2666. Allow One Function Call Easy

Problem Description

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.

The problem statement requires us to create a higher-order function that takes a function fn as an argument and returns a new

Leetcode Link

as specified by the problem description.

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.

Intuition

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

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

return any type.

Example Walkthrough

flag to satisfy the provided problem constraints.

function.

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

• The called variable is set to true to prevent any further invocation of fn.

encapsulated within the returned function, adhering to good coding practices.

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

return function(...args: Parameters<T>): ReturnType<T> | undefined {

- 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.
- of n 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

- 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
- 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

• Parameters<T> is a utility type that extracts the argument types of T so we can ensure the returned function accepts the same

and returns the result. We want to ensure that fn can only be invoked once. Firstly, we define the function fn:

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

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

Let's use a small example to illustrate the solution approach. Suppose we have a function fn that simply multiplies any two numbers

return undefined; 10

1 const onceFn = once(fn);

During this initial call:

On this second invocation:

Python Solution

def once(func):

16

17

18

19

20

21

22

23

24

25

27

30

31

35

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

71

70 }

C++ Solution

1 #include <iostream>

#include <tuple>

2 #include <functional>

};

};

// Example usage:

int sum = 0;

return sum;

sum += n;

11 }

called = true;

return fn(...args);

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

The called variable is still false at the start.

2 console.log(secondResult); // Outputs: undefined

maintain type safety in higher-order function scenarios.

3 # of the first invocation, otherwise returns None.

def wrapper(*args, **kwargs):

if not called_and_result[0]:

41 print(once_sum_fn(2, 3, 6)) # Expected output: None

2 console.log(result); // Outputs: 6

let called = false;

if (!called) {

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

1 const result = onceFn(2, 3); // 'onceFn' invokes 'fn', calculates 2 * 3, and returns 6

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

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

1 const secondResult = onceFn(4, 5); // 'onceFn' does not invoke 'fn' this time, returns `undefined`

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

Define a generic function `once` which takes a function (`func`) and returns

called_and_result[1] = func(*args, **kwargs)

"""Function that calculates the sum of three numbers."""

28 # Define `sum_fn` as a function that takes three numbers and returns their sum.

// Return the stored result.

Function<int[], Integer> sumFn = (int[] numbers) -> {

// Create a once-wrappable version of the `sumFn` function.

OnceFunction<int[], Integer> onceSumFn = OnceExample.once(sumFn);

// If the function was already called, return the stored result.

// Define a function that takes an integer array and returns the sum of all elements.

// Call `onceSumFn` with an integer array. It should return the sum of the numbers.

// Since `onceSumFn` has already been called, it should return the result of the first call.

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.

System.out.println(onceSumFn.apply(new int[]{2, 3, 4})); // Expected output: 6

return firstResult;

return firstResult;

public static void main(String[] args) {

for (int n : numbers) {

33 # Create a function `once_sum_fn` that encapsulates `sum_fn` and restricts it to a single call.

2 # a new function that can be called only once. Subsequent calls return the result

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.

called_and_result[0] = True # Update the state to prevent further invocations.

return called_and_result[1] # Return the stored result or None if function has been called before.

Call the original function with the provided arguments and store the result.

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.

10 :param func: The function to restrict to be called once. :return: A new function that is restricted to invoking `func` only once. 11 12 # A list is used here to hold `called` and `result` as it can be modified within the inner function. 13 # In Python, nonlocal variables cannot be primitive types when modified within an inner function. 14 called_and_result = [False, None] 15

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

36 # Call `once_sum_fn` with arguments (1, 2, 3). It should return 6. 37 print(once_sum_fn(1, 2, 3)) # Expected output: 6 38 39 # Attempt to call `once_sum_fn` with different arguments (2, 3, 6). 40 # Since `once_sum_fn` was already called, it should return None without calling `sum_fn`.

26 # Example usage:

return wrapper

def sum_fn(a, b, c):

return a + b + c

34 once_sum_fn = once(sum_fn)

```
Java Solution
   import java.util.function.Function;
    // Create a generic functional interface `OnceFunction` that extends Function,
  4 // 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);
  9
 10
 11
    public class OnceExample {
 13
 14
         /**
 15
          * Creates a function that invokes the given function once, no matter how many times it's called.
 16
          * Subsequent calls to the created function return the result of the first invocation.
 17
 18
          * @param func The function to restrict to a single call.
          * @param <T> The input type of the function.
 19
 20
          * @param <R> The return type of the function.
 21
          * @return A new function that is restricted to invoking the given function only once.
 22
 23
         public static <T, R> OnceFunction<T, R> once(Function<T, R> func) {
 24
             // Create a new instance of `OnceFunction`.
 25
             return new OnceFunction<>() {
 26
                 // A flag to keep track if the function has been called.
 27
                 private boolean called = false;
 28
 29
                 // The result of the first call to remember.
                 private R firstResult = null;
 30
 31
 32
                 @Override
 33
                 public R apply(T t) {
                     // Check if the function has not been called yet.
 34
 35
                     if (!called) {
 36
                         // If not, invoke the function with the provided arguments and store the result.
                         firstResult = func.apply(t);
 37
 38
                         // Update the state to prevent further invocations.
 39
                         called = true;
```

