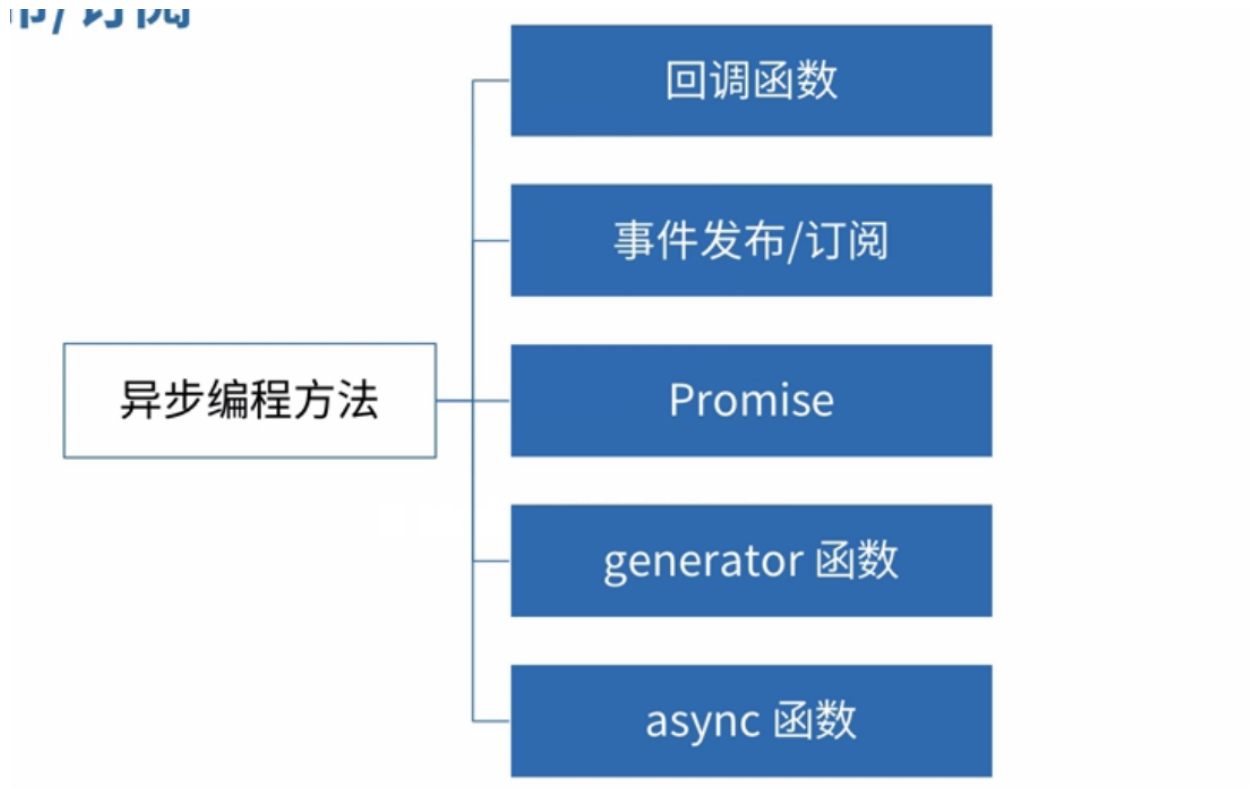


发布订阅

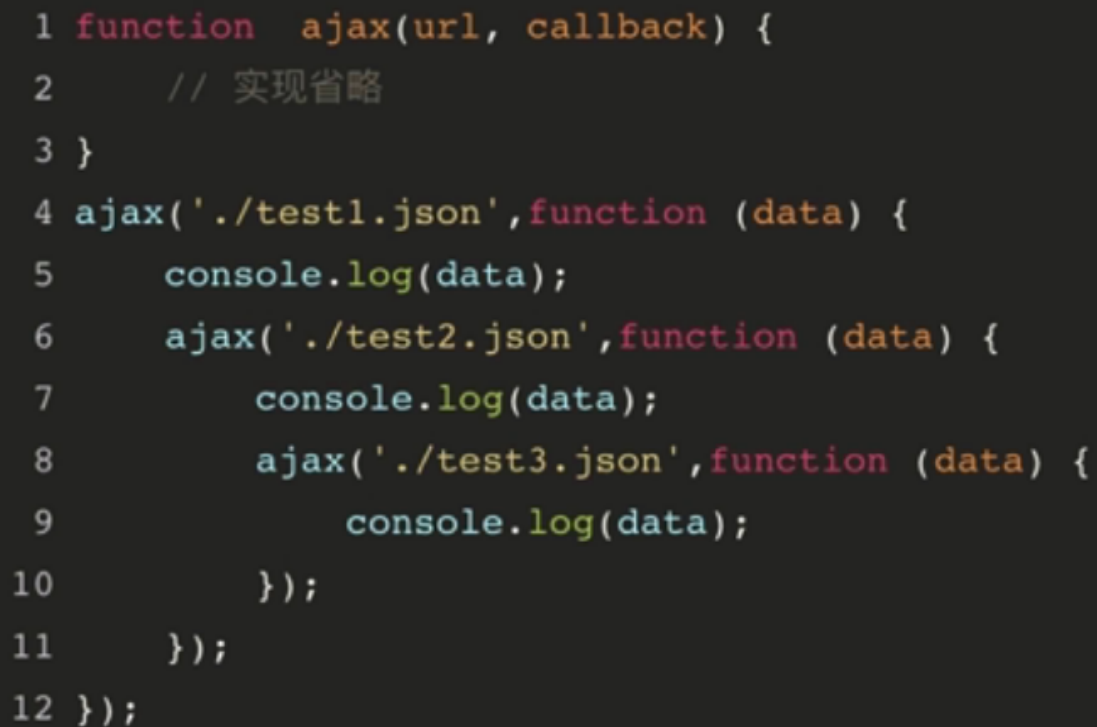
1. 发布订阅概念

发布订阅是什么? 发布订阅是处理消息的范式

在理解发布订阅前,我们总结一下常用的异步编程方法



我们在异步编程方法中 用的最多的就是 回调函数 Promise ,下面来讲讲事件发布订阅
先看下下面一段示例代码



```
1 function ajax(url, callback) {  
2     // 实现省略  
3 }  
4 ajax('./test1.json', function (data) {  
5     console.log(data);  
6     ajax('./test2.json', function (data) {  
7         console.log(data);  
8         ajax('./test3.json', function (data) {  
9             console.log(data);  
10        });  
11    });  
12 });
```

这段代码是封装一个ajax函数,然后再调用ajax函数,并在ajax回调函数里面调用ajax,以此下去嵌套了三层.这样的代码很垃圾,耦合度太高,一旦某一个回调出现问题,其他的回调都不会执行.

