Structure

TaskGroup

A group that contains dynamically created child tasks.

iOS 13.0+ | iPadOS 13.0+ | Mac Catalyst 13.0+ | macOS 10.15+ | tvOS 13.0+ | visionOS 1.0+ | watchOS 6.0+

```
@frozen
struct TaskGroup<ChildTaskResult> where ChildTaskResult : Sendable
```

Overview

To create a task group, call the `withTaskGroup(of:returning:body:)` method.

Don’t use a task group from outside the task where you created it. In most cases, the Swift type system prevents a task group from escaping like that because adding a child task to a task group is a mutating operation, and mutation operations can’t be performed from a concurrent execution context like a child task.

Structured Concurrency

Structured concurrency is a way to organize your program, and tasks, in such a way that tasks don’t outlive the scope in which they are created. Within a structured task hierarchy, no child task remains running longer than its parent task. This guarantee simplifies reasoning about resource usage, and is a powerful mechanism that you can use to write well-behaved concurrent programs.

A task group is the primary way to create structured concurrency tasks in Swift. Another way of creating structured tasks is an `async let` declaration.

Structured concurrency tasks are often called “child tasks” because of their relationship with their parent task. A child task inherits the parent’s priority, task-local values, and is structured in the sense that its lifetime never exceeds the lifetime of the parent task.

A task group *always* waits for all child tasks to complete before it's destroyed. Specifically, `with...TaskGroup` APIs don't return until all the child tasks created in the group's scope have completed running.

Structured concurrency APIs (including task groups and `async let`), *always* waits for the completion of tasks contained within their scope before returning. Specifically, this means that even if you await a single task result and return it from a `withTaskGroup` function body, the group automatically waits for all the remaining tasks before returning:

```
func takeFirst(actions: [@Sendable () -> Int]) async -> Int? {
    await withTaskGroup { group in
        for action in actions {
            group.addTask { action() }
        }

        return await group.next() // return the first action to complete
    } // the group will ALWAYS await the completion of all the actions (!)
}
```

In the above example, even though the code returns the first collected integer from all actions added to the task group, the task group *always*, automatically, waits for the completion of all the resulting tasks.

You can use `group.cancelAll()` to signal cancellation to the remaining in-progress tasks, however this doesn't interrupt their execution automatically. Rather, the child tasks need to cooperatively react to the cancellation, and return early if that's possible.

To create unstructured concurrency tasks, you can use `Task.init`, `Task.detached` or `Task.immediate`.

Task Group Cancellation

You can cancel a task group and all of its child tasks by calling the `cancelAll()` method on the task group, or by canceling the task in which the group is running.

If you call `addTask(name:priority:operation:)` to create a new task in a canceled group, that task is immediately canceled after creation. Alternatively, you can call `addTaskUnlessCancelled(name:priority:operation:)`, which doesn't create the task if the group has already been canceled. Choosing between these two functions lets you control how to react to cancellation within a group: some child tasks need to run regardless of cancellation, but other tasks are better not even being created when you know they can't produce useful results.

In nonthrowing task groups the tasks you add to a group with this method are nonthrowing, those tasks can't respond to cancellation by throwing `CancellationError`. The tasks must handle cancellation in some other way, such as returning the work completed so far, returning an empty result, or returning `nil`. For tasks that need to handle cancellation by throwing an error, use the `withThrowingTaskGroup(of:returning:body:)` method instead.

Task execution order

Tasks added to a task group execute concurrently, and may be scheduled in any order.

Cancellation behavior

A task group becomes canceled in one of the following ways:

- When `cancelAll()` is invoked on it.
- When the `Task` running this task group is canceled.

Because a `TaskGroup` is a structured concurrency primitive, cancellation is automatically propagated through all of its child-tasks (and their child tasks).

A canceled task group can still keep adding tasks, however they will start being immediately canceled, and might respond accordingly. To avoid adding new tasks to an already canceled task group, use `addTaskUnlessCancelled(name:priority:body:)` rather than the plain `addTask(name:priority:body:)` which adds tasks unconditionally.

For information about the language-level concurrency model that `TaskGroup` is part of, see Concurrency in The Swift Programming Language.

