

Многопоточность Advanced

Павел Галанин
Весна 2024



Java Memory Model

• • • •

Зачем понадобилась Java Memory Model?

Что напечатает? Варианты: 0, 41, 42, 43, <ничего>

```
int a = 0;
boolean ready = false;
—————
a = 41;    while(!ready);
a = 42;    println(a);
ready = true;
a = 43;
```

Напечатает или 0, или 41, или 42, или 43, или <ничего>.

Оптимизации: кэширование

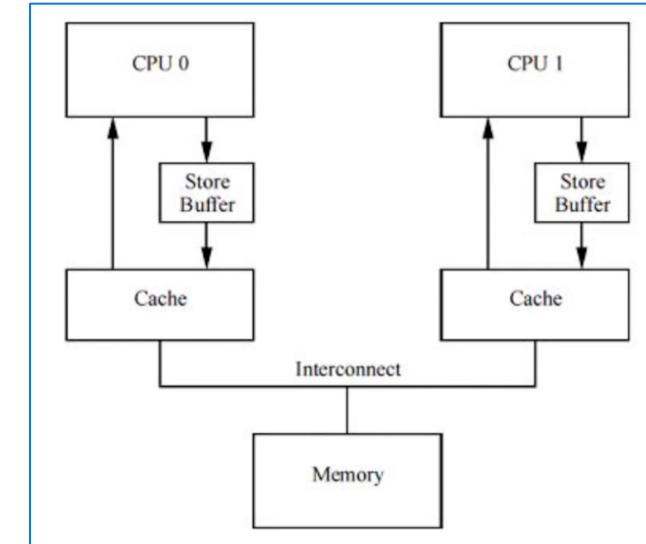
Для ядер работать всегда напрямую с памятью - очень долго.

→ Ядра имеют собственные **кэши**, в которых хранятся «копии» данных из основной памяти. (Записи могут вообще удерживаться в регистрах процессора).

→ Могут быть окна, когда данные между кэшами и памятью отличаются.

В это время одни потоки не видят изменений, сделанных в других потоках.

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	25 ns
Main memory reference	100 ns
Send 1K bytes over 1 Gbps network	10,000 ns
Read 4K randomly from SSD	150,000 ns
Read 1 MB sequentially from memory	250,000 ns
Read 1 MB sequentially from SSD	1,000,000 ns
Read 1 MB sequentially from disk	20,000,000 ns



Оптимизации: перестановки

Перестановки машинных инструкций:

- начать чтение раньше, чем по коду, потому что пока больше нечем заняться
- branch prediction + speculation - начать исполнение по одной из ветвей if/else до вычисления всех условий, по которому будет выбираться ветвь
- ...

Перестановки компилятора байткода:

- удаление промежуточных записей, если между ними не было чтения
- вынос вычислений за пределы цикла, если они не связаны с вычислениями в цикле
- перемещение инструкций под синхронизацией, чтобы их можно было использовать с одним локом
- ...

Зачем понадобилась Java Memory Model?

Что хотят разработчики программ?

- Писать мультипоточный код как однопоточный.
- Хоть бы видеть, что операции из разных потоков выстраиваются в последовательную цепочку:

```
int a = 0;  
boolean ready = false;  
a = 41;    while(!ready);  
a = 42;    println(a);  
ready = true;  
a = 43;
```

→ ожидаем видеть 42 или 43

Что хотят разработчики компиляторов и железа?

Максимально видоизменять код программы ради ускорения.

Модель памяти = trade-off между
долбанутостью программирования на языке,
долбанутостью быстрой и корректной реализации языка,
и долбанутостью хардвара

Java Memory Model

Java Memory Model - часть спецификации языка - определяет гарантии видимости изменений данных (другими потоками).

По сути, Java Memory Model даёт ответ на один вопрос:

Какие значения могут быть получены при выполнении чтения из памяти?

Чем оперирует Java Memory Model:

Операции:

- чтение/запись обычных переменных (read/write)
- чтение/запись volatile-переменных (volatile read/write)
- синхронизация (lock/unlock)

Переменные:

- static field
- instance field
- array element

Sequential Consistency, SC

Как хотелось бы:

Многопоточная программа работает как упорядочивание операций из разных потоков в одну последовательность. Операции не могут перемешиваться.

Можем ли сделать такую перестановку, не сломав SC?

```
int a = 0, b = 0;  
_____  
r1 = a;  
r2 = b;
```



```
int a = 0, b = 0;  
_____  
r2 = b;  
r1 = a;
```

Sequential Consistency, SC

Гарантия SC сильно урезает возможности оптимизации:

$$\frac{\text{int } a = 0, \ b = 0;}{\begin{array}{c|c} r1 = a; & b = 2; \\ r2 = b; & a = 1; \end{array}} \rightarrow \frac{\text{int } a = 0, \ b = 0;}{\begin{array}{c|c} r2 = b; & b = 2; \\ & a = 1; \\ r1 = a; & \end{array}}$$

- В исходной программе при SC обязательно либо « $r2 = b$ », либо « $a = 1$ » должно быть последним, а значит, $(r1, r2)$ либо $(*, 2)$, либо $(0, *)$.
- Новая программа приводит к $(r1, r2) = (1, 0)$

→ нужна более слабая модель

Действия синхронизации Synchronization actions, SA

Определяем специальные
операции - **действия
синхронизации**:

Synchronization Actions (SA):

- volatile read, volatile write
- lock monitor, unlock monitor
- первое и последнее действие в потоке
- операции, обнаруживающие завершение потока
(`Thread.join()`, `Thread.isInterrupted()` и т.п.)

Synchronization Order, SO

Synchronization Order: порядок гарантируется **не для всех**, а только для отдельных операций - **действий синхронизации**.

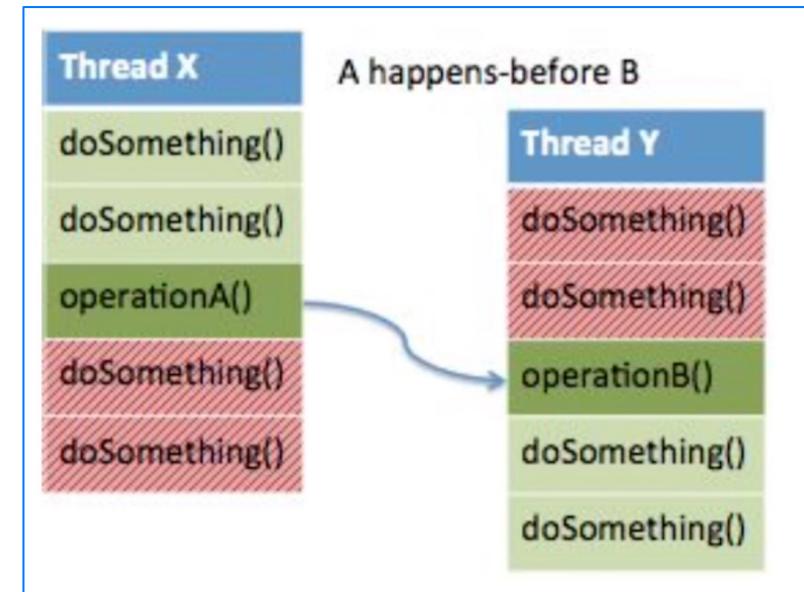
Все трэды видят один и тот же порядок между действиями синхронизации.

Но не между любыми действиями синхронизации, а по некоторым правилам 😊

Happens-before, HB

Если X happens-before Y, то Y будет видеть результат X.

- Program order: каждое действие потока happens-before любого действия, которое идёт позже по порядку программы в этом потоке.
 - Правило **блокировки монитора**: разблокировка монитора happens-before любого последовательного получения блокировки на **этом же мониторе**.
 - Правило **volatile-переменной**: запись в volatile-поле happens-before последующего чтения **этого же поля**.
 - **Транзитивность**: если A happens-before Б, а Б happens-before В, то A happens-before В
-
- Правило запуска потока: вызов Thread.start happens-before любого действия запускаемого потока.
 - Правило завершения потока: любое действие в потоке happens-before того, как другой поток обнаруживает, что поток завершён.
 - Правило прерывания: вызов interrupt для потока происходит до того, как прерываемый поток обнаруживает прерывание
 - Правило финализации: окончание конструирования объекта happens-before начала финализации объекта.

