

Наглядный пример, когда не стоит следовать **best practices for coroutines** от Google

Кому полезен доклад

Кому полезен доклад



Есть ли ограничения у советов Google?

Кому полезен доклад



Есть ли ограничения у советов Google?



Какой диспатчер выбрать?

Кому полезен доклад



Есть ли ограничения у советов Google?



Какой диспатчер выбрать?



Ну что повлияет неверный выбор?

Кому полезен доклад

- Есть ли ограничения у советов Google?
- Какой диспатчер выбрать?
- Ну что повлияет неверный выбор?
- Наглядные аргументы в споре

План

План



Реальный пример, когда Best practice вредит

План

- Реальный пример, когда Best practice вредит
- Как работает и чем занят Main Dispatcher

План

- Реальный пример, когда Best practice вредит
- Как работает и чем занят Main Dispatcher
- Принципы работы диспетчеров

План

- Реальный пример, когда Best practice вредит
- Как работает и чем занят Main Dispatcher
- Принципы работы диспетчеров
- Как быть и что делать?

Александр Таганов

Ведущий разработчик,
Тимлид



**Что не так с Google best
practices**

Базовый принцип

Suspend functions should be safe to call from the main thread

Suspend functions should be main-safe, meaning they're safe to call from the main thread. If a class is doing long-running blocking operations in a coroutine, it's in charge of moving the execution off the main thread using `withContext`. This applies to all classes in your app, regardless of the part of the architecture the class is in.

Пример репозитория

```
class NewsRepository(private val ioDispatcher: CoroutineDispatcher) {  
  
    // As this operation is manually retrieving the news from the server  
    // using a blocking HttpURLConnection, it needs to move the execution  
    // to an IO dispatcher to make it main-safe  
    suspend fun fetchLatestNews(): List<Article> {  
        withContext(ioDispatcher) { /* ... implementation ... */ }  
    }  
}
```

Пример репозитория

```
class NewsRepository(private val ioDispatcher: CoroutineDispatcher) {  
  
    // As this operation is manually retrieving the news from the server  
    // using a blocking HttpURLConnection, it needs to move the execution  
    // to an IO dispatcher to make it main-safe  
    suspend fun fetchLatestNews(): List<Article> {  
        withContext(ioDispatcher) { /* ... implementation ... */ }  
    }  
}
```

Пример юзкейса

```
// This use case fetches the latest news and the associated author.
class GetLatestNewsWithAuthorsUseCase(
    private val newsRepository: NewsRepository,
    private val authorsRepository: AuthorsRepository
) {
    // This method doesn't need to worry about moving the execution of the
    // coroutine to a different thread as newsRepository is main-safe.
    // The work done in the coroutine is lightweight as it only creates
    // a list and add elements to it
    suspend operator fun invoke(): List<ArticleWithAuthor> {
        val news = newsRepository.fetchLatestNews()

        val response: List<ArticleWithAuthor> = mutableListOf()
        for (article in news) {
            val author = authorsRepository.getAuthor(article.author)
            response.add(ArticleWithAuthor(article, author))
        }
        return Result.Success(response)
    }
}
```

Пример юзкейса

```
// This use case fetches the latest news and the associated author.
class GetLatestNewsWithAuthorsUseCase(
    private val newsRepository: NewsRepository,
    private val authorsRepository: AuthorsRepository
) {
    // This method doesn't need to worry about moving the execution of the
    // coroutine to a different thread as newsRepository is main-safe.
    // The work done in the coroutine is lightweight as it only creates
    // a list and add elements to it
    suspend operator fun invoke(): List<ArticleWithAuthor> {
        val news = newsRepository.fetchLatestNews()

        val response: List<ArticleWithAuthor> = mutableListOf()
        for (article in news) {
            val author = authorsRepository.getAuthor(article.author)
            response.add(ArticleWithAuthor(article, author))
        }
        return Result.Success(response)
    }
}
```

Старт корутины

```
// DO create coroutines in the ViewModel
class LatestNewsViewModel(
    private val getLatestNewsWithAuthors: GetLatestNewsWithAuthorsUseCase
) : ViewModel() {

    private val _uiState = MutableStateFlow<LatestNewsUiState>(LatestNewsUiState.Loading)
    val uiState: StateFlow<LatestNewsUiState> = _uiState

    fun loadNews() {
        viewModelScope.launch {
            val latestNewsWithAuthors = getLatestNewsWithAuthors()
            _uiState.value = LatestNewsUiState.Success(latestNewsWithAuthors)
        }
    }
}
```

Старт корутины

```
// DO create coroutines in the ViewModel
class LatestNewsViewModel(
    private val getLatestNewsWithAuthors: GetLatestNewsWithAuthorsUseCase
) : ViewModel() {

    private val _uiState = MutableStateFlow<LatestNewsUiState>(LatestNewsUiState.Loading)
    val uiState: StateFlow<LatestNewsUiState> = _uiState

    fun loadNews() {
        viewModelScope.launch {
            val latestNewsWithAuthors = getLatestNewsWithAuthors()
            _uiState.value = LatestNewsUiState.Success(latestNewsWithAuthors)
        }
    }
}
```

Старт корутины

```
// DO create coroutines in the ViewModel
class LatestNewsViewModel(
    private val getLatestNewsWithAuthors: GetLatestNewsWithAuthorsUseCase
) : ViewModel() {

    private val _uiState = MutableStateFlow<LatestNewsUiState>(LatestNewsUiState.Loading)
    val uiState: StateFlow<LatestNewsUiState> = _uiState

    fun loadNews() {
        viewModelScope.launch {
            val latestNewsWithAuthors = getLatestNewsWithAuthors()
            _uiState.value = LatestNewsUiState.Success(latestNewsWithAuthors)
        }
    }
}
```

Что может пойти не так?

ViewModel

```
internal fun createViewModelScope(): CloseableCoroutineScope {
    val dispatcher = try {
        // In platforms where `Dispatchers.Main` is not available, Kotlin Multiplatform will throw
        // an exception (the specific exception type may depend on the platform). Since there's no
        // direct functional alternative, we use `EmptyCoroutineContext` to ensure that a coroutine
        // launched within this scope will run in the same context as the caller.
        Dispatchers.Main.immediate
    } catch (_: NotImplementedError) {
        // In Native environments where `Dispatchers.Main` might not exist (e.g., Linux):
        EmptyCoroutineContext
    } catch (_: IllegalStateException) {
        // In JVM Desktop environments where `Dispatchers.Main` might not exist (e.g., Swing):
        EmptyCoroutineContext
    }
    return CloseableCoroutineScope(coroutineContext = dispatcher + SupervisorJob())
}
```

ViewModel

```
internal fun createViewModelScope(): CloseableCoroutineScope {
    val dispatcher = try {
        // In platforms where `Dispatchers.Main` is not available, Kotlin Multiplatform will throw
        // an exception (the specific exception type may depend on the platform). Since there's no
        // direct functional alternative, we use `EmptyCoroutineContext` to ensure that a coroutine
        // launched within this scope will run in the same context as the caller.
        Dispatchers.Main.immediate
    } catch (_: NotImplementedError) {
        // In Native environments where `Dispatchers.Main` might not exist (e.g., Linux):
        EmptyCoroutineContext
    } catch (_: IllegalStateException) {
        // In JVM Desktop environments where `Dispatchers.Main` might not exist (e.g., Swing):
        EmptyCoroutineContext
    }
    return CloseableCoroutineScope(coroutineContext = dispatcher + SupervisorJob())
}
```

Схема работы

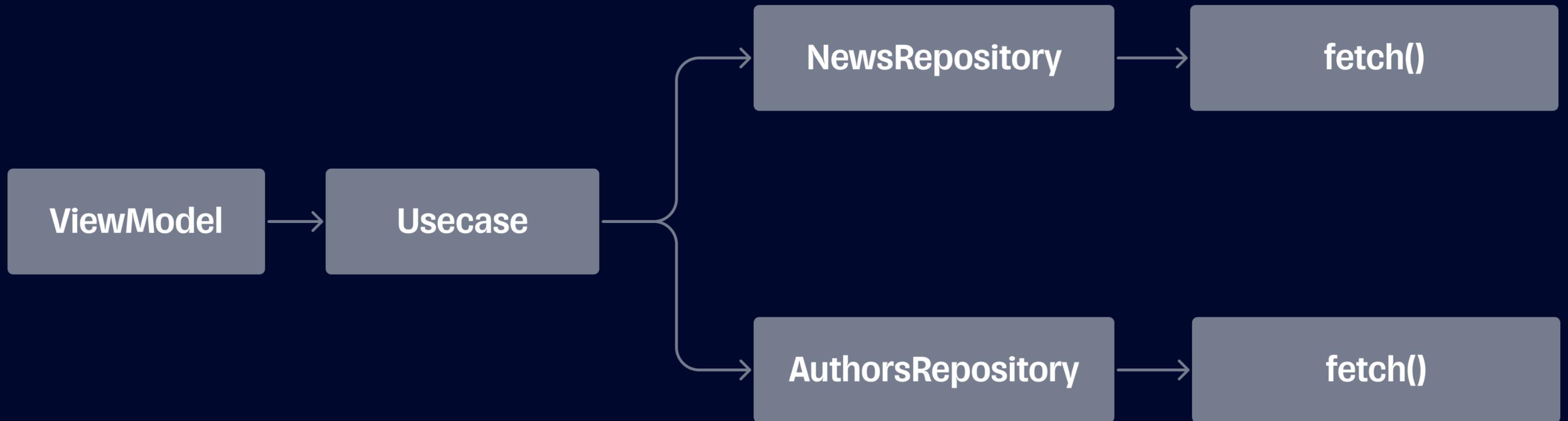
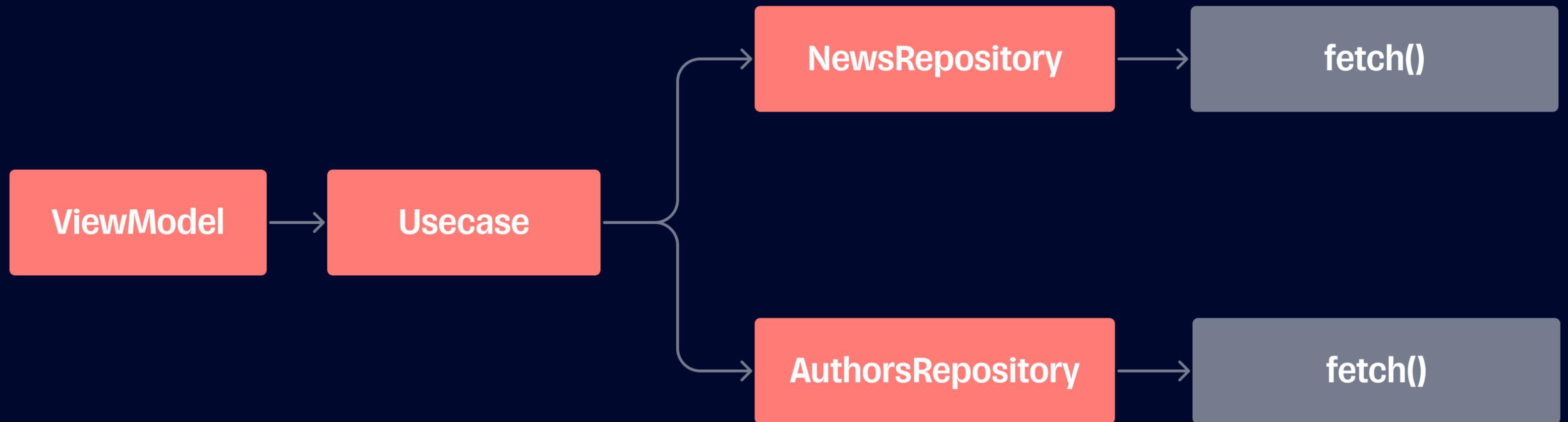


Схема работы



Пример юзкейса

```
// This use case fetches the latest news and the associated author.
class GetLatestNewsWithAuthorsUseCase(
    private val newsRepository: NewsRepository,
    private val authorsRepository: AuthorsRepository
) {
    // This method doesn't need to worry about moving the execution of the
    // coroutine to a different thread as newsRepository is main-safe.
    // The work done in the coroutine is lightweight as it only creates
    // a list and add elements to it
    suspend operator fun invoke(): List<ArticleWithAuthor> {
        val news = newsRepository.fetchLatestNews()

        val response: List<ArticleWithAuthor> = mutableListOf()
        for (article in news) {
            val author = authorsRepository.getAuthor(article.author)
            response.add(ArticleWithAuthor(article, author))
        }
        return Result.Success(response)
    }
}
```

Пример юзкейса

```
// This use case fetches the latest news and the associated author.
class GetLatestNewsWithAuthorsUseCase(
    private val newsRepository: NewsRepository,
    private val authorsRepository: AuthorsRepository
) {
    // This method doesn't need to worry about moving the execution of the
    // coroutine to a different thread as newsRepository is main-safe.
    // The work done in the coroutine is lightweight as it only creates
    // a list and add elements to it
    suspend operator fun invoke(): List<ArticleWithAuthor> {
        val news = newsRepository.fetchLatestNews()

        val response: List<ArticleWithAuthor> = mutableListOf()
        for (article in news) {
            val author = authorsRepository.getAuthor(article.author)
            response.add(ArticleWithAuthor(article, author))
        }
        return Result.Success(response)
    }
}
```

