Immutable.js在React/Redux中的应用

目录

§ 1.几个场景

⊙ 场景一：关于数据不可变

例1：

```
 let data={key:"value"};
 func(data);
 console.log(data)//"猜猜会打印什么？"

//function func(data) {
    //data.key="data的key被改变了"
   //let data1=Object.assign(data,{name:"名字"})
//}
```

不查看func方法，不知道它对data做了什么，无法确认会打印什么。但如果data是Immutable，你可以确定打印的就是value

```
 let data=Immutable.Map({key:"value"});
 func(data);
 console.log(data.get("key"))//打印的是Value
```

例2：

```
    let obj1={a:1,b:2,c:{d:3}};
    let obj2=obj1;
    obj2.a=11;
    console.log(obj1.a)//输出的是11
```

而如果用Immutable.js的话

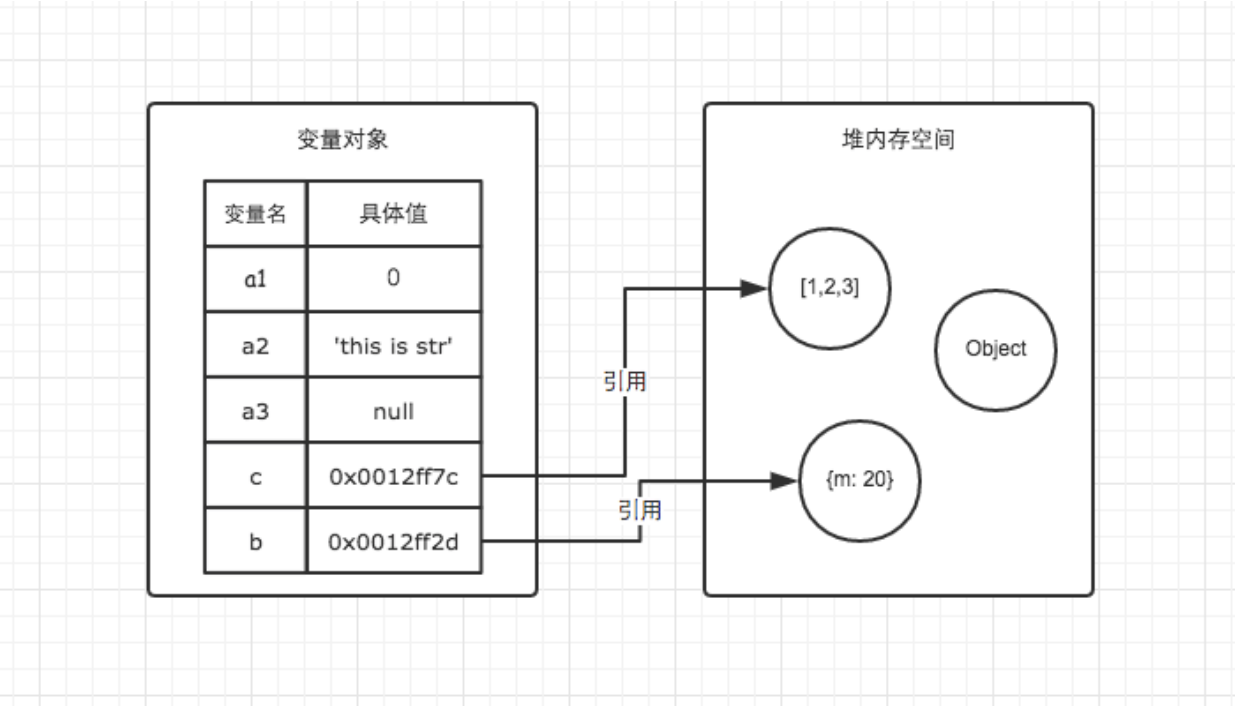
```
    let obj1=Immutable.fromJS({a:1,b:2,c:{d:3}});
    let obj2=obj1.merge(obj1);
    obj2=obj2.updateIn(["c","d"],()=>300);
    console.log(obj1.toJS(),obj2.toJS())//{a:1,b:2,c:{d:3}}, {a:100,b:2,c:
{d:300}}
```

§ 观后感：

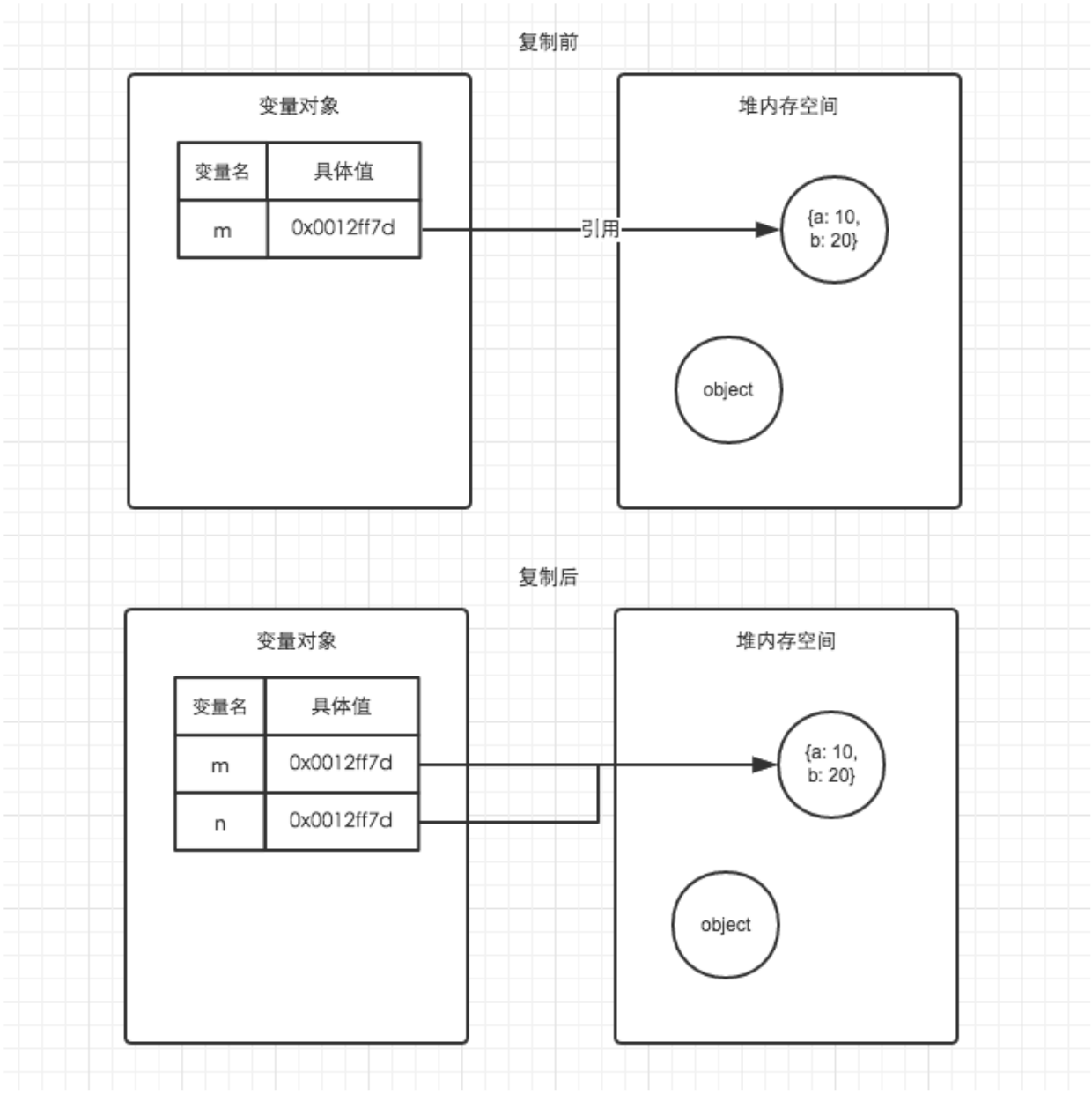
~问？为什么会出现上面这种情况
~答：在JS中的引用数据类型（如Object，Array等）使用的是引用赋值，如果新的对象简单的引用了原始对象，改变新的对象也将影响旧的；

> 与其他语言不通，JS引用数据类型（比如Object，Array）的值是保存在堆内存中的对象。JavaScript不允许直接访问堆内存中的位置，因此我们不能直接操作对象的堆内存空间。在操作对象时，实际上是在操作对象的引用而不是实际的对象（当复制保存着对象的某个变量时，操作的是对象的引用。但在为对象添加属性时，操作的是实际的对象）。因此，引用类型的值都是按引用访问的。这里的引用，我们可以粗浅地理解为保存在变量对象中的一个地址，该地址与堆内存的实际值相关联。当我们要访问堆内存中的引用数据类型时，

> 实际上我们首先是从变量对象中获取了该对象的地址引用（或者地址指针），然后再从堆内存中取得我们需要的数据。



- let m={a:10,b:20}在进行”=”赋值的时候是把存放于栈里面的标识符obj1通过引用指针指向了存放于堆里面的{a:10,b:2};
- 当let n=m的时候只是赋予n一个新的内存地址，但这个新的内存地址指向的还是存放于堆里面的同一个{a:10,b:2};
- 所以当更改n的a属性的时候（n.a=100），输出的m的a属性也会发生变化



JS中的引用数据类型自然也有优点，优点在于频繁的操作数据都是在原对象的基础上修改，不会创建新对象，从而可以有效的利用内存，不会浪费内存，这种特性称为**mutable**（可变），但恰恰它的优点也是它的缺点，太过于灵活多变在复杂数据的场景下也造成了它的不可控性，假设一个对象在多处用到，在某一处不小心修改了数据，其他地方很难预见到数据是如何改变的

~提出解决方案： 针对上面这些情况，会想着数据要是不可变就好了（数据b来源(Copy)于数据b，但a或者b的操作互不影响，数据可控）。（在“场景二”里面再结合补充“引用数据类型赋值(=)和浅拷贝的区别”）

- 一：会想到es6的`object.assign()`或者Rest参数(...)；`let obj1={a:1};let obj2=object.assign({},obj1);console.log(obj1.a)//这儿输出的是1`，咋看之下以为`object.assign()`是深拷贝,但其实`object.assign()`属于伪深拷贝；`Rest参数`和`object.assign()`一样属于伪深拷贝；

> Object.assign()属于伪深拷贝（第一层的深拷贝，嵌套层的浅拷贝）

```
let obj1={
    a:1,
```

```
    b:2,
    c:{
        d:3
    }
   };
 let obj2=Object.assign({},obj1);
 obj2.a=11;
 console.log(obj1.a); //输出的还是1，第一层深拷贝
```

```
let obj1={
    a:1,
    b:2,
    c:{
        d:3
    }
   };
 let obj2=Object.assign({},obj1);
 obj2.c={f:4};
 console.log(obj1.c); //输出的还是{d:3}，第一层深拷贝
```

```
let obj1={
  a:1,
  b:2,
  c:{
      d:3
     }
   };
 let obj2=Object.assign({},obj1);
 obj2.c.d=44;
 console.log(obj1.c.d); //输出的是44而不是3，嵌套层是浅拷贝
```

```
let obj1={
    a:1,
    b:2,
    c:{
        d:3
    }
};
let obj2=Object.assign({},obj1);
obj2.a=11;
obj2.c.d=44;
console.log(obj1,obj2); //输出的是{a:1,b:2,c:{d:44}}，{a:11,b:2,c:{d:44}}
```

```
let m={a:1,b:2,c:{d:3}};
let n={...m};
n.c.d=300;
```

```
console.log(m.c.d) //输出的是300
```

- 二：会想到用深拷贝，深拷贝确实是能解决上面这些情况，然而深拷贝的几种方法：大致原理为栈里新的标志符通过新的内存地址对存放于堆里的对象(引用数据类型的值)循环遍历后在堆里**重新给开辟的一块儿内存**进行引用；

