# COCOS CREATOR

# User Manual

# 目錄

## 新手入门



## 基本工作流程

# 子系统介绍

# 进阶使用

# Cocos Creator v1.6.x 用户手册

欢迎使用 Cocos Creator 用户手册！本手册包括详尽的使用说明、面向不同职能用户的工作流程和 step by step 的新手教程。能够帮您快速掌握使用 Cocos Creator 开发跨平台游戏的方法。

## 特别推荐

Cocos Creator 现已支持导出场景和 UI 到 cocos2d-x 引擎，详细信息请参阅 C++/Lua 引擎支持。

## 总导读

- Cocos Creator 入门
- 资源工作流程
- 场景制作工作流程
- 图像和渲染
- UI 系统
- 编程开发指南
- 动画系统
- 碰撞系统
- 音乐和音效
- 发布跨平台游戏
- 扩展编辑器
- 进阶主题
- 第三方 SDK 集成

## 视频教程

前往视频教程页面。

## 演示和范例项目

注意，所有 Github 上的演示和范例项目都会跟随版本进行更新，默认分支对应目前最新的 Cocos Creator 版本，老版本的项目会以 `v0.7` 这样的分支名区分，分支名会和相同版本的 Cocos Creator 对应，下载使用的时候请注意。

- 范例集合：从基本的组件到交互输入，这个项目里包括了 case by case 的功能点用法介绍。
- Star Catcher：也就是 快速上手 文档里分步讲解制作的游戏。
- 腾讯合作开发的21点游戏
- UI 展示 Demo
- Duang Sheep：复制 FlappyBird 的简单游戏，不过主角换成了绵羊。
- 暗黑斩 Cocos Creator 复刻版：由 Veewo Games 独家授权原版暗黑斩资源素材，在 Cocos Creator 里复刻的演示项目
- i18n 游戏多语言支持范例

# Cocos Creator 入门

本章内容将为您介绍 Cocos Creator 的定位、功能和特色，以及如何快速上手使用 Cocos Creator 开发包括 iOS, Android, HTML5 和 PC 客户端的跨平台游戏产品！

本章包括以下能够让您用最快速度上手的教程内容：

- 关于 Cocos Creator
- 安装和启动
- 使用 Dashboard
- Hello World!
- 快速上手: 制作第一个游戏
- 代码编辑环境配置
- 项目结构
- 编辑器基础
- Cocos2d-x 用户上手指南
- 获取帮助和支持

---

让我们现在就马上开始，请先阅读 关于 Cocos Creator 来了解这是一个怎样的游戏开发工具，能为开发者做什么。

# 关于 Cocos Creator

- **Q**: Cocos Creator 是游戏引擎吗?
- **A**: 它是一个完整的游戏开发解决方案，包括了 cocos2d-x 引擎的 JavaScript 实现（不需要学习一个新的引擎），以及能让你更快速开发游戏所需要的各种图形界面工具

- **Q**: Cocos Creator 的编辑器是什么样的?
- **A**: 完全为引擎定制打造，包含从设计、开发、预览、调试到发布的整个工作流所需的全功能一体化编辑器

- **Q**: 我不会写程序，也能使用 Cocos Creator 吗?
- **A**: 当然！Cocos Creator 编辑器提供面向设计和开发的两种工作流，提供简单顺畅的分工合作方式。

- **Q**: 我使用 Cocos Creator 能开发面向哪些平台的游戏?
- **A**: Cocos Creator 目前支持发布游戏到 Web、Android 和 iOS，以及点开即玩原生性能的 Cocos Play 手机页游平台，真正实现一次开发，全平台运行。

- **Q**: Cocos Creator 能开发Cocos2d-x C++ 或者 Lua 的游戏吗?

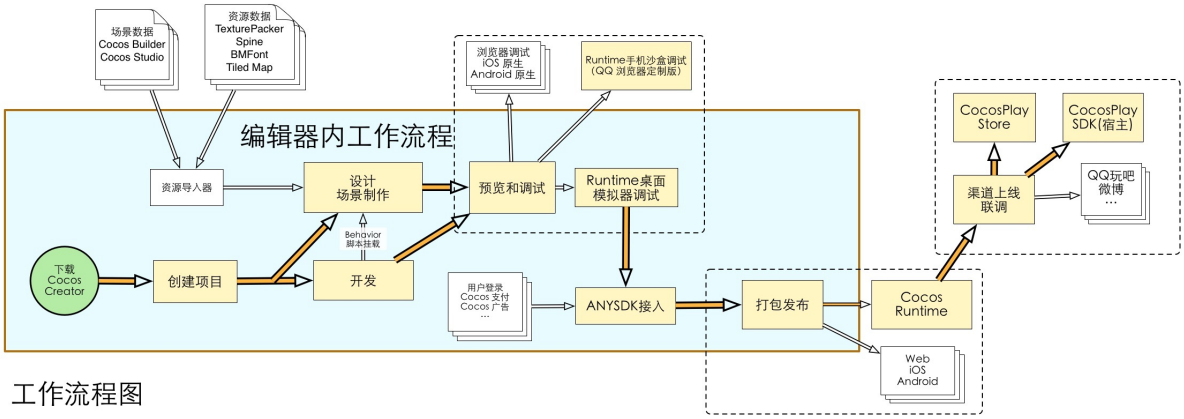
- **A**: Cocos Creator 可以通过安装C++/Lua for Creator插件，在编辑器里编辑UI和场景，导出通用的数据文件，在Cocos2d-x引擎中进行加载运行。

## 产品定位

Cocos Creator 是以内容创作为核心的游戏开发工具，在 Cocos2d-x 基础上实现了彻底脚本化、组件化和数据驱动等特点。

## 工作流程说明

在开发阶段，Cocos Creator 已经能够为用户带来巨大的效率和创造力提升，但我们所提供的工作流远不仅限于开发层面。对于成功的游戏来说，开发和调试、商业化 SDK 的集成、多平台发布、测试、上线这一整套工作流程不光缺一不可，而且要经过多次的迭代重复。




工作流程图

Cocos Creator 将整套手机页游解决方案整合在了编辑器工具里，无需在多个软件之间穿梭，只要打开 Cocos Creator 编辑器，各种一键式的自动化流程就能花最少的时间精力，解决上述所有问题。开发者就能够专注于开发阶段，提高产品竞争力和创造力！

### 创建或导入资源

将图片、声音等资源拖拽到编辑器的 **资源管理器** 面板中，即可完成资源导入。

此外，你也可以在编辑器中直接创建场景、预制、动画、脚本、粒子等各类资源。

### 建造场景内容

项目中有了一些基本资源后，我们就可以开始搭建场景了，场景是游戏内容最基本的组织方式，也是向玩家展示游戏的基本形态。

我们通过 **场景编辑器** 将添加各类节点，负责展示游戏的美术音效资源，并作为后续交互功能的承载。

### 添加组件脚本，实现交互功能

我们可以为场景中的节点挂载各种内置组件和自定义脚本组件，来实现游戏逻辑的运行和交互。包括从最基本的动画播放、按钮响应，到驱动整个游戏逻辑的主循环脚本和玩家角色的控制。几乎所有游戏逻辑功能都是通过挂载脚本到场景中的节点来实现的。

### 一键预览和发布

搭建场景和开发功能的过程中，你可以随时点击预览来查看当前场景的运行效果。使用手机扫描二维码，可以立即在手机上预览游戏。当开发告一段落时，通过 **构建发布** 面板可以一键发布游戏到包括桌面、手机、Web 等多个平台。

## 功能特性

Cocos Creator 功能上的突出特色包括：

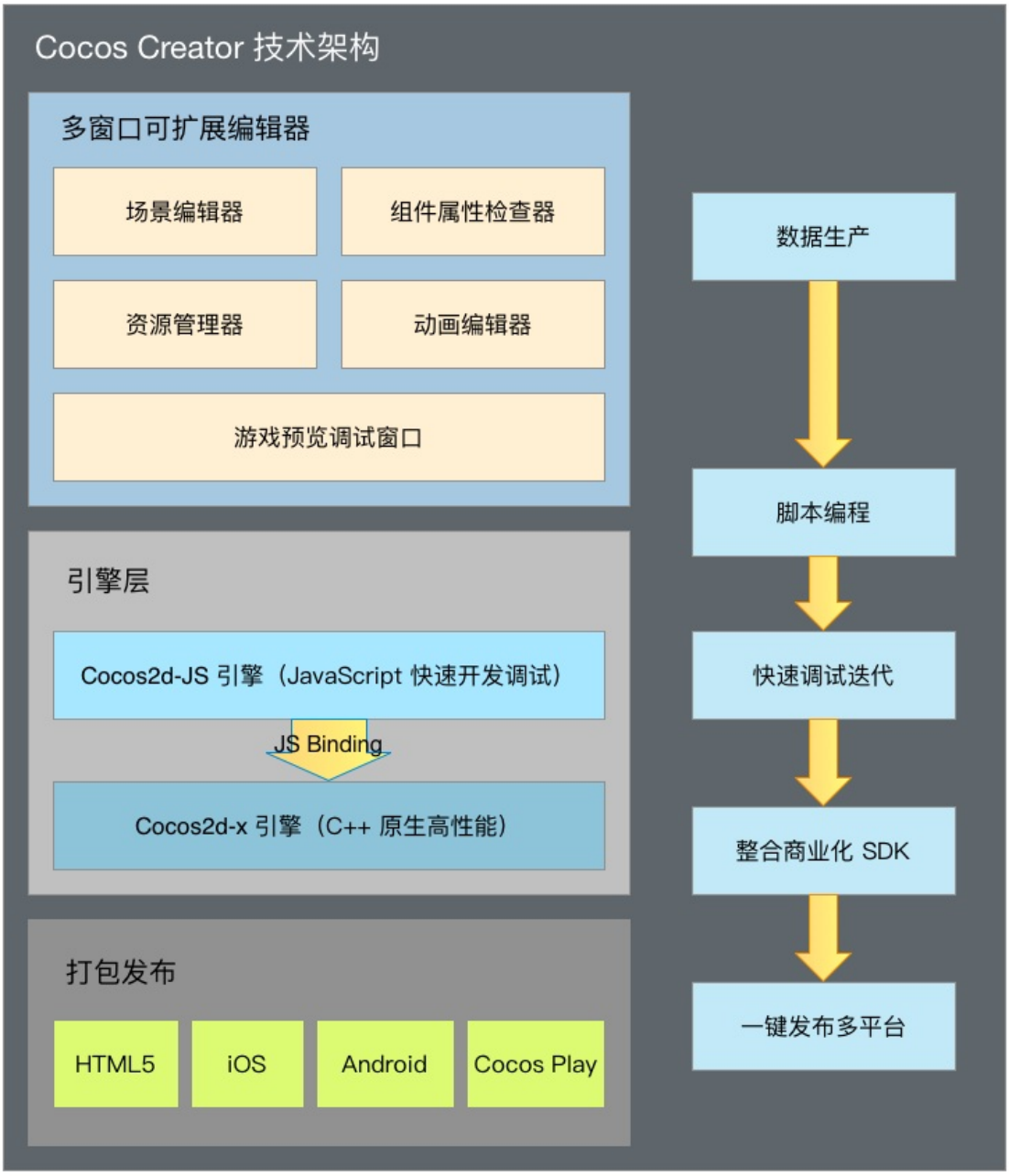
- 脚本中可以轻松声明可以在编辑器中随时调整的数据属性，对参数的调整可以由设计人员独立完成。
- 支持智能画布适配和免编程元素对齐的 UI 系统可以完美适配任意分辨率的设备屏幕。
- 专为 2D 游戏打造的动画系统，支持动画轨迹预览和复杂曲线编辑功能。
- 动态语言支持的脚本化开发，使得动态调试和移动设备远程调试变得异常轻松。
- 借助 Cocos2d-x 引擎，在享受脚本化开发的便捷同时，还能够一键发布到各类桌面和移动端平台，并保持原生级别的超高性能
- 脚本组件化和开放式的插件系统为开发者在不同深度上提供了定制工作流的方法，编辑器可以大尺度调教来适应不同团队和项目的需要。

## 架构特色

Cocos Creator 包含游戏引擎、资源管理、场景编辑、游戏预览和发布等游戏开发所需的全套功能，并且将所有的功能和工具链都整合在了一个统一的应用程序里。

它以数据驱动和组件化作为核心的游戏开发方式，并且在此基础上无缝融合了 Cocos 引擎成熟的 JavaScript API 体系，能够一方面适应 Cocos 系列引擎开发者用户习惯，另一方面为美术和策划人员提供前所未有的内容创作生产和即时预览测试环境。

编辑器在提供强大完整工具链的同时，提供了开放式的插件架构，开发者能够用 Html + JavaScript 等前端通用技术轻松扩展编辑器功能，定制个性化的工作流程。




引擎和编辑器的结合，带来的是数据驱动和组件化的功能开发方式，以及设计和程序两类人员的完美分工合作：

- 设计师在场景编辑器中搭建场景的图像表现
- 程序员开发可以挂载到场景任意物体上的功能组件
- 设计师负责为需要展现特定行为的物体挂载组件，并通过调试改善各项参数
- 程序员开发游戏所需要的数据结构和资源
- 设计师通过图形化的界面配置好各项数据和资源 -（就这样从简单到复杂，各种你能想像到的工作流程都可以实现）

以工作流为核心的开发理念，让不同职能的开发者能够快速找到最大化自己作用的工作切入点，并能够默契流畅的和团队其他成员配合。

# 使用说明

Cocos Creator 是一个支持 Windows 和 Mac 跨平台运行的应用程序，双击即可启动运行。相比传统的 Cocos2d-x 工作流程，将配置开发环境的要求完全免除，运行之后就可以立刻开始游戏内容创作或功能开发。

在数据驱动的工作流基础上，场景的创建和编辑成为了游戏开发的中心，设计工作和功能开发可以同步进行，无缝协作，不管是美术、策划还是程序员，都可以在生产过程的任意时刻点击预览按钮，在浏览器、移动设备模拟器或移动设备真机上测试游戏的最新状态。

程序员和设计人员现在可以实现各式各样的分工合作，不管是先搭建场景，再添加功能，还是先生产功能模块再由设计人员进行组合调试，Cocos Creator 都能满足开发团队的需要。脚本中定义的属性能够以最适合的视觉体验呈现在编辑器中，为内容生产者提供便利。

场景之外的内容资源可以由外部导入，比如图片、声音、图集、骨骼动画等等，除此之外我们还在不断完善编辑器生产资源的能力，包括目前已经完成的动画编辑器，美术人员可以使用这个工具制作出非常细腻富有表现力的动画资源，并可以随时在场景中看到动画的预览。

最后，开发完成的游戏可以通过图形工具一键发布到各个平台，从设计研发到测试发布，Cocos Creator 全部帮您搞定。

# 安装和启动

如果阅读此文档时您还没有下载和启动 Cocos Creator，请按照以下步骤开始。

# 下载 Cocos Creator

您可以通过访问 Cocos Creator 产品首页 上的下载链接获得 Cocos Creator 的安装包。

下载完成后双击安装包。

## Windows 安装说明

从 v1.3.0 开始，Windows 版 Cocos Creator 将不提供 32 位操作系统支持。

Windows 版的安装程序是一个 `.exe` 可执行文件，通常命名会是 `CocosCreator_vX.X.X_20XXXXXX_setup.exe`，其中 `vX.X.X` 是 Cocos Creator 的版本号，如 `v1.2.2`，后面的一串数字是版本日期编号。

**注意** 日期编号在使用内测版时会更新的比较频繁，注意如果当前 PC 上已安装的版本号和安装包的版本号相同时，无法自动覆盖安装注意相同版本号的安装包，需要先卸载之前的版本才能继续安装。

应用的安装路径默认选择了 `C:\CocosCreator`，可以在安装过程中进行指定。

Cocos Creator 将会占据系统盘中大约 1.25 GB 的空间，请在安装前整理您的系统盘空间。

**注意**：如果出现安装失败，请尝试通过命令行执行安装程序：

```
CocosCreator_v1.2.0_2016080301_setup.exe /exelog "exe_log.txt" /L*V "msi_log.txt"
```

用以下命令执行，或为安装程序创建一个快捷方式，并将该命令行参数填入快捷方式的 `目标` 属性中。然后将生成的安装日志（`exe_log.txt` 和 `msi_log.txt`）提交给开发团队寻求帮助。

## Mac 安装说明

Mac 版 Cocos Creator 的安装程序是 DMG 镜像文件，双击 DMG 文件，然后将 `CocosCreator.app` 拖拽到您的 **应用程序** 文件夹快捷方式，或任意其他位置。然后双击复制出来的 `CocosCreator.app` 就可以开始使用了。

**注意**：如果初次运行时出现下载的应用已损坏的提示，请前往并设置 系统偏好设置->安全性与隐私->允许任何来源的应用，首次打开后您可以马上恢复您的安全与隐私设置。

## 操作系统要求

Cocos Creator 所支持的系统环境是：

- Mac OS X 所支持的最低版本是 OS X 10.9。
- Windows 所支持的最低版本是 Windows 7 64位。

# 运行 Cocos Creator

在 Windows 系统，双击解压后文件夹中的 `CocosCreator.exe` 文件即可启动 Cocos Creator。

在 Mac 系统，双击解压后的 `CocosCreator.app` 应用图标即可启动 Cocos Creator。

您可以按照习惯为入口文件设置快速启动、Dock 或快捷方式，方便您随时运行使用。

### 禁用 GPU 加速

对于部分 windows 操作系统和显卡型号，可能会遇到

```
This browser does not support WebGL...
```

的报错信息，是显卡驱动对编辑器 WebGL 渲染模式的支持不正确导致的，如果出现这种情况，可以尝试使用命令行运行 `CocosCreator.exe` 并加上 `--disable-gpu` 运行参数，来禁用 GPU 加速功能，可以绕开部分显卡驱动的问题。

## 使用 Cocos 开发者帐号登录

如果您不需要发布游戏到原生平台，以上的两步简单操作就能为您准备好使用 Cocos Creator 制作游戏的一切开发环境。

Cocos Creator 启动后，会进入 Cocos 开发者帐号的登录界面。登录之后就可以享受我们为开发者提供的各种在线服务、产品更新通知和各种开发者福利。

如果之前没有 Cocos 开发者帐号，您可以使用登录界面中的 **注册** 按钮前往 Cocos 开发者中心进行注册。或直接使用下面的链接：

https://passport.cocos.com/auth/signup

注册完成后就可以回到 Cocos Creator 登录界面完成登录了！验证身份后，我们就会进入 Dashboard 界面。除了手动登出或登录信息过期，其他情况下都会用本地 session 保存的信息自动登录。

## 版本兼容性和回退方法

Cocos Creator 版本升级时，新版本的编辑器可以打开旧版本的项目，但当您在项目开发到一半时升级新版本的 Cocos Creator 时也可能会遇到一些问题。因为在早期版本中引擎和编辑器的实现可能存在 bug 和其他不合理的问题，这些问题可以通过用户项目和脚本的特定使用方法来规避，但当后续版本中修复了这些 bug 和问题时就可能会对现有项目造成影响。

在发现这种版本升级造成的问题时，除了联系开发团队寻求解决办法，您也可以卸载新版本的 Cocos Creator 并重新安装旧版本。安装旧版本过程中可能遇到的问题有：

- [Windows] 您可能会遇到安装旧版本时提示「已经有一个更新版本的应用程序已安装」的提示，如果确定已经通过控制面板正确卸载了新版本的 Cocos Creator 还不能安装旧版本，可以访问 微软官方解决无法安装或卸载程序 的帮助页，按照提示下载小工具并修复损坏的安装信息，即可继续安装旧版本了。
- 使用新版本 Cocos Creator 打开过的项目，在旧版本 Cocos Creator 中打开可能会遇到编辑器面板无法显示内容的问题，可以尝试选择主菜单中的「布局->恢复默认布局」来进行修复。

## 原生发布相关配置

如果您只想开发 Web 平台的游戏，完成上面的步骤就足够了。如果您希望发布游戏到原生平台，请阅读相关开发环境的设置说明 安装配置原生开发环境

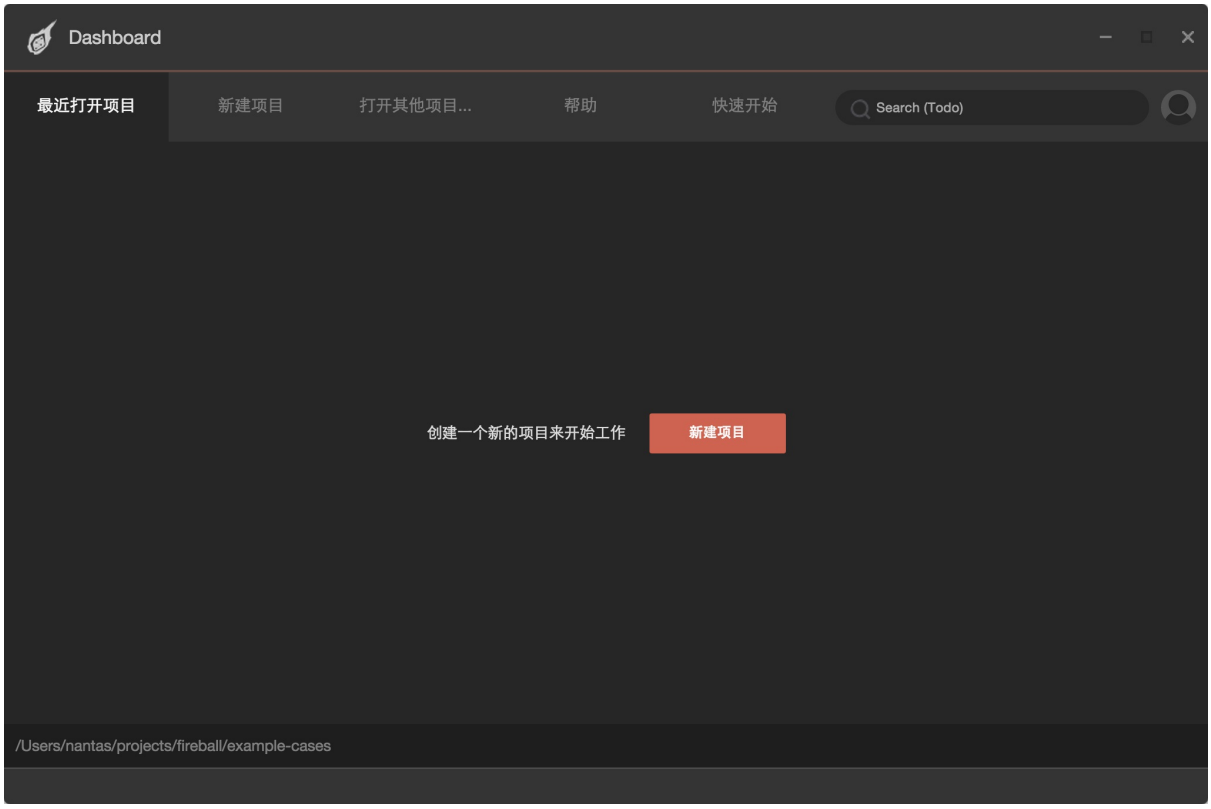
---

继续前往 使用 Dashboard 说明文档。

# Dashboard

启动 Cocos Creator 并使用 Cocos 开发者帐号登录以后，就会打开 Dashboard 界面，在这里你可以新建项目、打开已有项目或获得帮助信息。

## 界面总览



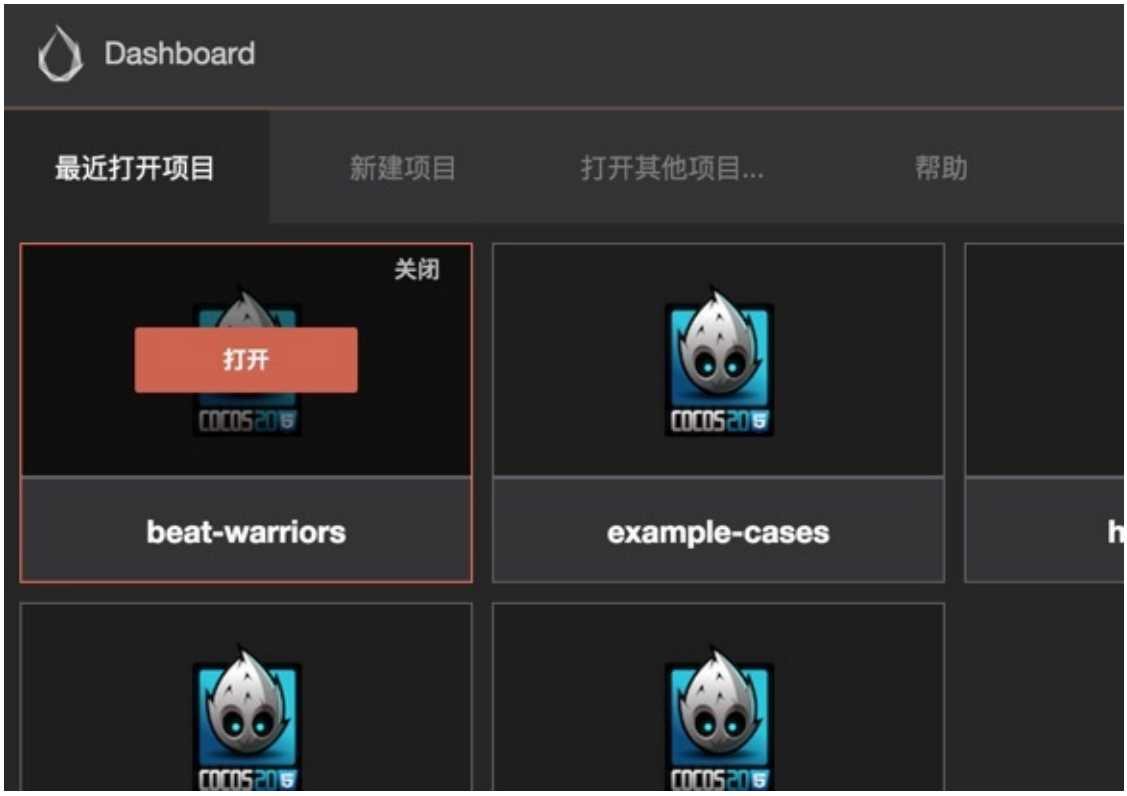

上图所示的就是 Cocos Creator 的 Dashboard 界面，包括以下几种选项卡：

- **最近打开项目**: 列出最近打开项目，第一次运行 Cocos Creator 时，这个列表是空的，会提示新建项目的按钮。
- **新建项目**: 选择这个选项卡，会进入到 Cocos Creator 新项目创建的指引界面。
- **打开其他项目**: 如果你的项目没有在最近打开的列表里，你也可以点击这个按钮来浏览和选择你要打开的项目。
- **帮助**: 帮助信息，一个包括各种新手指引信息和文档的静态页面。

下面我们来依次介绍这些分页面。

### 最近打开项目

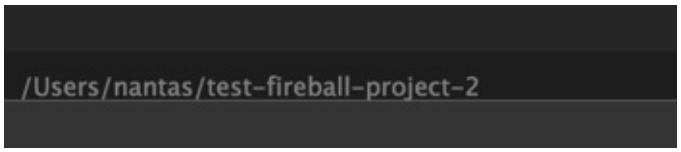
你可以通过 **最近打开项目** 选项卡快速访问近期打开过的项目。第一次运行 Cocos Creator 时，这个列表是空的，在界面上会显示 **新建项目** 的按钮。你可以在创建 了一些项目后回来，并看到你新建的项目出现在列表里。




当你的鼠标悬停在一个最近打开项目的条目上时，会显示出可以对该项目进行操作的行为：

- 点击 **打开** 在 Cocos Creator 编辑器中打开该项目
- 点击 **关闭** 将该项目从最近打开项目列表中移除，这个操作不会删除实际的项目文件夹。

此外，当鼠标点击选中或悬停在项目上时，你能够在 Dashboard 下方的状态栏看到该项目所在路径。



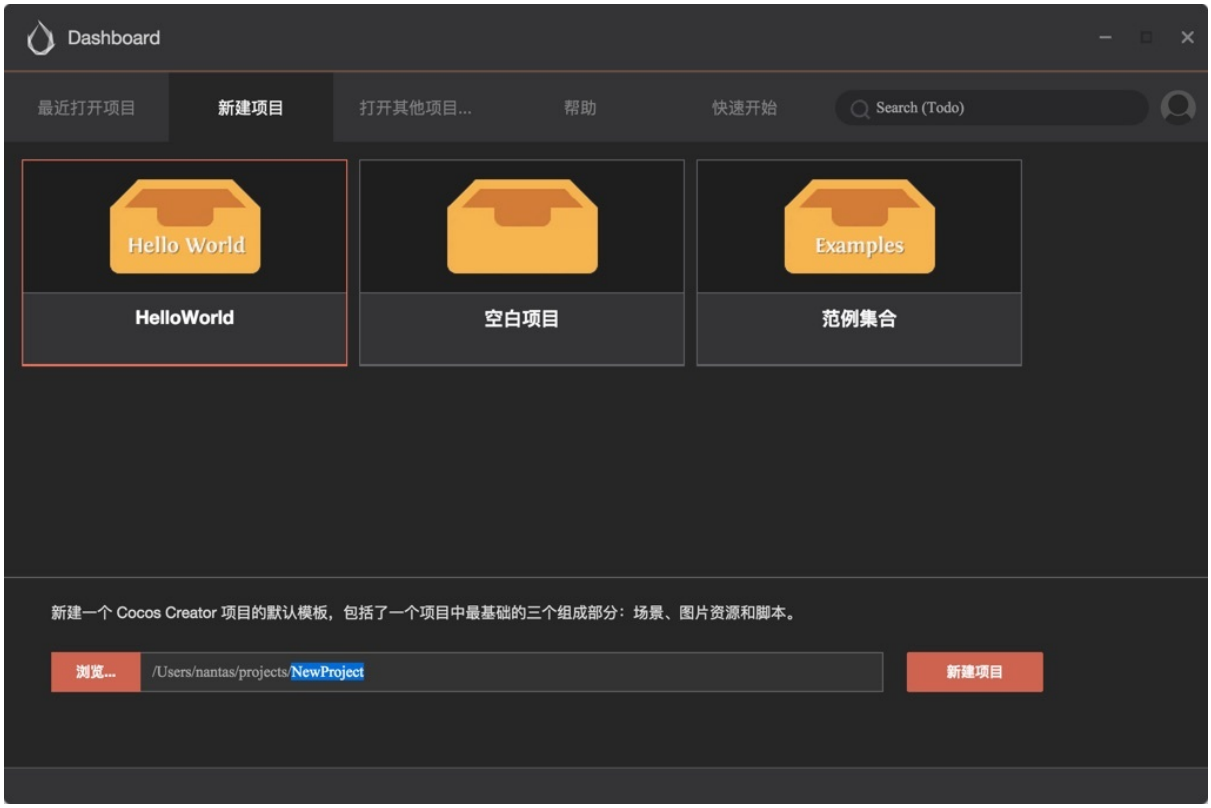

## 新建项目

你可以在 **新建项目** 选项卡里创建新的 Cocos Creator 项目。

在 **新建项目** 页面，我们首先需要选择一个项目模板，项目模板会包括各种不同类型的游戏基本架构，以及学习用的范例资源和脚本，来帮助你更快进入到创造性的工作当中。

注意： 早期的 *Cocos Creator* 版本中还没有很多可选择的项目模板，我们会随着 *Cocos Creator* 功能逐渐完整持续添加更多模板为用户提供方便。

点击选择一个模板，你可以在页面下方看到该模板的描述。




在页面下方你可以看到项目名称和将会保存到的地址。你可以在项目路径输入框手动输入项目所在路径和项目名称，路径的最后一节就是项目名称。

你也可以点击 **浏览** 按钮，打开浏览路径对话框，在你的本地文件系统中选择一个位置来存放新建项目。

一切都设置好后，点击 **新建项目** 按钮来完成项目的创建。Dashboard 界面会被关闭，然后新创建的项目会在 Cocos Creator 编辑器主窗口中打开。

## 打开其他项目

如果你在 **最近打开项目** 页面找不到你的项目，或者刚刚从网上下载了一个从未打开过的项目时，你可以通过 **打开其他项目** 选项卡按钮在本地文件系统浏览并打开项目。

