jQuery

1. 页面DOM加载完毕

1.1. 等着页面加载完毕再去执行js代码

1.1.1. $(document).ready(function(){ })

1.2. 等着页面加载完毕再去执行js代码(重点)

1.2.1. $(function(){ })

2. $ 是jQuery顶级对象

2.1. $是jQuery的别称(另外的名字)

2.2. $同时也是jQuery的顶级对象 相当于window

3. DOM对象和jQuery对象

3.1. 区别

3.1.1. DOM对象:用原生js获取过来的对象

3.1.2. jQuery对象:用jQuery方式获取过来的对象,本质是通过$把DOM元素进行了包装

3.1.3. jQuery对象只能使用jQuery方法,DOM对象只能使用原生的JavaScript属性和方法

3.2. 相互转换

3.2.1. DOM转为jQuery — $('元素');

3.2.2. jQuery转为DOM

1.$(元素)[序号].方法 例如:$(myVideo)[0].play()

2.例子:$(myVideo).get(0).play()

4. jQuery常用的API

4.1. jQuery基础和层级选择器

4.1.1. 基础选择器

名称	用法	描述
ID选择器	$("#id")	获取指定ID的元素
全选选择器	$("*")	匹配所有元素
类选择器	$(".class")	获取同一类class的元素
标签选择器	$("div")	获取同一类标签的所有元素
并集选择器	$("div,p,li")	选取多个元素
交集选择器	$("li.current")	交集元素

4.1.2. 层级选择器

名称	用法	描述
子代选择器	$("ul>li");	使用>号，获取亲儿子层级的元素；注意，并不会获取孙子层级的元素
后代选择器	$("ul li");	使用空格，代表后代选择器，获取ul下的所有li元素，包括孙子等

4.2. jQuery隐式迭代

4.2.1. 隐式迭代:把匹配的所有元素内部进行遍历循环,给每一个元素添加css这个方法

4.2.2. 例子:$('div').css('background','blue');//改背景色
$('ul li').css('color','pink');//改文字颜色

4.3. 筛选

4.3.1. 筛选选择器

语法	用法	描述
:first	$('li:first')	获取第一个li元素
:last	$('li:last')	获取最后一个li元素
:eq(index)	$("li:eq(2)")	获取到的li元素中，选择索引号为2的元素，索引号index从0开始。
:odd	$("li:odd")	获取到的li元素中，选择索引号为奇数的元素
:even	$("li:even")	获取到的li元素中，选择索引号为偶数的元素

4.3.2. 筛选方法

父 子

```
// 1. 父  parent()  返回的是 最近一级的父级元素 亲爸爸
console.log($(".son").parent());
// 2. 子
// (1) 亲儿子 children()  类似子代选择器  ul>li
// $(".nav").children("p").css("color", "red");
// (2) 可以选里面所有的孩子 包括儿子和孙子  find() 类似于后代选择器
$(".nav").find("p").css("color", "red");
```

兄弟

```
// 1. 兄弟元素siblings 除了自身元素之外的所有亲兄弟
$("ol .item").siblings("li").css("color", "red");
// 2. 第n个元素
// (1) 我们可以利用选择器的方式选择
// $("ul li:eq(2)").css("color", "blue");
// (2) 我们可以利用选择方法的方式选择 更推荐这种写法
$("ul li").eq(2).css("color", "blue");
```

4.4. jQuery排他思想

```
// 1. 隐式迭代 给所有的按钮都绑定了点击事件
$("button").click(function() {
    // 2. 当前的元素变化背景颜色
    $(this).css("background", "pink");
    // 3. 其余的兄弟去掉背景颜色 隐式迭代
    $(this).siblings("button").css("background", "");
});
```

4.4.1. 干掉所有人 保留它自己

4.5. jQuery链式编程

```
// $(this).css("color", "red").siblings().css("color", "");
// 我的颜色为红色, 我的兄弟的颜色为空
// $(this).siblings().css('color', 'red');
// 我的兄弟变为红色  ,我本身不变颜色
```

4.5.1. 链式编程一定要注意变换的对象是谁

- jQuery
  - 1. jQuery常用的API
    - 1.1. jQuery样式操作
      - 1.1.1. css方法
        - //参数只写属性名 则返回属性值
console.log($('.box').css('width'));
        - // 如果设置属性 属性名一定要加引号 值是数字 可以不加引号 不写单位
$('div').css('width',300);
        - // 设置多种样式时 以对象形式修改 值可以不加引号 复合属性要采取驼峰命名法
$('div').css({
width:400,
height:400,
backgroundColor:'red'
})
      - 1.1.2. 设置类样式方法
        - // 添加类
$('div').addClass('current')
        - // 移除类
$('div').removeClass('current')
        - //转换(添加移除)类
$('div').toggleClass('current')
      - 1.1.3. jQuery类操作与className的区别
        - 原生js中的className会覆盖元素原先里面的类名
        - jQuery里面的类操作只是对指定类进行操作,不影响原先的类名
    - 1.2. jQuery效果
      - 1.2.1. 显示 隐藏与转换
注意:一般情况下,我们都不加参数
        - 显示 show();
        - 隐藏 hide();
        - 切换 toggle();
        - 括号里有三个参数:
speed:速度("slow","normal", or "fast")或表示动画时长的毫秒数值(如：1000)

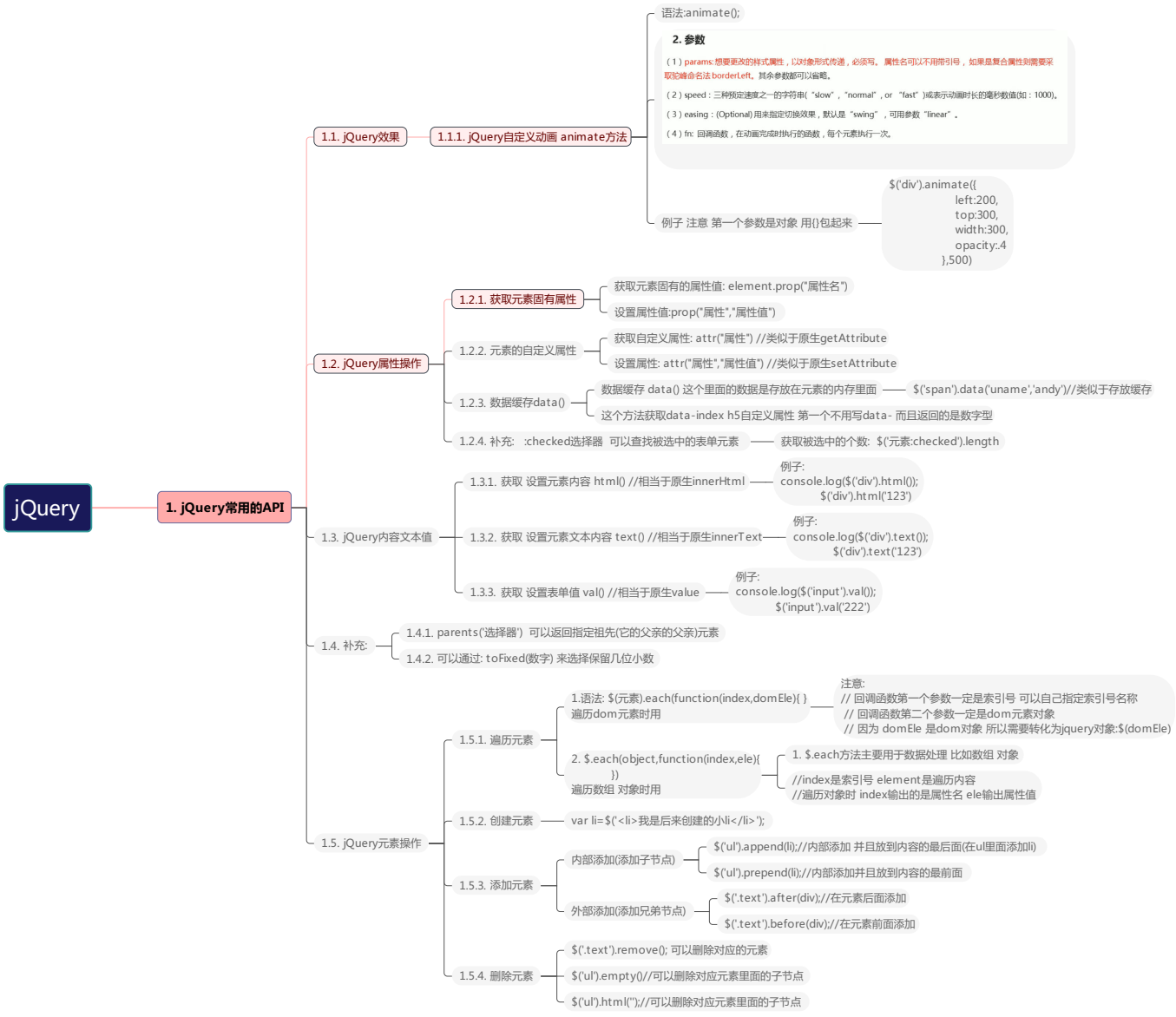
easing:用来指定切换效果，默认是"swing"，可用参数"linear"

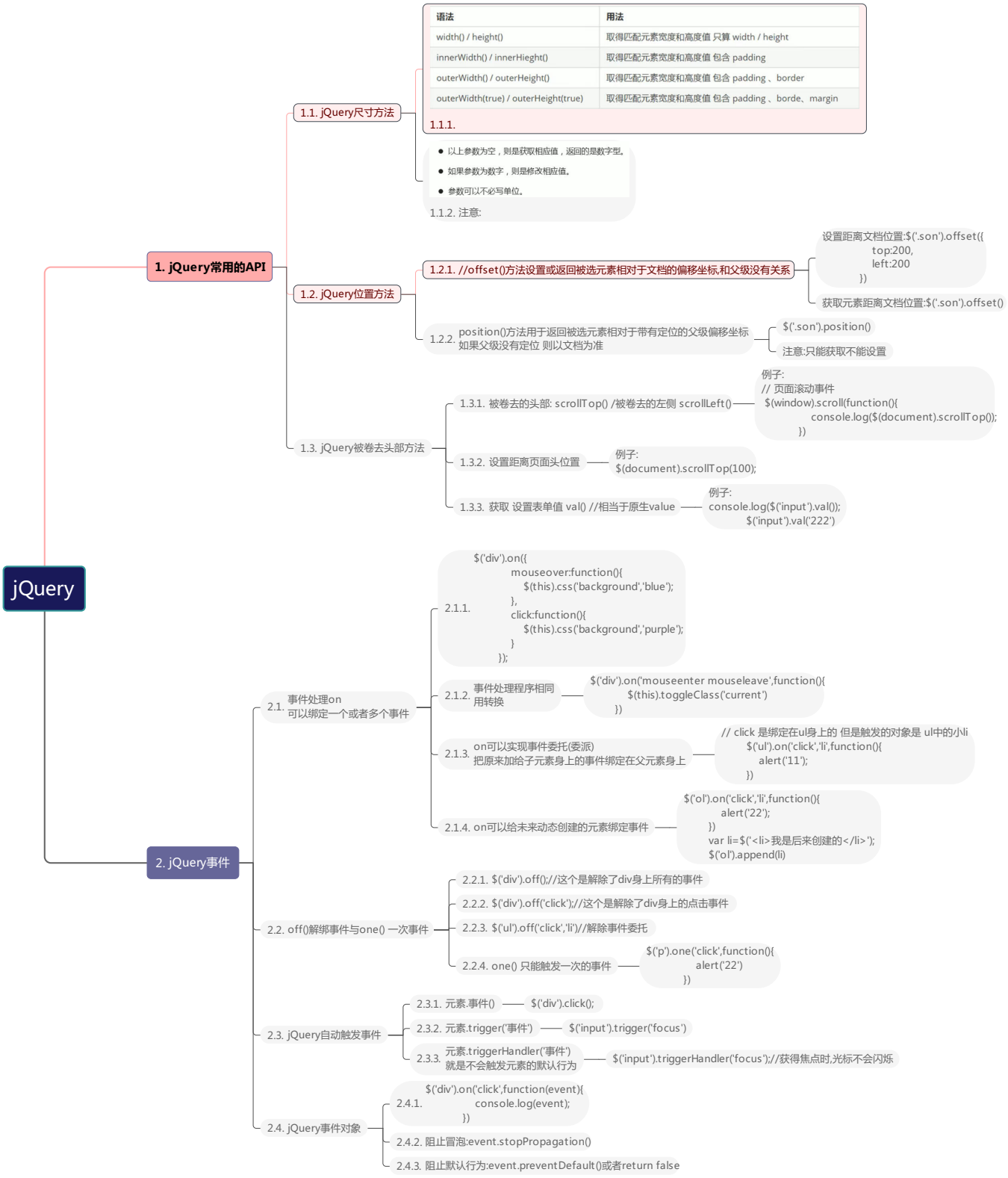
fn:在动画完成时执行的函数，每个元素执行一次。
      - 1.2.2. 滑动效果
        - 上拉滑动 — slideDown();
        - 下拉滑动 — slideUp();
        - 上 下拉切换滑动 — slideToggle();
      - 1.2.3. jQuery停止动画排队
        - 动画如果多次触发,就会造成多个动画或效果排队执行
解决方法:在效果前面加stop();
先把上一次动画停止 再执行这一次
      - 1.2.4. 淡入淡出效果
        - 淡入 fadeIn();
        - 淡出 fadeOut();
        - 淡入淡出切换 fadeToggle();
        - 修改透明度 fadeTo(1000,0.5); 这个速度和透明度必须要写
          - 四个参数 其中 透明度和速度必须要写
speed:三种预定速度之一的字符串("slow","normal", or "fast")或表示动画时长的毫秒数值(如：1000)

