Предисловие

Зачем читать эту книгу

Jetpack Compose официально стал новым стандартом для разработки UI на платформе Android. Даже если многие приложения по-прежнему опираются на систему View для уже существующего интерфейса, новые экраны и компоненты очень часто пишут с помощью Compose, так что эта технология становится обязательной к изучению. Я настоятельно советую уделить время детальному изучению её внутреннего устройства: это даст вам прочные знания и навыки для написания корректных, современных и эффективных Android-приложений.

С другой стороны, если вам интересны сценарии использования Jetpack Compose не только на Android, вы будете рады узнать, что эта книга охватывает и их. «Jetpack Compose internals» сфокусирована на деталях компилятора и рантайма, что делает изложение в значительной степени независимым от целевой платформы. Наличие опыта в Android не является обязательным для чтения. В книге также есть глава, посвящённая разным сценариям использования библиотеки, с рядом интересных примеров в коде.

Наконец, если вы любознательный инженер, это может стать отличным шансом узнать что-то новое и хорошим вызовом для вас.

О чём эта книга не рассказывает

Эта книга не ставит целью дублировать официальную документацию Jetpack Compose, которая уже достаточно хороша и является источником истины для всех, кто с ней начинает. По этой причине здесь вы не найдёте перечней или каталогов всех существующих компонентов и API библиотеки.

Если вы новичок в Compose, могу порекомендовать держать рядом с этой книгой другую: «Practical Jetpack Compose» Джо Бёрча. В ней много примеров и подробных объяснений компонентов и API. Книга учит Compose на примере приложений.

Зачем писать о внутреннем устройстве

Как Android-разработчик я со временем пришёл к пониманию того, насколько важно изучать внутреннее устройство платформы, с которой работаешь каждый день. Это меняет правила игры. Регулярная работа с этой областью знаний помогает мне понимать, какой код я хочу писать и зачем. Это

позволяет писать производительный код, соответствующий ожиданиям платформы, и понимать, почему всё устроено именно так. Для меня это, пожалуй, одно из главных отличий между менее опытными и опытными Android-разработчиками.

Моя личная цель как автора — дать вам все инструменты, чтобы сделать большой шаг вперёд в этой области.

Держите исходники под рукой

По моему мнению, умение читать исходный код — одно из самых полезных навыков, которые мы можем развить как разработчики, независимо от того, кем этот код написан. Я настоятельно рекомендую всем читающим эту книгу держать исходники как можно ближе во время чтения и исследовать их дальше. Всё можно найти на cs.android.com. Исходники также индексируются в Android Studio, так что по ним можно удобно перемещаться. Иметь под рукой игровой проект с Compose тоже может быть полезно.

Фрагменты кода и примеры

Один из сюжетов книги — Jetpack Compose можно использовать не только для представления UI-деревьев, но по сути для любых больших графов вызовов с узлами любого типа. Тем не менее многие фрагменты и примеры в книге будут ориентированы на UI для более лёгкого восприятия, поскольку к этому привыкло большинство разработчиков.

Добро пожаловать в Jetpack Compose Internals. Приготовьте кофе, устройтесь поудобнее и приятного чтения.

Хорхе.

1. Composable-функции

Смысл Composable-функций

Вероятно, самый уместный способ начать книгу о внутреннем устройстве Jetpack Compose — разобраться с Composable-функциями: они являются атомарными строительными блоками Jetpack Compose и той конструкцией, с помощью которой мы пишем composable-деревья. Я намеренно говорю здесь «деревья»: composable-функции можно понимать как узлы в большом дереве,

которое Compose runtime представляет в памяти. К деталям мы придём в своё время, но правильную модель полезно формировать с самого начала.

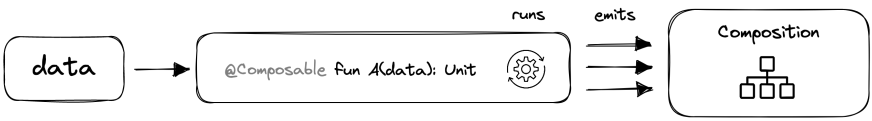
Если говорить о синтаксисе, любая обычная Kotlin-функция может стать Composable-функцией — достаточно аннотировать её как `@Composable`:

```
1 @Composable
2 fun NamePlate(name: String) {
3   // Our composable code
4 }
```

Этим мы по сути сообщаем компилятору, что функция призвана преобразовать некоторые данные в узел для регистрации в composable-дереве. То есть если читать Composable-функцию как `@Composable (Input) -> Unit`, то вход — это данные, а «выход» — не возвращаемое значение, как можно было бы подумать, а зарегистрированное действие по вставке элемента в дерево. Можно сказать, что это происходит как побочный эффект выполнения функции.

Обратите внимание: возврат `Unit` из функции, принимающей входные данные, означает, что мы так или иначе потребляем эти данные в теле функции.

Описанное действие в терминологии Compose обычно называют «emitting» (эмитирование). Composable-функции эмитируют при выполнении, и это происходит в процессе composition. Подробности этого процесса — в следующих главах. Пока же каждый раз, когда речь идёт о «composing» Composable-функции, можно мысленно заменять это на «выполнение».



Composable-функция эмитирует (схема)

Единственная цель выполнения наших Composable-функций — построить или обновить представление дерева в памяти. Так оно остаётся актуальным относительно структуры, которую описывает: Composable-функции будут выполняться заново при изменении прочитанных ими данных. Для обновления дерева они могут эмитировать действия вставки новых узлов, как сказано выше, а также удалять, заменять или перемещать узлы. Composable-функции также могут читать и записывать состояние в дерево / из дерева.

Свойства Composable-функций

Аннотирование функции как Composable имеет и другие важные следствия. Аннотация `@Composable` фактически **меняет тип** функции или выражения, к которому применена, и, как любой другой тип, накладывает на неё ограничения и свойства. Эти свойства важны для Jetpack Compose, так как раскрывают возможности библиотеки.

Compose runtime ожидает, что Composable-функции соблюдают упомянутые свойства — тогда он может опираться на определённое поведение и использовать различные оптимизации: параллельная composition, произвольный порядок composition по приоритетам, умная recomposition, позиционная мемоизация и др. Пока не стоит пугаться всех этих понятий — мы разберём каждое в нужный момент.

Вообще говоря, оптимизации на этапе выполнения возможны только тогда, когда runtime может опереться на определённые гарантии относительно кода: условия и поведение. Это позволяет выполнять (или «потреблять») этот код по разным стратегиям и техникам, использующим эти гарантии.

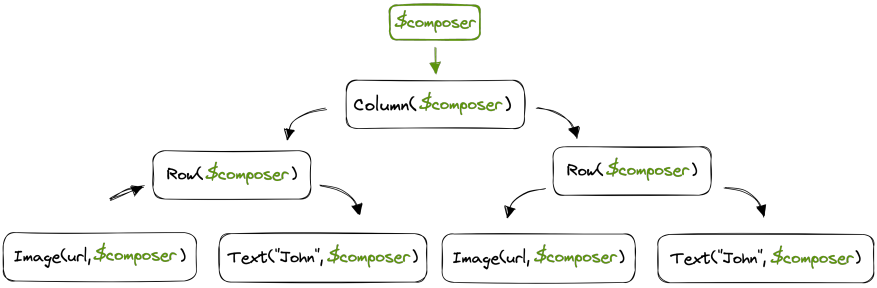
Пример таких гарантий — связь между элементами в коде. Зависят ли они друг от друга? Можно ли выполнять их параллельно или в другом порядке без изменения результата программы? Можно ли считать каждый атомарный фрагмент логики изолированной единицей?

Разберём свойства Composable-функций.

Контекст вызова (calling context)

Большинство свойств Composable-функций обеспечиваются компилятором Compose. Как плагин компилятора Kotlin он выполняется на обычных фазах компиляции и имеет доступ ко всей информации, которой располагает компилятор Kotlin. Это позволяет перехватывать и преобразовывать IR (промежуточное представление) всех Composable-функций в исходниках и добавлять к ним дополнительную информацию.

Один из добавляемых элементов — новый параметр в конце списка параметров: `Composer`. Этот параметр неявный, разработчик о нём не заботится. Его экземпляр внедряется во время выполнения и передаётся во все дочерние вызовы Composable, так что он доступен на всех уровнях дерева.



Контекст вызова Composable (схема)

В коде допустим, у нас есть такой Composable:

```
1 @Composable
2 fun NamePlate(name: String, lastname: String) {
3   Column(modifier = Modifier.padding(16.dp)) {
4     Text(text = name)
5     Text(text = lastname, style =
 ↪  MaterialTheme.typography.subtitle1)
6   }
7 }
```

Компилятор преобразует это во что-то вроде:

```
 1 fun NamePlate(name: String, lastname: String, $composer:
  ↪  Composer&lt;*&gt;) {
 2   ...
 3   Column(modifier = Modifier.padding(16.dp), $composer) {
 4     Text(
 5       text = name,
 6       $composer
 7     )
 8     Text(
 9       text = lastname,
10       style = MaterialTheme.typography.subtitle1,
11       $composer
12     )
13   }
14   ...
15 }
```

Как видно, `Composer` передаётся во все вызовы Composable в теле. Кроме того, компилятор Compose накладывает строгое правило: Composable-функции можно вызывать только из других Composable-функций. Это и есть требуемый **контекст вызова**; он гарантирует, что дерево состоит только из Composable-функций и `Composer` может передаваться вниз.

`Composer` — связующее звено между Composable-кодом, который мы пишем, и Compose runtime. Composable-функции используют его, чтобы эмитировать изменения в дерево и сообщать runtime о его форме для построения или обновления представления в памяти.

Идемпотентность

От Composable-функций ожидается идемпотентность относительно дерева узлов, которое они порождают. Повторное выполнение Composable-функции с теми же входными параметрами должно давать то же дерево. Jetpack Compose runtime опирается на это при recomposition и т.п.

В Jetpack Compose **recomposition** — это повторное выполнение Composable-функций при изменении их входных данных, чтобы они могли эмитировать обновлённую информацию и обновлять дерево. Runtime должен иметь возможность перезапускать наши Composable-функции в произвольные моменты и по разным причинам.

Процесс recomposition обходит дерево и определяет, какие узлы нужно перезапустить (выполнить заново). Перезапускаются только узлы с изменившимися входами, остальные **пропускаются**. Пропуск узла возможен только когда представляющая его Composable-функция идемпотентна: runtime может считать, что при тех же входах результат будет тем же. Этот результат уже есть в памяти, поэтому Compose не нужно выполнять функцию снова.

Отсутствие неконтролируемых побочных эффектов

Побочный эффект — любое действие, выходящее из-под контроля функции, в которой оно вызывается, и делающее что-то неожиданное «сбоку». К побочным эффектам относят чтение из локального кэша, сетевой запрос, установку глобальной переменной. Они делают вызывающую функцию зависимой от внешних факторов: внешнего состояния, которое могут менять другие потоки, сторонних API, которые могут выбросить исключение, и т.д. Иными словами, функция не зависит только от своих входов при получении результата.

Побочные эффекты — **источник неоднозначности**. Для Compose это плохо: runtime ожидает, что Composable-функции предсказуемы (детерминированы) и их можно безопасно выполнять многократно. Если Composable-функция выполняет побочные эффекты, при каждом запуске состояние программы может быть разным, и идемпотентность теряется.

Представим, что мы выполняем сетевой запрос прямо из тела Composable-функции:

```
@Composable
fun EventsFeed(networkService: EventsNetworkService) {
  val events = networkService.loadAllEvents()

  LazyColumn {
    items(events) { event -&gt;
      Text(text = event.name)
    }
  }
}
```

Это было бы очень рискованно: функция может многократно перезапускаться за короткое время со стороны Compose runtime, из-за чего сетевой запрос будет уходить снова и снова и выйдет из-под контроля. Ещё хуже то, что эти выполнения могут происходить из разных потоков без координации.

Compose runtime оставляет за собой право выбирать стратегии выполнения наших Composable-функций. Он может переносить recomposition на другие потоки для использования нескольких ядер или выполнять их в произвольном порядке по своим приоритетам (например, Composable, не отображаемые на экране, могут получить более низкий приоритет).

Ещё один типичный подводный камень — зависимость одной Composable-функции от результата другой и неявный порядок вызовов. Такого нужно избегать. Пример:

```
@Composable
fun MainScreen() {
  Header()
  ProfileDetail()
  EventList()
}
```

В этом фрагменте `Header`, `ProfileDetail` и `EventList` могут выполняться в

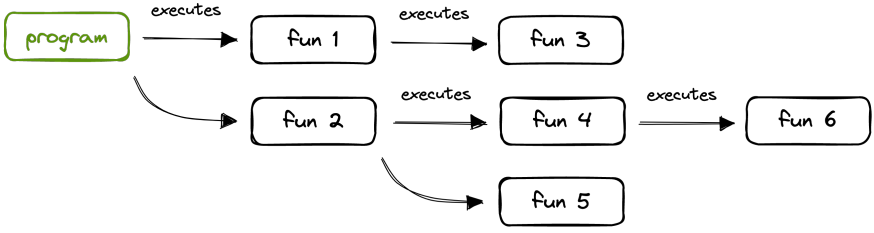
любом порядке или даже параллельно. Не стоит писать логику, опирающуюся на конкретный порядок — например, читать в `ProfileDetail` внешнюю переменную, которую по замыслу записывает `Header`.

В общем случае побочные эффекты в Composable-функциях нежелательны. Нужно стремиться к тому, чтобы все Composable-функции были без состояния: все входы передаются параметрами, и они используются только для получения результата. Так Composable становятся проще, «тупее» и хорошо переиспользуемыми. Но для написания программ с состоянием побочные эффекты нужны, так что на каком-то уровне их придётся вызывать (часто в корне composable-дерева). Программам нужны сетевые запросы, сохранение в БД, кэши в памяти и т.д. Поэтому Jetpack Compose предлагает механизмы безопасного вызова эффектов из Composable-функций в контролируемой среде — **обработчики эффектов (effect handlers)**.

Обработчики эффектов привязывают побочные эффекты к жизненному циклу Composable: эффекты могут быть ограничены им или управляться им. Они позволяют автоматически отменять/останавливать эффекты при выходе Composable из дерева, перезапускать их при изменении входов эффекта или растягивать один и тот же эффект на несколько выполнений (recomposition), так что он вызывается только один раз. Обработчики эффектов мы разберём в следующих главах. Они позволяют не вызывать эффекты напрямую из тела Composable без какого-либо контроля.

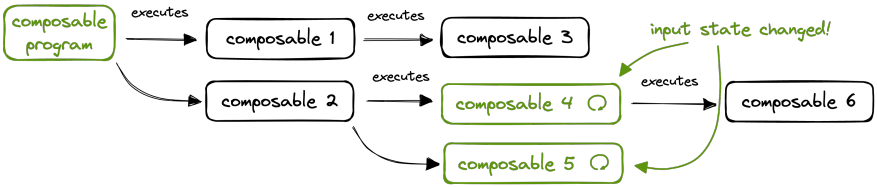
Restartable (перезапускаемость)

Мы уже упоминали это несколько раз. Composable-функции могут перезапускаться (recompose), то есть они не как обычные функции, которые вызываются один раз в рамках стека вызовов. Обычный стек вызовов выглядит так: каждая функция вызывается один раз и может вызвать одну или несколько других.



Обычный стек вызовов (схема)

С другой стороны, Composable-функции могут перезапускаться (выполняться заново, recompose) многократно, и runtime хранит на них ссылки для этого. Вот как может выглядеть дерево вызовов Composable:



Дерево вызовов Composable с перезапуском (схема)

Composable 4 и 5 выполняются заново после изменения их входов.

Compose выборочно решает, какие узлы дерева перезапускать, чтобы представление в памяти оставалось актуальным. Composable-функции спроектированы реактивными и перезапускаются при изменении наблюдаемого ими состояния.

Компилятор Compose находит все Composable-функции, читающие какое-либо состояние, и генерирует код, который сообщает runtime, как их перезапускать. Composable, не читающие состояние, перезапускать не нужно, поэтому нет смысла сообщать runtime, как это делать.

Быстрое выполнение

Composable-функции и дерево Composable-функций можно рассматривать как быстрый, декларативный и лёгкий способ построить описание программы, которое сохраняется в памяти и интерпретируется / материализуется на более позднем этапе.

Composable-функции не строят и не возвращают UI. Они лишь эмитируют данные для построения или обновления структуры в памяти. Поэтому они очень быстрые, и runtime может выполнять их многократно без опасений — иногда очень часто, например на каждом кадре анимации.

Разработчики должны соответствовать этому ожиданию. Любые тяжёлые вычисления нужно выносить в корутины и оборачивать в один из привязанных к жизненному циклу обработчиков эффектов, о которых мы узнаем дальше в книге.

Позиционная мемоизация

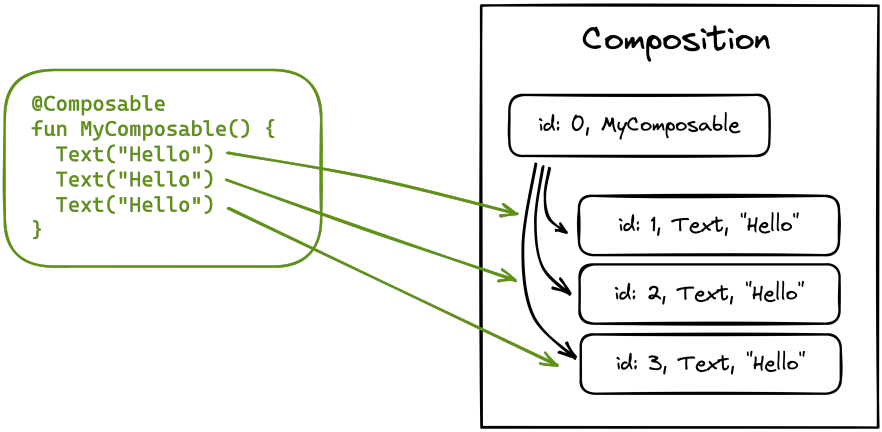
Позиционная мемоизация — разновидность мемоизации функций. Мемоизация функции — способность кэшировать результат по входам, чтобы не вычислять его заново при каждом вызове с теми же аргументами. Как мы уже видели, это возможно только для чистых (**детерминированных**) функций: мы уверены, что для одних и тех же входов результат всегда один и тот же, и его можно кэшировать и переиспользовать.

Мемоизация функций — широко известная техника в парадигме функционального программирования, где программы определяются как композиция чистых функций.

При мемоизации функций вызов можно идентифицировать по комбинации имени, типа и значений параметров. По этим элементам строится уникальный ключ для сохранения/индексации/чтения кэшированного результата при последующих вызовах. В Compose учитывается ещё один элемент: Composable-функции всегда «знают» **своё расположение в исходном коде**. Runtime будет генерировать разные id (уникальные в рамках родителя), когда одна и та же функция вызывается с теми же параметрами из разных мест:

```
1 @Composable
2 fun MyComposable() {
3   Text(&amp;quot;Hello&amp;quot;) // id 1
4   Text(&amp;quot;Hello&amp;quot;) // id 2
5   Text(&amp;quot;Hello&amp;quot;) // id 3
6 }
```

В памяти будет храниться три разных экземпляра, каждый со своей идентичностью.



Позиционная мемоизация (схема)

Идентичность Composable сохраняется между recomposition, так что runtime может обращаться к этой структуре, чтобы понять, вызывался ли Composable ранее, и при возможности пропустить его.

Иногда присвоить уникальные идентичности сложно. Один из примеров — списки Composable, порождённые циклом:

```
1 @Composable
2 fun TalksScreen(talks: List&lt;Talk&gt;) {
3   Column {
4     for (talk in talks) {
5       Talk(talk)
6     }
7   }
8 }
```

Здесь `Talk(talk)` каждый раз вызывается из одной и той же позиции, но каждый talk — другой элемент списка и значит другой узел в дереве. В таких случаях Compose runtime опирается на **порядок вызовов** для генерации уникального id и различения элементов. Это хорошо работает при добавлении элемента в конец списка — остальные вызовы остаются на тех же позициях. Но если добавлять элементы в начало или в середину? Тогда runtime перезапустит все `Talk` ниже точки вставки, так как их позиции сдвинулись, даже если их входы не изменились. Это неэффективно (особенно для длинных списков), ведь эти вызовы можно было бы пропустить.

Чтобы решить эту проблему, Compose предоставляет Composable `key` для явного присвоения ключа вызову:

```
@Composable
fun TalksScreen(talks: List&lt;Talk&gt;) {
  Column {
    for (talk in talks) {
      key(talk.id) { // Unique key
        Talk(talk)
      }
    }
  }
}
```

В этом примере мы используем id доклада (предположительно уникальный) как ключ для каждого `Talk` — это позволит runtime сохранять идентичность всех элементов списка **независимо от позиции**.

Позиционная мемоизация позволяет runtime по замыслу «запоминать» Composable-функции. Любая Composable-функция, которую компилятор Compose считает restartable, должна также быть skippable и тем самым **автоматически запоминаемой**. Compose построен на этом механизме.

Иногда разработчикам нужен доступ к этой структуре в памяти на уровне более мелком, чем scope одной Composable-функции. Например, нужно закэшировать результат тяжёлого вычисления внутри Composable. Для этого Compose runtime предоставляет функцию `remember`:

```
@Composable
fun FilteredImage(path: String) {
  val filters = remember { computeFilters(path) }
  ImageWithFiltersApplied(filters)
}

@Composable
fun ImageWithFiltersApplied(filters: List&lt;Filter&gt;) {
  TODO()
}
```

Здесь мы используем `remember`, чтобы закэшировать результат предвычисления фильтров изображения. Ключ для индексации кэшированного значения строится по позиции вызова в исходниках и по входу функции (в данном

случае путь к файлу). Функция `remember` — это просто Composable-функция, которая умеет читать и писать в структуру в памяти, хранящую состояние дерева. Она лишь выставляет механизм «позиционной мемоизации» разработчику.

В Compose мемоизация не распространяется на всё приложение. Когда что-то мемоизируется, это делается в контексте вызывающего это Composable. В примере выше это `FilteredImage`. На практике Compose обращается к структуре в памяти и ищет значение в диапазоне слотов, где хранится информация об охватывающем Composable. Получается что-то вроде **одиночки в рамках этого scope**. Если тот же Composable вызывался бы из другого родителя, вернулся бы новый экземпляр значения.

Сходство с suspend-функциями

Kotlin-функции с `suspend` можно вызывать только из других `suspend`-функций, то есть им тоже нужен контекст вызова. Это гарантирует, что `suspend`-функции можно объединять в цепочки и даёт компилятору Kotlin возможность внедрять и передавать среду выполнения по всем уровням вычислений. Эта среда добавляется к каждой `suspend`-функции как дополнительный параметр в конце списка — `Continuation`. Параметр тоже неявный, так что разработчик может о нём не думать. `Continuation` используется для включения новых возможностей языка.

Звучит знакомо?

В системе корутин Kotlin `Continuation` подобен колбэку: он указывает программе, как продолжить выполнение.

Пример. Код вида:

```
1 suspend fun publishTweet(tweet: Tweet): Post = ...
```

компилятор Kotlin заменяет на:

```
1 fun publishTweet(tweet: Tweet, callback:
 ↪  Continuation&lt;Post&gt;): Unit
```

`Continuation` несёт всю информацию, которая нужна runtime Kotlin для приостановки и возобновления выполнения в точках suspension в программе. Так `suspend` — ещё один хороший пример того, как требование контекста вызова может служить способом передачи неявной информации по дереву

выполнения — информации, которую runtime может использовать для продвинутых возможностей языка.

Аналогично можно воспринимать `@Composable` как возможность языка: она делает обычные Kotlin-функции перезапускаемыми, реактивными и т.д.

Резонный вопрос: почему команда Jetpack Compose не использовала `suspend` для нужного поведения? Хотя оба механизма реализуют похожий паттерн, они дают совершенно разные возможности. Интерфейс `Continuation` жёстко завязан на приостановку и возобновление выполнения и смоделирован как интерфейс колбэка; Kotlin генерирует для него реализацию со всей необходимой логикой для переходов, координации точек suspension, обмена данными и т.д. Сценарий Compose другой: цель — создать представление в памяти большого графа вызовов, который можно по-разному оптимизировать во время выполнения.

Поняв сходство Composable и `suspend`-функций, полезно вспомнить идею «раскраски функций».

«Цвет» Composable-функций

У Composable-функций другие ограничения и возможности, чем у обычных функций. У них другой тип (об этом позже), и они моделируют свою специфическую задачу. Эту дифференциацию можно понимать как «раскраску функций» — они образуют отдельную **категорию функций**.

«Раскраска функций» — концепция, которую объяснил Боб Нистром из команды Dart в Google в посте «What color is your function?» (2015). Он описал, как async- и sync-функции плохо комбинируются: нельзя вызвать async-функцию из sync без того, чтобы сделать вызывающую тоже async или предоставить механизм ожидания (await). Поэтому в языках и библиотеках появились Promises и `async/await` — попытка вернуть композируемость. Боб называет эти две категории разными «цветами» функций.

В Kotlin ту же задачу решает `suspend`. Но `suspend`-функции тоже «окрашены»: вызывать их можно только из других `suspend`-функций. Комбинировать программу из обычных и `suspend`-функций можно только через точки интеграции (запуск корутин). Интеграция для разработчика не прозрачна.

В целом такое ограничение ожидаемо. Мы моделируем две категории функций, представляющие концепции разной природы — как два разных языка: операции с немедленным результатом (sync) и операции, разворачивающиеся во времени и в итоге дающие результат (async), зачастую с задержкой.

В Jetpack Compose случай Composable-функций аналогичен. Вызывать Composable-функции из обычных функций напрямую нельзя. Для этого нужна точка интеграции (например, `Composition.setContent`). У Composable-функций цель совсем не та, что у обычных: они предназначены не для программной логики, а для описания изменений дерева узлов.

Может показаться, что я немного лукавлю. Одно из преимуществ Composable-функций — возможность описывать UI с помощью логики, то есть иногда нужно вызывать Composable из обычных функций. Например:

```
1 @Composable
2 fun SpeakerList(speakers: List&lt;Speaker&gt;) {
3   Column {
4     speakers.forEach {
5       Speaker(it)
6     }
7   }
8 }
```

Composable `Speaker` вызывается из лямбды `forEach`, и компилятор не ругается. Как тогда возможно смешивать «цвета» функций?

Причина — `inline`. Операторы коллекций объявлены как `inline`, поэтому их лямбды подставляются в вызывающий код, и по сути дополнительного уровня косвенности нет. В примере выше вызов Composable `Speaker` оказывается встроенным в тело `SpeakerList`, и это допустимо, так как оба — Composable-функции. Используя `inline`, мы обходим проблему раскраски и можем писать логику внутри Composable. Дерево по-прежнему будет состоять только из Composable-функций.

Но действительно ли раскраска — проблема?

Она могла бы быть проблемой, если бы приходилось постоянно переключаться между двумя типами функций. Но для `suspend` и `@Composable` это не так. Оба механизма требуют точки интеграции, и за ней получается полностью «окрашенный» стек вызовов (всё либо `suspend`, либо Composable). Это скорее преимущество: компилятор и runtime могут по-разному обрабатывать окрашенные функции и включать продвинутые возможности языка, недоступные для обычных функций.

В Kotlin `suspend` позволяет идиоматично и выразительно описывать асинхронные неблокирующие программы; язык получает способ представить сложную концепцию очень просто — модификатором `suspend` у функций. С

другой стороны, `@Composable` делает обычные функции перезапускаемыми, пропускаемыми и реактивными — возможностями, которых у обычных Kotlin-функций нет.

Типы Composable-функций

Аннотация `@Composable` на этапе компиляции фактически меняет тип функции. С точки зрения синтаксиса тип Composable-функции — `@Composable (T) -> A`, где `A` может быть `Unit` или любым другим типом, если функция возвращает значение (например, `remember`). Разработчики могут использовать этот тип для объявления Composable-лямбд так же, как обычных лямбд в Kotlin.

```
// This can be reused from any Composable tree
val textComposable: @Composable (String) -&gt; Unit = {
  Text(
    text = it,
    style = MaterialTheme.typography.subtitle1
  )
}

@Composable
fun NamePlate(name: String, lastname: String) {
  Column(modifier = Modifier.padding(16.dp)) {
    Text(
      text = name,
      style = MaterialTheme.typography.h6
    )
    textComposable(lastname)
  }
}
```

У Composable-функций может быть и тип `@Composable Scope.() -> A`, часто используемый для ограничения информации scope'ом конкретного Composable:

```
inline fun Box(
  ...,
  content: @Composable BoxScope.() -&gt; Unit
) {
  // ...
  Layout(
```

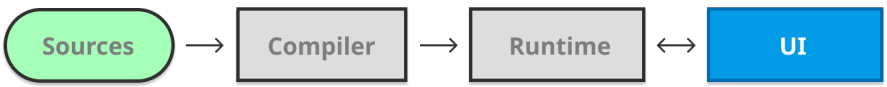
```
    content = { BoxScopeInstance.content() },
    measurePolicy = measurePolicy,
    modifier = modifier
  )
}
```

С точки зрения языка типы нужны, чтобы давать компилятору информацию для статической проверки, иногда генерировать удобный код и ограничивать/уточн использование данных во время выполнения. Аннотация `@Composable` меняет то, как функция проверяется и используется в runtime, поэтому считается, что у неё другой тип, чем у обычной функции.

2. Компилятор Compose

Jetpack Compose состоит из набора библиотек, но в этой книге основное внимание уделяется трём: компилятору Compose, рантайму Compose и Compose UI.

