

Cocos Creator 开发工具平台

(1.6+)

开 发 手 册

贝密游戏整理（来自于网络）

二〇一七年九月

目录

目录.....	2
第一章 JavaScript 快速入门.....	8
1.1 变量.....	8
1.2 函数.....	8
1.3 返回值.....	9
1.4 if/else 语句.....	9
1.5 JavaScript 数组 (Array)	10
1.6 JavaScript 对象 (Object)	10
1.7 匿名函数.....	12
1.8 链式语法.....	13
1.9 This.....	14
1.10 运算符	14
1.11 总结	15
第二章 创建和使用组件脚本	18
2.1 创建组件脚本.....	18
2.2 编辑脚本	19
2.3 添加脚本到场景节点中.....	19
第三章 使用 cc.Class 声明类型.....	21
3.1 定义 CCClass.....	21
3.2 实例化	21
3.3 判断类型	21

3.4 构造函数	21
3.5 实例方法	22
3.6 继承	22
3.7 声明属性	23
第四章 CCClass 进阶参考	29
术语.....	29
4.1 原型对象参数说明.....	29
4.2 判断类型	31
4.3 成员	32
4.4 继承	35
4.5 属性.....	38
4.6 属性参数	39
4.7 GetSet 方法.....	49
4.8 editor 参数	52
第五章 访问节点和组件.....	55
5.1 获得组件所在的节点.....	55
5.2 获得其它组件.....	55
5.3 获得其它节点及其组件.....	57
5.4 访问已有变量里的值.....	61
第六章 常用节点和组件接口	64
6.1 节点状态和层级操作.....	64
6.2 更改节点的变换（位置、旋转、缩放、尺寸）	65

6.3 颜色和不透明度	66
6.4 常用组件接口	67
第七章 生命周期回调.....	68
onLoad	68
start.....	69
update	70
lateUpdate	70
onEnable.....	71
onDisable.....	71
onDestroy	71
第八章 创建和销毁节点.....	72
8.1 创建新节点.....	72
8.2 克隆已有节点	72
8.3 创建预置节点	73
8.4 销毁节点	74
第九章 加载和切换场景.....	76
9.1 通过常驻节点进行场景资源管理和参数传递	76
9.2 场景加载回调	76
9.3 预加载场景	77
第十章 监听和发射事件.....	77
10.1 监听事件	77
10.2 发射事件.....	79

10.3 派送事件.....	80
10.4 事件对象.....	81
第十一章 系统内置事件.....	82
11.1 鼠标事件类型和事件对象.....	82
11.2 触摸事件类型和事件对象.....	84
11.3 鼠标和触摸事件冒泡.....	86
11.4 cc.Node 的其它事件.....	87
第十二章 玩家输入事件.....	88
12.1 如何定义输入事件.....	88
第十三章 在 Cocos Creator 中使用动作系统.....	94
13.1 动作系统简介.....	94
13.2 动作系统 API.....	94
13.3 动作类型.....	95
13.4 动作列表.....	99
基础动作类型.....	100
容器动作.....	100
即时动作.....	100
时间间隔动作.....	101
缓动动作.....	103
第十四章 使用计时器.....	105
第十五章 脚本执行顺序.....	107
15.1 使用统一的控制脚本来初始化其他脚本.....	107

15.2 在 Update 中用自定义方法控制更新顺序.....	108
15.3 控制同一个节点上的组件执行顺序	108
第十六章 标准网络接口.....	111
16.1 使用方法	111
16.2 SocketIO	112
第十七章 使用对象池.....	115
17.1 对象池的概念	115
17.2 流程介绍.....	115
17.3 使用组件对象.....	118
17.4 清除对象池.....	119
限制.....	119
第十八章 获取和加载资源	120
18.1 资源的分类	120
18.2 如何在属性检查器里设置资源	122
18.3 动态加载.....	124
第十九章 模块化脚本.....	129
19.1 引用模块	130
19.2 定义模块.....	131
19.3 更多示例.....	134
第二十章 插件脚本.....	137
脚本加载顺序.....	138
目标平台兼容性.....	138

全局变量.....	139
第二十一章 第三方 JavaScript 模块引用.....	142
21.1 如何使用 npm 模块.....	142
注意事项.....	142
21.2 未来其他可能的模块依赖方式	143

第一章 JavaScript 快速入门

1.1 变量

在 JavaScript 中，我们像这样声明一个变量：

```
var a;
```

保留字 `var` 之后紧跟着的，就是一个变量名，接下来我们可以为变量赋值：

```
var a = 12;
```

在阅读其他人的 JavaScript 代码时，你也会看到下面这样的变量声明：

```
a = 12;
```

如果你在浏览器控制台中尝试，会发现 JavaScript 在面对省略 `var` 时的变量声明并不会报错，但在 Cocos Creator 项目脚本中，声明变量时的 `var` 是不能省略的，否则编译器会报错。

1.2 函数

在 JavaScript 里我们像这样声明函数：

```
var myAwesomeFunction = function (myArgument) {  
    // do something  
}
```

像这样调用函数：

```
myAwesomeFunction(something);
```

在 JavaScript 里，函数和变量本质上是一样的，我们可以像下面这样把一个函数当做参数传入另一个函数中：

```
square = function (a) { return a * a; }
```

```
applyOperation = function (f, a) {
```



```
    return f(a);  
}  
  
applyOperation (square, 10); // 100
```

1.3 返回值

函数的返回值是由 `return` 打头的语句定义的，我们这里要了解的是函数体内 `return` 语句之后的内容是不会被执行的。

```
myFunction = function (a) {  
    return a * 3;  
    explodeComputer(); // will never get executed (hopefully!)  
}
```

1.4 if/else 语句

JavaScript 中条件判断语句 `if` 是这样用的：

```
if (foo) {  
    return bar;  
}
```

`if` 后的值如果为 `false`，会执行 `else` 中的语句：

```
if (foo) {  
    function1();  
}  
else {  
    function2();  
}
```

`if/else` 条件判断还可以像这样写成一行：`foo ? function1() : function2();`

当 `foo` 的值为 `true` 时，表达式会返回 `function1()` 的执行结果，反之会返回 `function2()` 的执行结果。当我们需要根据条件来为变量赋值时，这种写法就非常方便：

```
var n = foo ? 1 : 2;
```

上面的语句可以表述为“当 `foo` 是 `true` 时，将 `n` 的值赋为 1，否则赋为 2”。

当然我们还可以使用 `else if` 来处理更多的判断类型：

```
if (foo) {  
    function1();  
}else if (bar) {  
    function2();  
}else {  
    function3();  
}
```

1.5 JavaScript 数组 (Array)

JavaScript 里像这样声明数组：

```
a = [123, 456, 789];
```

像这样访问数组中的成员：（从 0 开始索引）

```
a[1]; // 456
```

1.6 JavaScript 对象 (Object)

我们像这样声明一个对象 (object)：

```
myProfile = {  
  
    name: "Jare Guo",  
  
    email: "blabla@gmail.com",  
  
    'zip code': 12345,  
  
    isInvited: true  
  
}
```

在对象声明的语法 (`myProfile = {...}`) 之中 , 有一组用逗号相隔的键值 对。每一对都包括一个 `key` (字符串类型 , 有时候会用双引号包裹) 和一个 `value` (可以是任何类型 : 包括 `string` , `number` , `boolean` , 变量名 , 数组 , 对象甚至是函数)。我们管这样的一对键值叫做对象的属性 (`property`) , `key` 是属性名 , `value` 是属性值。

你可以在 `value` 中嵌套其他对象 , 或者由一组对象组成的数组 :

```
myProfile = {  
  
    name: "Jare Guo",  
  
    email: "blabla@gmail.com",  
  
    city: "Xiamen",  
  
    points: 1234,  
  
    isInvited: true,  
  
    friends: [  
  
        {  
  
            name: "Johnny",  
  
            email: "blablabla@gmail.com"
```

```

    },
    {
        name: "Nantas",
        email: "piapiapia@gmail.com"
    }
]
}

```

访问对象的某个属性非常简单，我们只要使用 dot 语法就可以了，还可以和数组成员的访问结合起来：

```

myProfile.name;

myProfile.friends[1].name;

```

JavaScript 中的对象无处不在，在函数的参数传递中也会大量使用，比如在 Cocos Creator 中，我们就可以像这样定义 FireClass 对象：

```

var MyComponent = cc.Class({
    extends: cc.Component
});

```

{extends: cc.Component} 这就是一个用做函数参数的对象。在 JavaScript 中大多数情况我们使用对象时都不一定要为他命名，很可能会像这样直接使用。

1.7 匿名函数

当一个函数的参数是也是函数时，当调用这个函数时，在参数的位置直接写了一个新的函数体，这样的做法被称为匿名函数，在 JavaScript 中是最为广泛使用的模式。例如：

```

applyOperation = function (f, a) {
    return f(a);
}

applyOperation(
    function(a){
        return a*a;
    },
    10
) // 100

```

1.8 链式语法

下面我们介绍一种在数组和字符串操作中常用的语法：

```

var myArray = [123, 456];

myArray.push(789) // 123, 456, 789

var myString = "abcdef";

myString.replace("a", "z"); // "zbcdef"

```

上面代码中的点符号表示“调用 myString 字符串对象的 replace 函数，并且传递 a 和 z 作为参数，然后获得返回值”。

使用点符号的表达式，最大的优点是你可以把多项任务链接在一个表达式里，当然前提是每个调用的函数必须有合适的返回值。我们不会过多介绍如何定义可链接的函数，但是使用他们是非常简单的，只要使用以下的模式：

```

something.function1().function2().function3()

```

链条中的每个环节都会接到一个初始值，调用一个函数，然后把函数执行结果传递到下一环节。

1.9 This

简单地说，`this` 关键字表示当前对象：`this` 会随着执行环境的变化而变化。

```
myFunction = function (a, b) {  
    console.log(this);  
}
```

另外一个方法是将 `this` 赋值给另外一个变量：

```
myFunction = function (a, b) {  
    var myObject = this;  
}
```

1.10 运算符

`=` 是赋值运算符，`a = 12` 表示把 “12” 赋值给变量 `a`。

如果你需要比较两个值，可以使用 `==`，例如 `a == 12`。

JavaScript 中还有个独特的 `===` 运算符，它能够比较两边的值和类型是否全都相同。（类型是指 `string`, `number` 这些）：

```
a = "12";  
  
a == 12; // true  
  
a === 12; // false
```

大多数情况下，我们都推荐使用 `===` 运算符来比较两个值，因为希望比较两个不同类型但有着相同值的情况是比较少见的。

下面是 JavaScript 判断两个值是否不相等的比较运算符：

```
a = 12;
```

```
a !== 11; // true
```

! 运算符还可以单独使用，用来对一个 boolean 值取反：

```
a = true;
```

```
!a; // false
```

! 运算符总会得到一个 boolean 类型的值，所以可以用来将非 boolean 类型的值转为 boolean 类型：

```
a = 12;
```

```
!a; // false
```

```
!!a; // true
```

或者：

```
a = 0;
```

```
!a; // true
```

```
!!a; // false
```

1.11 总结

Cocos Creator 脚本代码：

```
var Comp = cc.Class({  
    extends: cc.Component,  
    properties: {  
        target: {  
            default: null,  
            type: cc.Entity
```

```

        }

    },

    onStart: function () {

        this.target = cc.Entity.find('/Main Player/Bip/Head');

    },

    update: function () {

        this.transform.worldPosition = this.target.transform.worldPosition;

    }

});

```

这段代码向引擎定义了一个新组件，这个组件具有一个 `target` 参数，在运行时会初始化为指定的对象，并且在运行的过程中每一帧都将自己设置成和 `target` 相同的坐标。

让我们分别看下每一句的作用（我会高亮有用的语法模式）：

`var Comp = cc.Class({`：这里我们使用 `cc` 这个对象，通过点语法来调用对象的 `Class()` 方法（该方法是 `cc` 对象的一个属性），调用时传递的参数是一个匿名的 JavaScript 对象（`{}`）。

`target: { default: null, type: cc.Entity }`：这个键值对声明了一个名为 `target` 的属性，值是另一个 JavaScript 匿名对象。这个对象定义了 `target` 的默认值和值类型。

`extends: cc.Component`：这个键值对声明这个 `Class` 的父类是

`cc.Component`。`cc.Component` 是 Cocos Creator 的内置类型。

`onStart: function () {`：这一对键值定义了一个成员方法，叫做 `onStart`，他的值是一个匿名函数。

`this.target = cc.Entity.find('`：在这一句的上下文中，`this` 表示正在被创建的
Component 组件，这里通过 `this.target` 来访问 `target` 属性。

第二章 创建和使用组件脚本

2.1 创建组件脚本

在 Cocos Creator 中，脚本也是资源的一部分。你可以在资源编辑器中通过点击"创建"按钮来添加并选择 JavaScript 或者 CoffeeScript 来创建一份组件脚本。此时你会在你的资源编辑器中得到一份新的脚本：



一份简单的组件脚本如下：

```
cc.Class({  
    extends: cc.Component,  
  
    properties: {  
  
    },  
  
    // use this for initialization  
  
    onLoad: function () {
```

```
    },  
  
    // called every frame, uncomment this function to activate update  
    callback  
  
    update: function (dt) {  
  
        },  
  
    });
```

2.2 编辑脚本

Cocos Creator 内置一个轻量级的 Code Editor 供用户进行快速的脚本编辑。

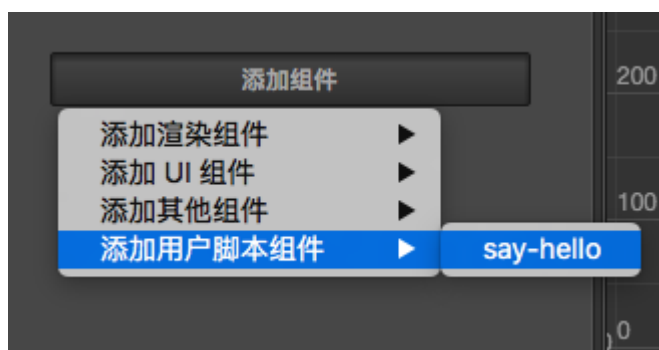
但用户也可以根据自己的需求，选择自己喜爱的文本工具（如：Vim，Sublime Text，Web Storm，Visual Studio...）进行脚本编辑。

通过双击脚本资源，可以直接打开内置的 Code Editor 编辑。如果用户需要使用外部工具，则先在文件系统中定位资源目录，再经由自己的文本工具打开来编辑。

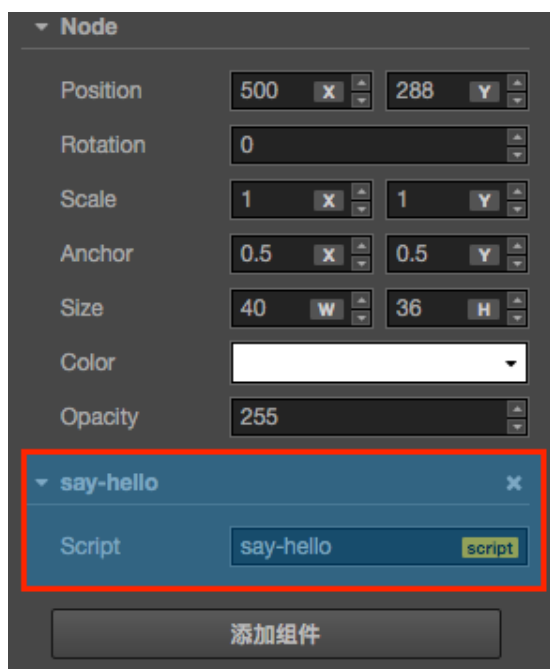
当编辑完脚本并保存，Cocos Creator 会自动检测到脚本的改动，并迅速编译。

2.3 添加脚本到场景节点中

将脚本添加到场景节点中，实际上就是为这个节点添加一份组件。我们先将刚刚创建出来的 “NewScript.js” 重命名为 “say-hello.js”。然后选中我们希望添加的场景节点，此时该节点的属性会显示在 属性检查器 中。在 属性检查器 的最下方有一个“添加组件”的按钮，点击按钮并选择：添加用户脚本 -> say-hello 来添加我们刚刚编写的脚本组件。



如果一切顺利，你将会看到你的脚本显示在 属性检查器 中：



注意：你也可以通过直接拖拽脚本资源到 属性检查器 的方式来添加脚本。

第三章 使用 cc.Class 声明类型

cc.Class 是一个很常用的 API ,用于声明 Cocos Creator 中的类 ,为了方便区分 ,我们把使用 cc.Class 声明的类叫做 CClass。

3.1 定义 CClass

调用 cc.Class ,传入一个原型对象 ,在原型对象中以键值对的形式设定所需的类型参数 ,就能创建出所需要的类。

```
var Sprite = cc.Class({  
    name: "sprite"  
});
```

以上代码用 cc.Class 创建了一个类型 ,并且赋给了 Sprite 变量。同时还将类名设为 "sprite" ,类名用于序列化 ,一般可以省略。

3.2 实例化

Sprite 变量保存的是一个 JavaScript 构造函数 ,可以直接 new 出一个对象 :

```
var obj = new Sprite();
```

