Integration of Kotlin Multiplatform Projects with Swift Package Manager Dependencies

Pavel Gromov
ITMO University
Saint Petersburg, Russia
pagrom27@gmail.com

Yaroslav Chernyshev
JetBrains
Saint Petersburg, Russia
yaroslav.chernyshev@jetbrains.com

***Abstract*—This paper will discuss cross-platform programming using the Kotlin language as an example. About what are the native ways of connecting dependencies to platforms of the Apple family and which of them are available in Kotlin Multiplatform. Looking at the Kotlin integration with the Cocoapods dependency manager and its drawbacks. Also I will offer my implementation for integrating Swift package dependencies in Kotlin.**

Keywords – Kotlin Multiplatform, cross-platform programming, dependency manager, Swift Package Manager, Cocoapods, Gradle, Gradle plugin.

I. Introduction

A. Cross-platform programming

Cross-platform is the ability of software to work across multiple hardware platforms and operating systems. In the software development industry, cross-platform programming is gaining popularity, as it allows you to reuse the source code written once in applications compiled for different platforms, at the same time, which can speed up product development, as well as reduce production costs.

One of the software development industries in which cross-platform programming technologies are becoming widespread is mobile development. A significant share of the mobile device market is made up of devices running Android and iOS mobile operating systems. Development for these devices is carried out in the corresponding programming languages: for Android devices, the Android Software Development Kit (SDK) is used, which is compatible with the Java and Kotlin programming languages. Development of applications for devices running iOS operating systems is performed in the Swift and Objective-C programming languages. The classical approach to mobile development assumes that to develop an application for these platforms, it is necessary to duplicate the source code of the program logic in the programming languages of each platform.

The use of cross-platform development technologies implies, first of all, the ability to describe the general application logic in one programming language and use it on different platforms. At the same time, the elements of the graphical interface of mobile applications are often described using platform-native tools.

One of the popular solutions for cross-platform development is the Kotlin Mutliplatform technology from JetBrains [1], which includes Kotlin Multiplatform Mobile - a plugin for the Android Studio [2] development environment. Kotlin Multiplatform technology is successfully used in the development of cross-platform applications by such companies as Yandex, Netflix, VMWare, Autodesk [3].

B. Kotlin Multiplatform

Kotlin Multiplatform technology allows you to write Kotlin code for multiple platforms. The main feature is that this technology does not fully follow the ”Write once, run everywhere” principle. That is, the common part of the code is taken out separately, and is used by each platform, and the platform-dependent code remains in the platform module. In this case, the final code for each platform is assembled from a common part and platform-dependent. This flexibility is achieved thanks to the language compilers: *Kotlin/JVM*, *Kotlin/Native*, *Kotlin/JS*.

Thus, Kotlin Mutliplatform provides:

- access to platform-specific API;
- IDE support;
- support from the build system (*Gradle* [4]).

To support the general part of the code, the constructions *expect* and *actual* were introduced into the language. The *expect* keyword is a declaration that only has a signature but no definitions. Unlike interfaces, this declaration allows the code to be used as if it were a function or a class. In this case, the developer is obliged to implement the *expect* declaration on each of the platforms. The implementation is carried out using the service word *actual*. Thus, Kotlin Mutliplatform provides platform-specific API access.

Libraries are accessed by a build system that works with a dependency manager. Kotlin Multiplatform uses Gradle as its build system. Dependencies for different platforms are collected using different package managers, for example, for Kotlin/JVM – this is the maven repository that uses gradle, for Kotlin/JS – NPM, and for Kotlin/Native (for platforms of the Apple family) – Cocoapods.

II. Integration of Kotlin Multiplatform projects with Cocoapods dependency manager

The ability to integrate Kotlin Multiplatform projects with a mobile development environment for iOS devices is provided by the following tools:

- a Kotlin / Native compiler that generates an Objective-C/Swift compatible artifact [5];
- a set of basic native libraries converted to Kotlin Library using the *cinterop* utility [6];
- plugin for automatic Gradle build system, which implements the integration of Kotlin Multiplatform projects with CocoaPods dependencies [7].

