

创建一个对象，必须在创建之后关闭该对象，那么如何在某个时候再访问该对象？这就要用到 2.1 节介绍的 CRxDbObjectId，也就是对象的 ID 号。在创建对象时，可以将图形数据库分配给该对象的 ID 保存起来，在需要访问该对象时，根据这个 ID 从数据库中获得指向该对象的指针，就可以修改或者查询该对象的特性。

CRxDbBlockTableRecord 类的 appendAcDbEntity 函数能够将一个实体添加到图形数据库中，并且返回分配给该实体的 ID，这个函数上一节已经介绍过；全局函数 acdbOpenAcDbEntity 用于从实体的 ID 号获得指向图形数据库中实体的指针，其定义为：

```
CDraft::ErrorStatus crxdbOpenAcDbEntity(
CRxDbEntity*& pEnt,
CRxDbObjectId id,
CRxDb::OpenMode mode,
bool openErasedEntity = false);
```

第一个参数返回指向图形数据库实体的指针；第二个参数输入了要获得的实体的 ID 号；第三个参数指定了打开该实体的方式，如果仅是查询该实体的特性用“读”模式打开即可，要修改实体的特性就必须用“写”模式打开；第四个参数指定是否允许访问一个已经被删除的实体。

ObjectCRX 提供了另外两个全局函数 crxdbOpenCRxDbObject 和 crxdbOpenObject 来实现类似的功能，这三个函数的区别在与适用范围：

 crxdbOpenCRxDbEntity：适用于打开继承于 CRxDbEntity 的数据库常驻对象，这类对象一般都能在图形窗口中显示，如直线、圆等。

□ crxdbOpenCRxDbObject：适用于打开未继承于 CRxDbEntity 的数据库常驻对象，这类对象不能在图形窗口中显示，如层表、线型表等。

□crxdbOpenObject：如何不知道要打开的对象是否继承于 CRxDbEntity 类，可以使用这个函数。

打开某个对象之后，使用 close 函数就可以将其关闭。

2．ID（CRxDbObjectId）、指针、句柄（Handle）和 crx_name

访问实体的特性必须通过对象指针，但是一旦你获得了实体的 ID、句柄或者 crx_name，都能通过 ID 作中介而获得对象的指针。ID、指针、句柄和 crx_name 的关系如图 2.5 所示，其中 ID 是一个桥梁。

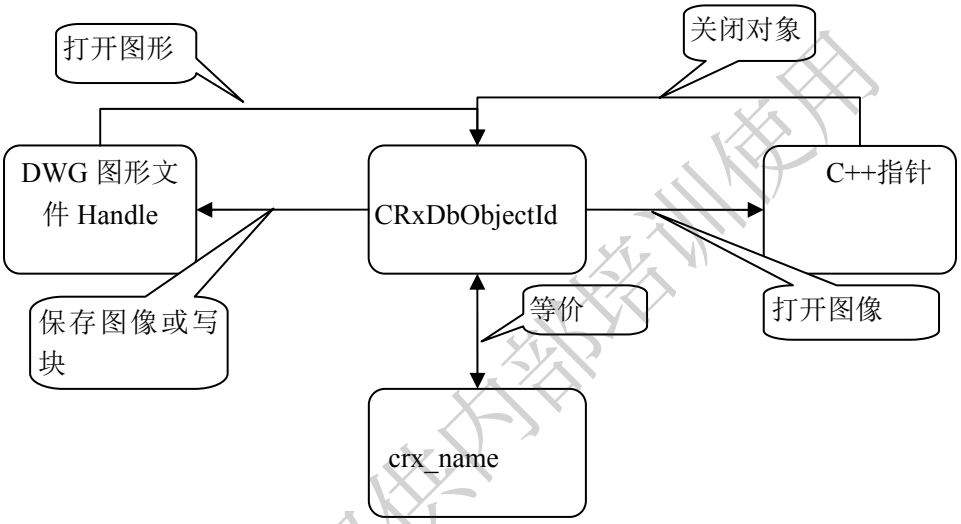


图2.5 ID、指针、句柄和crx_name 的关系

想来 ID 和指针大家都比较了解，有必要解释一下句柄和 crx_name。句柄是 Windows 编程一个常用的概念，在 ObjectCRX 编程中一般指 CRxDbHandle 类（如果是指 Windows 编程的界面元素，可以从上下文环境中区分），该类封装了一个 64 位整形标识符，随 EXB 文件一同保存。crx_name 实际上是一个二维数组，数组元素类型为长整型，在与用户交互的函数中还会经常用到。

ID、句柄和 crx_name 具有各自的特点：

□ ID：在一个电子图板任务中，可能会加载多个图形数据库，但是所有对象的 ID 在本次任务中都是独一无二的。在不同的电子图板任务中，同一个图形对象的 ID 可能不同。

□ 句柄：在一个 电子图板任务中，不能保证每个对象的句柄都唯一，但是在一个图形数据库中所有对象的句柄都是唯一的。句柄随 DWG 图形一起保存，在两次任务期间同一对象的句柄是相同的。

□ crx_name：是不稳定的，仅当你在电子图板的一个特定图形中工作时可以使用，一旦退出电子图板或者切换到另一个图形，crx_name 就会丢失。

具体来说，ID、指针、句柄和 crx_name 之间具有下面的转换关系（不完全归纳，不常用的转换并未提及）：

- 从 ID 到对象指针：通过打开数据库对象的三个函数 crxdbOpenAcDbEntity、crxdbOpenCRxDbObject 和 acdbOpenObject 中的任何一个。
- 从对象指针到 ID：所有的数据库常驻对象都继承自 CRxDbObject，而 CRxDbObject 类包含的 objectId 函数能获得所指向对象的 ID。
- 从句柄到 ID：使用 CRxDbDatabase::getCRxDbObjectId 函数。
- 从 ID 到句柄：使用 CRxDbObjectId::handle 函数。
- 从指针到句柄：使用 CRxDbObject::getCRxDbHandle 函数。
- 从 crx_name 到 ID：使用全局函数 crxdbGetObjectId。
- 从 ID 到 crx_name：使用全局函数 crxdbGetAdsName。

2.2.3 步骤

（1）Visual Studio 2010 中，修改已经创建一个上一节项目（名称为 CreateLine）

（2）为了充分利用上一节的成果，将 CreateLine 函数复制到本节工程的命令实现文件 CrxEntryPoint.cpp 中，然后进行一点改动，具体代码为：

```
CRxDbObjectId CreateLine()
{

    // 在内存上创建一个新的CRxDbLine对象
    CRxGePoint3d ptStart(0, 0, 0);
    CRxGePoint3d ptEnd(100, 100, 0);
    CRxDbLine *pLine = new CRxDbLine(ptStart, ptEnd);

    // 获得指向块表的指针
    CRxDbBlockTable *pBlockTable;
    crxdbHostApplicationServices()->workingDatabase()->getSymbolTable(pBlockTable,
CRxDb::kForRead);


    // 获得指向特定的块表记录（模型空间）的指针
    CRxDbBlockTableRecord *pBlockTableRecord;
  pBlockTable->getAt(CRXDB_MODEL_SPACE,
  pBlockTableRecord,CRxDb::kForWrite);

    // 将CRxDbLine类的对象添加到块表记录中
    CRxDbObjectId lineId;
    pBlockTableRecord->appendAcDbEntity(lineId, pLine);
```

```
    // 关闭图形数据库的各种对象
    pBlockTable->close();
    pBlockTableRecord->close();
    pLine->close();

    return lineId;

}
```

与上一节的函数相比，函数的返回值类型为 CRxDbObjectId，并且在函数实现部分增加了返回的语句。return 语句返回了图形数据库为新添加的直线分配的 ID。

此外，CRxDbLine 类所需要的头文件包含语句也要添加到该文件中：

```
#include "dbents.h"
```

（3）创建 ChangeColor 函数，用于修改指定实体的颜色。该函数包含了两个参数，第一个参数指定了要修改颜色的实体的 ID，第二个参数指定了要使用的颜色索引值，具体代码为：

```
CDraft::ErrorStatus ChangeColor(CRxDbObjectId entId, CAXA::UInt16 colorIndex)
{
CRxDbEntity *pEntity;
// 打开图形数据库中的对象
crxdbOpenObject(pEntity, entId, AcDb::kForWrite);
// 修改实体的颜色
pEntity->setColorIndex(colorIndex);
// 用完之后，及时关闭
pEntity->close();
return CDraft::eOk;
}
```

正如前面介绍，使用 crxdbOpenObject 函数实现了从 ID 到对象指针的转换，进而使用 CRxDbEntity::setColorIndex 函数设置对象的颜色。参数中的 colorIndex 实际上就是电子图板中所使用的颜色索引，可以指定 0～256 的值，其中 0 代表随块，256 代表随层。在设置特性之后，记得要将打开的实体关闭。

CDraft::ErrorStatus 的含义前面已经介绍过，它的目的就相当于一般使用 BOOL 类型作为函数的返回值，用来标识函数是否执行成功。CDraft::ErrorStatus 是一个枚举类型，定义了多个可取的值，用来作返回值能够显示各种不同类型的错误，比使用 BOOL 类型含义要丰富。

（4）在 ChangeColor 命令的实现函数中，添加下面的代码：

```
void ChangeColor()
{
// 创建直线
CRxDbObjectId lineId;
```

```
lineId = CreateLine();
// 修改直线的颜色
ChangeColor(lineId, 1);
}
```

代码很简单，首先调用自定义函数 CreateLine 创建一条直线，然后调用函数 ChangeColor 函数修改直线的颜色为红色。在这个过程中，**CRxDbObjectId** 类型的局部变量 **lineId** 作为了传递对象的中介，这种方法在 **ObjectCRX** 中非常普通。

2.2.4 效果

在 Visual Studio 2010 中，按下快捷键 F7，编译并运行程序。启动电子图板 2011 中，加载生成的 CRX 文件，在命令行执行 ChangeColor 命令，就能在图形窗口的左下角得到一条红色的直线。

2.2.5 小结

这一节是结束的时候了，但是还有点意犹未尽，总觉得还有两个重要的问题没有解决，于是在小结里一并提出。

1．用类来组织函数

如果你对 C 语言还算熟悉，你应该会主动将两个自定义函数 CreateLine 和 ChangeColor 放在 ChangeColor 函数前面，于是本节的程序成功编译；如果你对 C 语言的概念还有些含糊，没有这样做，你肯定在怪罪作者给出了一个无法成功编译的程序。真的出现的情况，你最好还是再温习一下 C 和 C++的知识。

即使能成功编译，还是有一个问题：如果这个项目注册了十几个命令，每个命令都要调用若干个自定义函数，那 CrxEntryPoint.cpp 文件的体积会有多大？这些东西看起来是多么难以维护？这才几十个函数，如果是上百个呢？一定要想办法解决这个问题才行。

如果你熟悉面向对象的程序设计，肯定会向导用类来组织这些函数，例如创建一个包含了创建实体的函数的新类 CCreateEnts，一个包含修改实体函数的类 CMofifyEnt。CreateLine

和 ChangeColor 分别作为两个类的成员函数。

由于在本章后面都会使用这种形式来组织函数，因此这里把本节的函数转换一下，用类来组织。下面的步骤创建一个新项目，作为后面部分的基础：

(1) Visual Studio 2010 中，使用 ObjectCRX 向导创建一个新项目（名称为 CRXCreateEnts），在向导中注意要选中【MFC Extention Support】复选框，如图 2.6 所示。

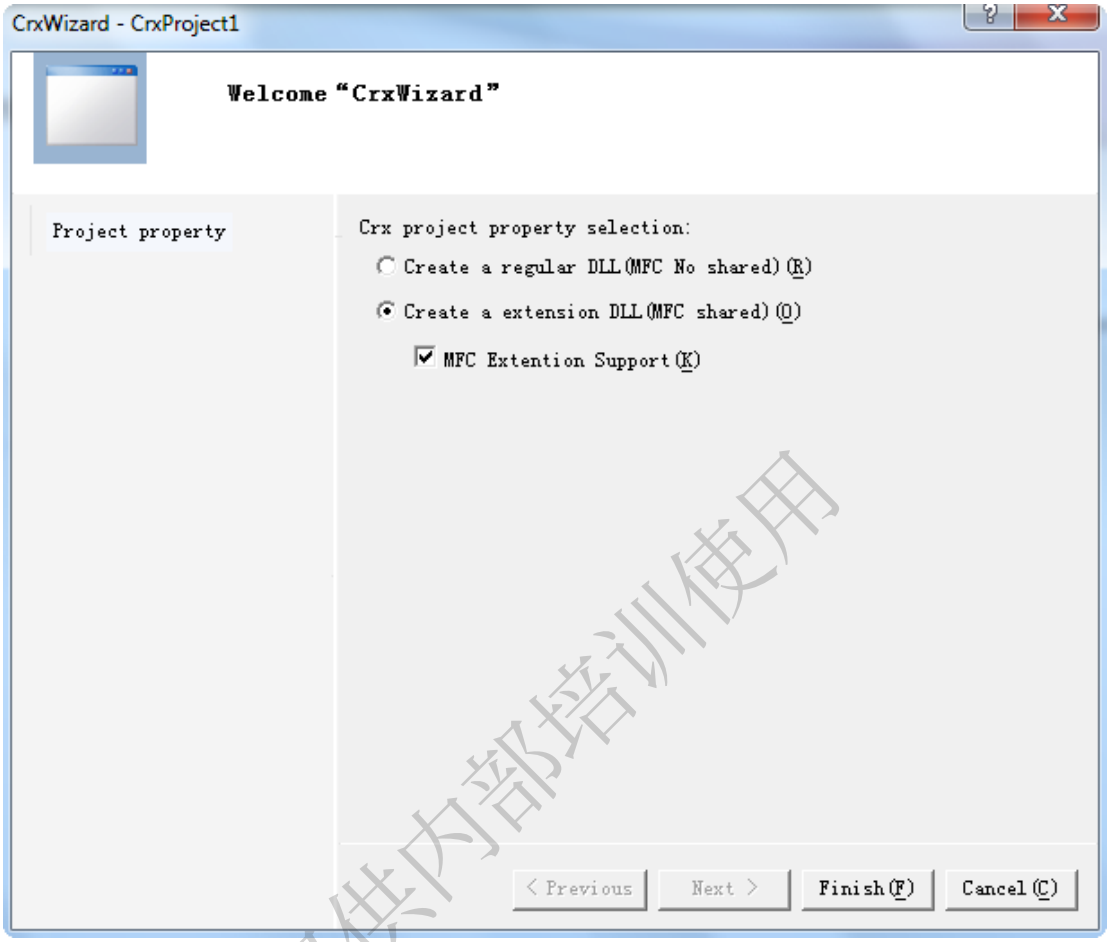


图2.6 设置项目的特性

（2）在visual studio2010的 solution explorer中CreateEnts项目右键，选择【Add/Class】菜单项，系统会弹出如图2.7所示的对话框。输入CCreateEntss作为新类的名称，单击【Finish】按钮创建新类。使用同样的方法，创建一个名为CmodifyEnt的类。

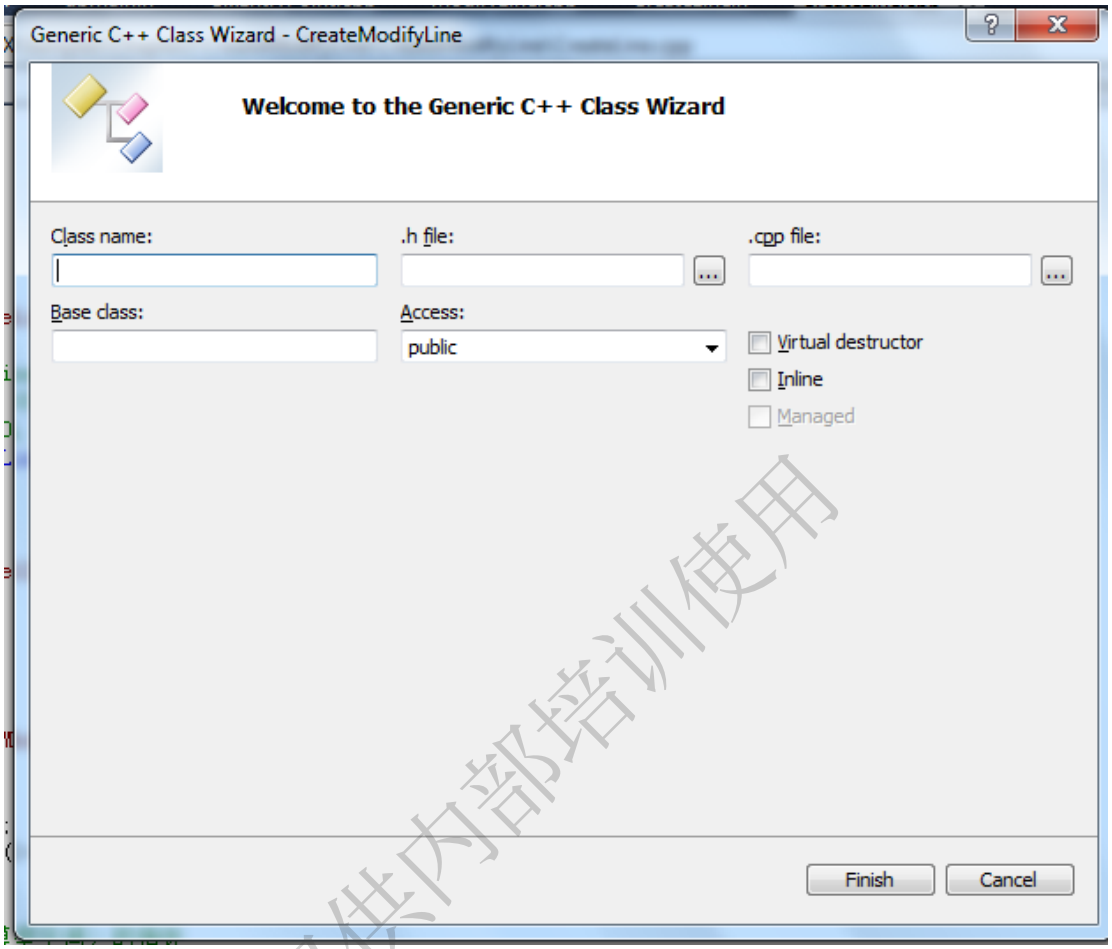


图2.7 创建新类

（3）在 CCreateEnts 和 CModifyEnt 类的头文件中，分别添加包含头文件的语句：

#include "StdAfx.h"

