Kotlin Coroutines First

ĶIRILL ŘOZOV

KIRILL ROZOV

Head of Android Development@Humans.net

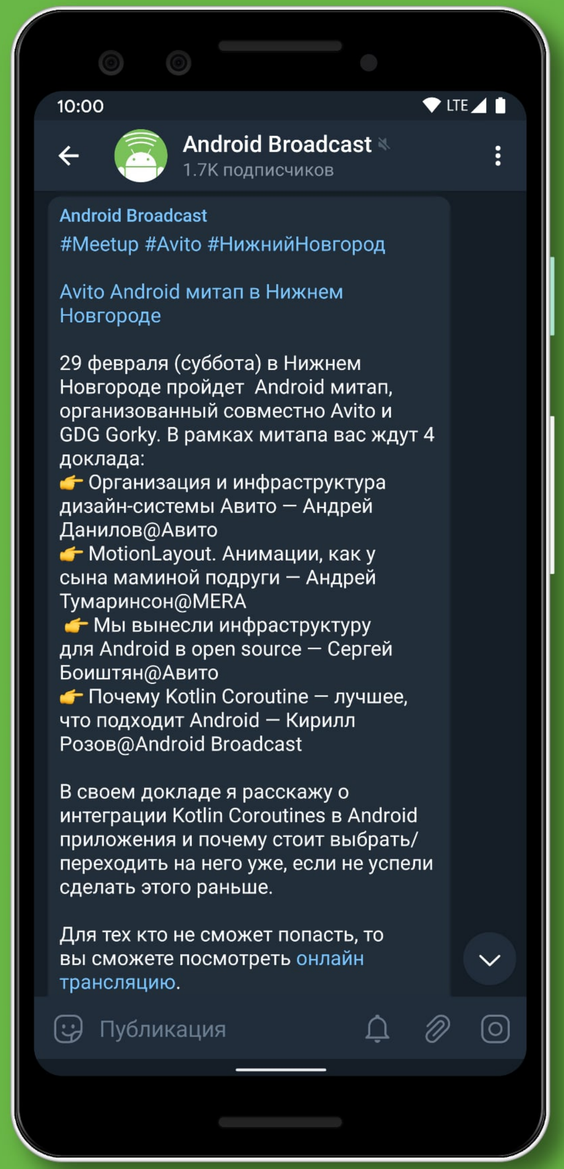
krl.rozov@gmail.com

@kirill_rozov

@kirillr

Android Broadcast

Подборка лучших новостей из мира Android разработки



t.me/android_broadcast

Modern Android Development

DI

Kotlin

MVP
MVVM
MVI
MVx

Android
Arch
Components

Gradle

RxJava
Coroutines
Reaktive

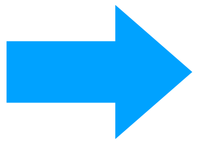
Multi
Modules

Android SDK Problems

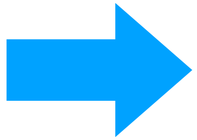
- Need to execute UI operations on the Main thread
- Need to execute network operations in background
- Component Lifecycle
- Activity/Fragment recreation
- No solution in Android SDK to solve issues

Modern Device

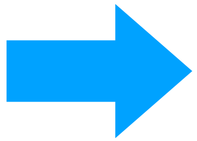
- Multiple CPU
- big.LITTLE architecture
- Big screen
- A lot of RAM
- Always connected to internet
- Hundreds of apps installed



- Multiply parallel operations
- Boost for hard operation

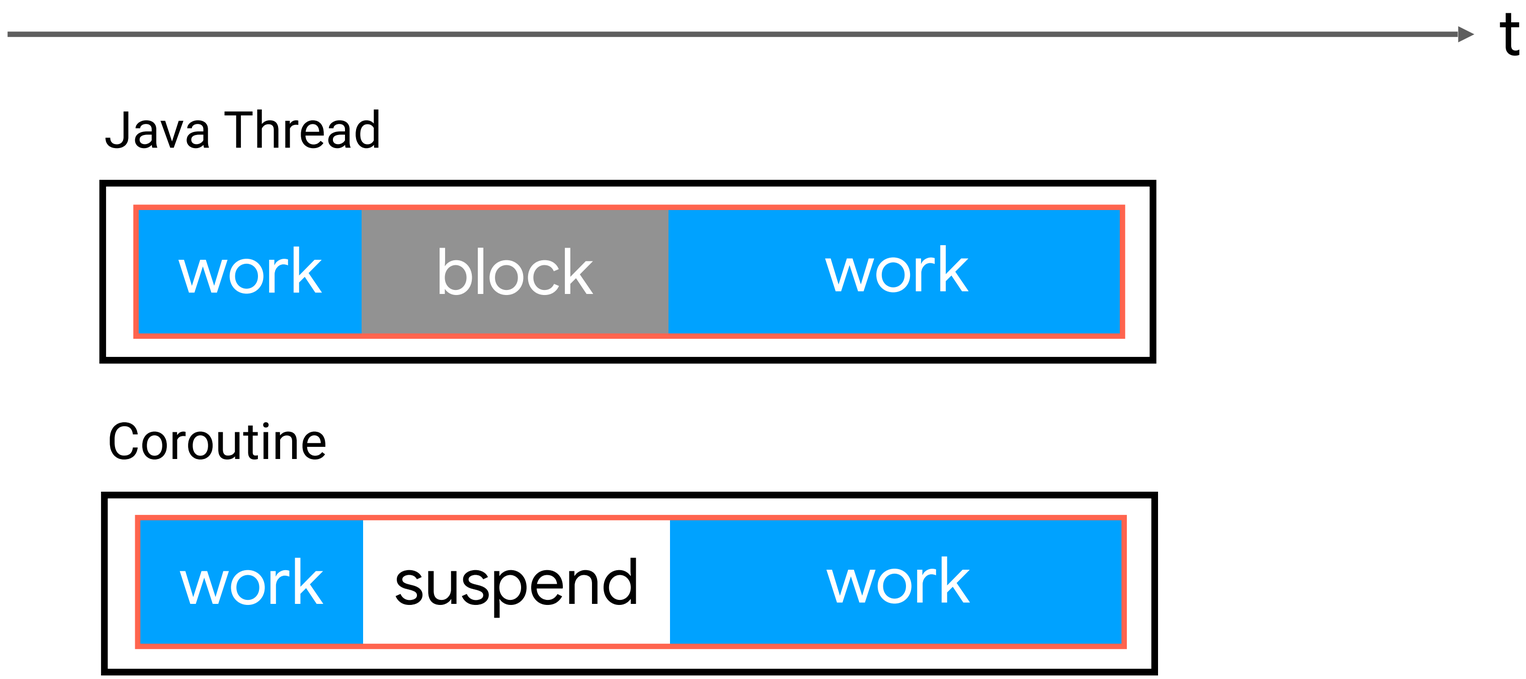


- Need more data to display
- More apps running simultaneously



- Every app check what’s new (of course using Push Notification)

