Browser

HTML元信息标签

```
<meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
<!-- 默认使用最新浏览器 -->
<meta http-equiv="Cache-Control" content="no-siteapp">
<!-- 不被网页(加速)转码 -->
<meta name="robots" content="index,follow">
<!-- 搜索引擎抓取 -->
<meta name="renderer" content="webkit">
<meta name="viewport" content="width=device-width, initial-scale=1, maximum-scale=1, minim
<meta name="apple-mobile-web-app-capable" content="yes">
<!-- 删除苹果默认的工具栏和菜单栏 -->
<meta name="apple-mobile-web-app-status-bar-style" content="black-translucent">
<!-- 设置苹果工具栏颜色 -->
```

H5标签信息

```
<!-- 页面字符编码 -->
<meta charset="utf-8">

<!-- 避免IE使用兼容模式 -->
<meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">

<!-- 启用360浏览器的极速模式(webkit) -->
<meta name="renderer" content="webkit">

<!-- 微软的老式浏览器 -->
<meta name="MobileOptimized" content="320">

<!-- 关键字描述 -->
<meta name="keywords" content="">
<meta name="description" content="">

<!-- 设置移动端视图 -->
<meta name="viewport" content="width=device-width, initial-scale=1, user-scalable=no" />

<!-- 针对手持设备优化，主要是针对一些老的不识别viewport的浏览器，比如黑莓 -->
<meta name="HandheldFriendly" content="true">

<!-- 删除苹果默认的工具栏和菜单栏 -->
<meta name="apple-mobile-web-app-capable" content="yes" />

<!-- 设置苹果工具栏颜色 -->
<meta name="apple-mobile-web-app-status-bar-style" content="black" />

<!-- 忽略页面中的数字识别为电话，忽略email识别 -->
<meta name="format-detection" content="telphone=no, email=no" />

<!-- uc强制竖屏 -->
<meta name="screen-orientation" content="portrait">

<!-- QQ强制竖屏 -->
<meta name="x5-orientation" content="portrait">

<!-- UC强制全屏 -->
<meta name="full-screen" content="yes">

<!-- QQ强制全屏 -->
<meta name="x5-fullscreen" content="true">

<!-- UC应用模式 -->
<meta name="browsermode" content="application">
```

```
<!-- QQ应用模式 -->
<meta name="x5-page-mode" content="app">

<!-- windows phone 点击无高光 -->
<meta name="msapplication-tap-highlight" content="no">
```

浏览器事件

跨域

什么是跨域

> 我们通常所说的跨域，是指由浏览器同源策略限制的一类请求场景。是指浏览器不能执行其他网站的脚本。是浏览器对JavaScript实施的安全限制。

什么是同源策略

> 所谓同源是指"协议+域名+端口"三者相同；即便是两个不同的域名指向同一个ip地址，也非同源

特别注意两点：

1. 如果是协议和端口造成的跨域问题“前台”是无能为力的。
2. 在跨域问题上，域仅仅是通过“URL的首部”来识别而不会去尝试判断相同的ip地址对应着两个域或两个域是否在同一个ip上。
   常见跨域场景

URL	说明	是否允许通信
http://www.domain.com/a.js http://www.domain.com/b.js http://www.domain.com/lab/c.js	同一域名，不同文件或路径	允许
http://www.domain.com:8000/a.js http://www.domain.com/b.js	同一域名，不同端口	不允许
http://www.domain.com/a.js https://www.domain.com/b.js	同一域名，不同协议	不允许
http://www.domain.com/a.js http://192.168.4.12/b.js	域名和域名对应相同ip	不允许
http://www.domain.com/a.js http://x.domain.com/b.js http://domain.com/c.js	主域相同，子域不同	不允许
http://www.domain1.com/a.js http://www.domain2.com/b.js	不同域名	不允许

跨域常见解决方案

- 通过JSONP跨域
- document.domain + iframe 解决跨域
- 跨域资源共享（CORS）

跨域资源共享（CORS）（cross-origin sharing standard ）

> MDN地址：https://developer.mozilla.org/zh-CN/docs/Web/HTTP/Access_control_CORS#Preflighted_requests

MDN定义：跨域资源共享(CORS) 是一种机制，它使用额外的 HTTP 头来告诉浏览器 让运行在一个 origin (domain) 上的Web应用被准许访问来自不同源服务器上的指定的资源。当一个资源从与该资源本身所在的服务器不同的域、协议或端口请求一个资源时，资源会发起一个跨域 HTTP 请求。

> 所有浏览器都支持该功能(IE8+：IE8/9需要使用XDomainRequest对象来支持CORS），CORS也已经成为主流的跨域解决方案。
> 普通跨域请求：只服务端设置Access-Control-Allow-Origin即可，前端无须设置，若要带cookie请求：前后端都需要设置。

