javaScript高级

一、 this指向

> 在 JavaScript 中，`this` 的指向取决于函数的调用方式。以下是几种常见的 `this` 指向情况：

1. 全局作用域中的函数调用

- `this` **指向：全局对象**
- 在非严格模式下，全局作用域中的函数的 `this` 指向 `window`（浏览器环境）或 `global`（Node.js 环境）。
- 在严格模式下，`this` 会是 `undefined`。

```
function showThis() {
    console.log(this);
}

showThis(); // 浏览器中输出：window
```

```
'use strict';
function showThis() {
    console.log(this);
}

showThis(); // 输出：undefined
```

2. 对象方法调用

- `this` **指向：调用该方法的对象**
- 如果函数作为对象的方法调用，`this` 会指向调用该方法的对象。

```
const obj = {
    name: 'Alice',
    greet: function() {
        console.log(this.name);
    }
};

obj.greet(); // 输出：Alice，this 指向 obj 对象
```

3. 构造函数调用

- `this` **指向：新创建的实例对象**
- 当使用 `new` 关键字调用构造函数时，`this` 指向由该构造函数创建的新的实例对象。

```
function Person(name) {
    this.name = name;
}

const person1 = new Person('Bob');
console.log(person1.name); // 输出: Bob
```

4. `call/apply/bind` 调用

- **`this` 指向：指定的对象**
- 通过 `call` 或 `apply` 方法调用函数时，`this` 被显式地指定为第一个参数。
- `bind` 返回一个新的函数，该函数在调用时 `this` 始终指向指定的对象。

```
function greet() {
    console.log(this.name);
}

const obj1 = { name: 'Alice' };
const obj2 = { name: 'Bob' };

greet.call(obj1); // 输出: Alice
greet.apply(obj2); // 输出: Bob

const boundGreet = greet.bind(obj1);
boundGreet(); // 输出: Alice
```

5. 箭头函数

- **`this` 指向：定义箭头函数时的外层作用域**
- 箭头函数没有自己的 `this`，它会继承定义时外层作用域的 `this`。

```
const obj = {
    name: 'Alice',
    greet: () => {
        console.log(this.name);
    }
};

obj.greet(); // 输出: undefined (因为 this 指向的是全局作用域)
```

但是在对象或方法中嵌套箭头函数时，它会继承外层函数的 `this`。

```
const obj = {
    name: 'Alice',
    greet: function() {
        const inner = () => {
            console.log(this.name);
        };
        inner();
    }
};

obj.greet(); // 输出: Alice (因为箭头函数继承了外层函数的 this)
```

6. 事件处理函数

- `this` **指向：触发事件的 DOM 元素**
- 在事件处理函数中，`this` 默认指向触发该事件的 DOM 元素。

```
const button = document.querySelector('button');
button.addEventListener('click', function() {
  console.log(this); // 输出: 被点击的 button 元素
});
```

总结：

- 普通函数调用：全局对象（严格模式下为 `undefined`）。
- 对象方法调用：调用该方法的对象。
- 构造函数调用：新创建的实例对象。
- `call`/`apply`/`bind`：显式指定的对象。
- 箭头函数：继承定义时的外层作用域。
- 事件处理函数：触发事件的 DOM 元素。

二、this绑定规则的优先级

> 在 JavaScript 中，`this` 的绑定规则存在优先级，不同调用方式之间可能会产生冲突。以下是 `this` 绑定规则的优先级从高到低的顺序：

1. `new` 绑定（构造函数调用）

当使用 `new` 关键字调用一个函数时，会创建一个新的对象，并将 `this` 绑定到这个新创建的对象上。此时 `new` 绑定的优先级最高，其他绑定规则会被忽略。

```
function Person(name) {
    this.name = name;
}

const person = new Person('Alice'); // this 指向新创建的 person 对象


// 3.2. new优先级高于bind
function foo() {
    console.log("foo:", this)
}
var bindFn = foo.bind("aaa")
new bindFn()
```

2. `call` / `apply` / `bind` 显式绑定

`call`、`apply` 和 `bind` 可以显式地指定 `this` 的绑定对象。显式绑定的优先级次于 `new` 绑定。如果函数是通过 `new` 关键字调用的，显式绑定会被忽略。