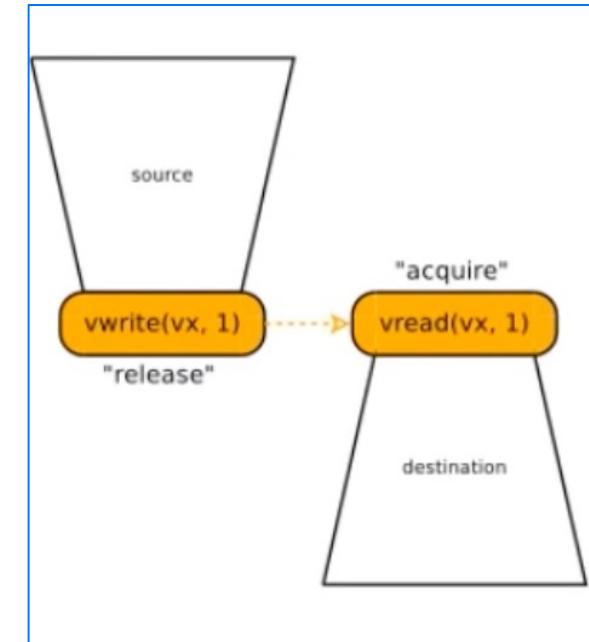
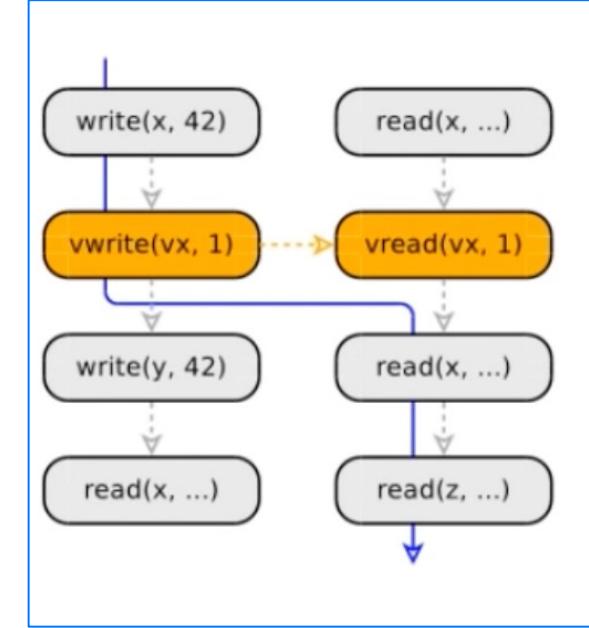


Happens-before, HB

Правило happens-before даёт ответ на тот самый основной вопрос - что может быть прочитано:

- последняя запись в happens-before
- или
- любая запись вне happens-before (через гонку)

→ [Гонка](#) - чтение переменной, которая была записана вне happens-before.



Roach Motel

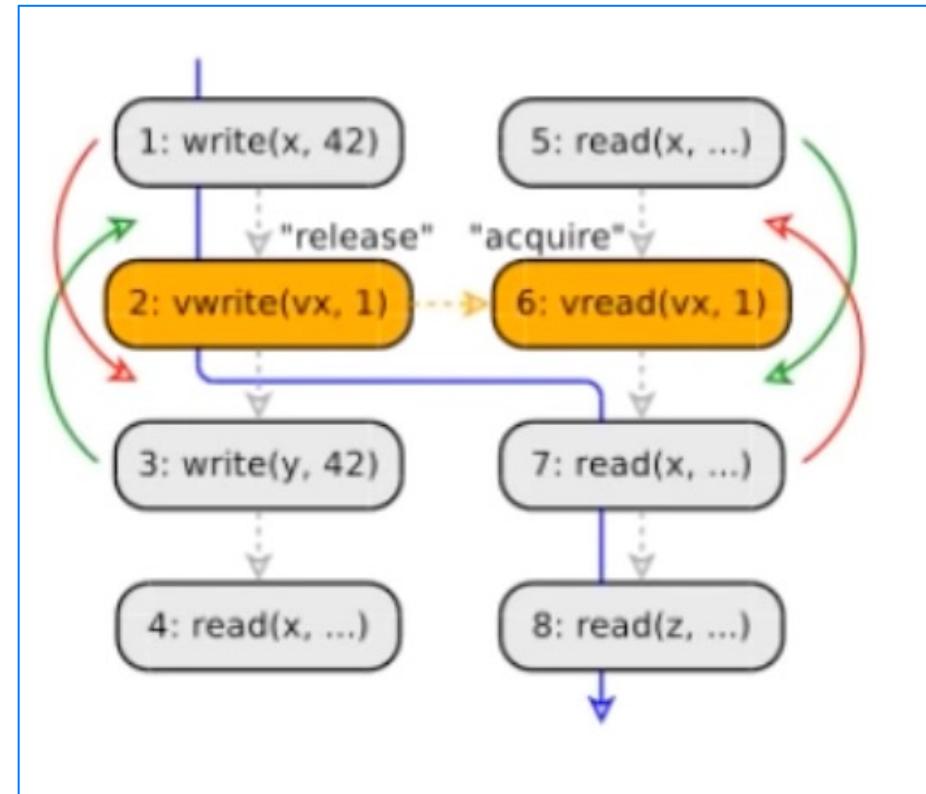
Roach Motel - оптимизация, не нарушающая happens-before.

Запрещены перестановки:

- (1) → после release: второй поток уже не увидит изменение через happens-before
- (7) → перед acquire: второй поток ещё не увидит изменение через happens-before

Можно сделать перестановки:

- (5) → после acquire: читает x через гонку - но последнее до release значение может быть увидено
- (3) → перед release: чтения справа могут увидеть эту запись через гонку, не мешает зависимостям по x

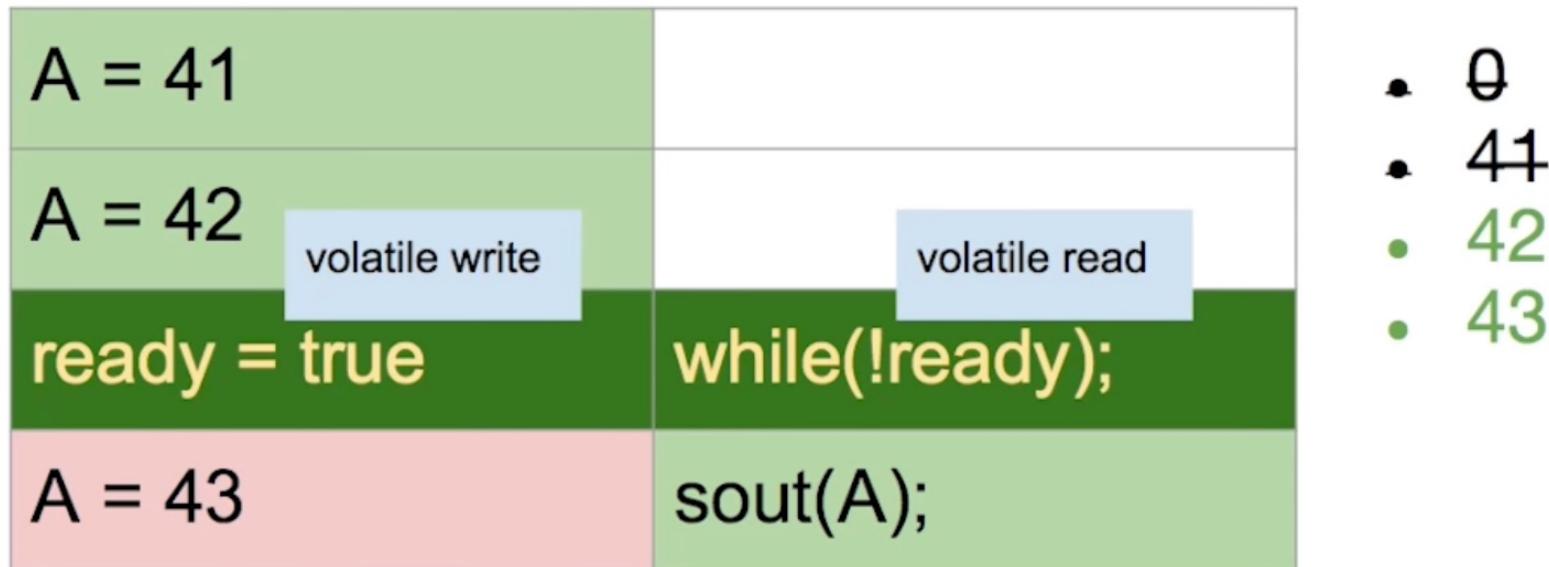


Synchronization Order

Что напечатает? Варианты: 0, 41, 42, 43, <ничего>

int a = 0; <hr/> a = 41; a = 42; ready = true; a = 43;	volatile boolean ready = false; <hr/> while(!ready); println(a);
--	--

