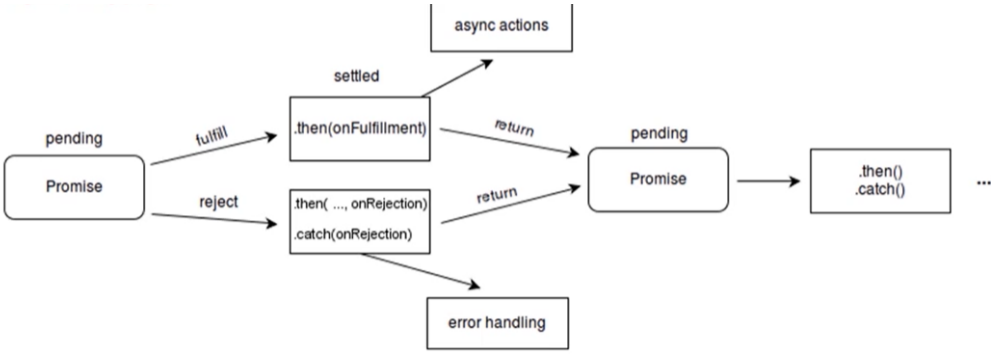
课程目标

promise神器

promise+Coding示例

promise代表异步求值的过程和结果，在很长一段时间没有的找到一种异步求值的方法，异步求值其实一种行为，这种行为我们需要用函数去描述，函数会给大家的架构带来困难，函数毕竟是需要一个执行的过程，前端我们需要用一个值来描述异步求值的过程，这时候就出现的promise， promise即将发生或者未来的一个值。

- pending
- fulfilled
- rejected



pending 代表过程

fulfilled 代表成功结果

rejected代表失败结果

注意点：async语法糖 配合await 一起使用

await 后面是promise的话 const a=await Promise a的值是resolve(value)值或者reject(reason)--注意reject需要catch捕获，或者使用try catch

await 后面是非promise的话 const a=await 非Promise a为非Promise值(无意义)

promise函数的常见用法(静态方法+实例化方法)

静态方法

Promise.reslove() 快速生成resove状态promise

Promise.reject() 快速生成reject状态promise

Promise.all() - 并发
Promise.race() - 竞争

实例化方法:
then() 返回promise
catch() 处理异常

手写promise代码(相关核心代码)

```
/**
 * js中类的两种使用方法
 * 1 使用函数
 *  var ClassName = function() {
        this.message = 'dat.gui';
        this.speed = 0.8;
        this.displayOutline = false;
        this.explode = function() {};
        // Define render logic ...
    };
 * 2.使用类和方法
    class ClassName {
        constructor() {
            this.message = 'dat.gui';
            this.speed = 0.8;
            this.displayOutline = false;
        }
        explode() {
            // Define render logic ...
        };
    }
 */

//类可以看作构造函数的加强版本


 const PENDING = 1
 const FULLFILED = 2
 const REJECTED = 3