- `call` 和 `apply` 会立即执行函数，并传递指定的 `this`。
- `bind` 会创建一个新的函数，`this` 被永久绑定到指定的对象，调用时不再改变。

```
function greet() {
    console.log(this.name);
}

const obj = { name: 'Bob' };
greet.call(obj); // this 指向 obj，输出: Bob

// 4.bind/apply优先级
// bind优先级高于apply/call
function foo() {
    console.log("foo:", this)
}
var bindFn = foo.bind("aaa")
bindFn.call("bbb")
```

3. 隐式绑定（对象方法调用）

如果函数作为某个对象的方法调用，`this` 会隐式绑定到该对象上。隐式绑定优先级低于 `new` 和显式绑定。

```
const obj = {
    name: 'Charlie',
    greet: function() {
        console.log(this.name);
    }
};

obj.greet(); // this 指向 obj，输出: Charlie
```

隐式绑定丢失：

如果将对象方法赋值给一个变量，隐式绑定会丢失，`this` 将会回退到默认绑定（在非严格模式下为全局对象，严格模式下为 `undefined`）。

```
const greet = obj.greet;
greet(); // this 指向全局对象，非严格模式下输出: undefined
```

4. 默认绑定

当没有任何绑定规则适用时，`this` 会采用默认绑定。在非严格模式下，默认绑定会将 `this` 指向全局对象（浏览器中是 `window`，Node.js 环境是 `global`）；而在严格模式下，`this` 会是 `undefined`。

```
function greet() {
    console.log(this.name);
}

const name = 'Global';
greet(); // 非严格模式下，this 指向 window，输出: Global
```

5. 箭头函数绑定（词法作用域绑定）

箭头函数不会创建自己的 `this`，它会捕获定义时外层作用域中的 `this`，并且无法通过 `new`、`call`、`apply` 或 `bind` 改变其 `this` 指向。因此，箭头函数的 `this` 绑定具有非常特殊的行为，优先级最低，但也最稳定，因为它的 `this` 是在定义时确定的，之后不再改变。

```
const obj = {
    name: 'David',
    greet: () => {
        console.log(this.name);
    }
};

obj.greet(); // this 指向定义箭头函数时的外层作用域（全局），输出: undefined
```

6. 其他特殊情况

我们讲到的规则已经足以应付平时的开发，但是总有一些语法，超出了我们的规则之外。（神话故事和动漫中总是有类似这样的人物）

```
// 1. 如果在显示绑定中，我们传入一个null或者undefined，那么这个显示绑定会被忽略，使用默认规则：
function foo() {
    console.log(this)
}

var obj = {
    name: 'why'
}

foo.call(obj) // why
```

```
foo.call(null) // window
foo.call(undefined) // window

var bar = foo.bind(null)
bar() // window


// 2. 创建一个函数的 间接引用，这种情况使用默认绑定规则
//      - 赋值(obj2.foo = obj1.foo)的结果是foo函数；
//      - foo函数被直接调用，那么是默认绑定；
function foo() {
    console.log(this)
}

var obj1 = {
    name: 'obj1',
    foo: foo
}

var obj2 = {
    name: 'obj2'
}

obj1.foo() // obj1对象
(obj2.foo = obj1.foo)() // window
```

综合优先级排序（从高到低）：

1. `new` **绑定**：`new` 操作符优先创建新对象并绑定 `this`。
2. **显式绑定（`call`/`apply`/`bind`）**：通过 `call`、`apply` 和 `bind` 显式指定 `this` 的指向。
3. **隐式绑定**：通过对象方法调用时，`this` 会绑定到调用该方法的对象上。
4. **默认绑定**：在没有其他绑定规则时，`this` 指向全局对象（非严格模式）或 `undefined`（严格模式）。
5. **箭头函数**：箭头函数的 `this` 是在定义时绑定的，无法通过任何方式改变。

这个优先级规则帮助我们理解在不同场景下 `this` 的指向如何确定。例如，使用 `new` 关键字时，尽管有显式绑定，`this` 仍然会优先指向新创建的对象。

三、浏览器内核

> 浏览器的**内核**（Rendering Engine 或称为浏览器引擎）是负责处理和渲染网页内容的核心部分。它是浏览器中的重要组件，负责解析 HTML、CSS、JavaScript 以及其他资源（如图像和视频），最终将其显示为用户可见的页面。浏览器的内核一般由两个主要部分组成：**渲染引擎** 和 **JavaScript 引擎**。