3.3 判断类型

需要做类型判断时 ,可以用 JavaScript 原生的 instanceof :

```
cc.log(obj instanceof Sprite); // true
```

3.4 构造函数

使用 ctor 声明构造函数 :

```
var Sprite = cc.Class({  
    ctor: function () {  
        cc.log(this instanceof Sprite);    // true
```

```
    }  
});
```

Component 是特殊的 CCClass ,不能定义构造函数 ,它的构造职能可由 onLoad 方法代替。

3.5 实例方法

```
var Sprite = cc.Class({  
    print: function () {} // 声明一个名叫 "print" 的实例方法  
});
```

3.6 继承

使用 **extends** 实现继承 :

```
var Shape = cc.Class();// 父类  
  
var Rect = cc.Class({ // 子类  
    extends: Shape  
});
```

继承后 , CCClass 会统一自动调用父构造函数 , 你不需要显式调用。

```
var Shape = cc.Class({  
    ctor: function () {  
        cc.log("Shape");    // 实例化时 , 父构造函数会自动调用  
    }  
});  
  
var Square = cc.Class({  
    extends: Shape,
```

```
ctor: function () {  
    cc.log("Square");    // 再调用子构造函数  
}  
});  
  
var square = new Square();
```

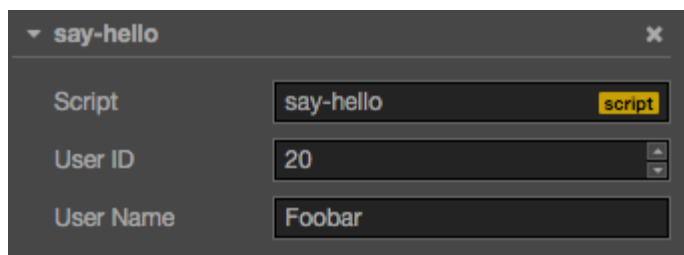
以上代码将依次输出 "Shape" 和 "Square"。

3.7 声明属性

要声明属性，仅需要在 `cc.Class` 定义的 `properties` 字段中，填写属性名字和属性参数即可，如：

```
cc.Class({  
    extends: cc.Component,  
    properties: {  
        userID: 20,  
        userName: "Foobar"  
    }  
});
```

这时候，你可以在 属性检查器 中看到，你刚刚定义的两个属性，显示在检查器的面板中：



在 Cocos Creator 中，我们提供两种形式的属性声明方法：

1、简单声明

当声明的属性为基本 JavaScript 类型时，可以直接赋予默认值：

```
properties: {  
    height: 20,      // number  
    type: "actor",   // string  
    loaded: false,   // boolean  
    target: null,    // object  
}
```

当声明的属性具备类型时（如：cc.Node，cc.Vec2 等），可以在声明处填写他们的构造函数来完成声明，如：

```
properties: {  
    target: cc.Node,  
    pos: cc.Vec2,  
}
```

当声明属性的类型继承自 cc.ValueType 时（如：cc.Vec2，cc.Color 或 cc.Rect），

除了上面的构造函数，还可以直接使用实例作为默认值：

```
properties: {  
    pos: new cc.Vec2(10, 20),  
    color: new cc.Color(255, 255, 255, 128),  
}
```


当声明属性是一个数组时，可以在声明处填写他们的类型或构造函数来完成声明，如：

```
properties: {  
  
    any: [],          // 不定义具体类型的数组  
  
    bools: [cc.Boolean],  
  
    strings: [cc.String],  
  
    floats: [cc.Float],  
  
    ints: [cc.Integer],  
  
  
    values: [cc.Vec2],  
  
    nodes: [cc.Node],  
  
    frames: [cc.SpriteFrame],  
  
}
```

注意：除了以上几种情况，其他类型我们都需要使用完整声明的方式来进行书写。

2、完整声明

有些情况下，我们需要为属性声明添加参数，这些参数控制了属性在 属性检查器 中的显示方式，以及属性在场景序列化过程中的行为。例如：

```
properties: {  
  
    score: {  
  
        default: 0,  
  
        displayName: "Score (player)",  
  
        tooltip: "The score of player",  
  
    },  
  
}
```

```

    }
}

```

以上代码为 `score` 属性设置了三个参数 `default`, `displayName` 和 `tooltip`。这几个参数分别指定了 `score` 的默认值为 0，在 属性检查器 里，其属性名将显示为：“Score (player)”，并且当鼠标移到参数上时，显示对应的 Tooltip。

下面是常用参数：

- `default`: 设置属性的默认值，这个默认值仅在组件第一次添加到节点上时才会用到
- `type`: 限定属性的数据类型，详见 [CCClass 属性类型](#) 文档
- `visible`: 设为 `false` 则不在 属性检查器 面板中显示该属性
- `serializable`: 设为 `false` 则不序列化（保存）该属性
- `displayName`: 在 属性检查器 面板中显示成指定名字
- `tooltip`: 在 属性检查器 面板中添加属性的 Tooltip

更多的属性参数，可阅读 [属性参数](#)

3、数组声明

数组的 `default` 必须设置为 `[]`，如果要在 属性检查器 中编辑，还需要设置 `type` 为构造函数，枚举，或者 `cc.Integer`，`cc.Float`，`cc.Boolean` 和 `cc.String`。

```

properties: {
    names: {
        default: [],
        type: [cc.String] // 用 type 指定数组的每个元素都是字符串类型
    }
}

```

```

    },

    enemies: {
        default: [],
        type: [cc.Node]    // type 同样写成数组，提高代码可读性
    },
}

```

4、get/set 声明

在属性中设置了 get 或 set 以后，访问属性的时候，就能触发预定义的 get 或 set 方法。定义方法如下：

```

properties: {
    width: {
        get: function () {
            return this._width;
        },
        set: function (value) {
            this._width = value;
        }
    }
}

```

如果你只定义 `get` 方法，那相当于属性只读。

如果你只定义 `set` 方法，将不会显示在 属性检查器 中。

第四章 CCClass 进阶参考

术语

- CCClass : 使用 `cc.Class` 声明的类。
- 原型对象 : 调用 `cc.Class` 时传入的字面量参数。
- 实例成员 : 实例成员 ("instance member") 包含“成员变量” (member variable) 和“成员方法” (instance method) 。
- 静态成员 : 静态成员包含“静态变量” (static variable) 和“类方法” (static method) 。
- 运行时 : 项目脱离编辑器独立运行时 , 或者在模拟器和浏览器里预览的时候。
- 序列化 : 解析内存中的对象 , 将它的信息编码为一个特殊的字符串 , 以便保存到硬盘上或传输到其它地方。

4.1 原型对象参数说明

所有原型对象的参数都可以省略 , 用户只需要声明用得到的部分即可。

```
cc.Class({  
  
    // 类名 , 用于序列化。值类型 : String  
  
    name: "Character",  
  
    // 基类 , 可以是任意创建好的 cc.Class。值类型 : Function  
  
    extends: cc.Component,  
  
    // 构造函数。值类型 : Function  
  
    ctor: function () {},  
  
    // 属性定义 ( 方式一 , 直接定义 )  
  
    properties: {  
  
        text: ""
```

```

    },

    // 属性定义 (方式二, 使用 ES6 的箭头函数, 详见下文)
    properties: () => ({
        text: ""
    }),

    // 实例方法
    print: function () {
        cc.log(this.text);
    },

    // 静态成员定义。值类型 : Object
    statics: {
        _count: 0,
        getCount: function () {}
    },

    // 提供给 Component 的子类专用的参数字段
    // 值类型 : Object
    editor: {
        disallowMultiple: true
    }
});

类名

```

类名可以是任意字符串，但不允许重复。可以使用 `cc.js.getClassName` 来获得类名，使用 `cc.js.getClassByName` 来查找对应的类。对在项目脚本里定义的组件来说，序列化其实并不使用类名，因此那些组件不需要指定类名。对其他类来说，类名用于序列化，如果不需要序列化，类名可以省略。

构造函数

为了保证反序列化能始终正确运行，CCClass 的构造函数不允许定义构造参数。

开发者如果确实需要使用构造参数，可以通过 `arguments` 获取，但要记

得如果这个类会被序列化，必须保证构造参数都缺省的情况下仍然能

`new` 出对象。

4.2 判断类型

判断实例

需要做类型判断时，可以用 JavaScript 原生的 `instanceof`：

```
var Sub = cc.Class({
    extends: Base
});

var sub = new Sub();

cc.log(sub instanceof Sub);      // true
cc.log(sub instanceof Base);     // true

var base = new Base();

cc.log(base instanceof Sub);     // false
```

判断类

使用 `cc.isChildClassOf` 来判断两个类的继承关系：

```
var Texture = cc.Class();

var Texture2D = cc.Class({
    extends: Texture
});

cc.log(cc.isChildClassOf(Texture2D, Texture)); // true
```

两个传入参数都必须是类的构造函数，而不是类的对象实例。如果传入的两个类相等，`isChildClassOf` 同样会返回 `true`。

4.3 成员

实例变量

当一个类的性能问题成为局部瓶颈时，可以试着直接在构造函数中定义实例变量。这样定义出来的变量不能被序列化，也不能在 属性检查器 中查看。

```
var Sprite = cc.Class({
    ctor: function () {
        // 声明实例变量并赋默认值

        this.url = "";

        this.id = 0;
    }
});
```

如果是私有的变量，建议在变量名前面添加下划线 `_` 以示区分。

实例方法

实例方法请在原型对象中声明：

```
var Sprite = cc.Class({
```



```

    ctor: function () {
        this.text = "this is sprite";
    },
    // 声明一个名叫 "print" 的实例方法
    print: function () {
        cc.log(this.text);
    }
});

var obj = new Sprite();

obj.print();// 调用实例方法

```

如果是私有的函数，建议在函数名前面添加下划线 `_` 以示区分。

静态变量和静态方法

静态变量或静态方法可以在原型对象的 `statics` 中声明：

```

var Sprite = cc.Class({
    statics: {
        // 声明静态变量
        count: 0,
        // 声明静态方法
        getBounds: function (spriteList) {
            // ...
        }
    }
});

```

```
});
```

上面的代码等价于：

```
var Sprite = cc.Class({ ... });
```

```
// 声明静态变量
```

```
Sprite.count = 0;
```

```
// 声明静态方法
```

```
Sprite.getBounds = function (spriteList) {
```

```
    // ...
```

```
};
```

静态成员会被子类继承，继承时会将父类的静态变量浅拷贝给子类，因此：

```
var Object = cc.Class({
```

```
    statics: {
```

```
        count: 11,
```

```
        range: { w: 100, h: 100 }
```

```
    }
```

```
});
```

```
var Sprite = cc.Class({
```

```
    extends: Object
```

```
});
```

```
cc.log(Sprite.count);    // 结果是 11
```

```
Sprite.range.w = 200;
```

```
cc.log(Object.range.w); // 结果是 200
```

如果你不需要考虑继承，私有的静态成员也可以直接定义在类的外面：

```
function doLoad (sprite) {// 局部方法};
```

```
var url = "foo.png";// 局部变量
```

```
var Sprite = cc.Class({
```

```
    load: function () {
```

```
        this.url = url;
```

```
        doLoad(this);
```

```
    };
```

```
});
```

4.4 继承

父构造函数

不论子类是否有定义构造函数，子类实例化前父类的构造函数都会被自动调用。

```
var Node = cc.Class({
```

```
    ctor: function () {
```

```
        this.name = "node";
```

```
    }
```

```
});
```

```
var Sprite = cc.Class({
```

```
    extends: Node,
```

```
    ctor: function () {
```

```
        // 子构造函数被调用前，父构造函数已经被调用过
```

```
        cc.log(this.name);    // "node"
```

```

        // 重新设置 this.name

        this.name = "sprite";

    }

});

```

```

var obj = new Sprite();

cc.log(obj.name);    // "sprite"

```

因此你不需要尝试调用父类的构造函数，否则父构造函数就会重复调用。

```

var Sprite = cc.Class({

    extends: Node,

    ctor: function () {

        Node.call(this);        // 别这么干！

        this._super();          // 也别这么干！

    }

});

```

重载

所有成员方法都是虚方法，子类方法可以直接重载父类方法：

```

var Shape = cc.Class({

    getName: function () {

        return "shape";

    }

});

```

```

var Rect = cc.Class({
    extends: Shape,
    getName: function () {
        return "rect";
    }
});

var obj = new Rect();

cc.log(obj.getName());    // "rect"

```

和构造函数不同的是，父类被重载的方法并不会被 CClass 自动调用，如果你要调用的话：

方法一：使用 CClass 封装的 `this._super`：

```

var Shape = cc.Class({
    getName: function () { return "shape"; }
});

var Rect = cc.Class({
    extends: Shape,
    getName: function () {
        var baseName = this._super();
        return baseName + " (rect)";
    }
});

var obj = new Rect();

```

```
cc.log(obj.getName());    // "shape (rect)"
```

方法二：使用 JavaScript 原生写法：

```
var Rect = cc.Class({  
    extends: Shape,  
    getName: function () {  
        var baseName = Shape.prototype.getName.call(this);  
        return baseName + " (rect)";  
    }  
});  
  
var obj = new Rect();  
  
cc.log(obj.getName());    // "shape (rect)"
```

如果你想实现继承的父类和子类都不是 CCClass ,只是原生的 JavaScript 构造函数，你可以用更底层的 API cc.js.extend 来实现。

4.5 属性

属性是特殊的实例变量，能够显示在 属性检查器 中，也能被序列化。

属性和构造函数

属性不用在构造函数里定义，在构造函数被调用前，属性已经被赋为默认值了，可以在构造函数内访问到。如果属性的默认值无法在定义 CCClass 时提供，需要在运行时才能获得，你也可以在构造函数中重新给属性赋默认值。

```
var Sprite = cc.Class({  
    ctor: function () {  
        this.img = LoadImage();  
    }  
});
```

```

    },
    properties: {
        img: {
            default: null,
            type: Image
        }
    }
});

```

不过要注意的是,属性被反序列化的过程紧接着发生在构造函数执行之后,因此构造函数中只能获得和修改属性的默认值,还无法获得和修改之前保存的值。

4.6 属性参数

所有属性参数都是可选的,但至少必须声明 `default`, `get`, `set` 参数中的其中一个。

default 参数

`default` 用于声明属性的默认值,声明了默认值的属性会被 `CCClass` 实现为成员变量。默认值只有在第一次创建对象的时候才会用到,也就是说修改默认值时,并不会改变已添加到场景里的组件的当前值。

当你在编辑器中添加了一个组件以后,再回到脚本中修改一个默认值的话,属性检查器 里面是看不到变化的。因为属性的当前值已经序列化到了场景中,不再是第一次创建时用到的默认值了。如果要强制把所有属性设回默认值,可以在 属性检查器 的组件菜单中选择 `Reset`。

`default` 允许设置为以下几种值类型:

- 1、任意 `number`, `string` 或 `boolean` 类型的值 `null` 或 `undefined`

2、继承自 `cc.ValueType` 的子类，如 `cc.Vec2`, `cc.Color` 或 `cc.Rect` 的实例化对象：

```
properties: {  
    pos: {  
        default: new cc.Vec2(),  
    }  
}
```

3、空数组 `[]` 或空对象 `{}`

4、一个允许返回任意类型值的 `function`，这个 `function` 会在每次实例化该类时重新调用，并且以返回值作为新的默认值：

```
properties: {  
    pos: {  
        default: function () {  
            return [1, 2, 3];  
        },  
    }  
}
```

`visible` 参数

默认情况下，是否显示在 属性检查器 取决于属性名是否以下划线 `_` 开头。如果以下划线开头，则默认不显示在 属性检查器，否则默认显示。

如果要强制显示在 属性检查器，可以设置 `visible` 参数为 `true`：

```
properties: {
```



```

    _id: {          // 下划线开头原本会隐藏

        default: 0,

        visible: true

    }

}

```

如果要强制隐藏，可以设置 `visible` 参数为 `false`:

```

properties: {

    id: {          // 非下划线开头原本会显示

        default: 0,

        visible: false

    }

}

```

`serializable` 参数

指定了 `default` 默认值的属性默认情况下都会被序列化，如果不想序列化，可以设置 `serializable: false`。

```

temp_url: {

    default: "",

    serializable: false

}

```

`type` 参数

当 `default` 不能提供足够详细的类型信息时，为了能在 属性检查器 显示正确的输入控件，就要用 `type` 显式声明具体的类型：

当默认值为 `null` 时，将 `type` 设置为指定类型的构造函数，这样 属性检查器 才知道应该显示一个 `Node` 控件。

```
enemy: {  
    default: null,  
    type: cc.Node  
}
```

当默认值为数值（`number`）类型时，将 `type` 设置为 `cc.Integer`，用来表示这是一个整数，这样属性在 属性检查器 里就不能输入小数点。

```
score: {  
    default: 0,  
    type: cc.Integer  
}
```