Компилятор Compose и рантайм — основа Jetpack Compose. Compose UI технически не входит в архитектуру Compose: рантайм и компилятор спроектированы универсально и могут использоваться любыми клиентскими библиотеками, соблюдающими их требования. Compose UI — лишь один из доступных клиентов. Есть и другие клиентские библиотеки в разработке, например для десктопа и веба от JetBrains. Тем не менее разбор Compose UI поможет понять, как Compose наполняет in-memory представление composable-дерева в рантайме и как в итоге материализует из него реальные элементы.



Архитектура Compose

При первом знакомстве с Compose может возникнуть путаница с точным порядком вещей. До этого момента в книге мы слышали, что компилятор и рантайм работают вместе и раскрывают возможности библиотеки, но без предварительного опыта это остаётся довольно абстрактным. Было бы полезно глубже разобраться: какие действия выполняет компилятор Compose,

чтобы код соответствовал требованиям рантайма, как работает рантайм, когда запускаются первичная composition и последующие recomposition, как формируется in-memory представление дерева и как эта информация используется для дальнейших recomposition. Понимание этого даёт общую картину работы библиотеки и ожиданий от неё при написании кода.

Приступим — начнём с компилятора.

Плагин компилятора Kotlin

Jetpack Compose частично опирается на генерацию кода. В мире Kotlin и JVM для этого обычно используют процессоры аннотаций через **kapt**, но Jetpack Compose устроен иначе: компилятор Compose — это плагин компилятора Kotlin. Это позволяет встроить работу на этапе компиляции в фазы компиляции Kotlin, получать более полную информацию о структуре кода и ускорять процесс в целом. kapt запускается до компиляции, а плагин компилятора **встраивается в процесс компиляции**.

Плагин компилятора Kotlin также даёт возможность сообщать диагностику на фронтенд-фазе компилятора, обеспечивая быструю обратную связь. Однако эта диагностика не показывается в IDE, так как IDEA не интегрирована с плагином напрямую. Все инспекции уровня IDEA в Compose добавлены отдельным плагином IDEA, не разделяющим код с плагином компилятора Compose. Диагностика фронтенда выдаётся сразу при нажатии кнопки компиляции. Ускорение обратной связи — главная польза статического анализа на фронтенд-фазе компилятора Kotlin, и компилятор Jetpack Compose этим активно пользуется.

Ещё одно важное преимущество плагинов компилятора Kotlin — они могут по своему усмотрению изменять существующие исходники (не только добавлять новый код, как процессоры аннотаций). Они способны модифицировать выходной IR для этих элементов до его понижения до более атомарных конструкций, которые затем переводятся в примитивы целевых платформ — помним, Kotlin мультиплатформенный. В этой главе мы углубимся в детали; в итоге компилятор Compose получает возможность **трансформировать Composable-функции** так, как того требует рантайм.

У плагинов компилятора Kotlin большое будущее. Многие известные процессоры аннотаций, вероятно, постепенно переедут в плагины компилятора или «лёгкие» плагины через KSP (Kotlin Symbol Processing). См. блок ниже.

Если вам интересны плагины компилятора Kotlin, рекомендую ознакомиться

с KSP (Kotlin Symbol Processing) — библиотекой, которую Google предлагает как замену Kapt. KSP предлагает нормализованный DSL для «написания лёгких плагинов компилятора», на который могут опираться библиотеки для метапрограммирования. Обязательно прочитайте раздел «Why KSP» в репозитории KSP.

Также учтите: компилятор Jetpack Compose сильно опирается на трансформации IR, и их массовое использование в метапрограммировании может быть рискованным. Если бы все процессоры аннотаций стали плагинами компилятора, трансформаций IR могло бы стать слишком много и это могло бы дестабилизировать язык. Изменение и расширение языка всегда несёт риск. Поэтому в общем случае KSP — более предпочтительный выбор.

Аннотации Compose

Вернёмся к порядку вещей. Сначала нужно понять, как мы аннотируем код, чтобы компилятор мог находить нужные элементы и делать свою работу. Начнём с доступных аннотаций Compose.

Даже если плагины компилятора умеют больше, чем процессоры аннотаций, у них есть общее. Один пример — фронтенд-фаза, часто используемая для статического анализа и проверок.

Компилятор Compose использует хуки/точки расширения во фронтенде компилятора Kotlin, чтобы проверять соблюдение накладываемых ограничений и то, что система типов правильно отличает функции, объявления и выражения с `@Composable` от не-Composable. Помимо этого Compose предоставляет дополнительные аннотации для дополнительных проверок и разных оптимизаций или «укорочений» в рантайме в определённых случаях. Все аннотации поставляются библиотекой Compose runtime.

Разберём самые важные аннотации.

Все аннотации Jetpack Compose предоставляются Compose runtime, так как и компилятор, и рантайм активно их используют.

@Composable

Эта аннотация уже подробно разобрана в главе 1. Тем не менее ей нужен отдельный подраздел — она явно главная. Поэтому она первая в списке.

Главное отличие компилятора Compose от процессора аннотаций в том, что Compose фактически **изменяет** объявление или выражение, к

которому применена аннотация. Большинство процессоров аннотаций так не умеют — они создают дополнительные или соседние объявления. Поэтому компилятор Compose использует трансформации IR. Аннотация `@Composable` действительно **меняет тип** элемента, а плагин компилятора обеспечивает соблюдение правил во фронтенде, чтобы типы с Composable не обрабатывались так же, как их не-Composable аналоги.

Изменение типа объявления или выражения через `@Composable` даёт ему «память»: возможность вызывать `remember` и использовать Composer/slot table. Также появляется жизненный цикл, с которым могут согласовываться эффекты, запускаемые в теле (например, продолжение job через recomposition). Composable-функции получают идентичность, которую сохраняют, и позицию в результирующем дереве — то есть могут эмитировать узлы в Composition и обращаться к `CompositionLocal`.

Кратко: Composable-функция — это отображение данных в узел, который эмитируется в дерево при выполнении. Узел может быть UI-узлом или узлом любой другой природы в зависимости от библиотеки, потребляющей Compose runtime. Jetpack Compose runtime работает с **обобщёнными типами узлов**, не привязанными к конкретному сценарию или семантике. Эту тему мы подробно разберём ближе к концу книги.

@ComposeCompilerApi

Этой аннотацией Compose помечает части API, предназначенные только для компилятора — чтобы сообщить об этом пользователям и предупредить о необходимости осторожного использования.

@InternalComposeApi

Некоторые API помечены в Compose как внутренние, так как они могут меняться внутри, даже если публичная поверхность API остаётся неизменной и замороженной к стабильному релизу. У этой аннотации шире область, чем у ключевого слова `internal` в языке: она допускает использование между модулями, чего в Kotlin нет.

@DisallowComposableCalls

Используется, чтобы запретить вызовы Composable внутри функции. Полезно для параметров-лямбд с `inline` у Composable-функций, внутри которых нельзя безопасно вызывать Composable. Лучше всего подходит для лямбд, которые **не** вызываются при каждой recomposition.

Пример — функция `remember` из Compose runtime. Эта Composable-функция запоминает значение, вычисленное блоком `calculation`. Блок вычисляется только при первичной composition, при дальнейших recomposition всегда возвращается уже вычисленное значение.

```
1 @Composable
2 inline fun &lt;T&gt; remember(calculation:
 ↪  @DisallowComposableCalls () -&gt; T): T =
3    currentComposer.cache(false, calculation)
```

Composables.kt

Благодаря этой аннотации вызовы Composable внутри лямбды `calculation` запрещены. Иначе при вызове они занимали бы место в slot table (эмитируя), а после первой composition эта область освобождалась бы, так как лямбда больше не вызывается.

Аннотацию лучше использовать для inline-лямбд, вызываемых условно как деталь реализации и не призванных быть «живыми», как composable. Это нужно потому, что inline-лямбды особенные: они «наследуют» способности Composable от контекста вызова. Например, лямбда вызова `forEach` не помечена как `@Composable`, но внутри неё можно вызывать Composable-функции, если сам `forEach` вызывается из Composable-функции. Для `forEach` и многих других inline API это желательно, но **не** желательно в случаях вроде `remember` — здесь и нужна эта аннотация.

Аннотация «заразная»: если вы вызываете inline-лямбду внутри inline-лямбды с `@DisallowComposableCalls`, компилятор потребует пометить и эту лямбду как `@DisallowComposableCalls`.

Как можно догадаться, в клиентских проектах эту аннотацию, скорее всего, использовать не придётся; она может пригодиться, если вы используете Jetpack Compose не для Compose UI и пишете свою клиентскую библиотеку для рантайма с соблюдением его ограничений.

@ReadOnlyComposable

В применении к Composable-функции означает: мы знаем, что тело этого Composable никогда не будет писать в composition, только читать. То же должно быть верно для всех вложенных вызовов Composable в теле. Это позволяет рантайму не генерировать код, который не понадобится, если Composable действительно соблюдает это условие.

Для любого Composable, который пишет в composition, компилятор генерирует «группу», оборачивающую его тело, и в рантайме эмитируется вся группа. Эти группы несут информацию о Composable в composition: как очистить записанные данные при замене при recomposition другим Composable или как перемещать данные, сохраняя идентичность Composable. Генерируются разные типы групп: restartable, movable и т.д.

Чтобы понять, что такое «группа», представьте пару указателей на начало и конец выделенного фрагмента текста. У всех групп есть ключ позиции в исходниках для хранения группы — он и обеспечивает позиционную мемоизацию. По нему же различаются идентичности веток `if` и `else` в условной логике:

```
1 if (condition) {
2   Text(&amp;quot;Hello&amp;quot;)
3 } else {
4   Text(&amp;quot;World&amp;quot;)
5 }
```

ConditionalTexts.kt

Оба вызова — `Text`, но у них разная идентичность, так как для вызывающего кода это разные сущности. У movable-групп есть ещё семантический ключ идентичности для переупорядочивания внутри родительской группы.

Когда наш Composable не пишет в composition, генерация таких групп не даёт пользы — его данные не будут заменяться или перемещаться. Эта аннотация помогает этого избежать.

Примеры read-only Composable в библиотеках Compose: многие значения по умолчанию и утилиты для `CompositionLocal` (например, Material `Colors`, `Typography`), функция `isSystemInDarkTheme()`, `LocalContext`, вызовы для получения `resources` приложения любого типа (они опираются на `LocalContext`), `LocalConfiguration`. В общем, это вещи, которые задаются один раз при запуске программы и остаются неизменными и доступными для чтения из Composable в дереве.

@NonRestartableComposable

В применении к функции или геттеру свойства делает её не перезапускаемым (non-restartable) Composable. (Не все Composable по умолчанию restartable: inline Composable или Composable с типом возврата не-`Unit` таковыми не являются.)

При добавлении аннотации компилятор не генерирует шаблонный код, позволяющий функции перезапускаться при recomposition или пропускаться. Использовать её нужно очень сдержанно: только для очень маленьких функций, которые, скорее всего, перезапускаются (restart) вызывающим их Composable, так как содержат мало логики и самостоятельная инвалидация для них не имеет смысла. Их инвалидация/recomposition по сути управляется родительским/охватывающим Composable.

Эта аннотация редко или никогда не нужна для «корректности», но может служить лёгкой оптимизацией производительности, если известно, что такое поведение даст выигрыш — иногда так и есть.

@StableMarker

Compose runtime также предоставляет аннотации для обозначения стабильности типа: мета-аннотацию `@StableMarker` и аннотации `@Immutable` и `@Stable`. Начнём с `@StableMarker`.

`@StableMarker` — мета-аннотация, которой аннотируют другие аннотации, например `@Immutable` и `@Stable`. Звучит избыточно, но цель — переиспользовани её следствия распространяются на все аннотации, помеченные ею.

`@StableMarker` подразумевает следующие требования к стабильности данных для помеченного типа:

- Результат вызовов `equals` для одних и тех же двух экземпляров всегда одинаков.
- Composition всегда уведомляется при изменении публичного свойства помеченного типа.
- Все публичные свойства помеченного типа тоже стабильны.

Типы, помеченные `@Immutable` или `@Stable`, должны удовлетворять этим требованиям, так как обе аннотации помечены как `@StableMarker`, то есть как маркеры стабильности.

Это обещания компилятору для допущений при обработке исходников; **на этапе компиляции они не проверяются**. Решать, когда все требования выполнены, остаётся разработчику.

Компилятор Compose по возможности выводит стабильность типов и обрабатывает их как стабильные без явной аннотации. Во многих случаях это предпочтительно и гарантированно корректно. Явная аннотация важна в двух случаях:

- Когда это контракт/ожидание интерфейса или абстрактного класса: аннотация становится не только обещанием компилятору, но и **требованием** к реализации (к сожалению, никак не проверяемым).
- Когда реализация мутабельна, но мутабельность безопасна в рамках допущений стабильности. Типичный пример — тип мутабелен из-за внутреннего кэша, но публичный API типа не зависит от состояния кэша.

Вывод стабильности классов подробнее разбирается ниже в этой главе.

@Immutable

Применяется к классу как строгое обещание компилятору: все публично доступные свойства и поля класса не изменяются после создания. Это **более сильное** обещание, чем ключевое слово `val`: `val` лишь запрещает переназначение через сеттер, но свойство может ссылаться на мутабельную структуру, так что данные остаются мутабельными при одних только `val`. Это нарушило бы ожидания Compose runtime. Иными словами, аннотация нужна Compose потому, что в Kotlin нет механизма (ключевого слова и т.п.) гарантии неизменности структуры данных.

Исходя из допущения, что значения, читаемые из типа, после инициализации не меняются, рантайм может применять оптимизации к умной recomposition и пропуску recomposition.

Хороший пример класса, который можно безопасно пометить `@Immutable`, — `data class` только с `val`-свойствами, без кастомных геттеров (иначе значение вычислялось бы при каждом вызове и могло бы отличаться, делая API нестабильным для чтения), причём все свойства — примитивы или типы, тоже помеченные `@Immutable`.

`@Immutable` также является `@StableMarker`, поэтому наследует все его следствия. Неизменяемый тип всегда удовлетворяет требованиям `@StableMarker`, так как его публичные значения не меняются. Аннотация `@Immutable` существует, чтобы помечать неизменяемые типы как стабильные.

Дополнение: неизменяемым типам не нужно уведомлять composition об изменении значений (одно из требований `@StableMarker`), потому что значения не меняются — ограничение и так выполняется.

@Stable

Эта аннотация — более мягкое обещание, чем `@Immutable`. Её смысл зависит от элемента языка, к которому она применена.

При применении к типу означает, что тип **мутабелен** (иначе использовали бы `@Immutable`) и подчиняется только следствиям `@StableMarker`. При необходимости перечитайте их.

Когда `@Stable` применена к функции или свойству, она сообщает компилятору, что функция всегда возвращает один и тот же результат для одних и тех же входов (чистая). Это возможно только когда параметры функции тоже `@Stable`, `@Immutable` или примитивы (они считаются стабильными).

В документации есть наглядный пример важности для рантайма: *Когда все типы, передаваемые параметрами в Composable-функцию, помечены как стабильные, значения параметров сравниваются на равенство по позиционной мемоизации и вызов пропускается, если все значения равны предыдущему вызову.*

Пример типа, который можно пометить `@Stable`, — объект, чьи публичные свойства не меняются, но который нельзя считать неизменяемым (например, есть приватное мутабельное состояние или делегирование свойства в MutableState, но снаружи он используется как неизменяемый).

Следствия этой аннотации компилятор и рантайм используют для допущений об эволюции (или отсутствии эволюции) данных и для сокращений там, где нужно. Аннотацию не стоит использовать, пока вы не уверены, что её условия выполнены. Иначе мы даём компилятору неверную информацию, что легко приведёт к ошибкам в рантайме. Поэтому все эти аннотации рекомендуется применять сдержанно.

Интересно: хотя `@Immutable` и `@Stable` — разные обещания с разным смыслом, компилятор Jetpack Compose **сейчас обрабатывает их одинаково** — для включения и оптимизации умной recomposition и пропуска recomposition. Обе аннотации существуют, чтобы в будущем при необходимости можно было разделить семантику для компилятора и рантайма.

Регистрация расширений компилятора

Раз мы познакомились с основными аннотациями **от рантайма**, пора понять, как работает плагин компилятора Compose и как он их использует.

Первое, что делает плагин, — регистрируется в пайплайне компилятора Kotlin через `ComponentRegistrar` (механизм компилятора Kotlin для этого). `ComposeComponentRegistrar` регистрирует ряд расширений компилятора для разных целей. Эти расширения обеспечивают удобство использования библиотеки и генерацию кода для рантайма. Все зарегистрированные расширения выполняются вместе с компилятором Kotlin.

Компилятор Compose также регистрирует расширения в зависимости от включённых флагов. Разработчики Jetpack Compose могут включать флаги для live literals, добавления информации об исходниках в сгенерированный код (для инспекции composition в Android Studio и других инструментах), оптимизаций для функций remember, подавления проверок совместимости версий Kotlin и/или генерации decoy-методов при трансформации IR.

Чтобы глубже разобраться в регистрации расширений плагином или в других деталях, можно смотреть исходники на cs.android.com.

Версия компилятора Kotlin

Компилятор Compose требует строго определённую версию Kotlin и проверяет её совпадение. Это первая проверка, так как при несовпадении компиляция блокируется.

Проверку можно отключить аргументом компилятора `suppressKotlinVersionCo`
но на свой риск: тогда Compose можно запускать с любой версией Kotlin, что легко приведёт к несовместимостям, особенно с учётом эволюции бэкендов компилятора Kotlin. Параметр, вероятно, добавлен для запуска и тестирования Compose на экспериментальных сборках Kotlin.

Статический анализ

Как и у типичного плагина компилятора, первым идёт **проверка** (linting). Статический анализ выполняется сканированием исходников в поисках аннотаций библиотеки и важных проверок корректности использования — в том смысле, как ожидает рантайм. Предупреждения и ошибки выдаются через context trace, к которому имеют доступ плагины компилятора. Это хорошо стыкуется с **IDEA**, которая умеет показывать их прямо в коде при наборе. Как упоминалось, все проверки выполняются на фронтенд-фазе компилятора, что даёт максимально быструю обратную связь.

Разберём некоторые из главных статических проверок.

Статические проверяющие (Static Checkers)

Часть зарегистрированных расширений — статические проверяющие, направляющие разработчика при написании кода. Compose регистрирует проверяющие вызовов, типов и объявлений. Они обеспечивают корректное использование библиотеки и ориентированы на решаемые ею задачи. Здесь проверяются требования к Composable-функциям из главы 1, при нарушении выдаются сообщения.

В компиляторах Kotlin есть разные анализаторы в зависимости от проверяемого элемента: инстанциация классов, типы, вызовы функций, устаревшие вызовы, контракты, захват в замыкании, инфиксные вызовы, вызовы корутин, операторные вызовы и др. — плагины могут анализировать соответствующие элементы исходников и сообщать информацию, предупреждения или ошибки.

Так как все проверяющие выполняются на фронтенд-фазе, они должны быть быстрыми и не содержать тяжёлых операций. Это ответственность разработчика плагина — проверки запускаются при наборе кода, и важно не создавать подвисаний.

Проверки вызовов

Один из видов проверяющих Compose — проверки вызовов. Компилятор Compose выполняет статические проверки вызовов Composable-функций в разных контекстах, например внутри scope с `@DisallowComposableCalls` или `@ReadOnlyComposable`.

Проверяющий вызовов — расширение компилятора для статического анализа всех вызовов в коде. У него есть функция `check`, рекурсивно вызываемая для обхода всех PSI-элементов, считающихся вызовами (всех узлов дерева PSI). Это реализация паттерна посетителя.

Некоторым проверкам нужен контекст шире текущего элемента: например, откуда вызывается Composable. Одного узла PSI недостаточно. Для сбора информации при обходе накапливаются мелкие фрагменты в **trace** контекста, как «хлебные крошки», чтобы затем выполнять более сложные проверки. Компилятор записывает эту информацию в контекстный **trace**, расширяя область проверок и позволяя искать охватывающие лямбды, блоки try/catch и т.п.

Пример вызова в компиляторе, записывающего информацию в trace и сообщающего об ошибке при вызове Composable в контексте с `@DisallowComposabl`

```
if (arg?.type?.hasDisallowComposableCallsAnnotation() == true) {
  context.trace.record(
    ComposeWritableSlices.LAMBDA_CAPABLE_OF_COMPOSER_CAPTURE,
    descriptor, // reference to the function literal
    false
  )
  context.trace.report(
    ComposeErrors.CAPTURED_COMPOSABLE_INVOCATION.on(
      reportOn,
      arg,
      arg.containingDeclaration
    )
  )
  return
}
```

ContextTraceExamples.kt

Контекст и context trace доступны при каждом вызове `check`; через тот же trace можно сообщать об ошибках, предупреждениях и информационных сообщениях. Trace можно представлять как мутабельную структуру, которую заполняют информацией для использования по ходу анализа.

Другие проверки проще и используют только данные текущего элемента; они выполняют действие и возвращают управление. При каждом вызове `check` плагин сопоставляет тип текущего узла и в зависимости от него выполняет проверку и возврат (если всё верно), сообщает об ошибке, записывает данные в trace или рекурсивно переходит к родителю. По пути выполняются разные проверки для разных аннотаций.

Компилятор Compose проверяет, что Composable не вызываются из запрещённых мест: из блока `try/catch` (не поддерживается), из функции без аннотации Composable или из лямбд с `@DisallowComposableCalls`. (Та аннотация как раз запрещала вызовы Composable внутри inline-лямбд.)

Для каждого вызова Composable компилятор поднимается по дереву PSI, проверяя вызывающих и их вызывающих, и убеждается, что все требования для этого вызова выполнены. Учитываются все сценарии: родители могут быть лямбдами, функциями, свойствами, аксессорами, блоками try/catch, классами, файлами и т.д.

PSI моделирует структуру языка для фронтенд-фаз компилятора, поэтому

понимание кода полностью синтаксическое и статическое.

Важно учитывать `inline`-функции: вызов Composable из inline-лямбды допустим **только если вызывающие эту лямбду тоже Composable**. Компилятор проверяет, что любая inline-лямбда, вызывающая Composable, на каком-то уровне вверх по стеку вызовов охвачена Composable-функцией.

Ещё одна проверка вызовов — обнаружение потенциально пропущенных аннотаций `@Composable` там, где они нужны (например, если Composable вызывается внутри лямбды, компилятор может предложить добавить аннотацию к лямбде). Статический анализ не только запрещает — он может подсказать, что нужно, или как улучшить код.

Есть проверки для Composable-функций с `@ReadOnlyComposable`: они могут вызывать только другие read-only Composable, иначе нарушается контракт оптимизации (read-only Composable только читает из composition, никогда не пишет). Так как это должно выполняться на всех уровнях вложенности, паттерн посетителя здесь уместен.

Ещё одна проверка — запрет использования ссылок на Composable-функции, так как Jetpack Compose это пока не поддерживает.

Проверки типов

Иногда мы аннотируем как Composable не только функции, но и типы. Для этого у компилятора Compose есть проверка, связанная с выводом типов: он сообщает, когда ожидался тип с `@Composable`, а найден тип без аннотации. Аналогично проверке вызовов выше. В сообщении выводятся ожидаемый и выведенный типы с аннотациями для наглядности.

Проверки объявлений

Проверок мест вызова и типов недостаточно — нужны и проверки мест объявления. Нужно анализировать свойства, аксессоры свойств, объявления функций и параметры функций.

Свойства, геттеры и функции могут переопределяться, в том числе с аннотацией Composable. Компилятор Compose проверяет, что любые переопределения таких элементов тоже помечены как Composable, для согласованности.

Ещё одна проверка объявлений — Composable-функции не должны быть `suspend`, это не поддерживается. Как в главе 1, у `suspend` иной смысл, чем у

`@Composable`; оба можно считать примитивами языка, но они представляют разные вещи и пока не поддерживаются вместе.

Объявлениями также запрещаются Composable-функция `main` и backing field у Composable-свойств.

Подавление диагностики

Плагины компилятора могут регистрировать подавители диагностики, чтобы в определённых случаях глушить сообщения (например, ошибки статических проверок). Так бывает, когда плагины генерируют или поддерживают код, который компилятор Kotlin обычно не принял бы — подавитель обходит соответствующие проверки.

Compose регистрирует `ComposeDiagnosticSuppressor` для обхода некоторых ограничений языка, иначе блокирующих компиляцию, чтобы разрешить отдельные сценарии.

Одно из ограничений — inline-лямбды с «не-исходными» аннотациями в местах вызова (аннотации с retention `BINARY` или `RUNTIME`). Такие аннотации сохраняются в бинарниках, в отличие от `SOURCE`. Inline-лямбды на этапе компиляции подставляются в вызывающий код и нигде не хранятся, так что аннотировать уже нечего. Поэтому Kotlin запрещает это и выдаёт:

«The lambda expression here is an inlined argument so this annotation cannot be stored anywhere.»

Пример кода на чистом Kotlin, который вызовет эту ошибку:

```
1 @Target(AnnotationTarget.FUNCTION)
2 annotation class FunAnn
3
4 inline fun myFun(a: Int, f: (Int) -&gt; String): String = f(a)
5
6 fun main() {
7   myFun(1) @FunAnn { it.toString() } // Call site annotation
8 }
```

AnnotatedInlineLambda.kt

Компилятор Compose подавляет эту проверку только для аннотации `@Composable`, так что допустим такой код:

```
@Composable
inline fun MyComposable(@StringRes nameResId: Int, resolver:
  ↪ (Int) -&gt; String) {
  val name = resolver(nameResId)
  Text(name)
}

@Composable
fun Screen() {
  MyComposable(nameResId = R.string.app_name) @Composable {
    LocalContext.current.resources.getString(it)
  }
}
```

AnnotatedComposableInlineLambda.kt

Это позволяет аннотировать параметры-лямбды как `@Composable` в месте вызова, а не только в объявлении функции, делая контракт функции гибче.

Ещё одно обходимое ограничение — именованные аргументы в местах, где компилятор Kotlin их не допускает, но только если функция помечена как `@Composable`.

Например, типы функций: в Kotlin у них нельзя использовать именованные аргументы, но Compose разрешает это для Composable-функций:

```
interface FileReaderScope {
  fun onFileOpen(): Unit
  fun onFileClosed(): Unit
  fun onLineRead(line: String): Unit
}

object Scope : FileReaderScope {
  override fun onFileOpen() = TODO()
  override fun onFileClosed() = TODO()
  override fun onLineRead(line: String) = TODO()
}

@Composable
fun FileReader(path: String, content: @Composable
  ↪ FileReaderScope.(path: String) -&gt; \
Unit) {
```

```
16   Column {
17     //...
18     Scope.content(path = path)
19   }
20 }
```

NamedParamsOnFunctionTypes.kt

Если убрать аннотацию `@Composable`, получим ошибку вида:

«Named arguments are not allowed for function types.»

Такое же требование подавляется для членов expect-классов. Jetpack Compose мультиплатформенный, поэтому рантайм должен принимать `expect`-функции и свойства с пометкой Composable.

Проверка версии рантайма

Статические проверяющие и подавитель диагностики установлены. Дальше — проверка версии Compose runtime. Компилятор Compose требует минимальную версию рантайма и проверяет, что она не устарела. Обнаруживаются и отсутствие рантайма, и устаревшая версия.

Одна версия компилятора Compose может поддерживать несколько версий рантайма, если они не ниже минимальной.

Это вторая проверка версий: сначала компилятор Kotlin, затем Jetpack Compose runtime.

Генерация кода

Наконец, компилятор переходит к фазе генерации кода. Это ещё одно общее с процессорами аннотаций — и те и другие часто синтезируют код для потребления рантайм-библиотеками.

IR в Kotlin

Как сказано ранее, плагины компилятора могут не только генерировать код, но и изменять исходники — у них есть доступ к **промежуточному представлению** (IR) до получения итогового кода для целевых платформ. Плагин может подставлять и заменять параметры, добавлять новые, менять структуру кода до «фиксации». Это происходит на бэкенд-фазах

компилятора Kotlin. Как можно догадаться, так Compose «внедряет» неявный дополнительный параметр `Composer` в каждый вызов Composable.

Плагины компилятора могут генерировать код в разных форматах. Для одной JVM можно было бы генерировать байткод, совместимый с Java, но с учётом планов команды Kotlin по стабилизации IR-бэкендов и сведению их к одному для всех платформ разумнее генерировать IR. IR — представление элементов языка **независимое от целевой платформы**. Генерация IR позволит сгенерированному коду Jetpack Compose быть мультиплатформенным.

Плагин компилятора Compose генерирует IR, регистрируя реализацию `IrGenerationExtension` — расширения общего IR-бэкенда компилятора Kotlin.

Для глубокого изучения Kotlin IR рекомендую серию постов Брайана Норманна про Kotlin IR и создание плагинов компилятора. Подробное изучение IR выходит за рамки этой книги.

Lowering (понижение)

Термин «lowering» — перевод компилятором более высокоуровневых или продвинутых концепций в комбинацию более низкоуровневых атомарных. В Kotlin это обычная практика: есть промежуточное представление (IR), выражающее продвинутые концепции, которые затем переводятся в низкоуровневые атомы перед преобразованием в байткод JVM, JavaScript, IR LLVM и т.д. У компилятора Kotlin есть фаза lowering; её можно понимать как нормализацию.

Компилятор Compose должен понижать некоторые поддерживаемые библиотекой концепции до представления, понятного рантайму. Процесс lowering — фаза генерации кода для плагина компилятора Compose: обход элементов дерева IR и изменение IR в соответствии с потребностями рантайма.

Кратко о важных вещах в фазе lowering в этом разделе:

- Вывод стабильности классов и добавление метаданных для понимания в рантайме.
- Трансформация выражений live literals для доступа к экземпляру MutableState, чтобы рантайм мог отражать изменения в исходниках без перекомпиляции (live literals).
- Внедрение неявного параметра `Composer` во все Composable-функции и передача его во все вызовы Composable.

