2676. Throttle

Medium

Leetcode Link

Problem Description

The given problem describes a scenario where we need to create a version of a function that is "throttled." Throttling in this context means that after the function is called once, it cannot be called again for a specified duration `t` (in milliseconds). This ensures that the function doesn't execute too frequently within a short time frame. If the function is triggered again during the throttle period, the parameters (arguments) of this call are saved but not immediately acted upon. Instead, once the throttle period elapses, the function should be executed with the latest saved arguments, resetting the throttle timer for another `t` milliseconds. This effect is like postponing all function calls within the throttle period, and only the latest call's arguments are considered when the throttle period ends.

To further simplify:

1. Call the function immediately if not within a throttle period.
2. If a function call is made during a throttle period, store the latest arguments.
3. When the throttle period ends, call the function with the last stored arguments (if any) and start a new throttle period.

An illustration provided shows the effect of throttle over time, emphasizing that only the latest inputs are executed after each throttle period.

Intuition

To achieve the throttling behavior, we maintain a state to track whether we are in a "pending" or throttle period. If the function is not in the pending state, we call it immediately and set it to pending. During this pending state, all subsequent calls will not trigger an immediate function execution but will update the arguments that the function will use after the pending state is resolved.

Here's a breakdown of the solution approach:

1. Initialize a `pending` state to `false` and a variable `nextArgs` to store the latest arguments during the pending state.
2. Create a `wrapper` function that encapsulates the throttling logic:
   - If `pending` is `false`, call the function immediately with the provided arguments and set `pending` to `true`. This starts the throttle period.
   - Afterwards, for any calls received during the throttle period (when `pending` is `true`), only update `nextArgs` with the latest arguments.
3. Set a `setTimeout` with the duration `t` that will reset the `pending` state to `false` after the specified time. It will also check if there are `nextArgs` stored, and if so, call the `wrapper` function with these arguments, effectively queuing the next execution and starting a new throttle period.

This approach ensures that the function is executed at the correct time intervals while discarding all intermediate calls except for the last one before the end of the throttle period.

Solution Approach

The solution implements throttling using closures and a simple state management system to track when the function should be executed or delayed. Here's a step-by-step breakdown of the algorithm:

1. **State Tracking**: Two important pieces of state are maintained:
   - A boolean `pending` flag, which is initially set to `false` and indicates whether the function is currently in a throttled period.
   - A variable `nextArgs`, which will store the arguments passed to the most recent call that occurred during the throttle period.
2. **Closure and the Wrapper Function**: A `wrapper` function is defined, which will be returned by the `throttle` function. The `wrapper` function serves as the throttled version of the original `fn`. Closures allow the `wrapper` to have access to the `pending` and `nextArgs` variables even after the `throttle` function has finished execution.
3. **Immediate Execution**: If the function is not currently in a pending state (throttle period), it is executed immediately with the given arguments, and `pending` is set to `true`.
4. **Setting Throttle Period**: After the first execution, a `setTimeout` is set for the duration `t` milliseconds. This timeout represents the delay during which the function cannot be executed again.
5. **Handling Subsequent Calls**: If the wrapper is called again within the duration `t` (while `pending` is `true`), the arguments are stored in `nextArgs`. These arguments will be used for the next execution. If further calls are made during the delay, they only update `nextArgs` with the latest arguments, and the previous stored arguments are discarded.
6. **End of Throttle Period**: Once the timer (`setTimeout`) completes, it will set `pending` to `false`, indicating that the throttle period has ended. If there were any calls during the delay period (`nextArgs` is not `undefined`), the `wrapper` function is called again with the last stored arguments, and a new throttle period begins.
7. **Continuous Throttling**: The cycle continues with the `wrapper` function being executed at most once per every `t` milliseconds with the latest arguments that were passed to it during the previous throttle period.

Here's a high-level overview of the coding pattern used:

```
const throttle = (fn: F, t: number): F => {
    let pending = false;
    let nextArgs;
    const wrapper = (...args) => {
        nextArgs = args;
        if (!pending) {
            fn(...args);
            pending = true;
            nextArgs = undefined;
            setTimeout(() => {
                pending = false;
                if (nextArgs) wrapper(...nextArgs);
            }, t);
        }
    };
    return wrapper;
};
```

The primary data structure used is the JavaScript variable, while the predominant pattern is the use of closure and timer (`setTimeout`) for managing the throttle state. This implementation ensures a function is throttled effectively, and its latest call arguments are respected.

Example Walkthrough

Let's illustrate the solution approach with a simple example. Suppose we have a function `logMessage` that takes a string message as an argument and prints it. We want to throttle this function such that it only executes once every 2 seconds, despite being called multiple times.

