Данная статья касается темы, сильно будоражащей умы людей, имеющих отношение к разработке на платформе 1с 8.3.х.х, и вместе с тем, достаточно непростой, – разработка внешних Native-компонент к мобильной платформе 1С.

Отчего будоражит.

Мобильная 1С платформа позволяет очень быстро и наглядно произвести разработку мобильного приложения для устройства, работающего под управлением ОС Android, реализующего бизнес-логику практически неограниченной сложности. Вместе с тем, последствиями упрощения процесса разработки явились:

- ограничение возможностей взаимодействия с операционной системой и аппаратной частью исполняющего устройства,

- снижение скорости выполнения кода,

- ограничение возможностей реализации визуальной составляющей интерфейса.

Если возможностей среды 1с для реализации интерфейса недостаточно, то для этого есть другие универсальные фреймворки: Xamarin, Flutter. Остальные 2 проблемы могут быть решены с использованием внешних компонент.

Почему непросто.

- Вопрос находится на стыке нескольких стеков технологий и сред разработки - нужно хоть немного понимать, как функционирует каждая из сред.

- захотеть разобраться в вопросе, осмыслить, повторить решение и потом расширить функционал.

- На курсах этому не учат.

Вообще, создается впечатление, что 1с намеренно затрудняет расширение возможностей платформы. Например, релиз мобильной платформы 1с состоялся в конце 2015 года, но за прошедшие 5 лет в платформу так и не был добавлен функционал приема броадкаст-сообщений, хотя это бы решило множество проблем, возникающих при организации взаимодействия с подсистемами устройства и другими приложениями.

Вместе с тем, после осмысления и освоения данного материала, бонусом для освоившего и осмыслившего станет принципиальная возможность разработки внешних компонент по технологии Native под любую архитектуру процессоров и любую операционную систему, которая допускает использование технологической платформы 1С.

Содержание.

1. Архитектура решения.
   1. ~~Принципы работы Android-Native части.~~
      1. ~~Кратко о broadcast’ах и фильтрах и зачем нам это.~~
   2. ~~Принципы работы транспортной части (назначение библиотеки динамической линковки в решении).~~
      1. ~~JNI (Java native interface) и JNI-calls.~~
   3. ~~Подготовка пакета (непосредственно файла) для загрузки в двоичные данные конфигурации и последующее использование в прикладном решении на мобильной платформе 1с.~~
      1. ~~Android ABI (Application binary interface).~~
      2. ~~Манифест.~~
      3. ~~Условие об именовании файлов пакета компоненты.~~
2. Практическая часть.

1. Архитектура решения.

Источники:

<https://its.1c.ru/db/metod8dev/content/3221/hdoc>

<https://infostart.ru/1c/articles/987286/>

<https://infostart.ru/public/184119/>

<https://developer.android.com/>

<https://stackoverflow.com/questions/32470463/what-is-the-naming-convention-for-java-native-interface-method-and-module-name>

Архитектура Native-компоненты под Андроид (далее – «ВК») состоит из 3х звеньев:

1. Динамической библиотеки \*\*.so, которая как-бы является транспортом между платформой 1с и Android-native приложением. Библиотека линкуется с платформой и одновременно линкуется с Android-native приложением. Это позволяет реализовать передачу данных между ними через JNI. Разработка на C++ (11 релиз).

2. Логика реализуется путем разработки Android-native приложения (можно вообще без Activivty - т.е. без какой-либо интерфейсной части). Эта часть ВК непосредственно взаимодействует с ОС устройства, другими приложениями. Разработка ведется в Android-Studio (Java).

Минимальную компоненту можно сделать на C++, используя API ядра Linux. Но, к сожалению, их множество очень ограничено, поэтому трудно обойтись без JNI. С его помощью можно создавать и вызывать функции объектов java и, за некоторыми исключениями, написать компоненту полностью. Но перевести весь java-код в JNI очень трудоемко и чревато трудно обнаруживаемыми ошибками. Это может пригодится только, если вы хотите защитить свой код. В обычной практике лучше использовать часть на C++ как промежуточный слой, а главную реализацию сделать на Java.[[1]](#footnote-1)

Например, на С++ без проблем можно реализовать работу с регулярными выражениями (https://infostart.ru/public/940766/). Несомненно, код на c++, на этапе сборки статически скомпилированный в машинный код, в среднем, будет работать быстрее, чем код, написанный под любую VM, в т.ч. и Android-native-код.

3. Прикладной уровень. Т.е. это уровень прикладного решения 1С, в котором, вызывая процедуры установки компоненты, мы заставляем все это работать. Ну, пытаемся…

На самом деле, архитектура работает на удивление стабильно. Падение возникало только в одном месте - когда библиотека заменялась на другую версию (без остановки всех потоков платформы 1С и очистки кэша приложения).