opacity:一个0至1之间表示透明度的数字。

easing:用来指定切换效果，默认是"swing"，可用参数"linear"

fn:在动画完成时执行的函数，每个元素执行一次。


jQuery
1. jQuery常用的API
1.1. jQuery效果
1.1.1. jQuery自定义动画 animate方法
语法:animate();
2. 参数
(1) params: 想要更改的样式属性，以对象形式传递，必须写。属性名可以不用带引号，如果是复合属性则需要采取驼峰命名法 borderLeft。其余参数都可以省略。
(2) speed : 三种预定速度之一的字符串(“slow” , “normal” , or “fast”)或表示动画时长的毫秒数值(如：1000)。
(3) easing : (Optional) 用来指定切换效果，默认是 “swing” ，可用参数 “linear” 。
(4) fn: 回调函数，在动画完成时执行的函数，每个元素执行一次。
例子 注意 第一个参数是对象 用{}包起来
$('div').animate({
left:200,
top:300,
width:300,
opacity:.4
},500)
1.2. jQuery属性操作
1.2.1. 获取元素固有属性
获取元素固有的属性值: element.prop("属性名")
设置属性值:prop("属性","属性值")
1.2.2. 元素的自定义属性
获取自定义属性: attr("属性") //类似于原生getAttribute
设置属性: attr("属性","属性值") //类似于原生setAttribute
1.2.3. 数据缓存data()
数据缓存 data() 这个里面的数据是存放在元素的内存里面
$('span').data('uname','andy')//类似于存放缓存
这个方法获取data-index h5自定义属性 第一个不用写data- 而且返回的是数字型
1.2.4. 补充: :checked选择器 可以查找被选中的表单元素
获取被选中的个数: $('元素:checked').length
1.3. jQuery内容文本值
1.3.1. 获取 设置元素内容 html() //相当于原生innerHtml
例子:
console.log($('div').html());
$('div').html('123')
1.3.2. 获取 设置元素文本内容 text() //相当于原生innerText
例子:
console.log($('div').text());
$('div').text('123')
1.3.3. 获取 设置表单值 val() //相当于原生value
例子:
console.log($('input').val());
$('input').val('222')
1.4. 补充:
1.4.1. parents('选择器') 可以返回指定祖先(它的父亲的父亲)元素
1.4.2. 可以通过: toFixed(数字) 来选择保留几位小数
1.5. jQuery元素操作
1.5.1. 遍历元素
1.语法: $(元素).each(function(index,domEle){ })
遍历dom元素时用
注意:
// 回调函数第一个参数一定是索引号 可以自己指定索引号名称
// 回调函数第二个参数一定是dom元素对象
// 因为 domEle 是dom对象 所以需要转化为jquery对象:$(domEle)
2. $.each(object,function(index,ele){ })
遍历数组 对象时用
1. $.each方法主要用于数据处理 比如数组 对象
//index是索引号 element是遍历内容
//遍历对象时 index输出的是属性名 ele输出属性值
1.5.2. 创建元素
var li=$('
我是后来创建的小li
');
1.5.3. 添加元素
内部添加(添加子节点)
$('ul').append(li);//内部添加 并且放到内容的最后面(在ul里面添加li)
$('ul').prepend(li);//内部添加并且放到内容的最前面
外部添加(添加兄弟节点)
$('.text').after(div);//在元素后面添加
$('.text').before(div);//在元素前面添加
1.5.4. 删除元素
$('.text').remove(); 可以删除对应的元素
$('ul').empty()//可以删除对应元素里面的子节点
$('ul').html('');//可以删除对应元素里面的子节点


jQuery
1. jQuery常用的API
1.1. jQuery尺寸方法
语法
用法
width() / height()
取得匹配元素宽度和高度值 只算 width / height
innerWidth() / innerHieght()
取得匹配元素宽度和高度值 包含 padding
outerWidth() / outerHeight()
取得匹配元素宽度和高度值 包含 padding、border
outerWidth(true) / outerHeight(true)
取得匹配元素宽度和高度值 包含 padding、borde、margin
1.1.1.
以上参数为空，则是获取相应值，返回的是数字型。
如果参数为数字，则是修改相应值。
参数可以不必写单位。
1.1.2. 注意:
1.2. jQuery位置方法
1.2.1. //offset()方法设置或返回被选元素相对于文档的偏移坐标,和父级没有关系
设置距离文档位置:$('.son').offset({ top:200, left:200 })
获取元素距离文档位置:$('.son').offset()
1.2.2. position()方法用于返回被选元素相对于带有定位的父级偏移坐标 如果父级没有定位 则以文档为准
$('.son').position()
注意:只能获取不能设置
1.3. jQuery被卷去头部方法
1.3.1. 被卷去的头部: scrollTop() /被卷去的左侧 scrollLeft()
例子:
// 页面滚动事件
$(window).scroll(function(){ console.log($(document).scrollTop()); })
1.3.2. 设置距离页面头位置
例子:
$(document).scrollTop(100);
1.3.3. 获取 设置表单值 val() //相当于原生value
例子:
console.log($('input').val());
$('input').val('222')
2. jQuery事件
2.1. 事件处理on 可以绑定一个或者多个事件
2.1.1.
$('div').on({ mouseover:function(){ $(this).css('background','blue'); }, click:function(){ $(this).css('background','purple'); } });
2.1.2. 事件处理程序相同 用转换
$('div').on('mouseenter mouseleave',function(){ $(this).toggleClass('current') })
2.1.3. on可以实现事件委托(委派) 把原来加给子元素身上的事件绑定在父元素身上
// click 是绑定在ul身上的 但是触发的对象是 ul中的小li
$('ul').on('click','li',function(){ alert('11'); })
2.1.4. on可以给未来动态创建的元素绑定事件
$('ol').on('click','li',function(){ alert('22'); })
var li=$('
我是后来创建的
');
$('ol').append(li)
2.2. off()解绑事件与one() 一次事件
2.2.1. $('div').off();//这个是解除了div身上所有的事件
2.2.2. $('div').off('click');//这个是解除了div身上的点击事件
2.2.3. $('ul').off('click','li')//解除事件委托
2.2.4. one() 只能触发一次的事件
$('p').one('click',function(){ alert('22') })
2.3. jQuery自动触发事件
2.3.1. 元素.事件()
$('div').click();
2.3.2. 元素.trigger('事件')
$('input').trigger('focus')
2.3.3. 元素.triggerHandler('事件') 就是不会触发元素的默认行为
$('input').triggerHandler('focus');//获得焦点时,光标不会闪烁
2.4. jQuery事件对象
2.4.1.
$('div').on('click',function(event){ console.log(event); })
2.4.2. 阻止冒泡:event.stopPropagation()
2.4.3. 阻止默认行为:event.preventDefault()或者return false

Ajax

Ajax

URL

URL地址一般由三部组成:

① 客户端与服务器之间的通信协议

② 存有该资源的服务器名称

③ 资源在服务器上具体的存放位置

http://www.cnblogs.com/liulongbinblogs/p/11649393.html

通信协议 | 服务器名称 | 资源在服务器上具体的存放位置

请求服务器的数据资源

XMLHttpRequest（简称 xhr）是浏览器提供的 js 成员，通过它，可以请求服务器上的数据资源。

最简单的用法 var xhrObj = new XMLHttpRequest()

客户端请求服务器资源的请求方式

- get请求
  - get 请求通常用于获取服务端资源（向服务器要资源）
  - 例如：根据 URL 地址，从服务器获取 HTML 文件、css 文件、js文件、图片文件、数据资源等
- post请求
  - post 请求通常用于向服务器提交数据（往服务器发送资源）
  - 例如：登录时向服务器提交的登录信息、注册时向服务器提交的注册信息、添加用户时向服务器提交的用户信息等各种数据提交操作

jQuery中发起Ajax请求

从服务器获取数据 — $.get()

```
// 1.$.get发起不带参数的请求
$('button').on('click',function(){
  $.get('http://www.liulongbin.top:3006/api/getbooks',function(res){
      console.log(res);
    })
  })

// 2.$.get发起带参数的请求
$('button').on('click',function(){
  $.get('http://www.liulongbin.top:3006/api/getbooks',{id:1},function(res){
      // 请求id为1的图书信息
      console.log(res);
    })
```

给服务器发送数据 — $.post()

```
$.post('http://www.liulongbin.top:3006/api/addbook',{
    bookname:'水浒传',
    author:'施耐庵',
    publisher:'上海图书出版社'
  },function(res){
    console.log(res);
  })
```

可获取可发送 — $.ajax()

注意:type里面的 大小写都可以

使用$.ajax发起get请求

```
$('.btn1').on('click',function(){
    $.ajax({
      type:'GET',
      url:'http://www.liulongbin.top:3006/api/getbooks',
      data:{ id:1 },
      success:function(res){
        console.log(res);
      }
    })
  })
```

使用$.ajax发起post请求

```
$('.btn2').on('click',function(){
    $.ajax({
      type:'POST',
      url:'http://www.liulongbin.top:3006/api/addbook',
      data:{ //要提交给服务器的数据
        bookname:'水浒传',
        author:'施耐庵',
        publisher:'上海图书出版社'},
      success:function(res){
        console.log(res);
      }
    })
  })
```

补充:trim()可以去除空格

接口

使用 Ajax 请求数据时，被请求的 URL 地址，就叫做数据接口（简称接口）。同时，每个接口必须有请求方式。

测试工具 — postman

Ajax

- 重置滚动条位置
使滚动条定位到最底部 — resetui();
- e.keyCode可以获取到当前按键的编码
- form表单
  - form表单的属性
    - action 属性用来规定当提交表单时，**向何处发送表单数据。**
      action 属性的值应该是后端提供的一个 URL 地址，这个 URL 地址专门负责接收表单提交过来的数据。
      当 <form> 表单在未指定 action 属性值的情况下，action 的默认值为当前页面的 URL 地址。
      注意:当提交表单后,页面会立即跳转到action属性指定的url地址
    - target属性 — target属性用来规定在何处打开action URL
它的值:
_blank 在新窗口中打开
_self 默认值 在相同的框架中打开
    - method属性
      - method 属性用来规定**以何种方式**把表单数据提交到 action URL。
        它的可选值有两个，分别是 get 和 post。
        默认情况下，method 的值为 get，表示通过URL地址的形式，把表单数据提交到 action URL。
      - get 方式适合用来提交少量的、简单的数据。
        post 方式适合用来提交**大量的、复杂的、或包含文件上传**的数据。
      - 实际开发中 post 用的最多
    - enctype

值	描述
application/x-www-form-urlencoded	在发送前编码所有字符（默认）
multipart/form-data	不对字符编码。 在使用包含文件上传控件的表单时，必须使用该值。

    - enctype属性用来规定在发送表单数据之前如何对数据进行编码
  - 表单同步提交的缺点 — 1.整个页面会发生跳转 跳转到action URL所指向的地址 用户体验差
2.页面之前的状态和数据会丢失
  - 通过Ajax提交表单数据
    -
```
$('#f1').on('submit',function(){
      alert('监听到了表单的提交')
    })
```
    - 阻止表单提交和页面跳转行为 — e.preventDefault()
    - 快速获取表单中的数据 — serialize()
好处:可以一次性获取到表单中的所有数据
注意:要使用serialize() 必须要为每个表单元素添加name属性
```
$('#f1').on('submit',function(e){
      // 阻止表单的默认提交行为
      e.preventDefault()
      var data=$(this).serialize()
      console.log(data);
    })
```
  - 模板引擎
    - art-template模板引擎的基本使用
```
<!-- 导入模板引擎 -->
  <script src="./lib/template-web.js"></script>

// 2.定义要渲染的数据 写在script标签中
    var data={ name:'zs'}

<!-- 3.定义模板 -->
  <!-- 模板的html结构 必须定义到script中 独立script -->
  <script type="text/html">
    <h1>{{name}}</h1>  //{{}}是占位符 里面放需要显示的数据
</script>

// 4.调用template模板渲染数据
 // template('模板的id',需要渲染的数据对象)
var str=template('tpl-user',data)

// 5.渲染html结构 将内容显示到页面上
    $('.container').html(str)
```
    - art-template基本语法
      - {{}}
可以进行变量输出,或者循环数组等操作
      - {{@ value}} 可以进行原文输出
      - 条件输出:
```
{{if flag===0}}
 flag的值是0
 {{else if flag===1}}
 flag的值是1
  {{/if}}  //这是结束标志
```
      - 循环输出
```
{{each hobby}}
      <li>循环的索引是{{$index}},循环的值是{{$value}}</li>
      {{/each}}
```
      - 过滤器
```
定义得到年月日的过滤器 注意一定要定义在最前面!
template.defaults.imports.dateFormat=function(date){
      var y=date.getFullYear()
      var m=date.getMonth()+1
      var d=date.getDate()
      return y+'-'+m+'-'+d //注意:过滤器最后一定要return一个值
    }

<h3>{{regTime | dateFormat}}</h3>   //值->新值
```

