```
// 對外擴充方法
const _$ImportModules = (function (G) {
    // ////////////////////////////////////////////////////////////////////////
    //
    // extension_1
    //
    // ////////////////////////////////////////////////////////////////////////
    // 資訊
    const _extension1 = {
        callback_uid: 1,
        callback_key: "$_callbackGuid",
        "_": null,
        info: {
            sourceScriptPath: null,
            extension1ScriptPath: null,
            environment: null,
        },
    };

    const ImportModuleList = [];
    // ========================================================================
    // 工廠函式
    class Factory {
        // ---------------------------
        // 取得 _, _extention 的路徑
        static _getPath(_) {
            // debugger;

            const info = _extension1.info;
            const environment = info.environment;

            if (/nodejs/.test(environment)) {

                info.extension1ScriptPath = __dirname;

            } else if (/browser/.test(environment)) {

                if (typeof document == 'undefined') {
                    return;
                }

                let scripts = Array.from(document.querySelectorAll('script'));
                let script = scripts.pop();

                info.extension1ScriptPath = script.src;
                // ---------------------------
                // find scriptPath
                let reg = /(\\|\/)?([^\/]*?(underscore|lodash)[^\/]*?)$/i;

                while ((script = scripts.pop()) != null) {
                    let src = script.src;
                    if (src && reg.test(src)) {
                        info.sourceScriptPath = src;
                        break;
                    }
                }
            }
        }
        // ---------------------------
        static _link(_) {
            Object.defineProperty(_, '$extension1', {
                value: _extension1,
```

```
63                  enumerable: false,
64                  writable: false,
65                  configurable: true
66              });
67          }
68          // ----------------------------
69          // m: 注入函式
70          static importModule(m) {
71              // debugger
72              const environment = _extension1.info.environment;
73              const _ = _extension1._;
74
75              switch (environment) {
76                  case 'nodejs':
77                      if (_ == null) {
78                          // _尚未引入，先將模組放置等候區
79                          ImportModuleList.push(m);
80                      } else {
81                          m(_);
82                      }
83                      break;
84                  default:
85                      if (_ == null) {
86                          throw new Error('_extension1 need import _');
87                      } else {
88                          m(_);
89                      }
90                      break;
91              }
92          }
93          // ----------------------------
94          // nodejs
95          static injectModules() {
96              const _ = _extension1._;
97
98              if (_ == null) {
99                  throw new Error('no import _');
100             }
101             let m;
102
103             while ((m = ImportModuleList.shift()) != null) {
104                 m(_);
105             }
106         }
107         // ----------------------------
108         static main() {
109
110             const _ = _extension1._;
111
112             if (_ == null) {
113                 throw new Error('no import _');
114             }
115
116             if (_.$extension1 != null) {
117                 // 避免重複
118                 return;
119             }
120
121             Factory._link(_);
122
123             Factory._getPath(_);
124         }
```

```
125     }
126     // ================================================================================
127
128     // 對外引入模組的函式
129     function ImportModule(m) {
130         Factory.importModule(m);
131     };
132
133     // 引入 _ 的窗口
134     // arg: [_.path: _的路徑， _obj: _本身]
135     function nodeJs_extension(arg) {
136         let _;
137         let _path;
138
139         if (typeof arg == 'string') {
140             _ = require(path);
141         } else {
142             _ = arg;
143         }
144         // 取得 _ 在本機的位置
145         _path = require.resolve(path);
146
147         _extension1.info.sourceScriptPath = _path;
148
149         Factory.main();
150
151         Factory.injectModules();
152
153         return _;
154     }
155     // ================================================================================
156
157     (function () {
158         // debugger
159         // 環境檢測
160
161         if (typeof window != 'undefined' && typeof document != 'undefined') {
162             // browser
163
164             let environment = 'browser';
165             let _ = window._;
166
167             _extension1._ = _;
168             _extension1.info.environment = environment;
169
170             Factory.main();
171         } else if (typeof (module) != 'undefined' && typeof (module.exports) != 'undefined') {
172
173             _extension1.info.environment = 'nodejs';
174
175             // nodejs 對外窗口
176             module.exports = {
177                 extension: nodeJs_extension,
178                 importModule: ImportModule,
179             };
180
181         } else if (typeof (window) == 'undefined' && self != 'undefined' && typeof
(importScripts) == 'function') {
182             // webWorker 環境
183
184             let _ = self._;
185             _extension1.info.environment = 'worker';
```

```
            _extension1._ = _;

            Factory.main();
        } else {
            throw new Error('no support current system');
        }
    }());

    // ====================================================================
    return ImportModule;
})(this);
```

```
 1 !(function (G, ImportModule) {
 2
 3     const functionMap = {};
 4     //==========================================================================
 5     // isPlainObject
 6     const isPlainObject = function (obj) {
 7         debugger;
 8
 9         if (typeof obj != "object") {
10             return false;
11         }
12
13         if (obj == null) {
14             return false;
15         }
16
17         let res = Object.prototype.toString.call(obj);
18
19         if (!/^\[object Object\]$/.test(res)) {
20             return false;
21         }
22
23         if (obj.constructor !== {}.constructor) {
24             return false;
25         }
26
27         return true;
28     };
29     functionMap['isPlainObject'] = isPlainObject;
30     //==========================================================================
31     // getClassName
32
33     functionMap['getClassName'] = function (data) {
34         let _toString = Object.prototype.toString;
35
36         let type = typeof (data);
37
38         if (/object/.test(type)) {
39
40             if (data === null) {
41                 type = "null";
42             } else {
43                 type = _toString.call(data);
44
45                 let res = /\[\w+\s+(\w+)\]/.exec(type);
46                 if (res && res[1]) {
47                     type = res[1];
48                 }
49             }
50         }
51         return type;
52     };
53
54     //==========================================================================
55     // ptimeout
56     //
57     // job: [function|promise]
58     functionMap['ptimeout'] = function (job, timeLimit, context) {
59         debugger;
60
61         let msg;
62
```

```
63          let p2;
64
65          if (typeof timeLimit != 'number') {
66              throw new TypeError("timeout arg[1] must be number");
67          }
68
69          if (typeof (job) == "function") {
70
71              if (context != null) {
72                  job = job.bind(context);
73              }
74              p2 = new Promise(job);
75          } else if (job instanceof Promise) {
76              p2 = job;
77          } else {
78              throw new TypeError("timeout arg[0] must be promise or function");
79          }
80          //-----------------------
81          let _res;
82          let _rej;
83
84          let p1 = new Promise(function (res, rej) {
85              debugger;
86              _res = res;
87              _rej = rej;
88          });