* 1. ~~Принципы работы Android-Native части.~~

~~Разработчики платформы 1С предоставляют нам доступ к движку Java через API getJniEnv() и интерфейс IAndroidComponentHelper. Последний обеспечивает доступ к контексту мобильного приложения через GetActivity() - функцию, возвращающую активность приложения. Класс java, отвечающий за взаимодействие с кодом на C++, реализует интерфейс Runnable. Это - признак многопоточности, т.е. компонента запускается в отдельном потоке приложения. Одно из следствий этого - невозможность взаимодействия с интерфейсом приложения и создания собственных активностей[[2]](#footnote-2).~~

~~Обратные вызовы платформы из Java-кода функций C++ выполняются через Java native call. Часто в технологии внешних компонент эти вызовы используются для генерации внешних событий.~~

~~Базовое Native – приложение состоит из класса, реализующего интерфейс Runnable.~~

~~В классе реализована процедура загрузки динамической библиотеки, процедура запуска в потоке, который выделила 1с для нашей ВК, конструктор (принимает контекст Activity приложения 1с и указатель на интерфейс платформы 1С) и полезной нагрузки.~~

~~1.1.1. Кратко о broadcast’ах и фильтрах.~~

[~~https://developer.android.com/guide/components/broadcasts~~](https://developer.android.com/guide/components/broadcasts)

~~В Android существует понятие широковещательных сообщений, которые можно отправлять или принимать. Оба процесса между собой не связаны и их можно использовать по отдельности.~~

~~Это все похоже на радио. Т.е. отправитель с помощью намерения (в терминологии ОС Андроид - Intent) говорит операционной системе, что хочет открыто заявить всем что-то. Для этого указывается фильтр (просто строка) и получатель, которым может быть какой-то пакет (другое приложение) или даже отдельный класс приложения. Получатель, впрочем, может и не указываться. Тогда сообщение становится широковещательным (Broadcast – в терминологии ОС Андроид).~~

~~Сообщение (в т.ч. броадкаст) может содержать дополнительные данные как в виде строки, так и просто массив байт. Например, модули лазерных сканеров практически любого современного терминала сбора данных («ТСД» - далее) при сканировании формируют соответствующий броадкаст, который можно перехватить и получить из него данные штрихкода. Единственное условие – знать установленный на него фильтр, например: «com.xcheng.scanner.action.BARCODE\_DECODING\_BROADCAST» и ключ доп. данных (“extras” – в терминологии Андроид), например: «EXTRA\_BARCODE\_DECODING\_DATA».~~

~~Перехват осуществляется в callback-методе (может быть безымянным), реализующем интерфейс «BroadcastReceiver» (см - код).~~

~~При разблокировке экрана ОС Андроид тоже шлет броадкаст «android.intent.action.SCREEN\_ON», при запуске ОС шлет броадкаст «boot\_completed» и т.д.~~

~~Т.е. схема передачи данных штрихкода из модуля лазерного сканера выглядит так:~~

~~1. Броадкаст,~~

~~2. Перехватчик,~~

~~3. JNI-Call (вызов библиотеки – транспорта через JNI),~~

~~4. Вызов метода «ExternalEvent» интерфейса «IAddInDefBaseEx» платформы 1с, в динамической библиотеке (которая «транспортная»),~~

~~5. Перехват ВнешнегоСобытия в 1с, на прикладном уровне.~~

~~1.2. Принципы работы транспортной части.~~

~~Транспортная часть ВК реализуется в виде библиотеки динамического связывания (расширение .so – в ОС Андроид), написанной на C++. В двух словах, это~~

~~Мобильная платформа 1С предоставляет разработчикам внешних компонент интерфейс «IComponetnBase» (файл «ComponentBase.h»). Он является базой функционирования как мобильной так и десктопной внешней компоненты и состоит из интерфейсов:~~

~~IInitDoneBase - шлюз для получения функциональности платформы: загрузка/выгрузка, управление памятью, генерация внешних событий.~~

~~ILanguageExtenderBase - описатель типа компоненты, предоставляет возможности для расширения встроенного языка 1С.~~

~~LocaleBase - управление локализацией компоненты.~~

~~При начале работы с компонентой 1С вызывает несколько классических экспортных C-функций из динамической библиотеки компоненты:~~

~~- GetClassNames - возвращает имена объектов вашей компоненты, это то, что пишется после на месте <объект> в коде 1С: Новый("AddIn.<псевдоним>.<объект>");~~

~~- GetClassObject - создает ваш объект компоненты, реализующий IComponentBase;~~

~~- DestroyObject - удаляет объект~~

~~- SetPlatformCapabilities - получает версию платформы (для мобильной платформы в настоящий момент версия не важна)~~

~~Таким образом, наша задача - реализовать методы интерфейса IComponentBase.~~

1. Практическая часть. Разработка.

Будем использовать Visual Studio 2019.

Сначала создадим пустой проект библиотеки динамической линковки и назовем его «TestPro»:

<http://wiki.coolbase.cd.local/wp-content/uploads/2021/05/new-SO.png>

После этого структура проекта будет такая:

<http://wiki.coolbase.cd.local/wp-content/uploads/2021/05/Pro_struct.png>

Выкидываем из него все, что VS2019 туда понасовало. После этого остается пустая болванка, пригодная для разработки ВК:

<http://wiki.coolbase.cd.local/wp-content/uploads/2021/05/empty.png>.

Добавляем в проект заголовки и 1с-ный функционал из шаблона 1с, скачанного с ИТС (<https://its.1c.ru/db/files/1CITS/EXE/VNCOMPS/VNCOMPS.zip>).

Тупо копируем папки:

http://wiki.coolbase.cd.local/wp-content/uploads/2021/05/Copy-headers-to-new-proj.png.

