

SignalR

1、SignalR 介绍。

Http 协议是浏览器端主动发起请求，服务器不能主动发起请求。有一些场景下需要服务器主动通知浏览器端，比如网站即时消息、审核通知、系统报警。可以使用 Ajax 长连接来解决，但是对于 Web 服务器的压力太大，尽量别用。如果浏览器端支持 WebSocket（基本上现代浏览器都支持），那么可以利用 WebSocket 实现，性能非常高。

使用原生 WebSocket 开发难度比较高，微软提供了一个高度封装的框架 SignalR，简化了 Websocket 开发。如果浏览器端支持 WebSocket 就用 Websocket，否则就使用长连接，代码不变。

SignalR 在 Windows7、Windows 2008 下作为服务器也可以正常调试运行，但是是用长连接的，服务器端只有在 Windows8 及以上（不会用作服务器系统）、Windows 2012 及以上才支持 WebSocket，服务器端的 WebSocket 支持如果没安装还需要安装 <https://docs.microsoft.com/en-us/iis/get-started/whats-new-in-iis-8/iis-80-websocket-protocol-support>

总结：浏览器端支持 Html5 WebSocket 并且服务器端为 Windows 2012 及以上才支持 WebSocket，否则性能惨死！

目前 .net core 2.0 还不正式支持 SignalR，所以这个项目还不能 .net core 开发，只能先用 .net framework 开发。

2、SignalR 基本使用

- 新建普通 ASP.Net 空项目，其实是可以独立的一个空项目的，也可以和 WebForm、asp.net MVC 等配合使用。我们后续要用到 MVC，所以这里再勾上 MVC。
- 新建项，选择【SignalR】下的【SignalR 集线器类】（【SignalR 持久连接】是底层机制），命名为 TestHub，内容是：

```
using Microsoft.AspNet.SignalR;
```

```
namespace WebApplication1
```

```
{
```

```
    public class TestHub : Hub
```

```
    {
```

```
        public void SendMessage(string name, string msg)
```

```
        {
```

```
            Clients.All.onMessage(name+"和我说"+msg); //调用所有连接上来的客户端（包括自己）
```

监听的 onMessage 事件。All 是一个 dynamic 属性，所以可以随意的监听

```
        }
```

```
    }
```

```
}
```

- 新建 mvc 的 View 页面 Index.cshtml，然后在头部引入自动生成的 Scripts 文件夹下的 jquery***.js 和 jquery.signalR***.js
- 再加入 `<script src="/signalr/hubs" type="text/javascript"></script>` 这是一个由 SignalR 处理的路径

- e) 如果没有建 **Startup** 文件可能会报错, 建一个 **startup** 文件就可以了, 并且要在 **Configuration** 中配置 **app.MapSignalR()**

```
using Microsoft.Owin;
```

```
using Owin;
```

```
[assembly: OwinStartup(typeof(WebApplication1.Startup))]
```

```
namespace WebApplication1
```

```
{
```

```
    public class Startup
```

```
    {
```

```
        public void Configuration(IApplicationBuilder app)
```

```
        {
```

```
            app.MapSignalR();
```

```
        }
```

```
    }
```

```
}
```

- f) 编写页面代码

```
<html>
```

```
<head>
```

```
    <meta name="viewport" content="width=device-width" />
```

```
    <title>Index</title>
```

```
    <script src="~/Scripts/jquery-1.10.2.js"></script>
```

```
    <script src="~/Scripts/jquery.signalR-2.1.2.js"></script>
```

```
    <script src="~/signalr/hubs" type="text/javascript"></script>
```

```
    <script type="text/javascript">
```

```
        $(function () {
```

```
            $.connection.testHub.client.onMessage = function (msg)//监听来自服务器端对onMessage  
的调用。注意是client.
```

```
            {
```

```
                alert("消息来了"+msg);
```

```
            }
```

```
            //要在hub.start()之前执行
```

```
            $.connection.hub.start().done(function () {
```

```
                alert("连接testHub成功");
```

```
            }).fail(function () {
```

```
                alert("连接testHub失败");
```

```
            });
```

```
            $("#btn1").click(function () {
```

```
                $.connection.testHub.server.sendMessage("yzk", "hello");//调用服务器端Hub的
```

```
sendMessage方法
```

```
            });
```

```
        });
```

```
    </script>
```

```
</head>
<body>
    <input type="button" id="btn1" value="click"/>
</body>
</html>
```

