# 浏览器原理

## 浏览器预备知识

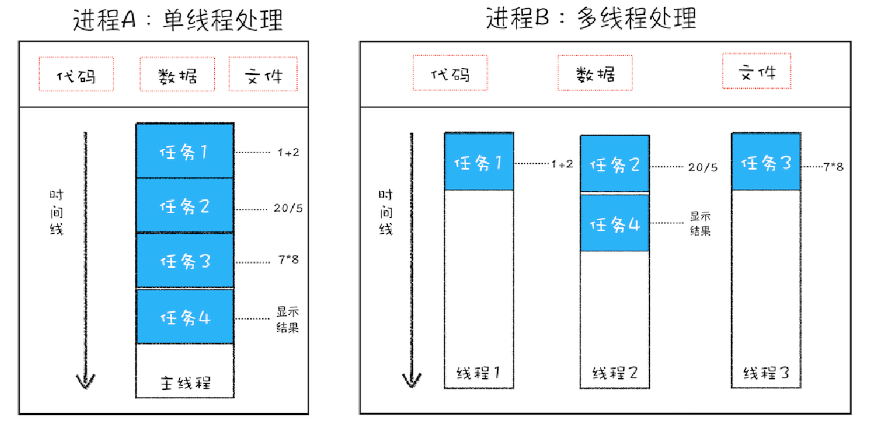
### 进程和线程

多线程可以并行处理任务，但是线程是不能单独存在的，它是由进程来启动和管理的

一个进程就是一个程序的运行实例。详细解释就是，启动一个程序的时候，操作系统会为该

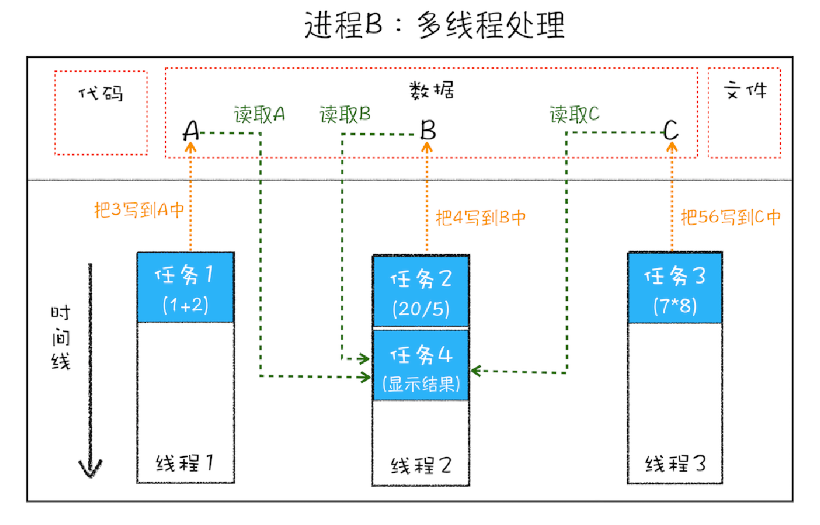
程序创建一块内存，用来存放代码、运行中的数据和一个执行任务的主线程，我们把这样的

一个运行环境叫进程。



**总结：**

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。**



1. **当一个进程关闭之后，操作系统会回收进程所占用的内存。**
2. **进程之间的内容相互隔离。**

### 并行处理

计算机中的并行处理就是同一时刻处理多个任务，

## 早期浏览器介绍

### 单进程浏览器时代

单进程浏览

器是指浏览器的所有功能模块都是运行在同一个进程里，这些模块包含了网络、插件、

JavaScript运行环境、渲染引擎和页面等。其实早在2007年之前，市面上浏览器都是单进程

的。单进程浏览器的架构如下图所示：



如此多的功能模块运行在一个进程里，是导致单进程浏览器不稳定、不流畅和不安全的一个

主要因素

### 缺点

#### 不稳定

早期浏览器需要借助于插件来实现诸如Web视频、Web游戏等各种强大的功能，但是插件是最容易出问题的模块，并且还运行在浏览器进程之中，所以一个插件的意外崩溃会引起整个浏览器的崩溃。

除了插件之外，渲染引擎模块也是不稳定的，通常一些复杂的JavaScript代码就有可能引起渲染引擎模块的崩溃。和插件一样，渲染引擎的崩溃也会导致整个浏览器的崩溃。

#### 不流畅

所有页面的渲染模块、JavaScript执行环境以及插件都是运行在同一个线程中的，这就意味着同一时刻只能有一个模块可以执行

**脚本**或者**插件**会让单进程浏览器变卡顿外，页面的**内存泄漏**也是单进程变慢的一个

重要原因。通常浏览器的内核都是非常复杂的，运行一个复杂点的页面再关闭页面，会存在

内存不能完全回收的情况，这样导致的问题是使用时间越长，内存占用越高，浏览器会变得

越慢。

#### 不安全

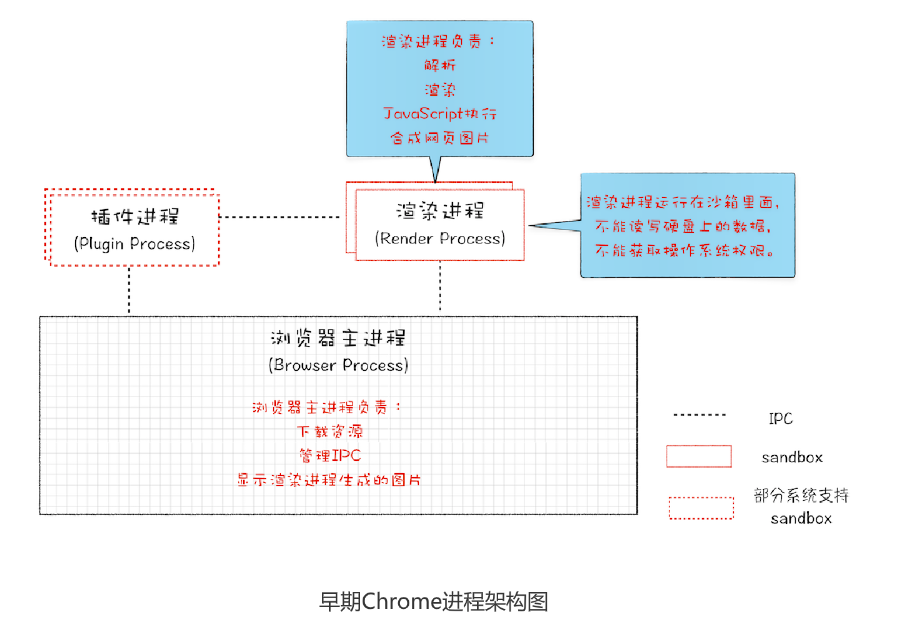
插件可以使用C/C++等代码编写，通过插件可以获取到操作系统的任意资源，当你在页面运

行一个插件时也就意味着这个插件能完全操作你的电脑。如果是个恶意插件，那么它就可以

释放病毒、窃取你的账号密码，引发安全性问题。

## 多进程浏览器时代

### 早期多进程架构



解决了单进程浏览器的缺点

**我们先看看如何解决不稳定的问题**。由于进程是相互隔离的，所以当一个页面或者插件崩溃时，影响到的仅仅是当前的页面进程或者插件进程，并不会影响到浏览器和其他页面，这就完美地解决了页面或者插件的崩溃会导致整个浏览器崩溃，也就是不稳定的问题。

**接下来再来看看不流畅的问题是如何解决的**。同样，JavaScript也是运行在渲染进程中的，所以即使JavaScript阻塞了渲染进程，影响到的也只是当前的渲染页面，而并不会影响浏览器和其他页面，因为其他页面的脚本是运行在它们自己的渲染进程中的。所以当我们再在Chrome中运行上面那个死循环的脚本时，没有响应的仅仅是当前的页面。对于内存泄漏的解决方法那就更简单了，因为当关闭一个页面时，整个渲染进程也会被关闭，之后该进程所占用的内存都会被系统回收，这样就轻松解决了浏览器页面的内存泄漏问题。

**最后我们再来看看上面的两个安全问题是怎么解决的**。采用多进程架构的额外好处是可以使用安全沙箱，你可以把沙箱看成是操作系统给进程上了一把锁，沙箱里面的程序可以运行，

但是不能在你的硬盘上写入任何数据，也不能在敏感位置读取任何数据，例如你的文档和桌

面。Chrome把插件进程和渲染进程锁在沙箱里面，这样即使在渲染进程或者插件进程里面执行了恶意程序，恶意程序也无法突破沙箱去获取系统权限。

### 现在的多进程架构



#### 各进程的主要功能：

**浏览器进程**。主要负责界面显示、用户交互、子进程管理，同时提供存储等功能。

**渲染进程**。核心任务是将 HTML、CSS 和 JavaScript 转换为用户可以与之交互的网页，排

版引擎Blink和JavaScript引擎V8都是运行在该进程中，默认情况下，Chrome会为每个Tab

标签创建一个渲染进程。出于安全考虑，渲染进程都是运行在沙箱模式下。

**GPU进程**。其实，Chrome刚开始发布的时候是没有GPU进程的。而GPU的使用初衷是为

了实现3D CSS的效果，只是随后网页、Chrome的UI界面都选择采用GPU来绘制，这使得

GPU成为浏览器普遍的需求。最后，Chrome在其多进程架构上也引入了GPU进程。

**网络进程**。主要负责页面的网络资源加载，之前是作为一个模块运行在浏览器进程里面

的，直至最近才独立出来，成为一个单独的进程。

**插件进程**。主要是负责插件的运行，因插件易崩溃，所以需要通过插件进程来隔离，以保

证插件进程崩溃不会对浏览器和页面造成影响。

#### 多进程缺点

不过凡事都有两面性，虽然多进程模型提升了浏览器的稳定性、流畅性和安全性，但同样不

可避免地带来了一些问题：

**更高的资源占用**。因为每个进程都会包含公共基础结构的副本（如JavaScript运行环境），

这就意味着浏览器会消耗更多的内存资源。

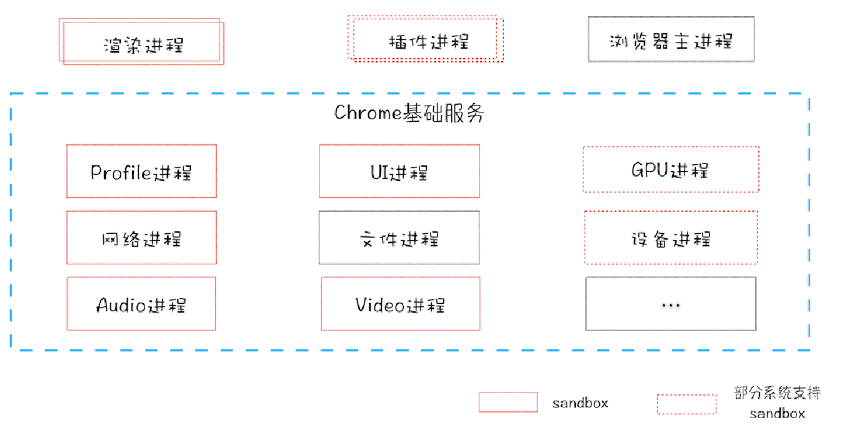
**更复杂的体系架构**。浏览器各模块之间耦合性高、扩展性差等问题，会导致现在的架构已

经很难适应新的需求了。

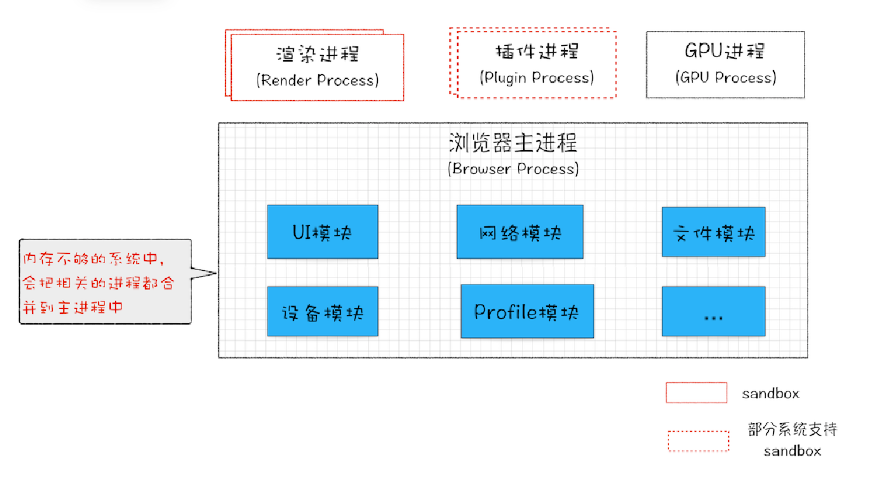
## 未来浏览器架构

在2016年，Chrome官方团队使用“**面向服务的架构**”（ServicesOriented Architecture，简称SOA）的思想设计了新的Chrome架构。也就是说 Chrome 整体架构会朝向现代操作系统所采用的“面向服务的架构” 方向发展，原来的各种模块会被重构成独立的服务（Service），每个服务（Service）都可以在独立的进程中运行，访问服务（Service）必须使用定义好的接口，通过IPC来通信，从而**构建一个更内聚、松耦合、易于维护和扩展的系统**，更好实现 Chrome 简单、稳定、高速、安全的目标。如果你对面向服务的架构感兴趣，你可以去网上搜索下资料，这里就不过多介绍了。

