

Table of Contents

suleimanovs.github.io	2
ViewModel под капотом: как она выживает при пересоздании Activity	3
ViewModel в Fragment под капотом: от ViewModelStore до Retain-фрагментов ...	28
ViewModel под капотом: как работает в Compose и View	63
SavedStateHandle и Bundle под капотом: как Android сохраняет состояние	93
Decompose и Essenty: под капотом сохранения состояния без ViewModel	176

suleimanovs.github.io

Репозиторий для публикации личного сайта-архива.

Здесь хранятся технические заметки, черновики и наблюдения о внутреннем устройстве Android, архитектуре, системных механизмах и смежных темах.

Материалы обычно на **русском языке** и носят исследовательский, архивный характер.

Разделы

- Технические заметки и примеры
- Разбор подкапотных механизмов Android и архитектурных компонентов
- Личные инженерные наблюдения и выводы

О проекте

Это не полноценный блог и не коммерческий сайт. Репозиторий служит как архив и рабочий черновик для фиксации мыслей и промежуточных версий текстов. Контент постепенно дополняется по мере появления новых заметок.

Ссылки

- Блог suleimanov.com (<https://www.suleimanov.com>)
- osman.suleimanovs@gmail.com

ViewModel под капотом: как она выживает при пересоздании Activity

Введение

В статье не рассматривается работа с ViewModel, предполагается, что эта тема уже знакома. Основное внимание уделяется тому, как ViewModel переживает изменение конфигурации. Но для начала — небольшое введение в ViewModel.

ViewModel - компонент архитектурного паттерна MVVM, который был предоставлен Google как примитив позволяющий пережить изменение конфигураций. Изменение конфигураций в свою очередь - это состояние, заставляющая activity/fragment пересоздаваться, это именно то состояние которое может пережить ViewModel. Популярные конфигурации которые приводят к пересозданию Activity:

1. Изменение ориентаций экрана(screenOrientation): portrait/landscape
2. Изменение направления экрана(layoutDirection): rtl/ltr
3. Изменение языка приложения(locale)
4. Изменение размера шрифтов/соотношение экрана

Есть конечно способ сообщать системе о том что пересоздавать Activity при изменении конфигураций не нужно. Флаг `android:configChanges` используется в `AndroidManifest.xml` в теге `<activity/>`, чтобы указать, какие изменения конфигурации система не должна пересоздавать Activity, а передавать управление методу `Activity.onConfigurationChanged()`.

```
<activity
    android:name="MainActivity"

    android:configChanges="layoutDirection|touchscreen|density|orientation|keyboard|locale|keyboardHidden|navigation|screenLayout|mcc|mnc"
```

```
cl fontScale|uiModel|screenSize|smallestScreenSize"  
/>
```

Однако сейчас речь не об этом. Наша цель — разобраться, каким образом `ViewModel` умудряется переживать все изменения конфигурации и сохранять своё состояние.

Объявление ViewModel

С появлением делегатов в Kotlin разработчики получили возможность значительно упростить создание и использование компонентов. Теперь объявление `ViewModel` с использованием делегатов выглядит следующим образом:

```
class MainActivity : ComponentActivity() {  
  
    private val viewModel by viewModel<MyViewModel>()  
  
}
```

Без делегатов создание объекта `ViewModel`, используя явный вызов `ViewModelProvider` выглядит следующий образом:

```
class MainActivity : ComponentActivity() {  
  
    private lateinit var viewModel: MyViewModel  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
  
        // В старых версиях ViewModelProvider был частью  
        lifecycle-viewmodel  
        viewModel =  
        ViewModelProvider(this).get(MyViewModel::class.java)  
  
        // После адаптации ViewModel под KMP и переноса  
        ViewModelProvider в lifecycle-viewmodel-android  
        // можно и рекомендуется через перегруженный фабричный  
        метод create:
```

```

        viewModel = ViewModelProvider.create(owner =
this).get(MyViewModel::class)

        // Альтернативный способ создания ViewModel (эквивалентен
предыдущему)
        viewModel = ViewModelProvider.create(store =
this.viewModelStore).get(MyViewModel::class)
    }
}

```

Метод `ViewModelProvider.create` имеет параметры со значениями по умолчанию, поэтому на уровне байткода компилятор создаст несколько перегруженных версий метода (overloads). Это позволяет вызывать его с разным количеством аргументов: только с `store`, с `store` и `factory`, либо со всеми параметрами, включая `extras`.

i **Jetpack ViewModel** теперь поддерживает **Kotlin Multiplatform (KMP)**, что позволяет использовать его не только на Android, но и на iOS, Desktop и Web. Это стало возможным благодаря разделению на два модуля:

lifecycle-viewmodel(expected): KMP-модуль без привязки к Android.

lifecycle-viewmodel-android(actual): модуль для работы с `ViewModelStoreOwner` и `ViewModelProvider` на Android.

Начиная с версии **2.8.0-alpha03**, артефакты **lifecycle-*** теперь официально поддерживают Kotlin Multiplatform! Это означает, что классы, такие как `ViewModel`, `ViewModelStore`, `ViewModelStoreOwner` и `ViewModelProvider`, теперь можно использовать в общем коде.


⚠ Далее в статье мы рассмотрим именно версию `viewmodel:2.8.0+`, если в версиях на которой вы находитесь сейчас немного отличаются исходники, то не переживайте, с добавлением поддержки kmp немного поменяли внутреннюю структуру, но реализация и внутренняя логика такая же что и до поддержки kmp

ViewModelStoreOwner ?

Как мы видим выше, мы вручную не создаём объект `ViewModel`, а только передаём тип его класса в `ViewModelProvider`, который самостоятельно занимается созданием экземпляра.

Обратите внимание, что мы также передаём в метод `ViewModelProvider.create` параметр `owner = this`. Если заглянуть в исходники метода `create`, можно заметить, что требуется тип `owner: ViewModelStoreOwner`:

```
public actual companion object {  
  
    @JvmStatic  
    @Suppress("MissingJvmstatic")  
    public actual fun create(  
        owner: ViewModelStoreOwner, // <- нас интересует этот тип  
        factory: Factory,  
        extras: CreationExtras,  
    ): ViewModelProvider = ViewModelProvider(owner.viewModelStore,  
        factory, extras)  
}
```

 Если интересно, почему метод `create()` можно вызывать без передачи значений для параметров `factory` и `extras` (хоть они и обязательны):

```
ViewModelProvider.create(owner = this)
```

Это связано с тем, что код использует KMP (Kotlin Multiplatform). В expect-объявлении для `create()` уже заданы значения по умолчанию для `factory` и `extras`, поэтому передавать их явно необязательно.

```
public expect class ViewModelProvider {  
    ....  
    public companion object {  
        public fun create(  
            owner: ViewModelStoreOwner,  
            factory: Factory =  
                ViewModelProviders.getDefaultFactory(owner),  
            extras: CreationExtras =
```

```

        ViewModelProviders.getDefaultCreationExtras(owner),
        ): ViewModelProvider

        }
        ....
    }

```

Подробнее можно посмотреть в исходниках: `ViewModelProvider.kt`
<https://github.com/androidx/androidx/blob/androidx-main/lifecycle/lifecycle-viewmodel/src/commonMain/kotlin/androidx/lifecycle/ViewModelProvider.kt>)

Углубляемся в `ViewModelStore / Owner`

Получается что при вызове метода `ViewModelProvider.create()` для параметра `owner` мы передаем `this` (само активити), и как можно догадаться, это означает, что `activity` реализует(наследуется) от интерфейса `ViewModelStoreOwner`. Давайте взглянем на исходники этого интерфейса: `ViewModelStoreOwner`

<https://github.com/androidx/androidx/blob/androidx-main/lifecycle/lifecycle-viewmodel/src/commonMain/kotlin/androidx/lifecycle/ViewModelStoreOwner.kt>):

```

public interface [[[[ViewModelStoreOwner
|https://github.com/androidx/androidx/blob/androidx-
main/lifecycle/lifecycle-
viewmodel/src/commonMain/kotlin/androidx/lifecycle/ViewModelProvid
er.kt]]]] {

    /**
     * The owned [ViewModelStore]
     */
    public val viewModelStore: ViewModelStore
}

```

`ViewModelStoreOwner` — это интерфейс с единственным полем, которое представляет собой `ViewModelStore` (хранитель `view models`). От `ViewModelStoreOwner` наследуются такие компоненты как: **`ComponentActivity`**, **`Fragment`**, * `NavBackStackEntry`.*

Официальная документация гласит:

⚠ A scope that owns ViewModelStore. A responsibility of an implementation of this interface is to retain owned ViewModelStore during the configuration changes and call ViewModelStore.clear, when this scope is going to be destroyed.

Обязанности ViewModelStoreOwner:

1. Хранение ViewModelStore во время изменения конфигураций.
2. Очистка ViewModelStore при уничтожении ComponentActivity/Fragment — в состоянии onDestroy(). Удаляются все ViewModel-и которые ViewModelStore хранить в себе.

Мы определили, что ViewModelStoreOwner — это всего лишь интерфейс, не содержащий собственной логики. Его реализуют такие компоненты, как:

- **ComponentActivity** (и его наследники: *FragmentActivity*, *AppCompatActivity*)
- **Fragment** (и его производные: *DialogFragment*, *BottomSheetDialogFragment*, *AppCompatDialogFragment*).
- **NavBackStackEntry** - Класс из библиотеки Jetpack Navigation (он же androidx.navigation)

Далее нас уже интересует сам ViewModelStore:

ViewModelStore — это класс, который внутри себя делегирует управление коллекцией Map (LinkedHashMap) для хранения ViewModel по ключу:

```
private val map = mutableMapOf<String, ViewModel>()
```

По умолчанию в качестве ключа используется полное имя класса (включая его пакет). Этот ключ генерируется следующим образом в исходниках утилитного класса ViewModelProviders (не путать с ViewModelProvider):


```
private const val VIEW_MODEL_PROVIDER_DEFAULT_KEY: String =
    "androidx.lifecycle.ViewModelProvider.DefaultKey"

internal fun <T : ViewModel> getDefaultKey(modelClass: KClass<T>):
String {
    return
    "$VIEW_MODEL_PROVIDER_DEFAULT_KEY:$modelClass.canonicalName"
}
```

Таким образом, для MyViewModel ключ будет выглядеть так:

```
androidx.lifecycle.ViewModelProvider.DefaultKey:com.example.MyViewModel.
```

Поскольку ViewModelStore основан на Map, он делегирует все основные операции, такие как put, get, keys и clear, внутреннему Map (LinkedHashMap).

Соответственно, так как внутренняя реализация ViewModelStore полагается на Map, он также делегирует свои методы put, get, key, clear внутреннему Map (LinkedHashMap). Особого внимания заслуживает метод clear():

```
public open class ViewModelStore {

    private val map = mutableMapOf<String, ViewModel>()
    ...
    /**
     * Clears internal storage and notifies `ViewModel`s that they
     are no longer used.
     */
    public fun clear() {
        for (vm in map.values) {
            vm.clear()
        }
        map.clear()
    }
}
```

Давайте разберёмся, что здесь происходит. Когда наш ViewModelStoreOwner (в лице ComponentActivity или Fragment) окончательно умирает (смерть не связана с

пересозданием из-за изменения конфигураций), он вызывает метод `clear()` у `ViewModelStore`.

В методе `clear()` цикл `for` проходит по всем значениям (`view models`), которые хранятся внутри внутреннего `HashMap`, и вызывает у каждой `ViewModel` внутренний метод `clear()`. Этот метод, в свою очередь, инициирует вызов метода `onCleared()` у нашей `ViewModel`.

`onCleared()` — это метод, который мы можем переопределить в своей `ViewModel`, и он вызывается только в момент окончательного уничтожения `ViewModel`, когда активности или фрагмент также окончательно завершают свою работу.

```
public actual abstract class ViewModel {
    ...
    protected actual open fun onCleared() {} // <- метод
    onCleared, который можно переопределить

    @MainThread
    internal actual fun clear() {
        impl?.clear()
        onCleared() // <- вызов метода onCleared
    }
}
```

Таким образом, метод `clear()` гарантирует, что все ресурсы и фоновые задачи, связанные с `ViewModel`, будут корректно освобождены перед уничтожением.

Соответственно, сам метод `viewModelStore.clear()` вызывается

`ViewModelStoreOwner` (в лице `ComponentActivity` или `Fragment`). Давайте в качестве примера выберем `ComponentActivity`, чтобы понять, как работает очистка.

Ниже приведён фрагмент кода из `ComponentActivity`, который отслеживает её уничтожение и вызывает `viewModelStore.clear()`:

```
@Suppress("LeakingThis")
lifecycle.addObserver(
    LifecycleEventObserver { _, event ->
        if (event == Lifecycle.Event.ON_DESTROY) { // <- состояние
            ON_DESTROY является триггером
            // Clear out the available context
        }
    }
)
```

```

        contextAwareHelper.clearAvailableContext()
        // And clear the ViewModelStore
        if (!isChangingConfigurations) { // <- проверка на то
можно ли очищать ViewModelStore
            viewModelStore.clear()      // <- очистка
        }
        reportFullyDrawnExecutor.activityDestroyed()
    }
}
)

```

В данном коде происходит добавление наблюдателя на жизненный цикл активности с использованием `LifecycleEventObserver`. Когда активность достигает состояния `ON_DESTROY`, запускается проверка, не происходит ли изменение конфигурации (`isChangingConfigurations`). Если активность действительно умирает окончательно (и не пересоздаётся), вызывается метод `viewModelStore.clear()`, который очищает все связанные с активностью `ViewModel`.

Мы видим, что проверка состояния `ON_DESTROY` в сочетании с условием `if (!isChangingConfigurations)` позволяет убедиться в том, что причиной уничтожения не является изменение конфигурации. Только в этом случае очищается `ViewModelStore` и удаляются все экземпляры `ViewModel`, связанные с данной активностью.

⚠ В этой статье мы подробно разбираем внутренние методы класса `ComponentActivity`, начиная с версии `**androidx.activity:activity:1.9.0-alpha01**` (<https://developer.android.com/jetpack/androidx/releases/activity#1.9.0-alpha01>), когда он был переписан на Kotlin.

Если у вас установлена более старая версия библиотеки, и вы видите реализацию на Java — не переживайте. Логика и основные методы остались прежними, поэтому все представленные концепции и объяснения будут актуальны.

Процесс очистки `ViewModel` при уничтожении активности:

- Уничтожение Activity (не связано с изменением конфигураций)
ComponentActivity.onDestroy() -> Очистка ViewModelStore
getViewModelStore().clear() -> Оповещение ViewModel
MyViewModel.onCleared()

Теперь мы разобрались с процессом очистки и уничтожения `ViewModel`. Перейдём к следующему этапу — рассмотрим подробнее, как происходит создание объекта `ViewModel`, когда мы передаём её в `ViewModelProvider`:

```
ViewModelProvider.create(owner = this).get(MyViewModel::class)
```

Да, можно уточнить, что `ViewModelProvider.create` — это функция с значениями по умолчанию. Например:

```
ViewModelProvider.create(owner = this).get(MyViewModel::class)
```

Ранее мы разобрали один из перегруженных методов `ViewModelProvider.create` (функция с аргументами по умолчанию). Это фабричный метод, который принимает минимум `ViewModelStore` или `ViewModelStoreOwner`, создаёт объект `ViewModelProvider` и на этом завершает свою работу.

Теперь нас интересует следующий ключевой метод — `get`, который принимает класс `ViewModel` в качестве параметра. `ViewModelProvider` делегирует свою работу классу `ViewModelProviderImpl`:

```
public actual open class ViewModelProvider private constructor(  
    private val impl: ViewModelProviderImpl,  
) {  
    ...  
    @MainThread  
    public actual operator fun <T : ViewModel> get(modelClass:  
        KClass<T>): T =  
        impl.getViewModel(modelClass) // <- вызов метода  
        getViewModel, принадлежащий ViewModelProviderImpl  
}
```

⚠ Разработчики Google вынесли общую логику создания ViewModel в отдельный объект ViewModelProviderImpl. Это позволило избежать дублирования кода на разных платформах в КМР. Причина в том, что expect-классы в Kotlin Multiplatform не могут содержать реализации методов по умолчанию. Если бы они могли, реализация находилась бы прямо внутри expect-версии ViewModelProvider, без необходимости выносить её в отдельный объект. Однако, из-за этого ограничения, была создана ViewModelProviderImpl, которая содержит общую логику создания ViewModel для всех платформ.

Оригинальный комментарий:

Kotlin Multiplatform does not support expect class with default implementation yet, so we extracted the common logic used by all platforms to this internal class.

Исходники метода `getViewModel()` в `ViewModelProviderImpl.kt`:

```
internal fun <T : ViewModel> getViewModel(
    modelClass: KClass<T>,
    key: String = ViewModelProviders.getDefaultKey(modelClass),
): T {
    val viewModel = store[key] // 1. Достается viewModel из
    ViewModelStore, если он существует
    if (modelClass.isInstance(viewModel)) {
        if (factory is ViewModelProvider.OnRequeryFactory) {
            factory.onRequery(viewModel!!)
        }
        return viewModel as T
    }

    val extras = MutableCreationExtras(extras)
    extras[ViewModelProviders.ViewModelKey] = key
    // 2. Создается viewModel и кладется в ViewModelStore
    return createViewModel(factory, modelClass, extras).also { vm
-> store.put(key, vm) }
}
```

При вызове `ViewModelProvider.create()` под капотом вызывается метод `getViewModel()`, который выполняет следующие шаги:

1. Проверяет наличие объекта `ViewModel` в `ViewModelStore` по заданному ключу. Если объект уже существует, он возвращается.
2. Если объект не найден, создаётся новый экземпляр `ViewModel`, который затем кладётся в `ViewModelStore` для последующего использования.

Где `ViewModelStore` сохраняется?

Теперь, когда мы знаем полный процесс создания `ViewModel` и её размещения в `ViewModelStore`, возникает логичный вопрос: если все `ViewModel`-и хранятся внутри `ViewModelStore`, а сам `ViewModelStore` находится в `ComponentActivity` или `Fragment`, которые реализуют интерфейс `ViewModelStoreOwner`, то где и как хранится сам объект `ViewModelStore`?

Для того чтобы найти ответ на вопрос о хранении `ViewModelStore`, давайте посмотрим, как `ComponentActivity` реализует интерфейс `ViewModelStoreOwner`:

```
override val viewModelStore: ViewModelStore
get() {
    checkNotNull(application) {
        ("Your activity is not yet attached to the " +
            "Application instance. You can't request ViewModel
before onCreate call.")
    }
    ensureViewModelStore()
    return _viewModelStore!!
}
```

Мы видим, что вызывается метод `ensureViewModelStore`, а затем возвращается поле `_viewModelStore`.

```
// Lazily recreated from NonConfigurationInstances by val
viewModelStore
private var _viewModelStore: ViewModelStore? = null
```

Поле `_viewModelStore` не имеет значения по умолчанию, поэтому перед возвратом оно инициализируется внутри метода `ensureViewModelStore`:

```
private fun ensureViewModelStore() {
    if (_viewModelStore == null) {
        // Извлекается ComponentActivity#NonConfigurationInstances
        // из метода Activity#getLastNonConfigurationInstance()
        val nc = lastNonConfigurationInstance as
        NonConfigurationInstances?
        if (nc != null) {
            // Восстанавливается ViewModelStore из
            NonConfigurationInstances
            _viewModelStore = nc.viewModelStore
        }
        if (_viewModelStore == null) {
            // Создается ViewModelStore если нет сохраненного
            // внутри объекта NonConfigurationInstances
            _viewModelStore = ViewModelStore()
        }
    }
}
```

Тут-то и начинается самое интересное. Если поле `_viewModelStore` равно `null`, сначала выполняется попытка получить его из метода `getLastNonConfigurationInstance()`, который возвращает объект класса `NonConfigurationInstances`.

Если `ViewModelStore` отсутствует и там, это может означать одно из двух:

1. Активность создаётся впервые и у неё ещё нет сохранённого `ViewModelStore`.
2. Система уничтожила процесс приложения (например, из-за нехватки памяти), а затем пользователь снова запустил приложение, из-за чего `ViewModelStore` не сохранился.

В любом из этих случаев создаётся новый экземпляр `ViewModelStore`.

Самая неочевидная часть — это вызов метода `getLastNonConfigurationInstance()`. Этот метод принадлежит классу `Activity`, а класс `NonConfigurationInstances`, у

которого даже само название выглядит интригующе, объявлен в `ComponentActivity`:

```
internal class NonConfigurationInstances {  
    var custom: Any? = null  
    var viewModelStore: ViewModelStore? = null  
}
```

Таким образом, объект `NonConfigurationInstances` используется для хранения `ViewModelStore` при изменении конфигурации активности. Это позволяет сохранить состояние `ViewModel` и восстановить его после пересоздания активности.

Переменная `custom` по умолчанию имеет значение `null` и фактически не используется, поскольку `ViewModelStore` более гибко выполняет всю работу по сохранению состояний для переживания изменений конфигураций. Тем не менее, переменную `custom` можно задействовать, переопределив такие функции, как `onRetainCustomNonConfigurationInstance` и `getLastCustomNonConfigurationInstance`. До появления `ViewModel` многие разработчики активно использовали(в 2012) именно её для сохранения данных при пересоздании активности когда менялась конфигурация.

Переменная `viewModelStore` имеет тип `ViewModelStore` и хранит ссылку на наш объект `ViewModelStore`. Значение в эту переменную `NonConfigurationInstances#viewModelStore` присваивается при вызове метода `onRetainNonConfigurationInstance`, а извлекается при вызове `getLastNonConfigurationInstance` (с этим методом мы уже столкнулись выше в методе `ensureViewModelStore`).

Разобравшись с классом `NonConfigurationInstances`, давайте выясним, где создаётся объект этого класса и каким образом в поле `viewModelStore` присваивается значение. Для этого обратимся к методам `onRetainNonConfigurationInstance` и `getLastNonConfigurationInstance`, которые присутствуют в `Activity` и `ComponentActivity`. Исходники метода в `ComponentActivity` выглядят следующим образом:

```
@Suppress("deprecation")  
final override fun onRetainNonConfigurationInstance(): Any? {  
    // Maintain backward compatibility.
```



```

    val custom = onRetainCustomNonConfigurationInstance()
    var viewModelStore = _viewModelStore
    if (viewModelStore == null) {
        // No one called getViewModelStore(), so see if there was
an existing
        // ViewModelStore from our last NonConfigurationInstance
        val nc = lastNonConfigurationInstance as
NonConfigurationInstances?
        if (nc != null) {
            viewModelStore = nc.viewModelStore
        }
    }
    if (viewModelStore == null && custom == null) {
        return null
    }
    val nci = NonConfigurationInstances()
    nci.custom = custom
    nci.viewModelStore = viewModelStore
    return nci
}

```

Метод `onRetainNonConfigurationInstance()` возвращает объект `NonConfigurationInstances`, содержащий ссылку на ранее созданный `ViewModelStore`.

Таким образом, при уничтожении активности (например, при повороте экрана) вызывается этот метод, и `ViewModelStore` сохраняется в экземпляре `NonConfigurationInstances`. Когда активность пересоздаётся, объект `NonConfigurationInstances` восстанавливается через вызов метода `getLastNonConfigurationInstance()`, и из него извлекается сохранённый `ViewModelStore`.

В методе `onRetainNonConfigurationInstance` реализована логика получения уже существующего `ViewModelStore` и объекта `Custom` (если он есть). После получения этих объектов они кладутся в экземпляр класса `NonConfigurationInstances`, который затем возвращается из метода.

Метод `onRetainNonConfigurationInstance` создаёт объект класса `NonConfigurationInstances`, помещает внутрь `viewModelStore` и кастомный объект, а

затем возвращает его. Возникает вопрос: кто именно вызывает этот метод?

Вызывающий метод внутри самого класса Activity(самый базовый Activity от которого наследуются все остальные):

```
NonConfigurationInstances retainNonConfigurationInstances() {
    Object activity = onRetainNonConfigurationInstance(); // <-
    вызов onRetainNonConfigurationInstance()

    //...code

    NonConfigurationInstances nci = new
    NonConfigurationInstances();
    nci.activity = activity; // <- присвоение извлеченного объекта
    из onRetainNonConfigurationInstance()
    nci.children = children;
    nci.fragments = fragments;
    nci.loaders = loaders;
    if (mVoiceInteractor != null) {
        mVoiceInteractor.retainInstance();
        nci.voiceInteractor = mVoiceInteractor;
    }
    return nci;
}
```

Как видно, сам класс Activity вызывает метод onRetainNonConfigurationInstance с которым мы ранее познакомились и сохраняет результат в поле activity класса NonConfigurationInstances. При этом мы снова сталкиваемся с классом NonConfigurationInstances, но на этот раз он объявлен в самой Activity и имеет дополнительные поля:

```
static final class NonConfigurationInstances {
    Object activity; // <- Здесь и будет храниться
    ComponentActivity.NonConfigurationInstances
    HashMap<String, Object> children;
    FragmentManagerNonConfig fragments;
    ArrayMap<String, LoaderManager> loaders;
```

```
VoiceInteractor voiceInteractor;  
}
```

Чтобы устранить путаницу:

- Объект `ViewModelStore` хранится внутри `ComponentActivity#NonConfigurationInstances`.
- Сам объект `ComponentActivity#NonConfigurationInstances` хранится в `Activity#NonConfigurationInstance`.
- Это достигается через метод `retainNonConfigurationInstances()` класса `Activity`.

Но кто же вызывает метод `retainNonConfigurationInstances()` и где хранится конечный объект `Activity#NonConfigurationInstance`, который содержит `ViewModelStore`?

Ответ на этот вопрос кроется в классе `ActivityThread`, который отвечает за управление жизненным циклом активностей и их взаимодействие с системой. Именно этот класс обрабатывает создание, уничтожение и повторное создание активности, а также отвечает за сохранение и восстановление данных при изменениях конфигурации.

Метод из `ActivityThread`, который непосредственно вызывает `Activity.retainNonConfigurationInstances()`, называется `ActivityThread.performDestroyActivity()`.

Рассмотрим его исходники в классе `ActivityThread`, далее исходники:

```
void performDestroyActivity(ActivityClientRecord r, boolean  
finishing,  
                                boolean getNonConfigInstance, String  
reason) {  
    //...  
    if (getNonConfigInstance) {  
        try {  
            // Вызов Activity.retainNonConfigurationInstances()  
            // и сохранение в r.lastNonConfigurationInstances  
            r.lastNonConfigurationInstances =  
            r.activity.retainNonConfigurationInstances();
```

```

        } catch (Exception e) {
            if (!mInstrumentation.onException(r.activity, e)) {
                throw new RuntimeException("Unable to retain
activity "
                                + r.intent.getComponent().toShortString()
+ ": " + e.toString(), e);
            }
        }
    }
    //...
}

```

i Чтобы найти исходники `ActivityThread`, достаточно в Android Studio воспользоваться поиском по имени класса: `ActivityThread`. Или зайти в исходники Android по одной из ссылок:

- Android Source (cs.android.com)
(<https://cs.android.com/android/platform/superproject/+/master:frameworks/base/core/java/android/app/ActivityThread.java>)
- Android Google Source (googlesource.com)
(<https://android.googlesource.com/platform/frameworks/base/+/0e40462e11d27eb859b829b112cecb8c6f0d7afb/core/java/android/app/ActivityThread.java>)

После вызова метода `retainNonConfigurationInstances()` результат сохраняется в поле `lastNonConfigurationInstances` объекта `ActivityClientRecord`:

```

r.lastNonConfigurationInstances =
r.activity.retainNonConfigurationInstances();

```

Класс `ActivityClientRecord` представляет собой запись активности и используется для хранения всей информации, связанной с реальным экземпляром активности. Это своего рода структура данных для ведения учета активности в процессе выполнения приложения.

Основные поля класса `ActivityClientRecord`:

- `lastNonConfigurationInstances` — объект `Activity#NonConfigurationInstance`, в котором хранится `ComponentActivity#NonConfigurationInstances` в котором хранится `ViewModelStore`.
- `state` — объект `Bundle`, содержащий сохраненное состояние активности. Да, да, это тот самый `Bundle` который мы получаем в методе `onCreate`, `onRestoreInstanceState` и `onSaveInstanceState`
- `intent` — объект `Intent`, представляющий намерение запуска активности.
- `window` — объект `Window`, связанный с активностью.
- `activity` — сам объект `Activity`.
- `parent` — родительская активность (если есть).
- `createdConfig` — объект `Configuration`, содержащий настройки, примененные при создании активности.
- `overrideConfig` — объект `Configuration`, содержащий текущие настройки активности.

В рамках данной статьи нас интересует только поле `lastNonConfigurationInstances`, так как именно оно связано с хранением и восстановлением `ViewModelStore`.

Теперь давайте разберемся, как вызывается метод `performDestroyActivity()` в рамках системного вызова.

Последовательность вызовов:

1. `ActivityTransactionItem.execute()`
2. `ActivityRelaunchItem.execute()`
3. `ActivityThread.handleRelaunchActivity()`
4. `ActivityThread.handleRelaunchActivityInner()`
5. `ActivityThread.handleDestroyActivity()`

6. ActivityThread.performDestroyActivity()

i Важно понимать, что на более высоком уровне в этой цепочке стоят такие классы, как `ClientTransactionItem`, `ClientTransaction` и `ClientLifecycleManager`, а еще выше — сама система, которая управляет взаимодействием устройства с сенсорами и другими компонентами. Однако, углубляться дальше в эту цепочку мы не будем, так как всего через пару слоев окажемся на уровне межпроцессного взаимодействия (IPC) и работы системы с процессами.

На вершине вызовов находится метод `ActivityTransactionItem.execute()`, который запускает цепочку: сначала вызывает `getActivityClientRecord()`, а затем тот вызывает `ClientTransactionHandler.getActivityClient()`.

```
public abstract class ActivityTransactionItem extends
ClientTransactionItem {
    @Override
    public final void execute(ClientTransactionHandler client,
IBinder token,
                                PendingTransactionActions
pendingActions) {
        final ActivityClientRecord r =
getActivityClientRecord(client, token); // <- Вызов
getActivityClientRecord

        execute(client, r, pendingActions);
    }

    @NonNull
    ActivityClientRecord getActivityClientRecord(
        @NonNull ClientTransactionHandler client, IBinder
token) {
        final ActivityClientRecord r =
client.getActivityClient(token); // <- получение клиент от
ClientTransactionHandler(ActivityThread)
        ...
    }
}
```

```

        return r;
    }
}

```

`ClientTransactionHandler` — это абстрактный класс, и одна из его реализаций — класс `ActivityThread`, с которым мы уже успели познакомиться.

```

public final class ActivityThread extends ClientTransactionHandler
    implements ActivityThreadInternal {
    ...

    @Override
    public ActivityClientRecord getActivityClient(IBinder token) {
        return mActivities.get(token); // <- Возвращает из Map
        ActivityClientRecord по ключу
    }
    ...
}

```

От `ActivityTransactionItem` — наследуется класс `ActivityRelaunchItem`, который и запускает у `ActivityThread` метод `handleRelaunchActivity`:

```

public class ActivityRelaunchItem extends ActivityTransactionItem
{

    @Override
    public void execute(@NonNull ClientTransactionHandler client,
        @NonNull ActivityClientRecord r,
        @NonNull PendingTransactionActions
        pendingActions) {
        ...
        client.handleRelaunchActivity(mActivityClientRecord,
        pendingActions);
        ...
    }
}

```

Все запущенные активности внутри нашего приложения хранятся в коллекции Map в объекте класса `ActivityThread`:

```
/**
 * Maps from activity token to local record of running activities
 * in this process.
 * ....
 */
@UnsupportedAppUsage
final ArrayMap<IBinder, ActivityClientRecord> mActivities = new
ArrayMap<>();
```

Таким образом, мы наконец выяснили, что наша `ViewModel` фактически хранится в объекте `ActivityThread`, который является синглтоном. Благодаря этому `ViewModel` не уничтожается при изменении конфигурации.

Важно: Экземпляр `ActivityThread` является синглтоном и существует на протяжении всего жизненного цикла процесса приложения. В методе `handleBindApplication()` внутри `ActivityThread` создается объект `Application`, который также живет до завершения процесса. Это означает, что `ActivityThread` и `Application` связаны общим жизненным циклом, за исключением того, что `ActivityThread` появляется раньше — еще до создания `Application` — и управляет его инициализацией.

Восстановление ViewModelStore

Исходя из того, что мы обнаружили ранее, цепочка хранения `ViewModel` выглядит следующим образом:

1. `ViewModel` хранится внутри `ViewModelStore`.
2. `ViewModelStore` хранится в `ComponentActivity#NonConfigurationInstances`.
3. `ComponentActivity#NonConfigurationInstances` хранится в `Activity#NonConfigurationInstance`.
4. `Activity#NonConfigurationInstance` хранится в `ActivityClientRecord`.
5. `ActivityClientRecord` хранится в `ActivityThread`.

При повторном создании `Activity` вызывается его метод `attach()`, одним из параметров которого является `Activity#NonConfigurationInstances`. Этот объект извлекается из связанного с `Activity` объекта `ActivityClientRecord`.

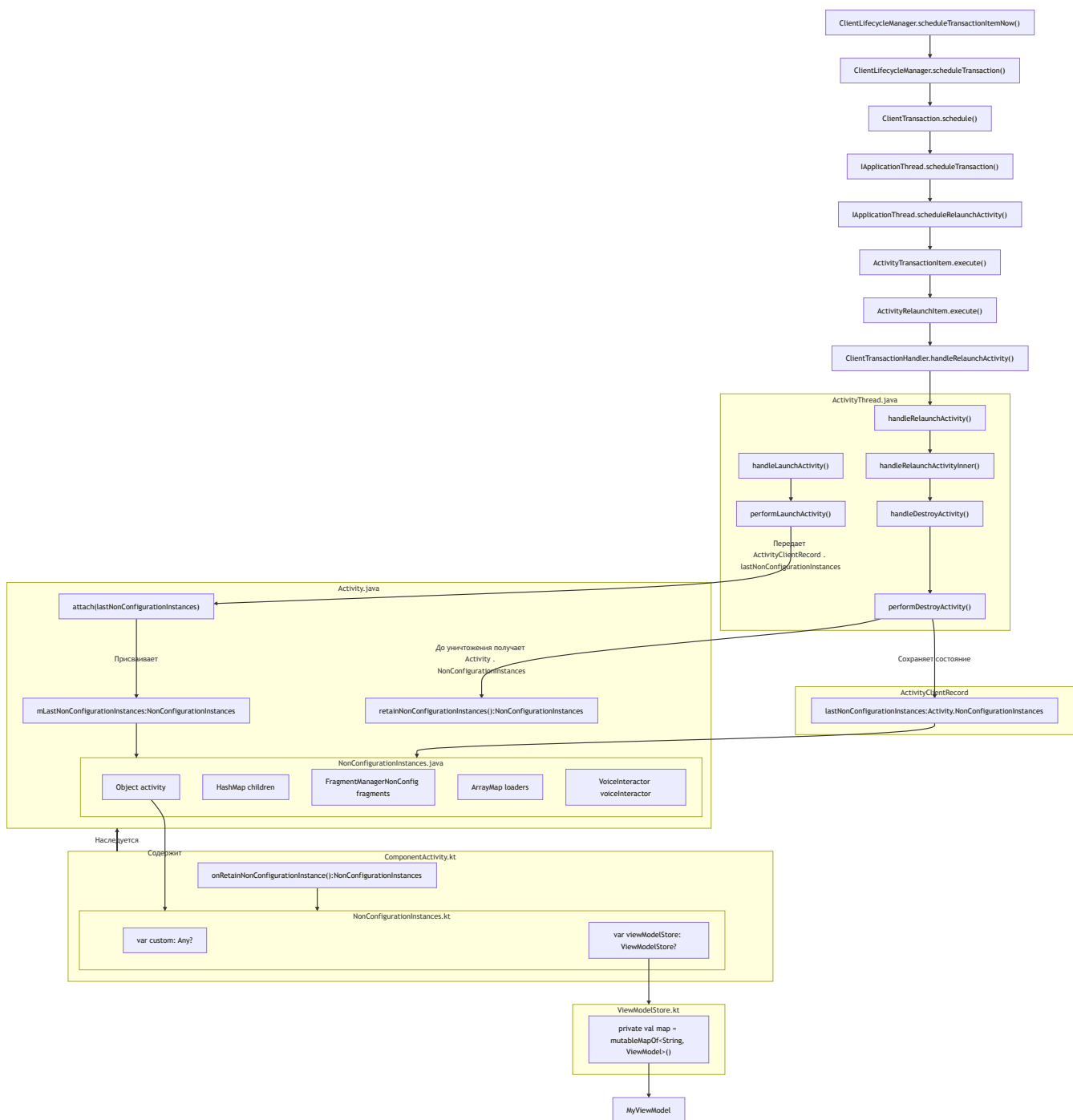
Когда у `Activity` меняется конфигурация, система сразу же перезапускает её, чтобы применить новые параметры. В этот момент `ActivityThread.java` мгновенно извлекает `ViewModelStore`, который хранится в `ComponentActivity#NonConfigurationInstances`. Этот объект, в свою очередь, находится внутри `Activity#NonConfigurationInstances`.

Далее `Activity#NonConfigurationInstances` сохраняется в `ActivityClientRecord`, связанном с пересоздаваемой `Activity`. Внутри `ActivityClientRecord` есть специальное поле `lastNonConfigurationInstances`, куда и помещается этот объект. Сам `ActivityClientRecord` хранится в `Map`-коллекции внутри `ActivityThread.java`, который является синглтоном в рамках процесса приложения и способен переживать изменения конфигурации.

После этого `ActivityThread` пересоздаёт `Activity`, применяя новые параметры конфигурации. При создании он передаёт в неё все сохранённые данные, включая `NonConfigurationInstances`, который, в конечном итоге, содержит `ViewModelStore`. А `ViewModelStore`, в свою очередь, хранит нашу `ViewModel`.

Диаграмма вызовов при сохранении и восстановлении `ViewModelStore`

Диаграмма ниже иллюстрирует цепочку вызовов. Ради упрощения некоторые детали опущены, а избыточные абстракции убраны:



Итоги

В этой статье мы не касались работы ViewModel как таковой — фокус был исключительно на том, **почему она не умирает при пересоздании Activity**, и за счёт чего это вообще возможно.

Мы проследили всю цепочку: **ViewModel** → **ViewModelStore** → **ComponentActivity#NonConfigurationInstances** → **Activity#NonConfigurationInstances**

→ `ActivityClientRecord` → `ActivityThread`. Именно в этой глубокой вложенности и заключается ответ: **ViewModel выживает, потому что сохраняется не в Activity напрямую, а в объекте, который система сама передаёт новой Activity при конфигурационных изменениях.**

Сам `ViewModelStore` создаётся либо с нуля, либо восстанавливается через `getLastNonConfigurationInstance()`. Он очищается только в `onDestroy()`, если `isChangingConfigurations == false`, — то есть если Activity действительно умирает, а не пересоздаётся.

