

jQuery

- jQuery的安装与导入
- jQuery的语法结构
- jQuery的选择器
- jQuery的方法
 - DOM方法
 - CSS样式方法
 - 事件方法
 - jQuery的on的事件绑定
 - jQuery同一个元素的多个事件绑定
 - jQuery中on的事件委托绑定
 - jQuery中的one单次事件绑定
 - jQuery中的off移除事件
 - 属性方法
 - 尺寸方法
 - 动画方法
 - jQuery动画的回调函数
 - Ajax方法
 - 其它方法
- jQuery的链式语法
- DOM对象与jQuery对象的切换
 - jQuery对象转普通的DOM对象
 - 普通的DOM对象如何转换成jQuery对象
- jQuery的文档就绪函数

jQuery事件对象及补充点

- jQuery的事件对象
- jQuery事件委托详解
- jQuery事件绑定的方式
- jQuery的扩展方法

jQuery的表单验证

- 安装
- 配置规则
- 验证规则
- 验证方式
 - 配置式的验证
 - 侵入式的验证规则
- 自定义验证规则
- 自定义消息提示的样式
- 自定义消息提示的位置

作业与练习

ECMAScript 6

- let变量定义
 - 暂时性死区
- const常量
 - const锁栈与锁堆原理
- let与const的闭包特性
- 解构
 - 解构取值

- 解构赋值
- 展开运算符
- 指数运算符
- 关于BigInt数据类型
- BigInt数据类型转换
- 字符串的扩展
- 模板字符串
- 字符串的扩展方法
- 数组的扩展
- 数组的扩展方法
- for...of遍历

Set单值集合

- Set的创建
- Set相关的方法
- Set的遍历

Map键值对集合

- Map的创建
- Map相关的方法
- Map的遍历

ECMAScript 6-2

- WeakSet与WeakMap
- Symbol数据类型
- Symbol的应用点
- Symbol做属性名

生成器函数

- 定义生成器函数
- 生成器函数运行
- 生成器函数调用另一个生成器函数

迭代器

- 迭代器的状态
- 迭代器遍历
- 迭代器接口

函数的扩展

- 无构造函数的函数
- 箭头函数
- 无参数的箭头函数
- 有一个参数的箭头函数
- 有多个参数的箭头函数
- 箭头函数的返回值
- 箭头函数的注意事项

ES6函数的参数

- 函数的默认参数【可选参数】
- 函数的剩余参数

函数的补充call/apply/bind

class关键字

- 认识关键字 `class`
- 认识一下 `constructor`
- class里面定义方法
- class里面的get/set访问器
- class里面的static关键字

extends关键字

无参数的继承

有参数的继承

静态继承

方法的重写 **override**

extends继承的特点

ECMAScript6-3

面向对象的扩展

不可扩展对象

密封对象

冻结对象

Object.is()

Reflect反射

Proxy代理

代理的概念

创建代理

代理的作用

代理的应用点

通过代理实现属性私有化

代理的操作列表

可取消的代理

代理小练习

ES6异步编程及模块化

异步编程

回调函数处理异步

Promise异步处理

Promise.all()

Promise.any()

Promise.race()

async/await关键字

模块化

模块化重点扩展【引用】

nodejs介绍与安装

关于node.js

为什么前端需要学习node.js

nodejs的安装

node.js基础

node.js的运行

CommonJs模块化

CommonJS的模块导入

CommonJS的缓存

CommonJS模块导出

module.exports

exports

nodejs内置模块

path模块

fs模块

os模块

课堂练习

nodejs第三方模块

npm工具

- npm初始化
- package.json介绍
- npmjs远端仓库
- npm info查看包的信息
- npm install安装包
- save记录安装信息
- save-dev记录安装信息

为什么要记录安装包的信息

npm uninstall卸载包

npm install指定包的版本

批量装包

npm install的简化命令

npm uninstall简化命令

npm命令总结

npm国内镜像设置

使用axios+cheerio来完成抓包

模块网络请求

分析得到的结果

抓取图片

使用nodemailer发送邮件

node-xlsx的使用

安装

读取excel文件

生成excel文件

综合练习

MySql基础

数据库连接

基础DDL语句

数据表的主键

删除主键

数据表的外键

MySql数据类型简述

练习

MySql的DML语句

新增SQL语句

修改的SQL语句

删除SQL语句

查询SQL语句

查询范例

模糊查询范例

多表联查【过渡方案】

内联查询与子查询

分页查询

结果集并联

行转列

DML正则查询

多表联查详细讲解

主外键与视图详细讲解

关于主键

[关于外键](#)

[构建主外键的前提条件](#)

[数据库视图](#)

[使用DDL语句来创建视图](#)

[使用navicat来创建视图](#)

[扩展](#)

[node.js连接mysql数据库](#)

[安装第三方模块mysql](#)

[连接mysql数据库](#)

[操作mysql数据库](#)

[小练习](#)

jQuery

jQuery是一个快速、简洁的 **JavaScript** 框架，是继Prototype之后又一个优秀的JavaScript代码库（框架）于2006年1月由 **John Resig** 发布。jQuery设计的宗旨是“write Less, Do More”，即倡导写更少的代码，做更多的事情。它封装JavaScript常用的功能代码，提供一种简便的JavaScript **设计模式**，优化 **HTML** 文档操作、事件处理、动画设计和 **Ajax** 交互。

jQuery的核心特性可以总结为：具有独特的链式语法和短小清晰的多功能接口；具有高效灵活的 **CSS选择器**，并且可对 **CSS** 选择器进行扩展；拥有便捷的插件扩展机制和丰富的插件。jQuery兼容各种主流浏览器，如 **IE** 6.0+、**FF** 1.5+、**Safari** 2.0+、**Opera** 9.0+等。

jQuery的核心点和应用点

1. 优化HTML的文档操作（DOM操作）
2. 事件处理
3. 动画设计
4. Ajax交互

jQuery的安装与导入

jQuery的安装与导入是非常简单的，它就是一个单独的JS文件，通过 **script** 导入进去就可以了

第一种方式：将jQuery的文件下载到本地，然后导入到项目

```
<script src=".//js/jquery-3.6.1.js"></script>
```

第二种方式：通过在线的CDN引入

```
<script src="http://libs.baidu.com/jquery/2.0.0/jquery.min.js"></script>
```

通过上面的2种方式我们都可以来导入jQuery，一旦导入jQuery以后就可以使用

jQuery目前的版本是3.6.1，这个新版本是不兼容低版本的IE的，如果以后在工作当中要兼容低版本的IE浏览器，建议将jQuery的版本降低到1.8以下。

工作当中jQuery的版本并不是越高越好，要根据自己的实际工作需求来选择

jQuery的语法结构

jQuery的语法主要是由三个部分组成

1. 起始标志符 \$
2. 选择器
3. 方法

```
//语法格式
$(选择器).方法();

$(".btn").click();
$(".btn").html();
```

jQuery的选择器

jQuery的选择器是通过 \$ 符号来快速的选择页面的元素的，这些选择器与我们之前所学习的CSS的选择保持一致，同时jQuery又扩展了一些新的选择器

第一类：普通的CSS选择器

选择器	实例	选取
<code>*</code>	<code>\$("")</code>	所有元素
<code>#id</code>	<code>\$("#lastname")</code>	id="lastname" 的元素
<code>.class</code>	<code>\$(".intro")</code>	class="intro" 的所有元素
<code>.class,.class</code>	<code>\$(".intro,.demo")</code>	class 为 "intro" 或 "demo" 的所有元素
<code>element</code>	<code>\$(“p”)</code>	所有 <code><p></code> 元素
<code>e1,e2,e3</code>	<code> \$("h1,div,p")</code>	所有 <code><h1></code> 、 <code><div></code> 和 <code><p></code> 元素
<code>:first</code>	<code>\$("p:first")</code>	第一个 <code><p></code> 元素
<code>:last</code>	<code> \$("p:last")</code>	最后一个 <code><p></code> 元素
<code>:even</code>	<code> \$("tr:even")</code>	所有偶数 <code><tr></code> 元素，索引值从 0 开始，第一个元素是偶数 (0)，第二个元素是奇数 (1)，以此类推。
<code>:odd</code>	<code> \$("tr:odd")</code>	所有奇数 <code><tr></code> 元素，索引值从 0 开始，第一个元素是偶数 (0)，第二个元素是奇数 (1)，以此类推。
<code>:first-child</code>	<code> \$("p:first-child")</code>	属于其父元素的第一个子元素的所有 <code><p></code> 元素
<code>:first-of-type</code>	<code> \$("p:first-of-type")</code>	属于其父元素的第一个 <code><p></code> 元素的所有 <code><p></code> 元素
<code>:last-child</code>	<code> \$("p:last-child")</code>	属于其父元素的最后一个子元素的所有 <code><p></code> 元素
<code>:last-of-type</code>	<code> \$("p:last-of-type")</code>	属于其父元素的最后一个 <code><p></code> 元素的所有 <code><p></code> 元素
<code>:nth-child(n)</code>	<code> \$("p:nth-child(2)")</code>	属于其父元素的第二个子元素的所有 <code><p></code> 元素
<code>:nth-last-child(n)</code>	<code> \$("p:nth-last-child(2)")</code>	属于其父元素的第二个子元素的所有 <code><p></code> 元素，从最后一个子元素开始计数
<code>:nth-of-type(n)</code>	<code> \$("p:nth-of-type(2)")</code>	属于其父元素的第二个 <code><p></code> 元素的所有 <code><p></code> 元素
<code>:nth-last-of-type(n)</code>	<code> \$("p:nth-last-of-type(2)")</code>	属于其父元素的第二个 <code><p></code> 元素的所有 <code><p></code> 元素，从最后一个子元素开始计数
<code>:only-child</code>	<code> \$("p:only-child")</code>	属于其父元素的唯一子元素的所有 <code><p></code> 元素
<code>:only-of-type</code>	<code> \$("p:only-of-type")</code>	属于其父元素的特定类型的唯一子元素的所有 <code><p></code> 元素

第二类：jQuery 扩展的选择器

<code>parent > child</code>	<code>\$("div > p")</code>	<div> 元素的直接子元素的所有 <p> 元素
<code>parent descendant</code>	<code>\$("div p")</code>	<div> 元素的后代的所有 <p> 元素
<code>element + next</code>	<code>\$("div + p")</code>	每个 <div> 元素相邻的下一个 <p> 元素
<code>element ~ siblings</code>	<code>\$("div ~ p")</code>	<div> 元素同级的所有 <p> 元素
<code>:eq(index)</code>	<code>\$("ul li:eq(3)")</code>	列表中的第四个元素 (index 值从 0 开始)
<code>:gt(no)</code>	<code>\$("ul li:gt(3)")</code>	列举 index 大于 3 的元素
<code>:lt(no)</code>	<code>\$("ul li:lt(3)")</code>	列举 index 小于 3 的元素
<code>:not(selector)</code>	<code>\$("input:not(:empty)")</code>	所有不为空的输入元素
<code>:header</code>	<code>\$(":header")</code>	所有标题元素 <h1>, <h2> ...
<code>:animated</code>	<code>\$(":animated")</code>	所有动画元素
<code>:focus</code>	<code>\$(":focus")</code>	当前具有焦点的元素
<code>:contains(text)</code>	<code>\$(":contains('Hello')")</code>	所有包含文本 "Hello" 的元素
<code>:has(selector)</code>	<code>\$("div:has(p)")</code>	所有包含有 <p> 元素在其内的 <div> 元素
<code>:empty</code>	<code>\$(":empty")</code>	所有空元素
<code>:parent</code>	<code>\$(":parent")</code>	匹配所有含有子元素或者文本的父元素。
<code>:hidden</code>	<code>\$("p:hidden")</code>	所有隐藏的 <p> 元素
<code>:visible</code>	<code>\$("table:visible")</code>	所有可见的表格
<code>:root</code>	<code>\$(":root")</code>	文档的根元素
<code>:lang(language)</code>	<code>\$("p:lang(de)")</code>	所有 lang 属性值为 "de" 的 <p> 元素

第三类：属性选择器的扩展

<code>[attribute]</code>	<code>\$("[href]")</code>	所有带有 href 属性的元素
<code>[attribute=value]</code>	<code>\$("[href='default.htm']")</code>	所有带有 href 属性且值等于 "default.htm" 的元素
<code>[attribute!=value]</code>	<code>\$("[href!='default.htm']")</code>	所有带有 href 属性且值不等于 "default.htm" 的元素
<code>[attribute\$=value]</code>	<code>\$("[href\$='.jpg']")</code>	所有带有 href 属性且值以 ".jpg" 结尾的元素
<code>[attribute =value]</code>	<code>\$("[title ='Tomorrow']")</code>	所有带有 title 属性且值等于 'Tomorrow' 或者以 'Tomorrow' 后跟连接符作为开头的字符串
<code>[attribute^=value]</code>	<code>\$("[title^='Tom'])")</code>	所有带有 title 属性且值以 "Tom" 开头的元素
<code>[attribute~价值]</code>	<code>\$("[title~='hello'])")</code>	所有带有 title 属性且值包含单词 "hello" 的元素
<code>[attribute*=value]</code>	<code>\$("[title*='hello'])")</code>	所有带有 title 属性且值包含字符串 "hello" 的元素
<code>[name=value]</code>	<code>\$("input[id][name\$='man']")</code>	带有 id 属性，并且 name 属性以 man 结尾的输入框
<code>[name2=value2]</code>		

第四类：表单相关的选择器扩展

<code>:input</code>	<code>\$(":input")</code>	所有 input 元素
<code>:text</code>	<code>\$(":text")</code>	所有带有 type="text" 的 input 元素
<code>:password</code>	<code>\$(":password")</code>	所有带有 type="password" 的 input 元素
<code>:radio</code>	<code>\$(":radio")</code>	所有带有 type="radio" 的 input 元素
<code>:checkbox</code>	<code>\$(":checkbox")</code>	所有带有 type="checkbox" 的 input 元素
<code>:submit</code>	<code>\$(":submit")</code>	所有带有 type="submit" 的 input 元素
<code>:reset</code>	<code>\$(":reset")</code>	所有带有 type="reset" 的 input 元素
<code>:button</code>	<code>\$(":button")</code>	所有带有 type="button" 的 input 元素
<code>:image</code>	<code>\$(":image")</code>	所有带有 type="image" 的 input 元素
<code>:file</code>	<code>\$(":file")</code>	所有带有 type="file" 的 input 元素
<code>:enabled</code>	<code>\$(":enabled")</code>	所有启用的元素
<code>:disabled</code>	<code>\$(":disabled")</code>	所有禁用的元素
<code>:selected</code>	<code>\$(":selected")</code>	所有选定的下拉列表元素
<code>:checked</code>	<code>\$(":checked")</code>	所有选中的复选框选项
<code>.selector</code>	<code>\$(selector).selector</code>	在jQuery 1.7中已经不被赞成使用。返回传给jQuery()的原始选择器
<code>:target</code>	<code>\$("p:target")</code>	选择器将选中ID和URI中一个格式化的标识符相匹配的<p>元素

jQuery的方法

jQuery是一个优秀的 JS 库，它是一系列方法的集合，我们可以直接通过它所封装的方法来完成一些基本的操作

jQuery所有的东西都是方法，只有一个属性 `length`

```
<button type="button" id="btn1">按钮1</button>
<ul class="ul1">
    <li>第1项</li>
    <li class="ccc">第2项</li>
    <li>第3项</li>
    <li>第4项</li>
    <li class="aaa">
        <div>第5项</div>
        <a id="a1" href="#">百度一下</a>
        <p>这是段落</p>
    </li>
    <li>第6项</li>
    <li>第7项</li>
    <li>第8项</li>
    <li class="bbb">第9项</li>
    <li>第10项</li>
</ul>
```

现在我们就已上面的HTML代码为例子来实现我们的方法操习

DOM方法

1. 向子级元素选取children(selector?)

```
$(".ul1>li");           //获取所有的li  
$(".ul1").children();    //与上面相同  
  
$(".ul1>.aaa");  
$(".ul1").children(".aaa"); //children在选择的时候可以传入选择器进行二次选择
```

2. 向后代元素选取find(selector?)

```
$(".ul1 p");  
$(".ul1").find("p");
```

3. 向父级元素选取

- parent(selector?)
- parents(selector?)
- parentsUntil(selector?)

```
//CSS的选择器是不能选父级的。这里只能用JS操作完成  
// id="a1"的父级  
$("#a1").parent();  
//找所有的父级，直到html元素  
$("#a1").parents();  
// 找所有的父级，直到until这个元素为止，但不包含这个until元素  
$("#a1").parentsUntil("body");
```

4. 向后面的元素选取

- next(selector?)
- nextAll(selector?)
- nextUntil(selector?)

```
//下一个元素  
$(".aaa").next();  
// 后面所有的元素  
$(".aaa").nextAll();  
// 所有的元素，直接到bbb为止，但不是包含bbb  
$(".aaa").nextUntil(".bbb");
```

5. 向前面的元素选取

- prev(selector?)
- prevAll(selector?)
- prevUntil(selector?)

```
//前一项  
$(".aaa").prev();  
//前面的所有  
$(".aaa").prevAll();  
//前面的直到.ccc停止，不包含.ccc  
$(".aaa").prevUntil(".ccc");
```

6. 兄弟元素选取siblings(selector)

```
$(".aaa").siblings();
$(".aaa").siblings(".bbb,.ccc");
```

7. 过滤选取filter(selector)

这个方法可以在已经选择的元素里面再次进行过滤

```
$(".ul1>li");
// 在前抽选择的所有元素里面，过滤符合要求的
$(".ul1>li").filter(".ccc");
```

8. 排除选取not(selector)

这个方法可以在已经选择的元素里面，再次进行排除不符合要求的元素

```
$(".ul1>li").not(".aaa");           //在选中的li里面排除.aaa
$(".ul1>li:not(.aaa)");             //与上面的效果一样
```

9. 指定索引选取

这个方法可以在已经选择的元素里面找到指定的索引选取

```
//第1项li
$(".ul1>li:first-child");
$(".ul1>li").first();
//最后1项
$(".ul1>li:last-child");
$(".ul1>li").last();
//第3项
$(".ul1>li:nth-child(3)");
$(".ul1>li").eq(2);      //注意eq的索引是从0开始的
```

10. 获取当前元素的索引

这个方法非常重要，一个元素可以通过这个方法来获取当前元素在父级里面的索引

```
$(".aaa").index();
```

CSS样式方法

在之前的DOM里面，我们已经通过很多种方式来获取元素的CSS样式，以及操作元素的样式，其实在jQuery里面也有很多方法

1. `classList`
 - `add()` 添加样式类
 - `remove()` 移除样式类
 - `toggle()` 如果有就删除，没有就添加，（切换）
2. `style` 属性操作，赋值style样式
3. `window.getComputedStyle()` 获取元素的样式

在jQuery里面，上面的操作演变成了下面的方法

1. `addClass()` 添加一个样式类
2. `removeClass()` 移除样式类
3. `toggleClass()` 如果有就删除，没有就添加，（切换）
4. `css()` 直接在元素的 `style` 上面设置

```

<style>
    .aaa{
        border: 2px solid red;
        color: blue;
    }
    .bbb{
        width: 300px;
        height: 300px;
    }
</style>
<div id="div1" style="font-weight: bold;" class="aaa">盒子</div>

```

现在我们就以上面的代码为例子，来完成相关操作过程

```

$("#div1").addClass("bbb");           //添加class样式
$("#div1").removeClass("bbb");         //删除class样式
$("#div1").toggleClass("bbb");        //切换样式

```

除了使用上面的方法添加样式与删除样式以外，我们还可以通过 `css()` 来设置某个样式属性

```

$("#div1").css("background-color","yellow");      //本质就是通过style去设置样式
//设置多个的时候就传入对象
$("#div1").css({
    borderWidth:"10px",
    "text-decoration":"underline black solid",
    height:"100px"
});

```

在通过 `css` 去设置样式的时候，一定要注意，如果原来的 `css` 属性名里面有 `-` 则要注意转义，当然如果不想转义也可以，那就加引号包裹

`css` 方法不仅仅可以用来设置，还可以用来获取

```

$("#div1").css("height");           //获取css当中的height属性
$("#div1").css("width");            //获取css当中的width属性
$("#div1").css("color");           //获取css当中的color属性
//这个东西的本质点就是window.getComputedStyle()来完成

```

事件方法

在DOM的学习里面，我们已经给同学们提过很多遍，DOM的重点在DOM操作与事件里面，同样，在jQuery里面，也有相应的DOM的简化操作

在jQuery里面，它将常用的DOM事件进行了封装，形成一些特有的方法

```

<body>
    <button type="button" class="btn">按钮1</button>
    <button type="button" class="btn">按钮2</button>
    <button type="button" class="btn">按钮3</button>
    <button type="button" class="btn">按钮4</button>
</body>
<script src="./js/jquery-3.6.1.js"></script>
<script>
    $(".btn").click(function(event){
        console.log(this);
    });

```

```

$( ".btn" ).mouseover(function(event){
    console.log("鼠标进来了")
});
</script>

```

常见 DOM 事件:

鼠标事件	键盘事件	表单事件	文档/窗口事件
<u>click</u>	<u>keypress</u>	<u>submit</u>	<u>load</u>
<u>dblclick</u>	<u>keydown</u>	<u>change</u>	<u>resize</u>
<u>mouseenter</u>	<u>keyup</u>	<u>focus</u>	<u>scroll</u>
<u>mouseleave</u>		<u>blur</u>	<u>unload</u>
<u>hover</u>			

注意: jQuery只是把常用的事件封装成了一个事件方法，那么对于那些学中用的事件，那怎么办呢？

这个时候jQuery提供一个 **on** 的方法单独的所有的事件进行绑定

JQuery的**on**的事件绑定

所有的事件都可以通过 **on** 的方式来绑定

```

$(".btn").on("click",function(event){
    console.log(this);
});
$(".btn").on("mouseover",function(event){
    console.log("鼠标进入了");
});

```

所有DOM元素的事件都 可以通过 **on** 来完成绑定，这一点就比较方便

JQuery同一个元素的多个事件绑定

在DOM的操作过程当中，我们经常会对同一个元素进行多个事件绑定，这个时候的jQuery就非常方便了

```

$(".btn").on({
    click:function(event){
        console.log(this,"单击");
    },
    mouseover:function(event){
        console.log(this,"鼠标进入");
    }
});

```

JQuery中**on**的事件委托绑定

事件委托是DOM操作当中的重点，它利用了事件传播的特点，将本来应该是自己绑定的事件结果绑定在它的外层元素，然后通过事件冒泡来触发事件

```

<ul class="ul1">
    <li>第一项</li>
    <li>第二项</li>
    <li>第三项</li>
</ul>

```

我们现在需要对 **ul1** 下面的所有 **li** 进行事件绑定

```
$(".ul1>li").on("click",function(event){
    console.log(this);
});
```

通过上面的方式绑定以后，如果我们又新增了某些内容，这些新增的内容就不会有事件

```
$(".ul1").append("<li>新加入的项</li>");
```

为了解决这样的问题，我们之间的原生的DOM操作里面使用了事件委托，现在jQuery里面也有事件委托

```
$(".ul1").on("click","li",function(event){
    console.log(this);
});
```

在上面的代码里面，我们的事件仍然是绑定在了 `ul1` 上面，但是它判断了事件的触发者 `li`，这就是典型的事件委托

jQuery中的one单次事件绑定

有时候我们希望某一些事件执行一次以后就解除绑定，这个时候的事件就只会触发一次。如果希望实现这样的效果，可以使用 `one` 来进行事件绑定

```
$(".btn1").one("click", function (event) {
    console.log(this);
});
```

注意事项

```
<body>
    <ul class="ul1">
        <li>第一项</li>
        <li class="aaa">第二项</li>
        <li>第三项</li>
        <li class="aaa">第四项</li>
        <li>第五项</li>
    </ul>
</body>
<script src="./js/jquery-3.6.1.js"></script>
<script>
    // 对.aaa做事件委托的单次绑定
    $(".ul1").one("click", ".aaa", function (event) {
        console.log(this);
    })
</script>
```

jQuery中的off移除事件

```
<body>
    <button type="button" id="btn1">按钮</button>
</body>
<script src="./js/jquery-3.6.1.js"></script>
<script>
    function abc() {
        console.log("你好啊");
    }
</script>
```

```

}

function def() {
    console.log("hello wolrd");
}

$("#btn1").on("click", abc);
$("#btn1").on("click", def);

// $("#btn1").off("click");           //移除了整个click事件
$("#btn1").off("click", abc);       //只移除abc的这次事件监听

// var btn1 = document.querySelector("#btn1");
// btn1.addEventListener("click",abc);
// btn1.removeEventListener("click",abc);

```

属性方法

属性方法主要用于操作DOM的属性，jQuery也封装了一些操作dom属性的方法

1. **html(value?)**方法，该方法用于获取或设置 `innerHTML` 的属性

```

// 没有放参数就是获取
$("#div1").html();
//如果放了一个参数就代表赋值
$("#div1").html("<input type='text' />")

```

2. **text(value?)**方法，该方法用于获取或设置 `innerText` 的属性

```

//没有参数
$("#div1").text();
//如果放了参数就是对innerText赋值
$("#div1").text("<input type='text' />");

```

3. **val(value?)**方法，该方法用于获取或设置表单元素的 `value` 属性

```

//没有参数就是取值
$("#txt1").val();
//有参数就是赋值
$("#txt1").val("标哥哥");

```

4. **attr(name,value)**方法，该方法相当于DOM里面的 `getAttribute` 与 `setAttribute`

```

//属性取值，相当于getAttribute("type")
$("#txt1").attr("type");
//属性赋值，相当于setAttribute("type","password")
$("#txt1").attr("type","password");

```

5. **prop(name,value)**方法，用于获取或设置DOM里面单属性的值,也可以获取DOM对象上面的属性

```

$("#ck1").prop("checked");           //这就是获取checked的属性值
$("#ck1").prop("checked",false);

```

6. **removeAttr(name)**删除一个属性，这个方法相当于DOM里面的 `removeAttribute()`

```
$("#txt1").removeAttr("type")
```

7. data()方法，用于获取或设置以 data- 开头的自定义属性

```
<body>
    <div id="div1" data-age="18" ></div>
</body>
<script src="./js/jquery-3.6.1.js"></script>
<script>
    $("#div1").attr("data-age");           //18
    $("#div1").data("age");                //18

    $("#div1").attr("data-age",50);         //设置为50
    $("#div1").attr("data-age");           //50
    $("#div1").data("age");                //18      问题就在这里
</script>
```

坑：data()方法获取的属性如果后期发生更改以后，这个值是不会自动更改的
请同学们慎用这个方法

8. remove()删除元素自身，这个方法与DOM里面的remove保持一致

```
$(".ul1").remove();
```

9. empty()清空所有的子元素

```
$(".ul1").empty();
```

10. 新增元素

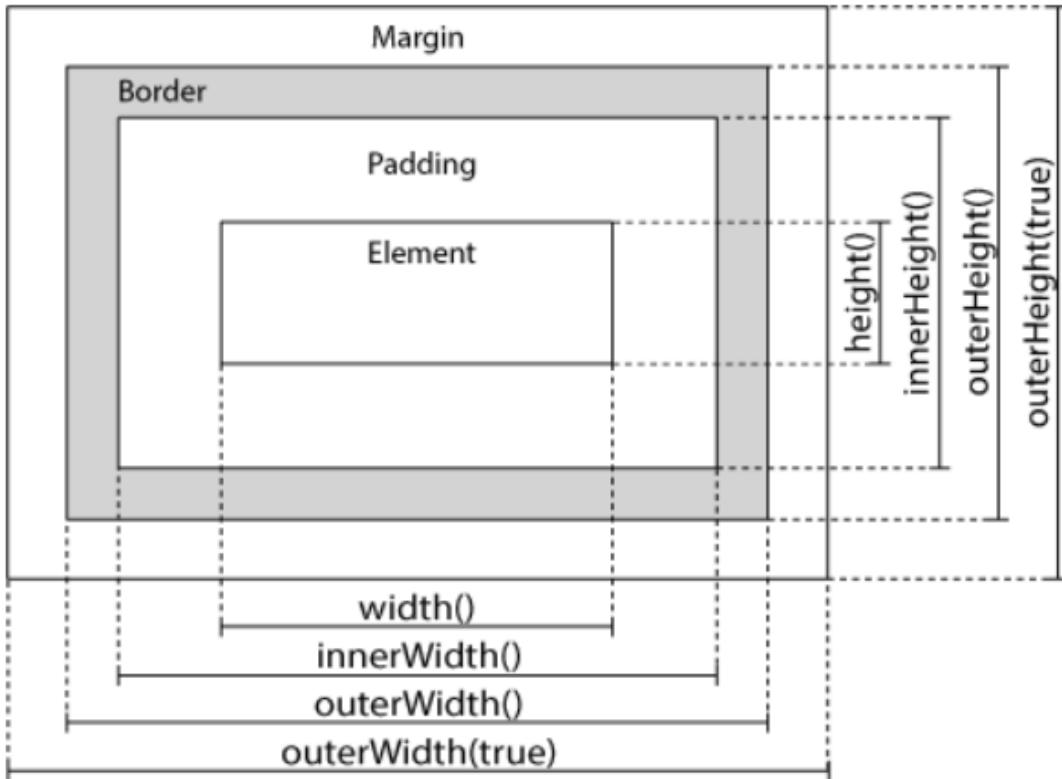
- append() - 在被选元素的结尾插入内容，相当于参数 beforeEnd
- prepend() - 在被选元素的开头插入内容，相当于 afterBegin
- after() - 在被选元素之后插入内容m，相当于 afterEnd
- before() - 在被选元素之前插入内容，相当于 beforeBegin

```
$(".ul1").append("<li>append</li>");
$(".ul1").prepend("<li>prepend</li>");
$(".ul1").before("<li>before</li>");
$(".ul1").after("<li>after</li>");
```

上面的方法不仅仅可以追加HTML的字符串，还可以追加元素

尺寸方法

在之前学习DOM的时候，我们有一些方法用来获取dom元素的大小，如
`offsetWidth/clientWidth/scrollWidth`，这些大小在 jQuery里面是可以快速的获取的



- **width()** 代表元素的大小
- **innerWidth()** 元素的大小+内边距的大小
- **outerWidth()** 元素大小+内边距大小+边框的大小
- **outerWidth(true)** 元素大小+内边距大小+边框大小+外边距

动画方法

jQuery里面的动画方法更多的情况下应该把它理解为CSS里面的过渡效果，它可以让元素在执行变化的时候有一个过渡的效果

1. **show(speed,callback)/hide(speed,callback)/toggle(speed,callback)** 隐藏与显示

当然们如果想要隐藏一个元素或显示一个元素的时候就可以使用上面的方法。我们也看到了上面的方法里面可以接收参数

如果没有写参数，则元素就是直接隐藏与显示，如果加了参数就代表元素会有一个速度进行过渡，它的速度值有以下几个

- **slow** 慢速的
- **normal** 正常的
- **fast** 快速的
- **miniSeconds** 毫秒

```
$(".box").hide();
$(".box").show();
$(".box").hide("slow");           //慢慢的隐藏掉
$(".box").toggle(5000);          //在隐藏与显示的时候会有5秒的时长
```

2. **slideUp(speed,callback)/slideDown(speed,callback)/slideToggle(speed,callback)** 上拉与下滑

这个里面的速度参数与上面的是一样的，如果不设置速度，默认就是 **normal**

3. `fadeIn(speed,callback)/fadeOut(speed,callback)/fadeToggle(speed,callback)/fadeTo(speed,opacity,callback)` 渐隐渐现

这里面多了一个 `fadeTo`，这个方法是将元素设置成指定的透明度

4. `animate(css属性,时间,callback)` 自定义动画方法

```
$(".box").animate({  
    marginLeft: "100px",  
    "border-radius": "50%"  
}, 5000);
```

`animate`这个方法本质上就是让元素在执行CSS样式变化的时候添加一个过渡的效果

5. `stop()` 停止之前未完成的动画

jQuery里面的动画内部是通过 `setInterval/setTimeout` 来实现的，所以它全部都是异步的

jQuery动画的回调函数

我们刚刚已经说过了，jQuery里面的动画内部是通过 `setInterval/setTimeout` 来实现的，所以它是异步，有了这个特点，我们就来看一看下面的代码

```
$("#btn1").click(function(){  
    $(".box").slideUp(5000);  
    console.log("hello");  
});
```

在上面的代码里面，虽然 `slideUp` 需要5000毫秒的时间，但是我们后面的代码 `console.log()` 并没有在5秒以后执行，而是立即执行，这就是因为 `slideUp()` 这个动画方法是异步的，而异步执行

现有有这么个需求，需要某些代码必须在动画结束以后才可以执行，怎么办呢？

```
$("#btn1").click(function () {  
    $(".box").slideUp(5000, function () {  
        //这里就是回调函数  
        console.log("hello");  
    });  
});
```

Ajax方法

在jQuery里面，我们发起一个 `ajax` 的请求是非常简单的

```
var str = "http://www.softteam.xin:8888/public/musicData/musicData.json";  
$("#btn1").click(function () {  
    $.ajax({  
        method: "GET",  
        url: str,  
        dataType: "json", //原生xhr对象里面的responseType  
        success: function(result){  
            //请求成功以后的回调  
            console.log(result)  
        },  
        error: function(error){  
            //请求失败以后的回调  
            console.log(error);  
        }  
    });  
});
```

```
        },
        complete:function(){
            // 请求完成以后的回调
            // 无论最终的结果是成功的，还是失败的，它都会完成
            console.log("请求完成")
        }
    });
});
```

上面的代码就是 `ajax` 最基本的jQuery封装代码，使用起来也非常方法

jQuery也知道我们以后会频繁的去使用ajax请求，jQuery也知道目前使用得最多的2种请求分别是 `get` 请求和 `post` 请求，所以jQuery在内部对这2个东西做了从化

get请求

```
var str = "http://www.softeam.xin:8888/public/musicData/musicData.json";
$("#btn1").click(function () {
    $.get(str, function (result) {
        console.log(result)
    });
});
```

post请求

```
$.post(str,function(result){
    //请求的结果
});
```

其它方法

1. `each` 遍历方法

之前在数组里面有一个方法叫 `forEach`，但是这个方法只有数组里面才有，其它的类数组里面可能没有，如果需要使用 `forEach` 就必须将类数组转换成数组

jQuery直接提供了一个像这样的方法 `each` 来方便我们遍历

```
var arr = ["a","b","c","d","e"];
$.each(arr,function(index,item){
    //index代表索引，item代表遍历的每一项
    console.log(index,item)
})
```

还可以这样写

```
$(arr).each(function(index,item){
    console.log(index,item);
})
```

2. `map` 遍历方法

这个方法与之前的数组里面的 `map` 方法一样的，可以接收每一次回调函数的返回值

```
<body>
<ul class="ul1">
```

```

<li>张三</li>
<li>李四</li>
<li>王五</li>
<li>赵六</li>
</ul>
</body>
<script src="./js/jquery-3.6.1.js"></script>
<script>
    var result = $(".ul1>li").map(function(index,item){
        // item仍然代表每一项dom
        return $(item).text();
    });
</script>

```

最终的结果如下

```

▼ jQuery.fn.init(4) ['张三', '李四', '王五', '赵六',
  prevObject: jQuery.fn.init(4)] ⓘ
  0: "张三"
  1: "李四"
  2: "王五"
  3: "赵六"
  length: 4
▶ prevObject: jQuery.fn.init(4) [li, li, li, li, p]
▶ [[Prototype]]: Object(0)

```

3. `toArray()` 方法

这个方法可以将类数组转换成数组

```

<body>
    <ul class="ul1">
        <li>张三</li>
        <li>李四</li>
        <li>王五</li>
        <li>赵六</li>
    </ul>
</body>
<script src="./js/jquery-3.6.1.js"></script>
<script>
    // toArray可以将类数组转换成数组
    var result = $(".ul1>li");
    // 要把上面的jQuery对象的类数组转换成数组
    var result2 = Array.prototype.slice.call(result);
    // 在jQuery里面，更方法

    var result3 = $(".ul1>li").toArray();
</script>

```

4. `makeArray()` 方法

这个方法与上面的方法保持一致，也是将类数组转换成数组

```

var result = $(".ul1>li");
var result4 = $.makeArray(result);

```

jQuery的链式语法

在学链式语法之前，一定要弄清楚jQuery所有的操作都是设置或获取。但是jQuery执行的是批量设置，单个获取

```
<body>
    <button type="button" class="btn1">按钮1</button>
    <button type="button" class="btn1">按钮2</button>
    <button type="button" class="btn1">按钮3</button>
    <button type="button" class="btn1">按钮4</button>
    <button type="button" class="btn1">按钮5</button>
</body>
<script src="./js/jquery-3.6.1.js"></script>
<script>
// 批量设置
$(".btn1");      //选中3个按钮
// $(".btn1").html("标哥哥");           //所有的按钮都会变成标哥哥
// 单个获取
var str = $(".btn1").html();
</script>
```

通过上面的例子我们可以看到jQuery设置与获取操作是不一样的

通过jQuery，可以把动作/方法链接在一起（仅限于设置操作）。如下所示

```
<body>
    <button type="button" class="btn1">按钮1</button>
</body>
<script src="./js/jquery-3.6.1.js"></script>
<script>
$( ".btn1");
// 将内容换成标哥哥
// 将宽度设置成150px
// 绑定一个单击事件
$(".btn1").html("标哥哥");
$(".btn1").css("width", "150px");
$(".btn1").on("click", function(event){
    console.log("你好");
})
</script>
```

上面的代码虽然完成了功能，但是，并不完美，我们完全可以使用链式语法去完成

```
//所谓的链式语法就是一次选择，多次设置
$("选择器").方法1().方法2().方法3();
//只要是设置的操作，可以一直添加
$(".btn1").html("标哥哥").css("width", "150px").on("click", function (event) {
    console.log("你好");
})
```

同时，jQuery的链式语法在DOM选取的时候也是可以进行的，如下所示

```
<body>
<ul>
    <li>第1项</li>
    <li class="aaa">第2项</li>
```

```
<li>第3项</li>
<li>
    <a href="#">第4项</a>
    <p>哈哈哈</p>
</li>
<li>第5项</li>
</ul>
</body>
<script src="./js/jquery-3.6.1.js"></script>
<script>
    $(".aaa").parent().parent(); //body
    $(".aaa").next().next().children("p")
</script>
```

DOM对象与jQuery对象的切换

jQuery对象转普通的DOM对象

我们可以通过 `$(选择器)` 的方式来得到jQuery选中以后的结果，这个结果它是一个包含了当前元素的类数组，如下所示

```
var result = $("#btn1");

> result
< ▶ jQuery.fn.init [button#btn1] ⓘ
  ▷ 0: button#btn1
  length: 1
  ▷ [[Prototype]]: Object(0)
```

它不是dom对象，它是一个jQuery对象，如果你想获取里面的第0个元素，应该怎么办呢？如果想获取里面某一个真实的DOM怎么办呢？

```
var btn1 = $("#btn1")[0]; //这个时候的btn就是真实的dom
```

普通的DOM对象如何转换成jQuery对象

当一个普通的DOM对象如果想转换成jQuery的对象，则可以直接用选择器选择

```
var btn1 = document.querySelector("#btn1");
$(btn1); //直接通过jQuery的选择器去完成操作
//所有后期经常会有这样一个这样的操作
$(this);
```

jQuery的文档就绪函数

我们要弄清楚一点，jQuery是进行DOM操作，而DOM在加载的时候是需要时间的，如 `img` 标签它会加载一张图片，如果网络情况不是很好，加载就会变非常慢。

网页上面的内容加载完之前，我们不应该去操作网页，一定要等网页加载完以后再操作网页

```
<!DOCTYPE html>
<html lang="zh">
<head>
```

```
<meta charset="UTF-8">
<meta http-equiv="X-UA-Compatible" content="IE=edge">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>加载</title>
<script src=".js/jquery-3.6.1.js"></script>
<script>
    $("#btn1").click(function(event){
        alert("你好啊");
    })
</script>
</head>
<body>
    <button type="button" id="btn1">按钮1</button>
</body>

</html>
```

在上面的代码里面，我们的事件是不会绑定成功的，因为在进行元素的选择的时候，`button` 按钮还没有被页面加载

为了解决上面的问题，我们通常这么做

第一种方式：通过window.onload事件来完成

```
<!DOCTYPE html>
<html lang="zh">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>加载</title>
    <script src=".js/jquery-3.6.1.js"></script>
    <script>
        window.onload = function () {
            console.log("网页加载完成");
            $("#btn1").click(function (event) {
                alert("你好啊");
            })
        }
    </script>
</head>
<body>
    <button type="button" id="btn1">按钮1</button>
</body>

</html>
```

这的`window.onload`代表网页加载完成以后的事件，所以当网页加载完了以后，我再去进行DOM的操作，这个时候的页面上面应该是已经有了`button` 按钮了，所以再对`button`进行事件绑定就不会有问题了

上面的方法好是好，但是有一个点不好，`window.onload`这个事件必须等待页面上面所有的元素都加载完成才会触发

```

<audio src="./music/02.mp3"></audio>
<video src="./v/03.mp4"></video>
```

上面的代码里面有三个标签，而3个标签都有 `src` 属性，有 `src` 属性就会有 `onload` 事件，所以如果 `window.onload` 的事件想要触发，必须要等上面的3个元素的全部加载完成

但是隐患非常大

```
           ←—— 加载成功，触发onload→
<audio src="./music/02.mp3"></audio>   ←—— 加载成功，触发onload→
<video src="./v/03.mp4"></video>      ←—— 加载失败 →
```

如果有元素加载比较慢或加载失败，就会影响 `window.onload` 的执行

通过 `DOMContentLoaded` 事件

3. `DOMContentLoaded` 事件

如前所述，`window` 的 `load` 事件会在页面中的一切都加载完毕时触发，但这个过程可能会因为要加载的外部资源过多而颇费周折。而 `DOMContentLoaded` 事件则在形成完整的 DOM 树之后就会触发，不理会图像、JavaScript 文件、CSS 文件或其他资源是否已经下载完毕。与 `load` 事件不同，`DOMContentLoaded` 支持在页面下载的早期添加事件处理程序，这也就意味着用户能够尽早地与页面进行交互。

要处理 `DOMContentLoaded` 事件，可以为 `document` 或 `window` 添加相应的事件处理程序（尽管这个事件会冒泡到 `window`，但它的目标实际上是 `document`）。来看下面的例子。

这个事件当网页一旦加载完成就可以立即触发

```
<!DOCTYPE html>
<html lang="zh">

    <head>
        <meta charset="UTF-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <title>加载</title>
        <script src="./js/jquery-3.6.1.js"></script>
        <script>
            document.addEventListener("DOMContentLoaded", function (event) {
                console.log("网页文档加载完成了");
                $("#btn1").click(function (event) {
                    alert("你好啊");
                })
            })
        </script>
    </head>

    <body>
        <button type="button" id="btn1">按钮1</button>
    </body>

</html>
```

jQuery的内部其实也是通过这个来实现的文档就绪，所以jQuery也自己封装了文档就绪函数

```
$(document).ready(function(event){
    console.log("jQuery的文档就绪事件");
})
```

上面文档就绪的函数可以简写

```
$(function(){
    //文档就绪函数
});
```

jQuery事件对象及补充点

jQuery的事件对象

只要是事件就会有事件对象，在jQuery里面也是有事件对象的，它的事件对象与原生DOM里面的事件对象是不一样的，有一定的区别

```
<script>
    var btn1 = document.querySelector("#btn1");
    btn1.addEventListener("click", function (event) {
        // 事件对象是有兼容性的
        event = event || window.event;

        console.log(event);
    });

    //—————正同是jQuery的代码—————
    $("#btn2").on("click", function(event){
        // 如果要获取事件对象，不需要做兼容性处理，jQuery的内部已经做了封装
        console.log(event);
    });
</script>
```

