理解JavaScript语言

> JavaScript 模块通过运行时、文法、执行过程去剖析JS的知识体系

语言通识

产生式 BNF

> 什么是BNF:Backus Normal Form 又成为**巴科斯-诺尔范式**,是一种用于表示上下文无关文法的语言，上下文无关文法描述了一类形式语言。
> 尽管巴科斯范式也能表示一部分自然语言的语法，它还是更广泛地使用于程序设计语言、指令集、通信协议的语法表示中。大多数程序设计语言或者形式语义方面的教科书都采用巴科斯范式

BNF 规定是推导规则(产生式)的集合，写为：
<符号> ::= <使用符号的表达式>
这里的 <符号> 是非终结符，而表达式由一个符号序列，或用指示选择的竖杠 '|' 分隔的多个符号序列构成，每个符号序列整体都是左端的符号的一种可能的替代。从未在左端出现的符号叫做终结符。

```
语法结构分成基础结构和需要用其他语法结构定义的复合结构:基础结构称为终结符、
复合结构称非终结符.
引号和中间的字符表示终结符
在双引号中的字 "word" 代表着这些字符本身。而double_quote用来代表双引号;
在双引号外的字（有可能有下划线）代表着语法部分;
尖括号 < > 内包含的为必选项;
方括号 [ ] 内包含的为可选项;
大括号 { } 内包含的为可重复0至无数次的项;
圆括号 ( ) 内包含的所有项为一组，用来控制表达式的优先级;
竖线 | 表示在其左右两边任选一项，相当于"OR"的意思;
```

```
::= 是“被定义为”的意思；
...  表示术语符号；
斜体字: 参数，在其它地方有解释；
```

BNF 实现四则运算

```
1 + 2 * 3
终结符:
Number
+ - * /
非终结符:
MultiplicativeExpression 乘法结构
AddtiveExpression 加法结构

BNF:
<MultiplicativeExpression>::=<Number> |
    <MultiplicativeExpression> "*"<Number> |
    <MultiplicativeExpression> "/"<Number> |
<AddtiveExpression>::=<MultiplicativeExpression>|
    <AddtiveExpression>"+"<MultiplicativeExpression> |
    <AddtiveExpression>"-"<MultiplicativeExpression> |
```

```
BracketsExpression: 定义为左括号结构
BracketsRightExpression:定义为右括号结构
<MultiplicativeExpression>::=<Number> |
[ <BracketsExpression>::="(" ] <MultiplicativeExpression> "*"<Numb
er>[<BracketsRightExpression>::=")"] |
[<BracketsExpression>]<MultiplicativeExpression> "/"<Number>[<Brac
ketsRightExpression>] |
<AddtiveExpression>::=<MultiplicativeExpression>|
[<BracketsExpression>]<AddtiveExpression>"+"<MultiplicativeExpress
ion> [<BracketsRightExpression>] |
[<BracketsExpression>]<AddtiveExpression>"-"<MultiplicativeExpress
ion> [<BracketsRightExpression>] |
例如:(1+2)*3的BNF范式:
<BracketsExpression><AddtiveExpression>"+"<AddtiveExpression><Brac
```

```
    ketsRightExpression> "*" <MultiplicativeExpression>
```

编程语言的性质

动态与静态:

动态:

- 在用户设备或服务器上
- 产品实际运行
- Runtime

静态:

- 在程序员设备上
- 产品开发时
- Compiletime

类型系统:

动态类型系统与静态类型系统
强类型与弱类型
复合类型:结构体 函数签名
子类型
泛型: 逆变/协变

学习的方式:语法 **语义 运行时,重点学习语义、运行时**

Number类型

JavaScript最小的结构是什么? – Atom 原子

Atom

Grammar

- Literal
- Variable
- Keywords
- Whitespace
- Line Terminator

Runtime(运行时)

- Types
- Execution Context

JavaScript的7种类型

- Number
- String

- Boolean
- Object
- Null(有值但是空,typeof 值是Object)
- Undefined(没有定义值)
- Symbol(一定程度代替了String的作用,用于Object属性名的特殊的基本类型)

> Number 叫做Double Float 双精度浮点类型,Float表示小数点可以随意浮动的. 双精度浮点数组成:1个符号位+11个指数位+52个精度位

Grammar:

- DecimalLiteral: 0 0. .2 1e3
- BinaryIntegerLiteral:ob111
- OctalIntegerLiteral:0o10
- HexIntegerLiteral:0xFF

