2721. Execute Asynchronous Functions in Parallel

Medium

Leetcode Link

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:

- Resolve when all of the promises returned from the functions in `functions` array have been resolved. The resolved value should be an array containing the resolved values from each promise, maintaining the same order as the original `functions` array.
- Reject if any 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 relying on `Promise.all`, which simplifies the coordination of asynchronous promises.

Intuition

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 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 have a rejection handler for each promise.
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(...)`) and performs the above-mentioned steps to resolve or reject based on the results of the promise executions.

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 execution and coordination of all promises returned by the `functions`.
2. We declare a counter variable `cnt` initialized to zero, which we'll use to keep track of how many promises have been settled (resolved).
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 we maintain the order of resolved values as per their corresponding functions.
4. We loop over the array of functions using `for (let i = 0; i < functions.length; ++i)` and call each function, which returns a promise.
5. For each returned promise:
   - 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 results.
   - We also attach a `.catch(err => {})` handler to handle any rejections. If a promise rejects, we call `reject(err)` immediately, forwarding the first error we encounter to the `promiseAll` function's promise, thus rejecting it.

This implementation makes use of closures to keep track of the index and result of each asynchronous operation, arrays to store the results, and promises' `.then()` and `.catch()` methods for asynchronous flow control.

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

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

```
const ans = new Array(functions.length);
for (let i = 0; i < functions.length; ++i) {
    functions[i]()
        .then(res => {
            ans[i] = res;
            ++cnt;
            if (cnt === functions.length) {
                resolve(ans);
            }
        })
        .catch(reject);
}
```

This approach effectively simulates the behavior of `Promise.all` without directly relying on it, demonstrating a fundamental understanding of concurrency and error handling with promises in JavaScript.

Example Walkthrough

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

- `function1` simulates an asynchronous operation that resolves after 1 second with the string `"Result 1"`.
- `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"`.

Using the given solution approach, we can execute these functions and manage their promises manually. Here's how the process would look step-by-step:

1. **Initialization:**
   - We define a new `Promise` object `promiseAll` that will manage the execution of all functions.
   - Inside this new `Promise`, we initialize a counter `cnt` to zero and an array `ans` of the same length as the `functions` array to store the resolved values.
2. **Execution and Management:**
   - We begin a for loop to execute each asynchronous function in the `functions` array.
   - For each function, we invoke it and receive a promise back.
3. **Resolution Handling:**
   - We attach a `.then(res => {})` handler to each promise. When `function1()` resolves after 1 second, the `.then()` handler for that function gets executed, 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, storing the result at the corresponding index in the `ans` array and incrementing `cnt`.
4. **Completion Check:**
   - 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 promise.
   - At this point, the `promiseAll` promise will successfully resolve with the array `["Result 1", "Result 2", "Result 3"]`.
5. **Error Handling:**
   - 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 rejection that occurred, and the process would stop without awaiting the resolution of other promises.

This example demonstrates how the provided solution approach coordinates multiple asynchronous operations and processes their results or errors as they occur, achieving the same behavior as `Promise.all` without using it directly.

Python Solution

```
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]]:
    """
    Executes an array of functions that each returns an asyncio coroutine (future),
    and resolves when all have successfully completed.
    If any of the coroutines rejects, this function will reject with the reason from
    the first coroutine that rejected.

    :param promise_fns: A list of functions that each returns a coroutine.
    :return: A coroutine that resolves with a list of the resolved values from the coroutines or rejects.
    """
    results_array = []

    for promise_fn in promise_fns:
        # We will collect all coroutine objects first to later await them concurrently
        results_array.append(promise_fn())

    # Now we await for all gathered coroutine objects concurrently
    return await asyncio.gather(*results_array)

# Example usage:
# async def main():
#     promise = lambda: asyncio.Future().set_result(42) # Creating a list of lambda functions returning coroutines
#     try:
#         results = await promise_all([promise])
#         print(results)  # Expected output: [42]
#     except Exception as e:
#         print("At least one coroutine rejected:", e)

# asyncio.run(main())
```