Ajax

模板引擎

正则

exec函数

exec() 函数用于检索字符串中的正则表达式的匹配。

如果字符串中有匹配的值，则返回该匹配值，否则返回 null。

```
var str='hello'
 var pattern=/o/
 var result=pattern.exec(str)
 console.log(result);
```

分组

正则表达式中 ( ) 包起来的内容表示一个分组，可以通过分组来提取自己想要的内容，

```
var str=' <div>我是{{name}}</div>'
    var pattern=/{{([a-zA-Z]+)}}/
    var result=pattern.exec(str)
    console.log(result);
```

replace函数

用于在字符串中用一些字符替换另一些字符

```
var str=' <div>我是{{name}}</div>'
    var pattern=/{{([a-zA-Z]+)}}/
    var result=pattern.exec(str)
    // 0: "{{name}}"
    // 1: "name"
    str=str.replace(result[0],result[1])
    console.log(str);
```

多次replace

```
var str1='<div>我是{{name}}今年{{age}}岁/div>'
var pattern1=/{{\s*([a-zA-Z]+)\s*}}/
var endresult=null
    while(endresult=pattern1.exec(str1)){
      str1=str1.replace(endresult[0],endresult[1])
    }
    console.log(str1);
```

Ajax高级

XMLHttpRequest

XMLHttpRequest（简称 xhr）是浏览器提供的 Javascript 对象，通过它，可以请求服务器上的数据资源。之前所学的 jQuery 中的 Ajax 函数，就是基于 xhr 对象封装出来的。

使用xhr发起get请求

```
// 1.创建xhr对象
  var xhr=new XMLHttpRequest()
  // 2.调用open函数
  xhr.open('GET','http://www.liulongbin.top:3006/api/getbooks')
  // 3.调用send函数发起请求
  xhr.send()
  // 4.监听 onreadystatechange 事件
  xhr.onreadystatechange=function(){
    // 这个是固定写法
    if(xhr.readyState===4&&xhr.status===200){
      // 获取服务器响应的数据
      console.log(xhr.responseText);
    }
  }
```

发起带参数的get请求

在调用xhr.open期间 为URL地址指定参数

```
xhr.open('GET','http://www.liulongbin.top:3006/api/getbooks?id=1')
```

查询字符串

是指在URL末尾加上用于向服务器发送信息的字符串(变量)

格式：将英文的？放在URL 的末尾，然后再加上 参数=值，想加上多个参数的话，使用 & 符号进行分隔。以这个形式，可以将想要发送给服务器的数据添加到 URL 中。

URL编码与解码

URL编码 —— encodeURI()

URL解码 —— decodeURI()

使用xhr发起post请求

```
var xhr=new XMLHttpRequest()
  xhr.open('POST','http://www.liulongbin.top:3006/api/addbook')
  // 3.设置Content-Type属性
  xhr.setRequestHeader('Content-Type','application/x-www-form-urlencoded')
  // 4.调用send函数
  xhr.send('bookname=水浒传&&author=施耐庵&&publisher=上海图书出版社')
  xhr.onreadystatechange=function(){
    if(xhr.readyState===4&&xhr.status===200){
      console.log(xhr.responseText);
    }
  }
```

Ajax

XMLHttpRequest

数据交换格式

数据交换格式，就是服务器端与客户端之间进行数据传输与交换的格式。

前端领域，经常提及的两种数据交换格式分别是 XML 和 JSON。其中 XML 用的非常少，所以，我们重点要学习的数据交换格式就是 JSON。

xml:标记语言 —— xml被设计用来传输和存储数据,是数据的载体

json

就是JavaScript对象和数组的字符串表示法.本质是字符串
也是用于传输和存储数据

json的两种结构

对象结构：对象结构在 JSON 中表示为 { } 括起来的内容。数据结构为 { key: value, key: value, ... } 的键值对结构。其中，key 必须是使用英文的双引号包裹的字符串，value 的数据类型可以是数字、字符串、布尔值、null、数组、对象6种类型。

示例代码

```
{
    "name": "zs",
    "age": 20,
    "gender": "男",
    "address": null,
    "hobby": ["吃饭", "睡觉", "打豆豆"]
}
```

数组结构：数组结构在 JSON 中表示为 [ ] 括起来的内容。数据结构为 ["java", "javascript", 30, true ...]。数组中数据的类型可以是数字、字符串、布尔值、null、数组、对象6种类型。

```
[ "java", "python", "php" ]
```

json和js对象的关系

JSON 是 JS 对象的字符串表示法，它使用文本表示一个 JS 对象的信息，本质是一个字符串。

```
//这是一个对象
var obj = {a: 'Hello', b: 'World'}

//这是一个 JSON 字符串，本质是一个字符串
var json = '{"a": "Hello", "b": "World"}'
```

json与js对象相互转换

json转换为js对象 —— var jsonStr='{"a":"hello","b":"world"}'
JSON反序列化 —— JSON.parse(jsonStr)

js对象转换为json —— var obj={a:'hello',b:'world',c:false}
JSON序列化 —— JSON.stringify(obj)

2. XMLHttpRequest Level2的新功能

① 可以设置 HTTP 请求的时限
② 可以使用 FormData 对象管理表单数据
③ 可以上传文件
④ 可以获得数据传输的进度信息

设置http请求时限

```
// 设置超时时间
    xhr.timeout=3000
    // 设置超时以后的处理函数
    xhr.timeout=function(){
      console.log('请求超时');
    }
```

- Ajax
  - XMLHttpRequest Level2新功能
    - 使用FormData对象管理表单数据

```
// 创建FormData实例
var fd=new FormData()
// 调用append函数,向fd中追加数据
fd.append('uname','zs')
fd.append('upwd','123456')
var xhr=new XMLHttpRequest()
xhr.open('POST','http://www.liulongbin.top:3006/api/formdata')
xhr.send(fd)
xhr.onreadystatechange=function(){
  if(xhr.readyState===4&&xhr.status===200){
    console.log(JSON.parse(xhr.responseText));
  }
}
```

    - 上传文件
      - 把文件追加到FormData中

```
var fd=new FormData()
fd.append('avatar',files[0])
```

      - 使用xhr发起上传文件请求

```
var xhr=new XMLHttpRequest()
xhr.open('post','http://www.liulongbin.top:3006/api/upload/avatar')
xhr.send(fd)
```

      - 监听onreadystatechange事件

```
xhr.onreadystatechange=function(){
  if(xhr.readyState===4&&xhr.status===200){
    var data=JSON.parse(xhr.responseText)
    if(xhr.status===200){
      // 上传成功
      document.querySelector('#img').src='http://www.liulongbin.top:3006'+data.url;
    }else{
      console.log('上传失败'+data.message);
    }
  }
}
```

    - 显示文件上传进度

```
xhr.upload.onprogress=function(e){
  if(e.lengthComputable){
    // 计算出上传的进度  已传输的字节/总字节
    var procentComplate=Math.ceil( e.loaded/e.total)*100
    console.log(procentComplate);
  }
}
```

  - 使用jQuery上传文件

```
$('#btnUpload').on('click',function(){
  var files=$('#file1')[0].files
  if(files.length<=0){
    return alert('请选择上传的文件')
  }else{
    // console.log('ok');
    var fd=new FormData()
    fd.append('avatar',files[0])
    $.ajax({
      method:'POST',
      url:'http://www.liulongbin.top:3006/api/upload/avatar',
      data:fd,
      // 固定写法
      processData:false,
      contentType:false,
      success:function(res){
        console.log(res);
      }
    })
  }
})
```

补充:

```
// 监听到Ajax请求被发起了
$(document).ajaxStart(function(){
  $('#loading').show()
})
// 监听到Ajax完成的事件
$(document).ajaxStop(function(){
  $('#loading').hide()
})
```

- Ajax
  - 跨域和JSONP
    - **同源**:如果两个页面的协议 域名和端口都相同,则两个页面具有相同的源,反之,则是**跨域**
    - 同源策略:是浏览器提供的一个安全功能.例如:浏览器规定,A网站的JavaScript,不允许和非同源的网站C之间,进行资源的交互
      注意:浏览器允许发起跨域请求,但是,跨域请求回来的数据,会被浏览器拦截,无法被页面获取
    - 实现跨域数据请求 两种方法:
    - JSONP：出现的早，兼容性好（兼容低版本IE）。是前端程序员为了解决跨域问题，被迫想出来的一种临时解决方案。缺点是只支持 GET 请求，不支持 POST 请求。
    - CORS：出现的较晚，它是 W3C 标准，属于跨域 Ajax 请求的根本解决方案。支持 GET 和 POST 请求。缺点是不兼容某些低版本的浏览器。
    - JSONP实现原理 可以通过查询字符串的方式 告诉外部服务器 应该调用哪个函数 注意:JSONP并没有用到XMLHttpRequest请求
    - 因此，JSONP 的实现原理，就是通过 <script> 标签的 src 属性，请求跨域的数据接口，并通过函数调用的形式，接收跨域接口响应回来的数据。
    - 使用jquery发起JSONP请求

```
$.ajax({
        url:'http://ajax.frontend.itheima.net:3006/api/jsonp?name=zs&age=20',
        dataType:'jsonp',
        // 发送到服务端的参数名称 默认值为callback
        jsonp:'callback',
        // 自定义的回调函数名称,默认值为jQueryxxx格式
        jsonpCallback:'abc',
        success:function(res){
          console.log(res);
        }
      })
```

    - 默认情况下,使用jquery发起jsonp请求,会自动携带一个callback=jQueryxxx参数
      jQueryxxx是随机生成的回调函数名称
  - 防抖策略
    - 是当事件被触发后,延迟n秒后再执行回调,如果在这n秒内事件又被触发,则重新计时
    - 用户在输入框中连续输入一串字符时,可以通过防抖策略,只在输入完后,彩之星查询的请求,可以有效减少请求次数

```
// 1.定义延时器的id
    var timer=null
    // 2.定义防抖函数
    function debounceSearch(kw){
      timer=setTimeout(function(){
      //获取用户输入的数据
        getSuggestList(kw)
      },500)
    }
3.// 清空timer
    clearTimeout(timer)
```

    - 缓存

```
// 定义缓存区域
 var cachObj={}

// 1.获取到用户输入的内容 当做键
  var k=$('#ipt').val().trim()
  // 2.需要将数据作为值,进行缓存
    cachObj[k]=res
```

    - 基本思想:当用户在搜索框输入一个值之后 会出来对应的建议项 将这些建议项先放在缓存区域
      再追加一个值 会出来一个新的建议项
      如果之后删掉了追加的值 则不会重复发起请求 而是将缓存区域的值显示出来
  - 节流
    减少一段时间内事件的触发频率 防止事件被无限制触发
    - 节流应用场景
    - 鼠标连续不断地触发某事件（如点击），只在单位时间内只触发一次；
    - 懒加载时要监听计算滚动条的位置，但不必每次滑动都触发，可以降低计算的频率，而不必去浪费 CPU 资源；
    - 节流阀为空，表示可以执行下次操作；不为空，表示不能执行下次操作。
    - 当前操作执行完，必须将节流阀重置为空，表示可以执行下次操作了。
    - 每次执行操作前，必须先判断节流阀是否为空。

Ajax

HTTP协议加强

客户端与服务器之间要实现网页内容的传输，则通信的双方必须遵守网页内容的传输协议。

网页内容又叫做**超文本**，因此网页内容的传输协议又叫做**超文本传输协议**（HyperText Transfer Protocol）简称 **HTTP 协议**。

HTTP协议采取了**请求/响应**交互模型

HTTP请求消息(报文) 由请求行 请求头部 空行和请求体组成

由于 HTTP 协议属于客户端浏览器和服务器之间的通信协议。因此，客户端发起的请求叫做 **HTTP 请求**，客户端发送到服务器的消息，叫做 **HTTP 请求消息**。

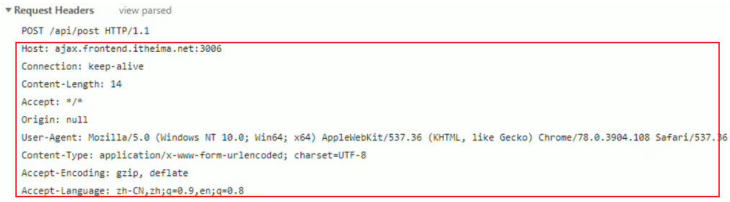
HTTP请求消息

请求行(请求方式 URL HTTP协议版本)

请求头部

头部字段	说明
Host	要请求的服务器域名
Connection	客户端与服务器的连接方式(close 或 keepalive)
Content-Length	用来描述请求体的大小
Accept	客户端可识别的响应内容类型列表
User-Agent	产生请求的浏览器类型
Content-Type	客户端告诉服务器实际发送的数据类型
Accept-Encoding	客户端可接收的内容压缩编码形式
Accept-Language	用户期望获得的自然语言的优先顺序

