2721. Execute Asynchronous Functions in Parallel

Medium

Problem Description In this problem, we are given an array functions which contains a list of asynchronous functions. Each of these functions, when

called, returns a promise. The task is to execute all these functions in parallel without using the built-in Promise.all function and return a new promise promise. The new promise should:

Leetcode Link

Resolve when all of the promises returned from the functions in functions array have been resolved. The resolved value should

relying on Promise.all, which simplifies the coordination of asynchronous promises.

be an array containing the resolved values from each promise, maintaining the same order as the original functions array. Reject if any one of the promises returned from the functions array is rejected. The reason for rejection should be the same as the first rejection that occurs among the promises.

The challenge is to manage these asynchronous operations manually to ensure correct synchronization and error handling without

To arrive at a solution, let's consider the manual steps we would need to track and synchronize the various promises:

1. We need to keep a count of how many promises have resolved so that we know when all promises have been settled. This can be done using a counter variable.

2. We must store the results of each promise in an array, maintaining the order of how they appeared in the input array. When a

have a rejection handler for each promise.

performs the above-mentioned steps to resolve or reject based on the results of the promise executions.

- promise resolves, we will save its result in the corresponding position in this array. 3. If any promise rejects, we should immediately reject the entire set without waiting for other promises to finish. This requires us to
- 4. As each promise resolves, we increment the counter. When the counter matches the number of functions, we know all promises have resolved, and we can resolve our promise with the array of results. 5. If a promise rejects, we call the reject function with the error that was thrown. This rejection should happen only once.

Implementing this logic, we can construct a promiseAll function, which creates a new promise (return new Promise<T[]>) and

Solution Approach

The implementation of the promiseAll function follows these steps: 1. We initialize a Promise object using new Promise<T[]>((resolve, reject) => {...}). Within this Promise, we manage the

2. We declare a counter variable cnt initialized to zero, which we'll use to keep track of how many promises have been settled

3. We create an array ans that will hold the resolved values of the promises. Its length is the same as the functions array, ensuring

4. We loop over the array of functions using for (let i = 0; i < functions.length; ++i) and call each function, which returns a

results.

5. For each returned promise:

(resolved).

promise.

Below is an excerpt of the essential part of the implementation:

const ans = new Array(functions.length);

ans[i] = res;

.then(res => {

Example Walkthrough

would look step-by-step:

store the resolved values.

2. Execution and Management:

1. Initialization:

12 }

for (let i = 0; i < functions.length; ++i) {</pre>

execution and coordination of all promises returned by the functions.

we maintain the order of resolved values as per their corresponding functions.

results, and promises' .then() and .catch() methods for asynchronous flow control.

understanding of concurrency and error handling with promises in JavaScript.

• We attach a .then(res => {}) handler that activates when the promise resolves. In this handler, we store the result as ans[i] = res and increment the cnt counter. We then check if cnt is equal to functions. length to determine if all promises have been resolved, in which case we call resolve(ans) to resolve the promiseAll function's promise with the array of

forwarding the first error we encounter to the promiseAll function's promise, thus rejecting it.

The absence of a <code>catch()</code> block after <code>resolve(ans)</code> is intentional as any error occurring in <code>resolve</code> (which should not happen in standard usage) should not be caught by this block, but be allowed to propagate naturally.

This implementation makes use of closures to keep track of the index and result of each asynchronous operation, arrays to store the

We also attach a .catch(err => {}) handler to handle any rejections. If a promise rejects, we call reject(err) immediately,

if (cnt === functions.length) { resolve(ans); 10 .catch(reject); 11

This approach effectively simulates the behavior of Promise.all without directly relying on it, demonstrating a fundamental

follows: function1() simulates an asynchronous operation that resolves after 1 second with the string "Result 1".

Using the given solution approach, we can execute these functions and manage their promises manually. Here's how the process

• Inside this new Promise, we initialize a counter cnt to zero and an array ans of the same length as the functions array to

Suppose we have an array of three asynchronous functions that we want to execute in parallel. These functions are defined as

• function2() simulates an asynchronous operation that resolves after 2 seconds with the string "Result 2".

function3() simulates an asynchronous operation that resolves after 3 seconds with the string "Result 3".

• We define a new Promise object promiseAll that will manage the execution of all functions.

We begin a for loop to execute each asynchronous function in the functions array.

storing the result at the corresponding index in the ans array and incrementing cnt.

results or errors as they occur, achieving the same behavior as Promise.all without using it directly.

Executes an array of functions that each returns an asyncio coroutine (future),

If any of the coroutine rejects, this function will reject with the reason from

We will collect all coroutine objects first to later await them concurrently

:return: A coroutine that resolves with a list of the resolved values from the coroutines or rejects.

