Function

withThrowingDiscardingTaskGroup(returning:isolation:body:)

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

iOS 17.0+ | iPadOS 17.0+ | Mac Catalyst 17.0+ | macOS 14.0+ | tvOS 17.0+ | visionOS 1.0+ | watchOS 10.0+

```
@backDeployed(before: macOS 15.0, iOS 18.0, watchOS 11.0, tvOS 18.0, visionOS 2.0)
func withThrowingDiscardingTaskGroup<GroupResult>(
    returning returnType: GroupResult.Type = GroupResult.self,
    isolation: isolated (any Actor)? = #isolation,
    body: (inout ThrowingDiscardingTaskGroup<any Error>) async throws -> Group
Result
) async throws -> GroupResult
```

Discussion

Unlike a `ThrowingTaskGroup`, the child tasks as well as their results are discarded as soon as the tasks complete. This prevents the discarding task group from accumulating many results waiting to be consumed, and is best applied in situations where the result of a child task is some form of side-effect.

A group *always* waits for all of its child tasks to complete before it returns. Even canceled tasks must run until completion before this function returns. Canceled child tasks cooperatively react to cancellation and attempt to return as early as possible. After this function returns, the task group is always empty.

It is not possible to explicitly await completion of child-tasks, however the group will automatically await *all* child task completions before returning from this function:

```
try await withThrowingDiscardingTaskGroup(of: Void.self) { group in
  group.addTask { /* slow-task */ }
  // slow-task executes...
}
// guaranteed that slow-task has completed and the group is empty & destroyed
```

Refer to `TaskGroup` documentation for detailed discussion of semantics shared between all task groups.

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(priority:operation:)` to create a new task in a canceled group, that task is immediately canceled after creation. Alternatively, you can call `asyncUnless Cancelled(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.

Error Handling and Implicit Cancellation

Since it is not possible to explicitly await individual task completions, it is also not possible to "re-throw" an error thrown by one of the child tasks using the same pattern as one would in a `ThrowingTaskGroup`:

```
// ThrowingTaskGroup, pattern not applicable to ThrowingDiscardingTaskGroup
try await withThrowingTaskGroup(of: Void.self) { group in
  group.addTask { try boom() }
  try await group.next() // re-throws "boom"
}
```

Since discarding task groups don't have access to `next()`, this pattern cannot be used. Instead, a *throwing discarding task group implicitly cancels itself whenever any of its child tasks throws*.

The *first error* thrown inside such task group is then retained and thrown out of the `withThrowingDiscardingTaskGroup` method when it returns.

```
try await withThrowingDiscardingTaskGroup { group in
  group.addTask { try boom(1) }
  group.addTask { try boom(2, after: .seconds(5)) }
  group.addTask { try boom(3, after: .seconds(5)) }
}
```

Generally, this suits the typical use cases of a discarding task group well, however, if you want to prevent specific errors from canceling the group you can catch them inside the child task's body like this:

```
try await withThrowingDiscardingTaskGroup { group in
  group.addTask {
    do {
      try boom(1)
    } catch is HarmlessError {
      return
    }
  }
  group.addTask {
    try boom(2, after: .seconds(5))
  }
}
```

See Also

Tasks

`struct Task`

A unit of asynchronous work.

`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.

```
struct UnsafeCurrentTask
```

An unsafe reference to the current task.