2636. Promise Pool

Medium Leetcode Link

The problem entails creating a promisePool function that manages the execution of an array of asynchronous functions within a

Problem Description

given limit of how many can run concurrently. The parameter functions is an array of these asynchronous functions, and n is the pool limit or the maximum number of promises that can be pending at once. The goal of promisePool is to ensure that at any given moment, no more than n functions from the functions array are in the process

of execution. If n is set to 1, then the functions are executed one after the other in series. For higher values of n, up to n functions are started simultaneously. Whenever one promise resolves (a function completes its execution), if there are still functions left to execute, another one is started

so that there are always up to n functions running concurrently, until there are no more functions left to start.

The function should return a promise that ultimately resolves when all of the asynchronous functions have resolved. One key aspect to note is that all the functions are guaranteed never to reject, which simplifies the error handling aspect of the implementation.

The solution provided outlines a strategy to manage the execution of asynchronous functions with a promise pool. The core idea is to organize the functions into two groups: running which contains up to n functions that are currently being executed, and waiting

which contains the rest of the functions that are queued to be executed as soon as there's a free slot in the running pool. To achieve this, the solution maps the original functions array to a new array called wrappers. Each element in wrappers is an async function that, when called, will execute its corresponding function from the functions array.

As each wrapper async function resolves, it checks the waiting array for any functions that are waiting to be executed. If there's a function waiting (waiting.shift()), the wrapper function will continue execution by awaiting the next waiting function. This chaining ensures that as soon as a function completes, a new one begins execution if available, maintaining the pool limit.

The initial running functions are started by slicing the wrappers list from the beginning up to the pool limit n. This starts execution for the first n functions. The remaining functions are placed in the waiting list and are executed in order as the running functions complete.

The Promise.all() function is then used on the initial running functions. This call will resolve once all the initially started functions (and as a result of the chaining, all the functions in waiting) have resolved. This way, when Promise.all() resolves, we know that all functions have been executed, respecting the pool limit at all times.

The resulting Promise.all() call is returned by the promisePool function, which resolves when the last promise in the pool completes, signaling that all asynchronous functions have finished executing.

closures, asynchronous functions, and the Promise, all() method to manage the concurrency. Here's a step-by-step walk-through: 1. Map Functions to Async Wrappers: The original array of async functions (functions) is mapped to a new array called wrappers.

The implementation of the solution for the promisePool can be divided into several steps, which involves the use of JavaScript

Awaits the completion of its corresponding function from the original array.

running functions completes.

operations as soon as older ones finish.

Example Walkthrough

Each element of wrappers is an async function that:

function potentially triggers the start of another.

promisePool has been executed respecting the pool limit.

Solution Approach

 Looks at a shared waiting queue to see if there are other functions waiting to execute. If a function is waiting, it dequeues (waiting.shift()) and awaits the completion of that function. This is essentially a recursive step -- the completion of one

2. Initialize Running and Waiting Pools: Divide the wrappers into two groups: the running group representing the concurrent n

functions that can run simultaneously and the waiting queue for the remaining functions that will be executed once one of the

- 3. Start Initial Execution: Begin executing functions from the running group using Promise.all(). The Promise.all() function takes an iterable of promises and returns a single Promise that resolves to an array of the results of the input promises when all of them resolve or when one is rejected (the latter is not a concern here as functions are guaranteed never to reject).
- 4. Chain Execution of Waiting Functions: As each wrapper async function resolves, it triggers the next function in the waiting queue. This is done by calling waiting.shift() inside the async wrapper function after the awaited function execution. This chaining maintains the invariant that up to n functions are running concurrently until there are no more functions left to execute.