- Оборачивание тел Composable-функций для: генерации разных типов групп по управлению потоком (replaceable, movable и т.д.); поддержки аргументов по умолчанию в scope сгенерированной группы функции; обучения функции пропуску recomposition; распространения информации об изменении состояния вниз по дереву для автоматической recomposition при изменении.

Разберём виды lowering в плагине компилятора Jetpack Compose.

Вывод стабильности класса

Умная recomposition означает пропуск recomposition Composable, когда их входы не изменились **и эти входы считаются стабильными**. Стабильность — важное понятие: рантайм Compose может безопасно читать и сравнивать эти входы для пропуска recomposition. Цель стабильности — помочь рантайму.

Напомним свойства стабильного типа:

- Вызовы `equals` для двух экземпляров всегда дают один и тот же результат для одних и тех же двух экземпляров — сравнение согласованно, рантайм может на него опереться.
- Composition всегда уведомляется при изменении публичного свойства типа. Иначе возможна рассинхронизация между входом Composable и актуальным состоянием при материализации. Чтобы этого не было, recomposition всегда запускается в таких случаях. **Умная recomposition не может опираться на такой вход.**
- Все публичные свойства — примитивы или типы, тоже считающиеся стабильными.

Все примитивные типы по умолчанию стабильны, а также `String` и все функциональные типы — они по определению неизменяемы. Неизменяемым типам не нужно уведомлять composition.

Мы также видели типы, которые не неизменяемы, но могут считаться стабильными — их можно пометить `@Stable`. Пример — `MutableState`: Compose уведомляется при каждом изменении, поэтому на него можно опираться для умной recomposition.

Для своих типов мы можем проверить выполнение перечисленных свойств и вручную пометить их как стабильные с помощью `@Immutable` или `@Stable`. Но полагаться на то, что разработчики будут соблюдать контракт, рискованно и сложно поддерживать. Желательно выводить стабильность класса автоматически.

Compose так и делает. Алгоритм вывода стабильности постоянно развивается; в общих чертах он обходит каждый класс и синтезирует для него аннотацию `@StabilityInferred`, а также синтетическое значение `static final int $stable`, кодирующее информацию о стабильности. Это значение поможет компилятору на следующих шагах генерировать код для определения стабильности класса в рантайме, чтобы Compose решал, нужно ли перезапускать зависящие от этого класса Composable-функции.

Не буквально каждый класс — только подходящие: публичные классы, не enum, не элемент enum, не интерфейс, не аннотация, не анонимный объект, не expect-элемент, не внутренний класс, не companion и не `inline`. И не те, что уже помечены как стабильные упомянутыми аннотациями. В итоге — обычные классы, data-классы и подобное, ещё не помеченные как стабильные. Это то, чем мы моделируем входы Composable-функций.

При выводе стабильности Compose учитывает разное. Тип считается стабильным, когда все поля класса только для чтения и стабильны (под «полем» — в смысле результирующего JVM-байткода). Классы вроде `class Foo` или `class Foo(val value: Int)` выводятся как стабильные (нет полей или только стабильные). Класс `class Foo(var value: Int)` сразу выводится как нестабильный.

На стабильность класса могут влиять параметры типа, например:

```
1 class Foo&lt;T&gt;(val value: T)
```

FooWithTypeParams.kt

В этом случае `T` используется в параметре класса, и стабильность `Foo` зависит от стабильности типа, подставленного вместо `T`. Так как `T` не reified, до рантайма он неизвестен. Нужен механизм определения стабильности класса в рантайме, когда тип для `T` станет известен. Компилятор Compose вычисляет и помещает в аннотацию `StabilityInferred` битовую маску, указывающую, что стабильность этого класса в рантайме должна зависеть от стабильности соответствующего параметра типа.

Наличие generic-типов не обязательно означает нестабильность. Компилятор знает, что код вроде `class Foo<T>(val a: Int, b: T) { val c: Int = b.hashCode() }` стабилен, так как hashCode по контракту всегда возвращает один и тот же результат для одного экземпляра.

Для классов, составленных из других классов, например `class Foo(val bar: Bar, val bazz: Bazz)`, стабильность выводится как комбинация стабильности

всех аргументов, рекурсивно.

Внутреннее мутабельное состояние тоже делает класс нестабильным. Пример:

```
class Counter {
 private var count: Int = 0
 fun getCount(): Int = count
 fun increment() { count++ }
}
```

Counter.kt

Состояние меняется со временем, даже если мутируется внутри самого класса. Рантайм не может полагаться на его согласованность.

В общем компилятор Compose считает тип стабильным только когда может это доказать. Например, интерфейс считается нестабильным — Compose не знает, как он будет реализован.

```
@Composable
fun &lt;T&gt; MyListOfItems(items: List&lt;T&gt;) {
  // ...
}
```

MyListOfItems.kt

Здесь аргумент — `List`, который может быть реализован мутабельно (например, `ArrayList`). Компилятору небезопасно предполагать только неизменяемые реализации, и вывести это непросто, поэтому тип считается нестабильным.

Другой пример — типы с мутабельными публичными свойствами, чья реализация может быть неизменяемой. Они тоже по умолчанию считаются нестабильными — компилятор не может вывести такую тонкость.

Это минус: часто такие вещи могут быть реализованы неизменяемо, и для Compose runtime этого было бы достаточно. Поэтому если модель, используемая как вход Composable, считается Compose нестабильной, мы можем явно пометить её `@Stable`, если у нас есть информация и мы контролируем тип. В официальной документации приведён такой пример:

```
// Marking the type as stable to favor skipping and smart
↪ recompositions.
@Stable
```

```
3 interface UiState&lt;T : Result&lt;T&gt;&gt; {
4     val value: T?
5     val exception: Throwable?
6
7     val hasError: Boolean
8         get() = exception != null
9 }
```

UiState.kt

Алгоритм вывода стабильности покрывает ещё случаи. По всем деталям рекомендую тесты библиотеки для `ClassStabilityTransform`.

Внутренние детали вывода стабильности могут со временем меняться и улучшаться. Для пользователей библиотеки это остаётся прозрачным.

Включение live literals

Оговорка: этот раздел близок к деталям реализации и постоянно развивается; в будущем может не раз измениться.

Один из флагов компилятора — live literals. Было две реализации этой возможности, включаемые флагами `liveLiterals` (v1) или `liveLiteralsEnabled` (v2).

Live literals — возможность отражать изменения в превью Compose в реальном времени без перекомпиляции. Компилятор Compose заменяет такие выражения версиями, читающими значения из `MutableState`. Рантайм получает уведомления об изменениях сразу, без пересборки проекта. Как сказано в kdoc библиотеки:

«Эта трансформация предназначена для улучшения опыта разработчика и не должна включаться в release-сборках — она заметно замедлит чувствительный к производительности код.»

Компилятор Compose генерирует уникальные id для каждого константного выражения в коде, затем превращает эти константы в геттеры свойств, читающие из некоего `MutableState`, хранящегося в сгенерированном singleton-классе на файл. В рантайме есть API для получения значения этих констант по сгенерированному ключу.

Пример из kdoc библиотеки. Исходный Composable:

```
1 @Composable
2 fun Foo() {
3   print(&amp;quot;Hello World&amp;quot;)
4 }
```

LiveLiterals1.kt

преобразуется в:

```
 1 // file: Foo.kt
 2 @Composable
 3 fun Foo() {
 4     print(LiveLiterals$FooKt.`getString$arg-0$call-print$fun-
   ↪ Foo`())
 5 }
 6
 7 object LiveLiterals$FooKt {
 8     var `String$arg-0$call-print$fun-Foo`: String = &amp;quot;Hello
   ↪ World&amp;quot;
 9     var `State$String$arg-0$call-print$fun-Foo`:
   ↪ MutableState&amp;lt;String&amp;gt;? = null
10     fun `getString$arg-0$call-print$fun-Foo`(): String {
11
12         val field = this.`String$arg-0$call-print$fun-Foo`
13
14         val state = if (field == null) {
15             val tmp = liveLiteral(
16                 &amp;quot;String$arg-0$call-print$fun-Foo&amp;quot;,
17                 this.`String$arg-0$call-print$fun-Foo`
18             )
19             this.`String$arg-0$call-print$fun-Foo` = tmp
20             tmp
21         } else field
22
23         return field.value
24     }
25 }
```

LiveLiterals2.kt

Константа заменена геттером, читающим из `MutableState` в сгенерированном

singleton для файла.

Мемоизация лямбд Compose

На этом шаге генерируется IR, чтобы научить рантайм оптимизировать выполнение лямбд, передаваемых в Composable-функции. Это делается для двух видов лямбд:

- **Не-Composable лямбды:** компилятор генерирует IR для их мемоизации, оборачивая каждую лямбду в вызов `remember`. Например, колбэк, передаваемый в Composable. Здесь `remember` позволяет использовать slot table для хранения и последующего чтения этих лямбд.
- **Composable лямбды:** компилятор генерирует IR для оборачивания и добавления информации, чтобы научить рантайм хранить и читать выражение в/из Composition. Цель та же, что у `remember`, но без его использования. Пример — content Composable лямбды, передаваемые в узлы Compose UI при их вызове.

Не-Composable лямбды

Этот шаг оптимизирует передаваемые в Composable лямбды для переиспользовани
Kotlin уже оптимизирует лямбды без захвата значений, представляя их синглтонами — один экземпляр на всю программу. Но при захвате значений оптимизация невозможна: значения могут различаться при каждом вызове, нужен отдельный экземпляр. Compose умнее в этом случае. Пример:

```
1 @Composable
2 fun NamePlate(name: String, onClick: () -&gt; Unit) {
3   // ...
4   // onClick()
5   // ...
6 }
```

NamePlateClickLambda.kt

Здесь `onClick` — обычная Kotlin-лямбда, передаваемая в Composable. Если лямбда с места вызова захватывает значения, Compose может научить рантайм мемоизировать её — по сути обернув в вызов `remember`. Это делается через сгенерированный IR. Вызов будет запоминать лямбду по захваченным значениям **при условии, что эти значения стабильны**. Рантайм сможет переиспользовать существующие лямбды вместо создания новых, если захваченные значения совпадают (включая параметры).

Требование стабильности захваченных значений нужно потому, что они используются как аргументы-условия для `remember` и должны быть надёжны для сравнения.

Мемоизированные лямбды **не могут быть inline** — после подстановки в вызывающий код при компиляции запоминать уже нечего.

Оптимизация имеет смысл только для лямбд с захватом; без захвата достаточно оптимизации Kotlin (синглтоны).

Мемоизация делается по захваченным значениям лямбды. При генерации IR компилятор добавляет перед выражением вызов `remember` с типом возврата, совпадающим с типом мемоизированного выражения, затем generic-аргумент типа вызова (`remember<T>...`), затем все захваченные значения как аргументы-условия (`remember<T>(arg1, arg2...)`) для сравнения и наконец саму лямбду (`remember<T>(arg1, arg2..., expression)` как trailing lambda).

Использование захваченных значений как аргументов-условий гарантирует, что они служат ключами для запоминания результата выражения — он будет инвалидироваться при их изменении.

Автоматическое запоминание лямбд, передаваемых в Composable, даёт переиспользование при recomposition.

Composable-лямбды

Компилятор Compose также умеет мемоизировать Composable-лямбды. Детали реализации для этого случая другие из-за «особой» реализации Composable-лямбд, но цель та же: хранить и читать эти лямбды в/из slot table.

Пример Composable-лямбды, которая будет мемоизирована:

```
1 @Composable
2 fun Container(content: @Composable () -&gt; Unit) {
3   // ...
4   // content()
5   // ...
6 }
```

Container.kt

Для этого IR лямбды изменяется так, чтобы сначала вызывать **composable factory function** с определёнными параметрами: `composableLambda(...)`.

Первым добавляется текущий `$composer` и передаётся: `composableLambda($comp ...)`.

Затем параметр `key` — комбинация `hashcode` полностью квалифицированного имени Composable-лямбды и **смещения начала выражения** в файле для уникальности ключа (позиционная мемоизация): `composableLambda($composer, $key, ...)`.

После ключа добавляется булев параметр `shouldBeTracked` — нужно ли отслеживать этот вызов Composable-лямбды. Когда у лямбд нет захватов, Kotlin превращает их в синглтоны, они не меняются и не требуют отслеживания Compose: `composableLambda($composer, $key, $shouldBeTracked, ...)`.

Опционально может добавляться параметр арности выражения — только при более чем 22 параметрах: `composableLambda($composer, $key, $shouldBeTracked, $arity, ...)`.

В конце в обёртку передаётся сама лямбда (trailing lambda): `composableLambda($c $key, $shouldBeTracked, $arity, expression)`.

Назначение Composable factory function — **добавить replaceable group в composition для хранения выражения лямбды по сгенерированному ключу**. Так Compose сообщает рантайму, как хранить и извлекать Composable-выражение.

Помимо оборачивания Compose может оптимизировать Composable-лямбды без захвата значений так же, как Kotlin — представляя их синглтонами. Для этого генерируется синтетический `internal object` «ComposableSingletons» на каждый файл с Composable-лямбдами. Объект хранит (мемоизирует) статические ссылки на эти лямбды и геттеры для их получения.

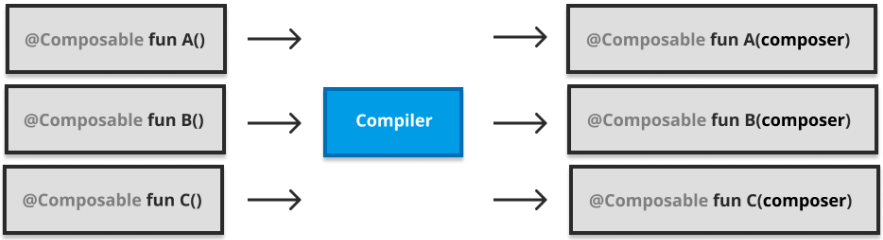
У Composable-лямбд есть ещё одна оптимизация за счёт реализации: аналогично `MutableState`, лямбду вида `@Composable (A, B) -> C` можно считать реализованной как `State<@Composable (A, B) -> C`. Места вызова лямбды (`lambda(a, b)`) можно заменить на `lambda.value.invoke(a, b)`.

Это оптимизация: создаётся snapshotted state для всех Composable-лямбд, что позволяет Compose умнее перезапускать поддеревья при изменении лямбд. Изначально это называли «donut-hole skipping»: лямбда может обновляться «высоко» в дереве, а Compose перезапускает только «низкую» часть, где значение реально читается. Для лямбд это выгодный компромисс: по природе экземпляр передаётся вниз по иерархии и часто «читается» (вызывается) только в нижней части.

Внедрение Composer

На этом шаге компилятор Compose заменяет все Composable-функции новыми версиями с добавленным синтетическим параметром `Composer`. Этот параметр также передаётся в каждый вызов Composable в коде, чтобы он был доступен в любой точке дерева, включая вызовы Composable-лямбд.

Требуется и переназначение типов: тип функции меняется при добавлении плагином дополнительных параметров.



Внедрение Composer

Так `Composer` становится доступен для любого поддерева и несёт всю информацию для материализации composable-дерева и его обновления.

Пример:

```
fun NamePlate(name: String, lastname: String, $composer:
  ↪ Composer) {
  $composer.start(123)
  Column(modifier = Modifier.padding(16.dp), $composer) {
    Text(
      text = name,
      $composer
    )
    Text(
      text = lastname,
      style = MaterialTheme.typography.subtitle1,
      $composer
    )
  }
  $composer.end()
```

```
15 }
```

NamePlate.kt

Inline-лямбды без Composable намеренно не трансформируются — при компиляции они подставляются в вызывающий код и исчезают. Функции `expect` тоже не трансформируются — при разрешении типов они сводятся к `actual`, и трансформируются уже они.

Распространение сравнения (comparison propagation)

Мы видели, как компилятор внедряет дополнительный параметр `$composer` и передаёт его во все вызовы Composable. Есть и другие метаданные, генерируемые и добавляемые к каждому Composable. Один из них — параметр `$changed`. Он передаёт информацию о том, могли ли входные параметры текущего Composable измениться с прошлой composition, и позволяет пропускать recomposition.

```
1 @Composable
2 fun Header(text: String, $composer: Composer&lt;*&gt;, $changed:
 ↪  Int)
```

SyntheticChangedParam1.kt

Параметр синтезируется как комбинация битов для **каждого** входного параметра функции (один параметр `$changed` кодирует условие для всех n входных параметров, порядка 10, в пределах числа бит; при большем числе параметров добавляются 2 или более флагов). Биты используются потому, что процессоры хорошо обрабатывают их.

Эта информация позволяет рантайму:

- Пропускать сравнения через `equals` для проверки изменения входного параметра относительно последнего сохранённого (из прошлых composition) в случаях, когда параметр известен как статический. Битовая маска `$changed` даёт эту информацию. Например, параметр — строковый литерал, константа и т.п. Биты флага сообщают рантайму, что значение **известно на этапе компиляции**, не изменится в рантайме, и рантайм может **никогда его не сравнивать**.
- В других случаях параметр **всегда** либо не менялся с прошлой composition, либо при изменении сравнение уже гарантированно

выполнено родительским Composable в дереве — пересчитывать не нужно. Состояние параметра считается «определённым».

- В остальных случаях состояние «неопределённое» — рантайм сравнивает через `equals` и сохраняет в slot table для последующего использования. Значение бита для этого случая — `0`, случай по умолчанию. При `$changed == 0` мы говорим рантайму делать всю работу (без сокращений).

Пример тела Composable после внедрения параметра `$changed` и логики для него:

```
1 @Composable
2 fun Header(text: String, $composer: Composer&lt;*&gt;,
  ↪ $changed: Int) {
3   var $dirty = $changed
4   if ($changed and 0b0110 === 0) {
5     $dirty = $dirty or if ($composer.changed(text)) 0b0010 else
  ↪ 0b0100
6   }
7   if ($dirty and 0b1011 xor 0b1010 !== 0 || !$composer.skipping)
  ↪ {
8     f(text) // executes body
9   } else {
10    $composer.skipToGroupEnd()
11  }
12 }
```

SyntheticChangedParam2.kt

Здесь манипуляции с битами; не вдаваясь в детали, видно использование локальной переменной `$dirty` — хранит, изменился ли параметр, по битовой маске `$changed` и при необходимости по значению из slot table. Если значение «грязное» (изменилось), выполняется тело функции (recomposition). Иначе Composable пропускает recomposition.

Так как recomposition может происходить часто, передача информации об эволюции входного состояния может сэкономить время и место. Параметры нередко проходят через много Composable, и Compose не хочет каждый раз хранить и сравнивать их — это занимает место в slot table.

Как наш Composable получает параметр `$changed` от вызывающего, так же он должен передавать вниз по дереву информацию о параметрах, которую знает.

Это называется «распространением сравнения» (comparison propagation). Информация доступна в теле во время composition: если мы уже знаем, что вход изменился, статичен и т.д., мы можем передать это в параметр `$changed` дочернего Composable, который использует тот же параметр.

Параметр `changed` также кодирует, стабилен ли переданный аргумент или нет. Это позволяет функции с более широким типом (например, `List<T>`) пропускать recomposition, если параметр стабилен на основе переданного аргумента (например, выражение `listOf(1, 2)`).

Подробнее об этом — в видео Лиланда Ричардсона про основы этого механизма и вывод стабильности.

Параметры по умолчанию

Ещё один вид метаданных, добавляемых к каждой Composable на этапе компиляции — параметр/параметры `$default`.

Поддержка аргументов по умолчанию в Kotlin не подходит для аргументов Composable-функций: выражения по умолчанию нужно вычислять внутри scope (сгенерированной группы) функции. Compose даёт альтернативную реализацию разрешения аргументов по умолчанию.

Compose представляет аргументы по умолчанию параметром-битовой маской `$default`, сопоставляющим каждый индекс параметра с битом. Аналогично параметру/параметрам `$changed`. Один параметр `$default` на каждые n входных параметров со значениями по умолчанию. Маска указывает, передан ли параметр в месте вызова или нужно использовать выражение по умолчанию.

Пример из документации библиотеки — Composable до и после внедрения маски `$default` и кода для её чтения и подстановки значения по умолчанию:

```
// Before compiler (sources)
@Composable fun A(x: Int = 0) {
  f(x)
}

// After compiler
@Composable fun A(x: Int, $changed: Int, $default: Int) {
  // ...
  val x = if ($default and 0b1 != 0) 0 else x
  f(x)
```

```
11    // ...
12 }
```

DefaultParam.kt

Снова битовые операции: проверка маски `$default` для подстановки `0` или переданного значения `x`.

Генерация групп управления потоком

Компилятор Compose также вставляет **группу** в тело каждой Composable-функции. Типы групп зависят от структур управления потоком в теле:

- Replaceable groups.
- Movable groups.
- Restartable groups.

Composable-функции в рантайме эмитируют группы; группы оборачивают информацию о текущем состоянии вызова Composable. Так Composition знает, как очищать записанные данные при замене группы (replaceable), перемещать данные с сохранением идентичности Composable (movable) или перезапускать функцию при recomposition (restartable).

В итоге рантайму нужно уметь обрабатывать управление потоком по информации, хранящейся в Composition в памяти.

Группы несут информацию о позиции вызова в исходниках. Они оборачивают span текста и имеют ключ, в котором одним из факторов служит позиция вызова. Это позволяет хранить группу и обеспечивает позиционную мемоизацию.

Replaceable groups

Ранее мы описали, что тело Composable-лямбды автоматически оборачивается вызовом Composable factory function с параметрами `$composer`, сгенерированным `$key` и самой лямбдой. В коде эта factory выглядит так:

```
1 fun composableLambda(
2     composer: Composer,
3     key: Int,
4     tracked: Boolean,
5     block: Any
6 ): ComposableLambda {
```

```
    composer.startReplaceableGroup(key)
    val slot = composer.rememberedValue()
    val result = if (slot === Composer.Empty) {
        val value = ComposableLambdaImpl(key, tracked)
        composer.updateRememberedValue(value)
        value
    } else {
        slot as ComposableLambdaImpl
    }
    result.update(block)
    composer.endReplaceableGroup()
    return result
}
```

ReplaceableGroup.kt

Эта factory вызывается для Composable-лямбд, например для content наших Composable-функций. Она сначала начинает replaceable group с ключом и в конце закрывает группу, оборачивая span в середине. Между start и end обновляет composition нужной информацией — в данном случае оборачиваемой лямбдой.

То же происходит и для других вызовов Composable. Пример трансформации кода обычной Composable-функции с пометкой non-restartable:

```
// Before compiler (sources)
@NonRestartableComposable
@Composable
fun Foo(x: Int) {
  Wat()

// After compiler
@NonRestartableComposable
@Composable
fun Foo(x: Int, %composer: Composer?, %changed: Int) {
  %composer.startReplaceableGroup(&lt;&gt;)
  Wat(%composer, 0)
  %composer.endReplaceableGroup()
}
```

ReplaceableGroup2.kt

Вызов Composable также эмитирует replaceable group для хранения в Composition. Группы образуют дерево; каждая группа может содержать любое число дочерних. Если вызов `Wat` тоже Composable, компилятор вставит для него группу.

Ранее мы использовали пример с двумя вызовами `Text` в ветках `if`/`else` — идентичность сохраняется и по позиции вызова, так что рантайм различает эти вызовы:

```
1 if (condition) {
2   Text(&amp;quot;Hello&amp;quot;)
3 } else {
4   Text(&amp;quot;World&amp;quot;)
5 }
```

ConditionalTexts.kt

Composable с такой условной логикой тоже эмитирует replaceable group, чтобы при переключении условия группу можно было заменить.

Movable groups

Группы, которые можно переупорядочивать без потери идентичности. Сейчас они нужны только для тела вызовов `key`. Напомним пример из предыдущей главы:

```
 1 @Composable
 2 fun TalksScreen(talks: List&amp;lt;Talk&amp;gt;) {
 3   Column {
 4     for (talk in talks) {
 5       key(talk.id) { // Unique key
 6         Talk(talk)
 7       }
 8     }
 9   }
10 }
```

MovableGroup.kt

Оборачивание `Talk` в `key` даёт уникальную идентичность. При оборачивании Composable в `key` генерируется movable group для переупорядочивания вызовов без потери идентичности элементов.

Пример трансформации Composable с key:

```
// Before compiler (sources)
@Composable
fun Test(value: Int) {
  key(value) {
    Wrapper {
      Leaf(&amp;quot;Value ${&amp;#39;$&amp;#39;}value&amp;quot;)
    }
  }
}

// After
@Composable
fun Test(value: Int, %composer: Composer?, %changed: Int) {
  // ...
  %composer.startMovableGroup(&amp;lt;&amp;gt;, value)
  Wrapper(composableLambda(%composer, &amp;lt;&amp;gt;, true) {
 %composer: Composer?, %changed: In\
t -&amp;gt;
    Leaf(&amp;quot;Value %value&amp;quot;, %composer, 0)
  }, %composer, 0b0110)
  %composer.endMovableGroup()
  // ...
}
```

MovableGroup2.kt

Restartable groups

Restartable groups, вероятно, самые интересные. Они вставляются только для restartable Composable-функций. Они тоже оборачивают вызов Composable, но кроме того расширяют вызов end, чтобы он возвращал nullable-значение. Оно будет null только когда тело Composable не читает никакого изменяемого состояния — **recomposition не потребуется**. В таком случае не нужно сообщать рантайму, как перезапускать этот Composable. Иначе возвращается ненулевое значение, и компилятор генерирует лямбду, сообщающую рантайму, как «перезапустить» (выполнить заново) Composable и обновить Composition.

В коде это выглядит так:

```
// Before compiler (sources)
@Composable fun A(x: Int) {
  f(x)
}

// After compiler
@Composable
fun A(x: Int, $composer: Composer&lt;*&gt;, $changed: Int) {
  $composer.startRestartGroup()
  // ...
  f(x)
  $composer.endRestartGroup()?.updateScope { next -&gt;
    A(x, next, $changed or 0b1)
  }
}
```

RestartableGroup.kt

Scope обновления для recomposition содержит новый вызов того же Composable.

Такой тип группы генерируется для всех Composable-функций, читающих состояние.

Из официальной документации — дополнительная логика компилятора для исполняемых блоков при генерации типов групп:

- Если блок выполняется ровно 1 раз всегда, группы не нужны.
- Если из набора блоков каждый раз выполняется ровно один (например, ветки if или when), вокруг каждого блока вставляется replaceable group. Это условная логика.
- Movable group нужна только для content-лямбды вызовов `key`.

Klib и генерация decoy

В компилятор Compose добавлена поддержка `.klib` (мультиплатформа) и Kotlin/JS. Она нужна из-за десериализации IR зависимостей в JS — после трансформации IR сигнатуры типов перестают совпадать (Compose добавляет синтетические параметры к объявлениям и вызовам Composable-функций).

Поэтому в Kotlin/JS Compose не заменяет IR функций (как в JVM), а создаёт копии. Оригинальное объявление функции сохраняется для связи

между функцией в метаданных Kotlin и её IR, ссылки в коде по-прежнему разрешаются, а IR копии изменяется Compose по необходимости. Чтобы различать их в рантайме, к имени добавляется суффикс `$composable`.

```
// Before compiler (sources)
@Composable
fun Counter() {
  ...
}

// Transformed
@Decoy(...)
fun Counter() { // signature is kept the same
  illegalDecoyCallException(&amp;quot;Counter&amp;quot;)
}

@DecoyImplementation(...)
fun Counter$composable( // signature is changed
    $composer: Composer,
    $changed: Int
) {
  ...transformed code...
}
```

KlibAndDecoys.kt

Подробнее о поддержке `.klib` и Kotlin/JS — в посте Андрея Шикова.

3. Compose runtime

Недавно я опубликовал в Twitter краткое описание внутренней работы архитектуры Compose — взаимодействие UI, компилятора и рантайма.

Твит об архитектуре Compose

Эта ветка может послужить введением в главу: она даёт общий обзор важных моментов. Глава посвящена Jetpack Compose runtime и закрепляет понимание того, как части Compose взаимодействуют. При желании можно сначала прочитать тред в Twitter.

В нём объясняется, как Composable-функции **эмитируют** изменения в Composition, чтобы обновить его актуальной информацией, и как это происходит через внедрённый экземпляр `$composer` благодаря компилятору (глава 2). Вызов получения текущего `Composer` и сама Composition — часть Jetpack Compose runtime.

Тред намеренно остаётся на поверхности — Twitter не лучшее место для глубокого разбора. В книге мы как раз углубимся.

До сих пор мы называли состояние в памяти рантайма «Composition» — намеренно упрощённо. Начнём с структур данных для хранения и обновления состояния Composition.

Slot table и список изменений

Различие между этими двумя структурами часто путают, во многом из-за нехватки литературы о внутреннем устройстве Compose. Сейчас важно прояснить это в первую очередь.

Slot table — оптимизированная in-memory структура, в которой рантайм хранит **текущее состояние Composition**. Она заполняется при первичной composition и обновляется при каждой recomposition. Можно представить её как след всех вызовов Composable-функций: расположение в исходниках, параметры, запомненные значения, `CompositionLocal` и т.д. Всё, что произошло во время composition, хранится там. Эта информация потом

используется `Composer` для построения следующего списка изменений — любые изменения дерева зависят от текущего состояния Composition.

Slot table фиксирует состояние; список изменений (change list) — то, что реально меняет дерево узлов. Его можно понимать как патч: после применения дерево обновляется. Все нужные изменения записываются, затем применяются. Применение списка изменений — задача `Applier`, абстракции, через которую рантайм в итоге материализует дерево. Подробнее — ниже.

Наконец, `Recomposer` координирует процесс: когда и в каком потоке выполнять recomposition и когда применять изменения. Об этом тоже ниже.

