

2020/07/13_MFC_第11课_CAD06_序列化与反序列化

笔记本: MFC

创建时间: 2020/7/13 星期一 15:36

作者: ileemi

- [序列化与反序列化 \(Serialize and UnSerialize\)](#)
- [C++序列化](#)
- [C++反序列化](#)
 - [C++序列化反序列化注意点](#)
- [MFC 序列化](#)

序列化与反序列化 (Serialize and UnSerialize)

类似游戏存档

存档 (序列化) -- 把内存中的对象存储到文件

读档 (反序列化) -- 把文件中的对象读到内存

C++ 序列化

不能直接将对象存储到文件内, 如果对象内部存在指针, 存储的时候只是将指针的值存储到文件内, 等到反序列化的时候, 原来对象内部指针指向的内存上存储的数据就不能被正常读取。

示例:


```

111 // 存储需啊存储的类对象个数
112 int nObjectCount = 5;
113 file.write((char*)&nObjectCount, sizeof(nObjectCount));
114 // 存储对象
115 stu1.Serialize(file);
116 stu2.Serialize(file);
117 stu3.Serialize(file);
118 stu4.Serialize(file);
119 stu5.Serialize(file);
120
121
122 // 反序列化
123 cout << "数据写入成功" << endl;
124 file.close(); // 关闭文件

```

C++ 反序列化

读取文件数据时，申请存储对应数据的对象应该放置再堆上（使用栈存储会引发一系列问题）。同时存储数据到文件，也有可能是别人申请的，文件内的对象数量并不明确，对于应该申请多少对象是个未知，所以，再存储的时候，应该存储要存储对象的个数。

代码示例：

```

// 类内序列化
void Serialize(fstream& file) { ... }

// 类内反序列化
void UnSerialize(fstream& file)
{
    // 读取ID
    file.read((char*)&m_nID, sizeof(m_nID));

    // 读取姓名的长度
    int nNameSize = 0;
    file.read((char*)&nNameSize, sizeof(nNameSize));
    // 根据姓名长度申请对应大小的空间
    char* szName = new char[nNameSize];
    file.read(szName, nNameSize);
    m_strName = szName;
    delete[] szName; // 释放空间
}

private:
    int m_nID; // ID
    string m_strName; // 姓名

```

```

// 反序列化
// 读取存储的对象个数
int nObjectCount = 0;
file.read((char*)&nObjectCount, sizeof(nObjectCount));

// 根据存储的对象个数申请对应存储类对象的对象个数
CStudent* pStus = new CStudent[nObjectCount];
for (int i = 0; i < nObjectCount; i++)
{
    pStus[i].UnSerialize(file); // 调用反序列化
}

file.close(); // 关闭文件
return 0;

```

名称	值	类型
pStus[0]	(m_nID=1 m_strName="xiaobai")	CStudent
pStus[1]	(m_nID=2 m_strName="xiaohong")	CStudent
pStus[2]	(m_nID=3 m_strName="xiaozhang")	CStudent
pStus[3]	(m_nID=4 m_strName="xiaoli")	CStudent
pStus[4]	(m_nID=5 m_strName="xiaohai")	CStudent
nObjectCount	5	int

C++ 序列化反序列化注意点

- 对象的类提供序列化和反序列化的成员
- 对象的个数

- 对象的类型信息要保存

MFC 序列化

MFC 提供了一套序列化的机制

API -- CArchive

MSDN Library - October 2001

文件(F) 编辑(E) 查看(V) 转到(G) 帮助(H)

隐藏 查找 上一步 下一步 上一步 前进 停止 刷新 主页 打印

活动子集(S)

目录(D) 索引(I) 搜索(S) 收藏夹(I)

键入关键字进行查找(F):

CArchive

- CArchive
- class members
- construction/destruction
- data members
- support for persistent data using with CFile
- CArchive basic input/output
- CArchive class
- CArchive data members
- CArchive methods
- CArchive object
- closing
- creating
- definition
- explicit creation
- implicit creation
- loading (example)
- storing (example)
- CArchive object input/output
- CArchive operators
- using
- vs. Serialize function
- CArchive overview
- CArchive status
- CArchive:
- bNoFlushOnDelete
- load
- store
- CArchiveException
- class members
- construction/destruction
- data members
- enumerators
- CArchiveException class
- CArchiveException data members

显示(O)

CArchive

CArchive does not have a base class.

