2723. Add Two Promises

Problem Description In this problem, we are working with asynchronous operations in JavaScript, specifically with promises. A promise is an object

Easy

promise2, both of which are expected to resolve with a numerical value. The task is to create and return a new promise that will resolve with the sum of the two numbers provided by promise1 and promise2. Intuition

representing the eventual completion or failure of an asynchronous operation. We are given two such objects, promise1 and

The key to solving this problem is understanding how promises work and how to handle the values they resolve with. The async keyword allows us to write asynchronous code in a way that looks synchronous, and the await keyword pauses the execution of

the function until a promise is resolved. By using await on both promise1 and promise2, we can get their resolved values. Once both promises have been resolved, we can then simply add these two numerical values together. The sum is then implicitly

returned as a resolved promise due to the async function's behavior—any value returned by an async function is automatically

Inside the function, we use the await keyword before promise1 and promise2. This instructs JavaScript to pause the

When await promise1 is executed, the function execution is paused until promise1 is either fulfilled or rejected. If it's fulfilled,

The pattern used is async/await, which is syntactic sugar over JavaScript Promises, providing a simpler and cleaner way to

handle asynchronous operations. It allows for writing promise-based code as if it were synchronous, but without blocking the

This solution leverages the concurrency nature of promises to potentially execute both promise1 and promise2 at the same time,

Therefore, the solution is concise, clear, and leverages modern JavaScript features to work with asynchronous operations

execution of the function until each promise resolves. The await keyword can only be used within an async function.

3. The async function will return a promise that resolves with the sum of the two numbers.

wrapped in a Promise.resolve().

So, the "intuition" behind this solution is:

1. Wait for each promise to resolve using await.

2. Once both numbers are available, add them together.

Solution Approach

The implementation of the solution is straightforward given the understanding of how JavaScript promises and async/await work.

Here's a step-by-step breakdown of the algorithm:

An async function named addTwoPromises is defined, which accepts two parameters: promise1 and promise2. Both

parameters are expected to be promises that resolve to numbers.

the resolved value (a number) is returned. If promise1 is rejected, an error will be thrown, which can be caught using try...catch blocks.

Similarly, await promise2 pauses the function execution until promise2 resolves, after which it provides the resolved value (a

number). After obtaining both numbers, the function adds these two numbers together with the + operator. The expression (await

promise1) + (await promise2) computes the sum.

The resulting sum is then returned from the async function. Because the function is async, this return value is automatically wrapped in a promise. Essentially, return (await promise1) + (await promise2); is equivalent to return Promise.resolve((await promise1) + (await promise2));.

reducing the overall waiting time for both promises to resolve.

const promise1 = new Promise((resolve, reject) => {

const promise2 = new Promise((resolve, reject) => {

The data structure used here is the built-in JavaScript Promise, which is a proxy for a value that is not necessarily known when the promise is created. This lets asynchronous methods return values like synchronous methods: instead of immediately returning the final value, the asynchronous method returns a promise to supply the value at some point in the future.

To illustrate the solution approach, let's walk through a small example using two sample promises that resolve to numerical values.

Suppose we have two promises:

Example Walkthrough

main thread.

effectively.

});

});

1. We define an async function addTwoPromises that takes promise1 and promise2 as parameters. async function addTwoPromises(promise1, promise2) {

// The function body will be implemented according to the steps.

Let's go through the steps of the solution with these promises:

setTimeout(() => resolve(10), 1000); // promise1 resolves with 10 after 1 second

setTimeout(() => resolve(20), 500); // promise2 resolves with 20 after 0.5 seconds

These promises, when executed, will resolve with the numbers 10 and 20, respectively, after their set timeouts.

If we want to use the result of the addTwoPromises function, we can do so by attaching a then method or using an await within another async function:

addTwoPromises(promise1. promise2).then(sum => {

async function displayResult() {

case.

});

or,

import asyncio

And that concludes the example walkthrough of how to use the solution approach to add two numbers provided by promises

Function that adds the values resolved from two futures.

async def add_two_promises(future1, future2):

Takes two futures that resolve to numbers as parameters.

const result = await addTwoPromises(promise1, promise2);

console.log(result); // This will also log 30 to the console

This coroutine waits for two futures to resolve and adds their results.

:param future1: A future (async result) that resolves to a number.

:param future2: A future (async result) that resolves to a number.

:return: The sum of the two numbers when both futures have resolved.

Await the resolution of the first future and store the resulting value.

value2 = await future2

Initialize two futures and set their results to 2.

import java.util.concurrent.CompletableFuture:

import java.util.concurrent.ExecutionException;

// Example usage of the addTwoFutures method

// Compute the sum of both future's results

return null;

});

sumFuture.get();

std::promise<int> result promise;

int value1 = future1.get();

int value2 = future2.get();

// Set values to the promises

// Get futures from promises

// Return the sum of the two values.

// Example usage of the addTwoPromises function:

.then(console.log); // Expected output: 4

Return the sum of the two values.

print(result) # Expected output: 4

Example usage of the add two promises coroutine:

Initialize two futures and set their results to 2.

future1 = asyncio.ensure future(asyncio.sleep(0, result=2))

return value1 + value2

addTwoPromises(Promise.resolve(2), Promise.resolve(2))

Function that adds the values resolved from two futures.

return value1 + value2;

promise1.set value(2);

promise2.set_value(2);

trv {

// Log the result once computation is complete

public static void main(String[] args) {

future1 = asyncio.ensure future(asyncio.sleep(0, result=2))

future2 = asyncio.ensure_future(asyncio.sleep(0, result=2))

Run the main coroutine that utilizes the add_two_promises coroutine.

value1 = await future1

// Function that adds the values from two CompletableFutures. // Takes two CompletableFutures that complete with numbers as parameters.

public class FutureAdder {

asyncio.run(main())

Java

C++ #include <iostream>

// Function that adds the values obtained from two futures.