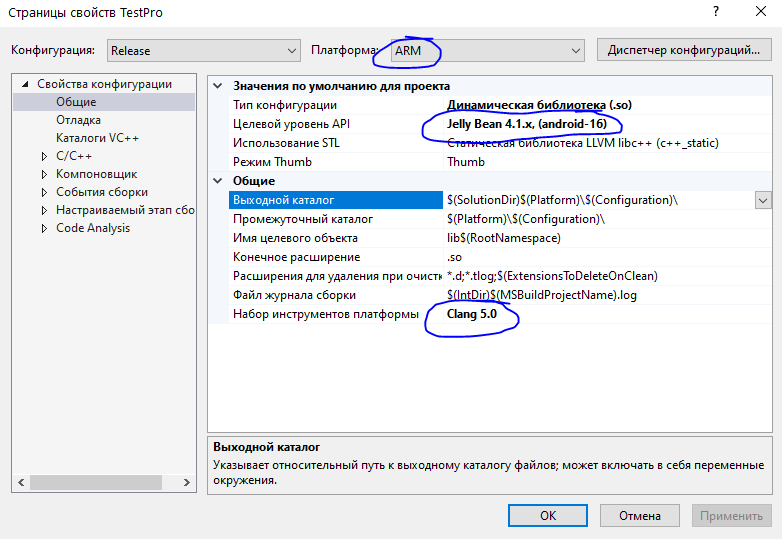
Потом добавляем все это в проект в VS + добавляем наш класс, в котором будем реализовывать взаимодействие с Java VM через JNI (назовем его как-нибудь тупо, например, «JNI\_Super»):

<http://wiki.coolbase.cd.local/wp-content/uploads/2021/05/Full-project.png>.

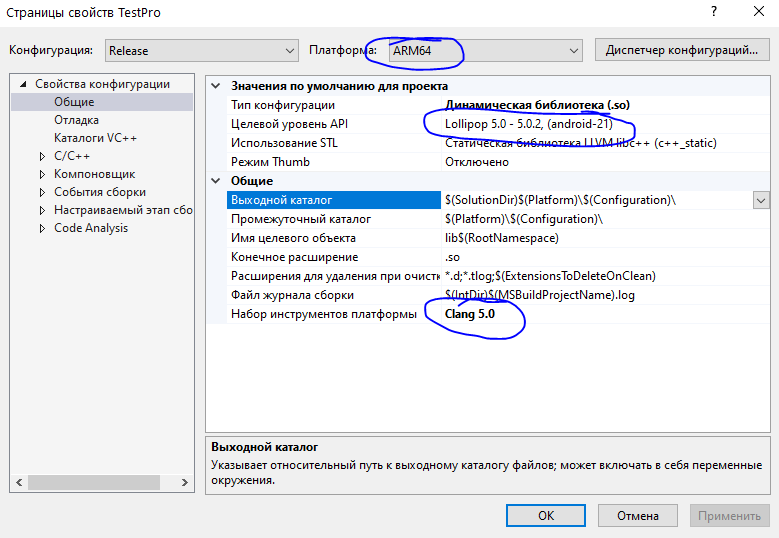
Если размещение заголовочных файлов в каталогах произведено не верно, то VS2019 вывалит кучу ошибок линковки. В этом случае могу порекомендовать еще раз внимательно посмотреть, куда были скопированы все файлы проекта.

Теперь следует настроить каждую из архитектур в свойствах проекта (указываем набор инструментов платформы и целевой уровень АПИ Андроида).

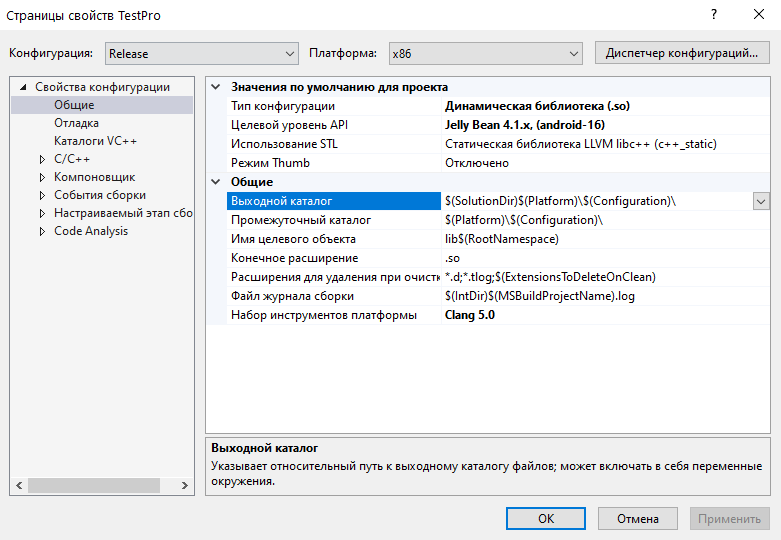
1. ARM:



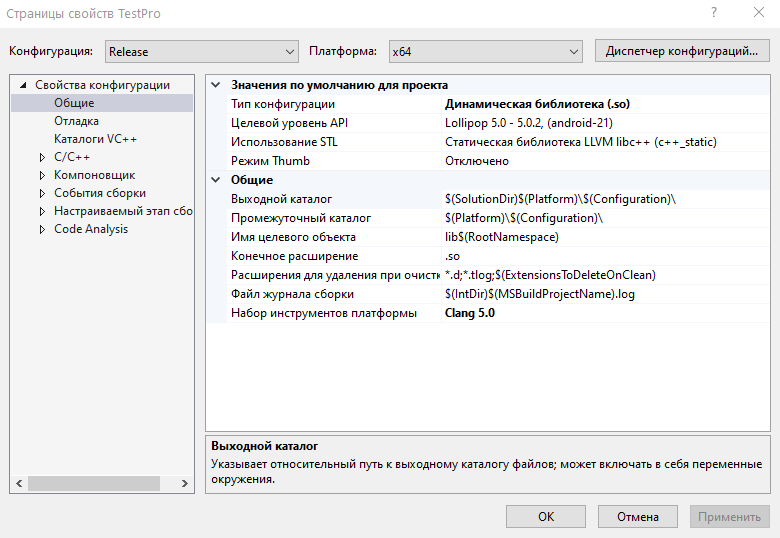
2. ARM64 **(для этой архитектуры целевой уровень АПИ не ниже 5.0)**



3. x86



4.x64 **(для этой архитектуры целевой уровень АПИ не ниже 5.0)**



Если все сделано правильно, то на этом этапе, проект уже должен собираться в пустую библиотеку.