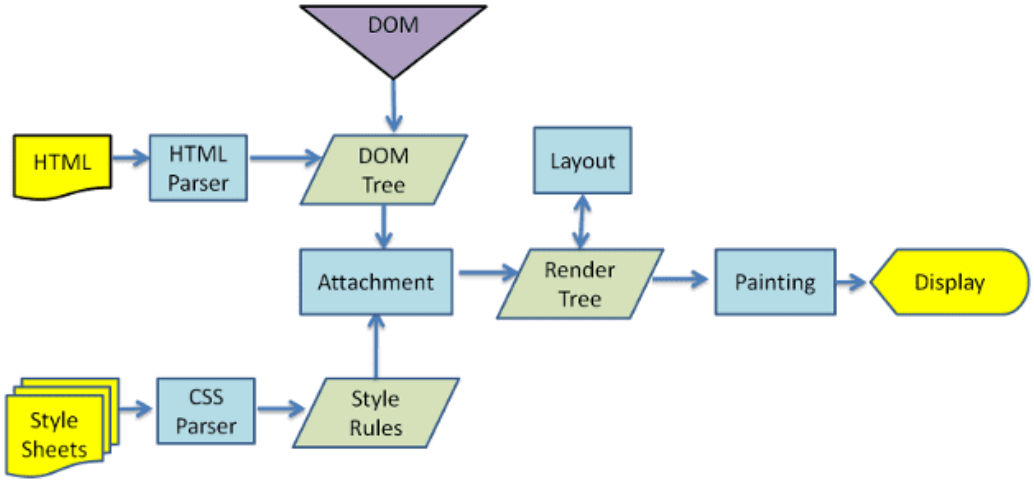
1. 渲染引擎

> 渲染引擎的主要职责是将 HTML、CSS 以及其他内容解析为网页，并呈现给用户。它负责构建 DOM 树、CSSOM 树，生成渲染树，进行布局和绘制。不同浏览器使用不同的渲染引擎，这导致了不同浏览器之间可能出现的页面显示差异。

1.1常见的渲染引擎有：

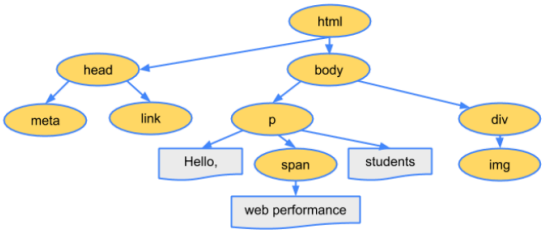
- **WebKit**：早期被 Safari 和 Chrome 使用。Chrome 后来从 WebKit 分支出 **Blink**，成为其自家的渲染引擎。
- **Blink**：现在用于 Google Chrome 和基于 Chromium 的浏览器（如 Microsoft Edge）。
- **Gecko**：Mozilla Firefox 使用的渲染引擎。
- **Trident** 和 **EdgeHTML**：分别是 Internet Explorer 和早期版本的 Microsoft Edge 使用的引擎。Edge 后来转向使用 Blink。

1.2渲染引擎渲染页面的详细流程：



1. 解析 HTML 文件，构建 DOM 树
   当浏览器接收到 HTML 文件后，渲染引擎首先开始解析 HTML 标记语言。它会根据 HTML 元素构建 **DOM 树**（Document Object Model），这是一个反映 HTML 结构的树形结构。

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <p>Hello <span>web performance</span> students!</p>
  <div>
    <img src="awesome-photo.jpg">
  </div>
</body>
</html>
```

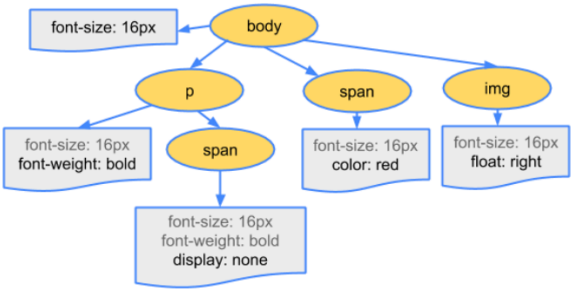


2. 解析 CSS 文件，构建 CSSOM 树

   在解析HTML文件的过程中，如果遇到CSS的link元素，那么会由浏览器负责下载对应的CSS文件（下载CSS文件是不会影响DOM的解析的）；浏览器下载完CSS文件后，就会对CSS文件进行解析。

   渲染引擎会加载与 HTML 相关联的所有样式资源（包括外部的 CSS 文件和嵌入在 HTML 中的样式）。CSS 文件会被解析为 **CSSOM 树**（CSS Object Model Tree），其中每个 CSS 规则与相应的 HTML 元素绑定。