89          //-----------------------
90          p2.then(function (data) {
91              debugger;
92              _res(data);
93          }, function (err) {
94              _rej(err);
95          });
96
97          setTimeout(function () {
98              _rej(new Error('timeout'));
99          }, timeLimit);
100         //-----------------------
101
102         return p1;
103     };
104
105     //=============================================================================
106     // promise
107     //
108     // callback: [function(返回 promise)|promise[]]
109     // context: 背後執行對象
110     functionMap['promise'] = function (callback, context) {
111         let p;
112
113         if (callback instanceof Promise) {
114             p = Promise.resolve(callback);
115         } else if (typeof (callback) == "function") {
116             callback = (context == null) ? callback : callback.bind(context);
117
118             p = new Promise(callback);
119         } else if (Array.isArray(callback)) {
120
121             if (context != null) {
122                 callback = callback.map(function (fn) {
123                     return fn.bind(context);
124                 });
```

```
            }

            p = Promise.all(callback);
        } else {
            p = Promise.resolve(callback);
        }
        //-----------------------
        if (p['$status'] == null) {
            Object.defineProperty(p, '$status', {
                value: 0,
                enumerable: false,
                writable: true,
                configurable: true
            });
        }

        p.then(function () {
            p['$status'] = 1;
        }, function (err) {
            p['$status'] = 2;
            err = (err instanceof Error) ? err : new Error(err);
            throw err;
        });

        return p;
    };
    //==========================================================================
    // deferred
    functionMap['deferred'] = (function () {

        (function () {
            // 對系統的 promise 擴增 API
            if (typeof Promise.prototype.thenWith == 'undefined') {
                Promise.prototype.thenWith = function (onFulfilled, onRejected, context) {

                    onFulfilled = onFulfilled.bind(context);
                    onRejected = onRejected.bind(context);

                    return this.then(onFulfilled, onRejected);
                };
            }
            //-----------------------------------------------------------------
            // promise.catchWith()
            if (typeof Promise.prototype.catchWith == 'undefined') {
                Promise.prototype.catchWith = function (onRejected, context) {

                    onRejected = onRejected.bind(context);

                    return this.then(null, onRejected);
                };
            }
            //-----------------------------------------------------------------
            // promise.always()
            if (typeof Promise.prototype.always == 'undefined') {
                Promise.prototype.always = function (callback) {

                    return this.then(function (data) {
                        callback(false, data);
                    }, function (error) {
                        callback(true, error);
                    });
                };
```

```
187             }
188             //---------------------------------------------------------------------
189             // promise.alwaysWith()
190             if (typeof Promise.prototype.alwaysWith === 'undefined') {
191                 Promise.prototype.alwaysWith = function (callback, context) {
192 
193                     callback = callback.bind(context);
194 
195                     return this.then(function (data) {
196                         callback(false, data);
197                     }, function (error) {
198                         callback(true, error);
199                     });
200                 };
201             }
202         })();
203 
204         const Deferred = (function () {
205             // 模組範圍
206 
207             class Deferred {
208 
209                 constructor() {
210                     this.fn = Deferred;
211                     this._reject;
212                     this._resolve;
213                     this._promise;
214 
215                     this._init();
216                 }
217 
218                 get allStatusList() {
219                     return ['pending', 'fulfilled', 'rejected'];
220                 }
221 
222                 _init = function () {
223                     let $this = this;
224 
225                     this._promise = _.promise(function (resolve, reject) {
226                         this._resolve = resolve;
227                         this._reject = reject;
228                     }, this);
229 
230                     this._setStatus(0);
231 
232                     this._setStatusGet();
233 
234                     this._promise.then(function (data) {
235                         $this._setStatus(1);
236                         return data;
237                     }, function (err) {
238                         $this._setStatus(2);
239                     });
240                 }
241 
242                 _setStatusGet() {
243                     // 防止修改 this.status
244 
245                     let target = this._promise;
246 
247                     Object.defineProperty(this, 'status', {
248                         enumerable: true,
```

```
249                         configurable: true,
250                         get: function () {
251                             return target['$status'];
252                         },
253                         set: function () {
254                             return;
255                         }
256                     });
257                 }
258 
259                 promise() {
260                     return this._promise;
261                 };
262                 //-------------------------------------------------------------------
263                 resolve(arg) {
264                     this._resolve(arg);
265                 };
266                 //-------------------------------------------------------------------
267                 reject(err) {
268                     this._reject(err);
269                 };
270                 //-------------------------------------------------------------------
271                 then = function (onFulfilled, onRejected) {
272                     var def = Deferred();
273                     var p = this.promise();
274 
275                     p = p.then(this._getCallback(onFulfilled),
276                         this._getErrorCallback(onRejected));
277                     //-----------------------
278                     p.then(function (data) {
279                         def.resolve(data);
280                     }, function (error) {
281                         def.reject(error);
282                     });
283                     return def;
284                 };
285                 //-------------------------------------------------------------------
286                 thenWith = function (onFulfilled, onRejected, context) {
287                     var def = Deferred();
288                     var p = this.promise();
289 
290                     p = p.then(this._getCallback(onFulfilled, context),
291                         this._getErrorCallback(onRejected, context));
292                     //-----------------------
293                     p.then(function (data) {
294                         def.resolve(data);
295                     }, function (error) {
296                         def.reject(error);
297                     });
298                     return def;
299                 };
300                 //-------------------------------------------------------------------
301                 catch = function (onRejected) {
302                     var def = Deferred();
303                     var p = this.promise();
304 
305                     p = p.catch(this._getErrorCallback(onRejected));
306                     //-----------------------
307                     p.then(function (data) {
308                         def.resolve(data);
309                     }, function (error) {
310                         def.reject(error);
```

```
311                     });
312                     return def;