Synchronization Order



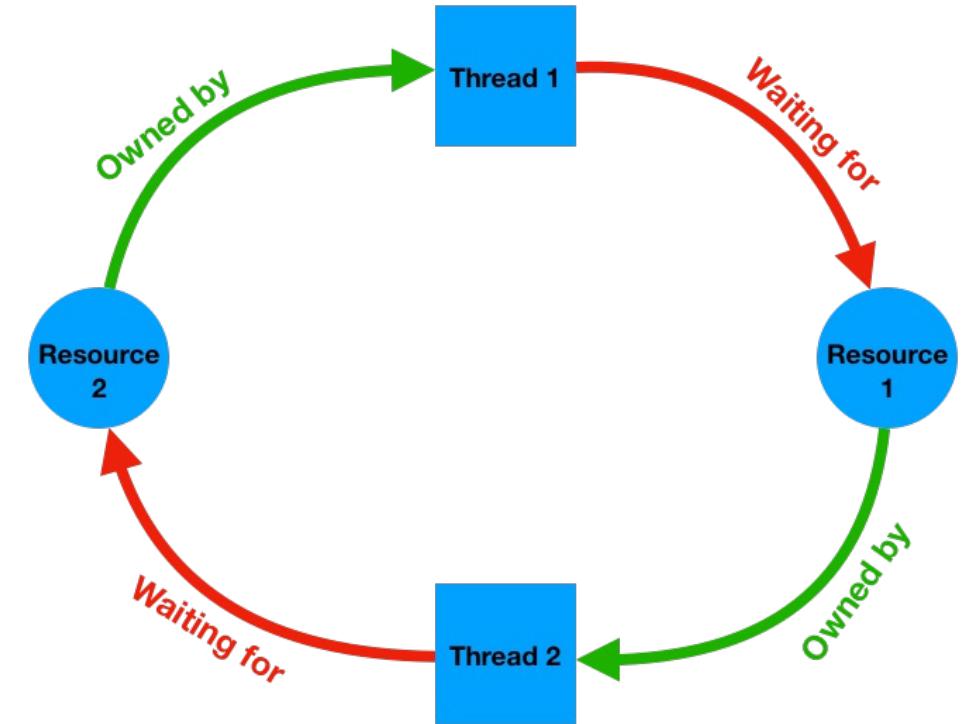
Проблемы многопоточности

• • • •

Deadlock - взаимная блокировка

Как бороться:

1. Устанавливать глобальный порядок получения блокировок потоками.
2. Устанавливать таймаут на удержание блокировок.

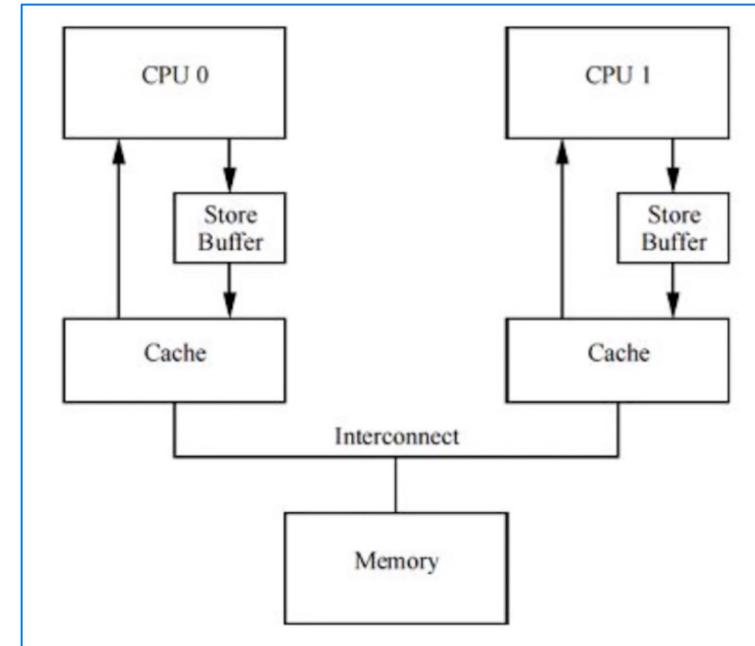


Проблема видимости

Из-за кеширования в ядре (или других оптимизаций) поток может не увидеть новое значение переменной, установленное другим потоком.

Как бороться:

1. Синхронизировать чтение и запись **общей блокировкой**.
2. Использовать **volatile-переменные** - для них гарантируется видимость изменений из других потоков.

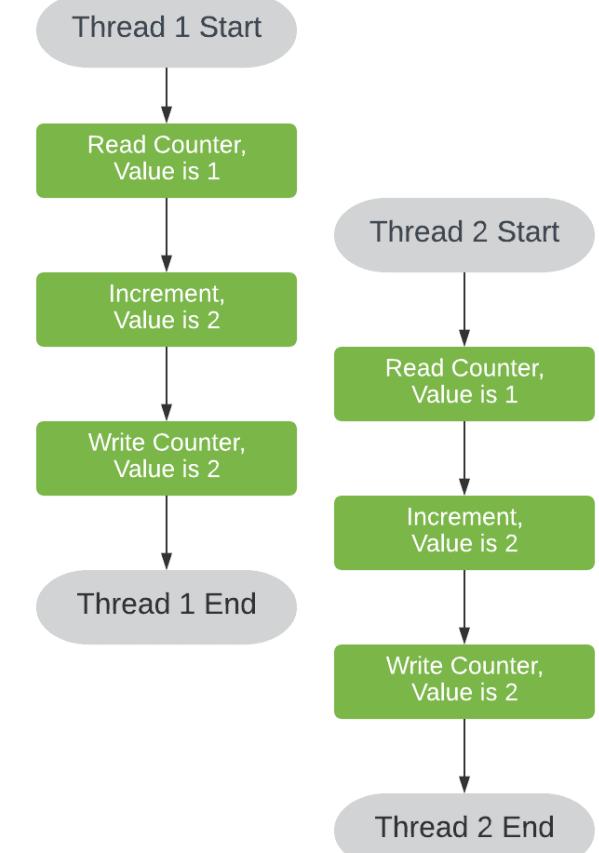


Race condition

Состояние гонки ([race condition](#)) - ситуация, когда корректность вычисления зависит от относительного времени исполнения потоков или образа их взаимодействия – то есть от стечения обстоятельств.

Состояние гонки часто возникает в случаях проверка-потом-действие ([check-then-act](#)), когда потенциально устаревшее наблюдение используется для принятия решения о дальнейших действиях

Каноничный пример - инкремент счётчика из разных потоков.



Как избежать гонок

- **Volatile** 

Операции А и Б являются атомарными относительно друг друга, если при наблюдении со стороны потока, исполняющего А, когда другой поток исполняет Б, – либо все команды операции Б выполнены, либо ни одна из них

- **Locks / synchronized**
- **Atomic-обёртки** 

Atomic-классы

`java.util.concurrent.atomic:`

`AtomicInteger`

`AtomicLong`

`AtomicBoolean`

`AtomicReference`

`AtomicIntegerArray`

`AtomicLongArray`

`AtomicReferenceArray`

В большинстве процессорных архитектур реализованы через инструкцию [compare-and-swap \(CAS\)](#).

CAS имеет три операнда: адрес в памяти, ожидаемое прежнее значение и новое значение. Если прежнее значение отличается от ожидаемого, значение в памяти не будет изменено на новое. В любом случае будет возвращено то значение, которой фактически находится в памяти по указанному адресу.

CAS реализует [оптимистичную](#) технику: “надеется” на успех, но способна обнаружить несоответствие. Когда несколько потоков пытается изменить значение переменной с использованием CAS, один из них побеждает и записывает своё значение. Другим потокам сообщается, что значение было изменено, и они не блокируются, а могут повторить запись, выполнить некоторое запланированное действие или ничего не делать.

Синхронизаторы



ПРИМИТИВЫ СИНХРОНИЗАЦИИ: МОНИТОР

Каждый Java-объект содержит методы:

```
wait(...)  
notify()  
notifyAll()
```

```
6 usages  
private int count = 0;  
  
1 usage  
public synchronized void increment() {  
    count++;  
    System.out.println("Increased to: " + count);  
    notifyAll();  
}  
  
1 usage  
public synchronized void decrement() {  
    while (count == 0) {  
        try {  
            wait();  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
    count--;  
    System.out.println("Decreased to: " + count);  
}
```

Synchronized

Позволяет создать [критическую секцию](#)
- пока один из потоков исполняет код в
этой секции, остальные потоки ждут
(блокируются).

