Чудинов Е.Д. Методика интеграции несовместимых SDK для обновления ANDROID приложений // Международный журнал информационных технологий и энергоэффективности. – 2023. –
Т. 8 № 5(31) ч.2 с. 109–115

Международный журнал информационных технологий и энергоэффективности

Сайт журнала:
http://www.openaccessscience.ru/index.php/ijcse/

УДК 004. 4

МЕТОДИКА ИНТЕГРАЦИИ НЕСОВМЕСТИМЫХ SDK ДЛЯ ОБНОВЛЕНИЯ ANDROID ПРИЛОЖЕНИЙ

Чудинов Е.Д.
ФГБОУ ВО «Челябинский государственный университет», Челябинск, Россия (454001, Челябинская область, город Челябинск, ул. Братьев Кашириных, д.129), e-mail: zotreex@ya.ru

В статье рассматривается применение возможностей механизма In-app-update при использовании несовместимых пакетов SDK в рамках магазинов приложений. Представленная реализация использует асинхронный подход к обновлению состояния. Рассмотрено на примерах Google Play SDK и Rustore SDK.

Ключевые слова: Андроид; обновление в приложении; разработка мобильных приложений; Android-приложение; Rustore SDK

METHODOLOGY FOR INTEGRATING INCOMPATIBLE SDKS TO UPDATE ANDROID APPS

Chudinov E.D.
Chelyabinsk State University, Chelyabinsk, Russia (454001, Chelyabinsk region, Chelyabinsk, Bratya Kashirin street 129), e-mail: zotreex@ya.ru

The article discusses the application of In-app-update mechanism capabilities when using incompatible SDKs within application stores. The presented implementation uses asynchronous state update approach. It is considered on the examples of Google Play SDK and Rustore SDK.

Keywords: Android; in-app-update; mobile application development; android application, rustore sdk.

Для современных мобильных приложений регулярные обновления стали обязательным условиям для поддержания функциональности и безопасности. Разработчикам необходимо регулярно актуализировать существующий функционал под новые требования операционной системы Android, а также удовлетворять запросу бизнеса на своевременную и оперативную доставку нового функционала пользователю. Здесь возникает проблема, многие пользователи отключают автоматическое обновление приложений и сообщить пользователю о выходе новой версии приложения становится затруднительно.

Для решения этой проблемы, существует механизм in-app-updates, позволяющий разработчикам, реализовать процесс обновления непосредственно из приложения, обеспечивая более плавный и быстрый процесс обновления для пользователей. Это помогает улучшить пользовательский опыт, обеспечить дополнительную безопасность и функциональность приложения.

Такое решение предоставляет каждый магазин приложений (Google Play[1], RuStore, Huawei Appgallery), однако в правила этих магазинов запрещают реализовывать сторонний механизм in-app-updates. Например, в Google play нельзя загрузить приложение с использованием RuStore SDK[2] (часть отвечающая за in-app-updates).

Flavors[3] отлично решают проблему использования разных SDK. На основе этой технологии будет рассмотрена методика интеграции несовместимых SDK In-app-updates и предложена практическая рекомендация для разработчиков по применению этого механизма. При реализации, будет использован подход, отличный от имеющихся примеров в документации, соответствующих SDK.

После разбиения проекта на несколько Flavors [1], проект должен обрести несколько новых sourceDir например:

- app/src/google/kotlin – Для сборки с Google Play SDK
- app/src/rustore/kotlin – для сборки с RuStore SDK
- app/src/main/kotlin – исходный код приложения независимый от внешних SDK

Для package main, необходимо создать интерфейс InAppUpdateManager

```
interface InAppUpdateManager {
    val updateState: MutableStateFlow<InAppUpdateState?>
    fun startUpdateFlow()
    fun updateAppInfo()
}
```

Рисунок 1 – Интерфейс InAppUpdateManager.

Здесь, updateState – Flow[4] с текущем состоянием обновления (Null, Available, Installing, Downloading, Downloaded)

startUpdateFlow() – метод для старта загрузки обновления (например, если пользователь нажмёт соответствующую кнопку)

updateAppInfo() – дополнительный метод, позволяющий принудительно обновить состояние обновления, необходим для реализации swipe-to-refresh механизма.

Здесь стоит обратить внимание, что в интерфейсе применяется ассинхронный подход к обновлению состояния по средствам использования потока данных – Flow[4]. При этом, руководства и Rustore SDK и Google Play SDK предлагает изначально только синхронный подход, основанный на использовании callbaks.

Для того чтобы продолжить релазизацию, необходимо подключить SDK, необхобимо перейти build.gradle(:app) и прописать следующие зависимости:

- rustoreImplementation "ru.rustore.sdk:appupdate:0.1.0"
- googleImplementation 'com.google.android.play:app-update:2.0.1'

Благодаря Flavors, сборщик Gradle автоматически будет подключать нужный SDK в соответствующую сборку. Нам же необходимо объяснить приложению, как работать одновременно с двумя SDK. Для этого был создан универсальный интерфейс, о котором будет знать всё приложение, без привязки на конкретное SDK, фактически, здесь применяется принцип инверсии зависимостей.

