《深入理解 ES6》读书笔记，初入前端必学

前言

如果想知道ES6是什么，又必然会说一说历史，文字不多，简单介绍一下，我们现在常用的JavaScript是ECMAScript的实现和扩展，它由ECMA（注：可以理解为类似的W3C）组织参与进行标准化，并且ECMAScript定义了：

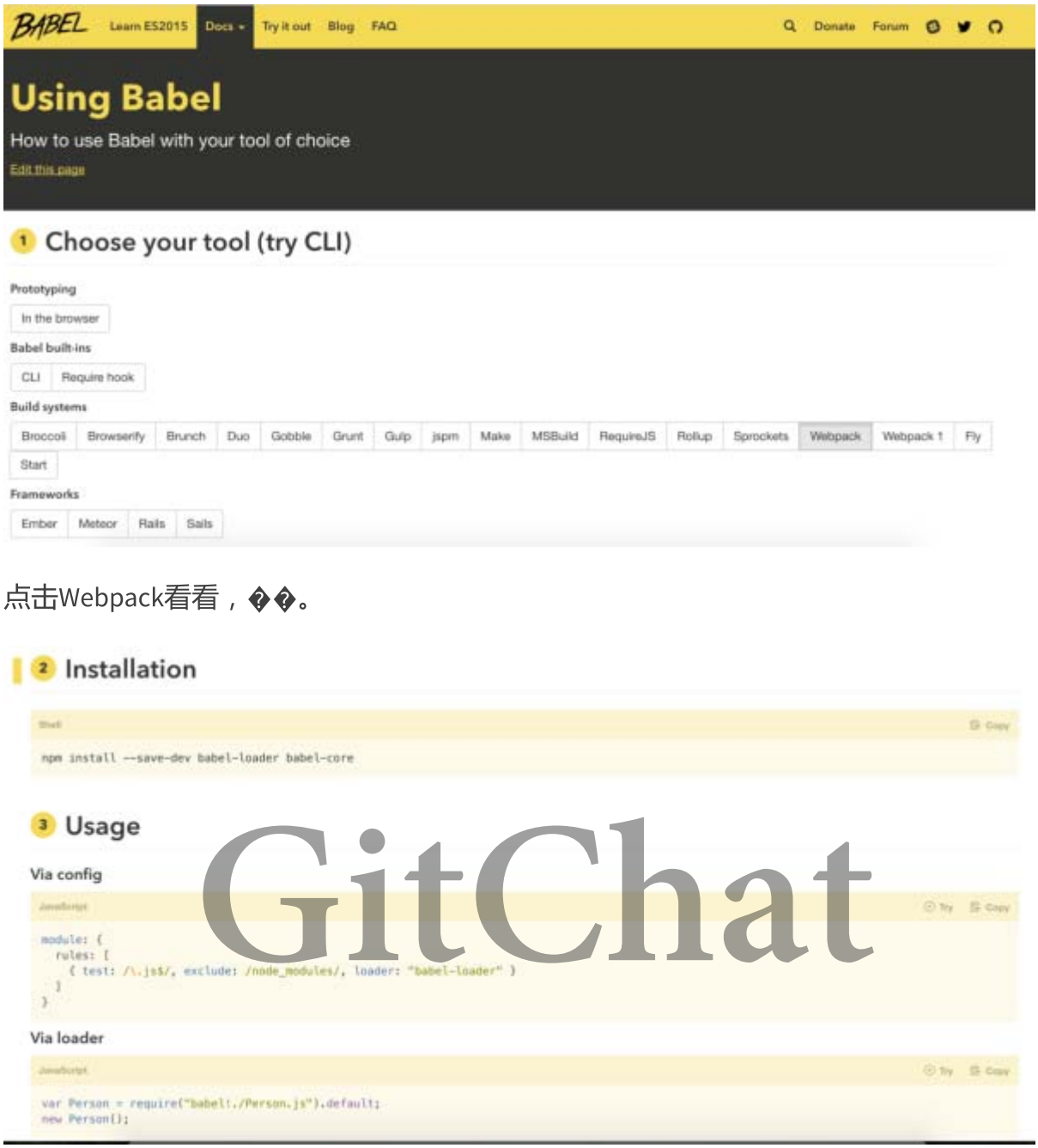
- 语言语法
- 类型
- 原型
- 内建对象和标准函数库

我们接下来说到的ES6已经通过了标准审核，并命名为**ES2015**（注：为了方便后续的文字全部用ES6来代表ES2015）。

这次的分享主要是读书笔记，原著是由社区知名的前端开发专家**Nicholas C. Zakas** 编写（注：同时也是《JavaScript高级程序设计》的作者），《深入理解ES6》本是一开源项目（https://github.com/nzakas/understandinges6）后出版，现中国大陆简体中文版可以在网路上购买。既然是阅读笔记，我想我并打算完全照搬书上的内容，因为那很无趣，反而这些读书笔记会贯穿我自己所积累的知识和理解。

由于本篇文章特　长，预期阅读需要30分钟之上。如果有不对的地方，也欢迎指出，可以在【https://zhuanlan.zhihu.com/fed-talk】留言并联系我。

搭建运行ES6的环境



点击Webpack看看，��。

试着通过从2-4步骤编写一个简单的例子，比如：

```
let name = 'icepy'  // -> index.js
```

```
const profile = { name: 'icepy', id: 'nices'}
console.log(...profile)
Experimental Presets
```

如果你仔细阅读过文档，你应该可以发现有Stage 0，Stage 1这样的字样，Stage 目前来说是按照JavaScript提案来区分的，数字越小，说明提案的时间与现在的时间越接近，这也意味着Stage 0 包含所有的 Stage，这些预设选项，你都可以通过配置来让其工作。

plugin

当然你也可以通过 plugin 找到对应的插件，如果你有兴趣，不妨阅读一下 https://github.com/babel/babel-preset-env/blob/master/data/plugin-features.js 这个文件。