Теперь, согласно поставленному тех. заданию следует реализовать 2 метода: запуск перехвата броадкаста и остановка перехвата.

1. Подготовка общих для интерфейса «IComponentBase» служебных методов и классов.

1.1. Сперва выковырим из файлов шаблона все, что относится к iOS. Мы пишем под Андроид.

1.2. Менеджер памяти 1с.

При вызове методов компоненты и передаче возвращаемых значений, которые не могут быть переданы полностью через стек, компонента должна выделять память с помощью функции AllocMemory, предоставляемую менеджером памяти. "1С:Предприятие" впоследствии освободит эту память с помощью функции FreeMemory.

Например, надо вернуть в платформу некую строку. Строка – это место в памяти с символами. Память под это дело должна быть зарезервирована (выделена) операционной системой. Есть заранее выделяемые области (выделенные при загрузке образа исполняемого файла или библиотеки в память) – статическая память и выделяемые по требованию – динамически. Также есть память процессорного стека. Стек – это удобно и быстро, но он, зараза, у каждого потока свой. Вот из-за этого приходится выделять глобальную память. Если об этом позаботилась вызвавшая нас процедура, то все ок: мы берем указатель и пишем туда все, что нужно. Но, если объем памяти, который нужен, определяем мы, то и выделять память придется нам, но функция, вызвавшая нас не всегда знает как, сколько и где эту память освобождать. Вот для этого и используем интерфейс менеджера памяти 1с, о котором платформа все знает.

Имплементация в проект менеджера памяти 1с.

Весь интерфейс работы с менеджером памяти 1С описан в файле «IMemoryManager.h».

Соответственно, этот заголовочный файл нужно подключить в «AddInNative.h»:

#include "../include/IMemoryManager.h"

Две точки вначале пути означают, что для поиска подключаемого файла следует перейти на один каталог вверх, а затем идти в «"/include/».

Далее, в «AddInNative.h» объявляем указатель на область памяти с менеджером.

IMemoryManager\* m\_iMemory;

В конструкторе «AddInNative» инициализируем менеджер пустым указателем.

AddInNative::AddInNative()

{

m\_iMemory = nullptr;

}

В функцию setMemManager платформа передает указатель на объект менеджера. Нам надо его сохранить:

bool AddInNative::setMemManager(void\* mem)

{

m\_iMemory = (IMemoryManager\*)mem;

return m\_iMemory != nullptr;

}

В методе «Done» установим в указатель ноль:

void AddInNative::Done()

{

m\_iMemory = nullptr;

}

1.3. Взаимодействие с платформой.

Чтобы как-то взаимодействовать с платформой (например, сгенерить внешнее событие), она в нашу библиотеку передает указатель на объект, реализующий интерфейс IAddInDefBase. Для взаимодействия с этим объектом, во внешней компоненте реализован (в шаблоне 1с его поставляет) одноименный класс «IAddInDefBase». Файл «AddInDefBase.h».

1.3.1.В «AddInNative.h» подключаем заголовочный файл «AddInDefBase.h».

1.3.2. В «AddInNative.h» объявляем переменную:

IAddInDefBaseEx\* m\_iConnect;

1.3.3. В конструкторе AddInNative инициализируем класс нулевым указателем

Hermes::Hermes() : m\_iConnect(nullptr), m\_iMemory(nullptr)

{

}

1.3.4. В экспортируемом методе «Init» сохраняем указатель на объект:

bool Hermes::Init(void\* pConnection)

{

m\_iConnect = (IAddInDefBaseEx\*)pConnection;

return m\_iConnect != nullptr;

}

1.3.5. В экспортируемом методе «Done» убиваем указатель (как-бы можно и не делать, но так принято):

void Hermes::Done()

{

m\_iConnect = nullptr;

m\_iMemory = nullptr;

}

1.4. Теперь реализуем все остальные процедуры, которые объявлены в интерфейсе IComponentBase и нужны для работы компоненты.

1.4.1. RegisterExtensionAs(WCHAR\_T\*\* wsLanguageExt)

В переменную wsExtName помещается наименование расширения. Память для строки выделяется объектом компоненты функцией AllocMemory менеджера памяти. "1С:Предприятие" освобождает эту память вызовом FreeMemory.

bool AddInNative::RegisterExtensionAs(WCHAR\_T\*\* wsExtensionName)

{

const wchar\_t\* wsExtension = g\_ComponentNameAddIn;

uint32\_t iActualSize = static\_cast<uint32\_t>(::wcslen(wsExtension) + 1);

if (m\_iMemory)

{

if (m\_iMemory->AllocMemory((void\*\*)wsExtensionName, iActualSize \* sizeof(WCHAR\_T)))

{

convToShortWchar(wsExtensionName, wsExtension, iActualSize);

return true;

}

}

return false;

}

1.4.2. Проверим, реализована-ли функция «GetClassNames». (Она возвращает платформе имя объекта компоненты).

const WCHAR\_T\* GetClassNames()

{

return s\_ComponentClass;

}

Почему возвращаем не **g\_kClassNames**, а **kClassNames**.

Дело в том, что платформа 1с работает со строками, в которых размер символа всегда 2 байта, но некоторые оси, в т.ч. в Андроид, может работать с символами размером 4 байта.

