## JavaScript

### 算法总结

#### 深度优先和广度优先

**深度优先**：

深度优先遍历（Depth-First-Search），是搜索算法的一种，它沿着树的深度遍历树的节点，尽可能深地搜索树的分支。当节点 v 的所有边都已被探寻过，将回溯到发现节点 v 的那条边的起始节点。这一过程一直进行到已探寻源节点到其他所有节点为止，如果还有未被发现的节点，则选择其中一个未被发现的节点为源节点并重复以上操作，直到所有节点都被探寻完成。

简单的说，DFS 就是从图中的一个节点开始追溯，直到最后一个节点，然后回溯，继续追溯下一条路径，直到到达所有的节点，如此往复，直到没有路径为止。

DFS 可以产生相应图的拓扑排序表，利用拓扑排序表可以解决很多问题，例如最大路径问题。一般用堆数据结构来辅助实现 DFS 算法。

**注意：深度 DFS 属于盲目搜索，无法保证搜索到的路径为最短路径，也不是在搜索特定的路径，而是通过搜索来查看图中有哪些路径可以选择**

**广度优先：**

广度优先遍历（BFS）

广度优先遍历（Breadth-First-Search）是从根节点开始，沿着图的宽度遍历节点，如果所有节点均被访问过，则算法终止，BFS 同样属于盲目搜索，一般用队列数据结构来辅助实现 BFS。

**BFS 从一个节点开始，尝试访问尽可能靠近它的目标节点。本质上这种遍历在图上是逐层移动的，首先检查最靠近第一个节点的层，再逐渐向下移动到离起始节点最远的层。**

步骤：

创建一个队列，并将开始节点放入队列中；

若队列非空，则从队列中取出第一个节点，并检测它是否为目标节点；

若是目标节点，则结束搜寻，并返回结果；  
若不是，则将它所有没有被检测过的字节点都加入队列中；  
若队列为空，表示图中并没有目标节点，则结束遍历。

### 前端优化

#### 前端SEO优化（移动端）

1. 合理的title、description、keywords：

　　title值强调重点即可，重要关键词出现不要超过2次，而且要靠前，不同页面title要有所不同；

　　description把页面内容高度概括，长度合适，不可过分堆砌关键词，不同页面description有所不同；

　　keywords列举出重要关键词即可

　　2. 语义化的HTML代码，符合W3C规范：语义化代码让搜索引擎容易理解网页

　　3. 重要内容HTML代码放在最前：搜索引擎抓取HTML顺序是从上到下，有的

　　搜索引擎对抓取长度有限制，保证重要内容一定会被抓取

　　4. 重要内容不要用js输出：爬虫不会执行js获取内容

　　5. 少用iframe：搜索引擎不会抓取iframe中的内容

　　6. 非装饰性图片必须加alt

7. 提高网站速度：网站速度是搜索引擎排序的一个重要指标

#### 前端图像优化

1、不用图片，尽量用css3代替。 比如说要实现修饰效果，如半透明、边框、圆角、阴影、渐变等等

　　2、使用矢量图SVG替代位图

　　3、使用字体图标webfont、CSS Sprites等

　　4、用CSS或JavaScript实现预加载

　　5、按照HTTP协议设置合理的缓存

6、WebP图片格式能给前端带来的优化。WebP支持无损、有损压缩，动态、静态图片，压缩比率优于GIF、JPEG、JPEG2000、PG等格式

#### 前端优化

content方向：

　　1. 减少HTTP请求：合并文件、CSS精灵、inline Image

　　2. 减少DNS查询：DNS查询完成之前浏览器不能从这个主机下载任何文件。方法：DNS缓存、将资源分布到恰当数量的主机名，平衡并行下载和DNS查询

　　3. 避免重定向：多余的中间访问

　　4. 使Ajax可缓存

　　5. 非必须组件延迟加载

　　6. 未来所需组件预加载

　　7. 减少DOM元素数量

　　8. 将资源放到不同的域下：浏览器同时从一个域下载资源的数目有限，增加域可以提高并行下载量

　　9. 减少iframe数量

　　10. 减少404页面

　　Server方面

　　1. 使用CDN

　　2. 添加Expires或者Cache-Control响应头

　　3. 对组件使用Gzip压缩

　　4. 配置ETag

　　5. Flush Buffer Early

　　6. Ajax使用GET进行请求

　　7. 避免空src的img标签

　　Cookie方面

　　1. 减小cookie大小

　　2. 引入资源的域名不要包含cookie

　　css方面

　　1. 将样式表放到页面顶部

　　2. 不使用CSS表达式

　　3. 使用不使用@import

　　4. 不使用IE的Filter

　　Javascript方面

　　1. 将脚本放到页面底部

　　2. 将javascript和css从外部引入

　　3. 压缩javascript和css

　　4. 删除不需要的脚本

　　5. 减少DOM访问

　　6. 合理设计事件监听器

　　图片方面

　　1. 优化图片：根据实际颜色需要选择色深、压缩

　　2. 优化css精灵

　　3. 不要在HTML中拉伸图片

4. 保证favicon.ico小并且可缓存

### 函数

#### 自调用函数

**定义：**声明一个函数，并马上调用这个匿名函数就叫做立即执行函数

**自调用函数（立即执行函数）写法：**

(function() {alert('匿名函数')}()) //用括号将整个表达式包起来

(function() {alert('匿名函数')})() //用括号将函数包起来

!function() {alert('匿名函数')}()

+function() {alert('匿名函数')}()

-function() {alert('匿名函数')}()

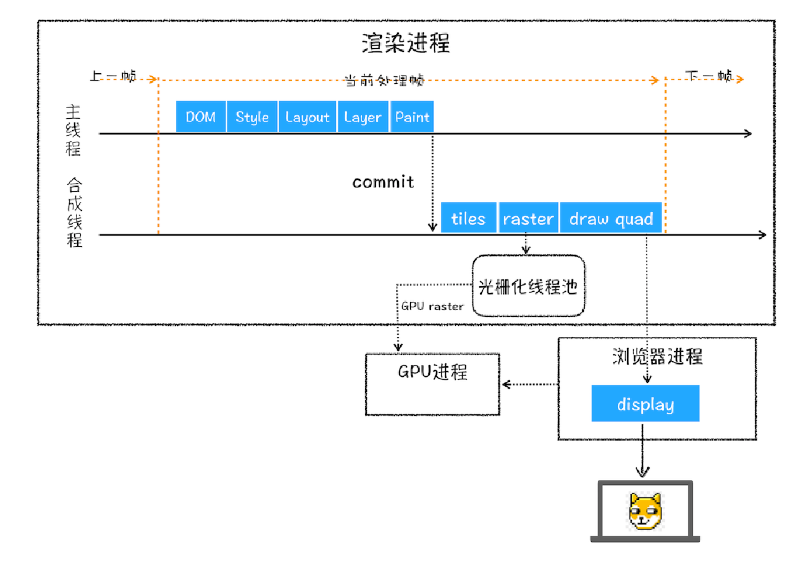
~function() {alert('匿名函数')}()

void function() {alert('匿名函数')}()

new function() {alert('匿名函数')}()

**自调用函数（立即执行函数）作用：**

### 浏览器渲染机制



上图总结：

1. 渲染进程将HTML内容转换为能够读懂的DOM树结构。

2. 渲染引擎将CSS样式表转化为浏览器可以理解的styleSheets，计算出DOM节点的样式。

3. 创建布局树，并计算元素的布局信息。

4. 对布局树进行分层，并生成分层树。

5. 为每个图层生成绘制列表，并将其提交到合成线程。

6. 合成线程将图层分成图块，并在光栅化线程池中将图块转换成位图。

7. 合成线程发送绘制图块命令DrawQuad给浏览器进程。

8. 浏览器进程根据DrawQuad消息生成页面，并显示到显示器上。

整个渲染流程： HTML 解析、样式计算、布局、分层、绘制、分块、光栅化、画

每个阶段都有明确的输入输出，上一个阶段的输出会成为下一个阶段的输入。

这样，整个渲染流程就形成了一套组织严密的生产流水线。

#### HTML 解析

解析过程中遇到 CSS 解析 CSS，遇到 JS 执行 JS。为了提高解析效率，浏览器在开始解析前，会启动一个预解析的线程，率先下载 HTML 中的外部 CSS 文件和 外部的 JS 文件。

如果主线程解析到`link`位置，此时外部的 CSS 文件还没有下载解析好，主线程不会等待，继续解析后续的 HTML。这是因为下载和解析 CSS 的工作是在预解析线程中进行的。这就是 CSS 不会阻塞 HTML 解析的根本原因。

如果主线程解析到`script`位置，会停止解析 HTML，转而等待 JS 文件下载好，并将全局代码解析执行完成后，才能继续解析 HTML。这是因为 JS 代码的执行过程可能会修改当前的 DOM 树，所以 DOM 树的生成必须暂停。这就是 JS 会阻塞 HTML 解析的根本原因。

第一步完成后，会得到 DOM 树和 CSSOM 树，浏览器的默认样式、内部样式、外部样式、行内样式均会包含在 CSSOM 树中。

#### 样式计算

主线程会遍历得到的 DOM 树，依次为树中的每个节点计算出它最终的样式，称之为 Computed Style。

在这一过程中，很多预设值会变成绝对值，比如`red`会变成`rgb(255,0,0)`；相对单位会变成绝对单位，比如`em`会变成`px`，这一步完成后，会得到一棵带有样式的 DOM 树。

#### 布局

布局阶段会依次遍历 DOM 树的每一个节点，计算每个节点的几何信息。例如节点的宽高、相对包含块的位置。

大部分时候，DOM 树和布局树并非一一对应。

比如`display:none`的节点没有几何信息，因此不会生成到布局树；又比如使用了伪元素选择器，虽然 DOM 树中不存在这些伪元素节点，但它们拥有几何信息，所以会生成到布局树中。还有匿名行盒、匿名块盒等等都会导致 DOM 树和布局树无法一一对应。

#### 分层

主线程会使用一套复杂的策略对整个布局树中进行分层。

分层的好处在于，将来某一个层改变后，仅会对该层进行后续处理，从而提升效率。

滚动条、堆叠上下文、transform、opacity 等样式都会或多或少的影响分层结果，也可以通过`will-change`属性更大程度的影响分层结果。

#### 绘制

主线程会为每个层单独产生绘制指令集，用于描述这一层的内容该如何画出来。

完成绘制后，主线程将每个图层的绘制信息提交给合成线程，剩余工作将由合成线程完成。

合成线程首先对每个图层进行分块，将其划分为更多的小区域。

它会从线程池中拿取多个线程来完成分块工作。

#### 光栅化

合成线程会将块信息交给 GPU 进程，以极高的速度完成光栅化。

GPU 进程会开启多个线程来完成光栅化，并且优先处理靠近视口区域的块。

光栅化的结果，就是一块一块的位图

#### 画（开始绘制）

合成线程拿到每个层、每个块的位图后，生成一个个「指引（quad）」信息。

指引会标识出每个位图应该画到屏幕的哪个位置，以及会考虑到旋转、缩放等变形。

变形发生在合成线程，与渲染主线程无关，这就是`transform`效率高的本质原因。

合成线程会把 quad 提交给 GPU 进程，由 GPU 进程产生系统调用，提交给 GPU 硬件，完成最终的屏幕成像。

### 事件

#### 事件流

概念： 描述的是从文档页面接受事件的顺序

分类：

冒泡事件流： 即事件最开始由具体的元素（文档中嵌套层次最深的那个节点）接收，然后逐级向上传播至最不具体的那个节点（文档）

捕获事件流：不太具体的节点应该更早的接收到事件，而最具体的节点最后接收到事件

#### 事件处理程序

1. **HTML事件处理程序**
2. **DOM 0级事件处理程序**

较为传统的方式：把一个函数值赋值给一个事件处理程序的属性。这种方法用的比较多，主要是因为它简单而且支持跨浏览器

1. **DOM 2级事件处理程序**

 DOM2级的事件定义了两个方法：用于处理指定和删除事件处理程序的操作。分别是addEventListener()和removeEventListener()两个方法，接收三个参 数:要处理的事件名（无‘on’），作为事件处理程序的函数，布尔值（冒泡还是捕获）

   DOM0、DOM2级都可以给一个元素添加多个事件和多个事件处理程序，其中多个事件处理程序按照添加顺序执行。

1. **IE事件处理程序**

分别是添加事件attachEvent()和删除事件detachEvent()，接收两个参数：事件处理程序的名称（有‘on’）和事件处理程序的函数；

支持IE事件处理程序的浏览器有：IE和Opera

不使用第三个参数的原因是：IE8及更早的浏览器版本只支持事件冒泡

1. **跨浏览器事件处理程序**

#### 事件对象

 在触发DOM上的某个事件时，都会产生一个事件对象event。只有在事件处理程序执行期间，event对象才会存在；一旦事件处理程序执行完成，event对象就会被销毁。

**（1）.DOM中的事件对象**（兼容DOM的浏览器会将一个event对象传入到事件处理程序中）

          ①、type属性：用于获取事件类型；

          ②、target属性：用于获取事件目标；

          ③、stopPropagation（）方法：用于阻止事件冒泡；

          ④、preventDefault（）方法：阻止事件的默认行为；

**（2）.IE中的事件对象**

          ①、type属性：用于获取事件类型；

          ②、srcElement属性：用于获取事件的目标；

          ③、cancelBubble属性：用于阻止事件冒泡；

          ④、returnValue属性：用于阻止事件的默认行为；

### 面向对象

**面向对象要素： 封装、继承、多态、抽象**

### 跨域

#### 含义：

现代浏览器出于安全考虑，都会去遵守一个叫做“同源策略”(**同源策略**就是用来限制从一个源加载的文档或脚本与来自另一个源的资源进行交互)的约定，同源的意思是两个地址的协议、域名、端口号都相同的情况下，才叫同源。这个时候两个地址才可以相互访问 cookie、localStorage、sessionStorage；发送 ajax 请求，如果三者有一个不同，就是不同源，这时再去访问这些资源就叫做跨域。

注意：

1. 端口和协议的不同，只能通过后台来解决

2、localhost和127.0.0.1虽然都指向本机，但也属于跨域，但请求不会携带 cookie时，也就没有跨域限制，如下：

* js、css、image 等静态文件
* form 表单提交

同源策略限制的对象：

Cookie、LocalStorage 和 IndexDB 无法读取

DOM 和 JS 对象无法获取

#### 如何解决跨域

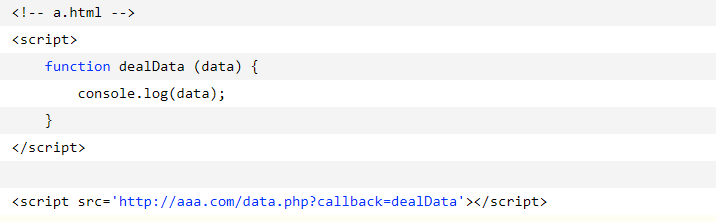
##### JSONP方法;

Jsonp(JSON with Padding) 是 json 的一种"使用模式"，可以让网页从别的域名（网站）那获取资料，即跨域读取数据。

JSONP的原理其实就是利用引入script不限制源的特点，把处理函数名作为参数传入，然后返回执行语句

**注意： 只支持get 请求**

假设这个json数据地址是[http://aaa.com/data.php](http://example.com/data.php),那么a.html中的代码就可以这样：



##### window.name方法:

关键点：window.name在页面的生命周期里共享一个window.name;

优点：

最简单的利用了浏览器的特性来做到不同域之间的数据传递；  
　不需要前端和后端的特殊配制；

缺点：  
　大小限制：window.name最大size是2M左右，不同浏览器中会有不同约定；  
　安全性：当前页面所有window都可以修改，很不安全；  
　数据类型：传递数据只能限于字符串。

##### document.domain方法;

设想场景：有一个页面 http://www.example.com/a.html ，它里面有一个iframe，这个iframe的源是 http://example.com/b.html ，很显然它们是不同源的，所以我们无法在父页面中操控子页面的内容。

解决方案：



##### window.postMessage方法；

window.postMessage(message,targetOrigin)  方法是html5新引进的特性，可以使用它来向其它的window对象发送消息，无论这个window对象是属于同源或不同源;

需要接收消息的window对象，可是通过监听自身的message事件来获取传过来的消息，消息内容储存在该事件对象的data属性中。



##### Cors

cors 全称跨资源共享 主要是后端工程师在€请求的页面配置好并且支持 所有的请求方式

* **cors的原理 定义一种跨域访问的机制让Ajax实现跨域访问**
* **cors允许一个域上的网络应用另外一个域提交Ajax请求**
* **实现这个功能只需要给服务器发送一个请求头**

###### 简单请求和复杂请求：

浏览器将cors请求分为简单请求和复杂请求。  
简单请求则直接发送请求到服务器，只请求一次。  
而复杂请求在正式请求前都会有预检请求，在浏览器中都能看到有OPTIONS请求，用于向服务器请求权限信息的，需要请求两次。

###### 简单请求必须要同时满足下面三个条件：

1. 请求方式只能是：GET、POST、HEAD
2. HTTP请求头限制这几种字段：Accept、Accept-Language、Content-Language、Content-Type、Last-Event-ID
3. Content-type只能取：application/x-www-form-urlencoded、multipart/form-data、text/plain

###### 复杂请求

不满足简单请求的条件，那么就是复杂请求。  
复杂请求会在正式请求发送之前，先发一个预检请求进行校验，校验通过后才能进行正式请求。  
举个  
浏览器现在要发送一个put的复杂请求，那么在put请求发送之前，浏览器先发送一个options请求。

Options请求缓存

在options请求中，我们可以通过设置响应头的参数Access-Control-Max-Age来对结果进行缓存  
比如： Access-Control-Max-Age: 600 表示对options检验结果进行十分钟的缓存

1. url变化会导致缓存失效,需要重新验证options请求的返回值
2. 预检不关心post data
3. header变化，如果是去掉了自定义的header使得请求变成简单请求，不会发送options请求。如果是增加其他的header，是会重新验证Access-Control-Allow-Headers的值。
4. cookie变化，只要后端允许发送cookie，cookie值变化不会导致缓存失效。

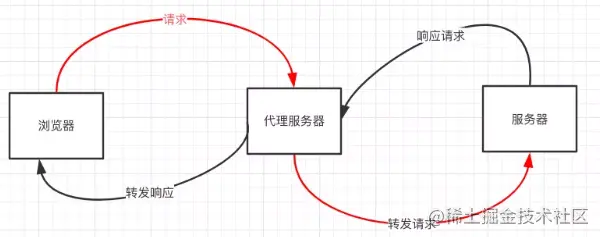
##### 代理：

通过前端代理 在[vue](https://so.csdn.net/so/search?q=vue" \t "_blank)中通过脚手架中的config 中 的index文件来配置的  
其中proxytable来配置跨域

##### Node中间件代理（两次跨域）

实现原理：**同源策略是浏览器需要遵循的标准，而如果是服务器向服务器请求就无需遵循同源策略。** 代理服务器，需要做以下几个步骤：

* 接受客户端请求 。
* 将请求 转发给服务器。
* 拿到服务器 响应 数据。
* 将 响应 转发给客户端。



##### Nginx 反向代理

1. 最简单的解决跨域的方式
2. 需要搭建一个中转Nginx 服务器
3. 实现思路：通过nginx配置一个代理服务器（域名与domain1相同，端口不同）做跳板机，反向代理访问domain2接口，并且可以顺便修改cookie中domain信息，方便当前域cookie写入，实现跨域登录

##### 总结：

 CORS支持所有类型的HTTP请求，是跨域HTTP请求的根本解决方案

 JSONP只支持GET请求，JSONP的优势在于支持老式浏览器，以及可以向不支持CORS的网站请求数据。

 不管是Node中间件代理还是nginx反向代理，主要是通过同源策略对服务器不加限制。

 日常工作中，用得比较多的跨域方案是cors和nginx反向代理

##### localStorage.hash

**实现原理**： a.html欲与c.html跨域相互通信，通过中间页b.html来实现。 三个页面，不同域之间利用iframe的location.hash传值，相同域之间直接js访问来通信。

**具体实现步骤**：一开始a.html给c.html传一个hash值，然后c.html收到hash值后，再把hash值传递给b.html，最后b.html将结果放到a.html的hash值中。

### 闭包

#### 函数柯理化

柯里化（Currying）是把接受多个参数的函数变换

成接受一个单一参数（最初函数的第一个参数）的函数，

并且返回接受余下的参数且返回结果的新函数的技术。

函数柯理化的主要作用和特点就是**参数复用**，**提前返回**和**延迟执行**

#### 闭包的含义

函数之间存在嵌套关系，内部函数使用到了外部函数中的变量，内部函数在执行函数定义的时候，就形成了闭包

**注意**：闭包本质上是一个对象，包含引用的变量

闭包是指那些能够访问自由变量的函数。

自由变量是指在函数中使用的，但既不是函数参数也不是函数的局部变量的变量。

闭包 = 函数 + 函数能够访问的自由变量

ECMAScript中，闭包指的是：

1. 从理论角度：所有的函数。因为它们都在创建的时候就将上层上下文的数据保存起来了。哪怕是简单的全局变量也是如此，因为函数中访问全局变量就相当于是在访问自由变量，这个时候使用最外层的作用域。
2. 从实践角度：以下函数才算是闭包：
   1. 即使创建它的上下文已经销毁，它仍然存在（比如，内部函数从父函数中返回）
   2. 在代码中引用了自由变量



#### 闭包形成条件：

1. 函数嵌套
2. 外部函数体执行
3. 内部函数定义执行（或者内部函数对象创建）
4. 内部函数用到外部函数的变量

#### 闭包的优点缺点

1. 延长外部函数变量对象的声明周期

2、不及时清理会造成内存溢出、泄漏

#### 内存泄漏的解决办法

让内部函数对象成为垃圾对象，断开指向它的所有引用

#### 内存泄漏了解

1. 内存泄漏：是指程序在申请内存之后，无法释放已申请的内存空间就造成了内存泄漏，一次内存泄漏似乎不会有大的影响，但是内存泄漏堆积会造成内存溢出
2. 内存溢出：指程序申请内存是，没有足够的内存供申请者使用

### 浏览器输入url之后

1. URL解析
2. DNS解析域名和ip
3. 发送请求，TCP连接三次握手
4. 服务器处理请求，（如果服务器配置了重定向，返回一个301永久重定向响应，然后重新执行上面的步骤）
5. 关闭TCP链接
6. 浏览器接受响应，处理数据资源
7. 渲染页面（
   * HTML解析
   * Css 解析
   * 渲染dom树
   * 布局与重绘
   * 合并渲染层
   * 回流与重绘
   * JavaScript 编译执行）