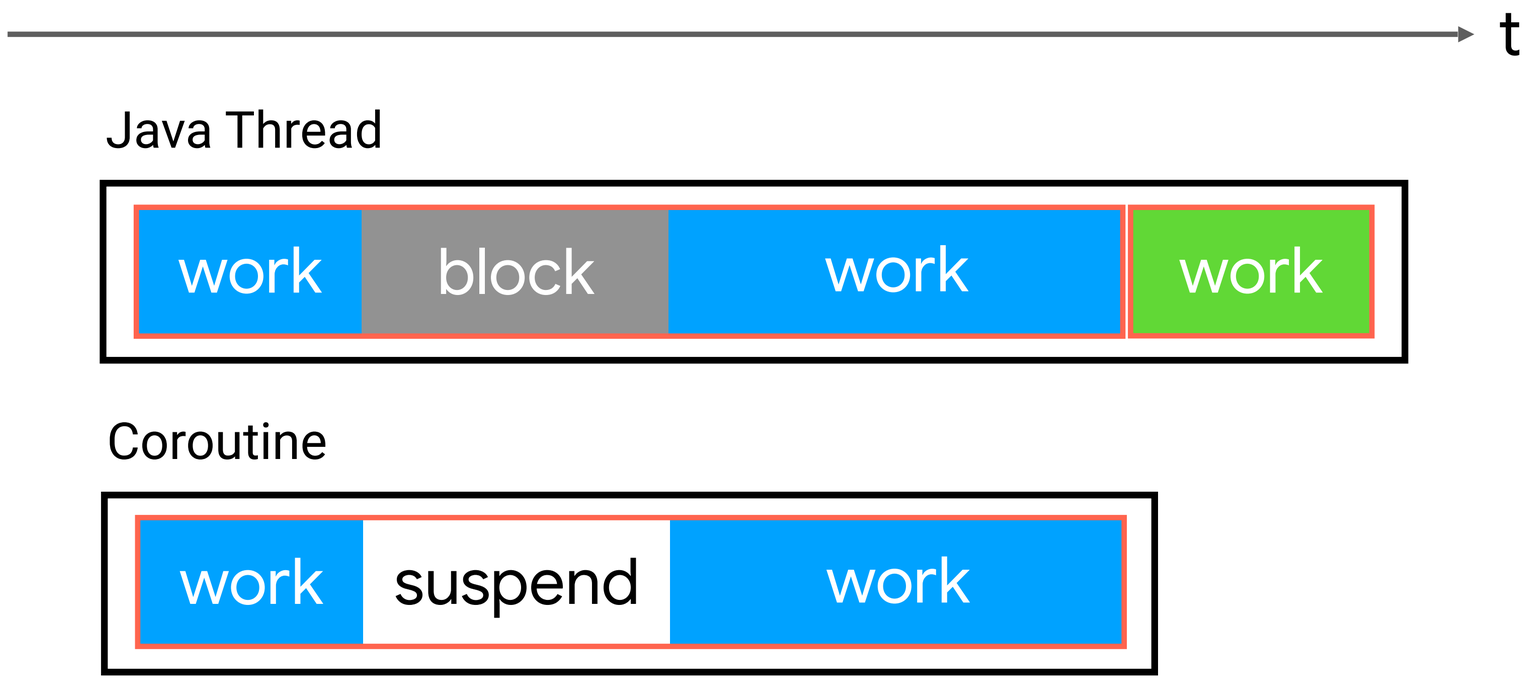
suspend vs blocking



Long Task

Thread

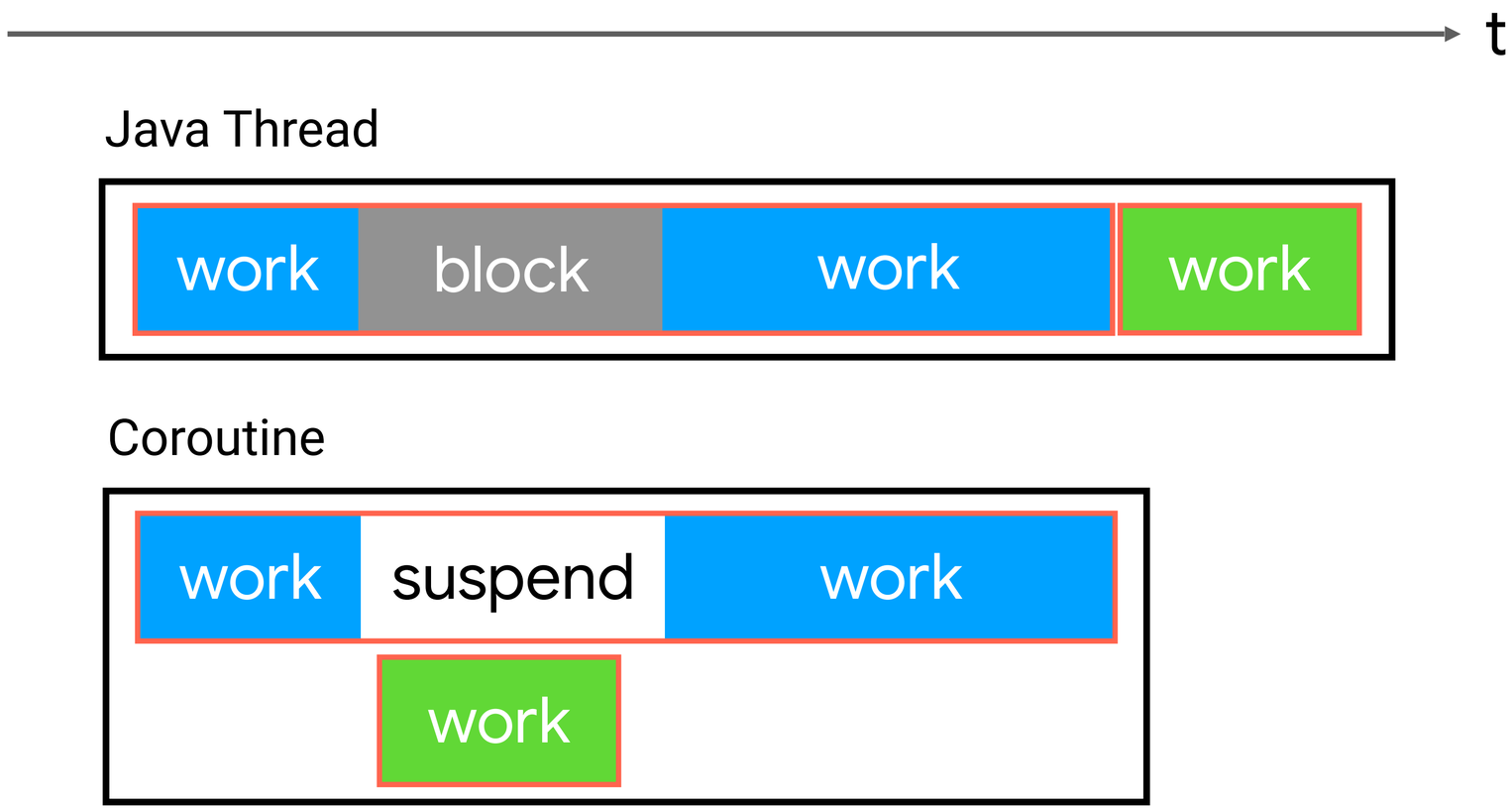
suspend vs blocking



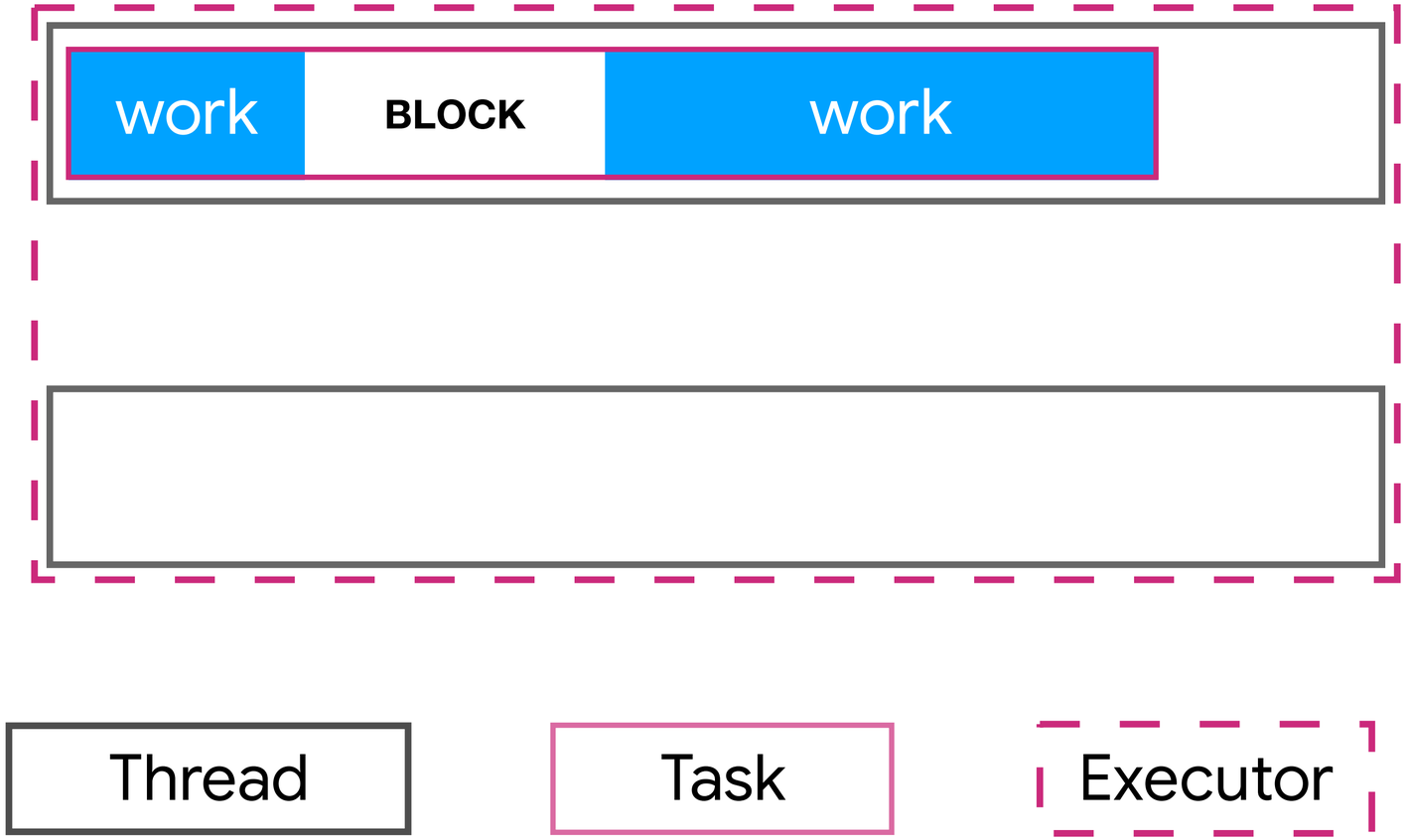
Long Task

Thread

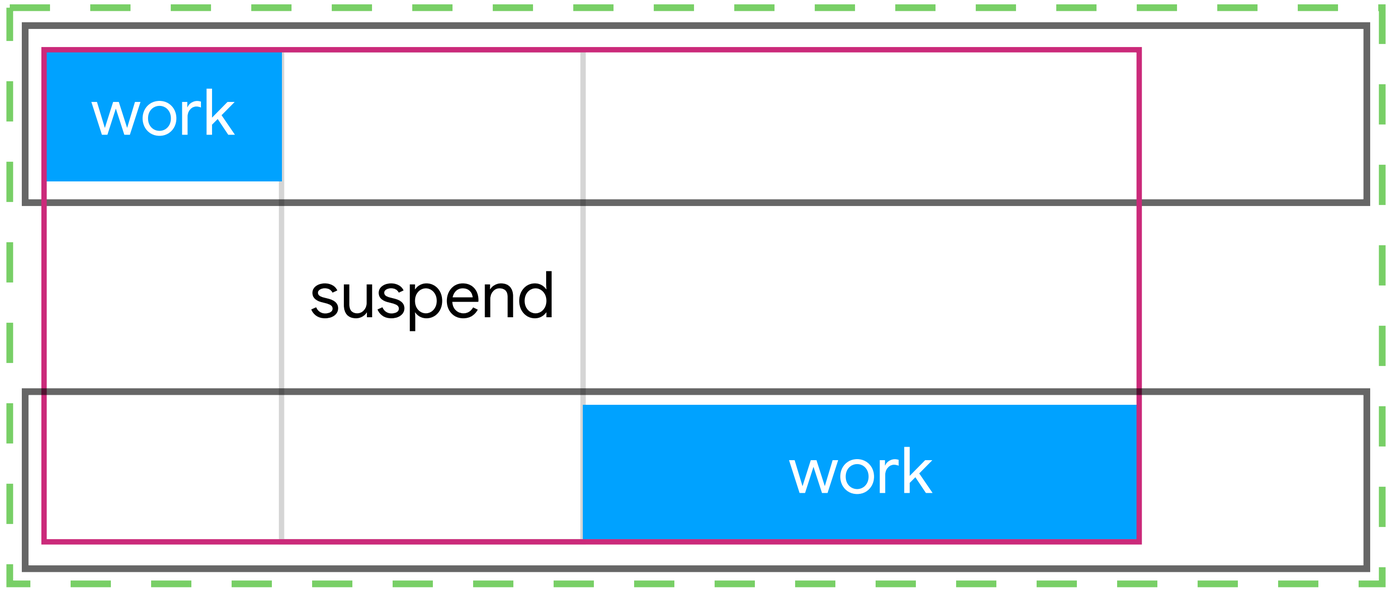
suspend vs blocking



Java Executors



Coroutine & thread reusing



Thread

Coroutine

Dispatcher

Strcutured Concurrency

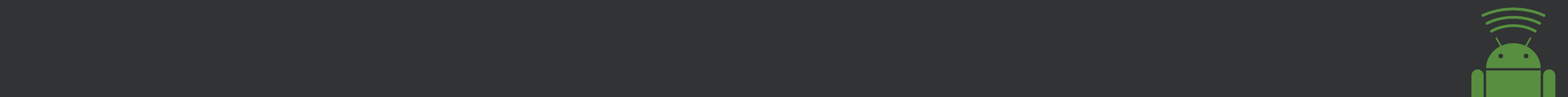
```
class MainActivity : AppCompatActivity() {

    private var asyncOperation: Async<Data>? = null

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        asyncOperation = runOperation()
    }

    override fun onDestroy() {
        super.onDestroy()
        asyncOperation?.cancel()
        asyncOperation = null
    }
}
```

RxJava Variant

```
class MainActivity : AppCompatActivity() {

    private val allDisposables = CompositeDisposable()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        val observable: Observable<Data> = runOperationObservable()
        val disposable = observable.subscribe()
        allDisposables.add(disposable)
    }

    override fun onDestroy() {
        super.onDestroy()
        if (!allDisposables.isDisposed) allDisposables.dispose()
    }
}
```

Strcutured Concurrency

```
class MainActivity : AppCompatActivity() {

    private val coroutineScope = CoroutineScope(…)

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        coroutineScope.launch { runOperation() }
    }

    override fun onDestroy() {
        super.onDestroy()
        coroutineScope.cancel()
    }
}
```



Strcutured Concurrency

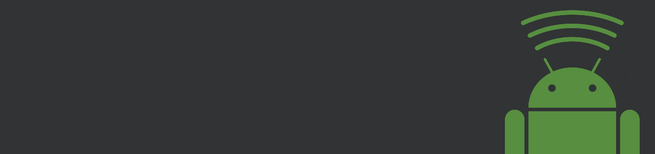
```
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        lifecycleScope.launch { runOperation() }
    }
}
```

