

目录

1.命名规范.....	2
1.1 类名/脚本文件名.....	2
1.2 非类的全局变量/局部变量/成员变量/成员函数.....	3
1.2 所有的常量:成员/全局/局部.....	3
1.4 局部/成员名字应该尽量短.....	3
1.5 临时变量:for 循环或者临时块内的变量.....	3
1.6 普通类名和文件名一致.....	3
1.7 方法前缀定义原则。.....	3
1.8 事件响应函数使用同样格式.....	3
1.9 枚举常量.....	4
2. 注释和说明.....	4
2.1 一个文件开头, 必须有这个文件大致作用的描述的注释.....	4
2.2 以及固定格式的创建时间, 作者, 修改时间(用脚本插件 fileheader 生成).....	4
2.3 方法注释.....	4
2.4 修改公用模块.....	5
2.5 svn 和 git 提交记录按固定格式.....	5
2.6 不能太久不提交代码, 也不能提交太频繁.....	5
2.8 git 和 svn 冲突.....	5
3.书写规范.....	5
3.1 常用流程结构使用固定的代码片段.....	5
3.2 对齐和缩进.....	6
3.3 一行代码不能超过半屏幕, 超过屏幕宽度一半的行需要断行, 易于阅读.....	6
3.4 任何时候, if else 语句需要用{}完整表达, 不能省略{}.....	7
3.5 三目运算符.....	7
超过 1 个条件的三目运算符, 都需要使用括号来区分逻辑。.....	7
3.6 在需要以{}闭合的代码段前增加换行.....	7
3.7 全局配置常量.....	8
3.8 全局环境.....	9
4. 变量有效性判断.....	9
4.1 函数参数有效性判断和参数默认值。.....	9
4.2 变量判断逻辑有序, 不用省这个代码量.....	10
4.3 函数指针判断逻辑有序。.....	10
5.代码效率相关, 固定写法, 规避出错, 提高效率.....	10
5.1 避免重复读取属性.....	10
5.2 避免循环里重复计算.....	11
5.3 循环查找的时候.....	11
5.4 跳出多重循环.....	11
5.5 后端数据一定要进行合法性判断.....	12
5.7 禁止使用 eval.....	12
5.9 函数长度控制.....	13
5.8 语句末尾尽量加分号.....	13
5.10 布尔变量的定义.....	13

5.11 self 和 this.....	13
5.12 UI 和数据分开.....	14
5.13 如何调用别的脚本.....	14
5.14 编辑器中 UI 组件和控件名字.....	14
5.15 事件/通知 管理.....	14
8. 客户端自测标准.....	15
8.1 cs 交互：数据模拟.....	15
8.2 正常功能逻辑流程自测。.....	15
8.3 需求功能点核对.....	15
9. 项目和引擎相关。.....	15
9.1 cocos 引擎代码.....	15
9.3 代码结构和项目管理.....	15
9.3 creator 项目 svn 提交原则.....	16
附录：.....	16

1.命名规范

总体原则：







变量作用域越大，名字应该越长，因为生命周期一般更长

单词应该尽量常用的，甚至拼音，不要生僻的单词

尽量使用驼峰式，名字能看出变量和方法作用是最好的命名方法

1.1 类名/脚本文件名

一律首字母大写，驼峰写法。

名称	修改日期	类型	大小
 AcademyProxy.js	2019/10/24 22:52	JScript Script 文件	4 KB
 AcademyProxy.js.meta	2019/12/2 15:00	META 文件	1 KB
 AchievementProxy.js	2019/10/24 22:52	JScript Script 文件	10 KB
 AchievementProxy.js.meta	2019/12/2 15:00	META 文件	1 KB
 ArborDayProxy.js	2019/10/24 22:52	JScript Script 文件	3 KB
 ArborDayProxy.js.meta	2019/12/2 15:00	META 文件	1 KB

```
function Utils() {
    //-----
    /** 颜色品质的颜色 */
    this.BLACK = cc.Color.BLACK;
    this.WHITE = cc.Color.WHITE;
    this.GREEN = cc.Color.WHITE.fromHEX("#12F849");
}
```

