本课时我们主要介绍高效操作 DOM 元素相关的内容。

什么是 DOM

DOM（Document Object Model，文档对象模型）是 JavaScript 操作 HTML 的接口（这里只讨论属于前端范畴的 HTML DOM），属于前端的入门知识，同样也是核心内容，因为大部分前端功能都需要借助 DOM 来实现，比如：

- 动态渲染列表、表格表单数据；
- 监听点击、提交事件；
- 懒加载一些脚本或样式文件；
- 实现动态展开树组件，表单组件级联等这类复杂的操作。

如果你查看过 DOM V3 标准，会发现包含多个内容，但归纳起来常用的主要由 3 个部分组成：

- DOM 节点
- DOM 事件
- 选择区域

选择区域的使用场景有限，一般用于富文本编辑类业务，我们不做深入讨论；**DOM 事件**有一定的关联性，将在下一课时中详细讨论；对于 **DOM 节点**，需与另外两个概念标签和元素进行区分：

- 标签是 HTML 的基本单位，比如 p、div、input；
- 节点是 DOM 树的基本单位，有多种类型，比如注释节点、文本节点；
- 元素是节点中的一种，与 HTML 标签相对应，比如 p 标签会对应 p 元素。

举例说明，在下面的代码中，“p”是标签， 生成 DOM 树的时候会产生两个节点，一个是元素节点 p，另一个是字符串为“亚里士朱德”的文本节点。

```
<p>亚里士朱德</p>
```

会框架更要会 DOM

有的前端工程师因为平常使用 Vue、React 这些框架比较多，觉得直接操作 DOM 的情况比较少，认为熟悉框架就行，不需要详细了解 DOM。这个观点对于初级工程师而言确实如此，能用框架写页面就算合格。

但对于屏幕前想成为高级/资深前端工程师的你而言，只会使用某个框架或者能答出 DOM 相关面试题，这些肯定是不够的。恰恰相反，作为高级/资深前端工程师，不仅应该对 DOM 有深入的理解，还应该能够借此开发框架插件、修改框架甚至能写出自己的框架。

因此，这一课时我们就深入了解 DOM，谈谈如何高效地操作 DOM。

为什么说 DOM 操作耗时

要解释 DOM 操作带来的性能问题，我们不得不提一下**浏览器的工作机制**。

线程切换

如果你对浏览器结构有一定了解，就会知道浏览器包含渲染引擎（也称浏览器内核）和 JavaScript 引擎，它们都是单线程运行。单线程的优势是开发方便，避免多线程下的死锁、竞争等问题，劣势是失去了并发能力。

浏览器为了避免两个引擎同时修改页面而造成渲染结果不一致的情况，增加了另外一个机制，这两个引擎具有互斥性，也就是说在某个时刻只有一个引擎在运行，另一个引擎会被阻塞。操作系统在进行线程切换的时候需要保存上一个线程执行时的状态信息并读取下一个线程的状态信息，俗称**上下文切换**。而这个操作相对而言是比较耗时的。

每次 DOM 操作就会引发线程的上下文切换——从 JavaScript 引擎切换到渲染引擎执行对应操作，然后再切换回 JavaScript 引擎继续执行，这就带来了**性能损耗**。单次切换消耗的时间是非常少的，但是如果频繁地大量切换，那么就会产生性能问题。

比如下面的测试代码，循环读取一百万次 DOM 中的 body 元素的耗时是读取 JSON 对象耗时的 10 倍。

```
// 测试次数：一百万次
const times = 1000000
// 缓存body元素
console.time('object')
let body = document.body
// 循环赋值对象作为对照参考
for(let i=0;i<times;i++) {
  let tmp = body
}
console.timeEnd('object')// object: 1.77197265625ms

console.time('dom')
// 循环读取body元素引发线程切换
for(let i=0;i<times;i++) {
  let tmp = document.body
}
console.timeEnd('dom')// dom: 18.302001953125ms
```

虽然这个例子比较极端，循环次数有些夸张，但如果在循环中包含一些复杂的逻辑或者说涉及到多个元素时，就会造成不可忽视的性能损耗。

重新渲染

另一个更加耗时的因素是元素及样式变化引起的再次渲染，在渲染过程中最耗时的两个步骤为**重排**（Reflow）与**重绘**（Repaint）。

浏览器在渲染页面时会将 HTML 和 CSS 分别解析成 DOM 树和 CSSOM 树，然后合并进行排布，再绘制成我们可见的页面。如果在操作 DOM 时涉及到元素、样式的修改，就会引起渲染引擎重新计算样式生成 CSSOM 树，同时还有可能触发对元素的重新排布（简称“重排”）和重新绘制（简称“重绘”）。

