潜力无限的编程语言 Javascript

诗鸣 – F2E@Taobao

TaobaoUED

淘宝北京研发中心 2012-04

- Who Are We?

  淘宝前端开发工程师

  http://ued.taobao.com

  Taobao UED

Topic

- Javascript语言
- 引擎
- Web性能

Javascript语言

- Javascript特性
- 作用域、作用域链、闭包
- 函数式编程

Javascript语言特性

Javascript语言——特性

高阶函数：
•将函数作为参数；
•可以返回一个函数。

动态类型：
• 晚绑定；
• 可以赋给变量任意类型的值，并可随时更改类型。

灵活的对象模型：
•JavaScript 的对象模型使用一种相对不常见的方式进行继承 —— 称为原型 ；
•不是 Java 语言中更常见的基于类的对象模型。

Javascript语言——特性

高阶函数-传递函数：

例

```
var woman = function () {
      alert('beauty');
}
var man = function () {
      alert('cool')
}
var swap = function () {
      var temp = woman,
            woman = man,
            man = temp;
      alert('交换成功')
}
```

查看示例

Javascript语言——特性

动态类型：

例

```
var a = new Object();
a.sex = ‘美女’;
a.age = 21;
a.say = function(){ return '帅哥你好~' };
alert(a.sex);
alert(a.age);
alert(a.say());
a.say = ‘从函数变成字符串’;
alert(a.say)
```

查看示例

可以赋给变量任意类型的值，并可随时更改类型

Javascript语言——特性

动态类型——晚绑定：

例

```
//先定义一个函数a
function a(){
      alert('hello world!')
}

//在点击按钮时才执行
<button onclick="a()">执行a</button>
```

查看示例

Javascript语言——特性

对象模型——创建一个构造函数：

例

```
//创建一个人并给Ta赋予一些属性
var person = function () {
      this.name = '多多';
      this.say = function () {
            return ‘亲~‘
      };
}
//基于person的原型定义一个新对象woman
var woman = new person();
alert('这位美女的名字是'+woman.name+', 她对大家说: “'+woman.say+'”');
```

查看示例

Javascript语言——特性

对象模型——通过原型继承：

查看示例

例

```
var person = function () {
      this.name = '多多';
      this.say = function () {return
'Remember'};
}
var woman = function () {
      this.say = function () {
            return '亲们~请多关照'
      }
}
woman.prototype = new person();
var x = new woman();
alert(‘这位美女的名字是’+x.name+‘; 美女对大家说: “’+x.say()+'”');
```

作用域、作用域链、闭包

Javascript语言——作用域、作用域链、闭包

作用域（Scope）

作用域就是变量与函数的可访问范围，即作用域控制着变量与函数的可见性和生命周期。在JavaScript中，变量的作用域有全局作用域和局部作用域两种。

Javascript语言——作用域、作用域链、闭包

全局作用域(Global Scope)

在代码中任何地方都能访问到的对象拥有全局作用域，以下几种情形拥有全局作用域：

Javascript语言——作用域、作用域链、闭包

全局作用域

① 最外层函数和在最外层函数外面定义的变量拥有全局作用域：

例

```
var a = 'global';
function b() {
        var c = 'local';
        function d() {alert(c);}
        d();
}
alert(a); //global
alert(c); //not defined
b(); //local
d(); //no defined
```

查看示例

Javascript语言——作用域、作用域链、闭包

全局作用域

②所有未定义直接赋值的变量自动声明为拥有全局作用域：

例

```
function a() {
        var b = 'local';
        c = 'global';
}
a();
alert(c); //global
alert(b); //not defined
```

变量c拥有全局作用域，而b在函数外部无法访问到。

查看示例

Javascript语言——作用域、作用域链、闭包

全局作用域

③所有window对象的属性拥有全局作用域：

一般情况下，window对象的内置属性都都拥有全局作用域，例如

window.name

window.location

window.top

……

Javascript语言——作用域、作用域链、闭包

局部作用域(Local Scope)

和全局作用域相反，局部作用域一般只在固定的代码片段内可访问到，最常见的例如函数内部，所以也被称为函数作用域，例如下列代码中的b和函数c都只拥有局部作用域。

例

```
function a() {
        var b = 'local';
        function c() {alert(b);}
        c();
}
alert(b); //not defined
c(); //not defined
```

Javascript语言——作用域、作用域链、闭包

作用域链(Scope Chain)

实际上，JavaScript全家都是对象。

在JavaScript中，函数也是对象，函数对象和其它对象一样，拥有可以通过代码访问的属性和一系列仅供JavaScript引擎访问的内部属性。其中一个内部属性是[[Scope]]，由ECMA-262标准第三版定义，该内部属性包含了函数被创建的作用域中对象的集合，这个集合被称为函数的作用域链，它决定了哪些数据能被函数访问。

Javascript语言——作用域、作用域链、闭包

当一个函数创建后，它的作用域链会被创建此函数的作用域中可访问的数据对象填充。

例

```
function a(x, y) {
        var b = x + y;
        return b;
}
```

在函数a创建时，它的作用域链中会填入一个全局对象，该全局对象包含了所有全局变量

Javascript语言——作用域、作用域链、闭包

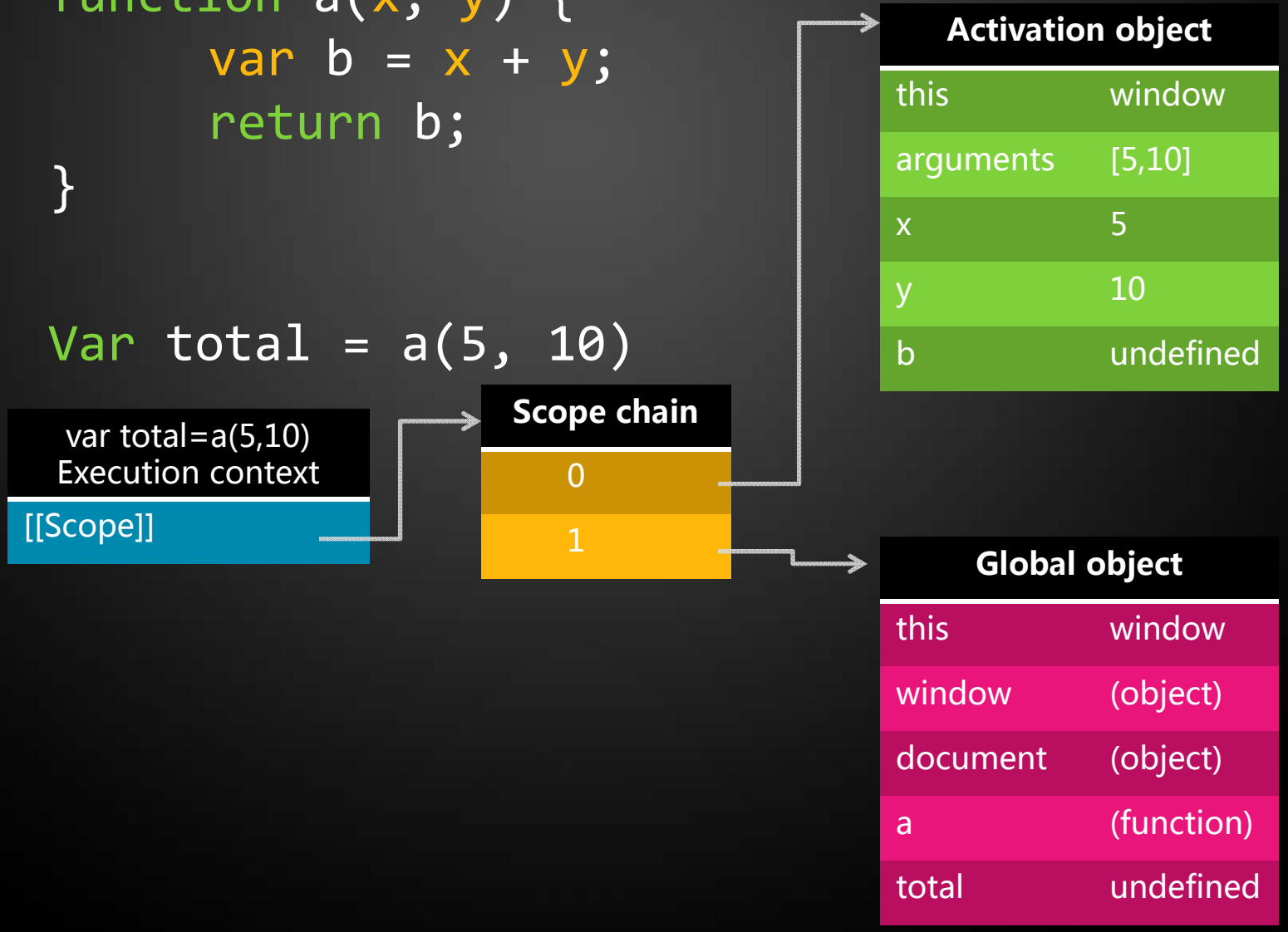
执行此函数时会创建一个称为“运行期上下文(execution context)”的内部对象，运行期上下文定义了函数执行时的环境。每个运行期上下文都有自己的作用域链，用于标识符解析，当运行期上下文被创建时，它的作用域链初始化为当前运行函数的[[Scope]]所包含的对象。

Javascript语言——作用域、作用域链、闭包

例

```
function a(x, y) {
        var b = x + y;
        return b;
}

Var total = a(5, 10)
```



Javascript语言——作用域、作用域链、闭包

在函数的运行过程中，每遇到一个变量，都会经历一次标示符解析过程以决定从哪里获取或存储数据。

- 该过程搜索运行期上下文的作用域链，查找同名的标示符；
- 搜索过程从作用域链顶部开始，也就是当前运行函数的活动对象；
- 如果找到，就使用这个标示符，反之继续搜索作用域链中的下一个对象；
- 搜索会持续进行，直到标示符被找到，或者没有可用于搜索的对象为止（返回undefined）；
- 这个搜索过程会消耗性能；

如果名字相同的两个变量存在于作用域链的不同部分，那么标示符就是遍历作用域链时最先找到的那个。

Javascript语言——作用域、作用域链、闭包

标示符解析

例

```
var a = ‘set1’
function foo() {
      a = ‘set2’;
      return a
}

alert(foo());
```

```
var a = ‘set1’
function foo(){
      b = ‘set2’
      return a
}

alert(foo());
```

查看示例

Javascript语言——作用域、作用域链、闭包

如何从函数外部读取局部变量？

例

```
function a() {
        var b = 999;
        alert(c);
        function d() {
                var c= 1;
                alert(b);
        }
}
```

Javascript语言——作用域、作用域链、闭包

如何从外部读取局部变量？

例

```
function a() {
      var b = 999;
      function d() {alert(b);}
      return d;
}

Var foobar = a();
foobar(); //999
```

Javascript语言——作用域、作用域链、闭包

闭包是 What ?

