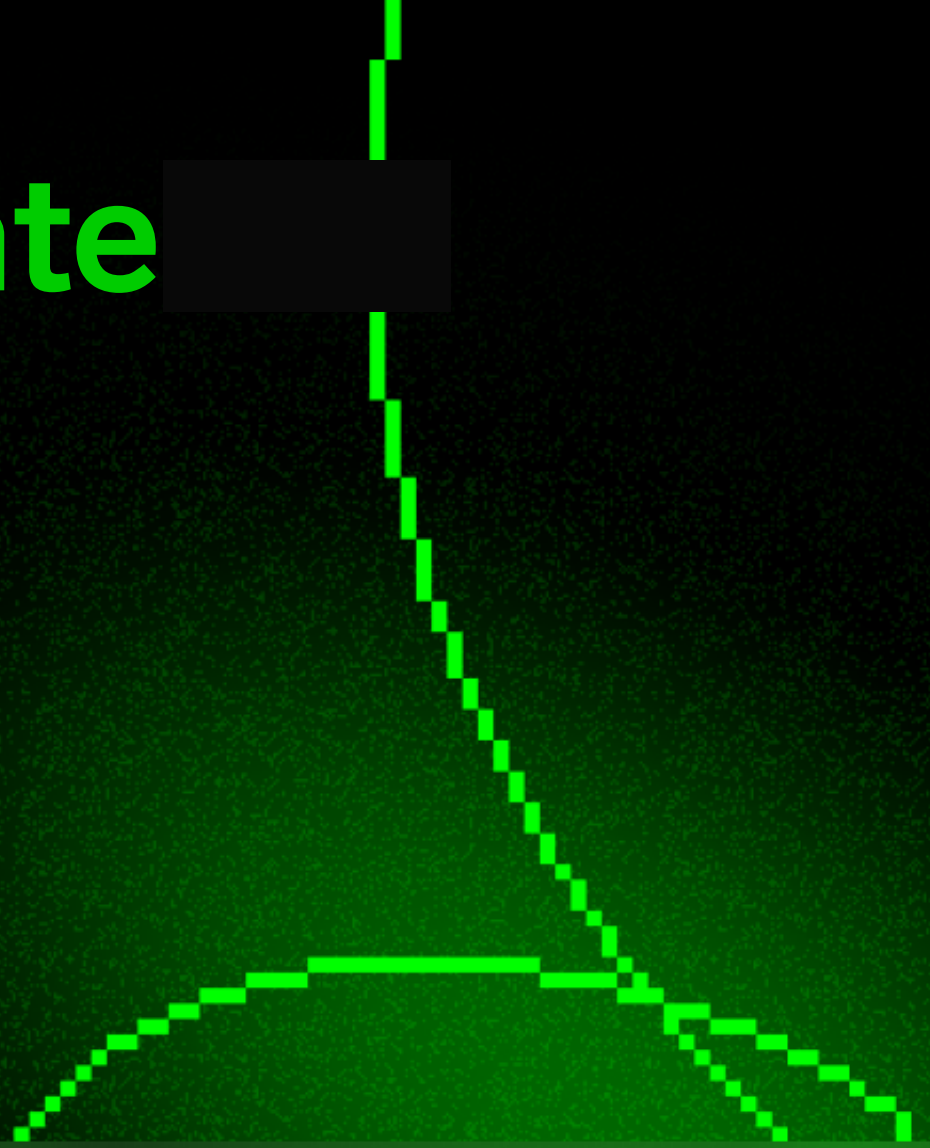
Что скрывает State в Compose



Контур

Алексей Панов

Ведущий инженер-программист


ОНЛАЙН СОБЕСЕДОВАНИЕ
COMPOSE
0:00 / 1:29:37

План доклада

1. Для чего Compose нужен State
2. Как устроен State под капотом
3. Как работать со Snapshot API
4. Делаем свой MutableState
5. Как происходит чтение и запись в MutableState
6. Разбираемся в магии рекомпозиии Compose

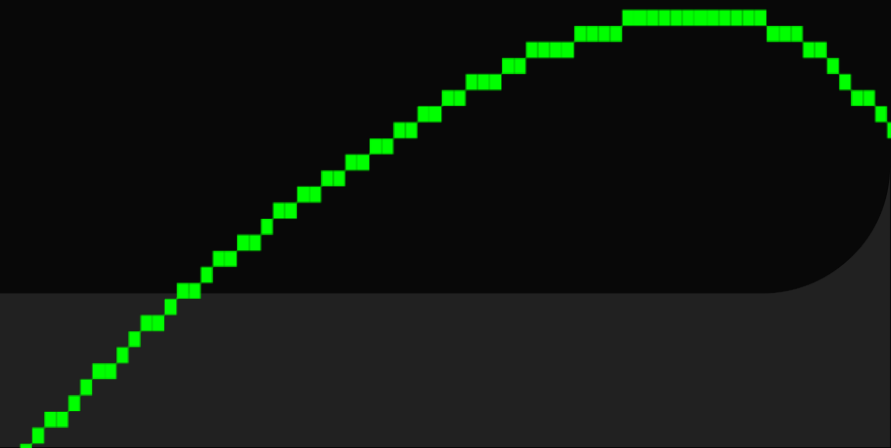
Обо мне

- Tech Lead мобильной core team в Контуре
- Автор telegram канала @kotlin_adept
- Mobile Broadcast Expert (Kotlin, Android)

ajiekcx

Алексей Панов

Для чего Compose нужен State



Рекомпозиция в Compose

```
@Composable
fun ClickCounter(clicks: Int, onClick: () → Unit) {
    Button(onClick = onClick) {
        Text("I've been clicked $clicks times")
    }
}
```

Рекомпозиция в Compose

```
@Composable
fun BadCounterSample() {
    var clicks = 0 // DON'T DO IT
    ClickCounter(clicks, onClick = { clicks++ })
}
```

Рекомпозиция в Compose

```
var clicks = 0 // DON'T DO IT

@Composable
fun VeryBadCounterSample() {
    val recomposeScope = currentRecomposeScope
    ClickCounter(
        clicks,
        onClick = { clicks++; recomposeScope.invalidate() }
    )
}
```

Рекомпозиция в Compose

```
@Composable
fun GoodCounterSample() {
    var clicks by remember { mutableStateOf(0) }
    ClickCounter(clicks, onClick = { clicks++ })
}
```

State ≠ Compose

```
var clicks by remember { mutableStateOf(0) }
```

- by — ключевое слово Kotlin для реализации делегатов

```
getValue(thisObj: Any?, property: KProperty<*>): T

setValue(thisObj: Any?, property: KProperty<*>, value: T)
```

State ≠ Compose

```
val clicks = remember { mutableStateOf(0) }
```

- by — ключевое слово Котлин для реализации делегатов

State ≠ Compose

```
val clicks = remember { mutableStateOf(0) }
```

- by — ключевое слово Котлин для реализации делегатов
- remember — сохраняет значение между вызовами функции


mobius
Positional memoization, или Как работает одна из главных концепций Jetpack Compose
Дмитрий Григорьев
EPAM Systems

State ≠ Compose

```
val clicks = viewModel.clicks
```

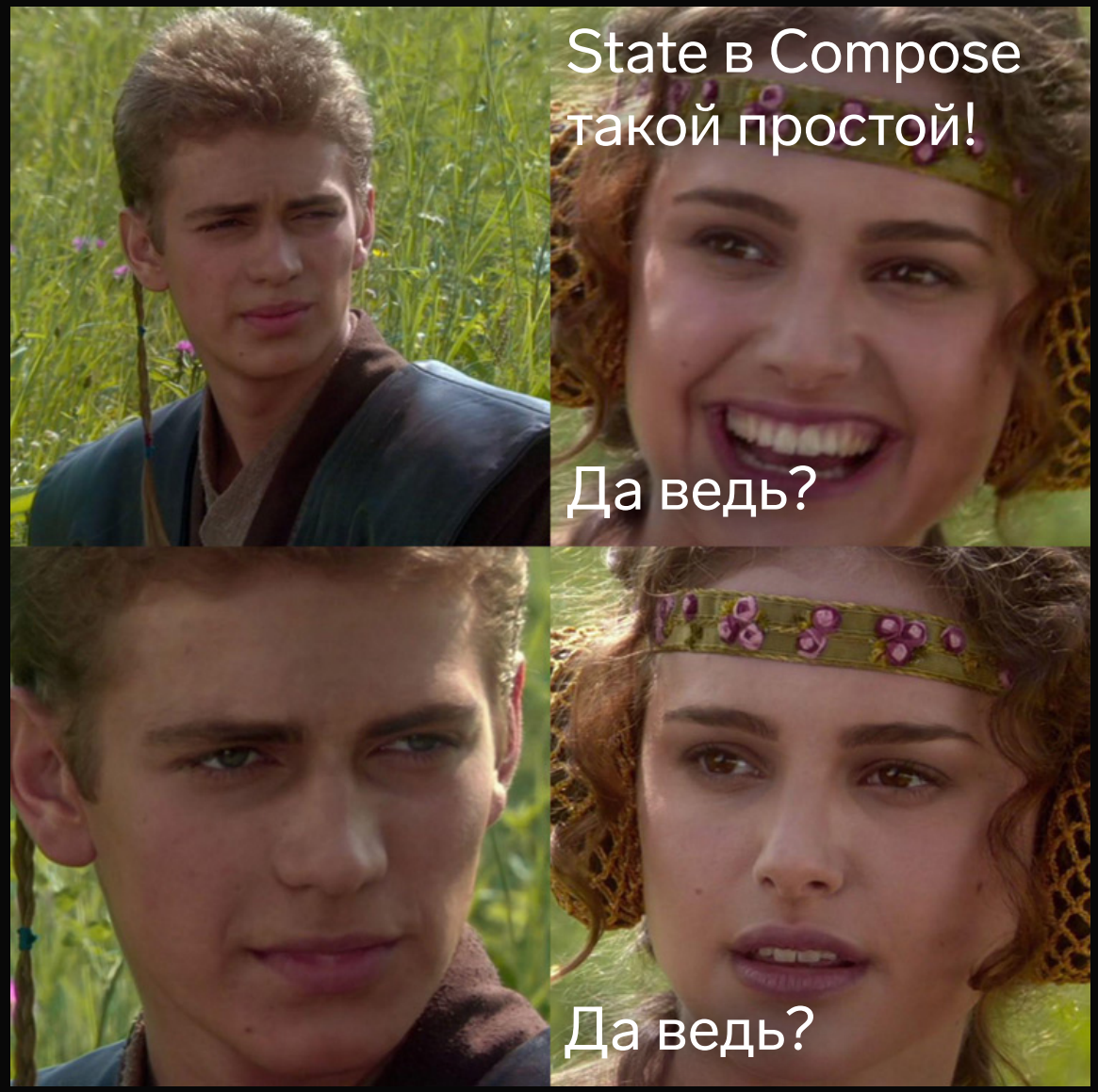
- by — ключевое слово Котлин для реализации делегатов
- remember — сохраняет значение между вызовами функции

```
class CounterViewModel: ViewModel() {
    val clicks = mutableStateOf(0)
}
```

State и MutableState

```
@Stable
interface State<out T> {
    val value: T
}
```

```
@Stable
interface MutableState<T> : State<T> {
    override var value: T
    operator fun component1(): T
    operator fun component2(): (T) → Unit
}
```


State в Compose
такой простой!
Да ведь?
Да ведь?

Flow to State конвертер

```
@Composable
fun <T : R, R> Flow<T>.collectAsState(
    initial: R,
    context: CoroutineContext = EmptyCoroutineContext
): State<R> = produceState(initial, this, context) {
    if (context == EmptyCoroutineContext) {
        collect { value = it }
    } else withContext(context) {
        collect { value = it }
    }
}
```

State to Flow конвертер

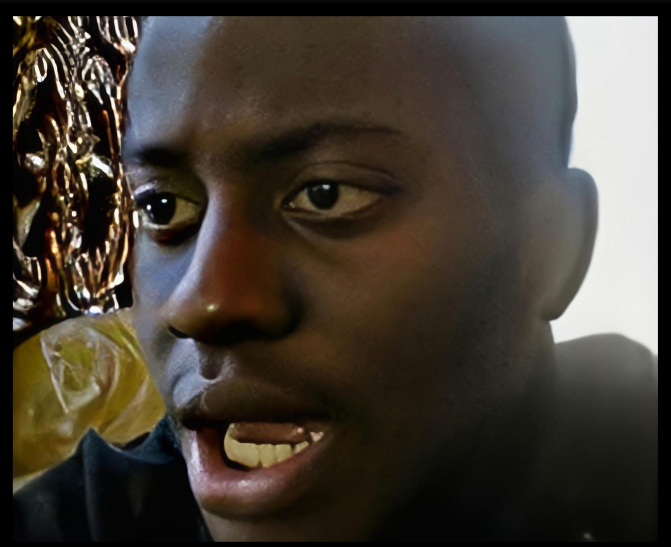
```
fun <T> snapshotFlow(
    block: () → T
): Flow<T> = flow {
    val readSet = mutableSetOf<Any>()
    val readObserver: (Any) → Unit = { readSet.add(it) }

    val appliedChanges = Channel<Set<Any>>(Channel.UNLIMITED)

    val unregisterApplyObserver = Snapshot.registerApplyObserver { changed, _ →
        appliedChanges.trySend(changed)
    }

    try {
        var lastValue = Snapshot.takeSnapshot(readObserver).run {
            try {
                enter(block)
            } finally {
                dispose()
            }
        }
        emit(lastValue)

        while (true) {
            var found = false
```



Зачем изобретать велосипед

Зачем изобретать велосипед

RxJava

Зачем изобретать велосипед

RxJava

Flow

Зачем изобретать велосипед

RxJava

Flow

LiveData


Flow
Developer
LiveData
ObservableField

Зачем изобретать велосипед

- Предоставляем простой и удобный API для пользователя

Зачем изобретать велосипед

- Предоставляем простой и удобный API для пользователя
- Убираем лишний boilerplate

Зачем изобретать велосипед

- Предоставляем простой и удобный API для пользователя
- Убираем лишний boilerplate
- Упрощаем работу с concurrency

Борьба с shared state в корутинах

Single thread dispatcher

Actors

Semaphore

Mutex

Atomics

Synchronized

Volatile

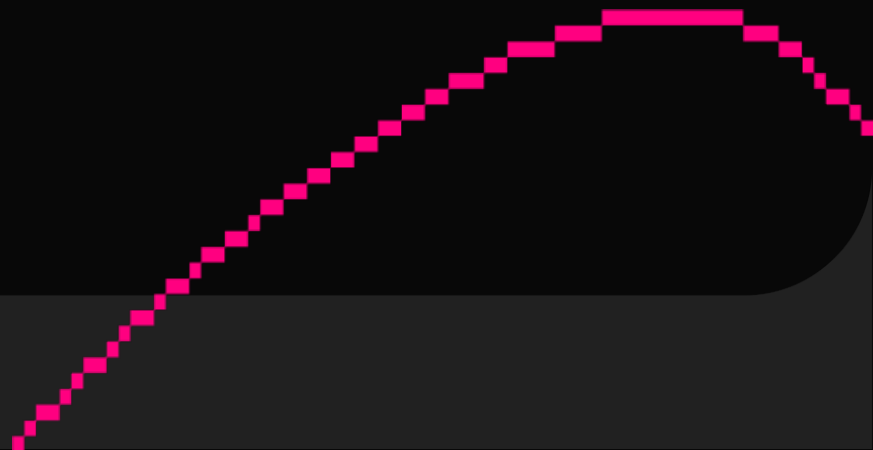
Channels

Борьба с shared state в Compose State

```
interface SnapshotMutationPolicy<T> {
    fun equivalent(a: T, b: T): Boolean

    fun merge(previous: T, current: T, applied: T): T? = null
}
```