Поэтому на всякий случай, перед передачей указателей на память со строками платформе 1с, следует переводить все строки в 2-байтный формат. Это и делает обертка «WcharWrapper». Для динамически выделенной памяти, люди из 1с написали соответствующую функцию:

uint32\_t convToShortWchar(WCHAR\_T\*\* Dest, const wchar\_t\* Source, uint32\_t len)

{

if (!len)

len = static\_cast<uint32\_t>(::wcslen(Source) + 1);

if (!\*Dest)

\*Dest = new WCHAR\_T[len];

WCHAR\_T\* tmpShort = \*Dest;

wchar\_t\* tmpWChar = (wchar\_t\*)Source;

uint32\_t res = 0;

for (; len; --len, ++res, ++tmpWChar, ++tmpShort)

{

\*tmpShort = (WCHAR\_T)\*tmpWChar;

}

return res;

}

И обратно (т.е. из WCHAR\_T в wchar\_t):

uint32\_t convFromShortWchar(wchar\_t\*\* Dest, const WCHAR\_T\* Source, uint32\_t len)

{

if (!len)

len = getLenShortWcharStr(Source) + 1;

if (!\*Dest)

\*Dest = new wchar\_t[len];

wchar\_t\* tmpWChar = \*Dest;

WCHAR\_T\* tmpShort = (WCHAR\_T\*)Source;

uint32\_t res = 0;

for (; len; --len, ++res, ++tmpWChar, ++tmpShort)

{

\*tmpWChar = (wchar\_t)\*tmpShort;

}

return res;

}

Все эти и другие вспомогательные функции лежат в файлике «ConversionWchar.cpp».

1.4.3. Теперь, функцию, которая по имени прикладного метода будет возвращать иго ИД.

В классе AddInNative есть объявленная функция с прототипом «long AddInNative::FindMethod(const WCHAR\_T\* wsMethodName)». Она возвращает платформе 1с идентификатор метода.

Реализуем ее.

long AddInNative::FindMethod(const WCHAR\_T\* wsMethodName)

{

long plMethodNum = -1;

wchar\_t\* name = 0;

convFromShortWchar(&name, wsMethodName);

plMethodNum = findName(g\_MethodNames, name, eMethLast);

if (plMethodNum == -1)

plMethodNum = findName(g\_MethodNamesRu, name, eMethLast);

delete[] name;

return plMethodNum;

}

Также объявляем в AddInNative.h и реализуем AddInNative.cpp служебную функцию поиска номера строки в массиве.

long AddInNative::findName(const wchar\_t\* names[], const wchar\_t\* name, const uint32\_t size) const

{

long ret = -1;

for (uint32\_t i = 0; i < size; i++)

{

if (!wcscmp(names[i], name))

{

ret = i;

return ret;

}

}

return ret;

}

1.4.4. Реализуем функцию, с помощью которой платформа сможет по полученному ранее ИД метода получить его имя.

Для этого компонента должна реализовать функцию «GetMethodName»:

const WCHAR\_T\* Hermes::GetMethodName(const long lMethodNum, const long lMethodAlias)

{

if (lMethodNum >= eMethLast)

return NULL;

wchar\_t \*wsCurrentName = NULL;

WCHAR\_T \*wsMethodName = NULL;

switch (lMethodAlias)

{

case 0: // First language (english)

wsCurrentName = (wchar\_t\*)g\_MethodNames[lMethodNum];

break;

case 1: // Second language (local)

wsCurrentName = (wchar\_t\*)g\_MethodNamesRu[lMethodNum];

break;

default:

return 0;

}

uint32\_t iActualSize = static\_cast<uint32\_t>(wcslen(wsCurrentName) + 1);

if (m\_iMemory && wsCurrentName)

{

if (m\_iMemory->AllocMemory((void\*\*)&wsMethodName, iActualSize \* sizeof(WCHAR\_T)))

convToShortWchar(&wsMethodName, wsCurrentName, iActualSize);

}

return wsMethodName;

}

Здесь нам и пригождается Менеджер памяти 1с (строка с именем файла возвращается в платформу, а размер строки заранее платформе не известен).

1.5. Реализуем функцию «GetNParams» (она по номеру метода из перечисления «Methods») говорит 1с сколько входных параметров должно быть у метода.

long AddInNative::GetNParams(const long lMethodNum)

{

switch (lMethodNum)

{

case eMethStartBroadcastsInterception:

return 1;

case eMethStopBroadcastsInterception:

return 0;

default:

break;

}

return 0;

}

1.6. Реализуем функцию «HasRetVal» (она по номеру метода из перечисления «Methods») говорит 1с возвращает-ли метод значение (т.е. а не функция-ли метод).

bool AddInNative::HasRetVal(const long lMethodNum)

{

switch (lMethodNum)

{

case eMethStartBroadcastsInterception:

return true;

case eMethStopBroadcastsInterception:

return true;

default:

break;

}

return false;

}

1.7. Теперь придумаем нашей компоненте имя объекта, допустим «TestPro».

1.7.1. Лезем в файл AddInNative.cpp.

Вверху ищем константы:

static const wchar\_t g\_ComponentNameAddIn[] = L"TestPro";

static WcharWrapper s\_ComponentClass(g\_ComponentNameAddIn);

Это будет статическая константа, т.е. она не может быть изменена кодом и память под нее выделяется в момент загрузки образа библиотеки в оперативную память. Префикс «L» перед строкой означает, что строка будет иметь тип «wchar\_t».