~抛出设想？：有没有既能保持数据的不可变，但又能避免使用 deepCopy（深拷贝）而造成的 CPU 和内存的浪费（*虽然大概知道了可以用Immutable，姑且保留这个设想在这儿，后面会讲为啥可以用Immutable*）？

⊙ 场景二：关于”===”比较

> 在不少时候，我们其实需要比较两个对象是否”相等”的（{a:1,b:2}==={a:1,b:2}），比如在React中我们可以在组件shouldComponentUpdate方法中在接收到新的props 或者 state，比较新旧props或者state的值是否相同而对组件是否进行重新渲染进行控制，其实这也是React性能优化的关键

例：

```
{a:1,b:2,c:3}==={a:1,b:2,c:3}; // false
[1,2,3] === [1,2,3]; // false
```

就如上面这样的例子，我们其实是想着左右完全相同，想返回的是一个true；至于原因，对于JS引用数据类型的===比较，比较的是**其引用是否指向同一个对象**；

```
let obj1={a:1};
let obj2=obj1;
let obj3={a:1};
obj2.a=100;
console.log(obj1===obj2);//输出的是true（obj1和pbj2指向于堆中的是同一个对象，所以为true）
console.log(obj1===obj3);//输出的是false（obj1和pbj2指向于堆中的不是同一个对象，所以为false）
```

~提出解决方案：对于上面采用deepCopy、deepCompare来比较，但其实这类比较的大致原理，是通过对要比较的两个对象（或者数组）进行层层循环，进而对各个值进行”===”比较；但这样比较麻烦而且耗性能（相对Immutable而言，后面会讲为啥）

```
//如只考虑对象只有一层（有多层的话，无外乎也是多了层判断，然后还是用同样的方式进行===比较）
function isEqual(a,b) {
        let result=true;
        for (let key in a){
            if(!b[key] || a[key]!==b[key]){
                result=false
```

```
            }
        }
        return result
    }
let obj1={a:1,b:2,c:3};
let obj2={a:1,b:2,c:3};
console.log(isEqual(obj1,obj2)) //输出的是true
```

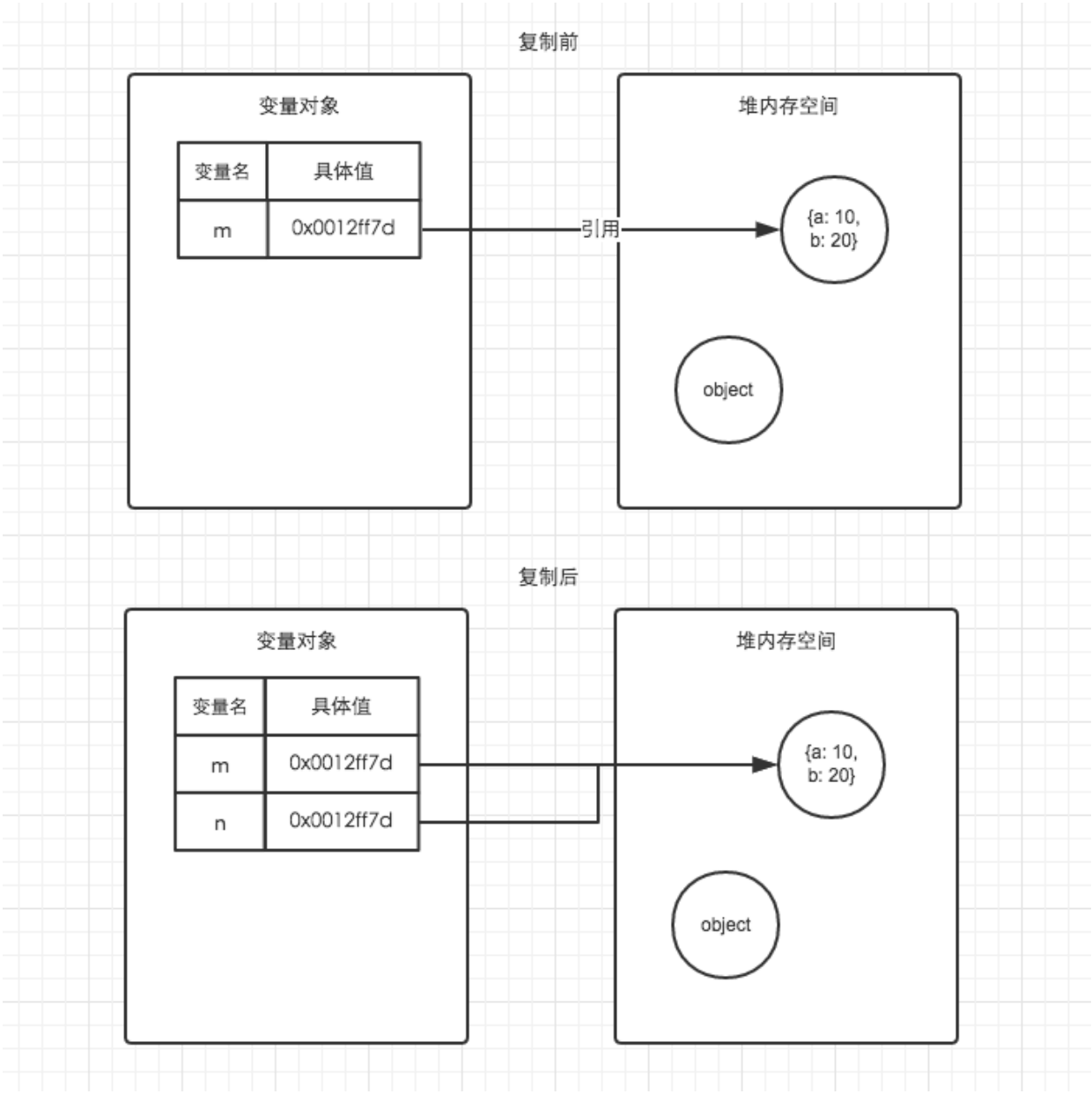
暂且先不论这儿为啥采用immutable.js，用immutable.js的话，

```
let m=Immutable.fromJS({a:1,b:2,c:{d:3}});
let n=Immutable.fromJS({a:1,b:2,c:{d:3}});
console.log(Immutable.is(m,n)); //输出true（通过hashCode比较值）
console.log(m===n); //输出false（m和n指向于堆里的对象不是同一个，所以为false）
```

~在这儿补充说下：引用数据类型赋值(=)，浅拷贝和深拷贝的区别：

- 引用数据类型赋值(=)操作（let m={a:10,b:20};let n=m;）是在栈里面重新给分配了一个内存地址，然后这个内存地址指向于堆里面的对象其实还是是同一个！

```
let m={a:10,b:20};
let n=m;
n.b=30;
console.log(m===n);//输出的是true（m和n指向于堆里的对象是同一个，所以为true）
```



- 浅拷贝是只复制对象（比如有let m={a:10,b:{c:20}}）的第一层，第一层的操作互不影响，但嵌套层的引用数据类型指向的还是于堆中的同一个对象，还是会相互影响，像Object.assign()这类方法如果非要严格的算浅拷贝还是深拷贝，其属于浅拷贝。

  Object.freeze 和 ES6 中新加入的 const 都可以达到防止对象被篡改的功能，但它们是shallowCopy （浅拷贝）的，对象层级一深就要特殊处理了。

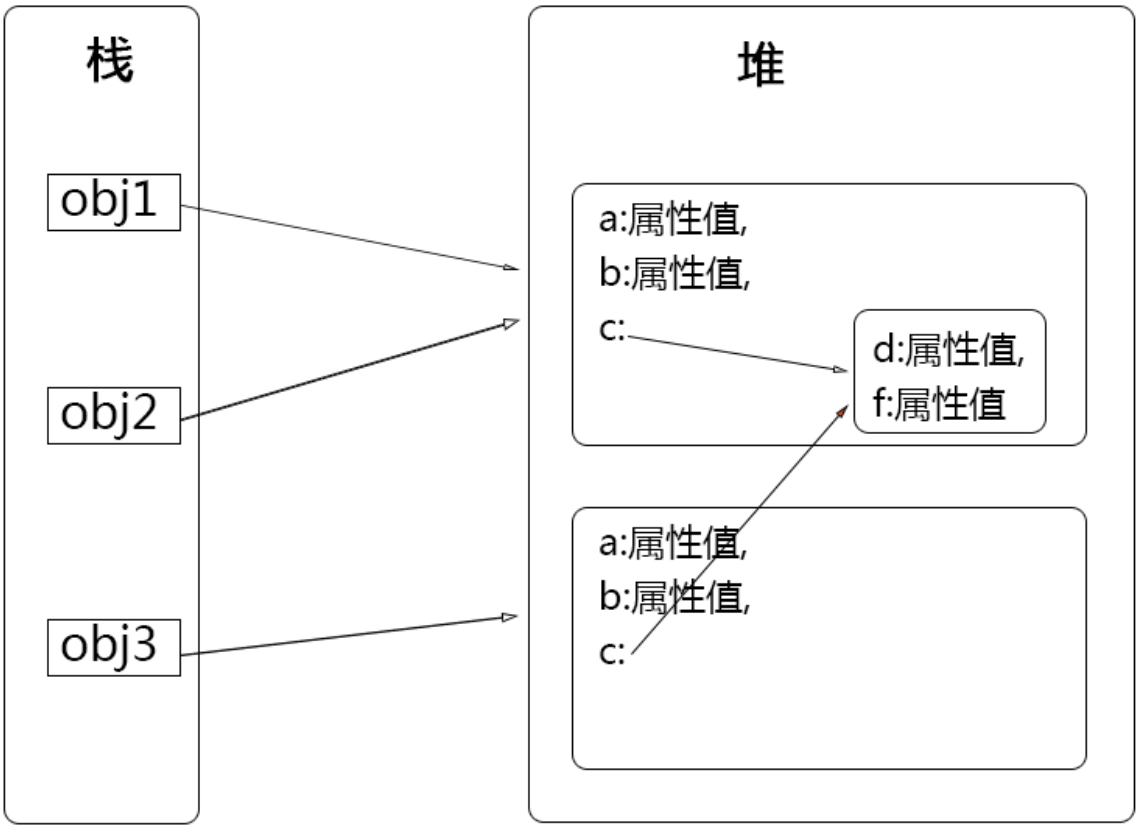
  而像上面的这个引用数据类型赋值(=)充其量能算是“引用”，而不是真正的浅拷贝。而浅拷贝之于引用数据类型赋值(=)的不同在于，浅拷贝的得到的对象值是在堆里面给重新生成了一个对象，前后两个对象引用的并不是同一个对象，两相比较的话，输出的会是false;

