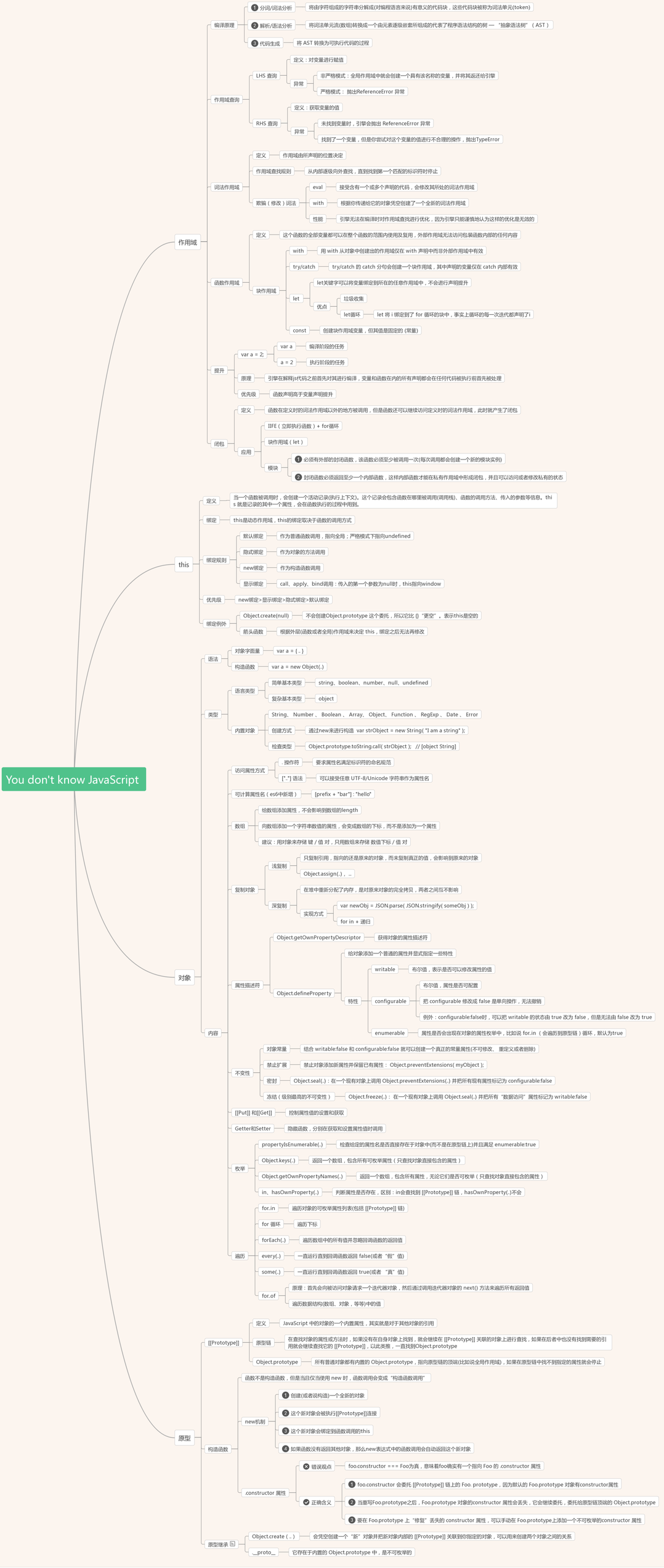
You don't know JavaScript
作用域
编译原理
1 分词/词法分析
将由字符组成的字符串分解成(对编程语言来说)有意义的代码块，这些代码块被称为词法单元(token)
2 解析/语法分析
将词法单元流(数组)转换成一个由元素逐级嵌套所组成的代表了程序语法结构的树 — “抽象语法树” （ AST ）
3 代码生成
将 AST 转换为可执行代码的过程
作用域查询
LHS 查询
定义：对变量进行赋值
异常
非严格模式：全局作用域中就会创建一个具有该名称的变量，并将其返还给引擎
严格模式： 抛出ReferenceError 异常
RHS 查询
定义：获取变量的值
异常
未找到变量时，引擎会抛出 ReferenceError 异常
找到了一个变量，但是你尝试对这个变量的值进行不合理的操作，抛出TypeError
词法作用域
定义
作用域由所声明的位置决定
作用域查找规则
从内部逐级向外查找，直到找到第一个匹配的标识符时停止
欺骗（ 修改 ）词法
eval
接受含有一个或多个声明的代码，会修改其所处的词法作用域
with
根据你传递给它的对象凭空创建了一个全新的词法作用域
性能
引擎无法在编译时对作用域查找进行优化，因为引擎只能谨慎地认为这样的优化是无效的
函数作用域
定义
这个函数的全部变量都可以在整个函数的范围内使用及复用，外部作用域无法访问包装函数内部的任何内容
块作用域
with
用 with 从对象中创建出的作用域仅在 with 声明中而非外部作用域中有效
try/catch
try/catch 的 catch 分句会创建一个块作用域，其中声明的变量仅在 catch 内部有效
let
let关键字可以将变量绑定到所在的任意作用域中，不会进行声明提升
优点
垃圾收集
let循环
let 将 i 绑定到了 for 循环的块中，事实上循环的每一次迭代都声明了i
const
创建块作用域变量，但其值是固定的 (常量)
提升
var a = 2;
var a
编译阶段的任务
a = 2
执行阶段的任务
原理
引擎在解释js代码之前首先对其进行编译，变量和函数在内的所有声明都会在任何代码被执行前首先被处理
优先级
函数声明高于变量声明提升
闭包
定义
函数在定义时的词法作用域以外的地方被调用，但是函数还可以继续访问定义时的词法作用域，此时就产生了闭包
应用
IIFE（ 立即执行函数 ）+ for循环
块作用域（ let ）
模块
