2666. Allow One Function Call

Easy

Leetcode Link

Problem Description

The problem statement requires us to create a higher-order function that takes a function `fn` as an argument and returns a new function. This new function must retain the behavior of the original `fn`, with the exception that it can only successfully execute once. This means that the very first time the new function is called, it should invoke `fn`, pass all given arguments to it, and then return the result of `fn`. Any calls to the new function after the first call should return `undefined` without invoking `fn` again. This ensures that `fn` is called at most once, regardless of how many times the new function is called.

Intuition

To build a solution for this problem, we need to create a function that "remembers" if it has been called before. We can achieve this by using a closure - a programming pattern that allows a function to access variables from the scope in which it was created, even after that scope has closed. We define a boolean variable `called` within the scope of the `once` function but outside of the scope of the returned function. This variable `called` is set to `false` initially, indicating that `fn` has not been called yet.

When the returned function is invoked for the first time, it checks `called` which is `false`, so it proceeds to set `called` to `true` to mark that `fn` has been called and then invokes `fn` with all the arguments it received using the `...args` spread syntax for parameters. It then returns whatever `fn` returns. For subsequent invocations, when the returned function is called again, `called` is now `true`, and the function simply returns `undefined` without calling `fn`. This logic ensures that `fn` is executed only once and enforces the restriction as specified by the problem description.

The TypeScript type annotations in the solution enforce that the types of the arguments and return of the returned function match those of the original function `fn`. This is done using generic type parameters and utility types like `Parameters` and `ReturnType` to derive the appropriate types for the arguments and return value of `fn`.

Solution Approach

The problem requires a solution that ensures a given function `fn` is only called once, regardless of how many times the newly created function is called. The solution provided uses a closure to capture the state of whether `fn` has already been called.

Here's the step-by-step implementation of the solution:

1. **Create Closure**: The main function `once` is given `fn` as an argument. Inside `once`, a variable `called` is declared and initialized to `false`. This variable captures the state within the closure of the function returned by `once`.

2. **Return New Function**: `once` returns a new function that takes a variable number of arguments using the `...args` rest syntax. This is where the closure comes into play. The variables defined in the scope of `once` (`fn` and `called`) are accessible to this returned function.

3. **Check and Invoke**: Every time the returned function is called, it first checks the value of `called`. If `called` is `false`, it means `fn` has not been called before, and the current invocation is the first.

    - The `called` variable is set to `true` to prevent any further invocation of `fn`.
    - `fn` is called with all arguments passed to the returned function, using `fn(...args)`, and the result is returned to the caller.
4. **Subsequent Calls**: If `called` is already `true` when the function is invoked (meaning `fn` has been called before), the function skips the invocation of `fn` and returns `undefined`.

The advantage of using a closure in this situation is that it allows us to maintain the state (`called`) across multiple invocations of the returned function without affecting the global scope or the scope of the function passed to `once`. Furthermore, the state is encapsulated within the returned function, adhering to good coding practices.

The solution approach also uses TypeScript generics and utility types for type inference:

- `T extends (...args: any[]) => any` defines a generic type `T` for the function which can take any number of arguments and return any type.
- `Parameters<T>` is a utility type that extracts the argument types of `T` so we can ensure the returned function accepts the same types of arguments.
- `ReturnType<T>` is used to specify that the returned function should have the same return type as `fn`.

In conclusion, the implementation uses a combination of closures, generically typed functions, and control flow based on a boolean flag to satisfy the provided problem constraints.

Example Walkthrough

Let's use a small example to illustrate the solution approach. Suppose we have a function `fn` that simply multiplies any two numbers and returns the result. We want to ensure that `fn` can only be invoked once.

Firstly, we define the function `fn`:

```
function fn(x, y) {
  return x * y;
}
```

Now, following the solution approach, we implement the `once` function:

```
function once<T extends (...args: any[]) => any>(fn: T): (...args: Parameters<T>) => ReturnType<T> | undefined {
  let called = false;

  return function(...args: Parameters<T>): ReturnType<T> | undefined {
    if (!called) {
      called = true;
      return fn(...args);
    }
    return undefined;
  };
}
```

Next, we wrap `fn` using `once` to create the `onceFn` function which will be our higher-order function:

```
const onceFn = once(fn);
```

Now, let's walk through how this would work:

1. We call `onceFn` with arguments `2` and `3` for the first time.

```
const result = onceFn(2, 3); // 'onceFn' invokes 'fn', calculates 2 * 3, and returns 6
console.log(result); // Outputs: 6
```

During this initial call:

- The `called` variable is still `false` at the start.
- The `onceFn` sees that `fn` hasn't been called, so it sets `called` to `true`.
- It invokes `fn` with the given arguments and returns the result.