:param promise_fns: A list of functions that each returns a coroutine.

Now we await for all gathered coroutine objects concurrently

and resolves when all have successfully completed.

the first coroutine that rejected.

for promise_fn in promise_fns:

except Exception as e:

results_array.append(promise_fn())

return await asyncio.gather(*results_array)

results = await promise_all(promises)

print(f"An error occurred: {e}")

2 import java.util.concurrent.CompletableFuture;

import java.util.function.Supplier;

import java.util.stream.Collectors;

import java.util.stream.IntStream;

public class PromiseAll {

* @param <T>

import java.util.concurrent.ExecutionException;

CompletableFuture<Void> allDoneFuture =

return allDoneFuture.thenApply(v ->

futuresList.stream()

public static void main(String[] args) {

// Example usage:

print(results) # Expected output: [42]

results_array = []

3. Resolution Handling:

4. Completion Check:

promise.

5. Error Handling:

Python Solution

9

10

11

13

14

15

16

17

19

20

21

22

23

24

29 #

30 #

31 #

32 #

33 #

34

9

10

11

12

13

14

15

16

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

42

43

44

45

46

52 /*

54

56

57

59

62

60 }

61 */

/**

*/

10

11

12

13

14

15

16

17

18

19

20

22

23

24

26

28

29

30

33

34

35

53 try {

});

auto results = promiseAll<int>({

} catch (const AsyncOperationFailed& e) {

// Handle the failure case here

let resolvedCount = 0;

promiseFn()

});

});

// Example usage:

Time Complexity

// Process results here

Typescript Solution

[]() { return std::async(std::launch::async, []() { return 42; }); }

* Executes an array of functions that returns Promises, and resolves when all have successfully completed.

* @param {(() => Promise<T>)[]} promiseFns - An array of functions that each returns a promise.

async function promiseAll<T>(promiseFns: (() => Promise<T>)[]): Promise<T[]> {

// Save the resolved value in the array

if (resolvedCount === promiseFns.length) {

promise.then(console.log).catch(console.error); // Expected output: [42]

return new Promise<T[]>((resolve, reject) => {

// Iterate over the promise functions

.then(value => {

.catch(error => {

reject(error);

// const promise = promiseAll([() => Promise.resolve(42)]);

promiseFns.forEach((promiseFn, index) => {

// Execute each promise function

resolvedCount++;

// Counter for promises that have been resolved

const resultsArray = new Array<T>(promiseFns.length);

resultsArray[index] = value;

resolve(resultsArray);

* If any of the Promises reject, this function will reject with the reason from the first promise that rejected.

// Array to accumulate the resolved results. Its length is pre-set to match the array of promises

* @returns {Promise<T[]>} A promise that resolves with an array of the resolved values from the input promises or rejects.

// If all promises have resolved, resolve the outer promise with the results array

// If any of the promises reject, reject the outer promise with the error

26 # Example usage:

27 # async def main():

try:

Java Solution

1 import java.util.List;

 We attach a .then(res => {}) handler to each promise. When function1() resolves after 1 second, the .then() handler for that function will execute, storing the result in ans [0] and incrementing cnt. • The same happens for function2() and function3(); each one's resolution is handled by the respective .then() handler,

After each resolution, inside its .then() handler, we check whether cnt has reached the length of the functions array. When

the last function (function3()) resolves and cnt becomes 3, we call resolve(ans) to resolve the promiseAll function's

• At this point, the promiseAll promise will successfully resolve with the array ["Result 1", "Result 2", "Result 3"].

This example demonstrates how the provided solution approach coordinates multiple asynchronous operations and processes their

rejection that occurred, and the process would stop without awaiting the resolution of other promises.

 Additionally, each promise has a catch(err => {}) handler attached. If any one of them had rejected, the reject(err) function within the .catch() handler would execute, rejecting the promiseAll function's promise with the reason for the first

For each function, we invoke it and receive a promise back.

1 import asyncio from typing import Callable, List, TypeVar, Generic # Creating a type variable to represent the generic type T = TypeVar('T') async def promise_all(promise_fns: List[Callable[[], asyncio.Future[T]]]) -> asyncio.Future[List[T]]:

35 # asyncio.run(main())

* Executes a list of suppliers that return CompletableFutures, and completes when all have successfully completed.

.map(Supplier::get)

.collect(Collectors.toList());

* If any of the CompletableFutures complete exceptionally, this function will complete exceptionally

The result type returned by this CompletableFuture's suppliers.

CompletableFuture.allOf(futuresList.toArray(new CompletableFuture[0]));

List<Integer> resultList = allFutures.get(); // This blocks until the future completes

* with the cause from the first future that completed exceptionally.

