2797. Partial Function with Placeholders

Easy

Leetcode Link

Problem Description

The challenge is to create a function called `partialFn` which modifies the behavior of an existing function `fn`. This modification involves pre-filling certain arguments of the `fn` function with values provided in an array called `args`, before the `partialFn` function is actually called. This concept is known as partial application or currying.

The `args` array may contain placeholders, represented by the underscore character `"_"`. When `partialFn` is later called with additional arguments (`restArgs`), these placeholders should be replaced in order with the provided `restArgs`. Moreover, if there are more values left in `restArgs` after filling all placeholders, these values should be appended to the `args` array.

In simple terms, when `partialFn` is eventually called, it should call the original `fn` function with a new list of arguments. This list starts with any specific values from `args`, with placeholders substituted by `restArgs`, and ends with any additional `restArgs` values appended.

Intuition

The solution to this problem centers around the concept of closures in programming. A closure is a function that captures the scope in which it was defined so that it can access variables from that scope even after the scope execution has finished.

With that in mind, the `partial` function does two key things:

1. It returns a new function, `partialFn`, that 'remembers' the original function `fn` and the pre-filled `args` array via a closure.
2. When `partialFn` is called, it replaces each placeholder (`"_"`) in `args` with the corresponding element from `restArgs`. Then, if there are additional `restArgs` left, it appends them to the end of `args`.

The intuition behind replacing placeholders and appending additional `restArgs` is similar to merging two lists with the twist that some elements in the first list are explicitly marked for replacement.

When all arguments are in place, the final step is to call the original function `fn` using the spread operator `...`, which allows us to pass the arguments as separate values even though they are contained in an array.

This implementation provides a flexible way to re-use functions by pre-filling some of their arguments and deciding others at a later point in time.

Solution Approach

The reference solution provided is an implementation of the partial function application using JavaScript (or TypeScript in this case) closures and the spread operator.

Here's a step-by-step breakdown of the implementation:

1. **Creating the Closure**: The function `partial` takes a function `fn` and an array `args` as parameters. It returns a new function `partialFn` that creates a closure over `fn` and `args`.
2. **Using the Spread Operator**: Within the body of `partialFn`, we use the rest parameter syntax (`...restArgs`) to collect all additional arguments supplied to `partialFn` into an array called `restArgs`.
3. **Replacing Placeholders**: We then iterate over `args` and replace any placeholders (`"_"`) with the corresponding values from `restArgs`. We use a variable `i` to keep track of the current index of `restArgs` from which we need to take the next value.

```
for (let i = 0; i < args.length; ++i) {
    if (args[i] === '_') {
        args[i] = restArgs[i++];
    }
}
```

4. **Appending Additional Arguments**: After all placeholders are replaced, we check if there are still any unused arguments left in `restArgs`. If there are, they are appended to the end of `args`.

```
while (i < restArgs.length) {
    args.push(restArgs[i++]);
}
```

5. **Invoking the Original Function**: Finally, we use the spread operator again to call the original function `fn` with the modified `args` array as individual arguments.

```
return fn(...args);
```

This solution approach is efficient because:

- We have a single pass over the `args` array to replace placeholders.
- We have a single pass over the remaining elements of `restArgs` (if any) to append them.
- We ensure that we call the original function with precisely the right set of arguments, respecting the order and placeholders' replacement requirements.

This implementation leverages common JavaScript features such as closures to ensure that the `partial` function retains access to the original `fn` and `args` variables when `partialFn` is later invoked. The spread operator simplifies argument handling, allowing us to treat the elements of an array as individual parameters to another function.

Example Walkthrough

Let's illustrate the solution approach with a simple example. Suppose we have a function `multiply` that takes two numbers and returns their product:

```
function multiply(a, b) {
  return a * b;
}
```

Now, we want to use the `partialFn` to create a new function where the first argument of `multiply` is pre-filled with `3`. We also want to use `_` as a placeholder for the second argument that we will provide later.

Here is how we use `partialFn`:

```
const double = partial(multiply, [3, '_']);
```

In the above code, `double` is a new function that, when called with a number, will multiply that number by `2`.

