精选

盒子模型

盒子模型可分为两种，W3C标准盒模型、IE怪异盒模型。盒模型有四中属性：外边距(margin)、边框(border)、内边距(padding)、实际内容(content)

标准盒模型计算

css中的`box-sizing:content-box;`

盒子的width/height = content的宽高

盒子的总宽高 = width/height + padding + border + margin

怪异盒模型

css中的`box-sizing:border-box;`

盒子的width/height = border + padding + content

盒子的总宽高 = 盒子的width/height + margin

浏览器缓存

强缓存

不会向服务器发送请求，主要以客户端来判断，直接从Memory Cache或者从Disk Cache 中拿，请求返回状态为200。强缓存通过HTTP header来设置。强缓存分为两种

1. Expires

   HTTP1.0的产物，它的值是一个具体的时间值，表示资源会在这个时间值之后过期。

   缺点：受限于本地时间，如果修改了本地时间，会造成缓存失效。

2. Cache-Control

   在HTTP1.1的版本中，通过指定指令来实现缓存机制，可以设置多少秒后过期，一般设置`Cache-control:max-age=秒`，Cache-control有多个缓存指令可以一起配合使用。

Cache-Control会优先于Expires。Expires现阶段为了向下兼容其他低版本浏览器

强缓存的缺点：强缓存判断是否超出某个时间或者某个时间段，而不关心服务端文件是否有更新，这个导致加载的文件可能不是服务器最新的内容，这时候需要协商缓存。

协商缓存

协商缓存会在强缓存失效后（比如强制刷新浏览器）浏览器携带标识向服务器请求，由服务器根据协商缓存标识决定是否使用缓存。强缓存分为两种

1. Last-Modified和If-Modified-Since

浏览器第一次请求资源，服务器返回的同时响应头中添加Last-Modified 字段，它的值表示这个资源在服务器最后的修改时间。浏览器下一次请求检查到Last-Modified这个字段，浏览器会添加If-Modified-Since这个字段，它的值就是Last-Modified中的时间值。服务器收到这个资源请求会根据传过来的If-Modified-Since的值于服务器中这个资源的最后修改时间对比，如果没有变化返回304和空的响应体。如果If-Modified-Since的值小于服务器中这个资源的最后修改时间，说明文件有更新，于是返回新的资源文件和200的状态。

缺点：如果打开本地缓存的文件，即使没有对文件做修改，Last-Modified的值也会改变，服务器不能命中导致发送相同的资源。

2. ETag表示将字符串

   ETag是由服务器为每个资源生成的唯一标识字符串，这个标识字符串是基于文件内容编码，只要文件不同，他们对应的ETag就不同

   浏览器在下一次向服务器发送请求，会将上一次返回的ETag值放到请求头里的If-None-Match字段中，服务器对比客服端传来的If-None-Match跟自己服务器上该资源的ETag是否一致。如果不匹配那么正常去请求返回200和新的ETag发送客户端。如果ETag一致，则返回304给客户端直接使用本地缓存即可

   ETag会优先于Last-Modified命中

HTTP请求响应的内容有哪些?

请求

1. 请求行 例如: POST https://www.baidu.com HTTP1.1
   - 请求方法（GET、POST、HEAD、PUT、DELETE、OPTIONS、PATCH）
   - 请求URL
   - HTTP协议版本
2. 请求头 就是Chorme中的Request Header中的键值对
   - User-Agent:浏览器类型
   - Accept:客户端可识别的内容类型列表
   - Host:主机地址
   - Content-Type
   - Cookie
   - ...
3. 请求主体 (一些请求参数，form data 等等)

响应

1. 响应行。包含协议/版本，响应的状态码,对响应状态码的描述
2. 响应头。与请求头一样
3. 响应主体。和网页右键"查看源码"看到的内容一样

get和post的区别

1. get和post各自的语义不同，get获取资源，post提交资源
2. get会把请求头和请求体一起发送出去，post浏览器会先发送请求头，服务器响应100 *continue*，然后在发送请求体
3. get在浏览器回退是无害的，而post会再次提交
4. get请求会被浏览器主动cache，而post不会
5. get请求只能进行url编码，而post支持多种编码方式

6. get请求url中的传送参数是有长度限制的，而post没有
7. get比post安全
8. get参数通过url传递，post放在Request body中