34 */ 37

```
5 // Define a template for a function that takes any number and type of arguments and
  6 // returns a `std::optional` value of the function's return type.
  7 template<typename Func>
  8 class OnceFunction {
    public:
         using ReturnType = std::optional<std::invoke_result_t<Func, decltype(std::placeholders::_1)>>;
 10
 11
         OnceFunction(Func&& fn) : function_(std::forward<Func>(fn)), called_(false) {}
 12
 13
 14
         // A function to invoke the stored function only once. Subsequent calls return `std::nullopt`.
 15
         template<class... Args>
 16
         ReturnType operator()(Args... args) {
 17
             if (!called_) {
                 called_ = true; // Mark as called
 18
 19
                 return function_(std::forward<Args>(args)...);
 20
 21
             return std::nullopt; // Function was already called, return `std::nullopt`
 22
 23
 24
    private:
         Func function; // The original function that should be called at most once.
 25
         bool called_; // Boolean flag to track if the function has been called or not.
 26
 27 };
 28
 29 /**
     * 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>
 36 auto once(Func&& func) {
         return OnceFunction<Func>(std::forward<Func>(func));
 38 }
 39
    // Example usage of the once function template:
 41
 42 // A function that takes three numbers and returns their sum.
    auto sumFn = [](int a, int b, int c) -> int {
         return a + b + c;
 44
 45 };
 46
 47 int main() {
         // Create a function `onceSumFn` that encapsulates `sumFn` and restricts it to a single call.
 48
         auto onceSumFn = once(sumFn);
 49
 50
 51
         // Call `onceSumFn` with arguments (1, 2, 3). It should return the sum, 6.
 52
         std::cout << "First call result: ";</pre>
         if (auto result = onceSumFn(1, 2, 3); result) {
 53
             std::cout << *result << std::endl; // Expected output: First call result: 6</pre>
 54
 55
         } else {
 56
             std::cout << "undefined" << std::endl;</pre>
 57
 58
 59
         // Attempt to call `onceSumFn` again with different arguments (2, 3, 6).
         // Since `onceSumFn` was already called, it should return 'undefined' without calling `sumFn`.
 60
         std::cout << "Second call result: ";</pre>
 61
         if (auto result = onceSumFn(2, 3, 6); result) {
 62
             std::cout << *result << std::endl;</pre>
 63
 64
         } else {
 65
             std::cout << "undefined" << std::endl; // Expected output: Second call result: undefined</pre>
 66
 67
 68
         return 0;
 69 }
 70
Typescript Solution
1 // Define a generic function type `OnceFunction` which takes a function with any number and types of arguments
2 // 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
```

// Example usage: // Let `sumFn` be a function that takes three numbers and returns their sum. 31 let sumFn = (a: number, b: number, c: number) => a + b + c; 32 // Create a function `onceSumFn` that encapsulates `sumFn` and restricts it to a single call.

let onceSumFn = once(sumFn);

Time and Space Complexity

};

let called = false;

if (!called) {

called = true;

return func(...args);

10

13

14

16

17

18

19

20

21

22

23

24

25

26

28

35

38

42

27 }

* to the created function return the result of the first invocation.

// Declare a variable to track if the function has been called.

// Check if the function has already been called.

36 // Call `onceSumFn` with arguments (1, 2, 3). It should return 6.

41 console.log(onceSumFn(2, 3, 6)); // Expected output: undefined

// Attempt to call `onceSumFn` with different arguments (2, 3, 6).

37 console.log(onceSumFn(1, 2, 3)); // Expected output: 6

* @returns A new function that is restricted to invoking `func` only once.

function once<T extends (...args: any[]) => any>(func: T): OnceFunction<T> {

return function (...args: Parameters<T>): ReturnType<T> | undefined {

// Update the state to prevent further invocations.

// No further action is taken, and the original function is not called.

40 // Since `onceSumFn` was already called, it should return undefined without calling `sumFn`.

// If the function was called before, return undefined.

// Return a new function that captures arguments and invokes the original function only if not called before.

// Call the original function with the provided arguments and return its result.

* @param func - The function to restrict to be called once.

Time Complexity The time complexity of the once function wrapper is 0(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 0(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.