Pool

August 17, 2026

Pool

Kind: Interface

Source: `src/utils/concurrent.ts`

Part of: Utils

`Pool<T>` defines an asynchronous work contract that returns a value of type `T`. Its `run` method lets concurrency code invoke pooled work through a shared `Promise`-based interface.

Diagram

```
graph LR
  Caller[Caller] --> Pool[Pool<T>]
  Pool --> Run[run()]
  Run --> Result[Promise<T>]
```

Usage

```
async function execute<T>(pool: Pool<T>): Promise<T> {
  return pool.run();
}

declare const taskPool: Pool<string>;

const result = await execute(taskPool);

console.log(result);
```

AI Coding Instructions

- Treat `run` as asynchronous and await or return its `Promise`.
- Preserve the generic `T` type when passing a `Pool` between functions.

- Implement `run` so it resolves with the expected result type.
- Handle rejected promises at the caller or at the integration boundary.

How it works

`Pool` is an exported TypeScript interface for an object that runs a zero-argument callback and returns a `Promise` for the callback's result. Its sole member is the generic `run<T>(fn: () => T): Promise<T>` method. [src/utils/concurrent.ts:8-10]

`createPool()` returns this interface and implements its concurrency behavior. It defaults an omitted or falsy `concurrency` value to `1024`; a concurrency value of `Infinity` returns a `Pool` whose `run` method immediately invokes the callback through an async function, without tracking active work. [src/utils/concurrent.ts:6,12-25]

For finite concurrency, each started callback occupies a marker in an internal `Set`. When the active-marker count is at least the configured limit, `run` creates or reuses a result promise, schedules another attempt with `setTimeout`, and returns that promise without invoking the callback yet. [src/utils/concurrent.ts:28-40] Once a callback can start, `run` awaits it and returns its result. [src/utils/concurrent.ts:39-52] Tests invoke multiple `run` calls and verify that only up to the configured concurrency are running before the blocked callbacks are released. [src/utils/concurrent.test.ts:13-37]

When `createPool` receives a truthy `interval`, a completed callback's marker remains in the active set until a timer deletes it after that many milliseconds; otherwise, the marker is deleted immediately after the callback resolves. [src/utils/concurrent.ts:41-46] Consequently, the interval delays later callbacks after completion, and the tests check that groups of callbacks begin at least approximately one interval apart. [src/utils/concurrent.test.ts:40-67]

The visible implementation has no input validation for the callback, concurrency, or interval. [src/utils/concurrent.ts:8-18] It also does not catch callback failures: if `await fn()` throws or rejects, execution exits before marker deletion, so the occupied marker remains in the finite pool. [src/utils/concurrent.ts:39-46]