1.2 非类的全局变量/局部变量/成员变量/成员函数

均使用小写字母/下划线开头，驼峰写法。

```
this.copyList = function (dList, s
```

1.2 所有的常量:成员/全局/局部

都使用全大写的方式，单词之间_隔开

```
this.BLACK_GREEN = cc.Color.WHITE.fromHEX("#309c1b");
```

1.4 局部/成员名字应该尽量短

降低代码量同时减少书写累赘，全局变量名字应该尽量长，避免重复，且给别人看的东西应该易懂

局部例子: let getArea, delay

全局例子: let getRectangleArea,

全局搜索的时候，会出现无数个 delay，都是局部变量混合全局变量。

那么 teamDelay 就是更好的全局变量。

1.5 临时变量:for 循环或者临时块内的变量

尽量用 temp, i, 等等简短的名字, 单个单词
这类变量可以随时覆盖，不用担心重复

1.6 普通类名和文件名一致.

原则上一个类对应一个文件，一个文件只做一个事，符合 js 本身一个文件一个模块的概念。

工具文件类除外，工具文件里可以有很多个小类。

比如 Utils 里, 可能有时处理类，层级处理类。

聚合这些类的原因包括，减少文件引用次数，方便使用他们之间的公用变量。

1.7 方法前缀定义原则。

尽量不要重复方法名。比如 resShoot 和 resShootFail
不如 resShootSuc 和 resShootFail 好，有辨识度，方便全局搜

1.8 事件响应函数使用同样格式

onSCGetEmail(); //收到服务器消息

onBtnFlowerClick(); //UI 按钮事件

1.9 枚举常量

需要全局使用的常量，使用

2. 注释和说明

2.1 一个文件开头，必须有这个文件大致作用的描述的注释

如下图的代码片段 ***

2.2 以及固定格式的创建时间，作者，修改时间（用脚本插件 fileheader 生成）

```
/*
 * @Author: wangqiang
 * @Date: 2019-12-06 10:38:41
 * @Last Modified by: wangqiang
 * @Last Modified time: 2019-12-06 10:44:29
 */

/**
 * 游戏测试数据
 */

exports.Config = {
  gameName: "hall",
  manifestDir: "res/raw-assets/project.manifest",
};
```

2.3 方法注释

每个方法的开始如果方法名意思难以表达清楚方法作用，必须有方法作用简短描述, 使用 /**/ 注释方法, 而不是 //

在很多编辑器里， /**/ 注释的内容在使用函数的时候可以显示为函数提示。

```
/*
 * @Author: wangqiang
 * @Date: 2019-12-03 14:54:30
 * @Last Modified by: mikey.zhaopengeng
 * @Last Modified time: 2019-12-03 14:56:31
 */

function testFunc({x = 1, y = 2, z} = {}) {
```

2.4 修改公用模块

必须在提交说明里有所说明，并告知主程

2.5 svn 和 git 提交记录按固定格式

禁止没有说明的提交

[版本] 提交内容

2.6 不能太久不提交代码，也不能提交太频繁

一天一次提交比较合适

2.7 git 或者 svn 提交前检查

每一次 git 和 svn 的提交，都应该是“正确的提交”，能够跑起来游戏，是每一次提交的标准。每个功能模块都应该有自己的开关，如果一个功能模块还没做完提交，保证不会影响别人拉代码后运行游戏。

2.8 git 和 svn 冲突

一般按照规则提交、拉取和合并不会产生太复杂的冲突。如果产生复杂冲突，需要跟冲突的人商量解决冲突，不能全部使用“我的”，或者“他的”来解决冲突。

3. 书写规范

3.1 常用流程结构使用固定的代码片段

不需要补充的情况下，尽量少自己写结构。尽量使用代码片段，保证格式统一固定到编辑器中，自动生成 if else, for, console.log 等常用结构
具体参考 jsCreator.code-snippets

```
"forb":{
  "prefix": "forb",
  "body": [
    "for( let i=0; i< ${1:arr}.length; i++) {",
    "\t let arrItem = ${1:arr}[i]",
    "\t if( arrItem ) {",
    "\t\t $2",
    "\t}",
    "}",
    "$3"
  ],
  "description": "基本for遍历数组"
},
```