Под капотом всё это обеспечивается `ActivityThread`, который сохраняет `NonConfigurationInstances` в `ActivityClientRecord`, а потом передаёт в метод `attach()` при создании новой Activity. `ActivityThread` — синглтон, живущий столько же, сколько и процесс, и именно он является опорной точкой, через которую проходит вся цепочка восстановления.

ViewModel выживает не потому, что её кто-то “сохраняет” — а потому, что никто её не убивает. Пока жив `ActivityThread`, жив и `ViewModelStore`.

Позже мы снова вернемся к `ActivityThread` и `ActivityClientRecord`, это будет в рамках следующих статей.

ViewModel в Fragment под капотом: от ViewModelStore до Retain-фрагментов

В предыдущей статье мы рассмотрели ViewModelStore ([ViewModel под капотом: как она выживает при пересоздании Activity](#)) и изучили полный путь от создания ViewModel до его хранения в ViewModelStore. Мы выяснили, где хранится сам ViewModelStore, но рассматривали это в контексте ComponentActivity и его родителя Activity.

А как обстоят дела у Fragment-ов? В этой статье мы ответим на вопрос:

Где хранятся ViewModelStore для Fragment-ов и как Retain-фрагменты переживают изменение конфигурации?

Вводная

ViewModelStore — это класс, который содержит внутри себя коллекцию Map<String, ViewModel>. ViewModel-и хранятся в этой коллекции по ключу, а ViewModelStoreOwner — в лице Fragment, ComponentActivity и NavBackStackEntry может очистить их при необходимости.

Fragment(Фрагменты) — это части UI, которые могут жить внутри активности или в другом фрагменте, обеспечивая гибкость и переиспользуемость интерфейса. Фрагменты управляются активностью и её жизненным циклом, а навигация часто строится на базе фрагментов с использованием подхода SingleActivity. Прямые наследники — DialogFragment, BottomSheetDialogFragment и AppCompatActivity — используются для отображения диалогов и нижних листов.

Retain Fragment (@Deprecated) — это фрагмент, который сохраняется при изменении конфигурации активности, вместо того чтобы пересоздаваться. Это достигается вызовом метода setRetainInstance(true) у Fragment, который указывает системе не уничтожать фрагмент при пересоздании активности.

Раньше механизм Retain Fragment использовался для хранения данных и фоновых операций, так как если жив фрагмент, то живы все его данные. Но сейчас он

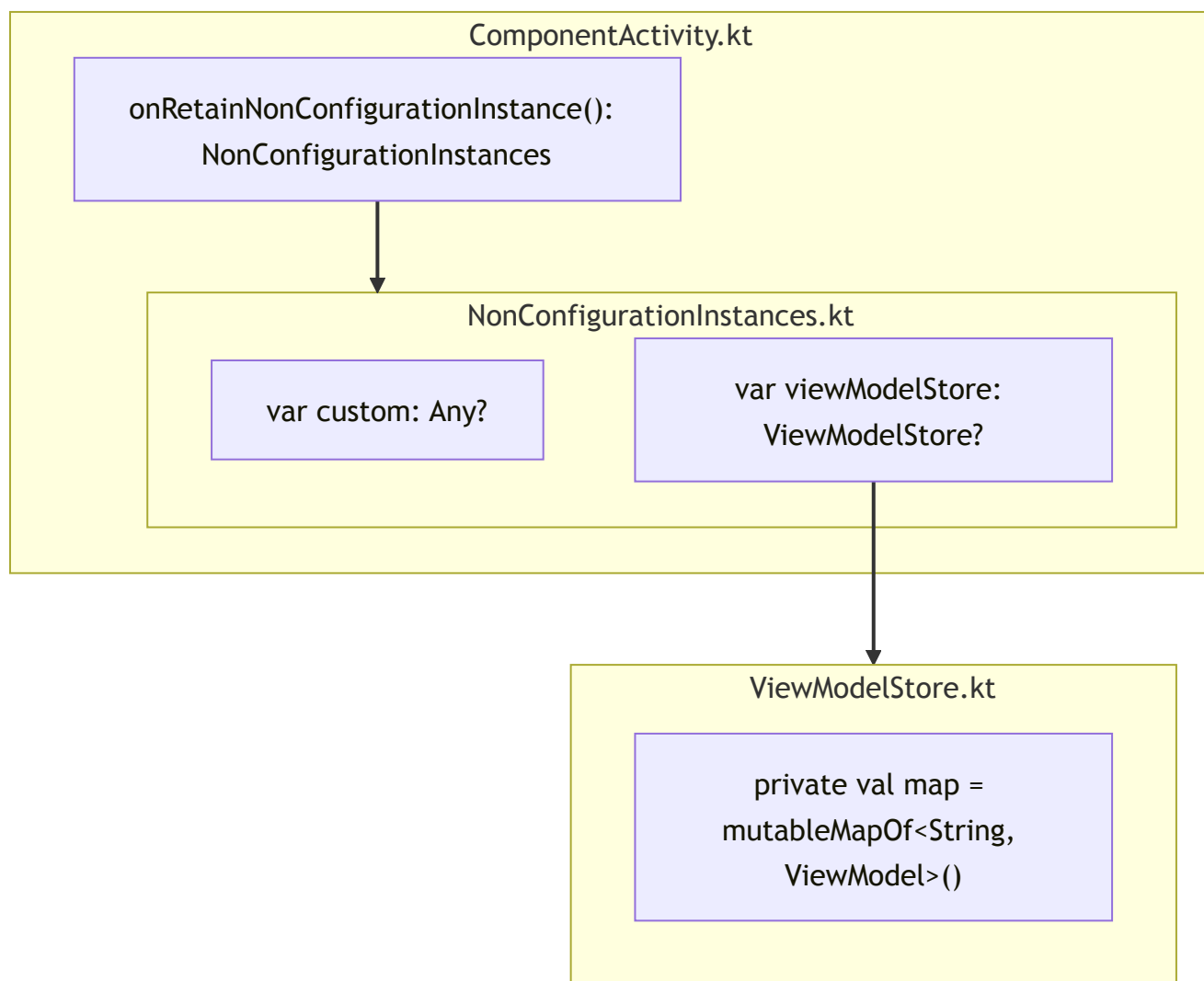
считается устаревшим и не рекомендуется к использованию. В современных приложениях его заменяет `ViewModel`.

Как сохраняется ViewModelStore у Fragment?

В этой статье я рассчитываю, что вы уже ознакомились со статьей [ViewModelStore \(ViewModel под капотом: как она выживает при пересоздании Activity\)](#).

В предыдущей статье мы **детально** рассмотрели процесс сохранения `ViewModelStore` для `Activity`. Цепочка вызовов содержала все шаги до конечной точки `ActivityThread` и даже выше.

Однако в случае `Fragment`-ов цепочка вызовов к счастью **короче** и проще. Поэтому мы рассмотрим сохранение `ViewModelStore` для `Fragment` и `Retain Fragment` отталкиваясь от следующей диаграммы и дополним ее для `Fragment`-ов:



Начнём работу с фрагментами. В этой статье мы не будем углубляться в работу `FragmentManager` и транзакциями — вместо этого сосредоточимся на том, **где и как** хранятся `ViewModel` и `ViewModelStore` в случае с фрагментами.

Как мы знаем, фрагменты не существуют сами по себе — они запускаются внутри **активности** или даже **внутри других фрагментов**.

Рассмотрим простой пример `Activity`, которая добавляет фрагмент в контейнер(`FrameLayout`):

```
class MainActivity : AppCompatActivity() {  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        ...  
        supportFragmentManager  
            .beginTransaction()  
            .add(R.id.frameLayoutContainer, FirstFragment())  
            .commit()  
    }  
}
```



Важно: Код в статье предназначен исключительно для демонстрации и **не претендует на best practices**. Примеры упрощены для лучшего понимания.

Теперь имея `Activity` и транзакцию создадим сам фрагмент и инициализируем в нём `ViewModel` стандартным способ:

```
class FirstFragment : Fragment() {  
  
    private lateinit var viewModel: MyViewModel  
    ...  
    override fun onViewCreated(view: View, savedInstanceState:  
Bundle?) {  
        super.onViewCreated(view, savedInstanceState)  
  
        viewModel = ViewModelProvider.create(owner =  
this).get(MyViewModel::class)
```

```

    }
}

```

Здесь, как и в предыдущих примерах (в прошлой статье), используется `ViewModelProvider.create`, который требует в качестве параметра **owner**. Это означает, что класс `Fragment` должен реализовывать некий интерфейс, позволяющий ему выступать в роли владельца `ViewModel`. Таким интерфейсом является `ViewModelStoreOwner`, который реализуют такие классы, как `Fragment`, `ComponentActivity` и `NavBackStackEntry`.

В исходном коде метода `create` у `ViewModelProvider` явно требуется именно этот интерфейс. Поскольку `ViewModelProvider` был переписан для KMP, его `expect`-объявление находится в `commonMain`:

```

public expect class ViewModelProvider {
    ....
    public companion object {
        public fun create(
            owner: ViewModelStoreOwner,
            factory: Factory =
                ViewModelProviders.getDefaultFactory(owner),
            extras: CreationExtras =
                ViewModelProviders.getDefaultCreationExtras(owner),
        ): ViewModelProvider
    }
}

```

Раз мы это выяснили, давайте сразу посмотрим, как **Fragment** реализует интерфейс `ViewModelStoreOwner`. Это важно, потому что такие классы, как `DialogFragment`, `BottomSheetDialogFragment`, `AppCompatActivity` — наследуются от `Fragment`, и среди них только он реализует этот интерфейс:

```

@NonNull
@Override
public ViewModelStore getViewModelStore() {
    if (mFragmentManager == null) {
        throw new IllegalStateException("Can't access ViewModels
            from detached fragment");
    }
}

```

```

    }
    if (getMinimumMaxLifecycleState() ==
Lifecycle.State.INITIALIZED.ordinal()) {
        throw new IllegalStateException("Calling
getViewModelState() before a Fragment "
            + "reaches onCreate() when using
setMaxLifecycle(INITIALIZED) is not "
            + "supported");
    }
    return mFragmentManager.getViewModelState(this);
}

```

Как видим, фрагмент для получения своего `ViewModelStore` обращается к `FragmentManager` и запрашивает у него нужный `ViewModelStore`, передавая **самого себя** в качестве ключа:

```

...
    return mFragmentManager.getViewModelState(this);
...

```



Напоминание:

`FragmentManager` — это основной компонент, управляющий фрагментами. Он управляет их стеком и позволяет добавлять фрагменты в back stack.

Далее нас интересует метод `getViewModelState`, который есть у класса `FragmentManager.java`:

```

@NonNull
ViewModelStore getViewModelState(@NonNull Fragment f) {
    return mNonConfig.getViewModelState(f);
}

```

Оказывается, тут есть ещё один вложенный вызов: у объекта `mNonConfig` вызывается метод `getViewModelState`, куда передаётся фрагмент в качестве ключа. Давайте посмотрим, что это за объект `mNonConfig`:


```
private FragmentManagerViewModel mNonConfig;
```

Вот это интересно: `FragmentManager` использует свою **ViewModel**, чтобы хранить информацию о `ViewModelStore` фрагментов которые он запускал. И это логично — ведь ему нужно как-то сохранять состояние фрагментов и их `ViewModel`-и при изменениях конфигурации.

Итак, мы выяснили следующий стек вызовов (по порядку):

1. `ViewModelProvider.create(owner = this).get(MyViewModel::class)`
2. `Fragment.getViewModelStore()`
3. `FragmentManager.getViewModelStore(fragment)`
4. `FragmentManagerViewModel.getViewModelStore(fragment)`

Поэтому дальше нас будет интересовать класс `FragmentManagerViewModel`. Свой путь начнем с его вызова метода

`FragmentManagerViewModel.getViewModelStore(fragment):`

FragmentManagerViewModel.java:

```
final class FragmentManagerViewModel extends ViewModel {
    ...

    @NonNull
    ViewModelStore getViewModelStore(@NonNull Fragment f) {
        ViewModelStore viewModelStore =
mViewModelStores.get(f.mWho);
        if (viewModelStore == null) {
            viewModelStore = new ViewModelStore();
            mViewModelStores.put(f.mWho, viewModelStore);
        }
        return viewModelStore;
    }
    ...
}
```

Как это работает? Внутри `FragmentManagerViewModel` есть коллекция `HashMap<String, ViewModelStore>()`, которая хранит `ViewModelStore` для каждого фрагмента, принадлежащего `FragmentManager`'у. То есть все фрагменты, которые были добавлены с помощью `FragmentManager`'а — при попытке получить `ViewModelStore`, сначала ищут его по ключу (`f.mWho`).

Если `ViewModelStore` не найден — это означает, что фрагмент впервые внутри себя создает `ViewModel`, и, соответственно, впервые ему требуется `ViewModelStore`. В этом случае `ViewModelStore` создается и помещается в `HashMap mViewModelStores`.

```
final class FragmentManagerViewModel extends ViewModel {  
    ...  
    private final HashMap<String, Fragment> mRetainedFragments =  
new HashMap<>();  
    private final HashMap<String, FragmentManagerViewModel>  
mChildNonConfigs = new HashMap<>();  
    private final HashMap<String, ViewModelStore> mViewModelStores  
= new HashMap<>();  
    ...  
}
```

⚠ `mViewModelStores` — это `HashMap`, в которой хранятся `ViewModelStore` всех фрагментов, находящихся * *внутри Activity или вложенных в родительский фрагмент* *. Каждый `ViewModelStore` связан с конкретным фрагментом по его уникальному ключу (`fragment.mWho`) и используется для хранения `ViewModel`, привязанных к жизненному циклу соответствующего фрагмента.

Что нам известно в данный момент? Когда мы создаем `ViewModel` внутри нашего `Fragment`'а, то его `ViewModelStore` хранится внутри `FragmentManager`, точнее — внутри его `FragmentManagerViewModel`'ки.

Вроде бы всё ясно: наша `ViewModel` хранится внутри `ViewModelStore`, который сам хранится внутри `FragmentManagerViewModel` (который тоже является `ViewModel`). И тут возникает логичный вопрос — **а где хранится сам**

FragmentManagerViewModel? Он ведь тоже `ViewModel`, а значит должен храниться внутри какого-то `ViewModelStore`.

Краткий ответ: он хранится внутри `ViewModelStore`, который принадлежит самой `Activity`.

Хочешь убедиться? Тогда читай дальше.

Чтобы ответить на наш вопрос, начнём с основ — с того, как работают фрагменты и откуда берётся `FragmentManager`. Но перед этим давайте взглянем на иерархию всех существующих видов `Activity`, чтобы понять, с какой цепочки мы начнём работу:

Иерархия Activity:

```
Activity
├── ComponentActivity
│   ├── FragmentActivity
│       └── AppCompatActivity
```

Класс	Назначение
<code>Activity</code>	Базовый низкоуровневый класс экрана в Android SDK. Напрямую и использовать не рекомендуется.
<code>ComponentActivity</code>	Современная основа для Jetpack компонентов: <code>ViewModel</code> , <code>SavedState</code> , <code>ActivityResult API</code> , <code>OnBackPressedDispatcher</code>
<code>FragmentActivity</code>	Добавляет поддержку фрагментов (через <code>AndroidX</code>). Фрагменты из <code>android.app.Fragment</code> больше не поддерживаются.
<code>AppCompatActivity</code>	Поддержка старых версий Android с <code>Deprecated Api</code> , <code>AppCompatActivity</code> , <code>ActionBar</code> , тем <code>AppCompat</code> , <code>Material UI</code> , .

Как вы наверняка догадались, нас будет интересовать именно `FragmentActivity`. `FragmentActivity` — это базовый класс, предоставляющий интеграцию с системой фрагментов. Именно он отвечает за создание и управление `FragmentManager`. На его основе построен и более часто используемый `AppCompatActivity`, который

расширяет функциональность за счёт поддержки компонентов из библиотеки поддержки (AppCompat).

Именно `FragmentActivity` (или его наследник `AppCompatActivity`) позволяет полноценно работать с фрагментами и `FragmentManager`. Остальные способы взаимодействия с фрагментами считаются устаревшими.

Рассмотрим исходный код `FragmentActivity`:

`FragmentActivity.java`

```
public class FragmentActivity extends ComponentActivity {
    ...
    final FragmentController mFragments =
    FragmentController.createController(new HostCallbacks());

    class HostCallbacks extends
    FragmentHostCallback<FragmentActivity> implements
    ViewModelStoreOwner {
        ...

        public ViewModelStore getViewModelStore() {
            return FragmentActivity.this.getViewModelStore();
        }
        ...
    }
    ...
}
```



`HostCallbacks` реализует множество интерфейсов помимо `ViewModelStoreOwner`, но в статье они опущены, чтобы не отвлекать от сути.

Мы видим переменную `mFragments`, которая имеет тип `FragmentController`. Этой переменной присваивается результат вызова статического метода `createController`, куда передаётся новый экземпляр `HostCallbacks`.

`HostCallbacks` — это класс, реализующий интерфейс `ViewModelStoreOwner`. В своём методе `getViewModelStore()` он возвращает `ViewModelStore`,

принадлежащий самому `FragmentManager`.

Кроме того, `HostCallbacks` наследуется от класса `FragmentManager.Callback`, который выглядит следующим образом:

```
@SuppressWarnings("deprecation")
abstract class FragmentManager.Callback internal constructor(
    ...
) : FragmentContainer() {

    @get:RestrictTo(RestrictTo.Scope.LIBRARY)
    val fragmentManager: FragmentManager = FragmentManagerImpl()
    ...
}
```

⚠ `FragmentManager.Callback` был переписан с Java на Kotlin, начиная с версии `androidx.fragment:fragment:*:1.7.0-beta01`.

Внутри `FragmentManager.Callback` создаётся объект `FragmentManager`. Зная это, возвращаемся к исходникам `FragmentManager`, где есть поле `mFragments`:

```
final FragmentController mFragments =
    FragmentController.createController(new HostCallbacks());
```

Здесь создаётся объект `HostCallbacks`, который наследуется от `FragmentManager.Callback` и реализует интерфейс `ViewModelStoreOwner`, в конечном итоге возвращая `ViewModelStore`, принадлежащий самой активности.

Посмотрим на исходники статического метода `FragmentController.createController()`:

```
public class FragmentController {

    private final FragmentManager.Callback mHost;

    /**
     * Returns a {@link FragmentController}.
     */
}
```

```

    @NonNull
    public static FragmentController createController(@NonNull
FragmentHostCallback<?> callbacks) {
        return new FragmentController(checkNotNull(callbacks,
"callbacks == null"));
    }

    private FragmentController(FragmentHostCallback<?> callbacks)
{
    mHost = callbacks;
}

}

```

Мы видим, что внутри `FragmentActivity` создаётся `FragmentController` посредством вызова метода `createController()`. Метод принимает объект `FragmentHostCallback` — в нашем случае это подкласс `HostCallbacks`, который реализует `ViewModelStoreOwner` и предоставляет `ViewModelStore` самой активности.

Чтобы лучше понять цепочку создания и передачи зависимостей, посмотрим на схему:

```

FragmentActivity
├─ Has a → FragmentController (mFragments)
│   └─ Created with → HostCallbacks
│       └─ Implements → ViewModelStoreOwner (delegates
to FragmentActivity)
│           └─ Extends → FragmentHostCallback
│               └─ Has a → FragmentManagerImpl (as
fragmentManager)

```

Эта структура позволяет `FragmentActivity` делегировать управление фрагментами специальному помощнику — `FragmentController`. Таким образом, `FragmentActivity` не занимается напрямую логикой работы с фрагментами, но при этом сохраняет доступ к ключевым компонентам: `FragmentManager` и `ViewModelStore`, благодаря вспомогательному классу `HostCallbacks`.

Теперь давайте подробнее рассмотрим, как создаётся и инициализируется `FragmentManager`. Обратим внимание на следующую строку:

```
final FragmentController mFragments =  
FragmentManager.createController(new HostCallbacks());
```

Здесь создаётся экземпляр `FragmentManager`, которому в качестве параметра передаётся объект `HostCallbacks`. Именно этот объект предоставляет необходимые зависимости, такие как `FragmentManager`.

Далее обратимся к конструктору `FragmentActivity`. В нём вызывается метод `init`, внутри которого регистрируется слушатель `OnContextAvailableListener`. Этот слушатель срабатывает, когда контекст становится доступен, и в этот момент вызывается метод `attachHost` у `FragmentManager`:

FragmentManager.java:

```
public class FragmentActivity extends ComponentActivity {  
    ...  
    final FragmentController mFragments =  
    FragmentManager.createController(new HostCallbacks());  
  
    public FragmentActivity() {  
        super();  
        init();  
    }  
  
    private void init() {  
        ...  
        addOnContextAvailableListener(context ->  
mFragments.attachHost(null /*parent*/));  
    }  
}
```

Теперь заглянем внутрь самого метода `attachHost`, который реализован в классе `FragmentManager`.

FragmentManager.java:

```

/**
 * Attaches the host to the FragmentManager for this controller.
 * The host must be
 * attached before the FragmentManager can be used to manage
 * Fragments.
 */
public void attachHost(@Nullable Fragment parent) {
    mHost.getFragmentManager().attachController(
        mHost, mHost /*container*/, parent);
}

```

Внутри этого метода вызывается `getFragmentManager()` у переменной `mHost`. Эта переменная представляет собой объект типа `FragmentHostCallback<?>`, а если точнее, то передается именно его наследник – объект `HostCallbacks`. Получив `FragmentManager`, у него вызывается метод `attachController`, которому передаются: сам `HostCallbacks` как хост, он же как контейнер, и опционально — родительский фрагмент (в данном случае `null`).

Сама переменная `mHost`, используемая внутри `FragmentController`, выглядит следующим образом:

FragmentController.java:

```
private final FragmentHostCallback<?> mHost;
```

На этапе инициализации `FragmentActivity` создается экземпляр `FragmentController`, которому делегируется управление фрагментами. Этот контроллер получает в конструктор объект `HostCallbacks`, обеспечивая тем самым связку между `FragmentManager` и жизненным циклом активности.

Мы уже вскользь рассмотрели, как инициализируется эта переменная, но давай коротко повторим:

Класс `HostCallbacks` — это внутренний класс `FragmentActivity`, который наследуется от `FragmentHostCallback` и одновременно реализует интерфейс `ViewModelStoreOwner`. Когда создается объект `FragmentController`, он получает в качестве параметра экземпляр `HostCallbacks`. Этот объект сохраняется во внутреннем поле `mHost` типа `FragmentHostCallback<?>`.

Поскольку `HostCallbacks` является потомком `FragmentManagerCallbacks`, ему также доступны методы родителя — в частности, `getFragmentManager()` (точнее, поле `fragmentManager`, полученное через геттер). В Java оно вызывается как `getFragmentManager()`, хотя в Kotlin это просто свойство. Далее мы уже можем передавать `mHost` в методы `FragmentManager`.

Теперь давай посмотрим, как именно `FragmentManager` получает доступ к `FragmentManagerViewModel`. Это происходит в методе `attachController`, который вызывается внутри `FragmentManager`:

FragmentManager.java:

```
void attachController(@NonNull FragmentHostCallback<?> host,
                     @NonNull FragmentContainer container,
                     @Nullable final Fragment parent) {
    ...
    // Get the FragmentManagerViewModel
    if (parent != null) {
        mNonConfig =
parent.mFragmentManager.getChildNonConfig(parent);
    } else if (host instanceof ViewModelStoreOwner) {
        ViewModelStore viewModelStore = ((ViewModelStoreOwner)
host).getViewModelStore();
        mNonConfig =
FragmentManagerViewModel.getInstance(viewModelStore);
    } else {
        mNonConfig = new FragmentManagerViewModel(false);
    }
    ...
}
```



Цепочка инициализации: `FragmentActivity` → `HostCallbacks` → `FragmentManager` → `FragmentManagerViewModel`

Эта последовательность отражает, как создаются и связываются между собой ключевые компоненты фреймворка фрагментов.

Теперь разберём, что именно происходит внутри метода `attachController`:

1. Если parent != null

Это означает, что мы имеем дело с **вложенными фрагментами**, которые управляются через `childFragmentManager`. В таком случае `FragmentManager` обращается к своему полю `mChildNonConfigs`, где хранятся `FragmentManagerViewModel`-ки для вложенных фрагментов. Если нужной `FragmentManagerViewModel` ещё нет, она будет создана и сохранена в `HashMap`, используя идентификатор родительского фрагмента в качестве ключа.

```
private final HashMap<String, FragmentManagerViewModel>
mChildNonConfigs = new HashMap<>();
```

2. Если parent == null, и host instanceof ViewModelStoreOwner

Это основной путь при работе с `FragmentActivity` и `AppCompatActivity`, потому что `HostCallbacks` реализует `ViewModelStoreOwner`. В этом случае `FragmentManager` получает `ViewModelStore`, привязанный к `FragmentActivity`, и передаёт его в `FragmentManagerViewModel.getInstance()`.

Таким образом, `FragmentManagerViewModel` сохраняется в **том же** `ViewModelStore`, что и остальные `ViewModel`-ки `Activity`, и будет жить столько же, сколько и сама `Activity`.

3. Если host не реализует ViewModelStoreOwner

Это **устаревший сценарий**, когда `Activity` напрямую наследуется от `Activity` или `ComponentActivity`, минуя `FragmentActivity/AppCompatActivity`.

В этом случае `FragmentManager` создаёт `FragmentManagerViewModel` без использования `ViewModelStore`. Такая `ViewModel` сохраняется через механизм `NonConfigurationInstances`, который Android применял до появления архитектурных компонентов.

Этот подход уже **не рекомендуется**, и с современными `androidx.fragment.app.Fragment` он **не работает**. Он применим только для старых `android.app.Fragment` и только при активном флаге `setRetainInstance(true)`. Когда мы добавляем фрагмент в активити через `supportFragmentManager`, мы всегда попадаем под **второе условие**, описанное выше: `host instanceof ViewModelStoreOwner`. В этой ситуации `FragmentManager` получает `ViewModelStore`

у `host` (то есть `FragmentActivity`) и передаёт его в метод `FragmentManagerViewModel.getInstance()`.

Внутри этого метода создаётся `FragmentManagerViewModel`, который сохраняется в `ViewModelStore`. Как мы уже говорили ранее, этот `ViewModelStore` принадлежит `FragmentActivity` (или её наследнику `AppCompatActivity`).

FragmentManagerViewModel.java:

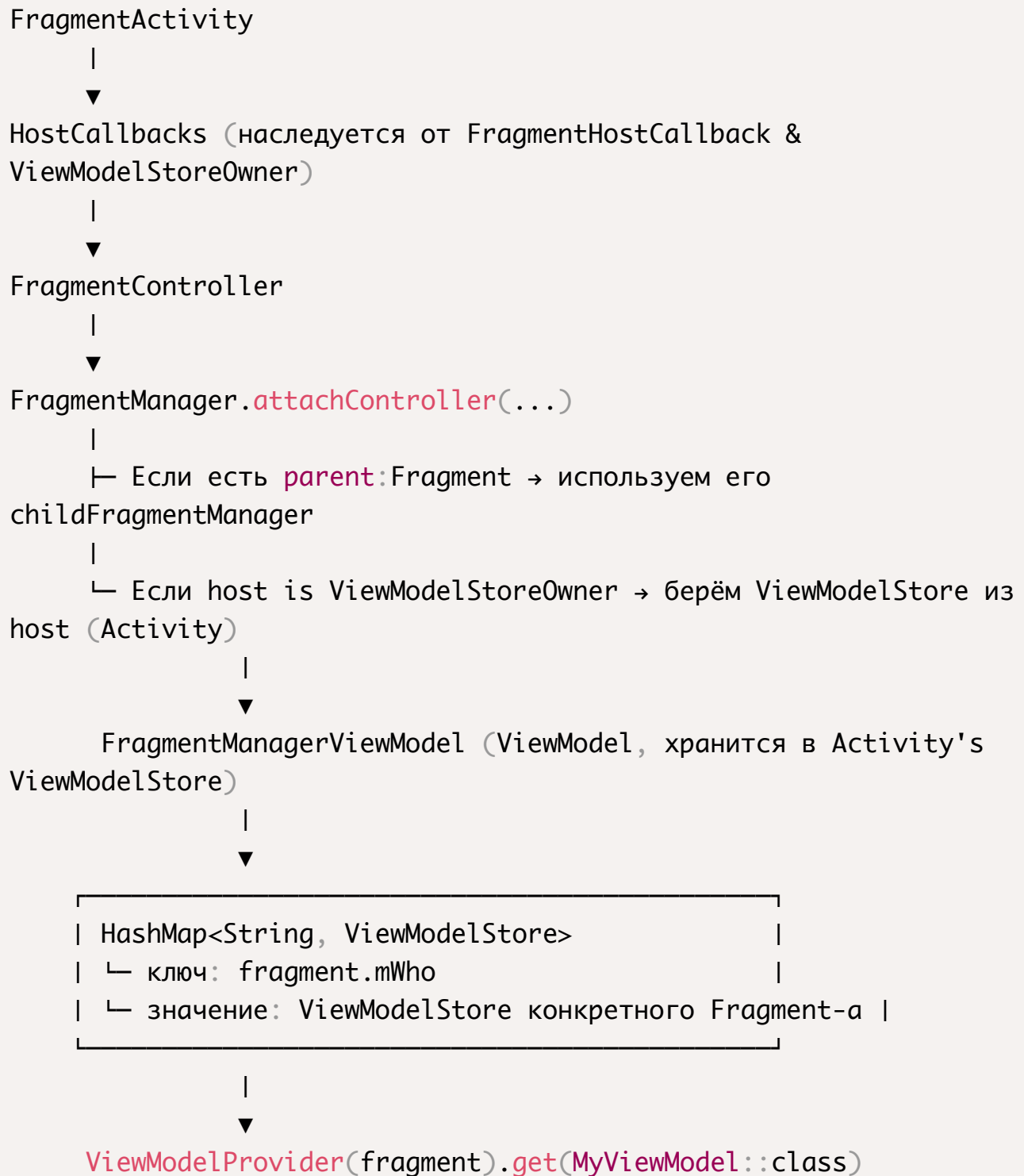
```
@NonNull
static FragmentManagerViewModel getInstance(ViewModelStore
viewModelStore) {
    ViewModelProvider viewModelProvider = new
    ViewModelProvider(viewModelStore, FACTORY);
    return viewModelProvider.get(FragmentManagerViewModel.class);
}
```

Теперь соберём всю цепочку шагов, которая выполняется при создании `ViewModel` внутри фрагмента:

```
viewModel = ViewModelProvider(owner =
this).get(MyViewModel::class)
```

1. `ViewModelProvider` запрашивает у `ViewModelStoreOwner` его `ViewModelStore`. В данном случае `owner = this`, и это фрагмент.
2. У фрагмента вызывается `getViewModelStore()`, поскольку он реализует интерфейс `ViewModelStoreOwner`.
3. Внутри `Fragment.getViewModelStore()` происходит обращение к `FragmentManager`, в котором зарегистрирован этот фрагмент. Вызов: `FragmentManager.getViewModelStore(fragment)`.
4. `FragmentManager` делегирует дальше и обращается к своей `ViewModel`-ке — `FragmentManagerViewModel`.
5. Внутри `FragmentManagerViewModel.getViewModelStore(fragment)` происходит поиск `ViewModelStore` по `fragment.mWho` в `HashMap<String, ViewModelStore>`.

6. Если `ViewModelStore` уже есть, он возвращается. Если нет — создаётся новый, сохраняется в мапу и возвращается.



В упрощённом виде, схема ниже иллюстрирует, как устроено взаимодействие между `Activity`, `FragmentManager` и `ViewModelStore`. Заметьте что это диаграмма

продолжение диаграммы которая была в начале статьи

У нас есть `Activity`, которая наследуется от `FragmentActivity` (а чаще — от его расширенного потомка `AppCompatActivity`). При создании `Activity` инициализируется `FragmentManager`, которому передаётся `FragmentManagerHostCallback` — точнее, его наследник `HostCallbacks`.

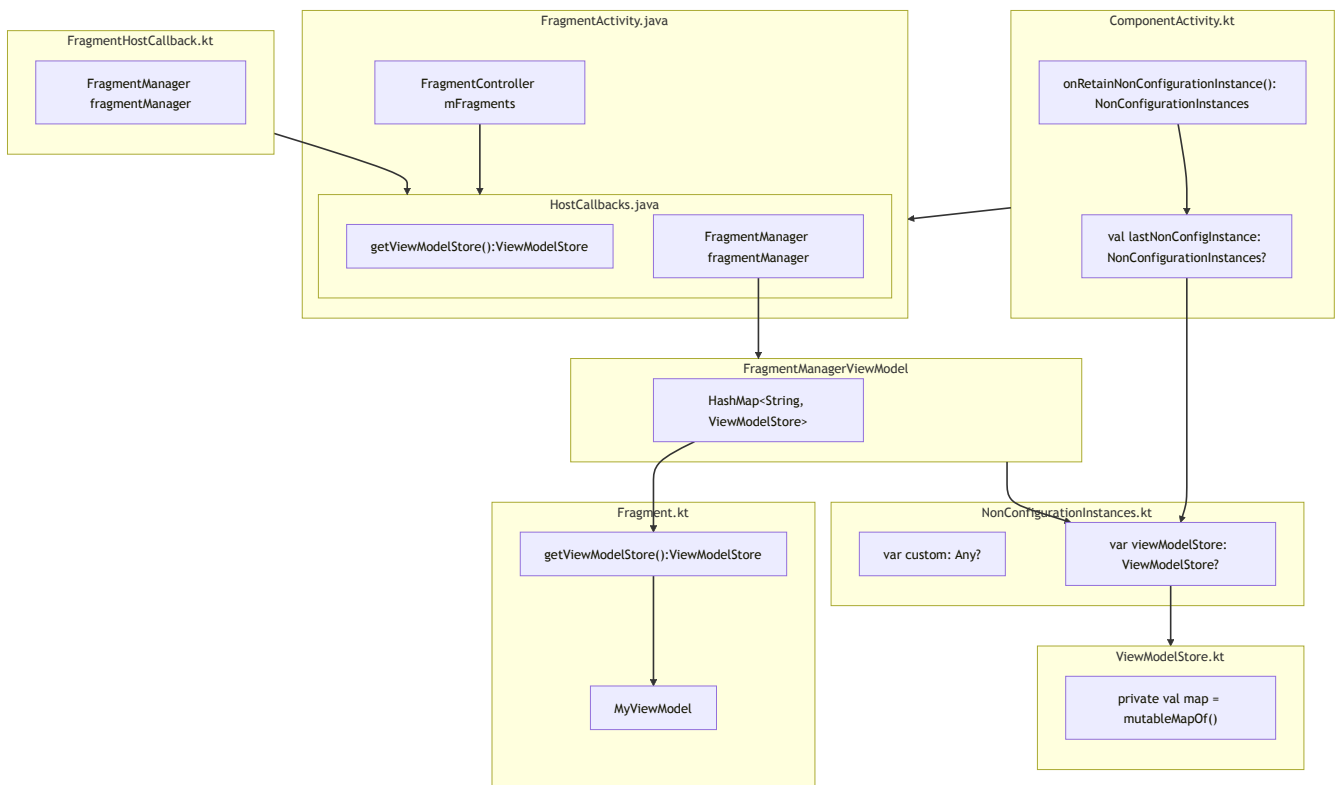
`HostCallbacks` реализует интерфейс `ViewModelStoreOwner`, но при этом **не создаёт** новый `ViewModelStore`, а возвращает уже существующий — тот, что принадлежит `Activity`.

Далее `FragmentManager` прикрепляет `FragmentManagerViewModel` к своему хосту (`Activity` или `ParentFragment`). `FragmentManager` создаёт `FragmentManagerViewModel` и сохраняет его во `ViewModelStore`, предоставленном `HostCallbacks`, то есть — в `ViewModelStore`, принадлежащем `Activity`.

Теперь, когда внутри `Activity` мы добавляем фрагмент через `supportFragmentManager`, инициализация `MyViewModel` во фрагменте приводит к тому, что `ViewModelProvider` запрашивает у фрагмента его `ViewModelStore`.

Фрагмент, в свою очередь, обращается к своему `FragmentManager` — *"дай мой `ViewModelStore`"*. `FragmentManager`, имея прямую ссылку на `FragmentManagerViewModel`, запрашивает у него `ViewModelStore` по ключу (обычно это `fragment.mWho`) — и возвращает `ViewModelStore`, связанный с этим фрагментом.

Именно туда, в этот `ViewModelStore`, и будет помещён `MyViewModel`.



Наконец, давайте убедимся, что `FragmentManagerViewModel`, привязанный к `FragmentManager` активности, действительно хранится внутри `ViewModelStore`, который принадлежит самой активности. Для этого в методе `onCreate()` можно залогировать все ключи, содержащиеся в `viewModelStore` активности:

```

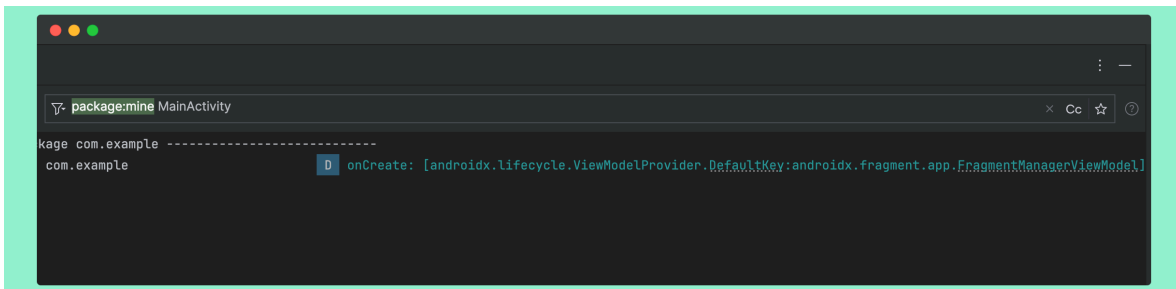
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    supportFragmentManager
        .beginTransaction()
        .add(R.id.frameLayoutContainer, FirstFragment())
        .commit()

    Log.d("MainActivity", "onCreate: ${viewModelStore.keys()}")
    // Output: onCreate:
    [androidx.lifecycle.ViewModelProvider.DefaultKey:androidx.fragment
    .app.FragmentManagerViewModel]
}

```

Скриншот: ключ `FragmentManagerViewModel`, зарегистрированный в `ViewModelStore` активности



Screenshot

На этом этапе мы полностью проследили весь флоу в случае, когда у нас есть `Activity`, поверх которой запускается `Fragment`, и внутри этого фрагмента инициализируется `ViewModel`. Мы дошли до конечной точки — увидели, где именно хранятся `ViewModel`-ы.

Вложенные фрагменты и `childFragmentManager`

Остался один важный кейс — **вложенные фрагменты**. То есть ситуация, когда мы запускаем один `Fragment` внутри другого с помощью `childFragmentManager`. До сих пор мы рассматривали только добавление фрагмента через `FragmentManager` активности (`supportFragmentManager`).