В Android разработке, часто применяется библиотека Dagger 2, которая позволяет реализовывать этот принцип относительно просто, помогает в целом хорошо структурировать код и строить чистую архитектуру. Чтобы объяснить библиотеки Dagger о том, что у нас будет несколько реализаций, нам потребуется реализовать сразу два модуля (Рисунок 2), с одинаковыми названиями, но лежащие в разных пакетах приложения (/src/google и /src/rustore).

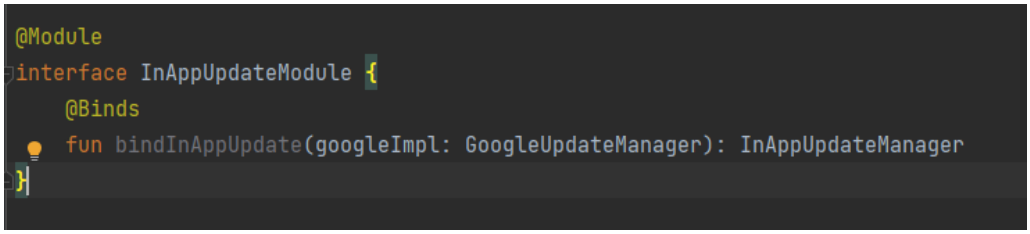

Рисунок 2 – Модуль InAppUpdateModule.

Предварительно, в компонент Dagger`ра, добавлен этот модуль, из-за особенностей реализации, его имя пакета для иморта всегда одино и тот же, для обоих вариаций сборки, а значит не возникнет проблемы, отсутствия этого модуля.

Всё отличие этих двух модулей, будет лишь в методе bindInAppUpdate, в котором параметром передаётся реализация нашего интерфейса. В таком случае, Dagger автоматически построит граф зависимостей для нужной нам сборки, в котором подменит интерфейс InAppUpdateManager на нужную реализацию. Тем самым, реализовав принцип инверсии зависимостей.

Разберём пример реализации GoogleUpdateManager.

Следуюя руководству SDK, необходимо проинциализировать AppUpdateManagerFactory. Создадим класс GoogleUpdateManager и добавим преременную updateManager, которая будет экземпляром AppUpdateManager из Google SDK.

Помимо этого, ещё существует AppUpdateInfo – класс, хранящий информацию о доступности обновлении и прочих вещей для разработчиков. У него есть важная особенность, он используется для старта “Загрузки” передаваясь туда параметром, после чего используемый экземпляр класса перестаёт быть валидным и требудется перезапросить его, используя соответствующий метод из SDK.

```
class GoogleUpdateManager @Inject constructor(
    private val context: Context,
    private val fragmentActivityHolder: FragmentActivityHolder
) : InAppUpdateManager {

    private val updateManager = AppUpdateManagerFactory.create(context)
    private var appUpdateInfo: AppUpdateInfo? = null

    private val listener = InstallStateUpdatedListener { it: InstallState
        handleNewState(it.installStatus(), it)
    }

    private fun getPercentDownloadedText(installState: InstallState?): String {...}

    private fun handleNewState(it: Int, installState: InstallState? = null) {...}

    init {
        updateAppInfo()
    }

    override fun updateAppInfo() {...}

    override val updateState: MutableStateFlow<InAppUpdateState?> =
        MutableStateFlow<InAppUpdateState?>( value: null)

    override fun startUpdateFlow() {...}
}
```

Рисунок 3 – Класс GoogleUpdateManager.

Как было сказано ранее, чтобы наш класс имел смысл, нам необходимо получить от SDK информацию, а именно AppUpdateInfo, для этого в init блок класса, помещается вызов функции updateAppInfo(). Этот метод будет доступен “снаружи” для механизмов принудительного обновления состояния класса.

```
override fun updateAppInfo() {
    updateManager
        .appUpdateInfo
        .addOnSuccessListener { it: AppUpdateInfo!
            appUpdateInfo = it
            if (
                it.updateAvailability() == UpdateAvailability.UPDATE_AVAILABLE ||
                it.updateAvailability() == DEVELOPER_TRIGGERED_UPDATE_IN_PROGRESS
            ) {
                handleNewState(it.installStatus())
            }
        }
}
```

Рисунок 4 – Метод updateAppInfo.