再开一个浏览器两个对聊，会发现都能收到对方发来的消息。

3、Hub 类的成员

- a) Clients 属性代表所有连接到 Hub 上的客户端；Groups 属性代表群组。后面后详细讲
- b) Context 属性代表请求的上下文，是 `HubCallerContext` 类型。
 - i. `HubCallerContext` 类型的 `ConnectionId` 属性代表的是连接的 Id，一个客户端连接对应一个 `ConnectionId`，本次连接期间这个 `ConnectionId` 一般不变化，到那时开新的页面或者刷新页面会得到一个新的 Id；
 - ii. `RequestCookies` 属性代表请求的 cookie；这个是和 Web 端共享 Cookie 的，所以只要在 Web 端或者在浏览器端设置 Cookie 即可。不过用 Cookie 耦合性太强，尽量不用 Cookie 机制。
 - iii. `Headers` 属性代表请求的报文头，貌似无法在 js 端设置自定义报文头；
 - iv. `QueryString` 属性代表请求的 `QueryString`，这个 `QueryString` 在 `hub.start()` 前通过 `connection.hub.qs = { username: "hello", age: 8 }`；这种方式设置，这个设置是全局的，对于当前连接的所有 SignalR 请求都会带着这个。注意一定要在 `hub.start()` 之前设置。通常用来设置全局的验证信息等，普通的数据通过 Hub 的方法来传递即可。后续不能再修改 `connection.hub.qs`。
- c) 几个可以重载的方法：`OnConnected()`，当连接的时候调用；`OnDisconnected()` 断开连接的时候调用；`OnReconnected()`，当网络不稳定导致断开连接以后重连时候调用，注意刷新页面不是重连。可以通过这个来跟踪用户的在线状态（保存到 Redis 等地方）。
- d) 结合 Redis 实现一下在线人数和谁在线的统计。

4、SignalR 群组管理

SignalR 的 Hub 提供了群组机制，可以把用户放到某一个或者多个组中，然后针对组进行消息推送。客户端和组之间是多对多的关系

Hub 的 Groups 属性是 `IGroupManager` 类型的，两个方法 `Add(string connectionId, string groupName)`、`Remove(string connectionId, string groupName)` 分别是根据 `connectionId` 把连接加入或者从某个组移除。组的名字自定义，如果之前不存在这个组，会自动创建一个组，因此不需要单独创建组，相同 `groupName` 为一组的。建议在 `OnDisconnected()` 中调用 `Remove` 来移除组。

Hub 的 Clients 属性是 `IHubCallerConnectionContext<dynamic>` 类型的、`IHubCallerConnectionContext` 又继承了 `IHubConnectionContext`。所有的成员几乎都是 `dynamic` 类型的，所以怎么操作都可以。主要成员有：

- 1) Caller 当前连接的客户端，可以通过 `Caller.aaa()`；直接调用浏览器端的 function
- 2) Others，除了当前连接外的其他客户端
- 3) `OthersInGroup(string groupName)` 名字为 `groupName` 的组的所有成员。
- 4) `OthersInGroups(IList<string> groupNames)` 在 `groupNames` 这些组中的所有成员
- 5) All：所有连接的客户端
- 6) `AllExcept(params string[] excludeConnectionIds)` 除了指定的多个 `ConnectionId` 之外的其他客户端
- 7) `Client(string connectionId)` 指定的 `ConnectionId` 对应的客户端
- 8) `Clients(IList<string> connectionIds)` 指定的多个 `ConnectionId` 对应的客户端
- 9) `Group(string groupName, params string[] excludeConnectionIds)` 组 `groupName` 中除了

excludeConnectionIds 之外的客户端

- 10) Groups(IList<string> groupNames, params string[] excludeConnectionIds) 组 groupNames 中除了 excludeConnectionIds 之外的客户端

这些成员的返回值都是 `dynamic` 类型的，有的对应一个客户端，有的对应多个客户端，无论对应几个，直接通过 `dynamic` 调用通过 `Clients.All.onMessage("hello")` 这样方式通知到所有相关的客户端的 `js function` 中。

- 5、Hub 的方法支持异步，方法声明为 `async Task<T>` 就可以了。如果 Hub 中调用了异步的方法，那么一定要写成 `async`。不能用 `TestAsync().Result` 这种写法，否则 Hub 方法只能调用一次就卡住了，后续调用就没反应了，不要使用 `WebClient` 的异步，都换用 `HttpClient`

- 6、SignalR 返回：Hub 中方法，客户端直接在方法调用之后的 `done` 事件里回调获得返回值，

```
public async Task<Person> hello()
{
    Person p = new Person();
    p.Id = 333;
    p.Name = "tom";
    return p;
}

chatHub.server. hello ().done(function (res) {
    alert(res);//拿到返回值
});
```

返回值一般是用来判断这次调用成功还是失败，如果失败的话失败原因是什么

- 7、Hub 的方法中如果发生异常，可以在 `fail()` 回调方法中得知：
`hub.test().done(function(){alert("成功");}).fail(function(){alert("错误");});`
- 8、虽然 SignalR 是在 ASP.Net 项目中运行的，但是 Hub 中的未处理异常是不会被 MVC 的 `ExceptionHandler` 截获的。在服务器端可以通过自定义 `HubPipelineModule` 来在服务器端捕获未处理异常，然后记录到日志等：

- a) 首先编写类继承自 `HubPipelineModule`

```
public class ExceptionHubPipelineModule: HubPipelineModule
{
    protected override void OnIncomingError(ExceptionContext exceptionContext,
                                           IHubIncomingInvokerContext invokerContext)
    {
        //exceptionContext.Error; 就是异常对象 可以记录到日志中
    }
}
```