Это та строка, которая будет указываться в 1с, при создании объекта компоненты: Новый("AddIn.<псевдоним>.<объект>").

(Псевдоним – это строка,, которую указываем во 2м параметре метода «ПодключитьВнешнююКомпоненту»).

1.7.2. Теперь объявляем наши функции, согласно методологии разработки Native-компонент для платформы 1с.

Идем в файл «AddInNative.h» и объявляем там несколько статических константных массивов строк с именами методов – на основном и дополнительном языках:

static const wchar\_t\* g\_MethodNames[] =

{

L"StartBroadcastsInterception",

L"StopBroadcastsInterception"

};

static const wchar\_t\* g\_MethodNamesRu[] =

{

L"ЗапуститьПерехватБроадкастов",

L"ОстановитьПерехватБроадкастов"

};

В этом-же файле, в перечисление добавим наши методы:

enum Methods

{

eMethStartBroadcastsInterception,

eMethStopBroadcastsInterception,

eMethLast // Always last

};

2. Теперь будем заниматься полезной нагрузкой.

Вообще, вся полезная нагрузка может быть реализована и в рамках класса «AddInNative», но это вредно с точки зрения ООП, и вообще - не путь джедая. Поэтому вынесем полезную нагрузку в отдельный класс. Назовем его ну, Elvis, например.

В нем будем усердно работать со строками std::wstring и поэтому привяжем пространство имен std к этому классу. Ну и включим заголовки iostream и файла заголовков в проект:

#include <iostream>

#include "Elvis.h"

Using namespace std;

Реализуем конструктор, деструктор класса.

Они нам пригодятся для реализации модели в рамках ооп. Ну, проще говоря, так принято.

Elvis.cpp:

Elvis::Elvis()

{

jClass = nullptr;

jObClass = nullptr;

};

Elvis::~Elvis()

{

}

Elvis.h:

#pragma once

#include <iostream>

#include "../jni/jnienv.h"

using std::wstring;

class Elvis

{

private:

jobject jObClass;

jclass jClass;

public:

Elvis();

~Elvis();

};

Две переменные jClass и jObClass нам понадобятся позже. Пока просто объявим их в файле заголовков и инициализируем нулями.

Согласно нашему техническому заданию, нам нужно реализовать как-то реализовать перехват системных броадкастов. Сделать это можно либо, используя какие-нибудь мощные с++ фреймворки, вроде QT, либо реализовать некий класс на java и как-то приспособиться передавать туда данные и получать что-то обратно. Поскольку тащить огромный фреймворк в свою библиотеку только ради перехвата броадкаста – дурной тон (да и с лицензиями на QT там разбираться надо), то 1с предложила вариант динамической линковки нашей библиотеки на с++ к другому приложению, написанному на java.

Если объяснять на пальцах, то работает так:

В zip-архив внешней компоненты помещается 2 файла: наша динамическая библиотека «TestPro.so» (собственно внешняя компонента) и пакет Андроид-приложения без активити «TestPro.apk».

В коде 1с, когда мы вызываем установку внешней компоненты, то платформа 1с динамически линкует к себе нашу библиотеку (в этот момент пакет Андроид-приложения «TestPro.apk» просто распаковывается в директорию КЭШа 1с и лежит там как файла, рядом с файлом библиотеки «TestPro.so». С ним ничего не происходит).

При линковке 1с запускает все интерфейсные методы, которые реализованы в нашей библиотеке, согласно интерфейсу «IComponentBase».

После этого мы можем вызвать в коде 1с метод старта перехвата броадкаста. Назовем его «СтартБК(jsonString js)».

Тут начинается самое интересное. Платформа 1с предоставляет ссылку на объект JNI (Java Native Interface) и ссылку на виртуальную машину Джава.

С помощью JNI наша библиотека, вызвав соответствующие методы JNC, может найти реализованный в пакете «TestPro.apk» Андроид-приложения соответствующий класс, создать в памяти объект этого класса и подергать из него нужные функции.

Так, прикольно, ну и что…

Вот запустили мы класс из нашего Андроид-приложения (фактически запускается оно в своем потоке, никак не связанном с платформой 1с), вот поймали мы в этом приложении броадкаст, но нам надо как-то передать данные из него в 1с, а потоки разные. Чтобы решить эту задачу, при инициализации Андроид-класса в него передается указатель на объект платформы 1с (IAddInDefBaseEx\* cnn) как (jlong)cnn. На стороне Андроид-класса этот указатель просто хранится, как переменная, до момента, когда нам нужно что-то вернуть в 1с. При наступлении такого момента, Андроид-приложуха производит Java Native Call: Приложение динамически линкует к себе нашу бибилотеку (еще раз, независимо от 1с!!! – получается второй экземпляр) и дергает в ней специально заточенную под JNC процедуру (про тщательную обработку напильником процедуры под JNC - далее). Но опять-же область памяти, в которую грузится второй экземпляр образа нашей динамической библиотеки (в потоке Андроид-приложения) никак не связана с 1с и областью памяти 1с, в которую платформа 1с загрузила первый экземпляр нашей библиотеки. (Это разные потоки вообще.) Но у нас есть ссылка на объект платформы «(jlong)cnn». Вот эту ссылку мы обязательно передаем, как параметр, при JNC. Когда мы оказываемся во втором экземпляре нашей библиотеки, то мы можем использовать переданный нам указатель чтобы пнуть платформу 1с в любое место, которое нам позволяет интерфейс класса “IAddInDefBaseEx”, например, вызвать внешнее событие:

IAddInDefBaseEx\* pAddIn = (IAddInDefBaseEx\*)pObject;

pAddIn->ExternalEvent(s\_EventSource, s\_EventName, WCHART);

Таким образом, при разработке динамической библиотеки, включаем в нее области, которые будут исполнятся только в потоке 1с и которые будут исполнятся только в потоке JavaVM.

