ES6

- **fetch**
  - 对比ajax,axios
    - axios
      - axios是一个基于promise的Http库，在浏览器环境使用XHR，在node环境中使用http模块发送请求
      - 用Promise吧XHR封装起来
      - 原理
        - promise链 —— 可以链式调用的axios的原因，因为最终返回的是一个promise对象
        - chain数组 —— 请求拦截器和相应拦截器插入到chain数组中。
        - 取消请求的是一个异步分离的设计方案，利用promise的异步效果，通过切换promise的状态，从而达到异步取消请求的实现
        - 拦截器的原理就是通过chain的Promise链，在请求之前加入请求拦截器，请求结束之后加入响应拦截器，循环遍历chain，从而达到链式调用的作用。在dispatchRequest的前后，传递的参数由request变成response
      - 优势
        - 支持 Promise API
        - Axios的一大卖点就是它提供了拦截器，可以统一对请求或响应进行一些处理 —— 可以为请求附加token、为请求增加时间戳防止请求缓存，以及拦截响应，一旦状态码不符合预期则直接将响应消息通过弹框的形式展示在界面上，
        - 自动转换请求和响应数据
        - 取消请求

          ```
          const controller = new AbortController()

          axios.get('/foo/bar', {
             signal: controller.signal
          }).then(function(response) {
             //...
          });
          // 取消请求
          controller.abort()
          ```
        - 客户端支持防止CSRF/XSRF
        - axios既提供了并发的封装，也没有fetch的各种问题，而且体积也较小，当之无愧现在最应该选用的请求的方式。
        - 非200状态码会直接进catch —— https://axios-http.com/zh/docs/handling_errors
        - 设置超时 —— 超时处理

          拦截timeout错误

          在Axios响应拦截器中，处理timeout错误，代码如下：

          ```
          service.interceptors.response.use(function (response) {
              // 对响应数据做点什么
              return response;
          }, function (error) {
              // 对响应错误做点什么
              if(error.message.includes('timeout')){   // 判断请求异常信息中是否含有超时timeout字符串
                console.log("错误回调", error);
                alert("网络超时");
              }
              return Promise.reject(error);
          });
          ```
    - ajax
  - 特点
    - 无论请求成功与否，它都返回一个 Promise 对象
    - 请求数据放在body中，并需要手动将对象转为字符串
    - 如果想流畅的使用fetch,需要手动进行封装
    - 需要手动对响应数据进行转换

      ```
      fetch("http://example.com/")
        .then((response) => response.json()) // 需要对响应数据进行转换
        .then((data) => {
          console.log(data);
        })
        .catch((error) => console.error(error));
      ```
      - arrayBuffer() / blob() / json() / text() / formData() —— 使用Fetch时你需要清楚请求后的数据类型是什么，然后再用对应的方法将它进行转换。
  - 缺点
    - 无法取消请求
    - 上传进度无法获取
    - 无法捕获到400，500状态码错误
      - 对于非200状态码，不会进入catch,而是在resolve中的OK值为false
      - 仅当网络故障时或请求被阻止时，才会标记为 reject。
    - **不会发送跨域 cookies** —— 要发送需要加上*credentials*
    - 超时处理麻烦
    - 有兼容问题
  - 优势
    - 原生API
    - 更低层，可以自行按需求封装
    - 符合关注分离，没有将输入、输出和用事件来跟踪的状态混杂在一个对象里
- **Proxy**
  - 核心功能 —— 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、函数调用等）
  - 用法 —— var proxy = new Proxy(target, handler)
    - target表示所要拦截的目标对象（任何类型的对象，包括原生数组，函数，甚至另一个代理））
    - handler对象，key为可代理的方法名，value为可代理方法的函数 —— 可代理的方法包括
      - get(target,propKey,receiver) —— 拦截对象取值
      - set(target,propKey,value,receiver) —— 拦截对象设置值
      - has —— 拦截对象中是否有值，并返回布尔值
      - deleteProperty
      - ownKeys（target） —— 拦截Object.keys(proxy)、for...in等循环，返回一个数组
      - ....
  - 使用场景
- **Map**
  - 一种类似Object的新的数据结构，一种叫做字典的数据结构,Map可以理解为是Object的超集，打破了以传统键值对形式定义对象，对象的key不再局限于字符串，也可以是Object。可以更加全面的描述对象的属性。
  - Map 对象保存键值对。任何值(对象或者原始值) 都可以作为一个键或一个值
  - map和object的区别
    - key
      - Object遵循普通的字典规则，键必须是单一类型，并且只能是整数、字符串或是Symbol类型
      - Map中，key可以为任意数据类型（Object, Array等）
    - 元素顺序 —— Map会保留所有元素的顺序，而Object并不会保证属性的顺序
    - 元素数量 —— map的size可以查看元素数量，object没办法查看属性数量
    - 性能 —— Map 在涉及频繁增删键值对的场景下会有些性能优势。
  - 使用场景
    - object
      - key简单
      - value值是函数
      - 转为JSON
    - Map
      - 需要了解size的时候
      - 有频繁增删操作时候
      - 需要多种类型的key时
  - weakMap
    - WeakMap 只能将对象作为键名 —— 只接受对象作为键名（null 除外），不接受其他类型的值作为键名
    - WeakMap 的键名引用的对象是弱引用
      - 强引用

