275.75
Рейтинг

Ozon Tech

Команда разработки ведущего e-com в России

Подписаться

tipapro 23 июн 2023 в 07:01

Осознанная оптимизация Compose

Средний 29 мин 63K

Блог компании Ozon Tech, Программирование*, Разработка мобильных приложений*, Android*, Kotlin*

Туториал ✏️ Технотекст 2023

Jetpack Compose — относительно молодая технология написания декларативного UI. Множество разработчиков даже не предполагают, что пишут неоптимальный код в такой критически важной части, и впоследствии это приводит к неожиданной низкой производительности и проседании метрик.

Наша команда Ozon Seller также столкнулась с этой проблемой. Мы решили собрать воедино все советы и наработки по написанию оптимизированного Compose-кода. Активное применение этих советов при оптимизации существующих экранов и написании новых существенно улучшило наши метрики: длительность лага по отношению к длительности скролла (hitch rate; чем меньше, тем лучше) экранов со списками упала в среднем с 15-19 % до 5-7 % (на 90-м перцентиле). Все эти советы и наработки мы описали в этой статье. Она будет полезна и начинающим, и опытным разработчикам, в ней подробно описаны оптимизации и механизмы Compose, а также рассказано про слабо задокументированные особенности и исправления ошибок, которые есть в других статьях. Давайте же начнём.

Серия статей:

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

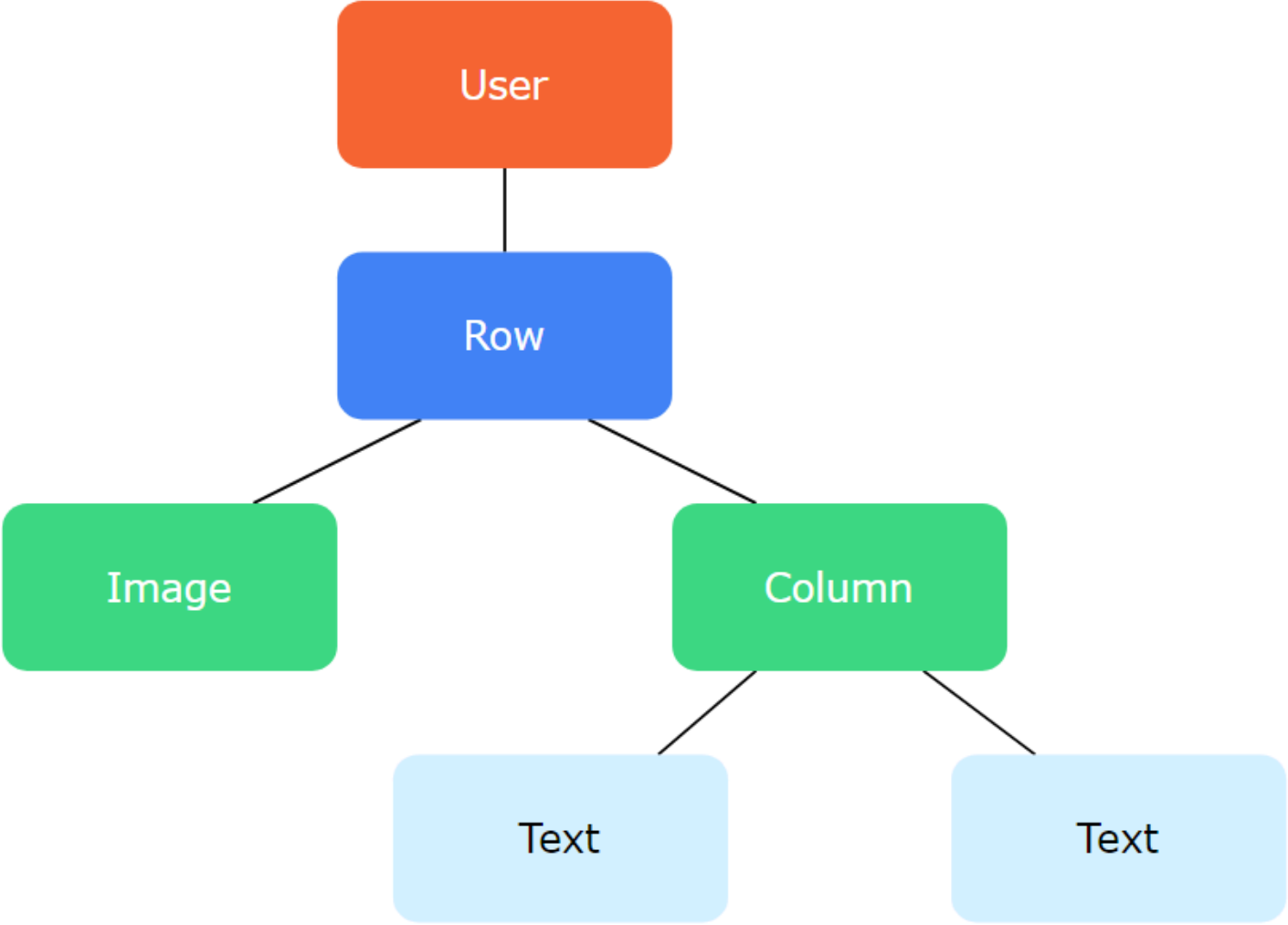
Composable-функции

Вначале погрузимся немного в работу Compose и его особенности, это и поможет нам понять, почему необходимы конкретные оптимизации и как они работают.

Основная идея

Построение UI-дерева — это основная идея composable-функций. Пройдя от начала до конца функцию User(), мы получим дерево как на картинке:

```
@Composable
fun User() {
    Row {
        Image()
        Column {
            Text()
            Text()
        }
    }
}
```



Чтобы строить такие деревья, нужно нечто большее, чем просто декларативный код. Об этом за нас беспокоится компилятор Compose.

Компилятор Compose — это плагин для компилятора Kotlin. А это значит, что, в отличие от плагина для kapt/ksp, он может модифицировать текущий код, а не просто генерировать новый. Во время компиляции он заменяет composable-функции новыми, в которые он добавляет вспомогательные конструкции и параметры, среди которых особенно важен `$composer`. Его можно воспринимать как контекст вызова. А процесс преобразования composable-функции можно представлять как то, что выполняет сам Kotlin с suspend-функциями.

Компилятор Compose добавляет вызовы методов `$composer` в начале и в конце сгенерированной composable-функции (см. код ниже). Эти методы начинают и заканчивают группу, которую можно представить как узел дерева, которое строит `Compose`. То есть начало и конец функции — это начало и конец описания узла. Слово `Restart` говорит о типе группы. В статье мы не будем глубоко погружаться в типы групп, но если интересно, то можно почитать об этом в книге “Jetpack Compose internals” (Глава 2. “The Compose compiler”, параграф “Control flow group generation”).

```
@Composable
fun User($composer: Composer) {
    $composer.startRestartGroup() // Начало группы

    // Тело функции
```

```
    $composer.endRestartGroup() // Конец группы
}
```

На основе данных из тела функции шаг за шагом `$composer` строит дерево. Это первая фаза Compose — **Composition**.

Фазы Compose

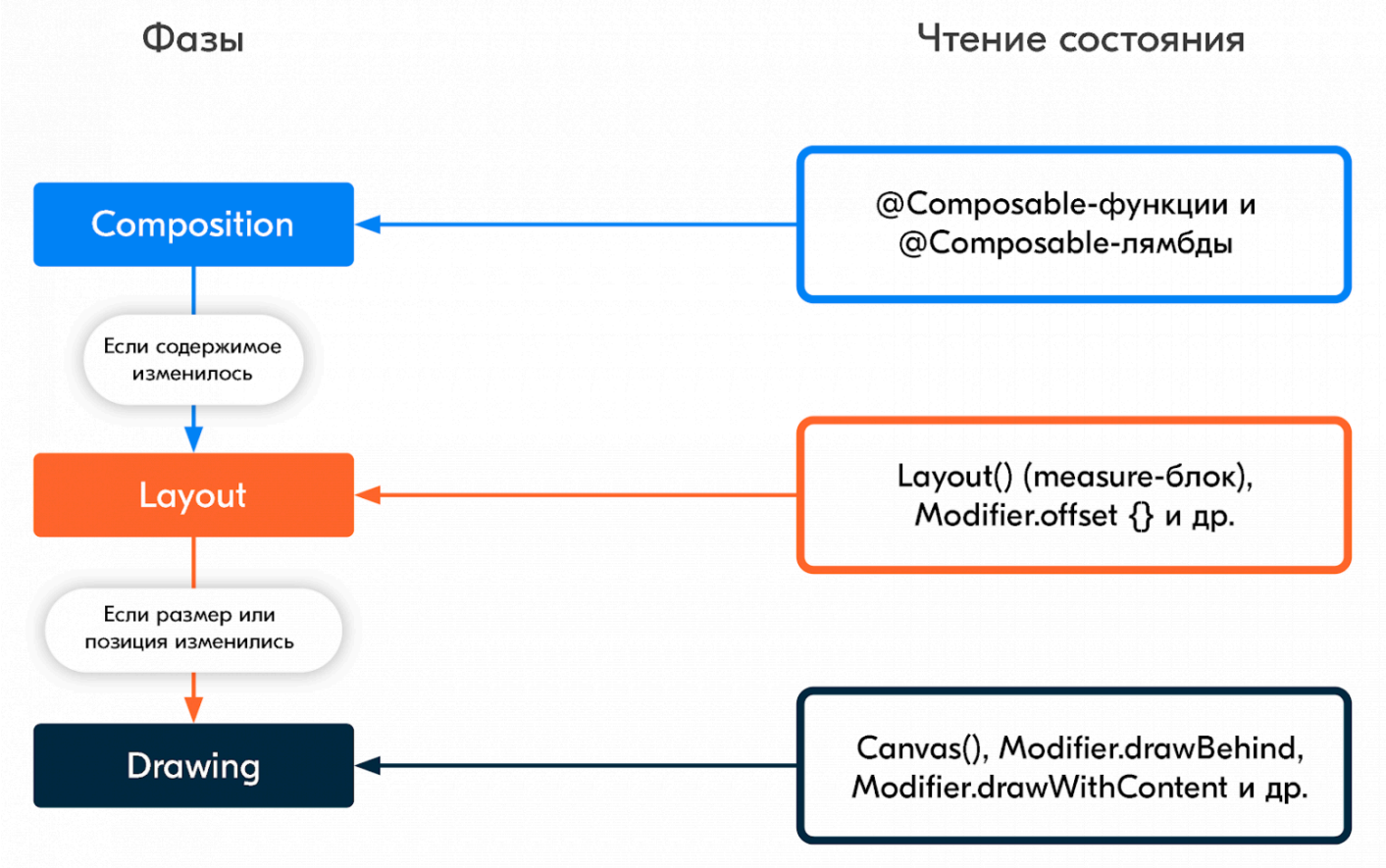
Как и большинство других UI-инструментариев, Compose отрисовывает кадр за несколько отдельных фаз.