#### 回流（重排）

回流一定会重绘，重绘不一定会回流

浏览器流式布局，DOM节点树形结构，一个节点改变，整个DOM结构重新计算，引发回流

触发条件：

* 首次渲染
* 改变DOM节点
* style样式的变化，元素位置，元素大小
* 页面大小，resize()事件
* 激活一些伪类(:hover)

#### 重绘：

元素改变不影响其位置（颜色、字体）

#### 减少重绘

* DOM的操作不放在循环中
* 将style样式改成class，改变class即可，只会重绘一次
* 动画HTML的position尽量设成fixd或absolute
* 不使用table

### 垃圾回收机制

1. 标记清除
2. 引用计数

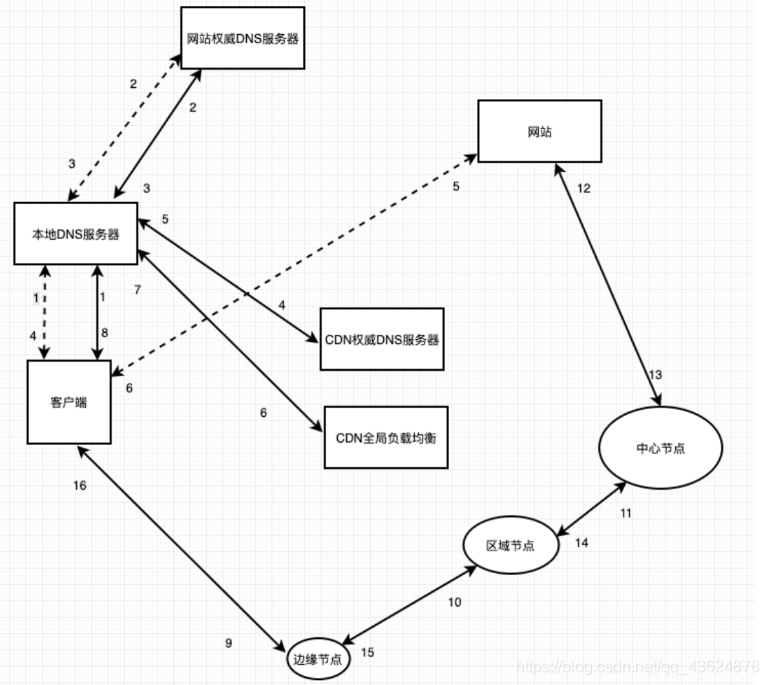
### 虚拟dom和真是dom区别

1. 虚拟dom不会进行排版与重绘 ，虚拟dom就是把真实的dom转化为JavaScript代码
2. 虚拟DOM 进行频繁的修改，然后一次性比较修改真实DOM中需要修改的部分，最后在真实DOM中进行排版与重绘，减少DOM节点排版与重绘的消耗
3. 真实DOM排版与重绘效率相对低
4. 虚拟DOM有效降低大面积DOM（真实）的排版与重绘，通过对比虚拟真实DOM部分渲染真实DOM

### 详说缓存

#### CDN缓存

**概念**： CDN通过在多个节点部署来减少请求时间，这样不需要每次都回到源站服务器进行请求。 CDN对于常见的HTTP请求都支持，但只对GET请求进行缓存。对于其它请求不做缓存，仅起到转发作用，相当于proxy）



**原理**：

Content Delivery Network，内容分发网络。基本原理是避开互联网上有可能影响数据传输速度和稳定性的瓶颈和环节，使内容能够传输的更快、更稳定。

**应用**：

1、使用第三方CDN服务：针对公司没有条件购买自己的CDN服务，如：https://www.bootcdn.cn/

2、CDN进行静态资源缓存：比如JavaScript、css、图片，甚至是将脚本和webpack结合直接放到CDN，按成一键自动部署整个项目。

3、直播传送：直播本质上是使用了流媒体进行传送，而cdn是支持流媒体传送的。so，直播完全可以用cdn来提高访问速度（回源的方式）

#### 数据库缓存

数据库缓存：将一些经常被访问到的资源直接放在内存中，当数据发生变化的时候才会操作数据库。

### 浏览器缓存

#### Cookie

1. 关闭浏览器cookie数据清空
2. 储存在浏览器
3. 安全性低
4. 保存数据一般不超过4K
5. 保存的是字符串
6. 设置路径、作用域，不可以跨域使用

弊端：

Cookie数量和长度的限制。每个domain最多只能有20条cookie，每个cookie长度不能超过4KB

安全性问题。如果cookie被人拦截了，那人就可以取得所有的session信息。

cookie在浏览器和服务器间**来回传递， 占用带宽**

**Cookie 容易被误删**

**Cookies欺骗，修改host 文件，可以非法访问目标站点的cookie**

##### 域名共享cookie

1，共享之间的二级域名必须是解析于同一个顶级域名之下；

2，比如现在有两个二级域名，a.xxx.com(域名A)和b.xxx.com(域名B)。两个都解析于域名xxx.com顶级域名之下。

3，现在域名A的登录cookie域名B下面需要使用。

4，域名A的登录cookie把cookie域设置成xxx.com（注意不要写成www.xxx.com，因为www.xxx.com也属于二级域名，不是顶级域名），path设置成’/’

5，此时域名B下面可以读取域名A写入的cookie

6，这样就可以实现域名A的cookie被域名B共享

7，如果要实现相互共享，就把两个域名的cookie域都设置成xxx.com

**httpONLY属性：**如果cookie中设置了HttpOnly属性，那么通过js脚本将无法读取到cookie信息，这样能有效的防止XSS攻击

#### IndexedDB

* 1. 浏览器提供的本地数据库
  2. 允许储存大量数据
  3. 提供查询接口，支持查询
  4. 能建立索引
  5. 不属于关系型数据库（不支持SQL语句），更接近NoSQL数据库
  6. 内部采用对象仓库，所有类型的数据都可以直接存入
  7. 支持异步操作（localStorage 是同步操作）
  8. IndexedDB 支持事务（transaction），这意味着一系列操作步骤之中，只要有一步失败，整个事务就都取消，数据库回滚到事务发生之前的状态，不存在只改写一部分数据的情况。
  9. 不能跨域
  10. 存储空间大250M左右（甚至没有上线）
  11. 支持二进制存储；；；IndexedDB 不仅可以储存字符串，还可以储存二进制数据（ArrayBuffer 对象和 Blob 对象）。

#### Session（基于cookie）

1. 存储在服务器内存中
2. 访问过多时，占用服务器性能
3. 保存类型是对象
4. 不区分路径，同一个用户访问网站期间，所有的session在任何地方都可以访问到
5. cookie 的生命周期是累计的
6. session 的生命周期是间隔的
7. cookie 是多个浏览器共享，session 是一个浏览器独享

#### Web Storage

1. 存储在浏览器中
2. 关闭窗口数据丢弃
3. 不会把数据主动发送给服务器
4. 大小5M或者更大
5. 减少网络流量：存储在本地、不需要向浏览器做不必要的请求

##### localStorage

* 1. 需要手动删除，否则长期有效
  2. 浏览器支持
  3. 只支持string 类型的储存，JSON对象需要转化
  4. 本质上是对字符串的读取，如果储存内容多的话会消耗内存空间，导致页面卡顿

##### sessionStorage

1. 引入浏览器窗口的概念
2. sessionStorage是在同源的同窗口中，始终存在的数据，也就是说只要这个浏览器窗口没有关闭，即使刷新页面或进入同源另一个页面，数据仍然存在，关闭窗口后，sessionStorage就会被销毁，同时“独立”打开的不同窗口，即使是同一页面，sessionStorage对象也是不同的

#### cookie、sessionStorage、localStorage区别

1. sessionStorage不在不同的浏览器窗口中共享，即使是同一个页面；localstorage在所有同源窗口中都是共享的；cookie也是在所有同源窗口中都是共享的
2. Storage支持事件通知机制，可以将数据更新的通知发送给监听者
3. cookie数据始终在同源的http请求中携带（即使不需要），即cookie在浏览器和服务器间来回传递，而sessionStorage和localStorage不会自动把数据发送给服务器，仅在本地保存。cookie数据还有路径（path）的概念，可以限制cookie只属于某个路径下
4. 存储大小限制也不同，cookie数据不能超过4K，同时因为每次http请求都会携带cookie、所以cookie只适合保存很小的数据，如会话标识。sessionStorage和localStorage虽然也有存储大小的限制，但比cookie大得多，可以达到5M或更大
5. 数据有效期不同，sessionStorage：仅在当前浏览器窗口关闭之前有效；localStorage：始终有效，窗口或浏览器关闭也一直保存，因此用作持久数据；cookie：只在设置的cookie过期时间之前有效，即使窗口关闭或浏览器关闭

#### RefleshToken

Refresh\_token的作用是刷新AccessToken。认证服务器会提供一个刷新接口，我们传入Refresh\_token和client\_id，认证服务器通过后会返回一个新的AccessToken

##### 相关安全措施

1，要求refresh\_token必须保存在客户端的服务器上，调用refresh\_token的时候一定是从服务器到服务器的访问。

2，OAuth2引入了Client\_Secret机制。每一个Client\_id,都对应一个Client\_Secret。这个Client\_Secret会在客户端申请Client\_id时，随Client\_id一起分配给客户端。客户端把他们都保存在服务器上，刷新Refresh\_token时，需要验证这个Client\_Secret。

### 深入了解JS

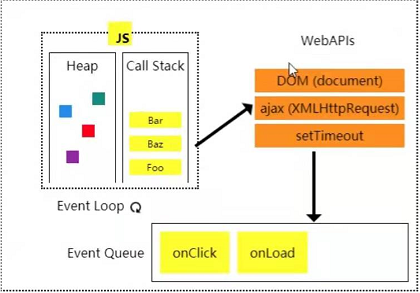
#### Js执行原理

* 1. JavaScript 是单线程编译解析代码（web worker除外），无论是同步代码还是异步代码
  2. 界面第一次渲染：初始化同步代码 ===》 所有微任务 ===》 渲染界面 ===》执行第一个宏任务 ===》宏任务中的微任务 ===》渲染界面 。。。
  3. 界面更新渲染：所有的微任务 ===》 渲染界面 ===》 执行宏任务 ===》宏任务中的微任务 ===》 渲染界面
  4. 浏览器在另一个线程（GUI渲染线程）进行页面渲染操作
  5. GUI 渲染线程与JS线程是互斥的（不会同时执行）因为JS代码可以修改DOM结构
  6. 遇到需要等待（网络请求、定时任务）不能卡顿，需要异步执行
  7. 回调callback 函数

#### 事件轮询机制

步骤：

首先执行初始化代码（同步代码），执行同步代码的时候可能会启动定时器、网络请求、绑定事件、执行这些代码的时候，会把回调函数交给对应的管理模块进行管理，而对应的管理模块会在分线程执行，不会影响JS代码的执行。管理模块会将异步数据处理分类，放进待执行的回调队列中；等到初始JS代码执行完毕会一个一个将异步事件取出并执行



### 宏任务、微任务

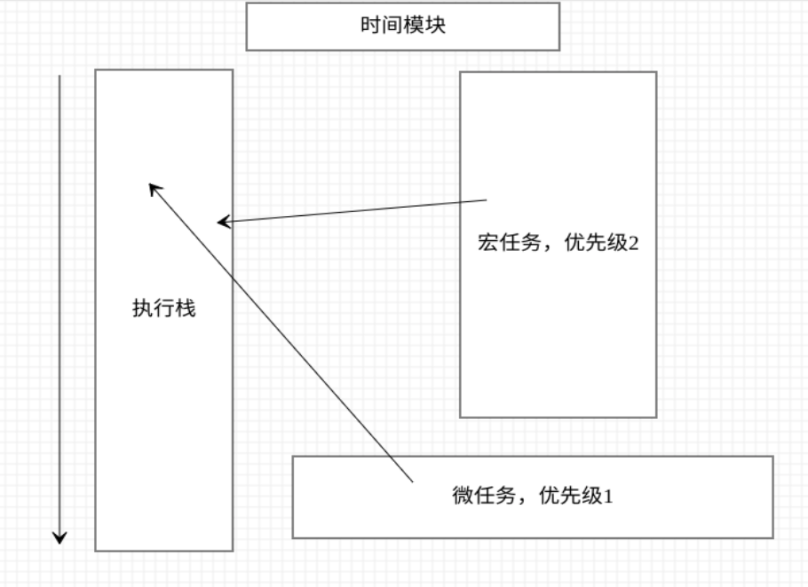
1. js是一门单线程语言 浏览器是多线程的
2. 同步进入主线程，主线程任务执行完毕之后 会去Event Queue读取相应的函数
3. 异步进入Event Table并注册函数，

**执行顺序：**

执行栈在执行完同步任务后，查看执行栈是否为空，如果执行栈为空，就会去检查微任务队列是否为空，如果为空的话，就执行宏任务，否则就一次性执行完所有微任务。

每次单个宏任务执行完毕后，检查微任务队列是否为空，如果不为空的话，会按照先入先出的规则全部执行完微任务后，设置微任务队列为null，然后再执行宏任务，如此循环。

总结：同步—>微任务—>宏任务



#### 宏任务

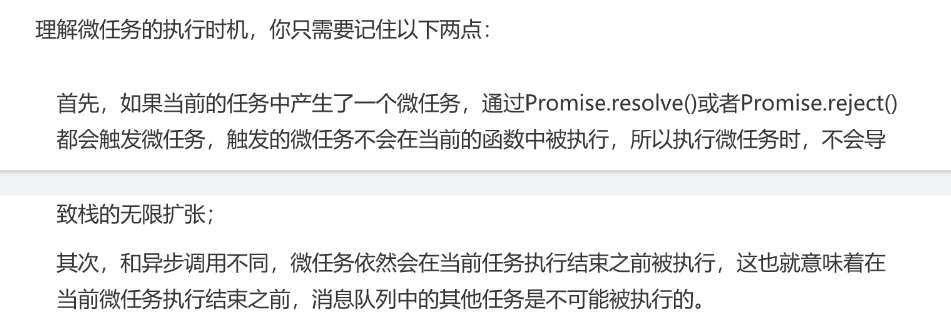
setInterval、setTimeOut、script、postmessage 、 setImmediate、 MessageChannel

事件绑定、ajax、回调函数

#### 微任务：

Promise.then（promise是同步的）、 async-await、 Object.observe、 MutaionObserver、 process.nextTick（node.js环境）

定义出一个动作，并且让这个动作在下一个事件轮询的时间点上执行)



### 异步操作、异步线程、轮询

#### 异步

如果函数是异步的，发出调用后马上返回，但是不会马上返回预期结果。

##### 异步调用

1. async-await
2. promise

##### promise、async-await、generater

###### promise

Promise,简单来说就是一个容器，里面保存着某个未来才会结束的时间(通常是一个异步操作的结果)

###### Async-await

将异步强行转换为同步处理、async/await是寄生于Promise，都是generater的语法糖

用法：

async用于申明一个function是异步的，而await可以认为是async wait的简写，等待一个异步方法执行完成

###### Generater

generator:可以将生成器视为可以暂停和恢复的进程（代码段),代码在执行的过程中可以主要交出控制权

genearator 语法: function\* 是一个新的关键字用于生成器函数（也有生成器方法）。

yield是generator可以自行暂停的运算符。此外，generator还可以通过yield接收输入和发送输出。

### 深、浅copy

#### 浅copy：

1. 对于基本类型来说浅拷贝就是赋值
2. 对于引用类型来说（obj,arr等），**浅复制是对对象地址的复制**，并没有开辟新的栈，也就是复制的结果是两个对象指向同一个地址，修改其中一个对象的属性，则另一个对象的属性也会改变

#### 深copy

**深复制则是开辟新的栈**，两个对象对应两个不同的地址，修改一个对象的属性，不会改变另一个对象的属性

### 原型链

#### 理解

**原型的产生**：

执行函数定义的时候，prototype 显示原型创建；

执行函数（实例化函数）的时候，\_\_proto\_\_ 隐式原型被创建

Prototype：浏览器标准属性，显示原型，存在于函数中

\_\_proto\_\_: 浏览器非标准属性，隐式原型，存在于实例对象中

实例的\_\_proto\_\_ 和其对应的prototype 都指向原型对象

无论是构造函数还是普通函数，或者是方法，只要是函数，内部就有prototype

**原型的作用**：

共享数据，节省内存空间

实现继承

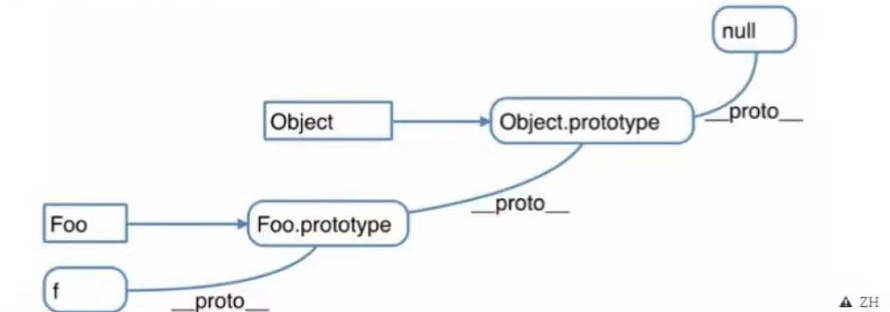
**扩展式理解**：

原型什么时候产生，原型的分类，原型的作用，继承的实现，面向对象的编程，面向过程，数据共享，数据缓存 -🡪 闭包 --🡪 预解析 --🡪 作用域 --🡪 执行上下文环境 --🡪 原型的应用 -🡪 vue中的时间总线 --🡪 vue中组件的通信方式。。。。

##### 原型链

概念：隐式原型和显示原型之间的关系（实际上是隐式原型链，言外之意就是和显示原型没有关系，显示原型产生实例的一瞬间起作用）

* 从对象的\_\_proto\_\_ 开始，连接所有与其相关对象，就是我们常说的原型链，也可以称为隐式原型链
* 查找对象属性：先在自身上查找，找不到就沿着原型链查找，如果还找不到返回undefined

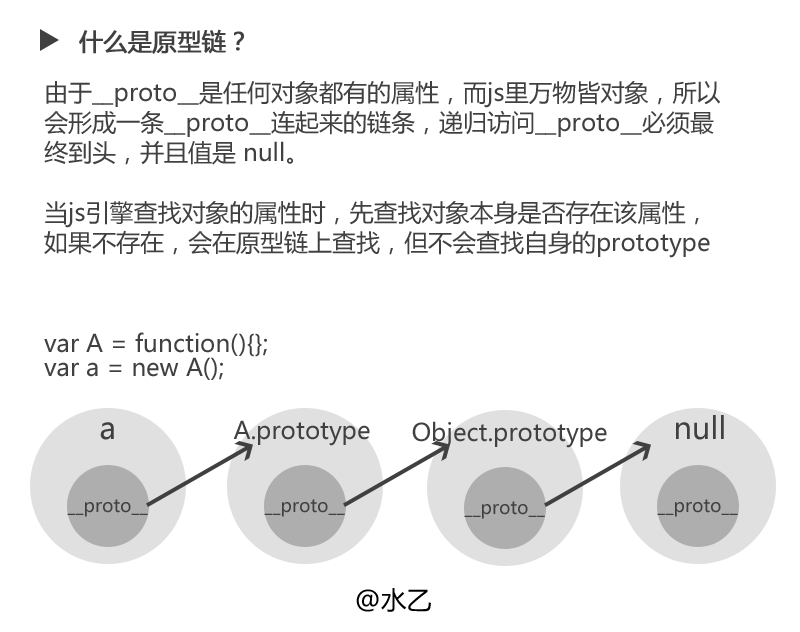


#### 概念

原型： 在JavaScript中原型是一个prototype对象，用于表示类型之间的关系（每一个javascript对象(除null外)创建的时候，就会与之关联另一个对象，这个对象就是我们所说的原型，每一个对象都会从原型中“继承”属性。）

原型链：

（原型链最终的指向是Object的prototype, 而Object中的\_\_proto\_\_是null）



### 执行上下文

#### 概念：

代表了代码执行的环境，包含： 执行环境，变量对象，this ，作用域链

#### 执行上下文流程：

当代码将要执行，在预解析之后，此时出现了全局执行上下文环境，创建一个变量对象用来收集，全局变量（var），全局函数（function），函数参数（arguments），确定this 指向，默认指向window，这个变量独享会被压入到栈中（全局执行上下文的变量对象在栈中的最下面），如果出现函数调用，此时出现局部执行上下文环境（局部执行上下文），再次创建一个变量对象，用来收集函数参数，变量，函数，改变this指向，这个变量会被再次压入栈中，在全局执行上下文的变量对象上面，如果当前函数执行完毕，此时出栈（把局部上下文对象干掉），一次弹出变量对象，就结束了；

#### 作用域链

当查找变量的时候，会先从当前上下文的变量对象中查找，如果没有找到，就会从父级(词法层面上的父级)执行上下文的变量对象中查找，一直找到全局上下文的变量对象，也就是全局对象。这样由多个执行上下文的变量对象构成的链表就叫做作用域链

#### 作用域和执行上下文

全局作用域之外，每个函数都会创建自己的作用域，作用域在函数定义时已经确定了，而不是函数调用时

全局执行上下文环境在全局作用域确定之后，js代码执行之前

函数执行上下文环境是在调用函数时，函数体代码执行之前创建

作用域是静态的，只要函数定义好了就一直存在，且不会再变化

执行上下文是动态的，调用函数是创建，函数调用结束时释放

上下文对象（环境）是从属于所在的作用域

全局上下文环境--🡪全局作用

作用域链：

用来决定代码执行的范围，变量的作用范围

作用域是代码定义的时候决定的

作用域链式如何产生的：

函数定义的时候自动添加一个属性’[[Scopes]]’ 该属性保存的是其上级作用域链；

当函数执行的时候，进入执行上下文环境，将创建对象添加到’[[Scopes]]’数组的第一个位置，形成新的数组

查找变量的规则：

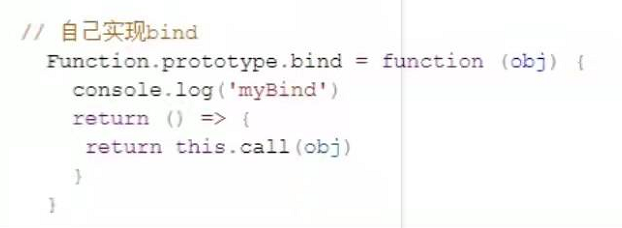
现在当前的作用域的变量对象中查找，找到就是用；没有找到则沿着作用域链的数组取上级作用域中的变量对象中查找，知道全局作用域的变量对象中没有对应的变量，返回报错