5. Complete All Executions: The promise returned by Promise.all() will only resolve once all of the promises in the running array

have resolved. However, due to the chaining implemented in the wrapper functions, this also implies that all functions in the

waiting queue have finished running. Hence, when the Promise all() promise resolves, we know that every function in the

- 6. Return Final Promise: Finally, the promise resolved by Promise.all() is returned from the promisePool function. This promise therefore represents the completion of all async function executions while respecting the concurrency limit. This solution fundamentally employs the concept of concurrency control with a finite number of workers (running functions) and a queue (waiting functions). It ensures that no more than n async operations are being processed at the same time, starting new
- Let's illustrate the solution approach with a smaller example. Suppose we have an array of asynchronous functions that simply resolve after a certain delay and log their index. Let's say we have 5 such functions and our pool limit n is 2. Our asynchronous functions might look like this:

2 const function2 = () => new Promise(resolve => setTimeout(() => { console.log("Function 2 done"); resolve(); }, 1000)); const function3 = () => new Promise(resolve => setTimeout(() => { console.log("Function 3 done"); resolve(); }, 1500)); const function4 = () => new Promise(resolve => setTimeout(() => { console.log("Function 4 done"); resolve(); }, 500)); const function5 = () => new Promise(resolve => setTimeout(() => { console.log("Function 5 done"); resolve(); }, 1000)); 7 const functions = [function1, function2, function3, function4, function5];

const function1 = () => new Promise(resolve => setTimeout(() => { console.log("Function 1 done"); resolve(); }, 2000));

1. Start with function1 and function2: Since our pool limit is 2, function1 and function2 will start executing simultaneously.

3. function3 and function1 are running: Now, function3 is running alongside function1, which has not yet completed.

4. function1 finishes; function4 starts: Once function1 completes, function4 is dequeued and starts executing.

Implementing the promisePool function with a pool limit of n = 2, we would expect the following behavior:

2. function2 finishes first: Despite function1 starting first, function2 will finish before function1 due to its shorter timeout. Upon function2's completion, function3 will start because it's next in the queue.

Python Solution

9

10

12

13

14

21

22

24

25

26

27

28

29

35

in the queue, no new functions are started.

from asyncio import Semaphore, create_task, gather

from typing import Callable, List

running at the same time.

async with semaphore:

return await fn()

function5 will begin. 6. function3 finishes; all functions started: By this time, function3 would likely have finished, and since function5 was the last one

5. function4 finishes quickly; function5 starts: Since function4 has the shortest timeout, it will finish quickly, after which

7. function5 finishes; end: Lastly, function5 finishes, and since it was the last function to be executed, the promisePool should resolve now. As we can see, the pool starts with two functions and whenever a function finishes, it triggers the next function in line if there is any.

At no point do more than two functions run concurrently. The promisePool function orchestrates the asynchronous functions'

execution respecting the concurrency limit and eventually resolves when all functions have completed their execution.

Executes a pool of asynchronous functions concurrently, but with a limited number of async functions

:return: A list of results from the async function executions, when all functions have completed.

Wraps an async function to acquire a semaphore before execution and release it afterward,

ensuring that the number of concurrently running functions does not exceed the limit.

Wrap each async function with the semaphore logic and schedule it as a task.

// Create a list of asynchronous operations using CompletableFuture.supplyAsync

List<Supplier<CompletableFuture<Object>>> asyncFunctions = new ArrayList<>();

CompletableFuture<List<Object>> result = promisePool(asyncFunctions, 2);

asyncFunctions.add(() -> CompletableFuture.supplyAsync(() -> {

asyncFunctions.add(() -> CompletableFuture.supplyAsync(() -> {

// Perform some asynchronous operation here

// Perform some asynchronous operation here

return "Result of async task 1";

return "Result of async task 2";

// Add more async tasks as needed

// Execute the promisePool method

tasks = [create_task(run_with_semaphore(fn)) for fn in async_functions]

:param limit: The maximum number of async functions that can be running simultaneously.

Defines a type for an asynchronous function that returns any value. AsyncFunction = Callable[[], "Coroutine[Any, Any, Any]"] async def promise_pool(async_functions: List[AsyncFunction], limit: int) -> List[Any]:

:param async_functions: A list of functions that are asynchronous and return promises (in Python, 'awaitables').