Flow

```
launch {
    flowOf(1, 2, 3)
        .map { it * it }
        .collect { print(it) }
}
```

“Coroutines Flow” youtu.be/0ioDlsVyw1g

```
class Observable<T> {

    final Observable<T> filter(Predicate<? super T> predicate)

}
```



```java
public class ObservableFilter<T> extends AbstractObservableWithUpstream<T, T> {
    final Predicate<? super T> predicate;
    public ObservableFilter(ObservableSource<T> source, Predicate<? super T> predicate) {
        super(source);
        this.predicate = predicate;
    }

    public void subscribeActual(Observer<? super T> observer) {
        source.subscribe(new FilterObserver<T>(observer, predicate));
    }

    static class FilterObserver<T> extends BasicFuseableObserver<T, T> {
        final Predicate<? super T> filter;

        FilterObserver(Observer<? super T> actual, Predicate<? super T> filter) {
            super(actual);
            this.filter = filter;
        }

        public void onNext(T t) {
            if (sourceMode == NONE) {
                boolean b;
                try { b = filter.test(t); }
                catch (Throwable e) {
                    fail(e); return;
                }
                if (b) downstream.onNext(t);
            } else downstream.onNext(null);
        }

        public int requestFusion(int mode) { return transitiveBoundaryFusion(mode); }

        public T poll() throws Exception {
            for (;;) {
                T v = qd.poll();
                if (v == null || filter.test(v)) return v;
            }
        }
    }
}
```

```kotlin
inline fun <T> Flow<T>.filter(
    crossinline predicate: suspend (T) -> Boolean
): Flow<T> = flow {
    collect {
        if (predicate(it)) emit(it)
    }
}
```

```kotlin
inline fun <T> Flow<T>.filter(
    crossinline predicate: suspend (T) → Boolean
): Flow<T> = flow {
    collect {
        if (predicate(it)) emit(it)
    }
}
```

Flow Features

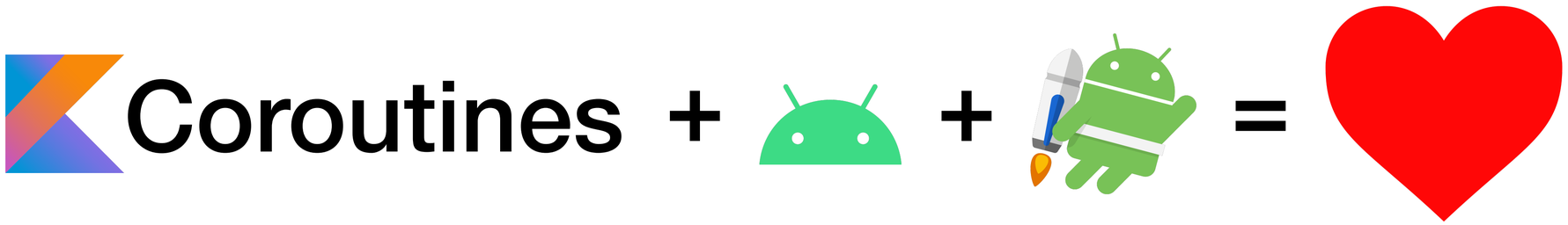
- Simple design
- Based on coroutines
- Have performance improvements during compilation
- No different between built-in and custom operators
- Easy to extend
- Compatibility with existed Reactive libraries (RxJava, Project Reactor) and easy migration
- Backpressure support

Kotlin Flow Plays Scrabble

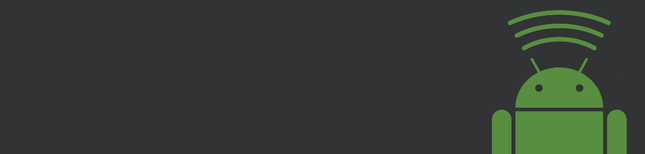
- Benchmark originally developed by José Paumard
- Implemented for RxJava by David Karnok

SequencePlaysScrabble	9.824 ± 0.190 ms/op
RxJava2PlaysScrabbleOpt	23.653 ± 0.379 ms/op
FlowPlaysScrabbleOpt	13.958 ± 0.278 ms/op

https://github.com/Kotlin/kotlinx.coroutines/tree/develop/benchmarks/src/jmh/kotlin/benchmarks/flow/scrabble/README.md


Coroutines + + =

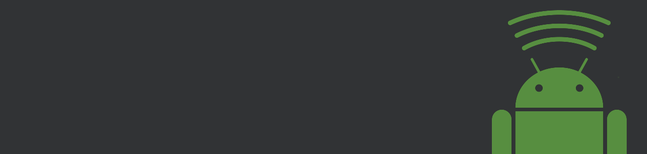
```
class SampleViewModel : ViewModel() {

    fun run() {
        viewModelScope.launch {
            // Your code here
        }
    }
}
```



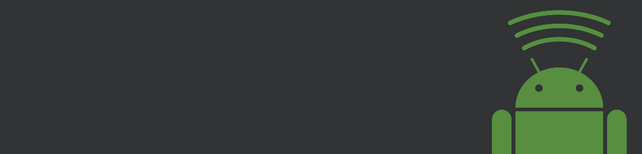
```
val ViewModel.viewModelScope: CoroutineScope
    get() {
        return CloseableCoroutineScope(
                    SupervisorJob() + Dispatchers.Main.immediate)
    }


class SampleViewModel : ViewModel() {

    fun run() {
        viewModelScope.launch {
            // Your code here
        }
    }
}
```

```
val ViewModel.viewModelScope: CoroutineScope
    get() {
        val scope: CoroutineScope? = this.getTag(JOB_KEY)
        if (scope ≠ null) return scope
        return setTagIfAbsent(JOB_KEY,
                CloseableCoroutineScope(
                    SupervisorJob() + Dispatchers.Main.immediate
                )
            )
    }
```

