《前端高手进阶》

朱德龙 前中兴软创主任工程师

— 拉勾教育出品 —

第02讲：如何高效操作 DOM

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

什么是 DOM

- 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

什么是 DOM

DOM V3 标准主要由 3 个部分组成

- DOM 节点
- DOM 事件
- 选择区域

什么是 DOM

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 事件和**终止控制器**有一定的关联性，将在下一课时中详细讨论

什么是 DOM

DOM 节点概念区分

- 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

什么是 DOM


```
<p>亚里士朱德</p>
```

“p” 是标签，生成 DOM 树的时候会产生两个节点

- 元素节点 p
- 字符串为“亚里士朱德”的文本节点

会框架更要会 DOM

有的前端工程师觉得认为熟悉框架就行，不需要详细了解 DOM

但作为**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

为什么说 DOM 操作耗时

线程切换

- **优势**：开发方便，避免多线程下的死锁、竞争等问题
- **劣势**：失去了并发能力

为什么说 DOM 操作耗时

线程切换

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

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**

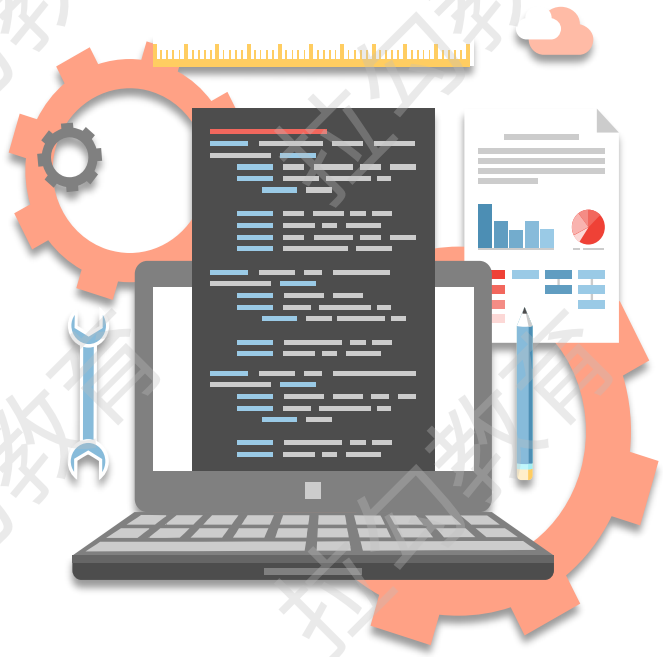
而这个操作相对而言是比较耗时的

为什么说 DOM 操作耗时

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**



为什么说 DOM 操作耗时


```
// 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
```

为什么说 DOM 操作耗时

重新渲染

渲染过程中最耗时的两个步骤为**重排（Reflow）**与**重绘（Repaint）**

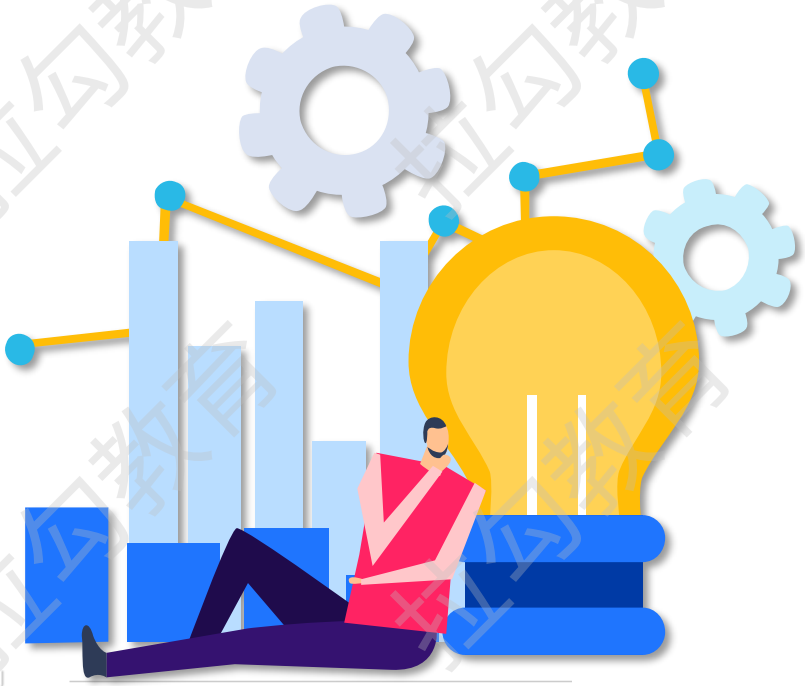
渲染页面时会将 HTML 和 CSS 分别解析成 DOM 树和 CSSOM 树
然后合并进行排布，再绘制成我们可见的页面
如果在操作 DOM 时涉及到**元素、样式的修改**
就会引起渲染引擎重新计算样式生成 CSSOM 树
同时还有可能触发对元素的重新排布（简称“重排”）和重新绘制（简称“重绘”）