Slot table подробнее

Разберём, как хранится состояние Composition. Slot table оптимизирована для быстрого линейного доступа и основана на идее «gap buffer», распространённой в текстовых редакторах. Данные хранятся в двух линейных массивах: в одном — информация о **группах** в Composition, в другом — **слоты** каждой группы.

```
1 var groups = IntArray(0)
2   private set
3
4 var slots = Array&lt;Any?&gt;(0) { null }
5   private set
```

LinearStructures.kt

В главе 2 мы видели, как компилятор оборачивает тела Composable-функций, чтобы они эмитировали группы. Группы задают идентичность Composable в памяти (уникальный ключ) для последующей идентификации. Группы оборачивают информацию о вызове Composable и его потомках и указывают, как обрабатывать Composable (как группу). Тип группы зависит от паттернов управления потоком в теле: restartable, movable, replaceable, reusable…

Массив `groups` хранит значения `Int` — только «поля групп» (метаданные). Родительские и дочерние группы хранятся в виде полей групп. Так как структура линейная, поля родителя идут первыми, затем поля всех детей. Так моделируется дерево групп с упором на линейный обход. Случайный доступ дорог, кроме как через anchor группы. `Anchor` — по сути указатели для быстрого доступа.

Массив `slots` хранит данные для каждой группы. Тип значений — `Any?`, так как нужно хранить любую информацию. Здесь лежат реальные

данные Composition. Каждая группа в `groups` описывает, как найти и интерпретировать её слоты в `slots` — группа связана с диапазоном слотов.

Slot table использует «gap» для чтения и записи — диапазон позиций в таблице. Gap перемещается и определяет, откуда читать и куда писать в массивах. У gap есть указатель начала записи; начало и конец могут сдвигаться, так что данные в таблице можно перезаписывать.

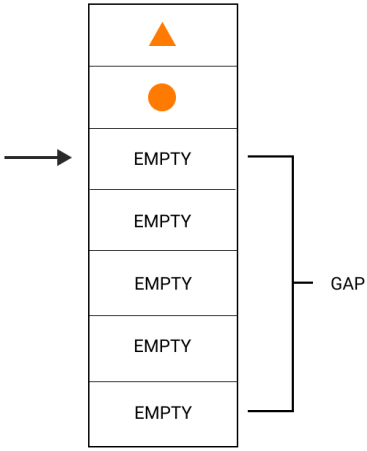


Схема slot table

Пример условной логики:

```
1 @Composable
2 @NonRestartableComposable
3 fun ConditionalText() {
4   if (a) {
5     Text(a)
6   } else {
7     Text(b)
8   }
9 }
```

ConditionalNonRestartable.kt

Так как этот Composable помечен как **non-restartable**, вставляется replaceable group (вместо restartable). Группа хранит в таблице данные текущего «активного» потомка — при `a == true` это `Text(a)`. При смене условия gap

возвращается к началу группы и запись идёт оттуда, перезаписывая слоты данными для `Text(b)`.

Для чтения и записи используются `SlotReader` и `SlotWriter`. У slot table может быть несколько активных читателей, но **только один активный писатель**. После каждой операции чтения/записи соответствующий reader/writer закрывается. Одновременно может быть открыто любое число читателей, но читать можно **только когда не идёт запись**. `SlotTable` считается невалидной, пока активный writer не закрыт — он напрямую меняет группы и слоты, одновременное чтение могло бы привести к гонкам.

Reader работает как **посетитель**: отслеживает текущую группу в массиве groups, её начало и конец, родителя (хранится сразу перед ней), текущий слот группы, количество слотов и т.д. Умеет перепозиционироваться, пропускать группы, читать значение текущего слота и по индексам. Иными словами, используется для чтения информации о группах и слотах из массивов.

Writer записывает группы и слоты в массивы. Как сказано выше, в таблицу можно записывать данные любого типа (`Any?`). `SlotWriter` опирается на **gap** для групп и слотов, чтобы определять позиции записи в массивах.

Gap — **перемещаемый и изменяемый по размеру диапазон позиций** в линейном массиве. Writer хранит начало, конец и длину каждого gap и может перемещать gap, обновляя эти позиции.

Writer может добавлять, заменять, перемещать и удалять группы и слоты — например, добавлять новый узел Composable в дерево или заменять Composable при смене условия. Он может пропускать группы и слоты, продвигаться на заданное число позиций, переходить к позиции по `Anchor` и т.д.

Writer ведёт список `Anchor` для быстрого доступа по индексам. Позиция каждой группы (group index) в таблице тоже отслеживается через `Anchor`; anchor обновляется при перемещении, замене, вставке или удалении групп перед указываемой позицией.

Slot table также выступает итератором по группам composition для инструментов, инспектирующих Composition.

Пора перейти к списку изменений.

Подробнее о slot table — в посте Лиланда Ричардсона из команды Jetpack Compose.

Список изменений

Мы разобрали slot table и то, как рантайм отслеживает текущее состояние Composition. Какова тогда роль списка изменений? Когда он создаётся? Что моделирует? Когда изменения применяются и зачем? Остаётся много вопросов. Этот раздел добавляет ещё один кусок пазла.

При каждой composition (или recomposition) выполняются Composable-функции и **эмитируют**. «Эмитирование» мы уже много раз использовали. Эмитирование — создание **отложенных изменений** для обновления slot table и в итоге материализованного дерева. Эти изменения хранятся в списке. Новый список строится на основе того, что уже есть в slot table: любые изменения дерева зависят от текущего состояния Composition.

Пример — перемещение узла. При переупорядочивании Composable в списке нужно найти старую позицию узла в таблице, удалить его слоты и записать их заново с новой позиции.

То есть при каждом эмитировании Composable смотрит в slot table, создаёт отложенное изменение по потребности и текущим данным и добавляет его в список. Позже, по завершении composition, наступает материализация и **записанные** изменения выполняются — обновляют slot table актуальной информацией Composition. Поэтому эмитирование быстро: создаётся отложенное действие, которое будет выполнено позже.

Список изменений в итоге и вносит изменения в таблицу. Сразу после этого он уведомляет Applier об обновлении материализованного дерева узлов.

Как сказано выше, `Recomposer` оркестрирует процесс: на каком потоке выполнять composition/recomposition и на каком применять изменения. Последний контекст также используется по умолчанию для запуска эффектов в `LaunchedEffect`.

Теперь яснее, как изменения записываются, откладываются и выполняются и как состояние хранится в slot table. Пора разобрать `Composer`.

Composer

Внедрённый `$composer` связывает написанные нами Composable-функции с Compose runtime.

Подача данных в Composer

Разберём, как узлы добавляются в in-memory представление дерева. Возьмём Composable `Layout`. `Layout` — основа всех UI-компонентов Compose UI. В коде это выглядит так:

```
@Suppress(&quot;ComposableLambdaParameterPosition&quot;)
@Composable inline fun Layout(
  content: @Composable () -&gt; Unit,
  modifier: Modifier = Modifier,
  measurePolicy: MeasurePolicy
) {
  val density = LocalDensity.current
  val layoutDirection = LocalLayoutDirection.current
  ReusableComposeNode&lt;ComposeUiNode, Applier&lt;Any&gt;&gt;(
    factory = ComposeUiNode.Constructor,
      update = {
        set(measurePolicy, ComposeUiNode.SetMeasurePolicy)
        set(density, ComposeUiNode.SetDensity)
        set(layoutDirection, ComposeUiNode.SetLayoutDirection)
      },
      skippableUpdate = materializerOf(modifier),
      content = content
  )
}
```

Layout.kt

`Layout` использует `ReusableComposeNode`, чтобы эмитировать `LayoutNode` в composition. Звучит как «создать и сразу добавить узел», но на деле это **обучение рантайма** тому, как создать, инициализировать и вставить узел в текущую позицию Composition **когда придёт время**. Код:

```
@Composable
inline fun &lt;T, reified E : Applier&lt;*&gt;&gt;
  ReusableComposeNode(
  noinline factory: () -&gt; T,
  update: @DisallowComposableCalls Updater&lt;T&gt;.() -&gt;
Unit,
  noinline skippableUpdate: @Composable
SkippableUpdater&lt;T&gt;.() -&gt; Unit,
  content: @Composable () -&gt; Unit
```

```
) {
  // ...
  currentComposer.startReusableNode()
  // ...
  currentComposer.createNode(factory)
  // ...
  Updater&lt;T&gt;(currentComposer).update() // initialization
  // ...
  currentComposer.startReplaceableGroup(0x7ab4aae9)
  content()
  currentComposer.endReplaceableGroup()
  currentComposer.endNode()
}
```

ReusableComposeNode.kt

Часть несущественного опущена; всё делегируется экземпляру `currentComposer`. Видно также использование replaceable group для оборачивания content при хранении. Всё, что эмитируется внутри лямбды `content`, будет храниться как дети этой группы (и значит этого Composable) в Composition.

Та же операция эмитирования выполняется для любых других Composable. Например, `remember`:

```
@Composable
inline fun &lt;T&gt; remember(calculation:
  @DisallowComposableCalls () -&gt; T): T =
  currentComposer.cache(invalid = false, calculation)
```

Composables.kt

Composable `remember` использует `currentComposer`, чтобы закэшировать (запомнить) значение лямбды в composition. Параметр `invalid` принудительно обновляет значение даже если оно уже сохранено. Функция `cache` реализована так:

```
@ComposeCompilerApi
inline fun &lt;T&gt; Composer.cache(invalid: Boolean, block: ()
  -&gt; T): T {
  return rememberedValue().let {
    if (invalid || it === Composer.Empty) {
      val value = block()
```

```
      updateRememberedValue(value)
      value
    } else it
  } as T
}
```

Composer.kt

Сначала ищется значение в Composition (slot table). Если не найдено — эмитируются изменения для **запланированного обновления** значения (то есть запись). Иначе возвращается сохранённое значение.

Моделирование изменений

Как в предыдущем разделе, все операции эмитирования, делегированные `currentComposer`, внутри представлены как `Change`, добавляемые в список. `Change` — отложенная функция с доступом к текущему `Applier` и `SlotWriter` (напоминание: активный writer в каждый момент один). В коде:

```
internal typealias Change = (
  applier: Applier&lt;*&gt;,
  slots: SlotWriter,
  rememberManager: RememberManager
) -&gt; Unit
```

Composer.kt

Эти изменения добавляются в список (записываются). «Эмитирование» по сути — создание таких `Change`, отложенных лямбд для добавления, удаления, замены или перемещения узлов в slot table и уведомления `Applier` (чтобы изменения материализовались).

Поэтому «эмитирование изменений» можно называть «записью» или «планированием» изменений — речь об одном и том же.

После composition, когда все вызовы Composable завершены и изменения записаны, **все они применяются пакетно через Applier**.

Сама composition моделируется классом `Composition`. Пока отложим его — процесс composition разберём ниже. Сначала ещё несколько деталей о Composer.

Оптимизация момента записи

Как мы видели, вставка новых узлов делегируется Composer — он всегда знает, когда уже идёт процесс **вставки** узлов в composition. В этом случае Composer может не откладывать запись и сразу писать в slot table при эмитировании изменений вместо их записи в список. В остальных случаях изменения записываются и откладываются — время ещё не пришло.

Запись и чтение групп

По завершении composition вызывается `composition.applyChanges()` для материализации дерева, и изменения записываются в slot table. Composer может записывать разные типы информации: данные, узлы, группы. Всё в итоге хранится в виде групп с разными полями для различения.

Composer может «начать» и «закончить» любую группу. Смысл зависит от действия: при записи это «группа создана» / «группа удалена» в slot table; при чтении `SlotReader` перемещает указатели чтения внутрь и наружу группы.

Узлы в дереве Composable (в таблице — группы) не только вставляются, но могут удаляться или перемещаться. Удаление группы — удаление её и всех её слотов из таблицы. Composer перепозиционирует `SlotReader`, заставляет его пропустить группу (её уже нет) и записывает операции удаления узлов в Applier. Все модификации планируются (записываются) и применяются пакетно позже — чтобы они были согласованы. Composer также отменяет все отложенные инвалидации удалённой группы.

Не все группы restartable, replaceable, movable или reusable. Среди прочего группами хранятся, например, блоки-обёртки для значений по умолчанию — они окружают запомненные значения для вызовов Composable с параметрами по умолчанию, например `model: Model = remember { DefaultModel() }`.

Когда Composer хочет начать группу:

- Если Composer в режиме **вставки**, он сразу пишет в slot table.
- Иначе при наличии отложенных операций записывает изменения для применения. Composer попытается переиспользовать группу, если она уже есть в таблице.
- Если группа уже сохранена **в другой позиции** (перемещена), записывается операция перемещения всех слотов группы.
- Если группа новая (не найдена в таблице), включается режим `inserting`: группа и все её дети записываются в промежуточную `insertTable`

(другой SlotTable) до завершения группы, затем планируется вставка групп в итоговую таблицу.
- Если Composer не в режиме вставки и нет отложенных операций записи, он пытается начать чтение группы.

Переиспользование групп часто: иногда новый узел не нужен, можно использовать существующий (см. `ReusableComposeNode` выше). Эмитируется (записывается) операция перехода к узлу через Applier, но пропускаются операции создания и инициализации.

Когда нужно обновить свойство узла, это действие тоже записывается как `Change`.

Запоминание значений

Composer умеет **запоминать** значения в Composition (писать их в slot table) и позже обновлять. Сравнение с прошлой composition выполняется при вызове `remember`, но действие обновления записывается как `Change`, если Composer не в режиме вставки.

Когда запоминаемое значение — `RememberObserver`, Composer также записывает неявный `Change` для отслеживания запоминания в Composition — это понадобится при «забывании» запомненных значений.

Recompose scope

Через Composer создаются и recompose scope, обеспечивающие умную recomposition. Они связаны с restart-группами. При создании restart group Composer создаёт для неё `RecomposeScope` и устанавливает его как `currentRecomposeScope` для Composition.

`RecomposeScope` — область Composition, которую можно перезапускать независимо от остального. Её можно вручную инвалидировать для запуска recomposition Composable: `composer.currentRecomposeScope().invalidate()`. При recomposition Composer позиционирует slot table на начало этой группы и вызывает переданную лямбду recompose — по сути снова вызывается Composable, он эмитирует ещё раз, и Composer перезаписывает его данные в таблице.

Composer ведёт `Stack` всех инвалидированных recompose scope — то есть ожидающих recomposition. `currentRecomposeScope` получается через peek в этот стек.

`RecomposeScope` не всегда активны — только когда Compose находит чтение из snapshot `State` внутри Composable. Тогда Composer помечает `RecomposeScope` как `used`, и вставленный вызов «end» в конце Composable **перестаёт возвращать null** — активируется следующая за ним лямбда recomposition (после ?):

```
// After compiler inserts boilerplate
@Composable
fun A(x: Int, $composer: Composer&lt;*&gt;, $changed: Int) {
  $composer.startRestartGroup()
  // ...
  f(x)
  $composer.endRestartGroup()?.updateScope { next -&gt;
    A(x, next, $changed or 0b1)
  }
}
```

RecomposeScope.kt

Composer может перезапустить все инвалидированные дочерние группы текущего родителя при необходимости или просто заставить reader пропустить группу до конца (см. раздел о comparison propagation в главе 2).

SideEffect в Composer

Composer также записывает `SideEffect`. `SideEffect` всегда выполняется **после composition**. Они записываются как функции для вызова, когда изменения соответствующего дерева **уже применены**. Это эффекты «сбоку», не привязанные к жизненному циклу Composable — не будет автоматической отмены при выходе из Composition и повторного запуска при recomposition. Такой эффект **не хранится в slot table** и при падении composition просто отбрасывается. Подробнее — в главе про обработчики эффектов; здесь важно, что они записываются через Composer.

Хранение CompositionLocal

Composer даёт возможность регистрировать `CompositionLocal` и получать значения по ключу. Вызовы `CompositionLocal.current` опираются на это. Provider и его значения тоже хранятся группой в slot table.

Хранение информации об исходниках

Composer сохраняет информацию об исходниках в виде `CompositionData`, собранной во время composition, для инструментов Compose.

Связывание Composition через CompositionContext

Composition не одна — есть дерево composition и subcomposition. Subcomposition — Composition, создаваемая inline в контексте текущей для независимой инвалидации.

Subcomposition связана с родительской Composition ссылкой на родительский `CompositionContext`. Контекст связывает composition и subcomposition в дерево и обеспечивает прозрачное разрешение/распространение `CompositionLoc` и инвалидаций вниз по дереву, как будто это одна Composition. Сам `CompositionContext` тоже записывается в slot table как группа.

Subcomposition обычно создаётся через `rememberCompositionContext`:

```
1 @Composable fun rememberCompositionContext(): CompositionContext
 ↪ {
2   return currentComposer.buildContext()
3 }
```

Composables.kt

Функция запоминает новую Composition в текущей позиции в slot table или возвращает уже запомненную. Используется для создания Subcomposition там, где нужна отдельная Composition: `VectorPainter`, `Dialog`, `SubcomposeLayout`, `Popup`, `AndroidView` (обёртка для интеграции Android View в composable-деревья).

Доступ к текущему State snapshot

Composer хранит ссылку на текущий snapshot — снимок значений mutable state и других state-объектов для текущего потока. В snapshot все state-объекты имеют те же значения, что в момент создания snapshot, пока их явно не изменят в нём. Подробнее — в главе об управлении состоянием.

Навигация по узлам

Навигация по дереву узлов выполняется Applier не напрямую: при обходе reader записываются все позиции узлов в массив `downNodes`, и

при материализации навигации все «down» проигрываются в Applier. Если «up» записан до соответствующего «down», он просто убирается из стека downNodes как сокращение.

Синхронизация reader и writer

На низком уровне: так как группы могут вставляться, удаляться или перемещаться, позиция группы у writer может какое-то время отличаться от позиции у reader (пока изменения не применены). Поэтому ведётся delta для учёта разницы; она обновляется при вставках, удалениях и перемещениях и отражает «нереализованное расстояние, на которое writer должен сдвинуться, чтобы совпасть с текущим слотом в reader» (по документации).

Применение изменений

Как мы многократно говорили, за это отвечает `Applier`. Текущий `Composer` делегирует этой абстракции применение всех записанных изменений после composition — это и есть «материализация». Процесс выполняет список Change, в результате обновляется slot table и интерпретируются данные Composition для получения результата.

Рантайм не зависит от реализации `Applier`. Он опирается на публичный контракт, который должны реализовать клиентские библиотеки. Applier — точка интеграции с платформой и зависит от сценария. Контракт:

```
interface Applier&lt;N&gt; {
  val current: N
  fun onBeginChanges() {}
  fun onEndChanges() {}
  fun down(node: N)
  fun up()
  fun insertTopDown(index: Int, instance: N)
  fun insertBottomUp(index: Int, instance: N)
  fun remove(index: Int, count: Int)
  fun move(from: Int, to: Int, count: Int)
  fun clear()
}
```

Applier.kt

Параметр типа `N` в контракте — тип узлов, к которым применяются изменения. Поэтому Compose может работать с произвольными графами вызовов

и деревьями узлов. Есть операции обхода дерева, вставки, удаления и перемещения узлов; тип узлов и способ их вставки не заданы. Спойлер: **это делегируется самим узлам**.

Контракт также задаёт удаление дочерних узлов в диапазоне и перемещение дочерних узлов. Операция `clear` сбрасывает состояние к корню и удаляет все узлы, подготавливая Applier и корень к новой composition.

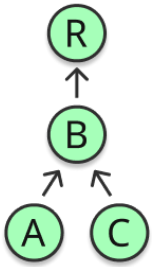
Applier обходит всё дерево и применяет изменения. Обход может быть сверху вниз или снизу вверх. Хранится ссылка на текущий посещаемый узел. Есть вызовы начала/окончания применения изменений (Composer вызывает их до и после) и средства вставки top-down или bottom-up и навигации вниз (к дочернему) или вверх (к родителю).

Производительность при построении дерева узлов

Важное различие — строить дерево сверху вниз или снизу вверх. Пример из официальной документации.

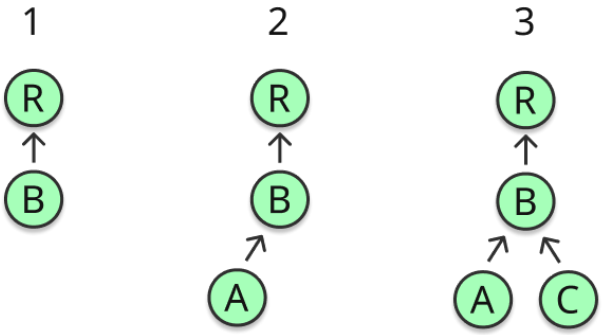
Вставка top-down

Дерево:



tree1

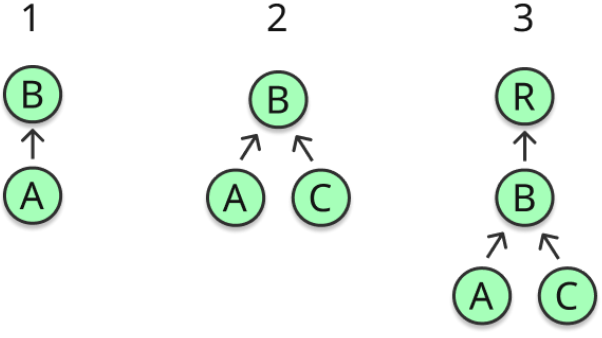
При построении сверху вниз: вставить `B` в `R`, затем `A` в `B`, затем `C` в `B`:



tree2

Вставка bottom-up

Снизу вверх: сначала вставить `A` и `C` в `B`, затем дерево `B` в `R`.



tree3

Производительность top-down и bottom-up может сильно различаться. Выбор за реализацией Applier и часто зависит от числа узлов, которых нужно уведомить при вставке каждого ребёнка. Если при вставке узла нужно уведомлять всех предков, при top-down каждая вставка может уведомлять много узлов (родитель, родитель родителя…), и число растёт экспоненциально. При bottom-up уведомляется только непосредственный родитель, так как он ещё не присоединён к дереву. Если стратегия — уведомлять детей, может быть наоборот. Итог: выбор стратегии вставки зависит от дерева и того, как нужно распространять уведомления; важно выбрать одну стратегию и не

смешивать.

Как применяются изменения

Клиентские библиотеки реализуют интерфейс `Applier`; для Android UI пример — `UiApplier`. По нему видно, что значит «применить узел» и как получаются видимые на экране компоненты.

Реализация узкая:

```
internal class UiApplier(
    root: LayoutNode
) : AbstractApplier&lt;LayoutNode&gt;(root) {

  override fun insertTopDown(index: Int, instance: LayoutNode) {
    // Ignored.
  }

  override fun insertBottomUp(index: Int, instance: LayoutNode)
  ↪ {
    current.insertAt(index, instance)
  }

  override fun remove(index: Int, count: Int) {
    current.removeAt(index, count)
  }

  override fun move(from: Int, to: Int, count: Int) {
    current.move(from, to, count)
  }

  override fun onClear() {
    root.removeAll()
  }

  override fun onEndChanges() {
    super.onEndChanges()
    (root.owner as?
  ↪ AndroidComposeView)?.clearInvalidObservations()
  }
}
```

UiApplier.kt

Параметр типа `N` зафиксирован как `LayoutNode` — тип узла Compose UI для отображаемых UI-узлов.

Видно наследование от `AbstractApplier` — дефолтная реализация хранит посещённые узлы в `Stack`: при спуске узел добавляется, при подъёме снимается с вершины. Это типично для applier, поэтому вынесено в общий базовый класс.

`insertTopDown` в `UiApplier` игнорируется — вставки выполняются снизу вверх на Android. Как сказано, важно выбрать одну стратегию. Снизу вверх здесь уместнее, чтобы избежать дублирующих уведомлений при вставке ребёнка.

Методы вставки, удаления и перемещения **делегируются самому узлу**. `LayoutNode` — модель UI-узла в Compose UI, он знает о родителе и детях. Вставка — присоединение к новому родителю в заданную позицию. Перемещение — переупорядочивание списка детей родителя. Удаление — удаление из списка.

По завершении применения изменений вызывается `onEndChanges()` (перед применением предполагается вызов `onBeginChanges()`). На корневом узле owner выполняется финальное действие — очистка отложенных invalid observations. Это наблюдения за snapshot для автоматического перезапуска layout или draw при изменении зависимых значений (например, при добавлении, вставке, замене или перемещении узлов).

Присоединение и отрисовка узлов

Как вставка узла в дерево (присоединение к родителю) в итоге приводит к отображению на экране? **Узел сам умеет присоединяться и отрисовываться.**

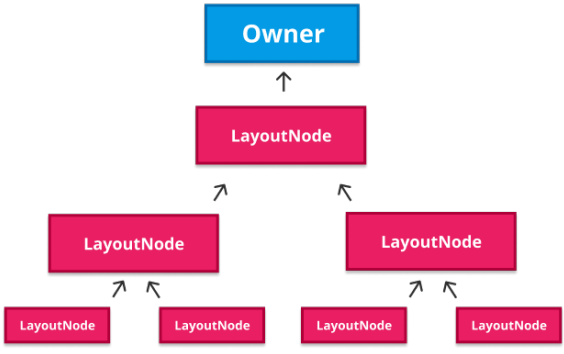
Кратко: полный разбор в главе 4 (Compose UI). Этого достаточно, чтобы замкнуть картину.

`LayoutNode` — тип узла для Android UI. При делегировании вставки ему в `UiApplier` происходит следующее:

- Проверка условий вставки (например, у узла ещё нет родителя).
- Инвалидация списка детей, отсортированных по Z-index (параллельный список для порядка отрисовки). Инвалидация заставляет список пересоздаваться при следующем обращении.
- Присоединение узла к родителю и к его `Owner` (см. ниже).

- Вызов invalidate.

Owner находится в корне дерева и реализует **связь composable-дерева с системой View**. Это тонкий слой интеграции с Android; реализация — `AndroidComposeView` (обычный `View`). Layout, draw, ввод и доступность завязаны на owner. `LayoutNode` должен быть присоединён к `Owner`, чтобы отображаться; owner должен совпадать с owner родителя. Owner — часть Compose UI. После присоединения можно вызвать `invalidate` через Owner для отрисовки дерева.



Иерархия LayoutNode

Точка интеграции — установка Owner при вызове `setContent` из Activity, Fragment или ComposeView. Создаётся `AndroidComposeView`, присоединяется к иерархии View и устанавливается как Owner для вызова invalidate по требованию.

Итог: мы проследили, как Compose UI материализует дерево узлов для Android. Подробнее — в следующей главе.

Цикл замкнут, но мы пока не разобрали сам процесс composition. Пора к нему.

Composition

Мы много узнали о Composer: как он записывает изменения для slot table, как изменения эмитируются при выполнении Composable во время composition и как записанные изменения применяются в конце. Но мы ещё не сказали, кто создаёт Composition, как и когда это происходит и какие шаги задействованы. Composition — недостающий кусок.

Мы говорили, что у Composer есть ссылка на Composition; может показаться,

что Composition создаётся и принадлежит Composer. На деле наоборот: при создании Composition он сам создаёт Composer. Composer доступен через механизм `currentComposer` и используется для создания и обновления дерева, которым управляет Composition.

Точка входа в Jetpack Compose runtime для клиентских библиотек состоит **из двух частей**:

- Написание Composable-функций — они эмитируют нужную информацию и связывают сценарий с рантаймом.
- Composable-функции не выполнятся без процесса composition. Нужна вторая точка входа — `setContent`. Это слой интеграции с платформой; здесь создаётся и запускается `Composition`.