313                 };
314                 //-------------------------------------------------------------------
315                 catchWith = function (onRejected, context) {
316                     var def = Deferred();
317                     var p = this.promise();
318 
319                     p = p.catch(this._getErrorCallback(onRejected, context));
320                     //-----------------------
321                     p.then(function (data) {
322                         def.resolve(data);
323                     }, function (error) {
324                         def.reject(error);
325                     });
326                     return def;
327                 };
328                 //-------------------------------------------------------------------
329                 always = function (callback) {
330                     var def = Deferred();
331                     var p = this.promise();
332 
333                     p = p.then(this._getAlwaysCallback(callback, false),
334                         this._getAlwaysCallback(callback, true));
335                     //-----------------------
336                     p.then(function (data) {
337                         def.resolve(data);
338                     }, function (error) {
339                         def.reject(error);
340                     });
341                     return def;
342                 };
343                 //-------------------------------------------------------------------
344                 alwaysWith = function (callback, context) {
345                     callback = callback.binf(context);
346 
347                     var def = Deferred();
348                     var p = this.promise();
349 
350                     p = p.then(this._getAlwaysCallback(callback, false, context),
351                         this._getAlwaysCallback(callback, true, context));
352                     /*-------------------------*/
353                     p.then(function (data) {
354                         def.resolve(data);
355                     }, function (error) {
356                         def.reject(error);
357                     });
358                     return def;
359                 };
360                 //-------------------------------------------------------------------
361                 isPending = function () {
362                     return (this._promise['$status'] == 0);
363                 };
364                 //-------------------------------------------------------------------
365                 isFulfilled = function () {
366                     return (this._promise['$status'] == 1);
367                 };
368                 //-------------------------------------------------------------------
369                 isRejected = function () {
370                     return (this._promise['$status'] == 2);
371                 };
372                 //-------------------------------------------------------------------
```

```
373                 _setStatus = function (status) {
374                     this._promise['$status'] = status;
375                 };
376                 //--------------------------------------------------------------------
377                 _getCallback = function (callback, context) {
378                     if (callback == null) {
379                         return null;
380                     }
381 
382                     callback = (context === undefined ? callback : callback.bind(context));
383 
384                     return function (d) {
385                         return callback(d);
386                     };
387                 };
388                 //--------------------------------------------------------------------
389                 _getErrorCallback = function (callback, context) {
390                     if (callback == null) {
391                         return null;
392                     }
393 
394                     callback = (context === undefined ? callback : callback.bind(context));
395 
396                     return function (err) {
397                         return callback(err);
398                     };
399                 };
400                 //--------------------------------------------------------------------
401                 _getAlwaysCallback = function (callback, args, context) {
402                     if (callback == null) {
403                         return null;
404                     }
405 
406                     callback = (context === undefined ? callback : callback.bind(context));
407 
408                     return function (d) {
409                         return callback(args, d);
410                     };
411                 }
412             }
413 
414             return Deferred;
415         })();
416 
417 
418         return function () {
419             return new Deferred();
420         };
421     })();
422     //================================================================================
423     if (ImportModule) {
424         // 注入功能
425         ImportModule(importFactory);
426     } else {
427         // nodejs 的引入窗口
428         module.exports = function (ImportModule) {
429             ImportModule(importFactory);
430         };
431     }
432     //----------------------------
433     // 對外曝露工廠
434     function importFactory(_) {
```

```
435         // 當前環境
436 
437         if(typeof _.$extension1 == "undefined"){
438             throw new Error("no import _");
439         }
440 
441         const _extension1 = _.$extension1;
442 
443         const environment = _extension1.info.environment;
444 
445         for (let funKey in functionMap) {
446             let m = functionMap[funKey];
447 
448             if (Array.isArray(m.unsupportEnvironment) &&
449 
450                 // 確定各函式能執行的環境
451                 m.unsupportEnvironment.includes(environment)) {
452                 delete functionMap[funKey];
453             }
454 
455             if (funKey in _) {
456                 // 避免衝突
457                 delete functionMap[funKey];
458             }
459         }
460         _.mixin(functionMap);
461     }
462 
463 })(this, (typeof _$ImportModules != "undefined" ? _$ImportModules : null));
```

```
 1 ////////////////////////////////////////////////////////////////////////////
 2 //
 3 // 任務
 4 //
 5 ////////////////////////////////////////////////////////////////////////////
 6 let jobUID = 0;
 7
 8 class Job {
 9     constructor(manager, args) {
10         this.id = jobUID++;
11
12         // 整合者
13         this.manager = manager;
14
15         // 執行命令
16         this.command;
17
18         // 使用者給的參數
19         this.args;
20
21         //------------------
22         this._promise;
23         this._resolve;
24         this._reject;
25         //------------------
26         this._init(args);
27     }
28     //--------------------------------------------------------------------
29     _init(args) {
30         const $this = this;
31
32         this.command = args.shift();
33
34         if (typeof (_[this.command]) !== "function") {
35             throw new TypeError("no this function(" + this.command + ")");
36         }
37
38         this.args = args;
39
40         this._promise = new Promise(function (res, rej) {
41             $this._resolve = res;
42             $this._reject = rej;
43         });
44     }
45     //--------------------------------------------------------------------
46     // 對 worker 發出命令
47     getCommand() {
48         let command = {
49             command: (this.command),
50             args: (this.args),
51             jobID: (this.id)
52         };
53         return command;
54     };
55     //--------------------------------------------------------------------
56     resolve(data) {
57         this._resolve(data);
58     };
59     //--------------------------------------------------------------------
60     reject(e) {
61         this._reject(e);
62     };
```

```
63      //-----------------------------------------------------------------
64      promise() {
65          return this._promise;
66      };
67  }
68
69  Job.GModules = {};
70
71  export { Job };
```

```
 1 import { workerSettings } from './settings_1.js';
 2 import { WorkerManager } from './workermanager_1.js';
 3
 4 ////////////////////////////////////////////////////////////////////////////
 5 //
 6 // 使用方式
 7 // _.worker(原本 _ 的命令, 原本的參數...)