常见的HTTP状态码

- 1xx:指示信息，表示请求已接收，继续处理
- 2xx:成功，表示请求已被成功接受
    - 200：客户端请求成功.
    - 206：客户端发送一个带有Range头的Get请求，服务器完成了
- 3xx：服务器虽然也处理了你的请求，但客户端还需要进一步的工作，才可以完成请求
    - 301：永久性重定向，表示资源已被分配了新的 URL
    - 302：临时性重定向，表示资源临时被分配了新的 URL
    - 304：表示服务器校验后发现资源没有改变，提醒客户端直接走缓存来取资源
- 4xx：客户端错误，请求有语法错误或者请求无法实现
    - 400：请求报文存在语法错误
    - 401：请求未经授权
    - 403：被请求页面的访问被禁止
    - 404：请求资源不存在
    - 405：资源被禁止，不允许使用请求行中所指定的方法
- 5xx：服务端错误，服务器未能实现合法的请求
    - 500 服务器发生不可预期的错误，原来缓存的文档还可以继续使用
    - 502：网关错误
    - 503 请求未完成，服务器临时过载或宕机，一段时间后可能恢复正常
    - 504：网关超时

HTTP2.0有哪些新特性

二进制分帧

在 HTTP1.x 中，数据以文本的格式进行传输，解析起来比较低效。

HTTP2.0 在传输消息时，首先会将消息划分为更小的消息和帧，然后再对其采取二进制格式的编码，确保高效的解析。

头部压缩

HTTP2.0 中，客户端和服务器分别会维护一份相同的静态字典，这个字典用来存储常见的头部名称，以及常见的头部名称和值的组合。同时还会维护一份相同的动态字典，这个字典可以实时被更新。

如此一来，第一次相互通信过后，后面的请求只需要发送与前面请求之间头部不同的地方，其它的头部信息都可以从字典中获取。相对于 HTTP1.x 中每次都要携带整个头部跑来跑去的笨重操作来说，大大节省了网络开销。

服务端推送

在 HTTP1.x 中，如果用户请求了资源 A，结果发现自己如果要用资源 A，那么必须依赖资源 B，这时他不得不再消耗一个请求。

而 HTTP2.0 中，允许服务器主动向客户端 push 资源。也就是说当服务器发现客户端请求了资源 A，却忘了请求资源 A 依赖的资源 B 时，它可以主动将资源 B 顺手推送给客户端。

这样一来，当客户端发现自己漏掉一个必要请求的时候，直接从缓存中就可以读到资源 B 了，而不必再消耗一个请求。

多路复用

在 HTTP 2.0 中，一次连接建立后，只要这个连接还在，那么客户端就可以在一个链接中批量发起多个请求。同时，请求与请求间完全不阻塞，A 请求的响应就算没回来，也不影响 B 请求收到自己的响应。请求与请求间做到了高度的独立，真正实现了并行请求。由此，彻底规避了队头阻塞问题。

React 生命周期

React 15以下的生命周期

实例化：

1. `constructor`
2. `componentWillMount`
3. `render`
4. `componentDidMount`

`props` 或者 `states` 状态发生改变:

1. `componentWillReceiveProps` 如果是states状态发生改变是不会触发这个生命周期
2. `shouldComponentUpdate(nextProps,nextState)` 如果是true会有后续生命周期，如果false就没有后续了
3. `componentWillUpdate`
4. `render`
5. `componentDidUpdate`

卸载

1. `componentWillUnmount`

React 16版本的class组件生命周期

实例化：

1. `constructor`
2. `static getDerivedStateFromProps(nextProps,prevState)`
3. `render`
4. `componentDidMount`

更新时：

1. `static getDerivedStateFromProps(nextProps,prevState)`

   一个静态函数，静态函数中不能使用this访问到class

   当父组件和自己本身状态改变时：

`nextProps`:最新的父组件传过来的`props`
`prevState`:当前自己组件最新的的`state`
返回一个对象来更新 `state` 或者返回 `null` 来表示接收到的 `props` 没有变化，不需要更新 `state`

2. `shouldComponentUpdate(nextProps, nextState)`

   当父组件状态变更时：

   `nextProps`:最新的父组件传过来的`props`,当前的`this.props`是上一次的`props`

   `nextState`:当前最新的`state`,当前的`this.state`是最新的`state`

   当自己组件状态变更时：

   `nextProps`:最新的父组件传过来的`props`,当前的`this.props`最新的父组件传过来的`props`

   `nextState`:当前最新的`state`,当前的`this.state`是上一次的`state`

   如果是true会有后续生命周期，如果false就没有后续了

3. `render`

4. `getSnapshotBeforeUpdate(prevProps, prevState)`

   当父组件状态变更时：

   `prevProps`:最新的父组件上一次传过来的`props`,当前的`this.props`是最新的`props`

   `prevState`:当前最新的`state`,当前的`this.state`是最新的`state`

   当自己组件状态变更时：

   `prevProps`:最新的父组件传过来的`props`,当前的`this.props`最新的父组件传过来的`props`

   `prevState`:是上一次的`state`,当前的`this.state`是最新`state`

   接收父组件传递过来的 props 和组件之前的状态,此方法必须有返回值，返回值作为第三个参数传递给`componentDidUpdate`，必须和 `componentDidUpdate` 一起使用，否则会报错

5. `componentDidUpdate`(prevProps, prevState, xx)