然后在 StdAfx.h 文件中，添加下面的包含语句：

#include "dbents.h"

（4）在 CCreateEnts 类中添加一个 CreateLine 成员函数，其声明和实现形式分别为：

static CRxDbObjectId CreateLine(); // 函数声明

```
CRxDbObjectId CCreateEnts::CreateLine() // 函数实现
{
……
}
```

由于函数的内容并未改变，因此这里不再给出具体的实现代码。

（5）在 CModifyEnt 类中添加一个 ChangeColor 成员函数，其声明和实现形式分别为：

```
static CDraft::ErrorStatus ChangeColor(CRxDbObjectId entId, CAXA::UInt16
colorIndex);
CDraft::ErrorStatus CModifyEnt::ChangeColor(CRxDbObjectId entId, CAXA::UInt16
colorIndex)
```

```
{
……
}
```

（6）此时，如果要注册一个新的命令，实现 ZffCHAP2ChangeColor 函数的功能，就要将代码写成这样：

```
void CRXChangeColor()
{
// 创建直线
CRxDbObjectId lineId;
lineId = CCreateEnts::CreateLine();
// 修改直线的颜色
CModifyEnt::ChangeColor(lineId, 1);
}
```

2．修改实体的其他特性

已经介绍了修改实体颜色的方法，你可以试着写几个其他的函数，例如修改实体的图层和线型，写完之后，可以和下面的样例函数对照一下：

```
CDraft::ErrorStatus CModifyEnt::ChangeLayer(CRxDbObjectId entId,
CString strLayerName)
{
CRxDbEntity *pEntity;
// 打开图形数据库中的对象
crxdbOpenObject(pEntity, entId, CRxDb::kForWrite);
// 修改实体的图层
pEntity->setLayer(strLayerName);
// 用完之后，及时关闭
pEntity->close();
return CDraft::eOk;
}
CDraft::ErrorStatus CModifyEnt::ChangeLinetype(CRxDbObjectId entId,
CString strLinetype)
{
CRxDbEntity *pEntity;
// 打开图形数据库中的对象
crxdbOpenObject(pEntity, entId, AcDb::kForWrite);
// 修改实体的线型
pEntity-> setLinetype (strLinetype);
// 用完之后，及时关闭
pEntity->close();
return CDraft::eOk;
}
```

可以看出，与修改实体颜色的函数相比，修改实体图层和线型的函数仅仅是使用了 CRxDbEntity 类的不同的编辑函数，在打开和关闭数据库对象方面没有任何的区别。

3．提高 CreateLine 函数的可重用性

CCreateEnts::CreateLine 函数仅能创建一条起点为（0，0，0）、终点为（100，100，0）的直线，局限性很大，因此要将它改造一下，便于在程序中调用。另外，考虑到所有的图形对象创建过程都要进行获得图形数据库的块表、获得模型空间的块表记录、将实体添加到模型空间的块表记录等操作，将这些步骤分离出来，封装成一个独立的函数。

于是，CCreateEnts 类现在包含了两个静态成员函数：

```
// 创建直线
CRxDbObjectId CCreateEnts::CreateLine(CRxGePoint3d ptStart,
CRxGePoint3d ptEnd)
{
CRxDbLine *pLine = new CRxDbLine(ptStart, ptEnd);
// 将实体添加到图形数据库
CRxDbObjectId lineId;
lineId = CCreateEnts::AddToModelSpace(pLine);
return lineId;
}
// 将实体添加到图形数据库的模型空间
CRxDbObjectId CCreateEnts::AddToModelSpace(CRxDbEntity* pEnt)
{
CRxDbBlockTable *pBlockTable;
acdbHostApplicationServices()->workingDatabase()
->getBlockTable(pBlockTable, CRxDb::kForRead);
CRxDbBlockTableRecord *pBlockTableRecord;
pBlockTable->getAt(CRXDB_MODEL_SPACE, pBlockTableRecord,
CRxDb::kForWrite);
CRxDbObjectId entId;
pBlockTableRecord->appendCRxDbEntity(entId, pEnt);
pBlockTable->close();
pBlockTableRecord->close();
pEnt->close();
return entId;
}
```

如果要测试上面创建的这些函数，可以在当前项目中注册一个命令 AddLine，其实现函数的代码为：

```
void CRXAddLine()
{
CRxGePoint3d ptStart(0, 0, 0);
CRxGePoint3d ptEnd(100, 100, 0);
```

```
CRxDbObjectId lineId;
lineId = CCreateEnts::CreateLine(ptStart, ptEnd);
CModifyEnt::ChangeColor(lineId, 1);
CModifyEnt::ChangeLayer(lineId, _T("虚线"));
CModifyEnt::ChangeLinetype(lineId, _T("中心线"));
}
```

这段代码同样在模型空间创建一条直线，并将其颜色设置为红色，图层设置为“虚线”层，线型为“中心线”。当然，在程序运行之前，要确保“虚线”层和“中心线”线型的存在。

2.3 创建圆

2.3.1 说明

前面已经分析了创建一个图形对象的基本过程，本节开始就要将着眼点放在创建实体的参数上。本节的实例分别用“圆心、半径”、“直径的两个端点”和“三点法”创建圆。

2.3.2 思路

在 ObjectCRX 中，CRxDbCircle 类用来表示圆。该类有两个构造函数，其形式分别为：

```
CRxDbCircle();
CRxDbCircle(const CRxGePoint3d& cntr, const CRxGeVector3d& nrm, double radius);
```

两个构造函数的名称相同，接受不同的参数，这是 C++中函数的重载。重载是 C++提供的一个很有用的特性，相同功能的函数采用同样的名称，大大减少了程序员的记忆量。

创建一个圆需要三个参数：圆心、半径和圆所在的平面（一般用平面的法向量来表示）。第一个构造函数不接受任何参数，创建一个圆心为（0，0，0）、半径为 0 的圆，其所在平面法向量为（0，0，1）；第二个构造函数则接受了圆心、圆所在平面法向量和半径三个参数。一般来说，习惯于在创建实体时直接将其初始化，很少用第一个构造函数。

2.3.3 步骤

（1）在 Visual Studio 2010 中，选择【File/Open/Project/Solution】菜单项，从弹出的对话框中选择上一节创建的 CreateEnts 项目，打开该项目。注意，直接选择【File/Open/File】菜单项仅会打开一个文件，而不会打开整个项目的所有文件。项目保存了项目所有文件的位置和名称，在打开工作区时就自动加载了项目中所有的文件。

（2）在 CCreateEnts 类中，添加一个新的函数 CreateCircle，该函数直接封装 CRxDbCircle 类的构造函数，其声明和实现分别为：

```
// 声明部分（在CreateEnt.h文件中）
```

```
static CRxDbObjectId CreateCircle(CRxGePoint3d ptCenter,
CRxGeVector3d vec, double radius);
// 实现部分（在CreateEnt.cpp文件中）
//用圆心半径创建圆
CRxDbObjectId CCreateEnts::CreateCircle(CRxGePoint3d ptCenter,CRxGeVector3d vec, double radius)
{
    CRxDbCircle *pCircle = new CRxDbCircle(ptCenter, vec, radius);
    // 将实体添加到图形数据库
    CRxDbObjectId circleId;
    circleId = CCreateEnts::AddToModelSpace(pCircle);
    return circleId;
}
```

（3）在 CCreateEnts 类中，再添加一个新的函数 CreateCircle（这就使用了函数重载的概念），用于创建位于 XOY 平面上的圆（一般创建的二维图形都是在 XOY 平面上），其声明和实现分别为：

```
// 声明部分
static CRxDbObjectId CreateCircle(CRxGePoint3d ptCenter, double radius);
// 实现部分
//圆心点 半径创建圆
CRxDbObjectId CCreateEnts::CreateCircle(CRxGePoint3d ptCenter, double radius)
{
    CRxGeVector3d vec(0, 0, 1);

    return CCreateEnts::CreateCircle(ptCenter, vec, radius);
}
```

代码相当简单，区别就在于输入了一个代表 XOY 平面的法向矢量。

（4）在项目中添加一个新类 CCalculation，用于封装计算的相关函数。添加两个重载静态函数 MiddlePoint，用于计算两点连线的中点：

```
CRxGePoint2d CCalculation::MiddlePoint(CRxGePoint2d pt1, CRxGePoint2d pt2)
{
    CRxGePoint2d pt;
    pt[X] = (pt1[X] + pt2[X]) / 2;
    pt[Y] = (pt1[Y] + pt2[Y]) / 2;
    return pt;
}
CRxGePoint3d CCalculation::MiddlePoint(CRxGePoint3d pt1, CRxGePoint3d pt2)
{
    CRxGePoint3d pt;
    pt[X] = (pt1[X] + pt2[X]) / 2;
```

```
    pt[Y] = (pt1[Y] + pt2[Y]) / 2;
    pt[Z] = (pt1[Z] + pt2[Z]) / 2;
    return pt;
}
```

（5）在 CCreateEnts 类中，添加一个两点法创建圆的函数：

```
/取两点中心为圆心
CRxDbObjectId CCreateEnts::CreateCircle(CRxGePoint2d pt1, CRxGePoint2d pt2)
{
    // 计算圆心和半径
    CRxGePoint2d pt = CCalculation::MiddlePoint(pt1, pt2);
    CRxGePoint3d ptCenter(pt[X], pt[Y], 0); // 圆心
    double radius = pt1.distanceTo(pt2) / 2;
    // 创建圆
    return CCreateEnts::CreateCircle(ptCenter, radius);
}
```

上面的代码中，调用 CCalculation::MiddlePoint 函数来获得两点连线的中点，也就是圆心；CRxGePoint2d::distanceTo 函数用于计算两点之间的距离。

（7）使用几何类来实现三点法画圆的函数。在 ObjectCRX 中提供了一个以 CRxGe 开头的类库（一般称为几何类），用来完成一些计算工作，关于几何类会在详细介绍，这里使用了 CRxGeCircArc3d 类来完整需要的工作。与数学方法的函数相比，代码相当简洁：

```
CRxDbObjectId CCreateEnts::CreateCircle(CRxGePoint2d pt1, CRxGePoint2d pt2,CRxGePoint2d pt3)
{
    // 使用几何类
    CRxGeCircArc2d geArc(pt1, pt2, pt3);
    CRxGePoint3d ptCenter(geArc.center().x, geArc.center().y, 0);
    return CCreateEnts::CreateCircle(ptCenter, geArc.radius());
}
```

CRxGeCircArc2d 类能够创建一个几何类的圆弧对象，该对象仅用来计算，不能在图形窗口中显示。CRxGeCircArc2d 类有四个构造函数，其中一个构造函数可以根据三个点创建圆弧，这里正是使用该函数创建了几何类的圆弧。要使用该类，必须添加下面的语句：

```
#include "gearc3d.h"
```

构建几何类的圆弧之后，就可以查询该对象的圆心、所在平面、半径等特性，将这些参数传递到创建圆的函数中，就可以实现三点法创建圆。

2.3.4 效果

（1）在项目中注册一个新命令 cecircle，对几个创建圆的函数进行测试，该命令的实现函数为：

```
void CRXCreateCircle()
{
```

```
    // “圆心、半径”法创建一个圆
    CRxGePoint3d ptCenter(100, 100, 0);
    CCreateEnts::CreateCircle(ptCenter, 20);
    // 两点法创建一个圆
    CRxGePoint2d pt1(70, 100);
    CRxGePoint2d pt2(130, 100);
    CCreateEnts::CreateCircle(pt1, pt2);
    // 三点法创建一个圆
    pt1.set(60, 100);
    pt2.set(140, 100);
    CRxGePoint2d pt3(100, 60);
    CCreateEnts::CreateCircle(pt1, pt2, pt3);
}
```

（2）编译并运行该程序，在电子图板 2011 中运行 cecircle 命令，就能得到如图 2.8 所示的结果。

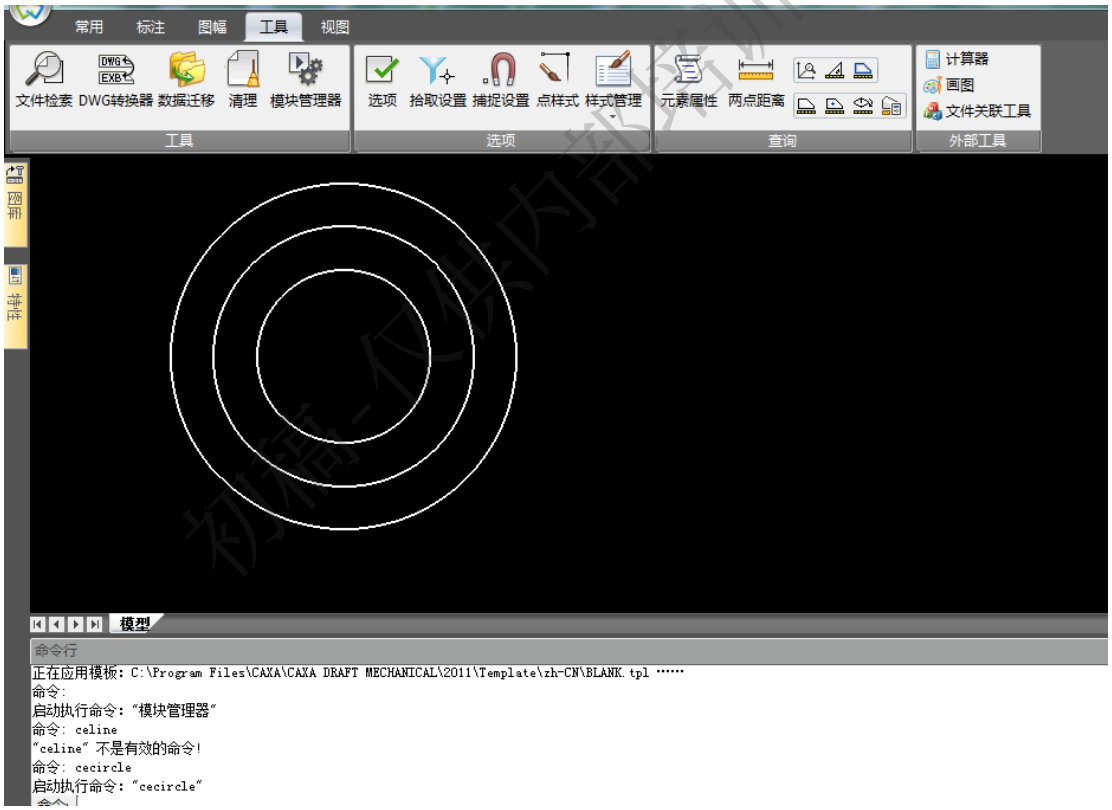


图2.8 命令执行结果

2.3.5 小结

本节开始，们已经将注意力集中在对象的创建上，本节有两个重点：

- 在封装代码时，注意函数重载的使用。
- 使用几何类，实现三点法创建圆。

2.4 创建圆弧

2.4.1 说明

与上一节创建圆的函数相对应，本节将要实现用“圆心、半径、圆弧所在的平面、起点角度和终点角度”、三点法、“起点、圆心、终点”和“起点、圆心、圆弧角度”几种方法来创建圆弧。

2.4.2 思路

在 ObjectCRX 中，CRxDbArc 类被用来表示圆弧，该类有三个构造函数：

```
CRxDbArc(
const CRxGePoint3d& center,
double radius,
double startAngle,
double endAngle);
CRxDbArc(
const CRxGePoint3d& center,
const CRxGeVector3d& normal,
double radius,
double startAngle,
double endAngle);
CRxDbArc();
```

第二个构造函数接受最多的参数，因此首先对该函数进行封装，其他几个函数均以封装后的函数为基础。

2.4.3 步骤

（1）打开 CreateEnts 项目，在 CCalculation 类中增加一个新的函数 Pt2dTo3d，其实现代码为：

```
CRxGePoint3d CCalculation::Pt2dTo3d(CRxGePoint2d pt)
{
CRxGePoint3d ptTemp(pt.x, pt.y, 0);
return ptTemp;
}
```

（2）在 CCreateEnts 类中添加一个函数 CreateArc，用于向模型空间添加一个圆弧，其实现代码为：

```
//创建圆弧
CRxDbObjectId CCreateEnts::CreateArc(CRxGePoint3d ptCenter, CRxGeVector3d vec,double radius,
double startAngle, double endAngle)
{
    CRxDbArc *pArc = new CRxDbArc(ptCenter, vec, radius, startAngle,
        endAngle);
    CRxDbObjectId arcId;
    arcId = CCreateEnts::AddToModelSpace(pArc);
    return arcId;
}
```