```
body { font-size: 16px }
p { font-weight: bold }
span { color: red }
p span { display: none }
img { float: right }
```

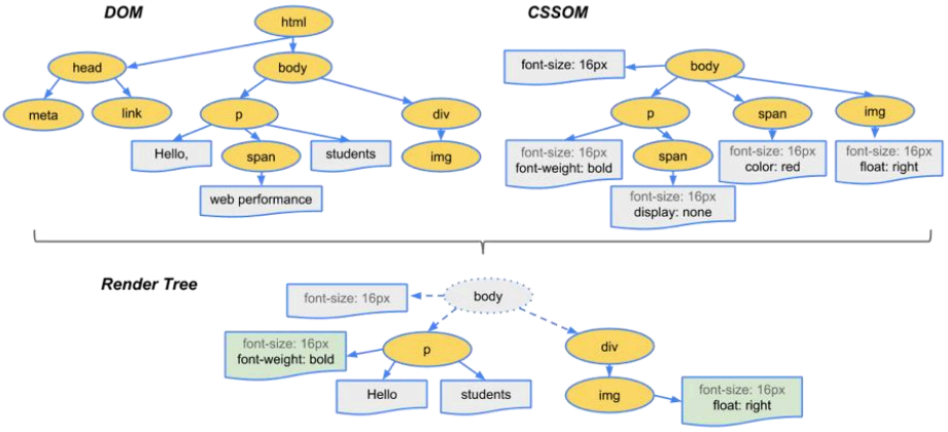


3. 生成渲染树（Render Tree）

   DOM 树和 CSSOM 树构建完毕后，渲染引擎会将这两者结合，生成 **渲染树**。渲染树的每个节点都是可见元素的“盒子”，并且带有样式信息。

   注意一：link元素不会阻塞DOM Tree的构建过程，但是会阻塞Render Tree的构建过程（因为Render Tree在构建时，需要对应的CSSOM Tree；）
   注意二：Render Tree和DOM Tree并不是一一对应的关系（比如对于display为none的元素，压根不会出现在render tree中）

   特点：不包含不可见的元素（如 `display: none` 的元素）；只包含需要呈现在屏幕上的节点。



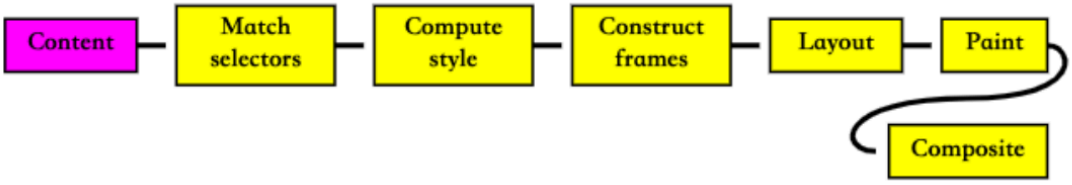
4. 布局（Layout）
   生成渲染树后，布局是在渲染树（Render Tree）上运行布局（Layout）以计算每个节点的几何体。**渲染引擎会开始进行** 布局计算，也称为 **Reflow**或 **重排**。这个阶段的目标是确定每个渲染对象的确切位置和尺寸（高度、宽度）。

   布局过程：

   1、计算每个盒子的位置（坐标）和尺寸（宽高）。

   2、从页面的根节点开始，逐层遍历所有渲染树节点。

   3、布局依赖父元素的尺寸和位置，因此是自上而下的。



5. 绘制（Painting）

   在布局完成后，浏览器会将渲染树中的节点转换为 **像素**，即将每个节点绘制到屏幕上。绘制过程由渲染引擎中的绘制模块负责，将每个节点的视觉属性（如颜色、背景、边框等）转换为图形。

   绘制阶段分为多个步骤：

* 背景颜色和图片的绘制。

* 边框的绘制。

* 文字的绘制。

* 其他装饰效果的绘制，如阴影、渐变等。

6. 合成层（Compositing Layers）

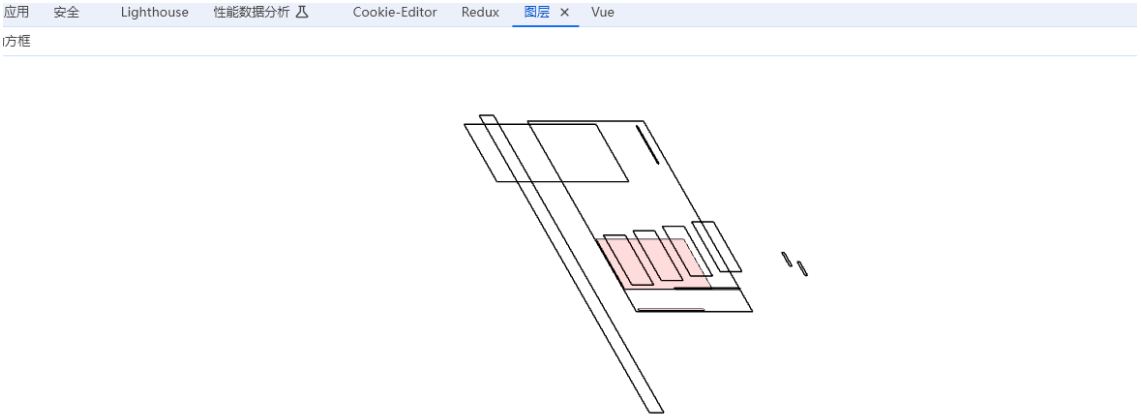
在一些复杂的场景下，渲染引擎会将页面分成多个 **层**，称为 **合成层**。这些层可能是由于 CSS3 的某些属性（如 `transform`、`opacity`、`position: fixed` 等）而被单独处理。每个合成层会被独立绘制，然后进行合成，最终生成完整的页面。这是浏览器的一种优化手段。

- 默认情况下，标准流中的内容都是被绘制在同一个图层（Layer）中的；
- 而一些特殊的属性，会创建一个新的合成层（**CompositingLayer**），并且新的图层可以利用GPU来加速绘制。（因为每个合成层都是单独渲染的）
- 那么哪些属性可以形成新的合成层呢？常见的一些属性
  - 3D transforms
  - video、canvas、iframe
  - opacity 动画转换时
  - position: fixed
  - will-change：一个实验性的属性，提前告诉浏览器元素可能发生哪些变化；
  - animation 或 transition 设置了opacity、transform；
- 分层确实可以提高性能，但是它以内存管理为代价，因此不应作为 web 性能优化策略的一部分过度使用。

为什么要使用合成层?

- 分离绘制复杂的元素，可以提高性能，避免重绘整个页面。
- 合成层独立于其他层更新，可以加速页面交互，例如滚动或动画效果。

