ES6 PPT1512021: Promise 对象

Thursday, November 5, 2015 10:15 AM

承诺 Promise 对象：

1. 典型应用：承诺立即进行一项任务(eg. mission)，待任务成败分晓后进行相应行动(eg. action)

```
var missionKillBill = new Promise(function kill(myActionAfterBillKilled, myActionAfterMeKilled) {
  while (true) {
    let res = myActionsToKill(); // res: FindBill |KillingBill | BillKilled | MeKilled | …
    if (res == 'BillKilled') { myActionAfterBillKilled(); break; }
    if (res == 'MeKilled') { myActionAfterMeKilled(); break; }
  }
  console.log('missionKillBill is over');
});
missionKillBill
  .then(function actionAfterBillKilled() { console.log('bill killed'); })
  .catch(function actionAfterMeKilled() { console.log('me killed'); });
```

2. 基本模型：3种状态 - 1. 执行中 pending、 2. 成功 fulfilled、 3. 失败 rejected

```
function fnRunMission(myfnRunAfterMissionSucceeded, myfnRunAfterMissionFailed) { // 定义执行任务函数
  // 执行代码段...确认任务成功
  if (myMissionSucceeded) myfnRunAfterMissionSucceeded(myParamSucceeded);
  // 执行代码段...确认任务失败
  if (myMissionFailed) myfnRunAfterMissionFailed(myParamFailed);
  // ...若代码段执行中抛出异常，会被确认为任务失败并自动调用 myfnRunAfterMissionFailed...
}
function fnRunAfterMissionSucceeded(paramSucceeded) {; } // 定义任务成功后调用函数，最多一个参数
function fnRunAfterMissionFailed(paramFailed) {; } // 定义任务失败后调用函数，最多一个参数
// 定义并执行任务
var mission = new Promise(fnRunMission);
// 任务成败分晓后的行动定义 Promise - then 方式
mission.then(fnRunAfterMissionSucceeded, fnRunAfterMissionFailed);
// 任务成败分晓后的行动定义 Promise - then - catch 方式（推荐）
mission.then(fnRunAfterMissionSucceeded).catch(fnRunAfterMissionFailed);
```

3. Promise 成败状态一旦确定不再更改，并且该状态会一直存在

```
new Promise((done, fail) => { fail(); done(); }).then(() => { console.log('成') }, () => { console.log('败') }); // 败
```

4. 使用链式 then 与 catch 串行依次处理多个 Promise：
   a. 原理：then 与 catch 均返回一新 Promise 实例
   b. 如果传递给 then 的函数不返回 Promise 实例，then 将创建一个并返回之

c. 典型应用：

```
function waitClick(ms) {
 return new Promise((done, fail) => {
  window.onclick = (e) => { done(`${ms}ms内完成点击`) };
  setTimeout(() => { fail(`${ms}ms超时`) }, ms);
 });
}
waitClick(5000)
 .then((msg) => { console.log('成功: ' + msg); return waitClick(3000); })
 .then((msg) => { console.log('成功: ' + msg); return waitClick(1000); })
 .then((msg) => { console.log('成功: ' + msg); })
 .catch((msg) => { console.log('失败: ' + msg); })
 .then(() => { console.log('感谢参与'); });
```

5. 并行处理多个 Promise：
   a. 使用 Promise.all([p1,p2,p3]) 包装多个参数实例并集成它们最终状态结果返回一个新实例
      i. 新实例：任一参数实例 pn: p1|p2|p3 失败 => 失败(pn调用参数)
      ii. 新实例：所有参数实例 p1&p2&p3 成功 => 成功([p1调用参数,p2调用参数,p3调用参数])
   b. 使用 Promise.race([p1,p2,p3]) 包装多个参数实例并返回最先获得状态结果的参数实例
6. Promise 快捷方法：
   a. 异常或失败状态处理：Promise.prototype.catch(fn)

```
myPromise.then(null, fnRunAfterMissionFailed)   <===>
myPromise.catch(fnRunAfterMissionFailed)
```

