2631. Group By

Medium

Leetcode Link

Problem Description

The given problem requires us to augment the prototype of the built-in Array object in JavaScript with a new method called `groupBy`. This `groupBy` method should be callable on any array instance. When activated, it takes a callback function `fn` as a parameter. This function is applied to each element of the array, and the return value of the callback function is used as a key to group the elements.

The result of the `groupBy` method is an object where each property key corresponds to the output of the callback function `fn`, and the associated property value is an array of all elements from the original array which, when `fn` was called on them, returned that key.

To reiterate, the requirements specify:

- The `fn` function takes an array item as an argument and returns a string that is used as a key.
- The object produced contains keys for every unique string returned by `fn`.
- The values of the object are arrays of items for which the `fn` returned the corresponding key.
- The items within each group (array) should remain in the same order they appeared in the original array.
- The order of the keys in the grouped object does not matter.
- Use of external libraries like lodash for the groupBy functionality is not allowed.

Intuition

The intuition behind the solution is to process each item in the array sequentially, apply the `fn` function to get a key, and then organize the items into groups based on that key.

Here's a step-by-step approach to arrive at the solution:

1. **Extending the Array Prototype**: Since we need to enhance all arrays with the new `groupBy` method, we extend the Array prototype. This ensures that all array instances inherit this new method.

2. **Iterating Using `reduce` Method**: We use the `reduce` method because it traverses the array, and for each element, it applies a reducer function. The reducer function lets us accumulate a single result - in this case, the grouped object - from the individual elements of the array.

3. **Applying the Callback Function**: Within the reducer function, we invoke the provided callback `fn` on the current item to get the key. This helps determine the group to which the current item belongs.

4. **Building the Grouped Object**: We check if the grouped object already has a property with the key. If it does, we push the current item to the corresponding array. If not, we initialize a new array with the current item in it. This step essentially constructs the output object with keys as the unique strings returned by `fn` and values as arrays of grouped items.

5. **Returning the Accumulated Grouped Object**: After the reduction is complete, we return the accumulated object, which represents the grouped structure.

6. **Maintaining Order Within Groups**: Since `reduce` traverses the array from left to right and we are appending items to groups as we encounter them, the order of items within each group is the same as in the original array.

The provided code demonstrates a neat implementation of this approach without using any external library, complying with the problem's requirements.

Solution Approach

The solution provided involves extending the Array prototype and adding a new method `groupBy`. Here's a walkthrough of the solution's implementation details:

- **Extending Array Prototype**: This is done to add a new method `groupBy` that becomes available to all instances of arrays. It's a common pattern in JavaScript known as monkey patching, which should be used with caution as it could lead to clashes if the JavaScript environment introduces a method with the same name in the future or if used with third-party libraries.

```
declare global {
    interface Array<T> {
        groupBy(fn: (item: T) => string): Record<string, T[]>;
    }
}
```

Above, an interface declaration in TypeScript augments the global `Array` interface to include the `groupBy` method, ensuring that TypeScript is aware of the new method signature.

- **groupBy Method**: The `groupBy` method is defined as a function on `Array.prototype`. This function uses the `reduce` method internally to iterate over the array elements and group them based on the callback function.

```
Array.prototype.groupBy = function (fn) {
    return this.reduce((acc, item) => {
        // ...
    }, {});
};
```

- **The `reduce` Function**: The `reduce` method takes two parameters: a reducer function and an initial value for the accumulator. Here, the initial value of the accumulator (`acc`) is an empty object `{}` because the desired output is an object with keys based on the return values of `fn`.

```
    return this.reduce((acc, item) => {
        //...
    }, {});
```

- **Accumulating Groups**: Inside the `reduce` method, for each item in the array, we first determine the group key by applying the callback function `fn(item)`. Then, we check if the key already exists in the accumulator object. If it does, we push the current item onto the existing array. Otherwise, we create a new array with the current item.

```
const key = fn(item);
if (acc[key]) {
    acc[key].push(item);
} else {
    acc[key] = [item];
}
```

- **Final Output**: Once all items have been processed, the `reduce` method returns the accumulator object `acc`, which now contains the grouped contents.

