# Http协议（网络基础）

## HTTP和HTTPS

HTTP是基于TCP/IP的关于数据如何在万维网中如何通信的协议。

HTTP本身是不保存状态的，它不对之前的请求和响应的状态进行管理，无状态协议

## HTTP持久化和管线化

* 1. 历史原因

HTTP最初的版本中，每进行一次HTTP通信，就要断开一次TCP连接。

以当年的通信情况来看，因为都是一些小的文本传输，所以这样并没有多大问题。随着HTTP的普及，文档中包含的图片越来越多，比如，当用户浏览一个包含多张图片的HTML页面时，在发送访问HTML页面同时，也会发送包含该页面的其他资源，这样每次的请求都会造成无谓的TCP连接建立和断开，增加通信量的开销。

* 1. 解决方法

为解决上述问题，HTTP/1.1和部分HTTP/1.0增加了持久连接（HTTP Persistent Connections ）的方法，其特点是，**只要一方未明确提出断开连接，则另一方保持TCP连接状态**。

这样的话，就减少了TCP连接的重复建立连接和断开连接的开销，减轻服务器的负载，另外，减少开销的那部分时间，HTTP请求和响应能够更早的结束，这样，Web页面显示速度也就提高了。

* + 1. HTTP/1.0+ keep-alive连接

使用原则

* 在HTTP/1.0中未标准化，虽然有一部分服务器通过非标准手段实现的持久连接，但客户端不一定支持持久连接。
* 当客户端请求的资源在一台服务器上，**请求被串联执行时**，这种时延将会被进一步放大。
* 在HTTP/1.0的各种增强版本中，通讯双方默认是不使用持久化连接的。
* 当客户端请求中含有Connection: Keep-Alive首部，服务器响应中也有Connection: Keep-Alive首部时，双方才会成功建立持久连接。
  + 1. HTTP/1.1持久连接

HTTP/1.1在废弃Keep-Alive后，提出了一种持久连接（persistent connection）的改进型设计取代它。

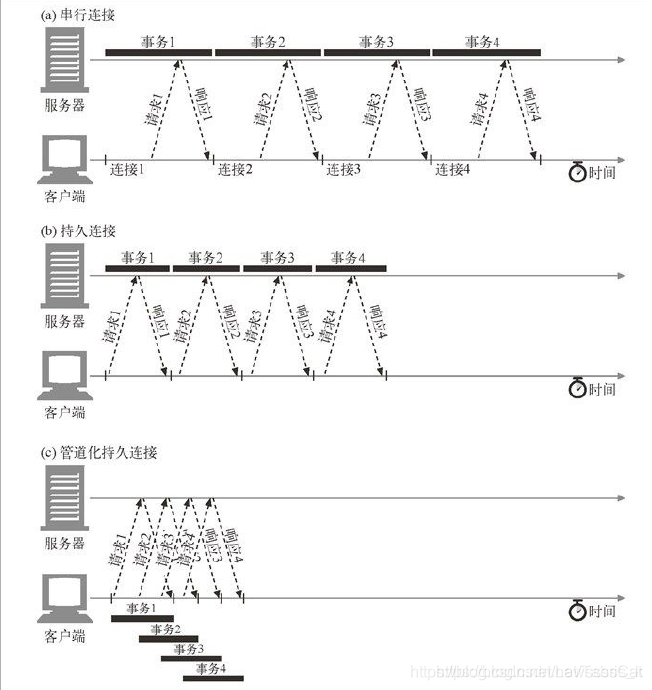
**所有的连接默认都是持久连接，包含服务器端和客户端。**除非特别指明，否则HTTP/1.1假定所有连接都是持久的。如果客户端需要事务处理完之后直接关闭TCP连接，需要显式地添加一个Connection: close首部。同样如果服务器返回的首部中没有Connection: close首部，客户端会认为连接仍维持在打开状态。

* + 1. 管道化连接

HTTP/1.1允许在持久连接上可选地使用请求管道。

在响应到达之前，可以将多余请求放入队列。

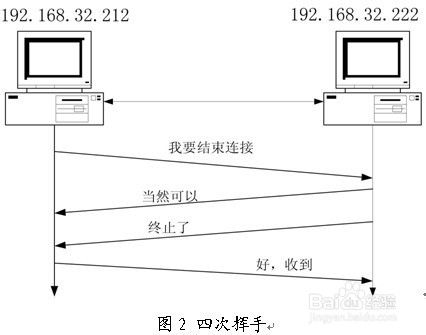
下图显示了持久化连接是怎样消除TCP连接时延，以及管道化请求是如何消除传输时延的。



* 1. HTTP的持久连接
* HTTP 1.1 规定了默认保持持久连接（HTTP Persistent Connection），数据传输完成了保持TCP连接不断开，等待在**同域名**下继续用这个通道传输数据。
* HTTP 1.1 版本支持持久连接 1.0版本不支持。

**与非持久连接的区别：**

持久连接使客户端到服务器端连接持续有效，避免了重新建立连接，大大减少了连接的建立以及关闭时延。HTTP连接是建立在TCP协议之上的，建立一条TCP连接需要三次握手，TCP连接关闭时需要四次挥手，这些都是需要时间的。



* 1. HTTP的管线化技术

管线化机制须通过永久连接（persistent connection）完成，仅HTTP/1.1支持此技术（HTTP/1.0不支持）。

管线化是指将**多个HTTP请求整批发送**，在发送过程中不用等待对方响应。

1. 在使用持久连接的情况下，普通情况下，某个连接消息的传递类似于：

请求1 -> 响应1 -> 请求2 -> 响应2

1. 而在管线化的方式下，某个连接上的消息变成了类似这样：

请求1 -> 请求2 -> 请求3 -> 响应1 -> 响应2 -> 响应3

管线化是在**持久连接**的基础上实现的，管线化的实现，能够同时并行发送多个请求，而不需要一个接一个的等待响应。并且**只有GET和HEAD请求可以进行管线化，而POST则有所限制**。此外，初次创建连接时也不应启动管线机制，因为对方（服务器）不一定支持HTTP/1.1版本的协议。

浏览器提交**批量的HTTP请求**可以缩短页面加载时间。此技术的关键在于HTTP的要求消息可以**同时塞入一个TCP分组中**，所以只提交一个分组即可同时发出多个要求。

例如，请求一个包含多张图片的HTML页面时，与逐一连接相比，用持久连接可以让请求更快结束，而管线化技术则更快，并且请求越多，时间差越明显。

* 1. HTTP的Cookie技术

1. 无状态协议的不保存浏览器状态

HTTP本身是不保存状态的，它不对之前的请求和响应的状态进行管理，就是说，无法根据之前的状态进行本次的请求处理。

假设要求登录认证的Web页面本身无法进行状态管理，不进行状态保存，那么每次跳转到新页面就要重新登录。

注：**无状态协议**的优点也是显而易见的，因为不用保存状态，自然可以减少服务器的CPU及内存资源的消耗。

1. 无状态协议的优化方案

引入了Cookie技术，Cookie技术通过在请求和响应报文中写入Cookie信息来控制客户端状态。

* Cookie会根据服务器发来的响应报文中一个叫做Set Cookie的首部字段信息通知客户端保存Cookie信息（该信息一般问客户端的身份信息等）。
* 当下次客户端再往该服务器上发送请求时，客户端会自动将Cookie信息加入请求报文中发送出去。
* 服务器端发现客户端发来的Cookie后，会去检查是哪一个客户端发来的Cookie,经对比服务器上的记录后，最后得到该客户端之前的状态信息。
  1. 持久连接和管线化的区别

持久连接的一个缺点是请求和响应式是顺序执行的，只有在请求1的响应收到之后，才会发送请求2，而管线化不需要等待上一次请求得到响应就可以进行下一次请求。实现并行发送请求。

只有GET和HEAD要求可以进行管线化，而POST则有所限制。

初次创建连接时也不应启动管线机制，因为对方（服务器）不一定支持HTTP/1.1版本的协议。

HTTP1.1要求服务器端支持管线化，但并不要求服务器端也对响应进行管线化处理，只是要求对于管线化的请求不失败，而且现在很多服务器端和代理程序对管线化的支持并不好，现代浏览器Chrome和Firefox默认并未开启管线化支持。

* 1. 设置HTTP连接方式



## 浏览器同域名请求的最大并发数限制

***：浏览器对同一 Host建立 TCP 连接到数量的限制***

前提：如果使用的是 HTTP/1.1 ，即没有多路传输，当浏览器需要加载有几十张图片的网页该怎么办呢？

肯定不能只开一个 TCP 连接顺序下载，浏览器的用户肯定不能接受。但是如果每个图片都开一个 TCP 连接发 HTTP 请求，那么电脑或者服务器都可能受不了。假设有 1000 张图片要下载，总不能开 1000 个TCP 连接去请求资源，即使用户的电脑硬件具有这个能力，服务器端也不定同意这样的请求方式。

因此，各大主流浏览器针对HTTP 1.1和HTTP 1.0协议都设置了相应的TCP连接个数限制。

在浏览网页的时候，有一个影响浏览速度的重要因素，就是浏览器的并发数量。并发数量通俗的理解就是，当浏览网页的时候同时工作的进行数量。

* HTTP客户端一般对同一个服务器的并发连接个数都是有限制的。
* 实际上，浏览器确实使用并行连接，但它们将并行连接的总数限制为少量（通常为四个）。服务器可以自由地关闭来自特定客户端的过多连接。
  1. 主流浏览器对HTTP 1.1和HTTP 1.0的最大并发连接数目

可以参考如下表格：

|  |  |  |
| --- | --- | --- |
| 浏览器 | HTTP / 1.1 | HTTP / 1.0 |
| IE 11 | 6 | 6 |
| IE 10 | 6 | 6 |
| IE 9 | 10 | 10 |
| IE 8 | 6 | 6 |
| 火狐 | 6 | 6 |
| Chrome 4+ | 6 | 6 |
| Opera 10.51+ | 8 | ？ |
|  |  |  |
| iPhone 4 | 4 | ？ |
| iphone 5 | 6 | ？ |
|  |  |  |
| Android2-4 | 4 | ？ |

* 1. 浏览器（客户端）与服务器之间的连接和断开的时机

TCP 连接与断开的问题，参考：nginx中http1.1 KeepAlive 失效性。

* Header 中 Connection 属性决定了连接是否持久；
* HTTP/1.0 中 Connection 默认是 keep-alive的，即每次请求都会重新建立和断开 TCP 连接；
* 而在 HTTP/1.1 中 Connection 默认是 持久化链接，即连接可以复用，不用每次都重新建立和断开 TCP 连接。**如果想主动断开连接，需要将客户端和服务器的 Connection 属性设置为 close ，即 Connection: close。**一般情况下复用的 TCP 连接在等待设置的超时时间之后还没有被任何连接使用的话，TCP 连接就会主动断开。
  1. HTTP/1.1 中一个TCP连接中对应多个 HTTP 请求
* HTTP/1.0 中，Connection 默认是 keep-alive，一个TCP包含一个HTTP请求；
* HTTP/1.1 中Connection 默认是 持久化连接，即连接可以复用，一个 TCP 连接是可以发送多个 HTTP 请求的
  1. HTTP/1.1 中一个TCP连接中不可以同时发送多个 HTTP 请求

HTTP/1.1 存在一个问题，单个 TCP 连接在**同一时刻**只能处理一个请求，即**两个请求的生命周期不能重叠**，任意两个 HTTP 请求从开始到结束的时间在同一个 TCP 连接里不能重叠。

* + 1. RFC 2616标准中 Pipelining规定

虽然 HTTP/1.1 规范中规定了 Pipelining 来试图解决这个问题，但是这个功能在浏览器中默认是关闭的。

一个支持持久连接的客户端可以在一个连接中发送多个请求（不需要等待任意请求的响应）。

收到请求的服务器必须按照请求收到的顺序发送响应。

**推测Pipelining这个设定的原因：**

由于 HTTP/1.1 是个文本协议，同时返回的内容也并不能区分对应于哪个发送的请求，所以顺序必须维持一致。

例如你向服务器发送了两个请求 GET/query?type=1 和 GET/query?type=2，服务器返回了两个结果，浏览器是没有办法根据响应结果来判断响应对应于哪一个请求的。

* + 1. HTTP2的Multiplexing多路传输特性优化Pipelining的问题

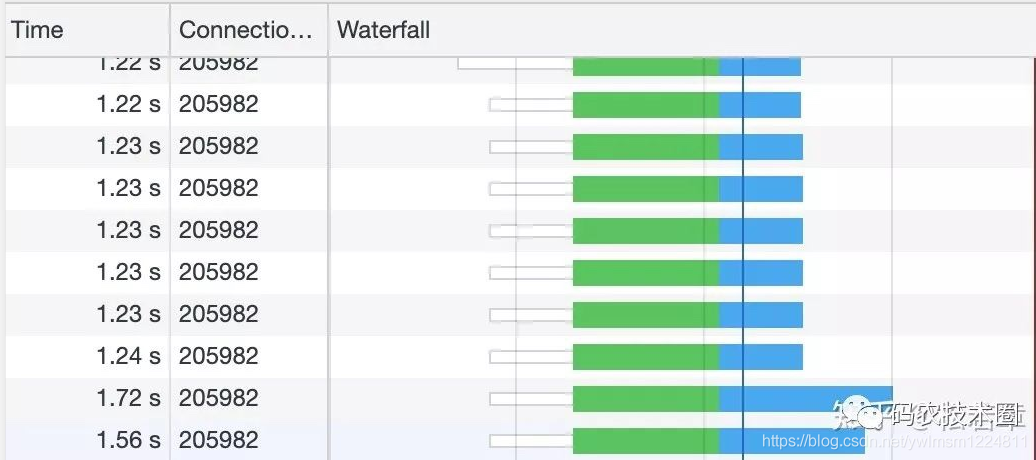
**虽然 Pipelining 设想看起来比较美好，但是在实践中会出现许多问题：**

* 一些代理服务器不能正确的处理 HTTP Pipelining；
* Head-of-line Blocking 连接头阻塞：在建立起一个 TCP 连接之后，假设客户端在这个连接连续向服务器发送了多个请求。如果按照标准的话，服务器应该按照收到请求的顺序返回结果，假设服务器在处理首个请求时花费了大量时间，那么后面所有的请求都需要等着首个请求结束才能响应，造成了阻塞。

基于 Pipelining 的两个问题，现代浏览器默认是不开启 HTTP Pipelining 的。

但是，HTTP2 提供了 Multiplexing 多路传输特性，可以在一个 TCP 连接中同时完成多个 HTTP 请求。

使用 HTTP2 的效果如下图：



绿色是发起请求到请求返回的等待时间，蓝色是响应的下载时间，可以看到都是在同一个 Connection，并行完成的。

1. **结论：**

* 在 HTTP/1.1 存在 Pipelining 技术可以完成这个多个请求同时发送，但是由于浏览器默认关闭，所以可以认为这是**不可行**的。
* 在 HTTP2 中由于 Multiplexing 特点的存在，多个 HTTP 请求可以在同一个 TCP 连接中**并行进行**。
  1. HTTP/1.1 中浏览器提高页面加载效率

主要有以下两点：

* **连接复用**，即维持和服务器已经建立的 TCP 连接，在同一连接上**顺序处理多个请求**
* 和服务器建立多个 TCP 连接
  1. 刷新页面，有时为什么不需要重新建立 SSL 连接

基于HTTP/1.1协议默认connection是持久化连接，前后端服务器中connection至少有一个未设置为close，则TCP不需要重复建立与断开，可以复用 TCP 连接的，刷新页面也不需要重新建立 SSL 连接。

* 1. 浏览器加载一个网页中的多张图片所建立的连接方式、顺序

首先判断根据http连接判断，图片资源是否在同一个域名下：

* 图片资源都是HTTPS 连接并且在同一个域名下，浏览器在SSL握手之后会和服务器商量能不能用 HTTP2，如果能的话就使用 Multiplexing 功能在这个连接上进行多路传输（一个 TCP 连接中同时完成多个 HTTP 请求）。不过也未必会所有挂在这个域名的资源都会使用一个 TCP 连接去获取，但是可以确定的是 Multiplexing 很可能会被用到。
* 如果用不了 HTTP2 或者用不了 HTTPS（现实中的 HTTP2 都是在 HTTPS 上实现的，所以也就是只能使用 HTTP/1.1）。只能使用持久化连接，一个TCP可以发送多个HTTP请求，服务器按顺序接收请求与返回响应资源。

## HTTP状态码

（1）五种类别

1xx消息—— 请求已被服务器接收，继续处理

2xx成功 —— 请求已成功被服务器接收、理解、并接受

3xx重定向 —— 需要后续操作才能完成这一请求

4xx请求错误 —— 请求含有词法错误或者无法被执行

5xx服务器错误 —— 服务器在处理某个正确请求时发生错误

（2）常见的状态码

200 OK //客户端请求成功

400 Bad Request //客户端请求有语法错误，不能被服务器所理解

401 Unauthorized //请求未经授权，这个状态代码必须和WWW-Authenticate报头域一起使用

403 Forbidden //服务器收到请求，但是拒绝提供服务

404 Not Found //请求资源不存在，比如：输入了错误的URL

500 Internal Server Error //服务器发生不可预期的错误

503 Server Unavailable //由于超载或系统维护，服务器暂时的无法处理客户端的请求

## GET和POST是HTTP请求的两种基本方法

* GET在浏览器回退时是无害的，而POST会再次提交请求。
* GET产生的URL地址可以被Bookmark，而POST不可以。
* GET请求会被浏览器主动cache，而POST不会，除非手动设置。
* GET请求只能进行url编码，而POST支持多种编码方式。
* GET请求参数会被完整保留在浏览器历史记录里，而POST中的参数不会被保留。
* GET请求在URL中传送的参数是有长度限制的，而POST么有。
* 对参数的数据类型，GET只接受ASCII字符，而POST没有限制。
* GET比POST更不安全，因为参数直接暴露在URL上，所以不能用来传递敏感信息。
* GET参数通过URL传递，POST放在Request body中。

HTTP的底层是TCP/IP。所以GET和POST的底层也是TCP/IP，也就是说，GET/POST都是TCP链接。GET和POST能做的事情是一样一样的。

GET和POST还有一个重大区别，简单的说：

GET产生一个TCP数据包；POST产生两个TCP数据包。

长的说：

对于GET方式的请求，浏览器会把http header和data一并发送出去，服务器响应200（返回数据）；

而对于POST，浏览器先发送header，服务器响应100 continue，浏览器再发送data，服务器响应200 ok（返回数据）。

也就是说，GET只需要汽车跑一趟就把货送到了，而POST得跑两趟，第一趟，先去和服务器打个招呼“嗨，我等下要送一批货来，你们打开门迎接我”，然后再回头把货送过去。

