第11讲 真实世界的并发编程

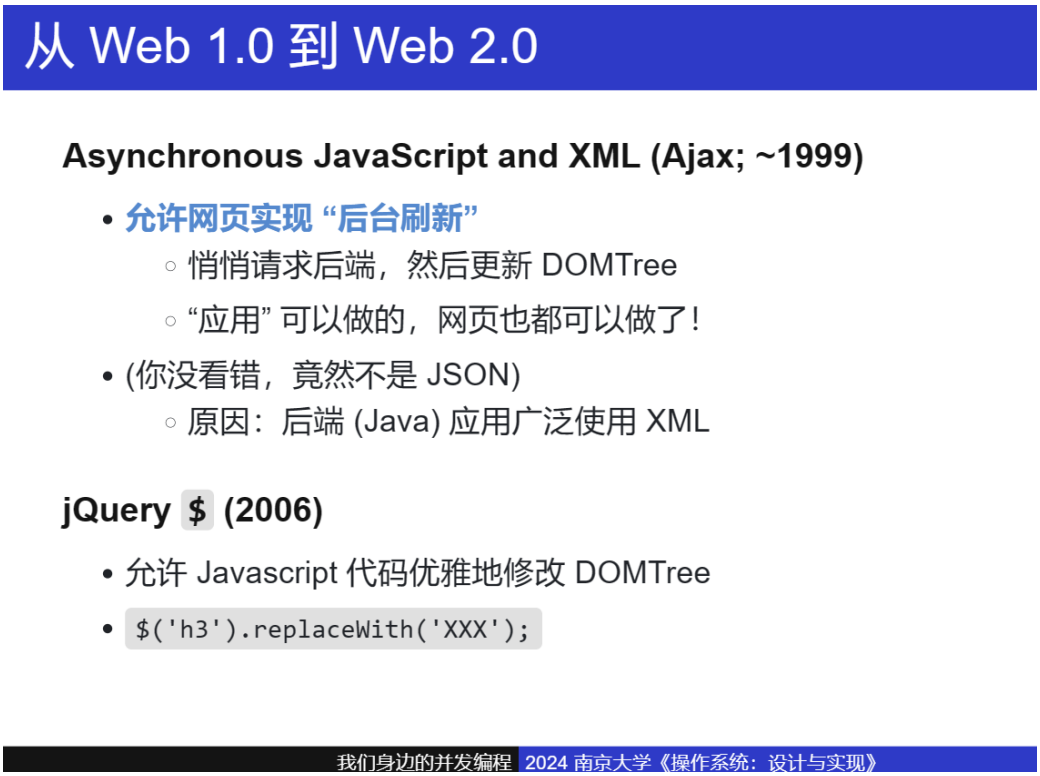
1. web中的并发编程

1. Web 1.0

这个时候没有CSS，你们想象没有 CSS 的时候我们怎么做网页呢？

- 我们有两个 html tag，一个叫table，table 是一个二维空间上的二维结构。当你有 table 的时候，这个 table 可以灵活的构造。比如说如果你想要构造一个像这样的页面的话，你会先把它切成两行，然后两行里面上面是它的这个导航条，下面是它的内容，然后在导航条上面你再分成几列。
- 然后你有大量的那个边距是 0 的表格，然后最终把那个界面给组合起来，然后在这个地方可能有一个长链接，然后你可以点击这个地方的按钮，这个就是 Web 1.0 时代，那个时候微软的 Internet Explorer 开始一家独大。
- 然后既然我刚才说到所有的网页是靠切图、贴图，就意味着是靠表格，这就意味着如果你要设计一个网页，你的设计师是很困难的，他会在比如说像 Photoshop 这样的软件里面先把那个底图给打好，比如说他要把他想象的界面画好以后，把它一个一个切出来，切成一个一个的图片，然后给文字留好位置，然后在那个地方放一个表格的格子，然后在那里填上文字，然后图片的地方把它切下来，最后就得到了一个完整的图片。

2. Ajax



然后在 1999 年的时候有了一项标志性的技术。

今天如果你在 Javascript 世界里面传送数据，你用的一定是json。但是在 1999 年的时候，我们传递在服务器的前端和后端交换的数据还是XML。

但是那个时候的事实后端用的是XML，所以有了一个 ajax 的API， ajax API 允许你在网页里，网页里面，在后台发送一个网页的请求，就像今天我们用的是fetch。这时候我的网页没有刷新，我刚才漏掉一个很重要的有意思的历史是，在过去，所有的网页都是静态的，如果你想要让它变一变，你就必须刷新整个页面。

也就是说如果你想要进入，比如说这边有个按钮叫做Messenger，你要进入 Messenger 的话，你要点一下，然后它会刷新这个页面，进入一个 Messenger 的页面，然后如果你要 check Email，你会点一下，然后它会刷新这个页面，会有一个全白重新加载的过程，但是 Ajax 实现了很重要的一个feature，是它可以在后台刷新，我可以去服务器问一问，唉，有没有新的消息，或者有没有新的邮件啊？如果有没有新的邮件，我的网页没有任何的变化。如果有新的邮件，我可以把这个邮件显示出来，然后给你一个弹出窗口，有新的邮件来了。