点击 **打开其他项目** 后，会弹出本地文件系统的选择对话框，在这个对话框中选中你的项目文件夹，并选择打开就可以打开项目。

注意：*Cocos Creator* 使用特定结构的文件夹来作为合法项目标识，而不是使用工程文件。选择项目时只要选中项目文件夹即可。

## 帮助

你可以通过 **帮助** 页面访问 Cocos Creator 用户手册和其他帮助文档。

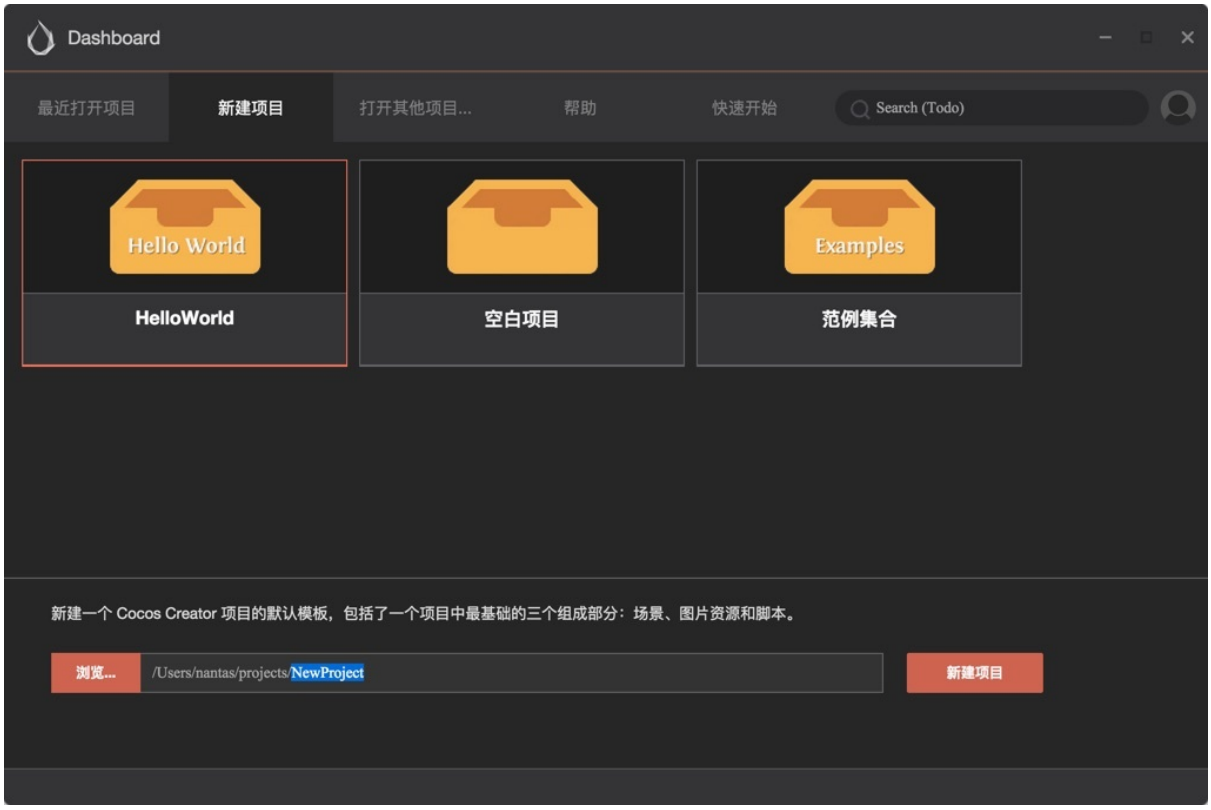

Dashboard
最近打开项目
新建项目
打开其他项目...
帮助
快速开始
Search (Todo)
COCOS CREATOR
Ver.0.7.0-rc8
欢迎使用 Cocos Creator，你可以采取下列行动来开始游戏开发工作：
打开最近编辑过的项目
使用项目模板创建新项目
使用文件系统浏览选择要打开的游戏项目
要学习使用 Cocos Creator，可以使用下面的学习资源：
教程：制作第一个游戏
观看快速开始视频
查看用户手册
查看 API 文档
教程和范例项目
范例集合
简单的动作游戏 Star Catcher
有完整菜单和复杂数据表现逻辑的21点游戏

# Hello World

了解 Dashboard 以后，我们现在看看如何创建和打开一个 Hello World 项目。

## 创建项目

在 Dashboard 中，打开 **新建项目** 选项卡，选中 `Hello World` 项目模板。

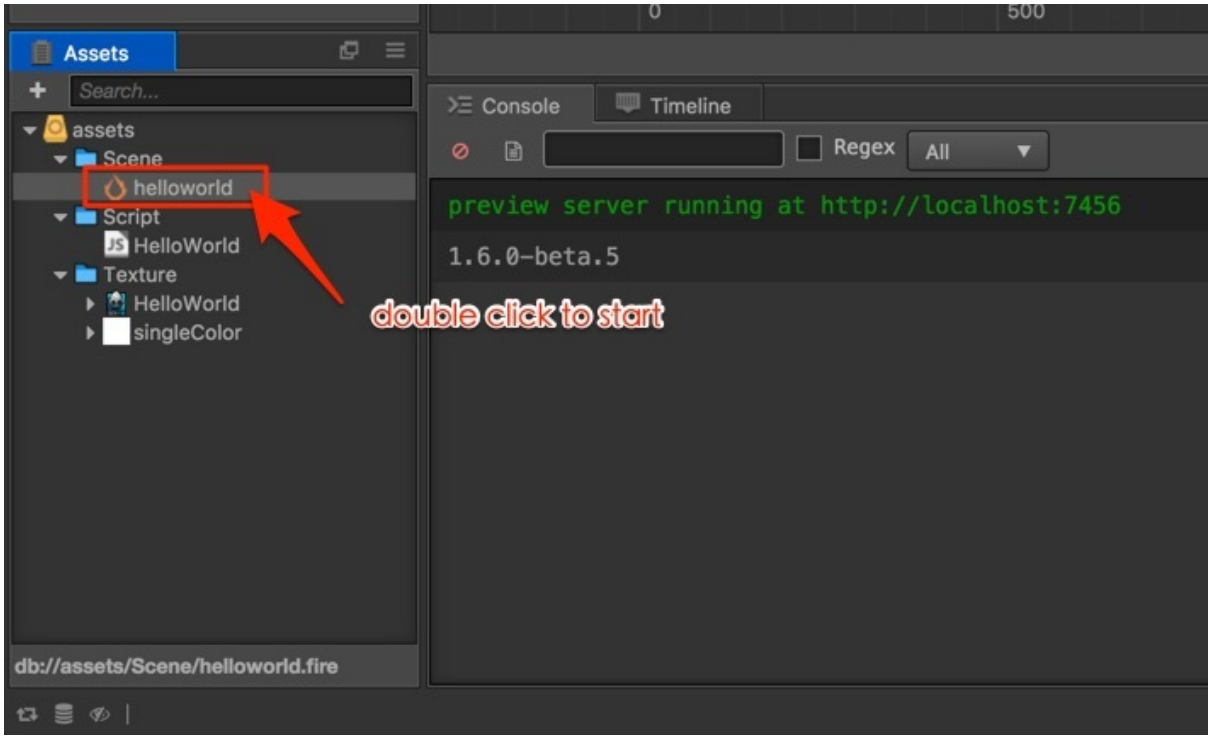



然后在下面的项目路径栏中指定一个新项目即将被创建的位置，路径的最后一部分就是项目文件夹。

填好路径后点击右下角的 **新建项目** 按钮，就会自动以 Hello World 项目模板创建项目并打开。

## 打开场景，开始工作

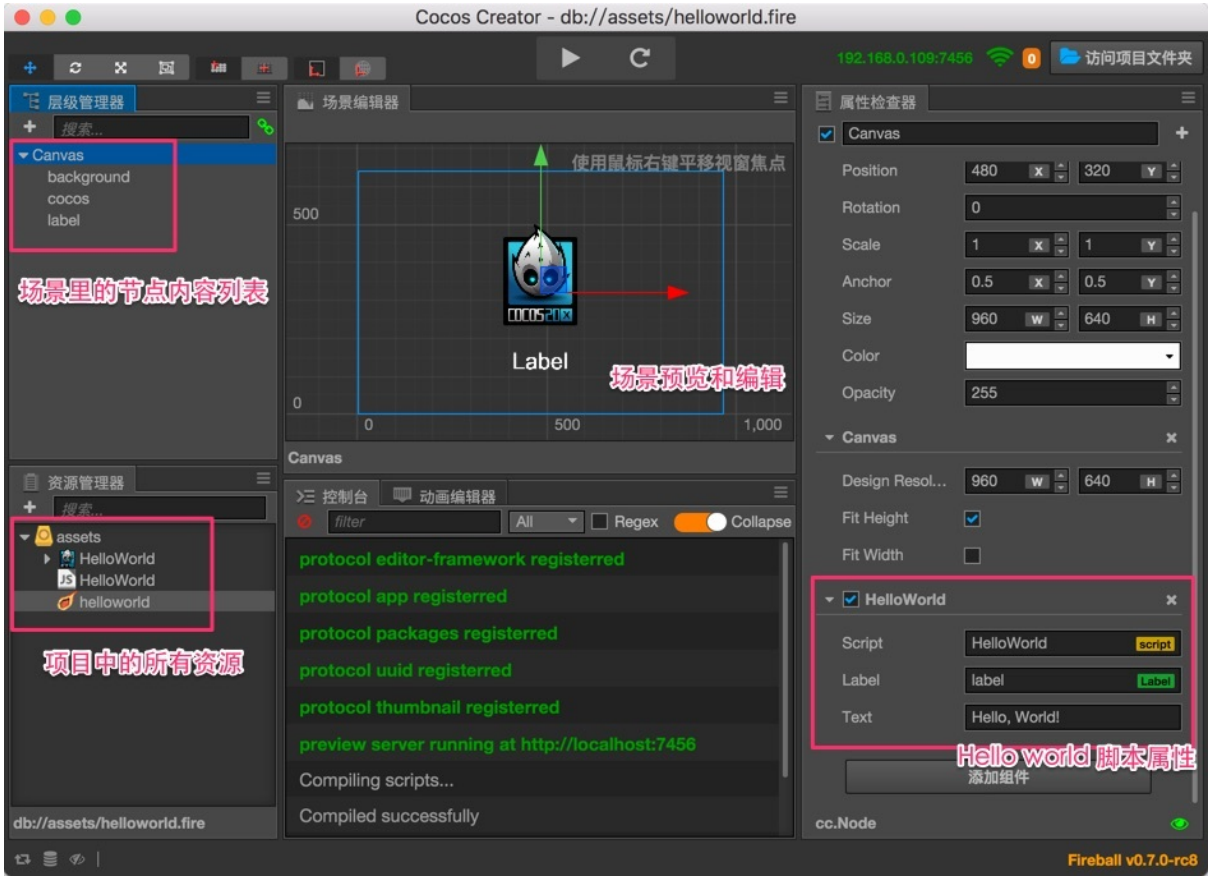
Cocos Creator 的工作流程是以数据驱动和场景为核心的，初次打开一个项目时，默认不会打开任何场景，要看到 Hello World 模板中的内容，我们需要先打开场景资源文件。




在 **资源管理器** 中双击箭头所指的 `helloworld` 场景文件。Cocos Creator 中所有场景文件都以  作为图标。

# Hello World 项目分解

打开 `helloworld` 场景后，我们就可以看到这个模板项目中的全部内容了。

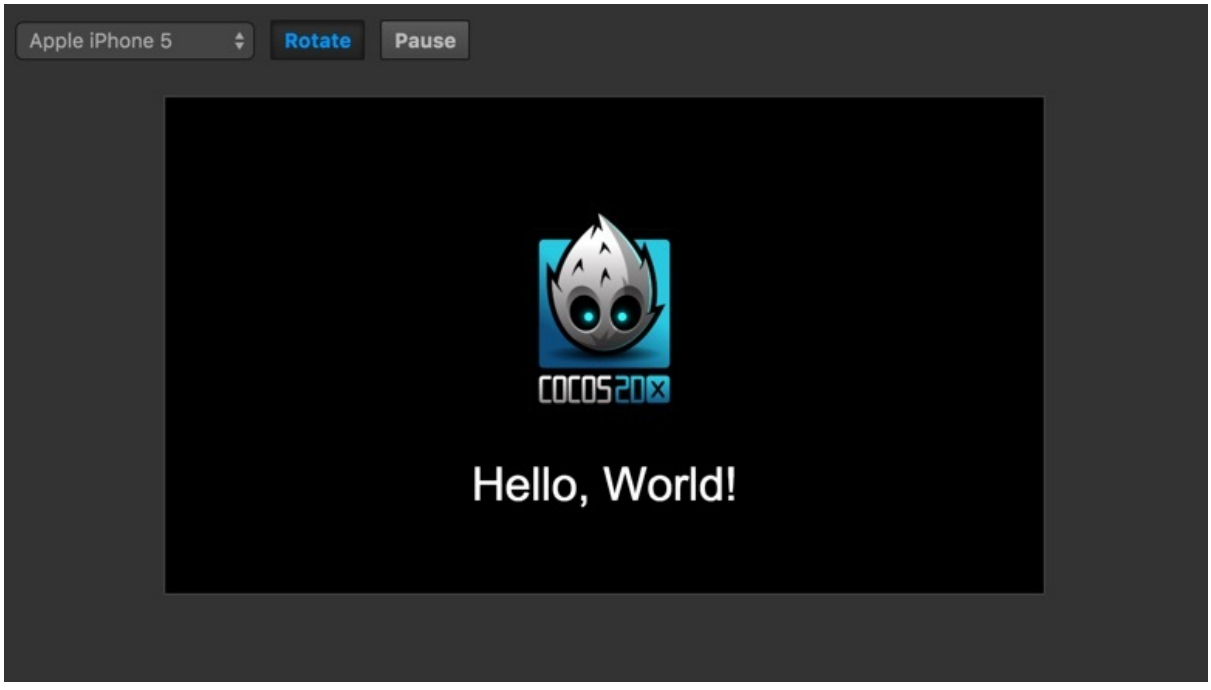


场景中的内容会按照工作流分别呈现在 **资源管理器**、**层级管理器**、**场景编辑器**、**属性检查器** 四个核心面板中，关于编辑器界面和主要面板的介绍我们会在后面的 编辑器界面介绍 部分详细介绍。

# 预览场景

要预览游戏场景，点击编辑器窗口正上方的 **预览游戏** 按钮。


Cocos Creator 会使用您的默认浏览器运行当前游戏场景，效果如图所示：



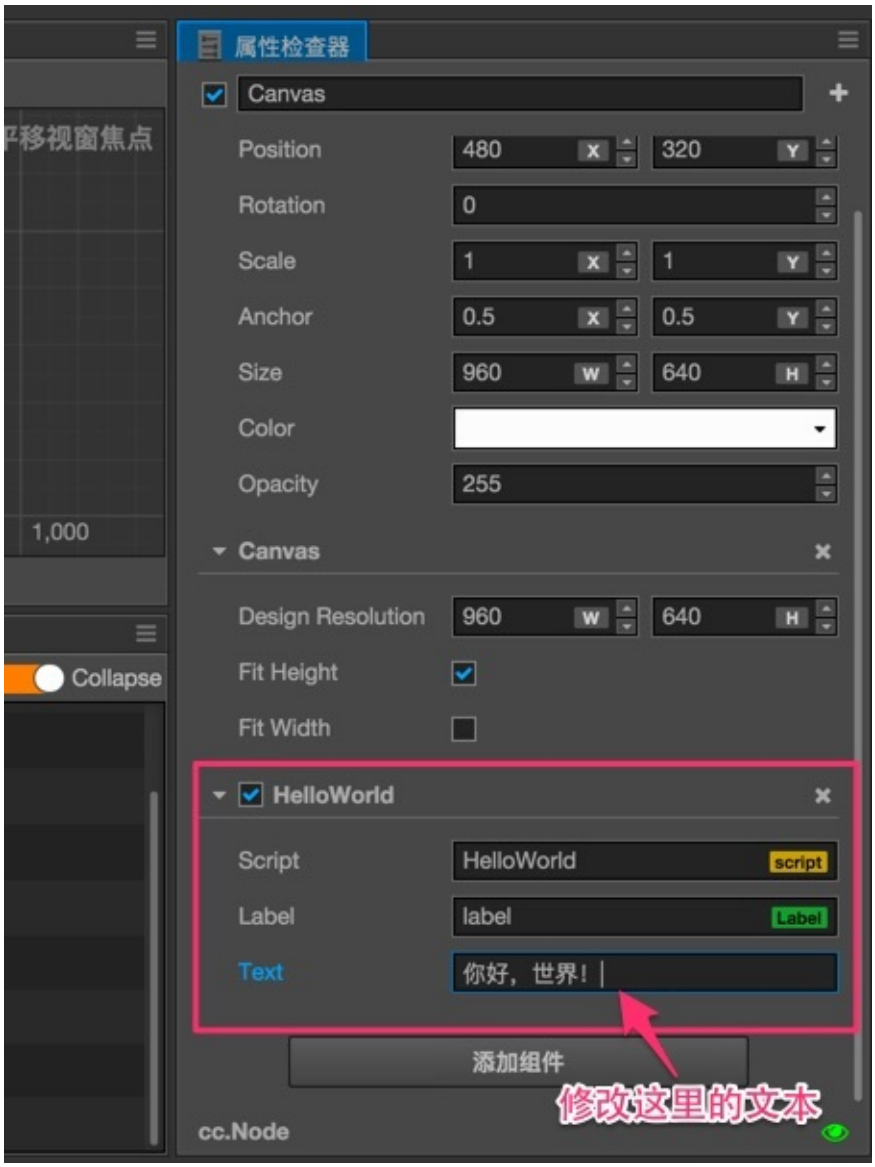

点击预览窗口左上角的下拉菜单，可以选择不同设备屏幕的预览效果。

# 修改欢迎文字

Cocos Creator 以数据驱动为核心的最初体现，就在于当我们需要改变 Hello World 的问候文字时，不需要再编辑脚本代码，而是直接修改场景中保存的文字属性。

首先在 **层级管理器** 中选中 `Canvas` 节点，我们的 `HelloWorld` 组件脚本就挂在这个节点上。

接下来在 **属性检查器** 面板下方找到 `HelloWorld` 组件属性，然后将 `Text` 属性里的文本改成 `你好，世界！` ：




再次运行预览，可以看到欢迎文字已经更新了：


## 小结

这一节的内容，让我们认识了如何从场景开始 Cocos Creator 的工作流程，并且通过修改欢迎文字小小展示了数据驱动的工作方式。接下去我们会用逐步讲解的方式引导大家完成一个较为完整的休闲游戏。相信之后您对 Cocos Creator 的工作流会有更完整的认识。

# 快速上手：制作第一个游戏

您正在阅读的手册文档包括了系统化的介绍 Cocos Creator 的编辑器界面、功能和工作流程，但如果您想快速上手体验使用 Cocos Creator 开发游戏的大体流程和方法，这一章将满足您的好奇心。完成本章教程之后，您应该能获得足够上手制作游戏的信息，不过我们还是推荐您继续阅读本手册来了解各个功能模块的细节和完整的工作流程。

接下来就让我们开始吧，跟随教程我们将会制作一款名叫**摘星星**的坑爹小游戏。这款游戏的玩家要操作一个反应**迟钝**却蹦跳不停的小怪物来碰触不断出现的星星，难以驾驭的加速度将给玩家带来很大挑战，和您的小伙伴比比谁能拿到更多星星吧！

可以在这里感受一下这款游戏的完成形态：

http://fbdemos.leanapp.cn/star-catcher/

## 准备项目和资源

我们已经为您准备好了制作这款游戏需要的全部资源，下载**初始项目**后，解压到您希望的位置，之后我们就可以开始了：

下载初始项目

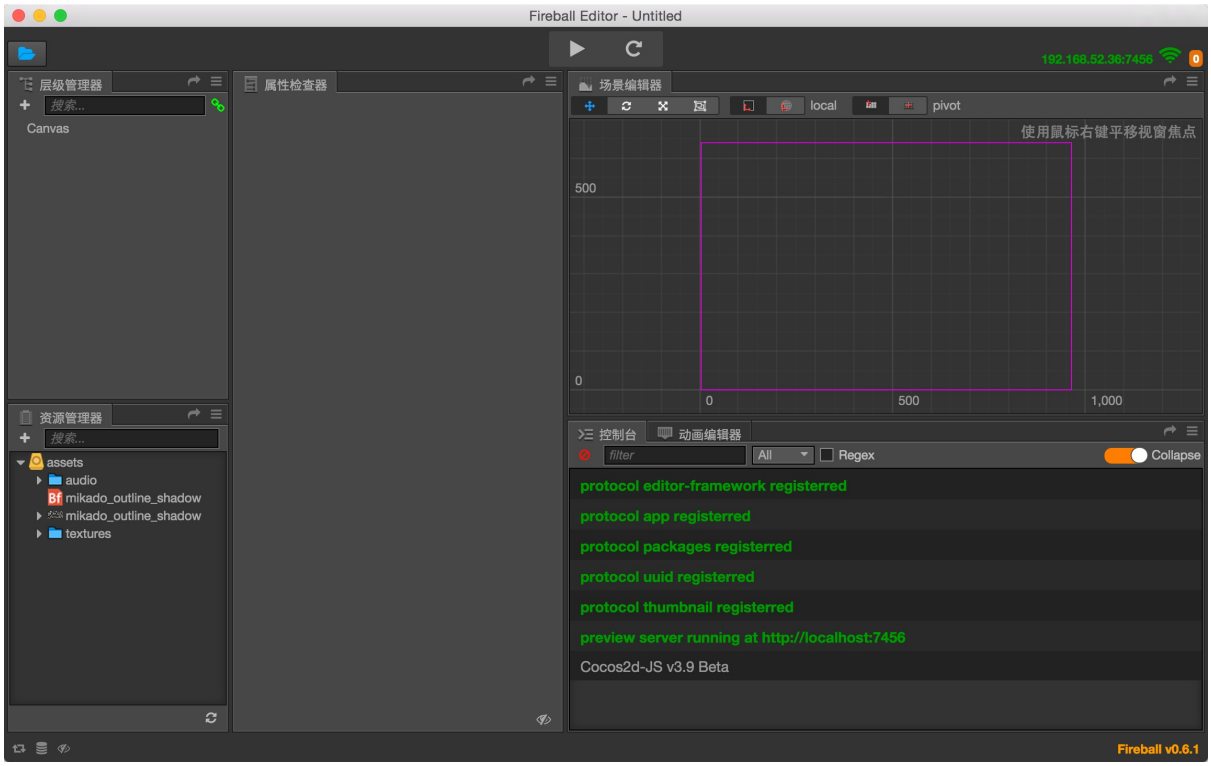
您也可以下载最终完成的项目，跟随教程制作过程中有任何不明白的地方都可以打开完成版的项目作为参考：

下载完成项目

## 打开初始项目

如果您还不了解如何获取和启动 Cocos Creator，请阅读安装和启动一节。

1. 我们首先启动 Cocos Creator，然后选择**打开其他项目**
2. 在弹出的文件夹选择对话框中，选中我们刚下载并解压完成的 `start_project` ，点击**打开**按钮
3. Cocos Creator 编辑器主窗口会打开，我们将看到如下的项目状态



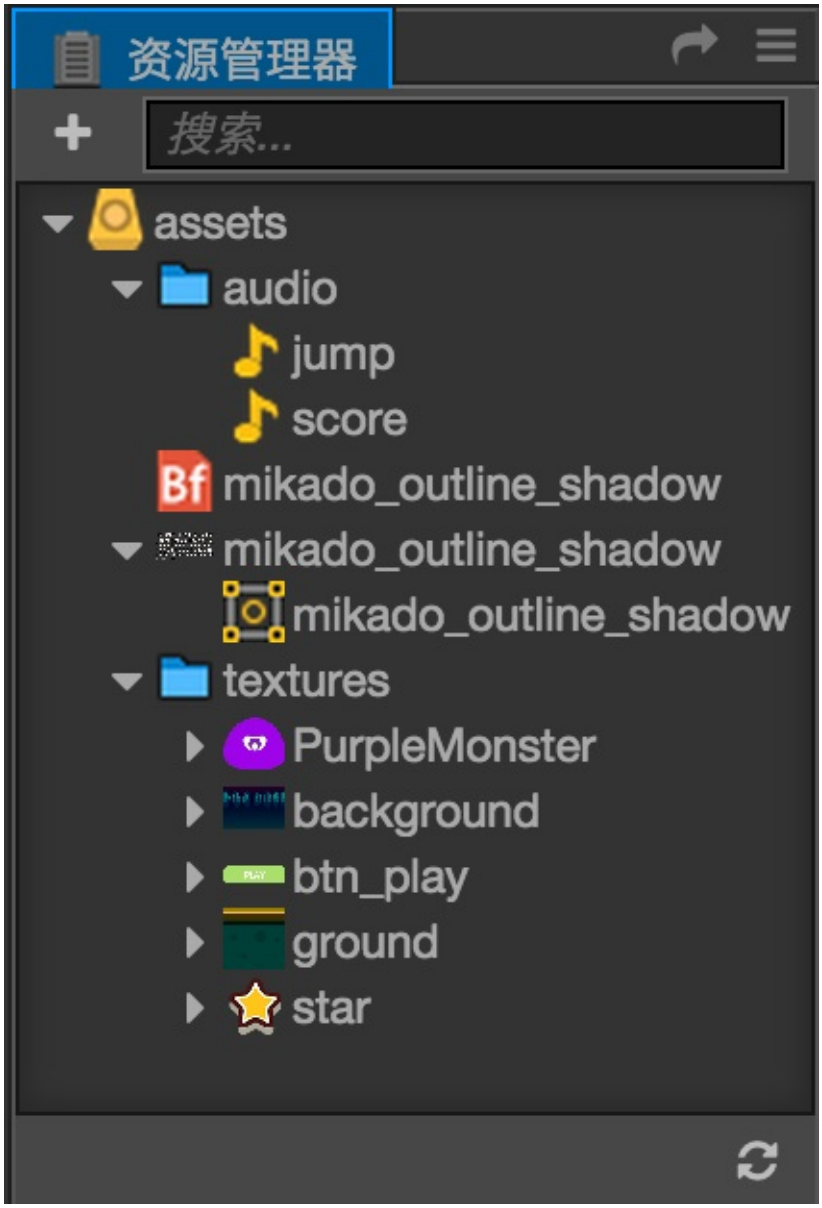

# 检查游戏资源

我们的初始项目中已经包含了所有需要的游戏资源，因此您不需要再导入任何其他资源。如果希望了解导入资源的方法，可以阅读资源工作流程的相关内容。

下面我们先来了解一下项目中都有哪些资源，请关注名为**资源管理器**的面板，这里显示的是项目中的所有资源树状结构。

可以看到，项目资源的根目录名叫**assets**，对应我们解压之后初始项目中的 `assets` 目录，只有这个目录下的资源才会被 Cocos Creator 导入项目并进行管理。

**资源管理器**可以显示任意层次的目录结构，我们可以看到  这样的图标就代表一个文件夹，点击文件夹左边的三角图标可以展开文件夹的内容。将文件夹全部展开后，**资源管理器**中就会呈现如下图的状态。




每个资源都是一个文件，导入项目后根据扩展名的不同而被识别为不同的资源类型，其图标也会有所区别，下面我们来看看项目中的资源各自的类型和作用：

- 声音文件，一般为 mp3 文件，我们将在主角跳跃和得分时播放名为 `jump` 和 `score` 的声音文件。
- 位图字体，由 fnt 文件和同名的 png 图片文件共同组成。位图字体（Bitmap Font）是一种游戏开发中常用的字体资源，详情请阅读位图字体资源
- 各式各样的缩略图标，这些都是图像资源，一般是 png 或 jpg 文件。图片文件导入项目后会经过简单的处理成为**texture**类型的资源。之后就可以将这些资源拖拽到场景或组件属性中去使用了。

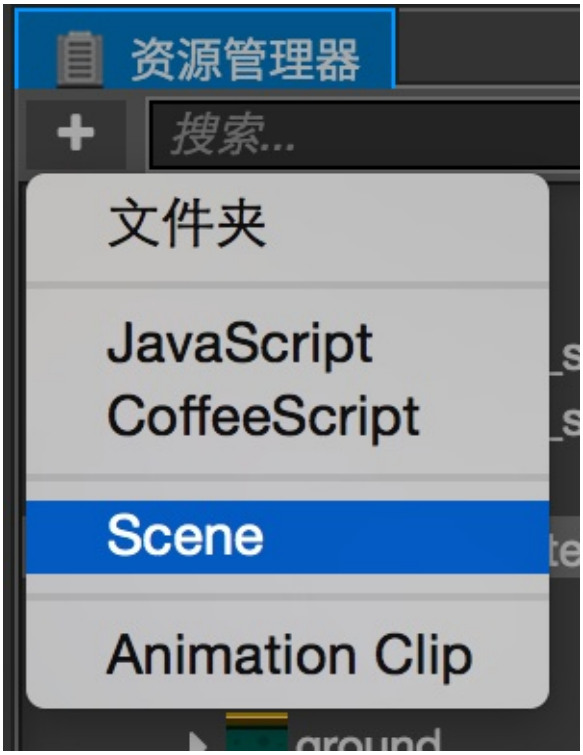
## 创建游戏场景

在 Cocos Creator 中，游戏场景（Scene）是开发时组织游戏内容的中心，也是呈现给玩家所有游戏内容的载体。游戏场景中一般会包括以下内容：

- 场景图像和文字（Sprite，Label）
- 角色
- 以组件形式附加在场景节点上的游戏逻辑脚本

当玩家运行游戏时，就会载入游戏场景，游戏场景加载后就会自动运行所包含组件的游戏脚本，实现各种各样开发者设置的逻辑功能。所以除了资源以外，游戏场景是一切内容创作的基础，让我们现在就新建一个场景。

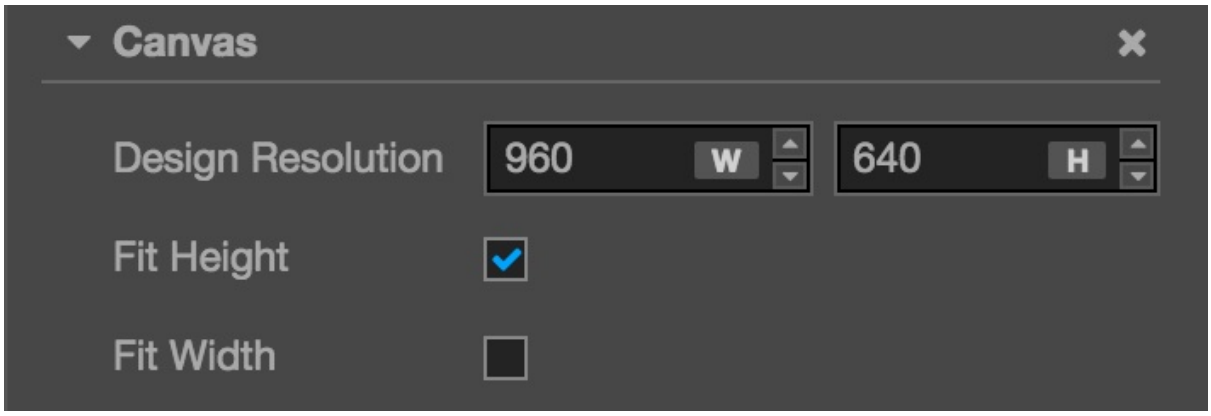
1. 在资源管理器中点击选中**assets**目录，确保我们的场景会被创建在这个目录下
2. 点击资源管理器左上角的加号按钮，在弹出的菜单中选择 **Scene**




3. 我们创建了一个名叫 `New Scene` 的场景文件，右键点击它并选择**重命名**，将它改名为 `game` 。
4. 双击 `game` ，就会在**场景编辑器**和**层级编辑器**中打开这个场景。

## 了解Canvas

打开场景后，**层级管理器**中会显示当前场景中的所有节点和他们的层级关系。我们刚刚新建的场景中只有一个名叫 `Canvas` 的节点，**Canvas**可以被称为画布节点或渲染根节点，点击选中**Canvas**，可以在**属性检查器**中看到他的属性。




这里的 `Design Resolution` 属性规定了游戏的设计分辨率， `Fit Height` 和 `Fit Width` 规定了在不同尺寸的屏幕上运行时，我们将如何缩放**Canvas**以适配不同的分辨率。

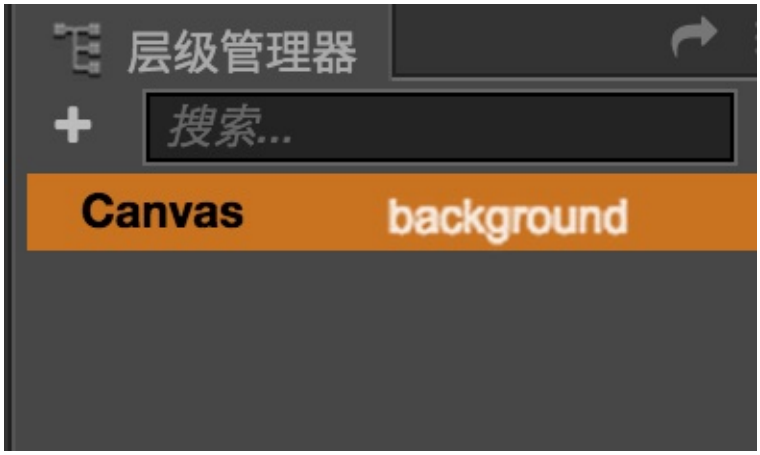
由于提供了多分辨率适配的功能，我们一般会将场景中的所有负责图像显示的节点都放在**Canvas**下面。这样当**Canvas**的 `scale` （缩放）属性改变时，所有作为其子节点的图像也会跟着一起缩放以适应不同屏幕的大小。

更详细的信息请阅读Canvas组件参考。目前我们只要知道接下来添加的场景图像都会放在**Canvas**节点下面就可以了。

# 设置场景图像

## 添加背景

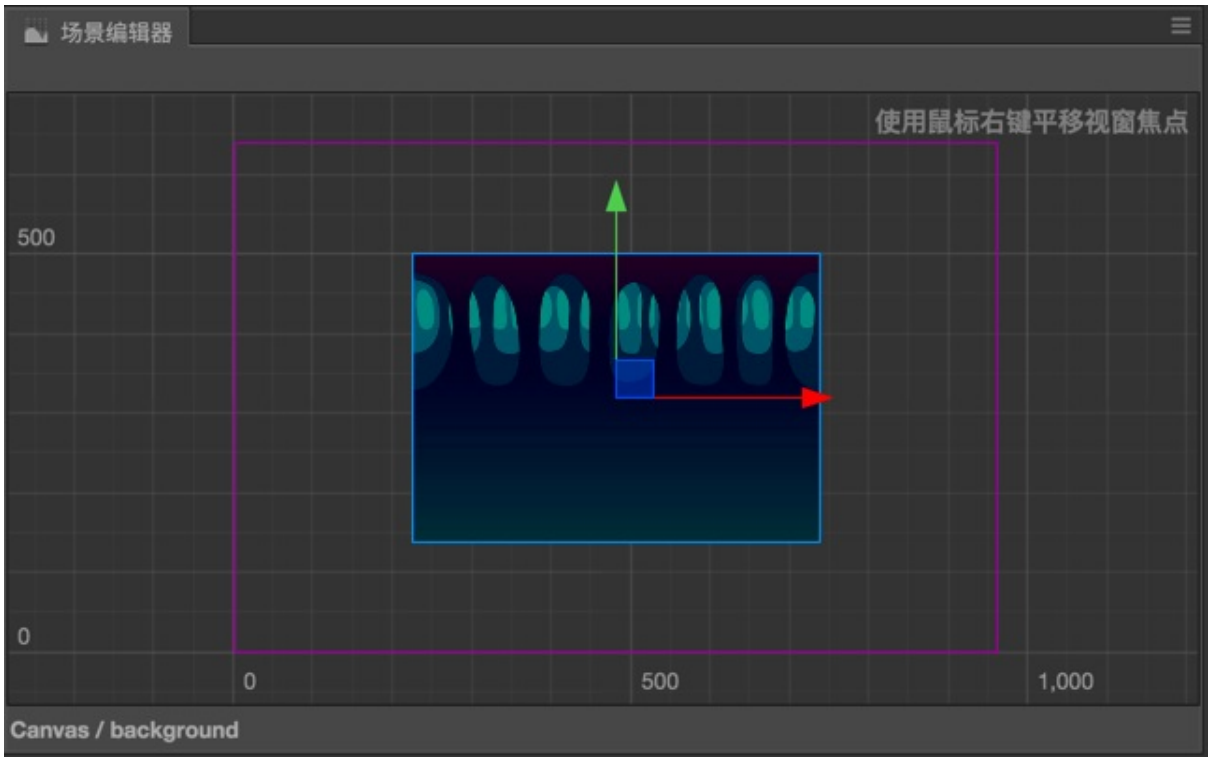
首先在**资源管理器**里按照 `assets/textures/background` 的路径找到我们的背景图像资源，点击并拖拽这个资源到**层级编辑器**中的**Canvas**节点上，直到**Canvas**节点显示橙色高亮，表示将会添加一个以 `background` 为贴图资源的子节点。