В качестве средства блокировки
используется монитор текущего
объекта.

Для «статических» методов берётся
монитор объекта Class для текущего
класса.

```
@Synchronized
fun increment() {
    println("Do some work by one thread at a time")
}
```

Семафор

Семафор - позволяет задать максимальное количество потоков, получающих доступ к ресурсу единовременно. Остальные потоки будут ожидать, пока какой-то из других потоков не освободит ресурс.

`java.util.concurrent.Semaphore:`

- `acquire()`
- `tryAcquired()`
- `release()`

Семафор с параметром 1 образует [mutex \(MUTual Exclusion\)](#) - только один поток получает доступ к ресурсу, т.е. создаётся критическая секция (как с `synchronized`).

В Java нет отдельного класса для Mutex.

```
class ParkingLot(spots: Int) {  
    private val parkingSpots = Semaphore(spots)  
  
    fun park(carName: String) {  
        try {  
            println("$carName is trying to park.")  
            parkingSpots.acquire()  
            println("$carName parked.")  
            Thread.sleep((Math.random() * 1000).toLong())  
        } catch (e: InterruptedException) {  
            e.printStackTrace()  
        } finally {  
            parkingSpots.release()  
            println("$carName left the parking.")  
        }  
    }  
}
```

CountDownLatch

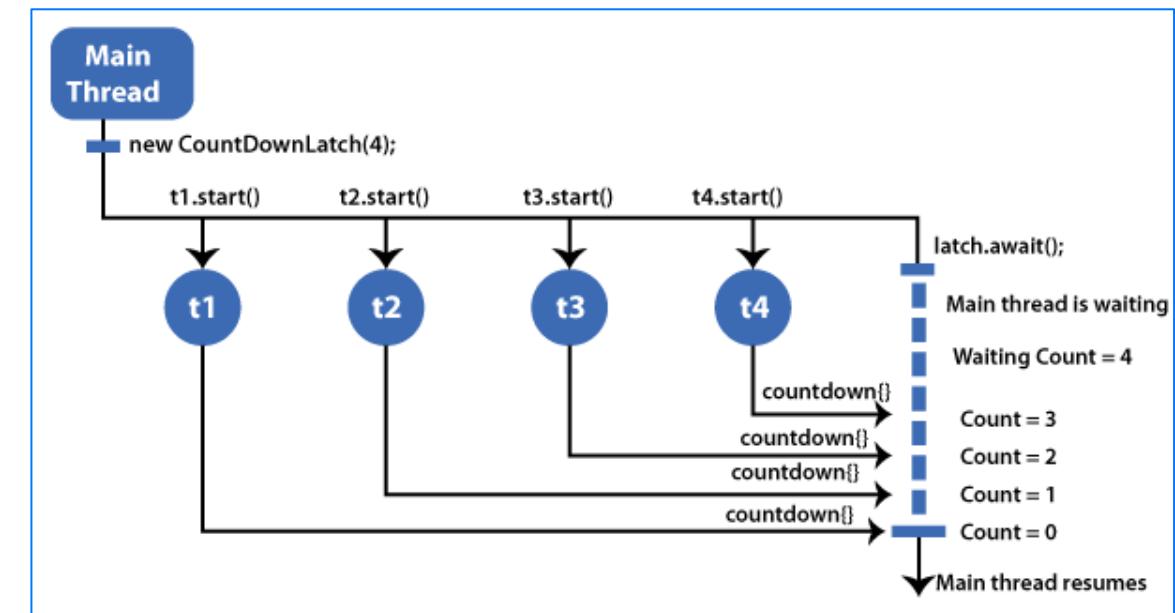


Защёлка ([latch](#), иногда её называют [воротами](#)) блокирует потоки до наступления некоторого события (терминального состояния). После наступления этого события все потоки продолжают выполнение. Защёлка не может быть использована повторно.

С помощью защёлки удобно реализовывать ожидание инициализации ресурса(ов).

java.util.concurrent.CountDownLatch

- параметр `count` - количество ожидаемых событий
- `await()`
- `countDown()`

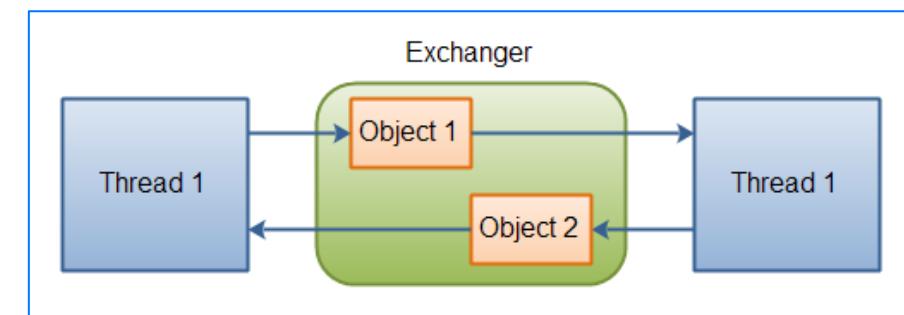
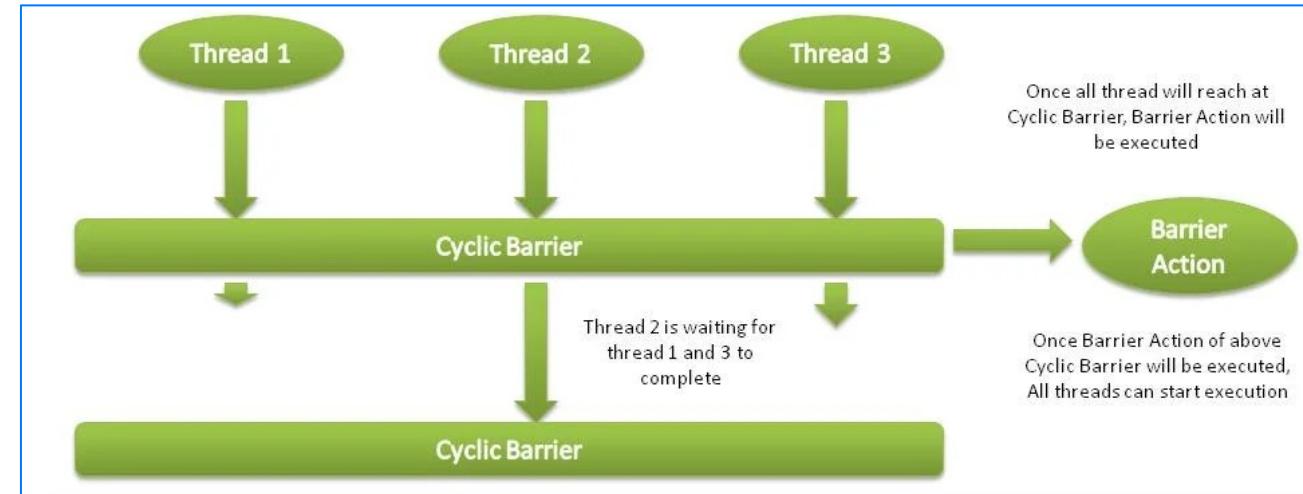


CyclicBarrier

Барьеры схожи с защёлками - блокируют группу потоков до достижения некоторого события. В случае барьеров этим событием является само достижение барьера определённым числом потоков одновременно. После этого барьер откроется, все потоки продолжат исполнение, и барьер снова закроется.

CyclicBarrier позволяет указать действие (с помощью `Runnable`), которое будет выполнено в одном из потоков, когда барьер откроется, но до разблокировки всех потоков.

Exchanger - двойной барьер, позволяющий сторонам обмениваться данными в точке встречи (barrier point).



ReadWriteLock

[ReadWriteLock](#) позволяет одновременно получить доступ множеству потоков-читателей и одному потоку-писателю.

ReentrantReadWriteLock:

- `readLock` - взять блокировку на чтение
- `writeLock` - взять блокировку на запись
- `xLock.lock` - заблокировать
- `xLock.unlock` - разблокировать

Предоставляет [справедливую](#) и [несправедливую](#) блокировки:

- Справедливую блокировку получает дольше всего ждущий поток.
- Порядок получения несправедливой блокировки не определён.

Если блокировка удерживается потоком-читателем и поток запрашивает блокировку для записи, никакие другие читатели не могут получить блокировку на чтение, пока поток-писатель не получит и не освободит блокировку.