Напомню, мы уже сталкивались с этим кейсом при разборе метода `attachController()`, в котором реализуется логика выбора источника `FragmentManagerViewModel`.

FragmentManager.java:

```
void attachController(@NonNull FragmentHostCallback<?> host,
                     @NonNull FragmentContainer container,
                     @Nullable final Fragment parent) {
    ...
    // Получение FragmentManagerViewModel
    if (parent != null) {
        mNonConfig =
parent.mFragmentManager.getChildNonConfig(parent);
    } else if (host instanceof ViewModelStoreOwner) {
        ViewModelStore viewModelStore = ((ViewModelStoreOwner)
```

```

host).getViewModelStore();
    mNonConfig =
FragmentManagerViewModel.getInstance(viewModelStore);
    } else {
        mNonConfig = new FragmentManagerViewModel(false);
    }
    ...
}

```

В случае, когда мы добавляем фрагмент поверх другого фрагмента через `childFragmentManager`, создавая вложенность, срабатывает первое условие, а именно — проверка `parent != null`. Ранее мы уже выяснили, в каких случаях это условие выполняется, но для понимания продублируем ещё раз:

Если `parent != null`

Это означает, что мы имеем дело с **вложенными фрагментами**, которые управляются через `childFragmentManager`. В таком случае `FragmentManager` обращается к своему полю `mChildNonConfigs`, где хранятся `FragmentManagerViewModel` для вложенных фрагментов. Если нужной `FragmentManagerViewModel` ещё нет, она создаётся и сохраняется в `HashMap`, используя `fragment.mWho` родительского фрагмента в качестве ключа.

При таком кейсе `FragmentManager` обращается к `parent`, вызывает у него метод `getChildNonConfig`, и попадает в следующий код: **FragmentManager.java**:

```

@NonNull
private FragmentManagerViewModel getChildNonConfig(@NonNull
Fragment f) {
    return mNonConfig.getChildNonConfig(f);
}

```

Здесь `mNonConfig` — это `FragmentManagerViewModel`, привязанный к родительскому `FragmentManager`. У него вызывается `getChildNonConfig(f)`, и происходит следующее в **FragmentManagerViewModel.java**

```

final class FragmentManagerViewModel extends ViewModel {

    private final HashMap<String, Fragment> mRetainedFragments =

```



```

new HashMap<>();
    private final HashMap<String, FragmentManagerViewModel>
mChildNonConfigs = new HashMap<>();
    private final HashMap<String, ViewModelStore> mViewModelStores
= new HashMap<>();

    @NonNull
    FragmentManagerViewModel getChildNonConfig(@NonNull Fragment
f) {
        FragmentManagerViewModel childNonConfig =
mChildNonConfigs.get(f.mWho);
        if (childNonConfig == null) {
            childNonConfig = new
FragmentManagerViewModel(mStateAutomaticallySaved);
            mChildNonConfigs.put(f.mWho, childNonConfig);
        }
        return childNonConfig;
    }
}

```

В этом методе мы пытаемся получить `FragmentManagerViewModel` для `childFragmentManager` родительского фрагмента, чтобы у `childFragmentManager` была собственная `FragmentManagerViewModel`, в которой можно будет хранить `ViewModelStore` всех фрагментов, которые будут запущены внутри `childFragmentManager`.

Если такого `FragmentManagerViewModel` ещё не существует, он создаётся, кладётся в `mChildNonConfigs`, и затем возвращается обратно в метод `attachController`, где продолжает использоваться для инициализации `childFragmentManager`.

Отличный запрос. Вот как можно лаконично и понятно сформулировать это как завершение или рефлексивный блок — с пояснением про дерево `FragmentManagerViewModel` и как оно строится:

Как формируется дерево `FragmentManagerViewModel`

Чтобы понять полную картину, важно представить, как строится иерархия `FragmentManagerViewModel` в реальном приложении:

- В начале у нас есть Activity, у которой есть FragmentManager (чаще всего это supportFragmentManager).
- У этого FragmentManager создаётся собственный FragmentManagerViewModel. Он сохраняется внутри ViewModelStore, который принадлежит самой Activity.

Теперь, если мы добавляем фрагменты через FragmentManager который принадлежит Activity (supportFragmentManager), то для каждого такого фрагмента будет создан свой ViewModelStore. Эти ViewModelStore будут храниться **внутри** FragmentManagerViewModel, связанного с FragmentManager самой Activity, в поле FragmentManagerViewModel#mViewModelStores:

```
final class FragmentManagerViewModel extends ViewModel {
    ...
    private final HashMap<String, ViewModelStore> mViewModelStores
    = new HashMap<>();
    ...
}
```

Каждый такой фрагмент, в свою очередь, тоже имеет собственный FragmentManager — это childFragmentManager. Он используется, если мы хотим внутри фрагмента запускать другие фрагменты (вложенность, локальный стек навигации).

- У childFragmentManager тоже должен быть свой FragmentManagerViewModel (как у всех FragmentManager-ов), чтобы он мог хранить ViewModelStore для фрагментов, запущенных внутри родительского фрагмента, то есть внутри него.
- Эти FragmentManagerViewModel хранятся в mChildNonConfigs — это Map<String, FragmentManagerViewModel> внутри FragmentManagerViewModel родителя.

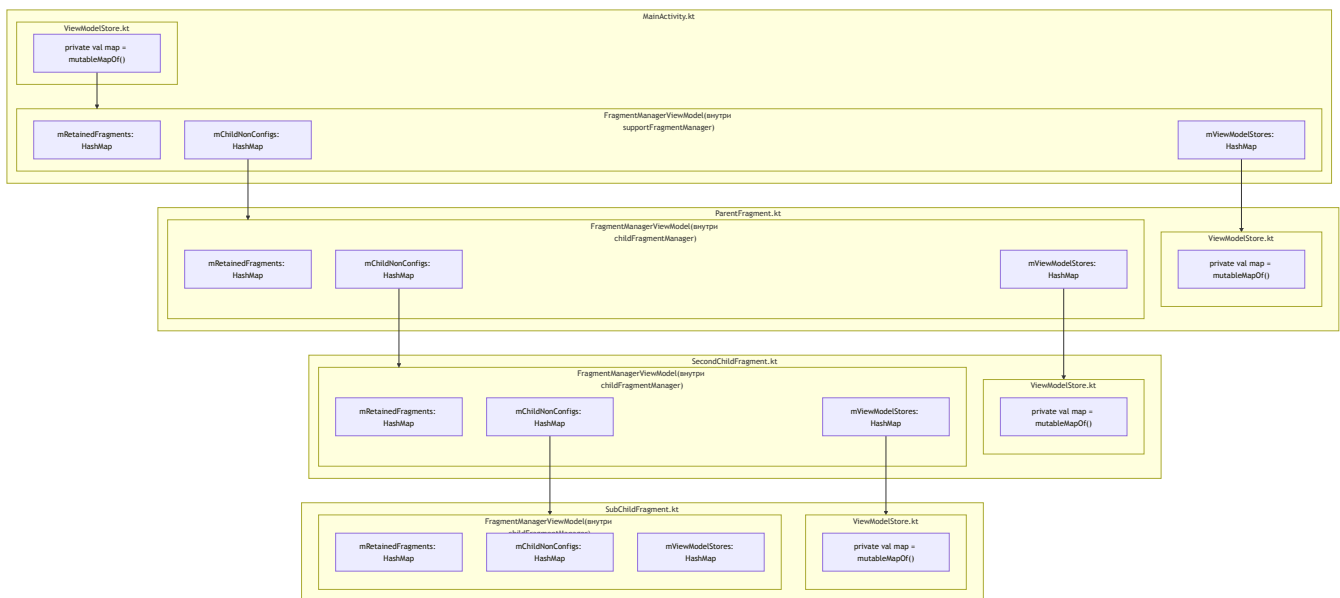
Таким образом, формируется дерево:

- Корень — это FragmentManagerViewModel, привязанный к FragmentManager самой Activity и хранящийся в её ViewModelStore.
- Далее — FragmentManagerViewModel для каждого вложенного

childFragmentManager, сохранённые внутри mChildNonConfigs.

- Это дерево может быть сколь угодно глубоким, повторяя структуру вложенности фрагментов в приложении. Каждый узел в этом дереве это `FragmentManagerViewModel`

Именно такая структура позволяет корректно управлять `ViewModel`, сохраняя их сквозь конфигурационные изменения и обеспечивая жизненный цикл, привязанный к конкретному фрагменту.



Думаю, теперь весь флоу хранения `ViewModelStore` должен быть полностью понятен.

Если у нас есть `FragmentActivity` или `AppCompatActivity`, то у неё есть свой собственный `ViewModelStore`. Когда мы добавляем фрагмент через её `FragmentManager`, для этого фрагмента создаётся отдельный `ViewModelStore`. Этот `ViewModelStore` будет храниться внутри `FragmentManagerViewModel`, который, в свою очередь, лежит внутри `ViewModelStore`, принадлежащего активности.

! `FragmentManagerViewModel` создаётся автоматически при инициализации `FragmentManager` и регистрируется как обычный `ViewModel` в `ViewModelStoreOwner` (например, в активности). Он предназначен именно для хранения `ViewModelStore`-ов всех дочерних фрагментов.

Если мы добавим ещё один фрагмент на тот же уровень — всё повторится: новый `ViewModelStore` → в `FragmentManagerViewModel` → в `ViewModelStore` активности.

Но фишка в том, что каждый фрагмент имеет свой `childFragmentManager`, то есть может быть контейнером для других фрагментов. И `childFragmentManager`, как и любой `FragmentManager`, имеет свой `FragmentManagerViewModel`.

⚠ При каждом вызове `getChildFragmentManager()` фреймворк создаёт или использует уже существующий `FragmentManagerViewModel`. Это гарантирует, что даже при пересоздании фрагмента `ViewModelStore` вложенных фрагментов не теряется.

Это значит: при добавлении вложенных фрагментов, `ViewModelStore` каждого из них будет храниться во внутренней `FragmentManagerViewModel`, принадлежащей `childFragmentManager` родительского фрагмента.

⚠ Внутри `FragmentManagerViewModel` используются ключи `Fragment.mWho`, чтобы сохранить и потом правильно восстановить соответствие между `Fragment` и его `ViewModelStore`.

Чем глубже вложенность, тем больше разрастается дерево.

Например:

```
Activity
├── ParentFragment1
│   ├── ParentFragment2
│   │   ├── ChildFragment1
│   │   └── ChildFragment2
```

В таком дереве:

- `ChildFragment1` и `ChildFragment2` — их `ViewModelStore` хранятся в `FragmentManagerViewModel`, принадлежащем `childFragmentManager` `ParentFragment2`.
- `ParentFragment2` — его `ViewModelStore` хранится в `FragmentManagerViewModel`,

принадлежащем `childFragmentManager` `ParentFragment1`.

- `ParentFragment1` — его `ViewModelStore` лежит в `FragmentManagerViewModel` от `supportFragmentManager` активности.
- А сама `FragmentManagerViewModel` из `supportFragmentManager` — хранится в `ViewModelStore` самой активности.

Такой флоу помогает сохранить `ViewModel` даже при сложной навигации и вложенности фрагментов.

⚠ Зачем вся эта сложность? Такой флоу помогает сохранить `ViewModel` даже при сложной навигации и глубокой вложенности фрагментов. Такая структура сохраняет иерархию `ViewModelStore`, обеспечивая корректное восстановление `ViewModel` даже при пересоздании компонентов.

Итак, мы рассмотрели весь флоу хранения `ViewModelStore` для фрагментов. Пора двигаться дальше, ведь тема Retain-фрагментов осталась нераскрытой — поэтому переходим к следующей части статьи.

Как Retain-фрагменты переживают изменение конфигурации?

Напоминаю ещё раз: Retain-фрагменты устарели довольно давно, и на практике их использование не рекомендуется. О них хорошо помнят разработчики, которые ещё писали на Java — Retain-фрагменты существовали задолго до появления `ViewModel`. Когда `ViewModel` стала стандартом, Retain-фрагменты официально объявили устаревшими. Но знать о них всё же полезно. Итак, начнём.

В начале статьи уже было дано определение Retain-фрагментам. А при разборе «внутренностей» `FragmentManagerViewModel` внимательные глаза могли заметить нечто, связанное с Retain-фрагментами — а именно, вот этот блок кода, который появлялся в статье уже не раз:

```
final class FragmentManagerViewModel extends ViewModel {  
    ...  
    private final HashMap<String, Fragment> mRetainedFragments =  
        new HashMap<>();  
}
```

```

    private final HashMap<String, FragmentManagerViewModel>
mChildNonConfigs = new HashMap<>();
    private final HashMap<String, ViewModelStore> mViewModelStores
= new HashMap<>();
    ...
}

```

Здесь есть три поля. Два из них мы уже подробно разобрали:

- `mViewModelStores` — для хранения `ViewModelStore` на одном уровне в дереве,
- `mChildNonConfigs` — для хранения вложенных `FragmentManagerViewModel`, соответствующих дочерним фрагментам / `FragmentManager`.

Но вот поле, которому мы до сих пор не уделяли внимания — это самое верхнее: `mRetainedFragments`. Это коллекция, которая хранит фрагменты по ключу. Стоп... что? Фрагменты **внутри** `ViewModel`?! Именно так.

Все фрагменты, у которых установлен флаг `setRetainInstance(true)`, попадают именно туда. Заинтриговал? Тогда давай разбираться глубже.

Как создать Retain Fragment? Retain-фрагменты — это не какой-то отдельный класс, наследник `Fragment`. Это всё тот же старый добрый `Fragment`, но с активированным флагом `setRetainInstance`:

```

class MyFragment : Fragment() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setRetainInstance(true)
    }
}

```



Так как Retain-фрагменты устарели, метод `setRetainInstance` также помечен аннотацией `@Deprecated`.

С этого момента наш фрагмент становится *Retain*, и он сможет пережить изменение конфигурации — по той же схеме, по которой выживают `ViewModel`. Как именно?

Мы уже немного знаем, но всё же давай проследим путь целиком — от вызова `setRetainInstance()` до хранения внутри `FragmentManagerViewModel#mRetainedFragments`.

Для этого заглянем в исходники метода `setRetainInstance`: **Fragment.java**:

```
@Deprecated
public void setRetainInstance(boolean retain) {
    ...
    if (retain) {
        mFragmentManager.addRetainedFragment(this);
    } else {
        mFragmentManager.removeRetainedFragment(this);
    }
    ...
}
```

Логика простая: если флаг `retain` установлен в `true`, фрагмент передаётся в `FragmentManager` как *Retain* — через метод `addRetainedFragment`. Если `false` — наоборот, удаляется из списка Retain-фрагментов через `removeRetainedFragment`.

Давайте продолжим и заглянем в сам `FragmentManager` и рассмотрим метод его `addRetainedFragment`:

FragmentManager.java:

```
void addRetainedFragment(@NonNull Fragment f) {
    mNonConfig.addRetainedFragment(f);
}
```

Как по старинке, метод передает управление на `mNonConfig`, который, как мы уже знаем, является экземпляром `FragmentManagerViewModel`.

FragmentManager.java:

```
private FragmentManagerViewModel mNonConfig;
```

Теперь давайте взглянем на метод `addRetainedFragment` внутри `FragmentManagerViewModel`:

FragmentManagerViewModel.java:

```
void addRetainedFragment(@NonNull Fragment fragment) {
    ...
    if (mRetainedFragments.containsKey(fragment.mWho)) {
        return;
    }
    mRetainedFragments.put(fragment.mWho, fragment);
    ...
}
```

Вот и все: мы разобрались, как фрагмент становится *Retain* и как его хранение работает в `FragmentManagerViewModel`.

Теперь рассмотрим метод удаления фрагмента из Retain-списка, который работает по аналогичному принципу — через тот же flow: **Fragment -> FragmentManager -> FragmentManagerViewModel**:

FragmentManagerViewModel.java:

```
void removeRetainedFragment(@NonNull Fragment fragment) {
    ...
    boolean removed = mRetainedFragments.remove(fragment.mWho) !=
    null;
    ...
}
```

Осталось понять как же потом эти фрагменты восстанавливаются после изменения конфигураций, одного их хранения не достаточно ведь их нужно обратно вернуть после того как Activity пересоздается, все фрагменты пересоздаются, FragmentManager тоже, но Retain фрагменты не должны пересоздаваться, а должны браться из `mRetainedFragments`, мы уже в начале статьи видели метод `attachController` у `FragmentManager`:

```
@SuppressWarnings("SyntheticAccessor")
void attachController(@NonNull FragmentHostCallback<?> host,
                     @NonNull FragmentContainer container,
                     @Nullable final Fragment parent) {
```



```

    ...

    if (savedInstanceState != null) {
        restoreSaveStateInternal(savedInstanceState);
    }
    ...
}

```

Видим что идет обращение к методу `restoreSaveStateInternal`:

```

void restoreSaveStateInternal(@Nullable Parcelable state) {
    ...
    Bundle bundle = (Bundle) state;
    ...
    FragmentManagerState fms =
bundle.getParcelable(FRAGMENT_MANAGER_STATE_KEY);
    ...

    for (String who : fms.mActive) {
        ...
        Fragment retainedFragment =
mNonConfig.findRetainedFragmentByWho(fs.mWho);
        ...
        mFragmentManager.makeActive(fragmentStateManager);
        ...
    }
}

```

Нас интересует это строка, очередное обращение к `mNonConfig`:

```

Fragment retainedFragment =
mNonConfig.findRetainedFragmentByWho(fs.mWho);

```

Вот и сам метод `findRetainedFragmentByWho` внутри `FragmentManagerViewModel`:

```

@Nullable
Fragment findRetainedFragmentByWho(String who) {
    return mRetainedFragments.get(who);
}

```

Таким образом, при восстановлении `FragmentManager` и пересоздании `Activity`, **Retain-фрагменты** переживают это пересоздание: они открепляются, а после восстановления `FragmentManager` и `Activity` — снова подключаются.

Ранее я упоминал, что **Retain-фрагменты** существовали до появления `ViewModel`. Но в текущей реализации мы видим, что они переживают пересоздание `Activity` благодаря хранению в `FragmentManagerViewModel`, и именно там они поддерживаются. Но как они работали до появления `ViewModel` в Android?

Кратко напомним: это было во времена `android.app.Fragment`. Сейчас они устарели и заменены на `androidx.fragment.app.Fragment`. В старой реализации механизм напоминал работу с `NonConfigurationInstances`. Если кратко, то для **Retain-фрагментов** в `android.app.Fragment` использовался следующий механизм — они хранились здесь:

```

@Deprecated
public class FragmentManagerNonConfig {
    private final List<Fragment> mFragments;
    private final List<FragmentManagerNonConfig> mChildNonConfigs;

    FragmentManagerNonConfig(List<Fragment> fragments,
        List<FragmentManagerNonConfig> childNonConfigs)
    {
        mFragments = fragments;
        mChildNonConfigs = childNonConfigs;
    }

    /**
     * @return the retained instance fragments returned by a
     * FragmentManager
     */
    List<Fragment> getFragments()
    {

```

```

        return mFragments;
    }

    /**
     * @return the FragmentManagerNonConfigs from any applicable
     * fragment's child FragmentManager
     */
    List<FragmentManagerNonConfig> getChildNonConfigs()
    {
        return mChildNonConfigs;
    }
}

```

Далее объект `FragmentManagerNonConfig` хранился внутри `NonConfigurationInstances` в поле `fragments` и переживал изменения конфигураций ровно по той же схеме, которую мы уже рассмотрели в первой статье:

```

public class Activity extends ContextThemeWrapper ...{

    static final class NonConfigurationInstances {
        Object activity;
        HashMap<String, Object> children;
        FragmentManagerNonConfig fragments;
        ArrayMap<String, LoaderManager> loaders;
        VoiceInteractor voiceInteractor;
    }
}

```

Мы кратко рассмотрели этот механизм, потому что он представляет собой тройное устаревание:

- сами `android.app.Fragment` устарели и были заменены на `androidx.fragment.app.Fragment`;
- концепция **Retain-фрагментов**, которая позволяла фрагментам переживать пересоздание `Activity`, устарела, и теперь вместо неё рекомендуется использовать `ViewModel`;

- способ хранения этих фрагментов через `FragmentManagerNonConfig` также устарел — его заменил более современный механизм с использованием `FragmentManagerViewModel`, несмотря на то, что концепция **Retain-фрагментов** уже не считается актуальной.

Таким образом, это не просто устаревшая реализация, а целая цепочка из трёх устаревших технологий, которые были полностью переработаны в современных версиях Android.

На этом, пожалуй, всё. В этой статье мы рассмотрели некоторые смежные моменты и пересечения, подведём итоги.

ViewModel в Fragment

```
MyViewModel -> ViewModelStore -> FragmentManagerViewModel ->
ViewModelStore(Activity's) ->
ComponentActivity.NonConfigurationInstances ->
Activity.NonConfigurationInstances ->
ActivityThread.ActivityClientRecord
```

Современный способ хранения состояний в `Fragment` основан на использовании `ViewModel`, которая помещается в `ViewModelStore`. Управление этим хранилищем осуществляется через `FragmentManagerViewModel`. В свою очередь, `FragmentManagerViewModel` привязан к `ViewModelStore` активности, которая сохраняет его в `NonConfigurationInstances`. Эта цепочка позволяет сохранять состояние фрагмента даже при изменении конфигурации, избегая пересоздания объектов, которые критичны для долгосрочного хранения данных.

RetainFragment в androidx.fragment.app.Fragment

```
MyRetainFragment -> FragmentManagerViewModel ->
ViewModelStore(Activity's) ->
ComponentActivity.NonConfigurationInstances ->
Activity.NonConfigurationInstances ->
ActivityThread.ActivityClientRecord
```

Термин **RetainFragment** в `androidx.fragment.app.Fragment` — это скорее пережиток старых версий API. В современных реализациях `androidx`, фрагменты с

сохранением состояния через `setRetainInstance(true)` фактически больше не рекомендуется использовать. Вместо этого управление состоянием переместилось в `ViewModel`, которая синхронизируется с жизненным циклом фрагмента через `FragmentManagerViewModel`. Сохранение происходит в `ViewModelStore` активности, которая, как и в первом случае, попадает в `NonConfigurationInstances` при пересоздании `Activity`. Таким образом, **RetainFragment** в классическом понимании уже не используется, его роль полностью взяла на себя связка `Fragment + ViewModel`.

RetainFragment в `android.app.Fragment` (устаревший механизм)

```
MyRetainFragment -> FragmentManagerNonConfig ->
Activity.NonConfigurationInstances ->
ActivityThread.ActivityClientRecord
```

В старой реализации Android, когда использовались `android.app.Fragment`, механизм пересоздания фрагментов реализовывался через `FragmentManagerNonConfig`. Объекты `RetainFragment` помещались в специальный контейнер, который сохранялся в `NonConfigurationInstances`. При пересоздании активности, эта структура восстанавливалась из `ActivityClientRecord` в `ActivityThread`. Этот механизм сейчас полностью устарел и был заменён на использование `ViewModel`, так как это более надёжный и гибкий способ сохранить данные на время изменения конфигурации.

Итоги

Эволюция механизмов хранения состояний в `Fragment` прошла несколько стадий:

1. **`android.app.Fragment`** с `FragmentManagerNonConfig` → полностью устарел, более не поддерживается.
2. **`RetainFragment`** в `androidx.fragment.app.Fragment` → больше не рекомендуется, его заменяет связка с `ViewModel`.
3. **Современный подход** — `ViewModelStore` внутри `FragmentManagerViewModel`, который напрямую привязан к жизненному циклу фрагмента и сохраняется в `Activity`.

Теперь вместо устаревших концепций рекомендуется использовать обычные фрагменты в паре с `ViewModel`, что делает код более предсказуемым и легко поддерживаемым.

ViewModel под капотом: как работает в Compose и View

Это продолжение двух предыдущих статей. Если в первой мы разобрали, где в конечном итоге хранится `ViewModelStore` в случае с `Activity`, а во второй — как это устроено во `Fragment`, то сегодня разберёмся, где хранятся `ViewModel`-и, когда мы используем `Compose` (или даже просто `View`). Особенно когда мы объявляем `ViewModel` прямо внутри `Composable` функций. Но, как всегда, начнём с базиса.

Есть такой подход — **View-based ViewModel scoping**. Что он значит? Мы все знаем стандартную практику, когда у каждого фрагмента или активности есть своя `ViewModel`. Но также существует и менее популярная история — когда у каждой `View` может быть своя собственная `ViewModel`. Насколько это полезно — решать вам. Вы спросите: а при чём тут `Compose`? А я отвечу: дело в том, что `Compose` работает примерно по той же схеме. Давайте начнём с простого примера:

View-based ViewModel scoping — первый взгляд

Создадим кастомную `View`. Пусть это будет `TranslatableTextView`. Для нашего примера не так важно, что именно делает эта вьюха — главное, что мы хотим рассмотреть подход `View-based ViewModel scoping`. Вот как это может выглядеть:

```
class TranslatableTextView(context: Context) :
    AppCompatActivity(context) {

    private val viewModel: TranslatableTextViewViewModel by lazy {
        val owner = findViewTreeViewModelStoreOwner() ?:
            error("ViewModelStoreOwner not found for TranslatableTextView")
        ViewModelProvider.create(owner =
            owner).get(TranslatableTextViewViewModel::class.java)
    }

    fun translateTo(locale: Locale) {
        text = viewModel.getTranslatedText(text.toString(),
            locale)
    }
}
```

```
}  
}
```

Представим, что `TranslatableTextView` умеет переводить текст, как, например, в Telegram. Если бы мы использовали обычную `ViewModel`, пришлось бы дублировать логику на всех экранах, где используется эта `View`. Но благодаря подходу **View-based ViewModel scoping**, у `TranslatableTextView` есть **своя собственная `ViewModel`**.

Что мы здесь видим? – Инициализацию `viewModel` напрямую через `ViewModelProvider` без делегатов, с передачей `ViewModelStoreOwner`. – Простой метод `translateTo`, который принимает `Locale` и обновляет текст вьюхи (`AppCompatActivity`) на переведённый.

Давайте взглянем и на саму `ViewModel`, чтобы пример был полноценным и наглядным:

```
class TranslatableTextViewViewModel : ViewModel() {  
    fun getTranslatedText(currentText: String, locale: Locale):  
    String {  
        // Здесь может быть настоящая локализация  
        return "Translated('$currentText') to  
        ${locale.displayLanguage}"  
    }  
}
```

Теперь снова вернёмся к `TranslatableTextView`, чтобы детальнее рассмотреть инициализацию `ViewModel`. Она выглядит немного необычно:

```
class TranslatableTextView(context: Context) :  
    AppCompatActivity(context) {  
  
    private val viewModel: TranslatableTextViewViewModel by lazy {  
        val owner = findViewTreeViewModelStoreOwner() ?:  
        error("ViewModelStoreOwner not found for TranslatableTextView")  
        ViewModelProvider.create(owner =  
        owner).get(TranslatableTextViewViewModel::class.java)  
    }  
}
```



```
...  
}
```

Первое, что бросается в глаза — это вызов метода `findViewTreeViewModelStoreOwner()`. Он возвращает нам `ViewModelStoreOwner`, а как мы помним, им могут быть только `ComponentActivity`, `Fragment` или `NavBackStackEntry`.

Затем этот `owner` мы передаём в `ViewModelProvider`, чтобы тот создал (или вернул) нужную `ViewModel` и поместил её в `ViewModelStore`. Напомню: `ViewModelStore` — это то место, где живёт и хранится наша `ViewModel`, и доступен он у каждого `ViewModelStoreOwner`.

Давайте заглянем, как устроен сам метод `findViewTreeViewModelStoreOwner()` и каким образом он умеет доставать `ViewModelStoreOwner`:

ViewTreeViewModelStoreOwner.android.kt:

```
/**  
 * Retrieve the [ViewModelStoreOwner] associated with the given  
 [View]. This may be used to retain  
 * state associated with this view across configuration changes.  
 *  
 * @return The [ViewModelStoreOwner] associated with this view  
 and/or some subset of its ancestors  
 */  
@JvmName("get")  
public fun View.findViewTreeViewModelStoreOwner():  
ViewModelStoreOwner? {  
    var currentView: View? = this  
    while (currentView != null) {  
        val storeOwner =  
  
currentView.getTag(R.id.view_tree_view_model_store_owner) as?  
ViewModelStoreOwner  
        if (storeOwner != null) {  
            return storeOwner  
        }  
        currentView =
```

```

currentView.getParentOrViewTreeDisjointParent() as? View
    }
    return null
}

```

Если коротко, то в этом методе происходит следующее: у текущей `View`, на которой вызвали `findViewTreeViewModelStoreOwner`, мы ищем тег с `id` `R.id.view_tree_view_model_store_owner`. Полученное значение приводим к `ViewModelStoreOwner`, и если он не `null` — возвращаем его. А если `null`, то начинаем подниматься вверх по иерархии `View`. Эту работу выполняет метод `getParentOrViewTreeDisjointParent`. В исходники его лезть не будем — он просто возвращает родителя текущей `View` (прямого родителя или не прямого родителя). Поскольку это происходит внутри цикла, мы поднимаемся по иерархии, пока не найдём одного из родителей, имеющий тег `R.id.view_tree_view_model_store_owner` и в котором уже есть `ViewModelStoreOwner`.

На этом, в стиле Кристофера Нолана, временно забываем про этот метод — и посмотрим, как мы будем использовать `TranslatableTextView`:

```

class MainActivity : AppCompatActivity() {

    private val frameRootLayout by lazy {
        findViewById<FrameLayout>(R.id.frameRootLayout) }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        // Привязываем ViewModelStoreOwner к дереву
        ViewView(frameRootLayout)
        frameRootLayout.setViewTreeViewModelStoreOwner(this)

        val translatableView = TranslatableTextView(this)
        translatableView.text = "Hello, world!"
        frameRootLayout.addView(translatableView)

        // Пример использования перевода
        translatableView.translateTo(Locale.ENGLISH)
    }
}

```

```
}  
}
```

Всё довольно просто, да? У нас есть некий layout, у которого root — это `FrameLayout` с id `R.id.frameRootLayout`. Мы находим этот `FrameLayout` и добавляем в него наш кастомный `View`: `TranslatableTextView`. Здесь всё понятно.

Но самое интересное — это вот эта строка:

```
// Привязываем ViewModelStoreOwner к дереву View(frameRootLayout)  
frameRootLayout.setViewTreeViewModelStoreOwner(this)
```

Мы вызываем `setViewTreeViewModelStoreOwner` и передаём в него `this` — то есть саму `Activity`. Как мы знаем, `Activity` реализует интерфейс `ViewModelStoreOwner`, поэтому мы спокойно можем передать её туда, где требуется `ViewModelStoreOwner`.

Вот как выглядит цепочка наследования начиная с интерфейса `ViewModelStoreOwner`:

```
[interface] ViewModelStoreOwner → ComponentActivity →  
FragmentActivity → AppCompatActivity
```

То есть, когда мы передаём `this` из `Activity` в `setViewTreeViewModelStoreOwner`, то передаём полностью валидный `ViewModelStoreOwner`, и всё работает как надо. Но как именно это связывание происходит внутри? За счёт чего потом `findViewTreeViewModelStoreOwner()` находит этого владельца(`ViewModelStoreOwner`)?

Чтобы в этом разобраться, давайте заглянем в исходники метода `setViewTreeViewModelStoreOwner`, который мы ранее уже встретили.

`ViewTreeViewModelStoreOwner.android.kt`:

```
/**  
 * Set the [ViewModelStoreOwner] associated with the given [View].  
 * Calls to [get] from this view or  
 * descendants will return `viewModelStoreOwner`.  
 *  
 * This should only be called by constructs such as activities or
```

```

fragments that manage a view tree
 * and retain state through a [ViewModelStoreOwner]. Callers
should only set a [ViewModelStoreOwner]
 * that will be *stable.* The associated [ViewModelStore] should
be cleared if the view tree is
 * removed and is not guaranteed to later become reattached to a
window.
 *
 * @param viewModelStoreOwner ViewModelStoreOwner associated with
the given view
 */
@JvmName("set")
public fun
View.setViewTreeViewModelStoreOwner(viewModelStoreOwner:
ViewModelStoreOwner?) {
    setTag(R.id.view_tree_view_model_store_owner,
viewModelStoreOwner)
}

```

Рядом также находится метод `findViewTreeViewModelStoreOwner`, с которым мы уже знакомы. Сейчас нас интересует `setViewTreeViewModelStoreOwner`. Как видим, он просто кладёт `viewModelStoreOwner` в виде тега в указанную `View` по ключу `R.id.view_tree_view_model_store_owner`:

```

setTag(R.id.view_tree_view_model_store_owner, viewModelStoreOwner)

```

Все, кто работал с `View`, знают метод `setTag(Object?)`, но помимо этого есть и перегруженный метод:

```

public void setTag(int key, final Object tag) {
    ...
}

```

Этот метод позволяет хранить разные теги по ключам, используя под капотом `SparseArray`. Это важный момент, потому что именно через этот механизм мы и будем передавать `ViewModelStoreOwner`.

Теперь давайте разберёмся, что происходит на практике.

В методе `onCreate` в `Activity` мы вызываем метод `setViewTreeViewModelStoreOwner` для рутовой `View` (`R.id.frameRootLayout`), передавая в качестве параметра `this`, то есть само `Activity`. Это потому, что `Activity` реализует интерфейс `ViewModelStoreOwner`. Мы связываем эту активность с деревом представлений (`View`), чтобы иметь доступ к `ViewModelStore` (так как `Activity` является `ViewModelStoreOwner`).

Далее мы добавляем нашу кастомную `View` (он же `TranslatableTextView`) в этот `frameRootLayout`. Пример:

```
class MainActivity : AppCompatActivity() {

    private val frameRootLayout by lazy {
        findViewById<FrameLayout>(R.id.frameRootLayout) }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        // Привязываем ViewModelStoreOwner к дереву View
        frameRootLayout.setViewTreeViewModelStoreOwner(this)

        val translatableView = TranslatableTextView(this)
        translatableView.text = "Hello, world!"
        frameRootLayout.addView(translatableView)

        // Пример использования перевода
        translatableView.translateTo(Locale.ENGLISH)
    }
}
```

Теперь, что происходит дальше?

Когда мы находимся в нашем кастомном `View`, мы вызываем метод `findViewTreeViewModelStoreOwner`. Этот метод начинает искать тег с ID `R.id.view_tree_view_model_store_owner` в самой вьюшке. Если он не находит нужный

тег, он поднимется по иерархии представлений, пока не найдёт родительский элемент, в котором этот тег присутствует:

```
class TranslatableTextView(context: Context) :
    AppCompatActivity(context) {

    private val viewModel: TranslatableTextViewViewModel by lazy {
        val owner = findViewTreeViewModelStoreOwner() ?:
        error("ViewTreeViewModelStoreOwner not found for
        TranslatableTextView")
        ViewModelProvider.create(owner =
        owner).get(TranslatableTextViewViewModel::class.java)
    }
    ...
}
```

Итак, этот механизм позволяет найти нужный `ViewModelStoreOwner` в дереве представлений, начиная с текущей вьюшки и двигаясь вверх по иерархии до родительского компонента, в котором хранятся `ViewModelStore`.

В нашем случае `findViewTreeViewModelStoreOwner` находит `ViewModelStoreOwner` у родительского view: `FrameLayout(R.id.frameRootLayout)`, и мы получаем `ViewModelStoreOwner` и по умолчанию создаём `ViewModel` вызовом `ViewModelProvider`. В конечном итоге таким образом наша `ViewModel`, которую создали внутри `TranslatableTextView`, будет храниться в `ViewModelStore`, принадлежащей `Activity`.

Теперь вопрос, а почему мы это рассмотрели? И при чём тут `Compose`? Ответ в следующей главе статьи.

Где Compose хранит ViewModel-и?

Давайте возьмём очень простую `ViewModel` и очень простой composable screen. Начнём с `ViewModel`:

```
class MyViewModel : ViewModel() {
    fun getName(): String = "Compose"
}
```

Наша `ViewModel` очень простая, и она нам нужна только в качестве примера, чтобы добраться до сути. Далее, наш Composable Screen:

```
@Composable
fun Greeting(modifier: Modifier = Modifier) {
    val viewModel =
    androidx.lifecycle.viewmodel.compose.viewModel<MyViewModel>()
    Text(
        text = "Hello ${viewModel.getName()}",
        modifier = modifier
    )
}
```

Теперь продолжим:

`viewModel()` — это функция из библиотеки: **`androidx.lifecycle:lifecycle-viewmodel-compose:2.8.7`**. Я специально указал полный путь к функции в примере, чтобы вас не смущало, где она хранится и откуда взялась. С использованием Koin, например, мы могли бы использовать `koinViewModel()` из библиотеки `io.insert-koin:koin-androidx-compose`, или даже `hiltViewModel()` из `androidx.hilt:hilt-navigation-compose`.

Независимо от того, какой метод мы бы использовали для получения `ViewModel` в Compose, все они работают под капотом одинаково, особенно в контексте получения `ViewModelStore`, так как его из воздуха не взять. Поэтому давайте начнём изучение с `androidx.lifecycle.viewmodel.compose.viewModel()`, потому что он был первым, а библиотеки вроде Hilt и Koin для создания `ViewModel` в Compose используют похожий механизм.

Далее, исходники метода `androidx.lifecycle.viewmodel.compose.viewModel` в файле:

`androidx.lifecycle.viewmodel.compose.ViewModel.kt`:


```
@Suppress("MissingJvmstatic")
@Composable
public inline fun <reified VM : ViewModel> viewModel(
    viewModelStoreOwner: ViewModelStoreOwner =
    checkNotNull(LocalViewModelStoreOwner.current) {
        "No ViewModelStoreOwner was provided via
    LocalViewModelStoreOwner"
    },
```

```
...
): VM = viewModel(VM::class, viewModelStoreOwner, key, factory,
extras)
```

Остальные входные параметры нас не интересуют в этой статье, кроме параметра `viewModelStoreOwner`:

```
viewModelStoreOwner: ViewModelStoreOwner =
checkNotNull(LocalViewModelStoreOwner.current) {
    "No ViewModelStoreOwner was provided via
LocalViewModelStoreOwner"
},
```

Далее нас будет интересовать `LocalViewModelStoreOwner.current` - так как он нам предоставляет `ViewModelStore`, судя по всему. `LocalViewModelStoreOwner.current` из названия и синтаксиса сразу понятно, что это `CompositionLocal`:

 `CompositionLocal` — это механизм в `Jetpack Compose`, позволяющий передавать значения по дереву UI без явной передачи через параметры, с доступом к ним через `.current` в любой точке композиции. Для использования необходимо предварительно предоставить значение через `CompositionLocalProvider` или задать его по умолчанию при создании.

