FEE-Development-guildline-points

JS Basic

function

- If we define two functions with same name, the last function will overwrite the prior functions.

```
var val1 = 1;
function add(pVal){
    alert(pVal + 1);
}

function add(pVal1,pVal2){
    alert(pVal1 + 2);
}

add(val1);//3
```

- You can use javascript's built-in variable *arguments* to get variable number parameters when you pass more parameters than the definition parameters of the function.

```
function showNames(pName){
    alert(pName);//Tom
    for (var i =0; i < arguments.length; i ++){
        alert(arguments[i]);
    }
}
showNames("Tom","Eric","Jim");
```

- The function will return undefined if the function doesn't have return value.

```
function defaultReturnValue(){

}
alert(defaultReturnValue());//undefined
```

- Variable anonymous function can help us to avoid function name pollute.

```
var anonymousFunc = function (pName){
    console.log("Hello " + pName);
}
anonymousFunc("Eric");
```

- We can use non-name anonymous function if we want the function to execute only once.

```
(function(pName){
    console.log("Hello " + pName);
})("Eric");
```

- *Closure function* need meet below conditions:
    1. FuncB is declared in FuncA
    2. FuncB refers to variable declared out of FuncA.
    3. FuncA return the reference of FuncB.
- When we declare a global variable to refer to the closure function, then the variables refered by the closure function will not be released unless we switch to other page or close the browser.
- The purpose of closure function is to ensure the safety of the variables declared in FuncA because we only can change variable of FuncA via closure function.

```
function funcA(){
    var count = 0;
    function funcB(){
        count ++;
        console.log(count);
    }
    return funcB;
}

var globalVar = funcA();//globalVar is a global variable

function partShowA(){
    var localVar = funcA();
    localVar();
}

globalVar();//1
globalVar();//2
globalVar();//3
globalVar();//4

partShowA();//1
partShowA();//1
partShowA();//1
```

call() and apply()

- The action of the two functions are the same: change a function's runtime context (the function's *this* object points to which object) and then execute it.

```
function fruits() {}

fruits.prototype = {
    color:"red",
    say:function(){
        console.log("The fruit color is " + this.color);
    }
}

var apple = new fruits();
apple.say();//The fruit color is red

function banana() {}
banana.prototype = {
    color:"yellow"
}

var ban = new banana();
apple.say.call(ban);//The fruit color is yellow
apple.say.apply(ban);//The fruit color is yellow
```

- The only difference between call() and apply() is the argument form :

```
func.call(obj,arg1,arg2); // call() accepts exactly arguments
func.apply(obj,[arg1,arg2])// apply() accept an array as argument.
```

- The second and other arguments will be passed to the original function:

```
 var arr1 = [1,2,3];
 var arr2 = [4,5];
 Array.prototype.push.apply(arr1,arr2);//1,2,3,4,5
```

- The usage of the built-in *arguments* property of current function:

```
function logto(){
     console.log.apply(console,arguments);//arguments represents actual parameter.
}
 logto("one","two");
```

func1.bind(thisObj,arg1...argn)

- Create a new binding function based on func1, and take thisObj as its *this* object, the after arguments will be passed to func1 during execution.

```
var func1 = function (){
    console.log(this.txt);
}

var func2 = {
    txt:"Eric"
}

var newFunc = func1.bind(func2);
newFunc();//Eric
```

Custom object

- Type 1: object literal

```
var obj = {
    name:"Eric",
    age:111
};
console.log(obj.constructor.prototype);//Object
console.log(obj.constructor.prototype === Object.prototype);//true
```

- Type 2: new constructor

```
function People(pName){
    this.name = pName;
}
var p = new People("Eric");
console.log(p.constructor.prototype);//constructor:function People(pName)
console.log(p.constructor.prototype === People.prototype);//true
console.log(p.name);//Eric

//multiple inheritance
function Student(pAge){
    this.age = pAge;
}
Student.prototype = new People();

var stu = new Student(111);
console.log(stu.constructor);//People constructor function
console.log(stu.constructor.prototype);//prototype object is People
console.log(stu.constructor.prototype === People.prototype);// true
```

- Object.create(prototype,propertyDescriptor)

```
//prototype is null
```

```
var obj1 = Object.create(null,{
    address:{
        city:"Xi'an",
        street:"18th"
    }
});
console.log(obj1.constructor);//undefined

//prototype is Array
var arr = Object.create(Array.prototype,{});
console.log(arr.constructor);//function Array
console.log(arr.constructor.prototype);//Array

//prototype is Custom Object
function Obj2() {}
var o1 = Object.create(Obj2.prototype,{});
```

prototype

```
function Base(){
    this.say = function() {
        console.log("Base.instance.say()");
    }
    this.sayInBase = function() {
        console.log("Base.instance.sayInBase()");
    }
}
function Extend(){
    this.say = function(){
        console.log("Extend.instance.say()");
    }
    this.hello = function(){
        console.log("Extend.instance.hello()");
    }
}
//this means it will clone all properties and functions of Base , then append them
to Extend as Extend's members
//so the instance of Extend can invoke functions of Base.
Extend.prototype = new Base();

var extendObj = new Extend();
extendObj.sayInBase();//Base.instance.sayInBase()
//The invoke order: instance function > prototype function, so below will first
invoke Extend's instance function.
extendObj.say();//Extend.instance.say()
extendObj.hello();//Extend.instance.hello()
```

Serialize object via JSON functions

- JSON.stringify(object): convert object to string

```
var obj = {
    x:1,
    y:2
}
console.log(JSON.stringify(obj)); //{"x":1,"y":2}
```

- JSON.parse(jsonStr): convert json string to object.

```
var jsonStr = '{"name":"Eric","age":"111"}';
var obj = JSON.parse(jsonStr);
console.log(obj.name);
```

Object oriented in javascript

- *this* point to current object instance in the constructor and functions of the object.

```
var obj = {
    name:"Eric",
    say:function(){
        console.log(this.name);//Eric
    }
}
obj.say();
```

- We can add new members to object in constructor and object literal via *this*.

```
function People(pName){
    this.name = pName;
}
var p = new People("Eric");
console.log(p.name);
```

