2775. Undefined to Null

Medium

Leetcode Link

Problem Description

The task is to write a function called `undefinedToNull` that accepts a parameter `obj`. This parameter could be a deeply nested object or an array. The purpose of this function is to traverse the entire structure of `obj`, and wherever an `undefined` value is found, it should be replaced with `null`. This process is essential because when JavaScript objects are serialized into a JSON string using `JSON.stringify()`, `undefined` values can cause problems while `null` does not. Ensuring all `undefined` values are converted to `null` can help avoid unexpected errors when the data is serialized.

Intuition

To solve this problem, we need to perform a deep traversal of the input object or array. We can use a recursive approach. The intuition is to iterate over each property if the input is an object or over each element if it's an array. If a property or element is another object or array, we should recursively call the same function on that sub-object or sub-array to handle any nested structures.

This process allows us to reach every level of nesting. When we encounter an `undefined` value at any level, we replace it with `null`. We continue this process until we have checked and possibly replaced every `undefined` value throughout the entire structure. The function then returns the modified object or array, which has all `undefined` values converted to `null`.

The solution provided does not check explicitly for arrays, assuming that an array is an object with numerical keys. However, this might have unintended side effects in TypeScript or JavaScript, considering Arrays have different characteristics than plain objects. For full robustness, the solution could be modified to handle arrays and objects distinctly.

Solution Approach

The solution uses a recursive function `undefinedToNull` to traverse the deeply nested objects or arrays and replace `undefined` values with `null`. The approach can be broken down into the following steps:

1. Iterate through each property of the object using a `for...in` loop.
2. Check if the current property's value is an object itself. If it is, we recursively call `undefinedToNull` on this sub-object to handle nested structures.
3. After the recursive call (or immediately, if the property is not an object), check if the current property's value is `undefined`.
4. If the value is `undefined`, update it to `null`.
5. Once the loop is complete, which signifies that all properties have been checked and updated if necessary, return the modified object.

Key Points of the Implementation:

- **Recursive Function:** The function `undefinedToNull` calls itself to handle nested objects or arrays. This allows the function to handle any level of nesting.
- **In-Place Updates:** The function modifies the input object directly, which may or may not be desired. To avoid mutating the input, one would need to create a copy of each nested object or array before making modifications.
- **TypeScript Signatures:** The function is written using TypeScript, with `Record<any, any>` used as the type signature. This indicates that the function expects an object-like structure with keys and values of any type.

Potential Improvements:

While this implementation achieves the task it sets out to do, a more robust implementation might consider the following:

- **Immutable Updates:** Instead of modifying the input object, returning a new object with the applied changes to keep the function pure, avoiding side effects.
- **Array Checks:** Adding explicit checks to determine if the object is `Array` to use array-specific methods might be necessary for some edge cases.
- **Handling Circular References:** In its current form, the function would be trapped in an infinite loop if the object contains circular references. Handling such cases requires tracking visited objects.

The resolved solution effectively ensures that all `undefined` values within the nested structure are turned to `null`, a critical requirement when serializing the data with `JSON.stringify()`, as `undefined` values are not included, while `null` values are preserved in the serialized JSON.

Example Walkthrough

Let's say we want to use the `undefinedToNull` function on a sample object to understand how it will replace `undefined` with `null`. Consider the following example object, which has multiple levels of nesting and contains both `undefined` and properly defined values:

```
let sampleObject = {
  level1: {
    level2a: undefined,
    level2b: {
      level3: 'data',
      level3_undefined: undefined
    }
  },
  level1_array: [1, undefined, 3],
  level1_undefined: undefined
};
```

Now, let's walk through how the solution approach will handle this `sampleObject`:

1. We start by calling `undefinedToNull(sampleObject)`.
2. The function begins iterating through the properties of `sampleObject`.
3. The first property is `level1`, an object, so we call `undefinedToNull` on `level1`.
4. Inside this call, we find `level2a` with a value of `undefined`. We replace it with `null`.
5. Next, `level2b` is an object, so we call `undefinedToNull` on `level2b`.
6. Here, we find `level3_undefined` set to `undefined` and update it to `null`. The `level3` property is not modified as it's a string.
7. Returning from the recursion, we are back at the top level and move to `level1_array`, an array.
8. The array contains `undefined` at index 1, which we replace with `null`.
9. Finally, we update `level1_undefined` to `null` since its value is `undefined`.