这时就可以松开鼠标按键，可以看到**Canvas**下面添加了一个名叫 `background` 的节点。当我们使用拖拽资源的方式添加节点时，节点会自动以贴图资源的文件名来命名。

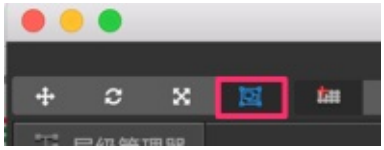
我们在对场景进行编辑修改时，可以通过主菜单 `文件->保存场景` 来及时保存我们的修改。也可以使用快捷键`Ctrl+S`（Windows）或`Cmd + S`（Mac）来保存。

## 修改背景尺寸

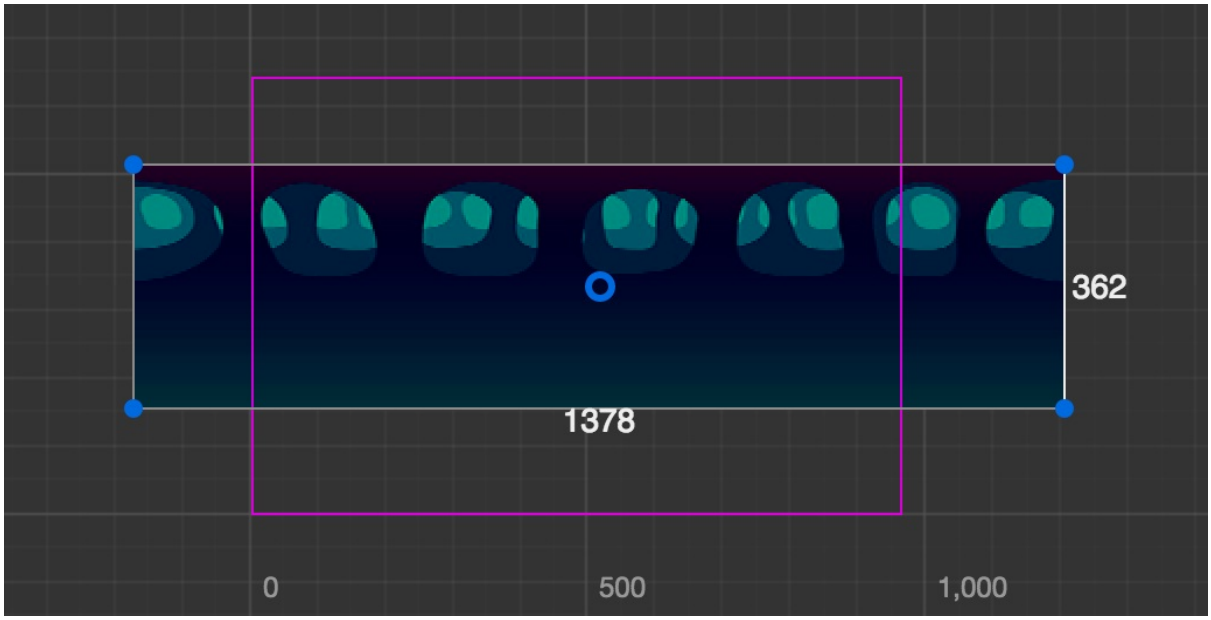


在 **场景编辑器** 中，可以看到我们刚刚添加的背景图像，下面我们将修改背景图像的尺寸，来让他覆盖整个屏幕。

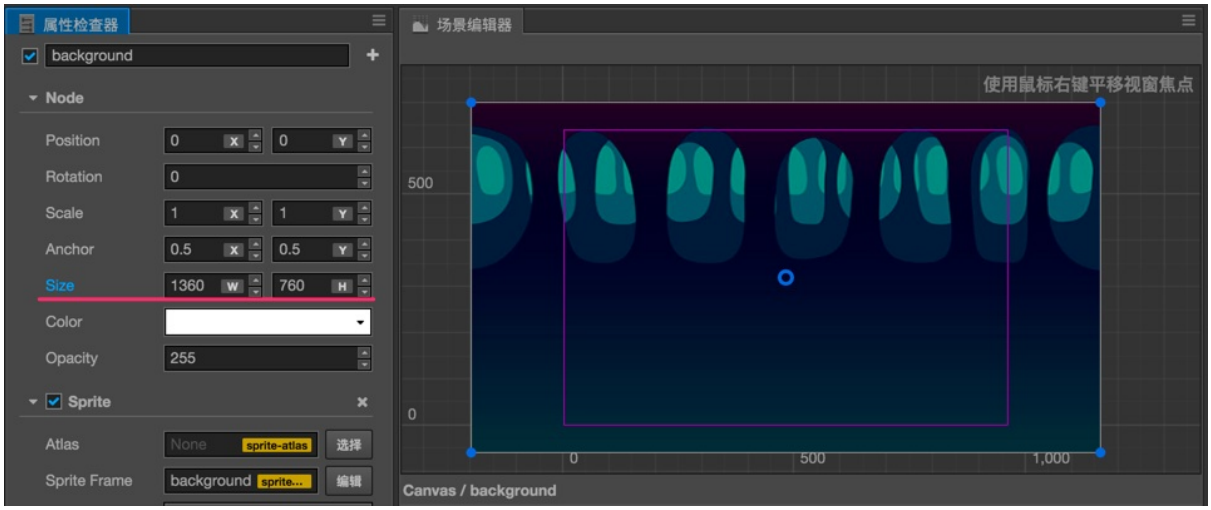
首先选中 `background` 节点，然后点击主窗口左上角工具栏第四个 **矩形变换工具**：



使用这个工具我们可以方便的修改图像节点的尺寸，将鼠标移动到 **场景编辑器** 中 `background` 的左边，按住并向左拖拽直到 `background` 的左边超出表示设计分辨率的蓝色线框。然后再用同样的方法将 `background` 的右边向右拖拽。




之后需要拖拽上下两边，使背景图的大小能够填满设计分辨率的线框。



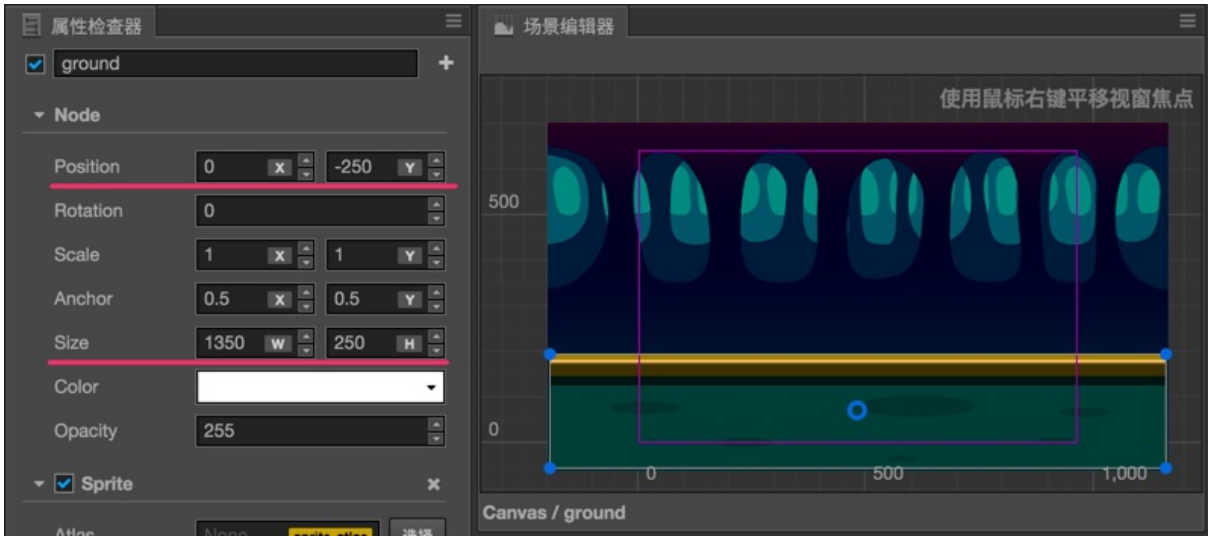

在使用 **矩形变换工具** 修改背景图尺寸时，可以在 **属性检查器** 中看到 **Node** （节点）中的 `Size` 属性也在随之变化，完成后我们的背景图尺寸大概是 `(1360, 760)` 。您也可以直接在 `Size` 属性的输入框中输入数值，和使用 **矩形变换工具** 可以达到同样效果。这样大小的背景图在市面流行的手机上都可以覆盖整个屏幕，不会出现穿帮情况。

## 添加地面

我们的主角需要一个可以在上面跳跃的地面，我们马上来添加一个。用和添加背景图相同的方式，拖拽 **资源管理器** 中 `assets/textures/ground` 资源到 **层级管理器** 的 `Canvas` 上。这次在拖拽时我们还可以选择新添加的节点和 `background` 节点的顺序关系。拖拽资源的状态下移动鼠标指针到 `background` 节点的下方，直到在 `Canvas` 上显示橙色高亮框，并同时在 `background` 下方显示表示插入位置的绿色线条，然后松开鼠标，这样 `ground` 在场景层级中就被放在了 `background` 下方，同时也是 `Canvas` 的一个子节点。

在**层级管理器**中，显示在下方的节点的渲染顺序会在上方节点的后面，我们可以看到位于最下的 `ground` 物体在**场景编辑器**中显示在了最前。另外子节点也会永远显示在父节点之前，我们可以随时调整节点的层级顺序和关系来控制他们的显示顺序。

按照修改背景的方法，我们也可以使用**矩形变换工具**来为地面节点设置一个合适的大小。在激活**矩形变换工具**的时候，如果拖拽节点顶点和四边之外的部分，就可以更改节点的位置。下图是我们设置好的地面节点状态：




除了**矩形变换工具**之外，我们还可以使用**移动工具**  来改变节点的位置。尝试按住**移动工具**显示在节点上的箭头并拖拽，就可以一次改变节点在单个坐标轴上的位置。

我们在设置背景和地面的位置和尺寸时不需要很精确的数值，可以凭感觉拖拽。如果您偏好比较完整的数字，也可以按照截图直接输入 `position` 和 `size` 的数值。

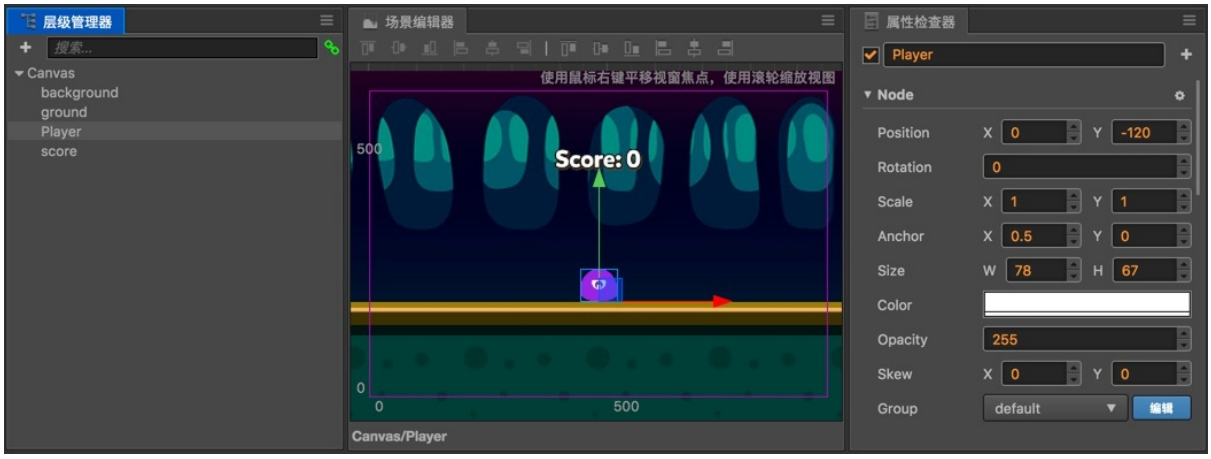
## 添加主角

接下来我们的主角小怪兽要登场了，从**资源管理器**拖拽 `assets/texture/PurpleMonster` 到**层级管理器**中 `Canvas` 的下面，并确保他的排序在 `ground` 之下，这样我们的主角会显示在最前面。 注意小怪兽节点应该是 `Canvas` 的子节点，和 `ground` 节点平行。

为了让主角的光环在场景节点中非常醒目，我们右键点击刚刚添加的 `PurpleMonster` 节点，选择 `重命名` 之后将其改名为 `Player` 。

接下来我们要对主角的属性进行一些设置，首先是改变**锚点(Anchor)**的位置。默认状态下，任何节点的锚点都会在节点的中心，也就是说该节点中心点所在的位置就是该节点的位置。我们希望控制主角的底部的位置来模拟在地面上跳跃的效果，所以现在我们需要把主角的锚点设置在脚下。在**属性检查器**里找到**Anchor**属性，把其中的 `y` 值设为 `0` ，可以看到**场景编辑器**中，表示主角位置的**移动工具**的箭头出现在了主角脚下。

接下来**场景编辑器**中拖拽 `Player` ，把他放在地面上，效果如下图：




这样我们基本的场景美术内容就配置好了。下面一节我们要编写代码让游戏里的内容生动起来。

# 编写主角脚本

Cocos Creator 开发游戏的一个核心理念就是让内容生产和功能开发可以流畅的并行协作，我们在上个部分着重于处理美术内容，而接下来就是通过编写脚本来开发功能的流程，之后我们还会看到写好的程序脚本可以很容易的被内容生产者使用。

如果您从没写过程序也不用担心，我们会在教程中提供所有需要的代码，只要复制粘贴到正确的位置就可以了，之后这部分工作可以找您的程序员小伙伴来解决。下面让我们开始创建驱动主角行动的脚本吧。

## 创建脚本

1. 首先在**资源管理器**中右键点击 `assets` 文件夹，选择 `新建->文件夹`


2. 右键点击 `New Folder`，选择 `重命名`，将其改名为 `scripts`，之后我们所有的脚本都会存放在这里。
3. 右键点击 `scripts` 文件夹，选择 `新建->JavaScript`，创建一个JavaScript脚本
4. 将新建脚本的名字改为 `Player`。双击这个脚本，打开代码编辑器。

**注意：** Cocos Creator 中脚本名称就是组件的名称，这个命名是大小写敏感的！如果组件名称的大小写不正确，将无法正确通过名称使用组件！

## 编写组件属性

打开的脚本里已经有了预先设置好的一些代码块，这些代码就是编写一个组件脚本所需的结构。具有这样结构的脚本就是 Cocos Creator 中的组件（Component），他们能够挂载到场景中的节点上，提供控制节点的各种功能。我们先来设置一些属性，然后看看怎样在场景中调整他们。

在**代码编辑器**中找到 `Player` 脚本里 `properties` 部分，将其改为以下内容并按`Ctrl + S`（Windows）或`Cmd + S`（Mac）保存：

```
// Player.js
    //...
    properties: {
        // 主角跳跃高度
        jumpHeight: 0,
        // 主角跳跃持续时间
        jumpDuration: 0,
        // 最大移动速度
        maxMoveSpeed: 0,
        // 加速度
        accel: 0,
    },
    //...
```

这些新增的属性将规定主角的移动方式，在代码中我们不需要关心这些数值是多少，因为我们之后会直接在**属性检查器**中设置这些数值。

现在我们可以把 `Player` 组件添加到主角节点上。在 **层级编辑器** 中选中 `Player` 节点，然后在 **属性检查器** 中点击 **添加组件** 按钮，选择 `添加用户脚本组件->Player`，为主角节点添加 `Player` 组件。


现在我们可以在 **属性检查器** 中（需要选中 `Player` 节点）看到刚添加的 `Player` 组件了，按照下图将主角跳跃和移动的相关属性设置好：


这些数值除了 jumpDuration 的单位是秒之外，其他的数值都是以像素为单位的，根据我们现在对 `Player` 组件的设置：我们的主角将能够跳跃 200 像素的高度，起跳到最高点所需的时间是 0.3 秒，最大水平方向移动速度是 400 像素每秒，水平加速度是 350 像素每秒。

这些数值都是建议，一会等游戏运行起来，您完全可以按照自己的喜好随时在**属性检查器**中修改这些数值，不需要改动任何代码，很方便吧！

## 编写跳跃和移动代码

下面我们添加一个方法，来让主角跳跃起来，在 `properties: {...},` 代码块的下面，添加叫做 `setJumpAction` 的方法：

```
// Player.js
    properties: {
        //...
    },

    setJumpAction: function () {
        // 跳跃上升
        var jumpUp = cc.moveBy(this.jumpDuration, cc.p(0, this.jumpHeight)).easing(cc.easeCubicActionOut());
        // 下落
        var jumpDown = cc.moveBy(this.jumpDuration, cc.p(0, -this.jumpHeight)).easing(cc.easeCubicActionIn());
        // 不断重复
        return cc.repeatForever(cc.sequence(jumpUp, jumpDown));
    },
```

这里用到了一些 Cocos2d-js 引擎中的 Action 来实现主角的跳跃动画，详情可以查询Cocos2d-js API.

接下来在 `onLoad` 方法里调用刚添加的 `setJumpAction` 方法，然后执行 `runAction` 来开始动作：

```
// Player.js
    onLoad: function () {
        // 初始化跳跃动作
        this.jumpAction = this.setJumpAction();
        this.node.runAction(this.jumpAction);
    },
```

`onLoad` 方法会在场景加载后立刻执行，所以我们会把初始化相关的操作和逻辑都放在这里面。

保存脚本，然后我们就可以开始第一次运行游戏了！

点击 Cocos Creator 编辑器上方正中的**预览游戏**按钮

，Cocos Creator 会自动打开您的默认浏览器并开始在里面运行游戏，现在应该可以看到我们的主角——紫色小怪兽在场景中间活泼的蹦个不停了。

## 移动控制

只能在原地傻蹦的主角可没前途，让我们为主角添加键盘输入，用A和D来控制他的跳跃方向。在 `setJumpAction` 方法的下面添加 `setInputControl` 方法：

```
// Player.js
    setJumpAction: function () {
        //...
    },

    setInputControl: function () {
        var self = this;
        // 添加键盘事件监听
        cc.eventManager.addListener({
            event: cc.EventListener.KEYBOARD,
            // 有按键按下时，判断是否是我们指定的方向控制键，并设置向对应方向加速
            onKeyPressed: function(keyCode, event) {
                switch(keyCode) {
                    case cc.KEY.a:
                        self.accLeft = true;
                        self.accRight = false;
                        break;
                    case cc.KEY.d:
                        self.accLeft = false;
                        self.accRight = true;
                        break;
                }
            },
            // 松开按键时，停止向该方向的加速
            onKeyReleased: function(keyCode, event) {
                switch(keyCode) {
                    case cc.KEY.a:
                        self.accLeft = false;
                        break;
                    case cc.KEY.d:
                        self.accRight = false;
                        break;
                }
            }
        }, self.node);
    },
```

然后修改 `onLoad` 方法，在其中加入向左和向右加速的开关，以及主角当前在水平方向的速度，最后再调用我们刚添加的 `setInputControl` 方法，在场景加载后就开始监听键盘输入：

```
// Player.js
    onLoad: function () {
        // 初始化跳跃动作
        this.jumpAction = this.setJumpAction();
        this.node.runAction(this.jumpAction);

        // 加速度方向开关
        this.accLeft = false;
        this.accRight = false;
        // 主角当前水平方向速度
        this.xSpeed = 0;

        // 初始化键盘输入监听
        this.setInputControl();
    },
```

最后修改 `update` 方法的内容，添加加速度、速度和主角当前位置的设置：

```
// Player.js
```

```
    update: function (dt) {
        // 根据当前加速度方向每帧更新速度
        if (this.accLeft) {
            this.xSpeed -= this.accel * dt;
        } else if (this.accRight) {
            this.xSpeed += this.accel * dt;
        }
        // 限制主角的速度不能超过最大值
        if ( Math.abs(this.xSpeed) > this.maxMoveSpeed ) {
            // if speed reach limit, use max speed with current direction
            this.xSpeed = this.maxMoveSpeed * this.xSpeed / Math.abs(this.xSpeed);
        }

        // 根据当前速度更新主角的位置
        this.node.x += this.xSpeed * dt;
    },
```

`update` 在场景加载后就会每帧调用一次，我们一般把需要经常计算或及时更新的逻辑内容放在这里。在我们的游戏中，根据键盘输入获得加速度方向后，就需要每帧在 `update` 中计算主角的速度和位置。

保存脚本后，下面就可以去泡杯茶，点击**预览游戏**来看看我们最新的成果。在浏览器打开预览后，用鼠标点击一下游戏画面（这是浏览器的限制，要点击游戏画面才能接受键盘输入），然后就可以按A和D键来控制主角左右移动了！

感觉移动起来有点迟缓？主角跳的不够高？希望跳跃时间长一些？没问题，这些都可以随时调整。只要为 `Player` 组件设置不同的属性值，就可以按照您的想法调整游戏。这里有一组设置可供参考：

```
Jump Height: 150
Jump Duration: 0.3
Max Move Speed: 400
Accel: 1000
```

这组属性设置会让主角变得灵活无比，至于如何选择，就看您想做一个什么风格的游戏了。

# 制作星星

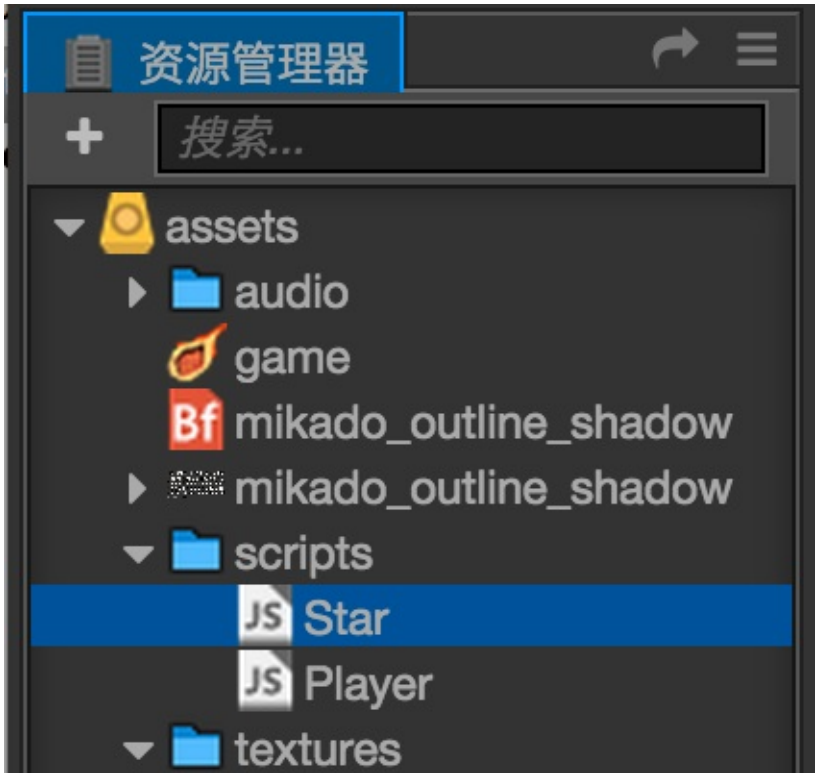
主角现在可以跳来跳去了，我们要给玩家一个目标，也就是会不断出现在场景中的星星，玩家需要引导小怪兽碰触星星来收集分数。被主角碰到的星星会消失，然后马上在随机位置重新生成一个。

## 制作Prefab

对于需要重复生成的节点，我们可以将他保存成 **Prefab（预制）** 资源，作为我们动态生成节点时使用的模板。关于 **Prefab** 的更多信息，请阅读 预制资源（Prefab） 。

首先从 **资源管理器** 中拖拽 `assets/textures/star` 资源到场景中，位置随意，我们只是需要借助场景作为我们制作 Prefab 的工作台，制作完成后会我们把这个节点从场景中删除。

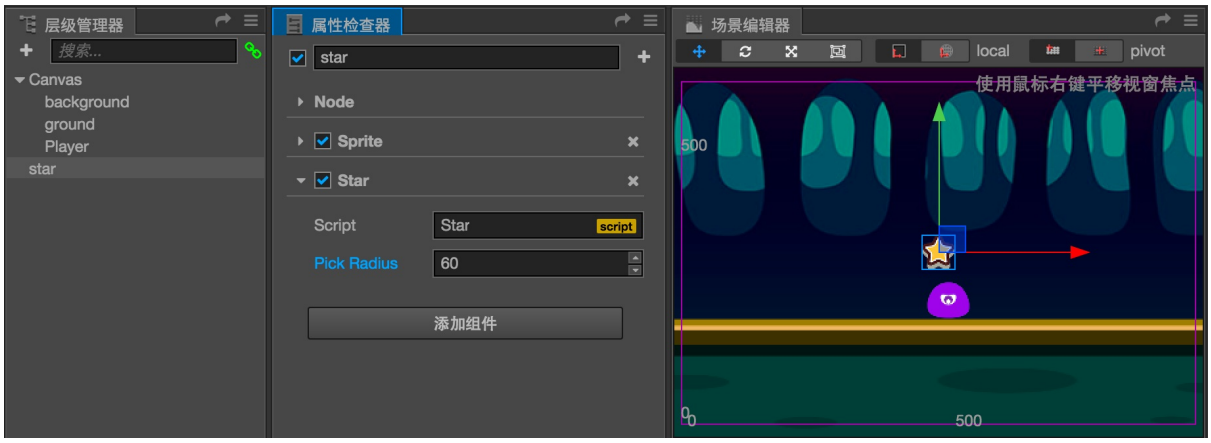
我们不需要修改星星的位置或渲染属性，但要让星星能够被主角碰触后消失，我们需要为星星也添加一个专门的组件。按照和添加 `Player` 脚本相同的方法，添加名叫 `Star` 的JavaScript脚本到 `assets/scripts/` 中。



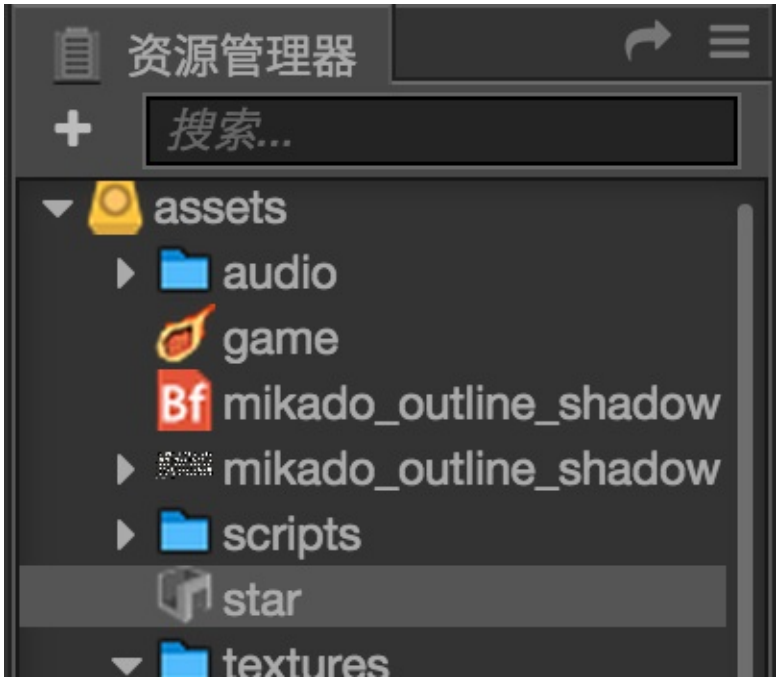

接下来双击这个脚本开始编辑，星星组件只需要一个属性用来规定主角距离星星多近时就可以完成收集，修改 `properties`，加入以下内容：

```
// Star.js
    properties: {
        // 星星和主角之间的距离小于这个数值时，就会完成收集
        pickRadius: 0
    },
```

保存脚本后，将这个脚本添加到刚创建的 `star` 节点上。然后在**属性检查器**中把 `Pick Radius` 属性值设为 `60`：




Star Prefab 需要的设置就完成了，现在从**层级管理器**中将 `star` 节点拖拽到**资源管理器**中的 `assets` 文件夹下，就生成了名叫 `star` 的 Prefab 资源。




现在可以从场景中删除 `star` 节点了，我们会在脚本中动态使用星星的 Prefab 资源生成星星。

## 添加游戏控制脚本

星星的生成是游戏主逻辑的一部分，所以我们要添加一个叫做 `Game` 的脚本作为游戏主逻辑脚本，这个脚本之后还会添加计分、游戏失败和重新开始的相关逻辑。

添加 `Game` 脚本到 `assets/scripts` 文件夹下，双击打开脚本。首先添加生成星星需要的属性：

```
// Game.js
    properties: {
        // 这个属性引用了星星预制资源
        starPrefab: {
            default: null,
            type: cc.Prefab
        },
        // 星星产生后消失时间的随机范围
        maxStarDuration: 0,
        minStarDuration: 0,
        // 地面节点，用于确定星星生成的高度
        ground: {
            default: null,
            type: cc.Node
        },
        // player 节点，用于获取主角弹跳的高度，和控制主角行动开关
        player: {
            default: null,
            type: cc.Node
        }
    },
```

保存脚本后将 `Game` 组件添加到**层级编辑器**中的 `Canvas` 节点上（选中 `Canvas` 节点后，拖拽脚本到 **属性检查器** 上，或点击 **属性检查器** 的 **添加组件** 按钮，并从 `用户自定义脚本` 中选择 `Game`，接下来从**资源管理器**中拖拽 `star` Prefab 资源到 `Game` 组件的 `Star Prefab` 属性中。这是我们第一次为属性设置引用，只有在属性声明时规定 `type` 为引用类型时（比如我们这里写的 `cc.Prefab` 类型），才能够将资源或节点拖拽到该属性上。

接下来从**层级编辑器**中拖拽 `ground` 和 `Player` 节点到组件中相同名字的属性上，完成节点引用。

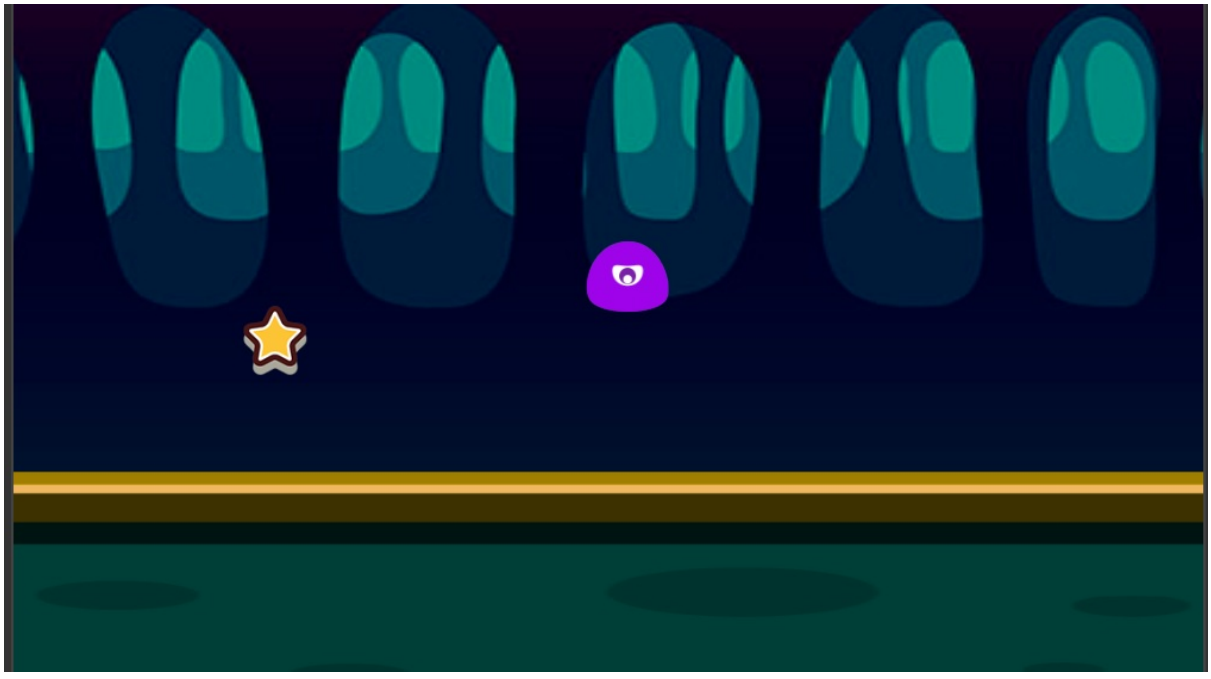
然后设置 `Min Star Duration` 和 `Max Star Duration` 属性的值为 `3` 和 `5`，之后我们生成星星时，会在这两个之间随机取值，就是星星消失前经过的时间。