每个合成层的绘制顺序会被确定，浏览器最终将这些层合成为一个完整的图像并呈现出来。



1.3 回流和重绘

回流（Reflow） 和 **重绘（Repaint）** 是浏览器渲染过程中涉及的两个重要概念，它们会影响网页的性能，特别是当网页发生频繁的变化时。了解它们的工作机制可以帮助开发者优化页面的渲染，减少性能瓶颈。

1. 回流（Reflow） 也称为重排（Layout）

回流 是当页面的布局和几何信息（如尺寸、位置）发生变化时，浏览器重新计算元素的布局并调整它们在页面中的位置的过程。回流是一个昂贵的操作，因为它需要浏览器重新计算许多元素的位置和大小。

- **第一次**确定节点的大小和位置，称之为**布局**（layout）。
- 之后对节点的大小、位置修改重新计算称之为**回流**。

触发回流的常见操作：

- **添加或删除 DOM 元素**。
- **修改元素的尺寸**（如 `width`、`height`、`padding`、`border`）。
- **修改元素的位置**（如 `top`、`left`）。
- **修改元素的 `display` 属性**（如从 `none` 变为 `block`）。
- **修改页面的字体大小**。
- **窗口大小变化**（如浏览器窗口的缩放）。
- **读取元素几何信息**（如 `offsetWidth`、`offsetHeight`、`scrollTop` 等），因为浏览器为了提供准确的几何数据，会先触发回流。

回流的性能影响：

- 回流会遍历整个渲染树或渲染树的某个子树，并且可能影响页面的多个部分。
- 页面元素越复杂，回流的性能开销就越大。
- 浏览器通常会通过批量处理回流来优化性能，但频繁触发回流仍会导致性能问题，特别是在动态网页或动画中。

2. 重绘（Repaint）

重绘 是指当元素的外观发生变化，但并不影响布局时，浏览器会重新绘制元素以反映这些变化。与回流不同，重绘不需要重新计算元素的位置和几何信息，因此它比回流的性能开销要小一些。

- **第一次**渲染内容称之为**绘制**（paint）。
- 之后重新渲染称之为**重绘**。

触发重绘的常见操作：

- **修改元素的外观**，如背景颜色、边框颜色、字体颜色等。
- **修改元素的 `visibility` 属性**（从 `hidden` 变为 `visible` 或相反）。

重绘的性能影响：

- 虽然重绘比回流开销小，但如果频繁修改元素的样式（例如通过动画），大量的重绘仍会影响性能，尤其是在低性能设备上。

3. 回流和重绘的区别

- **回流**：涉及布局的重新计算，影响元素的几何属性，如位置和尺寸，代价更高。
- **重绘**：只涉及元素的视觉变化，不影响布局，代价较低。

4. 如何减少回流和重绘

为了提高性能，前端开发中应尽量避免频繁触发回流和重绘。以下是一些优化建议：

1. **批量处理 DOM 操作**
   - 在避免频繁的 DOM 操作时，使用 **文档片段（DocumentFragment）** 和 **离线 DOM** 是常见的优化方法。它们的主要目的是减少对实际页面的反复修改，避免多次触发回流和重绘，从而提升性能。

文档片段（DocumentFragment）

`DocumentFragment` 是一个轻量的、无父级的 DOM 片段。当你在 `DocumentFragment` 中进行 DOM 操作时，它不会触发页面的重排（回流），因为它不是真实 DOM 树的一部分，直到你将它一次性插入到实际的 DOM 中为止。

使用 DocumentFragment 的示例：

假设我们有一个列表，想动态添加 1000 个列表项。如果直接操作 DOM，会触发 1000 次回流。而使用 `DocumentFragment` 可以一次性将 1000 个列表项插入 DOM 中，只触发一次回流。