可能会影响到其他元素排布的操作就会引起重排，继而引发重绘，比如：

- 修改元素边距、大小
- 添加、删除元素
- 改变窗口大小

与之相反的操作则只会引起重绘，比如：

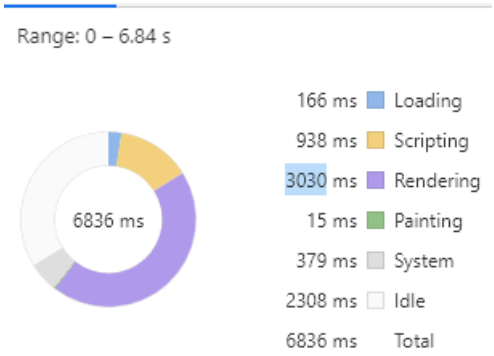
- 设置背景图片
- 修改字体颜色
- 改变 visibility 属性值

如果想了解更多关于重绘和重排的样式属性，可以参看这个网址：https://csstriggers.com/。

下面是两段验证代码，我们通过 Chrome 提供的性能分析工具来对渲染耗时进行分析。

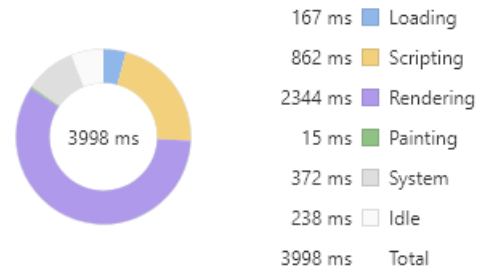
第一段代码，通过修改 div 元素的边距来触发重排，渲染耗时（粗略地认为渲染耗时为紫色 Rendering 事件和绿色 Painting 事件耗时之和）3045 毫秒。

```
const times = 100000
let html = ''
for(let i=0;i<times;i++) {
  html+= `<div>${i}</div>`
}
document.body.innerHTML += html
const divs = document.querySelectorAll('div')
Array.prototype.forEach.call(divs, (div, i) => {
  div.style.margin = i % 2 ? '10px' : 0;
})
```



第二段代码，修改 div 元素字体颜色来触发重绘，得到渲染耗时 2359 ms。

```
const times = 100000
let html = ''
for(let i=0;i<times;i++) {
  html+= `<div>${i}</div>`
}
document.body.innerHTML += html
const divs = document.querySelectorAll('div')
Array.prototype.forEach.call(divs, (div, i) => {
  div.style.color = i % 2 ? 'red' : 'green';
})
```



从两段测试代码中可以看出，重排渲染耗时明显高于重绘，同时两者的 Painting 事件耗时接近，也印证了重排会导致重绘。

如何高效操作 DOM

明白了 DOM 操作耗时之处后，要提升性能就变得很简单了，反其道而行之，减少这些操作即可。

在循环外操作元素

比如下面两段测试代码对比了读取 1000 次 JSON 对象以及访问 1000 次 body 元素的耗时差异，相差一个数量级。

```
const times = 10000;
console.time('switch')
for (let i = 0; i < times; i++) {
  document.body === 1 ? console.log(1) : void 0;
}
console.timeEnd('switch') // 1.873046875ms
var body = JSON.stringify(document.body)
console.time('batch')
for (let i = 0; i < times; i++) {
  body === 1 ? console.log(1) : void 0;
}
console.timeEnd('batch') // 0.846923828125ms
```

当然即使在循环外也要尽量减少操作元素，因为不知道他人调用你的代码时是否处于循环中。

批量操作元素

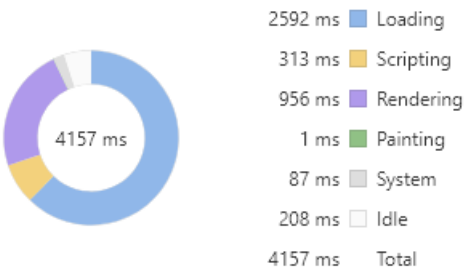
比如说要创建 1 万个 div 元素，在循环中直接创建再添加到父元素上耗时会非常多。如果采用字符串拼接的形式，先将 1 万个 div 元素的 html 字符串拼接成一个完整字符串，然后赋值给 body 元素的 innerHTML 属性就可以明显减少耗时。

```
const times = 10000;
console.time('createElement')
for (let i = 0; i < times; i++) {
  const div = document.createElement('div')
  document.body.appendChild(div)
}
console.timeEnd('createElement')// 54.964111328125ms
console.time('innerHTML')
let html=''
for (let i = 0; i < times; i++) {
  html+='<div></div>'
}
document.body.innerHTML += html // 31.919921875ms
console.timeEnd('innerHTML')
```

虽然通过修改 innerHTML 来实现批量操作的方式效率很高，但它并不是万能的。比如要在此基础上实现事件监听就会略微麻烦，只能通过事件代理或者重新选取元素再进行单独绑定。批量操作除了用在创建元素外也可以用于修改元素属性样式，比如下面的例子。