Изменение статуса с писателя на читателя разрешено; обратное – нет.

```
Creates a new ReentrantReadWriteLock with default (nonfair) ordering properties.

public ReentrantReadWriteLock() {
    this(fair: false);
}

Creates a new ReentrantReadWriteLock with the given fairness policy.

Params: fair – true if this lock should use a fair ordering policy

public ReentrantReadWriteLock(boolean fair) {
    sync = fair ? new FairSync() : new NonfairSync();
    readerLock = new ReadLock(this);
    writerLock = new WriteLock(this);
}

public ReentrantReadWriteLock.WriteLock writeLock() { return writerLock; }
public ReentrantReadWriteLock.ReadLock readLock() { return readerLock; }
```

Синхронизация в корутинах

При использовании в корутинах средств синхронизации из `java.util.concurrent` возможны проблемы:

- одна корутина может исполняться несколькими потоками - может сломаться логика захвата и возврата блокировок
- при блокировке потоки диспетчеров не выполняют никакой работы

Поэтому нужно использовать специальные средства синхронизации для корутин - из пакета `kotlinx.coroutines.sync`:

`Mutex`
`Semaphore`

При блокировке с использованием этих механизмов корутина приостанавливается, но сам поток не блокируется - он может быть использован в других корутинах.

```
val mutex = Mutex()
```

```
→ mutex.lock()
/* critical section here */
mutex.unlock()

→ mutex.withLock {
    /* critical section here */
}
```

Синхронизация коллекций

• • • • •

Синхронизированные коллекции

Синхронизированные коллекции добавляют блокировки на все операции.

Примеры:

- Vector и Hashtable
- Collection.synchronizedXxx

Проблемы:

- низкая производительность
- всё равно возможна неатомарная модификация

Конкурентные коллекции

Конкурентные коллекции позволяют выполнять некоторые операции без блокировок, но с некоторыми компромиссами.

Пример: `CopyOnWriteArrayList` / `CopyOnWriteArraySet` - при каждой модификации создаётся новая копия.

Итераторы синхронизируются только в момент проверки видимости содержимого массива - остальное взаимодействие с коллекцией выполняется без синхронизации.

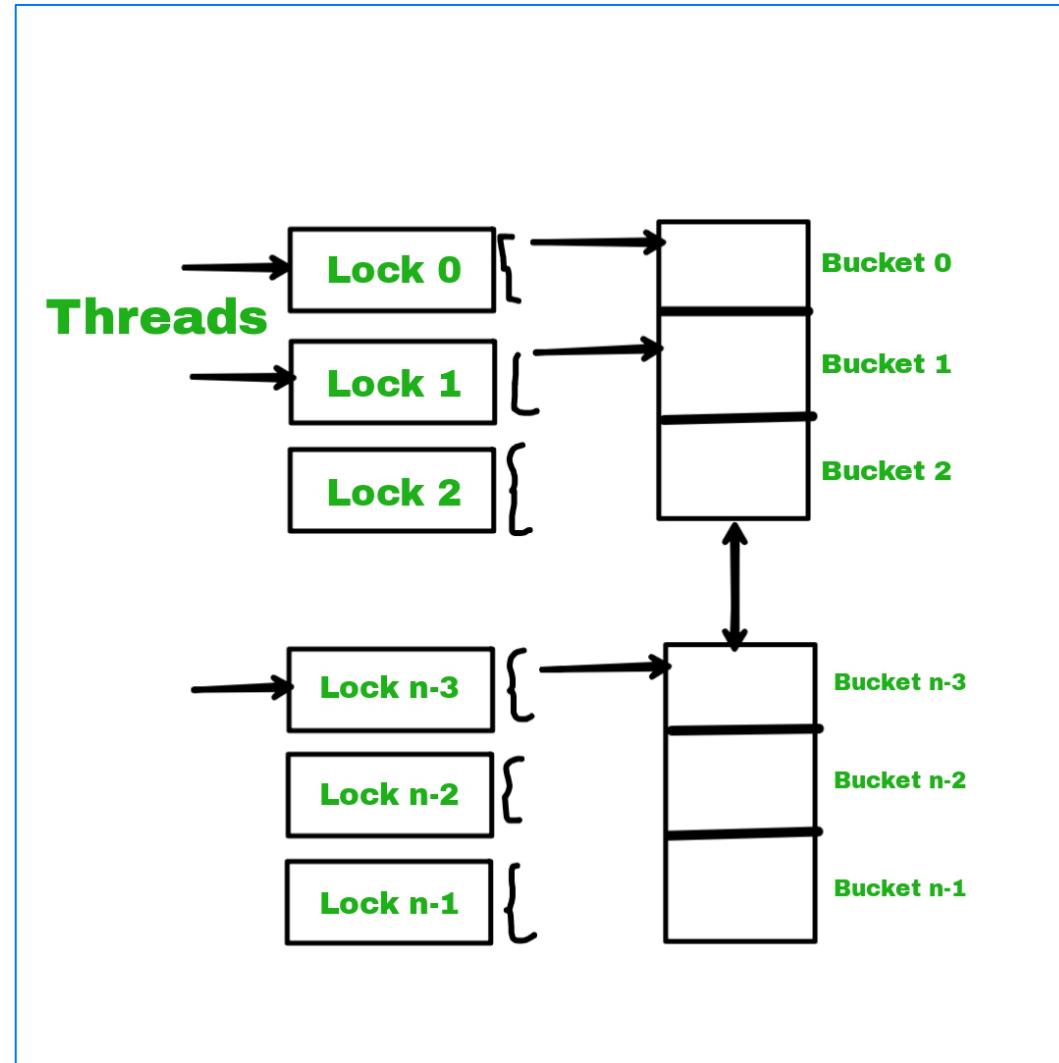
Итераторы возвращают элементы в таком виде, в каком они были при создании итератора, независимо от последующих модификаций. Итераторы не выбрасывают `ConcurrentModificationException`.

ConcurrentHashMap

ConcurrentHashMap вместо синхронизации каждого метода общей блокировкой механизм [членование блокировок \(lock striping\)](#).

- ограниченное число одновременных писателей
- неограниченное число конкурентных читателей - в том числе при наличии потоков-писателей
- итераторы не бросают `ConcurrentModificationException`, но коллекция будет иметь актуальный на момент создания итератора вид, при этом возможна (но не гарантируется) актуализация в соответствии с последующими модификациями.
- возвращаемый размер коллекции может уже быть неактуальным

→ Таким образом, требования к таким операциям были снижены ради оптимизации более важных операций: получения, записи и удаления элементов.



Очереди

BlockingQueue:

- LinkedBlockingQueue и ArrayBlockingQueue – FIFO-очереди с лучшей конкурентной производительностью, чем синхронизированные списки.
- PriorityBlockingQueue – упорядоченная по приоритету очередь
- Очередь с двусторонним доступом ([dequeue](#)) позволяет эффективно добавлять и удалять элементы с обеих сторон очереди. Реализации: ArrayDeque и LinkedBlockingDeque .

Такие очереди используются при реализации паттерна заимствования задач ([work stealing](#)). Данный паттерн предполагает наличие собственной очереди для каждого потребителя. Если потребитель выполнил все задачи своей очереди, он может забрать задачу из очереди другого потребителя. Это упрощает масштабирование, потребители не делят одну очередь, а большую часть времени работают только со своей. Задача другого исполнителя берётся с хвоста (сам исполнитель забирает задачи из головы), что также снижает конкуренцию.