Chrome最终要把UI、数据库、文件、设备、网络等模块重构为基础服务，类似操作系统底层服务，下面是Chrome“面向服务的架构”的进程模型图：



同时Chrome还提供灵活的弹性架构，在强大性能设备上会以多进程的方式运行基础服务，但是如果在资源受限的设备上（如下图），Chrome会将很多服务整合到一个进程中，从而节省内存占用。



## Tcp 协议

互联网，实际上是一套理念和协议组成的体系架构

协议是一套众所周知的规则和标准，如果各方都同意使用，那么它们之间的通信将变得毫无障碍。

互联网中的数据是通过数据包来传输的。如果发送的数据很大，那么该数据就会被拆分为很

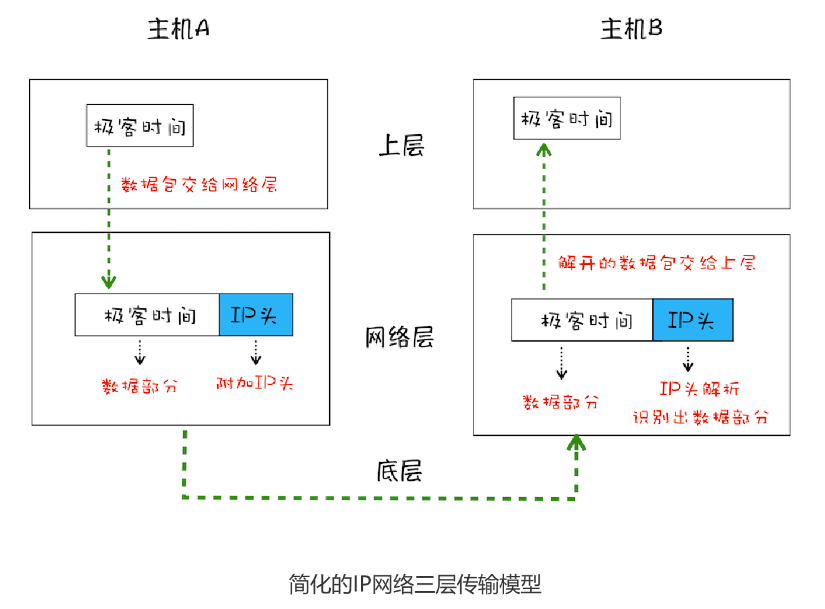
多小数据包来传输。

### IP： 把数据包送达目的主机

数据包要在互联网上进行传输，就要符合网际协议（Internet Protocol，简称IP）标准。

互联网上不同的在线设备都有唯一的地址

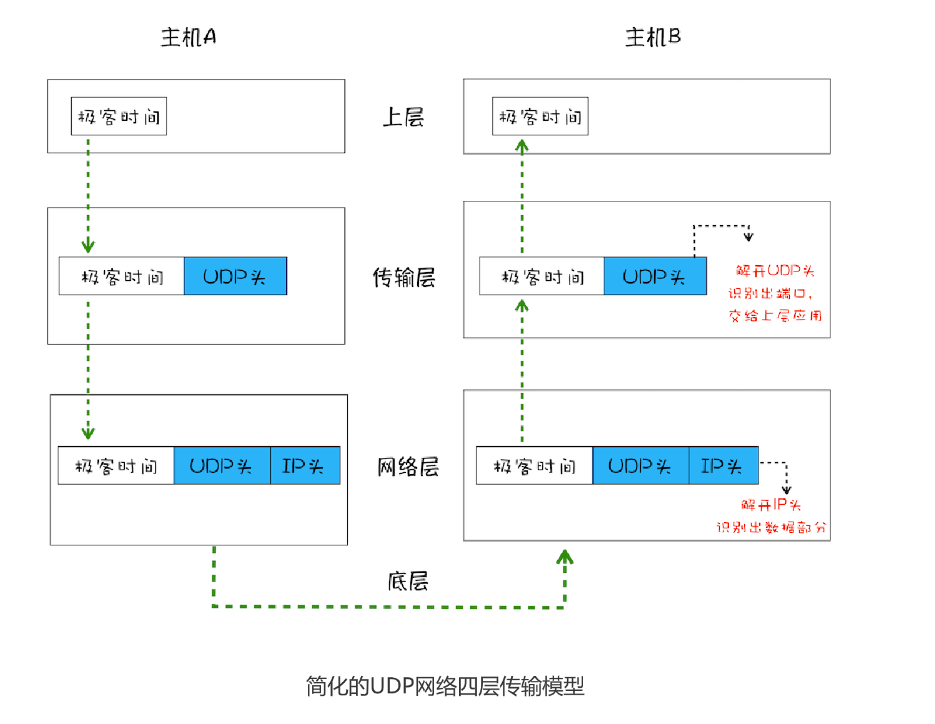
**计算机的地址就称为IP地址，访问任何网站实际上只是你的计算机向另外一台计算机请求信息。**



### UDP： 把数据送达应用程序

IP是非常底层的协议，只负责把数据包传送到对方电脑，但是对方电脑并不知道把数据包交给哪个程序，因此，需要基于IP之上开发能和应用打交道的协议，最常见的是“**用户数据包协议**（User Datagram Protocol）”，简称**UDP**。

UDP中一个最重要的信息是**端口号**，端口号其实就是一个数字，每个想访问网络的程序都需要绑定一个端口号。通过端口号UDP就能把指定的数据包发送给指定的程序了，所以IP通过IP地址信息把数据包发送给指定的电脑，而**UDP通过端口号把数据包分发给正确的程序**。



注意 ：UDP不能保证数据可靠性，但是传输速度却非常快

UDP缺点：

1. 数据包在传输过程中容易丢失；

2、大文件会被拆分成很多小的数据包来传输，这些小的数据包会经过不同的路由，并在不同的时间到达接收端，而UDP协议并不知道如何组装这些数据包，从而把这些数据包还原成完整的文件。

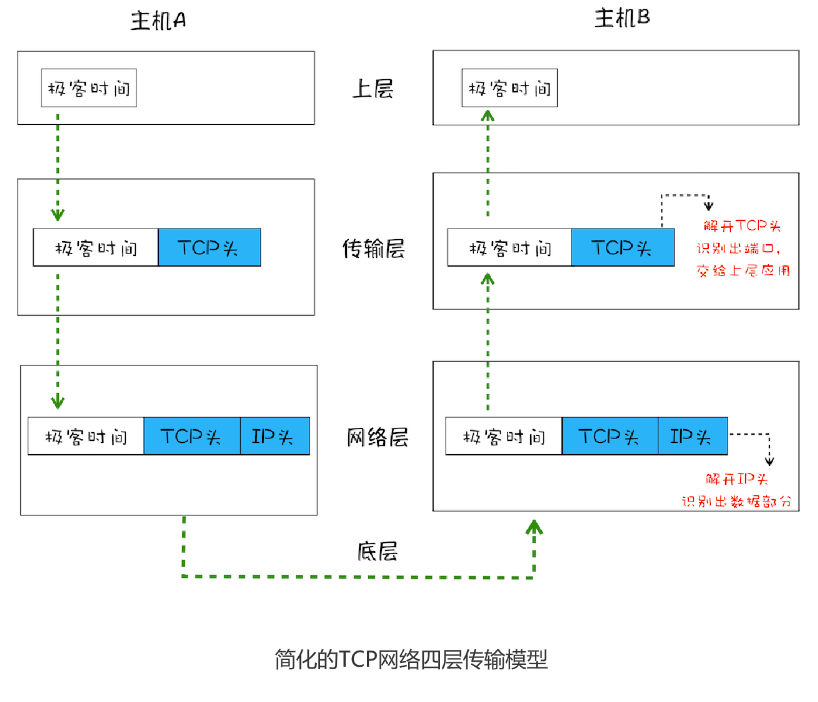
### TCP：把数据完整地送达应用程序

**TCP（Transmission Control Protocol，传输控制协议）是一种面向连接的、可靠的、基于字节流的传输层通信协议。**

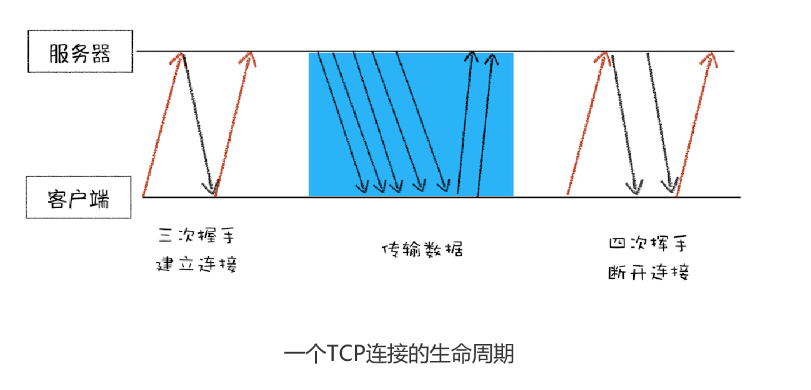
和UDP头一样，TCP头除了包含了目标端口和本机端口号外，还提供了用于排序的序列号，以便接收端通过序号来重排数据包。

**TCP相对UDP优点：**

1. 对于数据包丢失的情况，TCP提供重传机制；
2. TCP引入了数据包排序机制，用来保证把乱序的数据包组合成一个完整的文件。



### TCP连接生命周期：



#### 建立连接

这个阶段是通过“三次握手”来建立客户端和服务器之间的连接。TCP 提供**面向连接**的通信传输。

**面向连接**是指在数据通信开始之前先做好两端之间的准备工作

**三次握手**是指在建立一个TCP连接时，客户端和服务器总共要发送三个数据包

以确认连接的建立。

#### 传输数据

**接收端需要对每个数据包进行确认操作**也就是接收端在接收到数据包之后，需要发送确认数据包给发送端。

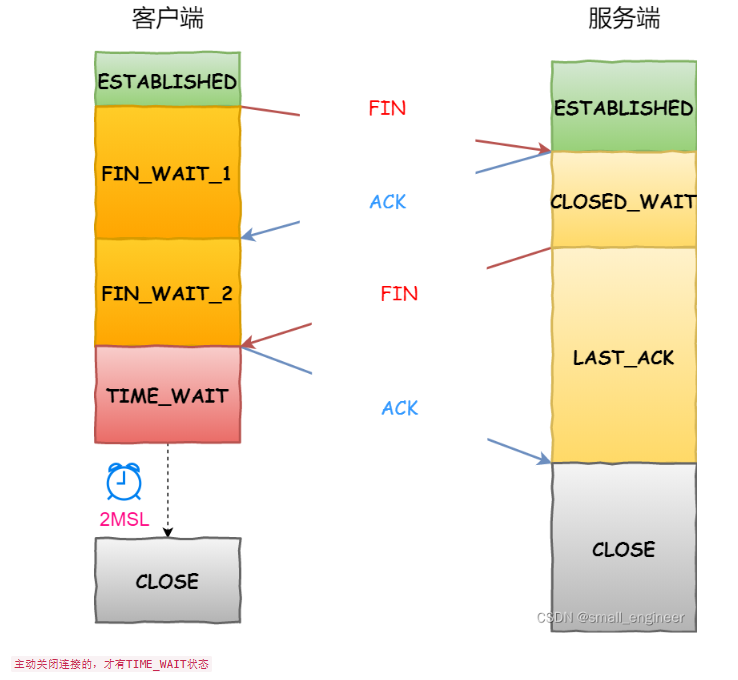
当发送端发送了一个数据包之后，在规定时间内没有接收到接收端反馈的确认消息，则判断为**数据包丢失，并触发发送端的重发机制**