例1：

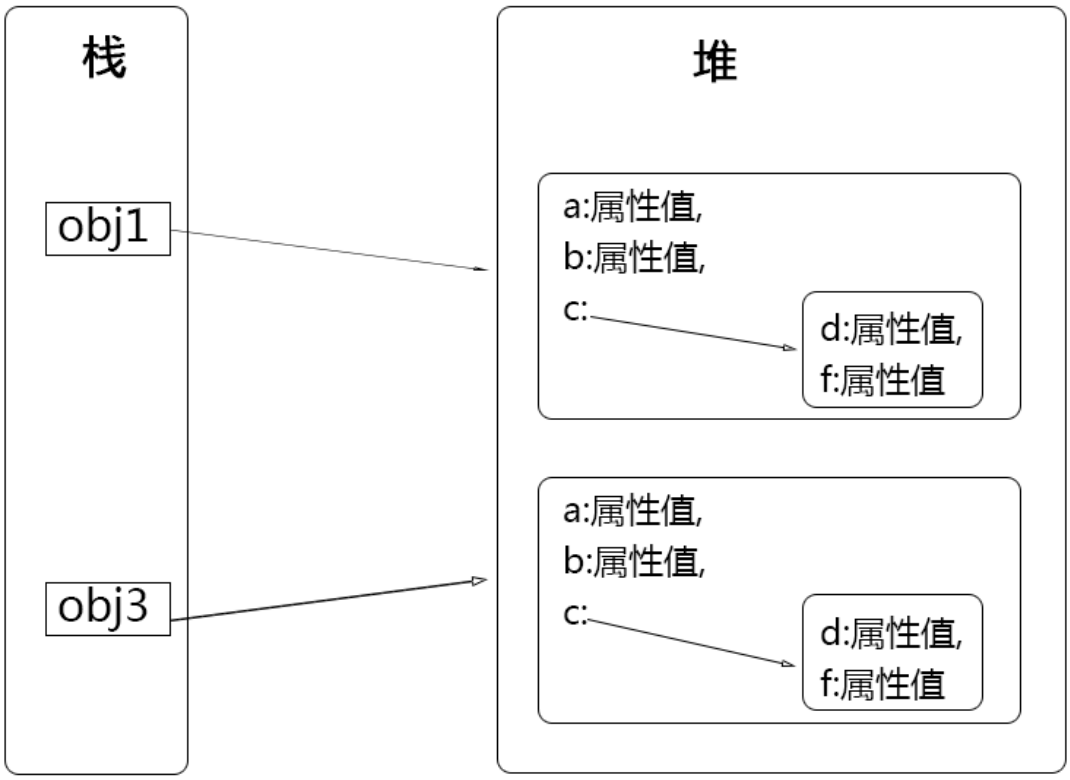
```
let m={a: 10, b: 20};
let n=Object.assign({}, m);
console.log(m===n); //输出的是false（m和n指向于堆里的对象不是同一个，所以为fals
e）
```

例2：

```
function shallowCopy(src){
    let target={};
    for (let key in src){
        if(src.hasOwnProperty(key)){
            target[key]=src[key];
        }
    }
    return target;
}

let obj1={
    a:10,
    b:20,
    c:{
        d:30,
        f:40
    }
};
let obj2=obj1;
let obj3=shallowCopy(obj1);

obj2.a=1000;
obj3.b=2000;

obj2.c.d=3000;
obj3.c.f=4000;

console.log(obj1); //{1000,20,c:{d:3000,e:4000}}
console.log(obj2); //{1000,20,c:{d:3000,e:4000}}
console.log(obj3); //{10,2000,c:{d:3000,e:4000}}
```



- 深拷贝是对对象以及对象的所有子对象进行拷贝，不管是嵌套层与否，指向于堆中的对象并不是同一个



§ 2.什么是 Immutable Data

> Immutable Data 就是一旦创建，就不能再被更改的数据。对 Immutable 对象的任何修改或添加删除操作都会返回一个新的 Immutable 对象而不会对旧对象产生任何影响。

目前流行的 Immutable 库有两个：

1、immutable.js

Immutable.js本质上是一个JavaScript的持久化数据结构的库 ，但是由于同期的React太火，并且和React在性能优化方面天衣无缝的配合，导致大家常常把它们两者绑定在一起。

Facebook 工程师 Lee Byron 花费 3 年时间打造，与 React 同期出现，但没有被默认放到 React 工具集里（React 提供了简化的 Helper）。它从头开始实现了完全的 `Persistent Data Structure（持久化数据结构）`，通过使用Trie 数据结构这样的先进技术来实现 `Structural Sharing（结构共享）`。所有的更新操作都会返回新的值，但是在内部结构是共享的，来减少内存占用(和垃圾回收的失效)，且数据结构和方法非常丰富（完全不像JS出身的好不好）。像Collection、List、Map、Set、Record、Seq。有非常全面的map、filter、groupBy、reduce``find函数式操作方法。同时 API 也尽量与 Object 或 Array 类似。

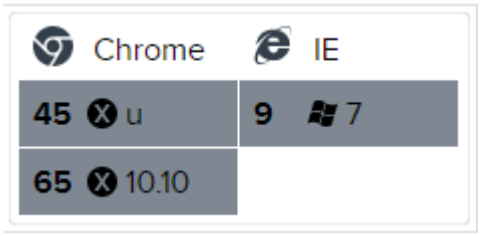
其中有 3 种最重要的数据结构说明一下：

- Map：键值对集合，对应于 Object，ES6 也有专门的 Map 对象；
- List：有序可重复的列表，对应于 Array
- Set：无序且不可重复的列表

2、seamless-immutable

与`Immutable.js`学院派的风格不同，`seamless-immutable`并没有实现完整的`Persistent Data Structure（持久化数据结构）`，而是使用 Object.defineProperty（因此只能在 IE9 及以上使用）扩展了 JavaScript 的 Array 和 Object 对象来实现，只支持 Array 和 Object 两种数据类型，API 基于与 Array 和 Object 操持不变。代码库非常小，压缩后下载只有 2K。而 Immutable.js 压缩后下载有 16K;

seamless-immutable的实现依赖于ECMAScript 5 的一些特性，如Object.defineProperty 和Object.freeze，因此会在浏览器兼容性方面有所欠缺：



不过这不是问题啦，可以使用 `polyfill es-shims/es5-shim` 来解决。

下面上代码来感受一下两者的不同：

```
// 原来的写法
let obj1={a: {b: 1}};
```

```
let obj2=obj1;
obj2.a.b=2;
console.log(obj1.a.b); //输出的是2
console.log(obj1===obj2); //输出的是true（obj1和obj2指向于堆里的对象是同一个，所以为true）

// 使用 immutable.js 后
import Immutable from "immutable";
let obj1=Immutable.fromJS({a:{b:1}});
let obj2=obj1.setIn(["a","b"],2); // 使用 setIn 赋值
console.log(obj1.getIn(["a","b"])); //使用 getIn 取值，输出的是1
console.log(Immutable.is(obj1,obj2)); //输出false（通过hashCode比较键值）
console.log(obj1===obj2);  //输出false（obj1和obj2指向于堆里的对象不是同一个，所以为false）

// 使用  seamless-immutable.js 后
import SImmutable from "seamless-immutable";
let obj1=SImmutable({a:{b:1}});
let obj2=obj1.merge({a:{b:2}}); // 使用 merge 赋值
console.log(obj1.a.b);  //像原生Object一样取值，输出的是1
console.log(obj1===obj2);  //输出false（obj1和obj2指向于堆里的对象不是同一个，所以为false）
```

§ 3.Immutable.js介绍

§ 3.1 Immutable.js 主要的三大特性

- 1、Persistent data structure （持久化数据结构）
- 2、structural sharing （结构共享）
- 3、support lazy operation （惰性操作）

下面我们来具体介绍下这三个特性：

1. Persistent data structure （持久化数据结构）

一般听到持久化，在编程中第一反应应该是，数据存在某个地方，需要用到的时候就能从这个地方拿出来直接使用。

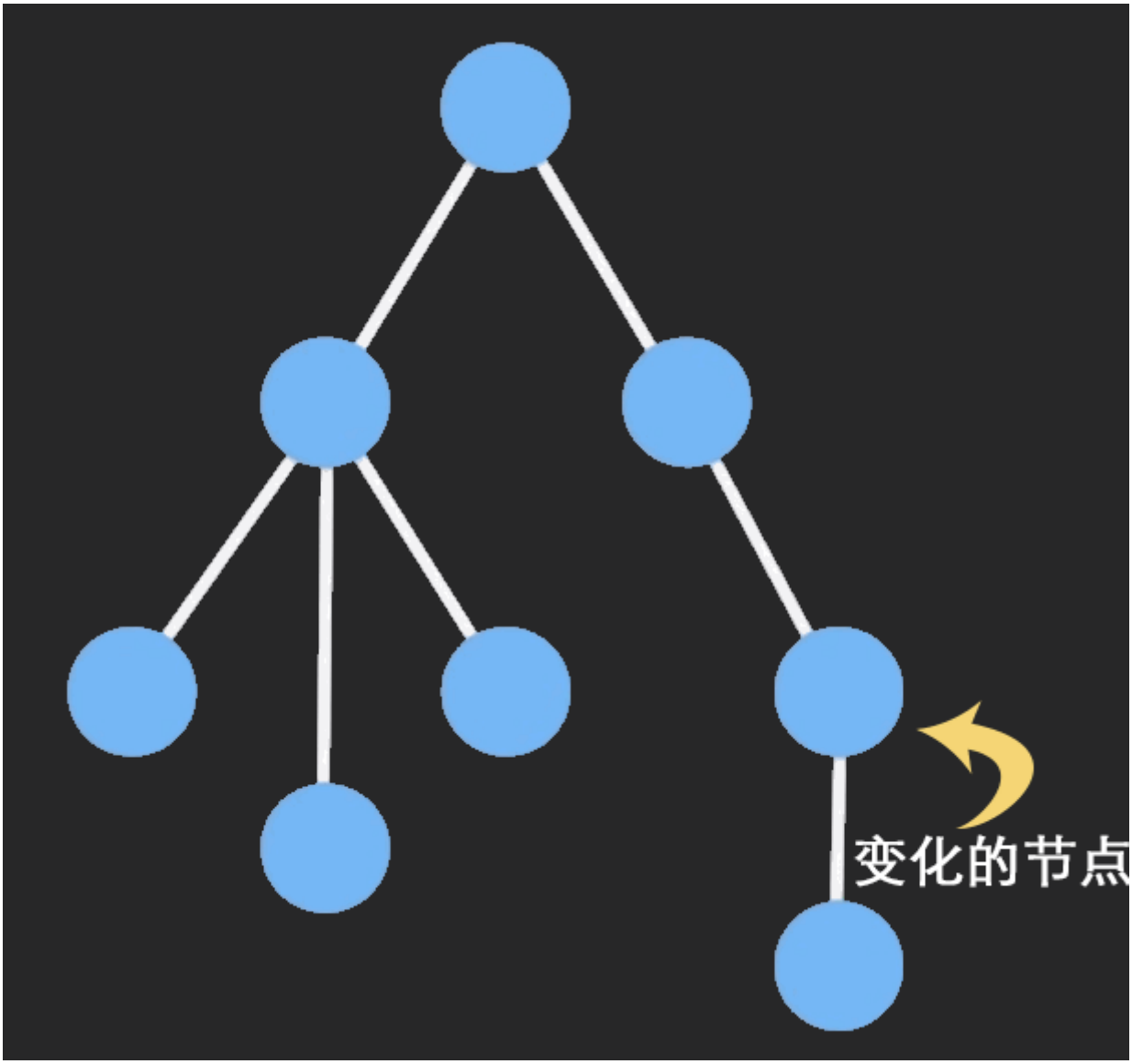
但这里说的持久化是另一个意思，用来描述一种数据结构，一般函数式编程中非常常见，指一个数据，在被修改时，仍然能够保持修改前的状态，从本质来说，这种数据类型就是不可变类型，也就是`immutable`

immutable.js提供了十余种不可变的类型（List，Map，Set，Seq，Collection，Range等）

到这，有些可能会有疑问，这和深拷贝有什么区别？也是每次都创建一个新对象，开销一样很大。OK，那接下来第二个特性会为你揭开疑惑。

2. structural sharing （结构共享）

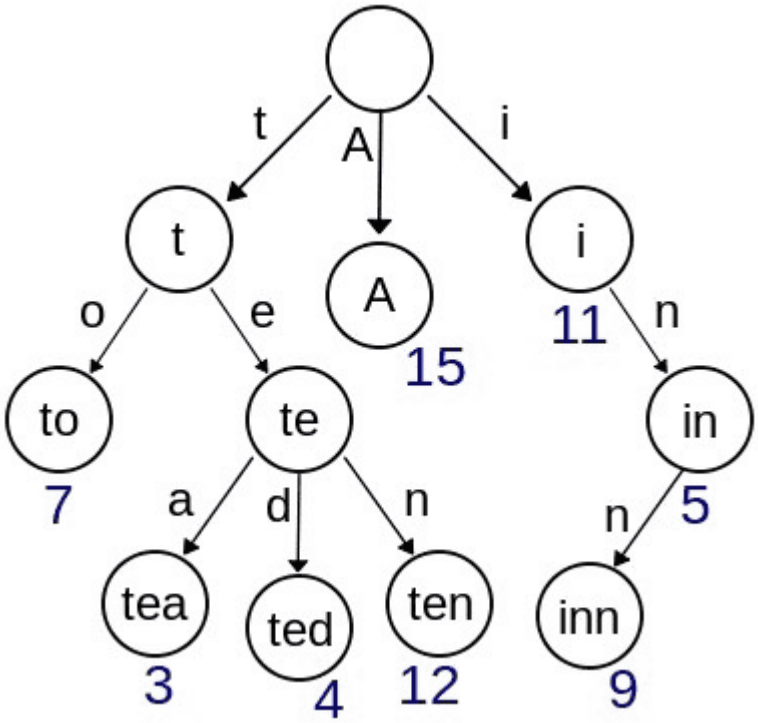
structural sharing （结构共享）即如果对象树中一个节点发生变化，只修改这个节点和受它影响的父节点，其它节点则进行共享；
请看下面动画：



