2627. Debounce

Medium

Leetcode Link

Problem Description

The core of this problem is to implement a `debounced` function. When we debounce a function, we're essentially delaying its execution until a certain amount of time (specified in milliseconds) has passed without it being called again. During this time, if the function is called again, the previous pending execution is cancelled, and the delay timer restarts. This is particularly useful in scenarios where the function is expected to be called frequently, but you only want it to execute after some quiet period, such as when the user stops typing in a search field.

It's important to note that the debounced function should be able to receive and pass on any parameters it is called with to the original function once the debounce time has elapsed without new calls.

Considering the examples provided, if the debounce time `t` is set to `50ms`:

- If the function is called at `30ms`, `60ms`, and `100ms`, the first two calls will be cancelled, and only the third call will actually be executed at `150ms`.
- If the debounce time `t` is `35ms`, the call at `30ms` will be cancelled due to the call at `60ms`, but the call at `60ms` will execute at `95ms`. The last call at `100ms` will then execute at `135ms`

Intuition

To implement the debounced function without external libraries like lodash, we start by defining a function that takes in two parameters: the function to be debounced `fn` and the debounce time `t`. Within our debounce implementation, we maintain a `timeout` variable that holds a reference to a timer created with `setTimeout`. This timer is responsible for actually calling our `fn` function after the delay `t`.

- Each time the debounced function is called, we check if there's an existing timeout timer running (which would mean a previous call is waiting to execute).
- If there is, we clear this timer using `clearTimeout` to cancel the previous call.
- We then set a new timeout with the delay `t` to call our `fn` function with the current set of arguments `args`. We use `fn.apply(this, args)` to ensure the function is called with the correct context (`this`) and arguments.

It is essential that we clear the previous timeout every time our function is called (if it exists) because this enforces the debouncing effect—only allowing the function to execute after `t` milliseconds have passed since the last call.

Solution Approach

The debounce mechanism relies on two main components: closure and timing control via `setTimeout` and `clearTimeout`. Here's how the solution provided makes use of these components:

1. Closure: We create an inner function within the `debounce` function that has access to the outer scope. This inner function uses variables `fn`, `t`, and `timeout` which are defined in the outer function's scope. The use of closures ensures that the same `timeout` variable is used across multiple calls to the debounced function, enabling us to keep track of and control the timer.

2. Timing Control: The `setTimeout` function is instrumental in creating the delay mechanism. When it's called, it sets up a timer that, after the specified time `t`, will execute the provided function—our original function `fn` applied with the call's arguments. If another call to the debounced function happens before the timer expires, `clearTimeout` is used to cancel the scheduled execution.

The detailed steps in the debounced function are as follows:

- We initialize a variable `timeout` to keep track of the `setTimeout` timer. It's declared outside the scope of the inner returned function so that it's not reinitialized with each call—this is critical for the debounced function to have memory of previous calls.
- We return an inner function that captures any arguments it's called with using the rest syntax `...args`. This function will act as our debounced function.
- Within this inner function, we first check if there's an existing timeout (which means a previous call was made within the delay window). If `timeout` is not `undefined`, we clear it using `clearTimeout`. This cancels the prior scheduled execution.
- Next, we create a new timeout using `setTimeout`, where we delay the execution of `fn` for `t` milliseconds. `fn.apply(this, args)` ensures that when the function is eventually called, it has the same `this` value as if the debounced function hadn't been used, and it also passes all the captured arguments (`args`) correctly.
- By setting the `timeout` variable with the result of `setTimeout`, we maintain a reference to the current timer, which can be cleared if the debounced function is called again within the delay period.

Through this implementation, the debounced function ensures that `fn` is called only after the debounce time has elapsed since the last invocation. This setup is particularly effective for event-handling scenarios where the frequency of events is high and we need to perform the action once the events have 'settled down.'

Example Walkthrough

Imagine you have developed a search feature in an application that searches for items as the user types in a search box. To improve performance and prevent excessive calls to the search service, you decide to implement a debounced version of the search function.

Consider your basic search function that performs a database query:

```
function searchDatabase(query) {
  console.log(`Searching for: ${query}`);
  // This would typically involve a database call
}
```

You want to ensure that this search function is called only after the user has stopped typing for at least 250ms. Here's how you could use the debouncing approach described above:

1. **Initialization**: You write a `debounce` function as described, which takes in `searchDatabase` and `250` (the delay in milliseconds) as parameters and returns a new function that you'll use for performing the search.

2. **Execution**: The user types 'apple', and on each keystroke, the debounced function is called. The following occurs:

   - User types 'a': `searchDatabase` is scheduled to be called after 250ms.
   - User types 'p': The previous timeout is cleared, and `searchDatabase` is scheduled again after 250ms.
   - This process repeats for 'p', 'l', and 'e'.

