《[浏览器工作原理与实践](https://time.geekbang.org/column/intro/216)》是极客时间上的一个浏览器学习系列，在学习之后特在此做记录和总结。

## 一、Chrome架构

**1）进程架构**

  Chrome打开一个页面会启动4个进程：网络进程、GPU进程、浏览器主进程和渲染进程。

  最新的 Chrome 浏览器包括：1 个浏览器（Browser）主进程、1 个 GPU 进程、1 个网络（NetWork）进程、多个渲染进程和多个插件进程。 1. 浏览器主进程：负责界面显示、用户交互、子进程管理，同时提供存储等功能。 2. 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。 3. GPU进程：GPU 的使用初衷是为了实现 3D CSS 的效果，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。 4. 网络进程：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。 5. 插件进程：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。

**2）安全沙箱**

  在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安全沙箱。

  浏览器中的安全沙箱是利用操作系统提供的安全技术，让渲染进程在执行过程中无法访问或者修改操作系统中的数据，在渲染进程需要访问系统资源的时候，得通过浏览器内核来实现，然后将访问的结果通过 IPC 转发给渲染进程。

  文件内容的读写都是在浏览器内核中完成的。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。

  操作系统没有将用户输入事件直接传递给渲染进程，而是将这些事件传递给浏览器内核。然后浏览器内核再根据当前浏览器界面的状态来判断如何调度这些事件，如果当前焦点位于浏览器地址栏中，则输入事件会在浏览器内核内部处理；如果当前焦点在页面的区域内，则浏览器内核会将输入事件转发给渲染进程。

## 二、HTTP协议

  浏览器端发起 HTTP 请求流程： 1. 构建请求，构建好后，浏览器准备发起网络请求。 2. 查找缓存，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。 3. 准备 IP 地址和端口，HTTP 的内容是通过 TCP 来传输的，第一步就是先建立TCP连接，而请求 DNS（DNS数据缓存服务） 可返回域名对应的 IP。 4. 等待 TCP 队列，同一个域名同时最多只能建立 6 个 TCP 连接，如果同时有 10 个请求发生，那么其中 4 个会进入等待状态。 5. 建立 TCP 连接，快乐地和服务器握手。 6. 发送 HTTP 请求，和服务器进行通信，向服务器发送请求行，包括请求方法、请求 URI（Uniform Resource Identifier）和 HTTP 版本协议。

  服务器端处理 HTTP 请求流程： 1. 返回请求，一旦服务器处理结束，便可以返回响应行，包括协议版本和状态码。 2. 断开连接，但浏览器或者服务器在其头信息中加入了：Connection:Keep-Alive，那么 TCP 连接在发送后将仍然保持打开状态。 3. 重定向，当两个 URL 不一样时，会涉及一个重定向操作。

  为什么很多站点第二次打开速度会很快？因为 DNS 缓存和页面资源缓存这两块数据是会被浏览器缓存的。

  登录状态是如何保持的？浏览器页面状态是通过使用 Cookie 来实现的。如果服务器端发送的响应头内有 Set-Cookie 的字段，那么浏览器就会将该字段的内容保存到本地。

## 三、导航流程

  在浏览器里，从输入 URL 到页面展示，这中间发生了什么？其中涉及到了网络、操作系统、Web 等一系列的知识。

  整个流程大致描述如下： 1. 首先，浏览器进程接收到用户输入的 URL 请求，浏览器进程便将该 URL 转发给网络进程。 2. 然后，在网络进程中发起真正的 URL 请求。 3. 接着，网络进程接收到了响应头数据，便解析响应头数据，并将数据转发给浏览器进程。 4. 浏览器进程接收到网络进程的响应头数据之后，发送“提交导航 (Commit Navigation)”消息到渲染进程； 5. 渲染进程接收到“提交导航”的消息之后，便开始准备接收 HTML 数据，接收数据的方式是直接和网络进程建立数据管道； 6. 最后，渲染进程会向浏览器进程“确认提交”，这是告诉浏览器进程：“已经准备好接受和解析页面数据了”。 7. 浏览器进程接收到渲染进程“提交文档”的消息之后，便开始移除之前旧的文档，然后更新浏览器进程中的页面状态。