因为POST需要两步，时间上消耗的要多一点，看起来GET比POST更有效。因此Yahoo团队有推荐用GET替换POST来优化网站性能。但这是一个坑！跳入需谨慎。为什么？

1. GET与POST都有自己的语义，不能随便混用。

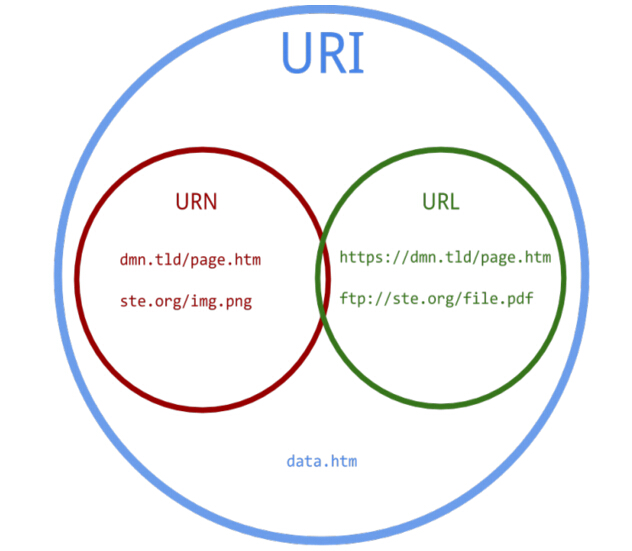
2. 据研究，在网络环境好的情况下，发一次包的时间和发两次包的时间差别基本可以无视。而在网络环境差的情况下，两次包的TCP在验证数据包完整性上，有非常大的优点。

3. 并不是所有浏览器都会在POST中发送两次包，Firefox就只发送一次。

## URI、 URL、 URN的区别

通过HTTP或者HTTPS协议请求的资源由统一资源标识符（uniform resource identifiers， URI）来标识。常用的是URL。

* URI = Uniform Resource Identifier 统一资源标识符，用来唯一标识符来标识一个资源，是一个通用的概念，URI由URL和URN两个子集组成。
* URL = Uniform Resource Locator 统一资源定位符，通过描述资源的位置来标识资源。
* URN = Universal Resource Name 统一资源名称，通过资源的名字来标识资源，与其所处的位置无关，这样即使资源的位置发生变动，其URN也不会变化。



## URL、HTTP、IP、域名、DNS的关系

* 1. URL：统一资源定位符

URL包括：协议部分、域名、端口、路径（虚拟路径）、携带的参数、哈希值；

http://www.jianshu.com/index.html?name=oldUath&age=25#dowell

协议 域名 路径 ?查询参数 #锚点

1. 协议部分：

该URL的协议部分为“http：”，这代表网页使用的是HTTP协议。在Internet中可以使用多种协议，如HTTP，FTP等等本例中使用的是HTTP协议。在“HTTP”后面的“//”为分隔符。

1. 域名部分：

该URL的域名部分为“ [www.jianshu.com](http://www.jianshu.com) ”。一个URL中，**也可以使用IP地址作为域名使用**

1. 端口部分：

跟在域名后面的是端口，域名和端口之间使用“:”作为分隔符。端口不是一个URL必须的部分，如果省略端口部分，将采用默认端口。

http：默认端口号80；

https：默认端口号443；

ftp：默认端口号21

1. 路径：

index.html：这里虚拟了一个目录；

http://www.jianshu.com/路径1/路径2，“/”表示根目录

1. 参数部分：

从“？”开始到“ # ”结束，之间的部分为参数部分，又称搜索部分、查询部分。参数可以允许有多个参数，参数与参数之间用“ & ”作为分隔符。

1. 锚部分：

从“ # ”开始到最后，都是锚部分。锚部分也不是一个URL必须的部分

* 1. DNS：域名解析服务

DNS (Domain Name Server ) 译为：域名解析服务

主要的作用：将网址 （域名） 如“www.baidu.com”“翻译”成电脑可以理解的 IP 地址

* 1. IP：网络之间互连的协议

IP (Internet Protocol) 译为：网络之间互连的协议

1. IP主要做两件事

* 如何定位一台设备
* 如何封装数据报文与其他设备交流

1. IP分为内网和外网

***几个特殊的IP地址***

* 127.0.0.1表示本地机器自身的IP地址
* localhost通过hosts指定本地机器自身。可以通过修改host替换掉localhost;C:\Windows\System32\drivers\etc 找到hosts文件

ping 命令，通常用来测试与目标主机的连通性

ping 主机地址

* 1. 域名：IP地址别称

1. IP地址的别名

为了更好的用户体验，记住网址，使用域名代替IP地址，例如：www.baidu.com 就是百度的域名。

一个域名可以对应不同IP这个叫做**均衡负载**，防止一台机器扛不住；

一个IP可以对应不同域名这个叫做**共享主机；**

**域名 和 IP 是通过DNS链接起来的**

1. 域名的级别

com是顶级域名

baidu.com是二级域名（俗称一级域名）

www.baidu.com是三级域名（俗称二级域名）

## 一个完整的HTTP请求过程

***：在浏览器中输入URL后，浏览器做了什么***

* 1. 简单的概括

1. 首先 Chrome 搜索自身的 DNS 缓存。(如果 DNS 缓存中找到百度的 IP 地址，就跳过了接下来查找 IP 地址步骤，直接访问该 IP 地址。)
2. 搜索操作系统自身的 DNS 缓存。(浏览器没有找到缓存或者缓存已经失效)
3. 读取硬盘中的 host 文件，里面记录着域名到 IP 地址的映射关系，Mac 电脑中位于 /etc/hosts。(如果前1.2步骤都没有找到)
4. 浏览器向宽带运营商服务器或者域名服务器发起一个 DNS 解析请求，（这里服务器有两种方式解析请求，这在稍后会讲到）之后浏览器获得了百度首页的 IP 地址。
5. 拿到 IP 地址后，浏览器就向该 IP 所在的服务器建立 TCP 连接(即三次握手)。
6. 连接建立起来之后，浏览器就可以向服务器发起 HTTP 请求了。(比如访问百度首页，就向服务器发起 HTTP 中的 GET 请求)
7. 服务器接受到这个请求后，根据路径参数，经过后台一些处理之后，把处理后的结果返回给浏览器，如果是百度首页，就可以把完整的 HTML 页面代码返回给浏览器。
8. 浏览器拿到了百度首页的完整 HTML 页面代码，内核和 JS 引擎就会解析和渲染这个页面，里面的 JS、CSS、图片等静态资源也通过一个个 HTTP 请求进行加载。
9. 浏览器根据拿到的资源对页面进行渲染，最终把完整的页面呈现给用户。
10. 如果浏览器没有后续的请求，那么就会跟服务器端发起 TCP 断开(即四次挥手)。

至此，整个访问过程就结束。

* 1. 更简单的概括

域名解析 —> 与服务器建立连接 —> 发起HTTP请求 —> 服务器响应HTTP请求，浏览器得到html代码 —> 浏览器解析html代码，并请求html代码中的资源（如js、css、图片） —> 浏览器对页面进行渲染呈现给用户

* 1. 过程流程图

浏览器地址栏输入URL

浏览器搜索自身DNS缓存中是否存在该IP

读取硬盘中host文件是否存在域名与IP的映射关系

浏览器向宽带运营商服务器

或域名服务器

发起一个 DNS 解析请求

拿到 IP 地址后

**否**

**否**

DOM树 + CSS对象

HTML页面/JS/CSS/图片等静态资源

是

是

浏览器与服务器建立TCP连接

内核和 JS 引擎，解析和渲染

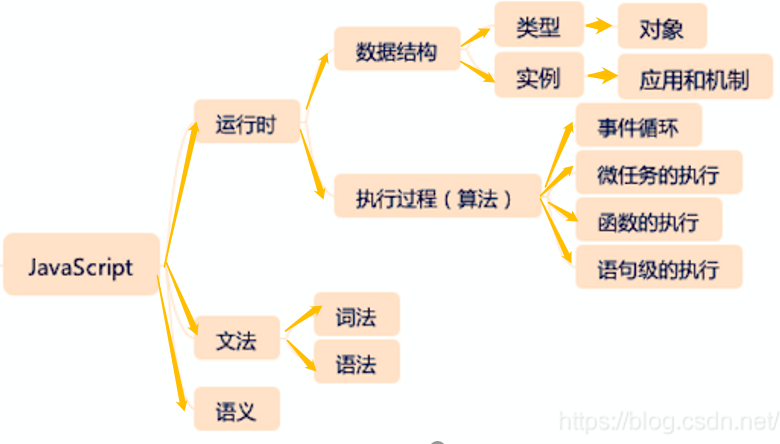
服务器接收、处理请求，返回资源

渲染HTML页面

## 浏览器内置数据库

# JavaScript（ES5）

## JavaScript的编程思想



## 数据类型

值类型：String、Symbol、Null、Undefined、Number、Boolean、

引用对象：Array、Data、Meth、Function、Object..

是

是

浏览器与服务器建立TCP连接

内核和 JS 引擎，解析和渲染

服务器接收、处理请求，返回资源

渲染HTML页面

JavaScript 对象的原型 – prototype

?

?

浏览器向宽带运营商服务器

或域名服务器

发起一个 DNS 解析请求

?

否

否

## 对象的原型（prototype） 和 原型链

Javascript中有一句话，叫一切皆是对象。当然这句话也不严谨，比如null和undefined就不是对象。除了这两个完全可以说Javascript一切皆是对象。

简介：

*// 父类*

class People {

constructor(name) {

this.name = name

}

eat() {

console.log(`${this.name} eat something`)

}

}

*// 子类*

class Student extends People {

constructor(name, number) {

super(name)

this.number = number

}

sayHi() {

console.log(`姓名 ${this.name} 学号 ${this.number}`)

}

}

*// 子类*

class Teacher extends People {

constructor(name, major) {

super(name)

this.major = major

}

teach() {

console.log(`${this.name} 教授 ${this.major}`)

}

}

const ming = new Student('小明', 100) *// 实例*

ming.sayHi() *// 子类的方法*

ming.eat() *// 继承父类的方法*

const wangTeacher = new Teacher('王老师', '语文') *// 实例*

wangTeacher.teach() *// 子类的方法*

wangTeacher.eat() *// 继承父类的方法*

console.log(Student.prototype.\_\_proto\_\_ === People.prototype); *// true*

console.log(Student.prototype === ming.\_\_proto\_\_); *// true*

console.log(ming.\_\_proto\_\_.\_\_proto\_\_ === People.prototype) *// true*

*// ming.\_\_proto\_\_ 叫* ***隐式原型***

*// Student.prototype 叫* ***显式原型***

**注意**

* 每个 class 都有 **显示原型**
* 每个 实例 都有 **隐式原型**
* 隐式原型的指针指向class对应显示原型，实例与class之间形成的关系叫原型链
* 所以原型的执行规则就是先到自身的属性和方法中寻找，找不到则到隐式原型中找

ming

name

小明

number

100

\_\_proto\_\_

Object.prototype

toString

函数

hasOwnProperty

函数

\_\_proto\_\_

Student.prototype

sayHi

函数

\_\_proto\_\_

Student

prototype

People

prototype

People.prototype

.prototype

eat

函数

\_\_proto\_\_

**new构造函数**

**extends继承**

null

Object

prototype

原型与原型链

* 1. 原型（prototype）
     1. 原型（prototype）的语法

原型的英文应该叫做 prototype，任何一个对象都有**原型**，可以通过*非标准属性* \_\_proto\_\_ 来访问一个对象的原型。

示例1：

// 纯对象的原型默认是个空对象

console.log({}.\_\_proto\_\_); // => {}

function Student(name, grade) {

this.name = name;

this.grade = grade;

}

const stu = new Student('xiaoMing', 6);

// Student 类型实例的原型，默认也是一个空对象

console.log(stu.\_\_proto\_\_); // => Student {}

console.log({}.\_\_proto\_\_ **===** Object.getPrototypeOf({})); *// => true*

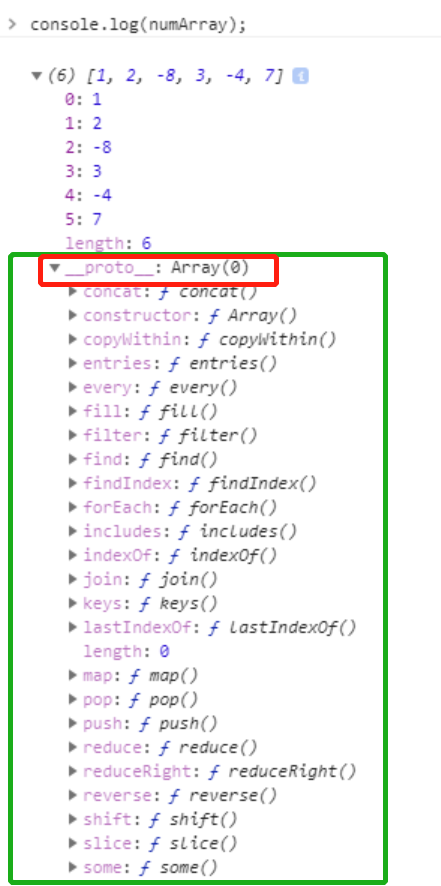
\_\_proto\_\_ 是非标准属性，如果要访问一个对象的原型，建议使用 ES6 新增的 Reflect.getPrototypeOf 或者 Object.getPrototypeOf() 方法。非标准属性意味着未来可能直接会修改或者移除该属性，说不定以后出了个新标准用 Symbol.proto 作为 key 来访问对象的原型，那这个非标准属性可能就要被移除了。

示例2：

**let** numArray **=** [1, 2, **-**8, 3, **-**4, 7];

因为Array对象本身有一个原型属性指向Array.prototype，而变量numArray数据类型是Array，所以继承了Array.prototype对象所有的属性和方法。

numArray.\_\_proto\_\_ **===** Array.prototype *// true*



* + 1. 原型赋值(null)的注意事项

1. 可以对 \_\_proto\_\_ 属性直接赋值的方式修改对象的原型
2. 更推荐是使用 ES6 的 Reflect.setPrototypeOf 或 Object.setPrototypeOf。

不论哪种方式，被**设置的值**的类型**只能是对象或者 null**，其它类型不起作用：

const obj = { name: 'xiaoMing' };

console.log(obj.\_\_proto\_\_); // => {} ：原型为空对象

obj.\_\_proto\_\_ = 666; // 给\_\_proto直接赋值成“非对象”和 null 不生效

console.log(obj.\_\_proto\_\_); // => {}

obj.\_\_proto\_\_ = { a: 1 };// 设置原型为对象

console.log(obj.\_\_proto\_\_); // => { a: 1 }

console.log(Reflect.getPrototypeOf(obj)); // => { a: 1 }

const obj = { name: 'xiaoming' };

obj.\_\_proto\_\_ = null; *//第一次，给赋值为null*

*//为什么不是 null，就好像 \_\_proto\_\_ 被删除了*

console.log(obj.\_\_proto\_\_); *// => undefined*

console.log(Reflect.getPrototypeOf(obj));*// null 确实将原型设置为 null了*

*//-------------------------------------*

obj.\_\_proto\_\_ = null; *// 第二次，再次赋值为 null 🡺赋值失败*

console.log(obj.\_\_proto\_\_); *// => null：为什么不是undefined了？*

obj.\_\_proto\_\_ **=** { a**:** 1 }; *// 第三次，赋值为一个对象 🡺****变成普通属性***

console.log(obj.\_\_proto\_\_); *// => { a: 1 }*

*// \_\_proto\_\_ 就像一个普通属性一样 obj.xxx = { a: 1 }*

*// 所以在第三次赋值时并没有将原型设置成功*

console.log(Reflect.getPrototypeOf(obj)); *// => null*

Reflect.setPrototypeOf(obj, { b**:** 2 }); *// 第四次，使用ES6中的方法再次赋值*

*// \_\_proto\_\_ 被设置为 null 后，obj 的 \_\_proto\_\_ 属性和一个普通的属性没有区别*

console.log(obj.\_\_proto\_\_); *// => { a: 1 }*

*// 但是使用 Reflect.setPrototypeOf 是可以设置原型的，设置成功*

console.log(Reflect.getPrototypeOf(obj)); *// => { b: 2 }*

其实 \_\_proto\_\_ 是个定义在 Object.prototype 上的**访问器属性**，也就是使用getter 和 setter 定义的属性，通过 \_\_proto\_\_ 的 getter可以获取到对象的[[Prototype]]，也就是原型。

**总的来说**：如果一个对象的 \_\_proto\_\_ 属性（在第一次已经）被赋值为 null，这个时候它的原型确实已经被修改为 null 了，（紧接着第二次）再通过对 \_\_proto\_\_ 赋值的方式设置原型时是无效的，这个时候 \_\_proto\_\_ 和一个普通属性没有区别，只能通过 Reflect.setPrototypeOf 或 Object.setPrototypeOf 才能修改原型。原型是对象内部的一个属性 [[prototype]]，而 Reflect.setPrototypeOf 之所以能修改原型是因为它是直接修改对象的原型属性，也就是内部直接对对象的 [[prototype]] 属性赋值，而不会通过 \_\_proto\_\_ 的 getter。