（3）添加一个创建位于 XOY 平面上的圆弧的函数，其实现代码为：

```
//XOY 平面创建圆弧
CRxDbObjectId CCreateEnts::CreateArc(CRxGePoint2d ptCenter, double radius,  double startAngle,
double endAngle)
{
    CRxGeVector3d vec(0, 0, 1);
    return CCreateEnts::CreateArc(CCalculation::Pt2dTo3d(ptCenter),
        vec, radius, startAngle, endAngle);
}
```

（4）三点法创建圆弧，可以使用下面的函数：

```
//三点法创建圆弧
CRxDbObjectId CCreateEnts::CreateArc(CRxGePoint2d ptStart, CRxGePoint2d ptOnArc,CRxGePoint2d
ptEnd)
{
    // 使用几何类获得圆心、半径
    CRxGeCircArc2d geArc(ptStart,ptOnArc,ptEnd);
    CRxGePoint2d ptCenter = geArc.center();
    double radius = geArc.radius();
    // 计算起始和终止角度
    CRxGeVector2d vecStart(ptStart.x - ptCenter.x, ptStart.y - ptCenter.y);
    CRxGeVector2d vecEnd(ptEnd.x - ptCenter.x, ptEnd.y - ptCenter.y);
    double startAngle = vecStart.angle();
    double endAngle = vecEnd.angle();
    return CCreateEnts::CreateArc(ptCenter, radius, startAngle, endAngle);
}
```

CRxGeVector2d 类用来表示一个二维空间中的矢量，其成员函数 angle 返回该矢量和 X 轴

正半轴的角度（用弧度来表示）。

（5）使用“起点、圆心、终点”方法来创建圆弧，可以使用下面的函数：

```
//起点 圆心 终点 创建圆弧
CRxDbObjectId CCreateEnts::CreateArcSCE(CRxGePoint2d ptStart, CRxGePoint2d
ptCenter,CRxGePoint2d ptEnd)
{
    // 计算半径
    double radius = ptCenter.distanceTo(ptStart);
    // 计算起、终点角度
    CRxGeVector2d vecStart(ptStart.x - ptCenter.x, ptStart.y - ptCenter.y);
    CRxGeVector2d vecEnd(ptEnd.x - ptCenter.x, ptEnd.y - ptCenter.y);
    double startAngle = vecStart.angle();
    double endAngle = vecEnd.angle();
    // 创建圆弧
    return CCreateEnts::CreateArc(ptCenter, radius, startAngle, endAngle);
}
```

这个函数的名称不再是 CreateArc，而是 CreateArcSCE，这是因为该函数的参数列表、返回值都与三点法的函数相同，无法实现函数的重载，就只能重新定义一个新的函数名称。

（6）使用“起点、圆心、圆弧角度”方法来创建圆弧，可以使用下面的函数：

```
//起点 圆心 圆弧角度 创建圆弧

CRxDbObjectId CCreateEnts::CreateArc(CRxGePoint2d ptStart, CRxGePoint2d ptCenter,double angle)
{
    // 计算半径
    double radius = ptCenter.distanceTo(ptStart);
    // 计算起、终点角度
    CRxGeVector2d vecStart(ptStart.x - ptCenter.x, ptStart.y - ptCenter.y);
    double startAngle = vecStart.angle();
    double endAngle = startAngle + angle;
    // 创建圆弧
    return CCreateEnts::CreateArc(ptCenter, radius, startAngle, endAngle);
}
```


2.4.4 效果

（1）在 CCalculation 类中添加一个新的函数 PI，用来计算常量 π 的值，其实现代码为：

```
double CCalculation::PI()
{
return 4 * atan(1.0);
}
```

atan 是一个 C 语言的库函数，用来计算反正切函数的值，在使用该函数时需要添加下面的语句：

```
#include <math.h>
```

（2）在项目中注册一个新命令 cearc，该命令的实现函数为：

```
void CRXCreateArc()
{
    // 创建位于XOY平面上的圆弧
    AcGePoint2d ptCenter(50, 50);
    CCreateEnts::CreateArc(ptCenter, 100 * sqrt(2.0)/2, 5 * CCalculation::PI() / 4, 7 * CCalculation::PI() / 4);
    // 三点法创建圆弧
    AcGePoint2d ptStart(100, 0);
    AcGePoint2d ptOnArc(120, 50);
    AcGePoint2d ptEnd(100, 100);
    CCreateEnts::CreateArc(ptStart, ptOnArc, ptEnd);
    // “起点、圆心、终点”创建圆弧
    ptStart.set(100, 100);
    ptCenter.set(50, 50);
    ptEnd.set(0, 100);
    CCreateEnts::CreateArcSCE(ptStart, ptCenter, ptEnd);
    // “起点、圆心、圆弧角度”创建圆弧
    ptStart.set(0, 100);
    ptCenter.set(50, 50);
    CCreateEnts::CreateArc(ptStart, ptCenter, CCalculation::PI() / 2);
}
```

要成功编译该程序，必须在 CrxEntryPoint.cpp 文件中添加下面的语句：

```
#include "Calculation.h"
```

（3）编译运行程序，在电子图板 2011 中运行 cearc 命令，能够得到如图 2.9 所示的结果。实际上，这个图形由四个圆弧组成。

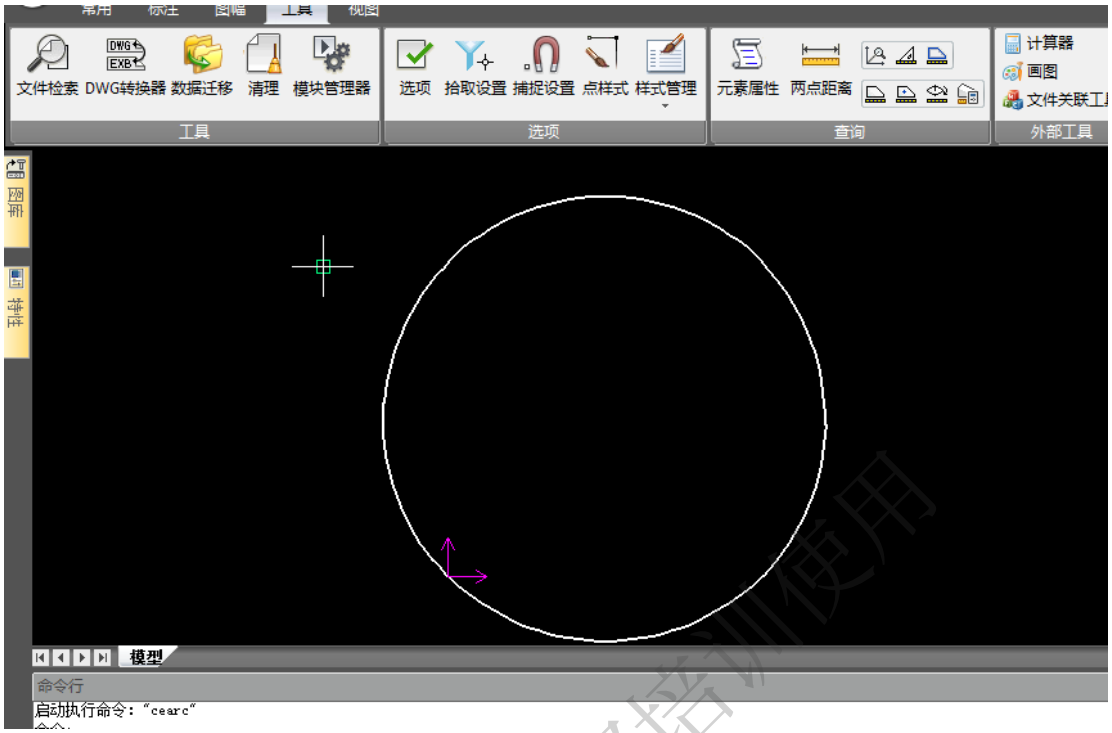


图2.9 命令执行结果

2.4.5 小结

学习本节的实例之后，需要注意下面的问题：

- 使用 CRxGeVector2d 类。
- 获得常量 π 的精确值。

2.5 创建多段线

2.5.1 说明

本节介绍若干个函数，分别用于创建轻量多段线、三维多段线、正多边形、矩形、圆（圆环）和圆弧。这里所说的圆形和 2.3 节创建的圆不一样，因为本节创建的圆实际上是一条闭合的多段线，可以设置线宽。

2.5.2 思路

ObjectCRX 中提供了三种多段线的相关类： CRxDbPolyline 、CRxDb2dPolyline

。其中，利用电子图板的内部命令可以创建 CRxDbPolyline 类的对象，用 PLINE 命令创建的对象是轻量多段线（CRxDbPolyline）。

创建轻量多段线和三维多段线的函数，直接封装 CRxDbPolyline 和 类的构造函数即可；创建正多边形、矩形、圆形和圆环，实际上都是创建了特殊形状的轻量多段线，创建这些对象的关键都在于顶点和凸度的确定。

2.5.3 步骤

（1）打开 CreateEnts 项目，添加一个函数 CreatePolyline，用于创建轻量多段线：

```
// 声明
static CRxDbObjectId CreatePolyline(CRxGePoint2dArray points, double width = 0);
// 实现
//创建多段线
CRxDbObjectId CCreateEnts::CreatePolyline (CRxGePoint2dArray points,double width)
{
    int numVertices = points.length();
    CRxDbPolyline *pPoly = new CRxDbPolyline();
    for(int i = 0; i < numVertices; i++)
    {
        pPoly->addVertexAt(i, points.at(i), 0, width,width);
    }
    CRxDbObjectId polyId;
    polyId = CCreateEnts::AddToModelSpace(pPoly);

    return polyId;
}
```

在 CreatePolyline 函数声明中，为 width 参数设置了默认的实参值 0，这样可以减少该函数被调用时的输入量，如果对此还有疑问，请参阅 C++语法中对默认实参值的说明。注意，不能同时在函数的声明和定义中指定默认参数值，一般可在函数的声明中指定。创建 CRxDbPolyline 对象可以分成两个步骤：创建类的实例和添加顶点。创建类的实例，以多段线的顶点个数作为参数调用 CRxDbPolyline::CRxDbPolyline 函数；添加顶点时，则使用了 CRxDbPolyline::addVertexAt 函数，将每一个顶点添加到多段线中。

addVertexAt 函数定义为：

```
CDraft::ErrorStatus addVertexAt(
unsigned int index,
const AcGePoint2d& pt,
double bulge = 0.,
double startWidth = -1.,
double endWidth = -1);
```

index 用来指定插入顶点的索引号（从 0 开始）；pt 指定顶点的位置；bulge 指出要

创建的顶点的凸度；startWidth 和 endWidth 指定了从该顶点到下一顶点之间连线的起始和终止线宽，利用该特性可以使用多段线创建一个实心箭头。凸度是多段线中一个比较难理解的概念，用于指定当前顶点的平滑性，其被定义为：在多段线顶点显示中，选取顶点与下一个顶点形成的弧之间角度的四分之一的正切值。凸度可以用来设置多段线某一段的凸出参数，0 表示直线，1 表示半圆，介于 0～1 之间为劣弧大于 1 为优弧。在创建圆的函数中，你会进一步学习。

（2）添加一个函数，创建仅包含一条直线的多段线，该函数的实现代码为：

```
//创建一条包含 一条直线的多段线
CRxDbObjectId CCreateEnts::CreatePolyline(CRxGePoint2d ptStart,CRxGePoint2d ptEnd, double width)
{
    CRxGePoint2dArray points;
    points.append(ptStart);
    points.append(ptEnd);
    return CCreateEnts::CreatePolyline(points, width);
}
```

CRxGePoint2dArray 类的使用非常简单，append 函数用于向数组中添加一个二维点，removeAt 函数用于从数组中删除指定的元素，length 函数返回数组的长度。

（3）在 CModifyEnt 类中，添加一个函数 Rotate，按照指定的角度（用弧度值表示）旋转指定的实体，其实现代码为：

```
//按照指定的角度旋转实体
CDraft::ErrorStatus CModifyEnt::Rotate(CRxDbObjectId entId, CRxGePoint2d ptBase, double rotation)
{
    CRxGeMatrix3d xform;
    CRxGeVector3d vec(0, 0, 1);
    xform.setToRotation(rotation, vec, CCalculation::Pt2dTo3d(ptBase));
    CRxDbEntity *pEnt;
    CDraft::ErrorStatus es = crxdbOpenObject(pEnt, entId, CRxDb::kForWrite,false);
    pEnt->transformBy(xform);
    pEnt->close();
    return es;
}
```

上面的函数使用 CRxDbEntity::transformBy 函数对实体进行旋转，该函数能对实体进行比例变换、移动和旋转操作，其输入参数是一个几何变换矩阵（CRxGeMatrix3d 类）。CRxGeMatrix3d 类用于实体的几何变换非常简单，只要使用 setToScaling、setToRotation 和 setToTranslation 三个函数设置所要进行的变换，然后对所要变换的实体执行 transformBy 函数即可。有了这个基础，不妨一鼓作气，写出另外两个函数 Move 和 Scale，以备以后使用：

```
CDraft::ErrorStatus CModifyEnt::Move(CRxDbObjectId entId, CRxGePoint3d ptBase,CRxGePoint3d
ptDest)
{
```

```
    // 设置变换矩阵的参数
    CRxGeMatrix3d xform;
    CRxGeVector3d vec(ptDest.x - ptBase.x, ptDest.y - ptBase.y,
        ptDest.z - ptBase.z);
    xform.setToTranslation(vec);
    CRxDbEntity *pEnt;
    CDraft::ErrorStatus es = crxdbOpenObject(pEnt, entId, CRxDb::kForWrite,false);
    pEnt->transformBy(xform);
    pEnt->close();
    return es;
}
CDraft::ErrorStatus CModifyEnt::Scale(CRxDbObjectId entId,CRxGePoint3d ptBase, double scaleFactor)
{
    // 设置变换矩阵的参数
    CRxGeMatrix3d xform;
    xform.setToScaling(scaleFactor, ptBase);
    CRxDbEntity *pEnt;
    CDraft::ErrorStatus es = acdbOpenObject(pEnt, entId, CRxDb::kForWrite,false);
    pEnt->transformBy(xform);
    pEnt->close();
    return es;
}
```

（4）添加创建正多边形的函数。创建正多边形的输入参数为中心、边数、外接圆半径、旋转角度（弧度值）和线宽，其实现代码为：

```
//创建正多边形
CRxDbObjectId CCreateEnts::CreatePolygon(CRxGePoint2d ptCenter, int number,double radius, double
rotation, double width)
{
    CRxGePoint2dArray points;
    double angle = 2 * CCalculation::PI() / (double)number;
    for (int i = 0; i < number; i++)
    {
        CRxGePoint2d pt;
        pt.x = ptCenter.x + radius * cos(i * angle);
        pt.y = ptCenter.y + radius * sin(i * angle);
        points.append(pt);
    }
    CRxDbObjectId polyId = CCreateEnts::CreatePolyline(points, width);
    // 将其闭合
    CRxDbEntity *pEnt;
```

```
    acdbOpenAcDbEntity(pEnt, polyId, CRxDb::kForWrite);
    CRxDbPolyline *pPoly = CRxDbPolyline::cast(pEnt);
    if (pPoly != NULL)
    {
        //多段线闭合　失败？
        pPoly->setClosed(CAXA::kTrue);
    }
    pEnt->close();
    CModifyEnt::Rotate(polyId, ptCenter, rotation);
    return polyId;
}
```

创建正多边形，实际上有四个步骤：（a）计算顶点位置；（b）根据顶点位置创建多段线；（c）闭合多段线；（d）旋转多段线。其中，步骤（a）、（b）和（d）不用再多说，步骤（c）是新接触到的内容。前面已经介绍过，使用 crxdbOpenAcDbEntity 函数能够得到一个指向 CRxDbEntity 对象的指针，但是如何将其转化为指向 CRxDbPolyline 的指针，从而利用 CRxDbPolylien 类的函数修改其特性呢？注意下面的语句：

```
CRxDbPolyline *pPoly = CRxDbPolyline::cast(pEnt);
```

cast 是 ObjectCRX 所有类的基类—CRxRxObject 中实现的一个函数，该函数提供了一种安全的类型转换机制，这里的作用就是从基类指针 pEnt 获得派生类的指针 pPoly。如果你想
为这条语句加上一个错误处理，还可以写成这样的形式：

```
if (pEnt->isKindOf(CRxDbPolyline::desc()) == CAXA::kTrue)
{
CRxDbPolyline *pPoly = CRxDbPolyline::cast(pEnt);
if (pPoly != NULL)
{


pPoly->setClosed(CAXA::kTrue);
}
}
```

isKindOf 和 desc 函数提供了 ObjectCRX 中一种常用的动态类型检查机制，这种方法在 ObjectCRX 编程中极为常见。

（5）添加用于创建矩形的函数。该函数根据两个角点和线宽来创建矩形，其实现代码为：

```
//创建矩形
CRxDbObjectId CCreateEnts::CreateRectangle(CRxGePoint2d pt1, CRxGePoint2d pt2,double width)
{
    // 提取两个角点的坐标值
```

```
    double x1 = pt1.x, x2 = pt2.x;
    double y1 = pt1.y, y2 = pt2.y;
    // 计算矩形的角点
    CRxGePoint2d ptLeftBottom(CCalculation::Min(x1, x2),
        CCalculation::Min(y1, y2));
    CRxGePoint2d ptRightBottom(CCalculation::Max(x1, x2),
        CCalculation::Min(y1, y2));
    CRxGePoint2d ptRightTop(CCalculation::Max(x1, x2),
        CCalculation::Max(y1, y2));
    CRxGePoint2d ptLeftTop(CCalculation::Min(x1, x2),
        CCalculation::Max(y1, y2));
    // 创建对应的多段线
    CRxDbPolyline *pPoly = new CRxDbPolyline();
    pPoly->addVertexAt(0, ptLeftBottom, 0, width, width);
    pPoly->addVertexAt(1, ptRightBottom, 0, width, width);
    pPoly->addVertexAt(2, ptRightTop, 0, width, width);
    pPoly->addVertexAt(3, ptLeftTop, 0, width, width);
    pPoly->setClosed(Adesk::kTrue);
    // 将多段线添加到模型空间
    CRxDbObjectId polyId;
    polyId = CCreateEnts::AddToModelSpace(pPoly);
    return polyId;
}
```

