

## 06 | 如何通过模块化、异步和观察做到动态加载？

2022-10-01 石川 来自北京



天下无鱼

<https://shikey.com/>

《JavaScript进阶实战课》

[课程介绍 >](#)



讲述：石川

时长 09:36 大小 8.77M




你好，我是石川。

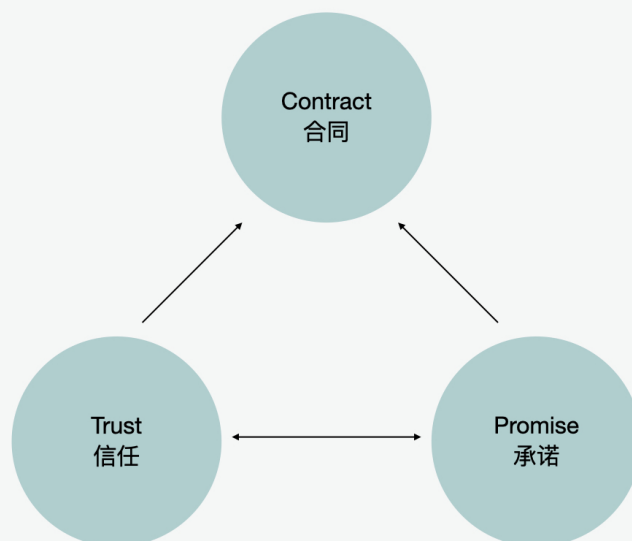
在前面几节讲函数式编程的课程里，我们了解了在函数式编程中副作用通常是来自于函数外部，多在输入的过程中会出现副作用。这实际上是从空间的角度来看的。

而今天这节课，我们会**从时间的角度**来看看异步中的事件如何能引起副作用，以及要如何管理这种副作用。

### 如何处理异步事件中的时间状态？

实际上，在函数式编程中我们在讨论异步的时候，经常会说到信任（**trustable**）和承诺（**promise**）。这个其实是源自于合同或者是契约法中的一个概念，而且它不只限于经典的合同，我们说的  **智能合约** 之类的概念中，**底层逻辑也都源于契约和共识**。