Если мы посмотрим на систему Android View, то она имеет три основные фазы: измерение (measure), компоновка (layout) и отрисовка (drawing). У Compose фазы похожи:

1. **Composition**: какой UI отобразить. Compose запускает composable-функции и создаёт описание UI.
2. **Layout**: как расположить UI. Состоит из двух шагов: измерение (measurement) и размещение (placement). Элементы лейаута измеряют и располагают себя и свои дочерние элементы в двумерных координатах.
3. **Drawing**: как отрисовать UI. Элементы отрисовывают себя на Canvas.

Эти три фазы выполняются практически для каждого кадра, но для повышения производительности Compose может пропускать некоторые фазы, если данные для них не поменялись.

Условия вызова фаз, а также примеры мест чтения состояния можно увидеть на картинке ниже. Что такое чтение состояния разберём позже, пока что просто представьте себе это как получение значения `muatbleStateOf()` . Подробнее о фазах можно прочитать на Android Developers.



Аргументы в composable-функциях

Compose побуждает нас писать чистые функции. Это делает их более детерминированными, а также позволяет разработчикам Compose сделать первую оптимизацию — попросту не выполнять composable-функцию, если аргументы не поменялись.

Введём сразу такие понятия, как **композиция (composition)** — построение дерева composable-функций, и **рекомпозиция (recomposition)** — обновление этого дерева при изменении данных.

Мы подошли к ещё одному параметру, который добавляет компилятор Compose в composable-функции — `$changed`. Это просто число типа Int, которое является битовой картой, в которой биты отвечают за информацию об аргументах composable-функции, их изменении.

```
// Аргументы Composable-функции после работы компилятора Compose
@Composable
fun Header(text: String, $composer: Composer<*>, $changed: Int)
```

Если в родительской composable-функции изменились некоторые параметры, а некоторые остались прежние, то информация о сравнении передаётся дочерним функциям через параметр `$changed`, чтобы они не делали лишних сравнений. Сами функции сравнивают только те аргументы, в которых не уверен родитель, или если аргументы установлены по умолчанию.

Убить весь смысл сравнения могут мутабельные аргументы — объекты, которые способны изменяться (изменять свои данные). Чтобы решить эту проблему, разработчики Compose

решили разделить все типы на стабильные и нестабильные. Если все аргументы функции стабильные и не изменились, то рекомпозиция пропускается, иначе придётся перезапускать эту функцию снова.

Функция, поддерживающая пропуски рекомпозиции, называется **пропускаемой (skippable)**. Мы должны стараться, чтобы почти все наши функции были пропускаемыми. Это очень хорошо повлияет на оптимизацию.

Стабильные типы

Классификация типов по стабильности

Компилятор Compose проходит по всем типам и добавляет в них информацию об их стабильности: аннотацию `@StabilityInferred` и статическое поле `$stable` с информацией о стабильности типа.

Стабильность типа означает, что рантайм Compose может безопасно читать и сравнивать входные данные такого типа, чтобы при необходимости пропустить рекомпозицию. Конечная цель стабильности — помочь рантайму Compose.

Стабильными типами считаются:

- Все примитивные типы и String.
- Функциональные типы (лямбды) (поэтому понятие «нестабильные лямбды» не совсем корректно, но об этом ниже).
- Классы, у которых все поля стабильного типа и объявлены как val, в том числе и sealed-классы. Стабильность полей класса проверяется рекурсивно, пока не найдётся тип, о стабильности которого уже однозначно известно.
- Enum (даже если вы у него укажите поле var и будете его менять).
- Типы, помеченные `@Immutable` или `@Stable` .

Все стабильные типы должны выполнять определённый контракт, который мы затронем далее.

Compose считает **НЕстабильными**:

- Классы, у которых хотя бы одно поле нестабильного типа или объявлено как var.
- Все классы из внешних модулей и библиотек, в которых нет компилятора Compose (`List` , `Set` , `Map` и прочие коллекции, `LocalDate` , `LocalTime` , `Flow` ...);

У **дженериков** (`MyClass<T>`), проверка идёт по структуре самого дженерика, а уже потом по указанному типу. Если структура дженерика нестабильна (есть поля нестабильного типа или поля с var), то он сразу считается нестабильным. Если мы сразу указываем тип дженерика, то Compose уже на этапе компиляции определит его как стабильный или нестабильный:

```
// Стабильный
class MyClassStable(
    val counter: Pair<Int, Int>
)

// Нестабильный
class MyClassUnstable(
    val counter: Pair<LocalDate, LocalDate>
)

// Нестабильный
class MyClassUnstable(
    val counter: Pair<*, *>
)
```

Если мы делаем composable-дженерик-функцию и передаём дженерик ей в аргументах (`@Composable fun <T> Item(arg: Pair<T, T>)`), то поведение будет такое же, как и у типов с вычисляемой стабильностью, про которые расскажем дальше.

Разработчики Compose также **заранее определили внешние типы**, которые будут считаться стабильными: `Pair` , `Result` , `Comparator` , `ClosedRange` , коллекции из библиотеки kotlinx.collections.immutable, `dagger.Lazy` и другие. Большинство из этих типов — дженерики, поэтому данный список лишь говорит о стабильности их структуры. То есть, если мы передадим этим дженерикам стабильный тип, то и они будут стабильными, а если нестабильный, то и они будут нестабильными. Можно сказать, что к этим типам просто не будет применяться принцип, что все классы из внешних модулей и библиотек, в которых нет компилятора Compose, нестабильны.

Есть также **типы с вычисляемой стабильностью** — о которых Compose не может сказать при компиляции, что они однозначно стабильные или нестабильные. Их стабильность проверяется уже в рантайме, при получении конкретных объектов. К таким типам относятся:

- Типы, которые объявлены в других модулях с включённым компилятором Compose. Если мы в модуле 1 используем тип из модуля 2, в котором у нас не включён Compose, то этот тип компилятор Compose просто не сможет проверить на стабильность, поэтому сразу будет считать нестабильным. А если в модуле 2 включён Compose, то компилятор Compose предполагает, что он в модуле 2 проверит этот тип: проставит аннотацию `@StabilityInferred` и добавит статическое поле `$stable` . И уже потом, в рантайме, а не на этапе компиляции, он прочитает это поле.
- Интерфейсы (проверка идёт по типу-наследнику, объект которого будет передан в аргументы).

▸ Про стабильность интерфейсов:

Дополнительно узнать стабильность типов можно в тестах или в composable-метриках, про которые рассказано в главе про отладку.

@Immutable и @Stable

Если вы уверены, что класс или интерфейс и все его потомки стабильные, то можете пометить их аннотацией `@Immutable`, если они неизменны, или `@Stable`, если они могут меняться, но сами оповещают Compose о своём изменении. Например, `@Stable` подойдёт, если в классе есть поле типа `State<T>` или `MutableState<T>` (`mutableStateOf()` создаёт такой объект).

```
@Immutable
data class MyUiState1(val items: List<String>)

@Stable
data class MyUiState2(val timer: MutableState<Int>)
```

Стабильность от таких аннотаций наследуется дочерними типами.

```
@Immutable
interface Parent // Стабильный тип

class Child1(val age: Int) : Parent // Стабильный тип

class Child2(var list: List<String>) : Parent // Тоже стабильный тип
```

Аннотации `@Immutable` и `@Stable` полезны для того, чтобы их повесить на типы, которые Compose считает нестабильными, но по факту они стабильные, либо вы уверены, что будут использоваться как стабильные.

Обе аннотации на данный момент просто выполняют логику объявления стабильного типа и не отличаются друг от друга для Compose, но желательно всё-таки использовать их по предназначению, так как в будущем разработчики Compose могут изменить поведение.

Помечая этими аннотациями, вы обещаете Compose, что ваш тип будет выполнять следующий контракт:

1. `equals` всегда будет возвращать одинаковое значение для одной и той же пары объектов.
2. Когда публичные поля типа изменяются, нужно оповестить об этом Compose.
3. Все публичные поля стабильны.

Этот контракт — лишь ваше обещание, и Compose никак не проверит, если вы его нарушите. Но тогда возможно неожиданное поведение composable-функций. Давайте разберём контракт

подробнее.

Первый пункт особенно важен и может выстрелить в вас, даже если сделаете все аргументы стабильными. Например, в коде ниже мы видим, что у `MyUiState` нет переопределённого `equals`, как у data-класса, а это значит, что проверка будет происходить по ссылке. Если в `MyComposable1` произойдёт рекомпозиция, то `MyUiState` будет пересоздан. При проверке по ссылке Compose будет считать его совершенно другим объектом и не пропустит `MyComposable2`, хотя поле `name` осталось тем же самым.

```
class MyUiState(val name: String)

@Composable
fun MyComposable1() {
    val myState = MyUiState("Name")
    MyComposable2(myState)
}

@Composable
fun MyComposable2(uiState: MyUiState) {
    Text(uiState)
}
```

Такая ситуация решается либо написанием своей реализации `equals` (или использованием data class), либо запоминанием этого объекта с помощью `remember`, чтобы при рекомпозиции он не пересоздавался (или аналогичными действиями в бизнес-логике, если объект пересоздаётся там).

Второй пункт реализован в `State<T>` и `MutableState<T>` (`mutableStateOf`), который под капотом оповещает Compose при изменении.

Третий пункт контракта подразумевает, что вы используете публичные поля как стабильные. То есть, если у вас поле формально нестабильного типа `List<T>` и вы не кастите его где-нибудь к `MutableList<T>`, то смело помечайте ваш класс как `@Immutable` или `@Stable`.

`@Stable` можно повесить и на обычные (non-composable) функции и свойства. Тогда Compose будет считать, что они вернут то же значение, если аргументы не изменились. На composable-функции аннотация не влияет. В основном нужна для оптимизации генерируемого кода для аргументов по умолчанию в composable-функциях.

Пример функций и свойств, помеченных `@Stable`: `Modifier.padding()`, `Modifier.width()`, `Int.dp;` функции перехода в анимациях: `fadeIn()`, `fadeOut()`, `slideIn()`.

▸ Про влиянии @Stable на функции и свойства

Дополнительно про эти аннотации можно посмотреть в видео от red_mad_robot.

Пропускаемость функций

Compose делает composable-функцию пропускаемой, только если все её аргументы стабильного типа и функция возвращает `Unit`. При этом игнорируются нестабильные аргументы, если они не используются в теле.

Для пропускаемых функций Compose специально генерирует код, который позволяет не вызывать их снова, если входные данные не поменялись.

```
@Composable
fun Header(text: String, $composer: Composer<*>, $changed: Int) {
    if (/* Логика проверки на необходимость пропуска */) {
        Text(text) // Выполняется тело функции
    } else {
        $composer.skipToGroupEnd() // Сообщаем Compose, что мы пропустили функцию
    }
}
```

Нестабильные типы встречаются и среди часто используемых в Compose, например, тот же `Painter`, так что стоит осторожнее использовать его, чтобы не потерять пропускаемость у функции.

