**UI多线程**

1. 浏览器常驻的系统线程
2. JS引擎线程（解释执行JS代码，用户输入，网络请求）
3. GUI线程（绘制用户界面，与JS主线程是互斥的）
4. http网络请求线程（处理用户的GET，POST请求，返回结果后将回调函数插入任务队列）
5. 定时触发器线程（setTimeout，setInterval等待时间结束后把执行函数加入任务队列中）
6. 浏览器事件处理线程（将click，mouse等交互事件发生后将这些事件放入事件队列中）
7. JS引擎线程：单线程，同一时间只能执行一个任务，JS引擎线程处于运行状态时，GUI渲染线程将处于冻结状态
8. JS设计出来就是为了与用户交互，处理DOM，加入JS是多线程同一时间一个线程想要修改DOM，另一个线程想要删除DOM，问题就会变得复杂，如果引入加锁机制，就会陷入其他语言尴尬的语境
9. 单线程计算能力有限，大量数据需要渲染，可以配合后端进行操作，如VUE与node.js配合，也就是SSR技术（服务器端渲染）
10. 单线程又异步执行基本都是基于事件驱动的
11. JS 执行机制：（栈是先进后出）

异步

Event Table

同步

主线程

同步任务还是异步任务

任务进入执行栈

1. 执行栈： 函数执行时，创建一帧（预编译过程），然后把这一帧压入栈中，嵌套函数后进先出。

**call,apply模拟实现**

1. eval()，会将作用域链改变
2. 模拟实现函数原型中的call方法

var obj={

name:'Bessi',

age:18

}

function show(name,age){

console.log('name:'+this.name,'age:'+this.age);

return 'hello world!';

}

// show.call(obj);

Function.prototype.newCall=function(){

var ctx=arguments[0]||window;

// ctx is obj now

ctx.fn=this;

// ctx's attribute fn is function named 'show'

var args=[];

// ctx.fn is Function.prototype now

for(var i=1,len=arguments.length;i<len;i++){

args.push('arguments['+i+']');

}

var result=eval('ctx.fn('+args.join()+')');

// inner 'this' is ctx(obj)

delete ctx.fn;

return result;

// simulate the value returned by function show(){}

}

show.newCall(obj)

1. 模拟封装function的apply方法：

function show(name,age){

console.log('name:'+name,'age:'+age);

return 'hello world!';

}

Function.prototype.newApply=function(ctx,arr){

var ctx=ctx||window;

var result;

// ctx is obj now

ctx.fn=this;

// ctx's attributes fn is function named 'show'

if(!arr){

result=ctx.fn();

delete ctx.fn;

return result;

}

else{

// ctx.fn is Function.prototype now

var args=[];

for(var i=0,len=arr.length;i<len;i++){

args.push('arr['+i+']');

}

var result=eval('ctx.fn('+args.join()+')');

// inner 'this' is ctx(obj)

delete ctx.fn;

return result;

// simulate the value returned by function show(){}

}

}