2776. Convert Callback Based Function to Promise Based Function

Medium

Leetcode Link

Problem Description

The given problem outlines the requirement to transform a conventional callback-based function into a promise-based function in JavaScript. A callback-based function typically takes a callback as its first argument, with subsequent arguments being the data or parameters it operates on. The callback function usually has two parameters: the first is the error (if any), and the second is the result of the operation.

The `promisify` function we need to write should accept a callback-based function `fn` and return a new function that, instead of using callbacks, returns a promise. This promise should resolve if the original function's operation is successful, passing the result as the resolution value, or reject if an error occurs, passing the error as the rejection reason.

For instance, if we have a function `sum` which adds two numbers but only if they're positive, it would call the callback with the sum if the numbers are positive or with an error if any of the numbers are negative. The objective is to create a promisified version of such a function, so instead of dealing with callbacks, we can work with the more modern promise structure that allows chaining and better error handling with `try-catch` blocks in `async` functions or with `.then().catch()` chains.

Intuition

The intuition behind converting a callback-based function to a promise-based one lies in understanding the behavior of callbacks and promises. Callbacks are functions passed into other functions as arguments to be executed later, often upon the completion of an asynchronous operation. Promises are objects that represent the eventual completion or failure of an asynchronous operation and its resulting value.

When "promisifying" a function, we are essentially encapsulating the callback mechanism within the promise's `resolve` (for success) and `reject` (for failure) methods. By doing so, we create a function that no longer requires a callback function as a parameter and instead returns a promise that can be handled using `.then()` for the resolved value or `.catch()` for the rejected reason.

The solution code uses TypeScript, which allows specifying types for better code reliability and maintainability. The provided `promisify` function returns an `async` function that, when called, returns a `Promise`. In the body of this function, the original function `fn` is called with a special wrapped callback that invokes `resolve` when the operation is successful or `reject` when an error is passed to the callback. The rest of the arguments are directly passed to the original function with the help of the spread operator, `...args`.

This approach decouples the original function's logic from the handling of asynchronous execution, providing a more flexible and modern pattern for managing asynchronous operations in JavaScript.

Solution Approach

The solution approach for the `promisify` function involves wrapping the callback-based function with a new function that returns a `Promise`. We're using a closure here, a powerful feature in JavaScript where an inner function has access to the variables of its enclosing scope, to retain access to the original function `fn` and its arguments.

Here are the steps the `promisify` function performs:

1. **Creating a New Function**: The `promisify` function returns a new `async` function, which when called, will execute the original callback-based function `fn`. As `async` functions always return a promise, it sets up a foundation for promisification.

2. **Returning a Promise**: Within the returned function, we construct a new `Promise` using the `new Promise` constructor. The promise constructor takes an executor function with two parameters: `resolve` and `reject`. These are functions that change the state of the promise: `resolve` will fulfill the promise, and `reject` will reject the promise.

3. **Executing the Original Function**: We then call the original function `fn` with a custom callback and the rest of the arguments. This custom callback adheres to the Node.js style of callbacks, where the first argument is an error, and the subsequent arguments represent successful response data.

4. **Handling the Callback Response**:
   - If the callback is called with an error as the second argument, we invoke the `reject` function, passing in the error. This changes the state of the promise to rejected and allows downstream error handling with `.catch()` or a `try-catch` block in an `async` function.
   - If the callback is called without an error, we pass the response data to the `resolve` method. This resolves the promise with the given value, which can be accessed through the `.then()` method on the returned promise.
5. **Spreading Arguments**: The returned function uses the spread operator `...args` to pass all provided arguments to the original function `fn`, after the callback. This spread operator ensures that however many arguments are provided, they are passed on correctly.

The TypeScript types `CallbackFn` and `Promisified` describe the shape of the functions being dealt with:

- `CallbackFn` represents the original callback-based function. It accepts a callback function as its first argument followed by any number of number arguments.
- `Promisified` represents the new function that returns a promise and takes the same number of number arguments as `CallbackFn` (minus the callback).

The algorithm does not use complex data structures but relies on higher-order functions, closures, and the JavaScript promise mechanism to achieve the desired functionality. The algorithm is rather straightforward and does not involve any changes to the original function's logic but simply wraps it to provide a different interface.

Example Walkthrough

Consider a simple callback-based function `calculateArea`, which accepts a callback, length, and width. It calculates the area of a rectangle if both length and width are positive numbers. Otherwise, it returns an error through the callback:

```javascript
function calculateArea(callback, length, width) {
    if (length <= 0 || width <= 0) {
        callback(new Error('Length and width must be positive numbers'), null);
    } else {
        callback(null, length * width);
    }
}
```

To demonstrate the solution approach, we will promisify this `calculateArea` function using the steps provided in the solution approach:

1. **Creating a New Function**: We would define `promisify` to return an `async` function that invokes `calculateArea`.