```

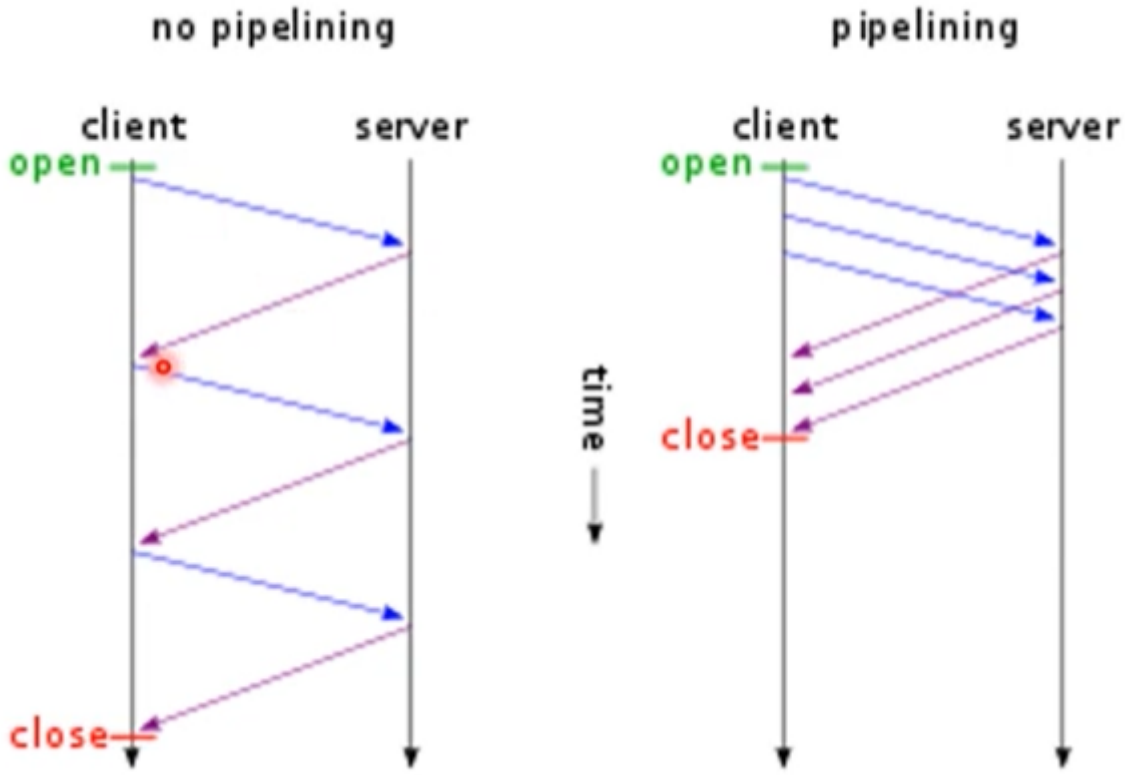
```
 //手撸Promise
class Promsie1{
 //Promise类
 //它的构造函数需要传入 executor
 constructor(executor){
   this.state =   PENDING
   //executor 立即被执行 executor函数里面两个参数  参数是两个函数
   this.fullfills = []
   const isObect = (obj)=>{
     return  Object.prototype.toString.call(obj) === '[object Object]'
  }
  const resolver = (value)=>{
      if(this.state === PENDING) {
        this.value = value
        this.state = FULLFILED
 //执行缓存数据中的回调，有多少执行多少
        for(let [onFullfill,resolve] of this.fullfills){
            const x = onFullfill(this.value)
            if(isObect(x)&&(x instanceof Promsie1)){
                x.then(res=>{
                    resolve(res)
                })
            }else{
                resolve(x)
            }

        }

      }

  }
  const rejecter = (reason)=>{
    if(this.state === PENDING) {
        this.reason = reason
        this.state = REJECTED
      }
  }
  executor(resolver,rejecter)
}
 //普通的then方法
```

```
 then(onFullfill){
    // 返回一个新的promise
    return new Promsie1((resolve,reject)=>{
        console.log(this.state=== 1?'PENDING':'FULLFILED')
        switch(this.state){//但是这个状态还是当前promise的实例状态
            //Pending状态
            case PENDING:
                this.fullfills.push([onFullfill,resolve]);
                //第一个Promise的状态还在PENDING,把一些回调函数和reslove都用fullfills存在起来,promise的状态变化时候
                //也就是第一个promise的状态变成FULLFILED，然后再去遍历fullfills。执行FULLFILED状态下的功能操作。
 //(执行回调，resolve数据,改变新的promise的状态)
                //这样一层一层 就实现了链式调用
                break
            //状态已经是FULLFILED的话
            case FULLFILED:
                //then中的onFullfill是一个函数，把onFullfill函数的返回值作为 then方法返回的Promise对象的value值。
                const x = onFullfill(this.value)
                resolve(x)
                break
        }
    } )
 }
}
const promise = new Promsie1((resolve)=>{
    setTimeout(()=>{
        resolve('第一个测试')
    },1000)
}).then(res=>{
     console.log(res)
 })


```

Fetch的基本用法

一个让处理http pipeline更容易的工具

http pipeline 一个在http1.1之前 client发送一个请求过去，server返回一个请求 然后客户端才能发第二个请求，服务端再回一个请求。这样的话时间浪费在哪里？服务端的负载不够，有点浪费，如果还有额外的cpu，额外的资源，完全可以去干其他的事情。这时候就是pipeling的场景了，客户端的一个请求一个请求的发出去，但是和http2.0的多路复用一起出去还是不一样，pipelining还是一个一个的发出去，也会产生队头阻塞的问题。但是都会把的请求发到服务端去，排队发送请求，而不是发一个等服务端回了之后，再发一个。这样服务端的资源能够很好的进行利用(前提是，服务端有足够多的资源)



- Fetch是用来发送请求的工具，在浏览器的window对象里面就有一个fetch方法
- window.fetch() 方法返回是Promise
- Resolve发生在网络通信正常(404,500也是resolve)
- Reject发生网络通信异常
- 默认不接收cookie

