

КАК СТАТЬ АВТОРОМ

Травка зеленеет, солнышко блестит, а скидки растут



2176.69

Рейтинг

MTC

Про жизнь и развитие в IT

[Подписаться](#)

zakh_serge

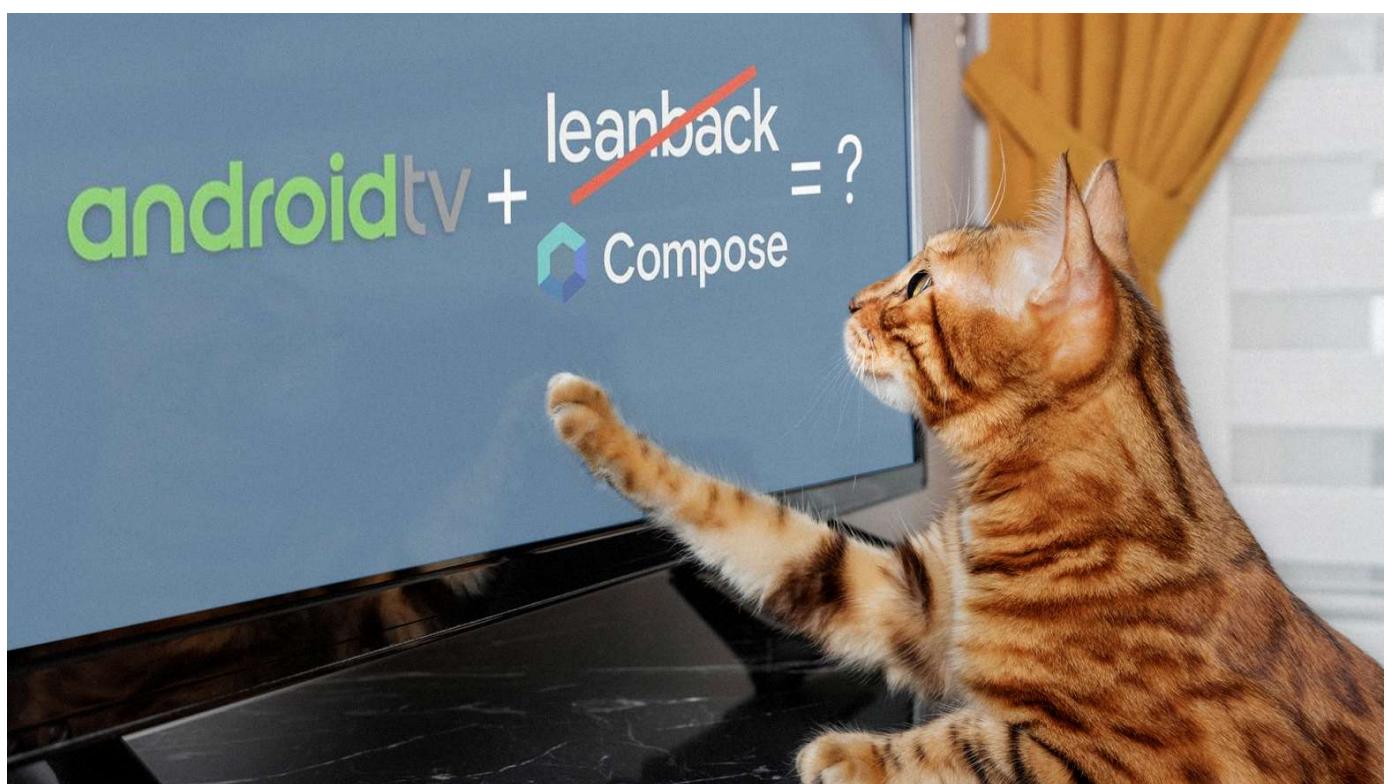
вчера в 12:54

Jetpack Compose для Android TV: как происходит перемещение фокуса

10 мин

755

Блог компании МТС, Jetpack Compose*, Разработка под Android*, Дизайн мобильных приложений*, Разработка



Привет, Хабр! Меня зовут Сергей Захаров, я разработчик в онлайн-кинотеатре KION. Jetpack Compose постепенно становится стандартом разработки UI для Android, и Android TV не остался в стороне. Вместо устаревшего и громоздкого Leanback теперь есть гибкий и мощный фреймворк, который обещает упростить разработку.

