Structure

Task

A unit of asynchronous work.

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

```
@frozen
struct Task<Success, Failure> where Success : Sendable, Failure : Error
```

Overview

When you create an instance of `Task`, you provide a closure that contains the work for that task to perform. Tasks can start running immediately after creation; you don't explicitly start or schedule them. After creating a task, you use the instance to interact with it — for example, to wait for it to complete or to cancel it. It's not a programming error to discard a reference to a task without waiting for that task to finish or canceling it. A task runs regardless of whether you keep a reference to it. However, if you discard the reference to a task, you give up the ability to wait for that task's result or cancel the task.

To support operations on the current task, which can be either a detached task or child task, `Task` also exposes class methods like `yield()`. Because these methods are asynchronous, they're always invoked as part of an existing task.

Only code that's running as part of the task can interact with that task. To interact with the current task, you call one of the static methods on `Task`.

A task's execution can be seen as a series of periods where the task ran. Each such period ends at a suspension point or the completion of the task. These periods of execution are represented by instances of `PartialAsyncTask`. Unless you're implementing a custom executor, you don't directly interact with partial tasks.

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

Task Cancellation

Tasks include a shared mechanism for indicating cancellation, but not a shared implementation for how to handle cancellation. Depending on the work you're doing in the task, the correct way to stop that work varies. Likewise, it's the responsibility of the code running as part of the task to check for cancellation whenever stopping is appropriate. In a long-task that includes multiple pieces, you might need to check for cancellation at several points, and handle cancellation differently at each point. If you only need to throw an error to stop the work, call the `Task.checkCancellation()` function to check for cancellation. Other responses to cancellation include returning the work completed so far, returning an empty result, or returning `nil`.

Cancellation is a purely Boolean state; there's no way to include additional information like the reason for cancellation. This reflects the fact that a task can be canceled for many reasons, and additional reasons can accrue during the cancellation process.

Task closure lifetime

Tasks are initialized by passing a closure containing the code that will be executed by a given task.

After this code has run to completion, the task has completed, resulting in either a failure or result value, this closure is eagerly released.

Retaining a task object doesn't indefinitely retain the closure, because any references that a task holds are released after the task completes. Consequently, tasks rarely need to capture weak references to values.

For example, in the following snippet of code it is not necessary to capture the actor as `weak`, because as the task completes it'll let go of the actor reference, breaking the reference cycle between the Task and the actor holding it.

```
struct Work: Sendable {}

actor Worker {
    var work: Task<Void, Never>?
    var result: Work?

    deinit {
        // even though the task is still retained,
        // once it completes it no longer causes a reference cycle with the actor

        print("deinit actor")
    }
```

```
    func start() {
        work = Task {
            print("start task work")
            try? await Task.sleep(for: .seconds(3))
            self.result = Work() // we captured self
            print("completed task work")
            // but as the task completes, this reference is released
        }
        // we keep a strong reference to the task
    }
}
```

And using it like this:

```
await Worker().start()
```

Note that the actor is only retained by the start() method's use of `self`, and that the start method immediately returns, without waiting for the unstructured `Task` to finish. Once the task is completed and its closure is destroyed, the strong reference to the actor is also released allowing the actor to deinitialize as expected.

Therefore, the above call will consistently result in the following output:

```
start task work
completed task work
deinit actor
```

Topics

Creating a Task

`init(name: String?, priority: TaskPriority?, operation: sending () async -> Success)`

Runs the given nonthrowing operation asynchronously as part of a new *unstructured* top-level task.

`init(name: String?, priority: TaskPriority?, operation: sending () async throws -> Success)`

Runs the given throwing operation asynchronously as part of a new *unstructured* top-level task.

`init(name: String?, executorPreference: (any TaskExecutor)?, priority: TaskPriority?, operation: sending () async throws -> Success)`

Runs the given throwing operation asynchronously as part of a new *unstructured* top-level task.

`init(name: String?, executorPreference: (any TaskExecutor)?, priority: TaskPriority?, operation: sending () async -> Success)`

Runs the given nonthrowing operation asynchronously as part of a new *unstructured* top-level task.

`static var currentPriority: TaskPriority`

The current task's priority.

`static var basePriority: TaskPriority?`

The current task's base priority.

`func withTaskPriorityEscalationHandler<T, E>(operation: () async throws (E) -> T, onPriorityEscalated: (TaskPriority, TaskPriority) -> Void, isolation: isolated (any Actor)?) async throws(E) -> T`

Runs the passed `operation` while registering a task priority escalation handler. The handler will be triggered concurrently to the current task if the current is subject to priority escalation.

Creating a Detached Task

`static func detached(name: String?, priority: TaskPriority?, operation: sending () async throws -> Success) -> Task<Success, any Error>`

Runs the given throwing operation asynchronously as part of a new *unstructured detached* top-level task.

`static func detached(name: String?, priority: TaskPriority?, operation: sending () async -> Success) -> Task<Success, Never>`

Runs the given nonthrowing operation asynchronously as part of a new *unstructured detached* top-level task.

`static func detached(name: String?, executorPreference: (any TaskExecutor)?, priority: TaskPriority?, operation: sending () async throws -> Success) -> Task<Success, any Error>`

Runs the given throwing operation asynchronously as part of a new *unstructured detached* top-level task.

```
static func detached(name: String?, executorPreference: (any Task
Executor)?, priority: TaskPriority?, operation: sending () async ->
Success) -> Task<Success, Never>
```

Runs the given nonthrowing operation asynchronously as part of a new *unstructured detached* top-level task.

