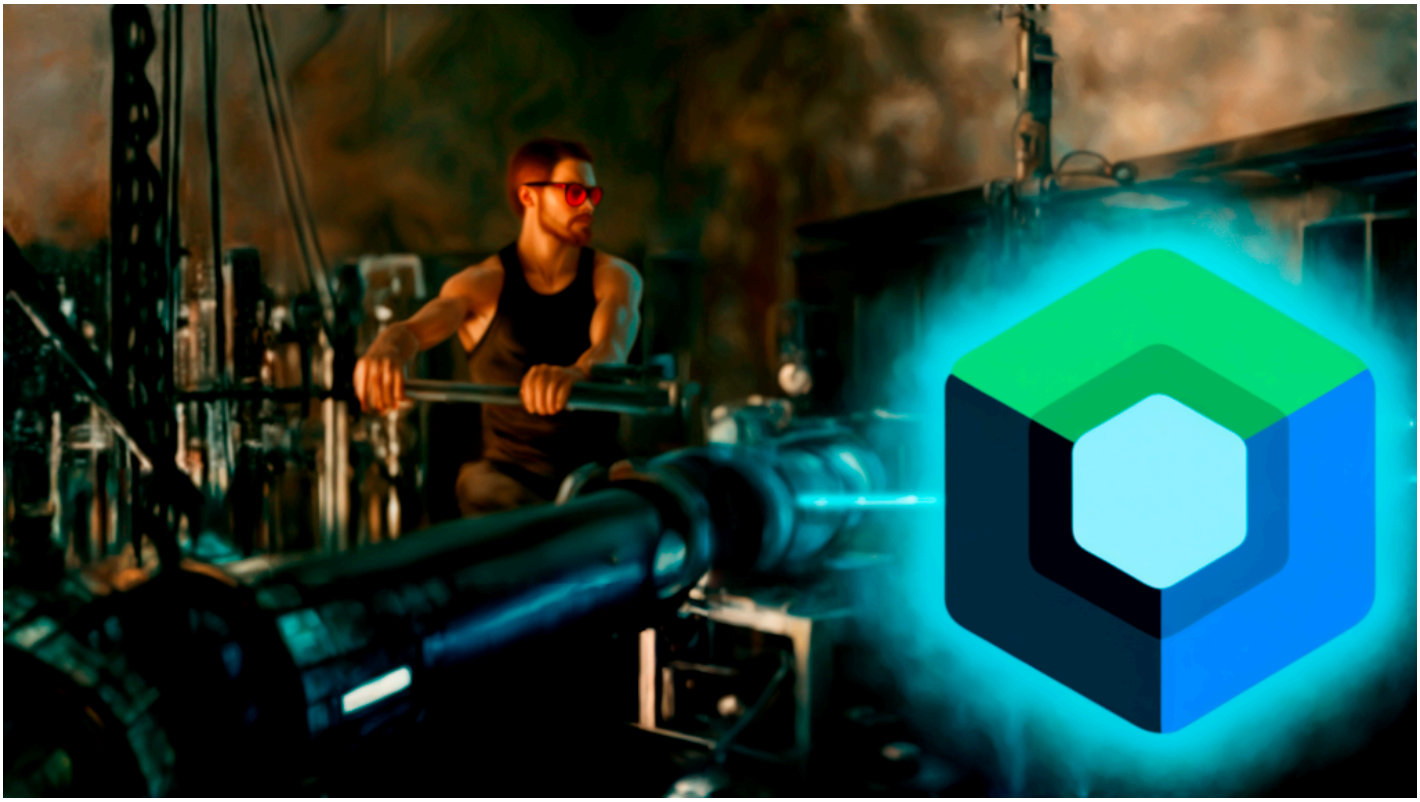
tipapro 28 фев 2024 в 07:00

Осознанная оптимизация Compose 2: В борьбе с композицией

Средний | 15 мин | 14K

Программирование*, Разработка мобильных приложений*, Android*, Kotlin*, Jetpack Compose*

Туториал



Jetpack Compose постоянно развивается, открывая перед разработчиками новые горизонты для оптимизации. С момента нашего последнего обзора, мы добились значительного прогресса, сократив задержки при скролле с 5-7% до нуля. В этом материале мы поделимся свежими находками и передовыми практиками в оптимизации Compose. Чтобы максимально углубиться в тему, рекомендуем ознакомиться с первой частью.