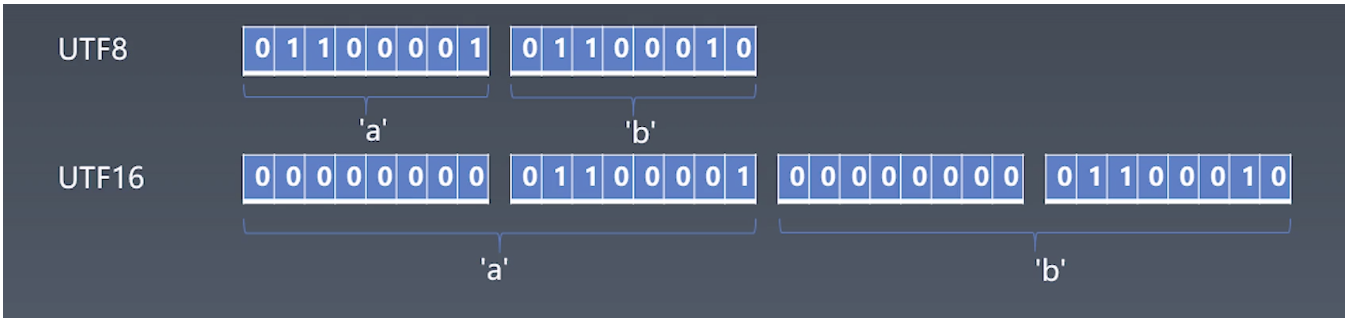
```
0.toString();//出现错误应该改为 0 .toString() 由于0. 可以表示小数,
//0后面跟一个空格即可.
```

String

- Character(字符)
- Code Point(数字代表字符)
- Encoding(编码方式)

Code Point:
ASCII 只能表示127个字符并不能表示所有的字符
Unicode 所有的字符联合的编码集
UCS:0000-FFFF 范围的字符集
GB:与unicode字符集不一致(GB2312 GBK GB18030)
ISO-8859
BIG5

Encoding:
UTF: UTF8 UTF16



String的语法(Grammar)

- "abc"
- 'abc'
- `abc`(字符串模版 `ab${x}abc${y}abc`)

Boolean 类型:true/false(都是关键字)

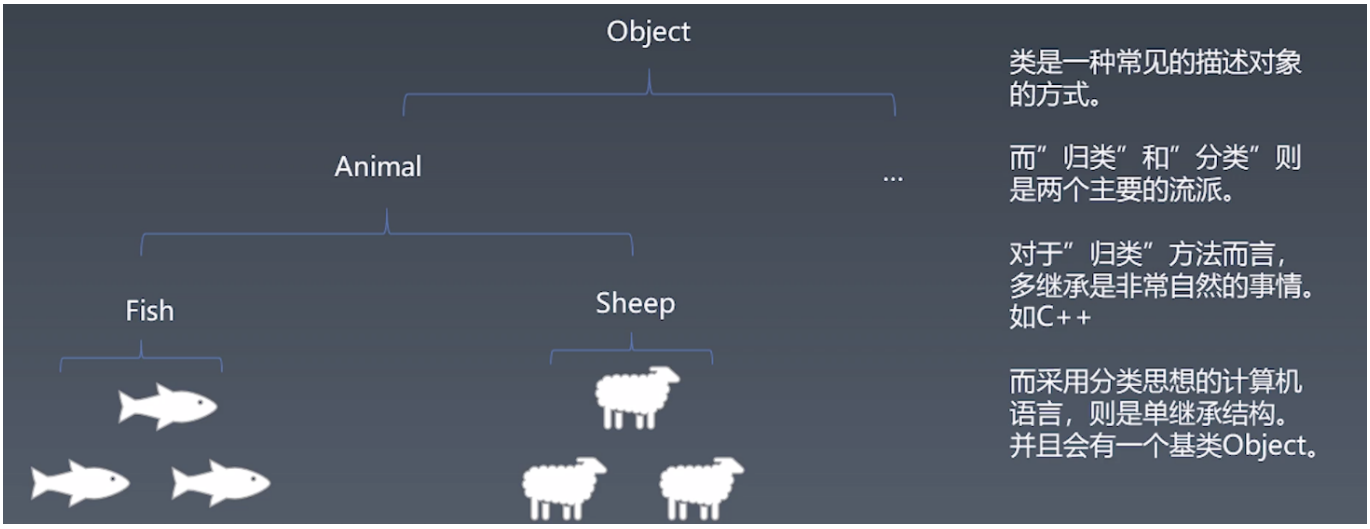
Null & Undefined
null是关键字
undefined是一个全局的变量,也就说undefined可以给他赋值 但是这样就会发生错误. 一般使用void 0来表示undefined

```
function f(){
    var undefined = 1;
  console.log(undefined);//1
}
function f(){
    var null = 0;
  console.log(null);//null是关键字不能使用报错
}
```