## 在随机位置生成星星

接下来我们继续修改 `Game` 脚本，在 `onLoad` 方法后面添加生成星星的逻辑：

```
// Game.js
    onLoad: function () {
        // 获取地平面的 y 轴坐标
        this.groundY = this.ground.y + this.ground.height/2;
        // 生成一个新的星星
        this.spawnNewStar();
    },

    spawnNewStar: function() {
        // 使用给定的模板在场景中生成一个新节点
        var newStar = cc.instantiate(this.starPrefab);
        // 将新增的节点添加到 Canvas 节点下面
        this.node.addChild(newStar);
        // 为星星设置一个随机位置
        newStar.setPosition(this.getNewStarPosition());
    },

    getNewStarPosition: function () {
        var randX = 0;
        // 根据地平面位置和主角跳跃高度，随机得到一个星星的 y 坐标
        var randY = this.groundY + cc.random0To1() * this.player.getComponent('Player').jumpHeight + 50;
        // 根据屏幕宽度，随机得到一个星星 x 坐标
        var maxX = this.node.width/2;
        randX = cc.randomMinus1To1() * maxX;
        // 返回星星坐标
        return cc.p(randX, randY);
    }
```

保存脚本以后点击**预览游戏**按钮，在浏览器中可以看到，游戏开始后动态生成了一颗星星！用同样的方法，您可以在游戏中动态生成任何预先设置好的以 Prefab 为模板的节点。



## 添加主角碰触收集星星的行为

现在要添加主角收集星星的行为逻辑了，这里的重点在于，星星要随时可以获得主角节点的位置，才能判断他们之间的距离是否小于可收集距离，如何获得主角节点的引用呢？别忘了我们前面做过的两件事：

1. `Game` 组件中有个名叫 `player` 的属性，保存了主角节点的引用。
2. 每个星星都是在 `Game` 脚本中动态生成的。

所以我们只要在 `Game` 脚本生成 `Star` 节点实例时，将 `Game` 组件的实例传入星星并保存起来就好了，之后我们可以随时通过 `game.player` 来访问到主角节点。让我们打开 `Game` 脚本，在 `spawnNewStar` 方法最后面添加这样一句：

```
// Game.js
    spawnNewStar: function() {
        // ...
        // 将 Game 组件的实例传入星星组件
        newStar.getComponent('Star').game = this;
    },
```

保存后打开 `Star` 脚本，现在我们可以利用 `Game` 组件中引用的 `player` 节点来判断距离了，在 `onLoad` 方法后面添加名为 `getPlayerDistance` 和 `onPicked` 的方法：

```
// Star.js
    getPlayerDistance: function () {
        // 根据 player 节点位置判断距离
        var playerPos = this.game.player.getPosition();
        // 根据两点位置计算两点之间距离
        var dist = cc.pDistance(this.node.position, playerPos);
        return dist;
    },

    onPicked: function() {
        // 当星星被收集时，调用 Game 脚本中的接口，生成一个新的星星
        this.game.spawnNewStar();
        // 然后销毁当前星星节点
        this.node.destroy();
    },
```

然后在 `update` 方法中添加每帧判断距离，如果距离小于 `pickRadius` 属性规定的收集距离，就执行收集行为：

```
// Star.js
    update: function (dt) {
        // 每帧判断和主角之间的距离是否小于收集距离
        if (this.getPlayerDistance() < this.pickRadius) {
            // 调用收集行为
            this.onPicked();
            return;
        }
    },
```

保存脚本，然后再次预览测试，可以看到控制主角靠近星星时，星星就会消失掉，然后在随机位置生成了新的星星！

# 添加得分

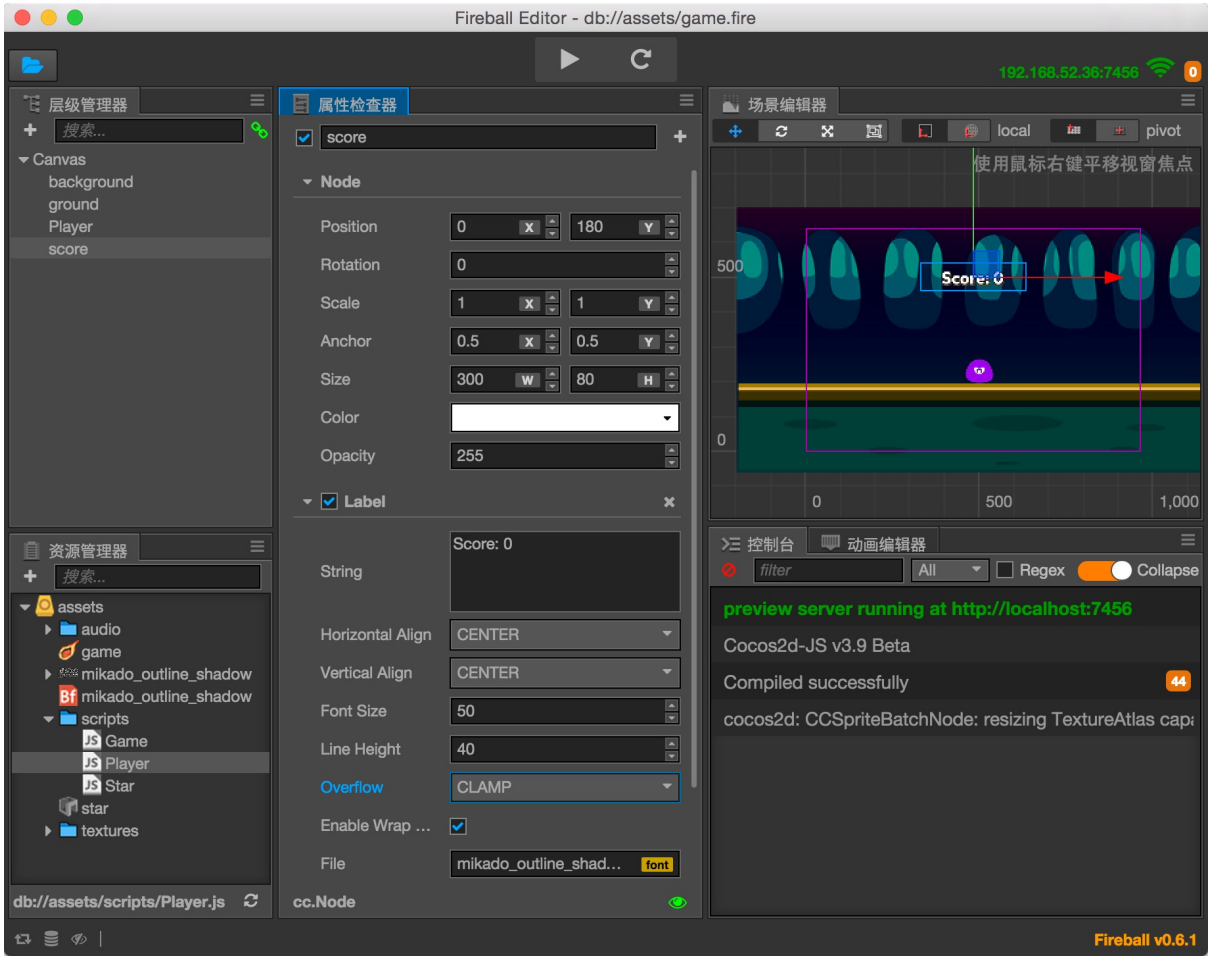
小怪兽辛辛苦苦的收集星星，没有奖励怎么行，让我们现在就在收集星星时添加得分奖励的逻辑和显示。

## 添加分数文字（Label）

游戏开始时得分从0开始，每收集一个星星分数就会加1。要显示得分，首先要创建一个 **Label** 节点。在**层级管理器**中选中 `Canvas` 节点，右键点击并选择菜单中的 `创建新节点->创建渲染节点->Label（文字）` ，一个新的 Label 节点会被创建在 `Canvas` 下面，而且顺序在最下面。接下来我们要用如下的步骤配置这个 Label 节点：

1. 将该节点名字改为 `score`
2. 将 `score` 节点的位置（ `position` 属性）设为 `(0, 180)` 。
3. 选中该节点，编辑**Label**组件的 `string` 属性，填入 `Score: 0` 的文字。
4. 将**Label**组件的 `Font Size` 属性设为 `50` 。
5. 从**资源管理器**中拖拽 `assets/mikado_outline_shadow` 位图字体资源（注意图标是  ）到**Label**组件的 `Font` 属性中，将文字的字体替换成我们项目资源中的位图字体。

完成后效果如下图所示：




## 在 Game 脚本中添加得分逻辑

我们将会把计分和更新分数显示的逻辑放在 `Game` 脚本里，打开 `Game` 脚本开始编辑，首先在 `properties` 区块的最后添加分数显示 Label 的引用属性：

```
// Game.js
    properties: {
        // ...
        // score label 的引用
        scoreDisplay: {
            default: null,
            type: cc.Label
        }
    },
```

接下来在 `onLoad` 方法里添加计分用的变量的初始化：

```
// Game.js
```

```
    onLoad: function () {
        // ...
        // 初始化计分
        this.score = 0;
    },
```

然后在 `update` 方法后面添加名叫 `gainScore` 的新方法：

```
// Game.js
    gainScore: function () {
        this.score += 1;
        // 更新 scoreDisplay Label 的文字
        this.scoreDisplay.string = 'Score: ' + this.score.toString();
    },
```

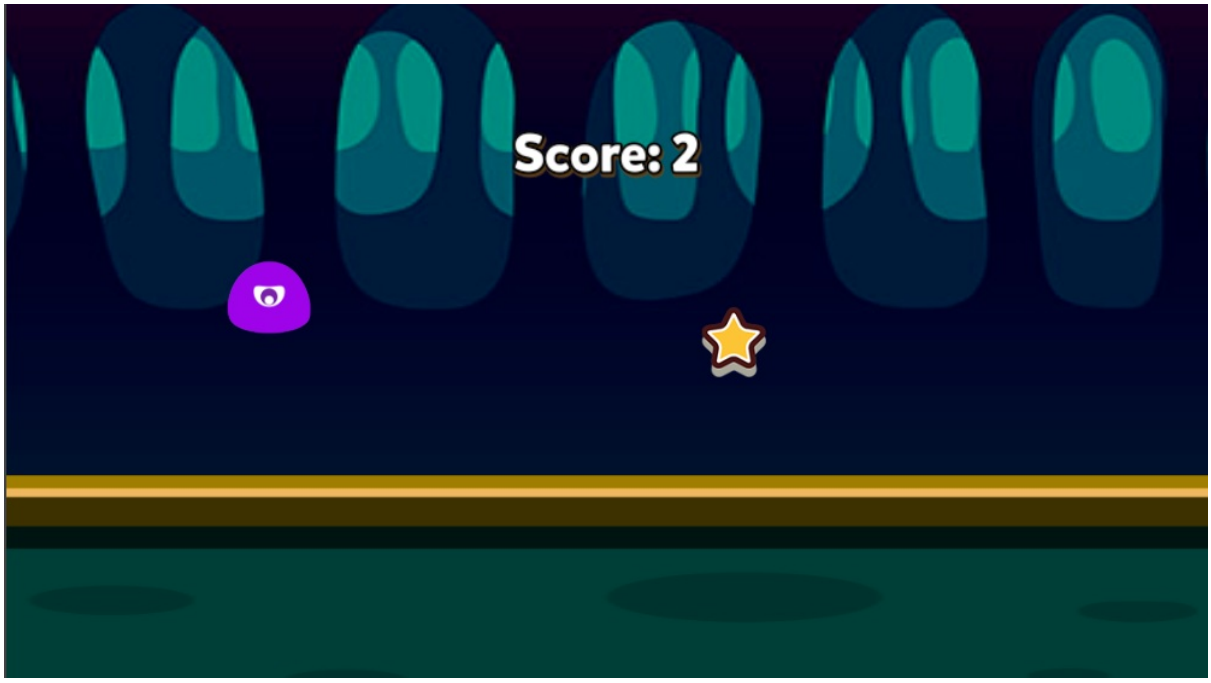
保存 `Game` 脚本后，回到 **层级管理器**，选中 `Canvas` 节点，然后把前面添加好的 `score` 节点拖拽到 **属性检查器** 里 `Game` 组件的 `Score Display` 属性中。

## 在 Star 脚本中调用 Game 中的得分逻辑

下面打开 `Star` 脚本，在 `onPicked` 方法中加入 `gainScore` 的调用：

```
// Star.js
    onPicked: function() {
        // 当星星被收集时，调用 Game 脚本中的接口，生成一个新的星星
        this.game.spawnNewStar();
        // 调用 Game 脚本的得分方法
        this.game.gainScore();
        // 然后销毁当前星星节点
        this.node.destroy();
    },
```

保存后预览，可以看到现在收集星星时屏幕正上方显示的分数会增加了！




# 失败判定和重新开始

现在我们的游戏已经初具规模，但得分再多，不可能失败的游戏也不会给人成就感。现在让我们加入星星定时消失的行为，而且让星星消失时就判定为游戏失败。也就是说，玩家需要在每颗星星消失之前完成收集，并不断重复这个过程完成玩法的循环。

## 为星星加入计时消失的逻辑

打开 `Game` 脚本，在 `onLoad` 方法的 `spawnNewStar` 调用之前加入计时需要的变量声明：

```
// Game.js
    onLoad: function () {
        // ...
        // 初始化计时器
        this.timer = 0;
        this.starDuration = 0;
        // 生成一个新的星星
        this.spawnNewStar();
        // 初始化计分
        this.score = 0;
    },
```

然后在 `spawnNewStar` 方法最后加入重置计时器的逻辑，其中 `this.minStarDuration` 和 `this.maxStarDuration` 是我们一开始声明的 `Game` 组件属性，用来规定星星消失时间的随机范围：

```
// Game.js
    spawnNewStar: function() {
        // ...
        // 重置计时器，根据消失时间范围随机取一个值
        this.starDuration = this.minStarDuration + cc.random0To1() * (this.maxStarDuration - this.minStarDuration);
        this.timer = 0;
    },
```

在 `update` 方法中加入计时器更新和判断超过时限的逻辑：

```
// Game.js
    update: function (dt) {
        // 每帧更新计时器，超过限度还没有生成新的星星
        // 就会调用游戏失败逻辑
        if (this.timer > this.starDuration) {
            this.gameOver();
            return;
        }
        this.timer += dt;
    },
```

最后加入 `gameOver` 方法，游戏失败时重新加载场景。

```
// Game.js
    gameOver: function () {
        this.player.stopAllActions(); //停止 player 节点的跳跃动作
        cc.director.loadScene('game');
    }
```

对 `Game` 脚本的修改就完成了，保存脚本，然后打开 `Star` 脚本，我们需要为即将消失的星星加入简单的视觉提示效果，在 `update` 方法最后加入以下代码：

```
// Star.js
    update: function() {
        // ...
        // 根据 Game 脚本中的计时器更新星星的透明度
```

```
        var opacityRatio = 1 - this.game.timer/this.game.starDuration;
        var minOpacity = 50;
        this.node.opacity = minOpacity + Math.floor(opacityRatio * (255 - minOpacity));
    }
```

保存 `Star` 脚本，我们的游戏玩法逻辑就全部完成了！现在点击**预览游戏**按钮，我们在浏览器看到的就是一个有核心玩法、激励机制、失败机制的合格游戏了。

# 加入音效

尽管很多人玩手游的时候会无视声音，我们为了教程展示的工作流程尽量完整，还是要补全加入音效的任务。

## 跳跃音效

首先加入跳跃音效，打开 `Player` 脚本，添加引用声音文件资源的 `jumpAudio` 属性：

```
// Player.js
    properties: {
        // ...
        // 跳跃音效资源
        jumpAudio: {
            default: null,
            url: cc.AudioClip
        },
    },
```

然后改写 `setJumpAction` 方法，插入播放音效的回调，并通过添加 `playJumpSound` 方法来播放声音：

```
// Player.js
    setJumpAction: function () {
        // 跳跃上升
        var jumpUp = cc.moveBy(this.jumpDuration, cc.p(0, this.jumpHeight)).easing(cc.easeCubicActionOut());
        // 下落
        var jumpDown = cc.moveBy(this.jumpDuration, cc.p(0, -this.jumpHeight)).easing(cc.easeCubicActionIn());
        // 添加一个回调函数，用于在动作结束时调用我们定义的其他方法
        var callback = cc.callFunc(this.playJumpSound, this);
        // 不断重复，而且每次完成落地动作后调用回调来播放声音
        return cc.repeatForever(cc.sequence(jumpUp, jumpDown, callback));
    },

    playJumpSound: function () {
        // 调用声音引擎播放声音
        cc.audioEngine.playEffect(this.jumpAudio, false);
    },
```

## 得分音效

保存 `Player` 脚本以后打开 `Game` 脚本，来添加得分音效，首先仍然是在 `properties` 中添加一个属性来引用声音文件资源：

```
// Game.js
    properties: {
        // ...
        // 得分音效资源
        scoreAudio: {
            default: null,
            url: cc.AudioClip
        }
    },
```

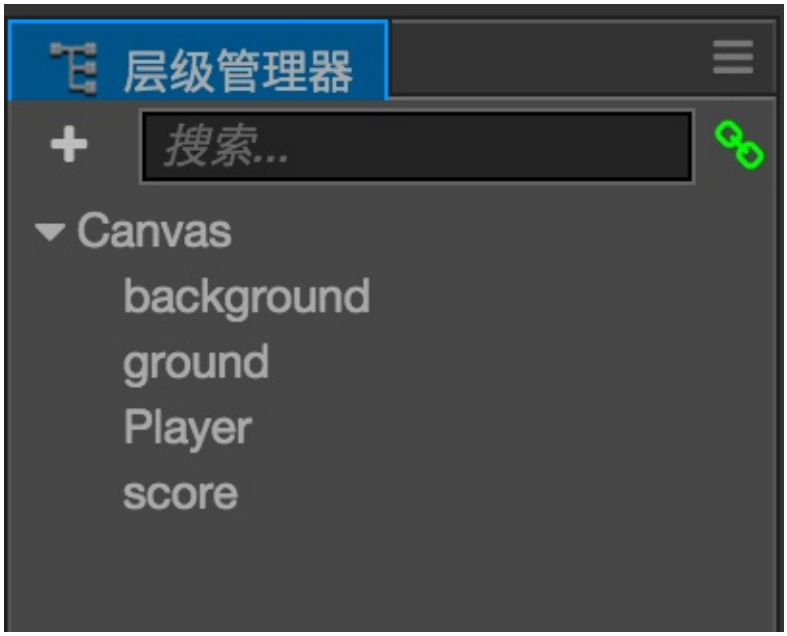
然后在 `gainScore` 方法里插入播放声音的代码：

```
// Game.js
    gainScore: function () {
        this.score += 1;
        // 更新 scoreDisplay Label 的文字
        this.scoreDisplay.string = 'Score: ' + this.score.toString();
        // 播放得分音效
        cc.audioEngine.playEffect(this.scoreAudio, false);
    },
```

保存脚本，回到**层级编辑器**，选中 `Player` 节点，然后从**资源管理器**里拖拽 `assets/audio/jump` 资源到 `Player` 组件的 `Jump Audio` 属性上。

然后选中 `Canvas` 节点，把 `assets/audio/score` 资源拖拽到 `Game` 组件的 `Score Audio` 属性上。

这样就大功告成了！完成形态的场景层级和各个关键组件的属性如下：




Game
Script
Game
script
Star Prefab
star
prefab
Max Star Dur...
3
Min Star Dura...
5
Ground
ground
Node
Player
Player
Node
Score Display
score
Label
Score Audio
score
audio-clip
添加组件


现在我们可以尽情享受刚制作完成的游戏了，您能打到多少分呢？别忘了您可以随时修改 `Player` 和 `Game` 组件里的移动控制和星星持续时间等游戏参数，来快速调节游戏的难度。修改组件属性之后需要保存场景，修改后的数值才会被记录下来。

# 总结

恭喜您完成了用 Cocos Creator 制作的第一个游戏！希望这篇快速入门教程能帮助您了解 Cocos Creator 游戏开发流程中的基本概念和工作流程。如果您对编写和学习脚本编程不感兴趣，也可以直接从完成版的项目中把写好的脚本复制过来使用。

接下来您还可以继续完善游戏的各方各面，以下是一些推荐的改进方向：

- 加入简单的开始菜单界面，在游戏运行的一开始显示开始按钮，点击按钮后才会开始游戏
- 为游戏失败加入简单的菜单界面，游戏失败后点击按钮才会重新开始
- 限制主角的移动不能超过视窗边界
- 为主角的跳跃动作加入更细腻的动画表现
- 为星星消失的状态加入计时进度条
- 收集星星时加入更华丽的效果
- 为触屏设备加入输入控制

以上这些方向都得到改善的游戏版本可以下载 进化版项目来参考和学习，这里就不再赘述了。

此外如果希望将完成的游戏发布到服务器上分享给好友玩耍，可以阅读预览和构建一节的内容。

今天的教程就到这里了，您可以立刻开始制作您的第二款 Cocos Creator 游戏，或者继续阅读本手册。关于本快速开始教程的任何问题，都可以在Github 上的本教程仓库提交反馈。

# 配置代码编辑环境

在快速上手教程中，我们介绍了在 **资源管理器** 中双击脚本文件打开代码编辑器快速编辑代码的方法。但编辑器内置的代码编辑器功能并不完善，只适合快速浏览和做少量编辑的需要。对程序员来说，我们需要更成熟完善的代码编辑环境。

# Visual Studio Code

Visual Studio Code （以下简称 VS Code）是微软新 推出的轻量化跨平台IDE，支持 Windows、Mac、Linux 平台，安装和配置非常简单。通过下面介绍的设置方法，使用 VS Code 管理和编辑项目脚本代码，可以轻松实现语法高亮、智能代码提示等功能，还可以直接使用 VS Code 调试网页和原生版本的游戏。

## 安装 VS Code

前往 VS Code 的官方网站，点击首页的下载链接即可下载。

Mac 用户解压下载包后双击 `Visual Studio Code` 即可运行。

Windows 用户下载后运行 `VSCodeSetup.exe` 按提示完成安装即可运行。

## 安装 Cocos Creator API 适配插件

在 Cocos Creator 中打开你的项目，然后选择主菜单里的 `开发者/安装 VS Code 扩展插件` 。

该操作会将 Cocos Creator API 适配插件安装到 VS Code 全局的插件文件夹中，一般在用户 Home 文件夹中的 `.vscode/extensions` 目录下。这个操作只需要执行一次，如果 API 适配插件更新了，则需要再次运行来更新插件。

安装成功后在 **控制台** 会显示绿色的提示： `VS Code extension installed to ...` 。这个插件的主要功能是为 VS Code 编辑状态下注入符合 Cocos Creator 组件脚本使用习惯的语法提示。

## 在项目中生成智能提示数据

如果希望在代码编写过程中自动提示 Cocos Creator 引擎 API，需要通过菜单生成 API 智能提示数据并自动放进项目路径下。

选择主菜单的 `开发者/更新 VS Code 智能提示数据` 。该操作会将根据引擎 API 生成的 `creator.d.ts` 数据文件复制到项目根目录下（注意是在 `assets` 目录外面），操作成功时会在 **控制台** 显示绿色提示： `API data generated and copied to ...` 。

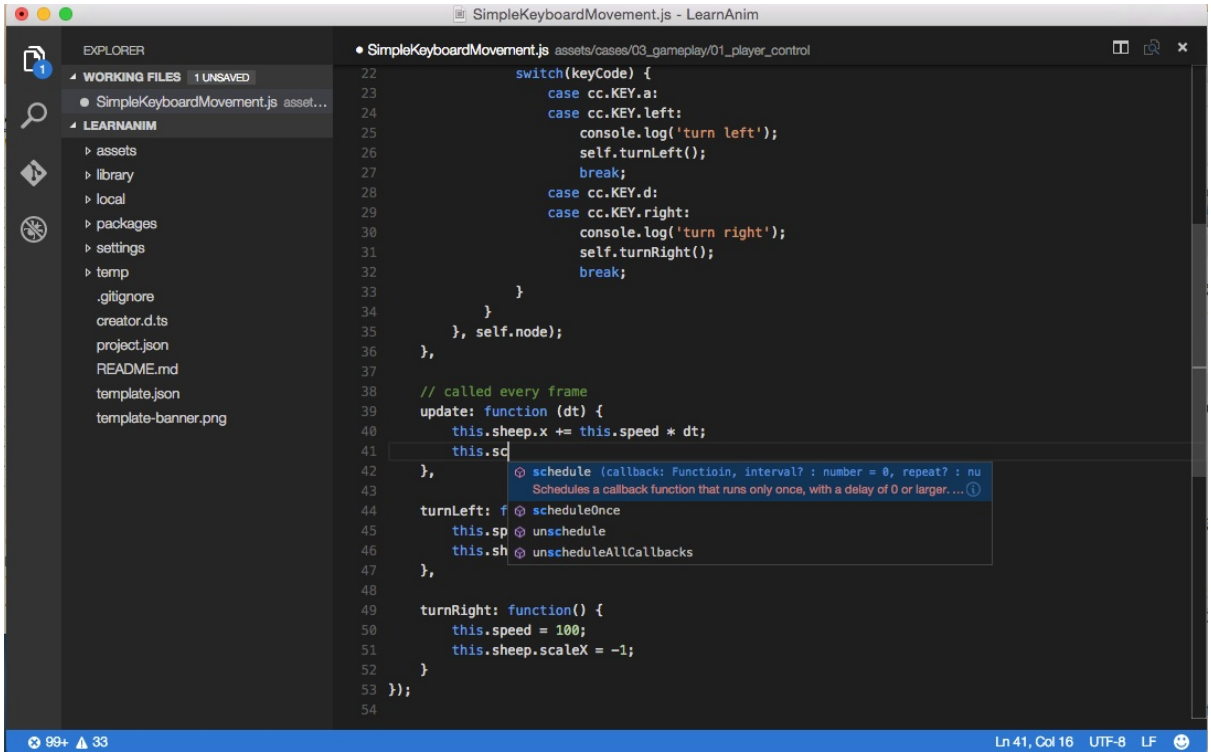
对于每个不同的项目都需要运行一次这个命令，如果 Cocos Creator 版本更新了，也需要打开您的项目重新运行一次这个命令，来同步最新引擎的 API 数据。

**注意** 从 VS Code 0.10.11 版开始，需要在项目根目录中添加 `jsconfig.json` 设置文件才能正确的使用包括智能提示在内的 JavaScript 语言功能，在执行上面的命令时，预设的 `jsconfig.json` 文件会和 `creator.d.ts` 一起拷贝到您的项目根目录中。

## 使用 VS Code 打开和编辑项目

现在可以运行我们之前下载安装好的 VS Code 了，启动后选择主菜单的 `File/Open...` ，在弹出的对话框中选择您的项目根目录，也就是 `assets` ，`project.json` 所在的路径。

现在新建一个脚本，或者打开原有的脚本编辑时，就可以享受智能语法提示的功能了。




注意 `creator.d.ts` 文件必须放在 VS Code 打开的项目路径下，才能使用智能提示功能。

## 设置文件显示和搜索过滤

在 VS Code 的主菜单中选择 `Code/Preferences/User Settings`，这个操作会打开用户配置文件，在配置文件中加入以下内容：

```
    "search.exclude": {
        "**/node_modules": true,
        "**/bower_components": true,
        "build/": true,
        "temp/": true,
        "library/": true,
        "**/*.anim": true
    },
    "files.exclude": {
        "**/.git": true,
        "**/.DS_Store": true,
        "**/*.meta": true,
        "library/": true,
        "local/": true,
        "temp/": true
    }
```

上面的字段将为 VS Code 设置搜索时排除的目录，和在文件列表中隐藏的文件类型。由于 `build`，`temp`，`library` 都是编辑器运行时自动生成的路径，而且会包含我们写入的脚本内容，所以应该在搜索中排除。而 `assets` 目录下的每个文件都会生成一个 `.meta` 文件，一般来说我们不需要关心他的内容，只要让编辑器帮我们管理这些文件就可以了。

## 使用 VS Code 激活脚本编译

使用外部文本编辑器修改项目脚本后，要重新激活 Cocos Creator 窗口才能触发脚本编译，我们在新版本的 Creator 中增加了一个预览服务器的 API，可以通过向特定地址发送请求来激活编辑器的编译。

## 安装 cURL

首先需要确保你的操作系统中可以运行 cURL 命令，如果在 Windows 操作系统的命令行中运行 `curl` 提示找不到命令，则需要先安装 curl 到你的系统：

- 前往 http://www.confusedbycode.com/curl/
- 点击下图箭头所示的控件，完成人机身份验证

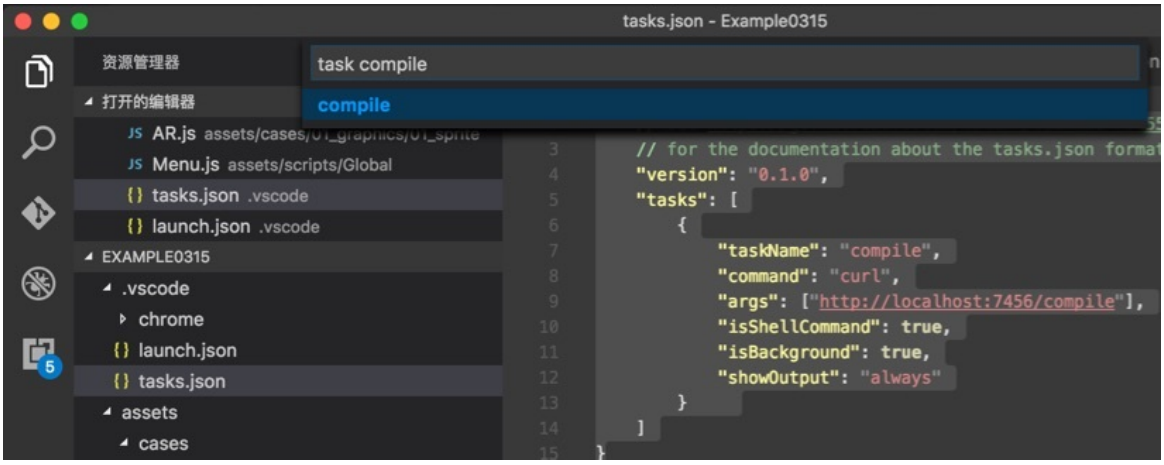

- 点击 `curl-7.46.0-win64.exe` 开始下载并安装

安装时请使用默认设置，安装完成后可以打开一个命令行窗口，输入 `curl`，如果提示 `curl: try 'curl --help' or 'curl --manual' for more information` 就表示安装成功了。

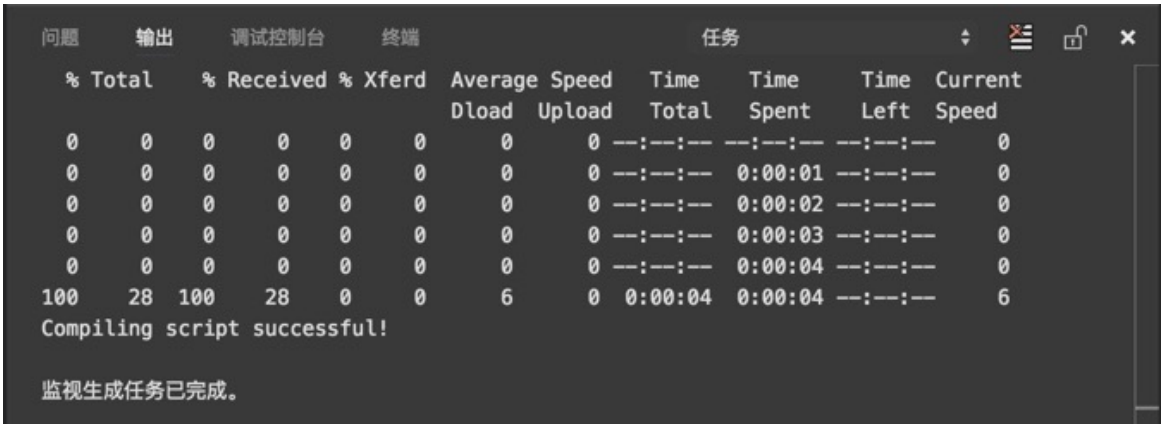
## 添加 VS Code 编译任务

要在 VS Code 中激活脚本编译，需要执行以下的工作流程：

1. 在编辑器主菜单里执行 `开发者->VS Code 工作流->添加编译任务` 。该操作会在项目的 `.vscode` 文件夹下添加 `tasks.json` 任务配置文件。
2. 在 VS Code 里按下 `Cmd/Ctrl+p`，激活 **快速打开** 输入框，然后输入 `task compile`




3. 任务运行成功的话，会在 VS Code 窗口下方的输出面板中显示如下结果




VS Code 还可以为任务配置快捷键，请打开主菜单的 `Code -> 首选项 -> 键盘快捷方式`，并在右侧的 `keybindings.json` 里添加以下条目：

```
{
    "key": "ctrl+p", //请配置自己习惯的快捷键
    "command": "workbench.action.tasks.runTask",
    "args": "compile"
}
```

接下来就可以在 VS Code 里一键完成项目脚本编译了！更多关于 VS Code 中配置和执行任务的信息，请参阅 Integrate with External Tools via Tasks 文档。

## 使用 VS Code 调试网页版游戏

VS Code 有着优秀的 debug 能力，我们可以直接在源码工程中调试网页版游戏程序。

首先需要安装：

- Chrome（谷歌浏览器）
- VS Code 插件：Debugger for Chrome

安装 VS Code 插件时，请点击 VS Code 左侧导航栏的 `扩展` 按钮打开扩展面板，并在搜索框中输入 `Debugger for Chrome` 并点击安装继续。安装之后可能需要重启 VS Code 才能生效。

接下来在 Cocos Creator 编辑器主菜单里执行 `开发者->VS Code 工作流->添加 Chrome Debug 配置`，这个菜单命令会在你的项目文件夹下添加一个 `.vscode/launch.json` 文件作为调试器的配置，之后你就可以在 VS Code 里点击左侧栏的 `调试` 按钮打开调试面板，并在最上方的调试配置中选择 `Creator Debug: Launch Chrome`，然后点击绿色的开始按钮开始调试。