- b) 在 Global 的 `Application_Start()` 中编写 `GlobalHost.HubPipeline.AddModule(new ExceptionHubPipelineModule ());`

- c) 像上面讲的，未处理异常可以在客户端每个方法的 `fail` 中得知，如果想统一在客户端得知异常，那么可以在浏览器端写一个 `function`，然后在服务器端调用这个 `function`。比如

浏览器端 `start` 之前写一个

```
$.connection.testHub.client.onGlobalException = function (errMsg)
{
```

```
alert("未处理异常"+errMsg);  
}  
然后 ExceptionHubPipelineModule 中写  
dynamic caller = invokerContext.Hub.Clients.Caller;  
caller. onGlobalException(exceptionContext.Error.Message);
```

9、Hub 获取登录状态

Hub 和 asp.net 的运行线程是独立的，在 Hub 中不能直接读取 asp.net 的 Session 登录时在 Web 中的，如何在 Hub 中获取登录信息呢？

可以在 ASP.Net 中生成一个类似于 SessionId 的唯一标识，然后在登录的时候把唯一标识和用户 Id 的对应关系记录到 Redis、Memcache 等地方，然后通过 Cookie 或者 QueryString 把这个唯一标识发给 Hub，在 Hub 中再根据这个唯一标识获取登录用户 Id。

也可以用 JWT 算法把用户的 Id 等信息保存成 Token 字符串，然后通过 Cookie 或者 QueryString 把这个唯一标识发给 Hub，在 Hub 中再解析 Token 获取登录用户 Id。

写程序演示一下。

- 10、缓存。Hub 中不能用 asp.net 的 cache，可以使用 Memcached 等缓存。当然如果缓存数据量不大的话，也可以使用 System.Runtime.Caching 这个程序及中的 MemoryCache。
- 11、SignalR 在 Chrome 等浏览器下同一个域名有最多连接数的限制，超过之后就卡住了，因此最好只有一个窗口获取消息，其他窗口使用父子窗口通讯或者 Html5 的 postMessage 等方法进行窗口间通讯。这个项目中会演示。

Vue.js

- 1、以前我们用 JSDom 或者 JQuery 写网站的时候，经常要进行 html 字符串拼接，而且还要清楚的了解 dom 结构、事件监听等，非常复杂且难以维护。

如下图的感觉：

```
modalHtml += "</div>";
modalHtml += "<div id='shareQrCode'>";
modalHtml += "</div>";
modalHtml += "<div>";
modalHtml += "注意: 我们的系统有防作弊机制, 对于非正常访问不会记录积分。";
modalHtml += "</div>";
modalHtml += "</div>";
modalHtml += "</div>";
modalHtml += "</div>";
modalHtml += "<script type='text/javascript' src='http://static.rupeng.com/st";
modalHtml += "<script type='text/javascript' src='http://static.rupeng.com/st";

$("body").append($(modalHtml)); //将分享对话框附加到body的最后
}
//点击分享按钮显示分享对话框
$("#btnShowShare").click(function () {
    var _desc = document.title;
    var url = document.URL;
    if (window.userId)
    {
        url += '#p=' + window.userId;
    }
    _desc += url;
    $("#shareurl").val(_desc);
    $("#shareQrCode").html(""); //清空之前的, 避免重复生成
    $("#shareQrCode").qrcode(url);
    $("#rpshare").modal('show');
    return false;
});
checkNavActive(); //判断当前连接选中状态
```

VueJs 路径: <https://cn.vuejs.org/v2/guide/>

```
<script src="https://unpkg.com/vue"></script>
```

Vue.JS、Angular、React 等框架让我们编写网页不再进行 html 字符串拼接, 而是以一种 MVC 的模式去开发, 结构更清晰。看看下面的例子:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <script src="https://cdn.bootcss.com/vue/2.3.3/vue.min.js"></script>
</head>
<body>
    <div id="app">
        <input type="text" v-model="i1"/>+
        <input type="text" v-model="i2" />
        <input type="button" value="=" v-on:click="calcAdd"/>
        <span>{{i3}}</span>
    </div>
    <script type="text/javascript">
//要写到元素的后面, 当然也可以写到window.onload中
var app = new Vue({
    el: '#app',
    data:
```

```
{
  i1: 0, i2: 0, i3: 0
},
methods:
{
  calcAdd: function ()
  {
    this.i3 = parseInt(this.i1) + parseInt(this.i2);
  }
}
});
</script>
</body>
</html>
```

做好 View 和属性绑定之后，我们只要写业务逻辑就可以，不需要在代码中再去操作 Dom。

VueJS2.x 比 1.x 变化了很多，看文档别搞乱了

`new Vue({el: '#app'})`;这是创建 Vue 的“势力范围”，在 el 指定选择器范围内的元素都受 Vue 控制。很显然 `new Vue({el: '#app'})`;要放到 html 元素之后或者放到 `window.onload` 中。

Vue 的 data 段中定义了模型数据，这些数据可以绑定到视图上，methods 段中定义了事件处理的函数。

v-model 是用来把输入控件和 model 中某个属性进行绑定的 VueJS 自定义的属性，这样这个输入控件的值就会和模型的属性值进行“双向绑定”（model 变了界面会变，界面中输入变化了 model 中也会变）。

如果只是简单的显示，那么就用 `{{i3}}` 就行了

`v-on:click="calcAdd"` 表示当点击这个按钮的时候执行 methods 中名字为 calcAdd 的方法。注意不要加括号

在 Vue 的 methods 段中定义的方法中 `this.i1` 表示当前 Vue 对象的 data 模型中的 i1，不能不写 this。注意给 data 模型赋值的时候不能给 data 重新赋值。

Vue 中所谓的 data、methods 都是符合标准 JS 语法的，因此不能写错了。

再体会一下双向绑定（View 和 Model 的改变可以影响对方）：

增加一个按钮：