Пример юзкейса

```
// This use case fetches the latest news and the associated author.
class GetLatestNewsWithAuthorsUseCase(
    private val newsRepository: NewsRepository,
    private val authorsRepository: AuthorsRepository
) {
    // This method doesn't need to worry about moving the execution of the
    // coroutine to a different thread as newsRepository is main-safe.
    // The work done in the coroutine is lightweight as it only creates
    // a list and add elements to it
    suspend operator fun invoke(): List<ArticleWithAuthor> {
        val news = newsRepository.fetchLatestNews()

        val response: List<ArticleWithAuthor> = mutableListOf()
        for (article in news) {
            val author = authorsRepository.getAuthor(article.author)
            response.add(ArticleWithAuthor(article, author))
        }
        return Result.Success(response)
    }
}
```

Схема вызовов

`fetchLatestNews()`

Схема ВЫЗОВОВ



Схема ВЫЗОВОВ



Схема вызовов



Можно асинхронно

```
fetchLatestNews()
```

```
fetch news
```

```
val news = newsResult
```

```
getAuthors()
```

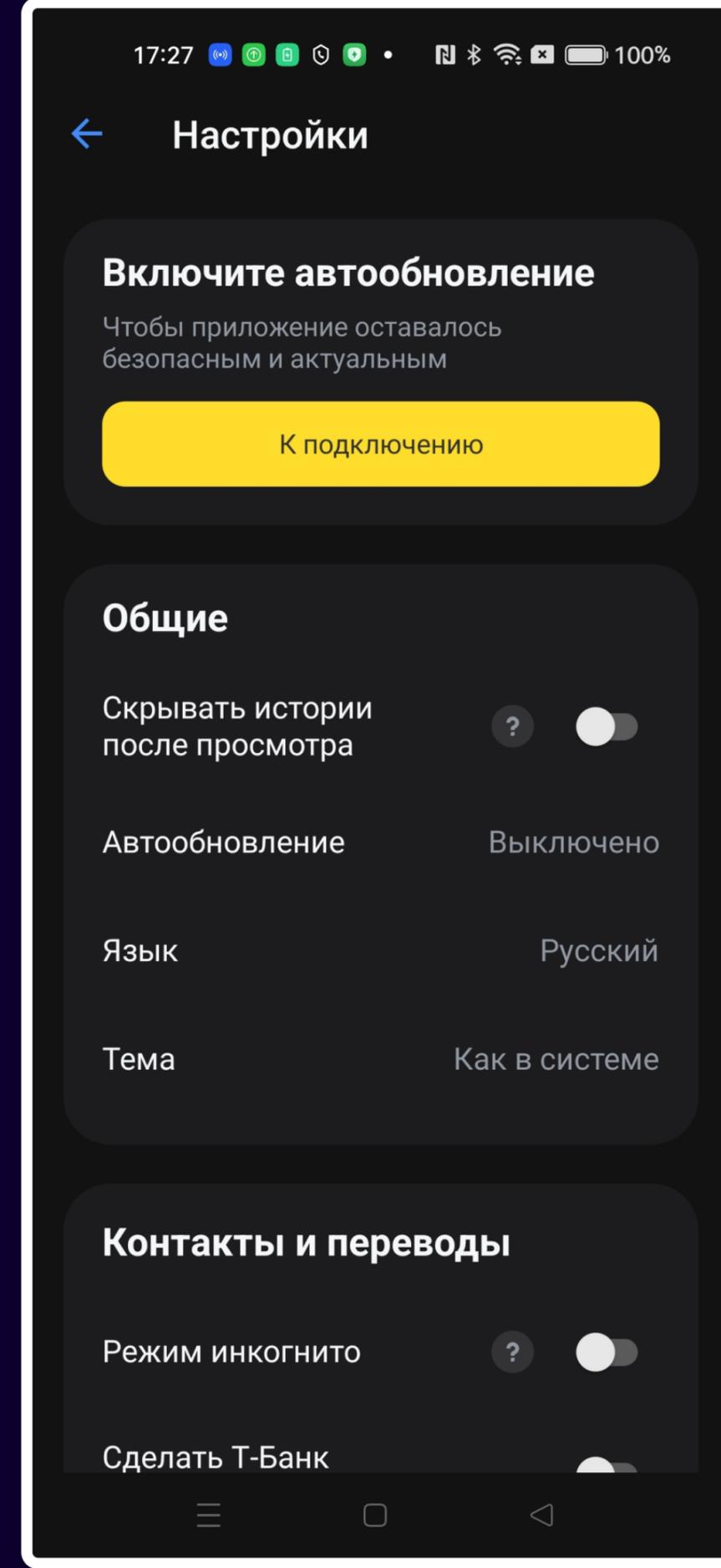
```
fetch authors
```

```
return result
```

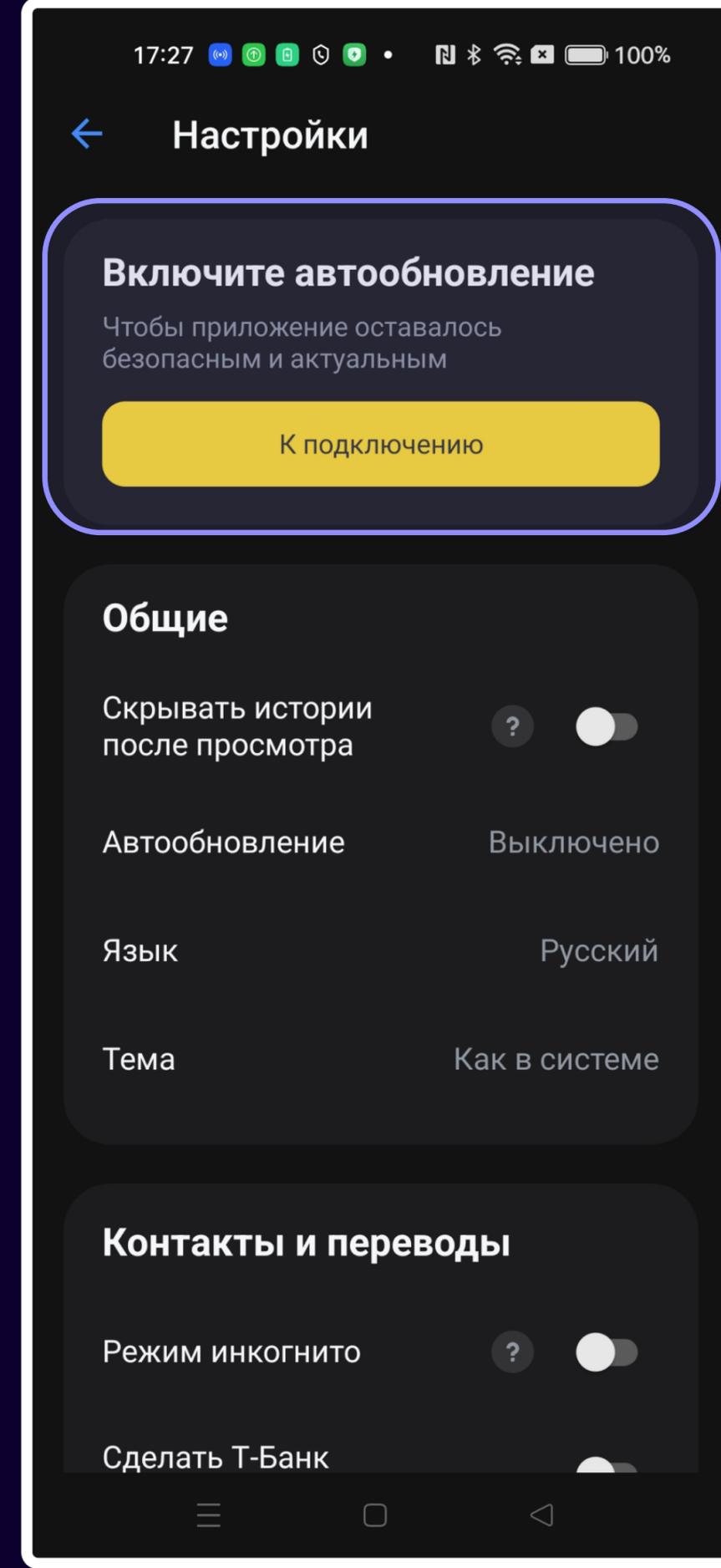
Что может пойти не так?

Реальный пример

Экран настроек



Экран настроек



Экран настроек

...

```
override fun onResume() {  
    super.onResume()  
    viewModel.checkAutoUpdateBannerVisibility()  
}
```

Экран настроек

```
fun checkAutoUpdateBannerVisibility() {  
    viewModelScope.launch {  
        val isAutoUpdateAvailable = autoUpdateSettingsWrapper.isAutoUpdateAvailable()  
        val isAutoUpdateEverNotTurnedOn = autoUpdateSettingsWrapper.isAutoUpdateEverTurnedOn()  
        val isAutoUpdateBannerToggleEnabled = featureToggleManager.getFeature(InterfaceSettings)  
        val isAutoUpdateNotEnabled = autoUpdateSettingsWrapper.isAutoUpdateEnabled().not()  
  
        val isBannerVisible = isAutoUpdateAvailable && isAutoUpdateEverNotTurnedOn && isAutoUpd  
  
        _autoUpdateBannerVisibilityEvent.emit(isBannerVisible)  
    }  
}
```

Экран настроек



```
val isAutoUpdateAvailable = autoUpdateSettingsWrapper.isAutoUpdateAvailable()
val isAutoUpdateEverNotTurnedOn = autoUpdateSettingsWrapper.isAutoUpdateEverTurnedOn().not()
val isAutoUpdateBannerToggleEnabled = featureToggleManager.getFeature(InterfaceSettingsAutoUpdateBannerToggleEnabled)
val isAutoUpdateNotEnabled = autoUpdateSettingsWrapper.isAutoUpdateEnabled().not()
```

Экран настроек

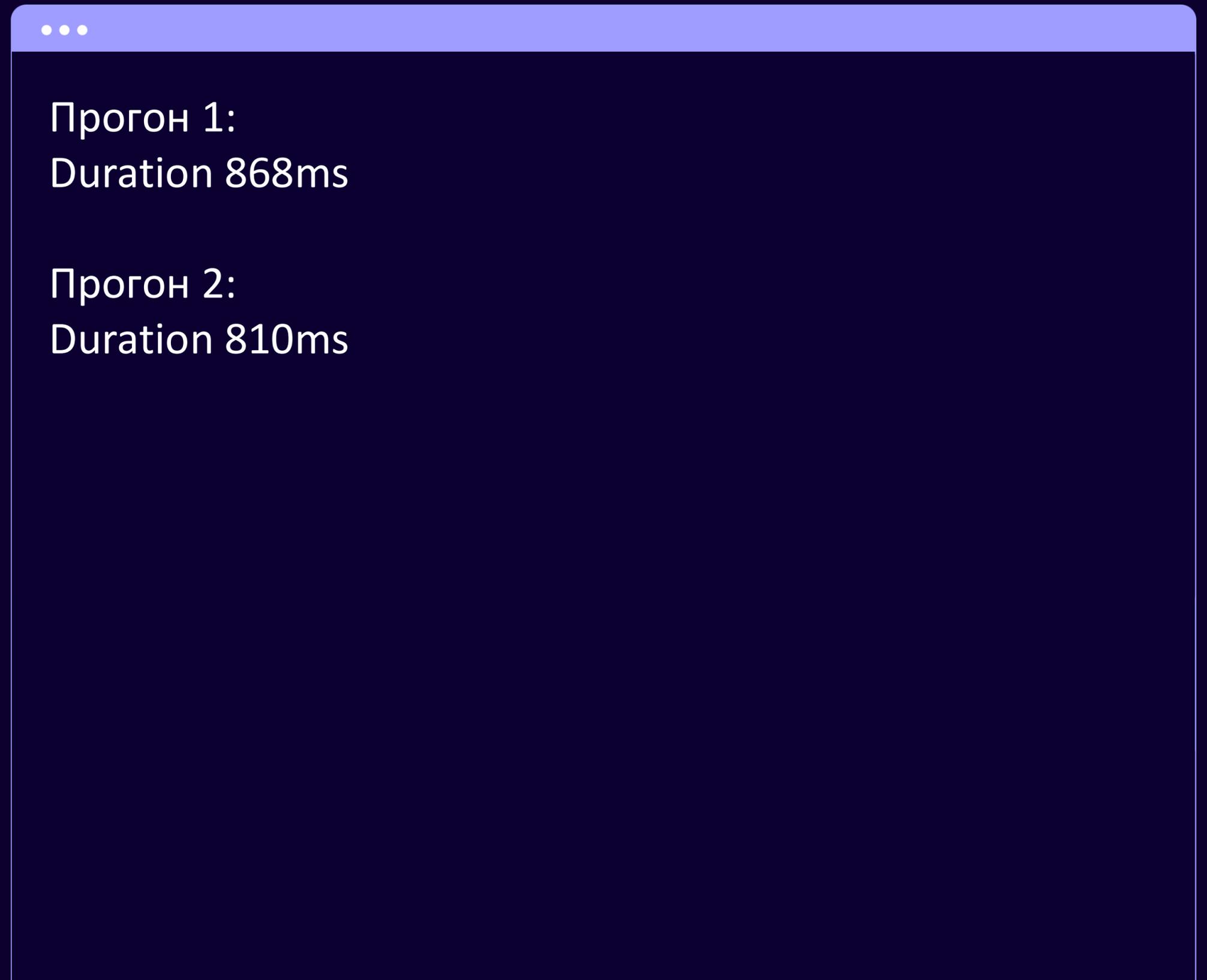


```
_autoUpdateBannerVisibilityEvent.emit(isBannerVisible)
```