调试的时候依赖 Cocos Creator 编辑器内置的 Web 服务器，所以需要在编辑器启动状态下才能进行调试。如果编辑器预览游戏时使用的端口不是默认端口，则需要手动修改 `launch.json` 里的 `url` 字段，将正确的端口添加上去。

调试过程中可以在源码文件上直接下断点，进行监控，是比使用 Chrome 内置的 DevTools 调试更方便和友好的工作流程。

## 使用 VS Code 调试原生工程

调试原生工程的工作流程请查阅 原生平台调试。

## 学习 VS Code 的使用方法

前往 VS Code 官网的文档页面，了解从编辑功能操作、个性化定制、语法高亮设置到插件扩展等各方面的使用方法。

# 项目结构

通过 Dashboard，我们可以创建一个 Hello World 项目作为开始，创建之后的项目有特定的文件夹结构，我们将在这一节熟悉 Cocos Creator 项目的文件夹结构。

# 项目文件夹结构

初次创建并打开一个 Cocos Creator 项目后，您的项目文件夹将会包括以下结构：

```
ProjectName（项目文件夹）
├──assets
├──library
├──local
├──settings
├──temp
└──project.json
```

下面我们将会介绍每个文件夹的功能。

## 资源文件夹（assets）

`assets` 将会用来放置您游戏中所有本地资源、脚本和第三方库文件。只有在 `assets` 目录下的内容才能显示在 **资源管理器** 中。`assets` 中的每个文件在导入项目后都会生成一个相同名字的 `.meta` 文件，用于存储该文件作为资源导入后的信息和与其他资源的关联。一些第三方工具生成的工程或设计原文件，如 TexturePacker 的 `.tps` 文件，或 Photoshop 的 `.psd` 文件，可以选择放在 `assets` 外面来管理。

## 资源库（library）

`library` 是将 `assets` 中的资源导入后生成的，在这里文件的结构和资源的格式将被处理成最终游戏发布时需要的形式。如果您使用版本控制系统管理您的项目，这个文件夹是不需要进入版本控制的。

当 `library` 丢失或损坏的时候，只要删除整个 `library` 文件夹再打开项目，就会重新生成资源库。

## 本地设置（local）

`local` 文件夹中包含该项目的本地设置，包括编辑器面板布局，窗口大小，位置等信息。您不需要关心这里的内容，只要按照您的习惯设置编辑器布局，这些就会自动保存在这个文件夹。一般 `local` 也不需要进入版本控制。

## 项目设置（settings）

`settings` 里保存项目相关的设置，如 **构建发布** 菜单里的包名、场景和平台选择等。这些设置需要和项目一起进行版本控制。

## project.json

`project.json` 文件和 `assets` 文件夹一起，作为验证 Cocos Creator 项目合法性的标志。只有包括了这两个内容的文件夹才能作为 Cocos Creator 项目打开。而 `project.json` 本身目前只用来规定当前使用的引擎类型和插件存储位置，不需要用户关心其内容。

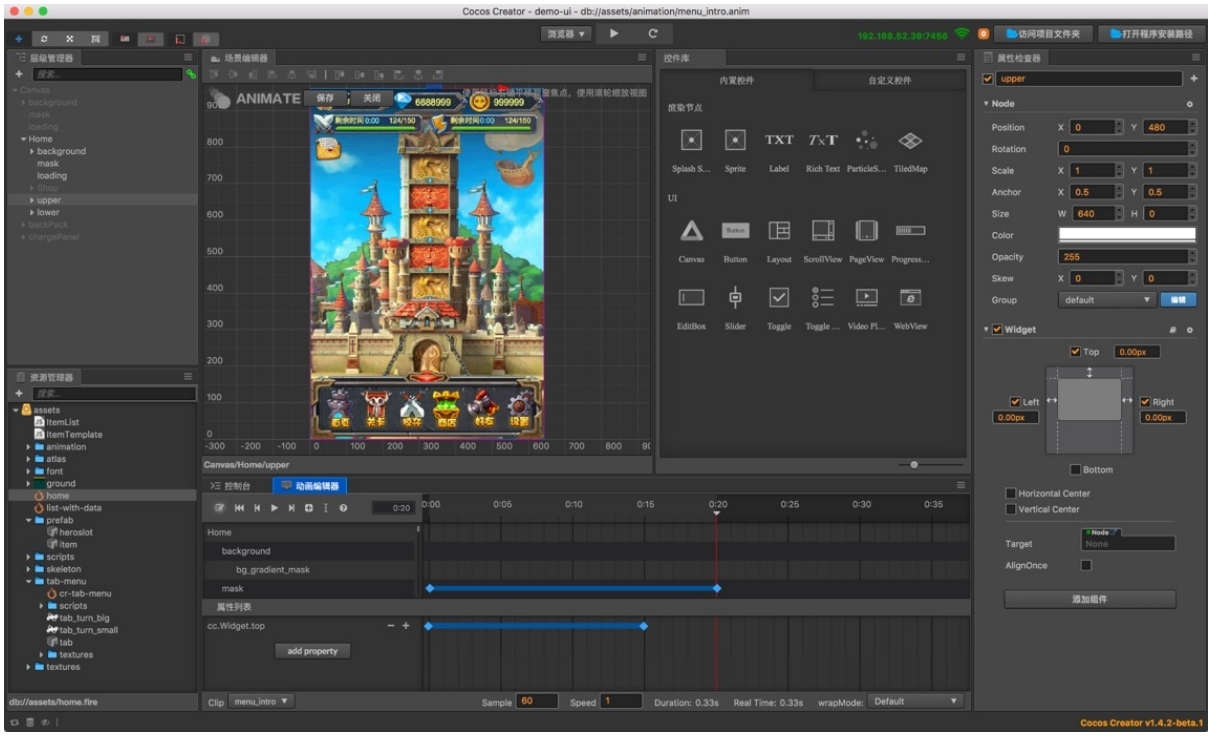
这个文件也应该纳入版本控制。

## 构建目标（build）

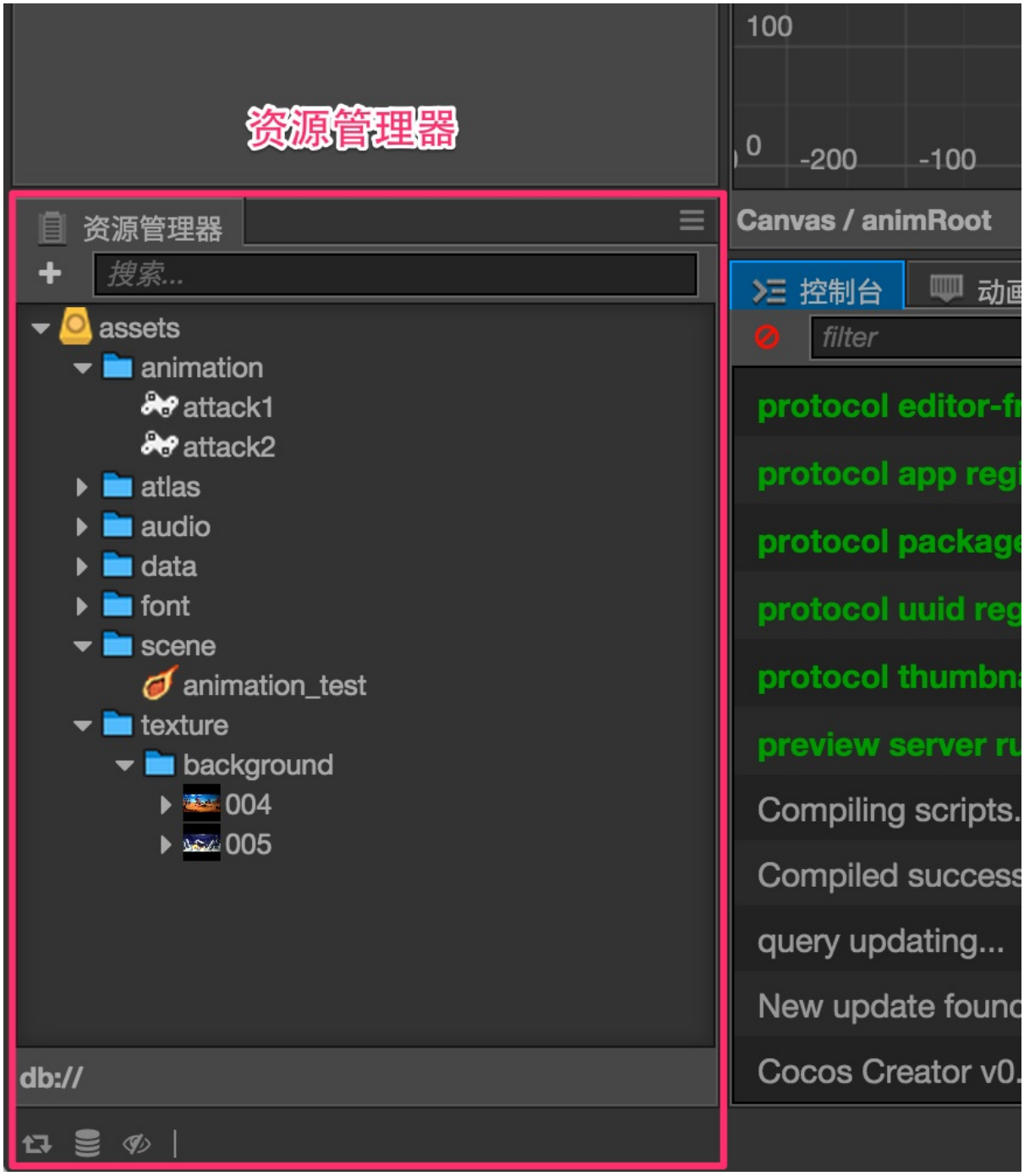
在使用主菜单中的 `项目->构建发布...` 使用默认发布路径发布项目后，编辑器会在项目路径下创建 `build` 目录，并存放所有目标平台的构建工程。由于每次发布项目后资源 id 可能会变化，而且构建原生工程时体积很大，所以此目录建议不进入版本控制。

# 编辑器界面介绍

这一章将会介绍编辑器界面，熟悉组成编辑器的各个面板、菜单和功能按钮。Cocos Creator 编辑器由多个面板组成，面板可以自由移动、组合，以适应不同项目和开发者的需要。我们在这里将会以默认编辑器布局为例，快速浏览各个面板的名称和作用：



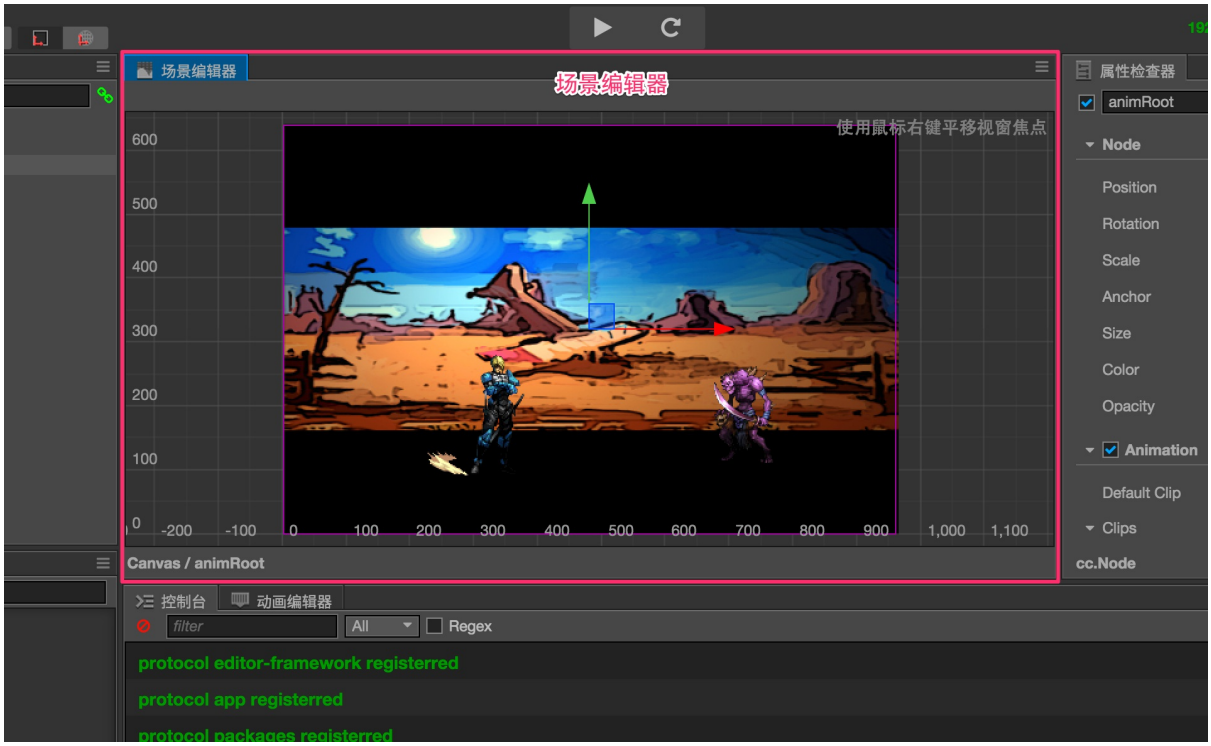

# 资源管理器




**资源管理器**里显示了项目资源文件夹（ assets ）中的所有资源。这里会以树状结构显示文件夹并自动同步在操作系统中对项目资源文件夹内容的修改。您可以将文件从项目外面直接拖拽进来，或使用菜单导入资源。

详情请阅读 资源管理器 一节。

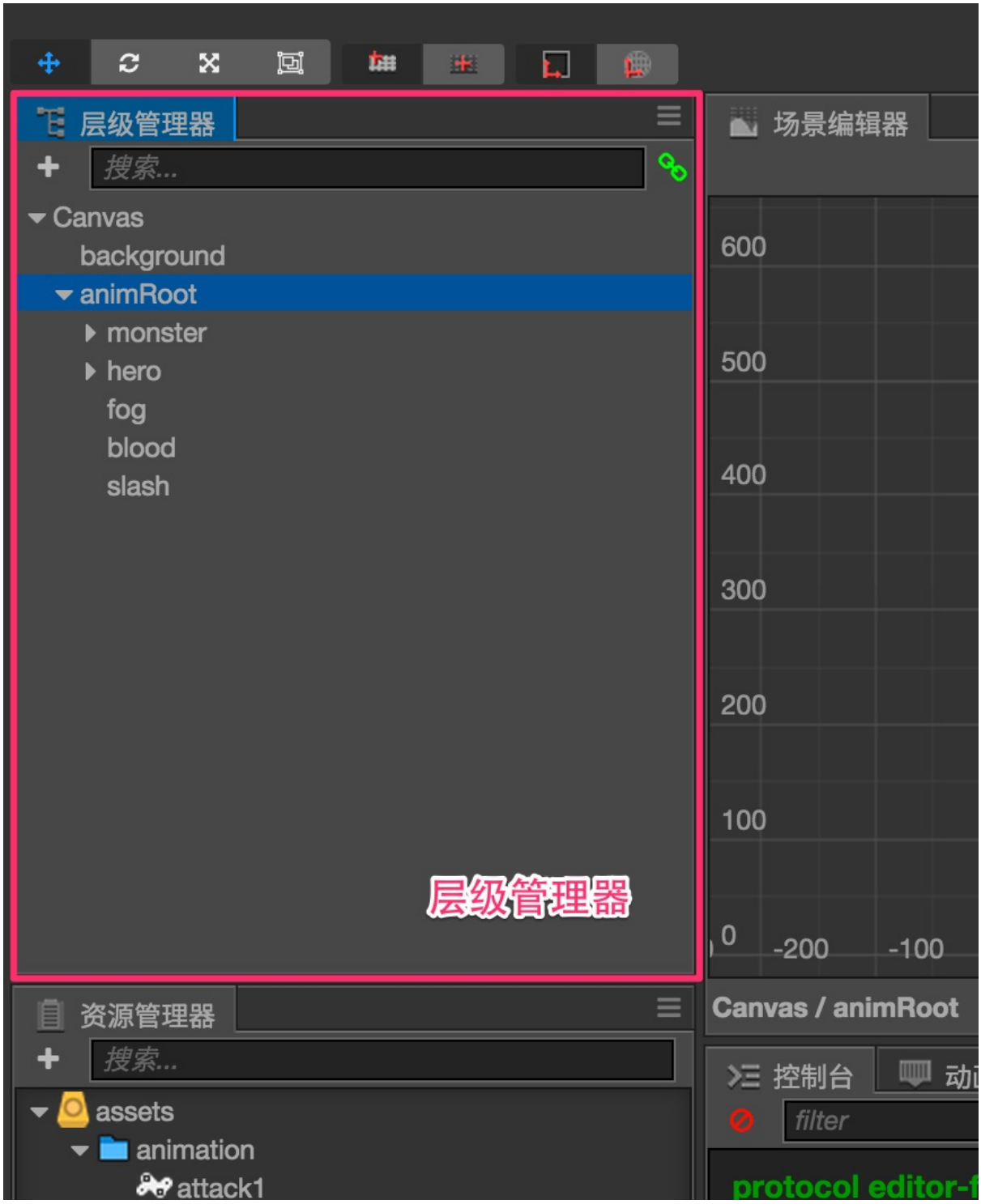
## 场景编辑器




**场景编辑器**是用来展示和编辑场景中可视内容的工作区域。所见即所得的场景搭建工作都依靠场景编辑器中的显示来完成。

详情请阅读 场景编辑器 一节。

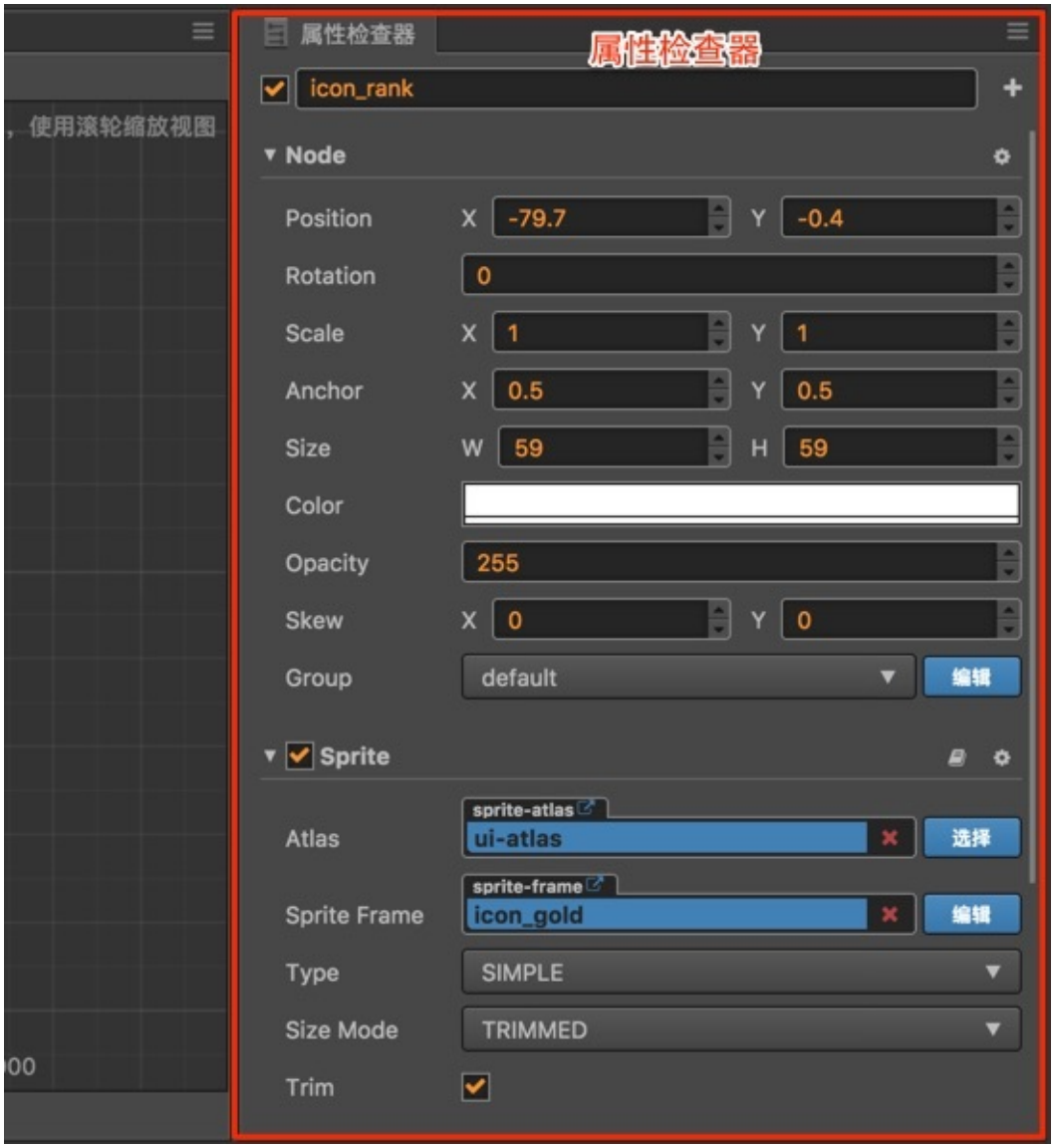
# 层级管理器




**层级管理器**用树状列表的形式展示场景中的所有节点和他们的层级关系，所有在**场景编辑器**中看到的内容都可以在**层级管理器**中找到对应的节点条目，在编辑场景时这两个面板的内容会同步显示，一般我们也会同时使用这两个面板来搭建场景。

详情请阅读 层级管理器 一节。

## 属性检查器




**属性检查器**是我们查看并编辑当前选中节点和组件属性的工作区域，这个面板会以最适合的形式展示和编辑来自脚本定义的属性数据。

详情请阅读 属性检查器 一节。

# 控件库


**控件库** 预设控件的仓库库，可以通过拖拽方式添加到场景中，并且可以将用户自己的预制资源（prefab）添加到控件库里方便再次使用。

详情请阅读 控件库 一节。

# 工具栏


**工具栏**上包括了场景编辑工具和预览游戏时的操作按钮，最右边显示了远程测试和调试时使用访问地址，以及连接中的设备数。

详情请阅读 工具栏 一节。

# 偏好设置


**偏好设置** 里提供各种编辑器个性化的全局设置，包括原生开发环境、游戏预览、脚本编辑工具等。

详情请阅读 偏好设置 一节。

# 项目设置


**项目设置** 里提供各种项目特定的个性化设置，包括分组设置、模块设置、预览运行等。

详情请阅读 项目设置 一节。

# 其他功能

其他重要的编辑器基础功能包括：

- 主菜单
- 工具栏
- 编辑器布局
- 构建预览

# 资源管理器（Assets）

**资源管理器**是我们用来访问和管理项目资源的工作区域。在开始制作游戏时，添加资源到这里通常是必须的步骤。您可以使用**HelloWorld**模板新建一个项目，就可以看到**资源管理器**中包含了一些基本资源类型。

# 界面介绍

**资源管理器**将项目资源文件夹中的内容以树状结构展示出来，注意只有放在项目文件夹的 `assets` 目录下的资源才会显示在这里。关于项目文件夹结构说明请阅读 项目结构 一节。下面我们介绍各个界面元素：


- 左上角的 **创建按钮** 用来创建新资源
- 右上的文本输入框可以用来搜索过滤文件名包含特定文本的资源
- 面板主体是资源文件夹的资源列表，可以在这里用右键菜单或拖拽操作对资源进行增删修改。

## 资源列表

资源列表中可以包括任意文件夹结构，文件夹在**资源管理器**中会以图标显示，点击图标左边的箭头就可以展开/折叠该文件夹中的内容。

除了文件夹之外列表中显示的都是资源文件，资源列表中的文件会隐藏扩展名，而以图标指示文件或资源的类型，比如**HelloWorld**模板创建出的项目中包括了三种核心资源：

- **图片资源**：目前包括 `jpg`，`png` 等图像文件，图标会显示为图片的缩略图。
- **脚本资源**：程序员编写的 JavaScript 脚本文件，以 `js` 为文件扩展名。我们通过编辑这些脚本为添加组件功能和游戏逻辑。
- **场景资源**：双击可以打开的场景文件，打开了场景文件我们才能继续进行内容创作和生产。

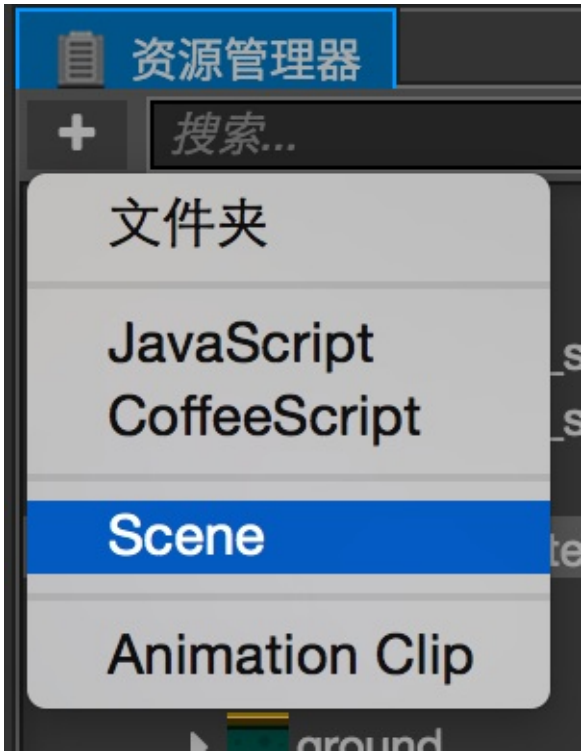
更多常见资源类型和资源工作流程，请阅读资源工作流程一章。

## 创建资源

目前可以在**资源管理器**中创建的资源有以下几类：

- 文件夹
- 脚本文件
- 场景
- 动画剪辑

点击左上角的**创建按钮**，就会弹出包括上述资源列表的创建资源菜单。点击其中的项目就会在当前选中的位置新建相应资源。



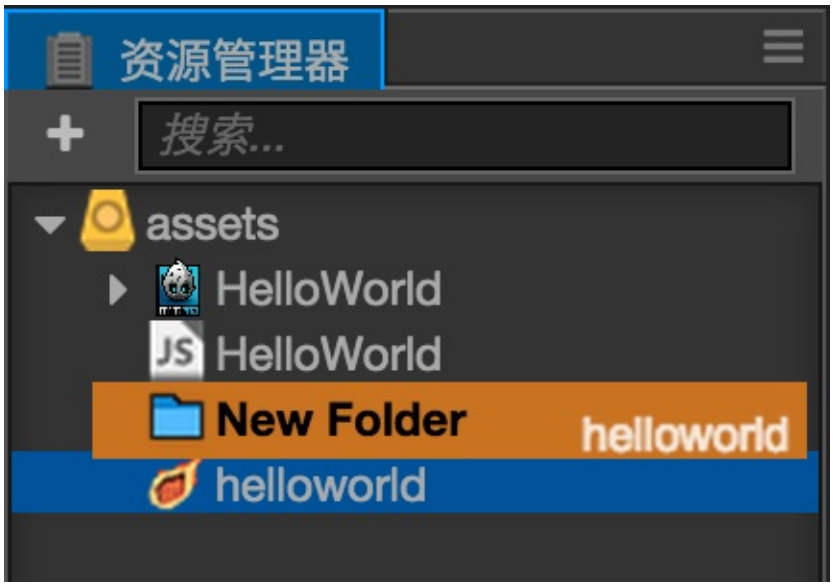

## 选择资源

在资源列表中可以使用以下的资源选择操作：

- 点击来选中单个资源
- 按住`Ctrl`或`Cmd`点击，可以将更多资源加入选择中
- 按住`Shift`点击，可以连续选中多个资源

对于选中的资源，可以执行移动、删除等操作。

## 移动资源

选中资源后（可多选），按住鼠标拖拽可以将资源移动到其他位置。将资源拖拽到文件夹上时，会看到鼠标悬停的文件夹以橙色高亮显示。 

这时松开鼠标，就会将资源移动到高亮显示的文件夹下。

## 删除资源

对于已经选中的资源，可以执行以下操作进行删除：

- 右键点击，并选择弹出菜单中的 `删除`
- 选中资源后直接按`Delete`（Windows）或`Cmd + Backspace`（Mac）

由于删除资源是不可撤销的操作，所以会弹出对话框要求用户确认。确定后资源就会被删除，无法从回收站（Windows）或废纸篓（Mac）找回！请一定要谨慎使用，做好版本管理或手动备份。

## 其他操作

**资源管理器**的右键菜单里还包括以下操作：

- `重命名`：对资源进行重命名
- `新建`：和**创建按钮**功能相同，会将资源添加到当前选中的文件夹下，如果当前选中的是资源文件，会将新增资源添加到和当前选中资源所在文件夹中。
- `在 资源管理器 （Windows）或 Finder （Mac）中显示`：在操作系统的文件管理器窗口中打开该资源所在的文件夹。
- `前往 Library 中的资源位置`：打开项目文件夹的 `Library` 中导入资源的位置，详情请阅读项目结构一节。
- `显示 UUID`：在**控制台**窗口显示当前选中资源的 UUID。
- `刷新`：重新执行该资源的导入操作。

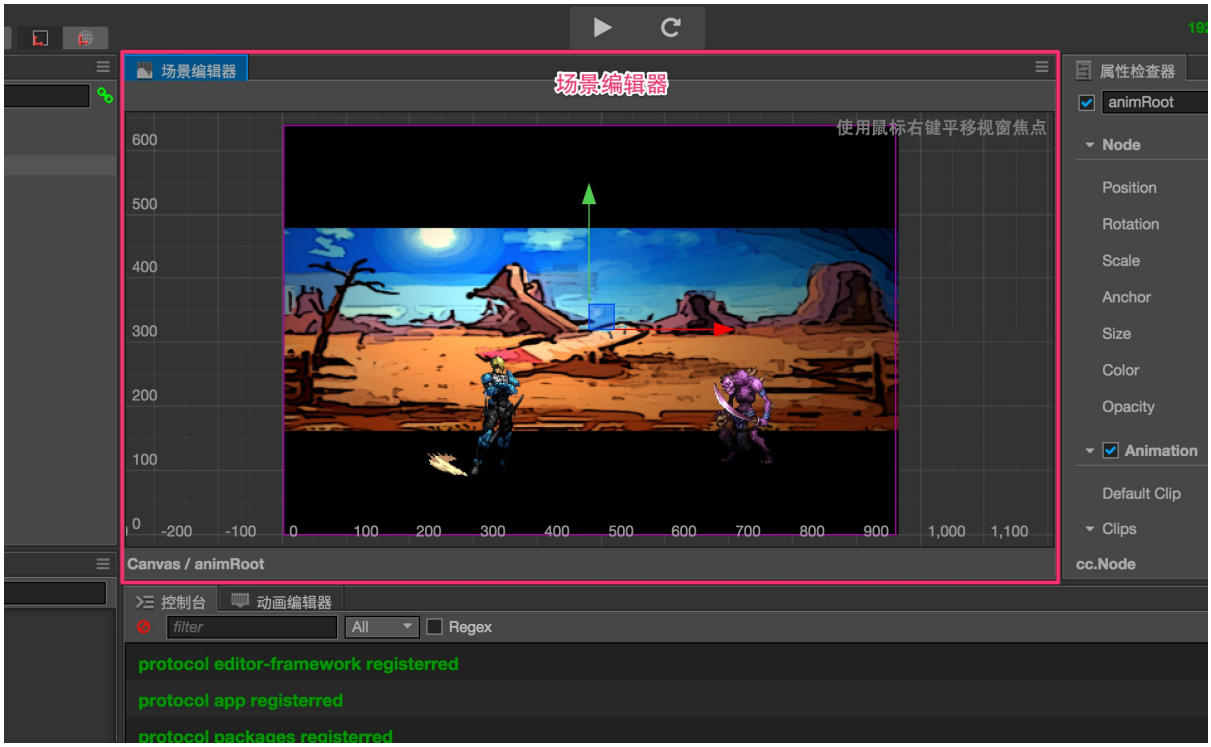
另外对于特定资源类型，双击资源可以进入该资源的编辑状态，如场景资源和脚本资源。

# 过滤资源

在**资源管理器**右上的搜索框中输入文本，可以过滤出文件名包括输入文本的所有资源。也可以输入 `*.png` 这样的文件扩展名，会列出所有特定扩展名的资源。


继续前往场景编辑器说明文档。

# Scene 场景编辑器




**场景编辑器**是内容创作的核心工作区域，您将使用它选择和摆放场景图像、角色、特效、UI 等各类游戏元素。在这个工作区域里，您可以选中并通过**变换工具**修改节点的位置、旋转、缩放、尺寸等属性，并可以获得所见即所得的场景效果预览。

# 视图介绍

## 导航

您可以通过以下的操作，来移动和定位**场景编辑器**的视图：

- 鼠标右键拖拽：平移视图。
- 鼠标滚轮：以当前鼠标悬停位置为中心缩放视图。

## 坐标系和网格

场景视图的背景会显示一组标尺和网格，表示**世界坐标系**中各个点的位置信息。读数为 `(0,0)` 的点为场景中世界坐标系的原点。使用鼠标滚轮缩小视图显示时，每一个刻度代表 `100` 像素的距离。根据当前视图缩放尺度的不同，会在不同刻度上显示代表该点到原点距离的数字，单位都是像素。

