特性	cookie	localStorage	sessionStorage	indexDB
与服务端通信	每次都会携带在 header 中，对于请求性能影响	不参与	不参与	不参与

> 从上表可以看到， cookie 已经不建议用于存储 。如果没有大量数据存储需求的话， 可以使用 localStorage 和 sessionStorage 。对于不怎么改变的数据尽量使用 localStorage 存储， 否则可以用 sessionStorage 存储

对于 cookie 来说，我们还需要注意安全性。

属性	作用
value	如果用于保存用户登录态，应该将该值加密，不能使用明文的用户标识
http-only	不能通过 JS 访问 Cookie ，减少 XSS 攻击
secure	只能在协议为 HTTPS 的请求中携带
same-site	规定浏览器不能在跨域请求中携带 Cookie ，减少 CSRF 攻击

Service Worker

- Service Worker 是运行在浏览器背后的独立线程，一般可以用来实现缓存功能 。使用 Service Worker 的话，传输协议必须为 HTTPS 。因为 Service Worker 中涉及到请求拦截，所以必须使用 HTTPS 协议来保障安全
- Service Worker 实现缓存功能一般分为三个步骤：首先需要先注册 Service Worker ，然后监听到 install 事件以后就可以缓存需要的文件，那么在下次用户访问的时候就可以通过拦截请求的方式查询是否存在缓存，存在缓存的话就可以直接读取缓存文件， 否则就去请求数据 。以下是这个步骤的实现：

```js
// index.js
if (navigator.serviceWorker) {
  navigator.serviceWorker
    .register( 'sw.js')
    .then(function(registration) {
      console.log( 'service worker 注册成功 ')
    })
    .catch(function(err) {
      console.log( 'servcie worker 注册失败')
    })
```

```
}
// sw.js
// 监听 `install` 事件，回调中缓存所需文件
self.addEventListener( 'install', e => {
  e.waitUntil(
    caches.open( 'my-cache').then(function(cache) {
      return cache.addAll( [ './index.html', './index.js'])
    })
  )
})

// 拦截所有请求事件
// 如果缓存中已经有请求的数据就直接用缓存，否则去请求数据
self.addEventListener( 'fetch', e => {
  e.respondWith(
    caches.match(e.request).then(function(response) {
      if (response) {
        return response
      }
      console.log( 'fetch source')
    })
  )
})
```

> 打开页面，可以在开发者工具中的 Application 看到 Service Worker 已经启动了

> 在 Cache 中也可以发现我们所需的文件已被缓存

> 当我们重新刷新页面可以发现我们缓存的数据是从 Service Worker 中读取的

20 浏览器缓存机制

> 注意：该知识点属于性能优化领域， 并且整一章节都是一个面试题

- 缓存可以说是性能优化中简单高效的一种优化方式了， 它可以显著减少网络传输所带来的损耗。
- 对于一个数据请求来说， 可以分为发起网络请求 、后端处理 、浏览器响应三个步骤 。浏览器缓存可以帮助我们在第一和第三步骤中优化性能 。比如说直接使用缓存而不发起请求，或者发起了请求但后端存储的数据和前端一致，那么就没有必要再将数据回传回来， 这样就减少了响应数据。

> 接下来的内容中我们将通过以下几个部分来探讨浏览器缓存机制：

- 缓存位置
- 缓存策略
- 实际场景应用缓存策略

20.1 缓存位置

> 从缓存位置上来说分为四种， 并且各自有优先级， 当依次查找缓存且都没有命中的时候，才会去请求网络

1. `Service Worker`
2. `Memory Cache`
3. `Disk Cache`
4. `Push Cache`
5. 网络请求

1. Service Worker

- `service Worker` 的缓存与浏览器其他内建的缓存机制不同， 它可以让我们自由控制缓存哪些文件 、如何匹配缓存 、如何读取缓存， 并且缓存是持续性的。
- 当 `Service Worker` 没有命中缓存的时候， 我们需要去调用 `fetch` 函数获取数据 。也就是说， 如果我们没有在 `Service Worker` 命中缓存的话，会根据缓存查找优先级去查找数据 。但是不管我们是从 `Memory Cache` 中还是从网络请求中获取的数据， 浏览器都会显示我们是从 `Service Worker` 中获取的内容。

2. Memory Cache

- Memory Cache 也就是内存中的缓存，读取内存中的数据肯定比磁盘快 。但是内存缓存虽然读取高效， 可是缓存持续性很短，会随着进程的释放而释放 。 一旦我们关闭 Tab 页面， 内存中的缓存也就被释放了。
- 当我们访问过页面以后， 再次刷新页面， 可以发现很多数据都来自于内存缓存

> 那么既然内存缓存这么高效， 我们是不是能让数据都存放在内存中呢?