预检请求 OPTIONS

对那些可能对服务器数据产生副作用的 HTTP 请求方法（特别是 GET 以外的 HTTP 请求，或者搭配某些 MIME 类型的 POST 请求），浏览器必须首先使用 OPTIONS 方法发起一个预检请求（preflight request），从而获知服务端是否允许该跨域请求。服务器确认允许之后，才发起实际的 HTTP 请求。在预检请求的返回中，服务器端也可以通知客户端，是否需要携带身份凭证（包括 Cookies 和 HTTP 认证相关数据）

简单请求

某些请求不会触发 CORS 预检请求。本文称这样的请求为“简单请求”，请注意，该术语并不属于 Fetch （其中定义了 CORS）规范。若请求满足所有下述条件，则该请求可视为“简单请求”：

```
使用下列方法之一：
* GET
* HEAD
* POST

没有人为设置该集合之外的其他首部字段：
    Accept
    Accept-Language
    Content-Language
    Content-Type （需要注意额外的限制）
    DPR
    Downlink
    Save-Data
    Viewport-Width
    Width
    Content-Type 的值仅限于下列三者之一：
    text/plain
    multipart/form-data
    application/x-www-form-urlencoded
```

阮一峰博客：http://www.ruanyifeng.com/blog/2016/04/cors.html

HTML

base标签

HTML 元素 指定用于一个文档中包含的所有相对 URL 的根 URL。一份中只能有一个 元素。

一个文档的基本 URL, 可以通过使用 document.baseURI 查询。如果文档不包含 元素, baseURI 默认为 document.location.href

href用于文档中相对 URL 地址的基础 URL。允许绝对和相对URL。

> 如果指定了多个 元素， 只会使用第一个 href 和 target 值, 其余都会被忽略。

页内锚
指向文档中某个片段的链接， 例如 用 解析， 触发对带有附加片段的基本 URL 的 HTTP 请求。

```
例如：给定 <base href="https://example.com">
以及此链接 <a href="#anchor">Anker</a>
链接指向 https://example.com/#anchor
```

> 简单理解：base标签的作用就是， 当页面中的A超链接标签没有设置href属性的值和没有设置target属性的值时， 默认使用base标签中的href属 性的值和target属性的值