头部字段示例



请求体(存放着要通过POST方式提交到服务器的数据)
只有**POST**请求才有请求体 **GET**请求没有请求体

Ajax

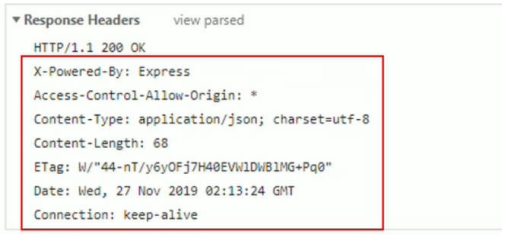
HTTP协议加强

HTTP响应消息

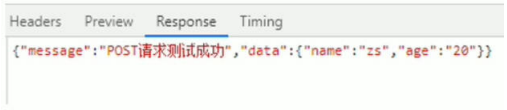
由状态行 响应头部 空行和响应体组成

状态行(HTTP协议版本 状态码 状态码的描述文本)

响应头部



响应体



HTTP请求方法

序号	方法	描述
1	GET	(查询)发送请求来获得服务器上的资源，请求体中不会包含请求数据，请求数据放在协议头中。
2	POST	(新增)向服务器提交资源（例如提交表单或上传文件）。数据被包含在请求体中提交给服务器。
3	PUT	(修改)向服务器提交资源，并使用提交的新资源，替换掉服务器对应的旧资源。
4	DELETE	(删除)请求服务器删除指定的资源。

HTTP响应状态码

也属于HTTP协议的一部分,用来表示响应的状态

200 即是状态码 ok表示状态码描述信息

HTTP状态码的类型

2**	成功，操作被成功接收并处理
3**	重定向，需要进一步的操作以完成请求
4**	客户端错误，请求包含语法错误或无法完成请求
5**	服务器错误，服务器在处理请求的过程中发生了错误

2xx状态码

状态码	状态码英文名称	中文描述
200	OK	请求成功。一般用于 GET 与 POST 请求
201	Created	已创建。成功请求并创建了新的资源，通常用于 POST 或 PUT 请求

3xx状态码

状态码	状态码英文名称	中文描述
301	Moved Permanently	永久移动。请求的资源已被永久的移动到新URI，返回信息会包括新的URI，浏览器会自动定向到新URI。今后任何新的请求都应使用新的URI代替
302	Found	临时移动。与301类似。但资源只是临时被移动。客户端应继续使用原有URI
304	Not Modified	未修改。所请求的资源未修改，服务器返回此状态码时，不会返回任何资源（响应消息中不包含响应体）。客户端通常会缓存访问过的资源。

4xx状态码

404	Not Found	服务器无法根据客户端的请求找到资源（网页）

5xx状态码

状态码	状态码英文名称	中文描述
500	Internal Server Error	服务器内部错误，无法完成请求。
501	Not Implemented	服务器不支持该请求方法，无法完成请求。只有 GET 和 HEAD 请求方法是要求每个服务器必须支持的，其它请求方法在不支持的服务器上会返回501
503	Service Unavailable	由于超载或系统维护，服务器暂时的无法处理客户端的请求。

Ajax

Axios

- Axios是专注于网络数据请求的库
- 基于axios发起get请求

```
document.querySelector('#btn').addEventListener('click',function(){
    var url='http://www.liulongbin.top:3006/api/get'
    var paramsObj={name:'zs',age:20}
    axios.get(url,{params:paramsObj}).then(function(res){
      console.log(res.data);//服务器返回的数据
    })
  })
```

- 基于axios发起post请求

```
document.querySelector('#btn2').addEventListener('click',function(){
    var url='http://www.liulongbin.top:3006/api/post'
    var dataObj={address:'北京',location:'顺义区'}
    axios.post(url,dataObj).then(function(res){
      console.log(res.data);//服务器返回的数据
    })
  })
```

- 直接使用axios发起请求

```
axios({
    method: '请求类型',
    url: '请求的URL地址',
    data: { /* POST数据 */ },
    params: { /* GET参数 */ }
}) .then(callback)
```

```
axios({
    method:'GET',
    url:'http://www.liulongbin.top:3006/api/get',
    params:{
      name:'ls',
      age:23
    }
  }).then(function(res){
    console.log(res.data);
  })
```

```
axios({
    method:'POST',
    url:'http://www.liulongbin.top:3006/api/post',
    data:{
      name:'钢铁侠',
      age:23
    }
  }).then(function(res){
    console.log(res.data);
  })
```

Ajax

Git
分布式版本控制系统

Git记录快照

Git 快照是在原有文件版本的基础上重新生成一份新的文件，类似于备份。为了效率，如果文件没有修改，Git 不再重新存储该文件，而是只保留一个链接指向之前存储的文件。

Git管理的项目 有三个区域 分别是工作区 暂存区 Git仓库

Git三种状态

- 工作区的文件被修改了，但还没有放到暂存区，就是已修改状态。
- 如果文件已修改并放入暂存区，就属于已暂存状态。
- 如果 Git 仓库中保存着特定版本的文件，就属于已提交状态。

Git基本操作

使用Git管理文件 —— 在目标文件夹右击 Git Bash Here —— 在终端输入 **git init**

查看文处于什么状态 —— **git status -s**

跟踪文件 —— **git add** 要跟踪文件的名字

提交更新 —— **git commit -m "新建了index.html文件"**

现在暂存区中有一个 index.html 文件等待被提交到 Git 仓库中进行保存。可以执行 git commit 命令进行提交，其中 -m 选项后面是本次的提交消息，用来对提交的内容做进一步的描述：

对已提交的文件进行修改 —— 文件 index.html 出现在 Changes not staged for commit 这行下面，说明**已跟踪文件的内容发生了变化，但还没有放到暂存区**。

注意：修改过的、没有放入暂存区的文件前面有红色的 M 标记。

暂存已修改的文件 —— 再次运行git add命令

是个多功能的命令，主要有如下 3 个功效：

① 可以用它开始跟踪新文件

② 把已跟踪的、且已修改的文件放到暂存区

③ 把有冲突的文件标记为已解决状态

撤销对文件的修改 —— **git checkout -- 文件名字**

撤销对文件的修改指的是：把对工作区中对应文件的修改，**还原**成 Git 仓库中所保存的版本。

操作的结果：所有的修改会丢失，且无法恢复！危险性比较高，请慎重操作！

向暂存区中一次性添加多个文件 —— **git add .**

取消暂存的文件 —— **git reset HEAD** 要移除的文件名称(从暂存区中移除对应的文件)
git reset HEAD .(移除所有的文件)

跳过使用暂存区,直接提交到Git仓库 —— **git commit -a -m "描述消息"**

移除文件 ——

```
# 从 Git 仓库和工作区中同时移除 index.js 文件
git rm -f index.js
# 只从 Git 仓库中移除 index.css，但保留工作区中的 index.css 文件
git rm --cached index.css
```

Ajax

Git
分布式版本控制系统

Git基本操作

忽略文件

有些文件无需纳入Git管理--创建一个名为.gitignore的配置文件

格式规范

① 以 # 开头的是注释
② 以 / 结尾的是目录
③ 以 / 开头防止递归
④ 以 ! 开头表示取反
⑤ 可以使用 glob 模式进行文件和文件夹的匹配（glob 指简化了的正则表达式）

glob模式

所谓的 glob 模式是指简化了的正则表达式：

① **星号 *** 匹配零个或多个任意字符
② **[abc]** 匹配任何一个列在方括号中的字符 （此案例匹配一个 a 或匹配一个 b 或匹配一个 c）
③ **问号 ?** 只匹配一个任意字符
④ 在方括号中使用**短划线**分隔两个字符， 表示所有在这两个字符范围内的都可以匹配（比如 [0-9] 表示匹配所有 0 到 9 的数字）
⑤ **两个星号 **** 表示匹配任意中间目录（比如 a/**/z 可以匹配 a/z 、 a/b/z 或 a/b/c/z 等）

```
# 忽略所有的 .a 文件
*.a

# 但跟踪所有的 lib.a，即便你在前面忽略了 .a 文件
!lib.a

# 只忽略当前目录下的 TODO 文件，而不忽略 subdir/TODO
/TODO

# 忽略任何目录下名为 build 的文件夹
build/

# 忽略 doc/notes.txt，但不忽略 doc/server/arch.txt
doc/*.txt

# 忽略 doc/ 目录及其所有子目录下的 .pdf 文件
doc/**/*.pdf
```

查看提交历史

git log 查看所有的提交历史

git log -数字 只查看最新的几条提交历史

git log --pretty=oneline 在一行上展示所有的提交历史

回退到指定的版本

git reset --hard <CommitID> <CommitID>是提交的id 可以在提交历史里查看

git reflog --pretty=oneline 如果退回旧版本之后 还想查看所有的提交历史

Ajax

Git
分布式版本控制系统

Git基本操作

忽略文件

有些文件无需纳入Git管理--创建一个名为**.gitignore**的配置文件

格式规范

① 以 **# 开头**的是注释
② 以 **/ 结尾**的是目录
③ 以 **/ 开头**防止递归
④ 以 **! 开头**表示取反
⑤ 可以使用 **glob 模式**进行文件和文件夹的匹配（glob 指简化了的正则表达式）

glob模式

所谓的 **glob 模式**是指简化了的正则表达式：

① **星号 *** 匹配**零个或多个任意字符**
② **[abc]** 匹配**任何一个列在方括号中的字符**（此案例匹配一个 a 或匹配一个 b 或匹配一个 c）
③ **问号 ?** 只**匹配一个任意字符**
④ 在方括号中使用**短划线**分隔两个字符， 表示所有在这两个字符范围内的都可以匹配（比如 [0-9] 表示匹配所有 0 到 9 的数字）
⑤ **两个星号 **** 表示**匹配任意中间目录**（比如 a/**/z 可以匹配 a/z 、 a/b/z 或 a/b/c/z 等）

```
# 忽略所有的 .a 文件
*.a

# 但跟踪所有的 lib.a，即便你在前面忽略了 .a 文件
!lib.a

# 只忽略当前目录下的 TODO 文件，而不忽略 subdir/TODO
/TODO

# 忽略任何目录下名为 build 的文件夹
build/

# 忽略 doc/notes.txt，但不忽略 doc/server/arch.txt
doc/*.txt

# 忽略 doc/ 目录及其所有子目录下的 .pdf 文件
doc/**/*.pdf
```

查看提交历史

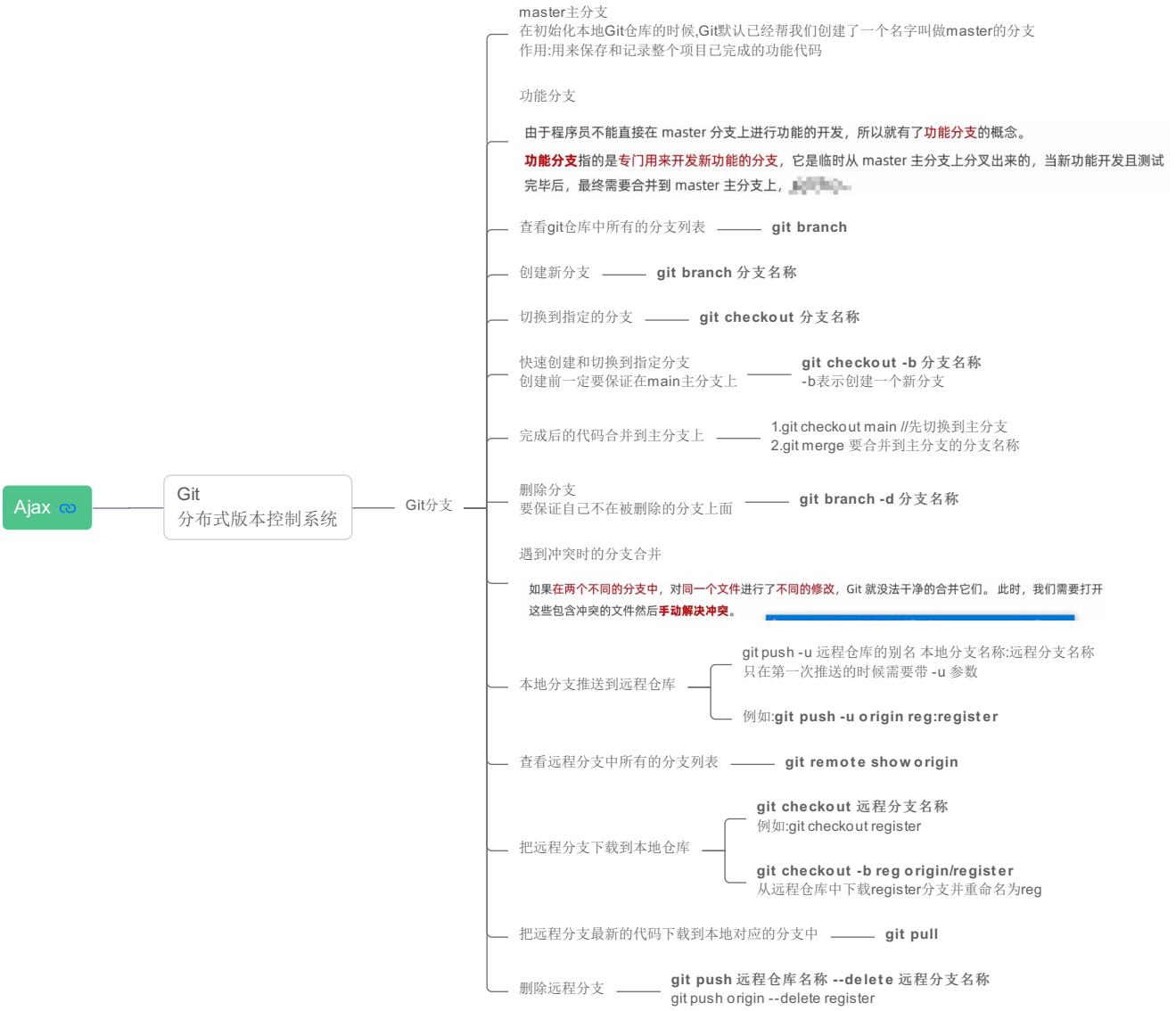
git log 查看所有的提交历史
git log -数字 只查看最新的几条提交历史
git log --pretty=oneline 在一行上展示所有的提交历史