Это в стихах. Теперь проза.

2.1. В класс Elvis добавляем процедуру «Initialize(IAddInDefBaseEx\* cnn)».

В ней создаем и инициализируем основной рабочий класс из приложения «TestPro.apk»:

Elvis.cpp:

void Elvis::Initialize(IAddInDefBaseEx \*cnn) //В процедуру передается указатель на объект платформы 1с.

{

if (!jObClass)

{

IAndroidComponentHelper\* helper = (IAndroidComponentHelper\*)cnn->GetInterface(eIAndroidComponentHelper);

//1с предоставляет нам некий интерфейс, который позволяет найти наш пакет, загрузить в память все его классы и запустить под него свой поток.

if (helper)

{

WCHAR\_T\* WCHART = nullptr;

convToShortWchar(&WCHART, DREAMCATCHER\_CLASS\_NAME); //Все интерфейсы, предоставляемые 1с работают со строками из двухбайтных символов.

jclass locClass = helper->FindClass(WCHART); //Получаем указатель на класс в пакете.

delete[] WCHART; //Процедура convToShortWchar выделяет память в хипе для новой строки - память нужно освободить, иначе будет утечка.

WCHART = nullptr;

if (locClass)

{

JNIEnv\* env = getJniEnv(); //Получаем указатель на JNI для вызова функций JNI

jClass = static\_cast<jclass>(env->NewGlobalRef(locClass)); //Локальный указатель на класс надо преобразовать в глобальный.

env->DeleteLocalRef(locClass); //И удалить локальный указатель (так рекомендует документация по JNI).

jobject activivty = helper->GetActivity(); //Получаем локальный указатель на активити 1с.

if (activivty)

{

jmethodID jmID = env->GetMethodID(jClass, "<init>", "(Landroid/app/Activity;J)V"); //Ищем в классе процедуру инициализации.

//Поиск происходит по описанию прототипа. Прототип описывается так:

//Имя процедуры (имя предопределенной процедуры init всегда указывается в квадратных скобках),

//третим аргументом передается список ТИПОВ параметров функции - в скобках, а за скобками - тип возвращаемого параметра (V - void - процедура не возвращает значение).

jobject locJO = env->NewObject(jClass, jmID, activivty, (jlong)cnn);

//Так в JNI инициализируется новый объект из класса. Последние 2 параметра - прикладные. Т.е. мы туда можем передать что хотим.

//Фактически - на стороне андроид-приложения вызывается процедура "init"

//Получаем локальный указатель на загруженный в память, инициализированный и готовый к работе класс "DreamCatcher"

env->DeleteLocalRef(activivty); //Гасим локальный указатель.

if (locJO)

{

jObClass = static\_cast<jobject>(env->NewGlobalRef(locJO));

env->DeleteLocalRef(locJO); //Удаляем локальный указатель

//На стороне приложения ищем и вызываем метод "show".

jmID = env->GetMethodID(jClass, "show", "()V");

env->CallVoidMethod(jObClass, jmID);

}

}

}

}

}

}

2.2. В деструктор класса Elvis добавляем код удаления глобальных ссылок:

Elvis::~Elvis()

{

JNIEnv\* je = getJniEnv();

if (jClass)

{

je->DeleteGlobalRef(jClass);

}

if (jObClass)

{

je->DeleteGlobalRef(jObClass);

}

}

2.3. Добавляем процедуры запуска перехвата броадкаста и остановки перехвата перехвата.

void Elvis::StartBroadcast(wstring \*std\_wstr\_filter, IAddInDefBaseEx\* cnn)

{

if (jObClass)

{

JNIEnv\* env = getJniEnv(); //получаем указатель на JNI

jmethodID jmID = env->GetMethodID(jClass, "start", "(Ljava/lang/String;)V"); //В созданном при инициализации классе "Dreamcatcher" ищем метод "старт"

jstring jstring\_ = ToJniString(std\_wstr\_filter); //JNI работает со строками jstring - конвертируем wstring в jstring.

env->CallVoidMethod(jObClass, jmID, jstring\_); //Вызываем метод на стороне Андроид-приложения.

env->DeleteLocalRef(jstring\_); //Обязательно удаляем локальную ссылку.

}

}

void Elvis::StopBroadcast()

{

if (jObClass)

{

JNIEnv\* env = getJniEnv();

jmethodID jmID = env->GetMethodID(jClass, "stop", "()V");

env->CallVoidMethod(jObClass, jmID);

}

}

2.4. Реализуем экспортную процедуру, которую будет вызывать через JNI-call андроид-приложение. В нее будут передаваться данные, которые нужно дальше послать в процедуру внешнего события 1с.

static const wchar\_t g\_EventSource[] = L"TestPro";

static const wchar\_t g\_EventName[] = L"BroadcastCatched";

static WcharWrapper s\_EventSource(g\_EventSource); //Это обертка, которая делает из wchar\_t WCHAR\_T (двухбайтную строку из четырехбайтной).

static WcharWrapper s\_EventName(g\_EventName); //Это обертка, которая делает из wchar\_t WCHAR\_T (двухбайтную строку из четырехбайтной).