Давайте глянем на исходники `LocalViewModelStoreOwner`:

```
/**
 * The CompositionLocal containing the current
 * [ViewModelStoreOwner].
 */
public object LocalViewModelStoreOwner {
    private val LocalViewModelStoreOwner =
        compositionLocalOf<ViewModelStoreOwner?> { null }

    /**
     * Returns current composition local value for the owner or
     * `null` if one has not
```



```

        * been provided nor is one available via
        [findViewTreeViewModelStoreOwner] on the
        * current [androidx.compose.ui.platform.LocalView].
        */
        public val current: ViewModelStoreOwner?
            @Composable
            get() = LocalViewModelStoreOwner.current ?:
findViewTreeViewModelStoreOwner()

    /**
     * Associates a [LocalViewModelStoreOwner] key to a value in a
    call to
     * [CompositionLocalProvider].
     */
    public infix fun provides(viewModelStoreOwner:
ViewModelStoreOwner):
        ProvidedValue<ViewModelStoreOwner?> {
            return
LocalViewModelStoreOwner.provides(viewModelStoreOwner)
        }
    }
}

```

Видим, что `LocalViewModelStoreOwner` — это просто обёртка над настоящим `CompositionLocal`. Мы обращаемся именно к его полю `current`, чтобы прочесть текущее значение. Мы либо попытаемся достать значение из поля `current` у `CompositionLocal` — это означает, что кто-то где-то должен был его `provide`-ить. Если же там пусто, то в таком случае вызывается метод `findViewTreeViewModelStoreOwner`. При обычном сценарии использования из коробки мы попадаем именно под второй кейс, когда вызывается метод `findViewTreeViewModelStoreOwner`. Поэтому далее рассмотрим его исходники:

LocalViewModelStoreOwner.android.kt

```

@Composable
internal actual fun findViewTreeViewModelStoreOwner():
ViewModelStoreOwner? =
    LocalView.current.findViewTreeViewModelStoreOwner()

```

И мы видим, что у другого `CompositionLocal` — `LocalView` вызывается метод `View.findViewTreeViewModelStoreOwner()` — это тот самый метод, который мы уже смотрели в первой части статьи. `LocalView.current` возвращает нам текущий `View`. Текущий `View`? Разве мы не работаем сейчас в `compose`? Откуда взялся текущий `View`? Об этом чуть позже узнаем, что это за `View` и откуда он взялся. Сейчас просто знайте, что под капотом `LocalView.current` нам возвращает текущий `View`, у которого мы можем вызвать extension-функцию `findViewTreeViewModelStoreOwner`, которую мы уже видели в первой части статьи, и положит `ViewModel` в `ViewModelStore`:

`ViewTreeLifecycleOwner.android.kt`

```
/**
 * Retrieve the [ViewModelStoreOwner] associated with the given
 * [View]. This may be used to retain
 * state associated with this view across configuration changes.
 *
 * @return The [ViewModelStoreOwner] associated with this view
 * and/or some subset of its ancestors
 */
@JvmName("get")
public fun View.findViewTreeViewModelStoreOwner():
    ViewModelStoreOwner? {
    var currentView: View? = this
    while (currentView != null) {
        val storeOwner =

currentView.getTag(R.id.view_tree_view_model_store_owner) as?
        ViewModelStoreOwner
        if (storeOwner != null) {
            return storeOwner
        }
        currentView =
currentView.getParentOrViewTreeDisjointParent() as? View
    }
    return null
}
```

Пройдёмся ещё раз по флоу:

Когда мы внутри нашего Composable-функций вызываем любую из extension-функций по созданию viewModel: то ли viewModel из библиотеки `androidx.lifecycle:lifecycle-viewmodel-compose`, или хоть даже `koinViewModel()` из библиотеки `io.insert-koin:koin-androidx-compose`, или даже `hiltViewModel()` из `androidx.hilt:hilt-navigation-compose`, то в конечном итоге мы обращаемся именно к CompositionLocal с названием `LocalViewModelStoreOwner` к его полю `current`. А тот, в свою очередь, либо достаёт значение, которое внутри него хранится, либо обращается к Composable-методу `findViewTreeViewModelStoreOwner`. А тот, в свою очередь, обращается к `LocalView` — это ещё один `CompositionLocal`, у которого есть текущее `View`, и для него запускается extension-метод `View.findViewTreeViewModelStoreOwner`, и происходит поиск по дереву `View` в поисках `ViewModelStoreOwner`. В итоге он его находит, но как? В голове возникают два вопроса:

1. При чём тут View-шки? Почему Compose обращается к `LocalView`, и `LocalView` откуда сам взялся?
2. Из предыдущей главы в статье мы увидели, что прежде чем вызывать метод `View.findViewTreeViewModelStoreOwner()`, до него мы клали `ViewModelStoreOwner` во внутренний тег внутри `FrameLayout`, который являлся рутовым `View` в нашем макете, с помощью метода `setViewTreeViewModelStoreOwner`. Но в примере с Compose мы ничего никуда не клали — как всё это работает само по себе?

Всё довольно просто, разработчики Google позаботились об этом за нас. Обычно в Composable есть два подхода:

1. Когда весь проект на Compose полностью, или как минимум в каждой активити UI-дерево начинается с `setContent{}` , а не с `setContentView`:

```
class MainActivity : ComponentActivity() {  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContent {  
            Greeting(modifier = Modifier.fillMaxWidth())  
        }  
    }  
}
```

```

    }
}
}

```

2. Гибридный UI, где часть на compose, а часть на View. Тогда прибегают к использованию ComposeView:

```

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/linearLayout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <androidx.compose.ui.platform.ComposeView
        android:id="@+id/composeView"
        android:layout_width="match_parent"
        android:layout_height="200dp"/>

</LinearLayout>

```

```

class MainActivity : ComponentActivity(R.layout.activity_main) {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        val composeView = findViewById<ComposeView>
(R.id.composeView)

        composeView.setContent { Greeting() }
    }
}

```

В обоих случаях, если запустить в таком виде, как сейчас, всё заработает: наша `ViewModel` внутри функции `Greeting` без проблем создастся и положится в `ViewModelStore`, который принадлежит Activity. Почему так происходит?

В обоих случаях мы вызываем метод `setContent()`, в первом кейсе это `ComponentActivity.setContent()`, а во втором `ComposeView.setContent()`, которые открывают Composable-область.

Рассмотрим сначала первый кейс, начнём с `setContent` для активности (`ComponentActivity`).

Использование `ComponentActivity.setContent`:

```
public fun ComponentActivity.setContent(
    parent: CompositionContext? = null,
    content: @Composable () -> Unit
) {
    val existingComposeView =
        window.decorView.findViewById<ViewGroup>
            (android.R.id.content).getChildAt(0) as? ComposeView

    if (existingComposeView != null)
        with(existingComposeView) {
            setParentCompositionContext(parent)
            setContent(content)
        }
    else
        ComposeView(this).apply {
            // Set content and parent **before** setContentView
            // to have ComposeView create the composition on
            attach
                setParentCompositionContext(parent)
                setContent(content)
            // Set the view tree owners before setting the content
            view so that the inflation
            // process and attach listeners will see them already
            present
                setOwners()
                setContentView(this,
                    DefaultActivityContentLayoutParams)
        }
}
```



Обратите внимание, что это функция расширения `setContent` является расширением для `ComponentActivity` и имеет дополнительную логику по инициализации `Owner`-ов и прочих компонентов. Внутри себя она использует `ComposeView` и его метод `setContent`.

Что здесь происходит? У `window` есть `DecorView`, внутри этого `DecorView` лежит ещё один `ViewGroup(FrameLayout)`. У этого `ViewGroup` извлекается `ComposeView` под индексом 0, если он есть. Если его нет, то создается новый и вызывается метод `setContentView` (который есть у всех активити и унаследован от самого `Activity`). Но то, что нам нужно, происходит до вызова метода `setContentView` — речь идёт о `setOwners`. Давайте глянем на его исходники тоже:

```
private fun ComponentActivity.setOwners() {
    val decorView = window.decorView
    ...
    if (decorView.findViewTreeViewModelStoreOwner() == null) {
        decorView.setViewTreeViewModelStoreOwner(this)
    }
    ...
}
```

И именно здесь `ViewModelStoreOwner` кладётся в `DecorView` посредством вызова метода `setViewTreeViewModelStoreOwner`, куда передается `this` — то есть само активити. `DecorView` является самым(почти) корневым `View` во всей иерархии `View`, выше его стоит только сам `Window`.

Общая картина взаимодействия `ViewModelStoreOwner`, `ComposeView` и `LocalView`

Теперь давайте обобщим весь процесс и сделаем итоги: когда мы используем `ComponentActivity` (или его наследников `FragmentActivity` и `AppCompatActivity`) в `Compose` и создаём `ViewModel`, используя делегаты `compose/hilt/koin`, то внутри идёт обращение к `LocalViewModelStoreOwner`. Тот отдаёт `ViewModelStoreOwner`, если он есть. Если нет, то обращается к `Composable`-методу `findViewTreeViewModelStoreOwner`. Тот, в свою очередь, внутри себя обращается к `composition local` — `LocalView.current`, получает `View` и у этого `View` вызывает другой

extension-метод `View.findViewTreeViewModelStoreOwner`. Этот метод рекурсивно, начиная с `LocalView`, ищет сохранённый `ViewModelStoreOwner` в тегах `View` и так добирается вверх по иерархии `View`, пока не найдёт. Если найдёт, то вернёт его; если не найдёт, то вернёт `null`, и выбросится ошибка: ***No ViewModelStoreOwner was provided via LocalViewModelStoreOwner***

Как мы видели выше, при вызове `ComponentActivity.setContent{}` под капотом внутри вызывается метод `ComponentActivity.setOwners()`, в котором помещается `ViewModelStoreOwner` в тег `DecorView`. Получается, что при вызове метода `View.findViewTreeViewModelStoreOwner()`, пробираясь по иерархии `View`, в конечном итоге найдётся `ViewModelStoreOwner` внутри самой верхней `View` (`DecorView`), но в `Compose` нет прямого доступа к `DecorView`, вместо этого идёт обращение к `LocalView.current`:

`LocalViewModelStoreOwner.android.kt`

```
@Composable
internal actual fun findViewTreeViewModelStoreOwner():
    ViewModelStoreOwner? =
        LocalView.current.findViewTreeViewModelStoreOwner()
```

В этой цепочке мы не рассмотрели только один момент — откуда берётся `LocalView`. Точнее, понятно, что это `CompositionLocal`, но откуда в нём ссылка на текущее `View`? или кем является текущее `View`?

Если кратко и абстрактно: `ComposeView` внутри себя сам вызывает `LocalView` и провайдит ему самого себя. Поэтому `LocalView` по умолчанию ссылается на тот `ComposeView`, в котором было запущено дерево `Composable`-функций. А дерево `Compose` в Android всегда начинается именно с `ComposeView`.

Ниже — полный путь до момента, где `LocalView` получает значение. Без подробных комментариев, просто цепочка:

```
class ComposeView @JvmOverloads constructor(...) :
    AbstractComposeView(context, attrs, defStyleAttr)
```

`ComposeView` наследуется от `AbstractComposeView`. Смотрим, что происходит внутри `AbstractComposeView`:

```

abstract class AbstractComposeView(...) : ViewGroup(...) {
    private fun ensureCompositionCreated() {
        if (composition == null) {
            composition =
setContent(resolveParentCompositionContext()) {
            Content()
        }
    }
}

```

В методе `ensureCompositionCreated`, который вызывается, например, при `onMeasure` или `onAttachedToWindow`, или когда вызываем `ComposeView.setContent`, нас интересует вызов функции `setContent`:

```

internal fun AbstractComposeView.setContent(...): Composition {
    val composeView = ... ?: AndroidComposeView(...).also {
        addView(it.view, DefaultLayoutParams)
    }
    return doSetContent(composeView, parent, content)
}

```

Тут происходит следующее: создаётся объект класса `AndroidComposeView`, этот же объект помещается внутрь `ComposeView` вызовом `addView`. Напоминаю, что `AbstractComposeView` это абстрактный класс, и один из его наследников — это `ComposeView`. Хотя здесь работа идёт на уровне абстракций, фактически когда вызывается `addView`, то он вызывается для `ComposeView`.

Если стало слишком много новых названий, которые вызывают путаницу, то вот краткое объяснение:

- `AbstractComposeView` - абстрактный класс, который является `ViewGroup` и имеет уже много реализаций внутри
- `ComposeView` - один из наследников `AbstractComposeView`, который позволяет нам запускать `Composable` функции внутри себя. В Android всё упирается в работу с ним в конечном итоге, так как в Android нет способа запускать

Composable напрямую на уровне Window. Между Window и нашими Composable экранами стоят куча View и ViewGroup, в том числе и сам ComposeView

- AndroidComposeView - низкоуровневый класс, внутри которого в конечном итоге и рисуются наши Composable экраны

Далее — doSetContent:

```
private fun doSetContent(
    owner: AndroidComposeView,
    parent: CompositionContext,
    content: @Composable () -> Unit
): Composition {
    ...
    val wrapped = owner.view.getTag(R.id.wrapped_composition_tag)
        as? WrappedComposition
    ?: WrappedComposition(owner, original).also {
        owner.view.setTag(R.id.wrapped_composition_tag, it)
    }
    wrapped.setContent(content)
}
```

Переходим в WrappedComposition.setContent:

```
private class WrappedComposition(
    val owner: AndroidComposeView,
    val original: Composition
) : Composition, LifecycleEventObserver, CompositionServices {
    override fun setContent(content: @Composable () -> Unit) {
        ...
        ProvideAndroidCompositionLocals(owner, content)
        ...
    }
}
```

И вот — ключевой момент:

```

@Composable
internal fun ProvideAndroidCompositionLocals(
    owner: AndroidComposeView,
    content: @Composable () -> Unit
) {
    CompositionLocalProvider(
        ...
        LocalView provides owner.view,
        ...
    ) {
        content()
    }
}

```

Здесь `LocalView` получает значение `owner.view`, где `owner` — это `AndroidComposeView`, созданный внутри `ComposeView`.

Вывод: `LocalView` получает ссылку на `View`, внутри которого выполняется композиция, за счёт того, что `ComposeView` сам инициализирует `AndroidComposeView`, который далее передаётся в `ProvideAndroidCompositionLocals`. `AndroidComposeView` создаётся и хранится **внутри** `ComposeView`, и `LocalView` ссылается именно на этот `AndroidComposeView`, а не на сам `ComposeView`.

`ComposeView` наследуется от `AbstractComposeView`, который в свою очередь — `ViewGroup`. То есть `ComposeView` — это не сам `AndroidComposeView`, а просто контейнер, который при вызове `setContent` создаёт `AndroidComposeView` и вставляет его внутрь.

Поэтому, когда в `ProvideAndroidCompositionLocals` происходит вот это:

```
LocalView provides owner.view
```

`owner.view` — это `AndroidComposeView`, а не `ComposeView`.

Иерархия `View`, если `Activity` — это `AppCompatActivity`, будет выглядеть так:

```

ViewRootImpl
└─ DecorView -> имеет слабую ссылку на ViewModelStoreOwner (то

```

есть активити)

- └─ LinearLayout
 - └─ FrameLayout
 - └─ FitWindowsLinearLayout (action_bar_root)
 - └─ ContentFrameLayout (android:id/content)
 - └─ ComposeView
 - └─ AndroidComposeView -> имеет слабую ссылку на ViewModelStoreOwner (то есть активити)


А если это `ComponentActivity` или `FragmentActivity`, то чуть короче:

`ViewRootImpl`

- └─ `DecorView` -> имеет слабую ссылку на `ViewModelStoreOwner` (то есть активити)
 - └─ `LinearLayout`
 - └─ `FrameLayout` (android:id/content)
 - └─ `ComposeView`
 - └─ `AndroidComposeView` -> имеет слабую ссылку на `ViewModelStoreOwner` (то есть активити)

Интересный факт

`ViewRootImpl` — это корневой элемент всей иерархии `View`. На практике каждый Android-разработчик хотя бы раз сталкивался с ошибкой:

 "Only the original thread that created a view hierarchy can touch its views."

Эта ошибка возникает, если попытаться обратиться к `View` из не-UI потока. А выбрасывает её как раз `ViewRootImpl` внутри метода `checkThread()`:

```
public final class ViewRootImpl implements ViewParent, ...
{

    void checkThread() {
        Thread current = Thread.currentThread();
        if (mThread != current) {
```

```

        throw new CalledFromWrongThreadException(
            "Only the original thread that created a
            view hierarchy can touch its views."
            + " Expected: " + mThread.getName()
            + " Calling: " + current.getName());
    }
}

```

Ключевая мысль — `LocalView` по умолчанию указывает на `AndroidComposeView`, который создаётся внутри `ComposeView` динамически. Сам `ComposeView` — просто оболочка, которая знает, как всё связать и встроить дерево `Composable` в нужное место иерархии.

Тут мы рассмотрели первый кейс, когда мы используем `ComponentActivity.setContent{}` с передачей нашей композиции и создания `ViewModel`. Второй флоу использования — это внутри иерархии `View`, например, если у нас все экраны на `Fragment/View`, и мы в каких-то местах используем `Compose`. Это возможно благодаря `ComposeView`. Рассмотрим такой кейс:

Использование `ComposeView.setContent`:

Вот пример кода из примеров выше:

```

class MainActivity : ComponentActivity(R.layout.activity_main) {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        val composeView = findViewById<ComposeView>(
            R.id.composeView)

        composeView.setContent { Greeting() }
    }
}

```

```

@Composable
fun Greeting(modifier: Modifier = Modifier) {

```

```

    val viewModel =
androidx.lifecycle.viewmodel.compose.viewModel<MyViewModel>()
    Text(
        text = "Hello ${viewModel.getName()}",
        modifier = modifier
    )
}

```

Как работает `setContent` у `ComposeView` мы уже рассмотрели. Внутри себя `ComposeView.setContent` не кладёт ссылку на `ViewModelStoreOwner` и не имеет внутри себя вызов функции `setViewTreeViewModelStoreOwner`, он только помогает провайдить `LocalView`.

Но если запустить код в текущем виде, всё заработает как ожидалось. В чём дело? Ситуация аналогичная, как и ранее, когда уже за нас предусмотрели такую логику. Дело в следующем: при вызове метода `setContentView(R.layout.activity_main)` или даже при передаче ссылки на `layout` в конструктор:

`ComponentActivity(R.layout.activity_main)` происходит следующая цепочка:

Если передаем `Layout Id` в конструктор:

```

open class ComponentActivity() ... {

    @ContentView
    constructor(@LayoutRes contentLayoutId: Int) : this() {
        this.contentLayoutId = contentLayoutId
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        ...
        if (contentLayoutId != 0) {
            setContentView(contentLayoutId)
        }
    }
}

```

В методе `onCreate` вызывается `setContentView`, если передали `contentLayoutId` в конструктор. Если же напрямую вызвали `setContentView`, то логика следующая:

Когда мы вызываем метод `setContentView()` и передаем нашу View или id макета, то под капотом происходит следующее (далее исходники метода `setContentView`):

```
open class ComponentActivity() ... {

    override fun setContentView(@LayoutRes layoutResID: Int) {
        initializeViewTreeOwners()
        reportFullyDrawnExecutor.viewCreated(window.decorView)
        super.setContentView(layoutResID)
    }
}
```

Название метода `initializeViewTreeOwners` выглядит заманчивым, поэтому глянем в исходники:

```
@CallSuper
open class ComponentActivity() ... {

    open fun initializeViewTreeOwners() {
        ...
        window.decorView.setViewTreeViewModelStoreOwner(this)
        ...
    }
}
```

И мы здесь видим, что у `window` вызывается метод `getDecorView` (в Kotlin все геттеры из Java имеют синтаксис как у переменной), и дальше вызывается функция `setViewTreeViewModelStoreOwner`, который помещает `this` (`ViewModelStoreOwner`) в тег внутри `DecorView`.

Сделаем итоги: когда мы начинаем свой UI с метода `setContentView` или передаем layout id в конструктор активности, то внутри самого `ComponentActivity` (он же родитель для `FragmentActivity` и `AppCompatActivity`) срабатывает логика, которая помещает себя (активности реализует интерфейс `ViewModelStoreOwner`) во внутренний тег `DecorView` (он же почти самый высокий по иерархии) посредством вызова метода `setViewTreeViewModelStoreOwner`. Далее, когда мы добавляем в иерархию View свой `ComposeView`, чтобы начать писать на Compose, то внутри `ComposeView` провайдится значение для `LocalView.current`. Затем при создании

ViewModel внутри Compose идет обращение к LocalViewModelStoreOwner, а именно к его полю current. Там проверяется, есть ли значение, и если нет, вызывается метод `findViewTreeViewModelStoreOwner` у LocalView, который ищет ViewModelStoreOwner, поднимаясь вверх по иерархии, пока не найдет. Таким образом, в конечном итоге находится ViewModelStoreOwner у DecorView. Вот так всё и работает. Далее диаграмма иерархии View:

```
ViewRootImpl
└─ DecorView -> имеет слабую ссылку на ViewModelStoreOwner (то
    есть активити)
    └─ LinearLayout
        └─ FrameLayout (android:id/content)
            └─ FrameLayout (app:id/frameRootLayout)
                └─ ComposeView (app:id/composeView)
                    └─ AndroidComposeView
```

На этом статья почти закончена, осталось пролить свет на один момент. К этому моменту вся информация выше наводит на мысль: а почему мы в начале статьи вручную сами вызывали метод `setViewTreeViewModelStoreOwner`, если всё это делается за нас?

(P.S. я возвращаюсь к примеру в начале статьи с View (TranslatableTextView))

Благодаря тому, что мы установили ViewModelStoreOwner для нашего корневого layout внутри нашего макета, тег внутри FrameLayout (frameRootLayout) имеет ссылку (weak) на ViewModelStoreOwner:

```
class MainActivity : AppCompatActivity() {

    private val frameRootLayout by lazy {
        findViewById<FrameLayout>(R.id.frameRootLayout) }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        ...
        // Привязываем ViewModelStoreOwner к дереву View
        frameRootLayout.setViewTreeViewModelStoreOwner(this)
```

```

    ...
}
}

```

И метод `findViewTreeViewModelStoreOwner`, когда пробегается по иерархии `View`, сначала поищет в `TranslatableTextView`, а затем, если он не найдет, будет подниматься вверх по родителям. Родитель — это `frameRootLayout` (`FrameLayout`), там он и найдет `ViewModelStoreOwner`. Но что, если мы удалим установку `frameRootLayout.setViewTreeViewModelStoreOwner(this)` и запустим код?

```

class TranslatableTextView(context: Context) :
    AppCompatActivity(context) {

    private val viewModel: TranslatableTextViewViewModel by lazy {
        val owner = findViewTreeViewModelStoreOwner() ?:
        error("ViewModelStoreOwner not found for TranslatableTextView")
        ViewModelProvider.create(owner =
        owner).get(TranslatableTextViewViewModel::class.java)
    }
    ...
}

```

То всё так же будет работать. Почему? Дело в том, что, как мы уже ранее рассмотрели в иерархии, есть ещё один родитель — `DecorView`. Как это выглядит:

```

ViewRootImpl
└─ DecorView -> имеет слабую ссылку на ViewModelStoreOwner (то
    есть активити)
    └─ LinearLayout
        └─ FrameLayout (android:id/content)
            └─ FrameLayout (app:id/frameRootLayout)
                └─ TranslatableTextView

```

И когда мы вызываем метод `AppCompatActivity.setContentView()` и передаем нашу `View` или `id` макета, то под капотом происходит следующее (далее исходники метода `setContentView`):


```
open class ComponentActivity() ... {

    override fun setContentView(@LayoutRes layoutResID: Int) {
        initializeViewTreeOwners()
        ...
    }
}
```

Название метода `initializeViewTreeOwners` выглядит заманчивым, поэтому глянем в исходники:

```
@CallSuper
open class ComponentActivity() ... {

    open fun initializeViewTreeOwners() {
        ...
        window.decorView.setViewTreeViewModelStoreOwner(this)
        ...
    }
}
```

Итог такой: вызывайте `setViewTreeViewModelStoreOwner` только если сами хотите указать, в какую `View` вы хотите поместить определенный `ViewModelStoreOwner`. В Compose вызывайте `LocalViewModelStoreOwner provides yourViewModelStoreOwner` только если у вас появилась в этом необходимость, но на практике не встречал, чтобы кто-то занимался этим, так как решения из коробки от Google всё решают, и в ручной работе обычно нет необходимости — unless вы реально что-то очень кастомное мутите.

ViewModel Compose DI Delegates:

Когда мы рассмотрели `ViewModel` для `Composable` функций, мы рассмотрели только `composable` функцию `viewModel()` — функцию из библиотеки: `androidx.lifecycle:lifecycle-viewmodel-compose:2.8.7` без DI. И инициализация была такая:

```

@Composable
fun Greeting(modifier: Modifier = Modifier) {
    // тут специально не импортировал функцию
    val viewModel =
androidx.lifecycle.viewmodel.compose.viewModel<MyViewModel>()
}

```

Ранее я говорил что:

- ⚠ Когда мы внутри нашего Composable-функций вызываем любую из extension-функций по созданию viewModel: то ли
1. viewModel из библиотеки **androidx.lifecycle:lifecycle-viewmodel-compose**,
 2. koinViewModel() из библиотеки **io.insert-koin:koin-androidx-compose**,
 3. hiltViewModel() из **androidx.hilt:hilt-navigation-compose**,

То в конечном итоге мы обращаемся именно к **CompositionLocal** с названием **LocalViewModelStoreOwner** к его полю **current**. Поэтому реализация везде одна и та же независимо от библиотеки, весь флоу который мы рассмотрели независимо от делегата и библиотеки будет работать так же.

Давайте убедимся в этом, просто рассмотрим сигнатуру всех троих:

1. Первый мы уже видели, посмотрим еще раз:

androidx.lifecycle.viewmodel.compose.ViewModel.kt

```

@Suppress("MissingJvmstatic")
@Composable
public inline fun <reified VM : ViewModel> viewModel(
    viewModelStoreOwner: ViewModelStoreOwner =
checkNotNull(LocalViewModelStoreOwner.current) {
    "No ViewModelStoreOwner was provided via
LocalViewModelStoreOwner"
},

```

```

    ...
): VM = viewModel(VM::class, viewModelStoreOwner, key, factory,
extras)

```

2. Koin: `org.koin.androidx.compose.ViewModel.kt`:

```

@OptIn(KoinInternalApi::class)
@Composable
inline fun <reified T : ViewModel> koinViewModel(
    qualifier: Qualifier? = null,
    viewModelStoreOwner: ViewModelStoreOwner =
checkNotNull(LocalViewModelStoreOwner.current) {
    "No ViewModelStoreOwner was provided via
LocalViewModelStoreOwner"
},
    ...
): T {
    return resolveViewModel(
        T::class, viewModelStoreOwner.viewModelStore, key, extras,
qualifier, scope, parameters
    )
}

```

3. Hilt: `androidx.hilt.navigation.compose.HiltViewModel.kt`:

```

@Composable
inline fun <reified VM : ViewModel> hiltViewModel(
    viewModelStoreOwner: ViewModelStoreOwner =
checkNotNull(LocalViewModelStoreOwner.current) {
    "No ViewModelStoreOwner was provided via
LocalViewModelStoreOwner"
},
    key: String? = null
): VM {
    val factory = createHiltViewModelFactory(viewModelStoreOwner)

```

```
    return viewModel(viewModelStoreOwner, key, factory = factory)
}
```

Как можно заметить, все три делегата — `viewModel()`, `koinViewModel()` и `hiltViewModel()` — используют один и тот же механизм получения `ViewModelStoreOwner` через `LocalViewModelStoreOwner.current`. Отличия лишь в синтаксисе и дополнительной логике, связанной с DI, но в основе всё сводится к одному — получению `ViewModelStoreOwner` из дерева `View`.

Причина проста: в Compose нет прямого доступа к `ComponentActivity` и её производным (`FragmentActivity`, `AppCompatActivity`), как и к `Fragment` или `NavBackStackEntry`. Поэтому используется `LocalViewModelStoreOwner`, который при отсутствии значения в `current` обращается к `LocalView.current` и уже для него вызывает метод `findViewTreeViewModelStoreOwner()` — стандартный способ получить ближайший `ViewModelStoreOwner` из иерархии `View`.

Именно поэтому `LocalViewModelStoreOwner` — ключевой элемент. Он — универсальный посредник между Compose и традиционным `ViewModel`-механизмом Android. И независимо от того, используете ли вы Hilt, Koin или ничего из DI, — всё работает через него.

SavedStateHandle и Bundle под капотом: как Android сохраняет состояние

Это продолжение трех предыдущих статей.

1. В первой мы разобрали, где в конечном итоге хранится `ViewModelStore` в случае с `Activity`,
2. Во второй — как это устроено во `Fragment`,
3. В третьей где хранятся `ViewModel`-и, когда мы используем **Compose** (или даже просто `View`).

В этой статье рассмотрим Где хранится `SavedStateHandle`, проверим `SavedStateHandle` vs `onSaveInstanceState` vs `ViewModel(ViewModelStore)` Поймем связку `SavedStateHandle` с `ViewModel`. И узнаем ответ на главный вопрос, где храниться `Bundle`. Но, как всегда, начнём с базиса.

Базис

В статье не будет описания того, как работать с этими API, а будет рассказано о том, как они устроены изнутри, поэтому я буду исходить из того, что вы уже работали с ними. Как всегда, начнём с базиса — дадим определения для `SavedStateHandle`, `onSaveInstanceState` и `ViewModel`:

ViewModel — компонент архитектурного паттерна MVVM, предоставленный Google как примитив, позволяющий пережить изменение конфигурации. Изменение конфигурации — это состояние, из-за которого `Activity/Fragment` пересоздаётся; именно это состояние может пережить `ViewModel`. Увы, на этом обязанности `ViewModel` по хранению данных в контексте Android заканчиваются.

Если же процесс приложения умирает или прерывается, `ViewModel` не справится; тогда на сцену выходят старые добрые методы `onSaveInstanceState/onRestoreInstanceState`.

onSaveInstanceState/onRestoreInstanceState — методы жизненного цикла Activity, Fragment и даже View (да, View тоже может сохранять состояние), которые позволяют сохранять и восстанавливать временное состояние пользовательского интерфейса при изменении конфигурации (например, при повороте экрана) или при полном уничтожении активности из-за нехватки ресурсов. В onSaveInstanceState данные сохраняются в Bundle, который автоматически передаётся в onRestoreInstanceState при восстановлении активности.

Это базовый механизм для хранения примитивных типов (и их массивов), Parcelable/Serializable и ещё пары нативных Android-типов. Эти методы требуют явного указания того, что именно нужно сохранить, а логика прописывается внутри Activity и Fragment. Большинство архитектурных паттернов (MVI, MVVM) гласят, что View (Fragment/Activity/Compose) должны быть максимально простыми и не содержать никакой логики, кроме отображения данных, поэтому прямое использование этих методов сейчас уступает место Saved State API, которое хорошо интегрируется с ViewModel, наделяя её не только возможностью «спасать» данные от изменений конфигурации, но и сохранять сериализуемые данные при уничтожении или остановке процесса по инициативе системы.

Saved State API — современная альтернатива

onSaveInstanceState/onRestoreInstanceState, более гибко управляющая состоянием, особенно в связке с ViewModel. **SavedStateHandle** — объект, передаваемый в конструктор ViewModel, который позволяет безопасно сохранять и восстанавливать данные даже после уничтожения процесса. В отличие от статичного onSaveInstanceState, SavedStateHandle также позволяет подписываться на Flow и LiveData тех данных, которые он хранит и восстанавливает. Он автоматически интегрирован с ViewModel и поддерживает сохранение состояния при изменениях конфигурации, а также при полном уничтожении процесса приложения. Дополнительное преимущество — возможность подписываться на изменения значений в SavedStateHandle и получать реактивное поведение прямо в ViewModel.



Под «уничтожением или прерыванием процесса», о котором идёт речь в статье, подразумевается ситуация, когда приложение находится в фоне и сохраняется в стеке задач. Обычно это происходит, когда пользователь сворачивает приложение, не закрывая его. Через некоторое время бездействия система может остановить процесс. Не стоит путать это с

кейсом, когда пользователь сам вручную закрывает приложение — это другой сценарий.

onSaveInstanceState / onRestoreInstanceState

Давайте также освежим память о методах `onSaveInstanceState` и `onRestoreInstanceState`:

```
class RestoreActivity : AppCompatActivity() {

    private var counter = 0

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        // Восстановление значения при пересоздании
        counter = savedInstanceState?.getInt("counter_key") ?: 0
    }

    override fun onRestoreInstanceState(savedInstanceState: Bundle) {
        super.onRestoreInstanceState(savedInstanceState)
        // Восстановление значения при пересоздании
        counter = savedInstanceState.getInt("counter_key")
    }

    override fun onSaveInstanceState(outState: Bundle) {
        super.onSaveInstanceState(outState)
        // Сохраняем значение
        outState.putInt("counter_key", counter)
        Log.d("RestoreActivity", "onSaveInstanceState: Counter saved = $counter")
    }
}
```

onSaveInstanceState — вызывается для получения состояния Activity перед её уничтожением, чтобы оно могло быть восстановлено в методах `onCreate` или

`onRestoreInstanceState`. `Bundle`, заполненный в этом методе, будет передан в оба метода.

Этот метод вызывается до того, как `Activity` может быть уничтожена, чтобы при повторном создании она могла восстановить своё состояние. Не следует путать его с методами жизненного цикла, такими как `onPause` (вызывается всегда, вызывается при частичной потере фокуса `Activity`) или `onStop` (когда `Activity` становится невидимой).

- **Пример**, когда `onPause` и `onStop` вызываются, но `onSaveInstanceState` — нет: при возвращении из `Activity B` в `Activity A`. В этом случае состояние `B` не требуется восстанавливать, поэтому `onSaveInstanceState` для `B` не вызывается.
- **Другой пример**: если `Activity B` запускается поверх `Activity A`, но `A` остаётся в памяти, то `onSaveInstanceState` для `A` также не вызывается, так как `Activity` остаётся в памяти и не требуется сохранять её состояние.

Реализация по умолчанию этого метода автоматически сохраняет большую часть состояния пользовательского интерфейса, вызывая `onSaveInstanceState()` у каждого `View` в иерархии, у которого есть ID, а также сохраняет ID элемента, находившегося в фокусе. Восстановление этих данных происходит в стандартной реализации `onRestoreInstanceState()`. Если вы переопределяете метод для сохранения дополнительной информации, рекомендуется вызвать реализацию по умолчанию через

```
super.onSaveInstanceState(outState)
```

— иначе придётся вручную сохранять состояние всех `View`.

Если метод вызывается, то это произойдёт **после** `onStop` для приложений, нацеленных на платформы, начиная с Android P. Для более ранних версий Android этот метод будет вызван **до** `onStop`, и нет никаких гарантий, будет ли он вызван до или после `onPause`.

Документация гласит:

If called, this method will occur after `onStop` for applications targeting platforms starting with `android.os.Build.VERSION_CODES.P`. For applications

targeting earlier platform versions this method will occur before onStop and there are no guarantees about whether it will occur before or after onPause.

onRestoreInstanceState — этот метод вызывается **после** **onStart**, когда активность повторно инициализируется из ранее сохранённого состояния, переданного в **savedInstanceState**. Большинство реализаций используют для восстановления состояния метод **onCreate**, но иногда бывает удобнее делать это здесь, после того как завершена вся инициализация, или чтобы подклассы могли решить, использовать ли вашу реализацию по умолчанию. Стандартная реализация этого метода восстанавливает состояние представлений (View), которое было ранее заморожено методом **onSaveInstanceState**. Этот метод вызывается **между onStart и onCreate**. Он срабатывает **только при повторном создании активности**; метод **не вызывается**, если **onStart** был вызван по любой другой причине (например, при переходе из фона на передний план).

На этом примере временно забываем о них, чуть позже мы их снова встретим в более низкоуровневых цепочках вызовов.

Saved State Api

- i** С версии 1.3.0-alpha02 androidx.savedstate:savedstate стала поддерживать Kotlin Multiplatform. Теперь SavedState работает не только на Android (Bundle), но и на iOS, JVM, Linux и macOS Map<String, Any>, сохраняя совместимость.

Что бы понять работу **Saved State Api** перепишем пример выше с **onSaveInstanceState** и **onRestoreInstanceState** используя Saved State Api, делает ровно тоже самое:

```
class RestoreActivity : AppCompatActivity() {  
  
    private var counter = 0  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
    }  
}
```

```

        // Восстановление значения при пересоздании
        counter =
savedStateRegistry.consumeRestoredStateForKey("counter_key").getInt("counter", 0) ?: 0

        savedStateRegistry.registerSavedStateProvider(
            key = "counter_key",
            provider = object :
SavedStateRegistry.SavedStateProvider {
                override fun saveState(): SavedState {
                    return SavedState(bundleOf("counter" to
counter))
                }
            }
        )
    }
}

```

Мы вызываем у объекта `savedStateRegistry` метод `registerSavedStateProvider` куда передаем `key` и анонимный объект `SavedStateRegistry.SavedStateProvider` который возвращает `Bundle` обернутый в объект `SavedState`, давайте сейчас же определим что из себя представляет этот тип `SavedState`, если зайти в исходники, а именно в `expect` логику, то тип описан следующим образом:

androidx.savedstate.SavedState.kt:

```

/**
 * An opaque (empty) common type that holds saveable values to be
 * saved and restored by native
 * platforms that have a concept of System-initiated Process
 * Death.
 *
 * That means, the OS will give the chance for the process to keep
 * the state of the application
 * (normally using a serialization mechanism), and allow the app
 * to restore its state later. That is
 * commonly referred to as "state restoration".
 * ...

```

```
*/  
public expect class SavedState
```

В контексте `android` нас интересует именно `actual` реализация, по этому далее специфичная для `android` `actual`

`androidx.savedstate.SavedState.android.kt`:

```
public actual typealias SavedState = android.os.Bundle
```

Как видим в `Android` нет на самом деле какого-то типа как `SavedState`, в `actual` реализаций это просто `typealias` который ссылается на тот же старый добрый родной класс `Bundle`, по этому всегда представляйте что там где используется `SavedState` - на самом деле используется класс `Bundle`, поэтому ничто не мешает нам отказаться от лишней обёртки и вернуть `Bundle` напрямую:

```
savedStateRegistry.registerSavedStateProvider(  
    key = "counter_key",  
    provider = object : SavedStateRegistry.SavedStateProvider {  
        override fun saveState(): Bundle {  
            return bundleOf("counter" to counter)  
        }  
    }  
)
```

Раз с этим разобрались, дальше давайте зайдём в исходники метода `registerSavedStateProvider` и `consumeRestoredStateForKey`, эти методы вызывается у переменной `savedStateRegistry` которая имеет тип `SavedStateRegistry`, давайте быстро узнаем определение этого класса:

`SavedStateRegistry` - управляет сохранением и восстановлением сохранённого состояния, чтобы данные не терялись при пересоздании компонентов. Реализация привязана к `SavedStateRegistryImpl`, которая отвечает за фактическое хранение и восстановление данных. Интерфейс для подключения компонентов, которые потребляют и вносят данные в сохранённое состояние. Объект имеет такой же жизненный цикл, как и его владелец (`Activity` или `Fragment`): когда `Activity` или `Fragment` пересоздаются (например, после уничтожения процесса или изменении конфигурации), создаётся новый экземпляр этого объекта.