回退到指定的版本

git reset --hard <CommitID> <CommitID>是提交的id 可以在提交历史里查看
git reflog --pretty=oneline 如果退回旧版本之后 还想查看所有的提交历史

git push 将本地的最新代码同步到GitHub仓库

git clone 仓库地址 将远程仓库克隆到本地


Ajax
Git
分布式版本控制系统
Git分支
master主分支
在初始化本地Git仓库的时候,Git默认已经帮我们创建了一个名字叫做master的分支
作用:用来保存和记录整个项目已完成的功能代码
功能分支
由于程序员不能直接在 master 分支上进行功能的开发，所以就有了功能分支的概念。
功能分支指的是专门用来开发新功能的分支，它是临时从 master 主分支上分叉出来的，当新功能开发且测试完毕后，最终需要合并到 master 主分支上，
查看git仓库中所有的分支列表
git branch
创建新分支
git branch 分支名称
切换到指定的分支
git checkout 分支名称
快速创建和切换到指定分支
创建前一定要保证在main主分支上
git checkout -b 分支名称
-b表示创建一个新分支
完成后的代码合并到主分支上
1.git checkout main //先切换到主分支
2.git merge 要合并到主分支的分支名称
删除分支
要保证自己不在被删除的分支上面
git branch -d 分支名称
遇到冲突时的分支合并
如果在两个不同的分支中，对同一个文件进行了不同的修改，Git 就没法干净的合并它们。 此时，我们需要打开这些包含冲突的文件然后手动解决冲突。
本地分支推送到远程仓库
git push -u 远程仓库的别名 本地分支名称:远程分支名称
只在第一次推送的时候需要带 -u 参数
例如:git push -u origin reg:register
查看远程分支中所有的分支列表
git remote show origin
把远程分支下载到本地仓库
git checkout 远程分支名称
例如:git checkout register
git checkout -b reg origin/register
从远程仓库中下载register分支并重命名为reg
把远程分支最新的代码下载到本地对应的分支中
git pull
删除远程分支
git push 远程仓库名称 --delete 远程分支名称
git push origin --delete register

node.js

node.js

了解node.js

- Node.js是一个基于 Chrome V8 引擎 的 JavaScript 运行环境。
- 浏览器是JavaScript的前端运行环境 node.js是JavaScript的后端运行环境
- node.js中无法调用DOM和BOM等浏览器内置API
- 运行js文件

```
E:\前端\node.js>node 1.js
hello node.js
```

- 终端中的快捷键
  - ① 使用 ↑ 键，可以快速定位到上一次执行的命令
  - ② 使用 tab 键，能够快速补全路径
  - ③ 使用 esc 键，能够快速清空当前已输入的命令
  - ④ 输入 cls 命令，可以清空终端

fs文件系统模块

- 导入 `const fs=require('fs')`
- 调用fs.readFile()方法读取文件

```
fs.readFile('./file/1.txt','utf8',function(err,dataStr){
    console.log(err);
//如果读取成功,则err的值为null
//如果读取失败 则err的值为错误对象 dataStr的值为undefined
    console.log('--------------');
    console.log(dataStr);
})
```

- 参数

```
参数1：读取文件的存放路径
参数2：读取文件时候采用的编码格式，一般默认指定 utf8
参数3：回调函数，拿到读取失败和成功的结果  err  dataStr
```

- 调用fs.writeFile()写入文件内容

```
const fs=require('fs');
fs.writeFile('./file/2.txt','abcd',function(err){
    console.log(err);
})
```

- 参数
  - 参数1:表示文件的存放路径
  - 参数2:表示要写入的内容
  - 参数3:回调函数
- 判断文件是否写入成功

```
if(err){
    return console.log()('文件写入失败'+err.message);
}else{
    console.log('文件写入成功');
}
```

- 关于文件路径问题: __dirname表示当前文件所处的目录 __dirname+'/file/1.txt' 不会出现路径拼接问题 同时 可移植性和维护性也很好

path()路径模块

- 导入:`const path=require('path')`
- path.join() 拼接路径

```
const path=require('path')
// 注意: ../ 会抵消前面的路径
const pathStr=path.join('/a','/b/c','../','./d','e')
console.log(pathStr);//  \a\b\d\e
```

以后涉及路径拼接 都要使用path.join()方法

- path.basename() 获取路径中的文件名

```
const path=require('path')
// 定义文件存放路径
const fpath='/a/b/c/index.html'
// 获取完整文件名(包含后缀名)
const fullName=path.basename(fpath)

// 获取完整文件名(不包含后缀名)
const fullName1=path.basename(fpath,'.html')
```

- path.extname()获取文件后缀名

```
const path=require('path')
const fpath='/a/b/c/index.html'
const fext=path.extname(fpath)
console.log(fext);//输出: .html
```

node.js

http模块

http 模块是 Node.js 官方提供的、用来创建 web 服务器的模块。通过 http 模块提供的 http.createServer() 方法，就能方便的把一台普通的电脑，变成一台 Web 服务器，从而对外提供 Web 资源服务。

创建web服务器

```
// 1.导入http模块
const http=require('http')

// 2.创建web服务器实例
const server=http.createServer();

// 3.为服务器实例绑定request事件 监听客户端的请求
server.on('request',function(res,req){
    console.log('someone visit this server');
})

// 4.启动服务器
server.listen('8080',function(){
    console.log('server running at http://127.0.0.1:8080');
});
```

req请求对象
只要服务器接收到了客户端的请求,就会调用server.on()为服务器绑定的request事件处理函数
如果想在事件处理函数中 访问与客户端相关的数据或属性 可以使用:

```
const http=require('http')
const server=http.createServer();
server.on('request',(req)=>{
    // req.url是客户端请求的url地址 从端口号后面开始
    const url=req.url;
    // req.method是客户端请求的method类型
    const method=req.method;
    // ${url}与${method} 相当于把变量加在字符串中 就不用 用'++' 来拼接字符串了
    // 但是注意:把字符串包裹的是`` 不是引号
    const str= `Your request url is ${url},and request method is ${method}`
    console.log(str);
})
```

补充

res响应对象 如果想访问与服务器相关的数据或属性 可以使用:

```
server.on('request',(req,res)=>{
    const url=req.url;
    const str= `Your request url is ${url},and request method is ${method}`
    //调用res.end()方法 向客户端响应一些内容
    res.end(str) //浏览器页面输出上面的字符串内容
})
```

解决中文乱码问题:res.setHeader()方法

```
server.on('request',(req,res)=>{
    const url=req.url;
    const method=req.method;
    const str= `您请求的url地址是${url},请求的method类型是 ${method}`
    // 调用 res.setHeader()方法 设置Content-Type响应头 解决中文乱码问题
    res.setHeader('Content-Type','text/html;charset=utf-8')
    //调用res.end()方法 向客户端响应一些内容
    res.end(str)
})
```

根据不同的url响应不同的html内容

```
server.on('request',(req,res)=>{
    // 1.获取请求的url地址
    const url=req.url;
    // 2.设置默认的响应内容为404 Not Found
    let content='<h1>404 Not Found</h1>'
    // 3.判断用户请求的是否为/或者/index.html首页
    // 4.判断用户请求的是否为/about.html关于页面
    if(url==='/'||url==='/index.html'){
        content='<h1>首页</h1>'
    }else if(url==='/about.html'){
        content='<h1>关于页面</h1>'
    }
    // 5.设置Content-Type 响应头 防止中文乱码
    res.setHeader('Content-Type','text/html;charset=utf-8')
    // 6.使用res.end()把内容响应给客户端
    res.end(content)
})
```

运行js文件

```
E:\前端\node.js>node 1.js
hello node.js
```

终端中的快捷键

① 使用 ↑ 键，可以快速定位到上一次执行的命令
② 使用 tab 键，能够快速补全路径
③ 使用 esc 键，能够快速清空当前已输入的命令
④ 输入 cls 命令，可以清空终端

node.js

模块化

编程领域中的模块化，就是**遵守固定的规则**，把一个大文件拆成独立并互相依赖的多个小模块。

模块化拆分的好处:提高了代码的复用性\可维护性,可以实现按需加载

node.js模块化

- 内置模块（内置模块是由 Node.js 官方提供的，例如 fs、path、http 等）
- 自定义模块（用户创建的每个 .js 文件，都是自定义模块）
- 第三方模块（由第三方开发出来的模块，并非官方提供的内置模块，也不是用户创建的自定义模块，使用前需要先下载）

使用require()加载模块

```
// 加载内置模块
const fs=require('fs')
// 加载自定义模块
const m1=require('./m1')
console.log(m1);
// 加载第三方模块
const moment=require('moment')
```

注意:使用require()方法加载其他模块时,会执行被加载模块中的代码,而且可以省略.js后缀名

模块作用域和module对象

和函数作用域类似，在自定义模块中定义的变量、方法等成员，只能在当前模块内被访问，这种模块级别的访问限制，叫做**模块作用域**。

在模块内定义的成员 无法被外部文件访问

在每个.js自定义模块中都有一个module对象,它里面存储了和当前模块有关的信息

向外共享模块(module)作用域中的成员

在自定义模块中，可以使用 module.exports 对象，将模块内的成员共享出去，供外界使用。

外界用 require() 方法导入自定义模块时，得到的就是 module.exports 所指向的对象。

module.exports对象

```
// 默认情况下 module.exports={}
```

```
外界可以访问的:(挂载时 module可以省略)
// 向module.exports对象挂载username属性
module.exports.username='zs'
// 向module.exports对象挂载hello方法
module.exports.hello=function(){
    console.log('hello!');
}
```

注意:module.exports对象 永远以最新的数据为准
以module.exports指向的对象为准
为了防止混乱,module.exports和exports不要在同一个模块中使用

CommonJS模块化规范

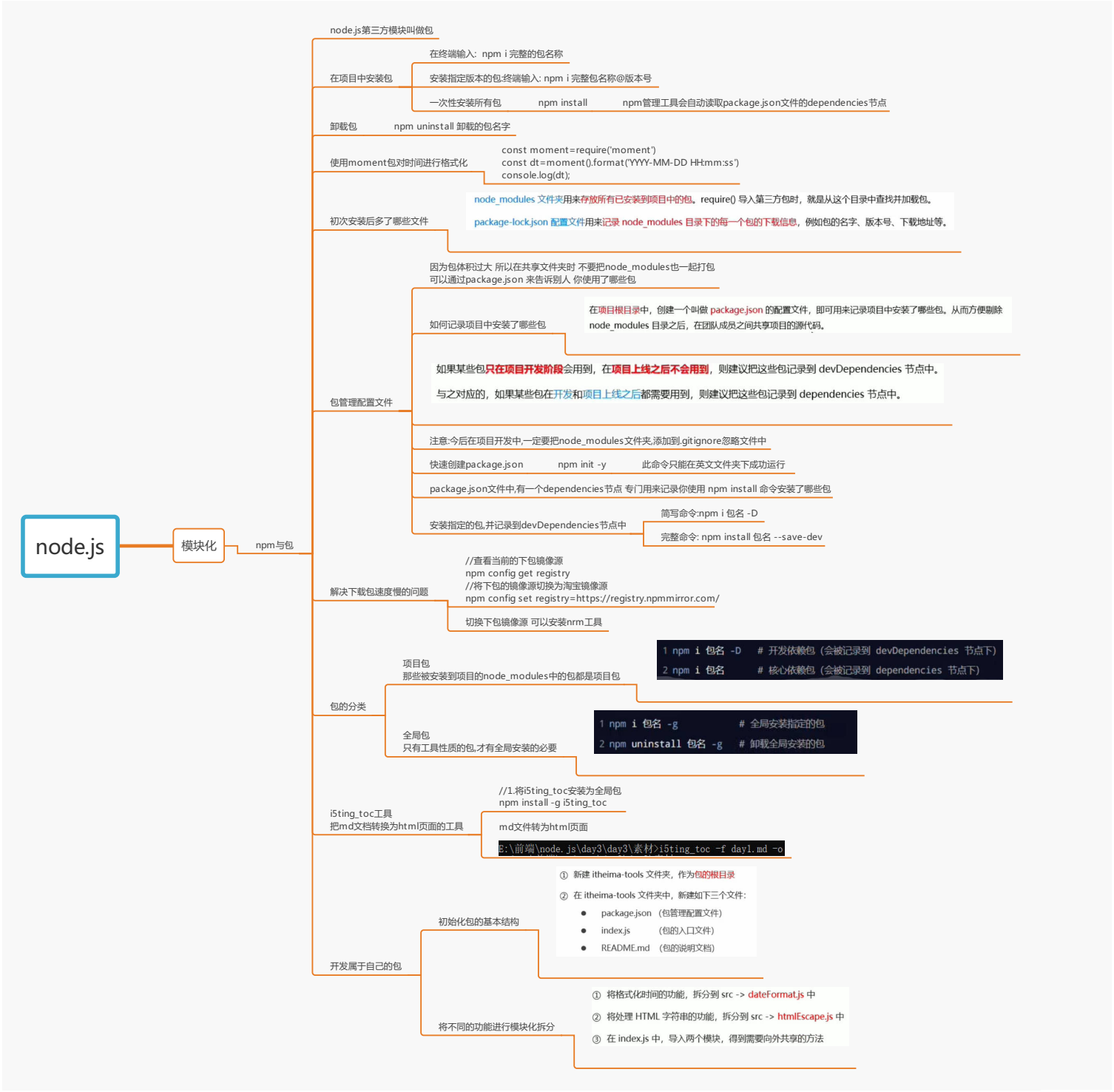
Node.js 遵循了 CommonJS 模块化规范，CommonJS 规定了模块的特性和各模块之间如何相互依赖。

