# \*\*\*\*\*\*\*\*JS模块化理解\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# 1.什么是模块化

将一个复杂的程序依据一定的规则(规范)封装成几个块(文件), 并进行组合在一起

模块的内部数据/实现是私有的, 只是向外部暴露一些接口(方法/数据)与外部其它模块通信

# 2.模块化的好处

避免命名冲突(减少命名空间污染)

更好的分离, 按需加载

更高复用性

高可维护性

# \*\*\*\*\*\*\*\*模块化规范\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# **1.commonjs的模块化规范**

nodejs平台使用模块化规范

***1. 引入模块  
 require(模块路径)  
 模块路径必须以 ./ 或 ../ 开头,否则就报错：cannot find module 'xxx'  
 模块的文件后缀名可以省略,因为解析时会自动补全： js文件和json文件,其他文件就不行***

***引入其他两个模块,使用  
 const m1 = require('./module1');  
 const m2 = require('./module2');  
 const m3 = require('./module3');  
2. 暴露模块  
 module.exports  
 module.exports = sum; --> sum  
 module.exports.sum = sum; --> {sum: sum}  
 exports  
 exports.add = add; --> {add: add}  
  
 模块向外暴露出去的是exports还是module.exports?  
 暴露的是module.exports 指向的值  
 exports 是 module.exports 的引用  
  
 module.exports = {}; //默认值  
 exports = module.exports;  
  
 exports.add = add;  
 exports = add; // 为什么不行？ exports不再是module.exports的***

***引用了,引入时读取不到,报错***

***当模块内部只有一项内容需要向外暴露出去,用module.exports = xxx;  
当模块内部有多项内容需要向外暴露出去,用module.exports = {xxx, yyy};***

***1. 问题：  
 浏览器不能识别commonjs模块化语法,只有nodejs平台识别  
 但是希望浏览器能识别  
2. 解决：  
 通过某个工具将commonjs模块化语法编译成浏览器能识别的js语法  
 工具：browserify  
 下载并安装： npm i browserify -g 在任意cmd窗口运行当前指令  
 使用：  
 - 来到当前文件的目录  
 - 运行 browserify ./app.js -o ./built.js***

# 2.ES6模块化规范

浏览器使用的模块化规范

***1. 引入模块  
 import  
 import xxx from '模块路径';  
 import { xxx, yyy } from '模块路径';  
  
 模块路径的规则与commonjs一致：  
 我们使用babel将ES6模块化语法编译成Commonjs模块化语法,这时nodejs平台能识别  
 如果需要浏览器识别,需要借助browserify工具将Commonjs模块化语法编译浏览器能识别的语法***

***2. 暴露模块  
 export  
 - 默认暴露  
 export default xxx; --> import xxx from '模块路径';  
 - 分别暴露  
 export const a = xxx;  
 export const b = function () {}; --> import { xxx, yyy } from '模块路径';  
 - 统一暴露  
 export {xxx, yyy}; --> import { xxx, yyy } from '模块路径';  
  
 ES6模块化规范问题：  
 nodejs平台不识别这个语法  
 浏览器平台也不识别这个语法  
   
 babel -- 专门用来编译语法（将高级语法/浏览器不识别的语法编译成浏览器能识别的低级语法）  
 借助 babel 将ES6模块化语法编译成 commonjs 语法,这个语法只能 nodejs 平台识别  
 借助 browserify 将 commonjs 语法编译成浏览器能识别的语法***

***用法：  
 0. 来到当前工作根目录  
 04\_ES6\_Babel\_Browserify  
 1. 初始化package.json  
 npm init -y  
 2. 下载并安装babel相关的包  
 npm install --save-dev @babel/core @babel/cli @babel/preset-env  
 3. 创建 .babelrc 配置文件（这个文件就是babel运行时读取的配置文件）  
 {  
 "presets": ["@babel/env"]  
 }  
 4. 运行指令：将ES6模块化语法编译成浏览器识别的语法  
 npx babel src -d dist  
 将src下面的所有js文件编译输出到dist目录下  
 browserify dist/app.js -o dist/built.js***

# \*\*\*\*\*\*\*\*性能优化\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

## 性能优化是什么？  
\* 优化网站性能,提升用户体验.  
## 为什么要性能优化？  
\* 从用户角度而言,优化能够让页面加载得更快、对用户的操作响应得更及时,能够给用户提供更为友好的体验.   
\* 从服务商角度而言,优化能够减少页面请求数、或者减小请求所占带宽,能够节省可观的资源.  
## 怎么进行性能优化？能优化哪些内容？  
\* 主要优化目标  
 \* 减少http请求数量  
 \* 减少请求资源文件大小  
\* 优化具体手段  
 \* 图片优化  
 \* 压缩css/js/html资源  
 \* 合并请求数量  
 \* ...

# 1.图片性能优化

1.1图片类型和特点  
1. jpg：有损压缩,不支持透明  
2. png：无损压缩,支持透明  
 \* png8 - 2^8色 + 支持透明  
 \* png24 - 2^24色 + 不支持透明  
 \* png32 - 2^32色 + 支持透明   
3. gif：动态图,无损压缩,支持透明  
4. webp：谷歌推出,压缩程度更好,不支持透明

5. icon: 字体图标

6. base64: 把图片转换成代码格式

1.2 图片使用场景  
\* jpg -- 大部分不需要透明图片的业务场景  
\* png -- 大部分需要透明图片的业务场景  
\* webp -- chrome 安卓 火狐  
\* gif -- 动态图业务场景  
\* svg矢量图 -- 图片样式相对简单的业务场景

\* 精灵图 -- 小图颜色丰富  
\* 字体图标 -- 页面图标颜色单一  
\* base64 -- 12kb以下图片

1.3优化措施  
\* css雪碧图/精灵图  
 \* 过去将多个小图标合并到一 大图中, 现在使用较少  
 \* 优点：减少请求数量  
 \* 缺点：图片较大,加载速度变慢,如果没加载回来问题很严重  
\* 图片inline  
 \* 以图片base64方式处理图片  
 \* 优点：不用发送格外请求,随着html或者css文件的加载而加载  
 \* 缺点：大小会变大,所以不适用于很大的图片,一般10KB以下  
 \* 适用于固定不变的小图  
\* 使用矢量图  
 \* 使用svg标签  
 \* 使用iconfont解决icon问题  
 \* 优点：随着加载页面字体,加载好图片,请求数量较少  
 \* 缺点：如果页面用不上字体,需要额外的开销加载字体图标  
\* 压缩图片  
 \* https://tinypng.com/ 在线压缩图片

# 2.图片懒加载与预加载

## 2.1图片懒加载

<**img src="./imgs/loading.gif" data-src="./imgs/1.jpg" class="lazy-load" alt="img"**>  
<**img src="./imgs/loading.gif" data-src="./imgs/2.jpg" class="lazy-load" alt="img"**>  
<**img src="./imgs/loading.gif" data-src="./imgs/3.jpg" class="lazy-load" alt="img"**>  
<**img src="./imgs/loading.gif" data-src="./imgs/4.jpg" class="lazy-load" alt="img"**>  
<**img src="./imgs/loading.gif" data-src="./imgs/5.jpg" class="lazy-load" alt="img"**>  
<**img src="./imgs/loading.gif" data-src="./imgs/6.jpg" class="lazy-load" alt="img"**>

......

***// 视口的高度*const *viewHeight* = document.documentElement.clientHeight;  
*// 一上来需要手动触发lazyLoad: 目的为了第一次加载的时候出现在可视区域内的图片能够加载到  
lazyLoad*();**

**function *lazyLoad*() {  
 const imgs = document.querySelectorAll('.lazy-load'); *// dom2* imgs.forEach((img) => {  
 *// 当前图片距离视口的top值* const { top } = img.getBoundingClientRect();  
 *// 判断图片是否出现在可视区域内* if (top < *viewHeight*) {  
 *// 图片出现在可视区域内,加载图片真实地址  
 // 通常来讲 data-xxx 使用用户自定义属性（用来保存数据,浏览器不能识别）.  
 // 读取 data-xxx 属性的值,需要通过 dom.dataset.xxx 读取对应自定义属性的值* const src = img.dataset.src;  
 *// 将图片src属性修改为图片真实地址* img.src = src;  
 *// 删除dom元素上的属性* img.removeAttribute('class');  
 img.removeAttribute('data-src');  
 }  
 })  
}  
*// 给document绑定滚轮事件*document.onscroll = *lazyLoad*;**

## 2.2图片预加载

***// 预加载：提取请求相应的资源*var *image* = new Image();  
*// 一旦给图片设置src属性,就会发送请求请求图片  
image*.src = './imgs/1.jpg';  
document.getElementById('btn').onclick = function () {  
 var image = new Image();  
 *// 加载重复的图片会直接找缓存* image.src = './imgs/1.jpg';  
 document.body.appendChild(image);  
}**

## 2.3 webp格式图片的处理

**<img src="./img/1.jpg\_\_.webp" alt="img">**

**const *img* = document.querySelector('img');  
*// 图片加载成功触发的事件(可不写)  
img*.onload = function () {  
 console.log('图片加载成功~');  
}  
*// 图片加载失败触发的事件  
img*.onerror = function () {  
 console.log('图片加载失败,当前浏览器不支持webp格式图片');  
 *img*.src = *img*.src.slice(0, -7);  
}**

# 3.浏览器基本渲染原理

## 3.1浏览器渲染引擎

### 3.1.1主要模块

一个渲染引擎主要包括：HTML解析器,CSS解析器,javascript引擎,布局layout模块,绘图模块

1.HTML解析器：解释HTML文档的解析器,主要作用是将HTML文本解释成DOM树.

2.CSS解析器：级联样式表的解析器,它的作用是为DOM中的各个元素对象计算出样式信息,为布局提供基础设施

3.Javascript引擎：使用Javascript代码可以修改网页的内容,也能修改css的信息,javascript引擎能够解释javascript代码,并通过DOM接口和CSSOM接口来修改网页内容和样式信息,从而改变渲染的结果.

4.布局（layout）：在DOM创建之后,Webkit需要将其中的元素对象同样式信息结合起来,计算他们的大小位置等布局信息,形成一个能表达这所有信息的内部表示模型

5.绘图模块（paint）：使用图形库将布局计算后的各个网页的节点绘制成图像结果

***以上这些模块依赖很多其他的基础模块,包括要使用到网络 存储 2D/3D图像 音频视频解码器 和 图片解码器. 所以渲染引擎中还会包括如何使用这些依赖模块的部分.***

### 3.1.2渲染过程

浏览器渲染页面的整个过程：

浏览器会从上到下解析文档.