Если есть аргументы с вычисляемой в рантайме стабильностью, то функция остаётся пропускаемой, но дополнительно генерируется код, который не пропускает её, если в рантайме аргумент оказывается нестабильного типа.

Исходя из вышеперечисленного, мы в команде договорились помечать все UI-модели и состояния как `@Immutable` или `@Stable`, так как изначально их проектируем таковыми. Особенно следим за стабильностью при разработке UI kit-проекта, так как цена ошибки становится выше. Чтобы проверить стабильность типов, вы можете использовать метрики Compose (к ним вернёмся в конце статьи).

Также можно просто передавать в функции как можно меньше лишних данных. Тут всё просто: меньше данных — меньше вероятность, что они поменяются.

Что делать, если нужно использовать стандартные коллекции или внешние классы и хочется пропускаемости функций? Пока пространство возможного сильно ограничено: либо делать класс-обёртку (value class тоже можно использовать как обёртку) и вешать на него аннотации `Immutable` или `Stable`, либо попросту избегать. Для стандартных коллекций есть вариант перехода в UI-моделях на коллекции из kotlinx.collections.immutable. Возможность объявлять стабильность внешних типов у разработчиков Compose в планах.

Лямбды

Давайте поговорим о том, как работают лямбды в Compose и как правильно их готовить. В этой статье приводится интересный пример с вызовом метода `ViewModel` внутри лямбды, что приводит к лишним рекомпозициям.

Кратко эту ситуацию можно представить так:

```
@Composable
fun MyScreen() {
    val viewModel = remember { MyViewModel() }
    val state by viewModel.state.collectAsState()

    MyComposableItem(
        name = state.name,
        onButtonClick = { viewModel.onAction() }
    )
}
```

Чтобы понять, что и почему, давайте разберём, как же Compose обрабатывает лямбды. Он делит их на non-composable, в которых не выполняется composable-код, и composable соответственно. Рассмотрим подробно первый тип.

Non-composable лямбды, которые создаются в composable-функции, при компиляции оборачиваются в `remember`. Все захваченные переменные кладутся в качестве ключа для remember:

```
// До компиляции
val number: Int = 6
val lambda = { Log.d(TAG, "number = $number" }

// После компиляции
val number: Int = 6
val lambda = remember(number) { { Log.d(TAG, "number = $number" } }
```

Если лямбда захватывает в себя переменную, тип которой **НЕ стабильный** (то есть нестабильный или вычисляемый в рантайме: условие строже, чем для пропускаемости) или переменная объявлена как var, то Compose не оборачивает её в `remember`, из-за чего при рекомпозициях она пересоздаётся. Дальше при сравнении прошлой и текущей лямбды Compose обнаружит, что они не равны, и из-за этого начнёт рекомпозицию даже пропускаемой функции (подразумевается, что `MyViewModel` — нестабильным тип).

Как решить эту проблему? Раньше работало использование ссылки на метод (`viewModel::onAction`), но начиная с Compose 1.4 перестало работать из-за использования сравнения по ссылке вместо кастомного `equals`, который генерирует Kotlin. Подробнее можно почитать в этом треде, а также в этом видео с 32:50.

▸ Скрины из треда

Работают способы:

- Запоминать лямбду самим (при этом ключ должен и сам не меняться при каждой рекомпозиции):

```
val onAction = remember { { viewModel.onAction() } }
```

Можно сделать так для краткости (почему запоминать лямбду, а не ссылку на метод, можно почитать здесь):

```
@Composable
inline fun <T : Any> MviViewModel.rememberOnAction(): ((T) -> Unit) {
    return remember { { this.onAction(it) } }
}

val onAction = viewModel.remberOnAction()
```

- Использовать верхнеуровневые (статические) функции и переменные. Здесь компилятор Kotlin будет напрямую их вызывать, так как они статические, а не передавать через конструктор класса, который будет создан для лямбды на этапе компиляции.
- Использовать внутри лямбды только стабильные внешние переменные.
- Использовать аргументы, полученные внутри лямбды. Это может не помочь, а только отложить или уменьшить проблему, но точно поможет, если вы раньше захватывали список, а теперь не будете захватывать ничего.

```
@Composable
fun MyComposableItem(items: List<MyClass>) {
    // Вместо такого
    ItemWidget { items[5].doSomething() }

    // Делать так
    ItemWidget(item[5]) { item -> item.doSomething() }
}
```

Лямбда также может неявно захватить внешнюю переменную, если вы в composable-функции внутри фрагмента в лямбде вызовите функцию из этого фрагмента. Тогда конструктор лямбды будет принимать фрагмент как аргумент, и `remember` вокруг лямбды не будет генерироваться.

```
class MyFragment : Fragment {
    fun onButtonClick() { ... }

    @Composable
    fun Screen() {
        MyButton(onClick = { onButtonClick() })
    }
}
```

У вас также мог появиться вопрос, как с лямбдой справляется сам `remember { }`, если он принимает лямбду? Дело в том, что `remember` — это inline-функция, и её лямбда превращается в обычный блок кода. Так, функция:

```
val resultValue = remember(key1, key2) {
    // Наши вычисления (например, создание лямбды)
}
```

превратится в следующий код:

```
// Получение запомненного значения
val rememberedValue = composer.rememberedValue()

val needUpdate = /* Проверка на то, изменились ли наши ключи key1 и key2,
    или значение ещё не инициализировано */

if (needUpdate) {
    // Наши вычисления. Inline-лямбда превратится в блок кода
    val value = calculation()

    // Обновление запомненного значения
    composer.updateRememberedValue(value)

    return value // Возвращает вычисленное и запомненное значение
} else {
    return rememberedValue // Возвращает запомненное значение
}
```

Код выше лишь отражает логику и имеет упущения.

Дополнительно про лямбды под капотом в контексте Compose можно посмотреть в этом видео с 25 минуты.

Оптимизация пропусков

Перезапускаемые функции

Для начала давайте разберёмся, что представляют собой перезапускаемые composable-функции. Как говорилось выше, в начале и в конце функции `$composer` начинает и заканчивает группу — условно узел дерева. Для перезапускаемых (restartable) функций вызывается перезапускаемая группа:

```
@Composable
fun MyComposable($composer: Composer) {
    $composer.startRestartGroup() // Начало группы

    // Тело функции

    $composer.endRestartGroup() // Конец группы
        ?.updateScope { $composer ->
            MyComposable($composer)
        }
}
```

В конце кода можно увидеть механизм перезапуска функции при изменении: если между началом и концом группы было прочитано состояние, которое умеет оповещать Compose о своём изменении (`State<T>` или `CompositionLocal`), то `$composer.endRestartGroup()` вернёт не null и Compose научится перезапускать нашу функцию. Если есть более близкая к месту чтения состояния перезапускаемая группа, то перезапускаться будет именно она, а не внешняя.

Давайте рассмотрим этот код:

```
@Composable
fun MyComposable1() {
    val counter: MutableState<Int> = remember { mutableStateOf(0) }
    MyComposable2(counter)
}

@Composable
fun MyComposable2(counter: State<Int>) {
    Text(text = "My counter = ${counter.value}")
}
```

В нём при изменении `counter` будет перезапускаться только `MyComposable2` , так как именно в её области видимости читается значение. Тот же `MutableState` можно представить себе как `MutableStateFlow` , который под капотом при чтении и записи выполняет необходимую логику подписки и оповещения. Это очень важная логика работы

Compose, так как перезапустится именно `MyComposable2`, не трогая остальные родительские функции. Именно на этом основан механизм рекомпозиции. Вместе с механизмом пропусков это даёт широкие возможности для оптимизации, особенно для часто меняющихся частей UI.

Для закрепления главы вот ещё примеры, из-за которых `MyComposable2` будет точкой перезапуска (рекомпозиции) и пойдёт по всем её детям, а `MyComposable1` не будет затронута. Можно добавить, что функции `animateColorAsState()`, `rememberScrollState()` и пр. тоже внутри содержат `State<T>`, и могут стать причиной рекомпозиции при изменении.

```
val LocalContentAlpha = compositionLocalOf { 1f }

@Composable
fun MyComposable1() {
    val counter1: MutableState<Int> = remember { mutableStateOf(0) }
    var counter2: Int by remember { mutableStateOf(0) }
    MyComposable2(counter1, { counter2 })
}

@Composable
fun MyComposable2(counter1: State<Int>, counterProvider2: () -> Int) {
    Text("Counter = ${counter1.value}") // Чтение состояния
    Text("Counter = ${counterProvider2()}") // Чтение состояния
    Text("Counter = ${LocalContentAlpha.current}") // Чтение состояния
}
```

Обратите внимание, что если вы используете `State<T>` как делегат, то в этом случае будьте осторожнее со случайным чтением состояния, особенно если оно часто меняющееся.

```
@Composable
fun MyComposable1()
    var counter: Int by remember { mutableStateOf(0) }

    // Чтение состояния произойдёт в MyComposable1, а не в MyComposable2!!!
    MyComposable2(counter)
}
```

Разработчики Compose советуют прокидывать не `State<T>`, а лямбду, так как могут возникнуть трудности и лишний код, если нужно будет что-нибудь захардкодить или при тестировании. Но, в целом, кардинальных отличий нет. Зачем всё это нужно — расскажем в главе про отложенное чтение состояний.

Также стоит добавить, что composable-лямбды, часто используемые в Slot API, тоже перезапускаемы и пропускаемы.

Перезапускаемость и пропускаемость

Чтобы вы не запутались в этих двух терминах, резюмирую здесь:

- **Перезапускаемая (restartable) функция** может перезапускаться, быть областью перезапуска.
- **Пропускаемую (skippable) функцию** можно пропустить, если её аргументы не изменились.

Вот как Compose обозначает в своих метриках функции, которые и перезапускаемы, и пропускаемы:

```
restartable skippable scheme("[androidx.compose.ui.UiComposable]") fun MyWidget(
  stable widget: WidgetUiModel,
  stable modifier: Modifier? = @static Companion
)
```

Если хотя бы один аргумент нестабильный, то функция останется только перезапускаемой (restartable).

Если функция только перезапускаемая (restartable), то стоит либо сделать её и пропускаемой (skippable), либо избавиться от перезапускаемости. Аннотация `@NonRestartableComposable` как раз убирает перезапускаемость и (если была) пропускаемость.

Когда перезапускаемость и пропускаемость не нужны

Все inline composable-функции неперезапускаемы (`Box` , `Column` и `Row`). Это значит, что чтение `State<T>` внутри одной из них при изменении вызовет рекомпозицию в ближайшей внешней перезапускаемой функции.

```
@Composable
fun MyComposable() {
    val counter: MutableState<Int> = remember { mutableStateOf(0) }
    Box {
        // Рекомпозиция затронет всю MyComposable(), так как код заинлайнится
        Text(text = "My counter = ${counter.value}")
    }
}
```

Не пропускаемы и функции, которые возвращают не `Unit` .