  这其中，用户发出 URL 请求到页面开始解析的这个过程，就叫做导航。

**1）用户输入**

  当用户在地址栏中输入一个查询关键字时，地址栏会判断输入的关键字是搜索内容，还是请求的 URL。

**2）URL 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，网络进程接收到 URL 请求后，会在这里发起真正的 URL 请求流程（参考上面的HTTP请求流程）。

**3）准备渲染进程**

  Chrome 会为每个页面分配一个渲染进程，也就是说，每打开一个新页面就会配套创建一个新的渲染进程。

**4）提交文档**

  提交文档，就是指浏览器进程将网络进程接收到的 HTML 数据提交给渲染进程。

**5）渲染阶段**

  一旦文档被提交，渲染进程便开始页面解析和子资源加载。

  一旦页面生成完成，渲染进程会发送一个消息给浏览器进程，浏览器接收到消息后，会停止标签图标上的加载动画。

## 四、渲染流程

  由于渲染机制过于复杂，所以渲染模块在执行过程中会被划分为很多子阶段，输入的 HTML 经过这些子阶段，最后输出像素，这样的一个处理流程叫做渲染流水线。

  按照渲染的时间顺序，流水线可分为如下几个子阶段：构建 DOM 树、样式计算、布局阶段、分层、绘制、分块、光栅化和合成。

**1）构建 DOM 树**

  浏览器无法直接理解和使用 HTML，所以需要将 HTML 转换为浏览器能够理解的结构——DOM 树

**2）样式计算**

  样式计算（Recalculate Style）地目的是为了计算出 DOM 节点中每个元素的具体样式，这个阶段大体可分为三步来完成。

1. 把 CSS 转换为浏览器能够理解的结构——styleSheets。
2. 转换样式表中的属性值，使其标准化，如 2em、blue、bold，需要将它们转换为渲染引擎容易理解的、标准化的计算值。
3. 计算出 DOM 树中每个节点的具体样式，涉及到 CSS 的继承规则和层叠规则。此阶段最终输出的内容是每个 DOM 节点的样式，并被保存在 ComputedStyle 的结构内。

**3）布局阶段**

  布局就是计算 DOM 树中可见元素几何位置的过程。Chrome 在布局阶段需要完成两个任务： 1. 创建只包含可见元素的布局树。 2. 计算布局树节点的坐标位置。

**4）分层**

  渲染引擎还需要为特定的节点生成专用的图层，并生成一棵对应的图层树（LayerTree）。

**5）图层绘制**

  渲染引擎实现图层的绘制与之类似，会把一个图层的绘制拆分成很多小的绘制指令，然后再把这些指令按照顺序组成一个待绘制列表。

**6）栅格化操作**

  绘制列表只是用来记录绘制顺序和绘制指令的列表，而实际上绘制操作是由渲染引擎中的合成线程来完成的。

  合成线程会将图层划分为图块（tile），这些图块的大小通常是 256x256 或者 512x512。

  然后合成线程会按照视口附近的图块来优先生成位图，实际生成位图的操作是由栅格化来执行的。

  所谓栅格化（raster），是指将图块转换为位图。而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的。

  通常，栅格化过程都会使用 GPU 来加速生成，使用 GPU 生成位图的过程叫快速栅格化，或者 GPU 栅格化，生成的位图被保存在 GPU 内存中。

**7）合成和显示**

  一旦所有图块都被光栅化，合成线程就会生成一个绘制图块的命令——“DrawQuad”，然后将该命令提交给浏览器进程。

  将其页面内容绘制到内存中，最后再将内存显示在屏幕上。

  一个完整的渲染流程大致可总结为如下： 1. 渲染进程将 HTML 内容转换为能够读懂的 DOM 树结构。 2. 渲染引擎将 CSS 样式表转化为浏览器可以理解的 styleSheets，计算出 DOM 节点的样式。 3. 创建布局树，并计算元素的布局信息。 4. 对布局树进行分层，并生成分层树。 5. 为每个图层生成绘制列表，并将其提交到合成线程。 6. 合成线程将图层分成图块，并在光栅化线程池中将图块转换成位图。 7. 合成线程发送绘制图块命令 DrawQuad 给浏览器进程。 8. 浏览器进程根据 DrawQuad 消息生成页面，并显示到显示器上。