   基本上同`getSnapshotBeforeUpdate`一致，第三个可选参数是`getSnapshotBeforeUpdate`返回的值

卸载

1. `componentWillUnmount()`

React diff算法

当对比两棵树不同时，使用逐层对比。diff算法会优先比较两棵树的根节点，如果他们的类型不同，比如之前的是div，现在变成p标签，那么就认为这两颗树完全不同，这是两个完全不同的组件。因此也没有必要再往下对比，直接把div删掉，重建为p。也就是卸载旧组件挂载新组件。

若根节点相同，在保留这个组件的基础上，检查其属性的变化，然后根据属性变化的情况去更新组件。

处理完根节点这个层次的对比，React会继承调到下个层次去对比子节点们。子节点的对比思路和根节点是一致的。

如果是同一层次的一组子节点，他们可以通过唯一的id进行区分，也就是key

React fiber了解

> Fiber 是React16引入的一种新的调和引擎

Fiber解决了什么问题

在Fiber架构前，当React决定要加载或者更新组件树时，会有一个大的动作。

这个动作包括生命周期的调用、diff过程的计算、DOM树的更新等等。这个动作很大，耗时很长，而且是同步进行的，一旦开始就不能中断。这意味这你的组件挂载或者更新结束前，什么都不能去做。

如果更新一个组件需要1毫秒，如果需要更新1000个组件，就会耗时1秒，在这1秒更新的过程中，主线程都在专心运行更新操作。浏览器绘制屏幕一般来说这个频率是每秒60次。也就说每16毫秒渲染1帧。如果执行js的时间过长超过16毫秒之后用户能察觉到页面卡顿，导致页面体验下降。

因为js单线程的特点，每个同步任务不能耗时太久，不然就会让程序不会对其他输入做出反应。React以前更新过程中就会出现上述情况，而现在React Fiber就要改变现状。

Fiber思想

面对单个任务耗时过长这个问题，把一个庞大的任务分成N多个微小的任务，这些个小任务就是一个fiber单元。每个fiber单元每次执行超过一定的时间意味着该任务需要暂停一下让出主线程。这个暂停一下意义重大，React会在这段时间里查找有没有优先级更高的事情需要处理。以此确保主线程总在做它当下最应该做的事情。

Fiber与生命周期

React在渲染之前会有两个工作阶段：

render/reconcilition:调和阶段。这个阶段就是diff过程。这个过程里，React在内存中做计算，不涉及真实DOM操作，也就是说你打断执执行、重复执行、用户都是不感知的。最后确认所有的更新行为。由于"切片"和“暂停”两个关键特性的实现，调和过程变成了一个可以被打断暂停的过程，查看优先级更高的任务并执行。有以下生命周期可以被打断：

1. getDerivedStateFromProps
2. shouldComponentUpdate
3. render

commit:执行调和阶段的计算结果，真正地去更新DOM，这个过程不允许被打断。会一直执行下去，这个过程涉及的生命周期有：

1. getSnapshotBeforeUpdate
2. componentDidMount
3. componentDidUpdate
4. componentWillUnmount

React性能优化

内在优化

1. class组件
    - 使用 `prueComponent` 自动对比组件的props来达到减少渲染次数
    - 使用 `shouldComponentUpdate` 生命周期函数来控制是否渲染避免重复渲染
    - 当类组件实例化的时候绑定好每个方法中的this，而不是在render dom中去重复绑定this
    - 当需要 `setState` 的前去判断状态有没有改变，如果有改变在去调用 `setState`。
    - 传递 `props` 只传递有用的 `props`，避免 `{...this.props}` 这样传递。

- 复杂组件尽量拆分小组件
- 使用`return null`而不是css的`display`来控制节点的显示隐藏。

2. 函数组件
   - 使用memo来自动对比props来达到减少渲染次数
   - 使用`useCallback、useMemo`来缓存函数，让Hooks根据后面的依赖判断是否重新创建函数
3. 其他
   - 使用不可变的数据结构，例如Immutable.js。
   - 使用react lazy或者第三方包的loadable.js来达到组件或者页面按需加载
   - 服务端渲染react

外在优化

1. webpack
   - 开启code splitting
   - 打包动态链接库dll.js
   - 把dll.js放入cdn
2. chorme Performance Tools分析每个阶段的性能

React this.setState

`setState`只在合成事件和钩子函数中是“异步”，在原生事件和`setTimeout`中是同步。

setState的“异步”并不是说内部由异步代码实现，其实本身执行的过程和代码都是同步的，只是合成事件和钩子函数的调用顺序在更新之前，导致在合成事件和钩子函数中没法立马拿到更新后的值，形成了所谓的“异步”，当然可以通过setState第二个函数中的callback拿到更新后的结果。

setState的批量更新优化也是建立在“异步”（合成事件、钩子函数）之上的，在原生事件和setTimeout中不会批量更新，在“异步”中如果对同一个值进行多次setState,setState的批量更新策略会对其覆盖，取最后一次的执行，如果是同时setState多个不同的值，在更新时会对其进行合并批量更新。