If the user stops typing after 'e', then:

3. **Completion**: Since no new keystrokes occurred for 250ms, the `searchDatabase` function is finally called with the complete query 'apple'.

The debounced function usage would look like this:

```
// This creates the debounced search function
let debouncedSearch = debounce(searchDatabase, 250);

// This simulates the user typing in the search box
debouncedSearch('a');
setTimeout(() => debouncedSearch('ap'), 100);  // After 100ms
setTimeout(() => debouncedSearch('app'), 160); // After another 60ms
setTimeout(() => debouncedSearch('appl'), 220); // After another 60ms
setTimeout(() => debouncedSearch('apple'), 400); // After another 180ms, total 400ms since first key stroke
```

What happens in this scenario is as follows:

- At 0ms, the user types 'a', and a timer is set to trigger in 250ms.
- At 100ms, the user types 'p', the previous timer is cleared, and a new timer is set.
- This repeats until the user types 'e' at 400ms. Since the user does not type any further, the last timer is not cleared and `searchDatabase('apple')` is executed at 650ms (400ms + 250ms).

In this way, the debounced function ensures that `searchDatabase` is not called constantly but only after a 250ms pause in typing, optimizing the search function's performance by reducing unnecessary database calls.

Python Solution

```python
from typing import Callable
import threading

# Type alias for a generic function that can take any number of parameters of any type
GenericFunction = Callable[..., any]

def debounce(target_function: GenericFunction, delay: int) -> GenericFunction:
    """
    Creates a debounced version of a function that delays invoking the `target_function`
    until after `delay` milliseconds have elapsed since the last time the debounced
    function was invoked.

    :param target_function: The function to debounce.
    :param delay: The number of milliseconds to delay; afterwards the `target_function` is called.
    :return: A new debounced version of the `target_function`.
    """
    # Variable to hold the timer object
    timer = None

    def debounced(*args, **kwargs):
        nonlocal timer
        # Function that will execute the target function and reset the timer
        def run_function():
            nonlocal timer
            timer = None
            target_function(*args, **kwargs)

        # If there is an existing timer, cancel it to reset the debounce timer
        if timer is not None:
            timer.cancel()

        # Set a new timer to invoke the target function after the specified delay
        timer = threading.Timer(delay / 1000.0, run_function)
        timer.start()

    return debounced

# Example usage
# def print_message(message):
#     print(message)

# debounced_print = debounce(print_message, 100)
# debounced_print('Hello')  # Invocation is cancelled
# debounced_print('Hello')  # Invocation is cancelled
# debounced_print('World')  # Actually printed to the console after 100ms

```

Java Solution

```java
import java.util.function.Consumer;

/**
 * Interface for a generic function that can take any number of parameters of any type.
 */
@FunctionalInterface
interface GenericFunction {
    void apply(Object... params);
}

/**
 * Creates a debounced version of a function that delays invoking the {@code targetFunction}
 * until after {@code delay} milliseconds have elapsed since the last time the debounced function was invoked.
 *
 * @param targetFunction The function to debounce.
 * @param delay The number of milliseconds to delay; afterwards, the {@code targetFunction} is called.
 * @return A new debounced version of the {@code targetFunction}.
 */
public static GenericFunction debounce(GenericFunction targetFunction, int delay) {
    // A holder for the timer so we can cancel it
    final Consumer<Void>[] timerHolder = new Consumer[1];

    // Return the debounced version of the target function
    return (Object... args) -> {
        // Cancel the current timer if one exists
        if (timerHolder[0] != null) {
            timerHolder[0].accept(null);
            timerHolder[0] = null;
        }

        // Create a new timer for the delay period
        timerHolder[0] = ignored -> new java.util.Timer().schedule(
            new java.util.TimerTask() {
                @Override
                public void run() {
                    targetFunction.apply(args); // Invoke the target function after the delay
                }
            },
            delay
        );
    };
}

// Example usage
public static void main(String[] args) {
    // Create a debounced version of System.out.println
    GenericFunction debouncedPrint = debounce(params -> System.out.println(params[0]), 100);

    // Call the debounced function multiple times
    debouncedPrint.apply("Hello"); // Invocation is cancelled
    debouncedPrint.apply("Hello"); // Invocation is cancelled
    new java.util.Timer().schedule(new java.util.TimerTask() { // Delay to simulate separation of calls
        @Override
        public void run() {
            debouncedPrint.apply("Hello"); // Actually printed to the console after 100ms
        }
    }, 150);
}

