js高级

# 1. 数据分类(2大类)

1.基本(值)类型

\* Number: 任意数值

\* String: 任意文本

\* Boolean: true/false

\* undefined: undefined

\* null: null

2.对象(引用)类型

\* Object: 任意对象

\* Array: 特别的对象类型(下标/内部数据有序)

\* Function: 特别的对象类型(可执行)

# 2. 判断

1. typeof: 得到数据类型的字符串名称

\* 可以区别: 数值, 字符串, 布尔值, undefined, function

\* 不能区别: null与对象, 一般对象与数组

2.instanceof: A instanceof B ==> 判断A是否是B这个类型的实例对象

\* 专门用来判断对象数据的具体类型: Object, Array与Function

3. ===

\* 可以判断: undefined和null

### 数据类型的分类和判断

1). 基本(值)类型

Number ----- 任意数值 -------- typeof

String ----- 任意字符串 ------ typeof

Boolean ---- true/false ----- typeof

undefined --- undefined ----- typeof/===

null -------- null ---------- ===

## 2). 对象(引用)类型

Object ----- typeof/instanceof

Array ------ instanceof

Function ---- typeof/instanceof

# 3. undefined与null的区别?

1.\* undefined代表没有赋值

\* null代表赋值了, 只是值为null

2. 什么时候给变量赋值为null呢?

\* var a = null //a将指向一个对象, 但对象此时还没有确定

\* a = null //让a指向的对象成为垃圾对象

3. 严格区别变量类型与数据类型?

\* js的变量本身是没有类型的, 变量的类型实际上是变量内存中数据的类型

\* 变量类型:

\* 基本类型: 保存基本类型数据的变量

\* 引用类型: 保存对象地址值的变量

\* 数据对象

\* 基本类型

\* 对象类型

. 什么是数据?

\* 存储于内存中代表特定信息的'东东', 本质就是0101二进制

\* 具有可读和可传递的基本特性

\* 万物(一切)皆数据, 函数也是数据

\* 程序中所有操作的目标: 数据

\* 算术运算

\* 逻辑运算

\* 赋值

\* 调用函数传参

...

4.1. 什么是内存?

\* 内存条通电后产生的存储空间(临时的)

\* 产生和死亡: 内存条(集成电路板)==>通电==>产生一定容量的存储空间==>存储各种数据==>断电==>内存全部消失

\* 内存的空间是临时的, 而硬盘的空间是持久的

\* 一块内存包含2个数据

\* 内部存储的数据(一般数据/地址数据)

\* 内存地址值数据

\* 内存分类

\* 栈: 全局变量, 局部变量 (空间较小)

\* 堆: 对象 (空间较大)

2. 什么是变量?

\* 值可以变化的量, 由变量名与变量值组成

\* 一个变量对应一块小内存, 变量名用来查找到内存, 变量值就是内存中保存的内容

3. 内存,数据, 变量三者之间的关系

\* 内存是一个容器, 用来存储程序运行需要操作的数据

\* 变量是内存的标识, 我们通过变量名找到对应的内存, 进而操作(读/写)内存中的数据

# 5.问题: var a = xxx, a内存中到底保存的是什么?

\* xxx是一个基本数据

\* xxx是一个对象

\* xxx是一个变量

# 6.关于引用变量赋值问题

1). 2个引用变量指向同一个对象, 通过一个引用变量修改对象内部数据, 另一个引用变量也看得见

2). 2个引用变量指向同一个对象,让一个引用变量指向另一个对象, 另一个引用变量还是指向原来的对象

# 7.问题: 在js调用函数时传递变量参数时, 是值传递还是引用传递

只有值传递, 没有引用传递, 传递的都是变量的值, 只是这个值可能是基本数据, 也可能是地址(引用)数据

# 8.问题: JS引擎如何管理内存?

1. 内存生命周期

1). 分配需要的内存: 定义变量 / 创建对象

2). 使用(存/取)分配到的内存: 赋值 / 创建对象

3). 不需要时将其释放/归还

2. 释放内存

\* 为执行函数分配的栈空间内存: 函数执行完自动释放

\* 存储对象的堆空间内存: 当内存没有引用指向时, 对象成为垃圾对象, 垃圾回收器后面就会回收释放此内存

# 9.

|  |
| --- |
| /\*执行全局代码\*/ |
|  | // 定义一个变量a, 分配对应的内存 定义变量 |
|  | var a |
|  | // 将数据保存到a内存中 赋值 |
|  | a = 3 |
|  | /\* |
|  | {}: 在堆空间分配对应内存, 将对象数据保存到内存中 |
|  | =: 读取对象内存的地址值数据保存到a内存中 |
|  | \*/ |
|  | a = {} |
|  |  |
|  | // 不需要时将其释放/归还 |
|  | a = null |
|  |  |
|  | /\* |
| 执行函数代码 |  |
|  | 调用函数, 就会为其分配对应的栈空间, 来存储其局部变量 |
|  | 函数执行完, 自动释放函数执行分配的空间, 局部变量内存数据没有了 |
|  | 局部变量指向的堆空间对象成为了垃圾对象, 后面就会被回收释放 |
|  | \* \*/ |

10.1. 什么是对象?

\* 代表现实中的某个事物, 是该事物在编程中的抽象

\* 多个数据的集合体(封装体)

\* 用于保存多个数据的容器

2. 为什么要用对象?

\* 便于对多个数据进行统一管理

3. 如何创建对象?

\* 后面专门讲解

4. 对象的组成

\* 属性

\* 代表现实事物的状态数据

\* 由属性名和属性值组成

\* 属性名都是字符串类型, 属性值是任意类型

\* 方法

\* 代表现实事物的行为数据

\* 是特别的属性==>属性值是函数

5. 如何访问对象内部数据?

\* .属性名: 编码简单, 但有时不能用

\* ['属性名']: 编码麻烦, 但通用

# 11.问题: 什么时候必须使用['属性名']的方式?

\* 属性名不是合法的标识名

\* 属性名不确定

12.1. 什么是函数?

\* 具有特定功能的n条语句的封装体

\* 只有函数是可执行的, 其它类型的数据是不可执行的

\* 函数也是对象

2. 为什么要用函数?

\* 提高代码复用

\* 便于阅读和交流

3. 如何定义函数?

\* 函数声明

\* 表达式

## 4.1. 函数的2种角色

\* 函数: 通过()使用

\* 对象: 通过.使用 ==> 称之为: 函数对象

## 2.函数的2种不同角色

函数

一般函数 : 直接调用

构造函数 : 通过new调用

方法: 通过对象调用

对象

通过.调用内部的属性/方法

5.// 函数声明

function fn1() {

}

// 表达式

var fn2 = function () {

}

6. 如何调用(执行)函数?

\* test() 直接调用 this是window

\* new test() new调用 this是新创建的对象

\* obj.test() 对象调用 this就是那个obj

\* test.call/apply(obj) 通过call调用, this就是obj

13.1. 什么函数才是回调函数?

\* 你定义的

\* 你没有直接调用

\* 但最终它执行了(在特定条件或时刻)

2. 常见的回调函数?

\* DOM事件函数

\* 定时器函数

\* ajax回调函数(后面学)

\* 生命周期回调函数(后面学)

1. IIFE

1. 理解

\* 全称: Immediately-Invoked Function Expression 立即调用函数表达式

\* 别名: 匿名函数自调用

2. 作用

\* 隐藏内部实现

\* 不污染外部命名空间

15.1. this是什么?