MVC的理解

MVC是一种架构模式。MVC模式把软件系统分为三个基本部分：

- 模型 Model 存放应用所有的数据，数据来源可能是接口或者本地缓存等。
- 视图 View 展示给用户的界面，可以接受用户输入。
- 控制器 Controller 负责连接Model和View,从View获取输入，修改相关Model的数据后，再去通知相关View进行更新

mvvm是Model-View-ViewModel。本质就是MCV的改进版，View和Model双向绑定。

从输入URL到渲染出整个页面的过程包括三个部分

一、 DNS解析URL的过程

> DNS解析的过程就是寻找哪个服务器上有请求的资源。因为ip地址不容易记忆，一般会使用URL域名（如www.baidu.com）作为网址。DNS解析就是将域名翻译成IP地址的过程。

1. 浏览器缓存：浏览器会按照一定的频率缓存DNS记录

2. 操作系统缓存：如果浏览器中找不到需要的DNS记录，就会去操作系统中找比如windows的host文件
3. 路由器缓存：路由器也有DNS缓存
4. ISP(互联网服务提供商)的DNS服务器：ISP有专门的DNS服务器应对DNS查询请求
5. 根服务器：ISP的DNS服务器找不到之后，就会向根服务器发出请求，并进行递归查询

二、浏览器与服务器交互过程

1. 从上面步骤中取得IP地址，浏览器利用TCP协议通过三次握手与服务器建立连接
2. 浏览器根据TCP连接发送http的get请求报文。等待响应
3. 服务器接收到http请求之后，处理http请求报文，返回响应报文。
4. 若状态码为200,浏览器收到返回的页面，开始进行页面渲染

三、浏览器页面渲染过程

1. 浏览器根据深度遍历的方式把html节点遍历成dom树
2. 将css解析成cssom树
3. 将dom树和cssom树组合成render树
4. 根据得到的render树计算所有的节点在屏幕上的位置，进行布局（回流）
5. 遍历render树并调用硬件API绘制所有的节点（重绘）

根据从输入URL到渲染出整个页面的过程做出性能优化

一、网络阶段

1. 减少请求次数
    - 使用精灵图合并小图标
    - 使用字体图标
    - 合并css和js文件
    - 按需加载
    - 合理设置HTTP缓存
2. 减少单次请求所花费的时间
    - 压缩js和css文件
    - 压缩图片文件
    - 使用CDN
    - 利用浏览器缓存
    - 减少cookie传递，比如图片请求时候也会带上cookie，利用不同域名的图片服务器解决问题
    - 开启Gzip压缩传输

二、渲染阶段

1. 服务端渲染
2. js文件放在html的底部
3. 使用script标签的defer 和async属性优化加载方式
4. 减少dom操作，避免回流和重绘
5. 耗时的操作使用异步处理方式
6. 使用节流和防抖的功能

浏览器渲染过程

用户看到页面渲染实际上可分为两个阶段：

- `DOMContentLoaded`事件触发时，仅当DOM加载完成，不包括样式表，图片资源等。
- `load`事件触发时，页面上所有的DOM，样式表，脚本，图片等资源都已加载完成。

实际浏览器渲染的过程主要包括以下五步：

1. 浏览器将获取的HTML文档根据深度遍历的方式把HTML的节点遍历成dom树
    - DOM树在构建的过程中可能会被css和js的加载而阻塞
    - `display:none`的元素也会在DOM树中
    - 注释也会在DOM树中
    - `script`标签会在DOM树中
2. 处理css标记，构成层叠样式表模型CSSOM(CSS Object Model)规则树
    - css解析可以与DOM解析同时进行
    - css解析与`script`的执行互斥
    - 在Webkit内核中进行了`script`执行优化，只有在JS访问CSS时才会发生互斥。
3. 将DOM树和CSSOM合并为渲染树(`rendering tree`)，代表一系列将被渲染的对象
    - Render Tree和DOM Tree不完全对应
    - `display:none`的元素不在Render Tree中
    - `visibility:hidden`的元素不在Render Tree中
4. 渲染树的每个元素包含的内容都是被计算过的，然后根据计算得到每个节点在屏幕上的位置，进行布局(`layout` 回流阶段)
    - 布局阶段的输出就是我们常说的盒子模型，它会精确地捕获每个元素在屏幕内的确切位置与大小。
    - `float`元素，`absoulte`元素，`fixed`元素会发生位置偏移。
    - 我们常说的脱离文档流，其实就是脱离Render Tree。
5. 遍历render树并调用硬件API绘制所有的节点(`painting` 绘制阶段)