```
<!DOCTYPE html>
<html>
<head>
    <title>DocumentFragment Example</title>
</head>
<body>
    <ul id="list"></ul>

    <script>
        const list = document.getElementById('list');
        const fragment = document.createDocumentFragment();  // 创建文档片段

        for (let i = 0; i < 1000; i++) {
            const listItem = document.createElement('li');
            listItem.textContent = `Item ${i + 1}`;
            fragment.appendChild(listItem);  // 将列表项添加到文档片段中
        }

        list.appendChild(fragment);  // 一次性将文档片段插入到 DOM 中，只触发一次回流
    </script>
</body>
</html>
```

执行步骤：

1. 创建一个 `DocumentFragment` 对象。
2. 将新的 `li` 元素添加到 `DocumentFragment`。
3. 最后一次性将 `DocumentFragment` 插入到实际的 DOM 中。

这减少了对实际 DOM 的操作，只触发了一次回流，而不是 1000 次。

离线 DOM

离线 DOM 是指创建一个不在当前文档中的 DOM 元素（例如一个 `div`），你可以在其上进行大量的 DOM 操作，然后一次性将这个元素或其内容插入到页面中。这和 `DocumentFragment` 的思想类似，但区别在于你操作的是真实 DOM 元素。

使用离线 DOM 的示例：

和上面例子类似，我们也可以通过创建一个新的 `div` 容器，先将新的元素插入到这个离线的 `div` 中，最后再一次性将 `div` 插入页面。

```
<!DOCTYPE html>
<html>
<head>
    <title>Offline DOM Example</title>
</head>
<body>
    <ul id="list"></ul>

    <script>
        const list = document.getElementById('list');
        const tempDiv = document.createElement('div');  // 创建一个离线的 div

        for (let i = 0; i < 1000; i++) {
            const listItem = document.createElement('li');
            listItem.textContent = `Item ${i + 1}`;
            tempDiv.appendChild(listItem);  // 将列表项添加到离线的 div 中
        }

        // 将离线 div 的内容一次性插入到实际 DOM 中
        list.innerHTML = tempDiv.innerHTML;
    </script>
</body>
</html>
```

执行步骤：

1. 创建一个离线的 `div` 元素。
2. 将新的 `li` 元素添加到这个 `div` 中。
3. 最后一次性将 `div` 的内容插入到 `ul` 中。

这减少了对实际 DOM 的频繁操作，降低了回流次数。

总结：

- **DocumentFragment** 是一个轻量的虚拟 DOM 片段，它不会被插入到页面中，因此不会触发回流。通过批量操作 `DocumentFragment`，然后一次性插入页面，可以减少多次回流。
- **离线 DOM** 通过在一个不在文档中的 DOM 元素上进行操作（如创建离线的 `div`），然后再一次性插入实际 DOM 中，也能有效避免频繁的回流操作。

这两种方法都可以大幅度提升动态页面的性能，尤其是在处理大量 DOM 操作时。

2. **避免逐项修改样式**
    - 当需要修改元素的多个样式时，避免逐项修改。例如：