### bind，call，apply区别

#### 深入理解

##### Bind

简单实现：



##### Apply、bind、call

这些方法都是（继承自）Function 的prototype 中的（原型对象中的）实例方法



F1 函数可以直接调用apply、bind、 call ，那么首先是f1这个函数对象中查找，没有找到，沿着隐式原型链查找，找到了Function 中的prototype 原型对象，可以直接使用

如果想要重新定义apply、bind 、call ，无非就是使用Function.prototype.(apply/bind/call) function () {} 自己定义

#### 区别

三者都是用来改变函数的this指向的

1、都是用来改变函数的this对象的指向的。

2、第一个参数都是this要指向的对象。

3、都可以利用后续参数传参。

**call方法，可以指定该函数内部this的指向（即函数执行时所在的作用域），然后在所指定的作用域中，调用该函数。并且会立即执行该函数。**

**apply方法的作用与call方法类似，也是改变this指向（函数执行时所在的作用域），然后在指定的作用域中，调用该函数。同时也会立即执行该函数。唯一的区别就是，它接收一个数组作为函数执行时的参数。**

**bind方法用于指定函数内部的this指向（执行时所在的作用域），然后返回一个新函数。bind方法并非立即执行一个函数。**

#### 不同之处：

apply：最多只能有两个参数——新this对象和一个数组

call：则是直接的参数列表，主要用在js对象各方法互相调用的时候，使当前this实例指针保持一致,或在特殊情况下需要改变this指针。如果没有提供 thisObj 参数，那么 Global 对象被用作 thisObj。

Bind: 会返回一个改变this指向的新函数，其与之前的使用的不是同一块内存地址

bind 方法 与 apply 和 call 比较类似，也能改变函数体内的 this 指向。不同的是，**bind 方法的返回值是函数，并且需要稍后调用，才会执行**。而 apply 和 call 则是立即调用。

### 数据类型

#### 基本类型：

Number、 Boolean、 String、 null、 undefined 、 symbol（ES新增）

存放位置： 存放在栈内存

#### 引用类型

Object 、 Array、 Function

存放位置： 堆内存

#### 堆内存和栈内存

引用类型， 值的大小不固定，栈内存中存放地址指向堆内存

基本类型，值大小固定，存放在栈内存分配的内存空间中

基本类型在当前执行环境结束时销毁

引用类型不会随执行环境结束而销毁，只有当所有引用他的变量不存在了，这个对象才会被垃圾回收机制回收

#### 对象内存查找（哈希）

字典：被称作映射、符号表或者关联数组，例如：散列表

##### 散列表：如何检查单词是否存在

如果你用过一些文档编辑的软件，应该很常用的一个功能就是拼写检查，这个检查是怎么做

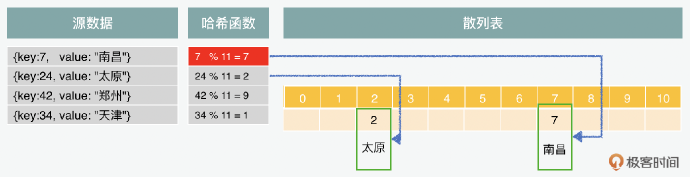
到的呢？从它的最底层逻辑来说，就是看一个单词存在与否。那么一个单词是否存在是如何

判断的呢？这里就需要用到**散列表**。**散列表**的实现逻辑就是基于每个单词都生成一个唯一的

**哈希值**，把这些值存放在一个数组中。当我们想查询一个词是否有效，就看这个词的**哈希值**

在数组中是否存在即可。

图例：



假设我们有上图中这样的一组城市的键值对组成的对象，我们可以看出，在哈希的过程中，

一个城市的键名，通过一个哈希函数，生成一个对应的唯一的哈希值，这个值被放到数组

中，形成一个哈希列表。下次，当我们想要访问其中数据的时候，就会通过对这个列表的遍

历来查询相关的值。

这里我们可以看到，图中间位置的是哈希函数，我们需要一个哈希函数来生成哈希值，那么

哈希值是怎么生成的呢？生成散列表中的哈希值有很多种方式，比如素数哈希、ASCII哈希，

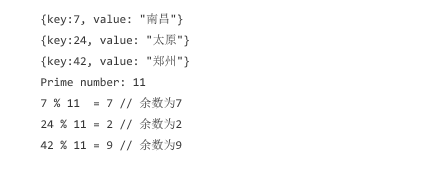
还有djb2等方式。

在哈希算法当中，最基础的就是素数（prime number）哈希。这里我们把一个素数作为模数

（modulus number），来给你举一个例子，在这个例子里，我们把11这个素数作为了模

数，用下面的一组键值对中的键除以模数，所获得的余数，放到一个数组中。就形成了一个

散列表。这样可以获得一个统一的索引。



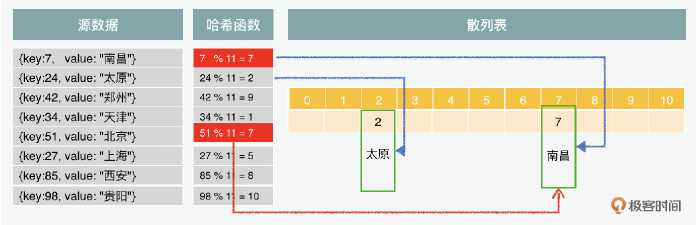
这个方式看似可以用来生成哈希值，但是也存在一个问题。在将余数放入数组的过程中，我

们会发现，如果处理的数据数量足够多，那么就会出现冲突的情况，比如下图中标红的两个

对象的键除以素数11的余数是相同的，7和51的余数都是7，这样就会造成冲突。一个完美的

哈希表是不应该存在冲突的，可是这样完美的哈希表其实在现实中并不存在，所以我们只能

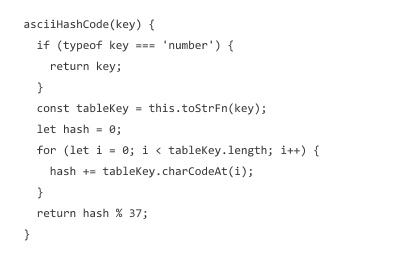
尽量减少这种情况。



为了尽量减少这种冲突，业界也在尝试其他办法，比如使用ASCII code和素数结合来生成哈

希，但这种方式和上面的素数哈希一样，**即使结合了ASCII**，哈希值也不能完全避免碰撞的产

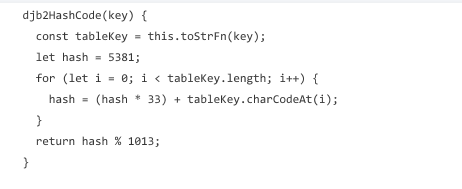
生，**只能减少冲突**。



djb2算法：

可以用来进一步减少这种问题的发生。它的做法是先用一个长质数5381作为哈希数，然后根据字符串长度循环，将哈希数乘以33，再加上字符的ASCII码迭代。结果和模数1013的余数结果就是最后的哈希值。

这里你可能会问，33和5381这两个数字是什么意思？这里乘以33呢，是因为更易于移位和加法计算。使用33可以复制累加器中的大多数输入位，然后将这些位分散开来。5的移位和32是互素的，这有助于雪崩。ASCII可以看做是2个4位字符类型选择器，比如说，数字的前四位都是0x3。所以2、4、8位都可能导致相似的位之间交互，而5位可以让一个字符中许多的4个低位与4个高位强烈交互。所以这就是选择33的原因。那么至于原则5381作为质数呢，则更多是一种习惯，也可以由其它大的质数代替。



##### 字典：如何查找对象的内存地址

定义：这种数据结构和我们平时用的字典类似，它和索引的主要作用都是快速查询和搜索。但是我们查字典的时候不光关心有没有这个词，更重要的事，我们知道这个单词对应的意思。所以我们需要通过一组键值对来表明它们的关系。

字典作为一种数据结构，又叫做映射（map）、符号表（symbol table）或者关联数组（associative array）。在JavaScript中，我们其实可以把对象看做是一种可以用来构建字典的一种散列表，因为对象里就包含key-value的属性。

结论：

对象在栈的引用和它在堆中的实际存储间的关联就是通过地址映射来实现的。这种映射关系就是通过字典来存储的。

##### 散列冲突：解决哈希碰撞的方式

###### 线性探查法

当一个散列碰撞发生时，程序会继续往下去找下一个空位置，比如在之前例子中，7被南昌占用了，北京就会顺移到8。

缺点：但是在查找的时候会有一定的问题，比如当我们想要查找某个数据的时候，

则需要在集群中迭代寻找。



###### 平方探查法

平方探测法用平方值来代替线性探查法中的往后顺移一位的方式，这样就可以做到基于有效的指数做更平均的分布。



###### 二度哈希法

二度哈希（Rehashing/Double-Hashing），也就是在第一次的哈希的基础上再次哈希。

公式：

在下面公式里，x是第一次哈希的结果，R小于哈希表。假设每次迭代序列号是i，每次哈希碰撞通过i \* hash2(x)来解决。

hash2(x) = R − (x % R)

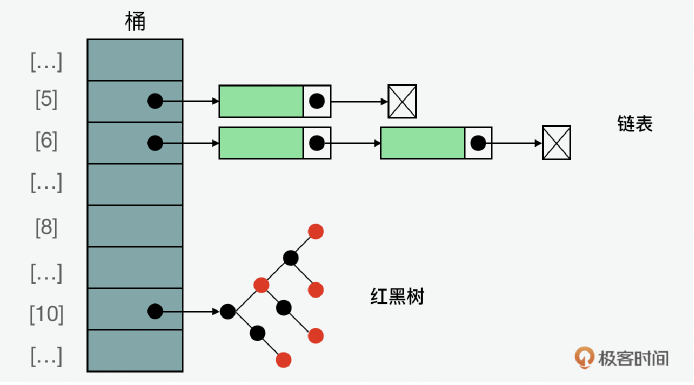


##### 拓展：Java 如何解决散列冲突

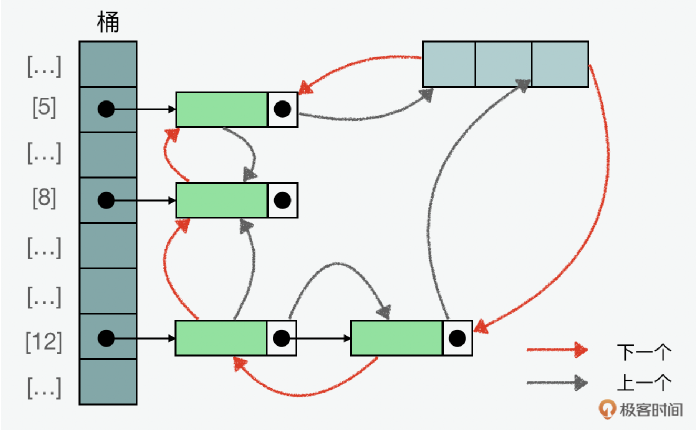
Java 解决散列冲突的集中方式：HashMap、LinkedHashMap和TreeMap。

HashMap的底层逻辑是通过链表和红黑树实现的。它最主要解决的问题就是哈希碰撞。

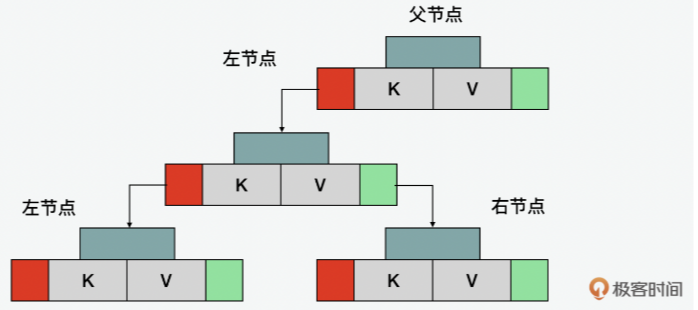
它的规则是，当哈希函数生成的哈希值有冲突的时候，就把有冲突的数据放到一个链表中，以此来解决哈希碰撞。那你可能会问，既然链表已经解决了这个问题，为什么还需要用到红黑树？这是因为当链表中元素的长度比较小的时候，链表性能还是可以的，但是当冲突的数据过多的时候，它就会产生性能上的问题，这个时候用增删改查的红黑树来代替会更合适。



LinkedHashMap是在HashMap的基础上，内部维持了一个双向链表（Doubly Linked List），它利用了双向链表的性能特点，可以起到另外一个非常重要的作用，就是可以保持插入的数据元素的顺序。



TreeMap是基于红黑树来实现的，TreeMap实现的是键值有序，它的查询效率不如HashMap和LinkedHashMap，但是相比前两者，它是线程安全的。



#### 数据类型判断

##### Typeof：

**主要的目的是检测一个变量是不是基本数据类型的变量**

同时也可以说是确定一个变量是字符串,数值,布尔值,还是undefined  
的最佳工具。

检测到函数时：返回function

检测到表达式时：在谷歌和Safari浏览器返回错误的返回function

其他的浏览器返回的是object.

##### Instanceof

 instanceof主要的目的是检测引用类型,判断对象是Array,还是RegExp!

**主要的实现原理**：只要右边变量的 prototype 在左边变量的原型链上即可

**其实typeof和instanceof的目的都是检测变量的类型,两个的区别在于typeof一般是检测的是基本数据类型,instanceof主要检测的是引用类型!**

##### Constructor

当一个函数F被定义时，JS引擎会为F添加prototype原型，然后在prototype上添加一个constructor属性，并让其指向F的引用，**F利用原型对象的constructor属性引用了自身**，当F作为构造函数创建对象时，原型上的constructor属性被遗传到了新创建的对象上，从原型链角度讲，构造函数F就是新对象的类型。这样做的意义是，让对象诞生以后，就具有可追溯的数据类型。

##### Object.prototype.toString.call()

toString()是Object的原型方法，调用该方法，默认返回当前对象的[[Class]]。这是一个内部属性，其格式为[object Xxx],其中Xxx就是对象的类型。

  对于Object对象，直接调用toString()就能返回[object Object],而对于其他对象，则需要通过call、apply来调用才能返回正确的类型信息。

### 继承

1、extends继承父类的this（ES6中引入了class关键字，class可以通过extends关键字实现继承，还可以通过static关键字定义类的静态方法,这比 ES5 的通过修改原型链实现继承）

2、通过原型链继承（子类的原型为父类的一个实例对象）

3、通过cal-apply继承（通过call 或者apply 调用父类的构造函数）

4、借用构造函数（类式继承）



通过在子函数里面使用call 或者apply

优点：避免应用类型的属性被所有实例共享； 可以向父级函数传参

5、组合继承：原型链+经典继承

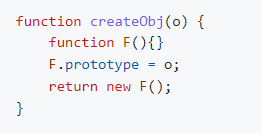


优点： 融合原型链继承和构造函数的优点、JavaScript 中最常用的继承模式

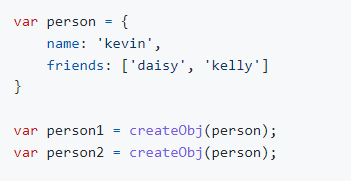
6、原型式继承：（ES5的Object.create的模拟实现，将传入的对象作为创建的对象的原型）

缺点：包含引用类型的属性值始终共享响应的值，和原型链继承一样

Create方法模拟：



使用：



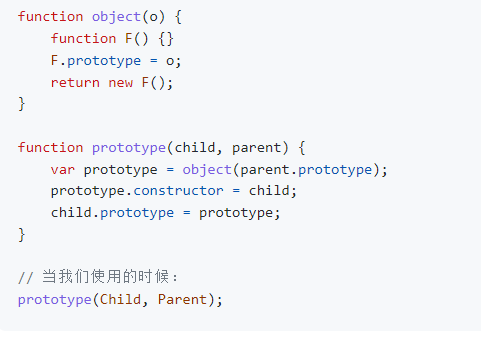
7、寄生式继承：（创建一个仅用于封装过程的函数，该函数在内部以某种形式做增强对象，最后返回）



8、寄生组合式继承



封装函数：



解决组合式继承的调用父级函数两次的缺点

（这种方式的高效率体现它只调用了一次 Parent 构造函数，并且因此避免了在 Parent.prototype 上面创建不必要的、多余的属性。与此同时，原型链还能保持不变；因此，还能够正常使用 instanceof 和 isPrototypeOf。开发人员普遍认为寄生组合式继承是引用类型最理想的继承范式。）

## HTML

### Script 标签

#### 注意

script标签不带defer和async属性：同步模式、会阻塞页面加载

浏览器都会按照<script>元素在页面中出现的先后顺序对他们依次进行解析

#### Defer属性：

defer属性：异步模式，只适合外部脚本文件，会被延迟到整个页面都解析完毕后再运行，脚本加载不阻塞页面的解析，同时带有defer的脚本彼此之间，能保证其执行顺序

#### Async属性

html5的新属性，只适合用于外部脚本文件，异步模式，通过createElement创建的script标签其属性async默认为true

#### Defer 和 async 区别

defer与async的区别是：defer要等到整个页面在内存中正常渲染结束（DOM 结构完全生成，以及其他脚本执行完成），才会执行；async一旦下载完，渲染引擎就会中断渲染，执行这个脚本以后，再继续渲染。一句话，defer是“渲染完再执行”，async是“下载完就执行”。另外，如果有多个defer脚本，会按照它们在页面出现的顺序加载，而多个async脚本是不能保证加载顺序的。

## CSS

### 动态样式绑定（css 使用变量、vars）

#### class动态绑定

1. 数组类型：

<div :class="['red', 'font16']">内容</div>

<div :class="['red', 'thin', isactive?'red':'yellow']">内容</div>

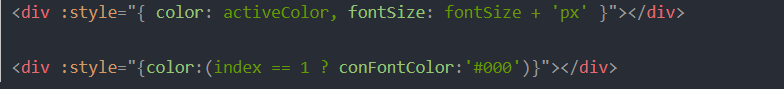
1. 对象类型

<div :class="{red:true, italic:true, active:true, thin:true}">内容</div>

<div :class="['red', 'thin', {'active': isactive}]">内容</div>

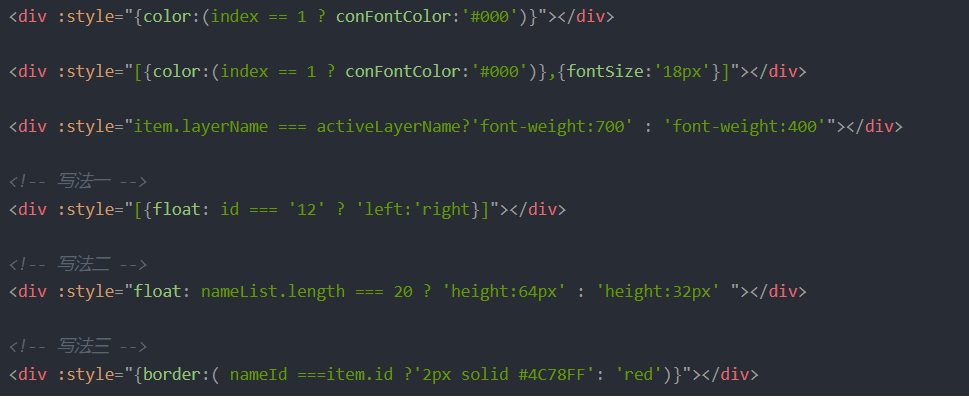
#### Style动态绑定

1、对象



2、数组

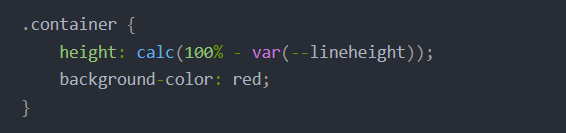




#### 变量结合css

Vue2 版本：（需要结合style）





注意：使用css var()，在ie浏览器中会有兼容问题，下面则是ie浏览器兼容处理（**css-vars-ponyfill**）

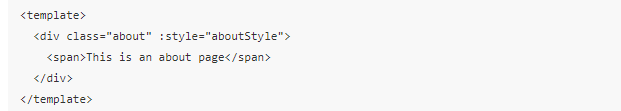
Vue3版本：

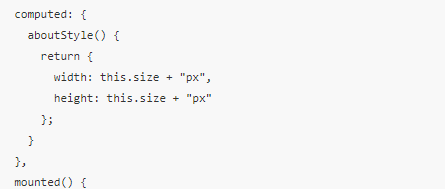




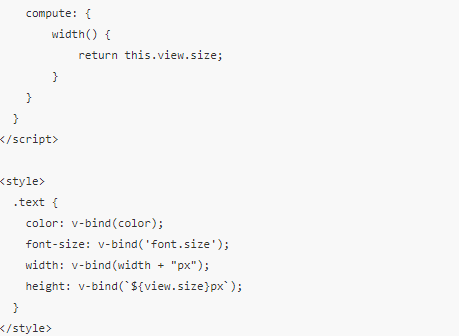
#### 计算属性与css 结合

Vue2：





Vue3：



### 长度单位换算

rem是CSS3新增的相对长度单位，是指相对于根元素html的font-size计算值的大小，可理解为**屏幕宽度的百分比**

em是相对于**父级字体大小**的，使用场景较少

rpx 规定屏幕宽为750rpx



### Sass/compass

### BFC布局

#### 理解：

BFC（Block Formatting Context）格式化上下文，是Web页面中盒模型布局的CSS渲染模式，指一个独立的渲染区域或者说是一个隔离的独立容器。

（让处于 BFC 内部的元素与外部的元素**相互隔离**，使内外元素的定位不会相互影响）

#### BFC触发条件

1、浮动元素，float 除 none 以外的值；   
      2、定位元素，position（absolute，fixed）；   
      3、display 为以下其中之一的值 inline-block，table-cell，table-caption；   
      4、overflow 除了 visible 以外的值（hidden，auto，scroll）

注意：在IE下, Layout,可通过zoom:1 触发

#### BFC布局特性

1.内部的Box会在垂直方向上一个接一个的放置。  
      2.垂直方向上的距离由margin决定  
      3.bfc的区域不会与float的元素区域重叠。  
      4.计算bfc的高度时，浮动元素也参与计算  
      5.bfc就是页面上的一个独立容器，容器里面的子元素不会影响外面元素。

#### BFC 布局与普通流布局的区别

普通流布局：

* + 浮动的元素是不会被父级计算高度
  + 非浮动元素会覆盖浮动元素的位置
  + margin会传递给父级元素
  + 两个相邻元素上下的margin会重叠