```
<input type="button" value="改变 i1" v-on:click="changei1"/>
```

在 methods 中增加 changei1 方法（时刻注意符合 js 语法）：

```
changei1: function ()
```

```
{
  this.i1 = 666;
}
```

想一下上面的代码如果用 JSDom 去写要多么复杂？

2、数据绑定的几种方式

1) 双向绑定用 `<input type="text" v-model="i1"/>`，双向绑定只限于输入控件

2) 只是显示的话用 `{{i1}}`，这种方式比较灵活。但是主要 `{{i1}}` 不是像 chtml 中的 @ 那样可以随意用，比如下面的代码不会如愿：

```
<span title="{{i3}}">{{i3}}</span>
```

要改成:

`{{i3}}` 表示 title 属性绑定到 i3。注意 v-bind 里的值不用双大括号, v-bind 后的名为标签的属性名。v-bind 是模型到 view 的单向绑定。

3) 尽管我们不应该在 js 中再去拼接 html, 但是有时候我们会通过其他一些途径拿到一些 html 格式的内容要显示到网页中。比如 data 中增加一项 `msg:'rupeng'`

如果这样显示 `{{msg}}` 会发现显示到页面上进行了 Html 转义, 主要是为了防止 XSS。

如果想强制不转义显示那么使用 v-html, 如下: ``

表示: 把 msg 属性的值作为这个 span 的 innerHtml。

3、如何控制元素的显示

绑定 v-show 属性。比如在 data 中增加: `isVip:true`, 然后 `<div v-show="isVip">老子是 vip</div>` 然后通过按钮切换 isVip 的值试一试。

也可以使用 v-if 实现同样的效果, 区别是: v-if 是动态增减节点, v-show 只是控制样式显示隐藏。

更多区别: <http://www.cnblogs.com/wmhuang/p/5420344.html>

v-show、v-if 也支持运算表达式, 比如: `<div v-show="i1>2">老子是 vip</div>` 这样输入改变 i1 的时候就会切换 div 的显示。

4、v-for 用来遍历生成节点

```
<div id="app">
  <ul>
    <li v-for="person in persons">Id:{{person.id}} name:{{person.name}} age:{{person.age}}</li>
  </ul>
</div>
<script type="text/javascript">
  var app = new Vue({
    el: '#app',
    data:
    {
      persons: [{ id: 1, name: 'yzk', age: 18 }, { id: 22, name: 'tom', age: 22 }, { id: 36, name:
'jerry', age: 8 }]
    }
  });
</script>
```

动态给集合添加元素 (比如 ajax 动态加载), 要往 data 对象中添加, 不能覆盖 data 中的对象:

```
<div id="app">
  <ul>
    <li v-for="person in persons">Id:{{person.id}} name:{{person.name}} age:{{person.age}}</li>
  </ul>
  <div>
    id:<input type="text" v-model="pId"/>
    Name:<input type="text" v-model="pName" />
    Age:<input type="text" v-model="pAge" />
  </div>
</div>
```

```
<input type="button" v-on:click="addNew" value="新增"/>
</div>
</div>
<script type="text/javascript">
  var app = new Vue({
    el: '#app',
    data:
    {
      pId:"pName:",pAge:",
      persons: [{ id: 1, name: 'yzk', age: 18 }, { id: 22, name: 'tom', age: 22 }, { id: 36, name:
'jerry', age: 8 }]
    },
    methods:
    {
      addNew: function () {
        this.persons.push({ id: this.pId, name: this.pName, age: this.pAge });
      }
    }
  });
</script>
```

如何在 for 里面的元素事件里获得当前元素的信息,比如点击 li 的时候 alert 显示当前用户的名字:

```
<li v-for="person in persons" v-on:click="personClick(person)">Id:{{person.id}} name:{{person.name}}
age:{{person.age}}</li>
```

在 vue 的 methods 里面增加一个函数:

```
personClick: function (p)
{
  alert(p.name);
}
```

根据输入自动对显示的数据进行过滤或者排序:

使用计算函数,在 Vue 中的 computed 中定义的函数,可以当成数据一样用(其实也可以定义在 methods 中,但是定义在 computed 中可以利用缓存)

computed:

```
{
  test1: function ()
  {
    return "hello" + this.searchName;
  },
  test2: function ()
  {
    return [{ name: 'tom', imgSrc: '1.jpg' }, { name: 'jacky', imgSrc: '2.jpg' }];
  }
}
```

然后 html 中:

test2:

```
<input type="text" v-model="searchName" />
```

```
<ul>
```

```
  <li v-for="item in test2">{{item.name}} </li>
```

```
</ul>
```

```
test1:{{test1}}
```

Vue 能够感知出来 test1 的计算依赖于 searchName，这样当我们在输入框中输入的时候{{test1}}的内容就会跟着变化。而如果把 test1 的方法实现改为 return “hell0”+new Date()则不会重新调用刷新。

案例，用 computed 实现实时班级搜索和排序：

```
<input type="text" v-model="searchName" />
```

```
<ul>
```

```
  <li v-for="person in filterPersons">Id:{{person.id}} name:{{person.name}} age:{{person.age}}</li>
```

```
</ul>
```

computed 中增加 filterPersons 函数：

```
filterPersons: function () {
```

```
  var searchName = this.searchName;
```

```
  return this.persons
```

```
    .sort(function (p1, p2) { return p1.age - p2.age; })
```

```
    .filter(function(p){
```

```
      if(searchName)
```

```
      {
```

```
        return p.name.indexOf(searchName) >= 0;
```

```
      }
```

```
      else
```

```
      {
```

```
        return true;
```

```
      }
```

```
    });
```

```
}
```

外界如何调用 Vue 的方法或者数据？尽量不要直接操作数据，而通过定义的函数来操作。

用了 VueJS 之后就不需要 JQuery 了，只有一种情况可能用 JQuery：VueJS 没有内置 Ajax 库，你自己自由选择 ajax 库，当然也可以用 jquery 的 ajax 库。

窗口通讯

有时候需要在窗口之间进行通讯。

对于使用 window.open 打开的窗口，可以直接通过 open 的返回值操作窗口，子窗口通过 opener 拿到父窗口。

对于支持 html5 的浏览器，两个窗口之间可以通过 postMessage 来发送消息，第一个参数传递数据，可以是任何 json 对象，第二个参数代表域，一般传“*”。在窗口中监听 message 消息

```
window.addEventListener("message",function(event){//必须要是" message"
    //event.data 是 postMessage 第一个参数的值
});
```

Main.html 文件

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title></title>
</head>
<body>
    <button id="btn1">打开窗口</button>
    <button id="btn2">调用子窗口方法</button>
    <script type="text/javascript">
        var newWin;

        var btn1 = document.getElementById("btn1");
        btn1.onclick = function () {
            newWin = window.open("Child.html","_blank");
        };
        var btn2 = document.getElementById("btn2");
        btn2.onclick = function () {
            newWin.postMessage({ name:"rupeng",id:9},'*');
        };

        window.addEventListener("message",function(event){
            alert("父窗口收到消息， msgName="+event.data.msgName+",age="+event.data.age);
        });
    </script>
</body>
</html>
```

Child.html

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title></title>
</head>
<body>
    <button id="btn1">调用父窗口方法</button>
    <script>
        var btn1 = document.getElementById("btn1");
        btn1.onclick = function () {
            opener.postMessage({ msgName: "test",age:666},'*');
        };
    </script>
</body>
</html>
```

```
};  
window.addEventListener("message",function(event){  
    alert("子窗口收到消息，name="+event.data.name+",id="+event.data.id);  
});  
</script>  
</body>  
</html>
```

注意：如果父窗口刷新了，则就不能控制子窗口了。

父窗口如何得知子窗口被关闭了？没有事件，可以读取窗口的 `closed` 属性。

Electron

现在主要的系统都是以网站或者 App 的形式出现了。但是还是有少数一些系统需要是本地的桌面程序形式运行。本地程序的优点：可以进行系统级的操作、读写本地文件、操作本地硬件资源、可以操作系统托盘图标、可以离线操作等。典型的本地程序：微信 PC 版、有道云笔记、微信小程序开发工具、VSCode。

开发本地桌面程序的传统技术有 C++(MFC/QT/GTK+)、Delphi、C#(WinForm/WPF)等。C++ 开发难度太高；Delphi 已经落伍；C# 非微软体系的人天然拒绝。现在很多本地桌面程序流行用 JavaScript+chromium 浏览器内核写，其实就是用 html+css+JavaScript 写界面运行在浏览器中，和操作系统底层的操作也是使用 JS 来编写。好处是 js 大家都会，而且开发的难度低、容易跨平台。

完全自己基于 chromium 去写的话开发难度太大，现在有很多封装好的框架 nw.js、electron 等，最流行的就是 electron。

一、electron 简单环境搭建

可以从头搭建 electron 开发环境，但是比较麻烦，可以用别人搭建好的“脚手架”项目。这里推荐用 electron-boilerplate

1、安装 nodejs

2、在文件夹下执行