1 必须有外部的封闭函数，该函数必须至少被调用一次(每次调用都会创建一个新的模块实例)
2 封闭函数必须返回至少一个内部函数，这样内部函数才能在私有作用域中形成闭包，并且可以访问或者修改私有的状态
this
定义
当一个函数被调用时，会创建一个活动记录(执行上下文)。这个记录会包含函数在哪里被调用(调用栈)、函数的调用方法、传入的参数等信息。this 就是记录的其中一个属性，会在函数执行的过程中用到。
绑定
this是动态作用域，this的绑定取决于函数的调用方式
绑定规则
默认绑定
作为普通函数调用，指向全局；严格模式下指向undefined
隐式绑定
作为对象的方法调用
new绑定
作为构造函数调用
显示绑定
call、apply、bind调用：传入的第一个参数为null时，this指向window
优先级
new绑定>显示绑定>隐式绑定>默认绑定
绑定例外
Object.create(null)
不会创建Object.prototype 这个委托，所以它比 {} “更空” ，表示this是空的
箭头函数
根据外层(函数或者全局)作用域来决定 this，绑定之后无法再修改
对象
语法
对象字面量
var a = { .. }
构造函数
var a = new Object(..)
类型
语言类型
简单基本类型
string、boolean、number、null、undefined
复杂基本类型
object
内置对象
String、 Number 、 Boolean 、 Array、 Object、 Function 、 RegExp 、 Date 、 Error
创建方式
通过new来进行构造 var strObject = new String("I am a string");
检查类型
Object.prototype.toString.call(strObject); // [object String]
内容
访问属性方式
. 操作符
要求属性名满足标识符的命名规范
["."] 语法
可以接受任意 UTF-8/Unicode 字符串作为属性名
可计算属性名（es6中新增） [prefix + "bar"] : "hello"
数组
给数组添加属性，不会影响到数组的length
向数组添加一个字符串数值的属性，会变成数组的下标，而不是添加为一个属性
建议：用对象来存储 键 / 值 对，只用数组来存储 数值下标 / 值 对
复制对象
浅复制
只复制引用，指向的还是原来的对象，而未复制真正的值，会影响到原来的对象
Object.assign(..) ，..