CommonJS 规定:

① 每个模块内部，module 变量代表当前模块。

② module 变量是一个对象，它的 exports 属性（即 module.exports）是对外的接口。

③ 加载某个模块，其实是加载该模块的 module.exports 属性。require() 方法用于加载模块。


node.js
模块化
npm与包
node.js第三方模块叫做包
在项目中安装包
在终端输入：npm i 完整的包名称
安装指定版本的包,终端输入: npm i 完整包名称@版本号
一次性安装所有包 npm install npm管理工具会自动读取package.json文件的dependencies节点
卸载包 npm uninstall 卸载的包名字
使用moment包对时间进行格式化
const moment=require('moment')
const dt=moment().format('YYYY-MM-DD HH:mm:ss')
console.log(dt);
初次安装后多了哪些文件
node_modules 文件夹用来存放所有已安装到项目中的包。require() 导入第三方包时，就是从这个目录中查找并加载包。
package-lock.json 配置文件用来记录 node_modules 目录下的每一个包的下载信息，例如包的名字、版本号、下载地址等。
包管理配置文件
因为包体积过大 所以在共享文件夹时 不要把node_modules也一起打包
可以通过package.json 来告诉别人 你使用了哪些包
如何记录项目中安装了哪些包
在项目根目录中，创建一个叫做 package.json 的配置文件，即可用来记录项目中安装了哪些包。从而方便剔除 node_modules 目录之后，在团队成员之间共享项目的源代码。
如果某些包只在项目开发阶段会用到，在项目上线之后不会用到，则建议把这些包记录到 devDependencies 节点中。
与之对应的，如果某些包在开发和项目上线之后都需要用到，则建议把这些包记录到 dependencies 节点中。
注意:今后在项目开发中,一定要把node_modules文件夹,添加到.gitignore忽略文件中
快速创建package.json npm init -y 此命令只能在英文文件夹下成功运行
package.json文件中,有一个dependencies节点 专门用来记录你使用 npm install 命令安装了哪些包
安装指定的包,并记录到devDependencies节点中
简写命令:npm i 包名 -D
完整命令: npm install 包名 --save-dev
解决下载包速度慢的问题
//查看当前的下包镜像源
npm config get registry
//将下包的镜像源切换为淘宝镜像源
npm config set registry=https://registry.npmmirror.com/
切换下包镜像源 可以安装nrm工具
包的分类
项目包
那些被安装到项目的node_modules中的包都是项目包
1 npm i 包名 -D # 开发依赖包（会被记录到 devDependencies 节点下）
2 npm i 包名 # 核心依赖包（会被记录到 dependencies 节点下）
全局包
只有工具性质的包,才有全局安装的必要
1 npm i 包名 -g # 全局安装指定的包
2 npm uninstall 包名 -g # 卸载全局安装的包
i5ting_toc工具
把md文档转换为html页面的工具
//1.将i5ting_toc安装为全局包
npm install -g i5ting_toc
md文件转为html页面
E:\前端\node.js\day3\day3\素材>i5ting_toc -f day1.md -o
开发属于自己的包
初始化包的基本结构
① 新建 itheima-tools 文件夹，作为包的根目录
② 在 itheima-tools 文件夹中，新建如下三个文件：
package.json （包管理配置文件）
index.js （包的入口文件）
README.md （包的说明文档）
将不同的功能进行模块化拆分
① 将格式化时间的功能，拆分到 src -> dateFormat.js 中
② 将处理 HTML 字符串的功能，拆分到 src -> htmlEscape.js 中
③ 在 index.js 中，导入两个模块，得到需要向外共享的方法

node.js

模块化

模块的加载机制

模块在第一次加载后会被缓存。 这也意味着多次调用 require() 不会导致模块的代码被执行多次。

注意：不论是内置模块、用户自定义模块、还是第三方模块，它们都会优先从缓存中加载，从而提高模块的加载效率。

内置模块的加载优先级最高

使用 require() 加载自定义模块时，必须指定以 ./ 或 ../ 开头的路径标识符。在加载自定义模块时，如果没有指定 ./ 或 ../ 这样的路径标识符，则 node 会把它当作内置模块或第三方模块进行加载。

如果传递给 require() 的模块标识符不是一个内置模块，也没有以 ‘./’ 或 ‘../’ 开头，则 Node.js 会从当前模块的父目录开始，尝试从 /node_modules 文件夹中加载第三方模块。

如果没有找到对应的第三方模块，则移动到再上一层父目录中，进行加载，直到文件系统的根目录。

Express(重点)

初识Express

Express是基于node.js平台,快速、开放、极简的Web开发框架

专门用来创建Web服务器

Express本质就是npm上的第三方包,提供了快速创建Web服务器的便捷方法

基本使用

安装　　在项目所处的目录中,在终端输入:　　npm i express@4.17.2

创建基本的web服务器

```
// 导入express
const express=require('express')
 // 创建web服务器
const app=express()
// 启动web服务器
app.listen('80',()=>{
    console.log('express server running at http://127.0.0.1');
})
```

监听GET请求和监听POST请求

```
const express=require('express')
const app=express()
app.get('/user',(req,res)=>{
// 使用express提供的res.send()方法,向客户端响应一个JSON对象
    res.send({username:'zs',age:'20',gender:'男'})
})
app.post('/user',(req,res)=>{
// 使用express提供的res.send()方法,向客户端响应一个 文本字符串
    res.send('请求成功')
})
app.listen('80',()=>{
    console.log('express server running at http://127.0.0.1');
})
```

参数

```
// 参数1: 客户端请求的 URL 地址
// 参数2: 请求对应的处理函数
//        req: 请求对象（包含了与请求相关的属性与方法）
//        res: 响应对象（包含了与响应相关的属性与方法）
```

获取url中携带的查询参数

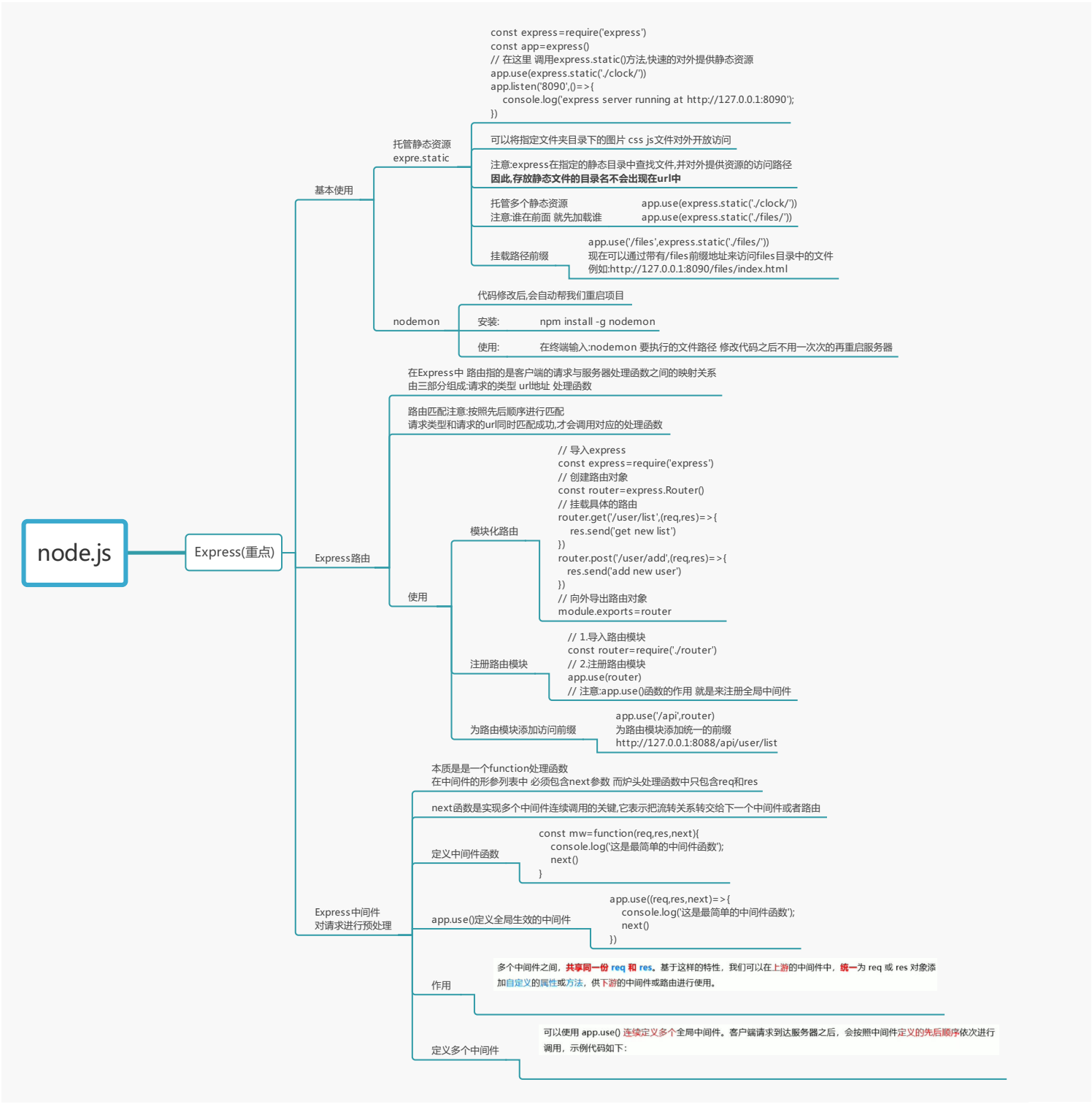
```
app.get('/',(req,res)=>{
  // 通过req.query可以获取到客户端发送过来的 查询参数
  // 客户端吧使用/?username=zs&age=20这种查询字符串形式,发送到服务器的参数
  // 注意:默认情况下 req.query是一个空对象
  console.log(req.query);
  res.send(req.query)
})
```