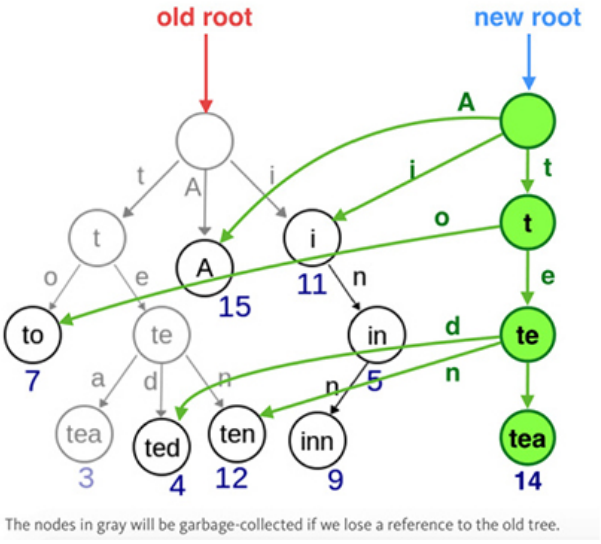
补充：

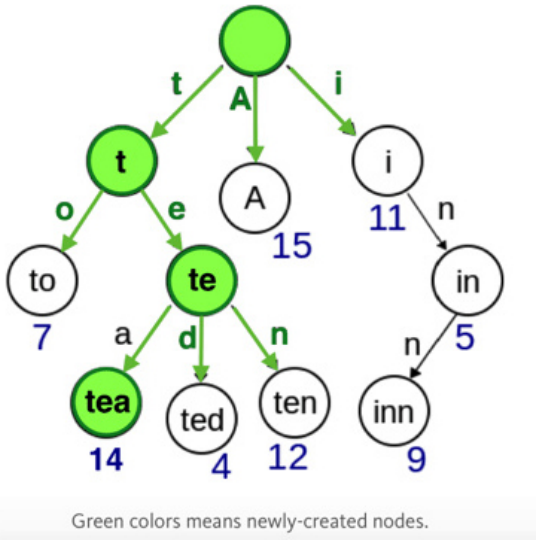
```
let data={to:7,tea:3,ted:4,ten:12,A:15,i:11,in:5,inn:9}
```

根据trie结构,存储的结构类似于



如果更改了了tea字段 3 为 14,那么只需改变四个节点,来更新形成新的树, 这就是结构共享。





而正是因为Immutable.js通过使用Trie 数据结构这样的先进技术实现了`Structural Sharing（结构共享）`，继而带了两大好处：

2.1：节省内存，避免CPU和内存的浪费

> 此好处回应解答上面`场景一`所为啥采用immutable.js

immutable.js 使用了Structure Sharing 会尽量复用内存（努力避免创建新的对象），甚至以前使用的对象也可以再次被复用，没有被引用的对象会被垃圾回收。

例1：

```
let obj1=Immutable.fromJS({
        a:1,
```

```
            b:{
                c:3
            }
        });
let obj2=obj1.set("a",100);
console.log(obj1===obj2);//输出的是false（obj1和obj2指向于堆里的对象并不是同一个，所以为false）
console.log(obj1.get("b")===obj2.get("b"));//输出的是true（obj1.get("b")和obj2.get("b")指向于堆里的对象是同一个，共享了没有变化的b节点，所以为true）
```

例2：

```
let obj1=Immutable.fromJS({
            a:1,
            b:{
                c:3
            }
        });
let obj2=obj1.set("a",100);
let obj3=obj1.set("a",1);
console.log(obj1===obj2);//输出的是false（obj1和obj2指向于堆里的对象并不是同一个，所以为false）
console.log(obj1===obj3);//输出的是true（虽然obj3进行了一顿操作，然而数据并没有改变,避免创建了新对象，复用了内存，所以obj1和obj3指向于堆里的对象还是同一个，所以为true）
```

2.1：性能的提升

> 此好处回应解答上面 场景二 对应的为啥用immutable.js

两个Immutable 对象可以使用”===” 来比较，这样是直接比较两个Immutable 对象是否指向于堆里的同一个对象，性能最好。但即使两个对象的值是一样的，也会返回 false：

```
let obj1=Immutable.Map({a:1,b:1,c:1});
let obj2=Immutable.Map({a:1,b:1,c:1});
console.log(obj1===obj2); //输出的是false（obj1和obj2指向于堆里的对象并不是同一个，所以为false）
```

为了直接比较对象的值，immutable.js 提供了 Immutable.is() 来做『值比较』。

Immutable.is() 比较的是两个对象的 hashCode 或 valueOf（对于 JavaScript 对象）。由于 immutable.js 内部使用了 Trie 数据结构来存储，只要两个对象的 hashCode 相等，值就是一样的。这样的算法避免了深度遍历比较，性能非常好。

结合上面2点好处，来看下下面这个例子，以更好的理解structural sharing （结构共享）

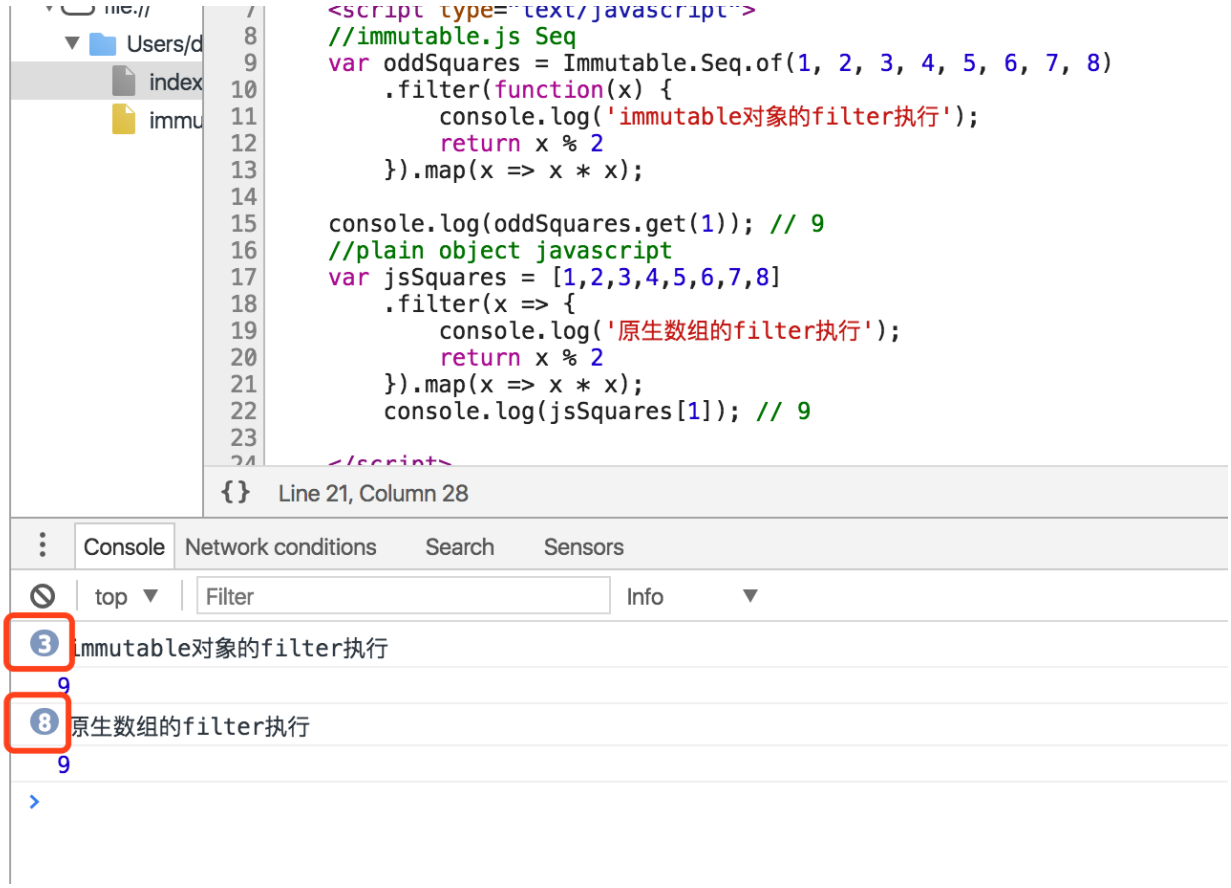
```
let obj1=Immutable.fromJS({
```

```
            a:1,
            b:{
                c:3
            }
        });
let obj2=obj1.set("a",100);
let obj3=Immutable.fromJS({
    a:1,
    b:{
        c:3
    }
});
console.log(obj1===obj2); //输出的是false（bj1和obj2指向于堆里的对象并不是同一个，所以为false）
console.log(obj1===obj3); //输出的是false（bj1和obj2指向于堆里的对象并不是同一个，所以为false）
console.log(obj1.get("b")===obj2.get("b")); //输出的是true（obj1.get("b")和obj2.get("b")指向于堆里的对象是同一个，共享了没有变化的b节点，所以为true）
console.log(obj1.get("b")===obj3.get("b")); //输出的是false（obj1.get("b")和obj3.get("b")指向于堆里的对象不是同一个，所以为false）；
//而如果我只想比较两个Immutable对象的键值是否一样，可以用Immutable.is()方法
console.log(Immutable.is(obj1,obj3)); //输出的是true(通过比较obj1和obj3的hash Code值一样进而比较出来值也一样)
```

3.support lazy operation （惰性操作）

- 惰性操作 Seq
- 特征1：Immutable (不可变)
- 特征2：lazy（惰性，延迟）

这个特性非常的有趣，这里的lazy指的是什么？很难用语言来描述，我们看一个demo，看完你就明白了



这段代码的意思就是，数组先取奇数，然后再对基数进行平方操作，然后在console.log第2个数，同样的代码，用immutable的seq对象来实现，filter只执行了3次，但原生执行了8次。

其实原理就是，用seq创建的对象，其实代码块没有被执行，只是被声明了，代码在get(1)的时候才会实际被执行，取到index=1的数之后，后面的就不会再执行了，所以在filter时，第三次就取到了要的数，从4-8都不会再执行。

想想，如果在实际业务中，数据量非常大，如在我们点餐业务中，商户的菜单列表可能有几百道菜，一个array的长度是几百，要操作这样一个array，如果应用惰性操作的特性，会节省非常多的性能

§ 3.2 Immutable.js 优点

1. Immutable 降低了 Mutable 带来的复杂度

参考”场景一”例1

2. 节省内存，性能提升

参考”Immutable.js 主要的三大特性之结构共享”

