1. Promise

1.1. 什么是Promise

> Promise是异步编程 的一种解决方案

在开发中，在进行一些网络请求或进行一些异步相关的操作的话，一般情况下就不再是同步了

同步的话，会有很严重的问题，假如用户在进行某一项操作的时候，需要向服务器发送网络请求，而网络请求一般是比较耗时的，如果程序执行还是同步的话，代码执行到网络请求这一步就不会再往下执行，而是等到网路请求成功，服务器将用户请求的数据发送到用户服务器上才会执行下面的代码。

这种情况称为同步阻塞，阻塞发生时，不仅用户界面上不会显示任何内容，而且浏览器也不会响应用户的操作。

所以一般情况下不会这样做。通常情况下，当需要向服务器发送网络请求时，就开启一个异步任务，当网络请求的数据回来的时候，一般会在下面有一个回调的函数，这个回调函数就会拿到服务器返回来的数据，之后再使用这个数据就可以了

但是，当网络请求非常复杂时，就会出现**回调地狱**

1.2. 定时器的异步事件

我们用一个定时器来模拟异步事件：

- 假设下面的data是从网络上1秒后请求的数据
- console.log就是我们的处理方式

```
setTimeout(function () {
    let data = 'hello world'
    console.log(data);
}, 1000)
```

Promise就是对异步操作进行一些封装：

```
new Promise((resolve, reject) => {

    // 第一次网络请求的代码
    setTimeout(() => {
        resolve()
    }, 1000).then(() => {

        // 第一次拿到结果的处理代码
        console.log("Hello,Wro");
        console.log("Hello,Wro");
        console.log("Hello,Wro");
        console.log("Hello,Wro");
        console.log("Hello,Wro");

        return new Promise((resolve, reject) => {

            // 第二次网络请求的代码
            setTimeout(() => {
                resolve()
```

```
            }, 1000)
        })
    }).then(() => {

        // 第二次拿到结果的处理代码
        console.log('Hello,Vuejs');
        console.log('Hello,Vuejs');
        console.log('Hello,Vuejs');
        console.log('Hello,Vuejs');
        console.log('Hello,Vuejs');
        console.log('Hello,Vuejs');

        return new Promise((resolve, reject) => {

            // 第三次网络请求的代码
            setTimeout(() => {
                resolve()
            }, 1000)
        })
    }).then(() => {

        // 第三次拿到结果的处理代码
        console.log('Hello,Python');
        console.log('Hello,Python');
        console.log('Hello,Python');
        console.log('Hello,Python');
        console.log('Hello,Python');
        console.log('Hello,Python');
    })
})
```

什么情况下会用到Promise？一般情况下是有异步操作时，使用Promise对这个异步操作进行封装

new ->构造函数（1. 保存了一些状态信息，2. 执行传入的参数）
在执行传入的回调函数时，会传入两个参数，resolve,reject，这两个参数本事又是函数