- Add new members to object via Class's prototype

```
function People(pName){
    this.name = pName;
}
var p = new People("Eric");
People.prototype.say = function(){
    alert("Hello " + this.name);
}
p.say();
```

- Add static member on Class name directly

```
function People(pName){
    this.name = pName;
}

People.hasName = function(p){
    if(p.name && p.name.length >0){
        return true;
    }
    return false;
}
var p = new People("Eirc");
console.log(People.hasName(p));//hasName is a static method
```

Inheritance

- Once we add member to Class's prototype, all the instances of the class will have the member.

```
function People(pName){
    this.name = pName;
}

var p1 = new People("Tom");
var p2 = new People("Eric");
People.prototype.say = function(){
    alert("Hello " + this.name);
}

p1.say();
p2.say();
```

- Once we make a class's prototype point to another object , then the class's instance will have all the instance memebers of the object except for static members.

```
function People(pName){
    this.name = pName;
    this.say = function(){
        alert("Hello " + this.name);
    }
}
```

```
function Student(pAge){
    this.age = pAge;
}

Student.prototype = new People();

var stu1 = new Student(111);
stu1.name = "Eric";
stu1.say();
```

APIs

- Fetch API , you can refer to FetchAPIDoc
- It's used to replace XMLHttpRequest, it provides a fetch() function and below objecs:
    - Request object: var req = new Request(reqUrl, {method:'GET', cache:'reload'}); Request object can be nested : var postReq = new Request(req, {method:'POST'});
    - Headers object: var hd = new Headers(); hd.append('Accept', 'application/json'); or like this: var hd = new Headers({"Content-Type":"application/json","Cache-Control":"max-age=3600" });
    - Response object: var res = new Response(JSON.stringify(jsonString), {status:200,headers:theHeaders});
- A sample:

```
var URL = 'https://api.corp.com/getOrders';
function fetchDemo() {
        //fetch() function return a promise object, so we can use promise's api.
        fetch(URL).then(function(response) {
                return response.json();
        }).then(function(json) {
                insertPhotos(json);
        });
}
fetchDemo();
```

Nodejs

- Execute a node js :node demo.js or node demo
- We can enter REPL(read-eval-print-loop) environment to execute js scripts:node or node --use_strict
- Almost all of node operation is asynchronous , so node use callback function to handle this.
    - Callback function should be the caller function's last parameter.
    - Callback function's first parameter is error object passed by caller function,the subsequent parameter can be data object passed from outer.

```
function getUserInfo(){
    try{
        db.User.get(userId,function(err,dataParam){
            if(err){
                throw err;
            }
        });
    }
    catch(e){
        console.log("Error happen!");
    }
}
```

Global object and variable

- Global object
  - global: node's global object,but you should know that it's different from browser's window object.
  - process: node's current process.
  - console: node's built-in module,provide standard input & output.
- Global function
  - setTimeout() : timer, it will return an integer to represent timer's number.
  - clearTimeout() : terminate a timer defined by setTimeout()
  - setInterval()
  - clearInterval()
  - require() : load a module.
  - buffer(): to handle binary data.
- Global variable
  - __filename : current execution script's file name.
  - __dirname: the directory in which current execution script.
- The below variable can be used in all modules but they are not really global variable.
  - module
  - module.exports
  - exports

Module

- Node is based on module structure and it's based on CommonJS specification.
- Module and File is one to one relation,one module corresponds to a file.
- We use require() to load a module, once a module is loaded, it will be cached by node, so a module will execute only once.
  - Load custom module: `var cart = require("./cart.js");` or `var cart = require("./cart.js");`
  - If we don't specify module's detail path ,node will try to load node in module's install directory(usually is node_modules.) like this: `var cart = require("cart");`
  - When a module is a directory, node will try to load module according to *main* property in package.json.

```
{
    name:"shop",
    main:"./lib/shop.js"
}
```

then var shop = require("shop"); if we don't define *main* property, node will try to load index.js or index.node in the module's directory.

Core module

- The core modules is built-in , so you don't need to install them.
- The core modules is loaded in first priority, node will load built-in module even if you define a same name module with core module.
- Core module list
    1. http : provide http server functions.
    2. url : parse URL
    3. fs: file system related functions.
    4. querystring : parse URL query string.
    5. child_process: create a new sub process.
    6. util : provide useful functions.
    7. path: handle file path functions.
    8. crypto : wrapper for OpenSSL to provide encrypt and decrypt functions.

Custom module

- *module* is a top-level variable, its *exports* property is used to output interface to outer, we can output variables,functions,objects ... to outer.
- Define a module to output a function

```
//txt.js
module.exports = function(pTxt){
    console.log("module get a text :" + pTxt);
};

//index.js
var showTxt = require("./txt");
showTxt("test");
```

- Define a module to output an object

```
//demo.js
var demo = {
    name:"demo app",
    display:function(pInfo){
        console.log(this.name + " : " + pInfo);
    }
}
module.exports = demo;
//index.js
var demo = require("./demo");
demo.display("Eric");//demo app : Eric
```

Exception Handler

- Node is running on single thread, so the whole process will crash once the exception is not handled.
- Callback function to handle the passed error object

```
var fs = require("fs");
fs.readFile("./demo.js",function(err,data){
    if(err !== null){
        throw err;
    }
    console.log(data);
});
```

- EventEmitter can throw error event when an exception happens.

```
var EventEmit = require("events").EventEmitter;
var emitter = new EventEmit();

//we must add below event handler function otherwise the whole process will crash
once error happens
emitter.on("error",function(err){
    console.error("Error: " + err.message);
});

emitter.emit("error",new Error("Oh shit, error coming.."));
```

- We can add callback function to handle uncaughtException which will be throwed when an exception is not catched.

```
process.on("uncaughtException",function(err){
    console.error("uncaughtException event : " + err);
});

try {
    setTimeout(function(){
        throw new Error("this is an error.");
    },1);
} catch (error) {
    //can not catch it because setTimeout's callback function is an asynchronous
function '
    console.log(error);
}
```

Nodejs modules

object-assign

- syntax: objectAssign(target,source,[source1,source2...])
- description: assign all properties of source object to target, the additional source objects will overwrite previous source object.
- Install npm install --save-dev object-assign
- usage

```
const objectAssign = require("object-assign");
objectAssign({name:"Eric"}, {age:111}); // will return {name:"Eric",age:111}
objectAssign({name:"Eric"}, {age:111}, {city:"Xian"}); // will return
{name:"Eric",age:111,city:"Xian"}
//overwrites equal keys
objectAssign({count:1}, {count:2}, {count:3});//{count:3}
//ignore null and undefined sources
objectAssign({count:1},null, {num:2},undefined);// {count:1,num:2}
```

gulp : A FEE automation development tool which provide compress, build .....

