03 手撕Promise之从0开始实现完整的Promise对象

1.定义对象

分析经典Promise对象

经典的Promise对象结构代码如下：

1.查看空Promise对象的结构和输出结果：

```
var p = new Promise(function(resolve,reject){
console.log(resolve,reject)
})
console.log(p)
```

输出结果如下：

```
ƒ () { [native code] } ƒ () { [native code] }
Promise
[[Prototype]]: Promise
[[PromiseState]]: "pending"
[[PromiseResult]]: undefined
```

2.查看fulfilled状态下的Promise对象：

```
var p = new Promise(function(resolve,reject){
  resolve('已完成')
})
console.log(p)
```

输出结果如下：

```
Promise {<fulfilled>: '已完成'}
[[Prototype]]: Promise
[[PromiseState]]: "fulfilled"
[[PromiseResult]]: "已完成"
```

3.查看rejected状态下的Promise对象：

```
var p = new Promise(function(resolve,reject){
reject('已拒绝')
})
console.log(p)
```

输出结果如下：

```
Promise {<rejected>: '已拒绝'}
[[Prototype]]: Promise
[[PromiseState]]: "rejected"
[[PromiseResult]]: "已拒绝"
Uncaught (in promise) 已拒绝
```

Promise对象的基本结构定义

根据Promise对象的特点分析，Promise存在状态属性和Promise的值的属性。初始化Promise时需要传入一个回调函数来进行对象的基本设置，回调函数具备两个参数resolve和reject，两个参数均为函数。所以初始化代码如下：

```
function MyPromise(fn){
  //promise的初始状态为pending，可变成fulfilled或rejected其中之一
  this.promiseState = 'pending'
  this.promiseValue = undefined
  var resolve = function(){

  }
  var reject = function(){

  }
  if(fn){
    fn(resolve,reject)
  }else{
    throw('Init Error,Please use a function to init MyPromise!')
  }
}
```

根据对象特性，初始化Promise时的回调函数是同步执行的，所以此时的fn直接调用即可。

在调用resolve和reject时，需要将Promise对象的状态设置为对应的fulfilled和rejected，其中需要传入Promise当前的结果，所以此时应该将resolve和reject修改为如下结构。

```
//保存上下文对象
var _this = this
var resolve = function(value){
  if(_this.promiseState == 'pending'){
    _this.promiseState = 'fulfilled'
    _this.promiseValue = value
  }
}
var reject = function(value){
  if(_this.promiseState == 'pending'){
     _this.promiseState = 'rejected'
     _this.promiseValue = value
  }
```

```
}
```

定义完内部结构之后需要思考Promise在状态变更为fulfilled以及状态变更为rejected时对应的then和catch会相应执行，所以需要将对象的两个函数初始化：

```
MyPromise.prototype.then = function(callback){

}
MyPromise.prototype.catch = function(callback){

}
```

那么初始对象的结构应该整体是这样的：

```
function MyPromise(fn){
  //promise的初始状态为pending，可变成fulfilled或rejected其中之一
  this.promiseState = 'pending'
  this.promiseValue = undefined
  var _this = this
  var resolve = function(value){
    if(_this.promiseState == 'pending'){
      _this.promiseState = 'fulfilled'
      _this.promiseValue = value
    }
  }
  var reject = function(value){
     if(_this.promiseState == 'pending'){
       _this.promiseState = 'rejected'
       _this.promiseValue = value
    }
  }
  if(fn){
    fn(resolve,reject)
  }else{
    throw('Init Error,Please use a function to init MyPromise!')
  }
}
MyPromise.prototype.then = function(callback){

}
MyPromise.prototype.catch = function(callback){

}
```

实现then的调用

在实现了初始结构之后，我们需要使用MyPromise按照Promise的方式进行编程，来实现他的流程控制部分了。首先我们需要让then跑起来。

定义调用代码：

```
var p = new MyPromise(function(resolve,reject){
  resolve(123)
})
console.log(p)
p.then(function(res){
  console.log(res)
})
```

此时执行代码时控制台会输出如下内容：

```
MyPromise
promiseState: "fulfilled"
promiseValue: 123
[[Prototype]]: Object
```