Java Solution

```
import java.util.List;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;
import java.util.function.Supplier;
import java.util.stream.Collectors;
import java.util.stream.IntStream;

public class PromiseAll {

    /**
     * Executes a list of suppliers that return CompletableFutures, and completes when all have successfully completed.
     * If any of the CompletableFutures complete exceptionally, this function will complete exceptionally
     * with the cause from the first future that completed exceptionally.
     *
     * @param <T> The type of the results that the futures are expected to return.
     * @param <R> The result type returned by this CompletableFuture's suppliers.
     * @return A CompletableFuture that, when completed normally, contains a list of the values obtained by
     *         calling each of the suppliers. This future will complete exceptionally if any of the provided
     *         CompletableFutures complete exceptionally.
     */
    public static <T, R> CompletableFuture<List<R>> promiseAll(List<Supplier<CompletableFuture<T>>> promiseFns) {
        List<CompletableFuture<R>> futuresList = promiseFns.stream()
                                                           .map(Supplier::get)
                                                           .collect(Collectors.toList());

        CompletableFuture<Void> allOfFuture =
                CompletableFuture.allOf(futuresList.toArray(new CompletableFuture[0]));

        return allOfFuture.thenApply(v ->
                futuresList.stream()
                           .map(CompletableFuture::join)
                           .collect(Collectors.toList())
        );
    }

    public static void main(String[] args) {
        // Example usage:
        List<Supplier<CompletableFuture<Integer>>> suppliers = List.of(
                () -> CompletableFuture.completedFuture(42)
        );

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

        try {
            List<Integer> results = allFutures.get(); // This blocks until the future completes
            System.out.println(results); // Expected output: [42]
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace(); // Handle the exception if any of the futures completed exceptionally
        }
    }
}
```

C++ Solution

```
#include <vector>
#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) {}
};

/**
 * 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.
 *
 * @param T The type of the results that the futures are expected to return.
 * @param futuresFns 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>()>> futuresFns) {
    // 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
    futures.reserve(futuresFns.size());

    // Execute each function to get the future and add it to the vector
    for (auto& fn : futuresFns) {
        futures.push_back(fn());
    }

    // Vector to accumulate the results from the futures
    std::vector<T> results;
    results.reserve(futures.size());

    // Iterate through the futures and get their results
    for (auto& fut : futures) {
        try {
            // Get the result from the future and add it to the results vector
            results.push_back(fut.get());
        } catch (...) {
            // If getting the result throws, forward the exception
            throw AsyncOperationFailed("One of the asynchronous operations failed.");
        }
    }

    // Return the full vector of results when all futures have been processed
    return results;
}

// Example usage (within a function that allows exceptions):
try {
    auto results = promiseAll<int>({
        [](){ return std::async(std::launch::async, [](){ return 42; }); }
    });

    // Process results here
} catch (const AsyncOperationFailed& e) {
    // Handle the failure case here
}
```

Typescript Solution

```
/**
 * Executes an array of functions that returns Promises, and resolves when all have successfully completed.
 * If any of the Promises rejects, this function will reject with the reason from the first promise that rejected.
 *
 * @param {(() => Promise<T>)[]} promiseFns - An array of functions that each returns a promise.
 * @returns {Promise<T[]>} A promise that resolves with an array of the resolved values from the input promises or rejects.
 */
async function promiseAll<T>(promiseFns: (() => Promise<T>)[]): Promise<T[]> {
    return new Promise<T[]>((resolve, reject) => {
        // Counter for tracking how many promises have been resolved
        let cnt = 0;
        // Array to accumulate the resolved results. Its length is pre-set to match the array of promises
        const results = new Array<T>(promiseFns.length);

        // Iterate over the promise functions
        promiseFns.forEach((promiseFn, index) => {
            // Execute each promise function
            promiseFn()
                .then(value => {
                    // Save the resolved value in the array
                    results[index] = value;
                    // Increment the counter
                    cnt += 1;
                    // If all promises have resolved, resolve the outer promise with the results array
                    if (cnt === functions.length) {
                        resolve(results as T[]);
                    }
                })
                .catch(error => {
                    // If any of the promises reject, reject the outer promise with the error
                    reject(error);
                });
        });
    });
}

// Example usage:
// const promise = promiseAll([() => Promise.resolve(42)]);
// promise.then(console.log).catch(console.error); // Expected output: [42]
```

Time and Space Complexity

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 Complexity

For time complexity, the `promiseAll` function operates by initiating all the promises nearly simultaneously (though there is a 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 $O(m)$ time, then the `promiseAll` function would also have a time complexity of $O(m)$. However, if we consider the loop's contribution, since there is a one-time initialization for each promise, this is effectively $O(1)$ for each promise being $O(m)$ for n 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 still be dominated by the promise that takes the longest time. Therefore,

- **Time Complexity:** $O(m)$

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

Space Complexity

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

- **Space Complexity:** $O(n)$

where n is the number of promises. We do not consider the space taken by the input itself or the promises' internal space usage; we 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 $O(m)$ 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.