那么，为什么我们在处理异步时需要用到这个概念呢？下面我就先带你来看看在异步时，程序都会遇到哪些问题。



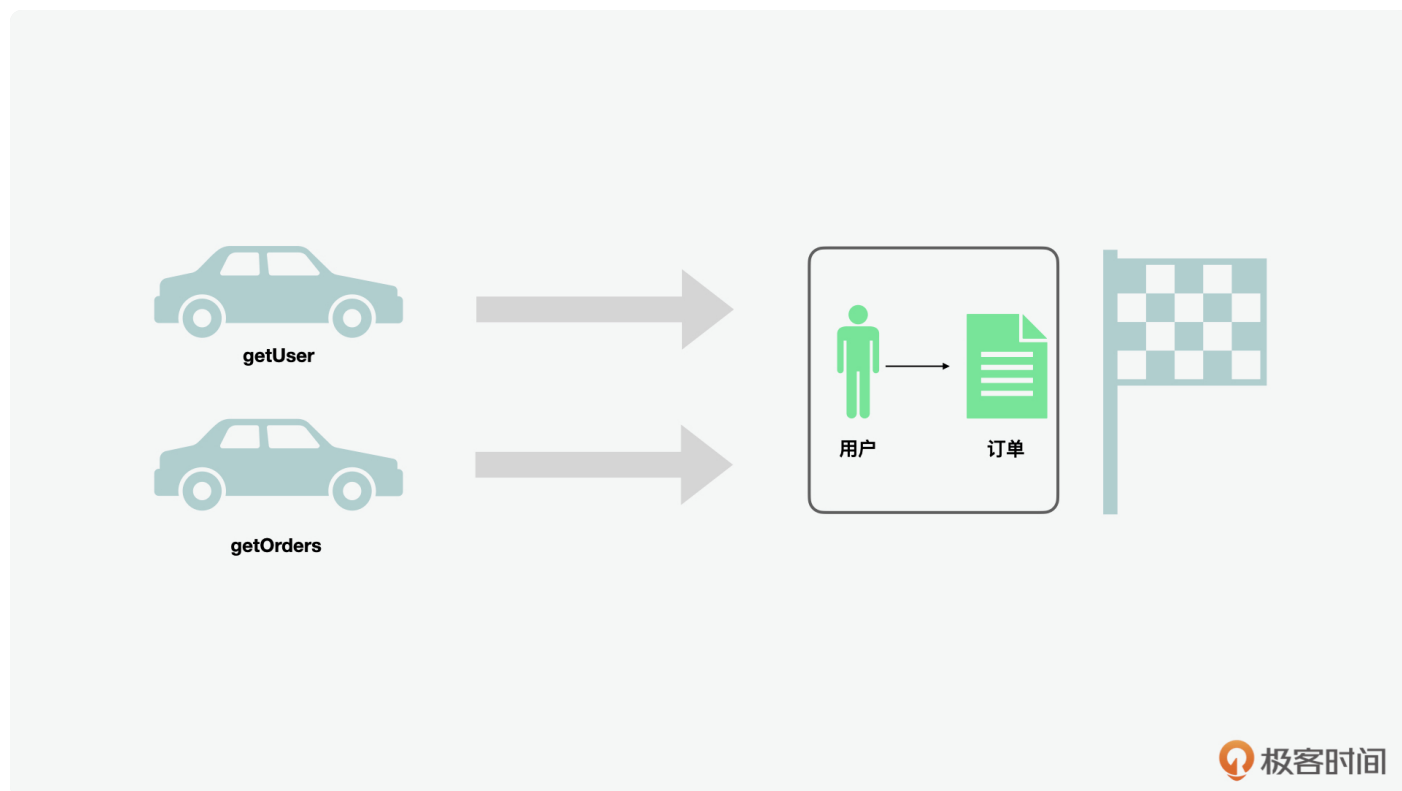
假设，我们有以下 `getUser` 和 `getOrders` 两个函数，分别通过用户 ID 来获取用户信息和订单信息。如果 `getUser` 先得到响应的话，那么它就没法获得订单信息。同样地，如果 `getOrders` 先得到响应的话，那么它就没办法获得用户信息。

这样一来，我们说这两个函数就形成了一个**竞争条件**（Race Condition）。

 复制代码

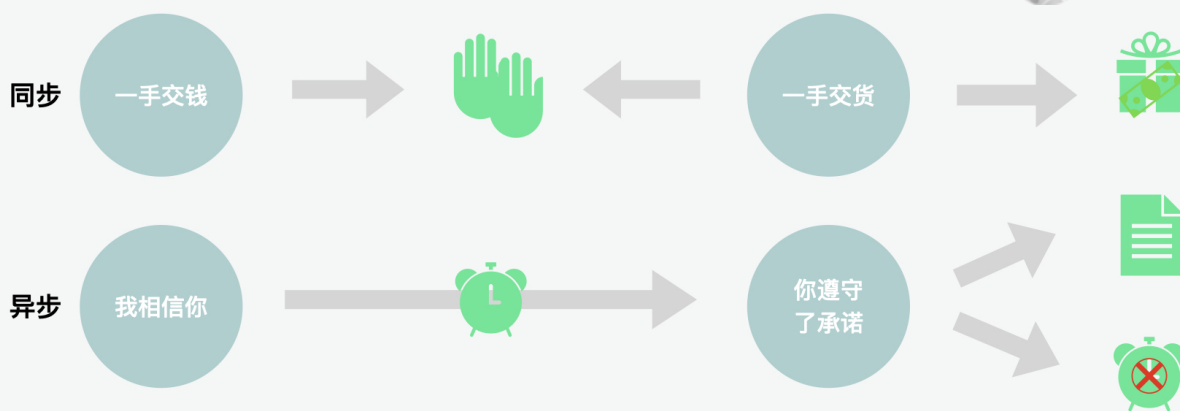
```
1 var user;
2
3 getUser( userId, function onUser(userProfile){
4     var orders = user ? user.orders : null;
5     user = userProfile;
6     if (orders) {
7         user.orders = orders;
8     }
9 } );
10
11 getOrders( userId, function onOrders(userOrders){
12     if (!user) {
13         user = {};
14     }
15     user.orders = userOrders;
16 } );
```

从下图也可以看出，无论是谁先到达都会出现问题。那么你可能会说，把它们一前一后分开处理不就行了吗？但是这样的话就没有办法做到并行，只能串行，而串行所花的总时间一般会高于并行。