1.遇见 HTML 标记,调用HTML解析器解析为对应的 token （一个token就是一个标签文本的序列化）并构建 DOM 树（就是一块内存,保存着tokens,建立它们之间的关系）.

2.遇见 style/link 标记 调用css解析器 处理 CSS 标记并构建 CSSOM 树.

3.遇见 script 标记 调用 javascript解析器 处理script标记,绑定事件、修改DOM树/CSSOM树 等

4.将 DOM 与 CSSOM 合并成一个渲染树.

5.根据渲染树来布局,以计算每个节点的几何信息.

6.将各个节点绘制到屏幕上.

***需要明白,这五个步骤并不一定按顺序执行完成.如果 DOM 或 CSSOM 被修改,以上过程需要重复执行,这样才能计算出哪些像素需要在屏幕上进行重新渲染. 实际页面中,CSS 与 JavaScript 往往会多次修改 DOM 和 CSSOM***

### 3.1.3阻塞渲染

#### css阻塞

css 在head中引入会阻塞页面的渲染 为什么？ 避免闪屏现象

css 阻塞js的执行,不阻塞js等其他资源的加载 为什么？ 脚本在文档解析阶段会请求样式信息.如果当时还没有加载和解析样式, 脚本就会获得错误的回复,这样显然会产生很多问题.这看上去是一个非典型案例, 但事实上非常普遍.Firefox 在样式表加载和解析的过程中,会禁止所有脚本. 而对于 WebKit 而言,仅当脚本尝试访问的样式属性可能受尚未加载的样式表影响时,它才会禁止该脚本.

#### js阻塞

直接引入的js会阻塞页面的渲染 为什么？ Javascript代码可能会修改DOM树的结构

js顺序执行,阻塞后续js逻辑的执行,不阻塞js等其他资源的加载 为什么？ 维护依赖关系

预解析 WebKit 和 Firefox 都进行了这项优化.在执行js脚本时,其他线程会解析文档的其余部分, 找出并加载需要通过网络加载的其他资源.通过这种方式,资源可以在并行连接上加载, 从而提高总体速度.请注意,预解析器不会修改 DOM 树,而是将这项工作交由主解析器处理； 预解析器只会解析外部资源（例如外部脚本、样式表和图片）的引用.

***在上述的过程中,网页在加载和渲染过程中会发出“DOMContentloaded”和“onload”事件 分别在DOM树构建完成之后,以及DOM树构建完并且网页所依赖的资源都加载完之后发生、 因为某些资源的加载并不会阻碍DOM树的创建,所以这两个事件多数是不同时发生的***

#### 从DOM树到可视化图像

1) CSS文件被CSS解析器解释成内部表示结构(CSSDOM)

2) CSS解析器工作完成之后,在DOM树上附加解释后的样式信息,这就是RenderObject树

3) RenderObject在创建的同时,Webkit会根据网页的结构创建RenderLayer,同时构建一个绘图上下文

4) 根据绘图上下文生成最终的图像（这一过程需要依赖图形库）

***上面介绍的是一个完整的渲染过程,但现代网页很多都是动态的,这意味着在渲染完成之后,由于网页的动画或者用户的交互, 浏览器其实一直在不停地重复执行渲染过程.（重绘重排）,以上的数字表示的是基本顺序,这不是严格一致的, 这个过程可能重复也可能交叉***

## 3.2重绘重排

### 3.2.1浏览器实际工作

在渲染DOM的时候,浏览器所做的工作实际上是：

获取DOM后分割为多个图层

对每个图层的节点计算样式结果 （Recalculate style--样式重计算）

为每个节点生成图形和位置 （Layout--重排）

将每个节点绘制填充到图层位图中 （Paint--重绘）

复合多个图层到页面上生成最终屏幕图像 （Composite Layers--图层重组）

### 3.2.2重绘(Repaint)

重绘是一个元素外观的改变所触发的浏览器行为,例如改变outline、背景色等属性.

浏览器会根据元素的新属性重新绘制,使元素呈现新的外观.

重绘不会带来重新布局,所以并不一定伴随重排.

### 3.2.3重排(Reflow 回流)

渲染对象在创建完成并添加到渲染树时,并不包含位置和大小信息.计算这些值的过程称为布局或重排

"重绘"不一定需要"重排",比如改变某个网页元素的颜色,就只会触发"重绘",不会触发"重排",因为布局没有改变.

但是,"重排"必然导致"重绘",比如改变一个网页元素的位置,就会同时触发"重排"和"重绘",因为布局改变了.

### 3.2.4触发重绘的属性

\* color \* background \* outline-color

\* border-style \* background-image \* outline

\* border-radius \* background-position \* outline-style

\* visibility \* background-repeat \* outline-width

\* text-decoration \* background-size \* box-shadow

### 3.2.5触发重排(回流)的属性

盒子模型相关属性会触发重布局 定位属性及浮动： 改变节点内部文字结构:

\* width \* top \* text-align

\* height \* bottom \* overflow-y

\* padding \* left \* font-weight

\* margin \* right \* overflow

\* display \* position \* font-family

\* border-width \* float \* line-height

\* border \* clear \* vertival-align

\* min-height \* white-space

### 3.2.6常见的触发重排的操作

Reflow 的成本比 Repaint 的成本高得多的多.DOM Tree 里的每个结点都会有 reflow 方法,

一个结点的 reflow 很有可能导致子结点,甚至父点以及同级结点的 reflow.在一些高性能的电脑上也许还没什么,

但是如果 reflow 发生在手机上,那么这个过程是非常痛苦和耗电的.

所以,下面这些动作有很大可能会是成本比较高的.

当你增加、删除、修改 DOM 结点时,会导致 Reflow , Repaint.

当你移动 DOM 的位置

当你修改 CSS 样式的时候.

当你 Resize 窗口的时候（移动端没有这个问题）

当你修改网页的默认字体时.

获取某些属性时(width,height...)

注：display:none 会触发 reflow,而 visibility:hidden 只会触发 repaint,因为没有发生位置变化.

### 3.2.7优化手段

1.元素位置移动变换时尽量使用CSS3的transform来代替对top left等的操作

2.将多次改变样式属性的操作合并成一次操作

不要一条一条地修改DOM的样式,预先定义好class,然后修改DOM的className

3.将DOM离线后再修改

由于display属性为none的元素不在渲染树中,对隐藏的元素操作不会引发其他元素的重排.

如果要对一个元素进行复杂的操作时,可以先隐藏它,操作完成后再显示.这样只在隐藏和显示时触发2次重排.

4.不要把某些DOM节点的属性值放在一个循环里当成循环的变量

当你请求向浏览器请求一些 style信息的时候,就会让浏览器刷新队列,比如：

offsetTop, offsetLeft, offsetWidth, offsetHeight

scrollTop/Left/Width/Height

clientTop/Left/Width/Height

width,height

当你请求上面的一些属性的时候,浏览器为了给你最精确的值,需要刷新队列,

因为队列中可能会有影响到这些值的操作.即使你获取元素的布局和样式信息跟最近发生或改变的布局信息无关,浏览器都会强行刷新渲染队列.

5.动画实现过程中,启用GPU硬件加速

transform: translateZ(0)

6.利用文档碎片

7.不要使用table布局

## **3.3window.requestAnimationFrame(DOMHighResTimeStamp)**

方法告诉浏览器,您希望执行动画,并请求浏览器在下一次重绘之前调用指定的函数来更新动画

该方法使用一个回调函数作为参数,这个回调函数会在浏览器重绘之前调用.

参数：DOMHighResTimeStamp,指示requestAnimationFrame() 开始触发回调函数的当前时间

返回值：

一个 long 整数,请求 ID ,是回调列表中唯一的标识.是个非零值,没别的意义.

你可以传这个值给 window.cancelAnimationFrame() 以取消回调函数.

取消方法：

window.cancelAnimationFrame(requestID)

requestID是先前调用window.requestAnimationFrame()方法时返回的ID.

# 4.DNS&CDN

## 4.1 Http协议

HTTP/0.9

HTTP/1.0

HTTP/1.1（主流）

HTTP/2.0

HTTP协议是在TCP/IP协议族的基础上运作起来的,用于客户端和服务端之间的通信

其属于TCP/IP协议族内的一个子集；HTTP协议是无状态的协议.

## 4.2 DNS域名解析服务

DNS服务是和HTTP协议一样位于应用层的协议,它提供域名到IP地址之间的解析服务.

计算机既可以被赋予IP地址,也可以被赋予主机名和域名.比如www.atguigu.com

用户通常使用主机名或域名来访问对方的计算机,而不是通过IP地址访问,因为与IP地址的一组纯数字相比,用域名来指定计算机名更符合人类的记忆习惯.

但是要让计算机去理解域名,相对而言就变得困难了,因为计算机更擅长处理一长串数字.

为了解决上述问题,DNS服务应运而生.NDS协议提供通过域名查找IP,或逆向从IP地址反查域名的服务时

chrome://dns/ :查看浏览器上的DNS缓存

ipconfig /displaydns :查看操作系统中的DNS缓存

## 4.3 DNS解析过程

查找浏览器缓存. 浏览器会检查缓存中有没有这个域名对应的解析过的IP地址,如果缓存中有,这个解析过程就将结束.

查找系统缓存 如果用户的浏览器缓存中没有,浏览器会查找操作系统缓存中是否有这个域名对应的DNS解析结果

查找路由器缓存. 如果系统缓存中也找不到,那么查询请求就会发向路由器,它一般会有自己的DNS缓存.

查找ISP DNS 缓存.(网络运营商) 运气实在不好,就只能查询ISP DNS 缓存服务器了.在我们的网络配置中都会有"DNS服务器地址"这一项, 操作系统会把这个域名发送给这里设置的DNS,也就是本地区的域名服务器,这个专门的域名解析服务器性能都会很好, 它们一般都会缓存域名解析结果,当然缓存时间是受域名的失效时间控制的.大约80%的域名解析都到这里就已经完成了, 所以ISP DNS主要承担了域名的解析工作.

递归搜索 最无奈的情况发生了, 在前面都没有办法命中的DNS缓存的情况下 (1) 本地 DNS服务器即将该请求转发到互联网上的根域 (2) 根域将所要查询域名中的顶级域（即blog.baidu.com中的com)的服务器IP地址返回到本地DNS. (3) 本地DNS根据返回的IP地址,再向顶级域（就是com域）发送请求. (4) com域服务器再将域名中的二级域（即blog.baidu.com中的baidu）的IP地址返回给本地DNS. (5) 本地DNS再向二级域发送请求进行查询. (6) 之后不断重复这样的过程 直到本地DNS服务器得到最终的查询结果,并返回到主机.这时候主机才能通过域名访问该网站.