基本使用方法

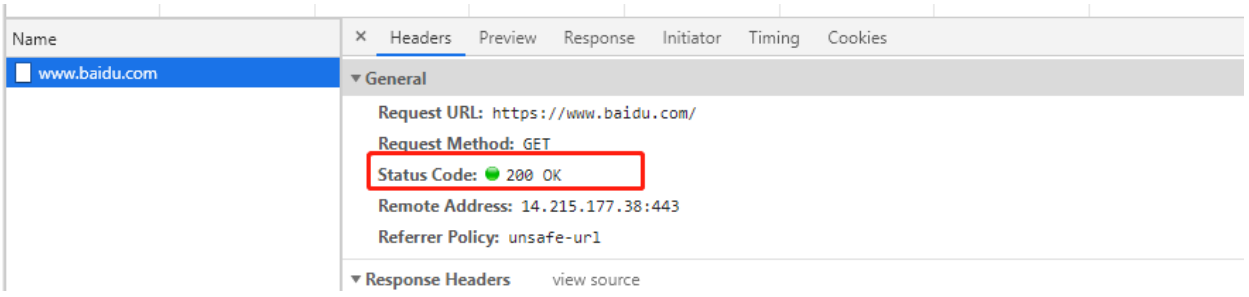
- GET/POST/PUT/DELETE
- headers
- Cookie
- 缓存

```
/**
 * fetch的基本用法
 */

const fetch = require('node-fetch')

async function foo(){
    const resp = await fetch('http://xwww.baidu.com',{
        method:'POST',
        headers:{
            'user-agent':'Mozilla',
  'Content-type':'application/json'
  ......
        }
    })
    const text = await resp.text()
    console.log(text)
}
foo()
```

缓存使用

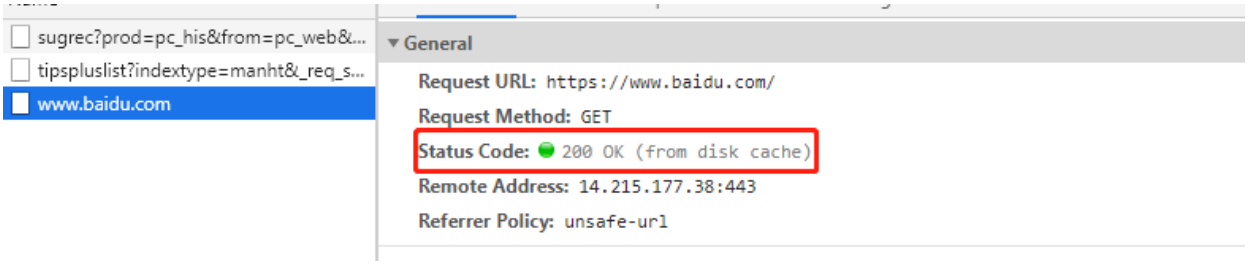
没有使用缓存的情况下，举例百度:

```
> fetch("https://www.baidu.com/")
< ▶ Promise {<pending>}
>
```



使用缓存的情况下

```
> fetch("https://www.baidu.com/",{cache:'force-cache'})
< ▶ Promise {<pending>}
```



Petch+Promise场景举例

1. **(实战) 应对不稳定的网络环境-指数补偿**

- 按照指数的时间倍数重复发送请求
  - 0ms
  - 200ms
  - 400ms
  - 800ms
  - 1600ms
  - 3200ms
  - fail

上述 在多少ms发送请求回来的话,后面的请求都不会再发了。如果超过的上述的3200ms后请求还没回来的话，网络就断开了，请求失败

```
//应对网络不稳定环境，进行网络补偿你
const fetch = require('node-fetch')
function request(url){
    let resolved = false
    let  t = 1
    return new Promise((resolve,reject)=>{
           function doFetch(){
               if(resolved||t>16){ //当网络请求回来后，或者请求指数大于16的时候，终止
                 console.log('请求终止')
                   return
                }else{
```

```
                    //请求
                    fetch(url).then(res=>{
                        return res.text()
                    }).then(res=>{
                        if(!resolved){
                            console.log('t的指数',t)
                            resolve(res)
                            resolved = true
                        }
                    }).catch(err=>{
                     reject(err)
                    })
                    //网络补偿，需要用到递归
                    console.log(t)
                    setTimeout(()=>{
                     doFetch()
                      t*=2
                    },t*100)
                }
            }
            //为0ms时候的请求
            doFetch()
    })
}

 const promise = request('https://github.com/')
 promise.then(res=>{
     console.log(res)
 })
```

