2637. Promise Time Limit

Medium

Leetcode Link

Problem Description

The task is to create a wrapper function that controls how long an asynchronous function is allowed to run before it's considered to have taken too long. An asynchronous function, which we'll call `fn`, is given to us along with a timeout threshold `t` in milliseconds. We need to return a new function that behaves in the following way:

- When called, it lets `fn` run with any arguments passed to it.
- If `fn` finishes its task within `t` milliseconds, the new function should complete with the same result as `fn`.
- If `fn` does not finish within the allotted `t` milliseconds, the new function should stop waiting for `fn` and instead return a rejection with the message "Time Limit Exceeded".

This problem combines asynchronous programming with timing control, requiring knowledge of promises and the race condition in asynchronous flows.

Intuition

The intuition behind the solution is to use concurrency in JavaScript Promise operations. We can race two promises against each other: one promise represents the completion of the input function `fn`, and the other represents the time limit as a timeout. Here's the thinking process:

1. Start by invoking the `fn` with its arguments, wrapped in a promise.
2. Create a timeout promise that will reject with "Time Limit Exceeded" after `t` milliseconds.
3. Use `Promise.race` to run both promises (the function promise and the timeout promise) against each other.
4. If `fn` completes first, the race is won by the function promise, and its result is returned.
5. If `fn` takes too long and the timeout elapses first, the race is won by the timeout promise, and "Time Limit Exceeded" is returned.

This approach ensures that regardless of what `fn` is doing, we're not waiting for it longer than `t` milliseconds, enforcing a strict time limit on its execution.

Solution Approach

The solution makes use of Promises and the `race` method provided by the Promise API. The idea is to have two promises: one that represents the asynchronous operation `fn` and another that acts as a timer. Whichever promise settles first determines the outcome. To implement this:

1. We define a function `timeLimit` that takes an asynchronous function `fn` and a timeout value `t`.
2. `timeLimit` returns a new function that accepts any number of arguments (`...args`).
3. Inside this new function, we set up a race condition between two promises:
   - The first promise is the result of calling `fn(...args)`. Since `fn` is asynchronous and returns a promise, it will either resolve with the result of `fn` or reject if `fn` fails.
   - The second promise is created with a call to `new Promise((_, reject) => setTimeout(() => reject('Time Limit Exceeded'), t))`. This promise waits for `t` milliseconds and then rejects with "Time Limit Exceeded".
4. We use `Promise.race` to run these two promises. This method returns a new promise that resolves or rejects as soon as one of the promises in the array resolves or rejects, with the value or reason from that promise.
5. When calling the race, if `fn` completes before the timeout, the resulting promise will resolve with the value provided by `fn`.
6. If `fn` does not complete within `t` milliseconds, the timer promise will reject first, causing the race to end with a rejection.

By using `Promise.race`, we ensure that our wrapped function never waits longer than `t` milliseconds to settle. This effectively creates a timeout behavior for any asynchronous operation.

Here's a detailed handling scenario:

- Assume the function call `fn(...args)` finishes in less time than `t`. In this case, `Promise.race` will resolve to whatever `fn(...args)` resolves to.
- On the other hand, if `fn(...args)` takes longer than `t` milliseconds, the promise from `setTimeout` will reject first, causing `Promise.race` to reject with "Time Limit Exceeded".

This pattern is a common solution in scenarios where a timeout needs to be enforced on asynchronous operations, ensuring that a system remains responsive and does not wait indefinitely for an action to complete.

Example Walkthrough

Let's say we have an asynchronous function `asyncOperation` that resolves after a random amount of time, which sometimes might exceed our threshold. We want to ensure that if `asyncOperation` takes more than 2000 milliseconds (2 seconds), it should be considered as failed due to taking too long. We will use the `timeLimit` function to enforce this rule.

```javascript
// Simulate an asynchronous operation that takes a random amount of time to complete,
// sometimes more than 2000 milliseconds.
function asyncOperation(resolveValue) {
  return new Promise((resolve) => {
    const delay = Math.floor(Math.random() * 3000); // Random delay from 0 to 2999 ms
    setTimeout(() => resolve(resolveValue), delay);
  });
}

// The `timeLimit` function wraps around our `asyncOperation`.
const timeLimit = (fn, t) => (...args) =>
  Promise.race([
    fn(...args),
    new Promise((_, reject) => setTimeout(() => reject('Time Limit Exceeded'), t))
  ]);

// Let's create a time-limited version of our `asyncOperation` with a 2000 ms limit.
const limitedAsyncOperation = timeLimit(asyncOperation, 2000);

// Call the time-limited asynchronous function with a sample resolve value.
limitedAsyncOperation('Sample resolve value')
  .then(result => console.log(`Operation successful: ${result}`))
  .catch(error => console.log(`Operation failed: ${error}`));
```

In this walkthrough:

1. We define an asynchronous function `asyncOperation` that would typically represent a more complex async process, such as an API call or a database transaction.
2. We set up a `timeLimit` function according to the solution approach.
3. We create `limitedAsyncOperation` by passing `asyncOperation` and the desired timeout of 2000 milliseconds to `timeLimit`.
4. When we call `limitedAsyncOperation`, it initiates two parallel promises:
   - The original `asyncOperation` promise that resolves after a random delay.
   - A new promise that will reject with "Time Limit Exceeded" after 2000 milliseconds.
5. `Promise.race` is used to return the outcome of whichever promise settles first.
   - If `asyncOperation` completes in less than 2 seconds, its resolve value will be logged to the console.
   - If `asyncOperation` takes more than 2 seconds, the console will log "Operation failed: Time Limit Exceeded".

