Оптимизация Compose

Проблемы, опыт,
тактика и стратегия

RuStore

Рассказать о своём опыте,
осветить проблемы,
попытаться выработать
типовые подходы
и практики

Содержание

Предпосылки и сложности

Множество старых проектов на View	Настороженность разработчиков к Compose, мало квалифицированных кадров. Слабое перетекание кадров из одной области в другую.
Легкость в создании кастомных компонентов	Быстрое и лёгкое создание сложной разметки, но без знания деталей, приводит к падению производительности. А когда уже всё сделано, выяснить причину очень сложно
Технология постоянно развивается	Сообщество ещё прорабатывает типовые подходы, практики и правила

А какие проблемы могут быть в большом проекте?

Откуда мы знаем,
что проблемы вообще есть?

Краегоульные случаи в применении Compase

Краеугольные случаи в применении Compose

которые усложнят вам жизнь, особенно в большой команде

1

Профилировщик

2

Рекомпозиции

3

Оптимизация анимаций

4

Работа с большими
и часто изменяемыми
стейтами. Аннотации

5

Онбоардинг
по Compose

Что делать?

Далее спойлер

Стратегия работы с Compose

Решения

— Типовые оптимизации

— Архитектурные решения

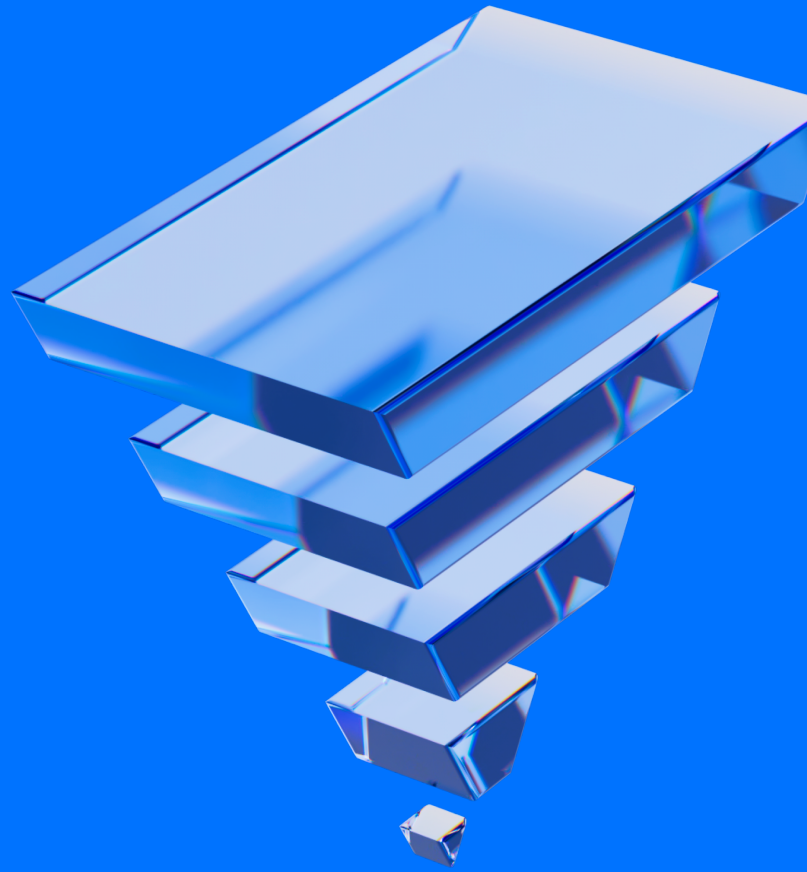
— Увеличить кол-во людей компетентных в компоуз

Мониторинг

— Compose-метрики: рекомпозиций и стабильности функций

— Профилировщик

Мониторинг



Профилировщик

Как давно вы запускали профилировщик?

Реальность

— Запускают только
в случае серьезных инцидентов

— Мощные телефоны
— скрывают проблемы

— Много экранов — слишком
много времени на поиск

Невидимая пост-анимация

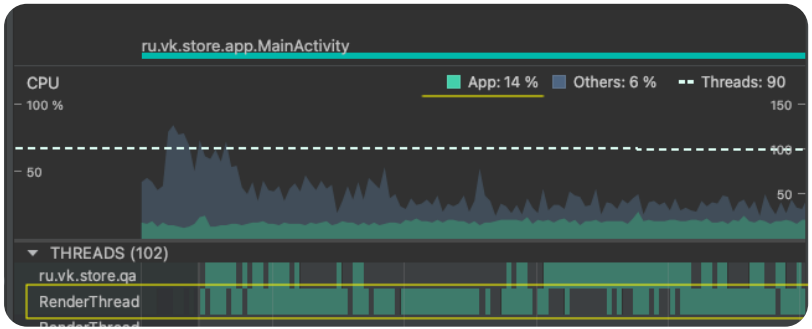
Статистика

До исправления:
Нагрузка: **14**% (общая)
Поток прорисовки почти **полностью забит**

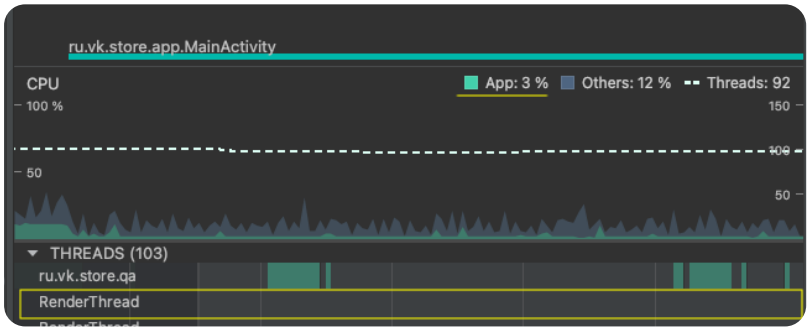
После исправления:
Нагрузка: **1-3**% (общая)
Поток прорисовки **свободен**

Вывод

Смотреть на экраны
с анимациями до и после анимации.
Оптимизировать стадию прорисовки.



