# Djinni .Net Extension

Easton

a7031x@gmail.com

## Purpose

To connect C++ with Windows RT .Net. In this article, let’s use C# to represent any Windows RT .Net language.

In the solution, I use C++/CLI as an intermediate language to connect C++ and Windows RT .Net.

## Type transition

Windows RT Component doesn’t support exporting IBox, so for optional types, need to see if the type is reference or value.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **primitive** | **string** | **binary** | **optional** | **containers** |
| C++ | value | value | value | value | value |
| CLI | value | value | reference | reference | reference |
| C# | value | reference | reference | reference | reference |

For binary and contains, either CLI or C# are all reference types, no extra wrapper for these types. For primitive and string types, need to wrap them as nullable types. Take integer for example, define Integer32 as reference type for int.

public ref class Integer32 sealed

{

public:

property int Value;

Integer32(int value) { Value = value; }

};

Although String in C# is nullable, when translated to CLI in WinRT system, the Platform::String^ is an empty string other than null, so need to wrap nullable string as,

public ref class StringRef sealed

{

public:

property Platform::String^ Value;

StringRef(Platform::String^ value) { Value = value; }

};

**Type translation table**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Djinni** | **primitive** | **string** | **binary** | **optional** | **list** | **map** | **set** |
| C++ | Same | std::string | std::vector<char> | std::optional | std::vector | std::map | std::set |
| CLI | Same | Platform::String | Platform::Array<char> | reference | wfc::IVector | wfc::IMap | wfc::IIterable |
| C# | Same | string | byte[] | reference | IList | IDictionary | IEnumerable |

## Interface to C++

CLI class can keep C++ type as its field, so proxy is mixed with interface definition.

public ref class SomeInterface sealed

{

public:

Platform::String^ getString ()

{

auto cppRet = \_cppRef->getString();

return transform<std::string, Platform::String^>()(cppRet);

}

internal:

SomeInterface(std::shared\_ptr<::cppns::SomeInterface > cppRef)

: \_cppRef(cppRef) {}

std::shared\_ptr<::cppns::SomeInterface> cppRef() {return \_cppRef;}

private:

std::shared\_ptr<::cppns::SomeInterface > \_cppRef;

};

Will talk about transform<> later.

## Interface to C#

Need to create a proxy. Note that C++ class cannot keep CLI types as fields, this approach plays a trick, wrapping the CLI reference in a lambda expression.

template<>

class CxInterfaceProxy<::cppns::ILog> : public ::cppns::ILog

{

public:

CxInterfaceProxy(ILog^ nativeRef)

{

\_nativeRef = [nativeRef]{ return nativeRef; };

}

void error(const std:: & tag, const std::string& message) override

{

nativeRef()->error(

transform<std::string, Platform::String^>()(tag), transform<std::string, Platform::String^>()(message));

}

void warning(const std::string& tag, const std::string& message) override

{

nativeRef()->warning(

transform<std::string, Platform::String^>()(tag),

transform<std::string, Platform::String^>()(message));

}

…

ILog^ nativeRef() { return \_nativeRef(); }

private:

std::function<ILog^()> \_nativeRef;

};

public interface class ILog

{

public:

virtual void error (Platform::String^ tag, Platform::String^ message);

virtual void warning (Platform::String^ tag, Platform::String^ message);

virtual void info (Platform::String^ tag, Platform::String^ message);

virtual void debug (Platform::String^ tag, Platform::String^ message);

virtual void verbose (Platform::String^ tag, Platform::String^ message);

};

## Record Translation

public ref class SomeRecord sealed

{

public:

property Platform::String^ uri;

property SomeOtherRecord^ data;

internal:

::cppns::SomeRecord toCpp()

{

::cppns::SomeRecord ret;

ret.uri = transform<std::string, Platform::String^>()(uri);

ret.data = transform<::cppns::SomeOtherRecord, SomeOtherRecord^>()(data);

return ret;

}

static SomeRecord^ fromCpp(const ::cppns::SomeRecord& value)

{

SomeRecord^ ret = ref new SomeRecord();

ret->uri = transform<std::string, Platform::String^>()(value.uri);

ret->data = transform<::cppns::SomeOtherRecord, SomeOtherRecord^>()(value.data);

return ret;

}

};

## All mighty translation

As mentioned previously, transform<> operator performs type translation between CLI and C++, leveraging the powerful C++ template deduction.



## Leftover

With the all-mighty translation, the scala code to generate CLI files would be straightforward.