Как устроен State под капотом



Concurrency control

- Свод правил для обеспечения корректности системы при конкурентных операциях

Concurrency control

- Свод правил для обеспечения корректности системы при конкурентных операциях

Optimistic

Pessimistic

Semi-optimistic

Механизм работы State в Compose является **оптимистичным**

MVCC

- Multiversion concurrency control используется СУБД для обеспечения одновременного доступа к базе данных и в языках программирования для реализации транзакционной памяти

ACID

ACID
Atomicity

«Всё или ничего»

ACID Consistency

«Не сломай»

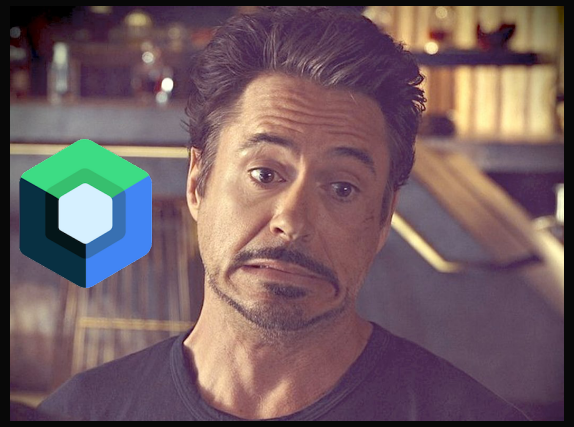
ACID Isolation

«Держи в тайне»

ACID Durability

«Не потеряй»

ACID ~~Durability~~



~~«Не потеряй»~~

Database

- ACID
- Transactions
- Commit/rollback

Compose State

- ACI
- Snapshots
- Apply/dispose

Как работать со Snapshot API

Snapshot

Snapshot

```
val name = mutableStateOf("Mobius")
val snapshot = Snapshot.takeSnapshot()
```

Snapshot

```
val name = mutableStateOf("Mobius")
val snapshot = Snapshot.takeSnapshot()

name.value = "Joker"
println(name.value)
```

Snapshot

```
val name = mutableStateOf("Mobius")
val snapshot = Snapshot.takeSnapshot()

name.value = "Joker"
println(name.value)

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

Joker
Mobius

Snapshot

```
val name = mutableStateOf("Mobius")
val snapshot = Snapshot.takeSnapshot()

name.value = "Joker"
println(name.value)

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

Joker
Mobius
Joker

Snapshot

```
val name = mutableStateOf("Mobius")
val snapshot = Snapshot.takeSnapshot()

name.value = "Joker"
println(name.value)

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

snapshot.dispose()
```

Joker
Mobius
Joker

Один Snapshot для всех стейтов

```
val name = mutableStateOf("Mobius")
val date = mutableStateOf("01.11.2023")
val snapshot = Snapshot.takeSnapshot()
name.value = "Joker"
date.value = "09.10.2023"

snapshot.enter {
    println(name.value)
    printDate(date)
}
```

Mobius
01.11.2023

Изменим данные в снапшоте

```
val name = mutableStateOf("Mobius")
val snapshot = Snapshot.takeSnapshot()

snapshot.enter {
    name.value = "Joker"
    println(name.value)
}
```

Изменим данные в снапшоте

```
val name = mutableStateOf("Mobius")
val snapshot = Snapshot.takeSnapshot()

snapshot.enter {
    name.value = "Joker"
    println(name.value)
}
```

Cannot modify a state object in a read-only snapshot

Изменяемый снапшот

```
val name = mutableStateOf("Mobius")
val snapshot = Snapshot.takeMutableSnapshot()

snapshot.enter {
    name.value = "Joker"
    println(name.value)
}
println(name.value)
```

Joker
Mobius

Применяем снапшот

```
val name = mutableStateOf("Mobius")
val snapshot = Snapshot.takeMutableSnapshot()

snapshot.enter {
    name.value = "Joker"
    println(name.value)
}
snapshot.apply().check()

println(name.value)
```

Joker
Joker

Сокращаем код

```
val name = mutableStateOf("Mobius")
Snapshot.withMutableSnapshot {
    name.value = "Joker"
    println(name.value)
}
println(name.value)
```

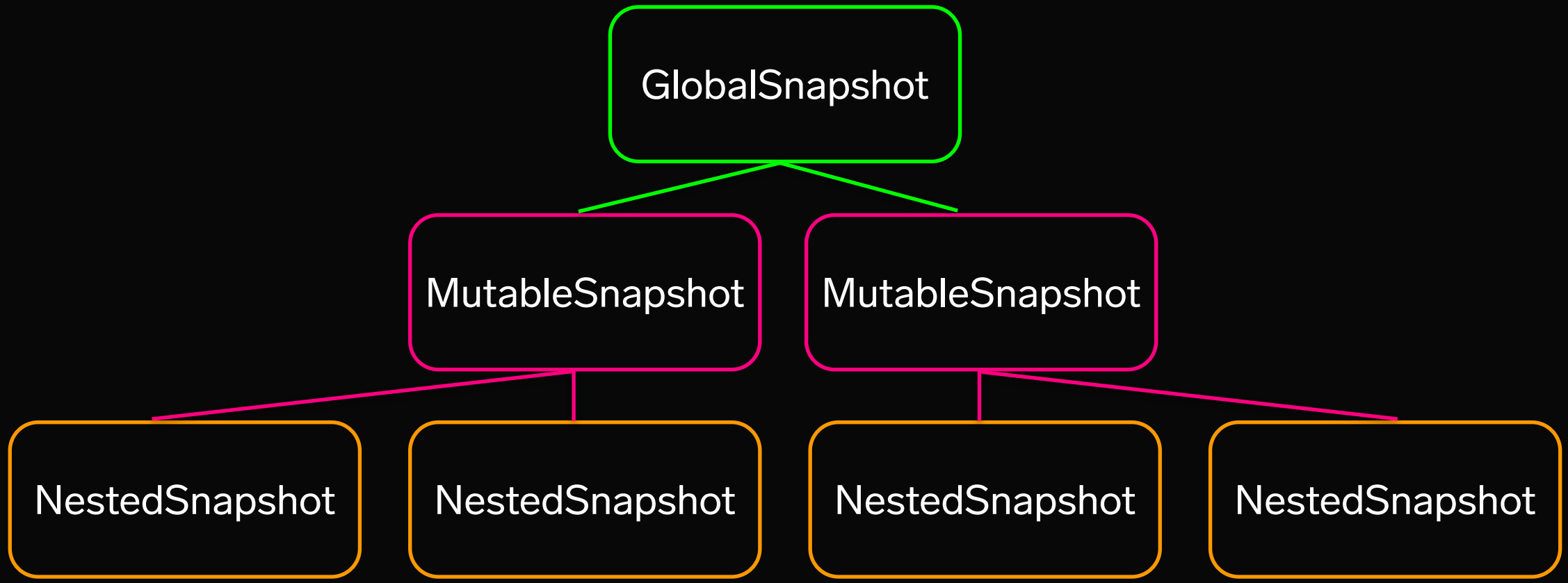
```
Joker
Joker
```

Вложенные снапшоты

```
val name = mutableStateOf("Mobius")
Snapshot.withMutableSnapshot {
    name.value = "Joker"
    println(name.value)
    Snapshot.withMutableSnapshot {
        name.value = "HolyJS"
    }
}
println(name.value)
```

Joker
HolyJS

Иерархия снапшотов



Global snapshot

- Глобальный снапшот только один
- Не может быть применен (apply) и удален (dispose)
- При изменении глобальный снапшот удаляется и создается новый (advanced)

Подписка на чтение и запись

```
val name = mutableStateOf("Mobius")

val readObserver: (Any) -> Unit = { readState ->
    if (readState == name) println("value was read")
}
val writeObserver: (Any) -> Unit = { writtenState ->
    if (writtenState == name) println("value was written")
}

val snapshot = Snapshot.takeMutableSnapshot(readObserver, writeObserver)
```

Подписка на глобальный снапшот

```
val name = mutableStateOf("Mobius")

Snapshot.registerApplyObserver { changedSet, snapshot →
    if (name in changedSet) println("name was changed")
}

name.value = "Joker"
Snapshot.sendApplyNotifications()
```

Конкурирующая запись

```
val conferences = mutableStateOf(listOf("Mobius"))
val snapshot1 = Snapshot.takeMutableSnapshot()
snapshot1.enter { conferences.value += "Joker" }
val snapshot2 = Snapshot.takeMutableSnapshot()
snapshot2.enter { conferences.value += "HolyJS" }

snapshot1.apply()
snapshot2.apply()
println(conferences.value)
```

Конкурирующая запись

```
val conferences = mutableStateOf(listOf("Mobius"))
val snapshot1 = Snapshot.takeMutableSnapshot()
snapshot1.enter { conferences.value += "Joker" }
val snapshot2 = Snapshot.takeMutableSnapshot()
snapshot2.enter { conferences.value += "HolyJS" }

snapshot1.apply()
snapshot2.apply()
println(conferences.value)
```

1. Не применится ничего
2. Упадем в runtime
3. Применится первый
4. Применится второй

Конкурирующая запись

```
val conferences = mutableStateOf(listOf("Mobius"))
val snapshot1 = Snapshot.takeMutableSnapshot()
snapshot1.enter { conferences.value += "Joker" }
val snapshot2 = Snapshot.takeMutableSnapshot()
snapshot2.enter { conferences.value += "HolyJS" }

snapshot1.apply()
snapshot2.apply()
println(conferences.value)
```

1. Не применится ничего
2. Упадем в runtime
3. ✅ Применится первый
4. Применится второй

Конкурирующая запись

```
val conferences = mutableStateOf(listOf("Mobius"))
val snapshot1 = Snapshot.takeMutableSnapshot()
snapshot1.enter { conferences.value += "Joker" }
val snapshot2 = Snapshot.takeMutableSnapshot()
snapshot2.enter { conferences.value += "HolyJS" }

snapshot1.apply()
snapshot2.apply()
println(conferences.value)
```

[Mobius, Joker]

Результат apply

```
sealed class SnapshotApplyResult {

    abstract fun check()

    object Success : SnapshotApplyResult()

    class Failure(val snapshot: Snapshot) : SnapshotApplyResult()
}
```

SnapshotMutationPolicy

```
class ConferenceMutationPolicy : SnapshotMutationPolicy<List<String>> {
    override fun equivalent(a: List<String>, b: List<String>): Boolean = a == b

    override fun merge(
        previous: List<String>, // [Mobius]
        current: List<String>, // [Mobius, Joker]
        applied: List<String> // [Mobius, HolyJS]
    ): List<String> {
        return (current + previous + applied).distinct()
    }
}
```

Конкурирующая запись

```
val conferences = mutableStateOf(
    value = listOf("Mobius"),
    policy = ConferenceMutationPolicy()
)
val snapshot1 = Snapshot.takeMutableSnapshot()
snapshot1.enter { conferences.value += "Joker" }
val snapshot2 = Snapshot.takeMutableSnapshot()
snapshot2.enter { conferences.value += "HolyJS" }

snapshot1.apply()
snapshot2.apply()
println(conferences.value)
```

[Mobius, Joker, HolyJS]


SnapshotMutationPolicy проставлю
Безопасно стейт менять буду

```
val conferences = mutableStateOf(
    value = listOf(),
    policy = ConferenceMutationPolicy()
)
```

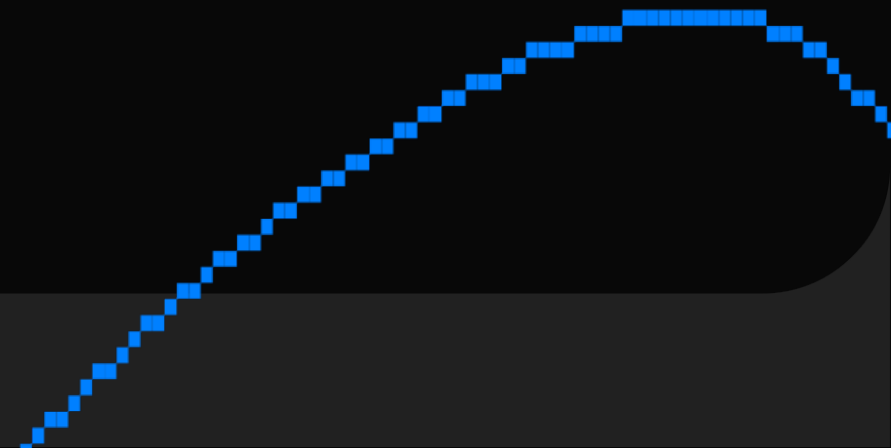
```
withContext(Dispatchers.Default) {
    massiveRun { value →
        Snapshot.withMutableSnapshot {
            conferences.value += value
        }
    }
}
```

```
withContext(Dispatchers.Default) {
    massiveRun { value →
        conferences.value += value
    }
}
```

Что мы узнали про снапшоты

- Как-то связаны со стейтом
- Могут быть глобальным, вложенным, изменяемым и неизменяемым
- Изолированы друг от друга
- Позволяют подписаться на изменение состояния

Делаем свой MutableState с блэкджеком и снапшотами



SimpleMutableState

```
class SimpleMutableState(_value: String) {
    var value: String = _value
}
```

SimpleMutableState

```
class SimpleMutableState(_value: String) {
    private val record = Record(_value)

    var value: String
        get() = record.value
        set(value) {
            record.value = value
        }

    private data class Record(
        var value: String
    )
}
```

SimpleMutableState

```
class SimpleMutableState(_value: String) {
    private val records = mutableListOf(Record(_value))
    private var currentRecordIndex = 0
    private val currentRecord
        get() = records[currentRecordIndex]

    var value: String
        get() = currentRecord.value
        set(value) {
            currentRecord.value = value
        }
...
}
```

SimpleMutableState

```
class SimpleMutableState(_value: String) {
...
    fun takeSnapshot(): Int {
        val id = currentRecordIndex
        records += currentRecord.copy()
        currentRecordIndex = records.lastIndex

        return id
    }

    fun restoreSnapshot(id: Int) {
        currentRecordIndex = id
    }
}
```

Проверяем

```
val name = SimpleMutableState("Mobius")
val snapshot = name.takeSnapshot()

name.value = "Joker"
println(name.value)

name.restoreSnapshot(snapshot)
println(name.value)
```

Joker
Mobius

SimpleMutableState

- Снапшоты живут вечно
- Записи хранятся в MutableList
- Нет поддержки вложенных снапшотов
- Нельзя подписаться на изменения

SnapshotState

- Снапшоты живут, пока явно не будут уничтожены
- Записи хранятся в связанном списке
- Есть поддержка вложенных снапшотов
- Можно подписаться на изменения

StateRecord

```
abstract class StateRecord {
    internal var snapshotId: Int = currentSnapshot().id

    internal var next: StateRecord? = null

    abstract fun assign(value: StateRecord)

    abstract fun create(): StateRecord
}
```

Наследуемся от StateRecord

```
private data class Record(
    var value: String
) : StateRecord() {
    override fun assign(value: StateRecord) {
        value as Record
        this.value = value.value
    }

    override fun create(): StateRecord = Record(value)
}
```

StateObject

```
interface StateObject {

    val firstStateRecord: StateRecord

    fun prependStateRecord(value: StateRecord)

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

Реализуем StateObject

```
class SimpleMutableState(_value: String) : StateObject {
    private var records = Record(_value)

    var value: String
        get() = ...
        set(value) {...}

    override val firstStateRecord: StateRecord
        get() = records

    override fun prependStateRecord(value: StateRecord) {
        records = value as Record
    }
}
```

Чтение и запись

```
class SimpleMutableState(_value: String) : StateObject {
    private var records = Record(_value)

    var value: String
        get() = records.readable(this).value
        set(value) {
            records.writable(this) {
                this.value = value
            }
        }
...
}
```

Проверяем

```
val name = SimpleMutableState("Mobius")
val snapshot = Snapshot.takeSnapshot()

name.value = "Joker"
println(name.value)

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

Joker
Mobius

Не стейтом едины

```
@Stable
class SnapshotStateList<T> : MutableList<T>, StateObject

@Stable
class SnapshotStateMap<K, V> : MutableMap<K, V>, StateObject

class DerivedSnapshotState<T>(
    private val calculation: () → T,
    override val policy: SnapshotMutationPolicy<T>?
) : StateObject, DerivedState<T>
```