До



После

Рекомпозиции

Layot inspector

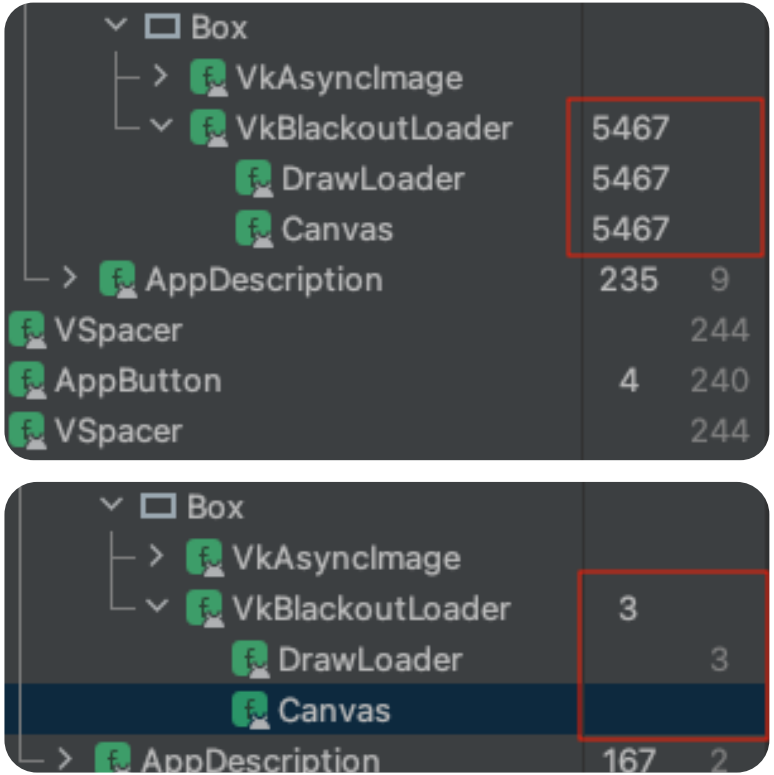
Возможные проблемы:
списки с анимацией

Решение

Анализировать
экраны со списками

Рефакторинг стейта.
Аннотации

Оптимизировать
анимацию



Все проблемы
от анимации и списков?

Оптимизация анимаций

Со сложным алгоритмом

Проблемы

— Нет людей, кто может качественно проревьюить, кроме самого автора.

— Решение зачастую одноразовое, труднорасширяемое

— Плохая производительность

— В случае проблем требуется полный рефакторинг

Решения

— **Типовые оптимизации**

— Compose-метрики:
рекомпозиций и стабильности функций

— Профилировщик

Стратегия работы с Compose

Решения

— **Типовые оптимизации**

— Архитектурные решения

— Увеличить кол-во людей компетентных в компоуз

Мониторинг

— Compose-метрики: рекомпозиций и стабильности функций

— Профилировщик

Типовые оптимизации

Типовые оптимизации

Которые должен уметь делать компоуз-разработчик

Критерии

Ограниченный список

Понимание области применения (когда и как)

Эффективность в большинстве случаев

Аннотации @Immutable и понимание эквивалентности

Стабильность

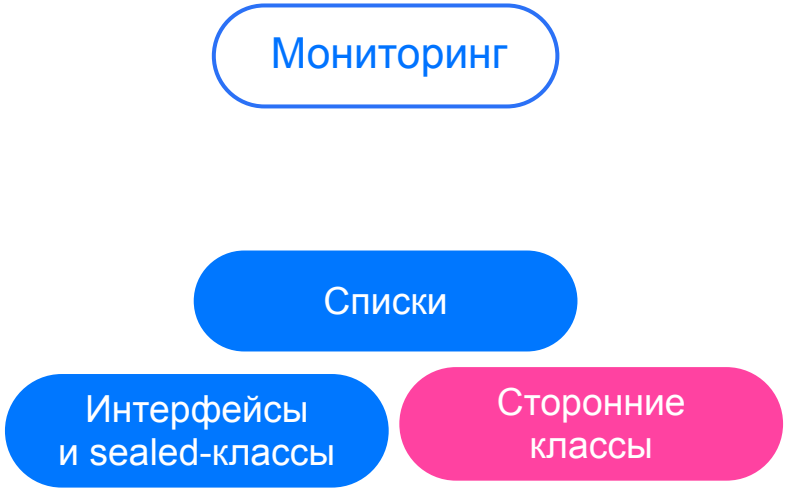
```
@Immutable
interface ViewModelCallbacks

@Immutable
data class Orders (val list: List<Order>)

@Immutable
sealed class Text
```

Эквивалентность

```
// data class умеет проверять
эквивалентность по значению полей.
data class SimpleText (val value: String): Text
```

Compose-метрики: стабильности функций

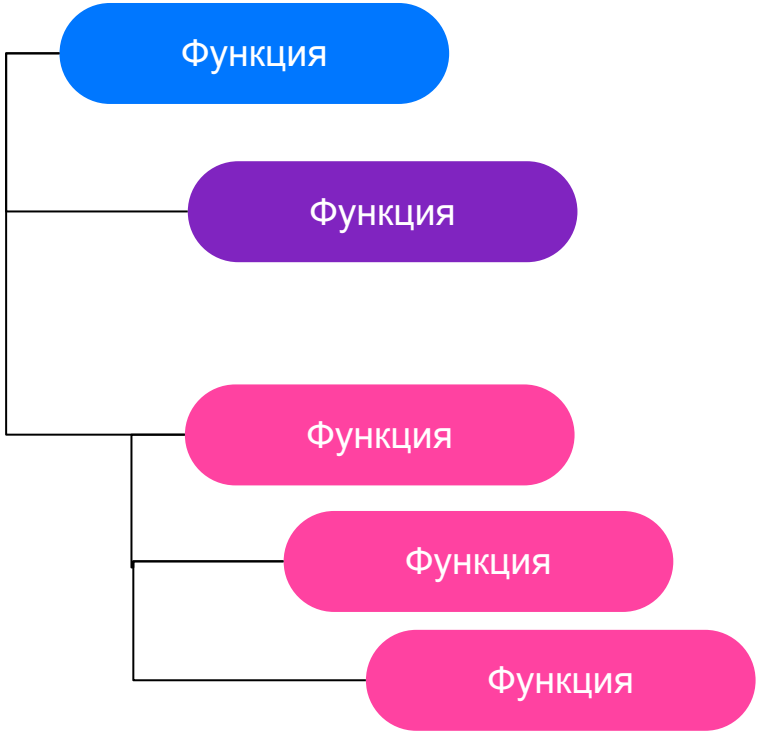
```
stable class CornerBasedShape {
  stable val topStart: CornerSize
  stable val topEnd: CornerSize
  stable val bottomEnd: CornerSize
  stable val bottomStart: CornerSize
  <runtime stability> = Stable
}

Composable Signatures (-composables.txt)
restartable fun Image(
  unstable bitmap: ImageBitmap
  stable contentDescription: String?
  stable modifier: Modifier? = @static Companion
  stable alignment: Alignment? = @dynamic Companion.Center
  stable contentScale: ContentScale? = @dynamic
Companion.Fit
  stable alpha: Float = @static DefaultAlpha
  stable colorFilter: ColorFilter? = @static null
)
```



https://github.com/androidx/androidx/blob/androidx-main/compose/compiler/design/compiler-metrics.md

Обозначения

Чтение (подписка) на State. Проброс State<T> до целевого элемента.