Однако на Android TV ключевую роль играет управление фокусом — без него невозможна навигация. Как же Compose справляется с этой задачей? Достаточно ли он удобен для больших TV-проектов, или лучше ограничиться им в мобильных приложениях? В этом и следующих материалах я отвечу на эти вопросы на примере конкретных кейсов.

Это первый пост моего цикла про работу с фокусом в Jetpack Compose. В нем я расскажу, как в Jetpack Compose перемещается фокус, рассмотрю используемые при этом компоненты, покажу их плюсы и минусы, а также дам рекомендации по их применению в приложениях для Android TV.

Работа с фокусом в мобильных устройствах

Кажется, что при разработке Jetpack Compose его создатели не уделяли особого внимания TV-сегменту и работе с фокусом в нем. Поэтому логично сначала рассмотреть Compose в мобильных приложениях, а затем перейти к Android TV.

На практике в мобильных приложениях необходимость вручную управлять фокусом возникает редко. Чаще всего это требуется при работе с полями ввода (TextField). Разберем два типичных кейса.

Перевод фокуса на TextField и открытие клавиатуры по нажатию на кнопку

Для этого используется FocusRequester. Мы создаем его экземпляр, передаем в TextField через модификатор focusRequester(), а затем вызываем requestFocus() при нажатии на кнопку:

```
var textField by remember { mutableStateOf("") }
val focusRequester = remember { FocusRequester() }
Column(modifier = Modifier.padding(32.dp)) {
    Button(onClick = {
        focusRequester.requestFocus()
    }) {
        Text("Request Focus")
    }
    Text("Textfield title", modifier = Modifier.padding(16.dp))
    TextField(
        value = textField,
        onValueChange = { textField = it },
        modifier = Modifier.focusRequester(focusRequester)
    )
}
```

Фокус переходит на TextField, и клавиатура открывается.

Перемещение фокуса между полями ввода

Предположим, у нас есть два TextField. Нам нужно, чтобы при нажатии Enter в первом поле фокус автоматически переходил на второе, а уже при нажатии Enter в нем — клавиатура закрывалась.

Для этого используется FocusManager, который можно получить через LocalFocusManager.current:

```
val focusManager = LocalFocusManager.current
var textField1 by remember { mutableStateOf("") }
var textField2 by remember { mutableStateOf("") }

LazyColumn(modifier = Modifier.padding(16.dp)) {
    item {
        Text("Textfield title", modifier = Modifier.padding(16.dp))
        TextField(
            value = textField1,
            onValueChange = {
                textField1 = it
            },
            singleLine = true,
            keyboardActions = KeyboardActions {
                focusManager.moveFocus(FocusDirection.Next)
            },
        )
    }
    item {
        Text("Textfield title ", modifier = Modifier.padding(16.dp))
        TextField(
            value = textField2,
            onValueChange = {
                textField2 = it
            },
            singleLine = true,
            keyboardActions = KeyboardActions {
                focusManager.clearFocus()
            },
        )
    }
}
```

В `keyboardActions` первого `TextField` вызываем `moveFocus(FocusDirection.Next)`, чтобы фокус перешел на следующее поле. Во втором `TextField` вызываем `clearFocus()` для закрытия клавиатуры.

В мобильных приложениях управление фокусом чаще всего сводится к использованию `FocusRequester` и `FocusManager`. Однако в Android TV это делать гораздо сложнее, и стандартные механизмы Compose не всегда функционируют ожидаемо. Давайте разберемся, как они устроены внутри и как с ними работать на телевизорах.

FocusRequester — самый проблемный класс

`FocusRequester` — это утилитный класс в Jetpack Compose, предназначенный для управления фокусом элементов UI. Он позволяет выполнять следующие операции с фокусом:

1. Запрашивать для компонента (например, для поля ввода или кнопки).
2. Перехватывать, чтобы другие элементы не могли его захватить.
3. Освобождать, разрешая иным компонентам получить его.
4. Сохранять и восстанавливать его состояние, чтобы вернуться к ранее сфокусированному элементу.