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>base标签</title>
<link rel="stylesheet" type="text/css" href="css/body.css" media="all">
<link rel="stylesheet" type="text/css" href="css/mark.css">
<base href="https://www.google.com.hk">
<base target="_blank">
</head>
<body>
    <div align="center">
        <a href="">测试1</a><br/><br/>
        <a href="">测试2</a><br/><br/>
        <a href="">测试3</a><br/><br/>
        <a href="">测试4</a><br/><br/>
        <a href="http://www.baidu.com" target="_self">测试5</a>
    </div>
</body>
</html>
```

HTTP
HTTP和HTTPS
http和https的基本概念基本概念

http: 超文本传输协议，是互联网上应用最为广泛的一种网络协议，是一个客户端和服 务器端请求和应答的标准(TCP)，用于从 WWW 服务器
https: 是以安全为目标的 HTTP 通道，简单讲是 HTTP 的安全版，即 HTTP 下加入 SSL 层，HTTPS 的安全基础是 SSL，因此加密的详
https 协议的主要作用是:建立一个信息安全通道，来确保数组的传输，确保网站的真实 性。

http 和 https 的区别

http 是超文本传输协议，信息是明文传输，https 则是具有安全性的 ssl 加密传输协议。 使用不同的链接方式，端口也不同，一般而言，
HTTPS 协议是由 SSL+HTTP 协议构建的可进行加密传输、身份认证的网络协议，要比 http 协议安全，可防止数据在传输过程中不被窃取、
HTTP状态码
* 100 Continue 继续。客户端应继续其请求
* 101 Switching Protocols 切换协议。服务器根据客户端的请求切换协议。只能切换到更 高级的协议，例如，切换到 HTTP 的新版本协
* **_200 OK 请求成功。一般用于 GET 与 POST 请求_**
* 201 Created 已创建。成功请求并创建了新的资源
* 202 Accepted 已接受。已经接受请求，但未处理完成
* 203 Non-Authoritative Information 非授权信息。请求成功。但返回的 meta 信息不在原始的服务器，而是一个副本
* 204 No Content 无内容。服务器成功处理，但未返回内容。在未更新网页的情况下， 可确保浏览器继续显示当前文档
* 205 Reset Content 重置内容。服务器处理成功，用户终端(例如:浏览器)应重置文 档视图。可通过此返回码清除浏览器的表单域
* 206 Partial Content 部分内容。服务器成功处理了部分 GET 请求
* 300 Multiple Choices 多种选择。请求的资源可包括多个位置，相应可返回一个资源特 征与地址的列表用于用户终端(例如:浏览器)选
* 301 Moved Permanently 永久移动。请求的资源已被永久的移动到新 URI，返回信息会 包括新的 URI，浏览器会自动定向到新 URI。

* 302 Found 临时移动。与 301 类似。但资源只是临时被移动。客户端应继续使用原有 URI
* 303 See Other 查看其它地址。与 301 类似。使用 GET 和 POST 请求查看
* _304 Not Modified 未修改。所请求的资源未修改，服务器返回此状态码时，不会返回
任何资源。客户端通常会缓存访问过的资源，通过提供一个头信息指出客户端希望只返 回在指定日期之后修改的资源_
* 305 Use Proxy 使用代理。所请求的资源必须通过代理访问
* 306 Unused 已经被废弃的 HTTP 状态码
* 307 Temporary Redirect 临时重定向。与 302 类似。使用 GET 请求重定向
* **_400 Bad Request 客户端请求的语法错误，服务器无法理解_**
* **_401 Unauthorized 请求要求用户的身份认证_**
* **_402 Payment Required 保留，将来使用_**
* **_403 Forbidden 服务器理解请求客户端的请求，但是拒绝执行此请求_**
* **_404 Not Found 服务器无法根据客户端的请求找到资源(网页)。通过此代码，网站
设计人员可设置"您所请求的资源无法找到"的个性页面_**
* **_405 Method Not Allowed 客户端请求中的方法被禁止_**
* 406 Not Acceptable 服务器无法根据客户端请求的内容特性完成请求
* 407 Proxy Authentication Required 请求要求代理的身份认证，与 401 类似，但请求者应当使用代理进行授权
* 408 Request Time-out 服务器等待客户端发送的请求时间过长，超时
* 409 Conflict 服务器完成客户端的 PUT 请求是可能返回此代码，服务器处理请求时发生了冲突
* 410 Gone 客户端请求的资源已经不存在。410 不同于 404，如果资源以前有现在被永 久删除了可使用 410 代码，网站设计人员可通过
* 411 Length Required 服务器无法处理客户端发送的不带 Content-Length 的请求信息
* 412 Precondition Failed 客户端请求信息的先决条件错误
* 413 Request Entity Too Large 由于请求的实体过大，服务器无法处理，因此拒绝请求。为防止客户端的连续请求，服务器可能会关
* 414 Request-URI Too Large 请求的 URI 过长(URI 通常为网址)，服务器无法处理
* 415 Unsupported Media Type 服务器无法处理请求附带的媒体格式
* 416 Requested range not satisfiable 客户端请求的范围无效
* 417 Expectation Failed 服务器无法满足 Expect 的请求头信息
* 500 _Internal Server Error 服务器内部错误，无法完成请求_
* 501 Not Implemented 服务器不支持请求的功能，无法完成请求
* **_502 Bad Gateway 作为网关或者代理工作的服务器尝试执行请求时，从远程服务器接收到了一个无效的响应_**
* 504 Gateway Time-out 充当网关或代理的服务器，未及时从远端服务器获取请求
* 505 HTTP Version not supported 服务器不支持请求的 HTTP 协议的版本，无法完成处理
TCP和UDP
1. TCP 是面向连接的，udp 是无连接的即发送数据前不需要先建立链接
2. TCP 提供可靠的服务。也就是说，通过 TCP 连接传送的数据，无差错，不丢失， 不重复，且按序到达；UDP 尽最大努力交付，即不保证
3. TCP 是面向字节流；UDP 面向报文，并且网络出现拥塞不会使得发送速率降低(因 此会出现丢包，对实时的应用比如 IP 电话和视频会议
4. TCP 只能是 1 对 1 的，UDP 支持 1 对 1、1 对多。
5. TCP 的首部较大为 20 字节，而 UDP 只有 8 字节
6. TCP 是面向连接的可靠性传输，而 UDP 是不可靠的。

浏览器输入URL之后
1. 输入地址后，首先进行DNS域名解析
> * 在浏览器DNS缓存中搜索
> * 在操作系统DNS缓存中搜索
> * 读取系统hosts文件，查找其中是否有对应的ip
> * 向本地配置的首选DNS服务器发起域名解析请求

2. 建立TCP连接
> TCP协议采用了三次握手策略：
> * 发送端首先发送一个带SYN（synchronize）标志的数据包给接收方
> * 接收方收到后，回传一个带有SYN/ACK(acknowledegment)标志的数据包以示传达确认信息
> * 最后发送方再回传一个带ACK标志的数据包，代表握手结束
3. 浏览器向 web 服务器发起HTTP/HTTPS请求
4. 服务器的永久重定向响应，浏览器跟踪重定向地址，服务器处理请求，服务器返回一个 http 响应。
5. 浏览器解析html
6. 浏览器布局渲染
简述HTTP2.0
http 和 https 的区别，相比于 http,https 是基于 ssl 加密的 http 协议

http2.0 是基于 1999 年发布的 http1.0 之后的首次更新

1. 提升访问速度(可以对于，请求资源所需时间更少，访问速度更快，相比 http1.0)
2. 允许多路复用：多路复用允许同时通过单一的 HTTP/2 连接发送多重请求-响应信息。
3. 改善了:在 http1.1 中，浏览器客户端在同一时间，针对同一域名下的请求有一定数量限 制(连接数量)，超过限制会被阻塞。
4. 二进制分帧:HTTP2.0 会将所有的传输信息分割为更小的信息或者帧，并对他们进行二进制编码
5. 首部压缩
6. 服务器端推送

JavaScript
数据类型定义
https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Data_structures https://developer.mozilla.org/zh-C

定义
> ECMAScript 有 6 种简单数据类型(也称为原始类型):Undefined、Null、Boolean、Number、 String 和 Symbol。Symbol(符号

类型检查
> typeof 操作符的唯一目的就是检查数据类型，如果我们希望检查任何从 Object 派生出来的结构类型，使用 typeof 是不起作用的，因为
>
* typeof
> 用于检测变量是不是基本数据类型，是哪种基本数据类型；
```javascript
const num = 1;
console.log(typeof num); // number
const str = 'str';
console.log(typeof str); // string
const bool = false;
console.log(typeof bool); // boolean
let unde = undefined;
console.log(typeof unde); // undefined
let nu = null;
console.log(typeof nu); // object
```