```
// 避免逐项修改
element.style.width = "100px";
element.style.height = "200px";
element.style.backgroundColor = "red";
```

应使用 **class** 或修改 `style.cssText` 一次性修改多个样式：

```
element.className = "new-style"; // 或者使用 class
// 或者
element.style.cssText = "width: 100px; height: 200px; background-color: red;";
```

3. **避免频繁读取导致回流的属性**
   - 读取某些属性（如 `offsetHeight`、`offsetWidth`）会强制浏览器进行回流，以确保获取到的几何信息是准确的。应避免在频繁更新页面时立即读取这些属性，尤其是在循环中。
   - 尽量避免通过getComputedStyle获取尺寸、位置等信息；
4. **使用 CSS3 动画和 GPU 加速**
   - 某些 CSS 属性，如 `transform` 和 `opacity`，可以通过 GPU 加速来进行动画，而不会触发回流和重绘。因此，优先使用这些属性进行动画操作，而不是直接修改几何属性或其他会影响布局的属性。
5. **使用 `will-change` 提示优化**
   - 可以使用 `will-change` CSS 属性来提示浏览器即将发生的变化，这样浏览器可以提前优化处理这些元素。例如：

```
.box {
  will-change: transform;
}
```

6. **对某些元素使用position的absolute或者fixed**
   - 使用这两个定位方式，可以将这些元素从文档的标准流（normal flow）中移除，减少它们对其他元素布局的影响，从而降低回流和重绘的开销。

总结

- **回流**：当页面的布局或几何信息发生变化时触发，性能开销较大，需避免频繁触发。
- **重绘**：当页面元素的外观发生变化但不涉及布局时触发，性能开销较小，但仍需注意避免过度重绘。
- **回流**一定会**引起重绘**，所以回流是一件很消耗性能的事情。
- **优化方法**：批量处理 DOM 操作、减少几何属性的读取、使用 CSS 动画、避免逐项修改样式等，都可以减少回流和重绘，提升网页的性能。

1.4 script元素和页面解析的关系

`<script>` 元素和页面解析的关系直接影响网页的加载顺序、性能和用户体验。JavaScript 的加载、解析和执行与 HTML 文档的解析是紧密相关的，不同的 `script` 加载方式会影响浏览器的解析和渲染流程。以下是 `script` 元素和页面解析之间的关键关系：

1. 默认情况下，JavaScript 阻塞 HTML 解析

当浏览器遇到一个普通的 `<script>` 标签时，HTML 解析会暂停，直到 JavaScript 文件下载完成并执行后才会继续。这是因为 JavaScript 可能会动态修改 DOM 结构（如通过 `document.write`），因此浏览器必须等待脚本执行完毕，以避免修改了未完全解析的 HTML 内容。

流程：

1. 浏览器开始解析 HTML 文档。
2. 遇到 `<script>` 标签，暂停 HTML 解析。
3. 下载并执行 JavaScript。
4. 执行完成后继续解析 HTML。

```
<!DOCTYPE html>
<html>
<head>
    <script src="script.js"></script>
</head>
<body>
    <h1>Hello, World!</h1>
</body>
</html>
```

在上述例子中，页面的解析会在遇到 `<script>` 标签时暂停，等到 `script.js` 下载并执行完后，才会继续解析剩下的 HTML（如 `<h1>` 元素）。

2. `defer` 属性：延迟执行且不阻塞解析

`defer` 属性告诉浏览器在解析完整个 HTML 文档后再执行 JavaScript 文件。这种方式不会阻塞 HTML 的解析，但脚本的执行顺序仍然按照它们在文档中出现的顺序。

- **特点**：
    - 不阻塞 HTML 解析。
    - 在 DOM 树完全构建后执行，优先级低于文档解析。
    - 如果脚本提前下载好了，它会等待DOM Tree构建完成，在**DOMContentLoaded**事件之前先执行defer中的代码；
    - 通常用于需要在文档解析后操作DOM的JavaScript代码，并且对多个script文件有顺序要求的，适合用于页面加载完成后才执行的脚本。