Создание Composition

На Android это может быть вызов `ViewGroup.setContent`, возвращающий новый `Composition`:

```
internal fun ViewGroup.setContent(
  parent: CompositionContext,
  content: @Composable () -&gt; Unit
): Composition {
  // ...
  val composeView = ...
  return doSetContent(composeView, parent, content)
}

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

```
22   return wrapped
23 }
```

Wrapper.android.kt

`WrappedComposition` — декоратор, связывающий `Composition` с `AndroidCompose` и с системой View Android. Он запускает контролируемые эффекты (например, видимость клавиатуры, доступность) и передаёт информацию об Android `Context` в Composition как `CompositionLocal` (контекст, конфигурация, текущий LifecycleOwner, savedStateRegistryOwner, view owner и т.д.). Так всё это становится неявно доступным в Composable-функциях.

В Composition передаётся экземпляр `UiApplier`, изначально указывающий на корневой `LayoutNode` (Applier — посетитель узлов, стартует с корня). Явно видно, что реализацию `Applier` выбирает клиентская библиотека.

В конце вызывается `composition.setContent(content)`. `Composition#setConte` задаёт содержимое Composition.

Другой пример — `VectorPainter` в Compose UI для отрисовки векторов. Vector painter создаёт и хранит свой `Composition`:

```
 1 @Composable
 2 internal fun RenderVector(
 3   name: String,
 4   viewportWidth: Float,
 5   viewportHeight: Float,
 6   content: @Composable (viewportWidth: Float, viewportHeight:
 ↪  Float) -&gt; Unit
 7 ) {
 8   // ...
 9   val composition = composeVector(rememberCompositionContext(),
  ↪  content)
10
11   DisposableEffect(composition) {
12     onDispose {
13       composition.dispose() // composition needs to be disposed
 ↪  in the end!
14     }
15   }
16 }
17
```

```
18 private fun composeVector(
19   parent: CompositionContext,
20   composable: @Composable (viewportWidth: Float,
 ↪  viewportHeight: Float) -&gt; Unit
21 ): Composition {
22   val existing = composition
23   val next = if (existing == null || existing.isDisposed) {
24     Composition(VectorApplier(vector.root), parent) // Here!
25   } else {
26     existing
27   }
28   composition = next
29   next.setContent {
30     composable(vector.viewportWidth, vector.viewportHeight)
31   }
32   return next
33 }
```

VectorPainter.kt

Другой выбор Applier — `VectorApplier` с корневым узлом дерева векторов (`VNode`). Подробнее — в главе о продвинутых сценариях Compose.

Ещё пример — `SubcomposeLayout` в Compose UI: Layout со своей Composition для subcompose контента во время фазы измерения (когда размер родителя нужен для composition детей).

При создании Composition можно передать родительский `CompositionContext` (он может быть `null`). Родительский контекст связывает новую composition с существующей логически, чтобы инвалидации и `CompositionLocal` разрешались через composition, как будто это одна. При создании можно также передать recompose context — `CoroutineContext` для Applier при применении изменений; по умолчанию от Recomposer (`EmptyCoroutineContext`), на Android обычно `AndroidUiDispatcher.Main`.

Composition нужно освобождать — `composition.dispose()`, когда он больше не нужен. Иногда освобождение скрыто (например, за lifecycle observer в `ViewGroup.setContent`), но оно всегда есть. Composition привязан к своему owner.

Процесс первичной composition

После создания Composition следует вызов `composition.setContent(content)` (см. фрагменты выше). Им изначально заполняется Composition (slot table).

Вызов делегируется **родительской** Composition для запуска первичной composition (Composition и Subcomposition связаны через родительский `CompositionContext`):

```
override fun setContent(content: @Composable () -&gt; Unit) {
  // ...
  this.composable = content
  parent.composeInitial(this, composable) // `this` is the
  ↪ current Composition
}
```

Composition.kt

Для Subcomposition родитель — другая Composition; для корневой — `Recomposer`. В любом случае логика первичной composition в итоге опирается на Recomposer: у Subcomposition вызов `composeInitial` делегируется родителю до корневой Composition.

Вызов `parent.composeInitial(composition, content)` сводится к `recomposer.composeInitial(composition, content)`. Recomposer делает следующее для заполнения первичной Composition:

- Берётся **snapshot** текущих значений всех State. Значения изолируются от изменений в других snapshot. Snapshot **мутабелен** и потокобезопасен: изменения State внутри него не затрагивают другие snapshot; позже все изменения атомарно синхронизируются с глобальным состоянием.
- Значения State в этом snapshot можно менять только внутри блока при вызове `snapshot.enter(block: () -> T)`.
- При создании snapshot Recomposer передаёт наблюдателей за чтениями и записями этих State, чтобы Composition получал уведомления и помечал затронутые recompose scope как `used` для последующей recomposition.
- Выполняется вход в snapshot — `snapshot.enter(block)` с блоком `composition.composeContent(content)`. **Здесь и происходит composition**. Вход в snapshot даёт Recomposer отслеживать чтения и записи State во время composition.
- Процесс composition делегируется Composer (подробнее ниже).
- После composition все изменения State остаются в текущем snapshot; их нужно распространить в глобальное состояние через `snapshot.apply()`.

Примерный порядок первичной composition. Система State snapshot подробнее — в следующей главе.

Сам процесс composition, делегированный Composer, в общих чертах:

- Composition не может начаться, если уже идёт — выбрасывается исключение, новая Composition отбрасывается. Повторный вход не поддерживается.
- При наличии отложенных инвалидаций они копируются в список инвалидаций Composer для RecomposeScope.
- Устанавливается флаг `isComposing = true`.
- Вызывается `startRoot()` — начало корневой группы в slot table и инициализация структур.
- `startGroup` для группы `content` в slot table.
- Вызов лямбды `content` — эмитирование изменений.
- `endGroup` — конец группы.
- `endRoot()` — конец composition.
- `isComposing = false`.
- Очистка временных структур.

Применение изменений после первичной composition

После первичной composition Applier уведомляется о применении записанных изменений: `composition.applyChanges()`. Composition вызывает `applier.onBeg` выполняет все изменения из списка, передавая им Applier и SlotWriter, затем `applier.onEndChanges()`.

После этого диспатчатся все зарегистрированные `RememberedObserver` — классы, реализующие `RememberObserver`, уведомляются о входе и выходе из Composition. Так работают `LaunchedEffect`, `DisposableEffect` и др. — привязка эффекта к жизненному циклу Composable в Composition.

Затем в порядке записи вызываются все `SideEffect`.

Дополнительно о Composition

Composition знает об отложенных инвалидациях для recomposition и о том, идёт ли сейчас composition. Это используется для немедленного применения инвалидаций (если composition идёт) или откладывания; Recomposer может отменять recomposition, когда composition активна.

Рантайм использует вариант Composition — `ControlledComposition` с дополнительными методами внешнего управления. Recomposer может

оркестрировать инвалидации и recomposition через функции вроде `composeConter` или `recompose`.

Composition позволяет проверить, наблюдает ли он за набором объектов, чтобы запускать recomposition при их изменении. Например, Recomposer запускает recomposition дочерней composition при изменении `CompositionLocal` в родительской (composition связаны через родительский CompositionContext).

При ошибке во время composition процесс может быть прерван — по сути сброс Composer и всех ссылок/стеков.

Composer считает, что выполняется пропуск recomposition, когда не идёт вставка и не переиспользование, нет инвалидированных provider и текущий `RecomposeScope` не требует recomposition. Умная recomposition разбирается в отдельной главе.

Recomposer

Мы знаем, как происходит первичная composition и что такое RecomposeScope и инвалидация. Но как именно работает Recomposer? Как и когда он создаётся и начинает слушать инвалидации для автоматической recomposition?

Recomposer управляет `ControlledComposition` и запускает recomposition при необходимости для применения обновлений. Он также решает, на каком потоке выполнять composition/recomposition и применять изменения.

Запуск Recomposer

Точка входа в Jetpack Compose для клиентских библиотек — создание Composition и вызов `setContent` (раздел **Создание Composition**). При создании Composition нужно передать родителя. Для корневой Composition родитель — `Recomposer`, поэтому его создают в тот же момент.

Эта точка входа — связь платформы с Compose runtime; код поставляет клиент. На Android — Compose UI: создаётся Composition (внутри — свой Composer) и Recomposer как родитель.

У каждого сценария на платформе может быть своя Composition и свой Recomposer.

На Android при вызове `ViewGroup.setContent` после нескольких уровней создание родительского контекста делегируется фабрике Recomposer:

```
fun interface WindowRecomposerFactory {
  fun createRecomposer(windowRootView: View): Recomposer
  companion object {
    val LifecycleAware: WindowRecomposerFactory =
  ↪ WindowRecomposerFactory { rootView\
 -&gt;
      rootView.createLifecycleAwareViewTreeRecomposer()
    }
  }
}
```

Фабрика создаёт Recomposer для текущего окна. Корневой view нужен, чтобы Recomposer был **lifecycle-aware** — привязан к `ViewTreeLifecycleOwner` и мог останавливаться при откреплении дерева View (важно для отсутствия утечек; процесс моделируется приостанавливаемой функцией).

Важно: в Compose UI всё на UI координируется через `AndroidUiDispatcher`, связанный с `Choreographer` и handler основного `Looper`. Диспетчер выполняет обработку в callback handler или на этапе кадра анимации Choreographer — **что наступит раньше**. У него есть связанный `MonotonicFrameClock` для координации кадров через `suspend` — основа UX в Compose, в том числе анимаций.

Фабрика сначала создаёт `PausableMonotonicFrameClock` — обёртку над монотонными часами `AndroidUiDispatcher` с возможностью приостанавливать диспетчеризацию `withFrameNanos` до возобновления. Нужно, когда кадры **не должны** производиться (например, окно не видно).

Любой `MonotonicFrameClock` — также `CoroutineContext.Element`. При создании Recomposer передаётся `CoroutineContext` — комбинация контекста текущего потока от `AndroidUiDispatcher` и только что созданного pausable clock:

```
val contextWithClock = currentThreadContext + (pausableClock ?:
↪ EmptyCoroutineContex\
t)
val recomposer = Recomposer(effectCoroutineContext =
↪ contextWithClock)
```

WindowRecomposer.android

Этот контекст используется Recomposer для внутренней `Job`, чтобы

composition/recomposition эффекты можно было отменять при остановке Recomposer (например, при уничтожении или откреплении окна). **Этот контекст используется для применения изменений** после composition/recompos и по умолчанию для запуска эффектов в `LaunchedEffect` (эффекты стартуют в том же потоке, что и применение изменений — на Android обычно main; внутри эффектов можно переключаться на другие потоки).

`LaunchedEffect` разбирается в соответствующей главе. Все обработчики эффектов — Composable и эмитируют изменения; `LaunchedEffect` записывается в slot table и привязан к жизненному циклу Composition, в отличие от `SideEffect`.

Создаётся scope корутин с тем же контекстом: `val runRecomposeScope = CoroutineScope(contextWithClock)`. В нём запускается job recomposition (suspend-функция), ожидающая инвалидаций и запускающая recomposition. Фрагмент кода:

```
viewTreeLifecycleOwner.lifecycle.addObserver(
  object : LifecycleEventObserver {
    override fun onStateChanged(lifecycleOwner: LifecycleOwner,
  ↪ event: Lifecycle.Eve\
nt) {
      val self = this
      when (event) {
        Lifecycle.Event.ON_CREATE -&gt;
          runRecomposeScope.launch(start =
  ↪ CoroutineStart.UNDISPATCHED) {
          try {
            recomposer.runRecomposeAndApplyChanges()
          } finally {
            // After completion or cancellation
            lifecycleOwner.lifecycle.removeObserver(self)
          }
        }
        Lifecycle.Event.ON_START -&gt; pausableClock?.resume()
        Lifecycle.Event.ON_STOP -&gt; pausableClock?.pause()
        Lifecycle.Event.ON_DESTROY -&gt; {
          recomposer.cancel()
        }
      }
    }
  }
```

```
25 )
```

WindowRecomposer.android.kt

Наблюдатель подписывается на lifecycle дерева View: при ON_START возобновляет pausable clock, при ON_STOP приостанавливает, при ON_DESTROY отменяет Recomposer, при ON_CREATE запускает job recomposition.

Job запускается вызовом `recomposer.runRecomposeAndApplyChanges()` — приостанавливаемая функция, ожидающая инвалидации связанных Composer (и их RecomposeScope), выполняющая recomposition и применение изменений к соответствующим Composition.

Так Compose UI запускает Recomposer, привязанный к жизненному циклу Android. Напоминание — создание composition при установке контента для ViewGroup:

```
 1 internal fun ViewGroup.setContent(
 2   parent: CompositionContext, // Recomposer is passed here!
 3   content: @Composable () -&gt; Unit
 4 ): Composition {
 5   // ...
 6   val composeView = ...
 7   return doSetContent(composeView, parent, content)
 8 }
 9
10 private fun doSetContent(
11   owner: AndroidComposeView,
12   parent: CompositionContext,
13   content: @Composable () -&gt; Unit
14 ): Composition {
15   // ...
16   val original = Composition(UiApplier(owner.root), parent) //
 ↪ Here!
17   val wrapped = owner.view.getTag(R.id.wrapped_composition_tag)
18     as? WrappedComposition ?: WrappedComposition(owner,
 ↪ original).also {
19       owner.view.setTag(R.id.wrapped_composition_tag, it)
20     }
21   wrapped.setContent(content)
```

```
22    return wrapped
23 }
```

Wrapper.android.kt

Параметр `parent` здесь — Recomposer; его передаёт вызывающий `setContent` (для этого сценария — `AbstractComposeView`).

Процесс recomposition

Функция `recomposer.runRecomposeAndApplyChanges()` запускает ожидание инвалидаций и автоматическую recomposition. Кратко шаги:

Ранее мы видели, что изменения State snapshot применяются в своём snapshot, затем распространяются в глобальное состояние через `snapshot.apply()`. При вызове `recomposer.runRecomposeAndApplyChanges()` первым делом регистрируется наблюдатель за этим распространением. При распространении наблюдатель просыпается и добавляет изменения в список инвалидаций snapshot, которые передаются всем известным Composer для записи частей composition, требующих recomposition. Проще: этот наблюдатель — ступенька для автоматической recomposition при изменении State.

После регистрации наблюдателя Recomposer инвалидирует все Composition, полагая, что всё изменилось — старт с чистого листа. Затем приостанавливается до появления работы для recomposition. «Работа» — отложенные инвалидации State snapshot или инвалидации composition от RecomposeScope.

Далее Recomposer использует переданный при создании монотонный clock и вызывает `parentFrameClock.withFrameNanos {}`, ожидая следующий кадр. Остальная работа выполняется в этот момент — объединение изменений к кадру.

Внутри блока сначала диспатчатся кадры монотонного clock для ожидающих (например, анимаций); это может породить новые инвалидации.

Затем Recomposer берёт все отложенные инвалидации snapshot (изменённые с прошлого вызова recompose значения State) и записывает их в composer как отложенные recomposition.

Могут быть инвалидированные Composition (через `composition.invalidate()`), например при записи State в лямбде Composable. Для каждой выполняется recomposition (см. ниже) и Composition добавляется в список с отложенными изменениями.

Recomposition — пересчёт всех нужных `Change` для состояния Composition (slot table) и материализованного дерева (Applier), как мы уже разбирали. Код тот же, что и для первичной composition.

Затем находятся возможные «хвостовые» recomposition из-за изменений в другой composition (например, изменение `CompositionLocal` в родителе, прочитанного в дочерней composition) и тоже планируются.

Наконец, по всем Composition с отложенными изменениями вызывается `composition.applyChanges()`, после чего обновляется состояние Recomposer.

Параллельная recomposition

Recomposer может выполнять recomposition параллельно, хотя Compose UI этим не пользуется. Другие клиентские библиотеки могут опираться на это.

У Recomposer есть вариант `runRecomposeAndApplyChanges` — `runRecomposeConc`
Это тоже приостанавливаемая функция ожидания инвалидаций State и автоматической recomposition, но recomposition инвалидированных Composition выполняется в переданном извне `CoroutineContext`:

```
1 suspend fun runRecomposeConcurrentlyAndApplyChanges(
2   recomposeCoroutineContext: CoroutineContext
3 ) { /* ... */ }
```

Recomposer.kt

Функция создаёт свой `CoroutineScope` с переданным контекстом и использует его для запуска и координации дочерних job параллельных recomposition.

Состояния Recomposer

В течение жизни Recomposer переключается между состояниями:

```
1 enum class State {
2   ShutDown,
3   ShuttingDown,
4   Inactive,
5   InactivePendingWork,
6   Idle,
7   PendingWork
8 }
```

Recomposer.kt

По kdoc, значения состояний:

- `ShutDown`: Recomposer отменён, очистка завершена. Использовать нельзя.
- `ShuttingDown`: Recomposer отменён, очистка в процессе. Использовать нельзя.
- `Inactive`: Recomposer игнорирует инвалидации от Composer и не запускает recomposition. Нужно вызвать `runRecomposeAndApplyChanges` для начала прослушивания. Начальное состояние после создания.
- `InactivePendingWork`: Recomposer неактивен, но уже есть отложенные эффекты, ожидающие кадра; кадр будет произведён при запуске.
- `Idle`: Recomposer отслеживает инвалидации composition и snapshot, работы сейчас нет.
- `PendingWork`: Recomposer уведомлён об отложенной работе и выполняет её или ждёт возможности (что такое «отложенная работа», мы уже описали).

4. Compose UI

Когда говорят о Jetpack Compose, обычно имеют в виду всё вместе: компилятор, рантайм и Compose UI. В предыдущих главах мы разобрали компилятор и то, как он обеспечивает оптимизации и возможности рантайма, затем сам рантайм и то, как в нём сосредоточена основная механика Compose. Теперь очередь Compose UI — клиентской библиотеки для рантайма.

Краткое уточнение: в книге Compose UI выбран как пример клиентской библиотеки для Compose runtime, но есть и другие — например Compose for Web от JetBrains или Mosaic (консольный UI от Jake Wharton). Последняя глава книги как раз посвящена тому, как писать клиентские библиотеки для Jetpack Compose.

Интеграция UI с Compose runtime

Compose UI — **Kotlin multiplatform** фреймворк. Он даёт строительные блоки и механику для эмитирования UI через Composable-функции. Кроме того, библиотека включает Android- и Desktop-исходники с слоями интеграции для Android и Desktop.

JetBrains ведёт Desktop, Google — Android и общий код. И Android, и Desktop

опираются на общий sourceset. Compose for Web пока вынесен из Compose UI и строится на DOM.

Цель интеграции UI с Compose runtime — построить дерево раскладки, которое пользователь видит на экране. Это дерево создаётся и обновляется выполнением Composable-функций, эмитирующих UI. Тип узла дерева известен только Compose UI, рантайм от него не зависит. Хотя Compose UI уже мультиплатформенный, его типы узлов пока поддерживаются только на Android и Desktop. Другие библиотеки (например Compose for Web) используют свои типы. Поэтому типы узлов, эмитируемые клиентской библиотекой, должны быть известны только ей, а рантайм делегирует ей вставку, удаление, перемещение и замену узлов. Подробнее — ниже в главе.

В построении и обновлении дерева раскладки участвуют первичная composition и последующие recomposition. Они выполняют наши Composable-функции, те планируют изменения (вставка, удаление, перемещение, замена узлов). Получается список изменений, который затем обходится с помощью `Applier`, чтобы превратить их в реальные изменения дерева для пользователя. При первичной composition изменения вставляют все узлы и строят дерево; при recomposition — обновляют его. Recomposition запускается при изменении входных данных Composable (параметры или читаемый mutable state).

В предыдущих главах мы бегло это затрагивали; эта глава развивает тему.

От запланированных изменений к реальным изменениям дерева

При выполнении Composable во время composition или recomposition они эмитируют изменения. Плюс используется побочная таблица `Composition` (далее с заглавной буквы, чтобы отличать от процесса composition). В ней хранятся данные для отображения выполнения Composable (запланированных изменений) в фактические изменения дерева узлов.

В приложении на Compose UI может быть столько Composition, сколько деревьев узлов нужно представить. Пока мы не говорили, что Composition может быть несколько — но их действительно может быть несколько. Ниже мы разберём, как строится дерево раскладки и какие типы узлов используются.

Composition с точки зрения Compose UI

Типичная точка входа из Compose UI в рантайм на Android — вызов `setContent`, например для экрана:

```
class MainActivity : ComponentActivity() {
  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContent {
      MaterialTheme {
        Text(&amp;quot;Hello Compose!&amp;quot;)
      }
    }
  }
}
```

MainActivity.kt

Экран (Activity/Fragment) — не единственное место вызова `setContent`. Он может быть и внутри иерархии View, например через `ComposeView` (гибридное Android-приложение):

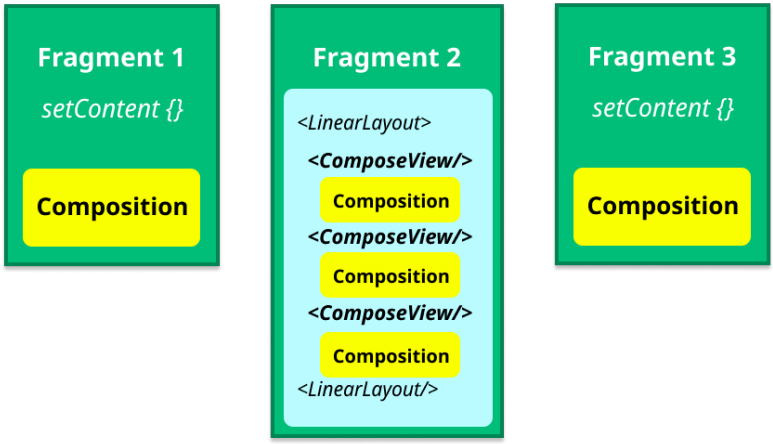
```
ComposeView(requireContext()).apply {
  setContent {
    MaterialTheme {
      Text(&amp;quot;Hello Compose!&amp;quot;)
    }
  }
}
```

ComposeView.setContent

В примере view создаётся программно; он может быть и частью layout из XML.

Функция `setContent` создаёт новую корневую Composition и по возможности переиспользует её. Я называю их «корневыми», так как каждая размещает своё независимое composable-дерево. Эти composition между собой не связаны. Сложность каждой зависит от представляемого UI.

В приложении может быть несколько деревьев узлов, каждое со своей Composition. Пример: приложение с тремя Fragment (см. рисунок). Fragment 1 и 3 вызывают `setContent` для своих composable-деревьев, Fragment 2 объявляет несколько `ComposeView` в layout и вызывает `setContent` для каждого. В итоге — 5 корневых Composition, все независимы.



Несколько корневых Composition

Для построения любой из этих иерархий связанный `Composer` запускает процесс composition. Выполняются все Composable внутри соответствующего вызова `setContent`, они эмитируют изменения. В Compose UI это изменения вставки, перемещения или замены UI-узлов, эмитируемые обычными UI-блоками: `Box`, `Column`, `LazyColumn` и т.д. Хотя они часто из разных библиотек (`foundation`, `material`), все в итоге определены как `Layout` (`compose-ui`) и эмитируют один тип узла — `LayoutNode`.

`LayoutNode` уже упоминался в предыдущей главе — представление UI-блока и самый частый тип узла для корневой Composition в Compose UI.

Любой Composable `Layout` эмитирует узел `LayoutNode` в composition через `ReusableComposeNode` (контракт `ComposeUiNode` реализован у `LayoutNode`):

```
@Composable inline fun Layout(
    content: @Composable () -&gt; Unit,
    modifier: Modifier = Modifier,
    measurePolicy: MeasurePolicy
) {
  val density = LocalDensity.current
  val layoutDirection = LocalLayoutDirection.current
  val viewConfiguration = LocalViewConfiguration.current

  // Emits a LayoutNode!
```

```
11   ReusableComposeNode&lt;ComposeUiNode, Applier&lt;Any&gt;&gt;(
12     factory = { LayoutNode() },
13     update = {
14       set(measurePolicy, { this.measurePolicy = it })
15       set(density, { this.density = it })
16       set(layoutDirection, { this.layoutDirection = it })
17       set(viewConfiguration, { this.viewConfiguration = it })
18     },
19     skippableUpdate = materializerOf(modifier),
20     content = content
21   )
22 }
```

Layout.kt

Так эмитируется изменение на вставку или обновление переиспользуемого узла в composition. То же происходит для любых UI-блоков.

Переиспользуемые узлы — оптимизация рантайма Compose. При смене `key` узла Composer может перезапустить контент узла (обновить на месте при recomposition), а не отбрасывать и создавать новый. Для этого composition ведёт себя как при создании нового контента, но slot table обходится как при recomposition. Оптимизация возможна только для узлов, полностью описываемых операциями `set` и `update` в вызове эмитирования, то есть без скрытого внутреннего состояния. Это верно для `LayoutNode`, но не для `AndroidView` — поэтому `AndroidView` использует обычный `ComposeNode`.

`ReusableComposeNode` создаёт узел (через `factory`), инициализирует его (лямбда `update`) и создаёт replaceable group вокруг всего контента. Группе присваивается уникальный ключ. Всё, что эмитируется при вызове лямбды `content` внутри replaceable group, становится дочерними узлами этого узла.

Вызовы `set` в блоке `update` планируют выполнение своих лямбд только при первом создании узла или при изменении значения соответствующего свойства с момента последнего запоминания.

Так `LayoutNode` попадают в каждую из множественных Composition приложения. Может показаться, что в Composition только они — но нет. Есть и другие типы Composition и узлов; об этом — ниже.

Subcomposition с точки зрения Compose UI

Composition существуют не только на корневом уровне. Composition может создаваться глубже в composable-дереве и связываться с родительской. Это **Subcomposition** (подчинённая composition). Мы уже видели, что composition связаны в дерево: у каждой есть ссылка на родительский `CompositionContext` (у корневой родитель — сам `Recomposer`). Так рантайм обеспечивает разрешение и распространение `CompositionLocal` и инвалидаций вниз по дереву, как будто это одна composition.

В Compose UI Subcomposition создают в основном по двум причинам:

- Отложить первичную composition до момента, когда известна некоторая информация.
- Сменить тип узла, порождаемого поддеревом.

Отложенная первичная composition

Пример — `SubcomposeLayout`, аналог `Layout`, который создаёт и запускает отдельную composition на фазе layout. Дочерние Composable могут зависеть от значений, вычисленных в нём. `SubcomposeLayout` используется, например, в `BoxWithConstraints`, который передаёт в блок ограничения родителя, чтобы контент мог от них зависеть. В примере из официальной документации `BoxWithConstraints` выбирает между двумя разными composable в зависимости от доступной `maxHeight`:

```
BoxWithConstraints {
  val rectangleHeight = 100.dp
  if (maxHeight &lt; rectangleHeight * 2) {
    Box(Modifier.size(50.dp,
  ↪ rectangleHeight).background(Color.Blue))
  } else {
    Column {
      Box(Modifier.size(50.dp,
  ↪ rectangleHeight).background(Color.Blue))
      Box(Modifier.size(50.dp,
  ↪ rectangleHeight).background(Color.Gray))
    }
  }
}
```

BoxWithConstraints Sample

Создатель Subcomposition решает, когда выполнять первичную composition; `SubcomposeLayout` делает это на фазе layout, а не при композировании корня.

Subcomposition позволяет перезапускать composition независимо от родителя. В `SubcomposeLayout` при каждом layout параметры лямбды могут меняться — тогда запускается recomposition. С другой стороны, при изменении состояния, читаемого в subcomposition, после выполнения первичной composition планируется recomposition родительской Composition.

По типу узлов `SubcomposeLayout` тоже эмитирует `LayoutNode`, то есть тип узлов поддерева совпадает с родительской Composition. Возникает вопрос: можно ли в одной Composition поддерживать разные типы узлов?

Технически да, если соответствующий `Applier` это допускает — всё упирается в то, что считать типом узла. Если тип общий для нескольких подтипов, разные типы возможны, хотя логика Applier может усложниться. В Compose UI реализации Applier зафиксированы на одном типе узла.

Subcomposition как раз позволяет **использовать в поддереве совершенно другой тип узла** — второй из перечисленных сценариев.

Смена типа узла в поддереве

Пример в Compose UI — composable для векторной графики (например `rememberVectorPainter`).

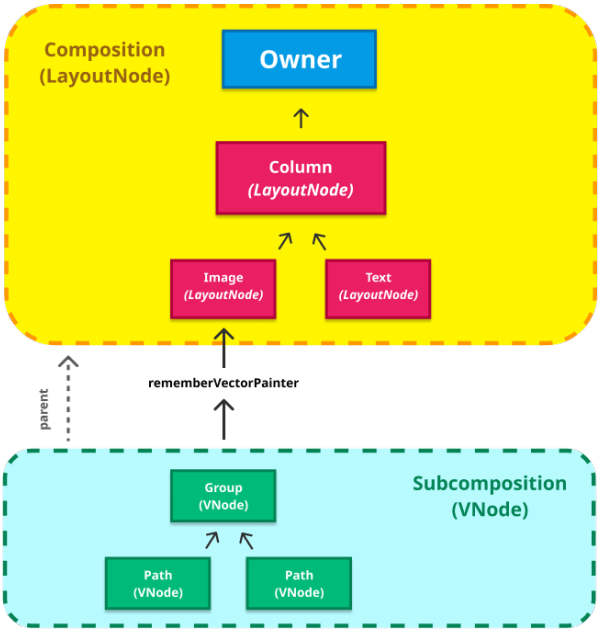
Векторные Composable — хороший кейс: они тоже создают свою Subcomposition и представляют вектор как дерево. При композировании векторный Composable эмитирует в свою Subcomposition другой тип узла — `VNode`, рекурсивный тип для отдельных Path или групп Path.