\* 一个关键字, 一个内置的引用变量

\* 在函数中都可以直接使用this

\* this代表调用函数的当前对象

\* 在定义函数时, this还没有确定, 只有在执行时才动态确定(绑定)的

### 2. 如何确定this的值?

函数中的this

直接调用: xxx() : window

new调用: new xxx() : 新创建的对象

对象调用: obj.xxx() : obj

call/apply调用: xxx.call(obj) : obj

总结: 函数的调用方式决定了this是谁

3.前置知识:

\* 本质上任何函数在执行时都是通过某个对象调用的

16.问题:

在js调用函数时传递变量参数时, 是值传递还是引用传递？

只有值传递, 没有引用传递, 传递的都是变量的值, 只是这个值可能是基本数据, 也可能是地址(引用)数据

# 17.问题: JS引擎如何管理内存?

1. 内存生命周期

1). 分配需要的内存: 定义变量 / 创建对象

2). 使用(存/取)分配到的内存: 赋值 / 创建对象

3). 不需要时将其释放/归还

2. 释放内存

\* 为执行函数分配的栈空间内存: 函数执行完自动释放

\* 存储对象的堆空间内存: 当内存没有引用指向时, 对象成为垃圾对象, 垃圾回收器后面就会回收释放此内存

# 18. git的6个基本操作

1). 创建本地仓库

创建.gitignore并配置忽略

git init

git add \*

git commit -m "xxx"

2). 创建远程仓库

New Repository

指定名称

创建

3). 将本地仓库推送到远程仓库

git remote add origin https://github.com/zxfjd3g/xxx.git 关联远程仓库

git push origin master

4). 如果本地有更新, 推送到远程

git add \*

git commit -m "xxx"

git push origin master

5). 如果远程有更新, 拉取到本地

git pull origin master

6). 克隆远程仓库到本地

git clone <https://github.com/zxfjd3g/xxx.git>

# 19.原型对象

1. 函数的prototype属性(图)

\* 每个函数都有一个prototype属性, 它默认指向一个Object空对象(即称为: 原型对象)

\* 原型对象中有一个属性constructor, 它指向函数对象

2. 给原型对象添加属性(一般都是方法)

\* 作用: 函数的所有实例对象自动拥有原型中的属性(方法)

# 显式原型与隐式原型

1. 每个函数function都有一个prototype，即显式原型

2. 每个实例对象都有一个\_\_proto\_\_，可称为隐式原型

3. 对象的隐式原型的值为其对应构造函数的显式原型的值

4. 内存结构(图)

5. 总结:

\* 函数的prototype属性: 在定义函数时自动添加的, 默认值是一个空Object对象

\* 对象的\_\_proto\_\_属性: 创建对象时自动添加的, 默认值为构造函数的prototype属性值

\* 程序员能直接操作显式原型, 但不能直接操作隐式原型(ES6之前)；

# 21.1. 原型链(图解)

\* 访问一个对象的属性时，

\* 先在自身属性中查找，找到返回

\* 如果没有, 再沿着\_\_proto\_\_这条链向上查找, 找到返回

\* 如果最终没找到, 返回undefined

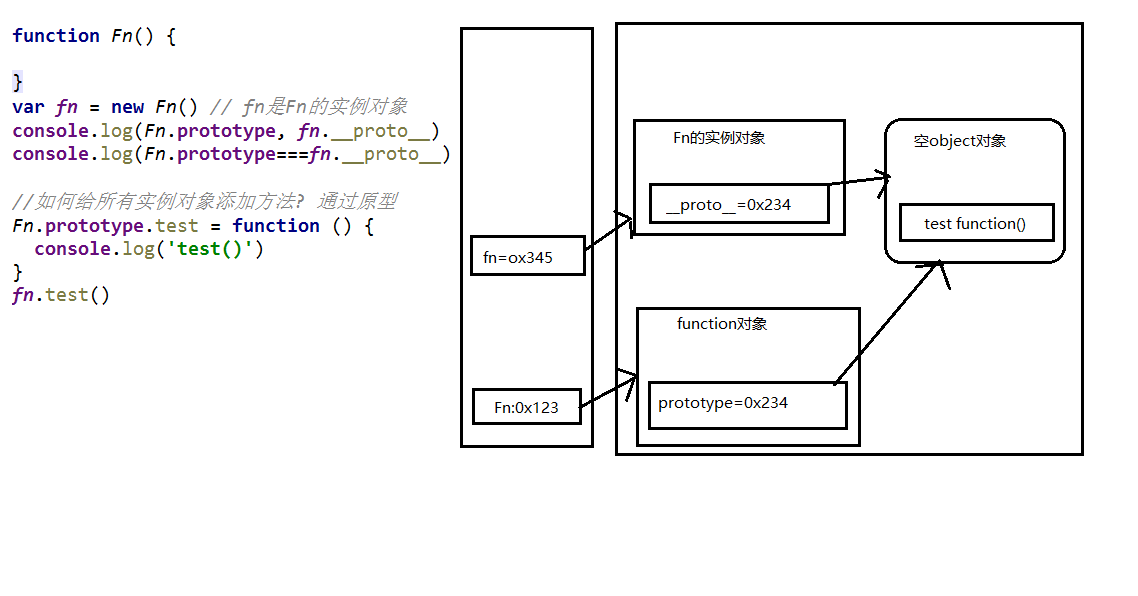
\* 别名: 隐式原型链

\* 作用: 查找对象的属性(方法)

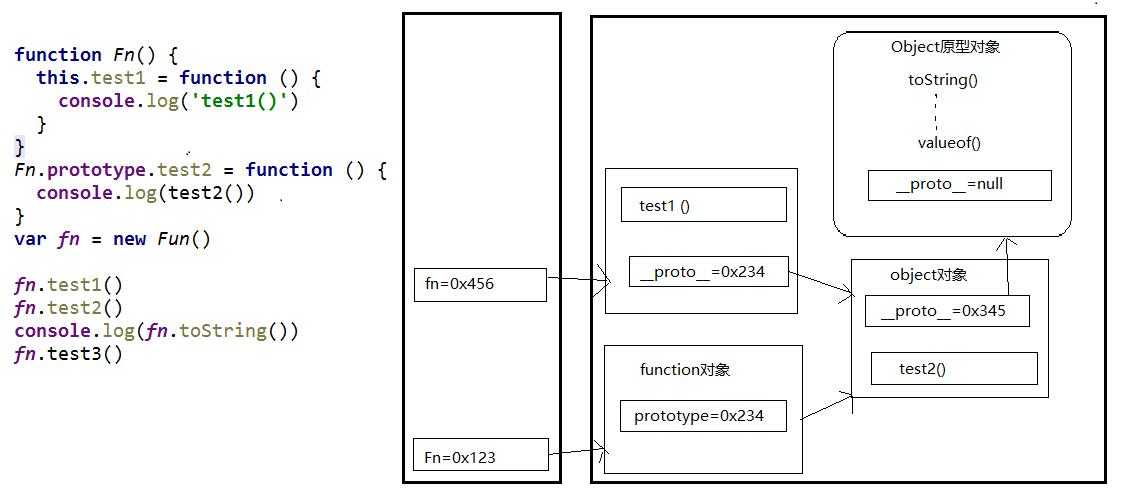
2. 构造函数/原型/实例对象的关系(图解)

3. 构造函数/原型/实例对象的关系2(图解)

