

The Death of Unnecessary Thread Hopping

Inside Swift 6.2 Approachable Concurrency



Nikita Vasilev

iOS Software Engineer

Where Does My Async Code Actually Run?

- 1 **The Old Mental Model:** Why async caused unexpected execution changes
- 2 **Approachable Concurrency:** What changed in Swift 6.2 and Xcode 26
- 3 **Under the Hood:** `nonisolated` · `nonsending` · `@concurrent`
- 4 **Let's Trace It:** Following execution step-by-step across actors
- 5 **The New Mental Model:** Practical implications for production code

The Old Mental Model

```
@MainActor
final class ViewModel {

    nonisolated
    func loadData() async {
        // Where am I actually running?
    }
}
```

❓ The Common Misconception

Intuition built up over time: Entering a nonisolated async function moves execution away from the caller thread.

💡 The Key Distinction

async ≠ background. async indicates potential suspension, not an instruction to hop to a background thread.

What Changed in Swift 6.2?

BEFORE SWIFT 6.2

Global Hop Default

Caller → await → Global Concurrent Executor

Execution left the caller's context automatically, creating unnecessary thread hops even when concurrency wasn't requested.

SWIFT 6.2 (SE-0461)

Preserve Context

Caller → await → Same Execution Context

Preserves caller isolation by default
(NonisolatedNonsendingByDefault). You remain where you are unless asked otherwise.

Three Core Concepts



nonisolated

"I don't belong to an actor"

Specifies that a function is not bound to a specific actor isolation domain.



nonsending

"I inherit caller isolation."

Execution stays associated with the caller context instead of forcing transfer.



@concurrent

"Run me concurrently."

Explicit attribute opting into background global executor execution.

Let's Run an Experiment

```
@MainActor
final class ViewModel {

    func updateData() {
        print("1:", Thread.isMain)
    }

    nonisolated
    func helperMethod() async {
        print("2:", Thread.isMain)
    }

    @concurrent
    func thirdMethod() async {
        print("3:", Thread.isMain)
    }
}
```

Called from @MainActor

We call all three methods sequentially from a main-actor isolated context.

Question: What thread will each print statement output under legacy vs. Swift 6.2 flags?

Experiment Results

Function Call	Approachable Concurrency OFF	Approachable Concurrency ON
updateData()	Main Thread Main Thread	Main Thread Main Thread
helperMethod()	Background Thread Background Thread	Main Thread Main Thread
thirdMethod()	Background Thread Background Thread	Background Thread Background Thread

Key Takeaway: async didn't change its fundamental semantics. The *default execution context preservation* changed.

await Does Not Mean Background

"await means: This operation may suspend.

It does not mean: Move this to a background thread."

|| **Suspension Point:** The function may yield control while waiting for I/O, tasks, or actors.

↺ **Orthogonal Choice:** Where execution resumes is governed by isolation, not suspension.

Explicit Concurrency with @concurrent

```
@concurrent
func processLargePayload() async → Result {
    // Expensive CPU computation
    // Parsing massive JSON
    // Image filtering / processing
}
```

🎯 Communicating Intent

Use @concurrent when offloading heavy work from caller threads is intentional:

- ✓ CPU-heavy calculations
- ✓ Isolating work from UI / MainActor
- ✓ Parallel background tasks

Tracing Across Actor Boundaries

```
@globalActor
actor BackgroundActor {
    static let shared = BackgroundActor()
}

@MainActor
final class ViewModel {
    func runTest() {
        Task {
            await complexHelper()
        }
    }

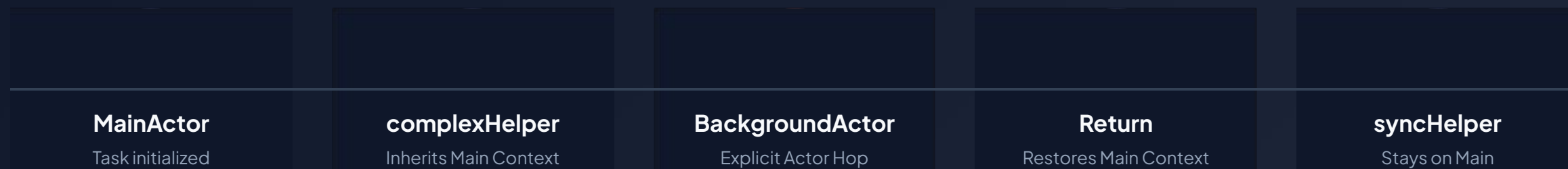
    nonisolated func complexHelper() async {
        print("2:", Thread.isMain)

        await BackgroundActor.shared.doWork {
            print("3:", Thread.isMain)
        }

        print("4:", Thread.isMain)
        syncHelper()
    }

    nonisolated func syncHelper() {
        print("5:", Thread.isMain)
    }
}
```

Execution Chain Flow



Actor boundaries are explicit. Unnecessary executor hops disappear.

Before vs. Swift 6.2 Comparison

✗ Before (Implicit Hops)

```
Main Actor
↓ (hop)
Global Executor
↓ (hop)
Background Actor
↓ (hop)
Global Executor
↓ (hop)
Main Actor
```

✓ Swift 6.2 (Explicit Hops)

```
Main Actor
|
|—— Background Actor (Explicit Boundary)
|
|—— Return to Main Context
```

Why This Matters in Production

- 🛡️ **Easier Reasoning:** Fewer implicit thread switches.
- 🐛 **Predictable Debugging:** Stack traces reflect actual intent.
- 📄 **Safer Refactoring:** Execution context stays local by default.

⚠️ Migration Warning

When enabling `NonisolatedNonsendingByDefault` in Xcode 26, heavy work in nonisolated async functions that previously ran on background threads will now stay on `@MainActor`. Audit CPU-heavy methods and add `@concurrent!`

The New Mental Model Checklist

- ✓ **async means suspension:** Not background execution.
- ✓ **nonisolated means no actor isolation:** Not "run somewhere else".
- ✓ **nonsending preserves caller context:** Don't hop unless required.
- ✓ **@concurrent requests concurrency:** Explicit, self-documenting intent.
- ✓ **Actors define boundaries:** Cross them intentionally with await.

THE GOLDEN RULE

**Stay where you are.
Unless there is a reason to move.**

Need Isolation?

Actor

Need
Concurrency?

@concurrent

Need
Suspension?

async

WRAP UP

Swift 6.2 doesn't remove concurrency.
It removes unnecessary concurrency.