3.2 对齐和缩进

对齐统一使用 4 个空格

缩进方式按照块结构。

最后完成一个文件统一使用 `beautify` 插件来格式化文件。

3.3 一行代码不能超过半屏幕，超过屏幕宽度一半的行需要断行，易于阅读

```
name = config.Config.skin + "/prefabs/ui/" + name + "fwefewew \
ewewewewewewewewewewewewewewewewewewewewewewewewewewewew";
var isDealSub = arguments.length > 4 && arguments[4] !== undefined &&
arguments[3] <10? arguments[4] : true;
```

3.4 任何时候，if else 语句需要用{}完整表达，不能省略{}

```
// 除了三目运算，if,else等禁止简写
// 正确的书写
if (true) {
    alert(name);
}
console.log(name);
// 不推荐的书写
if (true)
    alert(name);
console.log(name);
// 不推荐的书写
if (true)
    alert(name);
console.log(name)
```

3.5 三目运算符

超过 1 个条件的三目运算符，都需要使用括号来区分逻辑。

```
this.send = function(data, handler, target) {
    var isImWait = arguments.length > 3 && arguments[3] !== undefined ? arguments[3] :
    var isCode = (arguments.length > 4 && arguments[4] !== undefined) ? arguments[4] :
    try {
        var str = data != null ? data.toJson() : "";
    }
```

3.6 在需要以{}闭合的代码段前增加换行

如：for if

```

// 在需要以{}闭合的代码段前增加换行，如：for if
// 没有换行，小的代码段无法区分
if (wl && wl.length) {
    for (i = 0, l = wl.length; i < l; ++i) {
        p = wl[i];
        type = Y.Lang.type(r[p]);
        if (s.hasOwnProperty(p)) {
            if (merge && type == 'object') {
                Y.mix(r[p], s[p]);
            } else if (ov || !(p in r)) {
                r[p] = s[p];
            }
        }
    }
}
// 有了换行，逻辑清楚多了
if (wl && wl.length) {

    for (i = 0, l = wl.length; i < l; ++i) {
        p = wl[i];
        type = Y.Lang.type(r[p]);

        if (s.hasOwnProperty(p)) {
            // 处理merge逻辑
            if (merge && type == 'object') {
                Y.mix(r[p], s[p]);
            } else if (ov || !(p in r)) {
                r[p] = s[p];
            }
        }
    }
}
}

```

3.7 全局配置常量

都写到 config 文件中，通过 cc.zyConfig 引用。这个文件服务端保存一份。可以通过服务端覆盖前段更新全局配置，不用走热更。属于文件本身的常量，定义为文件本身的局部常量。


```
JS szhall.js JS Config.js X JS lwebsocket.js
assets > Script > hall_sz > static > JS Config.js > ...
1  exports.Config = {
2      testValue:"123",
3      isAutoLogin: false,
4      ip : "192.168.8.185",
5      port : "8088",
6      version : "1.0.0",
7      platform: 'sz',
8      enablesimulator:true, //允许模拟器游戏//
9      GOLDRATE:100, //金币比例//
10     //通用预制件名字//
11     okcanceltip:"szhall/prefabs/okcanceltip",
12     errortip:"szhall/prefabs/errortip",
13 };
14
```

```
cc.zyConfig =require("config").Config
```

3.8 全局环境

尽量不要往全局 cc 里写东西。如果是全局常量，写到 cc.zyConfig 里。
cc 里存放全局单例：工具类，数据类
cc 里存放全局经常使用的对象：网络连接

4. 变量有效性判断

4.1 函数参数有效性判断和参数默认值。

方法 1，参数少于 3 个的时候，采用经典方式

```
let funcTest = function(p1)
{
    if (p1 === void 0) { p1 = null; }
}
```

使用 p1 === void 0 而不是 p1 === undefined

一定返回 undefined，但是 undefined 值是可以赋值的，可能不是 undefined.