3. Undo/Redo，Copy/Paste，甚至时间旅行这些功能做起来小菜一碟

因为每次数据都是不一样的，只要把这些数据放到一个数组里储存起来，想回退到哪里就拿出对应数据即可，很容易开发出撤销重做这种功能。

4. 并发安全

传统的并发非常难做，因为要处理各种数据不一致问题，因此『聪明人』发明了各种锁来解决。但使用了 Immutable 之后，数据天生是不可变的，并发锁就不需要了。

然而现在并没什么卵用，因为 JavaScript 还是单线程运行的啊。但未来可能会加入，提前解决未来的问题不也挺好吗？

5. 拥抱函数式编程

Immutable 本身就是函数式编程中的概念，纯函数式编程比面向对象更适用于前端开发。因为只要输入一致，输出必然一致，这样开发的组件更易于调试和组装。

像 ClojureScript，Elm 等函数式编程语言中的数据类型天生都是 Immutable 的，这也是为什么 ClojureScript 基于 React 的框架 — Om 性能比 React 还要好的原因。

§ 3.3 Immutable.js 缺点

1. 需要学习新的 API

2. 增加了资源文件大小

3.容易与原生对象混淆

这点是我们使用 Immutable.js 过程中遇到最大的问题。写代码要做思维上的转变。

虽然 Immutable.js 尽量尝试把 API 设计的原生对象类似，有的时候还是很难区别到底是 Immutable 对象还是原生对象，容易混淆操作。

Immutable 中的 Map 和 List 虽对应原生 Object 和 Array，但操作非常不同，比如你要用 map.get('key') 而不是 map.key，array.get(0) 而不是 array[0]。另外 Immutable 每次修改都会返回新对象，也很容易忘记赋值。

当使用外部库的时候，一般需要使用原生对象，也很容易忘记转换。

下面给出一些办法来避免类似问题发生：

- 1.使用 Flow 或 TypeScript 这类有静态类型检查的工具;
- 2.约定变量命名规则：如所有 Immutable 类型对象以 $$ 开头;
- 3.使用 Immutable.fromJS 而不是 Immutable.Map 或 Immutable.List 来创建对象，这样可以避免 Immutable 和原生对象间的混用。

```
//使用 Immutable.fromJS
let obj=Immutable.fromJS({
    a:1,
    b:2,
    c:{
        d:3
    }
});
let getVal=obj.get("c");
```

```
console.log(getVal) //getVal是Immutable对象

//使用 Immutable.Map
let obj2=Immutable.Map({
    a:1,
    b:2,
    c:{
        d:3
    }
});
let getVal=obj2.get("c");
console.log(getVal) //getVal是JS原生对象{d:3}
```

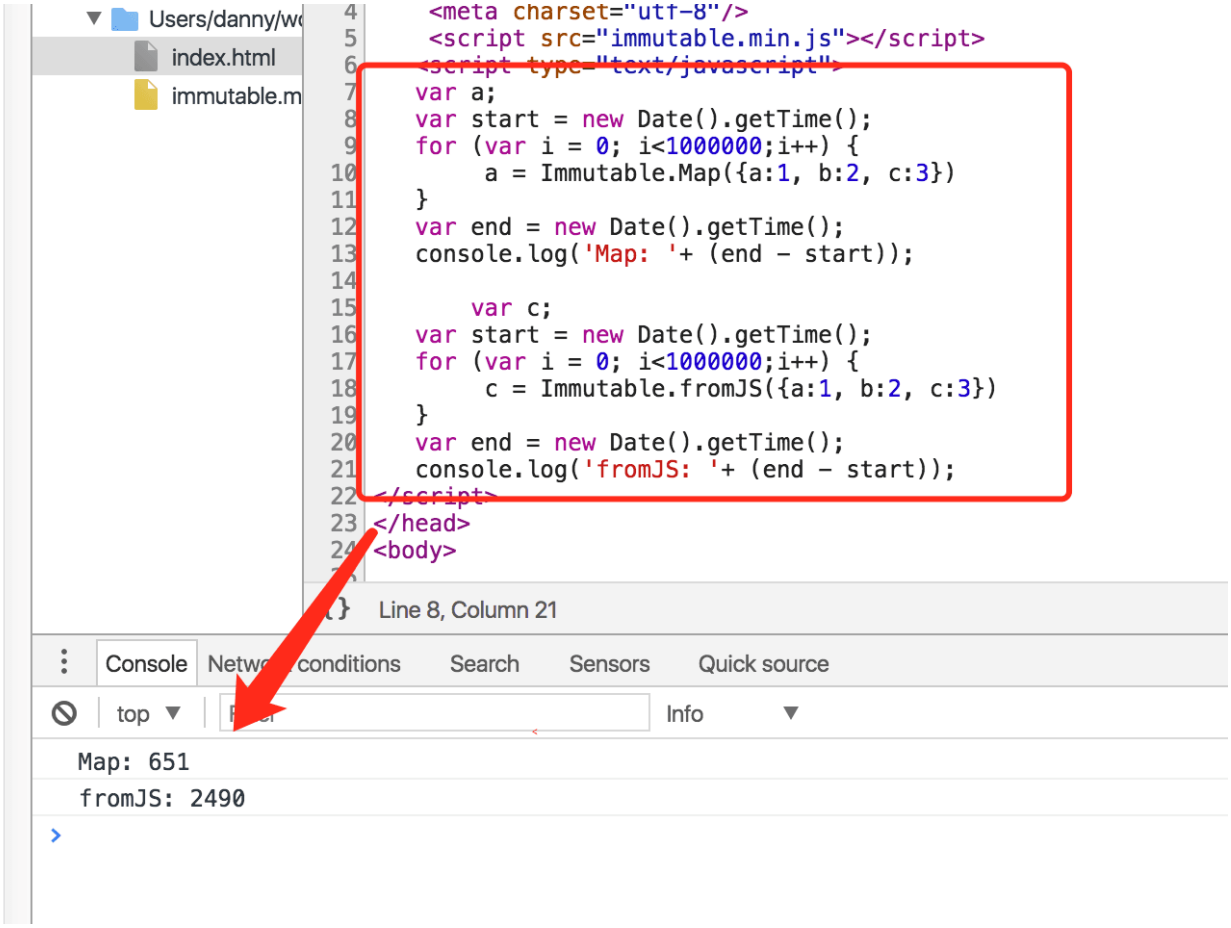
但是还有一个致命的问题是，对现有代码的改造，使用 Immutable.js 成本实在太大。

而seamless-immutable虽然数据结构和API不如Immutable.js丰富，但是对于只想使用Immutable Data来对React进行优化以避免重复渲染的我们来说，已经是绰绰有余了。而且Array和Object原生的方法等都可以直接使用，原有项目改动极小。

§ 3.4 Immutable.js使用过程中的一些注意点

- 1.fromJS和toJS会深度转换数据，随之带来的开销较大，尽可能避免使用，单层数据转换使用Map()和List()；

（做了个简单的fromJS和Map性能对比，同等条件下，分别用两种方法处理1000000条数据，可以看到fromJS开销是Map的4倍）



- 2.js是弱类型，但Map类型的key必须是string！(看下图官网说明)；

> Keep in mind, when using JS objects to construct Immutable Maps, that JavaScript Object properties are always strings, even if written in a quote-less shorthand, while Immutable Maps accept keys of any type.

```
let obj = { 1: "one" };
Object.keys(obj); // [ "1" ]
obj["1"]; // "one"
obj[1];   // "one"

let map = Map(obj);
map.get("1"); // "one"
map.get(1);   // undefined
```

- 3.所有针对immutable变量的增删改必须左边有赋值，因为所有操作都不会改变原来的值，只是生成一个新的变量；

```
//javascript
var arr = [1,2,3,4];
arr.push(5);
console.log(arr) //[1,2,3,4,5]

//immutable
var arr = immutable.List([1,2,3,4])
//错误用法
```

```
arr.push(5);
console.log(arr.toJS()) //[1,2,3,4]
//正确用法
arr = arr.push(5);
console.log(arr.toJS()) //[1,2,3,4,5]
```

- 4.引入immutablejs后，不应该再出现对象数组拷贝的代码(如下举例)；

```
//es6对象复制
var state = Object.assign({}, state, {
    key: value
});

//array复制
var newArr = [].concat([1,2,3])
```

- 5.获取深层深套对象的值时不需要做每一层级的判空；

```
//javascript
var obj = {a:1}
var res = obj.a.b.c    //error

//immutable
var immutableData=immutable.fromJS({a:1})
var res = immutableData.getIn(['a', 'b', 'c'])  //undefined
```

- 6.immutable对象直接可以转JSON.stringify(),不需要显式手动调用toJS()转原生；
- 7.判断对象是否是空可以直接用size；
- 8.调试过程中要看一个immutable变量中真实的值，可以chrome中加断点，在console中使用.toJS()方法来查看；

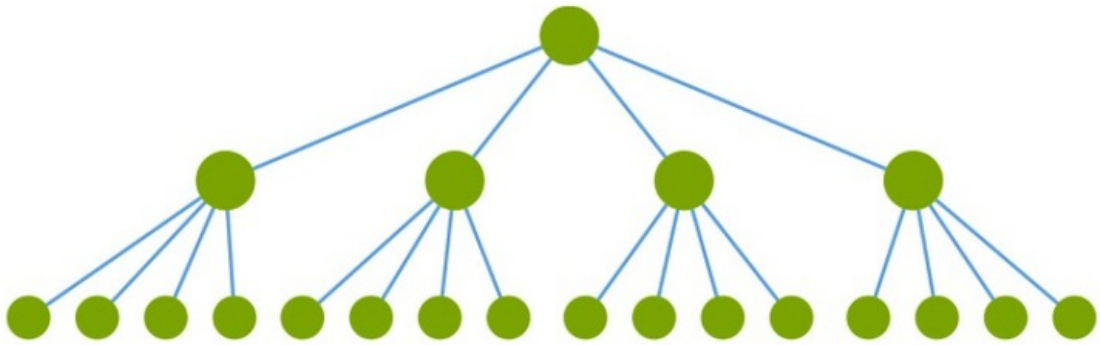
§ 4.React 默认的渲染行为

> React在减少重复渲染方面确实是有一套独特的处理办法，那就是虚拟DOM，但显然在首次渲染的时候React绝无可能超越原生的速度，或者一定能将其它的框架比下去。尤其是在优化前的React，每次数据变动都会执行re-render，大大影响了性能，特别是在移动端。