Now let's call `double` with an additional argument, `3`:

```
const result = double(3);
```

Here's what happens step-by-step:

1. We call `double` which is our `partialFn` with `3` as `restArgs`.
2. Inside `partialFn`, we iterate over `args` (`[3, '_']`), and when we find a placeholder (`'_'`), we replace it with the first element from `restArgs` (which is `3`).
3. Now `args` becomes `[3, 3]`.
4. There are no additional elements in `restArgs` to append to `args`.
5. We then call the original `multiply` function with the elements of `args` spread into separate arguments, effectively calling `multiply(3, 3)`.
6. `multiply` returns `9`, which is then returned by `double`.

The result is `9`, which is the product of `3` and `3`. Our `partialFn` successfully created a new function that doubled its input. This demonstrates how the `partial` function application uses closures and the spread operator to pre-fill and replace function arguments on-demand.

Python Solution

```
def partialFn(fn, preset_args):
    """
    Creates a partially applied function with the possibility to have placeholders.
    :param fn: The original function to be partially applied.
    :param preset_args: List of arguments to apply, where None represents placeholders for additional arguments.
    :return: A new function with some arguments preset and remaining arguments to be provided later.
    """
    def partially_applied(*additional_args):
        # Create a copy of the preset arguments to avoid modifying the original list.
        final_args = list(preset_args)
        add_args_index = 0

        # Iterate over the preset arguments and replace placeholders with additional arguments.
        for index in range(len(final_args)):
            if final_args[index] is None:
                final_args[index] = additional_args[add_args_index]
                add_args_index += 1

        # Append any remaining additional arguments that were not used as placeholders.
        while add_args_index < len(additional_args):
            final_args.append(additional_args[add_args_index])
            add_args_index += 1

        # Call the original function with the combined set of arguments.
        return fn(*final_args)

    return partially_applied

# Example usage:
# def add(a, b, c):
#     return a + b + c
# add_partial = partial(add, 1, None, 3)
# print(add_partial(2))  # Outputs 6 (1 + 2 + 3)
```

Java Solution

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

/**
 * This class provides a way to create a partially applied function with placeholders.
 */
public class PartialFunction {

    /**
     * Creates a partially applied function with the possibility to have placeholders.
     *
     * @param fn            The original function to be partially applied.
     * @param presetArgs    Array of arguments to apply, where null represents placeholders for additional arguments.
     * @return A new function with some arguments preset and remaining arguments to be provided later.
     */
    public static Function<Object[], Object> partialFunction(Object[], Object> fn, Object[] presetArgs) {
        return (additionalArgs) -> {
            Object[] finalArgs = new Object[presetArgs.length];
            int addArgsIndex = 0;

            // Iterate over the preset arguments and replace placeholders with additional arguments.
            for (int i = 0; i < presetArgs.length; i++) {
                if (presetArgs[i] == null) {
                    if (addArgsIndex < additionalArgs.length) {
                        finalArgs[i] = additionalArgs[addArgsIndex++];
                    }
                } else {
                    finalArgs[i] = presetArgs[i];
                }
            }

            // Append any remaining additional arguments that were not used as placeholders.
            Object[] combinedArgs = new Object[finalArgs.length + additionalArgs.length - addArgsIndex];
            System.arraycopy(finalArgs, addArgsIndex, combinedArgs, 0, finalArgs.length);
            if (addArgsIndex < additionalArgs.length) {
                System.arraycopy(additionalArgs, addArgsIndex, combinedArgs, finalArgs.length, additionalArgs.length - addArgsIndex);
            }

            // Call the original function with the combined set of arguments.
            return fn.apply(combinedArgs);
        };
    }

    // Example usage:
    public static void main(String[] args) {
        Function<Object[], Object> add = (Object[] params) -> {
            int sum = 0;
            for (Object param : params) {
                sum += (int) param;
            }
            return sum;
        };

        Function<Object[], Object> addPartially = partialFunction(add, new Object[]{1, null, 3});
        System.out.println(addPartially.apply(new Object[]{2}));  // Outputs 6 (1 + 2 + 3)
    }
}
```

C++ Solution

```
#include <functional>
#include <vector>
#include <iostream>

/**
 * Represents a placeholder for future arguments.
 * This could be replaced with std::optional or a dedicated placeholder type for more complex scenarios.
 */
struct auto _ = nullptr;