Разработчикам доступны следующие публичные функции:

```
class FocusRequester {  
  
    internal val focusRequesterNodes: MutableVector<FocusRequesterModifierNode> = mutableListOf()  
  
    //Запрашивает фокус для компонента, связанного с этим FocusRequester  
    fun requestFocus()  
  
    //Перехватывает фокус, чтобы запретить другим компонентам перехватить его  
    fun captureFocus(): Boolean  
  
    //Освобождает захваченный фокус, разрешая другим компонентам запрашивать фокус  
    fun freeFocus(): Boolean  
  
    //Сохраняет ссылку на текущий дочерний компонент, который имеет фокус, в состоянии  
    fun saveFocusedChild(): Boolean  
  
    //Восстанавливает фокус на дочернем компоненте, который был ранее сохранен методом saveFocusedChild  
    fun restoreFocusedChild()  
}
```

```
fun restoreFocusedChild(): Boolean  
}
```

Что внутри FocusRequester

Переменная focusRequesterNode

Она представляет собой коллекцию типа MutableVector. Это специализированная коллекция, оптимизированная для работы с изменяемыми списками данных в Compose. MutableVector был создан для повышения производительности в сценариях, где элементы часто добавляются, удаляются или обновляются.

Подробности этой коллекции можно оставить за пределами этого материала (<https://developer.android.com/reference/kotlin/androidx/compose/runtime/collection/MutableVector>), но важно понимать: она используется для хранения нод, связанных с запросом фокуса. Это значит, что один FocusRequester может быть связан с несколькими компонентами, что дает нам большую гибкость.

Например, можно использовать один FocusRequester для разных TextField:

```
var textField1 by remember { mutableStateOf("") }  
var textField2 by remember { mutableStateOf("") }  
val focusRequester = remember { FocusRequester() }  
Column(modifier = Modifier.padding(32.dp)) {  
    Button(onClick = {  
        focusRequester.requestFocus()  
    }) {  
        Text("Request Focus")  
    }  
    Text("Textfield title", modifier = Modifier.padding(16.dp))  
    TextField(  
        value = textField1,  
        onValueChange = {  
            textField1 = it  
        },  
        modifier = Modifier.focusRequester(focusRequester)  
    )  
  
    Text("Textfield title2", modifier = Modifier.padding(16.dp))  
    TextField(  
        value = textField2,
```

```

onValueChange = {
    textField2 = it
},
modifier = Modifier.focusRequester(focusRequester)
)
}

```

В таком случае фокус перейдет на последний элемент, который был добавлен в список. В данном примере это второй TextField. Такое поведение будет полезным, например, в динамических интерфейсах, где компоненты добавляются или исчезают в зависимости от состояния. Наличие нескольких нод позволяет адаптировать FocusRequester к таким изменениям: если один из элементов исчезает, фокус автоматически перемещается на другой.

Это также помогает избежать создания отдельного FocusRequester для каждой группы компонентов. Вместо этого можно создать один FocusRequester, который будет работать с разными группами.

Метод findFocusTarget

Он отвечает за поиск элемента, куда можно установить фокус. Рассмотрим его код:

```

@ExperimentalComposeUiApi
private inline fun findFocusTarget(onFound: (FocusTreeNode) -> Boolean): Boolean {
    check(this !== Default) { InvalidFocusRequesterInvocation }
    check(this !== Cancel) { InvalidFocusRequesterInvocation }
    check(focusRequesterNodes.isNotEmpty()) { FocusRequesterNotInitialized }
    var success = false
    focusRequesterNodes.forEach { node ->
        node.visitChildren(Nodes.FocusTarget) {
            if (onFound(it)) {
                success = true
                return@forEach
            }
        }
    }
    return success
}

```

Этот метод выполняет несколько действий:

1. Проверку на специальные объекты. Первые две строки смотрят, что текущий FocusRequester не является служебным объектом FocusRequester.Default или FocusRequester.Cancel. Они используются для управления поведением фокуса в особых ситуациях и не должны передаваться в Modifier.
2. Проверку на пустоту списка. Если в focusRequesterNodes пусто, выбрасывается исключение FocusRequesterNotInitialized. Это важно, потому что попытка запросить фокус при пустом списке приведет к ошибке. Заполнение focusRequesterNodes происходит через модификатор focusRequester.

Будьте аккуратными при вызове метода requestFocus(): сначала убедитесь, что нужный компонент действительно есть на экране. Иначе можно столкнуться с ситуацией, когда фокус будет запрашиваться на несуществующем элементе, что приведет к ошибке. Минимальной доработкой кажется добавление метода, который мог бы проверять список focusRequesterNodes на наличие в нем элементов:

```
fun hasActiveNodes(): Boolean = !focusRequesterNodes.isEmpty()
```

Тогда у нас было бы сильно больше контроля над происходящим, что облегчило бы жизнь во многих случаях. Мы еще вернемся к этой проблеме и обсудим варианты решений.

