设计模式实战

单例模式

开发中常遇到的单例模式

- 模块中的单例模式

```
import reducer from './reducer';
import {configureStore} from 'redux';

const store = configureStore({reducer});

export {store}

//  app-a.js
import {store} from './store'
// app-b.js
import {store} from './store'
```

上述代码是react中使用redux时常用的代码，其中a、b.js中的store是同一个实例。原因是在模块加载的时候store就已经生成了。

- 类的单例模式

```
class Eager {
  static instance = new Eager('eager')

  constructor(name) {
    console.log('Eageer constructor', name)
    this.name = name
  }
}

module.exports = { Eager }
```

上述代码是单例模式中的饿汉单例模式的实现，顾名思义是模块引入时单例就已经实例化完成了。

```
class Lazy {
  static instance = null;

  static getInstance() {
    if (!Lazy.instance) {
      Lazy.instace = new Lazy('lazy')
    }
    return Lazy.instance
  }

  constructor(name) {
    console.log('lazy constructor', name)
    this.name = name
  }
}

module.exports = { Lazy }
```

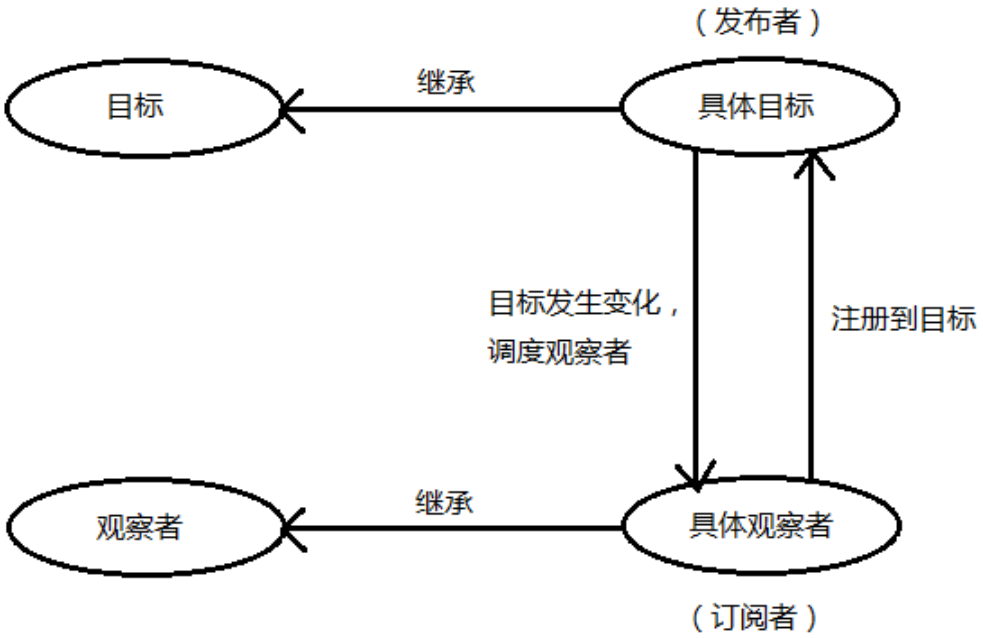
以上是单例模式的懒汉模式的实现，其中Lazy类的单例并不会在模块引入时自动初始化。需要用户手动调用getInstace来初始化单例，并且多次调用getInstance返回的都是同一个实例

```
class LodashLoader {
  static instance = null;

  static getInstance() {
    if (!LodashLoader.instance) {
      LodashLoader.instace = new LodashLoader
    }
    return LodashLoader.instance;
  }

  constructor() {

loadScript('https://cnd.jsdelivr.net/npm/lodash@4.17.15/lodash.min.js')
  }
}

function loadScript(url) {
  const $script = document.createElement('script')
  $script.src = url
  $script.onload = () => {
    console.log('loaded', url)
  }

  document.body.appendChild($scirpt)
}

window.LodashLoader = LodashLoader
```