Но откуда берется `savedStateRegistry` переменная внутри `Activity` мы рассмотрим позже, пока достаточно знать что он есть у `Activity`, далее исходники метода `registerSavedStateProvider` и `consumeRestoredStateForKey` принадлежащий классу `SavedStateRegistry` (expect): `androidx.savedstate.SavedStateRegistry.kt`

```
public expect class SavedStateRegistry internal constructor(
    impl: SavedStateRegistryImpl,
) {

    /** This interface marks a component that contributes to saved
    state. */
    public fun interface SavedStateProvider {

        public fun saveState(): SavedState
    }

    ...
    public val isRestored: Boolean
    ...
    @MainThread
    public fun consumeRestoredStateForKey(key: String):
    SavedState?
    ...
    @MainThread
    public fun registerSavedStateProvider(key: String, provider:
    SavedStateProvider)
    ...
    public fun getSavedStateProvider(key: String):
    SavedStateProvider?
    ...
    @MainThread
    public fun unregisterSavedStateProvider(key: String)
}
```

Как мы видим на самом деле тут много методов у `SavedStateRegistry`, для нашей статьи достаточно понимать работу методов `registerSavedStateProvider` и

`consumeRestoredStateForKey`, но что бы хоть какое-то понимание было, давайте быстро пройдемся по каждому:

1. **`consumeRestoredStateForKey`** — извлекает и удаляет из памяти `SavedState` (Bundle), который был зарегистрирован с помощью `registerSavedStateProvider`. При повторном вызове возвращает `null`.
2. **`registerSavedStateProvider`** — регистрирует `SavedStateProvider` с указанным ключом. Этот провайдер будет использоваться для сохранения состояния при вызове `onSaveInstanceState`.
3. **`getSavedStateProvider`** — возвращает зарегистрированный `SavedStateProvider` по ключу или `null`, если он не найден.
4. **`unregisterSavedStateProvider`** — удаляет из реестра ранее зарегистрированный `SavedStateProvider` по переданному ключу.
5. **`SavedStateProvider`** — интерфейс, предоставляющий объект `SavedState` (Bundle) при сохранении состояния.
6. **`isRestored`** — возвращает `true`, если состояние было восстановлено после создания компонента.

В `expect`-версиях отсутствуют реализации — там только сигнатуры методов. Также мы рассмотрели исходники интерфейса `SavedStateProvider`, который представляет собой callback для получения Bundle, подлежащего сохранению. Чтобы увидеть реализацию метода `registerSavedStateProvider`, необходимо найти **actual-реализацию**, а затем перейти к `actual`-реализации `SavedStateRegistry`.

`androidx.savedstate.SavedStateRegistry.android.kt`:

```
public actual class SavedStateRegistry internal actual
    constructor(
        private val impl: SavedStateRegistryImpl,
    ) {

        @get:MainThread
        public actual val isRestored: Boolean
            get() = impl.isRestored
```

```

    @MainThread
    public actual fun consumeRestoredStateForKey(key: String):
SavedState? =
        impl.consumeRestoredStateForKey(key)

    @MainThread
    public actual fun registerSavedStateProvider(key: String,
provider: SavedStateProvider) {
        impl.registerSavedStateProvider(key, provider)
    }

    public actual fun getSavedStateProvider(key: String):
SavedStateProvider? =
        impl.getSavedStateProvider(key)

    @MainThread
    public actual fun unregisterSavedStateProvider(key: String) {
        impl.unregisterSavedStateProvider(key)
    }

    public actual fun interface SavedStateProvider {
        public actual fun saveState(): SavedState
    }
    ...
}

```

actual реализация SavedStateRegistry делегирует все вызовы своих методов готовой имплементацией SavedStateRegistryImpl, по этому далее рассмотрим именно SavedStateRegistryImpl:

```

internal class SavedStateRegistryImpl(
    private val owner: SavedStateRegistryOwner,
    internal val onAttach: () -> Unit = {},
) {

    private val keyToProviders = mutableMapOf<String,

```

```

SavedStateProvider>()
    private var restoredState: SavedState? = null

    @MainThread
    fun consumeRestoredStateForKey(key: String): SavedState? {
        ...
        val state = restoredState ?: return null

        val consumed = state.read { if (contains(key))
getSavedState(key) else null }
        state.write { remove(key) }
        if (state.read { isEmpty() }) {
            restoredState = null
        }

        return consumed
    }

    @MainThread
    fun registerSavedStateProvider(key: String, provider:
SavedStateProvider) {
        ..
        keyToProviders[key] = provider
        ...
    }
    ...
}

```

Основные методы для сохранения, давайте просто поймем что здесь происходит:

1. `consumeRestoredStateForKey` - достает значение из `restoredState` (Bundle) по ключу, после того как достает значение, удаляет из `restoredState` (Bundle) значение и ключ, `restoredState` является самым коренным Bundle который внутри себя хранит все другие bundle
2. `registerSavedStateProvider` - просто добавляет объект `SavedStateProvider` внутрь карты `keyToProviders`

Эти методы — очень верхне уровневые и не раскрывают, как именно в итоге сохраняются данные, поэтому нужно копнуть глубже — внутри этого же класса `SavedStateRegistryImpl`:

```
internal class SavedStateRegistryImpl(
    private val owner: SavedStateRegistryOwner,
    internal val onAttach: () -> Unit = {},
) {
    private val keyToProviders = mutableMapOf<String,
SavedStateProvider>()
    private var restoredState: SavedState? = null

    @MainThread
    internal fun performRestore(savedState: SavedState?) {
        ...
        restoredState =
            savedState?.read {
                if (contains(SAVED_COMPONENTS_KEY))
getSavedState(SAVED_COMPONENTS_KEY) else null
            }
        isRestored = true
    }

    @MainThread
    internal fun performSave(outBundle: SavedState) {
        val inState = savedState {
            restoredState?.let { putAll(it) }
            synchronized(lock) {
                for ((key, provider) in keyToProviders) {
                    putSavedState(key, provider.saveState())
                }
            }
        }

        if (inState.read { !isEmpty() }) {
            outBundle.write { putSavedState(SAVED_COMPONENTS_KEY,
inState) }
        }
    }
}
```



```

    }
}

private companion object {
    private const val SAVED_COMPONENTS_KEY =
        "androidx.lifecycle.BundlableSavedStateRegistry.key"
}
}

```

1. `performSave` — вызывается, когда `Activity` или `Fragment` переходит в состояние `pause` → `stop`, то есть в момент вызова `onSaveInstanceState`. Этот метод отвечает за сохранение состояния всех `SavedStateProvider`, зарегистрированных через `registerSavedStateProvider`. Внутри метода создается объект `inState` типа `SavedState` (по сути, это сам `Bundle`). Если в `restoredState` уже есть данные, они добавляются в `inState`. Затем, в синхронизированном блоке, происходит обход всех зарегистрированных `SavedStateProvider`, вызывается метод `saveState()`, и результаты сохраняются в `inState`. В конце, если `inState` не пустой, его содержимое записывается в параметр `outBundle` под ключом `SAVED_COMPONENTS_KEY`.
2. `performRestore` — вызывается при создании или восстановлении `Activity` или `Fragment`. Этот метод просто читает из `savedState` значение по ключу `SAVED_COMPONENTS_KEY`, если оно существует. Найденное значение (вложенный `SavedState`) сохраняется в переменную `restoredState`, чтобы потом можно было передать его в соответствующие компоненты.

На данный момент мы увидели как работает логика сохранения и регистраций, теперь осталось понять кто же вызывает методы `performSave` и `performRestore` и в какой момент.

Этой логикой управляет `SavedStateRegistryController`, в связи с тем что `Saved State Api` тоже на `KMP`, по этому лучше сразу посмотрим actual версию:

```

public actual class SavedStateRegistryController private actual
constructor(
    private val impl: SavedStateRegistryImpl,
) {

```

```

    public actual val savedStateRegistry: SavedStateRegistry =
        SavedStateRegistry(impl)

    @MainThread
    public actual fun performAttach() {
        impl.performAttach()
    }

    @MainThread
    public actual fun performRestore(savedState: SavedState?) {
        impl.performRestore(savedState)
    }

    @MainThread
    public actual fun performSave(outBundle: SavedState) {
        impl.performSave(outBundle)
    }

    public actual companion object {

        @JvmStatic
        public actual fun create(owner: SavedStateRegistryOwner):
            SavedStateRegistryController {
            val impl =
                SavedStateRegistryImpl(
                    owner = owner,
                    onAttach = {
owner.lifecycle.addObserver(Recreator(owner)) },
                )
            return SavedStateRegistryController(impl)
        }
    }
}

```

И видим, что вызовами методов `SavedStateRegistryImpl.performSave` и `SavedStateRegistryImpl.performRestore` управляют одноимённые методы из

SavedStateRegistryController.

Также видим метод `create`, который создаёт `SavedStateRegistryImpl`, передаёт его в конструктор `SavedStateRegistryController` и возвращает сам `SavedStateRegistryController`.

Далее остаётся только понять, откуда вызываются сами методы `SavedStateRegistryController`. В начале статьи мы отложили разбор источника поля `savedStateRegistry` в `Activity`. Сейчас самое время разобраться.

Внутри `Activity` нам доступно поле `savedStateRegistry`. Это возможно потому, что `Activity` реализует интерфейс `SavedStateRegistryOwner`. Если посмотреть исходники, то можно увидеть, что `ComponentActivity` реализует `SavedStateRegistryOwner`. На самом деле `ComponentActivity` реализует множество интерфейсов, но ниже приведён фрагмент с опущенными остальными родителями:

```
open class ComponentActivity() : ..., SavedStateRegistryOwner, ...
{

    private val savedStateRegistryController:
    SavedStateRegistryController =
        SavedStateRegistryController.create(this)

    final override val savedStateRegistry: SavedStateRegistry
        get() = savedStateRegistryController.savedStateRegistry
}
```

`SavedStateRegistryOwner` - это просто interface который хранит в себе `SavedStateRegistry`, его реализует `Activity`, `Fragment` и `NavBackStackEntry`, выглядит он следующим образом:

```
public interface SavedStateRegistryOwner :
    androidx.lifecycle.LifecycleOwner {
    /** The [SavedStateRegistry] owned by this
    SavedStateRegistryOwner */
    public val savedStateRegistry: SavedStateRegistry
}
```

`SavedStateRegistry` доступен в любом компоненте, реализующем интерфейс `SavedStateRegistryOwner`. Этим интерфейсом обладают:

- `ComponentActivity` — это базовый класс для всех современных `Activity`.

```
open class ComponentActivity() : ..., SavedStateRegistryOwner,
... {

    private val savedStateRegistryController:
    SavedStateRegistryController =
        SavedStateRegistryController.create(this)

    final override val savedStateRegistry: SavedStateRegistry
        get() = savedStateRegistryController.savedStateRegistry
}
```

- `Fragment` — любой `Fragment` также реализует этот интерфейс.

```
public class Fragment implements ...SavedStateRegistryOwner,...{

    SavedStateRegistryController mSavedStateRegistryController;

    @NonNull
    @Override
    public final SavedStateRegistry getSavedStateRegistry() {
        return
        mSavedStateRegistryController.getSavedStateRegistry();
    }
}
```

- `NavBackStackEntry` - компонент навигаций из Jetpack Navigation

```
public expect class NavBackStackEntry : ...,
    SavedStateRegistryOwner {

    override val savedStateRegistry: SavedStateRegistry
```

```
}
```

Мы выяснили большую цепочку вызовов, давайте визуально посмотрим:

```
expect -> SavedStateRegistryController.performSave
-> actual SavedStateRegistryController.performSave
-> expect SavedStateRegistry
-> actual SavedStateRegistry
-> SavedStateRegistryImpl.performSave
-> SavedStateProvider.saveState()
-> // Bundle
```

Углубляться в работу `Fragment` и `NavBackStackEntry` не будем — разберёмся только с `Activity`. На данный момент мы понимаем, что в конечном итоге все вызовы идут в `SavedStateRegistryController`. Давай посмотрим, как `Activity` с ним взаимодействует:

Метод `performRestore` у `SavedStateRegistryController`, отвечающий за восстановление данных из `Bundle`, вызывается внутри `ComponentActivity.onCreate`, а метод `performSave`, сохраняющий данные в `Bundle`, — внутри `ComponentActivity.onSaveInstanceState`.

```
open class ComponentActivity() : ..., SavedStateRegistryOwner, ...
{

    override fun onCreate(savedInstanceState: Bundle?) {

        savedInstanceStateRegistryController.performRestore(savedInstanceState)
        super.onCreate(savedInstanceState)
        ...
    }

    @CallSuper
    override fun onSaveInstanceState(outState: Bundle) {
        ...
        super.onSaveInstanceState(outState)
        savedInstanceStateRegistryController.performSave(outState)
    }
}
```

```
}  
}
```

Здесь та самая точка, где `onSaveInstanceState/onRestoreInstanceState` объединяются с `SavedStateRegistryController/SavedStateRegistry`.

Теперь переключимся на `ViewModel` и его `SavedStateHandle`, чтобы понять, как он вписывается во всю эту логику. Для начала объявим обычную `ViewModel`, но в конструкторе передадим `SavedStateHandle`:

```
class MyViewModel(val savedStateHandle: SavedStateHandle) :  
    ViewModel()
```

i Как и говорилось в начале статьи, это не гайд по тому как пользоваться Saved State Api, тут больше ответ на вопрос как это работает под капотом

Далее пробуем инициализировать нашу `ViewModel` в `Activity`:

```
class MainActivity : ComponentActivity() {  
  
    private lateinit var viewModel: MyViewModel  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main)  
        viewModel =  
        ViewModelProvider.create(this).get(MyViewModel::class)  
    }  
  
}
```

Тут на первый взгляд можно ожидать, что будет краш при запуске приложения, так как если `ViewModel` на вход принимает какой-либо параметр, то нужна фабрика `ViewModel`, он же `ViewModelProvider.Factory`, где мы вручную должны каким-то образом положить требуемый параметр в конструктор. И в нашем примере

конструктор не пустой, но если мы запустим этот код, то никакого краша и ошибки не будет, всё запустится и инициализируется должным образом. Почему так?

Разработчики из google знали что часто понадобится передавать `SavedStateHandle` в `ViewModel`, и что бы разработчикам не приходилось каждый раз создавать фабрику для передачи - имеется готовая фабрика которая работает под капотом, так же имеются готовые классы вроде

`AbstractSavedStateViewModelFactory` - начиная с `lifecycle-viewmodel-savedstate-android-2.9.0` - объявлен устаревшим `SavedStateViewModelFactory` - актуален на данный момент для создания `ViewModel` с `SavedStateHandle`

Давайте теперь посмотрим как это работает на уровне `Activity`, логику

`ViewModelProvider/ViewModel` мы уже рассматривали в прошлых статьях, сейчас просто пройдемся по интересующей нас теме, когда мы обращаемся к `ViewModelProvider.create`:

```
public expect class ViewModelProvider {
    public companion object {
        ...
        public fun create(
            owner: ViewModelStoreOwner,
            factory: Factory =
                ViewModelProviders.getDefaultFactory(owner),
            extras: CreationExtras =
                ViewModelProviders.getDefaultCreationExtras(owner),
        ): ViewModelProvider
    }
}
```

То видим что в качестве `factory` идет обращение к методу

`ViewModelProviders.getDefaultFactory(owner)`, посмотрим его исходники тоже:

```
internal object ViewModelProviders {
    internal fun getDefaultFactory(owner: ViewModelStoreOwner):
        ViewModelProvider.Factory =
        if (owner is HasDefaultViewModelProviderFactory) {
            owner.defaultViewModelProviderFactory
        }
```

```

    } else {
        DefaultViewModelProviderFactory
    }
}

```

i `ViewModelProviders` — это утилитный класс, не стоит путать его с `ViewModelProvider`.

В этом методе нас интересует проверка на `is HasDefaultViewModelProviderFactory`:

```

if (owner is HasDefaultViewModelProviderFactory) {
    owner.defaultViewModelProviderFactory
}

```

Если `owner` (`ViewModelStoreOwner`, например `Activity` или `Fragment`) реализует интерфейс `HasDefaultViewModelProviderFactory`, то у него берётся поле `defaultViewModelProviderFactory`. Интерфейс `HasDefaultViewModelProviderFactory` выглядит следующим образом:

`androidx.lifecycle.HasDefaultViewModelProviderFactory.android.kt`

```

public interface HasDefaultViewModelProviderFactory {

    public val defaultViewModelProviderFactory:
    ViewModelProvider.Factory

    public val defaultViewModelCreationExtras: CreationExtras
        get() = CreationExtras.Empty
}

```

Реализация интерфейса `HasDefaultViewModelProviderFactory` в `Activity`:

```

open class ComponentActivity() : ..., SavedStateRegistryOwner,
HasDefaultViewModelProviderFactory, ... {
    ...
    override val defaultViewModelProviderFactory:
    ViewModelProvider.Factory by lazy {

```



```

        SavedStateViewModelFactory(application, this, if (intent
!= null) intent.extras else null)
    }

    @get:CallSuper
    override val defaultViewModelCreationExtras: CreationExtras
    /**
     * {@inheritDoc}
     *
     * The extras of [getIntent] when this is first called
will be used as the defaults to any
     * [androidx.lifecycle.SavedStateHandle] passed to a view
model created using this extra.
     */
    get() {
        val extras = MutableCreationExtras()
        if (application != null) {
            extras[APPLICATION_KEY] = application
        }
        extras[SAVED_STATE_REGISTRY_OWNER_KEY] = this
        extras[VIEW_MODEL_STORE_OWNER_KEY] = this
        val intentExtras = intent?.extras
        if (intentExtras != null) {
            extras[DEFAULT_ARGS_KEY] = intentExtras
        }
        return extras
    }
    ...
}

```

Тут происходят два очень важных момента:

1. `defaultViewModelProviderFactory` — в качестве фабрики по умолчанию используется `SavedStateViewModelFactory`.
2. `defaultViewModelCreationExtras` — в `CreationExtras` кладётся `SavedStateRegistryOwner` под ключом `SAVED_STATE_REGISTRY_OWNER_KEY` и

ViewModelStoreOwner под ключом VIEW_MODEL_STORE_OWNER_KEY.

Это ключевая часть того как в итоге SavedStateHandle подключается к ViewModel и к SavedStateRegistryOwner

Чтобы понять, как SavedStateHandle создаётся и восстанавливается для ViewModel, давайте разберёмся, что происходит в SavedStateViewModelFactory

androidx.lifecycle.SavedStateViewModelFactory.android.kt:

```
public actual class SavedStateViewModelFactory :
    ViewModelProvider.OnQueryFactory, ViewModelProvider.Factory
{
    override fun <T : ViewModel> create(modelClass: Class<T>,
extras: CreationExtras): T {
        ...
        return if (
            extras[SAVED_STATE_REGISTRY_OWNER_KEY] != null &&
            extras[VIEW_MODEL_STORE_OWNER_KEY] != null
        ) {
            ...
            newInstance(modelClass, constructor,
extras.createSavedStateHandle())
            ...
        }
        ...
    }
}

internal fun <T : ViewModel?> newInstance(
    modelClass: Class<T>,
    constructor: Constructor<T>,
    vararg params: Any
): T {
    return try {
        constructor.newInstance(*params)
    }
}
```

```
...  
}
```

Тут сокращена логика из исходников, чтобы сосредоточиться на главном. Внутри метода `create` у фабрики проверяется, содержат ли `extras` поля с ключами `SAVED_STATE_REGISTRY_OWNER_KEY` и `VIEW_MODEL_STORE_OWNER_KEY`. Если содержат — вызывается метод `newInstance`, который через рефлексию вызывает конструктор и передаёт параметры, одним из которых является `SavedStateHandle`.

Но нас интересует другой момент. Обратим внимание на вызов `createSavedStateHandle()`:

```
newInstance(modelClass, constructor,  
extras.createSavedStateHandle())
```

Что происходит внутри `createSavedStateHandle()`? Чтобы понять, как создаётся `SavedStateHandle`, нужно заглянуть в исходный код этого метода:

androidx.lifecycle.SavedStateHandleSupport.kt:

```
@MainThread  
public fun CreationExtras.createSavedStateHandle():  
    SavedStateHandle {  
    val savedStateRegistryOwner =  
        this[SAVED_STATE_REGISTRY_OWNER_KEY]  
        ?: throw IllegalArgumentException(  
            "CreationExtras must have a value by  
            `SAVED_STATE_REGISTRY_OWNER_KEY`"  
        )  
    val viewModelStateRegistryOwner =  
        this[VIEW_MODEL_STORE_OWNER_KEY]  
        ?: throw IllegalArgumentException(  
            "CreationExtras must have a value by  
            `VIEW_MODEL_STORE_OWNER_KEY`"  
        )  
  
    val defaultArgs = this[DEFAULT_ARGS_KEY]  
    val key =
```

```

        this[VIEW_MODEL_KEY]
            ?: throw IllegalArgumentException(
                "CreationExtras must have a value by
`VIEW_MODEL_KEY`"
            )
        return createSavedStateHandle(
            savedStateRegistryOwner,
            viewModelStateRegistryOwner,
            key,
            defaultArgs
        )
    }

```

Здесь из CreationExtras извлекаются три ключевых объекта:

1. savedStateRegistryOwner — ссылка на SavedStateRegistry для управления состоянием.
2. viewModelStateRegistryOwner — ссылка на ViewModelStore для привязки к жизненному циклу.
3. defaultArgs — начальные параметры, если они были переданы.

Все эти зависимости передаются в другой метод `createSavedStateHandle`, который как раз и занимается созданием или восстановлением SavedStateHandle для данной ViewModel.

androidx.lifecycle.SavedStateHandleSupport.kt:

```

private fun createSavedStateHandle(
    savedStateRegistryOwner: SavedStateRegistryOwner,
    viewModelStoreOwner: ViewModelStoreOwner,
    key: String,
    defaultArgs: SavedState?
): SavedStateHandle {
    val provider =
        savedStateRegistryOwner.savedStateHandlesProvider
    val viewModel = viewModelStoreOwner.savedStateHandlesVM
    return viewModel.handles[key]
}

```

```

        ?:
        SavedStateHandle.createHandle(provider.consumeRestoredStateForKey(
            key), defaultArgs)
            .also { viewModel.handles[key] = it }
    }

```

Тут сначала ищется нужный `SavedStateHandle` внутри `SavedStateHandlesVM`. Если он не найден — создаётся новый, сохраняется в `SavedStateHandlesVM`, а функция `createSavedStateHandle` возвращает управление обратно в `CreationExtras.createSavedStateHandle()`, которую мы уже видели. В конечном итоге управление возвращается в фабрику, таким образом создаётся `SavedStateHandle` для конкретной `ViewModel`.

Также в этом методе видим вызовы вроде `savedStateRegistryOwner.savedStateHandlesProvider` и `viewModelStoreOwner.savedStateHandlesVM`.

Теперь посмотрим, как это связано с провайдером. В коде вызывается `savedStateRegistryOwner.savedStateHandlesProvider`. На самом деле это просто `extension` свойство, которая вытаскивает объект (`SavedStateProvider`) из `SavedStateRegistry`.

Этот провайдер отвечает за доступ ко всем сохранённым состояниям (`SavedStateHandle`), привязанным к разным `ViewModel`. Перейдем к провайдеру: `savedStateHandlesProvider`

androidx.lifecycle.SavedStateHandleSupport.kt:

```

internal val SavedStateRegistryOwner.savedStateHandlesProvider:
    SavedStateHandlesProvider
    get() =
        savedStateRegistry.getSavedStateProvider(SAVED_STATE_KEY) as?
        SavedStateHandlesProvider
        ?: throw IllegalStateException(
            "enableSavedStateHandles() wasn't called " +
            "prior to createSavedStateHandle() call"
        )

internal class SavedStateHandlesProvider(

```

```

        private val savedStateRegistry: SavedStateRegistry,
        viewModelStoreOwner: ViewModelStoreOwner
    ) : SavedStateRegistry.SavedStateProvider {
        private var restored = false
        private var restoredState: SavedState? = null

        private val viewModel by lazy {
            viewModelStoreOwner.savedStateHandlesVM }

        override fun saveState(): SavedState {
            return savedState {
                restoredState?.let { putAll(it) }
                viewModel.handles.forEach { (key, handle) ->
                    val savedState =
                        handle.savedStateProvider().saveState()
                    if (savedState.read { !isEmpty() }) {
                        putSavedState(key, savedState)
                    }
                }
                restored = false
            }
        }

        fun performRestore() {
            ...
        }

        fun consumeRestoredStateForKey(key: String): SavedState? {
            ...
        }
    }
}

```

`SavedStateHandlesProvider` — это прослойка между `SavedStateRegistry` и `SavedStateHandle`, обеспечивающая централизованное сохранение и восстановление состояний `ViewModel`. В методе `saveState()` собираются все актуальные состояния из `viewModel.handles`, добавляется возможное ранее восстановленное состояние, и итог сохраняется в `SavedStateRegistry`.

Для выборочного восстановления используется метод `consumeRestoredStateForKey()`, позволяющий получить состояние по ключу без необходимости загружать всё сразу. Восстановление и подготовка состояний происходят в `performRestore()`.

По сути, `SavedStateHandlesProvider` управляет жизненным циклом всех `SavedStateHandle` в рамках владельца состояния, поддерживая логику ленивого восстановления и гарантируя корректное сохранение после процесса или конфигурационных изменений.

Взаимодействие с `SavedStateHandlesVM`:

Теперь перейдём к тому, как данные хранятся внутри `ViewModel`. `savedStateHandlesVM` — это расширение, которое создаёт или восстанавливает объект `SavedStateHandlesVM`, хранящий в себе Map из ключей на `SavedStateHandle`:

```
internal val ViewModelStoreOwner.savedStateHandlesVM:
SavedStateHandlesVM
get() =
    ViewModelProvider.create(
        owner = this,
        factory =
            object : ViewModelProvider.Factory {
                override fun <T : ViewModel> create(
                    modelClass: KClass<T>,
                    extras: CreationExtras
                ): T {
                    @Suppress("UNCHECKED_CAST") return
SavedStateHandlesVM() as T
                }
            }
    )[VIEWMODEL_KEY, SavedStateHandlesVM::class]

internal class SavedStateHandlesVM : ViewModel() {
    val handles = mutableMapOf<String, SavedStateHandle>()
}
```

Здесь создаётся объект `SavedStateHandlesVM`, внутри которого поддерживается `Map`, связывающая ключи с объектами `SavedStateHandle`. `SavedStateHandlesVM` нужен для того, чтобы хранить и управлять всеми `SavedStateHandle` всех `ViewModel` в рамках одного `ViewModelStoreOwner` и `SavedStateRegistryOwner`.

`SavedStateHandlesProvider` — класс, реализующий интерфейс `SavedStateProvider`. Когда `SavedStateController` вызывает `performSave`, он также обращается к `SavedStateHandlesProvider` и вызывает его метод `saveState`. Далее он кладёт все существующие `SavedStateHandle` в объект `SavedState` (`Bundle`) и возвращает его.

Но чтобы весь этот процесс работал, необходимо зарегистрировать `SavedStateHandlesProvider` в `SavedStateRegistry`, однако пока что в коде мы не встретили блок, отвечающий за регистрацию провайдера, то есть вызов метода: `savedStateRegistry.registerSavedStateProvider(...)`

На самом деле такая логика есть, и она триггерится внутри `ComponentActivity`, `Fragment` и `NavBackStackEntry`, то есть во всех `SavedStateRegistryOwner`. Давайте просто глянем, как это вызывается в `ComponentActivity`:

```
open class ComponentActivity() : ..., SavedStateRegistryOwner, ...
{
    init {
        ...
        enableSavedStateHandles()
        ...
    }
}
```

Видим вызов некоего метода `enableSavedStateHandles` — само название звучит заманчиво. Далее — исходники метода `enableSavedStateHandles`:

```
@MainThread
public fun <T> T.enableSavedStateHandles() where T :
    SavedStateRegistryOwner, T : ViewModelStoreOwner {
    ...
    if (savedStateRegistry.getSavedStateProvider(SAVED_STATE_KEY)
        == null) {
        val provider =
```



```

SavedStateHandlesProvider(savedStateRegistry, this)

savedStateRegistry.registerSavedStateProvider(SAVED_STATE_KEY,
provider)
    lifecycle.addObserver(SavedStateHandleAttacher(provider))
}
}

```

`enableSavedStateHandles` — это типизированный метод, который требует, чтобы вызывающая область одновременно являлась и `SavedStateRegistryOwner`, и `ViewModelStoreOwner`. `ComponentActivity/Fragment/NavBackStackEntry` идеально подходят под это — все трое реализуют оба интерфейса.

Давайте вкратце поймём, что происходит в этом методе. Для начала у `SavedStateRegistry` запрашивается сохранённый `provider` (`SavedStateProvider`) по ключу `SAVED_STATE_KEY`. Это ключ для хранения `SavedStateHandlesProvider` (он же `SavedStateProvider`).

Если по ключу ничего не найдено, то есть `null`, это означает, что `provider` ещё не был зарегистрирован. Тогда создаётся объект `SavedStateHandlesProvider` (он же `SavedStateProvider`) и регистрируется в `savedStateRegistry`.

Мы подробно разобрали, как механизм `SavedStateHandle` автоматически создаётся и подключается к `ViewModel`. Это достигается за счёт встроенного механизма фабрики `SavedStateViewModelFactory`, которая при создании `ViewModel` извлекает необходимые зависимости из объекта `CreationExtras`. Эти зависимости включают в себя:

1. **`SavedStateRegistryOwner`** — для управления сохранением и восстановлением состояния.
2. **`ViewModelStoreOwner`** — для привязки жизненного цикла `ViewModel`.
3. **`DefaultArgs`** — начальные параметры, если они были переданы.

В момент инициализации `ViewModel`, фабрика `SavedStateViewModelFactory` через метод `createSavedStateHandle` формирует объект `SavedStateHandle`. Этот объект связывается с `SavedStateRegistry` и регистрируется в нём посредством специального провайдера — `SavedStateHandlesProvider` (`SavedStateProvider`).

Механизм регистрации провайдера запускается автоматически при создании `ComponentActivity`, `Fragment` или `NavBackStackEntry`. Это обеспечивается вызовом метода `enableSavedStateHandles`, который регистрирует провайдер в `SavedStateRegistry` под ключом `SAVED_STATE_KEY`. В дальнейшем, при вызове `onSaveInstanceState`, этот провайдер сохраняет все текущие состояния из `SavedStateHandle`, привязанные к ключам `ViewModel`.

Таким образом, когда компонент пересоздаётся (например, при смене ориентации экрана или в случае уничтожения и восстановления `Activity`), механизм восстановления срабатывает автоматически. `SavedStateRegistry` восстанавливает состояние из провайдера, а `SavedStateHandle` вновь связывается с `ViewModel`, обеспечивая прозрачную работу с сохранёнными данными.

Это позволяет нам не заботиться о ручной передаче сохранённого состояния при каждом пересоздании `ViewModel`. Android-фреймворк делает это за нас, используя мощный механизм фабрик и хранилищ состояний, что делает `SavedStateHandle` удобным и надёжным инструментом для управления состоянием внутри `ViewModel`.

На текущий момент мы понимаем, как `SavedStateHandle` работает в связке с `ViewModel` и как он в итоге соединяется с `SavedStateRegistry`. Также до этого мы узнали, как работают сам `SavedStateRegistry` и `SavedStateRegistryController`, и увидели их связь с методами `onSaveInstanceState` и `onRestoreInstanceState`.

Оказалось, что и `Saved State API`, и древние методы `onSaveInstanceState`/`onRestoreInstanceState` в конечном итоге работают по одному и тому же пути. Давайте вернёмся к точке, где они встречаются. Далее — код, который мы уже видели:

```
open class ComponentActivity() : ..., SavedStateRegistryOwner, ...
{

    override fun onCreate(savedInstanceState: Bundle?) {

        savedInstanceStateRegistryController.performRestore(savedInstanceState)
        super.onCreate(savedInstanceState)

        ...
    }
}
```

```

@CallSuper
override fun onSaveInstanceState(outState: Bundle) {
    ...
    super.onSaveInstanceState(outState)
    savedInstanceStateRegistryController.performSave(outState)
}
}

```

То есть в стандартной практике при использовании механизма сохранения состояния применяются два метода:

- `onCreate` — получает на вход параметр `savedInstanceState` типа `Bundle`. Именно в этом методе читаются сохранённые значения.
- `onSaveInstanceState` — получает на вход параметр `outState` типа `Bundle`. В этот параметр записываются значения, которые должны быть сохранены.

Давайте разберёмся, каким образом вся эта конструкция работает: как значения, сохранённые в `outState` метода `onSaveInstanceState`, переживают изменение конфигурации и даже смерть процесса, и как эти сохранённые данные возвращаются обратно в `onCreate`.

Посмотрим на реализацию метода `onSaveInstanceState` в `super`, то есть в самом классе `Activity`:

```

public class Activity extends ContextThemeWrapper ...{

    final void performSaveInstanceState(@NonNull Bundle outState) {
        ...
        onSaveInstanceState(outState);
        ...
    }

    protected void onSaveInstanceState(@NonNull Bundle outState) {
        ...
    }
}

```

Всё, что происходит внутри этого метода, нас сейчас не волнует. Главное, что `onSaveInstanceState` вызывает другой финальный метод — `performSaveInstanceState`.

Теперь давайте поймём, кто вызывает `performSaveInstanceState`. Этот вызов инициируется классом `Instrumentation`:

android.app.Instrumentation.java:

```
@android.ravenwood.annotation.RavenwoodKeepPartialClass
public class Instrumentation {
    ...

    public void callActivityOnSaveInstanceState(@NonNull Activity
activity,
                                                @NonNull Bundle
outState) {
        activity.performSaveInstanceState(outState);
    }
    ...
}
```

i Официальная документация гласит следующее об этом классе:

Base class for implementing application instrumentation code. When running with instrumentation turned on, this class will be instantiated for you before any of the application code, allowing you to monitor all of the interaction the system has with the application. An Instrumentation implementation is described to the system through an `AndroidManifest.xml`'s tag.

Теперь нужно понять, кто же вызывает

`Instrumentation.callActivityOnSaveInstanceState`? И тут мы встречаем `ActivityThread`:

```
public final class ActivityThread extends ClientTransactionHandler
implements ActivityThreadInternal {
    ...
}
```

```

        private void
        callActivityOnSaveInstanceState(ActivityClientRecord r) {
            r.state = new Bundle();
            r.state.setAllowFds(false);
            if (r.isPersistable()) {
                r.persistentState = new PersistableBundle();
                mInstrumentation.callActivityOnSaveInstanceState(
                    r.activity, r.state,
                    r.persistentState
                );
            } else {

mInstrumentation.callActivityOnSaveInstanceState(r.activity,
r.state);
            }
        }
        ...
    }

```

Что здесь происходит? `callActivityOnSaveInstanceState` на вход принимает параметр `r` типа `ActivityClientRecord`. У этого класса есть поле `state`, которое является `Bundle`. Ему присваивается новый объект `Bundle`.

Класс `ActivityClientRecord` мы уже встречали, когда рассматривали `ViewModelStore`. `ActivityClientRecord` представляет собой запись активности и используется для хранения всей информации, связанной с реальным экземпляром активности. Это своего рода структура данных для учёта активности в процессе выполнения приложения.

Основные поля класса `ActivityClientRecord`:

- `state` — объект `Bundle`, содержащий сохраненное состояние активности. Да, да, это тот самый `Bundle` который мы получаем в методе `onCreate`, `onRestoreInstanceState` и `onSaveInstanceState`
- `lastNonConfigurationInstances` — объект `Activity#NonConfigurationInstance`, в котором хранится `ComponentActivity#NonConfigurationInstances` в котором хранится `ViewModelStore`.

- `intent` — объект `Intent`, представляющий намерение запуска активности.
- `window` — объект `Window`, связанный с активностью.
- `activity` — сам объект `Activity`.
- `parent` — родительская активность (если есть).
- `createdConfig` — объект `Configuration`, содержащий настройки, примененные при создании активности.
- `overrideConfig` — объект `Configuration`, содержащий текущие настройки активности.

Пока что не будем отвлекаться, и узнаем кто же вызывает `callActivityOnSaveInstanceState`:

```
public final class ActivityThread extends ClientTransactionHandler
implements ActivityThreadInternal {

    private void callActivityOnStop(ActivityClientRecord r,
boolean saveState, String reason) {
        final boolean shouldSaveState = saveState &&
!r.activity.mFinished && r.state == null
            && !r.isPreHoneycomb();
        final boolean isPreP = r.isPreP();
        if (shouldSaveState && isPreP) {
            callActivityOnSaveInstanceState(r);
        }
        ...
    }

    private Bundle performPauseActivity(ActivityClientRecord r,
boolean finished, String reason,
                                PendingTransactionActions
pendingActions) {
        ...
        final boolean shouldSaveState = !r.activity.mFinished &&
r.isPreHoneycomb();
```

```

        if (shouldSaveState) {
            callActivityOnSaveInstanceState(r);
        }
        ...
    }
}

```

Метод `callActivityOnStop` определяет, нужно ли сохранять состояние активности перед остановкой. Проверяется флаг `saveState`, активность не должна быть завершена (`!mFinished`), состояние (`r.state`) должно быть ещё не сохранено, и версия должна быть до Honeycomb (`!isPreHoneycomb()`). Если все условия выполняются и версия до Android P (`isPreP()`), вызывается `callActivityOnSaveInstanceState`, чтобы создать и заполнить `Bundle`

Метод `performPauseActivity` проверяет, нужно ли сохранить состояние перед паузой. Здесь условия упрощены: активность не должна быть завершена, версия — до Honeycomb. Если да, то снова вызывается `callActivityOnSaveInstanceState` для формирования `Bundle`.

```

public final class ActivityThread extends ClientTransactionHandler
implements ActivityThreadInternal {

    private void performStopActivityInner(ActivityClientRecord r,
        StopInfo info,
                                     boolean saveState,
        boolean finalStateRequest, String reason) {
        ...
        callActivityOnStop(r, saveState, reason);
    }

    private void handleRelaunchActivityInner(@NonNull
        ActivityClientRecord r,...) {
        ...
        if (!r.stopped) {
            callActivityOnStop(r, true /* saveState */, reason);
        }
        ...
    }
}

```

```

    }
}

```

performStopActivityInner используется при полной остановке активности. Внутри сразу вызывается callActivityOnStop, который проверяет и, если нужно, инициирует сохранение состояния. Это гарантирует, что состояние активности попадёт в Bundle до того, как активность будет остановлена и уничтожена.