  渲染引擎会通过合成线程直接去处理变换，这些变换并没有涉及到主线程，这样就大大提升了渲染的效率。这也是 CSS 动画比 JavaScript 动画高效的原因。

## 五、JavaScript

**1）执行流程**

  实际上变量和函数声明在代码里的位置是不会改变的，而且是在编译阶段被 JavaScript 引擎放入内存中。

  一段 JavaScript 代码在执行之前需要被 JavaScript 引擎编译，编译完成之后，才会进入执行阶段。

  经过编译后，会生成两部分内容：执行上下文（Execution context）和可执行代码。

  执行上下文是 JavaScript 执行一段代码时的运行环境，比如调用一个函数，就会进入这个函数的执行上下文，确定该函数在执行期间用到的诸如 this、变量、对象以及函数等。

**2）调用栈**

  在执行上下文创建好后，JavaScript 引擎会将执行上下文压入栈中，通常把这种用来管理执行上下文的栈称为执行上下文栈，又称调用栈。

**3）块级作用域**

  函数内部通过 var 声明的变量，在编译阶段全都被存放到变量环境里面了。

  通过 let 声明的变量，在编译阶段会被存放到词法环境（Lexical Environment）中。

  在函数的作用域块内部，通过 let 声明的变量并没有被存放到词法环境中。

  其实，在词法环境内部，维护了一个小型栈结构，栈底是函数最外层的变量，进入一个作用域块后，就会把该作用域块内部的变量压到栈顶；当作用域执行完成之后，该作用域的信息就会从栈顶弹出，这就是词法环境的结构。

**4）作用域链**

  在每个执行上下文的变量环境中，都包含了一个称为 outer的外部引用，用来指向外部的执行上下文。

  当一段代码使用了一个变量时，JavaScript 引擎首先会在“当前的执行上下文”中查找该变量。如果在当前的变量环境中没有查找到，那么 JavaScript 引擎会继续在 outer 所指向的执行上下文中查找。

**5）词法作用域**

  在 JavaScript 执行过程中，其作用域链是由词法作用域决定的。

  词法作用域就是指作用域是由代码中函数声明的位置来决定的，所以词法作用域是静态的作用域，通过它就能够预测代码在执行过程中如何查找标识符。

  词法作用域是代码编译阶段就决定好的，和函数是怎么调用的没有关系。

**6）闭包**

  在 JavaScript 中，根据词法作用域的规则，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了，但是内部函数引用外部函数的变量依然保存在内存中，把这些变量的集合称为闭包。

  foo 函数执行完成之后，其执行上下文从栈顶弹出了，但是由于返回的 setName 和 getName 方法中使用了 foo 函数内部的变量 myName 和 test1，所以这两个变量依然保存在内存中。这像极了 setName 和 getName 方法背的一个专属背包，之所以是专属背包，是因为除了 setName 和 getName 函数之外，其他任何地方都是无法访问该背包的，可以把这个背包称为 foo 函数的闭包。

  总的来说，产生闭包的核心有两步：第一步是需要预扫描内部函数；第二步是把内部函数引用的外部变量保存到堆中。

**7）闭包回收**

  如果引用闭包的函数是个全局变量，那么闭包会一直存在直到页面关闭；但如果这个闭包以后不再使用的话，就会造成内存泄漏。

  如果引用闭包的函数是个局部变量，等函数销毁后，在下次 JavaScript 引擎执行垃圾回收时，判断闭包这块内容如果已经不再被使用了，那么 JavaScript 引擎的垃圾回收器就会回收这块内存。

**8）this**

  this 是和执行上下文绑定的，也就是说每个执行上下文中都有一个 this。

  执行上下文主要分为三种——全局执行上下文、函数执行上下文和 eval 执行上下文 1. 全局执行上下文中的 this 是指向 window 对象的。这也是 this 和作用域链的唯一交点，作用域链的最底端包含了 window 对象。 2. 默认情况下调用一个函数，其执行上下文中的 this 也是指向 window 对象的。通常情况下，有下面三种方式来设置函数执行上下文中的 this 值。 \* 通过函数的 call 方法设置。 \* 通过对象调用方法设置。使用对象来调用其内部的一个方法，该方法的 this 是指向对象本身的。 \* 通过构造函数中设置。当执行 new CreateObj() 的时候，JavaScript 引擎做了如下四件事： 1. 首先创建了一个空对象 tempObj； 2. 接着调用 CreateObj.call 方法，并将 tempObj 作为 call 方法的参数，这样当 CreateObj 的执行上下文创建时，它的 this 就指向了 tempObj 对象； 3. 然后执行 CreateObj 函数，此时的 CreateObj 函数执行上下文中的 this 指向了 tempObj 对象； 4. 最后返回 tempObj 对象。