对闭包的理解

变量的作用域

在闭包中函数就像一层半透明的玻璃，在函数里面可以看到外面的变量，而在函数外面则无法看到函数里面的变量。这是因为当在函数中搜索一个变量的时候，如果该函数内并没有声明这个变量，那么此次搜索的过程会随着代码执行环境创建的作用域链往外层逐层搜索，一直搜索到全局对象位置。变量的搜索是从内到外而非从外到内的。

无论通过何种手段将内部函数传递到所在的词法作用域以外，它都会持有对原始定义作用域的引用，无论在任何处执行这个函数都会使用闭包。

变量的生存周期

当一个函数执行时，函数会被推入执行栈，执行函数中内容，执行完成函数会被推出执行栈，同时销毁函数中的变量。但是如果是函数A中返回一个函数B，返回的函数B使用了函数A里面的变量，就相等于函数A里面的变量被外界访问了，函数A里面的变量就有不被销毁的理由。

这时候函数B就产生了一个闭包结构，函数A中的变量生命看起来被延续了。

闭包的作用

闭包可以实现私有变量、特权变量、存储变量等

闭包的表现

- 函数作为参数被传递、
- 函数作为返回值

对原型链的理解

原型

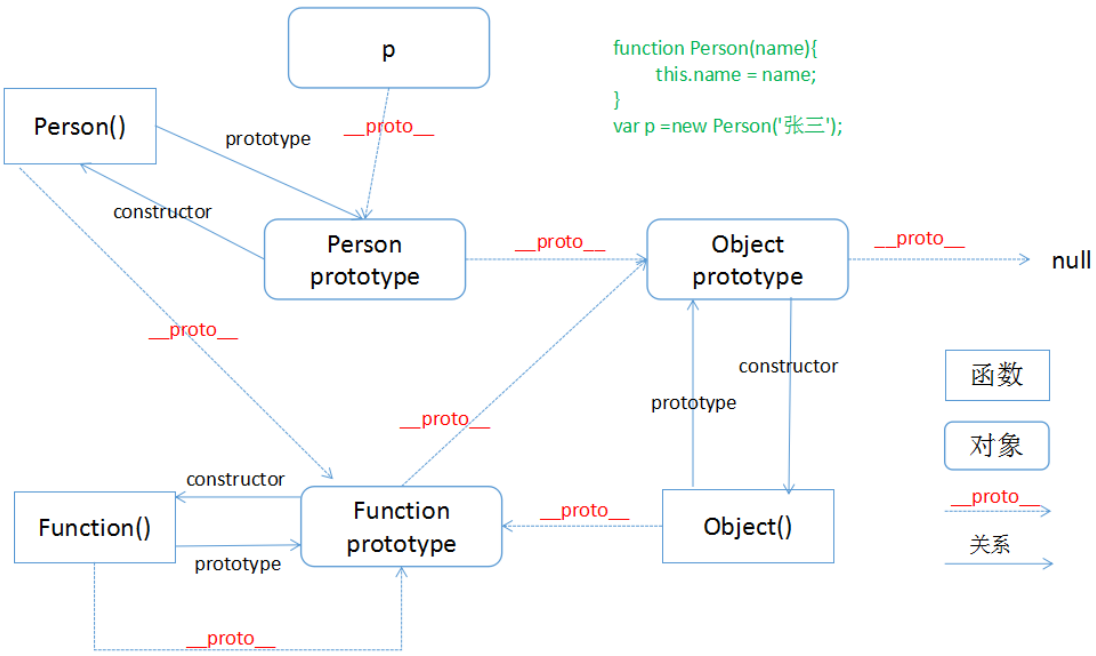
每个函数都有一个`prototype`属性，这个属性是一个指针，指向一个对象,这个就是原型对象，它里面有一个属性`construcotr`属性，其值指向函数本身，`prototype`可以自定义增加许多属性。实例化这个函数，实例对象就能继承构造器函数上的对象和方法。

原型链

每个函数或者对象都有一个`__proto__`隐藏属性，这个引用了创建这个对象的函数的`prototype`，这样这个对象就和创建它的函数关联起来。如果这个对象本身没有这个属性和方法，它就会顺着`__proto__`向创建它的函数的原型中查找。这样就行了原型链。

画出下题的完整原型链

```JavaScript
function Person(name){
    this.name = name;
}
var p = new Person('张三');
```



总结：所有的函数`__proto__`都指向`Function prototype`,包括`new Function()`本身。所有的函数`prototype`对象的`__proto__`(除了Object)都指向`Object prototype`。`Object prototype`对象的`__proto__`指向`null`。对象只有`__proto__`属性。

commonjs模块和es6 module区别

Commonjs

1. 对于基本数据类型，属于赋值。即会被模块缓存。同时，在另一个模块可以对该模块输出的变量重新赋值。
2. 对于复杂数据类型，属于浅拷贝。由于两个模块引用的对象指向同一个内存空间，因此对该模块的值做修改时会影响到另一个模块。
3. 当使用require命令加载某个模块时，就会运行整个模块的代码。
4. 当使用require命令加载某一个模块时，不会在执行该模块，而是去取缓存中的值。也就是说，CommonJS模块无论加载多少次，都只会在第一次加载时运行，以后在加载，就会返回第一次运行的结果，除非手动清除系统缓存。
5. 循环加载时，属于加载时执行。即脚本代码require的时候，就会全部执行。一旦出现某个模块被"循环加载"，就只输出已经执行的部分，还未执行的部分不会输出。