```

C++ Solution

```cpp
#include <iostream>
#include <functional>
#include <chrono>
#include <thread>

// Type alias for a generic function that can take any number of parameters of any type
using GenericFunction = std::function<void()>;

/**
 * Creates a debounced version of a function that delays invoking the `target_function`
 * until after `delay` milliseconds have elapsed since the last time the debounced function was invoked.
 *
 * @param target_function The function to debounce.
 * @param delay The number of milliseconds to delay; afterwards, the `target_function` is called.
 * @return A new debounced version of the `target_function`.
 */
GenericFunction debounce(const GenericFunction& target_function, int delay) {
    // Create a copy of target_function that is mutable
    auto target_function_copy = target_function;

    // Create a packaged task with the target function inside it
    std::packaged_task<void()> task(target_function_copy);

    // Variable to hold a future that is associated with the packaged task
    std::future<void> future = task.get_future();

    // Debounced version of the target function
    return [=]() mutable {
         // Static variable to hold the thread that runs the task
        static std::thread task_thread;

        // If a thread is currently running, join it.
        if(task_thread.joinable()) {
            task_thread.join();
        }

        // Create a new thread to run the task after the specified delay.
        task_thread = std::thread([=, &task]() {
            // Sleep for the delay period
            std::this_thread::sleep_for(std::chrono::milliseconds(delay));

            // Run the task which calls the original target function
            task();
        });

        // Detach the thread so it can independently complete its execution
        task_thread.detach();
    };
}

// Main function to demonstrate usage
int main() {
    // Example usage of the debounce function
    auto debounced_log = debounce([]{
        std::cout << "Hello" << std::endl;
    }, 100);

    // Invocation is cancelled
    debounced_log();

    // Invocation is cancelled
    debounced_log();

    // Actually logged to the console after 100ms
    debounced_log();

    // Make sure the main thread is kept alive until the debounced log has a chance to execute
    std::this_thread::sleep_for(std::chrono::milliseconds(200));

    return 0;
}

```

Typescript Solution

```typescript
// Type alias for a generic function that can take any number of parameters of any type
type GenericFunction = (...params: any[]) => any;

/**
 * Creates a debounced version of a function that delays invoking the `targetFunction`
 * until after `delay` milliseconds have elapsed since the last time the debounced function was invoked.
 *
 * @param targetFunction The function to debounce.
 * @param delay The number of milliseconds to delay; afterwards, the `targetFunction` is called.
 * @return A new debounced version of the `targetFunction`.
 */
function debounce(targetFunction: GenericFunction, delay: number): GenericFunction {
    // Variable to hold the timeout identifier
    let timeoutId: ReturnType<typeof setTimeout> | undefined;

    // Return the debounced version of the target function
    return function(...args: any[]): void {
        // If there is an existing timeout, clear it to reset the debounce timer
        if (timeoutId !== undefined) {
            clearTimeout(timeoutId);
        }

        // Set a new timeout to invoke the target function after the specified delay
        timeoutId = setTimeout(() => {
            targetFunction.apply(this, args);
        }, delay);
    };
}

// Example usage
// const debouncedLog = debounce(console.log, 100);
// debouncedLog('Hello'); // Invocation is cancelled
// debouncedLog('Hello'); // Invocation is cancelled
// debouncedLog('Hello'); // Actually logged to the console after 100ms

```

Time and Space Complexity

Time Complexity

The time complexity of the `debounce` function is primarily dependent on the execution of `setTimeout` and the operations within the function passed to `debounce`. Since `setTimeout` is a native web API that schedules a script to be run after a specified delay and does not block the execution flow, its complexity isn't measured in traditional algorithmic terms. However, it schedules a single delayed function execution, which gives it a constant time operation in this context (`O(1)`). The time complexity of the actual function `fn` applied is context-dependent and cannot be determined without specifics about `fn`. Therefore:

- Debouncing logic (scheduling and cancelling timeouts): `O(1)`
- Invoked function `fn`: `O(f(n))`, where `f(n)` represents the time complexity of `fn`.

Space Complexity

The space complexity of `debounce` function includes the space needed for the `timeout` variable and the arguments passed to `fn`. The `timeout` variable either holds a numeric identifier for the created timeout or is `undefined`. Hence, it consumes constant space (`O(1)`). The space consumed by arguments (`args`) depends on the number of arguments and their sizes; however, it is also a constant factor determined by the function's usage, not the `debounce` function itself. Therefore, space complexity for `debounce` is:

- For the closure and timeout identifier: `O(1)`
- For arguments and context of `fn`: `O(f(n))`, where `f(n)` represents the space complexity of maintaining the arguments and execution context of `fn`.