**9）内存空间**

  在 JavaScript 的执行过程中， 主要有三种类型内存空间，分别是代码空间、栈空间和堆空间。

  JavaScript 引擎并不是直接将该对象存放到变量环境中，而是将它分配到堆空间里面，分配后该对象会有一个在“堆”中的地址。

  原始类型的数据值都是直接保存在“栈”中的，引用类型的值是存放在“堆”中的。

  为什么一定要分“堆”和“栈”两个存储空间呢？

  因为 JavaScript 引擎需要用栈来维护程序执行期间上下文的状态，如果栈空间大了，所有的数据都存放在栈空间里面，那么会影响到上下文切换的效率，进而又影响到整个程序的执行效率。

**10）垃圾回收**

  栈中的数据回收，JavaScript 会将 ESP 下移到 foo 函数的执行上下文，这个下移操作就是销毁 showName 函数执行上下文的过程。

  堆中的数据回收，垃圾回收器的工作流程： 1. 第一步是标记空间中活动对象和非活动对象。所谓活动对象就是还在使用的对象，非活动对象就是可以进行垃圾回收的对象。 2. 第二步是回收非活动对象所占据的内存。其实就是在所有的标记完成之后，统一清理内存中所有被标记为可回收的对象。 3. 第三步是做内存整理，频繁回收对象后，内存中就会存在大量不连续空间，这些不连续的内存空间称为内存碎片。

  由于 JavaScript 是运行在主线程之上的，一旦执行垃圾回收算法，都需要将正在执行的 JavaScript 脚本暂停下来，待垃圾回收完毕后再恢复脚本执行，这种行为叫做全停顿（Stop-The-World）。

  为了降低老生代的垃圾回收而造成的卡顿，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 JavaScript 应用逻辑交替进行，直到标记阶段完成，把这个算法称为增量标记（Incremental Marking）算法。

## 六、事件循环

  事件循环是指渲染主线程会循环地从消息队列头部中读取任务，执行任务。

**1）消息队列**

  消息队列是一种数据结构，可以存放要执行的任务。它符合队列“先进先出”的特点，也就是说要添加任务的话，添加到队列的尾部；要取出任务的话，从队列头部去取。

  渲染进程专门有一个 IO 线程用来接收其他进程传进来的消息，接收到消息之后，会将这些消息组装成任务发送给渲染主线程。

**2）消息类型**

  包含很多内部消息类型，如输入事件（鼠标滚动、点击、移动）、微任务、文件读写、WebSocket、JavaScript 定时器等。

  除此之外，消息队列中还包含了很多与页面相关的事件，如 JavaScript 执行、解析 DOM、样式计算、布局计算、CSS 动画等。

**3）宏任务和微任务**

  通常把消息队列中的任务称为宏任务（如下所列），每个宏任务中都包含了一个微任务队列，在执行宏任务的过程中，如果 DOM 有变化，那么就会将该变化添加到微任务列表中，这样就不会影响到宏任务的继续执行，因此也就解决了执行效率的问题。 1. 渲染事件（如解析 DOM、计算布局、绘制）； 2. 用户交互事件（如鼠标点击、滚动页面、放大缩小等）； 3. JavaScript 脚本执行事件； 4. 网络请求完成、文件读写完成事件。

  等宏任务中的主要功能都执行完成之后，这时候，渲染引擎并不着急去执行下一个宏任务，而是执行当前宏任务中的微任务，因为 DOM 变化的事件都保存在这些微任务队列中，这样也就解决了实时性问题。

  当 JavaScript 执行一段脚本的时候，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。

  在现代浏览器里面，产生微任务有两种方式。 1. 第一种方式是使用 MutationObserver 监控某个 DOM 节点，然后再通过 JavaScript 来修改这个节点，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，就会产生 DOM 变化记录的微任务。 2. 第二种方式是使用 Promise，当调用 Promise.resolve() 或者 Promise.reject() 的时候，也会产生微任务。