当默认值是一个枚举（`cc.Enum`）时，由于枚举值本身其实也是一个数字（`number`），所以要将 `type` 设置为枚举类型，才能在 属性检查器 中显示为枚举下拉框。

```
wrap: {  
    default: Texture.WrapMode.Clamp,  
    type: Texture.WrapMode  
}
```

url 参数

如果属性是用来访问 Raw Asset 资源的 url , 为了能在 属性检查器 中选取资源 , 或者能正确序列化 , 你需要指定 url 参数 :

```
texture: {  
    default: "",  
    url: cc.Texture2D  
},
```

可参考 [获取和加载资源: Raw Asset](#)

override 参数

所有属性都将被子类继承 , 如果子类要覆盖父类同名属性 , 需要显式设置

override 参数 , 否则会有重名警告 :

```
_id: {  
    default: "",  
    tooltip: "my id",  
    override: true  
},  
name: {  
    get: function () {  
        return this._name;  
    },  
    displayName: "Name",  
    override: true
```

}

更多参数内容请查阅 下面的[属性参数](#)表。

属性检查器相关属性

参数名	说明	类型	默认值	备注
type	限定属性的数据类型	(Any)	undefined	详见 type 参数
visible	在 属性检视器 面板中显示或隐藏	boolean	(注 1)	详见 visible 参数
displayName	在 属性检视器 面板中显示为另一个名字	string	undefined	
tooltip	在 属性检视器 面板中添加属性的 Tooltip	string	undefined	
multiline	在 属性检视器 面板中使用多行文本框	boolean	false	
readonly	在 属性检视器 面板中只读	boolean	false	
range	限定数值在编辑器中输入的最大最小值	[min, max]	undefined	

序列化相关属性

这些属性不能用于 get 方法

参数名	说明	类型	默认值	备注
serializable	序列化该属性	boolean	true	详见 serializable 参数
editorOnly	在导出项目前剔除该属性	boolean	false	
其它属性				
参数名	说明	类型	默认值	备注
default	定义属性的默认值	(Any)	undefined	详见 default 参数
url	该属性为指定资源的 url	function (继承自 cc.RawAsset 的构造函数)	undefined	详见 获取和加载资源: Raw Asset
notify	当属性修改时触发指定方法	function (oldValue) {}	undefined	需要定义 default 属性
override	当重载父类属性时需要定义该参数为 true	boolean	false	详见 override 参数
animatable	该属性是否能被	boolean	true	

参数名	说明	类型	默认值	备注
-----	----	----	-----	----

动画修改

属性延迟定义

如果两个类相互引用，脚本加载阶段就会出现循环引用，循环引用将导致脚本加载出错：

Game.js	Item.js
<pre> var Item = require("Item"); var Game = cc.Class({ properties: { item: { default: null, type: Item } } }); module.exports = Game; </pre>	<pre> var Game = require("Game"); var Item = cc.Class({ properties: { game: { default: null, type: Game } } }); module.exports = Item; </pre>

上面两个脚本加载时，由于它们在 `require` 的过程中形成了闭环，因此加载会出现循环引用的错误，循环引用时 `type` 就会变为 `undefined`。因此我们提倡使用以下的属性定义方式：

Game.js	Item.js
<pre>var Game = cc.Class({ properties: () => ({ item: { default: null, type: require("Item") } }) }); module.exports = Game;</pre>	<pre>var Item = cc.Class({ properties: () => ({ game: { default: null, type: require("Game") } }) }); module.exports = Item;</pre>

这种方式就是将 `properties` 指定为一个 ES6 的箭头函数（lambda 表达式），箭头函数的内容在脚本加载过程中并不会同步执行，而是会被 `CCClass` 以异步的形式在所有脚本加载成功后才调用。因此加载过程中并不会出现循环引用，属性都可以正常初始化。

箭头函数的用法符合 JavaScript 的 ES6 标准，并且 `Creator` 会自动将 ES6 转义为 ES5，用户不用担心浏览器的兼容问题。

你可以这样来理解箭头函数：

// 箭头函数支持省略掉 `return` 语句，我们推荐的是这种省略后的写法：

`properties: () => {` // <- 箭头右边的括号 "(" 不可省略

```

    game: {
      default: null,
      type: require("Game")
    }
  })

```

// 如果要完整写出 `return`，那么上面的写法等价于：

```

properties: () => {
  return {
    game: {
      default: null,
      type: require("Game")
    }
  };
  // <- 这里 return 的内容，就是原先箭头右边括号里的部分
}

```

// 我们也可以不用箭头函数，而是用普通的匿名函数：

```

properties: function () {
  return {
    game: {
      default: null,
      type: require("Game")
    }
  }
}

```



```
};  
}
```

4.7 GetSet 方法

在属性中设置了 get 或 set 以后 ,访问属性的时候 ,就能触发预定义的 get 或 set 方法。

在属性中设置 get 方法 :

```
properties: {  
    width: {  
        get: function () {  
            return this.__width;  
        }  
    }  
}
```

get 方法可以返回任意类型的值。 这个属性同样能显示在 属性检查器 中 , 并且可以在包括构造函数内的所有代码里直接访问。

```
var Sprite = cc.Class({  
    ctor: function () {  
        this.__width = 128;  
        cc.log(this.width);    // 128  
    },  
    properties: {  
        width: {
```

```

        get: function () {
            return this.__width;
        }
    }
}
});

```

请注意：

设定了 `get` 以后，这个属性就不能被序列化，也不能指定默认值，但仍然可附带除了 `default`, `serializable` 外的大部分参数。

```

width: {
    get: function () {
        return this.__width;
    },
    type: cc.Integer,
    tooltip: "The width of sprite"
}

```

`get` 属性本身是只读的，但返回的对象并不是只读的。用户使用代码依然可以修改对象内部的属性，例如：

```

var Sprite = cc.Class({
    ...
    position: {
        get: function () {

```

```

        return this._position;

    },

}

...

});

var obj = new Sprite();

obj.position = new cc.Vec2(10, 20);    // 失败！position 是只读的！

obj.position.x = 100; //允许！position 返回的_position 对象本身可以修改！

```

在属性中设置 set 方法：

```

width: {

    set: function (value) {

        this._width = value;

    }

}

} //set 方法接收一个传入参数，这个参数可以是任意类型。

```

set 一般和 get 一起使用：

```

width: {

    get: function () {

        return this._width;

    },

    set: function (value) {

        this._width = value;

    },

}

```

```
    type: cc.Integer,  
    tooltip: "The width of sprite"  
}
```

如果没有和 `get` 一起定义，则 `set` 自身不能附带任何参数。和 `get` 一样，设定了 `set` 以后，这个属性就不能被序列化，也不能指定默认值。

4.8 editor 参数

`editor` 只能定义在 `cc.Component` 的子类。

```
cc.Class({  
    extends: cc.Component,  
    editor: {  
        // requireComponent 参数用来指定当前组件的依赖组件。  
        // 当组件添加到节点上时，如果依赖的组件不存在，引擎将会自动将依赖  
组件添加到同一个节点，防止脚本出错。该选项在运行时同样有效。  
        // 值类型：Function（必须是继承自 cc.Component 的构造函数，如  
cc.Sprite）默认值：null  
        requireComponent: null,  
        // 当本组件添加到节点上后，禁止 disallowMultiple 所指定类型（极其子  
类）的组件再添加到同一个节点，  
        // 防止逻辑发生冲突。  
        // 值类型：Function（必须是继承自 cc.Component 的构造函数，如  
cc.Sprite）默认值：null  
        disallowMultiple: null,
```

// menu 用来将当前组件添加到组件菜单中，方便用户查找。

// 值类型：String（如 "Rendering/Camera"）默认值：""

menu: "",

// 允许当前组件在编辑器模式下运行。

// 默认情况下，所有组件都只会在运行时执行，也就是说它们的生命周期回调在编辑器模式下并不会触发。

// 值类型：Boolean 默认值：false

executeInEditMode: false,

// 当设置了 "executeInEditMode" 以后，playOnFocus 可以用来设定选中当前组件所在的节点时，编辑器的场景刷新频率。

// playOnFocus 如果设置为 true，场景渲染将保持 60 FPS，如果为 false，场景就只会在必要的时候进行重绘。

// 值类型：Boolean 默认值：false

playOnFocus: false,

*// 自定义当前组件在 ****属性检查器**** 中渲染时所用的网页 url。*

// 值类型：String 默认值：""

inspector: "",

```
// 自定义当前组件在编辑器中显示的图标 url。

// 值类型 : String 默认值 : ""

icon: "",

// 指定当前组件的帮助文档的 url , 设置过后 , 在 **属性检查器** 中就会
出现一个帮助图标 ,

// 用户点击将打开指定的网页。

// 值类型 : String

// 默认值 : ""

help: "",

}

});
```

第五章 访问节点和组件

你可以在 属性检查器 里修改节点和组件，也能在脚本中动态修改。动态修改的好处是能够在一段时间内连续地修改属性、过渡属性，实现渐变效果。脚本还能够响应玩家输入，能够修改、创建和销毁节点或组件，实现各种各样的游戏逻辑。要实现这些效果，你需要先在脚本中获得你要修改的节点或组件。

在本篇教程，我们将介绍如何

- 获得组件所在的节点
- 获得其它组件
- 使用 属性检查器 设置节点和组件
- 查找子节点
- 全局节点查找
- 访问已有变量里的值

5.1 获得组件所在的节点

获得组件所在的节点很简单，只要在组件方法里访问 `this.node` 变量：

```
start: function () {  
    var node = this.node;  
    node.x = 100;  
}
```

5.2 获得其它组件

你会经常需要获得同一个节点上的其它组件，这就要用到 `getComponent` 这个 API，它会帮你查找你要的组件。

```

start: function () {

    var label = this.getComponent(cc.Label);

    var text = this.name + ' started';

    // 更改标签组件中的文本

    label.string = text;

}

```

你也可以为 `getComponent` 传入一个类名。对用户定义的组件而言，类名就是脚本的文件名，并且区分大小写。例如 "SinRotate.js" 里声明的组件，类名就是 "SinRotate"。

```

var label = this.getComponent("SinRotate");

```

在节点上也有一个 `getComponent` 方法，它们的作用是一样的：

```

this.node.getComponent(cc.Label) === this.getComponent(cc.Label);

```

如果在节点上找不到你要的组件，`getComponent` 将返回 `null`，如果你尝试访问 `null` 的值，将会在运行时抛出 "TypeError" 这个错误。因此如果你不确定组件是否存在，请记得判断一下：

```

start: function () {

    var label = this.getComponent(cc.Label);

    if (label) {

        label.string = "Hello";

    }

    else {

        cc.error("Something wrong?");

    }

}

```



```
    }  
}
```

5.3 获得其它节点及其组件

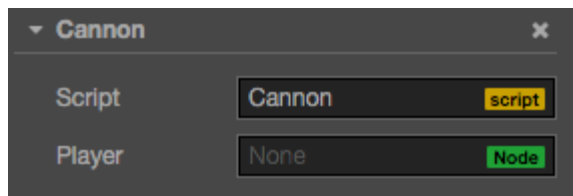
仅仅能访问节点自己的组件通常是不够的，脚本通常还需要进行多个节点之间的交互。例如，一门自动瞄准玩家的大炮，就需要不断获取玩家的最新位置。Cocos Creator 提供了一些不同的方法来获得其它节点或组件。

利用属性检查器设置节点

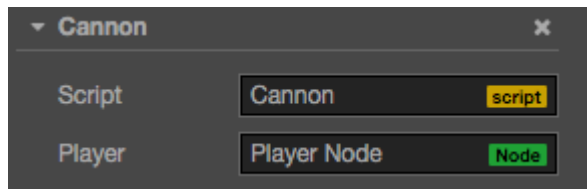
最直接的方式就是在 属性检查器 中设置你需要的对象。以节点为例，这只需要在脚本中声明一个 type 为 cc.Node 的属性：

```
// Cannon.js  
  
cc.Class({  
    extends: cc.Component,  
    properties: {  
        player: {// 声明 player 属性  
            default: null,  
            type: cc.Node  
        }  
    }  
});
```

这段代码在 properties 里面声明了一个 player 属性，默认值为 null，并且指定它的对象类型为 cc.Node。这就相当于在其它语言里声明了 public cc.Node player = null;。脚本编译之后，这个组件在 属性检查器 中看起来是这样的：



接着你就可以将层级管理器上的任意一个节点拖到这个 Player 控件：



这样一来它的 player 属性就会被设置成功，你可以直接在脚本里访问 player：

```
// Cannon.js

var Player = require("Player");

cc.Class({
    extends: cc.Component,

    properties: {
        player: {// 声明 player 属性
            default: null,
            type: cc.Node
        }
    },

    start: function () {
        var playerComp = this.player.getComponent(Player);

        this.checkPlayer(playerComp);
    },

    // ...
});
```

```
});
```

利用属性检查器设置组件

在上面的例子中，如果你将属性的 `type` 声明为 `Player` 组件，当你拖动节点 "Player Node" 到 属性检查器，`player` 属性就会被设置为这个节点里面的 `Player` 组件。这样你就不需要再自己调用 `getComponent` 啦。

```
// Cannon.js
```

```
var Player = require("Player");
```

```
cc.Class({
```

```
    extends: cc.Component,
```

```
    properties: {
```

```
        player: {// 声明 player 属性，这次直接是组件类型
```

```
            default: null,
```

```
            type: Player
```

```
        }
```

```
    },
```

```
    start: function () {
```

```
        var playerComp = this.player;
```

```
        this.checkPlayer(playerComp);
```

```
    },
```

```
    // ...
```

```
});
```

你还可以将属性的默认值由 `null` 改为数组`[]`，这样你就能在 属性检查器 中同时设置多个对象。不过如果需要在运行时动态获取其它对象，还需要用到下面介绍的查找方法。

查找子节点

有时候，游戏场景中会有很多个相同类型的对象，像是炮塔、敌人和特效，它们通常都有一个全局的脚本来统一管理。如果用 属性检查器 来一个一个将它们关联到这个脚本上，那工作就会很繁琐。为了更好地统一管理这些对象，我们可以把它们放到一个统一的父物体下，然后通过父物体来获得所有的子物体：

```
// CannonManager.js

cc.Class({

  extends: cc.Component,

  start: function () {

    this.cannons = [];

    this.cannons = this.node.getChildren();

  }

});
```

这里的 `getChildren` 是 `cc.Node` 原有的一个 API，可以获得一个包含所有子节点的数组。

你还可以使用 `getChildByName`：

```
this.node.getChildByName("Cannon 01");
```

如果子节点的层次较深，你还可以使用 `cc.find`，`cc.find` 将根据传入的路径进行逐级查找：

```
cc.find("Cannon 01/Barrel/SFX", this.node);
```

全局名字查找

当 `cc.find` 只传入第一个参数时，将从场景根节点开始逐级查找：

```
this.backNode = cc.find("Canvas/Menu/Back");
```

5.4 访问已有变量里的值

如果你已经在一个地方保存了节点或组件的引用，你也可以直接访问它们，一般有两种方式：

通过全局变量访问

你应当很谨慎地使用全局变量，当你要用全局变量时，应该很清楚自己在做什么，我们并不推荐滥用全局变量。如果你用了全局变量，被老板发现了，很有可能丢掉饭碗。

由于所有脚本都默认启用了 "use strict"，因此全局变量需要显式定义到 `window` 上。让我们试着定义一个全局对象 `window.Global`，这个对象里面包含了 `backNode` 和 `backLabel` 两个属性。

// Global.js, 此文件可以有任何名称

```
window.Global = {  
    backNode: null,  
    backLabel: null,  
};
```

你可以在合适的地方直接访问并初始化 `window.Global`:

```
// Back.js
```

```
cc.Class({
    extends: cc.Component,

    onLoad: function () {

        window.Global.backNode = this.node;

        window.Global.backLabel = this.getComponent(cc.Label);

    }

});
```

初始化后，你就能在任何地方访问到 `window.Global` 里的值：

// AnyScript.js

```
cc.Class({
    extends: cc.Component,

    // start 会在 onLoad 之后执行，所以这时 Global 已经初始化过了

    start: function () {

        var text = 'Back';

        window.Global.backLabel.string = text;

    }

});
```

通过模块访问

如果你不想用全局变量，你可以使用 `require` 来实现脚本的跨文件操作，让我们

看个示例：

// Global.js, now the filename matters

```
module.exports = {
```

```
        backNode: null,

        backLabel: null,

};
```

每个脚本都能用 `require + 文件名(不含路径)` 来获取到对方 `export` 的对象。

```
// Back.js

// 这感觉更安全，因为你知道对象是来自 var Global =
require("Global");

cc.Class({

    extends: cc.Component,

    onLoad: function () {

        Global.backNode = this.node;

        Global.backLabel = this.getComponent(cc.Label);

    }

});
```

```
// AnyScript.js

require("Global");

cc.Class({

    extends: cc.Component,

    // start 会在 onLoad 之后执行，所以这时 Global 已经初始化过了

    start: function () {

        var text = "Back";
```

```
Global.backLabel.string = text;

}

});
```