Javascript语言——作用域、作用域链、闭包

闭包是函数和执行它的作用域组成的综合体

——《JavaScript权威指南》

闭包就是一种在函数内访问和操作外部变量的方式

所有的函数都是闭包

函数可以访问它被创建时的上下文环境，称为闭包

——《JavaScript语言精粹》

内部函数比它的外部函数具有更长的生命周期

Javascript语言——作用域、作用域链、闭包

更简单的定义——
闭包就是能够读取其他函数内部变量的函数。

由于在Javascript语言中，只有函数内部的子函数才能读取局部变量，因此可以把闭包简单理解成"定义在一个函数内部的函数"。

所以，在本质上，闭包就是将函数内部和函数外部连接起来的一座桥梁。

Javascript语言——作用域、作用域链、闭包

例

```
for(var i = 0;i<elements.length;i++){
  elements[i].onclick = function(){
    alert(i);
  };
}
```

查看示例

Javascript语言——作用域、作用域链、闭包

例

```
for(var i = 0;i<elements.length;i++){
  (function(n){
    elements[n].onclick = function(){
      alert(n);
    });
  })(i);
}
```

查看示例

Javascript语言——作用域、作用域链、闭包

闭包的应用场景

- 实现私有成员
- 保护命名空间
- 避免污染全局变量
- 变量需要长期驻留在内存

闭包应用示例

Javascript语言——作用域、作用域链、闭包

使用闭包的注意事项

•由于闭包会使得函数中的变量都被保存在内存中，内存消耗很大，所以不能滥用闭包，否则会造成网页的性能问题，在IE中可能导致内存泄露。解决方法是，在退出函数之前，将不使用的局部变量全部删除。

•闭包会在父函数外部，改变父函数内部变量的值。所以，如果你把父函数当作对象（object）使用，把闭包当作它的公用方法（Public Method），把内部变量当作它的私有属性（private value），这时一定要小心，不要随便改变父函数内部变量的值。

函数式编程

Javascript语言——函数式编程

什么是函数式编程：

> 函数式编程是种编程范型，它将电脑运算视为函数的计算。函数式编程的重点是函数的定义而不是像命令式编程那样强调状态机（state machine）的实现。
>
> ——维基百科

Javascript语言——函数式编程

JavaScript中函数的特性:

- 函数并不总是需要名称。
- 函数可以像其他值一样分配给变量。
- 函数表达式可以编写并放在括号中，留待以后应用。
- 函数可以作为参数传递给其他函数。

Javascript语言——函数式编程

JavaScript中函数式编程的一些特性:

- 函数是顶层对象
- 高阶函数
- 闭包
- 函数柯里化（Currying）
- 函数式代码风格

Javascript语言——函数式编程

函数是顶层对象

- function是JavaScript中最基础的模块，本身为一种特殊对象（Object）；
- 不依赖于任何其他的对象而可以独立存在；
- 一切皆是可传入function的值，连function本身也不例外。

Javascript语言——函数式编程

高阶函数——对函数的进一步抽象

例

```
//Array对象的sort方法，需传入一比较函数
var myarr = [2, 5, 7, 3];
var byAsc = function (x,y) { return x-
y; };
var byDesc = function (x,y) { return y-
x; };
myarr.sort(byAsc);
alert(myarr); //2,3,5,7
myarr.sort(byDesc);
alert(myarr); //7,5,3,2
```

查看示例

Javascript语言——函数式编程

高阶函数——对函数的进一步抽象

例

```
//或者直接用一个匿名函数：
var myarr = [2, 5, 7, 3];
myarr.sort(function (x, y) {
        return x-y;
});
```

查看示例

sort既是JS引擎自身提供的一个高阶函数。sort传入的比较函数（byAsc, byDesc）是没有任何预先的假设的，sort是对整个排序方法的二阶抽象，因此称之为“高阶”函数。

Javascript语言——函数式编程

函数柯里化（Currying）

在计算机科学中，柯里化是把接受多个参数的函数变换成接受一个单一参数（最初函数的第一个参数）的函数，并且返回接受余下的参数而且返回结果的新函数的技术。
——详见 维基百科

Javascript语言——函数式编程

柯里化就是预先将函数的某些参数传入，得到一个简单的函数。但是预先传入的参数被保存在闭包中，因此会有一些奇特的特性。比如：

例

```
var adder = function(num){
        return function(y){
                return num + y;
        }
}
var inc = adder(1);
var dec = adder(-1);
//inc, dec现在是两个新的函数，作用是将传入的参数值(+/-)1
alert(inc(99));//100
alert(dec(101));//100
alert(adder(100)(2));//102
alert(adder(2)(100));//102
```

查看示例

Javascript语言——函数式编程

函数式代码风格——连续运算

例

```
//连续赋值
var a = b = c = d = 1000;

//短路条件
a = a || '';
a && a++;

//三元表达式
a>b ? a++ : a--;
```

Javascript语言——函数式编程

函数式代码风格——链式调用

例

```
KISSY.one('#foo')
            .height('100px')
            .width('100px')
            .addClass('unstyle')
            .click(function () {
                alert('hello')
            });
```

Javascript语言——函数式编程

使用函数式编程优点：

- 函数内的运算对函数外无副作用
- 便于调试及单元测试
- 编写更加优美的回调
- 高内聚，低耦合的一种体现

Javascript引擎

- 主流浏览器的JS引擎
- 加载和执行
- 垃圾回收

Javascript引擎——简介

JavaScript引擎是一个专门处理JavaScript脚本的软件程序，一般会附带在网页浏览器之中。

—— 引自 维基百科

V8

Carakan

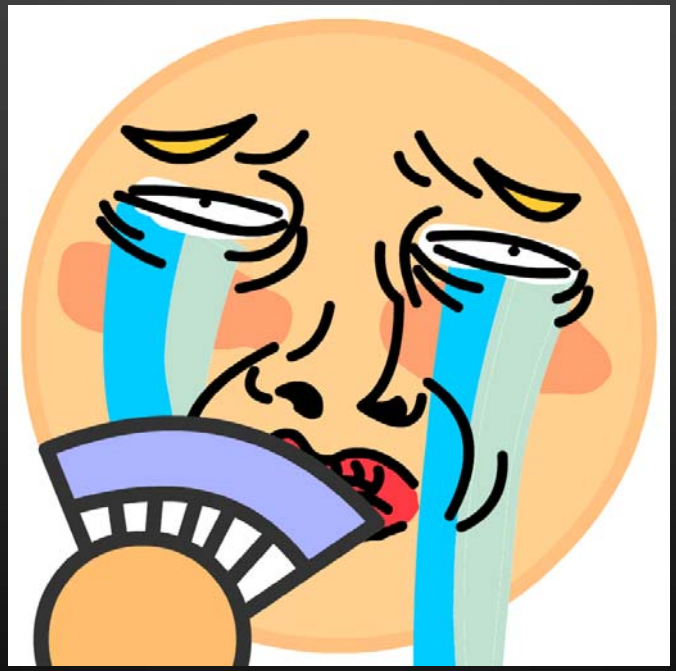
JaegerMonkey

Nitro

JScript

Javascript引擎——主流浏览器的JS引擎

正是由于浏览器JS引擎的不同，导致Javascript的不兼容。



Javascript引擎——加载和执行

从源码到可执行代码

编译(compiled)

解释
(interpreted)

Javascript引擎——加载和执行

source.js(源码)

```
(function foo(){
     alert('helloworld');
})();
```

编译

helloworld.exe
(二进制码)

```
01001011101101…
```

运行

运行时环境

```
Runtime.exec('helloworld.exe')
```

编译

Javascript引擎——加载和执行

source.js(源码)

```
(function foo(){
      alert('helloworld');
})();
```

编译&运行

helloworld.exe
(二进制码)