See Also

`ThrowingTaskGroup`

See Also

`DiscardingTaskGroup`

See Also

ThrowingDiscardingTaskGroup

Topics

Adding Tasks to a Task Group

```
func addTask(priority: TaskPriority?, operation: sending () async -> ChildTaskResult)
```

Adds a child task to the group.

```
func addTask(name: String?, priority: TaskPriority?, operation: sending () async -> ChildTaskResult)
```

Adds a child task to the group.

```
func addTask(executorPreference: (any TaskExecutor)?, priority: TaskPriority?, operation: sending () async -> ChildTaskResult)
```

Adds a child task to the group.

```
func addTask(name: String?, executorPreference: (any TaskExecutor)?, priority: TaskPriority?, operation: sending () async -> ChildTaskResult)
```

Adds a child task to the group.

```
func addTaskUnlessCancelled(name: String?, executorPreference: (any TaskExecutor)?, priority: TaskPriority?, operation: sending () async -> ChildTaskResult) -> Bool
```

Adds a child task to the group, unless the group has been canceled. Returns a boolean value indicating if the task was successfully added to the group or not.

```
func addTaskUnlessCancelled(executorPreference: (any TaskExecutor)?, priority: TaskPriority?, operation: sending () async -> ChildTaskResult) -> Bool
```

Adds a child task to the group, unless the group has been canceled. Returns a boolean value indicating if the task was successfully added to the group or not.

```
func addTaskUnlessCancelled(name: String?, priority: TaskPriority?, operation: sending () async -> ChildTaskResult) -> Bool
```

Adds a child task to the group, unless the group has been canceled. Returns a boolean value indicating if the task was successfully added to the group or not.

`func addTaskUnlessCancelled(priority: TaskPriority?, operation: sending () async -> ChildTaskResult) -> Bool`

Adds a child task to the group, unless the group has been canceled. Returns a boolean value indicating if the task was successfully added to the group or not.

`func addImmediateTask(name: String?, priority: TaskPriority?, executorPreference: consuming (any TaskExecutor)?, operation: sending () async -> ChildTaskResult)`

Add a child task to the group and immediately start running it in the context of the calling thread/task.

`func addImmediateTaskUnlessCancelled(name: String?, priority: TaskPriority?, executorPreference: consuming (any TaskExecutor)?, operation: sending () async -> ChildTaskResult) -> Bool`

Add a child task to the group and immediately start running it in the context of the calling thread/task.

Accessing Individual Results

`func next() async -> ChildTaskResult?`

`func next(isolation: isolated (any Actor)?) async -> ChildTaskResult?`

Waits for the next child task to complete, and returns the value it returned.

`var isEmpty: Bool`

A Boolean value that indicates whether the group has any remaining tasks.

`func waitForAll(isolation: isolated (any Actor)?) async`

Wait for all of the group's remaining tasks to complete.

Accessing an Asynchronous Sequence of Results

`func makeAsyncIterator() -> TaskGroup<ChildTaskResult>.Iterator`

Creates the asynchronous iterator that produces elements of this asynchronous sequence.

`func allSatisfy((Self.Element) async throws -> Bool) async rethrows -> Bool`

Returns a Boolean value that indicates whether all elements produced by the asynchronous sequence satisfy the given predicate.

`func compactMap<ElementOfResult>((Self.Element) async throws -> ElementOfResult?) -> AsyncThrowingCompactMapSequence<Self, ElementOfResult>`

Creates an asynchronous sequence that maps an error-throwing closure over the base sequence’s elements, omitting results that don’t return a value.

`func compactMap<ElementOfResult>((Self.Element) async -> ElementOfResult?) -> AsyncCompactMapSequence<Self, ElementOfResult>`

Creates an asynchronous sequence that maps the given closure over the asynchronous sequence’s elements, omitting results that don’t return a value.

`func contains(Self.Element) async rethrows -> Bool`

Returns a Boolean value that indicates whether the asynchronous sequence contains the given element.

`func contains(where: (Self.Element) async throws -> Bool) async rethrows -> Bool`

Returns a Boolean value that indicates whether the asynchronous sequence contains an element that satisfies the given predicate.

`func drop(while: (Self.Element) async -> Bool) -> AsyncDropWhileSequence<Self>`

Omits elements from the base asynchronous sequence until a given closure returns false, after which it passes through all remaining elements.

`func dropFirst(Int) -> AsyncDropFirstSequence<Self>`

Omits a specified number of elements from the base asynchronous sequence, then passes through all remaining elements.

`func filter((Self.Element) async -> Bool) -> AsyncFilterSequence<Self>`

Creates an asynchronous sequence that contains, in order, the elements of the base sequence that satisfy the given predicate.

`func first(where: (Self.Element) async throws -> Bool) async rethrows -> Self.Element?`

Returns the first element of the sequence that satisfies the given predicate.

`func flatMap<SegmentOfResult>((Self.Element) async throws -> SegmentOfResult) -> AsyncThrowingFlatMapSequence<Self, SegmentOfResult>`

Creates an asynchronous sequence that concatenates the results of calling the given error-throwing transformation with each element of this sequence.

`func map<Transformed>((Self.Element) async throws -> Transformed) -> AsyncThrowingMapSequence<Self, Transformed>`

Creates an asynchronous sequence that maps the given error-throwing closure over the asynchronous sequence's elements.

`func map<Transformed>((Self.Element) async -> Transformed) -> AsyncMapSequence<Self, Transformed>`

Creates an asynchronous sequence that maps the given closure over the asynchronous sequence's elements.