React的组件渲染分为初始化渲染（render）和更新渲染（re-render）。

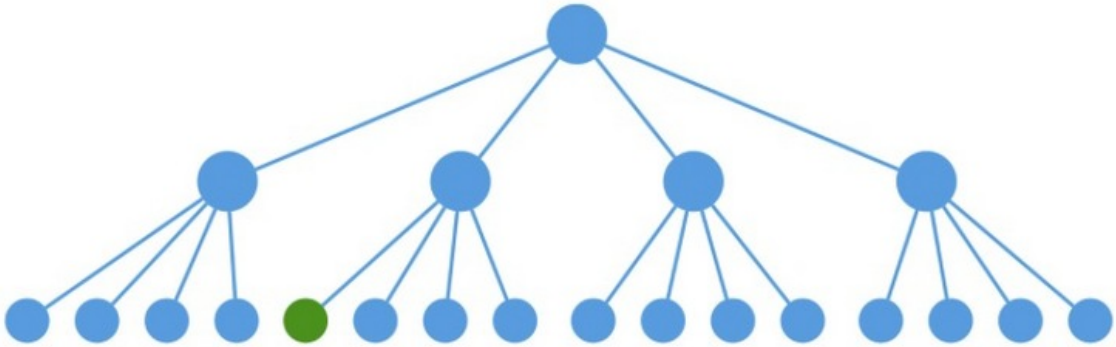
初始化渲染

在初始化渲染的时候会调用根组件下的所有组件的render方法进行渲染，如下图（绿色表示已渲染，这一层是没有问题的）：



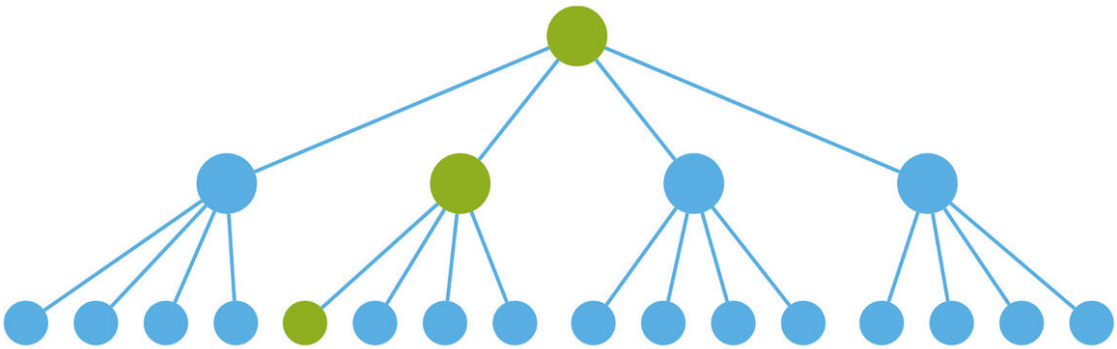
提出改变

但是当我们要更新某个子组件的时候，如下图的绿色组件（从根组件传递下来应用在绿色组件上的数据发生改变）：



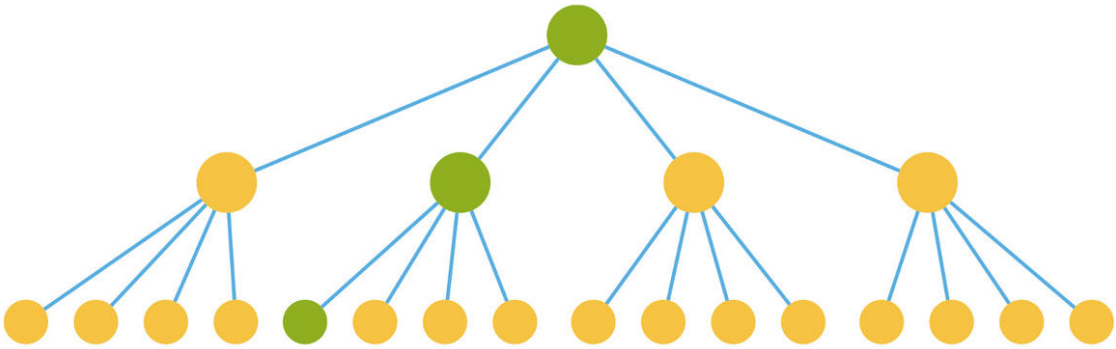
理想更新

我们的理想状态是只调用关键路径上组件的render，如下图：



默认行为

但是React的默认做法是调用所有组件的render，再对生成的虚拟DOM进行对比，如不变则不进行更新。这样的render和虚拟DOM的对比明显是在浪费，如下图（黄色表示浪费的render和虚拟DOM对比）



从上图可以看见，组件除了必要渲染的三个节点外，还渲染了其他不必要渲染的节点，这对性能是一个很大的浪费。如果对于复杂的页面，这将导致页面的整体体验效果非常差。

> §Tips：
>
> - 拆分组件是有利于复用和组件优化的
> - 生成虚拟DOM并进行比对发生在render()后，而不是render()前

§ 4.1更新阶段的生命周期

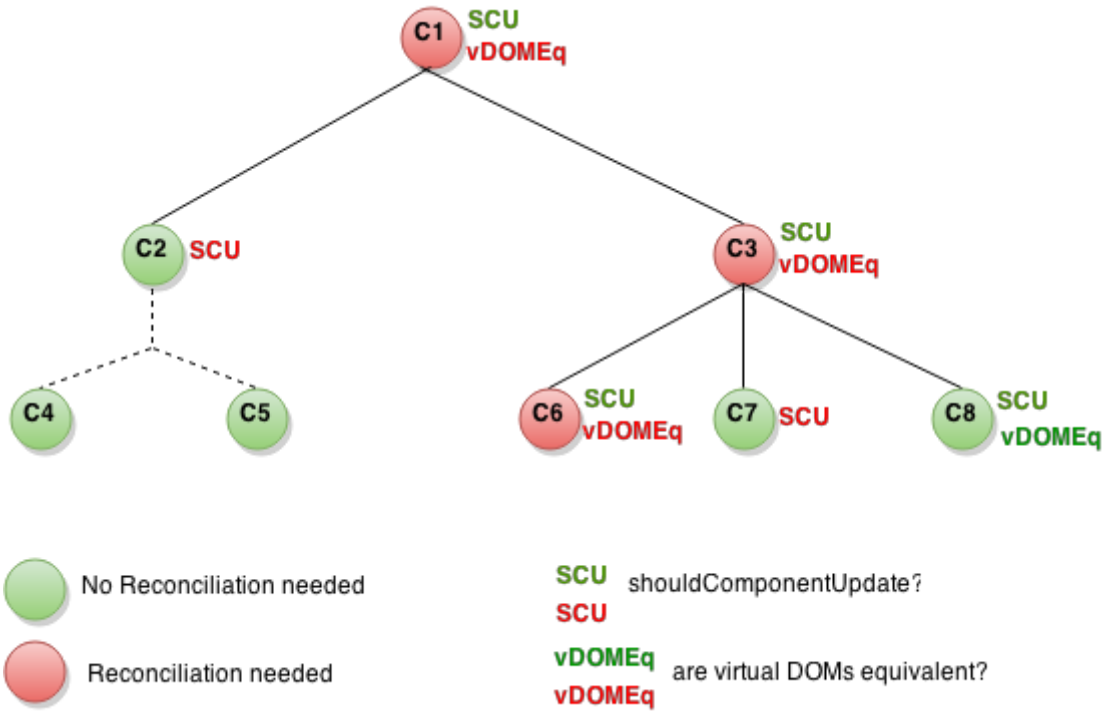
- `componentWillReceiveProps(object nextProps)`：当挂载的组件接收到新的props时被调用。此方法应该被用于比较this.props 和 nextProps以用于使用this.setState()执行状态转换。（组件内部数据有变化，使用state，但是在更新阶段又要在props改变的时候改变state，则在这个生命周期里面）；
- `shouldComponentUpdate(object nextProps, object nextState)`：返回布尔值，当有props或者state有改变发生时是否更新DOM时被调用；
- `componentWillUpdate(object nextProps, object nextState)`：在更新发生前被立即调用。你不能在此调用this.setState()；
- `componentDidUpdate(object prevProps, object prevState)`：在更新发生后被立即调用。（可以在DOM更新完之后，做一些收尾的工作）

> §Tips：
>
> - React的优化是基于`shouldComponentUpdate`的，该生命周期默认返回true，所以一旦prop或state有任何变化，都会引起重新re-render

§ 4.2关于shouldComponentUpdate

shouldComponentUpdate 是React性能优化的关键。（联系"场景二"，『===值比较』）
React的重复渲染优化的核心其实就是在shouldComponentUpdate里面做数据比较。在优化之前，shouldComponentUpdate是默认返回true的，这导致任何时候触发任何的数据变化都会使React组件component重新渲染。这必然会导致资源的浪费和性能的低下——你可能会感觉比较原生的响应更慢。

为了进一步说明问题，我们再引用一张官网的图来解释，如下图（ SCU表示shouldComponentUpdate，绿色表示返回true(需要更新)，红色表示返回false(不需要更新)；vDOMEq表示虚拟DOM比对，绿色表示一致(不需要更新)，红色表示发生改变(需要更新)）：



根据渲染流程，首先会判断shouldComponentUpdate(SCU)是否需要更新。如果需要更新，则调用组件的render生成新的虚拟DOM，然后再与旧的虚拟DOM对比(vDOMEq)，如果对比一致就不更新，如果对比不同，则根据最小粒度改变去更新DOM；如果SCU不需要更新，则直接保持不变，同时其子元素也保持不变。

- C1根节点，绿色SCU (true)，表示需要更新，然后vDOMEq红色，表示虚拟DOM不一致，需要更新。
- C2节点，红色SCU (false)，表示不需要更新，所以C4,C5均不再进行检查
- C3节点同C1，需要更新
- C6节点，绿色SCU (true)，表示需要更新，然后vDOMEq红色，表示虚拟DOM不一致，更新DOM。
- C7节点同C2
- C8节点，绿色SCU (true)，表示需要更新，然后vDOMEq绿色，表示虚拟DOM一致，不更新DOM。

> **§Tips：**
>
> - React渲染更新DOM与否是先根据render的逻辑生成虚拟DOM，再与旧的虚拟DOM进行对比，求出最小DOM更新操作，这是React做的事情。shouldComponentUpdate解决的是React的树形结构大了之后，虚拟DOM的生成非常卡的问题，因为render方法不加限制的话每次都会执行，而shouldComponentUpdate正是为了避免不必要的render，从而提高虚拟DOM的生成速度。 老实说如果不使用shouldComponentUpdate进行限制的话，react的性能是非常差的。

§ 4.3带坑的一些注意点

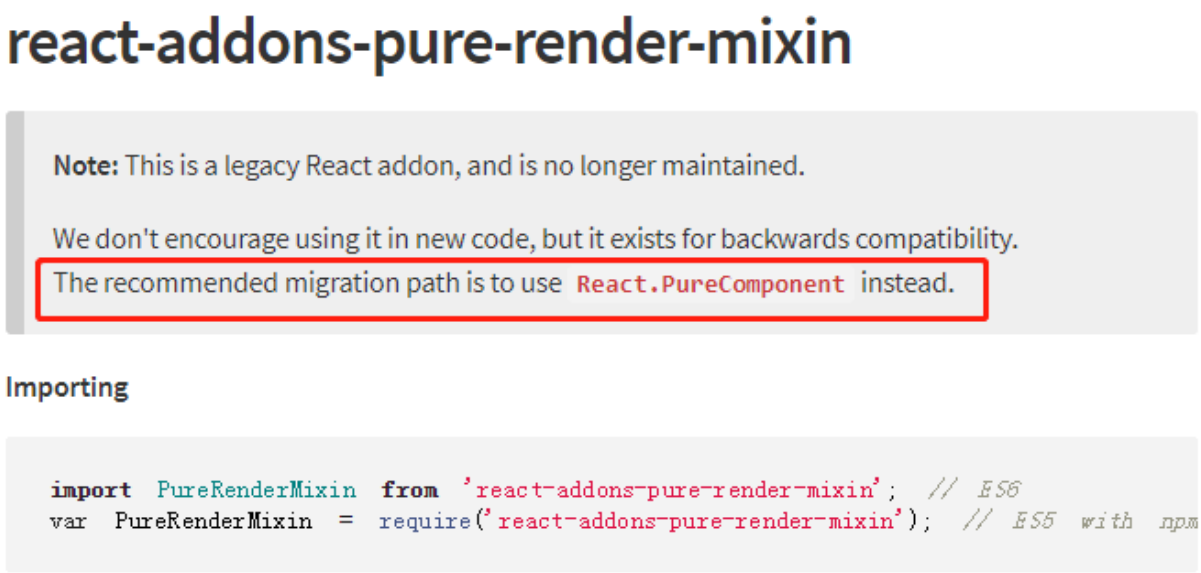
- {…this.props} (不要滥用，请只传递component需要的props，传得太多，或者层次传得太深，都会加重shouldComponentUpdate里面的数据比较负担，因此，请慎用spread attributes（ <Component {...props} /> ）。
- 请将方法的bind一律置于constructor（ Component的render里不动态bind方法，方法都在constructor里bind好，如果要动态传参，方法可使用闭包返回一个最终可执行函数。如：showDelBtn(item) { return (e) => {}; }。如果每次都在render里面的jsx去bind这个方法，每次都要绑定会消耗性能。 ）
- 复杂的页面不要在一个组件里面写完。
- 请尽量使用const element（这个用法是工业聚在React讨论微信群里教会的，我们可以将不怎么变动，或者不需要传入状态的component写成const element的形式，这样能加快这个element的初始渲染速度）
- map里面添加key，并且key不要使用index（可变的）。具体可参考使用Perf工具研究React Key对渲染的影响，React中key的必要性与使用
- 尽量少用setTimeOut或不可控的refs、DOM操作。
- props和state的数据尽可能简单明了，扁平化。
- 使用return null而不是CSS的display:none来控制节点的显示隐藏。保证同一时间页面的DOM节点尽可能的少。

§ 5.React官方的解决方案

随着React版本迭代，React官方有过以下解决方案：

§ 5.1PureRenderMixin

es5时期的`PureRenderMixin`(react-addons-pure-render-mixin)，但因为咱用的是es2015 class 的方式定义的 Component，已经不支持mixin了，这已经被从React 15.3.0 新增的`PureComponent`类替代



§ 5.2pureRender

带有es7装饰器@写法的`pureRender`(pure-render-decorator)，但这也已经被从React 15.3.0 新增的`PureComponent`类替代了

Pure render decorator

An ES7 decorator to make React components "pure".

build passing

Alternatives

- As of **v15.3.0**, React provides a `PureComponent` base class to make a component pure.
- **recompose** provides a clean and functional way to make components pure.