文件在传输过程中会被拆分成很多小的数据包，这些数据包到达接收端后，接收端会按照**TCP头中的序号**为其排序，从而保证组成完整的数据。

#### 断开链接

数据传输完毕之后，就要终止连接了，涉及到最后一个阶段“四次挥手”来保证双方都能断开连接

四次挥手：



## 浏览器端发起HTTP请求流程

### 构建请求

### 查找缓存

**浏览器缓存是一种在本地保存资源副本，以供下次请求时直接使用的技术**。

当浏览器发现请求的资源已经在浏览器缓存中存有副本，它会拦截请求，返回该资源的副

本，并直接结束请求，而不会再去源服务器重新下载。这样做的好处有：

缓解服务器端压力，提升性能（获取资源的耗时更短了）；

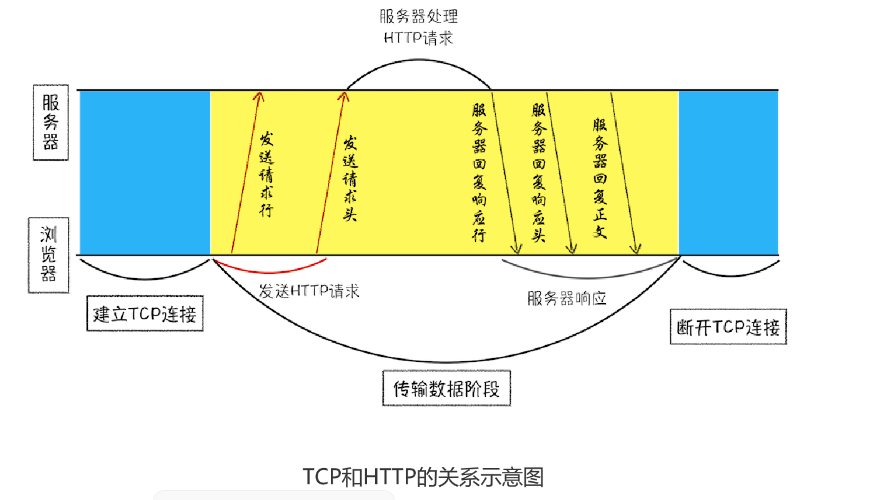
对于网站来说，缓存是实现快速资源加载的重要组成部分。

当然，如果缓存查找失败，就会进入网络请求过程了。

### 准备IP 地址和端口

浏览器使用**HTTP协议作为应用层协议**，用来封装请求的文本信息；

并使用**TCP/IP作传输层协议**将它发到网络上，所以在HTTP工作开始之前，浏览器需要通过TCP与服务器建立连接。也就是说**HTTP的内容是通过TCP的传输数据阶段来实现的**，你可以结合下图更好地理解这二者的关系。



由于IP地址是数字标识，比如极客时间网站的IP是39.106.233.176, 难以记忆，但使用极客时间的域名（time.geekbang.org）就好记多了，所以基于这个需求又出现了一个服务，负责把域名和IP地址做一一映射关系。这套域名映射为IP的系统就叫做“**域名系统**”，简称DNS（DomainName System）。

当然浏览器还提供了**DNS数据缓存服务**，如果某个域名已经解析过了，那么浏览器会缓存解析的结果，以供下次查询时直接使用，这样也会减少一次网络请求。

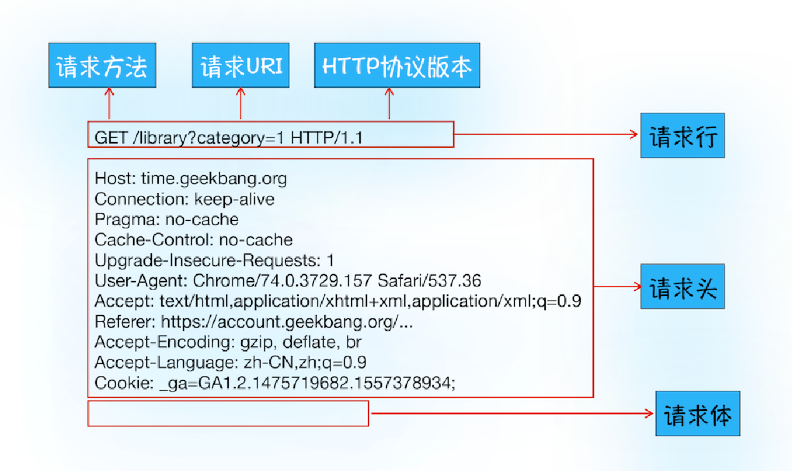
拿到IP之后，接下来就需要获取端口号了。通常情况下，如果URL没有特别指明端口号，那么HTTP协议默认是80端口

### 等待TCP队列

Chrome有个机制，同一个域名同时最多只能建立6个TCP连接，如果在同一个域名下同时有10个请求发生，那么其中4个请求会进入排队等待状态，直至进行中的请求完成。当然，如果当前请求数量少于6，会直接进入下一步，建立TCP连接。

### 建立TCP连接

### 发送HTTP请求



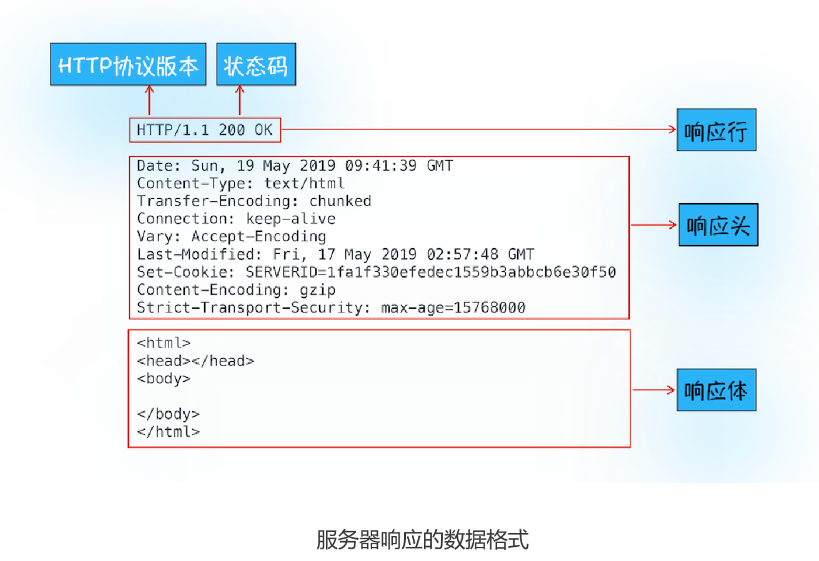
HTTP请求数据格式

首先浏览器会向服务器发送**请求行**，它包括了**请求方法**、**请求URI（Uniform Resource**

**Identifier）**和**HTTP版本协议**。

### 服务器端处理HTTP请求流程

#### 返回请求



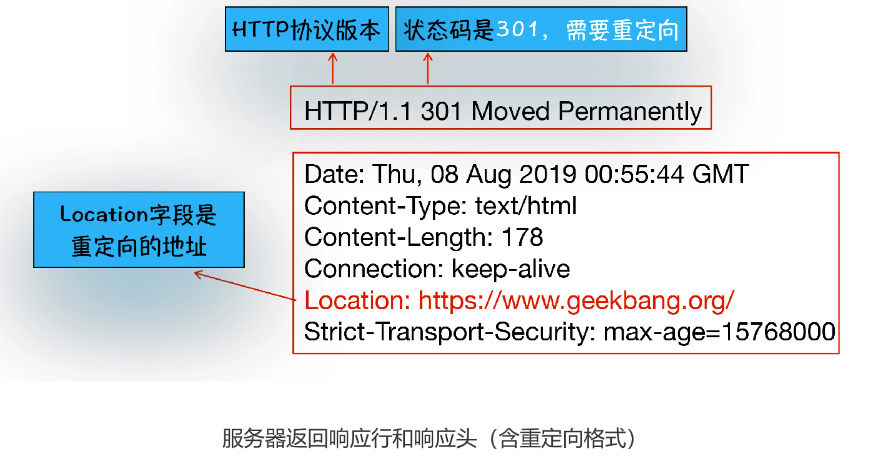
#### 断开连接

通常情况下，一旦服务器向客户端返回了请求数据，它就要关闭 TCP 连接。不过如果浏览器或者服务器在其头信息中加入了：



那么TCP连接在发送后将仍然保持打开状态，这样浏览器就可以继续通过同一个TCP连接发送请求。**保持TCP连接可以省去下次请求时需要建立连接的时间，提升资源加载速度**。

#### 重定向



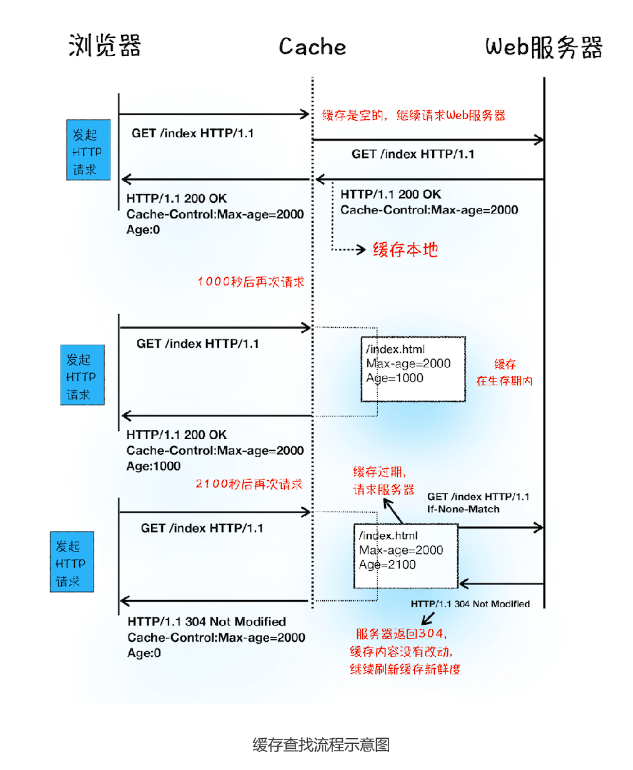
需要重定向的网址正是包含在响应头的Location字段中，接下来，浏览器获取Location字段中的地址，并使用该地址重新导航，这就是一个完整重定向的执行流程。

