2727. Is Object Empty

Easy

Leetcode Link

Problem Description

This LeetCode problem asks us to write a function that determines whether a given object or array is empty. An object is considered empty if it has no key-value pairs, while an array is deemed empty if it contains no elements. It's mentioned that we may assume the object or array input to our function is the result of `JSON.parse`, implying that these inputs are valid JSON objects and arrays.

In programming, detecting whether a collection like an array or an object is empty is a common task, since actions might be taken based on this state. An empty array might signal that there are no items to iterate over, while an empty object might indicate that there are no attributes to process. Thus, having a clear and efficient way to check for this condition is useful in many scenarios.

Intuition

The solution approach leverages a simple for...in loop to iterate over the properties of the input object or elements of the array. The for...in loop in TypeScript (and JavaScript) returns the keys of an enumerable object or indexes of an array.

Here's the intuition behind using this loop:

1. Begin iterating over the object or array using a for...in loop.
2. If the loop body is executed (even once), it means there is at least one key in the object or one item in the array which makes the object or array non-empty.
3. Therefore, if the loop executes, we immediately return `false` indicating the object/array is not empty.
4. If the loop completes without executing its body, it means there were no keys/items to iterate over, and hence we return `true` indicating that the object/array is indeed empty.

This approach capitalizes on the idea that an iteration would only occur if there is something in the object or array; otherwise, the loop would simply be skipped, validating that it is empty. It's an efficient approach because it stops checking as soon as a single property or element is found, without unnecessary iteration over the entire collection.

Solution Approach

The implementation of the solution is straightforward and utilizes the for...in loop, which is an efficient way to iterate over the enumerable properties of objects and the elements of arrays in JavaScript (and TypeScript).

Let's walk through the solution code piece by piece:

```typescript
function isEmpty(obj: Record<string, any> | any[]): boolean {
    for (const x in obj) {
        return false;
    }
    return true;
}
```

1. The `isEmpty` function accepts a parameter `obj` which can be either an object with string keys (`Record<string, any>`) or an array (`any[]`). This is indicated by the TypeScript type union syntax `Record<string, any> | any[]`.

2. Inside the function, we have a for...in loop `for (const x in obj) {...}`. This loop will iterate over all the enumerable properties if `obj` is an object, or over all the indices if `obj` is an array. Here `x` will be assigned the keys or indices.

3. If the loop body is executed, the function immediately returns `false` with `return false;`. This is because the presence of any key or index indicates that the object or array is not empty.

4. If the loop completes without executing the loop body (which would happen if there are no properties in an object or no elements in an array), the function proceeds to the next line after the loop which returns `true` with `return true;`.

This approach is algorithmically simple and does not require any additional data structures or complex patterns. It exploits the inherent behavior of the for...in loop to detect the presence of enumerable properties or elements. By returning immediately upon finding the first property or element, the function ensures a best-case runtime of O(1) when the object or array is not empty. In the worst-case scenario, when the input is actually empty, it goes through a single iteration checking all enumerable properties or elements which also takes O(1) time because there are none.

Overall, this solution is both time-efficient and space-efficient, requiring no extra memory beyond the input and a single variable assignment in the loop.

Example Walkthrough

Let's assume we need to use the `isEmpty` function for two different cases: one with an empty object and another with a non-empty array.

Case 1: Empty Object

Suppose we call `isEmpty({})` to determine if an empty object `{}` is indeed empty.

1. The `isEmpty` function begins a `for...in` loop, trying to iterate over any properties in `{}`.
2. Since `{}` has no properties, the loop body does not execute.
3. The function reaches the `return true;` statement after the loop, indicating the object is empty.
4. The function exits, returning `true`.

Case 2: Non-Empty Array

Now, consider we call `isEmpty([1, 2, 3])` to check if an array `[1, 2, 3]` is empty.

1. The `isEmpty` function starts the `for...in` loop, iterating over elements in `[1, 2, 3]`.
2. The loop body is executed because the array has elements (the indices 0, 1, 2 corresponding to the array values 1, 2, 3).
3. Upon the first iteration, the function hits the `return false;` statement within the loop.
4. It returns `false` since the array is not empty.
5. The function exits, confirming the array contains items.

