## 组件

 React 中，UI 以组件的形式来搭建，组件之间可以嵌套组合。另，React 中组件间通信的数据流是单向的，**顶层组件可以通过 props 属性向下层组件传递数据，而下层组件不能向上层组件传递数据，兄弟组件之间同样不能**。这样简单的**单向数据流**支撑起了 React 中的数据可控性。

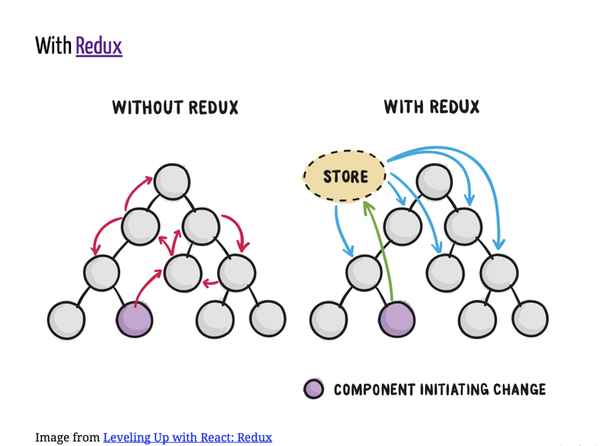
Redux 对 Flux 架构的一些简化。如 Redux 限定一个应用中只能有单一的 store，这样的限定能够让应用中数据结果集中化，提高可控性。

阿里 切流量:用户后两位,百分比进行切留

## 单向数据流

React redux单向数据流，view发出action后不修改原有state，而是返回一个新的state。

不过，就算双向（修改原有state），做这样一套历史记录也很容易吧，state改变的同时保存一下上一个state就可以了啊。



## Redux

Redux 主要分为三个部分 Action、Reducer、及 Store

### Action

在 Redux 中，**action 主要用来传递操作 State 的信息，以 Javascript Plain Object 的形式存在**，如

|  |
| --- |
| {  type : 'ADD\_FILM' ,  name : 'Mission: Impossible'  } |

在上面的 Plain Object 中，type 属性是必要的，除了 type 字段外，action 对象的结构完全取决于你，建议尽可能简单。type 一般用来表达处理 state 数据的方式。如上面的 'ADD\_FILM' 表达要增加一个电影。而 name 表达了增加这个电影的电影名为 'Mission: Impossible'。那么，当我们需要表达增加另一部电影时，就需要另外一个action，如

|  |
| --- |
| {  type : 'ADD\_FILM' ,  name : 'Minions'  } |

上面写法没有任何问题，但细想，当我们增加的电影越来越多的时候，那这种直接声明的 Plain Object 将越来越多，不好组织。实际上，我们可以通过创建函数来生产 action，这类函数统称为 Action Creator，如

|  |
| --- |
| function addFilm ( name ) {  return { type : 'ADD\_FILM' , name : name } ;  } |

这样，通过调用 addFilm(name) 就可以得到对应的 Action，非常直接。

### Reducer

有了 Action 来传达需要操作的信息，那么就需要有根据这个信息来做对应操作的方法，这就是 Reducer。 Reducer 一般为简单的处理函数，通过传入旧的 state 和指示操作的 action 来更新 state，如

function films ( state = initialState , action ) {

switch ( action . type ) {

case 'ADD\_FILM' :

// 更新 state 中的 films 字段

return [{

id : state.films.reduce( ( maxId , film ) = > Math . max ( film . id , maxId ) , - 1 ) + 1 ,

name : action . name

}, ...state] ;

case 'DELETE\_FILM' :

return state.films.reduce(film =>

film.id !== action . id

);

case 'SHOW\_ALL\_FILM' :

return Object.assign({}, state , {

visibilityFilter : action . filter

});

default :

return state ;

}

上面代码展示了 Reducer 根据传入的 action.type 来匹配 case 进行不同的 state 更新。

显然，当项目中存在越来越多的 action.type 时，上面的 films 函数（ Reducer ）将变得越来越大，越来越多的 case 将导致代码不够清晰。所以在代码组织上，通常会将 Reducer 拆分成一个个小的 reducer，每个 reducer 分别处理 state 中的一部分数据，最终将处理后的数据合并成为整个 state。