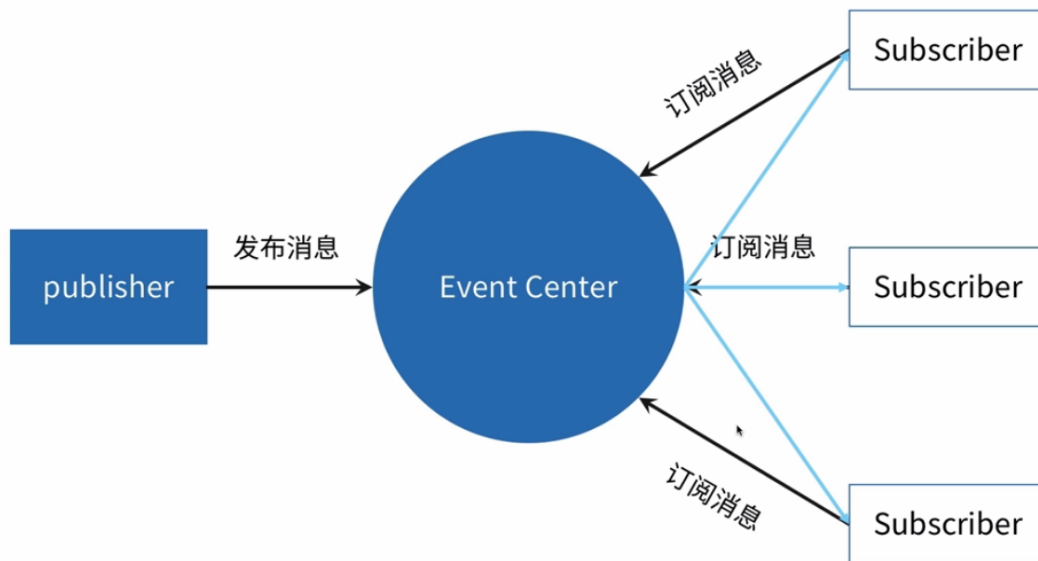
下面我们再看一段发布订阅的示例代码

```
1 const pbb = new PubSub();
2 ajax("./test1.json", function(data) {
3     pbb.publish("test1Success", data);
4 });
5 pbb.subscribe("test1Success", function(data) {
6     console.log(data);
7     ajax("./test2.json", function(data) {
8         pbb.publish("test2Success", data);
9     });
10 });
11 pbb.subscribe("test2Success", function(data) {
12     console.log(data);
13     ajax("./test3.json", function(data) {
14         pbb.publish("test3Success", data);
15     });
16 });
17 pbb.subscribe("test3Success", function(data) {
18     console.log(data);
19 });
```