В handleRelaunchActivityInner вызывается callActivityOnStop, если активность ещё не остановлена (!r.stopped). Это важно при пересоздании активности (например, при изменении конфигурации), чтобы сохранить состояние до пересоздания.

```

public final class ActivityThread extends ClientTransactionHandler
implements ActivityThreadInternal {
    @Override
    public void handleRelaunchActivity(@NonNull
ActivityClientRecord tmp,
                                     @NonNull
PendingTransactionActions pendingActions) {
        ...
        handleRelaunchActivityInner(r, tmp.pendingResults,
tmp.pendingIntents,
                                pendingActions, tmp.startsNotResumed,
tmp.overrideConfig, tmp.mActivityWindowInfo,
                                "handleRelaunchActivity");
    }

    @Override
    public void handleStopActivity(ActivityClientRecord r,
PendingTransactionActions
pendingActions, boolean finalStateRequest, String reason) {
        ...
        performStopActivityInner(r, stopInfo, true /* saveState
*/, finalStateRequest,
                                reason);
        ...
    }
}

```



```

    }
}

```

`handleRelaunchActivity` — внешний метод, который вызывает `handleRelaunchActivityInner`. Используется для обработки полного пересоздания активности. Все проверки и логика сохранения состояния уже находятся внутри `handleRelaunchActivityInner`.

`handleStopActivity` вызывает `performStopActivityInner`, передавая туда флаг `saveState = true`, чтобы принудительно сохранить состояние перед окончательной остановкой. Это используется, например, при закрытии приложения или выгрузке активности системой.

Последующие вызовы методов `performStopActivity` и `handleRelaunchActivity` упираются в классы `ActivityRelaunchItem.execute()` и `StopActivityItem.execute()`. Метод `performStopActivity` вызывается из `StopActivityItem.execute()`, а `handleRelaunchActivity` — из `ActivityRelaunchItem.execute()`.

```

public class StopActivityItem extends ActivityLifecycleItem {
    @Override
    public void execute(@NonNull ClientTransactionHandler client,
        @NonNull ActivityClientRecord r,
        @NonNull PendingTransactionActions
        pendingActions) {
        client.handleStopActivity(r, pendingActions,
            true /* finalStateRequest */,
            "STOP_ACTIVITY_ITEM");
        Trace.traceEnd(TRACE_TAG_ACTIVITY_MANAGER);
    }
}

```

В методе `StopActivityItem.execute` видим вызов `client.handleStopActivity`. Так как `client` — это `ClientTransactionHandler`, а `ActivityThread` наследуется от него, фактически здесь вызывается `ActivityThread.handleStopActivity`.

```

public class ActivityRelaunchItem extends ActivityTransactionItem
{
    @Override

```

```

    public void execute(@NonNull ClientTransactionHandler client,
        @NonNull ActivityClientRecord r,
        @NonNull PendingTransactionActions
        pendingActions) {
        client.handleRelaunchActivity(mActivityClientRecord,
            pendingActions);
    }
}

```

В методе `ActivityRelaunchItem.execute` видим вызов `client.handleRelaunchActivity`. По той же логике, фактически вызывается `ActivityThread.handleRelaunchActivity`.

На данный момент мы выследили следующую цепочку вызовов:

```

StopActivityItem.execute → ActivityThread.handleStopActivity →
ActivityThread.performStopActivityInner → ActivityThread.callActivityOnStop →
ActivityThread.callActivityOnSaveInstanceState →
Instrumentation.callActivityOnSaveInstanceState → Activity.performSaveInstanceState
→ Activity.onSaveInstanceState.

```

Это ключевая цепочка, которая обеспечивает сохранение состояния `Activity` при изменениях конфигурации или её завершении. Обратим внимание, что вызов `callActivityOnSaveInstanceState` из `Instrumentation` — это как раз та точка, где система передаёт управление обратно в `Activity`, вызывая метод `performSaveInstanceState`, который иницирует сохранение всех данных в объект `Bundle`.

Параллельно, в случае изменения конфигурации или пересоздания активности, запускается другая цепочка:

```

ActivityRelaunchItem.execute → ActivityThread.handleRelaunchActivity →
ActivityThread.handleRelaunchActivityInner → ActivityThread.callActivityOnStop →
ActivityThread.callActivityOnSaveInstanceState →
Instrumentation.callActivityOnSaveInstanceState → Activity.performSaveInstanceState
→ Activity.onSaveInstanceState.

```

Эти две цепочки работают независимо, но сходятся в методе `callActivityOnStop`, который гарантирует сохранение данных в `Bundle` перед тем, как `Activity` будет остановлена или пересоздана.

Далее, сформированный объект `Bundle`, содержащий состояние `Activity`, сохраняется в объекте `ActivityClientRecord`. Этот объект представляет собой структуру данных, хранящую всю необходимую информацию о `Activity` во время её жизненного цикла. Именно в поле `state` этого класса система сохраняет переданный `Bundle`, чтобы при пересоздании активности восстановить её состояние. `ActivityClientRecord` существует в процессе всех вызовов цепочки, перед тем как `Activity` перейдёт в состояние `STOP`. Внутри метода `ActivityThread.callActivityOnSaveInstanceState` полю `ActivityClientRecord.state` присваивается новый `Bundle`, в который активити и фрагменты кладут всё нужное — от состояния иерархий `View` до любых данных, которые разработчик решил сохранить.

Таким образом, мы видим, что эта цепочка запускается не из самой `Activity`, а из внутренней логики Android через `ActivityThread`. Это ещё раз подтверждает, что все жизненные циклы управляются системой через единый механизм клиент-серверных транзакций, а `ActivityThread` выполняет роль посредника, координирующего вызовы между `Activity` и системой.

Важный момент здесь — откуда берётся `ActivityClientRecord` и как его внутренний `Bundle` переживает смерть процесса. В случае сохранения между `PAUSE/STOP` мы увидели, где создаётся чистый `Bundle`, в который можно сохранять данные. Здесь особых секретов нет. Но то, как этот сохранённый `Bundle` внутри `ActivityClientRecord` переживает смерть системы и затем возвращается в `Activity.onCreate`, мы ещё не знаем. Следующая глава раскроет этот момент.

Цепочка вызова onCreate

Начнем наше движение с самого низа — с метода `onCreate`. Как видно из кода, его вызов происходит внутри метода `performCreate`, который, в свою очередь, вызывается из метода `callActivityOnCreate` класса `Instrumentation`.

```
public class Activity extends ContextThemeWrapper ...{

    public void onCreate(@Nullable Bundle savedInstanceState,
        @Nullable PersistableBundle persistentState) {
        onCreate(savedInstanceState);
    }
}
```

```

@MainThread
@CallSuper
protected void onCreate(@Nullable Bundle savedInstanceState) {
    ...
}

final void performCreate(Bundle icle) {
    performCreate(icle, null);
}

@UnsupportedAppUsage(maxTargetSdk = Build.VERSION_CODES.R,
trackingBug = 170729553)
final void performCreate(Bundle icle, PersistableBundle
persistentState) {
    ...
    if (persistentState != null) {
        onCreate(icle, persistentState);
    } else {
        onCreate(icle);
    }
    ...
}
}

```

Метод `performCreate` является связующим звеном между логикой вызова `onCreate` и более низкоуровневыми компонентами системы. Сам же вызов `performCreate` осуществляется в классе `Instrumentation`:

```

public class Instrumentation {
    ...

    public void callActivityOnCreate(Activity activity, Bundle
icle) {
        ...
        activity.performCreate(icle);
        ...
    }
}

```

```

    }
}

```

Класс `Instrumentation` управляет жизненным циклом `Activity` и вызывает `performCreate`, передавая ему объект `Bundle` для восстановления состояния. Теперь поднимемся выше. Кто же вызывает `callActivityOnCreate`? За это отвечает метод `performLaunchActivity` в классе `ActivityThread`:

```

public final class ActivityThread extends ClientTransactionHandler
implements ActivityThreadInternal {

    private Activity performLaunchActivity(ActivityClientRecord r,
Intent customIntent) {
        ...
        if (r.isPersistable()) {
            mInstrumentation.callActivityOnCreate(activity,
r.state, r.persistentState);
        } else {
            mInstrumentation.callActivityOnCreate(activity,
r.state);
        }
        ...
    }
}

```

Здесь мы видим, что в зависимости от состояния активности (сохранено ли оно в `PersistentState`), `callActivityOnCreate` вызывается с разным количеством параметров, но всегда через `Instrumentation`.

Далее, этот метод `performLaunchActivity` вызывается из метода `handleLaunchActivity` того же класса:

```

public final class ActivityThread extends ClientTransactionHandler
implements ActivityThreadInternal {

    @Override
    public Activity handleLaunchActivity(ActivityClientRecord r,
...) {

```

```

        ...
        final Activity a = performLaunchActivity(r, customIntent);
        ...
    }
}

```

Перезапуск Activity при релаунче (например, при повороте экрана)

При пересоздании Activity, например, при повороте экрана, срабатывает метод `handleRelaunchActivity`:

```

public final class ActivityThread extends ClientTransactionHandler
implements ActivityThreadInternal {

    @Override
    public void handleRelaunchActivity(@NonNull
ActivityClientRecord tmp,
                                     @NonNull
PendingTransactionActions pendingActions) {
        ...
        handleRelaunchActivityInner(r, tmp.pendingResults,
tmp.pendingIntents,
                                pendingActions, tmp.startsNotResumed,
tmp.overrideConfig, tmp.mActivityWindowInfo,
                                "handleRelaunchActivity");
    }

    private void handleRelaunchActivityInner(@NonNull
ActivityClientRecord r,...) {
        ....
        handleLaunchActivity(r, pendingActions,
mLastReportedDeviceId, customIntent);
    }
}

```

Вызов метода `handleRelaunchActivity` иницирует класс команда/транзакция `ActivityRelaunchItem`, которая действует как маркер для того, чтобы выполнить

перезапуск с сохранением состояния:

```
public class ActivityRelaunchItem extends ActivityTransactionItem
{
    @Override
    public void execute(@NonNull ClientTransactionHandler client,
        @NonNull ActivityClientRecord r,
        @NonNull PendingTransactionActions
        pendingActions) {
        ...
        client.handleRelaunchActivity(mActivityClientRecord,
        pendingActions);
        ...
    }
}
```

Эта команда инициирует следующую цепочку вызовов:

ActivityRelaunchItem.execute → handleRelaunchActivity → handleRelaunchActivityInner
→ handleLaunchActivity → performLaunchActivity → callActivityOnCreate →
performCreate → onCreate.

Создание Activity после уничтожения процесса или при первом запуске

В случае, если процесс был уничтожен или это первый запуск Activity, используется другая команда — LaunchActivityItem. Она запускает аналогичную, но отдельную цепочку вызовов:

```
public class LaunchActivityItem extends ClientTransactionItem {

    @Nullable
    private final Bundle mState;

    @Nullable
    private final PersistableBundle mPersistentState;

    public LaunchActivityItem(
```

```

        // остальные параметры
        @Nullable Bundle state,
        @Nullable PersistableBundle persistentState,
        // остальные параметры
    ) {
        this(
            // передаваемые аргументы до
            state != null ? new Bundle(state) : null,
            persistentState != null ? new
PersistableBundle(persistentState) : null,
            // оставшиеся аргументы
        );
        ...
    }

    @Override
    public void execute(@NonNull ClientTransactionHandler client,
        @NonNull PendingTransactions
pendingActions) {
        ...
        ActivityClientRecord r = new
ActivityClientRecord(...,mState, mPersistentState, ...);
        client.handleLaunchActivity(r, pendingActions, mDeviceId,
null /* customIntent */);
        ...
    }
}

```

Цепочка выглядит так: `LaunchActivityItem.execute` → `ActivityThread.handleLaunchActivity` → `ActivityThread.performLaunchActivity` → `ActivityThread.callActivityOnCreate` → `Activity.performCreate` → `Activity.onCreate`.

Следует запомнить важную вещь, прежде чем подниматься выше, нужно понимать что `LaunchActivityItem` — это транзакция, которая в своём конструкторе принимает `Bundle` и `PersistableBundle` (последний мы рассматривать не будем). Класс `LaunchActivityItem` наследуется от `ClientTransactionItem`.

`ClientTransactionItem` — это абстрактный базовый класс, от которого наследуются все транзакции, связанные с жизненным циклом `Activity`. В него входят `LaunchActivityItem`, `ActivityRelaunchItem`, `ResumeActivityItem` (последние — не прямые, а транзитивные наследники) и другие элементы, участвующие в управлении состоянием `Activity`.

Мы увидели что создание `ActivityClientRecord` происходит в `LaunchActivityItem.execute`, но она использует готовые данные которые были переданы ей в конструктор при созданий.

Наша цель дальше — выяснить два момента:

1. Кто создаёт `LaunchActivityItem` и передаёт в него `Bundle`, который как раз и переживает смерть или остановку процесса.
2. Кто вызывает метод `execute` у `LaunchActivityItem` и запускает описанную выше цепочку вызовов : `LaunchActivityItem.execute` → `handleLaunchActivity` → `performLaunchActivity` → `callActivityOnCreate` → `performCreate` → `onCreate`.

И так идем дальше, выше вызова `LaunchActivityItem.execute`, стоит класс `TransactionExecutor`

```
public class TransactionExecutor {

    private final ClientTransactionHandler mTransactionHandler;

    public TransactionExecutor(@NonNull ClientTransactionHandler
clientTransactionHandler) {
        mTransactionHandler = clientTransactionHandler;
    }

    public void execute(@NonNull ClientTransaction transaction) {
        ...
        executeTransactionItems(transaction);
        ...
    }

    public void executeTransactionItems(@NonNull ClientTransaction
transaction) {
```

```

        final List<ClientTransactionItem> items =
transaction.getTransactionItems();
        final int size = items.size();
        for (int i = 0; i < size; i++) {
            final ClientTransactionItem item = items.get(i);
            if (item.isActivityLifecycleItem()) {
                executeLifecycleItem(transaction,
(ActivityLifecycleItem) item);
            } else {
                executeNonLifecycleItem(transaction, item,
shouldExcludeLastLifecycleState(items,
i));
            }
        }
    }

    private void executeLifecycleItem(@NonNull ClientTransaction
transaction,
                                     @NonNull
ActivityLifecycleItem lifecycleItem) {
        ...
        lifecycleItem.execute(mTransactionHandler,
mPendingActions);
        ...
    }

    private void executeNonLifecycleItem(@NonNull
ClientTransaction transaction,
                                         @NonNull
ClientTransactionItem item, boolean
shouldExcludeLastLifecycleState) {
        ...
        item.execute(mTransactionHandler, mPendingActions);
        ...
    }
}

```

`TransactionExecutor` – это как раз класс который работает со всеми транзакциями, то есть с `ClientTransactionItem`, и `ClientTransaction` – который является массивом или очередью которая хранит `ClientTransactionItem`-ы,

Конструктор `TransactionExecutor` принимает на вход `ClientTransactionHandler`, если вы не забыли, то `ActivityThread` реализует абстрактный класс

`ClientTransactionHandler`, по этому фактический в конструктор `TransactionExecutor` прилетает `ActivityThread`.

У `TransactionExecutor` есть метод `execute` который вызывает другой метод `executeTransactionItems`, `executeTransactionItems` – в свою очередь пробегается по всем элементам внутри очереди транзакций, то есть в `ClientTransaction`, и в итоге определяет какой метод вызывать, `executeNonLifecycleItem` или `executeLifecycleItem`.

Различие этих методов в том, что `executeLifecycleItem` вызывается для транзакций, представляющих этапы жизненного цикла активности — такие как `ResumeActivityItem`, `PauseActivityItem`, `StopActivityItem`, `DestroyActivityItem`. Эти элементы отвечают за переходы между состояниями уже существующей `Activity`. Их назначение — вызвать соответствующие колбэки (`onPause`, `onStop`, и так далее) на объекте активности, который уже был создан и существует в памяти.

С другой стороны, `executeNonLifecycleItem` используется для выполнения транзакций, которые не относятся к жизненному циклу. Главный представитель — `LaunchActivityItem`, который отвечает за создание `Activity` с нуля. Это может происходить либо при первом запуске `Activity`, либо после того, как система уничтожила процесс, и теперь восстанавливает его. Внутри `executeNonLifecycleItem` вызывается `item.execute(...)`, который, в случае `LaunchActivityItem`, инициирует полную цепочку создания: от `ActivityClientRecord` до вызова `onCreate`.

Внутри `LaunchActivityItem`, в методе `executeNonLifecycleItem`, мы видим, что у `item` (экземпляр `ClientTransactionItem`) вызывается метод `execute`, которому передаются `ClientTransactionHandler` и `PendingTransactionActions`. Фактически в этот момент вызывается метод `execute` у `LaunchActivityItem`. Не забываем, что `LaunchActivityItem` наследуется от `ClientTransactionItem`.

Теперь разберёмся, кто вызывает метод `execute` у `TransactionExecutor`. Это делает внутренний класс `H`, являющийся `Handler`-ом:

```

public final class ActivityThread extends ClientTransactionHandler
implements ActivityThreadInternal {

    final H mH = new H();
    private final TransactionExecutor mTransactionExecutor = new
TransactionExecutor(this);

    class H extends Handler {

        public void handleMessage(Message msg) {
            switch (msg.what) {
                ...
                case EXECUTE_TRANSACTION:
                    final ClientTransaction transaction =
(ClientTransaction) msg.obj;
                    final ClientTransactionListenerController
controller = ClientTransactionListenerController.getInstance();
                    controller.onClientTransactionStarted();
                    try {
                        mTransactionExecutor.execute(transaction);
                    } finally {
                        controller.onClientTransactionFinished();
                    }
                ...
            }
        }
    }
}

```

Напомним, что `ClientTransactionHandler` — это абстрактный класс, от которого наследуется `ActivityThread`. Далее мы видим, что создаётся объект `H`, а также `TransactionExecutor`, которому в качестве аргумента передаётся `this`, то есть `ActivityThread`, реализующий `ClientTransactionHandler`.

Теперь обратим внимание на реализацию `handleMessage` внутри класса `H`: когда приходит сообщение с типом `EXECUTE_TRANSACTION`, из объекта `Message`

извлекается `ClientTransaction`, содержащий в себе список (`List`) транзакций. Затем вызывается метод `execute` у `TransactionExecutor`, что и запускает выполнение транзакции.

Сам метод `handleMessage` у класса `H` вызывает методы из самого класса `ActivityThread`:

```
public final class ActivityThread extends ClientTransactionHandler
implements ActivityThreadInternal {

    final H mH = new H();

    void sendMessage(int what, Object obj) {
        sendMessage(what, obj, 0, 0, false);
    }

    private void sendMessage(int what, Object obj, int arg1) {
        sendMessage(what, obj, arg1, 0, false);
    }

    private void sendMessage(int what, Object obj, int arg1, int
arg2) {
        sendMessage(what, obj, arg1, arg2, false);
    }

    private void sendMessage(int what, Object obj, int arg1, int
arg2, boolean async) {
        ...
        mH.sendMessage(msg);
    }
}
```

Видим что последний метод `sendMessage` и вызывает у класса `H` метод `sendMessage`, так как класс `H` наследуется от класса `Handler`, то у него есть метод `sendMessage` и вызывает метод `handleMessage`, надо понять кто вызывает `sendMessage` у `ActivityThread`,

```

public final class ActivityThread extends ClientTransactionHandler
implements ActivityThreadInternal {

    void sendMessage(int what, Object obj) {
        sendMessage(what, obj, 0, 0, false);
    }

    private class ApplicationThread extends
    IApplicationThread.Stub {

        @Override
        public void scheduleTransaction(ClientTransaction
transaction) throws RemoteException {
            ActivityThread.this.scheduleTransaction(transaction);
        }
    }
}

```

Этим занимается ApplicationThread, каким образом вызов метода ActivityThread.scheduleTransaction вызывает ActivityThread.sendMessage?

Дело в том что ActivityThread наследуется от ClientTransactionHandler, а ClientTransactionHandler выглядит следующим образом:

```

public abstract class ClientTransactionHandler {

    void scheduleTransaction(ClientTransaction transaction) {
        transaction.preExecute(this);
        sendMessage(ActivityThread.H.EXECUTE_TRANSACTION,
transaction);
    }

    abstract void sendMessage(int what, Object obj);
}

```

Получается у `ApplicationThread` вызывается метод `scheduleTransaction`, он вызывает у `ActivityThread` метод `scheduleTransaction` который он унаследовал от `ClientTransactionHandler`, внутри метода `scheduleTransaction` у `ClientTransactionHandler` мы видим что он вызывает метод `sendMessage` с двумя параметрами, `ActivityThread` как раз переопределяет этот метод, и далее вызов идет в `H.sendMessage`.

`ApplicationThread` - это Proxy который реализует AIDL интерфейс, этот класс отвечает за многие планирования, например сервисы, receiver или binding Application. Так же заметьте что он реализует `IApplicationThread.Stub`, то есть фактический сам AIDL интерфейс `IApplicationThread`

Дальше поймем откуда происходит вызов метода `ApplicationThread.scheduleTransaction`, и вуаля, этим занимается класс:

```
public class ClientTransaction implements Parcelable,
    ObjectPoolItem {

    private IApplicationThread mClient;

    public void schedule() throws RemoteException {
        mClient.scheduleTransaction(this);
    }
}
```

Он вызывает у `ApplicationThread.scheduleTransaction` передавая себя, тем самым запланируя себя и свои внутренние транзакций на выполнение, `IApplicationThread` это и есть класс `ActivityThread.ApplicationThread`, далее отследим вызов метода `ClientTransaction.schedule()`, встречайте еще один класс,

```
class ClientLifecycleManager {

    void scheduleTransactionItems(@NonNull IApplicationThread
    client,

                                boolean
    shouldDispatchImmediately,

                                @NonNull
    ClientTransactionItem... items) throws RemoteException {
```

```

        ...
        final ClientTransaction clientTransaction =
getOrCreatePendingTransaction(client);

        final int size = items.length;
        for (int i = 0; i < size; i++) {
            clientTransaction.addTransactionItem(items[i]);
        }

        onClientTransactionItemScheduled(clientTransaction,
shouldDispatchImmediately);
    }

    private void onClientTransactionItemScheduled(
        @NonNull ClientTransaction clientTransaction,
        boolean shouldDispatchImmediately) throws
RemoteException {
        ...
        scheduleTransaction(clientTransaction);
    }

    void scheduleTransaction(@NonNull ClientTransaction
transaction) throws RemoteException {
        ...
        transaction.schedule();
        ...
    }
}

```

Внутри него определён метод `scheduleTransactionItems`, который принимает `ApplicationThread` и массив `ClientTransactionItem`. Этот метод создаёт или достаёт транзакцию через `getOrCreatePendingTransaction`, добавляет в неё все `ClientTransactionItem` (например, `LaunchActivityItem`, `ResumeActivityItem`, `PauseActivityItem` и т.д.), после чего передаёт её в метод `onClientTransactionItemScheduled`, где вызывается `scheduleTransaction`.

После чего управление переходит в метод `scheduleTransaction`, внутри которого вызывается `transaction.schedule()`. А как мы уже знаем, метод `schedule` вызывает `ApplicationThread.scheduleTransaction`, то есть фактически мы возвращаемся обратно к AIDL-вызову, из которого всё и начинается.

Таким образом, `ClientLifecycleManager` собирает транзакцию, наполняет её нужными `ClientTransactionItem`, и отправляет её в исполнение. Это класс, который формирует цепочку действий, и делегирует выполнение низкоуровневому слою через AIDL.

`ClientLifecycleManager.scheduleTransactionItems` - вызовом метода занимается очень важный класс `ActivityTaskSupervisor`

```
public class ActivityTaskSupervisor implements
RecentTasks.Callbacks {
    ...
    final ActivityTaskManagerService mService;
    ...

    boolean realStartActivityLocked(ActivityRecord r,
WindowProcessController proc,
                                boolean andResume, boolean
checkConfig) throws RemoteException {

        // Create activity launch transaction.
        final LaunchActivityItem launchActivityItem = new
LaunchActivityItem(r.token,
                    ...,r.getSavedState(),
r.getPersistentSavedState(), ...,
                );
        ...
        mService.getLifecycleManager().scheduleTransactionItems(
            proc.getThread(),
            // Immediately dispatch the transaction, so that
if it fails, the server can
            // restart the process and retry now.
            true /* shouldDispatchImmediately */,
```

```

        launchActivityResult, lifecycleItem);
        ...
        return true;
    }
    ...
}

```

Видим очень ключевые моменты:

1. В методе `realStartActivityLocked` на вход передается объект класса `ActivityRecord`, который в себе хранит значения - `r.getSavedState()(Bundle)` и `r.getPersistentSavedState(PersistentBundle)` и прочие важные значения и информацию об активити
2. Наконецто видим создание транзакций `LaunchActivityResult` с передачей всех нужных аргументов, в числе и `Bundle`
3. Видим что у класса `ActivityTaskManagerService` вызывается метод `getLifecycleManager()` который возвращает объект класса `ClientLifecycleManager` и вызывает у него метод `scheduleTransactionItems` который мы уже видели, с передачей `LaunchActivityResult`

Давай убедимся что метод `getLifecycleManager` у `ActivityTaskManagerService` действительно возвращает `ClientLifecycleManager`:

```

public class ActivityTaskManagerService extends
    IActivityTaskManager.Stub {

    ClientLifecycleManager getLifecycleManager() {
        return mLifecycleManager;
    }
}

```

Убедились, прекрасно, идем дальше, отследим вызов метода `realStartActivityLocked` класса `ActivityTaskSupervisor`

```

class RootWindowContainer extends WindowContainer<DisplayContent>
implements DisplayManager.DisplayListener {

```

```

ActivityTaskSupervisor mTaskSupervisor;
ActivityTaskManagerService mService;

boolean attachApplication(WindowProcessController app) throws
RemoteException {
    final ArrayList<ActivityRecord> activities =
mService.mStartingProcessActivities;
    for (int i = activities.size() - 1; i >= 0; i--) {
        final ActivityRecord r = activities.get(i);
        ...
        if (mTaskSupervisor.realStartActivityLocked(r, app,
canResume,
            true /* checkConfig */)) {
            hasActivityStarted = true;
        }
        ...
        return hasActivityStarted;
    }
}
}

```

⚠ RootWindowContainer...

`RootWindowContainer` — это центральный компонент в системе управления окнами Android, который содержит в себе всю иерархию окон на всех дисплеях. Он управляет экземплярами `DisplayContent`, координирует `layout`, `input`, фокус, анимации, транзишены, `split-screen`, `picture-in-picture` и любые изменения, связанные с конфигурацией экрана. Всё, что должно появиться, исчезнуть, пересчитаться или анимироваться — сначала проходит через него. Это точка входа для всех транзакций окон, включая запуск и завершение активностей.

Он настолько крут, что может остановить перезапуск `activity`, если чувствует, что `layout` всё ещё "в пути". Ему не нужно подтверждение от `WindowManagerService` для показа `Window` и работы с контентом.

`RootWindowContainer` раньше назывался `RootActivityContainer`

Видим вызов метода `ActivityTaskSupervisor.realStartActivityLocked` происходит в классе `RootWindowContainer`, который в методе `attachApplication`, получает список `ActivityRecord` у `ActivityTaskManagerService`, и в цикле для всех вызывает метод `ActivityTaskSupervisor.realStartActivityLocked`.

Далее мы снова возвращаемся к `ActivityTaskManagerService`, потому что именно он вызывает метод `attachApplication` у `RootWindowContainer` и передает ему

```
public class ActivityTaskManagerService extends
    IActivityTaskManager.Stub {
    ...

    /** The starting activities which are waiting for their
    processes to attach. */
    final ArrayList<ActivityRecord> mStartingProcessActivities =
    new ArrayList<>();
    RootWindowContainer mRootWindowContainer;

    @HotPath(caller = HotPath.PROCESS_CHANGE)
    @Override
    public boolean attachApplication(WindowProcessController wpc)
    throws RemoteException {
```

```

        ...
        return mRootWindowContainer.attachApplication(wpc);
    }

    void startProcessAsync(ActivityRecord activity, boolean
knownToBeDead, boolean isTop,
                        String hostingType) {
        ...
        mStartingProcessActivities.add(activity);
        ...
    }

    ClientLifecycleManager getLifecycleManager() {
        return mLifecycleManager;
    }
    ...
}

```

Видим, что он хранит в себе список `ActivityRecord` в поле `mStartingProcessActivities` — вызов которого мы уже видели в `RootWindowContainer.attachApplication`.

Далее видим, что у него также есть ссылка на `RootWindowContainer`, и в методе `ActivityTaskManagerService.attachApplication` происходит вызов метода `RootWindowContainer.attachApplication`. `startProcessAsync` — также очень важный метод, который добавляет новые `ActivityRecord` в список `mStartingProcessActivities`, внутри которых хранится `Bundle` (его мы разберём позже).

Выше `ActivityTaskManagerService` находится класс `ActivityManagerService`, он и вызывает `attachApplication` у `ActivityTaskManagerService`:

```

public class ActivityManagerService extends IActivityManager.Stub
{

    public ActivityTaskManagerInternal mAtmInternal;
    final PidMap mPidsSelfLocked = new PidMap();

    @GuardedBy("this")

```

```

        private void attachApplicationLocked(@NonNull
IApplicationThread thread,
                                           int pid, int callingUid,
long startSeq) {
    ...
    finishAttachApplicationInner(startSeq, callingUid, pid);
    ...
}

    private void finishAttachApplicationInner(long startSeq, int
uid, int pid) {
        ...
        final ProcessRecord app;
        app = mPidsSelfLocked.get(pid);
        ...

        didSomething =
mAtmInternal.attachApplication(app.getWindowProcessController());
        ...
    }
}

```

Видим в методе `finishAttachApplicationInner` вызов метода `attachApplication` у `mAtmInternal`, `ActivityTaskManagerInternal`, который является абстрактным AIDL-интерфейсом для `ActivityTaskManagerService`, поэтому фактически здесь вызывается `ActivityTaskManagerService.attachApplication()`.

Сам метод `finishAttachApplicationInner` вызывается из `attachApplicationLocked`, где процесс извлекается из `mPidsSelfLocked` по ключу `pid` (то есть process id).

Сам `ActivityManagerService` является синглтоном в рамках всей системы Android, у него внутри есть структура `PidMap`, которая хранит объекты `ProcessRecord` по ключу `pid`. То есть вызов `mPidsSelfLocked.get(pid)` обращается к `PidMap`:

```

public class ActivityManagerService extends IActivityManager.Stub
{

    final PidMap mPidsSelfLocked = new PidMap();

```

```

...

static final class PidMap {
    private final SparseArray<ProcessRecord> mPidMap = new
SparseArray<>();

    ProcessRecord get(int pid) {
        return mPidMap.get(pid);
    }
    ...

    void doAddInternal(int pid, ProcessRecord app) {
        mPidMap.put(pid, app);
    }
    ...
}

public void setSystemProcess() {
    ...
    ProcessRecord app =
mProcessList.newProcessRecordLocked(info, info.processName,
        false,
        0,
        false,
        0,
        null,
        new
HostingRecord(HostingRecord.HOSTING_TYPE_SYSTEM));
    ...
    addPidLocked(app);
    ...
}

void addPidLocked(ProcessRecord app) {
    final int pid = app.getPid();
    synchronized (mPidsSelfLocked) {
        mPidsSelfLocked.doAddInternal(pid, app);
    }
}

```

```

        }
        ...
    }
}

```

Видим структуру `PidMap`, которая внутри себя хранит список записей процессов приложения.

Также видим методы `setSystemProcess` и `addPidLocked`. В `setSystemProcess` создаётся новый `ProcessRecord` и вызывается метод `addPidLocked`, который кладёт его в `mPidsSelfLocked`. Метод `setSystemProcess` вызывается из `SystemService` (он же `system_service`). Ниже краткий стек вызовов:

```

1. Загрузчик (Bootloader) → Ядро (Linux Kernel)
2. Процесс init (первый userspace-процесс)
   └─ Запуск zygote (через app_process)
      └─ ZygoteInit (singleton, подготавливает среду для Java-
процессов)
         └─ fork() → создаёт SystemServer
            └─ fork() → создаёт приложения
               └─ SystemServer (singleton, запускает все системные сервисы)
                  └─ RuntimeInit (инициализирует среду для SystemServer)
                     └─ ActivityManagerService (singleton, включая
`setSystemProcess()`)

```

Выше `ActivityManagerService` подниматься нет смысла, так как там `Bundle` не хранится, большинство этих компонентов — это синглтоны всей системы и не имеют прямого отношения к конкретному приложению.

На этом моменте уже многое стало ясно: мы рассмотрели очень длинный flow вызовов. Момент, который мы немного пропустили, — где именно создаются `ActivityRecord`. Ранее мы уже видели список `ActivityRecord`, получаемый из поля `mStartingProcessActivities` у `ActivityTaskManagerService`:

```

class RootWindowContainer extends WindowContainer<DisplayContent>
implements DisplayManager.DisplayListener {

    ActivityTaskSupervisor mTaskSupervisor;

```



```

    ActivityTaskManagerService mService;

    boolean attachApplication(WindowProcessController app) throws
    RemoteException {
        final ArrayList<ActivityRecord> activities =
mService.mStartingProcessActivities;
        for (int i = activities.size() - 1; i >= 0; i--) {
            final ActivityRecord r = activities.get(i);
            ...
            if (mTaskSupervisor.realStartActivityLocked(r, app,
canResume,
                true /* checkConfig */) {
                hasActivityStarted = true;
            }
            ...
        }
    }
}

```

В ActivityTaskManagerService это выглядит следующим образом. Как мы уже видели, поле mStartingProcessActivities является коллекцией, которая хранит объекты ActivityRecord. Есть один метод, который добавляет ActivityRecord в эту коллекцию — это метод startProcessAsync:

```

public class ActivityTaskManagerService extends
IActivityTaskManager.Stub {
    ...

    /** The starting activities which are waiting for their
processes to attach. */
    final ArrayList<ActivityRecord> mStartingProcessActivities =
new ArrayList<>();
    RootWindowContainer mRootWindowContainer;

    void startProcessAsync(ActivityRecord activity, boolean
knownToBeDead, boolean isTop,
                        String hostingType) {

```

```

        ...
        mStartingProcessActivities.add(activity);
        ...
    }
    ...
}

```

Следующая глава статьи будет раскрывать этот момент, где создается ActivityRecord и кто его кладет в ActivityTaskManagerService в поле mStartingProcessActivities

Пересоздание процесса с сохранением Bundle

```

ActivityManagerService.startActivity()
    → ActivityTaskManagerService.startActivityAsUser()
    → ActivityStartController.obtainStarter()
    → ActivityStarter.execute()
        → executeRequest():
            1. Создание ActivityRecord (новый объект)
            2. startActivityUnchecked()
                → startActivityInner()
                    → setInitialState(r) // сохраняем ActivityRecord в
mStartActivity
                →
RootWindowContainer.resumeFocusedTasksTopActivities(mStartActivity
)
                → Task.resumeTopActivityUncheckedLocked()
                →
ActivityTaskSupervisor.startSpecificActivity(r)
                → (если процесс не запущен)
                →
ActivityTaskManagerService.startProcessAsync(r)
                → mStartingProcessActivities.add(r) //
финальная точка

```

ActivityRecord (с Bundle) умеет переживать смерть процесса или его прерывание. Подразумевается ситуация, когда приложение уходит в фон и сохраняется в стеке

задач (Recents), система через какое-то время убивает процесс. Когда пользователь возвращается, система вызывает метод `startActivityFromRecents`, чтобы восстановить задачу (Task) и поднять процесс. Каждая задача, как правило, соответствует одной корневой Activity, но внутри может хранить дочерние Activity, которые тоже связаны с компонентами.

```
public class ActivityManagerService extends IActivityManager.Stub
{

    @Override
    public final int startActivityFromRecents(int taskId, Bundle
bOptions) {
        return
mActivityTaskManager.startActivityFromRecents(taskId, bOptions);
    }

}
```

Метод `startActivityFromRecents` внутри `ActivityManagerService` напрямую делегирует вызов в `ActivityTaskManagerService`. Сам по себе он ничего не делает, просто перекидывает управление дальше.

```
public class ActivityTaskManagerService extends
IActivityTaskManager.Stub {

    ActivityTaskSupervisor mTaskSupervisor;

    @Override
    public final int startActivityFromRecents(int taskId, Bundle
bOptions) {
        ...
        return
mTaskSupervisor.startActivityFromRecents(callingPid, callingUid,
taskId, safeOptions);
    }
}
```

В `ActivityTaskManagerService.startActivityFromRecents` происходит подготовка: извлекаются PID, UID, формируются безопасные опции запуска (`SafeActivityOptions`). Далее метод сразу передаёт выполнение в `ActivityTaskSupervisor`, где происходит основная логика обработки задачи.

```
public class ActivityTaskSupervisor implements
RecentTasks.Callbacks {

    final ActivityTaskManagerService mService;
    RootWindowContainer mRootWindowContainer;

    int startActivityFromRecents(int callingPid, int callingUid,
int taskId,
                                SafeActivityOptions options) {

        final Task task;

        task = mRootWindowContainer.anyTaskForId(taskId,
MATCH_ATTACHED_TASK_OR_RECENT_TASKS_AND_RESTORE, activityOptions,
ON_TOP);

        if
(!mService.mAmInternal.shouldConfirmCredentials(task.mUserId) &&
task.getRootActivity() != null) {
            final ActivityRecord targetActivity =
task.getTopNonFinishingActivity();
            ...
            mService.moveTaskToFrontLocked(...);
            ...
            return ActivityManager.START_TASK_TO_FRONT;
        }
    }
}
```

Внутри `startActivityFromRecents` у `ActivityTaskSupervisor` происходит уже настоящий разбор: сначала ищется нужная задача через `mRootWindowContainer.anyTaskForId(...)`, где передаются различные флаги

(например, `MATCH_ATTACHED_TASK_OR_RECENT_TASKS_AND_RESTORE`), чтобы восстановить задачу из списка недавних. Затем проверяется, нужно ли подтверждать учётные данные пользователя (например, если включён режим защиты профиля). После этого смотрится, есть ли у задачи root Activity (`getRootActivity()`), и извлекается верхняя невыполненная Activity через `getTopNonFinishingActivity()`.

Если все условия подходят, вызывается `moveTaskToFrontLocked(...)` у `ActivityTaskManagerService`, который отвечает за перенос задачи в передний план и дальнейший запуск. Всё это нужно для того, чтобы корректно восстановить состояние приложения из стека задач без необходимости полного пересоздания Activity с нуля.

```
public class ActivityTaskManagerService extends
    IActivityTaskManager.Stub {

    void moveTaskToFrontLocked(@Nullable IApplicationThread
appThread,

                                @Nullable String callingPackage,
int taskId, ...) {

        final Task task =
mRootWindowContainer.anyTaskForId(taskId);
        ...
        mTaskSupervisor.findTaskToMoveToFront(task, flags, ...);
    }

}
```

Метод `moveTaskToFrontLocked` после проверки передаёт управление в `findTaskToMoveToFront`. Здесь задача не просто находится, а действительно перемещается на передний план. В начале вытаскивается корневой контейнер задачи через `getRootTask()`. Если задача ещё не была «переподвешена» (reparented), вызывается `moveHomeRootTaskToFrontIfNeeded`, чтобы при необходимости поднять домашнюю задачу (например, если приложение долго не запускалось).