/**
 * Creates a partially applied function with the possibility to have placeholders.
 * This is a variadic template function to support any number of arguments.
 * @param presetArgs Vector of arguments to apply, where nullptr represents placeholders for additional arguments.
 * @return A new function with some arguments preset and remaining arguments to be provided later.
 */
template<typename ReturnType, typename... Args>
auto partialFn(ReturnType(&fn)(Args...), std::vector<std::variant<Args..., std::nullptr_t>> presetArgs) {
    // The returned lambda captures fn and presetArgs by value.
    return [fn, presetArgs](Args... additionalArgs) mutable -> ReturnType {
        std::vector<std::variant<Args..., std::nullptr_t>> finalArgs(presetArgs);
        std::queue<std::variant<Args..., std::nullptr_t>> addArgsQueue({additionalArgs...});

        // Iterates over the preset arguments and replaces placeholders with additional arguments.
        for (auto& arg : finalArgs) {
            if (std::holds_alternative<std::nullptr_t>(arg)) {
                if (!addArgsQueue.empty()) {
                    arg = addArgsQueue.front();
                    addArgsQueue.pop();
                }
            }
        }

        // Appends any remaining additional arguments that were not used as placeholders.
        while (!addArgsQueue.empty()) {
            finalArgs.push_back(addArgsQueue.front());
            addArgsQueue.pop();
        }

        // Calls the original function with the combined set of arguments using std::apply.
        return std::apply(fn, finalArgs);
    };
}

// Example usage
int add(int a, int b, int c) {
    return a + b + c;
}

int main() {
    std::function<int(int, int, int)> addFn = add;

    // Create a partially applied function with a placeholder for the second argument
    auto addPartially = partialFn(addFn, {1, _, 3});

    // Outputs 6 (1 + 2 + 3)
    std::cout << addPartially(2) << std::endl;

    return 0;
}
```

Typescript Solution

```
/**
 * Creates a partially applied function with the possibility to have placeholders.
 * @param fn The original function to be partially applied.
 * @param presetArgs Array of arguments to apply, where '_' represents placeholders for additional arguments.
 * @return A new function with some arguments preset and remaining arguments to be provided later.
 */
function partialFn(fn: (...args: any[]) => any, presetArgs: any[]): (...args: any[]) => any {
    return function(...additionalArgs: any[]): any {
        // Create a copy of the preset arguments to avoid modifying the original array.
        const finalArgs = [...presetArgs];
        let addArgsIndex = 0;

        // Iterate over the preset arguments and replace placeholders with additional arguments.
        for (let index = 0; index < finalArgs.length; ++index) {
            if (finalArgs[index] === '_') {
                if (addArgsIndex < additionalArgs.length) {
                    finalArgs[index] = additionalArgs[addArgsIndex++];
                }
            }
        }

        // Append any remaining additional arguments that were not used as placeholders.
        while (addArgsIndex < additionalArgs.length) {
            finalArgs.push(additionalArgs[addArgsIndex++]);
        }

        // Call the original function with the combined set of arguments.
        return fn(...finalArgs);
    };
}

// Example usage:
// const add = (a, b, c) => a + b + c;
// const addPartially = partial(add, [1, '_', 3]);
// console.log(addPartially(2));  // Outputs 6 (1 + 2 + 3)
```

Time and Space Complexity

The given TypeScript code defines a function named `partial`, which takes a function `fn` and an array `args` containing any number of arguments, where some arguments can be placeholders (denoted by `'_'`). It returns a new function that, when called, will invoke `fn` with a combination of the `args` provided during partial application and the arguments passed to the newly created function. Here is an analysis of its computational complexity:

Time Complexity

The time complexity of the `partial` function itself, during its creation, is $O(n)$ where n is the length of the `args` array. This is due to the initial for-loop which iterates over the entire `args` array to replace the placeholders with actual values when the returned function is eventually called.

When the returned function is called (`...restArgs`), it iterates over the `args` array up to a maximum of n times again (to replace placeholders), and then it iterates over `restArgs` which can be denoted as m, pushing any remaining arguments into the `args` array. Thus, the time complexity for the returned function invocation is $O(n + m)$, where n is the number of arguments passed when the returned function is called.

Combining these, invoking the returned function has a time complexity of $O(n + m)$.

Space Complexity

The space complexity of the `partial` function is also $O(n)$. Additional space is required for storing the partially applied arguments (`args`). When the returned function is called and `args` is updated, no additional space proportional to `args` length is required because the changes happen in place.

However, it should be noted that this analysis assumes the `fn` function that is eventually called does not allocate additional space, as the space complexity of `fn` might have implications on the overall space complexity of the returned function. But with the given code, ignoring `fn`'s own complexity, the space complexity remains $O(n)$.