在上面的代码里在，我们可以看到对2个元素同时绑定了事件，一个是通过原生的DOM操作来绑定事件的，一个是通过jQuery框架来绑定的事件的，最后在控制台打印事件对象

```
▶ PointerEvent {isTrusted: true, pointerId: 1, width: 1, height: 1, pressure: 0, ...} 01事件对象.html:24
▶ jQuery.Event {originalEvent: PointerEvent, type: 'click', target: button#btn2, currentTarget: button#btn2, isDefaultPrevented: false, ...} 01事件对象.html:30
```

上面的打印结果很明显，jQuery的事件对象是一个自定义事件对象，这是因为jQuery把事件做了封装处理，这样事件就没有兼容性了

jQuery的事件对象是一个封装好的自定义对象，它屏蔽了不同浏览器之间的兼容性，这样做的到高效快捷的使用

区别

1. jQuery的事件对象当中有一个属性叫 `which`，它代表限鼠标按下了哪一个键，1代表鼠标左键，2代表鼠标中键，3代表鼠标右键，这一点就与原生的事件对象不一样，在原生的事件对象里面，如果我们要判断鼠标按下了哪个键，我们应该用 `button`，但是原生事件的 `button` 是有兼容性的
2. jQuery事件里面的 `stopPropagation()` 这个方法其实也是后来的封装，它相当于原生事件里面的2行代码

```
//原生事件里面，如果要阻止事件冒泡  
event.cancelBubble = true;  
event.stopPropagation();
```

3. 在jQuery的事件对象里面,如果主动的 `return false` 则相当于这个事件要阻止默认行为, 还要停止事件传播, 它相当于下面的三行代码

```
//相当于原生事件对象里的三行代码  
event.cancelBubble = true;  
event.stopPropagation();  
event.preventDefault();
```

4. jQuery的事件对象是一个封装好的事件对象 , 但是我们仍然可以获取到原生的事件对象



```
jQuery.Event {originalEvent: PointerEvent, type: 'click', target: button#btn1, currentTarget: button#btn1, ...} 这一个就是原生的事件对象, 如果在开发当中需要使用原生的事件对象, 就找它  
▶ faultPrevented: f, ...} ⓘ  
▶ currentTarget: button#btn1  
  data: undefined  
▶ delegateTarget: button#btn1  
▶ handleObj: {type: 'click', origType: 'click', data: undefined, guid: 2, handler: f, ...}  
▶ isDefaultPrevented: f returnTrue()  
▶ isPropagationStopped: f returnTrue()  
  jQuery361002558946026435227: true  
▶ originalEvent: PointerEvent  
  isTrusted: true  
  altKey: false  
  altitudeAngle: 1.5707963267948966  
  azimuthAngle: a
```

这个地方的 `originalEvent` 就可以找到原生的事件对象

jQuery事件委托详解

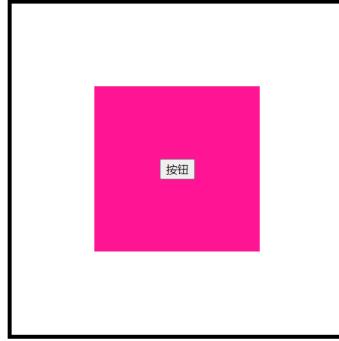
在之前DOM的事件委托里面, 我们已经讲到过2个基本的点

1. 事件的绑定者叫 `event.currentTarget`
2. 事件的触发者叫 `event.target`
3. 事件委托者叫 `event.delegateTarget` (这个是jQuery里面独有的)

但是在jQuery的下面, 还有一个对象叫事件委托者

我们先看原生的事件委托

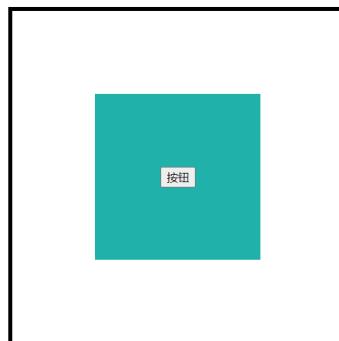
```
<body>  
  <div class="big-box">  
    <div class="box">  
      <button type="button" id="btn1">按钮</button>  
    </div>  
  </div>  
</body>  
<script>  
  var bigBox = document.querySelector(".big-box");  
  var box = document.querySelector(".box");  
  var btn1 = document.querySelector("#btn1");  
  // 我想绑定box的click事件, 但是委托给bigBox, 请问, 怎么办?  
  bigBox.addEventListener("click", function(event){  
    //判断事件的触发者  
    if(event.target.matches(".box")){  
      console.log("事件触发了");  
    }  
  });  
</script>
```



我们点击红色的盒子会产生事件，但是点击红色盒子里面的按钮，事件就没有效果，所以怎么办呢？

jQuery的事件委托

```
<body>
  <div class="big-box">
    <div class="box">
      <button type="button" id="btn1">按钮</button>
    </div>
  </div>
</body>
<script src="./js/jquery-3.6.1.js"></script>
<script>
  $(".big-box").on("click", ".box", function (event) {
    console.log("我是jQuery的事件委托");
  })
</script>
```



在真正的做事件委托的时候，一定是通过下面的代码来实现的

```
page1..  ↴
▼ path: Array(7)
  ► 0: button#btn1
  ► 1: div.box
  ► 2: div.big-box
  ► 3: body
  ► 4: html
  ► 5: document
  ► 6: Window {window: Window, self: Window, document: document, name: '', location: Location, ...}
    length: 7
  ► [[Prototype]]: Array(0)
```

```
<body>
  <div class="big-box">
    <div class="box">
      <button type="button" id="btn1">按钮</button>
    </div>
  </div>
```

```

    </div>
</body>
<script>
    var bigBox = document.querySelector(".big-box");
    var box = document.querySelector(".box");
    var btn1 = document.querySelector("#btn1");
    // 我想绑定box的click事件，但是委托给bigBox，请问，怎么办？
    bigBox.addEventListener("click",function(event){
        console.log(event);
        for(var i=0;i<event.path.length;i++){
            // console.log(event.path[i]);
            if(event.path[i] instanceof Element && event.path[i].matches(".box")){
                console.log("我触发了事件");
            }
        }
    });
</script>

```

jQuery事件绑定的方式

在目前的jQuery的版本里面，如果我们想绑定一个元素的事件，可以通过下面的方式来进行

```

<body>
    <div class="box">
        <button type="button" id="btn1">按钮1</button>
        <button type="button" id="btn2">按钮2</button>
        <button type="button" id="btn3">按钮3</button>
        <button type="button" id="btn4">按钮4</button>
        <button type="button" id="btn5">按钮5</button>
        <button type="button" id="btn6">按钮6</button>
        <button type="button" id="btn7">按钮7</button>
    </div>
</body>
<script src="./js/jquery-3.6.1.js"></script>
<script>
    $(function () {
        $("#btn1").click(function (event) {
            console.log("第一种：事件方法");
        });
        $("#btn2").on("click", function (event) {
            console.log("第二种：on完成");
        });
        $("#btn3").one("click", function (event) {
            console.log("第三种：one方式");
        });
        // 事件委托
        $(".box").on("click", "#btn4", function (event) {
            console.log("第四种：事件委托");
        });
        // ——————
        // jquery1.5-1.7的版本的事件绑定
        //这种方式，不要用，这是旧版本里面的
        $("#btn5").bind("click",function(event){
            console.log("第五种：bind绑定");
        });
    });

```

```

    });
    //移除bind绑定的事件
    $("#btn5").unbind("click");

    //这种是旧版本的事件委托，不要用
    $(".box").delegate("#btn6","click",function(event){
        console.log("第六种：委托");
    });
    $(".box").undelegate("#btn6","click");

    //jQuery1.5版本之前
    /*
    $("#btn7").live("click",function(event){

    });

    $("#btn").die("click",function(event){

    })
    */
})
</script>

```

一定看清楚上面的代码，事件绑定在不同的版本里面有不同的方式，终归而言有以下几种

1. 万能的 `on/off`
2. 单次的 `one`
3. 旧版本的 `bind/unbind`
4. 旧版本的事件委托 `delegate/undelegate`
5. 再旧一点的版本 `live/die`

jQuery的扩展方法

jQuery是一系列方法的集合，它把我们常用的就去进行了一些封装，所以我们在工作开发当中直接使用这些方法，但是如果我们要往这个上面添加一些方法呢？

```

$.get();
$.ajax();
$.each();
$.makeArray();
$(".div1").toArray()
$("button").show();

```

？思考：我们能否在这些方法再去扩展一些自己的方法，如下

```

$.biaogege();
$(".div1").hello();

```

这些方法都是jQuery不存在的，所以我们如果想使用就要自己扩展！

jQuery内部提供我们扩展方法的一个操作，可以直接来进行

第一种情况：直接在\$上面去扩展方法

```

$.extend({
    biaogege:function(){
        console.log("我是标哥");
    }
})
$.biaogege();

```

第二种情况：直接在选中的元素上面扩展

```

// 如果想在选择器上面扩展方法
$.fn.extend({
    hello:function(){
        console.log("你好啊，我是hello的方法")
    }
})
// $.hello();      报错
$("#btn1").hello(); //正常

```

有了这个东西，我们就可以结合我们之前所学习的模板引擎来完成相应的操作

```

<body>
    <ul class="ul1">
        </ul>
        <template id="temp1">
            {{each list item index}}
            <li>{{item}}</li>
            {{/each}}
        </template>
    </body>
<script src="./js/template-web.js"></script>
<script src="./js/jquery-3.6.1.js"></script>
<script>
    $.fn.extend({
        render:function(tempid,data){
            var htmlStr = template(tempid,data);
            // 这里的this就是你选择器选中的东西
            this.html(htmlStr);
        }
    })

    $(function () {
        var arr = ["张三", "李四", "王五", "赵六"];
        // var htmlStr = template("temp1",{
        //     list:arr
        // });
        // console.log(htmlStr);
        // $(".ul1").html(htmlStr)

        // 我想在jQuery上面扩展一个方法，直接通过下面的代码就可以完成
        $(".ul1").render("temp1",{
            list:arr
        })
    })
</script>

```

上面的代码就是直接在jQuery的选择器上面扩展了一个 `render` 的方法，用于渲染模板

jQuery的表单验证

jQuery的是我们的一个js框架，围绕这一个js框架，它有很多第三方的插件，如我们之前使用过的layer。所有依赖于jQuery的插件都必须导入在jQuery的后面。今天我们就来介绍一个工作当中使用得非常频繁的一个插件叫 `jquery.validator.js`

这个插件主要是做表单验证的

注册新学员

* 学生姓名 请输入学生姓名 学生姓名不能为空	* 性别 <input checked="" type="radio"/> 男 <input type="radio"/> 女
* 密码 请输入密码 密码不能为空	* 确认密码 请再次输入密码 确认密码不能为空
* 学校 请选择学校 学校不能为空	* 专业 请输入专业 专业不能为空
* 手机号 请输入手机号 手机号不能为空	QQ号 请输入QQ号
* 学历 请输入学历 学历不能为空	英语等级 请输入英语等级
* 毕业年限 请输入毕业年限 毕业年限不能为空	家庭电话 请输入家庭电话
* 所属班级 所属班级 所属班级不能为空	* 学生状态 <input checked="" type="radio"/> 正常 <input type="radio"/> 休学

上图就是一个表单，表单验证就是当提交数据或进行某个操作的时候 把表单里面的元素按照我们所需要的要求进行验证，在jQuery里面有一个现成的插件叫 `jquery.validator.js` 来完成

安装

这个插件直接在百度里面下载就可以了

```
<script src="./js/jquery-3.6.1.js"></script>
<script src="./js/jquery.validate.js"></script>
<script src="./js/messages_zh.js"></script>
```

配置规则

这个插件主要就是用来对表单里面的元素进行规则校验

```
<!DOCTYPE html>
<html lang="zh">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>表单验证</title>
</head>

<body>
    <form id="loginForm">
        <div>
            用户名: <input type="text" placeholder="请输入用户名" name="userName">
        </div>
        <div>
            密码: <input type="text" placeholder="请输入密码" name="pwd">
        </div>
    </form>
</body>
```

```

<div>
    <button type="button" id="btn1">登录</button>
</div>
</form>
</body>
<script src="./js/jquery-3.6.1.js"></script>
<script src="./js/jquery.validate.js"></script>
<script src="./js/messages_zh.js"></script>
<script>
$(function () {
    //第一步：要知道你要校验哪一个表单，我们给了表单一个id
    //第二步：你要校验哪些表单元素，就在这个元素上面添加一个name属性
    //第三步：在validate的下面写一个rules验证规则
    //第四步：当所有的规则配置好了以后，validate方法就会返回当前表单的一对象，有了这个表单的对象，我们就可以开始验证了
    var loginForm = $("#loginForm").validate({
        // 规则
        rules: {
            userName: {
                required: true
            },
            pwd: {
                required: true
            }
        },
        //自定义验证不通过的消息显示
        messages: {
            userName: {
                required: "兄弟，用户名不能为空"
            },
            pwd: {
                required: "小伙子，密码呢"
            }
        }
    });
    $("#btn1").click(function () {
        // 调用form()方法开始验证，得到一个boolean的结果，true代表验证通过，false代表验证不过
        var result = loginForm.form();
        if (result) {
            console.log("恭喜你表单验证通过了");
        } else {
            console.log("表单验证不通过");
        }
    })
})
</script>
</html>

```

上面的代码是一个最简单的表单验证的过程步骤代码，我们也已经看到了结果

？思考：上面的代码当中我们已经知道了jQuery.validator的使用方法和步骤，我们也知道 `required` 就代表这个字段必填，但是除了这个验证规则以后，还有没有其它的验证规则呢？

验证规则

jQuery.validator这个插件提供了很多内置的验证规则，如下表所示

默认校验规则

序号	规则	描述
1	<code>required:true</code>	必须输入的字段。
2	<code>remote:"check.php"</code>	使用 ajax 方法调用 check.php 验证输入值。
3	<code>email:true</code>	必须输入正确格式的电子邮件。
4	<code>url:true</code>	必须输入正确格式的网址。
5	<code>date:true</code>	必须输入正确格式的日期。日期校验 ie6 出错，慎用。
6	<code>dateISO:true</code>	必须输入正确格式的日期（ISO），例如：2009-06-23, 1998/01/22。只验证格式，不验证有效性。
7	<code>number:true</code>	必须输入合法的数字（负数，小数）。
8	<code>digits:true</code>	必须输入整数。
9	<code>creditcard:</code>	必须输入合法的信用卡号。
10	<code>equalTo:"#field"</code>	输入值必须和 #field 相同。
11	<code>accept:</code>	输入拥有合法后缀名的字符串（上传文件的后缀）。
12	<code>maxlength:5</code>	输入长度最多是 5 的字符串（汉字算一个字符）。
13	<code>minlength:10</code>	输入长度最小是 10 的字符串（汉字算一个字符）。
14	<code>rangelength:[5,10]</code>	输入长度必须介于 5 和 10 之间的字符串（汉字算一个字符）。
15	<code>range:[5,10]</code>	输入值必须介于 5 和 10 之间。
16	<code>max:5</code>	输入值不能大于 5。
17	<code>min:10</code>	输入值不能小于 10。

验证方式

在jQuery.validator的验证方式里面有两种验证方式

1. 配置式的验证
2. 侵入式的验证

配置式的验证

使用配置来完成相应的验证规则

```
<!DOCTYPE html>
<html lang="zh">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
```

```
<title>表单验证</title>
<style>
    .form-item {
        margin: 10px 0;
    }
</style>
</head>

<body>
    <form id="registerForm">
        <div class="form-item">
            用户名: <input type="text" name="userName">
        </div>
        <div class="form-item">
            密码: <input type="text" name="pwd" id="pwd">
        </div>
        <div class="form-item">
            确认密码: <input type="text" name="pwd2">
        </div>
        <div class="form-item">
            年龄: <input type="text" name="age">
        </div>
        <div class="form-item">
            身份证号: <input type="text" name="IDCard">
        </div>
        <div class="form-item">
            爱好: <input type="text" name="hobby">
        </div>
        <div class="form-item">
            <button type="button" id="btn-register">注册信息</button>
        </div>
    </form>
</body>
<script src="./js/jquery-3.6.1.js"></script>
<script src="./js/jquery.validate.js"></script>
<script src="./js/messages_zh.js"></script>
<script>
    $(function () {
        var registerForm = $("#registerForm").validate({
            // 验证规则
            rules: {
                userName: {
                    rangelength: [6, 10],
                    required: true
                },
                pwd: {
                    required: true
                },
                pwd2: {
                    required: true,
                    equalTo: "#pwd"
                },
                age: {
                    min: 0,
                    required: true
                },
                IDCard: {
                    required: true
                }
            }
        })
    })
</script>
```

```

        hobby: {
            required: true
        }
    },
    // 验证消息
    messages: {
        userName: {
            rangeLength: "用户名长度必须是6~10之间",
            required: "用户名必填"
        },
        pwd: {
            required: "密码不能为空"
        },
        pwd2: {
            required: "确认密码不能为空",
            equalTo: "两次密码必须相同"
        },
        age: {
            min: "年龄必须是大于0的数",
            required: "年龄不能为空"
        },
        hobby: {
            required: "爱好不能为空"
        }
    }
});

//当点击以后就开始验证
$("#btn-register").click(function () {
    var result = registerForm.form();
    if (result) {
        console.log("验证通过");
    } else {
        console.log("验证不能过");
    }
})
})
</script>

</html>
<!--

用户名: 长度必须是6-10位,必填
密码: 不能为空
确认密码: 必须与密码相同, 也不能为空
年龄: 正数
身份证: 必须填入身份证号
爱好: 不能为空

要求: 点击按钮以后去完成上面的验证, 怎么办
--&gt;
</pre>

```

像上面这种通过 `rules` 和 `messages` 来配置的我们叫配置式的验证规则

在上面有个问题要注意一下，我们的身份证号好像还没有验证，因为这个插件里面还没有身份证号的验证规则，怎么办呢？

侵入式的验证规则

侵入式的验证规则是把验证的规则不写在js里面，而是直接通过自定义属性写在html代码里面，jQuery表单验证里面新版本里面的特性

```
<!DOCTYPE html>
<html lang="zh">

    <head>
        <meta charset="UTF-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <title>表单验证</title>
    </head>

    <body>
        <form id="loginForm">
            <div>
                用户名: <input type="text" placeholder="请输入用户名" name="userName" data-rule-required="true"
                    data-msg-required="用户名不能为空">
            </div>
            <div>
                密码: <input type="text" placeholder="请输入密码" name="pwd" data-rule-required="true"
                    data-msg-required="密码不能为空">
            </div>
            <div>
                <button type="button" id="btn1">登录</button>
            </div>
        </form>
    </body>
    <script src="./js/jquery-3.6.1.js"></script>
    <script src="./js/jquery.validate.js"></script>
    <script src="./js/messages_zh.js"></script>
    <script>
        $(function () {
            var loginForm = $("#loginForm").validate();
            $("#btn1").click(function(){
                var result = loginForm.form();
                if(result){
                    console.log("验证通过了");
                }
                else{
                    console.log("很遗憾，验证不通过");
                }
            })
        })
    </script>
</html>
```

在上面的代码当中，我们没有把验证规则写在 `js` 的代码里面，而是写在了自定义属性 `data-` 里面

- `data-rule-xxx` 代表的是添加一个验证规则，如 `data-rule-required="true"` 或 `data-rule-min="0"`
- `data-msg-xxx` 代表的是当某一个规则校验不通过的时候的提示信息，如 `data-msg-required="用户名不能为空"`

无论是通过哪种方式来进行验证，表单元素上面的name的属性值都不能丢

自定义验证规则

虽然这个表单验证的插件帮助我们解决了很多验证的问题，但是仍然有些问题不能够实现，很多场景下面我们需要自定义校验规则，怎么办呢？

`jQuery.validator` 这个插件可以添加自定义的校验规则，最常见的就是把正则表达式的验证规则添加进去，所以这里我只展示正则表达式的扩展（其它的原理是一样的）

The screenshot shows a code editor with a tooltip for the `data-rule-regexp` attribute. The tooltip details the method implementation, parameters, value, element, params, and message.

method
Type: Function()
The actual method implementation, returning true if an element is valid. First argument: Current value. Second argument: Validated element. Third argument: Parameters.

value
Type: String
the current value of the validated element

element
Type: Element
the element to be validated

params
Type: Object
parameters specified for the method, e.g. for min: 5, the parameter is 5, [1, 5] if it's [1, 5]

message

```
<6> <meta http-equiv="X-UA-Compatible" content="IE=edge">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>自定义校验</title>
<script src=".js/jquery-3.6.1.js"></script>
<script src=".js/jquery.validate.js"></script>
<script src=".js/messages_zh.js"></script>
<script>
$(function() {
    // 添加自定义的验证规则这个插件可以添加自定义的校验规则，最常见的就是把正则表达式的验证规则添加进去，所以这里我只展示正则表达式的扩展（其它的原理是一样的）
    $.validator.addMethod("regexp", function(value, element, params) {
        // 这里的function就是你自定义的验证函数
        // params代表的就是data-rule-regexp="^等号后面的内容"
        // value代表当前表单元素的内容
        // element代表当前验证的元素
    });
});</script>
```

```
<!DOCTYPE html>
<html lang="zh">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>自定义校验</title>
</head>

<body>
    <form id="testForm">
        <div>
            用户名: <input type="text" name="userName">
        </div>
        <div>
            身份证号: <input type="text" name="IDCard" data-rule-regexp="^\d{18}$" data-msg-regexp="身份证号格式不对">
        </div>
    </form>
</body>
```

```

<div>
    <button type="button" id="btn1">我要添加</button>
</div>
</form>
</body>
<script src="./js/jquery-3.6.1.js"></script>
<script src="./js/jquery.validate.js"></script>
<script src="./js/messages_zh.js"></script>
<script>
$(function() {
    // 添加自定义的验证规则
    $.validator.addMethod("regexp", function(value, element, params) {
        //这里的function就是你自定义的验证函数
        //params代表的就是data-rule-regexp=""等号后面的内容
        //value代表当前表单元素的内容(值)
        //element代表当前验证元素
        var reg = new RegExp(params);
        return reg.test(value);
    });

    var testForm = $("#testForm").validate();
    $("#btn1").click(function(event){
        var result = testForm.form();
        if(result){
            console.log("验证通过");
        }
        else{
            console.log("验证不通过");
        }
    })
})
</script>
</html>

```

自定义消息提示的样式

当我们的验证不通过以后会有一个消息提示，其实这个消息提示的样式，我们是可以自己定义的

```

var testForm = $("#testForm").validate({
    errorClass:"error-text"
});

```

其中的 `error-text` 代表就是自己写的CSS样式，最终效果如下

用户名:	<input type="text"/>	用户名不能为空
身份证号:	<input type="text"/>	身份证号格式不对
<input type="button" value="我要添加"/>		

自定义消息提示的位置

jQuery的表单验证默认情况下会把所有验证不通过的消息内容追加到当前验证元素的后面

```
var testForm = $("#testForm").validate({  
    errorClass: "error-text",  
    errorPlacement: function(error, element){  
        // error代表的就是这个错误提示的消息  
        // element代表你目前正在验证的这个元素  
        $(".box").append(error);  
    }  
});
```

用户名：

身份证号：

我要添加

用户名不能为空

身份证号格式不对

上面的代码当中，它把我们的错误的消息放在了一个指定的盒子里面

作业与练习

请完成下面界面的布局，并完成表单验证的内容

新增学生信息

学生姓名	<input type="text" value="请输入学生姓名"/>	学生姓名不能为空
学生性别	<input type="text" value="男"/>	学生年龄不能为空
学生年龄	<input type="text" value="请输入学生年龄"/>	手机号不能为空
手机号码	<input type="text" value="请输入手机号码"/>	班级名称不能为空
所属班级	<input type="text" value="请输入班级名称"/>	地址不能为空
家庭地址	<input type="text" value="请输入家庭地址"/>	请至少选择一个爱好
爱好	<input type="checkbox"/> 看书 <input type="checkbox"/> 睡觉 <input type="checkbox"/> 玩游戏 <input type="checkbox"/> 逛街 <input type="checkbox"/> 写代码	
<button>保存数据</button> <button>返回列表</button>		

有上面一个表单，当点击“保存数据”以后，要对表单做一次数据校验，校验的要求如下



ECMAScript 6

之前我们学习过ES5，ES的全称叫ECMAScript，它是JavaScript的一部分，ES主要包含了语法，关键字，流程控制，运行符，面向对象，数据类型等。

ES6的全称叫ECMAScript6，也叫ECMAScript 2015，它是2015年发布的，它是新一代的ECMAScript的语法标准与规则，只涉及到了ES的部分，不涉及任何的DOM与BOM

ES6主要的技术如下

1. 变量，常量
 2. 取值与赋值，解构
 3. 运算符的扩展，展开运算符，指数运算符
 4. 字符串的扩展
 5. 数组的扩展
 6. 函数的扩展
 7. Set单值集合
 8. Map键值对集合
 9. 对象的扩展及class/extends关键字
 10. 生成器与迭代器及迭代器接口
 11. 反射Reflect
 12. 代理Proxy
 13. Promise异步处理，`async` 及 `await` 的使用
 14. ES6的模块化处理ESModule
 15. CommonJS模块化
-

let变量定义

在之前的ES5里面，如果我们想定义一个变量，我们可以使用 `var` 关键字来进行，关于 `var` 的特点，我们也知道以下几个

1. `var` 没有数据类型
2. `var` 有一个建立阶段
3. `var` 定义的变量没有块级作用，只能通过函数才能形成作用域

```
//var定义的变量有一个建立阶段，相当于变量提前声明
console.log(a);
var a = 123;
```

```
//这里不会报错，因为var定义的变量没有块级作用域
{
    var b = 456;
}
console.log(b);
```

针对上面的问题，其实就有很多不好的

为了解决上面的问题，ES6里面推出一个新的关键字叫 `let`，这个关键字也是用来定义变量的

1. `let` 定义的变量也是没有数据类型的
2. `let` 定义的变量没有建立阶段，只有执行阶段，所以必须先定义，后使用

```
let a = 123;
console.log(a);           //正常

console.log(b);           //这里会报错，let没有建立阶段，在定义之前不可使用
let b = "标哥哥";
```

3. let 定义的变量是有块级作用域的，它有花括号为作用域

```
{
  let c = "标哥哥";
  console.log(c);           //正确的
}
console.log(c);           //报错，访问不到内部的c变量
```

4. 在同一个作用域不可以定义同名变量（变量名不能重复）

```
let c = 123;
{
  let c = 456;
  console.log(c);           //因为在不同的作用域，不会报错
}
```

但是如果在同一个作用域里面就会报错

```
let c = 123;
console.log(c);
let c = 456;               //这里直接报错
console.log(c);
```

暂时性死区

let定义变量有它的特点，在使用的时候可能会因为一些使用不当照成一些错误

```
let a = 123;
{
  console.log(a);           //这里的代码是没有问题的，因为内部作用域有就会从外部去找
}
```

如果这个代码写成这样，就会有问题

```
let a = 123;
{
  console.log(a);           //这里就形成了一个暂时性死区
  let a = 456;
}
```

暂时性死区是因为代码的书写不当造成的一个问题

const常量

- 变量：可以变化的数据叫变量，它通过 `var, let` 来定义
- 常量：不会变化的数据叫常量，如 `Math.PI`，通过 `const` 来定义

```
// 定义变量
let a = 123;
console.log(a);
a = 456;
console.log(a);

// 常量，不可更改
const b = "标哥哥";
console.log(b);
b = "帅哥哥";           // 直接报错，常量一旦定久就不可更改
console.log(b);
```

注意： `const` 关键字具备 `let` 关键字的所有特点，并且定义以后不可改变

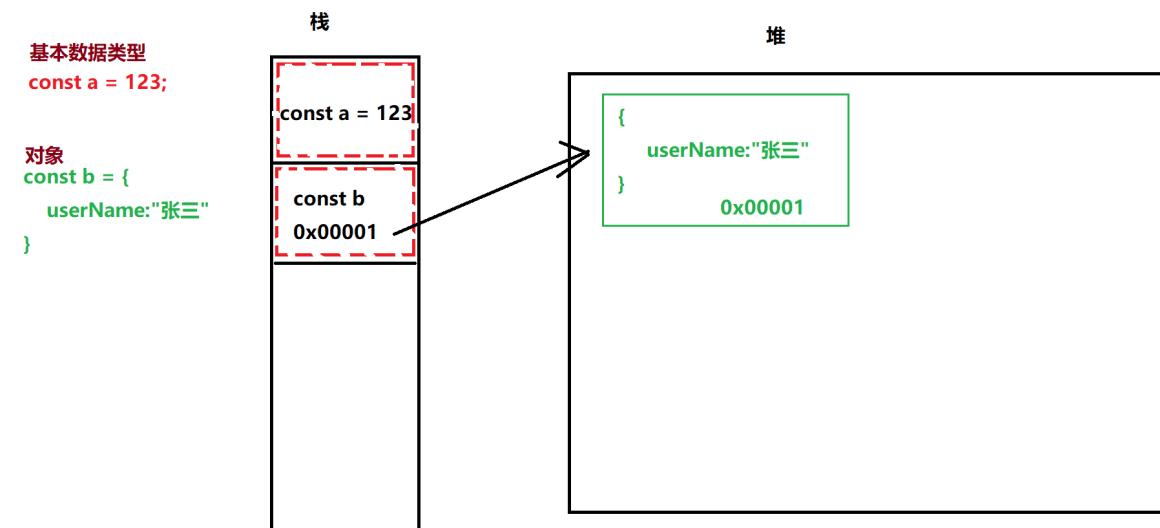
总结：

1. 没有建立阶段，先定义，后使用
2. 会形成块级作用域，以花括号来隔离
3. 在一个作用域内，不允许重复定义
4. `const` 定义的常量，一旦定义就不允许更改它的值

const锁栈与锁堆原理

`const` 定义的常量，只锁栈，不锁堆

```
const a = 123;
const b = {
  userName: "张三"
};
```

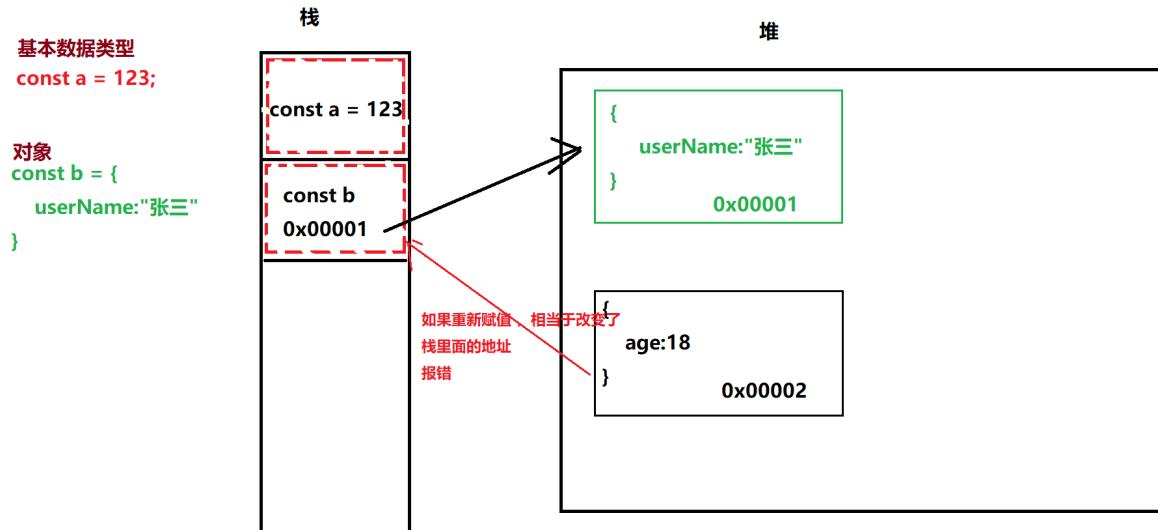


上面的图就是2个常量在内存当中的存储，我们可以看到，`const`仅仅只是把栈锁住了，堆没有锁住。现在我们就来看下面的代码是否正确

```

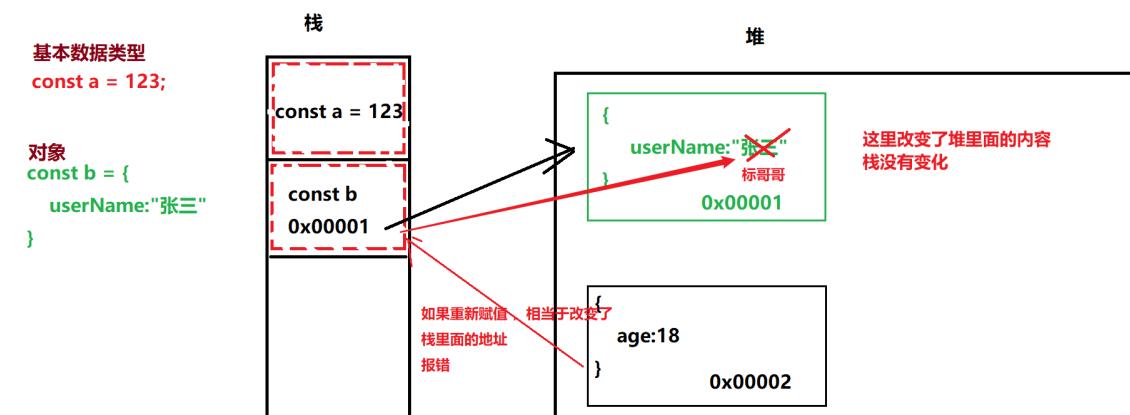
a = 456;           // 报错
b = {
  age:18
}    // 直接报错，因为修改了栈里面的内容

```



现在我们再考虑另外一个问题

```
b.userName = "标哥哥";           // 这么操作就不会有问题，因为栈没有变化，只改变了堆的内容
```



扩展：大多数的面试情况下，面试题目里面都会有问到 `var, let, const` 三者我区别

1. `var` 没有块级作用域，而 `let, const` 是有块级作用域的
2. `var` 有建立阶段，可以在定义之间来调用变量，而 `let, const` 没有建立阶段，不能在定义之间使用，应该先定义，后使用
3. `var` 可以重复定义变量，而 `let, const` 不可以重复定义
4. `var, let` 是定义变量，后面可以更改变量的值，而 `const` 定义的是常量，定义好以后，栈里的空间是不可以更改的

let与const的闭包特性

现在先看下面的现象

```
<body>
    <button type="button" class="btn">按钮0</button>
    <button type="button" class="btn">按钮1</button>
    <button type="button" class="btn">按钮2</button>
    <button type="button" class="btn">按钮3</button>
    <button type="button" class="btn">按钮4</button>
</body>
<script>
    var btns = document.querySelectorAll(".btn");
    for (var i = 0; i < btns.length; i++) {
        btns[i].addEventListener("click", function(event){
            console.log(i);
        });
    }
</script>
```

上面的代码是把所有的按钮都绑定了一个click事件，然后点击 click 事件以后执行 `console.log(i)`，最后就想问一下大家，打印的结果是什么？

按照我们最初的理解，应该是第一个按钮打印0，最后一个打印4，但是结果确是每一个都打印5，这是什么原因呢？

分析问题：当循环进行的时候，`i`是从0~4在循环，所以会把所有的按钮都遍历一遍，然后在每个按钮上面添加一个绑定事件，事件的代码就是 `console.log(i)`，关键点就在这个 `i`。当循环结束，事件绑定也结束

当用户去点击按钮的时候，才会触发 click 事件，而 click 事件就会去调用方法执行 `console.log(i)`，但是这个时候时候 `i` 在哪里？这里的`i`就是 `var i` 定义的变量，它没有区域性

ES5闭包解决

```
var btns = document.querySelectorAll(".btn");
var i = 0;
for (; i < btns.length; i++) {
    btns[i].addEventListener("click", (function(j){
        return function(event){
            console.log(j)
        }
    })(i))
}
```

ES6里面的let解决

```
var btns = document.querySelectorAll(".btn");
for (let i = 0; i < btns.length; i++) {
    btns[i].addEventListener("click", function(event){
        console.log(i);
    });
}
```

分析：因为 `let` 定义的变量是有区域性的，所以每次进入到一个{}就算进了个区域，上面的代码相当于循环了5次，定义了5个`let i`

```
{  
let i=0;  
}  
{  
let i=1;  
}  
{  
let i=2;  
}  
  
//依次类推，每个花括号都定义了一个i，因为每个花括号都是作用域，都不影响 外边
```

解构

解构与ES6里面的一个特色，它是一种特殊的取值与赋值方法，并关键字 `var`, `let`, `const` 没有任何关系

解构取值

1. 针对数组的解构取值

```
let arr = ["标哥", "海丽"];  
  
// 我想把数组里面的2个元素取出来  
/*  
let a = arr[0];  
let b = arr[1];  
*/  
  
let [a,b] = arr;  
  
console.log(a);  
console.log(b);
```

解构取值的时候也是一一对应的，请看下面代码

```
let arr = ["标哥", "海丽"];  
let [a,b,c] = arr;  
  
console.log(a);  
console.log(b);  
console.log(c); //当我们解构的时候如果发现没有这个值，那就是undefined
```

对于复杂的数组我们可以实现深度的解构

```
let arr = ["标哥", ["曹方", "曹慧"]];  
//要求解构取值a,b,c三个值对应数组里面的三个人名  
// let [a,temp] = arr;  
// let [b,c] = temp;  
  
// console.log(a);  
// console.log(temp);  
// console.log(b,c);  
  
let [a,[b,c]] = arr;  
console.log(a,b,c);
```

同时，解构还可以快速的交换变量

```
/**  
 * 解构妙用  
 */  
  
let a = 3;  
let b = 4;  
// 请同学们使用你们的方法，来交换变量的值  
  
//第三种方式  
// let arr = [a, b];  
// [b, a] = arr;  
[a, b] = [b, a];  
console.log(a,b);  
  
//第一种方式  
// let c = a;  
// a = b;  
// b = c;  
// console.log(a,b);  
  
// 第二种方式  
// a = a + b;           //a:7  
// b = a - b;           //a:7,b:3  
// a = a - b;           //a:4,b:3
```

2. 针对对象解构取值

```
let obj = {  
    userName: "标哥哥",  
    age: 18  
}  
//现在希望拿到2个属性值  
let userName = obj.userName;  
let age = obj.age;  
  
let {userName, age} = obj;  
console.log(userName, age);
```

在解构对象的时候有一个注意事项,如无特殊必要，不要去解构方法

```
let obj = {  
    userName: "标哥哥",  
    age: 18,  
    sayHello:function(){  
        console.log("大家好，我叫" + this.userName);  
    }  
}  
  
let {userName, age, sayHello} = obj;  
  
sayHello();      //在解构的过程当中不要解构方法，因为this的指向会发生变化，除非你明确的知道  
this指向了谁
```

对于复杂对象的解构，我们也是可以直接来进行的，如下所示

```
let stuInfo = {
  stuName: "张三",
  age: 18,
  telephone: {
    price: 1999,
    brand: "小米"
  }
}

// 请解构出 stuName, age, price, brand 四个属性

// let {stuName, age, telephone} = stuInfo;
// let {price, brand} = telephone;
let {
  stuName,
  age,
  telephone: {
    price,
    brand
  }
} = stuInfo;

console.log(stuName, age, price, brand);
```

解构赋值

解构的赋值其实质就是解构取值的反向操作，它是涉及到对象里面，数组不存在解构赋值

```
let userName = "标哥";
let age = 18;
// 我们现在希望把这2个变量封装成一个对象叫userInfo

let userInfo = {
  userName: userName,
  age: age
};
```

在上面的代码里面，我们将2个变量封装成了一个对象，这个时候我们发现，它的属性名与属性值相同的时候，我们就可以简化成下面的操作

```
let userInfo = {
  userName,
  age
}
console.log(userInfo);
```

展开运算符

它是ES6里面新出的一种运算符，并不是所有的东西都可以使用这个运算符，只有实现了 `Iterable` 接口的才可以使用展开运算符，在目前的系统里面，实现了 `Iterbale` 的数据类型有以下几种

1. 数组
2. `NodeList`

3. `HTMLCollection`
4. `Set` 单值集合
5. `Map` 键值对集合
6. `arguments` 实参数组

仔细看：展开运算符到底是什么，什么又是 `iterable` 接口

所有实现了 `Iterable` 接口的才有展开运算符，那么所有实现了 `Iterbale` 接口的都会有一个下面的方法

```
► unshift: f unshift()
► values: f values()
► Symbol(Symbol.iterator): f values()
► Symbol(Symbol.unscopables): {at: true, copyWithin: true, entries: true, fill: true, find: true, ...}
► [[Prototype]]: Object
```

如果发现一个对象上面有 `Symbol.iterator` 这个方法，就说明这个对象实现了 `Iterable` 的接口，所以它就可以使用迭代器，也就可以使用展开运算符

展开运算符在ES6里面使用 `...` 来表示，它会将一个集合展开。如下代码所示

```
var arr = ["张三", "李四", "王五"];
console.log(arr);
console.log(...arr);
console.log(arr[0], arr[1], arr[2]);
```

▶ (3) ['张三', '李四', '王五']

张三 李四 王五

张三 李四 王五

在上面的效果图里在我们可以看到 `... arr` 就相当于下面的 `arr[0], arr[1], arr[2]` 这个操作，把里面的每个值都展开了

```
let arr = [1, 8, 2, 3, 9, 7, 4, 6];

//求上面数组的最大值
// let max =Math.max(1, 8, 2, 3, 9, 7, 4, 6);
// console.log(max);

// 后面我们学了apply的方法
// let max = Math.max.apply(Math,arr);
// console.log(max);

let max = Math.max(...arr);
console.log(max);
```

上面的代码就是最典型的展开运算符的使用

展开运算符实现数组的拷贝

```
let arr1 = ["张三", "李四", "王五"];
// 希望拷贝一个数组arr2,两个数组互不影响

// let arr2 = arr1.slice();
// let arr3 = arr1.concat();

// console.log(arr1);
// console.log(arr2);

let arr2 = [...arr1];

console.log(arr2);
console.log(arr1 === arr2);           // false
```

展开运算符将类数组转换成数组

```
let liList = document.querySelectorAll(".ul1>li");
// liList可以使用展开运算符
// 如果要把liList转换成数组
let liArr = Array.prototype.slice.call(liList);

let liArr2 = [...liList];           // 现在我们就使用展开运算符来完成操作
```

使用展开运算符来合并数组

```
let arr1 = ["张三", "李四"];
let arr2 = ["a", "b"];
let arr3 = ["标哥", "桃哥", "飞哥"];

// 希望将上面的三个数组合并成一个新的数组
let arr4 = arr1.concat(arr2).concat(arr3);

// 使用展开运算符
let arr5 = [...arr1, ...arr2, ...arr3];
```

使用展开运算符实现对象的拷贝

```
let obj1 = {
  userName: "张三",
  age: 18
}
// 想得到一个obj2,与obj1相同,但互不影响
let obj2 = {};
Object.assign(obj2, obj1);

// console.log(...obj1);           // 报错

// ES6里面提供了一种特殊的场景
let obj3 = {
  ...obj1
};
```

使用展开运算符实现对象的合并

```
let obj1 = {
  userName: "张三",
```

```
age:18
}

let obj2 = {
  sex:"男",
  address:"湖北武汉",
  userName:"李四"
}

// 现在希望将2个对象合并

let obj3 = {
  ...obj2,
  ...obj1
};
```

如果属性出现了重复，后面的就是覆盖前面的

指数运算符

它是ES6当是针对数字类型或 `bigint` 类型进行运算的,指数运算就是 `**` 的运算

```
//假设，我们现在相求2的3次方，怎么办
// 求2的3次方
var a = 2 * 2 * 2;
var b = Math.pow(2,3);
var c = 2 ** 3;
console.log(a, b,c);
```

思考：已经有了第2个方式，为什么还要第3种方式？

关于BigInt数据类型

```
let a = Math.pow(2, 53);
console.log(a);          //9007199254740992

let b = Math.pow(2, 53) + 1;
console.log(b);          //9007199254740992
```