Далее через `getTopNonFinishingActivity()` достаётся верхняя невыполненная `ActivityRecord(Activity)` в задаче. Затем вызывается `currentRootTask.moveTaskToFront`, куда передаётся сама задача, опции анимации и другие параметры

```
public class ActivityTaskSupervisor implements
RecentTasks.Callbacks {

    void findTaskToMoveToFront(Task task, int flags,
ActivityOptions options, String reason,
                                boolean forceNonResizable) {
        Task currentRootTask = task.getRootTask();

        if (!reparented) {
            moveHomeRootTaskToFrontIfNeeded(flags,
currentRootTask.getDisplayArea(), reason);
        }

        final ActivityRecord r =
task.getTopNonFinishingActivity();
        currentRootTask.moveTaskToFront(task, false /* noAnimation
*/, options,
            r == null ? null : r.appTimeTracker, reason);
        ...
    }
}
```

В методе `moveTaskToFront` внутри класса `Task` мы видим финальный шаг — вызов `mRootWindowContainer.resumeFocusedTasksTopActivities()`. Этот вызов отвечает за то, чтобы на уровне контейнера окон (`WindowContainer`) запустить или возобновить верхнюю активность, сделать её активной и отрисовать.

```
class Task extends TaskFragment {

    final void moveTaskToFront(Task tr, boolean noAnimation,
ActivityOptions options,
```

```

AppTimeTracker timeTracker, boolean
deferResume, String reason) {
    ...
    mRootWindowContainer.resumeFocusedTasksTopActivities();
}
}

```

Метод `resumeFocusedTasksTopActivities` у `RootWindowContainer` проходит по всем дисплеям, чтобы определить, какая задача должна быть запущена или возобновлена. Для каждого дисплея вызывается `forAllRootTasks`, внутри которого берётся верхняя активность (`topRunningActivity`). Если она уже в состоянии `RESUMED`, то просто выполняется переход приложения (`executeAppTransition`). В противном случае активность активируется через `makeActivelfNeeded`.

Если на дисплее не оказалось ни одной подходящей активности, вызывается `resumeTopActivityUncheckedLocked` у фокусной задачи. А если вообще нет фокусных задач, система запускает домашнюю Activity через `resumeHomeActivity`.

```

class RootWindowContainer extends WindowContainer<DisplayContent>
    implements DisplayManager.DisplayListener {

    boolean resumeFocusedTasksTopActivities(
        Task targetRootTask, ActivityRecord target,
        ActivityOptions targetOptions,
        boolean deferPause) {

        for (int displayNdx = getChildCount() - 1; displayNdx >=
0; --displayNdx) {
            final DisplayContent display = getChildAt(displayNdx);
            final boolean curResult = result;
            boolean[] resumedOnDisplay = new boolean[1];
            final ActivityRecord topOfDisplay =
display.topRunningActivity();
            display.forAllRootTasks(rootTask -> {
                final ActivityRecord topRunningActivity =
rootTask.topRunningActivity();
                if (!rootTask.isFocusableAndVisible() ||

```

```

topRunningActivity == null) {
    return;
}
if (rootTask == targetRootTask) {
    resumedOnDisplay[0] != curResult;
    return;
}
if (topRunningActivity.isState(RESUMED) &&
topRunningActivity == topOfDisplay) {
    rootTask.executeAppTransition(targetOptions);
} else {
    resumedOnDisplay[0] !=
topRunningActivity.makeActiveIfNeeded(target);
}
});
result != resumedOnDisplay[0];
if (!resumedOnDisplay[0]) {

    final Task focusedRoot =
display.getFocusedRootTask();
    if (focusedRoot != null) {
        result !=
focusedRoot.resumeTopActivityUncheckedLocked(
            target, targetOptions, false /*
skipPause */);
    } else if (targetRootTask == null) {
        result != resumeHomeActivity(null /* prev */,
"no-focusable-task",
            display.getDefaultTaskDisplayArea());
    }
}
}

return result;
}
}

```


Таким образом, когда пользователь возвращается к приложению из Recents, система шаг за шагом поднимает задачу из стека, подготавливает корневую Activity и доводит её до состояния RESUMED. Всё это происходит последовательно: от поиска задачи в стеке — до финального вызова `makeActivelfNeeded`, который, по сути, завершает процесс восстановления.

После того как контейнер окон выбрал задачу для возобновления, управление переходит в метод `resumeTopActivityUncheckedLocked` внутри класса `Task`. Здесь вызывается внутренний метод `resumeTopActivityInnerLocked`, который уже окончательно определяет, какую Activity нужно запустить.

```
class Task extends TaskFragment {

    @GuardedBy("mService")
    boolean resumeTopActivityUncheckedLocked(ActivityRecord prev,
        ActivityOptions options,
                                           boolean deferPause) {
        someActivityResumed = resumeTopActivityInnerLocked(prev,
            options, deferPause);
    }

    @GuardedBy("mService")
    private boolean resumeTopActivityInnerLocked(ActivityRecord
        prev, ActivityOptions options,
                                           boolean
        deferPause) {
        final TaskFragment topFragment =
            topActivity.getTaskFragment();
        resumed[0] = topFragment.resumeTopActivity(prev, options,
            deferPause);
    }

}
```

В методе `resumeTopActivityInnerLocked` вытаскивается фрагмент задачи (`TaskFragment`), к которому привязана верхняя Activity. Именно тут начинается конкретная подготовка к запуску компонента приложения.

Дальше вызывается `resumeTopActivity` у `TaskFragment`. Здесь происходит поиск верхней активности (`topRunningActivity`) и запуск метода `startSpecificActivity`. По сути, `startSpecificActivity` — это последняя точка внутри ядра системы, где принимается решение: запустить новый процесс для активности или использовать уже существующий.

```
class TaskFragment extends WindowContainer<WindowContainer> {  
  
    final boolean resumeTopActivity(ActivityRecord prev,  
    ActivityOptions options,  
                                   boolean skipPause) {  
        ActivityRecord next = topRunningActivity(true /*  
focusableOnly */);  
        mTaskSupervisor.startSpecificActivity(next, true, false);  
        ...  
        return true;  
        ...  
    }  
  
}
```

Далее метод `startSpecificActivity` внутри `ActivityTaskSupervisor`. Здесь анализируется состояние процесса: если процесс уже существует и привязан, то активности будет сразу запущена. Если же процесс отсутствует или был завершён системой, тогда вызывается `startProcessAsync`, чтобы создать новый процесс для этой активности.

```
public class ActivityTaskSupervisor implements  
RecentTasks.Callbacks {  
    ...  
    final ActivityTaskManagerService mService;  
  
    void startSpecificActivity(ActivityRecord r, boolean  
andResume, boolean checkConfig) {  
        ...  
        mService.startProcessAsync(r, knownToBeDead, isTop,  
isTop ? HostingRecord.HOSTING_TYPE_TOP_ACTIVITY  
: HostingRecord.HOSTING_TYPE_ACTIVITY);  
    }  
}
```

```

    }
}

```

В методе `startProcessAsync` активности добавляются в список `mStartingProcessActivities`. Это своего рода «очередь на запуск», куда система кладёт активности, пока ожидает, что процесс для них будет создан и привязан.

```

public class ActivityTaskManagerService extends
    IActivityTaskManager.Stub {
    ...

    final ArrayList<ActivityRecord> mStartingProcessActivities =
new ArrayList<>();
    RootWindowContainer mRootWindowContainer;

    void startProcessAsync(ActivityRecord activity, boolean
knownToBeDead, boolean isTop,
                        String hostingType) {
        ...
        mStartingProcessActivities.add(activity);
        ...
    }
    ...
}

```

Таким образом, когда мы доходим до финальной стадии, встаёт важный вопрос: **где в конечном итоге хранится `ActivityRecord` и как устроены связи между ключевыми сущностями — `DisplayContent`, `WindowContainer`, `Task` (и `TaskFragment`)?** Это поможет окончательно понять, как именно система управляет состоянием и «жизнью» Activity на стороне System Server.

Общая структура иерархии Android управляет активностями и окнами в виде **иерархического дерева контейнеров**, где каждый контейнер реализован через базовый класс `WindowContainer`. Вся структура начинается с корневого контейнера `RootWindowContainer`, внутри которого для каждого физического или виртуального дисплея создается `DisplayContent`.

DisplayContent `DisplayContent` представляет отдельный физический или виртуальный дисплей. Он является прямым потомком `RootWindowContainer` и внутри себя хранит так называемые **DisplayAreas**, в которых сегментируются разные типы окон (например, область приложений, область системных оверлеев и т.д.). Внутри `DisplayContent` находится **TaskDisplayArea**, которая отвечает за размещение пользовательских задач (`Tasks`).

TaskDisplayArea `TaskDisplayArea` — это область дисплея, куда добавляются задачи (`Task`). В большинстве случаев, если нет multi-window или особых режимов, используется один **DefaultTaskDisplayArea**, где и размещаются все задачи приложения. В иерархии путь выглядит так: **DisplayContent** → **TaskDisplayArea** → **Task**.

Task `Task` (по сути, «стек задач») группирует одну или несколько активити, которые пользователь воспринимает как одно приложение в списке Recents. В Android `Task` наследуется от `TaskFragment`, что делает его контейнером, способным содержать дочерние `WindowContainer`. Обычно внутри задачи размещаются именно `ActivityRecord`, каждая из которых представляет конкретную активити. В более сложных случаях, например при split-screen, `Task` может содержать и другие задачи или `TaskFragments`. Однако в стандартном сценарии (одиночный экран без split) задача содержит список `ActivityRecords` напрямую.

Здесь ключевой момент: **Task** является прямым родителем для **ActivityRecord**. Это значит, что все состояния и контекст конкретной `Activity` хранятся внутри её `ActivityRecord`, который в свою очередь всегда находится внутри задачи. Таким образом, при возврате пользователя к приложению через Recents, система восстанавливает задачу, а вместе с ней и все вложенные `ActivityRecords`.

TaskFragment `TaskFragment` — это базовый класс, который используется для создания под-контейнеров внутри задачи. В обычных сценариях мы его напрямую не видим, потому что работаем с `Task`, который уже является расширением `TaskFragment`. В некоторых режимах (например, Activity Embedding) могут создаваться отдельные `TaskFragments`, чтобы разделить экран между несколькими активити. Но если таких сценариев нет, `Task` сам по себе содержит `ActivityRecords`, и дополнительных `TaskFragments` не используется.

ActivityRecord `ActivityRecord` представляет конкретный экземпляр `Activity` в системе. Он наследуется от `WindowToken`, который в свою очередь является дочерним классом `WindowContainer`. Таким образом, `ActivityRecord` — это одновременно и контейнер для окон активити, и токен, который `WindowManager`

использует для управления окнами. Обычно внутри `ActivityRecord` размещается один основной `WindowState` (окно приложения), а также любые дочерние окна (например, диалоги).

Путь в иерархии выглядит так: `RootWindowContainer` → `DisplayContent` → `TaskDisplayArea` → `Task` → `ActivityRecord` → `WindowState`.

Это означает, что `ActivityRecord` **всегда живёт внутри задачи** и никогда не существует сам по себе или в глобальном списке. Именно поэтому при возврате из `Recents` задача сначала поднимается целиком (`Task`), а затем уже внутри неё активируются нужные активности (`ActivityRecord`).

Такое дерево контейнеров позволяет системе Android централизованно управлять всей иерархией окон и задач. Например, при изменении конфигурации или выгрузке процесса, состояние активности остаётся «привязанным» к её `ActivityRecord`, который живёт внутри `Task`. Когда задача возвращается на экран, все объекты дерева последовательно восстанавливаются, и `Activity` получает свои данные обратно через `Bundle`, связанный с её `ActivityRecord`.

Сделаем краткий итог

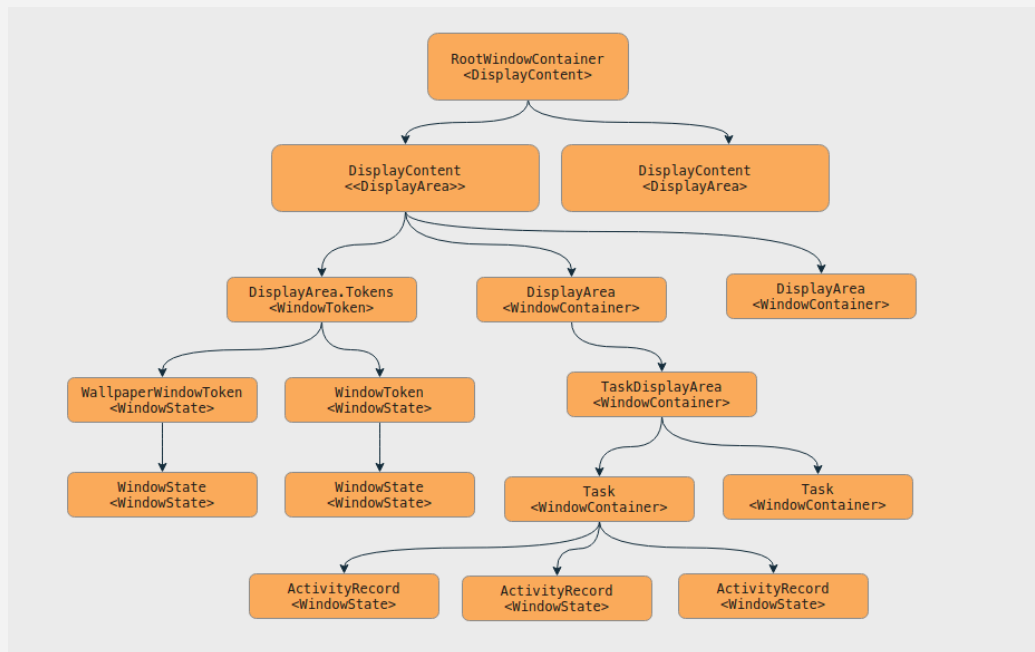
- **DisplayContent** — верхний контейнер для дисплея, включает `TaskDisplayArea`.
- **TaskDisplayArea** — область дисплея для задач.
- **Task** — контейнер, группирующий одну или несколько `ActivityRecords`.
- **TaskFragment** — промежуточный контейнер, используется при `embedding` или `split`, обычно не нужен в базовом сценарии.
- **ActivityRecord** — контейнер и токен конкретной `Activity`, всегда находится внутри `Task`.
- **WindowState** — дочерние окна `Activity`, живут внутри `ActivityRecord`.

Таким образом, вопрос «*где хранится ActivityRecord*» можно чётко ответить: **внутри Task**, как дочерний элемент в дереве контейнеров.

Эта архитектура делает поведение задач предсказуемым и позволяет системе сохранять, приостанавливать и восстанавливать активности, не нарушая общую

структуру приложения в памяти. Именно поэтому пользователь всегда видит «цельную» задачу в Recents, а не отдельные активности.

⚠ Для более наглядного понимания иерархии можно посмотреть диаграмму ниже, которая отлично иллюстрирует дерево контейнеров в Android WindowManager (начиная с Android 12).



Android WindowManager Hierarchy

Диаграмма взята с [sobyte.net](https://www.sobyte.net) — Android 12 WMS Hierarchy

(<https://www.sobyte.net/post/2022-02/android-12-wms-hierarchy/#:~:text=%2A%20RootWindowContainer%3A%20The%20top,%E2%80%A6>) для иллюстрации иерархии WindowManager.

Где и когда создается ActivityRecord в первые

После того как мы разобрали, где именно хранится ActivityRecord в иерархии контейнеров, возникает следующий важный вопрос: **а когда и как этот объект вообще появляется в системе?**

Все предыдущие главы показывали нам, как система управляет уже существующими ActivityRecord — как они восстанавливаются из стека задач (Recents), как переходят между состояниями, как сохраняются их состояния. Но откуда берётся первый экземпляр ActivityRecord, когда Activity запускается

впервые, например, при самом первом запуске приложения или при старте новой Activity через интент?

Именно этот момент — создание `ActivityRecord` — можно считать точкой входа активности в «жизнь» на стороне system server. На этом этапе создаётся основная структура, к которой в дальнейшем будут привязаны всё: и окна (`WindowState`), и состояния (`Bundle`), и привязки к задаче (`Task`).

Дальше система начинает «разворачивать» процесс по цепочке вызовов, начиная с верхнего уровня — `ActivityManagerService`. Когда приложение или другой компонент системы вызывает `startActivity(...)`, эта команда сначала попадает в публичный API `ActivityManagerService`, а уже оттуда прокладывает путь вниз через слои system server, где и подготавливаются все объекты, необходимые для старта.

Вот как выглядит эта цепочка вызовов на первых уровнях:

```
public class ActivityManagerService extends
    IActivityManager.Stub, ... {

    @Override
    public int startActivityWithFeature(IApplicationThread caller,
        String callingPackage, ...) {
        return mActivityTaskManager.startActivity(caller,
            callingPackage, callingFeatureId, intent, ...);
    }

}
```

Здесь `ActivityManagerService` лишь перенаправляет вызов в `ActivityTaskManagerService`, где начинается более детальная работа с профилями пользователей, флагами интентов и прочими проверками.

```
public class ActivityTaskManagerService extends
    IActivityTaskManager.Stub {

    @Override
    public final int startActivity(IApplicationThread caller,
        String callingPackage, ...) {
        return startActivityAsUser(caller, callingPackage,
```

```

callingFeatureId, intent, ...);
    }

    private int startActivityAsUser(IApplicationThread caller,
String callingPackage, ...) {

        return getActivityStartController().obtainStarter(intent,
"startActivityAsUser")
            ...
            .execute();
    }

    ActivityStartController getActivityStartController() {
        return mActivityStartController;
    }
}

```

В методе `startActivityAsUser` мы уже видим обращение к `ActivityStartController`, который управляет процессом создания и конфигурации старта активности. Метод `obtainStarter` возвращает объект `ActivityStarter`, который можно назвать настоящим «дирижёром» запуска. Он собирает все параметры, проверяет, нужна ли новая задача (`Task`) или можно использовать существующую, проверяет конфигурацию и наконец подготавливает `ActivityRecord`.

```

public class ActivityStartController {

    ActivityStarter obtainStarter(Intent intent, String reason) {
        return
mFactory.obtain().setIntent(intent).setReason(reason);
    }
}

```

После того как мы получаем `ActivityStarter` через `obtainStarter`, именно здесь происходит создание нового объекта `ActivityRecord`. `ActivityStarter` формирует все ключевые параметры запуска: интент, флаги, целевой `Task`, конфигурацию окна, а также решает, нужно ли создать новую задачу или использовать существующую.

Созданный `ActivityRecord` связывается с задачей, добавляется в иерархию контейнеров и становится частью общей структуры `RootWindowContainer`. После создания `ActivityRecord` хранится в дереве контейнеров до завершения активности или её удаления системой.

```
class ActivityStarter {

    private final ActivityTaskManagerService mService;
    private final RootWindowContainer mRootWindowContainer;
    ActivityRecord mStartActivity;

    int execute() {
        ...
        res = executeRequest(mRequest);
        ...
    }

    private int executeRequest(Request request) {
        final ActivityRecord r = new
ActivityRecord.Builder(mService)
            ... // параметры через билдер
            .build();

        mLastStartActivityResult = startActivityUnchecked(r, ...);
        ...
    }

    private int startActivityUnchecked(final ActivityRecord r,
...) {
        ...
        result = startActivityInner(r, ...);
        ...
    }

    int startActivityInner(final ActivityRecord r, ...) {
        setInitialState(r, ...);
    }
}
```

```

        mRootWindowContainer.resumeFocusedTasksTopActivities(
            mTargetRootTask, mStartActivity, mOptions,
            mTransientLaunch);
    }

    private void setInitialState(ActivityRecord r, ...) {
        ...
        mStartActivity = r;
        ...
    }
}

```

В методе `executeRequest` через билдер создаётся объект `ActivityRecord`. После инициализации передаётся в `startActivityUnchecked`, а затем в `startActivityInner`, где вызывается метод `setInitialState`. Здесь объект сохраняется в `mStartActivity` — это ссылка на текущую активность, которая будет запущена.

Далее активити подготавливается к запуску через вызов `resumeFocusedTasksTopActivities` у `RootWindowContainer`.

```

class RootWindowContainer extends WindowContainer<DisplayContent>
    implements DisplayManager.DisplayListener {

    boolean resumeFocusedTasksTopActivities(
        Task targetRootTask, ActivityRecord target,
        ActivityOptions targetOptions,
        boolean deferPause) {

        for (int displayNdx = getChildCount() - 1; displayNdx >=
0; --displayNdx) {
            final DisplayContent display = getChildAt(displayNdx);
            final boolean curResult = result;
            boolean[] resumedOnDisplay = new boolean[1];
            final ActivityRecord topOfDisplay =
display.topRunningActivity();
            display.forAllRootTasks(rootTask -> {
                final ActivityRecord topRunningActivity =
rootTask.topRunningActivity();

```

```

        if (!rootTask.isFocusableAndVisible() ||
topRunningActivity == null) {
            return;
        }
        if (rootTask == targetRootTask) {
            resumedOnDisplay[0] != curResult;
            return;
        }
        if (topRunningActivity.isState(RESUMED) &&
topRunningActivity == topOfDisplay) {
            rootTask.executeAppTransition(targetOptions);
        } else {
            resumedOnDisplay[0] !=
topRunningActivity.makeActiveIfNeeded(target);
        }
    });
    result != resumedOnDisplay[0];
    if (!resumedOnDisplay[0]) {
        final Task focusedRoot =
display.getFocusedRootTask();
        if (focusedRoot != null) {
            result !=
focusedRoot.resumeTopActivityUncheckedLocked(
                target, targetOptions, false /*
skipPause */);
        } else if (targetRootTask == null) {
            result != resumeHomeActivity(null /* prev */,
"no-focusable-task",
                display.getDefaultTaskDisplayArea());
        }
    }
}

return result;
}
}

```

В методе `resumeFocusedTasksTopActivities` происходит обход всех дисплеев и корневых задач. Для каждой задачи выбирается верхняя активити, проверяется её состояние и возможность активации. Если задача содержит целевую активити (`target`), она активируется вызовом `resumeTopActivityUncheckedLocked`.

Таким образом, после создания `ActivityRecord`, система полностью подготавливает задачу и активирует верхнюю активити, переводя её в состояние `RESUMED`.

Отлично, продолжим ровно в том же техническом, «ровном» стиле, учитывая, что эти методы мы действительно уже подробно разбирали ранее.

После того как контейнер окон выбрал задачу для возобновления, управление переходит в метод `resumeTopActivityUncheckedLocked` внутри класса `Task`. Мы уже встречали этот метод раньше — он отвечает за выбор и финальную подготовку верхней активити внутри задачи перед запуском. Внутри него вызывается `resumeTopActivityInnerLocked`, который в свою очередь извлекает нужный `TaskFragment`.

```
class Task extends TaskFragment {

    @GuardedBy("mService")
    boolean resumeTopActivityUncheckedLocked(ActivityRecord prev,
        ActivityOptions options,
                                           boolean deferPause) {
        someActivityResumed = resumeTopActivityInnerLocked(prev,
options, deferPause);
    }

    @GuardedBy("mService")
    private boolean resumeTopActivityInnerLocked(ActivityRecord
prev, ActivityOptions options,
                                           boolean
deferPause) {
        final TaskFragment topFragment =
topActivity.getTaskFragment();
        resumed[0] = topFragment.resumeTopActivity(prev, options,
deferPause);
    }
}
```

```
}
```

Как мы помним, в методе `resumeTopActivityInnerLocked` вытаскивается верхний фрагмент задачи (объект `TaskFragment`), который содержит активити, готовую к запуску.

Далее вызывается `resumeTopActivity` у `TaskFragment`. Этот метод ищет верхнюю активити в контейнере (`topRunningActivity`) и инициирует вызов `startSpecificActivity`. Здесь принимается решение, нужно ли запускать новый процесс или использовать уже существующий.

```
class TaskFragment extends WindowContainer<WindowContainer> {  
  
    final boolean resumeTopActivity(ActivityRecord prev,  
    ActivityOptions options,  
                                   boolean skipPause) {  
        ActivityRecord next = topRunningActivity(true /*  
focusableOnly */);  
        mTaskSupervisor.startSpecificActivity(next, true, false);  
        ...  
        return true;  
        ...  
    }  
  
}
```

Мы уже видели метод `startSpecificActivity` внутри `ActivityTaskSupervisor` в предыдущих главах. Он проверяет, существует ли уже процесс для текущей активности. Если процесс жив и активити привязана, то система продолжает её запуск напрямую. Если процесс отсутствует или был выгружен системой, вызывается метод `startProcessAsync`, который отвечает за асинхронный старт нового процесса.

```
public class ActivityTaskSupervisor implements  
RecentTasks.Callbacks {  
    ...  
    final ActivityTaskManagerService mService;
```

```

    void startSpecificActivity(ActivityRecord r, boolean
andResume, boolean checkConfig) {
        ...
        mService.startProcessAsync(r, knownToBeDead, isTop,
            isTop ? HostingRecord.HOSTING_TYPE_TOP_ACTIVITY
                : HostingRecord.HOSTING_TYPE_ACTIVITY);
    }
}

```

Внутри `startProcessAsync`, как мы уже подробно разбирали, активити добавляется в список `mStartingProcessActivities`. Это очередь для тех активити, которые ждут, пока процесс будет создан и привязан системой. Такая очередь позволяет системе контролировать порядок запуска и управлять ресурсами без потерь состояний.

```

public class ActivityTaskManagerService extends
IActivityTaskManager.Stub {
    ...

    final ArrayList<ActivityRecord> mStartingProcessActivities =
new ArrayList<>();
    RootWindowContainer mRootWindowContainer;

    void startProcessAsync(ActivityRecord activity, boolean
knownToBeDead, boolean isTop,
                        String hostingType) {
        ...
        mStartingProcessActivities.add(activity);
        ...
    }
    ...
}

```

Таким образом, вся эта цепочка методов, которые мы уже встречали ранее, замыкается именно здесь: от вызова из контейнеров окон до финального решения о создании нового процесса или продолжении в текущем. В результате создаётся, сохраняется и активируется `ActivityRecord`, и именно он становится ключевым

звеном между системой и пользовательским интерфейсом. Что происходит после вызова этого метода и последующую логику обработки мы уже подробно разбирали в предыдущих главах.

На этом, пожалуй, всё — это конец статьи.

Decompose и Essenty: под капотом сохранения состояния без ViewModel

Введение

Это продолжение четырех предыдущих статей.

1. В первой мы разобрали, где в конечном итоге хранится `ViewModelStore` в случае с `Activity`.
2. Во второй — как это устроено во `Fragment`.
3. В третьей — где хранятся `ViewModel`-и, когда мы используем `Compose` (или даже просто `View`).
4. В четвёртой — как работают методы `onSaveInstanceState`/`onRestoreInstanceState`, `Saved State API` и где хранится `Bundle`.

В этой статье разберёмся, как широко используемая в KMP библиотека **Decompose** справляется без `ViewModel` и методов `onSaveInstanceState`, ведь она является кроссплатформенной (KMP) библиотекой.

Статья не о том, *как* использовать эти API, а о том, *как* они работают изнутри. Поэтому я буду полагаться на то, что вы уже знакомы с ними или хотя бы имеете общее представление.

Как всегда, начнём с базиса. Давайте сначала дадим определение `Decompose`:

Базис

Decompose — это мультиплатформенная библиотека для разделения бизнес-логики и UI, разработанная Аркадием Ивановым. Она работает поверх `ComponentContext`, который управляет жизненным циклом, состоянием и навигацией между компонентами.

Поддерживает: Android, iOS, JS, JVM, macOS, watchOS, tvOS.

Зачем использовать:

- логика отделена от UI и легко тестируется
- работает с Compose, SwiftUI, React и др.
- навигация и состояние — кроссплатформенные
- компоненты переживают конфигурационные изменения (как `ViewModel`)
- можно расширять и кастомизировать `ComponentContext` под свои задачи

Decompose — это не фреймворк, а мощный инструмент, на котором можно построить свой API. Кратко говоря, это швейцарский нож.

В Android сложно представить приложение без стандартной `ViewModel`, и удивительно, что в **Decompose** её нет, но при этом она умеет сохранять данные как при изменении конфигурации, так и при уничтожении процесса.

Давайте быстро разберёмся с сущностями, на которых основана Decompose:

Всё в **Decompose** крутится вокруг `ComponentContext` — компонента, связанного с определённым экраном или набором дочерних компонентов. У каждого компонента есть свой `ComponentContext`, который реализует следующие интерфейсы:

- **LifecycleOwner** — предоставляется библиотекой **Essenty**, даёт каждому компоненту собственный жизненный цикл.
- **StateKeeperOwner** — позволяет сохранять любое состояние при конфигурационных изменениях и/или смерти процесса.
- **InstanceKeeperOwner** — даёт возможность сохранять любые объекты внутри компонента (аналог `ViewModel` в AndroidX).
- **BackHandlerOwner** — позволяет каждому компоненту обрабатывать нажатие кнопки «назад».

Основное внимание мы уделим именно `StateKeeperOwner` (`StateKeeper`) и `InstanceKeeperOwner` (`InstanceKeeper`). Как видно, они на самом деле тянутся из

библиотеки **Essenty**, которая также была создана Аркадием Ивановым. Однако особую популярность эта библиотека получила именно благодаря **Decompose**.

Начнём углубляться в работу `StateKeeperOwner` (`StateKeeper`). Я буду полагаться на то, что вы уже читали предыдущие статьи. Давайте начнём.

StateKeeperOwner

Чтобы понять, как он работает, давайте реализуем простой экран `Counter`. Цель — увидеть, как счётчик умеет переживать изменение конфигурации и даже смерть процесса.

Начнём с создания компонента для счетчика:

```
class DefaultCounterComponent(
    componentContext: ComponentContext
) : ComponentContext by componentContext {

    val model: StateFlow<Int> field =
MutableStateFlow(stateKeeper.consume(KEY, Int.serializer()) ?: 0)

    init {
        stateKeeper.register(KEY, Int.serializer()) { model.value
    }

    fun increase() {
        model.value++
    }

    fun decrease() {
        model.value--
    }

    companion object {
        private const val KEY = "counter_state"
    }
}
```

Довольно простая логика: у нас есть `model`, который хранит текущее значение счётчика, и два метода для его изменения. При инициализации переменной мы получаем значение из `stateKeeper` через `consume`, если оно отсутствует — используем `0` по умолчанию.

А в `init` блоке мы регистрируем лямбду, которая будет вызвана при сохранении состояния. Пока просто запомните этот момент — позже разберёмся, как и когда она срабатывает.

Теперь экран счетчика, который работает с `DefaultCounterComponent`:

```
@Composable
fun CounterScreen(component: DefaultCounterComponent) {
    val count by component.model.collectAsState()

    Column(
        modifier = Modifier.fillMaxSize(),
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.Center
    ) {
        Text(text = count.toString(), style =
MaterialTheme.typography.headlineLarge)
        Row(horizontalArrangement = Arrangement.spacedBy(40.dp)) {
            FloatingActionButton(onClick = { component.decrease()
        }) { Text("-", fontSize = 56.sp) }
            FloatingActionButton(onClick = { component.increase()
        }) { Text("+", fontSize = 56.sp) }
        }
    }
}
```

И, наконец, `Activity`, в которой инициализируется `ComponentContext` и вызывается экран `CounterScreen`:

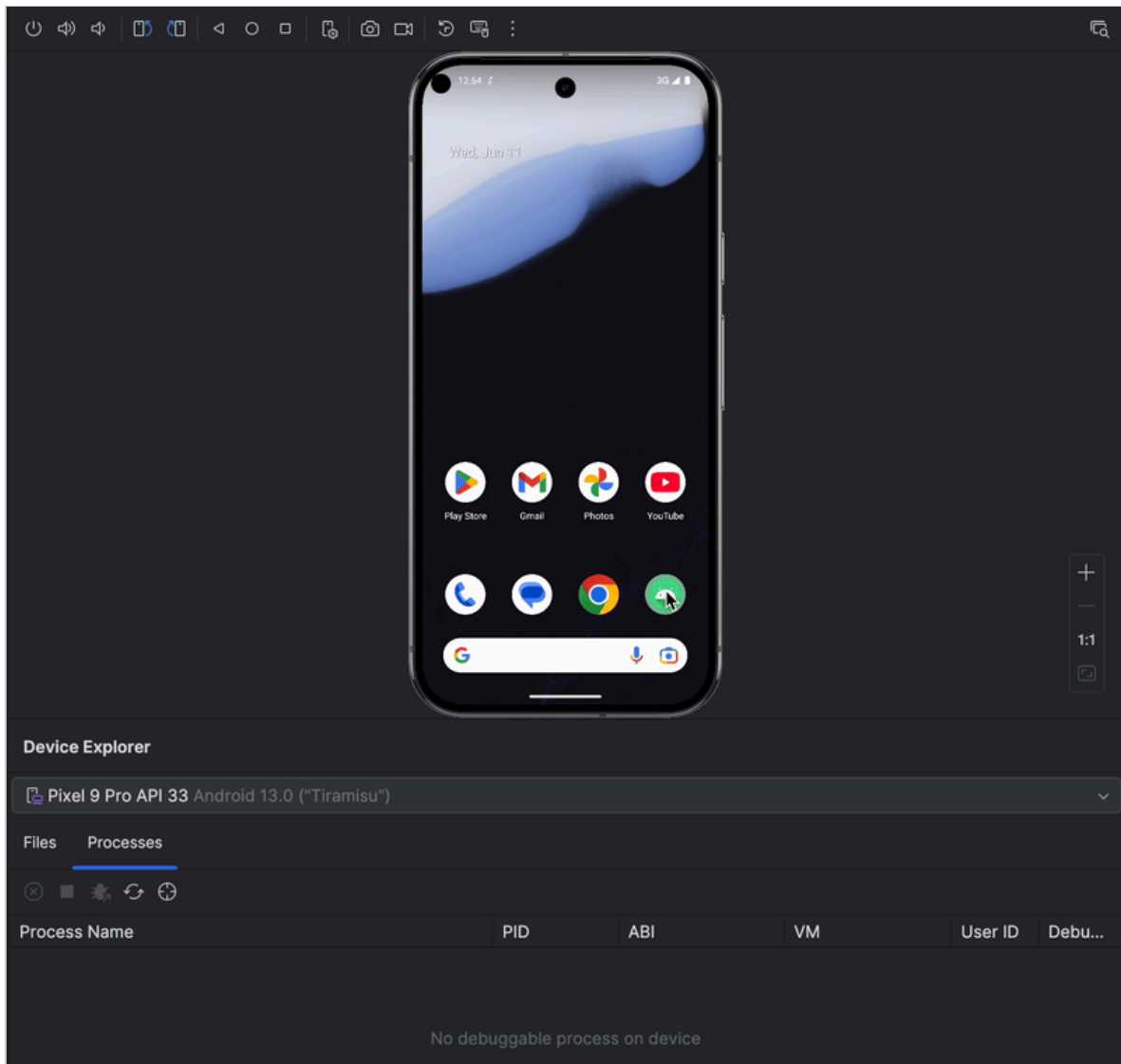
```
class MainActivity : ComponentActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
```

```
        val counterComponent =  
        DefaultCounterComponent(defaultComponentContext())  
        setContent { CounterScreen(component = counterComponent) }  
    }  
}
```

Теперь давайте проверим поведение визуально:

1. Как будет вести себя счётчик при изменении конфигурации (именно повороте экрана).
2. Как будет вести себя счётчик при уничтожении процесса, когда приложение находится в фоне.



Screenshot

Как видим, всё работает ровно так, как ожидалось. Значение счётчика сохраняется как при повороте экрана, так и после полного убийства процесса. При этом мы не видим здесь ни методов `onSaveInstanceState`, ни `ViewModel`. Давайте снова взглянем на компонент счётчика:

```
class DefaultCounterComponent(
    componentContext: ComponentContext
) : ComponentContext by componentContext {

    val model: StateFlow<Int> field =
        MutableStateFlow(stateKeeper.consume(KEY, Int.serializer()) ?: 0)
```

```

    init {
        stateKeeper.register(KEY, Int.serializer()) { model.value
    }
    }
    ...

    companion object {
        private const val KEY = "counter_state"
    }
}

```

При пересоздании активности — как из-за изменения конфигурации, так и после смерти процесса — `DefaultCounterComponent` будет создаваться заново, и вместе с ним создаётся и поле `model`. В таком случае мы обращаемся к `stateKeeper` и, вызывая у него метод `consume`, получаем по ключу сохранённое значение. Если сохранённого значения нет, используем значение по умолчанию — `0`.

В `init`-блоке мы регистрируем коллбэк через метод `stateKeeper.register`, передавая ему ключ, стратегию сериализации из `kotlinx.serialization` и лямбду, возвращающую текущее значение `model`.

Посмотрим на исходники, чтобы понять, откуда берётся поле `stateKeeper`. Наш `DefaultCounterComponent` реализует интерфейс `ComponentContext`, а поле `stateKeeper` приходит из `StateKeeperOwner`. Полная цепочка наследования следующая:

```

interface StateKeeperOwner {

    val stateKeeper: StateKeeper
}

interface GenericComponentContext<out T : Any> :
    LifecycleOwner,
    StateKeeperOwner,
    InstanceKeeperOwner,
    BackHandlerOwner,
    ComponentContextFactoryOwner<T>

```

```
interface ComponentContext :  
    GenericComponentContext<ComponentContext>
```

Таким образом, цепочка наследования выглядит так: `StateKeeperOwner` ← `GenericComponentContext` ← `ComponentContext` ← `DefaultCounterComponent`.

Мы реализуем `ComponentContext`, делегируя его переданному в конструктор параметру `componentContext`.

```
class DefaultCounterComponent(  
    componentContext: ComponentContext  
) : ComponentContext by componentContext {  
    ...  
}
```

А в `MainActivity` создаём `ComponentContext`, используя готовую extension-функцию `defaultComponentContext`, которая за нас уже создаёт `ComponentContext` со всеми нужными компонентами, вроде `StateKeeper`:

```
class MainActivity : ComponentActivity() {  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        ...  
        val counterComponent =  
            DefaultCounterComponent(defaultComponentContext())  
        ...  
    }  
}
```

Продолжаем разбор: цепочка до настоящего хранилища

Итак, мы уже увидели, как в компоненте вызываются `stateKeeper.consume()` и `stateKeeper.register()`, и знаем, что сам компонент получает `stateKeeper` через свой `ComponentContext`. Но что именно происходит между вызовом в `Activity/Fragment`

и конечным хранилищем? Пройдёмся по цепочке, которую мы только что вывели из исходников.