//https://www3.ntu.edu.sg/home/ehchua/programming/java/javanativeinterface.html

// name of function built according to Java native call

//

extern "C" JNIEXPORT void JNICALL Java\_ru\_coolclever\_dreamcatcher\_catcher\_OnBroadcastCatched(JNIEnv\* env, jclass jClass, jlong pObject, jstring inJNIStr)

{

wstring std\_wstring = ToWStringJni(inJNIStr); //jstring to wstring

WCHAR\_T\* WCHART = nullptr;

convToShortWchar(&WCHART, std\_wstring.c\_str()); //wstring to WCHAR\_T

IAddInDefBaseEx\* pAddIn = (IAddInDefBaseEx\*)pObject; //преобразуем jlong в указатель на IAddInDefBaseEx

pAddIn->ExternalEvent(s\_EventSource, s\_EventName, WCHART); //Вызываем метод интерфейса IAddInDefBaseEx - это инициирует внешнее событие в платформе 1с.

delete[] WCHART; //Освобождаем память, т.к. в функции convToShortWchar она выделяется из хипа потока методом new.

}

Функция не принадлежит ни одному из классов (экспортную не можно запихать в класс). Соответственно, все переменные, которые нужны для ее работы тоже объявляем, как глобальные, константные и статические.

Про то, как дорабатывать напильником процедуры для JNIC расписано здесь: <https://www3.ntu.edu.sg/home/ehchua/programming/java/javanativeinterface.html>.

Есть как-бы все, но много букав, по-этому объясню основное.

Ключ **extern** – делает функцию экспортной, т.е. она при сборке библиотеки объявляется в таблице экспортов библиотеки и ее можно ~~засплайсить~~ после линковки дергать из своего кода.

"C" – указание на конвенцию о передаче параметров.

JNIEXPORT – макрос, содержащий все директивы компилятора, необходимые для обеспечения правильного экспорта функции для JNI в таблицу экспортов динамической бибилотеки.

JNICALL – макрос, содержащий все директивы компилятора, необходимые для обеспечения того, чтобы данная функция обрабатывалась с надлежащим соглашением о вызове.

Чтобы все эти макросы отработали правильно, имя функции ОБЯЗАТЕЛЬНО строится по такой схеме:

[Java]\_[имя пакета приложения, из которого будет происходить jni-call, где точки заменены на нижние подчеркивания (ru.androidapp.testpro--> ru\_androidapp\_testpro)]\_[имя класса, из которого будет идти jni-call]\_[имя процедуры, из которой будет происходить jni-call].

Почему все так – не спрашивайте. Логика тут очень субъективная.

2.5. Теперь надо реализовать обработку вызова 2х наших функции для 1с.

Благодаря реализации процедур «findName», «FindMethod», «GetMethodName», «GetNParams» (есть-ли у метода входные параметры), «HasRetVal» (возвращает-ли метод значение (т.е. функция он или процедура)) платформа знает все что нужно о реализации наших функциях.

Оба наших метода возвращают значение (строку «ok\_en» - успех, все остальное – ошибка и ее описание), значит они функции и при их вызове в коде 1с, фактически будет вызываться функция «CallAsFunc»:

//---------------------------------------------------------------------------//

bool AddInNative::CallAsFunc(const long lMethodNum,

tVariant\* pvarRetValue, tVariant\* paParams, const long lSizeArray)

{

switch (lMethodNum)

{

case eMethStartBroadcastsInterception: //Если вызвана функция "ЗапуститьПерехватБроадкастов"

{

if (!lSizeArray || !paParams) //Если массив параметров нулевого размера или указатель пустой, то вызываем ошибку (значит параметров не хватает).

//В этом случае на стороне 1с вызывается исключение.

return false;

if (!isBKStarted) //Если броадкаст не запущен, то запускаем. (в заголовочном файле объявили приватную переменную)

{

wstring std\_wstr(paParams->pwstrVal, paParams->pwstrVal + paParams->strLen);

//инициализируем новую строку wstring. Используется такой конструктор: указатель на память с началом строки wchar\_t, указатель на память, где конец строки wchar\_t.

//В заголовочном файле объявили приватную переменную (m\_Elvis).

//При инициализации класса AddInNative сразу инициализируется и переменная "m\_Elvis" - вызывается конструктор класса "Elvis".

//После этого класс в памяти гарантировано и мы можем дергать его за методы.

m\_Elvis.Initialize(m\_iConnect);

m\_Elvis.StartBroadcast(&std\_wstr, m\_iConnect);

//Эта процедура идет в примере к внешней компоненте. Она сразу преобразует wchar\_t в tVariant и устанавливает ему 1сный тип "строка". Сладко...

ToV8String(L"ok\_en", pvarRetValue, m\_iMemory);

isBKStarted = true;

}

else //Если броадкаст уже запущен, то возвращаемся без запуска, а в 1с приходит результат функции - строка, в которой содержится описание ошибки.

{

ToV8String(L"Перехватчик уже запущен.", pvarRetValue, m\_iMemory);

return true;

}

}

break;

case eMethStopBroadcastsInterception: //Если вызвана функция "ОстановитьПерехватБроадкастов"

{

if (isBKStarted)

{

m\_Elvis.StopBroadcast();

ToV8String(L"ok\_en", pvarRetValue, m\_iMemory);

isBKStarted = false;

return true;

}

else

{

ToV8String(L"Перехватчик не запущен.", pvarRetValue, m\_iMemory);

return true;

}

}

break;

default:

break;

}

return true;

}

3. Сборка проекта «TestPro.apk».

4. Компоновка архива внешней компоненты.

1. https://infostart.ru/1c/articles/987286/ [↑](#footnote-ref-1)
2. https://infostart.ru/1c/articles/987286/ [↑](#footnote-ref-2)