JS

Reflect

Proxy

Proxy对象用于创建一个对象的代理，从而实现基本操作的拦截和自定义（如属性查找、赋值、枚举、函数调用）

```JavaScript
const p = new Proxy(target, handler)
```

js数组方法

forEach map push pop shift unshift splice slice concat join sort reverse some every filter

Promise

Promise 对象是异步编程的一种解决方案。Promise 是一个构造函数，接收一个函数作为参数，返回一个 Promise 实例。一个 Promise 实例有三种状态，分别是pending、*fulfilled* 和 rejected。实例的状态只能由 pending 转变 *fulfilled* 或者 rejected 状态，并且状态一经改变，无法再被改变了。

状态的改变是通过传入的 resolve() 和 reject() 函数来实现的，当我们调用resolve回调函数时，会执行Promise对象的then方法传入的第一个回调函数，当我们调用reject回调函数时，会执行Promise对象的then方法传入的第二个回调函数，或者catch方法传入的回调函数。

Promise的实例有**两个过程**：

- pending –> fulfilled : Resolved（已完成）
- pending –> rejected：Rejected（已拒绝）

一旦从进行状态变成为其他状态就永远不能更改状态了。

在通过new创建Promise对象时，我们需要传入一个回调函数，我们称之为executor

✓ 这个回调函数会被立即执行，并且给传入另外两个回调函数resolve、reject；

✓ 当我们调用resolve回调函数时，会执行Promise对象的then方法传入的回调函数；

✓ 当我们调用reject回调函数时，会执行Promise对象的catch方法传入的回调函数；

情况一：如果resolve传入一个普通的值或者对象，那么这个值会作为then回调的参数；

情况二：如果resolve中传入的是另外一个Promise，那么这个新Promise会决定原Promise的状态：

情况三：如果resolve中传入的是一个对象，并且这个对象有实现then方法，那么会执行该then方法，并且根据then方法的结 果来决定Promise的状态：

then方法接受两个参数：

fulfilled的回调函数：当状态变成fulfilled时会回调的函数；

reject的回调函数：当状态变成reject时会回调的函数；

Promise有三种状态，那么这个Promise处于什么状态呢?

当then方法中的回调函数本身在执行的时候，那么它处于pending状态；

当then方法中的回调函数返回一个结果时，那么它处于fulfilled状态，并且会将结果作为resolve的参数；

✓ 情况一：返回一个普通的值；

✓ 情况二：返回一个Promise；

✓ 情况三：返回一个thenable值；

当then方法抛出一个异常时，那么它处于reject状态

Promise有五个常用的方法：then()、catch()、all()、race()、finally。

Promise.allSettled() 方法以 promise 组成的可迭代对象作为输入，并且返回一个 Promise 实例。当输入的所有 promise 都已敲定时（包括传递空的可迭代类型），返回的 promise 将兑现，并带有描述每个 promsie 结果的对象数组。

async & await

ES7提出的关于异步的终极解决方案

async/await是Generator的语法糖

- 内置执行器：Generator函数的执行必须靠执行器，不能一次执行完成
- 可读性更好：async和 await，比起使用 *号和 yield，语义清晰明了

如果不使用async/await的话，Promise需要通过链式调用执行then之后的代码

Promise搭配async/await的使用才是正解！

async/await基于Promise。async把promise包装了一下，async函数更简洁，不需要像promise一样需要写then，不需要写匿名函数处理promise的resolve值。

async是Generator函数的语法糖，**async函数返回值是promise对象**，比generator函数返回值 iterator 对象更方便，*可使用 await 代替then 指定下一步操作(await==promise.then)*

原型与原型链

每个构造函数都有一个prototype属性，该属性指向的就是显示原型对象，这个对象包含了可以由该构造函数的所有实例共享的属性和方法，每个实例对象上有一个 `__proto__` 属性，该属性指向的就是隐式原型对象。

查找一个属性先在自身查找，如果找不到，就沿着 `__proto__` 属性在原型对象上进行查找，如果还找不到，就沿着原型对象的 `__proto__` 属性进行查找，直到查找到直到找到Object的原型对象，如果还没有找到就会返回undefined，沿着__proto__查找属性(方法)的这条链就是原型链。