To add dependencies to the Kotlin code, a DSL for build.gradle file was developed that allows you to customize the connection of dependencies. In addition, Gradle tasks are also implemented, each of which implements the dependency connection logic. Plugin build a graph, the vertices of which are the gradle tasks themselves. To connect the dependency, you need to completely traverse the graph. The edges of the graph describe the dependencies of one task on another, and often the result of the work of one task is passed to the next as input. During configuration, some vertices of the graph can be deleted or ignored.

An excerpt of the project configuration, in which the popular library for working with the AFNetworking library is connected to the targets corresponding to the iOS simulator and the device.

```
kotlin{
    android()
    ios()

    cocoapods{
        summary = "CocoaPods library"
        homepage = ".../JetBrains/kotlin"
        pod("AFNetworking"){
            version = "~> 4.0.0"
        }
    }
}
```

When the *podImport* task is called, the *podspec* configuration file is created, transitive dependencies are resolved, the assembly of intermediate artifacts and the generation of Kotlin Library from them, and then the generation of the final Kotlin Multiplatform project framework. This framework can be connected to an Xcode project via *Podfile* and use its declarations in the source Swift or Objective-C as well as declarations from other dependencies.

Cocoapods plugin dependency manager has a number of significant limitations, including:

- inability to connect more than 1 framework to an Xcode project, obtained from a Kotlin Multiplatform project;
- lack of a mechanism for publishing a framework obtained from a Kotlin Multiplatform project, and resolving a dependency on it in an arbitrary project;
- Lack of transitivity on pod-dependencies when there is a dependency on a Kotlin Multiplatform project module containing such a dependency.

A. Swift Package Manager

Swift Package Manager is Apple's official dependency manager announced in 2015. It is an Xcode tool and is thus updated along with Xcode updates. To connect dependencies, the *Package.swift* file is described in which the Package instance is initialized. Because the Swift Package Manager is an Xcode tool, dependency connection can be done in the Xcode GUI. Basically, open sources with a codebase (git repositories) or paths to local source files are used as dependencies.

The configuration file *Package.swift*, by analogy with the considered dependency managers, is also located in the root directory of the Xcode project.

```
import PackageDescription

let package = Package(
    name: "Example",

    dependencies: [
        .package(url: ".../Alamofire/Alamofire")
    ],

    targets: [
        .target(name: "Example")
    ]
)
```

In 2019 at WWDC, Apple announced a new data packing format - *XCframework* [8]. This is a binary format that has a number of advantages over the *framework*, among which the package of dependencies for all target platforms and architectures at once is guaranteed by the tool. This made it unnecessary to create a *fat-framework* as a binary artifact for a set of platforms.

III. Kotlin Integration with Swift Package Manager

According to the results of a survey of Kotlin Multiplatform users published on the JetBrains [9] blog, it was found that 52.6% of respondents use Cocoapods to integrate Kotlin Multiplatform projects with native dependency managers. However, 25% are interested in integrating Swift Package Manager.

Integration of Swift Package Manager with Kotlin Multiplatform projects implies interaction in both directions, as a result of which 2 global tasks arise:

- integration of Swift Package dependencies into the Kotlin multiplatform project;
- implement the ability to publish Kotlin Multiplatform projects as Swift Package Manager dependencies.

A. Connecting Swift Package Manager dependencies in Kotlin projects

Swift Package Manager allows you to add dependencies to an Xcode project. In the first task, it is supposed to use these capabilities to resolve dependencies and integrate them into a Kotlin Multiplatform project. Thanks to the implementation of this functionality in Kotlin Mutliplatform projects, it became possible to use ready-made libraries that were already written in Objective-C. With the advent of Kotlin/Native compatibility with libraries in the Swift language, there will also be support for connecting these libraries.

To complete this task, you need to provide the ability to configure dependencies in the Gradle configuration files of the Kotlin Mutliplatform project, corresponding to the settings of the Swift Package Manager.

```
kotlin{
  iosArm64()

  spm{
    ios("10"){
      dependencies{
        `package`(
           ".../AFNetworking/AFNetworking.git",
           "4.0.0",
           "AFNetworking"
        )
      }
    }
  }
}
```

In addition, native dependencies resolved using the Swift Package Manager toolkit must be integrated into the Kotlin Mutliplatform project. To solve this problem, you need to build the binary *framework* dependencies and convert it to Kotlin Library using the *cinterop* utility.