解释器执行伪代码

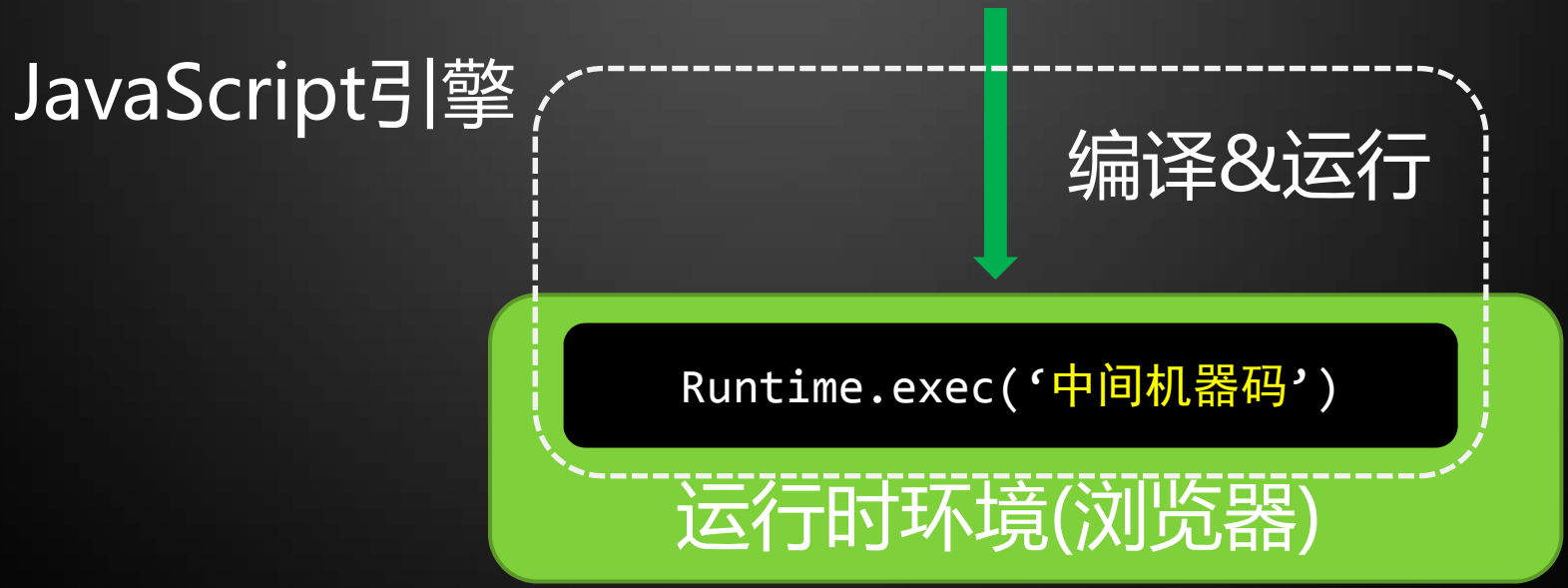
```
Runtime.exec('中间机器码')
```

运行时环境

解释

Javascript引擎——加载和执行

```
(function foo(){
     alert('helloworld');
})();
```



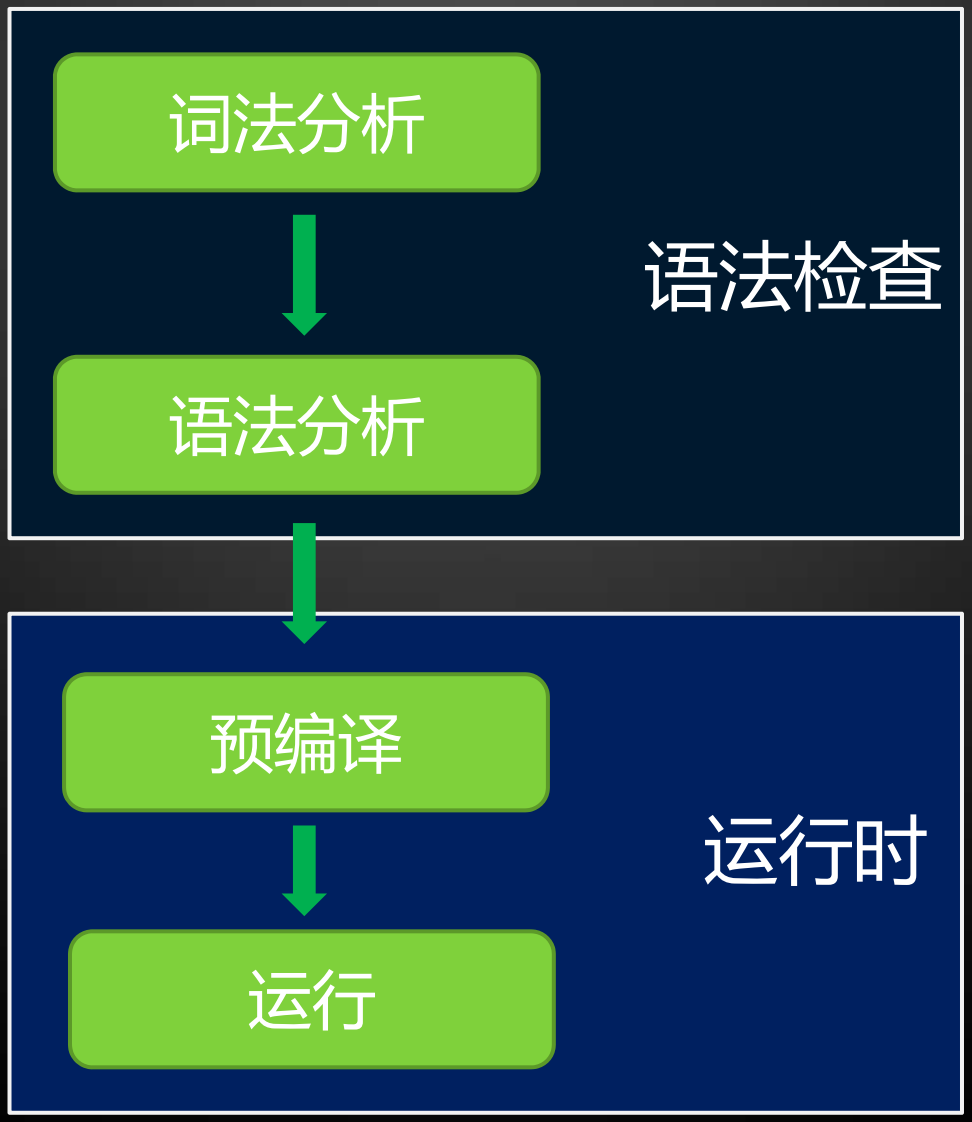
解释

Javascript引擎——加载和执行

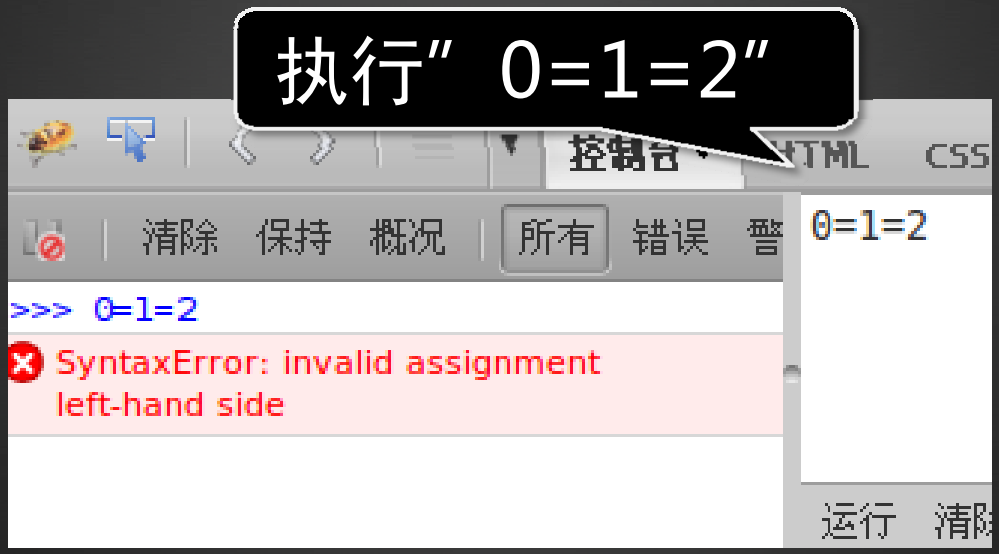
JavaScript 是解释型语言

Javascript引擎——加载和执行

JavaScript代码执行的过程



Javascript引擎——加载和执行



在语法检查阶段报错

这是一个**语法错误**

程序没有开始运行

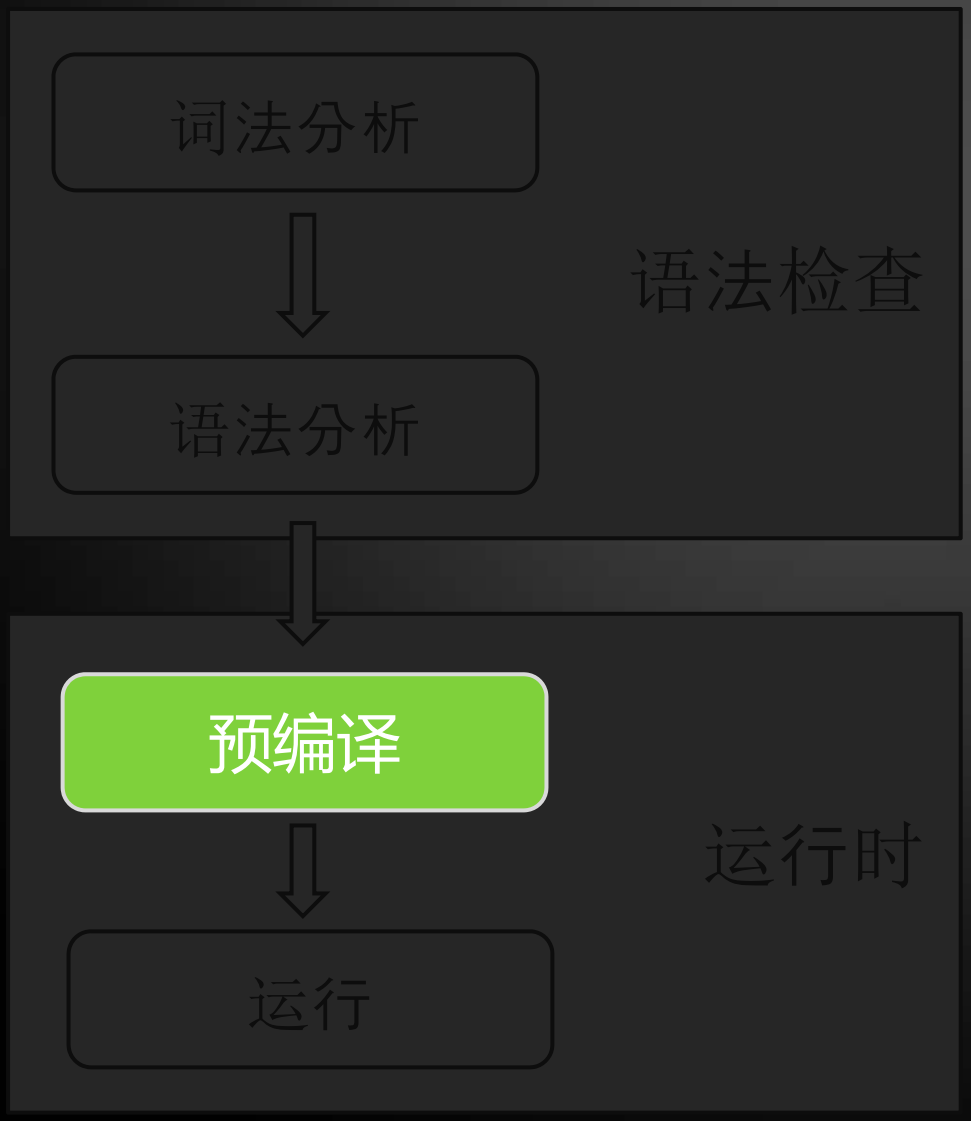
Javascript引擎——加载和执行

通过了语法检查

在**运行时报错**

程序已经开始运行

Javascript引擎——加载和执行



1,将上下文中var申明的变量放入”栈”中并赋值为undefined

2,读入”定义”的函数

Javascript引擎——加载和执行

例

```
alert(a); //会报错吗？
var a = 1;
```

Javascript引擎——加载和执行

扫描整段代码,将a放入当前上下文的栈中,赋值为undefined

在栈中找到a并得到值undefined
弹出”undefined”

将a赋值为1

释

```
alert(a);
var a = 1;
```

局部变量的预编译

Javascript引擎——加载和执行

例

```
a();
function a(){
  alert('Tom');
}
var a = function(){
  alert('Jim');
};
a();
```

2，预编译读入a的定义

1，变量a入栈

函数的预编译

Javascript引擎——加载和执行

例

```
a();
function a(){
  alert('Tom');
}
var a = function(){
  alert('Jim');
};
a();
```

执行a()，弹出Tom

a被重新赋值

执行a()，弹出Jim

编译后的运行

Javascript引擎——加载和执行

下面代码的运行结果？

Javascript引擎——加载和执行

题

```
a();  //2
function a(){
    alert(1);
}
a();  //2
function a(){
    alert(2);
}
a();  //2
var a = function(){
    alert(3);
};
a();  //3
```

Javascript引擎——加载和执行

常见问题

Javascript引擎——加载和执行

判断变量存在