Добавим логи

```
fun checkAutoUpdateBannerVisibility() {  
    viewModelScope.launch {  
        val startTime = System.currentTimeMillis()  
        Log.i(LOG_TAG, msg: "start")  
        val isAutoUpdateAvailable = autoUpdateSettingsWrapper.isAutoUpdateAvailable()  
        val isAutoUpdateEverNotTurnedOn = autoUpdateSettingsWrapper.isAutoUpdateEverTurnedOn()  
        val isAutoUpdateBannerToggleEnabled = featureToggleManager.getFeature(InterfaceSetting  
        val isAutoUpdateNotEnabled = autoUpdateSettingsWrapper.isAutoUpdateEnabled().not()  
  
        val isBannerVisible = isAutoUpdateAvailable && isAutoUpdateEverNotTurnedOn && isAutoUp  
  
        _autoUpdateBannerVisibilityEvent.emit(isBannerVisible)  
        val duration = System.currentTimeMillis() - startTime  
        Log.i(LOG_TAG, msg: "duration: $duration")  
    }  
}
```

Логи

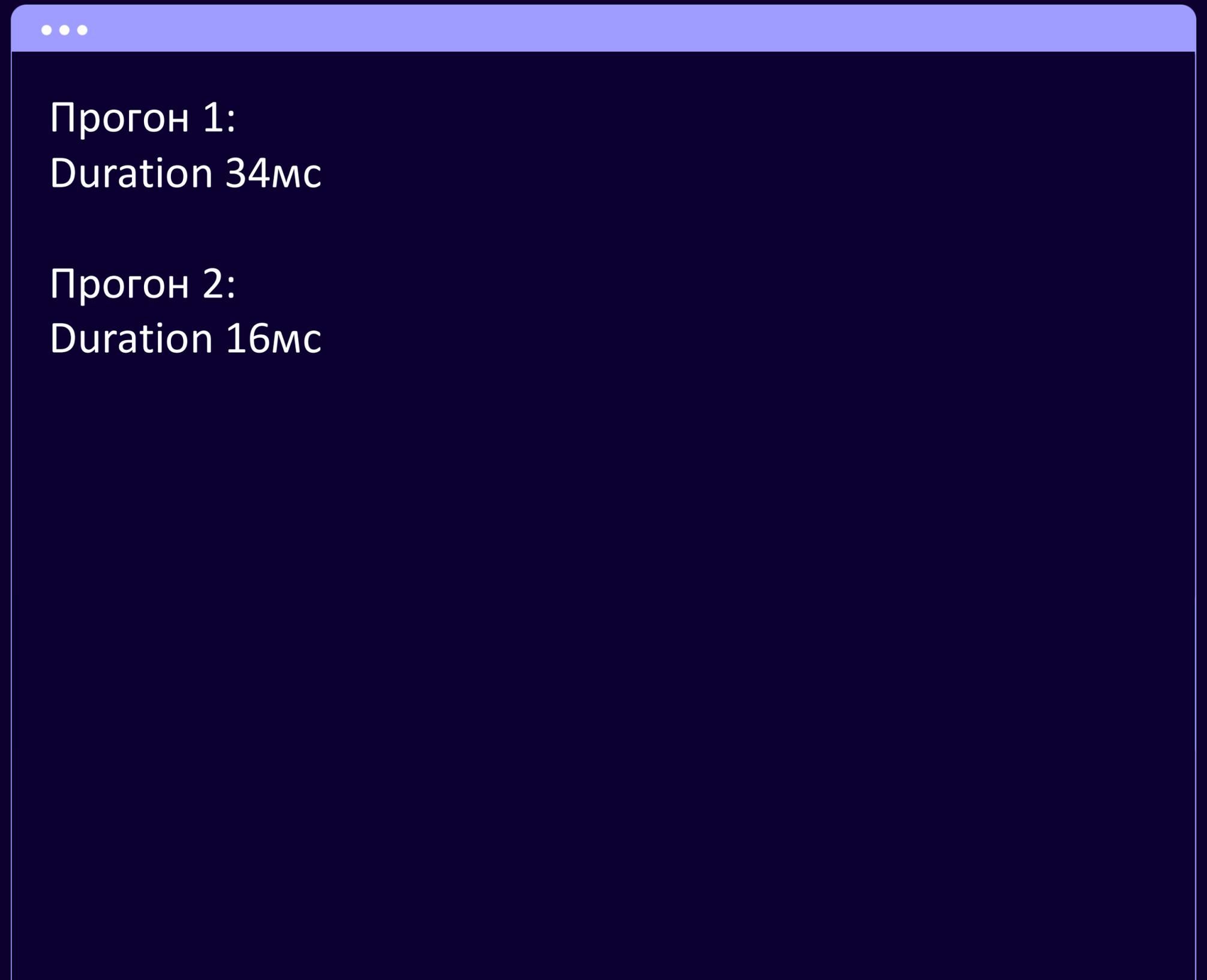


```
...  
Прогон 1:  
Duration 868ms  
  
Прогон 2:  
Duration 810ms
```

Переведем на default

```
fun checkAutoUpdateBannerVisibility() {  
    viewModelScope.launch(Dispatchers.Default) {  
        val startTime = System.currentTimeMillis()  
        Log.i(LOG_TAG, msg: "start")  
        val isAutoUpdateAvailable = autoUpdateSettingsWrapper.isAutoUpdateAvailable()  
        val isAutoUpdateEverNotTurnedOn = autoUpdateSettingsWrapper.isAutoUpdateEverTurnedOn()  
        val isAutoUpdateBannerToggleEnabled = featureToggleManager.getFeature(InterfaceSetting  
        val isAutoUpdateNotEnabled = autoUpdateSettingsWrapper.isAutoUpdateEnabled().not()  
  
        val isBannerVisible = isAutoUpdateAvailable && isAutoUpdateEverNotTurnedOn && isAutoUp  
  
        _autoUpdateBannerVisibilityEvent.emit(isBannerVisible)  
        val duration = System.currentTimeMillis() - startTime  
        Log.i(LOG_TAG, msg: "duration: $duration")  
    }  
}
```

Логи



```
...  
Прогон 1:  
Duration 34мс  
  
Прогон 2:  
Duration 16мс
```

Почему?

Попробуем объяснить

Где принимаем результат



```
autoUpdateBannerVisibilityEvent.collectWithLifecycle(  
    lifecycleOwner: this@ApplicationSettingsActivityViewBindingAdapter,  
    ::changeAutoUpdateBannerVisibility,  
)
```

Где принимаем результат

```
private fun changeAutoUpdateBannerVisibility(isVisible: Boolean) {  
    viewBinding.autoUpdateBannerSection.isVisible = isVisible  
}
```

Добавим больше логов

```
fun checkAutoUpdateBannerVisibility() {
    viewModelScope.launch {
        val startTime = System.currentTimeMillis()
        Log.i(LOG_TAG, msg: "start")
        val isAutoUpdateAvailable = autoUpdateSettingsWrapper.isAutoUpdateAvailable()
        Log.i(LOG_TAG, msg: "after1 ${System.currentTimeMillis() - startTime}")
        val isAutoUpdateEverNotTurnedOn = autoUpdateSettingsWrapper.isAutoUpdateEverTurnedOn()
        Log.i(LOG_TAG, msg: "after2 ${System.currentTimeMillis() - startTime}")
        val isAutoUpdateBannerToggleEnabled = featureToggleManager.getFeature(InterfaceSetting
        Log.i(LOG_TAG, msg: "after3 ${System.currentTimeMillis() - startTime}")
        val isAutoUpdateNotEnabled = autoUpdateSettingsWrapper.isAutoUpdateEnabled().not()
        Log.i(LOG_TAG, msg: "after4 ${System.currentTimeMillis() - startTime}")
        val isBannerVisible = isAutoUpdateAvailable && isAutoUpdateEverNotTurnedOn && isAutoUp

        _autoUpdateBannerVisibilityEvent.emit(isBannerVisible)
        val duration = System.currentTimeMillis() - startTime
        Log.i(LOG_TAG, msg: "duration: $duration")
    }
}
```

Добавим больше логов

```
private fun changeAutoUpdateBannerVisibility(isVisible: Boolean) {  
    viewBinding.autoUpdateBannerSection.isVisible = isVisible  
    Log.i(LOG_TAG, msg: "set")  
}
```

Логи

```
...  
  
start          main  
after1 524ms  main  
after2 613ms  main  
after3 664ms  main  
after4 688ms  main  
  
set          688ms  
  
end          688ms  main
```

Логи

```
...  
start          main  
after1 524ms   main  
after2 613ms   main  
after3 664ms   main  
after4 688ms   main  
  
set  
  
end    688ms   main
```

Логи

```
...  
  
start  
after1 524ms  
after2 613ms  
after3 664ms  
after4 688ms  
  
set  
  
end 688ms
```

Логи

```
...  
  
start  
after1 524ms  
after2 613ms  
after3 664ms  
after4 688ms  
  
set  
  
end 688ms
```

Логи

```
...  
  
start  
after1 524ms  
after2 613ms  
after3 664ms  
after4 688ms  
  
set  
  
end 688ms
```

Логи

```
...  
start    0  
after1   524ms  
after2   613ms  
after3   664ms  
after4   688ms  
  
set      688ms  
  
end      688ms
```

Сменим диспатчер

Применим Default

```
fun checkAutoUpdateBannerVisibility() {  
    viewModelScope.launch(Dispatchers.Default) {  
        val startTime = System.currentTimeMillis()  
        Log.i(LOG_TAG, msg: "start")  
        val isAutoUpdateAvailable = autoUpdateSettingsWrapper.isAutoUpdateAvailable()  
        Log.i(LOG_TAG, msg: "after1 ${System.currentTimeMillis() - startTime}")  
        val isAutoUpdateEverNotTurnedOn = autoUpdateSettingsWrapper.isAutoUpdateEverTurnedOn()  
        Log.i(LOG_TAG, msg: "after2 ${System.currentTimeMillis() - startTime}")  
        val isAutoUpdateBannerToggleEnabled = featureToggleManager.getFeature(InterfaceSetting  
        Log.i(LOG_TAG, msg: "after3 ${System.currentTimeMillis() - startTime}")  
        val isAutoUpdateNotEnabled = autoUpdateSettingsWrapper.isAutoUpdateEnabled().not()  
        Log.i(LOG_TAG, msg: "after4 ${System.currentTimeMillis() - startTime}")  
        val isBannerVisible = isAutoUpdateAvailable && isAutoUpdateEverNotTurnedOn && isAutoUp  
  
        _autoUpdateBannerVisibilityEvent.emit(isBannerVisible)  
        val duration = System.currentTimeMillis() - startTime  
        Log.i(LOG_TAG, msg: "duration: $duration")  
    }  
}
```

Логи Default

...

```
start          DefaultDispatcher-worker-29
after1 2ms     DefaultDispatcher-worker-21
after2 4ms     DefaultDispatcher-worker-24
after3 5ms     DefaultDispatcher-worker-24
after4 5ms     DefaultDispatcher-worker-24
end 5ms       DefaultDispatcher-worker-24

set 653ms
```

Логи Default

```
...  
  
start          DefaultDispatcher-worker-29  
after1 2ms    DefaultDispatcher-worker-21  
after2 4ms    DefaultDispatcher-worker-24  
after3 5ms    DefaultDispatcher-worker-24  
after4 5ms    DefaultDispatcher-worker-24  
end 5ms      DefaultDispatcher-worker-24  
  
set 653m
```

Логи Default

```
...  
  
start  
after1 2ms  
after2 4ms  
after3 5ms  
after4 5ms  
end 5ms  
  
set 653ms
```

**Ускорили выполнение, но не
обновление UI**

В чем причина?