```
{
  "plugins": [
    "transform-decorators-legacy",
    "transform-class-properties"
  ]
}
```


输出配置

当你对编译输出的代码有环境上的需求时，你还可以通过Options来开启一些额外的功能：

```
presets:['env',{ //Options }]
```

比如此刻我想设置一下我的代码需要支持的环境，你可以如此配置：

```
presets:['env',{
```

babel的东西不多，基本上如何配置你都可以在官网上找到答案，尝试着用一下吧。如果你对编写插件或preset有兴趣，你也可以阅读 ：

ES6块级作用域

这一小节让我想到很多年前阅读到Nicholas的那本红宝书，所讲的作用域以及Var。也许至今，还有一些人搞不明白变量提升的含义。如果你有幸在大学里写过C语言，其实这就很好理解了。一个我们所说明的变量其实包含声明，赋值两个部分，是否有看过在头文件里声明，或者在C函数体内先声明，比如：

```
int sayB(){
 int b;
 b = 1;
 return b;
}
sayB()
```

而JS中所提到的变量提升，与其非常类似，你在函数体内定义的变量，无论在哪里定义，都会提升到函数的顶部，比如：


```
function b(){
 console.log(v)  // ?会报错吗?
 if (true){
   var v = 1;
 }
}
```

// 其实不会，执行顺序会变成

```
function b(){
```

```
 if (true){
   let v = 1;
 }
}
```

这个时候再打印v是会报错的，不像之前那样给你一个undefined。

通过这一小节的阅读，这应该是其中一个很大的知识点，以及知晓了一些有趣的事情，这和 let const有关。let和const定义的变量不会像var一样，覆盖到全局。以前我们用var来定义变量时，如果多写一个重名的，只会是最后一个覆盖之前的。但是，这里如果你用let或者const来定义时，必然会报错。而且 const 定义的变量是不允许再赋值的，但是它允许对于键的再赋值，比如：

```
const b = {a:1}

// b.a = 2 (YES)
```