深复制
在堆中重新分配了内存，是对原来对象的完全拷贝，两者之间互不影响
实现方式
var newObj = JSON.parse(JSON.stringify(someObj));
for in + 递归
属性描述符
Object.getOwnPropertyDescriptor
获得对象的属性描述符
Object.defineProperty
给对象添加一个普通的属性并显式指定一些特性
特性
writable
布尔值，表示是否可以修改属性的值
configurable
布尔值，属性是否可配置
把 configurable 修改成 false 是单向操作，无法撤销
例外：configurable:false时，可以把 writable 的状态由 true 改为 false，但是无法由 false 改为 true
enumerable
属性是否会出现在对象的属性枚举中，比如说 for..in （ 会遍历到原型链 ）循环，默认为true
不变性
对象常量
结合 writable:false 和 configurable:false 就可以创建一个真正的常量属性(不可修改、 重定义或者删除)
禁止扩展
禁止对象添加新属性并保留已有属性： Object.preventExtensions(myObject);
密封
Object.seal(..)：在一个现有对象上调用 Object.preventExtensions(..) 并把所有现有属性标记为 configurable:false
冻结（ 极别瞄准的不可变性 ）
Object.freeze(..)： 在一个现有对象上调用 Object.seal(..) 并把所有 “数据访问” 属性标记为 writable:false
[[Put]] 和[[Get]]
控制属性值的设置和获取
Getter和Setter
隐藏函数，分别在获取和设置属性值时调用
枚举
propertyIsEnumerable(..)
检查给定的属性名是否直接存在于对象中(而不是在原型链上)并且满足 enumerable:true
Object.keys(..)
返回一个数组，包含所有可枚举属性（ 只查找对象直接包含的属性 ）
Object.getOwnPropertyNames(..)
返回一个数组，包含所有属性，无论它们是否可枚举（ 只查找对象直接包含的属性 ）
in、hasOwnProperty(..)
判断属性是否存在，区别：in会查找到 [[Prototype]] 链，hasOwnProperty(..)不会
遍历
for..in
遍历对象的可枚举属性列表(包括 [[Prototype]] 链)
for 循环
遍历下标
forEach(..)
遍历数组中的所有值并忽略回调函数的返回值
every(..)
一直运行直到回调函数返回 false(或者 “假” 值)
some(..)
一直运行直到回调函数返回 true(或者 “真” 值)
for..of
原理：首先会向被访问对象请求一个迭代器对象，然后通过调用迭代器对象的 next() 方法来遍历所有返回值
遍历数据结构(数组、对象，等等)中的值
原型
[[Prototype]]
定义
JavaScript 中的对象的一个内置属性，其实就是对于其他对象的引用
原型链
在查找对象的属性或方法时，如果没有在自身对象上找到，就会继续在 [[Prototype]] 关联的对象上进行查找，如果在后者中也没有找到需要的引用就会继续查找它的 [[Prototype]]，以此类推，一直找到Object.prototype
Object.prototype
所有普通对象都有内置的 Object.prototype，指向原型链的顶端(比如说全局作用域)，如果在原型链中找不到指定的属性就会停止
构造函数
函数不是构造函数，但是当且仅当使用 new 时，函数调用会变成 “构造函数调用”
new机制
1 创建(或者说构造)一个全新的对象
2 这个新对象会被执行[[Prototype]]连接
3 这个新对象会绑定到函数调用的this
4 如果函数没有返回其他对象，那么new表达式中的函数调用会自动返回这个新对象
.constructor 属性
错误观点
foo.constructor === Foo为真，意味着foo确实有一个指向 Foo 的 .constructor 属性
正确含义
1 foo.constructor 会委托 [[Prototype]] 链上的 Foo. prototype，因为默认的 Foo.prototype 对象有constructor属性
2 当重写Foo.prototype之后，Foo.prototype 对象的constructor 属性会丢失，它会继续委托，委托给原型链顶端的 Object.prototype
3 要在 Foo.prototype 上 “修复” 丢失的 constructor 属性，可以手动在 Foo.prototype上添加一个不可枚举的constructor 属性
原型继承
Object.create（..）
会凭空创建一个 “新” 对象并把新对象内部的 [[Prototype]] 关联到你指定的对象，可以用来创建两个对象之间的关系
__proto__
它存在于内置的 Object.prototype 中，是不可枚举的