第六章 常用节点和组件接口

这篇文章将会介绍通过节点和组件实例可以通过哪些常用接口实现我们需要的种种效果和操作。这一篇也可以认为是 [cc.Node](#) 和 [cc.Component](#) 类的 API 阅读指南，可以配合 API 一起学习理解。

6.1 节点状态和层级操作

假设我们在一个组件脚本中，通过 `this.node` 访问当前脚本所在节点。

关闭/激活节点

```
this.node.active = false;
```

该操作会关闭节点，意味着：

- 在场景中隐藏该节点和所有子节点
- 该节点和所有子节点上的所有组件都将被禁用，也就是不会执行这些组件中的 `update` 中的代码
- 这些组件上的如果有 `onDisable` 方法，这些方法将被执行

```
this.node.active = true;
```

该操作会激活一个节点：

- 在场景中重新显示该节点和所有子节点，除非子节点单独设置过关闭

- 该节点和所有子节点上的所有组件都会被启用，他们中的 `update` 方法之后每帧会执行
- 这些组件上如果有 `onEnable` 方法，这些方法将被执行。

更改节点的父节点

假设父节点为 `parentNode`，子节点为 `this.node`

您可以：

```
parentNode.addChild(this.node);
```

```
this.node.parent = parentNode;
```

这两种方法是等价的。

注意，通过 [创建和销毁节点](#) 介绍的方法创建出新节点后，要为节点设置一个父节点才能正确完成节点的初始化。

索引节点的子节点

`this.node.children` *//将返回节点的所有子节点数组。*

`this.node.childrenCount` *//将返回节点的子节点数量。*

注意 以上两个 API 都只会返回节点的直接子节点，不会返回子节点的子节点。

6.2 更改节点的变换（位置、旋转、缩放、尺寸）

更改节点位置

分别对 `x` 轴和 `y` 轴坐标赋值：

```
this.node.x = 100; this.node.y = 50;
```

设置 `position` 变量：

```
this.node.position = cc.p(0, 0);
```

使用 `setPosition` 方法：

```
node.setPosition(cc.p(0, 0)); node.setPosition(0, 0);
```

以上两种用法等价。

更改节点旋转

```
1、 this.node.rotation = 90;
```

```
2、 this.node.setRotation(90);
```

更改节点缩放

```
1、 this.node.scaleX = 2; this.node.scaleY = 2;
```

```
2、 this.node.setScale(2); this.node.setScale(2, 2);
```

以上两种方法等价。setScale 传入单个参数时，会同时修改 scaleX 和 scaleY。

更改节点尺寸

```
1、 this.node.setContentSize(100, 100); 或 this.node.setContentSize(cc.p(100, 100));
```

```
2、 this.node.width = 100; this.node.height = 100;
```

以上两种方式等价。

更改节点锚点位置

```
1、 this.node.anchorX = 1; this.node.anchorY = 0;
```

```
2、 this.node.setAnchorPoint(1, 0);
```

注意以上这些修改变换的方法会影响到节点上挂载的渲染组件，比如 Sprite 图片的尺寸、旋转等等。

6.3 颜色和不透明度

在使用 Sprite, Label 这些基本的渲染组件时，要注意修改颜色和不透明度的操作只能在节点的实例上进行，因为这些渲染组件本身并没有设置颜色和不透明度的接口。

假如我们有一个 Sprite 的实例为 mySprite，如果需要设置它的颜色：

```
mySprite.node.color = cc.Color.RED;
```

设置不透明度：

```
mySprite.node.opacity = 128;
```

6.4 常用组件接口

cc.Component 是所有组件的基类，任何组件都包括如下的常见接口（假设我们在该组件的脚本中，以 this 指代本组件）：

- this.node：该组件所属的节点实例
- this.enabled：是否每帧执行该组件的 update 方法，同时也用来控制渲染组件是否显示
- update(dt)：作为组件的成员方法，在组件的 enabled 属性为 true 时，其中的代码会每帧执行
- onLoad()：组件所在节点进行初始化时（创建之后通过设置父节点添加到节点树）执行
- start()：会在该组件第一次 update 之前执行，通常用于需要在 onLoad 初始化完毕后执行的逻辑。

第七章 生命周期回调

Cocos Creator 为组件脚本提供了生命周期的回调函数。用户通过定义特定的函数回调在特定的时期编写相关 脚本。目前提供给用户的声明周期回调函数有：

- onLoad
- start
- update
- lateUpdate
- onDestroy
- onEnable
- onDisable

onLoad

组件脚本的初始化阶段，我们提供了 onLoad 回调函数。onLoad 回调会在这个组件所在的场景被载入 的时候触发，在 onLoad 阶段，保证了你可以获取到场景中的其他节点，以及节点关联的资源数据。通常 我们会在 onLoad 阶段去做一些初始化相关的操作。例如：

```
cc.Class({  
    extends: cc.Component,  
  
    properties: {  
        bulletSprite: cc.SpriteFrame,  
        gun: cc.Node,  
    },
```

```

onLoad: function () {

    this._bulletRect = this.bulletSprite.getRect();

    this.gun = cc.find('hand/weapon', this.node);

},

});

```

start

start 回调函数会在组件第一次激活前 ,也就是第一次执行 update 之前触发。start 通常用于 初始化一些中间状态的数据 , 这些数据可能在 update 时会发生改变 , 并且被频繁的 enable 和 disable。

注意: 当组件从 disable 回到 enable 状态后 , start 会再次被调用。

```

cc.Class({

    extends: cc.Component,

    start: function () {

        this._timer = 0.0;

    },

    update: function (dt) {

        this._timer += dt;

        if ( this._timer >= 10.0 ) {

            console.log('I am done!');

```

```
        this.enabled = false;

    }

},

});
```

update

游戏开发的一个关键点是在每一帧渲染前更新物体的行为，状态和方位。这些更新操作通常都放在 `update` 回调中。

```
cc.Class({

    extends: cc.Component,

    update: function (dt) {

        this.node.setPosition( 0.0, 40.0 * dt );

    }

});
```

lateUpdate

`update` 会在所有动画更新前执行，但如果我们要在动画更新之后才进行一些额外操作，或者希望在所有组件的 `update` 都执行完之后才进行其它操作，那就需要用到 `lateUpdate` 回调。

```
cc.Class({

    extends: cc.Component,

    lateUpdate: function (dt) {
```

```
        this.node.rotation = 20;

    }

});
```

onEnable

当组件的 `enabled` 属性从 `false` 变为 `true` 时，会激活 `onEnable` 回调。倘若节点第一次被创建且 `enabled` 为 `true`，则会在 `onLoad` 之后，`start` 之前被调用。

onDisable

当组件的 `enabled` 属性从 `true` 变为 `false` 时，会激活 `onDisable` 回调。

onDestroy

当组件调用了 `destroy()`，会在该帧结束被统一回收，此时会调用 `onDestroy` 回调。

第八章 创建和销毁节点

8.1 创建新节点

除了通过场景编辑器创建节点外，我们也可以在脚本中动态创建节点。通过 `new cc.Node()` 并将它加入到场景中，可以实现整个创建过程。例：

```
cc.Class({  
    extends: cc.Component,  
    properties: {  
        sprite: {  
            default: null,  
            type: cc.SpriteFrame,  
        },  
    },  
    start: function () {  
        var node = new cc.Node('sprite ' + this.count);  
        var sp = node.addComponent(cc.Sprite);  
        sp.spriteFrame = this.sprite;  
        node.parent = this.node;  
        node.setPosition(0,0);  
    },  
});
```

8.2 克隆已有节点

有时我们希望动态的克隆场景中的已有节点,我们可以通过 `cc.instantiate` 方法完成。使用方法如下:

```
cc.Class({
    extends: cc.Component,

    properties: {
        target: {
            default: null,
            type: cc.Node,
        },
    },

    start: function () {
        var scene = cc.director.getScene();
        var node = cc.instantiate(this.target);

        node.parent = scene;
        node.setPosition(0,0);
    },
});
```

8.3 创建预置节点

和克隆已有节点相似，你也设置你的预置（prefab）节点并通过 `cc.instantiate` 生成。使用方法如下：

```
cc.Class({
    extends: cc.Component,
    properties: {
        target: {
            default: null,
            type: cc.Prefab,
        },
    },
    start: function () {
        var scene = cc.director.getScene();
        var node = cc.instantiate(this.target);
        node.parent = scene;
        node.setPosition(0,0);
    },
});
```

8.4 销毁节点

通过 `node.destroy()` 函数，可以销毁节点。值得一提的是，销毁节点并不会立刻发生，而是在当前帧逻辑更新结束后，统一执行。当一个节点销毁后，该节点就处于无效状态，可以通过 `cc.isValid` 判断当前节点是否已经被销毁。

使用方法如下：

```
cc.Class({  
    extends: cc.Component,  
    properties: {  
        target: cc.Node,  
    },  
    start: function () {  
        setTimeout(function () {  
            this.target.destroy();  
        }.bind(this), 5000);  
    },  
    update: function (dt) {  
        if ( !cc.isValid(this.target) ) {  
            this.enabled = false;  
            return;  
        }  
        this.target.rotation += dt * 10.0;  
    },  
});
```

第九章 加载和切换场景

在 Cocos Creator 中，我们使用场景文件名（不包含扩展名）来索引指代场景。

并通过以下接口进行加载和切换操作：

```
cc.director.loadScene('MyScene');
```

9.1 通过常驻节点进行场景资源管理和参数传递

通常我们同时只会加载运行一个场景，当切换场景时，默认会将场景内所有节点和其他实例销毁。如果我们需要用一个组件控制所有场景的加载，或在场景之间传递参数数据，就需要将该组件所在节点标记为「常驻节点」，使它在场景切换时不被自动销毁，常驻内存。我们使用以下接口：

```
cc.game.addPersistRootNode(myNode);
```

上面的接口会将 myNode 变为常驻节点，这样挂在上面的组件都可以在场景之间持续作用，我们可以用这样的方法来储存玩家信息，或下一个场景初始化时需要的各种数据。

如果要取消一个节点的常驻属性：

```
cc.game.removePersistRootNode(myNode)
```

9.2 场景加载回调

加载场景时，可以附加一个参数用来指定场景加载后的回调函数：

```
cc.director.loadScene('MyScene', onSceneLaunched);
```

上一行里 onSceneLaunched 就是声明在本脚本中的一个回调函数，在场景加载后可以用来进一步的进行初始化或数据传递的操作。

由于回调函数只能写在本脚本中，所以场景加载回调通常用来配合常驻节点，在常驻节点上挂载的脚本中使用。

9.3 预加载场景

`cc.director.loadScene` 会在加载场景之后自动切换运行新场景，有些时候我们需要在后台静默加载新场景，并在加载完成后手动进行切换。

可以预先使用 `preloadScene` 接口对场景进行预加载：

```
cc.director.preloadScene('table', function () {  
  
    cc.log('Next scene preloaded');  
  
});
```

之后在合适的时间调用 `loadScene`，就可以立即切换场景

```
cc.director.loadScene('table');
```

注意 使用预加载场景资源配合 `runScene` 的方式进行预加载场景的方法已被废除

第十章 监听和发射事件

10.1 监听事件

事件处理是在节点（`cc.Node`）中完成的。对于组件，可以通过访问节点 `this.node` 来注册和监听事件。监听事件可以通过 `this.node.on()` 函数来注册，方法如下：

```
cc.Class({  
  
    extends: cc.Component,  
  
    properties: {
```

```

    },

    onLoad: function () {

        this.node.on('mousedown', function ( event ) {

            console.log('Hello!');

        });

    },

});

```

值得一提的是，事件监听函数 `on` 可以传第三个参数 `target`，用于绑定响应函数的调用者。以下两种调用方式，效果上是相同的：

```

this.node.on('mousedown', function ( event ) {// 使用函数绑定

    this.enabled = false;

}.bind(this));

```

```

this.node.on('mousedown', function (event) {// 使用第三个参数

    this.enabled = false;

}, this);

```

除了使用 `on` 监听，我们还可以使用 `once` 方法。`once` 监听在监听函数响应后就会关闭监听事件。

关闭监听

当我们不再关心某个事件时，我们可以使用 `off` 方法关闭对应的监听事件。需要注意的是，`off` 方法的参数必须和 `on` 方法的参数一一对应，才能完成关闭。

我们推荐的书写方法如下：

```
cc.Class({
  extends: cc.Component,

  _sayHello: function () {
    console.log('Hello World');
  },

  onEnable: function () {
    this.node.on('foobar', this._sayHello, this);
  },

  onDisable: function () {
    this.node.off('foobar', this._sayHello, this);
  },
});
```

10.2 发射事件

我们可以通过两种方式发射事件：emit 和 dispatchEvent。两者的区别在于，后者可以做事件传递。我们先通过一个简单的例子来了解 emit 事件：

```
cc.Class({
  extends: cc.Component,

  onLoad: function () {
    this.node.on('say-hello', function (event) {
      console.log(event.detail.msg);
    });
  },
});
```

```

start: function () {

    this.node.emit('say-hello', {

        msg: 'Hello, this is Cocos Creator',

    });

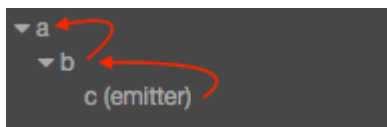
},

});

```

10.3 派送事件

上文提到了 `dispatchEvent` 方法，通过该方法发射的事件，会进入事件派送阶段。在 Cocos Creator 的事件派送系统中，我们采用冒泡派送的方式。冒泡派送会将事件从事件发起节点，不断地向上传递给他的父级节点，直到到达根节点或者在某个节点的响应函数中做了中断处理 `event.stopPropagation()`。



如上图所示，当我们从节点 `c` 发送事件 “foobar”，倘若节点 `a`，`b` 均做了 “foobar” 事件的监听，则事件会经由 `c` 依次传递给 `b`，`a` 节点。如：

// 节点 c 的组件脚本中

```
this.node.dispatchEvent( new cc.Event.EventCustom('foobar', true) );
```

如果我們希望在 `b` 节点截获事件后就不再将事件传递，我们可以通过调用 `event.stopPropagation()` 函数来完成。具体方法如下：

```

this.node.on('foobar', function (event) {// 节点 b 的组件脚本中

    event.stopPropagation();

});

```


请注意，在发送用户自定义事件的时候，请不要直接创建 `cc.Event` 对象，因为它是一个抽象类，请创建 `cc.Event.EventCustom` 对象来进行派发。

10.4 事件对象

在事件监听回调中，开发者会接收到一个 `cc.Event` 类型的事件对象 `event`，`stopPropagation` 就是 `cc.Event` 的标准 API，其它重要的 API 包含：

API 名	类型	意义
<code>type</code>	<code>String</code>	事件的类型（事件名）
<code>target</code>	<code>cc.Node</code>	接收到事件的原始对象
<code>currentTarget</code>	<code>cc.Node</code>	接收到事件的当前对象，事件在冒泡阶段 当前对象可能与原始对象不同
<code>getType</code>	<code>Function</code>	获取事件的类型
<code>stopPropagation</code>	<code>Function</code>	停止冒泡阶段，事件将不会继续向父节点 传递，当前节点的剩余监听器仍然会接收到事件
<code>stopPropagationImmediate</code>	<code>Function</code>	立即停止事件的传递，事件将不会传给父 节点以及当前节点的剩余监听器
<code>getCurrentTarget</code>	<code>Function</code>	获取当前接收到事件的目标节点
<code>detail</code>	<code>Function</code>	自定义事件的信息（属于 <code>cc.Event.EventCustom</code> ）
<code>setUserData</code>	<code>Function</code>	设置自定义事件的信息（属于 <code>cc.Event.EventCustom</code> ）

API 名	类型	意义
getUserData	Function	获取自定义事件的信息

完整的 API 列表可以参考 `cc.Event` 及其子类的 API 文档。

第十一章 系统内置事件

如上一篇文档所述，`cc.Node` 有一套完整的[事件监听和分发机制](#)。在这套机制之上，我们提供了一些基础的系统事件，这篇文档将介绍这些事件的使用方式。

系统事件遵守通用的注册方式，开发者既可以使用枚举类型也可以直接使用事件名来注册事件的监听器，事件名的定义遵循 DOM 事件标准。

// 使用枚举类型来注册

```
node.on(cc.Node.EventType.MOUSE_DOWN, function (event) {
    console.log('Mouse down');
}, this);
```

// 使用事件名来注册

```
node.on('mousedown', function (event) {
    console.log('Mouse down');
}, this);
```

11.1 鼠标事件类型和事件对象

鼠标事件在桌面平台才会触发，系统提供的事件类型如下：

枚举对象定义	对应的事件名	事件触发的时机
<code>cc.Node.EventType.MOUSE_DO</code>	<code>'mousedown'</code>	当鼠标在目标节点区域按下时