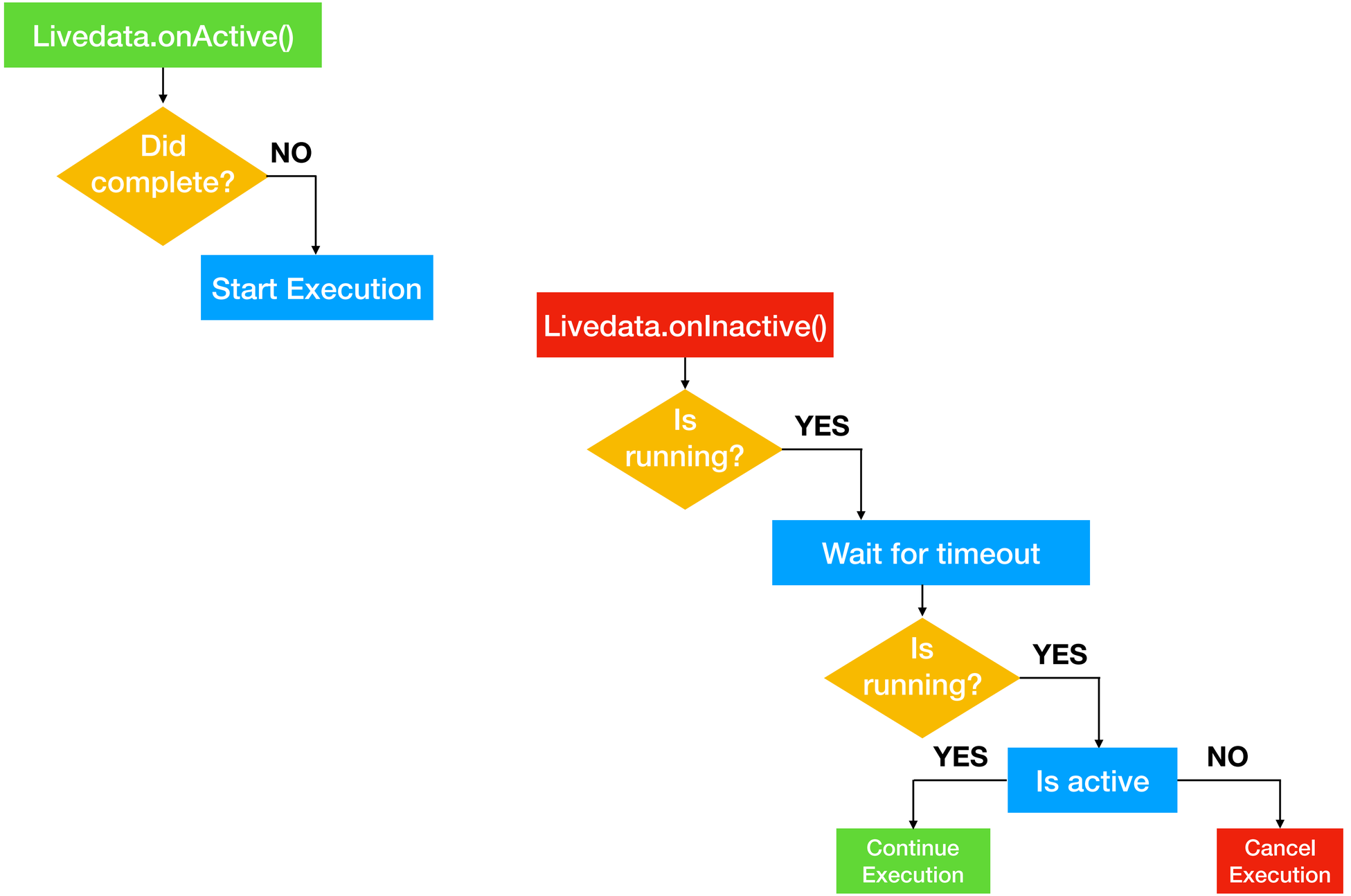
```
val liveData = liveData {
    val value = suspendFun()
    emit(value)
}
```



```kotlin
fun <T> liveData(
    context: CoroutineContext = EmptyCoroutineContext,
    timeoutInMs: Long = DEFAULT_TIMEOUT, // Default value = 5 sec
    block: suspend LiveDataScope<T>.() -> Unit
): LiveData<T> = CoroutineLiveData(context, timeoutInMs, block)
```

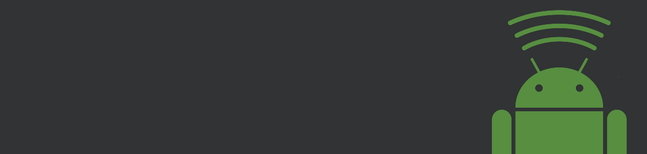


```
fun <T> liveData(
    context: CoroutineContext = EmptyCoroutineContext,
    timeoutInMs: Long = DEFAULT_TIMEOUT, // Default value = 5 sec
    block: suspend LiveDataScope<T>.() -> Unit
): LiveData<T> = CoroutineLiveData(context, timeoutInMs, block)


interface LiveDataScope<T> {

    suspend fun emit(value: T)

    suspend fun emitSource(source: LiveData<T>): DisposableHandle

    val latestValue: T?
}
```


Livedata.onActive()
Did complete?
NO
Start Execution
Livedata.onInactive()
Is running?
YES
Wait for timeout
Is running?
YES
YES
Is active
NO
Continue Execution
Cancel Execution

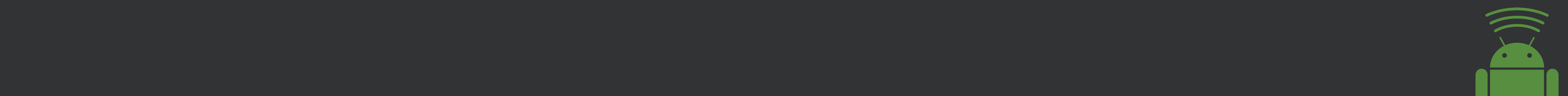
Sample 1

```
val userId: LiveData<String> = …
val user: LiveData<Data> = userId.switchMap { id →
    liveData {
        while (true) {
            val data = api.fetch(id)
            emit(data)
            delay(30_000)
        }
    }
}
```



Sample 2: Retrying with back-off

```
liveData {
    var backOffTime: Long = 1_000
    var succeeded = false
    while(!succeeded) {
        try {
            emit(api.fetch(id))
            succeeded = true
        } catch(error : IOException) {
            delay(backOffTime)
            backOffTime *= (backOffTime * 2).coerceAtLeast(60_000)
        }
    }
}
```



Sample 3

```
liveData {
    emit(LOADING(id))
    val cached = cache.loadUser(id)
    if (cached ≠ null) {
        emit(cached)
    }

    if (cached == null) {
        val fresh = api.fetch(id)
        cache.save(fresh)
        emit(fresh)
    }
}
```

Sample 4

```
liveData {
    val fromDb: LiveData<User> = roomDatabase.loadUser(id)
    emitSource(fromDb)
    val updated = api.fetch(id)
    roomDatabase.insert(updated)
}
```

LiveData Adapters

```
fun <T> Flow<T>.asLiveData(
    context: CoroutineContext = EmptyCoroutineContext,
    timeoutInMs: Long = DEFAULT_TIMEOUT
)

fun <T> LiveData<T>.asFlow(): Flow<T>
```

Lifecycle Coroutines Extensions

```
suspend fun <T> Lifecycle.whenCreated(block: suspend CoroutineScope.() → T): T

suspend fun <T> Lifecycle.whenStarted(block: suspend CoroutineScope.() → T): T

suspend fun <T> Lifecycle.whenResumed(block: suspend CoroutineScope.() → T): T


val Lifecycle.coroutineScope: LifecycleCoroutineScope
```

Lifecycle Coroutines Extensions

```
abstract class LifecycleCoroutineScope : CoroutineScope {
    internal abstract val lifecycle: Lifecycle

    fun launchWhenCreated(block: suspend CoroutineScope.() → Unit): Job =
        launch { lifecycle.whenCreated(block) }

    fun launchWhenStarted(block: suspend CoroutineScope.() → Unit): Job =
        launch { lifecycle.whenStarted(block) }

    fun launchWhenResumed(block: suspend CoroutineScope.() → Unit): Job =
        launch { lifecycle.whenResumed(block) }
}
```

Sample

```
class SampleFragment : Fragment() {

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)
        viewLifecycleScope.launchWhenStarted {
            val location = locationProvider.getCurrentLocation()
            drawLocationOnMap(location)
        }
    }
}

private interface LocationProvider {
    suspend fun getCurrentLocation(): Location
}
```

Room

```
@Dao
interface UsersDao {

    @Query("SELECT * FROM users")
    suspend fun getUsers(): List<User>

    @Query("UPDATE users SET age = age + 1 WHERE userId = :userId")
    suspend fun incrementUserAge(userId: String)

    @Insert suspend fun insertUser(user: User)

    @Update suspend fun updateUser(user: User)

    @Delete suspend fun deleteUser(user: User)
}
```

d.android.com/training/data-storage/room

Room

```
@Dao
interface UsersDao {

    @Query("SELECT * FROM users")
    fun observeUsers(): Flow<List<User>>
}
```

d.android.com/training/data-storage/room

Work Manager

```
class CoroutineDownloadWorker(
    context: Context,
    params: WorkerParameters
) : CoroutineWorker(context, params) {

    override suspend fun doWork(): Result = coroutineScope {
        val jobs = (1 until 100).map { index ->
            async {
                download("https://t.me/android_broadcast/$index")
            }
        }

        jobs.awaitAll()
        Result.success()
    }
}
```