在上面的代码里面我们可以看到，a,b的值竟然是一样的，这是因为JS只能够进行32位的运算，如果超过了位数，计算的值就不准确。为了弥补这个的缺陷，ES6新推出了一个数据类型叫 `BigInt`

```

let a = 2172141653n;
let b = 15346349309n;
console.log(typeof a);           //bigint 这是一个新的数据类型，它的存储范围比number还要大

let a1 = 2172141653;
let b1 = 15346349309;

let c = a * b;
let c1 = a1 * b1;

console.log(c);                //33334444555566667777n
console.log(c1);               //333344445555666670000

```

所以我们可以看清楚的看到，如果使用 `number` 运算，超过了一个数值范围就计算不准备，这个时候为了保证数据的准备确，我们就要使用 `bigint` 的数据类型

`bitint` 数据类型的定义非常简单，只需要在原来的数据后面加上 `n` 就可以了

现在再回到刚刚的问题，为什么有了 `Math.pow` 以后还需要指数运算符

```

var a = Math.pow(2, 53);          //这个值不准确

// var b= Math.pow(2n, 57n);      //这里会报错，因为Math.pow()只接收nubmer类型

var c = 2n ** 57n;
console.log(c);                  //144115188075855872n

```

BigInt数据类型转换

其它类型转换为bigint数据类型

```

/**
 * bigint数据类型转换
 * 1.不能对小数进行转换
 * 2.除此之外所有的规则遵守number的转换规则，在结果上面加n
 * 3.不能是null, undefined,NaN
 */
console.log(BigInt(123));        //123n
console.log(BigInt("123"));      //123n
// console.log(BigInt("aaa"));    //报错
// console.log(BigInt(null));    //报错
// console.log(BigInt(undefined)); //报错
// console.log(BigInt(NaN));     //报错
console.log(BigInt(false));      //0n
console.log(BigInt(""));         //0n
console.log(BigInt(1.23));       //报错

```

bigint转其它数据类型

```

/**
 * bigint转其它数据类型
 */
let a = 1n;
console.log(String(a));
console.log(Boolean(a));
console.log(Number(a));

```

字符串的扩展

模板字符串

模板字符串是一个非常特殊的字符串，它使用反引号来表示

```
let a = `我是一个模板字符串`;
//模板字符串就是你怎么写，我就怎么展示
let b = `大家好，我叫标哥哥！
        爱你们哟!!!`;
```

模板字符串就是你书写的时候是什么样子，我就展示成什么样式！

```
let msg = `
亲爱的XXX:
你好！
恭喜你，获取了标哥哥抽取的大礼包，价值9999元等你来拿！
现在只需要交纳1999元的保证金，幸运大礼带回家，具体详情可以咨询我们的客服人员
电话：110
联系人：杨先生

`;
console.log(msg);
```

xxx公司
2022年9月27日

亲爱的XXX：
你好！
恭喜你，获取了标哥哥抽取的大礼包，价值9999元等你来拿！
现在只需要交纳1999元的保证金，幸运大礼带回家，具体详情可以咨询我们的客服人员
电话：110
联系人：杨先生

xxx公司
2022年9月27日

怎么书写，就怎么得到结果，这样非常方便

模板字符串最大的优点还在于字符串的拼接

在以前的时候，如果我们要拼接一个字符串，非常麻烦，如下所示

```
let userName = "张珊";
let age = 18;

let str1 = "你好啊，我叫" + userName + "，我的年龄是" + age;
let str2 = "你好啊，我叫".concat(userName).concat("，我的年龄是").concat(age);

//模板字符串的拼接
let str3 = `你好啊，我叫${userName}，我的年龄是${age}`;

console.log(str1);
console.log(str2);
console.log(str3);
```

如果在模板字符串的内部要嵌入 js 代码，直接使用 `{}$` 即可

```
let nickName = "江海丽";
let tel = "027-110";

let msg = `亲爱的${nickName}：
你好！
恭喜你，获取了标哥抽取的大礼包，价值${~~(Math.random() * 10000)}元等你来拿！
现在只需要交纳1999元的保证金，幸运大礼带回家，具体详情可以咨询我们的客服人员
电话：${tel}
联系人：杨先生

xxx公司
${new Date().toLocaleDateString()}

`;
console.log(msg);
```

模板字符串的使用注意事项

```
alert("你好");
alert`你好`;

function abc(){
    console.log("你好");
}

abc();
abc`;
//模板字符串当成花括号来使用了
```

当模板字符串使用花括号去执行的时候，如果里面还有参数呢？

```
function abc(){
    console.log(arguments)
}

let userName = "标哥";
let age = 18;
let sex = "男";

abc`我叫${userName}我的年龄是${age}我的性别是${sex}`;
```

```
[Running] node "d:\杨柳的工作文件\班级数字笔记\H2204\0927\code\0927\0927.js"
[Arguments] {
    '0': [ '我叫', '我的年龄是', '我的性别是', '' ],
    '1': '标哥',
    '2': 18,
    '3': '男'
}
2      console.log(arguments)
3  }
4
5  let userName = "标哥";
```

字符串的扩展方法

1. `includes()` 判断某个字符串当中是否包含某个字符，以前如果要判断使用的方法是 `index0f()`/`lastIndex0f()` 的方法
2. `repeat()` 生成重复的字怎么会串

```
var str = "a";
var str2 = str.repeat(10); // "aaaaaaaaaa";
```

3. `trimStart()` 去除字符串开始的空格
4. `trimEnd()` 去除字符串结束的空格

数组的扩展

在之前学习ES5的时候，我们已经学到了很多数组的方法，现在ES6又扩展出了一些新的方法

数组的扩展方法

1. `Array.of()` 方法

它是一个定义数组的方法，解决了之前 `new Array()` 存在的一些问题

```
let arr1 = new Array(5); //这代表数组的长度为5

let arr2 = Array.of(5); //这代表数组有一个元素，里面的第一个元素是5
// 相当于
let arr3 = [5];
```

这一种 `Array.of` 的写法等价于 `[]` 的写法，所以它也可以同时放多个值

```
let arr4 = ["a", "b", "c"];
let arr5 = Array.of("a", "b", "c");
//上面的2种定义方式是一样的
```

2. `Array.from()` 将一个类数组转换成数组

```
let liList = document.querySelectorAll(".ul1>li");
// 第一种：转数组
let arr1 = Array.prototype.slice.call(liList);
// 第二种：展开运算符
let arr2 = [...liList];
// 第三种：Array.from
let arr3 = Array.from(liList);
```

看到上面的代码以后，肯定有很多同学会觉得上面的方式是多余

```
let obj = {
  0:"张三",
  1:"李四",
  2:"王五",
  length:3
};

// 上面是一个类数组，转换成数组
```

```

// 上面是一个对象，它不具备Iterable的接口，所以不能使用展开运算符，这个时候就只能使用原始方式或最新的方法

// 第一种：原始方法
let arr1 = Array.prototype.slice.call(obj);

// 第二种：ES6里面的Array.from()
let arr2 = Array.from(obj);

console.log(arr1);
console.log(arr2);

```

3. `Array.prototype.fill(value:any,start?:number,end?:number)` 填充数组

有些时候我们希望将数组中的元素填充为一个统一的值，就可以使用这个方法，`value` 代表要填充的值，`start` 代表开始填充的位置，`end` 代表结束填充的位置（不包含这个位置）

```

let arr1 = new Array(10);
// Array.prototype.fill就可以完成
// 我们现在怎么调用这个fill方法
// arr1.__proto__ === Array.prototype;
// arr1.__proto__.fill === Array.prototype.fill;
arr1.fill("标哥");

let arr2 = new Array(10);
arr2.fill("江海丽", 0, 5);

let arr3 = new Array(10);
arr3.fill("袁池康", 3, 7)
// console.log(arr3);

let arr4 = new Array(10);
arr4.fill("曹方", 3);
console.log(arr4);

```

4. `Array.prototype.flat(steps)` 拍平一个二维数组

里面的参数 `steps` 代表要拍到几维，如果填 `Infinity` 代表直接拍成一维数组

```

/**
 * 数组的flat方法
 */

let arr1 = [
    "张三", "李四", [
        "王五", [
            "赵六", "田七", ["周八"]
        ]
    ]
];

let arr2 = arr1.flat(Infinity);
console.log(arr2);

```

5. `Array.prototype.find()` 方法

在数组当中查询某一项的元素，这个方法就冕为了弥补 `indexOf()`/`lastIndexOf()` 无法去查找对象的问题

```
/**  
 * 数组find的问题  
 */  
  
let arr1 = [  
  {  
    userName: "张三",  
    age: 18  
  }, {  
    userName: "李四",  
    age: 20  
  }, {  
    userName: "王五",  
    age: 22  
  }, {  
    userName: "李四",  
    age: 50  
  }  
]  
// 要求：查找userName为李四的对象，取得这个对象  
  
// 原始的方法  
let obj = null;  
for (let i = 0; i < arr1.length; i++) {  
  if (arr1[i].userName === "李四") {  
    obj = arr1[i];  
    break;  
  }  
}  
console.log(obj);
```

上面的方法是以前的旧方法

现在在ES6里面就多了一个新的方法

```
var obj = arr1.find(function(item, index, _arr){  
  return item.userName === "李四";  
});  
console.log(obj);
```

find是找到这个元素就把这个元素，如果里面有多个符合要求，只返回第一次找到的，如果找不到就返回 `undefined`

提起这个方法，很多人就会联想到之前ES5里面所学习的 `filter` 方法

```
let result = arr1.filter(function (item, index, _arr) {  
  return item.userName === "李四";  
});  
console.log(result);
```

filter方法返回的一定是一个数组，如果找到符合要求的元素，把这个元素放在数组里面，如果找不到这个元素，就是一个空的数组

6. `Array.prototype.findIndex()` 找到某一个元素的索引，这一个就去和上面的 `find` 方向非常相似

```
let arr1 = [
  {
    userName: "张三",
    age: 18
  }, {
    userName: "李四",
    age: 20
  }, {
    userName: "王五",
    age: 22
  },
  {
    userName: "李四",
    age: 50
  }
]
//请找出userName为王五的索引

let index = arr1.findIndex(function(item, index, _arr){
  return item.userName === "王五";
});

console.log(index);
```

for...of遍历

of的遍历与之前的in的遍历是相似的，只是 `for ... in` 采用的是有序遍历（当遍历的时候需要依靠索引或属性名的时候就是有有序遍历），`for ... of` 称之为无序遍历（也就是遍历的时候不需要依靠属性名）

for...of就是基于属性值的遍历，但是并不是所有的对象都可以进行for...of的遍历，只有实现了 `Iterable` 接口的才可以进行

简而言之，能够使用展开运算符的就可以使用for...of

```
/** 
 * for ... of
 */

let arr = ["a", "b", "c", "d", "e"];
// 基于属性名的遍历，有序遍历
for(let i in arr){
  // console.log(i);
  // for ... in是基于属性名的遍历，所以i就是遍历出来的属性名，也就是数组的索引
  console.log(arr[i]);
}

console.log("_____");

// 基于属性值的遍历 无序遍历 for ... of
for(let i of arr){
```

```
        console.log(i);
        // 基于属性值的遍历，所以i就是每一个属性值
    }
```

Set单值集合

Set是ES6里面新出的一种数据结构，它是一种单值集合，没有属性名（没有key,没有索引），存入不重复的值的集合

Set的创建

```
let s1 = new Set();           //创建了一个空的Set集合
let s2 = new Set(["a", "b", "c", "d"]);
let s3 = new Set(["a", "b", "c", "b"]);           //不能重复
```

Set里面的内容是不能重复的，重复的值是加不进去的

```
let arr1 = ["a", "b", "c", 1, 3, 5, 7, 8, 3, 4, 7, 9, 0, "b"];
// 下面就是去重以后的数组
let arr2 = Array.from(new Set(arr1));
```

上面的代码就是利用Set去重

Set相关的方法

1. `add()` 向Set集合当中添加新的元素

```
let s1 = new Set(["a", "b"]);
s1.add("标哥");
s1.add("袁池康").add("曹方");
s1.add("b")           //不能添加重复的元素，所以b不能添加进去
console.log(s1);
```

2. `delete()` 向Set集合中删除一个元素，返回一个布尔值代表删除的结果

```
var result = s1.delete("a");
//result就是删除的结果，如果为true就代表删除成功，如果为false就代表删除失败
```

3. `has()` 判断集合当中是否存在某个元素，如果返回true就代表存在，如果返回false就代表不存在

```
let result1 = s1.has("标哥");
let result2 = s1.has("张三");
```

4. `size` 属性，表示Set集合当中元素的个数

5. `clear()` 方法，清空当前的Set集合

6. `values()` 它是一个生成器方法，返回一个Set集合当中的包含值的 `Iterator` 迭代器

7. `keys()` 它是一个生成器方法，本应该返回所有的键的，但是 `Set` 是单集合没有键 `key`，所以返回的内容与 `values` 一致

Set的遍历

因为Set是单值集合，它没有属性名，只有属性值，所以我们不可以使用原始的 `for ... in` 来解决

Set是一个单值集合，它有属性值，同时又实现了 `Iterable` 的接口，具备一个迭代器，所以我们可以使用 `for ... of` 来遍历

```
let s1 = new Set(["张三", "李四"]);
s1.add("王五").add("赵六").add("田七");
//怎么遍历

// 有序遍历，基于属性名或索引
// for(let i in s1){
//   console.log(i);
// }

//无序遍历，只在有属性值就可以了，然后还有迭代器接口就可以了
for(let i of s1){
  console.log(i);
}
```

Map键值对集合

Map是ES6里面新出的一种数据结构，它对应的是ES5里面的对象，它是采用键值对的形式存储，弥补了对象当中属性名只能使用数字与字母的缺点

Map也结合了上面的Set的特点，它可以实现Set的存储优点（存的快），也解决了Set的缺点（获取速度慢）

Map采用键值对存储，它的键不允许重复，值可以重复

Map的创建

```
let m1 = new Map();

let m2 = new Map([
  ["a", "袁池康"],
  ["b", "江海丽"],
  ["c", "曹方"],
  ["d", "曹慧"]
]);

// 键不可以重复，值可以重复
let m3 = new Map([
  ["a", "袁池康"],
  ["b", "江海丽"],
  ["c", "曹方"],
  ["d", "曹慧"],
  ["a", "标哥"]
]);

let m4 = new Map([
  ["a", "袁池康"],
  ["b", "江海丽"],
```

```
[ "c", "曹方" ],  
[ "d", "曹慧" ],  
[ "e", "袁池康" ]  
]);  
console.log(m4);
```

map的键与值可以是任何数据类型

```
let m1 = new Map([  
    [123, "hello"],  
    [true, { userName: "张三" }],  
    [null, "你好"],  
    [456, function () { console.log("hello") }]  
]);  
console.log(m1);
```

注意：

1. 键不可以重复，值可以重复
2. 键与值可以是任何数据类型

Map相关的方法

1. **size** 属性，用来描述当前的Map集合里面有多少个元素
2. **set(k,v)** 方法，向Map集合当中添加一对元素

```
let m1 = new Map();  
  
m1.set("a", "袁池康");  
m1.set("b", "曹方").set("c", "曹慧");  
m1.set("a", "曹操");  
m1.set("d", "曹方");  
  
console.log(m1);
```

3. **get(k)** 方法，通过键获取Map集合当中的值，如果获取不到就是undefined

```
m1.get("b");
```

4. **delete(k)** 方法，通过一个键删除Map中的某一项，返回一个boolean的结果，如果为true代表删除成功，如果为false代表删除失败

```
let result = m1.delete("c");
```

5. **clear()** 方法，清空Map集合
6. **keys()** 方法，返回当前Map集合中的key的迭代器
7. **values()** 方法，返回Map集合当中value的迭代器
8. **entries()** 返回当前Map中的key-value的键值对迭代器

Map的遍历

1. Map 实现了迭代器
2. Map 实现的是散列存储，没有索引这个说法

```
let m1 = new Map();

m1.set("a", "袁池康");
m1.set("b", "曹方").set("c", "曹慧");

//因为map使用的散列的无序存储，所以不能使用有序的方式来进行遍历
// for(let i in m1){
//   console.log(i);
// }

// for ... of的前提 无序遍历

// for(let item of m1){
//   console.log(item[0],item[1]);
// }

//在遍历的时候直接解构
for(let [k,v] of m1){
  console.log(k,v);
}
```

ECMAScript 6-2

接上文

WeakSet与WeakMap

这个东西与之前所学习的Map以及Set非常相似，只是有一些细微的区别，它们叫弱Map与弱Set

```
let s1 = new Set();
s1.add("张三").add(123).add(true).add(null);
console.log(s1);

let s2 = new WeakSet();
s2.add({userName:"张三"})
s2.add(["12313"]);
s2.add(new Date());
s2.add(123);           //报错
console.log(s2);
```

代码分析：

s1是一个普通的Set集合，所以可以添加任何值到Set集合当中去，但是WeakSet是一个弱Set，它只允许添加引用类型（对象）进去，基本数据类型是不能添加进去的

```
let m1 = new Map();

m1.set("a", "张三").set("b", "李四");
console.log(m1);

let m2 = new WeakMap();
m2.set(["a"], "张三");
m2.set("b", "李四");           //报错
console.log(m2);
```

代码分析：

m1是一个正常的Map，所以它的键可以是任何类型

这一个报错的原因上面的原因是一样的，WeakMap的键也只能是引用类型(对象),不能是基本数据类型

注意：WeakSet与WeakMap没有实现 Iterable 接口，所以不能使用for...of遍历，也不能使用展开运算符

Symbol数据类型

Symbol是Es6当中推出的一种新的数据类型，它的全称叫标识类型，通常叫做**全局唯一标识符**，这个东西对应的后端编程语言 **UUID** 或 **GUID**

```
//4c6b0921-3e3a-11ed-84ee-34298f73191b
let s1 = Symbol();

//aacb0921-3e3a-11ed-84ee-34298f73191b
let s2 = Symbol();

typeof s1;
typeof s2;

console.log(s1 === s2);    //false
```

Symbol数据类型的创建是要调用Symbol()来创建的，它每次创建的标识符都是不一样的，所以上面的s1与s2就不相等

如果想得到相同的 **Symbol** 只能通过下面的方法来进行

```
//4c6b0921-3e3a-11ed-84ee-34298f73191b
let s1 = Symbol();
let s2 = Symbol();
console.log(s1 === s2);      //false

//—————
let s3 = Symbol.for("标哥");
let s4 = Symbol.for("标哥");
console.log(s3 === s4);      //true
```

上面的代码我们可以这么理解，我们可以认为在“标哥”这个地方创建的标签是一样的

请注意下面的一个点

```
let s1 = Symbol.for("标哥");
let s2 = Symbol.for("标哥");
console.log(s1 === s2);           //true

// ----

let s3 = Symbol("袁池康");
let s4 = Symbol("袁池康");

console.log(s3 === s4);           //false
```

> s1

< Symbol(标哥)

> s2

< Symbol(标哥)

> s3

< Symbol(袁池康)

> s4

< Symbol(袁池康)

、

上面的情况应该怎么去理解呢？

`Symbol.for` 相当于从谁哪里获取标签，`Symbol.for("标哥")` 相当于从标哥那里得到了标签，所以 s1, s2 都是从标签那里得到的标签，所以它们就相同

`Symbol("袁池康")` 相当于把标签贴在了袁池康的身上，这个时候我不能保证贴在袁池康身上的标签是相同的

Symbol的应用点

Symbol的应用点就在于它的唯一性，不重复，它一点正好与我们之前所学习的Set很像，如果我们想利用Map去实现Set的操作，怎么办呢

```
let m1 = new Map();

m1.set(Symbol(), "张三");
m1.set(Symbol(), "李四");

let result1 = m1.get(Symbol());           //取不到。而我们之前讲Set的时候也说过，Set不能取值
console.log(result1);
```

上面的代码就是模拟了Set只能存，不能取的特点

除了上面的应用点，还有一个应用点就是使用Symbol做为对象的属性名

```
let obj1 = {
    userName: "张三",
    age: 18
}

let obj2 = {
    sex: "男",
    hobby: "看书",
    userName: "小四"
}

// 现在我想将这两个对象去合并，并保留所有的属性，怎么办呢

let obj3 = {
    ...obj1,
    ...obj2
}
```

因为上面的属性名在合并以后有重复，所以后面的 `userName` 就是覆盖前面的 `userName`

```
{ userName: '小四', age: 18, sex: '男', hobby: '看书' }
```

我们可以看到 `userName` 已经变成小四了，这说明属性冲突重复了

为了解决上面的问题，我们就会使用 `Symbol` 来做属性名，如下所示

```
let obj1 = {
    [Symbol("userName")]: "张三",
    age: 18
}

let obj2 = {
    sex: "男",
    hobby: "看书",
    [Symbol("userName")]: "小四"
}

// 现在我想将这两个对象去合并，并保留所有的属性，怎么办呢

let obj3 = {
    ...obj1,
    ...obj2
};

console.log(obj3);
```

最终的结果如下

```
{  
  age: 18,  
  sex: '男',  
  hobby: '看书',  
  [Symbol(userName)]: '张三',  
  [Symbol(userName)]: '小四'  
}
```

我们把不希望重复的属性名使用 `Symbol` 的数据类型去表示，这样就永远不会重复

Symbol做属性名

Symbol目前最大的优点就是用于做属性名，其实这个点我们在很早之前的ES5里面就接触到这个点，在之前讲面向对象的时候我们说过，如果想遍历一个对象的属性，我们应该用什么方法

1. `for....in`
2. `Object.keys()`
3. `Object.getOwnPropertyNames()`

当使用 `Symbol` 做属性名的时候，如果想遍历这个属性就很困难，上面的三个方法通通不适用

```
let obj = {  
  age: 18,  
  sex: '男',  
  hobby: '看书',  
  [Symbol("userName")]: '张三',  
  [Symbol("userName")]: '小四'  
}  
  
//不可以  
for(let i in obj){  
  console.log(i);  
}  
//不可以  
let keys = Object.keys(obj);  
console.log(keys);  
  
//不可以  
let names = Object.getOwnPropertyNames(obj);  
console.log(names);
```

如果想单独获取Symbol的属性名，只通过下面的方法来完成

```
let symbolNames = Object.getOwnPropertySymbols(obj);  
console.log(symbolNames);
```

最后一个点要注意，`JSON.stringify()` 在序列化对象的时候，不会操作 `Symbol` 的属性。如下所示

```
let obj2 = {
  age: 18,
  sex: '男',
  hobby: '看书',
  nickName: "张三",
  [Symbol("pwd")]: "123123"
}
let str2 = JSON.stringify(obj2);
console.log(str2);
```

结果如下,结果当中没有pwd这个属性

```
{"age":18,"sex":"男","hobby":"看书","nickName":"张三"}
```

生成器函数

生成器函数就是为了生成一个迭代器的，它的全称叫Generator Function

生成器函数是ES6里面新出一种函数类型，旨在解决迭代的问题

普通的函数如下所示

```
function abc(){
  return 123;
}

let x = abc();
console.log(x);
```

在上面的代码里面，我们可以看到，我们的函数如果要返回一个值到外边，就只能通过 `return`，并且只能返回一次，因为函数内部碰到`return`就结束了

试想一下：如果我想一个函数可以返回多次的值，怎么办呢？

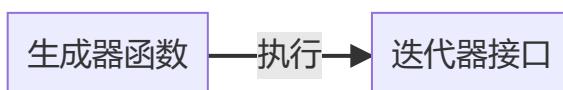
定义生成器函数

```
function* def(){

}
```

生成器函数在定义的时候在 `function` 关键字的后面添加一个 `*` 就可以了，生成函数的内部是可以进行多次返回的，我们把这个返回的过程称之为迭代的过程

生成器的函数调以后，并不会立即执行代码体，它会返回一个迭代器对象



在生成器函数的内部，如果想要多次返回，我们要使用关键字 `yield` 来进行

```
function* def(){
  // 在它的内部是可以多次返回的
  yield "a";
  yield "b";
```

```

    yield "c";
    return "曹方";
}

let x = def();           //x就是def函数执行以后返回的迭代器

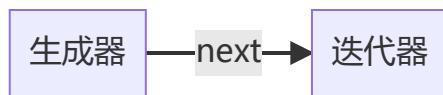
x.next();                //{value: 'a', done: false}
x.next();                //{value: 'b', done: false}
x.next();                //{value: 'c', done: false}
x.next();                //{value: '曹方', done: true}

```

在上面的代码当中，`yield`代表的是返回，它返回了 `a, b, c` 三个东西，最后又返回了“曹方”

生成器函数运行

我们上面可以的生成器函数在执行以后最终会返回一个迭代器，迭代器的内部是一个 `next()` 方法可以让程序运行到 `yield` 的地方拿到返回值然后暂停，直到继续 `next()` 进行下一步操作



```

function* def() {
    let k1 = yield "a";
    console.log(k1);          //李四
    let k2 = yield "b";
    console.log(k2);          //王五
    let k3 = yield "c";
    console.log(k3);          //赵六
    return "曹方";
}

let x = def();
let a1 = x.next("张三"); // {value: 'a', done: false}
let a2 = x.next("李四"); //
let a3 = x.next("王五"); //
let a4 = x.next("赵六"); //

```

我们可以看到生成器函数执行以后，张三没有打印，这是为什么呢

1. 遇到`yield`就暂停
2. 这一次`next`的参数会做为上一次`yield`的返回值接收

生成器函数调用另一个生成器函数

```

function* a(){
    yield "张三";
    yield "李四";
    return "标哥";
}

function* b(){
    yield "王五";
    // 调用上面的一个生成器函数，怎么办？

```

```
yield* a();
yield "赵六";
return "桃子";
}

let x = b();           //生成器函数返回了一个迭代器

let a1 = x.next();
console.log(a1);

let a2 = x.next();
console.log(a2);

let a3 = x.next();
console.log(a3);

let a4 = x.next();
console.log(a4);

let a5 = x.next();
console.log(a5);
```

如果在生成器函数里面要调用另一个生成器函数，需要使用 `yield*` 函数名() 来完成

迭代器

迭代器也叫迭代器对象，它的内部有一些方法和状态，我们可以通过生成器函数来得到迭代器，也可以手动创建

1. 每个迭代器最初的状态都是suspended暂停，同时每一个迭代器里面都有一个方法叫 `next()`，如果想让程序继续运行，我们要调用这个方法，程序会一直运行，直到遇到了 `yield` 停下来
2. `yield`返回的是一个对象，其中 `value` 代表返回的值，`done` 代表这个迭代器是否执行完成了，如果是 `false` 代表迭代器现在还没有完成，可以继续向下向执行

迭代器的状态

1. `suspended` 暂停状态
2. `running` 运行状态
3. `closed` 关闭状态，说明迭代已经完成了

迭代器遍历

生成器函数可以生成迭代器，迭代器的遍历之前就已经了解过，它是通过 `for ... of` 遍历的，所以可以看到下面的代码运行是成功的

```
function* a(){
  yield "张三";
  yield "李四";
  yield "王五";
  yield "赵六";
}
```

```
let x = a();           //x是生成器函数a生成的一个迭代器

//x.next();           //{value:"",done:false}

// let temp = {};
// while((temp = x.next()).done=false){
//   console.log(temp.value);
// }

//真正的迭代器是可以通过for ... of来遍历的
for(let item of x){
  console.log(item);
}
```

迭代器接口

在讲这个东西之前，我们先学2个单词

1. **Iterable**，可迭代的，代表一种能力
2. **Iterator**，迭代器

迭代就是把东西一个一个的拿出来，只能顺着拿（正向的拿），一旦结束了迭代了就不可以重新开始

迭代器是可以通过生成器函数得到？那什么是迭代器接口呢？

在系统当中，有一些对象它不是迭代器，但是它又可以实现遍历，还可以实现 **for ... of** 的遍历，如 **Array, Map, Set** 等一系列对象，这是为什么呢？

一个对象如果可以迭代，它要么是一个迭代器，要么实现了迭代器的接口

什么是接口？

接口也叫规范，只要实现了这个接口就具备这个接口的能力，所以我们只要去实现了迭代器的接口，就实现了迭代器的能力，迭代器的接口就叫 **Iterable**

主动去实现器的接口

```
let obj = {
  0: "张三",
  1: "标哥",
  2: "李四",
  3: "王五",
  4: "赵六",
  length: 5
}
//上面是我们自定义的一个对象，它是一个类数组
for(let item of obj){
  console.log(item);
}
```

当我们去执行上面的代码的时候，就会报错，如下图

```
for(let item of obj){  
    ^  
    9 //上面是我们自定义的一个对象，它是一个类数组  
    10 for(let item of obj){  
  
TypeError: obj is not iterable  
    at Object.<anonymous> (d:\杨标的互作文件\班级教学笔记\H2204\0928\code\09  
    .js:1:1)
```

通过上面的代码我们就可以发现，如果想使用 `for ... of` 遍历，那具备是具备可迭代的能力的。

之前我们就已经讲过，只要是实现了Iterable接口的，在它的内部就会有如下的方法

```
► unshift: f unshift()
► values: f values()
► Symbol(Symbol.iterator): f values()
► Symbol(Symbol.unscopables): {at: true, copyWithin: true, entries: true, fill: true, find: true, ...}
► [[Prototype]]: Object
```

```
/**  
 * 迭代器接口的实现  
 */  
  
let obj = {  
    0: "张三",  
    1: "标哥",  
    2: "李四",  
    3: "王五",  
    4: "赵六",  
    length: 5,  
    [Symbol.iterator]: function* () {  
        let index = 0;  
        while (index < this.length) {  
            yield this[index];  
            index++;  
        }  
    }  
}  
//上面是我们自定义的一个对象，它是一个类数组  
for (let item of obj) {  
    console.log(item);  
}
```

最后强调一点，`for...of`遍历的时候是可以使用 `break` 与 `continue` 来完成操作的

函数的扩展

在之前的ES5的学习里面，我们已经接触过了很多函数

1. 普通函数
 2. 构造函数
 3. 立即执行函数
 4. 函数表达式
 5. 回调函数
 6. 匿名函数
 7. 递归函数

现在在ES6当中对函数又做了一些扩展

无构造函数的函数

官方的说法叫成员函数，也叫属性函数

在之前的ES5里面，我们是通过 `function` 关键字来定义函数的，现在在ES6里面，ES希望尽量舍弃掉 `function`，因为通过 `function` 所定义的函数既可以通过普通函数去调用 `函数名+()` 调用，也可以当成构造函数去调用 `new 函数名()`

```
let obj = {
  userName: "张三",
  sayHello: function(){
    console.log(`大家好，我叫${this.userName}`);
  }
}

obj.sayHello();
new obj.sayHello;
```

在上面的代码当中我们可以看到，对象 `obj` 里面的函数 `sayHello` 可以当成一个普通的成员函数去调用，也可以当成构造函数 `new` 调用，这样就显得非常的不严谨

ES6为了解决这样的岐义，它直接改变了对象内部的函数的定义方式，如下所示

```
let obj2 = {
  userName: "李四",
  sayHello() {
    console.log(`大家好，我是第二个方法，我叫${this.userName}`);
  }
}

obj2.sayHello();           //正常调用
new obj2.sayHello;        //这样调用就会报错， obj2.sayHello is not a constructor
```

箭头函数

之前在ES5里面我们已经学习过了函数的定义是通过 `function` 关键字来完成的，现在我们要慢慢的放弃掉 `function` 的关键字的，所以我们需要换一种试试去定义函数

首先我们先来回顾一下之前是怎么义函数的

```
function sayHello(){
  console.log("大家好，我叫标哥");
}

var sayHello = function(){
  console.log("大家好，我叫标哥");
}
```

上面的2种方式对于同学们来说应该不陌生，在ES6里面对函数的定义做了新的规划

无参数的箭头函数

```
const sayHello2 = ()=>{
    console.log("大家好，我叫标哥");
}
```

有一个参数的箭头函数

```
const sayHello2 = (userName) => {
    console.log(`大家好，我叫${userName}`);
}
```

如果只有一个参数，还可以写成下面的样式

```
const sayHello3 = userName => {
    console.log(`大家好，我叫${userName}`);
}
```

有多个参数的箭头函数

```
function sayHello(userName,age){
    console.log(`大家好，我叫${userName}，我的年龄是${age}`);
}
//ES6的箭头函数
const sayHello2 = (userName, age) => {
    console.log(`大家好，我叫${userName}，我的年龄是${age}`);
}
```

如果有多个参数，则这个括号又不能省略

箭头函数的返回值

```
const a = () => {
    console.log("hello");
}

const b = () => console.log("hello");
```

代码分析

在上面的代码当中，两个函数执行了相同的代码，所以看起来是没有任何区别的，这两个函数体里面只有一行代码，所以省略花括号也是可以的，这两个函数是没有区别的

请看下面代码

```
const a = () => {
    return "张三";
}

//下面的箭头函数没有花括号，所以代表"李四"是一个返回值，它相当于`return "李四"`
const b = () => "李四";
```

如果一个箭头函数省略了花括号，则代表直接返回一个内容，上面的函数里面a,b两个函数的功能都是一样的

```

let arr = [1, 4, 6, 8, 2, 3, 7];
// 要求使用filter得到里面的偶数

//第一种:常规写法
let result = arr.filter(function (item) {
    return item % 2 === 0;
});

//第二种:箭头函数写法
let result2 = arr.filter(item => {
    return item % 2 === 0;
});

//第三种:箭头函数再简化返回值
let result3 = arr.filter(item => item % 2 === 0);
console.log(result3);

```

上面的写法就是箭头函数的写法，也是省略花括号的写法

箭头函数的注意事项

1. 箭头函数不具备构造函数的特点，不能使用 new 来调用
2. 箭头函数的内部没有 arguments
3. 箭头函数绑定的是外部的this,它的内部没有this指向（或者这么理解，箭头函数会跳过当前的作用域，去外边拿this）

```

let obj1 = {
    userName: "张三",
    sayHello() {
        console.log("我在外边打印的结果——", this.userName);
        var that = this;
        setTimeout(function() {
            //想在这里拿到张三,怎么办?
            // console.log(this) //指向了window
            console.log("我在里面打印——", that.userName);
        }, 2000);
    },
    sayHello2(){
        console.log("我在sayHello2的外部打印---",this.userName);
        setTimeout(()=>{
            // 箭头函数没有this,它拿的是外部的this
            console.log("我在sayHello2里面打印——",this.userName);
        },2000);
    }
}

```

同时请看下面的代码案例

```

4
5 let obj1 = {
6   userName:"张三",
7   sayHello(){
8     setTimeout(() => {
9       setTimeout(() => {
10        console.log(this.userName);
11      }, 2000);
12    }, 2000);
13  }
14}
15
16 obj1.sayHello();

```

在正向的场景下面是不允许使用箭头函数的

第一种场景：成员函数里面

```

var userName = "标哥哥";
let obj1 = {
  userName:"张三",
  // 下面的写法是完全不允许的
  sayHello:()=>{
    console.log(this.userName);
  }
}

obj1.sayHello();

```

第二种场景：事件绑定的回调函数

```

let btn1 = document.querySelector("#btn1");
btn1.addEventListener("click", event => {
  //DOM的事件绑定里面不要使用箭头函数
  console.log(this);
});

```

ES6函数的参数

之前学习函数都知道，函数在定义的时候是有参数的，那么，我们现在看下面的代码

```

//如果这个函数不传参数，默认值就是袁池康
const sayHello = (userName) => {
  console.log(`大家好，我叫${userName}`);
}

sayHello("标哥");
sayHello();

```

函数的默认参数【可选参数】

ES5里面的解决方法

```
//如果这个函数不传参数， 默认值就是袁池康
const sayHello = (userName) => {
    userName = userName || "袁池康"
    console.log(`大家好，我叫${userName}`);
}

sayHello("标哥");
sayHello();
```

在ES6里面，如果想设置一个函数的默认值，非常简单

```
const sayHello2 = (userName = "袁池康") => {
    console.log(`大家好，我叫${userName}`);
}

sayHello2("标哥哥");
sayHello2()
```

在上面的代码里面，我似乎可以看到 `userName` 在定义这个参数的时候给了一个默认值

注意事项， 默认参数只能放在最后

```
//如果这个函数不传参数， 默认值就是袁池康
const sayHello3 = (sex, userName = "袁池康") => {
    console.log(`大家好，我叫${userName}，我的性别是${sex}`);
}

sayHello3("男", "标哥哥");
sayHello3("女");
```

函数的剩余参数

在之前学习箭头函数的时候我们已经得到了一个结论，里面没有 `arguments`，那么，如果我们想使用 `arguments` 的功能，应该怎么办？

```
function getSum() {
    let sum = 0;
    for (let i = 0; i < arguments.length; i++) {
        sum += arguments[i];
    }
    return sum;
}

let x = getSum(11, 12, 14, 10);
console.log(x);
```

如果现在在ES6的箭头函数里面，怎么样去实现上面的功能呢？

现在没有 `arguments` 我们就必须想一个办法 把输入进去所有 数字全部接收到，怎么办呢？

```

let getSum2 = (... num) => {
    //没有arguments, 怎么办?
    // 这一个时候的num它就是一个数组了
    let sum = 0;
    for(let item of num){
        sum+=item;
    }
    return sum;
}
getSum2(11,12,13,10);

```

这个剩余参数的使用场景非常多，如下

```

const sayHello = (userName, hobby) => {
    console.log(`大家好, 我叫${userName}, 我的爱好是${hobby}`);
}

sayHello("杨标", "看书");

```

现在考虑一个场景，如果这个人要是有多个爱好，怎么办？

```

const sayHello2 = (userName, ... hobby) => {
    console.log(`大家好, 我叫${userName}, 我的爱好是${hobby.toString()}`);
}
sayHello2("张三", "看书", "睡觉");

```

函数的补充call/apply/bind

这个点本身应该是在ES5里面去讲的，当时因为时间关系，并且用得比较少，所以没有讲，现在补充进来

```

let obj1 = {
    userName: "张三",
    sayHello(){
        console.log(this.userName);
    }
}

let obj2 = {
    userName: "李四"
}

//呼叫谁过来调用自己,
obj1.sayHello.call(obj2);           //李四

//申请谁过来叫用自己
obj1.sayHello.apply(obj2);          //李四

//把sayHello的函数绑定在obj2上面
//它会生成一个新的方法，生成的这个新的方法this指向的就是你bind的对象
let aaa = obj1.sayHello.bind(obj2);
aaa();                            //李四

```

上面的那种 **bind** 的用法，在ES6里面其实已经很少使用了，因为它已经可以使用箭头函数来代替了

```
/**
```

```

* bind指代的就是箭头函数的特殊情况
*/

```

```

let obj1 = {
    userName: "张三",
    sayHello1() {
        console.log(this.userName);
        var that = this;
        setTimeout(function () {
            console.log("内部的——", that.userName);
        }, 2000);
    },
    sayHello2() {
        console.log(this.userName);
        setTimeout(() => {
            console.log("内部的——", this.userName);
        }, 2000);
    },
    sayHello3() {
        console.log(this.userName);
        /*
        function a(){
            console.log("内部的---",this.userName);
        }
        let b = a.bind(this);
        setTimeout(b,2000);
        */
    }
}

setTimeout(function () {
    console.log("内部的---", this.userName);
}.bind(this), 2000);
}

// obj1.sayHello1();
// obj1.sayHello2();

obj1.sayHello3();

```

面试点：

1. call/apply/bind 有什么区别?
2. 有哪些方法可以改变this指向
3. 简单的说明一下 var that=this 的情况

class关键字

在上面的函数的扩展里在，无论是成员函数还是箭头函数都不能已构造函数的形式 new 来调用，那么，我们样去创建构造函数呢

在ES6里在，新堆出了一个关键字叫 class ,如果想创建类似于ES5里面的构造函数，我们应该使用下面的方式来完成

认识关键字 `class`

`class`的关键在ES6里面是用来创建构造函数的，如下,我们先看ES5里面的代码

```
function Student(userName,sex){  
    this.userName = userName;  
    this.sex = sex;  
}  
  
let s1 = new Student("张三","男");      //得到s1的对象
```

代码分析：在上面的代码里面，我们的 `function` 定义了一个函数 `Student`，这是ES5里面的语法，这种定义方式是有歧义的，因为它既可以当成构造函数来调用，也可以当成普通函数来调用，这样就不严谨

在ES6里面，我们已经不在推荐使用 `function` 关键字来创建对象的原因就在这里，如果想创建普通函数可以使用箭头函数，如果想在对象里面创建函数直接使用属性函数（成员函数）的方式，如果想创建构造函数则使用关键字 `class`

```
class Student {  
  
}  
  
console.log(typeof Student);      //"function"  
let s1 = new Student();  
console.log(typeof s1);          //对象
```

上面的 `Student` 它就是构造函数，不能当成普通函数执行

如果我们直接通过 `Student()` 的方式去调用，就会报错，如下所示

```
Student()  
^  
  
TypeError: Class constructor Student cannot be invoked without 'new'
```

认识一下 `constructor`

每一个 `class` 的内部都会有一个 `constructor`，如果你不写系统会自动的给你创建一个，它的中文名称叫**构造器**，它的特点如下

1. 它是一个函数，这个名子不能更改
2. 如果不写，系统会自动给你创建一个，如果你写了，系统就不再给你创建
3. 你在 `new` 这一个 `class` 的一瞬间，这个 `constructor` 函数会自动调用
4. 它内部的`this`指向新对象

```
class Student{  
    constructor(){  
        console.log("标哥哥,好帅");  
    }  
}  
  
let s1 = new Student();
```

上面的代码会打印“标哥哥，好帅”。

现在我们再回到我们的本质点,怎么样在 `class` 里面接收参数

```
function Student(userName, sex){  
    this.userName = userName;  
    this.sex = sex;  
}
```

ES6的代码

```
class Student {  
    constructor(userName, sex) {  
        this.userName = userName;  
        this.sex = sex;  
    }  
}  
// new 一个class的时候,constructor自动执行  
  
let s1 = new Student("张三", "男");  
console.log(s1.userName, s1.sex);
```

class里面定义方法

在之前我们讲过, 对象里面是可以包含方法的, 现在对比如下

ES5的代码

```
function Student(userName, sex) {  
    this.userName = userName;  
    this.sex = sex;  
  
    this.sayHello = function () {  
        console.log(`大家好,我叫${this.userName}`);  
    }  
}
```

ES6的代码

```
class Student {  
    constructor(userName, sex) {  
        this.userName = userName;  
        this.sex = sex;  
    }  
    sayHello(){  
        console.log(`大家好,我叫${this.userName}`);  
    }  
}
```

class里面的get/set访问器

在之前的ES5的学习当中我们已经了解过 `get/set` 的访问器属性, 其实在ES6里面也可以, 在ES6里面它换了一种方式来进行

```
class Student {  
    constructor(firstName, lastName) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
    }  
}
```

```

    userName() {
        return this.firstName + this.lastName;
    }
}

let s1 = new Student("袁", "池康");

let name1 = s1.userName();
console.log(name1);

```

在上面的代码里面，即使在没有访问器属性的情况下，我们也可以实现取值，但是我们应该有要一个属性，而不是方法

ES6里面的方式

```

class Student {
    constructor(firstName, lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
    get userName() {
        return this.firstName + this.lastName;
    }
    set userName(v){
        console.log("你在赋值",v);
        //这个v代表就是赋的值
        this.firstName = v[0];
        this.lastName = v.slice(1);
    }
}

let s1 = new Student("袁", "池康");
s1.userName = "江海丽";           //在赋值的时候会自动触发 set方法
console.log(s1.userName);         //在取值的时候自动调用get方法

```

1. 在上面的代码里面，`get/set` 共同的构造了一个访问器属性
2. `get` 是在取值的时候自动调用，`set` 在赋值的时候自动调用

除了class里面使用get/set以外，普通的对象也是可以使用的

```

let obj = {
    firstName: "张",
    lastName: "三",
    get userName() {
        return this.firstName + this.lastName;
    },
    set userName(v) {
        console.log("你在赋值");
        this.firstName = v[0];
        this.lastName = v.slice(1);
    }
};

// console.log(obj.userName);
obj.userName = "江海丽";

