基本知识

第一章 浏览器输入URL到页面展示，都经历了哪些过程?

这个问题在面试中，基本属于必问的了，鉴于在面试中的时间局限，可能大家也就聊一聊dns查询，http请求，tcp三次握手四次挥手，解析 HTML构建DOM树，计算DOM树上的CSS，合成图片，绘制到屏幕上。这个问题是一个很开放的问题，涉及的内容，都能够让我水一篇文章了。
https://juejin.cn/post/7004638318843412493#heading-6

这个问题属于老生常谈的经典问题了 下面给出面试**简单版**作答

1. 浏览器地址栏输入 URL 并回车
2. 浏览器查找当前 URL 是否存在缓存，并比较缓存是否过期
3. DNS 解析 URL 对应的 IP
4. 根据 IP 建立 TCP 连接（三次握手）
5. 发送 http 请求
6. 服务器处理请求，浏览器接受 HTTP 响应
7. 浏览器解析并渲染页面
8. 关闭 TCP 连接（四次握手）

一、打开浏览器

在输入url之前，总得先打开浏览器，不然url往哪儿输呢，chrome会启动多个进程：主进程、渲染进程、网络进程、插件进程和GPU进程等。

进程和线程

首先呢，进程是CPU分配资源的最小单位，线程是CPU调度任务的最小单位。打个比方，工作中的各种部门，划分资源的时候肯定是给前端部门多少人给后端多少人，觉得自己资源不够可以再申请；那线程呢，就是部门内的员工了，各个员工都共享一套代码，但都是给部门内干活，前端当然不能去写后端的接口，那不串了么。

在启动chrome时，主要是有下面的几个进程。

1. 主进程相当于行政部门，管理着地址栏、书签、后退和前进按钮界面，还有标签页的创建销毁等；
2. 插件进程，谁用浏览器还没几个插件呢，比如Adblock，一个插件一个进程，用的时候才会创建；
3. GPU进程，就一个，以前是用来绘制3D的，不过现在网页和界面也都交给了GPU进行绘制；
4. 网络进程，这是近两年才被独立出来的，以前是放在主进程内，主要负责网络资源的请求；
5. 渲染进程，就是咱们前端关注的重点了。

首先，打开一个标签页，chrome就会为其创建一个进程，这个进程实际上就是一个渲染进程副本，在这个进程内，会有各种各样的线程，gui线程啦，网络请求啦，事件线程啦，js内核线程(v8引擎)。

此外，还有一个备用渲染进程是常驻的，不过这跟备胎还是有区别的，这个渲染进程能扶正，之后再去创建个备用的。

输入URL

URL的全称是Uniform Resource Locator，统一资源定位符。就拿`https://www.example.com/index.html?uid=1#ch1`这个链接地址来说。

- `https://`是协议方案部分，其他的协议方案还有`ftp:`或`javascript:`；
- `www.example.com`这个是经过DNS解析后转成ip地址，当然咱们也可以不用域名，直接用ip访问服务器，开发中常用的`192.168.1.1`就是指向本机中的node服务器；
- `/index.html`就是咱们要拿到的html文档；
- `?uid=1`查询字符串，用来传参的；
- `#ch1`这个叫片段标识符，用来标记已获取资源中子资源的位置。hashrouter的实现就基于此。

二、网络请求

打开了浏览器，输入了URL，找到了出口，这不就得去请求资源了么。

在请求资源之前呢，还有个问题得说清楚喽。

前文不是说有单独的一个网络进程了么，那咱们这新开的标签页，也就是渲染进程，里也有http请求的线程，那这个输入URL后回车的网络请求，到底是由网络进程还是渲染进程发起的呢?

答案是由标签页自身的渲染进程发起的。而网络进程，主要是处理浏览器本身的网络请求，比如谷歌账号，比如下载。

端口号

对网络请求有些了解的朋友知道，这个网络请求都是从一个网线/wifi里边出去的，一个计算机的网络流量可大了，又不止浏览器在用网络，还有什么微信啊网抑云啊都在请求，那计算机怎么分辨这个请求应该给到谁呢?