2. **Returning a Promise**: Within this new function, we encapsulate the logic of `calculateArea` by returning a new `Promise`.

3. **Executing the Original Function**: We call `calculateArea` with a custom callback that properly handles success or error using the promise's `resolve` and `reject` functions.

4. **Handling the Callback Response**: The custom callback uses the `error-first` callback pattern to determine how to settle the promise.

5. **Spreading Arguments**: Instead of pre-defined parameters, `calculateArea` takes a list of arguments passed by spread syntax `...args`.

Here is the promisified version of `calculateArea`:

```javascript
const promisify = (fn) => {
    return async (...args) => {
        return new Promise((resolve, reject) => {
            fn((error, result) => {
                if (error) {
                    reject(error);
                } else {
                    resolve(result);
                }
            }, ...args);
        });
    };
};

const promisifiedCalculateArea = promisify(calculateArea);

// Now we can use the `promisifiedCalculateArea` function with promises:
promisifiedCalculateArea(5, 3)
    .then(area => console.log(`Area: ${area}`))
    .catch(error => console.error(`Error: ${error.message}`));

// If we try calling it with non-positive numbers, it will handle the error:
promisifiedCalculateArea(-5, 3)
    .then(area => console.log(`Area: ${area}`))
    .catch(error => console.error(`Error: ${error.message}`));
```

In this example, calling `promisifiedCalculateArea(5, 3)` will output "Area: 15" because the dimensions are positive. However, `promisifiedCalculateArea(-5, 3)` will output "Error: Length and width must be positive numbers" because one of the dimensions is not valid.

This walk-through exemplifies how the `promisify` function can take an existing callback-based function and adapt it to a promise-based pattern without altering the original business logic.

Python Solution

```python
from typing import Callable, Any
import asyncio

# Define a type for the callback function.
CallbackFunction = Callable[..., Any]

# Define a type for the resulting promisified function, which returns an awaitable yielding a number.
PromisifiedFunction = Callable[..., Any]

def promisify(function_with_callback: CallbackFunction) -> PromisifiedFunction:
    """
    Converts a callback-based function into a promise-based one.

    Parameters:
    function_with_callback (CallbackFunction): The original function that uses a callback.

    Returns:
    PromisifiedFunction: A function that returns an awaitable resolving to the same value.
    """

    # Return a new function that accepts an arbitrary number of arguments.
    async def wrapper(*args: Any) -> Any:
        # The new function returns an awaitable.
        async def inner() -> Any:
            loop = asyncio.get_running_loop()

            # Invoke the original function with a callback function.
            result, error = await loop.run_in_executor(None, lambda: function_with_callback(*args))
            # If the callback is called with an error, raise an exception.
            if error:
                raise Exception(error)
            # If the callback does not provide an error, return the result.
            return result

        return await inner()

    return wrapper

# Example Usage:
# Define a function to be promisified.
def example_function(callback, *args):
    # Immediately call the callback function with a result value of 42.
    callback(42, None)

# Promisify the example function.
async_function = promisify(example_function)

# This is an example of how you might call the promisified function in an async context.
async def main():
    # Call the promisified function and print the result.
    result = await async_function()
    print(result)  # Expected output: 42

# To actually run the example, you would start the asyncio event loop like this:
# asyncio.run(main())

```

Java Solution

```java
import java.util.concurrent.CompletableFuture;
import java.util.function.BiConsumer;

// Define a functional interface for the callback which will be used inside the method to be promisified.
@FunctionalInterface
interface CallbackFunction {
    void apply(BiConsumer<Integer, String> callback, int... args);
}

// The promisified method will return a CompletableFuture that can contain the result of type Integer.
@FunctionalInterface
interface PromisifiedFunction {
    CompletableFuture<Integer> apply(int... args);
}

public class PromisifyExample {

    /**
     * Converts a callback-based method into a promise-based (CompletableFuture) one.
     *
     * @param functionWithCallback The original method that accepts a callback.
     * @return A method that returns a CompletableFuture resolving to the same value.
     */
    public static PromisifiedFunction promisify(CallbackFunction functionWithCallback) {
        // Return a new method that accepts an arbitrary number of integer arguments.
        return args -> {
            // The new method returns a CompletableFuture.
            CompletableFuture<Integer> future = new CompletableFuture<>();
            // Invoke the original method with a BiConsumer as the callback function.
            functionWithCallback.apply((data, error) -> {
                // If the callback is called with an error, complete the future exceptionally.
                if (error != null) {
                    future.completeExceptionally(new RuntimeException(error));
                } else {
                    // If there is no error, successfully complete the future with the data.
                    future.complete(data);
                }
            }, args); // Pass the arguments to the original method.

            return future;
        };
    }

    // Example Usage:
    public static void main(String[] args) {
        // Promisify a method that uses a callback, which will immediately call the callback with a value of 42.
        PromisifiedFunction asyncFunction = promisify((callback, arguments) -> callback.accept(42, null));
        // Invoke the promisified method and log the result to the console.
        asyncFunction.apply().thenAccept(System.out::println); // Expected output should be 42
    }
}