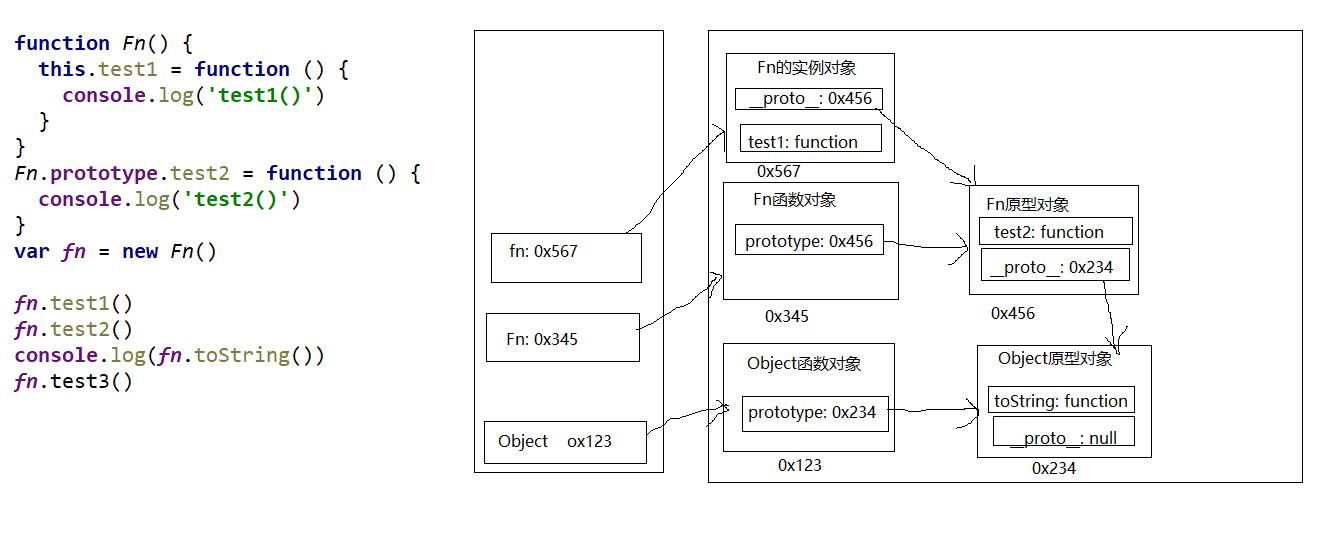
## 显式原型与隐式原型



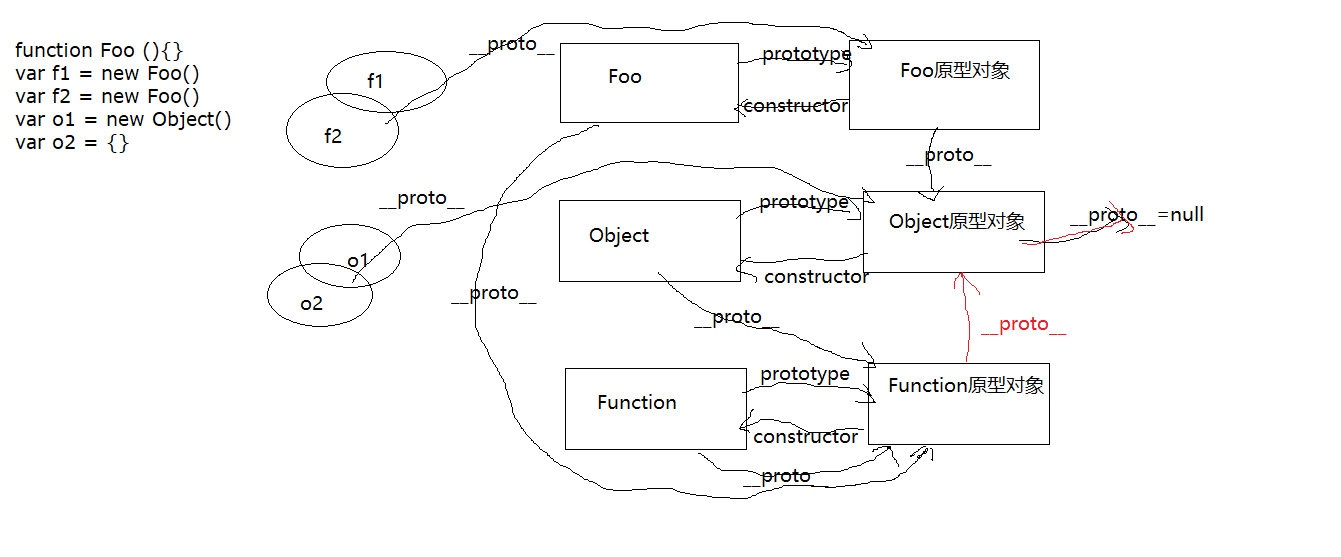
## 原型链分析



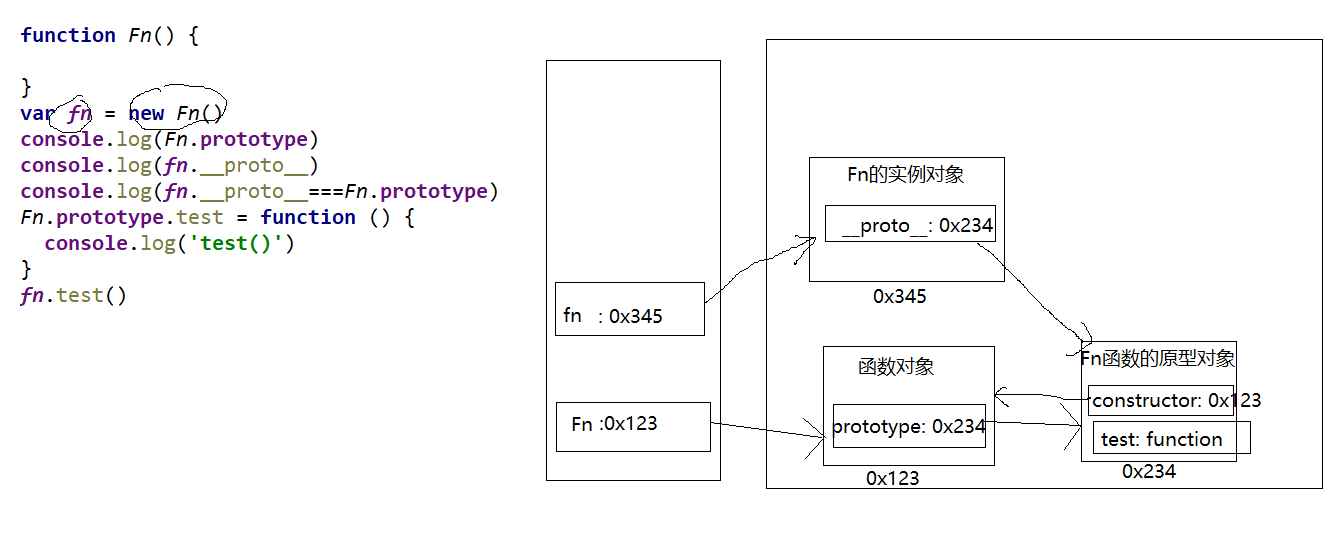
### 原型链



### 原型&原型链详图



### 显示原型与隐式原型



# 22.原型链\_属性问题

1. 读取对象的属性值时: 会自动到原型链中查找

2. 设置对象的属性值时: 不会查找原型链, 如果当前对象中没有此属性, 直接添加此属性并设置其值

3. 方法一般定义在原型中, 属性一般通过构造函数定义在对象本身上

# 23.探索instanceof

1. instanceof是如何判断的?

\* 表达式: A instanceof B

\* 如果B函数的显式原型对象在A对象的原型链上, 返回true, 否则返回false

2. Function是通过new自己产生的实例

# 24.变量提升与函数提升

1. 变量声明提升

\* 通过var定义(声明)的变量, 在定义语句之前就可以访问到

\* 值: undefined

2. 函数声明提升

\* 通过function声明的函数, 在之前就可以直接调用

\* 值: 函数定义(对象)

### 问题: 变量提升和函数提升是如何产生的?

变量提升: 将变量声明语句提前执行 变量声明提升；

函数提升: 将函数声明(定义)语句提前执行 函数声明提升

# 执行上下文

1. 代码分类(位置)

\* 全局代码

\* 函数代码

2. 全局执行上下文

\* 在执行全局代码前将window确定为全局执行上下文

\* 对全局数据进行预处理

\* var定义的全局变量==>undefined, 添加为window的属性

\* function声明的全局函数==>赋值(fun), 添加为window的方法

\* this==>赋值(window)

\* 开始执行全局代码

3. 函数执行上下文

\* 在调用函数, 准备执行函数体之前, 创建对应的函数执行上下文对象

\* 对局部数据进行预处理

\* 形参变量==>赋值(实参)==>添加为执行上下文的属性

\* arguments==>赋值(实参列表), 添加为执行上下文的属性

\* var定义的局部变量==>undefined, 添加为执行上下文的属性

\* function声明的函数 ==>赋值(fun), 添加为执行上下文的方法

\* this==>赋值(调用函数的对象)

\* 开始执行函数体代码

# 执行上下文栈

1. 在全局代码执行前, JS引擎就会创建一个栈来存储管理所有的执行上下文对象

2. 在全局执行上下文(window)确定后, 将其添加到栈中(压栈)

3. 在函数执行上下文创建后, 将其添加到栈中(压栈)

4. 在当前函数执行完后,将栈顶的对象移除(出栈)

5. 当所有的代码执行完后, 栈中只剩下window

### 6.先预处理函数, 后预处理变量, 如果已经存在就会被忽略

# 27.作用域

1. 理解

\* 就是一块"地盘", 一个代码段所在的区域

\* 它是静态的(相对于上下文对象), 在编写代码时就确定了

2. 分类

\* 全局作用域

\* 函数作用域

\* 没有块作用域(ES6有了)

3. 作用

\* 隔离变量，不同作用域下同名变量不会有冲突

# 28作用域与执行上下文

1. 区别1

\* 全局作用域之外，每个函数都会创建自己的作用域，作用域在函数定义时就已经确定了。而不是在函数调用时

\* 全局执行上下文环境是在全局作用域确定之后, js代码马上执行之前创建

\* 函数执行上下文环境是在调用函数时, 函数体代码执行之前创建

2. 区别2

\* 作用域是静态的, 只要函数定义好了就一直存在, 且不会再变化

\* 上下文环境是动态的, 调用函数时创建, 函数调用结束时上下文环境就会被释放

3. 联系

\* 上下文环境(对象)是从属于所在的作用域

\* 全局上下文环境==>全局作用域

\* 函数上下文环境==>对应的函数使用域

### 4.问题:

1. 有几个作用域? 1 + n(定义了几个函数)