在这里，时间就是状态。在同步操作中我们不需要考虑时间；而在异步中时间就产生了。**时间不仅仅是状态，也是最难管理的状态。**

这和信任和承诺又有什么关系呢？因为信任和承诺之间隔着的就是时间！还是以合同交易举例，如果是同步的话，相当于一手交钱、一手交货。而在异步中，则只能靠时间证明是否遵守了承诺。所以在 JavaScript 里，解决异步问题的工具就叫**承诺（promise）**。



可是，凭什么我们要相信一个承诺呢？在生活中的交易，大家通常会走个合同，剩下的交给时间来证明。而在 JavaScript 函数式编程当中，解决方案就是不用合同这么麻烦了，为了让你相信我的承诺，咱们干脆直接把时间给干掉。

是的，就是这么霸气。把时间干掉的方式就是按照同步的方式来处理异步，下面是一个例子：

[复制代码](#)

```
1 var userPromise = getUser( userId );
2 var ordersPromise = getOrders( userId );
3
4 userPromise.then( function onUser(user){
5     ordersPromise.then( function onOrders(orders){
6         user.orders = orders;
7     } );
8 } );
```

这样，即使是并行获取的用户和订单信息，在处理的时候我们也可以通过 **then** 按照同步处理时的先后顺序，来更新订单信息到用户对象上。

## 如何处理循环事件中的时间状态？

在函数式 + 响应式编程中，除了网络事件，还有更多的例子是通过去掉时间，比如循环或用户事件，都可以用类似同步的方式来处理异步事件。

举个例子，比如我们有生产者和消费者两个对象，消费者希望在生产者发生改变的时候，能随之映射出改变。这时候如果我们的生产者很“勤奋”，实时地在生产，消费者也可以实时地来消费。

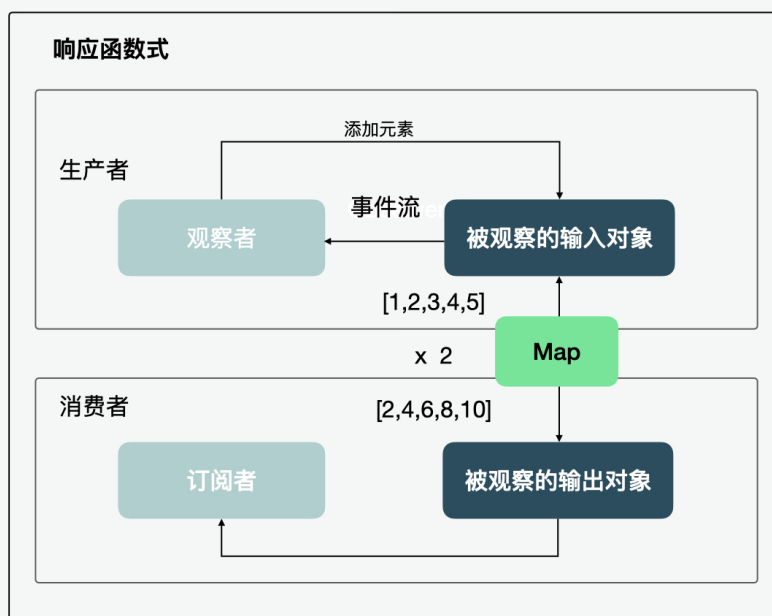


复制代码

```
1 // 勤奋生产者
2 var eagerProducer = [1,2,3,4,5];
3
4 // 消费者
5 var consumer = producer.map( function triple(v){
6     return v * 3;
7 } ); // [3,6,9,12,15];
8
```

但是如果有一个懒惰的生产者，消费者不知道在未来哪个时间该生产者会发生变化，那么要怎么办呢？

这时，我们就需要**把懒惰的生产者当做一个被观察对象**，每当它发生变化时，就随之做出反应，这就是“函数式中的异步模式”和“响应式中的观察者模式”的结合。



极客时间

下面是一个相对抽象的异步循环事件的例子。不过在现实当中，我们遇到的用户输入，比如鼠标的点击、键盘的输入等等的 DOM 事件也是异步的。所以这个时候，我们就可以用到“懒”这