想一想，外卖是怎么从楼下送到家门口的？是不是通过门牌号。对了，计算机里边也有门牌号，当然计算机不叫门牌号，叫端口号，渲染进程需要进行网络通信时就会向计算机申请端口，这里说的是端口可不是pid，这两个是不一样的。

那还有一个问题，现在家里都有大大小小的各种网络设备，计算机是怎么知道要把数据包发给路由器，而不是别的设备呢。就是上边提到的外卖例子，送上门之前，怎么找到楼下？地址，是的，每个设备都有一个地址，叫MAC地址，这个地址是唯一的，首先计算机在传输数据之前要进行广播，找到网关的MAC地址，就像商城找人那样，那商城也不知道人在哪儿，只能广播，等人回应了才行。

这样，浏览器找到了路由出口，可以开始向外传输数据了。

DNS解析

可是咱输入的是URL，服务器的IP地址是一组纯数字呀，这怎么访问呢，为了解决这个问题，DNS服务就应运而生了，可以通过域名或IP地址互查。因此呢，在填了URL之后，还不能去请求网络资源，还得先经过以下步骤，以下步骤是可以中断的，只要在任一阶段找到了IP地址，就会停止。

- chrome自己的DNS缓存
- 操作系统的缓存
- 本机host
- 向计算机系统配置的首选DNS服务器发起域名解析请求，比如114.114.114.114就是咱们国内电信移动联通的通用域名解析DNS服务器
- 电信运营商发起迭代DNS解析请求，先找到根域DNS的IP地址后发起请求，根域里边并不储存域名和IP地址的映射关系，返回的是域名后缀域的IP地址，比如example.com，根域返回的就是com域的IP地址。

- 运营商再次请求域名后缀域(com域)的DNS服务器，com域也不返回实际的IP地址，而是返回example.com这个域名的DNS地址，这个地址一般就是域名注册商提供的了，比如万网、新网。
- 域名注册商提供了IP地址后，返回给运营商，运营商再返回给操作系统，操作系统再给到浏览器。
- 如果都没有，操作系统会找到NetBios name Cache，这里存的主要是最近一段时间内和计算机成功通讯的IP地址。
- 查询WINS服务器
- 广播查找
- 读取LMHOSTS文件

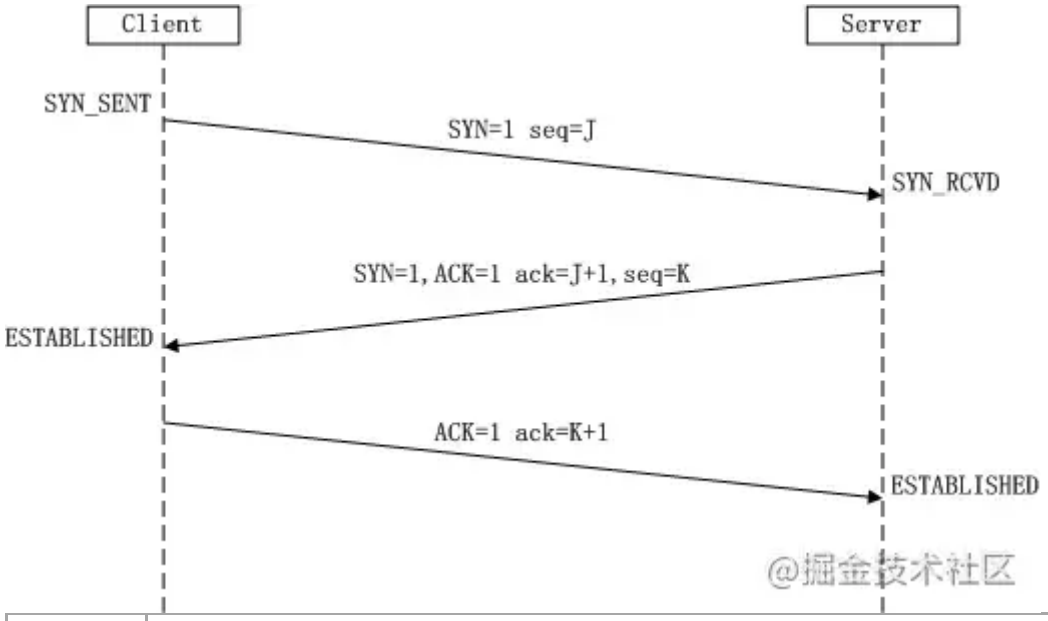
经过DNS解析之后，就拿到了目标服务器的IP地址，就可以快乐地请求数据了。当然如果都没有找到IP地址，那就宣告解析失败了。