枚举对象定义	对应的事件名	事件触发的时机
WN	'	触发一次
cc.Node.EventType.MOUSE_ENTER	'mouseenter'	当鼠标移入目标节点区域时 , 不论是否按下
cc.Node.EventType.MOUSE_MOVE	'mousemove'	当鼠标在目标节点在目标节点区域中移动时 , 不论是否按下
cc.Node.EventType.MOUSE_LEAVE	'mouseleave'	当鼠标移出目标节点区域时 , 不论是否按下
cc.Node.EventType.MOUSE_UP	'mouseup'	当鼠标从按下状态松开时触发一次
cc.Node.EventType.MOUSE_WHEEL	'mousewheel'	当鼠标滚轮滚动时

鼠标事件(cc.Event.EventMouse)的重要 API 如下(cc.Event 标准事件 API 之外) :

函数名	返回值 类型	意义
getScrollY	Number	获取滚轮滚动的 Y 轴距离 , 只有滚动时才有效
getLocation	Object	获取鼠标位置对象 , 对象包含 x 和 y 属性
getLocationX	Number	获取鼠标的 X 轴位置

函数名	返回值 类型	意义
getLocationY	Number	获取鼠标的 Y 轴位置
getPreviousLocation	Object	获取鼠标事件上次触发时的位置对象 ,对象包含 x 和 y 属性
getDelta	Object	获取鼠标距离上一次事件移动的距离对象 , 对象包含 x 和 y 属性
getButton	Number	cc.Event.EventMouse.BUTTON_LEFT 或 cc.Event.EventMouse.BUTTON_RIGHT 或 cc.Event.EventMouse.BUTTON_MIDDLE

11.2 触摸事件类型和事件对象

触摸事件在移动平台和桌面平台都会触发 ,这样做的目的是为了更好得服务开发者在桌面平台调试 ,只需要监听触摸事件即可同时响应移动平台的触摸事件和桌面端的鼠标事件。系统提供的触摸事件类型如下 :

枚举对象定义	对应的事件名	事件触发的时机
cc.Node.EventType.TOUCH_START	'touchstart'	当手指触点落在目标节点区域内时
cc.Node.EventType.TOUCH_MOVE	'touchmove'	当手指在屏幕上目标节点区域内移动时
cc.Node.EventType.TOUCH_END	'touchend'	当手指在目标节点区域内离

枚举对象定义	对应的事件名	事件触发的时机
		开屏幕时
cc.Node.EventType.TOUCH_CAN		当手指在目标节点区域外离
	'touchcancel'	
CEL		开屏幕时

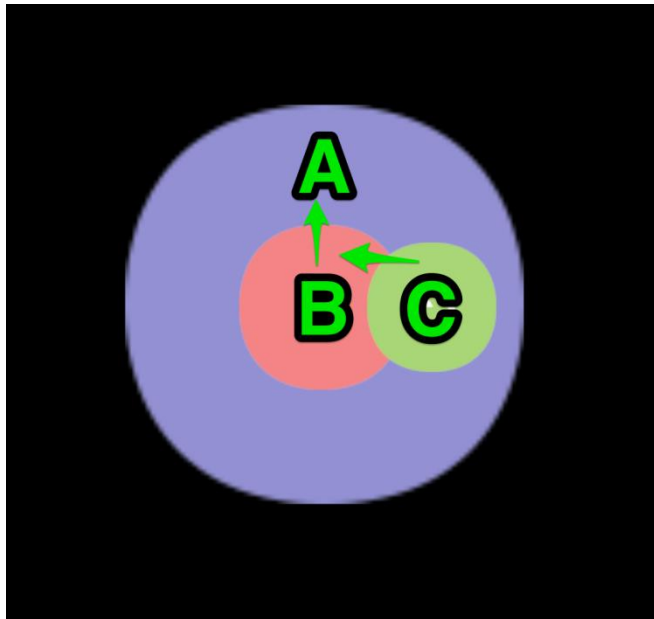
触摸事件(cc.Event.EventTouch)的重要 API 如下(cc.Event 标准事件 API 之外) :

API 名	类型	意义
touch	cc.Touch	与当前事件关联的触点对象
getID	Number	获取触点的 ID , 用于多点触摸的逻辑判断
getLocation	Object	获取触点位置对象 , 对象包含 x 和 y 属性
getLocationX	Number	获取触点的 X 轴位置
getLocationY	Number	获取触点的 Y 轴位置
getPreviousLocation	Object	获取触点上一次触发事件时的位置对象 , 对象包含 x 和 y 属性
getStartLocation	Object	获取触点初始时的位置对象 , 对象包含 x 和 y 属性
getDelta	Object	获取触点距离上一次事件移动的距离对象 , 对象包含 x 和 y 属性

需要注意的是, 触摸事件支持多点触摸, 每个触点都会发送一次事件给事件监听器。

11.3 鼠标和触摸事件冒泡

鼠标和触摸事件均支持节点树的事件冒泡，以下图为例：



在图中的场景里，A 节点拥有一个子节点 B，B 拥有一个子节点 C。假设开发者对 A、B、C 都监听了触摸事件。当鼠标或手指在 B 节点区域内按下时，事件将首先在 B 节点触发，B 节点监听器接收到事件。接着 B 节点会将事件向其父节点传递这个事件，A 节点的监听器将会接收到事件。这就是最基本的事件冒泡过程。

当鼠标或手指在 C 节点区域内按下时，事件将首先在 C 节点触发并通知 C 节点上注册的事件监听器。C 节点会通知 B 节点这个事件，B 节点内逻辑会负责检查触点是否发生在自身区域内，如果是则通知自己的监听器，否则什么都不做。紧接着 A 节点会收到事件，由于 C 节点完整处在 A 节点中，所以注册在 A 节点上的事件 监听器都将收到触摸按下事件。以上的过程解释了事件冒泡的过程和根据节点区域来判断是否分发事件的逻辑。

除了根据节点区域来判断是否分发事件外,鼠标和触摸事件的冒泡过程与普通事件的冒泡过程并没有区别。所以,调用 event 的 stopPropagation 函数可以主动停止冒泡过程。

11.4 cc.Node 的其它事件

枚举对象定义	对应的事件名	事件触发的时机
无	'position-changed'	当位置属性修改时
无	'rotation-changed'	当旋转属性修改时
无	'scale-changed'	当缩放属性修改时
无	'size-changed'	当宽高属性修改时
无	'anchor-changed'	当锚点属性修改时

第十二章 玩家输入事件

本篇教程，我们将介绍 Cocos Creator 的玩家输入事件。

目前支持了以下几种事件：

- 键盘事件
- 鼠标事件
- 触摸（单点与多点）事件

12.1 如何定义输入事件

所有的事件都是通过函数 `cc.eventManager.addListener(listener, target)` 来进行添加。

可选的 `event` 类型有：

1. `cc.EventListener.MOUSE` (鼠标)
2. `cc.EventListener.KEYBOARD` (键盘)
3. `cc.EventListener.TOUCH_ONE_BY_ONE` (单点触摸)
4. `cc.EventListener.TOUCH_ALL_AT_ONCE` (多点触摸)

1、鼠标事件

- 事件监听器类型：`cc.EventListener.MOUSE`
- 事件触发后的回调函数：
 - 鼠标按下：`onMouseDown(event);`
 - 鼠标释放：`onMouseUp(evnet);`
 - 鼠标移动：`onMouseMove(evnet);`
 - 鼠标滚轮：`onMouseScroll(evnet);`
- 回调参数：

- Event : [API 传送门](#)

// 添加鼠标事件监听器

```
var listener = {  
    event: cc.EventListener.MOUSE,  
    onMouseDown: function (event) { //鼠标按下  
        cc.log('Mouse Down: ' + event);  
    },  
    onMouseUp: function (event) { //鼠标释放  
        cc.log('Mouse Up: ' + event);  
    },  
    onMouseMove: function (event) { //鼠标移动  
        cc.log('Mouse Move: ' + event);  
    },  
    onMouseScroll: function (event) { //鼠标滚轮  
        cc.log('Mouse Scroll: ' + event);  
    }  
}  
cc.eventManager.addListener(listener, this.node); // 绑定鼠标事件
```

2、键盘事件

- 事件监听器类型 : cc.EventListener.KEYBOARD
- 事件触发后的回调函数 :
 - 键盘按下 : onKeyPressed(keyCode, event);

- 键盘释放 : onKeyReleased(keyCode, evnet);
- 回调参数 :
 - KeyCode: [API 传送门](#)
 - Event : [API 传送门](#)

// 添加键盘事件监听器

```
var listener = {
    event: cc.EventListener.KEYBOARD,
    onKeyPressed: function (keyCode, event) {
        cc.log('keyDown: ' + keyCode);
    },
    onKeyReleased: function (keyCode, event) {
        cc.log('keyUp: ' + keyCode);
    }
}

cc.eventManager.addListener(listener, this.node);// 绑定键盘事件
```

3、单点触摸事件

- 事件监听器类型 : cc.EventListener.TOUCH_ONE_BY_ONE
- 事件触发后的回调函数 :
 - 触摸开始 : onTouchBegan(touches, event);
 - 触摸移动时 : onTouchMoved(touches, event);
 - 触摸结束时 : onTouchEnded(touches, event);
 - 取消触摸 : onTouchCancelled(touches, event);

- 回调参数：
 - Touches: 触摸点的列表，单个 Touch [API 传送门](#)
 - Event : [API 传送门](#)

注意：onTouchBegan 回调事件里要 return true，这样后续的 onTouchEnded 和 onTouchMoved 才会触发事件。

// 添加单点触摸事件监听器

```
var listener = {  
  
    event: cc.EventListener.TOUCH_ONE_BY_ONE,  
  
    onTouchBegan: function (touches, event) {  
  
        cc.log('Touch Began: ' + event);  
  
        return true; //这里必须要写 return true  
  
    },  
  
    onTouchMoved: function (touches, event) {  
  
        cc.log('Touch Moved: ' + event);  
  
    },  
  
    onTouchEnded: function (touches, event) {  
  
        cc.log('Touch Ended: ' + event);  
  
    }  
  
    onTouchCancelled: function (touches, event) {  
  
        cc.log('Touch Cancelled: ' + event);  
  
    }  
}
```

```
}
```

```
cc.eventManager.addListener(listener, this.node);// 绑定单点触摸事件
```

4、多点触摸事件

- 事件监听器类型 : `cc.EventListener.TOUCH_ALL_AT_ONCE`
- 事件触发后的回调函数 :
 - 触摸开始 : `onTouchesBegan(touches, event);`
 - 触摸移动时 : `onTouchesMoved(touches, event);`
 - 触摸结束时 : `onTouchesEnded(touches, event);`
 - 取消触摸 : `onTouchesCancelled(touches, event);`
- 回调参数 :
 - Touches: 触摸点的列表, 单个 Touch [API 传送门](#)
 - Event : [API 传送门](#)

同理 : `onTouchesBegan` 回调事件里也要 `return true` , 这样后续的 `onTouchesEnded` 和 `onTouchesMoved` 才会触发事件。

```
// 添加多点触摸事件监听器
```

```
var listener = {  
  
  event: cc.EventListener.TOUCH_ALL_AT_ONCE,  
  
  onTouchesBegan: function (touches, event) {  
  
    // touches 触摸点的列表  
  
    cc.log('Touch Began: ' + event);  
  
    return true; //这里必须要写 return true
```

```
    },  
    onTouchesMoved: function (touches, event) {  
        cc.log('Touch Moved: ' + event);  
    },  
    onTouchesEnded: function (touches, event) {  
        cc.log('Touch Ended: ' + event);  
    },  
    onTouchesCancelled: function (touches, event) {  
        cc.log('Touch Cancelled: ' + event);  
    }  
}  
  
// 绑定多点触摸事件  
cc.eventManager.addListener(listener, this.node);
```

第十三章 在 Cocos Creator 中使用动作系统

13.1 动作系统简介

Cocos Creator 提供的动作系统源自 Cocos2d-x，API 和使用方法均一脉相承。

动作系统可以在一定时间内对节点完成位移，缩放，旋转等各种动作。

需要注意的是，动作系统并不能取代[动画系统](#)，动作系统提供的是面向程序员的 API 接口，而动画系统则是提供在编辑器中来设计的。同时，他们服务于不同的使用场景，动作系统比较适合来制作简单的形变和位移动画，而动画系统则强大许多，美术可以用编辑器制作支持各种属性，包含运动轨迹和缓动的复杂动画。

13.2 动作系统 API

动作系统的使用方式也很简单，在 cc.Node 中支持如下 API：

// 创建一个移动动作

```
var action = cc.moveTo(2, 100, 100);
```

```
node.runAction(action); // 执行动作
```

```
node.stopAction(action); // 停止一个动作
```

```
node.stopAllActions(); // 停止所有动作
```

开发者还可以给动作设置 tag，并通过 tag 来控制动作。

// 给 action 设置 tag

```
var ACTION_TAG = 1;
```

```
action.setTag(ACTION_TAG);
```

```
node.getActionByTag(ACTION_TAG); // 通过 tag 获取 action
```

```
node.stopActionByTag(ACTION_TAG); // 通过 tag 停止一个动作
```

13.3 动作类型

在 Cocos Creator 中支持非常丰富的各种动作,这些动作主要分为几大类:(由于动作类型过多,在这里不展开描述每个动作的用法,开发者可以参考[动作系统 API 列表](#)来查看所有动作)

1、基础动作

基础动作就是实现各种形变,位移动画的动作,比如 `cc.moveTo` 用来移动节点到某个位置;`cc.rotateBy` 用来旋转节点一定的角度;`cc.scaleTo` 用来缩放节点。

基础动作中分为时间间隔动作和即时动作,前者是在一定时间间隔内完成的渐变动作,前面提到的都是时间间隔动作,它们全部继承自 [cc.ActionInterval](#)。后者则是立即发生的,比如用来调用回调函数的 `cc.callFunc`;用来隐藏节点的 `cc.hide`,它们全部继承自 [cc.ActionInstant](#)。

2、容器动作

容器动作可以以不同的方式将动作组织起来,下面是几种容器动作的用途:

1、顺序动作 `cc.sequence` 顺序动作可以让一系列子动作按顺序一个个执行。示例:

```
// 让节点左右来回移动
```

```
var seq = cc.sequence(cc.moveBy(0.5, 200, 0), cc.moveBy(0.5, -200, 0));
```

```
node.runAction(seq);
```

2、同步动作 `cc.spawn` 同步动作可以同步执行对一系列子动作,子动作的执行结果会叠加起来修改节点的属性。示例:

```
// 让节点在向上移动的同时缩放
```

```
var spawn = cc.spawn(cc.moveBy(0.5, 0, 50), cc.scaleTo(0.5, 0.8, 1.4));  
node.runAction(spawn);
```

3、重复动作 `cc.repeat` 重复动作用来多次重复一个动作。示例：

// 让节点左右来回移动，并重复 5 次

```
var seq = cc.repeat(  
    cc.sequence(  
        cc.moveBy(2, 200, 0),  
        cc.moveBy(2, -200, 0)  
    ), 5);  
node.runAction(seq);
```

4、永远重复动作 `cc.repeatForever` 顾名思义，这个动作容器可以让目标动作一直重复，直到手动停止。

// 让节点左右来回移动并一直重复

```
var seq = cc.repeatForever(  
    cc.sequence(  
        cc.moveBy(2, 200, 0),  
        cc.moveBy(2, -200, 0)  
    ));
```

5、速度动作 `cc.speed` 速度动作可以改变目标动作的执行速率，让动作更快或者更慢完成。

// 让目标动作速度加快一倍，相当于原本 2 秒的动作在 1 秒内完成

```
var action = cc.speed(  
    cc.spawn(  
        cc.moveBy(2, 200, 0),  
        cc.moveBy(2, -200, 0)  
    ));
```



```

        cc.moveBy(2, 0, 50),

        cc.scaleTo(2, 0.8, 1.4)

        ), 0.5);

node.runAction(action);

```

从上面的示例中可以看出，不同容器类型是可以复合的，除此之外，我们给容器类型动作提供了更为方便的链式 API，动作对象支持以下三个 API：repeat、repeatForever、speed，这些 API 都会返回动作对象本身，支持继续链式调用。我们来看一个更复杂的动作示例：

// 一个复杂的跳跃动画

```

this.jumpAction = cc.sequence(

    cc.spawn(

        cc.scaleTo(0.1, 0.8, 1.2),

        cc.moveTo(0.1, 0, 10)

    ),

    cc.spawn(

        cc.scaleTo(0.2, 1, 1),

        cc.moveTo(0.2, 0, 0)

    ),

    cc.delayTime(0.5),

    cc.spawn(

        cc.scaleTo(0.1, 1.2, 0.8),

        cc.moveTo(0.1, 0, -10)

```

```

    ),
    cc.spawn(
        cc.scaleTo(0.2, 1, 1),
        cc.moveTo(0.2, 0, 0)
    )// 以 1/2 的速度慢放动画 , 并重复 5 次
).speed(2).repeat(5);