个概念，根据用户反应来“**懒加载**”一些内容。

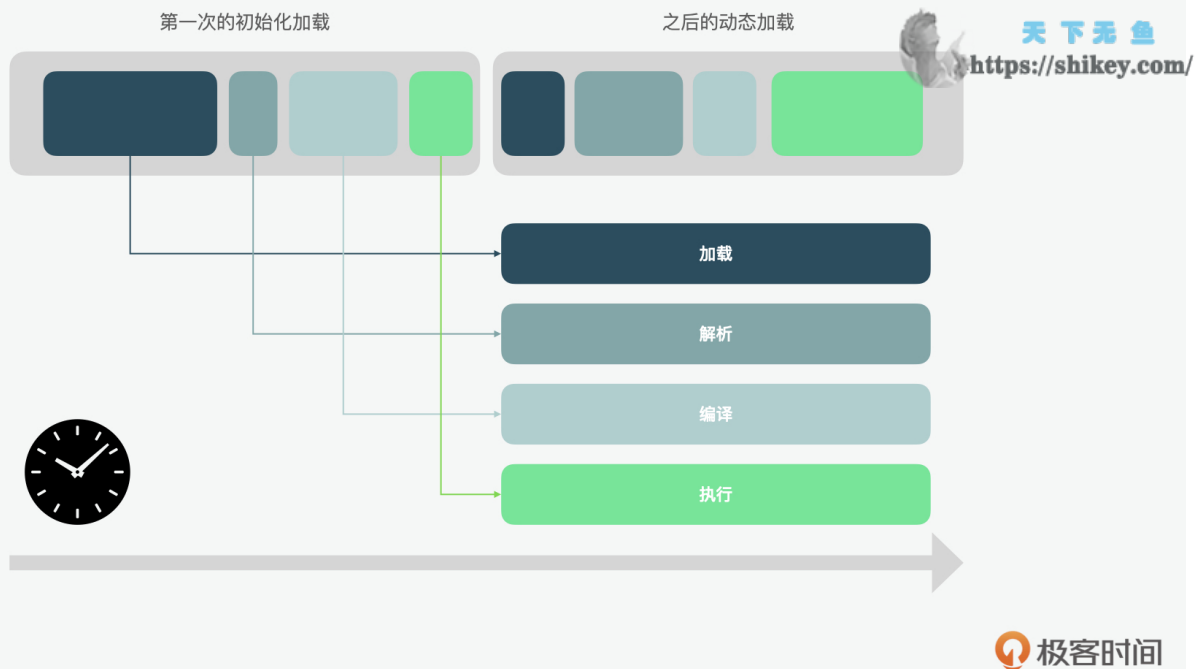


```
1 // 懒惰生产者
2 var producer = Rx.Observable.create( function onObserve(observer){
3     setInterval( function everySecond(){
4         observer.next( Math.random() );
5     }, 1000 );
6 } );
7
8 // 消费者
9 var consumer = producer.map( function triple(v){
10     return v * 3;
11 } );
12 consumer.subscribe( function onValue(v){
13     console.log( v );
14 } );
```

## 如何处理用户事件中的时间状态？

接着我们再从响应式和观察者模式延伸，来看看前端在处理页面上内容的动态加载时使用的一些方法。这就要说到**动态导入**了。

我们先来看看网页上的一个模块从加载到执行的顺序。可以看到，这个顺序大致分成了 4 个步骤，第一是加载，之后是解析、编译，最后是执行。如果是动态加载，就是在初始化之后，根据需求再继续加载。



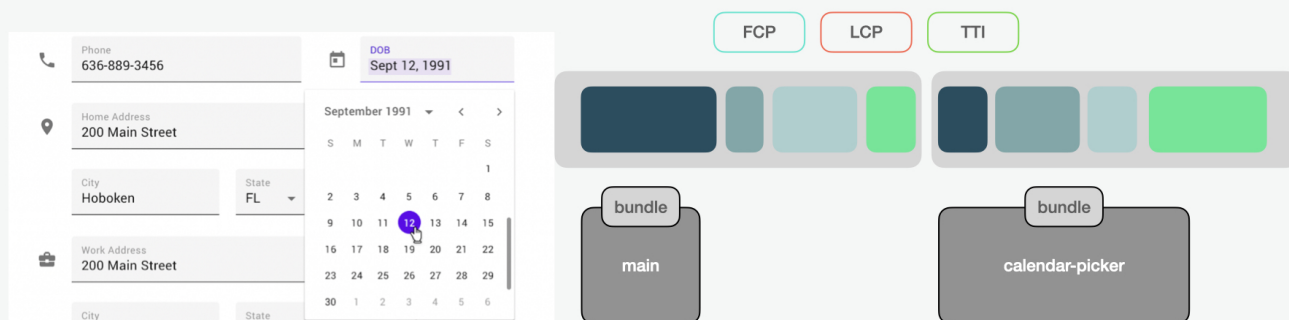
而说到动态导入，基本可以分成两类，一类是可视时加载（load on visibility），一种是交互时加载（load on interaction）。