// Takes two futures that will provide numbers as parameters.

// Lambda that processes the addition of two future values

result_promise.set_value(value1 + value2);

std::future<int> future1 = promise1.get future();

std::future<int> future2 = promise2.get_future();

// Create two promises std::promise<int> promise1; std::promise<int> promise2;

promise2: Promise<number>,): Promise<number> { const value1 = await promise1; // Await the resolution of the second promise and store the resulting value. const value2 = await promise2;

Takes two futures that resolve to numbers as parameters. async def add_two_promises(future1, future2):

async def main():

import asyncio

asyncio.run(main()) **Time Complexity**

2. We await both promise1 and promise2 using the await keyword inside the async function. This way we can get their resolved values: async function addTwoPromises(promise1, promise2) { const value1 = await promise1; // Execution pauses here until promise1 resolves. const value2 = await promise2; // Execution pauses here until promise2 resolves. // The rest of the function... After both promises resolve, value1 has the value 10, and value2 has the value 20. We then add value1 and value2 and return the result: async function addTwoPromises(promise1, promise2) { const value1 = await promise1; const value2 = await promise2; return value1 + value2; // The function will return a Promise that resolves to 30

Since addTwoPromises is an async function, the returned result will be wrapped in a promise. Therefore, calling

addTwoPromises(promise1, promise2) will return a promise that resolves to the sum of the two values, which is 30 in this

asynchronously. Solution Implementation **Python**

console.log(sum); // This will log 30 to the console once the promise resolves

Return the sum of the two values. return value1 + value2 # Example usage of the add two promises coroutine: # - Resolves two futures, each with the value 2, and logs the sum, which is 4. async def main():

public static CompletableFuture<Integer> addTwoFutures(CompletableFuture<Integer> future1, CompletableFuture<Integer> future2) { // Combine both futures to sum the resolved numbers once both are completed return future1.thenCombine(future2, Integer::sum);

CompletableFuture<Integer> completableFuture1 = CompletableFuture.supplyAsync(() -> 2);

// Class containing the method to add values from two CompletableFutures

} catch (InterruptedException | ExecutionException ex) { ex.printStackTrace(); #include <future> #include <functional>

std::future<int> addTwoFutures(std::future<int>&& future1, std::future<int>&& future2) {

// Create a new std::promise<int> that will eventually hold the result

}; // Start asynchronous execution std::async(std::launch::async, compute, std::move(future1), std::move(future2), std::move(result_promise)); // Return the future that will provide the sum return result_future; // Example usage of the addTwoFutures function // - Resolves two futures, each with the value 2, and logs the sum, which is 4. int main() {

std::future<int> result = addTwoFutures(std::move(future1), std::move(future2)); // Wait for the result future to be ready and get its value, then print std::cout << "Expected output: " << result.get() << std::endl; // Expected output: 4</pre> return 0; **TypeScript** // Function that adds the values resolved from two promises. // Takes two promises that resolve to numbers as parameters. async function addTwoPromises(promise1: Promise<number>,

This coroutine waits for two futures to resolve and adds their results. :param future1: A future (async result) that resolves to a number. :param future2: A future (async result) that resolves to a number. :return: The sum of the two numbers when both futures have resolved. # Await the resolution of the first future and store the resulting value. value1 = await future1 # Await the resolution of the second future and store the resulting value. value2 = await future2

- Resolves two futures, each with the value 2, and logs the sum, which is 4.

result = await add two promises(asyncio.Future(), asyncio.Future())

// - Resolves two promises, each with the value 2, and logs the sum, which is 4.

future2 = asyncio.ensure_future(asyncio.sleep(0, result=2)) # Run the main coroutine that utilizes the add_two_promises coroutine. Time and Space Complexity

promise2 were to be awaited in parallel, the time complexity could be improved to O(max(T1, T2)).

unnecessarily. Therefore, if T1 is the time taken for promise1 to resolve and T2 is the time taken for promise2 to resolve, the time complexity would be 0(T1 + T2) where T1 and T2 are the respective completion times for each promise. If promise1 and

Await the resolution of the second future and store the resulting value. result = await add two promises(asyncio.Future(), asyncio.Future()) print(result) # Expected output: 4

CompletableFuture<Integer> completableFuture2 = CompletableFuture.supplyAsync(() -> 2); CompletableFuture<Integer> sumFuture = addTwoFutures(completableFuture1, completableFuture2); sumFuture.thenAccept(System.out::println) // Expected output: 4 .exceptionally(ex -> { // In case of an exception, cope with it here System.out.println("An error occurred: " + ex.getMessage()); // To ensure that the application does not terminat before completing the future computation // Handle with care as it may cause the application to hang if futures don't complete

// Get the std::future<int> associated with the result promise std::future<int> result_future = result_promise.get_future(); auto compute = [](std::future<int>&& future1, std::future<int>&& future2, std::promise<int> result_promise) mutable { // Wait for the first future to be ready and get its value // Wait for the second future to be ready and get its value // Set the promise's value to the sum of the two obtained values

// Call addTwoFutures with the futures we got from the promises // Await the resolution of the first promise and store the resulting value.

The time complexity of the function addTwoPromises largely depends on how promise1 and promise2 resolve. However, since it is awaiting both promises one after the other, it will be at least as long as the longest time taken for either of the promises to

resolve. If the promises do not depend on each other and could resolve in parallel, this sequential waiting increases the total time