性能优化2-- 代码执行

知识点

雅虎军规

```
尽量减少 HTTP 请求个数——须权衡
使用 CDN（内容分发网络）
为文件头指定 Expires 或 Cache-Control ，使内容具有缓存性。
避免空的 src 和 href
使用 gzip 压缩内容
```

```
把 CSS 放到顶部
把 JS 放到底部
避免使用 CSS 表达式
将 CSS 和 JS 放到外部文件中
减少 DNS 查找次数
精简 CSS 和 JS
避免跳转
剔除重复的 JS 和 CSS
配置 ETags
使 AJAX 可缓存
尽早刷新输出缓冲
使用 GET 来完成 AJAX 请求
延迟加载
预加载
减少 DOM 元素个数
根据域名划分页面内容
尽量减少 iframe 的个数
避免 404
减少 Cookie 的大小
使用无 cookie 的域
减少 DOM 访问
开发智能事件处理程序
用  代替 @import
避免使用滤镜
优化图像
优化 CSS Spirite
不要在 HTML 中缩放图像——须权衡
favicon.ico要小而且可缓存
保持单个内容小于25K
打包组件成复合文本
```

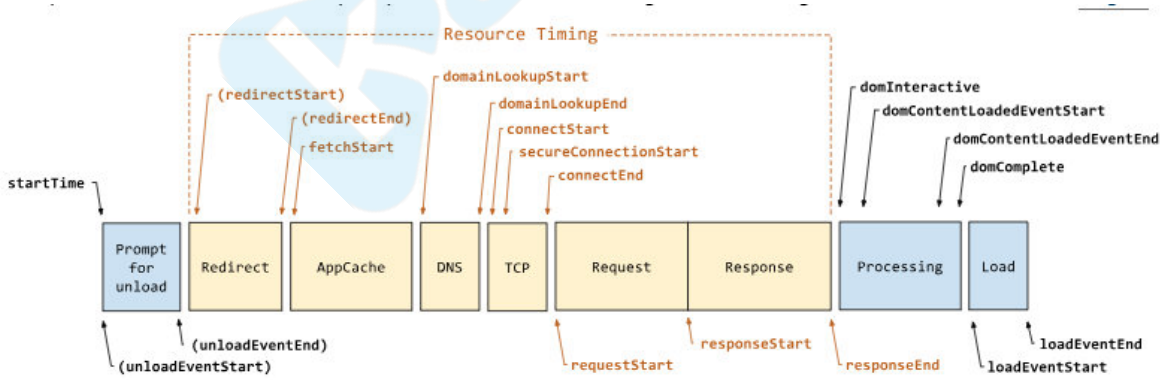
性能监控 Performance

https://developer.mozilla.org/zh-CN/docs/Web/API/Performance

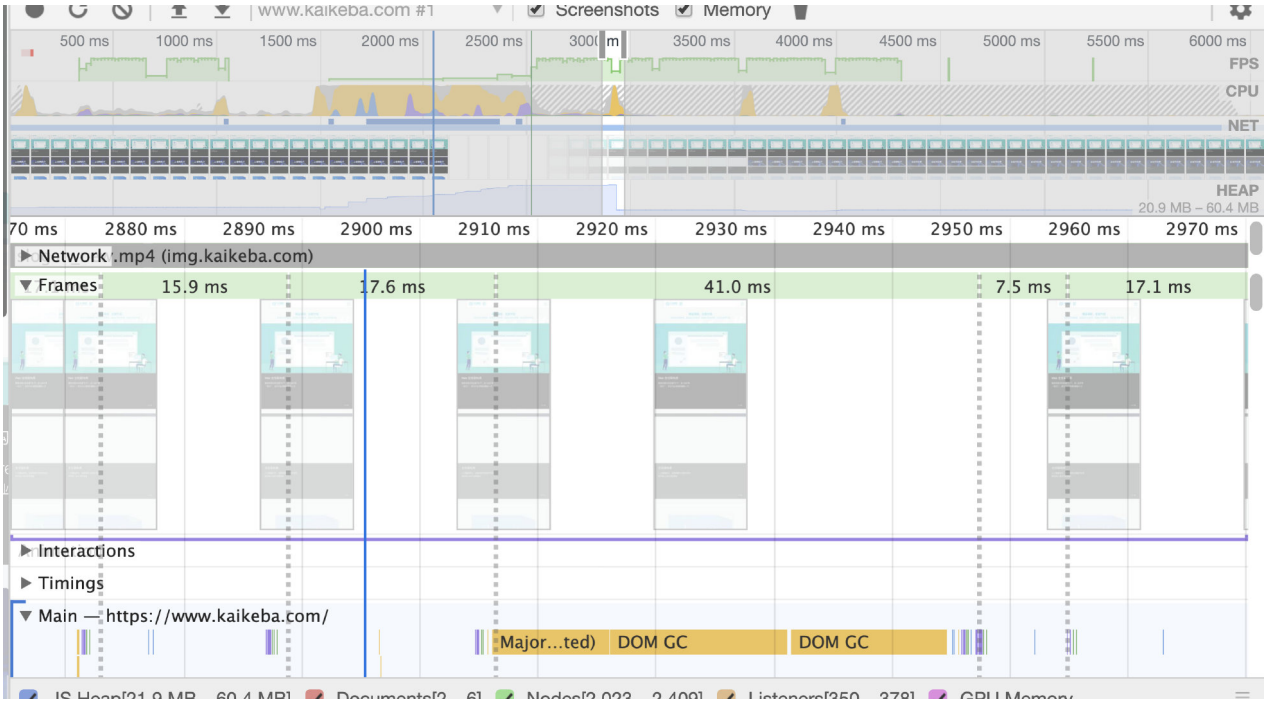
performance.getEntriesByType('navigation')

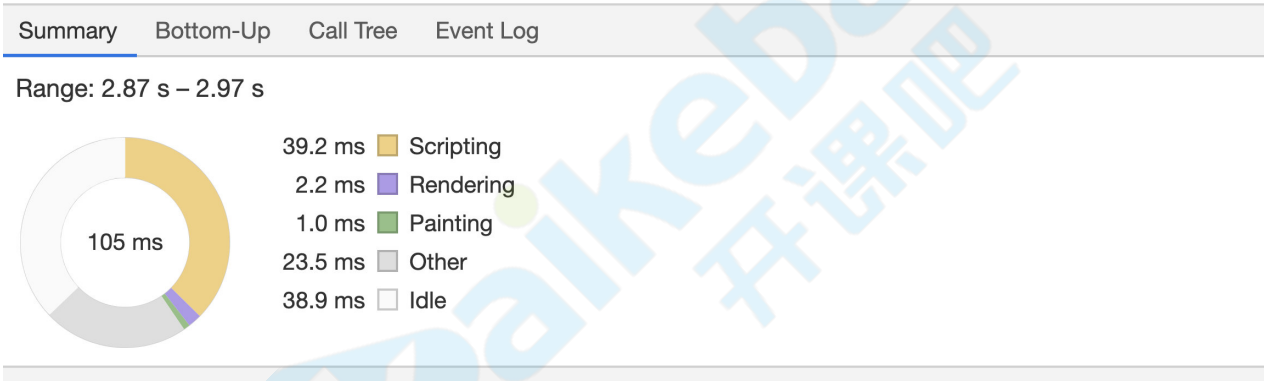
```
> performance.getEntriesByType('navigation')
▼ [PerformanceNavigationTiming]
  ▼ 0: PerformanceNavigationTiming
      connectEnd: 28.355000002193265
      connectStart: 8.065000001806766
      decodedBodySize: 17747
      domComplete: 756.359999999404
      domContentLoadedEventEnd: 505.1999999996042
      domContentLoadedEventStart: 504.99000000127126
      domInteractive: 504.95000000228174