- Gulp is based on nodejs, so we must ensure node environment is ready before install gulp.
- Install gulp: npm install gulp -g , then check gulp version: gulp -v
- Install gulp in current directory: npm install gulp
- gulp need a config file : gulpfile.js, gulp will execute specific tasks according to this file.
- gulp-uglify : a node module used to compress js file.
- gulp-watch-path: a node module used to re-compress js file which changed since last compress.
- gulp-rename: a node module used to rename the compressed file.
- Let's play a demo:
    1. create a folder *gulp-demo*
    2. create a subfolder *script* under gulp-demo
    3. open command line tool and then enter the folder.
    4. install required modules: npm install gulp gulp-uglify gulp-watch-path gulp-rename
    5. create a gulpfile.js under *gulp-demo*.

```
//gulpfile.js
var gulp = require("gulp");
var uglify = require("gulp-uglify");
var watchPath = require("gulp-watch-path");
var rename = require("gulp-rename");
```

```
gulp.task("compressTaskName",function(){
    gulp.src("script/*.js")
        .pipe(uglify())
        .pipe(rename({suffix:'.min'})) //add .min suffix name on the compressed
files
        .pipe(gulp.dest("compressedDir"));
});

gulp.task("auto",function(){
    gulp.watch("script/*.js",function(event){
        var paths = watchPath(event,'script','compressedDir');
        gulp.src(paths.srcPath)
            .pipe(uglify())
            .pipe(rename({suffix:'.min'}))
            .pipe(gulp.dest(paths.distDir))
    });
});
//the string "default" can not be modified, if we run gulp, it will run auto task
default.
gulp.task("default",['auto']);
```

```
1. create a test.js under *script*
```

```
// /script/test.js
function say(){
    console.log("Say hellow world11111222.");
}
```

```
1. Open command line tool on folder *gulp-demo*, run `gulp compressTaskName`  to
compress js first.
```

Then run `gulp` to start gulp to watch the files changes, now you can modify the test.js and watch the compressed files's change!

Webpack

- It is a whole solution tool for FEE , it can tranform, build, pack .....
- Webpack has two morst important concepts: loader and plugin. Various loaders can enhance webpack's function so that webpack can do more things.
- webpack need an entry file, and then it will handle all the dependencies by using loaders.
- webpack treats every file as a module, it finally pack the whole project to bundle.js which can run in browser.

Usual commands

- Install webpack: npm install --save-dev webpack
- Create package.json: npm init
- Install typings : this will provide intellisense for corresponding modules.
    - npm install typings --global
    - Install typings config files
        - typings install --save --global dt~node
        - typings install --save --global dt~react
- Uninstall tsd : tsd is another intellisense which has be replaced by typings.
    - npm rm -g tsd
- Install json-loader for webpack npm install --save-dev --global json-loader
- Install babel npm install --save-dev babel-core babel-loader babel-preset-es2015 babel-preset-react
- Install react npm install react react-dom
- Install CSS loader for webpack npm install --save-dev style-loader css-loader

webpack.config.js

```
module.exports = {
    devtool:"eval-source-map",
    entry: __dirname + "/app/main.js",
    output:{
        path:__dirname + "/public",
        filename:"bundle.js"
    },
    module:{
        loaders:[
            {
                test:/\.json$/,
                loader:"json"
            },
            {
                test:/\.js$/,
                exclude:/mode_modules/,
                loader:"babel"
            },
            {
                test:/\.css$/,
                loader:"style!css?modules"
            }
        ]
    }
}
```

.babelrc

```
{
    "presets":["es2015","react"]
}
```

webpack plugin

- html-webpack-plugin
- html-webpack-plugin official site

ES6

babel

- Description: A compiler which can transfer jsx and es2015 to normal js code which can be executed in browser.

babel-cli

- Install global `npm install --global babel-cli`
- compile single file `babel test.js babel test.js -o compiled.js` or `babel test.js --out-file compiled.js`
- compile whole directory `babel src --out-dir lib` or `babel src -d lib -s` //-s generate source map files.

- Install in project `npm install --save-dev babel-cli`
- Then add below code in "scripts" field of package.json `"build": "babel src -d lib"`
- Then run below command to transcode: `npm run build`

babel-node

- babel-node is installed with babel-cli, so you don't need to install babel-node separately if you have installed babel-cli.
- enter babel-node environment `babel-node`
- run es6 script using babel-node directely `babel-node es6.js`

babel-register

- Install: `npm install --save-dev babel-register`
- You should load babel-register firstly if you use it, like below:

```
require("babel-register");
require("./index.js");
```

- Transcode .js,.jsx,.es,.es6 files real time, when you use require to refer to a js file, it will transcode it first.
- It only transcode files which is refered by require function, and it doesn't transcode current file.So it is usually used in dev env.

babel-core

- It provides babel api to transcode.
- Install npm install --save babel-core
- Transcode by using babel api

```
var babel = require("babel-core");
//transform string
babel.transform("code();",options);

//transform file
babel.transformFile("file.js",options, function(err,result){
});
//transform file sync
babel.transformFileSync("file.js",options, function(err,result){
});
//babel ast transform
babel.transformFromAst(ast,code, options);
```

- For options,pls refer to http://babeljs.io/docs/usage/options/

babel-polyfill

- This emulates a full ES6 environment.
- babel only tranform new js syntax rather than new api, so we need babel-polyfill if we want the new api to run.

babel online transform tool

- It can transform es6,react .... to normal js code real time.
  https://babeljs.io/repl/#?
  babili=false&evaluate=true&lineWrap=false&presets=es2015%2Creact%2Cstage-2&code=

babel cooperates with other tools

- ESLint
  - A code check tool which check the syntac and style of code.
  - Install npm install --save-dev eslint babel-eslint
  - create .eslintrc in the root directory of your project like below:

```
{
"parser":"babel-eslint",
"rules":{}
}
```

```
- add below code in package.json
```

```
"scripts":{
"lint":"eslint my-files.js"
}
```