After all the recursive calls, `sampleObject` is directly modified, and the properties with `undefined` values have been replaced with `null`. The output would be:

```
{
  level1: {
    level2a: null,
    level2b: {
      level3: 'data',
      level3_undefined: null
    }
  },
  level1_array: [1, null, 3],
  level1_undefined: null
}
```

With the defined approach, we successfully traverse the entire structure of the object, replacing all `undefined` values with `null`, readying our `sampleObject` for any serialization that may occur, without any errors related to `undefined` values.

Python Solution

```python
def undefined_to_null(obj):
    """
    Recursively converts all 'None' values within an object (dictionary) or a list to 'None'.

    :param obj: The object (dictionary) or list to be processed.
    :return: The new object (dictionary) or list with 'None' instead of 'None' values.
    """
    # Check if 'obj' is a dictionary
    if isinstance(obj, dict):
        for key in obj:
            # If the value is a dictionary or a list, apply recursion
            if isinstance(obj[key], (dict, list)):
                obj[key] = undefined_to_null(obj[key])
            # If the value is 'None', replace with 'None'
            if obj[key] is None:
                obj[key] = None

    # Check if 'obj' is a list
    elif isinstance(obj, list):
        for index, value in enumerate(obj):
            # If the element is a dictionary or a list, apply recursion
            if isinstance(value, (dict, list)):
                obj[index] = undefined_to_null(value)
            # If the element is 'None', replace with 'None'
            if value is None:
                obj[index] = None

    return obj

# Usage examples:
# print(undefined_to_null({"a": None, "b": 3})) # {"a": None, "b": 3}
# print(undefined_to_null([None, None])) # [None, None]

```

Java Solution

```java
import java.util.List;
import java.util.Map;

public class UndefinedToNullConverter {

    /**
     * Recursively converts all null values within a map or list to a specific value.
     *
     * @param obj The map or list to be processed.
     * @return The new map or list with nulls instead of undefined values.
     */
    public static Object undefinedToNull(Object obj) {
        if (obj instanceof Map<?, ?>) {
            Map<?, ?> map = (Map<?, ?>) obj;
            for (Object key : map.keySet()) {
                Object value = map.get(key);
                if (value instanceof Map<?, ?> || value instanceof List<?>) {
                    map.put(key, undefinedToNull(value));
                }
                if (value == null) {
                    map.put(key, null);
                }
            }
        } else if (obj instanceof List<?>) {
            List<?> list = (List<?>) obj;
            for (int i = 0; i < list.size(); i++) {
                Object value = list.get(i);
                if (value instanceof Map<?, ?> || value instanceof List<?>) {
                    list.set(i, undefinedToNull(value));
                }
                if (value == null) {
                    list.set(i, null);
                }
            }
        }
        return obj;
    }

    // Usage examples:
    public static void main(String[] args) {
        Map<String, Object> map = Map.of("a", null, "b", 3);
        System.out.println(undefinedToNull(map)); // {a=null, b=3}

        List<Object> list = List.of(null, null);
        System.out.println(undefinedToNull(list)); // [null, null]
    }
}