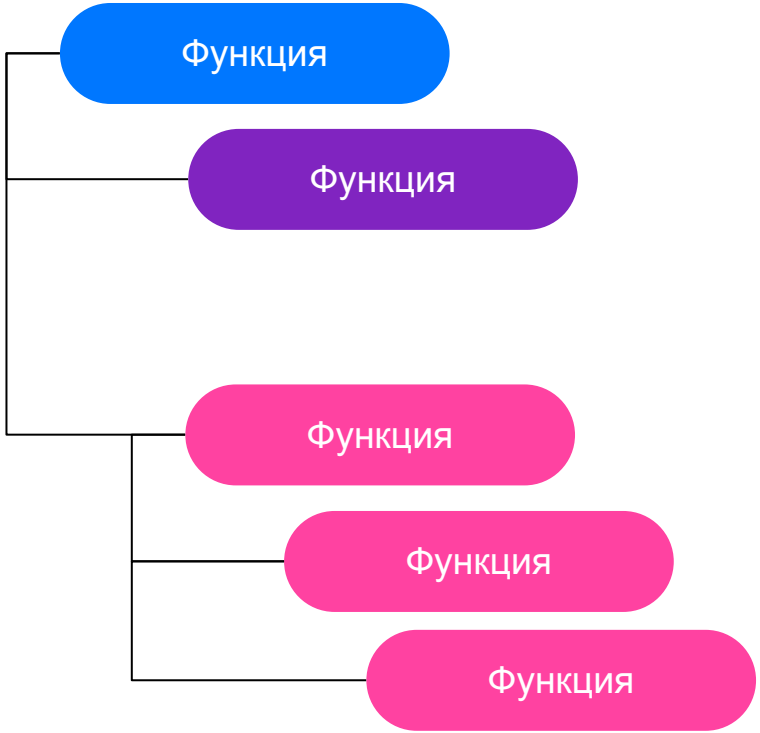
Оптимизированная
(редко выполняемая) функция

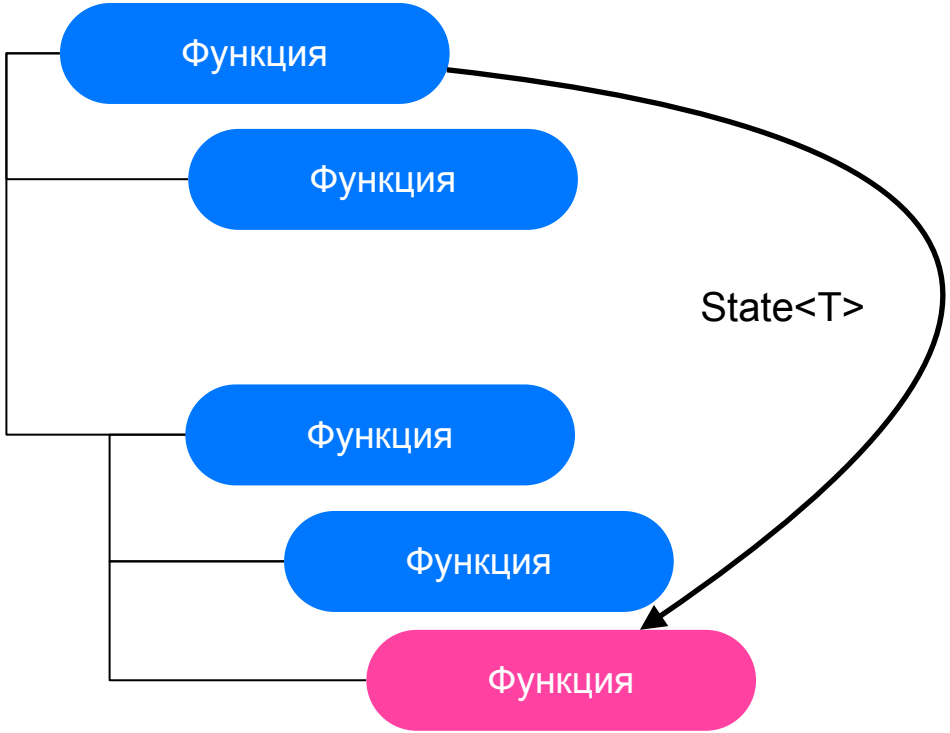
Неоптимизированная функция,
но в зависимости от условий
Compose может пропускать
рекомпозиции

Неоптимизированная (часто
выполняемая) функция

Избавляемся от промежуточных рекомпозиций

Чтение (подписка) на State. Проброс State<T> до целевого элемента





Избавляемся от промежуточных рекомпозиций

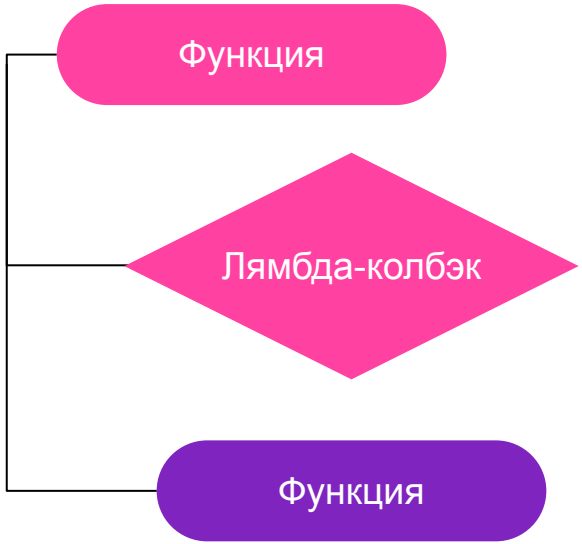
Чтение (подписка) на State. Проброс State<T> до целевого элемента.

