Promise、Async/Await

1. Promise是为了解决什么问题?

解决了回调地狱Callback Hell的问题。

回调地狱的问题并不只是在于缩进太多（如下图），至少在阅读如下代码的时候不会有什么障碍。

```
function hell(win) {
 // for listener purpose
 return function() {
   loadLink(win, REMOTE_SRC+'/assets/css/style.css', function() {
     loadLink(win, REMOTE_SRC+'/lib/async.js', function() {
       loadLink(win, REMOTE_SRC+'/lib/easyXDM.js', function() {
         loadLink(win, REMOTE_SRC+'/lib/json2.js', function() {
           loadLink(win, REMOTE_SRC+'/lib/underscode.min.js', function() {
             loadLink(win, REMOTE_SRC+'/lib/backbone.min.js', function() {
               loadLink(win, REMOTE_SRC+'/dev/base_dev.js', function() {
                 loadLink(win, REMOTE_SRC+'/assets/js/deps.js', function() {
                   loadLink(win, REMOTE_SRC+'/src/' + win.loader_path + '/loader.js', function() {
                     async.eachSeries(SCRIPTS, function(src, callback) {
                       loadScript(win, BASE_URL+src, callback);
                     });
                   });
                 });
               });
             });
           });
         });
       });
     });
   });
 };
}
```

1. 难于理解

真正的问题在于逻辑难于理解，如下图代码中，我们假定形如doSomething的调用又涉及到一步调用，那么阅读的人可能会需要把调用顺序记在脑袋里才行。

```
listen( "click", function handler(evt){
  doSomething1();
  doSomething2();
  doSomething3();
  doSomething4();
  setTimeout( function request(){
    doSomething8();
    doSomething9();
    doSomething10();
    ajax( "http:// some. url. 1", function response( text){
      if (text == "hello") {
        handler();
      } else if (text == "world") {
        request();
      }
    });
    doSomething11();
    doSomething12();
    doSomething13();
  }, 500);
  doSomething5();
  doSomething6();
  doSomething7();
});
```

相比较起来，如下的一个使用 Promise包装的例子就很简洁，其关键点在于Promise的构造函数中，只负责成功/失败的通知，而后续的操作放在了then中

```
const getJSON = function(url) {
  const promise = new Promise(function(resolve, reject){
    const handler = function() {
      if (this.readyState !== 4) {
        return;
      }
      if (this.status === 200) {
        resolve(this.response);
      } else {
        reject(new Error(this.statusText));
      }
    };
    const client = new XMLHttpRequest();
    client.open("GET", url);
    client.onreadystatechange = handler;
    client.responseType = "json";
    client.setRequestHeader("Accept", "application/json");
    client.send();

  });

  return promise;
};

getJSON("/posts.json").then(function(json) {
  console.log('Contents: ' + json);
}, function(error) {
  console.error('出错了', error);
});
```

2. 信任问题

经过Promise流程的调用，将同步调用放在then中就不会出现同步调用意外的早于异步调用的情况；而且Promise的结果不能被篡改，多次调用的回调结果都能保持一致。

2. 如何模拟实现Promise

```
let status = 'pending';
class Promise {
  constructor(func){
    func(this._resolve.bind(this), this._reject.bind(this))
  }
  _resolve() {
    status = 'fullfilled';
  }
  _reject() {
    status = 'rejected';
  }
  then(succCallback, failCallback) {
    if(status == 'pending') {
      this.succCbList.push(succCallback);
      this.failCbList.push(failCallback);
    } else if(status== 'fullfilled'){
      succCbList.forEach(f => {f();})
    } else {
      failCallback.forEach(f => {f();})
    }
  }
}
```

3. 实际的应用场景

```
service.interceptors.response.use(
  response => {
    const res = response.data;
    if (res.status == 10101 || res.status == 401) {
      G.U.clearCookie()
      G.U.removeCookie('token', document.domain)
      G.U.removeCookie('token', G.U.getTopDomain())
      Message.error({
        message: res.message || '未授权用户，即将跳转到登录界面',
        duration: 2 * 1000
      })
      setTimeout(() => {
        window.location.href = '/login.html'
      }, 2000)
    } else if (res.status == 404) {
      Message.error({
        message: res.message || '请求找不到',
        duration: 5 * 1000
      })
    } else if (res.status == 500 || res.status == 503) {
      Message.error({
        message: res.message || '服务器错误',
        duration: 5 * 1000
      })
    } else if (G.U.isBlob(res)) {
      // do nothing
    } else if (res.status != 0) {
      Message.error({
        message: res.message || res.errorMessage || '未知异常',
        duration: 5 * 1000
      })
    }
    return response.data
  },
  error => {
    Message.error({
      message: error.message,
      duration: 5 * 1000
    })
  }
)
```

```
service.interceptors.response.use(
  response => {
    const res = response.data;
    let message;

    return new Promise((resolve, reject) => {
      if (res.status == 10101 || res.status == 401) {
        G.U.clearCookie()
        G.U.removeCookie('token', document.domain)
        G.U.removeCookie('token', G.U.getTopDomain())
        setTimeout(() => { window.location.href = '/login.html' }, 2000)
        message = res.message || '未授权用户，即将跳转到登录界面';
      } else if (res.status == 404) {
        message = res.message || '请求找不到';
      } else if (res.status == 500 || res.status == 503) {
        message = res.message || '服务器错误';
      } else if (G.U.isBlob(res)) {
        // do nothing
      } else if (res.status != 0) {
        message = res.message || res.errorMessage || '未知异常'
      }

      if(message != undefined) {
        reject(message);
        Message.error({ message, duration: 3 * 1000 })
      } else {
        resolve(response.data);
      }
    });
  },
  error => {
    Message.error({ message: error.message, duration: 5 * 1000 })
    return Promise.reject(error.message)
  }
)
```