```
// typeof不适用于引用类型的检测
const obj = {};
console.log(typeof obj); // object
const arr = [];
console.log(typeof arr); // object
function func(params) {}
console.log(typeof func); // function
```

- instanceof
  result = variable instanceof constructor
  通过原型链判断，如果变量variable是给定的引用类型constructor的实例，则返回true；不适用于基本数据类型的检测

- Array.isArray

  类型转换

> 很多实践中推荐禁止使用“ ==”，而要求程序员进行显式地类型...

ES6的箭头函数

https://es6.ruanyifeng.com/#docs/function#箭头函数

箭头函数有几个使用注意点。

（1）箭头函数没有自己的this对象。

> 对于普通函数来说，内部的this指向函数运行时所在的对象，但是这一点对箭头函数不成立。
> 它没有自己的this对象，内部的this就是定义时上层作用域中的this

```
// 普通函数
function foo() {
  setTimeout(function() {
    console.log('id:', this.id);
  }, 100);
}

var id = 21;

foo.call({ id: 42 });
// 浏览器执行 id: 21  node执行 undefined

>----------------------------<

// 箭头函数
function foo() {
  setTimeout(() => {
    console.log('id:', this.id);
  }, 100);
}

var id = 21;

foo.call({ id: 42 });
// id: 42
```

（2）不可以当作构造函数，也就是说，不可以对箭头函数使用new命令，否则会抛出一个错误。

（3）不可以使用arguments对象，该对象在函数体内不存在。如果要用，可以用 rest 参数代替。

（4）不可以使用yield命令，因此箭头函数不能用作 Generator 函数。

this相关知识点

this相关知识点

> new绑定 > 显示绑定 > 隐式绑定 > 默认绑定

> this是在调用时绑定的

如果要判断一个运行中函数的this绑定，就需要找到这个函数的直接调用位置。找到之后就可以

顺序应用下面这四条规则来判断this的绑定对象。——你不知道的JavaScript（上卷）

> 1. 由new调用？绑定到新创建的对象。
> 2. 由call或者apply（或者bind）调用？绑定到指定的对象。
> 3. 由上下文对象调用？绑定到那个上下文对象。
> 4. 默认：在严格模式下绑定到undefined，否则绑定到全局对象。

```
/**
 * 优先级
 * new绑定和隐式绑定的优先级
 */

function foo(something) {
  this.a = something;
}
const obj1 = { foo };
const obj2 = {};

obj1.foo(2); // 此时foo的this指向obj1，所以foo执行时，this.a=2相当于执行了obj1.a=2
console.log(obj1.a);// 2

obj1.foo.call(obj2, 3); // 此时foo的this指向obj2，所以foo执行时，this.a=2相当于执行了obj1.a=2
console.log(obj2.a);// 3

const bar = new obj1.foo(4);
console.log(obj1.a);// 2
console.log(bar.a);// 4
```

new绑定

- 构造函数new一个对象实例的过程

1. 创建一个新对象实例；
2. 将构造函数的作用域赋给新对象实例
3. 执行构造函数中的代码，为新对象实例添加属性
4. 返回新对象实例

默认绑定

> 在不能应用其它绑定规则时使用的默认规则，通常是独立函数调用。