我们发现我们定义的Promise对象状态已经变更但是then中的回调函数没有执行。

接下来我们实现then的触发：

```
//在MyPromise中改造该部分代码如下
//定义then的回调函数
this.thenCallback = undefined

var resolve = function(value){
  if(_this.promiseState == 'pending'){
    _this.promiseState = 'fulfilled'
    _this.promiseValue = value
    //异步的执行then函数中注册的回调函数
    setTimeout(function(){
      if(_this.thenCallback){
        _this.thenCallback(value)
      }
    })
  }
}
```

```
//在then中编写如下代码
MyPromise.prototype.then = function(callback){
  //then第一次执行时注册回调函数到当前的Promise对象
  this.thenCallback = function(value){
    callback(value)
  }
}
```

在两处改造完成之后访问网页会发现控制台上可以输出then函数中的回调执行的结果并且该结果的参数就是resolve传入的值。

```
MyPromise {promiseState: 'fulfilled', promiseValue: 123, thenCallback: undefined}
promise.html:51 123
```

当前代码效果如下：

```
function MyPromise(fn){
  //promise的初始状态为pending，可变成fulfilled或rejected其中之一
  this.promiseState = 'pending'
  this.promiseValue = undefined
  var _this = this
  //定义then的回调函数
  this.thenCallback = undefined

  var resolve = function(value){
    if(_this.promiseState == 'pending'){
      _this.promiseState = 'fulfilled'
      _this.promiseValue = value
      //异步的执行then函数中注册的回调函数
      setTimeout(function(){
        if(_this.thenCallback){
          _this.thenCallback(value)
        }
      })
    }
  }
  var reject = function(value){
    if(_this.promiseState == 'pending'){
      _this.promiseState = 'rejected'
      _this.promiseValue = value
    }
  }
  if(fn){
    fn(resolve,reject)
  }else{
    throw('Init Error,Please use a function to init MyPromise!')
  }
}
MyPromise.prototype.then = function(callback){
  //then第一次执行时注册回调函数到当前的Promise对象
  this.thenCallback = function(value){
    callback(value)
  }
}
MyPromise.prototype.catch = function(callback){
```

```
}
var p = new MyPromise(function(resolve,reject){
  resolve(123)
})
console.log(p)
p.then(function(res){
  console.log(res)
})
```

实现then的异步链式调用

通过上面的编程已经可以实现then自动触发，但是当前我们如果将代码变成如下效果时只有一个then能执行。而且控制台会报错

```
var p = new MyPromise(function(resolve,reject){
  resolve(123)
})
console.log(p)
p.then(function(res){
  console.log(res)
}).then(function(res){
  console.log(res)
}).then(function(res){
  console.log(res)
})
```

控制台信息如下：

```
MyPromise {promiseState: 'fulfilled', promiseValue: 123, thenCallback: undefined}
promise.html:52 Uncaught TypeError: Cannot read properties of undefined (reading
'then')
    at promise.html:52
(anonymous) @ promise.html:52
promise.html:51 123
```

针对该情况，我们需要对Promise的流程控制代码做进一步的加强以实现链式调用，并且在链式调用的过程中将每次的结果顺利的向下传递。

```
//resolve部分代码实现
var resolve = function(value){
  if(_this.promiseState == 'pending'){
    _this.promiseValue = value
    _this.promiseState = 'fulfilled'
    //当传入的类型是Promise对象时
    if(value instanceof MyPromise){
      value.then(function(res){
        _this.thenCallback(res)
      })
```

```
    }else{
      //当传入的数据类型是普通变量时
      setTimeout(function(){
        if(_this.thenCallback){
          _this.thenCallback(value)
        }
      })
    }
  }
}
//then函数代码实现
MyPromise.prototype.then = function(callback){
  var _this = this
  return new MyPromise(function(resolve,reject){
    _this.thenCallback = function(value){
      var callbackRes = callback(value)
      resolve(callbackRes)
    }
  })
}
```

将then代码修改为如下之后我们将调用代码更改如下

```
var p = new MyPromise(function(resolve){
  resolve(new MyPromise(function(resolve1){
    resolve1('aaa')
  }))
})
p.then(function(res){
  console.log(res)
  return 123
}).then(function(res){
  console.log(res)
  return new MyPromise(function(resolve){
    setTimeout(function(){
      resolve('Promise')
    },2000)
  })
}).then(function(res){
  console.log(res)
})
console.log(p)
```

