



CSS魔术师Houdini

yuanzhijia@yidengxuetang.com



目 录

CONTENTS

- ① Parser、Paint、Layout API
- ② Worklets实战
- ③ Properties/Values
- ④ Typed OM Object

Houdini API介绍

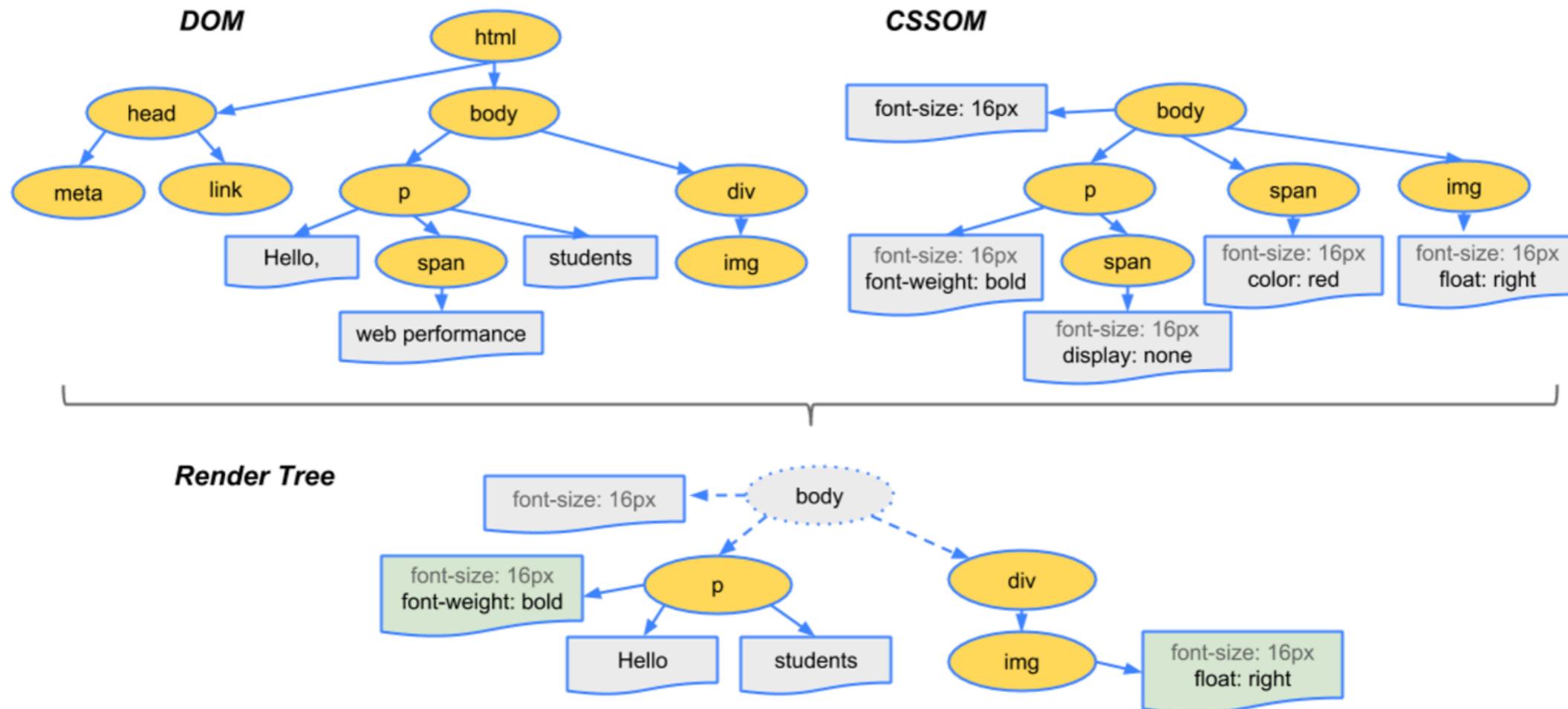
JavaScript

Style

Layout

Paint

Composite



Houdini API介绍

在现今的 Web 开发中，JavaScript 几乎占据所有版面，除了控制页面逻辑与操作 DOM 对象以外，连 CSS 都直接写在 JavaScript 里面了，就算浏览器都还没有实现的特性，总会有人做出对应的 Polyfills，让你快速的将新 Feature 应用到 Production 环境中，更别提我们还有 Babel 等工具帮忙转译。