## 4.4优化DNS

目的：减少DNS查询

一个多资源的站点最好使用2到4个不一样的主机来存放服务端资源. 这是在减少DNS查询和允许高度并行下载之间作出的最好权衡 (高度并行下载,浏览器一次能并发加载的量是受域名控制的)

使用Keep-alive进行持久连接

## 4.5内容发布网络CDN（Content Delivery Networks）

CDN是一组分布在多个不同地理位置的web服务器,用于更加有效的向用户发布内容

实现基本思路：

尽可能避开互联网上有可能影响数据传输速度和稳定性的瓶颈和环节,使内容传输的更快、更稳定.

通过在网络各处放置节点服务器所构成的在现有的互联网基础之上的一层智能虚拟网络,

CDN系统能够实时地根据网络流量和各节点的连接、负载状况以及到用户的距离和响应时间等综合信息

将用户的请求重新导向离用户最近的服务节点上.

# 5.浏览器存储

## **5.1浏览器存储**

SessionStorage, LocalStorage, Cookie这三者都可以被用来在浏览器端存储数据,而且都是字符串类型的键值对！

(另外一种存储模式叫：session;这种级别的存储属于服务端会话级别的存储)

## **5.2 Web Storage**

SessionStorage和LocalStorage都是本地存储,不会被发送到服务器上.同时空间比Cookie大很多,一般支持5-10M

浏览器端通过 Window.sessionStorage 和 Window.localStorage 属性来实现本地存储机制.

Storage.getItem('key');

该方法接受一个键名作为参数,返回键名对应的值.

Storage.setItem('key', 'value');

该方法接受一个键名和值作为参数,将会把键值对添加到存储中,如果键名存在,则更新其对应的值.

Storage.removeItem('key');

该方法接受一个键名作为参数,并把该键名从存储中删除.

Storage.clear()

调用该方法会清空存储中的所有键名

storage事件：

Storage 对象发生变化时（即创建/更新/删除数据项时,重复设置相同的键值不会触发该事件,Storage.clear() 方法至多触发一次该事件）会触发

在同一个页面内发生的改变不会起作用

在相同域名下的其他页面发生的改变才会起作用.(修改的页面不会触发事件,与它共享的页面会触发事件)

区别

LocalStorage是浏览器本地持久化存储技术,也叫永久存储

SessionStorage是浏览器本地临时存储技术,也叫会话存储

跨页面通信：  
 A页面：相办法将数据发送到B页面中  
 B页面：接受A页面发送的数据

**const *person* = {name: 'jack', age: 18};  
*// 将对象数据妆化为json数据* const *data* = *JSON*.stringify(*person*);  
*// 存储的数据必须为字符串类型才能存储  
// 如果不是会强制转换为字符串,再不行就会报错* sessionStorage.setItem('USER1', *data*);  
 sessionStorage.setItem('USER2', *data*);  
*// 读取本地存储的数据* const *data* = sessionStorage.getItem('USER1');  
 console.log(*data*);  
 const *person* = *JSON*.parse(*data*);  
 console.log(*person*);  
*// 删除本地储存数据* sessionStorage.removeItem('USER1');  
*// 清除所有数据* sessionStorage.clear();  
  
let *num* = 0;  
*// 跨页面通信***

***//a.html*document.getElementById('btn').onclick = function () {  
 localStorage.setItem('DATA', ++*num*);  
}**

***//b.html***

***// storage事件*window.addEventListener('storage', function (e) {  
 */\*  
 key: "DATA"  
 newValue: "5"  
 oldValue: "4"  
 \*/* console.log(e.key);  
 console.log(e.newValue);  
 console.log(e.oldValue)  
})**

## **5.3 IndexDB和WebSQL**

IndexedDB 和 WebSQL 都是一种低级API,用于客户端存储大量结构化数据.

该API使用索引来实现对该数据的高性能搜索.

不同的是IndexedDB是非关系型,而WebSQL是关系型.

WebSQL官方不在维护,但兼容性较好

IndexedDB在维护,兼容性较差

# \*\*\*\*\*\*\*\*nodejs\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# 1.执行nodejs代码

***如何执行nodejs代码：  
 1. 右键+run  
 2. 所有三角箭头点击运行  
 3. 在cmd/terminal中,来到当前的文件目录,运行 node 文件名***

***解决nodejs代码提示： file -- settings -- 在上面搜nodejs -- 打个勾/点击按钮***

***所有模块化的模块都被包裹一层函数中：  
 那么Commonjs和ES6模块化有没有函数呢？  
 答案：有 . 怎么样才能看到这个函数？***

***// 在函数内部打印当前函数  
console*.log(arguments.callee.toString());**

***function (exports, require, module, \_\_filename, \_\_dirname) {}  
 exports 暴露模块  
 require 引入模块  
 module module.exports 暴露模块  
 \_\_filename 当前文件的绝对路径  
 \_\_dirname 当前文件的目录的绝对路径***

# 2.js组成

***浏览器中js由什么组成：  
 DOM 文档对象模型 -- document 操作DOM  
 BOM 浏览器对象模型 -- window 操作浏览器  
 ESMA SCRIPT规范 -- 详细的js语法规范***

***nodejs中js包含哪些方面：  
 不包含DOM  
 基本不包含BOM,但少量的一些好用的方法保留下来了  
 console  
 setTimeout / setInterval  
 基本包含全部的ECMA SCRIPT规范***

***nodejs中全局对象叫 global  
 console  
 Buffer  
 process.nextTick 立即执行函数  
 setTimeout / clearTimeout  
 setInterval / clearInterval  
 setImmediate / clearImmediate 立即执行函数***

# 3.nodejs事件轮询机制

***nodejs的事件轮询机制：  
 概念：用来执行处理异步代码  
 分为6个阶段：  
 1. timers定时器阶段  
 执行定时器到点的回调函数（所有定时器setTimeout / setInterval的回调函数都在这个阶段执行）  
 2. idle,prepare准备阶段  
 TCP错误回调  
 3. 系统阶段  
 系统内部使用  
 4. poll轮询阶段  
 会执行绝大部分的回调函数,里面有一个回调队列,这个队列就保存着所有要执行的回调函数  
 查看回调队列是否有回调函数  
 有  
 依次取出,同步执行.  
 直到全部执行完毕,或者系统出错  
 没有  
 如果之前设置过 setImmediate 函数,就回去第五个阶段,如果之前没有设置过,就一直在第四阶段（当前阶段）停留  
 如果定时器到点了,为了去第一阶段执行定时器的回调函数,它也得去第五阶段  
 5. check检查阶段  
 只负责执行 setImmediate 的回调函数  
 6. 关闭阶段  
 执行关闭事件的回调函数 close end ...  
  
 process.nextTick 能在任意阶段,优先执行  
  
 事件轮询就是反复执行这6个阶段的函数.从第一个阶段开始,到第六个阶段结束,再重新执行第一个阶段....***

# 5.包-package

***package.json中5个字段：  
 name: 包名 今后下载时输入名称 （注意：要与下载的包名不一样）  
 version：版本号 x.x.x  
 例如 1.2.3  
 1 大版本：当这个包有巨大内容变化时（里面核心代码发生巨大变化）  
 2 中版本: 在当前大版本基础上,添加了一个功能（删除）  
 3 小版本：修复BUG  
 dependence: 生产依赖： jQuery bootstrap  
 devDependence：开发依赖: babel browserify  
 script 指令：里面包含运行项目的指令  
 script : {  
 "start" : "xxxx", 运行项目的指令  
 "build" : "xxxx", 构建项目生成打包后的文件的指令  
 "test": "xxxx" 测试项目的指令  
 }  
什么是生产 / 开发环境？  
 生产环境：项目运行时需要的环境  
 开发环境：项目构建打包时需要的环境  
什么是生产 / 开发依赖？  
 生产依赖：项目运行时需要的依赖包  
 开发依赖：项目构建打包时需要的依赖包***

# 6.npm

***npm 包管理器,用来管理（下载、删除、发布、更新...）所有包  
常用指令：  
1. 下载  
 npm install / i xxx  
 node\_modules/ 里面包含所有通过npm下载的包  
 package-lock.json 缓存文件、里面包含之前下载包的信息（版本号、下载地址、hash值）,目的为了下一次在下载重复的包时速度更快  
 将被下载的包 往package.json添加为生产依赖  
 等价于 npm i xxx --save / -S***

***npm i xxx --save / -S 下载并添加到生产依赖中  
 npm i react@16.7.0 下载指定版本的包  
 npm i xxx --save-dev / -D 下载并添加到开发依赖中  
 npm i 下载当前目录下package.json中所有依赖包  
 npm i xxx -g 全局安装.不是给模块引入使用的,而是使用其指令  
 C:\Users\XiongJian\AppData\Roaming\npm 这个安装包的位置  
2. 删除  
 npm remove xxx 删除node\_modules中的包,和package.json、package-lock.json中依赖关系  
 npm uninstall xxx  
3. 扩展  
 npm init -y 初始化包描述文件（包含基本配置） package.json  
注意：  
 1. 保证当前包中有package.json文件  
 2. 保证下载时是在当前package.json文件的目录下下载  
 在cmd中退出操作：ctrl + c  
 淘宝的镜像服务器：  
 npm config set registry https://registry.npm.taobao.org  
 yarn config set registry https://registry.npm.taobao.org  
1. 下载yarn  
 npm i yarn -g  
2. 使用：  
 yarn add xxx 下载并添加到生产依赖  
 yarn add xxx --dev 开发依赖  
如果npm和yarn混着用:  
 如果之前用npm下载过,再用yarn下载,再用npm下载,这时候npm会把yarn下载的包给删除掉  
 原因是因为npm默认看了package-lock.json文件,而yarn下载时并没有修改package-lock.json文件.  
 对于npm来讲它认为package-lock.json文件没有包含的包是多余的包,就给删除了***

# 7.buffer

***Buffer用来存储二进制数据  
 里面元素大小为1字节  
 1 byte 字节 = 8 bit 位  
 1 kb = 1024 byte  
 1 mb = 1024 kb  
 1 gb = 1024 mb  
 1 tb = 1024 gb  
 英文 --- 1字节  
 中文 --- 3字节  
使用：  
 创建Buffer  
 Buffer.alloc(10) 创建buffer,安全.里面的数据都是00,都是空.性能稍差  
 Buffer.allocUnsafe(10) 创建buffer,不安全.里面可能包含敏感数据,有值.性能最好  
 Buffer.from(string)  
// 这个方法被废弃（在未来的nodejs版本这个方法不能使用）了  
// const buffer = new Buffer(10);  
// console.log(buffer);  
  