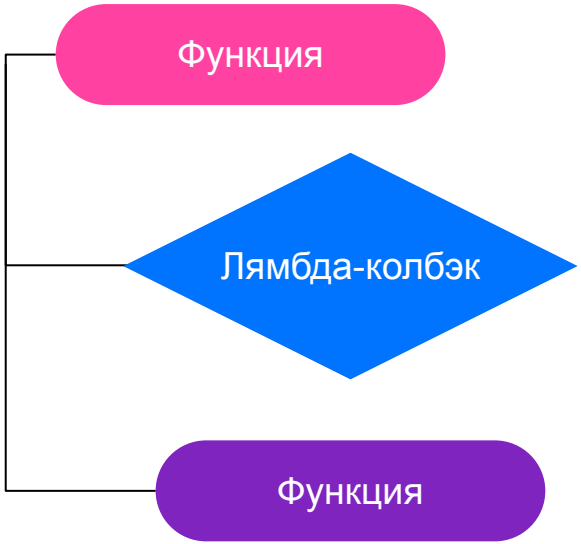
Рекомпозиция только того, что нужно

```
@Composable
fun SomeItem(item: ItemModel) {
    val counter = remember { mutableStateOf(0) } // 1. Не люблю делегаты
    // Клик
    Button(onClick = { counter.value = counter.value + 1  })

   // CustomText(counter.value.toString()).        // 2. String - Рекомпозиция!!
   CustomText(counter).        // 3. State<String> - нет рекомпозиции
}

@Composable
fun CustomText(text: State<String>) {
   Column {
      Text(text.value) // 4. Рекомпозиция
      SomeOtherFunction("something")
   }
}
```

Кэширование и внимательное отношение к лямбдам

rememberUpdateState





Кэширование и внимательное отношение к лямбдам

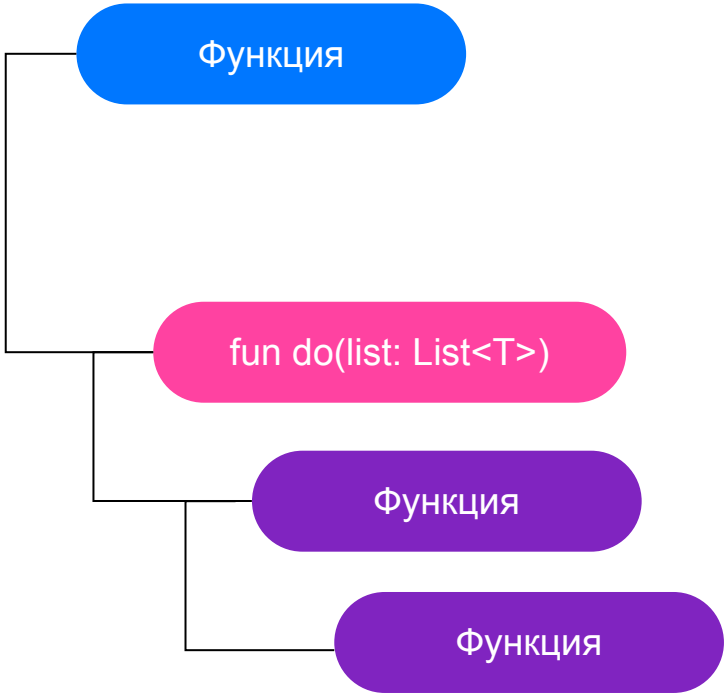
rememberUpdateState

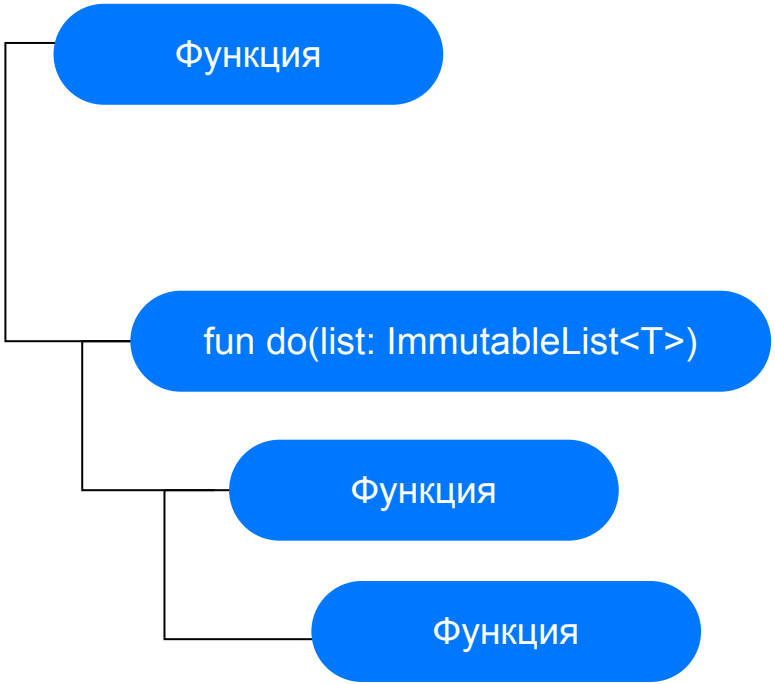
Редко изменяется — редко рекомпозируется

```
@Composable
fun Chapter(model: Model, onSelectItem: (String, Int) -> Unit) {
   // 1. Кэширование, когда title меняется реже чем весь model
    val title = rememberUpdatedState { model.title }

    Column {
        Text(item.title.text)
        InnerList(subtitle = section.subtitle.text) { index ->
            //onSelectItem.invoke(model.title, index) // 2. Так нельзя
            onSelectItem.invoke(title.value, index) // 3. Только при изменении
        }
    }
}
```

Иммутабельные встроенные интерфейсы

Обёртки для типовых нестабильных интерфейсов — List, Map, etc.





Иммутабельные встроенные интерфейсы

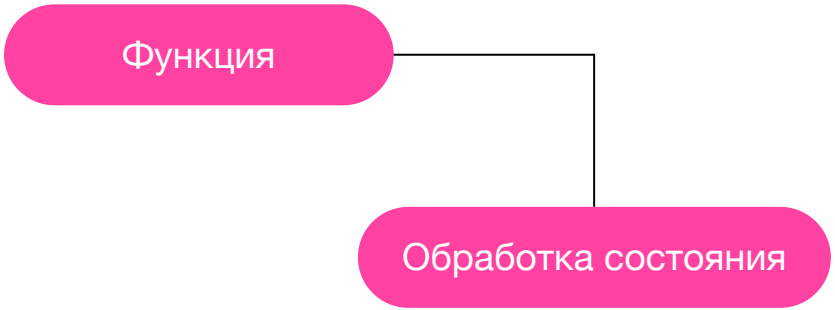
Обёртки для типовых нестабильных интерфейсов — List, Map, etc.