CCalculation::Min 和 CCalculation::Max 是两个自定义函数，分别用于获得两个数的最大值额最小值，其实现代码为：

```
double CCalculation::Max(double a, double b)
{
    if (a > b)
    {
        return a;
    }
    else
    {
        return b;
    }
}
double CCalculation::Min(double a, double b)
{
    if (a < b)
```

```
    {
        return a;
    }
    else
    {
        return b;
    }
}
```

（6）添加创建圆的函数，输入参数包括圆心、半径和线宽，其实现代码为：

```
//创建圆弧的函数，输入参数包括圆心、半径、起始角度、终止角度和线宽
CRxDbObjectId CCreateEnts::CreatePolyCircle(CRxGePoint2d ptCenter, double radius, double width)
{
    // 计算顶点的位置
    CRxGePoint2d pt1, pt2, pt3;
    pt1.x = ptCenter.x + radius;
    pt1.y = ptCenter.y;
    pt2.x = ptCenter.x - radius;
    pt2.y = ptCenter.y;
    pt3.x = ptCenter.x + radius;
    pt3.y = ptCenter.y;
    // 创建多段线
    CRxDbPolyline *pPoly = new CRxDbPolyline();

   pPoly->addVertexAt(0, pt1, 1, width, width);
    pPoly->addVertexAt(1, pt2, 1, width, width);
    pPoly->addVertexAt(2, pt3, 1, width, width);
    pPoly->setClosed(Adesk::kTrue);
    // 将多段线添加到模型空间
    CRxDbObjectId polyId;
    polyId = CCreateEnts::AddToModelSpace(pPoly);
    return polyId;
}
```

其他的东西无须多谈，继续来讨论一下凸度的问题。前面在使用 addVertexAt 函数时，总是指定 0 作为凸度值，上面的函数中则指定了 1 作为凸度。

图 2.10 用来帮助你更好地认识凸度。左侧的半圆起点（顶点索引号为 0）凸度为 1，右侧的半圆起点凸度则为－1，凸度的不同导致圆弧的方向不同。

图2.10 凸度的含义

（8）添加用于创建圆弧的函数，输入参数包括圆心、半径、起始角度、终止角度和线宽，其实现代码为：

```
//创建圆弧的函数，输入参数包括圆心、半径、起始角度、终止角度和线宽，其实现代码为：
    CRxDbObjectId CCreateEnts::CreatePolyArc(CRxGePoint2d ptCenter, double radius,double
angleStart, double angleEnd, double width)
{
    // 计算顶点的位置
    CRxGePoint2d pt1, pt2;
    pt1.x = ptCenter.x + radius * cos(angleStart);
    pt1.y = ptCenter.y + radius * sin(angleStart);
    pt2.x = ptCenter.x + radius * cos(angleEnd);
    pt2.y = ptCenter.y + radius * sin(angleEnd);
    // 创建多段线
    CRxDbPolyline *pPoly = new CRxDbPolyline();

    pPoly->addVertexAt(0, pt1, tan((angleEnd - angleStart) / 4), width,
        width);
    pPoly->addVertexAt(1, pt2, 0, width, width);
    // 将多段线添加到模型空间
    CRxDbObjectId polyId;
    polyId = CCreateEnts::AddToModelSpace(pPoly);
    return polyId;
}
```

如果能够独立编写这个函数，相信你已经领悟了凸度的含义。

2.5.4 效果

（1）在 CCalculation 类中添加两个函数，用于在角度和弧度之间转换，其实现代码为：

```
// 弧度转化为角度
// 弧度转化为角度
double CCalculation::RtoG(double angle)
{
    return angle * 180 / CCalculation::PI();
}
// 角度转化为弧度
double CCalculation::GtoR(double angle)
{
    return angle * CCalculation::PI() / 180;
}
```

（2）注册一个新命令 Cepolyline，该命令的实现函数中，添加对本节所有函数进行测试的代码：

```
//创建多段线
void CRXCreatePolyline()
{
    // 创建仅包含一段直线的多段线
    CRxGePoint2d ptStart(0, 0), ptEnd(100, 100);
    CCreateEnts::CreatePolyline(ptStart, ptEnd, 1);
// 创建正多边形
   CCreateEnts::CreatePolygon(CRxGePoint2d::kOrigin,6,30,0,1);
    // 创建矩形
    CRxGePoint2d pt(60, 70);
    CCreateEnts::CreateRectangle(pt, ptEnd, 1);
    //创建圆
    pt.set(50, 50);
    CCreateEnts::CreatePolyCircle(pt, 30, 1);
    // 创建圆弧
    CCreateEnts::CreatePolyArc(pt, 50, CCalculation::GtoR(45),CCalculation::GtoR(225), 1);
}
```

（3）编译运行程序，在电子图板 2011 中执行 Cepolyline 命令，得到如图 2.11 所示的结果。

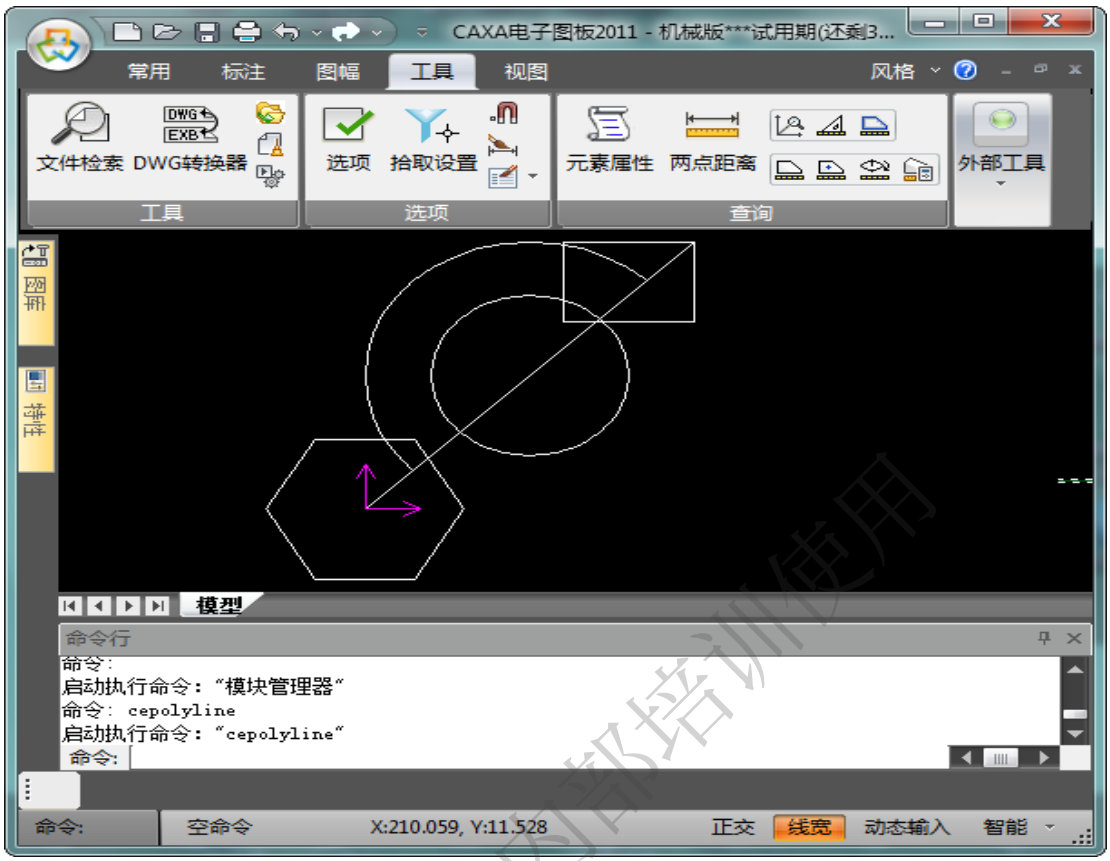


图2.11 测试多段线的创建

2.5.5 小结

学习本节内容之后，下面的几个知识点需要牢固掌握：

- 自定义函数中使用默认参数值；
- 多段线中凸度的含义；
- 使用变换矩阵对实体进行变换；
- ObjectCRX 的动态类型检查机制。

2.6 创建椭圆和样条曲线

2.6.1 说明

本节的实例介绍了创建椭圆和样条曲线的方法，创建椭圆的方法包括对 CRxDbEllipse 类构造函数的直接封装和根据外接矩形创建椭圆，创建样条曲线的方法仅提供了对

CRxDbSpline 类构造函数的封装。

2.6.2 思路

对于创建椭圆对象，可以直接使用 CRxDbEllipse 类的构造函数，给定中心点、所在平面、长轴的一个端点和半径比例来创建椭圆。半径比例是一个用来定义椭圆短轴相对于长轴的比例的参数，半径比例为 1 时椭圆变成圆，帮助系统中给出了一定半径比例时的椭圆形状，如图 2.12 所示。

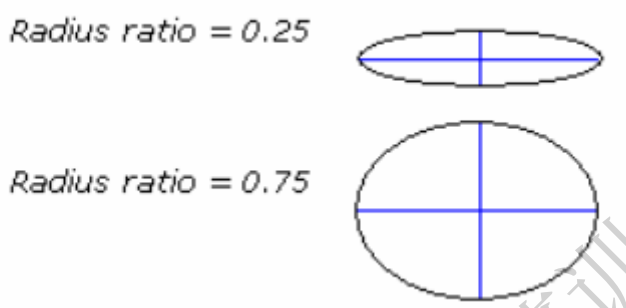


图2.12 半径比例的意义

根据外接矩形创建椭圆，椭圆长、短轴的端点是外接矩形四条边的中点，因此只要外接矩形的角点确定，椭圆的大小和形状就能计算出来。

样条曲线在工程中应用不是很多，因此本节仅对 CRxDbSpline 类的构造函数进行封装。

2.6.3 步骤

（1）打开 CreateEnts 项目，添加一个函数 CreateEllipse，用于创建椭圆：

```
//创建椭圆
CRxDbObjectId CCreateEnts::CreateEllipse(CRxGePoint3d ptCenter,CRxGeVector3d vecNormal,
CRxGeVector3d majorAxis, double ratio)
{
    CRxDbEllipse *pEllipse = new CRxDbEllipse(ptCenter, vecNormal,majorAxis, ratio);
    return CCreateEnts::AddToModelSpace(pEllipse);
}
```

CRxDbEllipse 类的构造函数定义为：

```
CRxDbEllipse( const CRxGePoint3d& center,
const CRxGeVector3d& unitNormal,
const CRxGeVector3d& majorAxis,
double radiusRatio,
double startAngle = 0.0,
double endAngle = 6.28318530717958647692);
```

center 指定了椭圆的中心；unitNormal 输入了椭圆所在的平面；majorAxis 输入代表 1/2
长轴的矢量，也就是说该矢量的起点是椭圆的中心，终点是椭圆长轴的一个端点；radiusRatio 给出了椭圆短轴与长轴的长度比例；startAngle 为椭圆的起始角度（弧度）；endAngle 为椭圆的终止角度（弧度）。通过指定椭圆的起始角度和终止角度，能够方便地创建一个椭圆弧，如果忽略这两个参数，将会创建一个完整的椭圆。此外，要使用 CRxDbEllipse 类，必须在对应的文件中添加对 dbelipse.h 文件的包含。

（2）添加根据外接矩形创建椭圆的函数，输入参数为外接矩形的两个角点，其实现代码为：

```
//添加根据外接矩形创建椭圆的函数，输入参数为外接矩形的两个角点
        CRxDbObjectId CCreateEnts::CreateEllipse(CRxGePoint2d pt1, CRxGePoint2d pt2)
    {
        // 计算椭圆的中心点
        CRxGePoint3d ptCenter;
        ptCenter = CCalculation::MiddlePoint(CCalculation::Pt2dTo3d(pt1),
            CCalculation::Pt2dTo3d(pt2));
        CRxGeVector3d vecNormal(0, 0, 1);
        CRxGeVector3d majorAxis(fabs(pt1.x - pt2.x) / 2, 0, 0);
        double ratio = fabs((pt1.y - pt2.y) / (pt1.x - pt2.x));
        return CCreateEnts::CreateEllipse(ptCenter, vecNormal, majorAxis,
            ratio);
    }
```

根据外接矩形的两个角点首先可以计算出椭圆的中心点，然后计算椭圆的长轴端点和短、长轴长度比例，最后调用创建椭圆的函数。其中，fabs 是一个 C 语言的标准库函数，用于获得指定双精度浮点类型变量的绝对值，使用该函数必须添加对 math.h 的包含。

（3）添加创建样条曲线的函数，输入参数包括样条曲线的拟合点、拟合曲线的阶数和允许的拟合误差，其实现代码为：

```
// 声明部分
static CRxDbObjectId CreateSpline(const CRxGePoint3dArray& points,
int order = 4, double fitTolerance = 0.0);
// 实现部分
//添加创建样条曲线的函数，输入参数包括样条曲线的拟合点、拟合曲线的阶数和允许的拟合误差
// 实现部分
CRxDbObjectId CCreateEnts::CreateSpline(const CRxGePoint3dArray& points,int order, double
fitTolerance)
{
    assert (order >= 2 && order <= 26);
    CRxDbSpline *pSpline = new CRxDbSpline(points,order, fitTolerance);
    CRxDbObjectId splineId;
    splineId = CCreateEnts::AddToModelSpace(pSpline);
```

```
    return splineId;
}
```

上面的代码中，需要注意 assert 函数的使用。assert 函数用于判断一个变量或表达式的值是否为 true，如果为 false 则弹出一个错误对话框，并且终止程序的运行。assert 函数适用于判断函数输入参数的有效性，CRxDbSpline 类的构造函数要求 order 参数的值在 2～26 之间，因此可以使用 assert 函数来检查其有效性。

要使用 CRxDbSpline 类，必须在文件中包含 dbspline.h 头文件。

（4）添加用于创建样条曲线的函数。与前面的函数相比，多了两个参数，分别用于指定样条曲线起点和终点的切线方向（前面的函数会自动根据各个拟合点的位置计算出起点和终点的切线方向）。其实现代码为：

```
// 声明部分
CRxDbObjectId CCreateEnts::CreateSpline(const CRxGePoint3dArray& points,const CRxGeVector3d&
startTangent, const CRxGeVector3d& endTangent,int order=4, double fitTolerance=0.0);
// 实现部分
CRxDbObjectId CCreateEnts::CreateSpline(const CRxGePoint3dArray& points,const CRxGeVector3d&
startTangent, const CRxGeVector3d& endTangent,int order, double fitTolerance)
{
    assert(order >= 2 && order <= 26);
    CRxDbSpline *pSpline = new CRxDbSpline(points, startTangent,
        endTangent,
        order, fitTolerance);
    return CCreateEnts::AddToModelSpace(pSpline);
}
```

该函数同样适用断言函数 assert 对 order 参数的有效性进行判断，并且封装了 CRxDbSpline 类的一个构造函数。

如果你足够留心，在 ObjectCRX 的函数中能够发现许多形如“const CRxGeVector3d& startTangent”的参数，该参数的含义与“CRxGeVector3d startTangent”相同，两者都保证传递的实参不会被修改。那为什么还要采用这种形式？在形参中使用取地址运算符（&），实参在传递给形参时不会被复制，而仅仅被作为引用，但是单独使用该运算符，实参就有可能会在函数中被修改。加上了 const 关键字，则能保证实参不会被修改。如果同时使用取地址运算符和 const 关键字，就能在确保实参不会被修改的情况下减少系统的开销，因此 ObjectCRX 库函数中绝大部分的函数都采用了这种形式。

2.6.4 效果

（1）在项目中注册一个 ceEllipse 命令，用于测试创建椭圆的两个函数，其实现代码为：

```
//创建椭圆
void CRXCreateEllipse()
{
```

```
    // 使用中心点、所在平面、长轴矢量和短长轴比例来创建椭圆
    CRxGeVector3d vecNormal(0, 0, 1);
    CRxGeVector3d majorAxis(40, 0, 0);
    CRxDbObjectId entId;
    entId = CCreateEnts::CreateEllipse(CRxGePoint3d::kOrigin, vecNormal,
        majorAxis, 0.5);
    // 使用外接矩形来创建椭圆
    CRxGePoint2d pt1(60, 80), pt2(140, 120);
    CCreateEnts::CreateEllipse(pt1, pt2);
}
```

CRxGePoint3d::kOrigin 返回点（0，0，0）。

（2）注册一个 ceSpline 命令，用于测试创建样条曲线的两个函数，其实现代码为：

```
//创建样条线
void CRXCreateSpline()
{
    // 使用样本点直接创建样条曲线
    CRxGePoint3d pt1(0, 0, 0), pt2(10, 30, 0), pt3(60, 80, 0), pt4(100, 100,
        0);
    CRxGePoint3dArray points;
    points.append(pt1);
    points.append(pt2);
    points.append(pt3);
    points.append(pt4);
    CCreateEnts::CreateSpline(points);
    // 指定起始点和终止点的切线方向，创建样条曲线
    pt2.set(30, 10, 0);
    pt3.set(80, 60, 0);
    points.removeSubArray(0, 3);
    points.append(pt1);
    points.append(pt2);
    points.append(pt3);
    points.append(pt4);
    CRxGeVector3d startTangent(5, 1, 0);
    CRxGeVector3d endTangent(5, 1, 0);
    CCreateEnts::CreateSpline(points, startTangent, endTangent);
}
```