* + 1. 构造器和prototype

构造器的英文就是 constructor，在 JavaScript 中，函数都可以用作构造器。构造器也可以称之为类。

1. 通过 new 构造器来构造一个实例。

function Apple() {} // 构造器，Apple构造器可以称之为 Apple类

const apple = new Apple();//通过 new 构造器来构造一个实例

console.log(apple instanceof Apple); // => true

1. **任何构造器都有一个 prototype 属性，默认是一个空的纯对象，所有由构造器构造的实例的原型都是指向它。**

// 实例的原型即 apple.\_\_proto\_\_

console.log(apple.\_\_proto\_\_ === Apple.prototype); // => true

下面的示例可以证明构造器的 prototype 属性默认是个**空对象（纯对象）**，注意这里说的空对象指的是该对象没有可遍历属性：

console.log(Apple.prototype); // => Apple {}

console.log(Object.keys(Apple.prototype)); // => []

console.log(Apple.prototype.\_\_proto\_\_ === {}.\_\_proto\_\_); // true

1. **构造器的 prototype**有一个 **constructor 属性**，指向构造器本身

console.log(Apple.prototype.constructor === Apple); // => true

这个**constructor 属性是不可遍历的**，可以理解是这样定义该属性的：

Object.defineProperty(Apple.prototype, 'constructor', {

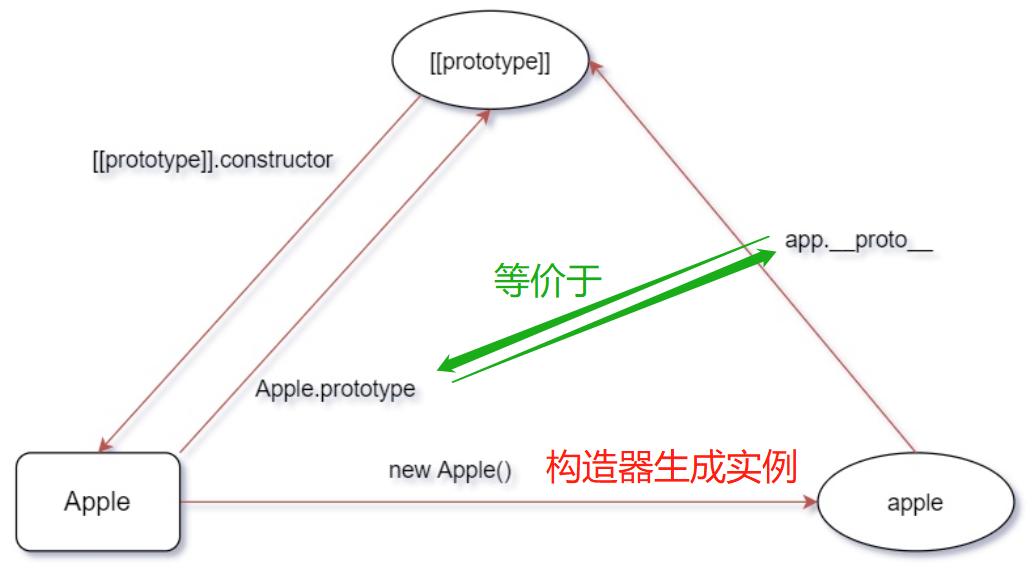
value: Student,

writable: true,

enumerable: fasle, // 不可枚举，无法通过 Object.keys() 获取到

});

\_\_proto\_\_ ，prototype ，constructor ，Apple函数，实例 apple 和原型对象 [[prototype]] 之间的关系：



* + 1. \_\_proto\_\_和prototype完全不同

\_\_proto\_\_ 存在于所有的对象上，prototype 存在于所有的函数上，两者的关系就是：函数的 prototype 是所有使用 new 这个函数构造的实例的 \_\_proto\_\_。函数也是对象，所以函数同时有 \_\_proto\_\_ 和prototype。

函数：prototype、\_\_proto\_\_

对象：\_\_proto\_\_

注意：这里提到的构造器new的原型，指的是构造器的 \_\_proto\_\_，而不是构造器的 prototype 属性。

当一个构造函数被创建后，实例对象会继承构造函数的原型属性，这是构造函数的一个非常重要的特性。在Javascript中使用new关键字来对构造函数进行实例化。

结论：每一个实例对象都有一个私有属性\_\_proto\_\_，指向它的构造函数new的原型对象(prototype)。

* 1. 原型链

**各个原型之间构成的链，称之为原型链。**

**对象的原型特点**：当想访问一个对象obj上的某个属性时，如果不存在于 obj上，那么便会去对象的原型上也就是 obj.\_\_proto\_\_ 上去找这个属性。如果有则返回这个属性，没有则去对象obj的原型的原型上也就是 obj.\_\_proto\_\_.\_\_proto\_\_去找，重复以上步骤。一直到访问**纯对象**的原型也就是 Object.prototype，没有的话续往上找也就是 Object.prototype.\_\_proto\_\_，返回null。

console.log(Object.prototype.\_\_proto\_\_); // => null

console.log(Object.\_\_proto\_\_.\_\_proto\_\_); // => {}

* 原型对象的尽头，所有对象的原型尽头是Object.prototype。
* 而Object.prototype这个对象的\_\_proto\_\_指向是null，返回 null。
* 日常开发中用到的绝大多数对象的\_\_proto\_\_基本不会直接指向Object.prototype，基本都是指向另一个对象。比如，所有的函数的\_\_proto\_\_都会指向Function.prototype，所有数组的\_\_proto\_\_都会指向Array.prototype。

结论：每一个实例对象都有一个私有属性\_\_proto\_\_，指向它的构造函数的原型对象(prototype)。原型对象也有自己的\_\_proto\_\_，层层向上直到一个对象的原型对象为null。**这一层层原型就是原型链。**

纯对象的原型的原型是 null：

console.log(new Object().\_\_proto\_\_.\_\_proto\_\_); // => null

console.log({}.\_\_proto\_\_.\_\_proto\_\_) // => null

* + 1. 对象的原型链

newObj.\_\_proto\_\_.\_\_proto\_\_

原始对象（纯对象）：Object.prototype

构造器实例：newObj

父级对象：parentObj.prototype

newObj.\_\_proto\_\_

parentObj.\_\_proto\_\_

null

Object.\_\_proto\_\_

newObj.\_\_proto\_\_.\_\_proto\_\_.\_\_proto\_\_

对象原型链

* + 1. 函数的原型链

newObj.\_\_proto\_\_.\_\_proto\_\_

原始对象（纯对象）：Object.prototype

对象a：aObj

constructor函数：Function.prototype

newObj.\_\_proto\_\_

null

Object.\_\_proto\_\_

newObj.\_\_proto\_\_.\_\_proto\_\_.\_\_proto\_\_

函数的原型链

* 1. 原型的用途
     1. 定义实例方法

在使用构造器定义一个类型的时候，一般会将类的方法定义在原型上，和 this 的指向特性简直是绝配。

function Engineer(workingYears) {

this.workingYears = workingYears;

}

// 不能使用箭头函数，箭头函数的 this 在声明的时候就根据上下文确定了

Engineer.prototype.built = function () {

// this 这里就是执行函数调用者

console.log(`我已经工作 ${this.workingYears} 年了`);

};

const engineer = new Engineer(5);

// this 会正确指向实例，所以 this.workingYears 是 5

engineer.built(); // => 我已经工作 5 年了

console.log(Object.keys(engineer)); // => [ 'workingYears' ]

通过这种方式，所有的实例都可以访问到这个方法，并且这个方法只需要占用一份内存，节省内存，this 的指向还能正确指向类的实例。

不过这种方式定义的方法无法通过 Object.keys() 访问，毕竟不是自身的属性：

const obj = {

func() {},

};

console.log(Object.keys(obj)); *// => [ 'func' ]*

function Func() {}

Func.prototype.func = function () {};

console.log(Object.keys(new Func())); *// => []*

JavaScript中很多方法都定义在构造器的原型上，例如 Array.prototype.slice，Object.prototype.toString等。

* + 1. 定义实例属性

定义实例的属性，通过 this.xxx = xxx 的方式定义：

function Engineer(workingYears) {

this.workingYears = workingYears;

this.built = function () {

console.log(`我已经工作 ${this.workingYears} 年了`);

};

}

const engineer = new Engineer(5);

console.log(Object.keys(engineer)); // => [ 'workingYears', 'built' ]

* 1. ES6 class和构造器的关系

很多语言都有拥有面向对象编程范式，例如 java，c#，python 等。ES6 class 让从它们转到 JavaScript 的开发者更容易进行面向对象编程。

* + 1. ES6 class

其实，ES6 class 就是构造器的语法糖。

babel 会将 ES6 class 编译成ES5（原代码）：

class Circle {

constructor(x, y, r) {

this.x = x;

this.y = y;

this.r = r;

}

draw() {

console.log(`画个坐标(${this.x}, ${this.y})，半径为 ${this.r} 的圆`);

}

}

babel + babel-preset-es2015-loose 编译出的结果：

'use strict';

// Circle class 可以理解为就是一个构造器函数

var Circle = (function () {

function Circle(x, y, r) {

this.x = x;

this.y = y;

this.r = r;

}

var \_proto = Circle.prototype;

// class 方法定义在 prototype 上

\_proto.draw = function draw() {

console.log('\u753B\u4E2A\u5750\u6807\u4E3A('+this.x+','+this.y+

')\uFF0C\u534A\u5F84\u4E3A ' + this.r + ' \u7684\u5706');

};

return Circle;

})();

总结：ES6 的 class 就是构造器，class 上的方法定义在构造器的 prototype 上。

* + 1. extends 继承

使用 extends 继承时的转换（原代码）:

class Shape {

constructor(x, y) {

this.x = x;

this.y = y;

}

}

class Circle extends Shape {

constructor(x, y, r) {

super(x, y);

this.r = r;

}

draw() {

console.log(`画个坐标(${this.x}, ${this.y})，半径为 ${this.r} 的圆`);

}

}

babel + babel-preset-es2015-loose 编译出的结果：

'use strict';

// 原型继承

function \_inheritsLoose(subClass, superClass) {

subClass.prototype = Object.create(superClass.prototype);

subClass.prototype.constructor = subClass;

// 让子类可以访问父类上的静态属性，其实就是定义在构造器自身上的属性

// 例如父类有 Person.say 属性，子类 Student 通过可以通过 Student.say 访问

subClass.\_\_proto\_\_ = superClass;

}

var Shape = function Shape(x, y) {

this.x = x;

this.y = y;

};

var Circle = (function (\_Shape) {

\_inheritsLoose(Circle, \_Shape);

function Circle(x, y, r) {

var \_this;

// 组合继承

\_this = \_Shape.call(this, x, y) || this;

\_this.r = r;

return \_this;

}

var \_proto = Circle.prototype;

\_proto.draw = function draw() {

console.log('\u753B\u4E2A\u5750\u6807\u4E3A('+this.x+','+this.y+

')\uFF0C\u534A\u5F84\u4E3A ' + this.r + ' \u7684\u5706');

};

return Circle;

})(Shape);

**整个 ES6 的 extends 实现的是原型继承 + 组合继承。**

子类构造器中调用了父类构造器并将 this 指向子类实例达到将父类的实例属性组合到子类实例上：

\_this = \_Shape.call(this, x, y) || this; // 组合继承

\_inheritsLoose 这个函数就是原型继承。

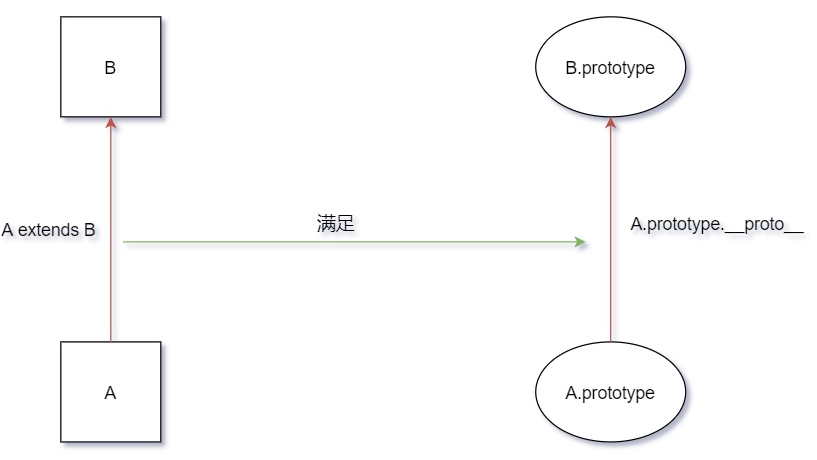
* + 1. 原型继承

通俗意义上的继承是说：如果类A 继承 类B，那么 A 的实例继承了 B 的实例属性。

原型继承和通俗意义上的继承还不太一样，它是：A 的实例能够继承 B 的原型上的属性。

**给原型继承下个定义：**

对于类A 和 类B，如果满足 A.prototype.\_\_proto\_\_ === B.prototype，那么 A 原型继承 B



原型继承的最简单实现方式就是直接设置 A.prototype === new B()，让 A 的 prototype 是 B 的实例即可：

function A() {}

function B() { this.xxx = '污染 A 的原型'; }

A.prototype = new B();

console.log(A.prototype.\_\_proto\_\_ === B.prototype); // => true

但是这种方式会导致 B 的实例属性污染 A 的原型。

解决办法，就是通过一个空的函数桥接一下，空的函数不会有实例属性污染原型链：

**function** A(p) {  this.p = p;}

**function** B()  {  this.xxx = '污染原型';}

**function** E()  {} *// 空函数*

console.log("new B(): ", new B()); *// => B { xxx: '污染原型' }*

console.log("B.prototype.constructor: ", B.prototype.constructor); *// => function B() { this.xxx = '污染原型';}*

E.prototype = B.prototype; *//****空函数转接后，E去除了B原型上定义的属性***

console.log("E.prototype: ", E.prototype); *// => B {}*

console.log("E.\_\_proto\_\_: ", E.\_\_proto\_\_); *// => function () { [native code] }*

console.log("E.prototype.xxx: ", E.prototype.xxx); *// => undefined*

console.log("E.\_\_proto\_\_.xxx: ", E.\_\_proto\_\_.xxx); *// => undefined*

console.log("E.prototype.\_\_proto\_\_: ", E.prototype.\_\_proto\_\_); *// => {}*

console.log("B.prototype: ", B.prototype); *// => B {}*

console.log("B.\_\_proto\_\_: ", B.\_\_proto\_\_); *// => function () { [native code] }*

A.prototype = new E();

console.log("A.prototype: ", A.prototype); *// => B {}*

console.log("A.\_\_proto\_\_: ", A.\_\_proto\_\_); *// => function () { [native code] }*

console.log("A.constructor: ", A.constructor); *// => function Function() { [native code] }*

console.log("A.constructor.prototype: ", A.constructor.prototype); *// => function () { [native code] }*

console.log("A.constructor.\_\_proto\_\_: ", A.constructor.\_\_proto\_\_); *// => function () { [native code] }*

console.log("A.prototype.constructor: ", A.prototype.constructor); *// => function B()  {  this.xxx = '污染原型';}* ***这里需要注意，原型上有属性***

console.log("A.prototype.constructor.prototype: ", A.prototype.constructor.prototype); *// => B {}*

console.log("A.prototype.constructor.\_\_proto\_\_: ", A.prototype.constructor.\_\_proto\_\_); *// => function () { [native code] }*

console.log("A.prototype.constructor.prototype.\_\_proto\_\_: ", A.prototype.constructor.prototype.\_\_proto\_\_); *// => {}*

A.prototype.constructor = A; *//****修正 constructor 指向，***

*//****满足原型继承的定义***

console.log("A.prototype.\_\_proto\_\_: ", A.prototype.\_\_proto\_\_);*//=> B SSS{}*

console.log("A.prototype.\_\_proto\_\_ === B.prototype: ", A.prototype.\_\_proto\_\_ === B.prototype); *// => true*

**const** a = new A('p');

console.log("a.\_\_proto\_\_: ", a.\_\_proto\_\_); *// => A { constructor: [Function: A] }*

console.log("a.\_\_proto\_\_.\_\_proto\_\_: ", a.\_\_proto\_\_.\_\_proto\_\_);*//=> B {}*

console.log("a.prototype: ", a.prototype); *// => undefined*

console.log("a.constructor: ", a.constructor); *// => function A(p) {  this.p = p;}*

console.log("a instanceof A: ", a instanceof A); *// => true*

**const** b = new B();

console.log("b instanceof B: ", b instanceof B); *// => true*

*// a 也是 B 的实例*

console.log("a instanceof B: ", a instanceof B); *// => true*

console.log("a.\_\_proto\_\_.\_\_proto\_\_ === B.prototype: ", a.\_\_proto\_\_.\_\_proto\_\_ === B.prototype); *// => true*

console.log("a instanceof E: ", a instanceof E); *// => true*

对象A 、B、空对象之间的原型链关系：

A.prototype

B.prototype

E.prototype

A.prototype.constructor

A

new E()

A.prototype.\_\_proto\_\_

a

new A()

a.\_\_proto\_\_.\_\_proto\_\_

**空对象转接**

**修正constructor指向**

**注意：**

1. B.prototype 直接赋值于 E.prototype，E.prototype 等于 B.prototype；
2. A.prototype 由构造器new E()赋值，A.prototype 不等于E.prototype；
3. 虽然B.prototype，E.prototype，A.prototype都为B 类（即，B {}）

利用 Object.create，可以更简单的实现原型继承，也就是上面的 babel 用到的工具函数 \_inheritsLoose：

function \_inheritsLoose(subClass, superClass) {

// Object.create(prototype) 返回一个以 prototype 为原型的对象

subClass.prototype = Object.create(superClass.prototype);

subClass.prototype.constructor = subClass;

// 我们上面实现的原型继承没有设置这个，但是 class 的继承会设置子类的原型为父类

subClass.\_\_proto\_\_ = superClass;

}

**JavaScript 和原型相关语法特性**

其实由很多语法特性是和原型有关系的，讲到原型那么我们就再继续讲讲 JavaScrip 语法特性中涉及到原型的一些知识点。

* + 1. new运算符

new 运算符创建一个用户定义的对象类型的实例或具有构造函数的内置对象的实例。

function Car(make, model, year) {

this.make = make;

this.model = model;

this.year = year;

}

const car1 = **new** Car('Eagle', 'Talon TSi', 1993);

console.log(car1.make); // Eagle

1. **语法**

new constructor[([*arguments*])]

constructor 一个指定对象实例的类型的类或函数。

arguments 一个用于被 constructor 调用的参数列表。

1. **描述**

new 关键字会进行如下的操作：

1. 创建一个空的简单JavaScript对象（即{}）；
2. 链接该对象（设置该对象的constructor）到另一个对象 ；
3. 将步骤a)新创建的对象作为this的上下文 ；
4. 如果该函数没有返回对象，则返回this。

模拟执行过程：

function isObject(value) {

const type = typeof value;

return value !== null && (type === 'object' || type === 'function');

}

/\*\*

\* constructor 表示 new 的构造器

\* args 表示传给构造器的参数

\*/

function New(constructor, ...args) {

// new 的对象不是函数就抛 TypeError

if (typeof constructor !== 'function') throw new TypeError(`${constructor} is not a constructor`);

// 创建一个原型为构造器的 prototype 的空对象 target

const target = Object.create(constructor.prototype);

// 将构造器的 this 指向上一步创建的空对象，并执行，为了给 this 添加实例属性

const result = constructor.apply(target, args);

// 上一步的返回如果是对象就直接返回，否则返回 target

return isObject(result) ? result : target;

}

测试一下：

function Computer(brand) {

this.brand = brand;