      domainLookupEnd: 8.065000001806766
      domainLookupStart: 2.3800000017217826
      duration: 764.1700000021956
      encodedBodySize: 11683
      entryType: "navigation"
      fetchStart: 0.9400000017194543
      initiatorType: "navigation"
      loadEventEnd: 764.1700000021956
      loadEventStart: 756.375000000844
      name: "https://juejin.im/book/5bdc715fe51d454e755f75ef/section/5bdc73396fb9a04a0c2ddfe2"
      nextHopProtocol: "http/1.1"
      redirectCount: 0
      redirectEnd: 0
      redirectStart: 0
      requestStart: 28.39999999923748
      responseEnd: 287.61999999915133
      responseStart: 281.1900000015157
      secureConnectionStart: 0
    ▶ serverTiming: []
      startTime: 0
      transferSize: 13054
      type: "reload"
      unloadEventEnd: 287.95499999978347
      unloadEventStart: 286.06000000218046
      workerStart: 0
    ▶ __proto__: PerformanceNavigationTiming
    length: 1
  ▶ __proto__: Array(0)

> performance
▼ Performance {timeOrigin: 1554719812775.267, onresourcetimingbufferfull: null, memory: MemoryInfo, navigation:
  PerformanceNavigation, timing: PerformanceTiming}
```