Немного про корутины

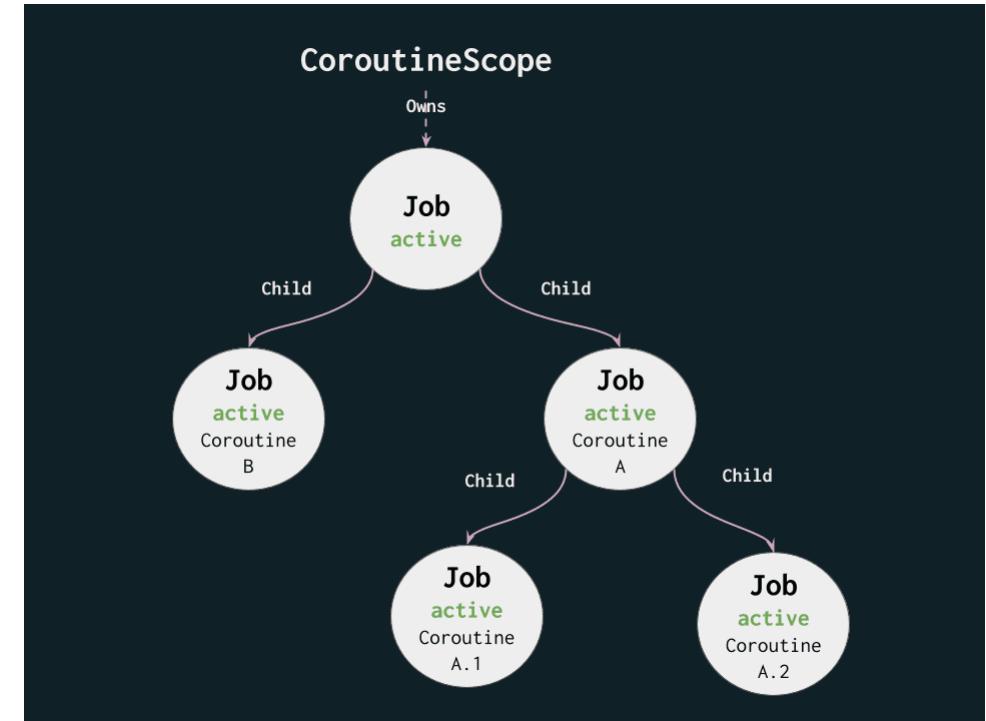


Structured concurrency

Корутины позволяют выстраивать **родительско-дочерние** отношения.

- можно запустить одну/несколько дочерних корутин
- можно прерывать дочерние корутины
- жизненный цикл дочерних корутин привязан к родительской корутине

Управление дочерней корутиной - через объект Job, возвращаемый launch() / async().



Parallel coroutine execution

```
↳ suspend fun computeInParallel(): Int = coroutineScope { this: CoroutineScope
    ↳     val one: Deferred<Int> = async { computeFirstAddendum() }
    ↳     val two: Deferred<Int> = async { computeSecondAddendum() }
    ↳     one.await() + two.await() ^coroutineScope
}

↳ suspend fun launchInParallel() = coroutineScope { this: CoroutineScope
    ↳     val one: Job = launch { computeFirstAddendum() }
    ↳     val two: Job = launch { computeSecondAddendum() }
    ↳     one.join()
    ↳     two.join()
}
```

Прерывание корутин

Процесс прерывания корутин - **кооперативный**: сам прерываемый код должен «участвовать» в прерывании.

Все функции из пакета `kotlinx.coroutines` проверяют, не прервали ли их. Если прервали, выбрасывают `CancellationException`. Поэтому, например, при вызове `cancel()` на корутине, которая находится в ожидании в `delay()`, корутина завершится.

Но если код в корутине не проверяет прерывание, вызов `cancel()` никак не повлияет на корутину.

```
val job: Job = launch { this: CoroutineScope
    longAction()
}
println("Action launched")
// ...
job.cancel()
```

```
suspend fun checkCancel() {
    coroutineScope { this: CoroutineScope
        while (isActive) {
            doNextPartOfTheJob()
        }
    }
}
```

Немного про Flow



Flow

Flow в общем случае генерирует поток значений, обработка которых выполняется в корутинах. Обработкой значений занимается наблюдатель(и).

Холодные потоки – потоки без состояния – создаются по требованию каждый раз, когда наблюдатель подписывается на поток. Каждый наблюдатель получает собственную последовательность значений.

Горячие потоки – генерируют значения независимо от наличия наблюдателей. Все наблюдатели получают одни и те же значения.

```
val flow = flow<String> { this: FlowCollector<String>
    ↗ emit( value: "1")
    ↗ delay( timeMillis: 1000L)
    ↗ emit( value: "2")
    ↗ delay( timeMillis: 1000L)
    ↗ emit( value: "3")
}
println("Flow created")
↗ delay( timeMillis: 1000L)
println("Subscribe to flow")

runBlocking { this: CoroutineScope
    ↗ flow.collect { it: String
        println("Value received: $it")
    }
}
```

StateFlow и SharedFlow

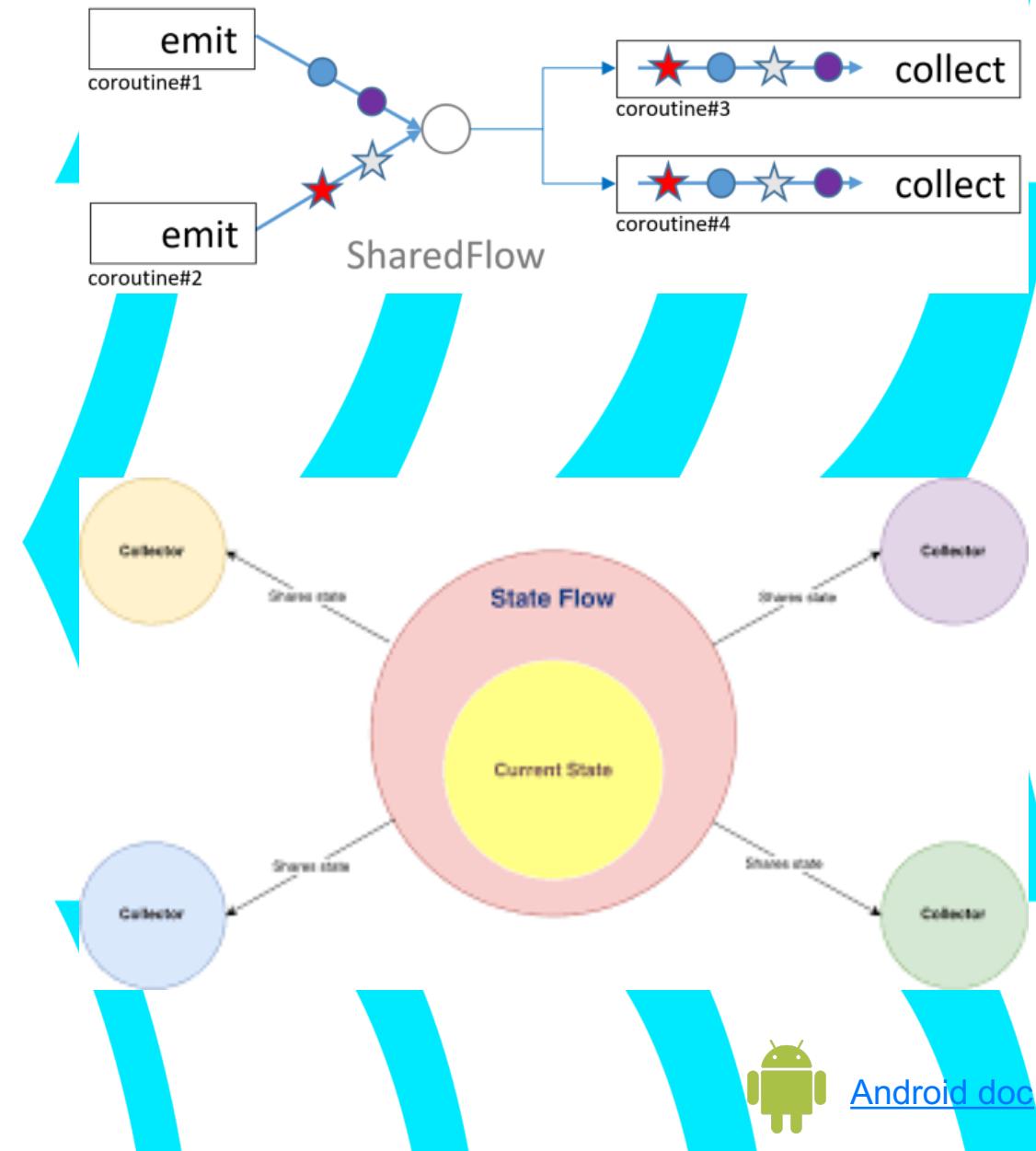
StateFlow

- горячий поток
- всегда имеет состояние (изначально – дефолтное)
- при подписке слушатель получает текущее состояние
- при изменении состояния все слушатели получают один и тот же его экземпляр

SharedFlow

- горячий поток
- может хранить и отдавать слушателям n последних значений (параметр `replay`)
- не требует дефолтного значения

SharedFlow – обобщение StateFlow, который можно воспринимать как SharedFlow с `replay=1`.