Here's the `logMessage` function:

```
function logMessage(message) {
    console.log(message);
}
```

We'll create a throttled version of `logMessage` using the `throttle` function provided in the solution approach.

```
const throttledLogMessage = throttle(logMessage, 2000);
```

Now let's simulate a series of calls to `throttledLogMessage` with different messages:

```
throttledLogMessage("First call - Immediate");  // Executed immediately
throttledLogMessage("Second call - Discarded"); // Ignored during throttle period
setTimeout(() => throttledLogMessage("Third call - 2s later"), 500);  // Ignored, still in throttle period
setTimeout(() => throttledLogMessage("Fourth call - 2.5s later"), 2500); // Executed after throttle period with these arguments
```

Here's how the throttling mechanism processes each call:

1. The first call to `throttledLogMessage("First call - Immediate")` executes immediately because we are not in a throttle period, prints "First call - Immediate", and starts the 2-second throttle period.
2. During the throttle period (2 seconds), any calls to `throttledLogMessage` do not execute immediately. The second call's arguments are stored, but are soon overwritten by the third call's arguments, as it is the most recent before the timeout finishes.
3. The `setTimeout` callback set during the first call times out after 2 seconds, ending the throttle period and checking `nextArgs`. It finds that `nextArgs` contains arguments from the third call and executes `throttledLogMessage("Third call - 2s later")`. However, since we deliberately placed the fourth call outside of the throttled period (2.5 seconds later), it becomes the pending arguments (not the third call).
4. The call with "Fourth call - 2.5s later" is then executed, and a new throttle period starts.

After all these calls, the printed messages to the console will be:

```
First call - Immediate
Fourth call - 2.5s later
```

This example demonstrates how the solution makes use of closures, state variables (`pending`, `nextArgs`), and the `setTimeout` function to throttle calls, ensuring the function execution is spaced out by at least 2 seconds and that only the most recent call's arguments are used.

Python Solution

```python
from typing import Callable, Any, List
import threading
import time

# Type definition for a function that accepts any number of arguments of any type.
FunctionWithAnyArgs = Callable[..., None]

def throttle(fn: FunctionWithAnyArgs, delay: float) -> FunctionWithAnyArgs:
    """
    Creates a throttled function that only invokes the provided function at most once per every `delay` seconds.

    :param fn: The function to throttle.
    :param delay: The number of seconds to throttle invocations to.
    :returns: A new throttled version of the provided function.
    """
    # Flag to indicate if there is a pending function execution.
    is_pending = False
    # An array to store the latest arguments with which to call the function when it becomes available.
    next_args: List[Any] = []
    # Lock to synchronize access to shared data.
    lock = threading.Lock()

    def reset_pending():
        nonlocal is_pending, next_args
        with lock:
            is_pending = False
            if next_args:
                # If there are pending arguments, invoke the wrapper function with them.
                throttled_wrapper(*next_args)
                # Clear the arguments once they've been used.
                next_args = []

    def throttled_wrapper(*args: Any) -> None:
        nonlocal is_pending, next_args
        with lock:
            next_args = args  # Store the latest arguments.
            if not is_pending:  # If there is no pending execution, proceed.
                fn(*args)  # Call the original function immediately with the provided arguments.
                is_pending = True  # Set the flag to True to prevent immediate succeeding calls.
                # Start a timer that will reset the flag and call the wrapper if there are pending arguments.
                threading.Timer(delay, reset_pending).start()

    return throttled_wrapper  # Return the throttled version of the function.

# Usage example:
# def print_log(message):
#     print(message)
#
# throttled_log = throttle(print_log, 0.1)
# throttled_log("log")  # This log statement is executed immediately.
# time.sleep(0.05)
# throttled_log("log")  # This log statement will be executed after the delay (t=0.1s) due to throttling.

```

Java Solution

```java
import java.util.Timer;
import java.util.TimerTask;

// Interface to represent any function that can take any number of arguments.
interface FunctionWithAnyArgs {
    void call(Object... args);
}

public class Throttler {
    /**
     * Creates a throttled function that only invokes the provided function
     * at most once per every 'delay' milliseconds.
     *
     * @param fn    The function to throttle.
     * @param delay The number of milliseconds to throttle invocations to.
     * @return A new throttled version of the provided function.
     */
    public static FunctionWithAnyArgs throttle(FunctionWithAnyArgs fn, long delay) {
        // Object to hold the state of the throttle mechanism.
        final Object state = new Object() {
            boolean isPending = false;
            Object[] nextArgs = null;
            Timer timer = new Timer();
        };

        // The wrapper function that manages the invocation of the original function with throttling.
        return (Object... args) -> {
            synchronized (state) {
                state.nextArgs = args; // Store the latest arguments.
                if (!state.isPending) { // If there is no pending execution, proceed.
                    fn.call(args); // Call the original function immediately with the provided arguments.
                    state.isPending = true; // Set the flag to true to prevent immediate succeeding calls.
                    state.nextArgs = null;  // Clear the arguments since the function has been called.

                    // After `delay` milliseconds, reset the flag and call the wrapper if there are pending arguments.
                    state.timer.schedule(new TimerTask() {
                        @Override
                        public void run() {
                            synchronized (state) {
                                state.isPending = false;
                                if (state.nextArgs != null) { // If there are pending arguments, invoke the wrapper function with t
                                    fn.call(state.nextArgs);
                                    state.nextArgs = null; // Clear nextArgs after the invocation.
                                }
                            }
                        }
                    }, delay);
                }
            }
        };
    }

    // Usage example within a main method or any other method:
    public static void main(String[] args) {
        FunctionWithAnyArgs throttledLog = throttle((Object... logArgs) -> {
            // Assume we're printing the first argument for simplicity, real-world use might differ
            if (logArgs.length > 0) {
                System.out.println(logArgs[0]);
            }
        }, 100);

        // This log statement is executed immediately.
        throttledLog.call("log1");
        // The following log statement will be rescheduled to execute after the throttling delay.
        new Timer().schedule(new TimerTask() {
            @Override
            public void run() {
                throttledLog.call("log2");
            }
        }, 50);
    }
}