可以说对于以前我印象中最深刻的是如果从数组里可以正确的获取到其数值，需要借助闭包来完成，而现在因为块级作用域的存在，你完全可以不必要像ES5那样借助闭包了。

```
for (let i = 0;i<j;i++){
  //i
}
```


这个时候可以很顺利的完成从[0...n] 的过程。

很明显，块级作用域在某些时候帮助我们节约了很多事情，不会像因为变量提升而带来的某些不可预知的奇怪问题。最后这一小节给出了最佳实践，同名而言，如果你定义的是一个预知的值（不再修改）那么你应该使用const，反之你应该使用let，尽量的避免使用var，当然你想定义一个全局变量除外。

始而已。当然，如果你还是想要文本的位置，那么你可以继续使用indexOf或lastIndexOf。哦，对了字符串还提供了一个repeat方法，用于返回一个将当前字符串重复几次之后的字符串，看起来，这玩意有些鸡肋，但是你要是开发编辑器之类的应用，它倒是非常有用。

说说“模板语法”吧。

```
const name = 'icepy'
const message = `${name} call u`
console.log(message)
```

如果在以前我们想完成谁call你这样的动态消息，都需要如此：

```
var name = 'icepy'
var message = name + 'call u'
console.log(message)
```

粗看起来也没有什么不好，但是如果这个消息会很长很长，那么就要用+来拼接几行了。那时候看起来，就比较搓了。如果有模板语法之后，我们会怎样？


```
const name = 'icepy'
const phone = '186xxxxxx'
const message = `
${name} call u
Phone Number: ${phone}
```

会不会舒服很多？当然这个特性带给你的不仅仅是这些，很重要的一点是完全可以支持“运算”式的表达式，你可以在{}来运算一些结果。看到这里，是不是感觉到和一些模板库很像？幸运的是，从今之后再也不用引入一些基础模板库了。

在模板语法中我可能用的比较多的就是tag，有时候我会比较偷懒，将字符串的转换写成函数，然后不同类型的转换，直接用tag就好，比如：

ES6的函数有几个显著的特点，方便了传入参数的处理，又不失优雅。提供的箭头函数虽然有诸多限制，但是在表达上减少了代码量还解决了我们在ES5中遇到的 `this` 丢失的问题。比较乐观的说，这是我非常喜欢使用的ES6新特性之一。

在参数处理上，ES6给大家提供了默认参数和不定参数的选择，而默认参数在某个形式上对于解决默认值的问题，终于不用再去处理很多的判断了，想想JavaScript在“布尔”逻辑判断上的隐式转换，比如”0” == false，如果你的参数传入的是0，本意它就是一个真实存在的参数，当在逻辑判断中时，这个参数被隐式的转换了，从而获取了默认值。

```
function logName(name='icepy') {
  console.log(name)
}
logName() // icepy
logName('wower') // wower
```

在你细心的处理默认参数时，有一点需要注意：

```
function logName(name='icepy',Qname = name){
 // name
 // Qname
}
logName() //icepy
```


你可以将第二个参数的默认值设置成第一个参数，但是不能将第一个参数的默认值设置成第二个参数。

关于不定参数我想和Function一起说说，先来看一段Weex执行业务Code的代码：

```
/**
 * Call a new function body with some global objects.
 * @param  {object} globalObjects
 * @param  {string} code
 * @return {any}
```

这个巧妙的callFunction就包含了我们想看到的不定参数与Function构造函数，不定参数使用…来表示，它用一个数组包含了其中的参数，相信读到这里，你会很快的联想到ES5中的arguments，你可以说和它类似，又非常不同。

如果在函数定义时写了不定参数，那么这个函数就只能接受唯一的一个，如果函数定义时定义了多个参数呢？

```
function (name,...profile){
  // profile []
  // arguments
  // name
}
```