在上面的代码中，我们可以把 'ADD\_FILM' 和 'DELETE\_FILM' 归为操作 state.films 的类，而 'SHOW\_ALL\_FILM' 为过滤显示类，所以可以把大的 film Reducer 拆分成 filmReducer 和 filterReducer，如

1 filmReducer

function filmReducer ( state = [ ] , action ) {

switch ( action . type ) {

case 'ADD\_FILM' :

// 更新 state 中的 films 字段

return [ {

id : state . films . reduce ( ( maxId , film ) = > Math . max ( film . id , maxId ) , - 1 ) + 1 ,

name : action . name

} , . . . state ] ;

case 'DELETE\_FILM' :

return state . films . filter ( film = >

film . id !== action . id

) ;

default :

return state ;

}

}

2 filterReducer

function filterReducer ( state , action ) {

switch ( action . type ) {

case 'SHOW\_ALL\_FILM' :

return Object . assign ( { } , state , {

visibilityFilter : action . filter

} ) ;

default :

return state ;

}

}

最后，通过组合函数将上面两个 reducers 组合起来，如

function rootReducer ( state = { } , action ) {

return {

films : filmReducer ( state . films , action ) ,

filter : filterReducer ( state . filter , action )

} ;

}

上面的 rootReducer 将不同部分的 state 传给对应的 reducer 处理，最终合并所有 reducer 的返回值，组成整个state。

实际上，Redux 提供了 combineReducers() 方法来做 rootReducer 所做的事情。使用 combineReducers 来重构 rootReducer，如

var rootReducer = combineReducers ( {

films : filmReducer ,

filter : filterReducer

} ) ;

combineReducers() 将调用一系列 reducer，并根据对应的 key 来筛选出 state 中的一部分数据给相应的 reducer，这样也意味着每一个小的 reducer 将只能处理 state 的一部分数据，如：filterReducer 将只能处理及返回 state.filter 的数据，如果需要使用到其他 state 数据，那还是需要为这类 reducer 传入整个 state。

在 Redux 中，一个 action 可以触发多个 reducer，一个 reducer 中也可以包含多种 action.type 的处理。属于多对多的关系。

### Store

## Redux

## 基础语法

### 三点运算符

Es6 扩展运算符

扩展运算符用三个点号表示，功能是把数组或类数组对象展开成一系列用逗号隔开的值

**var** foo = **function**(a, b, c) {

console.log(a);

console.log(b);

console.log(c);

}

**var** arr = [1, 2, 3];

//传统写法

foo(arr[0], arr[1], arr[2]);

//使用扩展运算符

foo(...arr);

//1

//2

//3

特殊应用场景：

//数组深拷贝

**var** arr2 = arr;

**var** arr3 = [...arr];

console.log(arr===arr2); //true, 说明arr和arr2指向同一个数组

console.log(arr===arr3); //false, 说明arr3和arr指向不同数组

//把一个数组插入另一个数组字面量

**var** arr4 = [...arr, 4, 5, 6];

console.log(arr4);//[1, 2, 3, 4, 5, 6]

//字符串转数组

**var** str = 'love';

**var** arr5 = [...str];

console.log(arr5);//[ 'l', 'o', 'v', 'e' ]

rest运算符也是三个点号，不过其功能与扩展运算符恰好相反，把逗号隔开的值序列组合成一个数组

//主要用于不定参数，所以ES6开始可以不再使用arguments对象

**var** bar = **function**(...args) {

**for** (**let** el **of** args) {

console.log(el);

}

}

bar(1, 2, 3, 4);

//1

//2

//3

//4

bar = **function**(a, ...args) {

console.log(a);

console.log(args);

}

bar(1, 2, 3, 4);

//1

//[ 2, 3, 4 ]

rest运算符配合解构使用：

**var** [a, ...rest] = [1, 2, 3, 4];

console.log(a);//1

console.log(rest);//[2, 3, 4]

### Object.assign

### [createClass( ) 和 Component( )](http://blog.csdn.net/u014695532/article/details/52830545)区别

The API (via ‘extends React.Component’) is similar to React.createClass with the exception of getInitialState. Instead of providing a separate getInitialState method, you set up your own state property in the constructor.

也就是说，当我们通过es6类的继承实现时去掉了getInitialState这个hook函数， 规定state的初始化要在constructor中声明。



Babel的Blog上还提到另外一种实现方法，即直接使用赋值语句：