场景中的标尺和网格是我们摆放场景元素时位置的重要参考信息，关于坐标系和位置等节点属性的关系，请阅读坐标系和变换一节。

## 设计分辨率指示框

视图中的紫色线框表示场景中默认会显示的内容区域，这块区域的大小由**设计分辨率**决定。关于设计分辨率的设置和效果请阅读Canvas组件参考一节。

# 选取节点

鼠标悬浮到场景中的节点上时，节点的约束框将会以灰色单线显示出来。此时单击鼠标，就会选中该节点。选择节点是使用变换工具设置节点位置、旋转、缩放等操作的前提。

选中的节点周围将会有蓝色的线框提示节点的约束框。

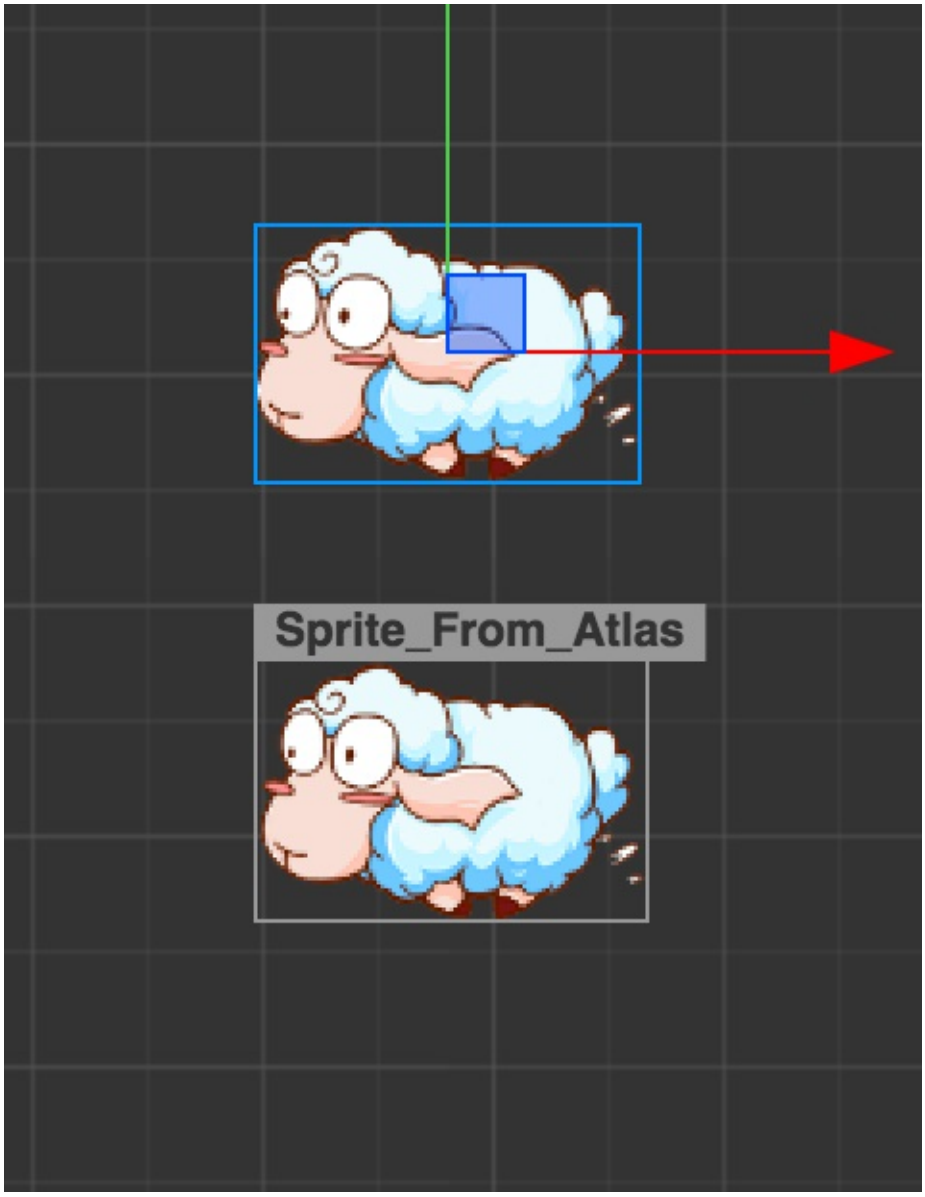
## 节点名称提示

鼠标悬浮在节点上时，与节点的约束框同时显示的还有节点的名称，在节点比较密集时可以先根据悬浮提示确定要选择的目标，然后点击确认您的选择。

关于节点的命名，请阅读层级管理器的介绍内容。

## 节点的约束框

节点在鼠标悬浮或选中状态下都能够看到约束框（灰色或蓝色的线框），约束框的矩形区域表示节点的尺寸（**size**）属性大小。即使节点没有包含图像渲染组件（如**Sprite**），也可以为节点设置 `size` 属性，而节点约束框以内的透明区域都可以被鼠标悬浮和点击选中。



节点的尺寸（ `size` ）属性在多分辨率适配和排版策略中有非常重要的作用，关于节点尺寸使用的更多信息请阅读多分辨率适配和元素对齐的相关内容。

### 多选节点

在**场景编辑器**中按住鼠标左键并拖拽，可以画出一个蓝色覆盖的选取框，和选取框有部分重合的节点，在放开鼠标按键后都会被一起选中。在放开鼠标键之前可以任意滑动鼠标来更改选取框的区域。

选中多个节点后，进行的任何变换操作都会同时作用于所有选中的节点。

## 使用变换工具布置节点

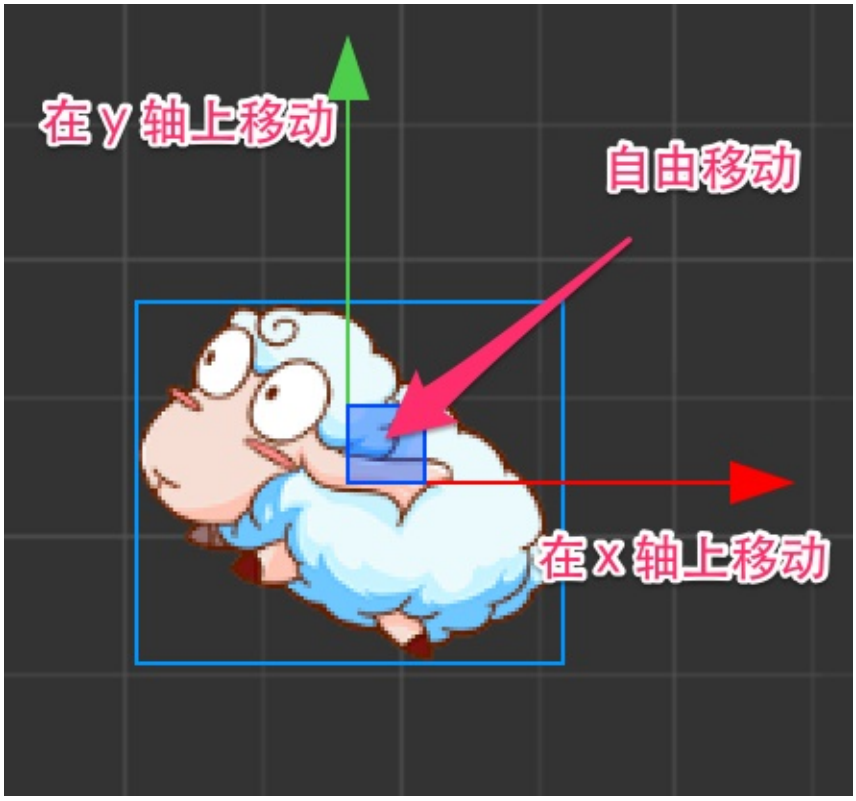
**场景编辑器**的核心功能就是以所见即所得的方式编辑和布置场景中的可见元素。我们主要通过主窗口工具栏左上角的一系列**变换工具**来将场景中的节点按我们希望的方式布置。

### 移动变换工具

**移动变换工具**是打开编辑器时默认处于激活状态的变换工具，之后这个工具也可以通过点击位于主窗口左上角工具栏第一个按钮来激活。

选中任何节点，就能看到节点中心（或锚点所在位置）上出现了由红绿两个箭头和蓝色方块组成的移动控制手柄（gizmo）。

**控制手柄**是指场景编辑器中在特定编辑状态下显示出的可用鼠标进行交互操作的控制器。这些控制器只用来辅助编辑，不会在游戏运行时显示。



移动变换工具激活时：

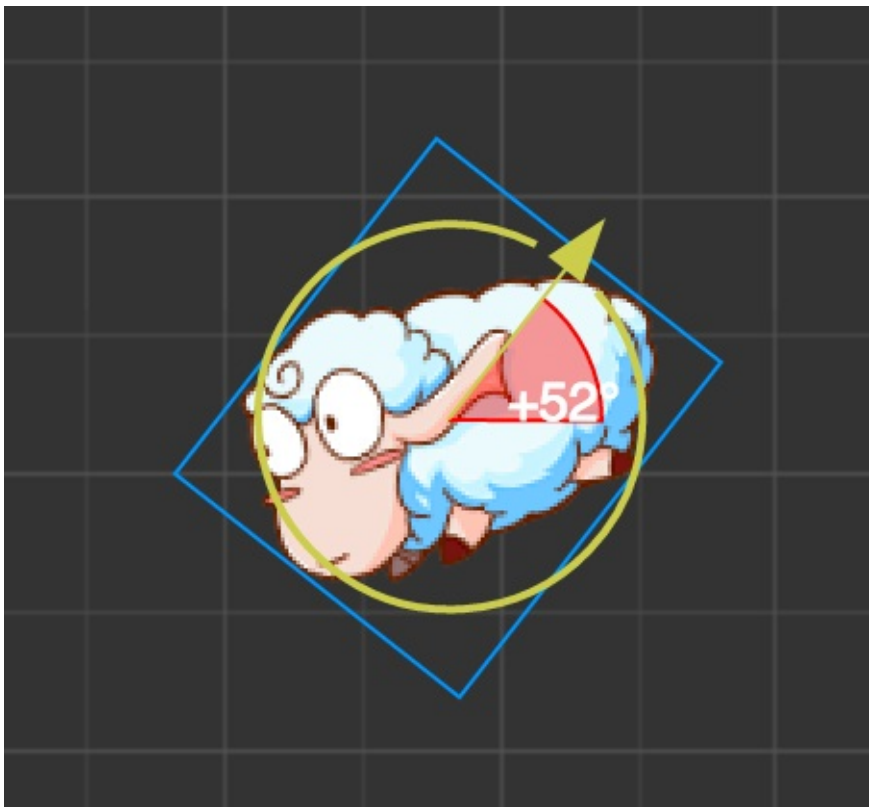
- 按住红色箭头拖拽鼠标，将在 x 轴方向上移动节点；
- 按住绿色箭头拖拽鼠标，将在 y 轴方向移动节点；
- 按住蓝色方块拖拽鼠标，可以同时在两个轴向自由移动节点。

场景编辑器处在激活状态时，按下快捷键W，即可随时切换到移动变换工具。

## 旋转变换工具

点击主窗口左上角工具栏第二个按钮，或在使用场景编辑器时按下E快捷键，即可激活**旋转变换工具**。

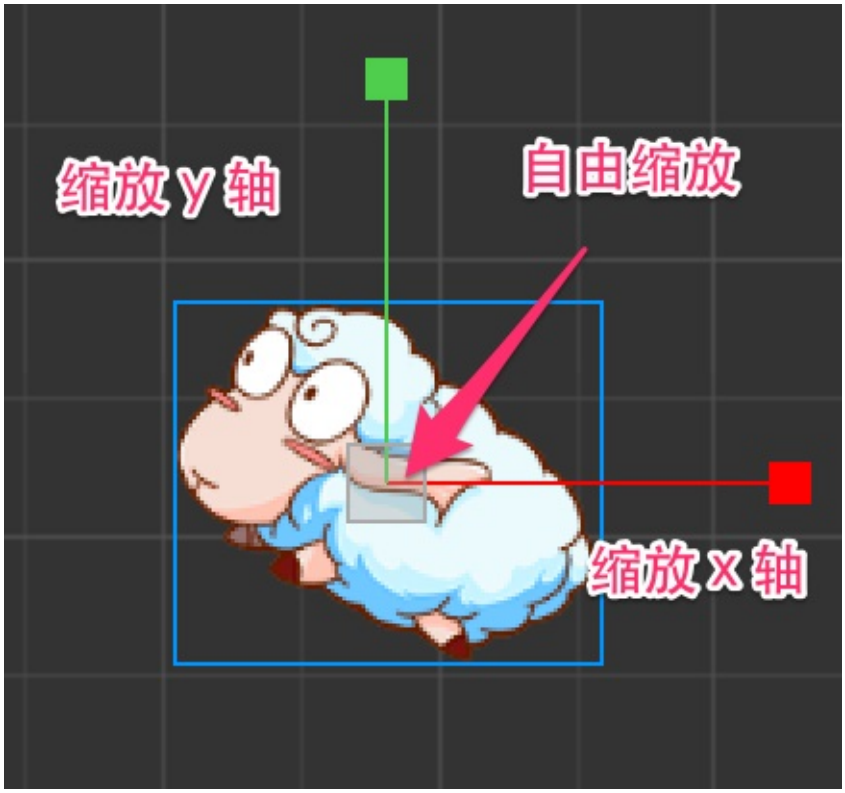
旋转变换工具的手柄主要是一个箭头和一个圆环组成，箭头所指的方向表示当前节点旋转属性（`rotation`）的角度。拖拽箭头或圆环内任意一点就可以旋转节点，放开鼠标之前，可以在控制手柄上看到当前旋转属性的角度值。




## 缩放变换工具

点击主窗口左上角工具栏第三个按钮，或在使用场景编辑器时按下R快捷键，即可激活**缩放变换工具**。

- 按住红色方块拖拽鼠标，在 x 轴方向上缩放节点图像；
- 按住绿色方块拖拽鼠标，在 y 轴方向上缩放节点图像；
- 按住中间的黄色方块，保持宽高比的前提下整体缩放节点图像。



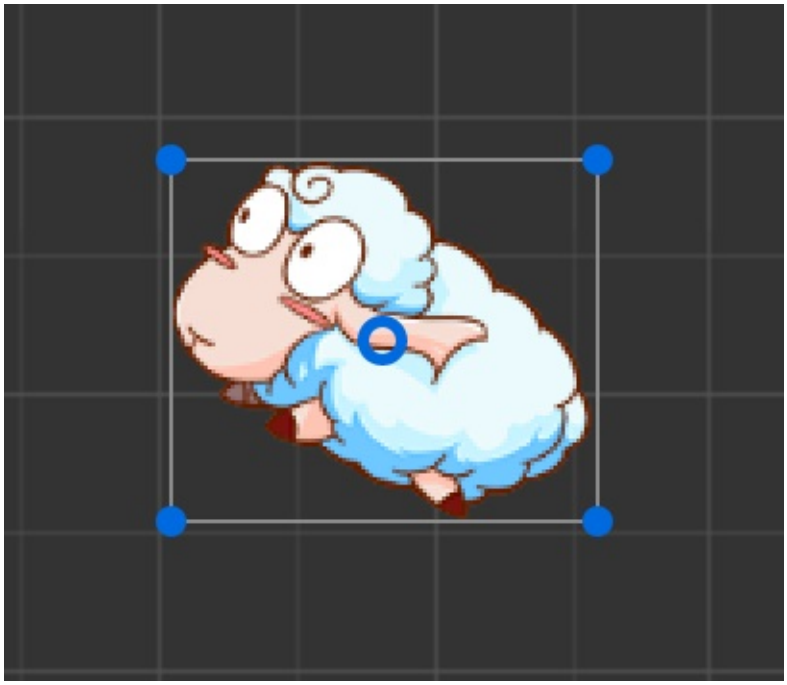

缩放节点时，会同比缩放所有的子节点。

## 矩形变换工具

点击主窗口左上角工具栏第四个按钮，或在使用场景编辑器时按下`T`快捷键，即可激活**矩形变换工具**。

拖拽控制手柄的任一顶点，可以在保持对角顶点位置不变的情况下，同时修改节点尺寸中的 `width` 和 `height` 属性。

拖拽控制手柄的任一边，可以在保持对边位置不变的情况下，修改节点尺寸中的 `width` 或 `height` 属性。

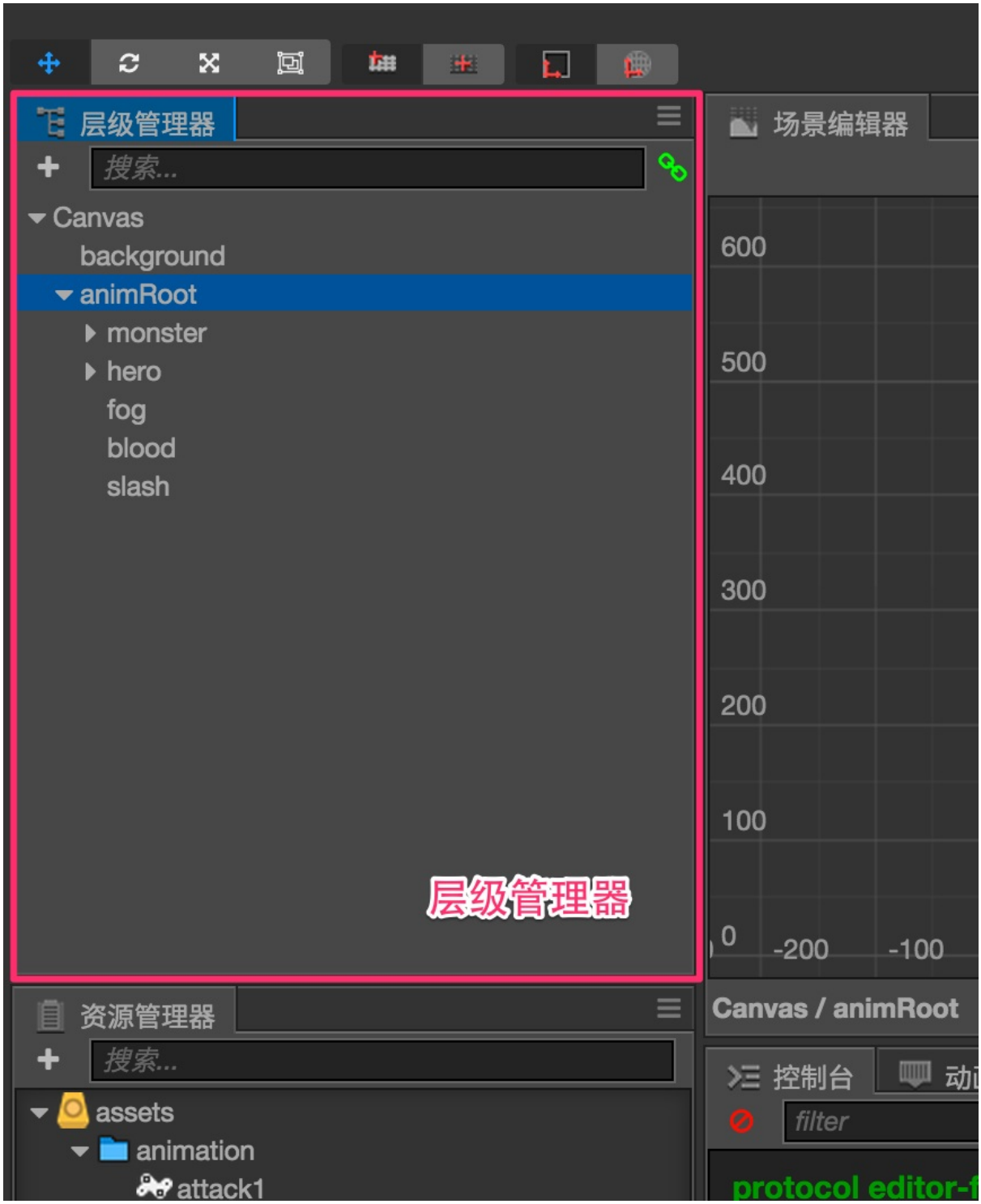


在UI元素的排版中，经常会需要使用 **矩形变换工具** 直接精确控制节点四条边的位置和长度。而对于必须保持原始图片宽高比的图像元素，通常不会使用矩形变换工具来调整尺寸。

---

继续前往层级管理器说明文档。

# Node Tree 层级管理器




**层级管理器** 中包括当前打开场景中的所有节点，不管节点是否包括可见的图像。你可以在这里选择、创建和删除节点，也可以通过拖拽一个节点到另一个上面来建立节点父子关系。

点击来选中节点，被选中的节点会以蓝底色高亮显示。当前选中的节点会在**场景编辑器**中显示蓝色边框，并更新**属性检查器**中的内容。

# 创建节点

在层级管理器中有两种方法可以创建节点：

- 点击左上角的 `+` 按钮，或右键点击鼠标并进入右键菜单中的**创建节点**子菜单。在这个子菜单中，你可以选择不同的节点类型，包括精灵（Sprite）、文字（Label）、按钮（Button）等有不同功能和表现节点。
- 从**资源管理器**中拖拽图片、字体或粒子等资源到层级管理器中。就能够用选中的资源创建出相应的图像渲染节点。

# 删除节点

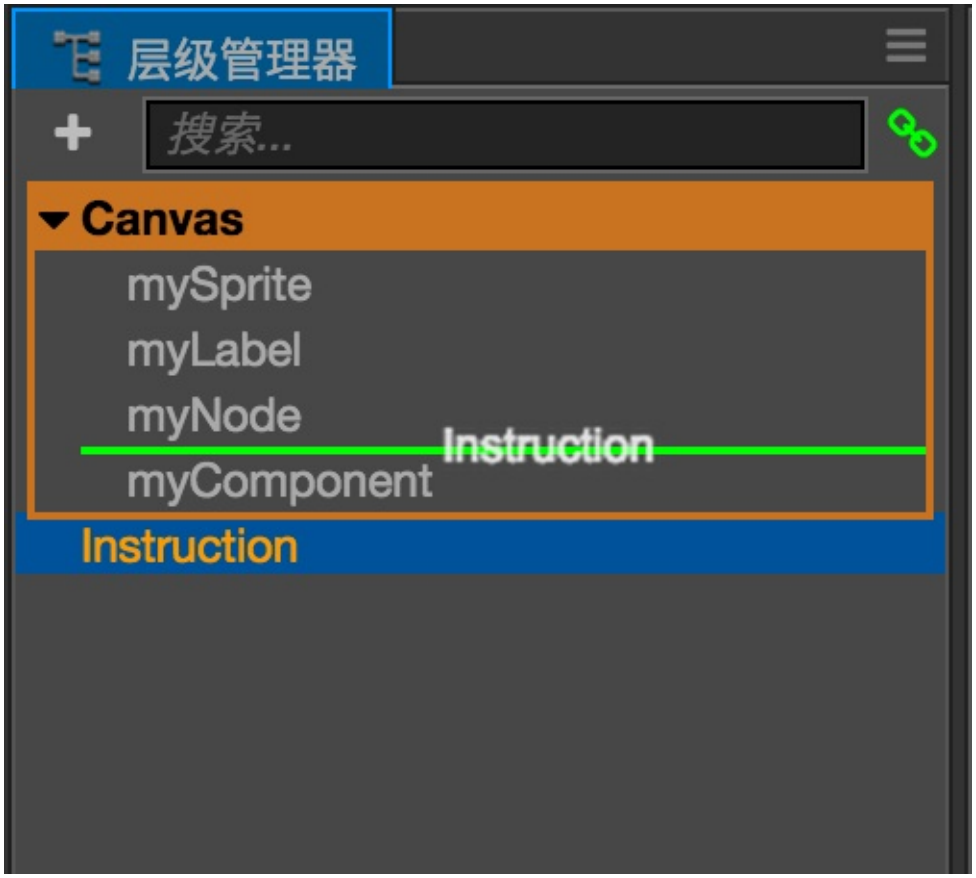
选中节点后，通过右键菜单里 `删除` 选项，或按下`Delete`（Windows）或`Cmd + Backspace`热键即可删除选中的节点。如果选中节点包括子节点，子节点也会被一起删除。

# 建立和编辑节点层级关系

将节点A拖拽到节点B上，就使节点A成为节点B的子节点。和资源管理器类似，层级管理器中也通过树状视图表示节点的层级关系。点击节点左边的三角图标，即可展开或收起子节点列表。

# 更改节点的显示顺序

除了将节点拖到另一个节点上，你还可以继续拖拽节点上下移动，来更改节点在列表中的排序。橙色的方框表示节点所属父节点的范围，绿色的线表示节点将会被插入的位置。




节点在列表中的排序决定了节点在场景中的显示次序。在层级管理器中位置越靠下的节点，在场景中的渲染就会更晚，也就会覆盖列表中位置较为靠上的节点。

# 其他操作

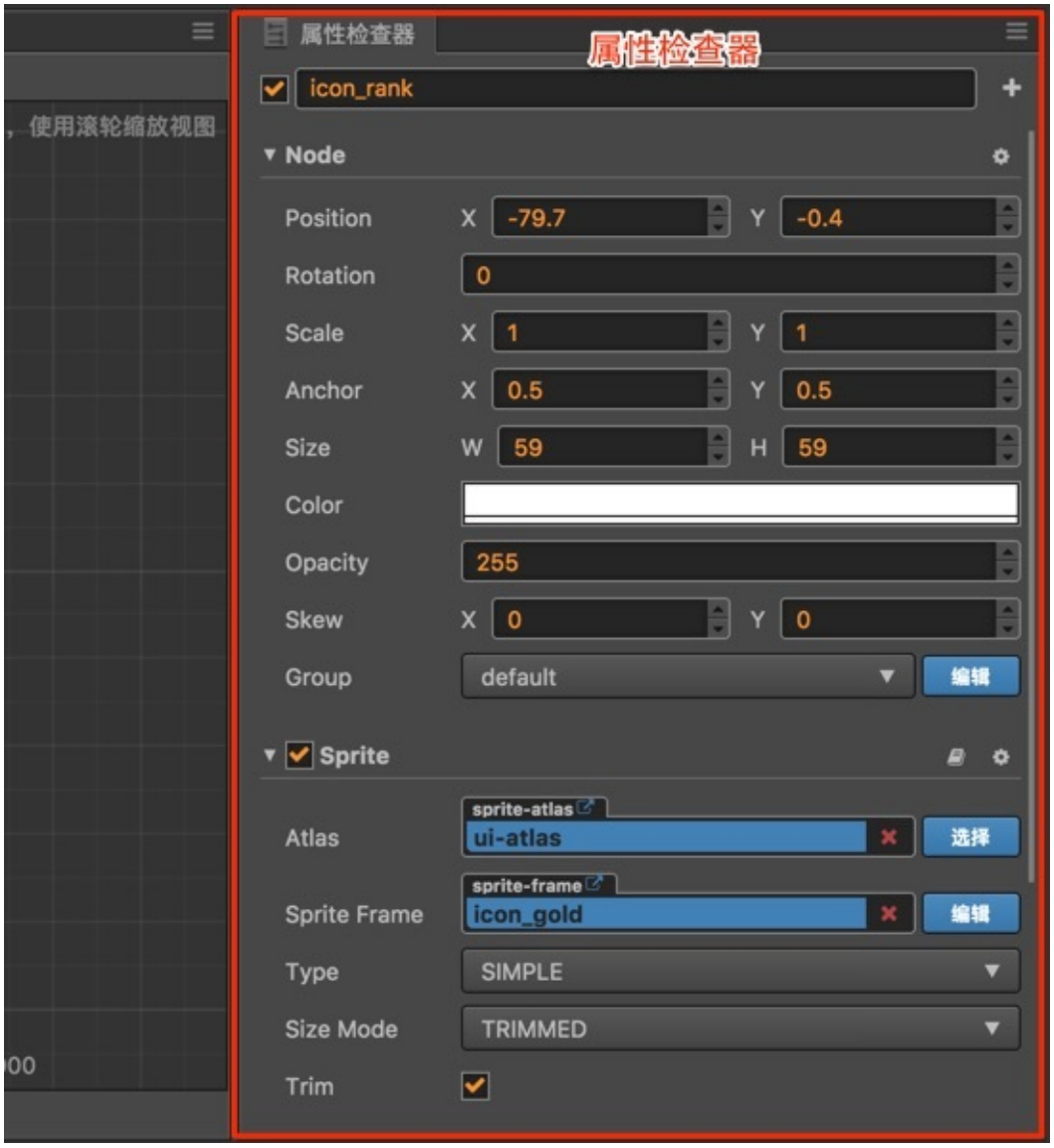
右键点击节点弹出的菜单里还包括下列操作：

- **拷贝/粘贴**：将节点复制到剪贴板上，然后可以粘贴到另外的位置，或打开另一个场景来粘贴刚才拷贝的节点。
- **复制节点**：生成和选中节点完全相同的节点副本，生成节点和选中节点在同一层级中。
- **重命名**：将节点改名。
- **在控制台显示路径**：在复杂场景中，我们有时候需要获取节点的完整层级路径，以便在脚本运行时访问该节点。点击这个选项，就可以在**控制台**中看到节点的路径。

---

继续前往属性检查器说明文档。

# 属性检查器（Properties）




**属性检查器**是我们查看并编辑当前选中节点和组件属性的工作区域。在**场景编辑器**或**层级管理器**中选中节点，就会在**属性检查器**中显示该节点的属性和节点上所有组件的属性以供您查询和编辑。

属性检查器面板从上到下依次是：

- 节点激活开关和节点名称
- 节点属性
- 组件属性

## 节点名称和激活开关

左上角的复选框表示节点的激活状态，使用节点处于非激活状态时，节点上所有图像渲染相关的组件都会被关闭，整个节点包括子节点就会被有效的隐藏。

节点激活开关右边显示的是节点的名称，和**层级管理器**中的节点显示名称一致。

# 节点属性

**属性检查器**接下来会显示节点的属性，节点的属性排列在 `Node` 标题的下面，点击 `Node` 可以将节点的属性折叠或展开。

节点的属性除了位置（Position）、旋转（Rotation）、缩放（Scale）、尺寸（Size）等变换属性以外，还包括锚点（Anchor）、颜色（Color）、不透明度（Opacity）。修改节点的属性通常可以立刻在场景编辑器中看到节点的外观或位置变化。

更多关于节点属性编辑的细节，请阅读坐标系和变换一节。

# 组件属性

节点属性下面，会列出节点上挂载的所有组件和组件的属性。和节点属性一样，点击组件的名称就会切换该组件属性的折叠/展开状态。在节点上挂载了很多组件的情况下，可以通过折叠不常修改的组件属性来获得更大的工作区域。

用户通过脚本创建的组件，其属性是由脚本声明的。不同类型的属性在**属性检查器**中有不同的控件外观和编辑方式。我们将在声明属性一节中详细介绍属性的定义方法。

# 编辑属性

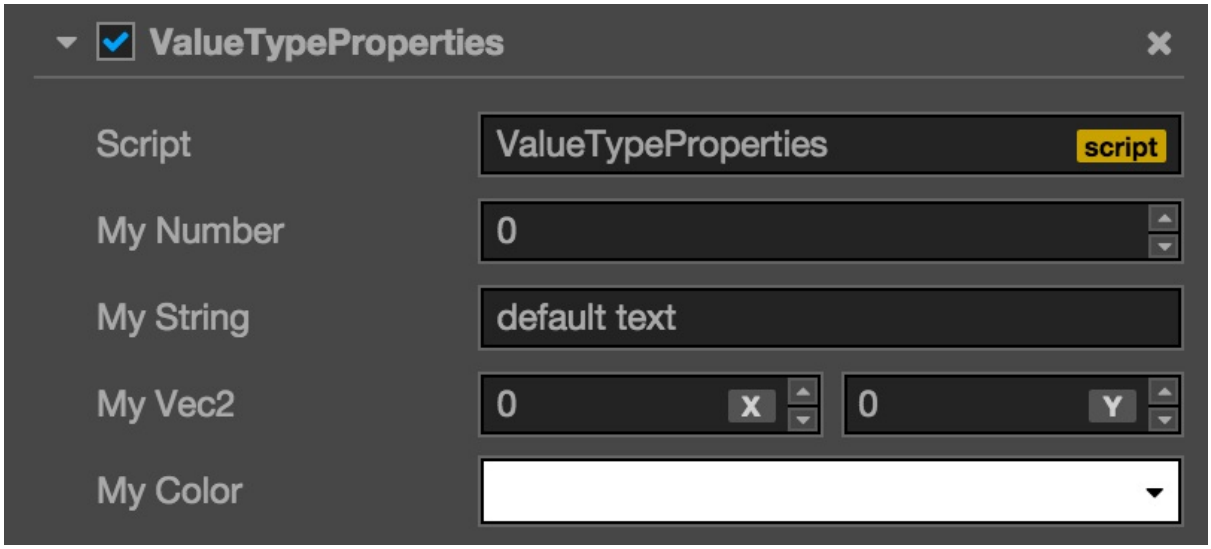
**属性**是组件脚本中声明的公开并可被序列化存储在场景和动画数据中的变量。通过**属性检查器**我们可以快捷的修改属性设置，达到不需要编程就可以调整游戏数据和玩法的目的。

通常可以根据变量使用内存位置不同将属性分为**值类型**和**引用类型**两大类。

## 值类型属性

**值类型**包括数字、字符串、枚举等简单的占用很少内存的变量类型：

- 数值（Number）：可以直接使用键盘输入，也可以按输入框旁边的上下箭头逐步增减属性值。
- 向量（Vec2）：向量的控件是两个数值输入组合在一起，并且输入框上会以 `x`，`y` 标识每个数值对应的子属性名。
- 字符串（String）：直接在文本框里用键盘输入字符串，字符串输入控件分为单行和多行两种，多行文本框可以按回车换行。
- 布尔（Boolean）：以复选框的形式来编辑，选中状态表示属性值为 `true`，非选中状态表示 `false`。
- 枚举（Enum）：以下拉菜单的形式编辑，点击枚举菜单，然后从弹出的菜单列表里选择一项，即可完成枚举值的修改。
- 颜色（Color）：点击颜色属性预览框，会弹出**颜色选择器**窗口，在这个窗口里可以用鼠标直接点选需要的颜色，或在下面的 RGBA 颜色输入框中直接输入指定的颜色。点击**颜色选择器**窗口以外的任何位置会关闭窗口并以最后选定的颜色作为属性颜色。