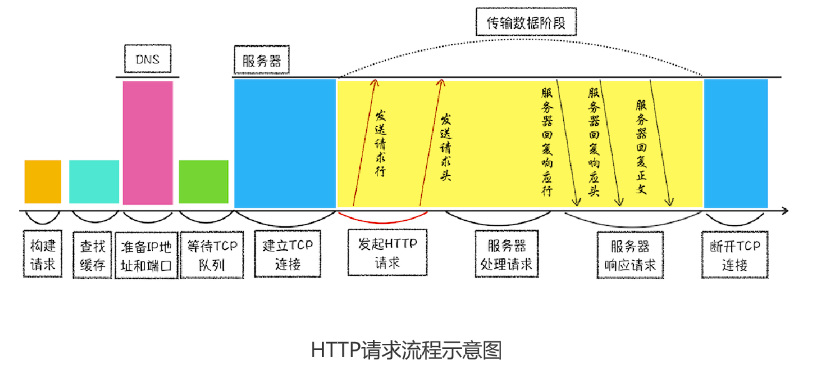
### 第二次打开页面速度更快的原因

#### DNS 缓存

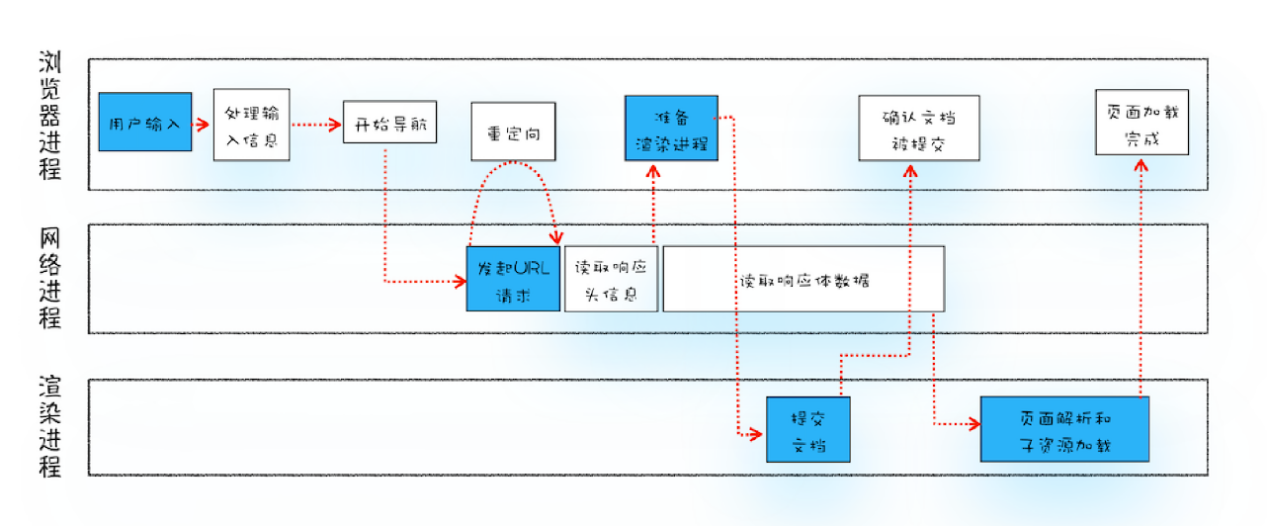
DNS缓存比较简单，它主要就是在浏览器本地把对应的IP和域名关联起来，

#### 页面资源缓存





### 从输入URL 到页面展示的完整流程



注意： **用户发出URL请求到页面开始解析的这个过程，就叫做导航**

## V8 如何执行JavaScript 代码

需要搞懂：**编译器（Compiler）、解释器（Interpreter）、抽象语法树（AST）、字节码（Bytecode）、即时编译器（JIT）**等概念。

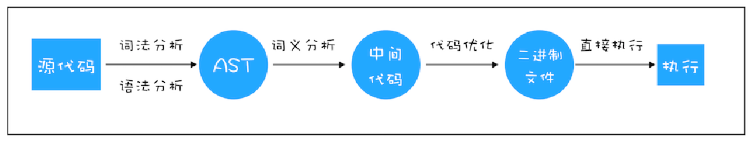
### 编译器和解释器

之所以存在编译器和解释器，是因为机器不能直接理解我们写的代码，所以执行程序之前，需要将我们写的代码“翻译”成机器呢个读懂的机器语言。按照语言执行的流程，可以把语言划分为编译型语言和解释型语言。

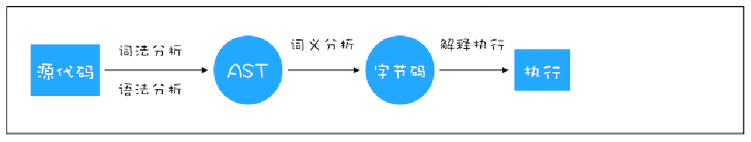
**编译型语言在程序执行之前，需要经过编译器的编译过程，并且编译之后会直接保留机器能读懂的二进制文件，这样每次运行程序时，都可以直接运行该二进制文件，而不需要再次重新编译了**。比如C/C++、GO等都是编译型语言。

**而由解释型语言编写的程序，在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。** 比如Python、JavaScript等都属于解释型语言。

编译器执行过程

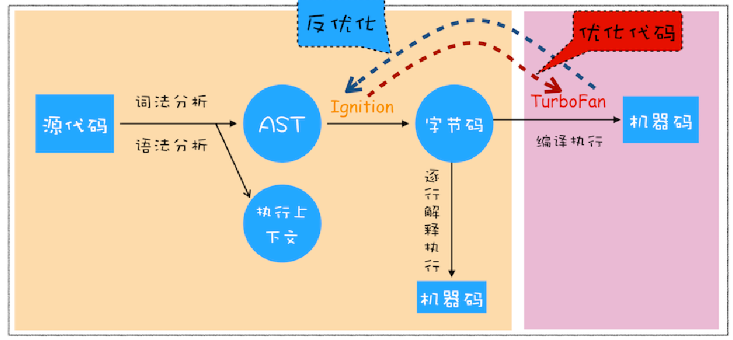


解释器执行过程



### V8 执行JavaScript 代码

执行流程图：



V8 的执行过程中既有**解释器Ignition**， 又有**编译器TurboFan，**

**执行步骤详解：**

1. **生成抽象语法书（AST）和执行上下文**

1/1分词（tokenize），又称为词法分析： 将代码一行行的拆分成token（语法上最小字符或字符串）

1/2 解析（parse）， 又称为语法分析，将上一步的token根据语法规则转为AST。

1. **生成字节码**

解释器（Ignition），根据AST 生成字节码，并解释执行字节码

**拓展**：字节码就是介于AST 和机器码 之间的一种代码，但是与特定类型的机器码无关，字节码需要通过解释器将其转换为机器码后才能执行

1. **执行代码**

## 宏任务和微任务

JavaScript 中，所有的操作可以分为两类：同步任务和异步任务。

宏任务和微任务都是：异步任务

### 宏任务：

宏任务（Macrotask）是指由浏览器或JavaScript引擎在任务队列中排队的异步任务

setTimeout、setInterval、I/O操作、DOM事件等

### 微任务

微任务（Microtask）则是指在当前任务执行完成后立即执行的异步任务

微任务包括Promise回调函数、MutationObserver的回调函数

### 两者之间的联系

微任务和宏任务是绑定的，每个宏任务在执行时，会创建自己的微任务队列。

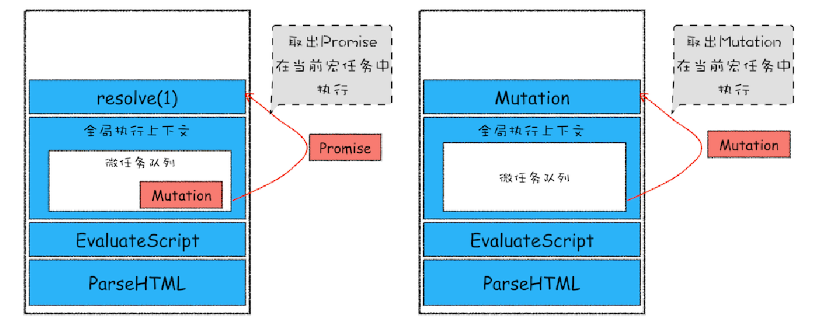
微任务的执行时长会影响到当前宏任务的时长。

在一个宏任务中，分别创建一个用于回调的宏任务和微任务，无论什么情况下，微任务都

早于宏任务执行。

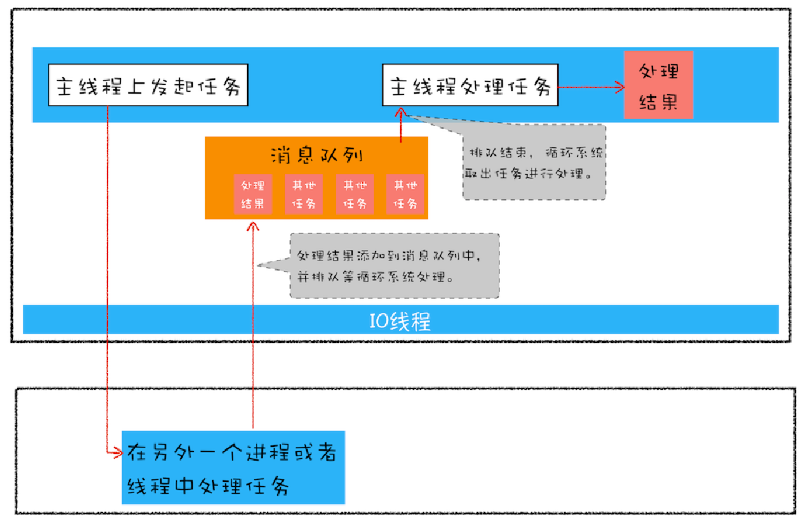
图例：





### Promise详解

#### 异步变成的问题：代码逻辑不连续



Web 应用的异步编程模型

#### 地狱回调问题

**地狱回调**：如果嵌套了太多的回调函数就很容易使得自己陷入了**回调地狱**，不能自拔。

#### Promise 与 微任务

### Async、await ： 使用同步的方法去写异步代码

使用Promise能很好地解决回调地狱的问题，但是这种方式充满了Promise的then()方法，如果处理流程比较复杂的话，那么整段代码将充斥着then，语义化不明显，代码不能很好地表示执行流程

**基于这个原因，ES7 引入了async/await，这是JavaScript异步编程的一个重大改进，提供了在不阻塞主线程的情况下使用同步代码实现异步访问资源的能力，并且使得代码逻辑更加清晰**

#### 生成器 VS 协程

生成器：**生成器函数是一个带星号函数，而且是可以暂停执行和恢复执行的。**



1. 在生成器函数内部执行一段代码，如果遇到yield关键字，那么JavaScript引擎将返回关键字后面的内容给外部，并暂停该函数的执行。
2. 外部函数可以通过next方法恢复函数的执行。

协程：**协程是一种比线程更加轻量级的存在**。

可以把**协程**看成是跑在**线程上的任务**，一个线程上可以存在多个协程，但是在线程上同时只能执行一个协程。

**如果从A协程启动B协程，我们就把A协程称为B协程的父协程**。



## Dom树

### Dom树 如何生成

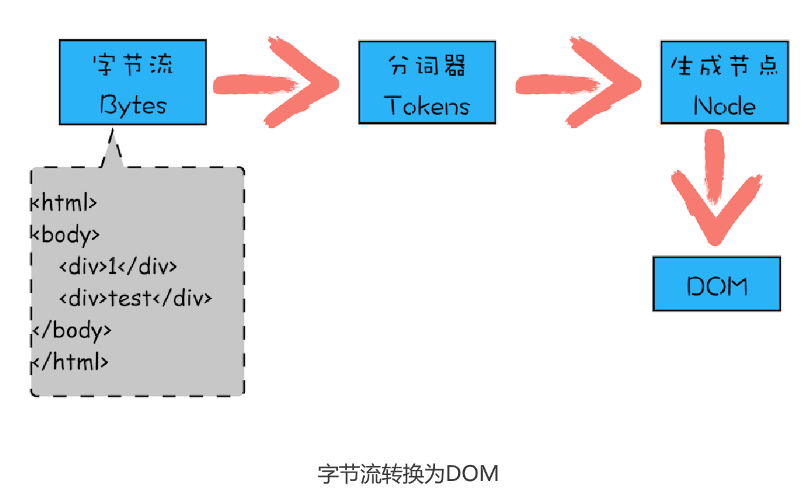
在渲染引擎内部，有**HTML解释器（HTMLParser）**的模块，它的作用就是将HTML 字节流转换为DOM 结构。

**HTML解析器是等整个HTML文档加载完成之后开始解析的，还是随着HTML文档边加载边解析的？**

**HTML解析器并不是等整个文档加载完成之后再解析的，而是网络进程加载了多少数据，HTML解析器便解析多少数据。**

具体执行流程：

1. 网络进程接收响应头，
2. 根据响应头content-type 字段判断文件类型
3. 为该请求选择或者创建一个渲染进程、
4. 网络进程和渲染进程之间建立共享数据通道
5. 网络进程-->渲染进程-->HTML 解析器
6. 将动态接收字节流解析为DOM



#### 第一阶段：通过分词器将字节流转换为Token

V8编译JavaScript过程中的第一步是做词法分析，将JavaScript先分解为一个个Token。解析HTML也是一样的，需要通过分词器先将字节流转换为一个个Token，分为Tag Token和文本Token。上述HTML代码通过词法分析生成的Token如下所示：



**至于后续的第二个和第三个阶段是同步进行的，需要将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。具体的处理规则如下所示：

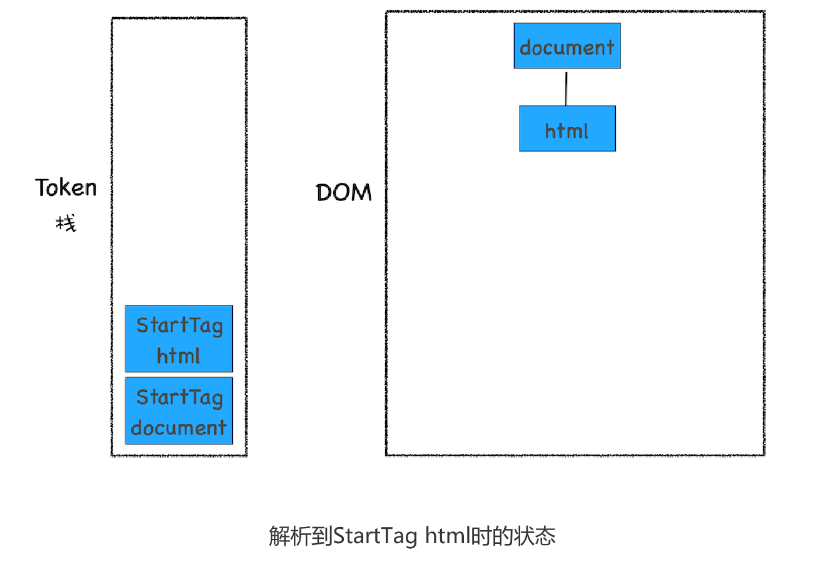
1. 如果压入到占中的是**StartTag Token**, HTML解析器会为该Token创建一个DOM节点，然后将该节点加入到DOM树中，它的父节点就是栈中相邻的那个元素生成的节点。
2. 如果分词器解析出来是**文本Token**，那么会生成一个文本节点，然后将该节点加入到DOM树中，文本Token是不需要压入到栈中，它的父节点就是当前栈顶Token所对应的DOM节点。