TCP连接

一个请求发出去，大概会经历这么几个步骤，网卡=>路由器=>交换机=>服务器，中间还会各种转发，物理层面的链路，本文的重点也不在此处，就不展开讲了。在进行HTTP请求之前，要先和服务器建立连接，这里用到的是TCP协议，除此之外还有UDP，这两个协议的主要区别是：UDP不需要建立连接，想发就发，会有不可靠性，除此之外还有单播多播广播，面向报文之类的特性。

TCP链接是一个全双工的连接，全双工的意思是数据可以同时在两个方向传输。TCP和服务器建立连接的过程，就是面试中常常会被问到的TCP三次握手。

1. 第一次握手是客户端主动发起的，发送一个SYN(syn=j)的数据包给服务器，同时进入SYN_SENT的状态，等待服务器回复。
2. 服务器接收到这个请求之后，确认客户端的SYN(ack=j+1)，同时发送一个SYN(syc=x)包，此时服务器进入SYN_RECV的状态，这是第二次握手。
3. 最后一次握手则是客户端向服务器发送ACK包，进行状态确认，服务器和客户端都进入ESTABLISHED的状态，TCP连接完成。



字段	含义（ACK等大写字母表示标志位，其值为1或0，ack等小写的单词表示序号）
URG	紧急指针是否有效。为1，表示某一位需要被优先处理。
ACK	确认号是否有效，一般置为1。
PSH	提示接收端应用程序立即从TCP缓冲区把数据读走。
RST	对方要求重新建立连接，复位。
SYN	请求建立连接，并在其序列号的字段进行序列号的初始值设定。建立连接，设置为1。
FIN	希望断开连接。

这里有一个问题，为什么需要三次握手?

这是因为网络环境的复杂，丢包是很常见的事情，服务端向客户端发送确认回执的时候，如果数据包丢失了，那么客户端将一直是SYN_SENT的状态，而服务端将会认为连接已经建立并且向客户端发送数据，由于客户端认为连接还没有建立，就会忽略掉服务端发送来的数据，而服务端在发出数据超时后会不断进行重试，如此一来，就造成了死锁。

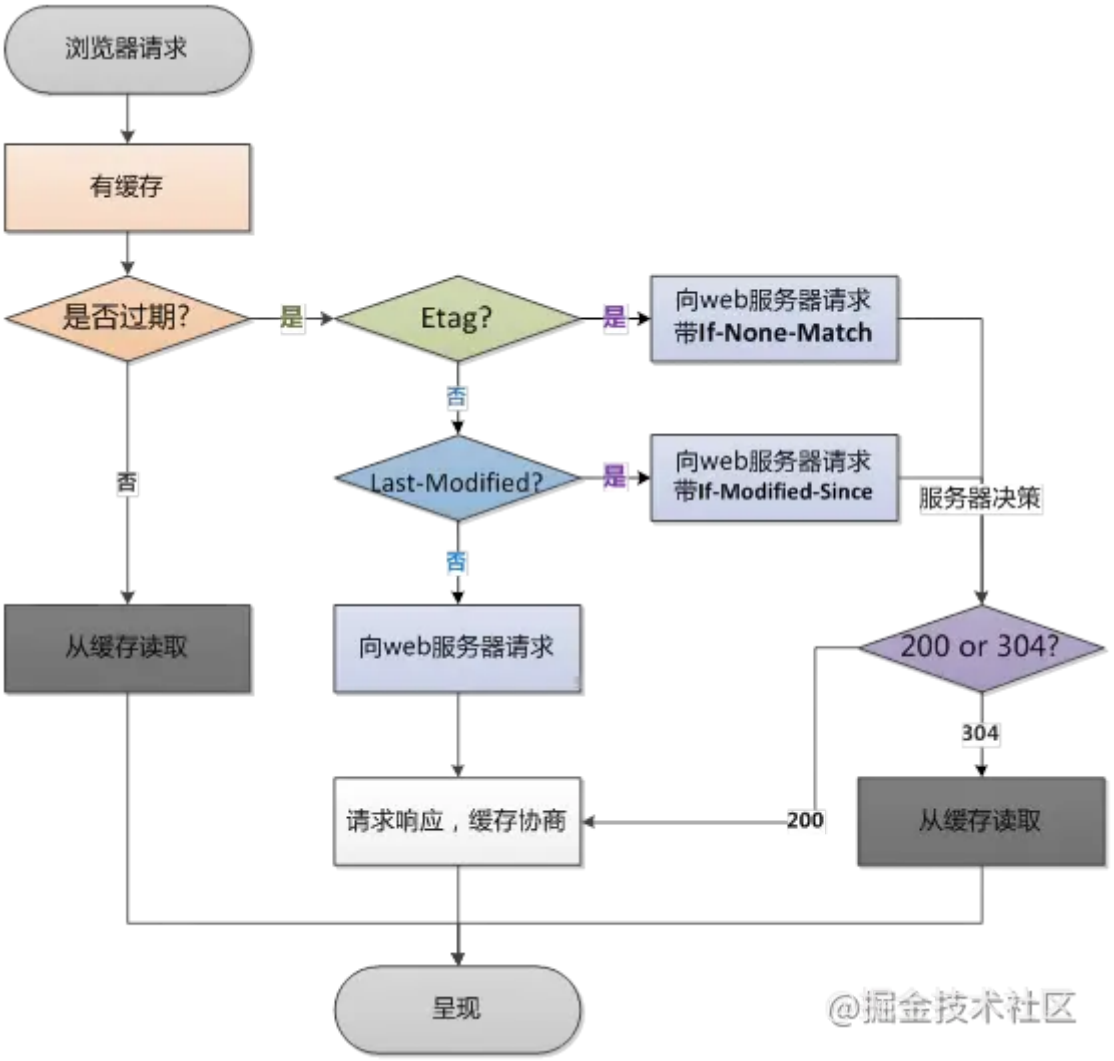
简单来说，就是服务端需要知道连接是否已经成功建立，而不是自己在那一厢情愿地发送数据包。