- Mocha
    - A unit test framework, you can modify package.json below to run es6 unit test code:

```
"scripts":
{
"test":"mocha --ui qunit --compilers js:babel-core/register"
}
```

---

let and const and top level object

- let : declare a local variable
- const : declare a const variable
- top level object's property
    - if use var , function to declare variable, then it belongs to top level object.
    - if use let, const, class to declare variable, then it doesn't belong to top level object.
- top level object
    - Top level object is different in different environment, in browser , it's window; in node, it's global
    - we can use system.global to emulate that we can get global in any environment:

```
//commonjs syntax
require("system.global/shim")();
//es6 syntax
import shim from "system.global/shim";shim();
const g = getGlobal()
```

Variable Destructuring

- Swap variables `[x,y] = [y,z]`
- Return multiple value from function

```
//return an array
function returnArr(){
return [1,2,3];
}
let [a,b,c] = returnArr();
```

```
//return an object
function returnObj(){
return {
    name:"eric",
    age:111
    }
}
let {name:yourName,age:yourAge} = returnObj();
console.log(yourName + "===" + yourAge);
```

- Use multiple function parameters

```
function arrParam([x,y,z]){
    let result = x + y + z;
}
arrParam([1,2,3]);

function objParam({x,y,z}){
....
}
objParam({x:1,y:5,z:7});
```

- Retrieving json data

```
var jsonData = {
    id:100,
    status:"ok",
    data:[10,90]
};
let {id,status,data:number} = jsonData;
console.log(id + " ," + status + " ," + number);
```

- Parameter default value of function

```
jquery.ajax = function(url, {
async:true,
cache:true,
...
});
```

- Iterate map value

```
var mapVal = new Map();
mapVal.set("key1","val1");
mapVal.set("key2","val2");
```

```
for(let [key,value] of mapVal){
console.log(key + " ," + value);
}
```

- Import specific method of module const {SourceMapConsumer,SourceNode} = require("source-map")

String extention

- Unicode : \u30887 , \u{41}\u{42}\u{43}
- codePointAt()

```
let str = '吉a';
str.codePointAt(0);
str.codePointAt(1);
str.codePointAt(2);
```

- String.fromCodePoint() String.fromCodePoint(0x20887)
- String iterate

```
for(let codePoint of 'str'){
console.log(codePoint);
}
```

- at() : 'aaa'.at(0);
- normalize()
- includes(),startsWith(),endsWith()

```
let str = "hello world!";
str.startsWith("world",6);//true, 6 means match from index 6, the 7th char.
str.endsWith("hello",5);//true, 5 means the first 5 chars of the string.
str.include("hello",6);//false
```

- repeat()

```
"x".repeat(3);// "xxx"
"hello".repeat(2);//"hellohello"
"no".repeat(0);//""
```

- padStart(),padEnd()

```
"x".padStart(4,"ab");//"abax"
"x".padEnd(4,"ab");//"xaba"
```

- Template String, it use backquote to represents a template string

```
function auth(user,action){
console.log(`User ${user.name} is
doing ${user.action}`);
}
```

- String.raw(): It's usually used to handle template string

Number extention

- Number.isFinite() ,Number.isNaN()
- Number.parseInt(), Number.parseFloat()
- Number.isInteger(), Number.isInteger(3);//true Number.isInteger(3.0) //true
- Number.EPSILON
- Number.isSafeInteger() check whether a number is in the integer range.
- Math.trunc(): trunc decimal, and let the integer partiton left.
- Math.sign() : check a number is positive or negative.

Array extention

- Array.from() : cast array-like object or iterable object(Map,Set) to an array.

```
let arrlikeObj = {
"0":"aa",
"1":"bb",
"2":"cc",
length:3
};
let arr2 = Array.from(arrlikeObj);
```

or

```
Array.from(arrLike,x => x*x);
```

- Array.of() : Cast a value list to an array.
- Array instance copyWithin()
- Array instance find(), findIndex()
- Array instance fill(): use specific values to fill an array.
- Array instance entries(), keys(),values()
- Array instance includes(): check whether array contains specific value.

Function extention

- rest parameter, actually it's a array object.

```
function add(...nums){
let sum = 0;
for(let val of nums){
    sum += val;
    }
}

add(1,3,5);
```

- Spread operator it is just ...,it's the reverse of the rest parameter,cast an array to a parameter list which is splited by comma.

```
var nums = [1,3,5];
add(...nums);
```

- Arrow function var f = v => v*2;
    - this object is the object in definition rather than the object in using.

Object extention

- Property simplify

```
var name = "eirc";
var obj = {name};
console.log(obj.name);
```

- Method simplify

```
let obj = {
fun1(){
    console.log("hello");
    }
}
//another example
function getPoint(){
    let x = 1;
    let y = 2;
    return {x,y};
}
```

Promise

- Promise is an object which wraps asynchronous operation and return the async result or error information to callback functions.

- Promise's three status:
  - Pending: The operation is running.
  - Resolved: The operation finished.
  - Rejected: The operation failed or some error happened.
- Promise's status can't be changed again once it becomes to Resolved or Rejected.
- It will executes immediately once we create a promise object.
- Declare a promise:

```
//The two parameters resolve and reject is two functions supported by js engine.We
don't need to implement it.
//1.resolve : it's used to change promise status to Resolved. you can pass the async
operation results to it, it finally
// pass the result to resolve callback function.
//2.reject: it's used to change promise status to Reject.you can pass the error info
to it, it finally pass the error to
//reject callback function.
var promise = new Promise(function(resolve, reject){
    //do something...

    //when the operations above completed
    if(aboveOperationDone){
        resolve(asyncOperationResults);
    }
    else if(meetSomeError){
        reject(new Error("Error information."));
    }
});

promise.then(function(asyncOperationResults){
    //the content of resolve callback function
}, function(meetSomeError){
    //the content of reject callback function
});
```