创建 2 万个 div 元素，以单节点树结构进行排布，每个元素有一个对应的序号作为文本内容。现在通过 style 属性对第 1 个 div 元素进行 2 万次样式调整。下面是直接操作 style 属性的代码：

```
const times = 20000;
let html = ''
for (let i = 0; i < times; i++) {
  html = `<div>${i}${html}</div>`
}
document.body.innerHTML += html
const div = document.querySelector('div')
for (let i = 0; i < times; i++) {
  div.style.fontSize = (i % 12) + 12 + 'px'
  div.style.color = i % 2 ? 'red' : 'green'
  div.style.margin = (i % 12) + 12 + 'px'
}
```



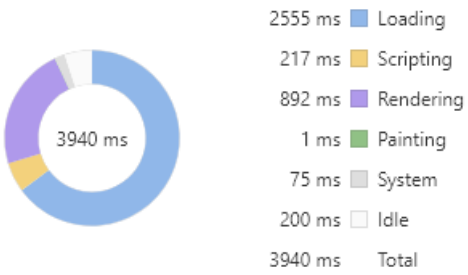
如果将需要修改的样式属性放入 JavaScript 数组，然后对这些修改进行 reduce 操作，得到最终需要的样式之后再设置元素属性，那么性能会提升很多。代码如下：

```
const times = 20000;
let html = ''
for (let i = 0; i < times; i++) {
  html = `<div>${i}${html}</div>`
}
document.body.innerHTML += html

let queue = [] //  创建缓存样式的数组
let microTask // 执行修改样式的微任务
const st = () => {
  const div = document.querySelector('div')
  // 合并样式
  const style = queue.reduce((acc, cur) => ({...acc, ...cur}), {})
  for(let prop in style) {
    div.style[prop] = style[prop]
  }
  queue = []
  microTask = null
}
const setStyle = (style) => {
  queue.push(style)
  // 创建微任务
  if(!microTask) microTask = Promise.resolve().then(st)
```

```
}
for (let i = 0; i < times; i++) {
  const style = {
    fontSize: (i % 12) + 12 + 'px',
    color: i % 2 ? 'red' : 'green',
    margin:  (i % 12) + 12 + 'px'
  }
  setStyle(style)
}
```

从下面的耗时占比图可以看到，紫色 Rendering 事件耗时有所减少。



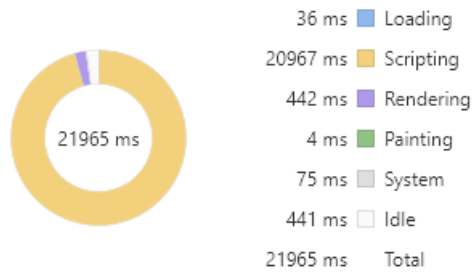
virtualDOM 之所以号称高性能，其实现原理就与此类似。

缓存元素集合

比如将通过选择器函数获取到的 DOM 元素赋值给变量，之后通过变量操作而不是再次使用选择器函数来获取。

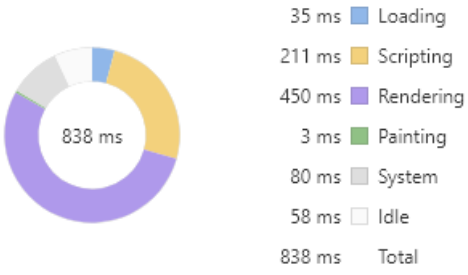
下面举例说明，假设我们现在要将上面代码所创建的 1 万个 div 元素的文本内容进行修改。每次重复使用获取选择器函数来获取元素，代码以及时间消耗如下所示。

```
for (let i = 0; i < document.querySelectorAll('div').length; i++) {
  document.querySelectorAll(`div`)[i].innerText = i
}
```



如果能够将元素集合赋值给 JavaScript 变量，每次通过变量去修改元素，那么性能将会得到不小的提升。

```
const divs = document.querySelectorAll('div')
for (let i = 0; i < divs.length; i++) {
  divs[i].innerText = i
}
```



对比两者耗时占比图可以看到，两者的渲染时间较为接近。但缓存元素的方式在黄色的 Scripting 耗时上具有明显优势。

总结

本课时从深入理解 DOM 的必要性说起，然后分析了 DOM 操作耗时的原因，最后再针对这些原因提出了可行的解决方法。

除了这些方法之外，还有一些原则也可能帮助我们提升渲染性能，比如：

- 尽量不要使用复杂的匹配规则和复杂的样式，从而减少渲染引擎计算样式规则生成 CSSOM 树的时间；
- 尽量减少重排和重绘影响的区域；
- 使用 CSS3 特性来实现动画效果。

希望你首先能理解原因，然后记住这些方法和原则，编写出高性能代码。

最后布置一道思考题：说一说你还知道哪些提升渲染速度的方法和原则？