        ```
        const e1 = document.getElementById('foo');
        const e2 = document.getElementById('bar');
        const arr = [
          [e1, 'foo 元素'],
          [e2, 'bar 元素'],
        ];
        ```

        ```
        // 不需要 e1 和 e2 的时候
        // 必须手动删除引用
        arr [0] = null;
        arr [1] = null;
        ```

        在上面的代码中，e1和e2是两个对象，通过arr数组对这两个对象添加一些文字说明。但是这样就形成了arr对e1和e2的引用，而这种引用又是强引用。它的区别就体现在。当我们不再需要这两个对象时，我们必须手动的删除这个引用，解除arr都两个对象的引用关系，否则垃圾回收机制不会释放e1和e2占用的内存。因为，**arr仍然存在着对对象的引用！** 当忘记了手动删除引用，就会造成内存泄漏
      - 弱引用

        ```
        let obj = new WeakObject()
        ```

        我们就可以静静的等待垃圾车来把它拖走了，obj所引用的对象就会被回收
      - 弱引用关系，数据的引用计数为 0 ，程序垃圾回收机制在执行时会将引用对象回收。而如果时强引用关系则引用计数为 1 ，不会被垃圾回收机制清除。
    - 不可遍历 —— 正因为WeakMap对键名所引用的对象是弱引用关系，因此WeakMap内部成员是会却决于垃圾回收机制有没有执行，运行前后成员个数很可能是不一样的，而垃圾回收机制的执行又是不可预测的，因此不可遍历
    - 应用场景
      - DOM 节点元数据

        ```
        const wm = new WeakMap();
        const loginButton = document.querySelector('#login');
        // 给这个节点关联一些元数据
        wm.set(loginButton, {disabled: true});
        ```
      - 注册监听事件的 listener 对象
      - 部署私有属性 —— 利用弱映射，将内部属性设置为实例的弱引用对象，当实例删除时，私有属性也会随之消失，因此不会内存泄漏

        ```
        const _counter = new WeakMap();
        const _action = new WeakMap();

        class Countdown {
          constructor(counter, action) {
            _counter.set(this, counter);
            _action.set(this, action);
          }
          dec() {
            let counter = _counter.get(this);
            if (counter < 1) return;
            counter--;
            _counter.set(this, counter);
            if (counter === 0) {
              _action.get(this)();
            }
          }
        }

        const c = new Countdown(2, () => console.log('DONE'));

        c.dec()
        c.dec()
        ```
      - 数据缓存 —— 当我们需要在不修改原有对象的情况下储存某些属性等，而又不想管理这些数据时，可以使用WeakMap