方法 2，参数大于 3 个的时候，采用解构赋值加默认值的方式。

优点：不用考虑参数书写顺序

```
function testFunc({x = 1 , y = 2, z} = {}) {
    console.log("x: " + x , "y: " + y, "z: " + z);
}
```

```
testFunc();  
testFunc({x:100,y:200});  
testFunc({x:100});
```

4.2 变量判断逻辑有序，不用省这个代码量

比如判断参数 p 值大于 10

```
if( p && p > 10) //正确  
if(p > 10) //错误
```

4.3 函数指针判断逻辑有序。

通过赋值来间接使用的函数，类似于函数指针，使用之前应该判断是否有效

```
let setHandler = function(t,handler)  
{  
  this.target = t  
  this.handler = handler  
}  
if(this.handler && this.target)  
{  
  this.handler.apply(this.target, param)  
}
```

5.代码效率相关，固定写法，规避出错，提高效率

5.1 避免重复读取属性

如果一个属性大量使用，先把属性取出来赋值给局部变量，重复使用在属性很多的时候，这个开销会很大。好的方式：

```
let name = node.name  
if(name && name.indexOf("like") > 0 && name.indexOf("aa") < 5)  
{  
  let newStr = name  
}
```

式：

不好的方式:

```
if(node.name && node.name.indexOf("like") > 0 && node.name.indexOf("aa") < 5)
{
    let newStr = node.name
}
```

5.2 避免循环里重复计算

固定的值应该放到循环外，赋值给局部变量。比如取时间，然后做很多次运算，那么一次取出来，重复利用更好。同时也避免产生误差。好的方式:

```
var nowTime = globalFun.getClientTime();
if(nowTime > time1){
    let timeStr1 = nowTime * nowTime + (nowTime/2)
}
```

不好的方式:

```
if(globalFun.getClientTime() > time1){
    let timeStr1 = globalFun.getClientTime() * globalFun.getClientTime() +
}
```

5.3 循环查找的时候

找到目标后，及时 break。

```
let target = null
for( let i=0; i< arr.length; i++) {
    if( arr[i].length == 10 ) {
        target = arr[i]
        break;
    }
}
```

5.4 跳出多重循环

js 的 break 一次只能跳出一层循环。
两重循环跳出:

```

var brokeed = false;
for (var i = 0; i < 3; i++) {
  for (var j = 0; j < 3; j++) {
    if (i === 1 && j === 1) {
      brokeed = true;
      break;
    }
    console.log('i=' + i + ',j=' + j);
  }
  if (brokeed) {
    break;
  }
}

```

两重以上跳出：

```

(function () {
  for (var i = 0; i < 3; i++) {
    for (var j = 0; j < 3; j++) {
      if (i === 1 && j === 1) {
        return;
      }
      console.log('i=' + i + ',j=' + j);
    }
  }
})();

```

5.5 后端数据一定要进行合法性判断

避免后端出错，前端跟着报错，可以 return 然后 log

```

var xhr = cc.loader.getXMLHttpRequest(),
xhr.onloadend = function() {
  if (xhr == null || xhr.status == null) {
    cc.warn(url + " request is error!!!")
  }
}

```

5.6 不要使用继承

尽量少使用 ES6, 不要使用 ES7。有些低端安卓机实现不支持，或者支持得不好。比如锤子手机，使用 ES6 的有些方法会崩溃

5.7 禁止使用 eval

需要使用的地方可以替代解决，不能替代的地方需要换静态方式处理。

原因 1: eval 本身效率低，很多时候可以使用 parseJSON 代替

原因 2: 有些平台不支持动态解析，比如微信小游戏，手 Q 等平台。

原因 3: 不方便 debug.

5.9 函数长度控制

局部函数一般不超过 100 行，全局函数不超过 300 行。太短调试麻烦，概念不清晰。太长同样影响阅读，影响调试和修改。基本的原则是：

1. UI 和数据分开。
2. 一个函数只做一件事。

5.8 语句末尾尽量加分号

js 本身支持行末不加分号，但是容易产生歧义，比如：

```
// 本意是a=b.然后执行函数。
a = b      // 赋值
(function(){
  //....
})();      // 自执行函数
// 未加分号，结果被解析成

a = b(function(){//...})(); //将b()()返回的结果赋值给a
```

5.10 布尔变量的定义

不要使用 1, 0，使用 true 和 false 本身，意义更明确。

变量为是或者否，直接定义 true, false, 不要用 1, 0 表示。意义不明确，也很难记 10

var isNewUserWx=0;//是否为新用户，0 为 true, 1 为 false

```
if(isNewUserWx!=1){
  wx.alidSendEvent("zipNew");
}
```