}

const c = New(Computer, 'Apple');

console.log(c); // => Computer { brand: 'Apple' }

如果没有使用 new 运算符， **构造函数会像其他的常规函数一样被调用，** 并不会创建一个对象**。**在这种情况下， this 的指向也是不一样的。

* + 1. instanceof 运算符原理

instanceof 运算符用于检测构造函数的 prototype 属性是否出现在某个实例对象的原型链上。

1. **语法**

object instanceof constructor

object 某个实例对象

constructor 某个构造函数

1. **描述**

instanceof 运算符用来检测 constructor.prototype 是否存在于参数 object 的原型链上。

// 定义构造函数

function C(){}

function D(){}

var o = new C();

o instanceof C; // true，因为 Object.getPrototypeOf(o) === C.prototype

o instanceof Object; // true，因为 Object.prototype.isPrototypeOf(o) 返回 true

C.prototype instanceof Object // true，同上

o instanceof D; // false，因为 D.prototype 不在 o 的原型链上

C.prototype = {}; // C.prototype 指向了一个空对象

var o2 = new C();

o2 instanceof C; // true

o instanceof C; // false，C.prototype 指向了一个空对象，这个空对象不在 o 的原型链上。

D.prototype = new C(); // 继承

var o3 = new D();

o3 instanceof D; // true

o3 instanceof C; // true 因为 C.prototype 现在在 o3 的原型链上

需要注意的是，如果表达式 obj instanceof Foo 返回 true，则并不意味着该表达式会永远返回 true，因为 Foo.prototype 属性的值有可能会改变，改变之后的值很有可能不存在于 obj 的原型链上，这时原表达式的值就会成为 false。另外一种情况下，原表达式的值也会改变，就是改变对象 obj 的原型链的情况，虽然在目前的ES规范中，我们只能读取对象的原型而不能改变它，但借助于非标准的 \_\_proto\_\_ 伪属性，是可以实现的。比如执行 obj.\_\_proto\_\_ = {} 之后，obj instanceof Foo 就会返回false了。

它的原理很简单，一句话概括就是：obj instanceof 构造器 A，等同于判断 A 的 prototype 是不是 obj 的原型（也可能是二级原型）。

代码实现：

function instanceOf(obj, constructor) {

if (!isObject(constructor)) {

throw new TypeError(`Right-hand side of 'instanceof' is not an object`);

} else if (typeof constructor !== 'function') {

throw new TypeError(`Right-hand side of 'instanceof' is not callable`);

}

// 主要就这一句

return constructor.prototype.isPrototypeOf(obj);

}

简单测试一下：

function A() {}

const a = new A();

console.log(a instanceof A); // => true

console.log(instanceOf(a, A)); // => true

--------

* 1. 扩展原型链的四种方法
     1. 构造函数创建

使用构造函数创建对象的例子：

**function** Animal(name) {

**this**.name **=** name;

}

Animal.prototype **=** {

run() {

console.log('跑步');

}

}

**let** cat **=** **new** Animal('Tom');

cat.\_\_proto\_\_ **===** Animal.prototype; *// true*

Animal.prototype.\_\_proto\_\_ **===** Object.prototype; *// true*

**优点：**支持目前以及所有可想象到的浏览器(IE5.5都可以使用). 这种方法非常快，**非常符合标准**，并且充分利用JIST优化。

**缺点**：为使用此方法，这个问题中的函数必须要被初始化。另外构造函数的初始化，可能会给生成对象带来并不想要的方法和属性。

* + 1. Object.create

ECMAScript 5 中引入了一个新方法: Object.create()。可以调用这个方法来创建一个新对象。新对象的原型就是调用 create 方法时传入的第一个参数：

**var** a **=** {a**:** 1};

*// a ---> Object.prototype ---> null*

**var** b **=** Object.create(a);

b.\_\_proto\_\_ **===** a; *// true*

**优点**： 支持当前所有非微软版本或者 IE9 以上版本的浏览器。允许一次性地直接设置 \_\_proto\_\_ 属性，以便浏览器能更好地优化对象。同时允许通过 Object.create(null)来创建一个没有原型的对象。

**缺点**：不支持 IE8 以下的版本；这个慢对象初始化在使用第二个参数的时候有可能成为一个性能黑洞，因为每个对象的描述符属性都有自己的描述对象。当以对象的格式处理成百上千的对象描述的时候，可能会造成严重的性能问题。

* + 1. Object.setPrototypeOf

**语法**：

Object.setPrototypeOf(obj, prototype)

**参数**：

obj: 要设置其原型的对象。

prototype: 该对象的新原型(一个对象 或 ref="https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global\_Objects/null"> null)

**var** a **=** { n**:** 1 };

**var** b **=** { m **:** 2 };

Object.setPrototypeOf(a, b);

a.\_\_proto\_\_ **===** b; *// true*

**优点**：支持所有现代浏览器和微软IE9+浏览器。允许动态操作对象的原型，甚至能强制给通过 Object.create(null)创建出来的没有原型的对象添加一个原型。

**缺点**：这个方式表现并不好，应该被弃用；动态设置原型会干扰浏览器对原型的优化；不支持 IE8 及以下的浏览器版本。

* + 1. \_\_proto\_\_

**var** a **=** { n**:** 1 };

**var** b **=** { m **:** 2 };

a.\_\_proto\_\_ **=** b;

a.\_\_proto\_\_ **===** b; *// true*

使用\_\_proto\_\_也可以动态设置对象的原型。

**优点**：支持所有现代非微软版本以及 IE11 以上版本的浏览器。将 \_\_proto\_\_ 设置为非对象的值会静默失败，并不会抛出错误。

**缺点**：应该完全将其抛弃因为这个行为完全不具备性能可言；干扰浏览器对原型的优化；不支持 IE10 及以下的浏览器版本。

以上。

## 继承与原型链

对于使用过基于类的语言 (如 Java 或 C++) 的开发人员来说，JavaScript 有点令人困惑，因为它是动态的，并且本身不提供一个 class 实现。（在 ES2015/ES6 中引入了 class 关键字，但那只是语法糖，JavaScript 仍然是基于原型的）。

当谈到继承时，JavaScript 只有一种结构：对象。

每个**实例对象**（ object ）都有一个私有属性（称之为 \_\_proto\_\_ ）指向它的构造函数的原型对象（prototype ）。该原型对象也有一个自己的原型对象( \_\_proto\_\_ ) ，层层向上直到一个对象的原型对象为 null。根据定义，null 没有原型，并作为这个原型链中的最后一个环节。

几乎所有 JavaScript 中的对象都是位于原型链顶端的 Object 的实例。

尽管这种原型继承通常被认为是 JavaScript 的弱点之一，但是原型继承模型本身实际上比经典模型更强大。例如，在原型模型的基础上构建经典模型相当简单。

* 1. [基于原型链的继承](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Inheritance_and_the_prototype_chain#%E5%9F%BA%E4%BA%8E%E5%8E%9F%E5%9E%8B%E9%93%BE%E7%9A%84%E7%BB%A7%E6%89%BF)
     1. [继承属性](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Inheritance_and_the_prototype_chain#%E7%BB%A7%E6%89%BF%E5%B1%9E%E6%80%A7)

JavaScript 对象是动态的属性“包”（指其自己的属性）。JavaScript 对象有一个指向一个原型对象的链。当试图访问一个对象的属性时，它不仅仅在该对象上搜寻，还会搜寻该对象的原型，以及该对象的原型的原型，依次层层向上搜索，直到找到一个名字匹配的属性或到达原型链的末尾。

1. 遵循ECMAScript标准，someObject.[[Prototype]] 符号是用于指向 someObject 的原型。从 ECMAScript 6 开始，[[Prototype]] 可以通过 [Object.getPrototypeOf()](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Object/GetPrototypeOf) 和 [Object.setPrototypeOf()](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Object/setPrototypeOf) 访问器来访问。这个等同于 JavaScript 的非标准但许多浏览器实现的属性 \_\_proto\_\_。
2. 但它不应该与构造函数 func 的 prototype 属性相混淆。被构造函数创建的实例对象的 [[Prototype]] 指向 func 的 prototype 属性。Object.prototype属性表示 [Object](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Object) 的原型对象。

*// 让我们从一个函数里创建一个对象o，它自身拥有属性a和b的：*

let f = function () {

this.a = 1;

this.b = 2;

}

*/\* 这么写也一样*

*function f() {*

*this.a = 1;*

*this.b = 2;*

*}*

*\*/*

let o = new f(); *// {a: 1, b: 2}*

*// 在f函数的原型上定义属性*

f.prototype.b = 3;

f.prototype.c = 4;

*// 不要在 f 函数的原型上直接定义 f.prototype = {b:3,c:4};这样会直接打破原型链*

*// o.[[Prototype]] 有属性 b 和 c*

*// (其实就是 o.\_\_proto\_\_ 或者 o.constructor.prototype)*

*// o.[[Prototype]].[[Prototype]] 是 Object.prototype.*

*// 最后o.[[Prototype]].[[Prototype]].[[Prototype]]是null*

*// 这就是原型链的末尾，即 null，*

*// 根据定义，null 就是没有 [[Prototype]]。*

*// 综上，整个原型链如下:*

*// {a:1, b:2} ---> {b:3, c:4} ---> Object.prototype---> null*

console.log(o.a); *// 1*

*// a是o的自身属性，该属性的值为 1*

console.log(o.b); *// 2*

*// b是o的自身属性，该属性的值为 2*

*// 原型上也有一个'b'属性，但是它不会被访问到。这种情况被称为"属性遮蔽 (property shadowing)"*

console.log(o.c); *// 4*

*// c不是o的自身属性，去它的原型上找，发现c是o.[[Prototype]]的属性，该属性的值为 4*

console.log(o.d); *// undefined*

*// d 不是 o 的自身属性，去它的原型上找，发现d也不是 o.[[Prototype]] 的属性，再去它的原型的原型上找 o.[[Prototype]].[[Prototype]] 为 null，停止搜索*

*// 找不到 d 属性，所以返回 undefined*

* + 1. [继承方法](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Inheritance_and_the_prototype_chain#%E7%BB%A7%E6%89%BF%E6%96%B9%E6%B3%95)

JavaScript 并没有其他基于类的语言所定义的“方法”。在 JavaScript 里，任何函数都可以添加到对象上作为对象的属性。函数的继承与其他的属性继承没有差别，包括上面的“属性遮蔽”（这种情况相当于其他语言的方法重写）。

当继承的函数被调用时，[**this**](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/this)**指向的是当前继承的对象**，而不是继承的函数所在的原型对象。

var o = {

a: 2,

m: function(){

return this.a + 1;

}

};

console.log(o.m()); *// 3*

*// 当调用 o.m 时，'this' 指向了 o.*

var p = Object.create(o);*// p是一个继承自 o 的对象*

p.a = 4; *// 创建 p 的自身属性 'a'*

console.log(p.m()); *// 5*

*//调用 p.m 时，'this' 指向了 p，又因为 p 继承了 o 的 m 函数，所以返回5*

*//此时的 'this.a' 即 p.a，就是 p 的自身属性 'a'*

* + 1. [在 JavaScript 中使用原型](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Inheritance_and_the_prototype_chain#%E5%9C%A8_javascript_%E4%B8%AD%E4%BD%BF%E7%94%A8%E5%8E%9F%E5%9E%8B)

接下去，来仔细分析一下这些应用场景下， JavaScript 在背后做了哪些事情。

正如之前提到的，在 JavaScript 中，函数（function）是允许拥有属性的。所有的函数会有一个特别的属性 —— prototype 。

示例

function doSomething(){}

console.log( doSomething.prototype );

*// 和声明函数的方式无关，*

*// JavaScript 中的函数永远有一个默认原型属性。*

var doSomething = function(){};

console.log( doSomething.prototype );

doSomething函数的一个默认属性prototype。这段代码运行之后，控制台应该显示类似如下的结果：

{

constructor: ƒ doSomething(),

\_\_proto\_\_: {

constructor: ƒ Object(),

hasOwnProperty: ƒ hasOwnProperty(),

isPrototypeOf: ƒ isPrototypeOf(),

propertyIsEnumerable: ƒ propertyIsEnumerable(),

toLocaleString: ƒ toLocaleString(),

toString: ƒ toString(),

valueOf: ƒ valueOf()

}

}

## 创建对象的四种方法

1. 借用构造函数继承，使用call或apply方法，将父对象的构造函数绑定在子对象上
2. 原型继承，将子对象的prototype指向父对象的一个实例
3. 组合继承
   1. 字面量对象

比较常用的一种方式：

**let** obj **=** {};

* 1. 构造函数创建

构造函数创建的方式更多用来在Javascript中**实现继承，多态，封装**等特性。在Javascript中使用new关键字来对构造函数进行实例化。

**function** Animal(name) {

**this**.name **=** name;

}

**let** cat **=** **new** Animal('Tom');

* 1. class创建

class关键字是ES6新引入的一个特性，它其实是基于原型和原型链实现的一个语法糖。

**class** Animal {

constructor(name) {

**this**.name **=** name;

}

}

**let** cat **=** **new** Animal('Tom');

## 单线程、多线程

## 同步、异步

## 微任务、宏任务

事件循环：javascript的执行规则里面有个事件循环Event Loot的规则，在事件循环中，异步事件会放到异步队列里面，但是异步队列里面又分为宏任务和微任务，浏览器端的宏任务一般有：script标签,setTimeout,setInterval,setImmediate,requestAnimationFrame。

微任务有：MutationObserver,Promise.then catch finally。宏任务会阻塞浏览器的渲染进程，微任务会在宏任务结束后立即执行，在渲染之前。

## ES6的箭头函数与普通方法的区别

箭头函数

本身没有property属性，不能用于构造函数，不能new对象；

自身没有this上下文，当函数内使用this时，是调用这个箭头函数的对象；

没有函数提升；

## 变量、函数提升

## 闭包/内部函数

实现了私有化的属性和方法

缺点是，过多的使用内部函数会导致内存溢出。因为当外部方法没有执行完的时候，内部方法并不会释放内部定义的属性变量内存

## 内存回收机制

两种方式：标记清除、（引用）计数清除

## 事件代理

## 递归函数

## this当前执行上下文

***：call、apply、bind、箭头函数在 函数上下文 中的应用***

**前提：**

函数的 this 关键字在 JavaScript 中的表现，在严格模式和非严格模式之间会有一些差别。

在绝大多数情况下，函数的**调用方式**决定了 this 的值（**运行时绑定**）。this 不能在执行期间被赋值，并且在每次函数被调用时 this 的值也可能会不同。ES5 引入了 bind 方法来设置函数的 this 值，而不用考虑函数如何被调用的。ES2015 引入了箭头函数，箭头函数不提供自身的 this 绑定（this 的值将保持为闭合词法上下文的值）。

* 1. this的语法
     1. 值

**当前执行上下文**（global、function 或 eval）的一个属性，**在非严格模式下，总是指向一个对象，在严格模式下可以是任意值。**

* + 1. 全局上下文

无论是否在严格模式下，在全局执行环境中（**在任何函数体外部**）this 都指向全局对象。

示例：

// 在浏览器中, window 对象同时也是全局对象：

console.log(this === window); // true

a = 37;

console.log(window.a); // 37

this.b = "MDN";

console.log(window.b) // "MDN"

console.log(b) // "MDN"

上例中，说明在浏览器中，window对象 === 全局对象。

* + 1. 函数上下文

**在函数内部**，this的值取决于函数被调用的方式。

* + - 1. 执行函数时没有设置调用的对象

以下2个示例，演示在非严格模式和严格模式下，执行函数时没有设置调用的对象，默认指向的区别。

示例1：在非严格模式下，且 this 的值不是由该调用设置的，所以 this 的值默认指向全局对象，浏览器中就是 window ，如下：

function f1(){

return this;

}

//在浏览器中：

f1() === window; //在浏览器中，全局对象是window

//在Node中：

f1() === globalThis;

示例2：在严格模式下，如果进入执行环境时没有设置 this 的值，this 会保持为 undefined，如下：

function f2(){

"use strict"; // 这里是严格模式

return this;

}

f2() === undefined; // true

**在第2个例子中，this 应是 undefined，因为 f2 是被直接调用的，而不是作为对象的属性或方法调用的（如 window.f2()）。有一些浏览器最初在支持严格模式时没有正确实现这个功能，于是它们错误地返回了window对象。**

* + - 1. call、apply、bind 改变函数上下文

如果要想把 this 的值从一个环境传到另一个，就要用 call 或者apply 方法，如下：

1. [函数上下文中的 this](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/this#%E5%87%BD%E6%95%B0%E4%B8%8A%E4%B8%8B%E6%96%87%E4%B8%AD%E7%9A%84_this)

函数上下文中的 this 的值，是调用该函数的对象。可以用 call 、apply 、bind方法，把 this 的值从一个环境传到另一个。

*// 对象可以作为 bind 或 apply 的第一个参数传递，并且该参数将绑定到该对象。*

var obj = {a: 'Custom'};

*// 声明一个变量，并将该变量作为全局对象 window 的属性。*

var a = 'Global';

function whatsThis() {

return this.a;  *// this 的值取决于函数被调用的方式*

}

*// 因为在这个函数中 this 没有被设定，所以它默认为 全局 或 window 对象*

whatsThis(); *// 'Global'*

whatsThis.call(obj); *// 'Custom' 因为函数中的 this 被设置为obj*

whatsThis.apply(obj); *// 'Custom' 因为函数中的 this 被设置为obj*

1. [this 和 对象 转换](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/this#this_%E5%92%8C%E5%AF%B9%E8%B1%A1%E8%BD%AC%E6%8D%A2)

函数中的this的设定是 对象 类型。

function add(c, d) {

return this.a + this.b + c + d;

}

var o = {a: 1, b: 3};

*// 第一个参数是用作“ this ”的对象*

*// 其余参数用作函数的参数*

add.call(o, 5, 7); *// 16*

*// 第一个参数是用作“ this ”的对象*

*// 第二个参数是一个 数组，数组中的两个成员用作函数参数*

add.apply(o, [10, 20]); *// 34*

在非严格模式下使用 call 和 apply 时，如果用作 this 的值不是对象，则会被尝试转换为对象。null 和 undefined 被转换为全局对象。

原始值如 7 或 'foo' 会使用相应构造函数转换为对象。因此 7 会被转换为 new Number(7) 生成的对象，字符串 'foo' 会转换为 new String('foo') 生成的对象。

function bar() {

console.log(Object.prototype.toString.call(this));

}

bar.call(7); *// [object Number]*

bar.call('foo'); *// [object String]*

bar.call(undefined); *// [object global]*

1. [bind方法](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/this#bind%E6%96%B9%E6%B3%95)