- 先说结论， 这是不可能的 。首先计算机中的内存一定比硬盘容量小得多， 操作系统需要精打细算内存的使用，所以能让我们使用的内存必然不多 。内存中其实可以存储大部分的文件， 比如说 JS 、 HTML 、 CSS 、图片等等
- 当然， 我通过一些实践和猜测也得出了一些结论:
- 对于大文件来说，大概率是不存储在内存中的， 反之优先当前系统内存使用率高的话，文件优先存储进硬盘

3. Disk Cache

- Disk Cache 也就是存储在硬盘中的缓存，读取速度慢点，但是什么都能存储到磁盘中，比之 Memory Cache 胜在容量和存储时效性上。
- 在所有浏览器缓存中， Disk Cache 覆盖面基本是最大的 。它会根据 · HTTP Herder· 中的字段判断哪些资源需要缓存， 哪些资源可以不请求直接使用， 哪些资源已经过期需要重新请求 。并且即使在跨站点的情况下， 相同地址的资源一旦被硬盘缓存下来，就不会再次去请求数据

4. Push Cache

- Push Cache 是 HTTP/2 中的内容， 当以上三种缓存都没有命中时， 它才会被使用 。并且缓存时间也很短暂， 只在会话 (Session) 中存在，一 旦会话结束就被释放。
- Push Cache 在国内能够查到的资料很少，也是因为 HTTP/2 在国内不够普及，但是 HTTP/2 将会是日后的一个趋势

> 结论

- 所有的资源都能被推送，但是 Edge 和 Safari 浏览器兼容性不怎么好
- 可以推送 no-cache 和 no-store 的资源
- 一旦连接被关闭， Push Cache 就被释放
- 多个页面可以使用相同的 HTTP/2 连接，也就是说能使用同样的缓存
- Push Cache 中的缓存只能被使用一次

- 浏览器可以拒绝接受已经存在的资源推送
- 你可以给其他域名推送资源

5. 网络请求

- 如果所有缓存都没有命中的话，那么只能发起请求来获取资源了。
- 那么为了性能上的考虑，大部分的接口都应该选择好缓存策略，接下来我们就来学习缓存策略这部分的内容

20.2 缓存策略

> 通常浏览器缓存策略分为两种：强缓存和协商缓存， 并且缓存策略都是通过设置 `HTTP Header` 来实现的

20.2.1 强缓存

> 强缓存可以通过设置两种 `HTTP Header` 实现： `Expires` 和 `Cache-Control` 。强缓存表示在缓存期间不需要请求， `state code` 为 `200`

Expires

```
Expires: Wed, 22 Oct 2018 08:41:00 GMT
```

> `Expires` 是 `HTTP/1` 的产物，表示资源会在 `Wed, 22 Oct 2018 08:41:00 GMT` 后过期， 需要再次请求 。并且 `Expires` 受限于本地时间， 如果修改了本地时间， 可能会造成缓存失效。

Cache-control

```
Cache-control: max-age=30
```

- `Cache-Control` 出现于 `HTTP/1.1` ，优先级高于 `Expires` 。该属性值表示资源会在 `30` 秒后过期， 需要再次请求。
- `Cache-Control` 可以在请求头或者响应头中设置， 并且可以组合使用多种指令

> 从图中我们可以看到， 我们可以将多个指令配合起来一起使用， 达到多个目的 。比如说我们希望资源能被缓存下来， 并且是客户端和代理服务器都能缓存， 还能设置缓存失效时间等

一些常见指令的作用

20.2.2 协商缓存

- 如果缓存过期了，就需要发起请求验证资源是否有更新 。协商缓存可以通过设置两种 `HTTP Header` 实现： `Last-Modified` 和 `ETag`
- 当浏览器发起请求验证资源时， 如果资源没有做改变，那么服务端就会返回 `304` 状态码， 并且更新浏览器缓存有效期。

Last-Modified 和 If-Modified-Since

> `Last-Modified` 表示本地文件最后修改日期， `If-Modified-Since` 会将 `Last-Modified` 的值发送给服务器，询问服务器在该日期后资源是否有更新，有更新的话就会将新的资源发送回来， 否则返回 `304` 状态码。