```
function sayHi(){
    console.log('Hello,', this.name);
}
var name = 'YvetteLau';
sayHi();
// 在调用 Hi() 时，应用了默认绑定，this 指向全局对象（非严格模式下），
// 严格模式下，this 指向 undefined，undefined 上没有 this 对象，会抛出错误。
```

隐式绑定

> 函数的调用是在某个对象上触发的，即调用位置上存在上下文对象。典型的形式为 XXX.fun().

> 对象属性链中只有最后一层会影响到调用位置。
> eg :person1.friend.sayHi();

显式绑定

> 就是通过 call,apply,bind 的方式

call 和 apply的功能相同，都是在调用函数，并修改this指向第一个参数；区别在于传参的方式不一样:

> - fn.call(obj, arg1, arg2, ...),调用一个函数, 具有一个指定的this值和分别地提供的参数(参数的列表)。
> - fn.apply(obj, [argsArray]),调用一个函数，具有一个指定的this值，以及作为一个数组（或类数组对象）提供的参数。

```
// node中执行
global.a = 3;

function foo() {
  console.log(this.a);
}

const obj = { a: 2 };

foo.call(obj); // 2
// 如果将第一个参数传为一个基本类型2 此时this指向Number引用类型
// 例如 Boolean，String，Number，这个将基本类型转为引用类型的操作成为“装箱”
foo.call(2);
foo.call(null); // 如果把undefined和null作为绑定对象传给call或者apply，此时应用的是默认绑定规则
```

> 如果把undefined和null作为绑定对象传给call或者apply，此时应用的是默认绑定规则；

bind

bind(..)会返回一个硬编码的新函数，它会把参数设置为this的上下文并调用原始函数。——你不知道的JavaScript（上卷）

硬绑定

应用场景：创建一个包裹函数，传入所有的参数并返回接收到的所有值；这就是ES5中bind的由来

```
/**
 * 硬绑定
 * 应用场景：创建一个包裹函数，传入所有的参数并返回接收到的所有值
 */

function foo() {
  console.log(`foo: ${this.a}`);
}
global.a = 3; // node
window.a = 3; // 浏览器

const obj = { a: 2 };
function bar() {
  // 强制将foo的this绑定到obj，对于bar函数的调用方式不会影响foo函数this的指向，
  // 这种显式的强制绑定，成为硬绑定
  foo.call(obj);
  console.log(`bar: ${this.a}`);
}
bar(); // foo: 2 bar: 3
setTimeout(bar, 100); // foo: 2  bar: node环境中是undefined，浏览器中是3
// 硬绑定的bar不可能再修改它的this
bar.call(global); // foo: 2 bar: 3
bar.call(window); // foo: 2 bar: 3
```

箭头函数

> - 箭头函数体内的this对象，就是定义时所在的对象，而不是使用时所在的对象，this的指向是不可变的;
> - 函数体内的 this 对象，继承的是外层代码块的 this。

> - 不可以使用arguments对象，该对象在函数体内不存在。如果要用，可以用 rest 参数代替。

- 不可以当作构造函数，也就是说，不可以使用new命令，否则会抛出一个错误。
- 不可以使用yield命令，因此箭头函数不能用作 Generator 函数。

原型

> 原型

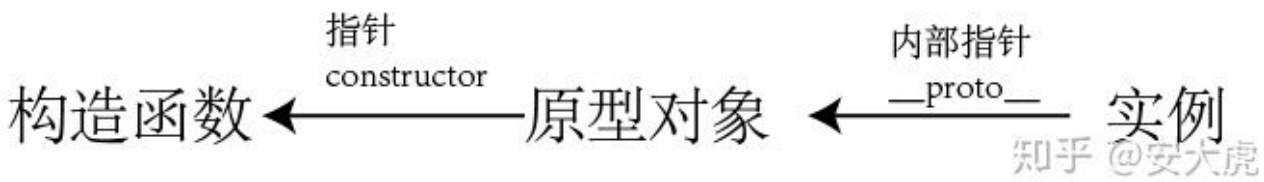
- 每个实例对象（object ）都有一个私有属性（称之为__proto__）指向它的原型对象（prototype）。
- 无论什么时候创建一个函数，该函数都有一个prototype（原型）属性，即原型对象
- 默认情况下，原型对象都会自动获取一个constructor（构造函数）属性，该属性指向prototype属性所在的函数
- prototype（原型）就是通过构造函数来创建的对象实例的原型对象，这个对象让所有对象实例可以共享它所包含的属性和方法
- 所有普通的Prototype链最终都会指向内置的Object.prototype

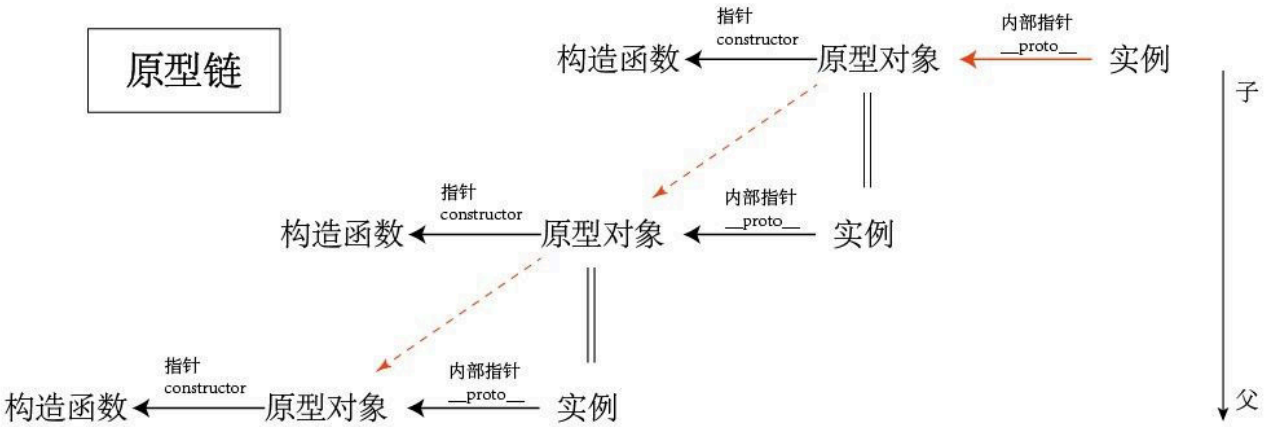
> 构造函数

- 任何函数，只要通过new操作符来调用，就可以作为构造函数
- 构造函数new一个对象实例的过程

1. 创建一个新对象实例；
2. 将构造函数的作用域赋给新对象实例
3. 执行构造函数中的代码，为新对象实例添加属性
4. 返回新对象实例

