2693. Call Function with Custom Context

Medium

# **Problem Description**

Function.call method without actually using it. The Function.call method is typically used to invoke a function with a specified this context and arguments. When the new callPolyfill method is called on any function, it should be capable of setting the this context of that function to an object that is passed as the first parameter to callPolyfill. Any subsequent parameters passed to callPolyfill should be treated as arguments to the original function. For example, if we have a function that calculates tax and logs the total cost of an item, normally calling this function without setting

This LeetCode problem requires us to implement a callPolyfill method that mimics the behavior of the JavaScript built-in

Leetcode Link

this would result in undefined being used for this item. But if we enhance the function with the callPolyfill method, we can specify an object that has an item property. The callPolyfill method would then ensure that within the function, this refers to the object provided, allowing the function to access the item property and log the correct message. The challenge is to enable this functionality for all functions within the global Function prototype, without relying on the existing

ntuition To arrive at the solution, we must first understand what the Function.prototype.bind method does. The bind method creates a new

function that, when called, has its this keyword set to the provided value, with a given sequence of arguments preceding any

provided when the new function is called. Since we can't use Function.call, Function.prototype.bind becomes a key part of our

### solution because it allows us to bind the this context to the function.

Function, call method.

The intuition then is to create a function from the original function using bind, where the context object is applied as this. The args that follow are then passed into this new function. Essentially, the callPolyfill function does two things: 1. It binds the given context (the obj) to the function it is called on, creating a new function where this refers to obj. 2. It immediately invokes this new function with the provided arguments.

When we use this bind (context), we're creating that new function with the context applied. The rest of the arguments are then spread into the call of this new function using ...args, which takes care of passing the arguments to the function.

This approach works because the intrinsic behavior of JavaScript's functions and the way this can be controlled with bind allows us

Remember that extending native prototypes in JavaScript can be dangerous as it might cause conflicts if the environment already

has a function with the same name or if future versions of JavaScript implement a function with that name. However, for the purpose

- to redirect the context and arguments. However, this does not alter the original function itself; instead, it creates a new function
- every time callPolyfill is called, which can be invoked right away with the correct context and arguments.

**Solution Approach** The solution approach draws upon the principles of functional programming that exist within JavaScript. In particular, it relies on the

Function.prototype.bind method to implement the functionality of Function.call without directly invoking it. Here is the step-by-step implementation using the provided TypeScript definition: 1. First, we extend the global Function interface to include the callPolyfill method. This ensures that TypeScript is aware of the

## 2. Next, we provide the actual implementation of callPolyfill by assigning a function to Function.prototype.callPolyfill. This

bound function to the context.

1 const fn = this.bind(context);

1 return fn(...args);

a callback.

of this coding problem, such considerations are out of scope.

new method and does not complain when it is used.

spread operator ...args to pass them as individual arguments.

used to pass an array of arguments as separate arguments to the function call.

method is intended to be available on all function instances since all functions inherit from Function.prototype. 3. Inside callPolyfill, we use the bind method. This method is called on the function that callPolyfill is being applied (this

refers to the function itself). We pass the context object as the first argument to bind, which will set the value of this within the

After calling bind, we now have a new function fn where this is set to context. 4. We then immediately invoke this bound function with the rest of the arguments that were passed to callPolyfill, using the

The crucial part of this approach is creating a binding between the function and a specific context, then invoking it with the desired arguments. It's effectively a manual implementation of Function. call, replicating its behavior without utilizing the inbuilt method itself.

The function bind is essential here as the mechanism to preset the this value. In JavaScript, bind is often used when the context of

this needs to be explicitly defined ahead of time, especially in cases where the function may be executed in a different scope or as

The use of the spread operator (...) is also an important aspect of the implementation because it allows an array of arguments to be

This implementation behaves much like the native Function. call, setting the this context for a function and passing along any given

arguments seamlessly, obviating the need for Function.call and showing the flexibility and power of JavaScript's function

passed to a function as if they were written out separately, making it very handy when forwarding arguments.