(3) 编译运行程序，执行 ceEllipse 和 ceSpline 命令，能够得到如图 2.13 所示的结果。

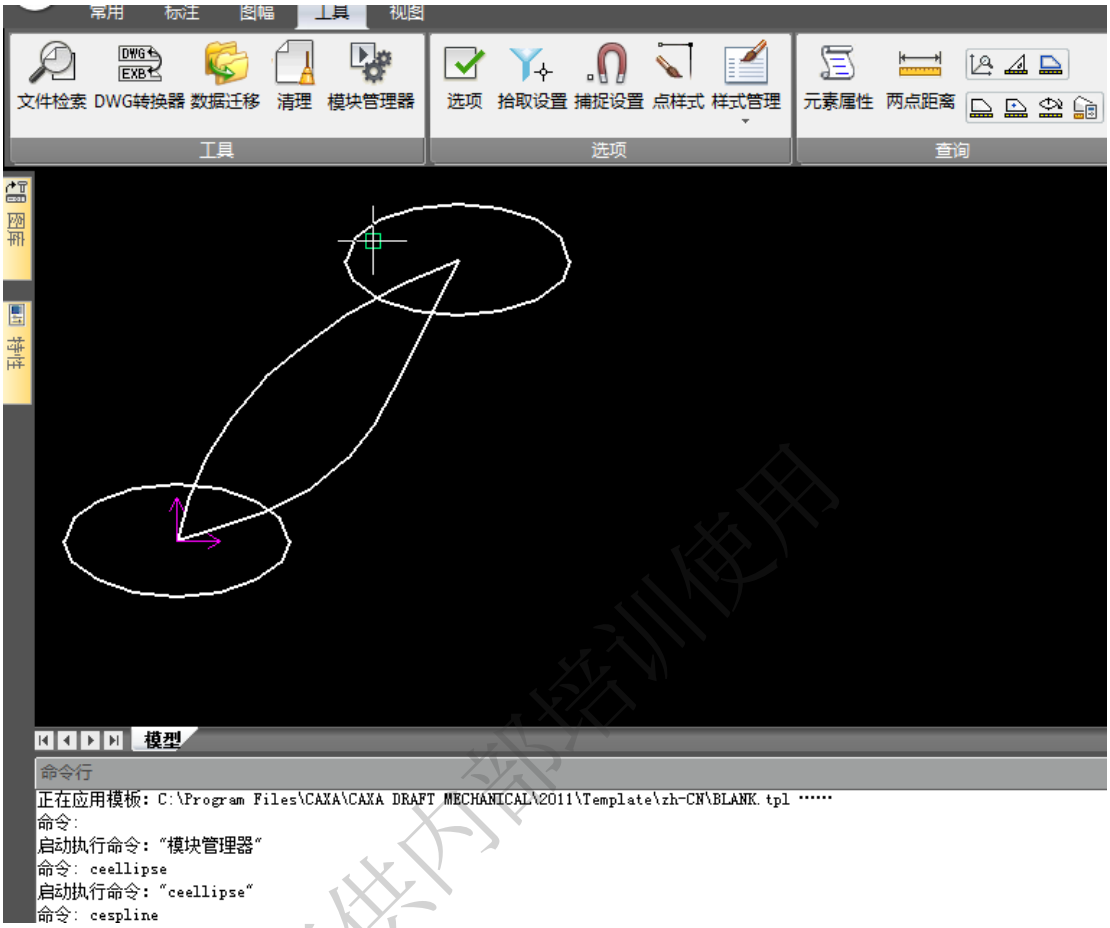


图2.13 创建椭圆和样条曲线

2.6.5 小结

学习本节内容之后，下面的几个知识点需要牢度掌握：

- 根据外接矩形创建椭圆的方法。
- 创建椭圆弧的方法。

2.7 创建文字

2.7.1 说明

本节实现了创建文字和多行文字的函数，分别对 CRxDbText 类和 CRxDbMText 类的相关函数进行封装。文字在 CAD 软件开发中涉及到较多的操作，在后面的章节中还会介绍。

2.7.2 思路

创建文字对象可以使用 CRxDbText 类的构造函数，其构造函数定义为：

```
CRxDbText( const CRxGePoint3d& position,
const CxCHAR* text,
CRxDbObjectId style = CRxDbObjectId::kNull,
double height = 0,
double rotation = 0);
```

position 指定文字的插入点；text 是将要创建的文字对象的内容；style 指定要使用的文字样式的 ID，默认情况下使用电子图板中缺省的文字样式；height 为文字的高度；rotation 为文字的旋转角度。CRxDbMText 类对应电子图板中的多行文字，该类的构造函数不接受任何参数，仅能创建一个空的多行文字对象。要创建多行文字，在调用构造函数之后，必须在将其添加到模型空间之前调用 setTextStyle 和 setContents 函数，也可同时调用其它函数来设置多行文字的特性。

2.7.3 步骤

（1）打开 CreateEnts 项目，添加一个函数 CreateText，用于创建文字对象：

```
// 声明部分
static CRxDbObjectId CreateText(const CRxGePoint3d& ptInsert,
const CxCHAR* text, CRxDbObjectId style = CRxDbObjectId::kNull,
double height = 2.5, double rotation = 0);
// 实现部分
//创建文字对象
CRxDbObjectId CCreateEnts::CreateText(const CRxGePoint3d& ptInsert,const CxCHAR* text,
CRxDbObjectId style,double height, double rotation)
{
    CRxDbText *pText = new CRxDbText(ptInsert, text, style, height,rotation);
    return CCreateEnts::AddToModelSpace(pText);
}
```

（2）添加一个 CreateMText 函数，用于添加多行文字：

```
// 声明部分
static CRxDbObjectId CreateMText(const CRxGePoint3d& ptInsert,
const CxCHAR* text, CRxDbObjectId style = CRxDbObjectId::kNull,
double height = 2.5, double width = 10);
// 实现部分
//添加多行文字
CRxDbObjectId CCreateEnts::CreateMText(const CRxGePoint3d& ptInsert,const CxCHAR* text,
CRxDbObjectId style,double height, double width)
{
```

```
    CRxDbMText *pMText = new CRxDbMText();
    // 设置多行文字的特性
    pMText->setTextStyle(style);
    pMText->setContents(text);
    pMText->setLocation(ptInsert);
    pMText->setTextHeight(height);
    pMText->setWidth(width);

    return CCreateEnts::AddToModelSpace(pMText);
}
```

上面的函数中，在创建多行文字对象之后，修改了它的文字样式、字符串内容、插入点、文字高度、宽度和文字的对齐方式。

要使用 CRxDbMText 类，必须在文件中包含 dbmtext.h 头文件。

2.7.4 效果

（1）注册一个 ceText 命令，用于测试本节创建的函数，其实现函数为：

```
//添加文字对象
void CRXCreateText()
{
    // 创建单行文字
    CRxGePoint3d ptInsert(0, 4, 0);
 CCreateEnts::CreateText(ptInsert, _T("北京数码大方科技有限公司
 （CAXA)")   ,CRxDbObjectId::kNull,2.5,0);
    // 创建多行文字
    ptInsert.set(0, 0, 0);
    CCreateEnts::CreateMText(ptInsert, _T("http://www.caxa.com"), CRxDbObjectId::kNull,2.5,10);
}
```

由于使用文字样式涉及到符号表的知识，本节不再介绍，且等第 4 章详细分解。

（2）编译运行程序，在电子图板 2011 中执行 ceText 命令，能够得到如图 2.14 所示的结果。

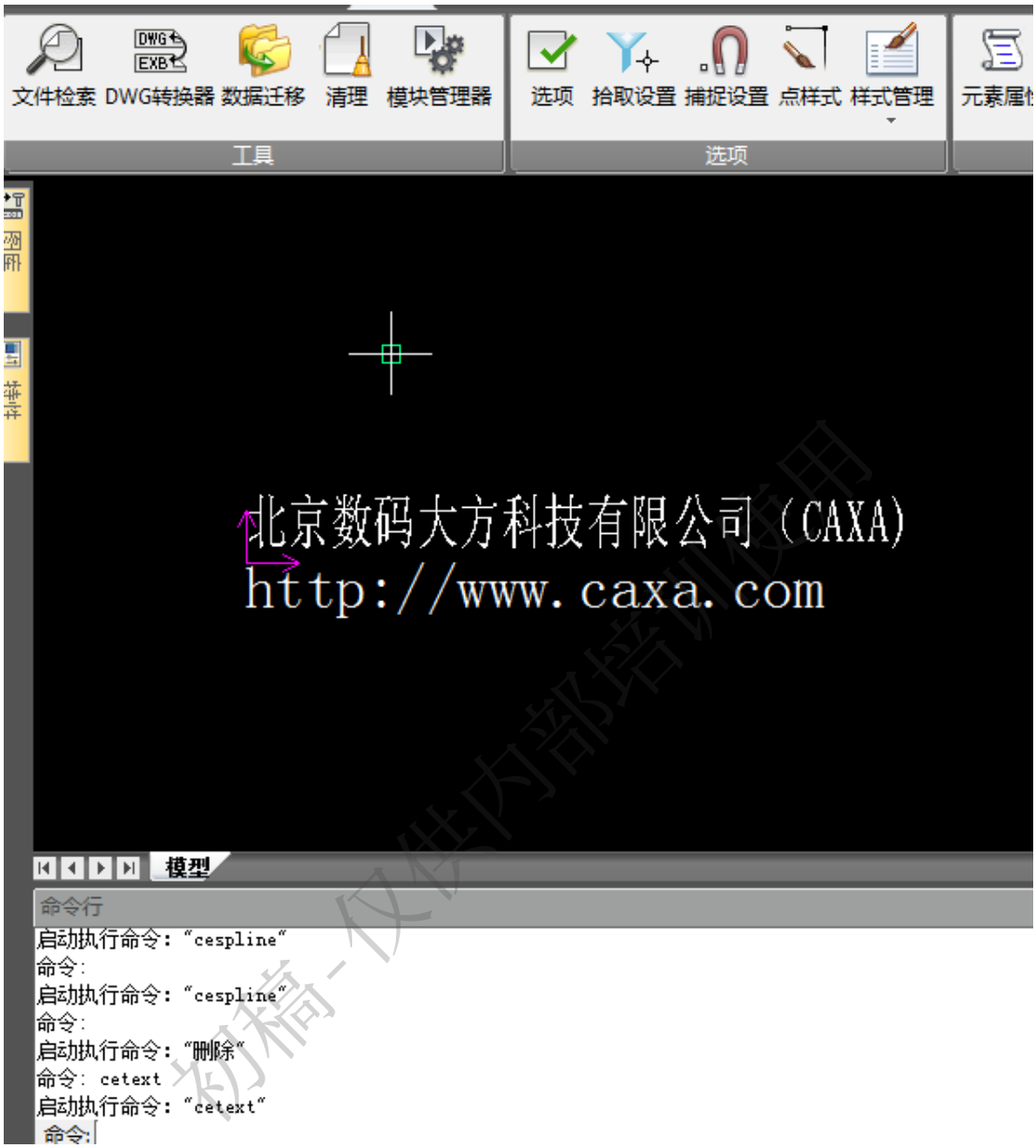


图2.14创建文字的结果

（3）设置字体样式。对于不熟悉电子图板操作的编程者，到这里可能会感到迷惑，为什么有时候汉字会被“？”代替了？因为当前使用的是西文字体，中文当然没法显示了。在电子图板 2011 中，在 ribbon bar 选择【标注 / 样式管理器/文字】菜单项，系统会弹出如图 2.15 所示的【文本风格设置】对话框。修改【中文字体】选项区的设置，选择一种中文字体，单击【确定】按钮，然后单击【关闭】按钮关闭该对话框。

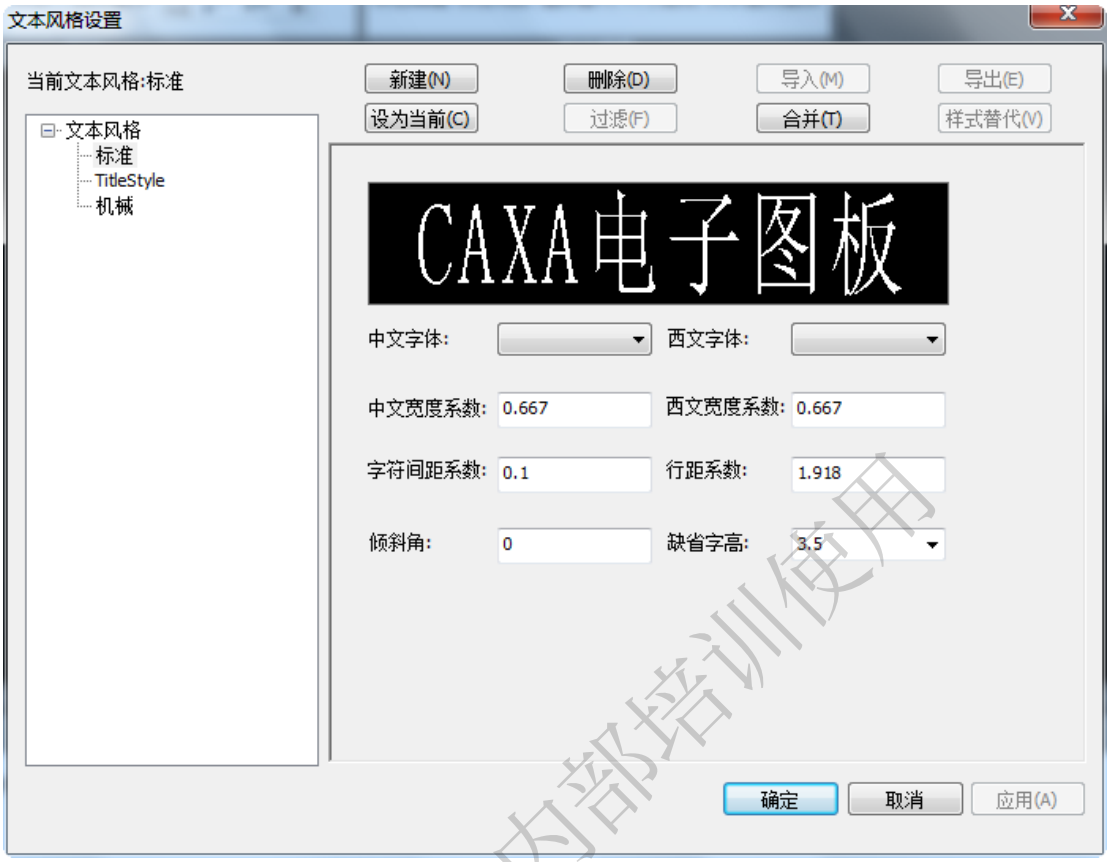


图2.15 设置文字样式

2.7.5 小结

学习本节内容之后，读者需要牢固掌握下面的知识点：

- 创建单行文字的方法。
- 创建多行文字的方法。

2.8 创建填充

2.8.1 说明

本节的实例能够提示用户选择所要填充的对象，然后根据用户选择的结果为该区域创建图案填充。

2.8.2 思路

CRxDbHatch 类代表电子图板中的填充对象，该类的构造函数仅创建一个空的填充对象，要想创建填充对象，必须按照下面的步骤进行：

（1）创建一个空的填充对象。

（2）指定填充对象所在的平面。

（3）设置填充对象的关联性；

（4）指定填充图案；

（5）添加填充边界；

（6）显示填充对象；

（7）将其添加到模型空间；

（8）如果是关联性的填充，将填充对象与边界绑定。

2.8.3 步骤

打开 CreateEnts 项目，添加一个 CreateHatch 函数，用于根据一组首尾连接的对象创建填充。该函数的输入参数为组成填充边界的对象 ID 数组、填充图案的名称和填充的关联性，其实现代码为：

```
//创建填充
CRxDbObjectId CCreateEnts::CreateHatch(CRxDbObjectIdArray objIds,const CxCHAR* patName, bool bAssociative)
{
    CDraft::ErrorStatus es;
    CRxDbHatch *pHatch = new CRxDbHatch();
     //设置填充平面
    CRxGeVector3d normal(0, 0, 1);

    pHatch->setNormal(normal);

    pHatch->setElevation(0);
    // 设置关联性
    pHatch->setAssociative(bAssociative);
    // 设置填充图案
    pHatch->setPattern(CRxDbHatch::kPreDefined, patName);
    // 添加填充边界
    es = pHatch->appendLoop(CRxDbHatch::kExternal, objIds);

     //添加到模型空间
    CRxDbObjectId hatchId;
    hatchId = CCreateEnts::AddToModelSpace(pHatch);
    // 如果是关联性的填充，将填充对象与边界绑定，以便使其能获得边界对象修改的通知
        if (bAssociative)
```

```
        {
            CRxDbEntity *pEnt;
            for (int i = 0; i < objIds.length(); i++)
            {
                es = crxdbOpenCRxDbEntity(pEnt, objIds[i],CRxDb::kForWrite);
                if (es == CDraft::eOk)
                {
                    //添加一个永久反应器
                    /*pEnt->addPersistentReactor(hatchId);*/
                    pEnt->close();
                }
            }
        }
        return hatchId;


}
```