  在执行微任务过程中产生的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中继续执行。

**4）延迟队列**

  在 Chrome 中除了正常使用的消息队列之外，还有另外一个消息队列，这个队列中维护了需要延迟执行的任务列表，包括了定时器和 Chromium 内部一些需要延迟执行的任务。

  所以当通过 JavaScript 创建一个定时器时，渲染进程会将该定时器的回调任务添加到延迟队列中。

  处理完消息队列中的一个任务之后，就开始执行延迟函数。该函数会根据发起时间和延迟时间计算出到期的任务，然后依次执行这些到期的任务。等到期的任务执行完成之后，再继续下一个循环过程。

**5）XMLHttpRequest**

  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。

## 七、requestAnimationFrame

  当显示器将一帧画面绘制完成后，并在准备读取下一帧之前，显示器会发出一个垂直同步信号（Vertical Synchronization）给 GPU，简称 VSync。

  具体地讲，当 GPU 接收到 VSync 信号后，会将 VSync 信号同步给浏览器进程，浏览器进程再将其同步到对应的渲染进程，渲染进程接收到 VSync 信号之后，就可以准备绘制新的一帧了。

  在合成完成之后，合成线程会提交给渲染主线程提交完成合成的消息，如果当前合成操作执行的非常快，比如从用户发出消息到完成合成操作只花了 8 毫秒，因为 VSync 同步周期是 16.66（1/60）毫秒，那么这个 VSync 时钟周期内就不需要再次生成新的页面了。那么从合成结束到下个 VSync 周期内，就进入了一个空闲时间阶段，那么就可以在这段空闲时间内执行一些不那么紧急的任务，比如 V8 的垃圾回收，或者通过 window.requestIdleCallback() 设置的回调任务等，都会在这段空闲时间内执行。

  CSS 动画是由渲染进程自动处理的，所以渲染进程会让 CSS 渲染每帧动画的过程与 VSync 的时钟保持一致, 这样就能保证 CSS 动画的高效率执行。

  但是 JavaScript 是由用户控制的，如果采用 setTimeout 来触发动画每帧的绘制，那么其绘制时机是很难和 VSync 时钟保持一致的，所以 JavaScript 中又引入了 window.requestAnimationFrame，用来和 VSync 的时钟周期同步，它的回调任务会在每一帧的开始执行。

## 八、Promise

**1）消灭嵌套**

  Promise 主要通过下面两步解决嵌套回调问题的。 1. 首先，Promise 实现了回调函数的延时绑定。回调函数的延时绑定在代码上体现就是先创建 Promise 对象 x1，通过 Promise 的构造函数 executor 来执行业务逻辑；创建好 Promise 对象 x1 之后，再使用 x1.then 来设置回调函数。 2. 其次，需要将回调函数 onResolve 的返回值穿透到最外层。因为根据 onResolve 函数的传入值来决定创建什么类型的 Promise 任务，创建好的 Promise 对象需要返回到最外层，这样就可以摆脱嵌套循环了。

**2）合并错误处理**

  无论哪个对象里面抛出异常，都可以通过最后一个对象 catch 来捕获异常，通过这种方式可以将所有 Promise 对象的错误合并到一个函数来处理，这样就解决了每个任务都需要单独处理异常的问题。

  之所以可以使用最后一个对象来捕获所有异常，是因为 Promise 对象的错误具有“冒泡”性质，会一直向后传递，直到被 onReject 函数处理或 catch 语句捕获为止。

**3）Promise与微任务**

  Promise之所以要使用微任务是由 Promise 回调函数延迟绑定技术导致的。

  为了避免在执行初始化回调函数时报错，将该函数改造成了微任务。

## 九、async/await

  ES7 引入了 async/await，提供了在不阻塞主线程的情况下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。

