第三版：JavaScript 24 道

有哪些数据类型?

根据 JavaScript 中的变量类型传递方式，分为基本数据类型和引用数据类型两大类七种。

基本数据类型包括 Undefined、Null、Boolean、Number、String、Symbol (ES6 新增)六种。 引用数据类型只有 Object 一种，主要包括对象、数组和函数。

判断数据类型采用 typeof 操作符，有两种语法：

```
typeof 123;//语法一

const FG = 123;
typeof FG;//语法二

typeof(null) //返回 object;
null == undefined //返回 true，因为 undefined 派生自 null;
null === undefined //返回 false。
```

基本数据类型和引用数据类型有什么区别?

两者作为函数的参数进行传递时：

1、 基本数据类型传入的是数据的副本，原数据的更改不会影响传入后的数据。

2、 引用数据类型传入的是数据的引用地址，原数据的更改会影响传入后的数据。

两者在内存中的存储位置：

1、 基本数据类型存储在栈中。

2、 引用数据类型在栈中存储了指针，该指针指向的数据实体存储在堆中。

判断数据类型的方法有哪些?

1、 利用 typeof 可以判断数据的类型；

2、 A instanceof B 可以用来判断 A 是否为 B 的实例，但它不能检测 null 和 undefined；

3、 B.constructor == A 可以判断 A 是否为 B 的原型，但 constructor 检测 Object 与 instanceof 不一样，还可以处理基本数据类型的检测。

不过函数的 constructor 是不稳定的，这个主要体现在把类的原型进行重写，在重写的过程中很有可能出现把之前的 constructor 给覆盖了，这样检测出来的结果就是不准确的。

4、 Object.prototype.toString.call()

Object.prototype.toString.call() 是最准确最常用的方式。

与深拷贝有何区别？如何实现?

浅拷贝只复制指向某个对象的指针，而不复制对象本身。浅拷贝的实现方式有：

1、 Object.assign()：需注意的是目标对象只有一层的时候，是深拷贝；

2、 扩展运算符；

深拷贝就是在拷贝数据的时候，将数据的所有引用结构都拷贝一份。深拷贝的实现方式有：

1、 手写遍历递归赋值；

2、 结合使用 JSON.parse()和 JSON.stringify()方法。

什么是执行上下文和执行栈?

变量或函数的执行上下文，决定了它们的行为以及可以访问哪些数据。每个上下文都有一个关联的变量对象，而这个上下文中定义的所有变量和函数都存在于这个对象上(如 DOM 中全局上下文关联的便是 window 对象)。
每个函数调用都有自己的上下文。当代码执行流进入函数时，函数的上下文被推到一个执行栈中。在函数执行完之后，执行栈会弹出该函数上下文，在其上的所有变量和函数都会被销毁，并将控制权返还给之前的执行上下文。 JS 的执行流就是通过这个执行栈进行控制的。

什么是作用域和作用域链?

作用域可以理解为一个独立的地盘，可以理解为标识符所能生效的范围。作用域最大的用处就是隔离变量，不同作用域下同名变量不会有冲突。ES6 中有全局作用域、函数作用域和块级作用域三层概念。
当一个变量在当前块级作用域中未被定义时，会向父级作用域(创建该函数的那个父级作用域)寻找。如果父级仍未找到，就会再一层一层向上寻找，直到找到全局作用域为止。这种一层一层的关系，就是作用域链 。

作用域和执行上下文的区别是什么?

1、 函数的执行上下文只在函数被调用时生成，而其作用域在创建时已经生成；

2、 函数的作用域会包含若干个执行上下文(有可能是零个，当函数未被调用时)。

this 指向的各种情况都有什么?

this 的指向只有在调用时才能被确定，因为 this 是执行上下文的一部分。

全局作用域中的函数：其内部 this 指向 window：

```
var a = 1;
function fn(){
console.log(this.a)
}
fn() //输出 1
```

对象内部的函数：其内部 this 指向对象本身：

```
var a = 1;
var obj = {
a:2,
fn:function(){
console.log(this.a)
}
}

obj.fn() //输出 2
```

构造函数：其内部 this 指向生成的实例：

```
function createP(name,age){
this.name = name //this.name 指向 P
this.age = age //this.age 指向 P
}
var p = new createP("老李",46)
```

由 apply、call、bind 改造的函数：其 this 指向第一个参数：

```
function add(c,d){
return this.a + this.b + c + d
}
var o = {a:1,b:2)
add.call(o,5,7) //输出 15
```

箭头函数：箭头函数没有自己的 this，看其外层的是否有函数，如果有，外层函数的 this 就是内部箭头函数的 this，如果没有，则 this 是 window。

如何改变 this 指针的指向?