为什么说 DOM 操作耗时

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



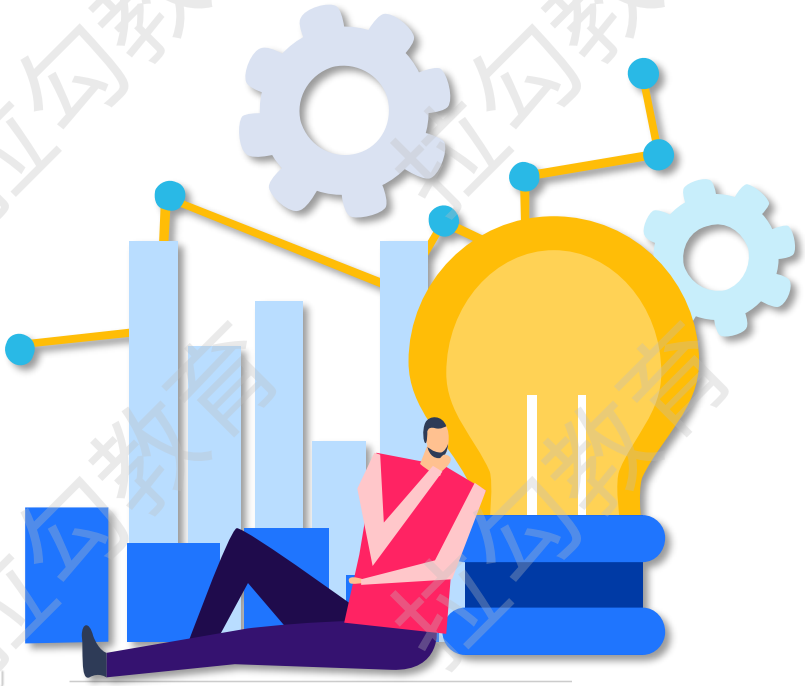
为什么说 DOM 操作耗时

与之相反的操作则只会引起重绘

比如：

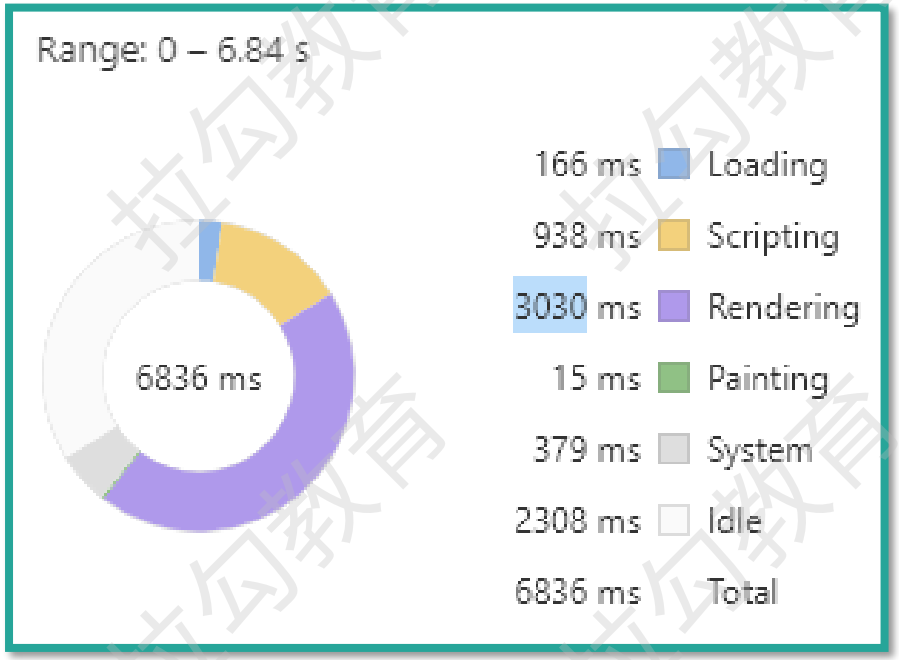
- 设置背景图片
- 修改字体颜色
- 改变 visibility 属性值

https://csstriggers.com/



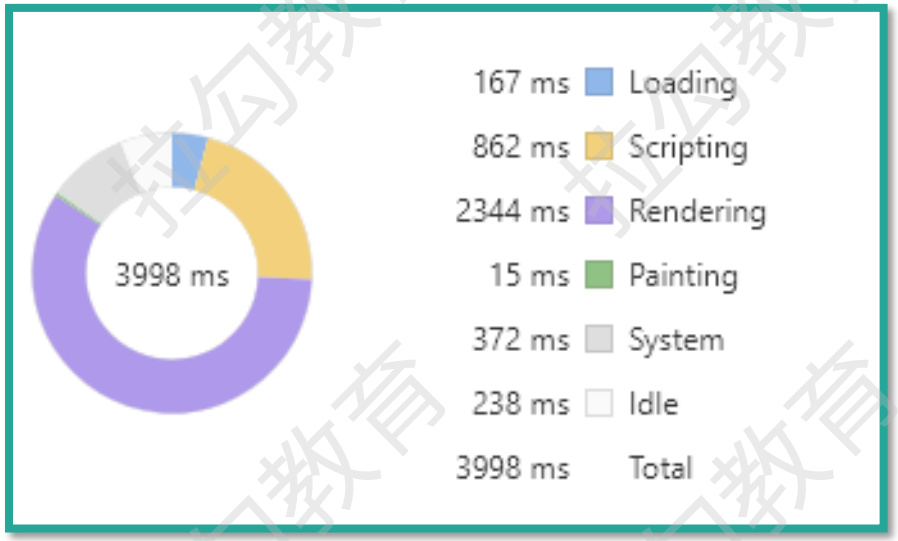
为什么说 DOM 操作耗时


```