挖掘瓶颈—火焰图





重定向耗时：redirectEnd - redirectStart

DNS查询耗时： domainLookupEnd - domainLookupStart

TCP链接耗时： connectEnd - connectStart

HTTP请求耗时： responseEnd - responseStart

解析dom树耗时： domComplete - domInteractive

白屏时间： responseStart - navigationStart

DOMready时间： domContentLoadedEventEnd - navigationStart

onload时间：loadEventEnd - navigationStart，也即是onload回调函数执行的时间。

LightHouse

chrome插件

命令行

```
npm install -g lighthouse
lighthouse https://www.kaikeba.com/ --view
```

节流

滚动，隔一段时间只触发一次 第一个人说了算 在时间结束出发

```
// func是用户传入需要防抖的函数
// wait是等待时间
const throttle = (func, wait = 50) => {
  // 上一次执行该函数的时间
  let lastTime = 0
  return (...args)=>{
    // 当前时间
    let now = +new Date()
    // 将当前时间和上一次执行函数时间对比
    // 如果差值大于设置的等待时间就执行函数
    if (now - lastTime > wait) {
      lastTime = now
      func.apply(this, args)
    }
  }
}

setInterval(
  throttle(() => {
    console.log(1)
  }, 500),
  1
)
```

防抖

输入，完成后再统一发送请求，最后一个人说了算 只认最后一次

```
// func是用户传入需要防抖的函数
// wait是等待时间
const debounce = (func, wait = 50) => {
  // 缓存一个定时器id
  let timer = 0
  // 这里返回的函数是每次用户实际调用的防抖函数
  // 如果已经设定过定时器了就清空上一次的定时器
  // 开始一个新的定时器，延迟执行用户传入的方法
  return function(...args) {
    if (timer) clearTimeout(timer)
    timer = setTimeout(() => {
      func.apply(this, args)
    }, wait)
  }
}
```

dom

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>DOM操作测试</title>
</head>
<body>
  <div id="container"></div>
</body>
</html>
```

```
for(var count=0;count<10000;count++){
  document.getElementById('container').innerHTML+='<span>我是一个小测试</span>'
}
```

重绘 回流

回流：当我们对 DOM 的修改引发了 DOM 几何尺寸的变化（比如修改元素的宽、高或隐藏元素等）时，浏览器需要重新计算元素的几何属性（其他元素的几何属性和位置也会因此受到影响），然后再将计算的结果绘制出来。这个过程就是回流（也叫重排）。

重绘：当我们对 DOM 的修改导致了样式的变化、却并未影响其几何属性（比如修改了颜色或背景色）时，浏览器不需重新计算元素的几何属性、直接为该元素绘制新的样式（跳过了上图所示的回流环节）。这个过程叫做重绘。

由此我们可以看出，**重绘不一定导致回流，回流一定会导致重绘**。

回流是影响最大的

1. 窗体，字体大小
2. 增加样式表
3. 内容变化
4. class属性
5. offfserWidth 和offsetHeight
6. fixed

1. DocumentFragment缓存dom

lazy-load

