Serializable Objects

Ice for Java and Ice for .NET allow you to send native Java and CLR objects as operation parameters. The Ice run time automatically serializes and deserializes the objects as part of an invocation. This mechanism allows you to transmit Java and CLR objects that do not have a corresponding Slice definition.

On this page:

The `serializable` Metadata Directive

To enable serialization, the parameter type must be a byte sequence with appropriate metadata. For example:

Slice

```
["java:serializable:SomePackage.JavaClass"]
sequence<byte> JavaObj;

interface JavaExample {
    void sendJavaObj(JavaObj o);
};

["clr:serializable:SomeNamespace.CLRClass"]
sequence<byte> CLRObj;

interface CLRExample {
    void sendCLRObj(CLRObj o);
};
```

The `java:serializable` metadata indicates that the corresponding byte sequence holds a Java serializable type named `SomePackage.JavaClass`. Your program must provide an implementation of this class; the class must implement `java.io.Serializable`.

Similarly, the `clr:serializable` metadata indicates that the corresponding byte sequences holds a CLR serializable type named `SomeNamespace.CLRClass`. Your program must provide an implementation of this class; the class must be marked with the `Serializable` attribute.

Architectural Implications

The `serializable` metadata directive permits you to transmit arbitrary Java and CLR objects across the network without the need to define corresponding Slice classes or structures. This is mainly a convenience feature: you could achieve the same thing by using ordinary Slice byte sequences and explicitly serializing your Java or CLR objects into byte sequences at the sending end, and deserializing them at the receiving end. The `serializable` metadata conveniently takes care of these chores for you and so is simpler to use.

Despite its convenience, you should use this feature with caution because it destroys language transparency. For example, a serialized Java object is useless to a C++ server. All the C++ server can do with such an object is to pass it on to some other process as a byte sequence. (Of course, if that receiving process is a Java process, it can deserialize the byte sequence.)

Further, similar to Slice classes with methods, a serialized object can be deserialized only if client and server agree on the definition of the serialized class. In Java, this is enforced by the `serialVersionUID` field of each instance; in the CLR, client and server must reference identical assembly versions. This creates much tighter coupling of client and server than exchanging Slice-defined types.

And, of course, if you build a system that relies on, for example, the exchange of serialized Java objects and you later find that you need to add C++ or C# components to the system, these components cannot do anything with the serialized Java objects other than pass them around as a blob of bytes.

So, if you do use these features, be clear that this implies tighter coupling between client and server, and that it creates additional library versioning and distribution issues because all parts of the system must agree on the implementation of the serialized objects.

See Also

- Serializable Objects in Java
- Serializable Objects in C#
- Architectural Implications of Classes

Copyright © 2017, ZeroC, Inc.

Copyright © 2017, ZeroC, Inc.