而 CSS 就不同了，除了制定 CSS 标准规范所需的时间外，各家浏览器的版本、实战进度差异更是旷日持久，顶多利用 PostCSS、Sass 等工具来帮我们转译出浏览器能接受的 CSS。开发者们能操作的就是通过 JS 去控制 DOM 与 CSSOM 来影响页面的变化，但是对于接下来的 Layout、Paint 与 Composite 就几乎没有控制权了。

为了解决上述问题，为了让 CSS 的魔力不再浏览器把持，Houdini 就诞生了！（Houdini 是美国的伟大魔术师，擅长逃脱术，很适合想将 CSS 从浏览器中解放的概念）

CSS Houdini 是由一群来自 Mozilla, Apple, Opera, Microsoft, HP, Intel, IBM, Adobe 与 Google 的工程师所组成的工作小组，志在建立一系列的 API，让开发者能够介入浏览器的 CSS engine

章节
Part

01

Parser、Paint、Layout API
扩展CSS此法分析器

几种新增API的详细解释

Parser、Paint、Layout API



CSS Parser API 还没有被写入规范，所以下面我要说的内容随时都会有变化，但是它的基本思想不会变：允许开发者自由扩展 CSS 词法分析器，引入新的结构（constructs），比如新的媒体规则、新的伪类、嵌套、`@extends`、`@apply` 等等。

只要新的词法分析器知道如何解析这些新结构，CSSOM 就不会直接忽略它们，而是把这些结构放到正确的地方。



CSS Layout API 允许开发者可以通过 CSS Layout API 实现自己的布局模块（layout module），这里的“布局模块”指的是 `display` 的属性值。也就是说，这个 API 实现以后，开发者首次拥有了像 CSS 原生代码（比如 `display:flex`、`display:table`）那样的布局能力。



CSS Paint API 和 Layout API 非常相似。它提供了一个 `registerPaint` 方法，操作方式和 `registerLayout` 方法也很相似。当想要构建一个 CSS 图像的时候，开发者随时可以调用 `paint()` 函数，也可以使用刚刚注册好的名字。



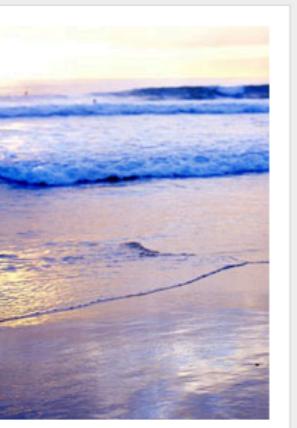
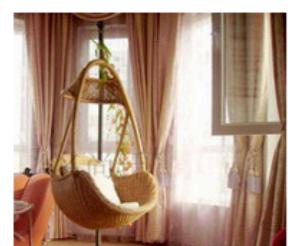
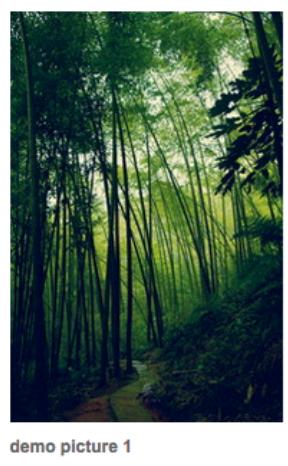
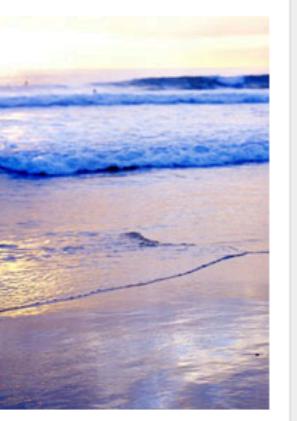
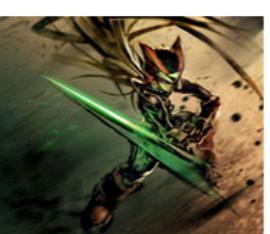
具体的 Coding 过程要借助 Worklet