你可以通过这样的关系看到不定参数与arguments之间的关系，arguments永远包含所有的参数，这也意味着它包含了name，而profile并不包含name。回到上面的那段Weex核心代码，细心阅读一下，相信你会有收获。

1. body是bundle.js的业务代码，将它加入到globalKeys最后一个位置。
2. 定义一个拥有不定参数globalKeys的Function来生成一个函数
3. Function你可以知道，它接受参数和函数体，并且最后一个永远是函数体
4. 最后执行生成的函数，将globalValues当不定参数传入
   说起来…还可以用来处理其他的事情，来多实践一下吧。

说完参数，我们再来说说平时我们可能大量使用的 () => {} 箭头函数，它可能是我们非常乐见的“语法糖”之一，不过在此之前有一些要说明的是：

- 自身没有this,super等，而这些需要外层的非箭头函数来决定
- 不能实例化
- 没有原型
- 不能改变this，arguments等

如果你写过React + Redux，相信你一定对箭头函数使用的非常频繁，比如：

有时候我们在写React组件时，可能会用到立即调用模式：

```
... {
  render(){
    return (
      <div>
       (() => {
         return (<div>icepy</div>)
       })()
      </div>
    );
  }
}
```

不过，说真的，我常常配合着数组用，Why？表达的很简单：

```
values.sort((a,b) => a - b )
```

解构和扩展对象的功能性

如果你正在开发使用数据驱动的应用，那么这一小节的内容对你来讲，就非常重要了，欢迎来到解构的世界。

何为“解构”？当我们定义了很多对象和数组，又必须从中提取有价值的数据时，这种行为在ES6中被称为“解构”，解构分为两种：对象解构和数组解构。

如果你定义过这样的对象：

```
const obj = {
  name: 'icepy',
```

这个时候work会是一个undefined，但是我想给予他一份“新的工作”，骑摩托车旅行：

```
const obj = {
  name: 'icepy'
}
const { name, work = '骑摩托车旅行'} = obj
```

当然，如果可以，我们也可以重新定义一个名字，比如work变成travel：

```
const obj = {
  name: 'icepy'
}
const { name, work:travel = '骑摩托车旅行'} = obj
```

解构在函数的参数中也可以定义，当然传参时需要传入的是对象了。如果你写过一些Redux，那么相信对这个会无比的熟悉：

```
export const changeDate = ({ mode, date }) => (dispatch) => {
  dispatch(setTime({ mode, date }));
};

// 在action里可以直接使用mode和date
```


如果你有多重嵌套的对象，解构也可以很方便的提取出来你想要的数据：

```
const obj = {
  home: {
    ad: 'hunan 湘西',
    mz: 'm'
  },
  name: 'icepy'
}
```

当然如果你只想取第三个元素，只需要用,占位即可，如果你想从嵌套的数组中提取数据，其实和对象非常类似：

```
const arr = [1,2,3]

let [,,t] = arr;

const arr = [1,[2],3]

let [,[f],] = arr;

// f
```

曾经我们可能会有这样的一个需求，有一个数组，需要第五个元素之后的元素，返回一个新数组：

```
const arr = [1,2,3,4,5,6,7,8,9]
let [,,,,,...f] = arr;
//f
```

而现在，这样多简单。

当然在这一小节中，主要的内容都是这些实践操作性的内容，如果有时间，不妨试试。

在《红宝书》里是这样来说对象初始值的：

```
var name = 'icepy'
var age = 18
var obj = {
  name: name,
  age: age,
  say: function(){
    console.log(this.name)
```

```
    }
}
```

比之ES5，这拥有了极致的简化，而且使用say(){}定义的属性可以调用super。

这一小节里讲的内容，比较重要的是讲原型的增强，其他的比如对对象的增强，仅是添加了几个我们已经比较常用的方法了：

```
Object.is
Object.assign
Object.setPrototypeOf
```