FocusManager

Этот интерфейс предоставляет доступ к управлению фокусом внутри иерархии пользовательского интерфейса и позволяет перемещать фокус в заданном направлении и освобождать его вручную:

```
interface FocusManager {  
  
    fun clearFocus(force: Boolean = false)  
  
    fun moveFocus(focusDirection: FocusDirection): Boolean  
}
```

Он доступен разработчику, однако внутри Jetpack Compose существует более глубокий уровень управления фокусом — интерфейс FocusOwner.

```
internal interface FocusOwner : FocusManager {  
  
    val modifier: Modifier  
  
    val focusTransactionManager: FocusTransactionManager  
  
    fun requestFocusForOwner(focusDirection: FocusDirection?, previouslyFocusedRect: Rect?)  
  
    fun focusSearch(  
        focusDirection: FocusDirection,  
        focusedRect: Rect?,  
        onFound: (FocusTargetNode) -> Boolean  
    ): Boolean?  
  
    fun takeFocus(focusDirection: FocusDirection, previouslyFocusedRect: Rect?): Boolean  
  
    fun releaseFocus()  
  
    fun clearFocus(  
        force: Boolean,  
        refreshFocusEvents: Boolean,  
        clearOwnerFocus: Boolean,  
        focusDirection: FocusDirection  
    ): Boolean  
  
    fun getFocusRect(): Rect?  
  
    fun dispatchKeyEvent(keyEvent: KeyEvent, onFocusedItem: () -> Boolean = { false }): Boolean  
  
    fun dispatchInterceptedSoftKeyboardEvent(keyEvent: KeyEvent): Boolean  
  
    fun dispatchRotaryEvent(event: RotaryScrollEvent): Boolean  
  
    fun scheduleInvalidation(node: FocusTargetNode)  
  
    fun scheduleInvalidation(node: FocusEventModifierNode)  
  
    fun scheduleInvalidation(node: FocusPropertiesModifierNode)  
  
    val rootState: FocusState  
}
```

Он уже имеет гораздо больше методов, и его структура дает нам больше представления о работе фокуса. Давайте подробней рассмотрим, как по команде происходит перемещение фокуса.

Вот код moveFocus:

```
override fun moveFocus(focusDirection: FocusDirection): Boolean {
    val source = rootFocusNode.findActiveFocusNode() ?: return false

    source.customFocusSearch(focusDirection, layoutDirection).also {
        if (it !== Default) {
            return it !== Cancel && it.focus()
        }
    }

    var isCancelled = false
    val foundNextItem =
        rootFocusNode.focusSearch(focusDirection, layoutDirection) { destination ->
            if (destination == source) return@focusSearch false
            checkNotNull(destination.nearestAncestor(Nodes.FocusTarget)) {
                "Focus search landed at the root."
            }
            when (destination.performCustomRequestFocus(focusDirection)) {
                Redirected -> true
                Cancelled, RedirectCancelled -> { isCancelled = true; true }
                None -> destination.performRequestFocus()
            }
        }
    return !isCancelled && (foundNextItem || wrapAroundFocus(focusDirection))
}
```

Если кастомного поведения нет, то используется focusSearch — стандартный алгоритм поиска узлов, который проходит по дереву фокуса. На его наличие проверяется каждая нода, а найденный узел становится новым кандидатом на фокус.

Если следующий узел найден, выполняется попытка установить на него фокус с помощью вызовов: performRequestFocus для случаев стандартного поведения или performCustomRequestFocus при наличии кастомных правил.

Если фокус не смог перейти ни на один узел в иерархии, метод проверяет возможность «обхода» (wrap-around). Это, например, позволяет переместиться с последнего элемента на первый.

Как же все-таки перемещается фокус?