Есть ситуации, когда пропускаемость и перезапускаемость не даёт реальных преимуществ, а только приводит к избыточной трате ресурсов:

- данные composable-функции меняются редко или никогда;
- composable-функция просто вызывает другие пропускаемые composable-функции:
  - функция без сложной логики и без `State<T>`, вызывающая минимум других composable-функций;
  - обёртка вокруг другой функции — служит неким маппером параметров или же для сокрытия ненужных параметров.

В таком случае можно пометить composable-функцию аннотацией `@NonRestartableComposable`, что уберёт перезапускаемость (а вместе с ней и пропускаемость).

```
@Composable
@NonRestartableComposable
fun ColumnScope.SpacerHeight(height: Dp) {
    Spacer(modifier = Modifier.height(height))
}
```

Если функция содержит в себе ветвление логики (`if`, `when`), то ориентируйтесь по вышеописанным правилам уже по отношению к её ветвям. Добавлять аннотацию или нет зависит от того, насколько часто ветви будут меняться во время использования и насколько сложный код в каждой из веток.

Как пример, `@NonRestartableComposable` помечены `Spacer` (нет логики и просто вызов Layout), некоторые перегрузки `Image` и `Icon` (маппинг параметров к своей перегрузке), `Card` (маппинг параметров к `Surface`). Выгода от отказа от перезапускаемости функции минимальна: не генерируется лишний код и не исполняется лишняя логика, но если вы проектируете UI kit, то стоит задуматься об этом, так как ваши элементы будут использоваться во многих местах, часто повторяться, и в сумме это даст эффект.

Оптимизация часто меняющихся элементов

Оптимизировать чтение State<T> нужно только там, где состояние часто меняется и затрагивает много контента. Иначе весь код будет переоптимизирован и станет непригоден для чтения и развития.

Derived state

`derivedStateOf` — производное (вычисляемое) состояние, что и отражает основной сценарий использования.

```
val listState = rememberLazyListState()
val showButton by remember {
    derivedStateOf { listState.firstVisibleItemIndex > 0 }
}
```

Допустим, у нас есть состояние списка, в котором мы читаем индекс первого видимого элемента списка. Но сам по себе он нам не нужен, мы хотим знать, показывать нам кнопку или нет. Чтобы рекомпозировать только при изменении видимости кнопки, а не каждый раз при изменении первого видимого элемента списка, мы можем читать состояние внутри лямбды `derivedStateOf`. Там Derived state подписывается на изменения состояний, которые были прочитаны за первый проход, и возвращает итоговый `State<T>`, который уже вызывает рекомпозицию только при изменении итогового состояния.

Важно, что Derived state реагирует на изменение только `State<T>`, а не обычных переменных, так как `State<T>` имеет функциональность подписки и оповещения при изменении снапшот системы Compose, с которой и работает Derived state.

Подчеркнём что нельзя в качестве ключа `remember` использовать значение часто меняющегося состояния, иначе теряется весь смысл Derived state и рекомпозиция в этом месте будет происходить часто, как и пересоздание Derived state:

```
// Не делать так
val listState = rememberMyListState()
val showButton by remember(listState.value) { // Чтение listState
    derivedStateOf { listState.value > 0 }
}

// Не делать так
val listState by rememberMyListState()
val showButton by remember(listState) { // Чтение listState
    derivedStateOf { listState > 0 }
}
```

Derived state следует использовать только тогда, когда производное состояние будет меняться реже, чем исходные:

```
// Не делать так
val derivedScrollOffset by remember {
    derivedStateOf { scrollOffset - 10f }
}
```

Derived state полезен для производных состояний от состояний прокрутки ленивого списка, свайпа и других часто меняющихся. Ещё несколько примеров:

- Слежение за тем, переходит ли прокрутка порог (`scrollPosition > 0`).
- Количество элементов в списке больше порога (`items > 0`).
- Валидация формы (`username.isValid()`).

Для вложенных Derived state периодически необходимо указывать политику мутации, чтобы не пересчитывать выражение при изменении первого (вложенного) `derivedStateOf`.

```
val showScrollToTop by remember {
    // Политика мутации — structuralEqualityPolicy()
    derivedStateOf(structuralEqualityPolicy()) { scroll0ffset > 0f }
}

var buttonHeight by remember {
    derivedStateOf {
        // Благодаря указанию политики мутации в showScrollToTop
        // этот блок вычисления будет вызываться только при изменении showScrollToTop
        if (showScrollToTop) 100f else 0f
    }
}
```

Подробнее про политики мутаций читайте в этой статье.

Отложенное чтение состояний в composable-функциях

В предыдущем пункте описан принцип, который использует Derived state: он читает состояние внутри себя и не даёт ему рекомпозировать всю функцию. Мы можем использовать этот же принцип, но уже для того, чтобы отложить чтение состояния из родительской функции в дочернюю. Такое следует делать также только для часто меняющихся состояний. Откладывать чтение можно с помощью лямбды или передачи State и чтения его в нужном месте.

```
@Composable
fun MyComposable1() {
    val scrollState = rememberScrollState()
    val counter = remember { mutableStateOf(0) }

    MyList(scrollState)
    MyComposable2(counter1, { scrollState.value })
}

@Composable
fun MyComposable2(counter: State<Int>, scrollProvider: () -> Int) {
    // Чтение состояния в MyComposable2
    Text(text = "My counter = ${counter.value}")
    Text(text = "My scroll = ${scrollProvider()}")
}
```

В коде выше из-за быстрого счётчика или прокрутки рекомпозироваться будет только функция `MyComposable2`, а не вся `MyComposable1`.

Отложенное чтение состояний в фазах Compose

Откладывать чтение состояния можно не только между composable-функциями, но и между фазами Compose (Composition → Layout → Drawing). Например, если у нас часто меняется цвет, то лучше вместо модификатора `background()` использовать `drawBehind { }`, который принимает лямбду и будет вызывать код при смене состояния только на стадии отрисовки, а не композиции, как `background()`.

Подобное можно использовать при прокрутке: модификатор `offset { }` с лямбдой вместо простого `offset(value)`. Так мы откладываем чтение состояния в фазу Layout.

```
@Composable
fun Example() {
    var state by remember { mutableStateOf(0) }

    Text(
        // Чтение состояния при композиции (Composition)
        "My state = $state",
        Modifier
            .layout { measurable, constraints ->
                // Чтение состояния при компоновке (Layout)
                val size = IntSize(state, state)
            }
            .drawWithCache {
                // Чтение состояния при рисовании (Drawing)
                val color = state
            }
    )
}
```

Дополнительно про оптимизацию пропусков в контексте анимаций можно посмотреть в видео от red_mad_robot.

Уменьшение области рекомпозиции

Нужно разбивать на небольшие функции там, где можно уберечь части кода от рекомпозиции. Если вы видите, что часть функции остаётся неизменной, а часть меняется довольно часто, то, вероятно, лучше разбить эту функцию на две. Таким образом, одна функция будет пропускаться, а часто меняющаяся будет рекомпозировать меньшую область. Но не нужно увлекаться и выделять `Divider` в отдельную функцию.

Вот пример выноса логики таймера в отдельную функцию с уменьшением количества рекомпозиций в Promo, так как перезапускаться будет только `Timer` (обратите внимание на место вызова `timer.value`, который и вызывает перезапуск при изменении):

```
@Composable
fun Promo(timer: State<Int>) {
    Text("Sample text")
```

```
    Image()

    // Старый код таймера прямо в функции Promo
    // Text("Осталось ${timer.value} секунд")

    // Новый код таймера
    Timer(timer)
}

@Composable
fun Timer(timer: State<Int>) {
    // Код таймера и чтение состояния timer (timer.value) внутри
}
```

Использование key и contentType в списках

В ленивых списках необходимо передавать ключ в `item()` для того, чтобы списки знали, как меняются данные. Также передавать contentType, чтобы списки знали, какие элементы можно переиспользовать.

```
LazyColumn {
    items(
        items = messages,
        key = { message -> message.id },
        contentType = { it.type }
    ) { message ->
         MessageRow(message)
    }
}
```

Если вы делаете список через `forEach`, то можно использовать `key() { }`, тогда при изменении списка Compose будет понимать, куда переместились элементы.

```
Column {
    widgets.forEach { widget ->
        key(widget.id) {
            MyWidget(widget)
        }
    }
}
```

Модификаторы

Кастомные модификаторы

Если вы пишете свой модификатор, то:

- Если он без состояния, используйте просто функции.
- Если он с состоянием, то используйте `Modifier.Node` (`ModifierNodeElement`). Раньше для такого рекомендовалось использовать composed. В целом, это можно делать и сейчас, так как подробного руководства по `Modifier.Node` пока что нет.
- Если в модификаторе вызывается composable-функция, то используйте `Modifier.composed`.

Подробнее про модификаторы смотрите в этом видео.

Переиспользование модификаторов

Если в области, где создаются модификаторы, происходит частая рекомпозиция, то стоит задуматься о том, чтобы вынести их создание за эту область. Это, например, актуально для анимаций:

```
val reusableModifier = Modifier
            .padding(12.dp)
            .background(Color.Gray),

@Composable
fun LoadingWheelAnimation() {
    val animatedState = animateFloatAsState(...)

    LoadingWheel(
        modifier = reusableModifier,
        // Чтение часто меняющегося состояния
        animatedState = animatedState.value
    )
}
```

Также рекомендуется выносить модификаторы в списках, чтобы все их элементы переиспользовали единый объект.

```
val reusableItemModifier = Modifier
    .padding(bottom = 12.dp)
    .size(216.dp)
    .clip(CircleShape)

@Composable
private fun AuthorList(authors: List) {
    LazyColumn {
        items(authors) {
            AsyncImage(modifier = reusableItemModifier)
        }
    }
}
```

Выносить модификаторы можно не только из функций, но и просто в родительскую composable-функцию, в которой реже происходит рекомпозиция. Дальше модификаторы можно спокойно дополнять.

```
reusableModifier.clickable { /*...*/ }
otherModifier.then(reusableModifier)
```

Долгие вычисления при рекомпозиции

Долгие вычисления только во ViewModel или в remember

Почти все вычисления должны проводиться только во `ViewModel`. При этом следите, чтобы колбэки (`onButtonClick`, `onIntent`, `onAction`, `onEvent`, `onMessage` ...) не выполняли тяжёлую работу в главном потоке. Если у вас в нём выполняется единая функция для обработки действий пользователя, то можно повесить на неё измерение времени работы и писать в лог о критических значениях длительности исполнения, чтобы разработчик не забывал выносить сложные и долгие вычисления в фоновые потоки.

Лучше выносить всю логику из composable-функций. В остальных случаях, когда долгие или затратные вычисления неудобно вынести во `ViewModel`, применяйте `remember`.