BFC布局：

* + 浮动的元素会被父级计算高度(父级元素触发了BFC)
  + 非浮动元素不会覆盖浮动元素的位置(非浮动元素触发了BFC)
  + margin不会传递给父级(父级触发BFC)
  + 属于同一个BFC的两个相邻元素上下margin会重叠

#### 实际应用的BFC布局：

* + 阻止margin重叠
  + 可以包含浮动元素 —— 清除内部浮动(清除浮动的原理是两个 div都位于同一个 BFC 区域之中)
  + 自适应两栏布局
  + 可以阻止元素被浮动元素覆盖

### Display 属性

|  |  |
| --- | --- |
| none | 此元素不会被显示。 |
| block | 此元素将显示为块级元素，此元素前后会带有换行符。 |
| inline | 默认。此元素会被显示为内联元素，元素前后没有换行符。 |
| inline-block | 行内块元素。（CSS2.1 新增的值） |
| list-item | 此元素会作为列表显示。 |
| run-in | 此元素会根据上下文作为块级元素或内联元素显示。 |
| compact | CSS 中有值 compact，不过由于缺乏广泛支持，已经从 CSS2.1 中删除。 |
| marker | CSS 中有值 marker，不过由于缺乏广泛支持，已经从 CSS2.1 中删除。 |
| table | 此元素会作为块级表格来显示（类似 <table>），表格前后带有换行符。 |
| inline-table | 此元素会作为内联表格来显示（类似 <table>），表格前后没有换行符。 |
| table-row-group | 此元素会作为一个或多个行的分组来显示（类似 <tbody>）。 |
| table-header-group | 此元素会作为一个或多个行的分组来显示（类似 <thead>）。 |
| table-footer-group | 此元素会作为一个或多个行的分组来显示（类似 <tfoot>）。 |
| table-row | 此元素会作为一个表格行显示（类似 <tr>）。 |
| table-column-group | 此元素会作为一个或多个列的分组来显示（类似 <colgroup>）。 |
| table-column | 此元素会作为一个单元格列显示（类似 <col>） |
| table-cell | 此元素会作为一个表格单元格显示（类似 <td> 和 <th>） |
| table-caption | 此元素会作为一个表格标题显示（类似 <caption>） |
| inherit | 规定应该从父元素继承 display 属性的值。 |

### 隐藏元素

#### Display

* none : 无 隐藏元素
* block : 显示 转换为块级元素

隐藏不会占用页面空间、会引起浏览器的重绘和重排

#### visibility

* visible ：显示
* hidden ：隐藏

隐藏之后依旧会占用页面空间

#### Opacity：”

设置透明度为0 ，隐藏后依旧会占用页面空间

#### Position

通过定位，将元素定位在非可视区域实现隐藏

#### overflow

* hidden :溢出的部分隐藏掉
* visible : 显示
* auto ：自动出现滚动条
* scroll ：一直有滚动条

### Transform

Scale：可以用来修改图片大小

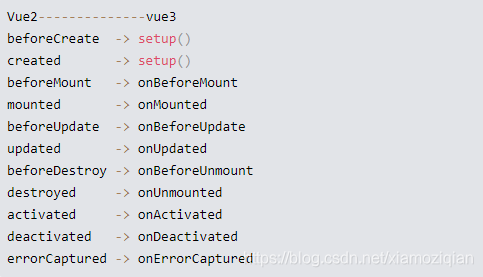
### 实现三角形

三角形实现原理：宽度width为0；height为0；

1. 有一条横竖边（上下左右）的设置为border-方向：长度 solid red，这个画的就是底部的直线。其他边使用border-方向：长度 solid transparent。
2. 有两个横竖边（上下左右）的设置，若斜边是在三角形的右边，这时候设置top或bottom的直线，和右边的斜线。若斜边是在三角形的左边，这时候设置top或bottom的直线，和左边的斜线。

## Vue3.0

#### 生命周期



#### 父子组件生命周期执行顺序：

创建与挂载

父beforeCreate > 父created > 父beforeMount > 子beforeCreate > 子created > 子beforeMount > 子mounted > 父mounted

更新  
父beforeUpdate > 子beforeUpdate > 子updated > 父updated

销毁

父beforeDestroy > 子beforeDestroy > 子destroyed > 父destroyed

#### Mixin生命周期执行顺序：

mixin的beforeCreate > 父beforeCreate > mixin的created > 父created > mixin的beforeMount > 父beforeMount > 子beforeCreate > 子created > 子beforeMount > 子mounted > mixin的mounted >父mounted

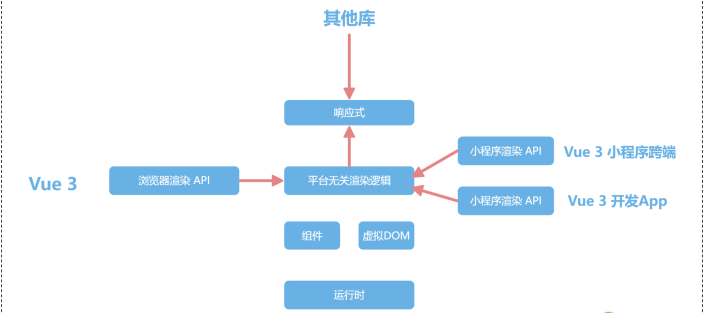
#### Keep-alive声明周期执行顺序：

|  |  |
| --- | --- |
| created | activated |
| 触发顺序 | 组件创建最初始 | created  =>  mounted =>activated |
| 触发次数 | 只在组件刚创建时创建 | 在使用keep-alive标签中有效，每次进入都会执行钩子中的函数 |

### Vue 2 和 vue 3 对比

#### 响应式

1. 响应式：Vue的响应式只服务于Vue；Vue3的响应式就和Vue解耦了渲染的逻辑也拆成了平台⽆关渲染逻辑和浏览器渲染API两部分

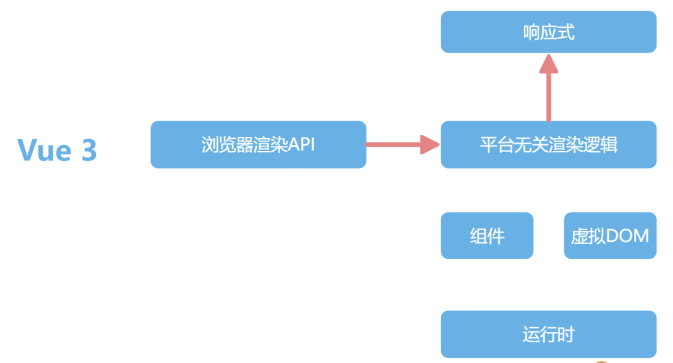


2、deﬁneProperty 和 proxy：deﬁneProperty对不存在的属性⽆法拦截，所以 Vue

中所有数据必须要在data⾥声明；Proxy还可以监听更多的数据格式，⽐如Set、Map，这是 Vue2做不到的

更重要的是，我觉得Proxy代表⼀种⽅向，就是框架会越来越多的拥抱浏览器的新特性

1. 自定义渲染器：



#### 响应式机制总结：

Vue 中用过三种响应式解决方案，分别是 defineProperty、Proxy 和 value setter。

1. [Object.defineProperty() (opens new window)](https://cloud.tencent.com/developer/tools/blog-entry?target=https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty&source=article&objectId=2287825" \t "https://cloud.tencent.com/developer/article/_blank) 方法会直接在一个对象上定义一个新属性，或者修改一个对象的现有属性，并返回此对象

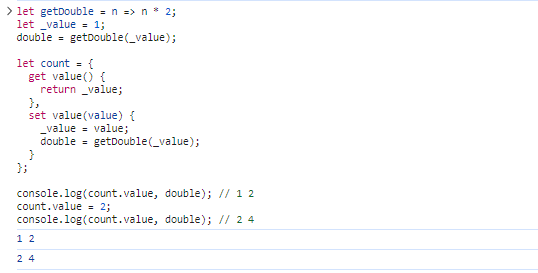
但 defineProperty API 作为 Vue 2 实现响应式的原理，它的语法中也有一些缺陷。比如在下面代码中，删除 obj.count 属性，set 函数就不会执行，double 还是之前的数值。这也是为什么在 Vue 2 中，需要 $delete 一个专门的函数去删除数据

1. Vue 3 的响应式机制是基于 [Proxy (opens new window)](https://cloud.tencent.com/developer/tools/blog-entry?target=https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Proxy&source=article&objectId=2287825" \t "https://cloud.tencent.com/developer/article/_blank) 实现的，其重要意义在于它解决了 Vue 2 响应式的缺陷

Proxy 是针对对象来监听，而不是针对某个具体属性，所以不仅可以代理那些定义时不存在的属性，还可以代理更丰富的数据结构，比如 Map、Set 等，并且也能通过 deleteProperty 实现对删除操作的代理。

Vue 3 的 reactive 函数可以把一个对象变成响应式数据，而 reactive 就是基于 Proxy 实现的。还可以通过 watchEffect，在 obj.count 修改之后，执行数据的打印

1. 在 Vue 3 中还有另一个响应式实现的逻辑，就是利用对象的 get 和 set 函数来进行监听，这种响应式的实现方式，只能拦截某一个属性的修改，这也是 Vue 3 中 ref 这个 API 的实现。



#### 全部模块使用typescript 重构：

1. 类型系统带来了更⽅便的提示
2. 并且让我们的代码能够更健壮

#### Composition API ：

使⽤CompositionAPI后，代码看起来很烦琐，没有Vue但OptionsAPI的写法也有⼏个很严重的问题：

1. 所有数据都挂载在this之上，因⽽OptionsAPI好，并且这样也不好做Tree-shaking清理代码
2. 新增功能基本都得修改data、method等配置，并且代码上300⾏之后，会经常上下反复横跳，开发很痛苦。
3. 代码不好复⽤，Vue2的组件很难抽离通⽤逻辑，只能使⽤mixin，还会带来命名冲突的题
4. Composition API 后，所有API都是import引⼊的（现在我们的例⼦还没有⼯程化，后续会加⼊）。⽤到的功能都import进来，对Tree-shaking很友好，没⽤到功能，打包的时候会被清理掉减⼩包的⼤⼩
5. 代码⽅便复⽤，可以把⼀个功能所有的methods、data封装在⼀个独⽴的函数⾥，复⽤代码⾮常容易。

#### 新的组件

Fragment: Vue3组件不再要求有⼀个唯⼀的根节点，清除了很多⽆⽤的占位div。

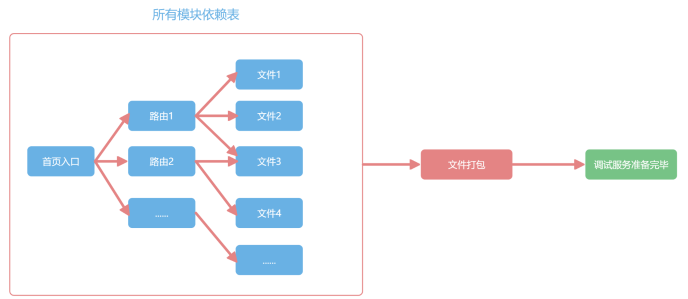
Teleport: 允许组件渲染在别的元素内，主要开发弹窗组件的时候特别有⽤

Suspense: 异步组件，更⽅便开发有异步请求的组件

#### 工程化工具vite

Webpack：根据你的import依赖逻辑，形成⼀个依赖图，然后调⽤对应的处理⼯具，把整个项⽬打包后，放在内存⾥再启动调试

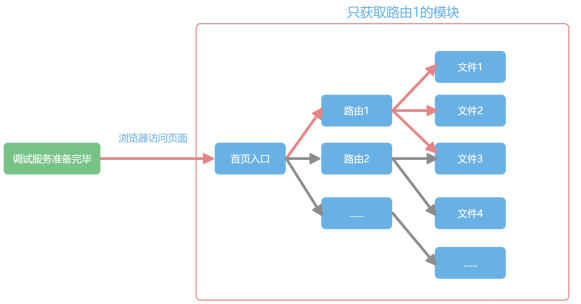
由于要预打包，所以复杂项⽬的开发，启动调试环境需要3分钟都很常⻅



Vite：

现代浏览器已经默认⽀持了ES6的import语法，Vite就是基于这个原理来实现的

在调试环境下，我们**不需要全部预打包**，只是把你⾸⻚依赖的⽂件，依次通过⽹络请求去获取，整个开发体验得到巨⼤提升，做到了复杂项⽬的秒级调试和热更新



#### Vue2 升级vue 3 的必要性

Vue 3 的缺点：

由于新的响应式系统⽤了Proxy，会存在兼容性问题

也就是说，如果你的应⽤被要求兼容IE11，就应该选择Vue2。⽽且，Vue团队也已经放弃

Vue3对**IE11**浏览器的⽀持

#### Vue3 不兼容的写法

组件注册：Vue.component('el-counter',...)

使用const app = createApp({}) app.component(...)替代

createApp还移除了很多我们常⻅的写法，⽐如在createApp中，就**不再⽀持ﬁlter、$on、$oﬀ、$set、$delete**等API

### Vue2 升级vue3

1. 项⽬依赖的 Vue版本换成Vue3
2. 引入@vue/compat
3. 给vue设置别名@vue/compat，也就是以compat作为⼊⼝

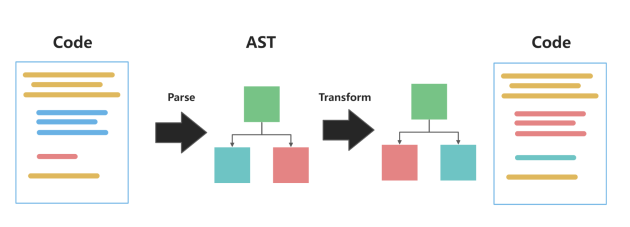


1. 这时你就会在控制台看到很多警告，以及很多优化的建议。我们参照建议，挨个去做优化就可以了
2. 在@vue/compat提供了很多建议后，我们⾃⼰还是要慢慢做修改。但从另⼀个⻆度看，“偷懒”是优秀程序员的标志，社区就有能够做⾃动化替换的⼯具，⽐较好⽤的就是“阿⾥妈妈”出品的gogocode，官⽅⽂档也写得很详细，就不在这⾥赘述了

⾃动化替换⼯具的原理很简单，和 Vue的Compiler优化的原理是⼀样的，也就是利⽤编译

原理做代码替换。如下图所示，我们利⽤ babel分析左边Vue2的源码，解析成AST，然后

根据Vue3的写法对AST进⾏转换，最后⽣成新的Vue3代码。



### Vue3.0 优点：

#### 响应式升级

* vue2在初始化的时候，对data中的每个属性使用definepropery调用getter和setter使之变为响应式对象。如果属性值为对象，还会递归调用defineproperty使之变为响应式对象。
* vue3使用proxy对象重写响应式。proxy的性能本来比defineproperty好，proxy可以拦截属性的访问、赋值、删除等操作，不需要初始化的时候遍历所有属性，另外有多层属性嵌套的话，只有访问某个属性的时候，才会递归处理下一级的属性。

##### Proxy优点：

proxy的性能本来比defineproperty好，proxy可以拦截属性的访问、赋值、删除等操作，不需要初始化的时候遍历所有属性，另外有多层属性嵌套的话，只有访问某个属性的时候，才会递归处理下一级的属性

#### 静态标记编译优化

* 优化编译和重写虚拟dom，让首次渲染和更新dom性能有更大的提升  
  vue2 通过标记静态根节点,优化 diff 算法  
  vue3 标记和提升所有静态根节点,diff 的时候只比较动态节点内容
* Fragments, 模板里面不用创建唯一根节点,可以直接放同级标签和文本内容
* 静态提升
* patch flag, 跳过静态节点,直接对比动态节点
* 缓存事件处理函数

#### tree-shaking

**特性：**tree shaking特性，将全局 API 进行分块。如果你不使用其某些功能，它们将不会包含在你的基础包中

**原理：**Tree shaking是基于ES6模板语法（import与exports），主要是借助ES6模块的静态编译思想，在编译时就能确定模块的依赖关系，以及输入和输出的变量

* 编译阶段利用ES6 Module判断哪些模块已经加载
* 判断那些模块和变量未被使用或者引用，进而删除对应代码

#### 组合式api

vue3 新增的一组 api，它是基于函数的 api，可以更灵活的组织组件的逻辑。  
解决options api在大型项目中，options api不好拆分和重用的问题

Composition API

Composition API最大的优点通俗的讲就是把跟一个功能相关的东西放在一个地方，它是目前最合理也是最容易维护的，你可以随时将功能的一部分拆分出去。你可以将每一个功能相关所有的东西比如methods，computed都放在如上图的function中，这个function可以独立的存在

Options API

代码拆分困难、代码复用困难使用minxin可能会导致命名重复的问题

一个功能往往需要在不同的vue配置项中定义属性和方法，比较分散，项目小还好，清晰明了，但是项目大了后，一个methods中可能包含很多个方法，往往分不清哪个方法对应着哪个功能，而且当你想要新增一个功能的时候你可能需要在 data，methods，computed，watch中都要写一些东西，但是这个时候每个选项里面的内容很多你需要上下来回的翻滚，特别影响效率。

总结

 在逻辑组织和逻辑复用方面，Composition API是优于Options API

 因为Composition API几乎是函数，会有更好的类型推断。

 Composition API对 tree-shaking 友好，代码也更容易压缩

 Composition API中见不到this的使用，减少了this指向不明的情况

 如果是小型组件，可以继续使用Options API，也是十分友好的

#### 实例对象的创建

Vue2.x中new出的实例对象，所有的东西都在这个vue对象上，这样其实无论你用到还是没用到，都会跑一变，这样不仅提高了性能消耗，也无疑增加了用户加载时间。

而vue3.0中可以用ES module imports按需引入

减少了内存消耗，也同时减少了用户加载时间

## Pinia

Vue3.0 用来替换vuex的状态管理器

### 优点

1. 支持vue3、vue2
2. 抛弃了mutations 的操作，只有state、getters 和actions
3. 不需要嵌套模块 ，符合vue3和组合式api
4. 完整支持typescript
5. 代码更加简介

### 学习笔记pinia

#### 添加pinia 依赖：

Yarn add pinia

Npm install pinia

#### Pinia 使用

在跟文件mian.js 中初始化注册pinia

Vue3.0

import { createPinia } from 'pinia'

app.*use*(*createPinia*())

vue2.0

import { createPinia, PiniaVuePlugin } from 'pinia'

Vue.*use*(PiniaVuePlugin)

const pinia = *createPinia*()

new Vue({

el: '#app',

*// other options...*

*// ...*

*// note the same `pinia` instance can be used across multiple Vue apps on*

*// the same page*

pinia,

})

#### State

1. 必须是函数：
2. 必须是箭头函数

#### Actions

类似组件的methods， 封装业务逻辑，修改state

#### Getter

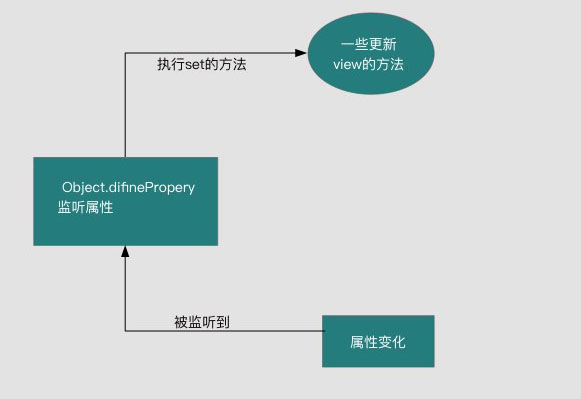
类似computed ，用来封装计算属性，有缓存的特性

## Vue

### 解读vue

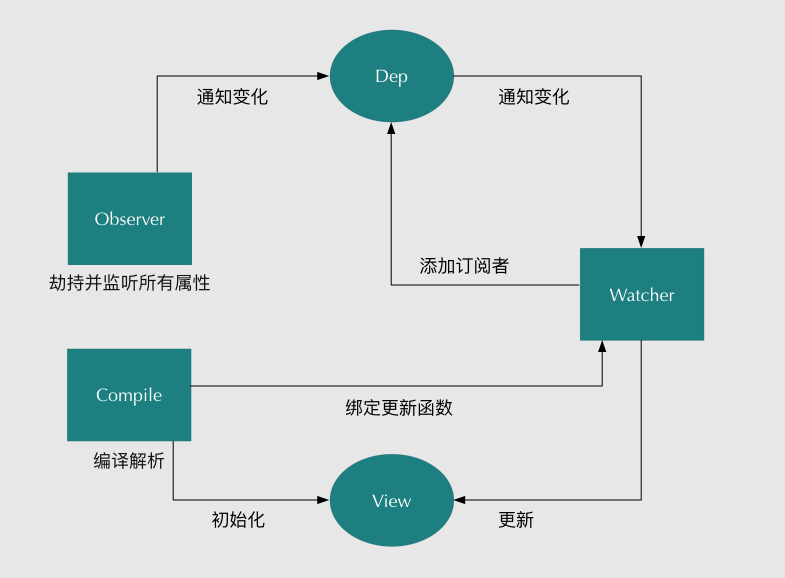
#### Vue双向绑定原理

MVVM双向绑定，采用的是**数据劫持结合发布者-订阅者模式**的方式，通过Object.defineProperty（）来劫持各个属性的setter、getter，在数据变动时发布消息给订阅者，触发监听回调



**具体步骤：**

1. 需要 observe 的数据对象进行递归遍历，包括子属性对象的属性，都加上 setter 和 getter  
   这样的话，给这个对象的某个值赋值，就会触发 setter，那么就能监听到了数据变化
2. compile 解析模板指令，将模板中的变量替换成数据，然后初始化渲染页面视图，并将每个指令对应的节点绑定更新函数，添加监听数据的订阅者，一旦数据有变动，收到通知，更新视图
3. Watcher 订阅者是 Observer 和 Compile 之间通信的桥梁，主要做的事情是:
   * 在自身实例化时往属性订阅器(dep)里面添加自己
   * 自身必须有一个 update() 方法
   * 待属性变动 dep.notice() 通知时，能调用自身的 update() 方法，并触发 Compile 中绑定的回调，则功成身退。
4. MVVM 作为数据绑定的入口，整合 Observer、Compile 和 Watcher 三者，通过Observer来监听自己的 model 数据变化，通过 Compile 来解析编译模板指令，最终利用 Watcher 搭起 Observer 和 Compile 之间的通信桥梁，达到数据变化 -> 视图更新；视图交互变化(input) -> 数据 model 变更的双向绑定效果。