15 # A semaphore to control the number of async functions that can run at the same time. 16 semaphore = Semaphore(limit) 17 18 19 async def run_with_semaphore(fn: AsyncFunction): 20

```
# Use asyncio.gather to run tasks concurrently and wait until all are finished.
30
31
       return await gather(*tasks)
32
  # Example usage of the function (assuming example async functions async_foo and async_bar):
  # results = asyncio.run(promise_pool([async_foo, async_bar], limit=2))
```

Java Solution

1 import java.util.ArrayList;

```
import java.util.LinkedList;
3 import java.util.List;
4 import java.util.concurrent.CompletableFuture;
5 import java.util.concurrent.Executor;
6 import java.util.concurrent.Executors;
7 import java.util.function.Supplier;
8 import java.util.stream.Collectors;
9
   /**
10
    * Executes a pool of asynchronous operations concurrently, but limits the number of operations
    * running at the same time.
13
    * @param asyncFunctions A list of suppliers that provide CompletableFutures.
   * @param limit The maximum number of CompletableFutures that can be running concurrently.
    * @return A CompletableFuture that resolves when all the given suppliers have completed.
17
   public CompletableFuture<List<Object>> promisePool(
           List<Supplier<CompletableFuture<Object>>> asyncFunctions, int limit) {
19
20
21
       // Create an executor with a fixed thread pool to control the number of concurrent threads.
22
       Executor executor = Executors.newFixedThreadPool(limit);
23
24
       // A list to hold the CompletableFutures created by running the async functions.
25
       List<CompletableFuture<Object>> futures = new LinkedList<>();
26
27
       // Add each async function as a CompletableFuture in the list to be executed.
28
       for (Supplier<CompletableFuture<Object>> function : asyncFunctions) {
           CompletableFuture<Object> future = CompletableFuture.supplyAsync(() -> function.get().join(), executor);
           futures.add(future);
30
31
32
33
       // Combine all the CompletableFutures into a single CompletableFuture that contains a list of results.
       // This CompletableFuture will complete when all the individual CompletableFutures are complete.
34
35
       return CompletableFuture.allOf(futures.toArray(new CompletableFuture[0]))
               .thenApply(v -> futures.stream()
36
37
                       .map(CompletableFuture::join)
38
                       .collect(Collectors.toList())
39
               );
40 }
41
42 // Example usage of the promisePool method:
   public static void main(String[] args) {
44
```

}));

}));

45

46

49

50

51

52

54

55

56

57

58

```
59
       // Process the result when all async operations are complete
60
       result.thenAccept(results -> results.forEach(System.out::println));
61
62 }
63
C++ Solution
  1 #include <vector>
                              // Required for std::vector
                              // Required for std::future and std::async
  2 #include <future>
                              // Required for std::function
    #include <functional>
                              // Required for std::queue
     #include <queue>
  6 // Define a type for a function that returns a std::future of any type
    using AsyncFunction = std::function<std::future<void>()>;
  8
     /**
  9
     * Executes a pool of std::future objects concurrently, but with a limited number of std::future
      * objects running at the same time.
     * @param async_functions - A vector of functions that return std::future objects.
     * @param limit - The maximum number of std::future objects that can be running at the same time.
      * @returns A std::future that resolves when all the given functions have completed.
 15
     */
 16 std::future<void> promise_pool(std::vector<AsyncFunction> async_functions, int limit) {
 17
         // A queue to hold the functions that are currently waiting to be executed.
         std::queue<AsyncFunction> waiting_functions;
 18
 19
 20
         // Vector to keep track of futures that are currently running
 21
         std::vector<std::future<void>> running_futures;
 22
 23
         // Iterate over async_functions and enqueue them into waiting_functions
 24
         for (auto& fn : async_functions) {
 25
             waiting_functions.push(fn);
 26
 27
 28
         // Function that checks and starts execution of waiting functions as slots become free.
 29
         auto start_waiting_functions = [&running_futures, &waiting_functions, limit]() {
             while (!waiting_functions.empty() && running_futures.size() < static_cast<size_t>(limit)) {
 30
 31
                 auto function_to_run = waiting_functions.front();
 32
                 waiting functions.pop();
 33
                 running_futures.push_back(function_to_run());
 34
         };
 35
 36
         // Function that waits for all futures to complete
 37
 38
         auto wait_all = [&running_futures]() -> void {
 39
             for (auto &f : running_futures) {
 40
                 if(f.valid()) {
 41
                     f.wait();
 42
 43
 44
         };
 45
 46
         // Start functions until reaching the limit
 47
         start_waiting_functions();
 48
 49
         // A promise for notifying completion of all async functions
 50
         std::promise<void> completion_promise;
 51
 52
         // Setup async task to wait for all tasks to finish
 53
         auto completion_task = std::async(std::launch::async, [&]() {
             while (!running_futures.empty()) {
 54
                 for (auto it = running_futures.begin(); it != running_futures.end();) {
 55
 56
                     if(it->wait_for(std::chrono::seconds(0)) == std::future_status::ready) {
                         it = running_futures.erase(it);
 57
 58
                         start_waiting_functions();
 59
                     } else {
 60
                         ++it;
 61
 63
 64
             completion_promise.set_value();
 65
         });
 66
 67
         // Return the future associated with the completion promise
 68
         return completion_promise.get_future();
```