Как происходит чтение и запись в MutableState

StateRecord

```
abstract class StateRecord {
    internal var snapshotId: Int = currentSnapshot().id

    internal var next: StateRecord? = null

    abstract fun assign(value: StateRecord)

    abstract fun create(): StateRecord
}
```

Чтение и запись

```
class SimpleMutableState(_value: String) : StateObject {
    private var records = Record(_value)

    var value: String
        get() = records.readable(this).value
        set(value) {
            records.writable(this) {
                this.value = value
            }
        }
...
}
```

Функция Readable

- Уведомляет подписчиков о том, что стейт прочитан
- Проходится по связанному списку записей и ищет ближайшую валидную с максимальным snapshot ID

Функция Readable

- Уведомляет подписчиков о том, что стейт прочитан
- Проходится по связанному списку записей и ищет ближайшую валидную с максимальным snapshot ID

Валидная запись

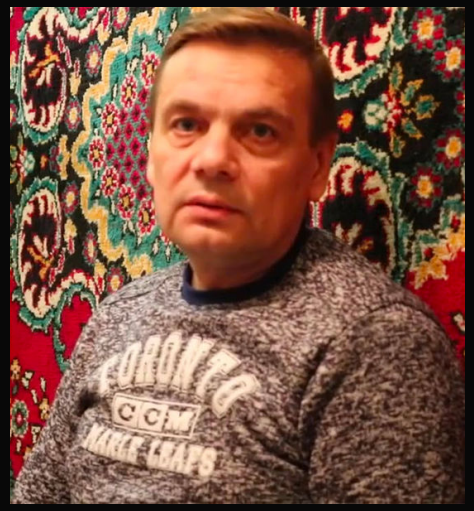
- ID ⩽ current snapshot ID
- not in invalid set

Функция Readable

- Уведомляет подписчиков о том, что стейт прочитан
- Проходится по связанному списку записей и ищет ближайшую валидную с максимальным snapshot ID

Валидная запись

- ID ⩽ current snapshot ID
- not in invalid set



Пример

Snapshot

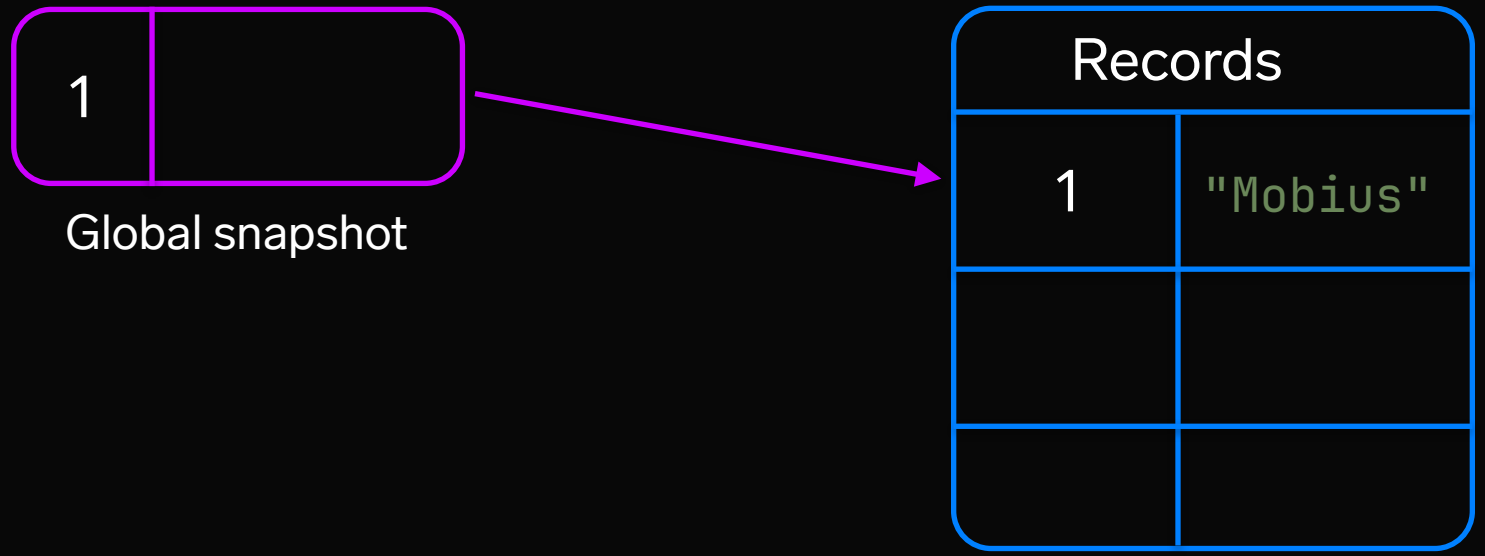
Пример

```
val name = mutableStateOf("Mobius")
val snapshot = Snapshot.takeMutableSnapshot()

snapshot.enter {
    name.value = "Joker"
    println(name.value)
}
snapshot.apply()
```

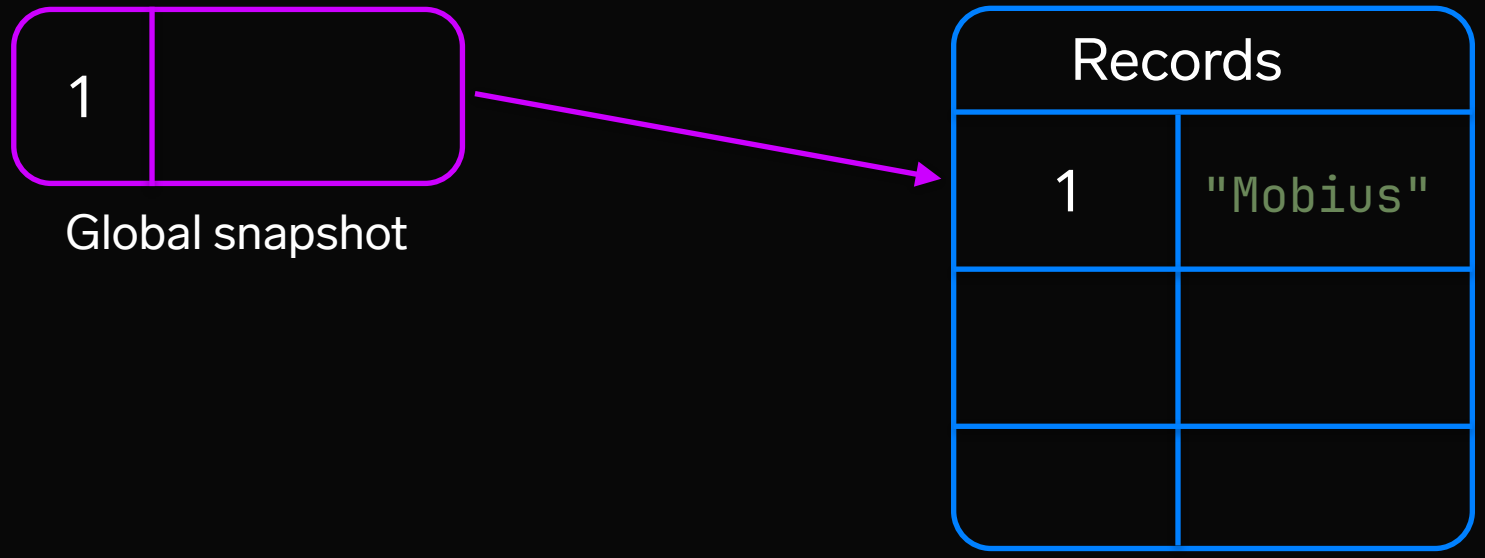
Пример

```
val name = mutableStateOf("Mobius")
```



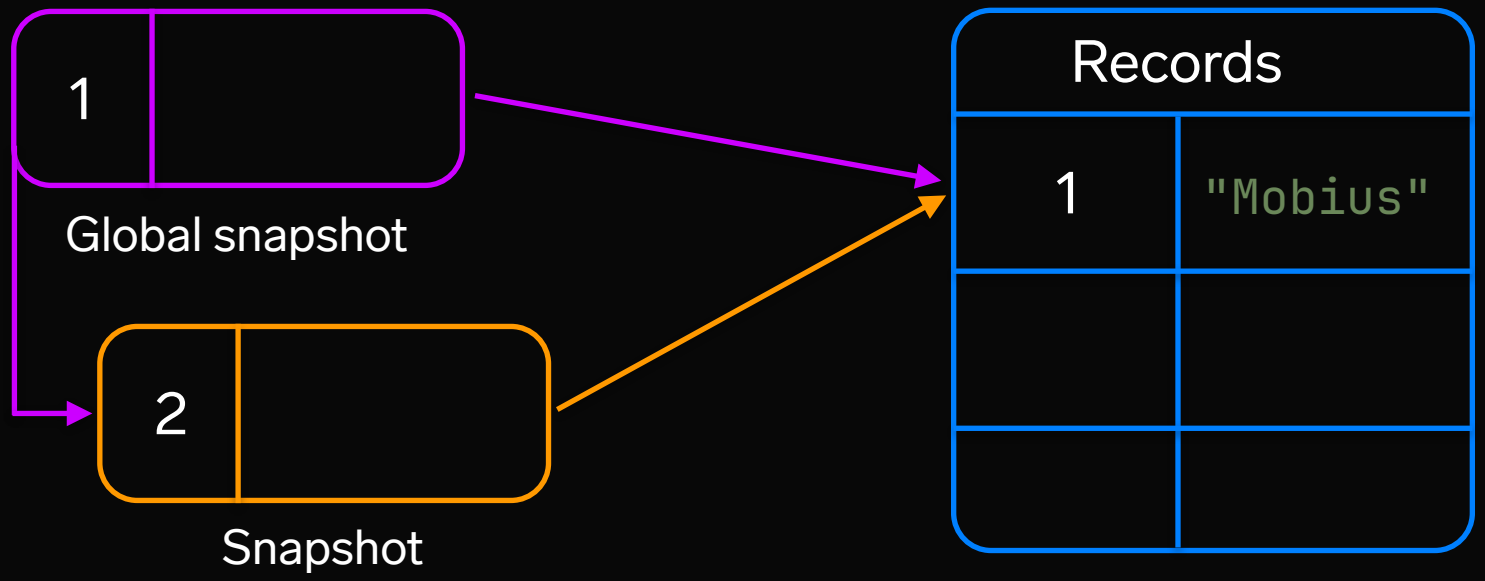
Пример

```
val snapshot = Snapshot.takeMutableSnapshot()
```



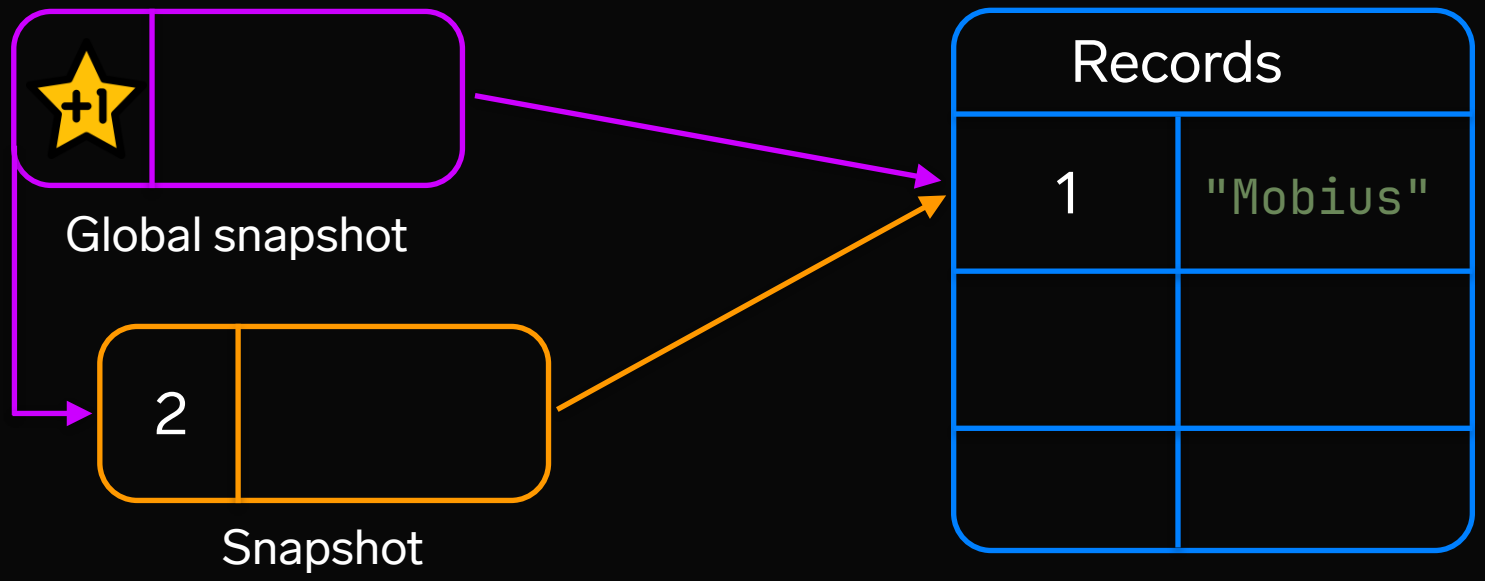
Пример

```
val snapshot = Snapshot.takeMutableSnapshot()
```



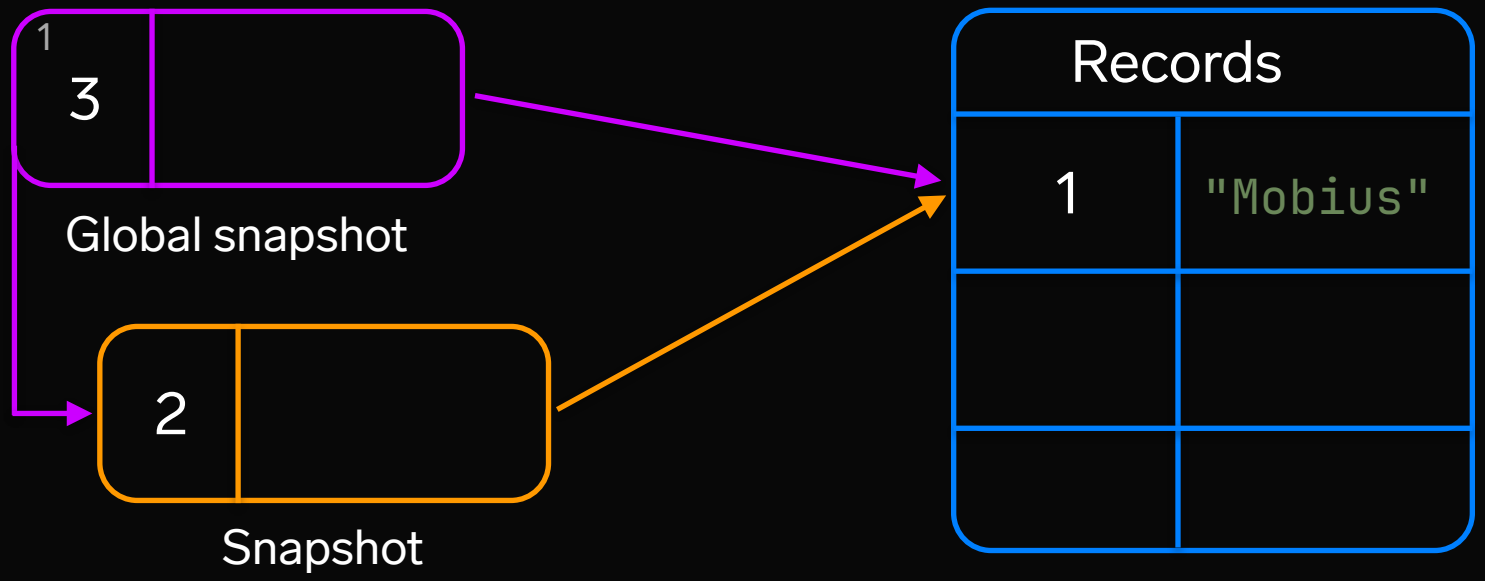
Пример

```
val snapshot = Snapshot.takeMutableSnapshot()
```