可能某些情况下会出现这样的状况，A类是B类的父类，A类定义了say方法，B类也定义了say方法，在B类的parentSay中调用A类的say方法，实例化B类，直接调用parentSay方法，这个时候我们就很尴尬了，如果调用this.say，不一定能调用到A类定义的say方法。在ES5中给我们提供了一个getPrototypeOf方法来获取任意一个对象的原型，根据JS的继承机制我们可以知道如果调用Object.getPrototypeOf(this)正好就可以获取到父类的原型，然后再调用say方法就可以了。

但，这也有一个很尴尬的问题，于是，ES6给我们提供了super来代表父类，这个时候，我们直接使用super.say()就很方便的区别调用A类的say方法和B类的say方法了。

有趣的是，ES6还为我们提供了一个设置原型的方法“setPrototypeOf”，这个方法可以设置原型对象。

如果曾经你使用过类似jQuery的extend方法，那么就应该对assign会有相当熟悉了感觉，它和extend非常类型，可以将一对象中的属性，方法赋值给另一个对象。至于is，就是一个正确的布尔值比对，大部分情况下与 === 行为一致。

什么是Symbol以及怎么使用

我们知道在JavaScript的世界里，唯一性是很难被描述的，哪怕你用任意的加密函数生成的Key也有可能被撞库，Symbol为我们带来了这样的唯一性：

```
let name = Symbol('name');
let max = Symbol();
let obj = {
  [name]:'icepy',
  [max]: 29
}

// Symbol()
```

Symbol可以接收一个参数，这个参数用来des，主要用于程序调试时的跟踪，当然你也可以不传入参数，同样的我们可以通过typeof来判断是否为Symbol类型。

Symbol属性中提供了for方法来处理全局共享的问题，它可以从指定的Symbol注册表中来搜索到具体的内容，反之你可以用keyFor来推断某个Symbol关联的键。

通过Symbol的属性来操作JavaScript内部的逻辑

以前我们比较难去操作JavaScript本身语言内部的逻辑，最多也是在原型链上去定义或者修改某个方法的实现，Symbol的属性中提供了很多去处理程序内部执行的逻辑，如果你有兴趣，可以仔细阅读一下MDN上的文档：Symbol，这里主要举一个例子来说明：

在以前我们想判断一个数组，可能会这样写：

```
let f = []
Object.prototype.toString.call(f)
// "[object Array]"
```

我们可以定义”[object Array]”来判断，但是，假设今天我们写的某些应用程序中，我想将某些类定义出来类型，通过类型的判断，想解决一些问题，最开始我们可能可以用instance来判断是哪个实例，但，这解决不了问题。

但是，在之前有说到可以通过instance来判断是否为某个类或函数的实例，有一天我想给予它就算它是真实的实例，我也想让它返回false，即：不是一个实例。

```
function Pre(){

}
Object.defineProperty(Pre,Symbol.hasInstance,{
  value: function(){
     return false
  }
})
```

这就是Symbol的作用之一，来改变语言程序内部的逻辑。

也许，这些特性在你真实的应用中，很少会用到，不过，看一看，也是没有坏处的。

新的 Set 集合，Map 集合

曾经在JavaScript中就只有一种集合可用：数组，如果可以你将对象也称之为一种集合。对于老程序员来说，数组可能会勾起很多大家的回忆，毕竟我们用数组，对象实现了诸如去重复，缓存等等，而今天，ES6带来了Set集合和Map集合，让我们更方便的去操作这些。

- Set
- WeakSet
- Map
- WeakMap

关于API有兴趣的朋友，可以去翻一翻文档，特别简单。
在使用Set或者WeakSet之前，你应该需要了解一下Set或WeakSet的特点：

它是一个不可重复的无序列表

- WeakMap的键名只能是对象，与WeakSet类似，它也是一个弱引用
- Map比WeakMap多了几个方法，但是基础的set get 都是相同的

目前，已知的Map的用途，可能就是来存储私有数据，存储对象引用（可以自动释放），大部分情况下，你基本不会用到Map。

```
let privateData = new WeakMap()

class Pre{
  constructor(name){
    privateData.set(this,{ name })
  }
  preName(){
    return privateData.get(this).name
  }
}

export default Pre
```