Promise将网络请求的代码和处理的代码进行了分离

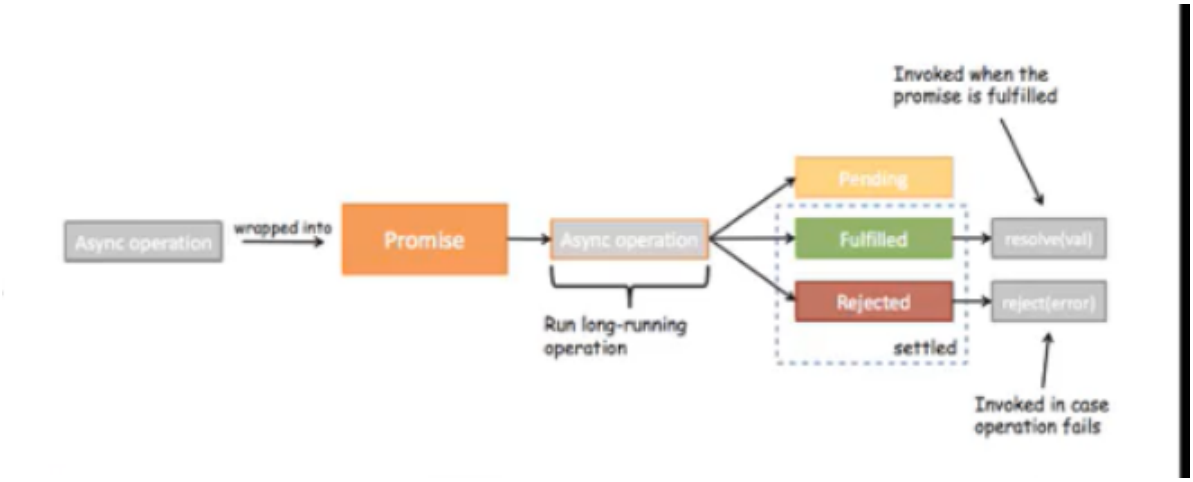
```
new Promise((resolve, reject) => {
        setTimeout((data) => {
            // 成功的时候调用resolve
            resolve(data)

            // 失败的时候调用reject
            reject('error message')
        }, 1000)
    }).then((data) => {
        处理的代码
        处理的代码
        处理的代码
        处理的代码
        处理的代码
        处理的代码
    }).catch((err)=>{
        console.log(err)
    })
```

1.3. Promise的三种状态

开发中有异步操作时，可以给异步操作包装一个Promise

异步操作之后有三种状态：

- Pending：等待状态，比如正在进行网络请求，或者定时器没有到时间
- fullfill:满足状态，当我们主动回调了resolve时，就处于该状态，并且会回调`.then()`
- reject：拒绝状态，当我们主动回调了reject时，就处于该状态，并且会回调`.catch()`



1.4. Promise另外一种写法

```
new Promise((resolve,reject)=>{
    setTimeout(()=>{
        // resolve('Hello, Vuejs')
        reject('error message')
    },1000)
}).then(data=>{
    console.log(data);
},err=>{
    console.log(err);
})
```

1.5. Promise链式调用

```
// 网络请求：aaa->自己处理（10行）
// 处理：aaa111->自己处理（10行）
// 处理：aaa111222->自己处理（10行）

// new Promise((resolve, reject) => {
//     setTimeout(() => {
//         resolve('aaa')
//     }, 1000)
// }).then(res=>{
//     // 1. 自己处理10行代码
//     console.log(res,'第一层的10行处理代码');

//     // 2. 对结果进行第1次的处理
//     return new Promise((resolve)=>{
//         resolve(res+'111')
//     })
// }).then(res=>{
```

```
//     console.log(res,'第二层10行处理代码');


//     return new Promise((resolve)=>{
//         resolve(res+'222')
//     })
// }).then(res=>{
//     console.log(res,'第三层10行处理代码');
// })


// 简写
// new Promise((resolve, reject) => {
//     setTimeout(() => {
//         resolve('aaa')
//     }, 1000)
// }).then(res=>{
//     // 1. 自己处理10行代码
//     console.log(res,'第一层的10行处理代码');

//     // 2. 对结果进行第1次的处理
//     return Promise.resolve(res+'111')

// }).then(res=>{
//     console.log(res,'第二层10行处理代码');


//     return  Promise.resolve(res+'222')

// }).then(res=>{
//     console.log(res,'第三层10行处理代码');
// })


// 更简介的版本：省略掉Promise.resolve()
// new Promise((resolve, reject) => {
//     setTimeout(() => {
//         resolve('aaa')
//     }, 1000)
// }).then(res => {
//     // 1. 自己处理10行代码
//     console.log(res, '第一层的10行处理代码');

//     // 2. 对结果进行第1次的处理
//     return res + '111'

// }).then(res => {
//     console.log(res, '第二层10行处理代码');


//     return res + '222'

// }).then(res => {
//     console.log(res, '第三层10行处理代码');
// })


// 包含reject
```

```
// new Promise((resolve, reject) => {
//     setTimeout(() => {
//         resolve('aaa')
//     }, 1000)
// }).then(res => {
//     // 1. 自己处理10行代码
//     console.log(res, '第一层的10行处理代码');

//     // 2. 对结果进行第1次的处理
//     return Promise.reject('error message')

// }).then(res => {
//     console.log(res, '第二层10行处理代码');


//     return res + '222'

// }).then(res => {
//     console.log(res, '第三层10行处理代码');
// }).catch(err=>{
//     console.log(err);
// })


// 包含reject的简写
new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve('aaa')
    }, 1000)
}).then(res => {
    // 1. 自己处理10行代码
    console.log(res, '第一层的10行处理代码');

    // 2. 对结果进行第1次的处理
    throw 'error message'

}).then(res => {
    console.log(res, '第二层10行处理代码');


    return res + '222'

}).then(res => {
    console.log(res, '第三层10行处理代码');
}).catch(err=>{
    console.log(err);
})
```

