**УДК 004.423**

**Связывание кода компилируемых языков с кодом интерпретируемых языков**

**Аннотация.** В статье рассматривается задача связывания кода, написанного на компилируемых языках таких как C и C++ с интерпретируемыми или исполняющимися на виртуальной машине языками такими как Java\Kotlin. Для решения задачи будут использованы такие инструменты как автоматический генератор интерфейсов и механизм для запуска кода под управлением виртуальной машины Java\Kotlin JVM, который дает возможность вызова функции С и С++ из программы на Java\Kotlin, и наоборот. Задача является актуальной по нескольким критериям: скорость исполнения кода, написанного на C и C++, как минимум в 6 раз быстрее кода на Java [4], что является существенным для приложений, работающих в «реальном времени», также возникает необходимость использования библиотек, которые написаны только на C и C++ и являются сложными для переноса на другую платформу, кроме этого языки С и С++ имеют более широкий доступ к системным функциям, нежели Java.

**Ключевые слова:** JNI, JVM, SWIG, Java, Kotlin, C, C++, GCC, wrappers, native java, интерфейсы, библиотеки, кроссплатформенный код.

**Введение**

Используя Java Native Interface (далее JNI) [2], можно вызывать код С/С++ из Java и, наоборот, из программы на С\С++, можно создавать Java объекты и вызывать Java методы. Использование JNI обычно ведет к потере платформо-независимости Java-кода. Использование JNI позволяет совместить объектно-ориентированный подход Java и существующий системно-зависимый код на С\С++. Это является важным и необходимым условием к использованию Java при разработке компонентов, которые должны работать быстро и безотказно, к примеру компоненты серверов.

Есть несколько причин совместного использования С\С++ и Java: стандартные библиотеки Java не всегда поддерживают низкоуровневые системно-зависимые возможности, такие как работа напрямую с памятью или устройствами. Так же существует необходимость использования уже написанного кода на других языках или желание эффективно реализовать участок кода, для которого критично время исполнения. Эти причины, возможно, не очень существенны при разработке клиентских приложений, однако в случае серверных они становятся основными.

**Java Native Interface**

Взаимодействие кодов Java и С\С++ может осуществляться несколькими способами:

* С\С++ код получает управление непосредственно из Java путем вызова native метода.
* С\С++ код динамически загружает JVM с помощью Invocation API.

Для того чтобы передать управление С\С++ - коду из Java, необходимо создать собственный Java\Kotlin метод, сгенерировать с помощью Javah заголовочный файл для С\С++, разработать сами функции, в которые будет передаваться управление и данные, и поместить их в файл библиотеки.

Для того, чтобы создать собственный метод, нужно дописать специальный спецификатор native. Этот спецификатор показывает компилятору, что реализация данного метода будет представлена в библиотеке на C\C++.

Кроме описания собственного метода, в Java\Kotlin необходимо динамически загрузить библиотеку, содержащую С\С++ код реализации данного метода. Для этого в классе Java.Lang.System существует метод public static void loadLibrary**(**String LibName**)**, загружающий указанную библиотеку. Следующий пример демонстрирует описание собственного метода [3].

Создание native-метода в Java\Kotlin:

|  |
| --- |
| class MyClass **{**  static **{**  System**.**loadLibrary**(**"test"**);**  **}**  public native int nativeMethod**();**  **}** |

В приведенном примере nativeMethod**() –** собственный метод, и его С\С++ реализация находится в библиотеке test. Метод loadLibrary**()** вызывается в статическом блоке, что обеспечивает вызов этого метода после загрузки класса MyClass. Метод loadLibrary**()** преобразует передаваемый параметр, имя, в соответствии с тем, на какой платформе будет запущен код. В данном примере test преобразуется в libtest.so для Linux\Unix и test.dll для Windows. Для Windows DLL должна находиться либо в текущем каталоге процесса, либо в каталоге, содержащем исполняемый exe файл, либо в системном каталоге Windows или в каталогах, указанных в переменной окружения PATH. Для UNIX библиотечный файл должен находиться либо в текущем каталоге процесса, либо в подкаталоге Lib основного каталога Java\Kotlin, либо в каталогах, перечисленных в переменной окружения LD\_LIBRARY\_PATH. Если указанную библиотеку найти не удается, метод loadLibrary**()** генерирует исключение Java.Lang.UnsatisfitedLinkError.