In terms of algorithms and patterns, the solution heavily relies on:

- **Functional Programming Patterns**: Leveraging functions like `reduce` to build new data structures in an immutable fashion.
- **Duck Typing**: Assuming the function `fn` will act in a predictable manner, as per the specifications (returning a string when invoked with an array element).
- **Higher-Order Functions**: Using a function (`fn`) passed as an argument to another function (`groupBy`), to create a more specific and customized behavior.

And finally, to test the implemented method, the sample usage could be:

```
/**
 * [1,2,3].groupBy(String) // {"1":[1],"2":[2],"3":[3]}
 */
```

In this example, each number in the array `[1, 2, 3]` is converted to a string by the `String` constructor function, which is passed as `fn`, and the resulting keys are strings of the numbers with arrays that contain the original numbers grouped accordingly.

```
Array.prototype.groupBy = function (fn) {
    return this.reduce((acc, item) => {
        const key = fn(item);
        if (acc[key]) {
            acc[key].push(item);
        } else {
            acc[key] = [item];
        }
        return acc;
    }, {});
};
```

In summary, this code snippet aptly demonstrates how to inherently sort and group elements of an array into an object by a defined characteristic without relying on any external libraries.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach using the `groupBy` method:

Assume we have an array of objects representing different fruits with properties `type` and `color`:

```
const fruits = [
  { type: 'apple', color: 'green' },
  { type: 'banana', color: 'yellow' },
  { type: 'grape', color: 'green' },
  { type: 'apple', color: 'red' },
  { type: 'lemon', color: 'yellow' },
];
```

We want to group these fruits by their color. To do that, we will use the new `groupBy` method we've added to the Array prototype. We'll pass in a callback function that takes a fruit and returns its `color` which will serve as the group key.

```
const groupedByColor = fruits.groupBy(fruit => fruit.color);
```

Here's what happens step-by-step when calling `groupBy` with our example array:

1. The `groupBy` method is invoked on `fruits`.

2. The `reduce` method starts executing with an empty object `{}` as the initial accumulator.

3. The `reduce` callback runs for the first fruit:
   - It extracts the key 'green' using the provided callback (`fruit => fruit.color`).
   - Since 'green' doesn't exist on the accumulator yet, it creates a new property with this key and sets its value to an array containing the current fruit item `[ { type: 'apple', color: 'green' } ]`.
4. The `reduce` callback runs for the second fruit:
   - It extracts the key 'yellow'.
   - A new property 'yellow' is added to the accumulator with the current fruit item in a new array.
5. This process repeats for each fruit item, appending to the existing array if a key already exists or creating a new one if it doesn't.

After the `reduce` function has processed all fruits, the accumulator object looks like this:

```
{
  'green': [
    { type: 'apple', color: 'green' },
    { type: 'grape', color: 'green' }
  ],
  'yellow': [
    { type: 'banana', color: 'yellow' },
    { type: 'lemon', color: 'yellow' }
  ],
  'red': [
    { type: 'apple', color: 'red' }
  ]
}
```

The output is an object with color keys, each key pointing to an array of fruits that have that color. The `groupBy` feature works as intended, allowing us to flexibly group array items based on the characteristics we define.

Python Solution

```python
# Extend the list class with a new method 'group_by'.
class ExtendedList(list):
    # The group_by method takes a key_selector function that
    # determines the key by which to group list elements.
    def group_by(self, key_selector):
        # Use a dictionary to accumulate groups.
        # The keys are determined by the key_selector function,
        # and the values are lists of items.
        grouped = {}
        for item in self:
            # Obtain the key by passing the current item to the key_selector function.
            key = key_selector(item)
            # If the key does not exist in the dictionary, create a new list for it.
            # Then append the item to the list corresponding to the key.
            if key not in grouped:
                grouped[key] = []
            grouped[key].append(item)
        # Return the dictionary containing the grouped items.
        return grouped

# To use the ExtendedList class, it must be instantiated with a list of items.
# Example usage:
# This call should group numbers by their string representation.
# ExtendedList([1, 2, 3]).group_by(str) # {"1": [1], "2": [2], "3": [3]}