#### 模板语法

* 1. 本质上是字符串
  2. 有逻辑，比如v-if 和v-for
  3. 与HTML格式很像，但是又有很大的区别
  4. 最终要转换为HTML来显示
  5. 模板需要解析威JavaScript（

有逻辑，必须使用JavaScript 实现； 模板转换为HTML进行页面渲染，基于这两个原因，模板最重要的是转换为一个JavaScript 函数（render函数）所以整体流程是： 模板字符串---JavaScript渲染函数---渲染为HTML）

#### Render函数解读



模板中所有信息都包含在了render函数中

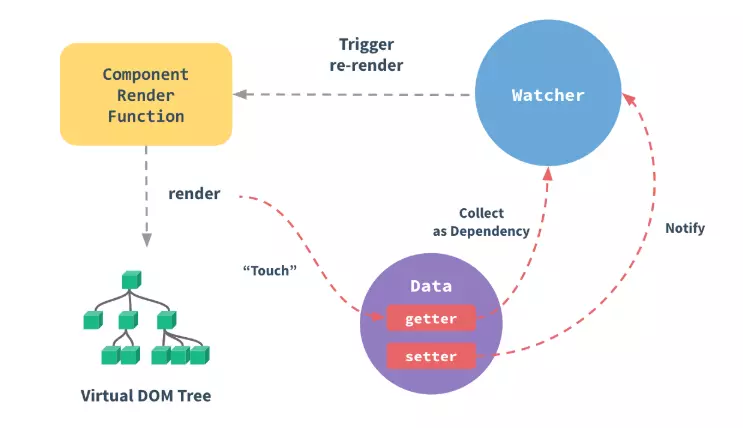
this即vm（所以在return作用下，都挂载了this/vm下面）

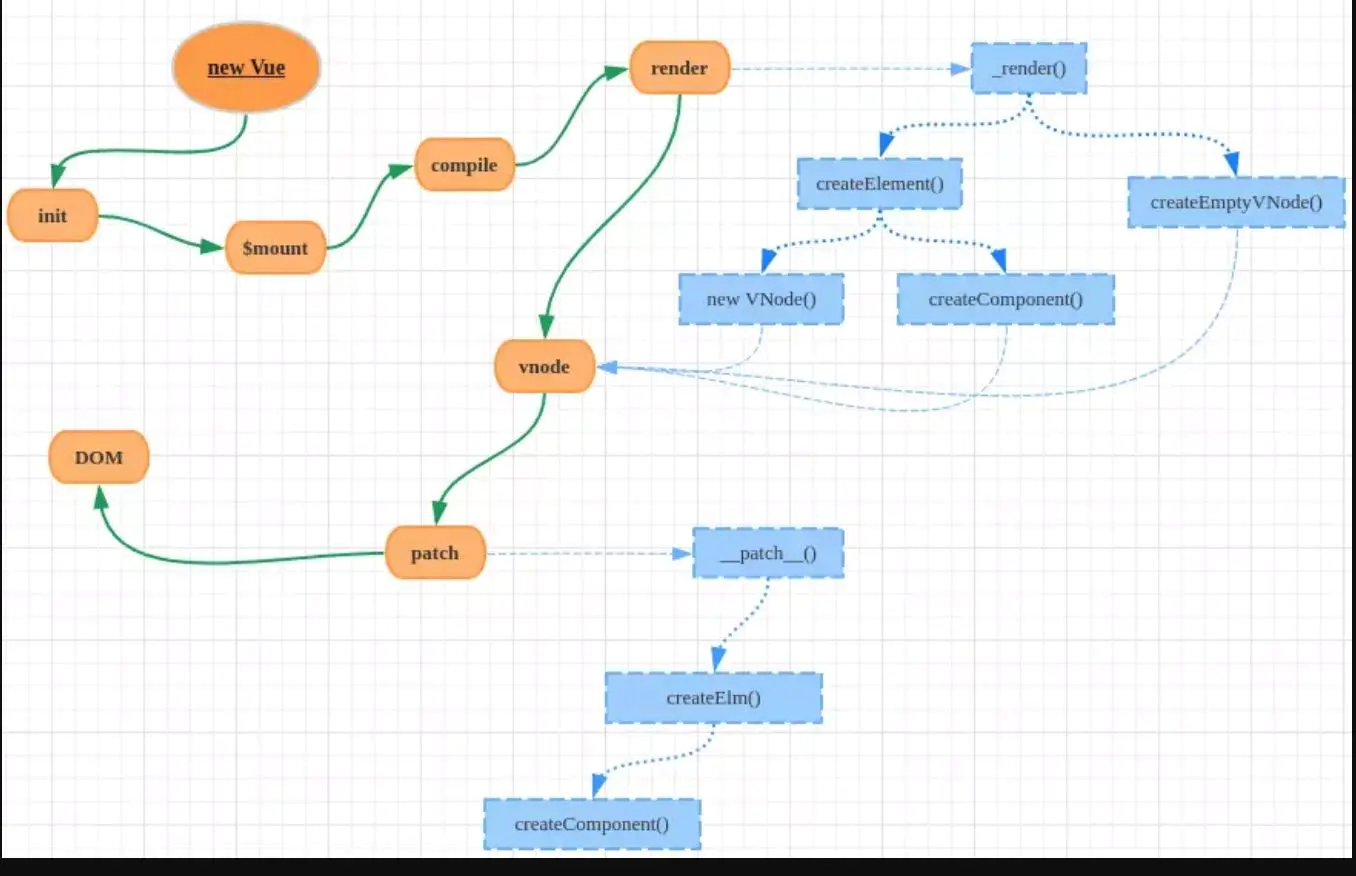
price即this.price即vm.price，即data中的price

\_c即this.\_c即vm.\_c

#### Vue 项目运行解析

生命周期：开始创建、初始化数据、编译模板、挂载Dom→渲染、更新→渲染、卸载





##### 第一步： 解析模板生成render函数

with的用法

模板中所有信息都被render函数包含

模板中用到的data中的属性，都变成了JS变量

模板中v-model，v-for，v-on都变成了JS逻辑

**render函数返回vnode**

##### 第二步： 响应式开始监听

Object.defineProperty

将data的属性代理到vm上

##### 第三步： 首次渲染，显示页面，且绑定依赖

1.初次渲染，执行updateComponent，执行vm.\_render()

2.执行render函数，会访问到vm.list和vm.title

3.会被响应式的get方法监听到（2，3步是因为生成的render函数中有get/set方法）

4.执行updateComponent，会走到vdom的patch方法

5.patch将vnode渲染成DOM，初次渲染完成

##### 第四步： data属性变化，触发rerender

修改属性，被响应式的set监听到  
set中执行updateComponent  
updateComponent重新执行vm.\_render()  
生成的vnode和prevVnode，通过patch进行对比  
渲染到HTML中

小结：

1、new Vue，执行初始化  
2、挂载$mount方法，通过自定义Render方法、template、el等生成Render函数  
3、通过Watcher监听数据的变化  
4、当数据发生变化时，Render函数执行生成VNode对象  
5、通过patch方法，对比新旧VNode对象，通过DOM Diff算法，添加、修改、删除真正的DOM元素  
至此，整个new Vue的渲染过程完毕。

### 路由守卫

**全局钩子函数**（beforeEach、afterEach)

**路由独享的钩子函数**（beforeEnter）

**组件内钩子函数**（beforeRouterEnter、beforeRouterUpdate、beforeRouterLeave）

### 事件绑定

#### 原生JavaScript事件绑定原理：

通过addEventListener方法进行事件绑定

#### Vue事件绑定原理

Vue中通过v-on或其语法糖@指令来给元素绑定事件并且提供了事件修饰符，基本流程是进行模板编译生成AST，生成render函数后并执行得到VNode，VNode生成真实DOM节点或者组件时候使用addEventListener方法进行事件绑定。

#### 单项数据流

子组件可以使用父组件传递过来的数据 但是不能修改这个数据

  解决方案：子组件复制一份父组件的参数自己使用

#### Vue事件绑定描述

v-on与@用于绑定事件监听器，事件类型由参数指定，表达式可以是一个方法的名字或一个内联语句，如果没有修饰符也可以省略，用在普通元素上时，只能监听原生DOM事件，用在自定义元素组件上时，也可以监听子组件触发的自定义事件，在监听原生DOM事件时，方法以事件为唯一的参数，如果使用内联语句，语句可以访问一个$event property：v-on:click="handle('param', $event)"，自2.4.0开始v-on同样支持不带参数绑定一个事件或监听器键值对的对象，注意当使用对象语法时，是不支持任何修饰器的。

#### 事件修饰符

* .stop: 调用event.stopPropagation()，即阻止事件冒泡。
* .prevent: 调用event.preventDefault()，即阻止默认事件。
* .capture: 添加事件侦听器时使用capture模式，即使用事件捕获模式处理事件。
* .self: 只当事件是从侦听器绑定的元素本身触发时才触发回调。
* .{keyCode | keyAlias}: 只当事件是从特定键触发时才触发回调。
* .native: 监听组件根元素的原生事件，即注册组件根元素的原生事件而不是组件自定义事件的。
* .once: 只触发一次回调。
* .left(2.2.0): 只当点击鼠标左键时触发。
* .right(2.2.0): 只当点击鼠标右键时触发。
* .middle(2.2.0): 只当点击鼠标中键时触发。
* .passive(2.3.0): 以{ passive: true }模式添加侦听器，表示listener永远不会调用preventDefault()。

### .sync 实现父子组件双向绑定

#### 如何实现

$emit(“update.属性名称”, 修改的值)

属性名称.sync

#### 缺点

子组件修改props的值现在vue会报错了

乱修改难定位问题，违反单一数据流原则了

### Methods、Watch、Computed 区别

#### Computed

当页面中有某些数据依赖其他数据进行变动的时候，可以使用计算属性。

不支持异步，当computed内有异步操作时无效，无法监听数据的变化

如果computed 属性属性值是函数，那么默认会走get方法；函数的返回值就是属性的属性值；在computed中的，属性都有一个get和一个set方法，当数据变化时，调用set方法。

computed是具有缓存的，这就意味着只要计算属性的依赖没有进行相应的数据更新，那么computed会直接从缓存中获取值，多次访问都会返回之前的计算结果。

#### Watch

watch用于观察和监听页面上的vue实例，当然在大部分情况下我们都会使用computed，但如果要在数据变化的同时进行异步操作或者是比较大的开销，那么watch为最佳选择

数据必须是data中声明过或者父组件传递过来的props中的数据，当数据变化时，触发其他操作，函数有两个参数

#### 总结

计算是通过变量计算来得出数据。而观察是观察一个特定的值，根据被观察者的变动进行相应的变化

### 组件

#### 异步组件

1.异步组件可以减少打包的结果。会将异步组件分开打包，会采用异步的方式加载组件，可以有效的解决一个组件过大的问题。不使用异步组件，如果组件功能比较多打包出来的结果就会变大。

2.异步组件的核心可以给组件定义变成一个函数，函数里面可以用import语法，实现文件的分割加载，import语法是webpack提供的，采用的就是jsonp。



##### 原理：

在createComponent方法中，会有相应的异步组件处理，首先定义一个asyncFactory变量，然后进行判断，如果组件是一个函数，然后会去调resolveAsyncComponent方法，然后将赋值在asyncFactory上的函数传进去，会让asyncFactory马上执行，执行的时候并不会马上返回结果，因为他是异步的，返回的是一个promise，这时候这个值就是undefined，然后就会先渲染一个异步组件的占位，空虚拟节点。如果加载完之后会调factory函数传入resolve和reject两个参数，执行后返回一个成功的回调和失败的回调，promise成功了就会调resolve，resolve中就会调取forceRender方法强制更新视图重新渲染，forceRender中调取的就是$forceUpdate，同时把结果放到factory.resolved上，如果强制刷新的时候就会再次走resolveAsyncComponent方法，这时候有个判断，如果有成功的结果就把结果直接放回去，这时候resolveAsyncComponent返回的就不是undefined了，就会接的创建组件，初始化组件，渲染组件。

#### 组件通信

* 父子组件的通信
* 非父子组件的eventBus通信
* 利用本地缓存实现组件通信
* Vuex通信

### 路由

#### 路由跳转相同路径解读

##### 原因：

vue官网详细解释说明使用同一路由携带不同参数，本质上是重用相同的组件实例，默认在跳转路由时会采用缓存策略,并不会刷新当前路由组件，因此不会调用组件的生命周期挂钩。

##### 解决办法

[跳转同一路由但是参数不同，组件不刷新的问题 - 掘金 (juejin.cn)](https://juejin.cn/post/6905729933558874119)

[【vue】---- 相同路由不同参数的模块展示进行跳转时页面不刷新问题 - ONE橙子 - 博客园 (cnblogs.com)](https://www.cnblogs.com/pinkpinkc/p/13223010.html)

#### Router 和 route 的区别

##### Route

Route 对象：表示当前路由信息，包含了当前URL解析得到的信息。包含当前路径，参数，query对象等

.    $route.path      字符串，对应当前路由的路径，总是解析为绝对路径，如"/foo/bar"。

    2.    $route.params      一个 key/value 对象，包含了 动态片段 和 全匹配片段，      如果没有路由参数，就是一个空对象。

    3.    $route.query      一个 key/value 对象，表示 URL 查询参数。      例如，对于路径 /foo?user=1，则有$route.query.user == 1，      如果没有查询参数，则是个空对象。

    4.    $route.hash      当前路由的hash值 (不带#) ，如果没有 hash 值，则为空字符串。锚点\*

    5.    $route.fullPath      完成解析后的 URL，包含查询参数和hash的完整路径。

    6.    $route.matched      数组，包含当前匹配的路径中所包含的所有片段所对应的配置参数对象。

    7.    $route.name    当前路径名字

8.    $route.meta  路由元信息

##### Router对象：

Router对象是全局路由实例，是router构造方法的实例。

#### 传参

1. 直接拼接在url 后面 （this.$route.params.XX）
2. 使用params 携带参数 （this.$route.params.XX）
3. 使用query携带参数 (this.$route.query.XX)

### VueX

#### 核心：

1. State

存储应用状态数据的对象，与vue组件中data类似

state的值可以是对象，也可以是返回对象的函数  
通过store.state访问状态数据

1. Getter

从state中派生的状态数据  
接收state作为第一个参数，第二个为可选参数  
类似组件中的 computed，派生数据，在数据出门后进行的加工（对原数据再加工,计算后的结果），这些处理都在getters中

1. Mutations

提交mutation来修改store中的状态，同步操作  
每个mutation都有一个字符串事件类型(type)与一个回调函数(handler),在回调函数中修改状态

注意：

* 不能直接去调用mutation的回调函数，需要当mutation类型为increment时，才能调用此函数
* mutation必须是同步的
* 在store中初始化时设置好所有的属性

1. Actions

与mutations类似，提交修改state的行为，处理异步任务  
注意：

* 提交的是mutation，不是直接修改状态
* 可以包含任意异步操作

1. Modules

将store分割成模块

每个模块拥有自己的 state、mutation、action、getters，

模块还可以嵌套子模块——从上至下进行类似的分割

注意：

将当前业务所需内容 封装到独立的js文件中，以对象的形式导出

在index.js（主控）, 的modules中 注册模块

组件中使用具体方法时，需要注意语法规范

#### 工作流程

Vue组件（action里面的dispatch )--> actions(commit方法） -->mutations（Mutate）--> state（getter） -->store更新所有调用vuex的组件（Vue Component组件）

### Vue项目优化

#### v-if 和v-show

**v-if** 是 **真正** 的条件渲染，因为它会确保在切换过程中条件块内的事件监听器和子组件适当地被销毁和重建；也是**惰性的**：如果在初始渲染时条件为假，则什么也不做——直到条件第一次变为真时，才会开始渲染条件块。

**v-show** 就简单得多， 不管初始条件是什么，元素总是会被渲染，并且只是简单地基于 CSS 的 display 属性进行切换。

所以，v-if 适用于在运行时很少改变条件，不需要频繁切换条件的场景；v-show 则适用于需要非常频繁切换条件的场景。

#### Computed 和watch

**computed：** 是计算属性，依赖其它属性值，并且 computed 的值有缓存，只有它依赖的属性值发生改变，下一次获取 computed 的值时才会重新计算 computed 的值；

**watch：** 更多的是「观察」的作用，类似于某些数据的监听回调 ，每当监听的数据变化时都会执行回调进行后续操作；

**运用场景：**

**当我们需要进行数值计算，并且依赖于其它数据时，应该使用 computed，因为可以利用 computed 的缓存特性，避免每次获取值时，都要重新计算；**

**当我们需要在数据变化时执行异步或开销较大的操作时，应该使用 watch，使用 watch 选项允许我们执行异步操作 ( 访问一个 API )，限制我们执行该操作的频率，并在我们得到最终结果前，设置中间状态。这些都是计算属性无法做到的。**

#### v-for 和v-if

**v-for 遍历必须为 item 添加 key**

方便 Vue.js 内部机制精准找到该条列表数据。当 state 更新时，新的状态值和旧的状态值对比，较快地定位到 diff 。

**v-for 遍历避免同时使用 v-if**

v-for 比 v-if 优先级高，如果每一次都需要遍历整个数组，将会影响速度，尤其是当之需要渲染很小一部分的时候，必要情况下应该替换成 computed 属性。

#### 长列表性能优化

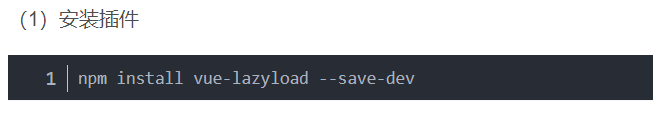
通过Object.freeze 方法冻结对象、可以禁止vue 劫持数据，减少初始化时间

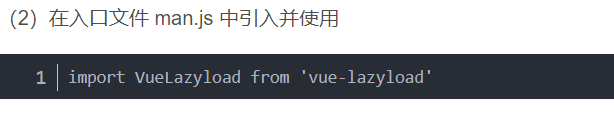
#### 事件销毁

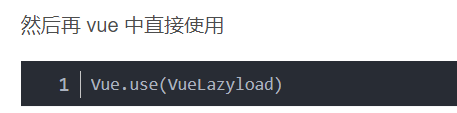
1. 解绑指令
2. 事件监听器 （避免内存泄漏）
3. 清除定时器

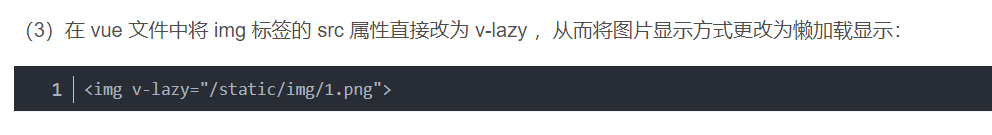
#### 图片资源懒加载

vue-lazyload 插件



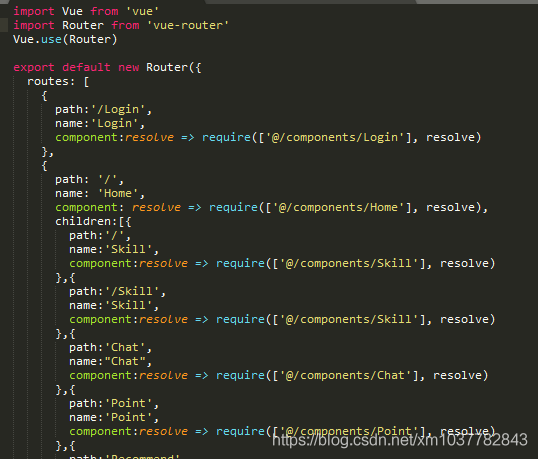


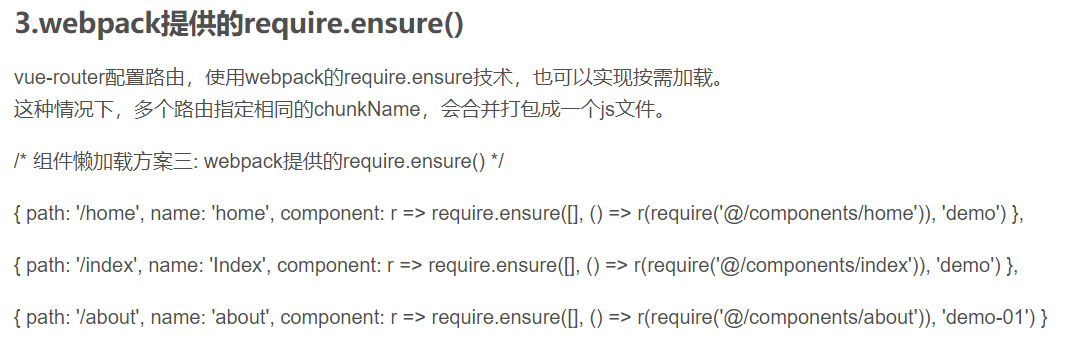




#### 路由懒加载

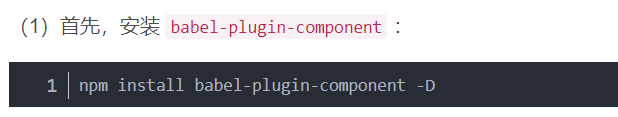
把不同路由对应的组件分割成不同的代码块，然后当路由被访问的时候才加载对应的组件



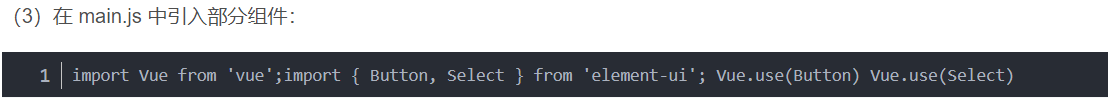


#### 按需引入第三方插件

借助 babel-plugin-component ，然后可以只引入需要的组件，以达到减小项目体积的目的







#### 服务器端渲染ssr 或者与渲染

### 深入剖析vue响应式

响应式核心： Observer/Watcher/Dep/ Compile

#### Observer

(使用Object.defineProperty 初始化数据，转化data中的数据转化为getter 、setter)