会惊喜的发现MyPromise对象可以正常的工作了并且还可以实现何时调用resolve何时执行then的操作

```
MyPromise {promiseValue: MyPromise, promiseState: 'fulfilled', catchCallback:
undefined, thenCallback: ƒ}
test.html:57 aaa
test.html:60 123
test.html:67 Promise
```

当前状态的代码如下

```
function MyPromise(fn){
  var _this = this
  this.promiseValue = undefined
  this.promiseState = 'pending'
  this.thenCallback = undefined
  this.catchCallback = undefined
  var resolve = function(value){
    if(_this.promiseState == 'pending'){
      _this.promiseValue = value
      _this.promiseState = 'fulfilled'
      if(value instanceof MyPromise){

        value.then(function(res){
          _this.thenCallback(res)
        })
      }else{
        setTimeout(function(){
          if(_this.thenCallback){
            _this.thenCallback(value)
          }
        })
      }
    }
  }
  var reject = function(err){

  }
  if(fn){
    fn(resolve,reject)
  }else{
    throw('Init Error,Please use a function to init MyPromise!')
  }
}
MyPromise.prototype.then = function(callback){
  var _this = this
  return new MyPromise(function(resolve,reject){
    _this.thenCallback = function(value){
      var callbackRes = callback(value)
      resolve(callbackRes)
    }
```

```
  })
}
var p = new MyPromise(function(resolve){
  resolve(new MyPromise(function(resolve1){
    resolve1('aaa')
  }))
})
```

实现catch的流程处理

当Promise的对象触发reject操作的时候他的状态会变更为rejected，此时会触发catch函数，并且catch函数触发后流程结束。

首先仿照then的方式在MyPromise对象中定义好初始通知函数

```
//定义catch的回调函数
this.catchCallback = undefined
var reject = function(err){
  if(_this.promiseState == 'pending'){
    _this.promiseValue = err
    _this.promiseState = 'rejected'
    setTimeout(function(){
      if(_this.catchCallback){
        _this.catchCallback(err)
      }
    })
  }
}
```

然后在catch函数中做如下处理

```
MyPromise.prototype.catch = function(callback){
  var _this = this
  return new MyPromise(function(resolve,reject){
    _this.catchCallback = function(errValue){
      var callbackRes = callback(errValue)
      resolve(callbackRes)
    }
  })
}
```

调用代码如下

```
var p = new MyPromise(function(resolve,reject){
  reject('err')
})
p.catch(function(err){
  console.log(err)
})
```

当运行此时代码时我们会发现我们的Promise对象在控制台上可以直接触发catch的回调执行并输出对应的结果

```
MyPromise {promiseValue: 'err', promiseState: 'rejected', thenCallback: undefined,
catchCallback: ƒ}
test.html:73 err
```

实现跨对象执行catch

在上面的案例中已经可以执行MyPromise的catch函数了，但是如果将调用代码改为如下行为会发现catch函数不会执行

```
var p = new MyPromise(function(resolve,reject){
  reject(123)
})
console.log(p)
p.then(function(res){
  console.log(res)
}).catch(function(err){
  console.log(err)
})
```

这是因为按照我们编写的代码流程Promise对象会自动变更状态为rejected并且catch的回调函数无法注册，所以Promise的流程就断了。这个时候需要追加判断代码让Promise在rejected时如果没有catchCallback再去检测是否存在thenCallback

```
var reject = function(err){
  if(_this.promiseState == 'pending'){
    _this.promiseValue = err
    _this.promiseState = 'rejected'
    setTimeout(function(){
      if(_this.catchCallback){
        _this.catchCallback(err)
      }else if(_this.thenCallback){
        _this.thenCallback(err)
      }else{
        throw('this Promise was reject,but can not found catch!')
      }
    })
  }
}
```

该步骤操作完毕之后我们需要将then函数中的逻辑再次更改为如下

```
MyPromise.prototype.then = function(callback){
  var _this = this
  //实现链式调用并且每个节点的状态是未知的所以每次都需要返回一个新的Proimse对象
  return new MyPromise(function(resolve,reject){
    //then第一次执行时注册回调函数到当前的Promise对象
    _this.thenCallback = function(value){
      //判断如果进入该回调时Promise的状态为rejected那么就直接触发后续Promise的catchCallback
      //直到找到catch
      if(_this.promiseState == 'rejected'){
        reject(value)
      }else{
        var callbackRes = callback(value)
        resolve(callbackRes)
      }
    }
  })
}
```