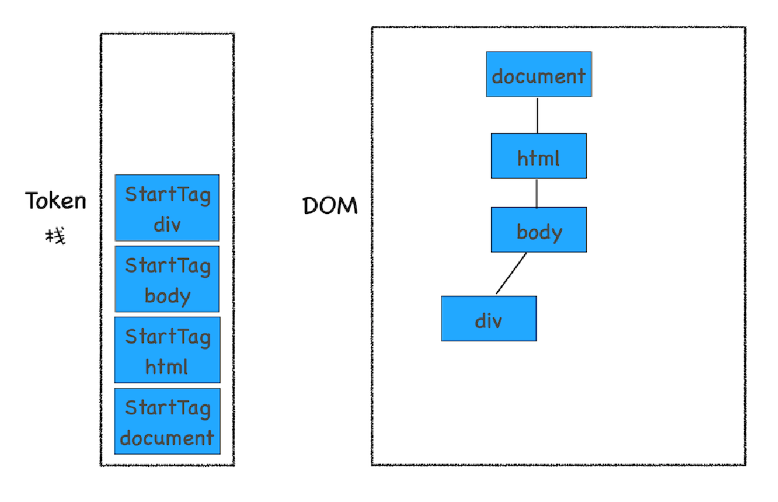
3、如果分词器解析出来的是**EndTag标签**，比如是EndTag div，HTML解析器会查看Token栈顶的元素是否是StarTag div，如果是，就将StartTag div从栈中弹出，表示该div元素解析完成。

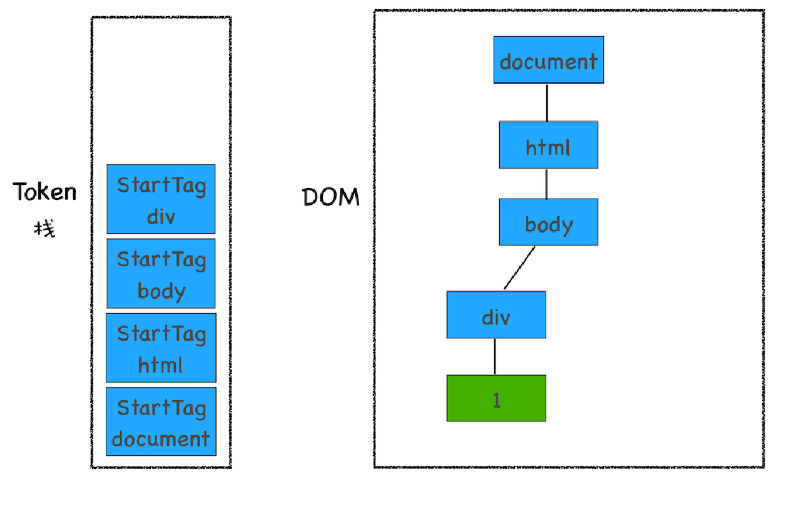
#### 补充：

**HTML解析器开始工作时，会默认创建了一个根为document的空DOM结构**，同时会将一个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

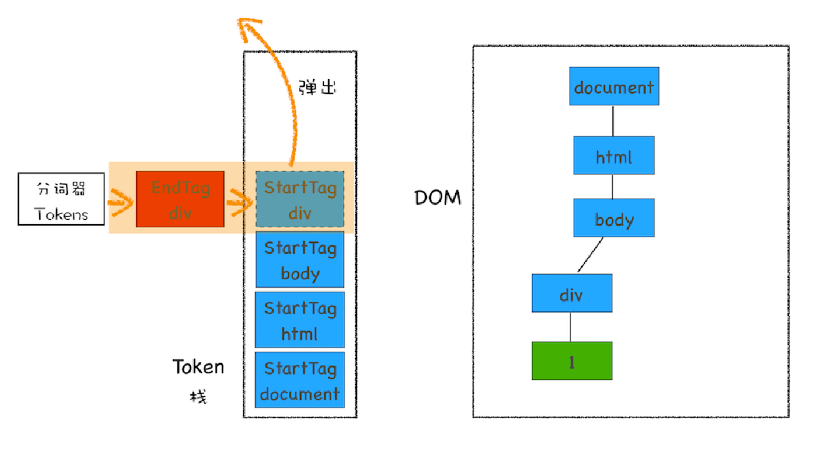
document上，如下图所示：

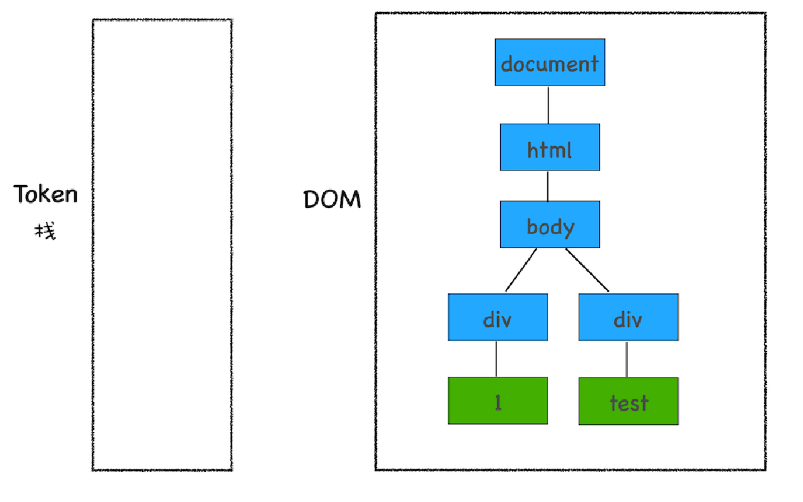






词器解析出来第一个EndTag div，这时候HTML解析器会去判断当前栈顶的元素是否是StartTag div，如果是则从栈顶弹出StartTag div，

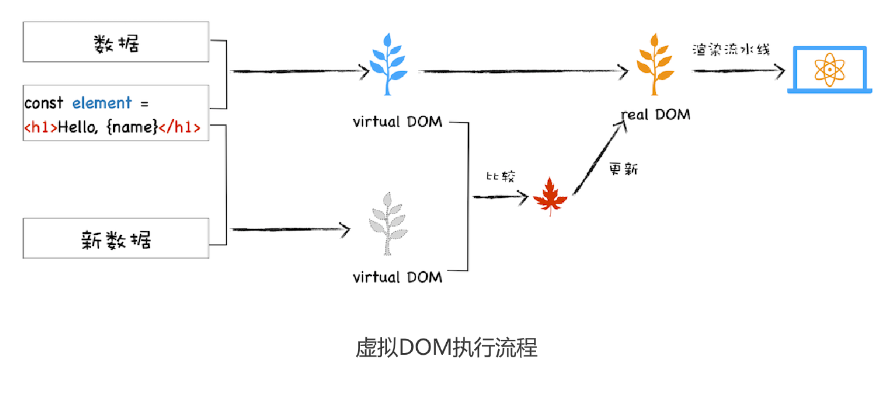




### 虚拟DOM

#### 虚拟DOM 需要解决的问题

1. 减轻操作DOM对页面渲染的负担
2. 代替真是DOM应对变化的频繁操作的业务
3. 将改变收集，一次性应用到真实的DOM上



#### 虚拟DOM运行流程

##### 创建阶段

1. 依据jsx 和 基础数据创建出来
2. 反映真实DOM树的结构
3. 然后由虚拟DOM 创建 真实的DOM 树
4. 真实DOM树生成之后，再触发渲染流水线
5. 屏幕显示页面

##### 更新阶段

1. 数据发生变化
2. 根据新数据创建一个新的虚拟DOM树
3. 对比新旧虚拟DOM 树找出变化（生成真实DOM之后，旧的虚拟DOM树被垃圾回收）
4. 把变化一次性更新到真实的DOM树上
5. 渲染流水线，生成页面

**注意**：这里我们重点关注下比较过程，最开始的时候，比较两个虚拟DOM的过程是在一个递归函数里执行的，其**核心算法是reconciliation**。通常情况下，这个比较过程执行得很快，不过当虚拟DOM比较复杂的时候，执行比较函数就有可能占据主线程比较久的时间，这样就会导致其他任务的等待，造成页面卡顿。为了解决这个问题，React团队重写了reconciliation算法，新的算法称为Fiber reconciler，之前老的算法称为Stack reconciler。

#### 双缓存

##### 解决了什么问题

在常规单缓存模式下，图像直接在屏幕上进行绘制。当需要更新图像时，可能会造成原有图像被擦除，并逐个绘制新的图像元素，这会导致图画切换和不流畅的动画效果，即屏幕闪烁

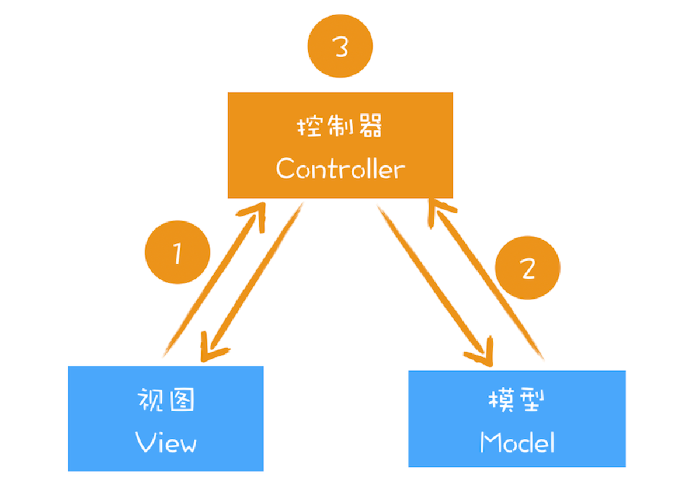
双缓存模式通过引入两个缓冲区：前缓冲区（Front Buffer） 和 后缓冲区（Back Buffer）

1. 前缓冲区是当前正在显示的图像缓冲区，它直接与屏幕进行交互。
2. 后缓冲区是另一个隐藏的图像缓冲区，用户可以在其中绘制、修改图像

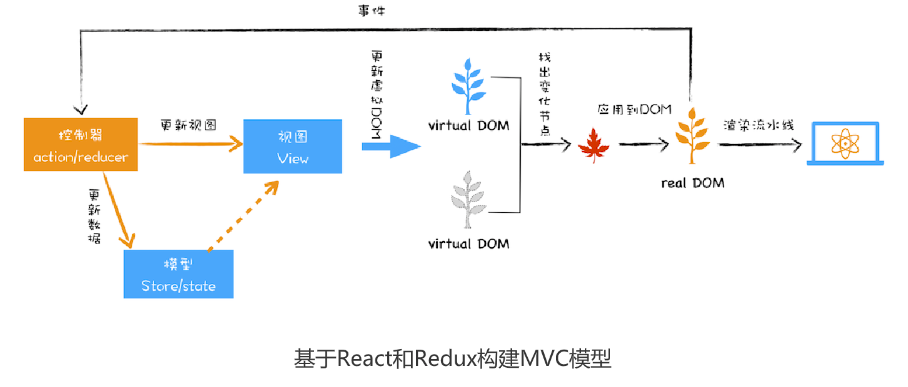
##### 工作流程

1. 首先，图像的绘制和修改操作均在后缓冲区进行，这样不会影响到前缓冲区中显示的内容。
2. 当所有图像处理完成后，将后缓冲区的内容一次性复制到前缓冲区。
3. 最后，前缓冲区的内容直接显示在屏幕上，实现平滑的图像切换，避免了屏幕闪烁。

