**响应式编程：RxJS的最佳实践**

1. **绪论**

**1.1什么是响应式编程？**

响应式编程(Reactive Programming)是一种面向异步数据流(Reactive Programming)和变化传播的编程方式，使用响应式编程，我们可以在编程语言中很方便地表达静态或动态的数据流，而相关的计算模型会自动将变化的值通过数据流进行传播。

响应式编程中最重要的就是stream与data 。

流(stream)廉价且常见，任何东西都可以是一个Stream：变量、用户输入、属性、Cache、数据结构等等。数据(data)是流发生的时候所产生的数据。

常见的异步数据流是异步事件（asynchronous event），比较典型的有user click event, mouse hover event 等等，通过这些事件会产生数据data和一系列的流操作stream。

顾名思义，Reactive Extensions 将event 延伸为data和stream ，也就是时间序列上的一连串资料事件，透过观察(Observe)观察这些data streams，并依据其造成的边界影响(side effects)进行相应的动作。

这种观察异步数据流进行编程的方式，就叫响应式编程。响应式编程的实现库有很多：Rx、Bacon.js、RAC等等。其中最为完善的就是RxJS，它几乎支持目前全部的编程语言，引入项目使用十分便利，数据流操作的相关功能接口也很全面。

**1.2什么是RxJS？**

RxJS是使用Observables进行响应式编程的库，使编写异步或基于回调的代码更容易。 该项目是Reactive-Extensions / RxJS的重写，具有更好的性能，更好的模块化，更好的可调试调用堆栈，同时保持大部分向后兼容，并且有一些突破性的更改，减少API的处理操作。RxJS是Observable的Javascript实现。

我们可以对比promise的编程方式来聊聊Observable的具体定义。

var promise = new Promise((resolve, reject) => {  
 setTimeout(() => resolve('foo'), 1000);  
});  
  
var observable = Observable.create((observer) => {  
 observer.onNext('foo');  
 observer.onNext('bar');  
 var complete = setTimeoutout(() => observer.*onComplete*(), 2000);  
 return () => {  
 clearTimeout(complete);  
 }  
});

从上面可以看出Observable和Promise的区别是功能，Promise的功能相对单一，因为异步处理单值流要比异步处理多值流，变化空间要小很多。Observable的变化要多得多。

promise只能resolve一次，输出单值，observable能输出多值。promise一旦建立就只有两种可能，要么resolve，要么reject，对应observable的complete，error，但observable可以定义如何cancel。

let disposable = observable.forEach((value) => console.log(value));  
disposable(); *// 在observable输出任何值前就取消了*

上面的Observable是懒惰的，observable对象建立并不立即输出值，而是等到有人来subscribe的时候才开始输出。

做一个搜索功能在前端开发中其实并不陌生，一般的实现方式是：监听文本框的输入事件，将输入内容发送到后台，最终将后台返回的数据进行处理并展示成搜索结果。  
  
<input id="text"></input>  
<*script*>  
var *text* = document.querySelector('#text');  
*text*.addEventListener('keyup', (e) =>{  
 var searchText = e.target.value;  
 *// 发送输入内容到后台  
 $*.ajax({  
 url: `search.qq.com/${searchText}`,  
 success: data => {  
 *// 拿到后台返回数据，并展示搜索结果  
 render*(data);  
 }  
 });  
});  
</script>  
上面代码实现我们要的功能，但存在两个较大的问题：  
  
多余的请求  
当想搜索“爱迪生”时，输入框可能会存在三种情况，“爱”、“爱迪”、“爱迪生”。而这三种情况将会发起 3 次请求，存在 2 次多余的请求。  
  
已无用的请求仍然执行  
一开始搜了“爱迪生”，然后马上改搜索“达尔文”。结果后台返回了“爱迪生”的搜索结果，执行渲染逻辑后结果框展示了“爱迪生”的结果，而不是当前正在搜索的“达尔文”，这是不正确的。  
  