Посмотрим системные логи

Perfetto

Trace config

- Recording settings
- Instructions

Probes

- CPU
- GPU
- Power
- Memory
- Android apps & svcs
- Chrome
- Advanced settings

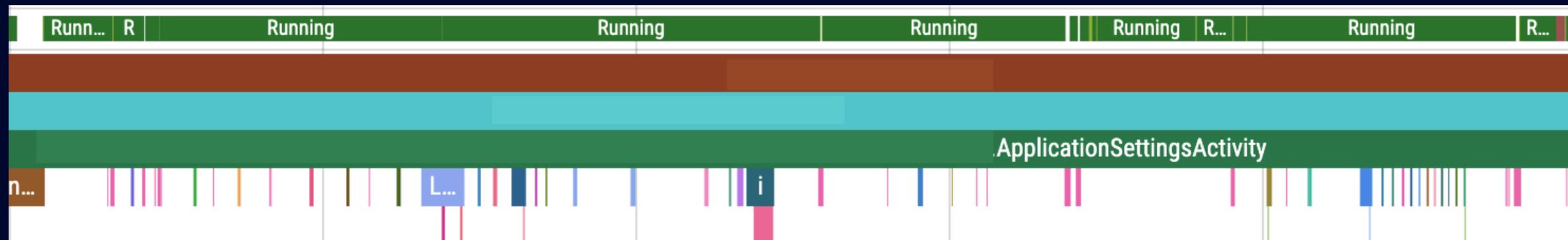
Recording mode

- Stop when full
- Ring buffer
- Long trace

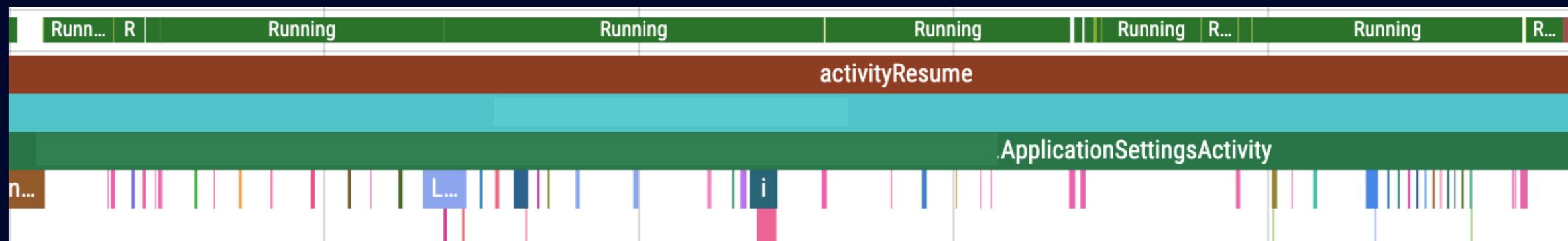
In-memory buffer size: 10 MB

Max duration: 00:00:10 h:m:s

Ищем экран



Ищем activityResume



ActivityResume

activityResume ~ 52 ms

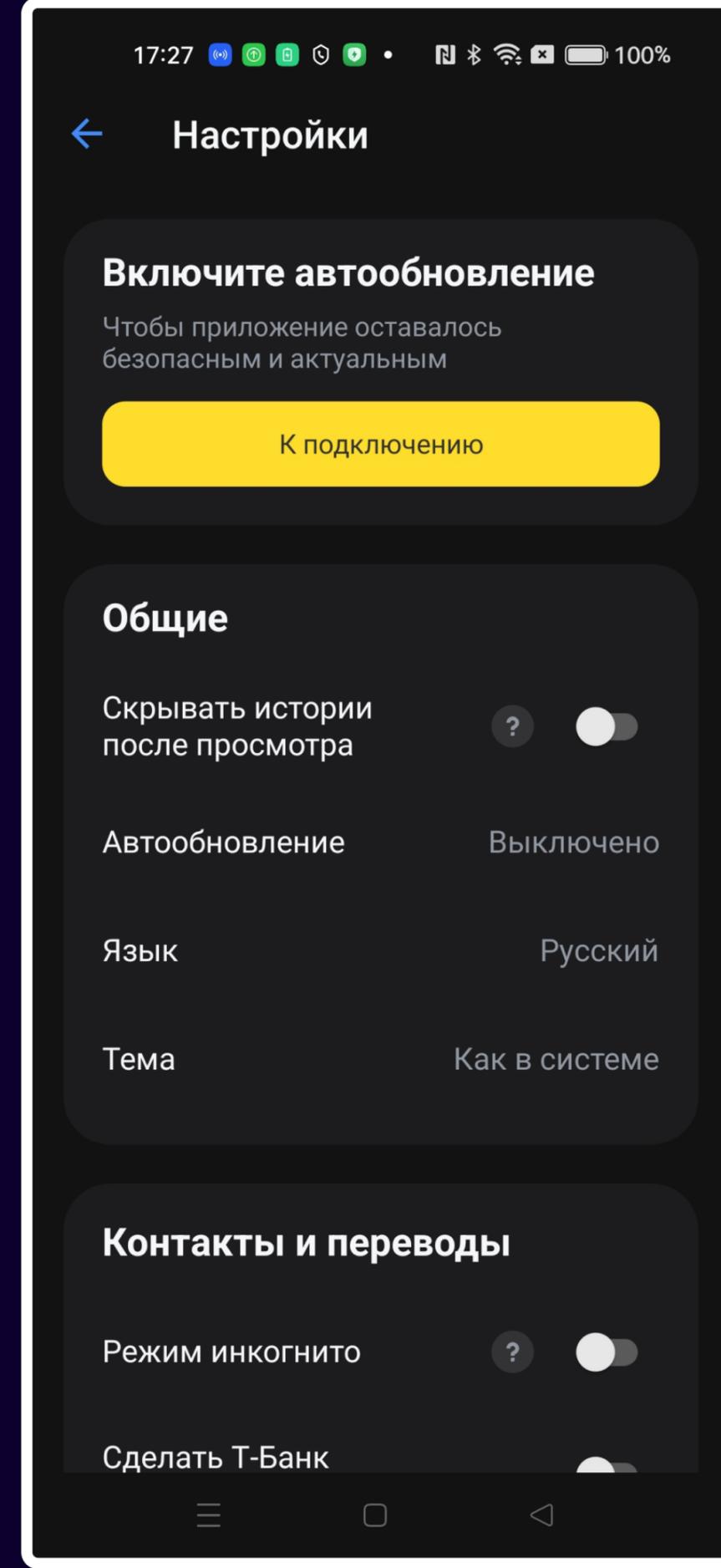


После `activityResume`

После activityResume

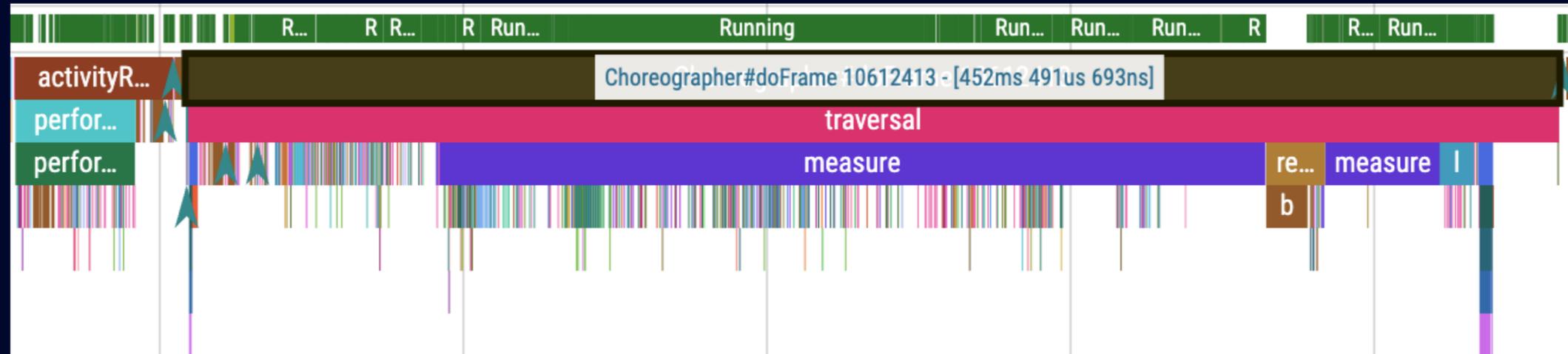


Экран настроек

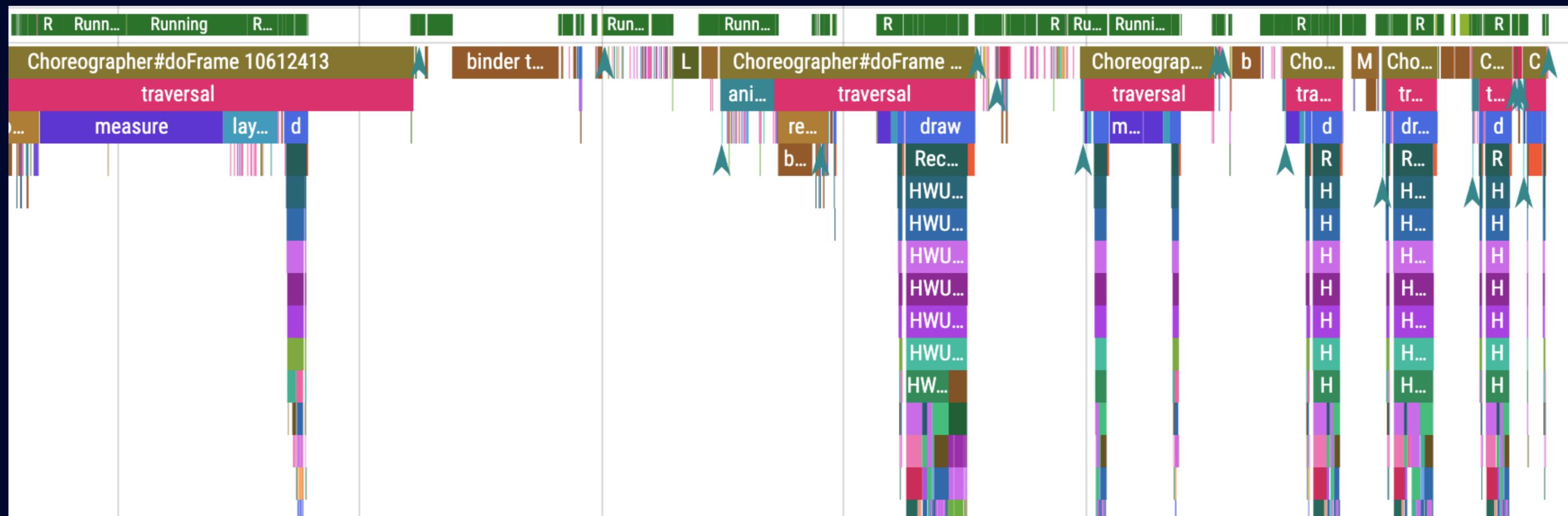


Отрисовка первого кадра