**1）生成器**

  生成器（Generator）函数是一个带星号函数，而且可以暂停和恢复执行。

**2）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，就把 A 协程称为 B 协程的父协程。

  协程不是被操作系统内核所管理，而完全是由程序所控制（也就是在用户态执行）。这样带来的好处就是性能得到了很大的提升，不会像线程切换那样消耗资源。

**3）async/await**

  async/await 技术背后的秘密就是 Promise 和生成器应用，往低层说就是微任务和协程应用。

  根据 MDN 定义，async 是一个通过异步执行并隐式返回 Promise 作为结果的函数。

  当执行到await 100时，会默认创建一个 Promise 对象：promise\_。

  然后 JavaScript 引擎会暂停当前协程的执行，将主线程的控制权转交给父协程执行，同时会将 promise\_ 对象返回给父协程。

  接下来继续执行父协程的流程。随后父协程将执行结束，在结束之前，会进入微任务的检查点，然后执行微任务队列。

  最后触发 promise\_.then 中的回调函数，将主线程的控制权交给 foo 函数的协程，并同时将 value 值传给该协程，并继续执行后续打印语句。

## 十、页面性能

  通常一个页面有三个阶段：加载阶段、交互阶段和关闭阶段。 1. 加载阶段，是指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 JavaScript 脚本。 2. 交互阶段，主要是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是 JavaScript 脚本。 3. 关闭阶段，主要是用户发出关闭指令后页面所做的一些清理操作。

**1）白屏**

  从发起 URL 请求开始，到首次显示页面的内容，在视觉上经历的三个阶段。 1. 第一个阶段，等请求发出去之后，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。 2. 第二个阶段，提交数据之后渲染进程会创建一个空白页面，通常把这段时间称为解析白屏，并等待 CSS 文件和 JavaScript 文件的加载完成，生成 CSSOM 和 DOM，然后合成布局树，最后还要经过一系列的步骤准备首次渲染。 3. 第三个阶段，等首次渲染完成之后，就开始进入完整页面的生成阶段了，然后页面会一点点被绘制出来。

  要想缩短白屏时长，可以有以下策略： 1. 通过内联 JavaScript、内联 CSS 来移除这两种类型的文件下载，这样获取到 HTML 文件之后就可以直接开始渲染流程了。 2. 还可以尽量减少文件大小，比如通过 webpack 等工具移除一些不必要的注释，并压缩 JavaScript 文件。 3. 将一些不需要在解析 HTML 阶段使用的 JavaScript 标记上 sync 或者 defer。 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。

**2）加载阶段**

  把这些能阻塞网页首次渲染的资源称为关键资源，例如JavaScript、首次请求的 HTML 资源文件、CSS 文件。

  总的优化原则就是减少关键资源个数，降低关键资源大小，降低关键资源的 RTT 次数。

  RTT（Round Trip Time） 就是这里的往返时延。它是网络中一个重要的性能指标，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，总共经历的时长。 1. 将 JavaScript 和 CSS 改成内联的形式。如果 JavaScript 代码没有 DOM 或者 CSSOM 的操作，则可以改成 async 或者 defer 属性；同样对于 CSS，如果不是在构建页面之前加载的，则可以添加媒体取消阻止显现的标志。 2. 压缩 CSS 和 JavaScript 资源，移除 HTML、CSS、JavaScript 文件中一些注释内容，也可以通过取消 CSS 或者 JavaScript 中关键资源的方式。 3. 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。除此之外，还可以使用 CDN 来减少每次 RTT 时长。

**3）交互阶段**

  谈交互阶段的优化，其实就是在谈渲染进程渲染帧的速度，因为在交互阶段，帧的渲染速度决定了交互的流畅度。

  大部分情况下，生成一个新的帧都是由 JavaScript 通过修改 DOM 或者 CSSOM 来触发的。还有另外一部分帧是由 CSS 来触发的。

  一个大的原则就是让单个帧的生成速度变快。 1. 减少 JavaScript 脚本执行时间，不要一次霸占太久主线程。一种策略是将一次执行的函数分解为多个任务，另一种是把一些和 DOM 操作无关且耗时的任务放到 Web Workers 中去执行。 2. 避免强制同步布局。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。 3. 避免布局抖动。所谓布局抖动，是指在一次 JavaScript 执行过程中，多次执行强制布局和抖动操作。 4. 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。 5. 避免频繁的垃圾回收。要尽量避免产生那些临时垃圾数据。尽可能优化储存结构，尽可能避免小颗粒对象的产生。