ES6模块

1. ES6模块中的值属于动态只读引用
2. 对于只读来说，即不允许修改引入变量的值，import的变量是只读的，不论是基本数据类型还是复杂数据类型。当模块遇到`import`命令时，就会生成一个只读引用。等到脚本真正执行时，在根据这个只读引用，到被加载的那个模块里面去取值。
3. 对于动态来说原始值发送变化，`import`加载的值也会发生变化。不论是基本数据类型还是复杂数据类型。
4. 循环加载时，ES6模块是动态引用。只要两个模块之间存在某个引用，代码就能够执行。
5. 静态化，必须在顶部，不能使用条件语句，自动采用严格模式
6. 能得到webpack treeshaking和编译优化，以及webpack中的作用域提升

两者最大差异

- Commonjs模块输出的是一个值的拷贝，ES6模块输出的是值的引用。
- Commonjs模块是运行时加载，es6模块是编译时输出接口。

浏览器的Event Loop

在浏览器的事件循环中有三个角色：

- 函数调用栈
- 宏任务(macro-task)队列
- 微任务(micro-task)队列

函数调用栈

当js引擎第一次遇到js代码时，会产生一个全局执行上下文并压入调用栈。后面每遇到一个函数调用，就会往栈中压入一个新的函数上下文。JS引擎会执行栈顶的函数，执行完毕后弹出对应的上下文。

任务队列

某些任务，他们不需要立刻被执行，所以它们在刚刚被派发的时候，并不具备进入调用栈的资格，于是这些等待执行的任务，按照一定的规则排起长度，等待被推入调用栈，这个队列就叫做任务队列。

任务队列分为**宏任务**和**微任务**：

常见的**宏任务**有：

- setTimeout
- setInterval
- setImmediate（Nodejs独有）
- script（整体代码）
- I/O 操作等

常见的**微任务**有：

- process.nextTick（Nodejs独有）
- Promise
- MutationObserver 等

循环过程解读

一个完整的Event Loop过程，可以概况为一下阶段：

1. 执行并出队一个宏任务。注意如果是初始状态：调用栈空。微任务队列空，宏任务队列有且只有一个script脚本（整体代码）。这时执行并出队的就是script脚本全局上下文。
2. 全局上下文(script标签)被推入调用栈，同步代码执行。在执行的过程中，通过对一些接口的调用，可以产生新的宏任务与微任务，它们会分别被推入各自的任务队列里。这个过程本质上是队列的宏任务的执行和出队的过程。
3. 上一步我们出队的是一个宏任务，这一步我们处理是微任务。但需要注意的是：当宏任务出队是，任务是一个一个执行的；而微任务出队时，任务是一队一队执行的。因此，我们处理微任务队列这一步，会逐个执行队列中的任务把他们出队，直到队列被清空。
4. 执行渲染操作，更新界面。
5. 检查是否存在web worker任务，如果有，则对其进行处理。

总结

先执行宏认为，在执行微任务，这就是一个循环。

因为整个js代码就是一个宏任务，所以可认为宏任务先执行，当整个js代码执行完毕后，也就是第一个宏任务执行完了，转而去执行微任务，把当前微任务队列中的所有微任务执行完，这样一次循环就结束了。

第二次事件循环，首先从宏任务队列拿出一个宏任务去执行，执行完毕再去微任务队列，将里面的所有微任务全部执行（因为宏任务代码里面可能又创建了新的微任务）。

接下来的循环类似，即在一次循环中，宏任务执行一个，微任务执行一队。

Node和浏览器的Event Loop的区别

Node架构组成你

- 应用层：Node.js代码，包括Node应用以及一些标准库
- 桥接层：Node底层是C++实现的。桥接层负责封装底层依赖的C++模块将其简化为API向应用层提供服务。
- 底层依赖：这里就是最底层的C++库。其中前端比较关注的是V8和libuv
    - V8是js运行引擎，它负责把js代码转换为C++,然后去跑这层C++代码
    - libuv:它对跨平台的异步I/O能力进行封装，Node中的事件循环就是由libuv来初始化的

libuv中的Event-Loop实现

libuv主导循环机制共有六个循环阶段：

- **timers阶段**：执行`setTimeout`和`setInterval`回调
- **pending callbacks**:被挂起的回调，如果网络I/O或者文件I/O的过程中出现了错误，就会在这个阶段处理错误的回调。
- **idle，prepare**:系统内部使用。
- **poll（轮询阶段）**:重点阶段，这个阶段会执行I/O回调，同时还会检查定时器是否过期
- **check（检查阶段）**:处理`setImmediate`中的定义回调
- **close callbacks**:处理一些"关闭"的回调，比如`socket.on('close',...)`就会在这个阶段被触发