```

Java Solution

```java
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.function.Function;
import java.util.stream.Collectors;

public class ArrayUtils {

    /**
     * Groups elements of a list by a key generated from the elements themselves.
     *
     * @param <T> The type of elements in the list
     * @param <K> The type of the key used for grouping elements
     * @param list The list to be grouped
     * @param keySelector A function that generates keys from list elements
     * @return A map where each key is associated with a list of elements
     */
    public static <T, K> Map<K, List<T>> groupBy(List<T> list, Function<T, K> keySelector) {
        Map<K, List<T>> groupedMap = new HashMap<>();
        for (T element : list) {
            K key = keySelector.apply(element);
            // Get the current list of elements for this key, or initialize it if it does not exist
            if (!groupedMap.containsKey(key)) {
                groupedMap.put(key, new ArrayList<>());
            }
            groupedMap.get(key).add(element);
        }
        return groupedMap;
    }

    public static void main(String[] args) {
        // Example usage:
        // Should group numbers by their string representation.
        List<Integer> numbers = List.of(1, 2, 3);
        Map<String, List<Integer>> grouped = groupBy(numbers, String::valueOf);
        System.out.println(grouped); // Outputs: {"1"=[1], "2"=[2], "3"=[3]}
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <string>
#include <unordered_map>
#include <functional>

// Define a template function 'groupBy' that can be used with any vector of type T.
template<typename T>
std::unordered_map<std::string, std::vector<T>> groupBy(const std::vector<T>& items, std::function<std::string(const T&)> keySelector
    // The 'groupBy' function returns a map from strings to vectors of items.
    std::unordered_map<std::string, std::vector<T>> grouped;

    // Iterate through all items in the input vector 'items'.
    for (const T& item : items) {
        // Obtain the key for the current item using the provided 'keySelector' function.
        std::string key = keySelector(item);

        // Insert the item into the map under the appropriate key.
        // The map will automatically create a new vector if this is the first item for this key.
        grouped[key].push_back(item);
    }

    // Return the resulting map of grouped items.
    return grouped;
}

// Example usage:
// Create a lambda function that converts integers to their string representation.
auto intToString = [](const int& item) -> std::string {
    return std::to_string(item);
};

// Call 'groupBy' on a vector of integers using the lambda function to group by number as string.
std::vector<int> numbers = {1, 2, 3};
auto groupedNumbers = groupBy(numbers, intToString);
// groupedNumbers will be a map that contains {"1": [1], "2": [2], "3": [3]}

```

Typescript Solution

```typescript
// Extend the Array prototype with a new method 'groupBy'.
declare global {
  interface Array<T> {
    // The groupBy method is generic and takes a callback function that
    // determines the key by which to group array elements.
    groupBy(keySelector: (item: T) => string): Record<string, T[]>;
  }
}

// Implement the groupBy method for the Array prototype.
Array.prototype.groupBy = function <T>(keySelector: (item: T) => string): Record<string, T[]> {
  // Use the reduce method to accumulate groups. The accumulator 'grouped' is an object
  // whose keys are determined by the keySelector function and whose values are arrays of items.
  return this.reduce<Record<string, T[]>>((grouped, item) => {
    // Obtain the key by passing the current item to the keySelector function.
    const key = keySelector(item);
    // If the key already exists in the accumulator object, push the item into the existing array.
    if (grouped[key]) {
      grouped[key].push(item);
    } else {
      // If the key does not exist, create a new array with the item.
      grouped[key] = [item];
    }
    // Return the updated accumulator object for the next iteration.
    return grouped;
  }, {}); // Initialize the accumulator as an empty object.
};

// Example usage:
// This call should group numbers by their string representation.
// [1,2,3].groupBy(String) // {"1":[1],"2":[2],"3":[3]}

```

Time and Space Complexity

Time Complexity

The time complexity of the `groupBy` method is `O(n)`, where `n` is the number of elements in the array. This is because the method iterates over all elements in the array exactly once. Inside the iteration, the grouping function `fn` is called once for each element, and the key assignment along with the conditional check and array push operation can be considered to take constant time (`O(1)`). Therefore, the overall time complexity is determined by the loop, making it `O(n)`.

Space Complexity

The space complexity of the `groupBy` method is also `O(n)`, where `n` is the number of elements in the array. A new `Record<string, T[]>` object is created where the worst-case scenario involves every element being placed into its own unique key, requiring storage space for `n` keys and `n` singleton arrays. In the best case, all elements fall under one key, in which case there's one key with one array of size `n`. However, since the space required grows linearly with the input array size, it's considered `O(n)`.