The above code demonstrates how the `timeLimit` function effectively imposes a timeout constraint on an asynchronous operation.

Python Solution

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

# Define a generic asynchronous function type that returns a Future.
AsyncFunction = Callable[..., Any]

async def time_limit(async_function: AsyncFunction, time_limit_millis: int) -> AsyncFunction:
    """
    Wraps an asynchronous function with a time limit, enforcing it to either
    resolve or reject within a set timeframe.

    :param async_function: The asynchronous function to wrap.
    :param time_limit_millis: The maximum amount of time (in milliseconds) to wait before cancelling.
    :returns: A function that behaves like the original async function but with a time limit.
    """
    async def wrapper(*args, **kwargs):
        # Use asyncio.wait_for to apply a timeout to the async function
        try:
            return await asyncio.wait_for(async_function(*args, **kwargs), time_limit_millis / 1000.0)
        except asyncio.TimeoutError:
            # Raise a custom exception if the function times out
            raise TimeoutError('Time Limit Exceeded')

    return wrapper

# Usage example:
# Define an asynchronous operation that may take longer than the allocated time limit.
async def example_async_function(duration):
    await asyncio.sleep(duration)

# Wrap the async function with a time limit.
limited_function = time_limit(example_async_function, 100)

# Use the wrapped function with a time-out that exceeds the time limit and handle exceptions.
async def main():
    try:
        # This should raise a TimeoutError after 100ms
        await limited_function(0.150)
    except TimeoutError as e:
        print(e)

# Run the main function to demonstrate usage.
if __name__ == '__main__':
    asyncio.run(main())

```

Java Solution

```java
import java.util.concurrent.*;
import java.util.function.*;

/**
 * Represents a generic function that returns a Future.
 */
@FunctionalInterface
interface AsyncFunction<T> {
    Future<T> apply(Object... params);
}

/**
 * Wraps an asynchronous function with a time limit, enforcing it to either
 * complete or cancel within a set timeframe.
 *
 * @param asyncFunction The asynchronous function to wrap.
 * @param timeLimitMillis The maximum amount of time (in milliseconds) to wait before cancelling.
 * @param <T> The type of the result provided by the asynchronous function.
 * @return A function that behaves like the original async function but with a time limit.
 */
public static <T> AsyncFunction<T> timeLimit(AsyncFunction<T> asyncFunction, long timeLimitMillis) {
    // Return a new function that upon invocation, submits the original task to an executor
    // and applies the timeout.
    return (Object... args) -> {
        // Create a new executor to run the asynchronous function
        ExecutorService executor = Executors.newSingleThreadExecutor();

        // Submit the original asynchronous function as a callable task to the executor
        Callable<T> task = () -> {
            try {
                return asyncFunction.apply(args).get();
            } catch (ExecutionException | InterruptedException e) {
                throw new RuntimeException(e);
            }
        };

        Future<T> future = executor.submit(task);

        // Schedule a task to cancel the future after the time limit
        Executors.newSingleThreadScheduledExecutor()
                .schedule(() -> future.cancel(true), timeLimitMillis, TimeUnit.MILLISECONDS);

        try {
            // Return the result of the future, awaiting termination with the given time limit
            return CompletableFuture.completedFuture(future.get(timeLimitMillis, TimeUnit.MILLISECONDS));
        } catch (TimeoutException | InterruptedException | ExecutionException e) {
            // Cancel the future if it times out or encounters an issue
            future.cancel(true);
            throw new RuntimeException("Time Limit Exceeded", e);
        } finally {
            // Shutdown the executor service to prevent lingering threads
            executor.shutdown();
        }
    };
}

// Usage example (uncomment to use within a main method or other appropriate context):
/*
AsyncFunction<Void> limited = timeLimit(duration -> {
    CompletableFuture<Void> future = new CompletableFuture<>();
    new Thread(() -> {
        try {
            Thread.sleep((Long) duration);
            future.complete(null); // Resolve the future upon successful completion
        } catch (InterruptedException e) {
            future.completeExceptionally(e);
        }
    }).start();
    return future;
}, 100);

// Call the wrapped function with a duration that exceeds the limit, catching any exceptions
try {
    limited.apply(150).get();
} catch (Exception e) {
    System.out.println(e.getMessage());  // Expected output: "Time Limit Exceeded"
}
 */

```

C++ Solution

```cpp
#include <iostream> // For std::cout and std::endl
#include <future>   // For std::async, std::future, and std::chrono
#include <functional> // For std::function
#include <stdexcept> // For std::runtime_error
#include <chrono> // For std::chrono
#include <thread> // For std::this_thread::sleep_for