宏任务与微任务

Node中的宏任务和微任务和浏览器中的一致。

重点的三个循环阶段

1. **timer**

   timers阶段会执行`setTimeout`和`setInterval`回调，，并且是由poll阶段控制的。

2. **poll**

   这一阶段中，系统会做两件事情

   - 回到timer阶段执行回调
   - 执行I/O回调

   并且在进入该阶段时如果没有设定timer的话，会发生以下两件事情

   - 如果poll队列不为空，会遍历回调队列并同步执行，直到队列为空或者达到系统限制。
   - 如果poll队列为空，会有两件事情：
     - 如果`setImmediate`回调需要执行，poll阶段会停止并且进入到check阶段执行回调。
     - 如果没有`setImmediate`回调需要执行，会等待回调被加入队列中并执行回调，这里会有个超时设置防止一直等待下去，

   同样设定了timer的话poll队列为空，则会判断是否有timer超时，如果有的话timer阶段执行回调

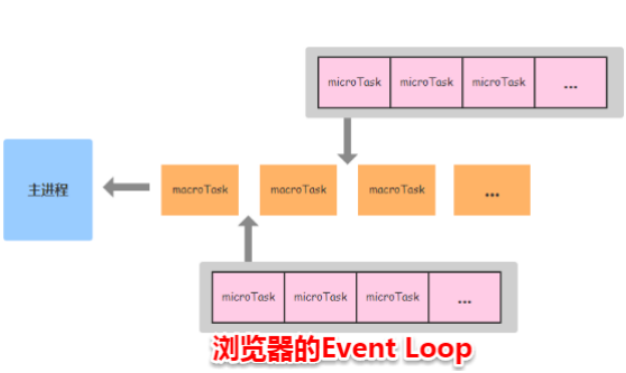
3. **check阶段**

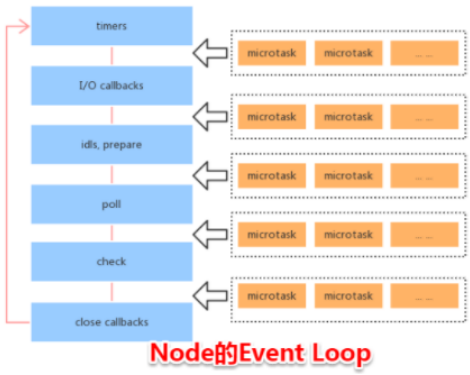
   check 阶段执行`setImmediate`

总结

浏览器和Node环境下，微任务队列的执行时机不同

- 浏览器端，每执行完一个宏任务会把这个宏任务下的一队微任务执行完。
- Node，微任务在事件循环的各个阶段之间执行。





this指向

1. 函数是箭头函数 ===> this绑定的是包裹箭头函数的函数。
2. bind、call、apply ===> this绑定是这些函数中的第一个参数
3. 普通函数
    1. new函数 ===> this绑定的是实例化出来的这个对象
    2. 非new函数
        1. 直接调用 ===> this指向全局环境
        2. 对象.调用 ===> this指向的是该对象

箭头函数与普通函数有什么不同

1. 箭头函数是匿名函数，不能作为构造函数，不能使用`new`
2. 箭头函数不绑定`arguments`
3. 箭头`this`永远指向其上下文的this，任何方法都改变不了其指向，如`call bind apply`
4. 箭头函数没有原型属性
5. 箭头函数不能当做`generator`函数，不能使用`yield`关键字

for in 和for of的区别

`for of`是作为遍历所有数据结构的统一方法，一个数据结构只要部署了`Symbol.iterator`属性，就被视为具有iterator接口，就可以使用`for of`遍历它的成员。包括遍历`map set`

`for in`循环遍历对象的属性。

`for in` 循环特别适合遍历对象。

webpack production模式下的打包优化

1. tree shaking剔除没有用到的代码
2. scope hoisting 分析模块关系，推断结果，使webpack打包出来的代码文件更小。
3. 代码压缩、混淆

tree shaking

tree shaking通常用于打包时移除javascript中未引用的代码。其原理是依赖ES模块系统中的`import`和`export`的静态结构特征

开发时引入一个模块后，如果只使用其中的一个功能，上线打包时只会把用到的功能打包到bundle，其它没有用到的功能不会打包进来，可实现最基础的优化。

scope hoisting

scope hoisting的作用是将模块之间的关系进行结果推测，可以将webpack打包出来的代码文件更小、运行的更快

scope hoisting的实现原理：分析出模块之间的依赖关系，尽可能的把打散的模块合并到一个函数中去，但前提是不能造成代码冗余。因此只有那些被引用了一次的模块才能被合并。