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;
})
```



为什么说 DOM 操作耗时


```
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';
})
```



如何高效操作 DOM

在循环外操作元素

```
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
```

如何高效操作 DOM

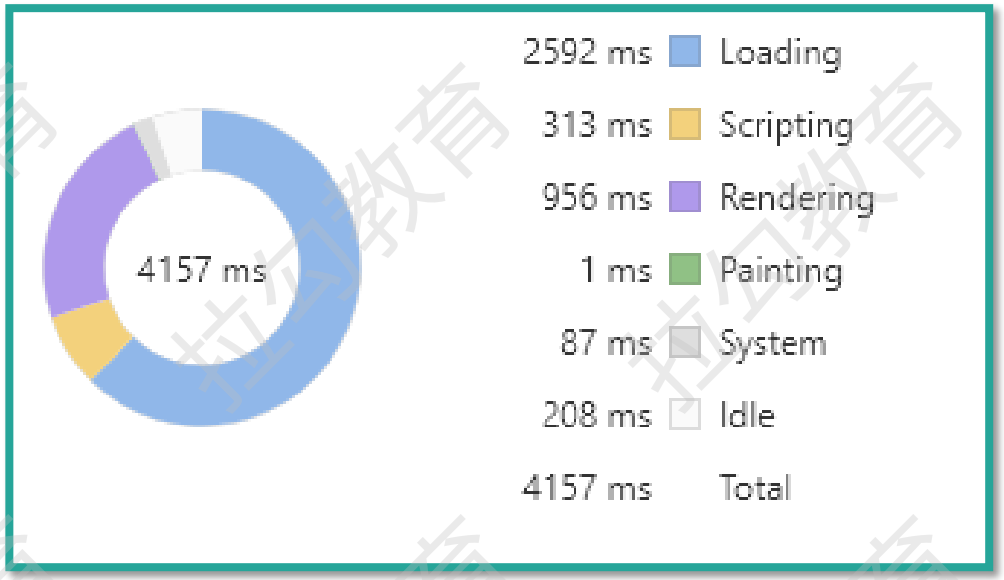
批量操作元素

```
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')
```