创建图案填充时最关键的就是定义填充边界，在 ObjectCRX 使用 appendLoop 函数来向填充对象添加边界，该函数定义为：

```
CDraft::ErrorStatus appendLoop(
CAXA::Int32 loopType,
const CRxDbObjectIdArray& dbObjIds);
```

第一个参数指定了边界类型；第二个参数输入一组实体的 ID，用来定义填充边界。此外，还有两种重载形式，但是不常用。初学者容易忽略在创建填充之后调用 evaluateHatch 函数来显示填充。对于关联性填充，必须使用反应器来绑定填充和边界，这样边界发生变化的时候填充对象才能随之变化。关于反应器的问题，将在错误！未找到引用源。详细讨论，这里暂时不作介绍。

要使用 CRxDbHatch 类，必须在文件中包含 dbhatch.h 头文件。

2.8.4 效果

（1）在项目中注册一 ceHatch 命令，其实现函数为：

```
//添加填充
void CRXCreateHatch()
{
    // 提示用户选择填充边界
    crx_name ss;
    int rt = crxedSSGet(NULL, NULL, NULL, NULL, ss);
    CRxDbObjectIdArray objIds;
    // 初始化填充边界的ID数组
```

```
    if (rt == RTNORM)
    {
        long length;
        crxedSSLength(ss, &length);
        for (int i = 0; i < length; i++)
        {
            crx_name ent;
            crxedSSName(ss, i, ent);
            CRxDbObjectId objId;
            crxdbGetObjectId(objId, ent);
            objIds.append(objId);
        }
    }
    crxedSSFree(ss); // 释放选择集
    CCreateEnts::CreateHatch(objIds, "SOLID", true);
}
```

该函数中再次使用了选择集的操作，但是目前还不是最佳时机，暂且不需要了解它的详细使用。

（2）编译运行程序，在电子图板 2011 中，调用 LINE 命令创建一个三角形，如图 2.16 所示。

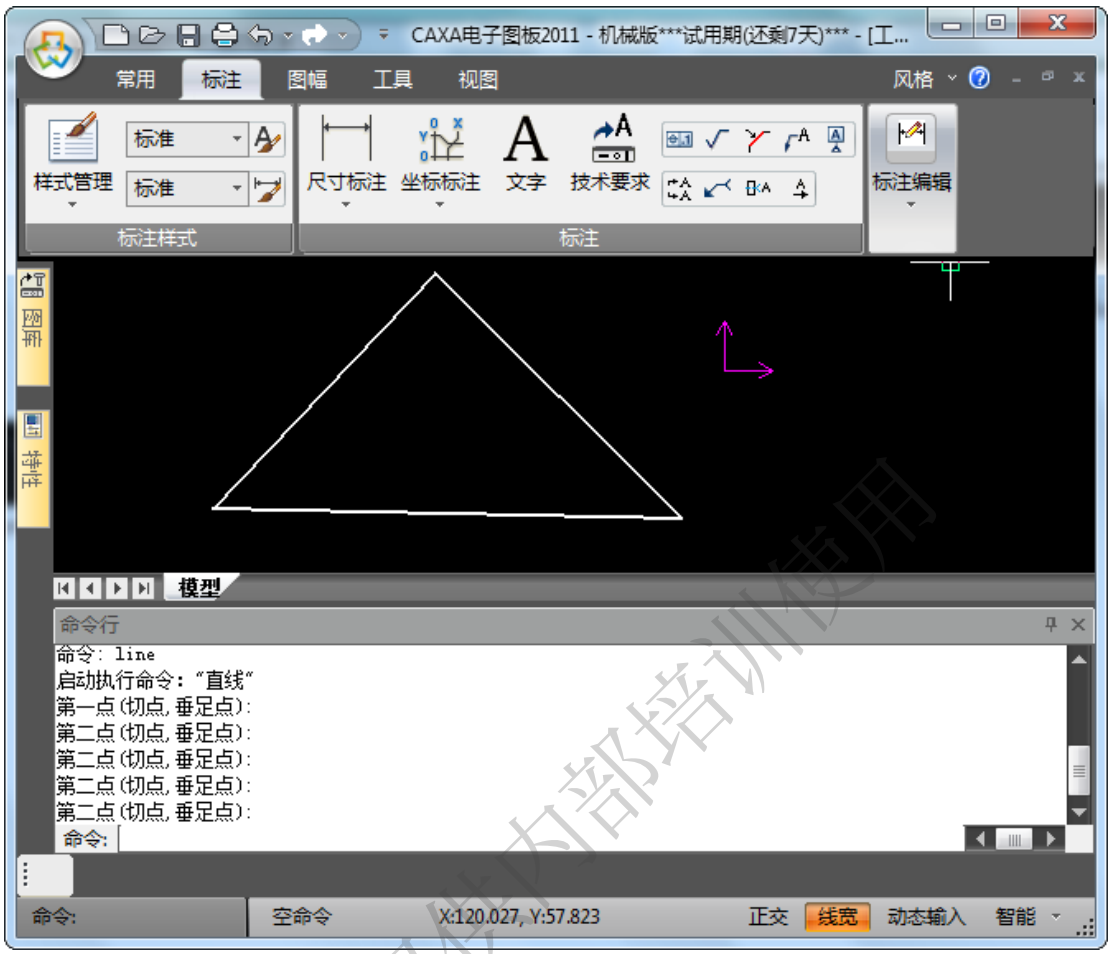


图2.16 创建三角形

（3）执行 ceHatch 命令，按照命令行提示进行操作：

命令：cehatch

选择对象：指定对角点：找到 3 个 〖选择组成三角形的三条直线〗

选择对象：〖按下Enter完成选择〗

完成操作后，得到如 图2.17所示的结果。

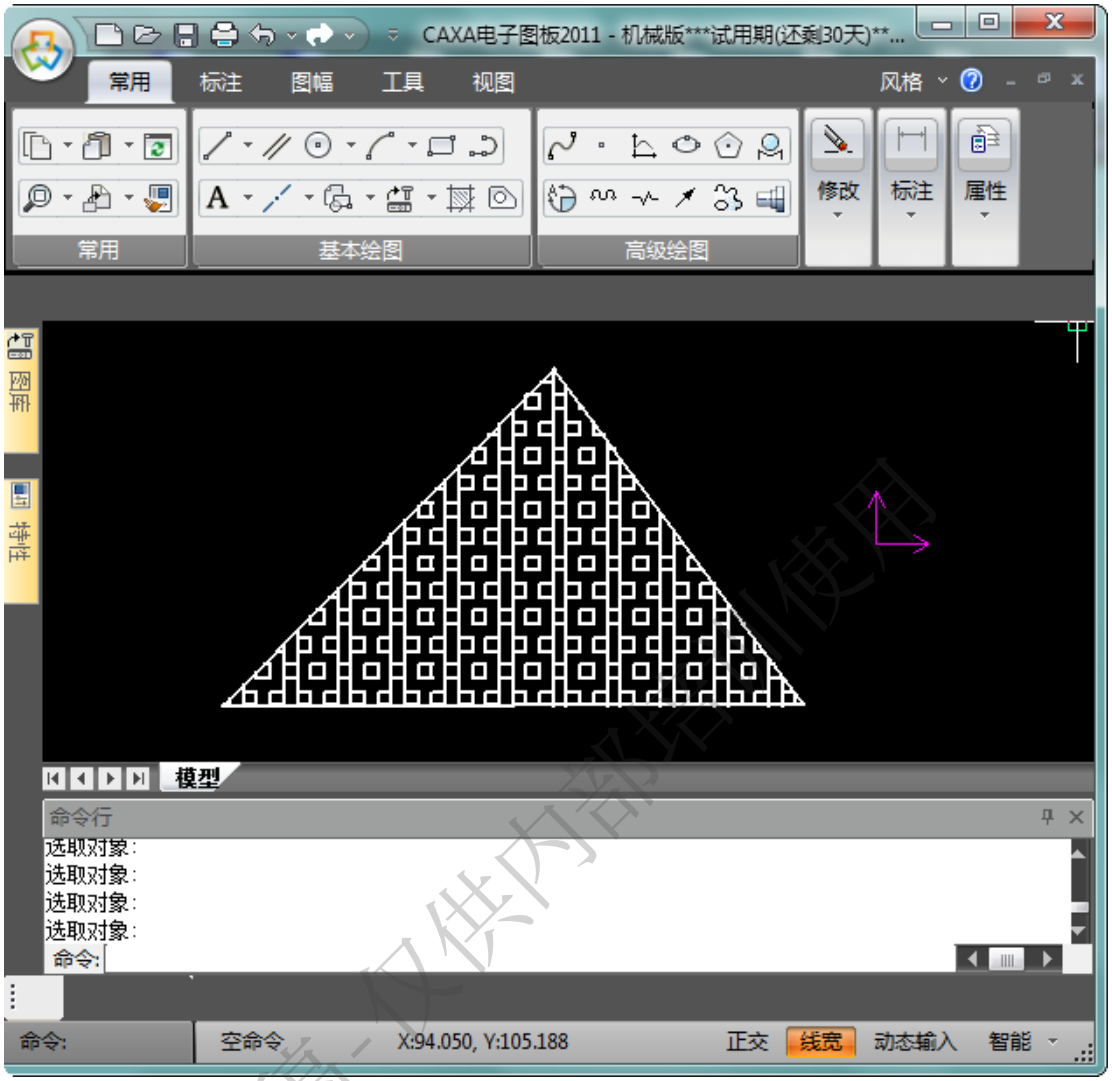


图2.17 创建填充的结果

2.8.5 小结

完成本节的学习，读者需要注意下面的知识点：

- 创建填充的一般步骤。
- 创建关联性和非关联性填充的区别。

2.9 创建尺寸标注

2.10.1 说明

本实例介绍了 12 个用于尺寸标注的函数，包括了转角标注、对齐标注、角度标注、半径标注、直径标注和坐标标注的创建函数。这些函数分别封装了系统提供的一些基本方法，并在实用性上进行了扩充。

2.10.2 思路

CRxDbAlignedDimension 类对应的是对齐标注，该类的构造函数接受 5 个参数：第一条尺寸边界线的起点、第二条尺寸边界线的起点、通过尺寸线的一点、标注文字和样式。本节创建一个函数，对该类的构造函数直接进行封装，另外创建一个函数，可以在创建标注时修改标注文字的位置。

设置尺寸文字的替代和位置修改，在实际工程中是很有必要的，对于错综复杂的图形，很多情况下，长度可能会用字母或者其他的汉字替代，而多个按照标准方式放置的标注文字很可能发生重叠，这时候这两个预留的参数就能给你带来极大的方便。

CRxDbRotatedDimension 类对应转角标注，该类的构造函数接受 6 个参数：标注的旋转角度、第一条尺寸边界线的起点、第二条尺寸边界线的起点、通过尺寸线的一点、标注文字和样式。本节创建的函数直接对该函数进行封装。

CRxDbRadialDimension 类对应半径标注，该类的构造函数需要输入标注曲线的中心点、引线附着的坐标、引线长度、标注文字和样式。本节除了对构造函数进行封装之外，提供了根据圆心、半径、标注尺寸线旋转角度和引线长度来创建半径标注的函数，其关键点就在于根据已知的参数计算出构造函数需要的参数。

CRxDbDiametricDimension 类对应直径标注，其构造函数需要输入标注直径的两个端点、引线长度、标注文字和样式。本节除对其构造函数直接封装之外，还提供了根据圆弧的圆心、半径、引线放置角度和引线长度创建标注的方法。

ObjectCRX 提供了两各类来对应角度标注，分别是：CRxDb2LineAngularDimension 类和 CRxDb3PointAngularDimension 。本节分别对这两个类的构造函数进行了封装。CRxDbOrdinateDimension 类对应坐标标注，其构造函数需要输入是否是 X 轴标注（布尔类型变量）、标注箭头的起始位置、标注箭头的终止位置、标注文字和样式。本节对该函数封装之后，又创建了两个新函数，能够同时创建 X、Y 两个坐标值，并且根据相对坐标修改引线端点的位置。

2.10.3 步骤

（1）打开 CreateEnts 项目，在 CreateEnt 类中添加一个新函数 CreateDimAligned，用于创建对齐标注：

```
// 声明部分

static CRxDbObjectId CreateDimAligned(const CRxGePoint3d& pt1,
const CRxGePoint3d& pt2, const CRxGePoint3d& ptLine,
const CxCHAR* dimText = NULL,
```

```
CRxDbObjectId dimStyle = CRxDbObjectId::kNull);
// 实现部分
//创建对齐标注
CRxDbObjectId CCreateEnts::CreateDimAligned(const CRxGePoint3d& pt1,const CRxGePoint3d& pt2,
const CRxGePoint3d& ptLine,const CxCHAR* dimText,     CRxDbObjectId dimStyle)
{
    CRxDbAlignedDimension *pDim = new CRxDbAlignedDimension(pt1, pt2,ptLine, dimText,
dimStyle);
    return CCreateEnts::AddToModelSpace(pDim);
}
```

要使用标注的相关类，必须在文件中包含 dbdim.h 头文件。

（2）创建一个重载的函数，允许用户输入 vecOffset 作为标注文字位置的偏移量，其实现代码为：

```
// 声明部分
static CRxDbObjectId CreateDimAligned(const CRxGePoint3d& pt1,
const CRxGePoint3d& pt2, const CRxGePoint3d& ptLine,
const CRxGeVector3d& vecOffset = CRxGeVector3d::kIdentity,
const CxCHAR* dimText = NULL);
// 实现部分
//标注文字位置的偏移量
// 实现部分
CRxDbObjectId CCreateEnts::CreateDimAligned(const CRxGePoint3d& pt1,const CRxGePoint3d& pt2,
const CRxGePoint3d& ptLine,const CRxGeVector3d& vecOffset,const CxCHAR* dimText)
{
    CRxDbAlignedDimension *pDim = new CRxDbAlignedDimension(pt1, pt2,ptLine, dimText,
CRxDbObjectId::kNull);
    CRxDbObjectId dimensionId;
    dimensionId = CCreateEnts::AddToModelSpace(pDim);
    // 打开已经创建的标注，对文字的位置进行修改
    CRxDbEntity *pEnt;
    CDraft::ErrorStatus es;
    es = crxdbOpenCRxDbEntity(pEnt, dimensionId, CRxDb::kForWrite);
        CRxDbAlignedDimension *pDimension =CRxDbAlignedDimension::cast(pEnt);
    if (pDimension != NULL)
    {
        // 移动文字位置前，需先指定尺寸线的变化情况（这里指定为：
        // 尺寸线不动，在文字和尺寸线之间加箭头）
        pDimension->setDimtmove(1);
        // 根据偏移向量修正文字插入点的位置
        CRxGePoint3d ptText = pDimension->textPosition();
```

```
            ptText = ptText + vecOffset;
            pDimension->setTextPosition(ptText);
        }
        pEnt->close();
        return dimensionId;
}
```

注意，移动标注文字必须在将其添加到模型空间之后进行。

（3）创建一个新函数 CreateDimRotated，用于添加转角标注：

```
// 声明部分
static CRxDbObjectId CreateDimRotated(const CRxGePoint3d& pt1,
const CRxGePoint3d& pt2, const CRxGePoint3d& ptLine,
double rotation, const CxCHAR* dimText = NULL,
CRxDbObjectId dimStyle = CRxDbObjectId::kNull);
// 实现部分
//创建一个新函数CreateDimRotated，用于添加转角标注：
// 实现部分
CRxDbObjectId CCreateEnts::CreateDimRotated(const CRxGePoint3d& pt1,const CRxGePoint3d& pt2,
const CRxGePoint3d& ptLine,    double rotation, const CxCHAR* dimText,CRxDbObjectId dimStyle)
{
        CRxDbRotatedDimension *pDim = new
            CRxDbRotatedDimension(rotation,pt1, pt2, ptLine, dimText, dimStyle);
        return CCreateEnts::AddToModelSpace(pDim);
}
```

（4）创建一个新函数CreateDimRadial，用于创建半径标注：

```
// 声明部分
static CRxDbObjectId CreateDimRadial(const CRxGePoint3d& ptCenter,
const CRxGePoint3d& ptChord, double leaderLength,
const CxCHAR* dimText = NULL,
CRxDbObjectId dimStyle = CRxDbObjectId::kNull);
// 实现部分
//创建一个新函数CreateDimRadial，用于创建半径标注：
// 实现部分
CRxDbObjectId CCreateEnts::CreateDimRadial(const CRxGePoint3d& ptCenter,  const CRxGePoint3d&
ptChord, double leaderLength,    const CxCHAR* dimText,  CRxDbObjectId dimStyle)
{
        CRxDbRadialDimension *pDim = new CRxDbRadialDimension(ptCenter,ptChord, leaderLength,
dimText, dimStyle);
        return CCreateEnts::AddToModelSpace(pDim);
}
```

（5）创建一个重载的函数，用于根据圆心、半径、标注尺寸线的旋转角度和引线长度来创建半径标注：

```
// 声明部分
static CRxDbObjectId CreateDimRadial(const CRxGePoint3d& ptCenter,
double radius, double angle, double leaderLength = 5);
// 实现部分
//根据圆心、半径、标注尺寸线的旋转角度和引线长度来创建半径标注：
// 实现部分
CRxDbObjectId CCreateEnts::CreateDimRadial(const CRxGePoint3d& ptCenter,  double radius, double angle, double leaderLength)
{
    CRxGePoint3d ptChord = CCalculation::PolarPoint(ptCenter, angle,    radius);
    return CCreateEnts::CreateDimRadial(ptCenter, ptChord,leaderLength);
}
```