```
function Person() {
  this.sayHello = function (params) {
    console.log('hello');
  };
}

Person.prototype.name = 'person';
const person1 = new Person();

// Person.prototype === person1.__proto__
console.log(JSON.stringify(Person.prototype)); // {"name":"person"}
console.log(JSON.stringify(person1.__proto__)); // {"name":"person"}
console.log(JSON.stringify(person1.prototype)); // undefined
console.log(Person.prototype === person1.__proto__); // true
console.log(`getPrototypeOf:  ${Object.getPrototypeOf(person1) === Person.prototype}`); //
console.log(`constructor:  ${Person.prototype.constructor === Person}`); // true

console.log(person1.__proto__.constructor === Person.prototype.constructor); // true
console.log(person1.__proto__.constructor === Person); // true
console.log(person1.__proto__.constructor.prototype === Person.prototype); // true

// 对象实例中有一个constructor属性，指向构造函数
console.log(JSON.stringify(person1.constructor === Person))
```



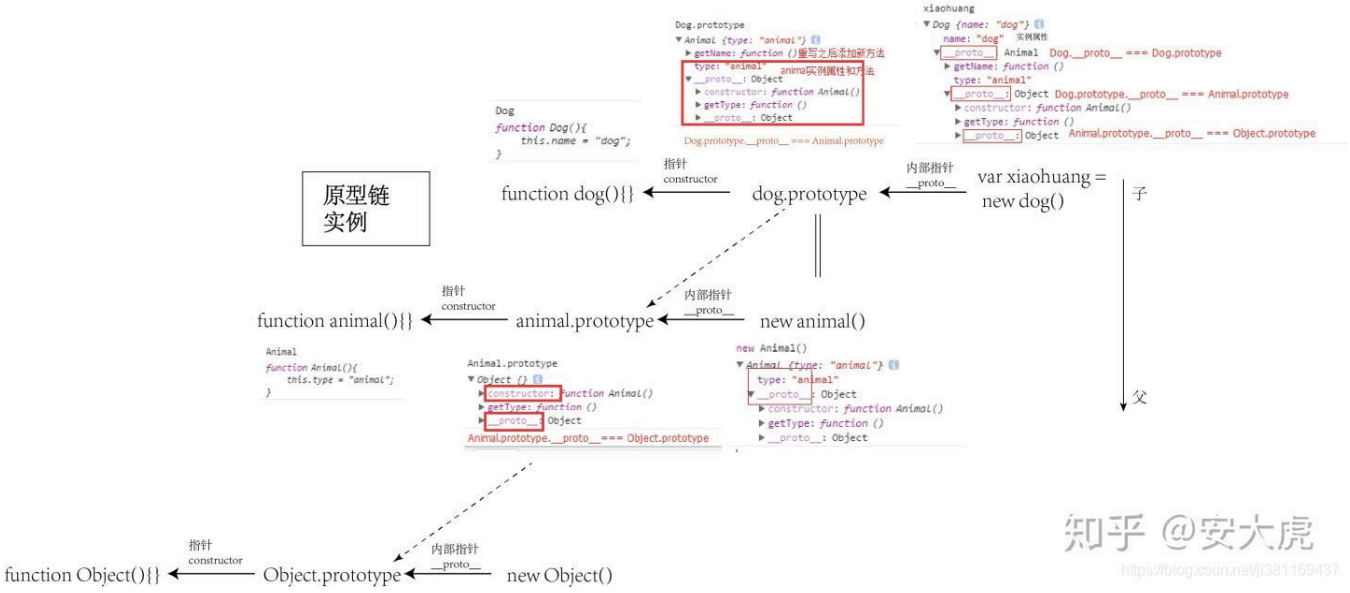


继承：通过原型链来实现。

本质：重写原型对象，代之以一个新类型的实例。

思想：利用原型让一个引用类型继承另一个引用类型的属性和方法。

@guifangzhang



最后一张图片的代码