```
1 @Composable
2 fun MenuButton(onMenuClick: () -&gt; Unit) {
3   Icon(
4     painter = rememberVectorPainter(image = Icons.Rounded.Menu),
5     contentDescription = &quot;Menu button&quot;,
6     modifier = Modifier.clickable { onMenuClick() }
7   )
8 }
```

Vector painter example

Обычно векторы рисуют через `VectorPainter` внутри `Image`, `Icon` или подобного Composable — то есть охватывающий Composable это `Layout` и

эмитирует `LayoutNode` в свою Composition. При этом `VectorPainter` создаёт свою Subcomposition для вектора и связывает её с этой Composition как с родительской. Схема:



Composition и Subcomposition

Так поддерево вектора (Subcomposition) использует другой тип узла — `VNode`.

Векторы моделируются через Subcomposition, чтобы из блока векторного Composable (например `rememberVectorPainter`) можно было обращаться к части `CompositionLocal` родительской Composition (цвета темы, density и т.д.).

Subcomposition для векторов освобождается, когда соответствующий `VectorPainter` выходит из родительской Composition (когда выходит и охватывающий его Composable). Подробнее о жизненном цикле Composable — в одной из следующих глав.

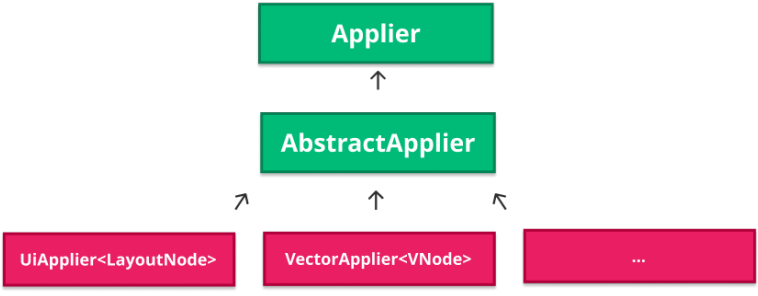
Итак, картина того, как выглядит дерево в типичном приложении Compose UI (Android или Desktop), с корневыми Composition и Subcomposition, яснее. Дальше — вторая сторона интеграции с платформой: материализация изменений для отображения на экране.

Отражение изменений в UI

Мы разобрали, как UI-узлы эмитируются и передаются в рантайм при первичной composition и recomposition. Дальше рантайм выполняет свою работу (глава 3). Но это только одна сторона — нужна интеграция, чтобы все эти изменения отразились в реальном UI. Этот процесс часто называют «материализацией» дерева узлов; за него отвечает клиентская библиотека, здесь Compose UI. Подробнее — ниже.

Разные типы Applier

Ранее мы описали `Applier` как абстракцию, через которую рантайм в итоге материализует изменения дерева. Это инвертирует зависимости: рантайм не привязан к платформе. Клиентские библиотеки вроде Compose UI подключают свои реализации `Applier` и **выбирают свои типы узлов** для интеграции с платформой. Схема:



Архитектура Compose

Верхние два блока (Applier и AbstractApplier) — часть Compose runtime. Нижние — примеры реализаций из Compose UI.

`AbstractApplier` — базовая реализация от Compose runtime с общей логикой. В ней **посещённые** узлы хранятся в `Stack` и ведётся ссылка на текущий узел. При спуске по дереву Composer уведомляет Applier вызовом `applier#down(node: N)` — узел кладётся в стек. При подъёме вызывается `applier#up()`, с вершины стека снимается последний узел.

Пример: дерево для материализации:

```
1 Column {
2   Row {
```

```
    Text(&quot;Some text&quot;)
    if (condition) {
      Text(&quot;Some conditional text&quot;)
    }
  }
  if (condition) {
    Text(&quot;Some more conditional text&quot;)
  }
}
```

AbstractApplier example

При изменении `condition` Applier получит: `down` для Column; затем `down` для Row; затем delete (или insert) для условного Text; затем `up` обратно к Column; наконец delete (или insert) для второго условного текста.

Стек и операции `down`/`up` вынесены в `AbstractApplier`, чтобы разные applier разделяли одну и ту же логику навигации независимо от типа узлов. Связь родитель–потомок при этом обеспечивается, и конкретные типы узлов при необходимости могут хранить её сами — как `LayoutNode`, у которого не все операции выполняются во время composition (например, при перерисовке узла Compose UI поднимается по родителям, чтобы найти узел с нужным слоем и вызвать invalidate).

Напоминание из главы 3: дерево узлов можно строить **сверху вниз** или **снизу вверх**, с разными последствиями по производительности в зависимости от числа уведомляемых узлов при вставке. В Compose UI есть примеры обеих стратегий — две реализации `Applier`:

- **`UiApplier`**: для большей части Android UI; тип узла — `LayoutNode`.
- **`VectorApplier`**: для векторной графики; тип узла — `VNode`.

Для корневой Composition с `LayoutNode` и Subcomposition с `VNode` используются оба applier.

`UiApplier` вставляет узлы снизу вверх, чтобы избежать дублирующих уведомлений (схема из главы 2: сначала вставка A и C в B, затем дерева B в R — уведомляется только непосредственный родитель). Для Android UI с глубокой вложенностью это важно.

`VectorApplier` строит дерево сверху вниз. При вставке нового узла пришлось бы уведомлять всех предков, но для векторной графики уведомления никому

не нужны — обе стратегии равнозначны. При вставке ребёнка в `VNode` уведомляется только слушатель этого узла.

Материализация нового LayoutNode

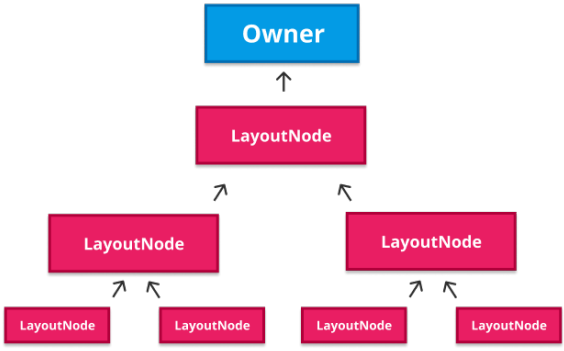
Упрощённый `UiApplier` из Compose UI:

```
internal class UiApplier(
    root: LayoutNode
) : AbstractApplier&lt;LayoutNode&gt;(root) {

  override fun insertTopDown(index: Int, instance: LayoutNode) {
    // Ignored. (The tree is built bottom-up with this one).
  }

  override fun insertBottomUp(index: Int, instance: LayoutNode)
  ↪ {
    current.insertAt(index, instance)
  }

  override fun remove(index: Int, count: Int) {
    current.removeAt(index, count)
  }

  override fun move(from: Int, to: Int, count: Int) {
    current.move(from, to, count)
  }

  override fun onClear() {
    root.removeAll()
  }
  ...
}
```

UiApplier.android.kt

Тип узла зафиксирован как `LayoutNode`; все операции вставки, удаления и перемещения делегируются текущему посещаемому узлу. `LayoutNode` — чистый Kotlin-класс без зависимостей от Android, модель UI-узла для нескольких платформ (Android, Desktop). Он хранит список детей и операции вставки/удаления/перемещения. Узлы связаны в дерево, у каждого есть

ссылка на родителя, все привязаны к одному `Owner`. Схема иерархии:



Дерево LayoutNode

`Owner` — абстракция для интеграции с платформой. На Android это `View` (`AndroidComposeView`) — связь composable-дерева (`LayoutNode`) с системой View. При присоединении, откреплении, переупорядочивании, перемерке или обновлении узла через Owner можно вызвать invalidate (через API View), и изменения отобразятся при следующей отрисовке.

Упрощённо: как новый узел вставляется и материализуется в `LayoutNode#insertA`

```
internal fun insertAt(index: Int, instance: LayoutNode) {
  check(instance._foldedParent == null) {
    &amp;quot;Cannot insert, it already has a parent!&amp;quot;
  }
  check(instance.owner == null) {
    &amp;quot;Cannot insert, it already has an owner!&amp;quot;
  }

  instance._foldedParent = this
  _foldedChildren.add(index, instance)
  onZSortedChildrenInvalidated()

  instance.outerLayoutNodeWrapper.wrappedBy =
↪ innerLayoutNodeWrapper

  val owner = this.owner
  if (owner != null) {
    instance.attach(owner)
```

```
18   }
19 }
```

LayoutNode#insertAt.kt

После проверок текущий узел задаётся родителем вставляемого, новый узел добавляется в список детей, инвалидируется список детей, отсортированных по Z-index (параллельный список для порядка отрисовки). Затем связываются outer и inner `LayoutNodeWrapper` (подробнее в разделе об измерении). В конце узел присоединяется к тому же `Owner`, что и родитель. Упрощённо `attach`:

```
 1 internal fun attach(owner: Owner) {
 2   check(_foldedParent == null || _foldedParent?.owner == owner)
  ↪  {
 3     &quot;Attaching to a different owner than the parent&#39;s
  ↪  owner&quot;
 4   }
 5   val parent = this.parent // [this] is the node being attached
 6
 7   this.owner = owner
 8
 9   if (outerSemantics != null) {
10     owner.onSemanticsChange()
11   }
12   owner.onAttach(this)
13   _foldedChildren.forEach { child -&gt;
14     child.attach(owner)
15   }
16
17   requestRemeasure()
18   parent?.requestRemeasure()
19 }
```

LayoutNode#attach.kt

Здесь проверяется, что у всех дочерних узлов тот же Owner, что у родителя; `attach` вызывается рекурсивно по детям. Затем назначается owner, при наличии семантики уведомляется Owner, запрашивается перемерка для нового узла и родителя — через Owner это приводит к вызову `invalidate`/`requestLayout` и в итоге к появлению узла на экране.

Замыкание цикла

Owner присоединяется к иерархии View при вызове `setContent` в Activity, Fragment или ComposeView — этого не хватало для полной картины.

Кратко по шагам: при `Activity#setContent` создаётся и присоединяется `AndroidComposeView`. При вызове Applier'ом `current.insertAt(index, instance)` для вставки `LayoutNode(C)` новый узел присоединяется и запрашивает перемерку себя и родителя через Owner. Обычно вызывается `AndroidComposeView#invalidate`. Несколько инвалидаций одного View между кадрами просто помечают его как «грязный» — перерисовка будет одна. Затем вызывается `AndroidComposeView#dispatchDraw`: там выполняется перемерка и layout запрошенных узлов, затем отрисовка от корневого `LayoutNode`. Узлы всегда сначала измеряются, затем раскладываются, затем рисуются.

Материализация удаления узлов

Удаление одного или нескольких дочерних узлов устроено похоже. `UiApplier` вызывает `current.removeAt(index, count)`. Родитель перебирает удаляемых детей (с конца), для каждого удаляет ребёнка из списка, инвалидирует Z-список и открепляет ребёнка и всех его потомков (owner = null), запрашивает перемерку родителя. Owner уведомляется при изменении семантики.

Материализация перемещения узлов

То есть переупорядочивание детей. При `current.move(from, to, count)` узлы сначала удаляются через `removeAt`, затем вставляются в новую позицию, запрашивается перемерка родителя.

Материализация очистки всех узлов

Аналогично удалению нескольких узлов: перебор всех детей (с конца), открепление каждого и запрос перемерки родителя.

Измерение в Compose UI

Мы уже знаем, как и когда запрашивается перемерка. Пора разобраться, как измерение реально выполняется.

Любой `LayoutNode` может запросить перемерку через `Owner` — например при присоединении, откреплении или перемещении дочернего узла. View (`Owner`)

помечается как «грязный» (`invalidate`), а узел попадает в **список узлов для перемерки и переразметки**. В следующем проходе отрисовки вызывается `AndroidComposeView#dispatchDraw` (как для любого инвалидированного `ViewGroup`), и `AndroidComposeView` перебирает список и через делегат выполняет нужные действия.

Для каждого узла из списка выполняются 3 шага (в таком порядке):

1. Проверка, нужна ли перемерка узла, и при необходимости — выполнение перемерки.
2. После измерения — проверка, нужна ли переразметка, и при необходимости — выполнение переразметки.
3. Проверка, есть ли отложенные запросы измерения для каких-либо узлов; при наличии — постановка их в список на следующий проход (возврат к шагу 1). Запросы перемерки откладываются, если они возникают уже во время текущего прохода измерения.

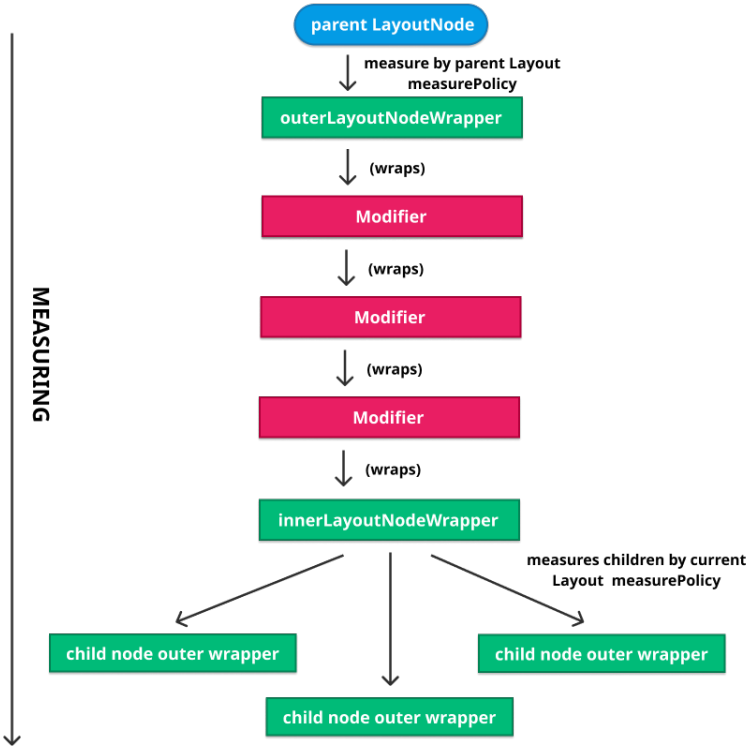
При измерении узла (шаг 1) делегат — внешний `LayoutNodeWrapper`. У каждого `LayoutNode` есть внешний и внутренний wrapper'ы. Внешний отвечает за измерение и отрисовку текущего узла, внутренний — за то же для его детей. Если в результате измерения размер узла изменился и у узла есть родитель, запрашивается перемерка или переразметка родителя.

При вставке узла (`LayoutNode#insertAt`, вызываемом `UiApplier`) выставляется связь между wrapper'ами:

```
1 internal fun insertAt(index: Int, instance: LayoutNode) {
2   ...
3   instance.outerLayoutNodeWrapper.wrappedBy =
 ↪  innerLayoutNodeWrapper
4   ...
5 }
```

LayoutNode#insertAt.kt

У узла могут быть модификаторы, и они тоже **влияют на измерение**, поэтому при измерении нужно учитывать цепочку. **Modifier — вещь без состояния**, поэтому состояние (в т.ч. измеренный размер) хранится в wrapper'ах. У `LayoutNode` есть не только внешний и внутренний wrapper'ы, но и по wrapper'у на каждый модификатор; все они выстроены в цепочку и обрабатываются по порядку. Wrapper модификатора хранит измеренный размер и другие вещи (например, отрисовку для `Modifier.drawBehind()`, hit-test). Схема связи wrapper'ов:



Разрешение модификаторов при измерении

Связь wrapper'ов: родитель через свой `measurePolicy` измеряет внешний wrapper каждого ребёнка; внешний wrapper оборачивает первый модификатор, тот — второй, и т.д., до внутреннего wrapper'а; внутренний снова использует `measurePolicy` текущего узла для измерения детей. Так обеспечивается порядок измерения с учётом модификаторов. С Compose 1.2 в цепочку попадают только `LayoutModifier`; для остальных типов используются обёртки. При отрисовке на последнем шаге внутренний wrapper перебирает детей по Z-индексу и вызывает у каждого draw.

При присоединении нового узла уведомляются все `LayoutNodeWrapper` (у них есть жизненный цикл). При запросе перемерки узла действие передаётся его внешнему `LayoutNodeWrapper`, который использует measure policy родителя; затем по цепочке перемеряются модификаторы и в конце внутренний wrapper перемеряет детей через measure policy текущего узла. Чтения mutable state

внутри measure-lambda (measure policy) записываются — при изменении этого state lambda выполнится снова. После измерения новый размер сравнивается с предыдущим и при изменении запрашивается перемерка родителя.

Политики измерения (Measuring policies)

Когда узел нужно измерить, соответствующий `LayoutNodeWrapper` опирается на политику измерения, переданную при эмиссии узла. В `Layout` политика передаётся снаружи; `LayoutNode` не привязан к конкретной политике. При смене политики запрашивается перемерка. Политики — это лямбда, которую передают в кастомный `Layout`; см. официальную документацию по кастомным лейаутам.

Пример — `Spacer`:

```
@Composable inline fun Layout(
  content: @Composable () -&gt; Unit,
  modifier: Modifier = Modifier,
  measurePolicy: MeasurePolicy
) {
  ...
  ReusableComposeNode&lt;ComposeUiNode, Applier&lt;Any&gt;&gt;(
    factory = { LayoutNode() },
    update = {
      set(measurePolicy, { this.measurePolicy = it })
      ...
    },
    skippableUpdate = materializerOf(modifier),
    content = content
  )
}
```

Layout.kt

```
@Composable
fun Spacer(modifier: Modifier) {
  Layout({}, modifier) { _, constraints -&gt;
    with(constraints) {
      val width = if (hasFixedWidth) maxWidth else 0
      val height = if (hasFixedHeight) maxHeight else 0
      layout(width, height) {}
    }
```

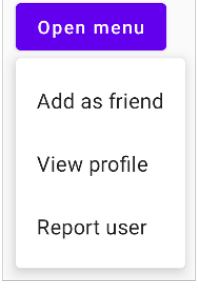
```
 9   }
10 }
```

Spacer.kt

Ещё пример — `Box`: политика зависит от выравнивания и от `propagateMinConstra` много компонентов построены поверх Box. Если детей нет — размер по min ограничениям. Один ребёнок: если не match parent — измеряем его, размер Box по ребёнку; если match parent — размер Box по min ограничениям, ребёнок с фиксированными ограничениями. Несколько детей: сначала измеряются все не-match-parent, считаются boxWidth/boxHeight; затем match-parent дети с ограничениями (boxWidth, boxHeight); в конце layout и размещение через placeInBox. Подробности — в исходниках `Box` и документации.

Внутренние измерения (Intrinsic measurements)

`MeasurePolicy` содержит методы для вычисления внутреннего (intrinsic) размера лейаута — ориентировочного размера при отсутствии ограничений. Внутренние измерения нужны, когда размер ребёнка нужно оценить до измерения (например, выровнять высоту по самому высокому соседу). Двойное измерение в Compose запрещено. У `LayoutNode` есть политика внутренних измерений, зависящая от measure policy. Она даёт: `minIntrinsicWidt` `minIntrinsicHeight(width)`, `maxIntrinsicWidth(height)`, `maxIntrinsicHeigh` — всегда нужна противоположная размерность. Пример: `Modifier.width(Intri` — ширина по максимальной внутренней ширине; так в `DropdownMenuContent` делают ширину колонки по самому широкому пункту меню.



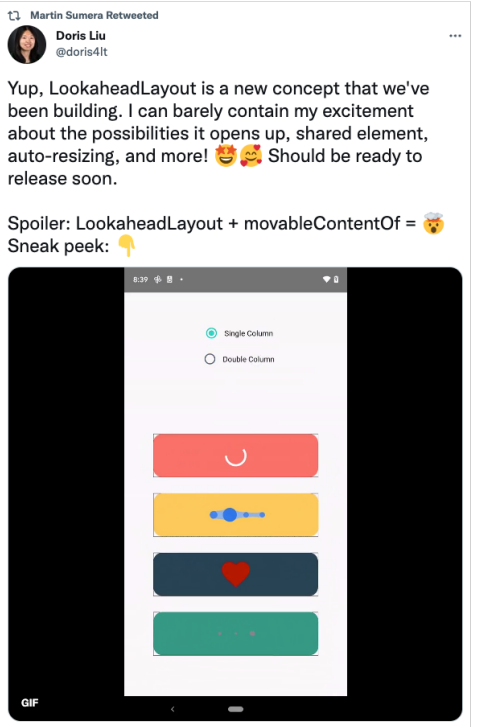
Dropdown Menu

Подробнее: документация по intrinsic measurements.

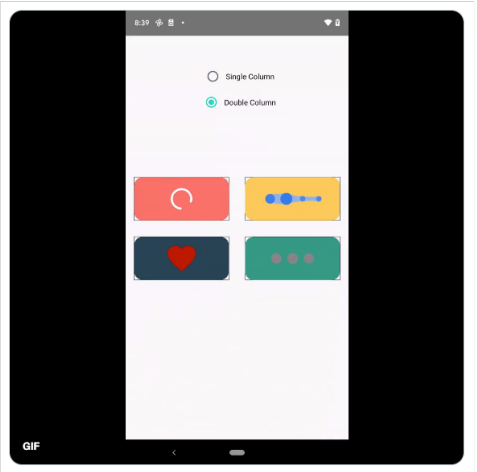
Ограничения лейаута (Layout Constraints)

Ограничения приходят от родительского `LayoutNode` или модификатора: min/max по ширине и высоте в пикселях. Измеренные дети должны попадать в эти границы. Многие лейауты передают ограничения детям без изменений или с нулевым min. Родитель может передать бесконечные max (`Constraints.Infinity`) по одной оси — тогда ребёнок сам выбирает размер по этой оси (например, `LazyColumn` даёт бесконечную высоту). При фиксированном размере выставляют minWidth == maxWidth и minHeight == maxHeight. Пример: `LazyVerticalGrid` задаёт детям фиксированную ширину по числу колонок. `Constraints` смоделированы как inline-класс с одной `Long` и битовыми масками.

LookaheadLayout



LookaheadLayout



LookaheadLayout

`LookaheadLayout` связан с измерением и разметкой: позволяет заранее узнать целевые размеры и позиции при смене состояния (например, для shared element transitions). В твите Doris Liu показаны анимации перехода между состояниями. Работа: во время composition вычисляется «lookahead»-разметка; на её основе можно анимировать переход от текущих размеров/позиций к целевым. Внутри — отдельный проход измерения для lookahead-дерева; детали см. в исходниках и докладах по LookaheadLayout.

Ещё один способ предрасчёта лейаутов

Lookahead даёт альтернативный способ заранее вычислить разметку.

Как это устроено

Описание механизма lookahead-измерения и применения результатов.

Внутренности LookaheadLayout

Детали реализации lookahead-дерева и его связи с основным деревом узлов.

Дополнительные аспекты

Нюансы и граничные случаи при использовании LookaheadLayout.

Моделирование цепочек модификаторов

Модификаторы образуют цепочку; каждый узел хранит список модификаторов. При применении к `LayoutNode` цепочка обходится и для каждого модификатора создаётся или переиспользуется wrapper. Порядок обхода определяет порядок измерения и отрисовки (например, padding, затем background). Подробности — в разделах про установку модификаторов на `LayoutNode` и их «поглощение» узлом.

Установка модификаторов на LayoutNode и отрисовка дерева узлов

Модификаторы передаются в узел через materializer; каждый модификатор сопоставляется с соответствующим `LayoutNodeWrapper`. При изменении цепочки модификаторов узлы wrapper'ов пересоздаются или обновляются. Отрисовка: при вызове draw на узле обходится цепочка wrapper'ов; каждый wrapper может рисовать до или после детей (drawBehind / drawContent / drawFront). Дерево рисуется в порядке Z-индекса детей.

Семантика в Jetpack Compose

Семантическое дерево описывает смысл UI для тестов и accessibility. Узлы семантики создаются для composable'ов и модификаторов с семантическими свойствами (`semantics`, `Modifier.clickable` и т.д.). Компоненты из библиотек `material` и `foundation` обычно уже подключают семантику неявно; в кастомных `Layout` семантику нужно задавать явно. Accessibility и тестирование **должны быть приоритетом**. Подробнее: официальная документация по семантике.

Уведомление об изменениях семантики

Изменения семантики передаются в Android SDK через `Owner`. Библиотека AndroidX Core добавляет `AccessibilityDelegateCompat` для единообразной работы с accessibility на разных версиях системы. `Owner` иерархии `LayoutNode` использует реализацию этого делегата. Реализация обращается к системным accessibility-сервисам через `Context` из `Owner`. При уведомлении об изменении семантики через `Owner` в main looper через `Handler` откладывается проверка изменений в семантическом дереве. Действия по порядку: (1) Сравнение старого и нового дерева на структурные изменения (добавление/удаление детей); при их обнаружении делегат через conflated `Channel` уведомляет

сервисы (код уведомления — suspend-функция в корутине, обрабатывающая изменения батчами раз в 100ms). (2) Сравнение на изменение свойств узлов; при изменении свойств — уведомление через `ViewParent#requestSendAccessib`
(3) Обновление списка предыдущих семантических узлов текущими.

Объединённое и необъединённое семантические деревья

В Jetpack Compose есть два семантических дерева: **объединённое (merged)** и **необъединённое (unmerged)**. Иногда группу composable'ов имеет смысл объединить в один семантический узел (например, чтобы TalkBack читал целую строку списка, а не каждый элемент по отдельности). Объединение задаётся свойством `mergeDescendants` у узла семантики. Объединённое дерево выполняет слияние согласно `mergeDescendants`; необъединённое хранит узлы раздельно. Инструменты выбирают, какое дерево использовать. Семантические свойства имеют политику слияния (merge policy). Например, для `contentDescription` при слиянии значения потомков добавляются в список. Ключи свойств задаются типобезопасно и требуют имя и политику слияния. Политика по умолчанию — не сливать, сохранять значение родителя при наличии. Подробности API — в документации по семантике.

5. Система state snapshot

У Jetpack Compose свой способ представлять состояние и распространять его изменения — система state snapshot, которая и даёт реактивность. Компоненты могут перезапускаться автоматически по своим входам и только когда нужно, без ручного уведомления об изменениях (как в старой системе View на Android).

Начнём с термина «snapshot state».

Что такое snapshot state

Snapshot state — это **изолированное состояние, которое можно запомнить и наблюдать на предмет изменений**. Оно получается при вызове функций вроде `mutableStateOf`, `mutableStateListOf`, `mutableStateMapOf`, `derivedStateOf`, `produceState`, `collectAsState` и т.п. Все они возвращают какой-то тип `State`; разработчики называют это snapshot state.

Название связано с системой state snapshot в Jetpack Compose runtime: она задаёт модель и координацию изменений состояния и их распространения.

Реализация вынесена в отдельный слой, так что теоретически ею могут пользоваться и другие библиотеки с наблюдаемым состоянием.

В главе 2 мы видели, что компилятор Compose оборачивает объявления и выражения Composable так, чтобы **автоматически отслеживать чтения snapshot state в их теле**. Так snapshot state наблюдается. Цель — при любом изменении прочитанного Composable состояния инвалидировать его `RecomposeScope` и выполнить его снова при следующей recomposition.

Это инфраструктурный код Compose; клиентскому коду он не нужен. Клиенты вроде Compose UI могут не знать, как делаются инвалидация, распространение состояния и запуск recomposition, и только поставлять строительные блоки — Composable-функции.

Snapshot state нужен не только для уведомления об изменениях и recomposition. Слово «snapshot» в названии важно из-за **изоляции состояния** в контексте параллелизма.

Мутабельное состояние между потоками легко превращается в хаос: нужна жёсткая координация и синхронизация, иначе возможны коллизии, трудноуловимые баги и гонки. Языки решают это по-разному: неизменяемость данных, акторная модель с **изоляцией состояния** между потоками (у каждого актора своя копия, обмен через сообщения). Система snapshot в Compose не акторная, но к этому подходу близка.

Jetpack Compose опирается на мутабельное состояние, чтобы Composable автоматически реагировали на обновления. Только неизменяемое состояние не дало бы такой модели. Значит, нужно решать задачу общего состояния при параллельном выполнении (composition может идти в нескольких потоках, глава 1). Решение Compose — система state snapshot: изоляция состояния и последующее распространение изменений для **безопасной работы с мутабельным состоянием между потоками**.

Система snapshot state смоделирована как система управления параллелизмом: нужно **согласовывать состояние между потоками**. Общее мутабельное состояние в параллельной среде — сложная и универсальная задача. Ниже мы разберём такие системы и их использование в Compose.

Полезно взглянуть на интерфейс `State`, который реализует любой snapshot state:

```
1 @Stable
2 interface State&lt;out T&gt; {
3   val value: T
```

```
4 }
```

SnapshotState.kt

Контракт помечен `@Stable`: Compose по замыслу даёт и использует только стабильные реализации. Напоминание: любая реализация **должна** обеспечивать согласованность `equals`, уведомление composition при изменении публичного свойства (`value`) и стабильность типов публичных свойств. В следующих разделах мы увидим, как при каждой записи (модификации) snapshot state object composition уведомляется.

Рекомендую пост Зака Клиппа с введением в эти идеи.

Системы управления параллелизмом

Система state snapshot реализована как система управления параллелизмом — начнём с этого понятия.

В информатике «управление параллелизмом» — обеспечение корректного результата при параллельных операциях, то есть координация и синхронизация. Оно задаётся набором правил для корректности системы в целом. Координация имеет цену и часто бьёт по производительности, поэтому важно проектировать подход по возможности эффективно.

Пример — транзакции в СУБД. Там управление параллелизмом гарантирует безопасное выполнение транзакций при параллельной работе без нарушения целостности. «Безопасность» включает атомарность, возможность отката, отсутствие потери зафиксированных эффектов и сохранение эффектов отменённых транзакций. В языках программирования похожие идеи используются, например, для транзакционной памяти — как раз случай системы state snapshot. Транзакционная память упрощает параллельное программирование, позволяя выполнять группу операций загрузки и сохранения атомарно. В Compose запись состояния применяется **одной атомарной операцией** при распространении изменений из snapshot в другие. Такая группировка упрощает согласование параллельных чтений и записей; атомарные изменения проще отменять, откатывать или воспроизводить.

Категории систем управления параллелизмом:

- **Оптимистичные:** не блокируют чтения и записи, предполагают безопасность, при нарушении правил при коммите отменяют

транзакцию. Отменённая транзакция перезапускается (оверхед). Уместно, когда доля отменённых транзакций невелика.
- **Пессимистичные:** блокируют операцию транзакции при нарушении правил, пока возможность нарушения не исчезнет.
- **Полуоптимистичные:** гибрид — блокировка только в части случаев, в остальных оптимистичный подход с отменой при коммите.

Производительность зависит от пропускной способности, уровня параллелизма, риска взаимоблокировок. Неоптимистичные варианты сильнее склонны к дедлокам; часто решают отменой зависшей транзакции и перезапуском.

Jetpack Compose **оптимистичный**. Коллизии обновлений состояния обнаруживаются при распространении изменений (в конце), затем делается попытка автоматического слияния или отбрасывания (отмена изменений). Подробнее ниже.

Подход Compose проще, чем в СУБД: цель — только корректность. Свойства вроде восстанавливаемости, устойчивости, распределённости или репликации у Compose snapshot state нет (нет «D» в ACID). Snapshots Compose — только in-memory, в рамках процесса. Они атомарны, согласованы и изолированы.

Наряду с категориями (оптимистичная, пессимистичная, полуоптимистичная) есть типы; один из них — **многоверсионное управление параллелизмом (MVCC)**. Его использует Jetpack Compose. Система повышает параллелизм и производительность за счёт **создания новой версии объекта при каждой записи и возможности читать несколько последних релевантных версий**.

Многоверсионное управление параллелизмом (MVCC)

Глобальное состояние Compose общее для всех Composition, то есть для **потоков**. Composable могут выполняться параллельно (дверь для параллельной recomposition открыта). При параллельном выполнении они могут читать и изменять snapshot state одновременно — нужна изоляция состояния.

Одно из ключевых свойств управления параллелизмом — **изоляция**: корректность при параллельном доступе к данным. Простейший способ — блокировать всех читателей до завершения писателей, но это плохо для производительности. MVCC (и Compose) делают иначе.

Для изоляции MVCC хранит **несколько копий** данных (snapshot), и каждый поток работает со своей изолированной копией состояния в данный момент —

разными **версиями** состояния («многоверсионность»). Изменения потока не видны другим до завершения и распространения локальных изменений.

В теории управления параллелизмом это называется «snapshot isolation» — уровень изоляции, определяющий, какую версию видит каждая «транзакция» (snapshot).

MVCC опирается на неизменяемость: при записи создаётся новая копия данных. В памяти оказывается **несколько версий одних и тех же данных** — история изменений. В Compose это «state records»; мы к ним вернёмся.

Ещё одна особенность MVCC — **согласованные на момент времени представления** состояния (как у резервных копий). В MVCC это часто обеспечивается идентификатором транзакции. В Jetpack Compose так и есть: **каждому snapshot присваивается свой ID**. ID монотонно возрастают, snapshot упорядочены. Разные ID дают изоляцию чтений и записей без блокировок.

Рекомендую почитать про Concurrency control и Multiversion concurrency control.

Snapshot

Snapshot можно взять в любой момент. Он отражает текущее состояние программы (все snapshot state объекты) в этот момент. Можно брать несколько snapshot; у каждого будет **своя изолированная копия состояния программы** — копия текущего состояния всех snapshot state объектов (реализующих `State`) в момент создания snapshot.

Так состояние безопасно менять: обновление в одном snapshot не затронет копии того же состояния в других. В многопоточном сценарии каждый поток может работать со своим snapshot и своей копией.

В Compose runtime класс `Snapshot` моделирует **текущее** состояние программы. Чтобы взять snapshot: `val snapshot = Snapshot.takeSnapshot()`. Будут зафиксированы текущие значения всех state объектов до вызова `snapshot.dispose()` — этим задаётся время жизни snapshot.

У snapshot есть жизненный цикл. После использования snapshot нужно освободить. Без вызова `snapshot.dispose()` утекут ресурсы и сохранённое состояние. Snapshot считается **активным** между созданием и освобождением.

При создании snapshot получает ID, чтобы его состояние можно было отличать от версий в других snapshot. Так состояние программы **версионируется**

(многоверсионное управление параллелизмом).

Пример из поста Зака Клиппа:

```
fun main() {
  val dog = Dog()
  dog.name.value = &amp;quot;Spot&amp;quot;
  val snapshot = Snapshot.takeSnapshot()
  dog.name.value = &amp;quot;Fido&amp;quot;

  println(dog.name.value)
  snapshot.enter { println(dog.name.value) }
  println(dog.name.value)
}

// Output:
Fido
Spot
Fido
```

SnapshotSample.kt

Функция `enter` («вход в snapshot») **выполняет лямбду в контексте snapshot** — snapshot становится источником истины для любого состояния: все чтения внутри лямбды получают значения из snapshot. Так Compose и другие библиотеки могут выполнять логику работы с состоянием в контексте заданного snapshot — локально в потоке, до возврата из `enter`. Другие потоки не затрагиваются.

В примере после обновления имя «Fido», но при чтении внутри `enter` возвращается «Spot» — **значение на момент создания snapshot**.

Внутри `enter` в зависимости от типа snapshot (только чтение или мутабельный) возможны и чтение, и запись. Мутабельные snapshot разберём позже.

Snapshot, созданный через `Snapshot.takeSnapshot()`, только для чтения. Записывать в его state объекты нельзя — будет исключение.

Compose предоставляет реализацию контракта `Snapshot`, позволяющую мутировать состояние: `MutableSnapshot`. Есть и другие реализации. Иерархия типов:

```
sealed class Snapshot(...) {
  class ReadonlySnapshot(...) : Snapshot() {...}
```

```
3   class NestedReadonlySnapshot(...) : Snapshot() {...}
4   open class MutableSnapshot(...) : Snapshot() {...}
5   class NestedMutableSnapshot(...) : MutableSnapshot() {...}
6   class GlobalSnapshot(...) : MutableSnapshot() {...}
7   class TransparentObserverMutableSnapshot(...) :
 ↪  MutableSnapshot() {...}
8 }
```

Snapshot.kt

Кратко по типам:

- **ReadonlySnapshot:** хранимые state объекты только читаются.
- **MutableSnapshot:** хранимые state объекты можно читать и менять.
- **NestedReadonlySnapshot** и **NestedMutableSnapshot:** дочерние snapshot, так как snapshot образуют дерево.
- **GlobalSnapshot:** мутабельный snapshot глобального (общего) состояния программы, корень дерева snapshot.
- **TransparentObserverMutableSnapshot:** не изолирует состояние, только уведомляет наблюдателей при чтении/записи. Все его state records помечаются невалидными. Его ID всегда совпадает с родительским; операции выглядят как выполненные в родителе.

Дерево snapshot

Snapshot образуют дерево. Есть вложенные только для чтения и мутабельные. Корень дерева — `GlobalSnapshot`. Вложенные snapshot можно освобождать независимо, **оставляя родителя активным**. Это часто используется при **subcomposition** (глава 2): subcomposition создаётся inline для независимой инвалидации. Примеры: элемент lazy list, `BoxWithConstraints`, `SubcomposeLayou` `VectorPainter`. При subcomposition создаётся вложенный snapshot для изоляции состояния; когда subcomposition исчезает, snapshot освобождается, родительская composition и snapshot остаются. Изменения вложенного snapshot распространяются в родителя.

У всех типов snapshot есть методы вида `takeNestedSnapshot()` / `takeNestedMutabl` Read-only дочерний snapshot можно создать от любого типа; мутабельный — только от мутабельного (или от глобального).

Snapshots и потоки

Snapshot стоит представлять как структуры вне привязки к потоку. Поток может иметь текущий snapshot, но snapshot **не привязаны к потоку**. Поток может входить и выходить из snapshot; дочерний snapshot может быть «введён» в другом потоке. Параллельная работа — один из сценариев. Несколько потоков могут работать со своими snapshot; при введении мутабельных snapshot мы увидим, как дочерние должны сообщать изменения родителю. Коллизии между потоками будут обнаруживаться и обрабатываться. Текущий snapshot потока: `Snapshot.current` (если есть — потоковый, иначе глобальный).

Наблюдение за чтениями и записями

Рантайм Compose умеет запускать recomposition при записи в наблюдаемое состояние. Разберём связь с системой state snapshot — начнём с наблюдения за чтениями.

При вызове `Snapshot.takeSnapshot()` возвращается `ReadonlySnapshot`. В него можно передать необязательный `readObserver`; он будет вызываться при каждом чтении state из snapshot **внутри вызова `enter`**:

```
1 // simple observer to track the total number of reads
2 val snapshot = Snapshot.takeSnapshot { reads++ }
3 // ...
4 snapshot.enter { /* some state reads */ }
5 // ...
```

ReadOnlySnapshot.kt

Пример — `snapshotFlow(block: () -> T): Flow<T>`: превращает `State<T>` в `Flow`. При коллекции выполняется блок и эмитируется результат; при мутации прочитанного State Flow эмитирует новое значение. Для этого нужно записывать все чтения state. Это делается через read-only snapshot с read observer, который сохраняет их в `Set`:

```
1 fun &lt;T&gt; snapshotFlow(block: () -&gt; T): Flow&lt;T&gt; {
2   // ...
3   snapshot.takeSnapshot { readSet.add(it) }
4   // ...
5   // Do something with the Set
6 }
```

SnapshotFlow.kt

У read-only snapshot уведомляется не только его read observer, но и read observer родителя — чтение во вложенном snapshot должно быть видно всем родителям.

Для наблюдения за **записями** можно передать `writeObserver` при создании **мутабельного** snapshot: `Snapshot.takeMutableSnapshot()`. Пример — `Recomposer`, отслеживающий чтения и записи в Composition для автоматического запуска recomposition:

```
1 private fun readObserverOf(composition: ControlledComposition):
  ↪ (Any) -&gt; Unit {
2   return { value -&gt; composition.recordReadOf(value) } //
  ↪ recording reads
3 }
4
5 private fun writeObserverOf(
6   composition: ControlledComposition,
7   modifiedValues: IdentityArraySet&lt;Any&gt;?
8 ): (Any) -&gt; Unit {
9   return { value -&gt;
10    composition.recordWriteOf(value) // recording writes
11    modifiedValues?.add(value)
12  }
13 }
14
15 private inline fun &lt;T&gt; composing(
16   composition: ControlledComposition,
17   modifiedValues: IdentityArraySet&lt;Any&gt;?,
18   block: () -&gt; T
19 ): T {
20   val snapshot = Snapshot.takeMutableSnapshot(
21     readObserverOf(composition),
22     writeObserverOf(composition, modifiedValues)
23   )
24   try {
25     return snapshot.enter(block)
26   } finally {
27     applyAndCheck(snapshot)
28   }
29 }
```

Recomposer.kt

Функция `composing` вызывается при первичной composition и при каждой recomposition. Используется `MutableSnapshot`; все чтения и записи в `block` отслеживаются Composition. Блок — по сути код composition/recomposition (выполнение Composable). При записи в snapshot state инвалидируются соответствующие `RecomposeScope`, прочитавшие этот state, и запускается recomposition. В конце `applyAndCheck(snapshot)` распространяет изменения в другие snapshot и глобальное состояние.

Сигнатуры наблюдателей:

```
1 readObserver: ((Any) -&gt; Unit)?
2 writeObserver: ((Any) -&gt; Unit)?
```

ReadAndWriteObservers.kt

Есть утилита `Snapshot.observe(readObserver, writeObserver, block)` для наблюдения в текущем потоке. Её использует, например, `derivedStateOf`. Там же используется `TransparentObserverMutableSnapshot` — создаётся родительский snapshot только для уведомления наблюдателей о чтениях.

MutableSnapshot

В мутабельном snapshot любой state объект имеет то же значение, что при создании snapshot, **если только он не изменён локально в этом snapshot**. Все изменения в `MutableSnapshot` **изолированы** от других snapshot. Распространение идёт снизу вверх: вложенный мутабельный snapshot сначала применяет свои изменения, затем они распространяются в родителя или в глобальный snapshot (если это корень). Вызов `NestedMutableSnapshot#apply` или `MutableSnapshot#apply`.

По kdoc рантайма: *Composition использует мутабельные snapshot, чтобы изменения в Composable временно изолировать от глобального состояния и применить их при применении composition. При неудаче MutableSnapshot.apply snapshot и изменения отбрасываются и планируется новый расчёт composition.*

При применении composition (через Applier) изменения мутабельных snapshot применяются и уведомляют родителя или глобальный snapshot. Жизненный цикл мутабельного snapshot завершается вызовом `apply` и/или `dispose`. Изменения при `apply` применяются **атомарно**. Если snapshot освобождён без применения, все отложенные изменения отбрасываются.

Пример `apply` в клиентском коде:

```
class Address {
  var streetname: MutableState&lt;String&gt; =
  ↪ mutableStateOf(&quot;&quot;)
}

fun main() {
  val address = Address()
  address.streetname.value = &quot;Some street&quot;

  val snapshot = Snapshot.takeMutableSnapshot()
  println(address.streetname.value)
  snapshot.enter {
    address.streetname.value = &quot;Another street&quot;
    println(address.streetname.value)
  }
  println(address.streetname.value)
  snapshot.apply()
  println(address.streetname.value)
}

// This prints the following:

// Some street
// Another street
// Some street
// Another street
```

ApplyMutableSnapshotSample.kt

Внутри `enter` видно «Another street»; снаружи до `apply` — снова «Some street». После `apply` — «Another street». Есть сокращение: `Snapshot.withMutableSnapsh { ... }` — гарантирует вызов `apply` в конце.

Идея та же, что у списка изменений в Composer (глава 3): записать/отложить, применить в нужном порядке. Наблюдатели применения: `Snapshot.registerApp`

GlobalSnapshot и вложенные snapshot

`GlobalSnapshot` — мутабельный snapshot глобального состояния. Он не вкладывается; он один и является корнем дерева snapshot. Его не

«применяют» — его **продвигают**: `Snapshot.advanceGlobalSnapshot()` создаёт новый глобальный snapshot, перенося в него валидное состояние предыдущего; apply observers уведомляются. `dispose()` у него не вызывают — «освобождение» тоже через продвижение.

На JVM глобальный snapshot создаётся при инициализации системы snapshot. Менеджер глобального snapshot запускается при создании Composer. Каждая composition (первичная и recomposition) создаёт свой вложенный мутабельный snapshot и регистрирует read/write observers. Subcomposition могут создавать свои вложенные snapshot. При создании Composition вызывается `GlobalSnapshotManager.ensureStarted()` — начинается наблюдение за записями в глобальное состояние и периодическая диспетчеризация уведомлений apply в контексте `AndroidUiDispatcher.Main`.

StateObject и StateRecord

При каждой записи создаётся новая версия (copy-on-write). В памяти может храниться несколько версий одного snapshot state объекта. Внутри такой объект моделируется как `StateObject`, каждая версия — `StateRecord`. Запись валидна для snapshot, если её ID меньше или равен ID snapshot и она не в множестве `invalid` и не помечена невалидной. Невалидными считаются записи, созданные после текущего snapshot, в snapshot, уже открытом при создании текущего, или в snapshot, освобождённом до применения.

Интерфейс в коде:

```
interface StateObject {
  val firstStateRecord: StateRecord

  fun prependStateRecord(value: StateRecord)

  fun mergeRecords(
    previous: StateRecord,
    current: StateRecord,
    applied: StateRecord
  ): StateRecord? = null
}
```

Snapshot.kt

`mutableStateOf(value, policy)` возвращает `SnapshotMutableState` — `StateObject` со связным списком записей. При чтении обходится

список в поисках **последней валидной** записи. `mergeRecords` нужна для автоматического слияния конфликтов при применении.

`StateRecord` привязан к ID snapshot, имеет `next`, методы `assign` и `create`. У `mutableStateOf` записи типа `StateStateRecord` (обёртка над `value: T`). У `mutableStateListOf` — `SnapshotStateList` с записями `StateListStateRecord` (используют `PersistentList`).

Чтение и запись состояния

При чтении обходится список записей StateObject в поисках последней валидной для текущего snapshot. Геттер `value` в `SnapshotMutableStateImpl` вызывает `next.readable(this).value` (итерация + уведомление read observers). Сеттер через `withCurrent` и `overwritable`: проверка эквивалентности по `SnapshotMutationPolicy`, при отличии — запись через writable record и уведомление write observers.

Удаление и переиспользование устаревших записей

Отслеживается минимальный открытый snapshot ID. Если запись валидна, но не видна в нём, её можно безопасно переиспользовать. Обычно у мутабельного state 1–2 записи. При применении snapshot затемнённая запись переиспользуется; при dispose до apply все записи помечаются невалидными и могут переиспользоваться сразу.

Распространение изменений

Закрытие snapshot — удаление его ID из множества открытых; тогда его записи становятся видимыми новым snapshot. Продвижение — закрытие и сразу создание нового snapshot с новым ID. Глобальный snapshot не применяют, а продвигают. При `snapshot.apply()` локальные изменения мутабельного snapshot распространяются в родителя или в глобальное состояние. Сначала проверяются коллизии и при возможности делается merge (оптимистично). Для каждого изменения сравнивается с текущим значением; при отличии с учётом merge создаётся новая запись и дописывается в список. При ошибке применяется тот же путь, что и при отсутствии локальных изменений (закрытие, продвижение глобального, уведомление apply observers). Для вложенных мутабельных snapshot изменения добавляются в множество изменённых родителя и ID вложенного удаляется из множества невалидных родителя.

Слияние конфликтов записи

При слиянии для каждого изменённого state берутся текущее значение в родителе/глобальном, предыдущее и значение после применения; слияние делегируется объекту состояния (политика слияния). Сейчас в рантайме ни одна политика не выполняет настоящее слияние — при коллизии выбрасывается исключение. Compose избегает коллизий за счёт уникальных ключей доступа к state (например, `remember` в Composable). `mutableStateOf` по умолчанию использует `StructuralEqualityPolicy` — глубокое сравнение, включая ключ объекта. Можно задать свой `SnapshotMutationPolicy`. Пример из документации — политика-счётчик для `MutableState<Int>`: `equivalent(a,b) = (a==b)`, `merge(previous, current, applied) = current + (applied - previous)`. Тогда две параллельные «транзакции», добавляющие 10 и 20, дадут в итоге 30.

```
1 fun counterPolicy(): SnapshotMutationPolicy&lt;Int&gt; = object :
 ↪  SnapshotMutationPolicy&lt;I\
2 nt&gt; {
3   override fun equivalent(a: Int, b: Int): Boolean = a == b
4   override fun merge(previous: Int, current: Int, applied: Int) =
5     current + (applied - previous)
6 }
```

CounterPolicy.kt

```
 1 val state = mutableStateOf(0, counterPolicy())
 2 val snapshot1 = Snapshot.takeMutableSnapshot()
 3 val snapshot2 = Snapshot.takeMutableSnapshot()
 4 try {
 5   snapshot1.enter { state.value += 10 }
 6   snapshot2.enter { state.value += 20 }
 7   snapshot1.apply().check()
 8   snapshot2.apply().check()
 9 } finally {
10   snapshot1.dispose()
11   snapshot2.dispose()
12 }
13
14 // State is now 30 as the changes made in the snapshots are
 ↪  added together.
```

CounterPolicy2.kt

Можно задавать политики, допускающие коллизии и разрешающие их через `merge`.

6. Эффекты и обработчики эффектов

Перед разбором обработчиков эффектов полезно вспомнить, что считается побочным эффектом — так будет яснее, почему важно держать их под контролем в composable-деревьях.

Побочные эффекты

В главе 1 мы говорили о свойствах Composable-функций: побочные эффекты делают функции недетерминированными и мешают рассуждать о коде.

По сути побочный эффект — всё, что выходит из-под контроля и области видимости функции. Чистая функция «сложить два числа»:

```
1 fun add(a: Int, b: Int) = a + b
```

Add.kt

Это «чистая» функция: только входы и результат, результат не меняется при тех же входах — функция **детерминирована**.

Теперь добавим побочные действия:

```
1 fun add(a: Int, b: Int) =
2   calculationsCache.get(a, b) ?:
3   (a + b).also { calculationsCache.store(a, b, it) }
4 }
```

AddWithSideEffect.kt

Вводим кэш вычислений; он выходит из-под контроля функции. Если кэш обновляется из другого потока:

```
1 fun main() {
2   add(1, 2) // 3
3   // Другой поток: cache.store(1, 2, res = 4)
4   add(1, 2) // 4
5 }
```

AddWithSideEffect2.kt

Функция перестаёт быть детерминированной. Итого: побочные эффекты усложняют рассуждения и тестирование. Примеры: глобальные переменные, кэш, БД, сеть, вывод, файлы.

Побочные эффекты в Compose

При выполнении побочных эффектов внутри Composable мы попадаем в те же проблемы: эффект выходит из-под контроля жизненного цикла Composable. Любой Composable может перезапускаться многократно, поэтому запускать эффекты прямо в теле Composable рискованно (глава 1: Composable перезапускаемые).

Пример: Composable, загружающий данные из сети в теле:

```
@Composable
fun EventsFeed(networkService: EventsNetworkService) {
  val events = networkService.loadAllEvents() // side effect

  LazyColumn {
    items(events) { event -&gt;
      Text(text = event.name)
    }
  }
}
```

SideEffect.kt

Эффект будет выполняться при каждой recomposition, возможны множественные параллельные запросы без координации. Нужно выполнять эффект один раз при первой composition и хранить состояние на весь жизненный цикл Composable.

Другой пример — обновление внешнего состояния (включение/выключение TouchHandler в зависимости от состояния drawer):

```
@Composable
fun MyScreen(drawerTouchHandler: TouchHandler) {
  val drawerState = rememberDrawerState(DrawerValue.Closed)

  drawerTouchHandler.enabled = drawerState.isOpen

  // ...
}
```

SideEffect2.kt

Строка `drawerTouchHandler.enabled = drawerState.isOpen` — побочный эффект composition. Проблема: эффект выполняется при каждой composition/reco и **никогда не освобождается**, возможны утечки. Если Composable, запустивший сетевой запрос, выйдет из composition до завершения запроса, мы скорее всего захотим отменить задачу.

Jetpack Compose предлагает механизмы запуска побочных эффектов с учётом жизненного цикла: можно растянуть задачу на несколько recomposition или автоматически отменить при выходе Composable из composition. Эти механизмы называются **обработчиками эффектов (effect handlers)**.

Что нам нужно

Composition может **выполняться в разных потоках**, параллельно или в разном порядке. Нужны механизмы, чтобы:

- Эффекты запускались на нужном шаге жизненного цикла Composable.
- Приостанавливающиеся эффекты выполнялись в подходящем контексте (корутины, `CoroutineContext`).
- Эффекты с захваченными ссылками могли освобождать их при выходе из composition.
- Текущие приостановленные эффекты отменялись при выходе из composition.
- Эффекты, зависящие от меняющегося ключа, автоматически отменялись/пе при его изменении.

Всё это дают **обработчики эффектов** Jetpack Compose.

> Обработчики эффектов из поста доступны в `1.0.0-beta02` и позже.
> Публичный API Compose заморожен с момента входа в beta.

Обработчики эффектов

Composable «входит» в composition при материализации на экране и «выходит» при удалении из дерева. Между этими событиями могут выполняться эффекты; некоторые могут переживать жизненный цикл Composable (растягиваться на несколько composition).

Две категории:

- **Неприостанавливающиеся эффекты:** например, инициализация колбэка при входе и его освобождение при выходе.
- **Приостанавливающиеся эффекты:** например, загрузка данных из сети для UI.

Неприостанавливающиеся эффекты

DisposableEffect

Побочный эффект жизненного цикла composition.

- Для эффектов, **требующих освобождения**.
- Запускается при первом входе и при каждом изменении ключей.
- Обязателен колбэк **onDispose**. Освобождение — при выходе из composition и при каждой recomposition с изменившимися ключами (тогда эффект освобождается и перезапускается).

```kotlin
@Composable
fun backPressHandler(onBackPressed: () -&gt; Unit, enabled:
  ↪ Boolean = true) {
  val dispatcher =
  ↪ LocalOnBackPressedDispatcherOwner.current.onBackPressedDispatche

  val backCallback = remember {
    object : OnBackPressedCallback(enabled) {
      override fun handleOnBackPressed() {
        onBackPressed()
      }
    }
  }

   DisposableEffect(dispatcher) { // dispose/relaunch if
  ↪ dispatcher changes
    dispatcher.addCallback(backCallback)
    onDispose {
      backCallback.remove() // avoid leaks!
    }
  }
}
```

DisposableEffect.kt

Колбэк прикрепляется к диспетчеру из `CompositionLocal`. Чтобы эффект перезапускался при смене диспетчера, **передаём диспетчер как ключ**. При выходе Composable колбэк освобождается. Чтобы запустить эффект один раз при входе и освободить при выходе, можно **передать константу в качестве ключа**: `DisposableEffect(true)` или `DisposableEffect(Unit)`. У `DisposableEffect` всегда должен быть хотя бы один ключ.

SideEffect

Ещё один побочный эффект composition. Особенность: «выполнить при этой composition или забыть» — при падении composition эффект **отбрасывается**. Он **не хранится в slot table**, не переживает composition и не перезапускается при следующих.

- Для эффектов **без необходимости освобождения**.
- Выполняется после каждой composition/recomposition.
- Удобен для **публикации обновлений во внешнее состояние**.

```
@Composable
fun MyScreen(drawerTouchHandler: TouchHandler) {
  val drawerState = rememberDrawerState(DrawerValue.Closed)

  SideEffect {
    drawerTouchHandler.enabled = drawerState.isOpen
  }

  // ...
}
```

SideEffect.kt

Так мы синхронизируем внешнее состояние (TouchHandler) с текущим состоянием drawer при каждой composition. `SideEffect` — для **публикации обновлений** во внешнее состояние, не управляемое системой Compose State.

currentRecomposeScope

Скорее эффект, чем обработчик, но полезно упомянуть.

В Android вы могли сталкиваться с аналогом `invalidate` в системе View — он запускает новый проход measure/layout/draw. Раньше так часто делали покадровую анимацию на Canvas: на каждом кадре вызывали invalidate и рисовали с учётом прошедшего времени.

`currentRecomposeScope` — интерфейс с одной целью:

```
interface RecomposeScope {
    /**
     * Invalidate the corresponding scope, requesting the composer recompose this scope.
     */
    fun invalidate()
}
```

RecomposeScope.kt

Вызов `currentRecomposeScope.invalidate()` инвалидирует composition локально и **запускает recomposition**. Полезно, когда источник истины **не snapshot State** Compose.

```
interface Presenter {
  fun loadUser(after: @Composable () -&gt; Unit): User
}

@Composable
fun MyComposable(presenter: Presenter) {
  val user = presenter.loadUser { currentRecomposeScope.invalidate() } // not a State!

  Text(&quot;The loaded user: ${user.name}&quot;)
}
```

MyComposable.kt

Здесь мы вручную инвалидируем при появлении результата, так как не используем `State`. Это крайний случай; в большинстве ситуаций лучше опираться на `State` и умную recomposition. Итого: □ использовать редко! □ Опирайтесь на `State` для умной recomposition при изменении данных — так вы максимально используете возможности Compose runtime.

> Для покадровой анимации в Compose есть API приостановки до следующего кадра Choreographer и обновления state по прошедшему времени — см. официальную документацию по анимации.

Приостанавливающиеся эффекты

rememberCoroutineScope

Создаёт `CoroutineScope`, привязанный к жизненному циклу composition.

- Для **приостанавливающихся эффектов, привязанных к жизненному циклу composition**.
- Scope **отменяется при выходе из composition**.
- Один и тот же scope возвращается при recomposition — можно продолжать запускать задачи; все они отменятся при выходе.
- Удобен для запуска задач **в ответ на действия пользователя**.
- Выполняется на диспетчере Applier при входе (обычно `AndroidUiDispatche`

```
@Composable
fun SearchScreen() {
  val scope = rememberCoroutineScope()
  var currentJob by remember { mutableStateOf&lt;Job?&gt;(null)
  ↪ }
  var items by remember {
  ↪ mutableStateOf&lt;List&lt;Item&gt;&gt;(emptyList()) }

  Column {
    Row {
      TextField(&quot;Start typing to search&quot;,
        onValueChange = { text -&gt;
          currentJob?.cancel()
          currentJob = scope.async {
            delay(threshold)
            items = viewModel.search(query = text)
          }
        }
      )
    }
    Row { ItemsVerticalList(items) }
  }
}
```

rememberCoroutineScope.kt

Каждое изменение ввода отменяет предыдущую задачу и запускает новую с задержкой — троттлинг на стороне UI.

> Отличие от `LaunchedEffect`: `LaunchedEffect` используют для задач, инициированных composition, а `rememberCoroutineScope` — для задач **по действию пользователя**.

LaunchedEffect

Приостанавливающийся вариант для загрузки начального состояния при входе в composition.

- Запускается при входе в composition.
- Отменяется при выходе.
- Отменяется и перезапускается при изменении ключа/ключей.
- Удобен, чтобы **растянуть задачу на несколько recomposition**.
- Выполняется на диспетчере Applier при входе (обычно `AndroidUiDispatche`
  Требуется хотя бы один ключ.

Эффект запускается один раз при входе и затем при каждом изменении ключа. Не забывайте: при перезапуске эффект каждый раз отменяется.

```
1 @Composable
2 fun SpeakerList(eventId: String) {
3   var speakers by remember {
 ↪  mutableStateOf&lt;List&lt;Speaker&gt;&gt;(emptyList()) }
4   LaunchedEffect(eventId) { // cancelled / relaunched when
 ↪  eventId varies
5     speakers = viewModel.loadSpeakers(eventId) // suspended
 ↪  effect
6   }
7
8   ItemsVerticalList(speakers)
9 }
```

LaunchedEffect.kt

produceState

Удобная обёртка над `LaunchedEffect` для случая, когда эффект в итоге заполняет `State`.

- Можно задать начальное значение и один или несколько ключей.
- Если ключ не передать, внутри вызывается `LaunchedEffect(Unit)` — эффект **растягивается на все composition**. API это явно не отражает.