**可视时加载**就是我们经常说的懒加载（Lazy loading），这种方式经常用在长页面当中。比如产品详情页，一般都是用讲故事的方式一步步介绍产品卖点，用图说话，最后再展示参数、一键购买以及加购物车的部分。所以也就是说我们不需要一上来就加载整个页面，而是当用户滑动到了某个部分的时候，再加载相关的内容。

**交互时加载**就是当用户和页面进行交互时，比如点击了某个按钮后，可能产生的加载。举个例子，有些应用中的日历，只有用户在进行特定操作的时候才会显示并且和用户交互。这样的模块，我们就可以考虑动态加载。

注意，这里有几个重要的指标。在初始化的加载中，我们关注的通常是首次渲染时间（FCP，First Contentful Paint）和最大内容渲染时间（LCP，Largest Contentful Paint），也就是页面首次加载的时候。在后续的动态加载中，我们关注的是首次交互时间（TTI，Time to Interactive），也就是当用户开始从首屏开始往下滑动，或者点击了某个按钮开启了日历弹窗的时候。



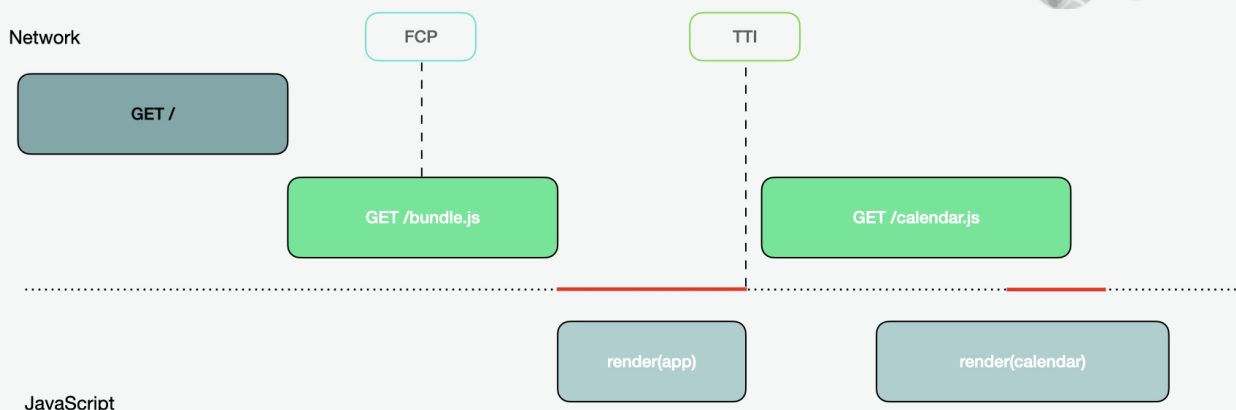


你可能会觉得，这样的优化只能省下一两百 KB 或几个 MB，是否值得？但是苍蝇腿也是肉，而且积少成多，当你在开发一个复杂的 Web 应用需要不断扩充模块的时候，这样的操作可能就会产生质和量上的变化。

然后在这个时候，我们通常会通过一些打包工具，比如用 Webpack 先加载核心的组件，渲染主程序，之后根据交互的需要，按需加载某个模块。

另外，对于动态的加载，其实也有很多三方的库可以支持，其中一个例子就是 React 中的 Suspense。如果是 Node 服务器端的加载渲染的话，也有 Loadable Components 这样的库可以用来参考。当然，如果你不使用这些三方的库，自己开发也可以，但是原理始终是类似的。