2. 产生过几个上下文环境对象? 1 + n(执行了几次函数)

# 作用域链

1. 理解

\* 由内向外的多个作用域形成的链

\* 查找变量时就是沿着作用域链来查找的

2. 查找一个变量的查找规则

\* 在当前作用域下的执行上下文中查找对应的属性, 如果有直接返回, 否则进入2

\* 在上一级作用域的执行上下文中查找对应的属性, 如果有直接返回, 否则进入3

\* 再次执行2的相同操作, 直到全局作用域, 如果还找不到就抛出找不到的异常

性能优化

## **1.1** 性能优化是什么？

\* 优化网站性能，提高用户体验

2. 为什么要性能优化？

\* 从用户角度而言，优化能够让页面加载的更快、对用户的操作响应的更及时，能够给用户提供更为友好的体验。

\* 从服务商角度而言，优化能够减少页面请求数、减轻服务器压力，能够节省可观资源。

# 2. 浏览器应该有的功能

1.网络：

浏览器通过网络模块来下载各式各样的资源，例如html文本；javascript代码；样式表；图片；音视频文件等。

网络部分本质上十分重要，因为它耗时长，而且需要安全访问互联网上的资源。

2.资源管理：

从网络下载，或者本地获取到的资源需要有高效的机制来管理它们。

例如如何避免重复下载，资源如何缓存等

3.网页浏览：

这是浏览器的核心也是最基本的功能，最重要的功能。

如何将资源转变为可视化的结果。

多页面管理，插件与管理，账户和同步，安全机制，开发者工具

...

...

## 浏览器的主要功能总结起来就是一句话:

## 将用户输入的url转变成可视化的图像

1.从url到dom树

2.从dom树到可视化图像

这两个过程之间的关系并没有那么明确,我们可以统称这两个过程为页面的渲染

3.浏览器的内核(渲染引擎)

1.在浏览器中有一个最重要的模块，它主要的作用是将页面转变为可视化的图像结果。这个模块就是浏览器内核，通常它也被称为渲染引擎。

## 1.IE----->Trident

2.Safari------>WebKit

WebKit本身主要是由两个引擎构成的，一个正是渲染引擎“WebCore”，另一个则是javascript解释引擎“JSCore”，

它们均是从KDE的渲染引擎KHTML及javascript解释引擎KJS衍生而来。

3.Chrome----->WebKit的分支引擎----->Blink

在13年发布的Chrome 28.0.1469.0版本开始，Chrome放弃Chromium引擎转，而使用最新的Blink引擎（基于WebKit2——苹果公司于2010年推出的新的WebKit引擎），

Blink对比上一代的引擎精简了代码、改善了DOM框架，也提升了安全性。

4.Opera

旧版Opera 4至6版本 :Elektra排版引擎

Opera7.0 :Presto渲染引擎

Opera在2013年2月宣布放弃Presto:

采用Chromium引擎;

又转为Blink引擎;

5.Firefox------>Gecko

# 4.进程 线程

1.进程: 程序的一次执行, 它占有一片独有的内存空间.是操作系统执行的基本单元。

一个进程中至少有一个运行的线程: 主线程, 进程启动后自动创建;

一个进程中也可以同时运行多个线程, 我们会说程序是多线程运行的;

一个进程内的数据可以供其中的多个线程直接共享，多个进程之间的数据是不能直接共享的;

2.线程：是进程内的一个独立执行单元,是CPU调度的最小单元。程序运行的基本单元;

线程池(thread pool): 保存多个线程对象的容器, 实现线程对象的反复利用;

3.JS引擎是单线程运行的！

5.现代浏览器的

1.不堪回首的过去:

当你通过浏览器打开很多页面的时候,如果其中一个页面不响应了或者崩溃了,那么随之而来的将会是更不幸的事情,你开打的所有页面都会得不到响应,最让人不能忍受的是,其中的一些页面可能还包含了未保存或者未发送的信息.

2.浏览器产商如何解决

采用多进程模型,该模型可以带来的好处

①.避免因单个页面的不响应或者崩溃影响整个浏览器的稳定性;

②.当第三方插件崩溃时,也不会影响整个浏览器的稳定性;

③.安全;

3.浏览器到底有些什么进程

①.Browser进程:

浏览器的主进程,负责浏览器界面的显示,和各个页面的管理,浏览器中所有其他类型进程的祖先,负责其他进程的的创建和销毁;

它有且只有一个!!!!!

②.Renderer进程:

网页渲染进程,负责页面的渲染,可以有多个

当然渲染进程的数量不一定等于你开打网页的个数

③.VUE插件进程

④.React插件进程

⑤.GPU进程

移动设备的浏览器可能不太一样:

Android不支持插件,所以就没有插件进程

GPU演化成了Browser进程的一个线程

Renderer进程演化成了操作系统的一个服务进程,它仍然是独立的;

4.每个进程内部又有很多线程

多线程的目的主要是保持用户界面的高度响应

例如:为了不让Browser进程的UI线程被其他耗时的操作(数据库读写,本地文件读写)所阻塞,那么我们就把这些操作放到分线程中去处理在Renderer进程中,为了不让其他操作阻止渲染线程的高速执行,我们通常会将渲染过程管线化,

利用计算机的多核优势,让渲染的不同阶段在不同的线程中执行.

# 6.浏览器渲染引擎

## 1.主要模块

\* 一个渲染引擎主要包括：HTML解析器，CSS解析器，javascript引擎，布局layout模块，绘图模块

\* HTML解析器：解释HTML文档的解析器，主要作用是将HTML文本解释成DOM树。

\* CSS解析器：级联样式表的解析器，它的作用是为DOM中的各个元素对象计算出样式信息，为布局提供基础设施

\* Javascript引擎：使用Javascript代码可以修改网页的内容，也能修改css的信息，javascript引擎能够解释javascript代码，并通过DOM接口和CSSOM接口来修改网页内容和样式信息，从而改变渲染的结果。

\* 布局（layout）：在DOM创建之后，Webkit需要将其中的元素对象同样式信息结合起来，计算他们的大小位置等布局信息，形成一个能表达这所有信息的内部表示模型

\* 绘图模块（paint）：使用图形库将布局计算后的各个网页的节点绘制成图像结果

>备注：文档对象模型（Document Object Model，简称DOM）

## 2. 渲染过程

\* 浏览器渲染页面的整个过程：浏览器会从上到下解析文档。

1. 遇见 HTML 标记，调用HTML解析器解析为对应的 token （一个token就是一个标签文本的序列化）并构建 DOM 树（就是一块内存，保存着tokens，建立它们之间的关系）。

2. 遇见 style/link 标记 调用css解析器 处理 CSS 标记并构建 CSSOM 树。

3. 遇见 script 标记 调用 javascript解析器 处理script标记，绑定事件、修改DOM树/CSSOM树 等

4. 将 DOM 与 CSSOM 合并成一个渲染树。

5. 根据渲染树来布局，以计算每个节点的几何信息。

6. 将各个节点绘制到屏幕上。

>以上这些模块依赖很多其他的基础模块，包括要使用到网络 存储 2D/3D图像 音频视频解码器 和 图片解码器。

>所以渲染引擎中还会包括如何使用这些依赖模块的部分。

# 7.阻塞渲染

## 1.\* css阻塞

\* css在head中通过link引入会阻塞页面的渲染(推荐的方式)

为什么？======避免闪屏现象;

2\* css会阻塞js的执行，不阻塞js等其他资源的加载

为什么？

脚本在文档解析阶段会请求样式信息。如果当时还没有加载和解析样式，脚本就会获得错误的回复，这样显然会产生很多问题。这看上去是一个非典型案例， 但事实上非常普遍。Firefox 在样式表加载和解析的过程中，会禁止所有脚本。 而对于 WebKit 而言，仅当脚本尝试访问的样式属性可能受尚未加载的样式表影响时，它才会禁止该脚本。

## 3. \* js阻塞

### \* 直接引入的js会阻塞页面的渲染

### 为什么？

\*Javascript代码可能会修改DOM树/CSSOM树的结构

### \* js顺序执行，阻塞后续js逻辑的执行，不阻塞js等其他资源的加载

为什么？ 维护依赖关系

## 4.\* 预解析

1. WebKit 和 Firefox 都进行了这项优化。在执行js脚本时，其他线程会解析文档的其余部分，找出并加载需要通过网络加载的其他资源。通过这种方式，资源可以在并行连接上加载，从而提高总体速度。请注意，预解析器不会修改 DOM 树，而是将这项工作交由主解析器处理；预解析器只会解析外部资源（例如外部脚本、样式表和图片）的引用。