由于scope hosisting需要分析模块之间的依赖关系，因此源码必须采用es6模块系统，不然它将无法生效。原理和tree shaking一样。

webpack打包过程

当webpack处理应用程序时，它会从入口文件开始递归地构建一个依赖关系图，其中包含应用程序需要的每一个模块，然后将所有这些模块打包成一个或多个bundle。

webpack打包步骤：

1. 初始化参数，从配置文件和Shell语句中 读取与合并参数，得出最终参数
2. 用上一步得到的参数初始Compiler对象，加载`webpack.config.js`的配置。加载所有配置的插件，通过执行对象run方法开始执行编译
3. 确定入口，根据配置中的Entry找出所有入口文件。使用node中的fs去读取文件。
4. 编译模块，从入口文件出发，调用所有配置的Loader对模块进行编译，再找出模块依赖的模块，在递归本步骤，直所有入口依赖的文件都经过了本步骤的处理
5. 完成模块编译，在经过第四步使用Loader翻译完所有模块后，得到每个模块被编译后的最终内容以及它们之间的依赖关系
6. 输出文件。根据入口和模块之间的依赖关系，组装成一个个包含多个模块的Chunk，再将每个Chunk转化成一个单独的文件加入输出列表中。
7. 输出完成。在确定好输出内容后，根据配置确定输出的路径和文件名，将文件的内容写入文件系统中。

整个流程分为3个阶段，初始化、编译、输出。而在每个阶段又会发生很多事件，Webpack会将这些事情用tapable进行事件广播出来供Plugin使用。

webpack的loader原理

loader就像一个翻译员，能将源文件经过转化后输出新的结果，并且一个文件还可以链式地经过多个loader翻译。loader其实就是一个Node.js模块，这个模块需要导出一个函数。函数其中的参数source(源文件代码)是compiler对象传递的。一般使用`loaderUtils`这个库来获取loader中的参数。

webpack的插件原理

webpack的插件主要基于Tapable实现的，tapable是webpack项目组的一个内部库，主要抽象了一套事件，这些事件贯穿整个webpack的生命周期。

- 插件是一个独立的模块
- 模块对外暴露一个js函数
- 函数的原型上定义一个注入complier对象的apply方法。apply方法中需要通过complier对象挂载的webpack事件钩子，钩子的回调中能拿到当前编译的compilation对象，如果是异步编译插件的话可以拿到回调callback
- 完成自定义编译流程并处理compliation对象的内部数据
- 如果异步编译插件的话，数据梳理完后执行callback回调

webpack Complier和Compliation

complier对象是webpack的编译器对象，complier对象会在启动webpack的时候被一次性地初始化。complier对象中包含了所有webpack可自定义操作的配置。例如loader的配置、plugin的配置、entry的配置等各种原始webpack配置等

compilation实例继承与complier,compliation对象代表了一次单一的版本webpack构建和生成编译资源的过程。

当运行webpack开发环境中间件时，每当检测到一个文件的变化，一次新的编译将被创建，从而生成一组新的编译资源以及新的compliation对象。一个compliation对象包含了当前模块资源、编译生成资源、变化的文件、以及跟踪依赖的状态信息。编译对象也提供了很多关键点回调供插件做自定义处理时选择使用

根据区别：**Complier代表了整个webpack从启动到关闭的生命周期，而Compliation只代表一次新的编译**

webpack热更新原理

webpack-dev-server启动本地服务，内部实现主要使用了webpack,express,webscoket。

热更新过程

- 使用`express`启动本地服务，当浏览器访问资源时对此做响应
- 服务端和客户端使用webscoket实现长连接
- webpack监听源文件的变化，即当开发者保存文件时触发webpack的重新编译
    - 每次编译都会生成新的hash值、已改动模块的json文件、已改动模块代码的js文件
    - 编译完成后通过scoket向客户端推送当前编译的hash戳
- 客户端的websocket监听到有文件改动推送过来，会拿推送过来的hash戳和上一次对比
    - 一致则走缓存
    - 不一致则通过ajax和jsonp向服务端获取最新资源
- 使用内存文件系统去替换有修改的内容实现局部刷新

webpack性能优化

- 缩小文件搜索范围，loader编译范围
- 使用dllplugin，一般不会变动的代码打包一个dll.js引用。
- 开启多线程，thread-loader
- 自动刷新与模块热替换
- 区分环境
- 压缩代码
- CDN加速
- 使用Tree Shaking
- 按需加载
- 开启Scope Hoisting

对设计模式的理解

对前端工程的理解

前端工程化就是为了让前端开发能够“自成体系”，个人认为主要应该从**模块化**、**组件化**、**规范化**、**自动化**四个方面思考。

https://www.jianshu.com/p/88ed70476adb

https://www.jianshu.com/p/0c0ea944a60a

https://www.cnblogs.com/onebox/p/9570518.html

详细见