我们的 javascript 世界就是你看到的前端，我们看到网页上面所有东西都叫 DOM tree，它是一棵树，是由组件一层一层套出来的 DOM tree。

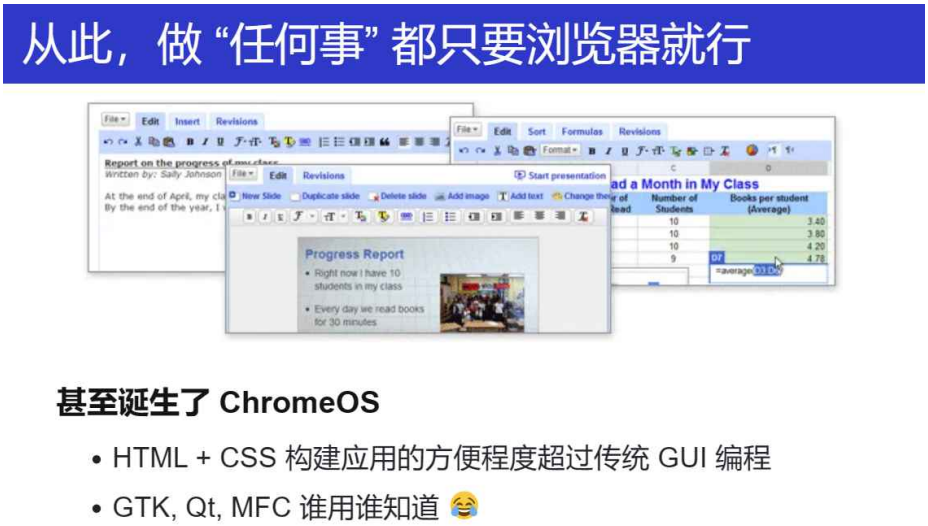
既然我们现在有 DOM tree，有 Js 的世界：

- 第一可以显示界面
- 第二可以后台悄悄地请求一个服务器

这意味着什么？这意味着任何你可以在桌面上实现的东西，都可以在浏览器里实现了。你只要能画一个方块，在方块里面填字，然后你可以去服务器上面请求服务器远端互联网的服务器任何的数据回来，然后根据请求回来的数据你可以执行代码，然后你可以把这个字换成任何你想要的字，然后你可

以把这个框移到任何你想要的位置，你可以在任何一个位置换增加一个新的框，这就是一个全新的世界。

所以 Ajax 是一个某种程度上颠覆性的技术，它把 Web 从 1.0 变成了 Web 2.0。 1.0 的时候是静态页面，静态刷新。 2.0 的时候，我们可以从网站上面拉一个邮件，拉一个及时的消息，这样所有的东西都可以放到网页里了。



后来我们发现这个用 HTML+CSS 构建应用的方便程度远远超过传统的一些方法。像如果你们写过传统的 QT， GTK，就真谁用谁知道，和这个比起来，它的开发效率是天天翻地覆的不一样。

然后我们来想一想在这个 Java script 的世界里面有没有并发编程？它并发在哪里呢？我们的并发编程来自于我们会要请求一个网页，但是这个网页不知道什么时候才能返回给我，对吧？我们并不知道是服务器挂了还是网络超时了。

所以我们的在 Web 2.0 时代其实也是有很复杂的并发的，因为你同时可能会请求多个来源的数据，比如说你可能会同时去请求你的消息，同时可能会去请求你的图片，也就是存在多个同时运行的ajax请求。可能请求已经发出去了，但是结果还没有返回来，这是不是就并发了？

同时，你不希望的是你在 fetch 一个页面，比如login的时候别的东西都不能做了，对吧？你还是希望他们能并行的。我去抓取一个图片和我的登录操作，他们都是很耗时间的操作，所以你希望他们是并行的，所以你希望的是把它们放在两个thread里。你希望的是好 T1 积极执行先 fetch login，如果登录失败，我就要把那个登录失败的消息显示在网页上。如果登录成功，我就要去再去抓取下一个网页，然后把它显示出来，把你个人的信息显示出来。

同时可能我们前端还有一个定时器，我们实现了一个小动画，每隔100ms的时候，它都会动一动，我们就需要另一个线程，它每隔100ms被唤醒一下，你去改一改那个东西的xy坐标。

所以你实际上是希望在 Java script 里面创建 3 个线程，一个线程管登陆，一个线程管抓取，另一个线程管你界面上面的一些定时刷新的元素。好，那就麻烦了，因为在 1990 年代的时候，你看九几年的时候，CPU 还很慢。

- 第一，线程的开销很大。
- 第二，线程同步很难写。

那并发编程如果很难的话，我们怎么帮助程序员来写 Javascript 里的并发程序呢？答案就是并发编程很难。不要紧，我们可以不并发。又回到互斥的时候，我们也是这样解决。

