2805. Custom Interval

Medium

Leetcode Link

Problem Description

The problem `customInterval` involves creating a scheduler that executes a function `fn` repeatedly with intervals that increase linearly over time. Each execution interval is determined by the formula `delay + period * count`, where `delay` is the initial delay before the first execution of `fn`, `period` is the increment factor which is added to the total delay for each subsequent execution, and `count` is a counter that keeps track of how many times `fn` has been executed. The function `customInterval` must return an identifier `id` that can be used to stop the scheduled executions.

`customClearInterval` is the function used to stop the execution of `fn`. It takes an `id` as input, which corresponds to the identifier returned by `customInterval`, and stops the scheduled execution that was associated with that `id`.

Intuition

The `customInterval` function needs to schedule and execute the given function `fn` at the incrementally increasing intervals. The approach to this is to set up a recursive timeout system, where after each execution of `fn`, a new timeout is scheduled with the updated interval based on the linear pattern. This ensures that `fn` is executed repeatedly at the correct intervals.

To facilitate starting the timeout, we define a `recursiveTimeout` function inside `customInterval`. This helper function uses JavaScript's `setTimeout` to schedule `fn`'s next execution, increases the `count` by 1 after each execution, and then calls itself to schedule the next execution.

An `intervalMap` is used to keep track of all scheduled timeouts using a unique identifier `id` which in this case is generated by `Date.now()`. This ensures each scheduled set of executions has a unique identifier, which is necessary for the `customClearInterval` function to properly identify and stop the correct execution.

The `customClearInterval` function's intuition is to provide a way to cancel the scheduled execution. It uses the unique identifier `id` to look up and clear the scheduled timeout from `intervalMap`. This prevents future calls of `fn` and also removes the entry from the map to avoid potential memory leaks from keeping unnecessary timeouts in the map.

Solution Approach

The solution to the problem uses a combination of a map for storing timeout references, recursion for scheduling future calls, and the `setTimeout` function for executing the provided function `fn` at increasing intervals.

Here's how the implementation breaks down step by step:

1. **Global Map for Timeout References (`intervalMap`)**: We initialize `intervalMap`, a JavaScript `Map` object, to store and retrieve the timeout references using a unique number `id`. This `id` is used later to clear the timeout if needed.
2. **Custom Interval Scheduling (`customInterval`)**:
    - We first initialize a local `count` variable to 0 inside `customInterval`. This variable tracks the number of times `fn` has been called.
    - We declare `recursiveTimeout`, a recursive function that will handle the scheduling of `fn` executions. Inside this function, the `setTimeout` method is called with a delay calculated using the formula `delay + period * count`. Each time `fn` is executed, `count` is incremented.
    - We use `Date.now()` to generate a unique `id` for the current scheduling of `fn`. `Date.now()` returns the current timestamp in milliseconds, so it has a high probability of being unique.
    - We store the result of `setTimeout` in the `intervalMap`. The key is the `id`, and the value is the timeout reference returned by `setTimeout`. This allows us to clear the specific timeout later.
    - We immediately call `recursiveTimeout` to start the interval process.
3. **Custom Interval Clearing (`customClearInterval`)**:
    - This function accepts an `id` and checks if this `id` exists in the `intervalMap`.
    - If it exists, `clearTimeout` is called with the associated timeout reference, effectively stopping further executions of `fn`.
    - The `id` and its corresponding timeout reference are removed from `intervalMap` to free up memory and keep the map clean.

Recursion is a key pattern used in the solution for scheduling future calls. Each time `fn` executes, `recursiveTimeout` schedules the next execution. Instead of a standard interval where the delay between calls is constant, this pattern allows for the dynamic calculation of the delay based on the execution count, providing a linearly increasing interval between the calls.

Data structure usage (`Map`) allows us to efficiently keep track of and manage the different timeouts, providing a clean way to cancel them when necessary.

Example Walkthrough

Let's illustrate the solution approach with an example:

Suppose we want to schedule a function `fn` that simply logs "Hello, World!" to the console. We want the first execution to happen after an initial delay of `1000ms` (1 second), and then each subsequent execution to be delayed by an additional `500ms`.

We can declare `fn` as follows:

```javascript
function fn() {
  console.log("Hello, World!");
}
```

Now let's use the `customInterval` with `delay` of `1000ms` and `period` of `500ms`:

```javascript
const id = customInterval(fn, 1000, 500);
```

Here's a step-by-step explanation of what happens next:

1. A unique `id` for this interval is generated. Let's assume the `id` returned by `Date.now()` is `1623249048531`.

2. The `intervalMap` will hold this `id` as a key with the initial timeout reference as its value.

3. The `fn` function is scheduled to execute after `1000ms` for the first time, as `count` is `0`.

4. After `1000ms`, `fn` executes and logs "Hello, World!" to the console for the first time. The `count` is then incremented to `1`.