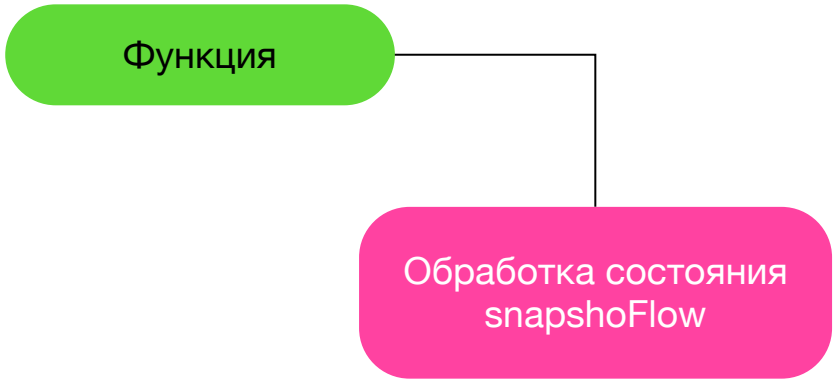
Можно работать как раньше

```
@JvmInline
@Immutable
value class ImmutableList<T> (val data: List<T>) : List<T> by data

@JvmInline
@Immutable
value class ImmutableMap<K,V> (val data: Map<K,V>) : Map<K,V> by data

@Compose
fun accountList(list: ImmutableList<Account>) {
  list.forEach { … } // 1. Работа как с обычным List
```

Обработка изменения стейта

SnapshoFlow





Обработка изменения стейта

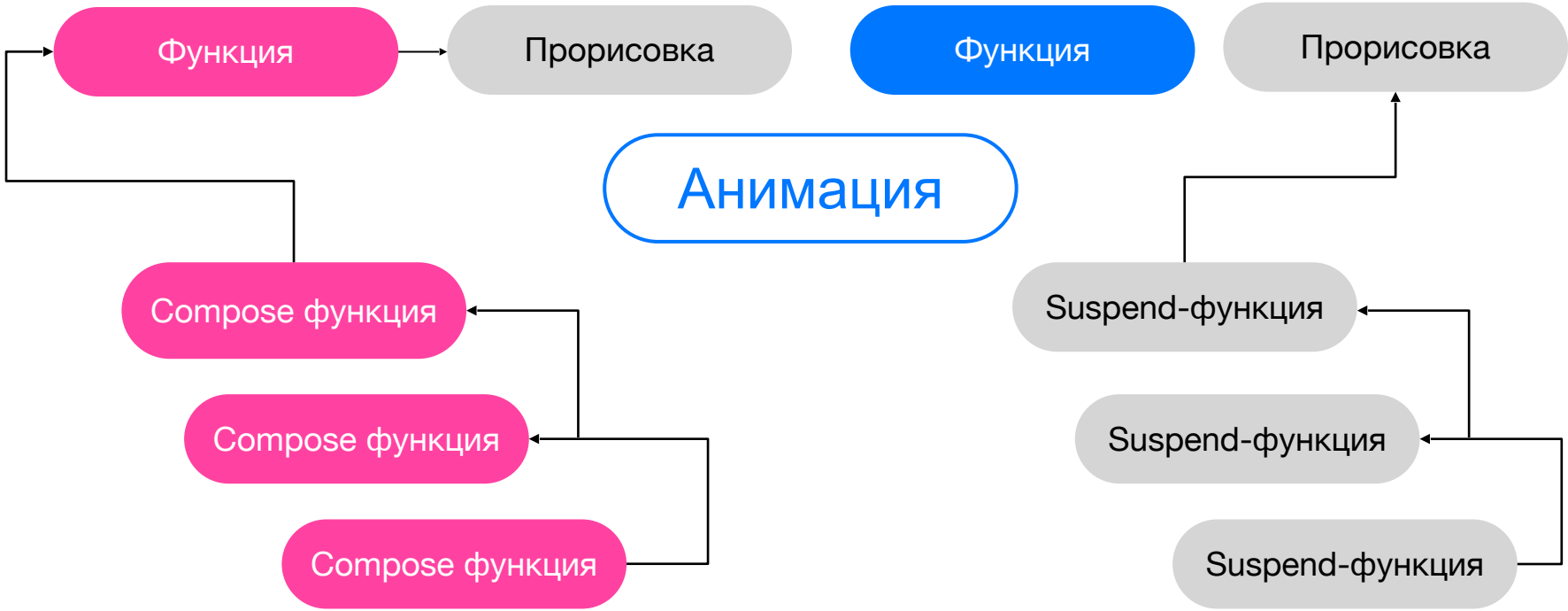
SnapshoFlow

Нет влияния на UI

```
@Composable
fun AccountsPager(list: ImmutableList<Account>, onPage: (Int) -> Unit) {
    val pagerState = rememberPagerState()

    HorizontalPager(pageCount = list.size, state = pagerState,) { page ->
        Column { … }
    }

    // 1. Так нельзя
    // LaunchedEffect(pagerState.currentPage) { onPage(pagerState.currentPage) }

    LaunchedEffect(pagerState) { // 2. Так правильно
        snapshotFlow { pagerState.currentPage } // 3. Подписка на снимок (snapshot)
            .collect { currentPage ->
                onPage(currentPage)
            }
    }
}
```

Анимация без рекомпозиций

Suspend-функции анимации



Анимация без рекомпозиций

Suspend-функции анимации

Производительность

Очень актуально для цепочки вложенных функций анимации

```
@Composable
fun AnimateSomething(something: State<Int>) {
    val animatedValue = remember { Animatable(initialValue = 0f) }

    LaunchedEffect(Unit) {
        snapshotFlow { something.value } // 1. Вне рекомпозиции
            .collect { newValue ->
                animatedValue.animateTo(newValue, tween(DURATION)) // 2. Анимация вне рекомпозиции
            }
    }

    Canvas { drawSomething(animatedValue.value) } // 3. Вне рекомпозиции
}
```

Стратегия работы с Compose

Решения

— Типовые оптимизации

— Архитектурные решения

— Увеличить кол-во людей компетентных в компоуз

Мониторинг

— Compose-метрики: рекомпозиций и стабильности функций

— Профилировщик

Архитектурные решения для UI-стейта

Работа с большими
и часто изменеямыми UI-стейтами

```
@Immutable
data class UiState(
    val header: String,
    val imageUrl: String,
    val items: List<Item>,
    val section: List<Section>,
    val status: Status,
)
```

Меняется очень редко

Меняется очень часто

Работа с большими и часто изменеямыми UI-стейтами

Контекст

Большой иммутабельный стейт (со списками)

Частоизменяемый параметр (статус)

```
@Immutable
data class UiState(
    val header: String,
    val imageUrl: String,
    val items: List<Item>,
    val section: List<Section>,
    val status: Status,
)
```

Меняется очень редко

Меняется очень часто

Проблемы

Пересоздание стейта (copy — нагружаем сборщик мусора)

Рекомпозиции (а если ещё и тяжёлые Lazy-списки!!)