   b. 转化生成一个 Promise 实例：Promise.resolve(param)

```
Promise.resolve('foo')   <===>   new Promise(resolve => resolve('foo'))
Promise.resolve(jQuery.ajax('/data.json'))  // 转化 jQuery 对象 deferred 为 Promise
```

   c. 转化生成一个状态失败的 Promise 实例：Promise.reject(param)

```
Promise.reject('foo')   <===>   new Promise((resolve, reject) => reject('foo'))
```

7. 缺点：无法中途取消 Promise 执行，除非设置回调函数 callback 否则外部无法获取 Promise 执行中产生的异常，Promise 执行时外部无法获知其执行进度
8. 代码应用：
   a. 实现 done() 以便全局最终可处理任何 Promise 执行中抛出的异常：

```
Promise.prototype.done = function (onFulfilled, onRejected) {
 this.then(onFulfilled, onRejected)
  .catch(function (reason) {
   // 抛出一个全局错误
   setTimeout(() => { throw reason }, 0);
  });
};
asyncFunc()
 .then(f1)
 .catch(r1)
```

```
  .then(f2)
  .done();
```

b. 实现 finally() 以便最终无论状态如何均执行指定回调：

```
Promise.prototype.finally = function (callback) {
  let P = this.constructor;
  return this.then(
    value  => P.resolve(callback()).then(() => value),
    reason => P.resolve(callback()).then(() => { throw reason })
    );
};
server.listen(0)
  .then(function () {
    // run test
  })
  .finally(server.stop);
```

c. 使用 Promise.all + jQuery deferred API并行加载数据：

```
function loadScript(url) {
  console.log('开始加载脚本: ' + url);
  return Promise.resolve($.getScript(url, (msg) => {
    console.log('已完成加载脚本: ' + url); // 即使已超时，仍然会继续完成加载脚本
  }));
}
Promise.all([
  loadScript('//cdn.bootcss.com/react/0.14.2/react.js'),
  loadScript('//cdn.bootcss.com/react/0.14.2/react-dom.js'),
])
  .then((msgs) => { console.log('脚本全部成功加载: ' + msgs); })
  .catch((msg) => { console.log('脚本加载失败: ' + msg); });
```

d. 探索异步执行顺序：

```
var html = document.body;
var myPrm = new Promise(function (resolve, reject) {
        alert('alert 1');
        resolve();
        html.innerHTML = 'info 1';
});
myPrm.then(function () {
        html.innerHTML = 'info 2';
        alert('alert 2');
        return new Promise(function (resv, rejt) {
                alert('alert 2.1');
                resv(2.1);
                rejt(2.1); // 无效
                alert('alert 2.2');
        });
}).then(function (param) {
        html.innerHTML = 'info 3 with param ' + param;
        alert('alert 3');
}).then(function (param) {
        html.innerHTML = 'info 4 with param ' + param;
        alert('alert 4');
```

```
}).catch(function (param) {
        html.innerHTML = 'error with param ' + param;
});
alert("[outer alert] html is " + html.innerHTML);
```

各类应用：异步数据加载(图像、JSON 数据...)，限时/定时任务处理，一次性事件，串行顺序获取异步数据...

附录：jQuery 对象 deferred 可实现 promise 类似功能 $.Deferred() <==> new Promise()

myDeferred = $.Deferred() <==> myPromise = new Promise()
myDeferred.resolve(...params) <==> Promise.resolve(param)
myDeferred.reject(...params) <==> Promise.reject(param)
myDeferred.fail(...fnAfterFails) <==> myPromise.catch(fnAfterFail)
myDeferred.then(fnAfterSuccess, fnAfterFail, progress) <==> myPromise.then(fnAfterSuccess, fnAfterFail)
$.when(...deferreds) <==> Promise.all([...promises])