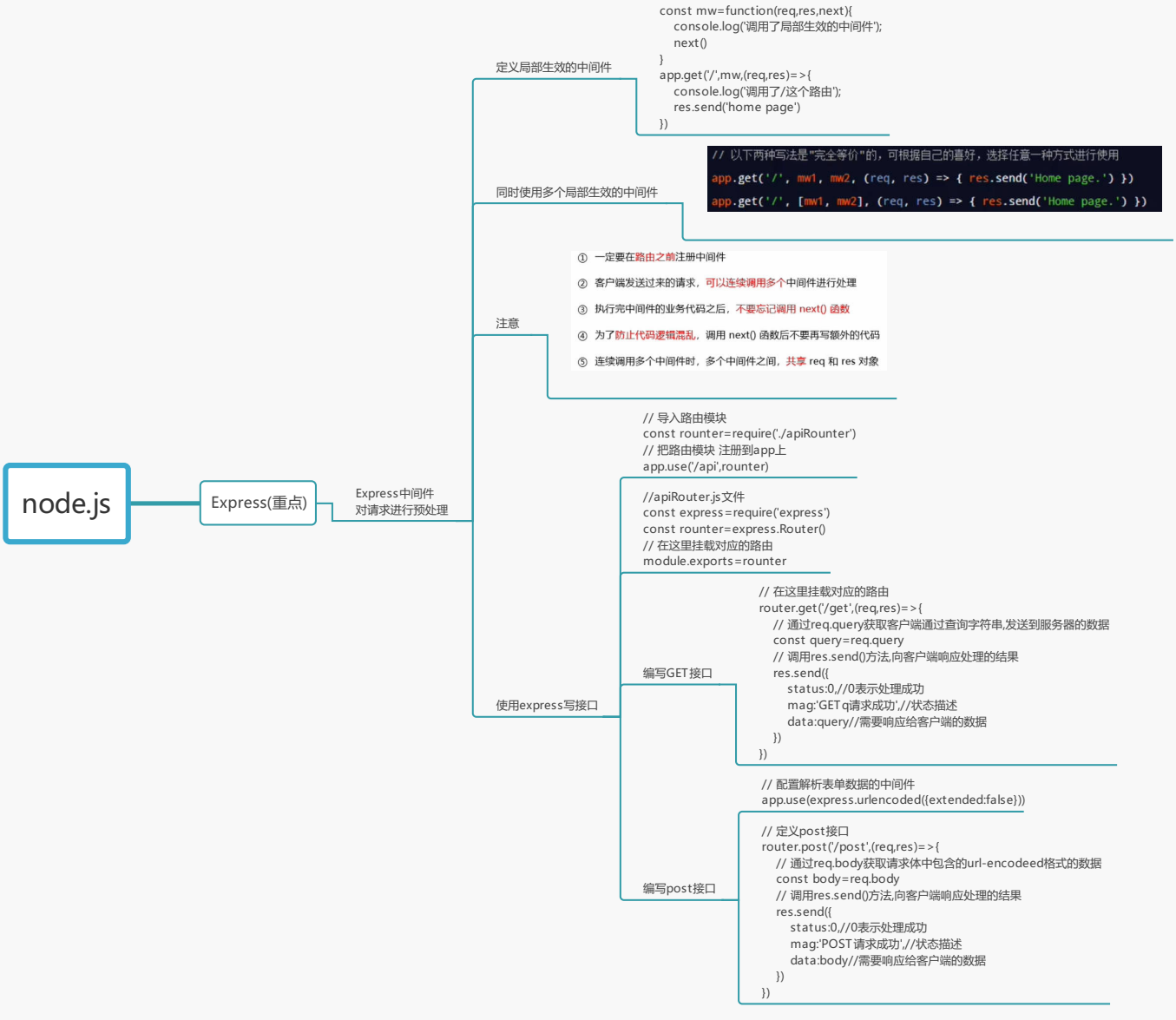
获取url中的动态参数

```
app.get('/user/:id',(req,res)=>{
  // req.params默认是一个空对象
  // 里面存放着通过 : 动态匹配到的参数值
  console.log(req.params);
  res.send(req.params)
})
```

http://127.0.0.1:8080/user/1
返回的是一个对象: {"id": "1"}

匹配两个参数:app.get('/user/:id/:username',


node.js
Express(重点)
基本使用
托管静态资源
expre.static
const express=require('express')
const app=express()
// 在这里 调用express.static()方法,快速的对外提供静态资源
app.use(express.static('./clock/'))
app.listen('8090',()=>{
console.log('express server running at http://127.0.0.1:8090');
})
可以将指定文件夹目录下的图片 css js文件对外开放访问
注意:express在指定的静态目录中查找文件,并对外提供资源的访问路径
因此,存放静态文件的目录名不会出现在url中
托管多个静态资源
注意:谁在前面 就先加载谁
app.use(express.static('./clock/'))
app.use(express.static('./files/'))
挂载路径前缀
app.use('/files',express.static('./files/'))
现在可以通过带有/files前缀地址来访问files目录中的文件
例如:http://127.0.0.1:8090/files/index.html
nodemon
代码修改后,会自动帮我们重启项目
安装:
npm install -g nodemon
使用:
在终端输入:nodemon 要执行的文件路径 修改代码之后不用一次次的再重启服务器
Express路由
在Express中 路由指的是客户端的请求与服务器处理函数之间的映射关系
由三部分组成:请求的类型 url地址 处理函数
路由匹配注意:按照先后顺序进行匹配
请求类型和请求的url同时匹配成功,才会调用对应的处理函数
使用
模块化路由
// 导入express
const express=require('express')
// 创建路由对象
const router=express.Router()
// 挂载具体的路由
router.get('/user/list',(req,res)=>{
res.send('get new list')
})
router.post('/user/add',(req,res)=>{
res.send('add new user')
})
// 向外导出路由对象
module.exports=router
注册路由模块
// 1.导入路由模块
const router=require('./router')
// 2.注册路由模块
app.use(router)
// 注意:app.use()函数的作用 就是来注册全局中间件
为路由模块添加访问前缀
app.use('/api',router)
为路由模块添加统一的前缀
http://127.0.0.1:8088/api/user/list
Express中间件
对请求进行预处理
本质是是一个function处理函数
在中间件的形参列表中 必须包含next参数 而炉头处理函数中只包含req和res
next函数是实现多个中间件连续调用的关键,它表示把流转关系转交给下一个中间件或者路由
定义中间件函数
const mw=function(req,res,next){
console.log('这是最简单的中间件函数');
next()
}
app.use()定义全局生效的中间件
app.use((req,res,next)=>{
console.log('这是最简单的中间件函数');
next()
})
作用
多个中间件之间，共享同一份 req 和 res。基于这样的特性，我们可以在上游的中间件中，统一为 req 或 res 对象添加自定义的属性或方法，供下游的中间件或路由进行使用。
定义多个中间件
可以使用 app.use() 连续定义多个全局中间件。客户端请求到达服务器之后，会按照中间件定义的先后顺序依次进行调用，示例代码如下：


node.js
Express(重点)
Express中间件
对请求进行预处理
定义局部生效的中间件
const mw=function(req,res,next){
console.log('调用了局部生效的中间件');
next()
}
app.get('/',mw,(req,res)=>{
console.log('调用了/这个路由');
res.send('home page')
})
同时使用多个局部生效的中间件
// 以下两种写法是"完全等价"的，可根据自己的喜好，选择任意一种方式进行使用
app.get('/', mw1, mw2, (req, res) => { res.send('Home page.') })
app.get('/', [mw1, mw2], (req, res) => { res.send('Home page.') })
注意
① 一定要在路由之前注册中间件
② 客户端发送过来的请求，可以连续调用多个中间件进行处理
③ 执行完中间件的业务代码之后，不要忘记调用 next() 函数
④ 为了防止代码逻辑混乱，调用 next() 函数后不要再写额外的代码
⑤ 连续调用多个中间件时，多个中间件之间，共享 req 和 res 对象
使用express写接口
// 导入路由模块
const rounter=require('./apiRounter')
// 把路由模块 注册到app上
app.use('/api',rounter)
//apiRouter.js文件
const express=require('express')
const rounter=express.Router()
// 在这里挂载对应的路由
module.exports=rounter
编写GET接口
// 在这里挂载对应的路由
router.get('/get',(req,res)=>{
// 通过req.query获取客户端通过查询字符串,发送到服务器的数据
const query=req.query
// 调用res.send()方法,向客户端响应处理的结果
res.send({
status:0,//0表示处理成功
mag:'GET q请求成功',//状态描述
data:query//需要响应给客户端的数据
})
})
编写post接口
// 配置解析表单数据的中间件
app.use(express.urlencoded({extended:false}))
// 定义post接口
router.post('/post',(req,res)=>{
// 通过req.body获取请求体中包含的url-encodeed格式的数据
const body=req.body
// 调用res.send()方法,向客户端响应处理的结果
res.send({
status:0,//0表示处理成功
mag:'POST 请求成功',//状态描述
data:body//需要响应给客户端的数据
})
})

node.js

1. 身份认证

1.1. 不同开发模式下的身份认证

对于服务端渲染和前后端分离这两种开发模式来说，分别有着不同的身份认证方案：

① 服务端渲染推荐使用 **Session 认证机制**

② 前后端分离推荐使用 **JWT 认证机制**

2. Session认证机制 (服务器端渲染)

2.1. HTTP协议的无状态性

HTTP 协议的无状态性，指的是客户端**的每次 HTTP 请求都是独立的**，连续多个请求之间没有直接的关系，**服务器不会主动保留每次 HTTP 请求的状态。**

2.2. cookie
相当于现实生活中的会员卡身份认证方式

2.3. cookie几大特性: 自动发送 域名独立 过期时限 4kb限制

Cookie 是**存储在用户浏览器中的一段不超过 4 KB 的字符串。**它由一个名称（Name）、一个值（Value）和其它几个用于控制 Cookie 有效期、安全性、使用范围的可选属性组成。

不同域名下的 Cookie 各自独立，每当客户端发起请求时，会**自动**把**当前域名下**所有**未过期的 Cookie** 一同发送到服务器。

2.4. cookie在身份认证中的作用

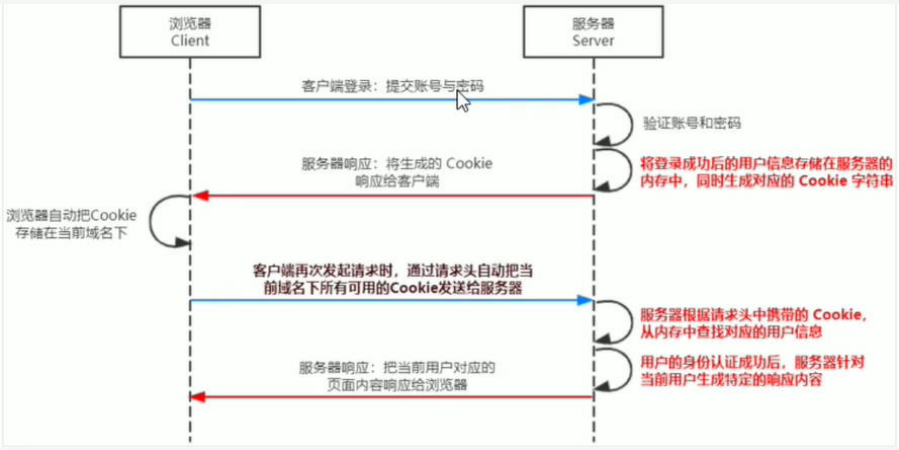
客户端第一次请求服务器的时候，服务器**通过响应头的形式**，向客户端发送一个身份认证的 Cookie，客户端会自动将 Cookie 保存在浏览器中。

随后，当客户端浏览器每次请求服务器的时候，浏览器会**自动**将身份认证相关的 Cookie，**通过请求头的形式**发送给服务器，服务器即可验明客户端的身份。

2.5. **cookie不具有安全性 不要存储用户的身份信息以及密码**

由于 Cookie 是存储在浏览器中的，而且**浏览器也提供了读写 Cookie 的 API**，因此 **Cookie 很容易被伪造**，不具有安全性。因此不建议服务器将重要的隐私数据，通过 Cookie 的形式发送给浏览器。

2.6. **session工作原理**



node.js

1. session中间件的使用

1.1. 安装:npm i express-session

1.2.
```
// 配置 Session 中间件
const session=require('express-session')
app.use(session({
  secret:'itheima',
//固定写法
  resave:false,
  saveUninitialized:true,
}))
```

1.3. 向session中存数据 —— 1.3.1.
```
//注意:只有成功配置了express-session这个中间件之后,
才能通过req点出来session这个属性
  req.session.user=body//用户的信息
  req.session.islogin=true//用户的登录状态
以上名字自己定义
```

1.4. 向session中取数据 —— 1.4.1.
```
// 获取用户姓名的接口
app.get('/api/username', (req, res) => {
  // TODO_03：从 Session 中获取用户的名称，响应给客户端
  if(!req.session.islogin){
    return res.send({status:0,msg:'fail'})
  }else{
    return res.send({status:1,msg:'success',username:req.session.user.username})
  }
})
```

1.5. 清空session 用户调用退出登录的接口 —— 1.5.1. req.session.destroy() 清空当前客户端对应的session信息

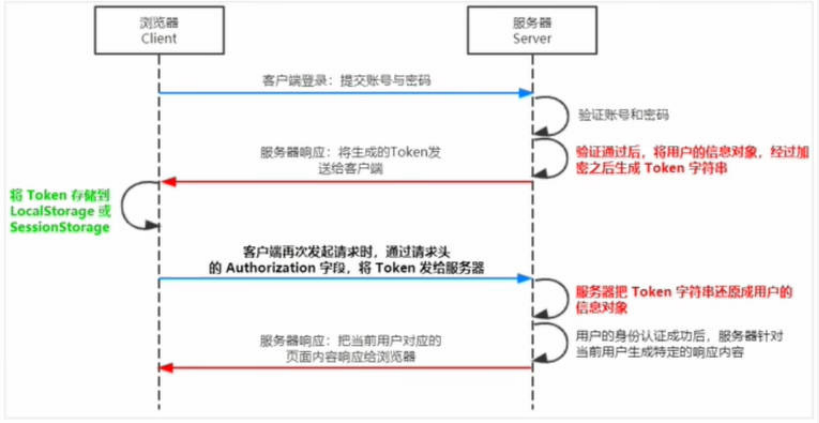
1.6. session认证局限性 —— 1.6.1.