提前发送请求，提前分析外部资源内容。

>在上述的过程中，网页在加载和渲染过程中会发出“DOMContentloaded”和“onload”事件

>分别在DOM树构建（解析）完成之后，以及DOM树构建完并且网页所依赖的资源都加载完之后发生、

>因为某些资源的加载并不会阻碍DOM树的创建，所以这两个事件多数不同时发生。

## 5.从DOM树到可视化图像

1) CSS文件被CSS解析器解释成内部表示结构(CSSDOM)

2) CSS解析器工作完成之后，在DOM树上附加解释后的样式信息，这就是RenderObject树

3) RenderObject在创建的同时，Webkit会根据网页的结构创建RenderLayer，同时构建一个绘图上下文

4) 根据绘图上下文生成最终的图像（这一过程需要依赖图形库）

\* 上面介绍的是一个完整的渲染过程，但现代网页很多都是动态的，这意味着在渲染完成之后，由于网页的动画或者用户的交互，

浏览器其实一直在不停地重复执行渲染过程。（重绘重排），以上的数字表示的是基本顺序，这不是严格一致的，

这个过程可能重复也可能交叉

# 8.css图层

浏览器在渲染一个页面时，会将页面分为很多个图层，图层有大有小，每个图层上有一个或多个节点。在渲染DOM的时候，浏览器所做的工作实际上是：

1. 获取DOM后分割为多个图层

2. 对每个图层的节点计算样式结果 （Recalculate style--样式重计算）

3. 为每个节点生成图形和位置 （Layout--重排,回流）

4. 将每个节点绘制填充到图层位图中 （Paint--重绘）

5. 图层作为纹理上传至GPU

6. 符合多个图层到页面上生成最终屏幕图像 （Composite Layers--图层重组）

# 9.图层创建的条件

Chrome中满足以下任意情况就会创建图层：

1. 拥有具有3D变换的CSS属性

2. 使用加速视频解码的<video>节点

3. <canvas>节点

4. CSS3动画的节点

5. 拥有CSS加速属性的元素(will-change)

# 10.重绘(Repaint)

重绘是一个元素外观的改变所触发的浏览器行为，例如改变outline、背景色等属性。浏览器会根据元素的新属性重新绘制，使元素呈现新的外观。重绘不会带来重新布局，所以并不一定伴随重排。

需要注意的是：重绘是以图层为单位，如果图层中某个元素需要重绘，那么整个图层都需要重绘。

比如一个图层包含很多节点，其中有个gif图，gif图的每一帧，都会重回整个图层的其他节点，然后生成最终的图层位图。

所以这需要通过特殊的方式来强制gif图属于自己一个图层（translateZ(0)或者translate3d(0,0,0)

CSS3的动画也是一样（好在绝大部分情况浏览器自己会为CSS3动画的节点创建图层）

# 11.重排(Reflow 回流)

渲染对象在创建完成并添加到渲染树时，并不包含位置和大小信息。计算这些值的过程称为布局或重排,"重绘"不一定需要"重排"，比如改变某个网页元素的颜色，就只会触发"重绘"，不会触发"重排"，因为布局没有改变。

但是，"重排"必然导致"重绘"，比如改变一个网页元素的位置，就会同时触发"重排"和"重绘"，因为布局改变了。

# 12.触发重绘的属性

\* color \* background \* outline-color

\* border-style \* background-image \* outline

\* border-radius \* background-position \* outline-style \* visibility \* background-repeat \* outline-width \* text-decoration

\* background-size \* box-shadow

# 13.#触发重排(回流)的属性

盒子模型相关属性会触发重布局 定位属性及浮动也会触发重布局： 改变节点内部文字结构也会触发重布局：

\* width \* top \* text-align

\* height \* bottom \* overflow-y

\* padding \* left \* font-weight

\* margin \* right \* overflow

\* display \* position \* font-family

\* border-width \* float \* line-height

\* border \* clear \* vertival-align

\* min-height \* white-space

# 14.常见的触发重排的操作

Reflow 的成本比 Repaint 的成本高得多的多。DOM Tree 里的每个结点都会有 reflow 方法，一个结点的 reflow 很有可能导致子结点，甚至父点以及同级结点的 reflow。在一些高性能的电脑上也许还没什么，但是如果 reflow 发生在手机上，那么这个过程是非常痛苦和耗电的。所以，下面这些动作有很大可能会是成本比较高的。

当你增加、删除、修改 DOM 结点时，会导致 Reflow , Repaint。

### 当你移动 DOM 的位置

### 当你修改 CSS 样式的时候。

### 当你 Resize 窗口的时候（移动端没有这个问题，因为移动端的缩放没有影响布局视口）

### 当你修改网页的默认字体时。

### 获取某些属性时(width,height...)

注：display:none 会触发 reflow，而 visibility:hidden 只会触发 repaint，因为没有发生位置变化。

# 15.优化方案

一、如果我们需要使得动画或其他节点渲染的性能提高，需要做的就是减少浏览器在运行时所需要做的工作（减少1234中的步骤）

1. 计算需要被加载到节点上的样式结果（Recalculate style--样式重计算）

2. 为每个节点生成图形和位置（Layout--回流和重布局）

3. 将每个节点填充到图层中（Paint Setup和Paint--重绘）

4. 组合图层到页面上（Composite Layers--图层重组）

二、具体操作：

1.（移动元素）元素位置移动变换时尽量使用CSS3的transform来代替对top left等的操作

变换（transform）和透明度（opacity）的改变仅仅影响图层的组合

2.（隐藏元素）使用opacity来代替visibility

(1).使用visibility不触发重排，但是依然重绘。

(2).直接使用opacity即触发重绘，又触发重排（就是这样设计的！！！）。

(3).opacity配合图层使用，即不触发重绘也不触发重排。

原因：

透明度的改变时，GPU在绘画时只是简单的降低之前已经画好的纹理的alpha值来达到效果，并不需要整体的重绘。

不过这个前提是这个被修改opacity本身必须是一个图层，如果图层下还有其他节点，GPU也会将他们透明化

3.不要使用table布局

table-cell

4.将多次改变样式属性的操作合并成一次操作

不要一条一条地修改DOM的样式，预先定义好class，然后修改DOM的className

5.将DOM离线后再修改

由于display属性为none的元素不在渲染树中，对隐藏的元素操作不会引发其他元素的重排。

如果要对一个元素进行复杂的操作时，可以先隐藏它，操作完成后再显示。这样只在隐藏和显示时触发2次重排。

6.利用文档碎片(documentFragment)

7.不要把某些DOM节点的属性值放在一个循环里当成循环的变量

当你请求向浏览器请求一些 style信息的时候，就会让浏览器flush队列，比如：

1. offsetTop, offsetLeft, offsetWidth, offsetHeight

2. scrollTop/Left/Width/Height

3. clientTop/Left/Width/Height

4. width,height

当你请求上面的一些属性的时候，浏览器为了给你最精确的值，需要flush队列，

因为队列中可能会有影响到这些值的操作。即使你获取元素的布局和样式信息跟最近发生或改变的布局信息无关，

浏览器都会强行刷新渲染队列。

### 动画实现过程中，启用GPU硬件加速:transform: tranlateZ(0)

requestAnimationFrame

window.requestAnimationFrame()

window.requestAnimationFrame() 方法告诉浏览器您希望执行动画并请求浏览器在下一次重绘之前调用指定的函数来更新动画,该方法使用一个回调函数作为参数，这个回调函数会在浏览器重绘之前调用。回调函数会被传入一个参数，DOMHighResTimeStamp，指示requestAnimationFrame() 开始触发回调函数的当前时间

返回值

一个 long 整数，请求 ID ，是回调列表中唯一的标识。是个非零值，没别的意义。你可以传这个值给 window.cancelAnimationFrame() 以取消回调函数。

window.cancelAnimationFrame(requestID)取消一个先前通过调用window.requestAnimationFrame()方法添加到计划中的动画帧请求.

requestID先前调用window.requestAnimationFrame()方法时返回的ID.