Observer 阶段，会为每一个key 创建一个dep 实例，如果key 被watch实例 get到

会把watcher实例添加到dep实例队列里面，如果key被set 则通知key对应的dep实例，

然后dep 实例会将依次通知队列里面的watcher实例，然他们执行自身的回调方法

#### Watcher

调用$watch，创建watch 实例监听每一个data 中的数据

每次监听到data中数据的变化都会调用update 方法，update方法对比修改的新旧值

#### Dep

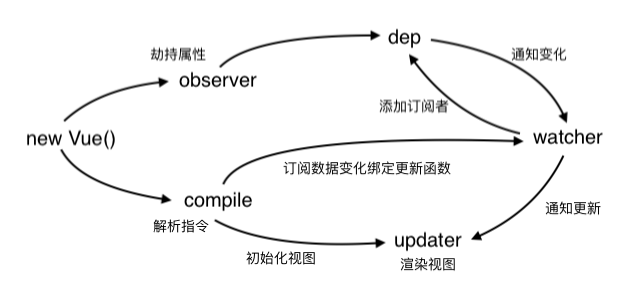
Dep 实例是收集该key 所有的watcher实例的地方

#### Compile

（解析指令）

#### 总结：

MVVM 作为入口函数，整合 Observer、Compile、Watcher 三者，通过 Observer 监听数据变化，通过 Compile 解析编译模版指令，通过 Watcher 连接 Observer 和 Compile ，达到数据更新时视图更新，视图交互更新时数据更新的双向绑定效果



### 响应式

#### Vue2.x不响应的几种情况

数组：

* 1. 直接利用数组索引设置修改项时
  2. 修改数组长度

对象：

* 1. 向响应式对象添加属性
  2. 向响应式对象添加属性

#### 原因

无法监测到对象属性的新增和删除

Object.defineProperty 可以监测到数组的下标的操作，但是vue为了性能考虑操作下标、数组长度的时候不响应

#### 解决办法：

1、创建新的数组替换原有数组值

2、使用JavaScript的数组操作函数，这些方法都会返回一个新数组，也是数组替换原理；

支持的方法：

Vue 包含一组观察数组的变异方法，所以它们也将会触发视图更新。这些方法如下：

push()

pop()

shift()

unshift()

splice()

sort()

reverse()

不支持的方法：

filter()、 concat()、slice()

3、使用vue自带的 vue.set(object , key , value ); 向响应式对象添加属性；

4、使用vue自带的 vue.delete(object , key ); 向响应式对象删除属性；

5、对象添加属性还可以使用Object.assign({},obj1,obj2)返回获取的新对象替换原有对象；

// 代替 `Object.assign(this.someObject, { a: 1, b: 2 })`this.someObject = Object.assign({}, this.someObject, { a: 1, b: 2 })

### Watch 总结

1. immediate

当不设置该属性时执行顺序是在mounted 之后

当设置该属性时执行顺序将会提前到created之前

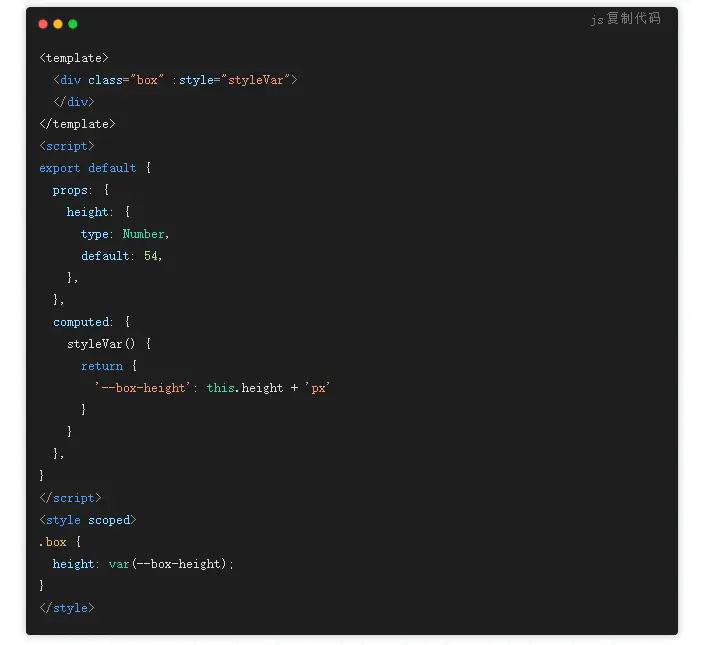
1. deep

当需要监听引用类型的数据时设置

deep的意思就是深入观察，监听器会一层层的往下遍历，给对象的所有属性都加上这个监听器，但是这样性能开销就会非常大了，任何修改obj里面任何一个属性都会触发这个监听器里的 handler。

### Vue与Css

#### 如何在css 中使用data中的变量



#### Vue2.x 如何监听数组变化

使用了函数劫持的方式，重写了数组的方法，Vue将data中的数组进行了原型链重写，指向了自己定义的数组原型方法。这样当调用数组api时，可以通知依赖更新。如果数组中包含着引用类型，会对数组中的引用类型再次递归遍历进行监控。这样就实现了监测数组变化。

## Typescript

### 接口（interface）和类型（type）

#### 区别

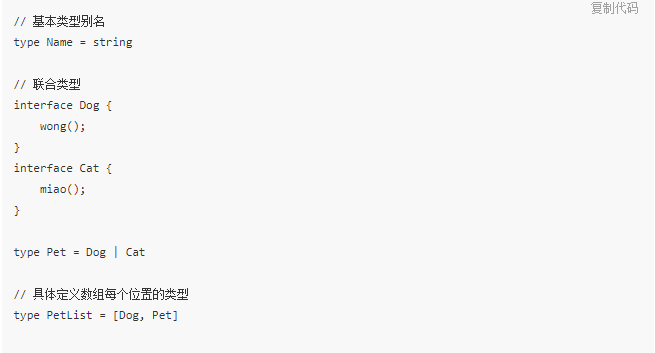
**官方文档：**

 An interface can be named in an extends or implements clause, but a type alias for an object type literal cannot.

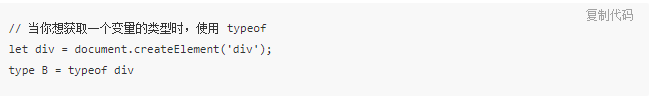
 An interface can have multiple merged declarations, but a type alias for an object type literal cannot.

**type可以而interface不可以**

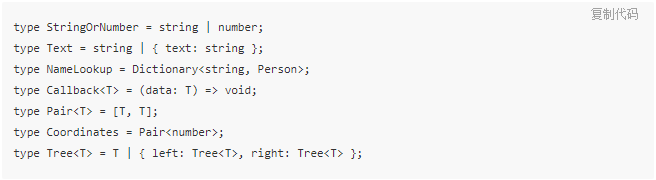
* type 可以声明基本类型别名，联合类型，元组等类型



* type 语句中还可以使用 typeof 获取实例的 类型进行赋值



* 其他骚操作



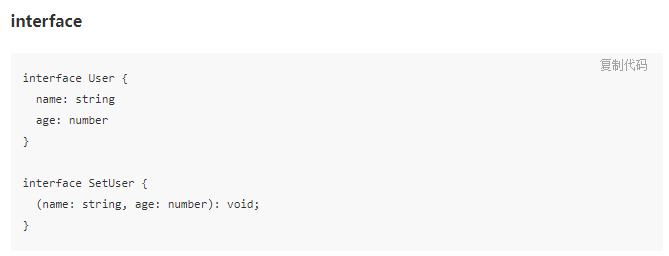
**interface可以而type不可以**

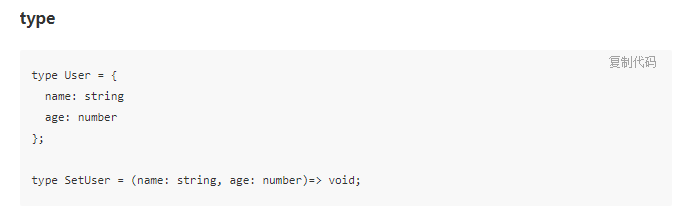
interface能够声明合并



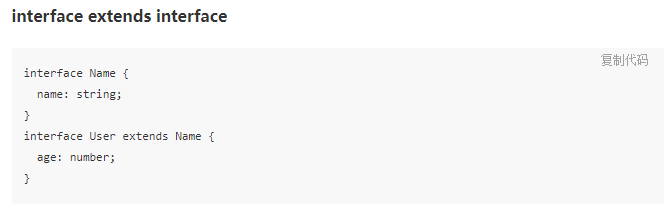
#### 相同点

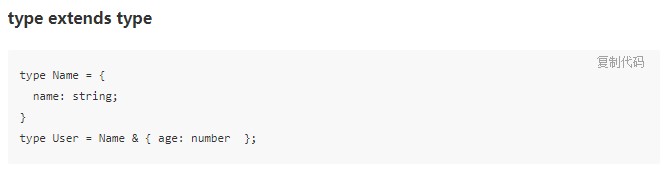
* 1. 都可以描述一个对象或者函数

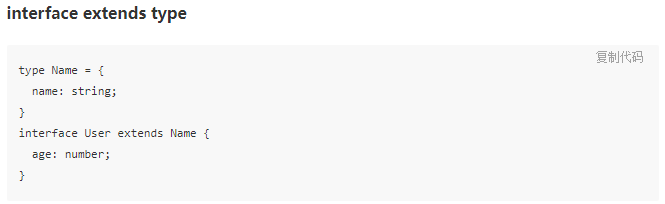


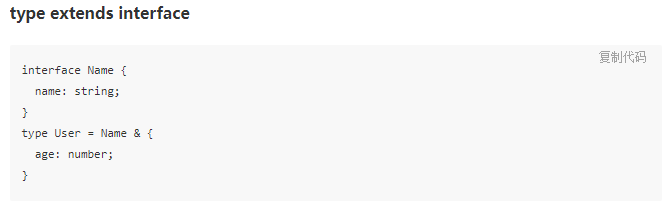


* 1. 都允许拓展（extends）









## EcmaScript

### 操作符

#### 空值合并操作符（??）

只有当左侧为null和undefined时，才会返回右侧的数

空值合并操作符（??）是一个逻辑操作符，当左侧的操作数为 [null](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/null) 或者 [undefined](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/undefined) 时，返回其右侧操作数，否则返回左侧操作数

区别：

与[逻辑或操作符（||）](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Logical_Operators" \l "Logical_OR_2)不同，逻辑或操作符会在左侧操作数为[假值](https://developer.mozilla.org/zh-CN/docs/Glossary/Falsy)时返回右侧操作数。也就是说，如果使用 || 来为某些变量设置默认值，可能会遇到意料之外的行为。比如为假值（例如，'' 或 0）时。见下面的例子。

#### 可选链操作符( ?. )

选链操作符( **?.** )允许读取位于连接对象链深处的属性的值，而不必明确验证链中的每个引用是否有效。?. 操作符的功能类似于 . 链式操作符，不同之处在于，在引用为空([nullish](https://wiki.developer.mozilla.org/en-US/docs/Glossary/nullish) ) ([null](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/null) 或者 [undefined](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/undefined)) 的情况下不会引起错误，该表达式短路返回值

#### 逻辑运算符（|| 、&&）

### Set()

ES6提供了新的数据结构Set。它类似于数组，但是成员的值都是唯一的，没有重复的值

Set本身是一个构造函数，用来生成Set数据结构。

* 1. Set函数可以接受一个数组（或类似数组的对象）作为参数，用来初始化
  2. 除数组重复成员的方法
  3. 向Set加入值的时候，不会发生类型转换

#### 方法属性

* add(value)：添加某个值，返回Set结构本身。
* delete(value)：删除某个值，返回一个布尔值，表示删除是否成功。
* has(value)：返回一个布尔值，表示该值是否为Set的成员。
* clear()：清除所有成员，没有返回值。

#### 实现遍历的方法

Set结构的实例有四个遍历方法，可以用于遍历成员。

* keys()：返回键名的遍历器
* values()：返回键值的遍历器
* entries()：返回键值对的遍历器
* forEach()：使用回调函数遍历每个成员

### map

#### object弊端

JavaScript的对象（Object），本质上是键值对的集合（Hash结构），但是传统上只能用字符串当作键。这给它的使用带来了很大的限制

#### Map优点

似于对象，也是键值对的集合，但是“键”的范围不限于字符串，各种类型的值（包括对象）都可以当作键。也就是说，Object结构提供了“字符串—值”的对应，Map结构提供了“值—值”的对应，是一种更完善的Hash结构实现

#### Map属性和方法

* 1. size
  2. set
  3. get
  4. has
  5. delete
  6. clear

### weakMap

WeakMap结构与Map结构基本类似，唯一的区别是它只接受对象作为键名（null除外），不接受其他类型的值作为键名，而且键名所指向的对象，不计入垃圾回收机制。

### weakSet

前景：

　在这个示例中，将变量key设置为null时便清除了对初始对象的引用，但是Set集合却保留了这个引用，仍然可以使用展开运算符将Set集合转换成数组格式并从数组的首个元素取出该引用

　　大部分情况下这段代码运行良好，但有时候会希望当其他所有引用都不再存在时，让Set集合中的这些引用随之消失。举个例子，如果在Web页面中通过JS代码记录了一些DOM元素，这些元素有可能被另一段脚本移除，而又不希望自己的代码保留这些DOM元素的最后一个引用

　　为了解决这个问题，ES6中引入了另外一个类型：WeakSet集合(弱引用Set集合)

用Weakset构造函数可以创建WeakSet集合，集合支持3个方法：add()、has()和delete()

　WeakSet集合的使用方式与Set集合类似，可以向集合中添加引用，从中移除引用，也可以检査集合中是否存在指定对象的引用。也可以调用WeakSet构造函数并传入一个可迭代对象来创建WeakSet集合

区别：

WeakSet与Set最大的区别是WeakSet中的对象都是弱引用，即垃圾回收机制不考虑WeakSet对该对象的引用，也就是说，如果其他对象都不再引用该对象，那么垃圾回收机制会自动回收该对象所占用的内存，不考虑该对象还存在于WeakSet之中

由于上面这个特点，WeakSet的成员是不适合引用的，因为它会随时消失。另外，由于WeakSet内部有多少个成员，取决于垃圾回收机制有没有运行，运行前后很可能成员个数是不一样的，而垃圾回收机制何时运行是不可预测的，因此ES6规定WeakSet不可遍历

　1、在Weakset的实例中，如果向add()、has()和delete()这3个方法传入非对象参数都会导致程序报错

　　2、WeakSet集合不可迭代，所以不能被用于for-of循环

　　3、WeakSet集合不暴露任何迭代器(例如keys()和values()方法)，所以无法通过程序本身来检测其中的内容

　　4、WeakSet集合不支持forEach()方法

5、WeakSet集合不支持size属性

注意：WeakMap 和WeakSet 都是弱类型，代表没有键 的强引用所以JavaScript 在垃圾回收时可以清理掉整条记录、没有键值的迭代，只能通过钥匙才能取到相关的值，保证了内部的封装私有属性。

### Object.freeze()

模拟Object.freeze() 主要使用两个关键方法：Object.defineProperty（）、Object.seal()。

Object.defineProperty 可以定义对象已有的属性：不可删除、修改

Object.seal() 可以定义对象不能扩展、删除属性

### 对象、数组

#### 数组

##### For循环

最简单的一种，也是使用频率最高的一种，虽然性能不弱，但仍有优化空间

##### Foreach循环

数组自带的foreach循环，使用频率较高，实际上性能比普通for循环弱

##### For..in

效率是最低的

##### For…of

这种方式是es6里面用到的，性能要好于forin，但仍然比不上普通for循环

##### Map

实际效率还比不上foreach

##### Array.prototype.flat和 flatMap

将多维数组转换成低维数组

##### array.concat

**concat()**方法用于合并两个或更多数组。此方法不会更改现有的数组，而是返回一个新的数组

##### array.copyWithin

**copyWithin()**方法将数组的一部分简单地复制到同一数组中的另一个位置，并将其返回，而不修改其大小。

###### Parameters

target0 为基底的索引，复制序列到该位置。如果是负数，target 将从末尾开始计算。如果target大于等于arr.length，将会不发生拷贝。如果target在start之后，复制的序列将被修改以符合arr.length。

start0 为基底的索引，开始复制元素的起始位置。如果是负数，start 将从末尾开始计算。如果 start 被忽略，copyWithin 将会从0开始复制。

end0 为基底的索引，开始复制元素的结束位置。copyWithin 将会拷贝到该位置，但不包括end这个位置的元素。如果是负数， end 将从末尾开始计算。如果end 被忽略，copyWithin 将会复制到arr.length。

###### 返回值

改变了的数组。

##### array.entries

**entries()**方法返回一个新的**Array Iterator**对象，该对象包含数组中每个索引的键/值对。

##### array.some

**some()** 方法测试数组中的某些元素是否通过由提供的函数实现的测试。

##### array.every

**every()**方法测试数组的所有元素是否都通过了指定函数的测试。

###### 参数

callback函数为每个元素测试，取三个参数：currentValue（必需）在数组中处理的当前元素。index（可选）数组中正在处理的元素的索引。array（可选）数组every被调用。thisArg可选。执行callback时使用this的值。

###### 返回值

如果callback函数为每个数组元素返回一个正确值，则为true; 否则，为**false**。

##### array.fill

**fill()** 方法用一个固定值填充一个数组中从起始索引到终止索引内的全部元素。

##### array.filter

**filter()**方法创建一个新数组, 其包含通过所提供函数实现的测试的所有元素。

##### array.find

**find()** 方法返回数组中满足提供的测试函数的第一个元素的值。否则返回undefined。

##### array.reduce

**reduce()** 方法对累加器和数组中的每个元素（从左到右）应用一个函数，将其减少为单个值。

##### array.reduceRight

**reduceRight()** 方法接受一个函数作为累加器（accumulator）和数组的每个值（从右到左）将其减少为单个值

##### array.values

**values()**方法返回一个新的 **Array Iterator** 对象，该对象包含数组每个索引的值。

var a = ['w', 'y', 'k', 'o', 'p'];

var iterator = a.values();

console.log(iterator.next().value); // w

console.log(iterator.next().value); // y

console.log(iterator.next().value); // k

console.log(iterator.next().value); // o

console.log(iterator.next().value); // p

##### array.findIndex

**findIndex()**方法返回数组中满足提供的测试函数的第一个元素的**索引**。否则返回-1。

##### Array.from

**Array.from()**方法从一个类似数组或可迭代对象中创建一个新的数组实例。

##### array.includes

**includes()**方法用来判断一个数组是否包含一个指定的值，根据情况返回 true否false。

##### array.indexOf

**indexOf()**方法返回在数组中可以找到一个给定元素的第一个索引，如果不存在，则返回

-1。

##### array.lastIndexOf

**lastIndexOf()**方法返回指定元素（也即有效的 JavaScript 值或变量）在数组中的最后一个的索引，如果不存在则返回 -1。从数组的后面向前查找，从fromIndex处开始

##### Array.isArray

**Array.isArray()**用于确定传递的值是否是一个Array。

##### array.join

**join()**方法将一个数组（或一个[类数组对象](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Indexed_collections" \l "Working_with_array-like_objects" \t "_blank)）的所有元素连接成一个字符串并返回这个字符串。

##### Array.of

**Array.of()**方法创建一个具有可变数量参数的新数组实例，而不考虑参数的数量或类型。

**Array.of()** 和 **Array** 构造函数之间的区别在于处理整数参数：**Array.of(7)**创建一个具有单个元素**7**的数组，而 **Array(7)**创建一个包含**7**个**undefined**元素的数组

##### array.pop

**pop()**方法从数组中**删除**最后一个**元素**，并**返回**该元素的**值**。此方法**更改**数组的**长度**。

##### array.shift

**shift()**方法从数组中**删除**第一个元素，并返回该元素的值。此方法更改数组的长度。

##### array.unshift

**unshift()**方法将一个或多个元素添加到数组的开头，并返回新数组的长度。

##### array.push

**push()** 方法将一个或多个元素添加到数组的末尾，并返回新数组的长度。

##### array.slice

**slice()** 方法返回一个从开始到结束（**不包括结束**）选择的数组的一部分**浅拷贝**到一个新数组对象。原始数组不会被修改。

##### array.splice

**splice()**方法通过删除现有元素和/或添加新元素来更改一个数组的内容。

##### array.sort

**sort()**方法用[就地（ in-place ）的算法](https://en.wikipedia.org/wiki/Sorting_algorithm" \l "Stability" \t "_blank)对数组的元素进行排序，并返回数组。 sort 排序不一定是[稳定的](https://zh.wikipedia.org/wiki/%E6%8E%92%E5%BA%8F%E7%AE%97%E6%B3%95" \l ".E7.A9.A9.E5.AE.9A.E6.80.A7" \t "_blank)。默认排序顺序是根据字符串Unicode码点

##### array.toLocaleString

**toLocaleString()**返回一个字符串表示数组中的元素。数组中的元素将使用各自的 toLocaleString方法转成字符串，这些字符串将使用一个特定语言环境的字符串（例如一个逗号 ","）隔开。

语法：

arr.toLocaleString();

arr.toLocaleString(locales);

arr.toLocaleString(locales, options);

##### array.toString

**toString()**返回一个字符串，表示指定的数组及其元素。

#### 对象

##### For…in

##### Object.keys(obj)

##### Object.values

Object.values()方法返回一个给定对象的所有可枚举属性值的数组

##### Object.entries

Object.entries()方法返回一个给定对象自身可遍历属性 [key,value] 的数组

##### Object.fromEntries

将二维数组或者map转换成对象；

##### Object.getOwnPropertyNames

Object.getOwnPropertyNames()方法返回直接在给定对象上找到的所有属性(**或不**属性)的数组.[enumerable](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/propertyIsEnumerable)

* 1. **获取对象的原型对象上的所有自有属性和方法**
  2. **只获取对象的原型对象上的不可枚举属性**

var str = 'abcde'; // 字符串类型

// 获取所有自有属性和方法，str会被强制转为new String('adcde'); ES5下会报错

var properties = Object.getOwnPropertyNames(str); // ["0", "1", "2", "3", "4", "length"]

// 获取所有可枚举的属性和方法

var keys = Object.keys(str); // ["0", "1", "2", "3", "4"]

// 过滤得到不可枚举的属性和方法

var no\_enumerate = properties.filter(function (value) {

if (keys.indexOf(value) == -1) {

return true;

}

return false;

});

no\_enumerate; // ["length"]

##### Object.getOwnPropertySymbols

Object.getOwnPropertySymbols()方法返回[symbol](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Symbol)直接在给定对象上找到的所有属性的数组.