ECMAScript 5 引入了 [Function.prototype.bind()](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Function/bind)。调用f.bind(someObject)会创建一个与f具有相同函数体和作用域的函数，但是在这个新函数中，this将永久地被绑定到了bind的第一个参数，无论这个函数是如何被调用的。

function f(){

return this.a;

}

var g = f.bind({a:"azerty"}); // 此时函数 f 没有被执行

console.log(g()); // azerty ， 函数 f 被执行了

var h = g.bind({a:'yoo'}); // **bind只生效一次**！

console.log(h()); // azerty

var o = {a:37, f:f, g:g, h:h};

console.log(o.a, o.f(), o.g(), o.h()); // 37, 37, azerty, azerty

* + - 1. [箭头函数](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/this#%E7%AE%AD%E5%A4%B4%E5%87%BD%E6%95%B0)

在[箭头函数](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Functions/Arrow_functions)中，this与封闭词法环境的this保持一致。在全局代码中，它将被设置为全局对象。

var globalObject = this;

var foo = (() => this);

console.log(foo() === globalObject); // true

注意：如果将this传递给call、bind、或者apply来调用箭头函数，它将被忽略。为调用添加参数时，第一个参数（thisArg）应该设置为null。

接着上面的代码：

// foo函数 作为对象的一个方法调用

var obj = {foo: foo};

console.log(obj.foo() === globalObject); // true

// 尝试使用call来设定this

console.log(foo.call(obj) === globalObject); // true

// 尝试使用bind来设定this

foo = foo.bind(obj);

console.log(foo() === globalObject); // true

无论如何，**foo 的 this 被设置为他被创建时的环境**（在上面的例子中，就是全局对象）。

这同样适用于在其他函数内创建的箭头函数：这些箭头函数的this被设置为封闭的词法环境的。

// 创建一个含有bar方法的obj对象，bar返回一个函数，这个函数返回this，

// 这个[返回的函数]是箭头函数创建的，所以它的this被永久绑定到了它外层函数bar的调用者上。

// bar的值可以在调用中设置，这反过来又设置了返回函数的值。

var obj = {

bar: function() {

var x = (() => this);

return x;

}

};

var fn1 = obj.bar(); // obj对象**调用bar方法**，把bar的this绑定到obj上，将该 [返回的函数] 赋值给fn。

// 直接调用fn1 即[返回函数]，而不重新设置调用fn1的对象，则由箭头函数创建的[返回函数]内部的this任然指向obj。

console.log(fn1() === obj); // true

// 但是注意，如果只是引用obj的方法，而没有调用它

var fn11 = obj.bar;

// 那么调用箭头函数后，this指向window，因为它从 bar 继承了this。

console.log(fn11()() == window); // true

// --------------------------------------------------

var name = 'window'; // 全局变量

var obj2 = {

name: 'obj2 name', // 对象属性

bar: function() {

var x = (() => this.name);

return x;

}

};

// 只是引用obj2的方法，没有调用它

var fn2 = obj2.bar;

console.log(fn2()); // [Function: x]

// 没有设置调用对象，则默认为全局对象调用，若在非严格模式为undefined，严格模式下直接报错

console.log(fn2()()); // undefined

var fn22 = obj2.bar(); // obj2返回bar方法调用返回值：[返回函数]

console.log(fn22); // [Function: x]

console.log(fn22()); // obj2 name ， 调用[返回函数]的调用返回值

// --------------------------------------------------

// 重新设定对象obj3

var obj3 = {

name: 'obj3 name'

}

obj3.bar1 = obj2.bar;

console.log(obj3.bar1()); // [Function: x]

console.log(obj3.bar1()()); // obj3 name

obj3.bar2 = obj2.bar();

console.log(obj3.bar2()); // obj2 name

分析：

在上面的例子中，一个赋值给了 obj.bar的函数（称为匿名函数 A），返回了另一个箭头函数（称为匿名函数 B）。因此，在 A 调用时，函数B的this被永久设置为obj.bar（函数A）的this。当返回的函数（函数B）被调用时，它this始终是最初设置的。在上面的代码示例中，函数B的this被设置为函数A的this，即obj，所以即使被调用的方式通常将其设置为 undefined 或全局对象（或者如前面示例中的其他全局执行环境中的方法），它的 this 也仍然是 obj 。

当创建新对象obj3时，在给obj3.bar1 和 obj3.bar2 赋值时，因为赋值方式的区别，可以看出引用对象的改变。

* + - 1. [作为对象的方法](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/this#%E4%BD%9C%E4%B8%BA%E5%AF%B9%E8%B1%A1%E7%9A%84%E6%96%B9%E6%B3%95)

当函数作为对象里的方法被调用时，this 被设置为调用该函数的对象。

下面的例子中，当 o.f() 被调用时，函数内的 this 将绑定到 o 对象。

var o = {

prop: 37,

f: function() {

return this.prop;

}

};

console.log(o.f()); // 37

请注意，这样的行为完全不会受函数定义方式或位置的影响。在前面的例子中，我们在定义对象o的同时，将其中的函数定义为成员 f 。但是，我们也可以先定义函数，然后再将其附属到o.f。这样做的结果是一样的：

var o = {prop: 37};

function independent() {

return this.prop;

}

o.f = independent;

console.log(o.f()); // 37

这表明函数是从 o 的 f 成员调用的才是重点。

同样，this 的绑定只受最接近的成员引用的影响。在下面的这个例子中，我们把一个方法g当作对象o.b的函数调用。在这次执行期间，函数中的this将指向o.b。事实证明，这与他是对象 o 的成员没有多大关系，最近的引用才是最重要的。

o.b = {g: independent, prop: 42}; // 赋值{g: independent, prop: 42}是一个完整对象，此时this绑定的是该对象

console.log(o.b.g()); // 42

* + - 1. 原型链中的 this

对于在对象原型链上某处定义的方法，同样的概念也适用。如果该方法存在于一个对象的原型链上，那么 this 指向的是调用这个方法的对象，就像该方法就在这个对象上一样。

var o = {

f: function() {

return this.a + this.b;

}

};

var p = Object.create(o);

p.a = 1;

p.b = 4;

console.log(p.f()); // 5

在这个例子中，对象 p 没有属于它自己的 f 属性，它的 f 属性继承自它的原型。虽然最终是在 o 中找到 f 属性的，这并没有关系；查找过程首先从 p.f 的引用开始，所以函数中的 this 指向p。也就是说，因为f是作为p的方法调用的，所以它的this指向了p。这是 JavaScript 的原型继承中的一个有趣的特性。

* + - 1. getter 与 setter 中的 this

相同的概念也适用于当函数在一个 getter 或者 setter 中被调用。用作 getter 或 setter 的函数都会把 this 绑定到设置或获取属性的对象。

function sum() {

return this.a + this.b + this.c;

}

var o = {

a: 1,

b: 2,

c: 3,

get average() {

return (this.a + this.b + this.c) / 3;

}

};

Object.defineProperty(o, 'sum', {

get: sum, enumerable: true, configurable: true});

console.log(o.average, o.sum); // logs 2, 6

* + - 1. [作为构造函数](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/this#%E4%BD%9C%E4%B8%BA%E6%9E%84%E9%80%A0%E5%87%BD%E6%95%B0)

当一个函数用作构造函数时（使用[new](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/new)关键字），它的this被绑定到正在构造的新对象。

虽然构造函数返回的默认值，是 this 所指的那个对象，但它仍可以手动返回其他的对象（如果返回值不是一个对象，则返回 this 对象）。

/\*

\* 构造函数这样工作:

\* function MyConstructor(){

\* // 函数实体写在这里

\* // 根据需要在this上创建属性，然后赋值给它们，比如：

\* this.fum = "nom";

\* // 等等...

\* // 如果函数具有返回对象的return语句，则该对象将是 new 表达式的结果。

\* // 否则，表达式的结果是当前绑定到 this 的对象。

\* //（即通常看到的常见情况）。

\* }

\*/

function C(){

this.a = 37;

}

var o = new C();

console.log(o.a); // 37

function C2(){

this.a = 37;

return {a:38};

}

o = new C2();

console.log(o.a); // 38

在例子中C2，因为在调用构造函数的过程中，重新设置了返回对象 {a:38} ，与this绑定的默认属性 a 被丢弃了。使得语句 this.a = 37; 成了“僵尸”代码，但实际上这条语句执行了，只是对于外部没有任何影响，因此完全忽略了它。

* + - 1. [作为一个DOM事件处理函数](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/this#%E4%BD%9C%E4%B8%BA%E4%B8%80%E4%B8%AAdom%E4%BA%8B%E4%BB%B6%E5%A4%84%E7%90%86%E5%87%BD%E6%95%B0)

在浏览器中，当函数被用作事件处理函数时，它的 this 指向触发事件的元素（一些浏览器在使用非 addEventListener 的函数动态地添加监听函数时不遵守这个约定）。

// 被调用时，将关联的元素变成蓝色

function bluify(e){

console.log(this === e.currentTarget); // 总是 true

// 当 currentTarget 和 target 是同一个对象时为 true

console.log(this === e.target);

this.style.backgroundColor = '#A5D9F3';

}

// 获取文档中的所有元素的列表

var elements = document.getElementsByTagName('\*');

// 将bluify作为元素的点击监听函数，当元素被点击时，就会变成蓝色

for(var i=0 ; i<elements.length ; i++){

elements[i].addEventListener('click', bluify, false);

}

* + - 1. [作为一个内联事件处理函数](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/this#%E4%BD%9C%E4%B8%BA%E4%B8%80%E4%B8%AA%E5%86%85%E8%81%94%E4%BA%8B%E4%BB%B6%E5%A4%84%E7%90%86%E5%87%BD%E6%95%B0)

在浏览器中，当代码被内联 [on-event 处理函数](https://developer.mozilla.org/zh-CN/docs/orphaned/Web/Guide/Events/Event_handlers) 调用时，它的this指向监听器所在的DOM元素：

<button onclick="alert( this.tagName.toLowerCase() );">

Show this

</button>

上面的 alert 会显示 button。

<button onclick="alert( ( function(){return this} )() );">

Show inner this

</button>

上面的 alert 会显示 [object Window]。

在这种情况下，**没有设置内部函数的 this，所以它指向 global/window 对象**（即非严格模式下调用的函数未设置 this 时指向的默认对象）。

* + 1. 类上下文

this **在 类 中**的表现与在函数中类似，因为类本质上也是函数，但也有一些区别和注意事项。

**在类的构造函数中，this 是一个常规对象。**类中所有非静态的方法都会被添加到 this 的原型中：

class Example {

constructor() {

const proto = Object.getPrototypeOf(this);

console.log(Object.getOwnPropertyNames(proto));

}

first(){}

second(){}

static third(){} // 静态方法

}

new Example(); // ['constructor', 'first', 'second']

注意：静态方法不是 this 的属性，它们只是类自身的属性。

1. [类中的this](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/this#%E7%B1%BB%E4%B8%AD%E7%9A%84this)

和其他普通函数一样，方法中的 this 值取决于它们如何被调用。

有时，改写这个行为，让类中的 this 值总是指向这个类实例会很有用。为了做到这一点，可在构造函数中绑定类方法：

class Car {

constructor() {

// Bind sayBye but not sayHi to show the difference

this.sayBye = this.sayBye.bind(this);

}

sayHi() {

console.log(`Hello from ${this.name}`);

}

sayBye() {

console.log(`Bye from ${this.name}`);

}

get name() {

return 'Ferrari';

}

}

class Bird {

get name() {

return 'Tweety';

}

}

const car = new Car();

const bird = new Bird();

// The value of 'this' in methods depends on their caller

car.sayHi(); // Hello from Ferrari

bird.sayHi = car.sayHi;

bird.sayHi(); // Hello from Tweety

// For bound methods, 'this' doesn't depend on the caller

bird.sayBye = car.sayBye;

bird.sayBye(); // Bye from Ferrari

**注意：类内部总是严格模式。**调用一个 this 值为 undefined 的方法会抛出错误。

* + 1. 派生类

不像基类的构造函数，派生类的构造函数没有初始的 this 绑定。

在构造函数中调用 super() 会生成一个 this 绑定，并相当于执行如下代码，Base为基类：

this = new Base();

**警告：**在调用 super() 之前引用 this 会抛出错误。

派生类不能在调用 super() 之前返回，除非其构造函数返回的是一个对象，或者根本没有构造函数。

class Base {}

class Good extends Base {} // 继承

class AlsoGood extends Base {

// 继承，并创建构造函数，返回一个新对象

constructor() {

return {a: 5};

}

}

class Bad extends Base {

// 继承，并创建一个空构造函数

constructor() {}

}

new Good();

new AlsoGood();

new Bad(); // ReferenceError

call、apply、bind的应用

箭头函数的应用

## for、for…in、for…of、forEach

for数组，可以用break

for…in数组、Object…key，不可以用break

for…of数组，不可以用break

ArrayObj.foreach((obj,index,ArrayObj)=>{})数组，不可以用break

* 1. 性能优先级

for循环 > for-of > forEach > for-in



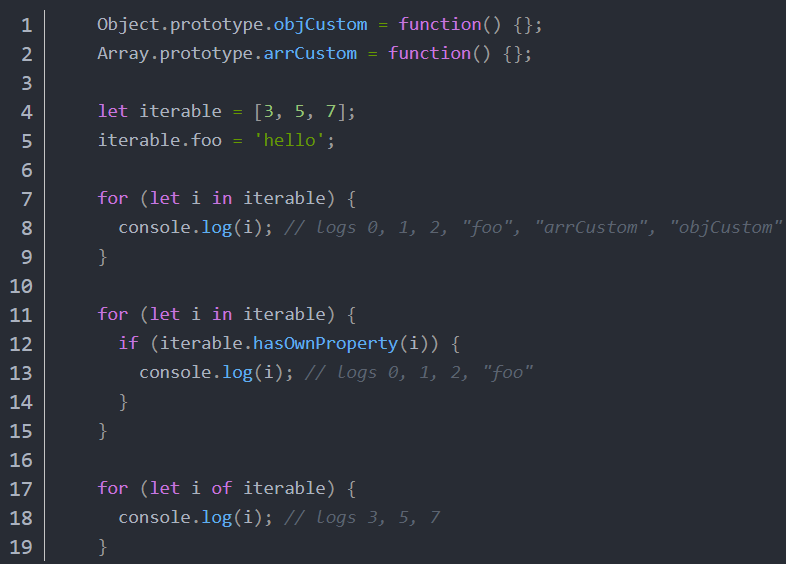
注解：缓存数组长度的for循环是最优的。但是在chrome的v8引擎中对数组长度的缓存做了优化，只要数组长度不变，for循环不会每次去计算数组长度的，因此缓存数组长度可能在基于ie内核的浏览器里可能会有性能的提升。这并不意味着为了追求性能就要使用for循环。还是要看情况，毕竟现在硬件的发展，对js的性能的要求也不像以前那么偏执了。

* 1. 应用场景
* **forEach只能用于数组**，在iteration中无法返回，所以every和some来取代它。
* for in 语句用于迭代对象的可枚举属性，至于迭代顺序需要注意使用场景。
* for of 可以遍历**所有可迭代对象**。
* 和for of的区别，for in遍历的值是数组的索引。

1. for…in

* 根据 ECMA-262（ECMAScript）第三版中描述，for-in 语句的属性遍历的顺序是**由对象定义时属性的书写顺序决定的**。
* 在现有最新的 ECMA-262（ECMAScript）第五版规范中，对 for-in 语句的遍历机制又做了调整，**属性遍历的顺序是没有被规定的**。

对于 **原始插入顺序** 的理解，是按照继承和原型链的顺序的。以数组类型对象举例：自身 -> Array.prototype -> Object.prototype



始终是自身的属性先于数组原型的属性再是对象原型的属性这个顺序。

**解决方案**

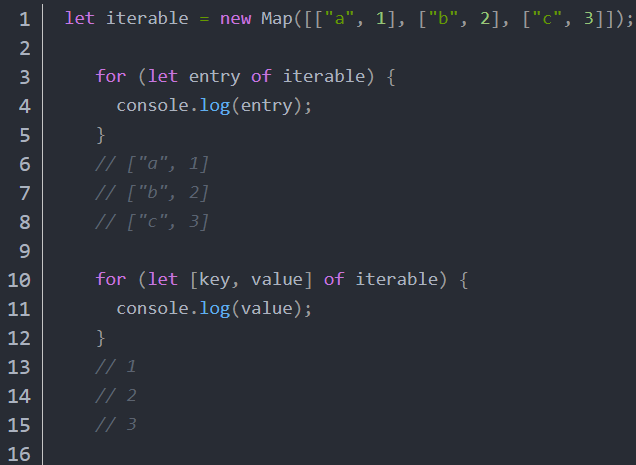
for-in 语句无法保证遍历顺序，应尽量避免编写依赖对象属性顺序的代码。如果想顺序遍历一组数据，请使用数组并使用 for 语句遍历。 如果想按照定义的次序遍历对象属性，请参考本文针对各浏览器编写特殊代码。

1. for…of

for...of**语句**在[可迭代对象](https://links.jianshu.com/go?to=https%3A%2F%2Fdeveloper.mozilla.org%2Fzh-CN%2Fdocs%2FWeb%2FJavaScript%2FGuide%2Fiterable)（ [Array](https://links.jianshu.com/go?to=https%3A%2F%2Fdeveloper.mozilla.org%2Fzh-CN%2Fdocs%2FWeb%2FJavaScript%2FReference%2FArray)，[Map](https://links.jianshu.com/go?to=https%3A%2F%2Fdeveloper.mozilla.org%2Fzh-CN%2Fdocs%2FWeb%2FJavaScript%2FReference%2FMap)，[Set](https://links.jianshu.com/go?to=https%3A%2F%2Fdeveloper.mozilla.org%2Fzh-CN%2Fdocs%2FWeb%2FJavaScript%2FReference%2FGlobal_Objects%2FSet)，[String](https://links.jianshu.com/go?to=https%3A%2F%2Fdeveloper.mozilla.org%2Fzh-CN%2Fdocs%2FWeb%2FJavaScript%2FReference%2FString)，[TypedArray](https://links.jianshu.com/go?to=https%3A%2F%2Fdeveloper.mozilla.org%2Fzh-CN%2Fdocs%2FWeb%2FJavaScript%2FReference%2FGlobal_Objects%2FTypedArray)，[arguments](https://links.jianshu.com/go?to=https%3A%2F%2Fdeveloper.mozilla.org%2Fen-US%2Fdocs%2FWeb%2FJavaScript%2FReference%2FFunctions_and_function_scope%2Farguments) 对象等等）上***创建一个迭代循环调用自定义迭代钩子***，并为每个不同属性的值执行语句。

for…of迭代出的数据是基于可迭代对象的默认迭代器的。这一点和 …扩展运算符 分割元素异曲同工。

以Map为例



* 1. 针对break，continue， return 的表现
* return

return是表示从被调函数返回到主调函数继续执行，返回时可附带一个返回值，由return后面的参数指定。**return后函数就结束了**，后面的语句不再执行

* break

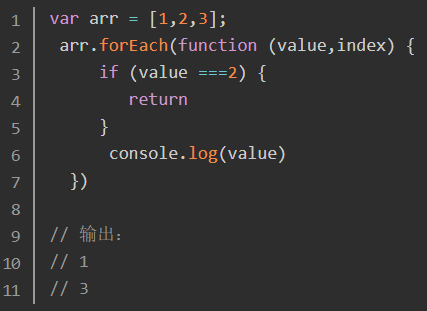
此语句导致程序终止包含它的循环，开始执行整个循环后面的语句。例如：如果 break 包含嵌套循环里，**它只跳出最里面的循环**。