布局模块

CSS Layout API

.item-list { display: layout(waterfall); }

<https://www.jasondavies.com/wordcloud/>



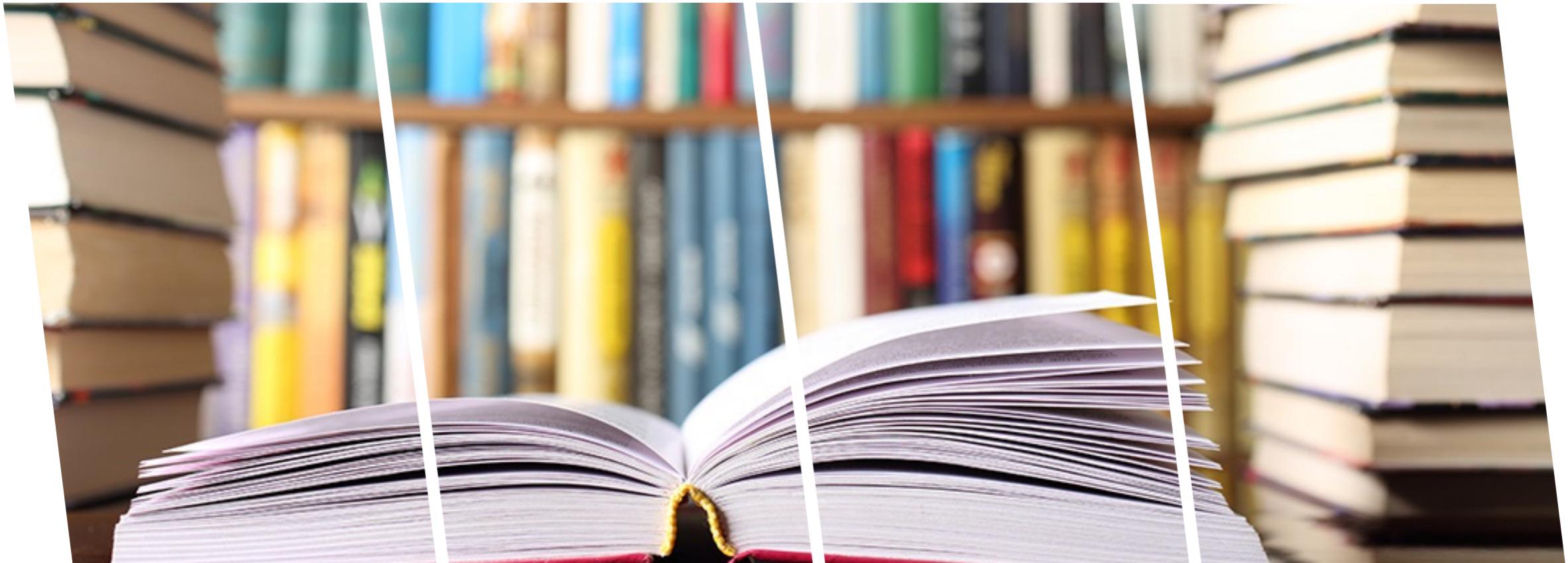
.item-list { display: layout(waterfall); }

```
registerLayout('waterfall', class {
  static get inputProperties () ...
  static get childrenInputProperties () ...
  static get childDisplay () ...
  *intrinsicSizes(children, styleMap)
  *layout(constraints, children,
    styleMap, edges, breakToken)
})
```

```
registerLayout(waterfall, class {
  static get inputProperties() {
    return ['width', 'height']
  }
  static get childrenInputProperties() {
    return ['x', 'y', 'position']
  }
  layout(children, constraintSpace, styleMap, breakToken) {
    // Layout logic goes here.
  }
})
```

布局模块

CSS Layout API



Box tree API 目的就是希望让开发者能够取得这些 fragments 的信息，至于取得后要如何使用，基本上应该会跟后面介绍的 Parser API、Layout API 与 Paint API 有关联，当我们能取得详细的 Box Modal 信息时，要客制化 Layout Module 才更为方便。

布局模块

CSS Layout API

```
<div class="wrapper">  
<p>foo <i>bar baz</i></p>  
</div>
```

```
p::first-line { color: green; }  
  
p::first-letter { color: red; }  
  
.wrapper {  
    width: 60px;  
}
```

foo bar
baz

上面的 HTML 总共就会拆出七个 fragments :

- 最外层的 div
- 第一行的 box (包含 foo bar)
- 第二行的 box (包含 baz)
- 吃到 ::first-line 与 ::first-letter 的 f 也会被拆出来成独立的 fragments
- 只吃到 ::first-line 的 oo 只好也独立出来
- 吃到 ::first-line 与 包在 <i> 内的 bar 当然也是
- 在第二行底下且为 italic 的 baz