Серия статей:

1. Осознанная оптимизация Compose (medium, eng)
2. Осознанная оптимизация Compose 2: В борьбе с композицией (текущая) (medium, eng)

Содержание

    - Проблема начальной композиции
    - Modifier.Node
    - Проблемы DerivedState и remember
    - Пересядь с иглы Compose на старый добрый Kotlin
    - Пререндер
    - Отложенная композиция
- Painter
    - Xml-иконки vs Compose-иконки
    - Нестабильность Painter
    - Вынос Painter
- Дизайн система
    - Цветовая схема
    - Ошибки прошлого
- Форматтеры
- Оптимизация на спичках
    - Автоупаковка
    - Быстрые методы
    - Эффективные структуры
    - Вложенные if
- Нововведения
    - Указание стабильности внешних типов
    - Режим сильной пропускаемости
- Инструментарий
    - Просмотр исходного кода Compose
    - Vkompose плагин
    - Detekt
- Итог

Композиция - низвергнутый бог

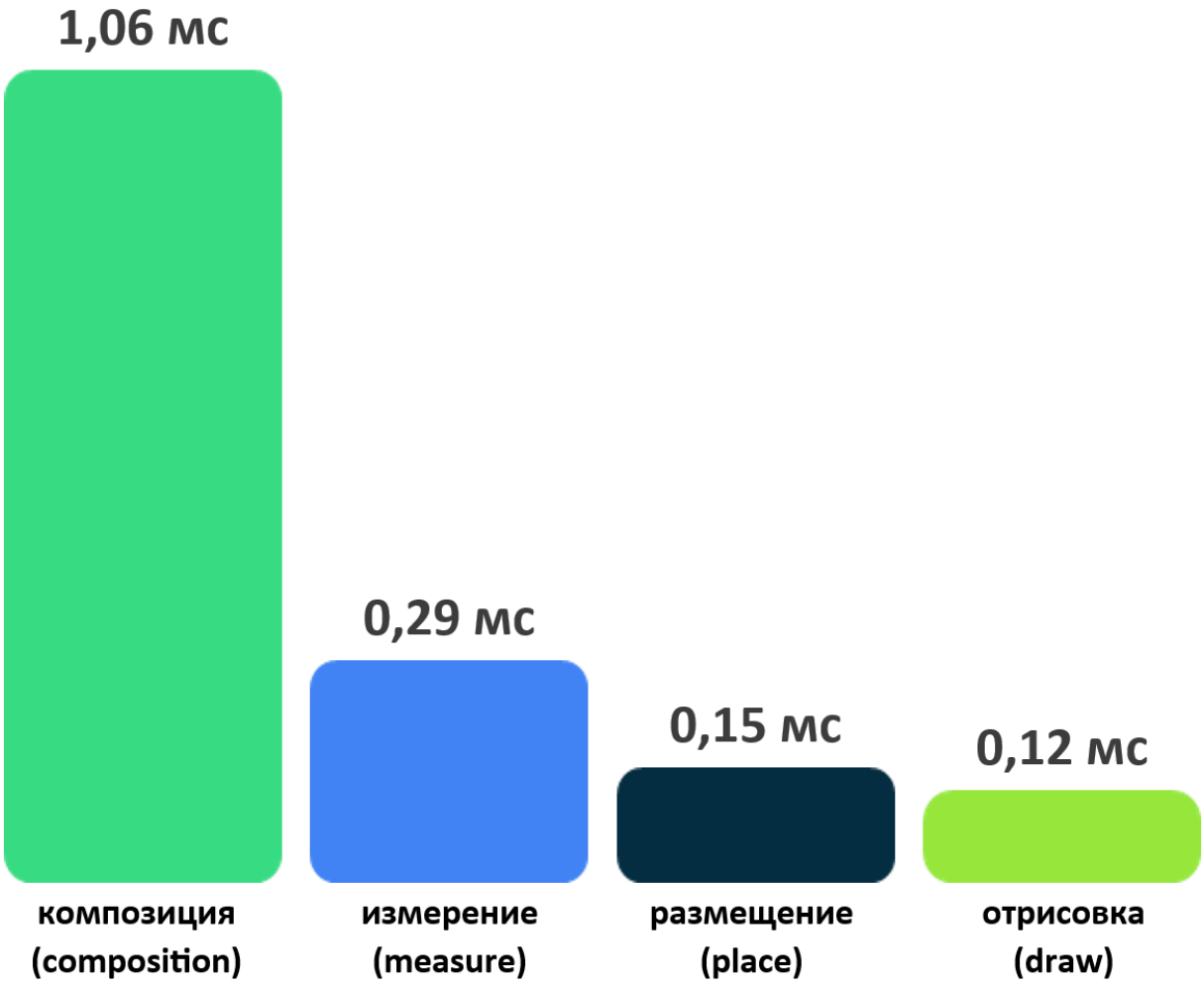
Проблема начальной композиции

В первой части была описана одна из проблем ленивых списков - использование под капотом `SubcomposeLayout`. По словам разработчиков, ещё одной проблемой является скорость начальной композиции. Во время этого затратного этапа в первый раз строится дерево элементов. Для ленивых списков этот момент особенно критичен, поскольку начальная композиция происходит при создании каждого элемента.