```
<!DOCTYPE html>
<html>
<head>
    <script src="script.js" defer></script>
</head>
<body>
    <h1>Hello, World!</h1>
</body>
</html>
```

在这个例子中，浏览器会继续解析整个页面，直到 HTML 文档全部解析完毕后，再执行 `script.js`，从而加快页面的初始加载速度。

3. `async` 属性：异步加载且不阻塞解析

`async` 属性会告诉浏览器异步下载 JavaScript 文件，不会阻塞 HTML 的解析。一旦脚本下载完毕，它会立即执行，而不等待 HTML 完全解析。这意味着脚本的执行顺序不一定与文档中的顺序一致。

- **特点**：
  - 不阻塞 HTML 解析。
  - 一旦脚本下载完成，立即执行，可能在 HTML 解析完成之前。
  - async不能保证在**DOMContentLoaded**之前或者之后执行；
  - 适合用于与页面无强关联、独立执行的脚本，如广告、分析工具等。

```
<!DOCTYPE html>
<html>
<head>
    <script src="script.js" async></script>
</head>
<body>
    <h1>Hello, World!</h1>
</body>
</html>
```

在这个例子中，`script.js` 会异步加载，不阻塞 HTML 的解析。然而，脚本的执行可能在页面完全解析之前发生。

4. 行内脚本（Inline Script）

当 JavaScript 直接嵌入到 HTML 中时，解析器会在遇到 `<script>` 标签时立即执行该代码，仍然会阻塞 HTML 的解析。

```
<!DOCTYPE html>
<html>
<head>
    <script>
        console.log('This will execute immediately.');
    </script>
</head>
<body>
    <h1>Hello, World!</h1>
</body>
</html>
```

在此示例中，`console.log` 会在浏览器继续解析 `<h1>` 元素之前执行。

5. 放置 `<script>` 标签的位置

将 `<script>` 标签放在 `<body>` 标签的底部（如 `<body>` 结束标签之前），可以减少页面加载初期的阻塞效果。这是因为 HTML 解析和渲染可以先完成，最后才下载并执行 JavaScript 文件。这种方式和 `defer` 属性的效果类似。

```html
<!DOCTYPE html>
<html>
<head>
    <title>Example</title>
</head>
<body>
    <h1>Hello, World!</h1>
    <script src="script.js"></script>
</body>
</html>
```

此时，JavaScript 只有在页面结构已经加载完成后才会被加载和执行。

总结

- **默认** `<script>`：阻塞 HTML 解析，等待脚本加载和执行。
- `defer`：不会阻塞解析，页面解析完成后按顺序执行脚本。
- `async`：不会阻塞解析，脚本加载完成后立即执行，可能在解析结束之前。
- **放置位置**：将 `<script>` 放在 `<body>` 底部可以减少阻塞，提高加载性能。

选择适当的方式加载 JavaScript，可以显著改善页面的加载性能和用户体验。

2. JavaScript 引擎

> JavaScript 引擎专门负责执行网页中的 JavaScript 代码。在现代网页中，JavaScript 用于控制动态行为，如用户交互、动画、数据请求等。JavaScript 引擎是浏览器内核的一个关键部分，负责解释和执行 JavaScript 脚本。

常见的 JavaScript 引擎有：

- **V8**：由 Google 开发，应用在 Chrome 和基于 Chromium 的浏览器中。它被广泛应用于各种环境中，包括 Node.js。
- **SpiderMonkey**：Mozilla 开发的 JavaScript 引擎，用于 Firefox 浏览器。
- **JavaScriptCore（Nitro）**：Apple 的 WebKit 中使用的 JavaScript 引擎，主要用于 Safari。
- **Chakra**：Microsoft 开发，用于 Internet Explorer 和旧版本的 Edge 浏览器。

内核的作用和工作流程

当浏览器加载网页时，内核会执行以下步骤：

1. **解析**：内核从服务器获取 HTML 文件，解析它并生成 DOM 树。
2. **样式计算**：解析 CSS 文件，构建 CSSOM 树。
3. **布局和渲染**：将 DOM 树和 CSSOM 树结合起来生成渲染树，计算页面中各个元素的布局，并绘制在屏幕上。
4. **执行 JavaScript**：JavaScript 引擎解析并执行页面中的 JavaScript 代码，可能会修改 DOM 树或 CSSOM 树，从而影响页面的渲染。

内核和浏览器的区别

浏览器是整个应用程序，而内核只是其中负责渲染网页内容的部分。浏览器还包括其他组件，例如：

- **用户界面**：如地址栏、前进和后退按钮等。
- **网络层**：负责处理 HTTP 请求和响应。
- **数据存储**：管理 cookies、localStorage 等数据存储机制。

浏览器内核的工作是幕后进行的，它处理了大量技术细节，确保用户在访问网页时得到无缝的体验。