1.6. Promise.all()

如果在开发中遇到一个需求需要发送多次请求才可以完成，那么可以使用`Promise.all()`对他们进行包装

```
Promise.all([
    // new Promise((resolve, reject) => {
    //     $ajax({
    //         url: 'url1',
```

```
    //         success: function (data) {
    //             resolve(data)
    //         }
    //     })
    // }),
    // new Promise((resolve, reject) => {
    //   $ajax({
    //       url:'url2',
    //       success:function(data){
    //           resolve(data)
    //       }
    //   })
    // })

    new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve('result1')
        }, 1000)
    }),

    new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve('results2')
        }, 1000)
    })


]).then(results => {
    console.log(results);
})
```

2. Vuex

> Vuex是一个专门为Vue.js应用程序开发的**状态管理模式**

- 它采用集中式存储管理应用的所有组价的状态，并以响应的规则保证状态以一种可预测的方式发生变化
- Vuex也集成到vue官方调试工具`devtools extension`，提供了诸如零配置的`time-travel`调试，状态快照导入导出等高级调试功能

2.1. 状态管理是什么?

状态管理可以简单地将其看成把需要多个组件共享的变量全部存储在一个对象里面
然后将这个对象放置在顶层的Vue实例中，让其他组件可以使用
多个组件就可以共享这个对象中所有变量属性了

但是，上述对象自己也可以封装，Vue官方为什么要专门出一个插件Vuex?
自己封装的对象不能保证它里面的所有属性都做到响应式
Vuex就是为了提供这样一个在多个组件间共享状态的插件

2.2. 管理什么状态呢?

有什么状态需要我们在多个组件间共享呢?
比如用户的登录状态，用户名称、头像、地理信息位置等，商品的收藏、购物车中的物品等
这些状态信息，我们可以都放在统一的地方，对他们进行保存和管理，而且他们还是响应式的

2.3. 多界面的状态管理

Vue已经帮我们做好了单个页面的状态管理，但是如果是多个界面呢?
多个视图都依赖同一个状态（一个状态改了，多个界面需要进行更新）
不同界面的Action都想修改同一个状态（Home.vue需要修改，Profile.vue也需要修改这个状态）

也就是说对于某些状态（状态1/状态2/状态3）来说只属于我们某一个视图，但是也有状态（状态a/状态b/状态c）属于多个视图共同想要维护的
状态1/状态2/状态3放在自己的房间中，自己管理自己用，没问题
但是状态a/状态b/状态c我们希望交给一个大管家来统一帮助我们管理
Vuex就是我们提供这个大管家的工具

全局单例模式（大管家）
我们现在要做的就是将共享的状态抽取出来，交给我们的大管家，统一进行管理
之后，每个视图，按照规定，进行访问和修改等操作
这就是Vuex背后的基本思想

2.4. Vuex核心概念

Vuex有几个比较核心的概念:

- State
- Getters
- Mutation
- Action
- Module

2.4.1. State单一状态树

Vuex提出使用单一状态树：英文名称是Single Source Truth,可以翻译成单一数据源

如果状态信息是保存到多个Store对象中的，那么之后的管理和维护等等都会变得非常困难
所以vuex使用了单一状态树来管理应用层级的全部状态
单一状态树能够让我们以最直接的方式找到某个状态的片段，而且在之后的维护和调试过程中，也可以非常方便的管理和维护