* continue

循环执行到此语句，不再执行continue后面的语句而是**执行下一次循环，并不会终止包含他的循环**

* for in 循环： break，continue， return 的表现跟 for 循环一致。
* forEach里面不能使用 break 和 continue ，否则会报错。
* for of的循环，break，continue， return 的表现跟 for 循环一致。

注：但是可以使用 return，return 的效果是跳出本次循环的函数回调，并不会跳出整个循环。



## some、every、map、filter

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 方法 | 判断条件 | 返回值 | 检测空数组 | 改变原始数组 |
| every | 所有元素都符合指定条件 | true | N | N |
| some | 有至少一个元素满足指定条件 | true | N | N |
| map | 返回原始数组每个元素调用自定义函数处理后的值 | 新数组 | N | N |
| fiter | 返回原始数组符合自定义函数（筛选条件）处理后的元素的值 | 新数组 |  |  |

指定函数处理数组的每个元素

every() 是对数组中每一项运行给定的自定义函数，如果该函数对***每一项***返回***true***，则返回true。**every从迭代开始，一旦有一个不符合条件，则不会继续迭代下去。**不会对空数组进行检测，不会改变原来的数组。

some() 是对数组中每一项运行给定的自定义函数，如果该函数对***任一项***返回***true***，则返回true。**some一直在找符合条件的值，一旦找到，则不会继续迭代下去。**不会对空数组进行检测，不会改变原来的数组。

var arr = [ 1, 2, 3, 4, 5, 6 ];

console.log( arr.some( function( item, index, array ){

console.log('item=' + item + ',index=' + index+',array=' + array);

return item > 3;

}));

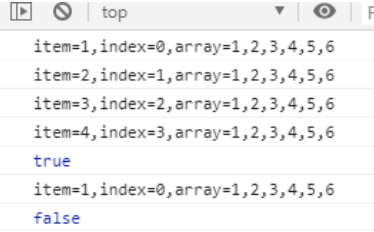
console.log( arr.every( function( item, index, array ){

console.log('item=' + item + ',index=' + index+',array=' + array);

return item > 3;

}));

运行结果：



## splice、split

splice按照指定字符串标识进行分割，返回新数组，不改变原数组

split截取

## Ajax轮询

网站实现的推送技术，实现方式就是ajax轮询。

在指定时间间隔，由浏览器向服务器端发起http请求，然后服务器端将最新数据返回。

缺点：服务器端的数据是否更新都会被返回，~~在发起http请求时，包含的请求头部信息可能会很长~~，这样就占用了宽度和服务器端的资源

# ES6

## Generator函数、asycn函数、Promise函数

*generator函数、async/await函数、promise函数*

*1、async await基于generator函数的语法糖，await后面必然是一个promise对象，await可以等待异步函数执行完毕再继续执行后面的代码，可以控制代码的执行顺序；*

*2、promise将多个回调函数嵌套的回调地狱 ，变成了链式的写法；*

*3、简单的说async函数就相当于自执行的Generator函数，相当于自带一个状态机，在await的部分等待返回， 返回后自动执行下一步。而且相较于Promise，async的优越性就是把每次异步返回的结果从then中拿到最外层的方法中，不需要链式调用，只要用同步的写法就可以了。更加直观而且，更适合处理并发调用的问题。但是async必须以一个Promise对象开始 ，所以async通常是和Promise结合使用的。*

* 1. Generator函数

ES6的新特性，是异步编程常用的解决方案，通过yied关键字，可以让函数的执行流挂起，那么便为改变执行流程提供了可能。

* Generator语法：

1.Generator函数有两个特征，一是function关键字和函数名之间有一个\*；二是具体函数内部使用yield表达式来定义不同的内部状态。

2.Generator函数的调用方法与普通函数一样，也是在函数名后面加上（），但不同的是，调用Generator函数后，该函数并不执行，返回的也不是函数运行结果，而是一个指向内部的指针对象（即遍历器对象Iterator Object，有value和done两个属性，value属性表示当前的内部状态值即yield表达式的值；done属性表示是否遍历结束）。要想该函数执行就必须调用遍历器对象的next方法，使指针移向下一个状态。

function \* generatorDemo（）{

yield "Hello";

yield "World";

return "ending";

}

let grenDemo = generatorDome()

grenDemo.next(); //{value:'Hello',done:false}

grenDemo.next(); //{value:'World',done:false}

grenDemo.next(); //{value:'ending',done:true}

grenDemo.next(); //{value:undefined,done:true}

* 1. async函数
* 含义：

async就是Generator函数的语法糖

* 语法：

在function关键字前加上 async表示函数里有异步操作，具体函数内部使用await表示紧跟在后面的表达式需要等待结果

* 基本用法：

async函数返回一个Promise对象，可以使用then方法添加回调函数。当函数执行的时候一旦遇到await就会先返回，等到异步操作完成再接着执行函数体内后面的语句。

（1）resolve状态：async函数内部return语句返回的值会成为then方法回调函数的参数。

async function getStockPriceByName(name) {

const symbol = await getStockSymbol(name);

const stockPrice = await getStockPrice(symbol);

return stockPrice;

}

getStockPriceByName('goog').then(function (result) {

console.log(result);

});

（2）reject状态：await命令后面的Promise对象如果变成了reject状态，那么整个async函数都会中断执行，后面的await语句不会再执行，reject的参数也会被catch方法的回调函数接收到。

async function f() {

await Promise.reject('出错了');

await Promise.resolve('hello world'); // 不会执行

}

f().then(v => console.log(v))

.catch(e => console.log(e)) // 出错了

（3）如果我们希望前一个异步操作失败也不要中断后面的异步操作，就可以将await放在try...catch结构里面，这样不管异步操作是否成功，第二个await都会执行

async function f() {

try {

await Promise.reject('出错了');

} catch(e) { }

return await Promise.resolve('hello world');

}

f().then(v => console.log(v)) // hello world

# vue

*react和vue的异同*

*相似点：都是SPA应用；都是操作虚拟DOM*

*不同点：vue是MVVM模式中的vm模型，react是MVC模式中的View模型；vue是数据双向绑定模式，react是单向的；*

*当应用状态改变时，vue由观察者和订阅者模式组合实现，更快的操作虚拟DOM比较需要更新的子组件，不会渲染整个组件树， 而react是全部子组件都会重新渲染，或者调用钩子函数shouldCompontentUpdate来实现局部渲染。*

*Virtual DOM是一个映射真实DOM的JavaScript对象，如果需要改变任何元素的状态，那么是先在Virtual DOM上进行改变，而不是直接改变真实的DOM。*

*当新一项被加进去这个JavaScript对象时，一个函数会计算新旧Virtual DOM之间的差异并反应在真实的DOM上。计算差异的算法是高性能框架的秘密所在，React和Vue在实现上有点不同。 Vue宣称可以更快地计算出Virtual DOM的差异，这是由于它在渲染过程中，会跟踪每一个组件的依赖关系，不需要重新渲染整个组件树。 而对于React而言，每当应用的状态被改变时，全部子组件都会重新渲染。当然，这可以通过shouldComponentUpdate这个生命周期方法来进行控制，但Vue将此视为默认的优化。 小结：如果你的应用中，交互复杂，需要处理大量的UI变化，那么使用Virtual DOM是一个好主意。如果你更新元素并不频繁，那么Virtual DOM并不一定适用，性能很可能还不如直接操控DOM。*

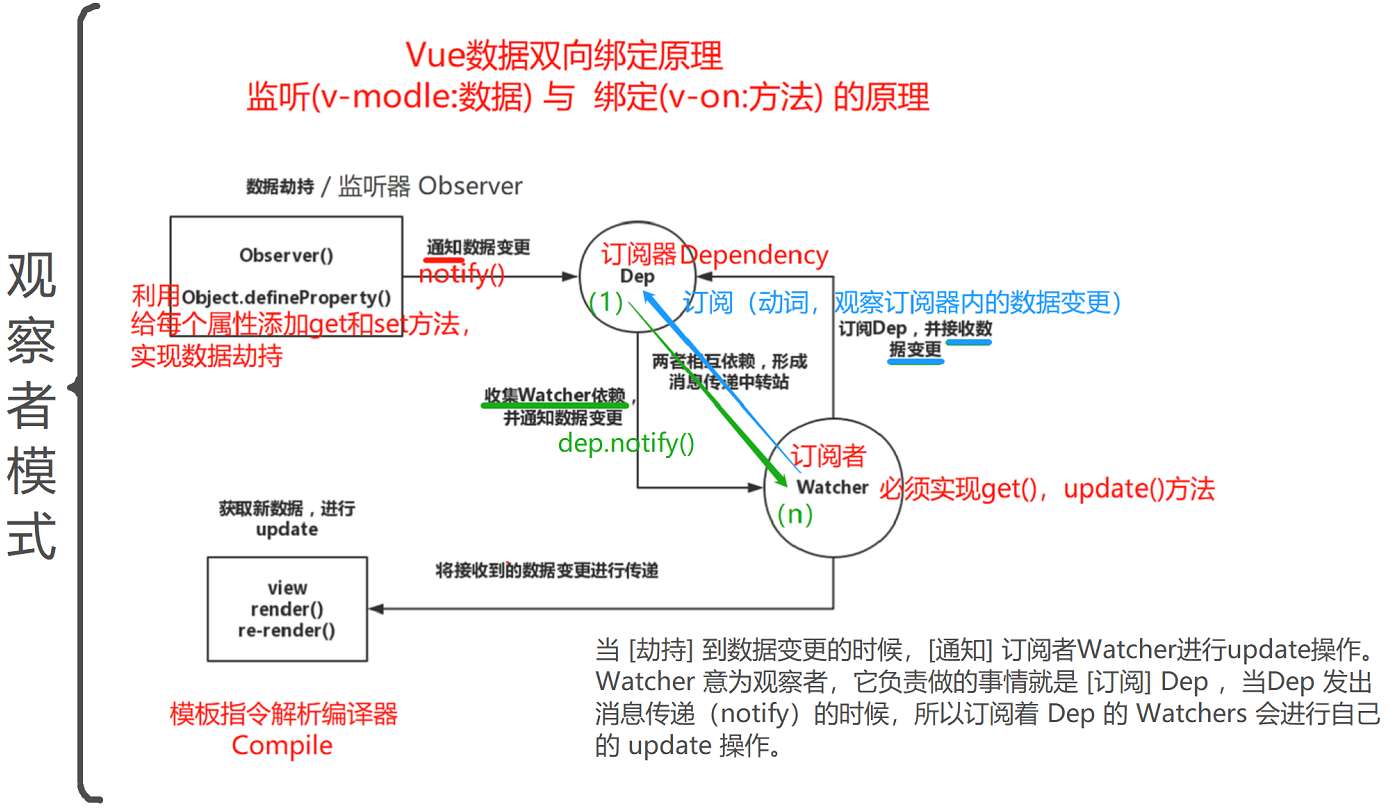
## vue数据双向绑定原理/响应式原理

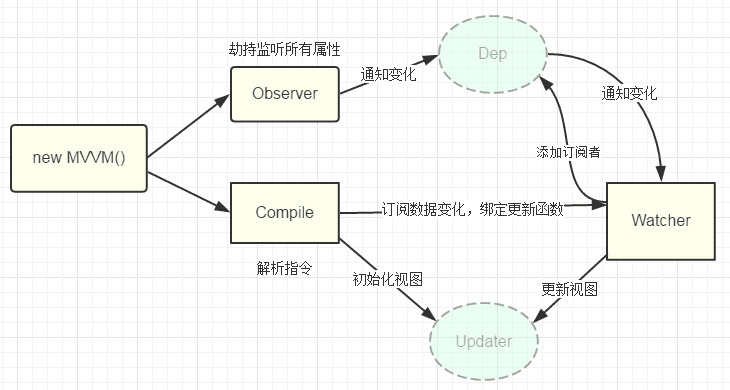
* 1. 响应式原理

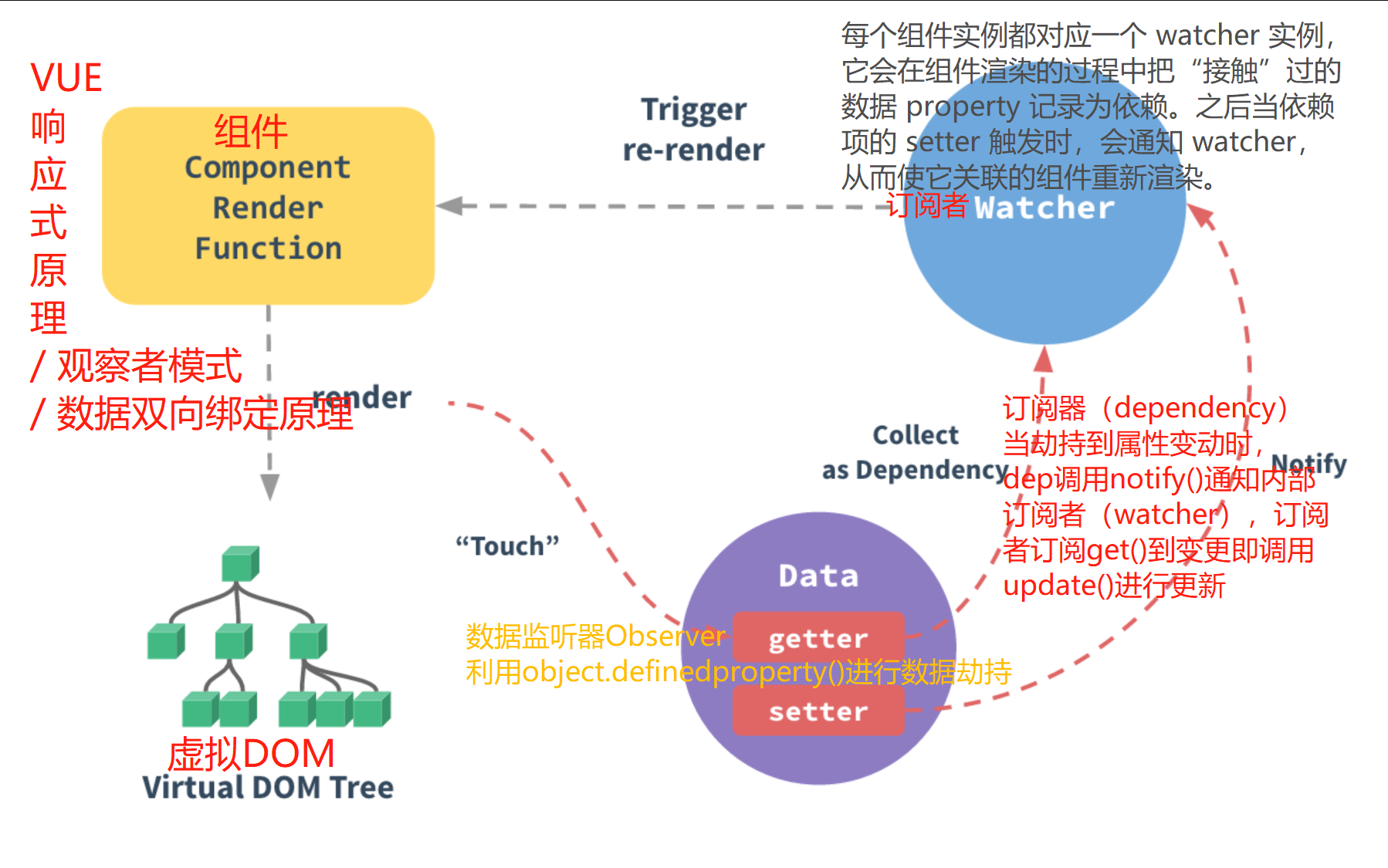
把一个普通的 JavaScript 对象传入 Vue 实例作为 data 选项，Vue 将遍历此对象所有的 property，并使用 [Object.defineProperty](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty) 把这些 property 全部转为 [getter/setter](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Working_with_Objects#%E5%AE%9A%E4%B9%89_getters_%E4%B8%8E_setters)。

注：Object.defineProperty 是 ES5 中一个无法 shim 的特性，这也就是 Vue 不支持 IE8 以及更低版本浏览器的原因。

每个组件实例都对应一个 **watcher** 实例，它会在组件渲染的过程中把“接触”过的数据 property 记录为依赖。之后当依赖项的 setter 触发时，会通知 watcher，从而使它关联的组件重新渲染。







* 1. 数据监听器/数据劫持 Observer

Vue当中的观察者模式

**观察者模式**： 当一个变量值被修改时，可以自动通知所有关注这个变量的其他对象，自动重新更新获取这个变量的新值。

* 1. 订阅者Watcher
  2. 模板解析编译器 Compile
  3. 检测变化的注意事项

由于 JavaScript 的限制，**Vue 不能检测数组和对象的变化**。因此需要一些办法来回避这些限制并保证它们的响应性。

* + 1. [对象](https://cn.vuejs.org/v2/guide/reactivity.html#%E5%AF%B9%E4%BA%8E%E5%AF%B9%E8%B1%A1)

**Vue 无法检测 property 的添加或移除。**由于 Vue 会在初始化实例时对 property 执行 getter/setter 转化（使用 [Object.defineProperty](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty) 把这些 property 全部转为 [getter/setter](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Working_with_Objects#%E5%AE%9A%E4%B9%89_getters_%E4%B8%8E_setters)），所以 property 必须在 data 对象上存在才能让 Vue 将它转换为响应式的。例如：

var vm = new Vue({

data:{

a:1 //在data内的对象，property已经转换为get/setter

}

})

// `vm.a` 是响应式的

vm.b = 2 // `vm.b` 是非响应式的，未添加在data中，没有转换property

**注：对于已经创建的实例，Vue 不允许动态添加根级别的响应式 property。**

1. 为已有对象添加单个新property

可以使用 Vue.set(object, propertyName, value) 方法向嵌套对象添加响应式 property。例如：

Vue.set(vm.someObject, 'b', 2)