修改如下之后调用代码改造为

```
var p = new MyPromise(function(resolve,reject){
  reject('err')
})
p.then(function(res){
  console.log(res)
  return 111
}).then(function(res){
  console.log(res)
  return 111
}).then(function(res){
  console.log(res)
  return 111
}).catch(function(err){
  console.log(err)
})
console.log(p)
```

输出结果为

```
MyPromise {promiseValue: 'err', promiseState: 'rejected', catchCallback: undefined,
thenCallback: ƒ}
test.html:91 err
```

实现链式调用的中断

本文仅介绍通过返回Promise对象来中断链式调用，首先在Promise的原型对象上增加reject方法如下:

```
MyPromise.reject = function(value){
  return new MyPromise(function(resolve,reject){
    reject(value)
  })
}
```

然后初始化如下调用代码

```
var p = new MyPromise(function(resolve,reject){
  resolve(123)
})
console.log(p)
p.then(function(res){
  console.log('then1执行')
  return 456
}).then(function(res){
  console.log('then2执行')
  return MyPromise.reject('中断了')
}).then(function(res){
  console.log('then3执行')
  return 789
}).then(function(res){
  console.log('then4执行')
  return 666
}).catch(function(err){
  console.log('catch执行')
  console.log(err)
})
```

最后修改调试代码中的then

```
MyPromise.prototype.then = function(callback){
  var _this = this
  return new MyPromise(function(resolve,reject){
    _this.thenCallback = function(value){
      if(_this.promiseState == 'rejected'){
        reject(value)
      }else{
        var callbackRes = callback(value)
        if(callbackRes instanceof MyPromise){
          if(callbackRes.promiseState == 'rejected'){
            callbackRes.catch(function(errValue){
              reject(errValue)
            })
          }
        }else{
          resolve(callbackRes)
        }
```

```
      }
    }
  })
}
```

根据代码分析处理逻辑，然后查看运行结果：

```
MyPromise {promiseState: 'fulfilled', promiseValue: 123, thenCallback: undefined,
catchCallback: undefined}
promise.html:100 then1执行
promise.html:103 then2执行
promise.html:112 catch执行
promise.html:113 中断了
```

最后我们发现在返回Promise.reject()之后then的链式调用便中断了。

实现all和race

1.Promise.all的实现

```
MyPromise.all = function(promiseArr){
  var resArr = []
  var errValue = undefined
  var isRejected = false
  return new MyPromise(function(resolve,reject){
    for(var i=0;i<promiseArr.length;i++){
      (function(i){
        promiseArr[i].then(function(res){
          resArr[i] = res
          let r = promiseArr.every(item => {
            return item.promiseState == 'fulfilled'
          })
          if(r){
            resolve(resArr)
          }
        }).catch(function(err){
          isRejected = true
          errValue = err
          reject(err)
        })
      })(i)

      if(isRejected){
        break
      }
    }
  })
}
```

1.Promise.race的实现

```
MyPromise.race = function(promiseArr){
  var end = false
  return new MyPromise(function(resolve,reject){
    for(var i=0;i<promiseArr.length;i++){
      (function(i){
        promiseArr[i].then(function(res){
          if(end == false){
            end = true
            resolve(res)
          }

        }).catch(function(err){
          if(end ==false){
            end = true
            reject(err)
          }
        })
      })(i)
    }
  })
}
```

实现generator的调用

执行器代码如下：

```
/**
      * fn:Generator函数对象
      */
function generatorFunctionRunner(fn){
  //定义分步对象
  let generator = fn()
  //执行到第一个yield
  let step = generator.next()
  //定义递归函数
  function loop(stepArg,generator){
    //获取本次的yield右侧的结果
    let value = stepArg.value
    //判断结果是不是Promise对象
    if(value instanceof MyPromise|| value instanceof Promise){
      //如果是Promise对象就在then函数的回调中获取本次程序结果
      //并且等待回调执行的时候进入下一次递归
      value.then(function(promiseValue){
        if(stepArg.done == false){
          loop(generator.next(promiseValue),generator)
        }
```

```
      })
    }else{
      //判断程序没有执行完就将本次的结果传入下一步进入下一次递归
      if(stepArg.done == false){
        loop(generator.next(stepArg.value),generator)
      }
    }
  }
  //执行动态调用
  loop(step,generator)
}
```