5. The `recursiveTimeout` function inside `customInterval` schedules the next execution of `fn` using a new timeout. The new delay is calculated as `1000 + (500 * 1) = 1500ms`.

6. After `1500ms`, `fn` executes for the second time, logging "Hello, World!" again, and `count` increases to `2`.

7. This process continues with the delay increasing by `500ms` after each execution.

If we want to stop this pattern, we can call `customClearInterval` and pass our `id` (`1623249048531`) as an argument:

```javascript
customClearInterval(id);
```

Upon calling this function:

1. The `customClearInterval` function looks for the `id` in `intervalMap`.

2. If it finds the `id`, it calls `clearTimeout`, using the timeout reference, which stops future scheduled executions of `fn`.

3. The `id` is removed from `intervalMap`, cleaning up the reference and ensuring no memory leaks.

This example shows how the `customInterval` and `customClearInterval` functions manage the scheduled execution of `fn` with increasing intervals and how we can stop the schedule whenever needed.

Python Solution

```python
from threading import Timer
from time import time

# Dictionary to keep track of custom intervals
interval_map = {}


def custom_interval(fn, delay, period):
    """
    Creates a custom interval that mimics setInterval but allows for increasing delay durations.

    :param fn: The function to execute after each delay period.
    :param delay: The initial delay before the function execution in milliseconds.
    :param period: The additional delay added after each execution in milliseconds.
    :return: An ID that can be used to clear the interval.
    """
    def recursive_timeout(count):
        """
        Helper function that recursively sets timers.

        :param count: The number of times the function has been called.
        """
        # Calculate new delay with added period for each interval
        current_delay = delay + period * count

        # Schedule the function to be called after the calculated delay
        timer = Timer(current_delay / 1000, lambda: recursive_timeout(count + 1))
        timer.start()  # Start the timer

        fn()  # Execute the provided function

        # Store the timer reference using the interval id
        interval_map[interval_id] = timer

    # Generate a unique ID for this interval
    interval_id = int(time() * 1000)
    recursive_timeout(0)  # Start the interval with count 0
    return interval_id  # Return the unique ID


def custom_clear_interval(interval_id):
    """
    Clears a custom interval by ID.

    :param interval_id: The ID of the interval to clear.
    """
    # Check if the interval ID exists in the dictionary
    if interval_id in interval_map:
        # Get the timer and cancel it
        interval_map[interval_id].cancel()

        # Remove the interval ID from the dictionary
        del interval_map[interval_id]
```

Java Solution

```java
import java.util.concurrent.*;
import java.util.Map;
import java.util.HashMap;

public class CustomInterval {
    private Map<Long, ScheduledFuture<?>> intervalMap = new HashMap<>(); // Map to keep track of custom intervals
    private ScheduledExecutorService executorService = Executors.newScheduledThreadPool(10);
    // Assuming a maximum of 10 concurrent intervals to handle, can be adjusted as needed.

    /**
     * Creates a custom interval that mimics setInterval but allows for increasing delay durations.
     *
     * @param runnable The function (Runnable) to execute after each delay period.
     * @param delay The initial delay before the function execution in ms.
     * @param period The additional delay added after each execution in ms.
     * @return An ID that can be used to clear the interval.
     */
    public long customInterval(Runnable runnable, long delay, long period) {
        long[] count = new long[]{0}; // Counter to track the number of times the function has been called

        // Helper class that recursively schedules the runnables
        class RecursiveTask implements Runnable {
            public void run() {
                // Execute the provided runnable
                runnable.run();
                // Increment the count
                count[0]++;
                // Reschedule this task with increased delay
                ScheduledFuture<?> scheduledFuture = executorService.schedule(this, delay + period * count[0], TimeUnit.MILLISECONDS)
                intervalMap.put(id, scheduledFuture);
            }
        }

        // Generate a unique ID for this interval
        long id = System.currentTimeMillis();
        RecursiveTask task = new RecursiveTask();
        // Schedule the task to start after the initial delay
        ScheduledFuture<?> scheduledFuture = executorService.schedule(task, delay, TimeUnit.MILLISECONDS);
        intervalMap.put(id, scheduledFuture);

        return id; // Return the unique ID
    }

    /**
     * Clears a custom interval by ID.
     *
     * @param id The ID of the interval to clear.
     */
    public void customClearInterval(long id) {
        ScheduledFuture<?> scheduledFuture = intervalMap.get(id);
        if (scheduledFuture != null) {
            // Cancel the scheduled task
            scheduledFuture.cancel(false);
            // Remove the interval ID from the map
            intervalMap.remove(id);
        }
    }
}
```

