X 对原生动态库方法的调用说明

文档库地址 https://xlang.link/documents/index.html

X 为了更好的利用现有的生态资源，对 C/C++ 以及其他更低级语言生成的原生态的 Native 动态链接库(例如*.dll \ *.so \ *.Dylib，以下简称为 NativeLib)的调用有两种方式.

1. **从 X -> NativeLib 的调用. 即从 X 项目中调用 dll 或者 so 提供的方法.**
2. **从 Native Lib -> X， 即从 C++项目调用 X 中的方法.**

1). X 调用 NativeLib

目前支持动态链接和静态链接两种形式来调用 C/C++或者汇编编写的动态链接库，此种方法无需对 NativeLib 进行任何更改， 支持调用各种闭源的动态链接库，仅需要在 X 中声明方法原型即可使用。

在 X 中需要对将调用 Nativelib 的导出方法 (以下简称 Native 方法) 进行声明，方法声明与 X 中的一般方法略有不同，且参数类型有限制；

声明形式:

```
返回值 调用约定 方法名(参数列表);
```

如：

DLL 中方法的 C 形式声明:

```
extern“C” int __cdecl add(int x,int y);
```

则在 X 中进行如下声明:

```
int cdecl add(int x,int y);
```

注意：native 方法声明中必须有调用约定，对于固定调用约定(如 x86_64)的操作系统，调用约定将被忽略.

支持的调用约定:

X 调用约定声明	对应 C++中的调用约定
stdcall	__stdcall
cdecl	__cdecl
fastcall	__fastcall
pascal	__pascal

注意:参数个数至多支持 32 个参数, native 参数类型详见文档尾部说明.

1.静态链接；

说明：静态加载是需要将 NativeLib 的路径和名称直接写入代码中，X 虚拟机在启动时将对此 NativeLib 进行动态链接， 若加载失败，则程序不能运行，类似 C++项目中使用 lib 或者.a 进行静态链接。

例:

```
class NativeLibrary{
    import "libtestxnl"{
        int cdecl add(int x,int y);
    };
};
```

该示例声明了从名为 libtestxnl 的动态库中链接 add 方法，（nativeLib 的扩展名可以不用写全，虚拟机会根据不同的平台自动寻找），程序加载前会依次在系统目录、工作目录、程序所在目录寻找该库，若没有找到则打印错误消息并终止退出。

使用方法：

```
int n = NativeLibrary.add(1, 2);
```

2.动态链接；

说明：动态加载是根据代码逻辑在需要的时候进行加载，加载失败将抛出异常，但不会导致程序终止，类似 C/C++里面的 LoadLibraryA 、dlopen；

例:

```
class NativeLibrary : Library{
    import {
        int cdecl add(int x,int y);
    };
};
```

该示例代码在使用前需要手动显式加载模块 ：

```
try{
    NativeLibrary.loadLibrary("testxnl.dll");
}catch(Exception e){
    // 将 错 误 打 印 到 调 试 器 输 出 窗 口
    _system_.output(e.getMessage());
}
```

使用方法：

```
int n = NativeLibrary.add(1, 2);
```

2). NativeLib 调用 X

从 NativeLib 调用 X 中的方法\类成员方法相对需要稍做更多工作，这种方法需要在 Nativelib 的代码中借助 X 和 Native 交互的 C++接口实现；

在 XStudio 的安装目录下的 xnl 文件夹有 xnl.h 头文件，该文件描述了从 C++代码调用 X 的接口规范(XNLEnv 接口)。

从 C++代码中调用 X 方法的 NativeLib 中需要实现入口方法和卸载方法:

```
XNLEXPORT int XI_CDECL XNLMain(XNLEnv * env, int version);
XNLEXPORT int XI_CDECL XNLExit(XNLEnv * env);
```

程序启动时将调用 XNLMain 进行初始化:

原型：XNLEXPORT int XI_CDECL XNLMain(XNLEnv * env, int version);

XNLEnv * env 为 X 虚拟机环境接口， 该对象通过 getEnv 获取的接口副本可以在 NativeLib 中作为全局保存，接下来的一切调用 X 的方法都依赖此接口对象；

version 为 X 虚拟机版本；

程序退出前将对 XNLExit 进行调用， 以告知 NativeLib 程序运行结束，需销毁对象；

原型：XNLEXPORT int XI_CDECL XNLExit(XNLEnv * env);

该方法被调用时必须释放在 XNLMain 中保存的虚拟机对象接口；

入口和退出的示例:

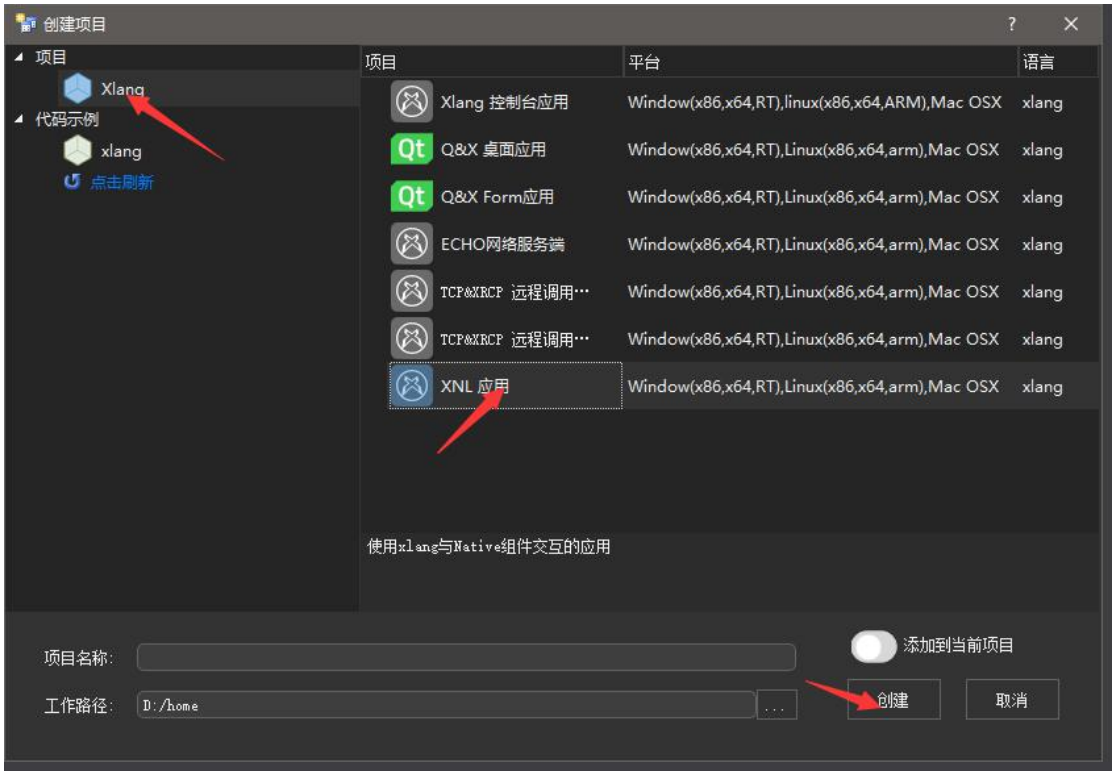
```
XNLEnv * gs_env = 0;

XNLEXPORT int XI_CDECL    XNLMain(XNLEnv * env, int version){
      gs_env = env->getEnv();
      return 0;
}

XNLEXPORT int XI_CDECL    XNLExit(XNLEnv * env){
      gs_env->releaseEnv(gs_env);
      return 0;
}
```

代码示例:

请使用 XStudio 新建模板:



该项目模板中包含了一个 X->C++和 C++->X 的调用示例；

其中包含的 C++项目可以在 Windows 下可使用 Visual Studio 系列进行编译，

在 Linux 或者 Darwin 下使用 Make 编译

说明：Linux 和 Darwin UNIX 下 XStudio 组建项目时检测到存在 makefile 将自动调用 make 对 Native 项目进行编译,省去手动编译的步骤.

Native 方法声明中支持的数据类型:

X 中的 Native 数据类型	对应 C++中的数据类型	用途	说明
void	void	用于方法返回值	不关心返回值
int	int32_t	返回值或者参数类型	32 位有符号整数
long	long long	返回值或者参数类型	64 位有符号整数
byte	unsigned char	返回值或者参数类型	8 位无符号整数
char	signed short	返回值或者参数类型	16 位有符号整数
bool	bool	返回值或者参数类型	BOOLEAN 类型
double	double	返回值或者参数类型	64 位双精度浮点数
String	const char *	返回值或者参数类型	char 类型指针 用于一般字符串
ObjectPtr	void*	参数类型	对 X 中的简单类型对象取地址操作；见例 1
Object	XObject *	返回值或者参数类型	用于 C++中操作 X 复杂对象；见例 2
Pointer	void*	返回值或者参数类型	用于 C++<->X 的指针类型转换.见例 3
ObjectRef(ver2.7 新增)	XObject *	对 xlang 中对象的引用传参	xlang 对象的指针