The new function fn is invoked with the provided arguments, and the result of the invocation is returned. The ...args syntax is

manipulation capabilities. Example Walkthrough

To illustrate the solution approach in the given content, let's consider a simple scenario where we have a function showDetails that

6 function showDetails(taxRate) { console.log(`\${this.name} costs \${(this.price \* (1 + taxRate)).toFixed(2)}`); 8 } Normally, you would use showDetails.call(product, 0.08) to call showDetails with this pointing to the product object and the tax

However, since the aim is to create a callPolyfill that does not use Function. call, let's walk through how it would be achieved

#### 1. We first add the callPolyfill function to the Function.prototype, so it becomes available on all functions: 1 Function.prototype.callPolyfill = function(context, ...args) {

Python Solution

8

9

10

11

12

13

14

16

17

18

26

29

30

31

32

33

34

36

43

47

50

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

51

52

class FunctionPolyfill:

Args:

Returns:

def increment(context):

def method(\*inner\_args):

ппп

def call\_polyfill(self, context, \*args):

\*args: Variable length argument list.

return self(context, \*inner\_args)

int: The incremented 'count' value.

print(result) # Expected output: 2

import java.lang.reflect.Method;

Object polyfillMethod(Object... args);

return method.invoke(context, args);

public static void main(String[] args) {

context.put("count", 1);

} catch (Exception e) {

e.printStackTrace();

class Incrementer implements CallPolyfill {

private HashMap<String, Integer> context;

2 import java.util.HashMap;

interface CallPolyfill {

// Example usage

try {

4 };

return fn(...args);

with our solution approach.

1 let product = {

name: 'Chocolate',

price: 1.99

needs to access the properties of an object.

Let's define an object and the showDetails function as follows:

rate of 8%, which would output "Chocolate costs 2.15".

const fn = this.bind(context);

1 showDetails.callPolyfill(product, 0.08);

When we use callPolyfill, we are essentially doing the following things:

• fn(...args): The new bound function is called immediately with the taxRate argument spread into it. In our case, it is as if we

The execution of showDetails.callPolyfill(product, 0.08) will output the expected "Chocolate costs 2.15" by setting this to the

• this.bind(context): We are creating a new function where this is permanently set to the product object. Thus, inside

product object and passing the tax rate correctly to the function. This example demonstrates how the callPolyfill method functions similarly to the native Function.call method, providing the necessary functionality without directly using Function.call.

The return value of the function after being called with the bound context and arguments.

called fn(0.08), which results in the showDetails function being executed with the tax rate as its argument.

2. Now let's use our new callPolyfill method to achieve the same result as the Function.call method.

showDetails, this name will refer to 'Chocolate' and this price will refer to 1.99.

"""Simulate the `Function.prototype.call` method in Python.

This method allows binding an object's attributes to a function,

enabling the function to access those attributes as self attributes.

context (dict): The object whose attributes should be bound.

# An example function 'increment' that increases a 'count' attribute in a context.

"""Increment a 'count' attribute within the given context.

context (dict): The context object with 'count' attribute.

# The result should be 2, as 'call\_polyfill' increments the count from 1.

result = increment\_function.call\_polyfill({'count': 1}, increment) # result is 2

# Print the result to verify the correctness of the 'call\_polyfill' implementation.

# A closure is defined to capture the function (self) and the provided context.

# Apply the 'call\_polyfill' function, setting 'this' (context) to an object with a 'count' property.

19 20 # The context's attributes are added to the function as self attributes. self.\_\_dict\_\_.update(context) 23 # Call the closure with any additional arguments provided. return method(\*args) 25

```
37
       context['count'] += 1
38
       return context['count']
39
40
   # Instantiate the FunctionPolyfill class to use the 'call_polyfill'.
   increment_function = FunctionPolyfill()
```

Java Solution

Returns:

Args:

```
public class FunctionPolyfill {
10
11
       // Static method that takes a method, a context object, and varargs for parameters
12
       // It mimics the call() function from JavaScript
13
       public static Object callPolyfill(Method method, Object context, Object... args) throws Exception {
14
           // Sets the context object's class as the method's declaring class if it's not already the same
15
           if (!context.getClass().equals(method.getDeclaringClass())) {
               throw new IllegalArgumentException("Incompatible method context provided.");
16
17
18
19
           // Invokes the method on the context with the provided arguments
```

// Create an instance of the class where increment method is defined

Method incrementMethod = Incrementer.class.getMethod("increment");

Object result = callPolyfill(incrementMethod, incrementer, context);

// Logging the result to verify the correctness of the callPolyfill implementation.

// setting 'this' to refer to a context with a count property.

// The result should be 2, as it increments the count from 1.

HashMap<String, Integer> context = new HashMap<>();

// Get the increment method from the Incrementer instance

// Apply the polyfill function to the increment method,

System.out.println(result); // Expected output: 2

// Instantiate a FunctionWrapper with the increment member function.

// Call the increment method on obj context using the polyfill method.

// Logging the result to verify the correctness of the callPolyfill implementation.

std::cout << "Result is: " << result << std::endl; // Expected output: Result is: 2

// The result should be 2, as it increments the count from 1.