但是 `Last-Modified` 存在一些弊端：

- 如果本地打开缓存文件， 即使没有对文件进行修改，但还是会造成 `Last-Modified` 被修改， 服务端不能命中缓存导致发送相同的资源
- 因为 `Last-Modified` 只能以秒计时， 如果在不可感知的时间内修改完成文件，那么服务端会认为资源还是命中了，不会返回正确的资源 因为以上这些弊端，所以在 `HTTP / 1.1` 出现了 `ETag`

ETag 和 If-None-Match

- `ETag` 类似于文件指纹， `If-None-Match` 会将当前 `ETag` 发送给服务器，询问该资源 `ETag` 是否变动，有变动的话就将新的资源发送回来 。并且 `ETag` 优先级比 `Last-Modified` 高。

> 以上就是缓存策略的所有内容了，看到这里，不知道你是否存在这样一个疑问 。如果什么缓存策略都没设置，那么浏览器会怎么处理?

对于这种情况， 浏览器会采用一个启发式的算法， 通常会取响应头中的 `Date` 减去 `Last-Modified` 值的 `10%` 作为缓存时间。

20.3 实际场景应用缓存策略

频繁变动的资源

> 对于频繁变动的资源， 首先需要使用 `Cache-Control: no-cache` 使浏览器每次都请求服务器，然后配合 `ETag` 或者 `Last-Modified` 来验证资源是否有效 。这样的做法虽然不能节省请求数量，但是能显著减少响应数据大小。

代码文件

> 这里特指除了 `HTML` 外的代码文件， 因为 `HTML` 文件一般不缓存或者缓存时间很短。

一般来说，现在都会使用工具来打包代码，那么我们就可以对文件名进行哈希处理， 只有当代码修改后才会生成新的文件名 。基于此， 我们就可以给代码文件设置缓存有效期一年 `Cache-Control: max-age=31536000` ， 这样只有当 `HTML` 文件中引入的文件名发生了改变才会去下载最新的代码文件， 否则就一直使用缓存

> 更多缓存知识详解 http://blog.poetries.top/2019/01/02/browser-cache

21 浏览器渲染原理

> 注意：该章节都是一个面试题。

21.1 渲染过程

1. 浏览器接收到 HTML 文件并转换为 DOM 树

> 当我们打开一个网页时， 浏览器都会去请求对应的 `HTML` 文件 。虽然平时我们写代码时都会分为 `JS` 、 `CSS` 、 `HTML` 文件，也就是字符串，但是计算机硬件是不理解这些字符串的，所以在网络中传输的内容其实都是 `0` 和 `1` 这

> 些字节数据 。当浏览器接收到这些字节数据以后， 它会将这些字节数据转换为字符串，也就是我们写的代码。

> 当数据转换为字符串以后， 浏览器会先将这些字符串通过词法分析转换为标记（ token ）， 这一过程在词法分析中叫做标记化（ tokenization ）

> 那么什么是标记呢？这其实属于编译原理这一块的内容了 。简单来说，标记还是字符串， 是构成代码的最小单位 。这一过程会将代码分拆成一块块， 并给这些内容打上标记，便于理解这些最小单位的代码是什么意思

> 当结束标记化后， 这些标记会紧接着转换为 Node ， 最后这些 Node 会根据不同 Node 之前的联系构建为一颗 DOM 树

> 以上就是浏览器从网络中接收到 HTML 文件然后一系列的转换过程

> 当然，在解析 HTML 文件的时候， 浏览器还会遇到 CSS 和 JS 文件， 这时候浏览器也会去下载并解析这些文件，接下来就让我们先来学习浏览器如何解析 CSS 文件

2. 将 CSS 文件转换为 CSSOM 树

> 其实转换 CSS 到 CSSOM 树的过程和上一小节的过程是极其类似的

- 在这一过程中， 浏览器会确定下每一个节点的样式到底是什么， 并且这一过程其实是很消耗资源的 。因为样式你可以自行设置给某个节点，也可以通过继承获得 。在这一过程中，浏览器得递归 CSSOM 树，然后确定具体的元素到底是什么样式。

如果你有点不理解为什么会消耗资源的话， 我这里举个例子

```html
<div>
  <a> <span></span> </a>
</div>
<style>
  span {
    color: red;
  }
  div > a > span {
    color: red;
  }
</style>
```

> 对于第一种设置样式的方式来说， 浏览器只需要找到页面中所有的 span 标签然后设置颜色，但是对于第二种设置样式的方式来说， 浏览器首先需要找到所有的 span 标签，然后找到 span 标签上的 a 标签， 最后再去找到 div 标签，然后给符合这种条件的 span 标签设置颜色， 这样的递归过程就很复杂 。所以我们应该尽可能的避免写过于具体的 CSS 选择器，然后对于 HTML 来说也尽量少的添加无意义标签，保证层级扁平