可以使用 apply、call、bind 方法改变 this 指向(并不会改变函数的作用域)。比较如下：

1、 三者第一个参数都是 this 要指向的对象，也就是想指定的上下文，上下文就是指调用函数的那个对象(没有就指向全局 window)；

2、 bind 和 call 的第二个参数都是数组，apply 接收多个参数并用逗号隔开；

3、 apply 和 call 只对原函数做改动，bind 会返回新的函数(要生效还得再调用一次)。

什么是闭包?

闭包就是引用了其他函数作用域中变量的函数，这种模式通常在函数嵌套结构中实现。里面的函数可以访问外面函数的变量，外面的变量的是这个内部函数的一部分。闭包有如下作用：

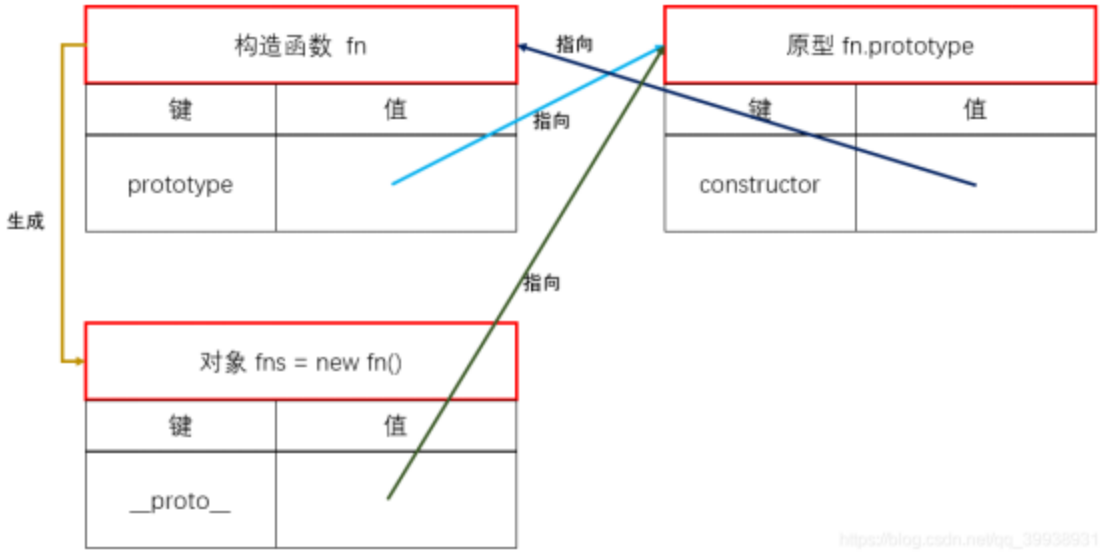
1、 加强封装，模拟实现私有变量；

2、 实现常驻内存的变量。

闭包不能滥用，否则会导致内存泄露，影响网页的性能。闭包使用完了后，要立即释放资源，将引用变量指向 null。

什么是原型、原型链?

原型：JS 声明构造函数(用来实例化对象的函数)时，会在内存中创建一个对应的对象，这个对象就是原函数的原型。构造函数默认有一个 prototype 属性，prototype 的值指向函数的原型。同时原型中也有一个 constructor 属性，constructor 的值指向原函数。
通过构造函数实例化出来的对象，并不具有 prototype 属性，其默认有一个 __proto__属性，__proto__的值指向构造函数的原型对象。在原型对象上添加或修改的属性，在所有实例化出的对象上都可共享。



当在实例化的对象中访问一个属性时，首先会在该对象内部寻找，如找不到，则会向其__proto__指向的原型中寻找，如仍找不到，则继续向原型中__proto__指向的上级原型中寻找，直至找到或 Object.prototype 为止，这种链状过程即为原型链。

何为防抖和节流？如何实现？

1、 防抖和节流都是防止短时间内高频触发事件的方案。

2、 防抖的原理是：如果一定时间内多次执行了某事件，则只执行其中的最后一次。

3、 节流的原理是：要执行的事件每隔一段时间会被冷却，无法执行。

4、 应用场景有：搜索框实时搜索，滚动改变相关的事件。

```
//@fn: 要执行的函数
//@delay: 设定的时限
```

```
//防抖函数
function debunce(fn, delay) {
    let flag = null;
    return function() {
        if (flag) clearTimeout(flag)
        //利用 apply 改变函数指向，使得封装后的函数可以接收 event 本身
        flag = setTimeout(() => fn.apply(this, arguments), delay)
    }
}
//节流函数
function throttle(fn, delay) {
    let flag = true;
    return function() {
        if (!flag) return false;
        flag = false;
        setTimeout(() => {
            fn.apply(this, arguments)
            flag = true
        }, delay)
    }
}
```

如何理解同步和异步?