The **CArchive** class allows you to save a complex network of objects in a permanent binary form (usually disk storage) that persists after those objects are deleted. Later you can load the objects from persistent storage, reconstituting them in memory. This process of making data persistent is called "serialization."

You can think of an archive object as a kind of binary stream. Like an input/output stream, an archive is associated with a file and permits the buffered writing and reading of data to and from storage. An input/output stream processes sequences of ASCII characters, but an archive processes binary object data in an efficient, nonredundant format.

You must create a **CFile** object before you can create a **CArchive** object. In addition, you must ensure that the archive's load/store status is compatible with the file's open mode. You are limited to one active archive per file.

When you construct a **CArchive** object, you attach it to an object of class **CFile** (or a derived class) that represents an open file. You also specify whether the archive will be used for loading or storing. A **CArchive** object can process not only primitive types but also objects of **CObject**-derived classes designed for serialization. A serializable class usually has a **Serialize** member function, and it usually uses the **DECLARE_SERIAL** and **IMPLEMENT_SERIAL** macros, as described under class **CObject**.

The overloaded extraction (>>) and insertion (<<) operators are convenient archive programming interfaces that support both primitive types and **CObject**-derived classes.

CArchive also supports programming with the MFC Windows Sockets classes **CSocket** and **CSocketFile**. The **IsBufferEmpty** member function supports that usage.

For more information on **CArchive**, see the articles [Serialization \(Object Persistence\)](#) and [Windows Sockets: Using Sockets with Archives in Visual C++ Programmer's Guide](#).

#include <afx.h>

Class Members | Hierarchy Chart

Sample [MFC Sample MULTIPAD](#)

See Also [CFile](#), [CObject](#), [CSocket](#), [CSocketFile](#)

[Send feedback to MSDN.](#) [Look here for MSDN Online resources.](#)

MSDN Library - October 2001

文件(F) 编辑(E) 查看(V) 转到(G) 帮助(H)

隐藏 查找 上一步 下一步 上一步 前进 停止 刷新 主页 打印

活动子集(S)

目录(D) 索引(I) 搜索(S) 收藏夹(I)

键入关键字进行查找(F):

CArchive

- CArchive
- class members
- construction/destruction
- data members
- support for persistent data using with CFile
- CArchive basic input/output
- CArchive class
- CArchive data members
- CArchive methods
- CArchive object
- closing
- creating
- definition
- explicit creation
- implicit creation
- loading (example)
- storing (example)
- CArchive object input/output
- CArchive operators
- using
- vs. Serialize function
- CArchive overview
- CArchive status
- CArchive:
- bNoFlushOnDelete
- load
- store
- CArchiveException
- class members
- construction/destruction
- data members
- enumerators
- CArchiveException class
- CArchiveException data members

显示(O)

Serialization (Object Persistence)

[Home](#) | [Overview](#) | [How Do I](#) | [Tutorial](#)

This article explains the serialization mechanism provided in the Microsoft Foundation Class Library (MFC) to allow objects to persist between runs of your program.

"Serialization" is the process of writing or reading an object to or from a persistent storage medium, such as a disk file. MFC supplies built-in support for serialization in the class **CObject**. Thus, all classes derived from **CObject** can take advantage of **CObject**'s serialization protocol.