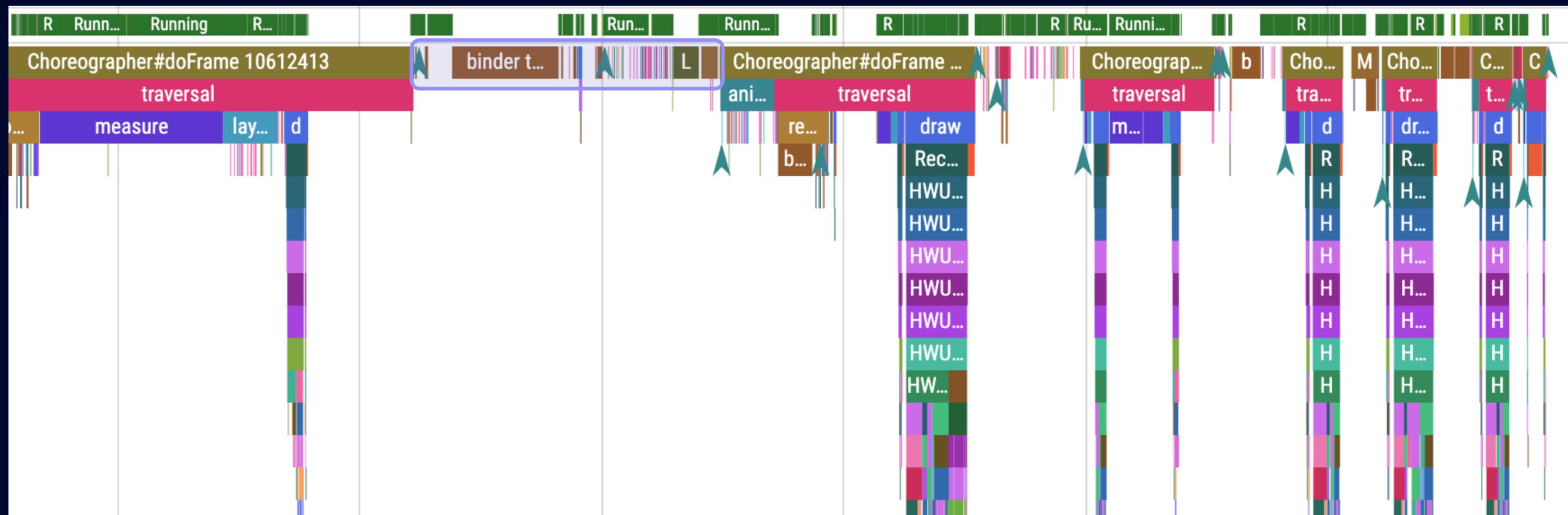
Choreographer#doFrame - 452ms



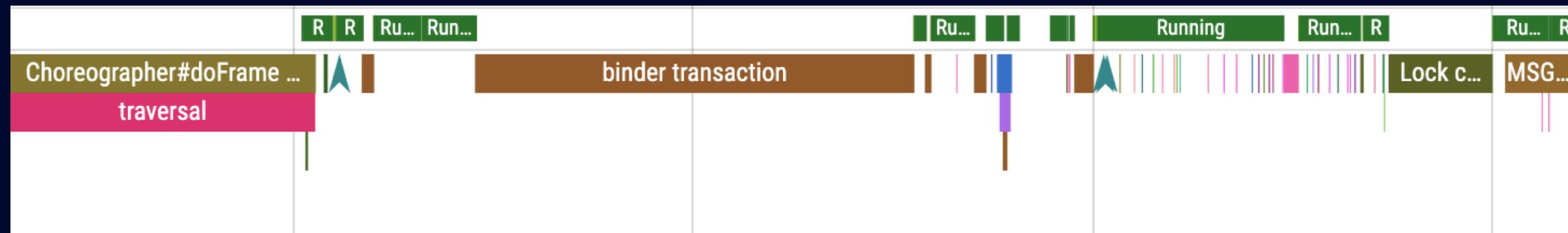
Что дальше



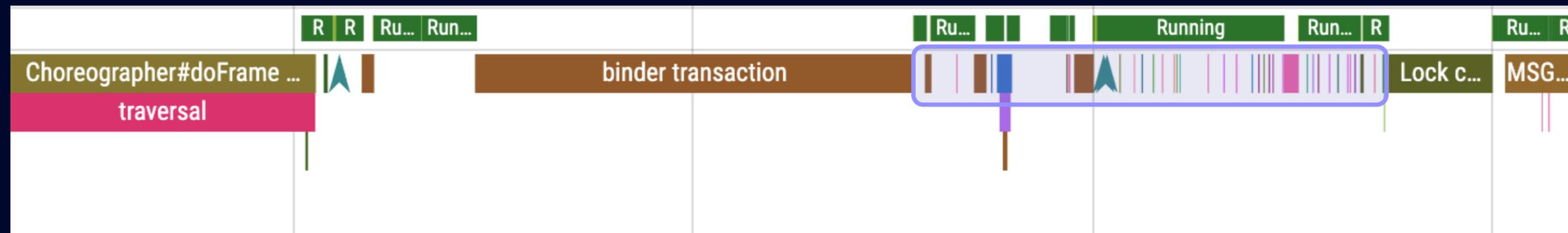
Что дальше



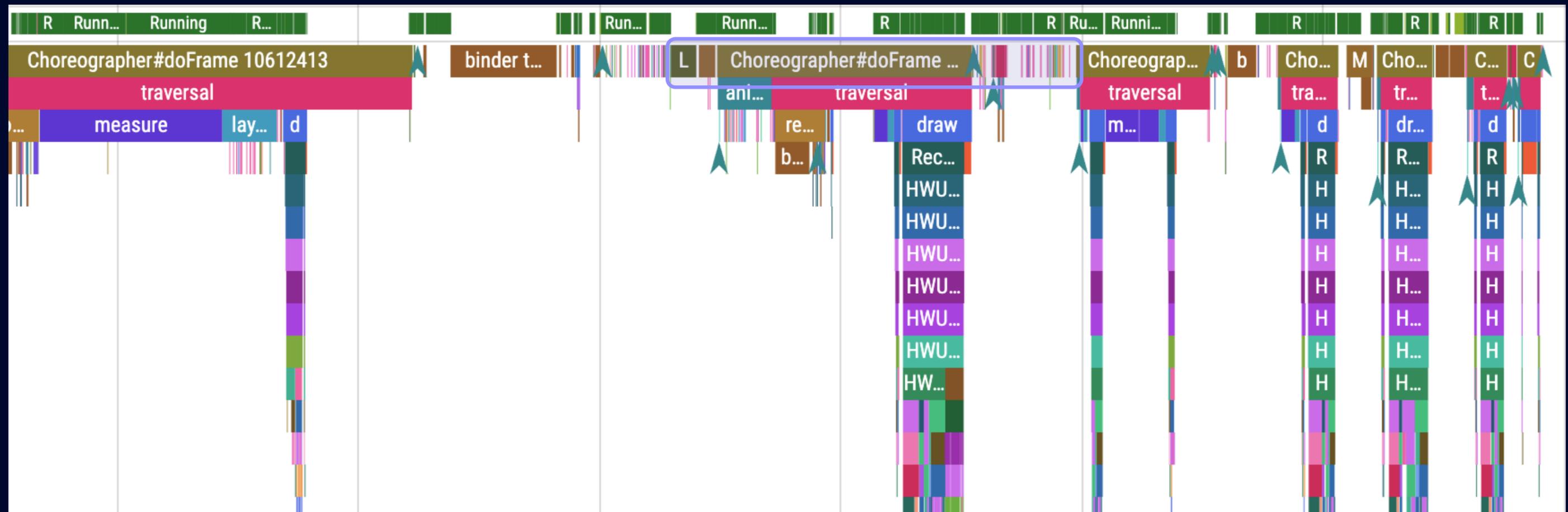
Binder transaction



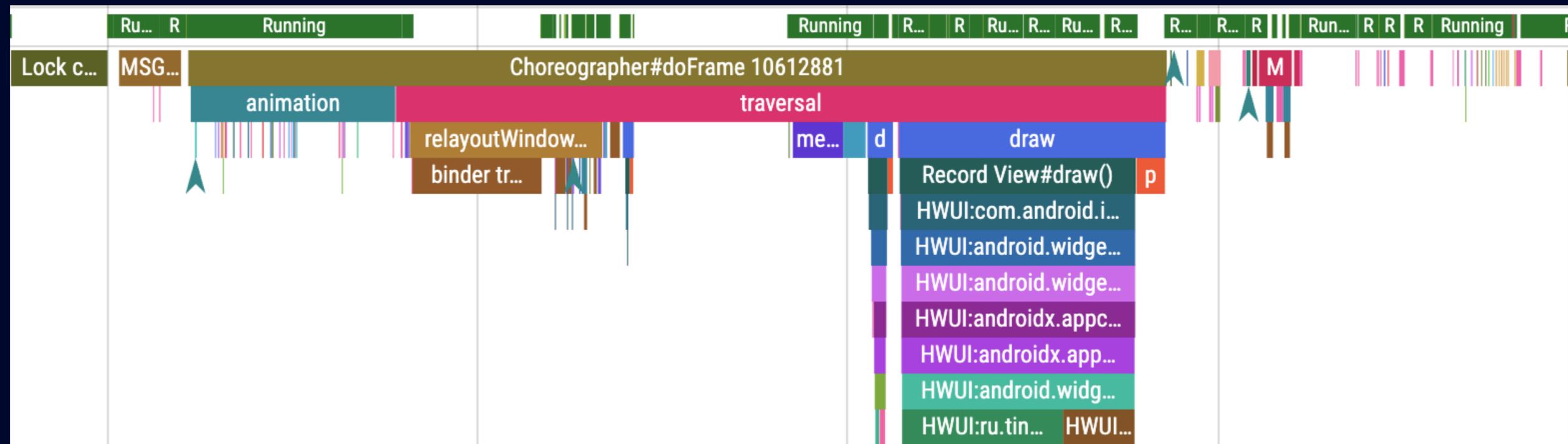
Binder transaction



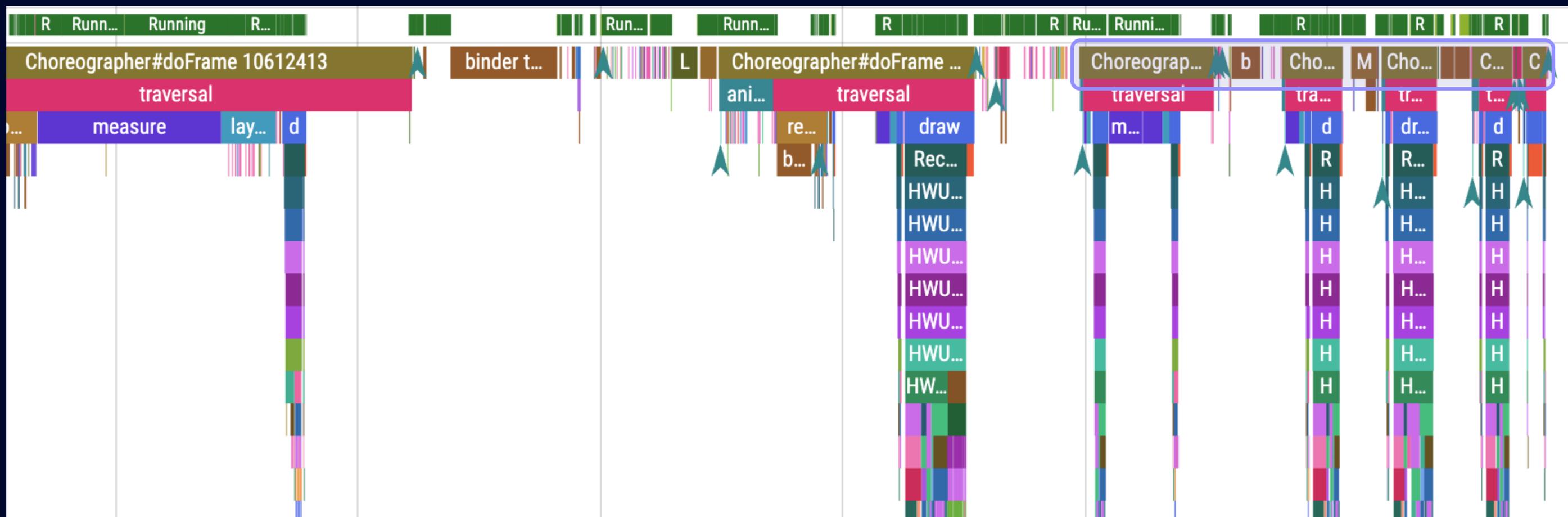
Еще дальше



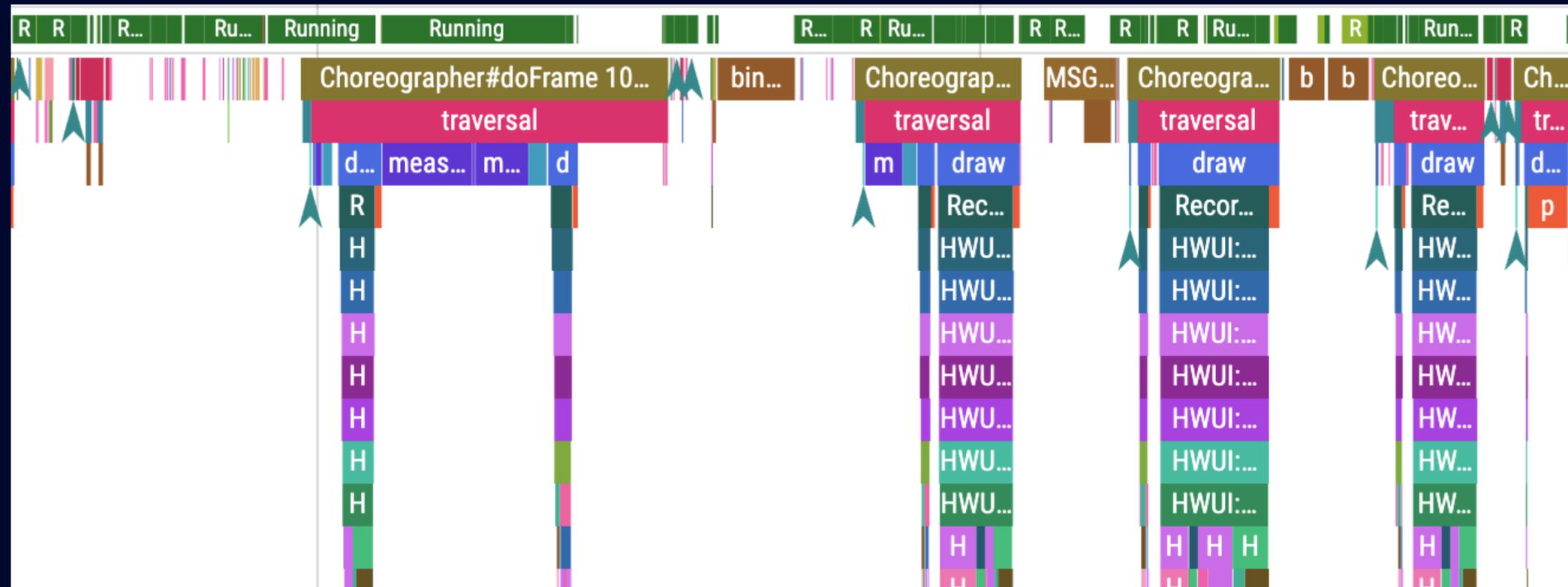
Отрисовка второго кадра



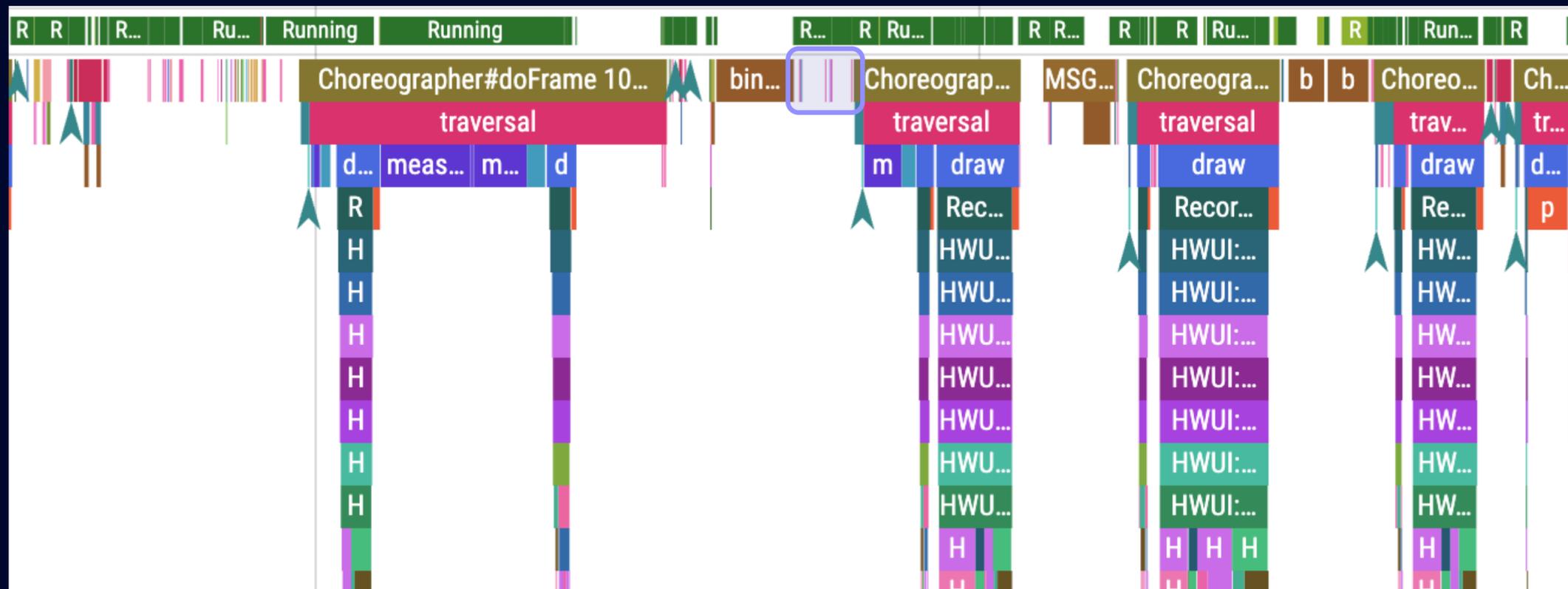
Еще дальше



Отрисовка последующих кадров



Отрисовка последующих кадров



Визуализируем в схеме

Схема с Main dispatcher

RESUME

CHOREOGRAPHER 1 FRAME

2 FRAME

3 FRAME