例

```
//a未声明时报错，不推荐
alert(a === undefined);


//推荐
alert(typeof a === ‘undefined’);
```

Javascript引擎——加载和执行

函数执行前，函数内部变量均被声明

例

```
function(){
    alert(a); //显示undefined，不报错
    if(false){
        var a = 1; //不会执行到，亦被声明
    }
}()
```

Javascript引擎——垃圾回收

JavaScript不需要手动地释放内存，它使用一种自动垃圾回收机制（garbage collection）。当一个对象无用的时候，即程序中无变量引用这个对象时，就会从内存中释放掉这个变量。

Javascript引擎——垃圾回收

例

```
function a() {
  this.text = 'a的弹窗'
}
function b() {
  this.text = 'b的弹窗'
}
function c() {
  var x = new a();
  y = new b();
  return y;
}
c();
alert(y.text)
```

Javascript引擎——垃圾回收

闭包和垃圾回收

Javascript引擎——垃圾回收

例

```
function a() {
  var i = 0;
  function b() {
      i++;
      alert(i)
  }
  return b
}
var c = a();
c(); //1
c(); //2
c(); //3
```

Web性能

- 快速响应的用户界面
- DOM编程
- Ajax性能

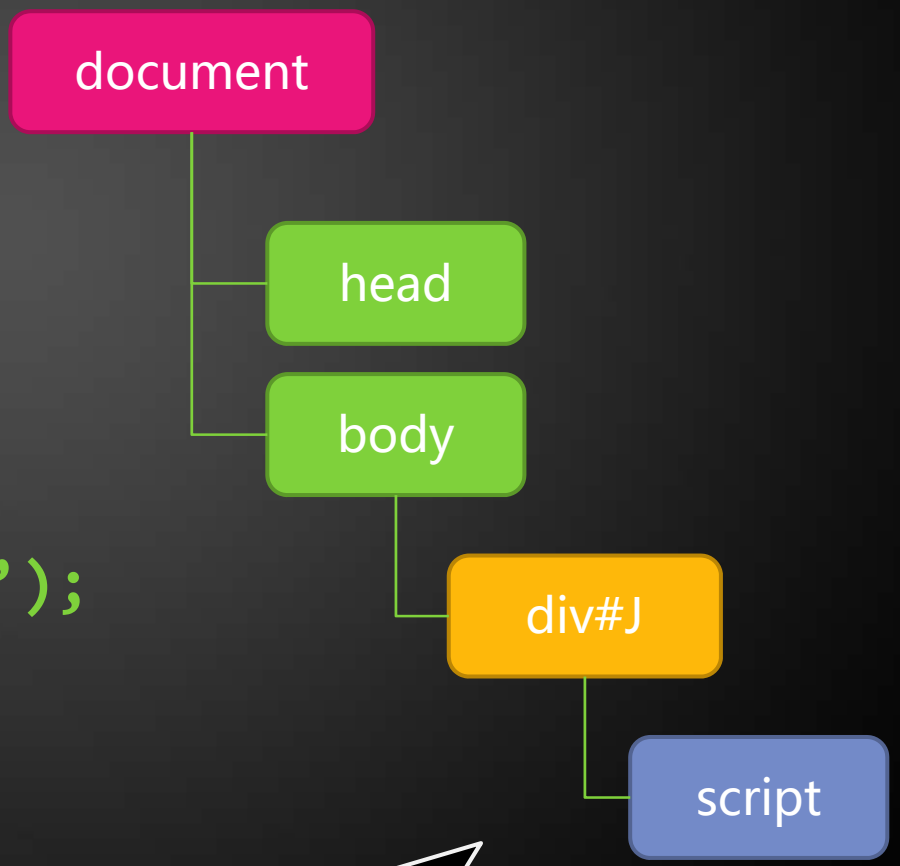
快速响应的用户界面

Web性能——快速响应的用户界面

Js执行和浏览器渲染

Web性能——快速响应的用户界面

```
<!DOCTYPE HTML>
<html lang="zh">
  <head>head</head>
<body>
  <div id=“J”>
      <script>
      $.write(‘helloworld’);
      </script>
  </div>
…
</body>
</html>
```

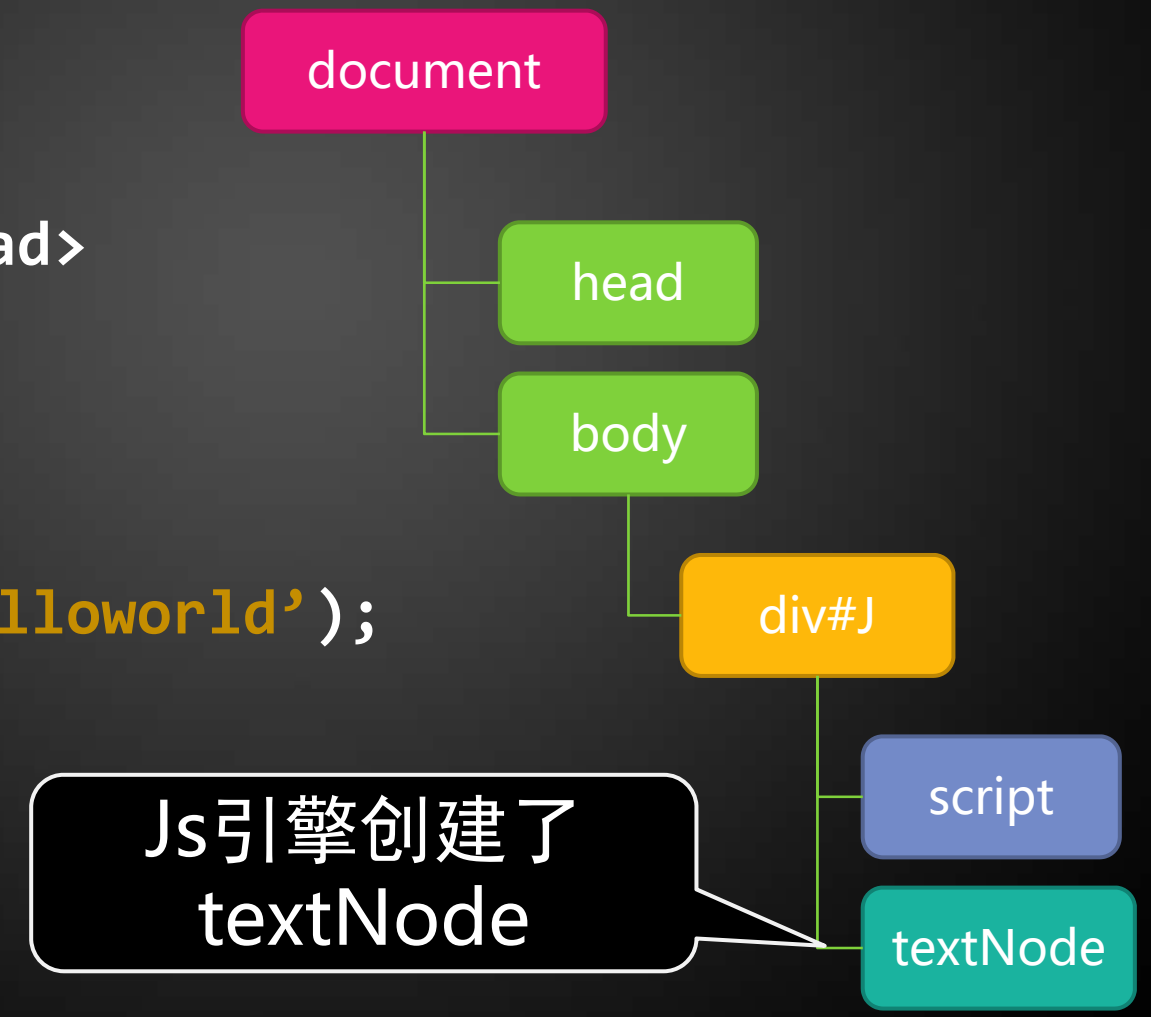


渲染出Script 节点

浏览器暂停渲染HTML
将script交由js引擎编译执行

Web性能——快速响应的用户界面

```
<!DOCTYPE HTML>
<html lang="zh">
  <head>head</head>
<body>
  <div id="J">
      <script>
      $.write('helloworld');
      </script>
  </div>
…
</body>
</html>
```