如何高效操作 DOM

批量操作元素

```
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'
}
```

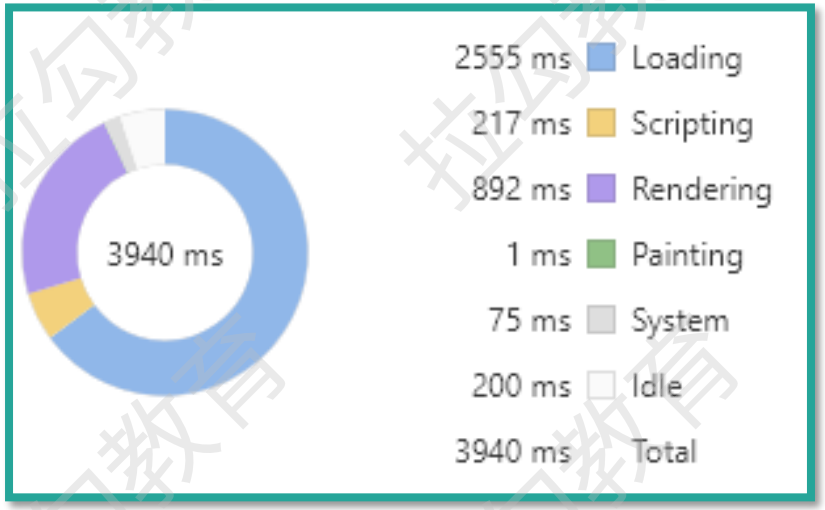


如何高效操作 DOM

拉勾教育
互联网人实战大学

批量操作元素

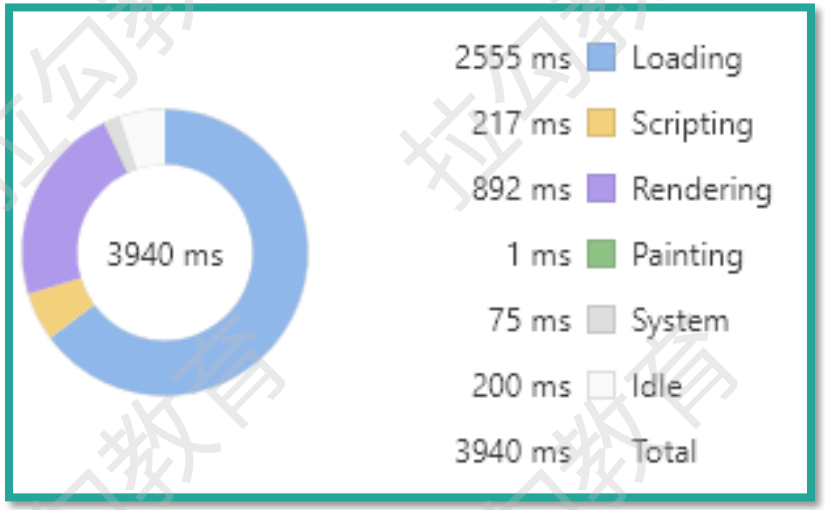
```
const times = 20000;
let html = ''
for (let i = 0; i < times; i++) {
  html = `<div>${i}${html}</div>`
}
document.body.innerHTML += html

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) {
```



如何高效操作 DOM

批量操作元素

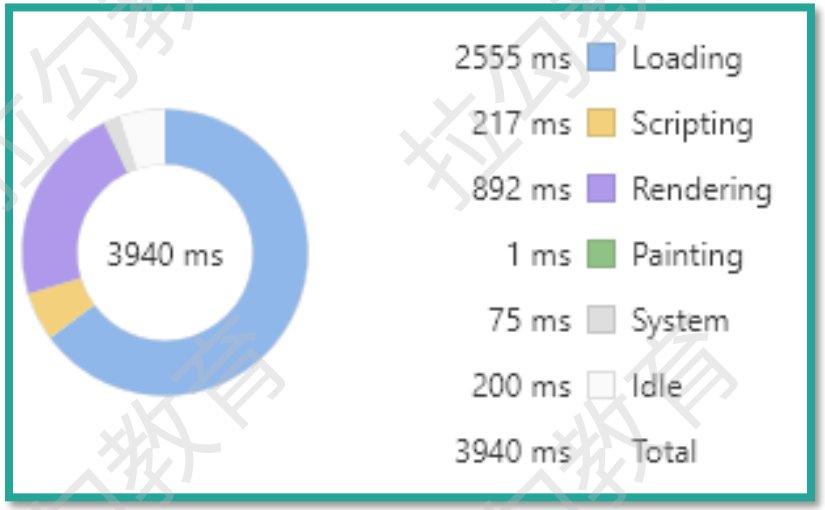
```
  for(let prop in style) {
    div.style[prop] = style[prop]
  }
  queue = []
  microTask = null
}
const setStyle = (style) => {
  queue.push(style)
  // 创建微任务
  if(!microTask) microTask = Promise.resolve().then(st)
}
for (let i = 0; i < times; i++) {
  const style = {
    fontSize: (i % 12) + 12 + 'px',
```