```

C++ Solution

```cpp
#include <iostream>
#include <functional>
#include <future>
#include <exception>

// Define a type for the callback function which will be wrapped by Promisify.
using CallbackFunction = std::function<void(std::function<void(int, std::string)>, int)>;

// Define a type for the resulting 'promisified' function, which returns a future of an int.
using PromisifiedFunction = std::function<std::future<int>(int)>;

/**
 * Converts a callback-based function into a future-based one.
 *
 * @param function_with_callback The original function that uses a callback.
 * @returns A function that returns a future resolving to the same value.
 */
PromisifiedFunction Promisify(CallbackFunction function_with_callback) {
    // Return a new function that accepts an integer.
    return [function_with_callback](int arg) -> std::future<int> {
        // The new function returns a future.
        return std::async(std::launch::async, [function_with_callback, arg]() -> int {
            // Create a promise to hold the result.
            std::promise<int> promise;
            std::future<int> result = promise.get_future();

            // Invoke the original function with a callback function.
            function_with_callback(
                [&promise](int data, std::string error) {
                    // If the callback is called with an error, set the exception in the promise.
                    if (!error.empty()) {
                        promise.set_exception(std::make_exception_ptr(std::runtime_error(error)));
                    } else {
                        // If the callback does not provide an error, set the data in the promise.
                        promise.set_value(data);
                    }
                },
                arg
            );

            // Wait for the result to become available and return it.
            return result.get();
        });
    };
}

// Example usage:
int main() {
    // Promisify a function that takes a callback, which will immediately call the callback with a value of 42.
    auto async_function = Promisify([](auto callback, int) { callback(42, ""); });

    // Invoke the promisified function and log the result to the console.
    try {
        std::future<int> future_result = async_function(0);
        std::cout << "Result: " << future_result.get() << std::endl; // Expected output: 42
    } catch (const std::exception& ex) {
        std::cerr << "Error: " << ex.what() << std::endl;
    }

    return 0;
}

```

Typescript Solution

```typescript
// Define a type for the callback function which will be wrapped by promisify.
type CallbackFunction = (next: (data: number, error?: string) => void, ...args: number[]) => void;

// Define a type for the resulting promisified function, which returns a promise of a number.
type PromisifiedFunction = (...args: number[]) => Promise<number>;

/**
 * Converts a callback-based function into a promised-based one.
 *
 * @param functionWithCallback The original function that uses a callback.
 * @returns A function that returns a promise resolving to the same value.
 */
function promisify(functionWithCallback: CallbackFunction): PromisifiedFunction {
    // Return a new function that accepts an arbitrary number of numeric arguments.
    return async function(...args: number[]): Promise<number> {
        // The new function returns a promise.
        return new Promise((resolve, reject) => {
            // Invoke the original function with a callback function.
            functionWithCallback((data, error) => {
                // If the callback is called with an error, reject the promise.
                if (error) {
                    reject(error);
                } else {
                    // If the callback does not provide an error, resolve the promise.
                    resolve(data);
                }
            }, ...args); // Spread the arguments into the original function.
        });
    };
}

// Example Usage:
// Promisify a function that takes a callback, which will immediately call the callback with a value of 42.
// const asyncFunction = promisify((callback) => callback(42));
// Invoke the promisified function and log the result to the console.
// asyncFunction().then(console.log); // Expected output: 42

```

Time and Space Complexity

The time complexity of the `promisify` function itself is `O(1)`, meaning it runs in constant time. This is because the `promisify` function merely constructs and returns a new function without performing any computation that depends on the size of the input.

The provided `promisified` function will have the time complexity of the original `CallbackFn` since it is essentially wrapping the existing callback function with promise handling logic. Hence, the time complexity of the `promisified` function will be the same as the time complexity of the `CallbackFn` it wraps. If the time complexity of `CallbackFn` is `O(f(n))`, where `f(n)` is a function that describes how the execution time scales with the input size `n`, then the time complexity of the `promisified` function is also `O(f(n))`.

The space complexity of the `promisify` function is also `O(1)`, or constant space complexity, as it doesn't allocate any additional space that depends on the input size. It just returns a new function object.

The space complexity of the returned `promisified` function depends on the implementation of the original `CallbackFn`. However, the use of a promise introduces additional space overhead. This space is for the closure that includes the original `CallbackFn`, the `resolve` and `reject` functions, and any arguments passed to the `promisified` function. In general, if the space complexity of `CallbackFn` is `O(g(n))`, then the space complexity of the created `promisified` function would be `O(g(n))`, not including the space used by the Promise internals, which is generally a constant overhead.