还可以使用 vm.$set 实例方法，这也是全局 Vue.set 方法的别名：

this.$set(this.someObject,'b',2)

1. 为已有对象赋值多个新 property

可以使用 Object.assign() 或 \_.extend()。但是，这样添加到对象上的新 property 不会触发更新。在这种情况下，可以用原对象与要混合进去的对象的 property 一起创建一个新的对象。

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

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

* + 1. [数组](https://cn.vuejs.org/v2/guide/reactivity.html#%E5%AF%B9%E4%BA%8E%E6%95%B0%E7%BB%84)

Vue 不能检测以下数组的变动：

* 利用索引直接设置一个数组项时，例如：vm.items[indexOfItem] = newValue
* 修改数组的长度时，例如：vm.items.length = newLength

举例：

var vm = new Vue({

data: {

items: ['a', 'b', 'c']

}

})

vm.items[1] = 'x' // 不是响应性的

vm.items.length = 2 // 不是响应性的

1. 给数组元素赋值

为实现 vm.items[indexOfItem] = newValue 的目的，同时也将在响应式系统内触发状态更新，使用以下2种方法：

Vue.set(vm.items, indexOfItem, newValue) // Vue.set

vm.items.splice(indexOfItem, 1, newValue) // Array.prototype.splice

也可以使用 [vm.$set](https://cn.vuejs.org/v2/api/#vm-set) 实例方法，该方法是全局方法 Vue.set 的一个别名：

vm.$set(vm.items, indexOfItem, newValue)

也可以使用 splice：

vm.items.splice(newLength)

* 1. [声明响应式 property](https://cn.vuejs.org/v2/guide/reactivity.html#%E5%A3%B0%E6%98%8E%E5%93%8D%E5%BA%94%E5%BC%8F-property)

由于 Vue 不允许动态添加根级响应式 property，所以必须在初始化实例前声明所有根级响应式 property，哪怕只是一个空值：

var vm = new Vue({

data: {

message: '' // 声明 message 为一个空值字符串

},

template: '<div>{{ message }}</div>'

})

vm.message = 'Hello!' // 之后设置 `message`，符合响应式原理，message更新

如果初始化前，未在 data 选项中声明 message，Vue 将警告渲染函数render正在试图访问不存在的 property。

* 1. 维护响应式原理的必要性

这样的限制在背后是有其技术原因的，它消除了在依赖项跟踪系统中的一类边界情况，也使 Vue 实例能更好地配合类型检查系统工作。但与此同时在代码可维护性方面也有一点重要的考虑：data 对象就像组件状态的结构 (schema)。提前声明所有的响应式 property，可以让组件代码在未来修改或给其他开发人员阅读时更易于理解。

* 1. [异步更新队列](https://cn.vuejs.org/v2/guide/reactivity.html#%E5%BC%82%E6%AD%A5%E6%9B%B4%E6%96%B0%E9%98%9F%E5%88%97)

**Vue 在更新 DOM 时是异步执行的。**

只要侦听到数据变化，Vue 将开启一个队列，并缓冲在同一事件循环中发生的所有数据变更。如果同一个 watcher 被多次触发，只会被推入到队列中一次。这种在缓冲时去除重复数据对于避免不必要的计算和 DOM 操作是非常重要的。然后，在下一个的事件循环“tick”中，Vue 刷新队列并执行实际 (已去重的) 工作。

Vue 在内部对异步队列尝试使用原生的 Promise.then、MutationObserver和 setImmediate，如果执行环境不支持，则会采用 setTimeout(fn, 0) 代替。

例如，当设置 vm.someData = 'new value'，该组件不会立即重新渲染。当刷新队列时，组件会在下一个事件循环“tick”中更新。

多数情况不需要关心组件不会立即重新渲染的问题，但是想基于更新后的 DOM 状态做点进行其他操作，就存在问题了。

Vue.js 通常鼓励开发人员使用“数据驱动”的方式思考，避免直接接触 DOM。

为了在数据变化之后等待 Vue 完成更新 DOM后，再进行下一步操作，可以在数据变化之后立即使用 Vue.nextTick(callback)。这样回调函数将在 DOM 更新完成后被调用。例如：

<div id="example">{{message}}</div>

var vm = new Vue({

el: '#example',

data: {

message: '123'

}

})

vm.message = 'new message' // 更改数据

vm.$el.textContent === 'new message' // false

Vue.nextTick(function () {

vm.$el.textContent === 'new message' // true

})

在组件内使用 vm.$nextTick() 实例方法特别方便，因为它不需要全局 Vue，并且回调函数中的 this 将自动绑定到当前的 Vue 实例上：

Vue.component('example', {

template: '<span>{{ message }}</span>',

data: function () {

return {

message: '未更新'

}

},

methods: {

updateMessage: function () {

this.message = '已更新'

console.log(this.$el.textContent) // => '未更新'

this.$nextTick(function () {

console.log(this.$el.textContent) // => '已更新'

})

}

}

})

因为 $nextTick() 返回一个 Promise 对象，所以可以使用新的 [ES2017 async/await](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Statements/async_function) 语法完成相同的事情：

methods: {

updateMessage: async function () {

this.message = '已更新'

console.log(this.$el.textContent) // => '未更新'

await this.$nextTick()

console.log(this.$el.textContent) // => '已更新'

}

}

## 组件间六种数据通信方式

1、父子，props/$emit，父组件 A 通过 props 的方式向子组件 B 传递，B to A 通过在 B 组件中 $emit, A 组件中 v-on 的方式实现。

2、父子、兄弟、跨级，$emit/$on（EventBus：通过EventBus进行信息的发布与订阅），通过一个空的Vue 实例作为中央事件中心，用它来触发事件和监听事件，巧妙而轻量地实现了任何组件间的通信。

具体实现方式：

var Event = new Vue();

Event.$emit(事件名, 传递的参数对象);

Event.$on(事件名, data => {

this.data //传递的参数对象，箭头函数内部不会产生新的this，这边如果不用 => , this指代Event

});

3、vuex，对象包含{ namespaced: true, state, getters, mutations }

流程：1）Vue 组件内，执行 dispatch 方法触发对应 action 的方法（actions，操作行为处理模块,由组件中的$store.dispatch('action 名称', data1)来触发）； 2）然后由 commit()来触发 mutation 的调用（mutations，状态改变操作方法，由 actions 中的commit('mutation 名称')来触发，同步操作）, 间接更新 state; 3）由于vuex存储的数据是响应式的。但是并不会保存起来，刷新之后就回到了初始状态，具体做法应该在 vuex 里数据改变的时候把数据拷贝一份保存到 localStorage 里面，刷新之后，如果 localStorage 里有保存的数据，取出来再替换 store 里的 state。

state：页面状态管理容器对象。全局唯一，以进行统一的状态管理。

getters：state 对象读取方法，返回state中的属性。

4、多级组件嵌套的跨级通信，$attrs/$listeners，在父组件中引用子组件A时，传入绑定数据（v-bind="属性名称" 或 v-bind="$attrs"），在子组件A中用this.$attrs接收而非props接收，如果使用props接收某个数据对象，会导致this.$attrs接收不到。

$attrs：包含了父作用域中不被 prop 所识别 (且获取) 的特性绑定 (class 和 style 除外)。当一个组件没有声明任何 prop 时，这里会包含所有父作用域的绑定 (class 和 style 除外)，并且可以通过 v-bind="$attrs" 传入内部组件。通常配合 interitAttrs 选项一起使用。

$listeners：包含了父作用域中的 (不含 .native 修饰器的) v-on 事件监听器。它可以通过 v-on="$listeners" 传入内部组件。

简单来说：$attrs与$listeners 是两个对象，$attrs 里存放的是父组件中绑定的非 Props 属性，$listeners里存放的是父组件中绑定的非原生事件。

5、跨级组件间的通信，provide/inject，这对选项需要一起使用，以允许一个祖先组件向其所有子孙后代注入一个依赖，不论组件层次有多深，并在起上下游关系成立的时间里始终生效。祖先组件中通过 provider （provide: {name: '浪里行舟'}）来提供变量，然后在子孙组件中通过 inject （inject: ['name'],）来注入变量。

provide 和 inject 绑定并不是可响应的。这是刻意为之的。然而，如果你传入了一个可监听的对象，那么其对象的属性还是可响应的。

6、$parent / $children与 ref

ref：如果在普通的 DOM 元素上使用，引用指向的就是 DOM 元素；如果用在子组件上，引用就指向组件实例。在父组件中引入子组件<component-a ref="comA"></component-a>及使用this.$refs.comA即可获得子组件中data方法返回的对象实例。

$parent / $children：访问父 / 子实例。

需要注意的是：这两种都是直接得到组件实例，使用后可以直接调用组件的方法或访问数据。

总体分为三类：

父子通信：props和events( $emit )、ref、provide/inject API、$attrs/$listeners、vuex、EventBus

兄弟通信：EventBus；Vuex

跨级通信：EventBus；Vuex；provide / inject API、$attrs/$listeners

## Vue刷新浏览器数据不丢失的方法

将vuex中的数据，在刷新监听/钩子函数中，把数据存储到Localstorage中，再把localstorage同步到vuex中

# Vue-router

## $router 和 $route 的区别

$router 为VueRouter实例，想要导航到不同的url，则使用 $router.push方法；

$route 为当前router跳转对象里面可以获取name、path、query、params等。

## 导航钩子的种类

全局导航钩子：beforEach(to, from, next) 、afterEach(to, from)

单独路由独享组件：beforeEnter

组件内导航钩子：beforeRouteEnter、beforeRouteUpdate、beforeRouteLeave、

全局解析守卫：router.beforeResolve 注册一个全局守卫，和 router.beforeEach 类似，区别是在导航被确认之前，同时在所有组件内守卫和异步路由组件被解析之后，解析守卫就被调用。

完整的导航解析流程：

1、导航被触发；

2、在失活的组件里调用离开守卫beforeRouteLeave；

3、调用全局的 beforeEach 守卫；

4、在重用的组件里调用 beforeRouteUpdate 守卫；

5、在路由配置里调用 beforEnter；

6、解析异步路由组件；

7、在被激活的组件里调用 beforeRouteEnter；

8、调用全局的 beforeResolve 守卫；

9、导航被确认；

10、调用全局的 afterEach 钩子；

11、触发 DOM 更新；

12、在创建好的实例调用 beforeRouteEnter 守卫中传给 next 的回调函数。

## 传递参数两种方式

* 1. 编程式的导航 router.push

传递参数有两种类型：字符串（this.$router.push("home")）、对象（可传递参数，分为两种方式：命名路由、查询参数），

* 1. 声明式的导航 <router-link>

1）**命名路由**，前提就是在注册路由的地方用name给路由命名，使用params来传递参数，这里一定要使用params不是query。目标页面接收传递参数时使用params（ 特别注意：命名路由这种方式传递的参数，如果在目标页面刷新是会出错的）this.$router.push({ name: 'news', params: { userId: 123 }})

2）**查询参数**，其实是在路由地址后面带上参数和传统的url参数一致的，传递参数使用query而且必须配合path来传递参数而不能用name，目标页面接收传递的参数使用query（注意：和name配对的是params，和path配对的是query）this.$router.push({ path: '/news', query: { userId: 123 }});

## 导航、页面的加载流程

# 小程序

# CSS、Html5

## 盒子模型

盒子模型也称为框模型，就是从盒子顶部俯视所得的一张平面图，用于描述元素所占用的空间。

盒子模型（Box Modle）可以用来对元素进行布局，由实际内容（content）、内边距（padding）、边框（border）与外边距（margin）这几个部分组成。



1. W3C盒模型和IE盒模型的区别

理论上两者的主要区别是二者的盒子宽高是否包括元素的边框和内边距。当用CSS给某个元素定义高或宽时，**IE盒模型（怪异盒模型）中内容的宽或高将会包含内边距和边框**，而W3C盒模型（标准盒模型）并不会。

在IE盒子模型中，width/height = content + padding + border 3个部分的总和

在标准盒子模型中，width/height = content这部分的宽度或高度

1. box-sizing的使用

* box-sizing: content-box 是W3C盒子模型
* box-sizing: border-box 是IE盒子模型

box-sizing的默认属性是content-box

## 文档流

* 1. 文档流分为定位流、浮动流、普通流三种

普通流指FC（Formatting Context）格式化上下文。它是页面中的一块渲染区域，有一套渲染规则，决定子元素如何布局，以及和其他元素之间的关系和作用。常见的FC有BFC、IFC（行内格式化上下文），还有GFC和FFC。

1. 块级元素： CSS 属性值 display 为 block，list-item，table 的元素。

独占一行，对宽高的属性值生效；如果不给宽度，块级元素就默认为浏览器的宽度，即就是100%宽。

每个块级元素都会至少生成一个块级盒，称为主块级盒；一些元素可能会生成额外的块级盒，比如 < li >，用来存放项目符号。

1. 行内级元素：CSS 属性值 display 为 inline，inline-block，inline-table 的元素。

可以多个标签存在一行，对宽高属性值不生效，完全靠内容撑开宽高。

1. 行内块元素(inline-block)：结合的行内和块级的优点，既可以设置长宽，可以让padding和margin生效，又可以和其他行内元素并排。

img、input为行内块元素。

所有的非可替换行内元素（display 值为 inline-block 或 inline-table）生成的盒称为原子行内级盒，不参与 IFC 创建。

* 1. 三种文档流的定位方案
     1. 常规流(Normal flow)
* 在常规流中，盒一个接着一个排列；
* 在块级格式化上下文里面， 它们竖着排列；
* 在行内格式化上下文里面， 它们横着排列；

当position为static或relative，并且float为none时会触发常规流；

对于静态定位(static positioning)：position: static，盒的位置是常规流布局里的位置；

对于相对定位(relative positioning)：position: relative，盒偏移位置由top、bottom、left、right属性定义。即使有偏移，仍然保留原有的位置，其它常规流不能占用这个位置。

* + 1. 浮动(Floats)
* 左浮动元素尽量靠左、靠上，右浮动同理。这导致常规流环绕在它的周边，除非设置 clear 属性。
* 浮动元素不会影响块级元素的布局，但会影响行内元素的布局，让其围绕在自己周围，撑大父级元素，从而间接影响块级元素布局。
* 最高点不会超过当前行的最高点，它前面的浮动元素的最高点
* 不超过它的包含块，除非元素本身已经比包含块更宽。
* 行内元素出现在左浮动元素的右边和右浮动元素的左边。
  + 1. 绝对定位(Absolute positioning)
* 绝对定位方案，盒从常规流中被移除，不影响常规流的布局。
* 它的定位相对于它的包含块，相关CSS属性：top、bottom、left、right；
* 如果元素的属性position为absolute或fixed，它是绝对定位元素（固定定位fixed，包含块为浏览器视窗）；
* 对于position: absolute，元素定位将相对于上级元素中最近的一个relative、fixed、absolute，如果没有则相对于body；

## BFC块级格式化上下文

块级格式上下文BFC(block formatting context)是一种规定渲染区域元素排列的规则，包括子元素如何定位以及和其它元素的作用关系。块级盒参与块级格式上下文。

* 1. 触发该规则

浮动元素、绝对定位元素、非块级盒的块级容器[如行内块级元素(display:line-block)、表格单元格元素(display:table-cell)以及表格标题元素(display:table-caption)]以及overflow属性不为visible的元素。

* 1. 排列规则

在BFC区域，块级元素在垂直方向由顶部从上至下依次堆叠，上下外边距折叠；在水平方向尽量向左边缘排列(对于从左往右的格式化)，即使存在浮动也是如此。

* 1. 其它要点

BFC在页面上一个隔离的独立容器区域，容器里面的子元素不会影响到外面的元素，反之亦然。

BFC不会与外部float box重叠。

BFC计算高度时包含浮动元素。

* 1. BFC产生的常见情况（触发方式）
* 根元素，即HTML标签
* 浮动元素：float值为left、right
* 绝对定位元素：position值为 absolute、fixed
* display值为 inline-block、table-cell、table-caption、table、inline-table、flex、inline-flex、grid、inline-grid
* overflow值不为 visible，为 auto、scroll、hidden

注意：display:table也可以生成BFC的原因在于Table会默认生成一个匿名的table-cell，是这个匿名的table-cell生成了BFC

* 1. 使用注意事项
* 一个BFC包含所有在其中创建的内容。
* BFC对定位和清除浮动非常重要，定位和清除浮动规则只应用于同一个BFC中。
* 浮动不影响其他块BFC中内容的布局，清除只清除同一BFC中过去的浮动。
* 页边距折叠（collapse）也只发生在属于同一BFC的块之间。
  1. 如何创建BFC

<div class="container">

Some Content here

</div>

通过给container添加符合BFC要求的任何一个属性，都可以让该div成为一个BFC，如overflow: scroll，overflow: hidden，display: flex，float: left等等。

但是，有些属性可能会存在一些问题：

display:table，可能在响应方面会产生一些问题；

overflow:scroll，可能会显示不必要的滚动条；

float:left，将会把元素置于容器的左边，其他元素环绕着它；

overflow:hidden，将会剪切掉溢出的元素；

根据实际情况，可以选择不同的方式去建立所需要的BFC。

* 1. 利用块级格式上线文进行布局
     1. 自适应两栏布局

对于宽度不定的包含块，划分为两栏布局。要求左边(.aside)固定宽度的栏，右边(.main)宽度随整体宽度大小自适应。

视图效果

<div class='container'>

<div class="aside"></div>

<div class="main"></div>

</div>

.container {

    width: 500px;/\* 宽度值可变化 \*/

}

.aside {

    width: 100px;

    height: 150px;

    background: #f66;

}

.main {

    height: 200px;

    background: #fcc;

}

* + 1. 固定右侧栏

需要右侧固定栏，可以通过左浮动来实现。

.aside {

    float: left;

}

视图效果

* + 1. 利用BFC原理解决浮动元素的重叠问题

由于浮动元素脱离了正常文档流，用户代理在渲染时，会导致正常的块级元素“看不到”浮动元素。在视觉堆叠层面，就像浮动元素“漂浮”到了块级元素前面。因为BFC不会与外部float box重叠，通过设置 .main { overflow: hidden;} 将右侧元素(.main ) 转换为块级格式上下文，实现实现两栏分栏排列。

视图效果

* 1. 清除内部浮动对布局的影响

利用BFC解决浮动元素导致的父元素容器没有高度的问题，BFC 可以包含浮动的元素（清除浮动）。

<div class='container'>

<div class="floatBox"></div>

<div class="floatBox"></div>

</div>

.container {

width: 800px;