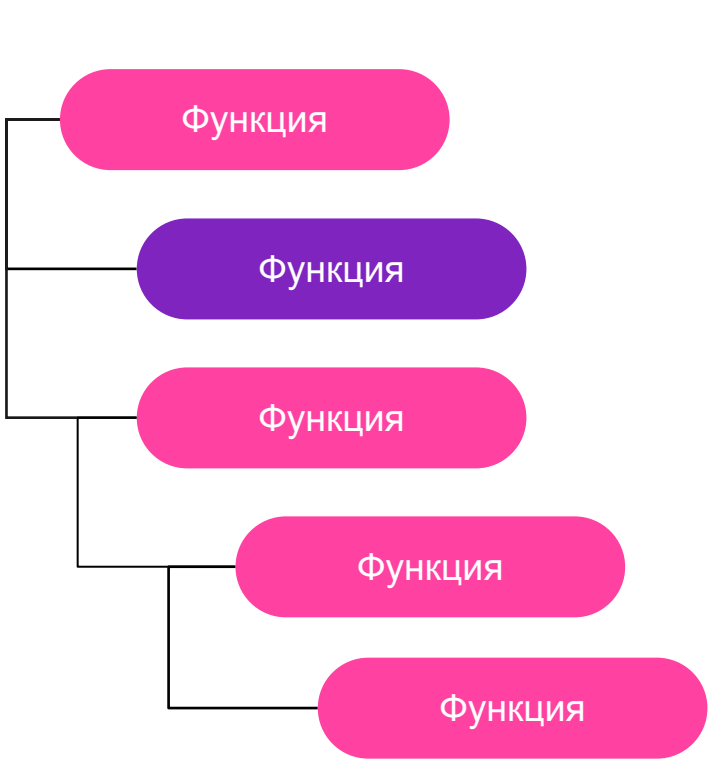
Решение

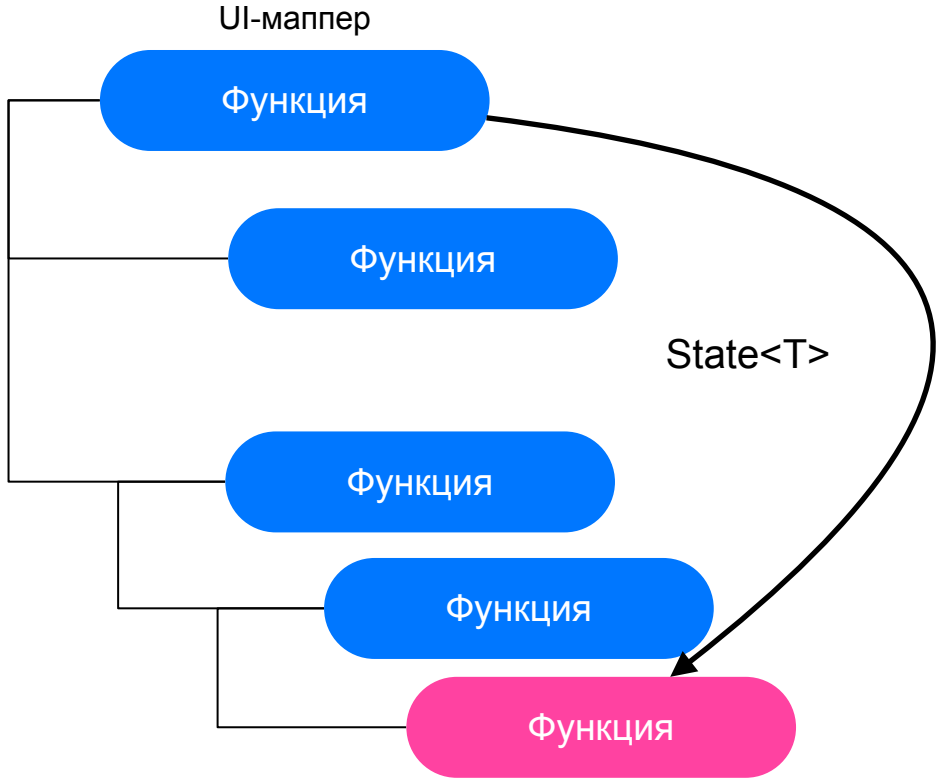
Тактика: типовые оптимизации + везде @Immutable + внимательность!

Стратегия: отойти от полностью иммутабельного стейта

Быстрое решение

Выделяем State<T> для частоизменямых значений

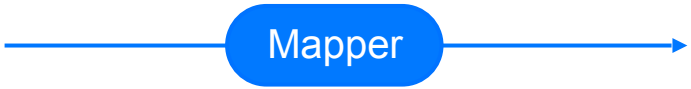




Быстрое решение

Выделяем State<T> для частоизменямых значений

Избыточный код | Логика в UI | **Быстрореализуемо**



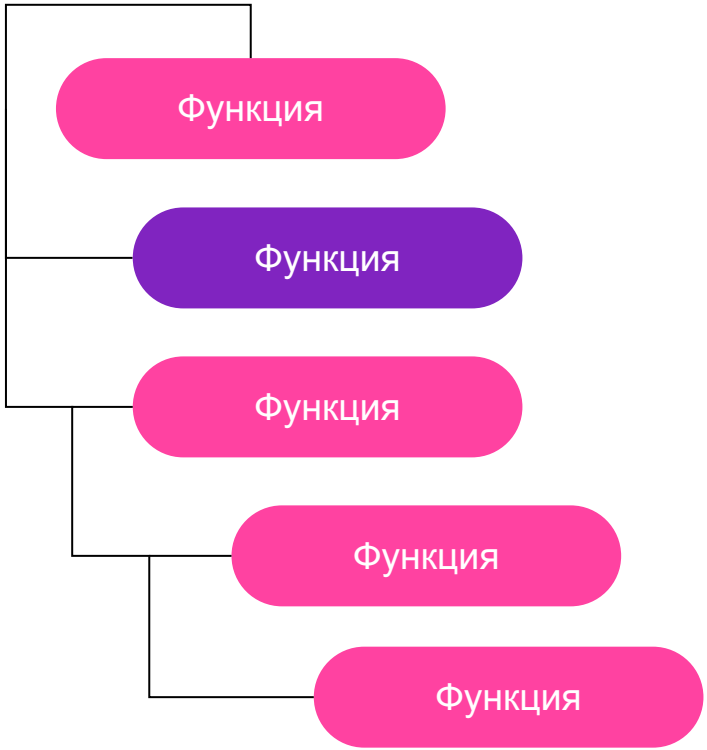

```
@Immutable
data class UiState(
    val header: String,
    val items: List<Item>,
    val status: Status,
)
```