以上是开发中对于loadsh使用单例模式加载的示例。

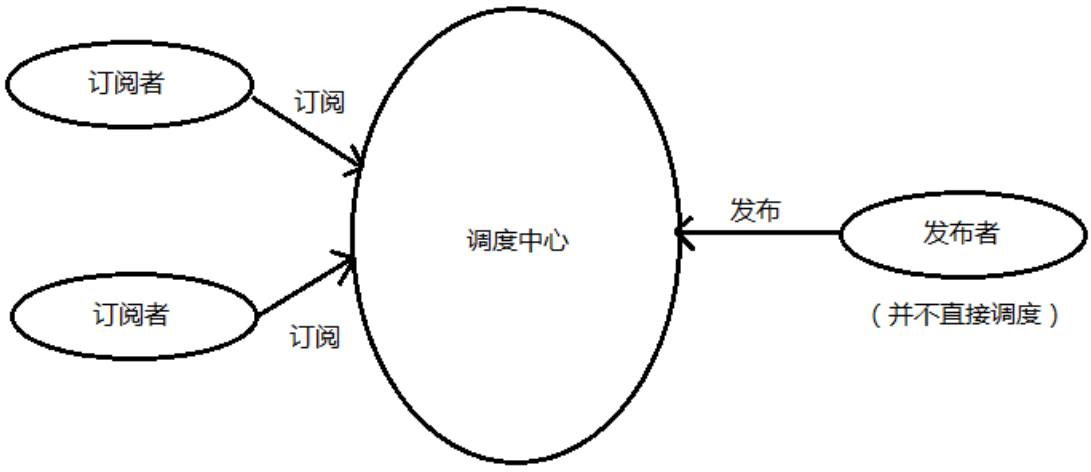
发布订阅模式、观察者模式

> 观察者模式中，目标和观察者是基类，目标提供维护观察者的一系列方法，观察者提供更新接口。具体观察者和具体目标继承各自的基类，然后具体观察者把自己注册到具体目标里，在具体目标发生变化时候，调度观察者的更新方法。



观察者模式

> 发布订阅模式中，订阅者把自己想订阅的事件注册到调度中心，当该事件触发时候，发布者发布该事件到调度中心（顺带上下文），由调度中心统一调度订阅者注册到调度中心的处理代码。



发布/订阅模式

实现一个基于观察者模式的EventEmitter

```
class EventEmitter {
  constructor() {
    this._events = {}
  }
  // 查看要监听的事件是否存在，不存在就初始化为空数组。存在直接push对应的回调
  on(name, cb) {
    if (!this._events[name]) {
      this._events[name] = []
    }
    this._events[name].push(cb)
  }

  // 将对应事件名的回调数组依次执行一遍
  emit(name, ...args) {
    if (!this._events[name]) return
    for (const fn of this._events[name]) {
      fn.apply(null, args)
    }
  }

  // 如果监听的事件不存在，直接返回；如果存在则找到数组中的回调并且移除
  off(name, cb) {
    if (!this._events[name]) return
    const index = this._events[name].findIndex(evt => evt === cb)
    if (index >= 0) {
      this._events[name].splice(index, 1)
    }
  }
}
```

如何使用EventEmitter类呢?

- on 为指定事件注册一个监听器，接受一个字符串 event 和一个回调函数。
- emit 按监听器的顺序执行执行每个监听器
- off 移除指定事件的某个监听回调

```
const { EventEmitter } = require('./events.js')
const eventEmitter = new EventEmitter();

eventEmitter.on('data', (value) => {
  console.log('on data', value)
})

const callback = () => {
  console.log('cb')
}

eventEmitter.on('data', cb)

eventEmitter.emit('data', 'hello')

eventEmitter.off('data', cb)

eventEmitter.emit('data', 'hey')
```

发布订阅模式的基本实现

```
class Observable {
  constructor(subscriber) {
    this._subscriber = subscriber;
  }

  subscribe(observer) {
    if ('object' !== typeof observer || observer === null) {
      observer = {
        next: observer
      }
    }

    return new Subscription(observer, this._subscriber)
  }
}

class Subscription {
  constructor(observer, subscriber) {
    this._observer = observer;
    const subscriptionObserver = new subscriptionObserver()
    subscriber.call(null.subscriptionObserver);
  }
}

class SubscriptionObserver {
  constructor(subscription) {
    this._subscription = subscription
  }
  next(value) {
    notify(this._subscription, 'next', value)
  }
}

function notify(subscription, type, ...args) {
  if (subscription._observer[type]) {
    subscription._observer[type].apply(null, args)
  }
}
```

vue 中的发布订阅模式

```
observe(value) {
   if(!value || typeof value !== 'object'){
     return
   }
   // 遍历该对象
```

```
    Object.keys(value).forEach(key => {
      this.defineReactive(value, key, value[key])
      // 代理data的中属性到vue实例上
      this.proxyData(key)
    })
  }

  defineReactive(obj, key, val){
    this.observe(val);  // 解决数据嵌套：递归

    const dep = new Dep();

    Object.defineProperty(obj, key, {
     get: function(){
       return val;
     },
     set: function(newVal) {
       if(val === newVal){
         return
       }
       val = newVal;
     }
    })
  }

  proxyData(key) {       //  执行一个代理proxy。这样我们就把data上面的属性代理到了vm实例上。
    Object.defineProperty(this, key, {
      get(){
        return this.$data[key];
      },
      set(newVal){
        this.$data[key] = newVal
      }
    })
  }

 class Watcher {
  constructor(vm, key, cb) {
    this.vm = vm;
    this.key = key;
    this.cb = cb;

    // 在这里将观察者本身赋值给全局的target，只有被target标记过的才会进行依赖收集
    Dep.target = this;
    // 触发getter,添加依赖
    this.vm[this.key];
    Dep.target = null
  }
```

```
  update() {
    //  将回调函数代理到this.vm实例，并传入对应属性的value值
    this.cb.call(this.vm, this.vm[this.key]);
  }
}

class Dep {
  constructor() {
   this.deps = [];
  }
   addDep(dep) {
     this.deps.push(dep)
   }
   notify() {
     this.deps.forEach(dep => {
       dep.update()
     });
   }
}

defineReactive(obj, key, val){
    this.observe(val);  // 解决数据嵌套：递归

    const dep = new Dep();

    Object.defineProperty(obj, key, {
     get: function(){
       /*Watcher对象存在全局的Dep.target中，  只有被target标记过的才会进行依赖收集*/
       Dep.target && dep.addDep(Dep.target)
       return val;
     },
     set: function(newVal){
       if(val === newVal){
         return
       }
       val = newVal;
       /*只有之前addSub中的函数才会触发*/
       dep.notify();
     }
    })
  }
```