减少多余请求数，可以用 setTimeout 函数节流的方式来处理，核心代码如下  
  
<input id="text"></input>  
<script>  
 var text = document.querySelector('#text'),  
 timer = null;  
 text.addEventListener('keyup', (e) =>{  
 *// 在 250 毫秒内进行其他输入，则清除上一个定时器* clearTimeout(*timer*);  
 *// 定时器，在 250 毫秒后触发* timer = setTimeout(() => {  
 console.log('发起请求..');  
},250)  
})  
</script>  
已无用的请求仍然执行的解决方式，可以在发起请求前声明一个当前搜索的状态变量，后台将搜索的内容及结果一起返回，前端判断返回数据与当前搜索是否一致，一致才走到渲染逻辑。最终代码为  
  
<input id="text"></input>  
<script>  
 var text = document.querySelector('#text'),  
 timer = null,  
 currentSearch = '';  
  
 text.addEventListener('keyup', (e) =>{  
 clearTimeout(*timer*)  
 timer = setTimeout(() => {  
 *// 声明一个当前所搜的状态变量* currentSearch ＝ '书';  
  
 var searchText = e.target.value;  
 $.ajax({  
 url: `search.qq.com/${searchText}`,  
 success: data => {  
 *// 判断后台返回的标志与我们存的当前搜索变量是否一致* if (data.search === currentSearch) {  
 *// 渲染展示* render(data);  
} else {  
 *// ..*}  
}  
});  
},250)  
})  
</script>  
上面代码基本满足需求，但代码开始显得乱糟糟。我们来使用 RxJS 实现上面代码功能，如下  
  
var *text* = document.querySelector('#text');  
var *inputStream* = Rx.Observable.*fromEvent*(*text*, 'keyup')  
 .*debounceTime*(250)  
 .pluck('target', 'value')  
 .switchMap(url => Http.get(url))  
 .subscribe(data => render(data));

可以明显看出，基于 RxJS 的实现，代码十分简洁！

1. **RxJS在项目中的应用**

**2.1 RxJS方法简介**

在BDP中，很多的组件存在需要同时控制处理多个Promise的情况，每个promise的结果，需要处理不同的

**引入RxJS**

import Rx from 'RxJS/Rx';

**初始化RxJS的对象Subject**

constructor() {  
 this.subject = new Rx.*Subject*();  
 this.subject.filter(context => context.isChildMatch)  
 .distinctUntilChanged(null, context => context.sidebarFocusTimestamp)  
 .subscribe(this.*refocus*);  
}

**将需要观察的对象observable传递给Subject进行观察**

componentWillMount() {  
 this.userSubject.next(this.props);  
}

**componentWillReceiveProps是react的六大周期函数之一，该函数可以嗅探到props的变化并且获取新的props，配合该函数可以将新的props传递给观察者Observer，触发消费函数Subscribe()。**

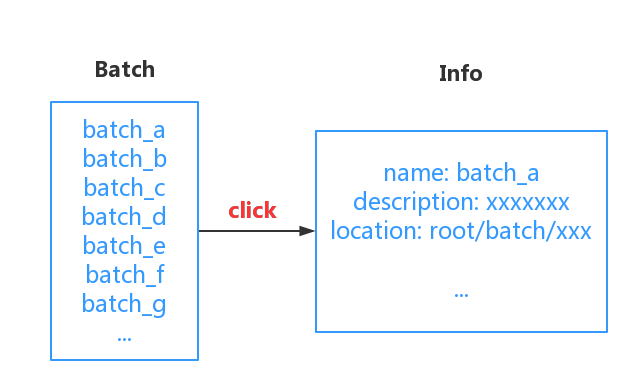
componentWillReceiveProps(nextProps) {  
 this.userSubject.next(nextProps);  
}