同步：按照代码书写顺序一一执行处理指令的一种模式，上一段代码执行完才能执行下一段代码。

异步：可以理解为一种并行处理的方式，不必等待一个程序执行完，可以执行其它的任务。

JS 之所以需要异步的原因在于 JS 是单线程运行的。常用的异步场景有：定时器、ajax 请求、事件绑定。

JS 是如何实现异步的?

JS 引擎是单线程的，但又能实现异步的原因在于事件循环和任务队列体系。

事件循环：

1、 JS 会创建一个类似于 while (true) 的循环，每执行一次循环体的过程称之为 Tick。每次 Tick 的过程就是查看是否有待处理事件，如果有则取出相关事件及回调函数放入执行栈中由主线程执行。待处理的事件会存储在一个任务队列中，也就是每次 Tick 会查看任务队列中是否有需要执行的任务。

任务队列：

1、 异步操作会将相关回调添加到任务队列中。而不同的异步操作添加到任务队列的时机也不同，如 onclick, setTimeout, ajax 处理的方式都不同，这些异步操作是由浏览器内核的 webcore 来执行的，浏览器内核包含 3 种 webAPI，分别是 DOM Binding、network、timer 模块。

2、 onclick 由 DOM Binding 模块来处理，当事件触发的时候，回调函数会立即添加到任务队列中。 setTimeout 由 timer 模块来进行延时处理，当时间到达的时候，才会将回调函数添加到任务队列中。 ajax 由 network 模块来处理，在网络请求完成返回之后，才将回调添加到任务队列中。

主线程：

1、 JS 只有一个线程，称之为主线程。而事件循环是主线程中执行栈里的代码执行完毕之后，才开始执行的。所以，主线程中要执行的代码时间过长，会阻塞事件循环的执行，也就会阻塞异步操作的执行。

2、 只有当主线程中执行栈为空的时候（即同步代码执行完后），才会进行事件循环来观察要执行的事件回调，当事件循环检测到任务队列中有事件就取出相关回调放入执行栈中由主线程执行。

什么是 AJAX？如何实现？

ajax 是一种能够实现局部网页刷新的技术，可以使网页异步刷新。
ajax 的实现主要包括四个步骤：

1、 创建核心对象 XMLhttpRequest；

2、 利用 open 方法打开与服务器的连接；

3、 利用 send 方法发送请求；（"POST"请求时，还需额外设置请求头）

4、 监听服务器响应，接收返回值。

```
//1-创建核心对象
//该对象有兼容问题，低版本浏览器应使用 ActiveXObject
const xthhp = new XMLHttpRequest();
//2-连接服务器
//open(method,url,async)
xhttp.open("POST", "http://localhost:3000", true)
//设置请求头
xmlHttp.setRequestHeader("Content-Type",
"application/x-www-form-urlencoded");
```

```
//3-发送请求
//send 方法发送请求参数，如为 GET 方法，则在 open 中 url 后拼接
xhttp.send({
    _id: 123
})
//4-接收服务器响应
//onreadystatechange 事件，会在 xhttp 的状态发生变化时自动调用
xhttp.onreadystatechange = function() {
    //状态码共 5 种：0-未 open  1-已 open  2-已 send  3-读取响应  4-响应读取结束
    if (xhttp.readyState == 4 && xhttp.status == 200) {
        alert("ajax 请求已完成")
    }
}
```

实现异步的方式有哪些?

1、 回调函数模式：将需要异步执行的函数作为回调函数执行，其缺点在于处理复杂逻辑异步逻辑时，会造成回调地狱(回调嵌套层数太多，代码结构混乱)；

2、 事件监听模式：采用事件驱动的思想，当某一事件发生时触发执行异步函数，其缺点在于整个代码全部得变为事件驱动模式，难以分辨主流程；

3、 发布订阅模式：当异步任务执行完成时发布消息给信号中心，其他任务通过在信号中心中订阅消息来确定自己是否开始执行；

4、 Promise(ES6)：Promise 对象共有三种状态 pending(初始化状态)、fulfilled(成功状态)、rejected(失败状态)。

5、 async/await(ES7)：基于 Promise 实现的异步函数； （6）利用生成器实现。

怎么理解 Promise 对象?