// const buffer = Buffer.alloc(10);***

***// <Buffer 00 00 00 00 00 00 00 00 00 00>  
// const buffer = Buffer.allocUnsafe(10);***

***// <Buffer 30 6c ee 67 9b 02 00 00 00 01>  
// const buffer = Buffer.from('hello atguigu');***

***// <Buffer 68 65 6c 6c 6f 20 61 74 67 75 69 67 75>*const *buffer* = Buffer.from('尚硅谷');**

***// <Buffer e5 b0 9a e7 a1 85 e8 b0 b7>  
console*.log(*buffer*);  
*console*.log(*buffer*.toString()); *//尚硅谷 能将buffer数据转换为字符串***

# 8.fs-file system 文件系统

***// fs是nodejs的核心模块,需要用户手动引入才能使用*const fs = *require*('fs');**

## 8.1异步文件写入

***fs.writeFile(path, data[, options], callback)  
 - path 要写入文件路径  
 - data 要写入文件的内容  
 - options 可选值（可传可不传）：一定有默认值  
 配置对象：属性名固定的对象  
 mode: 设置文件操作权限,默认值是0o666  
 0o111 文件可执行  
 0o222 文件可写入  
 0o444 文件可读取  
 0o666 文件可读可写  
 flag: 要怎么操作当前文件,默认值是 'w'  
 'w' 写入文件  
 'r' 读取文件  
 'a' 追加文件  
 - callback 回调函数（当文件写入成功/失败,都会触发当前回调函数）  
 - error 错误对象  
 如果有错误,值就是对象,对象里面包含出错原因  
 如果没有错误,值就是null***

**const fs = *require*('fs');  
*// 异步写入文件*fs.writeFile('./c.txt', '今天天气真晴朗', {mode: 0o666, flag: 'a'}, function (error) {  
 if (!error) {  
 *// 方法没出错* console.log('文件写入成功~');  
 } else {  
 console.log('文件写入失败', error);  
 }  
})**

## 8.2流式文件写入

***fs.createWriteStream(path)  
将文件内容一段一段分别写入***

**const fs = *require*('fs');  
*// 创建一个可写流*let ws = fs.createWriteStream('./d.txt');  
*// 监听写入状态（什么时候开始写入,什么时候写入完毕）  
// 绑定一次性事件,执行一次后会自动的解绑*ws.once('open', function () {  
 console.log('文件开始写入了~');  
})  
*// 没有触发文件写入完毕,导致文件流一直在等待.  
// 内存泄漏： 占用一片内存,但是没有实际用途  
// 内存溢出： 超过了当前内存的最大空间,直接崩溃*ws.once('close', function () {  
 console.log('文件写入完毕~');  
})  
*// 要往文件中写入内容*ws.write('态度');  
ws.write('细节');  
ws.write('目标');  
ws.write('行动');  
*// 手动关闭可写流*ws.end();  
ws = null;**

## 8.3异步文件读取

***fs.readFile(path[, options], callback)***

**const fs = *require*('fs');  
fs.readFile('./c.txt', function (error, data) {  
 if (!error) {  
 *// data是读取的文件内容,默认是buffer数据* console.log(data.toString());  
 } else {  
 console.log(error);  
 }  
})**

## 8.4流式文件读取

***fs.createReadStream(path)***

**const fs = *require*('fs');  
*// 创建一个可读流*const rs = fs.createReadStream('C:\\Users\\XiongJian\\Desktop\\流失写入文件.avi');  
rs.once('open', function () {  
 console.log('文件开始读取');  
})  
*// 可读流会自动关闭*rs.once('close', function () {  
 console.log('文件读取完毕');  
})  
*// 查看读取的内容*rs.on('data', function (data) {  
 console.log(data.length); *// 65536 byte --- 64 kb*})**

## 8.5 pipe

***实现：将一个文件复制到另外一个位置***

**const fs = *require*('fs');  
*// 创建一个可读流*const rs = fs.createReadStream('C:\\Users\\XiongJian\\Desktop\\流失写入文件.avi');  
*// 创建一个可写流*const ws = fs.createWriteStream('./test.avi');  
*/\*  
rs.once('open', function () {  
 console.log('文件开始读取');  
})  
// 可读流会自动关闭  
rs.once('close', function () {  
 console.log('文件读取完毕');  
 // 关闭可写流  
 ws.end();  
})  
// 查看读取的内容  
rs.on('data', function (data) {  
 ws.write(data);  
})\*/  
// 简写方式*rs.pipe(ws);**

# 9. mongoose

***1. 引入mongoose  
2. 连接本地mongodb数据库  
3. 创建Schema模式对象  
4. 创建Model模型对象  
5. 创建Document文档对象  
6. 保存数据***

***// 1. 引入mongoose*const mongoose = *require*('mongoose');  
*// 2. 连接本地mongodb数据库*mongoose.connect('mongodb://localhost:27017/mongoose\_test', { useNewUrlParser: true, useCreateIndex: true });  
const promise = new Promise((resolve, reject) => {  
 *// 绑定数据库连接事件* mongoose.connection.once('open', (error) => {  
 if (!error) {  
 console.log('数据库连接成功了~');  
 *// 将promise对象状态改成成功的状态* resolve();  
 } else {  
 console.log('数据库连接失败~', error);  
 *// 将promise对象状态改成失败的状态* reject();  
 }  
 })  
});  
(async () => {  
 *// 等待数据库连接成功* await promise;  
 *// 3. 创建Schema模式对象* const Schema = mongoose.Schema;  
 *// 通过Schema对象创建一个针对集合中的文档的约束对象* const stuSchema = new Schema({  
 *// 这个对象就是用来约束集合中文档的对象  
 // 里面放置着约束条件（1. 约束文档可以保存哪些属性 2. 约束文档中属性的值的类型）* name: String,  
 age: Number,  
 hobby: [String], *// 值是数组,数组里面必须放字符串* info: Schema.Types.Mixed, *// 混合,什么数据类型都行* phone: {  
 type: String,  
 unique: true, *// 唯一的* required: true, *// 必须的  
 // default: '11111111111' // 默认值* }  
 })  
 *// 4. 创建Model模型对象* const Students = mongoose.model('students', stuSchema);  
 *// 5. 创建Document文档对象* const s1 = new Students({  
 name: 'jack',  
 age: 18,  
 hobby: ['rose'],  
 info: '这是一个coder',  
 phone: '222222222'  
 })  
 console.log(s1);  
 *// 6. 保存数据* s1.save((error) => {  
 if (!error) {  
 console.log('数据保存成功~');  
 } else {  
 console.log(error);  
 }  
 });  
})();**

# 10. model

***1. 引入mongoose  
2. 连接本地mongodb数据库  
3. 创建Schema模式对象  
4. 创建Model模型对象  
5. 创建Document文档对象  
6. 保存数据***

***// 1. 引入mongoose*const mongoose = *require*('mongoose');  
*// 2. 连接本地mongodb数据库*mongoose.connect('mongodb://localhost:27017/mongoose\_test', { useNewUrlParser: true, useCreateIndex: true });  
const promise = new Promise((resolve, reject) => {  
 *// 绑定数据库连接事件* mongoose.connection.once('open', (error) => {  
 if (!error) {  
 console.log('数据库连接成功了~');  
 *// 将promise对象状态改成成功的状态* resolve();  
 } else {  
 console.log('数据库连接失败~', error);  
 *// 将promise对象状态改成失败的状态* reject();  
 }  
 })  
});  
(async () => {  
 *// 等待数据库连接成功* await promise;  
 *// 3. 创建Schema模式对象* const Schema = mongoose.Schema;  
 *// 通过Schema对象创建一个针对集合中的文档的约束对象* const stuSchema = new Schema({  
 *// 这个对象就是用来约束集合中文档的对象  
 // 里面放置着约束条件（1. 约束文档可以保存哪些属性 2. 约束文档中属性的值的类型）* name: String,  
 age: Number,  
 hobby: [String], *// 值是数组,数组里面必须放字符串* info: Schema.Types.Mixed, *// 混合,什么数据类型都行* phone: {  
 type: String,  
 unique: true, *// 唯一的* required: true, *// 必须的  
 // default: '11111111111' // 默认值* }  
 })  
 *// 4. 创建Model模型对象* const Students = mongoose.model('students', stuSchema);  
 */\*  
 模型对象方法：CRUD 增删改查  
 - create 增  
 Model.create(文档对象, 回调函数) 函数返回值是undefined  
 Model.create(文档对象) 函数返回值是promise对象  
 创建一条文档对象 文档对象就是对象  
 创建多条文档对象 文档对象就是数组  
 - read 查  
 Model.find(查询条件[, 投影], 回调函数) 找没找到,返回值都是一个数组  
 Model.findOne(查询条件[, 投影], 回调函数) 会返回找到的第一个文档对象,没有找到是null  
 - 操作符：  
 1. < <= > >= != $lt $lte $gt $gte $ne  
 2. 或 $or $in  
 3. 正则表达式  
 - 投影：用来过滤一些不想要的数据  
 如果是 xxx: 0 ,这时候就是过滤掉指定属性,而其他属性保留  
 如果是 xxx: 1 ,这时候就是保留指定属性,而过滤掉其他属性  
 一般不能混着用. \_id例外  
 - update 改  
 Model.updateOne(查询条件, 要更新的内容) 更新一个  
 Model.updateMany(查询条件, 要更新的内容) 更新多个  
 - delete 删  
 Model.deleteOne(查询条件)  
 Model.deleteMany(查询条件)  
 \*/* const result = await Students.deleteOne({name: '汪伦'});  
 *// const result = await Students.updateOne({name: '汪伦'}, {age: 21});  
 // const result = await Students.findOne({age: 18, name: '汪伦'});  
 // const result = await Students.findOne({age: {$gte: 20}});  
 // const result = await Students.find({$or: [{age: 16}, {age: 18}]});  
 // sort({age: 1}) 年龄按照从小到大的顺序排序 sort({age: -1}) 年龄按照从大到小的顺序排序  
 // const result = await Students.find({age: {$in: [16, 18]}}).sort({age: -1});  
 // 模糊匹配  
 // const result = await Students.find({name: /^汪/}, {\_\_v: 0, \_id: 0});  
 // const result = await Students.find({name: /^汪/}, {\_\_v: 1, \_id: 1});  
 // const result = await Students.find({name: /^汪/}, {name: 1, \_id: 0});  
  