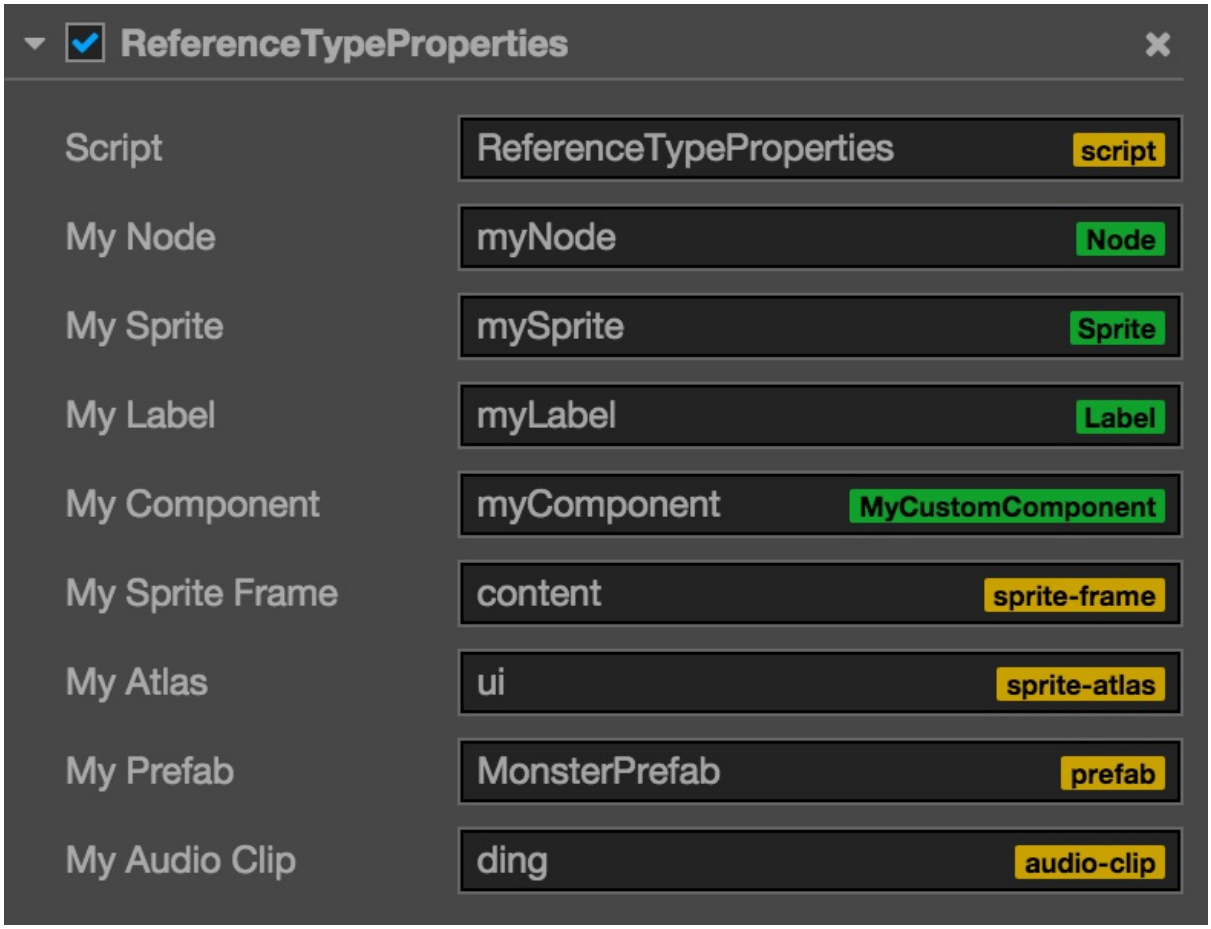

## 引用类型属性

**引用类型**包括更复杂的对象，比如节点、组件或资源。和值类型各式各样的编辑方式不同，引用类型通常只有一种编辑方式：拖拽节点或资源到属性栏中。

引用类型的属性在初始化后会显示 `None`，因为无法通过脚本为引用类型的属性设置初始值。这时可以根据属性的类型将相应类型的节点或资源拖拽上去，即可完成引用赋值。

需要拖拽节点来赋值的属性栏上会显示绿色的标签，标签上可能会显示 `Node`，表示任意节点都可以拖拽上去，或者标签显示组件名如 `Sprite`， `Animation` 等，这时需要拖拽挂载了相应组件的节点才行。

需要拖拽资源赋值的属性栏上会显示黄色的标签，标签上显示的是资源的类型，如 `sprite-frame`， `prefab`， `font` 等。只要从**资源管理器**中拖拽相应类型的资源过来就可以完成赋值。




值得注意的是，脚本文件也是一种资源，所以上图中最上面表示组件使用的脚本资源引用属性也是用黄色标签表示的。

---

继续前往控制台说明文档。

# 控件库


**控件库** 是一个非常简单直接的可视化控件仓库，您可以将这里列出的控件拖拽到 **场景编辑器** 或 **层级管理器** 中，快速完成预设控件的创建。

使用默认窗口布局时，控件库会默认显示在编辑器中，如果您之前使用的编辑器布局中没有包括控件库，您可以通过主菜单的 `面板->控件库` 来打开控件库，并拖拽它到编辑器中您希望的任意位置。

目前 **控件库** 包括两个类别，由两个分页栏表示：

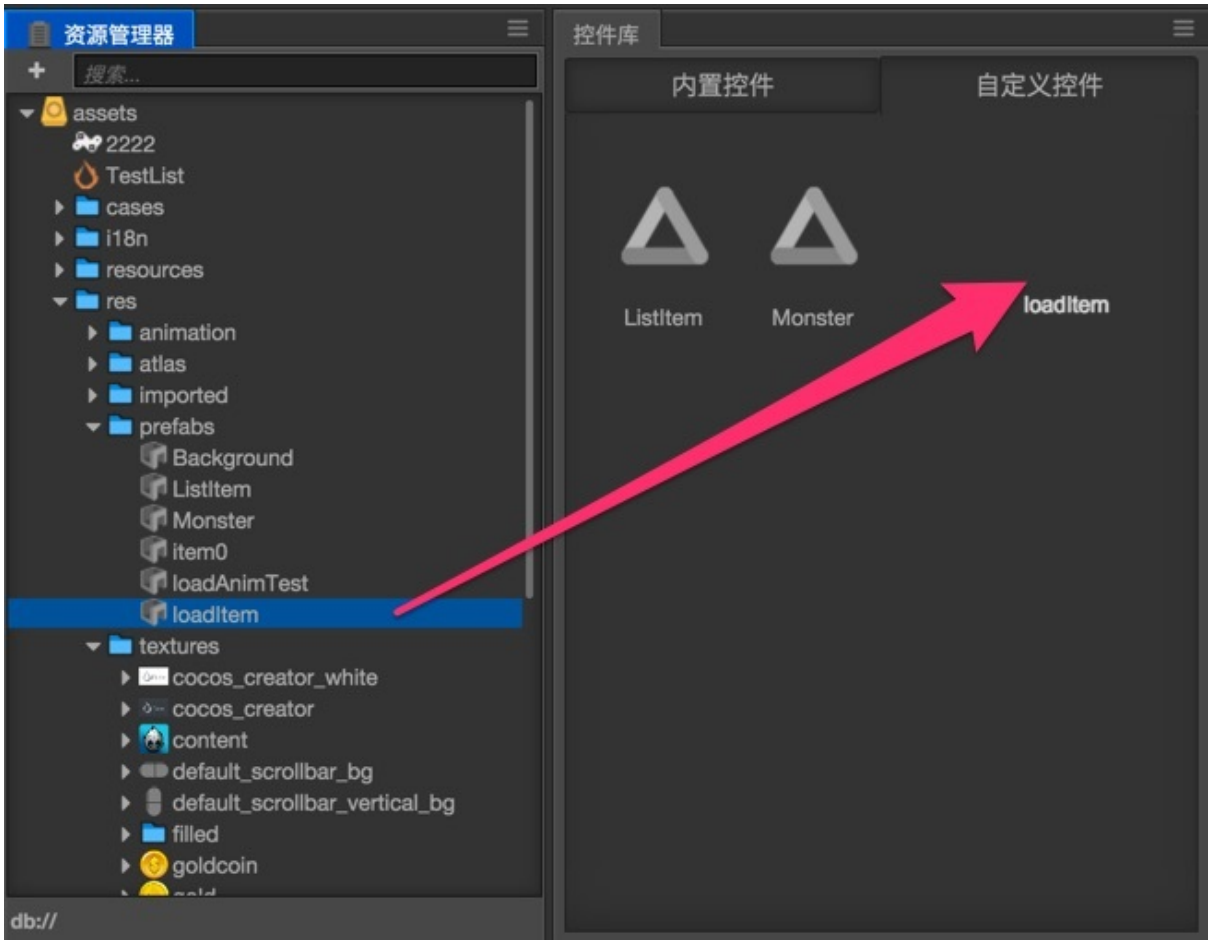
# 内置控件

如上图所示，这里列出了所有编辑器内置的预设节点，通过拖拽这些控件到场景中，您可以快速生成包括默认资源的精灵（Sprite）、包含背景图和文字标题的按钮（Button）以及已经配置好内容和滚动条的滚动视图（ScrollView）等。

**控件库** 里包含的控件内容和主菜单中的 **节点** 菜单里可以添加的预设节点是一致的，但通过控件库创建新节点更加方便快捷。

后续随着更多功能的添加，控件库里的节点类型也会不断补充增加。

# 自定义控件



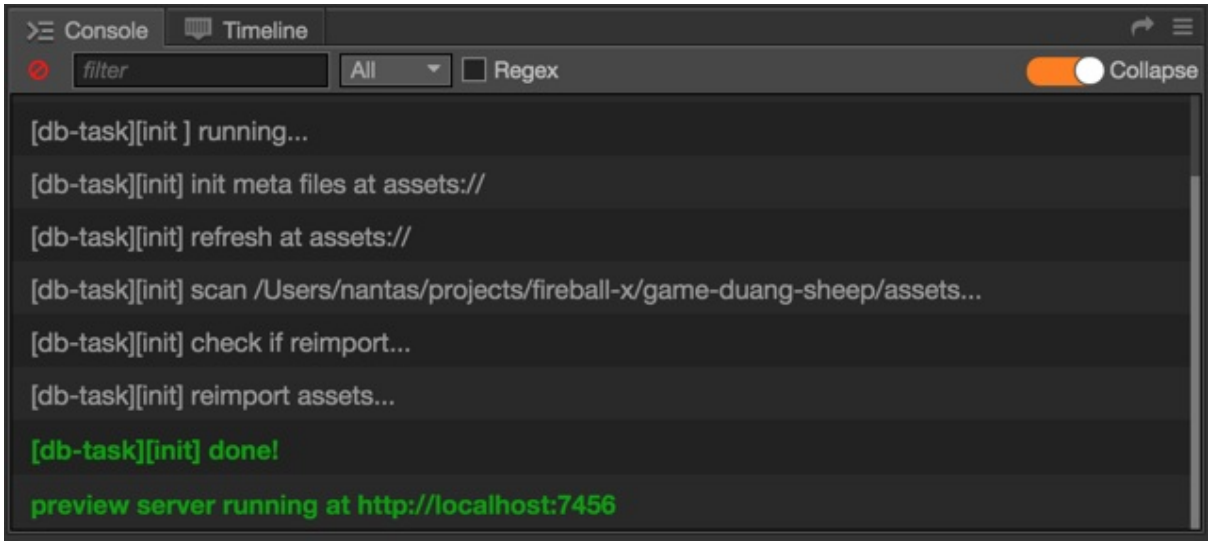

这个分页栏中可以收集用户自己建立的预制资源（Prefab），方便重复多次创建和使用。

要添加自定义的预制控件，只需要从 **资源管理器** 中拖拽相应的预制资源（Prefab）到自定义控件分页，即可完成创建。

右键点击自定义控件中的元素，可以选择重命名或从控件库中删除该控件。

之后您就可以像使用内置控件一样，用拖拽的方式在场景中创建您自定义的控件了！

# 控制台（Console）




**控制台**会显示报错、警告或其他 Cocos Creator 编辑器和引擎生成的日志信息。不同重要级别的信息会以不同颜色显示：

- **日志（Log）**：灰色文字，通常用来显示正在进行的操作。
- **提示（Info）**：蓝色文字，用来显示重要提示信息。
- **成功（Success）**：绿色文字，表示当前执行的操作已成功完成。
- **警告（Warn）**：黄色文字，用来提示用户最好进行处理的异常情况，但不处理也不会影响运行。
- **报错（Error）**：红色文字，表示出现了严重错误，必须解决才能进行下一步操作或运行游戏。

在控制台中信息量很大时，你可以通过控制台中的控件来有效的过滤信息，这些操作包括：

- **清除**  清除控制台面板中的所有当前信息。
- **过滤输入** 

根据输入的文本过滤控制台中的信息，如果勾选了旁边的 `Regex`，输入的文本会被当做正则表达式来匹配文本。
- **信息级别** 

这个下拉菜单里可以选择某一种信息级别，从日志级到报错级，选择后控制台中将只显示指定级别的信息。默认的选项 `All` 表示所有级别的信息都会显示。
- **合并同类信息** 

该选项处于激活状态时，相同而重复的信息会被合并成一条，在信息旁边会以黄色数字提示有多少条同类信息被合并了。

---

继续前往工具栏说明文档。

# 偏好设置

**偏好设置** 面板中里提供各种编辑器个性化的全局设置，要打开 **偏好设置** 窗口，请选择主菜单的 `CocosCreator->偏好设置` 。

**偏好设置** 由几个不同的分页组成，将各种设置分为以下几类。修改设置之后点击 **保存并退出** 按钮，设置才会生效。

# 常规


## 编辑器语言

可以选择中文或英文，修改语言设置后要重新启动 Cocos Creator 才能生效。

## 层级管理器节点树状态

切换层级管理器节点树默认状态是展开或折叠所有子节点，可以以下三种中选择

- 全部展开
- 全部折叠
- 记住上一次状态

## 选择本机 IP 地址

用户可以在本机有多个 IP 地址的情况下，手动选择其中之一作为预览时的默认地址和二维码地址。这里会列出所有本机的 IP，您也可以选择 **自动** 让编辑器帮您自动挑选一个 IP。

## 构建日志是否在控制台显示

这一项在选中状态时，构建发布原生项目的过程日志会直接显示在 **控制台** 面板里。非选中状态时，构建发布原生项目的日志会保存在 `%USER/.CocosCreator/logs/native.log`，您也可以通过 **控制台** 左上角的日志按钮的 **Cocos Console 日志** 选项打开这份文件。

## 数值调节钮步长

在 **属性检查器** 里，所有数值属性输入框的旁边都有一组上下箭头，可以用于步进输入的数值：


当鼠标悬浮在数值属性的名称附近时，光标会变成  这样的形状，然后左右拖动鼠标，也可以按照一定的步进幅度连续增加或减小数值。

以上两种修改数值的方式，默认的步长都是 `0.1`，而偏好设置里 **数值调节钮步长** 这里设置的就是每次点击步进按钮或拖拽鼠标时数值变化的步长幅度。举例来说，如果您在脚本中使用的数字以整数为主，就可以把这个步长设置为 `1`，可以更方便的进行调节。

**注意**：修改步长后要刷新编辑器窗口（按 Ctrl/Cmd + R），设置的步长才会生效。

## meta 文件备份时显示确认

在 meta 文件所属的资源丢失时，是否弹出对话框提示备份或删除 meta 文件。如果选择备份，可以在稍后手动恢复资源，并将 meta 文件手动拷贝回项目 `assets` 目录，放止资源相关的重要设置（如场景、prefab）丢失。

## 导入图片自动裁剪

导入图片时，是否自动裁剪掉图片的透明像素。不管默认选择如何，导入图片之后可以在图片资源上手动设置裁剪选项。

## 默认开启 prefab 自动同步模式

新建 prefab 时，是否自动开启 prefab 资源上的「自动同步」选项。开启自动同步后，保存 prefab 资源修改时会自动同步场景中所有该 prefab 的实例。

# 数据编辑


这一类别用来设置脚本和资源的默认打开方式。

### 自动编译脚本

是否自动监测项目中脚本文件的变化，并自动触发编译。如果关闭自动编译脚本选项后，可以手动通过菜单「开发者->手动编译脚本」或按 F7 来编译。

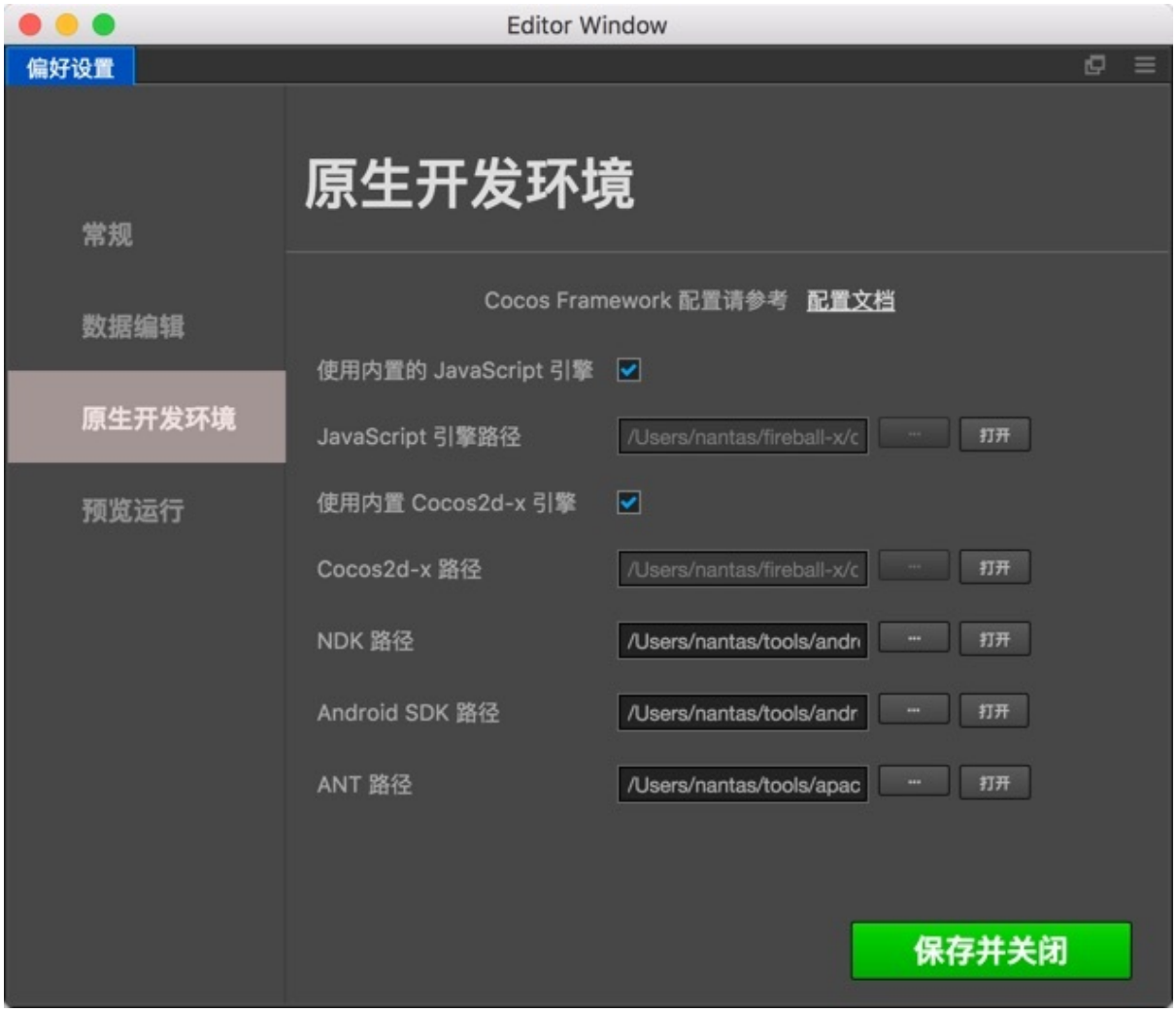
### 外部脚本编辑器

可以选用内置代码编辑器或任意外部文本编辑工具的可执行文件，作为在 **资源管理器** 里双击脚本文件时的打开方式。您可以在下拉菜单中选择 **内置**，或点击 **浏览** 按钮选择偏好的文本编辑器的可执行文件。

### 外部图片编辑器

和上面的选项类似，这里用来设置在 **资源管理器** 中双击图片文件时，默认打开图片用的应用程序路径。

## 原生开发环境




这个分类用于设置构建发布到原生平台（iOS, Android, Mac, Windows）时，所需的开发环境路径。

## 使用内置 JavaScript 引擎

是否使用 Cocos Creator 安装路径下自带的 `engine` 路径作为 JavaScript 引擎路径。这个引擎用于 **场景编辑器** 里场景的渲染，内置组件的声明和其他 Web 环境下的引擎模块。

## JavaScript 引擎路径

除了使用自带的 `engine`，您也可以前往 https://github.com/cocos-creator/engine 来克隆或 fork 一份引擎到本地的任意位置进行定制，然后取消勾选 **使用内置 JavaScript 引擎**，然后设置 **JavaScript 引擎路径** 到您定制好的引擎路径，就可以在编辑器中使用这份定制后的引擎了。

## 使用内置 Cocos2d-x 引擎

是否使用 Cocos Creator 安装路径下自带的 `cocos2d-x` 路径作为 cocos2d-x C++ 引擎路径。这个引擎用于 **构建发布** 时所有原生平台（iOS, Android, Mac, Windows）的工程构建和编译。

## Cocos2d-x 路径

取消上一项 **使用内置 cocos2d-x 引擎** 的选择后，就可以手动指定 cocos2d-x 路径了。注意这里使用的 cocos2d-x 引擎必须从 https://github.com/cocos-creator/cocos2d-x-lite 或该仓库的 fork 下载。

### NDK 路径

设置 NDK 路径，详情见 安装配置原生开发环境。

### Android SDK 路径

设置 Android SDK 路径，详情见 安装配置原生开发环境。

### ANT 路径

设置 ANT 路径，详情见 安装配置原生开发环境。

## 预览运行


使用主窗口正上方的 **运行预览** 按钮时，可以设置的各种选项。

### 自动刷新已启动的预览

当已经有浏览器或模拟器在运行你的场景时，保存场景或重新编译脚本后是否应该刷新这些正在预览的设备。

### 预览使用浏览器

可以从下拉菜单中选择系统默认的浏览器，或点击 **浏览** 按钮手动指定一个浏览器的路径。

## 模拟器路径

从 v1.1.0 版开始， Cocos Creator 中使用的 Cocos 模拟器会放置在 cocos2d-x 引擎路径下。在使用定制版引擎时，需要自己编译模拟器到引擎路径下。点击 **打开** 按钮可以在文件系统中打开当前指定的模拟器路径，方便调试时定位。

## 模拟器横竖屏设置

指定模拟器运行时是横屏还是竖屏。

## 模拟器分辨率设置

从预设的设备分辨率中选择一个作为模拟器的分辨率。

## 自定义分辨率设置

如果预设的分辨率不能满足要求，您可以手动输入屏幕宽高来设置模拟器分辨率。

# 项目设置

**项目设置** 面板通过主菜单的 「项目->项目设置...」 菜单打开，这里包括所有特定项目相关的设置。这些设置会保存在项目的 `settings/project.json` 文件里。如果需要在不同开发者之间同步项目设置，请将 `settings` 目录加入到版本控制。

## 分组管理


目前项目设置中的分组管理主要为 碰撞体系统 提供分组支持，详情请参考 碰撞分组管理。

## 模块设置

这里的设置是针对发布 Web 版游戏时引擎中使用的模块进行裁剪，达到减小发布版引擎包体的效果。在列表中选中的模块在打包时将被引擎包括，未选中的模块会被裁剪掉。

在这里设置裁剪能够大幅度的减小引擎包体，建议打包后进行完整的测试，避免在场景和脚本中使用裁剪掉的模块。

# 预览运行

**预览运行** 分页提供的选项和 偏好设置面板 里的 **预览运行** 分页类似，用于设置初始预览场景、分辨率等等，但只对当前项目生效。

## 初始预览场景

指定按下 **预览运行** 按钮时，会打开项目中哪个场景。如果设置为 **当前打开场景**，就会运行当前正在编辑的场景，此外也可以设置成一个固定的场景（比如项目总是需要从登录场景开始游戏）。

## 设计分辨率，适配屏幕宽度/高度

用于规定在新建场景或 **Canvas 组件** 时，Canvas 中默认的设计分辨率数值，以及 **Fit Height**, **Fit Width** 选项。

## 模拟器设置类型

用于设置模拟器预览分辨率和屏幕朝向，当这个选项设为 **全局** 时，会使用 **偏好设置** 里的模拟器分辨率和屏幕朝向设置。当设为 **项目** 时，会显示以下模拟器设置：

- 模拟器横竖屏设置
- 模拟器分辨率设置
- 模拟器自定义分辨率

以上选项和 **偏好设置** 面板中的设置方法一致。

# 主菜单

## Cocos Creator

包括软件信息，偏好设置，窗口控制等功能。

- **关于 Cocos Creator**：显示 Cocos Creator 的版本和版权信息。
- **偏好设置**：打开 偏好设置 面板，设置编辑器的个性化选项。
- **隐藏 Cocos Creator (`Ctrl/Command + H`)**：（Mac 专属）隐藏编辑器窗口。
- **隐藏其他应用 (`Shift + Ctrl/Command + H`)**：（Mac 专属）隐藏 Cocos Creator 之外的其他应用窗口。
- **显示全部**：（Mac 专属）显示所有窗口。
- **最小化（`Ctrl/Command + M`）**：最小化 Cocos Creator 编辑器窗口。
- **退出 (`Ctrl/Command + Q`)**：退出编辑器。

## 文件

包括场景文件的打开和保存，从其他项目导入场景和资源的功能。

- **打开项目...**：关闭当前打开的项目，并打开 Dashboard 的 **最近打开项目** 分页。
- **新建场景 (`Ctrl/Command + N`)**：关闭当前场景并创建一个新场景，新创建的场景需要手动保存才会添加到项目路径下。
- **保存场景 (`Ctrl/Command + S`)**：保存当前正在编辑的场景，如果是使用 **新建场景** 菜单项创建的场景，在第一次保存时会弹出对话框，选择场景文件保存的位置和文件名。场景文件以 *.fire 作为扩展名。
- **导入项目**：从其他场景和 UI 编辑工具中导入场景和项目资源，详情请参考 导入其他编辑器项目
  - **导入 Cocos Studio 项目 (*.ccs)**
  - **导入 Cocos Builder 项目 (*.ccbproj)**

## 编辑

包括撤销重做、复制粘贴等常用编辑功能。

- **撤销 (`Ctrl/Command + Z`)**：撤销上一次对场景的修改。
- **重做 (`Shift + Ctrl/Command + Z`)**：重新执行上一次撤销的对场景的修改。
- **拷贝 (`Ctrl/Command + C`)**：复制当前选中的节点或字符到剪贴板。
- **粘贴 (`Ctrl/Command + V`)**：粘贴剪贴板中的内容到场景或属性输入框中。
- **选择全部 (`Ctrl/Command + A`)**：选择场景中所有的节点。

## 节点

通过这个菜单创建节点，并控制节点到预制的转化。

- **关联节点到预制**：同时选中场景中的一个节点和资源管理器中的一个预制（prefab），然后选中此菜单项，即可关联选中的节点和预制。
- **还原成普通节点**：选中场景中一个预制节点，执行此命令会将预制节点转化成普通节点。
- **创建空节点**：在场景中创建一个空节点，如果执行命令前场景中已经选中了节点，新建的节点会成为选中节点的子节点。
- **创建渲染节点**：创建预设好的包含渲染组件的节点，关于渲染组件的使用方法请参考 图像和渲染 一章。

- **创建 UI 节点**：创建预设好的包含 UI 组件的节点，详情请参考 UI 系统 一章。

# 组件

通过这个菜单在当前选中的节点上添加各类组件。

- **添加碰撞组件**：详情请参考 碰撞组件 一节。
- **添加其他组件**：包括动画、音源、拖尾等组件。
- **添加渲染组件**：详情请参考 图像和渲染 一章。
- **添加用户脚本组件**：这里可以添加用户在项目中创建的脚本组件。
- **添加 UI 组件**：详情请参考 UI 系统 一章。

# 项目

运行、构建项目，以及项目专用个性化配置。

- **运行预览 (`Ctrl/Command + P`)**：在浏览器或模拟器中运行项目。
- **刷新已运行的预览 (`Shift + Ctrl/Command + P`)**：刷新已经打开的预览窗口。
- **构建发布... (`Shift + Ctrl/Command + B`)**：打开 构建发布 面板。
- **项目设置...**：打开 项目设置 面板。

# 面板

- **资源管理器 (`Ctrl/Command + 2`**：打开 资源管理器 面板。
- **层级管理器 (`Ctrl/Command + 4`**：打开 层级管理器 面板。
- **属性检查器 (`Ctrl/Command + 3`**：打开 属性检查器 面板。
- **场景编辑器 (`Ctrl/Command + 1`**：打开 场景编辑器 面板。
- **控件库 (`Ctrl/Command + 5`**：打开 控件库 面板。
- **动画编辑器 (`Ctrl/Command + 6`**：打开 动画编辑器 面板。
- **控制台 (`Ctrl/Command + 0`**：打开 控制台 面板。

# 布局

从预设编辑器布局中选择一个。

- **默认布局**
- **竖屏布局**
- **经典布局**

# 扩展

和扩展插件相关的菜单项，详情请阅读 编辑器扩展 一章。

- **创建新扩展插件...**
  - **全局扩展**
  - **项目专用扩展**
- **扩展商店...**：打开扩展商店，下载官方和社区提供的扩展插件。
- **AnySDK**：打开 AnySDK 打包界面，详见 AnySDK。

# 开发者

脚本和编辑器扩展开发相关的菜单功能。

- **VS Code 工作流**：VS Code 代码编辑器的工作环境相关功能，详情请阅读 代码编辑环境配置 一节。
  - **更新 VS Code 智能提示数据**
  - **安装 VS Code 扩展插件**
  - **添加 Chrome Debug 配置**
- **重新加载界面**：重新加载编辑器界面。
- **手动编译脚本**：触发脚本编译流程。
- **检视页面元素**：在 Chrome 开发者工具里检视编辑器界面元素
- **开发者工具**：打开 Chrome 开发者工具，用于编辑器界面扩展的开发

# 帮助

- **搜索**：（Mac 专属）搜索特定菜单项。
- **使用手册**：在浏览器打开手册文档。
- **API 文档**：在浏览器打开 API 参考文档。
- **论坛**：在浏览器打开 Cocos Creator 论坛。
- **登出**：登出帐号。

# 工具栏

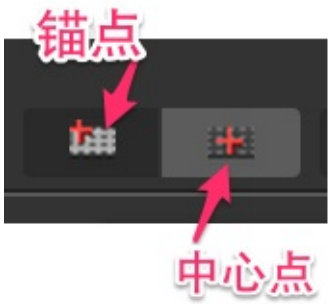

**工具栏** 位于编辑器主窗口的正上方，包含了五组控制按钮或信息，用来为特定面板提供编辑功能或方便我们实施工作流。

## 选择变换工具

为 **场景编辑器** 提供编辑节点变换属性（位置、旋转、缩放、尺寸）的功能，详情请阅读使用变换工具布置节点

## 变换工具显示模式

使用以下两组按钮控制 **场景编辑器** 中 **变换工具** 的显示模式。




位置模式：

- 锚点：变换工具将显示在节点 **锚点（Anchor）** 所在位置。
- 中心点：变换工具将显示在节点中心点所在位置（受约束框大小影响）。


旋转模式：

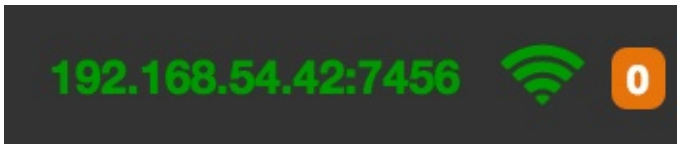
- 本地：变换工具的旋转（手柄方向）将和节点的 **旋转（Rotation）** 属性保持一致。
- 世界：变换工具的旋转保持不变，x 轴手柄和 y 轴手柄和世界坐标系方向保持一致。

## 运行预览游戏


包括两个按钮：

- 运行预览：点击后在浏览器中运行当前编辑的场景。
- 刷新设备：在所有正在连接本机预览游戏的设备上重新加载当前场景（包括本机浏览器和其他链接本机的移动端设备）。