Session 认证机制需要配合 Cookie 才能实现。由于 Cookie 默认不支持跨域访问，所以，当涉及到前端跨域请求后端接口的时候，需要做很多额外的配置，才能实现跨域 Session 认证。

2. JWT认证机制 跨域认证解决方案

2.1. JWT工作原理

用户的信息通过Token字符串的形式,保存在客户端浏览器中.服务器通过还原Token字符串的形式来认证用户的身份



2.2. 由三部分组成:Header(头部) Payload(有效荷载) Signature(签名)

2.3. 使用方式 格式如下:**Authorization: Bearer <token>**

客户端收到服务器返回的 JWT 之后，通常会将它储存在 localStorage 或 sessionStorage 中。

此后，客户端每次与服务器通信，都要带上这个 JWT 的字符串，从而进行身份认证。推荐的做法是**把 JWT 放在 HTTP 请求头的 Authorization 字段中**，格式如下:

node.js

1. JWT认证机制
跨域认证解决方案

1.1. 安装JWT相关包 ——
npm i jsonwebtoken express-jwt
jsonwebtoken用于生成JWT字符串
1.1.1. **express-jwt**用于将JWT字符串解析还原成JSON对象
如果后面提示express-jwt is not function
express-jwt 安装5.3.3版本

1.2. 导入相关包 ——
```
//安装并导入 JWT 相关的两个包，分别是 jsonwebtoken 和 express-jwt
1.2.1. const jwt=require('jsonwebtoken')
const expressjwt=require('express-jwt')
```

1.3. 定义secret密钥-保证JWT字符串的安全性

① 当生成 JWT 字符串的时候，需要使用 secret 密钥对用户的信息进行加密，最终得到加密好的 JWT 字符串

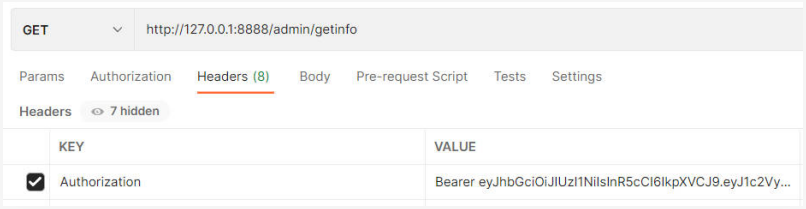
② 当把 JWT 字符串解析还原成 JSON 对象的时候，需要使用 secret 密钥进行解密

1.4.
```
在登陆成功后生成JWT字符串
// 在登录成功之后，调用 jwt.sign() 方法生成 JWT 字符串。并通过 token 属性发送给客户端
 // 参数1:用户的信息对象
 // 参数2:加密的密钥
 // 参数3:配置对象,可以配置当前token的有效期
 const tokenStr=jwt.sign({username:userinfo.username},secretKey,{expiresIn:'30s'})
//将token响应给客户端
res.send({
   status: 200,
   message: '登录成功！',
   token: tokenStr// 要发送给客户端的 token 字符串
 })
```

1.5.
```
将JWT字符串还原为JSON对象
注意:只要配置成功了express-jwt这个中间件,就可以把解析出来的用户信息,挂载到req.user属性上
app.use(expressjwt({secret:secretKey}).unless({path:[/^\/api\//]}))
记住不要把密码加密到token字符串中
```

```
// 使用 app.use() 来注册中间件
// expressJWT({ secret: secretKey }) 就是用来解析 Token 的中间件
// .unless({ path: [/^\/api\//] }) 用来指定哪些接口不需要访问权限
```

1.6. JWT字符串解析为JSON对象时注意



1.7. 捕获解析JWT失败后产生的错误

当使用 express-jwt 解析 Token 字符串时，如果客户端发送过来的 Token 字符串**过期**或**不合法**，会产生一个**解析失败**的错误，影响项目的正常运行。我们可以通过 **Express 的错误中间件**，捕获这个错误并进行相关的处理，示例代码如下：

1.8.
```
app.use((err,req,res,next)=>{
  if(err.name==='UnauthorizedError'){
    return res.send({status:401, message:'无效的token'
    })
  }else{
    res.send({  status:500,message:'未知的错误' })
  }
})
```

MySQL

1. 创建表

1.1. DataType数据类型
- 1.1.1. int 整数
- 1.1.2. varchar(len)字符串
- 1.1.3. tinyint(1)布尔值

1.2.

字段的特殊标识：

① PK（Primary Key）主键、唯一标识

② NN（Not Null）值不允许为空

③ UQ（Unique）值唯一

④ AI（Auto Increment）值自动增长

2. SQL

2.1.

SQL（英文全称：Structured Query Language）是结构化查询语言，专门用来访问和处理数据库的编程语言。能够让我们以编程的形式，操作数据库里面的数据。

三个关键点:
1.SQL是一门数据库编程语言
2.2. 2.使用SQL语言编写出来的代码,叫做SQL语句
3.SQL语言只能在关系型数据库中使用(例如MySQL,Oracle,SQL server).非关系型数据库(MongoDB)不支持SQL语言
SQL语句的关键字对大小写不敏感

3. SQL语句

3.1. select-查询数据

3.1.1.

```
-- 这是注释
-- 从 FROM 指定的【表中】，查询出【所有的】数据。 * 表示【所有列】
SELECT * FROM 表名称

-- 从 FROM 指定的【表中】，查询出指定 列名称（字段） 的数据。
SELECT 列名称 FROM 表名称
```

3.1.2.
```
-- 通过*把users 表中所有的数据查询出来
select * from users
-- 从user表中把username和password对应的数据查询出来
select username,password from users
```

3.2. insert into-向数据表中插入新的数据行

3.2.1. 子主题

```
-- 语法解读：向指定的表中，插入如下几列数据，列的值通过 values 一一指定
-- 注意：  列和值要一一对应，多个列和多个值之间，使用英文的逗号分隔
INSERT INTO table_name (列1, 列2,...) VALUES (值1, 值2,....)
```

3.2.2.
```
-- 向users表中,插入新数据,username的值为sss, password 的值为098123
insert into users(username,password) values('sss','098123')
```

3.3. update-修改表中的数据

3.3.1.
```
-- 将id 为 4 的用户名密码 更新为888888
update users set password='888888' where id=4
```

3.3.2.
```
-- 把users 表中id为2的用户密码和用户状态,分别更新为admin23和1
update users set password='admin123',status=1 where id=2
```

3.4. delete-删除表中的行

3.4.1.
```
-- 从user表中 删除id 为5的用户
delete from users where id=5
```

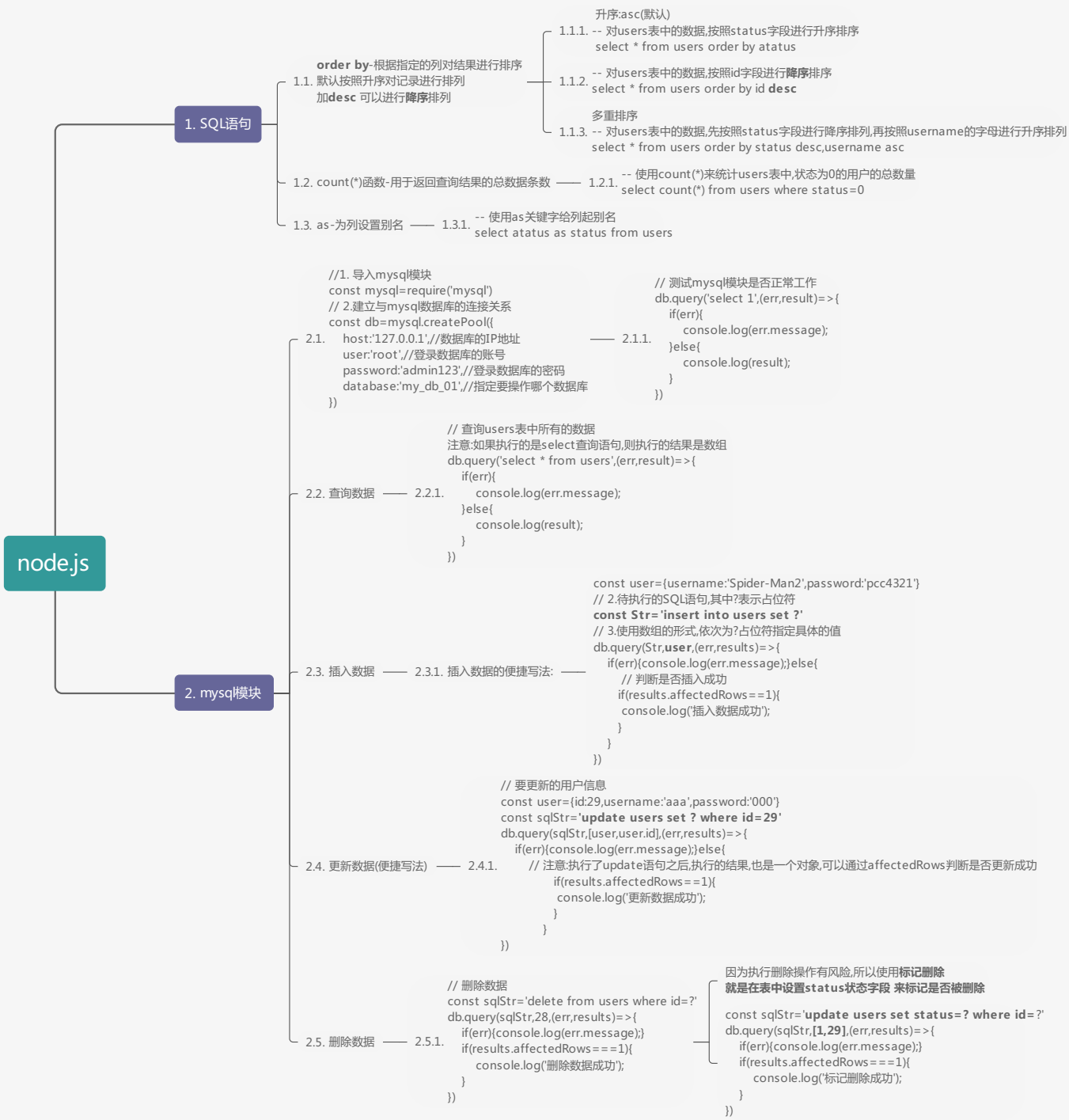
3.5. where子句用于限定选择的标准

3.5.1.
```
示例:
-- select * from users where status=1
-- select * from users where id>2
select * from users where username!='ls'
```

3.6. and运算符和or运算符

3.6.1.
```
-- 使用and来显示所有status为0 并且id小于3 的用户
select * from users where status=0 and id<3
```

3.6.2.
```
-- 使用or来显示所有状态为1或username为zs的用户
select * from users where status=1 or username='zs'
```


node.js
1. SQL语句
1.1. 默认按照升序对记录进行排列
order by-根据指定的列对结果进行排序
加desc 可以进行降序排列
升序:asc(默认)
1.1.1. -- 对users表中的数据,按照status字段进行升序排序
select * from users order by atatus
1.1.2. -- 对users表中的数据,按照id字段进行降序排序
select * from users order by id desc
多重排序
1.1.3. -- 对users表中的数据,先按照status字段进行降序排列,再按照username的字母进行升序排列
select * from users order by status desc,username asc
1.2. count(*)函数-用于返回查询结果的总数据条数
1.2.1. -- 使用count(*)来统计users表中,状态为0的用户的总数量
select count(*) from users where status=0
1.3. as-为列设置别名
1.3.1. -- 使用as关键字给列起别名
select atatus as status from users
2. mysql模块
2.1.
//1. 导入mysql模块
const mysql=require('mysql')
// 2.建立与mysql数据库的连接关系
const db=mysql.createPool({
host:'127.0.0.1',//数据库的IP地址
user:'root',//登录数据库的账号
password:'admin123',//登录数据库的密码
database:'my_db_01',//指定要操作哪个数据库
})
2.1.1.
// 测试mysql模块是否正常工作
db.query('select 1',(err,result)=>{
if(err){
console.log(err.message);
}else{
console.log(result);
}
})
2.2. 查询数据
2.2.1.
// 查询users表中所有的数据
注意:如果执行的是select查询语句,则执行的结果是数组
db.query('select * from users',(err,result)=>{
if(err){
console.log(err.message);
}else{
console.log(result);
}
})
2.3. 插入数据
2.3.1. 插入数据的便捷写法:
const user={username:'Spider-Man2',password:'pcc4321'}
// 2.待执行的SQL语句,其中?表示占位符
const Str='insert into users set ?'
// 3.使用数组的形式,依次为?占位符指定具体的值
db.query(Str,user,(err,results)=>{
if(err){console.log(err.message);}else{
// 判断是否插入成功
if(results.affectedRows==1){
console.log('插入数据成功');
}
}
})
2.4. 更新数据(便捷写法)
2.4.1.
// 要更新的用户信息
const user={id:29,username:'aaa',password:'000'}
const sqlStr='update users set ? where id=29'
db.query(sqlStr,[user,user.id],(err,results)=>{
if(err){console.log(err.message);}else{
// 注意:执行了update语句之后,执行的结果,也是一个对象,可以通过affectedRows判断是否更新成功
if(results.affectedRows==1){
console.log('更新数据成功');
}
}
})
2.5. 删除数据
2.5.1.
// 删除数据
const sqlStr='delete from users where id=?'
db.query(sqlStr,28,(err,results)=>{
if(err){console.log(err.message);}
if(results.affectedRows===1){
console.log('删除数据成功');
}
})
因为执行删除操作有风险,所以使用标记删除
就是在表中设置status状态字段 来标记是否被删除
const sqlStr='update users set status=? where id=?'
db.query(sqlStr,[1,29],(err,results)=>{
if(err){console.log(err.message);}
if(results.affectedRows===1){
console.log('标记删除成功');
}
})