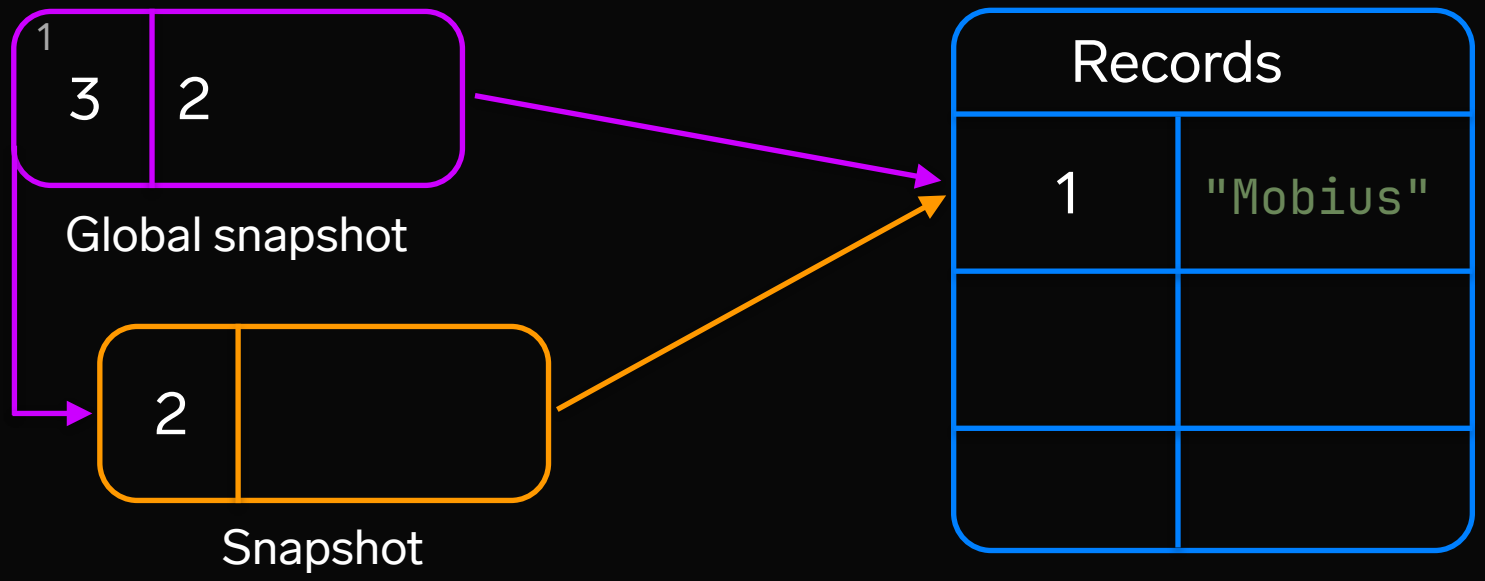
Пример

```
val snapshot = Snapshot.takeMutableSnapshot()
```



Пример

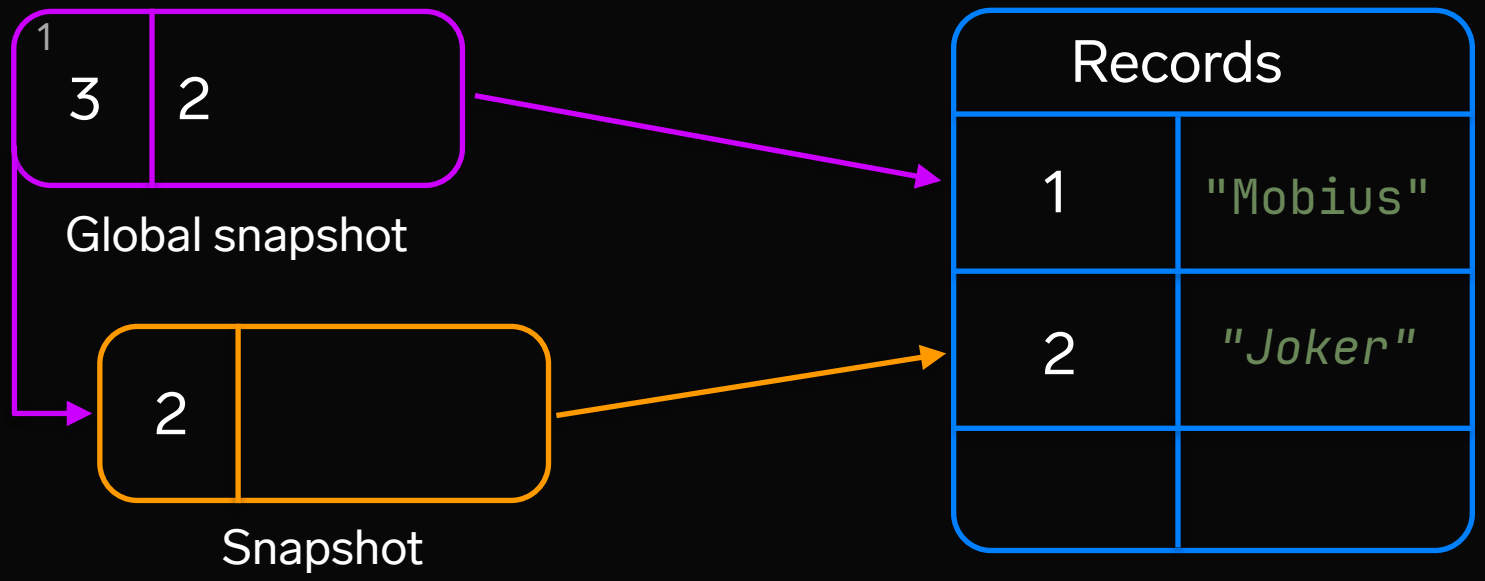
```
val snapshot = Snapshot.takeMutableSnapshot()
```



invalid_set(1..3) = 2

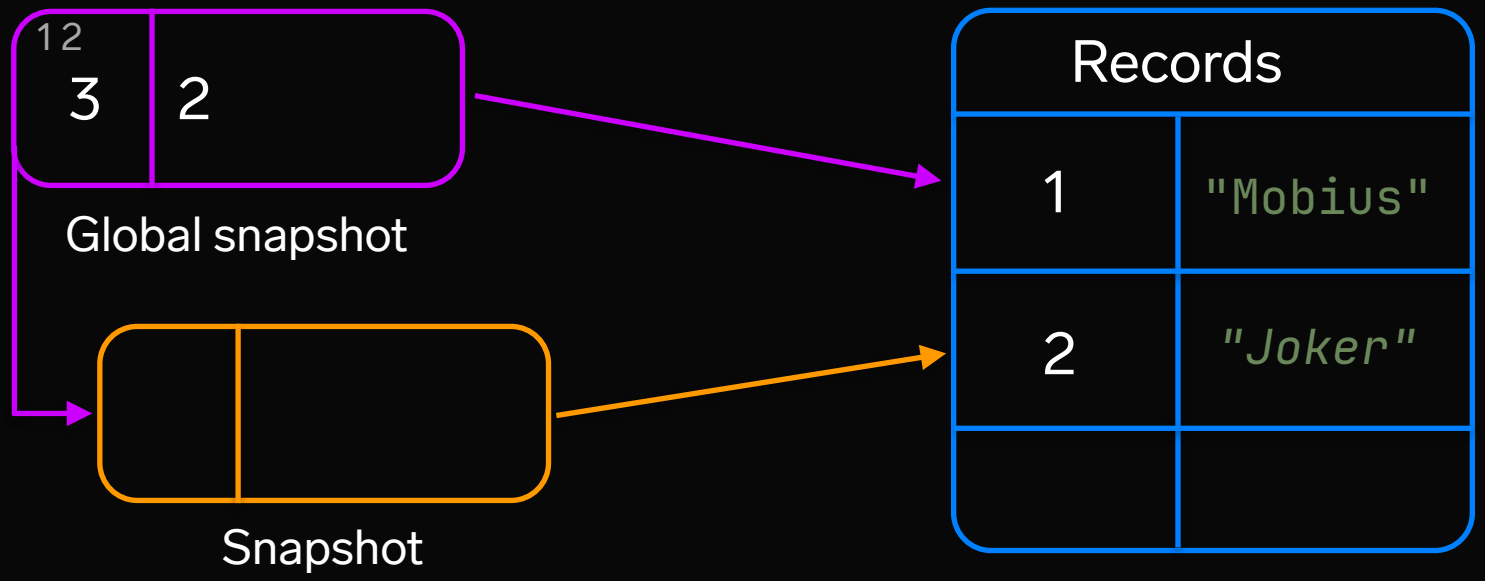
Пример

```
snapshot.enter { name.value = "Joker" }
```



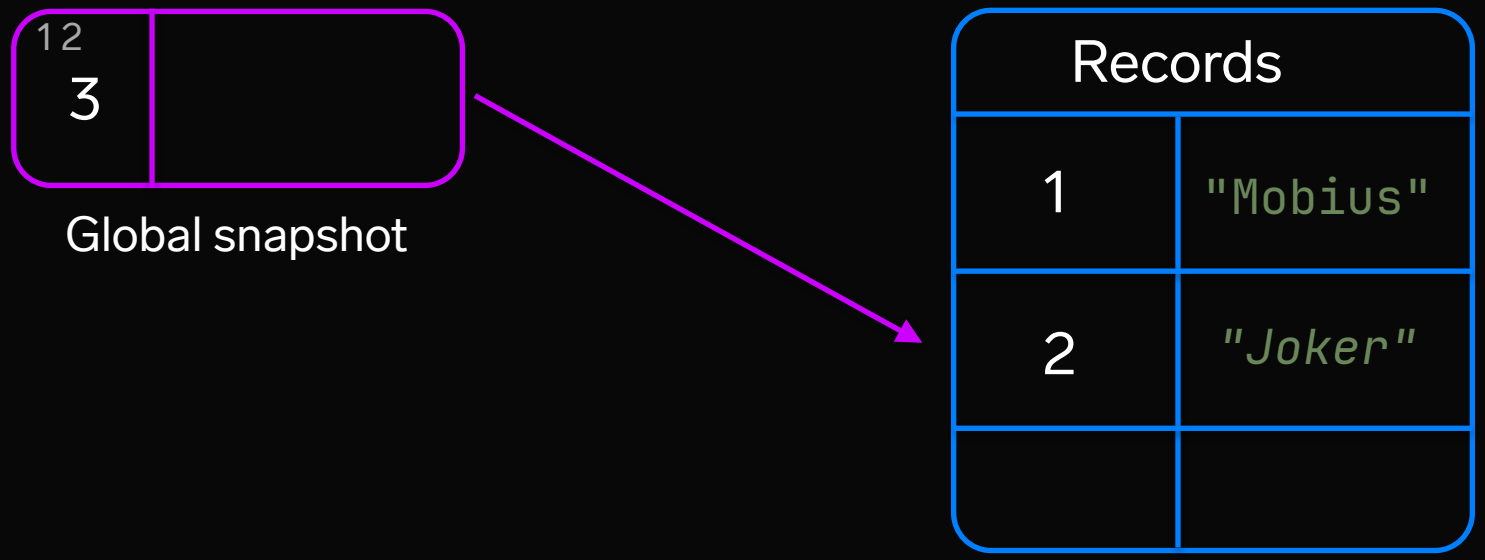
Пример

snapshot.apply()



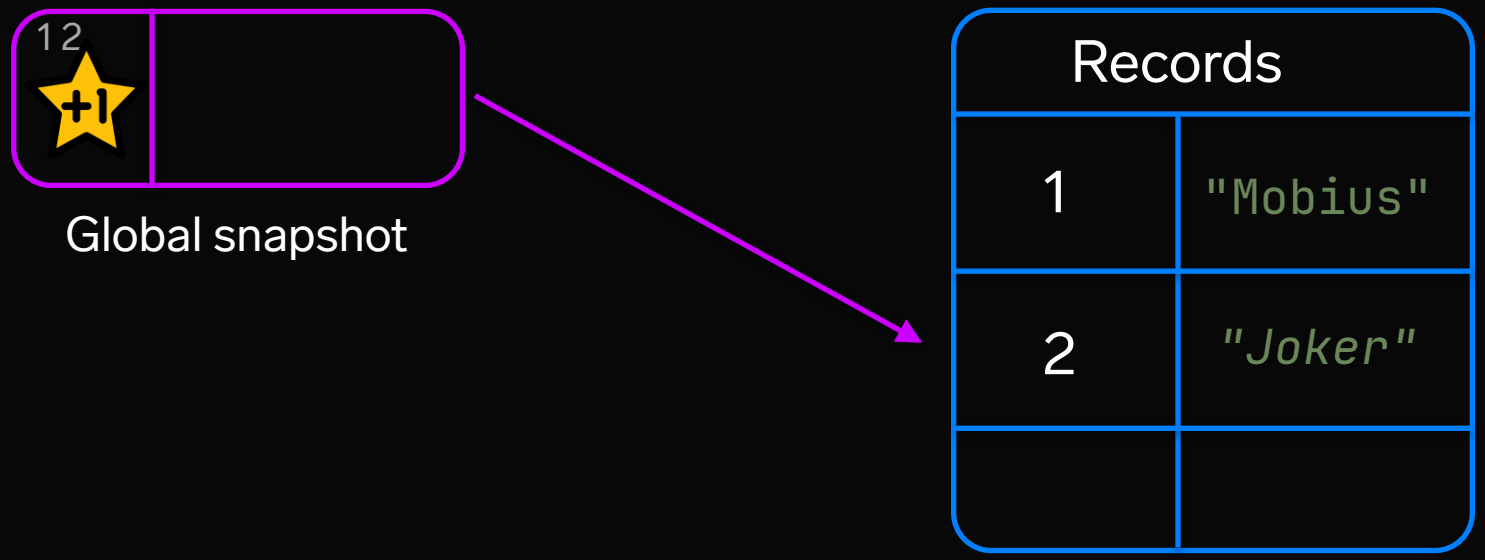
Пример

snapshot.apply()



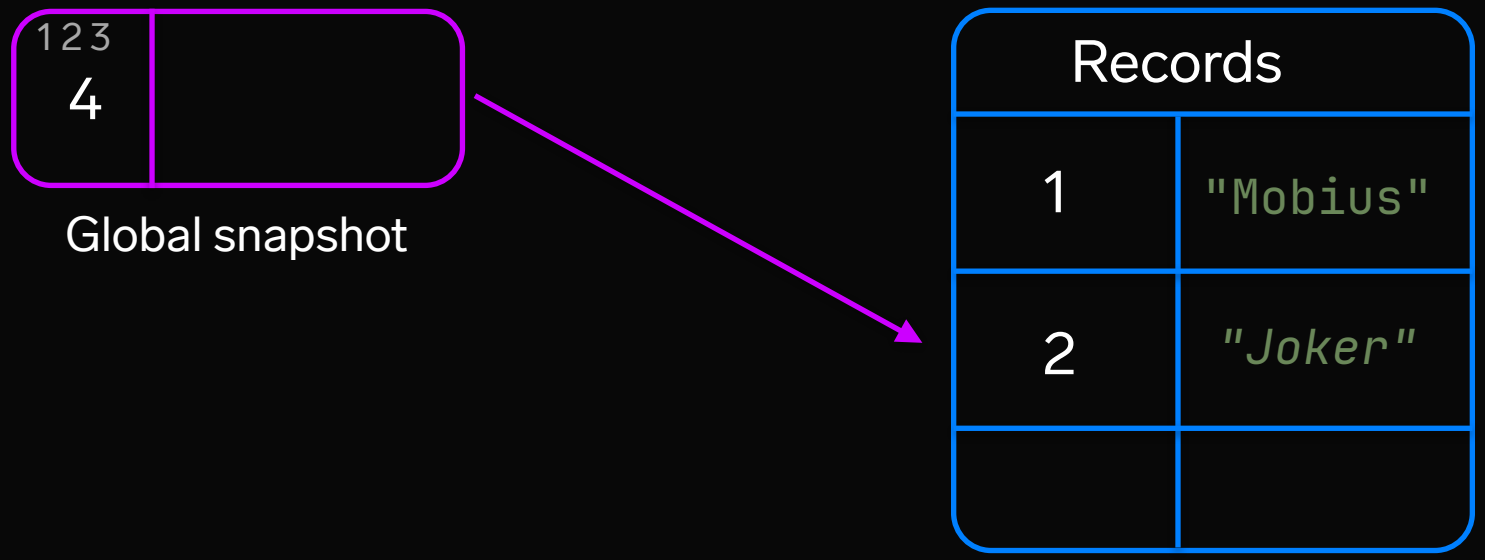
Пример

snapshot.apply()