- Instance functions
  - then(): It specifies the callback function for Resoved and Rejected status like above example. then() function will return a new promise object , wo we can use linked then() to impement some cases.

```
promise.then(function(asyncOperationResults){
    //the content of resolve callback function ...

    return resultFromThen1;
}).then(function(resultFromThen1){
    //do something....

    return resultFromThen2;
});
```

```
* catch(): We usually put catch() to the last of the linked functions to handle the error from prior functions,
```

another situation is that the catch function will be invoked once promise becomes to Rejected status or invoke reject() function. The error from promise will not transmitted to outer scope,so we should use catch() function to catch and handle the errors.

```
* done(): This function should always in the end of linked functions, it can catch any possible errors from promise and throw it globally.
* finally(): It accepts an function as parameter , the function will execute anyway at last.
```

- Static functions
    - Promise.all(): It accepts an array as parameter and finally return a new promise object, the array item will be converted to promise object if it isn't. var p = Promise.all([p1,p2,p3]) p can become to Resolved status only when all the array items become to Resolve status. p will become to rejected status once any array item become rejected status. and the return value of the first rejected promise will be passed to p's callback function.
    - Promise.race(): p will become to the array item's status whose status changed firstly. Like a race ,right? ~~
    - Promise.resolve(param): change param to a promise object.
    - Promise.reject(reason): return a new promise object whose status is rejected.

React

Syntac & API

- ReactDOM.render(): Transform template code to html code and insert to the specific element.

```
ReactDOM.render({
    <h1>this is a test</h1>,
    document.getElementById("elemId")
});
```

- You can insert variable and js code to jsx code by starting with {:

```
var arr = ["one","two","threee"];
ReactDOM.render({
    <div>
     arr.map(function(arrItemValue){
     return <div>This is {arrItemValue} </div>
    });
    </div>
});
```

- If the variable is an array, then jsx will put all the array items to the template:

```
var arr = [
    <h1> Arr one  </h1>,
    <ht> Arr two </h1>
];
ReactDOM.render(
<div> {arr} </div>,
document.getElementById("elemId")
);
```

Component

- You can use React.createClass() to create a react component to use like a html tag.
- Components definition specification:
  - Must start with uppercase letter, otherwise you will meet error.
  - Must have render() function to output component's content.
  - Can only have one top tag.
  - Can add any attributes to the component, and you can get them from this.props in the component class. But you can't use keywords of javascript as attribute name, like class (use className instead), for(use htmlFor instead)

```
Var Component1 = React.createClass({
    render: function(){
        return <div>
        Attribute value: {this.props.name}
        </div>
    }
});
//you can use it like below:
<Component1 name = "attr value" />
```

- You can get all the sub nodes of a component by this.props.children:

```
var NodeList = React.createClass({
render:function(){
return (
<ol>
{
React.Children.map(this.props.children,function(child){
    return <li> {child} </li>
})
}
</ol>
);
}
});

ReactDOM.render(
<NodeList>
<span> one </span>,
<span> two </span>
</NodeList>,
document.body
);
```

- You can validate compnent's attributes via PropTypes

```
var TheButton = React.createClass({
    propTypes:{
    name:React.PropTypes.string.isRequired, // this means the name attribute must be
a string and it is neceaary for the //component.
    },
    render: function(){
        return <h1> {this.props.title}</h1>
    }
});
```

- You can set attribute's default value by getDefaultProps:

```
var TheButton = React.createClass({
    getDefaultProps:function(){
    name:"Default Name"
    },
    render:function(){
    return <h1> {this.props.title}</h1>
    }
});
```

- You can get real DOM node by ref attribute:

```
var DivInput = React.createClass({
    handleClick:function(){
    this.refs.txt1.focus();
    },
    render:function(){
```

```
    return (
        <div>
        <input type="text" ref="txt1" />
        <input type="button" value = "button value" onClick = {this.handleClick} />
        </div>
    );
    }
});
Remark; You must set ref to the real DOM element if you want to get real DOM in
react class, but you should make sure that the real DOM has been loaded first.
```

- You can use this.state to store component's state info:

```
var ToggleButton = React.createClass({
    getInitialState:function(){
        return {flag:false};
    },
    handleClick:function(){
        this.setState({flag: !this.state.flag});
    },
    render:function(){
        var flagValue= this.state.flag ? "True" : "False";
        return (
        <div onClick = {this.handleClick}>
            The flag is {flagValue}
        </div>
        );
    }
});
//Remark: this.state represents the object getInitialState returns,you can call
setState if you want to change state value.
```

- You can get user's input value by event.target.value as this.props can't get user's input value.

```
var MyInput = React.createClass({
    getInitialState:function(){
    return {theValue: "testValue"};
    },
    handleChange:function(event){
    this.setState({thevalue:event.target.value});
    },
    render:function(){
    var tmpValue = this.state.theValue;
    return (
        <div>
        <input type="text" value = {tmpValue} onChange = {this.handleChange} />
        <p> {tmpValue} </p>
        </div>
    );
    }
});
```

Component's lifecycle

- Component's three states:
    - Mounting : the virtual DOM(components) has become to real DOM and inserted .
        - componentWillMount() // will be called before becoming to this state
        - componentDidMount() // will be called after becoming to this state
    - Updating : the component is re-rendering.
        - componentWillUpdate(object nextProps,object nextState);
        - componentDidlUpdate(object nextProps,object nextState);
    - Unmounting : has removed the real DOM.
        - componentWillUnmount
- Two special state handle functions
    - componentWillReceiveProps(object nextProps);//will be called when the loaded component receives new parameters.
    - shouldComponentUpdate(object nextProps,object nextState);//will be called when checking whether re-render.

```
var MyOpacity = React.createClass({
    getInitialState:function(){
        myOpacity:1.0
    },
    componentDidMount:function(){
        this.timer = setInterval(function(){
            var tmpOpacity = this.state.myOpacity;
            tmpOpacity -= 0.05;
            if(tmpOpacity < 0.1){
                tmpOpacity = 1.0;
            }
            this.setState({myOpacity:tmpOpacity});
        }.bind(this),100);//
    },
    render:function(){
        <div style={{opacity:this.state.myOpacity}}> //outer {} means this is js
code,inner {} means this is react css object.
            The name is {this.props.name}
        </div>
    }
});
```

```
ReactDOM.render(
    <MyOpacity name="Eric" />,
    document.body
);
```