В подкасте разработчики подчеркивают, что Compose проектировался исходя из того, что людям проще думать в категориях простых лейаутов (`Box`, `Column`, `Row`), а не таких сложных, как `ConstraintLayout`. В отличие от View, у Compose нет проблемы экспоненциального увеличения количества измерений, так как они ограничены одним проходом. Пропуски функций также способствуют решению проблемы вложенности. Но не в случае начальной композиции, когда нужно пройтись по тысяче лейаутов. Разработчики стремятся снизить и эту нагрузку: переход с версии Compose 1.4 на 1.6, по их словам, ускорит работу списков на 40%, что служит весомым аргументом в пользу обновления. Однако для уже оптимизированных списков прирост производительности может быть не так заметен.

Композиция в Compose обходится дорого, но это цена за удобство декларативного подхода. В данной статье мы предложим способы минимизации затрат на композицию и снижения их стоимости при рекомпозициях.

Для начала давайте посмотрим, как время композиции соотносится с другими фазами Compose:



Среднее время каждой фазы. Данные для слабого устройства. Источник

Из этого графика становится ясно, почему перенос работы на следующую фазу из композиции оказывается таким эффективным. Однако при этом не следует забывать о затратах на создание лямбд и потенциальных проблемах с их равенством при неудачном замыкании. Поэтому и стоит их использовать для откладывания частых изменений, а не единичных.

Modifier.Node

`Modifier.composed` долгое время был основным инструментом для создания кастомных модификаторов. Однако, несмотря на его гибкость, этот подход имеет свои недостатки. `Modifier.composed` требует передачи composable-лямбды, внутри которой генерируется restartable-группа и множество других конструкций. Во время композиции под капотом вызываются эти composable-лямбды для получения конечного модификатора. Этот процесс называется материализацией, что усложняет начальную композицию. Кроме этого из-за лямбд страдало и сравнение модификаторов, так как каждый раз создаётся новая лямбда, которая не равна прошлой, следовательно цепочка модификаторов тоже считается другой из-за чего Compose делает меньше пропусков.

Теперь на помощь приходит `Modifier.Node` – новый, более эффективный способ создания кастомных модификаторов. Этот подход позволяет реализовать все задачи, для которых ранее использовался `Modifier.composed`, но без лишних накладных расходов, связанных с composable-лямбдами и композицией. Самое важное преимущество – классы, созданные с помощью `Modifier.Node`, могут быть сравнены и переиспользованы, что значительно улучшает производительность.

Google недавно обновила документацию, добавив множество примеров и рекомендаций по использованию `Modifier.Node`. Это отличный повод пересмотреть существующие реализации модификаторов в вашем проекте и заменить их на более оптимизированные `Modifier.Node`.

Проблемы DerivedState и remember

Как мы уже выяснили, композиция сама по себе дорогая и стоит использовать `DerivedState`, чтобы ограничить число рекомпозиций. Однако часто разработчики применяют его некорректно, что было подробно разобрано в первой части. Сейчас стоит объяснить, почему так важно следить за этим.

Рассмотрим время, затрачиваемое на чтение тех или иных данных, чтобы понять масштабы проблемы (источник):

- Чтение локальной переменной - 1 нс
- Чтение из поля класса - 5 нс
- Вызов метода - 10 нс
- Чтение с синхронизацией - 50 нс
- map.get - 150 нс

- state.value - 2 500 нс
- derivedState.value - 10 000 нс

Учитывая, что для поддержания частоты обновления экрана в 120 кадров в секунду максимальное время отрисовки одного кадра не должно превышать 8 333 333 нс, становится очевидным, что неправильное использование `DerivedState` может значительно замедлить ваше приложение. Если такой подход не приводит к заметному уменьшению числа рекомпозиций, это может сделать код медленнее, чем при прямом обращении к исходному State<T>.

Та же логика применима и к функции `remember { }`. Необдуманное "запоминание" простых вычислений может неоправданно замедлить работу приложения. Например, чтобы запомнить обычное выражение, потребуется сперва сравнить ключи с их предыдущими значениями, что уже само по себе будет дороже, и это не говоря про чтение и запись в Slot Table.

```
// Антипример
val expr = remember(a, b, c) { a + b * c }
```

Поэтому и нужно хоть немного знать, как Compose работает под капотом, чтобы не использовать его функции себе во вред.

Пересядь с иглы Compose на старый добрый Kotlin

Для эффективной работы с Jetpack Compose ключевым является умение находить золотую середину между использованием возможностей Compose и стандартным Kotlin кодом. Как отметил разработчик Compose Андрей Шиков: "Изучая Compose, мы позабыли как использовать Kotlin". Давайте взглянем на его пример, демонстрирующий этот подход. В исходной версии кода использовалось множество отдельных состояний и сайд-эффектов:

```
@Composable
fun MyComponent(modifier: Modifier, config: Config) {
    val interactionSource = remember { MutableInteractionSource() }
    val activeInteractions = remember { mutableStateListOf<Interaction>() }
    val config by rememberUpdatedState(config)

    LaunchedEffect(interaction Source) {
        interactionSource.interactions.collect {
            // Обновление списка взаимодействий
        }
    }

    val animatableColor = remember {
        Animatable(config.defaultColor, Color.VectorConverter)
    }

    LaunchedEffect(config) {
```

```
        // Обновление state.animatable
        // Обновление state.config
    }

    LaunchedEffect(interactions) {
        snapshotFlow { activeInteractions.lastOrNull() }.collect {
            // Обновление animatableColor на основе взаимодействий и конфига
        }
    }

    // Использование animatableColor при отрисовке
}
```