```
a) git clone https://github.com/szwacz/electron-boilerplate.git  
b) cd electron-boilerplate  
c) npm install //非常慢  
d) npm start 启动
```

3、如果上一步执行出错，可能是网络原因，需要 <https://segmentfault.com/q/1010000007594059>

a) 运行 npm config set registry <https://registry.npm.taobao.org>

b) 运行 npm 查看 .npmrc 文件路径

c) 以管理员身份运行记事本，打开 .npmrc（如果不存在则新建），写入：如下内容

```
registry=https://registry.npm.taobao.org  
sass_binary_site=https://npm.taobao.org/mirrors/node-sass/  
phantomjs_cdnurl=http://npm.taobao.org/mirrors/phantomjs
```

ELECTRON_MIRROR=<http://npm.taobao.org/mirrors/electron/>

- 4、package.json 文件是项目的配置文件，可以配置项目名称、作者信息、构建配置等等；main 设定的是入口的 js 文件。src 中是“源代码”，app 中是“编译后代码”，实际运行的是 app 中的代码，这个后续会详细讲。
- 5、解析项目的主要结构、主要代码。
- 6、打包成 exe 的方法：npm run release 不翻墙也能生成 exe（在 dist 文件夹下），不过会报错。把 dist 下的东西打包放到其他人的电脑上就可以运行，不需要对方安装 nodejs 等东西。自己用 InstallShield、InnoSetup 之类的做安装包即可，当然大公司都是自己开发安装程序。
- 7、如果使用 electron-boilerplate，那么可以运行“npm run release”来生成 exe，并且生成安装包。但是生成的是 x64 的。如何生成 32 位的呢（64 位系统也可以运行 32 位程序）？改 package.json，把 scripts 下的“release”改成“build --ia32 --win”
- 8、按照 electron-boilerplate 的项目结构，src 是 js、less 等的源文件夹，会生成到 app 文件夹下（做新 js 语法编译成低版本语法、less 翻译成 css、typescript 翻译成 js 等工作），app 文件夹下的 js、css 不应该被编辑，但是 html、图片等还是放到 app 文件夹下。

我感觉这样不好，我认为 src 文件夹就应该是源代码，app 文件夹应该全部都是生成的。开发者只修改 src 中的源代码，app 中是编译后的 js。所以我改造如下，修改 tasks/build_app.js，在 gulp.task('less'这一段之前加入：

```
gulp.task('copy', () => {  
  return Promise.all([  
    gulp.src(srcDir.path('*.*html')).pipe(gulp.dest(destDir.path("./")))  
  ]);  
});
```

这表示定义拷贝任务，把 src 目录下的*.html 拷贝到 app 文件夹下

在 watch('src/**/*.less'这段之前加入：

```
watch('src/**/*.html', batch((events, done) => {  
  gulp.start('copy', beepOnError(done));  
}));
```

这表示监视*.html 的变化，当变化的时候执行 copy 任务

在 gulp.task('build', ['bundle', 'less', 'environment']); 中加入 'copy' 任务，变成：

```
gulp.task('build', ['bundle', 'copy', 'less', 'environment']);
```

这表示脚本运行起来之后立即执行一次 copy 等任务（也就是 npm start 之后）。这样修改代码后立即就可以起作用了，不用重启程序。

二、electron 渲染进程和主进程

chromium 是多进程的浏览器，每个网页的界面的显示是一个进程，叫渲染进程(render)，本地操作是主进程，和界面相关的代码(alert 等)写到渲染进程中，本地操作代码写到主进程中。

主进程有一个，每个浏览器窗口都有一个渲染进程。

主进程和渲染进程之间使用 ipc 进行通讯。

background.js 中的代码运行在主进程；html 中的 js 以及 html 引用的 js 运行在渲染进程。

console.log()在渲染进程中执行是显示在 DevTools 中；在主进程中执行会显示在控制台中；setInterval()、setTimeout()、parseInt、math 对象等和 Dom 无关的对象都可以在主进程中

使用。

1) 渲染进程调用主进程:

主进程中:

```
import {ipcMain} from 'electron';
ipcMain.on('rendermsg',(event,arg)=>{
    console.log("rendermsg"+arg);
});
```

渲染进程的 html 中:

```
const {ipcRenderer} = require('electron');
btn1.onclick=function(){
    ipcRenderer.send("rendermsg",{name:'rupeng',id:3});
};
```

send 是异步调用的。如果在渲染进程中使用 sendSync 进行同步调用, 那么通过 sendSync 进行同步调用的返回值获得在主进程中通过 event.returnValue 设置的的返回值。尽量避免用 sendSync。

主进程:

```
ipcMain.on('rendermsg',(event,arg)=>{
    console.log("rendermsg"+arg);
    event.returnValue=666;
});
```

渲染进程:

```
btn1.onclick=function(){
    var v = ipcRenderer.sendSync("rendermsg",{name:'rupeng',id:3});
    alert(v);
};
```

2) 主进程调用渲染进程:

首先要拿到要操作的窗口对象, 或者 BrowserWindow.getAllWindows() 获取所有窗口, 再遍历查找要用的窗口。

主进程中:

```
setInterval(function(){
    mainWindow.send("hello",{name:'rupeng'});
},3000);
```

渲染进程中:

```
ipcRenderer.on("hello",function(event,arg){
    alert(arg);
});
```

三、主进程模块

以前 javascript 只能写浏览器中的程序, nodejs 出现让我们可以像 Java、C# 一样编写服务器程序、编写本地程序, 运行效率非常高。使用 nodejs 就是服务器端、客户端都用 JavaScript 编写。

electron 基于 nodejs 的, 因此在主进程中可以使用几乎所有的 nodejs 的模块, nodejs 中

的模块涵盖文件访问、数据库访问、网络操作、操作系统底层、串口通讯、win32 等所有的操作，electron 中也有特有的模块。下面只是告诉大家 nodejs 可以做什么，具体详细用法需要去深入研究 nodejs

下面的写法在两个进程中都可以调用（需要渲染进程启用 nodeIntegration，默认是启用的）

1、文件系统，：

```
var fs = require('fs');
fs.readFile("d:/1.txt","utf8",function(err,data){
    alert(data);
});
```

2、shell

用操作系统默认的编辑器打开文件：

```
const {shell} = require('electron');
shell.openItem("d:/1.txt");
```

用默认浏览器打开网址：

```
shell.openExternal(url)
```

资源管理器中定位文件：

```
shell.showItemInFolder(fullPath)
```

更多：<https://weishuai.gitbooks.io/electron-/content/api/shell.html>

3、托盘图标

在 app 文件夹下放一个 icon.ico 图标，然后再主进程中

```
const electron = require('electron');
const Tray = electron.Tray;

var appIcon = null;
app.on('ready', function(){
    var appIconPath = path.join(__dirname,'icon.ico');
    appIcon = new Tray(appIconPath);
    var contextMenu = Menu.buildFromTemplate([
        { label: 'Item1', type: 'radio',click:function(){console.log('Item1 clicked')}}},
        { label: 'Item2', type: 'radio' },
        { label: 'Item3', type: 'radio', checked: true },
        { label: 'Item4', type: 'radio' }
    ]);
    appIcon.setToolTip('This is my application.');
```

<https://weishuai.gitbooks.io/electron-/content/api/tray.html>

图标闪烁：

搞一个空白的 ico 图片 empty.ico

```
const electron = require('electron');
const Tray = electron.Tray;

const applIconPath = path.join(__dirname, 'icon.ico');//图标路径，app\images 下面
const emptyIconPath = path.join(__dirname, 'empty.ico');//空白图标，托盘图标闪烁使用
var tray = null;
app.on('ready', function(){
    var applIconPath = path.join(__dirname, 'icon.ico');
    tray = new Tray(applIconPath);
    var contextMenu = Menu.buildFromTemplate([
        { label: 'Item1', type: 'radio', click: startTrayIconBlink },
        { label: 'Item2', type: 'radio', click: stopTrayIconBlink },
        { label: 'Item3', type: 'radio', checked: true },
        { label: 'Item4', type: 'radio' }
    ]);
    tray.setToolTip('This is my application. ');
    tray.setContextMenu(contextMenu);
});

function setTrayIcon(iconPath)
{
    try
    {
        tray.setImage(iconPath);
    }
    catch(err)//这是 electron 的 bug，偶尔调用 setImage 会报异常，目前没有解决方案
    {
        console.error("设置图片出错,"+err.description);
    }
}

//开始托盘图标闪烁
function startTrayIconBlink()
{
    if(tray.intervalId)//如果设置了 tray.intervalId，说明还有在闪动的图标，不要多次执行闪
    动逻辑，否则会越闪越快
    {
        return;
    }
    var i=0;
    tray.intervalId=setInterval(()=>>
    {
        i++;
    }, 1000);
}
```

```
if(i%2==0)//偶数次数显示程序图标，奇数次数显示空白图标
{
    setTrayIcon(appIconPath);
}
else
{
    setTrayIcon(emptyIconPath);
}
},500);
}
```

//停止托盘图标闪烁

```
function stopTrayIconBlink()
{
    if(tray.intervalId)
    {
        clearInterval(tray.intervalId);
        tray.intervalId=null;//清空 tray.intervalId
        setTrayIcon(appIconPath);//确保恢复到正常图标
    }
}
```

四、 集成的问题

因为 electron 会把渲染进程的浏览器默认配置为 nodeIntegration，这样就默认引入了 module、require 变量，会和 jquery、requirejs 等冲突。有很多种解决方法，推荐的解决方法：

把下面的 js 代码放到单独的 js 文件中，然后放到所有框架加载之前