HTTP传输

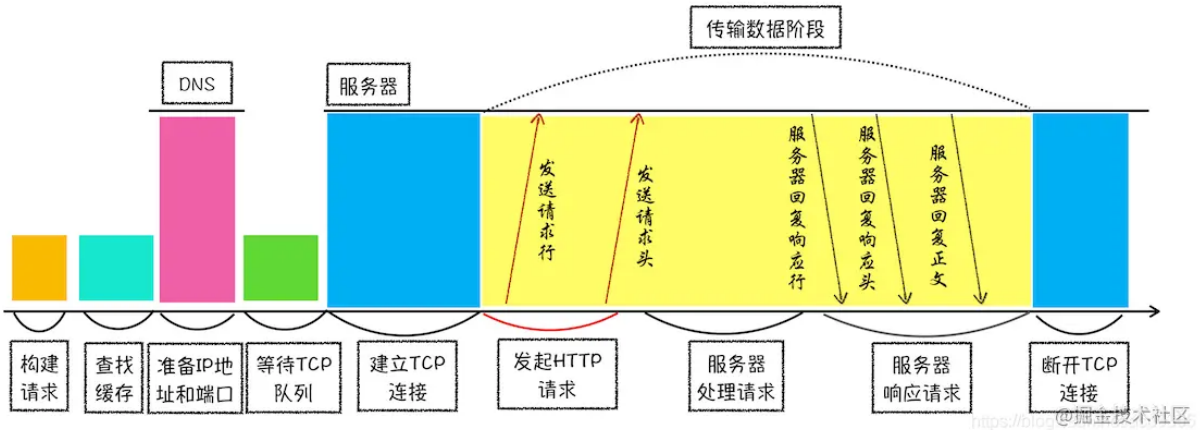
连接建立后，就要开始HTTP协议传输了，HTTP协议在TCP通道中传输的完全是文本，请求网页，用的都是GET请求。在初次连接的时候，会向服务器请求资源，如果有做CDN缓存的话，则会从CDN服务器中请求资源。

倘若我们之前已经访问过这个网页了，则会检查HTTP请求中的请求头设置，有一个Cache-Control的设置，倘若命中缓存且expires未过期，则从缓存中获取，此时是不需要从服务器获取资源，也不需要TCP连接的，此时HTTP的状态为200，这属于强缓存。

而协商缓存是，请求还是要发送到服务器，由服务器来决定缓存是否可用，相关字段有Last-Modified/If-Modified-Since和Etag/If-None-Match，服务器会优先验证ETag，一致的情况下，才会继续比对Last-Modified，最后决定是否返回304。



流水线过程



三、渲染页面

拿到了HTML文档之后，就要开始进行网页渲染了。前面所述的网络请求也是在当前标签页的渲染进程中完成的，之后，这个渲染进程会使用RCP通信和浏览器进程进行通信，通知浏