Без долгих вычислений в геттерах UI State

```
data class MyUiState(
    val list1: List<Int> = emptyList(),
    val list2: List<Int> = emptyList(),
) {
    // Не делать так
    val isTextVisible
        get() = list1.any { it == 1 } || list2.any { it != 0 }
}
```

Если у вас в `UiState` есть что-то подобное, чтобы не задавать поле каждый раз, то пересчёт будет происходить при каждой рекомпозиции, так как это просто выполнение метода `getIsTextVisible()`. Поэтому либо уберите геттер (оставив поле в теле класса или перенеся в первичный конструктор), либо убедитесь, что в месте вызова геттера происходит минимум рекомпозиций.

Когда использовать remember

Использовать:

- Для любых долгих или затратных по памяти операций, которые могут быть выполнены более одного раза, но не должны выполняться до изменения ключей (если они нужны), переданных в `remember()`, особенно при частой рекомпозиции.

```
val brush = remember(key1 = avatarRes) {
    ShaderBrush(
        BitmapShader(
            ImageBitmap.imageResource(res, avatarRes).asAndroidBitmap(),
            Shader.TileMode.REPEAT,
            Shader.TileMode.REPEAT
        )
    )
}
```

- Для обёртки лямбд с нестабильными внешними переменными.
- Для классов без переопределённого `equals`.
- Для объектов, которые нужно сохранить между рекомпозициями (например, `State<T>`).

Если ключи `remember` обновляются очень часто, то стоит задуматься, а нужен ли в этом случае `remember`. Также в ключи не нужно добавлять вообще всё, от чего зависит вычисление внутри `remember`: если вы понимаете, что эти значения никогда не поменяются за время жизни composable-функции и самого `remember`, то не добавляйте их в ключи.

Layout

Кастомные Layout

Не бойтесь делать кастомные лейауты: они намного проще, чем во View, главное начать. Полезные видео и статьи по этой теме в Compose: тык1, тык2, тык3, тык4.

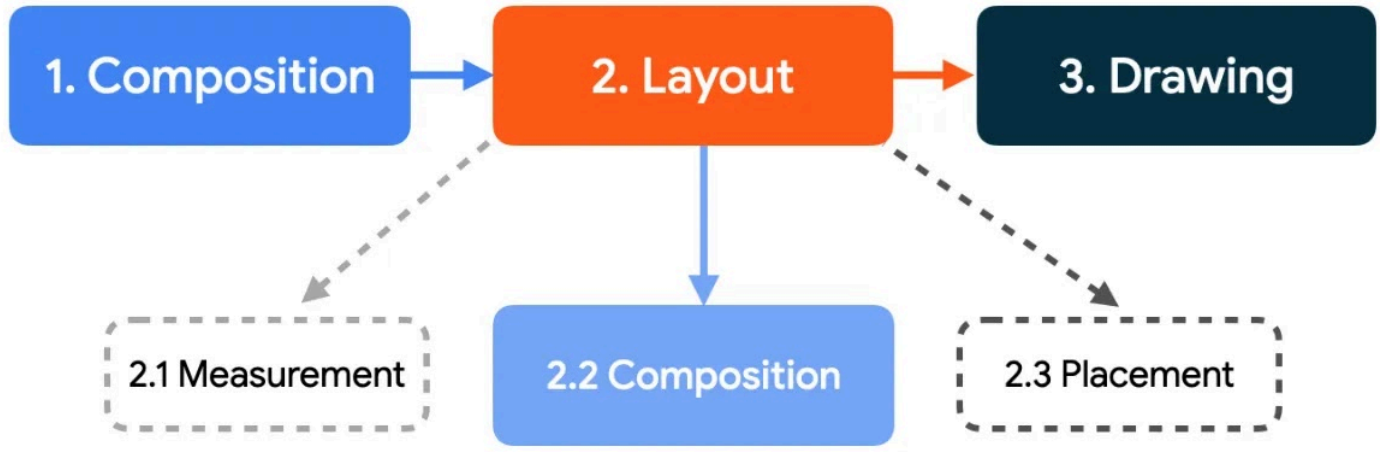
Необоснованное изменение размера и расположения

Избегайте необоснованного изменения размера Compose-элементов, особенно в списках. Такая проблема может возникнуть, если не установить фиксированный размер картинки и после загрузки из интернета она поменяет свой размер. Необоснованное изменение размера или расположения может произойти из-за модификаторов `onGloballyPositioned()`, `onSizeChanged()` и подобных. Из-за этого может происходить множество лишних рекомпозиций. Если элементам нужно знать о расположении и размерах других элементов, то чаще всего это означает, что вы либо используете не тот лейаут, либо вам нужно сделать кастомный.

Предварительное вычисление Layout

SubcomposeLayout

`SubcomposeLayout` откладывает композицию до измерения в фазе Layout, чтобы мы могли использовать доступное пространство для композиции дочерних элементов. Второе полезное применение — это условная композиция. Например, в зависимости от размера окна приложения мы можем по-разному располагать элементы (для планшета или для телефона, или вообще для окна, размер которого можно менять). Или же в зависимости от состояния прокрутки вызывать композицию конкретных элементов для реализации ленивого списка. `SubcomposeLayout` довольно дорогой, поэтому не стоит использовать его для предварительного вычисления лейаута в любом другом случае.



Основные реализации `SubcomposeLayout` — `BoxWithConstraint`, `LazyRow` и `LazyColumn` — покрывают большую часть потребностей, которую не может покрыть Layout.

Также использование под капотом `SubcomposeLayout` объясняет, почему `LazyRow` и `LazyColumn` проигрывают по производительности `Row` и `Column` при малом количестве элементов в списке. Так что, если у вас небольшой список, то используйте для него `Row` и `Column`.

`SubcomposeLayout` иногда ошибочно используют для реализации Slot API:

```
// Не нужно так делать!
@Composable
fun DontDoThis(
    slot1: @Composable () -> Unit,
    slot2: @Composable () -> Unit
) {
    SubcomposeLayout { constraints ->
        val slot1Measurables = subcompose("slot1", slot1)
        val slot2Measurable = subcompose("slot2", slot2)

        layout(width, height) {
            ...
        }
    }
}
```

Для Slot API есть более правильный вариант: через модификатор `layoutId()` и поиск среди measurables по полю `layoutId` или `Layout` с передачей списка composable и деконструкцией списка measureable по порядку.

```
@Composable
fun DoThis(
    slot1: @Composable () -> Unit,
    slot2: @Composable () -> Unit
) {
    Layout(
        contents = listOf(slot1, slot2)
    ) { (slot1Measurables, slot2Measurables), constraints ->
        ...
        layout(width, height) {
            ...
        }
}
```

Intrinsic measurements

Intrinsic measurements более эффективны, чем `SubcomposeLayout`, и под капотом работают очень похоже на `LookaheadLayout`. Оба подхода вызывают лямбду измерений (которая передаётся в `LayoutModifiers` или `MeasurePolicy`) с различными ограничениями (constraints) в одном и том же кадре. Но в случае с Intrinsics это предварительный расчёт для того, чтобы выполнить реальное измерение, используя полученные значения.

Представьте себе строку (`Row`) с тремя дочерними элементами. Для того чтобы её высота соответствовала высоте самого высокого ребёнка, строке нужно получить внутренние измерения (intrinsic measurements) всех своих детей и после этого измерить себя, используя максимальное значение. Подробнее читайте на Android Developers.

Intrinsic measurements может негативно влиять при сложных лейаутах в ленивых списках, но незначительно.

LookaheadLayout

Используется для точного предварительного расчёта размера и положения любого (прямого или косвенного) дочернего элемента для обеспечения анимации (например, перехода одного элемента в другой). `LookaheadLayout` выполняет более агрессивное кэширование, чем `intrinsic`, чтобы не заглядывать вперёд, если дерево не изменилось. Подробнее можно прочитать в статье от Jorge Castillo.

Прочие советы

Не менять состояние, которое только прочитали

Эта ошибка более характерна для новичков. Мы сразу после чтения вызываем изменение состояния, из-за этого сразу выполняется рекомпозиция.

```
@Composable
fun BadComposable() {
    var count by remember { mutableStateOf(0) }

    // Вызывает рекомпозицию при клике
    Button(onClick = { count++ }) {
        Text("Recompose")
    }

    Text("$count")
    count++
    // Обратная запись: запись в состояние сразу после того,
    // как оно было прочитано
}
```

MovableContentOf

`movableContentOf` полезен, когда мы хотим переместить наши элементы в другое место, но при этом не вызвав заново рекомпозицию и не теряя запомненное состояние:

```
val myItems = remember {
    movableContentOf {
        MyItem(1)
        MyItem(2)
    }
}

if (isHorizontal) {
    Row {
        myItems()
    }
} else {
    Column {
        myItems()
    }
}
```

Подробнее читайте в статье от Jorge Castillo.

Интересный факт: конструкция `key()` похожа по логике на `movableContentOf()`, так как оба подхода используют movable-группу, которая позволяет перемещать Compose-код без рекомпозиции.

staticCompositionLocalOf и compositionLocalOf

`staticCompositionLocalOf` обычно нужен, когда Composition Local используется огромным количеством composable-функций и значение вряд ли будет меняться. Примеры реализаций: `LocalContext`, `LocalLifecycleOwner`, `LocalDensity`, `LocalFocusManager` и др.

`compositionLocalOf` влечёт дополнительные затраты при начальном построении дерева композиции, так как все composable-функции, которые читают текущее значение, должны быть отслежены. Если значение будет часто меняться, то `compositionLocalOf`, возможно, будет лучшим выбором. Примеры реализаций: `LocalConfiguration`, `LocalAlpha` и др.

@ReadOnlyComposable

Если composable-функция выполняет только операции чтения, то можно пометить её аннотацией `@ReadOnlyComposable`. В результате у неё не будет сгенерирована группа. Это даст небольшой прирост производительности. Основной сценарий использования - функция, которой аннотация `@Composable` нужна только для чтения `CompositionLocal` (например, чтение цвета из темы), а не для вызова других composable-функций. Подробнее на Android Developers.

Использовать меньше ComposeView

Тут всё просто: чем меньше `ComposeView` вы используете для моста с View, тем быстрее Compose будет работать. Также чем раньше при запуске приложения появляется Compose-код, тем лучше будет работать дальнейший код, так как Compose успеет «прогреться».

Использовать последнюю версию Compose

Разработчики Compose почти каждую версию улучшают производительность, поэтому не забывайте обновляться.

Baseline Profiles

Baseline Profiles хорошо описаны на Android Developers. В целом, Google уже делает подобную работу за нас с помощью Cloud Profiles. Baseline Profiles помогут, если мы хотим улучшить наши метрики на старых версиях Android (7 — 8.1) и в начале нового релиза на новых Android (9+).

Отладка и мониторинг производительности

Оптимизация — это хорошо, но лучше всегда проверять свои изменения, чтобы понимать, что вы реально решили проблему, а не добавили новую.