`func max() async rethrows -> Self.Element?`

Returns the maximum element in an asynchronous sequence of comparable elements.

`func max(by: (Self.Element, Self.Element) async throws -> Bool) async rethrows -> Self.Element?`

Returns the maximum element in the asynchronous sequence, using the given predicate as the comparison between elements.

`func min() async rethrows -> Self.Element?`

Returns the minimum element in an asynchronous sequence of comparable elements.

`func min(by: (Self.Element, Self.Element) async throws -> Bool) async rethrows -> Self.Element?`

Returns the minimum element in the asynchronous sequence, using the given predicate as the comparison between elements.

`func prefix(Int) -> AsyncPrefixSequence<Self>`

Returns an asynchronous sequence, up to the specified maximum length, containing the initial elements of the base asynchronous sequence.

`func prefix(while: (Self.Element) async -> Bool) rethrows -> AsyncPrefixWhileSequence<Self>`

Returns an asynchronous sequence, containing the initial, consecutive elements of the base sequence that satisfy the given predicate.

`func reduce<Result>(Result, (Result, Self.Element) async throws -> Result) async rethrows -> Result`

Returns the result of combining the elements of the asynchronous sequence using the given closure.

`func reduce<Result>(into: Result, (inout Result, Self.Element) async throws -> Void) async rethrows -> Result`

Returns the result of combining the elements of the asynchronous sequence using the given closure, given a mutable initial value.

Canceling Tasks

`var isCancelled: Bool`

A Boolean value that indicates whether the group was canceled.

`func cancelAll()`

Cancel all of the remaining tasks in the group.

Supporting Types

`typealias Element`

The type of element produced by this asynchronous sequence.

`struct Iterator`

A type that provides an iteration interface over the results of tasks added to the group.

`typealias AsyncIterator`

The type of asynchronous iterator that produces elements of this asynchronous sequence.

Deprecated

~~`func add(priority: TaskPriority?, operation: () async -> ChildTaskResult) async -> Bool`~~

Deprecated

~~`func async(priority: TaskPriority?, operation: () async -> ChildTaskResult)`~~

Deprecated

~~`func asyncUnlessCancelled(priority: TaskPriority?, operation: () async -> ChildTaskResult) -> Bool`~~

Deprecated

~~`func spawn(priority: TaskPriority?, operation: () async -> ChildTaskResult)`~~

Deprecated

~~`func spawnUnlessCancelled(priority: TaskPriority?, operation: () async -> ChildTaskResult) -> Bool`~~

Deprecated

Default Implementations

AsyncSequence Implementations

Relationships

Conforms To

`AsyncSequence`

Conforms when `ChildTaskResult` conforms to `Copyable`, `Escapable`, and `Sendable`.

`BitwiseCopyable`

Conforms when `ChildTaskResult` conforms to `Copyable`, `Escapable`, and `Sendable`.

`Copyable`

Conforms when `ChildTaskResult` conforms to `Copyable`, `Escapable`, and `Sendable`.

See Also

Tasks

```
struct Task
```

A unit of asynchronous work.

```
func withTaskGroup<ChildTaskResult, GroupResult>(of: ChildTaskResult
.Type, returning: GroupResult.Type, isolation: isolated (any Actor)?,
body: (inout TaskGroup<ChildTaskResult>) async -> GroupResult) async ->
GroupResult
```

Starts a new scope that can contain a dynamic number of child tasks.

```
struct ThrowingTaskGroup
```

A group that contains throwing, dynamically created child tasks.

```
func withThrowingTaskGroup<ChildTaskResult, GroupResult>(of: ChildTask
Result.Type, returning: GroupResult.Type, isolation: isolated (any
Actor)?, body: (inout ThrowingTaskGroup<ChildTaskResult, any Error>)
async throws -> GroupResult) async rethrows -> GroupResult
```

Starts a new scope that can contain a dynamic number of throwing child tasks.

```
struct TaskPriority
```

The priority of a task.

```
struct DiscardingTaskGroup
```

A discarding group that contains dynamically created child tasks.

```
func withDiscardingTaskGroup<GroupResult>(returning: GroupResult.Type,
isolation: isolated (any Actor)?, body: (inout DiscardingTaskGroup)
async -> GroupResult) async -> GroupResult
```

Starts a new scope that can contain a dynamic number of child tasks.

```
struct ThrowingDiscardingTaskGroup
```

A throwing discarding group that contains dynamically created child tasks.

```
func withThrowingDiscardingTaskGroup<GroupResult>(returning: Group
Result.Type, isolation: isolated (any Actor)?, body: (inout Throwing
DiscardingTaskGroup<any Error>) async throws -> GroupResult) async
throws -> GroupResult
```

Starts a new scope that can contain a dynamic number of child tasks.

```
struct UnsafeCurrentTask
```

An unsafe reference to the current task.