这段代码 首先自定义了一个PubSub类,这个类实现了发布订阅,调用这个类后const pbb=new PubSub(),发送第一个请求 '/tex1.json' 然后在它的callbacks里面 调用pbb.publish发送一个test1Success事件,然后在外层通过pbb.subscribe订阅这个test1Success事件,然后对请求 '/tex1.json'的结果进行处理,然后再发送一个'/tex2.json'请求,然后在它的callbacks里面,再次调用pbb.publish 发送一个test2Success事件, 然后在外层通过pbb.subscribe订阅这个test2Success事件,然后对请求 '/tex2.json'的结果进行处理,,接着再发送一个'/tex3.json'请求,然后在它的callbacks里面,再次调用pbb.publish 发送一个test3Success事件, 然后在外层通过pbb.subscribe订阅这个test3Success事件,然后对请求 '/tex3.json'的结果进行处理.....

以上示例代码 就是通过发布订阅写一个顺序的ajax事件

发布订阅流程图



流程图过程

1. 首先有一个publisher对象(发布者),负责发布消息到事件中心 Event Center
2. 然后有个Subscriber对象(订阅者)去事件中心订阅消息,Subscriber可以有多个
3. 事件中心会向订阅者触发消息,订阅事件的处理函数就会执行.

2.实现事件发布订阅

```

1 class PubSub {
2     constructor() {
3         this.events = {};
4     }
5     publish(eventName, data) {
6         if(this.events[eventName]){
7             this.events[eventName].forEach(cb => {
8                 cb.apply(this, data)
9             });
10        }
11    }
12    subscribe(eventName, callback) {
13        if (this.events[eventName]) {
14            this.events[eventName].push(callback);
15        } else {
16            this.events[eventName] = [callback];
17        }
18    }
19    unsubscribe(eventName, callback) {
20        if (this.events[eventName]) {
21            this.events[eventName] = this.events[eventName].filter(
22                cb => cb !== callback
23            );
24        }
25    }
26 }

```

代码分析:

1. 首先自定义一个 PubSub的类,在其构造器constructor里面定义一个事件events的对象,来存储所有的事件,为什么events要是个对象呢?因为对象的key 和value 能够让事件的名称和事件的处理函数很好的对应起来,方便调用
2. 定义一个publish方法,在上一个示例代码中,在ajaxcallbacks里面调用了一个publish发布消息的方法,就是此方法.这里方法有两个形参,一个eventName和data,判断eventName是在事件中心是否存在,如果存在的话,就去取出事件对应处理函数,并执行相应的处理函数.注意事件对应的处理函数是一个数组,为什么要是一个数组呢?因为发布订阅事件,可以进行多次订阅,正好符合流程图上面说的.

3. subscribe订阅方法 ,这个方法两个形参,一个eventName和callback,先判断事件中心是否有该eventName,如果存在,说明之前已经被订阅过了,只需要把callpack存入该eventName对应的处理函数数组中,如果没有,就创建一个新的空处理函数数组,并把当前的callbackpush到数组中.

4. unsubscribe取消订阅,,这个方法也是有两个形参,一个eventName和callback,先判断事件中心是否存在此事件方法,如果不存在的话,就需要处理,如果存在的话,过滤的处理函数数组中callback即可

发布订阅有什么优点和缺点么?