}

.floatBox {

width: 300px;

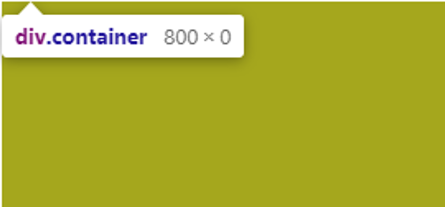
height: 200px;

background: #afaf00;

float: left;

}

由于元素浮动，从正常文档流中脱离，导致容器（ .container）高度塌陷，在控制台中查看元素可以发现容器高度为0。根据BFC布局规则，计算块级格式上下文高度时，需要包含浮动元素。故而通过将容器增加属性overflow:hidden将其排布规则遵循块级格式上下文，这样容器（ .container）高度就包含了浮动元素。

视图效果

* 1. 防止外间距折叠

由于BFC在页面上一个隔离的独立容器区域，容器里面的子元素不会影响到外面的元素，所以可以利用将元素至于BFC区域来脱离外部元素之间的影响，譬如避免外边距的折叠。

<div class='container'>

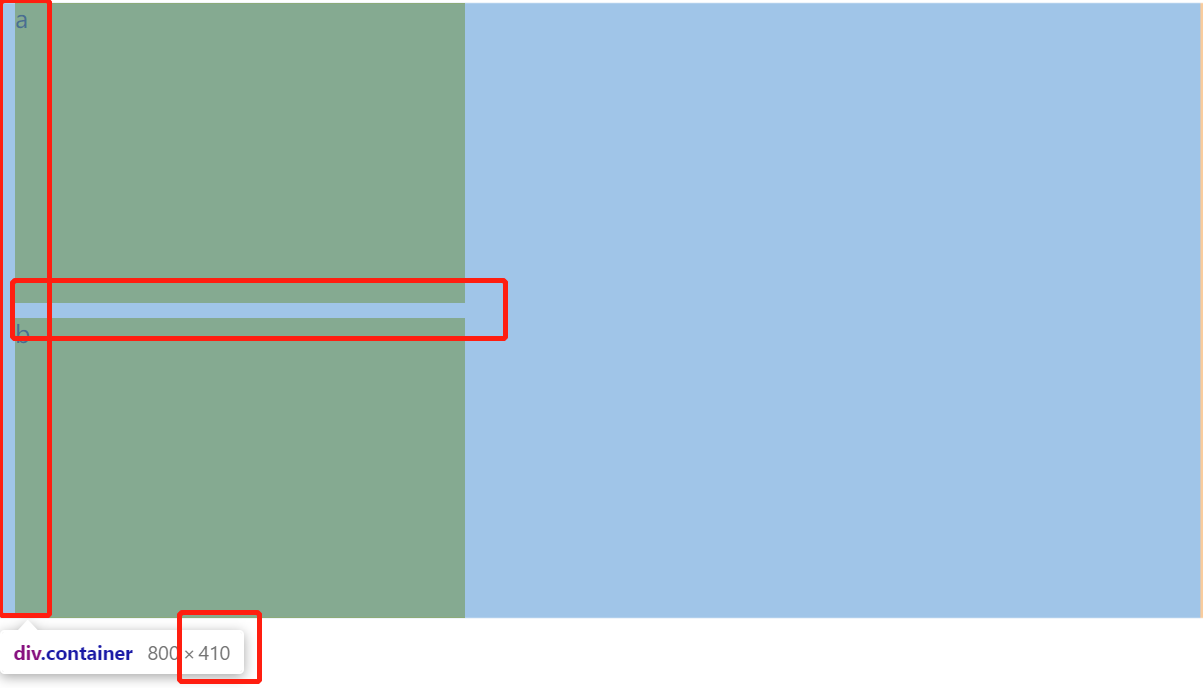
<div class="Box">a</div>

<div class="Box">b</div>

</div>

.container { width: 800px; }

.Box { width: 300px; height: 200px; background: #afaf00;  margin:10px; }

视图效果

很明显，上面两个子元素（.Box）的margin发生了折叠，两个元素的上下之间内容盒间距实际只有10px。

<div class='container'>

    <div class='BFC'>

        <div class="Box">a</div>

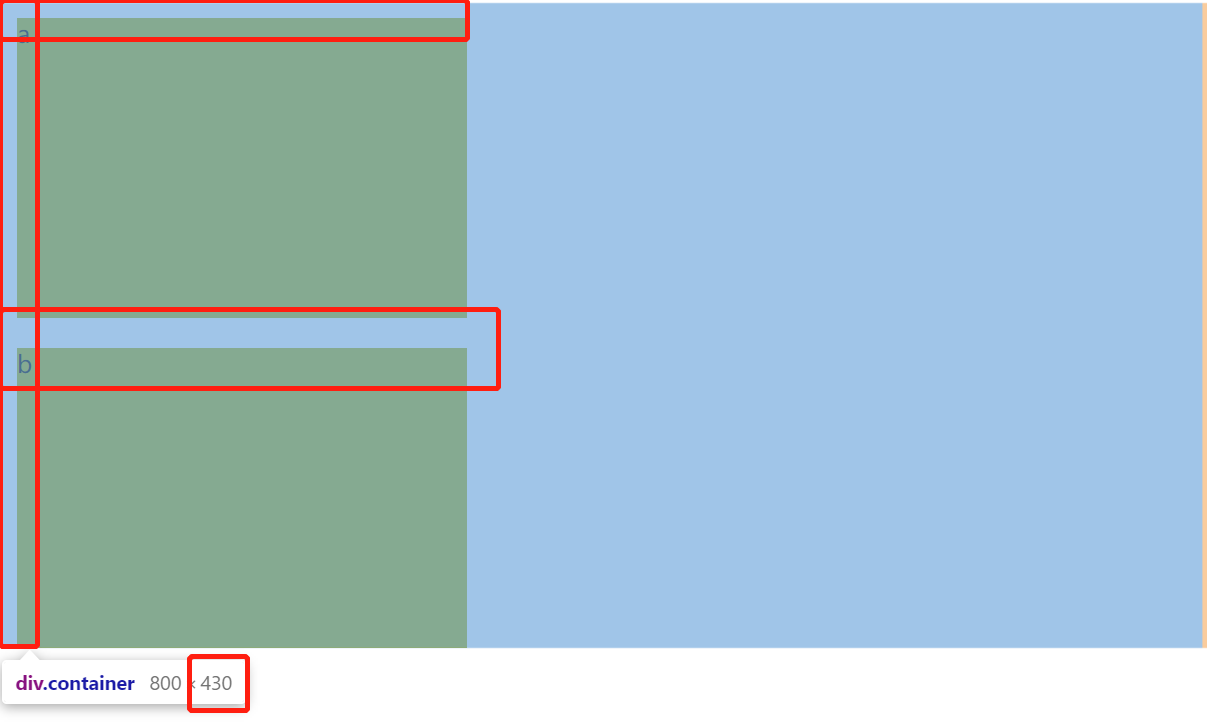
    </div>

    <div class="Box">b</div>

</div>

.BFC{   overflow:hidden;   }

通过对子元素包裹一个块级格式上下文，可以避免外间距的折叠。

视图效果

## 清除浮动的几种方式及原理

清除浮动简单，但这题要引出的是BFC的基础知识点。

## css水平、垂直居中的4种写法

（1）水平居中

* 行内元素: text-align: center;
* 块级元素: margin: 0 auto;
* position:absolute; left:50%; transform:translateX(-50%);
* 在子项中添加 display:flex; justify-content: center;

（2）垂直居中

* 设置line-height 等于height;
* position：absolute; top:50%; transform:translateY(-50%);
* 在子项中添加 display:flex; align-items: center;
* display:table; display:table-cell; vertical-align: middle;

## 浏览器的存储方式

浏览器新增支持H5的几种存储方式， 如 localStorage、sessionstorage 、application 、 cache 、Web SQL 、IndexedDB

目前最主要用到就是localStorage、indexDB

动画过渡

transform 动画方式

translate 水平垂直移动

scale 水平垂直缩放

ratate 旋转

transition 动画方式按照什么属性进行过渡，过渡多长时间

hight、width、transform

# Webpack

webpack是一个打包模块化javascript的工具，在webpack里一切文件皆模块，通过loader转换文件，通过plugin注入钩子，最后输出由多个模块组合成的文件，webpack专注构建模块化项目。

Webpack的工作方式是：把你的项目当做一个整体，通过一个给定的主文件（如：index.js），Webpack将从这个文件开始找到你的项目的所有依赖文件，使用loaders处理它们，最后打包为一个（或多个）浏览器可识别的JavaScript文件。

三者都是前端构建工具，grunt和gulp在早期比较流行，现在webpack相对来说比较主流，不过一些轻量化的任务还是会用gulp来处理，比如单独打包CSS文件等。

Grunt和Gulp的工作方式是：在一个配置文件中，指明对某些文件进行类似编译，组合，压缩等任务的具体步骤，工具之后可以自动替你完成这些任务。

## webpack的构建流程

Webpack 的运行流程是一个串行的过程，从启动到结束会依次执行以下流程：

从读取配置到输出文件过程。

1. 初始化参数：从配置文件和 Shell 语句中读取与合并参数，得出最终的参数；
2. 开始编译：用上一步得到的参数初始化 Compiler 对象，加载所有配置的插件，执行对象的 run 方法开始执行编译；
3. 确定入口：根据配置中的 entry 找出所有的入口文件；
4. 编译模块：从入口文件出发，调用所有配置的 Loader 对模块进行翻译，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：在经过第4步使用 Loader 翻译完所有模块后，得到了每个模块被翻译后的最终内容以及它们之间的依赖关系；
6. 输出资源：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，再把每个 Chunk 转换成一个单独的文件加入到输出列表，这步是可以修改输出内容的最后机会；
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，把文件内容写入到文件系统。
8. 在以上过程中，Webpack 会在特定的时间点广播出特定的事件，插件在监听到感兴趣的事件后会执行特定的逻辑，并且插件可以调用 Webpack 提供的 API 改变 Webpack 的运行结果。

## webpack有哪些优缺点

（1）优点

* 专注于处理模块化的项目，能做到开箱即用，一步到位
* 可通过plugin扩展，完整好用又不失灵活
* 使用场景不局限于web开发
* 社区庞大活跃，经常引入紧跟时代发展的新特性，能为大多数场景找到已有的开源扩展
* 良好的开发体验

（2）缺点

* webpack的缺点是只能用于采用模块化开发的项目

## bundle、chunk、module分别是什么

bundle：是由webpack打包出来的文件。

chunk： 代码块，一个chunk由多个模块组合而成，用于代码的合并和分割。

module：是开发中的单个模块，在webpack的世界，一切皆模块，一个模块对应一个文件，webpack会从配置的entry中递归开始找出所有依赖的模块。

## Loader、plugin分别是什么

简单的解释：

loader：模块转换器，用于将模块的原内容按照需要转成你想要的内容。

plugin：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，是用来自定义webpack打包过程的方式，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(生命周期)。

（1）不同的作用

Loader直译为"加载器"。Webpack将一切文件视为模块，但是webpack原生是只能解析js文件，如果想将其他文件也打包的话，就会用到loader。 所以Loader的作用是让webpack拥有了加载和解析非JavaScript文件的能力。

Plugin直译为"插件"。Plugin可以扩展webpack的功能，让webpack具有更多的灵活性。 在 Webpack 运行的生命周期中会广播出许多事件，Plugin 可以监听这些事件，在合适的时机通过 Webpack 提供的 API 改变输出结果。

（2）不同的用法

Loader在module.rules中配置，也就是说他作为模块的解析规则而存在。 类型为数组，每一项都是一个Object，里面描述了对于什么类型的文件（test），使用什么加载(loader)和使用的参数（options）

Plugin在plugins中单独配置。 类型为数组，每一项是一个plugin的实例，参数都通过构造函数传入。

## 常见的loader

* 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
* url-loader：和 file-loader 类似，但是能在文件很小的情况下以 base64 的方式把文件内容注入到代码中去
* source-map-loader：加载额外的 Source Map 文件，以方便断点调试
* image-loader：加载并且压缩图片文件
* babel-loader：把 ES6 转换成 ES5
* css-loader：加载 CSS，支持模块化、压缩、文件导入等特性
* style-loader：把 CSS 代码注入到 JavaScript 中，通过 DOM 操作去加载 CSS。
* eslint-loader：通过 ESLint 检查 JavaScript 代码

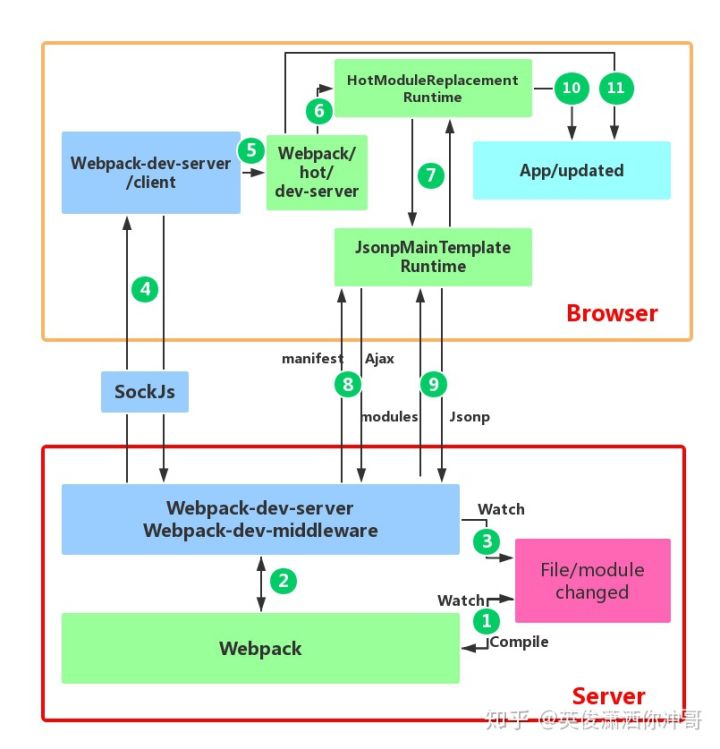
## 常见的plugin

* define-plugin：定义环境变量
* terser-webpack-plugin：通过TerserPlugin压缩ES6代码
* 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
* happypack：实现多线程加速编译
* commons-chunk-plugin：提取公共代码
* uglifyjs-webpack-plugin：通过UglifyES压缩ES6代码

## webpack的热更新 及 原理

webpack的热更新又称热替换（Hot Module Replacement），缩写为HMR。 这个机制可以做到不用刷新浏览器而将新变更的模块替换掉旧的模块。

原理



首先要知道server端和client端都做了处理工作

1. 第一步，在 webpack 的 watch 模式下，文件系统中某一个文件发生修改，webpack 监听到文件变化，根据配置文件对模块重新编译打包，并将打包后的代码通过简单的 JavaScript 对象保存在内存中。
2. 第二步是 webpack-dev-server 和 webpack 之间的接口交互，而在这一步，主要是 dev-server 的中间件 webpack-dev-middleware 和 webpack 之间的交互，webpack-dev-middleware 调用 webpack 暴露的 API对代码变化进行监控，并且告诉 webpack，将代码打包到内存中。
3. 第三步是 webpack-dev-server 对文件变化的一个监控，这一步不同于第一步，并不是监控代码变化重新打包。当我们在配置文件中配置了devServer.watchContentBase 为 true 的时候，Server 会监听这些配置文件夹中静态文件的变化，变化后会通知浏览器端对应用进行 live reload。注意，这儿是浏览器刷新，和 HMR 是两个概念。
4. 第四步也是 webpack-dev-server 代码的工作，该步骤主要是通过 sockjs（webpack-dev-server 的依赖）在浏览器端和服务端之间建立一个 websocket 长连接，将 webpack 编译打包的各个阶段的状态信息告知浏览器端，同时也包括第三步中 Server 监听静态文件变化的信息。浏览器端根据这些 socket 消息进行不同的操作。当然服务端传递的最主要信息还是新模块的 hash 值，后面的步骤根据这一 hash 值来进行模块热替换。
5. webpack-dev-server/client 端并不能够请求更新的代码，也不会执行热更模块操作，而把这些工作又交回给了 webpack，webpack/hot/dev-server 的工作就是根据 webpack-dev-server/client 传给它的信息以及 dev-server 的配置决定是刷新浏览器呢还是进行模块热更新。当然如果仅仅是刷新浏览器，也就没有后面那些步骤了。
6. HotModuleReplacement.runtime 是客户端 HMR 的中枢，它接收到上一步传递给他的新模块的 hash 值，它通过 JsonpMainTemplate.runtime 向 server 端发送 Ajax 请求，服务端返回一个 json，该 json 包含了所有要更新的模块的 hash 值，获取到更新列表后，该模块再次通过 jsonp 请求，获取到最新的模块代码。这就是上图中 7、8、9 步骤。
7. 而第 10 步是决定 HMR 成功与否的关键步骤，在该步骤中，HotModulePlugin 将会对新旧模块进行对比，决定是否更新模块，在决定更新模块后，检查模块之间的依赖关系，更新模块的同时更新模块间的依赖引用。
8. 最后一步，当 HMR 失败后，回退到 live reload 操作，也就是进行浏览器刷新来获取最新打包代码。

## Tree-shaking

## 通过webpack处理长缓存

## 如何提高webpack的构建速度

# Nodejs

# 其它

## Git，Github和Gitlab简介和基本使用

* 1. Git 版本控制系统

Git是一个版本控制系统（Version Control System，VCS）。

版本控制是一种记录一个或若干文件内容变化，以便将来查阅特定版本修订情况的系统。

常用的Git命令:

git checkout

git status

git checkout -b

git commit

git diff

git add

git clean -fdx

git fetch --prune

git pull --ff-only

git branch -D

git push

git merge

git rm

git rebase

git reset

git revert

git push origin :yourbranch

* 1. Github 代码托管服务/代码仓库

Github 和 Git 是两回事。

Git是版本控制系统，Github是在线的基于Git的代码托管服务。

GitHub是2008年由Ruby on Rails编写而成。

GitHub同时提供付费账户和免费账户。这两种账户都可以创建公开的代码仓库，但是付费账户也可以创建私有的代码仓库。

* 1. Gitlab 免费的代码仓库

Github有局限性，就是用户创建的repo(repository的缩写，表示“仓库”)都需要public(公开的)，如果用户想要创建private(私人)的repo，那得付钱。

而Gitlab解决了这个局限，可以在上面创建免费的私人repo。

浏览器的内置数据库