2.4.2. Getters基本使用

```
getters: {
powerCounter(state) {
    return state.counter * state.counter
},
more20stu(state) {
    return state.students.filter(s => s.age > 15)
},
more20stuLength(state, getters) {
    return getters.more20stu.length
},
moreAgeStu(state) {
    return function (age) {
    return state.students.filter(s => s.age > age)
    }
```

```
}
},
```

2.4.3. Mutation状态更新

Vuex的store状态的更新唯一方式:**提交Mutation**

Mutation主要包含两部分:

- 字符串的**事件类型**（type)
- 一个**回调函数**（handler），该回调函数的第一个参数就是state

Mutation定义方式:

```
mutations: {
  // 方法
  increment(state) {
    state.counter++
  },
  decrement(state) {
    state.counter--
  }
},
  incrementCount(state,count){
    state.counter+=count
```

通过Mutation更新:

```
  addition() {
    this.$store.commit("increment");
  },
  substraction() {
    this.$store.commit("decrement");
  },
  addCount(count){
    this.$store.commit("incrementCount",count)
  }
```

Mutation传递参数

在通过Mutation更新数据的时候，有可能我们希望携带一些额外的参数
参数被称为是Mutation的载荷(Payload)

Mutation中的代码:

```
addStudent(state,stu){
    state.students.push(stu)
  }
```

通过Mutation更新:

```
  addStudent(){
    const stu={id:105,name:'e',age:18}
    this.$store.commit('addStudent',stu)
  }
```

如果参数不是一个，这个时候，我么通常会以对象的形式传递参数，也就是payload是一个对象
这个时候可以再从对象中取出相关的信息

Mutation提交风格

上面的通过commit进行提交的是一种普通的方式
Vue还提供了另外一种风格，它是包含type属性的对象

```
addCount(count){
    // 普通的提交
    // this.$store.commit("incrementCount",count)


    // 特殊的提交风格
    this.$store.commit({
      type:'incrementCount',
      count
    })
  },
```

注意：这里的特殊提交风格中的count就不是一个数字了，而是变成了一个payload，这个payload是以对象的形式进行传值的
Mutation中最好写成以下形式：

```
  incrementCount(state,payload){
    state.counter+=payload.count
  },
```

Mutation响应规则

Vuex的store的state是响应式的，当state内的数据发生改变时，vue组件会自动更新
这就要求我们必须遵守一些Vuex对应的规则：

- 提前在store中初始化好所需的属性
- 当给state中的对象添加新属性时，使用下面的方式：
  - 方式1：使用Vue.set(obj,'newProp',123)
  - 方式2：用新对象给旧对象重新赋值

Mutation类型常量

在Mutation中，我么定义了很多的事件类型（也就是其中的方法名称）
当我们的项目增大时，Vuex管理的状态越来越多，需要更新状态的情况越来越多，那么意味着Mutation中的方法越来越多
方法过多，使用者需要花大量的精力去记住这些方法，甚至是多个文件间来回切换，查看方法名称，甚至如果不是复制的时候，可能还会出现写错的情况

Mutation同步函数

通常情况下，Vuex要求我们Mutation的方法必须是同步方法

- 主要的原因是当我们使用devtools时，devtools可以帮助我们捕捉mutation的快照
- 但是如果是异步操作，那么devtools将不能很好的跟踪这个操作什么时候会被完成

通常情况下，不要在mutation中进行异步的操作

2.4.4. Action

我们强调，不要用Mutation进行异步操作
但是某些情况下，我们确实希望Vuex中进行一些异步操作，比如网络请求，必然是异步的，这个时候应该怎么办呢?
这时候可以用Action来替代Mutation进行异步操作

2.4.5. Module

vue使用单一状态树，那么也意味着很多状态都会交给Vuex来管理
当应用变得非常复杂时，store对象就有可能变得相当臃肿
为了解决这个问题，Vuex允许我们将store分割成模块，而 每个模块拥有自己的state，mutation，action，getters等