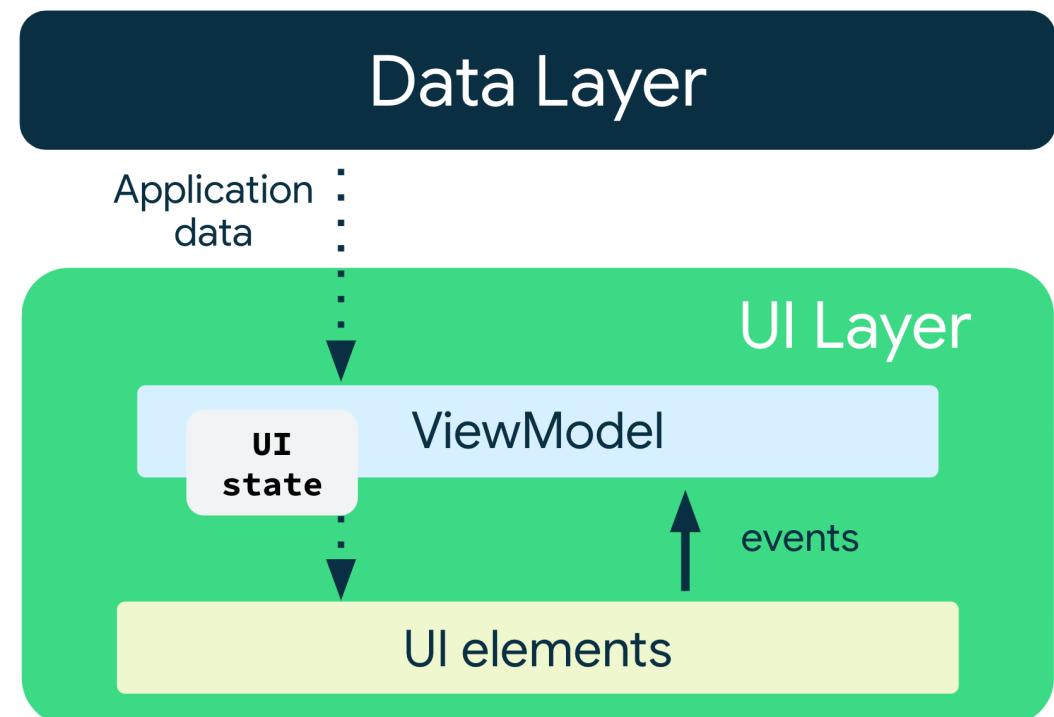


Android doc

Flow стал стандартом разработки Android

StateFlow и SharedFlow позволяют безопасно (в контексте многопоточности) менять значение, а также получать его из других потоков через механизм подписки.

- + При подписке на изменения значения задаётся контекст корутины, который может быть связан с интересующим жизненным циклом (например Activity).
- + Удобные функции для преобразования, комбинирования и т.д.
- + Позволяет сохранить [Unidirectional Data Flow \(UDF\)](#) в Android-приложениях



Подписка на Flow

Подписка на Flow - всегда происходит в контексте вызывающей корутины.

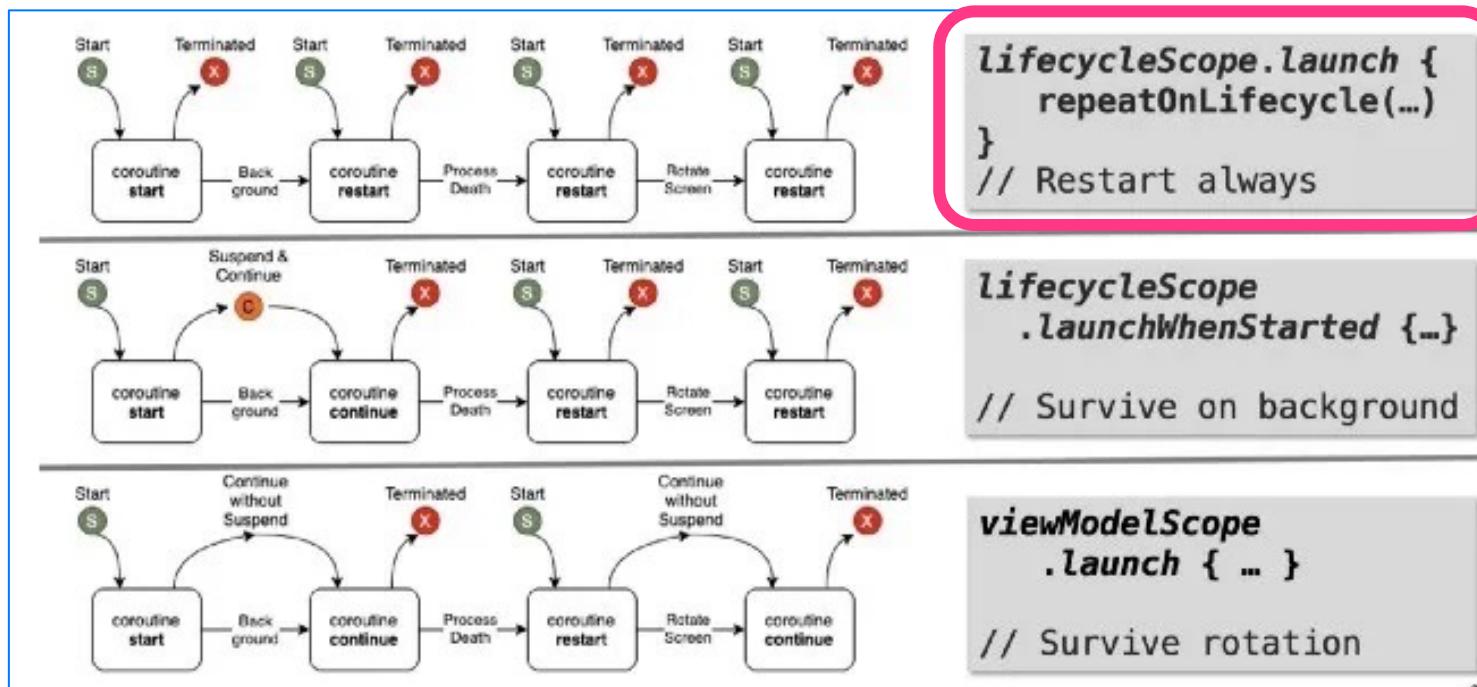
Методы подписки (`suspend`):

- `collect` - обрабатывает каждое значение
- `collectLatest` - при появлении нового значения обработка предыдущего прерывается



Подписка на Flow в Android

Из какого скоупа подписываться на изменения состояния?



Изменение Flow

- map
- filter
- transform - произвольное количество emit
- stateIn - преобразует холодный Flow в горячий

```
flowOf( ...elements: 1, 2, 3).transform { this: FlowCollector<Int>, it: Int
    emit(it)
    emit( value: it * 10)
}
```

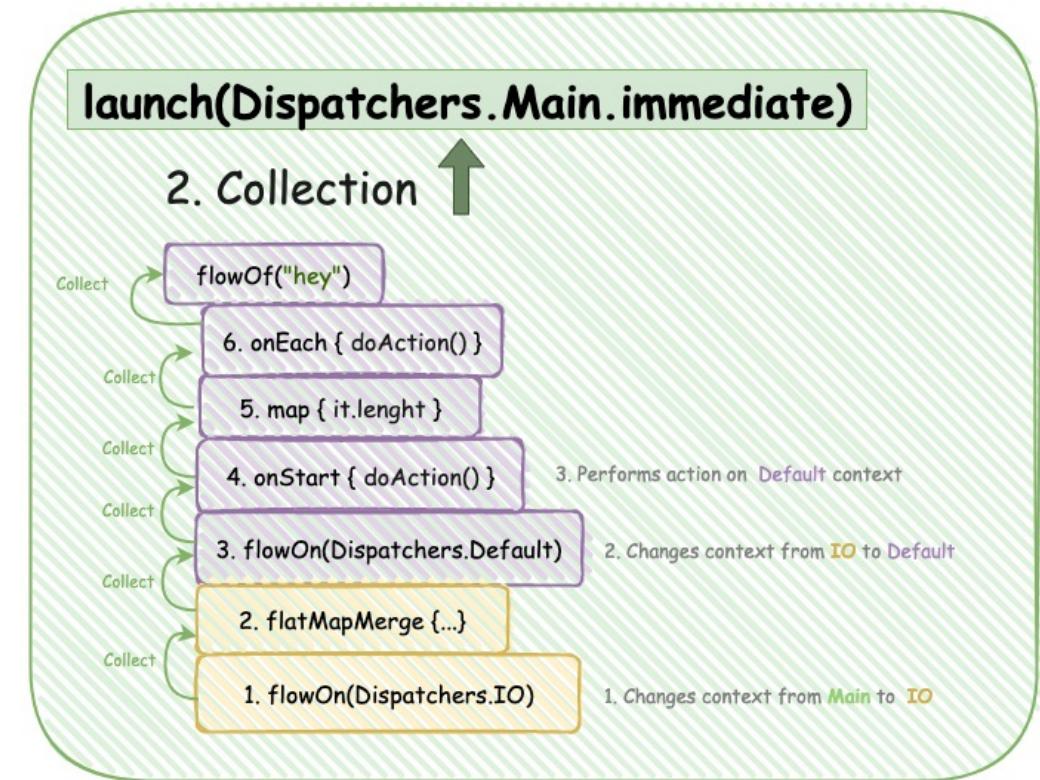
```
val backendState: Flow<Int> = flow { this: FlowCollector<Int>
    connectToBackend() // takes a lot of time
    while (true) {
        emit(receiveStateUpdateFromBackend())
    }
}
```

```
val hotFlow = StateInDemo()
    .backendState
    .stateIn( scope: this, SharingStarted.Eagerly, initialValue: -1)
/* ... */
// ViewModel 1
viewModelScope.launch { this: CoroutineScope
    hotFlow.collect( /* */)
}
// ViewModel 2
viewModelScope.launch { this: CoroutineScope
    hotFlow.collect( /* */)
}
// ViewModel 3
viewModelScope.launch { this: CoroutineScope
    hotFlow.collect( /* */)
} ^runBlocking
```