Это происходит с помощью нескольких последовательных операций:

- **поиск активного элемента.** Иерархия фокуса просматривается рекурсивно для поиска узла, который находится в состоянии Active. Это нормально для мобильных телефонов, когда у тебя мало элементов на экране и смена фокуса происходит редко. Для постоянного перемещения такая система выглядит слишком энергозатратной. Если фокусный узел найден, то он становится отправной точкой для вычисления следующего узла;
- **выполнение перемещения.** Когда поиск завершен успешно алгоритм переходит к попытке сместить фокус. При наличии сначала выполняется пользовательская логика. Если она вернула следующий узел, перемещение завершается. Если кастомное перемещение не задано, то фокус переходит к следующему элементу, который определяется на основе направления (FocusDirection);
- **проверка узлов на возможность получения фокуса.** Если подходящий узел найден, он получает фокус. Если стандартный поиск не смог найти нужный узел, то метод запускает «обход». Если он невозможен, фокус остается на текущем узле и метод возвращает false. При ошибке в процессе поиска или выполнения (например, узел не имеет фокусируемого родителя или структура иерархии нарушена) логика сбрасывает фокус, чтобы переместиться к корневому узлу. И фокус переходит на первый элемент дерева (если это возможно).

И тут есть вопрос: а может ли где-то скрываться проблема? Что будет, если пользователь начнем многократно перемещать фокус в разные стороны?

Разработчик обычно нажимает кнопку на пульте, ждет завершения анимации и только потом нажимает другую кнопку. Пользователи же жмут как попало и что попало.

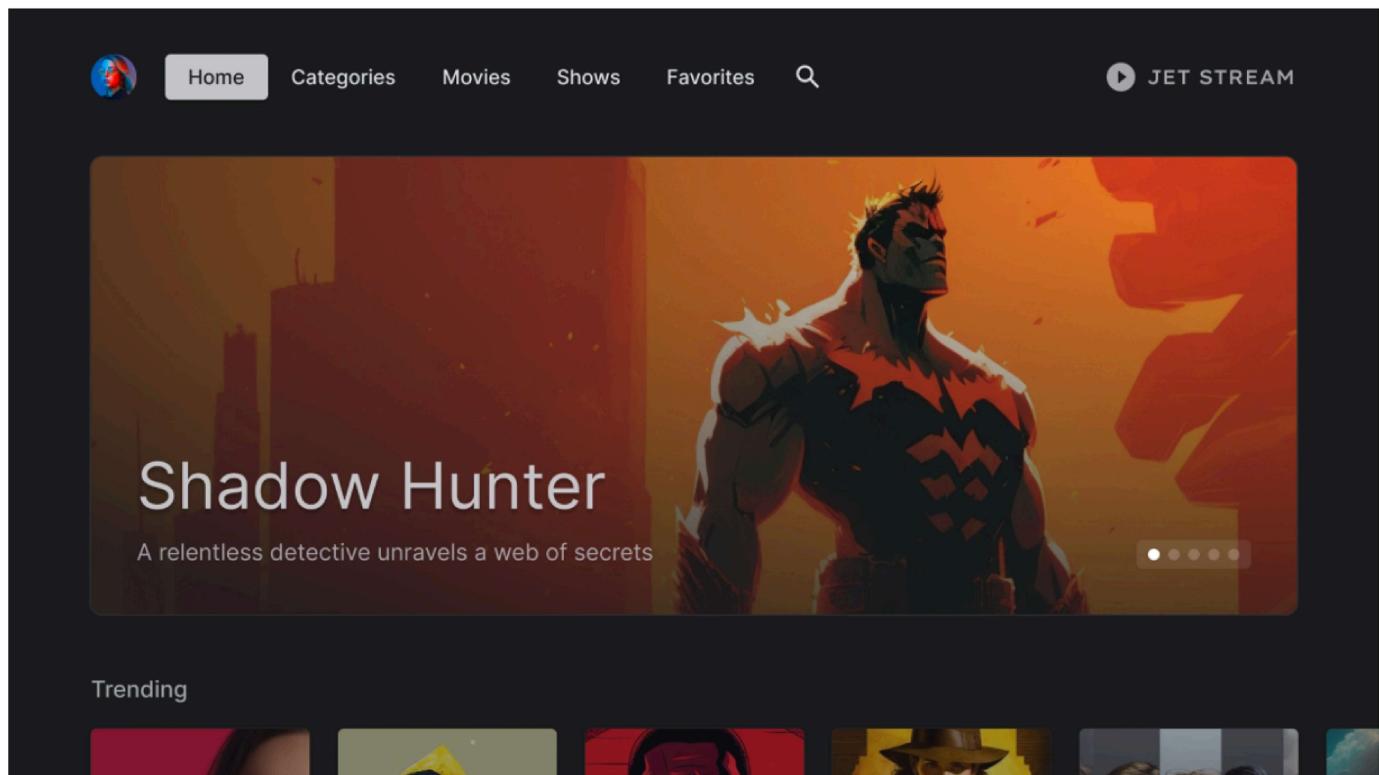


В случае быстрого перемещения фокуса возникают проблемы, связанные с состоянием и одновременной обработкой нескольких запросов.