原型链终点是 `Object.prototype.__proto__`

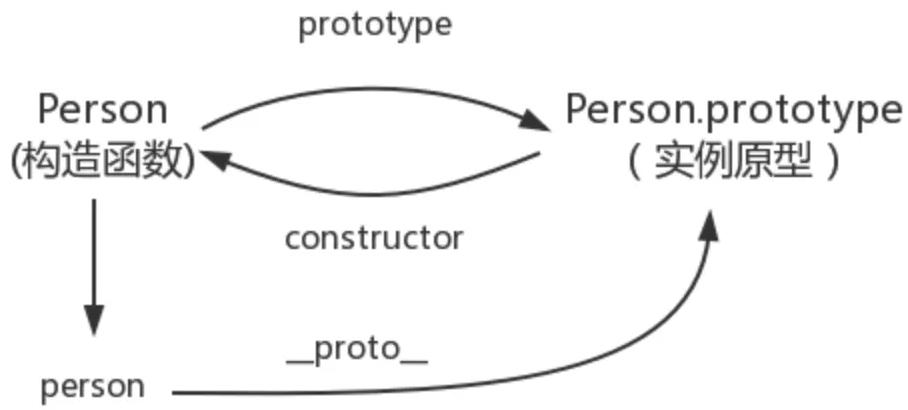
使用 `hasOwnProperty()` 方法来判断属性是否属于原型链的属性：

每个实例对象都有私有属性（**proto**）指向它构造函数的原型对象。

每个构造函数都有prototype原型对象

prototype原型对象的constructor指向构造函数本身

有默认constructor属性，记录实例由哪个构造函数创建



ES6新特性

1. const、let
2. 模板字符串
3. 箭头函数
4. 函数参数默认值
5. 解构赋值
6. for...of 用于数组，for...in用于对象
7. Promise
8. 展开运算符(...)
9. 对象字面量、class(原型链的语法糖)

匿名函数

匿名函数在声明时不用带上函数名 没有函数提升

匿名函数可以有效的保证在页面上写入Javascript，而不会造成全局变量的污染。

不改变原数组

- concat() map()

返回新数组

- slice(start,end)左闭右开，可以为负数

返回一个包含原有数组中一个或多个元素的新数组

- filter

判断所有元素，将满足条件的元素作为一个新的数组返回

- join()
- find()
- findIndex()
- indexOf()
- includes()

let const var 相关

var没有块级作用域，只有函数作用域。var只有在function{ }内部才有作用域的概念，其他地方没有。意味着函数以外用var定义的变量是同一个，我们所有的修改都是针对他的

1. let和const增加**块级作用域**（JS没有块级作用域）
2. let和const存在**暂时性死区**，不存在**变量提升**，不能在初始化前引用，调用 返回 `uninitialized`
3. let和const禁止**重复声明**，不能重新声明
4. let和`const`不会成为全局对象属性，var声明的变量自动成为全局对象属性
5. var 存在变量提升（执行前，编译器对代码预编译，当前作用域的变量/函数提升到作用域顶部），`let`约束变量提升。let和var都发生了变量提升，只是`es6`进行了约束，在我们看来，就像`let`禁止了变量提升
6. 使用`var`，我们能对变量多次声明，后面声明的变量会覆盖前面的声明

```
var a = 123
if (true) {
    a = 'abc' // ReferenceError 因为下面的 let
    let a;
}
```

const实际保证的并不是变量的值，而是变量指向的内存地址

闭包

内部函数 可以访问其外部函数中声明变量，调用 外部函数返回 内部函数后，即使 外部函数执行结束了，但 内部函数引用外部函数的变量依然保存在内存 ，这些变量的集合——闭包

作用

1. 独立作用域，避免变量污染
2. 实现缓存计算结果，延长变量生命周期
3. 创建私有变量

运用：

防抖节流 模拟块级作用域 对象中创建私有变量

内存管理GC

栈中的变量js会自动清除

JS单线程机制，GC过程阻碍了主线程 执行

堆内存中的变量只有在 所有对它的引用都 结束 时被回收

自动垃圾回收机制：找出不使用的值，释放内存

函数运行结束，没有闭包或引用，局部变量被 标记 清除

全局变量：浏览器卸载页面 被清除

引用：显式引用（对象有对其属性的引用） 和 隐式引用（对象对其原型的引用）

引用计数 标记清除

💜 事件流?

JS和HTML的交互 通过 事件 实现，使用侦听器 预定事件，便于事件发生时执行相应代码

手指放在一组同心圆的圆心上，手指指向不是一个圆 ，而是纸上的所有圆，单击按钮时 单击事件不止发生在按钮上，同时 也单击了按钮的容器元素，甚至也单击了整个页面

事件流描述 从页面接收事件的顺序

事件发生时会在元素节点和根节点之间按照特定的顺序传播，路径所经过的节点都会收到该事件——DOM事件流

1. 捕获：不太具体的节点应该更早接收到事件，而最具体的节点最后收到事件。目的是在事件到达预定目标之前捕获它
2. 冒泡：事件开始由最具体的元素接收，逐级向上传播到不具体的节点，document对象首先收到click事件，事件沿着DOM树依次往下，传播到事件的具体目标
3. DOM标准规定事件流包括3个阶段：事件捕获、处于目标阶段和事件冒泡