.test { background-image: paint(circle); }

```
registerPaint('circle', class {
  static get inputProperties() { return ['--circle-color']; }
  paint(ctx, geom, properties) {
    // 改变填充颜色
    const color = properties.get('--circle-color');
    ctx.fillStyle = color;
    // 确定圆心和半径
    const x = geom.width / 2;
    const y = geom.height / 2;
    const radius = Math.min(x, y);
    // 画圆
    ctx.beginPath();
    ctx.arc(x, y, radius, 0, 2 * Math.PI, false);
    ctx.fill();
  }
});
```

章节
Part

02 来一些Worklets的实战
Worklets 的概念和 web worker 类似

xxxWorklet.addModule('xxx.js').then(...)



Worklets

(registerLayout 和 registerPaint) 已经了解过了，估计现在你想知道的是，这些代码得放在哪里呢？答案就是 worklet 脚本（工作流脚本文件）。

Worklets 的概念和 web worker 类似，它们允许你引入脚本文件并执行特定的 JS 代码，这样的 JS 代码要满足两个条件：第一，可以在渲染流程中调用；第二，和主线程独立。

Worklet 脚本严格控制了开发者所能执行的操作类型，这就保证了性能。Worklets 的特点就是轻量以及生命周期较短。

Worklets实战

Worklets Demo

```
CSS.paintWorklet.addModule('xxx.js')
CSS.layoutWorklet.addModule( 'xxx.js')
```

```
// xxx.js
registerPaint('xxx', class {
  static get inputProperties () { ... }
  static get inputArguments () { ... }
  paint (ctx, geom, props) { ... }
})
```



章节
Part

03 Properties/Values
首先至少你要掌握CSS WorkFlow

Properties/Values

var(--stop-color)

```
CSS.registerProperty({  
  name: '--stop-color',  
  syntax: '<color>',  
  inherits: false,  
  initialValue: 'rgba(0,0,0,0)'  
})
```

```
CSS.unregisterProperty(  
  '--stop-color'  
)
```

Properties/Values

var(--stop-color)

```
.button {  
  --stop-color: red; /* red */  
  background: linear-gradient(  
    var(--stop-color), black);  
  transition: --stop-color 1s;  
}  
  
.button:hover {  
  --stop-color: green; /* green */  
}
```



```
.button { transition: --stop-color 1s; }
```

章节
Part

04 Typed OM Object

解决目前模型的一些问题，并实现 CSS Parsing API 和 CSS 属性与值 API 相关的特性。

Typed OM Object

Typed OM Demo

```
// <position> 5px 10px
let pos = new CSSPositionValue(
  new CSSUnitValue(5, "px"),
  new CSSUnitValue(10, "px")
);
```

```
el.style.width
el.attributeStyleMap.get('width')

getComputedStyle(el).width
el.computedStyleMap().get('width')
```

code1

```
el.attributeStyleMap.set('padding', CSS.px(42))  
const padding = el.attributeStyleMap.get('padding')  
console.log(padding.value, padding.unit) // 42, 'px'
```

- 更少的bug。例如数字值总是以数字形式返回，而不是字符串。
- `el.style.opacity += 0.1; el.style.opacity === '0.30.1' // dragons!`
- 算术运算和单位转换。在绝对长度单位（例如 px -> cm）之间进行转换并进行基本的数学运算。
- 数值范围限制和舍入。Typed OM 通过对值进行范围限制和舍入，以使其在属性的可接受范围内。
- 更好的性能。浏览器必须做更少的工作序列化和反序列化字符串值。现在，对于 CSS 值，引擎可以对 JS 和 C++ 使用相似的理解。Tab Atkins 已经展示了一些早期的性能基准测试，与使用旧的 CSSOM 和字符串相比，Typed OM 的运行速度快了 ~30%。这对使用 `requestAnimationFrame()` 处理快速 CSS 动画可能很重要。
crbug.com/808933 可以追踪 Blink 的更多性能演示。
- 错误处理。新的解析方法带来了 CSS 世界中的错误处理。
- “我应该使用骆驼式的 CSS 名称还是字符串呢？”你不再需要猜测名字是骆驼还或字符串（例如 `el.style.backgroundColor` vs `el.style['background-color']`）。Typed OM 中的 CSS 属性名称始终是字符串，与您实际在 CSS 中编写的内容一致：）



Q&A