        ```
        const cache = new WeakMap();
        function countOwnKeys(obj) {
            if (cache.has(obj)) {
                return cache.get(obj);
            } else {
                const count = Object.keys(obj).length;
                cache.set(obj, count);
                return count;
            }
        }
        ```
  - 参考资料 —— https://juejin.cn/post/6993101968545677319#heading-17
- **weakset和weakmap**
  - 及时清除引用非常重要。但是，你不可能记得那么多，有时候一疏忽就忘了，所以才有那么多内存泄漏。ES6 考虑到这点，推出了两种新的数据结构： weakset 和 weakmap 。
  - 他们对值的引用都是不计入垃圾回收机制的，也就是说，如果其他对象都不再引用该对象，那么垃圾回收机制会自动回收该对象所占用的内存。
  - 应用场景：注册监听事件的 listener 对象很适合用 WeakMap 来实现。

    ```
    // 代码1
    ele.addEventListener('click', handler, false)

    // 代码2
    const listener = new WeakMap()
    listener.set(ele, handler)
    ele.addEventListener('click', listener.get(ele), false)
    ```

    代码 2 比起代码 1 的好处是：由于监听函数是放在 WeakMap 里面，一旦 dom 对象 ele 消失，与它绑定的监听函数 handler 也会自动消失。
- **Set**
  - 无序且唯一
  - 类似Array的新的数据结构，Set是一种叫做集合的数据结构，类似于数组，但没有重复的值
  - Set 本身是一个构造函数，用来生成 Set 数据结构
  - Set的属性：size：返回集合所包含元素的数量，相当于数组的length
  - 操作方法
    - * add(value)：向集合添加一个新的项
    - * has(value)：如果值在集合中存在，返回true,否则false
    - * delete(value)：从集合中移除一个值
    - * clear(): 移除集合里所有的项
  - 遍历方法
    - * keys()：返回一个包含集合中所有键的遍历器

      ```
      let set = new Set(['a', 'b', 'c'])
      console.log('keys',set.values())
      console.log('values',set.values())
      keys ▸SetIterator {'a', 'b', 'c'}
      values ▸SetIterator {'a', 'b', 'c'}
      ```
    - * values()：返回一个包含集合中所有值的遍历器
    - * entries：返回一个包含集合中所有键值对的数组(感觉没什么用就不实现了)
    - * forEach()：用于对集合成员执行某种操作，没有返回值
  - Array.from 方法可以将 Set 结构转为数组

    ```
    const items = new Set([1, 2, 3, 2])
    const array = Array.from(items)
    console.log(array)    // [1, 2, 3]
    ```
  - 应用场景：
    - 交集、并集、差集

      ```
      let set1 = new Set([1, 2, 3])
      let set2 = new Set([4, 3, 2])

      let intersect = new Set([...set1].filter(value => set2.has(value)))
      let union = new Set([...set1, ...set2])
      let difference = new Set([...set1].filter(value => !set2.has(value)))

      console.log(intersect)    // Set {2, 3}
      console.log(union)        // Set {1, 2, 3, 4}
      console.log(difference)   // Set {1}
      ```
    - 数组去重 —— [...new Set([1, 2, 2, 3])]
  - 与Map的区别 —— 集合 是以 [value, value]的形式储存元素，字典 是以 [key, value] 的形式储存
  - set和weakSet区别
    - WeakSet 只能储存对象引用
    - weakSet无法遍历
    - WeakSet 对象中储存的对象值都是被弱引用的 —— 即垃圾回收机制不考虑 WeakSet 对该对象的应用，如果没有其他的变量或属性引用这个对象值，则这个对象将会被垃圾回收掉（不考虑该对象还存在于 WeakSet 中），所以，WeakSet 对象里有多少个成员元素，取决于垃圾回收机制有没有运行，运行前后成员个数可能不一致，遍历结束之后，有的成员可能取不到了（被垃圾回收了），WeakSet 对象是无法被遍历的（ES6 规定 WeakSet 不可遍历），也没有办法拿到它包含的所有元素
- **Class**
  - constructor
  - 继承 —— extends和super
  - static —— **static** 关键字定义静态方法。不能在类的实例上调用静态方法，而应该通过类本身调用
  - get和set
- **Symbol**
  - 属于原始值，而不是引用类型
  - 作为对象属性的唯一标识符
  - 对象中Symbol()属性不能被for...in遍历 —— 只能通过Object.getOwnPropertySymbols(含有symbol的对象)来读取
  - Symbol不能作为构造函数使用,就是说，不能用new
  - 应用场景：
    - Symbol作为key未被包含在对象自身的属性名集合(property names)之中。所以，利用该特性，我们可以把一些不需要对外操作和访问的属性使用Symbol来定义。
    - 定义类的私有属性和方法
  - 参考链接 —— https://juejin.cn/post/7028830157230047263
- **迭代器（Iterator）与生成器（Generator）**
  - Iterator
    - 所有的集合对象（数组、Set集合及Map集合）和字符串都是可迭代对象，这样就需要一种统一的接口机制，来处理所有不同的数据结构。Iterator就是这种机制.任何数据接口只要部署Iterator接口，**就可以使用for...of。** —— 也可以理解为 Iterator 接口主要为 for of 服务的，供for...of进行消费。
    - 迭代器是一个拥有next()方法的特殊对象，每次调用next()都返回一个结果对象,结果对象是:

      ```
      {
          value: item,
          done: true/false
      }
      ```
      - value是可迭代对象的元素，done表示是否可以继续迭代
      - 迭代到最后时，value:undefined,done:true
    - 其他
      - ES6规定，默认的Iterator的接口部署在数据结构的Symbol,iterator属性，或者说，一个数据接口只要部署了Symbol.iterator属性，**就可以使用for...of。**
      - 扩展运算符：只要数据结构部署了Iterator就可以使用扩张运算符
    - 方法
      - next()
      - return() —— 遍历器遍历过程中，提前退出，会调用遍历器对象的return()。（必须返回一个对象）
      - throw（） —— 配合Generator函数使用
    - fakeIterator

      ```
      function getIterator(list) {
          var i = 0;
          return {
              next: function() {
                  var done = (i >= list.length);
                  var value = !done ? list[i++] : undefined;
                  return {
                      done: done,
                      value: value
                  };
              }
          };
      }
      var it = getIterator(['a', 'b', 'c']);
      console.log(it.next());// {value: "a", done: false}
      console.log(it.next());// {value: "b", done: false}
      console.log(it.next());// {value: "c", done: false}
      console.log(it.next());// "{ value: undefined, done: true }"
      ```
    - 判断是否是可迭代对象

      ```
      function isIterable(object) {
          return typeof object[Symbol.iterator] === "function";
      }
      console.log(isIterable('abcdefg')); // true
      ```
    - Iterator对象作用：
      - 可以使用for...of...
      - 可以使用解构赋值
      - 可以使用**扩展运算符...**
  - Generator
    - 生成器是一种返回迭代器的函数
    - 特点：
      - * function关键字与函数名之间有一个星号；
      - * 函数体内部使用yield表达式，定义不同的内部状态
    - Generator 函数是分段执行的，调用next方法函数内部逻辑开始执行，遇到yield表达式停止，返回{value: yield后的表达式结果/undefined, done: false/true},再次调用next方法会从上一次停止时的yield处开始，直到最后。
    - 与普通函数的区别
      - 调用生成器函数后，函数并不立即执行，返回的不是函数运行结果，而是一个迭代器对象。
      - 通过迭代器对象next方法分段执行函数，返回yield后面表达式的值
    - 使用方式：

      ```
      function* foo(x) {
          yield x + 1;
          yield x + 2;
          return x + 3;
          yield x + 4;
      }
      ```

      ```
      const r = foo(0);
      r.next();// { value: 1, done: false }
      r.next();// { value: 2, done: false }
      r.next();// { value: 3, done: true }
      r.next();// { value: undefined, done: true } 被return终结了
      r.next();// { value: undefined, done: true }
      ```
    - 注意：使用for...of...