```

C++ Solution

```cpp
#include <iostream>
#include <map>
#include <vector>
#include <any>

// Function template that works with both std::map and std::vector as a container.
template <typename T>
T undefinedToNull(T container) {
    // Iterate over all keys if container is a map or over indices if it's a vector
    for (auto& element : container) {
        // std::any is used to hold any type, similar to the JavaScript object values
        auto& value = element.second; // For maps, access the value part
        if constexpr (std::is_same_v<T, std::vector<std::any>>) {
            value = element; // For vectors, each element is the value itself
        }

        // If the value is an object (std::map) or an array (std::vector) itself, apply the function recursively
        if (value.has_value() && value.type() == typeid(T)) {
            value = std::any_cast<T>(value);
            value = undefinedToNull(std::any_cast<T>(value));
        }
        // Convert any undefined (empty std::any) values to null (std::any with nullptr)
        if (!value.has_value()) {
            value = std::any(nullptr);
        }
    }

    return container;
}

// Overload of undefinedToNull for std::map, since the original JavaScript function supports objects.
template <typename K, typename V>
std::map<K, V> undefinedToNull(std::map<K, V> obj) {
    for (auto& [key, value] : obj) {
        // If the value is an object (std::map) or an array (std::vector) itself, apply the function recursively
        if (std::holds_alternative<std::map<K, V>>(value) || std::holds_alternative<std::vector<V>>(value)) {
            value = std::visit([](auto&& arg) -> V { return undefinedToNull(arg); }, value);
        }
        // Convert any undefined (empty std::variant) values to null (std::variant with nullptr)
        if (!std::holds_alternative<V>(value)) {
            value = nullptr;
        }
    }

    return obj;
}

// Example usage:
int main() {
    // Create a map with one element being undefined (empty std::any)
    std::map<std::string, std::any> myMap = {{"a", std::any()}, {"b", 3}};
    myMap = undefinedToNull(myMap);

    // Create a vector with elements being undefined (empty std::any)
    std::vector<std::any> myVector = {std::any(), std::any()};
    myVector = undefinedToNull(myVector);

    // Output sanitized containers
    // To print the values from myMap and myVector, you would need to add custom code to handle the type-erased std::any values.

    return 0;
}

```

Typescript Solution

```typescript
/**
 * Recursively converts all undefined values within an object or array to null.
 * @param {Record<any, any> | any[]} obj - The object or array to be processed.
 * @returns {Record<any, any> | any[]} The new object or array with nulls instead of undefined values.
 */
function undefinedToNull(obj: Record<any, any> | any[]): Record<any, any> | any[] {
    // Iterate over all keys if obj is an object or over indices if it's an array
    for (const key in obj) {
        // If the value is an object or an array itself, apply the function recursively
        if (obj[key] && typeof obj[key] === 'object') {
            obj[key] = undefinedToNull(obj[key]);
        }
        // Convert any undefined values to null
        if (obj[key] === undefined) {
            obj[key] = null;
        }
    }

    return obj;
}

// Usage examples:
// console.log(undefinedToNull({"a": undefined, "b": 3})); // {"a": null, "b": 3}
// console.log(undefinedToNull([undefined, undefined])); // [null, null]

```

Time and Space Complexity

Time Complexity

The time complexity of the `undefinedToNull` function is essentially O(N), where N is the total number of elements and nested elements within the object. This is because the function must visit each element once to check for `undefined` and convert it to `null`.

Here's the breakdown:

- The `for` loop iterates over each property in the input object.
- The `if (typeof obj[key] === 'object')` check runs in constant time O(1) for each element.
- The recursive call to `undefinedToNull` for objects ensures that every nested object is also traversed.

So if we have a single-level object with `n` keys, the complexity is O(n). For nested objects, the function will traverse into each nested object, which could contain up until n elements itself, resulting in the time complexity of O(n + n^2) for two levels of nesting, and so on, depending on the depth and structure of the nesting. Despite the potential for factorial growth with nested structures, it is more practical to consider this an O(n) operation, where n is the total count of elements including nested ones, because each is only visited once.

Space Complexity

The space complexity of the function is also O(N). Although the function runs in-place by altering the passed object, due to recursive calls, each call adds a new frame to the call stack with its own execution context; this leads to additional space usage in proportion to the depth of recursion.

For an object with a depth of `d` and branching factor `b` at each level (assuming each property is an object), we would effectively have a recursion depth of `d`, each adding to the space complexity. Hence, the space complexity could be considered to be O(d), where `d` is the depth of the object.

However, if we consider `N` to represent the total number of element and nested elements within the object (accounting for all levels of nesting), then the space complexity simplifies to O(N) since we accumulate a call stack frame for each element we visit.