3. 生成渲染树

> 当我们生成 DOM 树和 CSSOM 树以后，就需要将这两棵树组合为渲染树

- 在这一过程中，不是简单的将两者合并就行了 。渲染树只会包括需要显示的节点和这些节点的样式信息， 如果某个节点是 display: none 的，那么就不会在渲染树中显示。
- 当浏览器生成渲染树以后，就会根据渲染树来进行布局 (也可以叫做回流)， 然后调用 GPU 绘制，合成图层， 显示在屏幕上 。对于这一部分的内容因为过于底层， 还涉及到了硬件相关的知识， 这里就不再继续展开内容了。

21.2 为什么操作 DOM 慢

> 想必大家都听过操作 DOM 性能很差，但是这其中的原因是什么呢？

- 因为 DOM 是属于渲染引擎中的东西， 而 JS 又是 JS 引擎中的东西 。当我们通过 JS 操作 DOM 的时候， 其实这个操作涉及到了两个线程之间的通信，那么势必会带来一些性能上的损耗 。操作 DOM 次数一多，也就等同于一直在进行线程之间的通信， 并且操作 DOM 可能还会带来重绘回流的情况，所以也就导致了性能上的问题。

> 经典面试题：插入几万个 DOM， 如何实现页面不卡顿？

- 对于这道题目来说， 首先我们肯定不能一次性把几万个 DOM 全部插入， 这样肯定会造成卡顿，所以解决问题的重点应该是如何分批次部分渲染 DOM 。大部分人应该可以想到通过 requestAnimationFrame 的方式去循环的插入 DOM ， 其实还有种方式去解决这个问题：虚拟滚动 （ virtualized scroller ） 。
- 这种技术的原理就是只渲染可视区域内的内容，非可见区域的那就完全不渲染了， 当用户在滚动的时候就实时去替换渲染的内容

> 从上图中我们可以发现， 即使列表很长 ，但是渲染的 DOM 元素永远只有那么几个， 当我们滚动页面的时候就会实时去更新 DOM ， 这个技术就能顺利解决这道经典面试题

21.3 什么情况阻塞渲染

- 首先渲染的前提是生成渲染树，所以 HTML 和 CSS 肯定会阻塞渲染 。如果你想渲染的越快，你越应该降低一开始需要渲染的文件大小， 并且扁平层级，优化选择器。
- 然后当浏览器在解析到 script 标签时，会暂停构建 DOM ， 完成后才会从暂停的地方重新开始 。也就是说， 如果你想首屏渲染的越快，就越不应该在首屏就加载 JS 文件， 这也是都建议将 script 标签放在 body 标签底部的原因。
- 当然在当下， 并不是说 script 标签必须放在底部， 因为你可以给 script 标签添加 defer 或者 async 属性。
- 当 script 标签加上 defer 属性以后，表示该 JS 文件会并行下载，但是会放到 HTML 解析完成后顺序执行 ，所以对于这种情况你可以把 script 标签放在任意位置。
- 对于没有任何依赖的 JS 文件可以加上 async 属性，表示 JS 文件下载和解析不会阻塞渲染。

21.4 重绘（Repaint）和回流（Reflow）

> 重绘和回流会在我们设置节点样式时频繁出现， 同时也会很大程度上影响性能。

- 重绘是当节点需要更改外观而不会影响布局的， 比如改变 `color` 就叫称为重绘
- 回流是布局或者几何属性需要改变就称为回流。
- 回流必定会发生重绘， 重绘不一定会引发回流 。回流所需的成本比重绘高的多， 改变父节点里的子节点很可能会导致父节点的一系列回流。

以下几个动作可能会导致性能问题：

- 改变 `window` 大小
- 改变字体
- 添加或删除样式
- 文字改变
- 定位或者浮动
- 盒模型

> 并且很多人不知道的是， 重绘和回流其实也和 `Eventloop` 有关。

- 当 `Eventloop` 执行完 `Microtasks` 后，会判断 `document` 是否需要更新， 因为浏览器是 `60Hz` 的刷新率，每 `16.6ms` 才会更新一次。
- 然后判断是否有 `resize` 或者 `scroll` 事件，有的话会去触发事件，所以 `resize` 和 `scroll` 事件也是至少 `16ms` 才会触发一次， 并且自带节流功能。
- 判断是否触发了 `media query`
- 更新动画并且发送事件
- 判断是否有全屏操作事件
- 执行 `requestAnimationFrame` 回调
- 执行 `IntersectionObserver` 回调，该方法用于判断元素是否可见， 可以用于懒加载上， 但是兼容性不好 更新界面
- 以上就是一帧中可能会做的事情 。如果在一帧中有空闲时间，就会去执行 `requestIdleCallback` 回调