5.11 self 和 this

统一用 self 暂存 this, js 里的 this 有他的作用域局限, 在对象方法里使用回调函数的时候, 调用回调函数之前一定要用 self 保存 this 指针，否则回调里的 this 会指向错误。

```
var self = this;

ws.onopen = function (event) {
  console.log("wqinfo socket opened.");
  self.reConCount = 0;
  self.state = 'connected';
}
```

5.12 UI 和数据分开

UI 文件包括编辑器 UI 文件、手动生成的 ui。

数据包括后端数据、前端存储的数据。

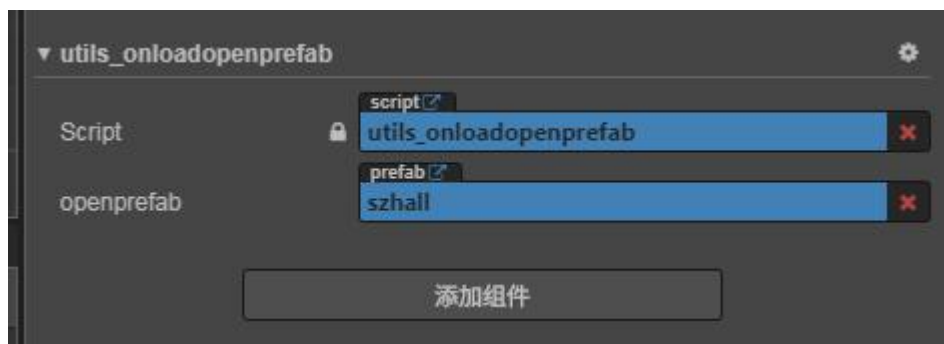
数据的获取和分发是单独的模块。对外通过抛事件通知 UI 进行更新，相反 UI 修改数据应该通知到数据类，数据类进行数据修改。数据和 UI 应该严格分开。

5.13 如何调用别的脚本

全局使用的工具类, 第三方类, 在 launch 里设置为 `cc.xxxx = require("xxxx")`

```
7      },
8      onLoad: function () {
9          console.log("onLoad launch:");
10         //加载通用的功能cc.//
11         cc.zyConfig =require("config").Config
```