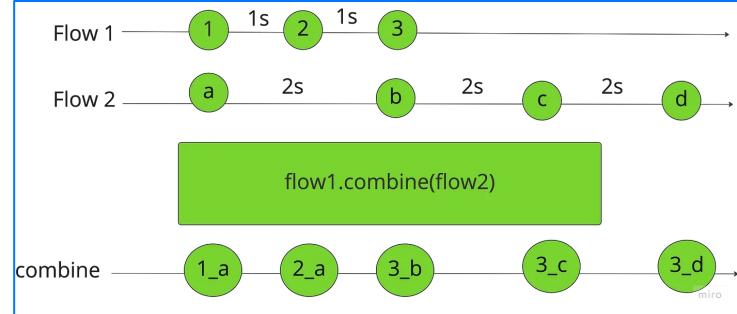
Переключение потоков

Промежуточный оператор `flowOn` - позволяет менять контекст исполнения `flow`.

Контекст меняется для `upstream`-операций - которые располагаются «выше» `flowOn`.



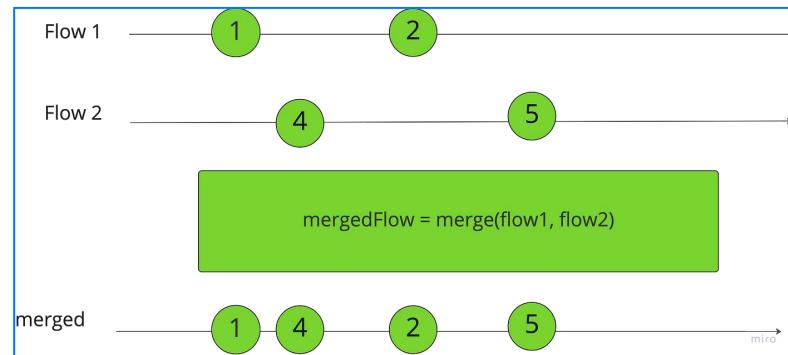
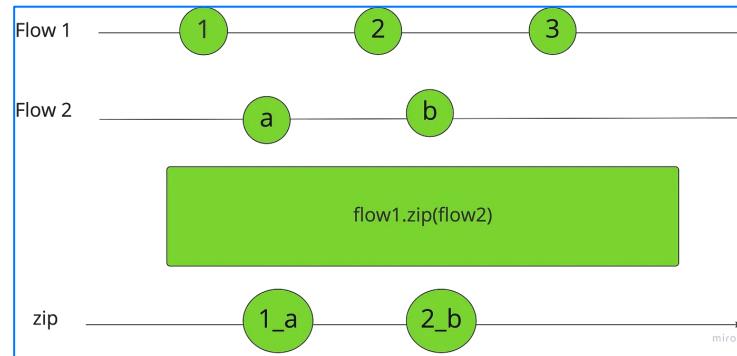
Работа с несколькими flow



`combine` - последние значения каждого из flow при появлении нового значения в любом из них

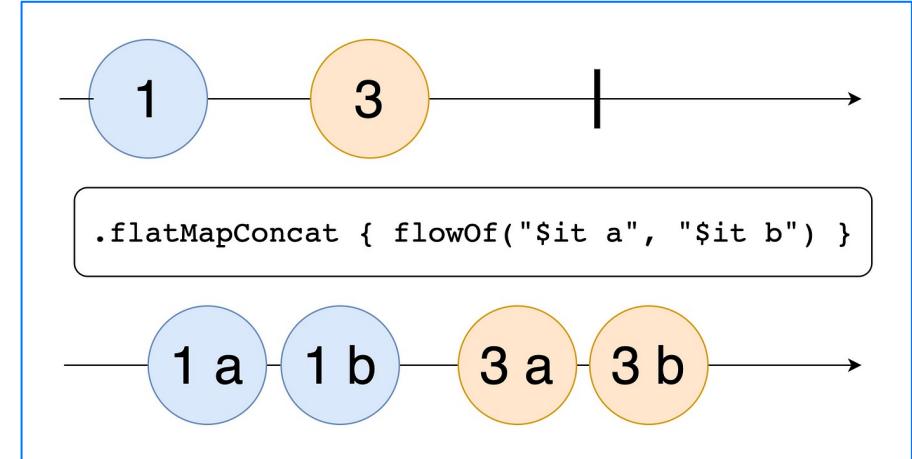
`zip` - ждёт «пары» новых значений из каждого flow

`merge` - последовательно отдаёт новые значения каждого из flow



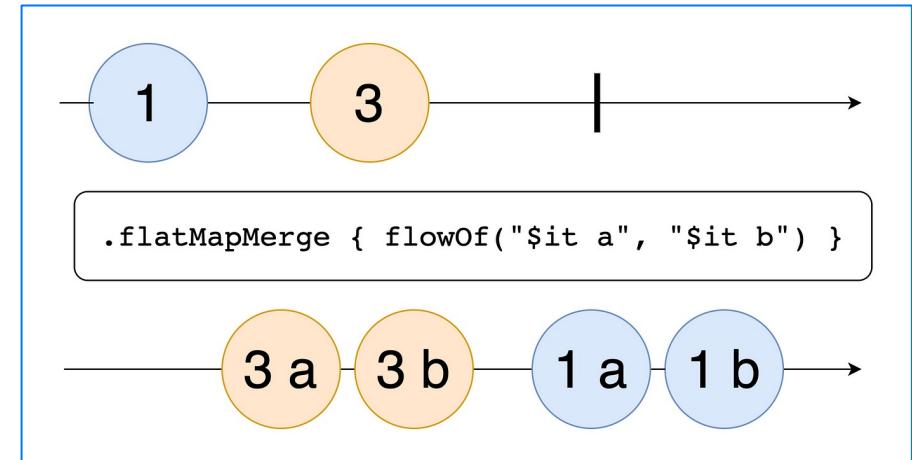
Работа с несколькими flow

flatMapConcat - преобразует каждое значение в новый flow. Подписчикам отправляет элементы, созданные новыми flow.



flatMapMerge - аналогичен flatMapConcat, но значения новых flow генерируются конкурентно - может меняться порядок элементов.

flatMapLatest - создаёт новый flow, но при появлении нового элемента прерывает обработку предыдущего. Подписчики всегда работают с последним элементом.



Обработка ошибок

Как можно:

- использовать try-catch при collect
- использовать оператор catch - захватит только исключения, выброшенные в upstream-операциях

```
fun main() = runBlocking<Unit> {
    try {
        simple().collect { value ->
            println(value)
            check(value <= 1) { "Collected $value" }
        }
    } catch (e: Throwable) {
        println("Caught $e")
    }
}
```

```
fun main() = runBlocking<Unit> {
    simple()
        .catch { e -> println("Caught $e") } // does not catch downstream
        .collect { value ->
            check(value <= 1) { "Collected $value" }
            println(value)
        }
}
```

Спасибо за
внимание!



Полезные ссылки

Java Memory Model:

[Алексей Шипилёв — Прагматика Java Memory Model](#)

[JAVA. Memory Model | Технострим](#)

Фундаментальная книга по многопоточности для платформы Java

- [Java Concurrency in Practice \(Brian Goetz\)](#)

BRIAN GOETZ
WITH TIM PEIERLS, JOSHUA BLOCH,
JOSEPH BOWBEER, DAVID HOLMES,
AND DOUG LEA