**RxJS在BDP中的带来的好处**

在BDP中经常会涉及到频繁处理多个服务端实时消息流的情况，如输入一个关键词进行API查询然后返回数据重新渲染视图。下面以两个较为典型的示例来进行对比，从而可以看出RxJS的优势。

**2.2.RxJS应用实例**

**2.2.1.myBatch页面**

用户选择左侧的列表中的BatchList，则实时读取API显示该Batch的Kerberos Information。

**a.非RxJS方法（代码均已简化）**

我们先来看一下非RxJS的处理流程以及存在的问题。

整个链式流程，用户选某个batch，触发Ajax获取batch的信息，重新渲染页面：

onSelect => onChange => loadInfo => reset

非RxJS方法代码简要逻辑：

Constructor()中建立一个state对象，其中包含一个form对象，用以维护右侧的数据

constructor() {  
 super();

*//初始化状态控制容器form*  
 this.state = {  
 form: {},  
 };  
}

componentWillMount()中对form对象进行初始化，赋予初值类型。

componentWillMount() {  
 const { a*List* = [] } = this.props;  
 this.setState({  
 form: {  
 batch: *aList*[0],  
 admins: [],  
 },  
 });

*//promiseList是页面同构的重要判断依据*  
 promiseList.push(this.loadInfo(*aList*[0]));  
}

设置选择batch时触发函数：onSelect，在该函数中用setstate控制右侧显示已经选择的batch。同时触发onChange函数，然后触发loadInfo函数，从API获取最新的kerberos information并用reset显示。

onSelect = (event, props) => {  
 const Name = props['data-batch'];  
 this.loadInfo (Name);  
};

在这一系列状态控制中，我们会发现存在这样的问题，当用户选择了一个新的batch：b\_A，在b\_A的kernos信息仍然在loading时，选择另外一个batch：b\_B，因为获取API的Promise操作是并发的，所以获取b\_A和b\_B的Ajax请求会同时发生， 若b\_B的Ajax请求先完成，则会出现先显示b\_B的信息然后再显示b\_A的信息的情况。那么如何避免这样的情况呢？  
答案是利用redux的数据视图分离，每次渲染时候的batch值只以最后点击产生的batch值为准，具体代码如下：  
loadInfo = (name, force) => {  
 if (!name) return null;  
 const { dispatch, userInfo } = this.props;  
 if (!info[name] || force) {  
 *//userLoad会调用ajax请求将获得的Info存入到Store之中。* dispatch(*userLoad*(name)).then(() => {  
 const form = Object.assign({}, this.state.form, {  
 batch: name,  
 admins: info[name].admins,  
 });  
 this.setState({ form });  
 });  
 }  
 return null;  
};

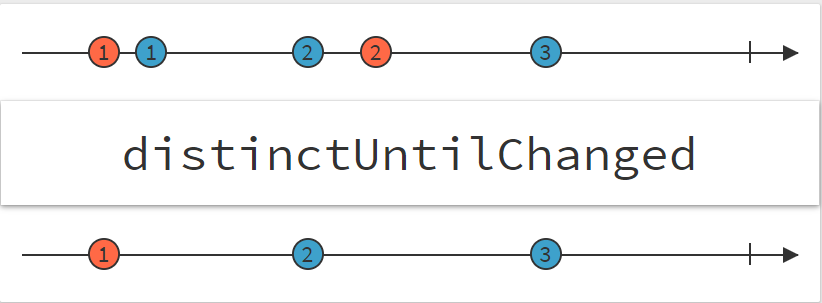
存在的问题：

1. 用户连续短时间内多次选择同一个batch的时候，会触发相同的API请求，因为Ajax请求是并发的，因此资源会浪费严重。
2. 用户选择了batch就一定会触发相应的Ajax请求，资源浪费也很严重。