 /\*const result = Students.create({  
 name: 'rose',  
 age: 16,  
 hobby: ['jack'],  
 phone: '3333333333',  
 info: 'i like jack'  
 }, (error) => {  
 // 在nodejs中回调函数默认的第一个参数,往往都是error 错误对象  
 // 这个机制叫做错误优先机制.nodejs要求开发者优先处理错误  
 if (!error) {  
 console.log('数据保存成功');  
 } else {  
 console.log(error);  
 }  
 })\*/  
 /\*const result = await Students.create([{  
 name: '汪伦',  
 age: 20,  
 hobby: ['睡觉', '迟到'],  
 phone: '66666666666',  
 info: '不及汪伦送我钱'  
 }, {  
 name: '汪伦',  
 age: 20,  
 hobby: ['睡觉', '迟到'],  
 phone: '77777777777',  
 info: '不及汪伦送我钱'  
 }])\*/* console.log(result);  
})();**

# 11. nodejs服务器

***// http模块是nodejs核心模块*const http = *require*('http');  
  
*// 通过http模块来创建服务*const server = http.*createServer*((request, response) => {  
 */\*  
 request 请求信息：客户端发送请求到服务器  
 response 响应信息: 服务器响应给客户端  
 \*/  
 // charset=utf-8 告诉浏览器请你用utf8解码  
 // text/html 告诉浏览器当前返回的响应是一个html文本* response.setHeader('content-type', 'text/html;charset=utf-8');  
  
 *// 返回一个响应给客户端  
 // 修改了服务器代码,一定要重启服务器* response.end('<h1>这是nodejs返回的响应</h1>');  
});  
  
*// 通过server监听端口号  
// 端口号： 27017 系统想要运行某个东西,必须运行在某一个端口号上.如果有其他程序占用了端口号,就不能再使用了.  
// 1 - 65536 三位数以下的端口号,往往都被系统占用了,不建议使用  
// 建议使用 3000 4000 3030 8080 8000....  
// 访问服务器的地址: http://localhost:3000*server.listen(3000, (err) => {  
 if (!err) {  
 *console*.log('服务器启动成功了~');  
 } else {  
 *console*.log('服务器启动失败了~', err);  
 }  
})**

# 12. express中服务器

***// 引入express模块*const express = *require*('express');  
*// 创建App应用对象: 基本上设置服务器的方法都在app上  
// application: express框架中最核心的对象,特点：唯一的*const app = express();  
*// 设置路由  
// 请求方式：get / post  
// 请求路径：http://localhost:4000/test  
/\*  
 http://localhost:4000/test  
 localhost 代表本机的默认域名  
 http://127.0.0.1:4000/test  
 127.0.0.1 代表本机的默认IP地址  
 http://192.168.13.35:4000/test  
 192.168.13.35 代表本机的局域网ip地址  
  
 访问服务器的地址：  
 协议名://域名（ip地址）:端口号/资源名称  
  
 默认端口号 (在网页地址栏中默认不显示)  
 http 80  
 https 443  
 默认资源名 index.html  
 \*/  
// 处理请求的回调函数：*app.get('/test', (request, response) => {  
 */\*  
 request 请求信息：客户端发送请求到服务器  
 response 响应信息: 服务器响应给客户端  
 \*/  
 // 返回响应* response.send('<h1>这是express111框架服务器返回的响应</h1>');  
});  
app.post('/', (request, response) => {  
 */\*  
 request 请求信息：客户端发送请求到服务器  
 response 响应信息: 服务器响应给客户端  
 \*/  
 // 返回响应* response.send('<h1>这是express111框架服务器返回的响应</h1>');  
});  
*// 监听端口号：为了将服务器启动在4000端口下*app.listen(4000, (err) => {  
 if (!err) *console*.log('服务器启动成功了~');  
 else *console*.log('服务器启动失败', err);  
})**

# 13. http协议

***1. 规定互联网中客户端和服务器之间的通信的规则  
 2. 超文本传输协议,基于TCP/IP协议位于应用层的协议  
 3. 协议规定的具体内容：我们称之为报文：请求报文和响应报文***

# 14. get请求报文

***\* 请求首行  
GET http://localhost:4000/login?username=sunwukong&password=123123 HTTP/1.1  
 请求方式 GET  
 请求地址 http://localhost:4000/login  
 请求参数 ?username=sunwukong&password=123123   
 查询字符串：?key1=value1&key2=value2 通常就是GET请求提交的参数  
 协议/协议版本号 0.9 1.0 1.1（最常用） 2.0（功能最强大,但是兼容性差）...  
\* 请求头  
Host: localhost:4000  
 要访问服务器地址  
Connection: keep-alive  
 保持长连接：共用TCP连接、DNS解析等服务  
Upgrade-Insecure-Requests: 1  
 允许使用https协议  
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/73.0.3683.86 Safari/537.36  
 用户代理：浏览器内核版本  
 再过去服务器就是通过当前字段来识别浏览器的,如果是ie,不好意思,就不返回响应  
Accept:text/html,application/xhtml+xml,application/xml;q=0.9,image/***

***webp,image/apng,\*/\*;q=0.8,application/signed-exchange;v=b3  
 浏览器允许接受的文件类型  
 application/xml;q=0.9 这是一个整体.q=0.9,***

***当前文件类型的优先级为0.9,默认优先级是1  
 \*/\*;q=0.8 所有文件类型都行,优先级为0.8  
Referer:http://localhost:63342/class0105/03.nodejs/day03/index.html?\_ijt=m6j4vqth5b4f9kfc8ropre52m8  
 发送请求方的地址：  
 防盗链：防止别人盗取这链接.被访问的资源不允许其他人使用.  
 广告计费:   
Accept-Encoding: gzip, deflate, br  
 允许接收的压缩格式：一旦接收了,就会自动解压  
Accept-Language: zh-CN,zh;q=0.9  
 允许接收的语言  
Cookie: Webstorm-d632f404=6dd4cb30-4200-4430-b5a8-a4ca17524f5d  
 饼干、甜心. 解决http协议的问题.  
\* 空行  
 就是回车,就是空一行  
\* 请求体  
 这里没有,所以空一行***

# 15. post请求报文

***\* 请求首行  
POST http://localhost:4000/login HTTP/1.1  
 POST请求也可以携带查询字符串参数,只不过通常不用  
\* 请求头  
Host: localhost:4000  
Connection: keep-alive  
Content-Length: 34  
 请求体内容长度,单位是byte  
Cache-Control: max-age=0  
 缓存控制: 有效期为0 --> 不强制缓存,走协商缓存  
Origin: http://localhost:63342  
 请求来源的地址  
Upgrade-Insecure-Requests: 1  
Content-Type: application/x-www-form-urlencoded  
 请求内容类型  
 application/x-www-form-urlencoded：***

***当前数据是以表单方式发送的.并且采用了urlencoded编码***

***(key1=value1&key2=value2)  
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/73.0.3683.86 Safari/537.36  
Accept:text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,\*/\*;q=0.8,application/signed-exchange;v=b3  
Referer:http://localhost:63342/class0105/03.nodejs/day03/index.html?\_ijt=59g4fpasrkbcetndql8thik684  
Accept-Encoding: gzip, deflate, br  
Accept-Language: zh-CN,zh;q=0.9  
Cookie: Webstorm-d632f404=6dd4cb30-4200-4430-b5a8-a4ca17524f5d  
\* 空行  
   
\* 请求体  
username=sunwukong&password=123123  
 POST请求的请求参数***

# 16. get和post请求的区别

***get和post请求的区别：  
 1. 请求参数位置不一样：  
 get请求位于请求首行,以查询字符串格式  
 post请求位于请求体内  
 2. 安全性问题  
 get请求请求参数在浏览器地址栏直接可见,所以不太安全  
 post请求请求参数位于请求体内,直接不可见,所以相对更加安全  
 绝对意义上都不是安全的  
 3. 请求参数的大小：  
 get请求的参数位于浏览器地址栏上,所以如果浏览器对地址栏长度有限制（现代基本没限制,再过去有限制）,参数就有限制  
 post请求的参数位于请求体内,理论上是没有限制的  
  
用法：  
 get：  
 不包含用户的敏感数据（账户名、密码）  
 通常get请求对应数据库的操作是查  
 post:  
 包含用户的敏感数据（账户名、密码）  
 通常post请求对应数据库的操作是增  
 一般put和delete用的少,所以put和delete就放在post中做  
 put  
 通常put请求对应数据库的操作是改  
 delete  
 通常delete请求对应数据库的操作是删***

# 17.响应报文

***\* 响应首行  
 HTTP/1.1 200 OK  
 协议名/协议版本号 响应状态码  
 1xx 请求还未完成,还需要接下来处理（请求正在处理中）  
 2xx 请求成功 200  
 3xx 请求资源重定向  
 301 请求资源重定向到新的一个地址, 永久重定向：对于搜索引擎来讲,保留新网址和新内容  
 302 请求资源重定向到新的一个地址,临时重定向: 对于搜索引擎来讲,保留旧网址和新内容  
 304 请求资源未修改,请去浏览器缓存中读取内容  
 4xx 请求资源在服务器没有找到 404  
 5xx 请求的服务器出了问题 500  
\* 响应头  
 X-Powered-By: Express  
 标识服务器使用express框架写的  
 X-xxx X开头的字段不是HTTP协议规定的内容,是用户自己定义的  
 Content-Type: text/html; charset=utf-8  
 这是express框架自动加上的内容  
 Content-Length: 31  
 ETag: W/"1f-zolrBgoiQbKOSXIBRFTQ+wQFe0k"  
 协商缓存使用的  
 Date: Mon, 22 Apr 2019 08:48:32 GMT  
 日期：北京处于东八区,所以计算时需要加上8小时  
 Connection: keep-alive  
\* 空行  
  
\* 响应体  
 这是服务器返回的响应~  
 具体响应给客户端的内容***

# 18.路由(route)

***1. 组成：  
 请求方式 method : GET POST PUT DELETE  
 请求路径： 定义访问服务器的网址  
 一对一：  
 '/' --> http://localhost:3000/  
 '/login' --> http://localhost:3000/login  
 多对一：  
 '/hotel/:xxx' --> http://localhost:3000/hotel/123***