详细作用见附表 1.(ver2.7 新增)

例 1：

使用 ObjectPtr 进行对象取地址操作：

C++中的代码:

```
void __cdecl getIntValue(int * pValue){
    *pValue = 99;
}
```

对应 X 中的声明:

//参数类型为 ObjectPtr 表示对 X 中输入的参数进行取地址操作

```
void cdecl getIntValue(ObjectPtr pValue);
```

例 2：

使用 Object 对 X 中的对象进行操作:

C++ 中的代码:

```
extern ”C”  XObject * XI_CDECL getFileContent(const char * path){
    FILE * fp = fopen(path, "rb");
    if (fp){
        char buffer[1024];
        int rd = fread(buffer, 1, 1024, fp);
        fclose(fp);
        //使用 X 接口创建一个 Byte 数组的 Object 对象;
        XObject * pb = gs_env->createByteArray(rd);
        //填充 Object 数据
        gs_env->setElementValue(pb, 0, &buffer[0], rd);
        //返回 Object 对象
        return pb;
    }
    //返回 0 或者 NULL X 中对应为 nilptr
    return 0;
}
```

X 中的声明:

```
Object cdecl getFileContent(String path);
```

例 3 ：

使用 Pointer 对 C++的指针进行操作;

C/C++ 中的代码(模块 mysqlite.dll):

```
class Sqlite{
    bool openDatabase(const char * path){
        return true;
    }
};

extern ”C”  Sqlite* __cdecl createSqlite(){
    return new Sqlite();
}

extern ”C”  bool __cdecl openDatabase(Sqlite* sql, const char * path){
    return sql->openDatabase(path);
}
```

X 中的代码:

```
class Sqlobj : Library{
    import{
        //对应 C++ 代码中的 extern ”C”  Sqlite* __cdecl createSqlite()
        Pointer cdecl createSqlite();
        //对应 C++代码中的 extern ”C”  bool __cdecl openDatabase(Sqlite* sql,const char * path)
        bool cdecl openDatabase(Pointer hdb, String path);
    }
};
int main(String[] args){
    try{
        Sqlobj.loadLibrary(“mysqlite.dll”);
    } catch (Exception e){
    }
    //具体用法
    long hdb = Sqlobj.createSqlite();
    Sqlobj.openDatabase(hdb, ”d:\my.db”);
}
```

注意：在 X 中须使用 long 类型和 Pointer 互相操作， 返回值为 Pointer 的用 long 接收，参数为 Pointer 的也需使用 long 类型的数据进行传递，无须担心 32 和 64 位指针大小不一致的情况;

附表 1：

Object 类型传参:

Object 类型将参数**临时对象**传给 Native 层，但不引用，Native 层对**该参数对象**本身进行的任何更改都不会影响 xlang 层对象的值。**注意：如 xlang 中对象为 nilptr 时，native 层收到的参数为 NULL。**
如:
void cdecl native(Object obj);

xlang 代码
```
int k = 0;
native(k);
```

c++代码:
```
void native(XObject * obj){
    env->setValue(obj, 1);     // 并不会改变 xlang 中 k 的值;
}
```

ObjectPtr 类型传参：

ObjectPtr 的参数传递时根据不同类型 native 层收到的类型也不同:

xlang 中的数据类型	C++收到的数据类型	C++类型
byte	byte*	unsigned char *
char	xchar *	unsigned int16_t *
short	xshort *	signed int16_t *
int	xint *	int32_t *
long	xlong *	int64_t *
double	double *	double *
bool	bool *	bool *
class[]	class *	class *
String	xstring *	const char *(utf8)
其他类型的数据	XObject *	XObject *

数组对象使用 ObjectPtr 传参

byte[]	byte*	unsigned char *
char[]	xchar *	unsigned int16_t *
short[]	xshort *	signed int16_t *
int[]	xint *	int32_t *
long[]	xlong *	int64_t *
double[]	double *	double *
bool[]	bool *	bool *
String []	XObject *	XObject *
Object []	XObject *	XObject *

ObjectRef 传参:

ObjectRef 类型将参数对象直接传给 Native 层，注意是引用，Native 层对**该参数对象**本身进行的更改会影响 xlang 层对象的值。
如:

xlang 代码

```
void cdecl native(ObjectRef    obj);

int k = 0;
native(k);
```

c++代码:

```
void native(XObject * obj){
    env->setValue(obj, 1);      // 会改变 xlang 中 k 的值;
}
```

有任何疑问请进入 

QQ 群：591392649，进行交流探讨.