Этот код был оптимизирован путём объединения нескольких состояний и сокращения количества сайд-эффектов за счёт переноса логики в одну корутину:

```
@Composable
fun MyComponent(modifier: Modifier, config: Config) {
    val state = remember { MyComponentState(config) }

    LaunchedEffect(state) {
        state.collectUpdates()
    }

    SideEffect {
        state.config.value = config
    }

    // Использование state.animatableColor при отрисовке
}
```

Благодаря такому рефакторингу удалось ускорить работу кода на **9%**.

Пререндер

При первом открытии экрана приложения, время ожидания пользователя состоит из двух основных этапов: загрузки данных и отрисовки UI. Для минимизации времени ожидания появления контента на экране можно использовать технику пререндера.

Суть метода заключается в том, что во время загрузки данных параллельно происходит отрисовка экрана с использованием моковых данных. Очень важно, чтобы структура композиции UI не менялась сильно после замены моковых данных на реальные. Учитывая высокую стоимость процесса композиции, такой подход позволяет заранее "прогреть" композицию, существенно сокращая время до появления актуального контента на экране.

Один из вариантов реализации пререндеринга – использование индикатора загрузки, отрисованного поверх всего экрана:

```
ProductDetailScreen(state: ProductUiState) {
    // Вначале state содержит пустые данные для пререндера контента
    ProductDetailContent(state)

    // Отображение индикатора загрузки поверх контента, а не вместо него
    if (state.isLoading) {
        FullscreenLoader()
    }
}
```

Также можно применять эффект шиммера на моковых данных, используя модификатор `placeholder` из библиотеки accompanist. Этот метод не влияет на структуру композиции после загрузки реальных данных и обеспечивает плавный визуальный переход. Однако, он требует дополнительной адаптации существующих элементов для корректной отрисовки шиммера поверх контента.

В контексте использования ленивых списков особое внимание следует уделить однородности элементов с одинаковым `contentType`. Если элементы списка не будут сильно отличаться по структуре композиции, это снизит необходимость в дополнительной работе по перестройке дерева композиции при их переиспользовании, что также способствует ускорению скролла.

Отложенная композиция

Ленивые лейауты используют `SubcomposeLayout` для определения того, какие элементы должны быть отображены на экране. Для этого происходит внутренняя композиция во время фазы компоновки (layout). Этот процесс происходит за один кадр, что может стать проблемой при работе с тяжёлыми экранами.

В таких случаях может быть полезной техника отложенной композиции, которая позволяет распределить процесс композиции элементов по нескольким кадрам, улучшая тем самым скорость отрисовки кадров. О применении данного метода можно почитать в статье по ссылке.

В примере ниже демонстрируется как отложить композицию части экрана, используя метод `withFrameNanos { }`, который аналогично `delay()` останавливает корутины, но делает это ровно до начала следующего кадра:

```
@Composable
fun ProductDetail(
    productInfo: ProductInfo
) {
    var blockState by remember { mutableStateOf(0) }

    LaunchedEffect(Unit) {
        while (blockState < 3) {
            // Откладываем композицию каждого блока на 1 кадр
            withFrameNanos {  }
```

```
                blockState += 1
            }
        }

        ProductBlock0(productInfo)

        if (blockState >= 1) {
            ProductBlock1(productInfo)
        }
        if (blockState >= 2) {
            ProductBlock2(productInfo)
        }
        if (blockState >= 3) {
            ProductBlock3(productInfo)
        }
    }
```

Painter

Xml-иконки vs Compose-иконки

При работе с иконками в Compose существует два основных подхода: использование иконок в формате XML и прямое создание иконок с помощью кода. Рассмотрим процесс и эффективность каждого из них.

```
// Xml-иконка
Image(
    painter = painterResource(R.drawable.my_icon),
    contentDescription = null
)

// Compose-иконка
Image(
    painter = rememberVectorPainter(Icons.Filled.Home),
    contentDescription = null
)
```

Для XML иконки процесс загрузки включает вызов `painterResource(R.drawable.my_icon)` , состоящий из следующих шагов:

1. Чтение ресурса из файла или получение его из кэша.
2. Преобразования XML в `ImageVector` .
3. Создание `Painter` с помощью функции `rememberVectorPainter()` .

В случае с иконками, созданными непосредственно в Compose, процесс выглядит следующим образом:

1. Вызов `Icons.Filled.Home` сразу инициирует создание объекта `ImageVector`.
2. Создание `Painter` с помощью функции `rememberVectorPainter()`.