##### Reflect.ownKeys

[Reflect.ownKeys(obj)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Reflect/ownKeys) 返回相当于:

**Object**.getOwnPropertyNames(**target**).

**concat**(**Object**.getOwnPropertySymbols(**target**))

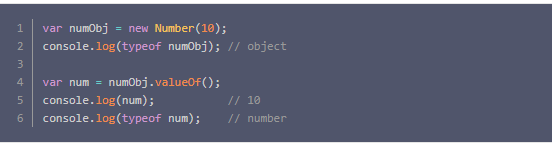
##### Object.getOwnPropertyDescriptors

该方法返回指定对象所有自身属性的描述对象

### 基础数据类型

#### Number

* 1. Number.isFinite() 用来检查一个数值是否为有限的
  2. Number.isNaN() 用来检查一个值是否为 NaN
  3. Number.parseInt() 方法可以根据给定的进制数把一个字符串解析成整数。
  4. Number.parseFloat() 方法可以把一个字符串解析成浮点数
  5. **toFixed()**方法使用定点表示法来格式化一个数
  6. **toLocaleString()** 方法返回这个数字在特定语言环境下的表示字符串。
  7. **toPrecision()**方法以指定的精度返回该数值对象的字符串表示
  8. **toString()**方法返回指定[Number](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Number" \t "_blank) 对象的字符串表示形式。
  9. **valueOf()**方法返回一个被 [Number](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Number" \t "_blank)对象包装的原始值。



* 1. Math.trunc 用于去除一个数的小数部分，返回整数部分。
  2. Number.isInteger() 用来判断一个数值是否为整数

#### String

1. **charAt()**方法从一个字符串中返回指定的字符。
2. **codePointAt()**方法返回 一个 Unicode 编码点值的非负整数。
3. **charCodeAt()** 方法返回0到65535之间的整数，表示给定索引处的UTF-16代码单元 (在 Unicode 编码单元表示一个单一的 UTF-16 编码单元的情况下，UTF-16 编码单元匹配 Unicode 编码单元。但在——例如 Unicode 编码单元 > 0x10000 的这种——不能被一个 UTF-16 编码单元单独表示的情况下，只能匹配 Unicode 代理对的第一个编码单元) 。如果你想要整个代码点的值，使用 **codePointAt**()。
4. **concat()**方法将一个或多个字符串与原字符串连接合并，形成一个新的字符串并返回
5. **endsWith()**方法用来判断当前字符串是否是以另外一个给定的子字符串“结尾”的，根据判断结果返回 true 或 false。
6. **String.fromCharCode()** 方法返回使用指定的Unicode值序列创建的字符串。
7. String.fromCodePoint()方法返回使用指定序列的代码点创建的字符串。
8. **includes()** 方法用于判断一个字符串是否包含在另一个字符串中，根据情况返回true或false。
9. **indexOf()** 方法返回调用  [String](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/String" \t "_blank) 对象中第一次出现的指定值的索引，开始在 fromIndex进行搜索。如果未找到该值，则返回-1
10. **lastIndexOf()** 方法返回指定值在调用该方法的字符串中最后出现的位置，如果没找到则返回 -1。从该字符串的后面向前查找，从fromIndex处开始
11. **valueOf()**方法返回一个[String](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/String" \t "_blank)对象的原始值（primitive value）
12. **toUpperCase()** 将调用该方法的字符串值转换为大写形式，并返回
13. **toString()**方法返回指定对象的字符串形式。

#### Symbol

##### Symbol.prototype.description

获取Symbol的描述字符串；

### 变量

#### Const、 var、 let 区别

const定义的变量不可以修改，只能在块作用域里访问，而且必须初始化。

var定义的变量可以修改 ， 可以跨块访问, 不能跨函数访问（变量提升），如果不初始化会输出undefined，不会报错。

1. let 声明的变量具有块作用域的特征。
2. 在同一个块级作用域，不能重复声明变量。
3. let 声明的变量不存在变量提升，换一种说法，就是 let 声明存在**暂时性死区**（TDZ）。

**总结：**

1. var 声明的变量属于函数作用域，let 和 const 声明的变量属于块级作用域；
2. var 存在变量提升现象，而 let 和 const 没有此类现象；
3. var 变量可以重复声明，而在同一个块级作用域，let 变量不能重新声明，const 变量不能修改。

### 普通函数和箭头函数的区别

#### 箭头函数

**1、相当于匿名函数， 不能作为构造函数、不能使用new**

1. 箭头函数不绑定arguments，取而代之用rest参数...解决

3、箭头函数不绑定this，会捕获其所在的上下文的this值，作为自己的this值

4、箭头函数通过 call()  或   apply() 方法调用一个函数时，只传入了一个参数，对 this 并没有影响。

1. 箭头函数没有原型属性

* 箭头函数的 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()
* 普通函数的this指向调用它的那个对象

#### 总结

### 构造函数

#### Super

class继承中，子类必须在constructor方法中调用super方法，否则新建实例时会报错

子类自己的this对象，必须先通过父类的构造函数完成塑造，得到与父类同样的实例属性和方法，然后再对其进行加工，加上子类自己的实例属性和方法。如果不调用super方法，子类就得不到this对象。

##### 当做函数使用

super 虽然代表了父类 A 的构造函数，但是返回的是子类 B 的实例，即 super 内部的 this 指的是 B，因此 super() 在这里相当于 ```A.prototype.constructor.call(this, props)``。

##### 当做对象使用

在普通方法中，指向父类的原型对象；在静态方法中，指向父类。

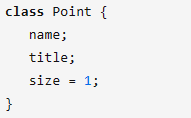
##### 注意

使用super时，必须显式指出是作为函数还是作为对象使用，否则会报错

### 新特性：

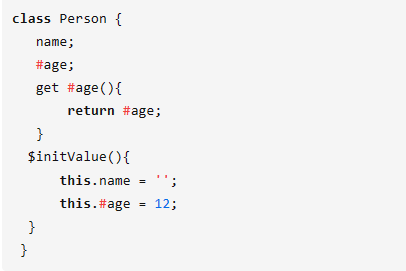
#### ES13（2022）

##### 类的声明字段class



##### 私有方法&字段

用#前缀来定义类的私有方法和字段。



##### 类的静态公共方法和字段

在之前的类的字段和私有方法提案的基础上，为JavaScript类增加了静态公共字段、静态私有方法和静态私有字段的特性

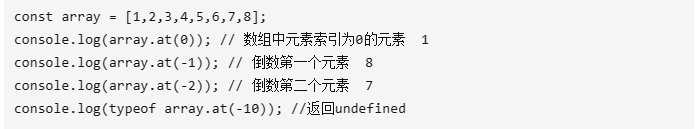


#### ES14（2023）

##### At()

at()方法是JavaScript中数组对象的一个新方法，它可以根据指定的索引返回数组中对应的元素。如果指定的索引超出数组范围，at()方法返回undefined

取出数组中特定顺序的元素



除了数组，字符串和 TypedArray 对象现在也有 at() 方法

##### Object.hasOwn

如果指定的对象自身有指定的属性，则静态方法Object.hasOwn()返回true。如果属性是继承的或者不存在，该方法返回false

##### 模块顶层使用await

在 JavaScript 中，await 关键字用于暂停异步函数的执行，直到一个 Promise 被解决或拒绝

只能在异步函数内部使用，而不能在普通函数或全局作用域中使用

在 ES13 之后，我们可以在模块的顶层使用 await

##### 使用“#”声明私有属性

以前我们用“\_”来表示私有属性，但是不安全，仍然有可能被外部修改。在 JavaScript 中，使用 "#" 符号声明私有属性是一种相对较新的特性，被称为“私有字段”（Private Fields）。它是在 ECMAScript 2019 标准中引入的，主要用于防止类的属性被意外修改或访问

##### toReversed()

数组取反、返回一个新的数组

Reverse 改变元素组不返回新数组

##### toSorted（）

数组排序返回一个新的数组

Sort 不返回新数组、改变原数组

##### toSpliced()

返回新的数组

Splice 修改原数组

##### Width

返回一个新的数组、讲给定索引替换为给定值

## Node面试题

#### Node 特性：

特征：单线程、事件驱动、非阻塞I/O

　　node 无法直接渲染静态页面，提供静态服务

　　node 没有根目录的概念

　　node 必须通过路由程序指定文件才能渲染文件

　　node 比其他服务端性能更好，速度更快

### 面试题

#### CommonJS中require/exports和ES6中import/export区别：

CommonJS模块的重要特性是加载时执行，及脚本代码在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，还没有执行的部分是不输出的

　　ES6模块是动态引用，如果使用import从一个模块加载变量，那些变量不会缓存，而是成为一个指向被加载模块的引用,impor/export最终都是编译为require/exports来执行的

#### 使用npm有哪些好处

通过NPM，你可以安装和管理项目的依赖，并且能够指明依赖项的具体版本号，可以通过package.json文件来管理项目信息，配置脚本

#### AMD 和CMD的区别

CommonJS和AMD都是JavaScript模块化规范

　　CMD依赖就近，而AMD依赖前置

　　CMD是延迟执行的，而AMD是提前执行的

　　AMD的API默认是一个当多个用，CMD的API严格区分，推崇职责单一

#### 如何判断当前脚本运行在node 和浏览器

通过判断 Global 对象是否为 window ，如果不为window ，当前脚本没有运行在浏览器中

#### 简述同步异步，如何避免地狱回调

同步方法调用一旦开始，调用者必须等到方法调用返回后，才能继续后续的行为

异步方法调用一旦开始，方法调用就会立即返回，调用者就可以继续后续的操作。而异步方法通常会在另外一个线程中，整个过程，不会阻碍调用者的工作

　避免回调地狱：

　　1）Promise

　　2）async/await

　　3）generator

　　4）事件发布/监听模式

#### 几种模块化规范

CommonJS规范主要用于服务端编程，加载模块是同步的，这并不适合在浏览器环境，因为同步意味着阻塞加载，浏览器资源是异步加载的

AMD规范在浏览器环境中异步加载模块，而且可以并行加载多个模块。不过，AMD规范开发成本高，代码的阅读和书写比较困难

CMD规范与AMD规范很相似，都用于浏览器编程，依赖就近，延迟执行，可以很容易在Node.js中运行（依赖SPM 打包，模块的加载逻辑偏重）

ES6 在语言标准的层面上，实现了模块功能，而且实现得相当简单，完全可以取代 CommonJS 和 AMD 规范，成为浏览器和服务器通用的模块解决方案

#### **app.use和app.get区别**

app.use(path,callback)中的callback既可以是router(路由)对象又可以是函数

　　app.get(path,callback)中的callback只能是函数

#### HTTP 2.0

* 1、HTTP/2引入了“服务端推（server push）”的概念，它允许服务端在客户端需要数据之前就主动地将数据发送到客户端缓存中，从而提高性能。
* 2、HTTP/2提供更多的加密支持
* 3、HTTP/2使用多路技术，允许多个消息在一个连接上同时进行。

4、它增加了头压缩（header compression），因此即使非常小的请求，其请求和响应的header都只会占用很小比例的带宽

#### TCP 和UDP的区别

* 1、TCP（Transmission Control Protocol，传输控制协议）是基于连接的协议，也就是说，在正式收发数据前，必须和对方建立可靠的连接。一个TCP连接必须要经过三次“对话”才能建立起来

2、UDP（User Data Protocol，用户数据报协议）是与TCP相对应的协议。它是面向非连接的协议，它不与对方建立连接，而是直接就把数据包发送过去！ UDP适用于一次只传送少量数据、对可靠性要求不高的应用环境

## Webpack

### 详解

Webpack是一个打包模块化 javascript 的工具，它会从 main.js 出发，识别出源码中的模块化导入语句，递归地找出入口文件的所有依赖，将入口和其所有依赖打包到一个单独的文件中。

### 原理：

（1）entry：一个可执行模块或者库的入口。

（2）chunk：多个文件组成一个代码块。可以将可执行的模块和他所依赖的模块组合成一个chunk，这是打包。

（3）loader：文件转换器。例如把es6转为es5，scss转为css等

（4）plugin：扩展webpack功能的插件。在webpack构建的生命周期节点上加入扩展hook，添加功能。

#### Loader 和 Plugin 的区别：

loader：模块转换器，用于把模块原内容按照需求转换成新内容  
　　通过使用不同的Loader，Webpack可以要把不同的文件都转成JS文件,比如CSS、ES6/7、JSX等

　　plugin：扩展插件  
　　在 Webpack 构建流程中的特定时机注入扩展逻辑来改变构建结果或做你想要的事情  
　　一个插件是含有apply方法的一个对象，通过这个方法可以参与到整个webpack打包的各个流程

　　loader是用来对模块的源代码进行转换,而插件目的在于解决 loader 无法实现的其他事  
　　因为plugin可以在任何阶段调用,能够跨Loader进一步加工Loader的输出

#### 常见的loader 和 plugin：

plugin:

　　html-webpack-plugin 为html文件中引入的外部资源，可以生成创建html入口文件

　　mini-css-extract-plugin 分离css文件

　　clean-webpack-plugin 删除打包文件

　　HotModuleReplacementPlugin 热更新应用

　　copy-webpack-plugin 拷贝静态文件

　　terser-webpack-plugin 通过TerserPlugin压缩ES6代码

loader:

　　css-loader 加载 CSS，支持模块化、压缩、文件导入等特性

　　image-loader 加载并且压缩图片文件

　　babel-loader 把 ES6 转换成 ES5

　　style-loader 把 CSS 代码注入到 JavaScript 中，通过 DOM 操作去加载 CSS

　　file-loader 把文件输出到一个文件夹中，在代码中通过相对 URL 去引用输出的文件

source-map-loader 加载额外的 Source Map 文件，以方便断点调试

#### 长缓存

浏览器在用户访问页面的时候，为了加快加载速度会对用户访问的静态资源进行存储，但是每一次代码升级或更新都需要浏览器下载新的代码，最简单方便的方式就是引入新的文件名称

　　webpack中可以在output中指定chunkhash，并且分离经常更新的代码和框架代码。通过NameModulesPlugin或HashedModuleIdsPlugin使再次打包文件名不变

### 流程

（1）初始化参数：解析webpack配置参数，合并shell传入和webpack.config.js文件配置的参数，形成最后的配置结果。

（2）开始编译：上一步得到的参数初始化compiler对象，注册所有配置的插件，插件监听webpack构建生命周期的事件节点，做出相应的反应，执行对象的 run 方法开始执行编译。

（3）确定入口：从配置的entry入口，开始解析文件构建AST语法树，找出依赖，递归下去。

（4）编译模块：递归中根据文件类型和loader配置，调用所有配置的loader对文件进行转换，再找出该模块依赖的模块，再递归本步骤直到所有入口依赖的文件都经过了本步骤的处理。

（5）完成模块编译并输出：递归完事后，得到每个文件结果，包含每个模块以及他们之间的依赖关系，根据entry配置生成代码块chunk。

（6）输出完成：输出所有的chunk到文件系统。

### 扩展插件：

1、babel-loader把es6转为es5；

2、file-loader把文件替换成对应的url；

3、raw-loader注入文本文件内容到代码中。

## 小程序总结

### 参数传值

* 1. 给HTML元素添加data-\*属性来传递我们需要的值，然后通过e.currentTarget.dataset或onload的param参数获取。但data-名称不能有大写字母和不可以存放对象
  2. 设置id 的方法标识来传值通过e.currentTarget.id获取设置的id的值,然后通过设置全局对象的方式来传递数值
  3. 在navigator中添加参数传值（？传的值的名称=所传的值在onLoad(option)用option来接收）

### 小程序运行优化

#### 提高页面加载速度

#### 用户行为预测

#### 减少默认data的大小

#### 组件化方案

### 小程序与原生app的区别

#### 小程序

##### 优点：

1、基于微信平台开发，享受微信本身自带的流量，这个是最大的优势

2、无需安装，只要打开微信就能用，不占用用户手机内存，体验好

3、开发周期短，一般最多一个月可以上线完成

4、开发所需的资金少，所需资金是开发原生APP一半不到

5、小程序名称是唯一性的，在微信的搜索里权重很高

6、容易上手，只要之前有HTML+CSS+JS基础知识，写小程序基本上没有大问题；当然如果了解ES6+CSS3则完全可以编写出即精简又动感的小程序；

7、基本上不需要考虑兼容性问题，只要微信可以正常运行的机器，就可以运行小程序

发布、审核高效，基本上上午发布审核，下午就审核通过，

8、升级简单，而且支持灰度发布

9、开发文档比较完善，开发社区比较活跃

10、最近刚开放的牛x功能，新增webview组件，可以展示网页啦，这个比较爽；

11、支持插件式开发，一些基本功能可以开发成插件，供多个小程序调用

##### 缺点：

* 1. 局限性很强，（比如页面大小不能超过1M。不能打开超过5个层级的页面。样式单一。小程序的部分组件已经是成型的了，样式不可以修改。例如：幻灯片、导航。）只能依赖于微信依托于微信，无法开发后台管理功能
  2. 不利于推广推广面窄，不能分享朋友圈，只能通过分享给朋友，附近小程序推广。其中附近小程序也受到微信的限制
  3. 后台调试麻烦，因为API接口必须https请求，且公网地址，也就是说后台代码必须发布到远程服务器上；当然我们可以修改host进行dns映射把远程服务器转到本地，或者开启tomcat远程调试；不管怎么说终归调试比较麻烦
  4. 后台调试麻烦，因为API接口必须https请求，且公网地址，也就是说后台代码必须发布到远程服务器上；当然我们可以修改host进行dns映射把远程服务器转到本地，或者开启tomcat远程调试；不管怎么说终归调试比较麻烦
  5. js引用只能使用绝对路径，很蛋疼；基于安全性及MINA框架实现原理，小程序中对js使用做了很多限制，不能使用：new Function，eval，Generator，不能操作cookie，不能操作DOM；

#### 原生app：

##### 优点：

1. 原生的响应速度快
2. 对于有无网络操作时，譬如离线操作基本选用原生开发
3. 需要调用系统硬件的功能（摄像头、方向传感器、重力传感器、拨号、GPS、语音、短信、蓝牙等功能）
4. 在无网络或者若网的情况下体验好

##### 缺点

1. 开发周期长，开发成本高
2. 需要下载

#### 区别

* 1. 运行环境不同
  2. 开发成本
  3. 系统权限获取
  4. 生产环境运行流畅度不同

## HTTP

### 前景提要

#### HTTP 是站在TCP之上的

HTTP协议的瓶颈及其优化技巧都是基于TCP协议本身的特性

#### 请求延迟

影响网络请求的因素主要有：带宽（今天的带宽已经极大提升）、延迟（今天影响请求的主要原因）。

### http版本迭代

#### HTTP1.0

http1.0 被抱怨的两大缺点：连接无法复用、head of line blocking

http1.0的实现：

第一个请求没有收到返回之前，后续应用层请求只能排队

http1.0 复用：

通过设置connection：keep-alive 可以在一定时间内复用连接、具体的时间可以由服务器控制

（http1.0 connection的默认值就是keep-alive）

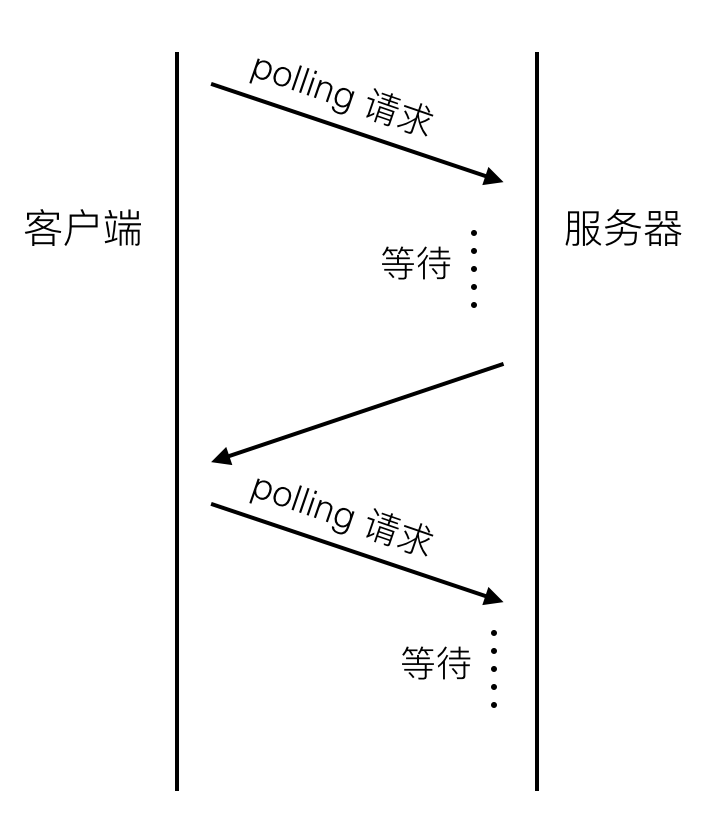
一段时间内的连接复用对PC端浏览器的体验帮助很大，因为大部分的请求在集中在一小段时间以内。但对移动app来说，成效不大，app端的请求比较分散且时间跨度相对较大。所以移动端app一般会从应用层寻求其它解决方案，长连接方案或者伪长连接方案：

##### 解决连接无法复用

方案一：基于TCP的长链接：

现在越来越多的移动端app都会建立一条自己的长链接通道，通道的实现是基于tcp协议。基于tcp的socket编程技术难度相对复杂很多，而且需要自己制定协议，但带来的回报也很大。信息的上报和推送变得更及时，在请求量爆发的时间点还能减轻服务器压力（http短连接模式会频繁的创建和销毁连接）

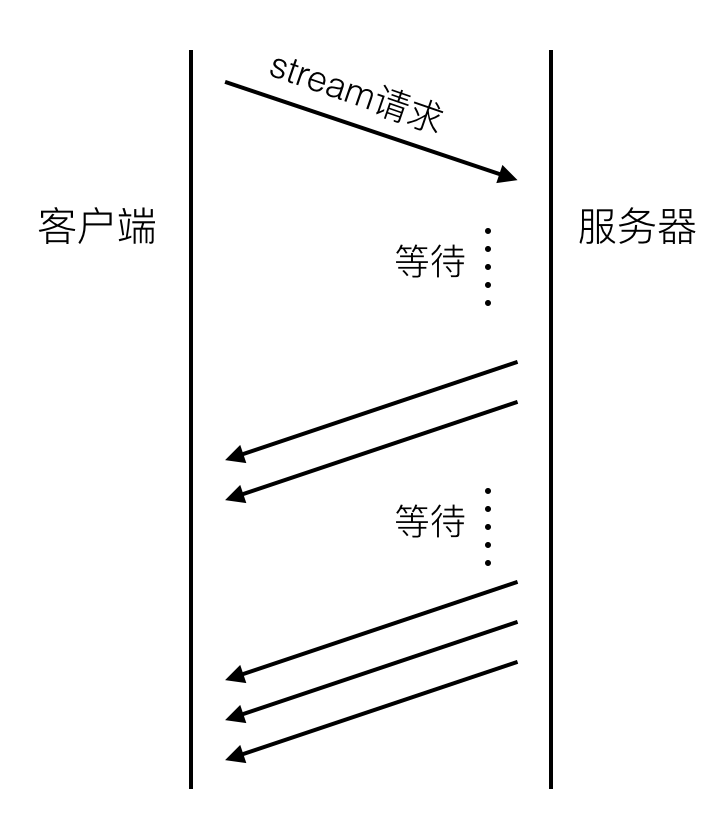
方案二：http long-polling



1. 和传统的http短链接相比，长连接会在用户增长的时候极大的增加服务器压力
2. 移动端网络环境复杂，像wifi和4g的网络切换，进电梯导致网络临时断掉等，这些场景都需要考虑怎么重建健康的连接通道。
3. 这种polling的方式稳定性并不好，需要做好数据可靠性的保证，比如重发和ack机制。
4. polling的response有可能会被中间代理cache住，要处理好业务数据的过期机制。

