June 5th 2016

What are mutable and immutable data structures?

Sometimes concepts and ideas slowly grow in a programming community, sometimes they seem to appear in a flash. For the first several years I wrote JavaScript, I don't recall ever seeing anything written online about immutable data. Since React hit the scene in the last 2 years however, articles mentioning mutable and immutable data seem to have multiplied, as have libraries like Immutable.js and alternate front end languages like Elm that allow users to "use immutable data". I'm not going to address those libraries, but thought it would be useful to throw out a quick primer on what immutable data actually is, how it differs from mutable data, and why anyone cares.

A mutable object is an object whose state can be modified after it is created. An immutable object is an object whose state cannot be modified after it is created. Examples of native JavaScript values that are immutable are numbers and strings. Examples of native JavaScript values that are mutable include objects, arrays, functions, classes, sets, and maps.

Implications of mutable objects

So why does this matter? Consider the following code examples:

```
let a = {
    foo: 'bar'
};

let b = a;

a.foo = 'test';

console.log(b.foo); // test
console.log(a === b) // true
```

```
let a = 'test';
let b = a;
a = a.substring(2);

console.log(a) //st
console.log(b) //test
console.log(a === b) //false
```

```
let a = ['foo', 'bar'];
let b = a;

a.push('baz')

console.log(b); // ['foo', 'bar', 'baz']
console.log(a === b) // true
```

```
let a = 1;
let b = a;
a++;

console.log(a) //2
console.log(b) //1
console.log(a === b) //false
```

What we see is that for mutable values, updating state applies across all *references* to that variable. So changing a value in one place, changes it for all references to that object. For the immutable data types, we have no way of changing the internal state of the data, so the reference always gets reassigned to a new object. The biggest implication of this is that for immutable data, equality is more reliable since we know that a value's state won't be changed out from under us.

Finally, its worth noting that it's still possible to treat JavaScript objects as immutable. This can first be done through Object.freeze, which shallowly renders a JavaScript object immutable. But it can also be done with programmer discipline. If we want to rely on object's being immutable, it's possible to enforce that all object updates are done through something like `Object.assign(a, {foo: 'bar'})` rather than `a.foo = 'bar'`, and all array updates are done through functions that generate new arrays like `Array.prototype.map`, `Array.prototype.filter`, or `Array.prototype.concat`, rather than mutating methods like `Array.prototype.push`, `Array.prototye.pop`, or `Array.prototype.sort`. This is less reliable without language level constraints, but has become popular in the React ecosystem for dealing with data for folks who don't want to introduce abstractions like Immutable.js.

You Might Also Like These Articles

Lessons Backbone Developers Can Learn From React
A look at the lessons that Backbone developers can learn from React
Sep 9th 2015

Orthogonality and CSS in JS
Separation of concerns in the context of CSS and JavaScript
Jan 3rd 2017