Тесты показали, что иконки, созданные с помощью Compose, загружаются от 5% до 18% быстрее по сравнению с иконками в формате XML. Скорость загрузки напрямую зависит от сложности структуры иконки и размера исходного файла. Важно отметить, что общее влияние иконок на производительность приложения может сильно варьироваться в зависимости от их количества на экране и использования в ленивых списках.

Для создания собственных Compose-иконок можно воспользоваться инструментом SVG to Compose, который поддерживает конвертацию как SVG, так и XML файлов. Также стоит упомянуть, что эти иконки убавят вам проблем с ресурсами при использовании Compose Multiplatform.

Нестабильность Painter

В этом параграфе предполагалось обсудить, как нестабильность `Painter` влияет на производительность отображения изображений и иконок и в каких случаях целесообразно использовать обёртку для `Painter`. Однако, с внедрением возможности объявлять внешние типы как стабильные, данный вопрос теряет актуальность. Об этом в главе про нововведения.

Вынос Painter

Вынесение создания объекта `Painter` за пределы списка позволяет заметно увеличить скорость отрисовки. Это особенно критично в списках, где каждый элемент требует инициализации собственного `Painter`. Несмотря на наличие кэша для XML ресурсов, каждый такой вызов создает дополнительную нагрузку. Создание единого `Painter` для всего списка значительно уменьшает эту нагрузку. Однако, как и в случае с выносом модификаторов, может пострадать читабельность кода.

Пример до оптимизации:

```
@Composable
fun MyList(products: ImmutableList<ProductUiState>) {
    LazyColumn {
        items(products) { product ->
            // Painter внутри элемента списка
            MyProductItem(product, painterResource(R.drawable.ic_menu))
        }
    }
}
```

Пример после оптимизации:

```
@Composable
fun MyList(products: ImmutableList<ProductUiState>) {
    // Выносим Painter для общей иконки из списка
    val menuPainter = painterResource(R.drawable.ic_menu)

    LazyColumn {
        items(products) { product ->
            MyProductItem(product, menuPainter)
        }
    }
}
```

Этот метод эффективен не только для иконок, но и для других ресурсов, хотя для последних прирост производительности может быть менее заметен. Подход актуален не только для списков, но и для мест с частой рекомпозицией из-за анимаций, где рекомендуется вообще избегать создания дорогих объектов.

Дизайн система

Создание дизайн-системы является ключевым этапом при внедрении Jetpack Compose в проекты. Разработчики на этом этапе могут столкнуться с проблемами из-за недостаточного опыта разработки на новой технологии, что ведет к появлению неэффективного кода в важнейших частях дизайн-системы.

Цветовая схема

Традиционно при создании кастомной цветовой палитры разработчики копируют подход, используемый в `MaterialTheme`. Однако недавние коммиты разработчиков указывают на недостатки такой реализации.

В традиционной реализации для каждого цвета создавался отдельный `State<T>`, что вело к необходимости подписки на изменение состояния каждый раз при его чтении. Это могло негативно повлиять на производительность, особенно если в дизайн-системе использовалось множество цветов.

Пример до оптимизации:

```
@Stable
class ColorScheme(
    primary: Color,
    onPrimary: Color,
) {
    // State<T> для каждого цвета
    var primary by mutableStateOf(primary)
        internal set
    var onPrimary by mutableStateOf(onPrimary)
```

```
        internal set
}
```

Изначально такая реализация имела преимущество в гибкости — возможность изменять каждый цвет отдельно без значительных затрат на производительность. Однако, как показывает практика, в большинстве приложений цветовая схема меняется лишь при переключении между светлой и темной темами, а не при индивидуальном изменении отдельных цветов.

Переход к использованию обычного `data class` для описания цветовой схемы устраняет необходимость в подписках на состояние и улучшает производительность приложения.

Пример после оптимизации:

```
@Immutable
data class ColorScheme(
    val primary: Color,
    val onPrimary: Color,
)
```

Ошибки прошлого

Код, о котором будет идти речь, был написан давно и с тех пор не подвергался изменениям. Однако именно он использовался на всех наших экранах и существенно влиял на время отрисовки кадров из-за неэффективной работы с типографией в `AppTheme`.

`AppTheme.typography` создаёт новую типографию при каждом вызове, загружая шрифты и цвета из ресурсов для каждого стиля текста отдельно. Это приводило к множественным обращениям к ресурсам и чтению `AppTheme.colors.textPrimary` 16 раз для каждого `TextStyle`.

```
object AppTheme {
    @Composable
    @ReadonlyComposable
    val typography
        get() = DefaultAppTypography
}

@Composable
@ReadonlyComposable
val DefaultAppTypography
    get() = AppTypography(
        headXXL = TextStyle(
            color = AppTheme.colors.textPrimary,
            ...
        ),
```

```
        // ...
        // Создание 15 других стилей текста
    )
```

Эта реализация особенно заметно замедляла отрисовку на экранах с большим количеством текста и различными стилями, поскольку каждый текстовый элемент инициировал вызов условного `AppTheme.typography.headXXL`. Нашли мы эту проблему после исследования нескольких экранов с помощью трассировки композиции.