Promise 对象有如下两个特点：

1、对象的状态不受外界影响。Promise 对象共有三种状态 pending、fulfilled、rejected。状态值只会被异步结果决定，其他任何操作无法改变。

2、 状态一旦成型，就不会再变，且任何时候都可得到这个结果。状态值会由 pending 变为 fulfilled 或 rejected，这时即为 resolved。

Promise 的缺点有如下三个缺点：

1、 Promise 一旦执行便无法被取消；

2、 不可设置回调函数，其内部发生的错误无法捕获；

3、 当处于 pending 状态时，无法得知其具体发展到了哪个阶段。

Pomise 中常用的方法有：

1、 Promise.prototype.then()：Promise 实例的状态发生改变时，会调用 then 内部的回调函数。then 方法接受两个参数（第一个为 resolved 状态时时执行的回调，第一个为 rejected 状态时时执行的回调）

2、 Promise.prototype.catch()：.then(null, rejection) 或.then(undefined, rejection)的别名，用于指定发生错误时的回调函数。

怎么理解宏任务，微任务？？？

1、 宏任务有：script(整体代码)、setTimeout、setInterval、I/O、页面渲染；

2、 微任务有：Promise.then、Object.observe、MutationObserver。

3、 执行顺序大致如下：

4、 主线程任务——>宏任务——>微任务——>微任务里的宏任务——>.....——>直到任务全部完成

什么是跨域？怎么解决跨域问题？

跨域问题实际是由同源策略衍生出的一个问题，当传输协议、域名、端口任一部分不一致时，便会产生跨域问题，从而拒绝请求，但<img src=XXX> <link href=XXX> <script src=XXX>;天然允许跨域加载资源。解决方案有：

JSONP

1、 原理：利用<script>;标签没有跨域限制的漏洞，使得网页可以得到从其他来源动态产生的 JSON 数据（前提是服务器支持）。

2、 优点：实现简单，兼容性好。

3、 缺点：仅支持 get 方法，容易受到 XSS 攻击。

CORS

1、 原理：服务器端设置 Access-Control-Allow-Origin 以开启 CORS。该属性表示哪些域名可以访问资源，如设置通配符则表示所有网站均可访问。

2、 实现实例（express）：

```
//app.js 中设置
var app = express();
//CORS 跨域
--------------------------------------------------------------------------------
------
// CORS：设置允许跨域中间件
var allowCrossDomain = function(req, res, next) {
    // 设置允许跨域访问的 URL(* 表示允许任意 URL 访问)
    res.header("Access-Control-Allow-Origin", "*");
    // 设置允许跨域访问的请求头
    res.header("Access-Control-Allow-Headers",
"X-Requested-With,Origin,Content-Type,Accept,Authorization");
    // 设置允许跨域访问的请求类型
    res.header("Access-Control-Allow-Methods",
"PUT,POST,GET,DELETE,OPTIONS");
    // 设置允许服务器接收 cookie
    res.header('Access-Control-Allow-Credentials', 'true');
    next();
};
app.use(allowCrossDomain);
//--------------------------------------------------------------------------------
-------
```

Node 中间件代理

原理：同源策略仅是浏览器需要遵循的策略，故搭建中间件服务器转发请求与响应，达到跨域目的。

```
/* server1.js 代理服务器(http://localhost:3000)*/
const http = require('http')
// 第一步：接受客户端请求
const server = http.createServer((request, response) = > {
    // 代理服务器，直接和浏览器直接交互，需要设置 CORS 的首部字段
    response.writeHead(200, {
        'Access-Control-Allow-Origin': '*',
        'Access-Control-Allow-Methods': '*',
        'Access-Control-Allow-Headers': 'Content-Type'
    })
    // 第二步：将请求转发给服务器
    const proxyRequest = http.request({
        host: '127.0.0.1',
        port: 4000,
        url: '/',
        method: request.method,
        headers: request.headers
    }, serverResponse = > {
        // 第三步：收到服务器的响应
        var body = ''
        serverResponse.on('data', chunk = > {
            body += chunk
        })
        serverResponse.on('end', () = > {
            console.log('The data is ' + body)
            // 第四步：将响应结果转发给浏览器
            response.end(body)
        })
```

```
    }).end()
})
server.listen(3000, () = > {
    console.log('中间件服务器地址：  http://localhost:3000')
})
// server2.js(http://localhost:4000)
const http = require("http");
const data = {
    title: "fontend",
    password: "123456"
};
const server = http.createServer((request, response) = > {
    if (request.url === "/") {
        response.end(JSON.stringify(data));
    }
});
server.listen(4000, () = > {
    console.log("The server is running at http://localhost:4000");
});
```

nginx 反向代理

1、 原理：类似 Node 中间件服务器，通过 nginx 代理服务器实现。

2、 实现方法：下载安装 nginx，修改配置。

实现继承的方法有哪些？？？

实现继承的方法有：

class+extends 继承（ES6）

```
//类模板
class Animal {
    constructor(name) {
        this.name = name
    }
}
//继承类
class Cat extends Animal { //重点。extends 方法，内部用 constructor+super
    constructor(name) {
        super(name);
        //super 作为函数调用时，代表父类的构造函数
    } //constructor 可省略
    eat() {
        console.log("eating")
    }
}
```