结果如下:补偿到指定的指数后,请求终止。

```
16
(node:18388) UnhandledPromiseRejectionWarning: FetchError: request to https://github.com/ failed, reason: connect ETIMEDOUT 192.30.255.113:443
PS D:\project\projectTest\promise> node request.js
1
1
2
4
8
16
请求终止
(node:2308) UnhandledPromiseRejectionWarning: FetchError: request to https://github.com/ failed, reason: connect ETIMEDOUT 192.30.255.113:443
    at ClientRequest.<anonymous> (D:\project\projectTest\node_modules\node-fetch\lib\index.js:1461:11)
    at ClientRequest.emit (events.js:210:5)
    at TLSSocket.socketErrorListener (_http_client.js:406:9)
    at TLSSocket.emit (events.js:210:5)
    at emitErrorNT (internal/streams/destroy.js:92:8)
    at emitErrorAndCloseNT (internal/streams/destroy.js:60:3)
    at processTicksAndRejections (internal/process/task_queues.js:80:21)
(node:2308) UnhandledPromiseRejectionWarning: Unhandled promise rejection. This error originated either by throwing inside of an async function
h(). (rejection id: 1)
(node:2308) [DEP0018] DeprecationWarning: Unhandled promise rejections are deprecated. In the future, promise rejections that are not handled w
PS D:\project\projectTest\promise>
```

2.(实战)并发处理和时间窗口

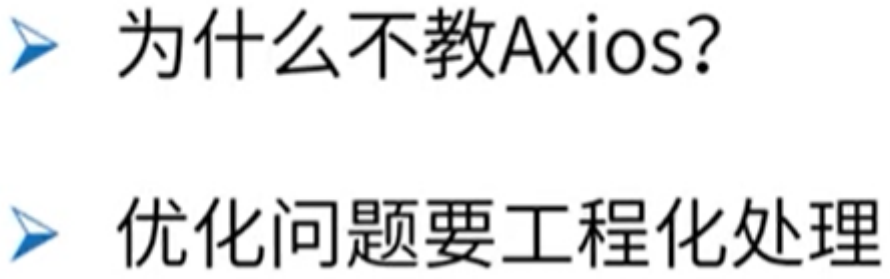
时间窗口：我们在发很多请求的时候，同时一下就发过去了，前端开发的东西有时候是高度组件化的，很多组件请求的地址是相同的，比如第一个组件请求的是商品列表 第五个组件请求的是相同的商品列表，这两个组件同时发出去，这样做就产生了两次西请求，这时候可以用时间窗口做处理。用到两个点

1. 多个资源并发请求
2. 基于时间窗口过滤重复请求

```
const fetch = require('node-fetch')


/**
 * 这个时间窗口,在这个时间内,相同的参数的请求，只执行一次
 * 这个需要用到高阶函数(所谓高阶函数,变量可以指向函数，函数的参数能接收变量,
 * 那么一个函数就可以接收另外一个函数作为参数，返回的肯定是一个函数，这种函数称之为高阶函数)
 * @param {*} f request函数
 * @param {*} time 时间间隔
 */


//需要用一个hash函数 把形参组织成一个唯一的key
function hash(...args){
   return args.join(',')
}
function window_it(f,time=50){
      let w = {} //这个时间窗口存储进来的请求
      let flag = false //一个标识类而已 判断是否有时间窗口
     return (...args)=>{
         return new Promise((resolve,reject)=>{
              // 存储相关请求
              if(!w[hash(args)]){
                 w[hash(args)] = {
                    func:f,
                    args,
                    resolvers:[]
                 }
              }
              //是否有创建时间窗口
```

```
                if(!flag){
                    flag = true
                     setTimeout(()=>{
                      //主要处理就是在这时间窗口的时间内 有多个相同的请求,
                      //处理函数只执行一次,但是可以reslove多次，这是我们的目标
                       Object.keys(w).forEach(key=>{
                           const {func,args,resolvers} = w[key]
                           console.log('run once')
                          func(...args)
                             .then(res=>res.text())
                             .then(text=>{
                              //请求一次  resolve 多次
                              resolvers.forEach(r=>{
                                  console.log('result respone')
                                    r(text)
                              })
                              //全部都成功后，初始化
                                flag =false
                                w = {}
                             })
                       })

                    },time)
                }
                w[hash(args)].resolvers.push(resolve)
            })
        }
}


const  request  = window_it(fetch,20) //返回的是一个新的函数 (...args)=>{}
request('https://www.baidu.com')
request('https://www.baidu.com')
request('https://www.baidu.com')
request('https://www.baidu.com')

```

课程小结



Fetch是标准所以没有使用AXIOS