Проверять производительность необходимо почти всегда в релизном режиме и с R8. Режим отладки имеет множество плюшек при разработке, что замедляет и искажает итоговый код приложения. Компилятор R8 также значительно оптимизирует код.

Также про отладку Compose хорошо написано в этой статье и рассказано в видео от Android Developers.

Проверка стабильности и пропускаемости

Чтобы проверить типы и composable-функции в проекте на стабильность, нужно запустить генерацию метрик. Как это сделать: Composable metrics и Interpreting Compose Compiler Metrics.

Метрики будут лежать по пути: модуль/build/compose_metrics. Среди них важны два файла:

- -classes.txt для метрик типов;

```
unstable class WidgetUiState {
  unstable val widgets: List<Widget>
  stable val showLoader: Boolean
  <runtime stability> = Unstable
}
```

- -composables.txt и .csv для метрик функций;

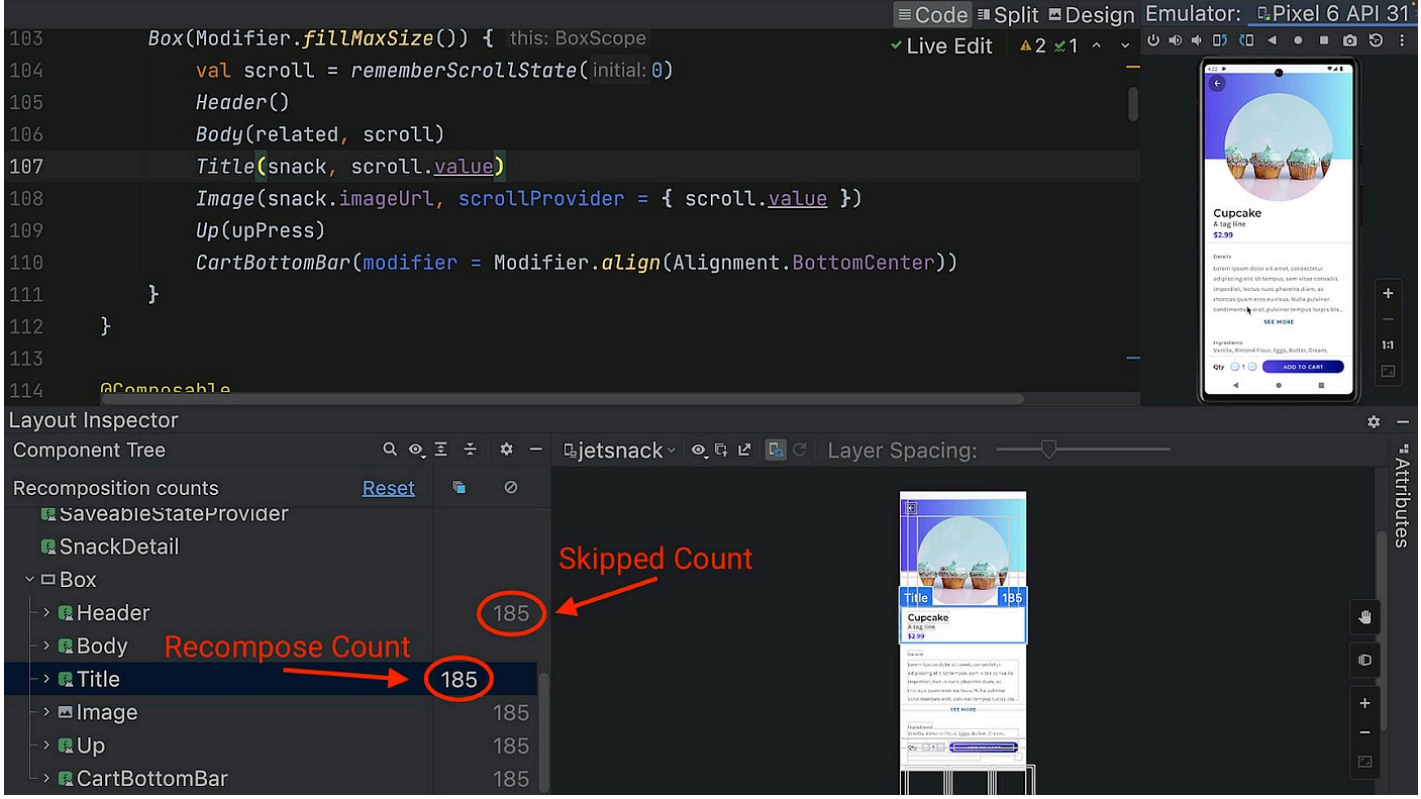
```
restartable scheme("[androidx.compose.ui.UiComposable]") fun MyWidgets(
  unstable widgets: List<Widget>
  stable modifier: Modifier? = @static Companion
)
```

- -module.json для статистики по модулю.

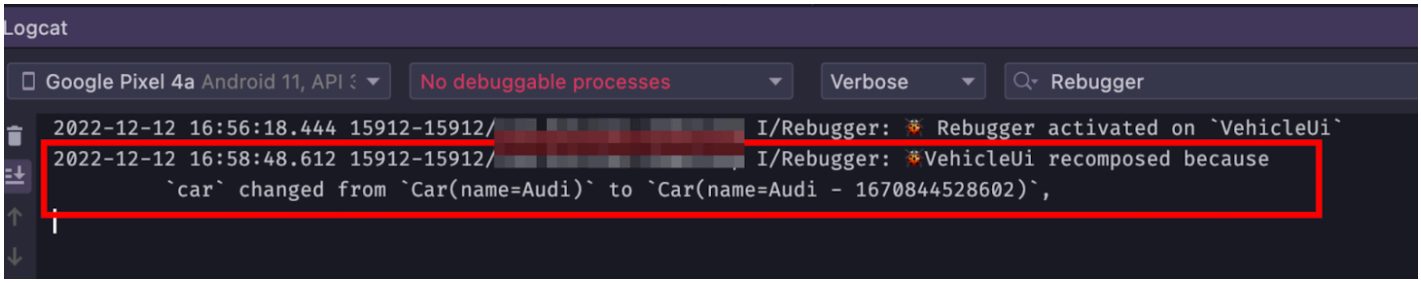
Также есть библиотека для отображения метрик в HTML: Compose Compiler Report to HTML.

Отладка рекомпозиций и пропусков

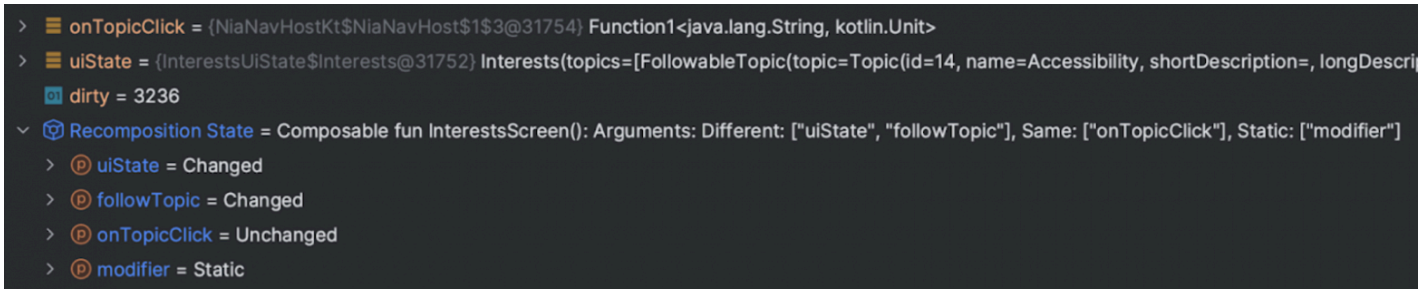
Чтобы не повторяться, советую посмотреть видео про отладку рекомпозиций и пропусков. Смысл в том, что вы включаете подсчёт рекомпозиций и пропусков. После этого на своём экране делаете обычные действия. Дальше смотрите, где есть лишние рекомпозиции, которых можно было бы избежать, или смотрите, где происходит рекомпозиция, хотя данные не менялись и мог бы происходить пропуск.



Также для отладки рекомпозиций можно использовать Rebugger. Эта библиотека позволяет отслеживать изменения в заданных аргумента и выводить причины рекомпозиции.

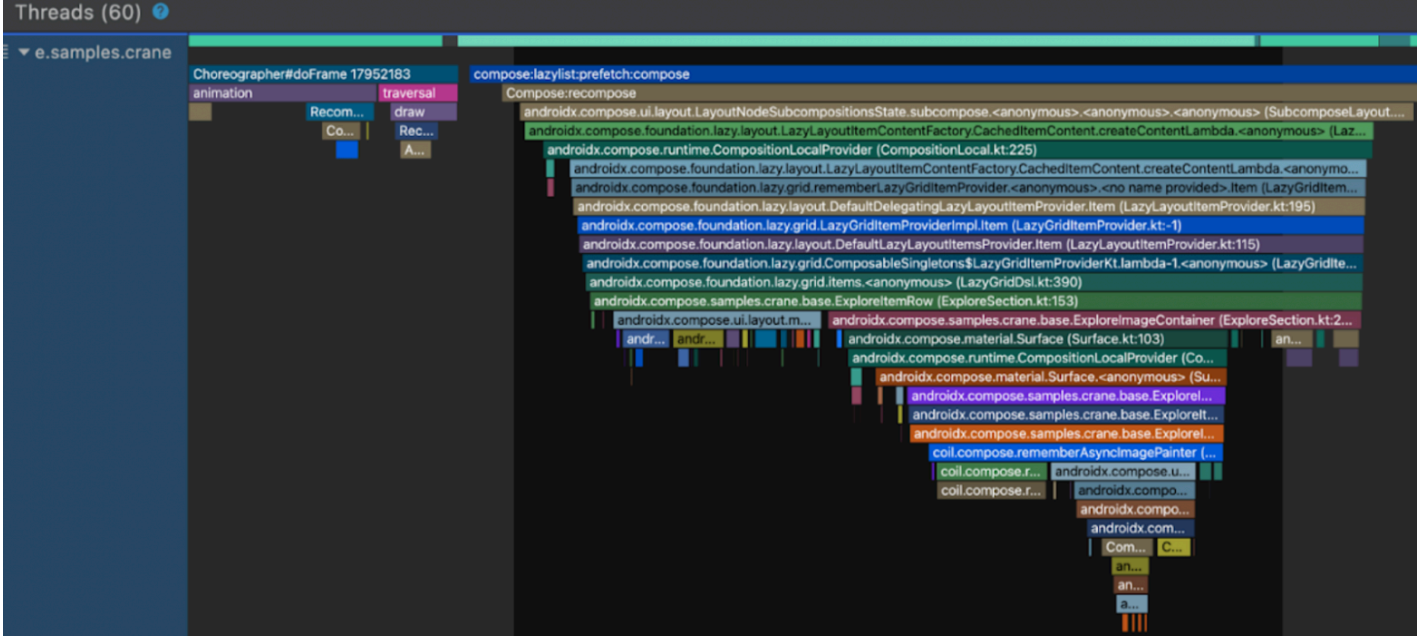


В Android Studio Hedgehog добавят дополнительную информацию в отладчики для просмотра состояния Compose.