```
    // 获取所有的图片标签
    const imgs = document.getElementsByTagName('img')
    // 获取可视区域的高度
    const viewHeight = window.innerHeight ||
document.documentElement.clientHeight
    // num用于统计当前显示到了哪一张图片，避免每次都从第一张图片开始检查是否露出
    let num = 0
    function lazyload(){
        for(let i=num; i<imgs.length; i++) {
            // 用可视区域高度减去元素顶部距离可视区域顶部的高度
            let distance = viewHeight - imgs[i].getBoundingClientRect().top
            // 如果可视区域高度大于等于元素顶部距离可视区域顶部的高度，说明元素露出
            if(distance >= 0 ){
                // 给元素写入真实的src，展示图片
                imgs[i].src = imgs[i].getAttribute('data-src')
                // 前i张图片已经加载完毕，下次从第i+1张开始检查是否露出
                num = i + 1
            }
        }
    }
    // 监听Scroll事件
    window.addEventListener('scroll', lazyload, false);
```

vue

其实我们只要做好异步组件，vue本身已经足够快乐，但是还是有一些可以优化的点

v-if vs v-show

初始性能 VS 频繁切换性能

和渲染无关的数据，不要放在data上

data也不要嵌套过多层

nextTick

修改数据的当下，视图不会立刻更新，而是等同一事件循环中的所有数据变化完成之后，再统一进行视图更新

```
        this.$nextTick(function() {
            // DOM 更新了

        })
```

Object.freeze()

冻结数据 取消setters

react

只传递需要的props

不要随便 <Component {...props} />

key

无状态组件

pureComponent shouldComponentUpdate

渲染时机

少在render中绑定事件

redux + reselect

data扁平化

每当store发生改变的时候，connect就会触发重新计算，为了减少重复的不必要计算，减少大型项目的性能开支，需要对selector函数做缓存。推荐使用[reactjs/reselect](), 缓存的部分实现代码如下。

长列表 react-virtualized

只渲染可见的，

```
renderRow(item) {
  return (
    <div key={item.id} className="row">
      <div className="image">
        <img src={item.image} alt="" />
      </div>
      <div className="content">
        <div>{item.name}</div>
        <div>{item.text}</div>
      </div>
    </div>
  );
}
```

如果有1000个， 只渲染20个，鼠标滚动的时候，新节点替换老节点、

```
import { List } from "react-virtualized";


<div className="list">

  {this.list.map(this.renderRow.bind(this))}

</div>



// 改为
const listHeight = 600;

const rowHeight = 50;

const rowWidth = 800;
```

```
//...