# 16.DNS域名解析过程

第1步，查找浏览器缓存。

浏览器会检查缓存中有没有这个域名对应的解析过的IP地址，如果缓存中有，这个解析过程就将结束。

第2步，查找系统缓存

如果用户的浏览器缓存中没有，浏览器会查找操作系统缓存中是否有这个域名对应的DNS解析结果

第3步，查找路由器缓存。

如果系统缓存中也找不到，那么查询请求就会发向路由器，它一般会有自己的DNS缓存。

第4步，查找ISP DNS 缓存。(网络运营商)

运气实在不好，就只能查询ISP DNS 缓存服务器了。在我们的网络配置中都会有"DNS服务器地址"这一项，

操作系统会把这个域名发送给这里设置的DNS，也就是本地区的域名服务器，这个专门的域名解析服务器性能都会很好，它们一般都会缓存域名解析结果，当然缓存时间是受域名的失效时间控制的。大约80%的域名解析都到这里就已经完成了，所以ISP DNS主要承担了域名的解析工作。

第5步，递归搜索

最无奈的情况发生了, 在前面都没有办法命中的DNS缓存的情况下

(1) 本地 DNS服务器即将该请求转发到互联网上的根域

(2) 根域将所要查询域名中的顶级域（即blog.baidu.com中的com)的服务器IP地址返回到本地DNS。

(3) 本地DNS根据返回的IP地址，再向顶级域（就是com域）发送请求。

(4) com域服务器再将域名中的二级域（即blog.baidu.com中的baidu）的IP地址返回给本地DNS。

(5) 本地DNS再向二级域发送请求进行查询。

(6) 之后不断重复这样的过程

直到本地DNS服务器得到最终的查询结果，并返回到主机。这时候主机才能通过域名访问该网站。

# 17.减少DNS查询

1.一个多资源的站点最好使用2到4个不一样的主机来存放服务端资源。这是在减少DNS查询和允许高度并行下载之间作出的最好权衡(高度并行下载,浏览器一次能并发加载的量是受域名控制的)

2.使用Keep-alive进行持久连接

# 18. 缓存

## 1. 缓存理解

1. 缓存定义: 浏览器在本地磁盘上将用户之前请求的数据存储起来，当访问者再次需要改数据的时候无需再次发送请求，直接从浏览器本地获取数据

2. 缓存的好处:

1. 减少请求的个数

2. 节省带宽，避免浪费不必要的网络资源

3. 减轻服务器压力

4. 提高浏览器网页的加载速度，提高用户体验

## 2. 缓存分类

1. 强缓存

1. 不会向服务器发送请求，直接从本地缓存中获取数据

2. 请求资源的的状态码为: 200 ok(from memory cache)

2. 协商缓存

1. 向服务器发送请求，服务器会根据请求头的资源判断是否命中协商缓存

2. 如果命中，则返回304状态码通知浏览器从缓存中读取资源

3. 强缓存 & 协商缓存的共同点

1. 都是从浏览器端读取资源

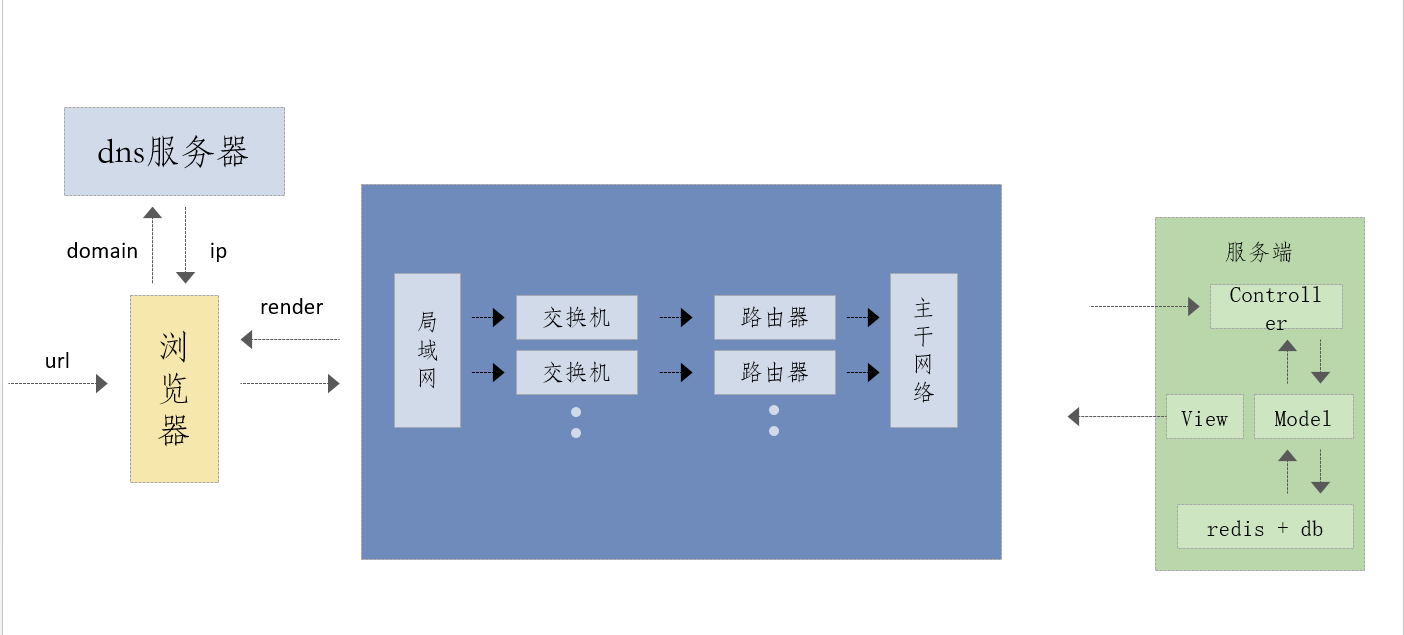
4. 强缓存 VS 协商缓存的不同点

1. 强缓存不发请求给服务器

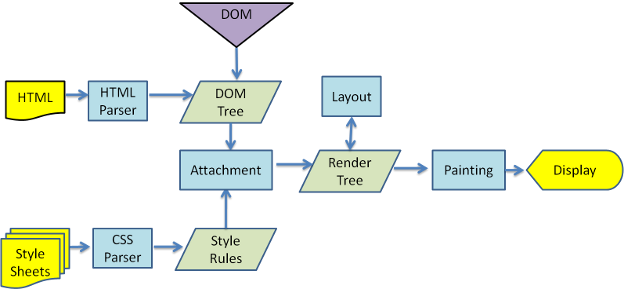
2. 协商缓存发请求给服务器，根据服务器返回的信息决定是否使用缓存

图示

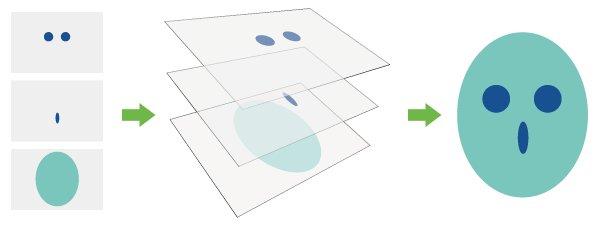
1. web前后端整体图示.



2.浏览器渲染过程



3.图层举例



# 总结

# 1.关于css阻塞：

1.style标签中的样式：

(1). 由html解析器进行解析；

(2). 不阻塞浏览器渲染；

(3). 不阻塞DOM解析；

2.link引入的外部css样式：

(1). 由CSS解析器进行解析。

(2). 阻塞浏览器渲染(原因：避免闪屏)。

(3). 不阻塞DOM结构的解析：

原因：DOM解析和CSS解析是两个并行的进程，浏览器解析DOM生成DOM Tree，

解析CSS生成CSS Tree，最终组成render Tree，再渲染页面，

DOM的解析，和CSS的解析并行的。

(4).外部css的加载会阻塞后面js语句的执行：

原因：如果脚本的内容是获取元素的样式，宽高等CSS控制的属性，

浏览器是需要计算的，也就是依赖于CSS。浏览器不知道脚本的具体内容，

因而只好等前面所有的样式下载完后，再执行JS。

3.为什么推荐使用<link>方式引入外部css？

原因：可以避免闪屏现象

4.优化方案：尽可能快的提高css加载速度

(1).使用CDN加速。

(2).对css进行压缩(用打包工具，比如webpack,gulp等，也可以通过开启gzip压缩)。

(3).减少http请求数，将多个css文件合并。

# 关于js阻塞：

### 1.阻塞DOM解析

原因：浏览器不知道脚本的内容，如果先行去解析下面的DOM，万一脚本内全删了后面的DOM，浏览器就做了无用功，浏览器无法预估里面具体做了什么操作，例如向像document.write这种操作，索性全部停住，等脚本执行完了，浏览器再干活。

### 2.阻塞页面渲染

原因：同理js中也可以给DOM设置样式，浏览器同样等该脚本执行完毕，再继续干活，避免做无用功。

### 3.阻塞后续js逻辑的执行，但不阻塞js等其他资源的加载

原因：维护依赖关系

备注：css的解析和js的执行是互斥（互相排斥），css解析的时候js停止执行，js执行的时候css停止解析。

闭包

# 理解闭包

1. 如何产生闭包?