```

class里面的static关键字

`static` 关键字并不是JS里面独有的，其它的编程语言里面也有，它主要是用于构建静态的东西。在ES6里在，语法如下

```
class Person{
    constructor(userName){
        this.userName = userName;
    }
    sayHello(){
        console.log(`我的名字叫${this.userName}`);
    }
    // 静态方法
    static sleep(){
        console.log(`这个人正在睡觉`);
        console.log(this.age);           //10,静态的方法可以调用静态的属性
    }
    // 静态属性
    static age = 10;
}

// 如果想使用普通的方法或普通的属性，我们要定要通过构造函数得到对象，用对象来调用
let p1 = new Person("张三");
console.log(p1.userName);
p1.sayHello();

// 静态的东西，只有一个特点，不需要new，直接通过构造函数来调用
Person.sleep();
console.log(Person.age);
```

代码分析：

1. 只需要在属性或方法的前面去添加 `static` 关键字就可以了
2. 静态方法或静态属性的调用不需要 `new`，直接通过构造函数来调用
3. 静态的里面不要调用非静态的东西，但是静态的方法可以调用静态的属性

extends关键字

在Es5里面，如果我们要实现对象的继承会非常的麻烦，最后的一步推断里面，我们使用组合继承与寄生继承来完成了我们的操作，但是这样做非常麻烦，在ES6里面有了更好的方式，就是使用 `extends` 关键字

无参数的继承

```
class Person {
    sleep() {
        console.log("我在睡觉");
    }
}

class Student extends Person {
    sayHello() {
```

```
        console.log("我是学生，爱学习的学生");
    }
}

let s1 = new Student();
s1.sayHello();
s1.sleep();
```

在上面的代码里面，我们让Student继承了 Person，同时这两个class我们都没有写构造函数 constructor，我们让系统自动生成了

如果我想自己手动来写constructor，不要系统自动生成，怎么办呢

```
class Person {
    constructor(){
        }

    sleep() {
        console.log("我在睡觉");
    }
}

class Student extends Person {
    constructor(){
        // 如果是继承，应该先调用父级，才有自己
        // 这里调用了父级的构造函数
        super();
    }

    sayHello() {
        console.log("我是学生，爱学习的学生");
    }
}

let s1 = new Student();
s1.sleep();
```

代码分析：如果使用了 extends 继承，那么，在系统自动生成的构造函数里面会自动调用 super，如果自己手写了 constructor，一定不要忘记这里有一个 super，这个关键字 super 指向的是父级对象

有参数的继承

```
/**
 * 无参数的继承
 */

class Person {
    constructor(userName, sex) {
        this.userName = userName;
        this.sex = sex;
    }

    sleep() {
        console.log(` ${this.userName} 在睡觉 `);
    }
}
```

```

        }

    }

    class Student extends Person {
        constructor(userName, sex, age) {
            super(userName, sex);
            this.age = age;
        }

        study() {
            console.log(`我是${this.sex}学生，我爱学习`);
        }
    }

    let s1 = new Student("江海丽", "女", 18);
    s1.sleep();
    s1.study();
}

```

注意事项：this关键字只能放在 super 关键字的后面

静态继承

在之前我们讲 `class` 的时候我们讲过，`class`会有静态的方法与静态的属性，现在我们需要了解一下 `static` 的东西是否可以继承

```

class Person {
    constructor(userName, sex) {
        this.userName = userName;
        this.sex = sex;
    }

    sleep() {
        console.log(`${this.userName}在睡觉`);
    }

    //静态方法
    static sayHello(){
        console.log("我在打招呼");
        // 静态的方法是可以使用静态的属性的
        console.log(`我的爱好是${this.hobby}`);
    }

    //静态属性
    static hobby = "看书";
}

class Student extends Person {
    constructor(userName, sex, age) {
        super(userName, sex);
        this.age = age;
    }

    study() {
        console.log(`我是${this.sex}学生，我爱学习`);
    }
}

Person.sayHello();

```

```
Student.sayHello();
console.log(Person.hobby);
console.log(Student.hobby);
```

1. 静态的方法与属性也是可以继承的
2. 静态的与非静态的仍然处于隔离状态，不要相互调用
3. 静态的方法可以使用静态的属性

方法的重写 **override**

当一个对象继承另一个对象的时候，默认就可以使用父级对象所有的属性及方法，但是如果父级对象的方法不满足条件的时候我们会**重写**这个方法，这种现象在ES5很常见，在ES6里面也常见，但是实现起来更简单了

```
class Person {
  constructor(userName) {
    this.userName = userName;
  }
  sayHello() {
    console.log(`大家好，我叫${this.userName}`);
  }
}

class Student extends Person {
  constructor(userName, sex) {
    super(userName);
    this.sex = sex;
  }
  //重写了一个sayHello的方法
  sayHello() {
    console.log(`我的性别是${this.sex}`);
    // 在这里又调用了父级的sayHello方法
    super.sayHello();
  }
}

let s1 = new Student("江海丽", "女");
s1.sayHello();
```

extends继承的特点

在上面我们看到 **extends** 可以实现快速的继承，通过 **extends** 继承有什么特点呢？

```
class Person {
  constructor(userName) {
    this.userName = userName;
  }
  sayHello() {
    console.log(`大家好，我叫${this.userName}`);
  }
}

class Student extends Person {
  constructor(userName, sex) {
```

```

        super(userName);
        this.sex = sex;
    }
    study() {
        console.log(`我是${this.sex}女生,我在学习`);
        super.sayHello(); //正常
        console.log(super.userName); //undefined
    }
}

let s1 = new Student("张珊", "女");

```

通过上面的方式实现继承以后，我们可以看到在效果上面展示如下

```

▼ Student {userName: '张珊', sex: '女'} ⓘ
  sex: "女"
  userName: "张珊"
▼ [[Prototype]]: Person
  ► constructor: class Student
  ► study: f study()
▼ [[Prototype]]: Object
  ► constructor: class Person
  ► sayHello: f sayHello()
  ► [[Prototype]]: Object

```

代码分析：

1. 把所有的属性放在了子级，把所有的方法放在了父级
2. 正是因为把所有的属性放在了子级，把所有的方法放在了父级，所以在调用父级的方法的时候可以使用 `super`，但是属性就一定使用 `this` 来调用

ES6里面的 `class、extends` 的继承其实本质上面就是ES5里面的组合继承与寄生继承，我们将上面的代码使用ES5去实现一次

```

function Person(userName){
    this.userName = userName;
}
Person.prototype.sayHello=function(){
    console.log(`大家好, 我叫${this.userName}`);
}

function Student(userName,sex){
    Person.call(this,userName);
    this.sex = sex;
}

```

```

Student.prototype = Object.create(Person.prototype);
Student.prototype.constructor = Student;

Student.prototype.study = function(){
    console.log(`我是${this.sex}女生，我在学习`);
}

let s1 = new Student("张珊", "女");

```

ES的语法

ES5的语法

ECMAScript6-3

接上往篇

面向对象的扩展

不可扩展对象

一级保护：不可扩展

在我们之前创建变量的时候，我们已经知道 `const` 定义的变量是不可更改的，但是 `const` 是锁栈不锁堆，所以通过 `const` 定义的基本数据类型是不可以更改的，但是定义的对象又可以更改堆里面的值

ES6为了更好的保护对象的内部，就堆出了新的技术

```

const userInfo = {
    userName: "张三"
};

console.log(userInfo.userName); //这里是正常的
userInfo.sex = "女";
console.log(userInfo);

userInfo = 123; //肯定报错

```

在上面的代码里面我们可以看到，虽然我们通过 `const` 去定义了一个对象，这个对象不能直接赋值改变，但是我们可以在这个对象上面扩展新的属性

现在我们希望这个对象构造好了以后，不要再扩展新的属性了，怎么办了？

现在我们就要使用一级保持：不可扩展对象

```
const userInfo = {
    userName: "张三"
};

//一级保持，阻止扩展
Object.preventExtensions(userInfo);

userInfo.sex = "女"; //这个时候的属性是添加不进去的
console.log(userInfo);
```

密封对象

二级保护：密封，不可扩展，不可删除

```
const userInfo = {
    userName: "张三"
};

//二级保护 密封
Object.seal(userInfo);
userInfo.sex = "女"; //扩展失败

delete userInfo.userName; //删除失败

console.log(userInfo);
```

冻结对象

三级保护：冻结，不可扩展，不可删除，不可更改

```
const obj = {
    userName: "张三"
};

// 开启三级保护 冻结
Object.freeze(obj);

obj.userName = "李四"; //不可更改
obj.sex = "男"; //不可扩展
delete obj.userName; //不可删除
console.log(obj);
```

Object.is()

在之前ES5的运算符里面，我们学过了 `==` 的全等操作，那么 `Object.is()` 其实就相当于这个操作，但是它比这个操作更加严格一点，它会深入到内存的当中去判断值是否相等，并且判断的是二进制值是否相等

```
console.log("1" == 1); //true
console.log("1" === 1); //false

console.log(Object.is(1, 1)); //true
console.log(Object.is("1", 1)); //false
```

```
//—————  
let obj1 = {  
    userName: "张三"  
};  
let obj2 = obj1; //浅拷贝，栈里面的地址相同  
  
let obj3 = {  
    userName: "张三"  
}  
console.log(Object.is(obj1, obj2)); //true  
console.log(Object.is(obj1, obj3)); //false
```

在上面的代码里面，我们可以看到，Object.is相当于是 `==` 的严格相等，在判断对象的时候，判断的是对象在内存当中栈的地址是否相同

`Object.is()` 会深入到内存当中去比较二进制

```
console.log(0 == 0); //true  
console.log(0 === 0); //true  
console.log(-0 === +0); //true  
  
let result = Object.is(-0, +0);  
console.log(result); //false 它的二进制不相等  
  
console.log(NaN === NaN); //false  
  
let result2 = Object.is(NaN, NaN);  
console.log(result2); //true
```

Reflect反射

提到反射，你们就应该想到一个东西镜子，它可以反射出任何东西。在JS里面的反射我们可以理解为 `Object` 的镜子。

标哥的理解：反射是对象的另一种操作方式，它弥补了 `Object` 下面操作对象的不足，相当于个“照妖镜”，可以把对象内部所有的东西都照出来

```
let obj = {  
    nickName: "小花花",  
    [Symbol("age")]: 18  
}  
  
Object.defineProperty(obj, "sex", {  
    enumerable: false,  
    value: "女"  
});  
  
let result1 = Object.keys(obj); //["nickName"]  
console.log(result1);  
  
let result2 = Object.getOwnPropertyNames(obj); //["nickName", "sex"]  
console.log(result2);
```

```
let result3 = Object.getOwnPropertySymbols(obj);           // [Symbol("age")]
console.log(result3);
```

//如果按照我们以前旧的方法，我们想获取这些属性是非常麻烦的

如果现在我们使用反射

```
let result4 = Reflect.ownKeys(obj);
console.log(result4);           // [ 'nickName', 'sex', Symbol(age) ]
```

在上面的代码当中，我们可以很清楚的感觉到这一点，反射就是操作对象的另一种方式，我们可以通过 `Reflect.ownKeys()` 来获取对象内部的所有属性

以下主要是要反射的方法列表

```
declare namespace Reflect {
    function apply(target: Function, thisArgument: any, argumentsList: ArrayLike<any>): any;
    function construct(target: Function, argumentsList: ArrayLike<any>, newTarget?: Function): any;
    function defineProperty(target: object, propertyKey: PropertyKey, attributes:PropertyDescriptor & ThisType<any>): boolean;
    function deleteProperty(target: object, propertyKey: PropertyKey): boolean;
    function get(target: object, propertyKey: PropertyKey, receiver?: any): any;
    function getOwnPropertyDescriptor(target: object, propertyKey: PropertyKey): PropertyDescriptor | undefined;
    function getPrototypeOf(target: object): object | null;
    function has(target: object, propertyKey: PropertyKey): boolean;
    function isExtensible(target: object): boolean;
    function ownKeys(target: object): (string | symbol)[];
    function preventExtensions(target: object): boolean;
    function set(target: object, propertyKey: PropertyKey, value: any, receiver?: any): boolean;
    function setPrototypeOf(target: object, proto: object | null): boolean;
}
```

1. `apply` 方法，这个就去相当于之前的方法名`.apply`的调用

```
let obj = {
    userName: "张三",
    sayHello(str){
        console.log(`大家好，我叫${this.userName}`);
        console.log(`我接收到的值是${str}`);
    }
}
// obj.sayHello("普通调用");
// obj.sayHello.apply(obj, ["apply调用"]);
Reflect.apply(obj.sayHello, obj, ["Reflect调用"]);
```

2. `construct` 方法，执行一个构造函数，相当于`new`了一个构造函数

```
class Student{
    constructor(userName){
        this.userName = userName;
    }
}
let s1 = new Student("张三");
let s2 = Reflect.construct(Student, ["李四"]);
```

3. `defineProperty` 这个就是原来的 `Object.defineProperty`

```
let obj = {
    userName: "张三"
}

Object.defineProperty(obj, "sex", {
    configurable: false,
    value: "女"
});

Reflect.defineProperty(obj, "age", {
    value: 18,
    configurable: false
});
```

4. `deleteProperty` 删除一个属性

```
let obj = {
    userName: "张三",
    sex: "女",
    age: 18
}

delete obj.sex;           //删除一个属性
Reflect.deleteProperty(obj, "age");           //删除age属性
console.log(obj);
```

5. `get` 获取某一个属性的属性值

```
let obj = {
    userName: "张三",
    sex: "女",
    age: 18
};

console.log(obj.userName);
console.log(obj["sex"]);
console.log(Reflect.get(obj, "age"));
```

6. `getOwnPropertyDescriptor` 这个方法相当于对象里面的 `Object.getOwnPropertyDescriptor()`

7. `getPrototypeOf` 获取某一个对象的原型对象(也叫父级对象), 相当于获取 `__proto__`

8. `has()` 判断某一个对象上面是否有某一属性, 相当于 `hasOwnProperty()` 方法

9. `isExtensible()` 当前这个对象是否不可扩展 (也就是是否处于一级保持) ;

10. `ownKeys()` 获取当前对象上面所有的属性名

11. `preventExtensions()` 把某一个对象设置为不可扩展对象，相当于
`Object.preventExtensions()` 设置对象的一级保护
12. `set()` 相当于对对象的某一个属性进行赋值

```
let obj = {  
    userName: "张三",  
    age: 19  
}  
  
obj.userName = "李四";  
Reflect.set(obj, "age", 20);
```

13. `setPrototypeOf()` 设置某一个对象的原型

```
let obj1 = {  
    userName: "张三"  
}  
let obj2 = {  
    age: 18  
}  
obj1.__proto__ = obj2;  
Reflect.setPrototypeOf(obj1, obj2);
```

Proxy代理

代理其实可以理解为明星的经纪人（代理人，律师），**在JS当中的代理指的是全局拦截**

代理的概念

场景：一般的明星都有经纪人，明星的大部分工作与生活都与经纪人脱不开关系，也都是由经纪人来安排，我们就可以理解为经纪人就是这个明星的代理

如：现有一个明星叫胡歌，我做为它的粉丝，我向他写了一封信，劝它在拍戏之余多多注意一下息的自体，多拍一点好戏。这个信肯定是先到胡歌的经纪人手上，然后才有可能交给胡歌！

如：标哥是胡歌的粉丝，现在标哥赚钱了，发达了，想请胡歌拍电影，片酬200W，这个时候我并不需要直接与胡歌见面，找它的经纪人就可以了，不用与胡歌商量

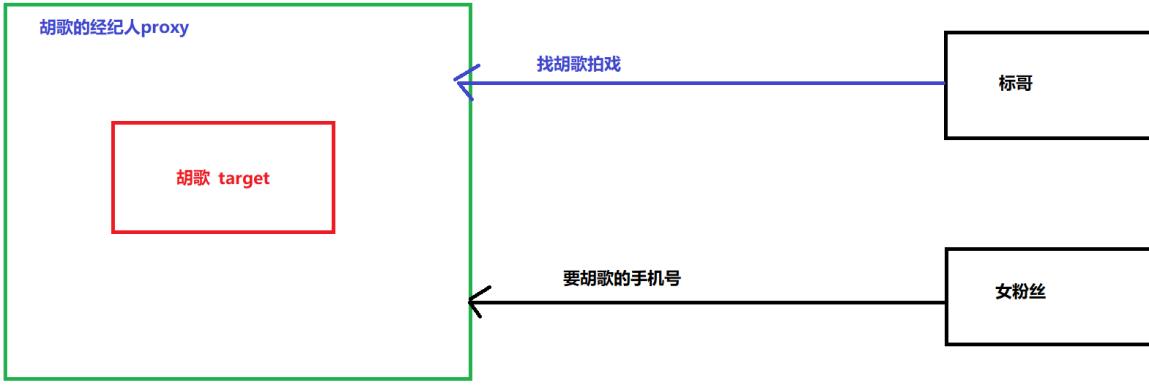
如：某一个女粉丝约胡歌晚上去某个小酒吧喝酒，还要胡歌的微信及手机号，这时候胡歌的经纪人得到这个要求以后觉得这个要求不合理，可以直接拒绝，这个要求就不会传递给胡歌

在上面的场景里面，我们可以看到胡歌的大部分工作都由经纪人在代理，胡歌本身是一个目标，胡歌的经纪人我们就可以认为是胡歌的代理，**它可以代理胡歌的所有事情**

1. 代理的对象叫 `proxy`
2. 目标对象叫 `target`

一般情况下是 `proxy` 代理 `target`

在上面的场景里面面 `target` 指的就是胡歌，而他的经纪人就是 `proxy`



从上面的图当中我们可以看到，无论我们对胡歌进行什么操作都会先经过他的经纪人（代理对象）。这一种场景在JS里面就叫代理，所以JS里面的代理也叫“全局拦截”

创建代理

在ES6里面如果要创建代理，我们就可以使用下面的方法

```
// 下面就是明星胡歌
let huge = {
  userName: "胡歌",
  work: "拍戏",
  money: 5000,
  address: "上海市静安区"
}

let huge_proxy = new Proxy(huge, {
  //handler这里决定了如果代理胡歌，怎么样操作胡歌的事情
});

> huge
< ▼ {userName: '胡歌', work: '拍戏', money: 5000, address: '上海市静安区'} ⓘ
  address: "上海市静安区"
  money: 5000
  userName: "胡歌"
  work: "拍戏"
  ▶ [[Prototype]]: Object

> huge_proxy
< ▼ Proxy {userName: '胡歌', work: '拍戏', money: 5000, address: '上海市静安区'} ⓘ
  ▶ [[Handler]]: Object
  ▼ [[Target]]: Object
    address: "上海市静安区"
    money: 5000
    userName: "胡歌"
    work: "拍戏"
    ▶ [[Prototype]]: Object
    [[IsRevoked]]: false
```

在上面的图片当中，我们可以的到以下2点

1. Handler 代表上体代理目标的哪些操作
2. Target 代理的目标对象

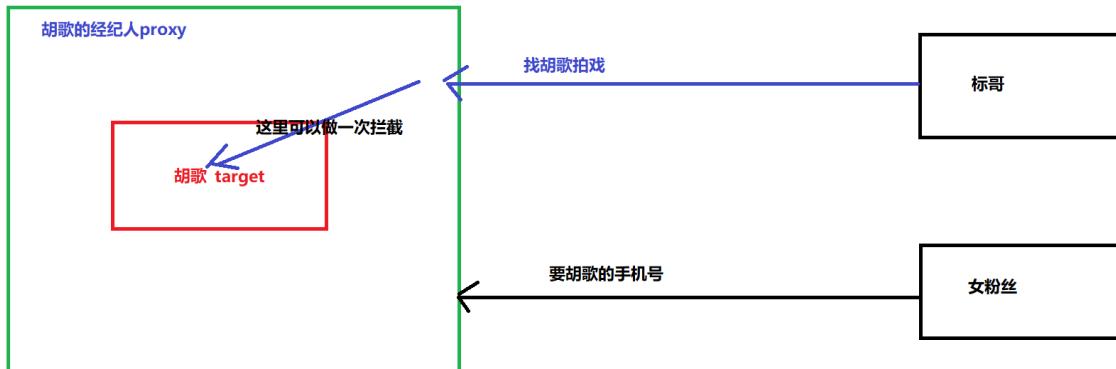
我们在操作胡歌的代理对象 `huge_proxy` 的时候相当于就是在操作胡歌 `huge`

```
> huge.userName  
< '胡歌'  
> huge_proxy.userName  
< '胡歌'
```

现在我们可以看到，我们操作代理对象 `proxy` 与操作目标对象 `target` 结果是一样的，那我们为什么需要代理呢？

代理的作用

从上一个部分我们可以看到，当我们需要去操作一个对象的时候，其实是可以操作它的代理对象的，代理对象是可以实现全局



在上图当中我们看到，当代理对象与目标对象沟通的时候可以做一次拦截，我们的重点应这一个拦截里面

```
// 下面就是明星胡歌
let huge = {
  userName: "胡歌",
  work: "拍戏",
  money: 5000,
  address: "上海市静安区"
}

let huge_proxy = new Proxy(huge, {
  // 当要取值的时候触发
  get(target, p) {
    //target代表的是目标对象，也就是huge
    //p代表代要操作哪个属性
    if (p === "money") {
      return "胡歌的薪资保密";
    } else {
      return target[p];
    }
  },
  //当要赋值的时候就会触发set
  set(target, p, v) {
    //target目标对象 huge
    //p属性值
    //v属性值
    if (p === "money") {
```

```
        console.log("经纪人贪污了，不给胡歌")
    }
    else{
        target[p] = v;
    }
}
});
```

1. 在上面的代码里面，我们就可以看到代理的作用了，代理就是一个全局的拦截，我们可以拦截所有的操作
2. `get` 是在取值的时候触发的，`set` 是在赋值的时候触发了
3. `target` 代表目录对象，`p` 代表属性名，`v` 代表属性值

代理的应用点

我们刚刚已经看到了，对代理对象进行取值或赋值是会触发相应的操作的，这个操作有什么应用点呢？

场景：我们想创建一个对象，这个对象下面有 `userName`, `pwd` 两个属性，这两个属性里面只有 `userName` 可以取值, `pwd` 不能取值

```
let obj = {
    userName: "张三",
    pwd: 123456
}

let obj_proxy = new Proxy(obj, {
    get(target, p){
        if(p === "pwd"){
            return undefined;
        }
        else{
            return target[p];
        }
    }
});

console.log(obj_proxy.userName);
console.log(obj_proxy.pwd);
```

但是上面的场景是有隐患的，我们的目标对象还是暴露在全局

```
console.log(obj.pwd);
```

正常情况下：代理对象要包裹目标对象

```
let obj_proxy = new Proxy((( )) => {
    return {
        userName: "张三",
        pwd: 123456
    }
}(), {
    //代理的操作
    get(target, p) {
        if (p === "pwd") {
            return undefined;
        }
    }
});
```

```

        }
        else {
            return target[p];
        }
    );
}

console.log(obj_proxy.userName);
console.log(obj_proxy.pwd);

```

在上面的代码当中我们可以看到已不能访问目标对象了，因为目标对象通过立即执行函数进行了返回，这个时候我们间接的操作代理对象的时候就可以实现目标对象的全局理了

同时，我们已做了相应的限制条件，对于这个地方的 `pwd` 属性的访问只返回 `undefined`

通过代理实现属性私有化

```

let student = {
    userName: "袁池康",
    sex: "男",
    age: 19,
    hobby: "读书，睡觉",
    tel: 18526374859,
    address: "湖北省武汉市江夏区关山大道华夏学院101宿舍上铺",
    IDCard: "420000200901011234"
}

```

我们希望实现一些私有化的属性，外部只可以访问，但是不能赋值，怎么办呢？

场景：我们现在希望 `tel` 和 `IDCard` 这两个属性只可以取值，不可以赋值，怎么办呢？

```

let student = {
    userName: "袁池康",
    sex: "男",
    age: 19,
    hobby: "读书，睡觉",
    _tel: 18526374859,
    address: "湖北省武汉市江夏区关山大道华夏学院101宿舍上铺",
    _IDCard: "420000200901011234"
}

```

我们在需要私有化的属性前面添加了一个 `_` 来表示，这个我们再通过代理去设置

```

let student_proxy = new Proxy((( ) => {
    return {
        userName: "袁池康",
        sex: "男",
        age: 19,
        hobby: "读书，睡觉",
        _tel: 18526374859,
        address: "湖北省武汉市江夏区关山大道华夏学院101宿舍上铺",
        _IDCard: "420000200901011234"
    }
})(), {
    // 赋值的会调用
    set(target, p, v) {

```

```

        if (p.startsWith("_")) {
            return;
        }
        else {
            target[p] = v;
        }
    },
    // 代理了删除属性的操作
deleteProperty(target, p) {
    if (p.startsWith("_")) {
        return false;
    }
    else {
        return delete target[p];
    }
}
);

student_proxy._tel = 12312312312;           //不会成功，会被拦下来
student_proxy.userName = "袁老大";          //成功
delete student_proxy._IDCard;               //制作，会被拉下来

console.log(student_proxy);

```

代理的操作列表

```

interface ProxyHandler<T extends object> {
    apply?(target: T, thisArg: any, argArray: any[]): any;
    construct?(target: T, argArray: any[], newTarget: Function): object;
    defineProperty?(target: T, property: string | symbol, attributes: PropertyDescriptor): boolean;
    deleteProperty?(target: T, p: string | symbol): boolean;
    get?(target: T, p: string | symbol, receiver: any): any;
    getOwnPropertyDescriptor?(target: T, p: string | symbol): PropertyDescriptor | undefined;
    getPrototypeOf?(target: T): object | null;
    has?(target: T, p: string | symbol): boolean;
    isExtensible?(target: T): boolean;
    ownKeys?(target: T): ArrayLike<string | symbol>;
    preventExtensions?(target: T): boolean;
    set?(target: T, p: string | symbol, newValue: any, receiver: any): boolean;
    setPrototypeOf?(target: T, v: object | null): boolean;
}

```

可取消的代理

正常情况下，我们可以代理一个对象，但是如果不想代理这个对象了，也可以取消这个对象的代理

```

let huge = {
    userName: "胡歌",
    work: "拍戏",
    money: 5000,
    address: "上海市静安区"
}

let huge_proxy = Proxy.revocable(huge, {

```

```

        get(target, p) {
            return target[p];
        },
        set(target, p, v) {
            target[p] = v;
        }
    );
}

console.log(huge_proxy.proxy.money);
huge_proxy.revoke();           //撤销代理
console.log(huge_proxy.proxy.address);

```

上面的代码其实用解构会更好一些

```

//proxy就是代理的对象，revoke就是取消代理的方法
let { proxy, revoke } = huge_proxy;
console.log(proxy.userName);
revoke();
console.log(proxy.address);

```

代理小练习

有一本书的属性为：{"name":“《ES6基础系列》”, “price”: 56}; 要求使用Proxy对象对其进行拦截处理，name属性对外为“《ES6入门到懵逼》”, price属性为只读。 (练习题)

```

let book = new Proxy(() => {
    return {
        name: "《ES6基础系列》",
        price: 56
    }
})(), {
    get(target, p) {
        if (p === "name") {
            return "《ES6从入门到懵逼》";
        }
        else {
            return target[p];
        }
    },
    set(target, p, v) {
        if (p === "price") {
            return;
        }
        else {
            target[p] = v;
        }
    },
    deleteProperty(target, p) {
        if (p === "price") {
            return false;
        }
        else {
            return delete target[p];
        }
    }
}

```

ES6异步编程及模块化

异步编程

在之前的ES5里面，我们已经接触过异步的概念了，并且得到了一个非常好的口诀：同步等待，异步执行。

异步最大的缺点就是无法拿到返回值

在之前的ES5里面，如果我们想得到返回值，我们只能使用回调函数

```
const abc = (callBack) => {
  setTimeout(() => {
    let num = ~~(Math.random() * 100)
    if (typeof callBack === "function") {
      callBack(num);
    }
  }, 2000);
}
abc(num => {
  console.log(`你的随机数的返回值是${num}`);
});
```

在ES6里面如果想解决异步，它连续推出了几种解决方案

1. 回调函数
2. 生成器函数
3. **Promise**
4. `async/await`

异步一直是ES6里面的难点，所以w3c一直在尝试对异步编程做好的解决方法

回调函数处理异步

在之前的ES5里面，最典型的异步场景有2个

1. **setTimeout/setInterval**
2. **ajax**

现在我们就使用定义器来模拟异步的场景，来模拟驾照考试的场景

```
// 假设陈怡静要考驾照
// 科目一的方法
const kemu1 = (callBack) => {
  console.log(`陈怡静在考科目一`);
  // 30分钟以后可以得到结果
  setTimeout(() => {
    //随机产生了一个成功
    let result1 = ~~(Math.random() * 15) + 85;
    if (typeof callBack === "function") {
      callBack(result1);
    }
  }, 2000);
}

//科目二的方法
```

```

const kemu2 = (callBack) => {
    console.log(`陈怡静在考科目二`);
    setTimeout(() => {
        let result2 = ~~(Math.random() * 30) + 70;
        if (typeof callBack === "function") {
            callBack(result2);
        }
    }, 2000);
}

// 科目三的方法
const kemu3 = (callBack) => {
    console.log(`陈怡静在考科目三`);
    setTimeout(() => {
        let result3 = ~~(Math.random() * 25) + 75;
        if (typeof callBack === "function") {
            callBack(result3);
        }
    }, 2000);
}

kemu1(result1 => {
    if (result1 >= 90) {
        console.log(`陈怡静的科目一的考度成绩为:${result1},请继续考试`);
        kemu2(result2 => {
            if (result2 >= 80) {
                console.log(`陈怡静科目二的考试成绩为${result2},请继续考试`);
                kemu3(result3 => {
                    if (result3 >= 90) {
                        console.log(`你的科目三的成绩为${result3},考试结束,拿到驾照`);
                    } else {
                        console.log(`你的科三的成绩为${result3},成绩不合格,请回中心打印成绩单`);
                    }
                });
            }
        });
    } else {
        console.log(`陈怡静,科目考试成为线${result2},不合格,请回中心打印成绩单`);
    }
});
else {
    console.log(`陈怡静,你的科目一的成绩不合格,请回中心打印成绩单`);
}
);
}

```

使用回调来处理异步编程这是一个非常简单的，但是也容易产生一个很大的问题，它会形成**回调地狱**，每一次的回调都要形成一个嵌套

Promise异步处理

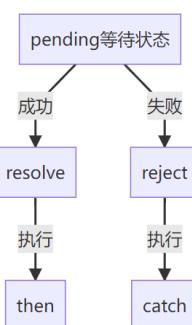
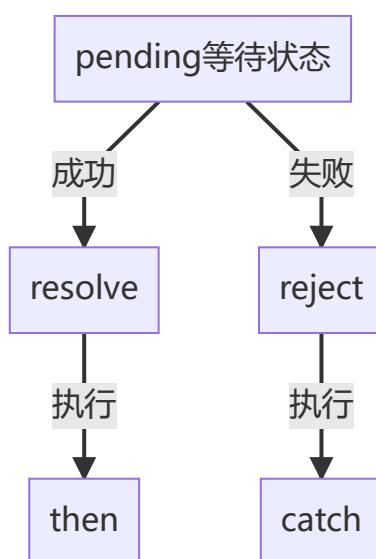
【重点，超重点，必考题，必面点】

Promise是ES6下面用来处理异步非常好的一个东西，全称叫“承诺”

承诺会有几个特点

1. 承诺是一定会得到结果的
2. 承诺是有状态的，分别是三个状态
 - pending 等待状态
 - resolve/fulfilled 成功的状态
 - reject 失败的状态

状态是不可逆的，如果状态一旦由等待变成一个状态，就不能再变回来。如就pending--->fulfilled
成功状态就不可能再次变成 pending 状态，更不可能变为失败 reject 状态



```
const kemu1 = () => {
  console.log("陈静怡在考科目一");
  // 约定好，你给一个承诺 有结果了立即告诉我
  let p = new Promise((resolve, reject) => {
    setTimeout(() => {
      let result1 = ~~(Math.random() * 15) + 85;
      if (result1 >= 90) {
        // 成功
        resolve(result1);
      }
      else {
```

```

        //失败
        reject(result1);
    }
}, 2000);
});

//将承诺返回到外边
return p;
}

let p1 = kemui();
//p1就是陈怡静给我们的承诺
//如果承诺的内部调用的是`resolve`将会自动执行`then`
//如果承诺的内部调用的是`reject`，将会自动执行`catch`


p1.then(result1 => {
    console.log(`你执行的是resolve, 你考试成功了, 你的成绩是${result1}`);
}).catch(result1 => {
    console.log(`你执行的是reject, 你考试失败了, 小家伙, 又只考了${result1}分`);
});

```

```

1 const kemui = () => {
2     console.log("陈静怡在考科目一");
3     // 约定好, 你给一个承诺 有结果了立即告诉我
4     let p = new Promise((resolve, reject) => {
5         setTimeout(() => {
6             let result1 = ~~(Math.random() * 15) + 85;
7             if (result1 >= 90) {
8                 //成功
9                 resolve(result1);
10            }
11            else {
12                //失败
13                reject(result1);
14            }
15        }, 2000);
16    });
17    //将承诺返回到外边
18    return p;
19 }

20
21 let p1 = kemui();
22 //p1就是陈怡静给我们的承诺
23 //如果承诺的内部调用的是`resolve`将会自动执行`then`
24 //如果承诺的内部调用的是`reject`，将会自动执行`catch`


25 p1.then(result1 => {
26     console.log(`你执行的是resolve, 你考试成功了, 你的成绩是${result1}`);
27 }).catch(result1 => {
28     console.log(`你执行的是reject, 你考试失败了, 小家伙, 又只考了${result1}分`);
29 });

```

resolve代表成功，会执行then
reject代表失败，会执行catch



代码分析：

1. Promise就是一个承诺，陈怡静现在去考科目一去了，它给了我们一个承诺，只要考试成功出来，立即告诉我们结果，所以最开始的时候 `Promise` 的状态只可能是 `pending` 等待状态
2. 定时器经过2s钟以后坐产生一个分数的结果，这个结果如果大于90分，我们就调用 `resolve` 成功的转换，这个时候这个承诺就会由 `pending` 的状态转换成成功的状态 `fulfilled`
3. 如果这个分数小于90分，就调用 `reject()` 方法承诺的状态转换成失败状态
4. 如果成功以后，这个承诺会自动调用 `then` 的方法，如果失败以后，这个承诺会自动调用 `catch` 的方法

在上面的代码里面，如果我们有多个考试，怎么办呢？

```
const kemu1 = () => {
    console.log("陈静怡在考科目一");
    let p = new Promise((resolve, reject) => {
        setTimeout(() => {
            let result1 = ~~(Math.random() * 15) + 85;
            if (result1 >= 90) {
                resolve(result1);
            } else {
                reject(result1);
            }
        }, 2000);
    });
    return p;
}

const kemu2 = () => {
    console.log("陈静怡在考科目二");
    let p = new Promise((resolve, reject) => {
        setTimeout(() => {
            let result2 = ~~(Math.random() * 25) + 75;
            if (result2 >= 80) {
                resolve(result2);
            } else {
                reject(result2);
            }
        }, 2000);
    });
    return p;
}

const kemu3 = () => {
    console.log("陈静怡在考科目三");
    let p = new Promise((resolve, reject) => {
        setTimeout(() => {
            let result3 = ~~(Math.random() * 15) + 85;
            if (result3 >= 90) {
                resolve(result3);
            } else {
                reject(result3);
            }
        }, 2000);
    });
    return p;
}

let p1 = kemu1();
p1.then(result1 => {
    console.log(`陈怡静科目一的成绩为:${result1}`);
    let p2 = kemu2();
    return p2; //将这个承诺直接返回到外边
}
```

```

}).then(result2 => {
  console.log(`陈怡静科目二的结果为${result2}`);
  let p3 = kemu3();
  return p3;
}).then(result3 => {
  console.log(`陈怡静科目三的成绩为${result3}, 考试结束拿驾照`);
}).catch(error => {
  //所有的考试失败都会在这里
  console.log(`考试不合格, 你的成绩为${error}`);
});

```

代码说明：

1. 每次的一科目考试，我们返回的都是一个承诺
2. `then` 是在成功以后会调用的，`catch` 是在失败以后才会调用的，所以进入到 `then` 就说明这次考试是成功的，我们就可以进行下一次的考试

Promise.all()

如果有多个 `Promise`，我们希望这些 `promise` 都成功以后再去执行最后的操作，怎么办呢？

```

/**
 * Promise.all 所有的Promise都成功以后，才会进行的操作
 */

let a = () => {
  console.log(`第1个Promise在执行`);
  let p = new Promise((resolve, reject) => {
    setTimeout(() => {
      let x = 100;
      resolve(x);
    }, 3000);
  })
  return p;
}

let b = () => {
  console.log(`第2个Promise也在执行`);
  let p = new Promise((resolve, reject) => {
    setTimeout(() => {
      let x = 200;
      resolve(x);
    }, 5000);
  });
  return p;
}

// 我们希望这2个函数同时运行，都成功了以后再执行后面的操作
Promise.all([a(), b()]).then(result => {
  console.log(`两个承诺同时成功了`, result);
  //这里的result的结果就是一个数组，数组里面依次存放的就是每个promise的结果
  // [100, 200]
}).catch(error => {
  console.log(`其中至少一个失败了, ${error}`);
});

```

代码分析：

在上面的代码里面，我们有2个方法，这2个方法都返回了 `Promise`，但是在这两个方法里面，我们如果希望这两个方法都成功以后再去执行某些操作，这个时候我们就要把这些东西放在一个数组里面，再通过 `Promise.all` 的方式来调用

`Promise.all()` 去调用的时候，必须里面所有的承诺的结果都成功才可以进入到 `then`，任何一个失败了都会进入到 `catch`

`Promise.any()`

这一个方法正好与上面的方法相反，只有任何一个成功了，那么 `Promise` 就会转成成功，去执行 `then`。如果所有的 `Promise` 都失败了，才会转向 `catch`

```
let a = () => {
    console.log(`第1个Promise在执行`);
    let p = new Promise((resolve, reject) => {
        setTimeout(() => {
            let x = 100;
            resolve(x); // 成功
        }, 3000);
    })
    return p;
}

let b = () => {
    console.log(`第2个Promise也在执行`);
    let p = new Promise((resolve, reject) => {
        setTimeout(() => {
            let x = 200;
            reject(x); // 失败
        }, 5000);
    });
    return p;
}

Promise.any([a(), b()]).then(result => {
    console.log(`其中有某一个成功了`, result);
}).catch(error => {
    console.log(`你两个都失败了`, error);
});
```

代码分析：

`any` 与 `all` 正好相反，只要其中有任何一个 `Promise` 成功了 `resolve` 了，那么，这个就会执行 `then`，如果是所有的承诺都失败 `reject` 了，那么才会进入到 `catch` 里面

`Promise.race()`

这个 `race` 有比赛的意思，它代表多个 `promise` 同时执行，谁先得到结果就听谁的

```
let a = () => {
    console.log(`第1个Promise在执行`);
    let p = new Promise((resolve, reject) => {
```

```

        setTimeout(() => {
            let x = 100;
            resolve(x); // 成功
        }, 3000);
    })
    return p;
}

let b = () => {
    console.log(`第2个Promise也在执行`);
    let p = new Promise((resolve, reject) => {
        setTimeout(() => {
            let x = 200;
            reject(x); // 失败
        }, 5000);
    });
    return p;
}

// race 比赛，哪一个先得到结果，Promise 执行的就是这个结果
Promise.race([a(), b()]).then(result => {
    console.log(`你成功了，结果是`, result);
}).catch(error => {
    console.log(`你失败了，结果是`, error);
})

```

代码分析：

上面的代码仍然同时执行了2个 `Promise`，但是在这个地方要注意，谁先得到结果，最后就执行执行

我们可以看到 `a()` 里面只需要3000毫秒就得到了结果，它是最先出结果的，所以，最后就以它的 `resolve` 为主，整个 `Promise` 执行的就 `then`

async/await关键字

在之前的代码里，如果我们要将多个Promise依次执行，我们需要多次操作 `then` 的方法，如下所示

```

const kemu1 = () => {
    console.log("陈静怡在考科目一");
    let p = new Promise((resolve, reject) => {
        setTimeout(() => {
            let result1 = ~~(Math.random() * 15) + 85;
            if (result1 >= 90) {
                resolve(result1);
            } else {
                reject(result1);
            }
        }, 2000);
    });
    return p;
}

```

```

const kemu2 = () => {
    console.log("陈静怡在考科目二");
    let p = new Promise((resolve, reject) => {
        setTimeout(() => {
            let result2 = ~(Math.random() * 25) + 75;
            if (result2 >= 80) {
                resolve(result2);
            } else {
                reject(result2);
            }
        }, 2000);
    });
    return p;
}

const kemu3 = () => {
    console.log("陈静怡在考科目三");
    let p = new Promise((resolve, reject) => {
        setTimeout(() => {
            let result3 = ~(Math.random() * 15) + 85;
            if (result3 >= 90) {
                resolve(result3);
            } else {
                reject(result3);
            }
        }, 2000);
    });
    return p;
}

let p1 = kemu1();
p1.then(result1 => {
    console.log(`陈怡静科目一的成绩是${result1},继续开始下一门考试`);
    let p2 = kemu2();
    return p2;
}).then(result2 => {
    console.log(`陈怡静科目二的成绩是${result2},继续开始下一门考试`);
    let p3 = kemu3();
    return p3;
}).then(result3 => {
    console.log(`陈怡静科目三的成绩是${result3},考试完成,拿驾照`);
}).catch(error => {
    console.log(`看什么看,你考挂了, ${error}`);
})

```

上面的代码是 ES6 里面的代码，我们仍然感觉到这个代码不是了，因为它部分的 `then` 里面还是使用了回调。我们能不能不使用回调

为了解决这样的问题，ES7 里面推出了新的关键字，叫 `async, await`，这两个关键字的作用就是将异步代码转换成同步代码

```
const a = () => {
```

```

console.log(`函数开始执行`);

let p = new Promise((resolve, reject) => {
    setTimeout(() => {
        let x = 100;
        resolve(x);
    }, 3000);
});

return p;
}

/*
let p = a();
p.then(result => {
    console.log("你resolve的结果是", result);
});
*/

```

//这个x我就想得到resovle的100怎么办?
// let x = a(); //如果这么写，只能得到a()返回的结果，它是一个承诺
//但是如果我们要等到承诺的结果，我们就必须使用`await`关键字
let x = await a();

代码分析：

1. 如果没有await那么我们得到的结果将会是一个承诺Promise
2. 如果得等待到承诺的结果，在这个返回Promise函数的外部使用 **await** 关键字

如果我们直接去执行上面的代码，这个时候会报错，如下所示

```

let x = await a();
          ^^^^^^
代码分析:

SyntaxError: await is only valid in async functions and the top level bodies of modules
    at Object.compileFunction (node:internal:752:18)

```

这个错误的原因在于， **await 关键字必须与 **async** 关键字结合**

```

const a = () => {
    console.log(`函数开始执行`);

    let p = new Promise((resolve, reject) => {
        setTimeout(() => {
            let x = 100;
            resolve(x);
        }, 3000);
    });

    return p;
}

async function ff() {
    let n = await a();           //这个时候的n就是resolve的值
    console.log(`你成功以后的结果是`, n);
}

ff();

```

现在我们就来完整的演示一下 `async/await` 的使用

```
const a = () => {
    console.log(`函数开始执行`);
    let p = new Promise((resolve, reject) => {
        setTimeout(() => {
            let x = ~~(Math.random() * 100);
            if(x >= 50){
                resolve(x); // 成功
            } else{
                reject(x); // 失败
            }
        }, 3000);
    });
    return p;
}

/*
let p = a();
p.then(result=>{
    // 成功以后会进入then, result就是你resolve()里面的东西
}).catch(error=>{
    // 失败以后就会进入 catch, error就是你reject()里面的东西
}).finally(()=>{
    // 无论成功还是失败都会执行
})
*/

async function ff() {
    // await等待的是promise，并且只能等待到promise成功的结果，也就是resolve的结果
    try {
        let n = await a();
        console.log(`你resolve的值是`, n)
    } catch (error) {
        // reject就会执行catch里面的代码
        console.log(`你reject的值是`, error);
    }
    finally{
        console.log(`无论你是成功，还是失败，都会成功`)
    }
}
ff();
```