B. Connecting Kotlin libraries in XCode projects as Swift Package Manager dependency

To solve the problem of connecting a Kotlin Mutliplatform project as a native dependency, you need to convert the source code in the Kotlin language into a binary artifact suitable for the Xcode project. One of the candidates for such a binary artifact is the *framework*. Kotlin Multiplatform includes a Kotlin/Native compiler that implements the logic for converting Kotlin source code into an executable binary *framework* containing LLVM IR. This format is compatible with Objective-C and Swift source code, and therefore is suitable for the implementation of this task within the same platform. Since, in general, the Swift Package generated by a Kotlin Multiplatform project can include an arbitrary set of platforms, the preferred format is the universal *Xcframework*. In the course of this work, the task of implementing the functionality of converting a set of platform *frameworks* into a universal *Xcframework* was solved.

IV. Implementation features

As described earlier, Kotlin Multiplatform technology is based on the Gradle automated build system, extending its functionality through the implementation of a set of plugins. In this regard, it was decided to implement the functionality for connecting native Swift Package Manager dependencies in a Kotlin Multiplatform project, as well as building and connecting native Kotlin Multiplatform artifacts to an Xcode project through the Swift Package Manager configuration file by developing a plugin for Gradle - Kotlin SPM Plugin.

The main entities of the Kotlin SPM Plugin are: the plugin extension *SpmExtension*, which encapsulates the configuration logic and implements the DSL described above, as well as a set of assembly execution units, which in Gradle terms are usually called tasks - *Task*.

The *SpmExtension* extension is implemented in such a way that it is understandable for both experienced iOS developers who use the Swift Package Manager in their work environment, and those developers who are not familiar with this toolkit. This effect is achieved through the implementation of DSL syntax and a set of abstractions that are similar in interface and functionality to the corresponding entities in the Swift Package Manager. The public interface is shown in Fig. 1.

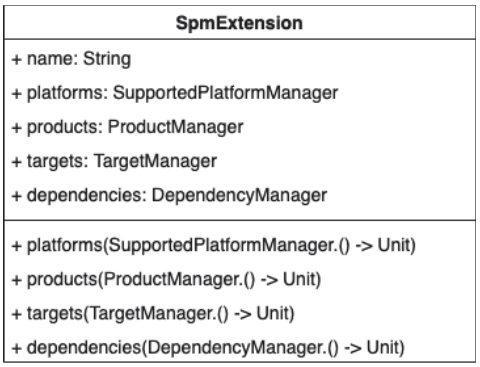

Fig. 1. UML diagram of the SpmExtension class

To solve the problem of connecting Swift Package Manager dependencies in the Kotlin Multiplatform project using the above DSL, a set of tasks has been implemented, which, according to their functions, can be divided into 5 groups, performed in stages one after another:

1) Generate manifests *Package.swift*;
2) Resolution of native dependencies;
3) Build native dependencies in the *framework*;
4) Converting framework dependencies to klib;
5) Connecting the received klibs to the Kotlin Multiplatform project.

The Gradle system, when building the project build process, creates and configures a directed acyclic task graph. This means that the assembly sets up a set of tasks and ties them together based on dependencies to create a directed acyclic graph. After creating a graph of tasks, Gradle determines in which order to run tasks, and then proceeds to perform them. At the same time, Gradle introduces a number of abstractions for lazy computation, such as Provider, Property and ”live” collections, thus avoiding redundant and premature computation for a specific use case. Following the above, the implementation of the solution to the task of connecting the Swift Package Manager dependencies in the Kotlin Multiplatform project boils down to building and executing the following task graph (Fig. 2):

From the point of view of units of performance, the tasks described in the graph perform the following functions:

- *generateSpmManifest* - generates correct Swift Package Manager manifest;