- 事件捕获——为截获事件提供机会
- 处于目标阶：事件在 上发生并处理
- 冒泡阶段：事件又传播回文档

所有事件都要经过捕获阶段和处于目标阶段

focus(获得输入焦点)和失去焦点blur事件没有冒泡，无法委托

原始事件模型

```
Plain Text
<input type="button" onclick="fun()">
var btn = document.getElementById('.btn');
btn.onclick = fun;
```

- 绑定速度快

页面还未完全加载，事件可能无法正常运行

- 只支持冒泡，不支持捕获
- 同一个类型的事件只能绑定一次

```
<input type="button" id="btn" onclick="fun1()">

var btn = document.getElementById('.btn');
btn.onclick = fun2;
//出错 后绑定的事件会覆盖掉之前的事件
```

删除事件处理程序 将对应事件属性置为null

btn.onclick = null;

标准事件模型

- 事件捕获：从document一直向下传播到目标元素， 依次检查经过节点是否绑定了事件监听函数，有则执行
- 事件处理：到达目标元素， 触发目标元素的监听函数
- 事件冒泡：从目标元素冒泡到document， 依次检查经过节点是否绑定了事件监听函数，如果有则执行

事件绑定监听函数

addEventListener(eventType, handler, useCapture)

事件移除监听函数

removeEventListener(eventType, handler, useCapture)

- eventType事件类型(不要加on)
- handler事件处理函数
- ***useCapture，是否在捕获阶段处理，默认false***

举个例子：

```
var btn = document.getElementById('.btn');
    btn.addEventListener('click', showMessage, false);
    btn.removeEventListener('click', showMessage, false);
```

一个DOM上绑定多个事件处理器，不会冲突

```
Plain Text
1  btn.addEventListener('click', showMessage1, false);
2  btn.addEventListener('click', showMessage2, false);
3  btn.addEventListener('click', showMessage3, false);
```

事件代理

原理

事件委托，把一个或者一组元素的事件委托到它的父层或者更外层元素上，真正绑定事件的是外层元素，不是目标元素

只指定一个事件处理程序，管理某一类型 所有事件

把一个元素响应事件（click、keydown……）的函数委托到另一个元素，冒泡阶段完成

对“事件处理程序过多”问题的解决方案就是事件委托

使用事件委托，只需在DOM树中尽量高的一层添加一个事件处理程序

举例

代 取快递

优点

- 节省内存，减少dom操作
- 不需要给子节点注销事件
- 动态绑定事件
- 提高性能
- **新添加的元素还会有之前的事件**

为啥用

事件冒泡过程中上传到父节点，父节点通过事件对象获取到目标节点，把子节点的监听函数定义在父节点上，由父节点的监听函数统一处理子元素的事件

比如100个li，每个都有click，如果使用for遍历 添加事件，关系页面整体性能，需要不断交互 访问dom次数过多，引起重排，延长交互时间

事件委托的话，将操作放进JS，只需要和dom交互一次，提高性能，还节约内存

第三个参数(useCapture)为true在捕获过程执行，反之在冒泡过程执行

数据类型（原始数据类型和引用数据类型，堆栈内存），数据类型检测和数据类型转换

数组常用方法、对象常用方法

1. 类数组和数组的区别和转换，数组的检测方式，数组去重和排序
2. 函数执行机制和执行上下文，执行上下文栈（普通函数，立即执行函数，箭头函数、构造函数）
3. 闭包作用域（变量提升，arguments, 作用域与作用域链，闭包的应用场景，闭包优缺点）
4. 内存泄漏和垃圾回收机制
5. this指向的规则（call/bind/apply的使用）
6. new和构造函数
7. 继承实现的方式和区别?
8. 深拷贝与浅拷贝

事件循环机制（任务队列，微任务，宏任务，同步，异步，进程和线程，浏览器线程）

宏任务主要包括：setTimeout、setInterval

微任务主要包括：promise、process.nextTick()

执行规则：同步代码直接进入主线程执行，JS引擎判断主线程是否为空，如果为空，则读取 微任务Event Queue 中所有的消息，并依次执行。主线程和微任务 Event Queue 都为空后，读取 宏任务Event Queue 中的第一个消息进入主线程执行，来回微宏。

9. JS异步解决方案（回调函数、Promise、Generator、async、定时器）
10. DOM选择器
11. 常见DOM操作（增，删，改）
12. 事件流的过程（事件冒泡和事件捕获）、事件处理程序（DOM0级事件处理程序和DOM2级事件处理程序）、事件对象、事件委托

真假值的隐式转换

数据类型检测

1、typeof　其中数组、对象、null都会被判断为object，其他判断都正确。

2、instanceof