13 14 15 16 await fn();

Typescript Solution

type AsyncFunction = () => Promise<any>;

// Defines a type for a function that returns a Promise of any type.

* @param asyncFunctions - An array of functions that return Promises.

69 }

/**

70

```
* @param limit - The maximum number of Promises that can be running at the same time.
    * @returns A Promise that resolves when all the given functions have completed.
    function promisePool(asyncFunctions: AsyncFunction[], limit: number): Promise<any[]> {
       // An array to hold the functions that are currently waiting to be executed.
12
       const waitingFunctions: AsyncFunction[] = [];
       // Wraps each async function to manage the execution of the next function in the queue.
       const wrappedFunctions = asyncFunctions.map(fn => async () => {
17
           const next = waitingFunctions.shift();
           if (next) {
18
               await next();
19
       });
21
22
23
       // Initialize two arrays: 'currentlyRunning' for functions that should start immediately, and
24
       // 'waitingFunctions' for the rest that should wait for free slots.
       const currentlyRunning = wrappedFunctions.slice(0, limit);
25
       waitingFunctions.push(...wrappedFunctions.slice(limit));
26
27
28
       // Trigger the execution of the currently running functions and use Promise.all() to wait until all have finished.
       return Promise.all(currentlyRunning.map(fn => fn()));
29
30 }
31
Time and Space Complexity
The time complexity of the promisePool function is determined by the number of function executions (functions, length) and the
concurrency level (n).
Since you have not provided the specifics of what each function in the functions array does, if we assume that each function takes
an equal amount of time to complete and that there is perfect parallelism with no additional overhead for managing the promises, the
```

* Executes a pool of Promises concurrently, but with a limited number of Promises running at the same time.

time complexity would be O(functions.length / n). This is because you run n functions in parallel until all functions have been executed. However, it should be noted that in practice, the actual time complexity can be affected by various factors, such as the nature of the

functions. length wrapper functions, but they are derived from wrappers, so there is no additional space used that scales with the input.

If we include auxiliary space for the execution context of asynchronous function calls, the space complexity might be higher due to

the async calls and closures created for each function. This complexity can be considered as O(n) because at most n functions will

I/O operations, the performance characteristics of the underlying system, and the overhead of promise scheduling and execution.

The space complexity of this code is O(functions.length) because it creates a wrapping function for each function in functions—

stored in wrappers. It also stores the actively running and waiting promises. Both running and waiting arrays store at most

be running at the same time, each needing its own execution context. The final space complexity is thus the larger of O(functions.length) and O(n), which is the space needed for the wrappers array and the maximum concurrent async execution contexts.