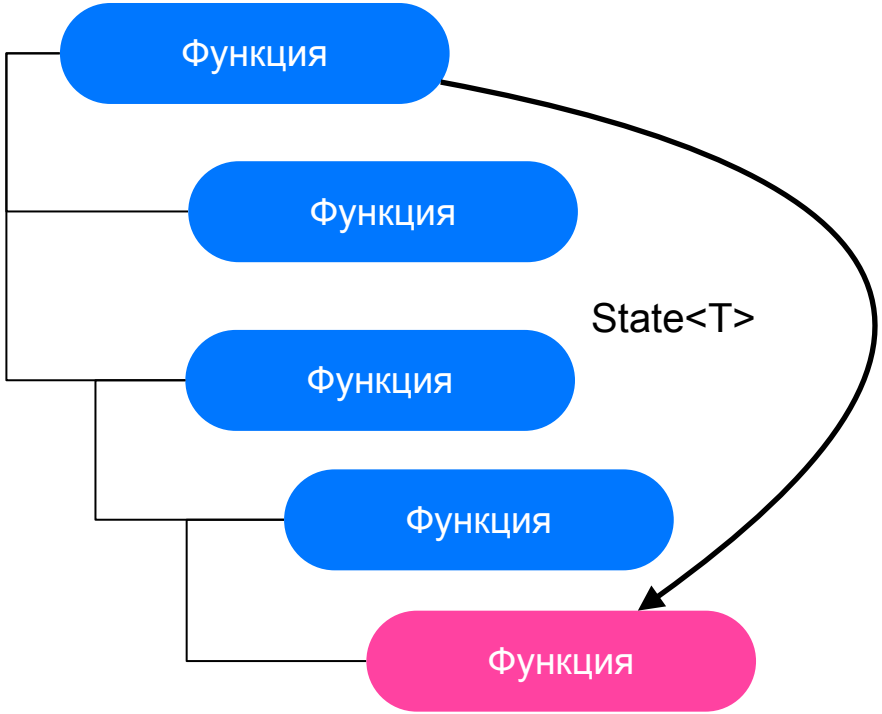
```
@Immutable
data class UiStateWithoutStatus(
    val header: String,
    val items: List<Item>,
)
```

```
@Composable
fun Screen(viewModel: ViewModel) {
    val uiState = viewModel.collectAsState()
    val status = rememberUpdatedState { uiState.status } // 2. Выделяем статус отдельный стейт
    val uiStateWithoutStatus = rememberUpdatedState { uiState.toStateWithoutStatus() } // 3. Меняется редко

    InnerLayout(uiStateWithoutStatus, status) // 4. Передача без рекомпозиции
}

@Composable
fun InnerLayout(uiState: State<UiStateWithoutStatus>, status: State<Status>) {
    Column { Text(uiState.value.header) } // 5. Редкая рекомпозиция
    StatusLayout(status) // 6. Передаём State дальше без рекомпозиции
}
```

Грамотное решение

Выделяем Flow<T> во ViewModel для частоизменямых значений





Грамотное решение

Выделяем Flow<T> во ViewModel для частоизменямых значений

Удобно, эффективно

```
@Immutable
data class StateWithoutStatus(
    val header: String,
    val items: List<Item>,
    val section: List<Section>,
)

@Composable
fun Screen(viewModel: ViewModel) {
   val status = viewModel.collectAsState()

    InnerLayout(viewModel.stateWithoutStatus, status) // 2. Передача стейта статуса без рекомпозиии
}

@Composable
fun InnerLayout(uiState: UiStateWithoutStatus, status: State<Status>) {
    Column { Text(uiState.header) } // 3. Редкая рекомпозиция

     StatusLayout(status) // 4. Передаём State дальше без рекомпозиции
}
```

```
class viewModel: ViewModel {
   // 1. Отельный поток для частоизменяемых значений
   val statusFlow: Flow<Status> = useCase.statusFlow()
}
```

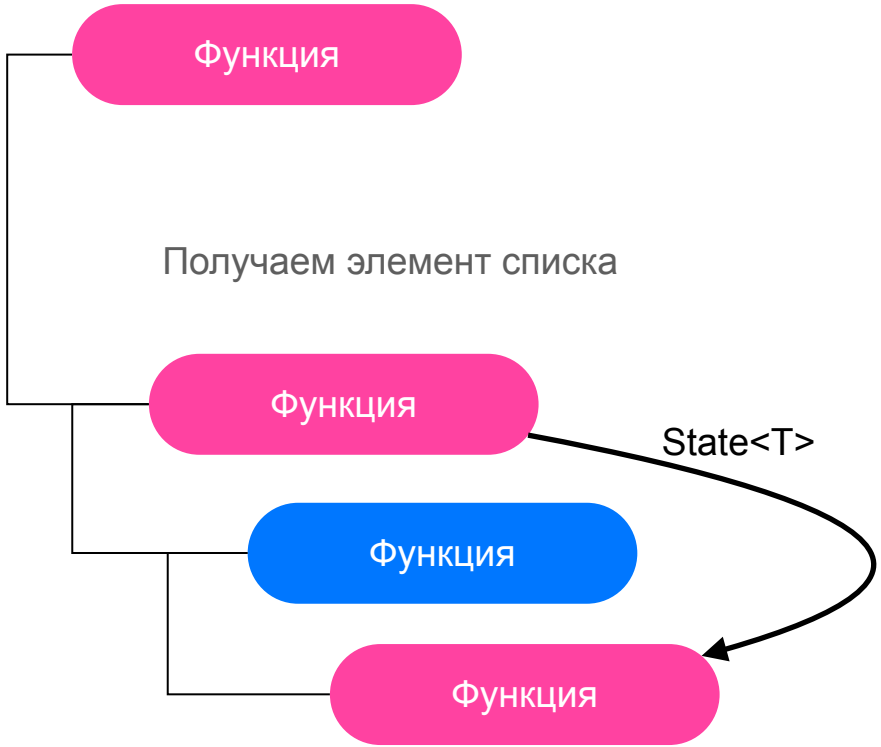
А если много элементов со статусом?

