**Brief Introduction about Memory Queue -- CUQueue and CScopeUQueue Objects**

*UDAParts*[*support@udaparts.com*](mailto:support@udaparts.com) *Updated on Nov. 22, 2016*

Contents

1. *Introduction*
2. *C++, C#/VB.NET, Java and Python samples*
3. *Key design considerations on CUQueue and CScopeUQueue*
4. *Compatibility among different development languages*
5. *Object within different development languages*
6. *Use of .NET serialization with CUQueue*
7. *String serialization*
8. *Generics Save/Load for .NET and Java as well as operators <</>> in C++*
9. *IUSerializer of .NET, Java and Python for complex user-defined classes and structures*
10. *null/nothing objects serialization supported*
11. *Use CScopeUQueue within function stack whenever possible*

**1.****Introduction**

SocketPro development requires converting different types of data into bytes at adapter level for all supported development languages. The conversion is not difficult at all, but it is considerably tedious. In order to ease SocketPro development, SocketPro provides a helper class *CUQueue* within each adapter of languages to efficiently pack various types of data compatibly into bytes. Later, you can effortlessly unpack requests and returned results in binary format into original types of data compatibly at both server and client sides.

**2.    C++, C#/VB.NET, Java and Python samples**

See the following demonstration applications:

* Simple serialization and de-serialization: C++, C#/VB.NET, Java and Python projects at the directory ..\SocketPro\tutorials\(cplusplus|csharp|vbnet|java\src|ce|python)\uqueue\_demo.
* Exchange of complex structure between client and server: C++, C#/VB.NET, Java and Python projects at the directory ..\SocketPro\tutorials\(cplusplus|csharp|vbnet|java\src|ce|python)\hello\_world.

**3.****Key design considerations on *CUQueue* and *CScopeUQueue***

*CUQueue* is designed with a set of key considerations:

* **Easy to use**. You are never required to manage internal memory directly.
* **Auto-manage memory.** When you save, push or insert a data into an instance of *CUQueue*, it will automatically increase memory if necessary.
* **High efficiency.** Use of *CUQueue* can eliminate allocating and de-allocating memories repeatedly, which significantly improves application performance and reduces memory resources and page faults.
* **Great compatibility among different development languages such as Python, Java, VB.NET, C# and C/C++.** Compatibility is guaranteed across development languages and platforms.
* **CScopeUQueue***.* CScopeUQueue is created for reuse of CUQueue without repeatedly allocating memory. When an instance of CScopeUQueue is disposed or destroyed, its internal CUQueue instance will be put back into a pool of CUQueue objects for reuse. When creating an instance of CScopeUQueue, it automatically takes one from the pooled CUQueue objects if available. If there is no CUQueue object available, a new CUQueue object will be created on-the-fly.

**4.****Compatibility among different development languages**

SocketPro supports development using different languages. For example, you can even develop a C++ client application that directly accesses codes of any dotNet languages without requiring any other middle components. To achieve compatibility, the class *CUQueue* is written in mind of compatibility among different development languages for a set of basic data types. Here is the table for compatibility among five major development languages, Python, Java, C#, VB.NET and C/C++.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Data Type | Size (byte) | Java | .NET (C#/VB.NET) | C/C++ | Python |
| unsigned byte | 1 | byte | byte | unsigned char | Save/LoadByte |
| signed byte | 1 | byte | sbyte | signed char | Save/LoadAChar |
| bool | 1 | boolean | bool | bool | Save/LoadBool |
| short | 2 | short | short | short | Save/LoadShort |
| int | 4 | int | int | int | Save/LoadInt |
| float | 4 | float | float | float | Save/LoadFloat |
| double | 8 | double | double | double | Save/LoadDouble |
| unsigned short | 2 | short | ushort | unsigned short | Save/LoadUShort |
| unsigned int | 4 | int | uint | unsigned int | Save/LoadUInt |
| decimal | 16 | Java.math.BigDecimal | decimal | DECIMAL | Save/LoadDecimal |
| Currency | 8 | N/A | decimal | Currency | N/A |
| LONGLONG | 8 | long | long | LONGLONG | Save/LoadLong |
| ULONGLONG | 8 | long | ulong | ULONGLONG | Save/LoadULong |
| wide char | 2 | char | char | UTF16 low-endian | Save/LoadChar |
| GUID | 16 | Java.util.UUID | Guid | GUID | Save/LoadUUID |
| Date time\* | 8 | Java.util.Date | DateTime | UDateTime | Save/LoadDate |
| ASCII string |  | byte[] | byte[] or sbyte[] | std::string | Save/LoadAString |
| wide string |  | String | string | std::wstring or CComBSTR on win | Save/LoadString |
| Object |  | Object | object | CComVaraint (win)  boost::variant(Linux) | Save/LoadObject |