- Ajax:We can use componentDidMount to set ajax request

```
var MyList = React.createClass({
    getInitialState:function(){
        return {
            key:"key1",
            value:"value1"
        };
    },

    componentDidMount:function(){
        $.get(this.props.url,function(data){
            var theData = data[0];
            if(this.isMounted()){
                this.setState({
                    key:theData.id,
                    value:theData.name
                });
            }
        }.bind(this));
    },

    render:function(){
        return (
            <div>
            Key: {this.state.key}, Value:{this.state.value}
            </div>
        );
    }
});

ReactDOM.render(
    <MyList url = "http://host.eric.com" />,
    document.body
);
```

Jade: A html template engine based on nodejs

- Install jdade npm install jade
- Online converter
- Official Site
- Html2Jade
- Other sites
  - https://www.npmjs.com/package/jade

API

- Label
  - html to <html> </html>
  - div#container to <div id = "container"> </div>
  - div.user-details to <div class="user-details"> <div>
  - div#id1.class1.class2 to <div id ="id1" class = "class1 class2">

```
#id1
.class1
```

to

```
<div id="id1"></div><div class="class1"></div>
```

- Label text
  - p hello! to <p>hello!</p>
  - large text

```
p
    | this is
    | a test
    | so...
```

to

```
<p>this is a test so...</p>
```

- use variable For json {"name":"eric",age:10} #user #{name} &lt;#{age}&gt; to <div id ="user">eric &lt;10&gt; <div>
- escape p \#{something} to <p>#{something}</p>
- use non-escape variable

```
* var html = "<script> </script>"
| !{html}
```

to

<script></script>

- Inline tag can also contain text block

```
label
  | Username:
  input(name='user[name]')
```

to

```
<label>Username:
  <input name="user[name]"/>
</label>
```

- Or use label text directly

```
label Username:
  input(name='user[name]')
```

to

```
<label>Username:
  <input name="user[name]"/>
</label>
```

- You can use a dot to start a block text

```
html
 head
  title Example
  script.
   if(1=1){
    var a = 1;
   }
```

to

```
<html>
  <head>
    <title>Example</title>
    <script>
      if(1=1){
       var a = 1;
      }
    </script>
  </head>
</html>
```

- Single line comments:

```
// just some paragraphs
p foo
p bar
```

to

```
<!-- just some paragraphs-->
<p>foo</p>
<p>bar</p>
```

- Single comments but not output to html

```
//- will not output to html
p foo
p bar
```

to

```
<p>foo</p>
<p>bar</p>
```

- Block comments

```
body
 //
  aaa
  bbb
  ccc
```

to

```
<body>
  <!--
  aaa
  bbb
  ccc
  -->
</body>
```

- Tag nests

```
ul
```

```
  li.first
    a(href='#') foo
  li
    a(href='#') bar
  li.last
    a(href='#') baz
```

or

```
ul
  li.first: a(href='#') foo
  li: a(href='#') bar
  li.last: a(href='#') baz
```

to

```
<ul>
  <li class="first"><a href="#">foo</a></li>
  <li><a href="#">bar</a></li>
  <li class="last"><a href="#">baz</a></li>
</ul>
```

- Case statement

```
html
  body
    - var friends = 10
    case friends
      when 0
        p you have no friends
      when 1
        p you have a friend
      default
        p you have #{friends} friends
```

to

```
<html>
  <body>
    <p>you have 10 friends</p>
  </body>
</html>
```

- Condition statements

```
- var user = { description: 'foo bar baz' }
- var authorised = false
#user
  if user.description
    h2.green Description
    p.description= user.description
  else if authorised
    h2.blue Description
    p.description.
      User has no description,
      why not add one...
  else
    h2.red Description
    p.description User has no description
```

to

```
<div id="user">
  <h2 class="green">Description</h2>
  <p class="description">foo bar baz</p>
</div>
```

mixin ,can create reusable blocks

- Pass parameter to mixin

```
mixin link(href, name)
  //- attributes == {class: "btn"}
  a(class!=attributes.cls href=href)= name

+link('/foo', 'foo')(cls="btn")
```

to

```
<a href="/foo" class="btn">foo</a>
```

- Pass parameter to mixin 2

```
mixin link(href, name)
  a(href=href)&attributes(attributes)= name

+link('/foo', 'foo')(cls="btn")
```

to

```
<a href="/foo" cls="btn">foo</a>
```

- Rest arguments

```
mixin list(pid, ...items)
  ul(ids=pid)
    each item in items
      li= item

+list('my-list', 1, 2, 3, 4)
```

to

```
<ul ids="my-list">
  <li>1</li>
  <li>2</li>
  <li>3</li>
  <li>4</li>
</ul>
```

Flux

- It is a framework idea to improve FEE software development structure , and make the project low coupling and easy maintenance.
- It is usually used with React.
- Some important concepts
  - View : can generate actions and send them to Dispatcher.
  - Action: the message that View sends.
  - Dispatcher : accept actions,and then execute callback function.
  - Store: the app's data and state info, it will notify Views to update once changes happen via sending "changing" events.

A whole example app with concept explanation

- We define a store object to store all data for the whole app.The store must implement event interface to notify view to re-render when state changes happen.

```
//stores/ListStore.js
var EventEmitter = require("events").EventEmitter;
var assign = require("object-assign");

var ListStore = {
    items:[],
    getAll:function(){
        return this.items;
    },
    addNewItemHandler:function(itemValue){
        this.items.push(itemValue);
    },
    emitChange:function(){
        this.emit("change");
    },
    addChangeListener:function(callback){
        this.on("change",callback);
    },
    removeChangeListener:function(callback){
        this.removeListener("change",callback);
    }
};

module.exports = ListStore;
```

- Define a pure view to display store data:

```
//components/MyButton.jsx
var React = require("react");
var MyButton = function(props){
    var items = props.items;
    var itemHtml = items.map(function(listItem,idx){
        return <li key = {idx}> {listItem} </li>
    });

    return <div>
        <ul> {itemHtml} </ul>
        <button onClick = {props.onClick} >Add New Item</button>
    </div>
};

module.exports = MyButton;
```