Пример

snapshot.apply()

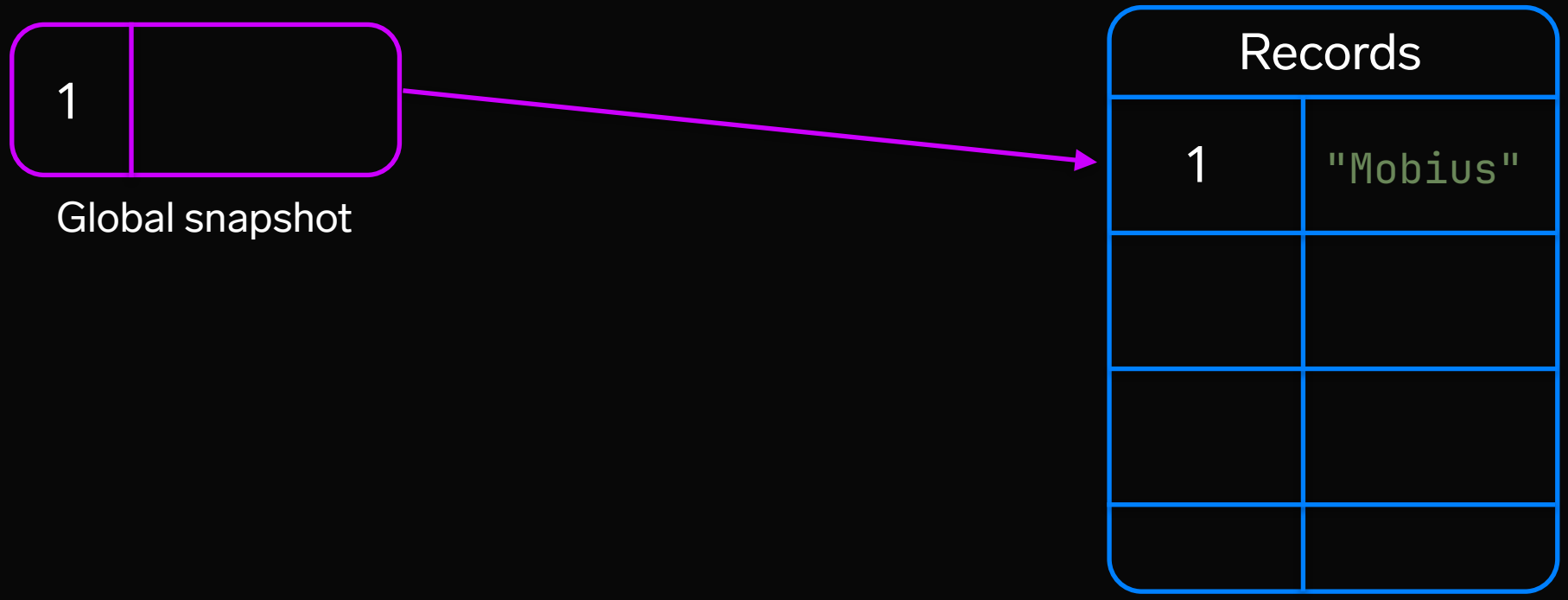


Пример с вложенными снапшотами

```
val name = mutableStateOf("Mobius")
val outerSnapshot = Snapshot.takeMutableSnapshot()
outerSnapshot.enter {
    name.value = "Joker"
    val innerSnapshot = Snapshot.takeMutableSnapshot()
    innerSnapshot.enter {
        name.value = "HolyJS"
    }
    innerSnapshot.apply()
}
outerSnapshot.apply()
```

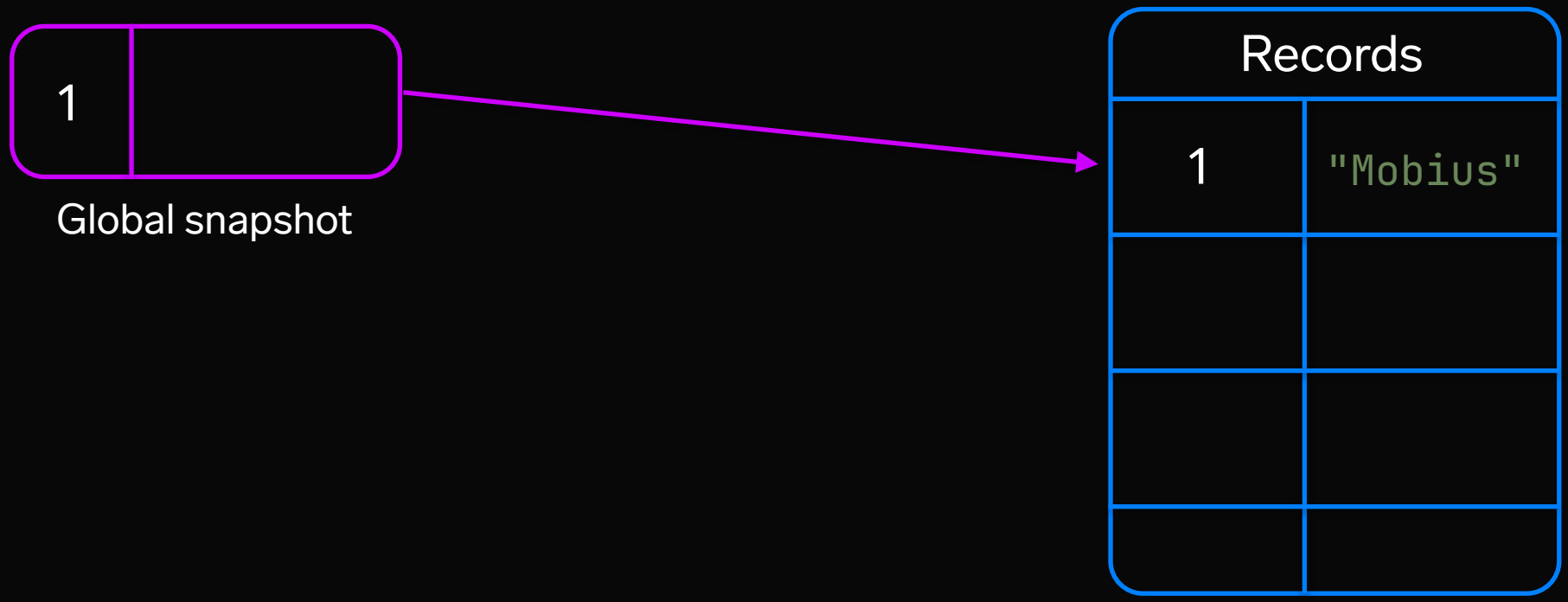
Пример с вложенными снапшотами

```
val name = mutableStateOf("Mobius")
```



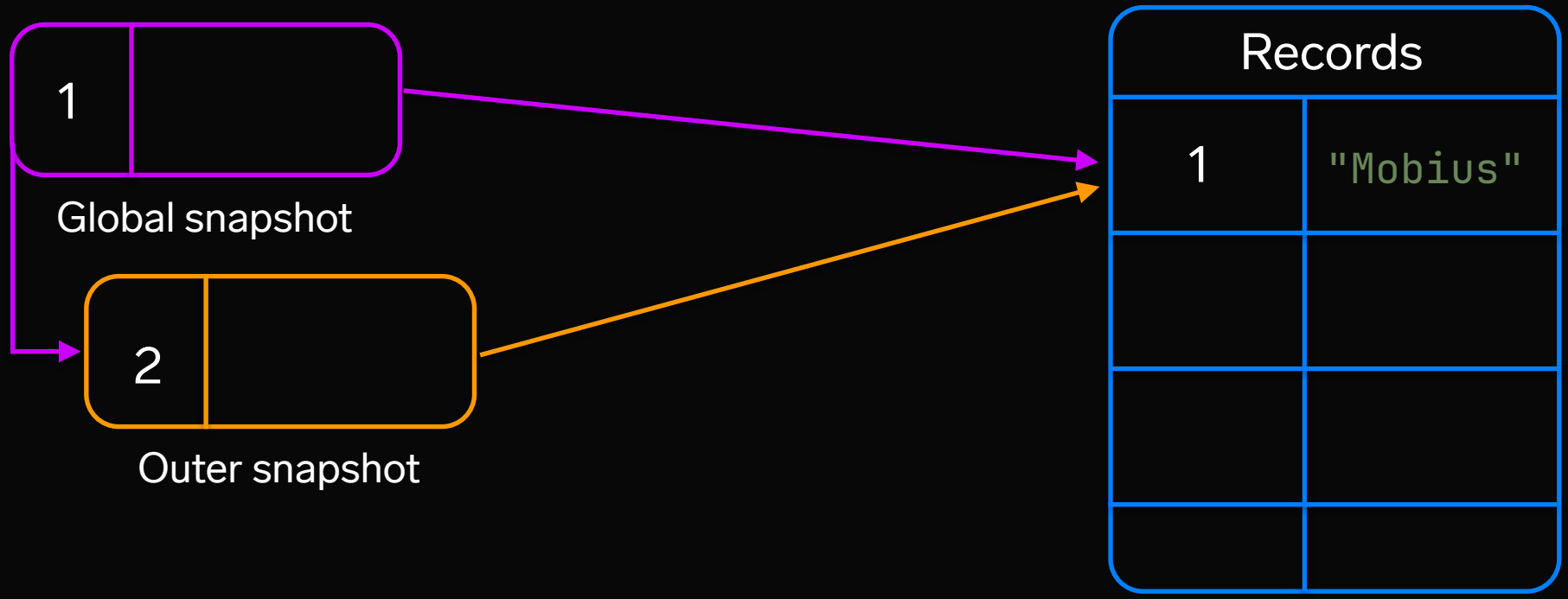
Пример с вложенными снапшотами

```
val outerSnapshot = Snapshot.takeMutableSnapshot()
```



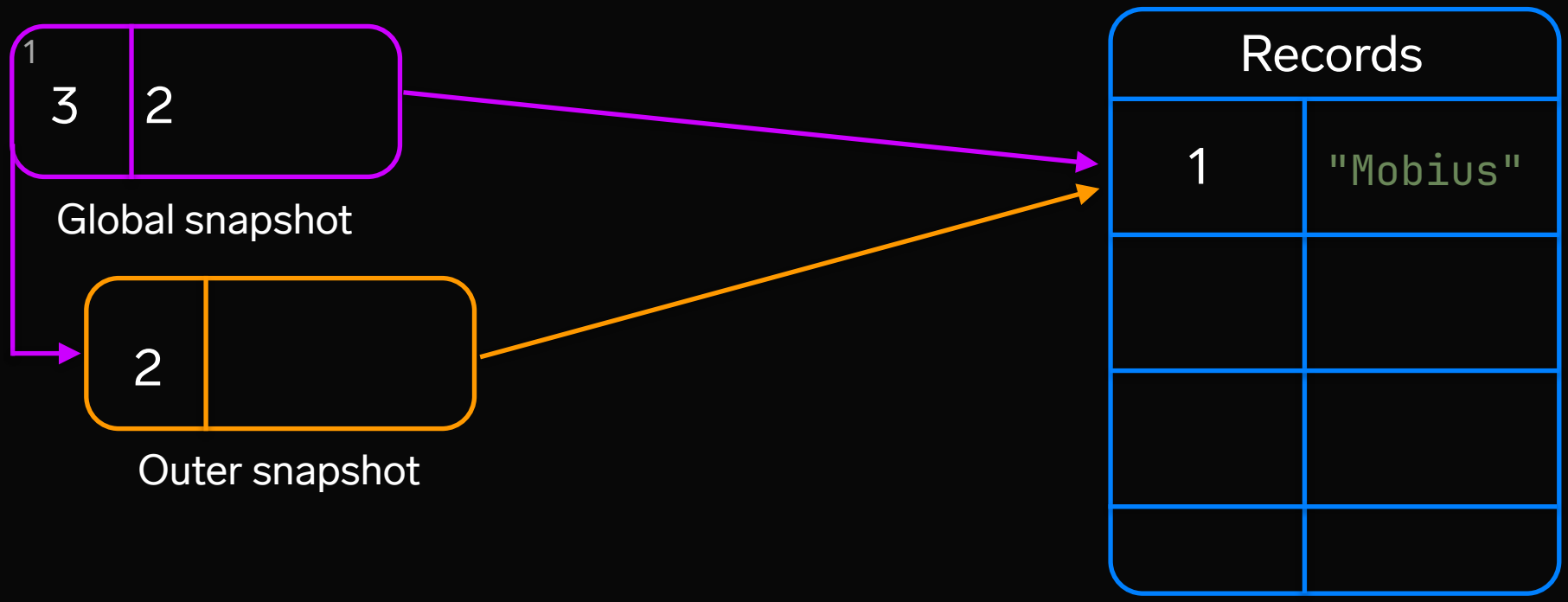
Пример с вложенными снапшотами

```
val outerSnapshot = Snapshot.takeMutableSnapshot()
```



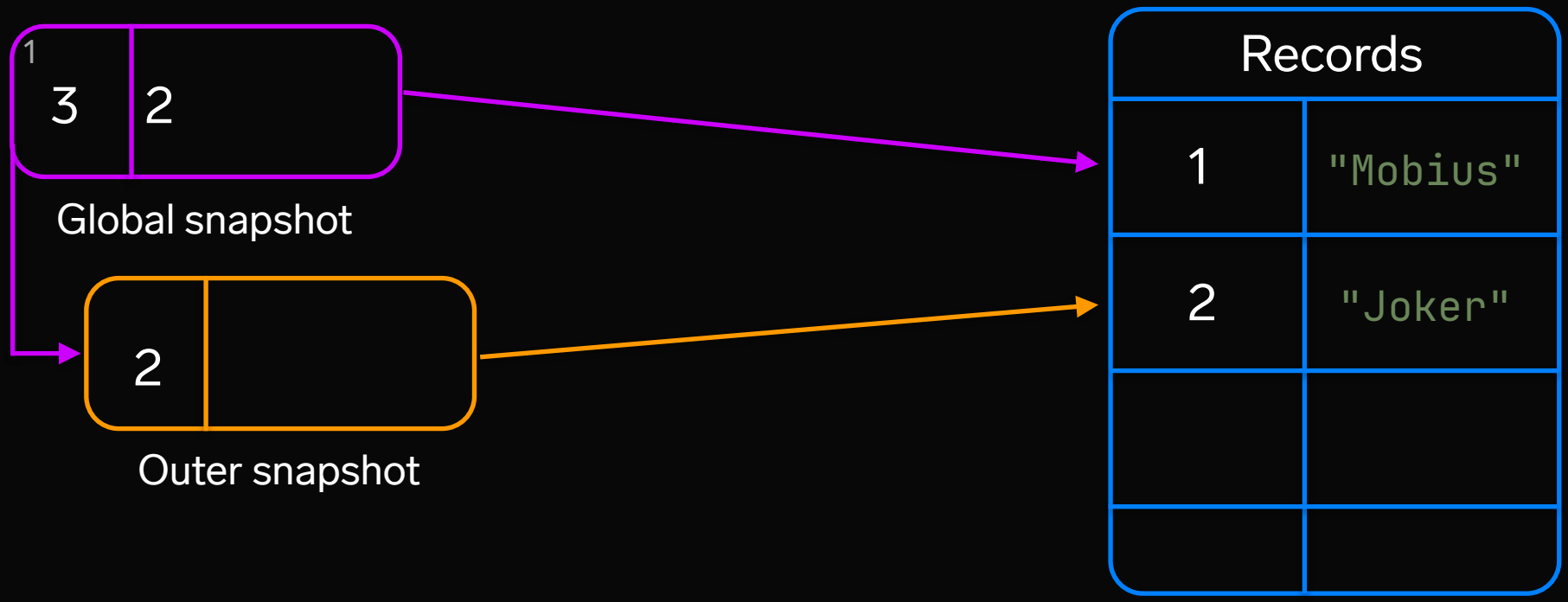
Пример с вложенными снапшотами

```
val outerSnapshot = Snapshot.takeMutableSnapshot()
```