Если задействованы узлы, которые еще не полностью инициализированы (например, из-за задержек в рендеринге или скроллинге), то первый запрос попытается переместить фокус к следующему узлу, пока тот еще не готов.

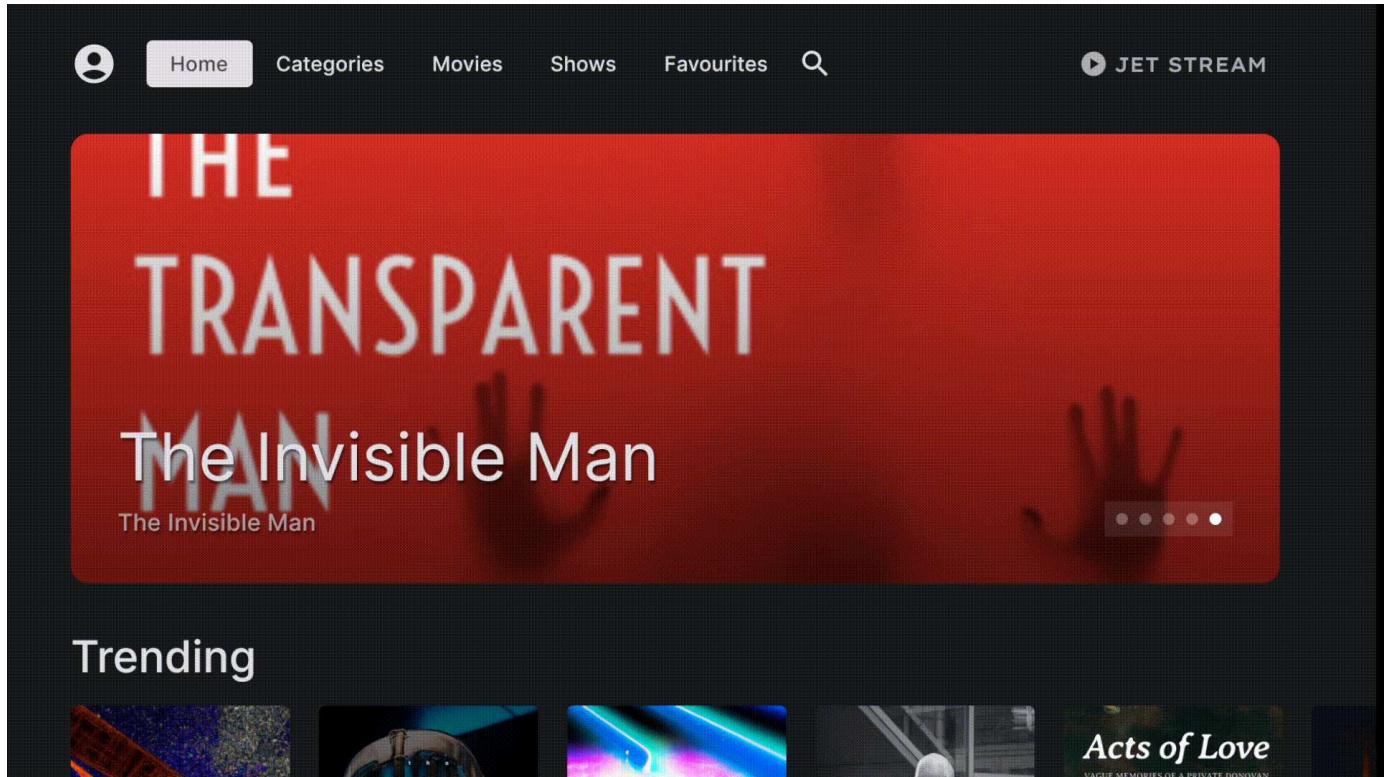
Второй запрос может завершить перемещение, но вернет неправильное состояние или вызовет сброс фокуса. Если же система обнаруживает недопустимое состояние (например, узел потерял фокус), то он сбрасывается к корню.

