Immutable Data

Last updated last month

CONTENTS

Table of Contents

What are the benefits of immutability?

Immutability can bring increased performance to your app, and leads to simpler programming and debugging, as data that never changes is easier to reason about than data that is free to be changed arbitrarily throughout your app.

In particular, immutability in the context of a Web app enables sophisticated change detection techniques to be implemented simply and cheaply, ensuring the computationally expensive process of updating the DOM occurs only when it absolutely has to (a cornerstone of React's performance improvements over other libraries).

Further information

Articles

- Introduction to Immutable.js and Functional Programming Concepts
- JavaScript Immutability presentation (PDF - see slide 12 for benefits)
- Immutable.js - Immutable Collections for JavaScript
- React: Optimizing Performance
- JavaScript Application Architecture On The Road To 2015

Why is immutability required by Redux?

- Both Redux and React-Redux employ shallow equality checking. In particular:
  - Redux's `combineReducers` utility shallowly checks for reference changes caused by the reducers that it calls.
  - React-Redux's `connect` method generates components that shallowly check reference changes to the root state, and the return values from the `mapStateToProps` function to see if the wrapped components actually need to re-render.
    Such shallow checking requires immutability to function correctly.
- Immutable data management ultimately makes data handling safer.
- Time-travel debugging requires that reducers be pure functions with no side effects, so that you can correctly jump between different states.

Further Information

Documentation

- Recipes: Prerequisite Reducer Concepts

Discussions

- Reddit: Why Redux Needs Reducers To Be Pure Functions

Why does Redux's use of shallow equality checking require immutability?

Redux's use of shallow equality checking requires immutability if any connected components are to be updated correctly. To see why, we need to understand the difference between shallow and deep equality checking in JavaScript.

How do shallow and deep equality checking differ?

Shallow equality checking (or *reference equality*) simply checks that two different *variables* reference the same object; in contrast, deep equality checking (or *value equality*) must check every *value* of two objects' properties.

A shallow equality check is therefore as simple (and as fast) as `a === b`, whereas a deep equality check involves a recursive traversal through the properties of two objects, comparing the value of each property at each step.

It's for this improvement in performance that Redux uses shallow equality checking.

Further Information

Articles

- Pros and Cons of using immutability with React.js

How does Redux use shallow equality checking?

Redux uses shallow equality checking in its `combineReducers` function to return either a new mutated copy of the root state object, or, if no mutations have been made, the current root state object.

Further Information

Documentation

- API: combineReducers

How does `combineReducers` use shallow equality checking?

The suggested structure for a Redux store is to split the state object into multiple "slices" or "domains" by key, and provide a separate reducer function to manage each individual data slice.

`combineReducers` makes working with this style of structure easier by taking a `reducers` argument that's defined as a hash table comprising a set of key/value pairs, where each key is the name of a state slice, and the corresponding value is the reducer function that will act on it.

So, for example, if your state shape is `{ todos, counter }`, the call to `combineReducers` would be:

```
combineReducers({ todos: myTodosReducer, counter: myCounterReducer })
```

where:

- the keys `todos` and `counter` each refer to a separate state slice;
- the values `myTodosReducer` and `myCounterReducer` are reducer functions, with each acting on the state slice identified by the respective key.

`combineReducers` iterates through each of these key/value pairs. For each iteration, it:

- creates a reference to the current state slice referred to by each key;
- calls the appropriate reducer and passes it the slice;
- creates a reference to the possibly-mutated state slice that's returned by the reducer.

As it continues through the iterations, `combineReducers` will construct a new state object with the state slices returned from each reducer. This new state object may or may not be different from the current state object. It is here that `combineReducers` uses shallow equality checking to determine whether the state has changed.

Specifically, at each stage of the iteration, `combineReducers` performs a shallow equality check on the current state slice and the state slice returned from the reducer. If the reducer returns a new object, the shallow equality check will fail, and `combineReducers` will set a `hasChanged` flag to true.

After the iterations have completed, `combineReducers` will check the state of the `hasChanged` flag. If it's true, the newly-constructed state object will be returned. If it's false, the *current* state object is returned.