调用代码如下：

```
function * test(){
  let res1 = yield new MyPromise(function(resolve){
    setTimeout(function(){
      resolve('第一秒')
    },1000)
  })
  console.log(res1)
  let res2 = yield new MyPromise(function(resolve){
    setTimeout(function(){
      resolve('第二秒')
    },1000)
  })
  console.log(res2)
  let res3 = yield new MyPromise(function(resolve){
    setTimeout(function(){
      resolve('第三秒')
    },1000)
  })
  console.log(res3)
}
generatorFunctionRunner(test)
```

总结

通过简单的代码片段我们便可以快速的实现一个微型的Promise对象， 实现手写代码封装Promise对象虽然对工作没有太大的帮助，但是如果可以根据分析Promise的特性，通过JS原始的异步流程控制方式，来仿真成功Promise对象的内部逻辑，就代表你的JS编程水平已经脱离了普通业务开发的程序员的水平了，希望本文的思路可以对前端路上进修的同学们提供帮助，最后附上源代码。

源代码

```
function MyPromise(fn){
  var _this = this
```

```
  this.promiseValue = undefined
  this.promiseState = 'pending'
  this.thenCallback = undefined
  this.catchCallback = undefined
  var resolve = function(value){
    if(_this.promiseState == 'pending'){
      _this.promiseValue = value
      _this.promiseState = 'fulfilled'
      if(value instanceof MyPromise){
        if(_this.thenCallback){
          value.then(function(res){
            _this.thenCallback(res)
          })
        }
      }else{
        setTimeout(function(){
          if(_this.thenCallback){
            _this.thenCallback(value)
          }
        })
      }
    }
  }
  var reject = function(err){
    if(_this.promiseState == 'pending'){
      _this.promiseValue = err
      _this.promiseState = 'rejected'
      setTimeout(function(){
        if(_this.catchCallback){
          _this.catchCallback(err)
        }else if(_this.thenCallback){
          _this.thenCallback(err)
        }else{
          throw('this Promise was reject,but can not found catch!')
        }
      })
    }
  }
  if(fn){
    fn(resolve,reject)
  }else{
    throw('Init Error,Please use a function to init MyPromise!')
  }
}
MyPromise.prototype.then = function(callback){
  var _this = this
  return new MyPromise(function(resolve,reject){
    _this.thenCallback = function(value){
      if(_this.promiseState == 'rejected'){
```

```
        reject(value)
      }else{
        var callbackRes = callback(value)
        if(callbackRes instanceof MyPromise){
          if(callbackRes.promiseState == 'rejected'){
            callbackRes.catch(function(errValue){
              reject(errValue)
            })
          }
        }else{
          resolve(callbackRes)
        }

      }

    }
  })
}
MyPromise.prototype.catch = function(callback){
  var _this = this
  return new MyPromise(function(resolve,reject){
    _this.catchCallback = function(errValue){
      var callbackRes = callback(errValue)
      resolve(callbackRes)
    }
  })
}
MyPromise.reject = function(value){
  return new MyPromise(function(resolve,reject){
    reject(value)
  })
}
MyPromise.resolve = function(value){
  return new MyPromise(function(resolve){
    resolve(value)
  })
}
MyPromise.all = function(promiseArr){
  var resArr = []
  var errValue = undefined
  var isRejected = false
  return new MyPromise(function(resolve,reject){
    for(var i=0;i<promiseArr.length;i++){
      (function(i){
        promiseArr[i].then(function(res){
          resArr[i] = res
          let r = promiseArr.every(item => {
            return item.promiseState == 'fulfilled'
          })
```

```
            if(r){
              resolve(resArr)
            }
          }).catch(function(err){
            isRejected = true
            errValue = err
            reject(err)
          })
        })(i)

        if(isRejected){
          break
        }
      }
    })
}

MyPromise.race = function(promiseArr){
  var end = false
  return new MyPromise(function(resolve,reject){
    for(var i=0;i<promiseArr.length;i++){
      (function(i){
        promiseArr[i].then(function(res){
          if(end == false){
            end = true
            resolve(res)
          }

        }).catch(function(err){
          if(end ==false){
            end = true
            reject(err)
          }
        })
      })(i)
    }
  })
}
```