***http://localhost:3000/hotel/456  
 正则表达式： /\/shop\/(1\d{2}|200)/ -->***

***http://localhost:3000/shop/123 ...  
 请求方式和请求路径：当服务器接收到请求后,遍历所有的路由,只有满足了请求方式和请求路径这两个条件,才会由这个路由处理  
 回调函数（句柄函数、钩子函数）：用来处理请求,返回响应的函数  
 request 请求信息：浏览器给服务器  
 req.params 获取params参数：请求路径的:id  
 req.query 获取查询字符串参数  
 req.body 获取请求体参数(默认express服务器不解析请求体数据,所以默认{})  
 req.headers 获取请求头所有信息  
 req.get() 获取请求头中一条信息  
 req.cookies 获取cookie  
 response 响应信息：服务器给浏览器  
 // 返回响应的方法  
 res.download(文件路径) 返回一个文件,浏览器会自动下载（需要下载）  
 res.sendFile(文件路径) 返回一个文件,浏览器会自动打开展示（需要打开文件显示）  
 res.end() 快速返回响应,响应内容不做任何处理（通常返回英文或数字的响应）  
 res.json() 将响应内容转化为json字符串,在返回响应（如果响应数据是object/array）  
 res.send()根据不同的响应内容做不同的处理(如果是标签字符串/中文/Buffer)  
 如果是buffer数据,就会设置一个响应头 Content-Type: application/octet-stream  
 如果是string数据,就会设置一个响应头 Content-Type: text/html;charset=utf8  
 如果是object/array数据,就转化为json字符串,在返回响应  
 res.redirect() 请求资源重定向,重定向到新的网址（重新访问另外一个网址）  
 // 设置响应信息  
 res.get() 获取响应头信息  
 res.set() 设置响应头信息  
 res.status() 设置响应状态码***

***自动检测代码的变化,一旦变化,会自动重启服务器  
 - 下载  
 npm i nodemon -g  
 - 启动项目  
 右键 + run  
 nodejs xxxx  
 需要控制台使用 nodemon + 文件名 运行  
 2. 是什么？  
 一种key-value映射关系：key值的是请求路径path,value指的是回调函数callback  
 3. 作用：  
 接收请求、返回响应  
 定义请求路径（定义访问服务器的网址）  
 4. 执行规则：  
 express会将所有路由按照先后顺序添加一个数组中.  
 当请求访问服务器时,服务器就会遍历这个数组,取出路由看是否匹配(请求方式和请求路径)  
 满足条件就会执行路由的回调函数.然后请求终止了~（后面的就不看了）  
 不满足条件,再看下一个路由看是否满足条件,如果都没满足呢？返回404 cannot get /xxx***

# 19.中间件(middleware)

***1. 是什么？  
 本质上是一个函数  
 2. 组成:  
 request 和路由一致,都是请求信息  
 response 和路由一致,都是响应信息  
 next 函数,作用：用来调用堆栈下一个中间件函数  
 3. 作用:  
 执行任意代码（函数）  
 接收请求(所有请求都能接收),返回响应  
 调用堆栈下一个中间件函数/路由  
 4. 执行规则：  
 express会将所有路由和中间件按照先后顺序添加一个数组中.  
 当请求访问服务器时,服务器就会遍历这个数组,取出对应中间件（所有请求都匹配）/路由看是否匹配(请求方式和请求路径)  
 满足条件就会执行路由/中间件的回调函数.  
 如果中间件调用了next方法,还会看一下路由或者中间件  
 如果没有调用,请求终止了~（后面的就不看了）  
 不满足条件,再看下一个路由/中间件看是否满足条件,如果都没满足呢？返回404 cannot get /xxx  
 5. 应用：  
 内置中间件：  
 指得是express框架内置的一些直接可以使用的中间件函数  
 express.static() 用来向外暴露服务器的静态资源  
 express.urlencoded({extended: true}) 用来解析请求体参数***

***// 引入内置中间件解析请求体参数：获取到请求体数据,将其解析为{},放在req.body上* app.use(express.urlencoded({extended: true}));  
 *// 向外暴露静态资源（长期不会发生变化的资源）,外部就能直接访问静态资源* app.use(express.static('public'));**

***第三方中间件  
 指得是其他社区开发者开发的开源中间件  
 cookieParser() 用来解析cookie数据  
 应用级中间件  
 开发者自己开发的中间件：  
 提取重复代码  
 路由器中间件  
 错误处理中间件  
 用来处理中间件/路由的错误: 必须传四个参数  
 app.use((err, req, res, next) => {})***

# 20.router路由器

***/\*  
 路由器：  
 1. 是什么？  
 一个集中性管理路由和中间件的对象  
 看做是一个小型的app应用对象（有app对象的部分功能：能够设置路由和中间件）  
 2. 作用：  
 满足多人协同开发  
 按照功能对路由和中间件进行模块化  
 3. 使用  
 \*/***

***// 引入express*const express = *require*('express');  
*// 获取Router路由器对象*const Router = express.Router;  
*// 创建路由器实例对象  
// router: 可以看做是一个小型的app应用对象（有app对象的部分功能：能够设置路由和中间件）*const router = new Router();  
  
router.use((req, res, next) => {  
 console.log('中间件执行了~');  
 next();  
});  
router.get('/', (req, res) => {  
 console.log(req.query);  
 res.send('服务器返回的响应');  
});  
*// router要想生效：得想法办放在app上  
// 将router暴露出去*module.exports = router;**

***\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\****

**const express = *require*('express');  
*// 引入路由器实例对象*const router = *require*('./router');  
const app = express();  
*// 放在app上. router就是一个中间件函数  
// 路由器中间件*app.use(router);  
  
app.listen(3000, (err) => {  
 if (!err) console.log('服务器启动成功了~');  
 else console.log(err);  
})**

# 21.ejs

***场景：我需要在页面中渲染错误提示数据  
解决: 服务器渲染技术  
  
ejs：  
 1. 是什么？  
 高效的 JavaScript 模板引擎. 用来编译运行js模板代码的,性能高  
 之前文件 html 模板代码文件 ejs  
 2. 作用：  
 用来服务器渲染：将位于服务器端的数据渲染到指定页面上  
 3. 使用：  
 - 下载ejs  
 npm i ejs  
 - 配置项：  
 告诉express,你的模板资源放在哪里.（才能找到）  
 app.set('views', '模板资源目录');  
 告诉express,你使用哪个模板引擎解析模板资源  
 app.set('view engine', '使用哪个模板引擎解析模板资源')***

***// views就是模板资源目录(新建文件夹views)*app.set('views', './views');  
*// 使用ejs解析模板资源*app.set('view engine', 'ejs');  
  
app.get('/', (req, res) => {  
 *// 错误提示* const err = '<h1>用户名密码错误</h1>';  
 const data = [{name: 'jack', age: 18}, {name: 'jack', age: 18},{name: 'jack', age: 18},{name: 'jack', age: 18},{name: 'jack', age: 18},{name: 'jack', age: 18},]  
 *// 使用配置好的模板引擎解析传入的模板资源  
 // 将指定数据渲染到ejs页面上, 并返回渲染完成数据的页面给前端 res.render('模板页面', 指定数据)  
 // 指定数据：必须是对象结构,ejs页面直接使用* res.render('index.ejs', {err, data});  
})  
app.listen(3000, (err) => {  
 if (!err) console.log('服务器启动成功了~');  
 else console.log(err);  
})**

***ejs模板资源文件：  
 有一些特定语法,专门用来实现服务器渲染的  
 这些语法能让我在写html的地方写js代码,所以所有写js代码都应该用ejs语法包起来  
 修改模板资源不需要重启服务器  
  
 <% %> 能够执行任意的js代码,没有输出到模板页面上  
 遍历生成类似的结构  
 <%= %> 输出转义后的数据到模板页面上：用的最多,基本上都用.  
 安全性的问题：我们不能相信用户的任何输入（凡是用户需要输入的内容,后台都必须要转义处理）  
 <%- %> 输出非转义的数据到模板页面上*<%= err %>  
<%- err %>  
<ul>  
 <% data.forEach((item) => { %>  
 <li>  
 <span>姓名：<%= item.name %></span>  
 <span>年龄：<%= item.age %></span>  
 </li>  
 <% }) %>  
</ul>**

# 22.cookie

***1. 是什么？  
 本质上就是一个存储在浏览器端的文本：key-value关系  
 2. 作用：  
 用来存储特定数据  
 用来在网络通信中解决http协议无状态的问题（可以用来区分不同的请求）  
 3. 工作原理：  
 第一次：浏览器发送请求到服务器上,服务器判断请求通过,返回一个cookie（包含用户信息）给浏览器,浏览器接收到cookie并保存下来  
 下次：浏览器发送请求时会将所有cookie都自动携带上,服务器接收到请求并分析其中的cookie,就能知道哪个用户发送的请求,从而针对当前用户进行响应  
 4. 使用：***

***- 下载***

***npm i cookie-parser  
- 引入***

***const cookieParser = require('cookie-parser');  
- 设置cookie,返回给浏览器  
 res.cookie(key, value[, options]);  
- 读取cookie,分析其中内容  
 req.cookies  
 我们需要通过第三方中间件解析cookie --- cookie-parser  
- 删除cookie  
 res.clearCookie(key);***

***5. 时效性  
 会话cookie (当服务器返回cookie给浏览器时,开始会话.关闭浏览器结束会话)  
 res.cookie('user', 'sunwukong');  
 持久化cookie  
 res.cookie('user', 'sunwukong', {maxAge: 1000 \* 3600 \* 24 \* 7}); // 7天后才消失  
 6. 缺点：  
 长度和数量都有限制：数量为20个左右,大小为4kb左右  
 传输流量大：（发送请求占的带宽更多,产生流量更多）  
 安全性更低：存储在浏览器端***

# 23.session

***1. 是什么？  
 本质上一个存储在服务器端的对象  
 2. 作用：  
 解决cookie的痛点  
 3. 工作原理  
 1. 当需要创建session对象时,中间件会帮助我们自动创建session对象,并保存在数据库中,还返回一个携带session\_id的cookie给浏览器  
 req.session.xxx = xxx; // 创建操作***

***// 将用户数据保存在session中  
req.session.user = result.\_id;***

***2. 当请求经过中间件时,会自动解析cookie,去数据库中找到cookie里面session\_id对应数据,挂载到req.session上  
 req.session.xxx // 读取之前保存的值***

***// 读取session中的值  
const { user } = req.session;***

**const session = *require*('express-session');  
const MongoStore = *require*('connect-mongo')(session);**

**app.use(session({  
 secret: 'hello Atguigu 0105', *// 秘密：参与加密的一个串* saveUninitialized: false, *// 如果不需要保存数据,就不创建sessin对象.* resave: false, *// 如果数据没有修改,不会重新保存数据* store: new MongoStore({ *// 保存在哪里* url: 'mongodb://localhost:27017/exec', *// mongoDB服务器地址* touchAfter: 24 \* 3600 *// 会话只在24小时内更新一次,无论发生多少请求（除了那些改变会话数据的东西)* }),  
 cookie: { *// 设置cookie的有效期* maxAge: 1000 \* 3600 \* 24 \* 7  
 }  
}));**