发布订阅是一种处理消息的范式,我们将传统的callBack转成发布订阅,肯定有它独有优势,现在很多ajax事件都转化成了发布订阅模式.它松耦合,更加灵活,同时它有缺点 无法确保消息被触发,或者触发几次,同时也不太好维护,发布订阅在promise未出来前,它是异步编程的主要方法.

3.Node.js的发布订阅

预习资料

资料名称	链接	备注
EventEmitter实现源码	https://github.com/nodejs/node/blob/master/lib/events.js	建议看完视频阅读
FsWatch实现源码	https://github.com/nodejs/node/blob/master/lib/internal/fs/watchers.js	建议看完视频阅读

在看Node.js的发布订阅前,我们先看一下,node.js的callBack

方法设计默认异步，同步方法为readFileSync

回调函数置尾

```
1 const fs = require("fs");
2
3 fs.readFile("/Users/kitty/test1.json", "utf8", function(err, data) {
4   console.log(data.toString());
5 });
```

错误优先暴露

代码分析：

1. readFile 方法设计默认异步，同步方法为readFileSync
2. 回调函数置尾
3. 回调函数的第一个参数的err,第二个才是主流的数据

我们可以看到callback的把err当成一个参数，后面才是相应的数据，这种回调的设计模式，我们可以学习一下。一开始就处理err，后面才去处理相关data。

node.js发布订阅是通过什么来实现的？

我们知道node它分了很多个模块，其中有个模块叫做Event模块，EventEmitter是Event模块下的一个类下面我们来说一下eventEmitter是做什么的？

1. 是事件触发与事件监听功能的封装
2. 通过 `const { EventEmitter } = require('events')`
3. 产生事件的对象都是events.EventEmitter的实例（如http模块，它都是events.EventEmitter的实例）
4. 它是以继承方式使用（例如 http模块继承了EventEmitter）
5. EventEmitter有emit on addListener removeListener多种方法

下面我来看一下其中一种emit方法

```

EventEmitter.prototype.emit = function emit(type, ...args) {
  let doError = type === "error";
  const events = this._events;
  if (events !== undefined) {
    if (doError && events[kErrorMonitor] !== undefined)
      this.emit(kErrorMonitor, ...args);
    doError = doError && events.error === undefined;
  } else if (!doError) return false;

  if (doError) {
    let er;
    if (args.length > 0) er = args[0];
    if (er instanceof Error) {
      try {
        const capture = {};
        Error.captureStackTrace(capture, EventEmitter.prototype.emit);
        Object.defineProperty(er, kEnhanceStackBeforeInspector, {
          value: enhanceStackTrace.bind(this, er, capture),
          configurable: true
        });
      } catch {}
      throw er;
    }
    let stringifiedEr;
    const { inspect } = require("internal/util/inspect");
    try {
      stringifiedEr = inspect(er);
    } catch {
      stringifiedEr = er;
    }
    const err = new ERR_UNHANDLED_ERROR(stringifiedEr);
    err.context = er;
    throw err;
  }
}

```



```

const handler = events[type];
if (handler === undefined) return false;

if (typeof handler === "function") {
  const result = ReflectApply(handler, this, args);
  if (result !== undefined && result !== null) {
    addCatch(this, result, type, args);
  }
} else {
  const len = handler.length;
  const listeners = arrayClone(handler, len);
  for (let i = 0; i < len; ++i) {
    const result = ReflectApply(listeners[i], this, args);
    if (result !== undefined && result !== null) {
      addCatch(this, result, type, args);
    }
  }
}
return true;
};

```

代码执行解析：

1. type形参,...args所有的其他形参，这里大部分代码都是在容错。
2. 容错处理给我们自己设计一个库的该学习的一个方向。实现一个功能不能只它主流上的逻辑，这会出现很多意想不到的bug，所以 还得区分写大量的容错处理，保证功能健壮性
3. 下面代码就是代码错误异常相关处理
4. 对空的type参数处理

Node.js可以通过

emit方法对事件进行发布

on方法监听事件，并触发对应的回调

addListener 订阅消息 存入回调函数

removeListener 解除订阅 过滤该回调函数