Быстрое решение

```
@Immutable
data class UiState(
    val header: String,
    val items: ImmutableList<ItemWithStatus>,
)
```

Получаем элемент внутри Compose-функции,
далее как в «Быстром решении»

Быстрое решение (делим на State<T>)

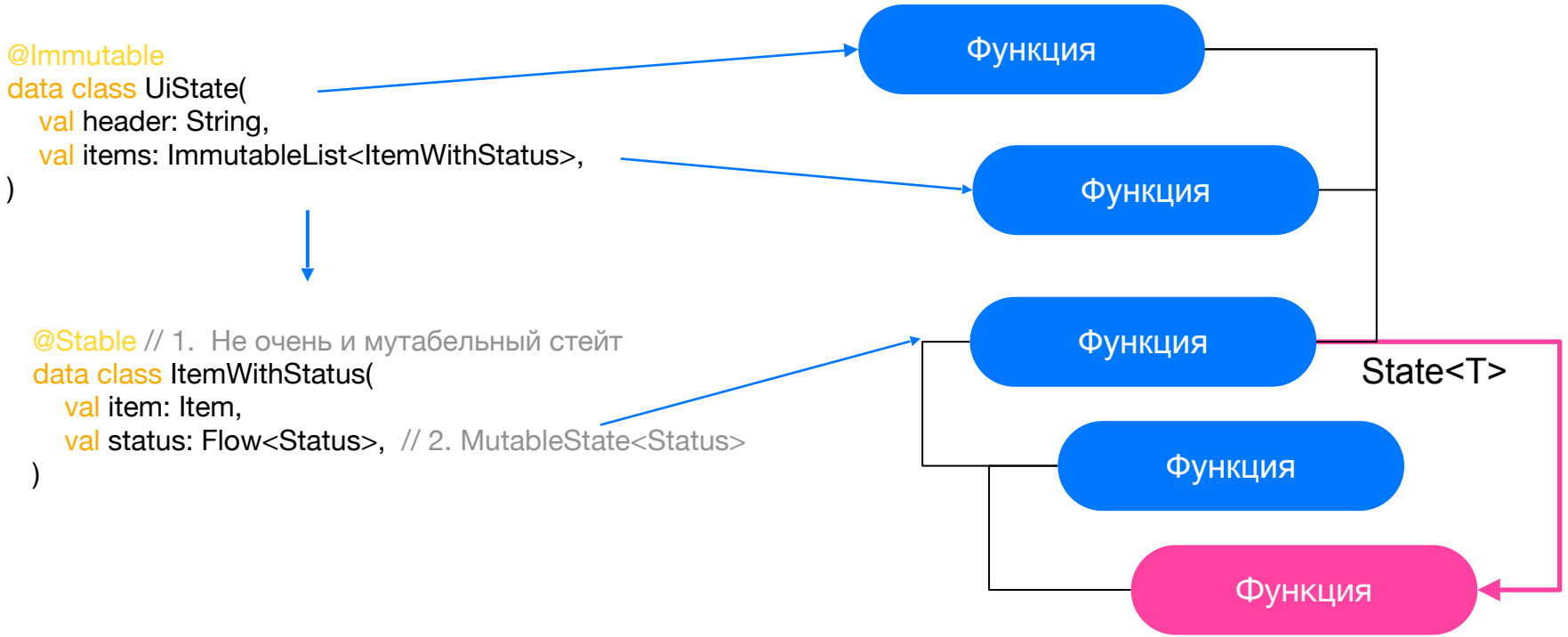


Новое решение — от @Immutable к @Stable

Множественные Flow<T> (State<T>)

```
@Immutable
data class UiState(
    val header: String,
    val items: ImmutableList<ItemWithStatus>,
)
```

```
@Stable // 1.  Не очень и мутабельный стейт
data class ItemWithStatus(
    val item: Item,
    val status: Flow<Status>,  // 2. MutableState<Status>
)
```



Новое решение — от @Immutable к @Stable

Множественные Flow<T> (State<T>)

```
@Immutable
data class UiState(
    val header: String,
    val items: ImmutableList<ItemWithStatus>,
)
```

```
@Immutable
data class ItemWithStatus(
    val item: Item,
    val status: Status,
)
```



```
@Stable // 1.  Не очень и мутабельный стейт
data class ItemWithStatus(
    val item: Item,
    val status: Flow<Status>,  // 2. MutableState<Status>
)
```

```
class viewModel: ViewModel {
  val stateFlow: Flow<UiState> = dataUseCaseFlow()
    .onEach { data ->
      UiState(                          // 3. Создаём стейт
        header = data.header,
        Items = data.items.map { dataItem ->
          ItemWithStatus(          // 4. Отельный объект элемента со статусом
            item = dataItem.item
            status = statusFlowUseCase(dataItem.index) // 4. Отдельный поток на каждый
элемент
          )
        }
    }
```

Новое решение — от @Immutable к @Stable

```
@Composable
fun InnerLayout(uiState: UiState) {
    Column { Text(uiState.header) } // 1. Редкая рекомпозиция

    Items(uiState.items) // 2. Элементы созданы один раз и не рекомпозируются
}
```

```
@Composable
fun Items(items: ImmutableList<ItemWithStatus>) {
    LazyColumn {
        items(items.size) { index ->
            Row {
                Text(items[index].text) // 3. Редкая рекомпозиция

                val status = items[index].status.collectAsState()
                StatusLayout(status) // 4. Нет рекомпозиции
            }
        }
    }
}
```

```
@Composable
fun StatusLayout(status: State<Status>) {
    // 5. Частая рекомпозиция
    Column { Text(status.value.text) }
}
```

```
@Stable
data class ItemWithStatus(
    val item: Item,
    val status: Flow<Status>,  // MutableState<Status>
)
```

Стратегия работы с Compose

Решения

— Типовые оптимизации

— Архитектурные решения

— **Увеличить кол-во людей компетентных в компоуз**

Мониторинг

— Compose-метрики: рекомпозиций и стабильности функций

— Профилировщик

Онбоардинг в Compose

Онбоардинг в Compose

Типовые оптимизации

Архитектурные решения

— Slot-api:
когда применяем и зачем

— Колбэки: передаём отдельно
или интерфейсом

— Иммутабельный стейт

— И т.п.

Code-style и **detect**

— *Compose-метрики:*
рекомпозиций
и стабильности функций

— *Профилировщик*

Выводы

Выводы

Решения

— Типовые оптимизации

— Архитектурные решения

— Увеличить кол-во людей компетентных в компоуз

Мониторинг

— Compose-метрики: рекомпозиций и стабильности функций

— Профилировщик

Выводы

Обращаем внимание на списки и анимации (особенно). Ревью.

Определяемся с типовыми практиками.

Делимся знаниями.

Поменьше «супервелосипедо».

Мониторим. *(Хотя бы иногда)*

Q&A