```
if (typeof (require) !== "undefined")
{
    window.nodeRequire = require;
    delete window.require;
    delete window.exports;
    delete window.module;
}
```

然后通过 `const electron = nodeRequire("electron");` 这种方式进行 require，也就是给 require 取一个别名，然后再把 require 等定义销毁掉。<https://zhuanlan.zhihu.com/p/21440362>

electron 中文文档 <https://www.gitbook.com/book/weishuai/electron-/details>

讲解项目

- 1、演示功能：直接 electron 方式运行；
- 2、讲解 electron 客户端部分代码；
- 3、讲解服务器端部分代码：
 - a) signalR 默认是不处理属性首字母小写的，因此 C# 中的大写属性传递到 js 中很难看。如何采用驼峰命名法，Filters 下 SignalRContractResolver.cs，然后在 Startup 中配置。
参考：<https://stackoverflow.com/questions/30005575/signalr-use-camel-case>
 - b) MVC 中配置 JsonNetActionFilter 设置 mvc 的首字母小写；
 - c) Filters 中 SignalRHubPipelineModule 记录 SignalR 未处理异常；
 - d) Helpers 下：UserCenterApi 是用来访问 WebAPI 课程最后项目中开发的 UserCenter 服务器的，读取配置文件，引用了 WebAPI 中开发的 UserCenter .Net SDK，改了一些 bug，以我的为准。把 UserCenter 配置到 IIS 中执行。
 - e) Helpers 下：登录、获取组信息、获取组成员信息通过 UserCenter 来进行，但是 UserCenter 只是提供了，等的接口，所以需要 IM 服务器提供登录状态的保存。项目中使用 JWT 来把登录信息放到 Cookie 中，详细看 JWTHelper
 - f) Helpers 下 RedisHelper 把 Redis 的创建封装了
 - g) HomeController: Login 把登录用户信息加密成 JWT Token 保存到 Cookie 中；GroupMain 是显示班级聊天室主页，当前用户如果不属于这一组则不能打开（安全第一）；LoadGroups 是供软件主页 ajax 加载用户所属组用的；LoadGroupUsers 是用来在班级聊天室主页显示班级成员用的，注意数据安全，而且不要泄露其他用户的手机号，ChatHub 中会把用户的在线状态保存到 Redis 中，所以 LoadGroupUsers 还会从 Redis 中加载每个用户的在线状态；LoadHistoryMessages 是在班级聊天室主页加载聊天历史消息用的，因为 SignalR 只有在连接的时候才能得到消息推送，因此每次发消息的时候也存到 Redis 中一份，以班级 Id 为 key，聊天消息为 value 的 list 中，这样聊天室打开的时候先到 redis 中取 N 条历史聊天消息。
 - h) ChatHub 是聊天的核心 Hub。SendGroupMessage 用来向某个组发送消息用，还是要从 JWT 中取出当前登录用户信息，判断当前用户是否在这个组中，同时要把聊天记录保存到 Redis 中；OnConnected 客户端连接的时候，把当前用户加入所属的组用，用组在数据库中的 Id 作为 SignalR 的 groupName，并且更新在线状态；OnDisconnected 中与 OnConnected 相反；
 - i) Vue 中没有提供 Ajax 库，可以选择任意 Ajax 库，比如 JQuery 等。推荐大家用 axios

```

    axios.post('/Home/LoadHistoryMessages', {
      groupId: this.groupId
    })
    .then(function (resp) {
      var data = resp.data;
      if (data.status !== "ok") {
        alert(data.msg);
        return;
      }
      for (var i = 0; i < data.data.length; i++) {
        var msg = data.data[i];
        messages.push(msg);
      }
    })
    .catch(function (error) {
      alert("获取历史消息错误, " + error);
    });
  
```

请求体数据

服务器成功返回
如果返回是application/json
resp.data就是反序列化js对象

发生错误

- j) 为了界面好看点，界面使用的是腾讯的 WeUI
- k) Main.cshtml: 学习 SignalR 的时候讲过，最好只有一个页面连接 Hub，其他页面和这个页面通讯来完成交互。因此把所有的和 Hub 的操作放到 Main.cshtml 中。
{{g.unReadCount}}显示的未读消息（如果组聊天窗口没有打开，来了消息则显示未读消息数量，点击之后打开窗口，然后未读消息归零）
- l) Main.cshtml: 每个打开的组聊天窗口对象放到 groupWindows 中，window.open 打开组聊天窗口的时候，把窗口对象放到 groupWindows 中；findWindowByGroupId 函数用来根据组 id 找到对应的群组窗口；
- m) Main.cshtml: VueJS 中如何在窗口加载完成后执行代码？放到 mounted 中；groupClick 点击群组名字的时候打开窗口或者激活窗口；
- n) Main.cshtml: \$.connection.chatHub.client.onMessage 消息来了之后，如果对应的班级窗口打开了，则通过 postMessage 告诉窗口添加聊天消息；如果窗口没有打开，则更新未读消息，并且托盘图标闪烁、播放声音提醒（使用 html5 的 audio 标签）。群组窗口要发送消息的时候其实是给 Main 发送 sendGroupMessage 消息，由 Main 来发送消息；
- o) GroupMain.cshtml: 当消息添加到消息列表中滚动到底部，要使用 watch 监控 getMessages()数据变化，然后 chatbox.scrollTop = chatbox.scrollHeight;滚动到底部，但是需要注意在 Vue 的 function 中要直接操作 Dom，需要放到\$nextTick()中。\$nextTick 是在下次 DOM 更新循环结束之后执行延迟回调，在修改数据之后使用 \$nextTick，则可以在回调中获取更新后的 DOM。