Js引擎执行代码段结束
将渲染主动权交给浏览器继续渲染HTML

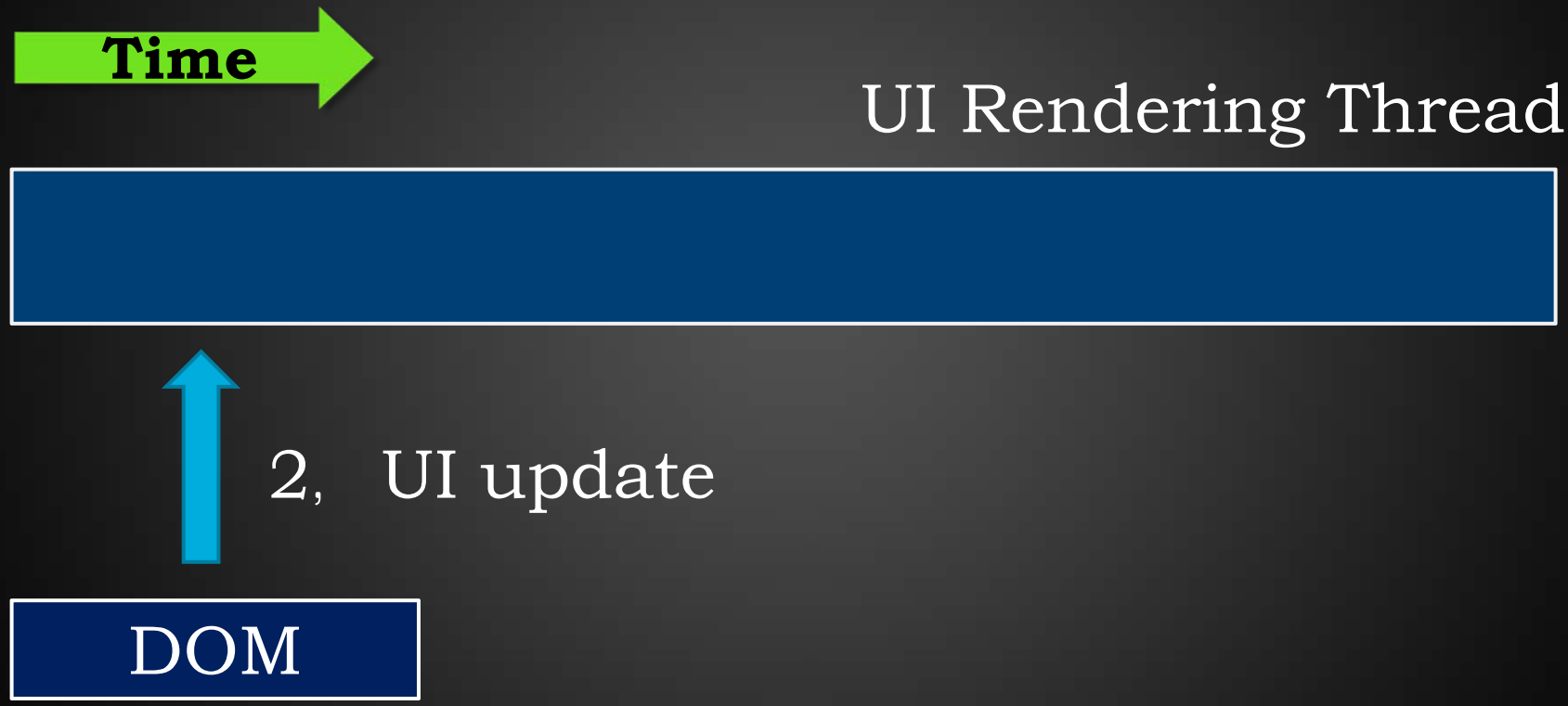
Web性能——快速响应的用户界面

阻塞：

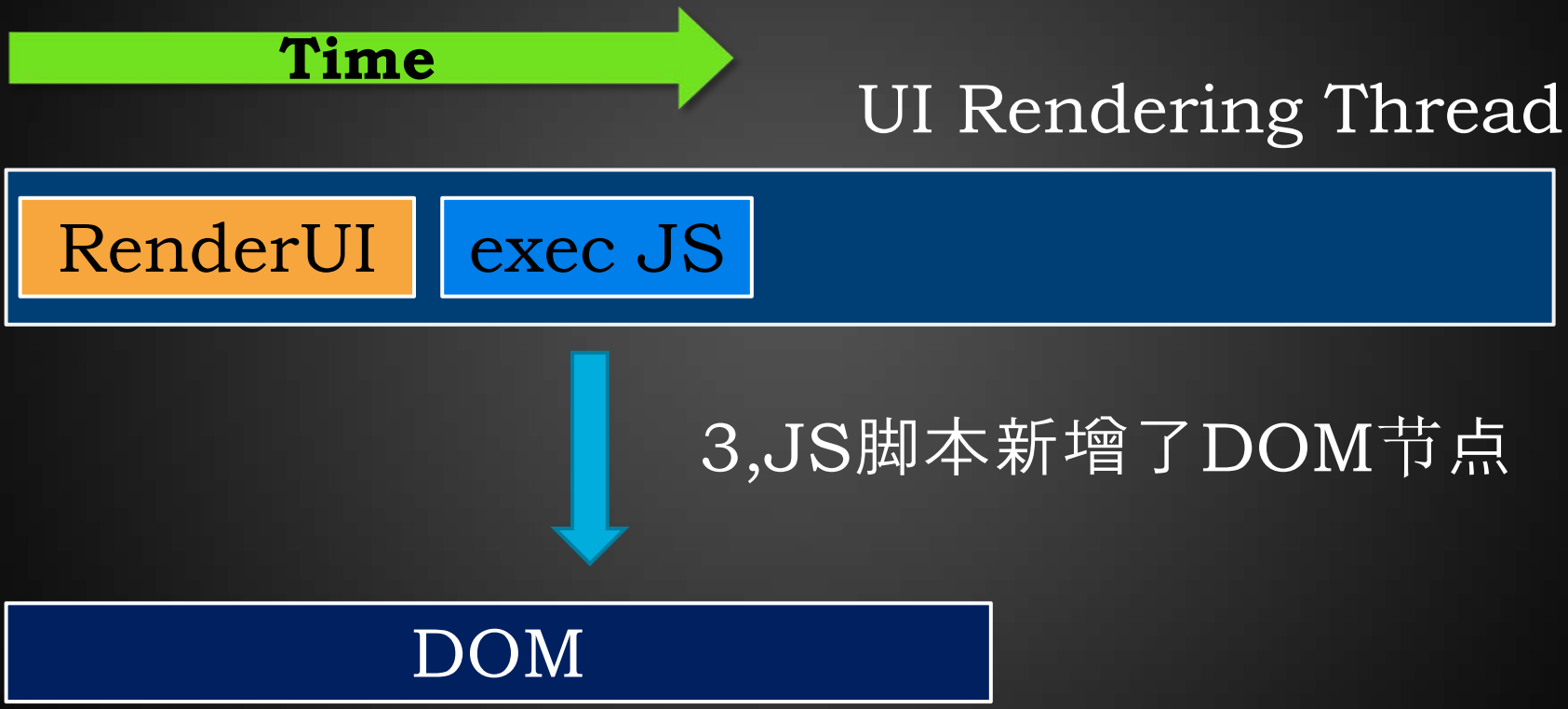
Js的执行会中断HTML的渲染

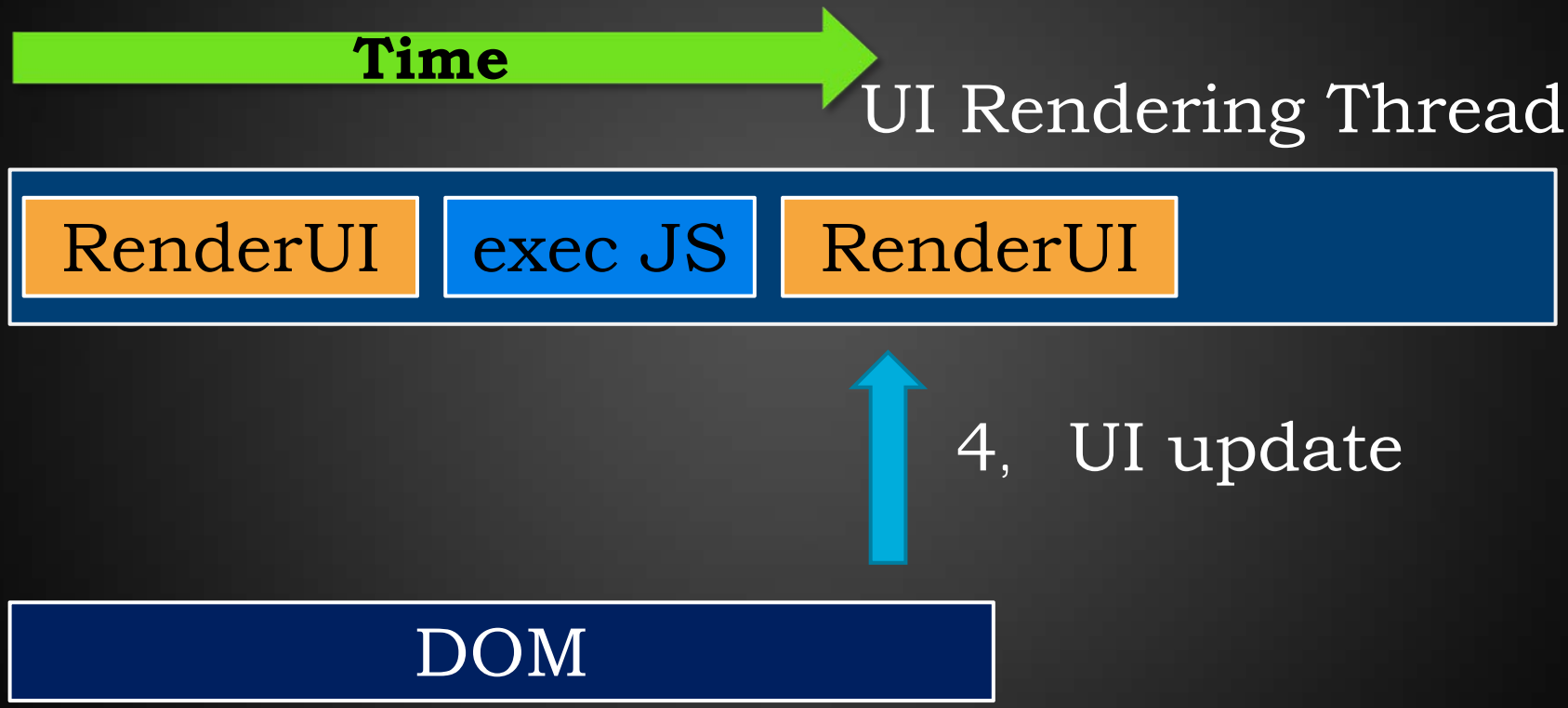
Web性能——快速响应的用户界面

单线程的UI渲染


Time
UI Rendering Thread
2, UI update
DOM
1, 构建出DOM


Time
UI Rendering Thread
RenderUI
渲染出此时的Dom
DOM


Time
UI Rendering Thread
RenderUI
exec JS
3,JS脚本新增了DOM节点
DOM


Time
UI Rendering Thread
RenderUI
exec JS
RenderUI
4, UI update
DOM

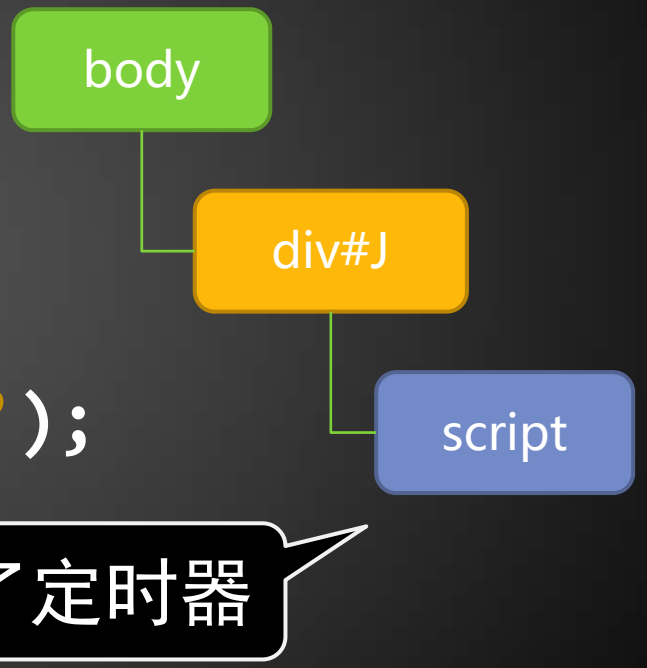
Web性能——快速响应的用户界面