\* 当一个嵌套的内部(子)函数引用了嵌套的外部(父)函数的变量(函数)时, 就产生了闭包

2. 闭包到底是什么?

\* 使用chrome调试查看

\* 理解一: 闭包是嵌套的内部函数(绝大部分人)

\* 理解二: 包含被引用变量(函数)的对象(极少数人)

\* 注意: 闭包存在于嵌套的内部函数中

3. 产生闭包的条件?

\* 函数嵌套

\* 内部函数引用了外部函数的数据(变量/函数)

\* 执行外部函数

# 常见的闭包

1. 将函数作为另一个函数的返回值

2. 将函数作为实参传递给另一个函数调用

# 3.闭包的作用

1. 使函数内部的变量在函数执行完后, 仍然存活在内存中(延长了局部变量的生命周期)

2. 让函数外部可以间接操作(读写)到函数内部的数据(变量/函数)

# 4.闭包的生命周期

1. 产生: 在嵌套内部函数定义执行完时就产生了(不是在调用)

2. 死亡: 在嵌套的内部函数成为垃圾对象时

# 5.闭包的应用\_自定义JS模块

闭包的应用2 : 定义JS模块

\* 具有特定功能的js文件

\* 将所有的数据和功能都封装在一个函数内部(私有的)

\* 只向外暴露一个包信n个方法的对象或函数

\* 模块的使用者, 只需要通过模块暴露的对象调用方法来实现对应的功能

# 闭包的缺点及解决

1. 缺点

\* 函数执行完后, 函数内的局部变量没有释放, 占用内存时间会变长

\* 容易造成内存泄露

2. 解决

\* 能不用闭包就不用

\* 及时释放

# Object构造函数模式

Object构造函数模式

\* 套路: 先创建空Object对象, 再动态添加属性/方法

\* 适用场景: 起始时不确定对象内部数据

\* 问题: 语句太多

# 8.对象字面量模式

\* 套路: 使用{}创建对象, 同时指定属性/方法

\* 适用场景: 起始时对象内部数据是确定的

\* 问题: 如果创建多个对象, 有重复代码(方法)

# 9.工厂模式

\* 套路: 通过工厂函数动态创建对象并返回

\* 适用场景: 需要创建多个对象

\* 问题: 对象没有一个具体的类型, 都是Object类型

# 例子

function createPerson(name, age) {

const obj = {

name: name,

age: age,

setName: function (name) {

this.name = name

}

}

return obj

}

# 10. 自定义构造函数模式

\* 套路: 自定义构造函数, 通过new创建对象

\* 适用场景: 需要创建多个类型确定的对象

\* 问题: 每个对象都有相同的数据, 浪费内存

例子function Person(name, age) {

this.name = name

this.age = age

this.setName = function (name) {

this.name = name

}

}

# 11.构造函数+原型的组合模式

\* 套路: 自定义构造函数, 属性在函数中初始化, 方法添加到原型上

\* 适用场景: 需要创建多个类型确定的对象

例子

// 定义人的类型

function Person(name, age) {

// 在构造函数给实例添加属性

this.name = name

this.age = age

}

// 给原型对象添加方法

Person.prototype.setName = function (name) {

this.name = name

}

var p1 = new Person('tom', 12)

var p2 = new Person('jack', 13)

console.log(p1, p2)

# 12.原型链继承

1. 套路

1. 定义父类型构造函数

2. 给父类型的原型添加方法

3. 定义子类型的构造函数

4. 创建父类型的对象赋值给子类型的原型

5. 将子类型原型的构造属性设置为子类型

6. 给子类型原型添加方法

7. 创建子类型的对象: 可以调用父类型的方法

2. 关键

1. 子类型的原型为父类型的一个实例对象

# 重点

// 让子类型的原型指向父类型的实例

Child.prototype = new Parent()

// 给创建的原型对象指定constructor指向当前Child

Child.prototype.constructor = Child

13. 借用构造函数继承(假的)

1. 套路:

1. 定义父类型构造函数

2. 定义子类型构造函数

3. 在子类型构造函数中调用父类型构造

2. 关键:

1. 在子类型构造函数中通过call()调用父类型构造函数

注意Person.call(this, name, age) // 相当于执行: this.Person(name, age)；

obj1.fn.call(obj2) // 通过call/apply()让任意函数/方法成指定的任意对象的方法进行调用

# 13. 原型链+借用构造函数的组合继承

1. 利用原型链实现对父类型对象的方法继承

2. 利用call()借用父类型构建函数初始化相同属性

function Student(name, age, price) {

Person.call(this, name, age) // 为了得到父类型的属性

this.price = price

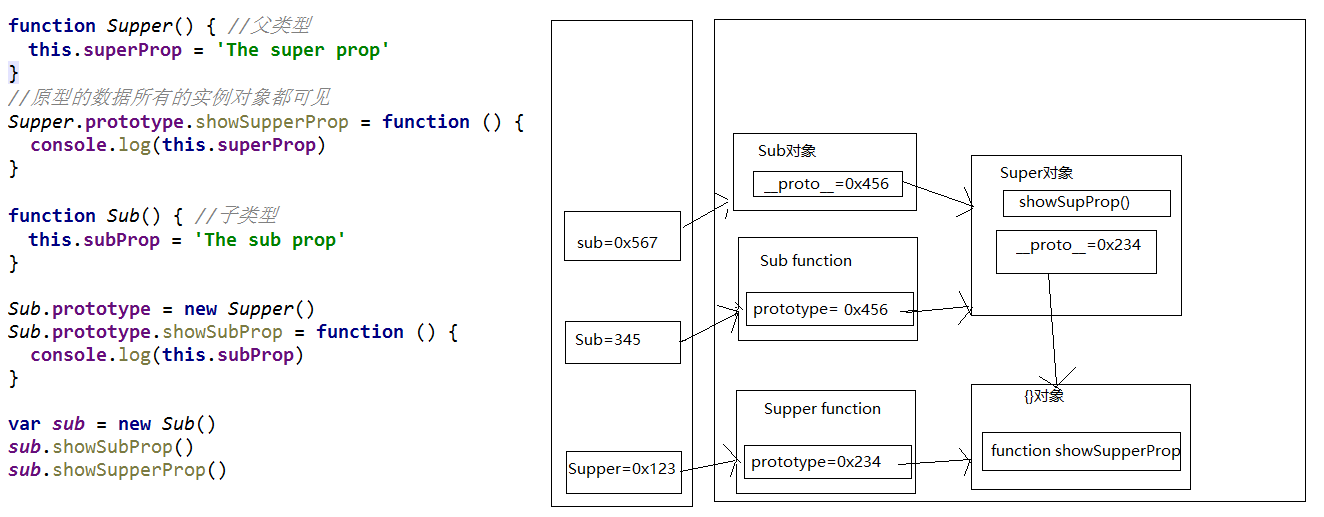
}

// 让子类型原型指向父类型的实例

Student.prototype = new Person()

// 让新的原型对象的constructor指向子类型函数

Student.prototype.constructor = Student



# 14.浏览器内核

1. 什么是浏览器内核?

\* 支持浏览器运行的最核心的程序

2. 不同的浏览器可能不太一样

\* Chrome, Safari: webkit

\* firefox: Gecko

\* IE: Trident

\* 360,搜狗等国内浏览器: Trident + webkit

3. 内核由很多模块组成

\* html,css文档解析模块 : 负责页面文本的解析

\* dom/css模块 : 负责dom/css在内存中的相关处理

\* 布局和渲染模块 : 负责页面的布局和效果的绘制

\* 布局和渲染模块 : 负责页面的布局和效果的绘制

\* 定时器模块 : 负责定时器的管理

\* 网络请求模块 : 负责服务器请求(常规/Ajax)

\* 事件响应模块 : 负责事件的管理

# 定时器引发的思考

1. 定时器真是定时执行的吗?

\* 定时器并不能保证真正定时执行

\* 一般会延迟一丁点(可以接受), 也有可能延迟很长时间(不能接受)

2. 定时器回调函数是在分线程执行的吗?

\* 在主线程执行的, js是单线程的

3. 定时器是如何实现的?

\* 事件循环模型(后面讲)

# 16.JS是单线程的

1. 如何证明js执行是单线程的?

\* setTimeout()的回调函数是在主线程执行的

\* 定时器回调函数只有在运行栈中的代码全部执行完后才有可能执行

2. 为什么js要用单线程模式, 而不用多线程模式?