注意：long-polling方式还有一些缺点是无法克服的，比如每次新的请求都会带上重复的header信息，还有数据通道是单向的，主动权掌握在server这边，客户端有新的业务请求的时候无法及时传送。

方案三：http streaming



与long-polling区别：

1. server并不会结束初始的streaming请求，而是持续的通过这个通道返回最新的业务数据
2. 数据通道单向的
3. 通过在server response的头部里增加”Transfer Encoding: chunked”来告诉客户端后续还会有新的数据到来
4. 业务数据无法按照请求来做分割，客户端没收到一块数据都需要自己做协议解析，也就是说要做自己的协议定制
5. 不会产生重复的header数据

方案四： web socket

1. WebSocket和传统的tcp socket连接相似，也是基于tcp协议，提供双向的数据通道
2. WebSocket优势在于提供了message的概念，比基于字节流的tcp socket使用更简单，同时又提供了传统的http所缺少的长连接功能

其他奇技淫巧：

Spriting(图片合并)

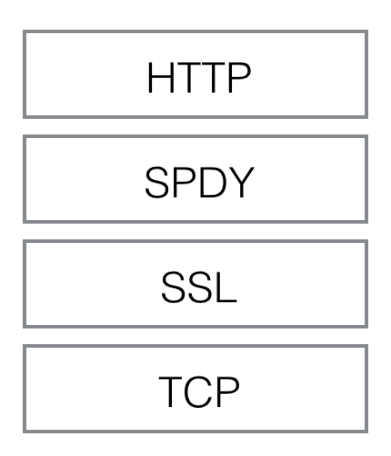
Inlining（内容内嵌）

Concatenation（文件合并：针对js）

Domain Sharding （域名分片）

###### SPDY 的目标

SPDY的目标在一开始就是瞄准http1.x的痛点，即延迟和安全性



SPDY位于HTTP之下，TCP和SSL之上，这样可以轻松兼容老版本的HTTP协议(将http1.x的内容封装成一种新的frame格式)，同时可以使用已有的SSL功能。SPDY的功能可以分为基础功能和高级功能两部分，基础功能默认启用，高级功能需要手动启用。

基础功能

多路复用（multiplexing）：多路复用通过多个请求stream共享一个tcp连接的方式，解决了http1.x holb（head of line blocking）的问题，降低了延迟同时提高了带宽的利用率。（路复用带来一个新的问题是，在连接共享的基础之上有可能会导致关键请求被阻塞）

请求优先级（request prioritization）：

SPDY允许给每个request设置优先级，重要的请求就会优先得到响应

SPDY 高级功能

server推送（server push）：

server通过X-Associated-Content header（X-开头的header都属于非标准的，自定义header）告知客户端会有新的内容推送过来。在用户第一次打开网站首页的时候，server将资源主动推送过来可以极大的提升用户体验

server暗示（server hint）：

和server push不同的是，server hint并不会主动推送内容，只是告诉有新的内容产生，内容的下载还是需要客户端主动发起请求。server hint通过X-Subresources header来通知，一般应用场景是客户端需要先查询server状态，然后再下载资源，可以节约一次查询请求。

#### HTTP2.0

设计前提：

1. 客户端向server发送request这种基本模型不会变。
2. 老的scheme不会变，使用http://和https://的服务和应用不会要做任何更改，不会有http2://。
3. 使用http1.x的客户端和服务器可以无缝的通过代理方式转接到http2.0上。
4. 不识别http2.0的代理服务器可以将请求降级到http1.x。

##### HTTP2.0 主要优化点

###### 新的二进制格式（Binary Format）

http1.x 明文协议（需要协议解析）：

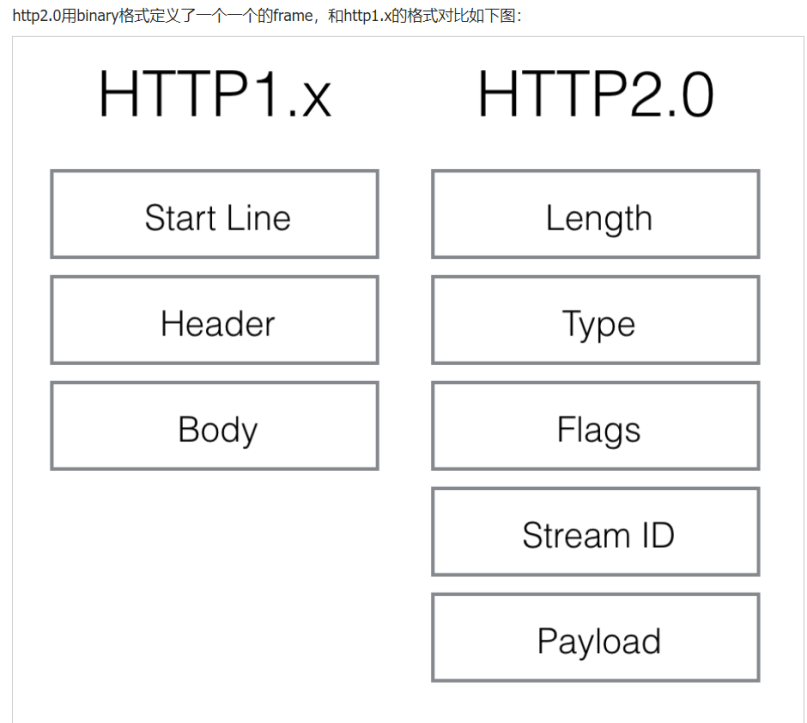
start line（request line或者status line）

header

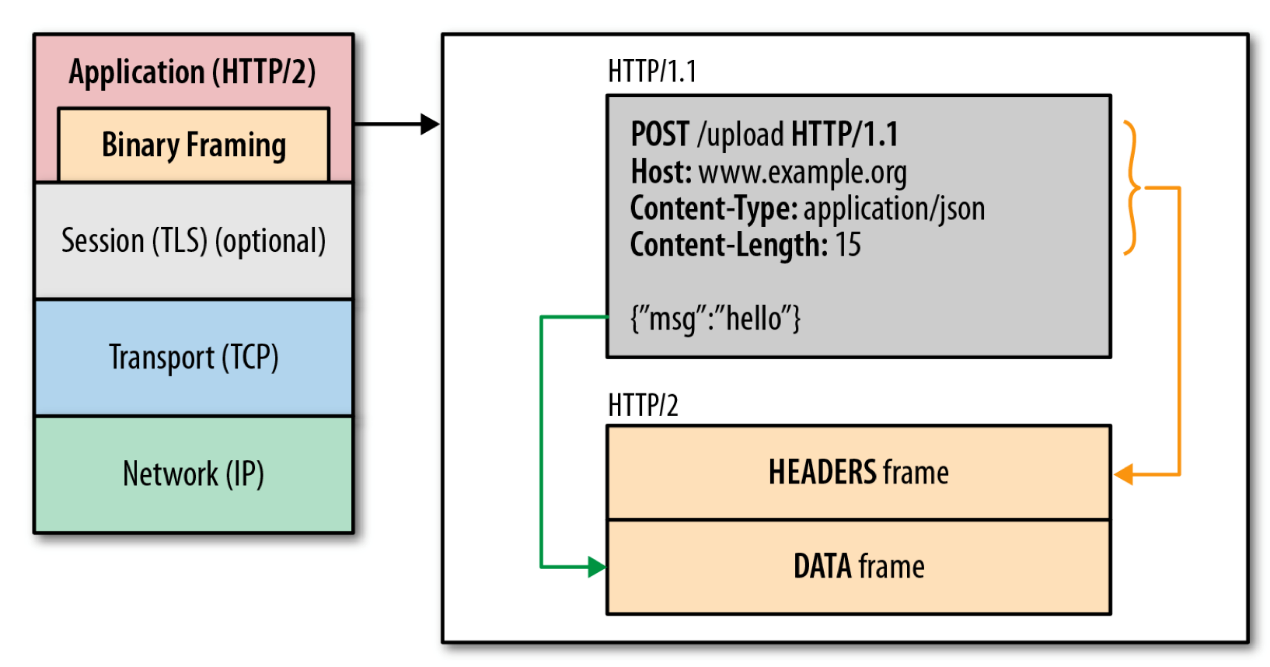
body

注意： http1.x 解析基于文本

基于文本协议的格式解析存在天然缺陷，文本的表现形式有多样性，要做到健壮性考虑的场景必然很多，二进制则不同，只认0和1的组合。基于这种考虑http2.0的协议解析决定采用二进制格式，实现方便且健壮。



虽然看上去协议的格式和http1.x完全不同了，实际上http2.0并没有改变http1.x的语义，只是把原来http1.x的header和body部分用frame重新封装了一层而已。调试的时候浏览器甚至会把http2.0的frame自动还原成http1.x的格式，具体的协议关系可以用下图表示：



###### 连接共享

http2.0要解决的一大难题就是多路复用（MultiPlexing），即连接共享

上面协议解析中提到的stream id就是用作连接共享机制的。一个request对应一个stream并分配一个id，这样一个连接上可以有多个stream，每个stream的frame可以随机的混杂在一起，接收方可以根据stream id将frame再归属到各自不同的request里面。

###### Header压缩

前面提到过http1.x的header由于cookie和user agent很容易膨胀，而且每次都要重复发送。http2.0使用encoder来减少需要传输的header大小，通讯双方各自cache一份header fields表，既避免了重复header的传输，又减小了需要传输的大小。高效的压缩算法可以很大的压缩header，减少发送包的数量从而降低延迟。

###### 压缩算法选择

SPDY/2使用的是gZIP压缩算法，但后来出现的两种攻击方式BREACH和CRIME使得即使走ssl的SPDY也可以被破解内容，最后综合考虑采用的是一种叫HPACK的压缩算法。这两个漏洞和相关算法可以点击链接查看更多的细节，不过这种漏洞主要存在于浏览器端，因为需要通过javascript来注入内容并观察payload的变化。

###### 重置连接表现更好

很多app客户端都有取消图片下载的功能场景，对于http1.x来说，是通过设置tcp segment里的reset flag来通知对端关闭连接的。这种方式会直接断开连接，下次再发请求就必须重新建立连接。http2.0引入RST\_STREAM类型的frame，可以在不断开连接的前提下取消某个request的stream，表现更好。

###### Server Push

http2.0能通过push的方式将客户端需要的内容预先推送过去，所以也叫“cache push”。另外有一点值得注意的是，客户端如果退出某个业务场景，出于流量或者其它因素需要取消server push，也可以通过发送RST\_STREAM类型的frame来做到。

###### 流量控制（Flow Control）

TCP协议通过sliding window的算法来做流量控制。发送方有个sending window，接收方有receive window。http2.0的flow control是类似receive window的做法，数据的接收方通过告知对方自己的flow window大小表明自己还能接收多少数据。只有Data类型的frame才有flow control的功能。对于flow control，如果接收方在flow window为零的情况下依然更多的frame，则会返回block类型的frame，这张场景一般表明http2.0的部署出了问题。

###### Nagle Algorithm vs TCP Delayed Ack

tcp协议优化的一个经典场景是：Nagle算法和Berkeley的delayed ack算法的对立。http2.0并没有对tcp层做任何修改，所以这种对立导致的高延迟问题依然存在。要么通过TCP\_NODELAY禁用Nagle算法，要么通过TCP\_QUICKACK禁用delayed ack算法。貌似http2.0官方建议是设置TCP\_NODELAY。

###### 更安全的SSL

HTTP2.0使用了tls的拓展ALPN来做协议升级，除此之外加密这块还有一个改动，HTTP2.0对tls的安全性做了近一步加强，通过黑名单机制禁用了几百种不再安全的加密算法，一些加密算法可能还在被继续使用。如果在ssl协商过程当中，客户端和server的cipher suite没有交集，直接就会导致协商失败，从而请求失败。在server端部署http2.0的时候要特别注意这一点。

### Web安全防护

#### SQL注入

原理：通过SQL命令插入web表单递交或者输入域名或者页面请求的查询字符串，最终达到欺骗服务器执行恶意的SQL命令的目的。

防护措施：

1、永远不要信任用户的输入，要对用户的输入进行校验，可以通过正则表达式，或限制长度，对单引号双“--”进行转换等。

2、永远不要使用动态拼装sql，可以使用参数化的sql或者直接使用存储过程进行数据查询存取。

3、永远不要使用管理员权限进行数据库连接，为每个应用使用单独的权限有限的数据库连接。

4、不要把机密信息明文存放，请加密或者hash掉密码和敏感的信息。

#### XSS 原理及防范

XSS（cross-site scripting） 攻击指的是攻击者往web 页面插入恶意html标签或者JavaScript 代码。比如：攻击者在qq 中发送一个看似安全的链接，骗取用户点击之后，窃取cookie中的私密信息；或者攻击者在论坛中加一个恶意表单，当用户提交表单的时候，却把信息发送到攻击者的服务器中，而不是用户原本信任的站点。

防范：

1. 输入时对符号过滤（‘< > ; ‘’）
2. 页面显示内容加密（encode）
3. 避免cookie中存放私密信息、密码等
4. Cookie绑定ip系统
5. 不需要对cookie操作的，可以在set-cookie 末尾加上 httpOnly 防止JavaScript 直接获取cookie
6. 尽量采用post 提交表单数据

#### XXS 与CSRF区别

XSS 是获取信息，不需要提前知道其他用户页面和数据包。

CSRF是代替用户完成指定动作，需要知道页面代码和数据包。

CSRF攻击步骤：

1. 登录收信人网站A，并在本地生成cookie
2. 在不登出A的情况下，访问危险网站B

##### CSRF防范

在客户端页面增加伪随机数

### HTTP缓存

#### 含义

当浏览器访问一个网站的时候，如果是第一次访问，就需要加载各种资源，比如html、js、css、img等，我们之后再访问网站的时候，就不需要加载各种资源，可以通过HTTP缓存策略来缓存一些资源，这样浏览器就不需要重新从服务器获取，可以直接从缓存中获取，从而提高网站的访问速度。

##### 强缓存

当浏览器第一次访问网站的时候，就会向服务器发送请求，服务器会发送一些资源。

如果服务器觉得这些资源应该被缓存下来，就会在响应头设置Cache-Control:max-age=(单位是秒)。 这样浏览器就会在本地缓存中存下资源文件，下一次重新请求资源文件的时候，浏览器就会检查这个max-age有没有过期，没有过期就直接从缓存中获取，过期了就要重新请求接口获取资源。

如果服务器觉得这些资源不应该被缓存下来，那就设置Cache-Control:no-cache，这样就不会被缓存下来。

##### 协商缓存

当浏览器访问网站的时候，服务器就会返回资源和资源标识，浏览器可以把资源和资源标识缓存到浏览器，当后续再次访问网站的时候，浏览器会把请求和资源标识一起发给服务器，这样服务器就会根据资源标识判断当前本地缓存的版本是否跟服务器的版本一致：

如果一致，服务器就会返回304状态码，浏览器就可以直接从缓存中拿资源文件。

如果不一致，服务器就会返回200状态码，同时会把新的资源和资源标识一起返回浏览器，浏览器再进行重新缓存。

**资源标识**

1. **Last-Modified**

**上次修改时间**

**浏览器下次请求的时候就会将资源标识（if-Modified-Since）这个值（就是上一次从服务器获得的Last-Modified值）放在请求头里发送给服务器，服务器会将资源标识（Last-Modified）放在响应头中返回，通过对比这两个修改时间来判断本地缓存的资源文件是否一致，从而选择从哪里获取资源。**

1. **Etag**

**资源文件唯一字符串**

**浏览器下次请求的时候就会将资源标识（if-None-Match）这个值（就是上一次从服务器获得的Etag值）放在请求头里发送给服务器，服务器会将资源标识（Etag）放在响应头中返回，通过对比这两个唯一字符串来判断本地缓存的资源文件是否一致，从而选择从哪里获取资源。**

#### 缓存解析流程

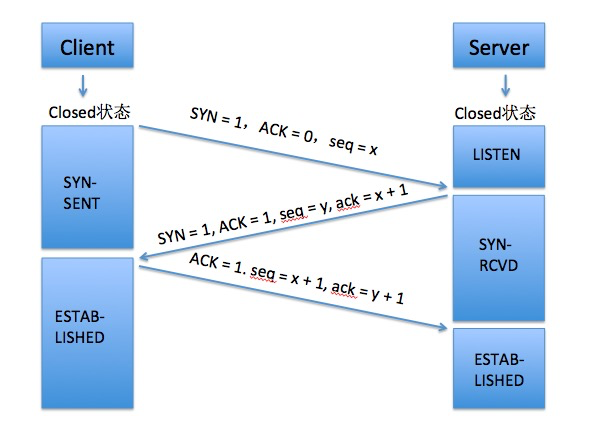
浏览器缓存分为强缓存和协商缓存，浏览器加载一个页面的简单流程如下

1. 浏览器先根据这个资源的http头信息来判断是否命中强缓存。如果命中则直接加在缓存中的资源，并不会将请求发送到服务器。（强缓存）
2. 如果未命中强缓存，则浏览器会将资源加载请求发送到服务器。服务器来判断浏览器本地缓存是否失效。若可以使用，则服务器并不会返回资源信息，浏览器继续从缓存加载资源。（协商缓存）
3. 如果未命中协商缓存，则服务器会将完整的资源返回给浏览器，浏览器加载新资源，并更新缓存。（新的请求）

### 三次握手

#### 简介：

TCP协议在传输数据的时候，客户端(Client)跟服务端(Server)会建立连接，然后把需要传输的文件进行分段，以及提供可靠传输和流量控制，在数据传输完成后，当前的会话也要断开连接，避免资源浪费。其实TCP的三次握手就是建立连接的过程，而四次挥手就是断开连接的过程。



**注意**：TCP协议还有一个特点就是面向字节流，它会把数据都分成一个个字节，然后进行分段传输，在分段传输的时候，每一段是由不同的字节序号组成的。

**图片中字段关键字介绍**

SYN：同步序列号，是用来建立连接的握手信号。

ack：确认序号，当ACK为1时，ack有效，当ACK为0时，ack无效。

seq：序号。

ACK：确认序号有效。

FIN: 结束标志，用来表示断开连接。

**总结**：最开始的时候，客户端和服务端都是Closed(关闭)状态，准备发送连接请求前，Server会进入LISTEN(监听)状态。

#### 三次握手的步骤：

第一次握手： 客户端(Client)会给服务端(Server)发送请求报文段，并指定同步序列号SYN = 1，ACK = 0， 初始序号为seq = x，(seq里面就是字节的序号)，同时TCP的客户端进程进入SYN-SENT(同步已发送)状态。

第二次握手： 服务端收到客户端发送的请求报文SYN后，会向客户端发送一个SYN报文作为应答，表示同意建立连接，同时指定了自己的SYN = 1， ACK = 1，还会向客户端发送seq = y，来表示自己的一个初始序号，同时也会告诉客户端下一次应该从哪开始发送的确认序号，由于客户端发送过来的初始序号seq = x, 所以确认序号ack = x + 1，这时，TCP的服务端进入SYN-RCVD(同步收到)状态。

第三次握手： 客户端收到服务端的确认报文之后，会再次向服务端发送确认信息，表示已经收到。所以ACK = 1, seq = x + 1, ack = y + 1。TCP建立连接，客户端和服务器进入ESTAB-LISTEND(已建立连接状态)状态。

## 开发问题总结

### Webview 和 iframe 总结

相同点：

1. 嵌入内容：WebView 和 iFrame 都用于在一个容器中嵌入其他网页或应用程序。WebView 用于在原生应用程序中显示网页内容，而 iFrame 则用于在当前网页中嵌入其他网页或文档。

2、提供浏览器功能：无论是在 WebView 中还是通过 iFrame，用户都可以访问到完整的浏览器功能，例如加载网页、点击链接、提交表单等。这使得嵌入的内容具有交互性，并且可以与用户进行实时的互动。

3、跨域访问：WebView 和 iFrame 都可以用于跨域访问其他域名下的内容。这意味着可以在一个页面中显示来自不同域的内容，扩展了页面所能呈现的内容范围

不同点：

1、定义：

* WebView：WebView 是一种用于在 Android 或 iOS 应用程序中显示网页内容的组件。它基于系统提供的浏览器引擎，并提供了完整的浏览器体验，包括地址栏、前进和后退按钮等。
* iFrame：iFrame 是一种 HTML 标签，可以在当前网页中嵌入其他网页或文档。它通常用于显示广告、嵌入第三方视频或音频、以及展示来自不同域名的内容

1. 安全性：

* WebView：由于 WebView 是直接使用系统提供的浏览器引擎来渲染网页，因此存在安全风险。如果 WebView 配置不当，可能会导致 XSS 攻击、跨站点脚本攻击等安全问题。
* iFrame：iFrame 也存在安全风险，因为它可以嵌入来自不同域名的内容。如果嵌入的内容存在恶意代码，可能会通过 iFrame 访问到当前网页的敏感信息。但是，iFrame 可以通过设置 sandbox 属性来限制其访问权限，从而提高安全性。

1. 可控性

* WebView：由于 WebView 是基于系统提供的浏览器引擎，因此开发者无法完全控制其行为。例如，WebView 无法禁用 JavaScript、CSS 或图片加载等功能。
* iFrame：iFrame 可以通过设置属性来完全控制其显示和行为。例如，可以设置 iFrame 的宽度、高度、边框等样式，还可以使用 JavaScript 控制 iFrame 中的内容

1. 性能：

* WebView：由于 WebView 是在原生应用程序中运行的，因此具有较高的性能。同时，WebView 可以使用本地缓存，从而提高网页的加载速度。
* iFrame：iFrame 的性能较差，因为它需要在当前网页中加载另一个网页，会导致额外的网络请求和页面渲染开销。