这些对于内存管理倒是很方便了，不需要自己手动的去跟踪信息，再手动的去设置null来断开引用。

iterator 和 generator 的使用

这一小节的内容，比较鼓舞的是终于可以在JS语言层面，能看见Iterator和Generator了。说到迭代器，也许你会有疑问，可以预期的，你能看到Generator的实现也是依赖迭代器。我所接触到的编程语言中，最早让我理解这个特性的是Python。迭代器是一种特殊的对象，它的设计有专门的接口（描述）来完成我们常说的迭代（循环）过程。

每一个迭代器对象，都具备next方法（当然它也具备一些比如throw方法），当你执行next方法时，会返回一个（描述）对象，这个对象中，存在value，done属性，你想要的值就是value，而done则是用来描述整个迭代过程是否结束，可以想象，当迭代过程结束

如果你不想调用三下迭代器，你可以自己写一个小的循环，比如：

```
let res = iterator.next();
while(!res.done){
  res = iterator.next()
  console.log(res)
}
```

所谓的Generator也就是一种可以返回迭代器的函数，只不过它用*和yield来表示（描述过程）。

```
function * createIterator(){
   yield 1;
   yield 2;
}

let iterator = createIterator()

console.log(iterator.next())
console.log(iterator.next())
```


其实这很好理解，（语言）在背后帮我们对这个函数进行了包装，每一个yield都会返回一个迭代器对象，你想执行真正的逻辑，或者你想获取值，都需要通过这个迭代器对象来获取。

Generator可以辅助我们完成很多复杂的任务，而这些基础知识，又与iterator息息相关，举一个很简单的例子，相信有很多朋友，应该使用过co这个异步编程的库，它就是用Generator来实现，当然它的设计会比例子要复杂的多，我们先来看一个co简单的用法：

```
import co from 'co'
co(function* () {
```

```
  while(!resl.done){
    console.log(resl);
    resl = _task.next(resl.value);
  }
}

function sayName(){
  return {
    name: 'icepy'
  }
}

function assign *(f){
  console.log(f)
  let g = yield sayName()
  return Object.assign(g,{age:f});
}

co(function *(){
  let info = yield *assign(18)
  console.log(info)
})
```

虽然，这个例子中，还不能很好的看出来“异步”的场景，但是它很好的描述了Generator的使用方式。

从最开始的定义中，已经和大家说明了，Generator最终返回的依然是一个迭代器对象，有了这个迭代器对象，当你在处理某些场景时，你可以通过yield来控制，流程的走向。通过co函数，我们可以看出，先来执行next方法，然后通过一个while循环，来判断done是否为true，如果为true则代表整个迭代过程的结束，于是，这里就可以退出循环了。在Generator中的返回值，可以通过给next方法传递参数的方式来实现，也就是遇上第一个yield的返回值。

有逻辑，自然会存在错误，在Generator捕获错误的时机与执行throw方法的顺序有关系，一个小例子：

```
console.log(_it.next())
console.log(_it.throw(new Error('hu error')))
```

当我能捕获到错误的时机是允许完第二次的yield，这个时候就可以try了。

Iterator和generator给了我们很多启发，在编程的维度上，我能想到的就是去处理异步代码时为我们提供便捷的方式。当然迭代器，这个方向上，可以做的事情有很多，如果你悉心去寻找，相信，很快能找到答案。

怎么写类，以及 Promise 和如何封装处理异步

大部分面向对象的语言都支持类和类继承的特性
从ECMA1-ECMA5的版本都不支持类和类继承的特性，于是开发者们通过原型，构造函数等来模拟类和类继承特性，这里不在复述，如果你有兴趣的话，可以阅读一下红包书（JavaScript高级程序设计）中关于类，类继承这两章。ECMA6终于至少在语言层（依然是基于原型的语法糖）面看起来支持了类和类继承，理解类的基本原理有助于理解ES6关于类的设计。