Схема с Main dispatcher

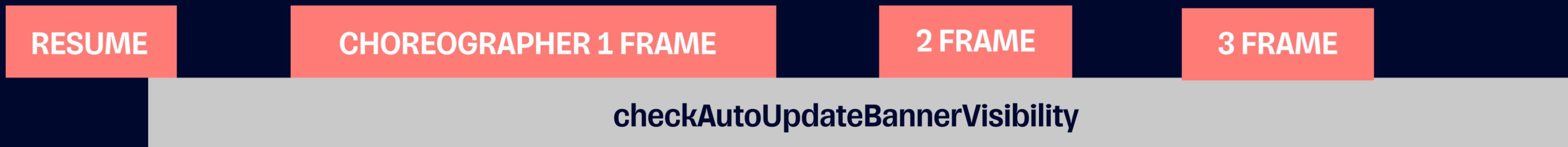


Схема с Main dispatcher

a

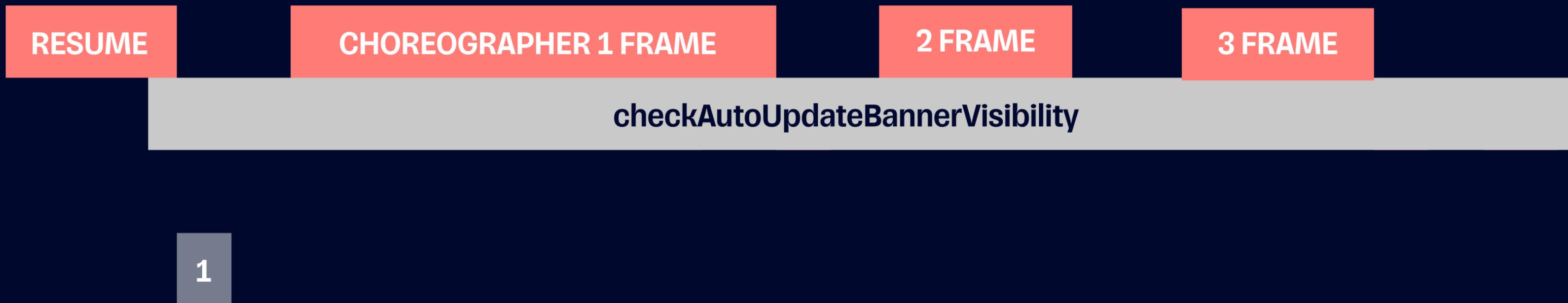


Схема с Main dispatcher

a

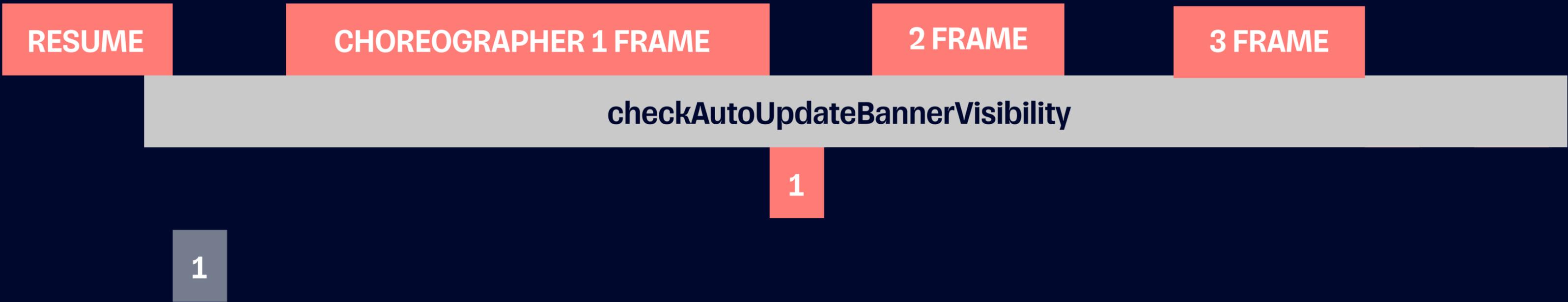


Схема с Main dispatcher

a

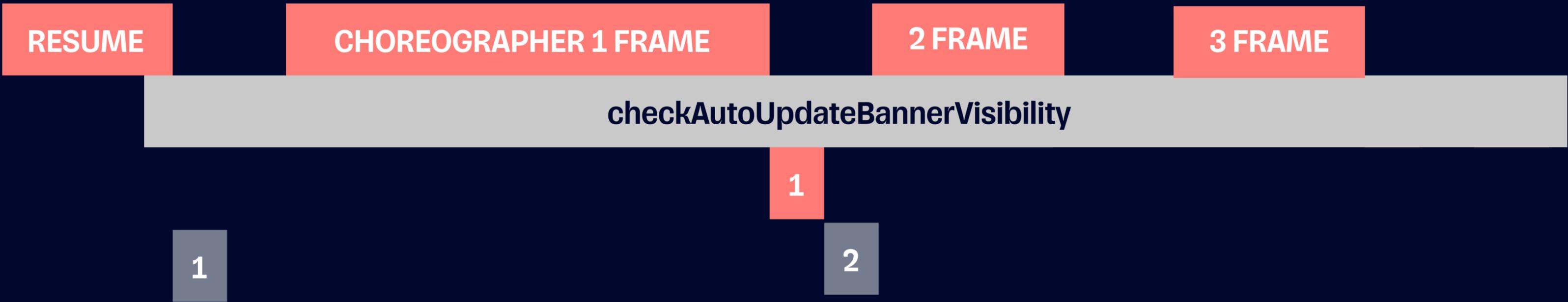


Схема с Main dispatcher

a

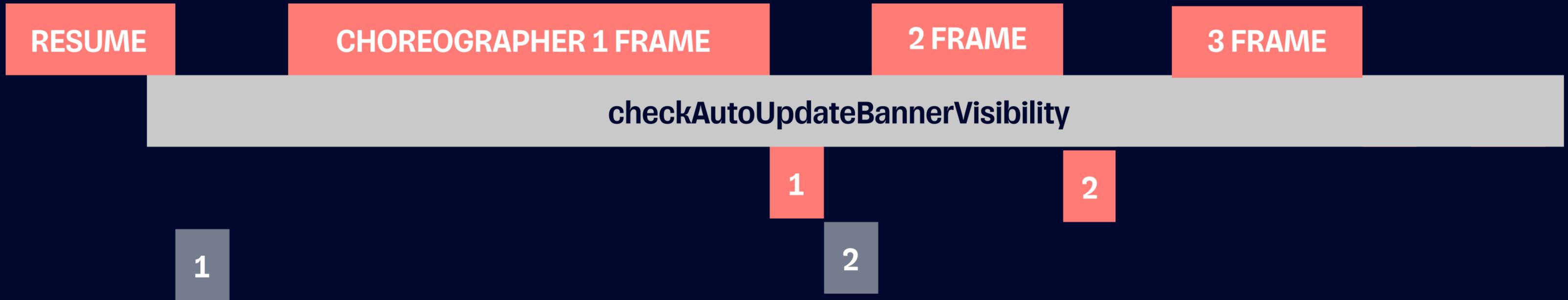


Схема с Main dispatcher

a

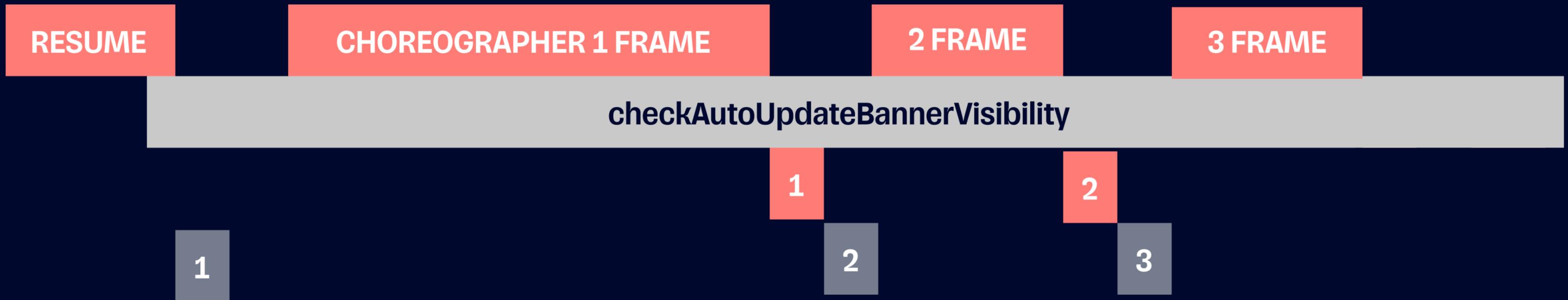


Схема с Main dispatcher

a

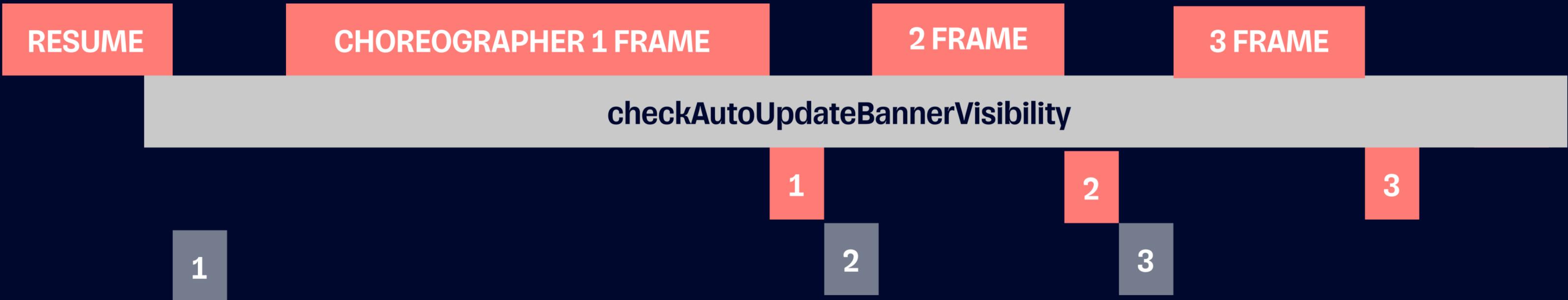
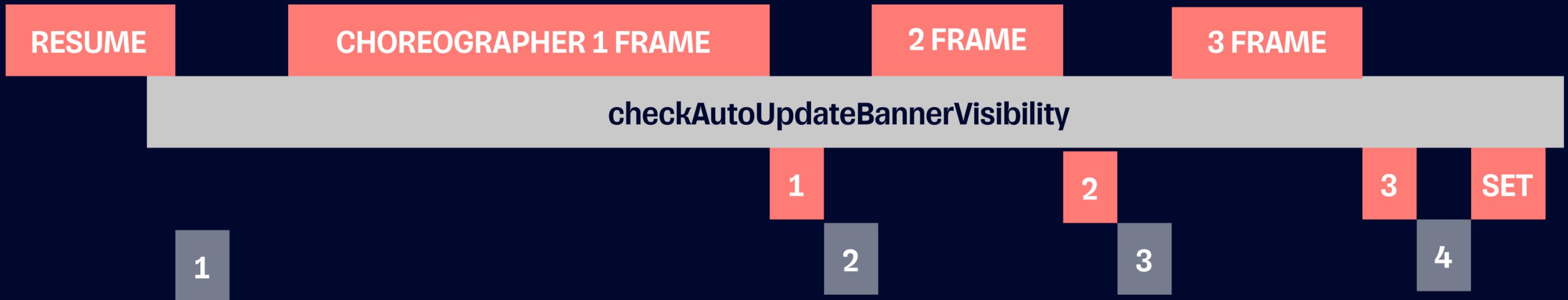


Схема с Main dispatcher

a



Логи

```
...  
  
start  
after1 524ms  
after2 613ms  
after3 664ms  
after4 688ms  
  
set  
  
end 688ms
```