2. We attempt to call `onceFn` again with any arguments - let's say `4` and `5`:

```
const secondResult = onceFn(4, 5); // 'onceFn' does not invoke 'fn' this time, returns `undefined`
console.log(secondResult); // Outputs: undefined
```

On this second invocation:

- The `called` variable is already `true`.
- The `onceFn` does not call `fn` again and simply returns `undefined`.

The internal `called` state is maintained across these calls due to the closure created within the `once` function. The `onceFn` has remembered that `fn` has already been invoked once, and it upholds our rule that `fn` only gets called one time. This example illustrates the utility of closures to encapsulate state in a functional programming context, while also showing how TypeScript can be used to maintain type safety in higher-order function scenarios.

Python Solution

```python
# Define a generic function `once` which takes a function (`func`) and returns
# a new function that can be called only once. Subsequent calls return the result
# of the first invocation, otherwise returns None.

def once(func):
    """
    Creates a function that invokes `func` once, no matter how many times it is called.
    Subsequent calls to the created function return the result of the first invocation.

    :param func: The function to restrict to be called once.
    :return: A new function that is restricted to invoking `func` only once.
    """
    # A list is used here to hold `called` and `result` as it can be modified within the inner function.
    # In Python, nonlocal variables cannot be primitive types when modified within an inner function.
    called_and_result = [False, None]

    def wrapper(*args, **kwargs):
        if not called_and_result[0]:
            called_and_result[0] = True  # Update the state to prevent further invocations.
            # Call the original function with the provided arguments and store the result.
            called_and_result[1] = func(*args, **kwargs)
        return called_and_result[1]  # Return the stored result or None if function has been called before.

    return wrapper

# Example usage:

# Define `sum_fn` as a function that takes three numbers and returns their sum.
def sum_fn(a, b, c):
    """Function that calculates the sum of three numbers."""
    return a + b + c

# Create a function `once_sum_fn` that encapsulates `sum_fn` and restricts it to a single call.
once_sum_fn = once(sum_fn)

# Call `once_sum_fn` with arguments (1, 2, 3). It should return 6.
print(once_sum_fn(1, 2, 3))  # Expected output: 6

# Attempt to call `once_sum_fn` with different arguments (2, 3, 6).
# Since `once_sum_fn` was already called, it should return None without calling `sum_fn`.
print(once_sum_fn(2, 3, 6))  # Expected output: None

```

Java Solution

```java
import java.util.function.Function;

// Create a generic functional interface `OnceFunction` that extends Function,
// specifying its argument and return type with generics.
@FunctionalInterface
interface OnceFunction<T, R> extends Function<T, R> {
    // Override the apply method from Function to define custom behavior.
    @Override
    R apply(T t);
}

public class OnceExample {

    /**
     * Creates a function that invokes the given function once, no matter how many times it's called.
     * Subsequent calls to the created function return the result of the first invocation.
     *
     * @param func The function to restrict to a single call.
     * @param <T>  The input type of the function.
     * @param <R>  The return type of the function.
     * @return A new function that is restricted to invoking the given function only once.
     */
    public static <T, R> OnceFunction<T, R> once(Function<T, R> func) {
        // Create a new instance of `OnceFunction`.
        return new OnceFunction<>() {
            // A flag to keep track if the function has been called.
            private boolean called = false;

            // The result of the first call to remember.
            private R firstResult = null;

            @Override
            public R apply(T t) {
                // Check if the function has not been called yet.
                if (!called) {
                    // If not, invoke the function with the provided arguments and store the result.
                    firstResult = func.apply(t);
                    // Update the state to prevent further invocations.
                    called = true;
                    // Return the stored result.
                    return firstResult;
                }
                // If the function was already called, return the stored result.
                return firstResult;
            }
        };
    }

    // Example usage:
    public static void main(String[] args) {
        // Define a function that takes an integer array and returns the sum of all elements.
        Function<int[], Integer> sumFn = (int[] numbers) -> {
            int sum = 0;
            for (int n : numbers) {
                sum += n;
            }
            return sum;
        };

        // Create a once-wrappable version of the `sumFn` function.
        OnceFunction<int[], Integer> onceSumFn = OnceExample.once(sumFn);

        // Call `onceSumFn` with an integer array. It should return the sum of the numbers.
        System.out.println(onceSumFn.apply(new int[]{1, 2, 3})); // Expected output: 6

        // Attempt to call `onceSumFn` again, this time with a different integer array.
        // Since `onceSumFn` has already been called, it should return the result of the first call.
        System.out.println(onceSumFn.apply(new int[]{2, 3, 4})); // Expected output: 6
    }
}