Для более надежной работы с собственными методами можно использовать, к примеру, следующий код:

|  |
| --- |
| pulibc class App **{**  pulibc static void main**(**String args**)** **{**  MyClass myClassInstance **=** **new** MyClass**();**  **try** **{**  myClassInstance**.**nativeMethod**();**  **}**  **catch** **(**UnsatisfiedLinkError e**)** **{**  System**.**out**.**println**(**"Method not found (" **+** e **+** ")"**);**  **}**  **}**  **}**    class MyClass **{**  static **{**  **try** **{**  System**.**loadLibrary**(**"test"**);**  **}**  **catch** **(**UnsatisfiedLinkError e**)** **{**  System**.**out**.**println**(**"Library not found (" **+** e **+** ")"**);**  **}**  **}**    native void nativeMethod**();**  **}** |

Компиляция программ, содержащих собственные методы, ничем не отличается от компиляции обычных программ. Например, если записать предыдущий пример в файл с именем MyClass.java, то для его компиляции необходимо выполнить следующую команду:

|  |
| --- |
| # javac MyClass.java |

**Создание заголовочного файла**

Создание С\С++ кода необходимо начинать с создания заголовочного файла. Его можно написать вручную или воспользоваться утилитой Javah. Второй путь предпочтительней, так как допускает меньшее количество ошибок. При обращении к утилите Javah указывается имя класса и параметр -JNI. Без него Javah будет генерировать файл в формате JDK 1.0 NI. Имя класса представляет собой полное квалифицированное имя класса.

Пример:

|  |
| --- |
| #javah -JNI Java.Lang.Runtime |

Перед использованием утилиты Javah соответствующий Java\Kotlin класс должен быть скомпилирован в class-файл. Утилита Javah анализирует class-файл и строит заголовочный файл, в котором перечислены объявления С\С++ функций, представляющих реализации соответствующих собственных методов.

Например, если выполнить следующие команды:

|  |
| --- |
| # javac MyClass.java  # javah -jni MyClass |

то javah сгенерирует следующий файл MyClass.h:

|  |
| --- |
| /\* Do not edit this file - it is machine generated \*/  #include <jni.h>  /\* Header for class MyClass \*/  #ifndef \_included\_MyClass  #define \_included\_MyClass  #ifdef \_ \_cplusplus  extern "c" **{**  #endif  /\*  \* class: MyClass  \* method: nativeMethod  \* signature: ()  \*/  jniexport void jnicall Java\_MyClass\_nativeMethod**(**jnienv **\*,** jobject**);**  #ifdef \_ \_cplusplus  **}**  #endif  #endif |

Данный файл можно создать вручную или с помощью утилиты Javah, далее будет продемонстрирован более удобный способ генерации заголовочного файла.

Директива препроцессора #include <jni.h> включает файл jni.h в котором находятся все необходимые объявления типов и функций для реализации собственного метода.

Макросы jniexport и jnicall необходимы для платформы Windows, где они превращаются соответственно в \_\_declspec (dllexport) и \_\_stdcall и позволяют более эффективно строить DLL.

Имя С\С++ функции значительно отличается от имени собственного Java\Kotlin метода. Важным понятием при построении имени С\С++ функции и использовании JNI-функций является сигнатура метода (signature или method arguments signature).

**SWIG**

За основу связывания библиотек, написанных на C и C++, и проекта на Java\Kotlin был взят SWIG [1] (англ. simplified wrapper and interface generator), свободный инструмент для связывания программ и библиотек, написанных на языках C и C++, с интерпретируемыми или компилируемыми языками: Perl, Python, Ruby, PHP, Java, C#. Для использования SWIG необходимо создать файл \*.i с описанием экспортируемых функций; SWIG генерирует С\С++ код для связывания с нужным языком[1].

Пример интерфейсного файла (test.i):

|  |
| --- |
| **%**module**(**directors**=**"1"**)** test  **%**javaconst**(**1**);**  **%**include "enumtypeunsafe.swg"  **%**include "std\_vector.i"  **%**include "std\_string.i"  **%**include "stdint.i"  **%**include "cpointer.i"  **%{**  #include "Data.h"  #include "Types.h"  #include "Keys.h"  ………  **%}**  **%**feature**(**"director"**)** CryptoLayer**;**  **%**feature**(**"director"**)** Transport**;**  **namespace** std  **{**  **%**template**(**vector\_ca\_key**)** vector**<**Keys**::**ca\_key**>;**  **%**template**(**vector\_char**)** vector**<**char**>;**  **%**template**(**vector\_appinfo**)** vector**<**Txn**::**ApplInfo**>;**  **%**template**(**vector\_revocation**)** vector**<**RevocationInfo**>;**  **%**template**(**vector\_string**)** vector**<**std**::**string**>;**  **};**  **%**include "Data.h"  **%**include "Types.h"  **%**include "Keys.h"  **%**include "CryptoLayer.h"  **%**include "Errors.h"  **%**pointer\_class**(**CRYPTOGRAM\_TYPE**,** CryptogramType**)** |

В интерфейсном файле необходимо указать все заголовочные .h из которых будут извлечены методы и объекты для последующей обработки SWIG-ом. Так же для работы с некоторыми объектами языков С\С++, с которыми нельзя работать напрямую, необходимо сделать обертку в виде template**(output type\_name)** vector**<**input type\class**>**. Есть возможность создания указателей на класс pointer\_class**(**input class\type**,** output class\type**).**

Далее после написания интерфейсного файла необходимо выполнить следующую команду:

|  |
| --- |
| swig -java test.i |

После выполнения данной команды сгенерируются файлы \*.cxx и \*.java.