- Define action

```
// actions/ButtonAction.js
// Every action is an object which contain data which will be tranfered to
dispatcher
var AppDispatcher = require("../dispatcher/AppDispatcher");

var ButtonActions = {
    addNewItem : function(pText){
        //note: this use AppDispatcher to transfer action ADD_NEW_ITEM to
dispatcher.
        AppDispatcher.dispatch({
            actionType:"ADD_NEW_ITEM",
            text:pText
        });
    }
};
```

- Define dispatcher which accept and handle actions.

```
// dispatcher/AppDispatcher.js
//note: the whole app can only have only one dispatcher.
//it accept action from view and notify store that data changes
var ListStore = require("../stores/ListStore");

AppDispatcher.register(function(action){
    switch(action.actionType){
        case "ADD_NEW_ITEM":
            ListStore.addNewItemHandler(action.text);
            ListStore.emitChange();
            break;
        default:
            //do nothing
    }
});
```

- Create a controller view as a top level component that holds all state and pass the state to children as props.

```
// component/MyButtonController.jsx
var React = require("react");
var ListStore = require("../stores/ListStore");
var ButtonActions = require("../actions/ButtonActions");
var MyButton = require("./MyButton");

//this is a really react component
var MyButtonController = React.createClass({
    getInitialState: function(){
        return {
            items:ListStore.getAll();
        };
    },

    componentDidMount:function(){
        ListStore.addNewItem(this._onChange);
    },

    componentWillUnmount:function(){
        ListStore.removeChangeListener(this._onChange);
    },

    _onChange:function(){
        this.setState({
            items:ListStore.getAll()
        });
    },

    createNewItem:function(event){
        ButtonActions.addNewItem("New Item");
    },

    render:function(){
        return <MyButton items = {this.state.items} onClick = {this.createNewItem}
/>
    }
});
```

- Create index.jsx to use the react component

```
//index.jsx
var React = require("react");
var ReactDOM = require("react-dom");
var MyButtonController = require("./components/MyButtonController");

ReactDOM.render(
    <MyButtonController />,
    document.querySelector("#container")
);
```

Redux

- Description: Redux is an FEE framework based on Flux. You can get more on official site
- Core concepts:
  - Web app is a state machine, a specific view corresponds to a specific state.
  - All state stores in an object(store).

Concepts and API

- Store: Data container, we can only have one store in whole app.

    - store.getState: get current state object.
    - store.dispatch: send an action to store.
    - store.subscribe: set a listen function, execute the function when state changes.
- State: Data snapshot, a state corresponds to a view.we can get state from store: `store.getState();`
- Action: Describe what happens on the view, it also can bring some data. the *type* property is required.
- Reducer: A function which accepts a state and action, then return a new state according to business logic.
- View: Render data, send action to store,re-render data which from store.

Play a demo

- create a folder *redux-demo*
- Open CLI and cd to the folder, run `npm init` to create package.json file.
- Install related modules: `npm install --save react react-dom redux react-redux`
- Install react-scripts `npm install --save-dev react-scripts`
    - react-scripts is a convenient integration tool which help you to setup an app without configuration.
- The content of package.json is:

```
{
  "name": "redux-demo",
  "version": "1.0.0",
  "description": "A redux simple demo",
  "main": "index.js",
   "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "eject": "react-scripts eject",
    "test": "react-scripts test"
  },
  "keywords": [
    "Redux",
    "Redux-demo"
  ],
  "author": "eric",
  "license": "ISC",
  "dependencies": {
    "react": "^15.4.1",
    "react-dom": "^15.4.1",
    "react-redux": "^5.0.1",
    "redux": "^3.6.0"
  },
  "devDependencies": {
    "react-scripts": "^0.8.4"
  }
}
```

- Create a folder *src* under *redux-demo*
- Create a folder *components* under *src*
- Create a view *Counter.js* under *components*, the code like below:

```
// /src/components/Counter.js
import React,{Component,PropTypes} from 'react'

class Counter extends Component{
    // the propTypes is a special react property, it will validate properties of
props
    static propTypes = {
        varlue: PropTypes.number.isRequired,
        onIncrement:PropTypes.func.isRequired,
        onDecrement:PropTypes.func.isRequired
    }

    //function
    incrementIfOdd = () => {
        if (this.props.value % 2 !== 0) {
            this.props.onIncrement()
        }
    }

    //function
    incrementAsync = () => {
        setTimeout(this.props.onIncrement,1000)
    }

    //render() is indispensable
    render() {
        //as props is an array, so we can destruct it to some variables
        const {value,onIncrement,onDecrement} = this.props;
        return (
            <p>
                Clicked:{value} times.
                {' '}
                <button onClick={onIncrement} > + </button>
                {' '}
                <button onClick={onDecrement} > - </button>
                {' '}
                <button onClick={this.incrementIfOdd} > Increment if odd </button>
                {' '}
                <button onClick={this.incrementAsync} > Increment async </button>
            </p>
        );
    }

}

export default Counter
```

- Create a folder *reducers* under *src*
- Create a reducer in *index.js* under *reducers*

```
// src/reducers/index.js
//The below is a reducre function which is declared by ES6 syntax
export default (state =0, action ) => {
    switch(action.type){
        case "INCREMENT":
            return state + 1;
        case "DECREMENT":
            return state - 1;
        default:
            return state;
    }
}
```

- Create main entry point file *index.js* under *src*, in this file, we create a store, subscribe state change event, at last render the view.

```
// src/index.js
import React from "react"
import ReactDOM from "react-dom"
import {createStore} from "redux"
import Counter from "./components/Counter"
import reducer from "./reducres"

//createStore syntac: createStore(reducer, initialState[optional])
//so once we pass a reducre when create store, the reducre will auto execute when
invoking sotre.dispatch function.
const store = createStore(reducer);
const rootElem = document.getElementById("root");

const render = () => ReactDOM.render(
    <Counter
        value = {store.getState()}
        onIncrement = {() => store.dispatch({type:"INCREMENT"}) }
        onDecrement = {() => store.dispatch({type: "DECREMENT"}) }
    />.
    rootElem
);

render();
//when state changes, the render function will execute automatically
store.subscribe(render);
```

- create a folder *public* under *redux-demo*
- create a html file *index.html* to host the view.

```
<!doctype html>
<html>
    <head>
        <title> Redux demo </title>
    </head>
    <body>
        <div id ="root" >
        </div>
    </body>
</html>
```

- run `npm start` to start the app and play it !
  - you can also run `npm run build` to build the project and observe the changes of folder.

