promise

一、什么是promise?

二、promise规范

三、promise对象的实例方法

四、promise对象的静态方法

五、promise的使用

六、promise实现的原理

简介

Promise 是异步编程的一种解决方案，比传统的解决方案——回调函数和事件——更合理更强大。它由社区最早提出和实现，ES6 将其写进了语言标准，统一了用法，原生提供了Promise对象

所谓Promise，简单说就是一个容器，里面保存着某个未来才会结束的事件（通常是一个异步操作）的结果。从语法上说，Promise 是一个对象，从它可以获取异步操作的消息。

简介

使用new关键字创建一个promise实例, 构造函数promise必须接受一个函数(handle)作为参数
该函数（handle）包含resolve和reject两个函数，可以用于改变Promise的状态和传Promise的值

```
var promise = new Promise((resolve, reject) => {
  // 异步处理回调函数
  // 处理结束后调用resolve 或 reject方法
  if (操作成功){
    resolve(value);
  } else {
    reject(error);
  }
});
```

规范

- promise三种状态：pending(等待)、fulfilled或resolved(成功-已完成)、rejected(失败-已拒绝)
- 一个promise的状态只可能从等待到完成或者拒绝状态，不能逆向转换，同时完成和拒绝状态不能相互转换
- promise实例必须实现then方法，而且then必须返回一个promise,同一个promise的then可以调用多次，并且回调的执行顺序跟它们被定义时的顺序一致
- then方法接受两个参数：第一个参数是成功的回调，另一个是失败时的回调

Promise对象中的实例方法

实例方法是指在原型链prototype上的方法,Promise对象中有两个实例方法：

Promise.prototype.then

Promise.prototype.catch

```
> Promise
ƒ Promise() { [native code] }
> Promise.prototype
Promise {constructor: ƒ, then: ƒ, catch: ƒ, finally: ƒ, Symbol(Symbol.toStringTag): "Promise"}
  catch: ƒ catch()
  constructor: ƒ Promise()
  finally: ƒ finally()
  then: ƒ then()
  Symbol(Symbol.toStringTag): "Promise"
  __proto__: Object
>
```

```
promise.then((data) => {
  /*
    then方法提供一个供自定义的回调函数，若传入非函数，则会忽略当前then方法。
    回调函数中会把上一个then中返回的值当做参数值供当前then方法调用。
  */
}).catch((error) => {
  // .then()的一个子集，专用于接收promise对象的reject()传过来的error参数的
});
```

回调地狱写法

```
setTimeout(() => {
  let one = 2;
  console.log('第一个回调');
  setTimeout(() => {
    let two = one * 2;
    console.log(one * 2, '第二个回调');
    setTimeout(() => {
      console.log(two * 3, '第三个回调');
    }, 1000);
  }, 2000);
}, 1000);
```

promise写法

```
let promise = new Promise((resolve, reject) => {
  setTimeout(() => {
    console.log('第一个回调');
    var one = 2;
    resolve(one);
  })
});
promise.then(value => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      console.log(value * 2, '第二回调');
      resolve(value * 2);
    }, 2000)
  });
}).then(val => {
  setTimeout(() => {
    console.log(val * 3, '第三个回调');
    return val * 3;
  }, 1000)
});
```

Promise对象的静态方法

Promise.all

- 参数：promiseArray，是一个promise实例数组
- 方法作用：将多个Promise实例包装，生成并返回一个新的promise实例。参数传递promise数组中所有的promise实例都变为fulfilled的时候，该方法才会返回， 新创建的promise则会使用这些promise的值。如果参数中的任何一个promise为reject的话，则整个Promise.all调用会立即终止，并返回一个reject的新的promise对象

```
var p1 = new Promise(function(resolve, reject) {
  setTimeout(function() {
    resolve(3);
  }, 3000);
});
var p2 = new Promise(function(resolve, reject) {
    setTimeout(function() {
      resolve(2);
    }, 2000);
});
Promise.all([p1, p2]).then(function(value) {
  // 返回所有结果组合的数组 [3, 2]
});
```

Promise对象的静态方法

Promise.race

- 参数：promiseArray，是一个promise实例数组
- 方法作用：将多个Promise实例包装，生成并返回一个新的promise实例。参数promise数组中的任何一个promise实例如果变为fulfilled或者rejected的话，该函数就会返回，并使用这个promise实例的值进行fulfilled或者rejected。

```
let data1 = Promise.resolve(1);
let data2 = Promise.resolve(2);
Promise.race([data1, data2]).then(result => {
  // 返回最先完成的结果
  console.log(result);  // 1
});
```

Promise.resolve

将现有对象转为Promise对象

Promise.reject

Promise.reject()方法也会返回一个新的promise实例，该实例的状态为rejected

```
var p = Promise.resolve('Hello');

p.then(function (s){
      console.log(s) // Hello
});
```

promise的使用

一、使用promise实现一个简单的axios

```
axios(obj) {
  const {url, method, data} = obj;
  return new Promise((resolve, reject) => {
    var xhr = new XMLHttpRequest();
    xhr.open(method, url, true);
    xhr.responseType = 'json';
    xhr.setRequestHeader('Accept', 'application/json');
    xhr.onreadystatechange = () => {
      if (xhr.readyState !== 4) {
        return;
      }
      if (xhr.status === 200) {
        resolve(xhr.response);
      } else {
        reject(new Error(xhr.statusText));
      }
    };
    xhr.onerror = () => {
      reject(new Error(xhr.statusText));
    };
    xhr.send(data);
  });
},
```