\* JavaScript的单线程，与它的用途有关。

\* 作为浏览器脚本语言，JavaScript的主要用途是与用户互动，以及操作DOM。

\* 这决定了它只能是单线程，否则会带来很复杂的同步问题

3. 代码的分类:

\* 初始化代码

\* 回调代码

4. js引擎执行代码的基本流程

\* 先执行初始化代码: 包含一些特别的代码

\* 设置定时器

\* 绑定监听

\* 发送ajax请求

\* 后面在某个时刻才会执行回调代码

# 注意

1. 先执行初始化代码

2. 后执行回调代码

# 17.事件循环模型

1. 所有代码分类

\* 初始化执行代码(同步代码): 包含绑定dom事件监听, 设置定时器, 发送ajax请求的代码

\* 回调执行代码(异步代码): 处理回调逻辑

2. js引擎执行代码的基本流程:

\* 初始化代码===>回调代码

3. 模型的2个重要组成部分:

\* 事件管理模块

\* 回调队列

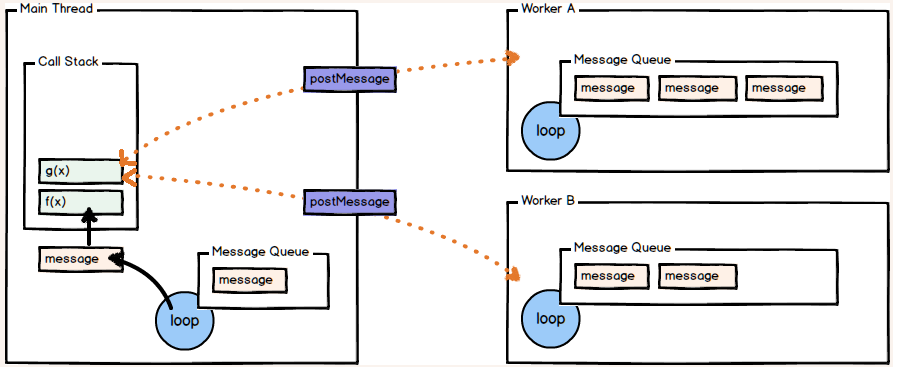
4. 模型的运转流程

\* 执行初始化代码, 将事件回调函数交给对应模块管理

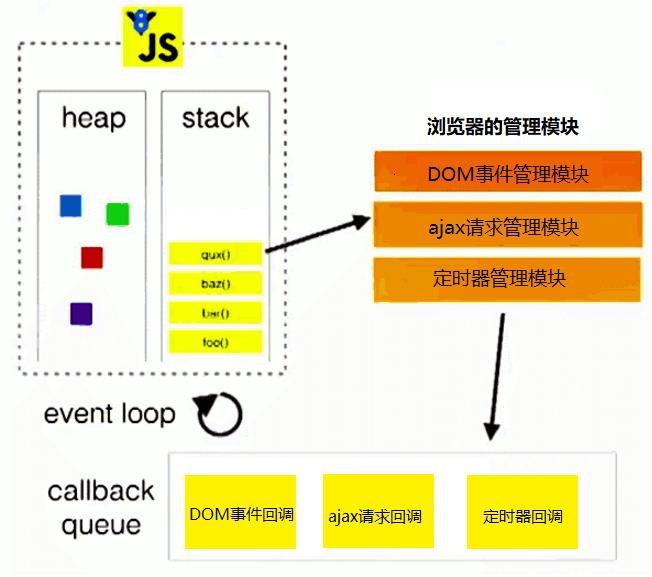
\* 当事件发生时, 管理模块会将回调函数及其数据添加到回调列队中

\* 只有当初始化代码执行完后(可能要一定时间), 才会遍历读取回调队列中的回调函数执行

H5 Web Workers(多线程)



事件循环模型



总结

1. 闭包

## 1). 理解:

当嵌套的内部函数引用了外部函数的变量时就产生了闭包

通过chrome工具得知: 闭包本质是内部函数中的一个对象, 这个对象中包含引用的变量属性

## 2). 作用:

延长局部变量的生命周期

让函数外部能操作内部的局部变量

## 3). 写一个闭包程序

function fn1() {

var a = 2;

function fn2() {

a++;

console.log(a);

}

return fn2;

}

var f = fn1();

f();

f();

## 4). 闭包应用:

循环遍历加监听: 给多个li加点击监听, 读取当前下标

模块化: 封装一些数据以及操作数据的函数, 向外暴露一些行为

JS框架(jQuery)大量使用了闭包

## 5). 缺点:

变量占用内存的时间可能会过长

可能导致内存泄露

解决: 及时释放 : f = null; //让内部函数对象成为垃圾对象

# 2. 内存溢出与内存泄露

## 1). 内存溢出

一种程序运行出现的错误

当程序运行需要的内存超过了剩余的内存时, 就出抛出内存溢出的错误

## 2). 内存泄露

占用的内存没有及时释放

内存泄露积累多了就容易导致内存溢出

常见的内存泄露:

意外的全局变量

没有及时清理的计时器或回调函数

闭包

# 3. 对象的创建模式

## 1). Object构造函数模式

var obj = {}

obj.name = 'Tom'

obj.setName = function(name){this.name=name}

## 2). 对象字面量模式

var obj = {

name : 'Tom',

setName : function(name){this.name = name}

}

## 3). 构造函数模式

function Person(name, age) {

this.name = name

this.age = age

this.setName = function(name){this.name=name}

}

new Person('tom', 12)

## 4). 构造函数+原型的组合模式

function Person(name, age) {

this.name = name

this.age = age

}

Person.prototype.setName = function(name){this.name=name}

new Person('tom', 12)

# 4. 继承模式

## 1). 原型链继承 : 得到方法

function Parent(){}

Parent.prototype.test = function(){}

function Child(){}

Child.prototype = new Parent() //子类原型指向父类的实例

Child.prototype.constructor = Child //原型的构造器指向构造函数

var child = new Child()

child.test() //调用父类型的方法

## 2). 借用构造函数 : 得到属性

function Parent(xxx){this.xxx = xxx}

Parent.prototype.test = function(){}

function Child(xxx,yyy){

Parent.call(this, xxx) //借用父类型的构造函数 相当于:this.Parent(xxx)

}

var child = new Child('a', 'b') //child.xxx为'a', 但child没有test()

## 3). 组合

function Parent(xxx){this.xxx = xxx}

Parent.prototype.test = function(){}

function Child(xxx,yyy){

Parent.call(this, xxx) //借用构造函数 this.Parent(xxx)

}

Child.prototype = new Parent() //得到test()

Child.proptotype.constructor = Child

var child = new Child() //child.xxx为'a', 也有test()

# 5. 线程与进程

## 1). 进程:

程序的一次执行, 它占有一片独有的内存空间

可以通过windows任务管理器查看进程

## 2). 线程:

是进程内的一个独立执行单元

是程序执行的一个完整流程

是CPU的最小的调度单元

## 3). 关系

一个进程至少有一个线程(主)

程序是在某个进程中的某个线程执行的

# 6. 浏览器内核模块组成

## 1). 主线程

js引擎模块 : 负责js程序的编译与运行

html,css文档解析模块 : 负责页面文本的解析

DOM/CSS模块 : 负责dom/css在内存中的相关处理

布局和渲染模块 : 负责页面的布局和效果的绘制(内存中的对象)

## 2). 分线程

定时器模块 : 负责定时器的管理

事件响应模块 : 负责事件的管理

网络请求模块 : 负责Ajax请求

## 3). js线程

js是单线程执行的(回调函数也是在主线程)

H5提出了实现多线程的方案: Web Workers --->Worker

只能是主线程更新界面

# 3. 定时器问题:

定时器并不真正完全定时

如果在主线程执行了一个长时间的操作, 可能导致延时才处理

# 7. 事件循环机制

![事件循环](http://i.imgur.com/sKxdHu9.png)

## 1). 代码分类

初始化执行代码: 包含绑定dom事件监听, 设置定时器, 发送ajax请求的代码

回调执行代码: 处理回调逻辑

## 2). js引擎执行代码的基本流程:

初始化代码===>回调代码

## 3). 模型的2个重要组成部分:

事件管理模块

回调队列

## 4). 模型的运转流程

执行初始化代码, 将事件回调函数交给对应模块管理

当事件发生时, 管理模块会将回调函数及其数据添加到回调列队中

只有当初始化代码执行完后(可能要一定时间), 才会遍历读取回调队列中的回调函数执行