Трассировка композиции

Для глубокого анализа проблем ваших UI-элементов используйте трассировку композиции. Она доступна начиная с Android Studio Flamingo. Как это делать, описано в статье “Composition tracing”, а также немного в этом видео.



Бенчмаркинг

Проверяйте свои экраны после оптимизаций с помощью бенчмарков (например, бенчмарк на прокрутку списка с измерением длительности отрисовки кадра).

Итог

В этой статье мы описали весь список оптимизаций, которые нам встретились, а также кое-что из практики нашей команды. Некоторые советы можно преобразовать в кодстайл вашей команды и свободно использовать. Другие полезны только в том случае, если ваш элемент часто меняется. Проверяйте производительность после каждой оптимизации, чтобы не снизить скорость работы. А также не обязательно заранее чрезмерно оптимизировать код, иначе он превратится в нечитаемое нечто.

Бонус (чит-лист)

Оптимизация Jetpack Compose

OZON tech

Стабильность типов

Стабильные:

- Все примитивные типы и String.
- Функциональные типы (лямбды).
- Классы, у которых все поля стабильного типа и объявлены как val.
- Типы, помеченные `@Immutable` или `@Stable`.

НЕстабильные:

- Классы, у которых хотя бы одно поле нестабильного типа или объявлено как var.
- Классы из внешних модулей и библиотек, где нет Compose компилятора (`List`, `Set`, `Map` и пр. коллекции, `LocalDateTime`, `Flow`...).

Дженерики (`MyClass<T>`)—
проверка сперва по структуре дженерика, а потом по указанному типу.

Предопределённые стабильные внешние типы и дженерики:
`Pair`, `Result`, `Comparator`, `ClosedRange`, коллекции из библиотеки `kotlinx.collections.immutable`, `dagger.Lazy` и др.

Типы с вычисляемой стабильностью —
проверка при получении конкретных объектов в рантайме:

- Типы, которые объявлены в других модулях с включённым компилятором Compose.
- Интерфейсы (проверка идёт по типу-наследнику, объект которого будет передан в аргументы).

Контракт `@Immutable` и `@Stable`

1. `equals` всегда будет возвращать одинаковое значение для одной и той же пары объектов.
2. Когда публичные поля типа изменяются, нужно оповестить об этом Compose.
3. Все публичные поля стабильны.

Обратите особое внимание на `equals`: использовать data class или написать свою реализацию.
Стабильность от аннотаций наследуется.

Пропускаемость функций

Функция пропускаема, если нет нестабильных аргументов.

- ✓ Помечайте UI модели `@Immutable` или `@Stable`.
- ✓ Не передавайте лишних данных.
- ✓ Для нестабильных внешних типов используйте обёртку value class с аннотацией.
- ✓ Лямбда не оборачивается в remember, если захватывает нестабильные и вычисляемые в рантайме переменные или var.
- ✓ Вызовы ViewModel (в т.ч. по ссылке на метод) всегда оборачивайте в remember.
- ✓ Предпочитайте аргументы, полученные внутри лямбды:

```
@Composable
fun MyComposableItem(items: List<MyClass>) {
    // Вместо такого
    ItemWidget { items[5].doSomething() }

    // Делать так
    ItemWidget(item[5]) { item → item.doSomething() }
}
```

Чтение состояния:

```
val LocalContentAlpha = compositionLocalOf { 1f }

@Composable
fun MyComposable1() {
    val counter1: MutableState<Int> = remember { mutableStateOf(0) }
    var counter2: Int by remember { mutableStateOf(0) }
    MyComposable2(counter1, { counter2 })
}

@Composable
fun MyComposable2(counter1: State<Int>, counterProvider2: () → Int) {
    Text("Counter = ${counter1.value}") // Чтение состояния
    Text("Counter = ${counterProvider2()}") // Чтение состояния
    Text("Counter = ${LocalContentAlpha.current}") // Чтение состояния
}
```

Перезапускаемая функция — функция, которая может перезапустить себя.

Перезапускаемость и пропускаемость не нужны:

- composable-функция меняется редко или никогда;
- composable-функция вызывает другие пропускаемые composable-функции:
  - глупая функция без State<T>, минимум других функций;
  - враппер над другой функцией (маппер параметров).

`@NonRestartableComposable` убирает перезапускаемость и пропускаемость.

```
@Composable
@NonRestartableComposable
fun ColumnScope.SpacerHeight(height: Dp) {
    Spacer(modifier = Modifier.height(height))
}
```

Оптимизация часто меняющихся элементов

Оптимизировать чтение State<T> нужно только там, где состояние часто меняется и затрагивает много контента!

Derived state — если производное состояние будет меняться реже, чем исходные.

```
val listState = rememberLazyListState()
val showButton by remember {
    derivedStateOf { listState.firstVisibleItemIndex > 0 }
}
```

Отложенное чтение состояний в composable-функциях:

```
@Composable
fun MyComposable1() {
    val scrollState = rememberScrollState()
    val counter = remember { mutableStateOf(0) }

    MyList(scrollState)
    MyComposable2(counter1, { scrollState.value })
}

@Composable
fun MyComposable2(counter: State<Int>, scrollProvider: () → Int) {
    // Чтение состояния в MyComposable2
    Text(text = "My counter = ${counter.value}")
    Text(text = "My scroll = ${scrollProvider()}")
}
```

Отложенное чтение состояний в фазах Compose:

```
@Composable
fun Example() {
    var state by remember { mutableStateOf(0) }

    Text(
        // Чтение состояния при композиции (Composition)
        "My state = $state",
        Modifier
            .layout { measurable, constraints →
                // Чтение состояния при компоновке (Layout)
                val size = IntSize(state, state)
            }
            .drawWithCache {
                // Чтение состояния при рисовании (Drawing)
                val color = state
            }
    )
}
```

Уменьшение области рекомпозиции:

```
@Composable
fun Promo(timer: State<Int>) {
    Text("Sample text")
    Image()

    // Старый код таймера прямо в функции Promo
    // Text("Осталось ${timer.value} секунд")

    // Новый код таймера
    Timer(timer)
}

@Composable
fun Timer(timer: State<Int>) {
    // Код таймера
}
```

key и contentType:

```
LazyColumn {
    items(
        items = messages,
        key = { message → message.id },
        contentType = { it.type }
    ) { message →
        MessageRow(message)
    }
}
```

```
Column {
    widgets.forEach { widget →
        key(widget.id) {
            MyWidget(widget)
        }
    }
}
```

Layout

- ✓ Пишите кастомные лейауты.
- ✓ Не меняйте лишний раз размер и положение элементов списка.
- ✓ Ставьте фиксированный размер картинки.
- ✓ Избегайте `onGloballyPositioned()` и `onSizeChanged()`.
- ✓ `SubcomposeLayout` откладывает композицию до измерения в фазе Layout, из-за этого медленный. Основные реализации — `BoxWithConstraint`, `LazyRow` и `LazyColumn`.
- ✓ Для небольших списков используйте `Row` и `Column`.
- ✓ **Intrinsic measurements** выполняют предварительные вычисления дочерних элементов и почти не влияют на производительность.

Модификаторы

Кастомные модификаторы

- Если модификатор без состояния, то использовать просто функции.
- Если модификатор с состоянием, то использовать `Modifier.Node` (`ModifierNodeElement`) (или `composed`).
- Если в модификаторе вызывается composable-функция, то использовать `Modifier.composed`.

Переиспользование модификаторов

При частой рекомпозиции:

```
val reusableModifier = Modifier
        .padding(12.dp)
        .background(Color.Gray),

@Composable
fun LoadingWheelAnimation() {
    val animatedState = animateFloatAsState(...)

    LoadingWheel(
        modifier = reusableModifier,
        // Чтение часто меняющегося состояния
        animatedState = animatedState.value
    )
}
```

В списках:

```
val reusableItemModifier = Modifier
    .padding(bottom = 12.dp)
    .size(216.dp)
    .clip(CircleShape)

@Composable
private fun AuthorList(authors: List) {
    LazyColumn {
        items(authors) {
            AsyncImage(modifier = reusableItemModifier)
        }
    }
}
```

Долгие вычисления при рекомпозиции

- ✓ Долгие вычисления только во `ViewModel` или в `remember`.
- ✓ Без долгих вычислений в геттерах UI State.

Когда использовать `remember`:

- Для любых долгих или затратных по памяти операций, которые могут быть выполнены более одного раза, но не должны выполняться до изменения ключей (если они нужны), переданных в `remember()`, особенно при частой рекомпозиции.
- Для обёртки лямбд с нестабильными внешними переменными и для классов без переопределённого `equals`.

Прочее

- ✓ `staticCompositionLocalOf` — если значение будет редко меняться. При использовании огромным количеством composable-функций.
  Примеры: `LocalContext`, `LocalLifecycleOwner`, `LocalDensity`, `LocalFocusManager`.
- ✓ `compositionLocalOf` — если значение будет часто меняться. Есть затраты на подписку.
  Примеры: `LocalConfiguration`, `LocalAlpha`.
- ✓ `@ReadOnlyComposable` — если composable-функция только выполняет чтение.
  Примеры: чтение только `CompositionLocal` (цвет из темы).
- ✓ Используйте меньше ComposeView.

Отладка и мониторинг

- Composable metrics: проверка стабильности и пропускаемости.
- Layout Inspector: отладка рекомпозиций и пропусков.
- Rebugger.
- Compose State в отладчике.
- Трассировка композиции.
- Бенчмаркинг.

Теги: android, android development, compose, kotlin, jetpack compose, jetpack, мобильная разработка, ozon tech, оптимизация кода

Хабы: Блог компании Ozon Tech, Программирование, Разработка мобильных приложений, Android, Kotlin

+37 194 16 +16

Ozon Tech

Команда разработки ведущего e-com в России

Подписаться

16 Карма 0 Рейтинг

Андрей Богомолов @tipapro

Android developer

Подписаться

Комментарии 16

dyadyaSerezha 23 июн 2023 в 11:28

> улучшило наши метрики: длительность лага по отношению к длительности скролла

Серьезно? У вас такие метрики работы программистов? Дальше читать не смог)

-7 Ответить

comanch_mai 23 июн 2023 в 13:16

Спасибо, много полезной инфы.

+3 Ответить

SchwarzerEngel 25 июн 2023 в 14:29

Вот это материал! Спасибо, очень полезно!

+1 Ответить

keotjape 26 июн 2023 в 11:58

Привет! Не понял один момент.