览器页面进行相应的变更，比如加载图标的停止。

编译

从HTML字符流中读取文档，先利用状态机进行token分词，同时构建DOM树，DOM树的栈顶一般是<html>这个元素，DOM元素都添加到树下，遇到属性就添加到当前节点，如果是文本内容，则分析当前节点是否是文本节点，否则就添加为子节点；除此之外，浏览器还要构建CSS RenderObject树，这两者是流式处理的，依次拿到DOM树构建的元素，然后按照CSS选择器的优先级进行匹配覆盖，之后会确定每个元素在文档中的位置，也就是进行排版，生成一个render树。

渲染

图形的渲染，在Chrome当中，是交给skia这个引擎来处理的，顺带一提，安卓和flutter的绘图引擎也是这个。skia绘制出来的实际上是图片，每张图片就是一帧。因此，展现在我们面前的网页，实际上是不断重新绘制渲染的图片。当然不是全图绘制，而是分区域分块，将需要更新的部分重新绘制。

由于在JS当中，是可以操作DOM元素的，这就会产生一个重绘和回流的概念。回流是指render树需要进行重新构建，比如修改了元素的display:none中的值，将元素显示出来，就会引发回流。而重绘，只是修改的了render tree中的值。回流必然会触发重绘，但无论是回流还是重绘，都会导致渲染引擎重绘合成位图。

同时，GUI和JS是互斥的，当JS引擎在执行的时候，GUI就会被挂起，GUI的更新会被保存在一个队列当中，等待JS空闲时立刻切换执行。这是为了保证在渲染引擎在获得的元素数据前后一致，避免JS和GUI同时操作一个元素。

任务队列

我们都知道，在整个页面被加载之后，就会触发window.onload这个钩子。在这之后，就是JS的事件逻辑了。JS是单线程的，所有的任务都要排队，但JS引擎完全可以不管IO(网络请求、鼠标点击等)，而是先处理其他的任务，等IO返回了结果之后，再处理挂起的任务。

三、结语

至此，这篇文章也就说完了，参考了很多的资料，其中李兵老师的《浏览器工作原理与实践》给了我很大的帮助，很值得一读。由于这个问题所涉及到的内容之多，核心的部分还是网络请求这块。至于后面提到的绘图渲染，那是计算机图形学的研究方向了。

第二章 Dom

获取元素，也叫标签

```
Window.idxxx 或者直接 idxxx
document.getElementById('idxxx')
document.getElementsByTagName('div')[0]
document.getElementsByClassName('classxxx')[0]
document.querySelector('#idxxx')
document.querySelectorAll('xxx')[0]
```

复制代码

- 现在常用*querySelector querySelectorAll*
- *getElement(s)Byxxx* 需要兼容 ie 才用

获取特定元素

```
// 1.获取html元素
document.documentElement
// 2.获取head元素
document.head
// 3.获取body元素
document.body
// 4.获取窗口
window
//5.获取所有元素
document.all // 它是一个falsy值
if(document.all){
  console('只能在ie运行')
}else{
  console('其他浏览器，非ie')
}
```

复制代码

节点的增删改查

增

- 创建一个标签节点

```
let div1 = document.creatElement('div')
document.creatElement('style')
document.creatElement('script')
document.creatElement('li')
```
复制代码

- 创建一个文本节点

```
text1 = document.creatTextNode('hellow')
```
复制代码

- 标签里面插入文本

```
div1.appendChild(text1)
div.innerText = 'hellow' | div1.textContent = 'hellow'
```
复制代码

删

old

```
parentNode.childChild(childNode)
```
复制代码

new

```
childNode.remove()
```
复制代码

```
// old
div1.parentNode.removeChild(div1)
// new
div1.remove()
// 让删除的节点回来
document.body.appendchild(div1)
// 彻底被干掉
div1 = null
```
复制代码

改

- 写标准属性

```
// 改class
div.className = 'pink red' // 全覆盖
div.classList.add('red')
// 改style
```

```
  div.style = 'width: 100px; color: blue;'
  // 改部分style
  div.style.width = '200px'
  // 大小写
  div.style.backgroundColor = '#e3e3e3'
  // 改data-*属性
  div.dataset.x = 'xxx'
```
复制代码

- 读标准属性

```
  div.classList / a.href
  div.getattribute('class') / a.getAttribute('href')
```
复制代码