Arch Components KTX Artifacts

- androix.lifecycle:**lifecycle-livedata-core-ktx**
- androix.lifecycle:**lifecycle-livedata-ktx**
- androix.lifecycle:**lifecycle-runtime-ktx**
- androix.lifecycle:**lifecycle-viewmodel-ktx**

- androidx.room:**room-ktx**
- androidx.work:**work-runtime-ktx**

- androidx.paging:**paging-common-ktx**
- androidx.paging:**paging-runtime-ktx**

CoIL (Coroutine Image Loader)

```
viewLifecycleScope.launch {
    Coil.get("https://www.example.com/image.jpg") {
        memoryCachePolicy(CachePolicy.DISABLED)
        diskCachePolicy(CachePolicy.DISABLED)
        transformations(CircleCropTransformation())
    }
}
```

coil-kt.github.io/coil/

FlowBinding

```
val button: View = view.findViewById(R.id.send)
val clicks: Flow<Unit> = button.clicks()
clicks.onEach { viewModel.send() }
    .launchIn(fragment.viewLifecycleOwner.lifecycleScope)
```

reactivecircus.github.io/FlowBinding/

FlowBinding

```
val button: View = view.findViewById(R.id.send)
viewLifecycleOwner.lifecycle.coroutineScope.launchWhenStarted {
    button.clicks()
      .collect { viewModel.send() }
}
```

reactivecircus.github.io/FlowBinding/

Retrofit

```
interface GitHubService {

    @GET("users/{user}/repos")
    suspend fun listRepos(@Path("user") user: String): List<Repo>
}
```

square.github.io/retrofit

Async Single Value

```
interface Async<T> {

    fun execute(c: Callback<T>)

    interface Callback<T> {

        fun onComplete(value: T) {}

        fun onCanceled() {}

        fun onError(error: Throwable) {}
    }
}
```

Async Single Value

```
suspend fun <T> Async<T>.await(): T =
    suspendCoroutine { cont: Continuation<T> →
        execute(object : Async.Callback<T> {

            override fun onComplete(value: T) {
                cont.resume(value)
            }

            override fun onError(error: Throwable) {
                cont.resumeWithException(error)
            }
        })
    }
```

Async Single Value

```
suspend fun <T> Async<T>.await(): T =
    suspendCancellableCoroutine { cont: CancellableContinuation<T> →
        execute(object : Async.Callback<T> {

            override fun onComplete(value: T) {
                cont.resume(value)
            }

            override fun onError(error: Throwable) {
                cont.resumeWithException(error)
            }

            override fun onCanceled() {
                cont.cancel()
            }
        })
    }
```

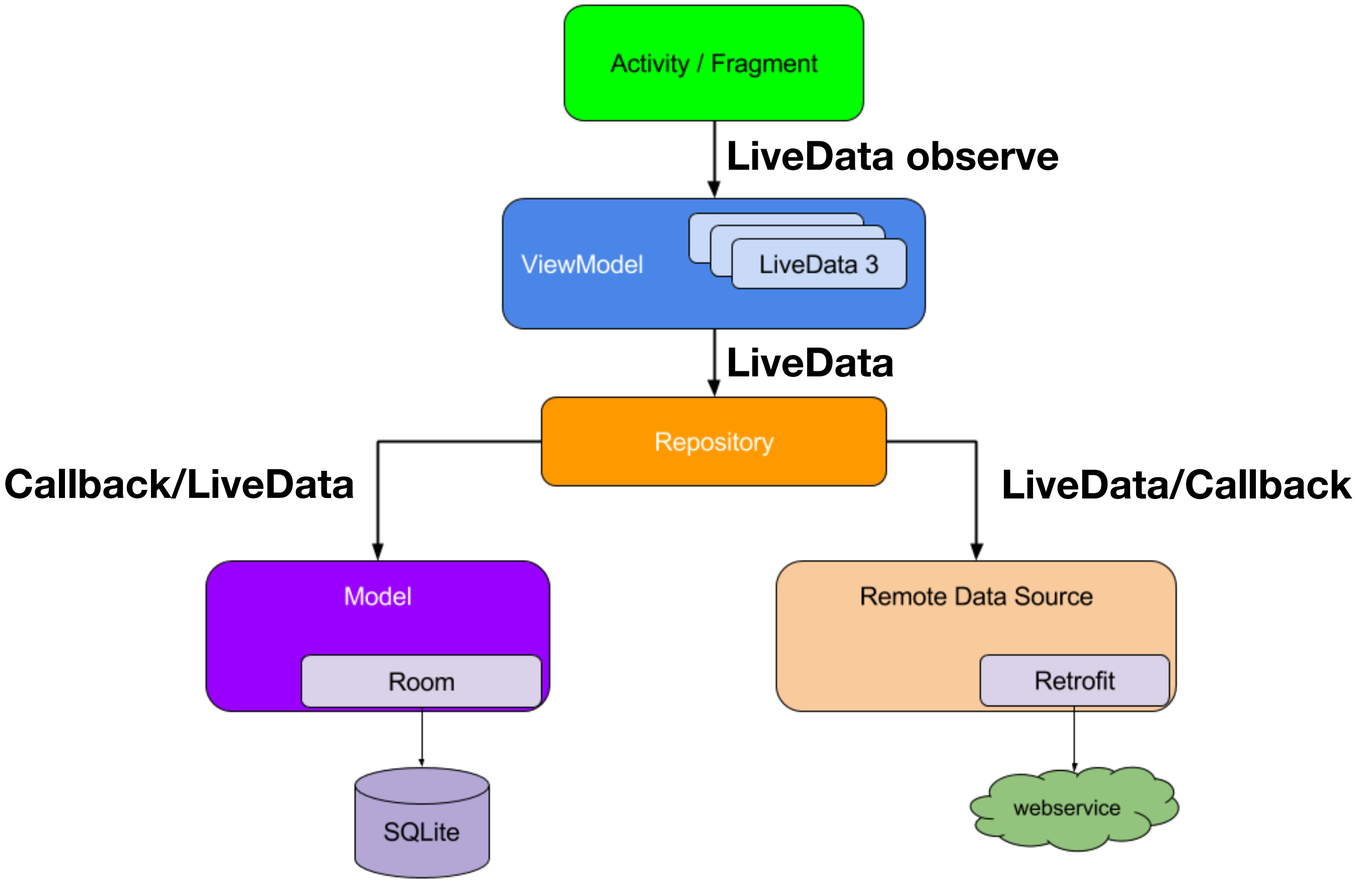
Stream

```
interface Stream<T> {

    fun subscribe(c: Callback<T>)

    fun unsubscribe(c: Callback<T>)

    interface Callback<T> {

        fun onNext(item: T)

        fun onComplete()

        fun onCanceled()

        fun onError(error: Throwable)
    }
}
```