代理模式

- 职责清晰 真实的角色就是实现实际的业务逻辑,不用关心其他非本职责的事务,通过后期的代理完成一件完成事务,附带的结果就是编程简洁清晰。
- 代理对象可以在客户端和目标对象之间起到中介的作用,这样起到了中介的作用和保护了目标对象的作用。
- 高扩展性

代理模式的基本实现

```
const { Question } = require('./request')

let totalCount = 0;
const question = new Question();

const proxyQuestion = new Proxy(question, {
  get: function(target, key, receiver) {
    console.log('fetching...', totalCount)
    return Reflect.get(target, key, receiver)
  }
})

main();

async function main() {
  await proxyQuestion.all();
  await proxyQuestion.all();
  await proxyQuestion.all();
  console.log('totalCount', totalCount)
}
// 通过代理模式，我们将代码很好的解耦。有着很高的拓展性，此处通过封装了一层proxyQuestion,在不改动Question模块的前提下新增了很多功能
```

装饰器

> 通过装饰器可以在不修改类的前提下为类新增功能，并且可以在装饰器中做很多其他操作

以下是一个类似于装饰器思想的的函数

```
const decorator = (obj) => {
  obj.send = function(method, ...args) {
    if (!this[method]) {
      return this.methodMissing.apply(this, [method, ...args])
    }
  }
  obj.methodMissing = obj.methodMissing || function(..args) {
    console.log(...args)
  }
  return obj
}

module.exports = { decorator }
```

ES6中decorator中的应用

```
import { Context } from 'koa';
import * as assert from 'assert';
import * as Router from 'koa-router';

type Middleware = Router.IMiddleware;

export enum RequestMethod {
  GET = 'get',
  POST = 'post',
  DELETE = 'delete',
  ALL = 'all',
  PUT = 'put',
  HEAD = 'head',
  PATCH = 'patch',
}

// tslint:disable-next-line:no-any
const methodList = Object.keys(RequestMethod).map((k: any) =>
RequestMethod[k]);

type Method = 'get' | 'post' | 'put' | 'delete' | 'all' | 'head' |
'patch';

const rootRouter = new Router();

export function route(url: string | string[],
                      method?: Method,
                      // tslint:disable-next-line:no-any
                      middlewares: Middleware[] | Middleware = []): any
{
```

```
  // tslint:disable-next-line:no-any
  return (target: any, name: string, descriptor?: any) => {

    const midws = Array.isArray(middlewares) ? middlewares :
[middlewares];

    /**
     * 装饰类
     */
    if (typeof target === 'function' && name === undefined  &&
descriptor === undefined) {
      assert(!method, '@route 装饰Class时，不能有method 参数' );

      /**
       * 我们将router绑定在 原型上，方便访问
       */
      if (!target.prototype.router) {
        target.prototype.router = new Router();
      }
      /**
       * 仅仅设置Controller 前缀
       */
      target.prototype.router.prefix(url);

      /**
       * 使得当前Controller 可以执行一些公共的中间件
       */
      if (middlewares.length > 0) {
        target.prototype.router.use(...midws);
      }
      return;
    }

    /**
     * 装饰方法
     */
    if (!target.router) {
      target.router = new Router();
    }

    if (!method) {
      method = 'get';
    }

    assert(!!target.router[method], `第二个参数只能是如下值之一
${methodList}`);
    assert(typeof target[name] === 'function', `@route 只能装饰Class 或者
方法`);
```

```
    /**
     * 使用router
     */
    target.router[method](url, ...midws, async (ctx: Context, next:
Function) => {
      /**
       * 执行原型方法
       */
      const result = await descriptor.value(ctx, next);
      ctx.body = ctx.body || result;
    });

    /**
     * 将所有被装饰的路由挂载到rootRouter，为了暴露出去给 koa 使用
     */
    rootRouter.use(target.router.routes());
  };
}

// koa中使用方法如下去简化路由的书写
import { route } from '@server/decorator/router';
@route('/api/monitor')
export default class {
  @route('/alive')
  monitor() {
    return {
      data: true,
      message: '成功'
    };
  }
}
```