 8 //
 9 // 參數設定
10 // maxWorkers: 同時最多能跑 worker 的數量
11 // idleTime: worker 沒工作後多久會被銷毀
12 // sourceScriptPath: 告訴 worker _ 的路徑在哪
13 // extension1Path: 告訴 worker _.extension1 的路徑在哪
14 // settings: 得知所有設定的數值(唯讀)
15 //
16 ////////////////////////////////////////////////////////////////////////////
17
18 function root() {
19     let args = Array.from(arguments);
20     let manager = WorkerManager.get_instance(root);
21
22     // debugger;
23     let p = manager.addJob(args);
24
25     return p;
26 }
27
28 root.GModules = {};
29
30 // webWorker 為了是怕與 window.Worker 撞名
31 export { root };
32
33 workerSettings.GModules["root"] = root;
34 WorkerManager.GModules["root"] = root;
35 //----------------------------
36 // 對外的設定
37 (function (fn) {
38     // 取得設定
39     Object.defineProperty(fn, 'settings', {
40         enumerable: true,
41         configurable: false,
42         writable: false,
43         value: workerSettings,
44     });
45     //----------------------------
46     // 要初始化幾個 workers
47     // 預設是只有啟動 _.worker() 才會建立 worker
48     fn.initWorkers = function (count) {
49         let manager = WorkerManager.get_instance();
50         manager.initWorkers(count);
51     };
52     //----------------------------
53     fn.setSourceScriptPath = function () {
54
55     };
56     //----------------------------
57     fn.setExtension1ScriptPath = function () {
58
59     };
60     //----------------------------
61     fn.setMaxWorkers = function () {
62
```

```
63      };
64      //----------------------------
65      fn.setMinWorkers = function () {
66
67      };
68      //----------------------------
69      fn.setIdleTime = function () {
70
71      };
72      //----------------------------
73      fn.addImportScript = function () {
74
75      };
76      //----------------------------
77      fn.getWorkers = function () {
78          let manager = WorkerManager.get_instance();
79          let workerList = Array.from(manager.workers);
80          return workerList;
81      };
82      //----------------------------
83      fn.getJobs = function () {
84
85      };
86  })(root);
87
88  //----------------------------
89  (function () {
90      // 注入 _ 的工廠
91      let _;
92
93      if (typeof (window) == "undefined" || typeof (document) == "undefined") {
94          // es6 只能在 browser 下跑
95          return;
96      }
97
98      if (typeof (window._) != "undefined") {
99          _ = window._;
100     }
101
102     root.GModules["_"] = _;
103
104     // 限制作用環境
105
106     Object.defineProperty(root, '_', {
107         enumerable: false,
108         configurable: false,
109         writable: false,
110         value: _,
111     });
112
113     _.mixin({
114         worker: root
115     });
116 })();
```

```
 1 ////////////////////////////////////////////////////////////////////////////
 2 //
 3 // 預設 worker 會彈性隨效能在(max_workers, min_workers)之間調整數量
 4 //
 5 // 但若不想 worker 的數量自動彈性調整，可以設定(max_workers = min_workers)
 6 //
 7 ////////////////////////////////////////////////////////////////////////////
 8
 9
10
11 const defaultSetting = {};
12
13 (function () {
14     const $d = defaultSetting;
15
16     // 同時能運行最大的 worker 數量
17     $d.max_workers = 2;
18
19     // idle 時要維持幾個 workers 待命
20     $d.min_workers = 0;
21
22     // 當 worker 沒任務可接時
23     // 閒置多久後會被銷毀
24     $d.idleTime = (1000 * 30);
25
26     // (uderscore, lodash) url
27     $d.sourceScriptPath;
28
29     // extension1.url
30     $d.extension1ScriptPath;
31
32     // 要被 import 的 script
33     $d.importScriptList = [];
34 })();
35 //--------------------------------------
36 const SettingProxy = {
37     addImportScript: function (script) {
38
39         if (!Array.isArray(script)) {
40             script = [script];
41         }
42         script.forEach(function (s) {
43             defaultSetting.importScriptList.push(s);
44         });
45     },
46 };
47
48 SettingProxy.GModules = {};
49
50 export { SettingProxy as workerSettings };
51
52
53 (function () {
54
55     Object.defineProperty(SettingProxy, 'max_workers', {
56         enumerable: true,
57         configurable: false,
58         // writable: false,
59         get: function () {
60             return defaultSetting.max_workers;
61         },
62         set: function (count) {
```

```
 63
 64             if (count < defaultSetting.min_workers) {
 65                 throw new Error('worker maxWorkersCount <
minWorkersCount(${defaultSetting.min_workers})');
 66             }
 67
 68             if (count < 1) {
 69                 errorList.push('max_workers must >= 1');
 70             }
 71
 72             defaultSetting.max_workers = count;
 73         }
 74     });
 75
 76     //------------------
 77     Object.defineProperty(SettingProxy, 'min_workers', {
 78         enumerable: true,
 79         configurable: false,
 80         // writable: false,
 81         get: function () {
 82             return defaultSetting.min_workers;
 83         },
 84         set: function (count) {
 85             if (count > defaultSetting.max_workers) {
 86                 throw new Error('workers count >
maxWorkersNum(${defaultSetting.max_workers})');
 87             }
 88             workerSettings.min_workers = count;
 89         }
 90     });
 91     //------------------
 92     Object.defineProperty(SettingProxy, 'idleTime', {
 93         enumerable: true,
 94         configurable: false,
 95         // writable: false,
 96         get: function () {
 97             return defaultSetting.idleTime;
 98         },
 99         set: function (time) {
100
101             if (time < 0) {
102                 throw new Error("idleTime must be >= 0");
103             }
104             defaultSetting.idleTime = time;
105         }
106     });
107     //------------------
108     Object.defineProperty(SettingProxy, 'sourceScriptPath', {
109         enumerable: true,
110         configurable: false,
111         // writable: false,
112         get: function () {
113             const root = SettingProxy.GModules["root"];
114             const _ = root.GModules["_"];
115
116             let info = _.$extension1.info;
117
118             let path = (defaultSetting.sourceScriptPath != null) ?