```
function Animal(type) {
  this.type = type || 'animal';

  this.getType = function getType() {

  };
}

function Dog() {
  this.name = 'dog';
}

Dog.prototype = new Animal();

const dog = new Dog();

console.log(dog);
```

闭包

- 闭包是指有权访问另一个函数作用域中变量的函数,
- 创建闭包的最常见的方式就是在一个函数内创建另一个函数,
- 通过另一个函数访问这个函数的局部变量,利用闭包可以突破作用链域, 将函数内部的变量和方法传递到外部

```
<ul id="testUL">
  <li> index = 0</li>
  <li> index = 1</li>
  <li> index = 2</li>
  <li> index = 3</li>
</ul>
```

```
const nodes = document.getElementsByTagName("li");
for (i = 0; i < nodes.length; i += 1) {
  // nodes[i].onclick = function () {
  //   console.log(i);   //不用闭包的话，console.log每次都是4
  // }
  // 闭包写法
  // 声明一个匿名函数作为闭包的外层
  // 调用outterFunc时将i作为参数index的值传入；
  // outterFunc方法返回的function(){console.log(idnex)}作为onclick的事件处理函数
  function outterFunc (index) {
    return function() {
      console.log(index);
    }
  }
  nodes[i].onclick = outterFunc(i);
}
// 那如果使用事件冒泡就也可以实现
const ulNode = document.getElementById("testUL");
ulNode.onclick = function (e) {
  console.log(e.target.innerText)
}
```

React

Portals

React中文Partals教程

> Portal 提供了一种将子节点渲染到存在于父组件以外的 DOM 节点的优秀的方案。

```
ReactDOM.createPortal(child, container)
// 第一个参数（child）是任何可渲染的 React 子元素，例如一个元素，字符串或 fragment。第二个参数（contai
```

一个 portal 的典型用例是当父组件有 `overflow: hidden` 或 `z-index` 样式时，但你需要子组件能够在视觉上“跳出”其容器。例如，对话框、悬浮卡以及提示框

> 当在使用 portal 时，记住管理键盘焦点就变得尤为重要。
>
> 对于模态对话框，通过遵循 WAI-ARIA 模态开发实践，来确保每个人都能够运用它。

通过 Portal 进行事件冒泡

尽管 portal 可以被放置在 DOM 树中的任何地方，但在任何其他方面，其行为和普通的 React 子节点行为一致

TypeScript

进阶用法

泛型

> 泛型就是不预先确定的数据类型，具体的类型在使用的时候再确定的一种类型约束规范

泛型的好处：

- 函数和类可以轻松的支持多种类型，增强程序的扩展性
- 不必写多条函数重载，冗长的联合类型声明，增强代码的可读性
- 灵活控制类型之间的约束

泛型可以应用于 function、interface、type 或者 class 中。但是注意，「泛型不能应用于类的静态成员」

工具泛型

```
interface TestInterface {
    a: string;
    b: number;
    c?: string;
}
interface Person {
    name: string;
    age: number;
}
const testObj:TestInterface = {a: '1', b: 1}
```

Partial

Partial的作用就是将传入的属性变为可选。

```
const testObj:Partial<TestInterface> = {a: '1'}
```

Required

Required 的作用是将传入的属性变为必选项

```
// Right
const testObjReqired:Required<TestInterface> = { a:'1', b:1, c: '1'}
```

Readonly

将传入的属性变为只读选项

Record

该类型可以将 K 中所有的属性的值转化为 T 类型

```
/**
 * Construct a type with a set of properties K of type T
 */
type Record<K extends keyof any, T> = {
    [P in K]: T;
};
```

```
const testObjRecord:Record<'a', Person> = {a: {name: '1', age: 1}}
```

Pick

从 T 中取出 一系列 K 的属性

```
const testObjPick:Pick<TestInterface, 'a'> = {a: '1'}
```

Exclude

Exclude 将某个类型中属于另一个的类型移除掉。

Extract

Extract 的作用是提取出 T 包含在 U 中的元素，换种更加贴近语义的说法就是从 T 中提取出 U

类型断言

推荐类型断言的预发使用 as关键字，而不是<> ,防止歧义
类型断言并非类型转换，类型断言发生在编译阶段。类型转换发生在运行时

函数重载

函数重载的基本语法：