FunctionWrapper<int, Context> fw(&Context::increment);

int result = fw.callPolyfill(obj); // result should be 2

// Extending the global Function interface to include callPolyfill

callPolyfill(context: Record<any, any>, ...args: any[]): any;

// This allows all functions to use the callPolyfill method

Incrementer incrementer = new Incrementer();

// Instantiate an example context object (HashMap simulates a JavaScript object with key-value pairs)

// Create a CallPolyfill interface with a polyfillMethod that needs to be implemented

```
53
         // Example method that increments a count value within a context
 54
         @Override
 55
         public Object polyfillMethod(Object... args) {
 56
             this.context = (HashMap<String, Integer>) args[0];
             this.context.put("count", this.context.get("count") + 1);
 57
 58
             return this.context.get("count");
 59
 60
         // Need to define an empty constructor since it is required when we reflectively invoke the method.
 61
         public Incrementer() {
 62
 63
             // Empty constructor
 64
 65
 66
         // Public method to be referred by the reflect method calling
 67
         public int increment() {
 68
             return (int) this.polyfillMethod(this.context);
 69
 70 }
 71
C++ Solution
 1 #include <iostream>
 2 #include <functional>
 3 #include <map>
   #include <string>
 6 // FunctionWrapper is a utility class that wraps a member function
 7 // and allows it to be called with a specific context.
 8 template<typename Return, typename Context, typename... Args>
 9 class FunctionWrapper {
10 public:
       // The constructor takes a member function and saves it.
11
       FunctionWrapper(Return(Context::*func)(Args...)) : function(func) {}
13
14
       // callPolyfill emulates the JavaScript Function.prototype.call using C++ features.
       // It calls the member function with the provided context and arguments.
15
       Return callPolyfill(Context& context, Args... args) {
16
17
           // Create a functor by binding the function to the context.
           std::function<Return(Args...)> functor = std::bind(function, &context, std::placeholders::_1, std::placeholders::_2);
19
20
           // Call the functor with the supplied arguments and return the result.
21
           return functor(args...);
22
23
   private:
       Return(Context::*function)(Args...); // Pointer to the member function
25
26 };
27
   // Context structure that will hold the 'this' context equivalent for the increment function.
   struct Context {
       int count;
30
31
       // The increment member function which will be called using callPolyfill.
32
33
       int increment() {
34
           count++;
35
           return count;
36
37 };
38
   int main() {
       Context obj = {1}; // Create a context object with count initialized to 1
40
41
```

### 13 14

return 0;

declare global {

Typescript Solution

interface Function {

42

43

44

45

46

47

48

49

50

51

52

53

54

```
// Implementing the callPolyfill method on Function's prototype.
   // This method replicates the behavior of Function.prototype.call but without using call directly.
   Function.prototype.callPolyfill = function (context: Record<any, any>, ...args: any[]): any {
       // The context object is where 'this' will point to inside the invoked function.
       // We bind 'this' (which refers to the currently called function) to the provided context.
       const boundFunction = this.bind(context);
15
       // We invoke the bound function with the provided arguments and return the result.
16
       return boundFunction(...args);
18 };
19
   // Explicitly adding the extension to the global scope.
21 export {};
22
   // Example usage:
   // Define an example function increment, which uses 'this' to refer to the context object.
   function increment(): number {
       this.count++;
26
       return this.count;
28 }
29
   // Applying the polyfill function to the increment function,
31 // setting 'this' to refer to an object with a count property.
32 // The result should be 2, as it increments the count from 1.
   const result = increment.callPolyfill({ count: 1 }); // result is 2
34
   // Logging the result to verify the correctness of the callPolyfill implementation.
   console.log(result); // Expected output: 2
37
Time and Space Complexity
The time complexity of the callPolyfill method is primarily governed by two operations:
```

## 2. The spread operator (...) which is used to pass all the arguments to the newly bound function. The .bind() method itself is generally 0(1) since it doesn't iterate over data, but the implementation might vary depending on the

JavaScript engine. The function call that happens after binding (with fn(...args)) has a time complexity dependent on the specific function being called. However, for the callPolyfill method itself, assuming that the subsequent function call is 0(1), we could consider callPolyfill to be 0(1) in the best case, where the time complexity is not influenced by the input size.

1. The .bind() method which creates a new function from the calling one, with this set to the provided context.

The space complexity of callPolyfill involves storing the newly bound function. This is 0(1) because a single function reference is

created regardless of the size of context or args. However, if the spread operator has to deal with a large number of args, this could lead to O(n) space complexity due to the creation of an arguments list to pass to the bound function, where n is the number of arguments provided.