```

3、动作回调

动作回调可以用以下的方式声明：

```
var finished = cc.callFunc(this.myMethod, this, opt);
```

cc.callFunc 第一个参数是处理回调的方法，即可以使用 CClass 的成员方法，也可以声明一个匿名函数：

```

var finished = cc.callFunc(function () {
    //doSomething
}, this, opt);

```

第二个参数指定了处理回调方法的 context (也就是绑定 this)，第三个参数是向处理回调方法的传参。您可以这样使用传参：

```

var finished = cc.callFunc(function(target, score) {
    this.score += score;
}, this, 100);//动作完成后会给玩家加 100 分

```

在声明了回调动作 finished 后，您可以配合 cc.sequence 来执行一整串动作并触发回调：

```
var myAction = cc.sequence(cc.moveBy(1, cc.p(0, 100)), cc.fadeOut(1),
finished);
```

在同一个 sequence 里也可以多次插入回调：

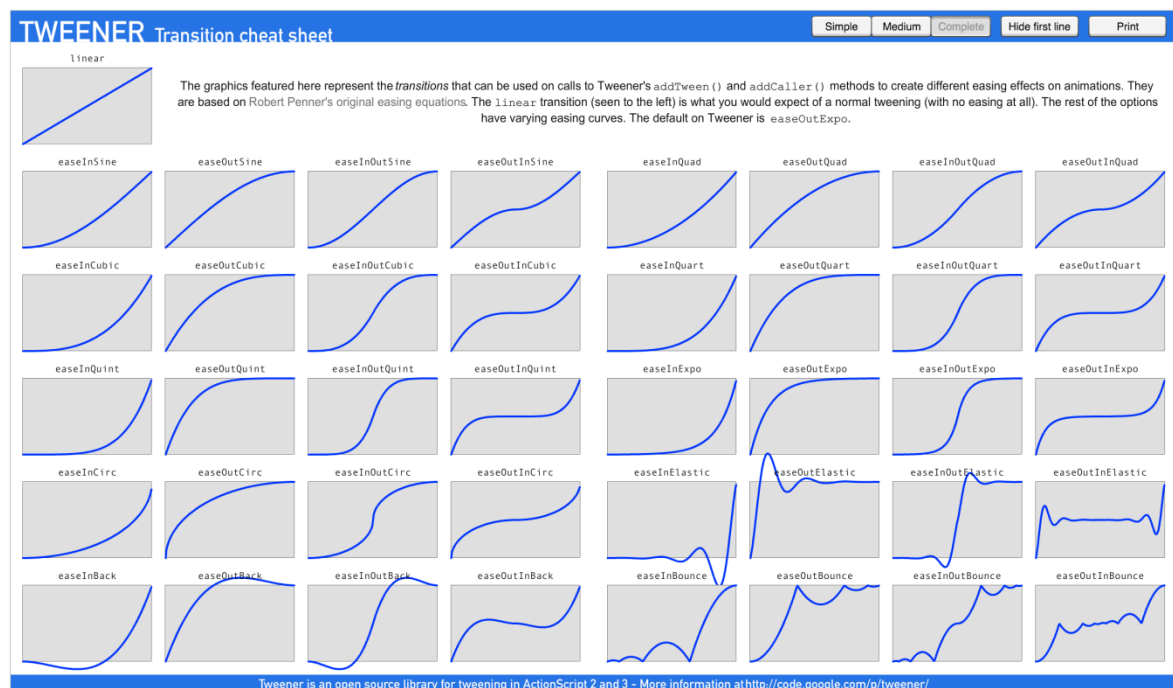
```
var myAction = cc.sequence(cc.moveTo(1, cc.p(0, 0)), finished1, cc.fadeOut(1),
finished2); //finished1, finished2 都是使用 cc.callFunc 定义的回调动作
```

4、缓动动作

缓动动作不可以单独存在，它永远是为了修饰基础动作而存在的，它可以用来修改基础动作的时间曲线，让动作有快入、缓入、快出或其它更复杂的特效。需要注意的是，只有时间间隔动作才支持缓动：

```
var aciton = cc.scaleTo(0.5, 2, 2);
action.easing(cc.easeIn(3.0));
```

基础的缓动动作类是 [cc.ActionEase](#)。各种缓动动作的时间曲线可以参考下图：



13.4 动作列表

基础动作类型

- [Action](#) : 所有动作类型的基类。
- [FiniteTimeAction](#) : 有限时间动作, 这种动作拥有时长 duration 属性。
- [ActionInstant](#) : 即时动作, 这种动作立即就会执行, 继承自 FiniteTimeAction。
- [ActionInterval](#) : 时间间隔动作, 这种动作在已定时间内完成, 继承自 FiniteTimeAction。
- [ActionEase](#) : 所有缓动动作基类, 用于修饰 ActionInterval。
- [EaseRateAction](#) : 拥有速率属性的缓动动作基类。
- [EaseElastic](#) : 弹性缓动动作基类。
- [EaseBounce](#) : 反弹缓动动作基类。

容器动作

动作名称	简介	文档链接
cc.sequence	顺序执行动作	API 描述
cc.spawn	同步执行动作	API 描述
cc.repeat	重复执行动作	API 描述
cc.repeatForever	永远重复动作	API 描述
cc.speed	修改动作速率	API 描述

即时动作

动作名称	简介	文档链接
cc.show	立即显示	API 描述

动作名称	简介	文档链接
cc.hide	立即隐藏	API 描述
cc.toggleVisibility	显隐状态切换	API 描述
cc.removeSelf	从父节点移除自身	API 描述
cc.flipX	X 轴翻转	API 描述
cc.flipY	Y 轴翻转	API 描述
cc.place	放置在目标位置	API 描述
cc.callFunc	执行回调函数	API 描述
cc.targetedAction	用已有动作和一个新的目标节点创建动作	API 描述

时间间隔动作

动作名称	简介	文档链接
cc.moveTo	移动到目标位置	API 描述
cc.moveBy	移动指定的距离	API 描述
cc.rotateTo	旋转到目标角度	API 描述
cc.rotateBy	旋转指定的角度	API 描述
cc.scaleTo	将节点大小缩放到指定的倍数	API 描述
cc.scaleBy	按指定的倍数缩放节点大小	API 描述
cc.skewTo	偏斜到目标角度	API 描述
cc.skewBy	偏斜指定的角度	API 描述
cc.jumpBy	用跳跃的方式移动指定的距离	API 描述

动作名称	简介	文档链接
cc.jumpTo	用跳跃的方式移动到目标位置	API 描述
cc.follow	追踪目标节点的位置	API 描述
cc.bezierTo	按贝赛尔曲线轨迹移动到目标位置	API 描述
cc.bezierBy	按贝赛尔曲线轨迹移动指定的距离	API 描述
cc.blink	闪烁（基于透明度）	API 描述
cc.fadeTo	修改透明度到指定值	API 描述
cc.fadeIn	渐显	API 描述
cc.fadeOut	渐隐	API 描述
cc.tintTo	修改颜色到指定值	API 描述
cc.tintBy	按照指定的增量修改颜色	API 描述
cc.delayTime	延迟指定的时间量	API 描述
cc.reverseTime	反转目标动作的时间轴	API 描述
cc.cardinalSplineTo	按基数样条曲线轨迹移动到目标位置	API 描述
cc.cardinalSplineBy	按基数样条曲线轨迹移动指定的距离	API 描述
cc.catmullRomTo	按 Catmull Rom 样条曲线轨迹移动到目标位置	API 描述
cc.catmullRomBy	按 Catmull Rom 样条曲线轨迹移动指定的距离	API 描述

缓动动作

动作名称	文档链接
cc.easeIn	API 描述
cc.easeOut	API 描述
cc.easeInOut	API 描述
cc.easeExponentialIn	API 描述
cc.easeExponentialOut	API 描述
cc.easeExponentialInOut	API 描述
cc.easeSineIn	API 描述
cc.easeSineOut	API 描述
cc.easeSineInOut	API 描述
cc.easeElasticIn	API 描述
cc.easeElasticOut	API 描述
cc.easeElasticInOut	API 描述
cc.easeBounceIn	API 描述
cc.easeBounceOut	API 描述
cc.easeBounceInOut	API 描述
cc.easeBackIn	API 描述
cc.easeBackOut	API 描述
cc.easeBackInOut	API 描述

动作名称	文档链接
cc.easeBezierAction	API 描述
cc.easeQuadraticActionIn	API 描述
cc.easeQuadraticActionOut	API 描述
cc.easeQuadraticActionInOut	API 描述
cc.easeQuarticActionIn	API 描述
cc.easeQuarticActionOut	API 描述
cc.easeQuarticActionInOut	API 描述
cc.easeQuinticActionIn	API 描述
cc.easeQuinticActionOut	API 描述
cc.easeQuinticActionInOut	API 描述
cc.easeCircleActionIn	API 描述
cc.easeCircleActionOut	API 描述
cc.easeCircleActionInOut	API 描述
cc.easeCubicActionIn	API 描述
cc.easeCubicActionOut	API 描述
cc.easeCubicActionInOut	API 描述

第十四章 使用计时器

在 Cocos Creator 中，我们为组件提供了方便的计时器，这个计时器源自于 Cocos2d-x 中的 `cc.Scheduler`，我们将它保留在了 Cocos Creator 中并适配了基于组件的使用方式。

下面来看看它的具体使用方式：

开始一个计时器

```
component.schedule(function() {  
    this.doSomething(); //这里的 this 指向 component  
}, 5);
```

上面这个计时器将每隔 5s 执行一次。

更灵活的计时器

```
var interval = 5; // 以秒为单位的时间间隔  
var repeat = 3; // 重复次数  
var delay = 10; // 开始延时  
component.schedule(function() {  
    this.doSomething();  
}, interval, repeat, delay);
```

上面的计时器将在 10 秒后开始计时，每 5 秒执行一次回调，重复 3 次。

只执行一次的计时器（快捷方式）

```
component.scheduleOnce(function() {  
    // 这里的 this 指向 component  
    this.doSomething();
```

```
}, 2);
```

上面的计时器将在两秒后执行一次回调函数，之后就停止计时。

取消计时器

开发者可以使用回调函数本身来取消计时器：

```
this.count = 0;

this.callback = function () {

    if (this.count === 5) { // 在第六次执行回调时取消这个计时器

        this.unschedule(this.callback);

    }

    this.doSomething();

    this.count++;

}

component.schedule(this.callback, 1);
```

下面是 Component 中所有关于计时器的函数：

- `schedule`：开始一个计时器
- `scheduleOnce`：开始一个只执行一次的计时器
- `unschedule`：取消一个计时器
- `unscheduleAllCallbacks`：取消这个组件的所有计时器

这些 API 的详细描述都可以在 [Component API](#) 文档中找到。

除此之外，如果需要每一帧都执行一个函数，请直接在 Component 中添加 `update` 函数，这个函数将默认被每帧调用，这在[生命周期文档](#)中有详细描述。

第十五章 脚本执行顺序

完善的脚本执行顺序控制将在新版本中添加，目前请使用下面的原则控制脚本执行顺序：

15.1 使用统一的控制脚本来初始化其他脚本

一般我都会有一个 `Game.js` 的脚本作为总的控制脚本，假如我还有 `Player.js`, `Enemy.js`, `Menu.js` 三个组件，那么他们的初始化过程是这样的：

```
// Game.js

const Player = require('Player');

const Enemy = require('Enemy');

const Menu = require('Menu');

cc.Class({

  extends: cc.Component,

  properties: {

    player: Player,

    enemy: Enemy,

    menu: Menu

  },

  onLoad: function () {

    this.player.init();

    this.enemy.init();

    this.menu.init();

  }

});
```

```
});
```

其中在 Player.js, Enemy.js 和 Menu.js 中需要实现 init 方法 , 并将初始化逻辑放进去。这样我们就可以保证 Player, Enemy 和 Menu 的初始化顺序。

15.2 在 Update 中用自定义方法控制更新顺序

同理如果要保证以上三个脚本的每帧更新顺序 , 我们也可以将分散在每个脚本里的 update 替换成自己定义的方法 :

```
// Player.js
```

```
updatePlayer: function (dt) {  
  
    // do player update  
  
}
```

然后在 Game.js 脚本的 update 里调用这些方法 :

```
// Game.js
```

```
update: function (dt) {  
  
    this.player.updatePlayer(dt);  
  
    this.enemy.updateEnemy(dt);  
  
    this.menu.updateMenu(dt);  
  
}
```

15.3 控制同一个节点上的组件执行顺序

在同一个节点上的组件脚本执行顺序 , 是可以通过组件在 属性检查器 里的排列顺序来控制的。排列在上的组件会先于排列在下的组件执行。我们可以通过组件右上角的齿轮按钮里的 Move Up 和 Move Down 菜单来调整组件的排列顺序和执行顺序。

假如我们有两个组件 CompA 和 CompB，他们的内容分别是：

```
// CompA.js
```

```
cc.Class({  
    extends: cc.Component,  
    onLoad: function () {  
        cc.log('CompA onLoad!');  
    },  
    start: function () {  
        cc.log('CompA start!');  
    },  
    update: function (dt) {  
        cc.log('CompA update!');  
    },  
});
```

```
// CompB.js
```

```
cc.Class({  
    extends: cc.Component,  
    onLoad: function () {  
        cc.log('CompB onLoad!');  
    },  
    start: function () {
```

```
        cc.log('CompB start!');  
    },  
    update: function (dt) {  
        cc.log('CompB update!');  
    },  
});
```

CompA 在 CompB 上面时，输出：CompA onLoad! CompB onLoad!

CompA start! CompB start! CompA update! CompB update!

在 属性检查器 里通过 CompA 组件右上角齿轮菜单里的 Move Down 将

CompA 移到 CompB 下面后，输出：CompB onLoad! CompA onLoad!

CompB start! CompA start! CompB update! CompA update!

第十六章 标准网络接口

在 Cocos Creator 中，我们支持 Web 平台上最广泛使用的标准网络接口：

- XMLHttpRequest：用于短连接
- WebSocket：用于长连接

16.1 使用方法

XMLHttpRequest 简单示例：

```
var xhr = new XMLHttpRequest();

xhr.onreadystatechange = function () {

    if (xhr.readyState == 4 && (xhr.status >= 200 && xhr.status < 400)) {

        var response = xhr.responseText;

        console.log(response);

    }

};

xhr.open("GET", url, true);

xhr.send();
```

开发者可以直接使用 `new XMLHttpRequest()` 来创建一个连接对象，也可以通过 `cc.loader.getXMLHttpRequest()` 来创建，两者效果一致。

XMLHttpRequest 的标准文档请参考 [MDN 中文文档](#)。

WebSocket 简单示例：

```
ws = new WebSocket("ws://echo.websocket.org");

ws.onopen = function (event) {
```

```
        console.log("Send Text WS was opened.");
    };

    ws.onmessage = function (event) {

        console.log("response text msg: " + event.data);

    };

    ws.onerror = function (event) {

        console.log("Send Text fired an error");

    };

    ws.onclose = function (event) {

        console.log("WebSocket instance closed.");

    };

    setTimeout(function () {

        if (ws.readyState === WebSocket.OPEN) {

            ws.send("Hello WebSocket, I'm a text message.");

        }

        else {

            console.log("WebSocket instance wasn't ready...");

        }

    }, 3);
```

WebSocket 的标准文档请参考 [MDN 中文文档](#)。

16.2 SocketIO

除此之外 ,SocketIO 提供一种基于 WebSocket API 的封装 ,可以用于 Node.js 服务端。如果需要使用这个库 ,开发者可以自己引用 SocketIO。

在脚本中引用 SocketIO :

- 1、下载 SocketIO : [下载地址](#)
- 2、将下载后的文件放入拖入资源管理器中你希望保存的路径

修改 SocketIO 脚本文件以避免在原生环境中被执行

由于 Web 版本 SocketIO 不能够在 JSB 中被正确解析 ,而后面我们将要使用的 require 导致 SocketIO 脚本必然被打包到用户代码中 ,所以我们需要一点 hack 的手段让 Web 版本 SocketIO 的脚本在原生环境中不生效。方法就是在 SocketIO 脚本文件中做如下修改 :

```
if (!cc.sys.isNative) {  
  
    // SocketIO 原始代码  
  
}
```

在组件脚本中引用 SocketIO :

//判断是否是 native 环境 , 如果是则不能够引用 , 因为 native 提供原生的 SocketIO 实现

```
if (!cc.sys.isNative) {  
  
    // 使用相对路径 , 不需要包含 .js 后缀  
  
    require('relative_path_to/socket.io');  
  
}  
  
else {  
  
    // 原生环境中 io 变量未定义 , 导出的变量实际上是 SocketIO
```

```
    window.io = SocketIO;  
  }
```

在组件中使用 SocketIO，可以参考 [SocketIO 官方网站](#) 查询 API 和文档等

第十七章 使用对象池

在运行时进行节点的创建(cc.instantiate)和销毁(node.destroy)操作是非常耗费性能的，因此我们在比较复杂的场景中，通常只有在场景初始化逻辑（onLoad）中才会进行节点的创建，在切换场景时才会进行节点的销毁。如果制作有大量敌人或子弹需要反复生成和被消灭的动作类游戏，我们要如何在游戏进行过程中随时创建和销毁节点呢？这里就需要对象池的帮助了。