# 24.ajax

## 24.1原生ajax

***// 1. 创建xhr实例对象： 用来发送ajax请求的对象*const xhr = new XMLHttpRequest();  
  
*// 2. 绑定事件监听： 用来监听响应结果（服务器返回的响应）  
// readystate xhr内部的状态码. 内部状态码发生变化时触发的事件  
/\*  
 readystate  
 0：代表open方法还未调用  
 1：代表send方法还未调用（还没有发送请求）,还来得及设置请求信息  
 2：代表send方法已经发送了（请求已经发送了）,并且接受到了部分的响应信息（响应首行和响应头 --> 响应状态码）  
 3：代表接受到了部分响应数据（如果数据较小,就全部接受了,如果数据比较大,就只接受部分数据） --> 响应体数据  
 4：代表全部接受完毕响应数据  
 \*/*xhr.onreadystatechange = function () {  
 *// 判断请求成功  
 // 获取响应状态码 200 xhr.status* console.log(xhr.status);  
 if (xhr.readyState === 4 && xhr.status === 200) {  
 *// 请求成功~ 打印输出响应的内容 xhr.responseText* console.log(xhr.responseText);  
 }  
};  
  
*// 3. 设置请求信息（请求方式、请求地址、请求头、请求参数等...）***

***// xhr.open('GET','http://localhost:3000/?username=sun&age=18')***

***// xhr.open(请求方式,请求地址?查询字符串)  
// xhr.setRequestHeader('xxx', 'xxxxxx'); // 设置请求头*xhr.open('POST', 'http://localhost:3000/');  
xhr.setRequestHeader('content-type','application/x-www-form-urlencoded');  
  
*/\*  
问题：  
 默认浏览器都能够缓存get请求（请求地址和请求参数不变）  
 chrome/firefox  
 如果后台服务器响应结果不变,那么会读取之前的缓存  
 如果后台服务器响应结果变了,那么会读取变化后结果  
 ie  
 不管后台服务器响应结果发生了什么变化,永远读取的是第一次缓存结果  
 带来的问题：如果数据是实时更新的,ie就读取不到最新内容  
 解决: 让ie不在读取缓存数据,要求请求参数实时变化（加上随机数或者时间戳）  
 \*/  
  
// 4. 发送请求  
// xhr.send();*xhr.send('username=sunwukong&age=19'); *// xhr.send(body) body就是post请求的请求体参数 参数按照urlencoded编码（key=value&key=value）***

## 24.2jQueryAjax

***// 发送请求*$.ajax({  
 url: 'http://localhost:3000', *// 请求地址* method: 'GET', *// 请求方式* data: { *// 请求参数* username: 'zhubajie',  
 age: 20  
 },  
 headers: { *// 请求头* xxx: 'xxx'  
 },  
 success(data) { *// 响应成功的回调函数* console.log(data);  
 },  
 error(error) { *// 响应失败的回调函数* console.log(error);  
 }  
})  
  
$.ajax({  
 url: 'http://localhost:3000', *// 请求地址* method: 'POST', *// 请求方式* data: { *// 请求参数* username: 'zhubajie',  
 age: 22  
 },  
 headers: { *// 请求头* 'content-type': 'application/x-www-form-urlencoded'  
 },  
 success(data) { *// 响应成功的回调函数* console.log(data);  
 },  
 error(error) { *// 响应失败的回调函数* console.log(error);  
 }  
})  
  
*// 简写方式  
// 当被访问的服务器和打开页面的服务器时同一个服务器时,可以用 / ： 当前服务器地址*$.get('/', {username: 'shaheshan', age: 25}, function (data) {  
 console.log(data);  
})  
  
$.post('/', {username: 'shaheshan', age: 30}, function (data) {  
 console.log(data);  
})**

# 25.template

***1. 引入 template-web.js 文件  
2. 创建模板代码  
 - 创建script标签  
 - 将type修改为text/html  
 - 添加一个id属性 -- 为了找到当前标签  
3. 编译模板代码：将模板代码编译成标签字符串  
 template(id值, {渲染的数据})  
4. 将编译后标签字符串插入页面指定容器中,方能生效  
  
应用：需要写很多重复性结构时,使用它***

<**script type="text/html" id="tpl"**>  
 *<!--* 写模板代码 *-->*  
 <**ul**>  
 {{each data}}  
 <**li**> {{$value}} - {{$index}} </**li**>  
 {{/each}}  
 </**ul**>  
 {{if flag}}  
 <**p**>判断条件成立！</**p**>  
 {{/if}}  
</**script**>

**const *html* = template('tpl', {data: ['jack', 'bob', 'lucy', 'tom', 'jerry'], flag: false});  
console.log(*html*);  
document.getElementById('box').innerHTML = *html*;**

# 26.跨域

***同源策略：这是浏览器的安全策略,保证浏览器能更加安全稳定的运行  
规定：域名/ip地址 协议名 端口号 必须三者完全一致  
同源策略：是浏览器的ajax引擎负责维护的  
  
一旦有一个不一样,就违背同源策略,产生跨域.  
  
往往服务器会有多个,资源在不同的服务器上,所以一旦请求就会导致跨域. 开发者就需要想办法解决跨域问题.  
  
程序员发现 像 script、link、img等标签发送请求时,没有跨域问题  
就想利用这些标签来发送请求  
  
程序员就利用了script发送了跨域请求 --- jsonp - json with padding***

## 26.1利用script发送跨域请求

***// 1. 创建script标签*const *script* = document.createElement('script');  
*// 2. 设置script标签的scr属性,为了让其发送请求  
script*.src = 'http://localhost:3000/jsonp?callback=getData';  
*// 3. 定义一个回调函数*function *getData*(data) {  
 console.log(data);  
 document.getElementById("box").innerHTML = `<h1>${data}</h1>`;  
}  
*// 4. 添加到页面中生效*document.body.appendChild(*script*);**

***// jQuery格式*$.getJSON('http://localhost:3000/jsonp?callback=?&xxx=xxx', function (data) {  
 console.log(data);  
})**

***// 服务器***

**app.get('/jsonp', (req, res) => {  
 const { callback } = req.query; *// callback: getData  
 // const data = '这是服务器返回的响应111';* const data = [{name: 'jack'}, {name: 'bob'}];**

**res.send(`${callback}(${*JSON*.stringify(data)})`);**

**});**

## 26.2使用cors解决跨域问题

***//发送请求***

**$.get('http://localhost:3000/cors', function (data) {  
 console.log(data);  
})**

***// 服务器***

**app.get('/cors', (req, res) => {  
 *// 使用cors解决 : 兼容性不好* const saftArr = ['http://localhost:4000','http://localhost:63342'];  
 *// res.set('Access-Control-Allow-origin', '\*'); // 允许所有地址跨域,访问当前资源* res.set('Access-Control-Allow-origin','http://localhost:4000');**

***// 允许单个地址跨域,访问当前资源* res.send('这是cors服务器返回的响应');  
})**

26.3 cors和jsonp区别 ***1. 兼容性：  
 jsonp 极好  
 cors 稍差  
 2. 处理请求的能力  
 jsonp 只能处理 get 请求（因为script只能发送get）  
 cors 能处理所有请求***

# 27.缓存

## 27.1缓存策略

***1. 强制缓存：  
 响应头中：  
 cache-control http1.1 优先级更高  
 expires http1.0  
  
 什么资源会强制缓存：长久不变的资源：logo、jquery  
  
 2. 协商缓存  
 etag / last-modified 位于响应头  
 if-none-match / if-modified-since 位于请求头  
  
 什么资源协商缓存: 会经常发生变化的资源.***

## 27.2缓存流程

***1. 首先看 强制缓存  
- 如果http协议是1.1 就看cache-control  
 - max-age的值 + date 看这个值有没有过期  
 -如果过期了.存在协商缓存就看协商缓存,不存在就直接发送请求读取最新的数据  
 -如果没有过期,不会发送请求,直接读取缓存的数据  
- 如果http协议是1.0 就看expires  
 - 他的值直接就是过期时间  
 -如果过期了.存在协商缓存就看协商缓存,不存在就直接发送请求读取最新的数据  
 -如果没有过期,不会发送请求,直接读取缓存的数据  
2. 协商缓存  
- 第一次：由服务器返回资源文件给浏览器,  
 此时会携带最新的etag（标识文件内容有没有变化）和last-modified（标识修改最近文件修改时间）作为响应头返回给浏览器  
- 下次：浏览器发送请求时,会作为请求头自动携带上if-none-match（上次etag的值） / if-modified-since（上次last-modified的值）  
 服务器接受了.  
 - 首先判断if-none-match和服务器最新的etag  
 如果有检查是否变化：  
 如果变化了.返回新文件给浏览器,状态码为200  
 如果没有变化,返回状态码304,浏览器就会自动读取缓存  
 如果没有if-none-match和服务器最新的etag,才会看if-modified-since  
 - 然后判断if-modified-since和last-modified的值  
 如果变化了.返回新文件给浏览器,状态码为200  
 如果没有变化,返回状态码304,浏览器就会自动读取缓存***

## 27.3原生js实现缓存

***// 设置路由*app.get('/test.js', (req, res) => {  
 console.log('test.js路由触发了~~');  
 readFile('./public/test.js', function (err, data) {  
 if (!err) {  
 *// 设置强制缓存* res.set('cache-control', 'max-age=86400'); *//强制缓存1天* res.set('expires', new Date(Date.now() + 1000 \* 3600 \* 24).toGMTString()); *//强制缓存1天* res.end(data);  
 } else {  
 console.log(err);  
 }  
 });  
})  
  
*// 在外面初始化 etag 和 lastModified*let etag = '222';  
let lastModified = Date.now();  
  
app.get('/test2.js', (req, res) => {  
 console.log('test2.js路由触发了~~');  
  