基本的类声明


```
class Human{
  constructor(name){
    this.name = name;
  }

  sayName(){
    console.log(this.name)
  }
}

let man = new Human('icepy')
man.sayName()
```

```
    this.name = name;
  }
  sayName(){
    console.log(this.name)
  }
}

function fetchObj(HumanClass){
  return new HumanClass('icepy')
}

let man = fetchObj(Human)
```

但是匿名的类表达式，有一个不好的地方，就是在调试的时候很难定位，不过我们可以像函数一样给表达式加上一个name：

```
let Human = class Human{
 ...
}
```

高阶知识

我们可以为类中的属性创建访问器，就像使用Object.defineProperty给对象的属性创建访问器一样的含义。


```
class Human{
 constructor(name){
    this.name = name;
 }
 get name(){
    return this.manName;
 }

 set name(newValue){
```

```
class Human{
  constructor(name){
    this.name = name;
  }

  *sayName(){
     yield 1;
  }
}
```

你如果有兴趣的话，可以把sayName改造成可支持异步的。

关于“静态”也就是说不必实例化就可以调用的方法，类语法也支持了这个：

```
class Human{
  constructor(name){
    this.name = name;
  }

  static sayWork(name){
    return new Human(name)
  }
}

Human.sayWork()
```


有趣的是，如果是在同一个类中两个静态方法，其中一个方法想调用另外一个静态方法，这个时候，也可以使用this。

```
class Human{
  constructor(name){
    this.name = name
  }
```

```
    this.name = name;
  }
}

class Icepy extends Human{
  constructor(name){
    super(name)
  }

  sayName(){
    console.log(this.name)
  }
}
```

关于super使用的时机，在前几章中有谈到，类中如果有继承，并且指定了constructor，那么就必须在使用this之前先调用super来初始化this。这个继承不仅仅是原型上，也包括静态成员，而且就算父类与子类都有同样的方法名，也不怕被覆盖，可以用http://this.xxx和http://super.xxx来分别调用，在以往我们用原型模拟时，就非常难界定这个方法到底调用来自哪里。

如果你想知道类是否被实例化，也可以通过new.target来确定，在别的语言中有抽象类的概念，也就是只定义描述不搞实现，并且不能被实例化，只能被继承。这个时候，new.target就能排上用场了。


```
class Human{
 constructor(){
   if (new.target === Human){
     throw new Error('抽象类不可以使用new')
   }
 }
 sayName(){}
}
```

最后一个想说一下的是关于Symbol.species属性，这个属性用来返回函数的静态访问器

类属于ES6的新特性，它让我们可以更方便，安全的定义类，使用类，而不是像ES5一样，需要搞那么多复杂的东西来模拟这个特性。

在异步编程概念已经普及的今天，我们依然要谈一谈它，对于我们做前端代码的意义。当你通过Ajax请求数据使用回调函数来获取数据时，这就是一种异步编程。

```
$.ajax({
  url: 'xxx',
  success: function(data){
    // data
  }
})
```

考虑到JavaScript属于单线程的特点，异步对于这门语言就可见多么的重要。你可以试着想想，当我们没有异步时，我正准备提交一个信息，界面就卡住了，一直要等待信息返回结果才能有接下来的其他操作，这很痛苦。在JavaScript的世界里，异步是非常重要的一件事情。

Promise正是想来处理这样的异步编程，如果我们用Promise该如何处理这段Ajax？

```
function fetch(){
  return new Promise(function(resolve,reject){
    $.ajax({
      url: 'xxx',
      success:function(data){
        resolve(data)
      },
      error:function(error){
        reject(error)
      }
    })
  })
}
```