Далее необходимо скомпилировать исходный код библиотеки:

|  |
| --- |
| g++ -fPIC -c \*.cpp |

После этого скомпилировать файлы \*.cxx созданные SWIG с указанием, в нашем случае, путь к файлам Java - Java Development Kit.

|  |
| --- |
| g++ -fPIC -c test\_wrap.c –I …/jdk\*\*\*/include  -I …/ jdk\*\*\*/include/linux |

После выполнения данных команд, будут собраны объектных файла \*.o.

Последним шагом, слинкуем объектные файлы:

|  |
| --- |
| g++ -shared test\_wrap.o test.o -o libtest.so |

Итогом данных действий, будет являться общая библиотека (shared library) libtest.so.

Для того, чтобы использовать собранную библиотеку, нужно ее загрузить.

Как видно из примера, использование SWIG позволяет автоматизировать создание интерфейсных, связующих файлов, для последующего использования с помощью JNI.

Для примера использования библиотек С++ в Java, реализуем Решето Эратосфена — алгоритм нахождения всех простых чисел до некоторого целого числа N. Решето Эратосфена является хорошим способом оценки производительности. Сложность алгоритма порядка , временная сложность вычисления всех простых чисел меньше n аппроксимируется . Однако алгоритм имеет экспоненциальную временную сложность в отношении размера входных данных, что делает его псевдополиномиальным алгоритмом. Памяти же для базового алгоритма требуется .

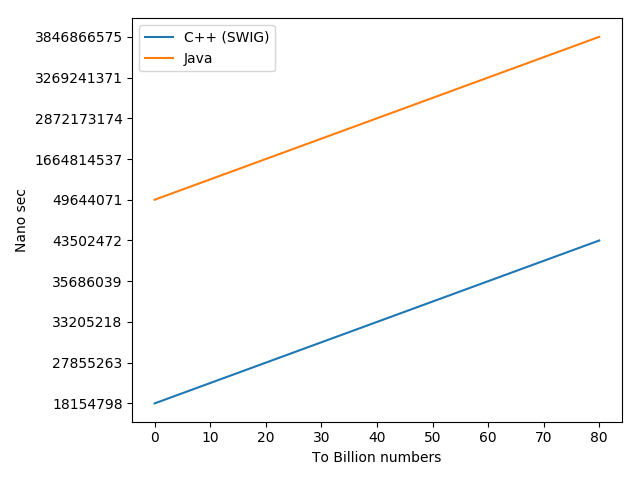


Рис. 1 Зависимость времени исполнения программы от максимального числа N.

Как можно увидеть на графике, в среднем время исполнения кода С++ на 2 порядка меньше чем на Java.

**Вывод:** представленный в статье метод связывания программ, написанных на С++ с Java\Kotlin, решает проблему использования уже написанных библиотек\кода на С\С++ без необходимости вручную писать JNI методы и связующую часть C\C++ кода.

**ЛИТЕРАТУРА**

1. Jesse Russell, Ronald Cohn - Swig, Book on Demand, 2012 - 168 c. ISBN 5512283324, 9785512283325.
2. The Java Native Interface: Programmer's Guide and Specification (The Java Series) 2015 - 318 с. ISBN 978-0201325775, 0201325772.
3. Java Native Interface: Programmer's Guide and Specification by Sheng Liang Prentice Hall PTR 1999 ISBN/ASIN: 0201325772 ISBN-13: 9780201325775
4. https://www.opennet.ru/opennews/art.shtml?num=30784
5. http://www.swig.org/

**Linking the code of compiled languages to the code of interpreted languages**

**Abstract.** The article considers the problem of linking code written in compiled languages ​​such as C and C ++ with interpreted or executable languages ​​on the virtual machine, such as Java\Kotlin. To solve the problem, tools such as the automatic interface generator and the mechanism for running code under the control of the Java\Kotlin JVM virtual machine will be used, which allows the C and C ++ functions to be called from the program in Java\Kotlin, and vice versa. The task is actual according to several criteria: the speed of execution of code written in C and C ++ is at least 6 times faster than Java code [4], which is essential for applications working in "real time", it also requires the use of libraries that written only in C and C ++ and are difficult to transfer to another platform, except that C and C ++ have more access to system functions than Java.

**Keywords:** JNI, JVM, SWIG, Java, Kotlin, C, C++, GCC, wrappers, native java, interfaces, shared libraries, cross platform code.