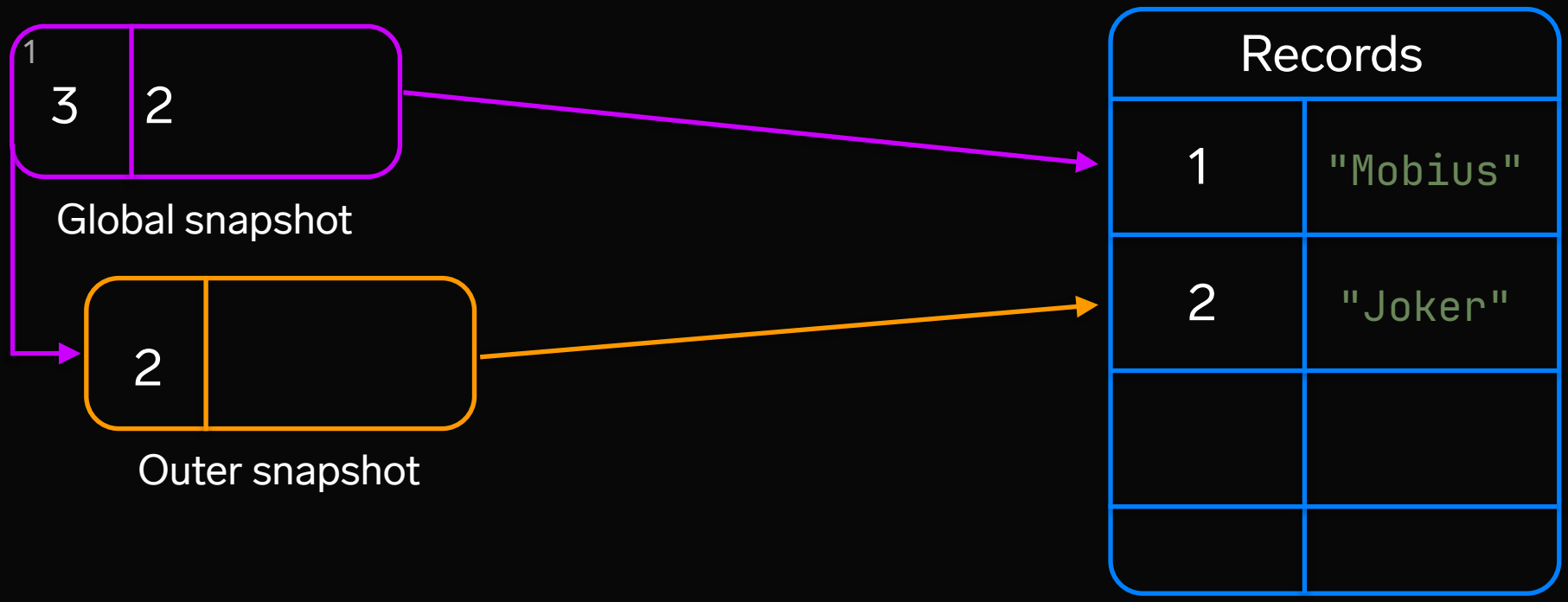
Пример с вложенными снапшотами

```
outerSnapshot.enter { name.value = "Joker" }
```



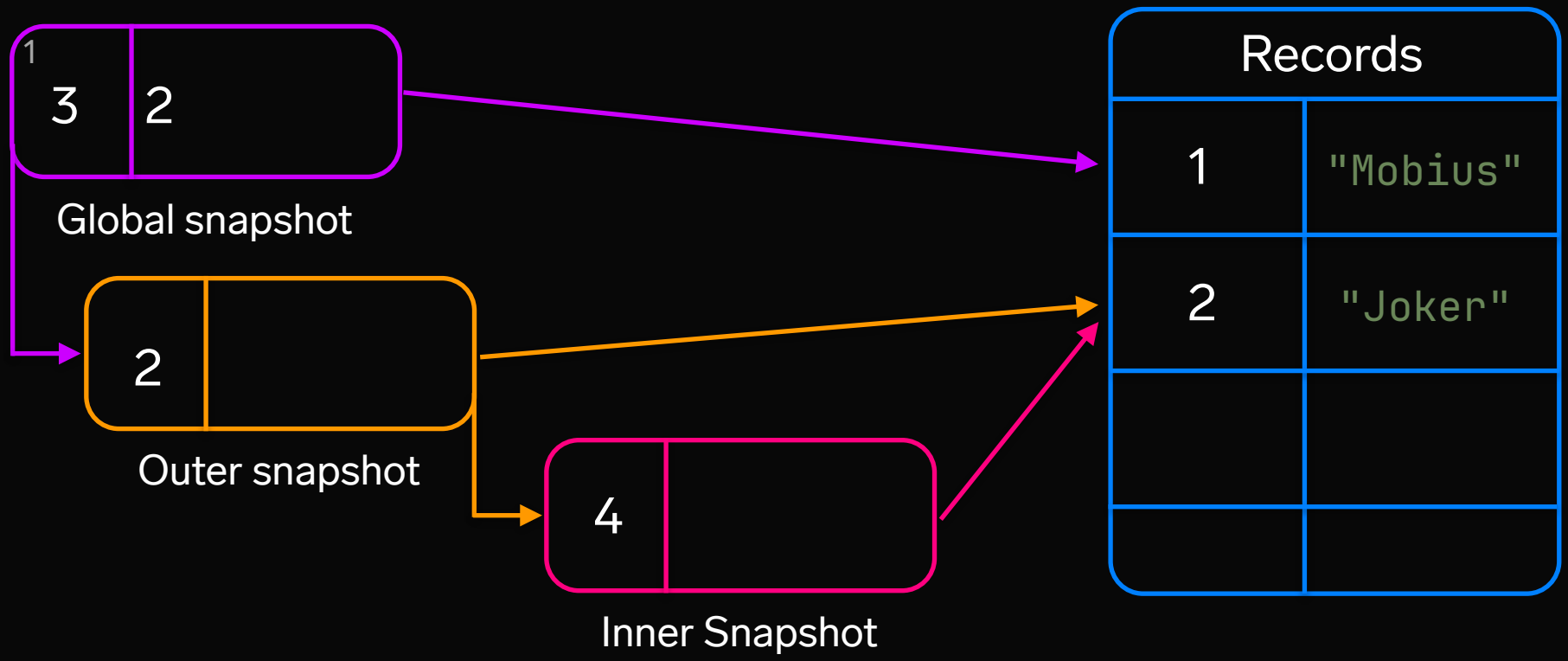
Пример с вложенными снапшотами

```
val innerSnapshot = Snapshot.takeMutableSnapshot()
```



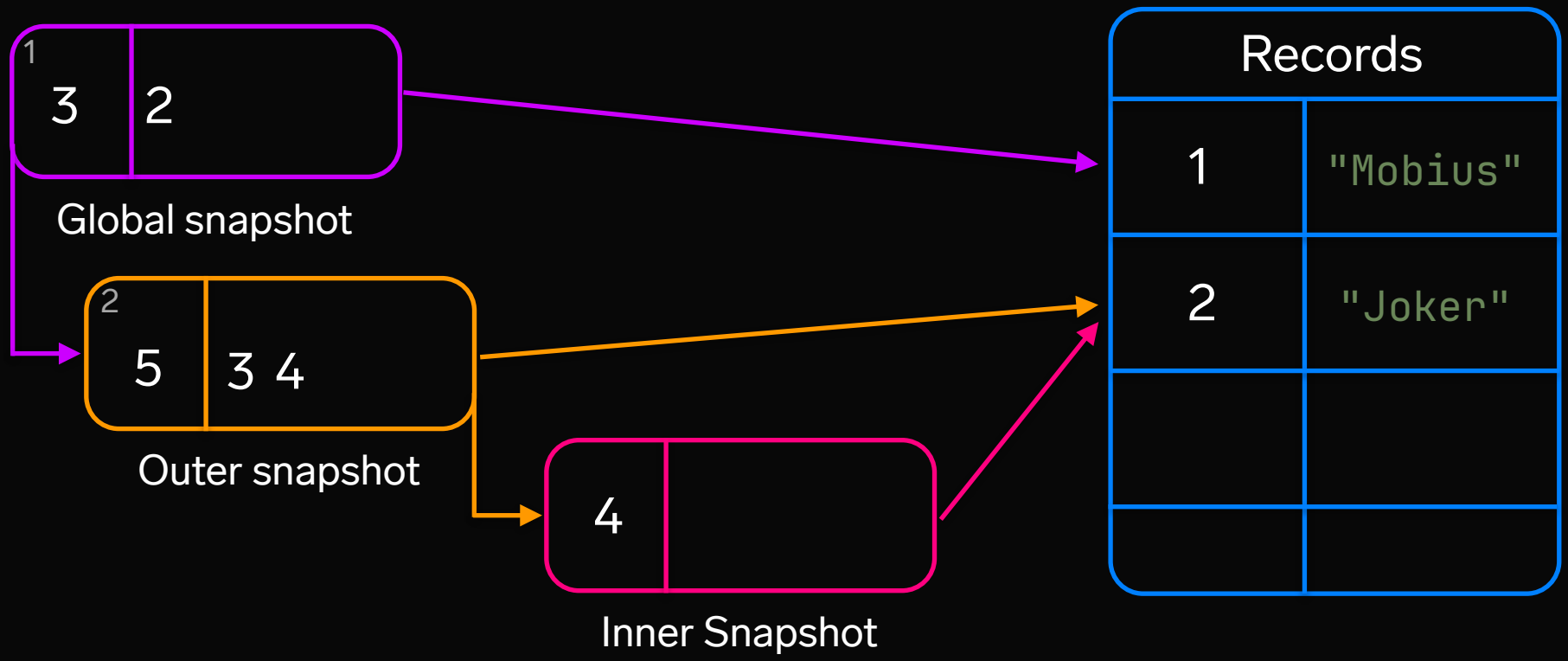
Пример с вложенными снапшотами

```
val innerSnapshot = Snapshot.takeMutableSnapshot()
```



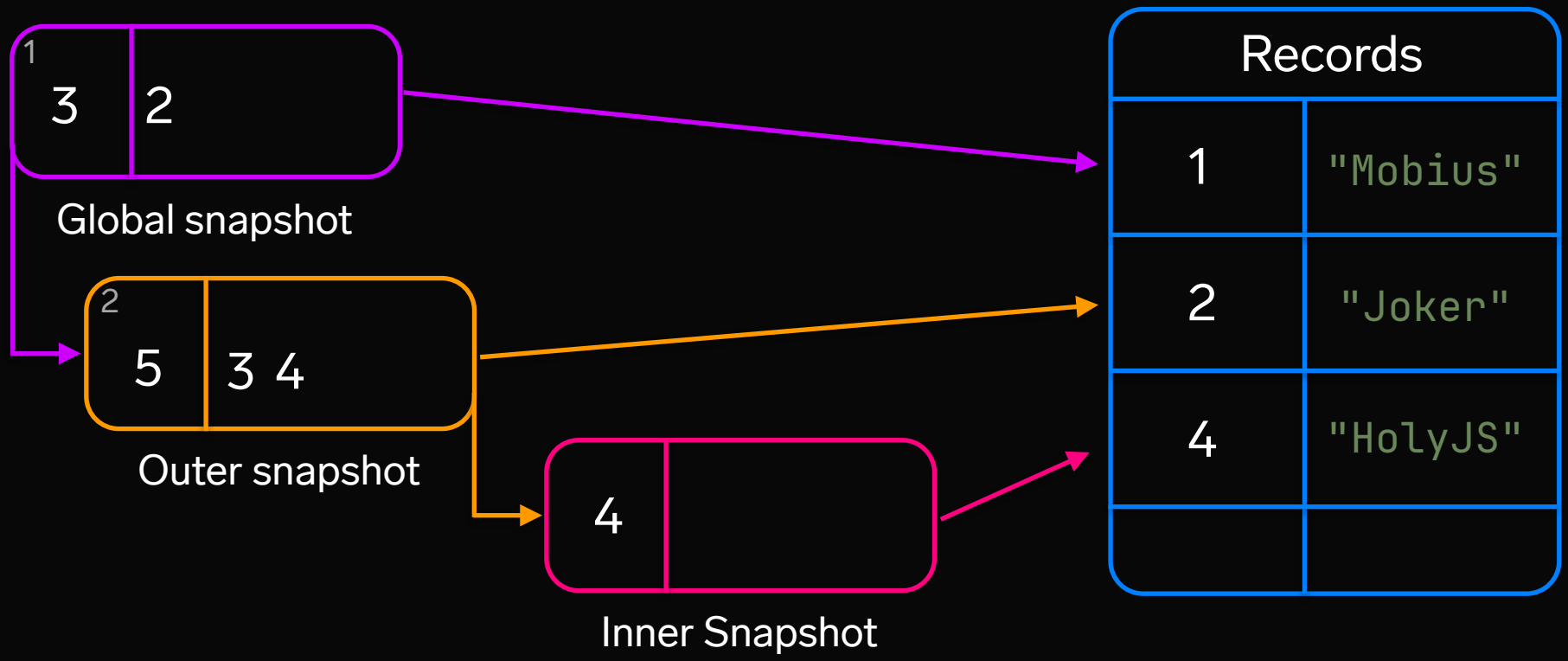
Пример с вложенными снапшотами

```
val innerSnapshot = Snapshot.takeMutableSnapshot()
```



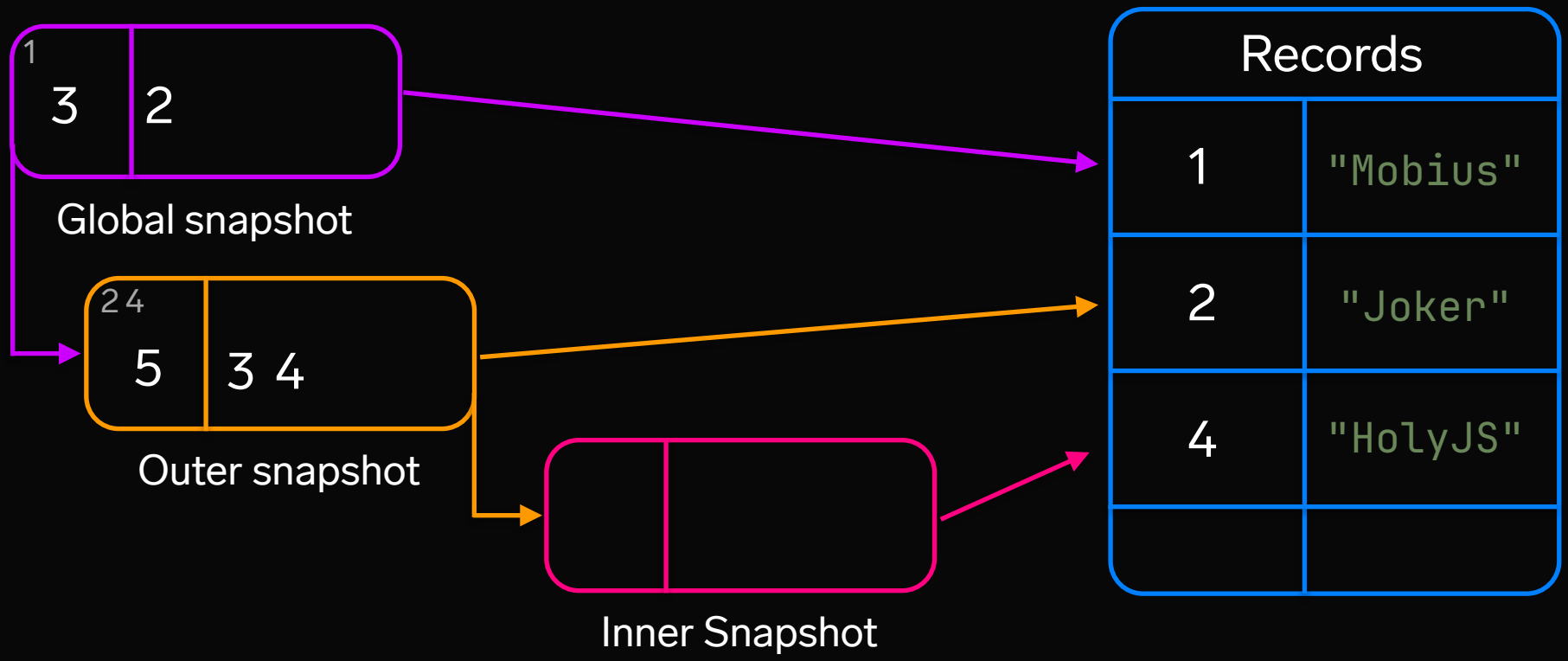
Пример с вложенными снапшотами

```
innerSnapshot.enter { name.value = "HolyJS" }
```