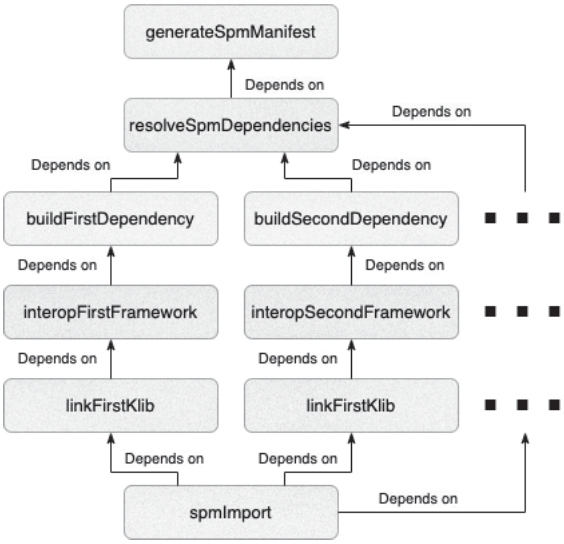

Fig. 2. Gradle task graph

- *resolveSpmDependencies* - resolves native project dependencies by loading from version control system or copying native dependencies from the file system for later build;
- *build[LibraryName]Dependency* - using the command line interface of the Xcode utility, it builds the binary artifact (framework) of the resolved dependency and generates the necessary meta information about it;
- *interop[LibraryName]Framework* - uses the generated meta information to run the Cinterop [10] utility on the assembled framework and get the Kotlin Library (klib);
- *link[LibraryName]Klib* - links the resulting Kotlin Library (klib) with the Kotlin Multiplatform project to provide support in the development environment;
- *spmImport* - starts automatically when importing a project in the development environment and acts as a trigger for previous tasks.

V. Conclusion

Thus, the integration of Kotlin Multiplatform projects with the Swift Package Manager dependencies provides more opportunities than the integration with Cocoapods dependencies. Among the advantages is the ability to publish the resulting artifact and reuse dependencies as a Swift Package in other projects.

To solve the problem of building a Kotlin Multiplatform project into a native artifact and connecting it to an Xcode project through the Swift Package Manager, tasks are also used, but their set and classification are different. In addition to those shown in Fig. 2, 3 groups of tasks are introduced for building and publishing Swift Package Manager dependencies from a Kotlin Multiplatform project:

1) Build generic artifact *XCframework*;
2) Publishing a generic artifact to git (http/ssh);
3) Publishing an archive with a generic artifact.

Taking into account the above groups, the task graph for the build and publish script of the Kotlin Multiplatform project, built by Gradle when analyzing the configuration files, can be represented by a chain of tasks, the functionality of which duplicates the classifier of the group above.

References

[1] "Kotlin: Multiplatform programming," https://kotlinlang.org/docs/reference/multiplatform.html, [Online; accessed January 15, 2021].
[2] "Kotlin multiplatform mobile goes alpha," https://blog.jetbrains.com/ru/kotlin/2020/08/kotlin-multiplatform-mobile-goes-alpha, [Online; accessed February 23, 2021].
[3] "Kotlin multiplatform: case studies," https://kotlinlang.org/lp/mobile/case-studies/, [Online; accessed March 3, 2021].
[4] "Gradle," https://gradle.org, [Online; accessed February 11, 2021].
[5] "Kotlin native," https://kotlinlang.org/docs/native-overview.html, [Online; accessed February 2, 2021].
[6] "Interoperability with c," https://kotlinlang.org/docs/native-c-interop.html, [Online; accessed January 22, 2021].
[7] "Cocoapods integration," https://kotlinlang.org/docs/native-cocoapods.html, [Online; accessed January 27, 2021].
[8] "Binary frameworks in swift." https://developer.apple.com/videos/play/wwdc2019/416, [Online; accessed February 5, 2021].
[9] "Results of the first kotlin multiplatform survey," https://blog.jetbrains.com/kotlin/2021/01/results-of-the-first-kotlin-multiplatform-survey/, [Online; accessed February 17, 2021].
[10] "Using c interop and libcurl for an app – tutorial," https://kotlinlang.org/docs/curl.html, [Online; accessed April 4, 2021].