其中，CCalculation::PolarPoint 是一个自定义函数，能够根据相对极坐标来确定一个点的位置：

```
CRxGePoint3d CCalculation::PolarPoint(const CRxGePoint3d& pt, double angle,
double distance)
{
ads_point ptForm, ptTo;
ptForm[X] = pt.x;
ptForm[Y] = pt.y;
ptForm[Z] = pt.z;
```



```
acutPolar(ptForm, angle, distance, ptTo);
return asPnt3d(ptTo);
}
```

（6）创建一个函数 CreateDimDiametric，用于创建直径标注：

```
// 声明部分
static CRxDbObjectId CreateDimDiametric(const CRxGePoint3d& ptChord1,
const CRxGePoint3d& ptChord2, double leaderLength,
const CxCHAR* dimText = NULL,
CRxDbObjectId dimStyle = CRxDbObjectId::kNull);
// 实现部分
//用于创建直径标注：
// 实现部分
CRxDbObjectId CCreateEnts::CreateDimDiametric(const CRxGePoint3d& ptChord1,const CRxGePoint3d& ptChord2, double leaderLength,const CxCHAR* dimText, CRxDbObjectId dimStyle)
```

```
{
    CRxDbDiametricDimension *pDim = new CRxDbDiametricDimension(ptChord1,ptChord2,
leaderLength, dimText, dimStyle);
    return CCreateEnts::AddToModelSpace(pDim);
}
```

（7）创建一个重载的函数，根据圆心、半径、标注尺寸线的旋转角度和引线长度来创建直径标注：

```
// 声明部分
static CRxDbObjectId CreateDimDiametric(const CRxGePoint3d& ptCenter,
double radius, double angle, double leaderLength = 5);
// 实现部分
//根据圆心、半径、标注尺寸线的旋转角度和引线长度来创建直径标注：
// 实现部分
CRxDbObjectId CCreateEnts::CreateDimDiametric(const CRxGePoint3d& ptCenter,double radius, double
angle, double leaderLength)
{
    // 计算标注通过点的位置
    CRxGePoint3d ptChord1, ptChord2;
    ptChord1 = CCalculation::PolarPoint(ptCenter, angle, radius);
    ptChord2 = CCalculation::PolarPoint(ptCenter,angle + CCalculation::PI(), radius);
    return CCreateEnts::CreateDimDiametric(ptChord1, ptChord2,leaderLength);
}
```

（8）创建一个函数，用于根据两条直线的关系来创建角度标注：

```
// 声明部分
static CRxDbObjectId CreateDim2LineAngular(const CRxGePoint3d& ptStart1,
const CRxGePoint3d& ptEnd1, const CRxGePoint3d& ptStart2,
const CRxGePoint3d& ptEnd2, const CRxGePoint3d& ptArc,
const CxCHAR* dimText = NULL,
CRxDbObjectId dimStyle = CRxDbObjectId::kNull);
// 实现部分
//根据两条直线的关系来创建角度标注：
// 实现部分
CRxDbObjectId CCreateEnts::CreateDim2LineAngular(const CRxGePoint3d& ptStart1,
    const CRxGePoint3d& ptEnd1, const CRxGePoint3d& ptStart2,
    const CRxGePoint3d& ptEnd2, const CRxGePoint3d& ptArc,
    const CxCHAR* dimText, CRxDbObjectId dimStyle)
{
    CRxDb2LineAngularDimension *pDim = new
```

```
            CRxDb2LineAngularDimension(
            ptStart1, ptEnd1, ptStart2, ptEnd2, ptArc, dimText, dimStyle);
    return CCreateEnts::AddToModelSpace(pDim);
}
```

（9）创建一个重载的函数，可以根据顶点、起始点、终止点和标注尺寸线通过点来创建角度标注：

```
// 声明部分
static CRxDbObjectId CreateDim3PtAngular(const CRxGePoint3d& ptCenter,
const CRxGePoint3d& ptEnd1, const CRxGePoint3d& ptEnd2,
const CRxGePoint3d& ptArc,
const CxCHAR* dimText = NULL,
CRxDbObjectId dimStyle = CRxDbObjectId::kNull);
// 实现部分
//根据顶点、起始点、终止点和标注尺寸线通过点来创建角度标注：
// 实现部分
CRxDbObjectId CCreateEnts::CreateDim3PtAngular(const CRxGePoint3d& ptCenter,
    const CRxGePoint3d& ptEnd1, const CRxGePoint3d& ptEnd2,
    const CRxGePoint3d& ptArc, const CxCHAR* dimText,
    CRxDbObjectId dimStyle)
{
    CRxDb3PointAngularDimension *pDim = new
        CRxDb3PointAngularDimension(
        ptCenter, ptEnd1, ptEnd2, ptArc, dimText, dimStyle);
    return CCreateEnts::AddToModelSpace(pDim);
}
```

（10）创建一个函数 CreateDimOrdinate，用于创建坐标标注，直接封装了类的构造函数，其实现代码为：

```
// 声明部分
static CRxDbObjectId CreateDimOrdinate(CAXA::Boolean xAxis,
const CRxGePoint3d& ptStart, const CRxGePoint3d& ptEnd,
const CxCHAR* dimText = NULL,
CRxDbObjectId dimStyle = CRxDbObjectId::kNull);
// 实现部分
AcDbObjectId CCreateEnts::CreateDimOrdinate(CAXA::Boolean xAxis,
const AcGePoint3d& ptStart, const AcGePoint3d& ptEnd,
const CxCHAR* dimText,
AcDbObjectId dimStyle)
{
AcDbOrdinateDimension *pDim = new
```

```
AcDbOrdinateDimension(xAxis,
ptStart, ptEnd, dimText, dimStyle);
return CCreateEnts::AddToModelSpace(pDim);
} //创建坐标标注，直接封装了类的构造函数，其实现代码为：
// 实现部分
CRxDbObjectId CCreateEnts::CreateDimOrdinate(Adesk::Boolean xAxis,const CRxGePoint3d& ptStart,
const CRxGePoint3d& ptEnd,const CxCHAR* dimText,CRxDbObjectId dimStyle)
{
    CRxDbOrdinateDimension *pDim = new CRxDbOrdinateDimension(xAxis,ptStart, ptEnd, dimText,
dimStyle);
    return CCreateEnts::AddToModelSpace(pDim);
}
```

（11）创建一个重载的函数，能够同时创建一个点的 X、Y 坐标标注：

```
// 声明部分
static CRxDbObjectIdArray CreateDimOrdinate(const CRxGePoint3d& ptDef,
const CRxGePoint3d& ptTextX, const CRxGePoint3d& ptTextY);
// 实现部分
//创建一个点的X、Y 坐标标注：
// 实现部分
CRxDbObjectIdArray CCreateEnts::CreateDimOrdinate(const CRxGePoint3d& ptDef,
    const CRxGePoint3d& ptTextX, const CRxGePoint3d& ptTextY)
{
    CRxDbObjectId dimId;
    CRxDbObjectIdArray dimIds;
    dimId = CCreateEnts::CreateDimOrdinate(CAXA::kTrue, ptDef,ptTextX);
    dimIds.append(dimId);
    dimId = CCreateEnts::CreateDimOrdinate(CAXA::kFalse, ptDef,ptTextY);
    dimIds.append(dimId);
    return dimIds;
}
```

（12）创建一个重载的函数，能够根据点的偏移位置来创建坐标标注：

```
// 声明部分
static CRxDbObjectIdArray CreateDimOrdinate(const CRxGePoint3d& ptDef,
const CRxGeVector3d& vecOffsetX, const CRxGeVector3d&
vecOffsetY);
// 实现部分
//根据点的偏移位置来创建坐标标注：
// 实现部分
```

```
CRxDbObjectIdArray CCreateEnts::CreateDimOrdinate(const CRxGePoint3d& ptDef,  const
CRxGeVector3d& vecOffsetX, const CRxGeVector3d&      vecOffsetY)
{
    CRxGePoint3d ptTextX = ptDef + vecOffsetX;
    CRxGePoint3d ptTextY = ptDef + vecOffsetY;
    return CCreateEnts::CreateDimOrdinate(ptDef, ptTextX, ptTextY);
}
```

2.10.4 说明

（1）在 CCalculation 类中增加一个函数 RelativePoint，用于根据相对直角坐标来计算一个点的位置：

```
CRxGePoint3d CCalculation::RelativePoint(const CRxGePoint3d& pt,
double x, double y)
{
CRxGePoint3d ptReturn(pt.x + x, pt.y + y, pt.z);
return ptReturn;
}
```

（2）在项目中注册一个命令ceDimension，用于测试本节创建的函数：

```
//添加标注
void CRXCreateDimension()
{
    // 指定起始点位置
    CRxGePoint3d pt1(200, 160, 0);
    CRxGePoint3d pt2= CCalculation::RelativePoint(pt1, -40, 0);
    CRxGePoint3d pt3 = CCalculation::PolarPoint(pt2,      7 * CCalculation::PI() / 6, 20);
    CRxGePoint3d pt4 = CCalculation::RelativePoint(pt3, 6, -10);
    CRxGePoint3d pt5 = CCalculation::RelativePoint(pt1, 0, -20);
    // 绘制外轮廓线
    CCreateEnts::CreateLine(pt1, pt2);
    CCreateEnts::CreateLine(pt2, pt3);
    CCreateEnts::CreateLine(pt3, pt4);
    CCreateEnts::CreateLine(pt4, pt5);
    CCreateEnts::CreateLine(pt5, pt1);
    // 绘制圆形
    CRxGePoint3d ptCenter1, ptCenter2;
    ptCenter1 = CCalculation::RelativePoint(pt3, 16, 0);
    ptCenter2 = CCalculation::RelativePoint(ptCenter1, 25, 0);
```

```
    CCreateEnts::CreateCircle(ptCenter1, 3);
    CCreateEnts::CreateCircle(ptCenter2, 4);
    CRxGePoint3d ptTemp1, ptTemp2;
    // 水平标注
    ptTemp1 = CCalculation::RelativePoint(pt1, -20, 3);
    CCreateEnts::CreateDimRotated(pt1, pt2, ptTemp1, 0);
    // 垂直标注
    ptTemp1 = CCalculation::RelativePoint(pt1, 4, 10);
    CCreateEnts::CreateDimRotated(pt1, pt5, ptTemp1,     CCalculation::PI() / 2);
    // 转角标注
    ptTemp1 = CCalculation::RelativePoint(pt3, -3, -6);
    CCreateEnts::CreateDimRotated(pt3, pt4, ptTemp1,     7 * CCalculation::PI() / 4);
    // 对齐标注
    ptTemp1 = CCalculation::RelativePoint(pt2, -3, 4);
    CCreateEnts::CreateDimAligned(pt2, pt3, ptTemp1,CRxGeVector3d(4, 10, 0), _T("new position"));
    // 角度标注
    ptTemp1 = CCalculation::RelativePoint(pt5, -5, 5);
    CCreateEnts::CreateDim3PtAngular(pt5, pt1, pt4, ptTemp1);
    // 半径标注
    ptTemp1 = CCalculation::PolarPoint(ptCenter1,   CCalculation::PI() / 4, 3);
    CCreateEnts::CreateDimRadial(ptCenter1, ptTemp1, -3);


        // 直径标注
        ptTemp1 = CCalculation::PolarPoint(ptCenter2,   CCalculation::PI() / 4, 4);
    ptTemp2 = CCalculation::PolarPoint(ptCenter2,   CCalculation::PI() / 4, -4);
    CCreateEnts::CreateDimDiametric(ptTemp1, ptTemp2, 0);
    // 坐标标注
    CCreateEnts::CreateDimOrdinate(ptCenter2, CRxGeVector3d(0, -10, 0),CRxGeVector3d(10, 0, 0));
}
```

（3）编译运行程序，在电子图板 2011 中执行 ceDimension 命令，能够得到如图 2.18 所示的结果。

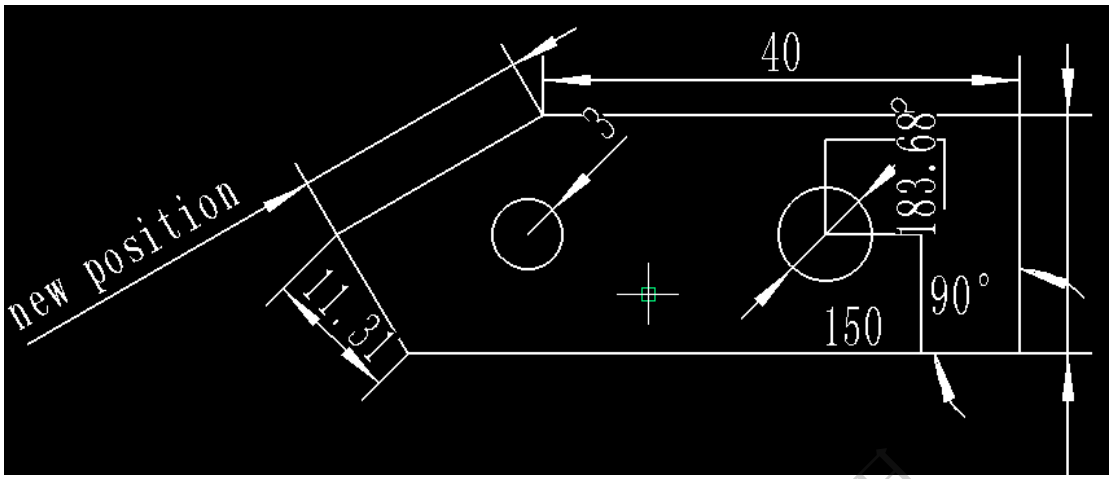


图2.18 程序运行结果

2.10.5 小结

学习本节内容之后，读者需要掌握下面的知识点：

□ 尺寸文本的替代：所有的函数都可直接指定 dimText 参数实现尺寸文本的替代。在创建标注之后，还可以使用 CRxDbDimension::setDimensionText 函数来设置尺寸文本的内容。

□ 尺寸文本的移动：使用 setTextPosition 函数可以实现尺寸文本的移动，在移动文字之前，最好使用 setDimtmove 函数设置文字和尺寸线移动时的关系，例如设置在移动文字时，不移动尺寸线，但是在两者之间加引线。

□ 尺寸标注的关联性：无法直接使用 CRxDbDimension 及其派生类创建关联性的尺寸标注，要实现尺寸标注的关联性，必须使用反应器。

2.10 动态拖动

2.11.1 说明

本节所介绍的实例，允许用户创建一个圆，在创建过程中允许用户拖动鼠标指定圆半径，并在屏幕预显，其使用方法与电子图板中提供的圆操作完全一致。

2.11.2 思路

CRxEdJig 类是 ObjectCRX 提供的动态拖画的基类，要实现动态拖画效果，可以从该类派生自己的拖画类。

CRxEdJig 类有三个重要函数：

```
virtual DragStatus   sampler();
virtual CAXA::Boolean   update();
virtual CRxDbEntity* entity() const;
```

当鼠标在屏幕上滑动时，每移动一次，sampler 和 update 都会被调用，sampler 用于在拖动过程中获得鼠标的当前信息，如当前位置等，update 用于根据鼠标当前位置更新拖动对象的参数，entity 用于返回拖动的对象。通常这三个函数需要被派生类重载。

2.11.3 步骤

1. 启动 Visual Studio 2010，使用 ObjectCRX 向导创建一个新工程，其名称为 CircleJig。在项目上右击选择【Add\Class\C++ Class】, Base class 填写 CRxEdJig

2. CRxEdJig 类声明中

```
class CircleJig : public CRxEdJig
{
/*public:
    CRX_DECLARE_DYNCREATE(CircleJig) */;

public:
    CircleJig(const CRxGePoint3d&);
    void doIt();  /*该函数创建一个圆对象，然后使用jig填充*/
    virtual DragStatus sampler();  /* 该函数有拖动函数调用，以获取一个样本输入*/
    virtual CAXA::Boolean update();
    virtual CRxDbEntity* entity() const;

private:
    CRxDbCircle *mpCircle;
    CRxGePoint3d mCenterPt;
    double mRadius;
};
```

3. 自定义类的函数实现如下：

```
CircleJig::CircleJig(
    const CRxGePoint3d& pt)
    : mCenterPt(pt),
    mRadius(0.5)
{

}
```

```
void CircleJig::doIt()
{
    //初始化拖画圆弧，并设置必要的参数
    mpCircle = new CRxDbCircle();
    mpCircle->setCenter(mCenterPt);
    mpCircle->setRadius(mRadius);

    //开始拖画
    CRxEdJig::DragStatus stat = drag();

    //将圆弧添加进数据库
    append();
}


CRxEdJig::DragStatus CircleJig::sampler()
{
    DragStatus stat;

    setUserInputControls((UserInputControls)
        (CRxEdJig::kAccept3dCoordinates
        | CRxEdJig::kNoNegativeResponseAccepted
        | CRxEdJig::kNoZeroResponseAccepted));

    //获得拖画圆弧半径
    stat = acquireDist( mRadius );


    return stat;
}

CAXA::Boolean CircleJig::update()
{
    //更新拖画圆弧半径
    mpCircle->setRadius(mRadius);

    return CAXA::kTrue;
}
```