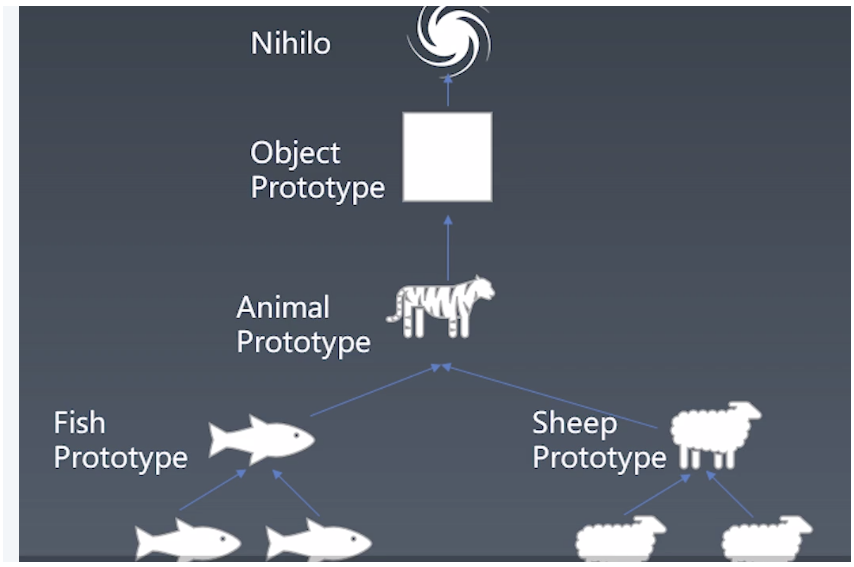
对象的基础知识

> 任何一个对象都是唯一的,这与它本身的状态无关,所以即使状态完全一致的两个对象,也并不相等. 用状态来描述对象,状态的改变即是行为.

类是一种常见的描述对象的方式



Object-Prototype:原型是一种更接近人类原始认知的描述对象的方法,任何对象仅仅需要描述它自己与原型的区别即可.



如下代码: 狗咬人的行为使用面向对象的形式表现

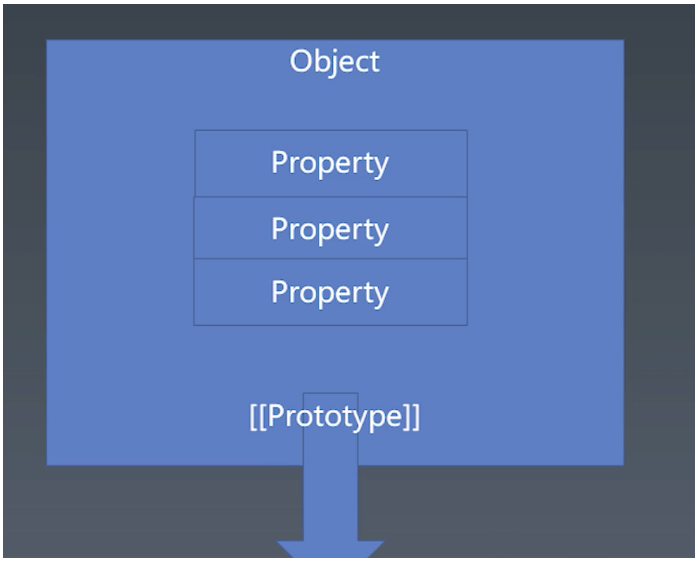
```
class Animal {
        constructor(name) {
            this.name = name;
        }
        bite(animal) {
            console.log(this.name + " 咬 " + animal.name);
        }
    }
    class Dog extends Animal {
        constructor(name) {
```

```
            super(name);
        }
    }

    class Person extends Animal {
        constructor(name) {
            super(name);
        }
        hurt(animal){
             console.log(this.name + " 咬 " + animal.name);
        }
    }

    let dog = new Dog("狗");
    let person = new Person("人");
    person.hurt(person);
```

> 在JavaScript 运行时,原生对象的描述方式非常简单,只需要关心原型和属性两个部分.使用内存地址来表示面向对象的唯一性



属性:是一个kv对,key可以是两种类型:Symbol、String.Value可以是:Data、Accessor
数据属性和访问属性

当我们访问属性时,如果当前对象没有,则会沿着原型找原型对象是否有此名称的属性,而原型对象还可能有原型,因此会有“原型链”这一说法.

语法Grammar:

- {} . [] Object.defineProperty 访问属性定义属性值等
- Object.create/Object.setPrototypeOf/Object.getPrototypeOf 基于原型的对象API
- new /class/extends 基于分类的方式
- new /function / prototype

Function Object 特殊的对象
Array 数组对象也是特殊的对象
Host Object
Object
[[call]]
[[construct]]