不过这里我还想说下，在使用动态导入前，一般应该先考虑预加载（pre-load）或预获取（pre-fetch）。

它们两个的区别是，前者是在页面开始加载时就提前开始加载后面需要用到的元素；后者是在页面的主要功能都加载完成后，再提前加载后面需要用到的素材。除非没法做到预加载和预获取，或者你加载的是三方的内容，不可控，不然的话，这些方式都可以带来比动态加载更好的用户体验。

那么，有没有没法儿做到、或者不适合做预加载的例子呢？也是有的，比如要预加载的内容过大，而且用户不一定会使用预加载的内容的时候。

这个时候如果你事先加载，一是会使用到用户的网络，占用了用户手机的存储空间；二是也会增加自己的 CDN 和服务器的资源消耗。这种情况下，就要用到动态加载了。

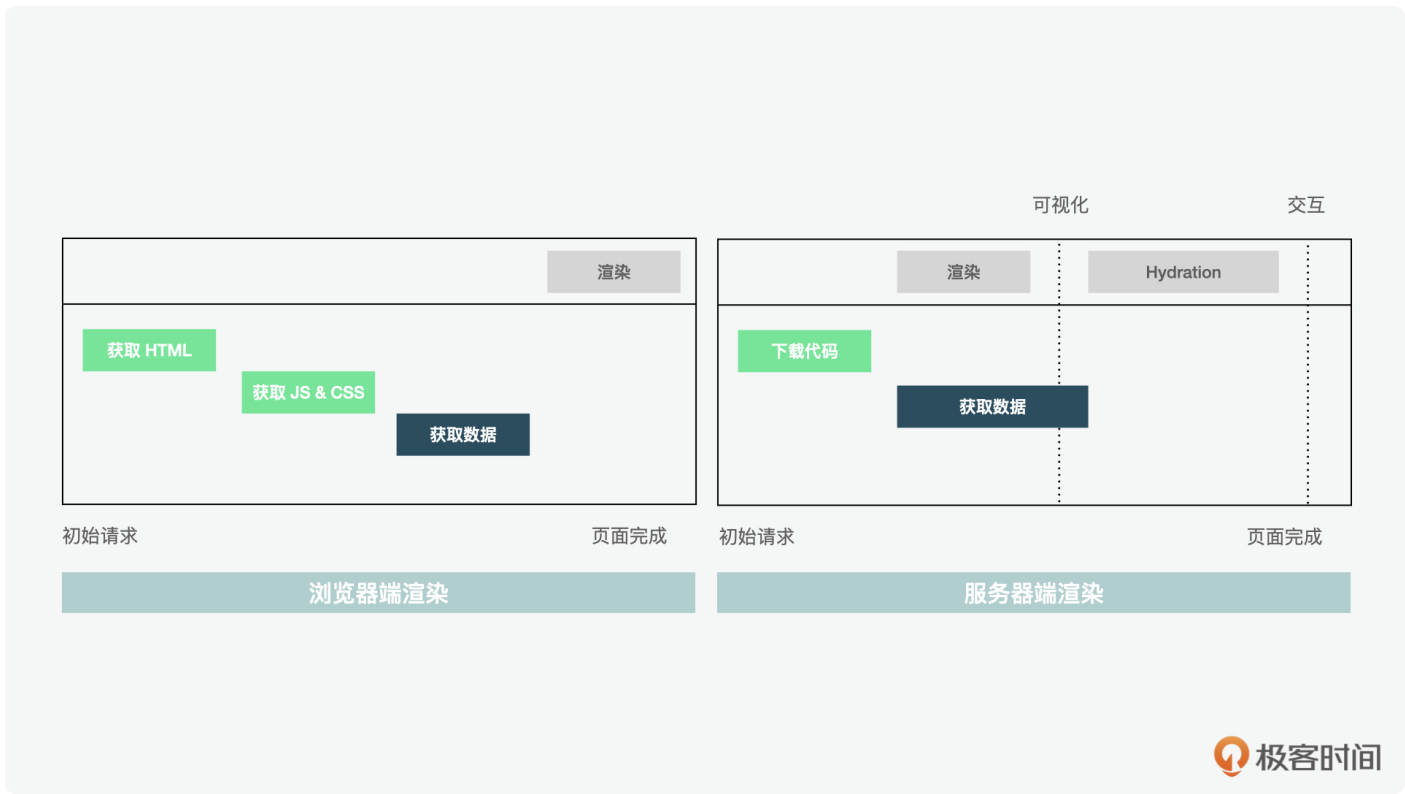
另外在进一步看动态加载前，我们还要了解两个基础概念，就是页面渲染的两种基础渲染模式。一种是**浏览器渲染**，一种是**服务器端渲染**。