- 改事件处理函数
    a. div.onclick 默认位 null
    - 默认点击div不会有任何事情发生
    - 如果吧 div.onclick 改为一个函数 fn
    - 那么点击div的时候，浏览器就会调用这个函数
    - 并且是这样调用的 fn.call(div, event)
    - div 会被当做 this
    - enent 则包含了点击事件的所有信息， 如坐标

    a. div.addEventListenter
    - 是 div.onclick 的升级版
- 改文本内容

```
  div.innerText = 'xxx'
  div.textcontent = 'xxx'
```
复制代码

- 改 HTML 内容

```
  div.innerHTML = '<strong> 注意 </strong>'
```
复制代码

- 改标签

```
  div.innerHTML = '' // 先清空
  div.appendChid(div2) // 再加内容
```
复制代码

- 改parent

```
  newParent.appendChild(div)
```
复制代码

查

```
  // 查爸爸
  node.parentNode | node.parentElement
  // 查爷爷
  node.parentNode.parentNode
  // 查子代
  node.childNodes // 子代node下有text和element
  node.children // 子代的element
  // 查兄弟姐妹
  node.parentNode.childNodes // 还要排除自己
  node.parentNode.children //还有排除自己
  node.firstChild // 老大
  node.lastChild // 老小
  node.previousSibling // 查哥哥/姐姐
  node.nextString // 查下一个弟弟/妹妹
```
复制代码

- 遍历一个 div 里面的所有元素

```
  travel1 = (node, fn) => {
    fn(node)
    if(node.children){
      for(let i=0; i<node.children.length; i++){
        travel(node.children[i], fn)
      }
    }
  }
  travel(div1, (node) => console.log(node))
```
复制代码

对象风格封装DOM

增

```
  dom.create(`<div></div>`) // 用于创建节点
  dom.after(node, node2) // 用于新增弟弟
  dom.before(node, node2) // 用于新增哥哥
```

```
  dom.append(parent, child) // 用于新增儿子
  dom.wrap(`<div></div>`) // 用于新增爸爸
```
复制代码

删

```
  dom.remove(node) // 用于删除节点
  dom.empty(parent) // 用于删除后代
```
复制代码

改

```
  dom.attr(node, 'title', ?) // 用于读写属性
  dom.text(node, ?) // 用于读写文本内容
  dom.html(node, ?) // 用于读写 html 内容
  dom.style(node, {color: 'red'}) // 用于修改 style
  dom.class.add(node, 'pink') // 用于添加 class
  dom.class.remove(node, 'pink') // 用于删除 class
  dom.on(node, 'click', fn) // 用于添加事件监听
  dom.off(node, 'click', fn) // 用于删除事件监听
```
复制代码

查

```
  dom.find('选择器') // 用于获取标签或标签们
  dom.parent(node) // 用于获取父元素
  dom.children(node) // 用于获取子元素
  dom.siblings(node) // 用于获取兄弟姐妹
  dom.next(node) // 用于获取弟弟
  dom.previoous(node) // 用于获取哥哥
  dom.each(nodes.fn) // 用于遍历所有节点
  dom.index(node) // 用于获取排行老几
```
复制代码

链式风格封装DOM(jQuery风格)

查

```
  jQuery('#xxx') // 返回值并不是元素，而是一个api对象
  jQuery('#xxx').find('.red') // 查找#xxx里的.red元素
  jQuery('#xxx').parent(node) // 用于获取父元素
  jQuery('#xxx').children(node) // 用于获取子元素
```

```
  jQuery('#xxx').siblings(node) // 用于获取兄弟姐妹
  jQuery('#xxx').next(node) // 用于获取弟弟
  jQuery('#xxx').prev(node) // 用于获取哥哥
  jQuery('#xxx').each(nodes.fn) // 用于遍历所有节点
  jQuery('#xxx').index(node) // 用于获取排行老几
  jQuery('.red').each(fn) // 遍历并对每个元素执行 fn
```

复制代码

增

```
  // jQuery('#xxx') 可以写成 $('#xxx')
  $('<div><span> 1 </span></div>') // 创建div
  .appendTo(document.body) //插入到body中
```

复制代码

删

```
  $div.remove()
  $div.empty()
```

复制代码

改

```
  $div.attr('title', ?) // 用于读写属性
  $div.text(?) // 用于读写文本内容
  $div.html(?) // 用于读写 html 内容
  $div.css({color: 'red'}) // 用于修改 style
  $div.addClass('pink') // 用于添加 class
  $div.on('click', fn) // 用于添加事件监听
  $div.off('click', fn) // 用于删除事件监听
```