避免阻塞：同样性能攸关的大事

（异步执行JavaScript）

Web性能——快速响应的用户界面

```
…
<div id="J">
  <script>
  setTimeout(function(){
     $.write('helloworld');
  },100);
  </script>
</div>
<div>doc</div>
</body>
</html>
```

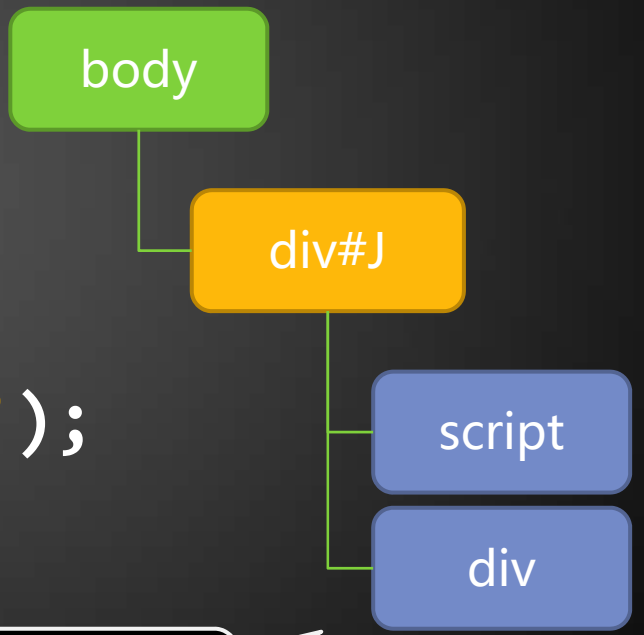


Js引擎只启动了定时器，没有”write”操作

浏览器可以很快获得DOM渲染权，继续渲染HTML

Web性能——快速响应的用户界面

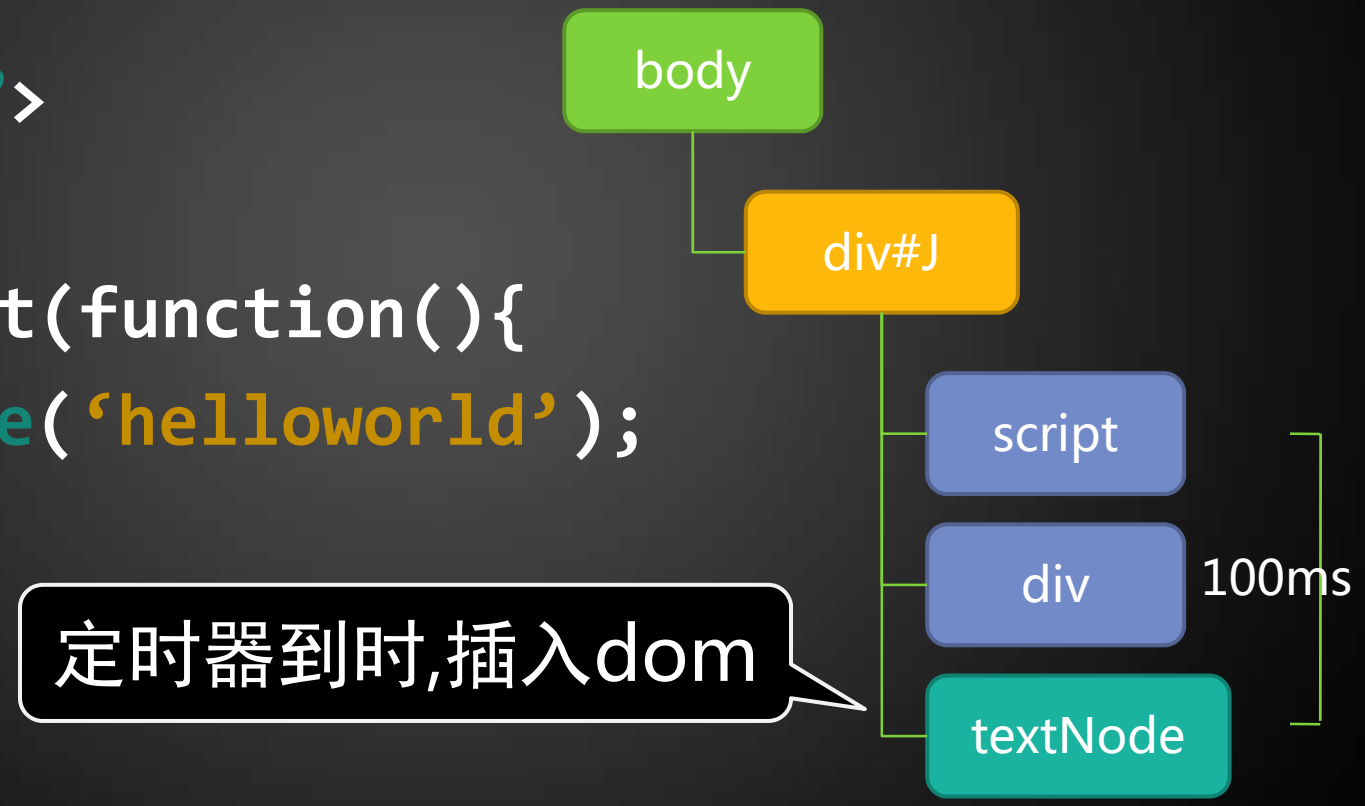
```
…
<div id="J">
  <script>
  setTimeout(function(){
     $.write('helloworld');
  },100);
  </script>
</div>
<div>doc</div>
</body>
</html>
```



