Loop much?

fp function currying closure javascript typescript

So you have multiple operations that you want to perform on a list, you're a good lad *(or lass)* and you're not using a giant `for`

loop; you're feeling clever today and you'll be using functional operators like `map`, `filter`, and `reduce`.

But the list is relatively big, and even though it looks elegant, iterating on it 10 times is not exactly up for a performance award.

```
[1, 2, 3, ..., 100000]
  .map(it ⇒ it * 2)
  .filter(it ⇒ it % 3 == 0)
  .moreOps(...)
  .filter(...)
  .moreOps(...)
  .filter(...)
  .reduce((acc, it) ⇒ acc + it);
```

No, I'm not here to talk about the order of operators

It's important, but it's not always applicable. Instead, I want to discuss `transducers`, not the energy transformation devices, but the concept itself.

You want to have your cake and eat it too—use the operators but with a single iteration. It's quite an old problem, and so is its solution. Today we'll focus on just `filter`.

Closures, not the ones you know

▶ Let's take a look at `filter`

We can create an object that keeps track of each predicate used on an array, and this is actually a perfect solution for implementing the same concept in other operators.

Without changing `filter` implementation or introducing a new `filter` function, **we can use a simpler approach** for the sake of this post, and it's a very powerful concept in many other applications as well: `Closures`.

You might know that a `closure` is basically capturing the enclosing environment in the body of an inner environment/ function. **But this is not the `closure` we are talking about** here; Being able to manipulate/combine functions to produce another function with the same characteristics is also called a `closure` ¯_(ツ)_/¯

> “
>
> An operation for combining data objects satisfies the closure property if the results of combining things with that operation can themselves be combined using the same operation.
>
> —Structure & Interpretation of Computer Programs

Allow me to elaborate, what's a predicate?

```
type Predicate<T> = (item: T) ⇒ boolean;
```

A simple `boolean` value of `true` can be transformed to another `boolean` value of `false` by different types of manipulation. Can we do the same to a function like a predicate?

let us look at different ways to manipulate a `boolean`:

```
false && false; // false
false || false; // false
false && true; // false
false || true; // true
!false // true
```

And much more. Can we apply `and`, `or`, `not` to our predicates? Why of course we can! Why do you think I'm writing this? We just need to make the right representation of the effect of these operators:

```
const and =
  <T>(predicate1: Predicate<T>) =>
  (predicate2: Predicate<T>) =>
  (item: T) =>
    predicate1(item) && predicate2(item);
```

```
const or =
  <T>(predicate1: Predicate<T>) =>
  (predicate2: Predicate<T>) =>
  (item: T) =>
    predicate1(item) || predicate2(item);
```

```
const not =
  <T>(predicate: Predicate<T>) =>
  (item: T) =>
```

```
    !predicate(item);
```

It's utilising the same primitive `boolean` operations, and at the same time **combines** the predicates. Well, it combines their results. Which is what we care about.

Most importantly, calls to `and`, `or`, `not` also return a predicate! This allows to endlessly combine results of our combinations! This is the power of *(the other)* closures!

Why are we returning a function that accepts the other predicate instead of taking it as another parameter? Glad you asked!

Why did we do this again?

Because now that you can combine predicates together, you can have the same elegance of separating your filters while making a single call to `filter`:

```
const even = (n) => n % 2 ==0;

const productOfThree = (n) => n % 3 == 0;

const evenAndProductOfThree = and(even)(productOfThree);

[3, 6, 9, 12]
```

```
  .filter(evenAndProductOfThree); // [6, 12]
```

Or on the fly, since `evenAndProductOfThree` is also a predicate, we can reuse it

```
[3, 6, 9, 12]
  .filter(not(evenAndProductOfThree)); // [3, 9]
```

You can combine and accumulate as much predicates as you want, it'll all be run in a single iteration. But it also allows you to maintain the elegance of separating your predicates and combining them as building blocks to build the most suitable filter you need for your data.

What did you mention transducers for?

Well, it's true this is not exactly a transducer, rather an introduction to the other meaning of `closures` so that in the next post, transducers will be our focus in sha' Allah.

tech guide

Comments

Be the first to add a comment

LOG IN TO COMMENT