如同上面Ajax的例子，我们可以很好的包装一个函数，让fetch函数返回一个Promise对象。在Promise构造函数里，可以传入一个callback，并且在这里完成主体逻辑的编写。唯一需要注意的是：Promise对象只能通过resolve和reject函数来返回，在外部使用then或catch来获取。如果你直接抛出一个错误（throw new Error('error')），catch也是可以正确的捕获到的。

Promise其他的方法

Promise.all（当所有在可迭代参数中的 promises 已完成，或者第一个传递的 promise（指 reject）失败时，返回 promise。）

```
var p1 = Promise.resolve(3);
var p2 = 1337;
var p3 = new Promise((resolve, reject) => {
  setTimeout(resolve, 100, "foo");
});

Promise.all([p1, p2, p3]).then(values => {
  console.log(values); // [3, 1337, "foo"]
});
```

Promise.race（返回一个新的 promise，参数iterable中只要有一个promise对象"完成（resolve）"或"失败（reject）"，新的promise就会立刻"完成（resolve）"或者"失败（reject）"，并获得之前那个promise对象的返回值或者错误原因。）


```
var p1 = new Promise(function(resolve, reject) {
    setTimeout(resolve, 500, "one");
});
var p2 = new Promise(function(resolve, reject) {
    setTimeout(resolve, 100, "two");
});

Promise.race([p1, p2]).then(function(value) {
  console.log(value); // "two"
```

```
callback -> Promise -> generator -> async await
```

未来，肯定属于async await了。

代理和反射

这两个API在实际的业务场景中几乎非常少的使用，至于具体的场景可能还需要大家来挖掘了。看起来每一个API都有一些其他的API可模拟，可使用，它们具体的内涵就是在于拦截，执行，返回结果。

Proxy（代理）

从字面上来看，这应该很好理解，就像nginx一样，将请求代理。我们来看一个非常小的例子：

```
let handler = {
    get: function(target, name){
        return name in target ? target[name] : 37;
    }
};

let p = new Proxy({}, handler);

p.a = 1;
p.b = undefined;

console.log(p.a, p.b);
// 1, undefined

console.log('c' in p, p.c);
```

```
let obj = {a:1}

console.log(Reflect.get(obj,'a'))
```

设置一个属性：

```
let obj = {};
Reflect.set(obj, "prop", "value"); // true
obj.prop; // "value"
```

对函数进行调用：

```
Reflect.apply(Math.floor, undefined, [1.75]);
```

这和函数的apply唯一不同的是第一个参数必须是要调用的函数，后两个参数与函数的apply保持一致。

更多的方法，有兴趣的话可以阅读：Reflect

说实话，这两个API的应用场景，其实我也有些糊涂，可能要经常使用，比对，才能发现比之对应有相同功能的方法的区别。

使用模块封装代码

不知不觉《深入理解ES6》阅读笔记就写到了最后一篇，完结之后可能会开启另外的一个系列，分享自己的知识点，让阅读到的人有一点点的收获，以及自己的成长。最后一篇主要是来写一写用模块封装代码的事情，回顾历史，从最早的立即执行函数，再到require.js，以及commonjs，今天我们面对的是语言标准给我们带来的模块化方案。如果说模块，我们应该可以从字面的意思上来看，这是一种可以自动运行在严格模式下并且没有办法退出的代码块。这种代码块在作用域上可以避免互相之间的污染，以及更好的

```
export function a(){
}
```

当然这样的方式，也可以导出类，对象，变量等。

有了导出，自然会有导入：

```
// 假设a.js

export function a(){}

// b.js使用a.js

import { a } from 'a.js'

a()
```

这样，也就构成了ES6的模块系统。

除此之外，整个模块系统给了我们很多其他的方式来操作，比如导出一个默认的函数：


```
// a.js
export default function a(){}

// b.js

import a from 'a.js'
```

如果可以，我想给a起一个别名：

```
// a.js
```

在浏览器中不借助webpack这样的工具也可以使用模块系统，只需要将type=module，这个模块需要注意的，可能就是关于路径的问题了，其他都和export import一样。