promise的使用

二、promise的reject实现$checkResp

```
const checkResp = function(resp, emptyDefault) {
  if (resp && (resp.code === 0 || resp.code === 1)) {
    if (resp.code === 1 && emptyDefault) {
      return emptyDefault;
    }
    return resp.data;
  }
  return Promise.reject(resp);
};

export default {
  install(Vue) {
    Vue.checkResp = checkResp;
    Object.defineProperty(Vue.prototype, '$checkResp', { value: Vue.checkResp });
  }
};
```

promise的使用

三、promiser的resolve在实例中的作用

```
OVERVIEW_GET_SUBJECTS(ctx, { classId }) {
  if (ctx.state.subjects.length) {
    return Promise.resolve(ctx.state.subjects);
  }
  return Vue.http.get(`/teacher-v2/classes/${classId}/subjects`)
    .then(resp => Vue.checkResp(resp, []))
    .then(data => {
      let newData = data.filter(item => {
        return item !== '未知';
      });
      // 代提测接口
      if (newData.length) {
        Vue.http.post('/teacher-v2/subjects/standard-subject/batch', { subjects: newData
          ctx.commit('OVERVIEW_SET_SUBJECTS', { subjects: newData, standardSubject: batch
        }).catch(() => {
          ctx.commit('OVERVIEW_SET_SUBJECTS', { subjects: newData, standardSubject: {} })
        });
      } else {
        ctx.commit('OVERVIEW_SET_SUBJECTS', { subjects: [], standardSubject: {} });
      }
      return newData;
    })
    .catch((err) => {
      Vue.handleError(err, '获取班级学科信息失败');
    });
```

promise的使用

四、promise.all在实例中的使用

```
let getProfile = Vue.http.get('/teacher-v2/teachers/info').then(resp => Vue.checkResp(resp, {}));
// 获取考试数据屏蔽配置
let getExamConfig = Vue.http.get('/teacher-v2/config/exam-config').then(resp => Vue.checkResp(res
return Promise.all([getProfile, getExamConfig]).then(result => {
  let profile = result[0];
  let examConfig = result[1];
  if (profile.classes) {
    profile.classes = initClasses(profile.classes);
  }
  window.user = profile && profile.name; // 监控收集用户名信息
  let data = { profile, examConfig };
  ctx.commit('USER_SET_PROFILE', data);
  return data;
}).catch(err => {
  Vue.handleError(err, '获取教师基本信息失败');
});
```

Promise实现的原理

1. 一、定义一个class，接收一个函数作为参数
2. 二、存储状态和值，三个状态，pending,fulfilled,rejected,值是在状态改变时传递给回调函数的值
3. 三、生成一个then方法，接收两个参数，参数可选，第一个是成功状态传入的值，第二个是失败状态传入的值
4. 四、生成catch方法，相当于调用then方法，但只接收rejected状态的回调函数
5. 五、生成resolve方法，静态方法，如果参数是promise实例，则直接返回这个实例
6. 六、生成reject方法，静态方法，同resolve
7. 七、生成all方法,作为返回值的集合
8. 八、生成race方法，只要有一个实例先改变状态，新的promise的实例状态就跟着改变
9. 九、生成finally方法，不管promise最后状态如何都会执行

课后小题练习

```
console.log(1);
const promise = new Promise((resolve, reject) => {
  console.log(2);
  resolve();
  console.log(3);
});
promise.then(() => {
  console.log(4);
});
console.log(5);
```

运行结果：1 2 3 5 4

总结：Promise 构造函数是同步执行的，promise.then中的函数是异步执行的

课后小题练习

```
console.log('start');
new Promise((resolve,reject) => {
    setTimeout(() => {
      resolve('hello');
    },2000);
}).then((value)=>{
  console.log(value);
  (() => {
      return new Promise((resolve) => {
        setTimeout(() => {
          resolve('world');
        },3000)
      })
  })();
}).then((value)=>{
  console.log(value);
})
```

运行结果：

立即输出　start

2秒后输出　hello

3秒后输出　undefined

总结：如果在一个then（）中没有返回一个新的promise，则return什么下一个then就接受什么，在实例中第一个then中并没有return任何值，所以在下一个then默认接收undefined

课后小题练习

```
let promise = new Promise((resolve, reject) => {
  setTimeout(() => {
    console.log('第一个回调');
    resolve(2);
  });
});
promise.then(val => {
  setTimeout(() => {
    console.log(val * 2, '第二回调');
    return (val * 2);
  }, 2000);
}).then(value => {
  setTimeout(() => {
    console.log(value * 3, '第三个回调');
    return value * 3;
  }, 1000);
});
```

运行结果：

立即输出　第一个回调
1秒后输出　NaN 第三个回调
2秒后输出　4 第二个回调

总结：当你给then返回一个非promise对象，then只接收同步的返回值，反之，当你给then返回一个promise对象，那么then就等待promise对象生成，然后等resolve和reject传递参数，等多久都能等

课后小题练习

```
new Promise((resolve,reject) => {
  resolve();
})
.then(value => {
  console.log('done 1');
  throw new Error('done 1 error');
})
.catch(err => {
  console.log('错误信息1: '+err);
  throw new Error('catch error');
})
.then(value => {
  console.log('done 2');
})
.catch(err => {
  console.log('错误信息2: '+err);
})
```

运行结果：

done 1

错误信息1：Error: done 1 error

错误信息2：Error: catch error

总结：如果在catch中也抛出了错误，则后面的then的第一个函数不会执行，因为返回的promise的状态已经为rejected了

THANKS~