119                 defaultSetting.sourceScriptPath : info.sourceScriptPath;
120
121             return path;
122         },
```

```
123         set: function (path) {
124             defaultSetting.sourceScriptPath = path;
125         }
126     });
127     //------------------
128     Object.defineProperty(SettingProxy, 'extension1ScriptPath', {
129         enumerable: true,
130         configurable: false,
131         // writable: false,
132         get: function () {
133             // debugger;
134             const root = SettingProxy.GModules["root"];
135             const _ = root.GModules["_"];
136 
137             let info = _.$extension1.info;
138 
139             let path = (defaultSetting.extension1ScriptPath != null) ?
140                 defaultSetting.extension1ScriptPath : info.extension1ScriptPath;
141 
142             return path;
143         },
144         set: function (path) {
145             defaultSetting.extension1Path = path;
146         }
147     });
148 
149     //------------------
150     Object.defineProperty(SettingProxy, 'importScriptList', {
151         enumerable: true,
152         configurable: false,
153         // writable: false,
154         get: function () {
155             return defaultSetting.importScriptList;
156         },
157         set: function () {
158 
159         }
160     });
161 
162 }());
```

```
 1
 2 import { WorkerClass } from './workerProxy_2.js';
 3 import { Job } from './Job_1.js';
 4
 5 ////////////////////////////////////////////////////////////////////////////
 6 //
 7 // 整合所有的 worker
 8 //
 9 ////////////////////////////////////////////////////////////////////////////
10
11 class WorkerManager {
12
13     static get_instance(workerCommand) {
14         if (WorkerManager.instance == null) {
15             WorkerManager.instance = new WorkerManager(workerCommand);
16         }
17         return WorkerManager.instance;
18     }
19     //-------------------------------------------------------------------
20     constructor(root) {
21
22         // 與外界橋接的橋樑
23         this.root = root;
24
25         //
26         this._;
27
28         this.settings;
29
30         this.environment;
31
32         // 記錄有多少在線的 worker
33         this.workers = new Set();
34
35         // 正在閒置中的 workers(等死中)
36         // this.idleWorks = new Set();
37         this.jobList = [];
38
39         // 因應各種環境引入不同的 worker
40         // 一個重要的設計點
41         this.workerClass;
42
43         //----------------------
44         this._getSettings();
45     }
46     //-------------------------------------------------------------------
47     _getSettings() {
48         debugger;
49
50         this.settings = this.root.settings;
51
52         this._ = this.root._;
53
54         if (this._ == null) {
55             throw new Error('no import _');
56             // console.log('no connection with _');
57         }
58
59         if (this._.$extension1 == null) {
60             throw new Error("need import _extension1");
61         }
62
```

```
63          // 取得環境
64          this.environment = this._.$extension1.info.environment;
65
66          //-----------------------
67          // 取得適合當前環境下的 workerClass
68
69          switch (this.environment) {
70              case "nodejs":
71                  this.workerClass = WorkerClass.NodeJsWorkerProxy;
72                  break;
73              default:
74                  this.workerClass = WorkerClass.WebWorkerProxy;
75                  break;
76          }
77      }
78      //-----------------------------------------------------------------
79      addJob(args) {
80          // debugger;
81          // console.log('--------------');
82          console.log('WorkerManager > 加入新工作');
83
84          let job = new Job(this, args);
85
86          // 把任務加入
87          this.jobList.unshift(job);
88          //----------------------------
89          // 檢查是否有 idle worker
90          this.noticeWorkers_checkJob();
91
92          return job.promise();
93      };
94      //-----------------------------------------------------------------
95      // 預先不會在一開始啟動 workers
96      // 通常只有在有指令後才會有 workers 待命
97      // 不過可以事先就請 workers 待命
98      initWorkers(count) {
99          debugger;
100
101         const min_workers = this.settings.min_workers;
102         const max_workers = this.settings.max_workers;
103
104         if(count > max_workers){
105             throw new Error('initWorkers.count > max_workers(${max_workers})');
106         }
107
108         for (let i = 0; i < count; i++) {
109             debugger;
110
111             if(this.workers.size >= max_workers){
112                 break;
113             }
114
115             let employment = false;
116             if (this.workers.size < min_workers) {
117                 // 正職還有缺額
118                 employment = true;
119             }
120             new this.workerClass(this, employment);
121         }
122     }
123     //-----------------------------------------------------------------
124     // 針對 node.js
```

```
125     terminateAllWorkers(){
126
127     }
128     //--------------------------------------------------------------------
129     // 請工作人員注意是否有工作進來
130     // worker: {worker: 由 worker 呼叫 , null: 由 manager 呼叫}
131     noticeWorkers_checkJob(worker) {
132         if (worker == null) {
133             // console.log('check by manager');
134         } else {
135             console.log('check by worker(%s)', worker.id);
136         }
137
138
139         while (this.jobList.length > 0) {
140
141             console.log('still have jobs(%s)', this.jobList.length);
142
143             // 盡可能招募 worker 來接工作
144             let w = this._checkIdleWorker();
145
146             if (w) {
147                 let job = this.jobList.pop();
148                 w.takeJob_callByManager(job);
149             } else {
150                 // 若找不到可用的 worker 作罷
151                 break;
152             }
153         }
154
155
156     }
157     //--------------------------------------------------------------------
158     // 新增 worker(由 worker 自己通報)
159     addWorker(workerProxy) {
160         this.workers.add(workerProxy);
161     }
162     //--------------------------------------------------------------------
163     // 移除指定的 worker(由 worker 自己通報)
164     removeWorker(workerProxy) {
165         this.workers.delete(workerProxy);
166     }
167     //--------------------------------------------------------------------
168     // worker 想取得 job
169     getJob_callByWorker = function () {
170         let job = null;
171
172         console.log('有(%s)項工作待領', this.jobList.length);
173
174         if (this.jobList.length > 0) {
175             job = this.jobList.pop();
176         }