This is worth emphasizing: *If the reducers all return the same* `state` *object passed to them, then* `combineReducers` *will return the current root state object, not the newly updated one.*

Further Information

Documentation

- API: combineReducers
- Redux FAQ - How do I share state between two reducers? do I have to use `combineReducers` ?

Video

- Egghead.io: Redux: Implementing combineReducers() from Scratch

How does React-Redux use shallow equality checking?

React-Redux uses shallow equality checking to determine whether the component it's wrapping needs to be re-rendered.

To do this, it assumes that the wrapped component is pure; that is, that the component will produce the same results given the same props and state.

By assuming the wrapped component is pure, it need only check whether the root state object or the values returned from `mapStateToProps` have changed. If they haven't, the wrapped component does not need re-rendering.

It detects a change by keeping a reference to the root state object, and a reference to *each value* in the props object that's returned from the `mapStateToProps` function.

It then runs a shallow equality check on its reference to the root state object and the state object passed to it, and a separate series of shallow checks on each reference to the props object's values and those that are returned from running the `mapStateToProps` function again.

Further Information

Documentation

- React-Redux Bindings

Articles

- API: React-Redux's connect function and `mapStateToProps`
- Troubleshooting: My views aren't updating when something changes outside of Redux

Why does React-Redux shallowly check each value within the props object returned from `mapStateToProp` ?

React-Redux performs a shallow equality check on each *value* within the props object, not on the props object itself.

It does so because the props object is actually a hash of prop names and their values (or selector functions that are used to retrieve or generate the values), such as in this example:

```
function mapStateToProps(state) {
  return {
    todos: state.todos, // prop value
    visibleTodos: getVisibleTodos(state) // selector
  }
}

export default connect(mapStateToProps)(TodoApp)
```

As such, a shallow equality check of the props object returned from repeated calls to `mapStateToProps` would always fail, as a new object would be returned each time.

React-Redux therefore maintains separate references to each *value* in the returned props object.

Further Information

Articles

- React.js pure render performance anti-pattern

How does React-Redux use shallow equality checking to determine whether a component needs re-rendering?

Each time React-Redux's `connect` function is called, it will perform a shallow equality check on its stored reference to the root state object, and the current root state object passed to it from the store. If the check passes, the root state object has not been updated, and so there is no need to re-render the component, or even call `mapStateToProps` .

If the check fails, however, the root state object *has* been updated, and so `connect` will call `mapStateToProps` to see if the props for the wrapped component have been updated.

It does this by performing a shallow equality check on each value within the object individually, and will only trigger a re-render if one of those checks fails.

In the example below, if `state.todos` and the value returned from `getVisibleTodos()` do not change on successive calls to `connect` , then the component will not re-render .

```
function mapStateToProps(state) {
  return {
    todos: state.todos, // prop value
    visibleTodos: getVisibleTodos(state) // selector
  }
}

export default connect(mapStateToProps)(TodoApp)
```

Conversely, in this next example (below), the component will *always* re-render, as the value of `todos` is always a new object, regardless of whether or not its values change:

```
// AVOID - will always cause a re-render
function mapStateToProps(state) {
  return {
    // todos always references a newly-created object
    todos: {
      all: state.todos,
      visibleTodos: getVisibleTodos(state)
    }
  }
}

export default connect(mapStateToProps)(TodoApp)
```

If the shallow equality check fails between the new values returned from `mapStateToProps` and the previous values that React-Redux kept a reference to, then a re-rendering of the component will be triggered.

Further Information

Articles

- Practical Redux, Part 6: Connected Lists, Forms, and Performance
- React.js Pure Render Performance Anti-Pattern
- High Performance Redux Apps

Discussions

- #1816: Component connected to state with `mapStateToProps`
- #300: Potential connect() optimization

Why will shallow equality checking not work with mutable objects?

Shallow equality checking cannot be used to detect if a function mutates an object passed into it if that object is mutable.

This is because two variables that reference the same object will *always* be equal, regardless of whether the object's values changes or not, as they're both referencing the same object. Thus, the following will always return true:

```
function mutateObj(obj) {
  obj.key = 'newValue'
  return obj
}

const param = { key: 'originalValue' }
const returnVal = mutateObj(param)

param === returnVal
//> true
```

The shallow check of `param` and `returnValue` simply checks whether both variables reference the same object, which they do. `mutateObj()` may return a mutated version of `obj`, but it's still the same object as that passed in. The fact that its values have been changed within `mutateObj` matters not at all to a shallow check.