Схема работы на Default

a

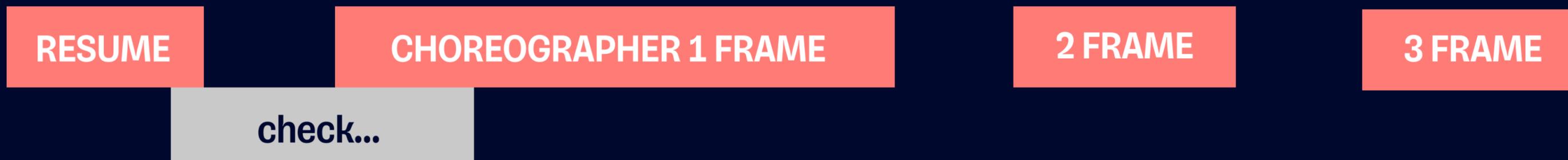


Схема работы на Default

a



Схема работы на Default

a

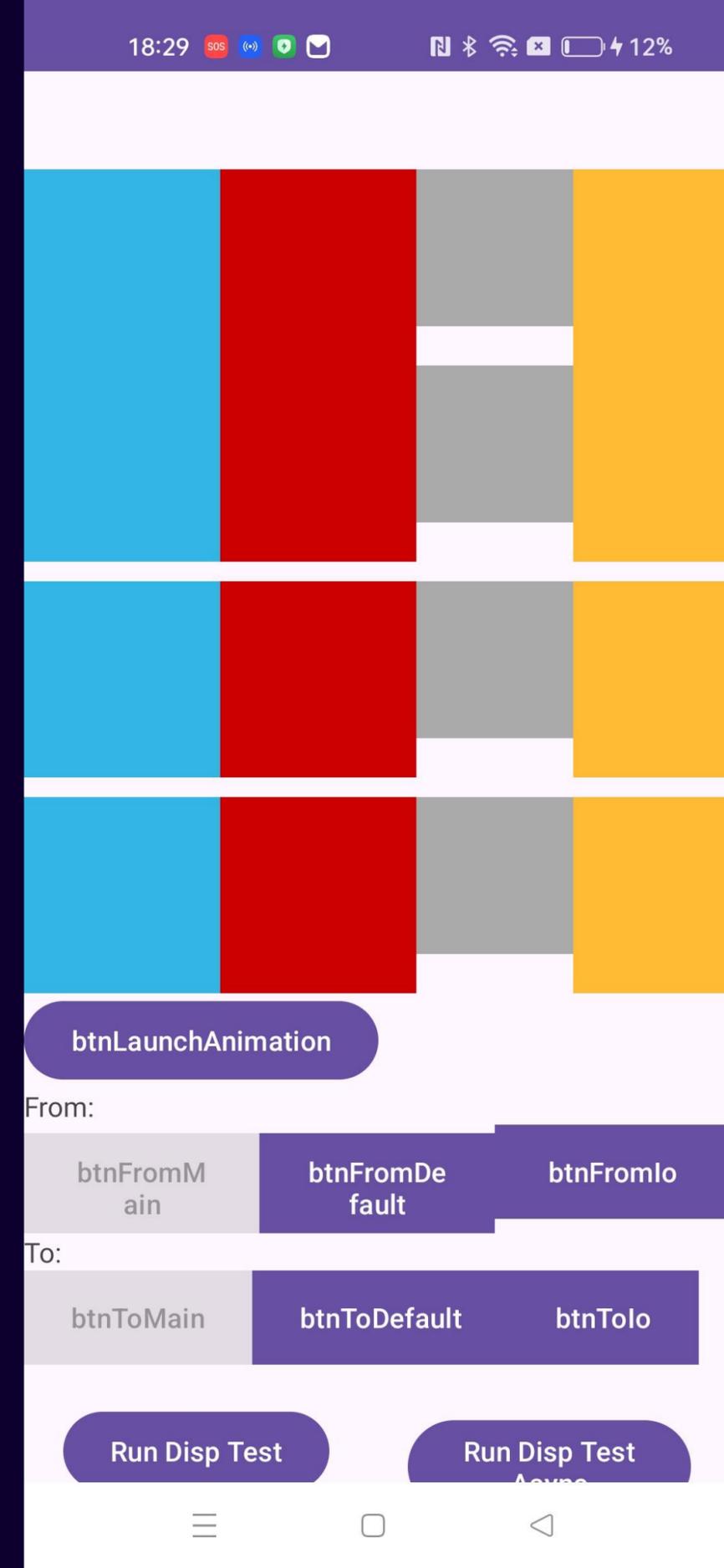


Логи Default

```
...  
  
start  
after1 2ms  
after2 4ms  
after3 5ms  
after4 5ms  
end 5ms  
  
set 653ms
```

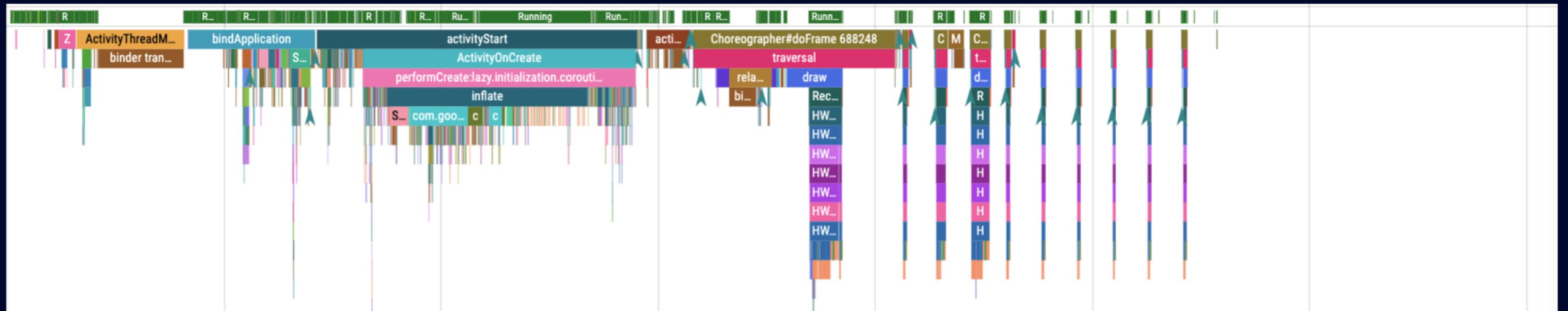
Посмотрим на другие экраны

Демо приложение



Старт Демо приложения

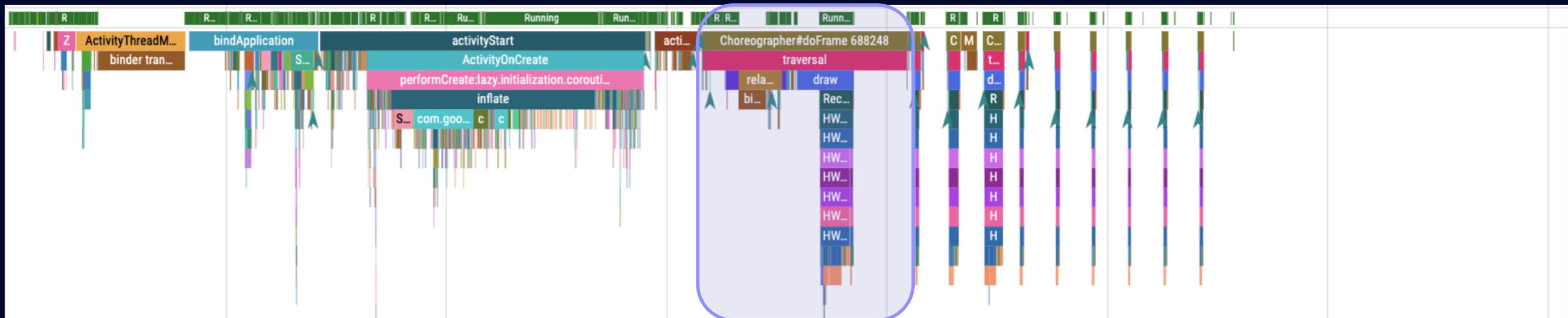
a



Старт Демо приложения

a

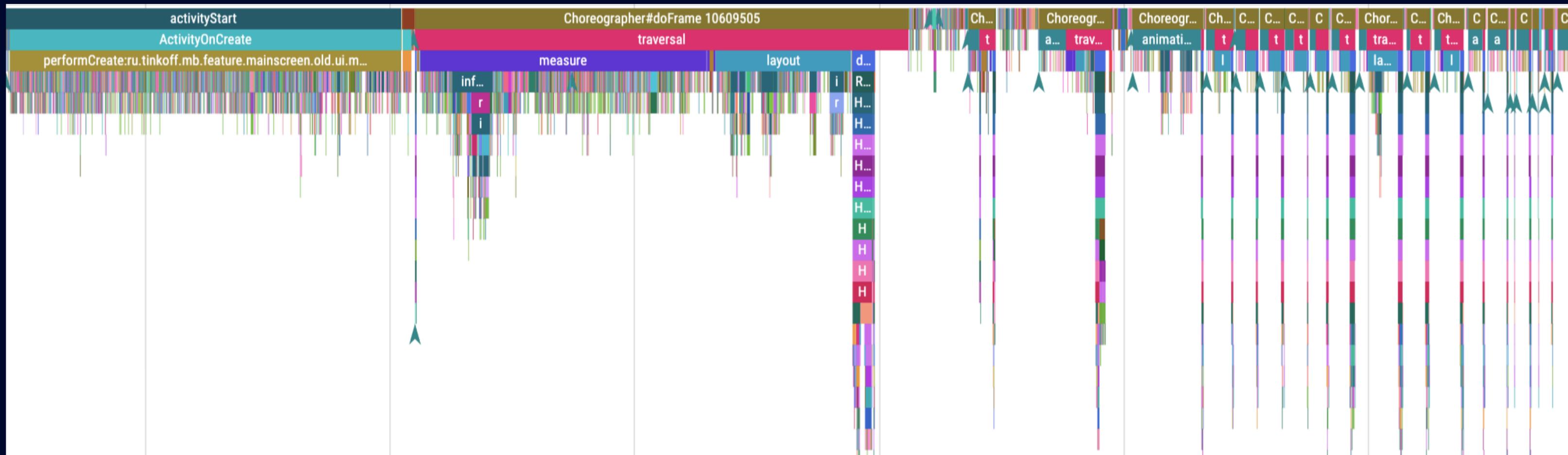
Choreographer#doFrame- 93ms



Главная супераппа

Старт главной Т-Банка

a



Резюме

Резюме



У главного потока много задач

Резюме



У главного потока много задач



При каждом переключении можно
зависнуть в ожидании

Резюме



У главного потока много задач



При каждом переключении можно зависнуть в ожидании



Чем больше переключений, тем дольше придется ждать

Резюме

- У главного потока много задач
- При каждом переключении можно зависнуть в ожидании
- Чем больше переключений, тем дольше придется ждать
- Используя Main dispatcher вы усугубляете проблему

**Избегайте Main Dispatcher
по максимуму**

Что делать?

Заменить score

Подводные камни

Подводные камни



Сохранение очередности событий

limitedParallelism(1)

channels

<https://kotlinlang.org/docs/channels.html>

Подводные камни



Сохранение очередности событий



Задержка при работе с UI

Разделить диспатчеры

Domain scope/dispatcher

All other work

Main scope/dispatcher

Change UI

Подводные камни

- Сохранение очередности событий
- Задержка при работе с UI
- Навигация

СЛОЖНАА

Можно прощe?

Google best practices

```
...  
  
// DO create coroutines in the ViewModel  
class LatestNewsViewModel(  
    private val getLatestNewsWithAuthors: GetLatestNewsWithAuthorsUseCase  
) : ViewModel() {  
  
    private val _uiState = MutableStateFlow<LatestNewsUiState>(LatestNewsUiState.Loading)  
    val uiState: StateFlow<LatestNewsUiState> = _uiState  
  
    fun loadNews() {  
        viewModelScope.launch {  
            val latestNewsWithAuthors = getLatestNewsWithAuthors()  
            _uiState.value = LatestNewsUiState.Success(latestNewsWithAuthors)  
        }  
    }  
}
```

Google best practices

```
...  
  
// DO create coroutines in the ViewModel  
class LatestNewsViewModel(  
    private val getLatestNewsWithAuthors: GetLatestNewsWithAuthorsUseCase  
) : ViewModel() {  
  
    private val _uiState = MutableStateFlow<LatestNewsUiState>(LatestNewsUiState.Loading)  
    val uiState: StateFlow<LatestNewsUiState> = _uiState  
  
    fun loadNews() {  
        viewModelScope.launch {  
            val latestNewsWithAuthors = getLatestNewsWithAuthors()  
            _uiState.value = LatestNewsUiState.Success(latestNewsWithAuthors)  
        }  
    }  
}
```

Google best practices

```
...  
  
// DO create coroutines in the ViewModel  
class LatestNewsViewModel(  
    private val getLatestNewsWithAuthors: GetLatestNewsWithAuthorsUseCase  
) : ViewModel() {  
  
    private val _uiState = MutableStateFlow<LatestNewsUiState>(LatestNewsUiState.Loading)  
    val uiState: StateFlow<LatestNewsUiState> = _uiState  
  
    fun loadNews() {  
        viewModelScope.launch {  
            val latestNewsWithAuthors = getLatestNewsWithAuthors()  
            _uiState.value = LatestNewsUiState.Success(latestNewsWithAuthors)  
        }  
    }  
}
```

Google best practices

```
...  
  
// DO create coroutines in the ViewModel  
class LatestNewsViewModel(  
    private val getLatestNewsWithAuthors: GetLatestNewsWithAuthorsUseCase  
) : ViewModel() {  
  
    private val _uiState = MutableStateFlow<LatestNewsUiState>(LatestNewsUiState.Loading)  
    val uiState: StateFlow<LatestNewsUiState> = _uiState  
  
    fun loadNews() {  
        viewModelScope.launch {  
            val latestNewsWithAuthors = getLatestNewsWithAuthors()  
            _uiState.value = LatestNewsUiState.Success(latestNewsWithAuthors)  
        }  
    }  
}
```

Google best practices

```
...  
  
// DO create coroutines in the ViewModel  
class LatestNewsViewModel(  
    private val getLatestNewsWithAuthors: GetLatestNewsWithAuthorsUseCase  
) : ViewModel() {  
  
    private val _uiState = MutableStateFlow<LatestNewsUiState>(LatestNewsUiState.Loading)  
    val uiState: StateFlow<LatestNewsUiState> = _uiState  
  
    fun loadNews() {  
        viewModelScope.launch (BackgroundDispatcher) {  
            val latestNewsWithAuthors = getLatestNewsWithAuthors()  
            _uiState.value = LatestNewsUiState.Success(latestNewsWithAuthors)  
        }  
    }  
}
```

Решение - Dispatcher в launch

Но какой именно?