* @param promiseFns A list of suppliers that each returns a CompletableFuture.

.map(CompletableFuture::join)

.collect(Collectors.toList())

List<Supplier<CompletableFuture<Integer>>> suppliers = List.of(

CompletableFuture<List<Integer>> allFutures = promiseAll(suppliers);

System.out.println(resultList); // Expected output: [42]

} catch (InterruptedException | ExecutionException e) {

() -> CompletableFuture.completedFuture(42)

promises = [lambda: asyncio.Future().set_result(42)] # Creating a list of lambda functions returning coroutines

```
* @return A CompletableFuture that, when completed normally, contains a list of the values obtained by
17
                  each of the supplied CompletableFutures, or completes exceptionally if any of the provided
18
19
                  CompletableFutures complete exceptionally.
20
21
       public static <T> CompletableFuture<List<T>> promiseAll(List<Supplier<CompletableFuture<T>>> promiseFns) {
           List<CompletableFuture<T>> futuresList = promiseFns.stream()
22
```

);

);

try {

/**

```
e.printStackTrace(); // Handle the exception if any of the futures completed exceptionally
48
49
50
51 }
52
C++ Solution
   #include <vector>
 2 #include <future>
   #include <functional>
   #include <stdexcept>
   // Custom exception type to handle when an asynchronous operation fails
   struct AsyncOperationFailed : public std::runtime_error {
       AsyncOperationFailed(const char* message) : std::runtime_error(message) {}
 9
10
11 /**
    * Executes a vector of functions that returns futures, and gets the results when all have successfully completed.
    * If any of the futures report a failure, this function throws an exception with the reason from the first failed one.
14
   * @tparam T The type of the results that the futures are expected to return.
   * @param futureFns A vector of functions that each returns a future of type T.
    * @return A vector of the resolved values from the futures.
   template<typename T>
   std::vector<T> promiseAll(std::vector<std::function<std::future<T>()>> futureFns) {
       // Vector to store the futures returned by the input functions
       std::vector<std::future<T>> futures;
       // Reserve the space needed in the vector to store the futures
24
       futures.reserve(futureFns.size());
25
26
27
       // Execute each function to get the future and add it to the vector
28
       for (auto& fn : futureFns) {
29
           futures.push_back(fn());
30
31
32
       // Vector to accumulate the results from the futures
33
       std::vector<T> results;
       results.reserve(futureFns.size());
34
35
       // Iterate through the futures and get their results
36
       for (auto& fut : futures) {
37
38
           try {
39
               // Get the result from the future and add it to the results vector
40
               results.push_back(fut.get());
           } catch (...) {
41
               // If getting the result throws, forward the exception
               throw AsyncOperationFailed("One of the asynchronous operations failed.");
43
44
45
46
       // Return the full vector of results when all futures have been processed
47
       return results;
48
50
51 // Example usage (within a function that allows exceptions):
```

The given promiseAll TypeScript code aims to mimic the behavior of Promise.all, running multiple promises in parallel and gathering their results. Now let's analyze its complexities:

Time and Space Complexity

function would also have a time complexity of 0(n). However, if we consider the loop's contribution, since there is a one-time initialization for each promise, this is effectively 0(1) for each promise leading to 0(m) for m promises, which could be considered as part of the setup process. But this does not affect the overall time complexity in terms of waiting for promises to resolve, which will

minuscule amount of time spent in the loop) and then handling their results as they resolve. The promise that takes the longest to

resolve dictates the overall time complexity. If we assume that the slowest promise in the array takes O(n) time, then the promiseAll

For time complexity, the promiseAll function operates by initiating all the promises nearly simultaneously (though there is a

Space complexity pertains to the extra space or storage needed by the algorithm as the input size grows. In this case, the

promiseAll function creates an array ans of size equal to the number of promises to store their results. Assuming each result takes

Space Complexity

• Time Complexity: 0(n)

 Space Complexity: 0(m) where m is the number of promises. We do not consider the space taken by the input itself or the promises' internal space usage; we

0(1) space, the total space required is proportional to the number of promises. Hence,

still be dominated by the promise that takes the longest time. Therefore,

where n is the time taken by the slowest individual promise to resolve.

only consider the extra space that the promiseAll function allocates, which is the ans array and the cnt variable, which is a single

counter. It's important to note, however, that the actual time a promise takes to resolve would be determined by the systems or processes it depends on (e.g., I/O operations, network requests), and the complexity O(n) represents a simplification for the time any single promise takes to resolve in the worst-case scenario. Since all promises are processed in parallel, the overall time until all promises resolve would depend on the one that takes the longest.