返回的Promise对象，可以让调用的位置，按照Promise的缘分书写then/catch。下面是来自components/activity/view.vue使用Promise改写后的代码，后者看看起来更简洁：

```
methods: {
  queryAuditStatus(id){
    G.R.Common.getAuditProgress('ACTIVITY_AUDIT', id).then(resp => {
      if(resp.status == 0) {
        this.processData = resp.data;
      }
    })
  }
},
mounted() {
  G.R.Product.getActivityDetail(params.id).then(resp => {
    if(resp.status == 0 && resp.data) {
      this.product = Activity.parse(resp.data);
      this.queryAuditStatus(params.id);
    } else{
      this.$message({
        type:'info',
        message:resp.message
      })
    }
    this.loading = false;
  })
}
```

```
let id = G.U.getParam('id', this);

this.loadingCount = 2;

G.R.Product.getActivityDetail(id).then(resp => {
  this.product = Activity.parse(resp.data || {});
  return G.R.Common.getAuditProgress('ACTIVITY_AUDIT', id);
}).then(resp => {
  this.processData = resp.data;
}).finally(() => {
  this.loadingCount --;
});
```

4.Async/Await的含义和基本用法

1. Generator

可以把Generator函数看成是一个状态机，封装了多个内部状态，执行 Generator 函数会返回一个遍历器对象，代表 Generator 函数的内部指针。虽然Generator函数是一个普通函数，但是有两个特征：（1）function关键字与函数名之间有一个星号；（2）函数体内部使用yield表达式，定义不同的内部状态。

```
function* helloWorldGenerator() {
  yield 'hello';
  yield 'world';
  return 'ending';
}

var hw = helloWorldGenerator();

hw.next()
// { value: 'hello', done: false }

hw.next()
// { value: 'world', done: false }

hw.next()
// { value: 'ending', done: true }

hw.next()
// { value: undefined, done: true }
```

1. Async/Await

async就是Generator函数的语法糖。如下是使用Generator函数依次读取两个文件的代码：

```
const fs = require('fs');

const readFile = function (fileName) {
  return new Promise(function (resolve, reject) {
    fs.readFile(fileName, function(error, data) {
      if (error) return reject(error);
      resolve(data);
    });
  });
};

const gen = function* () {
  const f1 = yield readFile('/etc/fstab');
  const f2 = yield readFile('/etc/shells');
  console.log(f1.toString());
  console.log(f2.toString());
};
```

使用async函数改写后如下，语法上看只是简单地把 * 换成 async，yield 替换成 await 而已。

```
const asyncReadFile = async function () {
  const f1 = await readFile('/etc/fstab');
  const f2 = await readFile('/etc/shells');
  console.log(f1.toString());
  console.log(f2.toString());
};
```

但async对Generator函数的改进体现在如下四点：

- Generator必须依靠内置执行器（co模块）通过next语法执行，而async函数自带执行器执行起来像普通函数
- 更好的语义，比起*和yield，使用async和await语义更清楚
- 更广的适用性，yield命令后面只能是Trunk函数和Promise对象，而async的await后面可以是Promisee对象和原始值
- async函数的返回值是Promise对象，比起Generator函数的返回值是Iteretor翻遍，可以使用then方法指定下一步操作

5. Async/Await的实际应用

未经过改写的代码

```
methods: {
  handlePreview(file) {
    this.dialogImageUrl = file.url;
    this.dialogVisible = true;
  },
  queryAuditStatus(id){
    G.R.Common.getAuditProgress('ACTIVITY_AUDIT', id).then(resp => {
      if(resp.status == 0) {
        this.processData = resp.data;
      }
    })
  },
  lookProduct(id){
    this.$router.push({name:'prod_product_manage_view',params:{id:id}})
  }
},
mounted() {
  let params = this.$route.params || {};
  this.loading = true;
  G.R.Product.getActivityDetail(params.id).then(resp => {
    console.log(resp)
    if(resp.status == 0 && resp.data) {
      this.product = Activity.parse(resp.data);
      this.queryAuditStatus(params.id);
    }else{
      this.$message({
        type:'info',
        message:resp.message
      })
    }
    this.loading = false;
  })
  let id = {
    id:params.id
  }
  G.R.Product.getActivityOtherDispose(id).then(resp => {
    console.log(21321321312312)
    console.log(resp)
    if(resp.status == 0 && resp.data) {
      if(resp.data.length){
        this.activityOther = resp.data;
      }
    }
    this.loading = false;
  })
}
```

使用await该改写后的代码：

-（1）代码更少，可读性更好 -（2）在loading的控制上更直接

```
# /components/activity/view.vue

async mounted() {
  let params = this.$route.params || {};
  let id = params.id;
  let activityRes = await G.R.Product.getActivityDetail(id);
  let processRes = await G.R.Common.getAuditProgress('ACTIVITY_AUDIT', id);
  let otherRes = await G.R.Product.getActivityOtherDispose({id});

  this.loading = true;

  if(activityRes.ok && processRes.ok && otherRes.ok) {
    this.product = Activity.parse(activityRes.data);
    this.processData = processRes.data;
    if(otherRes.data && otherRes.data.length) {
      this.activityOther = otherRes.data;
    }
  }

  this.loading = false;
}
```

将三个请求改为并行缩短请求时间

```
# /components/activity/view.vue
async mounted() {
  ...
  let [activityRes, processRes, otherRes] = await Promise.all([
    G.R.Product.getActivityDetail(id),
    G.R.Common.getAuditProgress('ACTIVITY_AUDIT', id),
    G.R.Product.getActivityOtherDispose({id})
  ])
  ...
}
```