17.1 对象池的概念

对象池就是一组可回收的节点对象，并通过 `cc.pool` 单例统一管理。当我们需要创建节点时，向对象池申请一个节点，如果对象池里有空闲的可用节点，就会把节点返回给用户，用户通过 `node.addChild` 将这个新节点加入到场景节点树中。

当我们需要销毁节点时，就把之前使用过的节点从场景节点树中移除（`node.removeFromParent`），然后返回给对象池。这样就实现了少数节点的循环利用。假如玩家在一关中要杀死 100 个敌人，但同时出现的敌人不超过 5 个，那我们就只需要生成 5 个节点大小的对象池，然后循环使用就可以了。

关于 `cc.pool` 的详细 API 说明，请参考 [cc.pool API 文档](#)。

17.2 流程介绍

下面是使用对象池的一般工作流程

- 1、准备好 Prefab

把你想要创建的节点事先设置好并做成 Prefab 资源，方法请查看 [预制资源工作流程](#)。

2、初始化对象池

在场景加载的初始化脚本中，我们可以将需要数量的节点创建出来，并放进对象池：

```
//...

properties: {

    enemyPrefab: cc.Prefab

},

onLoad: function () {

    let initCount = 5;

    for (let i = 0; i < initCount; ++i) {

        let enemy = cc.instantiate(this.enemyPrefab); // 创建节点

        cc.pool.putInPool(enemy); // 通过 putInPool 接口放入对象池

    }

}
```

对象池里需要的初始节点数量可以根据游戏的需要来控制，即使我们对初始节点数量的预估不准确也不要紧，后面我们会进行处理。

3、从对象池请求对象

接下来在我们的运行时代码中就可以用下面的方式来获得对象池中储存的对象了：

```
// ...

createEnemy: function (parentNode) {

    let enemy = null;
```

```

// 通过 hasObject 接口判断对象池中是否有空闲的对象，参数为该对象的类型
if (cc.pool.hasObject(cc.Node)) {

    enemy = cc.pool.getFromPool(cc.Node);

} else { // 如果没有空闲对象，也就是对象池中备用对象不够时，我们就用
cc.instantiate 重新创建

    enemy = cc.instantiate(this.enemyPrefab);

}

enemy.parent = parentNode; // 将生成的敌人加入节点树

//接下来就可以调用 enemy 身上的脚本进行初始化

enemy.getComponent('Enemy').init();

}

```

安全使用对象池的要点就是在 `getFromPool` 获取对象之前，永远都要先用 `hasObject` 来判断是否有可用的对象，如果没有就使用正常创建节点的方法，虽然会消耗一些运行时性能，但总比游戏崩溃要好！

4、将对象返回对象池

当我们杀死敌人时，需要将敌人节点退还给对象池，以备之后继续循环利用，我们用这样的方法：

```

// ...

onEnemyKilled: function (enemy) {

    enemy.removeFromParent(); // 首先将 enemy 节点从节点树中移除

    cc.pool.putInPool(enemy); // 和初始化时的方法一样，将节点放进对象池

}

```

这样我们就完成了一个完整的循环，主角需要刷多少怪都不成问题了！将节点放入和从对象池取出的操作不会带来额外的内存管理开销，因此只要是可能，应该尽量去利用。

17.3 使用组件对象

除了节点，我们还可以将组件对象放入和从对象池取出，当我们有多个不同的敌人组件 `Enemy1.js`、`Enemy2.js` 时，通过组件来区分不同类型的对象就会很方便：

```
properties: {  
    // 两个 prefab 节点上分别挂了 `Enemy1.js` 和 `Enemy2.js` 两个组件  
    enemy1Prefab: cc.Prefab,  
    enemy2Prefab: cc.Prefab  
},  
  
createEnemy1: function (parentNode) {  
    let Enemy1 = require('Enemy1'); //这个类型声明可以放在脚本最前面  
    let enemy = null;  
    if (cc.pool.hasObject(Enemy1) { // 通过 hasObject 接口判断对象池中是否有空闲的对象，参数为该对象的类型  
        enemy = cc.pool.getFromPool(Enemy1);  
    } else { // 如果没有空闲对象，也就是对象池中备用对象不够时，我们就用  
        cc.instantiate 重新创建  
        // 由于 `instantiate` 返回的是 cc.Node，因此我们需要再用  
        `getComponent` 取到组件对象
```

```

        enemy =
cc.instantiate(this.enemy1Prefab).getComponent('Enemy1');

    }

    enemy.node.parent = parentNode; // 注意这里需要调用组件的 node 属性

    enemy.init(); //这里可以直接调用初始化

}

```

这里要密切注意 enemy 的类型，请和前面使用节点作为对象时进行反复对比，理解对象类型的概念。

17.4 清除对象池

由于 cc.pool 是个单例，所以我们切换场景时需要手动重置对象池，以便进行全新的初始化逻辑：

```
cc.pool.drainAllPools(); // 调用这个方法就可以重置对象池
```

限制

cc.pool 总是根据对象类型来管理对象池的，如果我们有多个 Prefab 都使用了同一个组件，只是其中配置的属性数据不同时，就无法用 cc.pool 来分别获得不同的 Prefab。这种情况下就需要自己实现一个对象池。

对象池的基本功能其实非常简单，就是一个或多个数组来保存已经创建的节点实例。你可以参考 [暗黑斩 Demo 中的 PoolMng 脚本](#) 来实现。

第十八章 获取和加载资源

Cocos Creator 有一套统一的资源管理机制，在本篇教程，我们将介绍

- 资源的分类
- 如何在 属性检查器 里设置资源
- 动态加载 Asset
- 动态加载 Raw Asset

18.1 资源的分类

目前的资源分成两种，一种叫做 Asset，一种叫做 Raw Asset。

1、Asset

Creator 提供了名为 "Asset" 的资源类型，cc.SpriteFrame, cc.AnimationClip, cc.Prefab 等资源都属于 Asset。Asset 的加载是统一并且自动化的，相互依赖的 Asset 能够被自动预加载。

例如，当引擎在加载场景时，会先自动加载场景关联到的资源，这些资源如果再关联其它资源，其它也会被先被加载，等加载全部完成后，场景加载才会结束。

因此只要你拿到了一个 Asset 对象，这个对象一定是已经加载结束的，可以直接通过对象上的属性访问到资源的所有数据。当你要在引擎中使用这些资源，引擎的 API 接收的都必须是一个加载好的 Asset 对象。

脚本中可以这样定义一个 Asset 属性：

```
// NewScript.js
```

```
cc.Class({
```



```

    extends: cc.Component,

    properties: {

        spriteFrame: {

            default: null,

            type: cc.SpriteFrame

        },

    }

});

```

2、Raw Asset

为了兼容 Cocos2d 的一些既有 API ,我们把保留原始资源文件扩展名资源叫做 "Raw Asset"。图片 (cc.Texture2D) , 字体 (cc.Font) , 声音 (cc.AudioClip) , 粒子 (cc.ParticleAsset) 等资源都是 Raw Asset。Raw Asset 在脚本里由一个 url 字符串来表示 , 当你要在引擎中使用 Raw Asset , 只要把 url 传给引擎的 API , 引擎内部会自动加载这个 url 对应的资源。

在 CClass 中声明 Raw Asset 的属性时 ,要用 url: cc.Texture2D 而不是 type: cc.Texture2D。

```

cc.Class({

    extends: cc.Component,

    properties: {

        textureURL: {

            default: "",


```

```

        url: cc.Texture2D
    }
}
});

```

18.2 如何在属性检查器里设置资源

不论是 Asset 还是 Raw Asset , 只要在脚本中定义好类型 , 就能直接在 属性检查器 很方便地设置资源。假设我们有这样一个组件 :

```

// NewScript.js

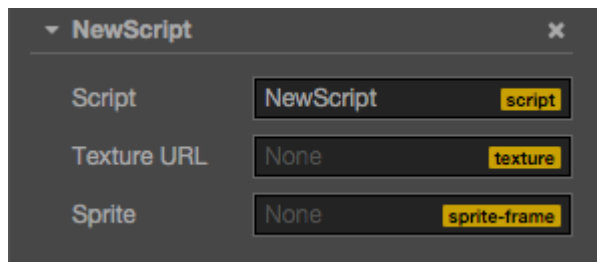
cc.Class({
    extends: cc.Component,

    properties: {
        textureURL: {
            default: "",
            url: cc.Texture2D
        },
        spriteFrame: {
            default: null,
            type: cc.SpriteFrame
        },
    },

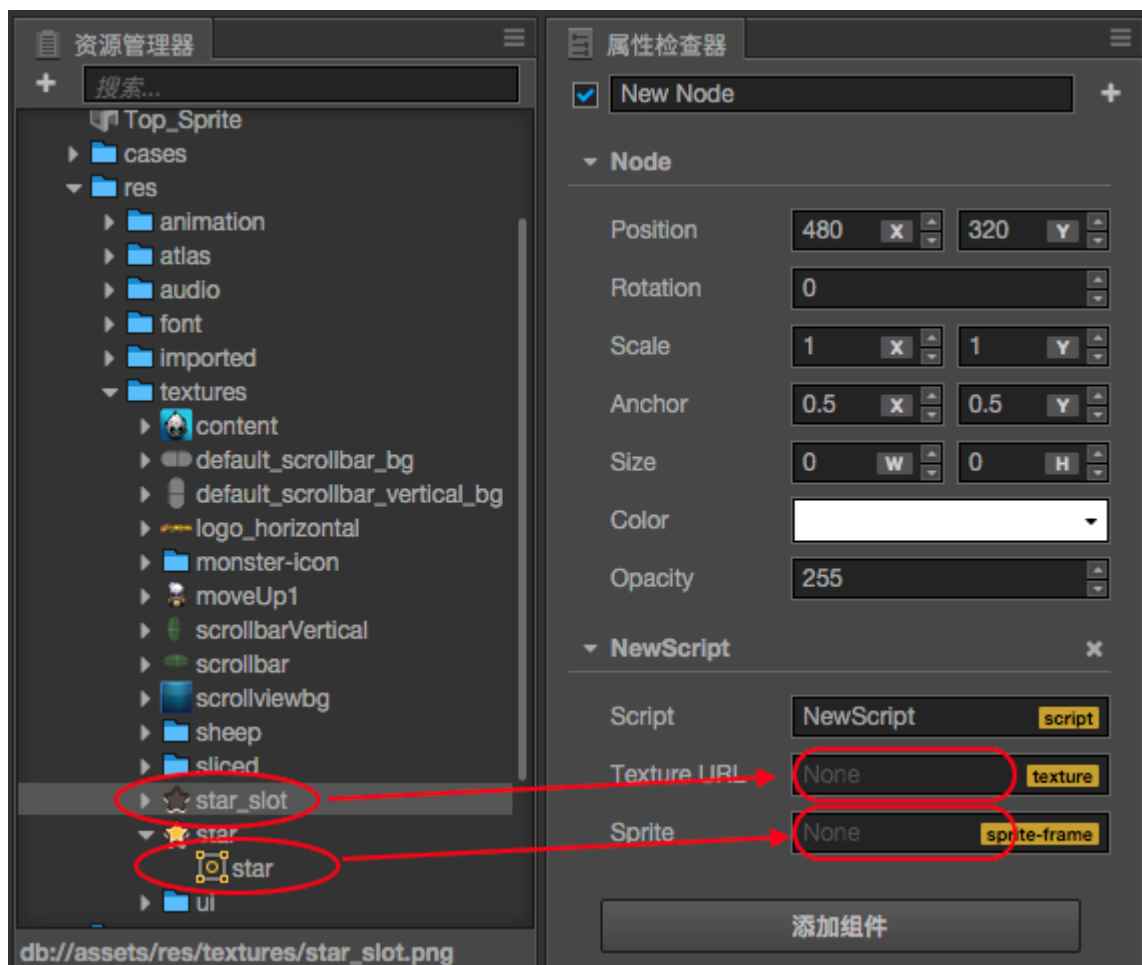
});

```

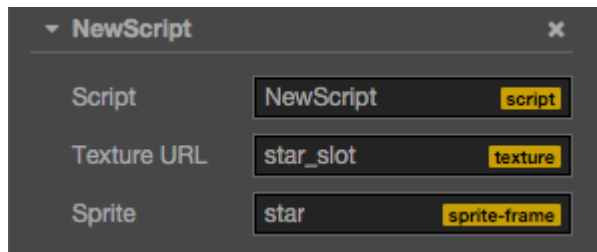
将它添加到场景后，属性检查器 里是这样的：



接下来我们从 资源管理器 里面分别将一张贴图和一个 SpriteFrame 拖到 属性检查器 的对应属性中：



结果如下：



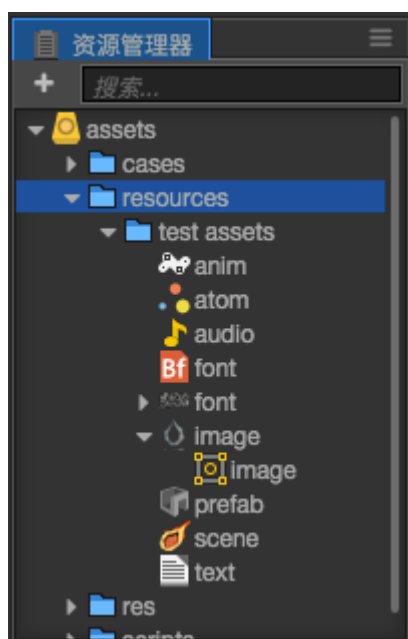
这样就能在脚本里直接拿到设置好的资源：

```
onLoad: function () {  
  
    var spriteFrame = this.spriteFrame;  
  
    var textureURL = this.textureURL;  
  
    spriteFrame.setTexture(textureURL);  
  
}
```

在 属性检查器 里设置资源虽然很直观，但资源只能在场景里预先设好，没办法动态切换。如果需要动态切换，你需要看看下面的内容。

18.3 动态加载

所有需要通过脚本动态加载的资源，都必须放置在 `resources` 文件夹或它的子文件夹下。`resources` 需要在 `assets` 文件夹中手工创建，并且必须位于 `assets` 的根目录，就像这样：



这里的 image/image, prefab, anim, font 都是常见的 Asset , 而 image, audio 则是常见的 Raw Asset。

resources 文件夹里面的资源，可以关联依赖到文件夹外部的其它资源，同样也可以被外部场景或资源引用到。项目构建时，除了已在 构建发布 面板勾选的场景外，resources 文件夹里面的所有资源，连同它们关联依赖的 resources 文件夹外部的资源，都会被导出。所以如果一份资源不需要由脚本直接动态加载，那么不用放在 resources 文件夹里。

1、动态加载 Asset

Creator 提供了 `cc.loader.loadRes` 这个 API 来专门加载那些位于 resources 目录下的 Asset。和 `cc.loader.load` 不同的是，loadRes 一次只能加载单个 Asset。调用时，你只要传入相对 resources 的路径即可，并且路径的结尾处不能包含文件扩展名。

// 加载 Prefab

```

cc.loader.loadRes("test assets/prefab", function (err, prefab) {

    var newNode = cc.instantiate(prefab);

    cc.director.getScene().addChild(newNode);

});

// 加载 AnimationClip

cc.loader.loadRes("test assets/anim", function (err, clip) {

    myNode.getComponent(cc.Animation).addClip(clip, "anim");

});

// 加载 SpriteAtlas ( 图集 ) , 并且获取其中的一个 SpriteFrame
// 注意 atlas 资源文件 ( plist ) 通常会和一个同名的图片文件 ( png ) 放在一个
目录下, 所以需要在第二个参数指定资源类型

cc.loader.loadRes("test assets/sheep", cc.SpriteAtlas, function (err, atlas) {

    var frame = atlas.getSpriteFrame('sheep_down_0');

    sprite.spriteFrame = frame;

});

```

加载独立的 SpriteFrame

图片设置为 Sprite 后，将会在资源管理器中生成一个对应的 SpriteFrame。但如果直接加载 test assets/image，得到的类型将会是 cc.Texture2D。你必须指定第二个参数为资源的类型，才能加载到图片生成的 cc.SpriteFrame：

```

// 加载 SpriteFrame

cc.loader.loadRes("test assets/image", cc.SpriteFrame, function (err,
spriteFrame) {

```

```
myNode.getComponent(cc.Sprite).spriteFrame = spriteFrame;
});
```

如果指定了类型参数，就会在路径下查找指定类型的资源。当你在同一个路径下同时包含了多个重名资源（例如同时包含 `player.clip` 和 `player.psd`），或者需要获取“子资源”（例如获取 `Texture2D` 生成的 `SpriteFrame`），就需要声明类型。

资源释放

`loadRes` 加载进来的单个资源如果需要释放，可以调用 `cc.loader.releaseRes`，`releaseRes` 只能传入一个和 `loadRes` 相同的路径，不支持类型参数。

```
cc.loader.releaseRes("test assets/anim");
```