Further Information

Articles

- Pros and Cons of using immutability with React.js

Does shallow equality checking with a mutable object cause problems with Redux?

Shallow equality checking with a mutable object will not cause problems with Redux, but it will cause problems with libraries that depend on the store, such as React-Redux.

Specifically, if the state slice passed to a reducer by `combineReducers` is a mutable object, the reducer can modify it directly and return it.

If it does, the shallow equality check that `combineReducers` performs will always pass, as the values of the state slice returned by the reducer may have been mutated, but the object itself has not - it's still the same object that was passed to the reducer.

Accordingly, `combineReducers` will not set its `hasChanged` flag, even though the state has changed. If none of the other reducers return a new, updated state slice, the `hasChanged` flag will remain set to false, causing `combineReducers` to return the *existing* root state object.

The store will still be updated with the new values for the root state, but because the root state object itself is still the same object, libraries that bind to Redux, such as React-Redux, will not be aware of the state's mutation, and so will not trigger a wrapped component's re-rendering.

Further Information

Documentation

- Recipes: Immutable Update Patterns
- Troubleshooting: Never mutate reducer arguments

Why does a reducer mutating the state prevent React-Redux from re-rendering a wrapped component?

If a Redux reducer directly mutates, and returns, the state object passed into it, the values of the root state object will change, but the object itself will not.

Because React-Redux performs a shallow check on the root state object to determine if its wrapped components need re-rendering or not, it will not be able to detect the state mutation, and so will not trigger a re-rendering.

Further Information

Documentation

- Troubleshooting: My views aren't updating when something changes outside of Redux

Why does a selector mutating and returning a persistent object to `mapStateToProps` prevent React-Redux from re-rendering a wrapped component?

If one of the values of the props object returned from `mapStateToProps` is an object that persists across calls to `connect` (such as, potentially, the root state object), yet is directly mutated and returned by a selector function, React-Redux will not be able to detect the mutation, and so will not trigger a re-render of the wrapped component.

As we've seen, the values in the mutable object returned by the selector function may have changed, but the object itself has not, and shallow equality checking only compares the objects themselves, not their values.

For example, the following `mapStateToProps` function will never trigger a re-render:

```
// State object held in the Redux store
const state = {
  user: {
    accessCount: 0,
    name: 'keith'
  }
}

// Selector function
const getUser = state => {
  ++state.user.accessCount // mutate the state object
  return state
}

// mapStateToProps
const mapStateToProps = state => ({
  // The object returned from getUser() is always
  // the same object, so this wrapped
  // component will never re-render, even though it's been
  // mutated
  userRecord: getUser(state)
})

const a = mapStateToProps(state)
const b = mapStateToProps(state)

a.userRecord === b.userRecord
//> true
```

Note that, conversely, if an *immutable* object is used, the component may re-render when it should not.

Further Information

Articles

- Practical Redux, Part 6: Connected Lists, Forms, and Performance

Discussions

- #1948: Is getMappedItems an anti-pattern in mapStateToProps?

How does immutability enable a shallow check to detect object mutations?

If an object is immutable, any changes that need to be made to it within a function must be made to a *copy* of the object.

This mutated copy is a *separate* object from that passed into the function, and so when it is returned, a shallow check will identify it as being a different object from that passed in, and so will fail.

Further Information

Articles

- Pros and Cons of using immutability with React.js

How can immutability in your reducers cause components to render unnecessarily?

You cannot mutate an immutable object; instead, you must mutate a copy of it, leaving the original intact.

That's perfectly OK when you mutate the copy, but in the context of a reducer, if you return a copy that *hasn't* been mutated, Redux's `combineReducers` function will still think that the state needs to be updated, as you're returning an entirely different object from the state slice object that was passed in.

`combineReducers` will then return this new root state object to the store. The new object will have the same values as the current root state object, but because it's a different object, it will cause the store to be updated, which will ultimately cause all connected components to be re-rendered unnecessarily.