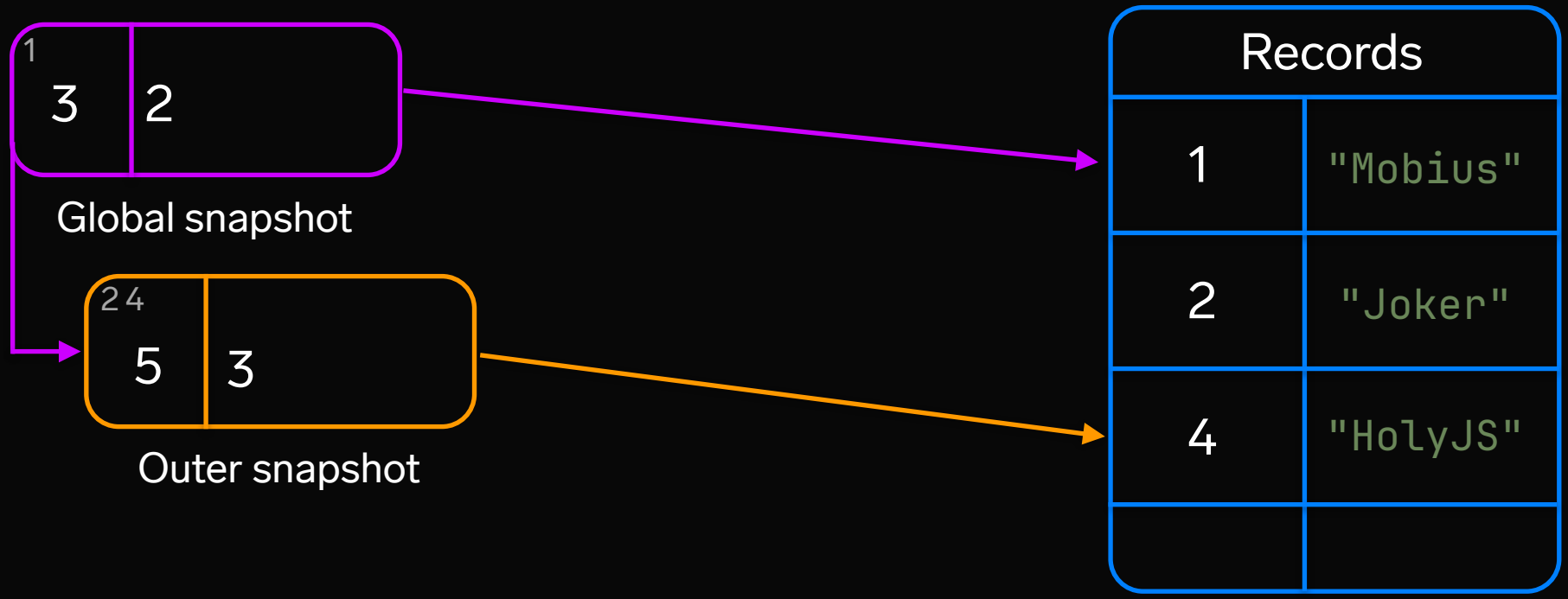
Пример с вложенными снапшотами

```
innerSnapshot.apply()
```



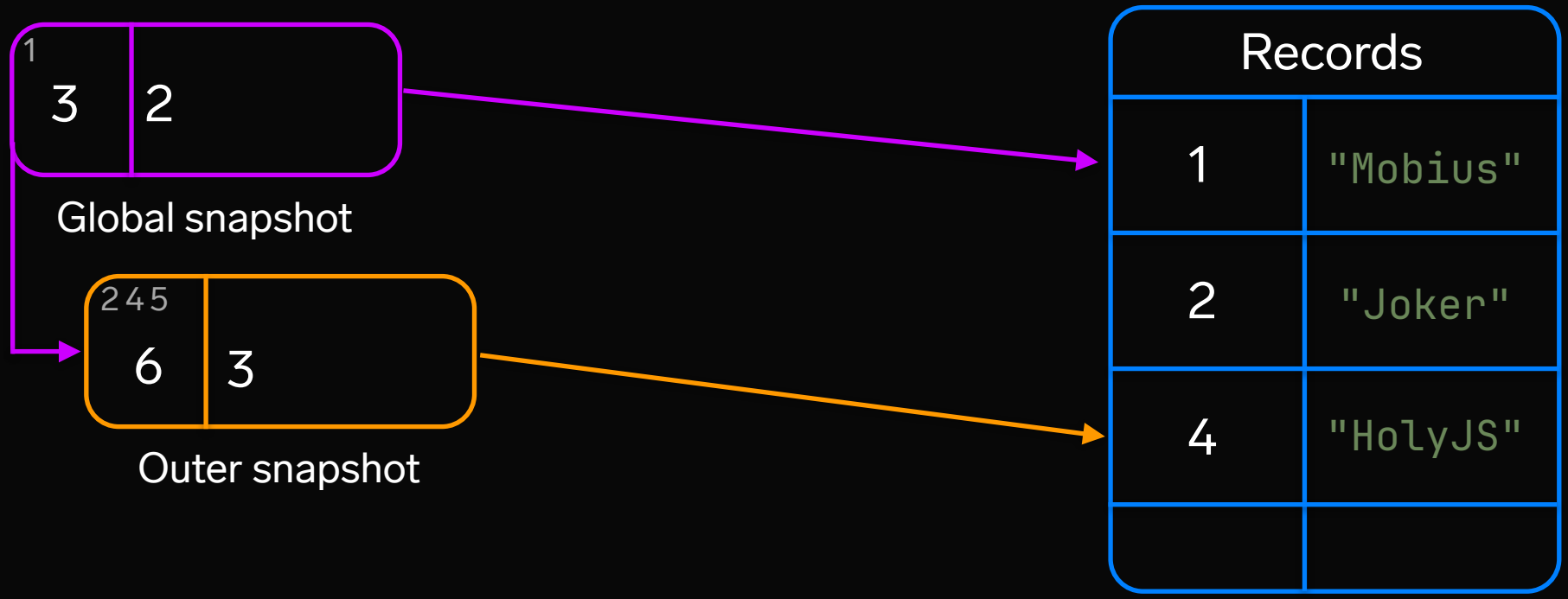
Пример с вложенными снапшотами

```
innerSnapshot.apply()
```



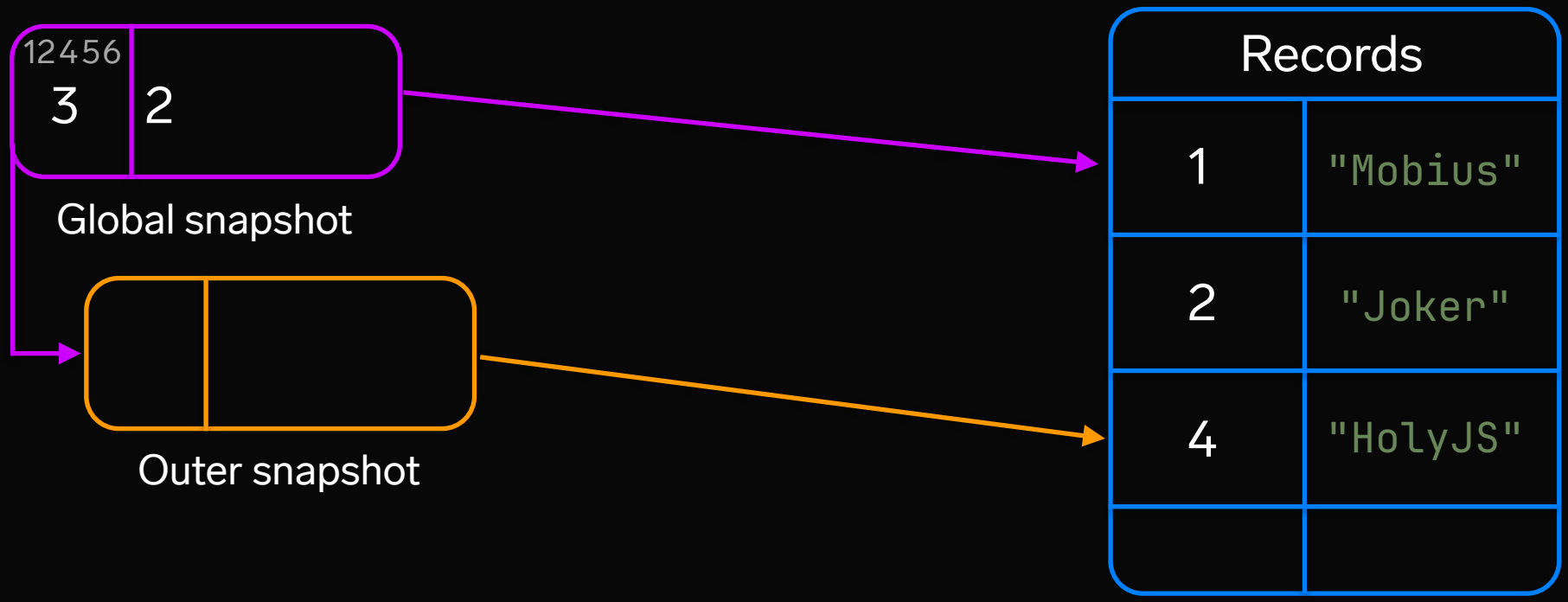
Пример с вложенными снапшотами

```
innerSnapshot.apply()
```



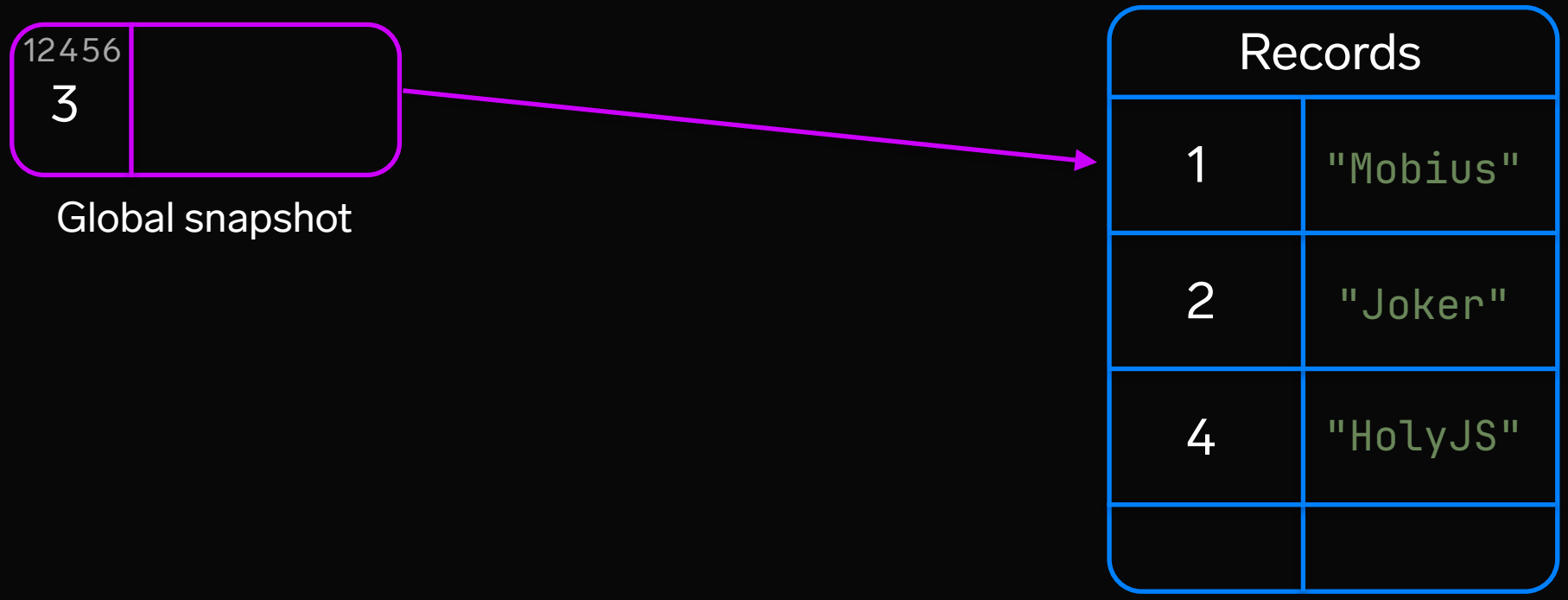
Пример с вложенными снапшотами

```
outerSnapshot.apply()
```



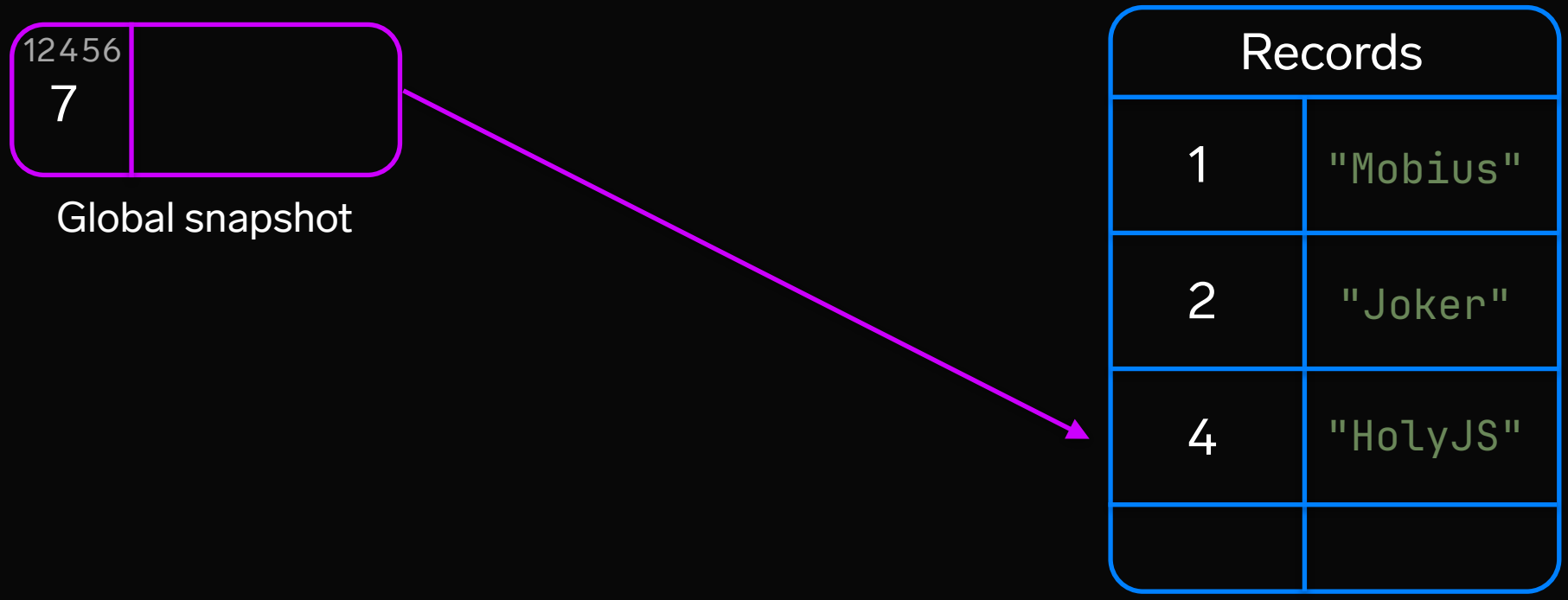
Пример с вложенными снапшотами

```
outerSnapshot.apply()
```