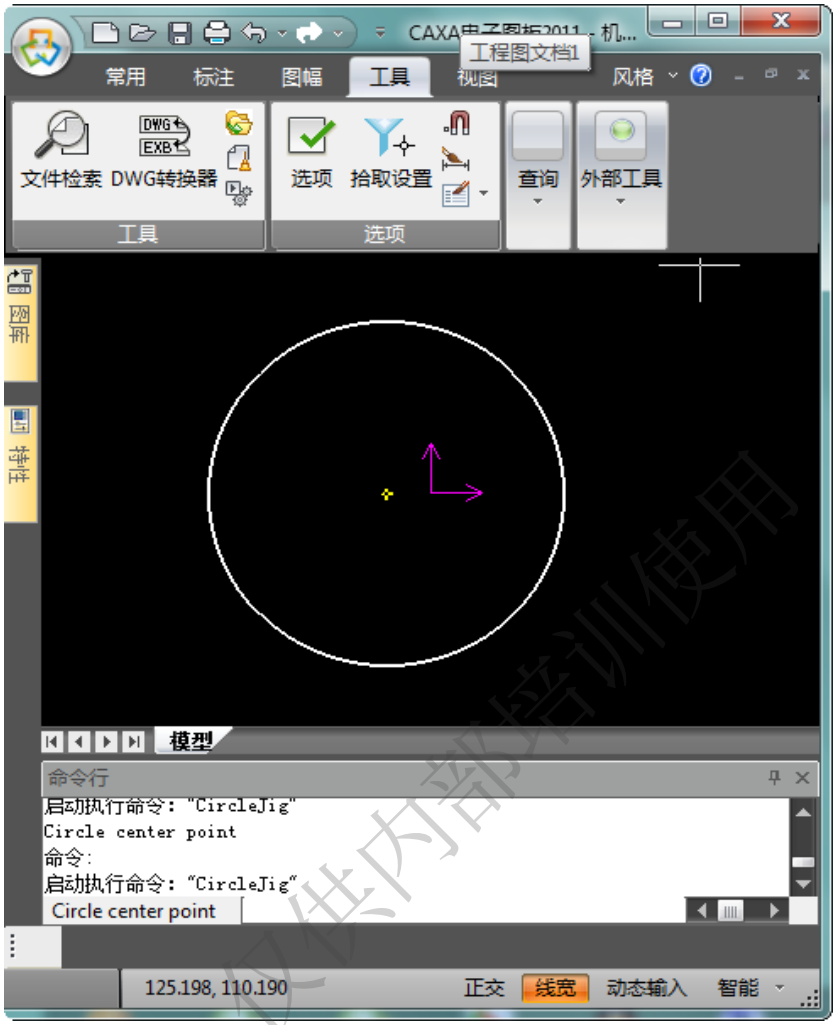
```
CRxDbEntity* CircleJig::entity() const
{
    //返回拖画圆弧
    return mpCircle;
}
```

4. 注册一个命令 CircleJig，用自定义类创建一个自定义实体。

```
void createCircle()
{
    CRxGePoint3d tempPt;

    //获得圆心
    if( RTNORM != acedGetPoint(NULL, _T("\n输入圆心点"),asDblArray(tempPt)) )
    {
        assert(0);
        return;
    }

    //创建动态拖画圆对象
    CircleJig *pJig   = new CircleJig(tempPt);

    //开始动态拖画
    pJig->doIt();

    //删除动态拖画对象
    delete pJig;
}
```


2.11.4 效果

（1）编译链接程序，启动电子图板 2011，加载生成的 CRX 文件。创建的自定义圆的时候，就会出现拖曳效果。生成的圆如下图：



9.2.5 小结

学习本节内容之后，读者应该掌握下面的知识点：

□CRxEdJig类派生。

2.11.5 小结

学习本章知识之后，读者需要掌握下面的知识点：

□ 提示用户选择一个实体的方法。

□ 创建和编辑对象时使用 CRxEdJig 实现拖动效果。

2.11　获得某一图层上所有的直线

2.11.1 说明

本节介绍的实例，能将当前图形中“测试”图层上所有直线对象的颜色变为红色。该实例演示了块表记录遍历器的使用，为获取图形中某一类具有相同特征的实体提供了一种方法。

2.11.2 思路

ObjectCRX 提供了一种称为遍历器的类，用来遍历（逐个访问）某一集合中所有的对象，譬如，遍历当前图形中所有的图层、实体等。

下面的代码显示了块表记录遍历器的使用要点：

```
// 创建块表记录遍历器
CRxDbBlockTableRecordIterator *pItr; // 块表记录遍历器
pBlkTblRcd->newIterator(pItr);
CRxDbEntity *pEnt; // 遍历的临时实体指针
for (pItr->start(); !pItr->done(); pItr->step())
{
// 利用遍历器获得每一个实体
    pItr->getEntity(pEnt, CRxDb::kForWrite);
   // 对pEnt所指向的实体进行各种编辑
   ……
   // 注意需要关闭实体
    pEnt->close();
   }
delete pItr; // 遍历器使用完毕之后一定要删除！
```

块表记录遍历器的使用非常简单，简单的说就是三个步骤：创建遍历器；使用遍历器遍历实体；删除遍历器。

2.11.3 步骤

在 Visual Studio 2010 中，使用 ObjectCRX 向导创建一个名为 GetEntsOnLayer 的项目，注册一个名为 GetEnts 的命令，其实现函数为：

```
void CRXGetEnts()
{
    // 判断是否存在名称为“测试”的图层
    CRxDbLayerTable *pLayerTbl;
    crxdbHostApplicationServices()->workingDatabase()
```

```
        ->getSymbolTable(pLayerTbl, CRxDb::kForRead);
    if (!pLayerTbl->has(_T("测试")))
    {
        crxutPrintf(_T("\n当前图形中未包含\"测试\"图层!"));
        pLayerTbl->close();
        return;
    }
    CRxDbObjectId layerId; // “测试”图层的ID
    pLayerTbl->getAt(_T("测试"), layerId);
    pLayerTbl->close();
    // 获得当前数据库的块表
    CRxDbBlockTable *pBlkTbl;
    crxdbHostApplicationServices()->workingDatabase()
        ->getBlockTable(pBlkTbl, CRxDb::kForRead);
    // 获得模型空间的块表记录
    CRxDbBlockTableRecord *pBlkTblRcd;
    pBlkTbl->getAt(CRXDB_MODEL_SPACE, pBlkTblRcd,
        CRxDb::kForRead);
    pBlkTbl->close();
    // 创建块表记录遍历器
    CRxDbBlockTableRecordIterator *pItr; // 块表记录遍历器

    pBlkTblRcd->newIterator(pItr,true,true);
    CRxDbEntity *pEnt; // 遍历的临时实体指针

        for ( ; !pItr->done(); pItr->step())
        {
            // 利用遍历器获得每一个实体
            pItr->getEntity(pEnt, CRxDb::kForWrite);
            // 是否在“测试”图层上
            if (pEnt->layerId() == layerId)
            {
                // 是否是直线
                CRxDbLine *pLine = CRxDbLine::cast(pEnt);
                if (pLine != NULL)
                {
                    pLine->setColorIndex(1); // 将直线的颜色修改为红色
                }
            }
            // 注意需要关闭实体
```

```
            pEnt->close();
        }
        delete pItr; // 遍历器使用完毕之后一定要删除！
        pBlkTblRcd->close();
}
```

在获得指定名称的层表记录之前，要判断当前图形中是否包含指定的图层，可以使用 CRxDbLayerTable::has 函数来实现。如果块表中包含与指定名称相同的层表记录，该函数返回 true，否则返回 false。

使用遍历器时必须注意，遍历一个集合对象要使用其对应的遍历器，例如本节的实例遍历块表记录就是用了块表记录遍历器。声明块表记录遍历器指针之后，还要使用 newIterator 函数创建当前图形模型空间块表记录的遍历器。遍历器在使用完毕后一定要删除，否则就会引起电子图板的错误退出。

判断实体是否在指定的图层上，可以使用两种方法：

□ 使用 CRxDbEntity 类的 layer 函数获得实体所在图层的名称，然后与指定图层的名称进行比较，例如（acutDelString 函数需要添加对 acutmem.h 头文件的包含）：

```
char *layerName = pEnt->layer();
if (strcmp(layerName, "测试") == 0)
{
// 执行需要的操作
……
}
acutDelString(layerName); // 释放layer函数返回的字符串所占用的内存
```

□ 使用 CRxDbEntity 类的 layerId 函数获得实体所在图层的 ID，然后与指定图层的 ID 进行比较。本节的实例中使用的就是这样方法。

2.11.4 效果

编译运行程序，在 电子图板 2011 中创建几个新图层，其中包含一个名为“测试”的图层，然后创建若干个图形对象，将其分别放置在不同的图层中。

执行 cegetents 命令，能够发现，“测试”图层中所有的直线变为红色，与程序设计的初衷完全一致。

2.11.5 小结

学习本节实例之后，读者可以在此基础上进行扩展：

□ 获得当前图形中所有半径小于 5 的圆。

□ 获得当前图形中所有颜色为红色的实体。

□ 获得当前图形中位于“测试”图层上的实体的集合。

本节的实例仅讨论了如何对满足条件的实体进行编辑，但是并未介绍如何获得满足特

定条件的实体的集合，实际上实现这一点已经非常简单了。一般来说，用 CRxDbObjectIdArray 类作为对象的集合比较合适，可以在使用遍历器之前声明一个 CRxDbObjectIdArray 类的对象：

```
CRxDbObjectIdArray entIds;
```

而在判断实体满足一定条件之后，可以将实体的 ID 添加到数组中：

```
entIds.append(pEnt->objectId());
```

这样，在删除遍历器之后，就得到了包含所有满足条件的实体的 ID 数组。

2.12 利用 Transform 实现复制、移动等操作

2.12.1 说明

创建 ObjectCRX 的应用程序经常要和用户进行交互，很可能会对实体执行一些移动、旋转、镜像等操作，这种情况下可以直接使用 crxedCommand 函数调用相关的电子图板内部命令，也可以使用 CRxDbEntity 类的 transformBy 函数，对实体进行相应的变换操作。

实际上使用第一种方法在某些特定的情况下有一个致命的问题。由于 电子图板内部命令和 ObjectCRX 注册的命令在不同的线程中执行，因此有可能导致使用 crxedCommand 函数的命令执行过程发生错乱，下面可以用一个例子来证明。

使用 ObjectCRX 向导创建一个新工程，其名称设置为 TransformEnt,然后注册一个名称为 TestCommand 的新命令，该命令的实现代码为：

```
void CRXTransformTestCommand()
{
crxedCommand(RTSTR, _T("_move"),RTNONE);
    AfxMessageBox(_T("先移动实体还是先弹出对话框?"));
}
```

在电子图板 2011 中执行此命令，你会发现在进行移动操作之前，对话框就已经弹出来了，如果在一个线程内部决不会发生，因此可以证明电子图板内部命令和 ObjectCRX 注册的命令不在同一个线程内部执行。基于这一点，对实体进行变换操作最好的方法还是使用 transformBy 函数。

> 注意：在 **ObjectCRX** 编程中尽量不要使用中文的标点符号，特别是不能在 **crxutPrintf**、**crxedGetPoint** 等向命令窗口输出文本的函数中使用，某些情况下使用中文标点符号会导致命令窗口中本应显示的中文提示变成乱码。

本节的程序介绍使用 transformBy 函数对实体进行几何变换，其中的一些方法使用 ObjectCRX 中提供的一些基于 COM（组件对象模型）的全局函数来实现。

2.12.2 思路

1．使用 transformBy 函数进行几何变换

transformBy 是 CRxDbEntity 类的一个成员函数，该函数使用一个 CRxGeMatrix3d 参数对实体进行相应的几何变换，其原型为：

```
virtual CDraft::ErrorStatus transformBy(const CRxGeMatrix3d& xform);
```

所有 CRxDbEntity 的派生类都实现了这个虚函数，因此所有的实体都可以使用这种方法进行几何变换。CRxGeMatrix3d 是一个几何类，用于表示一个四维矩阵，其基本形式如图 2.22 所示。

$$\begin{bmatrix} c_{00} & c_{01} & c_{02} & t_0 \\ c_{10} & c_{11} & c_{12} & t_1 \\ c_{20} & c_{21} & c_{22} & t_2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

图2.22 变换矩阵的内容

尽管可以像普通的矩阵那样修改变换矩阵中的元素，但是多种变换叠加的矩阵参数计算起来非常麻烦，所幸 CRxGeMatrix3d 提供了一些很有用的成员函数：

- setToTranslation：生成一个移动对象的矩阵。
- setToRotation：生成旋转矩阵。
- setToScaling：生成比例缩放矩阵。
- setToMirroring：生成镜像矩阵。

2．复制实体

CRxDbObject 类拥有一个 clone 函数，能否生成一个调用者的克隆对象，并返回指向克隆对象的指针。由于所有实体对应的类都间接继承于 CRxDbObject 类（CRxDbEntity 类从 CRxDbObject 类继承），因此所有实体都可以用这种方法进行克隆。

clone 函数仅仅会生成对象的一个克隆，对于实体对象来说，这样还没有完成复制操作的全部。在创建实体时们已经了解到，创建实体仅仅是第一个步骤，还必须把它添加到模型空间中才能被显示出来，对于克隆得到的实体同样需要这样做。

2.12.3 步骤

（1）使用 ObjectCRX 向导创建一个新工程，命名为 TransformEnt。然后在 Visual Studio 2010 中，在 solution explorer 窗口，右键先建立的项目选择【Add/Class】菜单项，在系统弹出的【New Class】对话框中，选择【C++ Class】，输入 CTransUtil 作为类的名称，单击【OK】按钮完成类的创建。

（2）在 CTransUtil 类中添加一个函数 Move，使用 transformBy 函数对实体进行移动：

```
//实体进行移动
CDraft::ErrorStatus CTransUtil::Move(CRxDbObjectId entId, const CRxGePoint3d &ptFrom, const
CRxGePoint3d &ptTo)
{
    // 构建用于实现移动实体的矩阵
```

```
    CRxGeVector3d vec(ptTo[X] - ptFrom[X], ptTo[Y] - ptFrom[X],
        ptTo[Z] - ptFrom[Z]);
    CRxGeMatrix3d mat;
    mat.setToTranslation(vec);
    CRxDbEntity *pEnt = NULL;
    CDraft::ErrorStatus es = crxdbOpenObject(pEnt, entId,CRxDb::kForWrite);
    if (es != CDraft::eOk)
        return es;
    es = pEnt->transformBy(mat);
    pEnt->close();
    return es;
}
```

使用 transformBy 函数移动实体的关键在于构建符合要求的 CRxGeMatrix3d 对象，CRxGeMatrix3d 类的 setToTranslation 函数用于完成这个功能，它所接受的参数是一个三维矢量，因此先根据移动的基点和目的点构建了一个 CRxGeVector3d 对象。

（3）添加一个函数 Copy，能够生成给定 ID 的实体的一个克隆，并且按照 ptFrom 和 ptTo 生成的矢量移动生成的克隆对象，其实现代码为：

```
//能够生成给定ID 的实体的一个克隆，并且按照ptFrom 和ptTo生成的矢量移动生成的克隆对象，其实现代码为：
BOOL CTransUtil::Copy(CRxDbObjectId entId, const CRxGePoint3d &ptFrom,const CRxGePoint3d &ptTo)
{
    CRxDbEntity *pEnt = NULL;
    if (acdbOpenObject(pEnt, entId, CRxDb::kForRead) != CDraft::eOk)
        return FALSE;
    CRxDbEntity *pCopyEnt = CRxDbEntity::cast(pEnt->clone());
    CRxDbObjectId copyEntId;
    if (pCopyEnt)
        copyEntId = AddToModelSpace(pCopyEnt);
      Move(copyEntId, ptFrom, ptTo);
    return TRUE;
}
```

clone 函数能够生成调用者的一个克隆，该函数是 CRxDbObject 类的一个成员函数，其原型为：

```
virtual CRxRxObject* clone() const;
```

由于函数的返回值不是 CRxDbEntity 类型的指针，因此必须通过一个很常用的方法进行实体指针的升级：

```
CRxDbEntity *pCopyEnt = CRxDbEntity::cast(pEnt->clone());
```

Copy 函数中，AddToModelSpace 函数用于将实体添加到模型空间，同样使用 static 关

键字来限制其名称的可见性：

```
static CRxDbObjectId AddToModelSpace(CRxDbEntity* pEnt)
    {
        CRxDbBlockTable *pBlockTable;
        crxdbHostApplicationServices()->workingDatabase()
            ->getBlockTable(pBlockTable, CRxDb::kForRead);
        CRxDbBlockTableRecord *pBlockTableRecord;
        pBlockTable->getAt(ACDB_MODEL_SPACE, pBlockTableRecord,
            CRxDb::kForWrite);
        CRxDbObjectId entId;
        pBlockTableRecord->appendAcDbEntity(entId, pEnt);
        pBlockTable->close();
        pBlockTableRecord->close();
        pEnt->close();
        return entId;
    }
```

（4）添加一个函数

```
CDraft::ErrorStatus CTransUtil::Rotate(CRxDbObjectId entId,const CRxGePoint2d &ptBase,double angle)
    {
        // 构建变换矩阵
        CRxGeMatrix3d mat;
        mat.setToRotation(angle, CRxGeVector3d::kZAxis,
            CRxGePoint3d(ptBase[X], ptBase[Y], 0.));
        // 对实体进行变换
        CRxDbEntity *pEnt = NULL;
        CDraft::ErrorStatus es = acdbOpenObject(pEnt, entId,
            CRxDb::kForWrite);
        if (es != CDraft::eOk)
            return es;
        es = pEnt->transformBy(mat);
        pEnt->close();
            return es;
    }
```

2.12.4 效果

2.12.5 小结

学习本节实例之后，读者可以在此基础上进行扩展：

- 掌握使用 transformBy 函数进行几何变换。
- 掌握复制实体方法。