原型继承

```
//类模板
function Animal(name) {
    this.name = name;
}
//添加原型方法
Animal.prototype.eat = function() {
    console.log("eating")
}

function Cat(furColor) {
```

```
    this.color = color;
};
//继承类
Cat.prototype = new Animal() //重点：子实例的原型等于父类的实例
```

借用构造函数继承

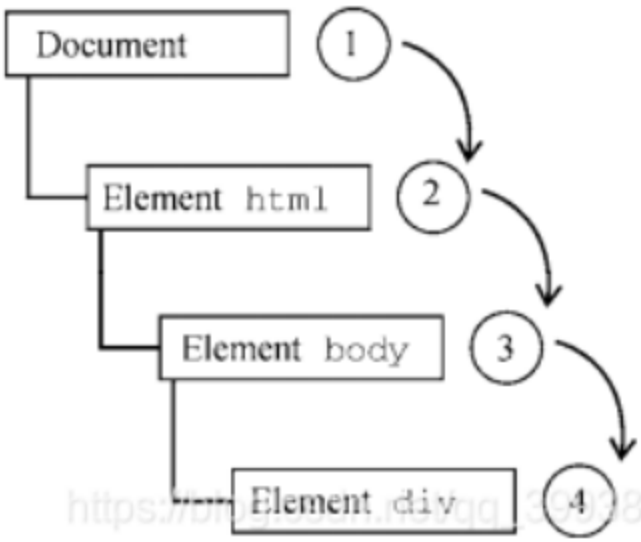
```
function Animal(name){
    this.name = name
}
function Cat(){
    Animal.call(this,"CatName")//重点，调用父类的 call 方法
}
```

寄生组合式继承（重点）

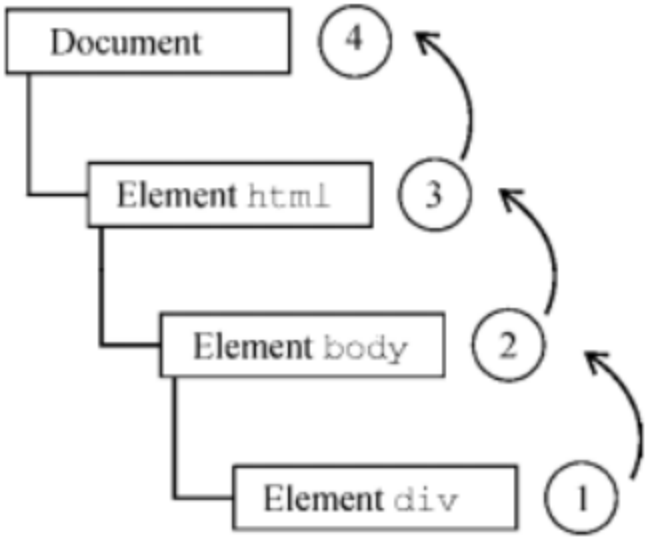
DOM 事件模型和事件流?

DOM 事件模型包括事件捕获(自上而下触发)与事件冒泡(自下而上触发，ie 用的就是冒泡)机制。基于事件冒泡机制可以完成事件代理。

事件捕获



事件冒泡



DOM 事件流包括三个阶段事件捕获阶段、处于目标阶段、事件冒泡阶段。

EventLoop 事件循环是什么?

js 是一门单线程的需要，它的异步操作都是通过事件循环来完成的。整个事件循环大体由执行栈、消息队列和微任务队列三个部分组成。

同步代码会直接在执行栈中调用执行。

定时器中的回调会在执行栈被清空且定时达成时推入执行栈中执行。

promise、async 异步函数的回调会被推入到微任务队列中，当执行栈被清空且异步操作完成时立即执行。

require/import 之间的区别?

1、 require 是 CommonJS 语法，import 是 ES6 语法；

2、 require 只在后端服务器支持，import 在高版本浏览器及 Node 中都可以支持；

3、 require 引入的是原始导出值的复制，import 则是导出值的引用；

4、 require 时运行时动态加载，import 是静态编译；

5、 require 调用时默认不是严格模式，import 则默认调用严格模式.