#### MVC模式与 虚拟DOM



核心思想：将数据和视图分离



图中的控制器是用来监控DOM的变化，一旦DOM发生变化，控制器便会通知模型，让其

更新数据；

模型数据更新好之后，控制器会通知视图，告诉它模型的数据发生了变化；

视图接收到更新消息之后，会根据模型所提供的数据来生成新的虚拟DOM；

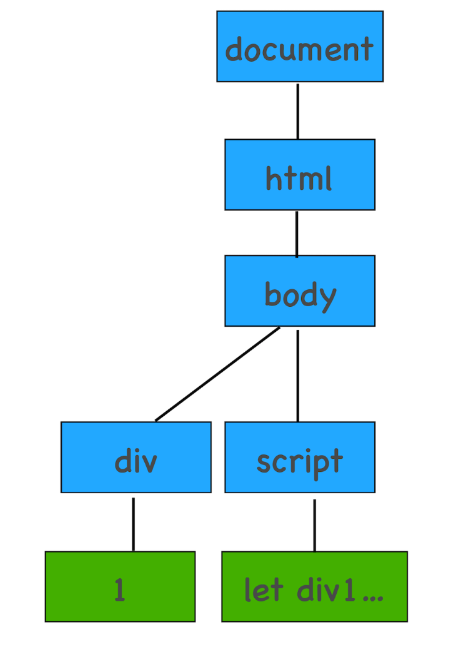
新的虚拟DOM生成好之后，就需要与之前的虚拟DOM进行比较，找出变化的节点；

比较出变化的节点之后，React将变化的虚拟节点应用到DOM上，这样就会触发DOM节点

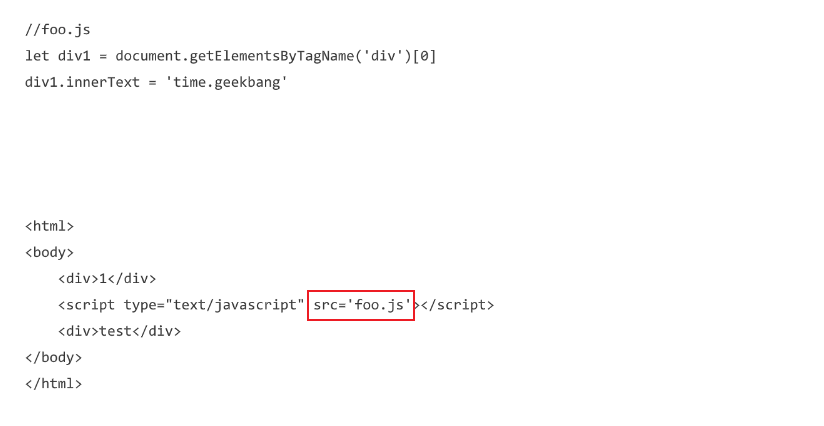
的更新；

DOM节点的变化又会触发后续一系列渲染流水线的变化，从而实现页面的更新。

### JavaScript 与 DOM



注意：当解析到script 脚本标签时：HTML解析器暂停工作，JavaScript引擎介入脚本执行完成之后，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。



需要先下载这段JavaScript代码、**JavaScript文件的下载过程会阻塞DOM解析**

Chrome浏览器做优化**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文件中包含的JavaScript、CSS等相关文

件，解析到相关文件之后，预解析线程会提前下载这些文件）

**注意**：异步加载 JavaScript 文件（async 或defer）

#### Async和defer

使用async标志的脚本文件一旦加载完成，会立即执行；

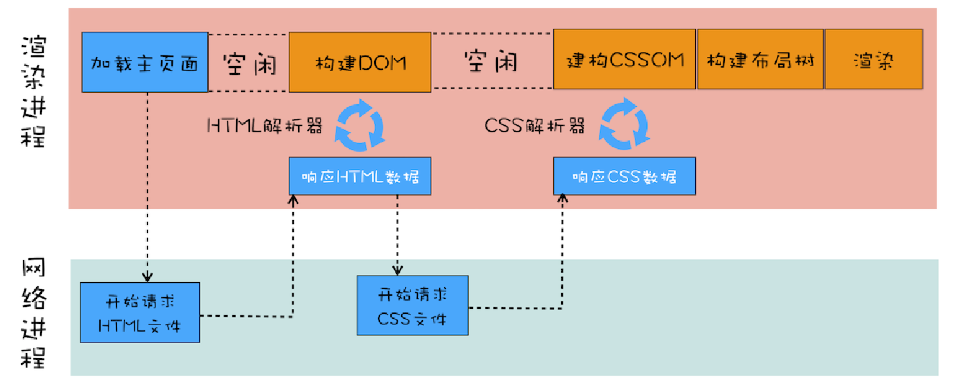
使用了defer标记的脚本文件，需要在DOMContentLoaded（**DOMContentLoaded 事件在解析和渲染 HTML 页面所需的基础资源加载完成之后就会触发**）事件之前执行。

如果代码里引用了**外部的CSS文件**，那么在执行JavaScript之前，还需要等待外部的CSS文件下载完成，并解析生成**CSSOM**对象之后，才能执行JavaScript脚本。

而JavaScript引擎在解析JavaScript之前，是不知道JavaScript是否操纵了CSSOM的，所以渲染引擎在遇到JavaScript脚本时，不管该脚本是否操纵了CSSOM，都会执行CSS文件下载，

解析操作，再执行JavaScript脚本。

### 渲染流水线：css 影响首次加载的白屏事件



DOM构建结束之后、theme.css文件还未下载完成的这段时间内，渲染流水线无事可做，因为下一步是合成布局树，而合成布局树需要CSSOM和DOM，所以这里需要等待CSS加载结束并解析成CSSOM。

#### CSSOM作用

和HTML一样，渲染引擎也是无法直接理解CSS文件内容的，所以需要将其解析成渲染引擎能够理解的结构，这个结构就是CSSOM。

##### **CSSOM作用**：

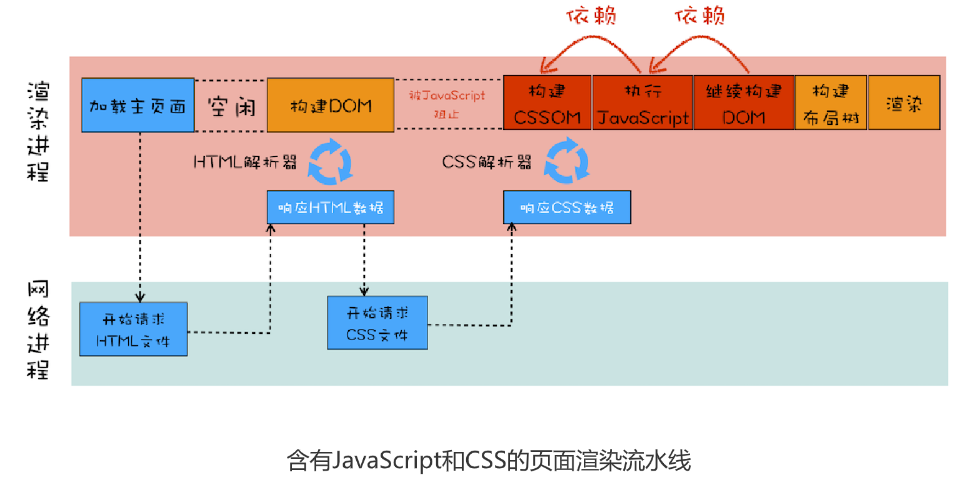
1. 提供JavaScript 操作样式表的能力
2. 为布局树的合成提供基础的样式信息

##### 样式计算

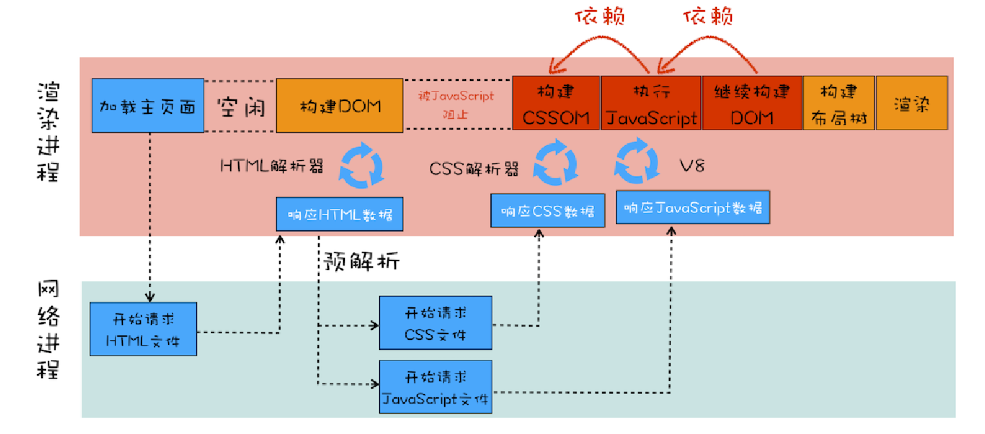
1. 渲染引擎构造布局树（复制DOM树的结构）、不需要显示的元素会被过滤
2. 渲染引擎会为对应的DOM元素选择对应的样式信息

**计**算**布局**

样式计算后，渲染引擎会需要计算布局树种每一个元素对应的几何位置



**含有外部JavaScript 文件 css 样式的渲染流水线**



从图中可以看出来，在接收到HTML数据之后的预解析过程中，HTML预解析器识别出来了有CSS文件和JavaScript文件需要下载，然后就同时发起这两个文件的下载请求，需要注意的是，**这两个文件的下载过程是重叠的，所以下载时间按照最久的那个文件来算**。

### 影响页面渲染的因素和优化策略

**渲染流水线影响到了首次页面展示的速度，而首次页面展示的速度又直接影响到了用户体验**，

#### 页面渲染历程：

1. 发送请求到提交数据阶段，页面渲染之前
2. 提交数据之后，渲染进程创建一个空白页面，这段时间统称为：解析白屏；等待css 文件和JavaScript 文件加载完成，生成CSSOM 和 DOM ，然后形成布局树，经过一系列步骤准备首次渲染
3. 等首次渲染完成后，开始进入完成页面的生成阶段，然后页面会一点点的被绘制出来

总结：

第一个步骤影响的主要因素：网络、服务器处理

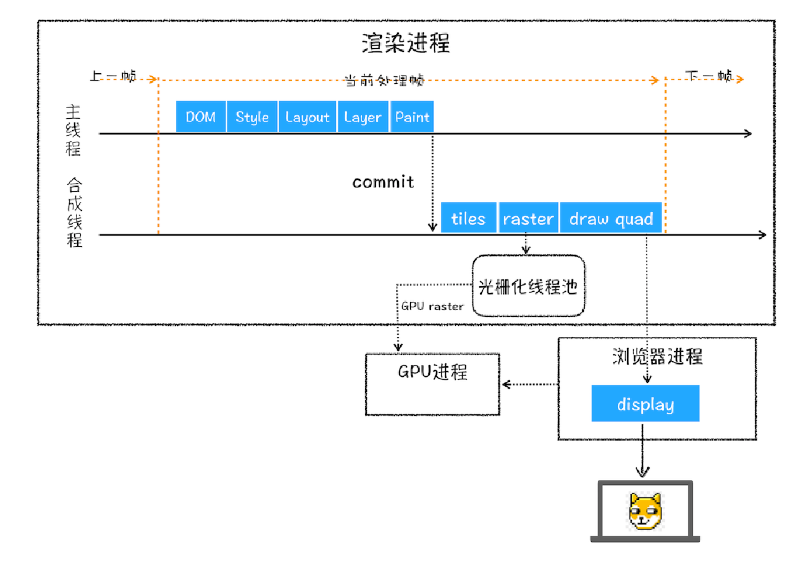
第二个阶段（主要）：主要任务包括： 解析HTML、下载CSS、下载JavaScript、生成CSSOM、执行JavaScript、生成布局树、绘制页面等一系列操作

主要瓶颈在：下载CSS文件和JavaScript 文件和JavaScript 执行

策略：

1. 内联JavaScript 、css （不用下载），这样获取到HTML 之后直接开始渲染流程
2. 不是所有场合都适合内联，尽量减少文件大小，通过打包工具删除不必要的代码注释、并压缩JavaScript 文件
3. 将不需要再解析HTML 阶段使用的JavaScript 标记上sync（下载完执行）或者defer（HTML解析资源加载完执行）
4. 对于大的css文件，可以通过媒体查询属性，将其拆分为多个不同用途的css 文件，这样只有在特定场景下才会加载特定的css 文件