Installation

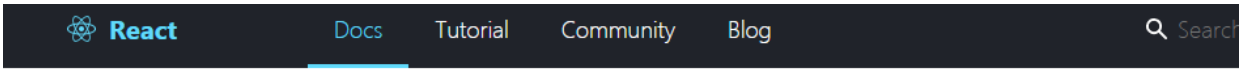
```
npm install pure-render-decorator
```

Usage

```
import {Component} from 'react';
import pureRender from 'pure-render-decorator';

@pureRender
export default class Test extends Component {
    render() {
        return <div />;
    }
}
```

§5.3shallowCompare

`shallowCompare`(react-addons-shallow-compare)，这也已经被从React 15.3.0 新增的`PureComponent`类替代了、、、



Shallow Compare

Note:

`shallowCompare` is a legacy add-on. Use `React.PureComponent` instead.

Importing

```
import shallowCompare from 'react-addons-shallow-compare'; // ES6
var shallowCompare = require('react-addons-shallow-compare'); // ES5 with npm
```

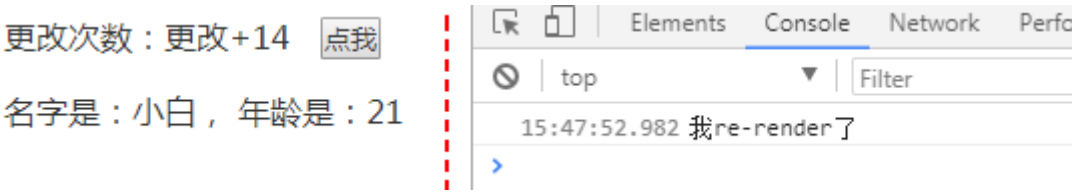
§ 5.4PureComponent

这个替代众多方案的PureComponent实际上跟之前面的方案是等价的，只是写起来会更加简介优雅，相比而言“根红苗正”是然后遗憾的是 `PureComponent` 也是shallowCompare（浅比较）、、、

例1：

```
//父组件
class PageOne extends React.Component{
    constructor(props){
        super(props);
        this.state={
            count:1,
            name:"小白",
            age:20
        };
        this.increaseTimes=this.increaseTimes.bind(this);
    }
    increaseTimes(){
        this.setState({
            count:this.state.count+1,
            name:"小白",
            age:21
        })
    }
    render(){
        let {count,name,age}=this.state;
        return(
            <ul className="pageOne">
                <li>
                    <span>更改次数：更改+{count}</span>
                    <button onClick={this.increaseTimes}>点我</button>
                </li>
```

```
                <Person name={name} age={age}/>
            </ul>
        )
    }
}

//子组件
class Person extends React.PureComponent{
    constructor(props){
        super(props)
    }
    render(){
        console.log("我re-render了");
        let {name,age}=this.props;
        return(
            <li>
                名字是：{name},
                年龄是：{age}
            </li>
        )
    }
}
```



- 我第一次去触发父组件中按钮button的onClick点击事件，由于传过去`Person`子组件的age从20更改到了21，子组件`Person`重渲染了一次，OK的；
- 而后我再去触发父组件中按钮button的onClick点击事件，传过`Person`子组件的age和name都没发生改变，还是上一次的20和"小白"，由于用了`React.PureComponent`，解决了如果用`React.Component`造成的子组件重渲染问题；

到此可能会觉着用`React.PureComponent`能解决React组件component的重渲染问题进而避免不要的性能浪费，然而遗憾的是**React.PureComponent只是shallowCompare（浅比较）（只比较第一层的值是否相同，而对于引用数据类型的『值比较』是比较是否指向的堆里的同一个对象）**，也就是说`React.PureComponent`只会当组件的 state 或者 props 没有嵌套结构的时候才会正确按照预期发挥作用，如果有嵌套层的话（在实际的项目中，嵌套的 state 或者 props 结构是很常见的），React.PureComponent就会因为浅比较而出现问题

例2：

```
//父组件
class PageOne extends React.Component{
    constructor(props){
        super(props);
        this.state={
            count:1,
```

```
            person:{
                name:"小白",
                age:20
            }
        };
        this.increaseTimes=this.increaseTimes.bind(this);
    }
    increaseTimes(){
        this.setState({
            count:this.state.count+1,
            person:{
                name:"小白",
                age:20
            }
        })
    }
    render(){
        let {count,person}=this.state;
        return(
            <ul className="pageOne">
                <li>
                    <span>更改次数：更改+{count}</span>
                    <button onClick={this.increaseTimes}>点我</button>
                </li>
                <Person person={person}/>
            </ul>
        )
    }
}

//子组件
class Person extends React.PureComponent{
    constructor(props){
        super(props)
    }
    render(){
        console.log("我re-render了");
        let {name,age}=this.props.person;
        return(
            <li>
                名字是：{name},
                年龄是：{age}
            </li>
        )
    }
}
```

- 我触发多少次父组件的按钮onClick点击事件，子组件`Person`就会重渲染多少次，虽然传过去的一直都是{name:”小白”,age:”20”}；

为什么呢？因为`React.PureComponent`的浅比较把{name:”小白”,age:”20”}和下一次的{name:”小白”,age:”20”}比较出来是不一样不一样的（因为指向于堆内存中的对象不是同一个），所以判断为要重新渲染！！！

然后我们再来通过代码看看这React官方提供的这几种解决方案

让我们扒拉一下React的源码，代码链接

```
if (this._compositeType === CompositeTypes.PureClass) {
  shouldUpdate =
    !shallowEqual(prevProps, nextProps) ||
    !shallowEqual(inst.state, nextState);
}
```

这里的 shouldUpdate 变量就是在后面的逻辑中用于判断该不该重新渲染组件的，这里有个 shallowEqual 函数，我们暂且不表。

我们再看一下 PureRenderMixin 的代码，代码链接

```
var ReactComponentWithPureRenderMixin = {
  shouldComponentUpdate: function(nextProps, nextState) {
    return shallowCompare(this, nextProps, nextState);
  },
};
module.exports = ReactComponentWithPureRenderMixin;
```

这个Mixin就是帮我们实现了 shouldComponentUpdate 函数，原来这里用到了 shallowCompare 方法，那好，我们继续看看 shallowCompare 方法的代码,代码链接

```
function shallowCompare(instance, nextProps, nextState) {
  return (
    !shallowEqual(instance.props, nextProps) ||
    !shallowEqual(instance.state, nextState)
```

```
    );
}
```

OK！OK！ 看到了吧，shallowCompare 也是对 shallowEqual 的封装，所以React官方提供的这几种解决方案归根揭底都是一样的。

那么我们现在只要搞清楚 shallowEqual 方法是怎么实现的，上面的问题就真相大白了，看代码：代码链接

> shallowEqual 不是ReactJS的代码，它是Facebook的一个工具库：fbjs

这个方法重点关注两个点：

- 1.如它的名字一样，这个方法只进行对象的浅比较，我们知道deepCompare是循环递归操作，开销会比较大，得不偿失的。
- 2.比较对象属性的值，用的是 Object.is 方法;

```
let isEqual=Object.is(
    {name:"小白",age:20},
    {name:"小白",age:20}
);
console.log(isEqual,"isEqual")//输出的是false
```

所以我们知道了:

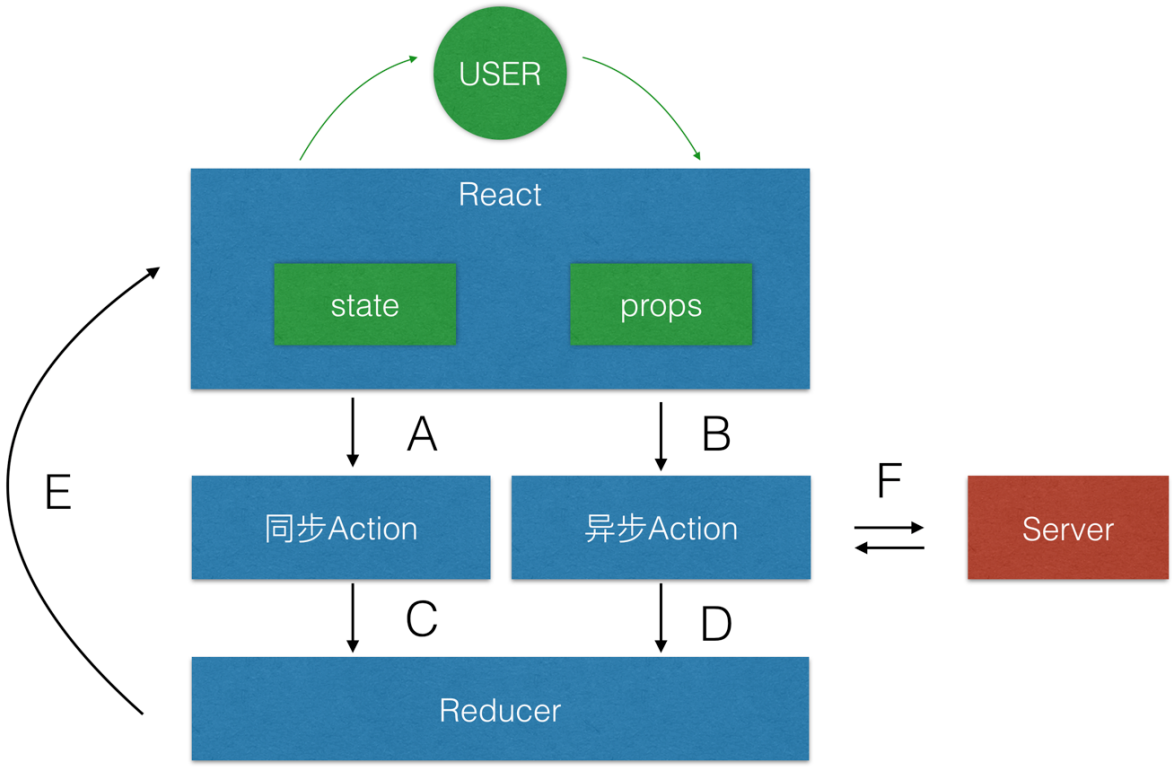
- 如何避免React组件的重渲染而造成不必要的浪费关键之处在于控制React生命周期中的shouldComponentUpdate()
- 然而React官方提供的几种解决方案都只是浅比较(shallowCompare)，当传入props或state不止一层，或者未array和object时，浅比较(shallowCompare)就失效了。
- 当然我们也可以在 shouldComponentUpdate() 中使用使用 deepCopy 和 deepCompare 来避免不必要的 re-render()，但 deepCopy（深拷贝）和 deepCompare（循环层层递归比较）一般都是非常耗性能的。
- 这个时候我们就需要 Immutable.js，Immutable.js的几大特性解决上面这个React性能问题（Immutable 则提供了简洁高效的判断数据是否变化的方法，只需 === 和 is 比较（比较 hashCode）就能知道是否需要执行 render()，而这个操作几乎 0 成本，所以可以极大提高性能），当然还有一点是Immutable.js提供了丰富的Api，终于回到了主线上 `Immutable.js在React/Redux中的应用` ！！！