浏览器继续渲染html

Web性能——快速响应的用户界面

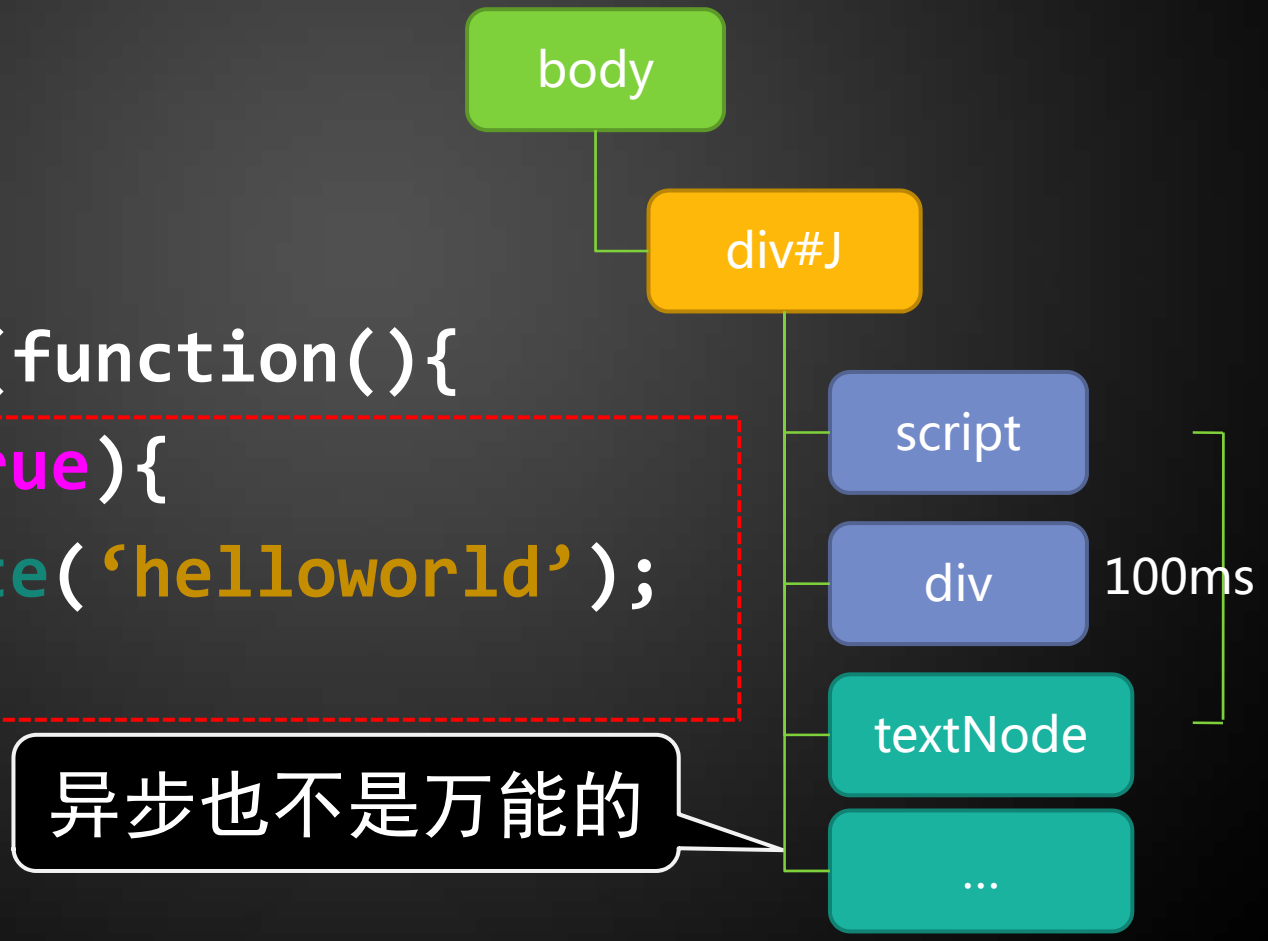
```
…
<div id="J">
  <script>
  setTimeout(function(){
     $.write('helloworld');
  },100);
  </script>
</div>
<div>doc</div>
</body>
</html>
```



从定时器开启，到write节点完成
中间的HTML渲染没有中断

Web性能——快速响应的用户界面

```
…
<div id=“J”>
  <script>
  setTimeout(function(){
    while(true){
      $.write(‘helloworld’);
    }
  },100);
  </script>
</div>
…
```



Js中低效的Dom操作依然会阻断浏览器的渲染，让浏览器看起来像“冷冻”住

Web性能——快速响应的用户界面

你有没有想过？

- 在运行大型复杂的JavaScript脚本的时候不会发生浏览器假死？
- JavaScript可以在后台运行？
- JavaScript函数甚至可以在多个进程中同时运行？

Web性能——快速响应的用户界面

Web性能——快速响应的用户界面

Web Workers的特点

- 运行于浏览器UI线程之外
- 已被firefox、chrome、safari原生支持
- Worker之间不会互相影响

Web性能——快速响应的用户界面

Worker运行环境由以下部分组成：

- navigator对象，appName\appVersion\user Agent\platform
- Location对象 == window.location，但只读
- Self对象，指向全局worker对象
- importScripts()对象，加载外部JS文件
- 所有ECMAScript对象，如：Object/Array/Date 等
- XMLHttpRequest构造器
- setTimeout()和setInterval()方法
- close()方法，立刻停止Worker运行。

Web性能——快速响应的用户界面

与Worker通信

消息系统是网页和Worker通信的唯一途径

postMessage()只能传递：

- 字符串、数字、布尔值、null和undefined
- Object 和Array

(目前safari只支持字符串)

Web性能——快速响应的用户界面

与Worker通信

例

```
//页面中的事件接口
var worker = new Worker('worker.js');
worker.onmessage = function(event){
  alert(event.data);
}
worker.postMessage('1');
```

Web性能——快速响应的用户界面

与Worker通信

例

```
//worker运行代码（worker.js文件）
self.onmessage = function(event){
  var a;
  if(event.data === '1'){
      a = 'result is 1';
  }
  if(event.data === '2'){
      a = 'result is 2';
  }
  self.postMessage(a);
};
```

查看示例

Web性能——快速响应的用户界面

Worker适用于处理

- 编码/ 解码大字符串（如：巨大的json）
- 复杂数学运算（包括图像或视频处理）
- 大数组排序

DOM编程

- UI Update
- 通过DOM事件处理与用户的交互

DOM为什么会慢？

访问和操作DOM是现代web应用的重要部分。但每次穿越链接ECMAScript和DOM两个岛屿之间的桥梁，都会被收取“过桥费”。因此我们要运用一些方法来减少DOM编程带来的性能损失。

Web性能——DOM编程

UI Update

- 重绘Repaint
- 重排Reflow

```
<div id="J">
  <script>
  $('J').css('color','red');
  </script>
</div>
```

重绘Repaint

Repaint:

- 透明度更改
- 文字颜色变化
- 背景颜色变化
- 背景图片替换

Reflow:

- 页面渲染过程中
- Dom结构变化
- 浏览器窗口大小改变
- 布局变化

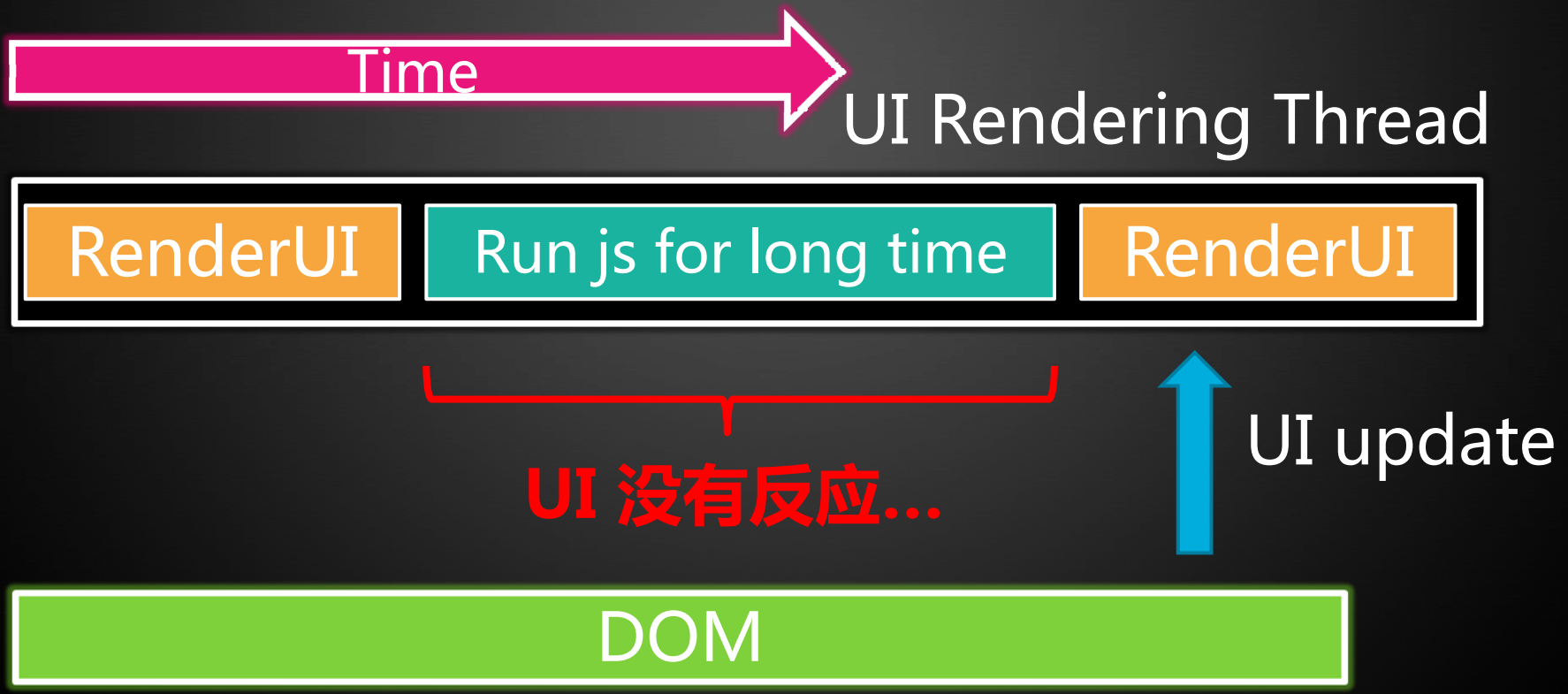
Web性能——DOM编程

减少Reflow/paint：性能攸关的大事！

Web性能——DOM编程

不好的做法

Web性能——DOM编程


Time
UI Rendering Thread
RenderUI
Run js for long time
RenderUI
UI 没有反应...
UI update
DOM