首先，在客户端渲染（CSR，client side rendering）模式下，我们是先下载 HTML、JS 和 CSS，包括数据，所有的内容都下载完成，然后再开始渲染。

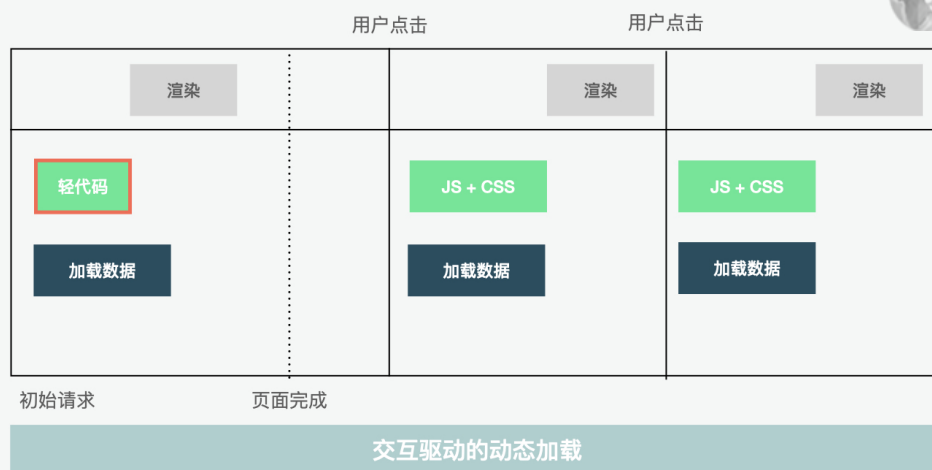
而 SSR 服务器端渲染（SSR，server side rendering）模式下，我们是先让用户能看到一个完整的页面，但是无法交互。只有等相关数据从服务器端加载和 hydrate 后，比如说一个按钮加上了的相关事件处理之后，才能交互。



这个方案看上去比 CSR 会好一些，但它也不是没有问题的。比如说，我们作为用户使用一些应用有时候，也会遇到类似的问题，就是我们在加载和 hydrate 前点击某个按钮的时候，就会发现某个组件没反应。



那么在交互驱动的动态加载中，上面这种问题怎么解决呢？比如 Google，他们会使用并且开源了一个叫 [JSAction](#) 的小工具，它的作用就是先加载一部分轻代码（tiny code），这部分代码可以“记住”用户的行为，然后根据用户的交互来加载组件，等加载完成再让组件执行之前“记住”的用户请求。这样就完美解决了上述问题。



## 总结

通过今天的学习，我们理解了函数式编程 + 响应式编程中，时间是一个状态，而且是一个最难管理的状态。而通过 **promise** 的概念我们可以消除时间，并且可以通过同步的方式来处理异步事件。

另外，通过观察者模式我们也可以更好地应对未知，通过行动来感知和响应，这样的加载方式，在应用的使用者达到一定规模化时候，可以减少不必要和大量的资源浪费。

## 思考题

我们说根据事件的动态加载可以起到降本增效的作用，那么你能说说你在前端开发中做资源加载设计、分析和优化的经验吗？

欢迎在留言区分享你的答案、交流学习心得或者提出问题，如果觉得有收获，也欢迎你把今天的内容分享给更多的朋友。

分享给需要的人，Ta购买本课程，你将得 18 元

生成海报并分享

上一篇 05 | map、reduce和monad如何围绕值进行操作？

下一篇 07 | 深入理解对象的私有和静态属性

## 精选留言 (1)

💬 写留言



lemon

2022-10-06 来自北京

“在初始化的加载中，我们关注的通常是首次渲染时间（CFP，First Contentful Paint）”，这里应该是 FCP

作者回复: 谢谢指正！是个勘误，我也编辑老师也说一声。