 *// 第二次判断请求头中if-none-match / if-modified-since 和 位于服务器的 etag last-modified 的值是否一致  
 // 获取请求头 if-none-match* const ifNoneMatch = req.get('if-none-match');  
  
 if (ifNoneMatch) {  
 *// 判断是否相等* if (ifNoneMatch === etag) {  
 *// 如果相等，走协商缓存，返回状态码304* res.status(304).end();  
 } else {  
 *readFileAsync*(res);  
 }  
 } else {  
 *// 说明ifNoneMatch不存在，看last-modified的值* const ifModifiedSince = Date.parse(req.get('if-modified-since'));  
  
 if (ifModifiedSince === lastModified) {  
 *// 如果相等，走协商缓存，返回状态码304* res.status(304).end();  
 } else {  
 *// 不相等，说明文件发生了变化，返回新的文件（新的etag和last-modified）  
 readFileAsync*(res);  
 }  
 }  
})  
  
function *readFileAsync*(res) {  
 *// 不相等，说明文件发生了变化，返回新的文件（新的etag和last-modified）* readFile('./public/test2.js', function (err, data) {  
 if (!err) {  
 *// 设置协商缓存 etag last-modified  
 // 第一次返回新资源和 etag last-modified* res.set('etag', etag);  
 res.set('last-modified',new Date(lastModified).toGMTString());  
 res.end(data);  
 } else {  
 console.log(err);  
 }  
 });  
}**

# 28. Koa

***Koa 是一个新的 web 框架，由 Express 幕后的原班人马打造， 致力于成为 web 应用和 API 开发领域中的一个更小、更富有表现力、更健壮的基石。 通过利用 async 函数，Koa 帮你丢弃回调函数，并有力地增强错误处理。 Koa 并没有捆绑任何中间件， 而是提供了一套优雅的方法，帮助您快速而愉快地编写服务端应用程序。***

**const Koa = *require*('koa');  
const bodyParser = *require*('koa-bodyparser');  
*// 创建App应用对象*const app = new Koa();  
*// 引入第三方中间件解析请求体数据*app.use(bodyParser());  
*// 设置中间件*app.use((context, next) => {  
 *// 接受请求参数  
 // console.log(context.query.username); // 查询字符串参数* context.body = context.request.body;  
 console.log(context.body);  
 *// 返回响应* context.body = '这是服务器返回的响应~';  
});  
*// 监听端口号*app.listen(3000, (err) => {  
 if (!err) console.log('服务器启动成功了·');  
 else console.log(err);  
});**

# 29.**微信js-sdk使用步骤**

***1.绑定js安全域名  
 来到测试号管理页面绑定安全域名  
 问题：本地的localhost域名,外界访问不了,需要将其变成一个互联网都能访问网址  
 解决： ngrok 内网穿透技术： 将本地localhost的服务器，映射外网能访问的网址。  
 指令：ngrok http 3000 生成一个网址，需要将网址填写到安全域名上  
2.JS-SDK使用权限签名算法  
 签名算法 需要使用jsapi\_ticket.jsapi\_ticket需要通过access\_token才能获取  
3.获取 access\_token： 开始开发 - 获取access——token  
4.获取 jsapi\_ticket：微信网页开发 - js-sdk说明文档 - 附录1***

# \*\*\*\*\*\*\*\*项目构建\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# 1.gulp

***gulpfile.js gulp的配置文件：当我运行gulp指令时，会自动找的配置文件  
 所有构建工具都是基于nodejs平台运行的（所以模块化使用commonjs）。  
  
 gulp配置任务过程：  
 1. 上插件网搜插件 gulp-xxx https://gulpjs.com/plugins/  
 2. 下载插件 npm i xxx -D  
 3. 引入插件  
 4. 配置插件任务 gulp.task(插件任务名, callback)  
 5. 运行任务 gulp 插件任务名 (需要在当前配置文件的目录运行)  
  
 npm  
 - npm init 手动一个个输入配置 / npm init -y 初始化一个默认配置  
 - npm i xxx -g 全局安装（当需要在cmd中运行指令时，这个指令需要全局安装）  
 - npm i xxx 局部/本地安装（当项目需要某个依赖包时，这个依赖包需要本地安装）  
  
gulp  
 gulp.task() 注册任务***

***gulp.task('任务名',gulp.series/parallel([]))执行gulp.series/***

***parallel里面的任务（作用：执行其他任务，自己没有任务要执行）  
gulp.task('任务名', function () {}) 只执行回调函数中的任务（作用:只执行自己任务，没有其他任务）***

***gulp.src(string/array) 输入（将指定文件导入到gulp的流中）  
 string 单个路径  
 array 多个路径  
 gulp.dest() 输出（将流中的文件写入到指定位置）  
 gulp.watch(源码目录, 一旦变化要执行的任务) 监视  
babel  
 1. 是什么? 一个js的编译器  
 2. 作用：  
 将typescript / jsx语法 编译成js语法  
 将 ES6及其以上的高级（新）语法 编译成 es5或以下的低级（旧）语法，这样绝大部分浏览器就能兼容了  
  
eslint:  
 1. 是什么？ 一个js语法检查工具  
 2. 作用：检查常见js语法错误  
 3. 使用：  
 需要在package.json中定义配置  
 "eslintConfig": {  
 "parserOptions": {  
 "ecmaVersion": 7, // 使用es7和一下的所有语法  
 "sourceType": "module" // 使用es6的模块化语法  
 },  
 "extends": "eslint:recommended", // 使用eslint推荐的默认配置  
 "env": {  
 "browser": true // 支持使用浏览器端的全局变量  
 }  
 }***

# 2.webpack

***webpack ./src/js/app.js -o ./build/js/built.js --mode=development  
 - 将es6模块化语法编译成浏览器能识别的语法  
 - 不能编译es6其他语法  
webpack ./src/js/app.js -o ./build/js/built.js --mode=production  
 - 将es6模块化语法编译成浏览器能识别的语法  
 - 不能编译es6其他语法  
 - 压缩js代码  
  
webpack只能识别js和json模块，其他资源模块不能识别。 需要使用loader帮助webpack解析这些资源***

***webpack.config.js 就是webpack的配置文件，当你执行webpack指令时，会默认读取配置文件  
  
 1. 入口 entry: string 单个路径/array/object 多个路径  
 指示webpack以入口文件为起点开始构建打包  
 2. 输出 output  
 构建打包后的新文件输出到哪里去  
 3. 加载器 loader  
 帮助webpack解析它解析不了的文件：less、img、html...  
 - 官网找loader，找不到上npm找  
 - 下载loader  
 - 直接配置使用  
 4. 插件 plugins  
 做一些功能更强大的事情。做一些loader做不了的事。  
 5. 模式 mode  
 development 开发  
 production 生产：多一个压缩代码***

**const { resolve } = require('path');  
const HtmlWebpackPlugin = require('html-webpack-plugin');  
const webpack = require('webpack');  
  
// 向外暴露的配置对象  
module.exports = {  
 // 入口文件：指示webpack以入口文件为起点开始构建打包  
 entry: './src/js/app.js',  
 // 输出: 构建打包后的新文件输出到哪里去  
 output: {  
 // 输出路径  
 path: resolve(\_\_dirname, 'build'),  
 // 输出文件名  
 filename: 'js/built.js'  
 },  
 module: {  
 rules: [  
 // 在这里面写具体loader的配置  
 {  
 // npm i eslint eslint-loader -D  
 enforce: 'pre', // 提前使用当前配置去检查 -- 提前使用  
 test: /\.js$/, // 检查是否js文件，一旦通过就会以下面的loader解析，没有通过就不处理  
 exclude: /node\_modules/, // 排除node\_modules的所有文件  
 include: [  
 resolve(\_\_dirname, 'src/js')  
 ], // 只处理当前文件夹下的文件(正则只能写一层目录,数组里面要写绝对路径)  
 loader: "eslint-loader", // 使用哪个loader处理这些文件  
 /\*options: {  
 // eslint options (if necessary)  
 }\*/  
 },  
 {  
 // npm i babel-loader @babel/core @babel/preset-env -D  
 test: /\.js$/,  
 exclude: /(node\_modules)/,  
 use: {  
 loader: 'babel-loader',  
 options: {  
 presets: ['@babel/preset-env']  
 }  
 }  
 },  
 {  
 // npm i less-loader less style-loader css-loader -D  
 test: /\.less$/,  
 use: [{ // 使用多个loader解析：从右往左、从下到上、从后往前执行loader  
 loader: "style-loader" // 从js中找到css代码，并创建一个style标签，放置css样式  
 }, {  
 loader: "css-loader" // 将css代码以commonjs模块化的方式整合在js代码中  
 }, {  
 loader: "less-loader" // 将less编译成css  
 }]  
 },  
 {  
 // npm i url-loader file-loader -D  
 test: /\.(png|jpg|gif|webp)$/,  
 use: {  
 loader: 'url-loader',  
 options: {  
 limit: 8192, // 8\*1024 = 8kb 8kb以下的图片会做base64处理  
 publicPath: 'images', // 修改样式文件中图片url路径  
 outputPath: 'images', // 决定图片输出到本地哪里去  
 name: '[hash:10].[ext]' // 重命名图片文件 hash:10 hash值取前面10位 ext以图片文件的扩展名取补全（之前是什么扩展名就是什么）  
 }  
 }  
 },  
 {  
 // npm i html-loader -D  
 test: /\.(html)$/,  
 use: {  
 loader: 'html-loader' // 处理html中图片  
 }  
 },  
 {  
 test: /\.(eot|svg|ttf|woff)$/,  
 use: {  
 loader: 'file-loader', // 处理其他资源：字体  
 options: {  
 publicPath: 'media', // 修改样式文件中图片url路径  
 outputPath: 'media', // 决定图片输出到本地哪里去  
 name: '[hash:10].[ext]' // 重命名图片文件 hash:10 hash值取前面10位 ext以图片文件的扩展名取补全（之前是什么扩展名就是什么）  
 }  
 }  
 }  
 ]  
 },  
 plugins: [  
 new HtmlWebpackPlugin({  
 template: './src/index.html' // 以当前文件为模板，创建新文件（自动引入打包后生成js、css文件）  
 }),  
 new webpack.NamedModulesPlugin(), // 针对模块热替换功能对每个模块进行命名的插件  
 new webpack.HotModuleReplacementPlugin(), // 模块热替换功能插件  
 ],  
 // 开发环境  
 mode: 'development',  
 // 开发的dev-server 开发服务器 npm i webpack-dev-server -D  
 // 运行devServer，指令为 webpack-dev-server时  
 // dev-server在配置中， 模块热替换在指南中  
 // 开发服务器：在内存中编译打包运行的，不会输出任何文件  
 devServer: {  
 contentBase: resolve(\_\_dirname, "build"), // 要运行代码的目录  
 compress: true, // 开启gzip压缩：压缩后文件体积更小，文件传输速度更快  
 port: 3000, // 端口号  
 open: true, // 自动打开浏览器  
 hot: true // 开启模块热替换（只更新局部模块，而不是刷新整体）  
 }  
};**