Как создаётся StateKeeper

В Activity (или Fragment) создаётся DefaultComponentContext, и ему передаётся результат вызова defaultComponentContext(). Заглянем внутрь:

```
fun <T> T.defaultComponentContext(
    discardSavedState: Boolean = false,
    isStateSavingAllowed: () -> Boolean = { true },
): DefaultComponentContext where
    T : SavedStateRegistryOwner, T :
    OnBackPressedDispatcherOwner, T : ViewModelStoreOwner, T :
    LifecycleOwner =
    defaultComponentContext(
        backHandler = BackHandler(onBackPressedDispatcher),
        discardSavedState = discardSavedState,
        isStateSavingAllowed = isStateSavingAllowed,
    )
```

Обратите внимание, что функция является расширением для T, где T должен быть объектом, реализующим интерфейсы SavedStateRegistryOwner, OnBackPressedDispatcherOwner, ViewModelStoreOwner, LifecycleOwner. Классы ComponentActivity, FragmentActivity, AppCompatActivity идеально подходят под эти требования.

Внутри по сути просто собираются все нужные зависимости и прокидываются чуть дальше — в ещё одну функцию-обёртку, где уже инициализируется всё, что нужно для хранения состояния:

```
private fun <T> T.defaultComponentContext(
    backHandler: BackHandler?,
    discardSavedState: Boolean,
    isStateSavingAllowed: () -> Boolean,
): DefaultComponentContext where
    T : SavedStateRegistryOwner, T : ViewModelStoreOwner, T :
    LifecycleOwner {
    val stateKeeper = stateKeeper(discardSavedState =
```



```

discardSavedState, isSavingAllowed = isStateSavingAllowed)
    ...
    return DefaultComponentContext(
        lifecycle = lifecycle.asEssentyLifecycle(),
        stateKeeper = stateKeeper,
        instanceKeeper = instanceKeeper(discardRetainedInstances =
marker == null),
        backHandler = backHandler,
    )
}

```

Вот тут и начинается самое интересное — создаётся объект `StateKeeper` вызовом функции `stateKeeper` и пробрасывается дальше.

Как создаётся сам `StateKeeper`

Теперь посмотрим, откуда взялся этот объект. Всё упирается в extension-функцию `stateKeeper`, которая является расширением для `SavedStateRegistryOwner`:

```

private const val KEY_STATE = "STATE_KEEPER_STATE"

fun SavedStateRegistryOwner.stateKeeper(
    discardSavedState: Boolean = false,
    isSavingAllowed: () -> Boolean = { true },
): StateKeeper =
    stateKeeper(
        key = KEY_STATE,
        discardSavedState = discardSavedState,
        isSavingAllowed = isSavingAllowed,
    )

```

Здесь просто прокидывается ключ (по умолчанию `"STATE_KEEPER_STATE"`), и происходит вызов другого метода `stateKeeper`:

```

fun SavedStateRegistryOwner.stateKeeper(
    key: String,
    discardSavedState: Boolean = false,
    isSavingAllowed: () -> Boolean = { true },

```

```

): StateKeeper =
    StateKeeper(
        savedStateRegistry = savedStateRegistry,
        key = key,
        discardSavedState = discardSavedState,
        isSavingAllowed = isSavingAllowed
    )

```

Тут мы уже явно вызываем конструктор `StateKeeper` (на самом деле это функция, а не класс). Сюда подаётся главный объект — `savedStateRegistry`. Да-да, тот самый из `AndroidX`, который находится внутри `Activity` и `Fragment` и используется системой для всех вызовов `onSaveInstanceState`.

Что реально происходит внутри `StateKeeper`

Вот теперь мы приблизились к сути. `StateKeeper` — это функция, которая создаёт реальный объект интерфейса `StateKeeper`:

```

fun StateKeeper(
    savedStateRegistry: SavedStateRegistry,
    key: String,
    discardSavedState: Boolean = false,
    isSavingAllowed: () -> Boolean = { true },
): StateKeeper {
    val dispatcher =
        StateKeeperDispatcher(
            savedState = savedStateRegistry
                .consumeRestoredStateForKey(key = key)
                ?.getSerializableContainer(key = KEY_STATE)
                ?.takeUnless { discardSavedState },
        )

    savedStateRegistry.registerSavedStateProvider(key = key) {
        Bundle().apply {
            if (isSavingAllowed()) {
                putSerializableContainer(key = KEY_STATE, value =
                    dispatcher.save())
            }
        }
    }
}

```

```

        }
    }

    return dispatcher
}

```

Вот он — наш главный гейтвей между миром Android и системой сохранения состояния в Essenty/Decompose. Давайте по строчкам:

- Извлекается ранее сохранённое состояние из `SavedStateRegistry` по ключу — по сути, из стандартного `Bundle`, в который Android сохраняет данные при `onPause/onStop`
- Создаётся объект `StateKeeperDispatcher` — это конкретная реализация интерфейса `StateKeeper`, которая умеет хранить сериализованные значения, зарегистрированные вручную, и при необходимости возвращать их обратно через `consume`.
- Регистрируется новый `SavedStateProvider` — это лямбда, которую Android вызовет при необходимости сохранить состояние. Именно в ней `dispatcher.save()` собирает зарегистрированные значения и подготавливает их к сохранению.

Вызов `SavedStateRegistry.registerSavedStateProvider` здесь — точка подключения к системе восстановления Android. Он позволяет сохранить состояние `StateKeeperDispatcher` в `Bundle`, чтобы при следующем запуске его можно было восстановить. Весь этот механизм — адаптер между KMP-механикой сохранения и Android API.

И вот тут вступает в игру `SerializableContainer`.

Когда вызывается `dispatcher.save()`, все значения, зарегистрированные через `stateKeeper.register(...)`, сериализуются и оборачиваются в `SerializableContainer`.

Это универсальная обёртка, которая хранит данные в виде `ByteArray`, а затем превращает их в строку с помощью `Base64`. Благодаря этому результат можно безопасно сохранить в `Bundle` как обычную строку — без `Parcelable`, `putSerializable()` и без Java `Serializable`. При восстановлении этот путь проходит в обратную сторону: строка → байты → объект через `kotlinx.serialization`.

Таким образом, при вызове `dispatcher.save()` мы получаем сериализуемый контейнер, который можно безопасно положить в `Bundle`. И вот здесь важна не просто сериализация, а то, как именно она устроена. Это не `Parcelable`, и не `Serializable` — это `SerializableContainer`.

`SerializableContainer` — это отдельная сущность, которая оборачивает объект и умеет работать с `kotlinx.serialization` напрямую. Она сама сериализуема, поскольку реализует `KSerializer`, и может быть сохранена в `Bundle` без дополнительных усилий. Ниже — её внутренняя реализация:

```
@Serializable(with = SerializableContainer.Serializer::class)
class SerializableContainer private constructor(
    private var data: ByteArray?,
) {
    constructor() : this(data = null)

    private var holder: Holder<*>? = null

    fun <T : Any> consume(strategy: DeserializationStrategy<T>):
    T? {
        val consumedValue: Any? = holder?.value ?:
        data?.deserialize(strategy)
        holder = null
        data = null
        @Suppress("UNCHECKED_CAST") return consumedValue as T?
    }

    fun <T : Any> set(value: T?, strategy:
    SerializationStrategy<T>) {
        holder = Holder(value = value, strategy = strategy)
        data = null
    }

    private class Holder<T : Any>(
        val value: T?,
        val strategy: SerializationStrategy<T>,
    )
}
```

```

internal object Serializer :
KSerializer<SerializableContainer> {
    private const val NULL_MARKER = "."
    override val descriptor =
PrimitiveSerialDescriptor("SerializableContainer",
PrimitiveKind.STRING)

    override fun serialize(encoder: Encoder, value:
SerializableContainer) {
        val bytes = value.holder?.serialize() ?: value.data
        encoder.encodeString(bytes?.toBase64() ?: NULL_MARKER)
    }

    override fun deserialize(decoder: Decoder):
SerializableContainer =
        SerializableContainer(data =
decoder.decodeString().takeUnless { it == NULL_MARKER
}?.base64ToByteArray())
    }
}

```

Что здесь важно:

- В методе `set(...)` сохраняется объект и соответствующая стратегия сериализации, но не происходит немедленной сериализации.
- Только при вызове сериализатора (`Serializer`) объект превращается в `ByteArray`, а затем в строку.
- После восстановления — `decodeString()` → `ByteArray` → десериализация с использованием заранее известной стратегии.

Это даёт контроль над моментом сериализации и возможность отложенной обработки.

Теперь о том, как это всё оказывается внутри `Bundle`. Ниже — вспомогательные функции, которые используются внутри библиотеки `Essenty/Decompose` для

сериализации и десериализации `SerializableContainer` и произвольных объектов, вызовы которых мы уже встречали в функциях `StateKeeper`:

```
fun <T : Any> Bundle.putSerializable(key: String?, value: T?,
strategy: SerializationStrategy<T>) {
    putParcelable(key, ValueHolder(value = value, bytes = lazy {
value?.serialize(strategy) })))
}

fun <T : Any> Bundle.getSerializable(key: String?, strategy:
DeserializationStrategy<T>): T? =
    getParcelableCompat<ValueHolder<T>>(key)?.let { holder ->
        holder.value ?: holder.bytes.value?.deserialize(strategy)
    }

@Suppress("DEPRECATION")
private inline fun <reified T : Parcelable>
Bundle.getParcelableCompat(key: String?): T? =
    classLoader.let { savedClassLoader ->
        try {
            classLoader = T::class.java.classLoader
            getParcelable(key) as T?
        } finally {
            classLoader = savedClassLoader
        }
    }

fun Bundle.putSerializableContainer(key: String?, value:
SerializableContainer?) {
    putSerializable(key = key, value = value, strategy =
SerializableContainer.serializer())
}

fun Bundle.getSerializableContainer(key: String?):
SerializableContainer? =
    getSerializable(key = key, strategy =
SerializableContainer.serializer())
```

Отдельно стоит упомянуть сущность `ValueHolder`:

```
private class ValueHolder<out T : Any>(
    val value: T?,
    val bytes: Lazy<ByteArray?>,
) : Parcelable {
    override fun writeToParcel(dest: Parcel, flags: Int) {
        dest.writeByteArray(bytes.value)
    }

    override fun describeContents(): Int = 0

    companion object CREATOR :
        Parcelable.Creator<ValueHolder<Any>> {
            override fun createFromParcel(parcel: Parcel):
                ValueHolder<Any> =
                    ValueHolder(value = null, bytes =
                        lazyOf(parcel.createByteArray()))

            override fun newArray(size: Int): Array<ValueHolder<Any>?>
                =
                    arrayOfNulls(size)
        }
}
```

`ValueHolder` здесь нужен для безопасной упаковки сериализованных байт в `Bundle` через `Parcelable`. Он не сериализует объект напрямую — он сохраняет только `ByteArray`, который позже может быть развёрнут обратно в объект через `kotlinx.serialization`. Истинная причина по которой нужен этот объект в том что `Bundle` может хранить `Parcelable` и `Java Serializable`, но он не умеет напрямую работать с `kotlinx.serialization`, по этому он служит в качестве обертки.

Таким образом, `SerializableContainer` + `ValueHolder` — это низкоуровневая инфраструктура сериализации, которая позволяет сохранить произвольные значения Kotlin Multiplatform без зависимостей на Android-специфичные интерфейсы, сохраняя кроссплатформенность и контроль над сериализацией.

К чему это всё ведёт

То есть, по факту, `StateKeeper` — это просто адаптер между внутренней системой хранения состояния в `Essenty/Decompose` и системным `SavedStateRegistry` (а значит — тем самым `onSaveInstanceState` в `Activity/Fragment`, только более удобно и декларативно, и с поддержкой сериализации через `kotlinx.serialization`).

Кратко по цепочке:

1. В компоненте `DefaultCounterComponent` мы вызываем `consume/register` через интерфейс `StateKeeper`.
2. `StateKeeper` реализован как `StateKeeperDispatcher`.
3. `StateKeeperDispatcher` внутри себя хранит значения, сериализует их и регистрирует функцию для сохранения в системный `Bundle` через `SavedStateRegistry`. Важно понять, что значения, которые мы регистрируем в `StateKeeper`, не вызывают напрямую `savedStateRegistry.registerSavedStateProvider` и не создают отдельные `SavedStateProvider`'ы. Всё сохраняется централизованно — в одном объекте `StateKeeperDispatcher`, и только он регистрируется в `SavedStateRegistry`.
4. Всё сериализуется и десериализуется через `kotlinx.serialization`, без `Parcelable`, `Bundle.putXXX()` и прочего boilerplate.

Посмотрим интерфейс `StateKeeper` и его прямого наследника `StateKeeperDispatcher`:

`com.arkivanov.essenty.statekeeper.StateKeeper.kt`:

```
interface StateKeeper {

    fun <T : Any> consume(key: String, strategy:
DeserializationStrategy<T>): T?

    fun <T : Any> register(key: String, strategy:
SerializationStrategy<T>, supplier: () -> T?)

    fun unregister(key: String)
```



```
fun isRegistered(key: String): Boolean
}
```

1. **consume** — извлекает и удаляет ранее сохранённое значение по заданному ключу, используя стратегию десериализации.
2. **register** — регистрирует поставщика значения, которое будет сериализовано и сохранено при следующем сохранении состояния.
3. **unregister** — удаляет ранее зарегистрированного поставщика, чтобы его значение больше не сохранялось.
4. **isRegistered** — возвращает `true`, если по указанному ключу уже зарегистрирован поставщик значения.

com.arkivanov.essenty.statekeeper.StateKeeperDispatcher.kt:

```
interface StateKeeperDispatcher : StateKeeper {

    fun save(): SerializableContainer
}

@JsName("stateKeeperDispatcher")
fun StateKeeperDispatcher(savedState: SerializableContainer? =
    null): StateKeeperDispatcher =
    DefaultStateKeeperDispatcher(savedState)
```

Метод `save()` в `StateKeeperDispatcher` — это тот самый метод, который мы уже встречали ранее: `dispatcher.save()`. Именно он вызывается в момент, когда Android собирается сохранить состояние активности или фрагмента, и через него сериализуются все зарегистрированные значения. Тут мы снова видим функцию `StateKeeperDispatcher`, которую уже встречали ранее. Напомню — это не класс, а фабричная функция, которая создаёт экземпляр `DefaultStateKeeperDispatcher` — единственную реализацию интерфейса `StateKeeperDispatcher`:

```
internal class DefaultStateKeeperDispatcher(
    savedState: SerializableContainer?,
) : StateKeeperDispatcher {
```

```

        private val savedState: MutableMap<String,
SerializableContainer>? = savedState?.consume(strategy =
SavedState.serializer())?.map
        private val suppliers = HashMap<String, Supplier<*>>()

        override fun save(): SerializableContainer {
            val map = savedState?.toMutableMap() ?: HashMap()

            suppliers.forEach { (key, supplier) ->
                supplier.toSerializableContainer()?.also { container -
>
                    map[key] = container
                }
            }

            return SerializableContainer(value = SavedState(map),
strategy = SavedState.serializer())
        }

        private fun <T : Any> Supplier<T>.toSerializableContainer():
SerializableContainer? =
            supplier()?.let { value ->
                SerializableContainer(value = value, strategy =
strategy)
            }

        override fun <T : Any> consume(key: String, strategy:
DeserializationStrategy<T>): T? =
            savedState
                ?.remove(key)
                ?.consume(strategy = strategy)

        override fun <T : Any> register(key: String, strategy:
SerializationStrategy<T>, supplier: () -> T?) {
            check(!isRegistered(key)) { "Another supplier is already
registered with the key: $key" }
            suppliers[key] = Supplier(strategy = strategy, supplier =

```

```

supplier)
    }

    override fun unregister(key: String) {
        check(isRegistered(key)) { "No supplier is registered with
the key: $key" }
        suppliers -= key
    }

    override fun isRegistered(key: String): Boolean = key in
suppliers

    private class Supplier<T : Any>{
        val strategy: SerializationStrategy<T>,
        val supplier: () -> T?,
    }

    @Serializable
    private class SavedState(
        val map: MutableMap<String, SerializableContainer>
    )
}

```

Эта реализация управляет двумя основными структурами:

- `savedState` — карта уже восстановленных значений из `SavedStateRegistry`, если они были сохранены ранее;
- `suppliers` — все зарегистрированные поставщики значений, которые должны быть сериализованы при следующем сохранении состояния.

Когда вызывается метод `save()`, он собирает все текущие значения из `suppliers`, сериализует их и упаковывает в `SerializableContainer`, который затем сохраняется системой. Восстановление происходит через метод `consume()`, где по ключу извлекается значение из `savedState` и десериализуется с помощью переданной стратегии.

Вывод

Мы прошли весь путь — от компонента, использующего `stateKeeper.consume()` и `register()`, до конечного объекта, сериализуемого в `Bundle`. Разобрали, как `StateKeeper` цепляется к `SavedStateRegistry`, как значения хранятся внутри `StateKeeperDispatcher`, и как именно они сохраняются и восстанавливаются через сериализацию.

`StateKeeper` — в android это обёртка над Android Saved State API, которая пришла на замену `onSaveInstanceState`, но реализована декларативно и кроссплатформенно. Она позволяет сохранять произвольные значения через `kotlinx.serialization`, без использования `Parcelable`, `Bundle.putX`, `reflection` и других низкоуровневых деталей.

Давайте визуально глянем на цепочку вызовов что бы понять работу `StateKeeper`:

`StateKeeper.register(...)`:

```
DefaultCounterComponent
├─ stateKeeper.register(...)
│   └─ StateKeeper (интерфейс)
│       └─ StateKeeperDispatcher (интерфейс)
│           └─ DefaultStateKeeperDispatcher.register(...)
│               └─ suppliers[key] = Supplier(...)
```

`StateKeeper(...)` // создание при инициализации

```
├─
SavedStateRegistry.registerSavedStateProvider("state_keeper_key")
├─ dispatcher.save()
│   └─ сериализация значений через
kotlinx.serialization
│   └─ оборачивание в SerializableContainer
│       └─ Bundle.putSerializable("state", ...)
```

`StateKeeper.consume(...)`:

```
defaultComponentContext()
├─ stateKeeper(...)
│   └─ StateKeeper(...)
│       └─ StateKeeperDispatcher(savedState = ...)
```

```

└─ DefaultStateKeeperDispatcher.consume(key,
strategy)
└─
savedState.remove(key)?.consume(strategy)
└─
SerializableContainer.consume(strategy)
└─
kotlinx.serialization.decodeFromByteArray(...)

```

Теперь разберём другой механизм сохранения состояния в Decompose — точнее, в библиотеке **Essenty**, на которой всё построено.

InstanceKeeper

InstanceKeeper — это один из "всадников" **ComponentContext**. Его задача — сохранять произвольные объекты, которые не должны уничтожаться при конфигурационных изменениях (например, при повороте экрана). Это аналог **ViewModel** из Android Jetpack, но в контексте кроссплатформенной разработки (KMP).

Переделаем наш компонент **DefaultCounterComponent**, чтобы вместо **StateKeeper** использовать **InstanceKeeper**:

```

class DefaultCounterComponent(
    componentContext: ComponentContext
) : ComponentContext by componentContext {

    val model: StateFlow<Int> field = instanceKeeper.getOrCreate(
        key = KEY,
        factory = {
            object : InstanceKeeper.Instance {
                val state = MutableStateFlow(0)
            }
        }
    ).state


    fun increase() {
        model.value++
    }
}

```

```
}

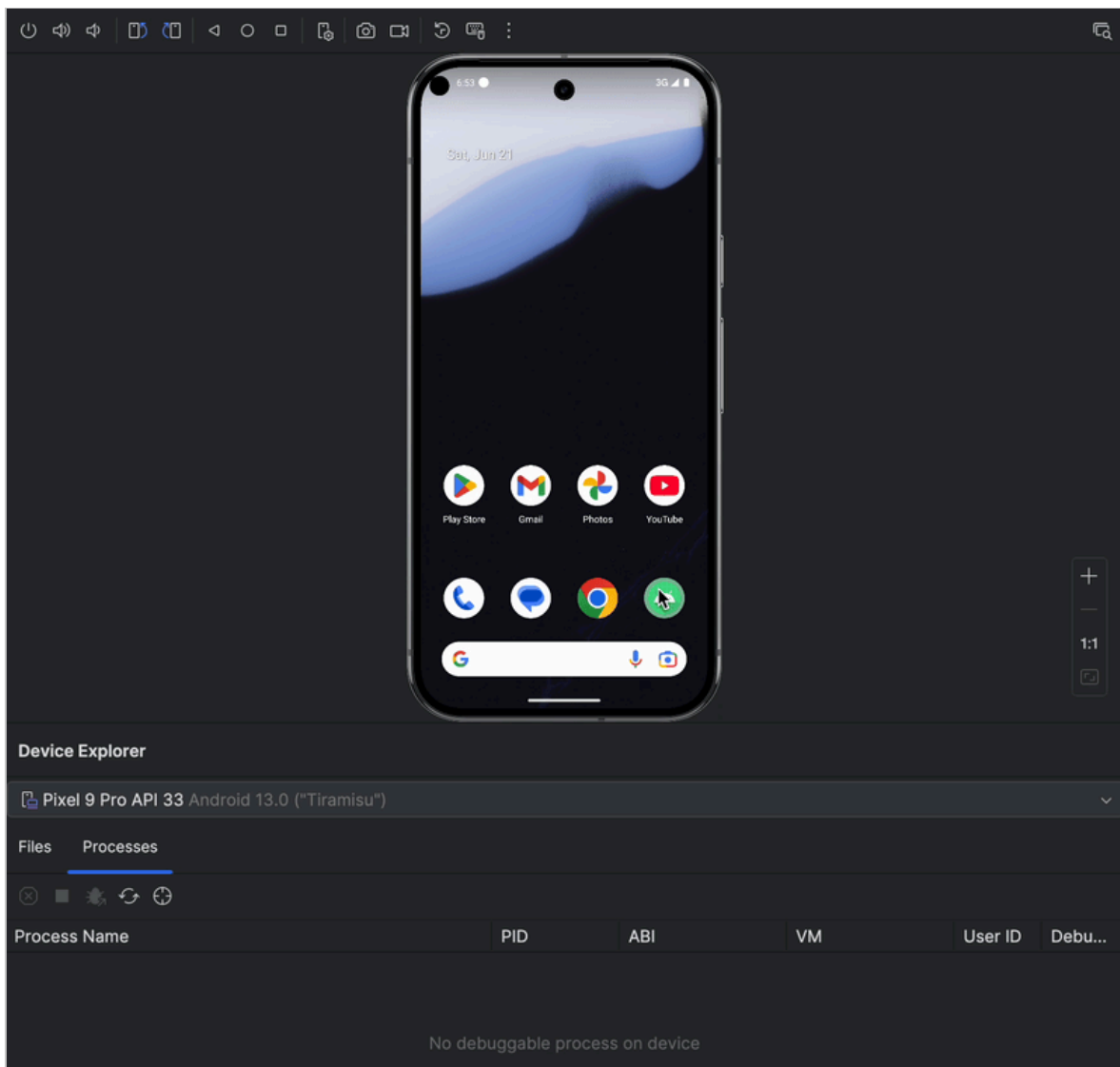
fun decrease() {
    model.value--
}

companion object {
    private const val KEY = "counter_state"
}
}
```

 Обратите внимание: блок `init` был удалён, а изменена только переменная `model`. Всё остальное осталось без изменений.

Теперь давайте проверим поведение визуально:

1. Как будет вести себя счётчик при изменении конфигурации (именно повороте экрана).
2. Как будет вести себя счётчик при уничтожении процесса, когда приложение находится в фоне.



Screenshot

Что мы видим? Счётчик переживает поворот экрана, но обнуляется при смерти процесса. Это как раз поведение `ViewModel`, и именно этого мы ожидаем от `InstanceKeeper`.

Теперь давайте посмотрим, как эта конструкция работает под капотом.

Для начала определим, кто вообще отвечает за хранение `InstanceKeeper`. В `Essenty` (и, соответственно, в `Decompose`) это интерфейс:

```
/**
 * Represents a holder of [InstanceKeeper].
 */
interface InstanceKeeperOwner {
```

```

    val instanceKeeper: InstanceKeeper
}

```

Он реализуется в `GenericComponentContext`, а значит, и в `ComponentContext`, который используется в каждом компоненте:

```

interface GenericComponentContext<out T : Any> :
    LifecycleOwner,
    StateKeeperOwner,
    InstanceKeeperOwner,
    BackHandlerOwner,
    ComponentContextFactoryOwner<T>

interface ComponentContext :
    GenericComponentContext<ComponentContext>

```

Таким образом, цепочка наследования выглядит так: `InstanceKeeperOwner` ← `GenericComponentContext` ← `ComponentContext` ← `DefaultCounterComponent`.

Теперь разберёмся, **откуда приходит реализация**.

В `MainActivity` мы создаём компонент верхнего уровня через функцию `defaultComponentContext()`. Именно она формирует `ComponentContext`, внедряя внутрь все нужные зависимости: `Lifecycle`, `StateKeeper`, `InstanceKeeper`, `BackHandler`.

```

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        ...
        val counterComponent =
            DefaultCounterComponent(defaultComponentContext())
        ...
    }
}

```

Посмотрим ещё раз на исходники `defaultComponentContext()`:


```

fun <T> T.defaultComponentContext(
    discardSavedState: Boolean = false,
    isStateSavingAllowed: () -> Boolean = { true },
): DefaultComponentContext where
    T : SavedStateRegistryOwner, T :
OnBackPressedDispatcherOwner, T : ViewModelStoreOwner, T :
LifecycleOwner =
    defaultComponentContext(
        backHandler = BackHandler(onBackPressedDispatcher),
        discardSavedState = discardSavedState,
        isStateSavingAllowed = isStateSavingAllowed,
    )

```

На этом уровне происходит лишь проксирование вызова — все зависимости собираются и передаются дальше, в приватную функцию:

```

private fun <T> T.defaultComponentContext(
    backHandler: BackHandler?,
    discardSavedState: Boolean,
    isStateSavingAllowed: () -> Boolean,
): DefaultComponentContext where
    T : SavedStateRegistryOwner, T : ViewModelStoreOwner, T :
LifecycleOwner {
    ...
    return DefaultComponentContext(
        lifecycle = lifecycle.asEssentyLifecycle(),
        stateKeeper = stateKeeper,
        instanceKeeper = instanceKeeper(discardRetainedInstances =
marker == null),
        backHandler = backHandler,
    )
}

```

Ключевая строка здесь — `instanceKeeper = instanceKeeper(...)`.

Это и есть та самая точка, где создаётся (или восстанавливается) `InstanceKeeper`. Теперь наша задача — разобраться, что это за функция `instanceKeeper(...)`, как она

устроена и как реализована логика хранения внутри.

Начнём с того, что `instanceKeeper` — это функция-расширение для `ViewModelStoreOwner`. Она становится доступной внутри `defaultComponentContext`, потому что его дженерик явно требует, чтобы вызывающий объект реализовывал интерфейс `ViewModelStoreOwner`. Это условие обеспечивает доступ к `ViewModelStore`, который и передаётся внутрь `InstanceKeeper(...)`. Вот сигнатура этой функции:

```
/**
 * Creates a new instance of [InstanceKeeper] and attaches it to
 * the AndroidX [ViewModelStore].
 *
 * @param discardRetainedInstances a flag indicating whether any
 * previously retained instances should be
 * discarded and destroyed or not, default value is `false`.
 */
fun ViewModelStoreOwner.instanceKeeper(discardRetainedInstances:
Boolean = false): InstanceKeeper =
    InstanceKeeper(viewModelStore = viewModelStore,
discardRetainedInstances = discardRetainedInstances)
```

На первый взгляд кажется, что `InstanceKeeper` — это класс, но в данном случае это вовсе не конструктор, а функция, возвращающая реализацию интерфейса `InstanceKeeper`. Вот как она устроена:

```
/**
 * Creates a new instance of [InstanceKeeper] and attaches it to
 * the provided AndroidX [ViewModelStore].
 *
 * @param discardRetainedInstances a flag indicating whether any
 * previously retained instances should be
 * discarded and destroyed or not, default value is `false`.
 */
fun InstanceKeeper(
    viewModelStore: ViewModelStore,
    discardRetainedInstances: Boolean = false,
): InstanceKeeper =
```

```

viewModelProvider(
    viewModelStore,
    object : ViewModelProvider.Factory {
        @Suppress("UNCHECKED_CAST")
        override fun <T : ViewModel> create(modelClass:
Class<T>): T = InstanceKeeperViewModel() as T
    }
)
    .get<InstanceKeeperViewModel>()
    .apply {
        if (discardRetainedInstances) {
            recreate()
        }
    }
    .instanceKeeperDispatcher

```

Теперь становится понятно: реализация `InstanceKeeper` на Android напрямую завязана на `ViewModelStore`. Концепция долгоживущих объектов реализована здесь через обёртку вокруг обычной `ViewModel`.

Создаётся `InstanceKeeperViewModel`, и далее из неё извлекается `instanceKeeperDispatcher`, который и возвращается как `InstanceKeeper`.

Само API на первый взгляд кажется абстрактным и независимым от Android, но под капотом — чистый `ViewModel`. Причём внутри всей этой логики нет даже намёка на то, что используется Android `ViewModel` — всё скрыто за интерфейсом `InstanceKeeper`.

Вот как устроена `InstanceKeeperViewModel`:

```

internal class InstanceKeeperViewModel : ViewModel() {
    var instanceKeeperDispatcher: InstanceKeeperDispatcher =
InstanceKeeperDispatcher()
    private set

    override fun onCleared() {
        instanceKeeperDispatcher.destroy()
    }
}

```

```

    fun recreate() {
        instanceKeeperDispatcher.destroy()
        instanceKeeperDispatcher = InstanceKeeperDispatcher()
    }
}

```

Что здесь важно:

- `instanceKeeperDispatcher` — это и есть хранилище всех зарегистрированных экземпляров (`InstanceKeeper.Instance`).
- Метод `onCleared()` вызывается, когда `ViewModel` удаляется из `ViewModelStore`. Он вызывает `destroy()` у `dispatcher`, уничтожая все зарегистрированные экземпляры.
- Метод `recreate()` позволяет вручную сбросить все ранее сохранённые экземпляры — полезно, если нужно очистить состояние при пересоздании компонента.

После того как мы поняли, что `InstanceKeeperViewModel` возвращает `instanceKeeperDispatcher`, возникает логичный вопрос — что он из себя представляет.

```

/**
 * Represents a destroyable [InstanceKeeper].
 */
interface InstanceKeeperDispatcher : InstanceKeeper {

    /**
     * Destroys all existing instances. Instances are not cleared,
     so that they can be
     * accessed later. Any new instances will be immediately
     destroyed.
     */
    fun destroy()
}

```

`InstanceKeeperDispatcher` — это интерфейс, расширяющий `InstanceKeeper` и добавляющий к нему жизненно важную функцию `destroy()`. Она уничтожает все текущие экземпляры `Instance`, но не очищает их из внутреннего хранилища — к ним всё ещё можно обращаться при необходимости. Однако любые новые экземпляры, созданные после вызова `destroy()`, уничтожаются сразу.

Метод `destroy()` вызывается системой тогда, когда жизненный цикл компонента подходит к концу — например, при полном удалении из `back stack`. Это позволяет вовремя освободить ресурсы и завершить фоновые задачи.

Реализация создаётся через фабричную функцию:

```
/**
 * Creates a default implementation of [InstanceKeeperDispatcher].
 */
@JsName("instanceKeeperDispatcher")
fun InstanceKeeperDispatcher(): InstanceKeeperDispatcher =
    DefaultInstanceKeeperDispatcher()
```

Теперь разберём, что собой представляет сам `InstanceKeeper`.

```
/**
 * A generic keyed store of [Instance] objects. Instances are
 * destroyed at the end of the
 * [InstanceKeeper]'s scope, which is typically tied to the scope
 * of a back stack entry.
 * E.g. instances are retained over Android configuration changes,
 * and destroyed when the
 * corresponding back stack entry is popped.
 */
interface InstanceKeeper {

    fun get(key: Any): Instance?

    fun put(key: Any, instance: Instance)

    fun remove(key: Any): Instance?
```

```

interface Instance {
    fun onDestroy() {}
}

class SimpleInstance<out T>(val instance: T) : Instance
{

```

`InstanceKeeper` — это ключевое хранилище долгоживущих объектов, которые переживают конфигурационные изменения, но уничтожаются при окончательном завершении жизненного цикла компонента. Типичный пример — удаление элемента из back stack.

Хранилище работает по принципу `key -> Instance` и предоставляет методы для получения, сохранения и удаления объектов.

Сам интерфейс `Instance` минимален: чтобы объект стал управляемым, нужно реализовать единственный метод `onDestroy()`. Он будет вызван системой при уничтожении компонента — это аналог `onCleared()` у `ViewModel`, но с более гибким контролем.

А для случаев, когда никакая очистка не требуется, можно использовать обёртку `SimpleInstance`. Она реализует `Instance`, но ничего не делает в `onDestroy()` — просто превращает любой объект в совместимый с `InstanceKeeper`.

Теперь давай посмотрим, как работает сама реализация хранилища:

```

internal class DefaultInstanceKeeperDispatcher :
    InstanceKeeperDispatcher {

    private val map = HashMap<Any, Instance>()
    private var isDestroyed = false

    override fun get(key: Any): Instance? =
        map[key]

    override fun put(key: Any, instance: Instance) {
        check(key !in map) { "Another instance is already
associated with the key: $key" }

        map[key] = instance
    }
}

```

```

        if (isDestroyed) {
            instance.onDestroy()
        }
    }

    override fun remove(key: Any): Instance? =
        map.remove(key)

    override fun destroy() {
        if (!isDestroyed) {
            isDestroyed = true
            map.values.toList().forEach(Instance::onDestroy)
        }
    }
}

```

`DefaultInstanceKeeperDispatcher` — это конкретная реализация `InstanceKeeperDispatcher`. Внутри у него обычная `HashMap`, где по ключу хранятся все текущие `Instance`. Метод `put()` добавляет объект, предварительно проверяя, что ключ не занят. Флаг `isDestroyed` позволяет отслеживать, завершена ли уже работа хранилища — если `true`, то даже только что добавленный объект сразу уничтожается через `onDestroy()`.

Метод `destroy()` проходит по всем зарегистрированным объектам и вызывает `onDestroy()` у каждого. При этом сами объекты остаются в `map`, чтобы, если нужно, можно было к ним обратиться позже — хотя новые уже не будут жить.

Теперь — о том, что мы используем в нашем компоненте `DefaultCounterComponent`. Там вызывается не `put`, а `getOrCreate`, и вот как он работает:

```

inline fun <T : InstanceKeeper.Instance>
InstanceKeeper.getOrCreate(key: Any, factory: () -> T): T {
    @Suppress("UNCHECKED_CAST")
    var instance: T? = get(key) as T?
    if (instance == null) {
        instance = factory()
        put(key, instance)
    }
}

```

```

    }

    return instance
}

```

Метод `getOrCreate()` — это удобный хелпер: сначала он пробует достать объект по ключу, и если такого ещё нет, — создаёт его через `factory()` и сохраняет.

Используется он в 90% случаев, потому что избавляет от ручной проверки наличия и двойного кода.

DefaultComponentContext

На протяжении всей статьи мы много раз касались функции

`defaultComponentContext()` — именно она выступает точкой входа, где собираются все зависимости компонента:

```

private fun <T> T.defaultComponentContext(
    backHandler: BackHandler?,
    discardSavedState: Boolean,
    isStateSavingAllowed: () -> Boolean,
): DefaultComponentContext where
    T : SavedStateRegistryOwner, T : ViewModelStoreOwner, T :
    LifecycleOwner {
    val stateKeeper = stateKeeper(discardSavedState =
    discardSavedState, isSavingAllowed = isStateSavingAllowed)
    val marker = stateKeeper.consume(key = KEY_STATE_MARKER,
    strategy = String.serializer())
    stateKeeper.register(key = KEY_STATE_MARKER, strategy =
    String.serializer()) { "marker" }

    return DefaultComponentContext(
        lifecycle = lifecycle.asEssentyLifecycle(),
        stateKeeper = stateKeeper,
        instanceKeeper = instanceKeeper(discardRetainedInstances =
    marker == null),
        backHandler = backHandler,
    )
}

```



```
}
```

```
private const val KEY_STATE_MARKER =  
    "DefaultComponentContext_state_marker"
```

Мы уже детально разобрали, откуда здесь берётся `StateKeeper`, как создаётся `InstanceKeeper`, и какую роль играет `marker`. Но до сих пор мы не смотрели внутрь самого `DefaultComponentContext` — давай это исправим:

```
class DefaultComponentContext(  
    override val lifecycle: Lifecycle,  
    stateKeeper: StateKeeper? = null,  
    instanceKeeper: InstanceKeeper? = null,  
    backHandler: BackHandler? = null,  
) : ComponentContext {  
  
    override val stateKeeper: StateKeeper = stateKeeper ?:  
        StateKeeperDispatcher()  
    override val instanceKeeper: InstanceKeeper = instanceKeeper  
        ?: InstanceKeeperDispatcher().attachTo(lifecycle)  
    override val backHandler: BackHandler = backHandler ?:  
        BackDispatcher()  
    override val componentContextFactory:  
        ComponentContextFactory<ComponentContext> =  
        ComponentContextFactory(::DefaultComponentContext)  
  
    constructor(lifecycle: Lifecycle) : this(  
        lifecycle = lifecycle,  
        stateKeeper = null,  
        instanceKeeper = null,  
        backHandler = null,  
    )  
}
```

Как видно, `DefaultComponentContext` — это просто удобный бандл, который объединяет в себе `Lifecycle`, `StateKeeper`, `InstanceKeeper` и `BackHandler`. Если какие-то зависимости не были переданы извне — он сам создаёт дефолтные

реализации. Всё это обернуто в единый объект `ComponentContext`, который дальше передаётся в компоненты и навигационные структуры.

Таким образом, `DefaultComponentContext` можно считать связующим звеном между Android-инфраструктурой и кроссплатформенной архитектурой `Decompose` — он превращает низкоуровневые сущности в универсальный интерфейс.

Финал

Если вы дошли до этого момента — значит, прошли со мной весь путь по хранению состояний в Android на глубоком, подкапотном уровне: от того, где реально живёт `ViewModelStore` в `Activity` и `Fragment`, до того, как `ViewModel` хранятся в `Compose` и `View`, как работает `Saved State API`, чем отличается от `onSaveInstanceState`, и где в итоге оказывается `Bundle`.

В последней части мы разобрали, как устроена логика сохранения состояния в `Decompose` и `Essenty`, чтобы снять иллюзию "магии" и показать, что под капотом — всё те же стандартные механизмы Android, просто обернутые в более универсальный API. Всё это рассматривалось строго через призму хранения и восстановления данных.

Эта статья завершает серию. Всё, что здесь написано — не документация и не руководство. Это просто попытка заглянуть внутрь, разобраться и собрать цельную картину.

Если посчитаете, что это может быть полезно кому-то ещё — можете поделиться. Если захотите обсудить или предложить правки — я открыт.