**Коротко – лучше Default,
можно IO**

Подробно

**Что нужно
понять перед
выбором?**

Что нужно понять перед выбором?



Как влияет количество потоков и переключений на скорость?

Что нужно понять перед выбором?



Как влияет количество потоков и переключений на скорость?



Как влияет на параллельные запросы?

Что нужно понять перед выбором?

- Как влияет количество потоков и переключений на скорость?
- Как влияет на параллельные запросы?
- Увеличим ли количество потоков?

Что нужно понять перед выбором?

- Как влияет количество потоков и переключений на скорость?
- Как влияет на параллельные запросы?
- Увеличим ли количество потоков?
- Можем получить блокировку аналогично с Main?

Как работают Dispatchers и Threads

Default

```
...  
  
// Instance of Dispatchers.Default  
internal object DefaultScheduler : SchedulerCoroutineDispatcher(  
    CORE_POOL_SIZE, MAX_POOL_SIZE,  
    IDLE_WORKER_KEEP_ALIVE_NS, DEFAULT_SCHEDULER_NAME  
) {  
  
    @ExperimentalCoroutinesApi  
    override fun limitedParallelism(parallelism: Int): CoroutineDispatcher {  
        parallelism.checkParallelism()  
        if (parallelism >= CORE_POOL_SIZE) return this  
        return super.limitedParallelism(parallelism)  
    }  
}
```

Default

```
...  
  
// Instance of Dispatchers.Default  
internal object DefaultScheduler : SchedulerCoroutineDispatcher(  
    CORE_POOL_SIZE MAX_POOL_SIZE,  
    IDLE_WORKER_KEEP_ALIVE_NS, DEFAULT_SCHEDULER_NAME  
) {  
  
    @ExperimentalCoroutinesApi  
    override fun limitedParallelism(parallelism: Int): CoroutineDispatcher {  
        parallelism.checkParallelism()  
        if (parallelism >= CORE_POOL_SIZE) return this  
        return super.limitedParallelism(parallelism)  
    }  
}
```

IO

```
...  
  
// Dispatchers.IO  
internal object DefaultIoScheduler : ExecutorCoroutineDispatcher(), Executor {  
  
    private val default = UnlimitedIoScheduler.limitedParallelism(  
        systemProp(  
            IO_PARALLELISM_PROPERTY_NAME,  
            64.coerceAtLeast(AVAILABLE_PROCESSORS)  
        )  
    )  
}
```

IO

```
...  
  
// Dispatchers.IO  
internal object DefaultIOScheduler : ExecutorCoroutineDispatcher(), Executor {  
  
    private val default = UnlimitedIOScheduler.limitedParallelism(  
        systemProp(  
            IO_PARALLELISM_PROPERTY_NAME,  
            64.coerceAtLeast(AVAILABLE_PROCESSORS)  
        )  
    )  
}
```

**Посмотрим разницу на
практике**

Тест на блокировку потоков

Запустим параллельно Thread.sleep

```
val deferredList = List(COUNT) {  
    async(dispatcher, start = CoroutineStart.LAZY) {  
        Thread.sleep(SECOND)  
    }  
}  
deferredList.awaitAll()
```

8 Thread.Sleep

	Количество созданных потоков	Время(секунды)
Default	8	
IO	8	

8 Thread.Sleep

	Количество созданных потоков	Время(секунды)
Default	8	1
IO	8	1

Запустим 100 раз

100 Thread.Sleep

	Количество созданных потоков	Время(секунды)
Default	8	
IO	64	

100 Thread.Sleep

	Количество созданных потоков	Время(секунды)
Default	8	13
IO	64	2

100 Thread.Sleep

	Количество созданных потоков	Время(секунды)
Default	8	13 = 104 / 8
IO	64	2 = 128 / 64

Выводы

Выводы



IO оптимален для ожидания ответа

Выводы



IO оптимален для ожидания ответа



Default можно заблокировать операцией ожидания ответа

Тест на нагрузку

Запустим параллельно долгие задачи

```
val deferredList = List(COUNT) {  
    async(dispatcher, start = CoroutineStart.LAZY) {  
        longTask()  
    }  
}  
deferredList.awaitAll()
```

16 раз

16 долгих задач

	Количество созданных потоков	Время (мс) P75
Default	8	
IO	16	

16 долгих задач

	Количество созданных потоков	Время (мс) P75
Default	8	140
IO	16	142

1000 раз

1000 долгих задач

	Количество созданных потоков P75	Время (секунды) P75
Default	8	
IO	66	

1000 долгих задач

	Количество созданных потоков P75	Время (секунды) P75
Default	8	11,3
IO	66	18,8

Выводы

Выводы



Default оптимален для операций CPU

Выводы



Default оптимален для операций CPU



IO может создать больше потоков, чем необходимо

Выводы

- Default оптимален для операций CPU
- IO может создать больше потоков, чем необходимо
- При нагрузке IO тратит много времени на переключения между потоками

Практический пример

Имитируем запуск юзкейса

```
async(dispatcher) {  
    withContext(Dispatchers.IO) {  
        Thread.sleep(millis: 100)  
    }  
}
```

16 раз

16 юзкейсов

	Созданных потоков P75	Время (мс) P75	Переключений туда P75	Переключений обратно P75
С Default на IO	21			
С IO на IO	16			

16 юзкейсов

	Созданных потоков P75	Время (мс) P75	Переключений туда P75	Переключений обратно P75
С Default на IO	21	107		
С IO на IO	16	104		

16 юзкейсов

	Созданных потоков P75	Время (мс) P75	Переключений туда P75	Переключений обратно P75
С Default на IO	21	107	8	14
С IO на IO	16	104	0	0

100 раз

100 юзкейсов

	Созданных потоков P75	Время (мс) P75	Переключений туда P75	Переключений обратно P75
С Default на IO	72			
С IO на IO	64			

100 юзкейсов

	Созданных потоков P75	Время (мс) P75	Переключений туда P75	Переключений обратно P75
С Default на IO	72	213		
С IO на IO	64	211		

100 юзкейсов

	Созданных потоков P75	Время (мс) P75	Переключений туда P75	Переключений обратно P75
С Default на IO	72	213	67	88
С IO на IO	64	211	0	0

Используем кастомный ThreadPool

Используем свой ThreadPool

```
async(dispatcher, start = CoroutineStart.LAZY) {  
    withContext(customDispatcher) {  
        Thread.sleep(millis: 100)  
    }  
}
```

Сравнение

	Созданных потоков P75	Время (мс) P75	Переключений P75
С Default на IO	72	213	155
С Default на Custom	72	212	200
С IO на IO	64	211	0
С IO на Custom	106	216	200

Сравнение

	Созданных потоков P75	Время (мс) P75	Переключений P75
С Default на IO		213	
С Default на Custom		212	
С IO на IO		211	
С IO на Custom		216	

Сравнение

	Созданных потоков P75	Время (мс) P75	Переключений P75
С Default на IO	72		155
С Default на Custom	72		200
С IO на IO	64		0
С IO на Custom	106		200

Выводы

Выводы



При низкой нагрузке Default и IO равноценны по времени выполнения

Выводы



При низкой нагрузке Default и IO равноценны по времени выполнения



Количество потоков и переключений не влияет на скорость само по себе

**Добавим имитацию обработки
ответа**

Имитация обработки ответа

```
async(dispatcher, start = CoroutineStart.LAZY) {  
    withContext(Dispatchers.IO) {  
        Thread.sleep(millis: 100)  
    }  
    LongTask()  
}
```

Сравнение

	Созданных потоков P75	Время (мс) P75	Переключений P75
С Default на IO	72	1281	169
С Default на Custom	72	1291	200
С IO на IO	64	1665	0
С IO на Custom	134	1743	200

Сравнение

	Созданных потоков P75	Время (мс) P75	Переключений P75
С Default на IO	72		169
С Default на Custom	72		200
С IO на IO	64		0
С IO на Custom	134		200

Сравнение

	Созданных потоков P75	Время (мс) P75	Переключений P75
С Default на IO		1281	
С Default на Custom		1291	
С IO на IO		1665	
С IO на Custom		1743	

Выводы

Выводы



Default лучше подходит для обработки ответа

Выводы



Default лучше подходит для обработки ответа



На скорость влияет не количество потоков и переключений, а загруженность диспатчера

Выводы

- Default лучше подходит для обработки ответа
- На скорость влияет не количество потоков и переключений, а загруженность диспатчера
- Важно правильно подобрать диспатчер под задачу

Выводы

- Default лучше подходит для обработки ответа
- На скорость влияет не количество потоков и переключений, а загруженность диспатчера
- Важно правильно подобрать диспатчер под задачу
- За ошибку придется заплатить временем, либо памятью

Какой dispatcher выбрать?

Какой dispatcher выбрать

	Когда использовать	Ограничения
Default	Выполнения кода	Можно заблокировать ожиданием

Какой dispatcher выбрать

	Когда использовать	Ограничения
Default	Выполнения кода	Можно заблокировать ожиданием
IO	Ожидания ответа	Если нагрузить, то тратит время на переключения между потоками

Какой dispatcher выбрать

	Когда использовать	Ограничения
Default	Выполнения кода	Можно заблокировать ожиданием
IO	Ожидания ответа	Если нагрузить, то тратит время на переключения между потоками
Custom	Исключения	Нет переиспользования потоков + + вне ограничения количества ядер + дополнительная настройка

Подведем итоги

У главного потока много работы

У главного потока много работы



ЖЦ

У главного потока много работы



ЖЦ



Binder

У главного потока много работы



ЖЦ



Binder



Choreographer

У главного потока много работы



ЖЦ



Binder



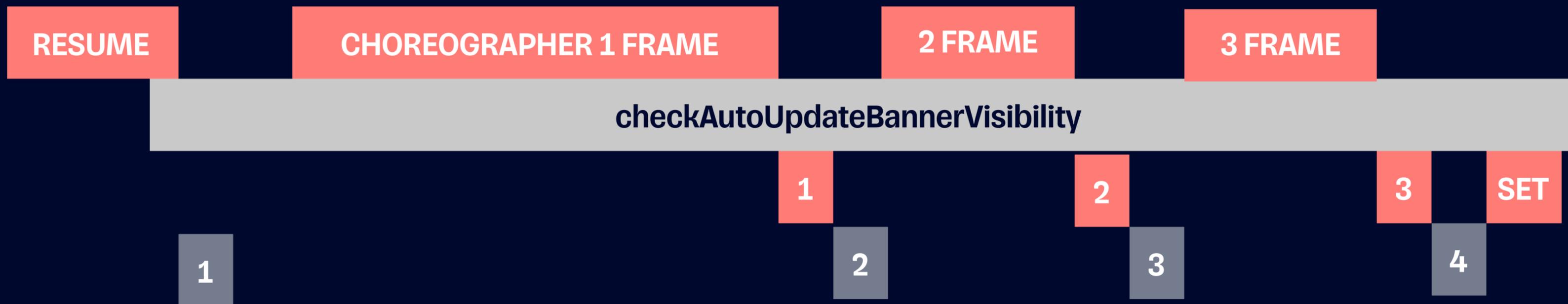
Choreographer



Колбеки и т.д.

Каждое переключение придется ждать

a



**Избегайте Main Dispatcher
по максимуму**

Ставьте диспатчер при загрузке данных

```
// DO create coroutines in the ViewModel
class LatestNewsViewModel(
    private val getLatestNewsWithAuthors: GetLatestNewsWithAuthorsUseCase
) : ViewModel() {

    private val _uiState = MutableStateFlow<LatestNewsUiState>(LatestNewsUiState.Loading)
    val uiState: StateFlow<LatestNewsUiState> = _uiState

    fun loadNews() {
        viewModelScope.launch (Dispatchers.Default) {
            val latestNewsWithAuthors = getLatestNewsWithAuthors()
            _uiState.value = LatestNewsUiState.Success(latestNewsWithAuthors)
        }
    }
}
```

Подбирайте диспатчер под задачи

	Когда использовать	Ограничения

Подбирайте диспатчер под задачи

	Когда использовать	Ограничения
Default	Выполнения кода	Можно заблокировать ожиданием

Подбирайте диспатчер под задачи

	Когда использовать	Ограничения
Default	Выполнения кода	Можно заблокировать ожиданием
IO	Ожидания ответа	Если нагрузить, то тратит время на переключения между потоками

Попробуйте и замерьте



Александр Таганов | Тимлид



a.taganov@tbank.ru

T: @aa_taganov

Главный вывод



Спасибо!