177
178         return job;
179     }
180     //--------------------------------------------------------------------
181     // 檢查是否有空閒的 worker
182     _checkIdleWorker() {
183         // debugger;
184
185         console.log('manager find worker');
186
```

```
187         const max_workers = this.settings.max_workers;
188         const min_workers = this.settings.min_workers;
189 
190         let idleWork;
191         // 找尋空嫌中的 worker
192         let idleWorks = this.findIdleWorkers();
193 
194         if (idleWorks.length > 0) {
195 
196             // 優先找正職者
197             idleWorks.some(function (w) {
198                 if (w.employment) {
199                     idleWork = w;
200                     return true;
201                 }
202             });
203 
204             idleWork = idleWork || idleWorks[0];
205 
206             // console.log('manager find idle worker(${idleWork.id})');
207 
208             return idleWork;
209         }
210         //----------------------------
211 
212         // 沒有空閒中的 worker
213         if (this.workers.size < max_workers) {
214             // 沒有閒置的 worker
215             // 但已有的 worker 數量尚未達上限
216 
217             let employment = false;
218 
219             if (this.workers.size < min_workers) {
220                 // 正職還有缺額
221                 employment = true;
222             }
223 
224             // console.log('manager employment new worker(employment: ${employment})');
225 
226             new this.workerClass(this, employment);
227 
228         } else {
229             console.log('manager no find worker');
230         }
231     }
232     //---------------------------------------------------------------------
233     // 找出閒置中的 worker
234     findIdleWorkers() {
235         let workers = Array.from(this.workers);
236 
237         workers = workers.slice();
238 
239         workers = workers.filter(function (w) {
240             if (w.isReady2TakeJob()) {
241                 return true;
242             }
243         });
244         return workers;
245     }
246     //---------------------------------------------------------------------
247     // 取得需要的資訊
248     getAllworkersInfo() {
```

```
249         let all = [];
250         let idle = [];
251         this.workers.forEach(function (w) {
252             all.push(w.id);
253 
254             if (!w.isBusy()) {
255                 idle.push(w.id);
256             }
257         });
258 
259         let jobCount = this.jobList.length;
260         //----------------
261         return {
262             all: all,
263             idle: idle,
264             jobCount: jobCount,
265         }
266     }
267     //-------------------------------------------------------------------
268 }
269 
270 WorkerManager.instance;
271 
272 //=============================================================================
273 WorkerClass.GModules["WorkerManager"] = WorkerManager;
274 Job.GModules["WorkerManager"] = WorkerManager;
275 
276 WorkerManager.GModules = {};
277 
278 export { WorkerManager };
```

```
 1 const _ = window._;
 2 const WorkerClass = {};
 3
 4 WorkerClass.GModules = {};
 5
 6 export { WorkerClass };
 7
 8 let WORKER_UID = 0;
 9
10 // 抽象類
11 class WorkerProxy {
12     // employment: 是否是僱員
13     constructor(manager, employment) {
14
15         this._;
16
17         this.manager = manager;
18
19         this.sourceScriptPath;
20
21         this.extension1Path;
22
23         this.idleTime;
24
25         this.workerUrl;
26
27         // 需要被 worker 導入的 scripts
28         this.importScriptList = [];
29         //------------------
30         this.id = WORKER_UID++;
31
32         // 當前任務
33         this.job;
34
35         // worker
36         this.worker;
37
38         this.timeHandle;
39         //------------------
40         this.flag_busy = false;
41
42         // 旗標
43         this.flag_inited = false;
44
45         // 是否是正職者
46         this.employment = !!employment;
47
48         // this.flag_waitCount = 0;
49
50         //------------------
51         this.event_end;
52         this.event_error;
53         //------------------
54
55         this._init();
56     }
57     //-----------------------------------------------------------
58     _init() {
59         this._getSettings();
60
61         this.initWorker();
62     }
```

```
 63     //-----------------------------------------------------------
 64     _getSettings() {
 65         // debugger;
 66 
 67         this._ = this.manager._;
 68 
 69         if (this._.$extension1 == null) {
 70             throw new Error("need import _extension1");
 71         }
 72 
 73         const info = _.$extension1.info;
 74         const settings = this.manager.settings;
 75 
 76         // sourceScriptPath
 77         this.sourceScriptPath = settings.sourceScriptPath;
 78 
 79         // extensionPath1
 80         this.extension1Path = settings.extension1ScriptPath;
 81 
 82         this.importScriptList = settings.importScriptList.slice();
 83 
 84         if (this.extension1Path) {
 85             this.importScriptList.unshift(this.extension1Path);
 86         } else {
 87             throw new TypeError('no _extension1 path');
 88         }
 89 
 90         if (this.sourceScriptPath) {
 91             this.importScriptList.unshift(this.sourceScriptPath);
 92         } else {
 93             throw new TypeError('no set (lodash|underscore) path');
 94         }
 95     }
 96     //-----------------------------------------------------------
 97     initWorker() {
 98         console.log('manager employment new worker(%s), employment: %s', this.id,
this.employment);
 99 
100         this.manager.addWorker(this);
101         this._initWorker();
102     }
103 
104     // @override
105     _initWorker(){
106         throw new Error('need override _initWorker');
107     }
108 
109     //-----------------------------------------------------------
110     // @override
111     _event_getEndEvent() {
112         throw new Error('need override _event_getEndEvent');
113     }
114     //-----------------------------------------------------------
115     // @override
116     _event_getErrorEvent() {
117         throw new Error('need override _event_getErrorEvent');
118     }
119     //-----------------------------------------------------------
120     // API
121     // CEO 請他接下一個任務
122     takeJob_callByManager(job) {
123         debugger;
```

```
124
125         // reset
126         // this.flag_waitCount = 0;
127
128         if (this.timeHandle) {
129             // 臨時僱員，正在閒置中
130
131             console.log('worker(%s)正在 idle，被 manager 叫來接工作', this.id);
132
133             clearTimeout(this.timeHandle);
134             this.timeHandle = undefined;