To prevent this from happening, you must *always return the state slice object that's passed into a reducer if the reducer does not mutate the state.*

Further Information

Articles

- React.js pure render performance anti-pattern
- Building Efficient UI with React and Redux

How can immutability in `mapStateToProps` cause components to render unnecessarily?

Certain immutable operations, such as an Array filter, will always return a new object, even if the values themselves have not changed.

If such an operation is used as a selector function in `mapStateToProps`, the shallow equality check that React-Redux performs on each value in the props object that's returned will always fail, as the selector is returning a new object each time.

As such, even though the values of that new object have not changed, the wrapped component will always be re-rendered,

For example, the following will always trigger a re-render:

```
// A JavaScript array's 'filter' method treats the array as immutable,
// and returns a filtered copy of the array.
const getVisibleTodos = todos => todos.filter(t => !t.completed)

const state = {
  todos: [
    {
      text: 'do todo 1',
      completed: false
    },
    {
      text: 'do todo 2',
      completed: true
    }
  ]
}

const mapStateToProps = state => ({
  // getVisibleTodos() always returns a new array, and so the
  // 'visibleToDos' prop will always reference a different array,
  // causing the wrapped component to re-render, even if the array's
  // values haven't changed
  visibleToDos: getVisibleTodos(state.todos)
})

const a = mapStateToProps(state)
//  Call mapStateToProps(state) again with exactly the same arguments
const b = mapStateToProps(state)

a.visibleToDos
//> { "completed": false, "text": "do todo 1" }

b.visibleToDos
//> { "completed": false, "text": "do todo 1" }

a.visibleToDos === b.visibleToDos
//> false
```

Note that, conversely, if the values in your props object refer to mutable objects, your component may not render when it should.

Further Information

Articles

- React.js pure render performance anti-pattern
- Building Efficient UI with React and Redux
- ImmutableJS: worth the price?

What approaches are there for handling data immutability? Do I have to use Immutable.JS?

You do not need to use Immutable.JS with Redux. Plain JavaScript, if written correctly, is perfectly capable of providing immutability without having to use an immutable-focused library.

However, guaranteeing immutability with JavaScript is difficult, and it can be easy to mutate an object accidentally, causing bugs in your app that are extremely difficult to locate. For this reason, using an immutable update utility library such as Immutable.JS can significantly improve the reliability of your app, and make your app's development much easier.

Further Information

Discussions

- #1185: Question: Should I use immutable data structures?
- Introduction to Immutable.js and Functional Programming Concepts

---

What are the issues with using plain JavaScript for immutable operations?

JavaScript was never designed to provide guaranteed immutable operations. Accordingly, there are several issues you need to be aware of if you choose to use it for your immutable operations in your Redux app.

Accidental Object Mutation

With JavaScript, you can accidentally mutate an object (such as the Redux state tree) quite easily without realizing it. For example, updating deeply nested properties, creating a new *reference* to an object instead of a new object, or performing a shallow copy rather than a deep copy, can all lead to inadvertent object mutations, and can trip up even the most experienced JavaScript coder.

To avoid these issues, ensure you follow the recommended immutable update patterns for ES6.

Verbose Code

Updating complex nested state trees can lead to verbose code that is tedious to write and difficult to debug.

Poor Performance

Operating on JavaScript objects and arrays in an immutable way can be slow, particularly as your state tree grows larger.

Remember, to change an immutable object, you must mutate a *copy* of it, and copying large objects can be slow as every property must be copied.

In contrast, immutable libraries such as Immutable.JS can employ sophisticated optimization techniques such as structural sharing , which effectively returns a new object that reuses much of the existing object being copied from.

For copying very large objects, plain JavaScript can be over 100 times slower than an optimized immutable library.

Further Information

Documentation

- Immutable Update Patterns for ES6

Articles

- Immutable.js, persistent data structures and structural sharing
- A deep dive into Clojure's data structures
- Introduction to Immutable.js and Functional Programming Concepts
- JavaScript and Immutability
- Immutable Javascript using ES6 and beyond
- Pros and Cons of using immutability with React.js - React Kung Fu

Next Code Structure

Previous Actions