// Define a type alias for a generic function that returns a std::future.
using AsyncFunction = std::function<std::future<void>(std::vector<int>)>;

/**
 * Wraps an asynchronous function with a time limit, enforcing it to either
 * resolve or fail within a set timeframe.
 *
 * @param asyncFunction The asynchronous function to wrap.
 * @param timeLimitMillis The maximum amount of time (in milliseconds) to wait before canceling.
 * @return A function that behaves like the original async function but with a time limit.
 */
auto timeLimit(AsyncFunction asyncFunction, int timeLimitMillis) -> AsyncFunction {
    // Return a new function which will race the original function against a timeout
    return [asyncFunction, timeLimitMillis](std::vector<int> args) -> std::future<void> {
        // Start the async function with the given arguments
        auto result = asyncFunction(args);

        // Wait for the result for the given time limit
        if (result.wait_for(std::chrono::milliseconds(timeLimitMillis)) == std::future_status::timeout) {
            // If it times out, throw a runtime_error
            throw std::runtime_error("Time Limit Exceeded");
        }

        // Otherwise, return the original function's result
        return result;
    };
}

// Usage example:
// The following lines of code provide a basic example of how the timeLimit function
// could be used in practice. It defines an asynchronous operation that could take
// longer than the specified time limit and handles timeout errors.

// Define the actual async operation function
AsyncFunction myAsyncOperation = [](std::vector<int> duration) -> std::future<void> {
    return std::async(std::launch::async, [duration]() {
        // Simulate a long running operation
        std::this_thread::sleep_for(std::chrono::milliseconds(duration[0]));
        std::cout << "Operation finished" << std::endl;
    });
};

// Use the timeLimit wrapper with the async operation with a timeout of 100ms
AsyncFunction limitedAsyncOperation = timeLimit(myAsyncOperation, 100);

// Run and catch any timeout errors
try {
    limitedAsyncOperation({150}).get(); // Using 150ms for the operation
} catch (std::runtime_error &e) {
    // Print out the error message if there's a timeout
    std::cout << e.what() << std::endl; // Expected output: "Time Limit Exceeded" after t=100ms
}

```

Typescript Solution

```typescript
// Define a generic function type that returns a Promise.
type AsyncFunction = (...params: any[]) => Promise<any>;

/**
 * Wraps an asynchronous function with a time limit, enforcing it to either
 * resolve or reject within a set timeframe.
 *
 * @param asyncFunction The asynchronous function to wrap.
 * @param timeLimitMillis The maximum amount of time (in milliseconds) to wait before rejecting.
 * @returns A function that behaves like the original async function but with a time limit.
 */
function timeLimit(asyncFunction: AsyncFunction, timeLimitMillis: number): AsyncFunction {
    // Return a new function which will race the original function against a timeout
    return async function (...args) {
        // Use Promise.race to compete the async function call against a timeout
        return Promise.race([
            asyncFunction(...args),
            // Create a new Promise that automatically rejects after timeLimitMillis
            new Promise((_, reject) => setTimeout(() => reject(new Error('Time Limit Exceeded')), timeLimitMillis)),
        ]);
    };
}

// Usage example:
// Define an asynchronous operation that may take longer than the allocated time limit.
// const limited = timeLimit((duration) => new Promise(resolve => setTimeout(resolve, duration)), 100);
// Use the wrapped operation with a timeout that exceeds the time limit and catch any errors.
// limited(150).catch(error => console.log(error.message)); // Expected output: "Time Limit Exceeded" at t=100ms

```

Time and Space Complexity

The code defines a function `timeLimit` that takes another function `fn` and a time limit `t`, and returns a new function that will reject the promise if it doesn't resolve within time `t`. The computational complexities are as follow:

Time Complexity

The time complexity of the `timeLimit` function itself is `O(1)` (constant time), as it simply sets up a `Promise.race()` construct without any loops or recursive calls.

However, the time complexity of the resulting function when called is determined by `fn`, which is an input parameter to `timeLimit`. Since `fn` could be any function, its complexity can vary. When this resulting function is called, it will execute `fn(...args)` and `setTimeout(..., t)` concurrently, and the `Promise.race()` will settle as soon as the first promise settles.

Thus, the overall time complexity of the resulting function is `O(f(n))` where `O(f(n))` represents the time complexity of the function `fn` that is passed to `timeLimit`.

Space Complexity

The space complexity of the `timeLimit` function is `O(1)`. It does not utilize any additional space that grows with the input size, so it uses constant space.

The space complexity of the resulting function when it is called with a specific `fn` is determined by the space that `fn` uses. If `fn` uses space that grows with the input, then the resulting function will also have a space complexity that reflects that growth. However, since we do not have specifics on what `fn` does, we denote the space complexity of the function as `O(g(n))`, where `O(g(n))` represents the space complexity of `fn`.

In addition to the space used by `fn`, the resulting function uses space for the `Promise.race()` and the `setTimeout`. However, this additional space does not grow with the input and is thus considered constant, not affecting the overall space complexity which remains `O(g(n))`.