Решение проблемы заключалась в изменении подхода к созданию и использованию типографии. Теперь типография `AppTheme` инициализируется единожды и доступна через `LocalAppTypography.current`, что значительно сокращает количество обращений к ресурсам и ускоряет работу с типографией во всем приложении:

```
object AppTheme {
    @Composable
    @ReadonlyComposable
    val typography
        get() = LocalAppTypography.current
}
```

Форматтеры

Важно помнить о правильном размещении логики форматирования чисел и валют. Интуитивно может показаться, что использование форматирования непосредственно в коде Compose — это удобно. Однако, этот подход может привести к неожиданным проблемам производительности. Рекомендуется переносить создание и использование форматтеров в бизнес-логику вашего приложения. Такой шаг позволяет не только уменьшить нагрузку на главный поток, но и избежать излишнего дублирования объектов форматтера, обеспечивая их переиспользование между экранами. Этот совет кажется простым, но на практике часто упускается из виду, особенно когда форматирование спрятано за утилитарные функции.

Антипример форматирования в Compose-коде:

```
Text(
    text = productItem.price.toMoneyFormat()
)
```

Однако, необходимо учитывать, что оптимизация за счёт переноса форматирования может не всегда давать ожидаемый результат. В случае обработки списка в бизнес-логике, форматирование применяется ко всему списку сразу, в то время как в Compose, в контексте ленивых списков, форматирование выполняется исключительно для элементов, видимых пользователю. Это означает, что перемещение логики форматирования может потенциально

увеличить время до отображения содержимого на экране. Важно помнить, что не каждая оптимизация ведёт к улучшению производительности, и нужно тщательно анализировать изменения, прежде чем их внедрять

Оптимизация на спичках

Эти рекомендации пригодятся вам при разработке утилитарных функций, основных компонентов приложения или при создании дизайн-системы. Особенно актуально это становится при работе с анимациями и графикой. При написании кода, который исполняется в фоновом потоке и не используется повсеместно, стремление к использованию структур, более эффективных, чем стандартные, может оказаться излишним. Важно помнить, что оптимизация алгоритмической сложности даст более заметный прирост в производительности, чем экономия на спичках.

Автоупаковка

Автоупаковка примитивных типов данных может незаметно замедлить выполнение кода, особенно когда он используется в большом количестве мест. В Jetpack Compose разработчики активно стремятся минимизировать подобные затраты, применяя различные способы:

- Использование специальных `MutableState` для примитивных типов (например, `mutableIntStateOf()`) помогает избежать упаковки.
- Введение специальных значений (`Unspecified`) для определенных классов вместо `null` позволяет избежать автоупаковки благодаря инлайнингу value классов. Например, недавно добавили Unspecified для `TextAlign` , `TextDirection` и др., чтобы избежать `null` .
- Замена `Pair<Int, Int>` на `value class` с полем типа `Long` значительно снижает затраты на хранение данных, используя стек вместо кучи. Тип `Long` содержит в два раза больше бит, чем `Int` , что позволяет ему хранить два числа типа `Int` и обращаться к ним с помощью побитовых операций.

Быстрые методы

Не все стандартные методы Kotlin идеально подходят для конкретных задач. Jetpack Compose предлагает альтернативы, например, `fastForEach` или `fastFirst` . О быстрых методах можно прочитать в блоге Romain Guy.

Эффективные структуры

Стандартные структуры данных в Kotlin могут быть не лучшим выбором для специализированных задач.

- Например, `mutableListOf` использует `ArrayList` , что может быть избыточным, если не требуется динамическое изменение размера коллекции или использование дженериков. В критически важных частях кода лучше применять специализированные массивы (например, `IntArray`).

- Метод `mutableMapOf` по умолчанию создаёт `LinkedHashMap`, что может быть менее эффективным, чем другие типы данных. Jetpack Compose использует, например, `ScatterMap`.

Внутри AndroidX Collections много оптимизированных структур, которые стоит использовать в критическом коде.

Вложенные if

При работе с условными конструкциями важно учитывать влияние вложенности на производительность. Каждая условная ветка создаёт дополнительные вызовы replaceable-группы для поддержки быстрой замены кода. Поэтому в случаях, когда вложенность условных блоков не является необходимостью, предпочтение следует отдавать плоской структуре. Это позволяет не только упростить код, но и повысить его производительность за счёт уменьшения количества операций.