Middleware and asynchronous operations

- Middleware is a function in redux, it's used to add new feature when dispatching actions.
- A simple middleware like below:

```
import React from 'react'
import ReactDOM from 'react-dom'
import { createStore } from 'redux'
import counter from './reducers'

const store = createStore(counter)

let next = store.dispatch;
store.dispatch = function dispatchAndLog(action){
  console.log("dispatching",action);
  next(action);
  console.log("next state",store.getState());
}
```

In above example, we re-write the dispatch function as dispatchAndLog and append log feature to it. The dispatchAndLog is the middleware.

- Usually we have many redux middlewares to use from thirdparty, for example, we can use redux-logger to add logger feature for redux.

```
import {applyMiddleware,createStore} from "redux";
import createLogger from "redux-logger";
const logger = createLogger();

const store = createStore(reducer,
initial_state,
applyMiddleware(logger,middleware2,middleware3...)
);
```

Note: applyMiddleware is redux function,it will execute the middleware function orderly and execute store.dispatch at last.

- redux-thunk middleware:enhance store.dispatch() function to make dispatch function can accept function as parameter.

```
import {createStore, applyMiddleware} from "redux";
import thunk from "redux-thunk";
import reducer from "./reducers"

const store = createStore(
    reducer,
    applyMiddleware(thunk)
);
```

- redux-promise middleware: enhance store.dispatch() function to make dispatch function can accept promise object as parameter.

```
import { createStore, applyMiddleware } from 'redux';
import promiseMiddleware from 'redux-promise';
import reducer from './reducers';

const store = createStore(
  reducer,
  applyMiddleware(promiseMiddleware)
);
//use it
const fetchPosts =
  (dispatch, postTitle) => new Promise(function (resolve, reject) {
     dispatch(requestPosts(postTitle));
     return fetch(`/some/API/${postTitle}.json`)
       .then(response => {
         type: 'FETCH_POSTS',
         payload: response.json()
       });
});
```

react-redux

- A redux library for react. there are only two kind of components: UI component and Container component in react-redux.
- UI Component
  - It should not contain any business logic, only to show UI.
  - Should not use this.state,so it doesn't have state.
  - All the data to show come from this.props.
  - Should not use any Redux API
  - It's provided by user.
- Container component
  - It's not used to show UI but to manage the data and business logic,
  - It has its state.
  - It use Redux API
  - It is generated automatically by React-Redux, use should not to create it manually.
- API
  - connect() : generate container component from UI component.
  - mapStateToProps() : map state to props of UI Component.
  - mapDispatchToProps(): map parameters of UI Component to store.dispatch. Note, it can be a function or an object.
    - When is a function: it will get two parameters: dispatch and ownProps (Container Component's props object). It finally return an object which contains key-value pairs, the key is the UI Component's parameter, the value is a function to dispatch an action, Let's see an example below:

  ```
  ```

const mapDispatchToProps = (dispatch, ownProps) => { return { // the onClick is UIComponent's parameter' onClick : () => { //send an action to store dispatch({ type:"ADD_ITEM", text:"new item1" }); }//onClick };//return } ``` * When it's an object , the key is the UI Component's parameter,the value is an Action Creater function, the action will be sended by redux automatically. Example below:

```
    ```
```

const mapDispatchToProps = { onClick: (pTxt) => { type: "ADD_ITEM", text: pTxt }//onClick }//mapDispatchToProps ```

```
* Provider component: It usually wrap the root component to help all the sub
component to get state object easily.
```
```

```
import {Provider} from "react-redux"
import {createStore} from "redux"

let store = createStore(reducer);

render(
    <Provider store = {store} >
        <App />
    </Provider>,
    document.getElementById('elemId')
);
```

- A whole example with explanation

```
import React, {Component,PropTypes } from "react";
import ReactDOM from "react-dom";
import { createStore } from "redux";
import { Provider,connect } from "react-redux";

//UI Component
class Counter extends Component{
    propTypes:{
        value: PropTypes.number.isRequired,
        onIncreaseClick: PropTypes.func.isRequired
    },

    render(){
        const {value, onIncreaseClick} = this.props;
        return (
            <div>
                <span>{value}</span>
                <button onClick ={onIncreaseClick}> Increase </button>
            </div>
        );
    }
}

//Action
const increaseAction = {type:"ADD_ITEM"}

//Reducer
function counterReducer(state ={ count: 0}, action ){
    const theCount = state.count;
    switch(action.type){
        case "ADD_ITEM":
            //return new state according to original state and action
            return {count:count +1 }
        default:
            return state
    }
}

//store
const store = createStore(counterReducer);

//map redux state to UI props
function mapStateToProps(state){
    return {
        //value is from UI props
        value: state.count
    }
}

//map redux action to UI props
function mapDispatchToProps(dispatch){
    return {
        onIncreaseClick : () => dispatch(increaseAction);
    }
}

//generate container Component by invoking connect()
const App = connect(
    mapStateToProps,
    mapDispatchToProps
)(Counter);

//render UI
ReactDOM.render(
    <Provider store = {store} >
        <App />
    </Provider>,
    document.getElementById("ElemId")
);
```

React-Router

- Just study the official example

Less

- Official site
- Less is a CSS pre-processor, it enhance the css's feature so that we can use css like a normal language,it also make css more maintaainable, extendable.
- Install Less: npm install -g less
- Install clean-css plugin which can compile less file to compressed css file: npm install -g less-plugin-clean-css
- Compile a less file to minified css file: lessc --clean-css test.less test.min.css
- We can also use less api in node js code:

```
var less = require("less");
less.render(".class {width: (1+1) }",
{
    path:['.', './lib'], //specify search paths for @import directives
    filename:"test.less", //the file to be compiled
    compress:true //Minify css output
},
 function(e, output) {
    //output the compiled css
    console.log(output.css);
});
```