总结

1. `await` 必须与 `async` 一起使用
2. `await` 等待到一只能是 `Promise`，并且能等待到 `Promise` 成功的结果，也就是 `resolve` 的结果
3. `await` 失败了就会跳转到 `catch` 里面去，`catch` 里面的 `error` 就是 `reject` 的东西
4. 在等待承诺的过程当中，无论是成功还是失败，都会执行 `finally`
5. `Promise` 的代码是可以转换成 `await` 的，`then` 就相当于 `try`，`catch` 对应的就是 `catch`，`finally` 对应的就是 `finally`

现在我们就通过 `async/await` 将上面的异步代码转换为同步代码

```
const kemu1 = () => {
    console.log("陈静怡在考科目一");
    let p = new Promise((resolve, reject) => {
        setTimeout(() => {
            let result1 = ~(Math.random() * 15) + 85;
            if (result1 >= 90) {
                resolve(result1);
            } else {
                reject(result1);
            }
        }, 2000);
    });
    return p;
}

const kemu2 = () => {
    console.log("陈静怡在考科目二");
    let p = new Promise((resolve, reject) => {
        setTimeout(() => {
            let result2 = ~(Math.random() * 25) + 75;
            if (result2 >= 80) {
                resolve(result2);
            } else {
                reject(result2);
            }
        }, 2000);
    });
    return p;
}

const kemu3 = () => {
    console.log("陈静怡在考科目三");
    let p = new Promise((resolve, reject) => {
        setTimeout(() => {
            let result3 = ~(Math.random() * 15) + 85;
            if (result3 >= 90) {
                resolve(result3);
            } else {
                reject(result3);
            }
        }, 2000);
    });
    return p;
}

/*
let p1 = kemu1();
p1.then(result1 => {
    console.log(`陈怡静科目一的成绩是${result1},继续开始下一门考试`);
    let p2 = kemu2();
}
*/
```

```

        return p2;
    }).then(result2 => {
        console.log(`陈怡静科目二的成绩是${result2},继续开始下一门考试`);
        let p3 = kemu3();
        return p3;
    }).then(result3 => {
        console.log(`陈怡静科目三的成绩是${result3},考试完成, 拿驾照`);
    }).catch(error => {
        console.log(`看什么看, 你考挂了, ${error}`);
    }).finally(()=>{
        console.log(`请回中心打印成绩单`);
    })
}

const ff = async () => {
    try {
        let result1 = await kemu1();
        console.log(`科目一成绩为${result1},继续开始考试`);
        let result2 = await kemu2();
        console.log(`科目二成绩为${result2},继续开始考试`);
        let result3 = await kemu3();
        console.log(`科目三成功为${result3}, 考试结束, 拿驾照`);
    } catch (error) {
        console.log(`你这都考不过, 才${error}分`);
    }
    finally {
        console.log(`请回中心打印成绩单`);
    }
}
ff();

```

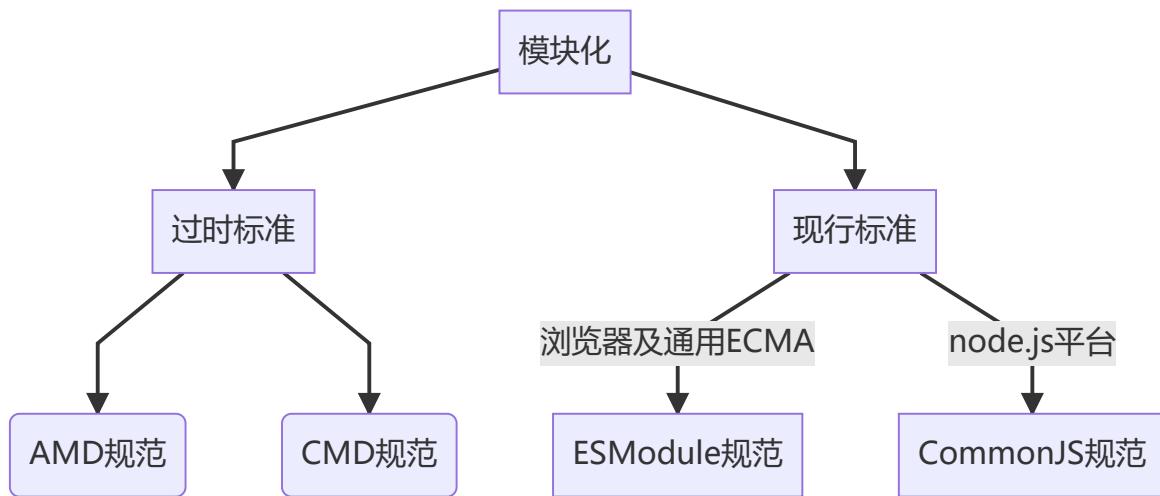
模块化

在之前的ES5时面，如果我们要去导入一个JS文件，我们就要把这个JS文件通过 `script` 标签的 `src` 属性引入到页面当中去，然后在页面里面去执行，但是这做会存在一些缺点

1. 如果直接在 `index.html` 里面导入，我们根本就看不出来文件的相互引用关系。因为只要把文件放在 `html` 里面它就可以使用了
2. 万一要是没有 `index.html`，这个时候怎么要让两个文件实现引用关系？`node.js` 没有 `html`。
就没有 `script` 标签

为了解决上面的问题，w3c就推出了模块化的概念

目前的模块化的标准有很多，我们大体上可以分为 现行标准 或 过时标准



在ESModule的模块化规范里面，只有2个关键字，分别是 `export` 导出， `import` 导入

Person.js文件

```

class Person {
    constructor(userName){
        this.userName = userName;
    }
    sayHello(){
        console.log(`大家好，我叫${this.userName}`);
    }
}

//默认导出Person
export default Person;

```

Student.js文件

```

//当Student要引入当前文件夹下面的`Person.js`的时候
import Person from "./Person.js";
class Student extends Person {
    constructor(userName, sex) {
        super(userName);
        this.sex = sex;
    }
    study() {
        console.log(`${this.userName}在学习`);
    }
}
export default Student;

```

index.html文件

```

<script type="module">
    import Student from "./js/Student.js";
    let s = new Student("张三", "男");
    s.study();
    s.sayHello();
</script>

```

这个地方请一定要注意，不能使用 `type="text/javascript"` 一定要指明它的 `type="module"`



通过上面的这种引用结构，我们也看得很清楚

注意事项：

1. `ESModule` 只支持 `http/https` 协议，不支持 `file` 协议

```
http://127.0.0.1:5007/index.html
```

上面的是可以的，而下面的就不可以

```
file:///D:/H2204/0930/code/093002/index.html
```

模块化重点扩展【引用】

在上一个问题里面，如果我们新建一个 `index.js` 文件以后，去导入，怎么办？

```
index.html - 093003 - Visual Studio Code
index.html
<html>
  <head>
    <script type="module">
      import "./index.js";
    </script>
  </head>
  <body>
    <script type="module">
      import "./index.js"; // 这里就不能from，直接导入就可以了
    </script>
  </body>
</html>
```

```
index.js
// 我希望在这里创建一个学生对象，怎么办？
import Student from "./Student.js";

let s = new Student("洪延军", "男");
s.study();
s.sayHello();
```

除了上面的方式以外，还可以使用下面的方式

```
index.html - 093003 - Visual Studio Code
index.html
<html>
  <head>
    <script type="module" src="./index.js"></script>
  </head>
  <body>
    <script type="module" src="./index.js"></script>
  </body>
</html>
```

```
index.js
// 我希望在这里创建一个学生对象，怎么办？
import Student from "./Student.js";

let s = new Student("洪延军", "男");
s.study();
s.sayHello();
```

nodejs介绍与安装

关于node.js

node.js其实就是运行在服务器的javascript(以前我们学习的js是运行在浏览器里面的)

在之前我们都认为JS只能运行在浏览器里面，这是不对的，现在的JS可以依托于 `node.js` 的平台运行在服务器里面

Node.js发布于2009年5月，由Ryan Dahl开发，是一个基于 `Chrome V8` 引擎的 `JavaScript` 运行环境，使用了一个 **事件驱动**、非阻塞式I/O模型，[1] 让JavaScript运行在 **服务端** 的开发平台，它让JavaScript成为与 `PHP`、`Python`、`Perl`、`Ruby` 等服务端语言平起平坐的 **脚本语言**。[2]

node.js只是一个运行平台，这个平台上面运行的是JS代码，所以无论是ES5的代码还是ES6的代码都可以在上面运行

为什么前端需要学习node.js

在平常的工作当中很我前端程序认为前端不需要去了解后端，因为现在的主流开发方式都是前后端分离开发，前端不需要太多的了解后端，但是真正的工作里面，前端是要参与后端的开发的，如服务器的搭建，http请求的处理，SQL语句的编写，缓存的应用，session与cookie的验证，JWT的使用这些都依托于后端。以前的前端并不是不应用后端，它们只是使用php来完成后的操作，但是现在php已经慢慢的淡出了前端的视线，转而使用了另一种语言，这种技术就是 `node.js`

node.js只是一个运行环境，也就是一个平台，它里面运行的是js代码，而js代码对于前端来说是非常熟悉的，所以相较于php来说，它不用重新花费精力去了解学习一个新的语言，直接使用就可以了

同时相较于php来说，nodejs提供了强大的第三方插件与模块。同时nodejs还具备以下几个优点

1. 非阻涉的IO
2. 使用事件驱动
3. 天生的高并发能力
4. 单线程的执行

nodejs是让js运行在服务器的平台，所以一般情况下它是没有兼容性的，我们可以直接使用ES6、ES7都可以

nodejs里面只有 `ECMAScript`，没有DOM，更没有BOM

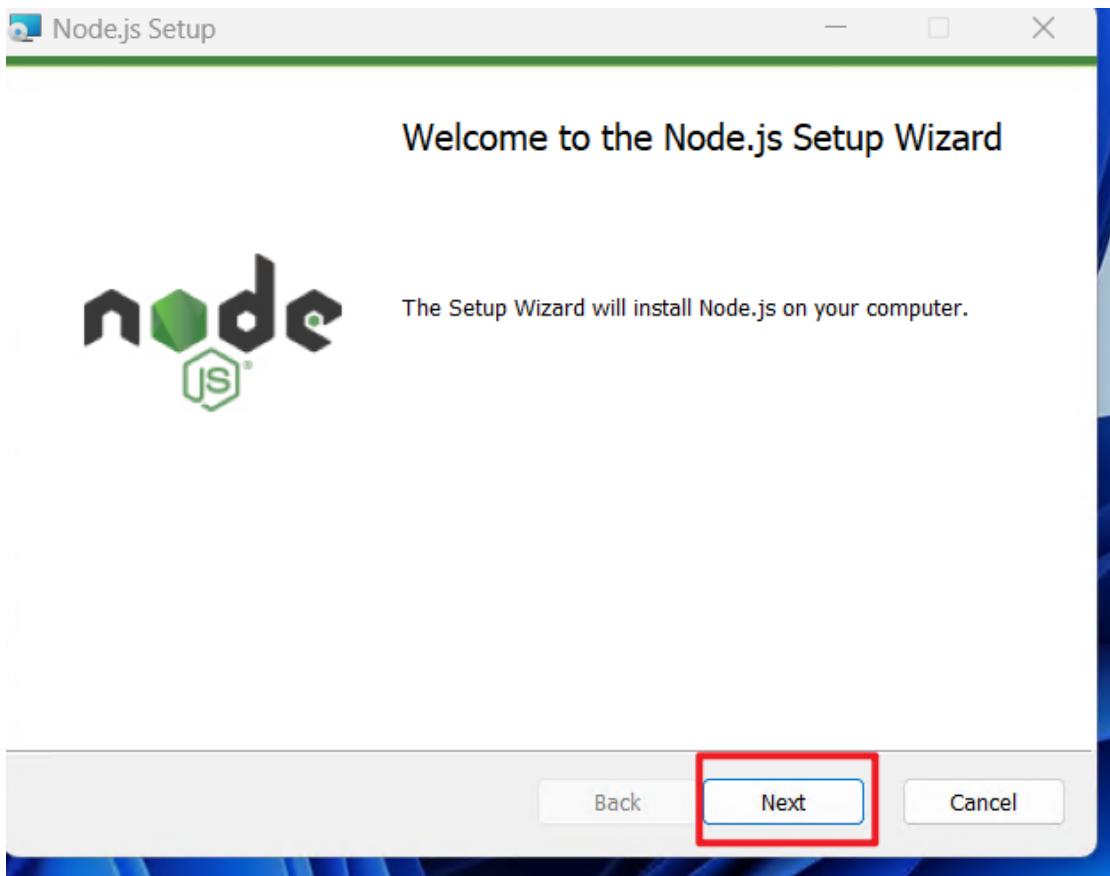
nodejs的安装

下载地址：[Index of /dist/ \(nodejs.org\)](#)

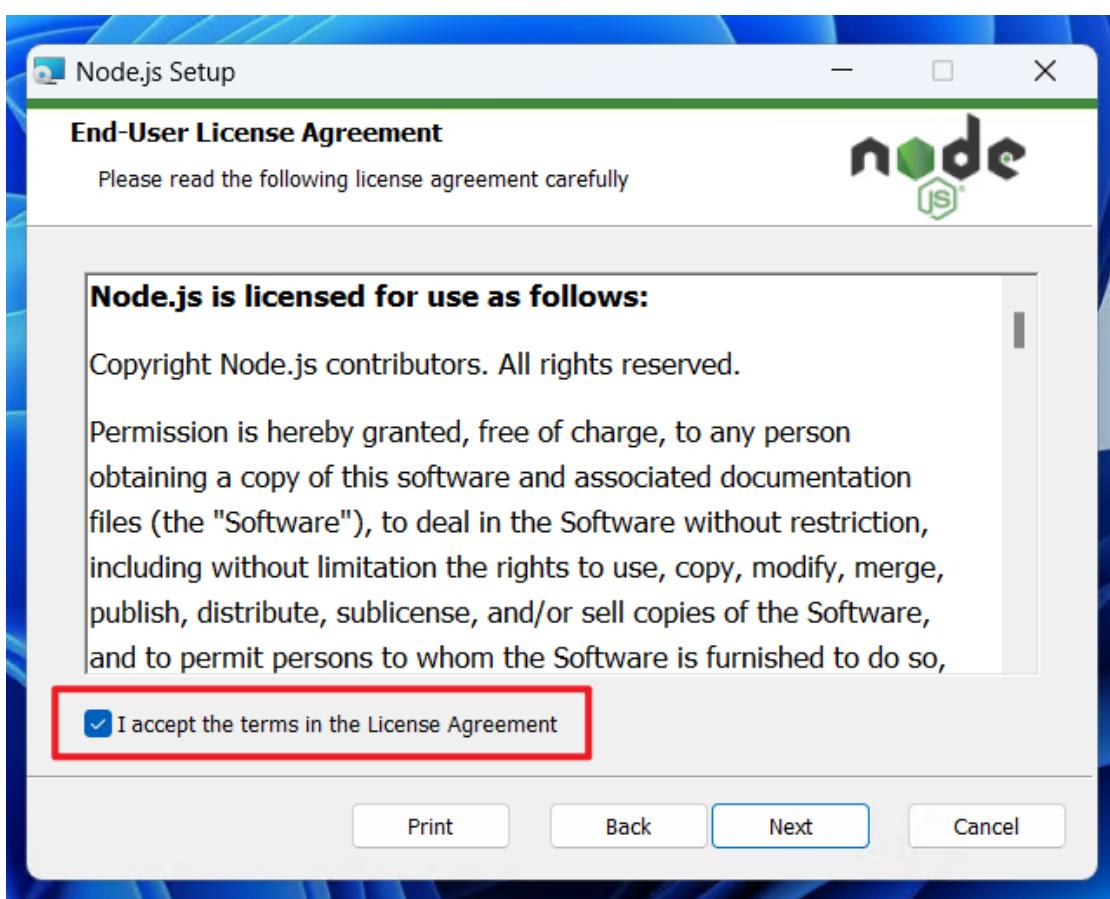
1. 双击下载好的安装包



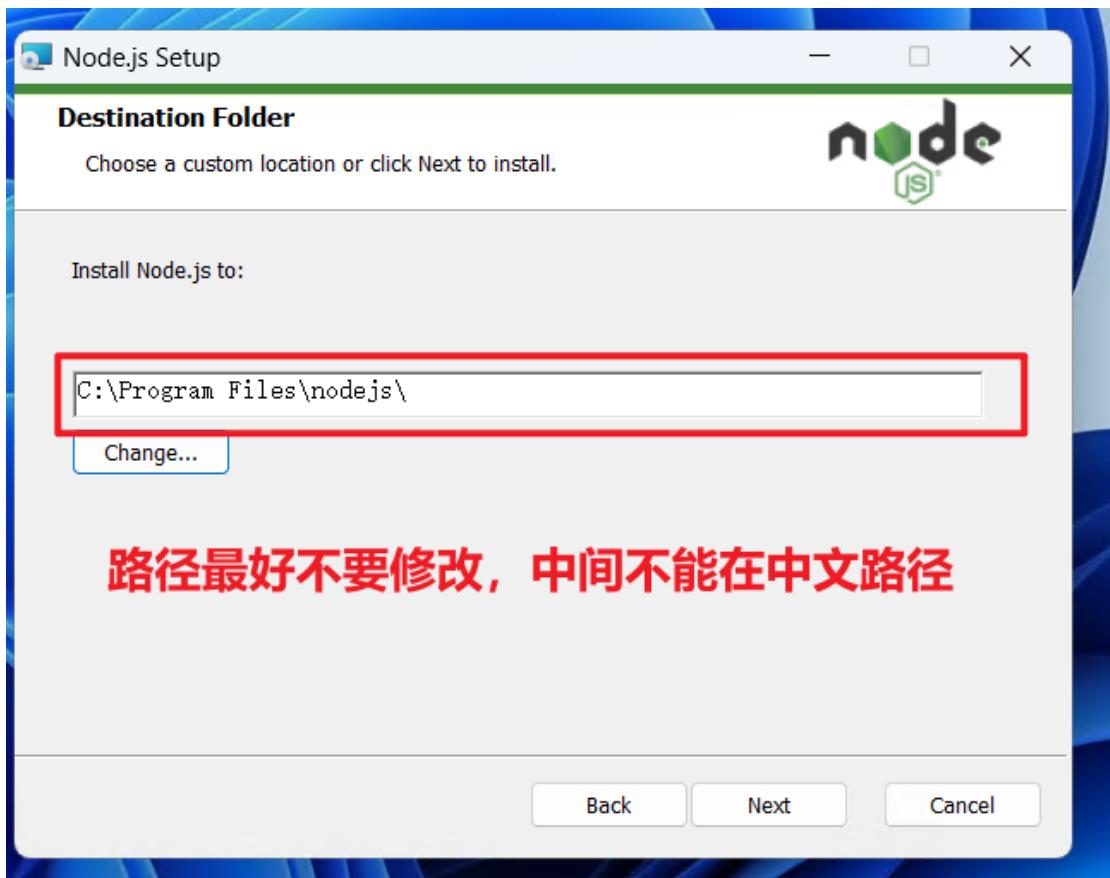
2. 点击next



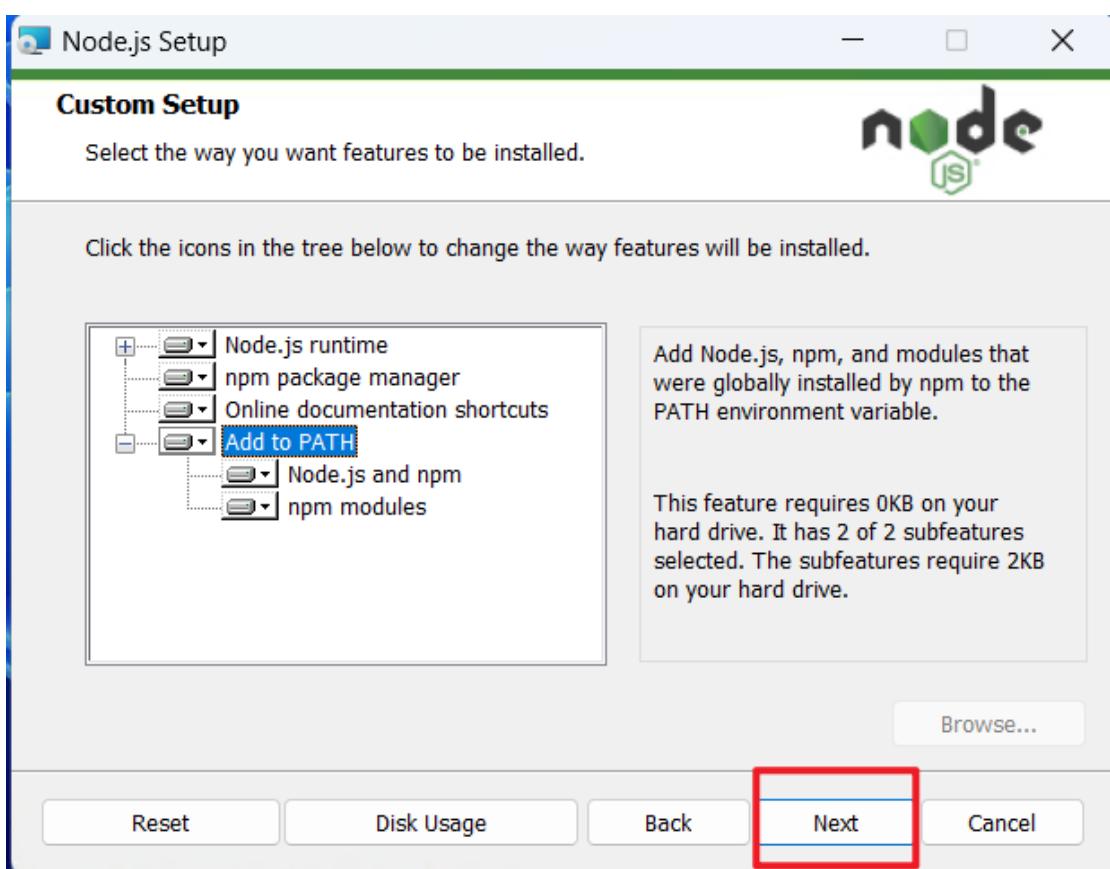
3. 同意



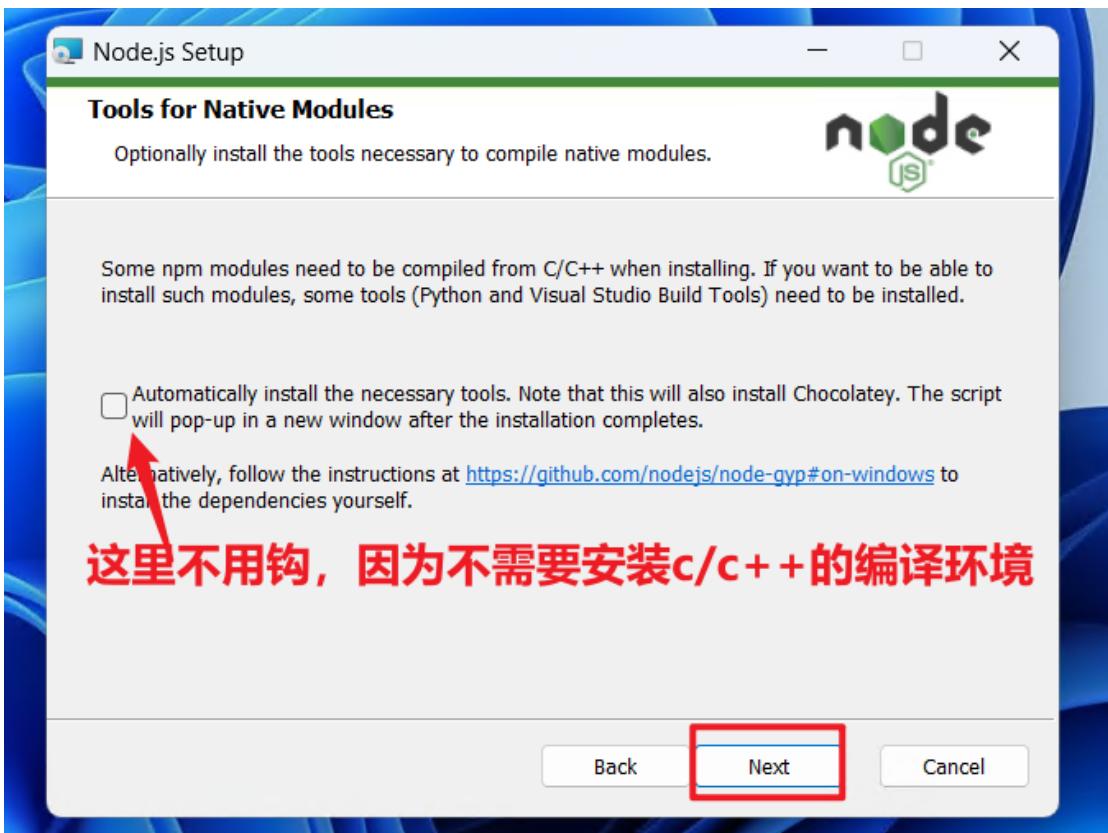
4. 设置安装路径



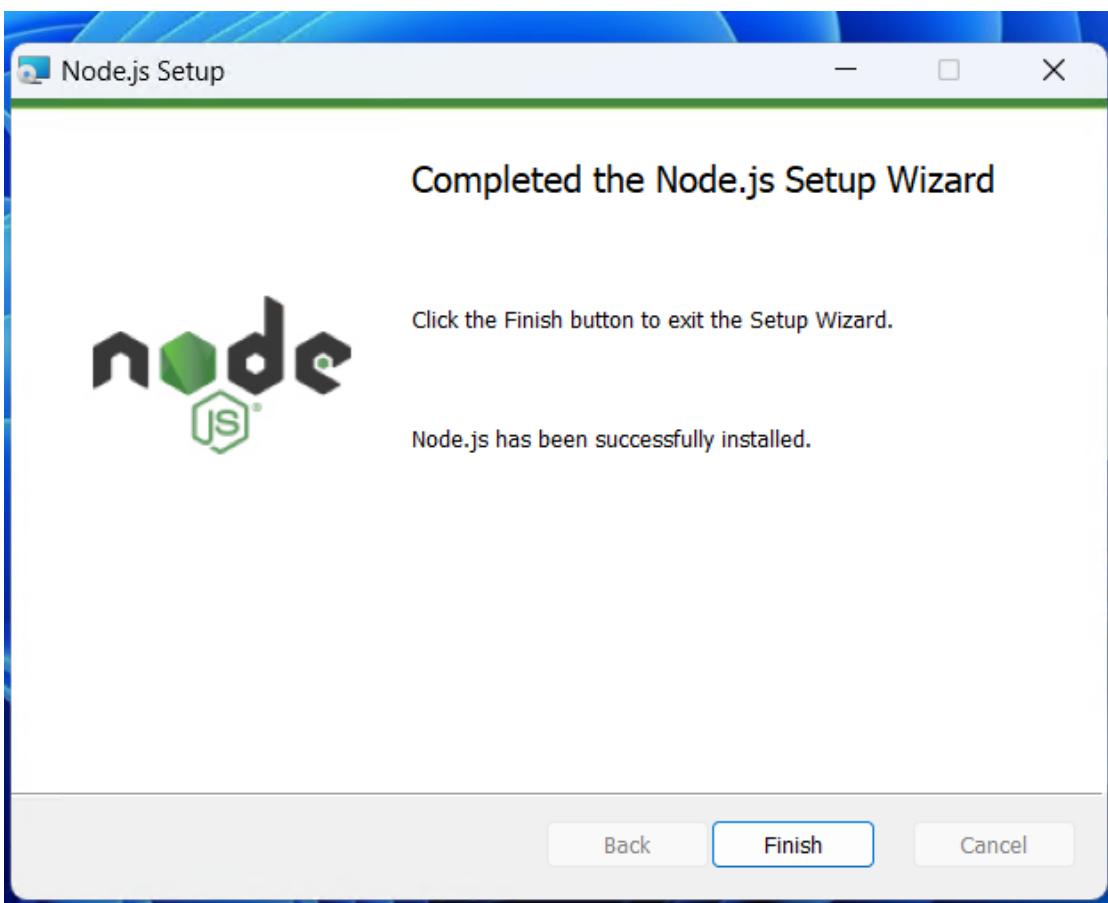
5. 设置相关信息



6. 不要选C、c++的编译环境



7. 安装完成



8. 验证是否安装成功

通过 `msi` 格式安装的环境成功以后，会自动帮我们添加到系统的环境变量当中（就不需要我们再次去配置环境变量了）

我们安装完成以后可以验证一下是否安装成功了

- 按 `win+r` 在弹出的容器里面输入 `cmd`

- 在 cmd 的窗口里面输入 node -v

```
C:\Users\YangBiao>node -v  
v16.5.0
```

如果出现了版本号，就代表安装成功了

附加：如果电脑是win10/win11的，需要加载powershell的远程执行权限，具体参考下面的教程

[Windows上PowerShell默认禁止运行 *.ps1 脚本原因与解决方法 \(softteam.xin\)](#)

node.js基础

node.js的运行

当我们安装好了node.js的环境以后，我们就可以在控制台输入 node -v 来查看环境了，如果显示正常则代表安装成功

安装成功以后，如果我们要运行node.js需要使用一些命令去完成

```
$ node 文件名.js
```

如

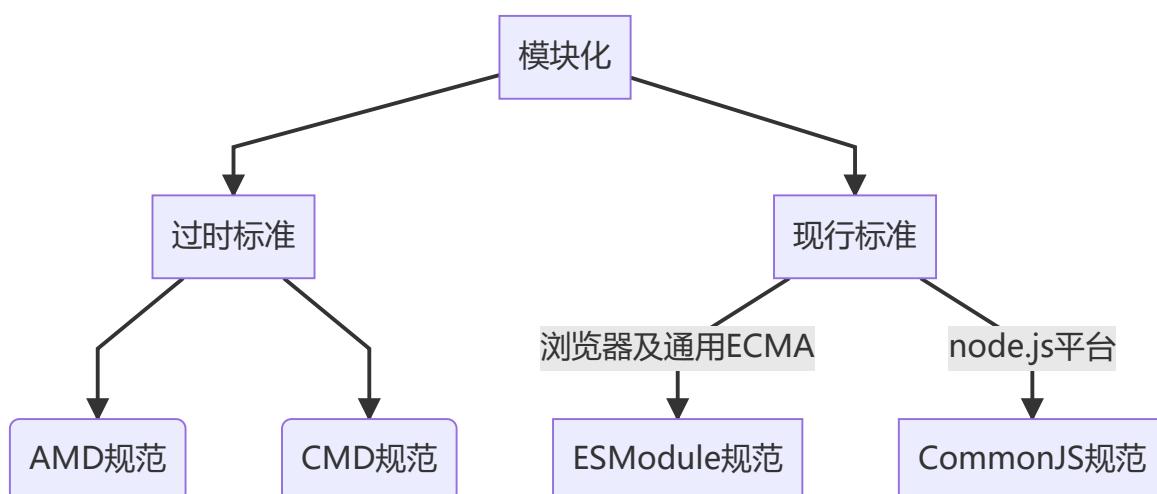
```
$ node 01.js
```

在node.js里面，只能执行 ES 的代码，不能执行 DOM 与 BOM

```
console.log("hello node.js");
console.log("标哥");
window.alert("hello 标哥");           //报错，因为node.js没有包含BOM里面的
```

CommonJs模块化

在之前的ES6里面，我们讲模块化的时候，提到过模块的分类应用



在现行的模块化标准里面有2个标准的，分为 `ESModule` 及 `CommonJS`，`ESModule` 主要针对的是 `ECMA` 及浏览器，而 `CommonJS` 主要针对的就 `node.js`

1. 在 `ESModule` 里面，我们使用 `export` 导出，使用 `import` 导入
2. 在 `nodejs` 里面，我们使用 `module.exports` 导出，使用 `require()` 导入

	导出	导入
<code>ESModule</code> 规范	<code>export</code> 关键字	<code>import</code> 关键字
<code>CommonJS</code> 规范	<code>module.exports</code>	<code>require()</code> 方法

现在我们先看下面的代码

Person.js

```
class Person{
    constructor(userName){
        this.userName = userName;
    }
    sayHello(){
        console.log(`我叫${this.userName}`);
    }
}

// export default Person;
module.exports = Person;
```

Student.js

```
// import Person from "./Person.js";
const Person = require("./Person.js");

class Student extends Person{
    constructor(userName,sex){
        super(userName);
        this.sex = sex;
    }
}

let s1 = new Student("张珊","女");
s1.sayHello();
```

当我们去执行 `node Student.js` 的时候，就不会报错了，因为我们已经改用 `CommonJS` 的规范来导出与导入

CommonJS的模块导入

在上一个章节我们已经了解了基本的 `CommonJS` 的导出与导入，我们也知道了如果要导入一个JS文件我们可以使用 `require()` 方法，现在仔细看一下它的使用

```
JS a.js
1 console.log(`我是a.js的文件`);

JS b.js
1 console.log(`我是b.js的文件 哈哈`);
2
3
4 // 导入了a.js的代码，相当于把a.js的代码拿到这里在运行
5 require("./a.js");
6
7
8
9
```

require其实就相当于把一个JS文件拿到另一个JS文件里面去执行

```
JS a.js
1 console.log(`我是a.js的文件`);

JS b.js
1 console.log(`我是b.js的文件 哈哈`);

Drive
  100601 2022-10-06 10:38 文件夹
  100602 2022-10-06 10:45 文件夹
```

同时我们发现一个特点，当我们执行多次导入的时候，最终 a.js 里面的代码也只会执行一次

require()方法在导入一个模块以后会把这个模块缓存下来，下次再导入的时候就直接从缓存里面拿出来使用，所以在上面的代码当中虽然我们导入了5次，但是真正导入的只有第一次导入，后面的几次导入都是直接从缓存里面拿东西

CommonJS的缓存

如果希望一个模块被导入以后，不要有缓存，可以在一个模块的最后面添加下面的代码

```
delete require.cache[module.filename];
```

```
JS a.js
1 console.log(`我是a.js的文件`);
2
3
4 // 删除当前这个模块的缓存
5 delete require.cache[module.filename];
6

JS b.js
1 console.log(`我是b.js的文件 哈哈`);

Drive
  100601 2022-10-06 10:38 文件夹
  100602 2022-10-06 10:45 文件夹
```

CommonJS模块导出

导入与导出是一对，在nodejs里面如果想导出可以使用下面的2种方式

1. `module.exports` 直接导出
2. `exports` 指针导出

首先我们先来看一下最基本的导入与导出

```
a.js
1 let userName = "标哥哥";
2
3 // 导出了userName
4 module.exports = userName;

JS b.js
1 const userName = require("./a.js");
2
3 console.log(`我导入的是${userName}`);
```

上图就是最基本的导入与导出

在每个 node.js 运行的文件里面，默认都会一个属性叫 `module.exports`。这个是专门用于负责导出模块的，同时还有一个 `exports` 指向了 `module.exports`

当前先不考虑 `exports`

module.exports

在每个文件里面都会有这个东西，它是专门负责导出的对象，叫 `module.exports`，我们直接打印 `module.exports` 发现它是一个空对象 {}

```
JS a1.js ...
1 let userName = "张珊";
2 let age = 18;
3
4 // 现在怎么样同时导出这两个变量
5 // let obj = {
6 //   userName: userName,
7 //   age: age
8 // }
9 // 1. 导出一对
10 // 2. exports 指针导出
11 // 解构
12 let obj = {
13   userName,
14   age
15 }
16 // 导出了userName和age
17 // module.exports = obj;
18
19 module.exports = obj;
```

```
JS b1.js ...
1 const obj = require("./a1.js");
2
3 console.log(obj.userName);
4 console.log(obj.age);
```

如果我们要导出多个变量，我们可以把它封装成对象，然后再导出。但是这么做还不够简洁，我们进一步简化

```
JS a2.js ...
1 let userName = "张珊";
2 let age = 18;
3
4 // 在每个 node.js 运行的文件里面，默认都会一个属性叫 module.exports 负责导出模块的，同时还有一个 exports 指向了 module.exports
5
6 // module.exports 专门负责导出
7 // 解构赋值
8 module.exports = {
9   userName,
10   age
11 };
12 // module.exports 发现它是一个空对象 {}
```

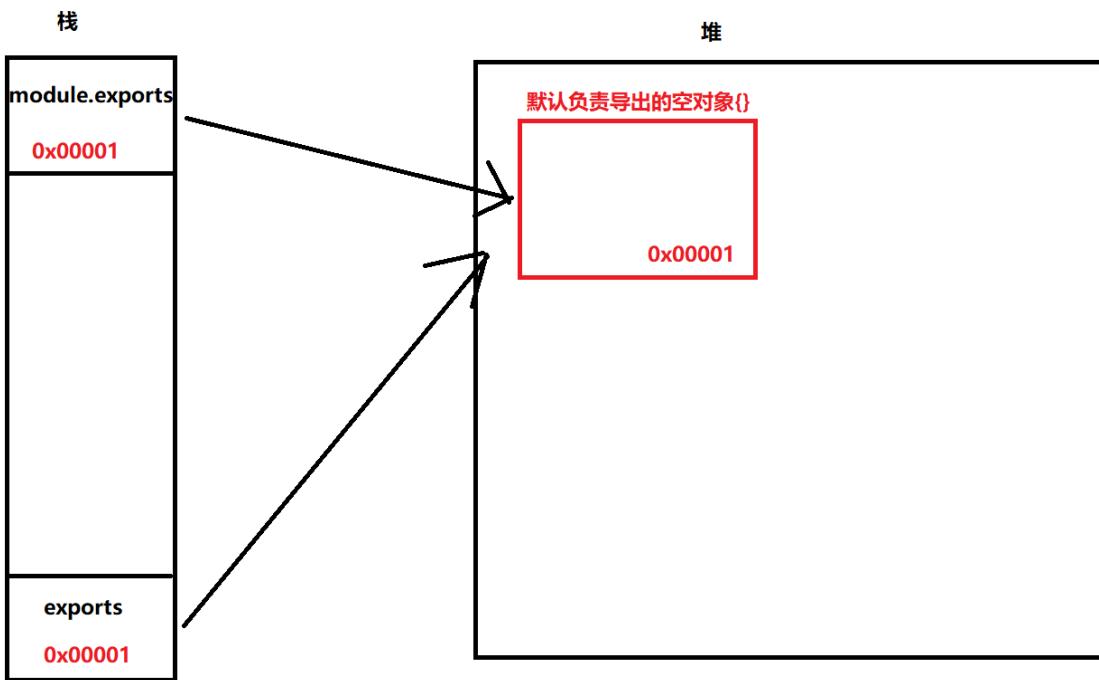
```
JS b2.js ...
1 const {userName, age} = require("./a2.js");
2
3 console.log(userName);
4 console.log(age);
```

如果要导出多个变量，我们可以使用解构的赋值与取值的方式快速的完成

exports

在CommonJS的模块化规范里面，负责导出的只有一个叫 `module.exports`，但是还有一个默认指向了 `module.exports`，这个就是 `exports`

我们现在通过内存图来看下看



```
console.log(module.exports === exports);
```

场景一

```
JS a.js
1 let userName = "张鹏";
2 let age = 18;
3 module.exports = userName;
4 exports = age;

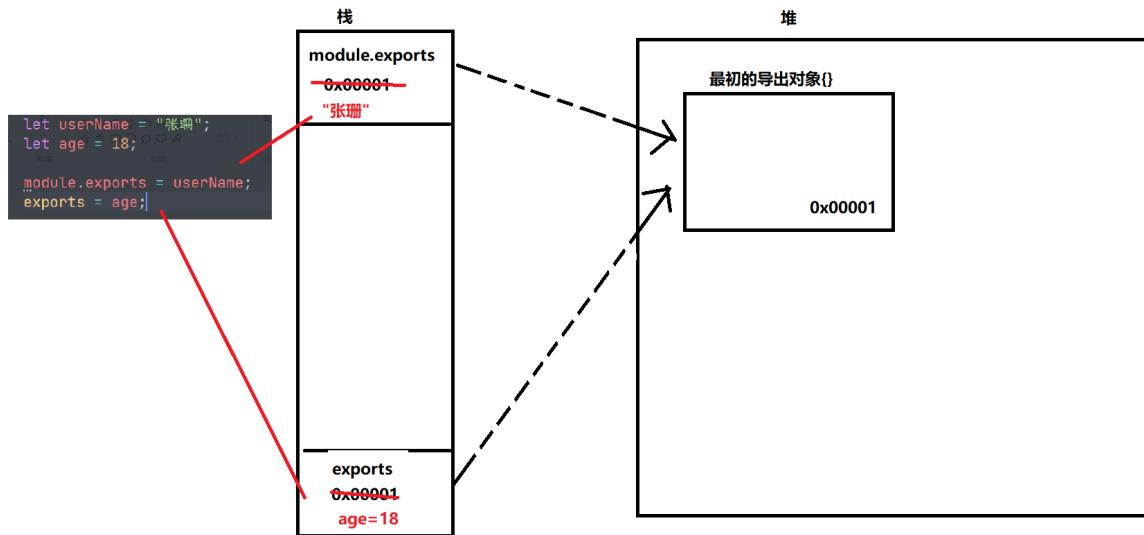
JS b.js
1 const obj = require("./a.js");
2 // 这个时候的obj到底是谁?
3
4 console.log(obj);
```

在上面的代码里面，我们通过 `module.exports` 导出，又通过 `exports` 导出，请问 `b.js` 导入的 `obj` 到底是谁？

为了弄清楚上面的东西，我们一定要谨记几个点

1. 真正负责导出的是 `module.exports`
2. `exports` 指向了最初的 `module.exports` 的堆里面的地址

现在我们根据上面的代码来画一个内存图



始终记得一句话，直接负责导出的是 `module.exports`，所以 b.js 导入的其实就是 `userName` 的值，结果就是“张珊”

场景二

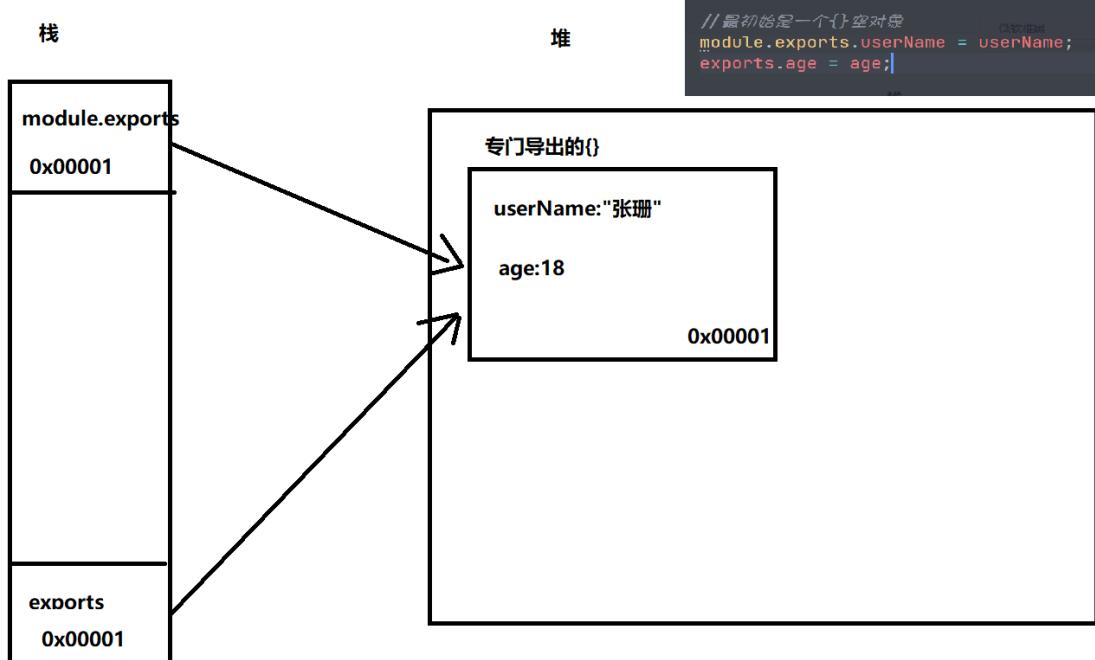
Two code editors are shown:

a.js:

```
1 let userName = "张珊";
2 let age = 18;
3 //最初是一个{}空对象
4 module.exports.userName = userName;
5 exports.age = age;
```

b.js:

```
1 const obj = require("./a.js");
2 //这个时候obj是谁
3 console.log(obj);
```