Through these two cases, we can see how the `isEmpty` function operates. It uses the for...in loop to immediately detect if a given object has keys or if an array has any items, allowing for quick determination of whether the collection is empty. If the collection has contents, the function returns `false` upon the first discovery of an element or property; otherwise, it completes the loop without finding anything and returns `true`.

Python Solution

```python
def is_empty(obj):
    """
    Checks if a dictionary or list is empty.

    Args:
        obj (dict | list): The dictionary or list to check for emptiness.

    Returns:
        bool: True if the dictionary or list is empty, False otherwise.
    """
    # Use the built-in 'not' operator to check for emptiness
    return not bool(obj)

```

Java Solution

```java
import java.util.Map;
import java.util.Collection;

public class ObjectUtil {

    /**
     * Checks if a Map or Collection is empty.
     *
     * @param obj the Map or Collection to check for emptiness
     * @return true if the Map or Collection is empty, false otherwise
     */
    public static boolean isEmpty(Object obj) {
        // Check null before instance check
        if (obj == null) {
            return true;
        }

        // If the object is a Map, check if it is empty
        if (obj instanceof Map) {
            return ((Map<?, ?>) obj).isEmpty();
        }

        // If the object is a Collection, check if it is empty
        if (obj instanceof Collection) {
            return ((Collection<?>) obj).isEmpty();
        }

        // If the object is neither a Map nor a Collection, throw an IllegalArgumentException
        throw new IllegalArgumentException("Input must be of type Map or Collection.");
    }
}

```

C++ Solution

```cpp
#include <iostream>
#include <unordered_map>
#include <vector>

/**
 * Checks if a map (object) or vector (array) is empty.
 *
 * @param obj - Map or vector to check for emptiness.
 * @return True if the map or vector is empty, false otherwise.
 */
template <typename T>
bool isEmpty(const T& obj) {
    // The provided object is a standard C++ container which has a member function empty()
    // to check for emptiness. Instead of manually iterating over the elements, we can just
    // call this function to determine if the container is empty.
    return obj.empty();
}

int main() {
    // Example usage:
    std::unordered_map<std::string, int> mapExample; // Represents an object with string keys and int values
    std::vector<int> vectorExample; // Represents an array of ints

    bool isMapEmpty = isEmpty(mapExample); // Should return true because the map is empty
    bool isVectorEmpty = isEmpty(vectorExample); // Should return true because the vector is empty

    std::cout << "Is the map empty? " << (isMapEmpty ? "Yes" : "No") << std::endl;
    std::cout << "Is the vector empty? " << (isVectorEmpty ? "Yes" : "No") << std::endl;

    return 0;
}

```

Typescript Solution

```typescript
/**
 * Checks if an object or array is empty.
 *
 * @param {Record<string, any> | any[]} obj - Object or array to check for emptiness.
 * @returns {boolean} - True if the object or array is empty, false otherwise.
 */
function isEmpty(obj: Record<string, any> | any[]): boolean {
    // Iterate over the elements or properties in the object or array
    for (const key in obj) {
        // If the loop has started, it means there is at least one property or element, so return false
        return false;
    }

    // If the loop did not find any properties or elements, return true indicating emptiness
    return true;
}

```

Time and Space Complexity

Time Complexity

The `isEmpty` function iteratively checks if there is at least one property or element in the input object or array. The time complexity is `O(N)` in the worst case, where `N` is the number of properties in the object or the number of elements in the array. This is because the function must iterate over all enumerable properties or elements until it finds one. If an enumerable property or element is found immediately, at the start of the iteration, the for-loop breaks and returns `false`, resulting in a best-case complexity of `O(1)`. The average-case complexity will also depend on the distribution of the input - if generally the inputs are not empty or have properties early in the iteration order, the average complexity will be closer to `O(1)`.

Space Complexity

The space complexity of the `isEmpty` function is `O(1)` because it does not allocate any additional memory that grows with the input size; it only uses a fixed amount of memory to store the loop variable and the return value.