Пример генерации кода для плоских if (также и для when):

```
if (condition1) {
    $composer.startReplaceableGroup()
    Content1()
    $composer.endReplaceableGroup()
} else if (condition2) {
    $composer.startReplaceableGroup()
    Content2()
    $composer.endReplaceableGroup()
} else {
    $composer.startReplaceableGroup()
    Content3()
    $composer.endReplaceableGroup()
}
```

Пример генерации кода для вложенных if:

```
if (condition1) {
    $composer.startReplaceableGroup()
    Content1()
    $composer.endReplaceableGroup()
} else {
    $composer.startReplaceableGroup()
    if (condition2) {
        $composer.startReplaceableGroup()
        Content2()
        $composer.endReplaceableGroup()
    } else {
        $composer.startReplaceableGroup()
        Content3()
```

```
            $composer.endReplaceableGroup()
        }
        $composer.endReplaceableGroup()
    }
```

Нововведения

Указание стабильности внешних типов

С релизом Compose Compiler версии 1.5.5 появилась возможность явно указывать стабильность внешних типов. Это нововведение позволяет избежать использования дополнительных обёрток для обеспечения стабильности. Рекомендуем добавить в список стабильных типов такие часто используемые классы, как стандартные коллекции из Kotlin и Painter. Это нужно указать в каждом модуле, где используется Compose. Указание коллекций особенно актуально для тех, кто предпочитает не использовать `immutable` коллекции из-за нахождения их в альфа-версии или из-за необходимости переписывания большого объема кода.

```
// Все коллекции из котлин
kotlin.collections.*

// Painter
androidx.compose.ui.graphics.painter.Painter
```

Режим сильной пропускаемости

На данный момент режим сильной пропускаемости является экспериментальным и активируется через специальный флаг. В перспективе он может стать настройкой по умолчанию. Что он даёт:

- Все перезапускаемые (restartable) функции станут пропускаемыми (skippable). Для нестабильных параметров сравнение по экземплярам, для стабильных - через `equals`.
- Лямбды, захватывающие нестабильные переменные, будут тоже обёрнуты в `remember`.

Инструментарий

Просмотр исходного кода Compose

Каждый раз, когда вы сомневаетесь в том, как будет работать тот или иной код после компилятора Compose, стоит просто посмотреть финальный Java код. Хоть вы и можете отлично знать генерацию Compose-кода из Jetpack Compose Internals или множества статей, это не спасёт вас от устаревания информации там. Для этого есть удобный gradle-плагин - decomposer.

Vkompose плагин

Среди полезных инструментов есть плагины от VK для Kotlin и IDE, которые выполняют:

- Подсветку нестабильных параметров и непропускаемых функций непосредственно в IDE.
- Визуальное выделение происходящих рекомпозиций в UI с помощью цветных границ.
- Логирование причин, по которым произошла рекомпозиция.

Detekt

Detekt, позволяет не только следить за стилем кода, но и защищать от не очень хороших практик, в том числе приводящих к проседанию производительности.

- compose-rules - большой список разнообразных правил.
- vkcompose - кроме плагина предоставляет правило, которое проверяет функции на пропускаемость.

Итог

Подводя итог, мы разобрались, как избегать проблемы с начальной композицией и что не стоит перегружать Compose лишней логикой, ведь за удобство и простоту приходится платить. Однако благодаря неустанной работе разработчиков, производительность Compose значительно выросла, давая нам свободу сосредоточиться на других аспектах разработки. А экраны с огромным DAU и на View приходилось оптимизировать за гранью обычных приёмов. Для Compose это далеко не предел: будущие оптимизации сделают его ещё более мощным инструментом, идеально подходящим для любых задач.

Теги: android, android development, compose, kotlin, jetpack compose, jetpack, мобильная разработка, оптимизация кода, производительность, performance optimization

Хабы: Программирование, Разработка мобильных приложений, Android, Kotlin, Jetpack Compose

Комментарии 14 +14

Публикации

ЛУЧШИЕ ЗА СУТКИ ПОХОЖИЕ

vital_pavlenko вчера в 02:59

Синдром бога: когда разработчик ждёт миллионы и поклонения просто за то, что пишет код

2 мин 16K

Мнение

+107 40 88 +88

antoshkka вчера в 10:00

Встреча ISO C++ в Софии: C++26 и рефлексия

9 мин 4.2K

+46 14 38 +38

ntsaplin вчера в 10:30

Цены на дата-центры растут, а ИИ может сдристнуть в Казахстан

7 мин 2.5K

+32 12 5 +5

DRoman0v вчера в 11:02

Acer Switch One 10: как я спас необычный планшет-трансформер с барахолки. Что это за устройство?