真正负责导出的仍然是 `module.exports`，这个对象被修改了2次，第一次是通过 `module.exports.userName` 来修改的，第二次是通过 `exports.age` 来修改的

所以导入的obj结果就是有2个属性

场景三

JS a1.js

```

1 console.log(module.exports === exports);
2
3 exports.userName = "张三";
4 exports.age = 18;
5

```

JS b1.js

```

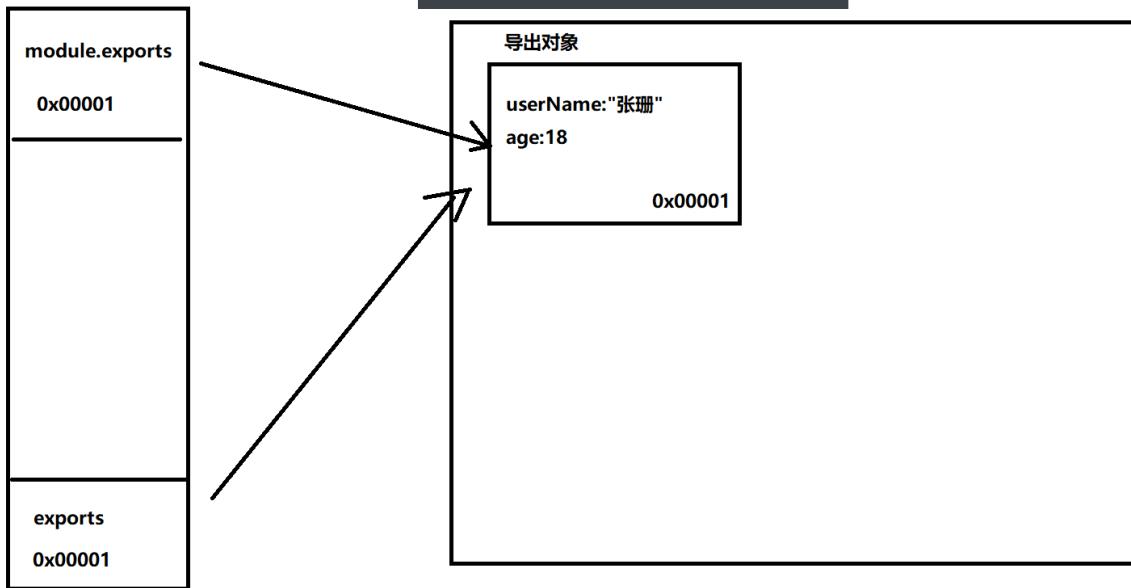
1 const obj = require("./a1.js");
2
3 //obj又是谁
4 console.log(obj);

```

```

exports.userName = "张三";
exports.age = 18;

```



场景四

JS a2.js

```

1 //最初时module.exports和exports指向了同一个对象
2
3 exports.userName = "标哥";
4 exports.age = 18;
5
6
7 module.exports = {
8   hobby:"看书"
9 }
10

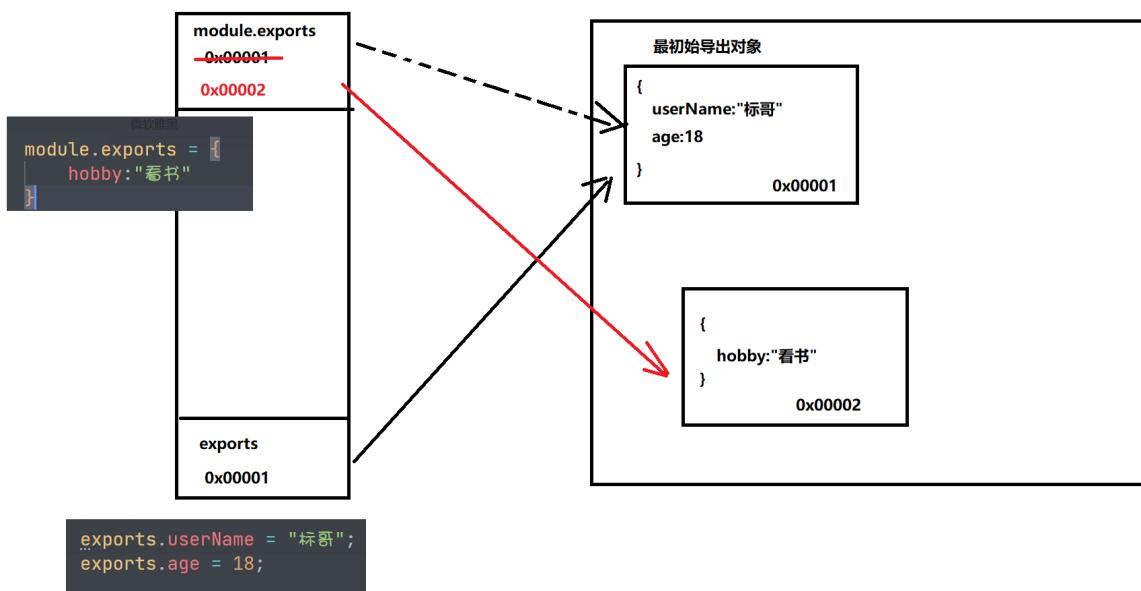
```

JS b2.js

```

1 const obj = require("./a2.js");
2
3 //obj是谁
4 console.log(obj);

```



真正负责导出的是 `module.exports`，所以最终得到的结果是 `hobby:看书`

为了避免后期开发的时候有歧意，我在后面的代码当中优先使用 `module.exports`

nodejs内置模块

node.js做为一个运行平台，它本身就内置了很多的模块，主要使用的模块有以下几个

1. `path` 路径模块
2. `fs` 文件系统模块
3. `os` 系统模块
4. `http` 网格请求模块
5. `net` 网络模块

path模块

这个模块主要是用于处理电脑上面的路径的，因为 `node.js` 是运行在本地的路径上面，所以每个js文件的路径都会不一样，所处的文件夹可能也不一样，如果要去处理这些路径的拼接，判断等问题就需要使用到这个模块

在使用这个模块之前，先了解2个内置的与路径相关的变量

1. `__dirname` 代表当前js运行的目录的路径
2. `__filename` 代表当前js运行的文件路径

```
//__dirname代表当前js运行的文件夹的路径
console.log(__dirname);
//D:\杨标的工作文件\班级教学笔记\H2204\1007\code\100701

//__filename代表当前js运行文件路径
console.log(__filename);
//D:\杨标的工作文件\班级教学笔记\H2204\1007\code\100701\01.js
```

```
const path = require("path");
```

1. `path.join()` 方法，方法可以将路径进行拼接

```
let str1 = path.join(__dirname, "a.txt");
console.log(str1);

let str2 = path.join(__dirname, "../..");
console.log(str2);
```

2. `path.extname()` 方法，获取一个路径上面的文件的后缀名

```
let str1 = path.extname(__filename);
console.log(str1); //.js

let str2 = path.extname("D:\\杨标的工作文件\\班级教学笔记\\H2103\\笔记整理\\02DOM+BOM.pdf");
console.log(str2);
```

3. `path.isAbsolute()` 方法，判断某一个路径是否是绝对路径

相对路径，相当于当前目录来进行设置

./ 相当于当前目录的当前目录
../ 相当于当前目录的上级目录
.img 相当于当前目录下面的img目录
..js 相当于当前目录的上级目录下面的js目录

绝对路径，一定是一个完整的路径

C:\windows
D:\software\yangbiao

```
let flag = path.isAbsolute("../");
console.log(flag);           //false

let flag2 = path.isAbsolute("C:\\windows");
console.log(flag2);          //true
```

4. `path.resolve()` 方法，将一个相对路转换成绝对路径

```
//相对路径
let flag = path.isAbsolute("../");           //false
console.log(flag);
//把相对路径转换成绝对路径
let str1 = path.resolve("../");
console.log(str1);
```

fs模块

重点，这个模块FS的全称有个意思，第一理解为 `File System` 文件系统，第二种理解叫 `File Stream` 文件流

这个模块也是nodejs的内置模块，专门用于处理路径下面的文件（有了路径，我们就可以操作文件与文件夹）

```
const fs = require("fs");
```

1. `fs.existsSync()` 方法，判断某一个文件或文件夹是否存在（判断某一个路径是否存在），`true` 代表在，`false` 代表不存在
2. `fs.rmdirSync()` 方法，删除一个文件夹，如果这个文件夹不为空，则不能删除
3. `fs.unlinkSync()`，删除一个文件，这个方法可以删除一个文件
4. `fs.copyFileSync()` 方法，复制一个文件
5. `fs.renameSync()` 方法，重命名一个文件
`fs.renameSync()` 方法本意是文件重命名，但是如果文件在重命名的时候不在同一个文件夹，则相当于剪切操作
6. `fs.mkdirSync()` 创建一个文件夹
7. `fs.readdirSync()` 读取一个文件夹的信息，它会返回一个数组，这个数组里面包含了所有的文件及文件夹的信息
8. `fs.statSync()` 读取某一个路径的信息，结果如下

```

Stats {
  dev: 44114680,
  mode: 33206,
  nlink: 1,
  uid: 0,
  gid: 0,
  rdev: 0,
  blksize: 4096,
  ino: 2533274792094419,
  size: 245, //文件大小
  blocks: 0,
  atimeMs: 1665105588523.2388,
  mtimeMs: 1665104647159.5562,
  ctimeMs: 1665105593992.7217,
  birthtimeMs: 1665105588444.7112,
  atime: 2022-10-07T01:19:48.523Z,
  mtime: 2022-10-07T01:04:07.160Z, //修改时间
  ctime: 2022-10-07T01:19:53.993Z, //创建时间
  birthtime: 2022-10-07T01:19:48.445Z //创建时间
}

```

在读取的结果里面，还有两个方法一定要注意

- `isFile()` 用于判断当前路径是否是文件
- `isDirectory()` 用于判断当前路径是否是文件夹

```

const fs = require("fs");
let info1 = fs.statSync("./01.js");
// console.log(info1);

console.log(info1.isFile()); //判断是否是一个文件
true

console.log(info1.isDirectory()); //判断是否是一个文件夹
//false

```

9. `fs.readFileSync()` 当前方法可以读取一个文件的内容

```

const fs = require("fs");
// 读取abc.txt的文件

//第一步：判断路径是否存在
let p1 = path.join(__dirname, "./abc.txt");
if (fs.existsSync(p1)) {
  //存在
  //第二步：判断这个路径是否是一个文件
  let p1Info = fs.statSync(p1);
  if (p1Info.isFile()) {
    //说明是文件，可以开始读了
    let result = fs.readFileSync(p1);
    console.log(result.toString());
  }
}
else {
  console.log("路径不存在");
}

```

在上面的代码里面，我们读的是文本文件

如果我们读取是其它类型的文件，还可以把读取的结果转换为特定的要求

```
const path = require("path");

//第一步：构建这个路径
let p1 = path.join(__dirname, "./img/w08.jpg");
//第二步：判断路径是否存在
if (fs.existsSync(p1)) {
    //第三步：判断这个路径是否文件
    let p1Info = fs.statSync(p1);
    if (p1Info.isFile()) {
        let result = fs.readFileSync(p1);
        console.log(result.toString("base64")); //将它转换成了base64
    }
}
else {
    console.log("路径不存在");
}
```

10. `fs.writeFileSync()` 将一个内容写入到文件

这个文件与上面的方法是相对应的，一个是读，一个是写

```
const path = require("path");
const fs = require("fs");

//第一步：构建路径
let p1 = path.join(__dirname, "./abc.txt");
fs.writeFileSync(p1, "标哥在讲课....");
console.log("写入完成");
```

上面的代码是一个最基本的最入过程，我们还可以有更高级的写法

```
/**
 * 将bbb.txt里面的base64字符串转换成图片
 */

const fs = require("fs");
const path = require("path");

//第一步：构建路径
let p1 = path.join(__dirname, "./bbb.txt"); //base64字符串
let img1 = path.join(__dirname, "./img/dijia.png"); //最终要生成的图片的
路径

//第二步：判断路径是否存在
if (fs.existsSync(p1)) {
    //第三步：判断这个路径是否是文件
    let p1Info = fs.statSync(p1);
    if (p1Info.isFile()) {
        let result = fs.readFileSync(p1);
        //第四步：将base64转换成图片
        fs.writeFileSync(img1, result.toString(), { encoding: "base64" });
        console.log("写入成功");
    }
}
```

```
        }
    }
else {
    console.log("路径不存在")
}
```

os模块

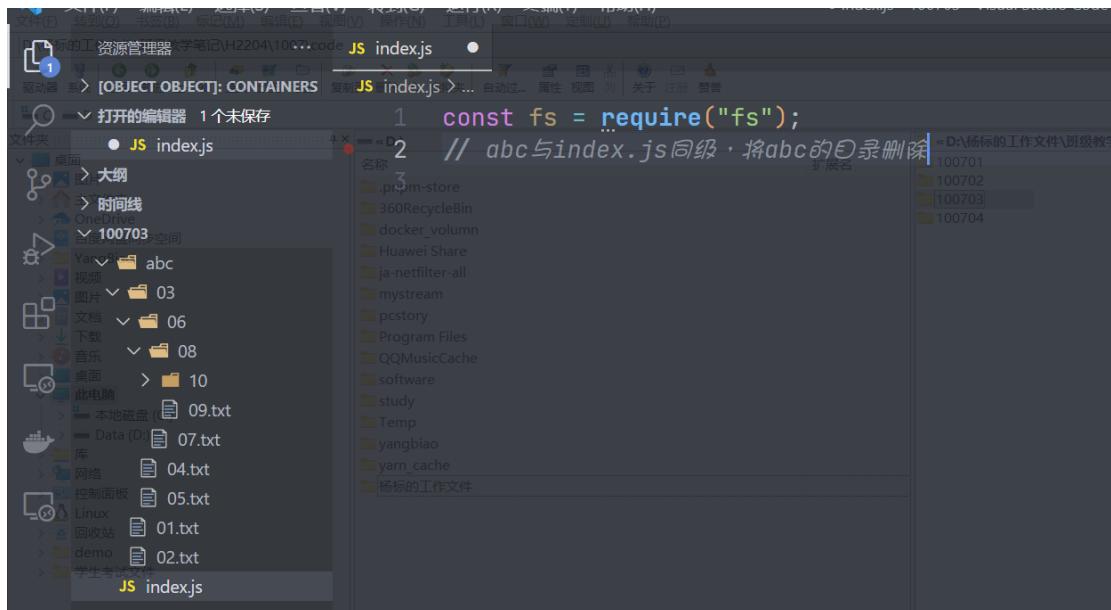
os:operation system操作系统

os模块是与系统相关的模块，可以通过调用里面的方法来实现获取系统的相关信息，如CPU，网卡，临时目录，内存等

1. `os.version()` 返回当前系统的版本信息
2. `os.hostname()` 返回目前电脑的主机名称
3. `os.cpus()` 返回当前电脑的cpu信息
4. `os.totalmem()` 返回当前电脑的总内存信息
5. `os.freemem()` 返回当前电脑的剩余内存
6. `os.networkInterfaces()` 返回当前电脑的网卡信息
7. `os.userInfo()` 返回当前登录到系统的用户信息
8. `os.homedir()` 返回家目录的地址
9. `os.tmpdir()` 当前系统的临时目录

课堂练习

1. 现有如下的文件结构，请编写代码删除目录



提示：

1. 会用到递归
2. `fs.existsSync().fs.readdirSync(), fs.unlinkSync(), fs.rmdirSync(), path.join(), fs.statSync(), isFile() / isDirectory()` 用到这些方法

```

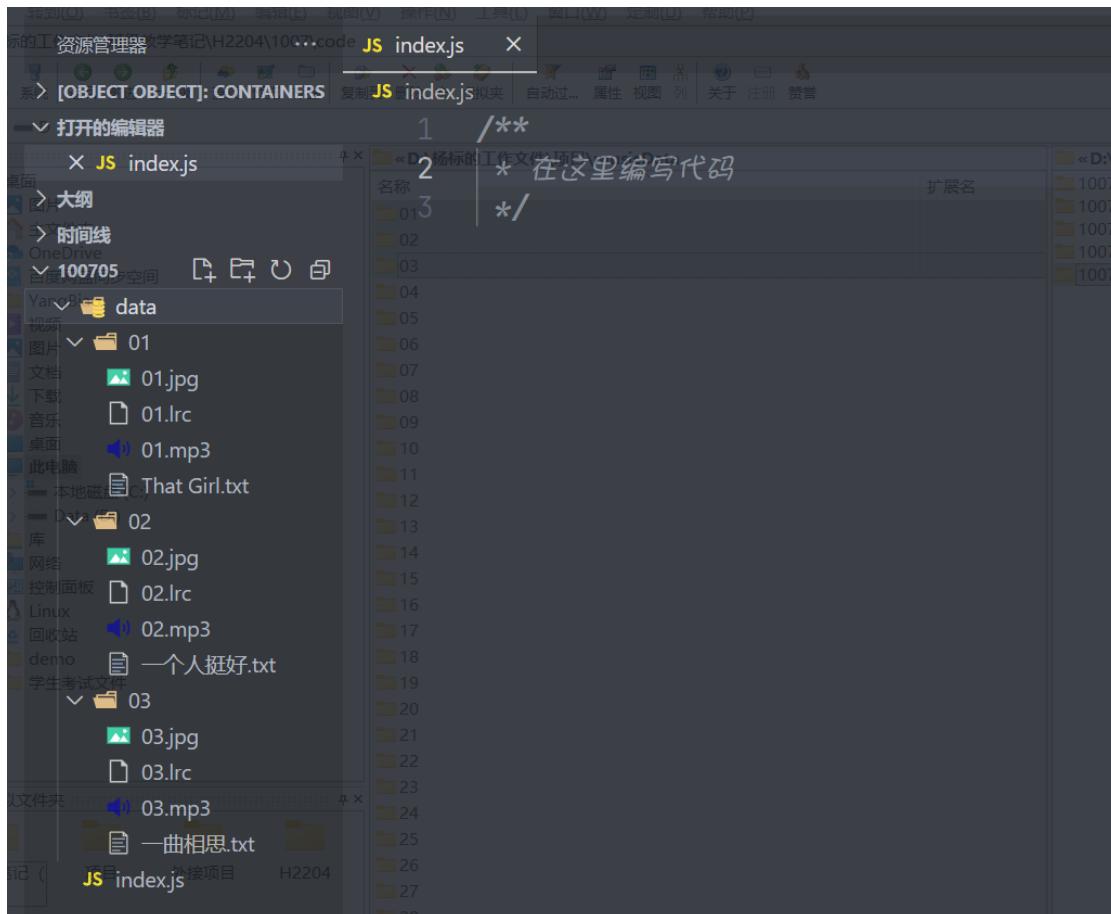
const fs = require("fs");
const path = require("path");

//我给你一个路径你，你帮我删除
const deleteAll = p => {
    //第一步：先判断这个路径是否存在
    if (fs.existsSync(p)) {
        //第二步：判断这个路径是文件夹还是文件
        let pInfo = fs.statSync(p);
        if (pInfo.isFile()) {
            //第三步：如果是文件，我们就直接删除
            fs.unlinkSync(p);
        }
        else if (pInfo.isDirectory()) {
            let arr = fs.readdirSync(p);
            for (let item of arr) {
                // 把item和之前的路径接起来，形成一个的路径
                let newPath = path.join(p, item);
                deleteAll(newPath);      //递归
            }
            //把所有的文件都删除以后，我们就可以删除这个文件夹了
            fs.rmdirSync(p);
        }
    }
}

deleteAll(path.join(__dirname, "./abc"));

```

2. 根据下面的文件结构，生成指定的文件



读取data目录，生成一个 `musicData.txt` 的文件，生成的内容如下

```
[  
  {  
    "picPath": "01/01.jpg",  
    "lrcPath": "01/01.lrc",  
    "lrcText": "[00:00.000]That Girl - Olly Murs\n[00:00 ... ",  
    "musicPath": "01/01.mp3",  
    "musicName": "That Girl"  
  },  
  {  
    "picPath": "02/02.jpg",  
    "lrcPath": "02/02.lrc",  
    "lrcText": "[00:00.000]一个人挺好 - 孟颖\n[00:05.320]词: 杨小壮.....",  
    "musicPath": "02/02.mp3",  
    "musicName": "一个人挺好"  
  },  
  {  
    "picPath": "03/03.jpg",  
    "lrcPath": "03/03.lrc",  
    "lrcText": "[00:00.000]一曲相思 - 半阳\n[00:02.690]词: .....",  
    "musicPath": "03/03.mp3",  
    "musicName": "一曲相思"  
  }  
]
```

提示

1. 会用到

```
fs.readdirSync(), fs.readFileSync(), fs.writeFileSync(), JSON.stringify()  
)
```

```
/**  
 * 在这里编写代码  
 */  
  
const path = require("path");  
const fs = require("fs");  
  
const getData = p => {  
  let result = [];  
  //第一步：判断路径是否存在，第二步：判断是否是一个文件夹  
  if (fs.existsSync(p) && fs.statSync(p).isDirectory()) {  
    let arr = fs.readdirSync(p);  
    for (let item of arr) {  
      let obj = {};  
      // 得到了新路径  
      let p2 = path.join(p, item);  
      let arr2 = fs.readdirSync(p2);  
      for (let item2 of arr2) {  
        //要获取每一个文件的后缀名  
        let extname = path.extname(item2);  
        switch (extname) {  
          case ".jpg":  
            obj[item] = item2;  
        }  
      }  
      result.push(obj);  
    }  
  }  
  return result;  
};
```

```

        obj.picPath = `${item}/${item2}`;
        break;
    case ".lrc":
        obj.lrcPath = `${item}/${item2}`;
        obj.lrcText = fs.readFileSync(path.join(p, item,
item2)).toString();
        break;
    case ".mp3":
        obj.musicPath = `${item}/${item2}`;
        break;
    case ".txt":
        obj.musicName = item2.replace(extname, "");
        break;
    }
}

result.push(obj);

}
}

return result;
}

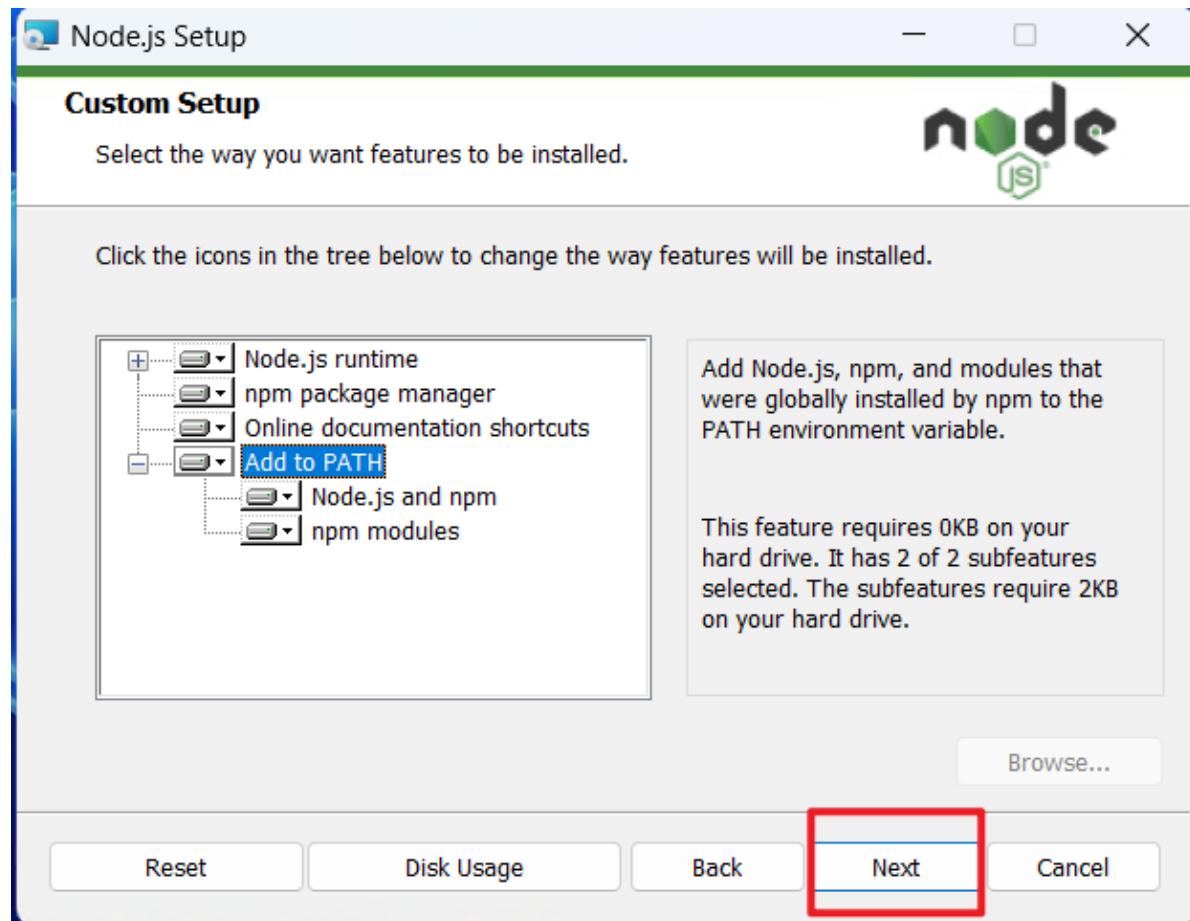
let result = getData(path.join(__dirname, "./data"));
let resultJsonStr = JSON.stringify(result);
fs.writeFileSync(path.join(__dirname, "./musicData.txt"), resultJsonStr);
console.log("写入完成");

```

nodejs第三方模块

之前我们所使用的 `path`, `fs` 等模块都是nodejs提供给我们的内置模块，可以方便我们直接使用。其实nodejs还有大量的第三模块来给我们使用，如进行数据请求我们使用 `axios`，如果抓包我们要使用 `cheerio`，如连接数据库我们要使用 `mysql/mysql2`，如使用MVC开发我们就要使用 `express`,`KOA`,`nest.js` 等

第三方模块有一个专门的管理员工具，叫 `npm`，在安装node.js的时候就已经安装了，如下图所示



这个东西是专门用于对第三方的模块进行管理的

1. npm 理解为 network package manager 网络包管理工具
2. npm 理解为 node.js package manager node.js的包管理工具

npm工具

nodejs是以文件夹为单位来管理项目的，所以同学们在学习node.js的时候不要在vscode下面同时打开多个项目文件夹

nodejs是以文件夹为单位来创建项目的，也是来管理项目的，那么nodejs是怎么样就认为这个文件夹就是一个项目呢

npm初始化

nodejs如果想以某一个文件夹为单位来管理项目，一定要对这个文件夹（项目）进行初始化，初始化以后，nodejs就认这个文件夹就是一个项目了

初始化的命令

```
$ npm init
```

当执行完一系列的操作以后，我们就可以看到在当前的文件夹下面会多出一个 package.json 的文件，这个文件记录了你初始化的时候一些信息

```
{
  "name": "100706",
  "version": "1.0.0",
  "description": "标哥哥的项目",
  "main": "index.js",
```

```
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1"
},
"keywords": [
  "管理系统",
  "标哥",
  "第一个项目"
],
"author": "杨标",
"license": "ISC"
}
```

当一个文件夹的下面有了 `package.json` 以后，系统不会认为这个文件夹是一个node.js的项目，不再仅仅只是一个文件夹了

package.json介绍

这个文件记录了当前项目的基本信息

- `name` 代表当前包的名称，也就是nodejs项目的名称
- `version` 代表当前项目的版本
- `description` 代表当前项目的描述信息
- `main` 代表当前项目的入口文件，可以告诉系统当前的项目从哪个文件开始启动
- `scripts` 脚本配置，后期我们可以通过这个东西来实现项目的快速启动
- `keywords` 项目的关键字，后期我们把项目发布到网上去以后别人可以通过这个关键字来搜索这个项目
- `author` 项目的作者
- `license` 版权

如果我们在初始化项目的时候希望快速的生成 `package.json` 文件，可以直接使用下面的命令

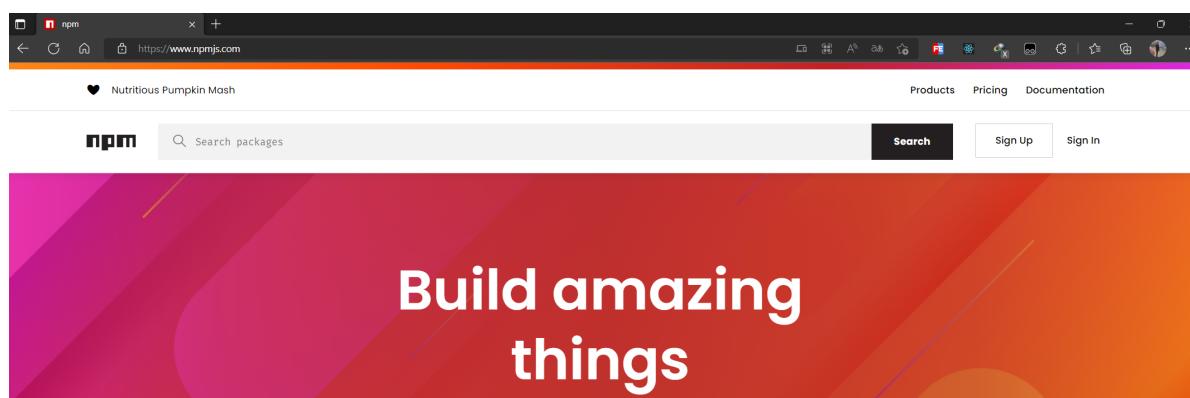
```
$ npm init --yes
```

它会直接以默认的形式帮我们生成 `package.json`，后期我们可以再去修改这个文件

npmjs远端仓库

有一个网站，上面保留了我们所有的第三方模块，这个网站就是

[npm \(npmjs.com\)](https://www.npmjs.com)



我们可以根据自己的需求在这个网站找到适合于自己的第三方模块，这个东西相当于java里面的 `maven`

npm info查看包的信息

如果我们想从远程服务器npmjs上面下载某一个包，应该先查看一下有没有这个包，并且了解一下这个包的相关信息

```
$ npm info 包名称
```

如我们输入下面的命令就可以查看 `jQuery` 的信息

```
$ npm info jquery
```

```
YangBiao@YB-Huawei D:\杨标的互作文件\班级教学笔记\H2204\1007\code\100706 => npm info jquery
jquery@3.6.1 | MIT | deps: none | versions: 54
JavaScript library for DOM operations
https://jquery.com

我们可以根据自己的需求在這個网站找到适合于自己的第三方模块，这个东西相当于java里面的maven

keywords: jquery, javascript, browser, library

dist
  npm info 查看包的信息
  .tarball: https://registry.npmmirror.com/jquery/-/jquery-3.6.1.tgz
  .shasum: fab0408f8b45fc19f956205773b62b292c147a16
  .integrity: sha512-opJe04nCucVnsjiX0E+/PcCgYw9Gwpvs/a6B1LL/lQhwWwpbVEVYDZ1FokFr8PRc7ghYlrFPuyHuiiDNTQxmcw==
  .unpackedSize: 1.3 MB

  $ npm info 包名称
  maintainers:
    - openjsfoundation <npm@openjsf.org>
    - dmethvin <dave.methvin@gmail.com> jQuery 的信息
    - timmywil <4timmywil@gmail.com>
    - mgol <m.goleb@gmail.com>
  $ npm info jquery
  dist-tags:
  beta: 3.6.1    latest: 3.6.1

  published a month ago by timmywil <4timmywil@gmail.com>
```

npm install安装包

当我们通过 `npm info` 查看了某一个包的信息以后，我们现在想把这个包下载下来，怎么办呢？

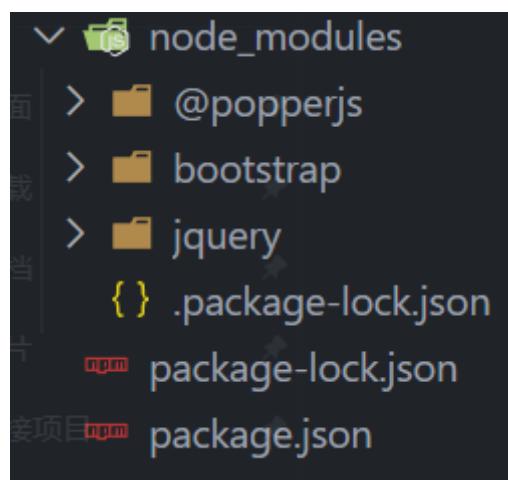
```
$ npm install 包名称
```

如

```
$ npm install jquery
```

上面的代码就下载了jquery的安装包，如果我们还想安装其它的包也一样的

```
$ npm install bootstrap
```



这个时候我们可以看到，当某一个安装成功以后就会在 `node_modules` 的目录下面显示出来

每次下载完了以后，它会把安装信息记录在 `package.json` 里面

```
"License": "ISC",
"dependencies": {
    "bootstrap": "^5.2.2",
    "jquery": "^3.6.1"
}
```

--save记录安装信息

在上面我们通过 `npm install` 安装某一个包的时候，这个包从服务器下载到了本地项目，后期我们的项目会越来越大，第三方的包也会越来越多，为了方便管理这些方，记录你的安装信息，我们一般在安装第三方模块的时候，都要添加一个 `--save` 的参数，如下所示

```
$ npm install jquery --save
```

在上面的命令里面，我们可以看到在安装的命令后面，我们添加了一个参数 `--save`，这代表从服务器安装 `jquery` 第三方模块，并且把这次的安装信息记录在 `package.json` 里面，这样其它的同事看到这个项目以后它就知道这个项目依赖于哪些第三方的包了

npm的版本如果大于6及以上，则默认可以不添加这个 `--save` 了，大于6的npm默认会自动携带这个 `--save`

--save-dev记录安装信息

这一种情况和上一种很像，只是它的安装信息会记录在一个 `devDependencies` 里面，这个叫开发依赖（在后面的项目当中讲解开发依赖）

```
$ npm install bootstrap --save-dev
```

这个时候也会记录安装信息，结果如下

```
{
  "name": "100706",
  "version": "1.0.0",
  "description": "标哥讲课的项目",
  "main": "index.js",
  "scripts": {
    "test": "echo \\"$Error: no test specified\\" && exit 1"
  },
  "keywords": [
    "标哥",
    "H2204"
  ],
  "author": "杨标",
  "license": "ISC",
  "dependencies": {
    "jquery": "^3.6.1"
  },
  "devDependencies": {
    "bootstrap": "^5.2.2"
  }
}
```

```
}
```

- `dependencies` 代表生产依赖
- `devDependencies` 代表开发依赖

开发依赖与依赖的区别？

举例：标哥去饭店吃饭，点了一个蛋炒饭

老板为了完成这个蛋炒饭，它需要什么东西？

1. 米饭【生产依赖】
2. 蛋，盐，酱油，油等配料【生产依赖】
3. 锅，锅铲，灶等一系列的工具【开发依赖】

为什么要记录安装包的信息

1. 后期的项目越来越大，第三方包也会越来越多，为了更方便管理自己的包，我们就需要记录这些包的信息
2. 每个项目都会下载第三方的依赖，但是npm会根据当前的系统再自动选择安装包的版本，如苹果的系统就下载苹果的依赖，linux的系统就下载linux的依赖，window10的系统就下载windows10的依赖，还会根据不同的node版本来下载依赖

这个时候就会出现一个问题，我的项目如果发给别人以后，别人要启动，但是它的系统跟不致，这个时候的`node_modules`里面所存放的第三方依赖包就不通用，会报错，

所以我们在传递项目给别人的时候，都会把`node_modules`的文件删除，别人打开项目就没有这个依赖，跑不起来

它就需要查看 `package.json` 的文件，看一下里面是否有记录安装包的信息，如果有，则只需要输入下面的命令就可以了

```
$ npm install
```

这个命令会自动读取当前文件夹下面的 `package.json`，再去读取里面记录的依赖包的信息，然后批量下载

npm uninstall 卸载包

安装完成一个包以后如果想再把这个包把它卸载掉，那么，我们就可以使用这个命令

```
$ npm uninstall 包名称
```

当卸载了某一个包以后 `node_modules` 里面就不会再有这个包了，同时 `package.json` 里面也不会有这个记录信息了

npm install 指定包的版本

当我们使用 `npm install` 去安装一个包的时候，它默认情况下是安装的最新的版本，如果我们想使用某一个特殊的版本，我们可以使用下面的命名

```
$ npm install 包名称@版本号
```

如

```
$ npm install bootstrap@3.3.7
```

批量装包

如果假设有多个包需要我们去安装，可以直接批量进行

```
$ npm install 包名称1 包名称2 ...
```

如

```
$ npm install jquery bootstrap vue
```

npm install的简化命令

npm install 是安装一个包，这也有简化的命令

```
$ npm install 包名
```

简写成

```
$ npm i
```

npm uninstall简化命令

npm uninstall 是卸载一个包，这也有简化的命令

```
$ npm uninstall vue
```

简写

```
$ npm un vue
```

npm命令总结

命令	说明
npm init	初始化
npm init --yes	以默认方式初始化
npm info 包名	查看某个包的信息
npm install 包名	安装一个指定的包
npm install 包名 --save	记录到生产依赖
npm install 包名 --save-dev	记录到开发依赖
npm install 包名@版本号	安装指定的版本号
npm uninstall 包名	卸载指定的包
npm install	根据package.json来自动装包
npm i 包名	简化版的安装包的命令
npm un 包名	简化版的卸载包的命令

npm国内镜像设置

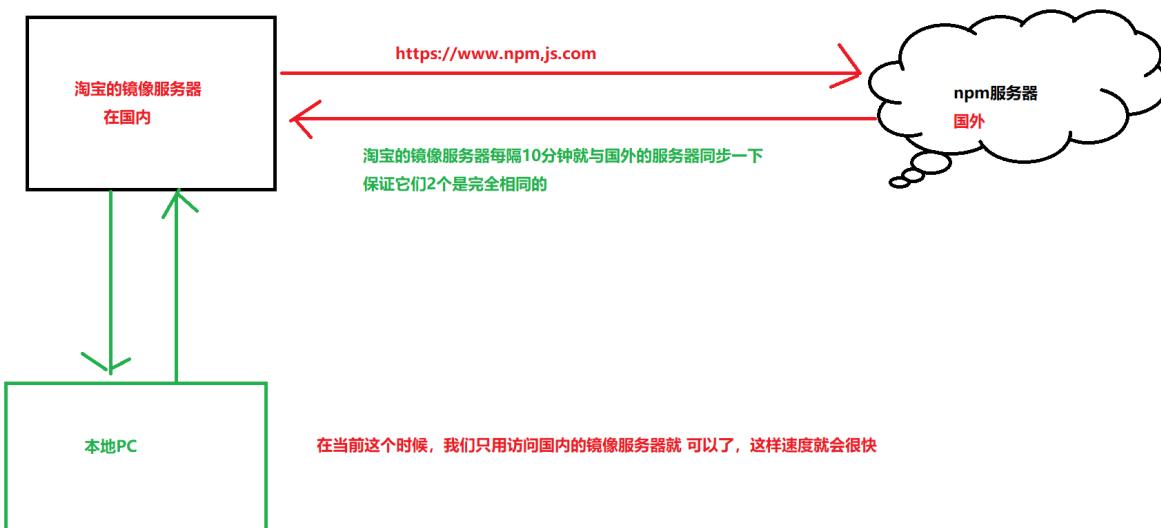
npm的服务器是在国外的，所以我们每次去下载的时候都会访问国外的服务器，这样非常慢



每次下载一个包我们都要从国外的服务器去下载，但是因为一些不可描述的原因，我们访问国外的网站会很慢，特别是后期有些包在 [github](#) 上面的时候更慢。针对这个问题，目前的解决方案有很多，我们大家推荐几种

1. 更改npm的镜像地址
2. 使用国内的包管理工具cnpm
3. 使用yarn包管理工具

目前最简单单的就是使用 [npm](#) 的镜像地址来完成



```
$ npm config set registry https://registry.npm.taobao.org
$ npm config set disturl https://npm.taobao.org/dist
$ npm config set electron_mirror https://npm.taobao.org/mirrors/electron/
$ npm config set sass_binary_site https://npm.taobao.org/mirrors/node-sass/
$ npm config set phantomjs_cdnurl https://npm.taobao.org/mirrors/phantomjs/
```

在自己的电脑上面，以管理员的身份打开控制台

使用axios+cheerio来完成抓包

如果想完成抓包，我们主要使用的就是下面2个包

1. [axios](#) 它是一个跨平台的用于发起http请求的一个包，可以在node.js的平台使用，也可以在浏览器下面使用
2. [cheerio](#) 这个包用于分析HTML的网页，提取网页当相关的信息

模块网络请求

想要更好的模块网络请求，我们需要使用一个包，这个包就是 `axios`

```
const axios = require('axios');

const catchData = async () => {
    console.log("开始请求.....");
    let str = "https://v.qq.com/channel/tv?listpage=1&channel=tv&feature=2";
    // 发起请求，等结果
    try {
        let resp = await axios.get(str, {
            headers: {
                accept: "text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9",
                cookie: "RK=0K+NQJHxW0;
ptcz=fb997511491c9ca0a5c7237f76ccc260449f28c806461ac45266c44285bf212b;
fqm_pvqid=671ce23e-3378-4520-918e-539cdde307bd; pac_uid=1_365055754; iip=0;
o_cookie=365055754; tvfe_boss_uuid=f6743d841fc5508a; ts_uid=2356158233;
Qs_lvt_323937=1648900734%2C1648900905%2C1654258098;
Qs_pv_323937=3912163211342407700%2C4490624458211209700%2C3177819044968152000;
video_guid=2ffe4b234d1926ad; compared_guid=582794d913223941;
ts_refer=www.baidu.com/s; tab_experiment_str=8752037#8826908;
ptui_loginuin=365055754@qq.com; pgv_pvid=6857396830; video_platform=2;
pgv_info:ssid=s6013868365; ts_last=v.qq.com/channel/tv; bucket_id=9231001;
ptag=channel; qv_als=00oH7obq+/JkA7y9A11665141916yoSxPw==; ad_play_index=15"
            }
        });
        // 响应的结果在data里面
        console.log(resp.data)
    } catch (error) {
        console.log("报错了");
        console.log(error);
    }
}

catchData();
```

上面的代码就是通过 `axios` 请求以后得到的结果，`resp.data` 就是结果，我们把结果打印了一下，发现这个结果就是通过上面的网址返回的网页内容

得到的网页的内容以后，我们就要知道我们要得到哪些数据

分析得到的结果

如果我想得到所有的电影的名称，我应该找到一个 `a` 标签，然后去它下面找名称

为了更好的解析网页上面的内容，我们需要借用一个第三方的包 `cheerio`

安装

```
npm install cheerio
```

特征

熟悉的语法：Cheerio 实现了核心 jQuery 的一个子集。Cheerio从jQuery库中删除了所有 DOM不一致和浏览器问题，揭示了其真正华丽的API。