<div className="list">

  <List

    width={rowWidth}

    height={listHeight}

    rowHeight={rowHeight}

    rowRenderer={this.renderRow.bind(this)}

    rowCount={this.list.length} />

</div>

renderRow({ index, key, style }) {
  return (
    <div key={key} style={style} className="row">
      <div className="image">
        <img src={this.list[index].image} alt="" />
      </div>
      <div className="content">
        <div>{this.list[index].name}</div>
        <div>{this.list[index].text}</div>
      </div>
    </div>
  );
}
```

只渲染可见的

Web Workers

浏览器渲染

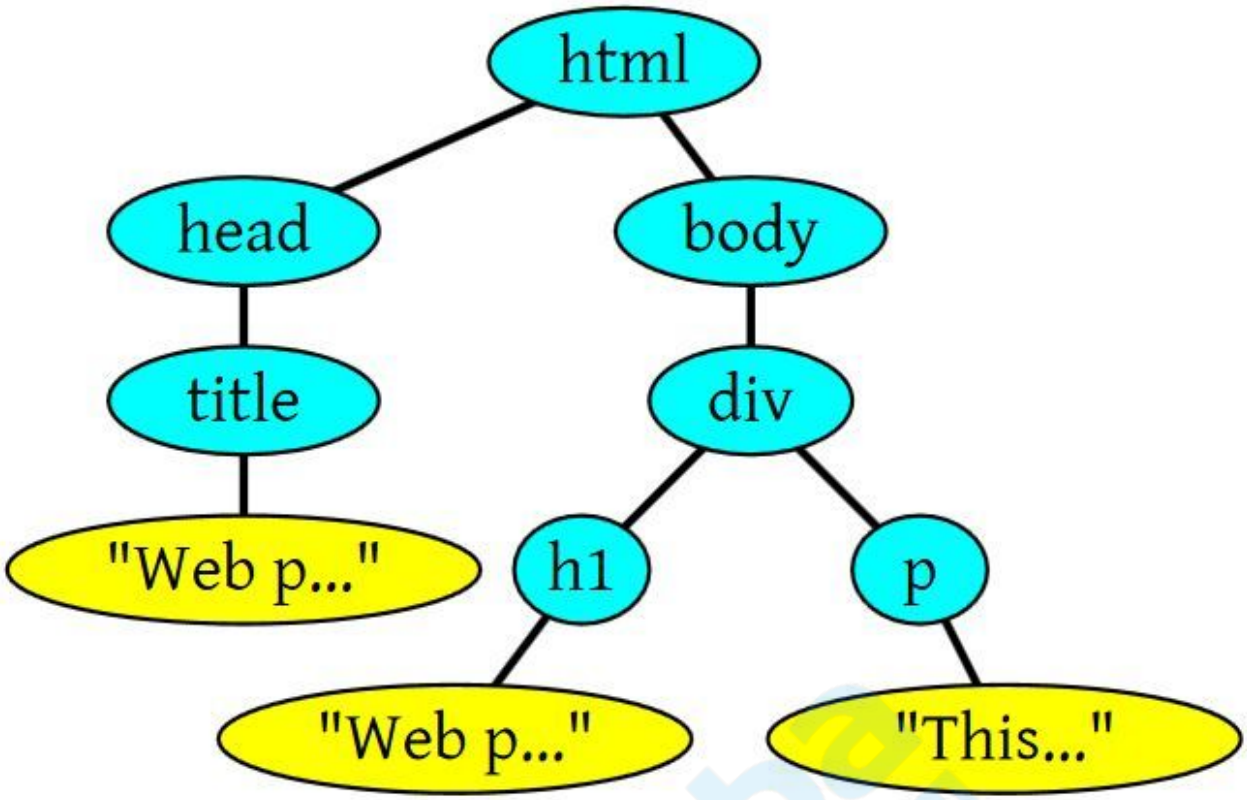
1. 在这一步浏览器执行了所有的加载解析逻辑，在解析 HTML 的过程中发出了页面渲染所需的各种外部资源请求。
2. 浏览器将识别并加载所有的 CSS 样式信息与 DOM 树合并，最终生成页面 render 树（:after :before 这样的伪元素会在这个环节被构建到 DOM 树中）。
3. 页面中所有元素的相对位置信息，大小等信息均在这一步得到计算。
4. 在这一步中浏览器会根据我们的 DOM 代码结果，把每一个页面图层转换为像素，并对所有的媒体文件进行解码。
5. 最后一步浏览器会合并合各个图层，将数据由 CPU 输出给 GPU 最终绘制在屏幕上。（复杂的视图层会给这个阶段的 GPU 计算带来一些压力，在实际应用中为了优化动画性能，我们有时会手动区分不同的图层）

渲染过程说白了，首先是基于 HTML 构建一个 DOM 树，这棵 DOM 树与 CSS 解释器解析出的 CSSOM 相结合，就有了布局渲染树。最后浏览器以布局渲染树为蓝本，去计算布局并绘制图像，我们页面的初次渲染就大功告成了。

之后每当一个新元素加入到这个 DOM 树当中，浏览器便会通过 CSS 引擎查遍 CSS 样式表，找到符合该元素的样式规则应用到这个元素上，然后再重新去绘制它。

dom

```
<html>
<html>
<head>
    <title>Web page parsing</title>
</head>
<body>
    <div>
        <h1>Web page parsing</h1>
        <p>This is an example Web page.</p>
    </div>
</body>
</html>
```



css

从右向左 照着domtree来的，所以css也有很多优化点

算法

排序

动态规划