## 分层和合成机制



### 重排、重绘和合成

这三种方式的渲染路径是不同的，**通常渲染路径越长，生成图像花费的时间就越多**。

#### 重排：

需要重新根据CSSOM和DOM来计算布局树，这样生成一幅图片时，会让整个渲染流

水线的每个阶段都执行一遍，如果布局复杂的话，就很难保证渲染的效率了

#### 重绘

没有了重新布局的阶段，操作效率稍微高点，但是依然需要重新计算绘制信息，并触发绘制操作之后的一系列操作。

#### 合成

并不需要触发布局和绘制两个阶段，如果采用了GPU，那么合成的效率会非常高。

合成技术：分层、分块和合成

### 分层和合成

通常页面的组成是非常复杂的，有的页面里要实现一些复杂的动画效果，比如点击菜单时弹

出菜单的动画特效，滚动鼠标滚轮时页面滚动的动画效果，当然还有一些炫酷的3D动画特

效。如果没有采用分层机制，从布局树直接生成目标图片的话，那么每次页面有很小的变化

时，都会触发重排或者重绘机制，这种“牵一发而动全身”的绘制策略会严重影响页面的渲

染效率。

**为了提升每帧的渲染效率，Chrome引入了分层和合成的机制。那该怎么来理解分层和合成机制呢**？

将素材分解为多个图层的操作就称为**分层**，最后将这些图层合并到一起的操

作就称为**合成**。

**理解了为什么要引入合成和分层机制，下面我们再来看看Chrome是怎么实现分层和合成机制的**。

在Chrome的渲染流水线中，**分层体现在生成布局树之后**，渲染引擎会根据布局树的特点将其转换为**层树（Layer Tree）**，层树是渲染流水线后续流程的基础结构。

层树中的每个节点都对应着一个图层，下一步的绘制阶段就依赖于层树中的节点。

将绘制指令组合成一个列表，有了**绘制列表**之后，就需要进入**光栅化**阶段了。

光栅化：按照绘制列表中的指令生成图片

每一个图层都对应一张图片，合成线程有了这些图片之后，会将这些图片合成为“一

张”图片，并最终将生成的图片发送到后缓冲区。这就是一个大致的分层、合成流程。

**需要重点关注的是，合成操作是在合成线程上完成的，这也就意味着在执行合成操作时，是不会影响到主线程执行的**。

#### 分块

如果说分层是从宏观上提升了渲染效率，那么分块则是从微观层面提升了渲染效率。

合成线程会将每个图层分割为大小固定的图块，然后优先绘制靠近视口的图块，这样

就可以大大加速页面的显示速度。

不过有时候， 即使只绘制那些优先级最高的图块，也要耗费不少的时间，因为涉及到一个很关键的因素——**纹理上传**，

因为从计算机内存上传到GPU内存的操作会比较慢。所以Chrome采用策略：**在首次合成图块的时候使用一个低分辨率的图片**。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。

#### 如何利用分层技术优化代码

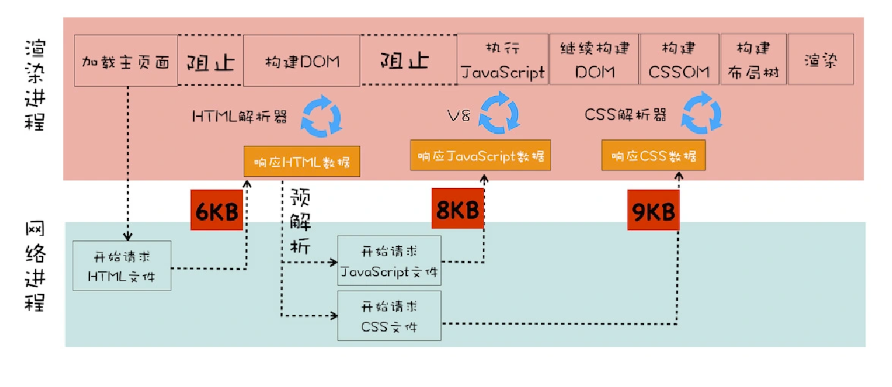
对某个元素做几何形状变换、透明度变换或者一些缩放操作，可以使用css 的will-change 属性替换JavaScript （会牵涉到整个渲染流水线，所以JavaScript的绘制效率会非常低下。）

## 浏览器渲染优化（总结）：

通常一个页面有三个阶段： **加载阶段**、**交互阶段**和**关闭阶段**。

### 加载阶段

指从发出请求到渲染出完整的的过程，影响到这个阶段的主要因素：网络和JavaScript脚本



**关键资源**： 能够阻塞网页首次渲染的资源

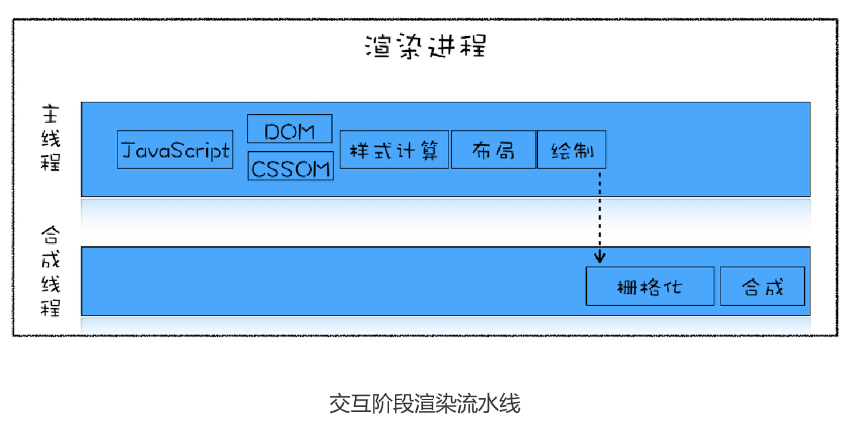
**影响页面首次渲染的核心因素**：关键资源的个数、关键资源的大小、请求关键资源需要多少RTT（Round Trip Time）

**RTT**： 它是网络中一个重要的性能指标，标识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

**加载阶段的优化原则**：减少关键资源个数，降低关键资源大小，降低关键资源RTT次数

### 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要因素：JavaScript脚本



**交互阶段优化原则**：让单个帧的生成速度变快

#### 减少JavaScript 脚本执行时间

有时JavaScript函数的一次执行时间可能有几百毫秒，这就严重霸占了主线程执行其他渲染任务的时间。针对这种情况我们可以采用以下两种策略：

1. 将函数分解为多个任务，每次执行时间不要太久
2. Web Workers，把一些和DOM操作无关且耗时的任务放到Web Workers中去执行

#### 避免强制同步布局

通过DOM接口执行添加元

素或者删除元素等操作后，是需要重新计算样式和布局的，不过正常情况下这些操作都是在

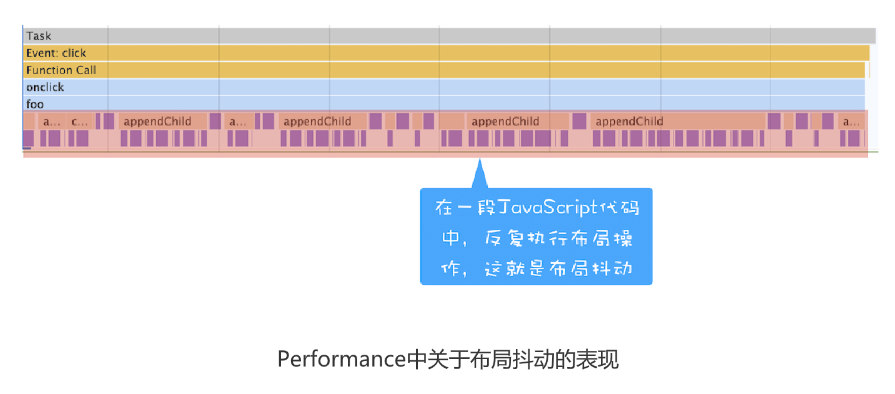
另外的任务中异步完成的，这样做是为了避免当前的任务占用太长的主线程时间

**所谓强制同步布局，是指JavaScript强制将计算样式和布局操作提前到当前的任务中**。

#### 避免抖动布局

**抖动布局**：是指在一次JavaScript执行过程中，多次执行强制布局和抖动操作。





#### 合理利用css 合成动画

合成动画是直接在合成线程上执行的，这和在主线程上执行的布局、绘制等操作不同，如果

主线程被JavaScript或者一些布局任务占用，CSS动画依然能继续执行。所以要尽量利用好

CSS合成动画，如果能让CSS处理动画，就尽量交给CSS来操作

另外，如果能提前知道对某个元素执行动画操作，那就最好将其标记为will-change，这是告

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

我们知道JavaScript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，那么垃圾回收器也会频繁地去执行垃圾回收策略。这样当垃圾回收操作发生时，就会占用主线程，

从而影响到其他任务的执行，严重的话还会让用户产生掉帧、不流畅的感觉。

所以要尽量避免产生那些临时垃圾数据。那该怎么做呢？可以尽可能优化储存结构，**尽可能避免小颗粒对象的产生**。

### 关闭阶段

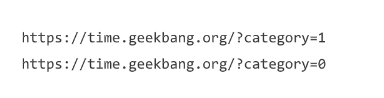
是用户发出关闭指令后页面所做出的一些清理操作

## 浏览器同源策略

### 什么是同源

**如果两个URL的协议、域名和端口都相同，我们就称这两个URL同源**。

比如下面这两个URL，它们具有相同的协议HTTPS、相同的域名time.geekbang.org，以及相同的端口443，所以我们就说这两个URL是同源的。



**同源策略**：浏览器默认两个相同的源之间是可以相互访问资源和操作DOM的。**两个不同的源之间若想要相互访问资源或者操作DOM**，那么会有一套基础的安全策略的制约，我们把这称为同源策

略。

### 同源策略具体体现

#### DOM层面

同源策略限制了来自不同源的JavaScript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。由于同源策略，我们依然无法通过第二个页面的opener来访问第一个

页面中的Cookie、IndexDB或者LocalStorage等内容。

#### 网络层面

网络请求跨域

### 安全和便利性的权衡

安全性和便利性是相互对立的，让不同的源之间绝对隔离，无疑是最安全的措施，但这

也会使得Web项目难以开发和使用。

#### 页面中可以嵌入第三方资源

为了解决XSS攻击，浏览器中引入了内容安全策略，称为CSP。

**CSP的核心思想是让服务器决定浏览器能够加载哪些资源，让服务器决定浏览器是否能够执行内联JavaScript代码**。通过这些手段就可以大大减少XSS攻击。

#### 跨域资源共享和跨文档消息机制

默认情况下，如果打开极客邦的官网页面，在官网页面中通过XMLHttpRequest或者Fetch来请求InfoQ中的资源，这时同源策略会阻止其向InfoQ发出请求，这样会大大制约我们的生产力。

为了解决这个问题，我们引入了**跨域资源共享（CORS）**，使用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。

在介绍同源策略时，我们说明了如果两个页面不是同源的，则无法相互操纵DOM。不过在实际应用中，经常需要两个不同源的DOM之间进行通信，于是浏览器中又引入了**跨文档消息机制**，可以通过window.postMessage的JavaScript接口来和不同源的DOM进行通信。

## 浏览器攻击

### 跨站脚本攻击（xss）

#### 什么是XSS 攻击

XSS全称是Cross Site Scripting， 为了与“CSS”区分开来故简称XSS，翻译过来就是“跨站脚本攻击”。XSS攻击是指黑客往HTML文件中或者DOM 中注入恶意脚本，从而在用户浏览页面时利用注入恶意脚本对用户实施攻击的一种手段。