Web性能——DOM编程

例

好的做法

Web性能——DOM编程

使用事件代理

1. 避免重复绑定
2. 减少事件绑定的处理时间
3. 减少内存占用
4. 对新插入的节点不用再次绑定

Web性能——DOM编程

例

```
E.on(document, 'click', function(e){
  var el = e.target;
  switch (el.tagName.toLowerCase()){
      case : 'a'
            //do something
            break
      case : 'button'
            //do something
            break
      defalut:
            ......
  }
})
```

使用框架的事件代理（KISSY）

Ajax性能

Web性能——Ajax性能

- 两种主流的请求方式
- 两种发送数据技术
- 数据格式
- Ajax性能指南

Web性能——Ajax性能

两种主流的请求方式

- XMLHttpRequest(XHR)
- Dynamic script tag insertion 动态脚本注入(JSON-P)

Web性能——Ajax性能

提高XMLHttpRequest(XHR)性能

一. 通过监听readyState 提高性能

例

```
readyState == 0 //尚未加载
readyState == 1 //正在加载
readyState == 2 //加载完毕
readyState == 3 //正在处理
readyState == 4 //处理完毕
```

通过监听readyState值等于3，说明此时正在与服务器交互，响应信息还在传输中。这就是传说中的“流”（streaming），它是提升数据请求性能强大的工具

Web性能——Ajax性能

提高XMLHttpRequest(XHR)性能

一. 通过监听readyState 提高性能

例

```
req.onreadystatechange = function(){
        if(req.readyState === 3){
                //接收到部分信息，但不是所有
                var dataSoFar = req.responseText;
                ...do someting
        }
        else if(req.readyState === 4){
                //所有信息接收完毕
                var data = req.responseText;
                ...do someting
        }
}
```

Web性能——Ajax性能

提高XMLHttpRequest(XHR)性能

一. 通过监听Http状态码 status 提高性能

例

```
status == 200 //请求成功
status == 202 //请求被接受但处理未完成
status == 400 //错误请求
status == 404 //请求资源未找到
status == 500 //内部服务器错误
```

如果遇到比较大的数据时，监听status，客户端可提前响应，避免用户等待解析数据的时间

Web性能——Ajax性能

两种发送数据技术

- XHR（XMLHttpRequest）
- 信标（Beacons）

Web性能——Ajax性能

XMLHttpRequest

- 使用GET或POST方式
- GET只发送 一个数据包，速度更快
- POST（头信息、正文）两个数据包，更适合发送大量的数据
- IE对URL长度有限制（2083字节）

Web性能——Ajax性能

信标（Beacons）

- 使用JS创建一个新Image对象，并把src设置为服务器脚本的URL
- 简单、性能消耗小，长度被限制，且无法发送POST数据
- 监听image的load事件来获知服务器响应

例

```
var url = '/a.php';
var params = ['step=2', 'time=1238027314'];
var beacon = new Image();
beacon.src = url + '?'+ params.join('&');
```

Web性能——Ajax性能

信标（Beacons）

例

```
//使用信标处理服务器返回状态
beacon.onload = function () {
        if(this.width == 1){
                //成功
        }
        else if(this.width == 2){
                //失败
        }
}
```

Web性能——Ajax性能

数据格式

- XML
- JSON
- JSON-P
- HTML
- 自定义

Web性能——Ajax性能

XML

优点:

•极佳的通用性、格式严格、易验证

缺点:

•极冗长.每个单独的数据片断都依赖大量结构,所以有效数据的比例非常低.

•语法模糊,当把一个数据结构转化为XML时,可以把对象参数放到对象元素的属性中,也可放在独立的子元素中,可以使用描述清晰的长标签名,也可以使用高效但难以辨认的短标签名.

•语法的解析过程含混,必须提前知道XML响应的布局

Web性能——Ajax性能

JSON

- JSON是一种使用JS对象和数组直接量编写的轻量级且易于解析的数据格式.
- 可以直接使用eval()来解析JSON字符串.但是在代码中使用eval是很危险的,特别是用它执行第三方的json数据.
- 尽可能使用JSON.parse()方法解析字符串本身.该以捕获JSON中的词法错误,并允许传入一个函数用来过滤或转换解析结果

Web性能——Ajax性能

JSON-P

- 使用动态脚本注入获取，把数据当做可执行javascript解析
- 解析速度快，可跨域
- 涉及敏感数据时不应该用它

Web性能——Ajax性能

HTML

- 服务器端生成HTML返回给客户端,JS使用innerHTML属性把它插入页面相应的位置.
- 在客户端的瓶颈是CPU而不是带宽时才使用此技术.

Web性能——Ajax性能

自定义格式

- 用数据中不会存在的单字符做分隔.解析时只需要用split()传入字符串或正则表达式进行.
- var rows=req.responseText.split(/\u0001/);

//正则表达式,IE中的split()会忽略紧挨着的两个分隔符中的第二个分隔符.\u0001是Unicode表示法.

- var rows=req.responseText.split("\u0001");

//字符串,更为保险

- 对于非常大的数据集,它是最快的格式.需要在很短的时间内向客户端传送大量数据时可以考虑使用此格式.

Web性能——Ajax性能

Ajax性能指南

缓存数据
最快的ajax请求就是没有请求，有两种方法可以避免发送不必要的请求：

1. 在服务端，设置HTTP头信息以确保你的响应会被浏览器缓存（使用简单，好维护）
2. 在客户端，把获取到的信息存储到本地，从而避免再次请求。（给你最大的控制权）

Web性能——Ajax性能

1. 设置HTTP头信息

如果希望ajax响应能被浏览器缓存,必须使用GET方式发出请求.并且还需要在响应中告诉浏览器应该缓存多久.

一个Expires头信息格式如下:

```
Expires:Mon,28 Jul 2014 23:30:00 GMT
```

告诉浏览器缓存此响应到2014年7月。

Web性能——Ajax性能

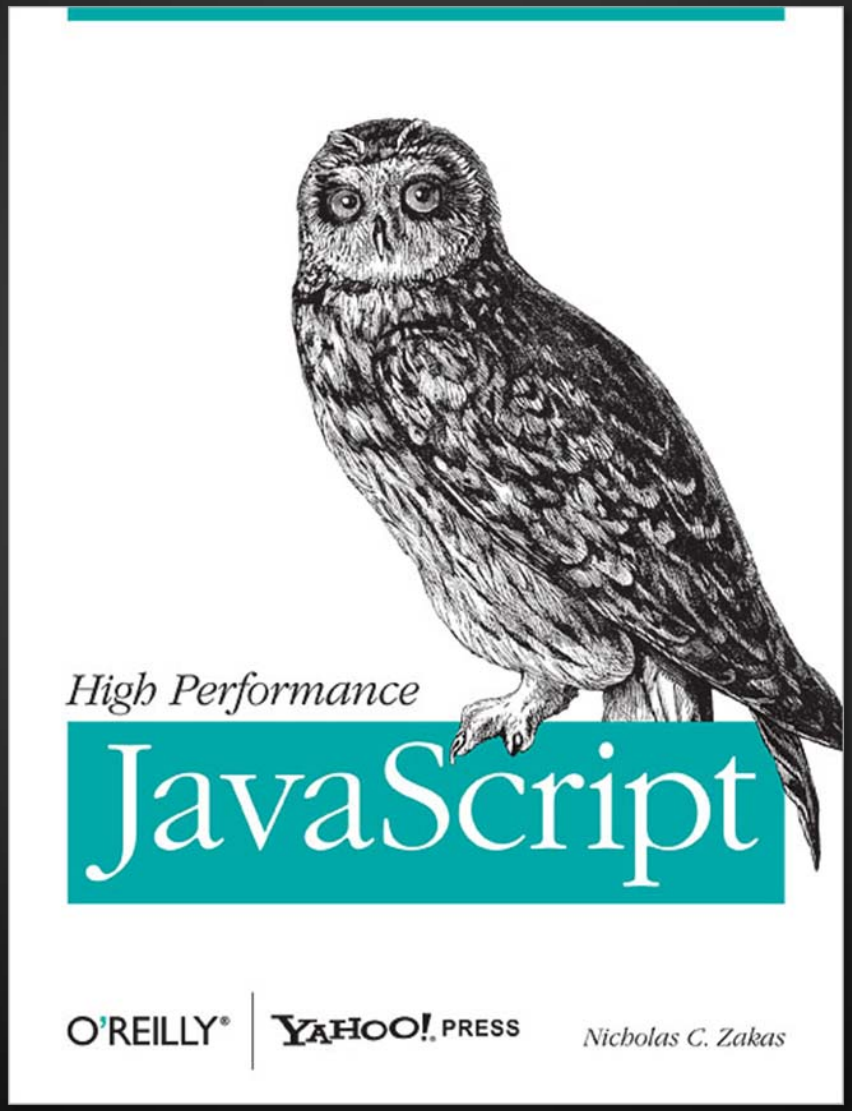
2. 本地数据存储

把数据从服务器接收后储存起来,把响应文本保存到一个对象中,以URL为键值作为索引.

例

```
var localCache = {
      url : req.responseText
}
```

每次发请求前，先检查url是否访问过，如果访问过，就从本地缓存中读取。



高性能网站建设(进阶)指南，高性能JavaScript

http://book.douban.com/subject/5362856/


THANK YOU