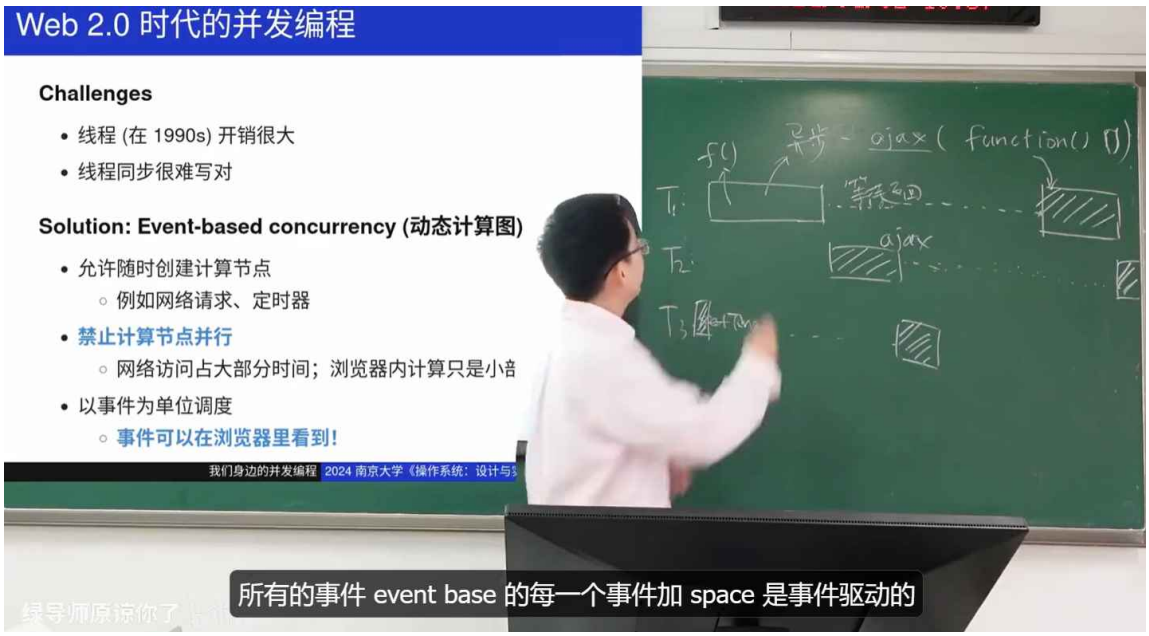
Java script 有一个基本的事件的编程模型，也就是说当你的任何一行 Javascript script 代码，比如说 fetch 开始执行的时候，那么这个世界上其他的 Java script 代码就不能再执行了。

就它好比仿佛还是有好几个线程，但是你可以还是像刚才一样，你创建三个线程，做三件事，但是 javascript 里面，在任何时候，当你一个函数开始运行了，或者一段代码开始运行了，那么这段代码就会一直持续的运行，并且其他人都不能运行了。（stop the world）

所以永远也不会有数据竞争。直到函数执行完毕，返回，其他人才开始执行。

但是我们这里就引入了一个机制，叫做异步操作。比如说Ajax，就是一种异步操作，比如说我可以 set 一个timer，等我 100 毫秒以后唤醒，就是一个异步操作。然后这个异步操作，刚才我们看到异步操作都会跟一个跟一段代码，这个代码叫回调函数，叫callback。

什么叫回调呢？就是我现在异步的Ajax，我要请求网络服务，那我不知道这个网络服务要多少时间才能返回，对吧？不知道多少时间才能返回，那我就等到返回的时候执行这段代码。然后这个时候这个异步事件就被放到 Jaw script 的运行时的环境里面去了，然后我们在等待返回的时候，其他线程的其他事件可以执行。



这样就没有任何的并发。但是你回过头来想，这个禁止计算并行好像没什么毛病，为什么呢？对于网页来说，绝大部分的时间都是花在等网络上，我请求一个服务可能要 100 毫秒，但是我比如说更改、更新一个浏览器的界面元素，比如说用jquery，他可能只要一微秒就行了。所以绝大部分的事件都非常的短小，因为绝大部分的事件都短小，所以它不能并行，也不是一个很大的麻烦。

所以我们就有了这样的一个 event base 的concurrency，你可以把它理解成是什么呢？是一个动态的计算图，你看每个事件就是动计算图上的一个节点。然后每当这个，比如说网页返回的时候，又有一

个新的计算节点被创建出来。如果我现在有同时有好几个计算节点可以执行，我必须执行完再执行一个。

混沌时代的计算图

“Callback hell (回调地狱)”

- 2024 年，教务系统里还能看到明文 💩⛰️

```
$.ajax({
    url: '/api/user',
    success: function(user) {
        $.ajax({
            url: `/api/user/${user.id}/friends`,
            success: function(friends) {
                $.ajax({
                    url: `/api/friend/${friends[0].id}`,
                    ...
                });
            },
            error: function(err) { ... }
        });
    }, ...
});
```


从 “前端” 到 “全栈”

ECMAScript 2015 (ES6)

- 一统第三方库 “军阀混战” 的局面
- 开源生态开始起飞

现代前端的代表作品

- Angular, React, Vue
- Express.js, Next.js
- Bootstrap, Tailwindcss
- Electron (vscode)
    - 2016 年，还用先烈 Github Atom 做过实验