```

C++ Solution

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

// Type for a function that accepts any number of arguments of any type.
using FunctionWithAnyArgs = std::function<void()>;

class Throttler {
private:
    FunctionWithAnyArgs fn;
    int delay;
    bool isPending;
    std::mutex mtx;

public:
    Throttler(const FunctionWithAnyArgs& fn, int delay)
    : fn(fn), delay(delay), isPending(false) {}

    // Wrapper function that manages the invocation of the original function with throttling.
    template<typename... Args>
    void operator()(Args... args) {
        std::unique_lock<std::mutex> lock(mtx);
        if (!isPending) { // If there is no pending execution, proceed.
            fn(); // Call the original function immediately with the provided arguments.
            isPending = true; // Set the flag to true to prevent immediate succeeding calls.
            lock.unlock(); // Unlock the mutex before sleeping the thread.

            std::thread([this](){
                std::this_thread::sleep_for(std::chrono::milliseconds(delay));

                std::lock_guard<std::mutex> guard(this->mtx);
                this->isPending = false; // After `delay` milliseconds, reset the flag.
            }).detach();
        }
    }
};

// Usage example:
int main() {
    // Creates a throttled function that prints the argument to console.
    Throttler throttledPrint([]() {
        std::cout << "Print function called." << std::endl;
    }, 100);

    throttledPrint(); // This print function is executed immediately.

    // Simulate a call to the function after a short delay.
    std::this_thread::sleep_for(std::chrono::milliseconds(50));
    throttledPrint(); // Due to throttling, this call doesn't do anything.

    // Wait some extra time to see the effects.
    std::this_thread::sleep_for(std::chrono::milliseconds(200));

    return 0;
}

```

Typescript Solution

```typescript
type FunctionWithAnyArgs = (...args: any[]) => void; // Type for a function that accepts any number of arguments of any type.

/**
 * Creates a throttled function that only invokes the provided function at most once per every `delay` milliseconds.
 *
 * @param fn - The function to throttle.
 * @param delay - The number of milliseconds to throttle invocations to.
 * @returns A new throttled version of the provided function.
 */
const throttle = (fn: FunctionWithAnyArgs, delay: number): FunctionWithAnyArgs => {
    // Flag to indicate if there is a pending function execution.
    let isPending = false;
    // An array to store the latest arguments with which to call the function when it becomes available.
    let nextArgs: any[] | undefined;

    /**
     * The wrapper function that manages the invocation of the original function with throttling.
     *
     * @param args - Arguments with which the function is expected to be called.
     */
    const throttledWrapper = (...args: any[]): void => {
        nextArgs = args; // Store the latest arguments.
        if (!isPending) { // If there is no pending execution, proceed.
            fn(...args); // Call the original function immediately with the provided arguments.
            isPending = true; // Set the flag to true to prevent immediate succeeding calls.
            nextArgs = undefined; // Clear the arguments since the function has been called.
            setTimeout(() => { // After `delay` milliseconds, reset the flag and call the wrapper if there are pending arguments.
                isPending = false;
                if (nextArgs) throttledWrapper(...nextArgs); // If there are pending arguments, invoke the wrapper function with them
            }, delay);
        }
    };

    return throttledWrapper; // Return the throttled version of the function.
};

// Usage example:
// const throttledLog = throttle(console.log, 100);
// throttledLog("log"); // This log statement is executed immediately.
// setTimeout(() => throttledLog("log"), 50); // This log statement will be executed at t=100ms (due to throttling).

```

Time and Space Complexity

The time complexity of the `throttle` function is `O(1)` for each call, regardless of the value of `t`. This is because each function call involves only a constant number of operations: setting variables, checking conditions, and possibly scheduling a `setTimeout`. The actual throttled function `fn` is called at most once every `t` milliseconds; however, the complexity of `fn` itself does not affect the throttling mechanism.

The space complexity of the `throttle` function is `O(1)`. This is because there are a fixed number of variables used (`pending`, `nextArgs`, and `wrapper`), and their sizes do not depend on the number of times `wrapper` is called or the size of the input to `fn`. However, be aware that if the `fn` function being throttled captures a large scope or requires significant memory, that would affect the overall space complexity of the system using the throttle, but not the throttle implementation itself.