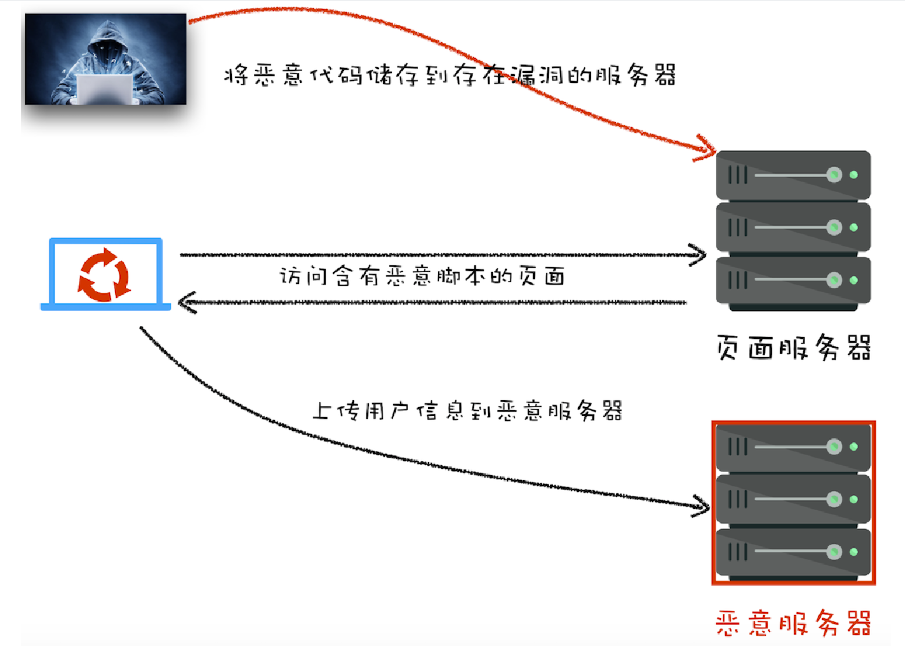
#### 恶意脚本主要作用

1. 获取cookie信息
2. 监听用户行为
3. 修改DOM
4. 在页面生成浮窗广告

#### 恶意脚本如何注入

Xss 攻击的类型： 存储型xss 攻击、 反射型XSS攻击、基于DOM的xss 攻击三种

##### 存储型XSS攻击



##### 反射型XSS攻击

Web服务器不会存储反射型XSS攻击的恶意脚本，这是和存储型XSS攻

击不同的地方。

##### 基于DOM的XSS 攻击

Web资源传输过程或者在用户使用页面的过程中修改Web页面的数据。

##### 如何阻止XSS攻击

存储型XSS攻击和反射型XSS攻击是针对服务端漏洞的攻击

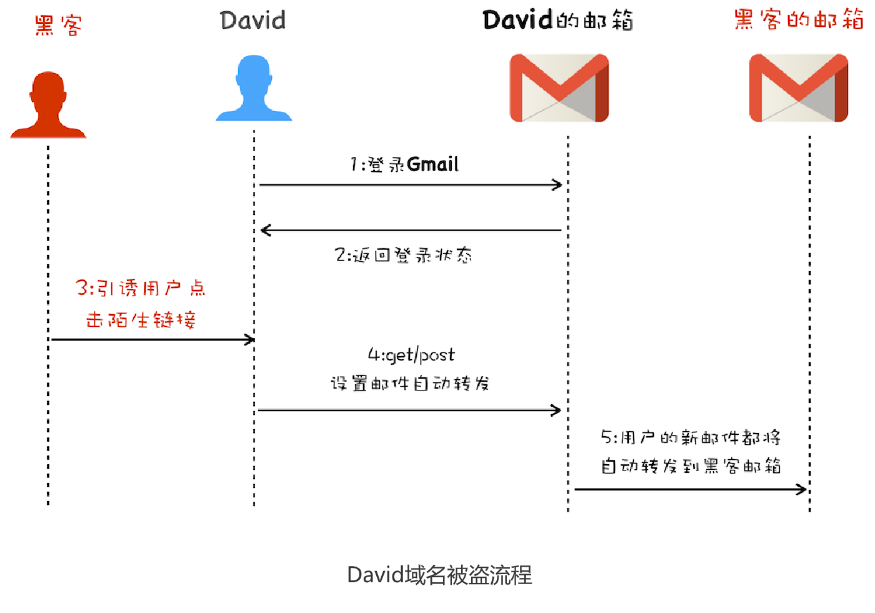
基于DOM 的XSS 攻击是针对前端安全漏洞的攻击

要阻止XSS 攻击，可以通过阻止恶意脚本注入和恶意消息的发送实现

1. 服务器对输入脚本进行过滤或者转码
2. 充分利用CSP（禁止向第三方域提交数据、禁止执行内联和未授权脚本）
3. 使用httpOnly属性（保护cookie 安全、无法通过document.Cookie读取交给服务端操作）

### CSRF 攻击

案例：

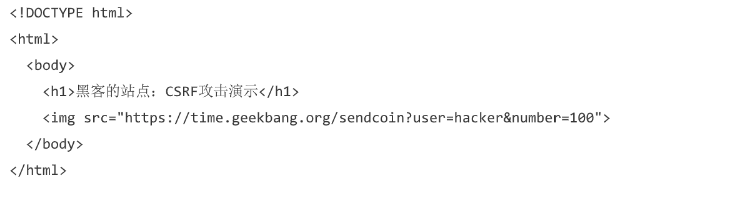


#### 什么是CSRF攻击

CSRF英文全称是Cross-site request forgery，所以又称为“跨站请求伪造”

CSRF攻击就是黑客利用了用户的登录状态，并通过第三方的站点来做一些坏事。

#### 自动发起get 请求



这是黑客页面的HTML代码，在这段代码中，黑客将转账的请求接口隐藏在img标签内，欺骗浏览器这是一张图片资源。当该页面被加载时，浏览器会自动发起img的资源请求，如果服务器没有对该请求做判断的话，那么服务器就会认为该请求是一个转账请求，于是用户账户上的100极客币就被转移到黑客的账户上去了。

#### 自动发起post请求



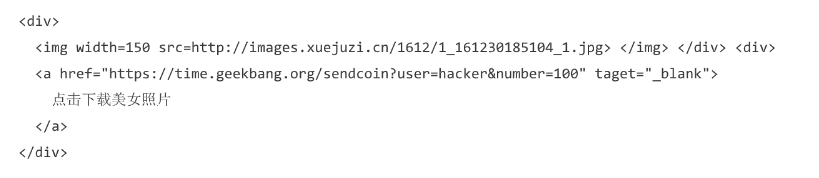
在这段代码中，我们可以看到黑客在他的页面中构建了一个隐藏的表单，该表单的内容就是

极客时间的转账接口。当用户打开该站点之后，这个表单会被自动执行提交；当表单被提交

之后，服务器就会执行转账操作。因此使用构建自动提交表单这种方式，就可以自动实现跨

站点POST数据提交

#### 引诱用户点击链接



这段黑客站点代码，页面上放了一张美女图片，下面放了图片下载地址，而这个下载地址实

际上是黑客用来转账的接口，一旦用户点击了这个链接，那么他的极客币就被转到黑客账户

上了。

**和XSS不同的是，CSRF攻击不需要将恶意代码注入用户的页面，仅仅是利用服务器的漏洞和用户的登录状态来实施攻击**。

#### 如何防止CSRF 攻击

##### 攻击要素：

第一个，目标站点一定要有CSRF漏洞；

第二个，用户要登录过目标站点，并且在浏览器上保持有该站点的登录状态；

第三个，需要用户打开一个第三方站点，可以是黑客的站点，也可以是一些论坛。

##### 防攻手段

###### 充分利用好Cookie 的 SameSite 属性

Cookie正是浏览器和服务器之间维护登录状态的一个关键数据

Cookie 中的SameSite属性正是为了解决这个问题的，通过使用SameSite可以有效地降低CSRF攻击的风险。

在HTTP响应头中，通过set-cookie字段设置Cookie时，可以带上SameSite选项，如下：



SameSite选项通常有Strict、Lax和None三个值。

Strict最为严格。如果SameSite的值是Strict，那么浏览器会完全禁止第三方 Cookie。

只有你从InfoQ的站点去请求InfoQ的资源时，才会带上这些Cookie。

Lax相对宽松一点。在跨站点的情况下，从第三方站点的链接打开和从第三方站点提交Get

方式的表单这两种方式都会携带Cookie。第三方站点中使用Post方法，或者通过

img、iframe等标签加载的URL，这些场景都不会携带Cookie。

None的话，在任何情况下都会发送Cookie数据。

###### 验证请求的来源站点

接着我们再来了解另外一种防止CSRF攻击的策略，那就是**在服务器端验证请求来源的站点**。由于CSRF攻击大多来自于第三方站点，因此服务器可以禁止来自第三方站点的请求。

那么该怎么判断请求是否来自第三方站点呢？

这就需要介绍HTTP请求头中的 Referer和Origin 属性了。

**Referer是HTTP请求头中的一个字段，记录了该HTTP请求的来源地址**。

但在服务器端验证请求头中的Referer并不是太可靠，因此标准委员会又制定了**Origin属性**，在一些重要的场合，比如通过XMLHttpRequest、Fecth发起跨站请求或者通过Post方法发送请求时，都会带上Origin属性，

**Origin属性只包含了域名信息，并没有包含具体的URL路径**。

因此，服务器的策略是优先判断Origin，如果请求头中没有包含Origin属性，再根据实际情

况判断是否使用Referer值。

###### CSRF Token

第一步，在浏览器向服务器发起请求时，服务器生成一个CSRF Token。

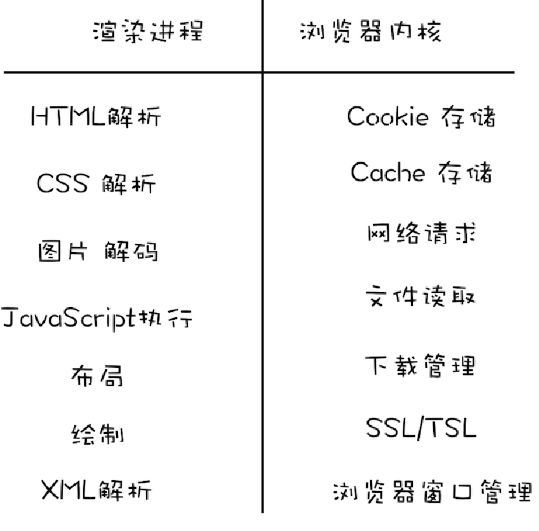
第二步，在浏览器端如果要发起转账的请求，那么需要带上页面中的CSRF Token，然后服务

器会验证该Token是否合法。

## 安全沙箱

渲染进程和操作系统之间建一道墙，即便渲染进程由于存在漏洞

被黑客攻击，但由于这道墙，黑客就获取不到渲染进程之外的任何操作权限。**将渲染进程和操作系统隔离的这道墙就是我们要聊的安全沙箱**。



### 持久储存

我们先来看看安全沙箱是如何影响到浏览器持久存储的。由于安全沙箱需要负责确保渲染进

程无法直接访问用户的文件系统，但是在渲染进程内部有访问Cookie的需求、有上传文件的需求，为了解决这些文件的访问需求，所以现代浏览器将读写文件的操作全部放在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。

### 网络访问

同样有了安全沙箱的保护，在渲染进程内部也是不能直接访问网络的，如果要访问网络，则

需要通过浏览器内核。不过浏览器内核在处理URL请求之前，会检查渲染进程是否有权限请求该URL，比如检查XMLHttpRequest或者Fetch是否是跨站点请求，或者检测HTTPS的站点中是否包含了HTTP的请求。

### 用户交互

### 站点隔离

目前所有操作系统都面临着两个A级漏洞——幽灵（Spectre）和熔毁（Meltdown），这两

个漏洞是由处理器架构导致的，很难修补，黑客通过这两个漏洞可以直接入侵到进程的内

部，如果入侵的进程没有安全沙箱的保护，那么黑客还可以发起对操作系统的攻击。

因此Chrome几年前就开始重构代码，将标签级的渲染进程重构为iframe级的渲染进程，然后严格按照同一站点的策略来分配渲染进程，这就是Chrome中的站点隔离。

实现了站点隔离，就可以将恶意的iframe隔离在恶意进程内部，使得它无法继续访问其他

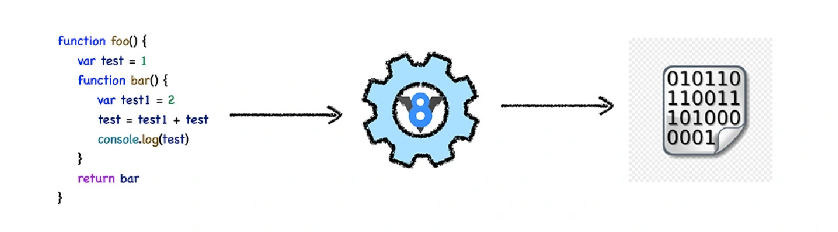
iframe进程的内容，因此也就无法攻击其他站点了。

# V8 引擎

## 前言

V8是JavaScript虚拟机的一种。我们可以简单地把JavaScript虚拟机理解成是一个翻译程序，

将人类能够理解的**编程语言JavaScript**，翻译成机器能够理解的**机器语言**。



市面上主流的JavaScript 引擎： SpiderMonkey、V8、JavaScriptCore