135         }else{
136             // 正職員工
137             console.log('mananger 指派 worker(%s)接工作', this.id);
138         }
139
140         // 非同步
141         // 執行任務
142         this._doJob(job);
143
144         // 是否還有多的工作
145         // 若有再看能否多請人來接
146         // this.manager.noticeWorkers_checkJob(this);
147
148     };
149     //-----------------------------------------------------------------
150     // API
151     takeJob_callBySelf() {
152         debugger;
153
154         console.log('worker(%s)自己檢查是否有工作可拿', this.id);
155
156         // 檢查是否有任務
157         let job = this.manager.getJob_callByWorker();
158
159         if (job == null) {
160
161             if (this.employment) {
162                 console.log('沒工作可拿，正職 worker(%s)，等待', this.id);
163             } else {
164                 console.log('沒工作可拿，兼職 worker(%s)沒工作，進入 idle', this.id)
165                 this._idle();
166             }
167         } else {
168
169             // 非同步
170             // 執行任務
171             this._doJob(job);
172
173             // 是否還有多的工作
174             // 若有再看能否多請人來接
175             // this.manager.noticeWorkers_checkJob(this);
176         }
177     }
178     //-----------------------------------------------------------------
179     // 執行任務
180     _doJob(job) {
181         this.flag_busy = true;
182
183         this.job = job;
184         // debugger;
185
```

```
186         let command = this.job.getCommand();
187         command.id = this.id;
188 
189         console.log('worker(%s)接任務(%s)', this.id, this.job.id);
190 
191         // 請 worker 工作
192         this.worker.postMessage(command);
193     };
194     //-----------------------------------------------------------------
195     // 進入 idle 狀態
196     // 若已在 idle 狀態中，就不動作
197     _idle() {
198         // this.waitDeadth = true;
199         debugger;
200 
201         const idleTime = this.manager.settings.idleTime;
202 
203         console.log('worker(%s)進入 idle(%sms)', this.id, idleTime);
204 
205         if (this.timeHandle != null) {
206             clearTimeout(this.timeHandle);
207             this.timeHandle = undefined;
208             console.log('worker(%s)問題點', this.id);
209         }
210         let $this = this;
211 
212         this.timeHandle = setTimeout(function () {
213             $this.timeHandle = undefined;
214             $this.terminate();
215         }, idleTime);
216     }
217     //-----------------------------------------------------------------
218     terminate() {
219         console.log('worker(%s) will terminate', this.id);
220 
221         let w_info = this.manager.getAllworkersInfo();
222 
223         console.log(JSON.stringify(w_info));
224 
225         let all = w_info.all;
226         let idle = w_info.idle;
227 
228         if (all.length >= 2 && idel.length == 1) {
229             // 大家都在忙，只剩我一個人有空
230             // 不要終結自己，等待工作
231             // 當事情很多時
232             // 多出一個閒置的 worker 在等待接工作
233             console.log('worker(%s) 大家都在忙，有空的只剩我一個人，在加班一下，再進入 idle
等工作', this.id);
234             this._idle();
235             return;
236         }
237         //-----------------------
238         console.log("worker(%s) terminate", this.id);
239         this.manager.removeWorker(this);
240 
241         this.closeWorker();
242     }
243     //-----------------------------------------------------------------
244     // @override
245     closeWorker() {
246         throw new Error('need override workerProxy.closeWorker()');
```

```
247     }
248     //-------------------------------------------------------------------
249     isReady2TakeJob() {
250         return (this.flag_inited && !this.flag_busy);
251     }
252     //-------------------------------------------------------------------
253     isBusy() {
254         return (!this.flag_inited || this.flag_busy);
255     }
256 }
257 //=============================================================================
258 // browser 環境用下用的 worker
259 class WebWorkerProxy extends WorkerProxy {
260     constructor(manager, employment) {
261         super(manager, employment);
262     }
263     //-----------------------------------------------------------
264     // @override
265     closeWorker() {
266         this.worker.removeEventListener('message', this._event_getEndEvent());
267         this.worker.removeEventListener('error', this._event_getErrorEvent());
268 
269         this.worker.terminate();
270         this.worker = undefined;
271     }
272     //-----------------------------------------------------------
273     _getWorkerUrl() {
274         // debugger;
275 
276         let workerContent;
277 
278         if (WorkerProxy.workerContent == null) {
279 
280             let fn = this.getWorkerFnContent();
281 
282             workerContent = fn.toString();
283 
284             // console.log(workerContent);
285 
286             let reg = /^[^{]+\{([\s\S]*)\}[^}]*$/;
287             let res = reg.exec(workerContent);
288             workerContent = res[1];
289 
290             WorkerProxy.workerContent = workerContent;
291         } else {
292             workerContent = WorkerProxy.workerContent;
293         }
294 
295         // 注入 importScriptList
296 
297         let scriptList = JSON.stringify(this.importScriptList);
298 
299         workerContent = 'const scriptList = ${scriptList};
300         ${workerContent}';
301 
302         // console.log(workerContent);
303 
304         let bb = new Blob([workerContent]);
305 
306         return URL.createObjectURL(bb);
307     }
308     //-----------------------------------------------------------
```

```
309     // @override
310     _initWorker() {
311
312         // debugger;
313
314         console.log('新進員工準備中');
315         this.workerUrl = this._getWorkerUrl();
316
317         this.worker = new Worker(this.workerUrl);
318
319         this.worker.addEventListener('error', this._event_getErrorEvent());
320         this.worker.addEventListener('message', this._event_getEndEvent());
321
322     }
323     //---------------------------------------------------------
324     // @override
325     _event_getEndEvent() {
326
327         if (this.event_end == null) {
328             // worker 工作完會呼叫此
329             this.event_end = (function (e) {
330                 // debugger;
331
332                 this.flag_busy = false;
333                 //----------------------------
334                 let data = e.data || {};
335                 let res;
336
337                 if (this.flag_inited) {
338                     // worker 已經初始化過
339
340                     if (this.job) {
341                         // 等待 worker 的工作完成
342                         console.log('worker(%s) job finished', this.id);
343