\* -- SocketPro uses a long integer (8 bytes) to represent date time across different platforms and languages. Date time can be accurate to one microsecond.

**5.    Object within different development languages**

All data types or their array can be boxed and unboxed into and from an Object data within Java and .NET platforms, which are similar to window *tagVARIANT* structure. SocketPro uses variant in boost libraries to represent the Object data type for C++ development on non-window platforms. The class *CUQueue* also supports one dimension array of data types listed in the above table through Object data type, including Object data type itself.

**6.****Use of dotNet serialization within CUQueue**

Dotnet version of *CUQueue* can take advantage of native dotNet serialization if your development is only involved with managed code. You can call methods *CUQueue*::*Serialize* and *Deserialize* to save and load a managed object into and from a *CUQueue* instance, respectively. Dotnet serialization makes the code simpler, but slower and is restricted for desktop applications and middle tiers running on .NET environment only. DotNet serialization is not interoperable with non-dotNet codes. Furthermore, note that .NET codes for devices do not support native .NET serialization either.

**7.****String serialization**

To serialize a string, we first save a length in unsigned int (4 bytes) in bytes, and actual string context afterward in UTF16 low-endian format by calling the method *Save* or operator << in C++. If the length is -1 or 0xFFFFFFFF, the string is a null for Java, C++ and C#. As you can see, the class *CUQueue* first pops out a length in unsigned int first, and string context afterwards when loading a string from an instance of *CUQueue*.

**8.****Generics Save/Load for .NET and Java as well as operators << and >> in C++**

To make code simpler, SocketPro adapter for .NET now uses the generics methods *Save* and *Load* for saving and reading data to or from an instance of *CUQueue*. Internally, the adapter will use a proper method (*Save*/*Load* for primitive data types, *Save/Load* for string, *IUSerializer* or *Serialize/Deserialize* for .NET objects) to save or load data from generics data type. In regards to C++, you may need to override operators << and >> to save or load instances of complex user-defined structures. For simple data types or structures, adapter for C++ will automatically handle them for you correctly through templates.

**9.****IUSerializer of .NET, Java and Python for complex user-defined classes or structures**

To serialize or de-serialized a complex user-defined class or structure, it is highly recommended that the class or structure be implemented with the interface *IUSerializer*. The implementation is very simple since you can easily make .NET code and native compatible to each other. Hence, it makes code run faster than .NET serialization without involvement of reflection. In addition, .NET serialization is not available to .NET compact environment, but *IUSerializer* is fully supported on devices. To see such details, please refer to the above samples.

**10.    null/nothing object serialization supported**

At this time, Java and .NET null/nothing objects serialization and de-serialization are well supported for three types of objects.

* A simple object containing primitive data types, string and arrays of primitive data types and string.
* An instance of a user-defined class or structure implemented with the interface *IUSerializer*.
* A native .NET serializable object.

Also, you can easily make the first two types of objects compatible with not only native code development like C/C++ but also .NET compact framework.

**11.** **Use CScopeUQueue within function stack whenever possible**

Whenever an instance of *CScopeUQueue* is disposed, its internal *CUQueue* will be recycled into an internal memory pool for use in the future as described in section 3 above. For example, you should use the code style below whenever possible.

**using (CScopeUQueue su = new CScopeUQueue())**

**{**

**int n = 12345;**

**string str = "test";**

**su.Save(n).Save(str);**

**//The above call is equal to the below three calls**

**//CUQueue UQueue = su.UQueue;**

**//UQueue.Save(n);**

**//UQueue.Save(str);**

**//do whatever you like here ......**

**} //Internal CUQueue will be recycled into internal pool for reuse in the future here**