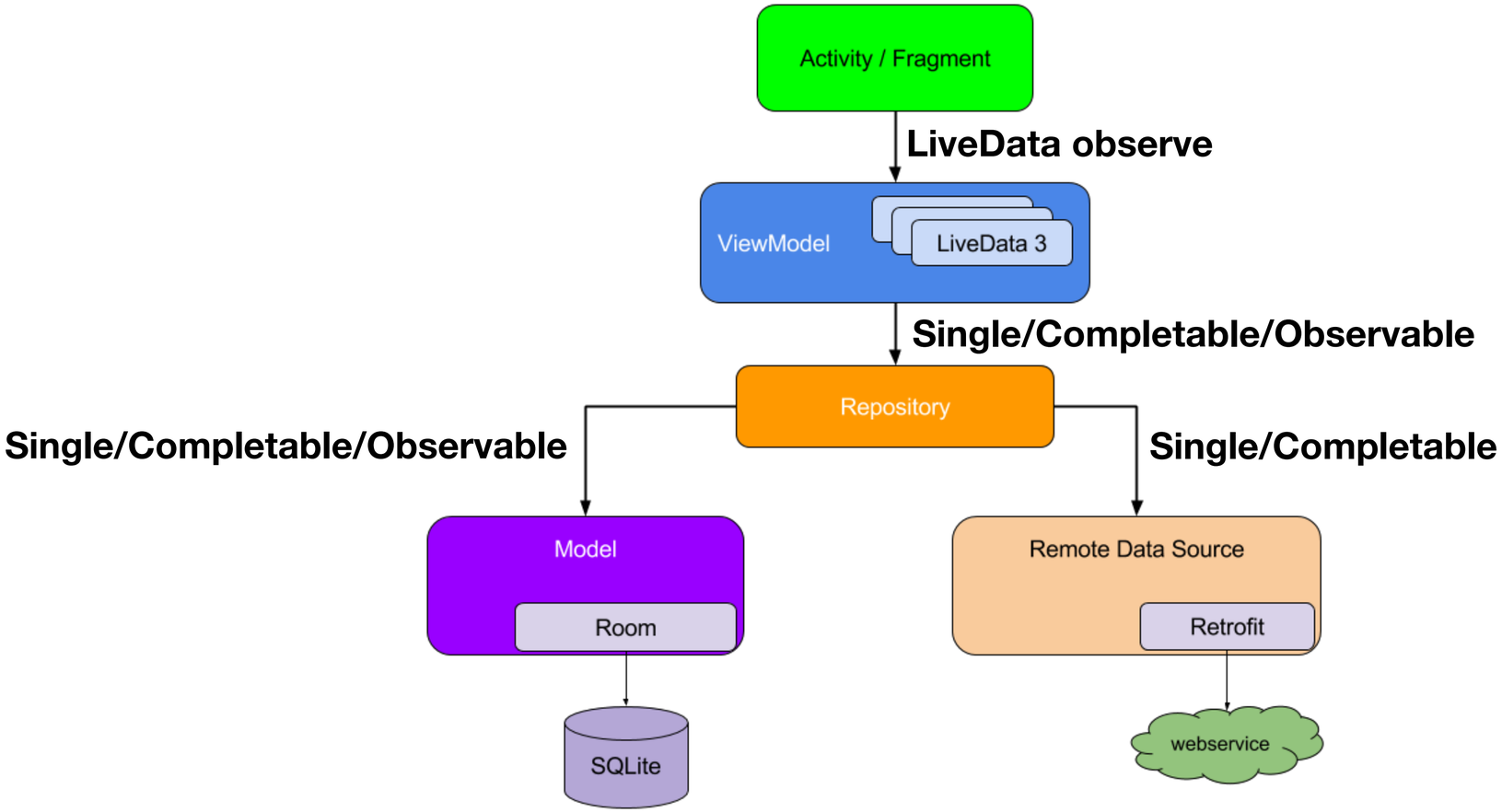
Stream

```
fun <T> Stream<T>.asFLow(): Flow<T> = callbackFlow { this: ProducerScope<T>
    val callback = object : Stream.Callback<T> {

        override fun onNext(item: T) { offer(item) }

        override fun onComplete() { close() }

        override fun onError(error: Throwable) { close(error) }

        override fun onCanceled() { cancel() }
    }
    subscribe(callback)
    invokeOnClose { unsubscribe(callback) }
}
```