21.5 减少重绘和回流

1. 使用 `transform` 替代 `top`

```html
<div class="test"></div>
<style>
```

```
    .test {
      position: absolute;
      top: 10px;
      width: 100px;
      height: 100px;
      background: red;
    }
  </style>
  <script>
    setTimeout(() => {
      // 引起回流
      document.querySelector( '.test').style.top = '100px'
    }, 1000)
  </script>
```

2. 使用 visibility 替换 display: none ， 因为前者只会引起重绘，后者会引发回流（改变了布局）
3. 不要把节点的属性值放在一个循环里当成循环里的变量

```
for(let i = 0; i < 1000; i++) {
    // 获取 offsetTop 会导致回流， 因为需要去获取正确的值
    console.log(document.querySelector( '.test').style.offsetTop)
}
```

4. 不要使用 table 布局， 可能很小的一个小改动会造成整个 table 的重新布局
5. 动画实现的速度的选择， 动画速度越快， 回流次数越多，也可以选择使用 requestAnimationFrame
6. CSS 选择符从右往左匹配查找， 避免节点层级过多
7. 将频繁重绘或者回流的节点设置为图层， 图层能够阻止该节点的渲染行为影响别的节点。比如对于 video 标签来说， 浏览器会自动将该节点变为图层。

> 设置节点为图层的方式有很多， 我们可以通过以下几个常用属性可以生成新图层

- will-change
- video 、 iframe 标签

22 安全防范

22.1 XSS

> 涉及面试题：什么是 XSS 攻击？如何防范 XSS 攻击？什么是 CSP ？

- XSS 简单点来说，就是攻击者想尽一切办法将可以执行的代码注入到网页中。
- XSS 可以分为多种类型，但是总体上我认为分为两类：持久型和非持久型。
- 持久型也就是攻击的代码被服务端写入进数据库中， 这种攻击危害性很大， 因为如果网站访问量很大的话，就会导致大量正常访问页面的用户都受到攻击。

> 举个例子，对于评论功能来说，就得防范持久型 XSS 攻击， 因为我可以在评论中输入以下内容

- 这种情况如果前后端没有做好防御的话， 这段评论就会被存储到数据库中， 这样每个打开该页面的用户都会被攻击到。
- 非持久型相比于前者危害就小的多了，一般通过修改 URL 参数的方式加入攻击代码，诱导用户访问链接从而进行攻击。

> 举个例子， 如果页面需要从 URL 中获取某些参数作为内容的话，不经过过滤就会导致攻击代码被执行

```html
<!-- http://www.domain.com?name=<script>alert(1)</script> -->
<div>{{name}}</div>
```

> 但是对于这种攻击方式来说， 如果用户使用 Chrome 这类浏览器的话， 浏览器就能自动帮助用户防御攻击 。但是我们不能因此就不防御此类攻击了， 因为我不能确保用户都使用了该类浏览器。

> 对于 XSS 攻击来说， 通常有两种方式可以用来防御。

1. 转义字符

> 首先，对于用户的输入应该是永远不信任的 。最普遍的做法就是转义输入输出的内容，对于引号 、尖括号 、斜杠进行转义

```js
function escape(str) {
  str = str.replace(/&/g, '&amp;')
  str = str.replace(/</g, '&lt;')
  str = str.replace(/>/g, '&gt;')
  str = str.replace(/"/g, '&quto;')
  str = str.replace(/'/g, '&#39;')
  str = str.replace(/`/g, '&#96;')
  str = str.replace(/\//g, '&#x2F;')
  return str
}
```

> 通过转义可以将攻击代码 <script>alert(1)</script> 变成

```js
// -> &lt;script&gt;alert(1)&lt;&#x2F;script&gt;
escape('<script>alert(1)</script>')
```

> 但是对于显示富文本来说， 显然不能通过上面的办法来转义所有字符， 因为这样会把需要的格式也过滤掉 。对于这种情况， 通常采用白名单过滤的办法， 当然也可以通过黑名单过滤，但是考虑到需要过滤的标签和标签属性实在太多，更加推荐使用白名单的方式

```js
const xss = require('xss')
let html = xss('<h1 id="title">XSS Demo</h1><script>alert("xss");</script>')
// -> <h1>XSS Demo</h1>&lt;script&gt;alert("xss");&lt;/script&gt;
console.log(html)
```