一类 UI 会用到的公用 UI 脚本，用公用脚本挂载的形式挂到 UI 上。



5.14 编辑器中 UI 组件和控件名字

应该和 js 中引用的名字相同。不同的名字容易混淆



5.15 事件/通知 管理

使用全局事件管理封装 facade 类，不使用 creator 自带的 on, off 机制
方便局部统一事件释放

方便所有的事件处理，比 on, off 更全面

所有非全局监听，都要在场景 destory 里清理

8. 客户端自测标准

8.1 cs 交互：数据模拟

所有前后端协议，交付策划之前，都需要协议自测通过，所有协议都可以模拟数据。

优点 1：使用测试数据专门类，TestData 类专门用来存放测试数据。方便重复使用。

优点 2：在出现 bug 以后，方便重现 bug，而不需要后端提供数据支持，以及调 gm 达到相应条件才能复现 bug

优点 3：提高代码质量，减少出现 bug 的几率。

8.2 正常功能逻辑流程自测。

功能模块，基本功能必须自测通过。

比如一个转盘活动。那么应该有如下测试：

打开界面，点转盘开始转动，等待转完后获得奖励及其动画，获得奖励显示，奖励获取后检查背包是否获取成功。

8.3 需求功能点核对

是否策划案上要求的主要模块，都已经完成，必须和策划完成这个交互确认，才能提交版本给策划验收。

9. 项目和引擎相关。

9.1 cocos 引擎代码

不能修改。遇到一定要改引擎的地方，应该告知主程，一起商量后所有人一起修改。禁止自行修改 cocos 引擎，而没有同步到其他同事

9.2 封装库代码

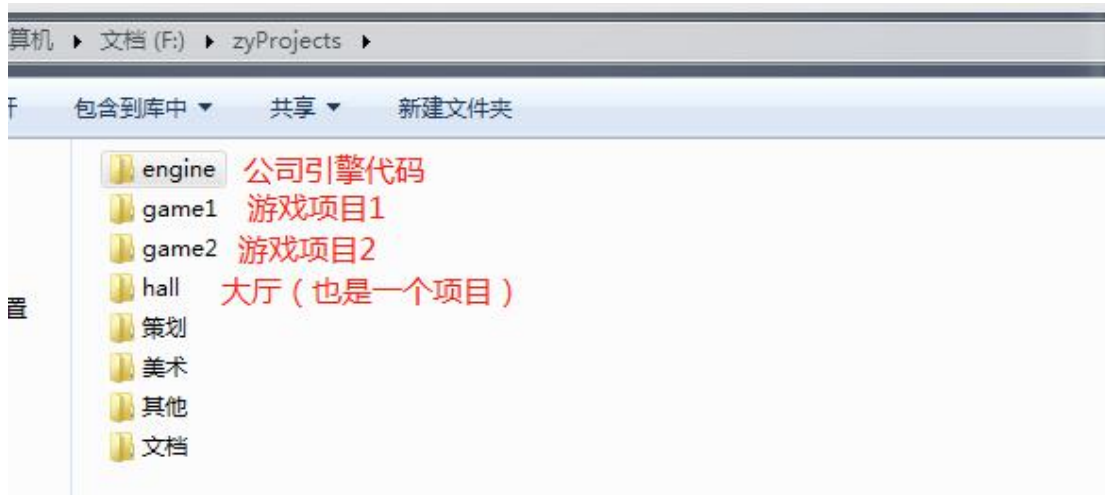
公司所有游戏公用的库代码（engine），和 cocos 引擎代码类似，也不能自己通过 git 提交。需要同步到主程，每个版本结束的时候，一起提交一次引擎代码

9.3 代码结构和项目管理

公司引擎代码包括公共的工具类代码，公共的资源，公共的场景和 sdk.用 git 管理版本所有游戏项目包括大厅都是一个单独的 cocosCreator 项目。用 svn 维护管理版本。

每个游戏项目可以作为一个游戏项目单独运行，也可以通过大厅整合到一起，在大厅里作为一个模块提供入口调用。每个游戏项目单独有自己的热更新，项目之间是平等的，都可以通过大厅调用。

从引擎代码到游戏项目拷贝代码和资源是单向的，只有主程有权限反向拷贝，以及提交引擎代码的权限。



9.3 creator 项目 svn 提交原则

功能开发只能提交 assets 目录内容

切记不能提交 build 目录、temp 目录, 以及其他的如.vscode 等编辑器相关目录
svn 每次提交必须按[版本] 提交内容概略 的格式

附录:

使用统一且常用的单词, 宁愿使用拼音也尽量少使用偏僻词:

get 获取 / set 设置,
add 增加 / remove 删除
create 创建 / destory 移除
start 启动 / stop 停止
open 打开 / close 关闭,
read 读取 / write 写入
load 载入 / save 保存,
create 创建 / destroy 销毁
begin 开始 / end 结束,
backup 备份 / restore 恢复
import 导入 /
export 导出,
split 分割 / merge 合并
inject 注入 / extract 提取,
attach 附着 / detach 脱离
bind 绑定 / separate 分离,
view 查看 / browse 浏览

edit 编辑 / modify 修改,
select 选取 / mark 标记
copy 复制 / paste 粘贴,
undo 撤销 / redo 重做
insert 插入 / delete 移除,
add 加入 / append 添加
clean 清理 / clear 清除,
index 索引 / sort 排序
find 查找 / search 搜索,
increase 增加 / decrease 减少
play 播放 / pause 暂停,
launch 启动 / run 运行
compile 编译 / execute 执行,
debug 调试 / trace 跟踪
observe 观察 / listen 监听,
build 构建 / publish 发布
input 输入 / output 输出,
encode 编码 / decode 解码
encrypt 加密 / decrypt 解密,
compress 压缩 / decompress 解压缩
pack 打包 / unpack 解包,
parse 解析 / emit 生成
connect 连接 / disconnect 断开,
send 发送 / receive 接收
download 下载 / upload 上传,
refresh 刷新 / synchronize 同步
update 更新 / revert 复原,
lock 锁定 / unlock 解锁
check out 签出 / check in 签入,
submit 提交 / commit 交付
push 推 / pull 拉,
expand 展开 / collapse 折叠
begin 起始 / end 结束,
start 开始 / finish 完成
enter 进入 / exit 退出,
abort 放弃 / quit 离开
obsolete 废弃 / depreciate 废旧,
collect 收集 / aggregate 聚集