```

C++ Solution

```cpp
#include <iostream>
#include <functional>
#include <tuple>

// Define a template for a function that takes any number and type of arguments and
// returns a `std::optional` value of the function's return type.
template<typename Func>
class OnceFunction {
public:
    using ReturnType = std::optional<std::invoke_result_t<Func, decltype(std::placeholders::_1)>>;

    OnceFunction(Func&& fn) : function_(std::forward<Func>(fn)), called_(false) {}

    // A function to invoke the stored function only once. Subsequent calls return `std::nullopt`.
    template<class... Args>
    ReturnType operator()(Args... args) {
        if (!called_) {
            called_ = true; // Mark as called
            return function_(std::forward<Args>(args)...);
        }
        return std::nullopt; // Function was already called, return `std::nullopt`
    }

private:
    Func function_; // The original function that should be called at most once.
    bool called_; // Boolean flag to track if the function has been called or not.
};

/**
 * Creates a function that invokes `func` once, no matter how many times it is called. Subsequent calls
 * to the created function return `std::nullopt`.
 * @param func - The function to restrict to be called once.
 * @returns An `OnceFunction` functor that invokes `func` only once.
 */
template<typename Func>
auto once(Func&& func) {
    return OnceFunction<Func>(std::forward<Func>(func));
}

// Example usage of the once function template:

// A function that takes three numbers and returns their sum.
auto sumFn = [](int a, int b, int c) -> int {
    return a + b + c;
};

int main() {
    // Create a function `onceSumFn` that encapsulates `sumFn` and restricts it to a single call.
    auto onceSumFn = once(sumFn);

    // Call `onceSumFn` with arguments (1, 2, 3). It should return the sum, 6.
    std::cout << "First call result: ";
    if (auto result = onceSumFn(1, 2, 3); result) {
        std::cout << *result << std::endl; // Expected output: First call result: 6
    } else {
        std::cout << "undefined" << std::endl;
    }

    // Attempt to call `onceSumFn` again with different arguments (2, 3, 6).
    // Since `onceSumFn` was already called, it should return 'undefined' without calling `sumFn`.
    std::cout << "Second call result: ";
    if (auto result = onceSumFn(2, 3, 6); result) {
        std::cout << *result << std::endl;
    } else {
        std::cout << "undefined" << std::endl; // Expected output: Second call result: undefined
    }

    return 0;
}

```

Typescript Solution

```typescript
// Define a generic function type `OnceFunction` which takes a function with any number and types of arguments
// and returns its return type or undefined.
type OnceFunction<T extends (...args: any[]) => any> = (...args: Parameters<T>) => ReturnType<T> | undefined;

/**
 * Creates a function that invokes `func` once, no matter how many times it is called. Subsequent calls
 * to the created function return the result of the first invocation.
 * @param func - The function to restrict to be called once.
 * @returns A new function that is restricted to invoking `func` only once.
 */
function once<T extends (...args: any[]) => any>(func: T): OnceFunction<T> {
    // Declare a variable to track if the function has been called.
    let called = false;

    // Return a new function that captures arguments and invokes the original function only if not called before.
    return function (...args: Parameters<T>): ReturnType<T> | undefined {
        // Check if the function has already been called.
        if (!called) {
            // Update the state to prevent further invocations.
            called = true;
            // Call the original function with the provided arguments and return its result.
            return func(...args);
        }
        // If the function was called before, return undefined.
        // No further action is taken, and the original function is not called.
    };
}

// Example usage:
// Let `sumFn` be a function that takes three numbers and returns their sum.
let sumFn = (a: number, b: number, c: number) => a + b + c;

// Create a function `onceSumFn` that encapsulates `sumFn` and restricts it to a single call.
let onceSumFn = once(sumFn);

// Call `onceSumFn` with arguments (1, 2, 3). It should return 6.
console.log(onceSumFn(1, 2, 3)); // Expected output: 6

// Attempt to call `onceSumFn` with different arguments (2, 3, 6).
// Since `onceSumFn` was already called, it should return undefined without calling `sumFn`.
console.log(onceSumFn(2, 3, 6)); // Expected output: undefined

```

Time and Space Complexity

Time Complexity

The time complexity of the `once` function wrapper is `O(1)`, also referred to as constant time complexity. This is because the wrapper does not perform any operations that scale with the size of the input, it only checks a boolean flag and, if not previously called, executes the function `fn` once. The time complexity of calling the wrapped `fn` function is not included in the complexity of the `once` wrapper itself. When the original function `fn` is called via `onceFn`, its time complexity is dependent on the implementation of `fn`.

Space Complexity

The space complexity of the `once` function wrapper is `O(1)`, which means it uses a constant amount of space. It only allocates space for one boolean variable `called`. It does not allocate additional space that grows with the size of the input. However, the space complexity of the closure that includes the `fn` function and the `called` value is subject to the space requirements of `fn` itself, which are not accounted for in the `once` wrapper's space complexity.