为了解决上述两个问题，我们引入了RxJS来进行Promise请求的筛选。

**b.RxJS方法：**

首先我们要明白RxJS是一个流式处理操作，我们没可以将整个数据处理想想成一个pipe管道，在这个管道中运行的数据流，可以通过设置一个变量作为锚点，来进行细致的函数筛选。本例中我们将筛选的函数是distinctUntilChanged、锚点是batch：连续相同的batch将被过滤只选择第一个。



采用了distinctUntilChanged筛选以后，我们可以有效的避免短时间内的多次相同Ajax请求造成的资源浪费。

this.subject.distinctUntilChanged(null, props => this.*getInfo*(props)).subscribe((props) => {

*//锚点选择通过this.getInfo,筛选去除相邻重复的请求。*  
const name = this.*getInfo*(props);

this.*loadInfo*(name);  
});

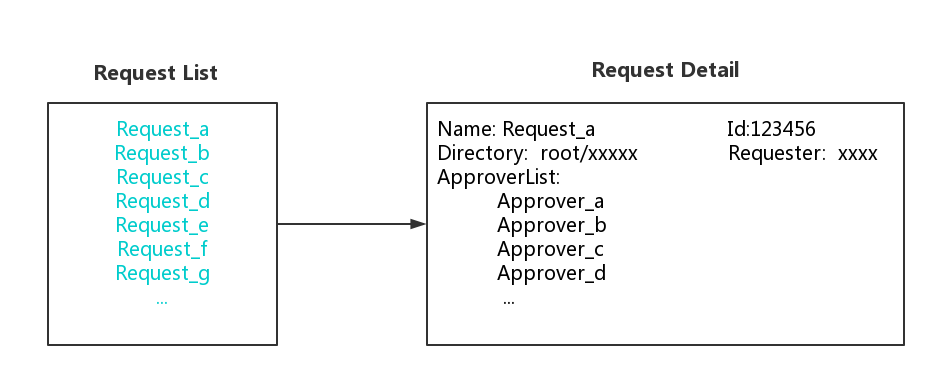
*getInfo*= (props) => {  
 const myProps = props || this.props;  
 return *getValue*(myProps, ['location', 'query', 'name'], name);  
*/\*这里返回的name是从props里面获得的，锚点通过this.getInfo选择,而props内的name是用户选择的时候放入props内的，筛选去除相邻重复的请求。因此数据里面的name永远是最新的\*/*  
};

在我们项目里，RxJS功能主要是：筛选isClient使得同构得以顺利进行，去除多余的异步请求（distinctUntilChanged），熔断select item和进行Ajax操作使得页面显示的永远只是用户最后选择的batch（onChange 和 loadInfo）。

通过如上代码，可以看到使用常规方式存在的Promise请求过多的问题，可以非常简单方便地使用RxJS解决。distinctUntilChanged方式可以筛选去除用户重复选择的batch数据流。同时与redux配合熔断select item和Ajax操作，使得页面显示的永远只是用户最后选择的batch。

**2.2.2 My Requests页面**

Request的列表如下。



用户选择列表中的Request，如果status为open或者manager approved的时候，则根据Request的ID读取API显示该Request可以approve的人员有哪些。

同样，我们先采用常规方法：

流程如下：

用户点击Request，判断status，若为open或者manager approved则触发API请求，显示approverList。

同样会存在例1中的问题：存在重复的Promise请求。

采用了RxJS之后可以很简单高效地解决这些问题。

this.subject  
 .map(({ match: { id }, requests }) => requests[id])  
 .filter(request => isClient && request)  
 .distinct()  
 .subscribe((request) => {  
 this.setState({ loading: true });  
 const promise = *get*(*url*(`ajax/${id}`));  
 promise.then(({ result }) => {  
 this.setState({ List: result });  
 }).finally(  
 () => this.setState({ loading: false }),  
 );  
 }  
 });

可以看到，在本例中，我们通过非常简便的方式就完成了客户端筛选（为同构服务）、重复Promise请求筛选（request不变的时候）。