      ```
      function* foo () {
        yield 1;
        yield 2;
        return 3;
      }

      for (const v of foo()) {
        console.log(v);
      }
      1
      2
      ```

      for...of...不执行return后的语句
    - 异常处理：Generator函数返回的遍历器对象都有一个throw方法，可以在函数体外抛出错误，然后在Generator函数体内捕获。

      ```
      const g = function* () {
        try {
          yield;
        } catch(e){
          console.log('内部捕获', e);// 捕获1对象的第一个throw
        }
      };

      const i = g();
      i.next();

      try {
        i.throw('a');// 在函数体外抛出错误, 在函数体内捕获错误
        i.throw('b');// 在函数体外抛出错误, 在函数体外捕获错误
      } catch (e) {
        console.log('外部捕获', e);
      }

      // 结果
      // 内部捕获 a
      // 外部捕获 b
      ```
    - 应用场景：
      - 流控制
      - 在对象上部署Iterator接口

        ```
        function* iterEntries(obj) {
          let keys = Object.keys(obj);
          for (let i=0; i < keys.length; i++) {
            let key = keys[i];
            yield [key, obj[key]];
          }
        }

        let myObj = { foo: 3, bar: 7 };

        for (let [key, value] of iterEntries(myObj)) {
          console.log(key, value);
        }

        // foo 3
        // bar 7
        ```
      - 对文件的批量处理

        ```
        function* doStuff() {
          yield fs.readFile.bind(null, 'hello.txt');
          yield fs.readFile.bind(null, 'world.txt');
          yield fs.readFile.bind(null, 'and-such.txt');
        }

        for (task of doStuff()) {
          // task是一个函数，可以像回调函数那样使用它
        }
        ```
- **promise**
  - promise.all
    - 报错处理
      - 给每一个参数都在后面添加.catch来return error —— 这样参数抛的错误会走进promise.all的.then中
      - 或者在参数中吧错误resolve出来而不是reject出来
      - 使用Primise.allSettled代理promise.all
        - Primise.allSettled使用方法跟Primise.all一样，不过不同之处在于无论参数实例resolve 还是 reject， Promise.allSettled 都会执行 then 方法的第一个回调函数（意思就是不会 catch 到参数实例的 reject 状态）
        - 其回调函数的参数返回的数组的每一项是一个包含 status 和 value 或者 reason 的一组对象
    - 传参的时候，参数已经在执行了
- **async/await**
  - 与Promise的关系
    - 执行async函数，返回的时Promise对象（实际async await就是语法糖）
    - await相当于Promise的then
    - try...catch可以捕获异常，代替了Promise的catch
  - 写是同步语法，但本质还是异步调用
  - async 函数是 Generator 函数的语法糖 —— 内置执行器。Generator 函数的执行必须依靠执行器，而 Aysnc 函数自带执行器，调用方式跟普通函数的调用一样
  - **Async 函数的错误处理**

    ```
    let a;
    async function f() {
        await Promise.reject('error');
        a = await 1; // 这段 await 并没有执行
    }
    f().then(v => console.log(a));
    ```

    改为：

    ```
    // 正确的写法
    let a;
    async function correct() {
        try {
            await Promise.reject('error')
        } catch (error) {
            console.log(error);
        }
        a = await 1;
        return a;
    }

    correct().then(v => console.log(a)); // 1
    ```
- **ES6常用语法**
  - let,const
  - 箭头函数
  - 解构赋值
  - 扩展运算符
  - Promise
  - 数组：
    - map
    - reduce
    - filter
    - forEach
  - 字符串
    - 模板字符串：`${name}是我`
    - startWith

      ```
      let str='git://www.baidu.com/23448';
      if(str.startsWith('http://')){
                      alert('普通地址');
      }
      ```
  - 可选链 —— a?.b
- **严格模式**
  - 目的
    - 消除Javascript语法的一些不合理、不严谨之处，减少一些怪异行为
    - 提高编译器效率，增加运行速度
    - 为未来新版本的Javascript做好铺垫
  - "use strict"; —— 将"use strict"放在脚本文件的第一行，则整个脚本都将以"严格模式"运行。如果这行语句不在第一行，则无效，整个脚本以"正常模式"运行
  - 那些约束
    - 全局变量显式声明
    - 静态绑定
    - 增强的安全措施 —— 禁止this关键字指向全局对象
    - 对象不能有重名的属性
    - 函数不能有重名的参数
    - 不允许对arguments赋值
  - this
    - 在严格模式下，在全局作用域中，this指向window对象
    - 在严格模式下，这种函数中的this等于undefined

      ```
      function f1(){
        console.log(this);
      }
      ```