```
declare function test(a: number): number;
declare function test(a: string): string;

const resS = test('Hello World');  // resS 被推断出类型为 string；
const resN = test(1234);           // resN 被推断出类型为 number;
```

这里我们申明了两次？！为什么我不能判断类型或者可选参数呢？后来我遇到这么一个场景，

```
interface User {
  name: string;
  age: number;
}

declare function test(para: User | number, flag?: boolean): number;
```

在这个 test 函数里，我们的本意可能是当传入参数 para 是 User 时，不传 flag，当传入 para 是 number 时，传入 flag。TypeScript 并不知道这些，当你传入 para 为 User 时，flag 同样允许你传入：

```
const user = {
  name: 'Jack',
  age: 666
}

// 没有报错，但是与想法违背
const res = test(user, false);
```

使用函数重载能帮助我们实现：

```
interface User {
  name: string;
  age: number;
}

declare function test(para: User): number;
declare function test(para: number, flag: boolean): number;

const user = {
  name: 'Jack',
  age: 666
};

// bingo
// Error: 参数不匹配
const res = test(user, false);
```

前端100问笔记

第1题

第 1 题：写 React / Vue 项目时为什么要在列表组件中写 key，其作用是什么？

Advanced-Frontend/Daily-Interview-Question#1

key的作用是为了在diff算法执行时更快的找到对应的节点，提高diff速度

第2题

第 2 题：['1', '2', '3'].map(parseInt) what & why ?

原文：https://github.com/Advanced-Frontend/Daily-Interview-Question/issues/4

parseInt(string, radix)接收两个参数，第一个表示被处理的值（字符串），第二个表示为解析时的基数。

> string要被解析的值。如果参数不是一个字符串，则将其转换为字符串(使用 ToString 抽象操作)。字符串开头的空白符将会被忽略。

> radix 可选从 2 到 36，表示字符串的基数。例如指定 16 表示被解析值是十六进制数。请注意，10不是默认值！

- parseInt('1', 0) //radix 为 0 时，且 string 参数不以“0x”和“0”开头时，按照 10 为基数处理。这个时候返回 1；
- parseInt('2', 1) // 1不是可选，所以NaN
- parseInt('3', 2) // 基数为 2（2 进制）表示的数中，最大值小于 3，所以无法解析，返回 NaN。

返回NaN情况：

- radix 小于 2 或大于 36 （radix为0的情况要参考文档里面具体介绍）
- 第一个非空格字符不能转换为数字。

MDN中有详细解释：

https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/parseInt

Map的MDN：
https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Array/map

```
['1', '1', '3'].map(parseInt)
['1', '0', '3'].map(parseInt)
['1', 1, '3'].map(parseInt)
答案都是： [1, NaN, NaN]

parseInt('2', 1)
parseInt('1', 1)
这俩都是 NaN
```

第3题

第 3 题：什么是防抖和节流？有什么区别？如何实现？

> 节流和防抖都是用来处理高频事件的，防抖是在高频事件最后一个事件触发n秒后执行，节流是让高频事件在n秒内只执行一次

防抖

> 触发高频事件后n秒内函数只会执行一次，如果n秒内高频事件再次被触发，则重新计算时间

- 思路：

> 每次触发事件时都取消之前的延时调用方法

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
    <input id="name" />
    <script>
      function debounce (fn, delay) {
          let timer = null;
          return function () {
            clearTimeout(timer);
            timer = setTimeout(() => {
              console.log(arguments); // Arguments [InputEvent, callee: ƒ, Symbol(Symbol.i
              fn.apply(this, arguments);
            }, delay);
          };
      }

      function onInputChange() {
        console.log('Input change')
      }

      const inputEl = document.getElementById('name');
      inputEl.addEventListener('input', debounce(onInputChange, 500))
    </script>
</body>
</html>
```

节流

> 高频事件触发，但在n秒内只会执行一次，所以节流会稀释函数的执行频率

- 思路：

> 每次触发事件时都判断当前是否有等待执行的延时函数

第4题

第 4 题：介绍下 Set、Map、WeakSet 和 WeakMap 的区别？

> https://es6.ruanyifeng.com/#docs/set-map

Set

> 它类似于数组，但是成员的值都是唯一的，没有重复的值。

```
// 去除数组的重复成员
[...new Set(array)]

// 去除字符串里面的重复字符。
[...new Set('ababbc')].join('')
// "abc"
```

> 向 Set 加入值的时候，不会发生类型转换

Set 实例的属性:

- Set.prototype.constructor：构造函数，默认就是Set函数。
- Set.prototype.size：返回Set实例的成员总数。

操作方法

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

遍历方法

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

> keys方法和values方法的行为完全一致

第5题

第 5 题：介绍下深度优先遍历和广度优先遍历，如何实现？

考察树状结构

权限项目的 表格 + checkbox 处理场景