Best practice от Google



Jet Stream — это эталонное приложение, которое показывает лучшие практики разработки для Android TV на Jetpack Compose. Я беру его как пример, потому что оно очень красивое, а еще в нем мы можем посмотреть решения самих разработчиков фреймворка. Спойлер: в приложении присутствуют далеко не все сложные кейсы.

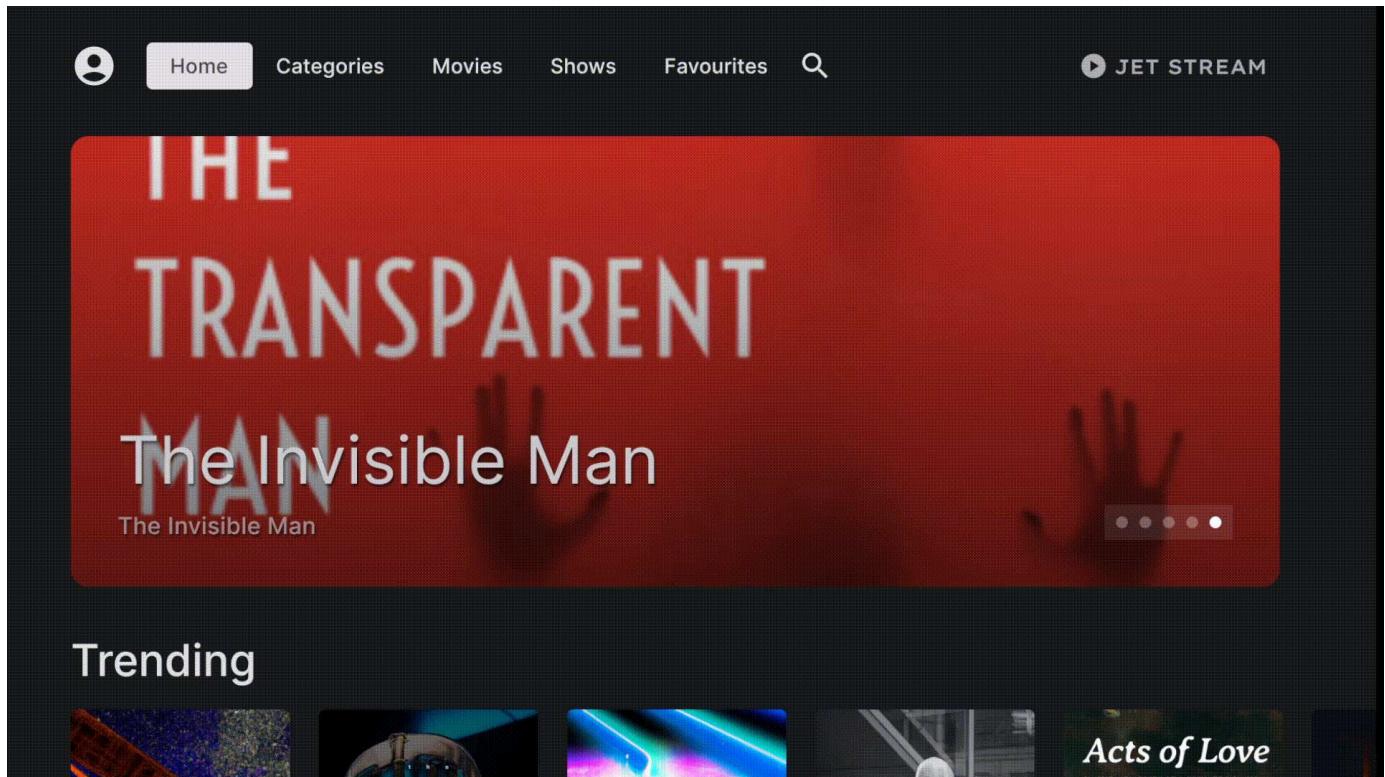
Начнем с перемещения фокуса. В Jet Stream есть верхнее меню — с его помощью можно переходить на разные страницы. Если сначала нажать на пульте кнопку «вправо», а после перемещения таба — «вниз», то все отрабатывает штатно. Мы перейдем на первый элемент в категориях:



Trending



Но что будет, если ускорить этот процесс?



Мы видим проблему, которую обсудили в прошлом разделе. При быстром перемещении экран не успевает перестроиться, и фокус падает на первую фокусируемую ноду в дереве. В данном случае это кнопка с профилем.

То есть даже в эталонном решении от Google присутствуют явные недочеты. Хотя по своему замыслу оно довольно простое и не учитывает множества более сложных кейсов.