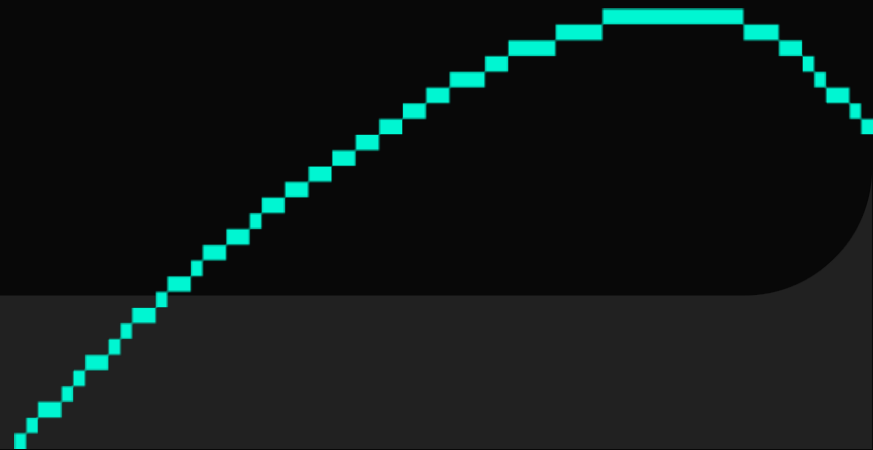


Пример с вложенными снапшотами

```
outerSnapshot.apply()
```



Разбираемся в магии рекомпозиции в Compose



Composer — контекст для Composable функции

Recomposer — планировщик рекомпозиций

Composition — менеджер композиции

Создание композиции

```
fun setContent(content: @Composable () → Unit) {
    shouldCreateCompositionOnAttachedToWindow = true
    this.content.value = content
    if (isAttachedToWindow) {
        createComposition()
    }
}
```

Создание Recomposer

```
private fun ensureCompositionCreated() {
    if (composition == null) {
        try {
            creatingComposition = true
            composition = setContent(resolveParentCompositionContext()) {
                Content()
            }
        } finally {
            creatingComposition = false
        }
    }
}
```

Начальная композиция

```
override fun setContent(content: @Composable () → Unit) {
    check(!disposed) { "The composition is disposed" }
    this.composable = content
    parent.composeInitial(this, composable)
}
```

Начальная композиция

```
internal override fun composeInitial(
    composition: ControlledComposition,
    content: @Composable () → Unit
) {
    val composerWasComposing = composition.isComposing
    try {
        composing(composition, null) {
            composition.composeContent(content)
        }
    } catch (e: Exception) {
        processCompositionError(e, composition, recoverable = true)
        return
    }
...
```

Создание снапшота

```
private inline fun <T> composing(
    composition: ControlledComposition,
    modifiedValues: IdentityArraySet<Any>?,
    block: () -> T
): T {
    val snapshot = Snapshot.takeMutableSnapshot(
        readObserverOf(composition), writeObserverOf(composition, modifiedValues)
    )
    try {
        return snapshot.enter(block)
    } finally {
        applyAndCheck(snapshot)
    }
}
```

Уведомление композиции о записи

```
private fun writeObserverOf(
    composition: ControlledComposition,
    modifiedValues: IdentityArraySet<Any>?
): (Any) → Unit {
    return { value →
        composition.recordWriteOf(value)
        modifiedValues?.add(value)
    }
}
```

Инвалидация recompose скоупа

```
override fun recordWriteOf(value: Any) = synchronized(lock) {
    invalidateScopeOfLocked(value)

    derivedStates.forEachScopeOf(value) {
        invalidateScopeOfLocked(it)
    }
}
```

Рекомпозиция

```
override fun recompose(): Boolean = synchronized(lock) {
    drainPendingModificationsForCompositionLocked()
    guardChanges {
        guardInvalidationsLocked { invalidations →
            composer.recompose(invalidations).also { shouldDrain →
                if (!shouldDrain) drainPendingModificationsLocked()
            }
        }
    }
}
```

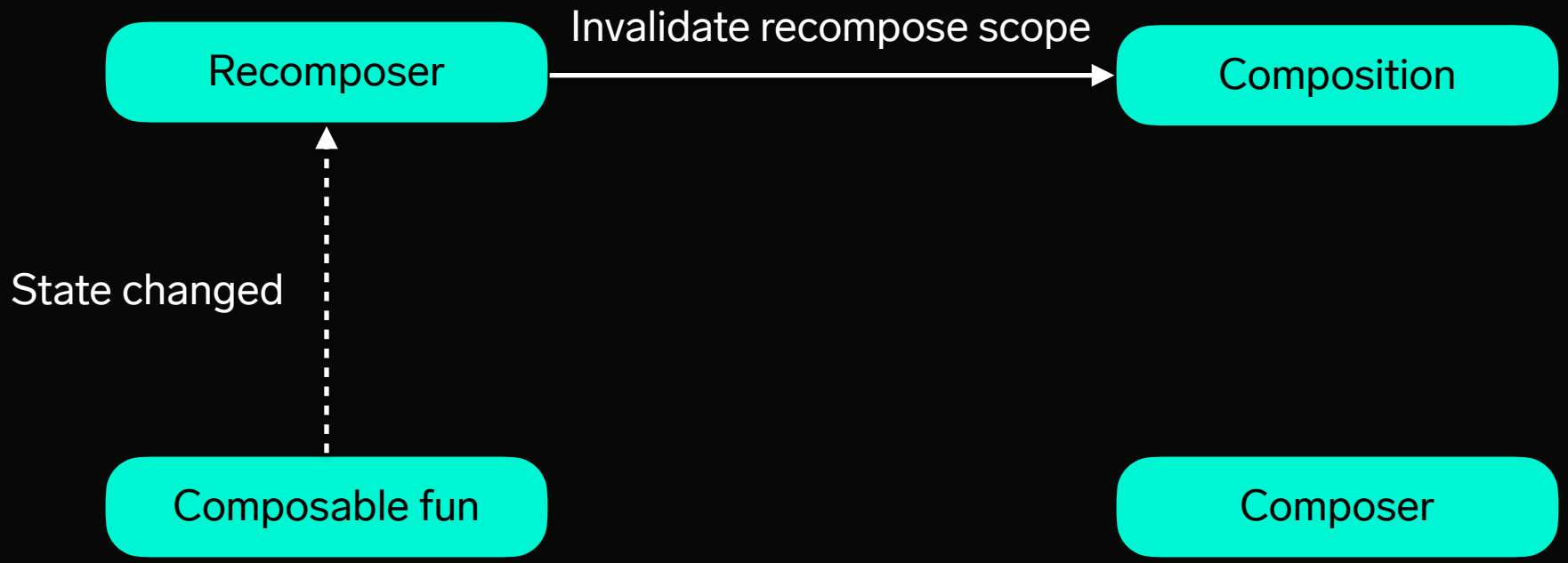
Recomposer

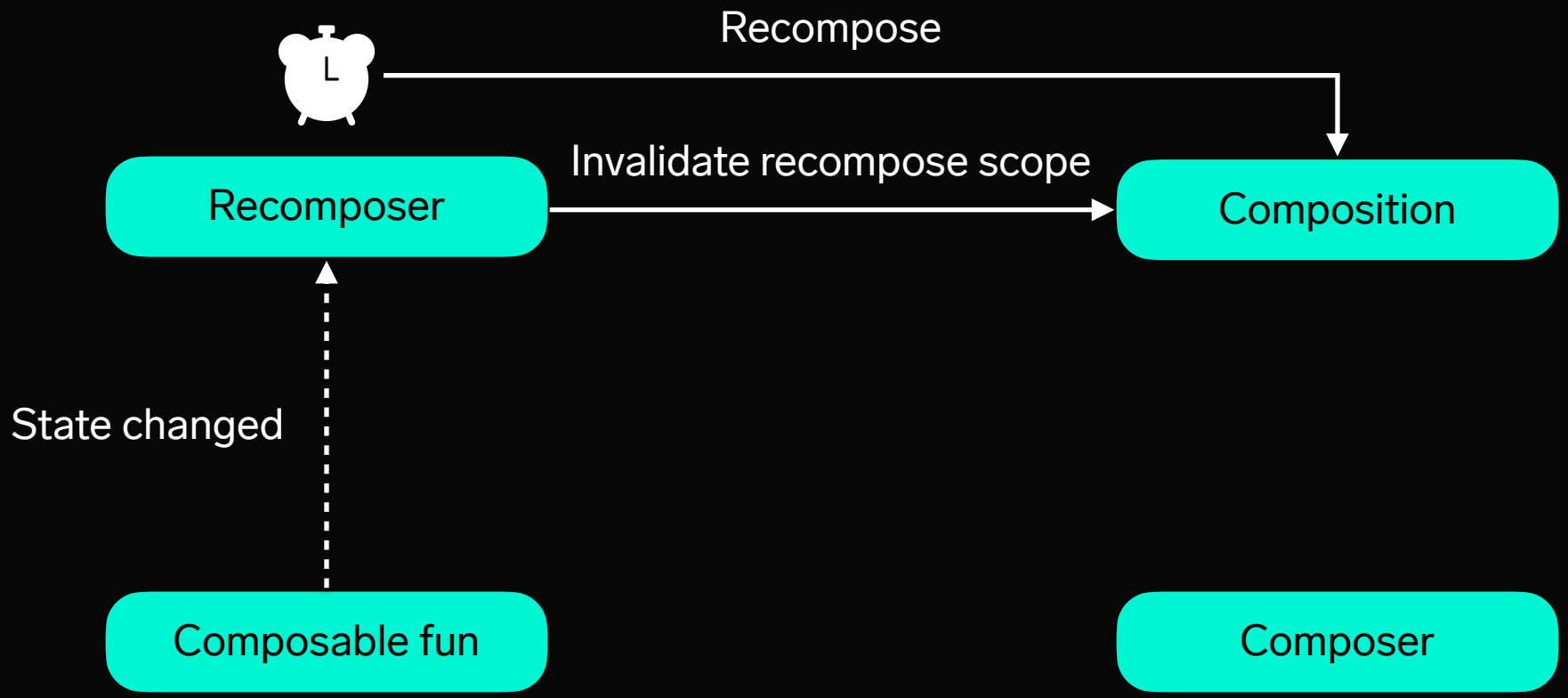
Composition

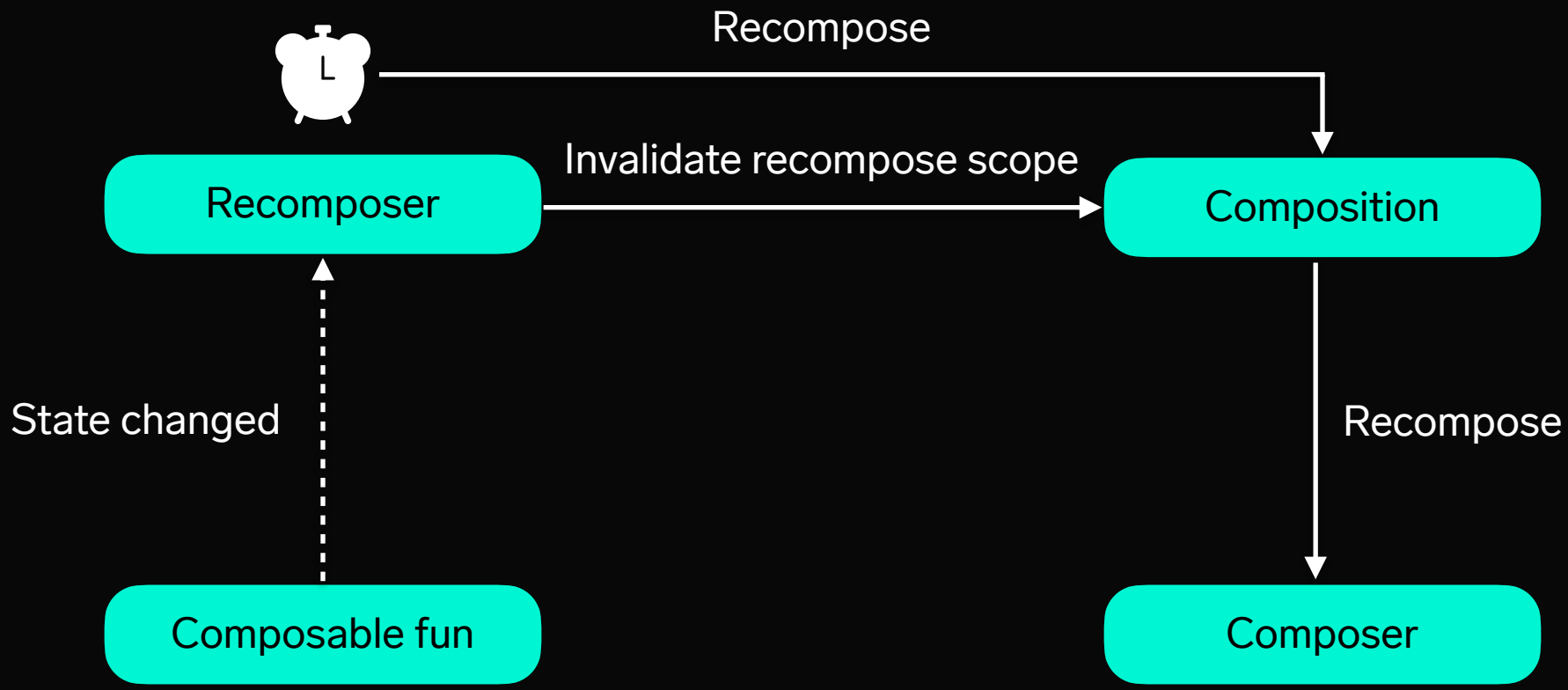
Composable fun

Composer


Recomposer
Composition
State changed
Composable fun
Composer


Recomposer
Invalidate recompose scope
Composition
State changed
Composable fun
Composer


Recompose
Invalidate recompose scope
Recomposer
Composition
State changed
Composable fun
Composer


Recompose
Invalidate recompose scope
Recomposer
Composition
State changed
Recompose
Composable fun
Composer

Выводы

Стейт использует механзим снапшотов

Выводы

Стейт использует механзим снапшотов

Снапшоты помогают достичь атомарности, согласованности и изолированности

Выводы

Стейт использует механзим снапшотов

Снапшоты помогают достичь атомарности, согласованности и изолированности

Знание снапшотов пригодится при создании кастомных типов

Выводы

Стейт использует механзим снапшотов

Снапшоты помогают достичь атомарности, согласованности и изолированности

Знание снапшотов пригодится при создании кастомных типов

Никакой магии в рекомпозиции нет, это просто сложно

Полезные ссылки

- Книга Compose Internals
- Серия статей про State в Compose
- Доклад Opening the shutter on snapshots
- Доклад Using Compose Runtime to create a client library


Посмотрел подкапотный
доклад по Compose


Дамы и господа
Мозг больше не работает

Вопросы?

Контакты

ajiekcx

kotlin_adept

Контур

Алексей Панов

Ведущий инженер-программист