

一、安装和使用饿了么组件

1、安装

```
cnpm i element-ui -S
```

2、在main.js 引入组件

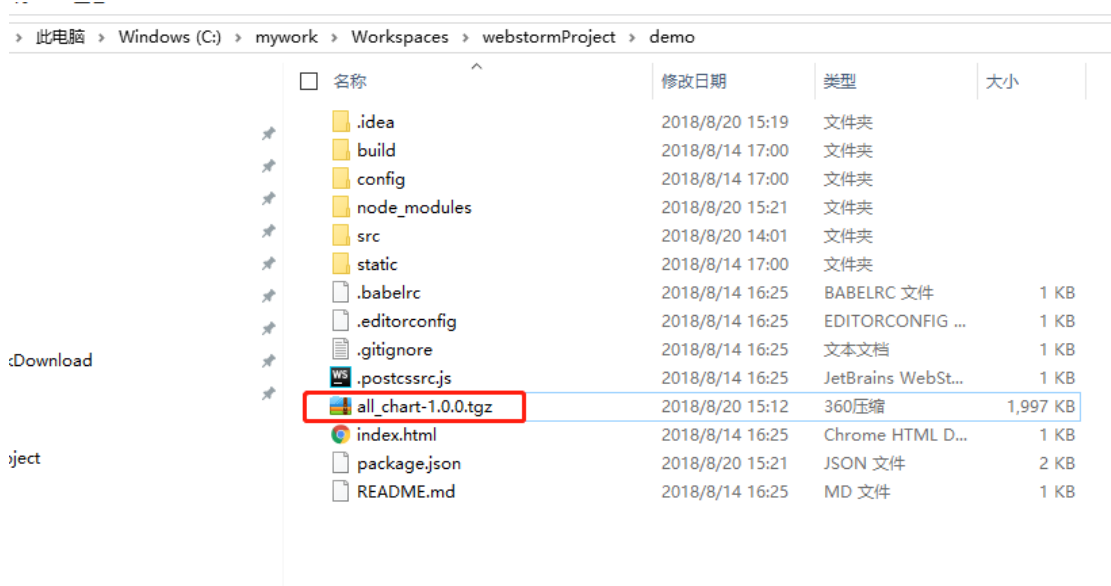
```
import Element from 'element-ui'
```

3、在component目录创建element table组件的table.vue（官网有样例代码，直接复制即可） table.vue组件包括html, js, css（template只能有一个根页签）

官网地址：<http://element-cn.eleme.io/#/zh-CN/component/form>

二、引入外部的包

1、将包copy到根目录下



2、运行命令安装包

```
cnpm install ./all_chart-1.0.0.tgz --save
```

3、在vue工程下面的main.js引入包

```
import all_chart from 'all_chart'
```

```
Vue.use(all_chart);
```

4、在需要使用的页面上进行使用。例如：使用饼型图

```
scale4:{ "position":["item","count","percent"] },
```

```
pie_data:[{ item: '事例一', count: 40, percent: 0.4 }, { item: '事例二', count: 21, percent: 0.21 }, { item: '事例三', count: 17, percent: 0.17 }, { item: '事例四', count: 13, percent: 0.13 }, { item: '事例五', count: 9, percent: 0.09 }]
```

三、URL带的#该怎么去掉

vue-router 默认 hash 模式 —— 使用 URL 的 hash 来模拟一个完整的 URL，于是当 URL 改变时，页面不会重新加载。

如果不要很丑的 hash，我们可以用路由的 history 模式，这种模式充分利用 history.pushState API 来完成 URL 跳转而无须重新加载页面。

```
const router = new VueRouter({
  mode: 'history',
  routes: [...]
})
```

当你使用 history 模式时，URL 就像正常的 url，例如 <http://yoursite.com/user/id>，也好看！

不过这种模式要玩好，还需要后台配置支持。因为我们的应用是个单页客户端应用，如果后台没有正确的配置，当用户在浏览器直接访问 <http://oursite.com/user/id> 就会返回 404，这就不好看了。

所以呢，你要在服务端增加一个覆盖所有情况的候选资源：如果 URL 匹配不到任何静态资源，则应该返回同一个 index.html 页面，这个页面就是你 app 依赖的页面。

来源: <https://router.vuejs.org/zh/guide/essentials/history-mode.html#%E5%90%8E%E7%AB%AF%E9%85%8D%E7%BD%AE%E4%BE%8B%E5%AD%90>

四、去掉Vue的图标

打开App.vue,删除img

```
<template>
  <div id="app">
    <!-- 
    <router-view/>
  </div>
</template>
```

五、绑定数据

页面:

数据:

编写调用方法的接口：

```
}
```

页面创建时调用方法，立马加载数据进来：

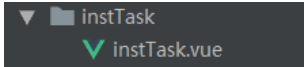
```
created:function() {  
  this.statistic();  
},
```

data()\ created \ methods 三个方法在同等级下。

六、跳转到页面上

打开router-》index.js

新建instTask.vue



添加路径

```
export default new Router({  
  routes: [  
    {  
      path: '/instTask',  
      name: 'instTask',  
      component: instTask  
    }  
  ]  
})
```