C++ Solution

```cpp
#include <functional>
#include <chrono>
#include <thread>
#include <map>
#include <mutex>

// A map to keep track of custom intervals
std::map<long long, std::thread> intervalMap;
std::mutex intervalMapMutex; // Mutex to protect access to the intervalMap

/**
 * Creates a custom interval that mimics setInterval but allows for increasing delay durations.
 *
 * @param fn A function to execute after each delay period.
 * @param delay The initial delay before the function execution in ms.
 * @param period The additional delay added after each execution in ms.
 * @return A unique ID that can be used to clear the interval.
 */
long long customInterval(const std::function<void()>& fn, int delay, int period) {
    // Counter to track the number of times the function has been called
    int count = 0;

    // Generate a unique ID for this interval
    long long id = std::chrono::system_clock::now().time_since_epoch().count();

    // Helper lambda function that recursively sets timeouts
    std::function<void()> recursiveTimeout = [&fn, delay, period, &count, id]() {
        std::this_thread::sleep_for(std::chrono::milliseconds(delay + period * count));
        {
            // Ensure exclusive access to the intervalMap before checking
            std::lock_guard<std::mutex> lock(intervalMapMutex);
            if (intervalMap.find(id) == intervalMap.end()) {
                return; // If the interval id is not found, stop the recursion.
            }
        }
        fn(); // Execute the provided function
        count++; // Increment the count
        recursiveTimeout(); // Set up the next interval
    };

    // Start the interval in a new thread
    std::thread intervalThread(recursiveTimeout);

    // Store the thread in the intervalMap
    {
        // Ensure exclusive access to the intervalMap before inserting
        std::lock_guard<std::mutex> lock(intervalMapMutex);
        intervalMap[id] = move(intervalThread);
    }

    // Return the unique ID
    return id;
}

/**
 * Clears a custom interval by ID.
 *
 * @param id The ID of the interval to clear.
 */
void customClearInterval(long long id) {
    // Check if the interval ID exists in the map
    std::lock_guard<std::mutex> lock(intervalMapMutex);
    auto it = intervalMap.find(id);
    if (it != intervalMap.end()) {
        // If found, it means the thread is running. Join the thread before erasing it.
        if(it->second.joinable()) {
            it->second.join();
        }
        // Remove the interval ID from the map
        intervalMap.erase(it);
    }
}
```

Typescript Solution

```typescript
const intervalMap = new Map<number, NodeJS.Timeout>(); // Map to keep track of custom intervals

/**
 * Creates a custom interval that mimics setInterval but allows for increasing delay durations.
 *
 * @param {Function} fn - The function to execute after each delay period.
 * @param {number} delay - The initial delay before the function execution in ms.
 * @param {number} period - The additional delay added after each execution in ms.
 * @returns {number} An ID that can be used to clear the interval.
 */
function customInterval(fn: Function, delay: number, period: number): number {
    let count = 0; // Counter to track the number of times the function has been called

    // Helper function that recursively sets timeouts
    function recursiveTimeout() {
        // Schedule the function to be called after the calculated delay
        const timeoutId = setTimeout(() => {
            fn(); // Execute the provided function
            count++; // Increment the count
            recursiveTimeout(); // Set up the next interval
        }, delay + period * count);

        intervalMap.set(id, timeoutId); // Store the timeout reference in the map
    }

    // Generate a unique ID for this interval
    const id = Date.now();
    recursiveTimeout(); // Start the interval
    return id; // Return the unique ID
}

/**
 * Clears a custom interval by ID.
 *
 * @param {number} id - The ID of the interval to clear.
 */
function customClearInterval(id: number) {
    // Check if the interval ID exists in the map
    if (intervalMap.has(id)) {
        // Get the timeout and clear it
        clearTimeout(intervalMap.get(id)!);
        // Remove the interval ID from the map
        intervalMap.delete(id);
    }
}
```

Time and Space Complexity

Time Complexity:

The time complexity of the `customInterval` function primarily depends on the number of times the callback function `fn` is executed. However, since the scheduling of the function execution isn't a part of the actual computation, the time complexity for the JavaScript runtime to handle the execution of `customInterval` itself is `O(1)`. The time complexity of `fn` will depend on the implementation of the function that is passed to `customInterval`.

For `customClearInterval`, the time complexity is `O(1)` since it performs a constant number of operations: checking if the `id` exists in the map and potentially clearing the timeout and removing the `id` from the map.

Space Complexity:

The space complexity of the `customInterval` function is `O(n)`, where `n` is the number of active intervals. This is because it stores each interval's ID and corresponding timeout in the `intervalMap`.

Similarly, the space complexity for `customClearInterval` is `O(1)`. It does not allocate additional space that grows with the size of the input, as it simply removes an element from `intervalMap`.