The basic idea of serialization is that an object should be able to write its current state, usually indicated by the value of its member variables, to persistent storage. Later, the object can be re-created by reading, or deserializing, the object's state from the storage. Serialization handles all the details of object pointers and circular references to objects that are used when you serialize an object. A key point is that the object itself is responsible for reading and writing its own state. Thus, for a class to be serializable, it must implement the basic serialization operations. As shown in the Serialization group of articles, it is easy to add this functionality to a class.

MFC uses an object of the **CArchive** class as an intermediary between the object to be serialized and the storage medium. This object is always associated with a **CFile** object, from which it obtains the necessary information for serialization, including the file name and whether the requested operation is a read or write. The object that performs a serialization operation can use the **CArchive** object without regard to the nature of the storage medium.

A **CArchive** object uses overloaded insertion (<<) and extraction (>>) operators to perform writing and reading operations. For more information, see [Storing and Loading CObjects via an Archive](#) in the article Serialization: Serializing an Object.

Note Do not confuse the **CArchive** class with general-purpose I/O stream classes, which are for formatted text only. The **CArchive** class is for binary-format serialized objects.

The following articles cover the two main tasks required for serialization:

- [Serialization: Making a Serializable Class](#)
- [Serialization: Serializing an Object](#)

The article [Serialization: Serialization vs. Database Input/Output](#) is part of the group of articles on database topics. The article describes when serialization is an appropriate input/output technique in database applications.

[Send feedback to MSDN.](#) [Look here for MSDN Online resources.](#)

MSDN Library - October 2001

文件(F) 编辑(E) 查看(V) 转到(G) 帮助(H)

隐藏 查找 上一步 下一步 上一步 前进 停止 刷新 主页 打印

活动子集(S)

目录(D) 索引(I) 搜索(S) 收藏夹(I)

键入关键字进行查找(F):

CArchive

- CArchive
- class members
- construction/destruction
- data members
- support for persistent data using with CFile
- CArchive basic input/output
- CArchive class
- CArchive data members
- CArchive methods
- CArchive object
- closing
- creating
- definition
- explicit creation
- implicit creation
- loading (example)
- storing (example)
- CArchive object input/output
- CArchive operators
- using
- vs. Serialize function
- CArchive overview
- CArchive status
- CArchive:
- bNoFlushOnDelete
- load
- store
- CArchiveException
- class members
- construction/destruction
- data members
- enumerators
- CArchiveException class
- CArchiveException data members

显示(O)

Serialization: Making a Serializable Class

[Home](#) | [Overview](#) | [How Do I](#) | [Tutorial](#)

Five main steps are required to make a class serializable. They are listed below and explained in the following sections:

- [Deriving your class from CObject](#) (or from some class derived from **CObject**).
- [Overriding the Serialize member function.](#)
- [Using the DECLARE_SERIAL macro](#) in the class declaration.
- [Defining a constructor that takes no arguments.](#)
- [Using the IMPLEMENT_SERIAL macro in the implementation file](#) for your class.

If you call **Serialize** directly rather than through the >> and << operators of [CArchive](#), the last three steps are not required for serialization.

Deriving Your Class from CObject

The basic serialization protocol and functionality are defined in the **CObject** class. By deriving your class from **CObject** (or from a class derived from **CObject**), as shown in the following declaration of class **CPerson**, you gain access to the serialization protocol and functionality of **CObject**.

Overriding the Serialize Member Function

The **Serialize** member function, which is defined in the **CObject** class, is responsible for actually serializing the data necessary to capture an object's current state. The **Serialize** function has a **CArchive** argument that it uses to read and write the object data. The **CArchive** object has a member function, **IsStoring**, which indicates whether **Serialize** is storing (writing data) or loading (reading data). Using the results of **IsStoring** as a guide, you either insert your object's data in the **CArchive** object with the insertion operator (<<) or extract data with the extraction operator (>>).