§ 6.Immutable.js在React/Redux中的应用

§ 6.1使用 immutable 的边界性问题

>

> 应用Immutable.js于React/Redux中，我们有必要来划分一下边界，哪些数据需要使用不可变数据，哪些数据要使用原生js数据结构，哪些地方需要做互相转换；



- 在redux中，全局state必须是immutable的，这点毋庸置疑是我们使用immutable来优化redux的核心；
- 组件props是通过redux的connect从state中获得的，并且引入immutableJS的另一个目的是减少组件shouldComponentUpdate中不必要渲染，shouldComponentUpdate中比对的是props，如果props是原生JS就失去了优化的意义；
- 组件内部state如果需要提交到store的，必须是immutable，否则不强制；
- 从视图层向同步和异步 action 发送的数据（A/B），必须是 immutable 的；
- Action 提交给 reducer 的数据（C/D），必须是 immutable 的；
- reducer中最终处理state(E)必须是以immutable的形式处理并返回；
- 与服务端ajax交互中返回的callback统一封装，第一时间转换成immutable数据；

这样似乎看起来，所有地方都是 immutable 的，除开异步 action 和服务器的交互是 JSD （F）。换句话说，我们要求，除了向服务端发送数据请求的时候，其他位置，不允许出现 `toJS` 的代码。而接收到服务端的数据后，在流转入全局 state 之前，统一转化为 immutable 数据。

为什么要做这种统一呢？是因为：

- 避免 JSD（js对象）和 immutable 对象的混用导致出错（可能在大型项目中，混用 js对象 和 immutable 对象，会让coder自己都不清楚一个变量中存储的到底是什么类型的数据）；
- 统一JSD（js对象）和 immutable对象 的转化路径。比如你在局部 state 里不使用 immutable，但是这些局部的 state 很可能被用来通过异步 action 提交的服务端，这样在数据流里就会同时存在 JSD 和 immutable，这对于代码的维护性是一种灾难。

有的人说，我觉得成本太大，我只想在 reducer 里局部使用 immutable，于是代码变成了这样：

```
export default function indexReducer(state, action) {
    switch (action.type) {
    case RECEIVE_MENU:
        state = immutable.fromJS(state); //转成immutable对象
        state = state.merge({a:1});
        return state.toJS() //转回原生js
    }
}
```

这样看起来只是让 immutable 侵入到 reducer 中，其实却是得不偿失的。原因有二：

- fromJS() 和 toJS() 是深层的互转immutable对象和原生对象，性能开销大，尽量不要使用；
- 组件中props和state还是原生js，shouldComponentUpdate仍然无法做利用immutablejs的优势做深度比较；

§ 6.2用Immutable.js改造shouldComponentUpdate

shouldComponentUpdate具体怎么封装有很多种办法，我们这里选择了封装一层component的基类，在基类中去统一处理shouldComponentUpdate，组件中直接继承基类的方式

> 注意：React 中规定 state 和 props 只能是一个普通对象，所以比较时要比较对象的 value

```
import React from 'react';
import {is} from 'immutable';

class BaseComponent extends React.Component {
    constructor(props, context, updater) {
        super(props, context, updater);
    }

    shouldComponentUpdate(nextProps, nextState) {
        const thisProps = this.props || {};
        const thisState = this.state || {};
        nextState = nextState || {};
        nextProps = nextProps || {};

        if (Object.keys(thisProps).length !== Object.keys(nextProps).leng
th ||
            Object.keys(thisState).length !== Object.keys(nextState).leng
th) {
            return true;
        }

        for (const key in nextProps) {
            if (!is(thisProps[key], nextProps[key])) {
                return true;
            }
        }
```

```
        for (const key in nextState) {
            if (!is(thisState[key], nextState[key])) {
                return true;
            }
        }
        return false;
    }
}

export default BaseComponent;
```

组件中如果需要使用统一封装的shouldComponentUpdate，则直接继承基类

```
import BaseComponent from './BaseComponent';
class Menu extends BaseComponent {
    constructor(props) {
        super(props);
    }
    ……
}
```

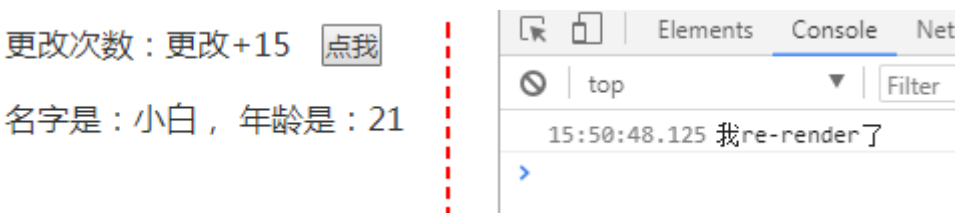
当然如果组件不想使用封装的方法，那直接在该组件中重写shouldComponentUpdate就行了

还是5.4PureComponent中的例子，然后我们用改造后的shouldComponentUpdate来验证下：

```
//父组件
import {fromJS,Map} from "immutable";

class PageOne extends React.Component{
    constructor(props){
        super(props);
        this.state={
            $state:fromJS({
                count:1,
                person:{
                    name:"小白",
                    age:20
                }
            })
        };
        this.increaseTimes=this.increaseTimes.bind(this);
    }
    increaseTimes(){
        this.setState(({$state})=>(
            {
                $state:$state.update("count",()=>this.state.$state.get("c
ount")+1)
```

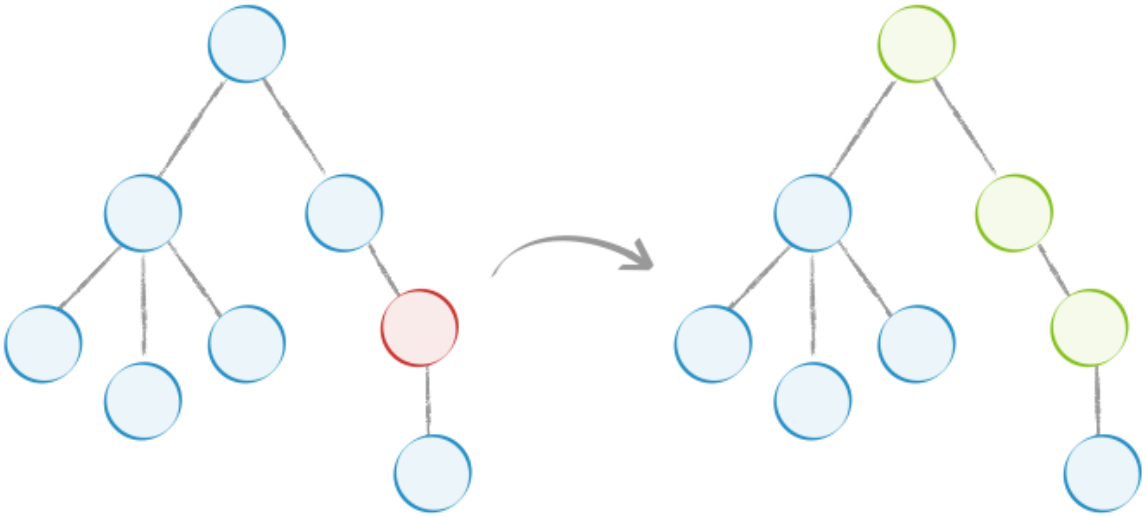
```
                                .update("person",()=>Map({name:"小
白",age:21}))
            }
        ))
    }
    render(){
        let {$state}=this.state;
        return(
            <ul className="pageOne">
                <li>
                    <span>更改次数：更改+{$state.get("count")}</span>
                    <button onClick={this.increaseTimes}>点我</button>
                </li>
                <Person $person={$state.get("person")}/>
            </ul>
        )
    }
}
```

```
//子组件
import BaseComponent from "../baseComponent/baseComponent.jsx";

class Person extends BaseComponent{
    constructor(props){
        super(props)
    }
    render(){
        console.log("我re-render了");
        let {$person}=this.props;
        return(
            <li>
                名字是：{$person.get("name")},
                年龄是：{$person.get("age")}
            </li>
        )
    }
}
```



可以明显看到，仅重新渲染了age由20更改到21这一次！

使用 Immutable 后，如下图，当红色节点的 state 变化后，不会再渲染树中的所有节点，而是只渲染图中绿色的部分：



你也可以不用自己去手动封装shouldComponentUpdate或者在组件中重写，可以用封装好了的react-immutable-render-mixin这个库就好了，还可以用es7的装饰器，哈哈哈

```
import React from "react";
import {immutableRenderDecorator} from "react-immutable-render-mixin";

immutableRenderDecorator
class Person extends React.Component{
    constructor(props){
        super(props)
    }
    render(){
        console.log("我re-render了");
        let {$person}=this.props;
        return(
            <li>
                名字是：{$person.get("name")},
                年龄是：{$person.get("age")}
            </li>
        )
    }
```

和上面的效果是一样一样的

另外在应用immutable.js于React的项目中 ，组件 props 的校验也需要与时俱进，使用 immutable 类型校验，这就需要我们 import 专门针对 immutable 类型进行校验的库：react-immutable-proptypes，使用方法基本上和普通的 PropTypes 一致：

```
import React from "react";
import ImmutablePropTypes from "react-immutable-proptypes";
import BaseComponent from "../baseComponent/baseComponent.jsx";

immutableRenderDecorator
class Person extends React.Component{
   static propTypes = {
```

```
        $person:ImmutablePropTypes.map
    };
    constructor(props){
        super(props)
    }
    render(){
        console.log("我re-render了");
        let {$person}=this.props;
        return(
            <li>
                名字是：{$person.get("name")},
                年龄是：{$person.get("age")}
            </li>
        )
    }
```

§ 6.3与Redux搭配使用

Redux 是目前流行的 Flux 衍生库。它简化了 Flux 中多个 Store 的概念，只有一个 Store，数据操作通过 Reducer 中实现；同时它提供更简洁和清晰的单向数据流（View -> Action -> Middleware -> Reducer），也更易于开发同构应用。目前已经在我们项目中大规模使用。

而且 Flux 并没有限定 Store 中数据的类型，使用 Immutable 非常简单。

按照 Redux 的工作流，我们从创建 store 开始。Redux 的 createStore 可以传递多个参数，前两个是： reducers 和 initialState。

reducers 我们用 redux-immutable 提供的 combineReducers 来处理，他可以将 immutable 类型的全局 state 进行分而治之：

```
import {combineReducers} from "redux-immutable";

const rootReducer = combineReducers({
    $left: leftReducer,
    $right: rightReducer
});
```

当然 initialState 需要是 immutable 的：

```
const initialState = Immutable.Map();
const store = createStore(rootReducer, initialState);
```

如果你不传递 initialState，redux-immutable也会帮助你在 store 初始化的时候，通过每个子 reducer 的初始值来构建一个全局 Map 作为全局 state。当然，这要求你的每个子 reducer 的默认初始值是 immutable的。

> **§参考地址：**

- 前端高质量知识(一)-JS内存空间详细图解
- javascripe中深度拷贝使用JSON.stringify和parse好么?
- 【JS】深拷贝 vs 浅拷贝
- 关于JavaScript的浅拷贝和深拷贝
- js内存堆栈，递归原理以及浅拷贝和深拷贝的理解
- 数据结构之Trie树
- Trie实践：一种比哈希表更快的数据结构
- 精读 Immutable 结构共享
- immutable入坑指南
- Immutable.js 以及在 react+redux 项目中的实践
- Immutable 详解及 React 中实践
- 在react/redux中使用Immutable
- 使用immutable优化React
- React性能优化总结
- 赢财富（微信版）目前存在的性能问题–reac性能问题
- React中key的必要性与使用
- React.PureComponent 配上 ImmutableJS 才更有意义
- react如何性能达到最大化(前传)，暨react为啥非得使用immutable.js