Метод updateAppInfo() – вызывает updateManager, запрашивает информацию appUpdateInfo и синхронным способом, через листенер (он же callback) ожидает результата. Если информация получена успешно, локально сохраняет информацию appUpdateInfo и проверяет два требования:

- Обновление доступно для пользователя
- Обновление уже происходит

Только в этих ситуациях, нам есть необходимость обновить данные во flow, так как иначе null значение, будет расцениваться как отсутствие обновления. Если мы прошли эти проверки, вызываем другой метод – handleNewState, передавая ему информацию о статусе установки (на самом деле, там хранится и информация о доступности обновления)

```
private fun handleNewState(it: Int, installState: InstallState? = null) {
    CoroutineScope(Dispatchers.IO).launch { this: CoroutineScope
        when (it) {
            InstallStatus.DOWNLOADED -> {
                updateAppInfo()
                updateState.emit(InAppUpdateState.Downloaded)
            }
            InstallStatus.DOWNLOADING ->
                if (updateState.value is InAppUpdateState.Downloading)
                    InAppUpdateState.Downloading.percent =
                        getPercentDownloadedText(installState)
                else updateState.emit(
                    InAppUpdateState.Downloading
                )
            InstallStatus.INSTALLING -> updateState.emit(InAppUpdateState.Installing)
            else -> updateState.emit(InAppUpdateState.Available)
        }
    }
}
```

Рисунок 5 – Метод handleNewState.

Данный метод перехватывает возвращаемый SDK Int и проверяет на соответствие определённому состоянию. Его основная задача вызывать updateState.emit передав текущее состояние доступности обновления (здесь уже невозможен вариант отсутствия обновления). В случае, если загрузка уже идёт, перед передачей данных в flow, дополнительно рассчитывается процент из скачанных и общего числа байт.

Остался последний ключевой метод startUpdateFlow() – его задача вызывать интерфейс из SDK для запроса хочет ли пользователь установить, скаченное обновление.

```
override fun startUpdateFlow() {
    appUpdateInfo?.let { it: AppUpdateInfo
        if (it.installStatus() == InstallStatus.DOWNLOADED) {
            updateManager
                .completeUpdate() ^let
        } else {
            fragmentActivityHolder.activity?.let { activity ->
                updateManager.startUpdateFlow(
                    it,
                    activity,
                    AppUpdateOptions.newBuilder(AppUpdateType.FLEXIBLE)
                        .setAllowAssetPackDeletion(true).build()
                ).addOnSuccessListener { resultCode ->
                    if (resultCode == Activity.RESULT_OK) {
                        updateManager.registerListener(listener)
                    } else {
                        updateAppInfo()
                    }
                }

            } ^let

        }
    }
}
```

Рисунок 6 – Метод startUpdateFlow.

Удостоверившись, что метод вызван не ошибочно, вызывается метод completeUpdate из SDK, чтобы произвести обновление приложения уже в системе, так как для этого уже скачена новая версия приложения, а так же пользователь подтвердил установку.

Блок else так же используется, в него мы попадем, если пользователю доступно обновление для скачивания, от него требуется подтверждение, что он хочет произвести загрузку обновления. В листенере, при положительном ответе, будет проинициализирован слушатель, который будет получать информацию о состоянии загрузки. Если пользователь откажет, то нам необходимо перезапросить информацию об обновлении вновь, т.к исчерпаем лимит на использование данных в updateAppInfo.

Для Rustore SDK[2] реализация будет аналогичная. Изменятся только импорты в классе, т.к отечественная реализация полностью совпадает с вариантом от Google.

Рассмотренные в статье реализации механизма InAppUpdates необходимо применять при разработке современных мобильных приложений. Разработанная методика предоставляет

последовательность действий для использования InAppUpdates в ситуациях с применением несовместимых SDK. Предоставленная реализация механизма значительно упрощает применение и поддержку работы InAppUpdate в приложении, публикуемым в несколько магазинов приложений. Подключение подобных SDK в проект может проходить затруднительно, но для этого есть все инструменты и подробная инструкция из этой статьи. Данная технология позволяет решать достаточно много задач, обеспечивая более плавный и быстрый процесс обновления для пользователей.

Список литературы

1. Google documentation In-app updates. URL: https://developer.android.com/guide/playcore/in-app-updates (дата обращения 03.05.2023).
2. RuStore Документация разработчика URL: https://help.rustore.ru/rustore/for_developers/developer-documentation (дата обращения 03.05.2023).
3. Advanced Android Flavors Part 2 — Enter Flavor Dimensions. URL: https://proandroiddev.com/advanced-android-flavors-part-2-enter-flavor-dimensions-4ad7f486f6 (дата обращения 03.05.2023).
4. Документация Kotlin. URL: https://kotlinlang.org/docs/flow.html (дата обращения 03.05.2023).

References

1. Google documentation In-app updates. URL: https://developer.android.com/guide/playcore/in-app-updates (Accessed on 03.05.2023).
2. RuStore Документация разработчика URL: https://help.rustore.ru/rustore/for_developers/developer-documentation (Accessed on 03.05.2023).).
3. Advanced Android Flavors Part 2 — Enter Flavor Dimensions. URL: https://proandroiddev.com/advanced-android-flavors-part-2-enter-flavor-dimensions-4ad7f486f6 ((Accessed on 03.05.2023).
4. Documentation Kotlin. URL: https://kotlinlang.org/docs/flow.html (Accessed on 03.05.2023).