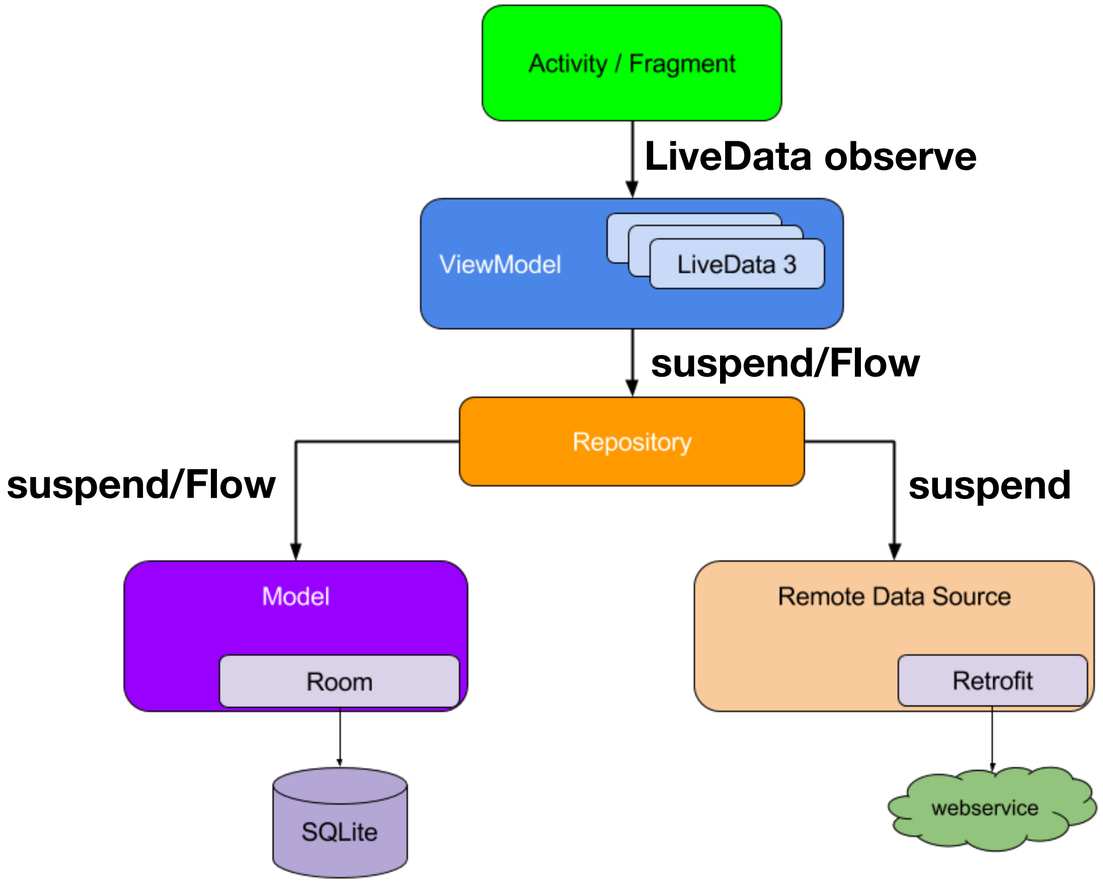
Recommended App Architecture by Google



RxJava + LiveData



Coroutines + Arch Component



Status & Roadmap

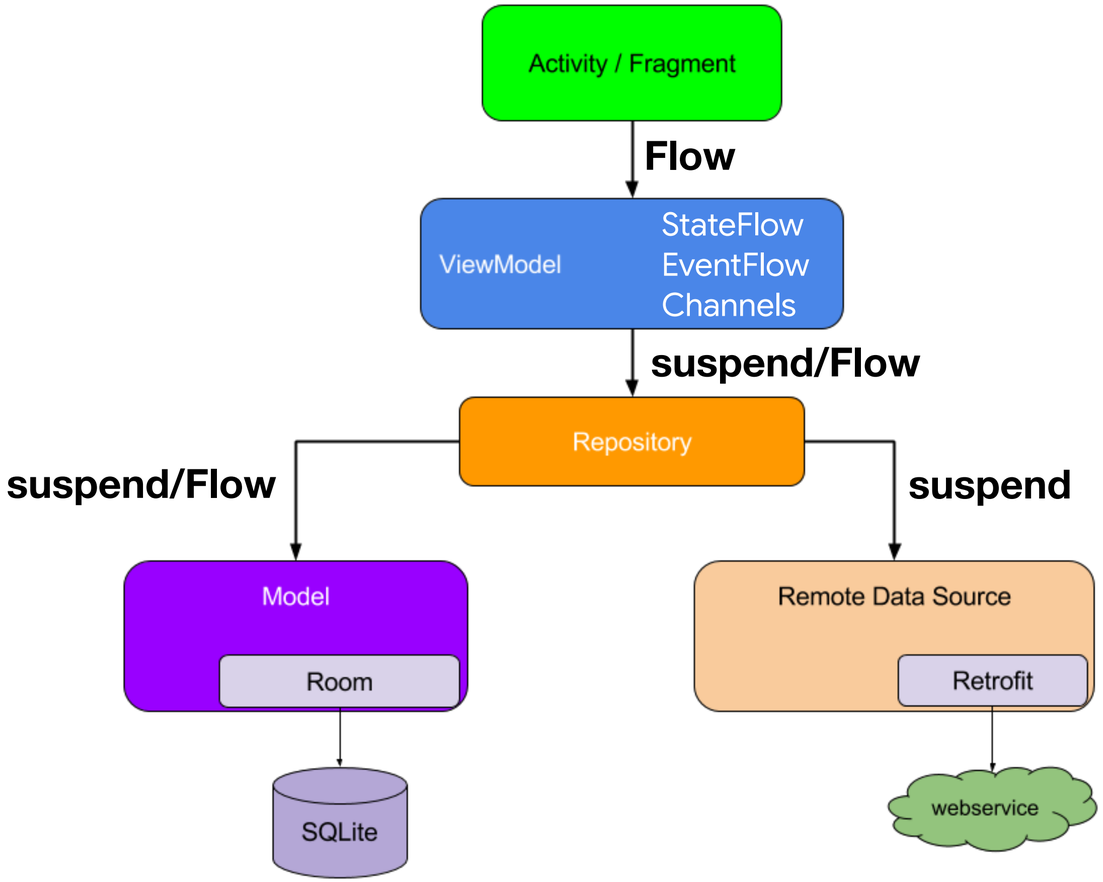
🚀 Flow is stable in kotlinx.coroutines version 1.3.0

🔮 Future improvements

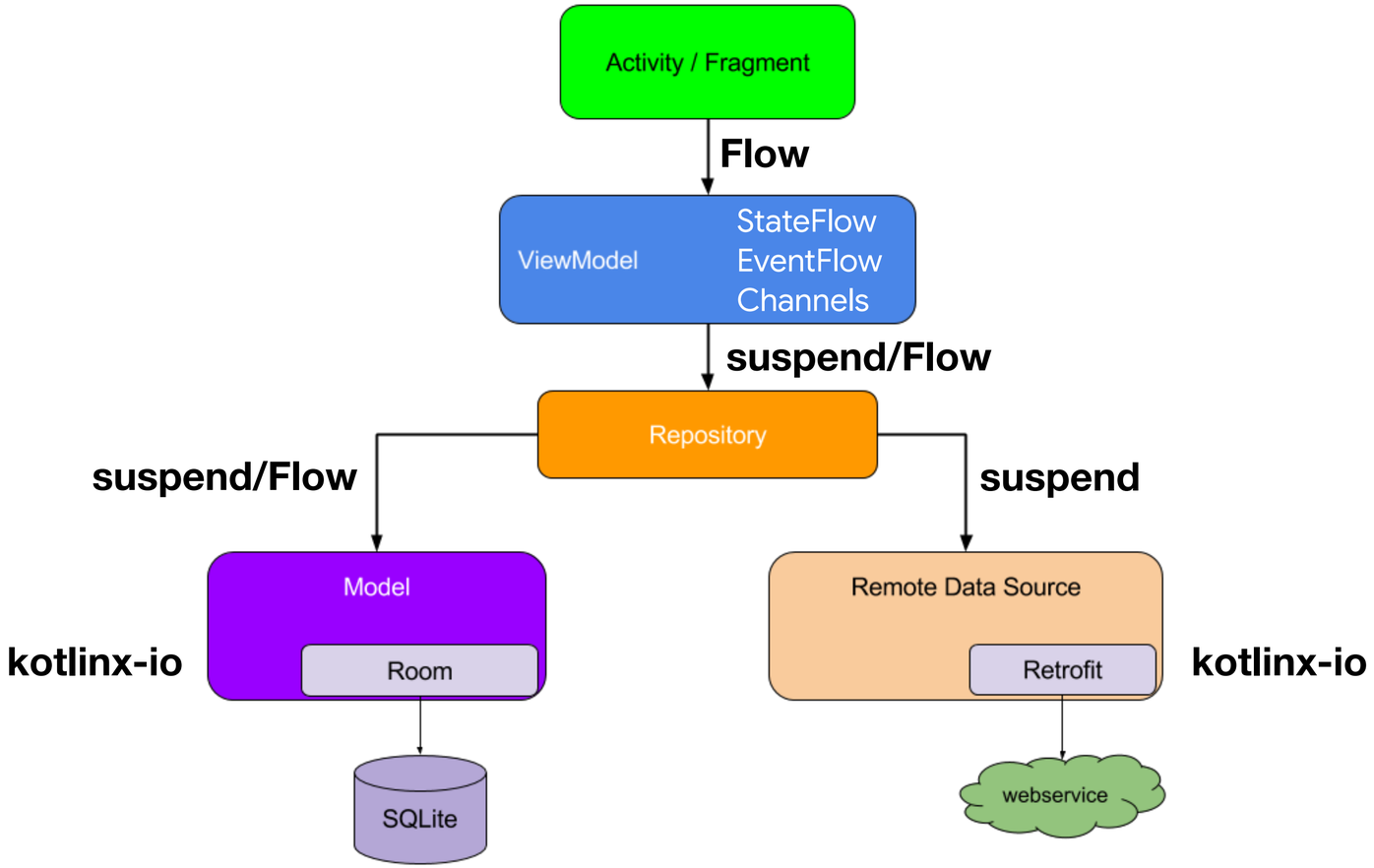
- Out-of-the box support for UI models (StateFlow / EventFlow)
- Sharing / caching flows
- Concurrency / parallelism operators
- Chunking / windowing operators

🦄 Want more? Give us your feedback
https://github.com/Kotlin/kotlinx.coroutines/issues

Full ~~House~~ Coroutines



Full ~~House~~ Coroutines



Summarize

- “Kotlin First” now, “Kotlin Coroutines First” is ringing in the door
- Amazing integration with Kotlin
- Kotlin effectively works for devices with limited CPU and multiple operations
- Kotlin MultiPlatform Projects support
- RxJava 3 has no serious evolution
- Waiting for kotlinx.io release for async I/O operations based on Coroutines

Thank you!

krl.rozov@gmail.com

@kirill_rozov

@kirillr