4 мин 1.6K

+26 3 2 +2

klauss_z вчера в 11:44

ИИ-помощник редактора на Хабре: семь раз вайб-код — один раз поймешь

Простой 18 мин 784

Туториал

+22 8 4 +4

net0pyr вчера в 16:30

Красивый GitLab CI: extends, якоря, include, trigger

Средний 9 мин 1.5K

Туториал

+20 33 1 +1

vsradkevich вчера в 01:15

AGI уже здесь

Простой 5 мин 5.2K

Мнение

+20 24 35 +35

virtual_explorer вчера в 13:44

Океан в качестве аккумулятора: как гигантские подводные шары могут помочь с сохранением энергии

5 мин 1.9K

+18 13 21 +21

easyprotech вчера в 10:13

Синдром Бога vs. Реальные Боги

Простой 2 мин 1.5K

Мнение

+16 13 5 +5

full_moon вчера в 13:07

Цукерберг переманивает сотрудников OpenAI, модели учатся шантажу: главные события июня в ИИ

22 мин 592

Дайджест

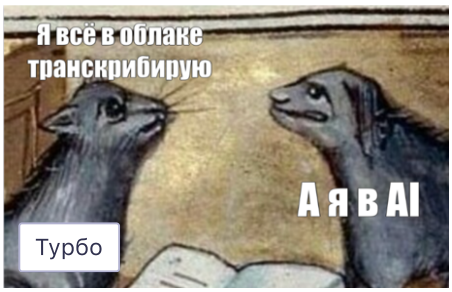
+15 3 0

Исследование: как и где ML-специалисты используют ИИ. Расскажите о своем опыте

Опрос

Показать еще

МИНУТОЧКУ ВНИМАНИЯ



Турбо

Whisper, Gemma и Obsidian: расшифровка аудио без облаков

Промо

Новый бонус к лету — освежающие скидки

Опрос

От экспериментов до продакшена: где вы применяете ИИ?

ВОПРОСЫ И ОТВЕТЫ

Android TV заражен вирусом. Что можно сделать?

Android · Простой · 1 ответ

Почему аргумент не видит функцию, как тип функция?

Kotlin · Простой · 1 ответ

Почему в лямба функции нельзя называть аргументы любым названием?

Kotlin · Простой · 2 ответа

Как присвоить переменной 2 типа данных?

Kotlin · Простой · 1 ответ

Как заставить снова работать камеры на Samsung Fold 3?

Android · Простой · 1 ответ

Больше вопросов на Хабр Q&A

ЧИТАЮТ СЕЙЧАС

Нейробиология восприятия: почему мы никогда не увидим мир «глазами» животного

152K 147 +147

В айти нет денег и повышений

60K 573 +573

Почему я не продаю мед, но зарабатываю на пчелах 3,6 млн в сезон

49K 47 +47

Я брала на себя слишком много лишних задач, пока не сделала это

8.7K 13 +13

Синдром бога: когда разработчик ждёт миллионы и поклонения просто за то, что пишет код

16K 88 +88

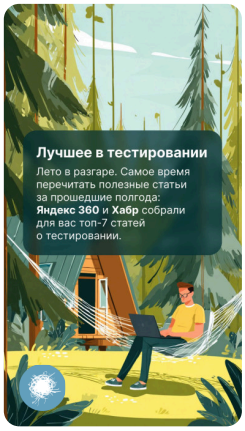
Исследование: как и где ML-специалисты используют ИИ. Расскажите о своем опыте

Опрос

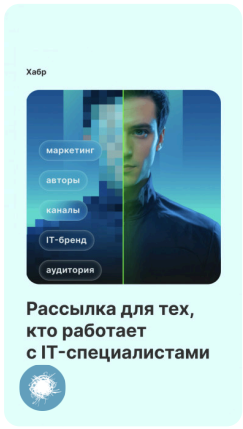
ИСТОРИИ

Годнота из блогов компаний

Кто победил в сезоне Open source?



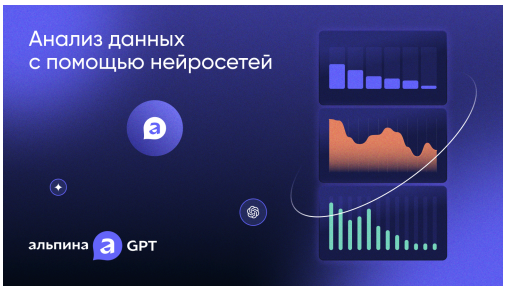
Лучшее в тестировании за полгода



Запустили авторские письма от Хабра

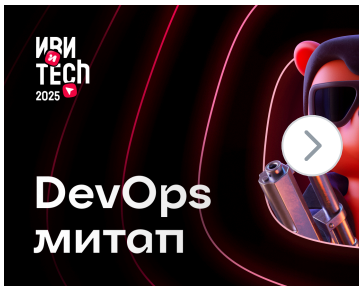
Жизнь замечательных животных

БЛИЖАЙШИЕ СОБЫТИЯ



1 июля

3 июля



3 июля

Вебинар «Анализ данных с помощью нейросетей»

Онлайн

Аналитика

Больше событий в календаре

Приглашаем на Digital-регату 2025!

Москва

Маркетинг

Больше событий в календаре

Митап «Три кита D маршрутизация, мониторинг и безопасность»

Москва • Онлайн