极快：Cheerio 使用非常简单、一致的 DOM 模型。因此，解析、操作和渲染非常高效。

令人难以置信的灵活性：Cheerio环绕解析器解析器，并且可以选择使用@FB55宽容的 `html解析器2`。Cheerio 几乎可以解析任何 HTML 或 XML 文档。

```
/**
 * 抓包
 */

const axios = require('axios');
// cheerio可以把我们抓取的网页像jQuery一样去操作
const cheerio = require('cheerio');
const path = require("path");
const fs = require("fs");

const catchData = async () => {
  console.log("开始请求.....");
  let str = "https://v.qq.com/channel/tv?listpage=1&channel=tv&feature=2";
  // 发起请求，等结果
  try {
    const res = await axios.get(str);
    const html = res.data;
    const $ = cheerio.load(html);
    const list = $('.mod_figure .bold');
    const titles = list.map(item => $(item).text());
    console.log(titles);
  } catch (err) {
    console.error(err);
  }
}

catchData();
```

```

        let resp = await axios.get(str, {
            headers: {
                accept:
                    "text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0
                    .8,application/signed-exchange;v=b3;q=0.9",
                cookie: "RK=0K+NQJHxW0;
ptcz=fb997511491c9ca0a5c7237f76ccc260449f28c806461ac45266c44285bf212b;
fqm_pvqid=671ce23e-3378-4520-918e-539cdde307bd; pac_uid=1_365055754; iip=0;
o_cookie=365055754; tvfe_boss_uuid=f6743d841fc5508a; ts_uid=2356158233;
Qs_lvt_323937=1648900734%2C1648900905%2C1654258098;
Qs_pv_323937=3912163211342407700%2C4490624458211209700%2C3177819044968152000;
video_guid=2ffe4b234d1926ad; compared_guid=582794d913223941;
ts_refer=www.baidu.com/s; tab_experiment_str=8752037#8826908;
ptui_loginuin=365055754@qq.com; pgv_pvid=6857396830; video_platform=2;
pgv_info:ssid=s6013868365; ts_last=v.qq.com/channel/tv; bucket_id=9231001;
ptag=channel; qv_als=00oH7obq+/JkA7y9A11665141916yoSxPw==; ad_play_index=15"
            }
        });
        // 响应的结果在data里面
        // console.log(resp.data);
        //开始使用cheerio来分析网页，并加载成$对象
        let $ = cheerio.load(resp.data);
        let result = [];
        $(".list_item").each((index, ele) => {
            let movieName = $(ele).find(".figure_title_two_row").text();
            let movieDesc = $(ele).find(".figure_desc").text();
            let movieCaption = $(ele).find(".figure_caption").text();
            let moviePic = $(ele).find(".figure_pic").attr("src");
            let obj = {
                movieName,
                movieDesc,
                movieCaption,
                moviePic
            }
            result.push(obj);
        });

        let resultJsonStr = JSON.stringify(result);
        fs.writeFileSync(path.join(__dirname, "./data.txt"), resultJsonStr);
        console.log("数据保存成功");
    } catch (error) {
        console.log("报错了");
        console.log(error);
    }
}

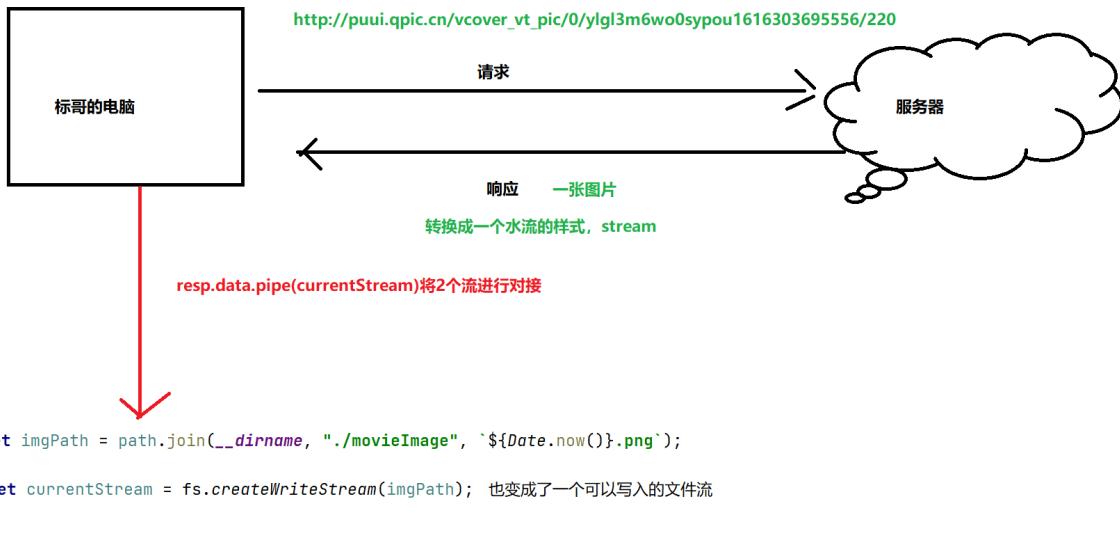
catchData();

```

抓取图片

在上面的代码里面，我们可以看到，我们已经把所有的数据组合成了一个对象，现在我们就要把这个图片下载下来，怎么办呢？

当我们得到了上面的信息以后，我们其实就已经完成了50%的功能了，现在我们在文件里面可以看到有一个属性叫 `moviePic`，这个属性用于保存了图片地址，所以我们可以再次根据这个地址去抓取图片，把它下载到本地。



上图就是从服务器接收数据以后再转换成我们所需要的流就可以了

```

/**
 * 2022-10-08
 * YangBiao
 **/


const axios = require("axios"); //专门用于模拟请求
const cheerio = require("cheerio");
const path = require("path");
const fs = require("fs");

const catchData = async () => {
  console.log("正在抓数据 ... ");
  //第一步: 准备请求的url
  let url = `https://v.qq.com/channel/tv?listpage=1&channel=tv&feature=2`;
  let config = {
    headers: {
      accept:
        "text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0
        .8,application/signed-exchange;v=b3;q=0.9",
      cookie: "RK=0K+NQJHxW0;
ptcz=fb997511491c9ca0a5c7237f76ccc260449f28c806461ac45266c44285bf212b;
fqm_pvqid=671ce23e-3378-4520-918e-539cdde307bd; pac_uid=1_365055754; iip=0;
o_cookie=365055754; tvfe_boss_uuid=f6743d841fc5508a; ts_uid=2356158233;
Qs_lvt_323937=1648900734%2C1648900905%2C1654258098;
Qs_pv_323937=3912163211342407700%2C4490624458211209700%2C3177819044968152000;
video_guid=2ffe4b234d1926ad; compared_guid=582794d913223941;
ts_refer=www.baidu.com/s; tab_experiment_str=8752037#8826908;
ptui_loginuin=365055754@qq.com; pgv_pvid=6857396830; video_platform=2;
bucket_id=9231001; ad_play_index=18; pgv_info=ssid=s5465147712;
ts_last=v.qq.com/channel/tv"
    }
  }
  //第二步: 开始请求
  let resp = await axios.get(url, config);
  //第三步: 加载网页为cheerio, 空上就可以像jquery一样去操作选取了
  let $ = cheerio.load(resp.data);
}

```

```

let result = [];
$(".list_item").each((index, ele) => {
    //第四步：遍历选中的元素，组合成对象
    let movieName = $(ele).find(".figure_title_two_row").text();
    let movieDesc = $(ele).find(".figure_desc").text();
    let movieCaption = $(ele).find(".figure_caption").text();
    let moviePic = "http:" + $(ele).find(".figure_pic").attr("src");
    let obj = {
        movieName,
        movieDesc,
        movieCaption,
        moviePic
    }
    result.push(obj)
});
//我可以把这个数组result遍历一下，拿到里面的moviePic，去模拟请求这个地址，这样这个地址就会返回图片
for (let {moviePic,movieName} of result) {
    //我们假设要把这个图片放在movieImage的目录下面去
    let movieImagePath = path.join(__dirname, "./movieImage");
    //如果没有这个目录，我们就创建这个目录
    if (!fs.existsSync(movieImagePath)) {
        fs.mkdirSync(movieImagePath);
    }
    //第五步：再次模拟请求，请求图片的地址
    let resp = await axios.get(moviePic, {
        ...config,
        //这个时候，请求的结果就会像水流一样
        responseType: "stream"
    });
    //第六步：接收到这个文件流以后，怎么办呢
    let imgPath = path.join(__dirname, "./movieImage", `${movieName}.png`);
    //第七步：将上面的图片地址变成一个可以写入的文件流
    let currentStream = fs.createWriteStream(imgPath);
    //第八步：将两个流对接
    resp.data.pipe(currentStream);
    console.log("图片保存成功");
}
catchData();

```

在上面的代码里面，有两个技术主体

1. 在 `axios` 的请求当中，我们把响应的数据类型改为了 `stream`
2. 我们使用文件流 `fs.createStream()` 来写入文件，这样会非常方便

使用nodemailer发送邮件

在我们以后开发的项目当中我们经常需要使用到邮件验证码，同时也会有些功能需要向用户发送邮件，批量发送等功能，这个时候我们就需要通过 `node.js` 来编程程序，让程序自动的发送邮件

如果要完成这个功能我们可以使用第三方的模块叫 `nodemailer`

安装包

```
$ npm install nodemailer --save
```

准备发送邮箱的账号与密码

目前我准备一个163的邮箱，其它的邮箱的设置也是一样的

The screenshot shows the 163 email account settings interface. At the top, there is a navigation bar with tabs: '设置' (Settings) (highlighted with a red box), '官方App' (Official App), '反馈' (Feedback), '自助查询' (Self-service Inquiry), and '开通邮箱会员' (Activate Email Member). Below the navigation bar, there are several sections: '常规设置' (General Settings), '邮箱密码修改' (Email Password Change), '帐号与邮箱中心' (Account and Email Center) (with a '去体验' (Experience) button), '邮箱安全设置' (Email Security Settings), and 'POP3/SMTP/IMAP' (highlighted with a red box). Further down, there is a '更换皮肤' (Change Skin) section with a '升级邮箱会员, 尊享专属皮肤' (Upgrade Email Member, Enjoy Exclusive Skin) button. At the bottom, there is a section for '开启服务' (Enable Services) with two options: 'IMAP/SMTP服务' (IMAP/SMTP Service) set to '已关闭 | 开启' (Disabled | Enable) and 'POP3/SMTP服务' (POP3/SMTP Service) set to '已开启 | 关闭' (Enabled | Disable) (highlighted with a red box). A note below states: 'POP3/SMTP/IMAP服务能让你在本地客户端上收发邮件, 了解更多 >' (POP3/SMTP/IMAP services allow you to receive and send emails locally, learn more >). A yellow warning box at the bottom right says: '温馨提示: 在第三方登录网易邮箱, 可能存在邮件泄露风险, 甚至危害Apple或其他平台账户安全' (Tip: When logging in to NetEase Email from a third party, there may be a risk of email leakage, even threatening Apple or other platform account security).

如果要使用第三方客户端发送邮件，则要开启上面的服务

上面的 `pop3, smtp, imap` 就是用于发送邮件和接收邮件，同时还要注意服务器的地址

服务器地址:	POP3服务器: <code>pop.163.com</code>
	SMTP服务器: <code>smtp.163.com</code>
	IMAP服务器: <code>imap.163.com</code>
安全支持:	POP3/SMTP/IMAP服务全部支持SSL连接

- pop3是用于接收邮件的服务器
- smtp是用于发送邮件的服务器

协议类型	协议功能	服务器地址	非SSL端口号	SSL端口号
SMTP	发送邮件	smtp.163.com	25	465
POP	接收邮件	pop.163.com	110	995
IMAP	接收邮件	imap.163.com	143	993

```
/**
 * 2022-10-08
 * YangBiao
 **/


const nodemailer = require("nodemailer");
const path = require("path");


//编写一个函数，用于发送邮件
const sendMail = async () => {
    //第一步：创建一个邮件传输对象
    let passport = nodemailer.createTransport({
        host: "smtp.163.com",
        port: 465,
        auth: {
            user: "mh475201314@163.com",
            //这里填自己的
            pass: "BZVQLUJXHPPQJZQF"
        }
    });
    try {
        //第二步：发送邮件
        let result = await passport.sendMail({
            subject: "高清大图等你来看",
            from: "mh475201314@163.com",
            to: [
                "mh475201314@163.com",
                "173731044@qq.com",
                "2240933562@qq.com",
                "1803334091@qq.com",
                "l635972779@163.com",
                "3256406860@qq.com",
                "1803334091@qq.com",
                "3160656756@qq.com",
                "2434893662@qq.com",
                "xxxiaohanxxx@163.com",
                "1871543651@qq.com",
                "121610681@qq.com",
                "2361229445@qq.com",
                "990715003@qq.com",
                "3089406860@qq.com",
                "1139746253@qq.com",
                "568545793@qq.com",
                "2677573347@qq.com"
            ],
            cc: "lovesnsfi@163.com",
            text: `
```

亲爱的各们小伙伴：
恭喜你！
你已被本公司抽取为幸运观众，请凭验证码\${~(Math.random() * 10000)}到本公司领
导假值19999的大礼一份，过时不候！
同时为了更好的保证您能够顺利的领导本奖品，请您在领奖之前交纳1000元保证金，恭候您的
光临。
如有疑问，请电联
18712345678

XXX骗子公司

```
`${new Date().toLocaleString()}`  
,
```

//附件

```
attachments: [  
    {  
        filename: "标哥哥的帅气照片.jpg",  
        path: path.join(__dirname, "./img/2018上.jpg")  
    }, {  
        filename: "美女的照片.jpg",  
        path: path.join(__dirname, "./img/w08.jpg")  
    }  
]  
});  
console.log("邮件发送成功");  
console.log(result);  
} catch (e) {  
    console.log("邮件发送失败了");  
    console.log(e);  
}  
}  
  
//调用发送邮件的方法  
sendMail();
```

node-xlsx的使用

在我们平常的开发与使用当中，我们经常看到下面的场景



实训班级考勤		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
编号	学生姓名	三	四	五	六	日	一	二	三	四	五	六	日	一	二	三	四	五	六	日	一	二	三	四
1	张三	90	85	92	88	95	80	93	87	91	89	94	86	92	84	96	90	88	93	86	91	89	95	92

如果想实现这个场景，我们就要知道怎么样去实现通过 node.js 去操作excel

如果要使用nodejs来操作Excel我们就要使用 node-xlsx ,这个包可以帮我们读取excel的文件，也可以把一些数据生成excel

安装

```
$ npm install node-xlsx --save
```

读取excel文件

```
const xlsx = require("node-xlsx").default;
const path = require("path");
const fs = require("fs");

//读取 excel文件
const readExcel = () => {
    //第一步：先构建excel文件的路径
    let excelPath = path.join(__dirname, "./H2204信息.xls");
    let result = xlsx.parse(excelPath);
    console.log(result);
}

readExcel();
```

结果如下

```
[  
  {  
    name: 'Sheet1',  
    data: [  
      [Array], [Array], [Array], [Array], [Array],  
      [Array], [Array], [Array], [Array], [Array]  
    ]  
  },  
  { name: 'Sheet2', data: [ [Array], [Array], [] ] }  
]
```

A	B	C	D	E	F	G	H	I	J	K	L
1	姓名	性别	年龄	毕业院校	专业						
2	何祥宇	男	25	武汉华夏理工学院	软件1196班						
3	肖中	男	20	文华学院	机械设计制造及自动化						
4	吴燃腾	女	23	湖北工程学院新技术学院	计算机科学与技术						
5	贺锐	男	22	武昌理工学院	计科						
6	方鹏	男	25	武汉东湖学院	软件工程						
7	徐吉成	男	18	湖北工程学院新技术学院	计算机科学与技术						
8	冉良超	男	18	武昌工学院	软件工程						
9	陈韩家	男	22	武汉华夏理工学院	软件工程						
10	刘诗霞	女	23	汉口学院	物联网工程						
11	陈祈成	男	19	武昌理工学院	软件工程						
12	程志超	男	22	武昌工学院	软件工程						
13	徐书畅	男	23	武汉工程科技学院	计算机科学与技术						
14	陈怡静	女	18	武汉华夏理工学院	软件工程						
15	陈文号	男	20	湖北工程学院新技术学院	计算机科学与技术						
16	栗秀峰	男	21	湖北工程学院新技术学院	软件工程						
17	王宁娟	女	24	武汉华夏理工学院	软件工程						
18	曹方	女	24	湖北工程学院新技术学院	计算机科学与技术						
19	赵腾	男	23	武汉工程科技学院	计算机科学与技术						
20	万金保	男	20	武汉华夏理工学院	软件工程						
21	郝柏林	男	25	湖北工程学院新技术学院	计算机科学与技术						
22	马淑圆	女	19	汉江师范学院	旅游英语						
23	李康	男	22	武昌理工学院	软件工程						
24	洪延军	男	18	武汉华夏理工学院	计算机科学与技术						
25	王杜婧	女	20	湖北工程学院新技术学院	计算机科学与技术						
26	王泽冰	男	22	武汉东湖学院	软件工程						
27	叶俊豪	男	18	湖北师范大学	体育运营与管理						
28	韩宏扬	男	24	武汉东湖学院	软件工程						
29	吕振阳	男	21	武汉华夏理工学院	软件工程						
30	陈润福	男	23	武汉工程大学邮电与信息工程学院	软件工程						
31	隗志勇	男	19	武汉工程大学邮电与信息工程学院	软件工程(大数据与云计算方向)						
32	陈文	男	23	湖北轻工职业技术学院	计算机网络基础						

当我们去打印第一个对象的时候，可以看到下面结果

```
{
  name: 'Sheet1',
  data: [
    [ '姓名', '性别', '年龄', '毕业院校', '专业' ],
    [ '何祥宇', '男', 25, '武汉华夏理工学院', '软件1196班' ],
    [ '肖中', '男', 20, '文华学院', '机械设计制造及自动化' ],
    [ '吴燃腾', '女', 23, '湖北工程学院新技术学院', '计算机科学与技术' ],
    [ '贺锐', '男', 22, '武昌理工学院', '计科' ],
    [ '方鹏', '男', 25, '武汉东湖学院', '软件工程' ],
    [ '徐吉成', '男', 18, '湖北工程学院新技术学院', '计算机科学与技术' ],
    [ '冉良超', '男', 18, '武昌工学院', '软件工程' ],
    [ '陈韩家', '男', 22, '武汉华夏理工学院', '软件工程' ],
    [ '刘诗霞', '女', 23, '汉口学院', '物联网工程' ],
    [ '陈祈成', '男', 19, '武昌理工学院', '软件工程' ],
    [ '程志超', '男', 22, '武昌工学院', '软件工程' ],
    [ '徐书畅', '男', 23, '武汉工程科技学院', '计算机科学与技术' ],
    [ '陈怡静', '女', 18, '武汉华夏理工学院', '软件工程' ],
    [ '陈立早', '男', 20, '湖北工程学院新技术学院', '计算机科学与技术' ]
  ]
}
```

这个结果就是第一个工作表的结果

现在我们就可以读取里面数据，转换成JS所需要的对象或保存为一个JSON文件

```
/**  
 * 2022-10-08  
 * YangBiao  
 */  
const xlsx = require("node-xlsx").default;  
const path = require("path");  
const fs = require("fs");  
  
//读取 excel文件  
const readExcel = () => {  
    //第一步：先构建excel文件的路径  
    let excelPath = path.join(__dirname, "./aaa.xlsx");  
    //第二步：读取excel工作簿，一个工作簿里面有多个工作表，我们现在只要第1个工作表  
    let [result] = xlsx.parse(excelPath);  
    //第一行是属性名  
    let keys = result.data.shift();  
    //现在开始遍历，组合成对象  
    let arr = [];  
    for(let item of result.data){  
        let obj = {};  
        for(let i in item){  
            obj[keys[i]] = item[i];  
        }  
        arr.push(obj);  
    }  
    //写成JSON文件  
    let jsonStr = JSON.stringify(arr);  
    //定义一个保存的位置  
    let savePath = path.join(__dirname, "./aaa.txt");  
    fs.writeFileSync(savePath, jsonStr);  
    console.log("保存成功");  
}  
  
readExcel();
```

生成excel文件

在上面的操作里面，我们是将一个excel文件读取出来，得到了我们想要的数据，那么我们如果将一些数据生成excel呢

```
const xlsx = require("node-xlsx").default;  
const path = require("path");  
const fs = require("fs");  
  
//要求，现在有一个data.txt的文件，将这个文件 生成一个excel的文件  
const writeExcel = () => {  
    //第一步：先构建需要读取的文件的路径  
    let p1 = path.join(__dirname, "./data.txt");  
    //第二步：读这个文件，buff【buffer就是内存当中的一些数据】  
    let buff = fs.readFileSync(p1);  
    let jsonStr = buff.toString();  
    //第三步：将json字符串转换成对象  
    /* @type {Array} */
```

```

let arr = JSON.parse(jsonStr);
//第四步：构建生成excel所需要的数据
let excelObj = {
    name: "Sheet1",
    data: []
}
//第四步：先构建excel的第一行
let firstRow = Object.keys(arr[0]);
excelObj.data.push(firstRow);
//第五步：开始构建第二行以后的东西
for (let item of arr) {
    excelObj.data.push(Object.values(item));
}
//第六步：准备生成的excel的路径
let excelSavePath = path.join(__dirname, "./data.xlsx");
//第七步：开始生成，它会生成一个buff【buffer就是内存当中的一些数据】
let excelBuff = xlsx.build([excelObj]);
//第八步：将刚刚生成的buff写入到文件里面
fs.writeFileSync(excelSavePath, excelBuff);
console.log("生成成功");
}

writeExcel();

```

在上面的代码当中，最重要的一点就是构建要生成excel的数据，然后再将这个数据通过 `fs.writeFileSync` 写入到文件就可以了

总结

1. `parse()` 用于读取 `excel` 的内容，它会返回一个数组，这个数组就是excel里面的数据
2. `build()` 用于将一些数据生成excel所需要的 `buffer` 【buffer就是内存当中的一些数据】，然后我们再通过 `node.js` 里面的 `fs.writeFileSync()` 写到文件里面
3. 一个excel的工作簿(workbook)里面应该是有多个工作表(sheet)的

综合练习

1. 请抓取华夏学院新闻数据，网址如下

```
http://www.hxut.edu.cn/plus/list.php?tid=69&TotalResult=3682&PageNo=1
```

- 我校2个基层党组织获批全省高校党建工作“标杆院系”“样板支部”创建培育单位 2022-10-03
- “才聚荆楚，勇往职前”——2022年湖北省大学生求职大赛武汉华夏理工学院决赛开赛 2022-10-01
- 校党委书记丁德智深入辅导员社区倾听学生诉求 2022-10-01
- 马边彝族自治县与我校校地合作项目启动会举行 2022-09-30
- 我校举办2022年“湖北百校联动”秋季专场招聘会 2022-09-29
- [我校艺术学子在第十届未来设计师·全国高校数字艺术设计大赛中斩获佳绩](#) 2022-09-29
- 学校举办“最美基层就业人物”主题宣传展览活动 2022-09-28
- 校党委书记丁德智和新进教师面对面谈职业发展 2022-09-27
- “我与祖国同奋进”——学校开展升国旗主题教育活动 2022-09-27
- 武汉理工大学马克思主义学院院长朱洁教授受聘为我校马克思主义学院名誉院长 2022-09-24
- 我校在湖北省第八届高校青年教师教学竞赛中获佳绩 2022-09-24
- 华夏建规工作室举办2022年暑期集训营成果展 2022-09-23
- 学校召开2022年高等教育质量监测国家数据平台数据填报工作布置会 2022-09-22
- 学校召开党委理论中心组学习会暨党委（扩大）会议 2022-09-21
- 我校教师在湖北省首届素质教育研究优秀成果评选活动中斩获多项荣誉 2022-09-21
- 我校制药学子在湖北省大学生生物实验技能竞赛中获佳绩 2022-09-21
- 我校教育基金会获评4A级社会组织 2022-09-20
- 开学第一课，新进老师激励新生将“我的梦”融入“中国梦” 2022-09-20
- 湖北省教育考试院一行莅临学校督导检查计算机等级考试准备工作 2022-09-19
- 学校举行2022级新生军训阅兵暨总结表彰大会 2022-09-19
- 土木建筑工程学院携手马克思主义学院举行主题党课 2022-09-18
- 我校外国语学院赴文华学院外语学部走访调研 2022-09-17
- 东湖新技术开发区教育局副局长沈爱珍来校检查指导疫情防控工作 2022-09-16
- 关南派出所走进校园开展2022级新生防电诈宣讲 2022-09-16
- 湖北省国家保密局检查组来校保密室开展专项检查 2022-09-16

7页的数据全部抓取出来



首页 1 2 3 4 5 6 7 下一页 末页

生成一个excel文件，里面有新闻的标题 `newsTitle`，要有时间 `newsTime`，新闻的链接 `newsLink`，新闻的作者 `newsAuthor`，新闻的内容 `newsContent`

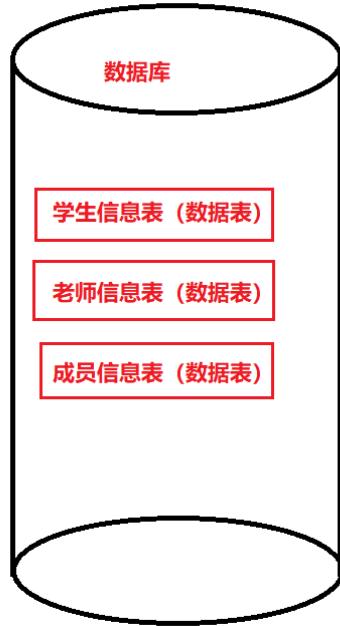
当excel文件生成了以后，请将它发送到 `mh475201314@163.com` 的邮箱，以附件的形式发送，注明姓名是谁

MySql基础

什么是数据库？它就是一个存放数据的仓库，官方的说话叫数据持久化，数据库与我们之前所学习的 `excel` 非常相似，一个Excel工作薄里面有很多个工作表



一个Excel就可以看成是一个数据表，里面可以存放我们的数据，一个数据库里面也会有很多的数据表



现在的数据库一般都分为2大类型

1. 关系型数据库

- mysql
- sql server
- oracle
- db2
- access

2. 非关系型数据型(NoSQL)

- mongodb
- redis

MySql是一种**关系型的服务器**数据库，它可以使用面向对象的思维去理解它，运用它

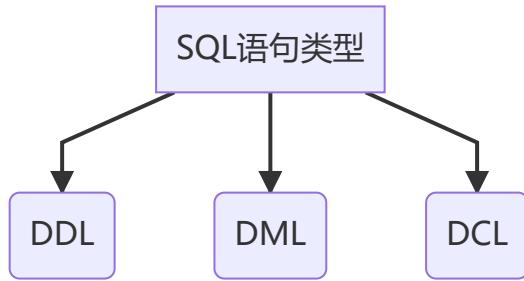
一个数据库里面有很多个表，一个表里面应该会有很多个数据，如下所示

学生信息表	老师信息表

在上面的图片里面我们可以看到，**数据库**就是由很多个**数据表**来构成的，而在一个表里面，它应该有**行**和**列**的概念

在数据库的学习里面，如果要操作数据库的结构与数据，需要使用一种专门的技术，这种技术叫**结构化查询语句 SQL (Structured Query Language)**

SQL功能强大，简单易学，使用方便，已经成为了数据库操作的基础。在学习SQL语句（命名）的时候，其实SQL也会为三在类



1. DDL 语句

全称叫数据库定义语句 (Database Define Language) , 它用创建库, 创建表, 定义表的数据结构以及数据类型, 不会操作数据库具体存放的数据。它主要的关键字有
`create,drop,alter,modify,change,show,desc` 等

DDL语句只有结构有关, 不与数据相关

2. DML 语句

全称叫数据库操作语句 (Database Manipulation Language) , 它不与数据库的结构有关, 只负责操作数据库当中存放的数据, 它主要的关键字有
`insert,delete,update,select,where,and,or` 等

3. DCL 语句

全称叫数据库控制语句 (Database control language) , 它主要负责控制数据库的权限, 一般情况下是DBA使用这种语句

数据库连接

在使用数据库之前, 我们要先连接数据库, 但是在初学阶段我们不建议同学们直接使用数据库管理工具来管理自己的数据库, 而是要通过命令行来操作自己的数据库 (也就是DDL语句)

在每个电脑上面我们都可以使用DOS命令来操作连接自己的数据库, 所以我们只需有DOS (linux系统叫终端工具) 就可以了

1. 按 `win+r` 键
2. 输入 `cmd` 打开DOS窗口, 如下所示

C:\WINDOWS\system32\cmd. + <

Microsoft Windows [版本 10.0.22623.730]
(c) Microsoft Corporation。保留所有权利。

C:\Users\YangBiao>

上在的窗口就是DOS界面的窗口，我们可以在里面执行一些常用的DOS命令

1. `dir` 命令，列出当前目录下现的文件夹与文件
2. `cd` 命令，切换文件夹，`cd .` 切换当前文件夹，`cd ..` 切换到上级目录，`cd yangbiao` 切换到 `yangbiao`这个文件夹，`cd \`直接切换到当前这个盘符的根目录
`cd` 命令的全称叫 `change directory` 切换文件夹
3. `ipconfig` 查看当前网络配置信息，如果要拿所有的网卡人话上，我们可以使用 `ipconfig /all`
4. `netstat -ano` 查看当前正在使用的端口号
5. `ping` 向某一个服务器发起请求，看是否可以得到回应

有了这个DOS命令以后，我们就可以开始使用DOS命令来连接我们的数据库

```
$ mysql -h 127.0.0.1 -uroot -p
```

-h代表的是host主机地址，也代表你要连接的mysql服务器地址，后面的127.0.0.1是本机IP,如果要连接其它的服务器，把这个IP换成其它服务器的IP，如果连接是自己的服务器，则可以把 `-h 127.0.0.1` 省略

-u代表user用户名，后面的root代表我们使用这个用户名 `root` 是mysql数据库当中的超级管理员

-p代表password，有这个东西代表我们要使用密码连接

```
YangBiao@YB-Huawei ~ >>> mysql -h 127.0.0.1 -uroot -p
Enter password: *****
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 6
Server version: 5.5.40 MySQL Community Server (GPL)

Copyright (c) 2000, 2014, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

当进入到这个界面的时候，我们就已经连接到数据库了，上面有个命令 `Commands end with ; or \g.` 代表 `mysql` 的命令以 ; 结束，所以我们输入每一个sql句以后都要添加分号； `Your MySQL connection id is 6` 你当前连接的编号是 6

一旦连接到数据库以后，我们接下来所输入应该是 SQL 语句了

基础DDL语句

1. 退出mysql

```
exit;
-- 或
quit;
```

2. 显示当前所有数据库

```
show databases;
```

3. 创建数据库

```
-- 创建一个test2的数据库
create database 数据库名;
create database test2;
```

4. 删除数据库

```
-- 删除数据库test2;
drop database 数据库名;
drop database test2;
```

5. 切换数据库

```
use 数据库名;
use h2204;
```

切换数据库就相当于打开了某一个数据库（在电脑上面的理解可以认为是打开了一个某一个Excel工作簿）

6. 显示数据库下面的所有的表

```
show tables;
```

7. 创建数据库

```
create table if not exists 表名(
    列名1 列数据类型,
    列名2 列数据类型
) engine = innodb default charset = UTF8;

create table if not exists teacherInfo(
    stuName varchar(10),
    stuSex varchar(2),
    stuAge int
) engine=innodb default charset = UTF8;
```

engine=innodb 代表当前数据表的存储引擎使用的是 **innodb**，这种存储引擎可以实现表与表之间的主外键关系

8. 重命名数据表

```
alter table 原表名 rename 新表名;
-- 将teacherinfo的表修改为 teacher
alter table teacherinfo rename teacher;
```

9. 显示数据表列的信息

```
show columns from 表名;
-- 显示teacher表里面的列的信息
show column from teacher;

-- 它有一个简洁的命令
desc 表名;
desc teacher;
```

Field	Type	Null	Key	Default	Extra
stuName	varchar(10)	YES		NULL	
stuSex	varchar(2)	YES		NULL	
stuAge	int(11)	YES		NULL	

上图就是显示的结果， **Field** 代表列名， **Type** 代表这列的数据类型， **Null** 代表这列是否可以为空值， **Default** 代表这一个如果在不输入的时候的默认值， **Extra** 代表是否有外键或约束， 索引等情况。 **Key** 代表这一列是否是主键

10. 修改数据表，添加数据列

```
alter table 表名 add column 列名 列的类型及描述信息
-- 如下，添加 stuAddress这一列，字符串长度为varchar(100)
alter table teacher add column stuAddress varchar(100);
```

11. 修改数据表，删除数据列

```
alter table 表名 drop column 列名;
-- 删除stuAddress这一列
alter table teacher drop column stuAddress;
```

12. 修改数据表，修改某一列的名称

```
alter table 表名 change 原列名 新列名 数据类型及描述信息;
-- 将stuAge的名称改为age
alter table teacher change stuAge age varchar(2);
```

这里用到了一个关键字 **change**，这个关键字用于修改名称

13. 修改列的属性信息

```
-- 将 age的属性由varchar(2) 改为 int(11)
alter table 表名 modify 列名 类型及描述信息;
alter table teacher modify sex int(11);
```

修改列的名称我们使用 **change**，修改列的属性我们使用 **modify**

14. 删除数据表

```
drop table 表名;
-- 删除teacher
drop table teacher;
```

15. 创建数据表，并指定数据列的特殊状态

```
create table if not exists stuInfo(
    stuName varchar(10) not null,
    stuSex varchar(2) not null,
    stuAge int not null default 18,
    stuAddress varchar(100)
)engine = innodb default charset = UTF8;
```

not null代表不能为空 default代表默认值

16. 在第15项的基础上面，进行修改，请将stuAddress 修改为不能为空，默认值为china

-- 修改表stuinfo,修改列的属性

数据表的主键

什么是数据表的主键？

主键是数据表当中最重要的一个必不可少的一个关键技术，在数据表当中，如果某一列设置成了主键列，则这一列不能空，并且不能重复。

在日常生活当中，我们有一些信息都是主键信息。如学生的学号**不能重复（唯一）**，**也不能为空**；每个人的身份证号不能重复，也不能为空，这种信息我们都会使用主键去表示

设置主键的方式有很多种

1. 如果一个表已经存在了，现在需要把某一列设置为主键

Field	Type	Null	Key	Default	Extra
stuName	varchar(10)	NO		NULL	
stuSex	varchar(2)	NO		NULL	
stuAge	int(11)	NO		18	
stuAddress	varchar(100)	NO		china	

在上图里面，我们想把 `stuName` 设置为主键，怎么办呢

```
-- 修改表，添加主键  
alter table 表名 add primary key(列名);  
-- 如  
alter table stuinfo add primary key(stuName);
```

2. 怎么样直接在创建表的时候就把某一列设置为主键

```
create table if not exists stuinfo(  
    id int primary key,  
    stuName varchar(20) not null,  
    stuSex varchar(2) not null,  
    stuAge int not null default 18,  
    stuAddress varchar(100)  
)engine = innodb default charset = UTF8;  
  
-- 第二种写法，直接放在最后  
create table if not exists stuinfo(  
    id int,  
    stuName varchar(20) not null,  
    stuSex varchar(2) not null,  
    stuAge int not null default 18,  
    stuAddress varchar(100),  
    primary key(id)  
)engine = innodb default charset = UTF8;
```

删除主键

```
alter table 表名 drop primary key;  
-- 如  
alter table stuinfo drop primary key;
```

数据表的外键

什么是外键？为什么需要外键

当一个表引用了另一个表的主键去存储数据时候，这一列我们就称之为外键列，如下所示

学生信息表			假期轰趴报名表		
姓名 (主键)	性别	年龄	id	报名人姓名 (外键)	
何祥宇	男	-1	1	肖冉	
肖中	男	22	2	贺锐	
吴燃腾	女	20	3	徐吉成	
贺锐	男	21	4	徐吉成	
方鹏	男	22	5	冉良超	
徐吉成	男	21	6	陈韩家	
冉良超	男	-1	7	刘诗霞	
陈韩家	男	20	8	陈祈成	
刘诗霞	女	-1	9	程志超	
陈祈成	男	-1	10	徐书畅	
程志超	男	-1	11	陈怡静	
徐书畅	男	20	12		
陈怡静	女	21	13		
陈文号	男	21	14		
栗秀峰	男	23	15		
王宁娟	女	20			
曹方	女	20			
赵腾	男	21			
万金保	男	-1			
郝柏林	男	20			
马淑圆	女	25			

```
create table if not exists 表名(  
    列名 数据类型....  
    foreign key(当前表某一列) references 其它表(某一列)  
)engine = innodb default charset = utf8;
```

现在我们根据下面的图来创建2个表，构建主外键的约束关系

A	B	C	D	E	F	G	H	I	J
booktype				bookinfo					
id	o type				id	typeid	bookname	price	
	1 小说				1	1	我在人间凑数的日子	99	
	2 漫画				2	1	三国演义	34	
	3 杂志				3	2	名侦探柯南	40	
					4	3	意林	33	

id 主键 int
type varchar(100) 不为空

id 主键 int类型
typeid int类型 外键，引用了booktype里面的id
bookname varchar(100) 不为空
price float 不为空

```
create table if not exists booktype (  
    id int primary key,  
    type varchar(100) not null  
)engine = innodb default charset = utf8;
```

```
create table if not exists bookinfo (  
    id int primary key,  
    typeid int not null,  
    bookname varchar(100) not null,  
    price float not null,
```

```
foreign key(typeid) references booktype(id)
)engine = innodb default charset = utf8;
```

在创建外键的要注意，你所创建的外键的数据类型要与引用的那个表的主键的数据类型保持一致，否则就会失败，同时键一般都是不能为空

MySql 数据类型简述

1. `varchar` 与 `char` 的区别

`char(10)`代表字符串长度为10,它是固定长度

stuName (char(10))	stuSex	stuAge
标哥	男	18
桃子	女	19
岳圣哲	男	20
易洋洋	男	20

标哥	标哥	标哥
桃子	桃子	桃子
岳圣哲	岳圣哲	岳圣哲
易洋洋	易洋洋	易洋洋

`varchar`称之为变长，就是格子的长度不固定

`varchar(10)`最长是10

stuName (varchar(10))	stuSex	stuAge
标哥	男	18
桃子	女	19
岳圣哲	男	20
易洋洋	男	20

标哥	标哥	标哥
桃子	桃子	桃子
岳圣哲	岳圣哲	岳圣哲
易洋洋	易洋洋	易洋洋

`char` 的优点是存储与读取都非常快，它是对齐的存储格式，但是很消耗存储空间

`varchar` 的优点是长度不固定，根据你的内容来决定长度，它不用占用很大的空间，但是因为在存储的时候内存没有对齐，所以查找与存储的时候速度稍微慢一点

这两种数据类型在后期都会用到

练习

1. 根据要求完成如下的表的创建及修改过程

- 新建一个数据库，名为test1010
- 进入到这个数据库test1010
- 创建一个表，信息如下

```
tid int(11),
tname varchar(20),
tsex varchar(2),
t_mail varchar(255)
```

表名为tinfo

- 将表名修改为teacher_info
- 新增一列 t_phone varchar(18)
- 删除一列 t_mail
- 将t_phone的列修改为 t_tel varchar(20)

```

create table if not exists tinfo(
    tid int(11),
    tname varchar(20),
    tsex varchar(2),
    t_mail varchar(255)
)engine = INNODB default charset = UTF8;

alter table tinfo rename teacher_info;

-- 我是注释
-- alter table tinfo rename teacher_info;

-- 新增一列
-- alter table teacher_info add column t_phone varchar(18);

-- 删除一列 t_mail
-- alter table teacher_info drop column t_mail;

-- 修改t_phone 为 t_mail
-- alter table teacher_info change t_phone t_tel varchar(20);

```

2. 根据要求，创建如下的数据表

stuInfo 表				stuScore学生成绩表			
id	stuName	stuSex	stuAge	id	sid	type	score
1	张珊	女	29	1	1	语文	100
2	李四	男	18	2	1	数学	90
3	王五	男	20	3	2	语文	88
				4	2	数学	67
				5	3	语言	96

stuInfo 表
 id为int类型， 主键
 stuName为varcahr(10)类型， 不能为空
 stuSex为varchar(2)类型， 不能为空
 stuAge为int类型,不能为空， 默认为18

stuScore学生成绩表
 id为int类型， 主键
 sid为int类型， 外键， 引用了stuInfo里在的id这一列数据
 type为varchar(20)不为空
 score为float类型， 不为空

MySql的DML语句

之前在进入到MySql的时候，我们就提到过，MySql需要使用专门的语句SQL语句来进行操作， SQL语句的分类又分为三大类

1. DDL语句，主要是用于控制数据库与数据表的结构， 使用 `create,drop,alter,show` 等关键字， 它不与数据库的内容相关
2. DML语句，主要是用于对数据表的内容进行增删改查， 使用 `insert,delete,update,select` 等关键字， 这种类型的SQL语句只操作数据库的内容， 不对数据库及数据表的结构进行操作
3. DCL语句，主要是用数据库的权限进行操作

我们已经了解了数据库表的创建及表的特点，现在我们需要去习怎么向操作表里面的数据，这个时候就需要学习DML这一个类型的语句（程序员的重点）

DML语句的学习主要是围绕“新增”，“删除”，“修改”，“查询”四个操作来完成，它分别会有一些特殊的关键字

操作	关键字
新增	insert
删除	delete
修改	update
查询	select

开发的时候经常所说的CRUD指的就是上面的四个操作

新增SQL语句

```
insert into 表名 (列名1,列名2....) values (值1,值2....);
```

上面的语句就是插入语句的SQL

```
insert into stuinfo (stuName,stuSex,stuAge) values ("标哥","男",18);
```

修改的SQL语句

```
update 表名 set 列名1 = 值1,列名2=值2 where 条件;
```

上面就是修改的SQL语句，在修改的时候，我们如果要修改某些列，只需要单独的列出要修改的列就行了，不要修改的就不要列举，后面的where则代表限定条件

```
update stuinfo set stuAge = 99 WHERE stuName = "标哥";
```

我们的修改的时候还可以同时修改多个列

```
update stuinfo SET stuSex = "保密", stuName = "标哥帅哥" where stuName = '标哥'
```

我们在修改的时候还可以直接在某一个列的基础上面修改，如我们希望在所有的姓名的后面追加一个666

```
update stuinfo set stuName = CONCAT(stuName,"888") WHERE stuName = '标哥帅哥666';
```

因为原来的 `stuName` 是一个字符串，所以这里我们使用了 `concat` 来拼接字符串

删除SQL语句

删除数据库是一个非常危险的操作，我们一定谨慎的处理，所以在执行语句之前一定要检测是否有添加限定条件

```
delete from 表名 where 条件;
```

在执行删除操作的时候，一定要添加限定的where条件，如果不添加，会把整个数据表都删除

```
delete from stuinfo where stuName = '标哥帅哥666888';
```

上面的三种SQL语句属于新增，修改，删除的操作，它是DML语句当中最常见的，也是比较简单的。在DML里面，最复杂的还是查询语句

查询SQL语句

查询是DML语句里面比较复杂的一个SQL语句，但是本身的语法又很简单，只要遵守查询的语法就可以得到结果

```
select 列名1,列名2 from 表名 where 条件 ;
```

这个 `select` 的关键字后面跟的就是列名，如果想把某些列出来就加上某些列的名称

```
select sid,sname from stuinfo;
```

如果要显示所有列的名称，可以直接使用*

```
select * from stuinfo;
```

查询范例

1. 查询性别为“男”的学生信息

```
select * from stuinfo where ssex = '男';
```

2. 查询cid小于4的学生

```
select * from stuinfo where cid < 4;
```

3. 查询cid不等于2的学生

```
select * from stuinfo where cid ≠ 2;  
-- 或  
select * from stuinfo where cid ◁ 2;
```

4. 查询所有 sphoto 为 null 的学生

```
select * from stuinfo where sphoto is null;
```

5. 查询所有 sphoto 不为 null 的学生

```
select * from stuinfo where sphoto is not null;
```

6. 查询 cid 在 1~3 之间的，包含 1 和 3

```
select * from stuinfo where cid ≥ 1 and cid ≤ 3;
```

如果有多个条件限定，我们就要使用 `and` 的关键字来进行拼接

小技巧：如果是数字表示的区间，我们还可以使用一个特殊的关键字 `between ... and`