Поэтому часто приходится искать свои решения.

Работа с фокусом — это краеугольный камень Android TV-разработки

В отличие от мобильных приложений, здесь фокус — это не просто дополнительная фича для удобства, а основной способ навигации. Мы разобрали базовые инструменты управления фокусом в Jetpack Compose: FocusManager и FocusRequester. Однако важно отметить: сам Jetpack Compose изначально не проектировался под Android TV. Управление фокусом здесь выглядит **надстройкой**, а не фундаментальной частью архитектуры.

Отсюда множество нюансов:

- рекурсивный поиск активной FocusNode через деревья композиций;
- конкурентный доступ к фокусу при динамическом UI;
- как следствие первых двух пунктов: нестабильное поведение при быстрой смене экранов или появлении новых элементов.

Несмотря на это, правильное понимание текущей модели позволяет эффективно использовать Compose даже под TV. Если дочитали до этого момента, вы уже сделали важный шаг в сторону грамотной архитектуры и UX в Android TV-приложениях на Compose.

В следующей части я расскажу, какое API нам предоставляется для кастомизации фокуса, и рассмотрю некоторые кейсы, важные для TV-разработки. Если у вас вопросы о Jetpack Compose — задавайте их в комментариях и читайте другие материалы из цикла:

- Управление фокусом в Jetpack Compose. Что доступно из коробки (coming soon)
- Кастомные реализации фокус-системы для Jetpack Compose (coming soon)

Теги: androidtv, Jetpack Compose, декларативное программирование, FocusManager, FocusRequester, findFocusTarget, мобильные приложения, разработка под android, Фокус в мобильных приложениях, дизайн мобильных приложений

Хабы: Блог компании МТС, Jetpack Compose, Разработка под Android, Дизайн мобильных приложений, Разработка мобильных приложений

Редакторский дайджест

Присыпаем лучшие статьи раз в месяц

Электропочта



MTC

Про жизнь и развитие в IT



3

5

Карма Рейтинг

Захаров Сергей @zakh_serge

Пользователь

[Подписаться](#)

Полезные ссылки

25 дек 2024 в 18:42

Remote Config и A/B-эксперименты: история разработки и основные возможности

8 мин 1.2K

+8

9



0

17 дек 2024 в 17:00

Путь видео в онлайн-кинотеатрах от «стекла до стекла». Часть первая: источники данных и headend

Простой 6 мин 1.3K

[Обзор](#)

+9

6



7

26 ноя 2024 в 19:01

Доверяй, но проверяй: как мы в KION идем по пути продуктовых событий

9 мин 1.1K

+7

6



4

25 окт 2024 в 17:24

Machine Learning в онлайн-кинотеатрах: как повысить время смотрения и понять, что одного ML мало. Часть 1

⌚ 5 мин ⚡ 2.1K

+6

18



8

Комментировать

ВАКАНСИИ КОМПАНИИ «МТС»

Senior разработчик 1С:Аренда [Центр 1С, МТС Веб Сервисы]

МТС · Москва · Можно удаленно

Solution архитектор [Enablement Platform, МТС Веб Сервисы]

МТС · Можно удаленно

Senior Golang разработчик в команду Network [MTC Web Services (MWS)]

МТС · Москва · Можно удаленно

Больше вакансий на Хабр Карьере

ИНФОРМАЦИЯ

Ваш аккаунт

Войти

Регистрация

Разделы

Статьи

Новости

Хабы

Компании

Авторы

Песочница

Информация

Устройство сайта

Для авторов

Для компаний

Документы

Соглашение

Конфиденциальность

Услуги

Корпоративный блог

Медийная реклама

Нативные проекты

Образовательные

программы

Стартапам



Настройка языка

© 2006–2025, Habr

БЛОГ НА ХАБРЕ

5 часов назад

Проблемы Joy-Con для Nintendo Switch: как ремонтировать контроллеры 431  0

10 часов назад

FreeIPA: как обнаружить атаку злоумышленника на любом этапе Kill Chain 563  0

вчера в 17:00

Готовимся к встрече с Python 3.14: разбор суперсил 11K  12

вчера в 12:54

Jetpack Compose для Android TV: как происходит перемещение фокуса 755  0

27 апр в 13:00

Пять свежих проектов на Raspberry Pi: мини-ПК, кибердек, медиаплеер и кое-что еще 8K  8