344                         let job = this.job;
345                         this.job = undefined;
346                         job.resolve(data.res);
347                     }
348
349                 } else {
350                     // worker 尚在初始化中
351                     if (this.job) {
352                         throw new Error('worker(${this.id}) get job but not initialize yet');
353                     }
354
355                     if (data.initialized != null && data.initialized) {
356                         // worker 傳來初始化的消息
357                         console.log('新員工(%s)準備好了', this.id);
358                         this.flag_inited = true;
359                     }
360                 }
361
362                 this.takeJob_callBySelf();
363             }).bind(this);
364         }
365
366         return this.event_end;
367     }
368     //---------------------------------------------------------
369     // @override
370     _event_getErrorEvent() {
```

```
371
372         if (this.event_error == null) {
373             // worker error 完會呼叫此
374             this.event_error = (function (e) {
375                 let job = this.job;
376                 this.job = undefined;
377                 this.flag_busy = false;
378
379                 if (job) {
380                     // 等待 worker 的工作錯誤
381                     job.reject(e);
382                 }
383
384                 console.log('worker(%s) error', this.id);
385
386                 // fix
387                 // 如何處理發生錯誤
388                 this.takeJob_callBySelf();
389
390             }).bind(this);
391         }
392         return this.event_error;
393     }
394     //-----------------------------------------------------------
395     // worker 的內文
396     getWorkerFnContent() {
397         return function () {
398             ////////////////////////////////////////////////////////////////////
399             //
400             // _.worker() 本體
401             //
402             ////////////////////////////////////////////////////////////////////
403             debugger;
404
405             // console.log('i am worker');
406
407             // console.log('href=(%s)', location.href);
408
409             // here
410             // let scriptList = "@@_scriptList_@@";
411
412
413             scriptList.forEach(function (scriptPath) {
414                 try {
415                     importScripts(scriptPath);
416                 } catch (error) {
417                     throw new Error('script(' + scriptPath + ')  load error');
418                 }
419             });
420
421             if (self._ == null) {
422                 throw new Error('(lodash|underscore) load error');
423             }
424
425             const $_ = self._;
426             //================================================
427             self.addEventListener('message', function (e) {
428                 // debugger;
429
430                 let data = e.data || {};
431                 //----------------------------
432                 // 命令
```

```
433                 let command = data['command'] || '';
434
435                 // 參數
436                 let args = data.args || [];
437
438                 let id = data.id;
439                 let jobID = data.jobID;
440
441                 //----------------------------
442                 // console.log('*********** in worker env >> worker(%s) start job(%s)', id,
jobID);
443
444                 if ($_ == null) {
445                     throw new Error('(lodash|underscore) load error');
446                 } else {
447                     // worker 接運算任務
448                     // debugger;
449
450                     if (!command && typeof $_[command] != 'function') {
451                         throw new TypeError('_ no this function');
452                     }
453                     // debugger;
454                     // _ 的運算
455                     let res = $_[command].apply($_, args);
456                     //----------------------------
457                     forTest(res, true);
458                 }
459
460                 function forTest(res, test) {
461
462                     if (test) {
463                         setTimeout(function () {
464                             // console.log('*********** in worker env >> worker(%s) finish
job(%s)', id, jobID);
465                             console.log('---------------');
466                             self.postMessage({
467                                 res: res
468                             });
469                         }, 1000);
470                     } else {
471                         // console.log('*********** in worker env >> worker(%s) finish
job(%s)', id, jobID);
472                         console.log('---------------');
473                         self.postMessage({
474                             res: res
475                         });
476                     }
477                 }
478             });
479             //================================================
480
481             // 通知已初始化完畢
482             self.postMessage({
483                 initialized: true
484             });
485         }
486     }
487 }
488
489 WorkerClass['WebWorkerProxy'] = WebWorkerProxy;
490 //============================================================================
491 // node.js 環境下用的 worker
```

```
492 class NodeJsWorkerProxy extends WorkerProxy {
493     constructor(manager, employment) {
494         super(manager, employment);
495     }
496     //-----------------------------------------------------------
497     _getWorkerUrl() {
498         let $extension1_path = this._.$extension1.info.extension1ScriptPath;
499 
500         // 返回 worker 的 realPath
501         // 必須在 extension 目錄下 /worker/worker 放置 nodeJsWorker.js 檔
502         this.workerUrl = $extension1_path + '/worker/worker/nodeJsWorker.js';
503     }
504     //-----------------------------------------------------------
505     _initWorker() {
506         super._initWorker();
507 
508         const importScriptList = this.importScriptList;
509 
510         this.worker = new Worker(this.workerUrl, {
511             workerData: {
512                 importScriptList: importScriptList
513             }
514         });
515 
516         this.worker.on('error', this._event_getErrorEvent());
517         this.worker.on('message', this._event_getEndEvent());
518 
519     }
520     //-----------------------------------------------------------
521     _event_getEndEvent() {
522     }
523     //-----------------------------------------------------------
524     _event_getErrorEvent() {
525 
526     }
527 }
528 WorkerClass['NodeJsWorkerProxy'] = NodeJsWorkerProxy;
```