Creating a Task that Starts Immediately

```
static func immediate(name: String?, priority: TaskPriority?, executor
Preference: consuming (any TaskExecutor)?, operation: sending () async
-> Success) -> Task<Success, Never>
```

Create and immediately start running a new detached task in the context of the calling thread/task.

```
static func immediate(name: String?, priority: TaskPriority?, executor
Preference: consuming (any TaskExecutor)?, operation: sending () async
throws -> Success) -> Task<Success, any Error>
```

Create and immediately start running a new detached task in the context of the calling thread/task.

```
static func immediateDetached(name: String?, priority: TaskPriority?,
executorPreference: consuming (any TaskExecutor)?, operation: sending
() async throws -> Success) -> Task<Success, any Error>
```

Create and immediately start running a new task in the context of the calling thread/task.

```
static func immediateDetached(name: String?, priority: TaskPriority?,
executorPreference: consuming (any TaskExecutor)?, operation: sending
() async -> Success) -> Task<Success, Never>
```

Create and immediately start running a new task in the context of the calling thread/task.

Accessing Results

```
var value: Success
```

The result from a throwing task, after it completes.

```
var value: Success
```

The result from a nonthrowing task, after it completes.

```
var result: Result<Success, Failure>
```

The result or error from a throwing task, after it completes.

Accessing the Current Task's Name

`static var name: String?`

Returns the human-readable name of the current task, if it was set during the tasks' creation.

Canceling Tasks

`struct CancellationError`

An error that indicates a task was canceled.

`func cancel()`

Cancels this task.

`var isCancelled: Bool`

A Boolean value that indicates whether the task should stop executing.

`static var isCancelled: Bool`

A Boolean value that indicates whether the task should stop executing.

`static func checkCancellation() throws`

Throws an error if the task was canceled.

~~`func withTaskCancellationHandler<T>(handler: () -> Void, operation: () async throws -> T) async rethrows -> T`~~

Deprecated

`func withTaskCancellationHandler<T>(operation: () async throws -> T, onCancel: () -> Void, isolation: isolated (any Actor)?) async rethrows -> T`

Execute an operation with a cancellation handler that's immediately invoked if the current task is canceled.

Suspending Execution

`static func yield() async`

Suspends the current task and allows other tasks to execute.

`static func sleep(nanoseconds: UInt64) async throws`

Suspends the current task for at least the given duration in nanoseconds.

```
static func sleep<C>(for: C.Instant.Duration, tolerance: C.Instant.Duration?, clock: C) async throws
```

Suspends the current task for the given duration.

```
static func sleep<C>(until: C.Instant, tolerance: C.Instant.Duration?, clock: C) async throws
```

Suspends the current task until the given deadline within a tolerance.

Escalating Tasks

```
func escalatePriority(to: TaskPriority)
```

Manually escalate the task `priority` of this task to the `newPriority`.

Comparing Tasks

```
static func == (Task<Success, Failure>, Task<Success, Failure>) -> Bool
```

Returns a Boolean value indicating whether two values are equal.

```
static func != (Self, Self) -> Bool
```

Returns a Boolean value indicating whether two values are not equal.

```
var hashValue: Int
```

The hash value.

```
func hash(into: inout Hasher)
```

Hashes the essential components of this value by feeding them into the given hasher.

Deprecated

~~`typealias Group`~~ Deprecated

~~`typealias Handle`~~ Deprecated

~~`typealias Priority`~~ Deprecated

~~`static func CancellationError() -> CancellationError`~~ Deprecated

~~`func getResult() async -> Result<Success, Failure>`~~ Deprecated

~~`func get() async throws -> Success`~~ Deprecated

~~`func get() async -> Success`~~ Deprecated

~~`static func sleep(UInt64) async`~~ Deprecated

~~static func suspend() async~~ Deprecated

~~static func runDetached(priority: TaskPriority?, operation: () async throws -> Success) -> Task<Success, any Error>~~

Deprecated, available only for source compatibility reasons.

Deprecated

~~static func runDetached(priority: TaskPriority?, operation: () async -> Success) -> Task<Success, Never>~~

Deprecated, available only for source compatibility reasons.

Deprecated

~~static func startSynchronously(name: String?, priority: TaskPriority?, sending () async -> Success) -> Task<Success, Never>~~

Deprecated

~~static func startSynchronously(name: String?, priority: TaskPriority?, sending () async throws -> Success) -> Task<Success, any Error>~~

Deprecated

~~static func withCancellationHandler<T>(handler: () -> Void, operation: () async throws -> T) async rethrows -> T~~

Deprecated

~~static func withGroup<TaskResult, BodyResult>(resultType: TaskResult.Type, returning: BodyResult.Type, body: (inout Task<Success, Failure>.Group<TaskResult>) async throws -> BodyResult) async rethrows -> BodyResult~~

Deprecated

Default Implementations

- Equatable Implementations
- Hashable Implementations

Relationships

Conforms To

`Copyable`

Conforms when `Success` conforms to `Copyable`, `Success` conforms to `Escapable`, `Success` conforms to `Sendable`, and `Failure` conforms to `Error`.

`Equatable`

Conforms when `Success` conforms to `Copyable`, `Success` conforms to `Escapable`, `Success` conforms to `Sendable`, and `Failure` conforms to `Error`.

`Hashable`

Conforms when `Success` conforms to `Copyable`, `Success` conforms to `Escapable`, `Success` conforms to `Sendable`, and `Failure` conforms to `Error`.

`Sendable`
`SendableMetatype`

See Also

Tasks

`struct TaskGroup`

A group that contains dynamically created child tasks.

`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: ChildTaskResult.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.