Для закрепления главы вот ещё примеры, из-за которых `MyComposable2` будет точкой перезапуска (рекомпозиции) и пойдёт по всем её детям, а `MyComposable1` не будет затронута. Можно добавить, что функции `animateColorAsState()`, `rememberScrollState()` и пр. тоже внутри содержат `State<T>`, и могут стать причиной рекомпозиции при изменении.

```
val LocalContentAlpha = compositionLocalOf { 1f }

@Composable
fun MyComposable1() {
    val counter1: MutableState<Int> = remember { mutableStateOf(0) }
    var counter2: Int by remember { mutableStateOf(0) }
    MyComposable2(counter1, { counter2 })
}

@Composable
fun MyComposable2(counter1: State<Int>, counterProvider2: () -> Int) {
    Text("Counter = ${counter1.value}") // Чтение состояния
    Text("Counter = ${counterProvider2()}") // Чтение состояния
    Text("Counter = ${LocalContentAlpha.current}") // Чтение состояния
}
```

Здесь вы говорите, что MyComposable1 не будет затронута, но при этом здесь есть следующий участок кода:

```
var counter2: Int by remember { mutableStateOf(0) }
```

про который вы в следующем абзаце говорите, что

> Обратите внимание, что если вы используете `State<T>` как делегат, то в этом случае будьте осторожнее со случайным чтением состояния, особенно если оно часто меняющееся.

```
@Composable
fun MyComposable1()
    var counter: Int by remember { mutableStateOf(0) }

    // Чтение состояния произойдёт в MyComposable1, а не в MyComposable2!!!
    MyComposable2(counter)
}
```

В итоге, MyComposable1 будет изменяться или нет ? Утверждения противоречат друг другу. Спасибо!

0 Ответить

tipapro 26 июн 2023 в 13:01

Привет, здесь работает принцип отложенного чтения через лямбду.

Для лямбды `{ counter }` в `MyComposable2({ counter })` сгенерируется подобный код (условно, не точно такой):

```
class Lambda(val counter: MutableState<String>) {
  fun invoke(): String {
    return counter.value
  }
}
```

И уже состояние прочитается в MyComposable2, так как именно там лямбда вызовется.

В тоже время если передавать состояние так:

```
@Composable
fun MyComposable1() {
  ...
  MyComposable2(counter)
}
```

то это превратится в

```
@Composable
fun MyComposable1() {
  ...
  MyComposable2(counter.value)
}
```

так как делегат по сути скрывает .value от нас, и чтение произойдёт уже в MyComposable1

0 Ответить

keotjape 26 июн 2023 в 17:05

Спасибо за ответ. Возник уже другой вопрос. Вы написали, что

{ counter } как я понимаю, non-composable функция ? (В ней же не выполняется никакой composable код)

Но в самой статье вы говорите следующее:

А что в итоге произойдет: создание анонимного класса или оборачивание функции в remember ?

0 Ответить

tipapro 26 июн 2023 в 17:19

Обе вещи. Такой код внутри composable-функции:

```
val counter = remember { mutableStateOf(3) }

val onClick = { counter.value }

onClick()
```

Превратится примерно в такой:

```
val counter = remember { mutableStateOf(3) }

val onClick = remember(counter) { Lambda(counter) }

onClick()
```

В конце главы про лямбды есть ссылка на видео, где с 25 минуты объясняется, во что превращается лямда в compose. Также лямбды ещё меняются после работы R8 (подобные лямбды объединяются в один класс), но это не так важно в этом контексте, так как на выходе всё равно будет класс (или объект, если не было захвата внешних переменных)

0 Ответить

tipapro 26 июн 2023 в 17:24

Просто не хотел удлинять и усложнять статью ещё и детальным описанием преобразованием лямбды в котлине

0 Ответить

oleg40a 28 июн 2023 в 00:37

Каким софтом нарисованы диаграммы в статье?

0 Ответить

tipapro 28 июн 2023 в 04:13

Привет, первая - draw io, вторая и третья взята из ресурсов Jetpack Compose (вторая просто через фотошоп переведена)

0 Ответить

jershell 8 ноя 2023 в 19:23

Стоит ли беспокоиться о частой рекомпозиции Image & Icon? У них Painter нестабилен, и как следствие функция не получает свойства skippable. Есть ли рекомендации на этот счёт?

0 Ответить

tipapro 8 ноя 2023 в 19:47

Обычно это не проблема, так Image/Icon - простые элементы (в плане структуры) и когда до них доходит рекомпозиция, то в большинстве случаев она реально нужна.
Избегать рекомпозицию можно за счёт того, чтобы просто не пускать её близко к ним.
Например, в коде ниже мы просто следим за пропускаемостью MyItem и этого достаточно, чтобы Image лишний раз не затрагивалась.

```
@Composable
fun MyItem(model: MyModel) {
  Column {
    Text(model.text)
    Image(
      painter = painterResource(model.image),
      contentDescription = null
    )
  }
}
```

В каких-то редких кейсам возможно полезно сделать функцию-обёртку для Image, но у нас такие кейсы не встречались.

Другое дело, если мы сами используем Painter в сложных элементах:

```
@Composable
fun MyComplexItem(image: Painter) {
  // ...
}
```

В таком случаем цена рекомпозиции из-за нестабильной Painter будет высокая и часто будет происходить, когда не нужно. Это можно исправить либо сделав обёртку для Painter и пометив

аннотацией, либо указав Painter стабильным с Compose Compiler 1.5.4+, либо передавать другие данные, чтобы создавать Painter внутри. Об этом написано также в этой статье.

+1 Ответить

Spinoza0 13 ноя 2023 в 20:22

Спасибо!
Тем, кто читает статью, совет - через некоторое время прочитайте еще раз, найдете то, что могли пропустить при первом прочтении)

0 Ответить

themen2 28 янв 2024 в 19:40

Compose это просто, говорили они???

0 Ответить

Libra_by 7 мар 2024 в 15:48

Спасибо за статью. Очень познавательно.

0 Ответить

faritowich 5 фев в 10:49

Не устаю удивляться таймингу статей. Полчаса блин. Часа 2 читал)

0 Ответить

Вы можете оставлять комментарии только к свежим публикациям

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

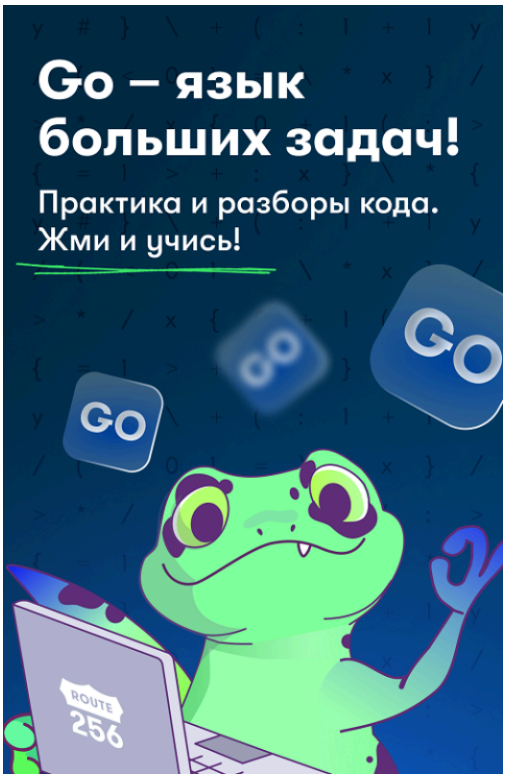
+15 3 0

Показать еще

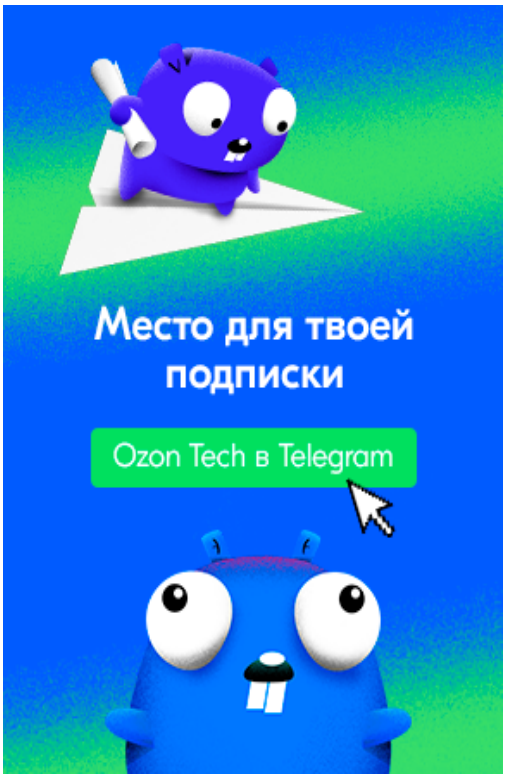
ИНФОРМАЦИЯ

Сайт	ozon.tech
Дата регистрации	3 декабря 2018
Дата основания	9 февраля 1998
Численность	5 001–10 000 человек
Местоположение	Россия

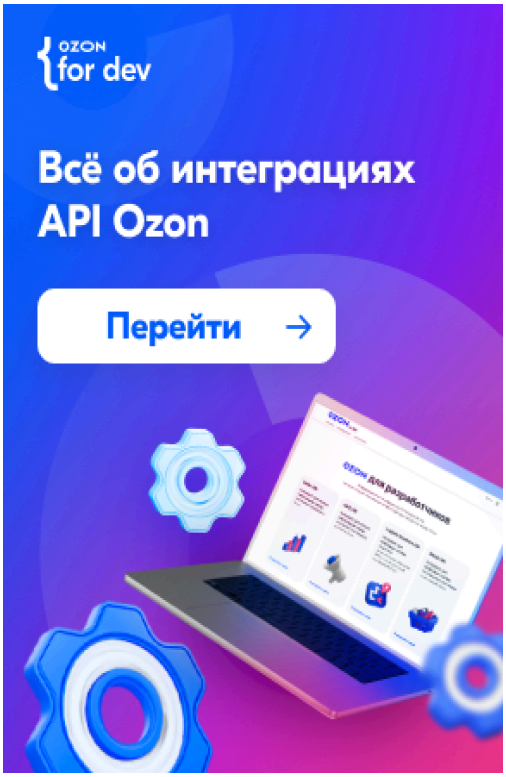
ВИДЖЕТ



ВИДЖЕТ



ВИДЖЕТ



ВИДЖЕТ

ВИДЖЕТ

ВКОНТАКТЕ

БЛОГ НА ХАБРЕ

27 июн в 11:51

Fleet&Osquery — швейцарский нож для ИБ, или Как мы сами себя успешно ддосили

1.1K 5 +5

26 июн в 09:18

Авторизация в Kafka: управление изменениями, когда у тебя тысячи клиентов и миллионы RPS

5.6K 16 +16

25 июн в 15:04

Испытательный срок: 12 шагов к успеху

3.4K 9 +9

31 мая в 12:37

От стажёра до лида. Дорога длиною в Travel

2.2K 2 +2

30 мая в 15:32

От depth map* до нейросети: практический опыт создания аппаратного решения по измерению товаров на складе

2K 3 +3