引入组件

```
import instTask from '@components/instTask/instTask'
```

这样输入路径后就跳到对应的组件页面中

七、页面之间的跳转

添加一个按钮，点击按钮后跳转页面

```
<a class="bgbtn" @click="getInstTaskById(item.id,item.name)" >  
    
</a>
```

方法如下，根据name来跳转到对应的页面：

```
getInstTaskById (taskgroupId,taskgroupName) {  
  this.$router.push({  
    name: 'instTask',  
    params: {  
      taskgroupId:taskgroupId,  
      taskgroupName: taskgroupName  
    }  
  })  
},
```

在data中接收到参数

```
export default {  
  name: "InstTask",
```

```
data() {
  return {
    taskgroupId: this.$route.params.taskgroupId,
    taskgroupName: this.$route.params.taskgroupName,
  }
},
```

初始化页面完成后调用该方法

```
created: function () {
  this.findByInstTaskGroupId();
}
```

方法的实现，根据传过来的参数，调用方法，返回数据到页面中：

```
methods: {
  findByInstTaskGroupId() {
    let data_new = {};
    data_new["taskgroupId"] = this.taskgroupId;
    let url = '/jobInstTask/findByInstTaskGroupId';
    let self = this;
    this.$axios({
      method: 'post',
      url: url,
      data: data_new,
      headers: {},
    }).then(function (response) {
      let data = response.data;
      let itemNew = [];
      for (let i = 0; i < data.length; i++) {
        itemNew.push(data[i]);
      }
      self.items = itemNew;
      console.log(self.items);
    }).catch(function (error) {
    });
  },
}
```

完成。

八、axios的配置方法

在main.js文件中引入axios

```
import axios from 'axios'
```

添加方法的前缀

```
var url_TEST = 'http://127.0.0.1:9301/';
var url_PROD = 'http://127.0.0.1:9301/';

var instance1 = axios.create({
  baseURL: url_TEST,
```

```
headers: {  
  'Content-Type': 'application/json;charset=UTF-8'  
}  
});
```

引入:

```
Vue.prototype.$axios=instance1;
```

九、前端Long类型字段丢失精度的问题

现象

项目中用到了唯一ID生成器.生成出的ID是long型的(比如说4616189619433466044).

通过某个rest接口中返回json数据后,发现浏览器解析完变成了4616189619433466000.

原因

大致描述:java中的long能表示的范围比js中number大,也就意味着部分数值在js中存不下(变成不准确的值).

详情参考这里<http://stackoverflow.com/questions/17320706/javascript-long-integer>

rest接口返回的json字符串中,数值还是对的.当js对json进行解析并转成js object的时候,出现了问题.

解决方法

将id字段序列化为json时,转换为字符串类型,前端传输到后端,反序列化时,再重新转换为Long。

在dto所在项目中,新建一个helper包(名字自定义,也可以放现有包里)。PS:为什么要建到dto项目中? 因为,这个包最后可能会给其他组使用,这样以来,所有的处理规则逻辑都是统一的,方便对接。

在包里添加类LongJsonSerializer,代码如下:

```
1  /**  
2   * Long 类型字段序列化时转为字符串,避免js丢失精度  
3   *  
4   */  
5  public class LongJsonSerializer extends JsonSerializer<Long> {  
6      @Override  
7      public void serialize(Long value, JsonGenerator jsonGenerator, SerializerProvider serializerProvider) throws  
8      IOException, JsonProcessingException {  
9          String text = (value == null ? null : String.valueOf(value));  
10         if (text != null) {  
11             jsonGenerator.writeString(text);  
12         }  
13     }
```

然后在包里再添加类LongJsonDeserializer,代码如下:

```
1  /**  
2   * 将字符串转为Long  
3   *  
4   */  
5  public class LongJsonDeserializer extends JsonDeserializer<Long> {  
6      private static final Logger logger = LoggerFactory.getLogger(LongJsonDeserializer.class);  
7      @Override  
8      public Long deserialize(JsonParser jsonParser, DeserializationContext deserializationContext) throws
```

```
9 IOException, JsonProcessingException {
10     String value = jsonParser.getText();
11     try {
12         return value == null ? null : Long.parseLong(value);
13     } catch (NumberFormatException e) {
14         logger.error("解析长整形错误", e);
15         return null;
16     }
17 }
18 }
```

好了，接下来是使用这两个类。

在需要处理的id字段上，加上注解。比如如下代码：

```
1 /**
2  * id
3  */
4 @JsonSerialize(using = LongJsonSerializer.class)
5 @JsonDeserialize(using = LongJsonDeserializer.class)
6 private Long id;
```

参考：<https://www.cnblogs.com/vgg/p/7475140.html>