```
1 @Composable
2 fun SearchScreen(eventId: String) {
3   val uiState = produceState(initialValue =
 ↪  emptyList&lt;Speaker&gt;(), eventId) {
4     viewModel.loadSpeakers(eventId) // suspended effect
5   }
6
7   ItemsVerticalList(uiState.value)
8 }
```

produceState.kt

Адаптеры для сторонних библиотек

Часто нужно потреблять типы вроде `Observable`, `Flow`, `LiveData`. Compose даёт адаптеры; подключается соответствующая зависимость:

```
1 implementation
 ↪  &quot;androidx.compose.runtime:runtime:$compose_version&quot;
 ↪  // includes Flow \
2 adapter
3 implementation &quot;androidx.compose.runtime:runtime-
 ↪  livedata:$compose_version&quot;
4 implementation &quot;androidx.compose.runtime:runtime-
 ↪  rxjava2:$compose_version&quot;
```

Dependencies.kt

Все адаптеры в итоге опираются на обработчики эффектов: подписываются через API библиотеки и отображают каждый элемент в свой `MutableState`, экспонируемый как неизменяемый `State`.

Примеры: `LiveData.observeAsState()` (реализация через `DisposableEffect`), RxJava2 `subscribeAsState()` (аналогично), Kotlin Flow `collectAsState()` (через `produceState`/`LaunchedEffect`, так как Flow нужно собирать из приостанавливающегося контекста). Примеры для разных библиотек:

LiveData

```
1 class MyComposableVM : ViewModel() {
2   private val _user = MutableLiveData(User(&quot;John&quot;))
```

```
  val user: LiveData&lt;User&gt; = _user
  //...
}

@Composable
fun MyComposable() {
  val viewModel = viewModel&lt;MyComposableVM&gt;()

  val user by viewModel.user.observeAsState()

  Text(&quot;Username: ${user?.name}&quot;)
}
```

LiveData.kt

Реализация `observeAsState` опирается на обработчик `DisposableEffect`.

RxJava2

```
class MyComposableVM : ViewModel() {
  val user: Observable&lt;ViewState&gt; =
  ↪ Observable.just(ViewState.Loading)
  //...
}

@Composable
fun MyComposable() {
  val viewModel = viewModel&lt;MyComposableVM&gt;()

  val uiState by
  ↪ viewModel.user.subscribeAsState(ViewState.Loading)

  when (uiState) {
    ViewState.Loading -&gt; TODO(&quot;Show loading&quot;)
    ViewState.Error -&gt; TODO(&quot;Show Snackbar&quot;)
    is ViewState.Content -&gt; TODO(&quot;Show content&quot;)
  }
}
```

RxJava2.kt

Реализация `subscribeAsState()` устроена аналогично. То же расширение доступно для `Flowable`.

KotlinX Coroutines Flow

```
class MyComposableVM : ViewModel() {
  val user: Flow&lt;ViewState&gt; = flowOf(ViewState.Loading)
  //...
}

@Composable
fun MyComposable() {
  val viewModel = viewModel&lt;MyComposableVM&gt;()

  val uiState by
 viewModel.user.collectAsState(ViewState.Loading)

  when (uiState) {
    ViewState.Loading -&gt; TODO(&quot;Show loading&quot;)
    ViewState.Error -&gt; TODO(&quot;Show Snackbar&quot;)
    is ViewState.Content -&gt; TODO(&quot;Show content&quot;)
  }
}
```

Flow.kt

Реализация `collectAsState` устроена иначе: `Flow` нужно собирать из приостанавливающегося контекста, поэтому используется `produceState`, который опирается на `LaunchedEffect`.

Все эти адаптеры опираются на обработчики эффектов из этой главы; по тому же паттерну можно написать свой адаптер для своей библиотеки.

7. Продвинутые сценарии использования Compose Runtime

До сих пор книга говорила о Compose в контексте Android, с которого приходит большинство разработчиков. Применения Compose, однако, выходят далеко за рамки Android и пользовательских интерфейсов. В этой главе разобраны такие продвинутые сценарии с практическими примерами.

Compose Runtime и Compose UI

Прежде чем углубляться во внутренности, важно разделить Compose UI и Compose runtime. **Compose UI** — новый UI-тулкит для Android с деревом `LayoutNode`, которые затем рисуют содержимое на канвасе. **Compose runtime** даёт базовую механику и примитивы, связанные с состоянием и composition.

Компилятор Compose теперь поддерживает весь спектр платформ Kotlin, поэтому runtime можно использовать для управления UI или любыми другими древовидными иерархиями почти везде (где запускается Kotlin). Обратите внимание на «другие древовидные иерархии»: в runtime почти ничего не привязано напрямую к UI (или Android). Runtime создавался и оптимизировался под UI, но остаётся достаточно универсальным для построения деревьев любого рода. В этом он близок к React JS: изначально — UI в вебе, но применяется и в синтезаторах, 3D-рендерерах. Большинство кастомных рендереров переиспользуют ядро React, подставляя свои строительные блоки вместо DOM браузера.

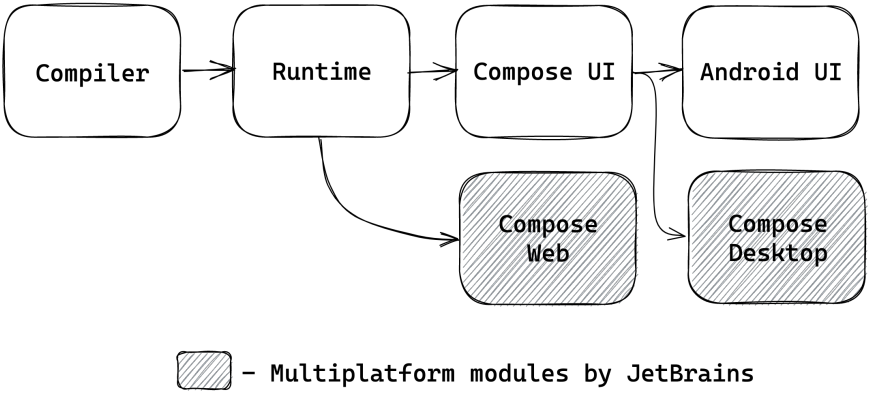
Разработчики Compose вдохновлялись React. Даже ранние прототипы — XML прямо в Kotlin — напоминают подход «HTML в JS» из React. Многие сторонние решения на React можно повторить на Compose и запускать в Kotlin multiplatform.

```
class ContactsView: Component() {

   var contacts: List<Contact>

   override fun compose() {
      <LinearLayout orientation="vertical">
         for (contact in contacts) {
            <LinearLayout>
               <ImageView image={contact.photo} />
               <TextView text={contact.name} />
            </LinearLayout>
         }
      </LinearLayout>
   }
}
```

Ранний прототип Compose

Ещё до выхода Android-версии Compose из беты JetBrains начала использовать Compose для Kotlin multiplatform: на момент написания ведутся версии для JVM (десктоп) и для JS (браузер), в планах — iOS. Все эти варианты переиспользуют разные части Compose:

- Compose for Desktop максимально близок к Android, переиспользуя слой отрисовки Compose UI благодаря портированным обёрткам Skia. Система событий расширена под мышь и клавиатуру.
- Compose for iOS (в разработке) тоже использует Skia и большую часть портируемой с JVM логики.
- Compose for Web опирается на DOM браузера для отображения, переиспользуя только компилятор и runtime. Компоненты строятся поверх HTML/CSS — система сильно отличается от Compose UI, но runtime и компилятор используются почти так же. С приближением поддержки Kotlin WASM набирает обороты и Skia-вариант для веба.



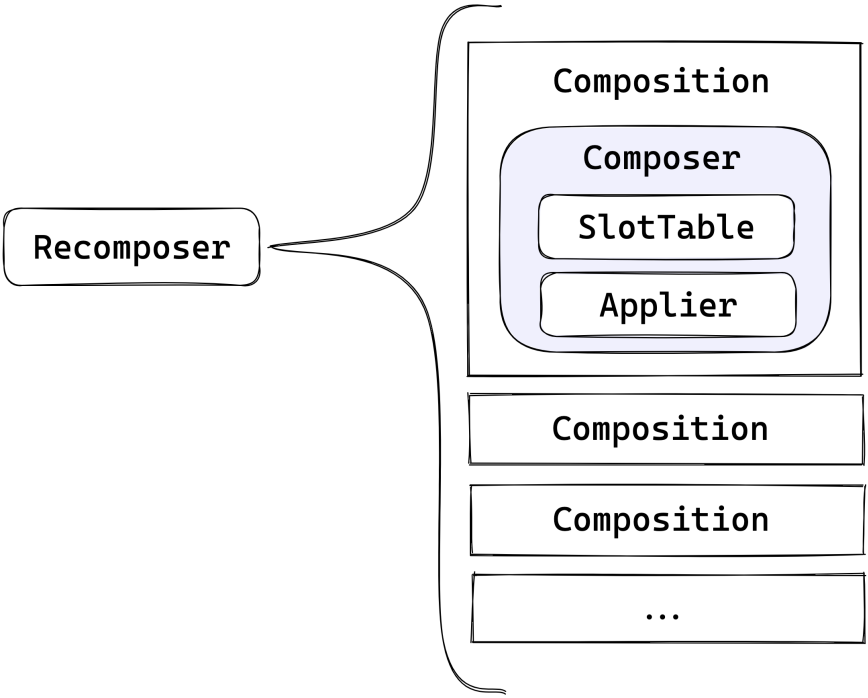
Модульная структура Compose в multiplatform

И снова к коду!

(Повторное) знакомство с Composition

`Composition` задаёт контекст для всех composable-функций: «кэш» на базе `SlotTable` и интерфейс создания кастомных деревьев через `Applier`. `Recomposer` управляет `Composition` и запускает recomposition при изменении чего-то релевантного (например, state). Как указано в документации,

`Composition` обычно создаётся фреймворком, но в этой главе мы будем создавать её сами.



Структура Composition

Фабричный метод:

```
1 fun Composition(
2     applier: Applier&lt;*&gt;,
3     parent: CompositionContext
4 ): Composition = ...
```

Composition.kt

- Родительский `context` доступен внутри любой composable через `rememberCompositionContext()`. Также `Recomposer` реализует `CompositionContext` и доступен на Android или создаётся отдельно под свои нужды.
- Второй параметр — `Applier`, определяющий, как создавать и связывать

дерево, производимое `Composition`. В предыдущих главах он уже разбирался; ниже будут примеры реализации.

Интересный факт: можно передать `Applier`, который ничего не делает, если нужны только свойства composable-функций (преобразования потоков данных, обработчики событий по изменению state) — см. Molecule от Cash App. Достаточно `Applier<Nothing>` и не использовать `ComposeNode`!

Дальше глава посвящена использованию **Compose runtime** без **Compose UI**: сначала пример из библиотеки Compose UI — кастомное дерево для векторной графики (кратко упоминалось ранее). Затем переключимся на Kotlin/JS и сделаем игрушечную библиотеку управления DOM браузера на Compose.

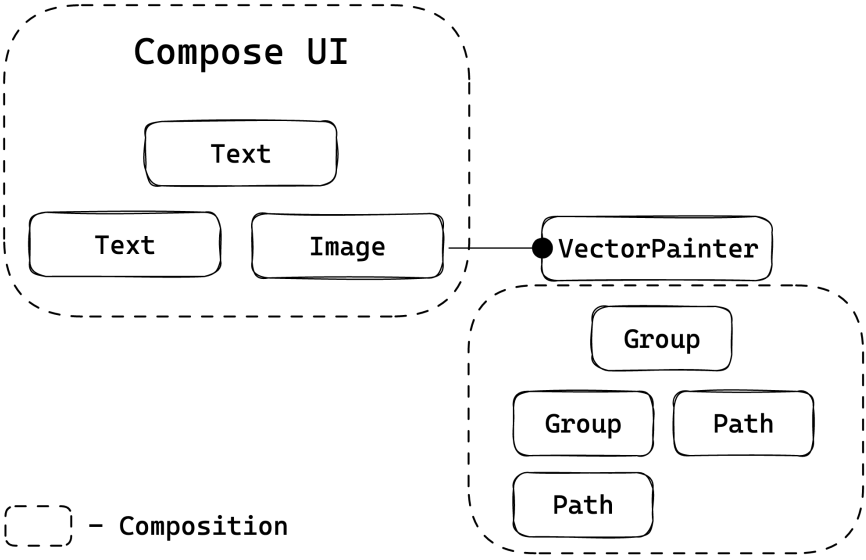
Composition векторной графики

Векторная отрисовка в Compose реализована через абстракцию `Painter`, по смыслу близкую к `Drawable` в классическом Android:

```
Image(
  painter = rememberVectorPainter { width, height -&gt;
    Group(
      scaleX = 0.75f,
      scaleY = 0.75f
    ) {
        val pathData = PathData { ... }
        Path(pathData = pathData)
    }
  }
)
```

VectorExample.kt

Функции внутри блока `rememberVectorPainter` (`Group`, `Path`) — тоже composable, но другого рода: они создают не `LayoutNode`, а элементы, специфичные для вектора. Из них собирается дерево вектора, которое затем рисуется на канвасе.



Compose UI и composition в VectorPainter.

`Group` и `Path` живут в отдельной **composition** от остального UI. Эта composition принадлежит `VectorPainter` и допускает только элементы описания векторного изображения; обычные UI-composable внутри блока запрещены.

Проверка выполняется в runtime, поэтому компилятор не помешает использовать внутри блока `Image` или `Box` — писать таких painter'ов пока потенциально небезопасно; в будущем планируют улучшить проверки на этапе компиляции.

Правила про state, эффекты и всё, что касается **runtime**, из предыдущих глав переносятся и на векторную composition (например, transition API для анимации изменений вектора вместе с UI). Подробнее: VectorGraphicsDemo.kt и AnimatedVectorGraphicsDemo.kt.

Построение дерева векторного изображения

Вектор строится из элементов проще, чем `LayoutNode`:

```
1 sealed class VNode {
2   abstract fun DrawScope.draw()
```

```
}

// the root node
internal class VectorComponent : VNode() {
  val root = GroupComponent()

  override fun DrawScope.draw() {
    // set up viewport size and cache drawing
  }
}

internal class PathComponent : VNode() {
  var pathData: List&lt;PathNode&gt;
  // more properties

  override fun DrawScope.draw() {
    // draw path
  }
}

internal class GroupComponent : VNode() {
  private val children = mutableListOf&lt;VNode&gt;()
  // more properties

  override fun DrawScope.draw() {
    // draw children with transform
  }
}
```

VNode.kt

Узлы задают структуру, похожую на классические vector drawable в XML: **GroupComponent** объединяет дочерние узлы и применяет к ним общее преобразование; **PathComponent** — лист (без детей), рисует `pathData`. `fun DrawScope.draw()` используется для отрисовки содержимого узла и детей; сигнатура совпадает с интерфейсом `Painter`, с которым позже интегрируется корень дерева.

Тот же `VectorPainter` используется для XML-ресурсов vector drawable из классического Android: парсер строит аналогичную структуру и превращает её в цепочку вызовов `@Composable`.

Узлы объявлены как internal; создаются только через соответствующие `@Composable`:

```
@Composable
fun Group(
    scaleX: Float = DefaultScaleX,
    scaleY: Float = DefaultScaleY,
    ...
    content: @Composable () -&gt; Unit
) {
    ComposeNode&lt;GroupComponent, VectorApplier&gt;(
        factory = { GroupComponent() },
        update = {
            set(scaleX) { this.scaleX = it }
            set(scaleY) { this.scaleY = it }
            ...
        },
        content = content
    )
}

@Composable
fun Path(
    pathData: List&lt;PathNode&gt;,
    ...
) {
    ComposeNode&lt;PathComponent, VectorApplier&gt;(
        factory = { PathComponent() },
        update = {
            set(pathData) { this.pathData = it }
            ...
        }
    )
}
```

VectorComposables.kt

Вызовы `ComposeNode` вносят узлы в composition и строят дерево. Параметр **factory** задаёт создание узла; **update** — инкрементальное обновление свойств уже созданного экземпляра (внутри — мемоизация через `set`/`update`). **content** добавляет дочерние узлы к текущему; выполняется после обновления узла. У

`ComposeNode` есть перегрузка без `content` для листьев (например, `Path`).

Для связи узлов используется `Applier`. Узлы `VNode` объединяются через `VectorApplier`:

```
class VectorApplier(root: VNode) : AbstractApplier&lt;VNode&gt;(root) {
  override fun insertTopDown(index: Int, instance: VNode) {
    current.asGroup().insertAt(index, instance)
  }

  override fun insertBottomUp(index: Int, instance: VNode) {
    // Ignored as the tree is built top-down.
  }

  override fun remove(index: Int, count: Int) {
     current.asGroup().remove(index, count)
   }

   override fun move(from: Int, to: Int, count: Int) {
     current.asGroup().move(from, to, count)
   }

   override fun onClear() {
     root.asGroup().let { it.remove(0, it.numChildren) }
   }

   // VectorApplier only works with [GroupComponent], as it cannot add
   // children to [PathComponent] by design
   private fun VNode.asGroup(): GroupComponent {
     return when (this) {
       is GroupComponent -&gt; this
       else -&gt; error(&quot;Cannot only insert VNode into Group&quot;)
     }
   }
}
```

VectorApplier.kt

Методы `Applier` часто сводятся к операциям над списком (insert/move/remove).

`AbstractApplier` даёт удобные расширения для `MutableList`. В `VectorApplier` эти операции реализованы прямо в `GroupComponent`. У `Applier` два варианта вставки: **topDown** (сначала узел, потом дети) и **bottomUp** (сначала дети, потом узел в дерево) — выбор связан с производительностью на разных платформах. Подробнее: документация Applier.

Интеграция векторной composition в Compose UI

Остаётся интегрировать векторную composition с `Painter`:

```
class VectorPainter internal constructor() : Painter() {
  ...

  // 1. Called in the context of UI composition
  @Composable
  internal fun RenderVector(
    content: @Composable (...) -&gt; Unit
  ) {
    // 2. The parent context is captured with
  [rememberCompositionContext]
    // to propagate its values, e.g. CompositionLocals.
    val composition = composeVector(
      rememberCompositionContext(),
      content
    )

    // 3. Whenever the UI &quot;forgets&quot; the VectorPainter,

    // the vector composition is disposed with
  [DisposableEffect] below.
    DisposableEffect(composition) {
      onDispose {
        composition.dispose()
      }
    }
  }

  private fun composeVector(
    parent: CompositionContext,
    composable: @Composable (...) -&gt; Unit
  ): Composition {
```

```
    ...
    // See implementation below
  }
}
```

VectorPainter.kt

Связь composition UI и векторной: (1) `RenderVector` принимает composable-описание вектора. (2) Родительский контекст берётся из UI-composition через `rememberCompositionContext`, чтобы оба дерева были связаны с одним `Recomposer` и общими значениями (например, `CompositionLocal` для плотности). (3) Composition нужно освобождать, когда `RenderVector` выходит из области видимости — за это отвечает `DisposableEffect`.

Заполнение composition содержимым и отрисовка:

```
class VectorPainter : Painter() {
  // The root component for the vector tree
  private val vector = VectorComponent()
  // 1. Composition with vector elements.
  private var composition: Composition? = null

  @Composable
  internal fun RenderVector(
    content: @Composable (...) -&gt; Unit
  ) {
    ...
    // See full implementation above
  }

  private fun composeVector(
    parent: CompositionContext,
    composable: @Composable (...) -&gt; Unit
  ): Composition {
    // 2. Creates composition or reuses an existing one
    val composition =
      if (this.composition == null ||
this.composition.isDisposed) {
        Composition(
          VectorApplier(vector.root),
          parent
```

```
        )
      } else {
        this.composition
      }
      this.composition = composition

    // 3. Sets the vector content to the updated composable value
    composition.setContent {
     // Vector composables can be called inside this block only
      composable(vector.viewportWidth, vector.viewportHeight)
    }

    return composition
  }

  // Painter interface integration, is called every time the system
  // needs to draw the vector image on screen
  override fun DrawScope.onDraw() {
    with(vector) {
        draw()
    }
  }
}
```

VectorPainter.kt

(1) Painter хранит свою composition, так как `ComposeNode` требует совпадения applier с переданным в composition, а UI использует applier, несовместимый с векторными узлами. (2) Composition создаётся или переиспользуется при инициализации или выходе из области видимости. (3) Содержимое задаётся через `setContent`, как в `ComposeView`; при новом `content` в `RenderVector` вызывается снова `setContent`. Содержимое добавляет детей к корневому узлу, который затем используется для отрисовки в `Painter`.

На этом интеграция завершена: `VectorPainter` рисует `@Composable`-содержимое на экране, а composable внутри painter'а имеют доступ к state и composition locals из UI-composition.

Дальше — создание отдельной Compose-системы по тем же принципам… на Kotlin/JS.

Управление DOM с помощью Compose

Мультиплатформенная поддержка Compose ещё молода: вне JVM доступны в основном runtime и компилятор. Этого достаточно, чтобы создавать composition и что-то в ней запускать.

Компилятор Compose из зависимостей Google поддерживает все платформы Kotlin; runtime по умолчанию распространяется только для Android. JetBrains публикует свою (почти совместимую) версию Compose с мультиплатформенными артефактами в том числе для JS.

Первый шаг — определить дерево, с которым будет работать Compose. В браузере уже есть «видовая» система на HTML/CSS. Элементами можно управлять из JS через DOM (Document Object Model), API которого доступен и в стандартной библиотеке Kotlin/JS.

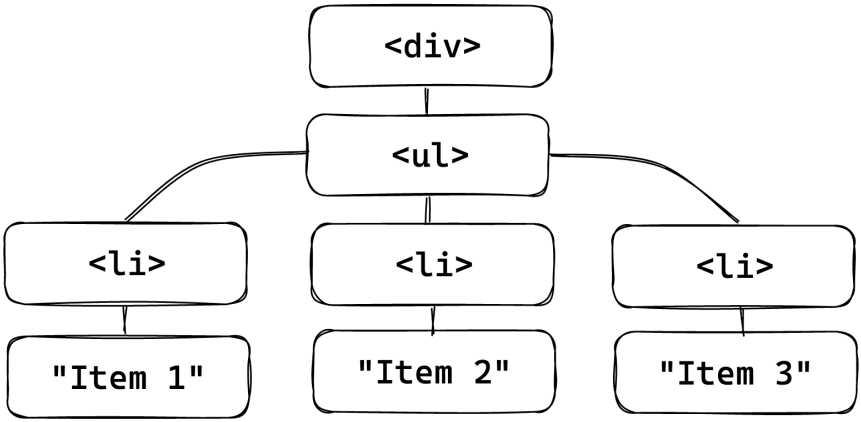
Пример HTML:

```
1 &lt;div&gt;
2   &lt;ul&gt;
3     &lt;li&gt;Item 1&lt;/li&gt;
4     &lt;li&gt;Item 2&lt;/li&gt;
5     &lt;li&gt;Item 3&lt;/li&gt;
6   &lt;/ul&gt;
7 &lt;/div&gt;
```

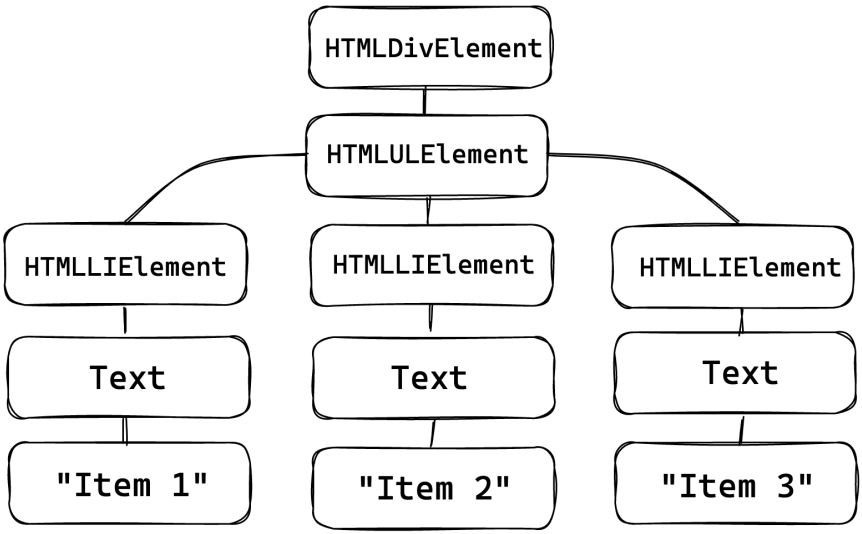
sample.html

В браузере это дерево. DOM — древовидная структура из элементов, в Kotlin/JS представленных как `org.w3c.dom.Node`:

- HTML-элементы (подтипы `org.w3c.dom.HTMLElement`) — теги (`li`, `div` и т.д.), создаются через `document.createElement(<tagName>)`.
- Текст между тегами — `org.w3c.dom.Text`, создаётся через `document.creat`



Представление HTML-дерева в браузере



Представление дерева для JS

Эти элементы станут основой дерева, управляемого Compose (аналогично `VNode` для вектора):

```
1 @Composable
2 fun Tag(tag: String, content: @Composable () -&gt; Unit) {
```

```
  ComposeNode&lt;HTMLElement, DomApplier&gt;(
    factory = { document.createElement(tag) as HTMLElement },
    update = {},
    content = content
  )
}

@Composable
fun Text(value: String) {
  ReusableComposeNode&lt;Text, DomApplier&gt;(
    factory = { document.createTextElement(&quot;&quot;) },
    update = {
      set(value) { this.data = it }
    }
  )
}
```

HtmlTags.kt

Теги нельзя менять «на месте» (например, `<audio>` и `<div>` — разные представления в браузере), поэтому при смене имени тега узел нужно пересоздавать. Compose этого не делает автоматически; в один и тот же composable не стоит передавать разные имена тегов. Проще всего обернуть каждый тип узла в свой composable (`Div`, `Ul` и т.д.) — получатся разные compile-time группы, и Compose будет заменять элементы целиком. Для `Text` узлы одинаковы по структуре, поэтому используется `ReusableComposeNode`: экземпляр переиспользуется между группами; текст задаётся в `update`.

Для сборки дерева нужен `Applier` для DOM-элементов. Логика похожа на `VectorApplier`, отличаются только методы добавления/удаления детей. Подробнее: DomApplier в Compose for Web.

Отдельная composition в браузере

Чтобы собирать наши composable в UI, нужна активная composition. В Compose UI всё уже настраивается в `ComposeView`; в браузере composition создаётся вручную. Те же принципы применимы и на других платформах — описанные компоненты есть в «общем» Kotlin-коде.

```
fun renderComposable(root: HTMLElement, content: @Composable ()
↪ -&gt; Unit) {
```

```
  GlobalSnapshotManager.ensureStarted()

  val recomposerContext = DefaultMonotonicFrameClock +
↪ Dispatchers.Main
  val recomposer = Recomposer(recomposerContext)

  val composition = ControlledComposition(
    applier = DomApplier(root),
    parent = recomposer
  )

  composition.setContent(content)

  CoroutineScope(recomposerContext).launch(start =
↪ UNDISPATCHED) {
    recomposer.runRecomposeAndApplyChanges()
  }
}
```

renderComposable.kt

`renderComposable` скрывает детали запуска composition и позволяет отрисовывать composable в DOM-элемент. Внутри: инициализация `Recomposer` с нужными clock и coroutine context:

- Сначала инициализируется система снимков (state). `GlobalSnapshotManage` намеренно не входит в runtime; при необходимости его можно скопировать из исходников Android.
- Контекст для `Recomposer` в JS: по умолчанию `MonotonicClock` на `requestAnimationFrame` (в реализации JetBrains), `Dispatchers.Main` — единственный поток JS. В этом контексте потом выполняются recomposition.
- Composition создаётся так же, как в векторном примере, но родителем выступает recomposer (у верхней composition родитель всегда recomposer).
- Содержимое задаётся через `setContent`. Все обновления должны происходить внутри переданного composable; повторный вызов `renderComposable` пересоздаёт всё с нуля.
- Запуск цикла recomposition — корутина с `Recomposer.runRecomposeAndApp` На Android этот процесс привязан к жизненному циклу activity/view, остановка — через `recomposer.cancel()`. Здесь жизненный цикл

привязан к странице, отмена не нужна.

Пример статического контента:

```
fun main() {
  renderComposable(document.body!!) {
    // equivalent of &lt;button&gt;Click me!&lt;/button&gt;
    Tag(&quot;button&quot;) {
      Text(&quot;Click me!&quot;)
    }
  }
}
```

HtmlSample1.kt

Статический контент можно сделать и проще; Compose здесь нужен для интерактивности. Обычно нужно реагировать на клик; в DOM это делается обработчиками клика, как у View в Android. В Compose UI многие обработчики задаются через расширения `Modifier`, но они завязаны на `LayoutNode`, поэтому для этой игрушечной веб-библиотеки недоступны. Поведение `Modifier` можно перенести и адаптировать под узлы — остаётся упражнением для читателя.

```
@Composable
fun Tag(
  tag: String,
  // this callback is invoked on click events
  onClick: () -&gt; Unit = {},
  content: @Composable () -&gt; Unit
) {
  ComposeNode&lt;HTMLElement, DomApplier&gt;(
    factory = { createTagElement(tag) },
    update = {
      // when listener changes, the listener on the DOM node is
↪ re-set
      set(onClick) {
        this.onclick = { _ -&gt; onClick() }
      }
    },
    content = content
  )
}
```

HtmlTags.kt

Тег может принимать колбэк клика и пробрасывать его в свойство `onclick` у `HTMLElement`. Счётчик по клику:

```
fun main() {
  renderComposable(document.body!!) {
    // Counter state is updated on click
    var counterState by remember { mutableStateOf(0) }

    Tag(&quot;h1&quot;) {
      Text(&quot;Counter value: $counterState&quot;)
    }

     Tag(&quot;button&quot;, onClick = { counterState++ }) {
       Text(&quot;Increment!&quot;)
     }
   }
}
```

HtmlSampleCounter.kt

Дальше библиотеку можно расширять: CSS, события, элементы. Команда JetBrains экспериментирует с более продвинутой версией Compose for Web на тех же принципах, но с большими возможностями. Можно попробовать техдемо в проектах на Kotlin/JS.

Заключение

В этой главе мы рассмотрели, как базовые концепции Compose применяются вне Compose UI. Кастомные composition встречаются реже, но это полезный инструмент, если вы уже работаете в экосистеме Kotlin/Compose.

Composition векторной графики — пример встраивания кастомного composable-дерева в Compose UI. По тем же принципам можно делать другие кастомные элементы с доступом к state, анимациям и composition locals из UI-composition.

Отдельные composition возможны на всех платформах Kotlin. Мы показали это на игрушечной библиотеке управления DOM в браузере на базе Compose runtime и Kotlin/JS. Аналогично Compose runtime уже используется для управления деревьями UI вне Android (например, Mosaic — CLI от Jake Wharton).

Рекомендуется экспериментировать с собственными идеями на Compose и делиться обратной связью с командой Compose в канале #compose в Kotlin Slack. Их основной фокус по-прежнему Compose UI, но им интересно узнавать о других сценариях использования Compose.