如何高效操作 DOM

批量操作元素

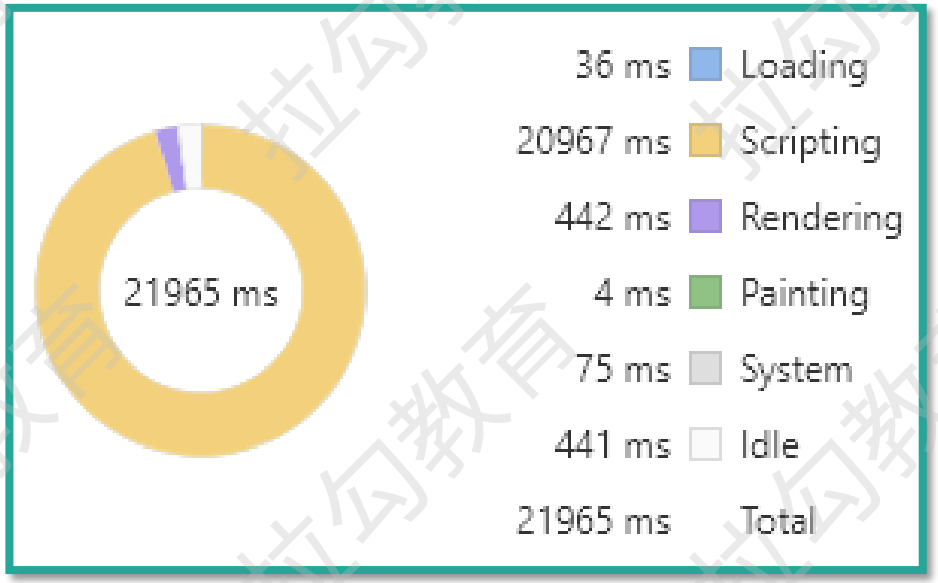
```
}
const setStyle = (style) => {
  queue.push(style)
  // 创建微任务
  if(!microTask) microTask = Promise.resolve().then(st)
}
for (let i = 0; i < times; i++) {
  const style = {
    fontSize: (i % 12) + 12 + 'px',
    color: i % 2 ? 'red' : 'green',
    margin:  (i % 12) + 12 + 'px'
  }
  setStyle(style)
}
```



如何高效操作 DOM

缓存元素集合

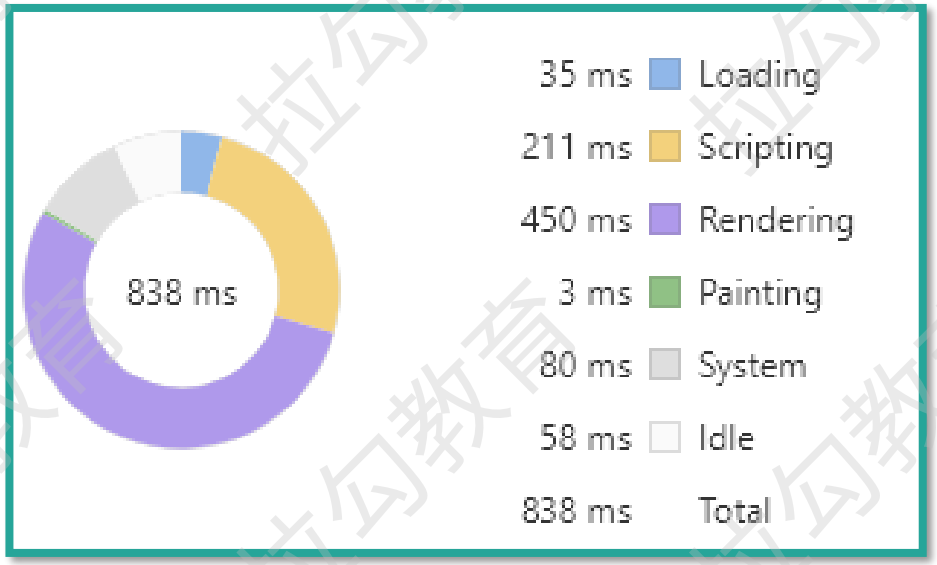
```
for (let i = 0; i < document.querySelectorAll('div').length; i++) {
  document.querySelectorAll(`div`)[i].innerText = i
}
```



如何高效操作 DOM

缓存元素集合

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



总结

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

总结

- 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

总结

说一说你还知道哪些提升渲染速度的方法和原则？

Next：第03讲《3 个使用场景助你用好 DOM 事件》