```
select * from stuinfo where cid between 1 and 3;
```

7. 查询性别为女，且cid为3的学生

```
select * from stuinfo where ssex = '女' and cid = 3;
```

8. 查询cid为1或cid为4的学生

```
select * from stuinfo where cid = 1 or cid = 4;
```

上面的or就是或的条件使用

9. 查询cid为1或cid为4的且是女生

```
select * from stuinfo where (cid = 1 or cid = 4) and ssex = '女';
```

10. 查询stuinfo表里面一共有多少条数据

```
select count(*) from stuinfo;
```

11. **as** 别名转换，可以将一个列或一个表或一个查询结果使用别名来表示

```
select count(*) as 'totalCount' from stuinfo;
```

12. 当 **as** 做别名转换的时候是可以省略掉的，所以上面的SQL语句可以转换为下面的写法

```
select count(*) 'totalCount' from stuinfo;
```

13. 查询stuinfo表里面,统计里面的女生有多少人

```
select count(*) 'girlCount' from stuinfo WHERE ssex = '女';
```

14. 查询stuinfo表里面,统计里面的男生有多少人

```
select count(*) 'boyCount' from stuinfo WHERE ssex = '男';
```

15. 查询stuinfo表里面，男生与女生各有多少人【一条SQL语句完成】

```
select ssex,count(*) 'totalCount' from stuinfo  
group by ssex;
```

16. 查询各民族各有多少人【分组聚合】

```
select snation,count(*) from stuinfo  
group by snation;
```

17. 查询每个班的男生女生各有多？【分组聚合】

```
select cid,ssex,count(*) 'totalCount' from stuinfo  
group by cid,ssex;
```

18. 查询汉族里面男生与女生各有多少？

```
select ssex,count(*) 'totalCount' from stuinfo  
where snation = '汉族'  
group by ssex;
```

分组聚合其实就是group by 的使用

group by语句要放在where语句的后面

19. 查询姓名为余珊珊,王静,李明的信息

```
select * from stuinfo where sname = '余珊珊' or sname = '王静' or sname = '李明';
```

上面的SQL语句虽然说是可以得到的结果的，但是我们发现这么写有一个问题：如果限定的人数越来越多，则后面的or条件也会越来越长。同时我们还发现一个点，它所有的or都只针对 `sname` 这个字段

```
select * from stuinfo where sname in ('余珊珊', '王静', '李明');
```

20. 查询姓名不为余珊珊,且不为王静,且不为李明的信息

```
select * from stuinfo where sname != '余珊珊' and sname != '王静' and sname != '李明';
```

对于上面的情况，也是可以简写的

```
select * from stuinfo where sname not in ('余珊珊', '王静', '李明')
```

上面的2个语句是in与not in的使用结合

21. 查询stuinfo里面所有的信息，结果以cid为升序排列

```
select * from stuinfo  
order by cid asc;
```

order by 是排序的意思，默认情况下是asc升序，如果想使用降序，那么就把asc换成desc

22. 查询成绩表里面最高分是多少

```
select max(score) 'max_score' from gradeinfo;
```

23. 查询成绩表里面最低分是多少

```
select min(score) 'min_score' from gradeinfo ;
```

24. 求成绩表里面的平均分

```
select avg(score) 'avg_score' from gradeinfo;
```

在上面的代码里面，`max` 函数求最大，`min` 函数求最小，`avg` 函数求平均值

25. 查询每个学生的平均成绩【分组聚合】

```
select sid,avg(score) 'avg_score' from gradeinfo  
group by sid;
```

上面的21条语句都是精确查询语句，使用 `=, !=, <, >, in, not in` 来执行过滤条件，其实在查询的时候还有一种查询方式叫模糊查询

模糊查询范例

模块查询以发以下2个点

1. 通配符 `%`
2. 占位符 `_`

模糊查询使用的匹配符号不再是之前精确查询的 = 之类的，它使用 like 来连接查询条件

1. 查询姓名以苏开头的学生

```
select * from stuinfo where sname like '苏%';
```

2. 查询所有生日是15号的学生

```
select * from stuinfo where sbirthday like '%15';
```

3. 查询所有sname包含"小"的学生信息

```
select * from stuinfo where sname like '%小%';
```

上面的3条语句就是模糊查询中的典型代表

4. 查询姓“王”的学生，名字只有1个字的

```
select * from stuinfo where sname like '王_';
```

5. 查询班级里面所有姓名是3个字的学生

```
select * from stuinfo where sname like '___';
```

多表联查【过渡方案】

多表联查就是多个表联合起来一起查询。

有些时候在存储数据的时候，我们的数据并不可能在同一张表里面，而是分别放在了多个表里面，如下所示

sid	sname	ssex	sbirthday	snation	sIDCard	saddr	smail	stel	isDel	cid	sphoto	spwd	slocal
2005010101	苏俊丹	女	1989/9/1	汉族	420984198901湖北省/武汉市	2005010101@13878841001 0			0	2 (Null)	123456 ::ffff:127.0.0.1		
2005010102	张小苗	男	1989/9/2	汉族	420984198901湖北省/武汉市	2005010102@13878941002 0			0	3 (Null)	123456 (Null)		
2005010201	赵希坤	男	1989/9/3	汉族	420984198901湖北省/武汉市	2005010201@13879041003 0			0	3 (Null)	123456 (Null)		
2005010202	阮志婷	女	1989/9/4	汉族	420984198901湖北省/武汉市	2005010202@13879141004 0			0	3 (Null)	123456 (Null)		
2005020101	程丽婷	男	1989/9/5	汉族	420984198901湖北省/武汉市	2005020101@13879241005 0			0	3 (Null)	123456 (Null)		
2005020102	戚正韦	男	1989/9/6	回族	420984198901湖北省/武汉市	2005020102@13879341006 0			0	3 (Null)	147258 ::ffff:127.0.0.1		

上面的表仅仅只显示了这个学生的信息，而没有显示出班级的信息，开始如果要找到班级信息我们应该去另一张表 classinfo 里面去查询信息。

如上面的 cid 中的字段值为1,2,3这到底代表的是什么意思我们需要通过 classinfo 这一张表来看

cid	classnumebr	cname	maxCount	did	instructorid	isDel	tid
1	2741 H1904		55	2	2 0	2005001	
2	2841 J1908		58	1	3 0	20050010	
3	2941 J1909		65	3	2 0	20050011	
4	3041 H1905		62	4	3 0	20050012	
5	1234 h210304		62	4	3 0	20050012	

这个时候就产生了一个现象，如果我们查询学生的时候，需要展示这个学生的信息，还要展示这个班级的信息，那么，我们就要使用多表联合查询

```
select 列名1,列名2 ... from 表名1,表2 ...
```

1. 查询学生信息并显示所对应的班级名称

```
select stuinfo.* , classinfo.cname from stuinfo, classinfo  
where stuinfo.cid = classinfo.cid;
```

2. 查询学生信息，并显示所对应的班级名称，但要求班级是H1904的学生

```
select stuinfo.* , classinfo.cname from stuinfo, classinfo  
where stuinfo.cid = classinfo.cid  
and classinfo.cname = 'H1904';
```

3. 查询学生信息，并显示所对应的班级名称，但要求classnumber是2941的，并且性别是女的学生

```
select stuinfo.* , classinfo.cname from stuinfo, classinfo  
where stuinfo.cid = classinfo.cid  
and classinfo.classnumebr = 2941  
and stuinfo.ssex = '女';
```

4. 查询班级信息，并追加列显示这个班级的实际学生人数

```
select *, (select count(*) from stuinfo where stuinfo.cid = classinfo.cid) as  
'totalCount' from classinfo;
```

上面的SQL语句其实执行效率是非常低的，在正式的开发环境下面是很少使用的

整体上来说多表联查其实是很方便的，可以快速的获取我们想要的数据，但是在工作当中，这种多表联查的写法都是很少去使用的，后面如查想使用多表联多，我们建议使用内联查询或子查询

内联查询与子查询

内联查询其实就是多表联查的另一种写法，这种写法很直观，很方便，并且很好理解，后期我们会使用这一种写法来替代上面多表联查的写法

它的语法格式如下

```
select 列名 ... from 表1  
inner join 表2 on 表1.列 = 表2.列  
inner join 表3 on 表1.列 = 表3.列;
```

1. 查询学生信息并显示所对应的班级名称

```
select stuinfo.* , classinfo.cname from stuinfo  
inner join classinfo on stuinfo.cid = classinfo.cid
```

2. 查询学生信息，并显示所对应的班级名称，但要求班级是H1904的学生

```
select stuinfo.* , classinfo.cname from stuinfo  
inner join classinfo on stuinfo.cid = classinfo.cid  
where classinfo.cname = 'H1904';
```

3. 查询学生信息，并显示所对应的班级名称，但要求classnumber是2941的，并且性别是女的学生

```
select a.* , b.cname from stuinfo a  
inner join classinfo b on a.cid = b.cid  
where b.classnumebr = 2941 and a.ssex = "女";
```

4. 查询班级信息，并追加列显示这个班级的实际学生人数

```
select b.* , a.totalCount from classinfo b  
inner join  
(select cid, count(*) 'totalCount' from stuinfo  
group by cid) as a on a.cid = b.cid;
```

5. 查询所有学生的信息，追加显示班级名称及系别名称【系别是departmentinfo,注意它与classinfo之间的关系】

```
select t1.* , t2.cname, t3.dname from stuinfo t1  
inner join classinfo t2 on t1.cid = t2.cid  
inner join departmentinfo t3 on t2.did = t3.did
```

6. 查询所有学生的信息，追加显示班级名称，系别名称及班主任姓名【系别是departmentinfo,班主任是instructor】

```
select a.* , b.cname, c.dname, d.instructorname from stuinfo a  
inner join classinfo b on a.cid = b.cid  
inner join departmentinfo c on b.did = c.did  
inner join instructor d on d.did = c.did;
```

7. 查询学生成绩表，显示每个学生的平均成绩，同时显示这个学生的姓名【分组聚合 + 内联 + 子查询】

```
select a.* , b.sname from  
(select sid , avg(score) 'avg_score' from gradeinfo  
group by sid) a  
inner join stuinfo b on a.sid = b.sid
```

8. 查询成绩最高分的学生的信息，显示学生的姓名

```
select b.* , c.sname from  
(select max(score) 'max_score' from gradeinfo) a  
inner join gradeinfo b on a.max_score = b.score  
inner join stuinfo c on b.sid = c.sid;  
  
-- 或  
  
select a.* , b.sname from gradeinfo a  
inner join stuinfo b on a.sid = b.sid  
where a.score = (select max(score) from gradeinfo);
```

9. 查询每个学生的平均成绩高于85分的学生

```
select * from  
(select sid, avg(score) 'avg_score' from gradeinfo  
group by sid) a  
where a.avg_score > 85;  
  
-- 更好的写法  
-- 如果我们要在分组聚合以后，再去过滤，可以直接使用一个关键字having  
  
select sid, avg(score) 'avg_score' from gradeinfo  
group by sid  
having avg_score > 85;
```

如果我们希望在分组聚合以后再次去过滤，有2个方法

1. 使用子查询
2. 使用 having 来过滤

10. 在成绩信息表里面，求出每个学生的总成绩

```
select sid,sum(score) 'sum_score' from gradeinfo
group by sid;
```

11. 在成绩信息表里面，求总分最高的学生成绩，并显示学生姓名

```
select b.* ,c.sname from
(select sid,max(sum_score) 'max_sum_score' from
(select sid,sum(score) 'sum_score' from gradeinfo
group by sid) a) b
inner join stuinfo c on b.sid = c.sid
```

上面的SQL语句里面有使用2个嵌套的子查询，但是无论嵌套几个，子查询总是从最里面的括号优先执行

12. 在成绩信息表里，求出学生在期中考试与期末考试的总成绩【分组聚合，注意examinfo这个考试信息表】

```
select sid,eid,sum(score) 'sum_score' from gradeinfo
group by sid,eid;
```

13. 查询老师里面年龄大于“孙鹏”老师的年龄的

```
select * from teacherinfo where tage > (select tage from teacherinfo where
tname = '孙鹏');

-- 或

select b.* from
(select tage from teacherinfo where tname = '孙鹏') a
INNER JOIN teacherinfo b on b.tage > a.tage;
```

这个例子与上面的第8个的例子情况是一样的，展示这个例子主要是为了带出内联查询里面的 on 后面进行链接的时候不仅仅可以只使用 = 还可以使用其它的符号

14. 查询老师里面年龄大于“孙鹏”老师的年龄的女老师

```
select * from teacherinfo where tage > (select tage from teacherinfo where
tname = '孙鹏') and tsex = "女";
select * from
(select tage from teacherinfo where tname = '孙鹏') a
inner join teacherinfo b on b.tage > a.tage
where b.tsex = '女';
```

15. 查询辅导员为桃子的所有学生

```
select a.* ,c.instructorname from stuinfo a
INNER JOIN classinfo b on a.cid = b.cid
INNER JOIN instructor c on b.instructorid = c.instructorid
where c.instructorname = '桃子';
```

16. 把班主任为“桃子”的所有学生分数+1

```
update gradeinfo set score = score+1
where sid in (
    select a.sid from stuinfo a
    INNER JOIN classinfo b on a.cid = b.cid
    INNER JOIN instructor c on b.instructorid = c.instructorid
    where c.instructorname = '桃子')
```

17. 把所有少数民族（非汉族）的学生的考试成绩都+1

```
update gradeinfo set score = score + 0.1
where sid in (select sid from stuinfo where snation ≠ "汉族")
```

18. 在老师信息表里面，加一个称呼，男老师显示小哥哥，女老师显示小姐姐

```
select *,if(tsex = '男','小哥哥','小姐姐') 'nickName' from teacherinfo;

-- 对于条件比较少的，我们可以使用if函数去完成

select *,
(case
    when tsex = "男" then "小哥哥"
    when tsex = "女" then "小姐姐"
    else "人妖"
end) 'nickName'
from teacherinfo;
```

19. 在学生信息表里面，对每个学生的平均成绩做一个评级（在平均成绩的后面追加一个列为评级列）

90分或以上为优秀，80~89为良好，70~79为中等，60~69为及格，否则为不及格`

```
select * ,
(case
    when a.avg_score ≥ 90 then '优秀'
    when a.avg_score ≥ 80 then '良好'
    when a.avg_score ≥ 70 then '中等'
    when a.avg_score ≥ 60 then '及格'
    else '不及格'
end) 'level'
from
(select sid,avg(score) 'avg_score' from gradeinfo
group by sid) a
```

20. 去重 distinct

```
select DISTINCT sid from gradeinfo;
```

如果想单独显示某一列，并且这一列要去重，我们就可以使用DISTINCT

21. 查询一下有谁没有参加期中考试

```
select * from stuinfo
where sid not in
(select DISTINCT sid from gradeinfo where eid = '0801');
```

分页查询

当里面的数据非常多的时候，我们就应该要使用分页查询， MySql里面的分页查询非常简单，只要使用 `limit` 关键字就可以了

它的语法格式就是 `limit offset,count`; `offset`代表要跳过多少条， `count`代表要取多少条

```
-- 分页查询  
-- limit offset,count;  
-- offset代表跳过多少条, count代表取多少条  
-- 查询学生信息 每页只显示5条  
-- 第一页 1~5  
select * from stuinfo limit 0,5;  
-- 第二页: 6~10, 跳过5  
select * from stuinfo limit 5,5;  
-- 第三页: 11~15, 跳过10  
select * from stuinfo limit 10,5;  
-- 第四页: 16~20 跳过15条  
select * from stuinfo limit 15,5;  
-- 第五页: 21~25, 跳过20条  
select * from stuinfo limit 20,5;  
-- 第六页: 26~30, 跳过25  
select * from stuinfo limit 25,5;
```

其实后期的时候，我们做分页查询是有一个公式的， `pageIndex` 代表第几页， `pageSize` 代表每一页显示的数量。 `limit` 的公式是：

```
select * from 表名 limit (pageIndex-1)*pageSize,pageSize;  
-- 假设pageSize = 5 代表每页取5条, pageIndex就是页面  
-- pageIndex = 1 , SQL结果 select * from 表名 limit 0,5  
-- pageIndex = 2 , SQL的结果 select * from 表名 limit 5,5
```

结果集并联

```
-- 查询一共有多少学生  
-- 查询一共有多少老师  
-- 查询一共有多少班级  
  
-- 查询一共有多少学生  
-- 查询一共有多少老师  
-- 查询一共有多少班级  
select count(*) 'totalCount' from stuinfo  
union all  
select count(*) 'totalCount' from teacherinfo  
union all  
select count(*) 'totalCount' from classinfo;
```

当多条SQL语句的查询结果需要并联在一起的时候， 我们就可以使用 `union all` 来完成

一般在并联的时候， 我们都会在前面追加一列名称， 如下所示

```
-- 查询一共有多少学生
-- 查询一共有多少老师
-- 查询一共有多少班级
select 'stuCount', count(*) 'totalCount' from stuinfo
union all
select 'teacherCount', count(*) 'totalCount' from teacherinfo
union all
select 'classCount', count(*) 'totalCount' from classinfo;
```

stuCount	totalCount
stuCount	26
teacherCount	15
classCount	6

但是上面的显示结果并不好，我们要得到下面的结果

stuCount	teacherCount	classCount
21	15	4

行转列

```
select
    max(if(stuCount='stuCount',totalCount,0)) 'stuCount',
    max(if(stuCount='classCount',totalCount,0)) 'classCount',
    max(if(stuCount='teacherCount',totalCount,0)) 'classCoount'
from
(select 'stuCount', count(*) 'totalCount' from stuinfo
union all
select 'classCount', count(*) 'totalCount' from classinfo
union all
select 'teacherCount', count(*) 'totalCount' from teacherinfo) a
```

上面的SQL语句运行以后，就可以得到我们所需要的结果

现在请看下面的2张图

图一	图二

```
select sid,
    max(if(subject = '语文',score,0)) '语文',
    max(if(subject = '数学',score,0)) '数学',
    max(if(subject = '外语',score,0)) '外语'
from abc
```

```
group by sid;

-- 还可以使用 case when去完成

select sid,
       max(case when subject = '语文' then score else 0 end) '语文',
       max(case when subject = '数学' then score else 0 end) '数学',
       max(case when subject = '外语' then score else 0 end) '外语'
  from abc
 group by sid;
```

DML正则查询

在MySQL里面也是支持正则表达式的，但是只支持基础的正则表达式，没有前瞻，没有后顾，也没有原子组编号

1. 查询一个学生，这个学生以蔡开头

```
select * from stuinfo where sname like '蔡%';
-- 以蔡开头 正则里面用^
select * from stuinfo where sname regexp '^蔡';
```

多表联查详细讲解

在明天我们讲解了多表联查，主要是使用2种语法

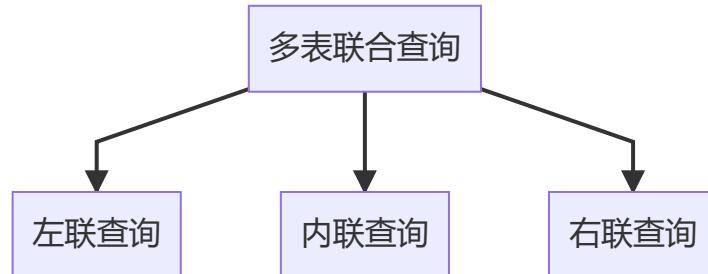
1. 旧的语法

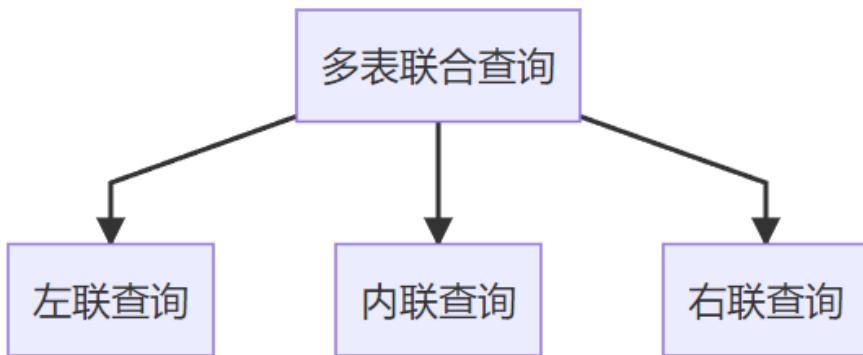
```
select 列名1,列名2 from 表1,表2
where 表1.列 = 表2.列;
```

2. 新的写法：内联查询

```
select 列名1,列名2 from 表1
inner join 表2 on 表1.列 = 表2.列;
```

这里面使用的 `inner join` 我们就叫内联查询





stu1		
sid	sname	ssex
1	岳圣哲	男
2	黄相立	男
3	马淑圆	女

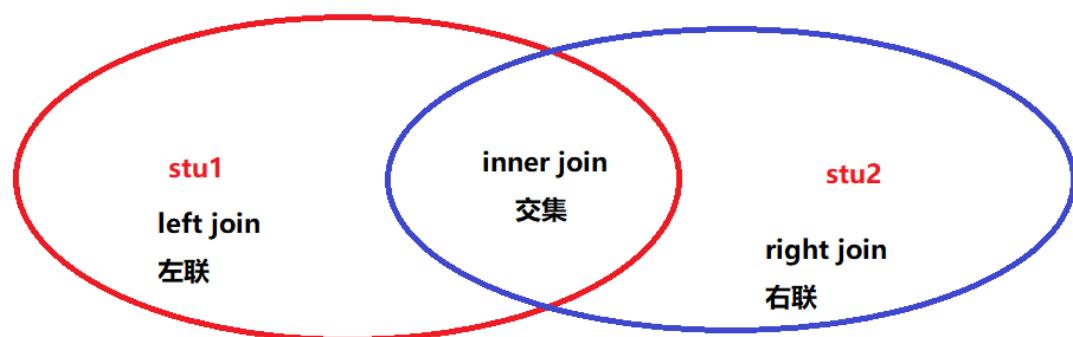
stu2		
sid	sname	hobby
1	岳圣哲	看书
2	黄相立	睡觉
4	陈怡静	玩游戏

现在我们有了上面的两张表，然后我们去执行内联查询

```
select * from stu1
inner join stu2 on stu1.sid = stu2.sid;
```

sid	sname	ssex	sid(1)	sname(1)	hobby
1	岳圣哲	男	1	岳圣哲	看书
2	黄相立	男	2	黄相立	睡觉

这个时候我们可以碰到，我们得到的结果只有2条数据



- `left join` 左联查询，结果以左边的表为主
- `inner join` 内联查询，取两个表的交集部分
- `right join` 右联查询，结果以右边的表为主

左联查询

```
select * from stu1
left join stu2 on stu1.sid = stu2.sid;
```

sid	sname	ssex	sid(1)	sname(1)	hobby
	1 岳圣哲	男	1	岳圣哲	看书
	2 黄相立	男	2	黄相立	睡觉
	3 马淑圆	女	(Null)	(Null)	(Null)

右联查询

```
select * from stu1
right join stu2 on stu1.sid = stu2.sid;
```

sid	sname	ssex	sid(1)	sname(1)	hobby
	1 岳圣哲	男	1	岳圣哲	看书
	2 黄相立	男	2	黄相立	睡觉
	(Null) (Null)	(Null)		4 陈怡静	玩游戏

主外键与视图详细讲解

关于主键

什么是主键？

1. 不能为空
2. 不能重复

一个数据表如果有了主键以后，它的操作效率将会大大提升，正常情况下，同学们如果想设计数据表，都应该要设置主键

我们现在尝试创建一个没有主键的表

开始事务			文本	筛选	排序	导入	导出
sname	ssex	sage					
杨标	男	18					
张三	男	19					
李四	男	20					
杨标	男	20					

现在在上面的表格里面，我想删除杨标，年龄为20的这条记录，怎么办？

```
delete from stuinfo where sname = "杨标" and sage = 20;
```

这个时候我们就可以看到，操作起来会变得非常麻烦，因为我们要找到所需要的数据，需要使用2个条件，但是如果后期的列变多了，我们操作就会变得更加麻烦

sname	ssex	sage
杨标	男	18
张三	男	19
李四	男	20
杨标	男	20
▶ 杨标	男	20

如果现在我们想删除最后一条记录，这个时候使用上面的SQL语句就不能完在了

现在我们添加一个id列为主键列，再插入数据如下所示

id	sname	ssex	sage
1	杨标	男	18
2	岳圣哲	男	20
3	陈怡静	女	21
4	杨标	男	20

现在有了id做主键，那么，我们如果需要删除最后一条记录，只需要执行下面的SQL即可

```
delete from stuinfo where id = 4;
```

但是现在又有一个问题，如果我们添加了一个主键id列以后，我们每次在插入数据的时候都要手动的填写这个id，很麻烦

字段	索引	外键	触发器	选项	注释	SQL 预览
名						
id				<input checked="" type="checkbox"/> 自动递增	<input checked="" type="checkbox"/> 不是 null	<input checked="" type="checkbox"/> 键 1
sname				<input type="checkbox"/>	<input checked="" type="checkbox"/>	
ssex				<input type="checkbox"/>	<input checked="" type="checkbox"/>	
sage				<input type="checkbox"/>	<input checked="" type="checkbox"/>	

默认：
 自动递增
 无符号
 填充零

有了这个自增的id以后，我们后面再添加数据的时候就不用再管这一项了，它会自动的递增

同时，在执行新增的SQL语句的时候，这个自增的列就可以不用管了

```
insert into stuinfo (sname,ssex,sage) values ("小宝贝","女",18);
```

使用DDL语句建表时设置自增的主键ID

```

create table if not exists stuinfo (
    id int primary key auto_increment,
    sname varchar(20) not null,
    ssex varchar(2) not null,
    sage int not null
)engine = innodb default charset = utf8mb4;

```

关键就是 `auto_increment` 的设置

关于外键

当一个表的某一列引用了另一个表的主键存储，那么这个表的这一列我们就叫外键列

班级信息表		
班级编号	班级名称	班级地址
1	H2201	金融港A4
2	H2202	金融港A4
3	H2203	金融港B27

学生信息表			
学生编号	学生姓名	学生性别	所属班级
1	张珊	女	H2201
2	李四	男	H2202
3	王五	男	C2203
4	赵六	男	H2201
5	田七	女	J2203

在上面的图里面，我们看到，当一个学生去存储信息的时候，我们在所属班级这个地方可以填写任意字段，这个时候就发现问题了，班级表里面没有 `C2203` 和 `J2203`，这个时候的数据就出现冲突了

按照正常的逻辑，一个学生应该是属于一个班级，如果这个班级都不存在，那么这个学生应该不能存在

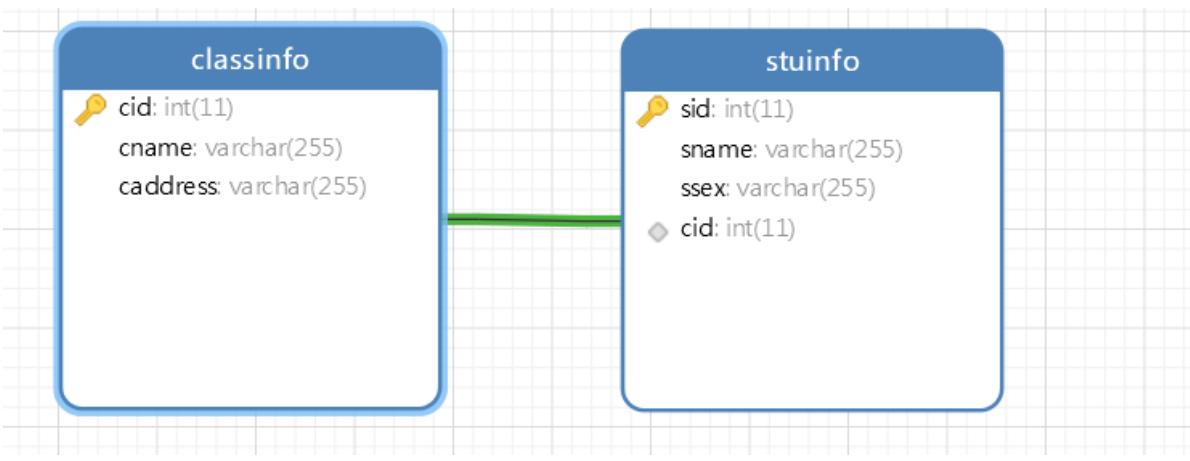
现在我们就要换一种方式去存储（ MySql关系型数据库），表与表之间是可以构建关系的（这个关系主要就是我们常说的主外键约束关系）

班级信息表		
班级编号	班级名称	班级地址
1	H2201	金融港A4
2	H2202	金融港A4
3	H2203	金融港B27

学生信息表			
学生编号	学生姓名	学生性别	所属班级
1	张珊	女	1
2	李四	男	2
3	王五	男	2
4	赵六	男	3
5	田七	女	1

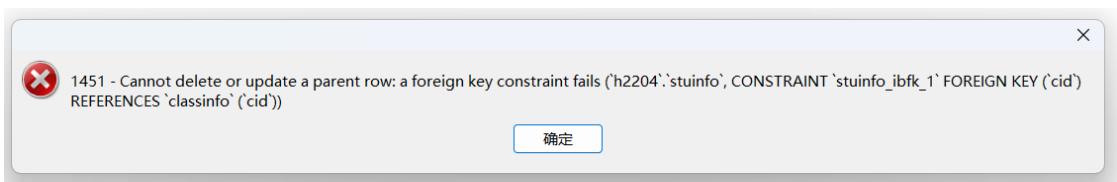
在上面的图里面，我们把学生信息表里面的 `所属班级` 改变了一下，这个表的这个字段引用了班级信息表里面的 `班级编号`【现在我们就发现，一个表（学生信息表）引用了另一个表（班级信息表）的主键列，这个时候我们就把这一列 `所属班级` 叫外键列】

外键：引用了另一个表的主键存储



上面的数据表就形成主外键的约束关系，它们的约束主要在于以下几点

1. stuinfo表里面的cid必须来源于classinfo里面的cid
2. 主键表里面不能随意删除数据，因为可能会被另一张表使用，如果删除时会报错



3. 如果非要删除一个主键数据，则这一行数据还没有被引用就可以了
4. 要注意主外键的级联状态



当我们去创建主外键的时候，默认它们之间的级联状态是上图的样式

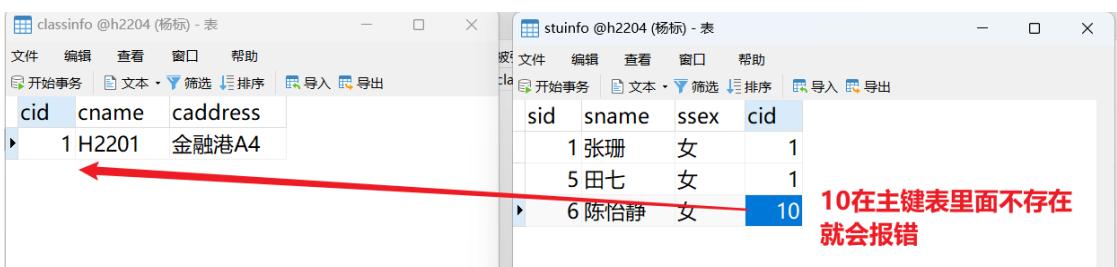
默认是 **restrict** 拒绝，约束状态



当我们把这个状态换成 **cascade** 级联状态以后，这样在删除数据的时候，所有的外键数据自动删除

构建主外键的前提条件

1. 主键的数据类型与外键的数据类型必须保持一致
2. 数据冲突



3. 数据表的引擎 **engine** 必须是 **innodb**，如果是 **MyISAM** 就不可以（这一种存储方式不支持主外键约束）

```

create table if not exists stuinfo(
    id int primary key auto_increment,
    sname varchar(20) not null,
    ssex varchar(2) not null,
    cid int not null,
    FOREIGN key(cid) REFERENCES classinfo(cid) on delete CASCADE on update CASCADE
)engine = innodb default charset = utf8mb4;

```

数据库视图

视图是一个虚拟的表，它的本质上面就是一段SQL语句，至于视图有什么作用，我们先通过下面的场景来实现

场景：查询学生信息，显示这个学生的所有信息，追加显示班名名称，系统别名名称，辅导员名称

```

select a.* , b.cname , c.dname , d.instructorname from stuinfo a
inner join classinfo b on a.cid = b.cid
inner join departmentinfo c on c.did = b.did
inner join instructor d on d.instructorid = b.instructorid

```

id	结果 1 剖析 状态	sIDCard	saddr	smail	stel	isDel	cid	sphoto	spwd	slocal	stime	cname	dname	instructornan
4209841989090710	湖北省/武汉市2005020201@13879441007 0	1 (Null)	123456 (Null)	(Null)	H1904	Java	蒋晓菁							
4209841989090810	湖北省/武汉市2005020202@13879541008 0	1 (Null)	123456 (Null)	(Null)	H1904	Java	蒋晓菁							
4209841989091010	湖北省/武汉市2005030102@13879741010 0	1 (Null)	123456 (Null)	(Null)	H1904	Java	蒋晓菁							
4209841989091210	湖北省/武汉市2005040102@13879941012 0	1 (Null)	123456 (Null)	(Null)	H1904	Java	蒋晓菁							
4209841989091410	湖北省/武汉市2005040202@13880141014 0	1 (Null)	123456 (Null)	(Null)	H1904	Java	蒋晓菁							
4209841989091610	湖北省/武汉市2005050102@13880341016 0	1 (Null)	123456 (Null)	(Null)	H1904	Java	蒋晓菁							
4209841989091810	湖北省/武汉市2005050202@13880541018 0	1 (Null)	123456 (Null)	(Null)	H1904	Java	蒋晓菁							
4209841989092010	湖北省/武汉市2005050209@13880741020 0	1 (Null)	123456 (Null)	(Null)	H1904	Java	蒋晓菁							
4209841989090310	湖北省/武汉市2005010201@13879041003 0	3 (Null)	123456 (Null)	(Null)	J1909	Python	蒋晓菁							
4209841989090410	湖北省/武汉市2005010202@13879141004 0	3 (Null)	123456 (Null)	(Null)	J1909	Python	蒋晓菁							
4209841989090510	湖北省/武汉市2005020101@13879241005 0	3 (Null)	123456 (Null)	(Null)	J1909	Python	蒋晓菁							
4209841989090610	湖北省/武汉市2005020102@13879341006 0	3 (Null)	147258::ffff:127.0.0.1 2019-11-07 2U1909	Python	蒋晓菁									

经过上面的SQL语句的执行以后，我们也得到我们想得到的结果，怎么样把这个结果保存起来做二次查询呢

我们如果要将一个查询结果保存起来，有一个方法，主要是把它变成一个视图（虚拟的表）

使用DDL语句来创建视图

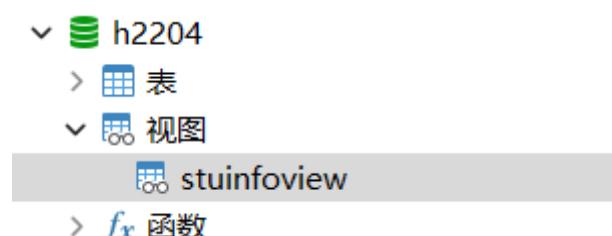
视图是一个虚拟的表，它的本质上面就是一段SQL语句

```

create view stuinfoview
as
(select a.* , b.cname , c.dname , d.instructorname from stuinfo a
inner join classinfo b on a.cid = b.cid
inner join departmentinfo c on c.did = b.did
inner join instructor d on d.instructorid = b.instructorid)

```

当上面的代码执行完毕以后，我们可以看到下面的结果



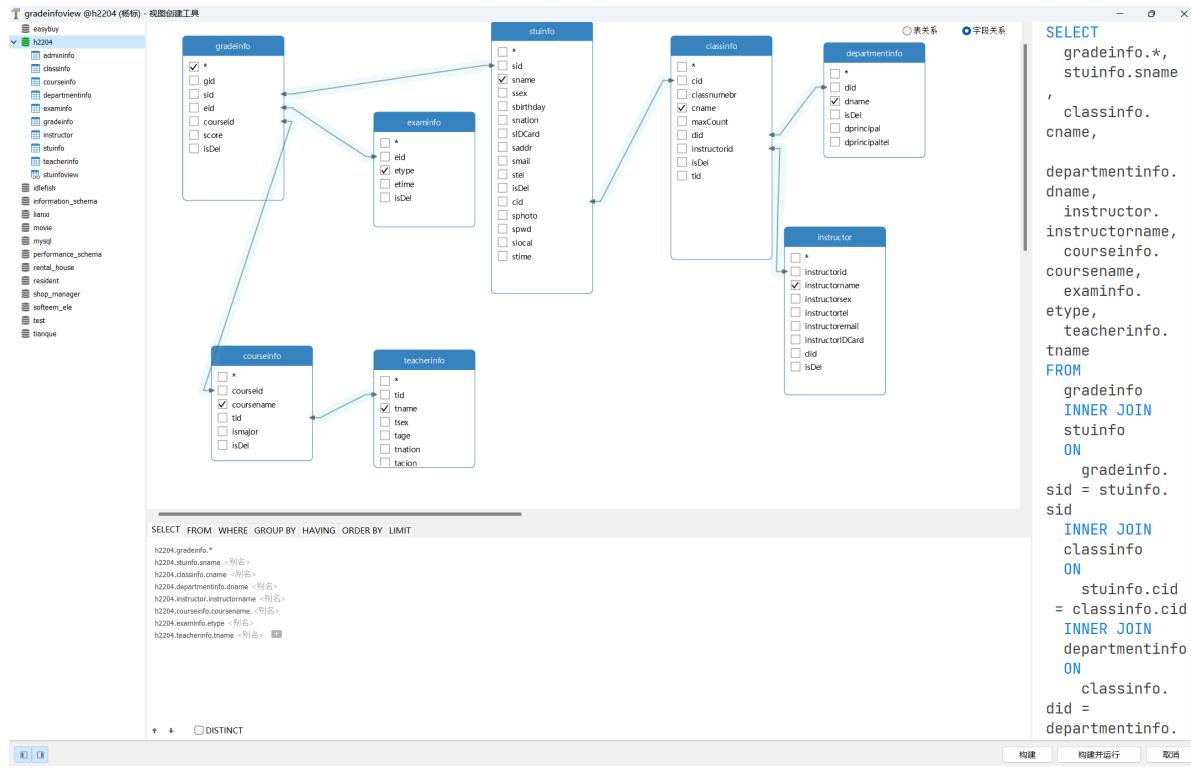
当视图创建成功以后，我们就可以像使用表一样去使用视图

```
select * from stuinfoview where instructorname = '桃子'
```

注意事项：视图本质上是一串SQL语句，它只能执行查询操作，不能执行insert/delete/update

使用navicat来创建视图

我要创建一个学生信息的视图，里面显示学生信息，追加显示班级名称，辅导员名称，系别名称



扩展

1. MySQL里面有哪些内置函数

- `max()`
- `min()`
- `count()`
- `avg()`
- `sum()`
- `if()`
- `year() / month() / day()`
- `left()` 左截取
- `right` 右截取
- `mid()` 中间截取
- `concat()` 拼接
- `rand()` 随机数
- `floor()` 向下取整
- `ceil()` 向上取整
- `now()` 当前日期时间
- `CURRENT_TIMESTAMP` 当前时间戳
- `trim() / ltrim() / rtrim()` 去除空格
- `UUID()` 生成一个uuid的全局唯一标识符

2. 如何在MySQL里面创建新用户，并限定新用户的权限

node.js连接mysql数据库

现在我们学习的node.js平台，以前大家应该听说过一个架构 `php+mysql`，目前在前端里面我们使用 `node.js` 代表了 `php` 来实现服务器的后端的功能

在nodejs下面如果我们要操作mysql的数据库，我们需要借用于第三方的模块，`mysql` 或 `mysql2`

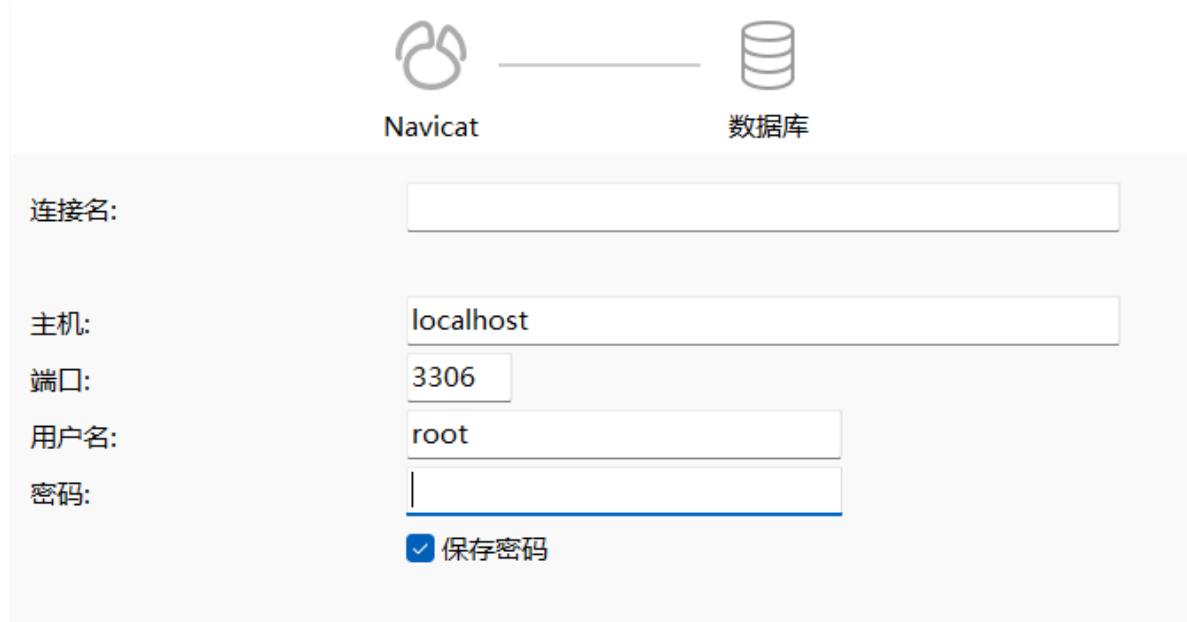
对于低版本的mysql的数据，我们可以使用 `mysql` 的包来完成，对于 `mysql7` 以上的版本我们则要使用 `mysql2` 去完成操作

安装第三方模块mysql

```
$ npm install mysql --save
```

连接mysql数据库

如果我们的程序要连接mysql的数据库，至少需要下面的信息



操作mysql数据库

```
/**  
 * 连接mysql数据库以后，步骤是什么？？？  
 */  
  
//第一步：看一下包准备好了没有  
//第二步：导入这个包  
const mysql = require("mysql");  
  
//第三步：创建了数据库的连接  
let conn = mysql.createConnection({  
    host: "127.0.0.1",  
    port: 3306,
```

```

    user: "sg",
    password: "123456",
    database: "h2204"
});

//第四步：开始连接
try {
    conn.connect();
    //第五步：准备要执行的SQL语句
    let strSql = `select * from stuinfo;`;
    //第六步：通过打开的链接去执行这一条SQL语句
    conn.query(strSql, (error, result) => {
        //error代表这条SQL语句执行失败,result代表SQL语句执行结果
        if (error) {
            console.log(error);
            console.log("数据库执行失败");
        }
        else {
            console.log(result);
            console.log("数据库执行成功");

        }
        //第七步：关闭链接
        conn.end();
    });
} catch (error) {
    console.log(error);
    console.log("数据库连接失败");
}

```

```

YangBiao@YB-Huawei D:\杨标的互作文件\班级教学笔记\H2204\1012\code\mysql_demo1 ))> node 01.js
[
  RowDataPacket {
    sid: 1,
    sname: '张璐',
    sage: 18,
    ssex: '女',
    shobby: '看书'
  },
  RowDataPacket {
    sid: 2,
    sname: '李四',
    sage: 19,
    ssex: '男',
    shobby: '睡觉'
  }
]
数据库执行成功

```

小练习

请将标哥数据库当中的stuinfo表导出为一个excel文件