Consider a class that is derived from **CObject** and has two new member variables, of types **CString** and **WORD**. The following class declaration fragment shows the new member variables and the declaration for the overridden **Serialize** member function:

```
class CPerson : public CObject
{
public:
    DECLARE_SERIAL(CPerson)
    // empty constructor is necessary
    CPerson();

    CString m_name;
    WORD m_number;
```


使用前，需要改造自己的类，改造步骤：

1. 继承 CObject
2. 重写（覆盖）Serialize 成员函数
3. 在类声明中使用宏 DECLARE_SERIAL
4. 定义一个没有参数的构造（无参构造）
5. 在类的实现文件中使用宏 IMPLEMENT_SERIAL

Using the IMPLEMENT_SERIAL Macro in the Implementation File

The **IMPLEMENT_SERIAL** macro is used to define the various functions needed when you derive a serializable class from **CObject**. You use this macro in the implementation file (.CPP) for your class. The first two arguments to the macro are the name of the class and the name of its immediate base class.

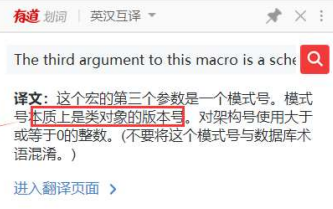
The third argument to this macro is a schema number. The schema number is essentially a version number for objects of the class. Use an integer greater than or equal to 0 for the schema number. (Don't confuse this schema number with database terminology.)

The MFC serialization code checks the schema number when reading objects into memory. If the schema number of the class in memory does not match the schema number of the class in the file, the library will throw a **CArchiveException** from reading an incorrect version of the object.

If you want your **Serialize** member function to be able to read multiple versions — the application — you can use the value **VERSIONABLE_SCHEMA** as an argument to the **IMPLEMENT_SERIAL** macro. For more information and an example, see the **GetObjectSchema** member function of class **CArchive**.

The following example shows how to use **IMPLEMENT_SERIAL** for a class, **CPerson**, to derive from **CObject**.

```
IMPLEMENT_SERIAL(CPerson, CObject, 1)
```



在控制台中使用MFC的库：

新建 "Windows桌面程序" --> 勾选"MFC标头"



MFC 文件类 -- **CFile**

CArchive -- 专门用于序列化和非序列化

CArchive提供了多种类型的运算符重载，同时支持 read 和 write 函数：

CArchive Class Members

Data Members

[m_pDocument](#) Points to the **CDocument** object being serialized.

Construction

[CArchive](#) Creates a **CArchive** object.

[Abort](#) Closes an archive without throwing an exception.

[Close](#) Flushes unwritten data and disconnects from the **CFile**.

Basic Input/Output

[Flush](#) Flushes unwritten data from the archive buffer.

[operator >>](#) Loads objects and primitive types from the archive.

[operator <<](#) Stores objects and primitive types to the archive.

[Read](#) Reads raw bytes.

[Write](#) Writes raw bytes.

[WriteString](#) Writes a single line of text.

[ReadString](#) Reads a single line of text.

CArchive::operator >>

```
friend CArchive& operator >>( CArchive& ar, CObject *& pObj );
throw( CArchiveException, CFileException, CMemoryException );

friend CArchive& operator >>( CArchive& ar, const CObject *& pObj );
throw( CArchiveException, CFileException, CMemoryException );

CArchive& operator >>( BYTE& by );
throw( CArchiveException, CFileException );

CArchive& operator >>( WORD& w );
throw( CArchiveException, CFileException );

CArchive& operator >>( int& i );
throw( CArchiveException, CFileException );

CArchive& operator >>( LONG& l );
throw( CArchiveException, CFileException );

CArchive& operator >>( DWORD& dw );
throw( CArchiveException, CFileException );

CArchive& operator >>( float& f );
throw( CArchiveException, CFileException );

CArchive& operator >>( double& d );
throw( CArchiveException, CFileException );
```

Return Value

A **CArchive** reference that enables multiple insertion operators on a single line.