此外，你也可以使用 `cc.loader.releaseAsset` 来释放一个具体的 `Asset` 实例。

```
cc.loader.releaseAsset(spriteFrame);
```

2、动态加载 Raw Asset

`Raw Asset` 可以直接使用 `url` 从远程服务器上加载，也可以从项目中动态加载。

对远程加载而言，原先 `Cocos2d` 的加载方式不变，使用 `cc.loader.load` 即可。

对项目里的 `Raw Asset`，加载方式和 `Asset` 一样：

// 加载 Texture，不需要后缀名

```
cc.loader.loadRes("test assets/image", function (err, texture) {
    ...
});
```

```
cc.url.raw
```

Raw Asset 加载成功后，如果需要传给一些 url 形式的 API，还是需要给出完整路径才行。你需要用 `cc.url.raw` 进行一次 url 的转换：

// 原 url 会报错！文件找不到

```
var texture = cc.textureCache.addImage("resources/test assets/image.png");
```

// 用 cc.url.raw，此时需要声明 resources 目录和文件后缀名

```
var realUrl = cc.url.raw("resources/test assets/image.png");var texture =  
cc.textureCache.addImage(realUrl);
```

3、资源批量加载

`cc.loader.loadResAll` 可以加载相同路径下的多个资源：

// 加载 test assets 目录下所有资源

```
cc.loader.loadResAll("test assets", function (err, assets) {
```

```
    // ...
```

```
});
```

// 加载 sheep.plist 图集中的所有 SpriteFrame

```
cc.loader.loadResAll("test assets/sheep", cc.SpriteFrame, function (err, assets)
```

```
{
```

```
    // assets 是一个 SpriteFrame 数组，已经包含了图集中的所有
```

```
SpriteFrame。
```

// 而 loadRes('test assets/sheep', function (err, atlas) {...}) 获得的则是整个 SpriteAtlas 对象。

```
});
```


第十九章 模块化脚本

Cocos Creator 允许你将代码拆分成多个脚本文件，并且让它们相互调用。要实现这点，你需要了解如何在 Cocos Creator 中定义和使用模块，这个步骤简称为模块化。

如果你还不确定模块化究竟能做什么，模块化相当于：

- C/C++ 中的 `include`
- C# 中的 `using`
- Java 和 Python 中的 `import`
- HTML 中的 `<link>`

模块化使你可以在 Cocos Creator 中引用其它脚本文件：

- 访问其它文件导出的参数
- 调用其它文件导出的方法
- 使用其它文件导出的类型
- 使用或继承其它 Component

Cocos Creator 中的 JavaScript 使用和 Node.js 几乎相同的 CommonJS 标准来实现模块化，简单来说：

- 每一个单独的脚本文件就构成一个模块
- 每个模块都是一个单独的作用域
- 以同步的 `require` 方法来引用其它模块
- 设置 `module.exports` 为导出的变量

如果你还不太明白，没关系，下面会详细讲解。

在本文中，“模块”和“脚本”这两个术语是等价的。所有“备注”都属于进阶内容，一开始不需要了解。

不论模块如何定义，所有用户代码最终会由 Cocos Creator 编译为原生的 JavaScript，可直接在浏览器中运行。

19.1 引用模块

1、require

除了 Cocos Creator 提供的接口，所有用户定义的模块都需要调用 `require` 来访问。例如我们有一个组件定义在 `Rotate.js`：

```
// Rotate.js  
  
cc.Class({  
    extends: cc.Component,  
  
    // ...  
  
});
```

现在要在别的脚本里访问它，可以：

```
var Rotate = require("Rotate");
```

`require` 返回的就是被模块导出的对象，通常我们都会将结果立即存到一个变量（`var Rotate`）。传入 `require` 的字符串就是模块的文件名，这个名字不包含路径也不包含后缀，而且大小写敏感。

2、require 完整范例

接着我们就可以使用 `Rotate` 派生一个子类，新建一个脚本 `SinRotate.js`：

```
// SinRotate.js
```

```
var Rotate = require("Rotate");

var SinRotate = cc.Class({

    extends: Rotate,

    update: function (dt) {

        this.rotation += this.speed * Math.sin(dt);

    }

});
```

这里我们定义了一个新的组件叫 SinRotate，它继承自 Rotate，并对 update 方法进行了重写。

同样的这个组件也可以被其它脚本接着访问，只要用 require("SinRotate")。

备注：

- require 可以在脚本的任何地方任意时刻进行调用。
- 游戏开始时会自动 require 所有脚本，这时每个模块内部定义的代码就会被执行一次，所以之后无论又被 require 几次，返回的始终是同一份实例。
- 调试时，可以随时在 Developer Tools 的 Console 中 require 项目里的任意模块。

19.2 定义模块

1、定义组件

每一个单独的脚本文件就是一个模块，例如前面新建的脚本 Rotate.js：

```
// Rotate.js
```

```

var Rotate = cc.Class({

    extends: cc.Component,

    properties: {

        speed: 1

    },

    update: function () {

        this.transform.rotation += this.speed;

    }

});

```

当你在脚本中声明了一个组件，Cocos Creator 会默认把它导出，其它脚本直接 `require` 这个模块就能使用这个组件。

2、定义普通 JavaScript 模块

模块里不单单能定义组件，实际上你可以导出任意 JavaScript 对象。假设有个脚本 `config.js`

// config.js

```

var cfg = {

    moveSpeed: 10,

    version: "0.15",

    showTutorial: true,

```

```
    load: function () {  
        // ...  
    }  
};  
  
cfg.load();
```

现在如果我们要在其它脚本中访问 config 对象：

// player.js

```
var config = require("config");  
  
cc.log("speed is", config.moveSpeed);
```

结果会有报错："TypeError: Cannot read property 'moveSpeed' of null"，这是

因为 cfg 没有被导出。由于 require 实际上获取的是目标脚本内的

module.exports 变量，所以我们还需要在 config.js 的最后设置 module.exports =

config：

// config.js - v2

```
var cfg = {  
    moveSpeed: 10,  
    version: "0.15",  
    showTutorial: true,  
  
    load: function () {
```

```
        // ...

    }

};

cfg.load();

module.exports = cfg;
```

这样 `player.js` 便能正确输出：`"speed is 10"`。

那为什么定义 `Component` 时不用设置 `exports` ？

因为 `Component` 是 `Cocos Creator` 中的特殊类型，如果一个脚本定义了 `Component` 却没有声明 `exports`，`Cocos Creator` 会自动将 `exports` 设置为 `Component`。

备注：

- 在 `module` 上增加的其它变量是不能导出的，也就是说 `exports` 不能替换成其它变量名，系统只会读取 `exports` 这个变量。

19.3 更多示例

1、导出变量

- `module.exports` 默认是一个空对象（`{}`），可以直接往里面增加新的字段。

// foobar.js:

```
module.exports.foo = function () {
    cc.log("foo");
};

module.exports.bar = function () {
    cc.log("bar");
};
```

// test.js:

```
var foobar = require("foobar");

foobar.foo();    // "foo"

foobar.bar();    // "bar"
```

};	
----	--

- `module.exports` 的值可以是任意 JavaScript 类型。

<pre>// foo.js: module.exports = { FOO: function () { this.type = "foo"; }, bar: "bar" };</pre>	<pre>// test.js: var foo = require("foo"); var foo = new foo.FOO(); cc.log(foo.type); // "foo" cc.log(foo.bar); // "bar"</pre>
---	--

2、封装私有变量

每个脚本都是一个单独的作用域，在脚本内使用 `var` 定义的局部变量，将无法被模块外部访问。我们可以很轻松的封装模块内的私有变量：

```
// foo.js:
var dirty = false; module.exports = {
  setDirty: function () {
    dirty = true;
  },
  isDirty: function () {
```

```
        return dirty;

    },

};

// test1.js:

var foo = require("foobar");

cc.log(typeof foo.dirty);          // "undefined"

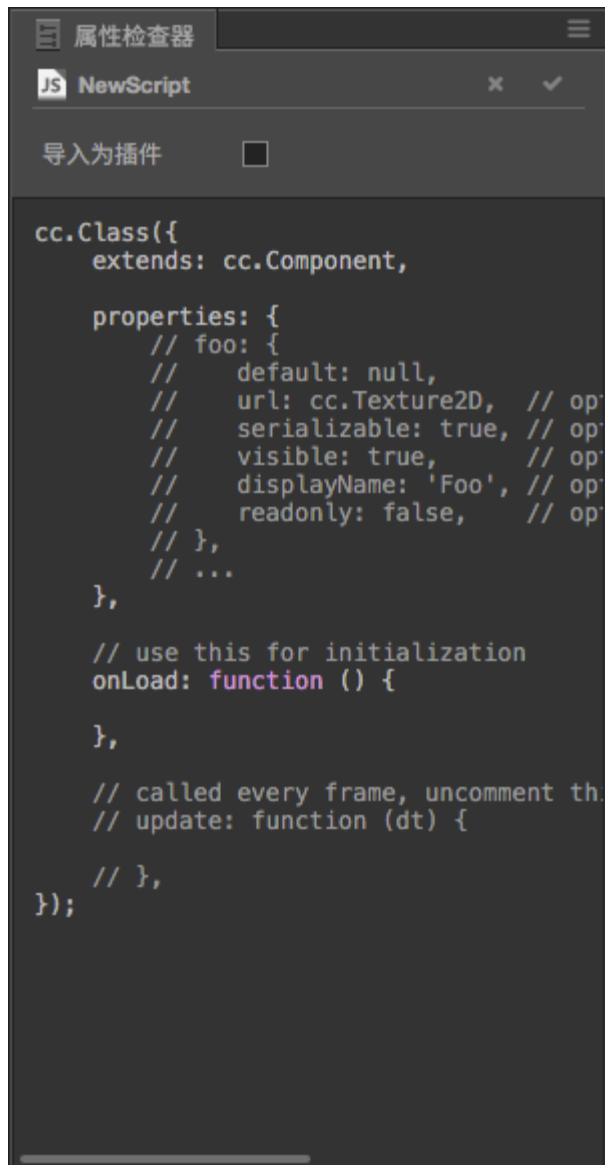
foo.setDirty();

// test2.js:

var foo = require("foobar");

cc.log(foo.isDirty());             // true
```


第二十章 插件脚本



在 资源管理器 中选中任意一个脚本,就能在 属性检查器 中看到这样一个设置界面,我们可以在这里设置脚本是否“导入为插件”。导入为插件是一个不太常用到的选项,初学者简单了解即可。

- 对组件脚本,数据逻辑而言,一般默认都取消这个选项,这样的脚本简称普通脚本。
- 对第三方插件,或者底层插件,就有可能需要选中选项,这样的脚本简称插件脚本。

这个选项只和脚本有关,具体影响有几个方面:

类型：	普通脚本	插件脚本
声明组件	支持	不支持
模块化	支持, 可以通过 require 引用其它普通脚本, 不能 require 插件脚本	不提供, 也不能 require 普通脚本
变量作用域	每个脚本内定义的局部变量不会暴露到全局	脚本内不在任何函数内的局部变量都会暴露成全局变量
use strict	强制开启, 未定义的变量不能赋值	需要手动声明, 否则未定义的变量一旦赋值就会变成全局变量
脚本导入编辑器时	脚本中的 ES2015 特性会先 转译 , 再进入统一的模块化解析	不做任何处理
项目构建阶段时	所有普通脚本都会打包成单个脚本文件, 非“调试模式”下还会压缩	不进行打包, 非“调试模式”下会被压缩
SourceMap	支持	不支持

脚本加载顺序

脚本加载顺序如下：（其中每一步都会等到上一步结束后才开始）

1. Cocos2d 引擎
2. 插件脚本（有多个的话按项目中的路径字母顺序依次加载）
3. 普通脚本（打包后只有一个文件，内部按 require 的依赖顺序依次初始化）

目标平台兼容性

插件发布后将直接被目标平台加载，所以请检查插件的目标平台兼容性，否则项目发布后插件有可能不能运行。

- 目标平台不提供原生 node.js 支持：例如很多 [npm](#) 模块都直接或间接依赖于 node.js，这样的话发布到原生或网页平台后是不能用的。
- 依赖 DOM API 的插件将无法发布到原生平台：网页中可以使用大量的前端插件，例如 jQuery，不过它们有可能依赖于浏览器的 DOM API。依赖这些 API 的插件不能用于原生平台中。

全局变量

由于所有插件脚本都保证了会在普通脚本之前加载，那么除了用来加载插件，你还可以利用这个特性声明一些特殊的全局变量。你可以在项目中添加这样一个脚本，并且设置“导入为插件”：

```
/* globals.js */  
  
// 定义新建组件的默认值 var DEFAULT_IP = "192.168.1.1";  
  
// 定义组件开关 var ENABLE_NET_DEBUGGER = true;  
  
// 定义引擎 API 缩写（仅适用于构造函数） var V2 = cc.Vec2;
```

在上面的插件脚本中，因为作用域是在全局，并不是在脚本内部，所以直接写 `var DEFAULT_IP = ...` 就能声明全局变量。

接下来你就能在任意的普通脚本中直接访问它们：

```
/* network.js */  
  
cc.Class({  
    extends: cc.Component,
```

```

    properties: {
        ip: {
            default: DEFAULT_IP
        }
    }
});

/* network_debugger.js */

if (ENABLE_NET_DEBUGGER) {
    // ENABLE_NET_DEBUGGER 时这个组件才生效

    cc.Class({
        extends: cc.Component,

        properties: {
            location: {
                default: new V2(100, 200)
            }
        },

        update: function () {
            ...
        },

    });
} else {
    // 否则这个组件什么也不做

```

```
cc.Class({  
    extends: cc.Component  
  
    });  
}
```

在这个案例中，由于 `network.js` 和 `network_debugger.js` 等脚本加载时就已经用到了 `globals.js` 的变量。如果 `globals.js` 不是插件脚本，则每个可能用到那些全局变量的脚本都要在最上面声明 `require("globals");`，才能保证 `globals.js` 先加载。

但假如一个全局变量本身就是要在组件 `onLoad` 时才能初始化，那么建议直接在普通脚本的 `onLoad` 里直接使用 `window.foo = bar` 来声明全局变量，不需要使用插件脚本，详见[通过全局变量访问](#)。

- 你应当很谨慎地使用全局变量，当你要用全局变量时，应该很清楚自己在做什么，我们并不推荐滥用全局变量，即使要用也最好保证全局变量只读。
- 添加全局变量时，请小心不要和系统已有的全局变量重名。
- 你需要小心确保全局变量使用之前都已初始化和赋值。
- 你可以在插件脚本中自由封装或者扩展 Cocos2d 引擎，但这会提高团队沟通成本，导致脚本难以复用，合并脚本时也容易冲突。

第二十一章 第三方 JavaScript 模块引用

目前在 Cocos Creator 中，暂时只支持对第三方 npm 模块引用，当然，如果开发者编写的脚本，符合 Node.js 的标准，也是可以被引用的。这里不深入介绍 Node.js 和 npm，放上官方文档页面大家可以自己了解：

- [Node.js modules](#)
- [What is npm](#)

21.1 如何使用 npm 模块

当你找到需要的 npm 模块后，第一步需要在自己的项目目录中安装该模块（以 box2dweb-commonjs 为例）：

```
> cd /path/to/project
```

```
> npm install box2dweb-commonjs
```

（这一步要确保项目的所有父级目录下都不含 node_modules 文件夹，否则会优先安装到父目录下）

第二步只需要在你需要使用该模块的组件脚本中 require 这个模块即可开始使用：

```
var box2d = require('box2dweb-commonjs');
```

这样你的游戏就能加载到第三方模块，打包过程中，第三方模块也会被一起导出。

注意事项

1. 仅支持纯 JavaScript 模块：npm 中包含诸多各式各样的模块，其中有很多使用了 Node.js 的 API，这样的模块是不支持的，因为组件最终的运行环境不在 Node.js。

2. 原生环境不支持 DOM API：众所周知，浏览器中包含大量的 DOM API，比如 jQuery 就是著名的 DOM 操作库。使用这些 API 的模块虽然可以在 HTML5 环境中运行，但却不可以在原生环境中运行，因为原生环境中不包含提供 DOM API 的页面排版引擎。
3. 注意模块嵌套依赖：npm 中的模块常常会嵌套依赖其它模块，这种嵌套层次有可能很深，导致大量的第三方模块都被加载进来。建议发生嵌套依赖时，小心检查依赖的模块是否都符合上面两点，并且小心依赖模块过多，导致编译时间过长，游戏体积过大。

21.2 未来其他可能的模块依赖方式

理论上，require 可以用来引用任何 JavaScript 脚本。虽然目前不建议通过这种方式引用第三方模块，不过以后会做到更好的支持。

另外，很多开发者习惯在 index.html 中引用外部 JavaScript 脚本，甚至离线脚本。目前 Cocos Creator 并没有开放如何使用 index.html，开发者只能在打包后的页面文件中手动添加引用，不过我们正在研究如何提供更友好的方式让开发者定制 index.html。