## 预览地址




这里显示运行 Cocos Creator 的桌面电脑的局域网地址，连接同一局域网的移动设备可以访问这个地址来预览和调试游戏。

## 打开项目文件夹


在操作系统的文件管理器（Explorer 或 Finder）打开项目所在的文件夹。

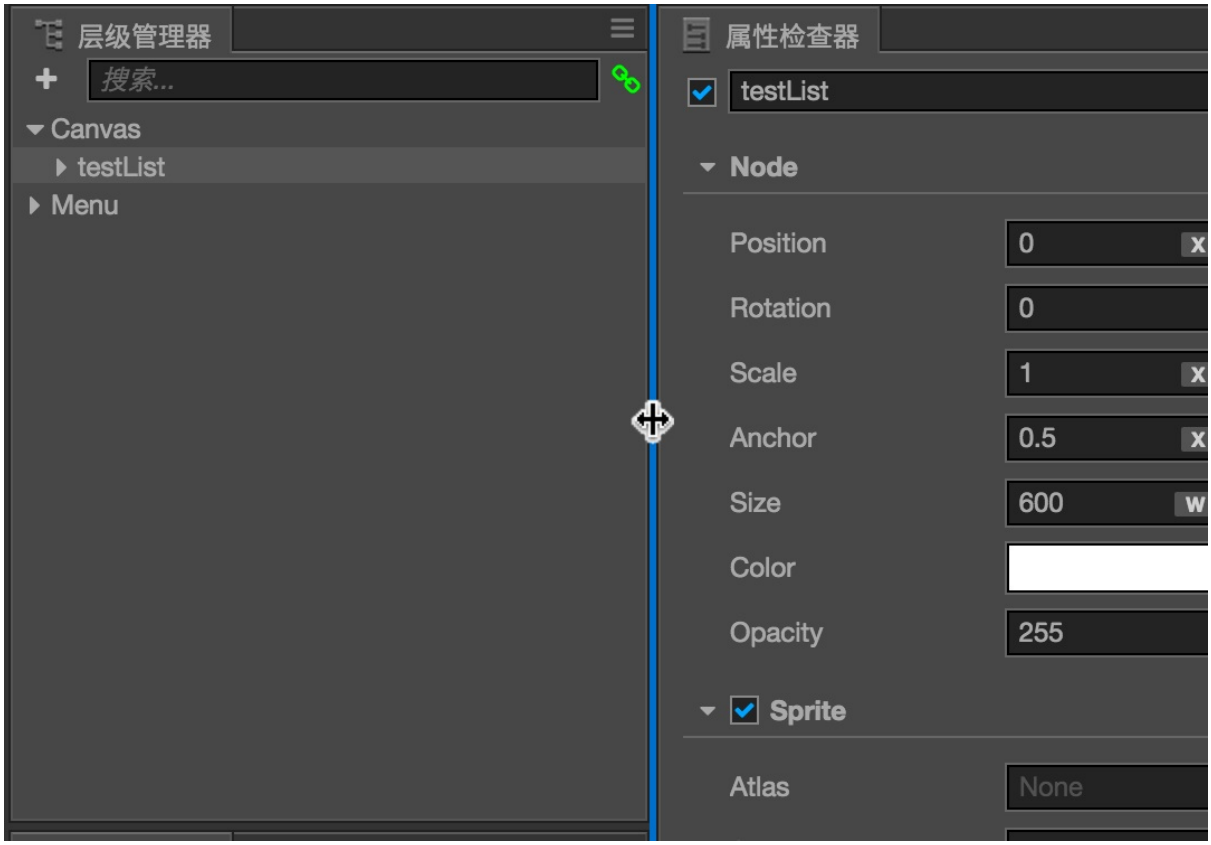
# 编辑器布局

编辑器布局是指 Cocos Creator 里各个面板的位置、大小和层叠情况。

选择主菜单里的 `布局` 菜单，可以从预设的几种编辑器面板布局中选择最适合当前项目的。在预设布局的基础上，您也可以继续对各个面板的位置和大小进行调节。对布局的修改会自动保存在项目所在文件夹下的 `local/layout.windows.json` 文件中。

## 调整面板大小

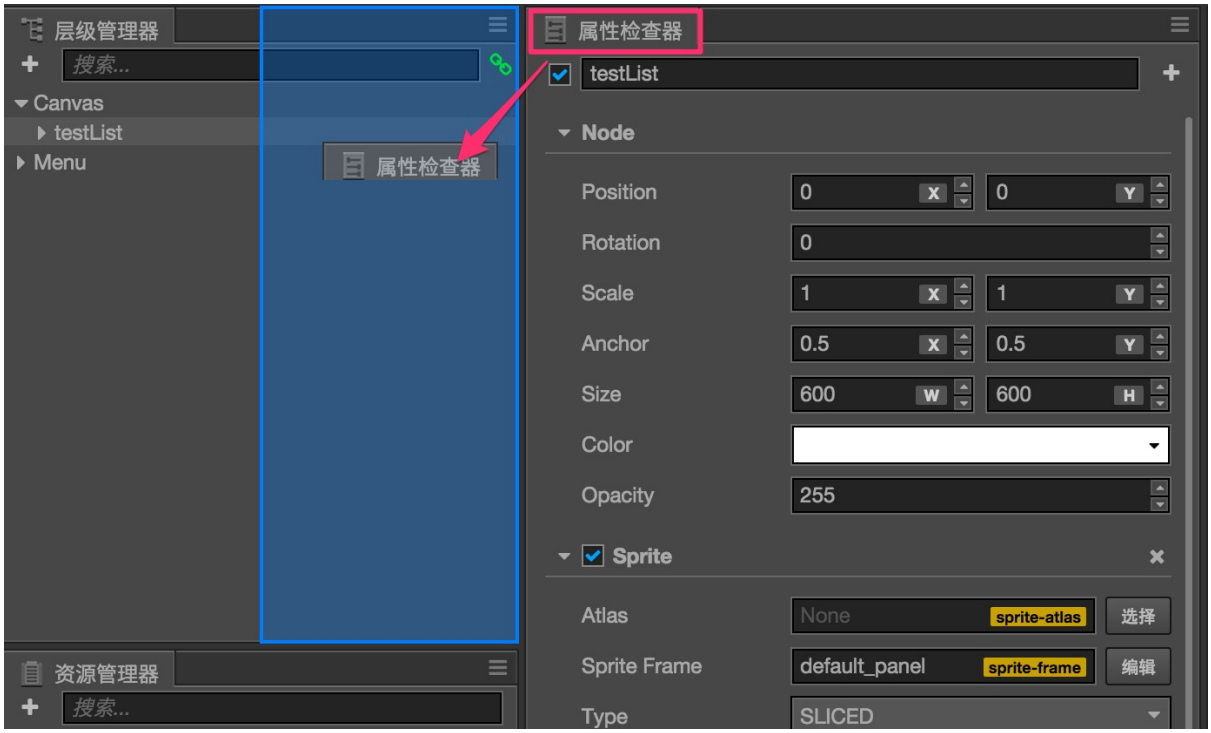
将鼠标悬浮到两个面板之间的边界线上，看到鼠标指针发生变化后，就可以按下鼠标拖动来修改相邻两个面板的大小。




部分面板设置了最小尺寸，当拖拽到最小尺寸限度后就无法再继续缩小面板了。

## 移动面板

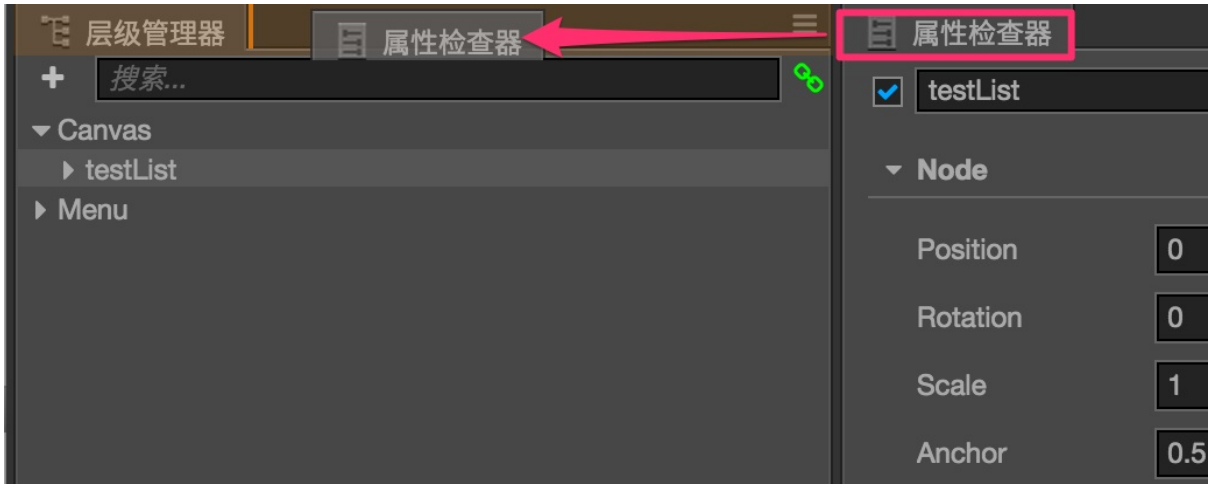
点击面板的标签栏并拖拽，可以将面板整个移动到编辑器窗口中的任意位置。下图中红框表示可拖拽的标签栏区域，箭头表示拖拽方向：




移动面板的过程中，蓝色半透明的方框会指示松开鼠标后面板将会被放置的位置。

# 层叠面板

除了移动面板位置，拖拽标签栏的时候还可以移动鼠标到另一个面板的标签栏区域：




在目标面板的标签栏出现橙色显示时松开鼠标，就能够将两个面板层叠在一起，同时只能显示一个面板：

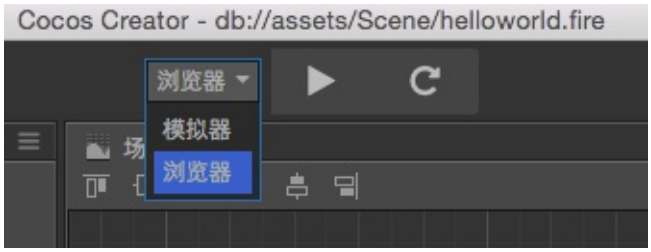

层叠面板在桌面分辨率不足，或排布使用率较低的面板时非常实用。层叠中的面板可以随时拖拽出来，恢复永远在最上的显示。

# 预览和构建

在使用主要编辑器面板进行资源导入、场景搭建、组件配置、属性调整之后，我们接下来就可以通过预览和构建来看到游戏在 Web 或原生平台运行的效果了。

# 选择预览平台

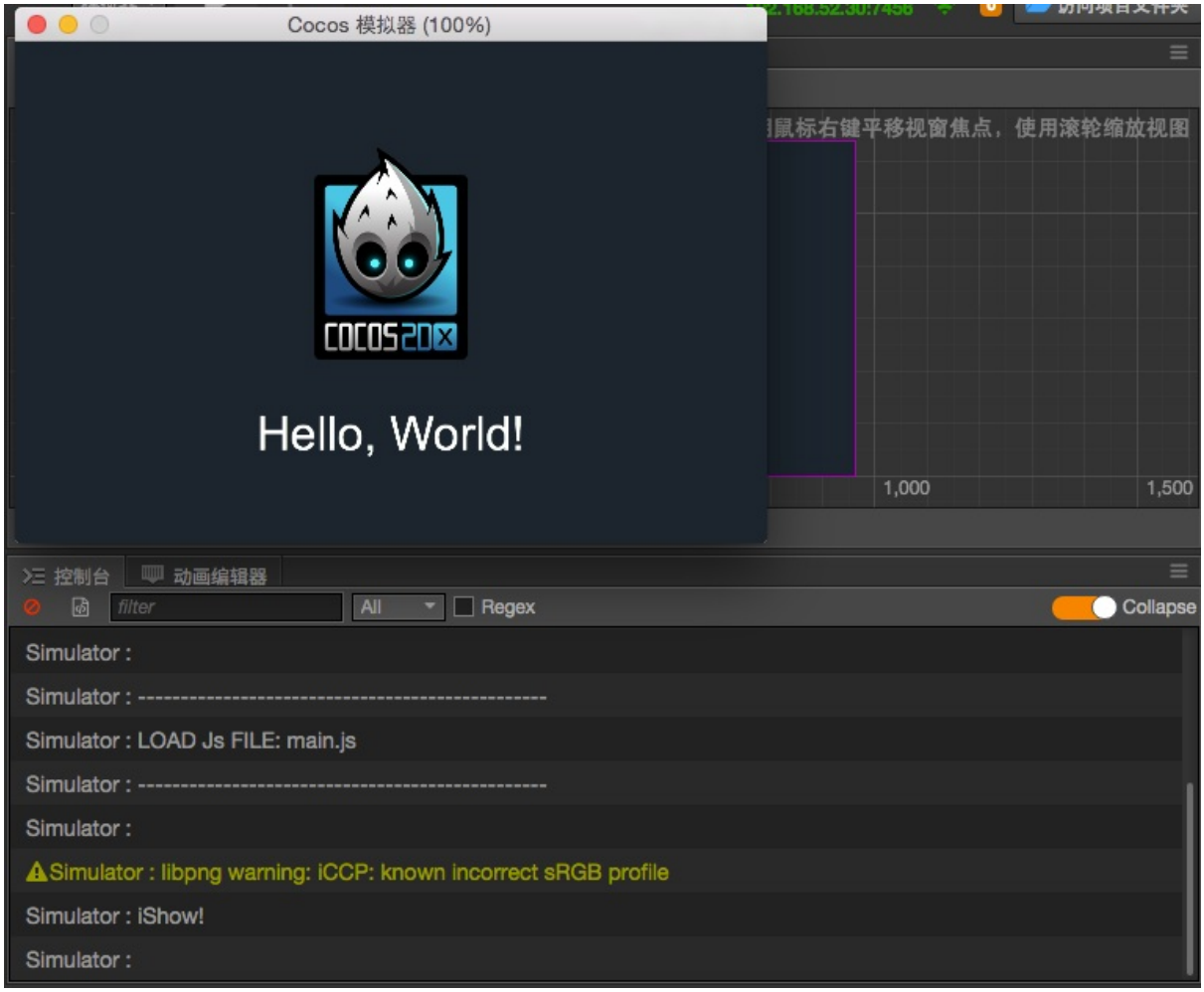
在游戏开发过程中我们可以随时点击编辑器窗口正上方的 **预览** 按钮，来看到游戏运行的实际情况。从 **预览** 按钮左边的下拉菜单我们可以从 **模拟器** 和 **浏览器** 中选择预览平台。




注意必须有当前打开的场景才能预览游戏内容，在没有打开任何场景，或者新建了一个空场景的情况下预览是看不到任何内容的。

## 模拟器

选择 **模拟器** 后运行预览，将会使用 Cocos Simulator（桌面模拟器）运行当前的游戏场景。



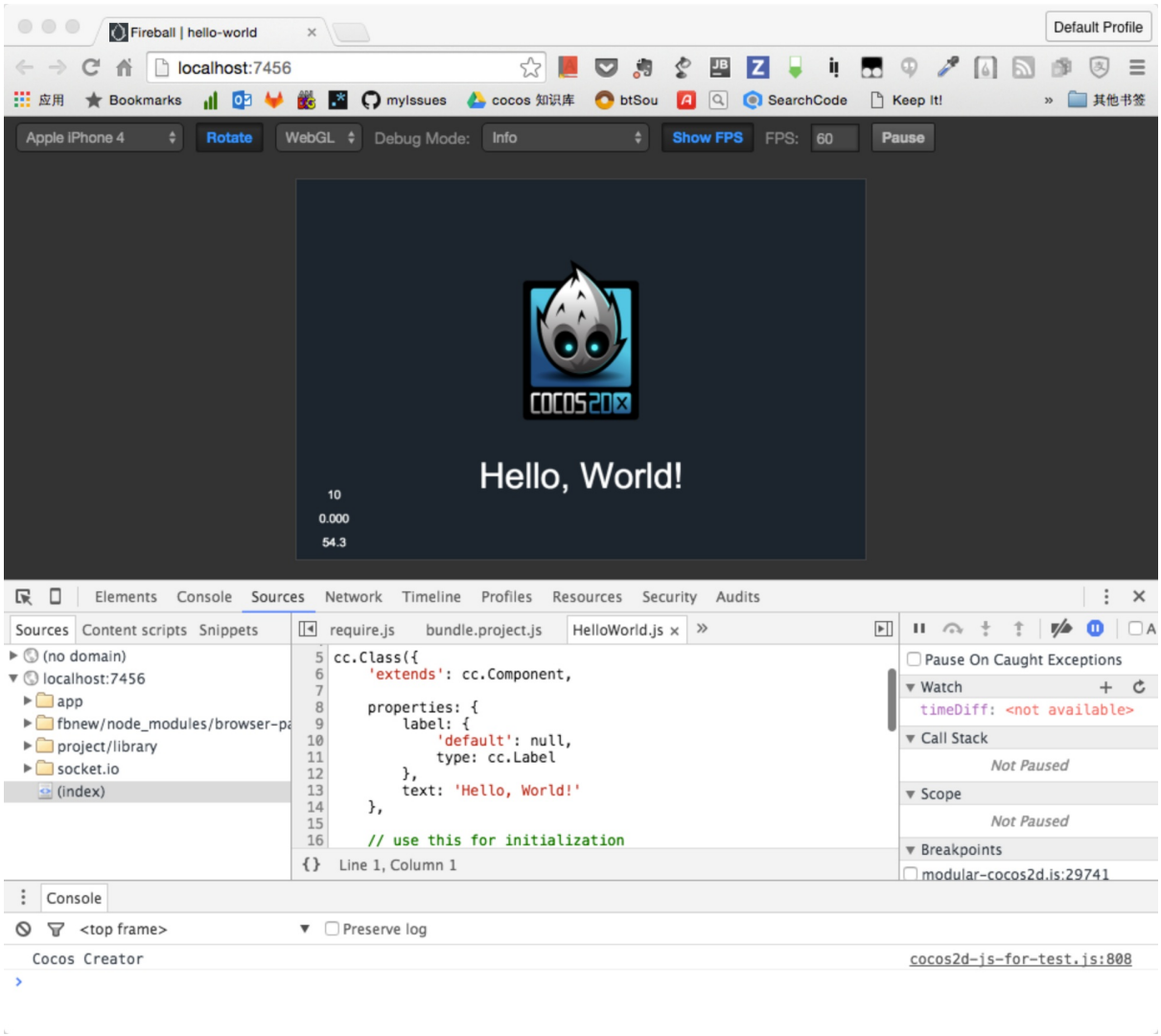

使用模拟器运行游戏时，脚本中的日志信息（使用 `cc.log` 打印的内容）和报错信息会出现在 **控制台** 面板中。

## 浏览器

选择 **浏览器** 后运行预览，会在用户的默认桌面浏览器中直接运行游戏的网页版本。推荐使用谷歌浏览器（Chrome）作为开发过程中预览调试用的浏览器，因为谷歌浏览器（Chrome）的开发者工具是最为全面强大的。

浏览器预览界面的最上边有一系列控制按钮可以对预览效果进行控制：

- 最左边选择预览窗口的比例大小，来模拟在不同移动设备上的显示效果
- **Rotate** 按钮决定显示横屏还是竖屏
- 左起第三个按钮可以选择 WebGL 或者 Canvas 渲染模式
- **Debug Mode** 里可以选择脚本中的哪些级别的日志会输出到浏览器控制台里。
- **Show FPS** 按钮切换每秒帧数和 Drawcall 数量显示
- **FPS** 限制最高每秒帧数
- **Pause** 暂停游戏




## 浏览器兼容性

Cocos Creator 开发过程中测试的桌面浏览器包括： Chrome，Firefox（火狐），IE11 其他浏览器只要内核版本够高也可以正常使用，对部分浏览器来说请勿开启 IE6 兼容模式。

移动设备上测试的浏览器包括：Safari (iOS)，Chrome，QQ 浏览器，UC 浏览器，百度浏览器，微信内置 Webview。

## 使用 VS Code 调试

您可以使用 VS Code 在项目工程里直接对游戏预览进行调试，详情请参阅 使用 VS Code 调试网页版游戏。

## 使用浏览器开发者工具进行调试

以谷歌浏览器为例，通过功能菜单的 `更多工具/开发者工具` 可以打开开发者工具界面，如上图所示。在开发者工具中，我们可以查看运行日志、打断点进行调试、在调用堆栈中查看每一步各个变量的值、甚至进行资源加载和性能分析。

要学习使用开发者工具进行调试，请阅读极客学院的 Chrome Dev Tools 使用指南，或其他浏览器的开发者工具帮助。

# 构建发布

预览和调试之后，如果您对您的游戏比较满意了，就可以通过主菜单的 项目/构建发布 打开 **构建发布** 窗口，来将游戏打包发布到您希望的目标平台上，包括 iOS、Android、HTML5、Windows、Mac、Cocos Play。

详细的构建发布流程，请查阅跨平台发布游戏一章的内容。

请注意，使用模拟器运行游戏的效果，和最终发布到原生平台可能会有一定差别，对于任何重要的游戏功能，都请以构建发布后的版本来做最终的测试。

# Cocos2d-x 用户上手指南

Cocos Creator 是一款以内容创作为导向的新型游戏开发工具，它完整集成了组件化的 Cocos2d-x WEB 版本，可发布游戏到 Web，iOS，Android，Mac，Windows等平台，更支持直接发布 Cocos Play 平台，把握手机页游渠道的快速发展带来的新机遇，更多关于 Cocos Creator 的介绍可参见介绍文档。

这篇文档旨在引导 Cocos2d-x 的用户开始使用 Cocos Creator 并尽量平滑得过渡到新编辑器的使用方式上来。文档中会讨论从 Cocos2d-x 到 Cocos Creator 开发中可能遇到的疑问并给出解答，不会深入到框架细节中去，而是以链接的方式给出不同部分的详细参考文档。

## 1. 典型误区

对于刚刚接触 Cocos Creator 的用户来说，可能会遇到下面几个典型的误区：

1. **希望配合 Cocos2d-x 来使用 Cocos Creator**：Cocos Creator 内部已经包含完整的 JavaScript 引擎和 cocos2d-x 原生引擎，不需要额外安装任何 cocos2d-x 引擎或 Cocos Framework。
2. **先搭建整体代码框架，再堆游戏内容**：Cocos Creator 的工作流是内容创作为导向的，所以对原型创作是非常友好的，编辑器中直接进行场景搭建和逻辑代码编写，即可驱动游戏场景运行起来，在下面的数据驱动章节会更详细介绍工作流上的变化
3. **在编码的时候直接查看 Cocos2d-JS 的 API**：Cocos Creator 可以说脱胎自 Cocos2d-JS，它们的 API 一脉相承，有很多相同的部分，但由于使用了全新的组件化框架，两者的 API 是有差异的，并且无法互相兼容
4. **希望将旧的 Cocos2d-JS 游戏直接运行到 Cocos Creator 上**：由于两者的 API 并不是100%兼容，所以这点是做不到的
5. **用继承的方式扩展功能**：在 Cocos2d-JS 中，继承是用来扩展节点功能的基本方法，但是在 Cocos Creator 中，不推荐对节点进行继承和扩展，节点只是一个实体，游戏逻辑应该实现在不同的组件中并组合到节点上

在文档开头就提及这些误区是希望开发者意识到，Cocos Creator 提供的工作流和开发思想和以往的 Cocos2d-x 引擎是存在很大差别的。为了更好得解答如何用正确的姿势在 Cocos Creator 中编码，下面两个章节会更详细介绍数据驱动带来的工作流变化和 API 层面的变化。

## 2. 数据驱动

在 Cocos2d-x 中，开发方式是以代码来驱动，游戏中的数据大多也是在代码中存储，除非开发者构建了自己的数据驱动框架。在 Cocos Creator 框架中，所有场景都会被序列化为纯数据，在运行时使用这些纯数据来重新构建场景，界面，动画甚至组件等元素。

### 何为代码驱动，何为数据驱动

为什么说 Cocos2d-x 是代码驱动的开发方式呢，举个例子，假设场景中有一个角色，它会不停地在一个区域来回走动，我们会写下面这样的代码：

```
var role = new cc.Sprite('role.png');
scene.addChild(role);
role.setPosition(100, 100);
var walk = cc.sequence(cc.moveBy(5, 100, 0), cc.moveBy(5, -100, 0)).repeatForever();
role.runAction(walk);
```

在这段代码中，`role` 的场景关系，位置信息，行动区间，动画信息全部都是通过代码实现的，所以称为代码驱动。也有一些开发者会将数据信息保存在别的文件中，但是仍然逃不了需要自己实现数据的解析代码。甚至在使用一些传统编辑器的过程中，也需要 Parser 来将编辑器导出的数据解析为场景。

而 Cocos Creator 提供的是更加彻底的数据驱动方式，在编辑器中编辑的所有信息都会被序列化到数据文件中，在运行时，引擎会通过反序列化的方式将数据直接转化为对象。这个过程和上面描述的过程又一个本质的区别：引擎中类的属性是可直接被序列化和反序列化的，不需要通过任何映射关系来转化。上面例子中的场景树，位置属性，动画等都可以被编辑器序列化为数据，加载场景的过程，不需要任何代码，只需要从场景数据中反序列化出整个场景即可：

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

## 序列化

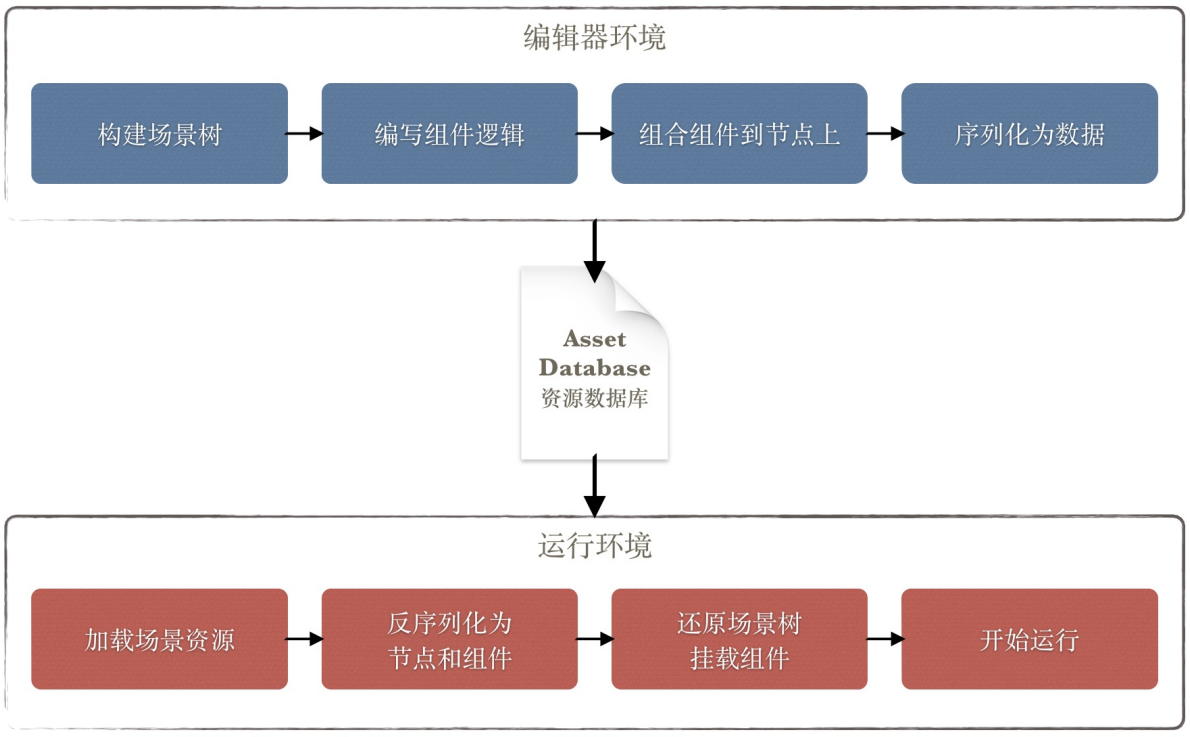
序列化和反序列化支持 Cocos Creator 中类的绝大多数公有属性，这些属性通过属性检查器面板暴露给开发者。开发者可以在编辑器中随意修改并保存，保存的过程就是将资源和场景数据序列化到资源数据库（Asset Database）中。反之，在加载场景的过程中，反序列化机制会根据场景数据实例化相应的对象，并还原编辑器中设置的所有属性。

不仅如此，数据驱动的强大之处在于，用户自己编辑的组件也可以进行属性声明。这些属性可以在编辑器中被编辑，也会被保存到场景数据中，最后在运行时被反序列化到游戏场景中。

资源数据库在编辑器中以资源管理器的形式呈现。

## 从数据驱动的角度理解 Cocos Creator 的工作流

与 Cocos2d-x 不同，Cocos Creator 的工作流是内容创作为导向的。也许开发者迁移过程中会遇到各种各样的困惑，但从数据驱动的角度来理解，这种工作流的变化就变得理所当然了。数据驱动使得场景可以被自由得进行编辑，不仅可以可视化得搭建整个场景，还可以对游戏逻辑进行编辑（编辑组件暴露出的属性）。这使得一切的入口点变成了编辑器，而不是代码。就像前面的示例，Cocos Creator 中，开发者首先用编辑器构建父子关系，摆放位置，设计动画；然后通过组件代码设计场景中节点的逻辑；最后将组件组合到不同的节点上。




# 3. Framework 层面的变化

开头已经提到，Cocos Creator 完整集成了组件化的 Cocos2d-JS。这是一个深度定制的版本，由于组件化的改造和数据驱动的需求，它与标准版本 Cocos2d-JS 拥有一脉相承但不互相兼容的 API 集。下面来具体说明一些重要的 API 差异：

## 逻辑树和渲染树

在 Cocos2d-JS 中，渲染器会遍历场景节点树来生成渲染队列，所以开发者构建的节点树实际上就是渲染树。而在 Cocos Creator 中我们引入了一个新的概念：逻辑树。开发者在编辑器中搭建的节点树和挂载的组件共同组成了逻辑树，其中节点构成实体单位，组件负责逻辑。

最重要的一点区别是：逻辑树关注的是游戏逻辑而不是渲染关系。

逻辑树会生成场景的渲染树，决定渲染顺序，不过开发者并不需要关心这些，只要在编辑器中保障显示效果正确即可。在编辑器的 Node Tree 层级管理器中，开发者可以调整逻辑树的顺序和父子关系。

## 场景管理

在 Cocos2d-JS 中，开发者用代码搭建完场景，通过 `cc.director.runScene` 来切换场景。在 Cocos Creator 中，开发者在编辑器中搭建完的场景，所有数据会保存为一个 scene-name.fire 文件，存在资源数据库（Asset Database）中。开发者可以通过 `cc.director.loadScene` 来加载一个场景资源，参见具体范例：

```
var sceneName = 'scene-name';
var onLaunched = function () {
    console.log('Scene ' + sceneName + ' launched');
};
// 第一个参数为场景的名字，第二个可选参数为场景加载后的回调函数
cc.director.loadScene(sceneName, onLaunched);
```

此外，我们提供了访问场景节点的接口：

```
// 获取逻辑树的场景节点
var logicScene = cc.director.getScene();
```

## 节点和组件

在 Cocos Creator 中，`cc.Node` 被替换为逻辑节点，旧的 Node 被改名为 `_ccsg.Node`，成为一个私有类，不再推荐使用。这么做的原因是开发者只需要关注逻辑节点即可，不需要关注底层的渲染节点。当然，我们尽量保留了它的 API 集，Transform 相关、节点树相关、Action 相关、属性等 API 都维持不变。

Cocos2d-JS 中曾经有一套简陋的组件机制，可以通过向 Node 添加组件并获得 onEnter、onExit、update 等回调。在 Cocos Creator 中，使用同样的接口 `addComponent`，只不过组件系统变成了整个引擎的核心，组件可以以各种各样的方式扩展逻辑节点的功能。甚至可以说，逻辑节点本身不应该包含任何实际游戏逻辑，它应该由各种逻辑组件组合出完整的逻辑。

这也导致了 Cocos2d-JS 和 Cocos Creator 中的最大区别：如何扩展节点的行为？

在 Cocos2d-JS 中可以通过继承来完成对各种节点类型的行为扩展，而在 Cocos Creator 中，请一定不要这样做，所有的扩展都应该通过添加组件的方式来完成。关于继承和组合的优劣之争由来已久，这里不做深入的探讨，但在 Cocos Creator 这样的组件式架构中，组合是最天然的扩展方式。

关于组件系统的更多信息可以参考节点和组件以及图像和渲染等章节。

## 坐标系和锚点

Cocos Creator 的坐标系仍然为左下角坐标系，不过在锚点上我们做了一些改变。先看图再解释：


假设一个背景节点的锚点为 (0.5, 0.5)，它拥有一个子节点，子节点位置为 (0, 0)，在 Cocos2d-JS 中将和左图中的表现一致，而在 Cocos Creator 中，将和右图一致。原因在于子节点的本地坐标系不同，在 Cocos2d-JS 设计中，不论父节点锚点位置如何，子节点坐标系的坐标原点都是父节点的左下角。而在 Cocos Creator 中，子节点坐标系的坐标原点就是父节点的坐标位置，也就是其锚点的位置。这样的修改对于编辑器更友好，一般对于用编辑器搭建的场景是非常自然的，而开发者在用代码构建节点时，就需要注意下了。

## 维持不变的类和对象

在 Cocos Creator 中，我们保留了很多重要类和对象的行为，包括：

- `cc.game`
- `cc.view`
- `cc.director`
- `cc.audioEngine`
- `cc.eventManager`
- `cc.scheduler`
- `cc.textureCache`
- `cc.sys`
- `cc.visibleRect`
- 动作系统
- 部分渲染节点类型：Menu 和 MenuItem、ClippingNode、ProgressTimer、MotionStreak、RenderTexture、DrawNode、Tile map 相关类等
- Chipmunk 物理引擎和 PhysicsDebugNode
- 基础数据类型

有几点需要注意的：

1. 上面保留的渲染节点类型只能和渲染树进行交互，不可以和逻辑节点以及组件混用。
2. 动作系统不仅支持渲染节点，所有对 Transform 进行操作的动作也支持逻辑节点。
3. 计时器（`cc.scheduler`）支持组件，组件中拥有 `schedule`、`scheduleOnce`、`unschedule`、`unscheduleAllCallbacks` 接口
4. 虽然保留了事件管理器，但是逻辑节点拥有一套新的事件 API，不建议直接使用事件管理器，具体参考后面关于事件系统的介绍

### 事件系统

在逻辑节点（ `cc.Node` ）中，我们添加了一系列全新的事件 API，从逻辑节点可以分发多种事件，也允许监听器向自己注册某种事件。监听器可以是一个简单的回调函数，也可以是回调函数和它的调用者组合，重要的API列表：

1. `node.emit(type, detail)` ： 通知所有监听 `type` 事件的监听器，可以发送一个附加参数。
2. `node.dispatchEvent(event)` ： 发送一个事件给它的监听器，支持冒泡。
3. `node.on(type, callback, target)` ： 持续监听 `node` 的 `type` 事件。
4. `node.once(type, callback, target)` ： 监听一次 `node` 的 `type` 事件。
5. `node.off(type, callback, target)` ： 取消监听所有 `type` 事件或取消 `type` 的某个监听器（用 callback 和 target 指定）。

这样的事件分发方式从集中由 `cc.eventManager` 分发，变为了拥有事件的节点本身分发自己的事件，可以说是集中式事件系统向离散式事件系统的转变。同时，Cocos Creator 还在节点中内建了鼠标和触摸两种系统事件：

| 枚举对象定义 | 对应的事件名 |
| --- | --- |
| `cc.Node.EventType.TOUCH_START` | 'touchstart' |
| `cc.Node.EventType.TOUCH_MOVE` | 'touchmove' |
| `cc.Node.EventType.TOUCH_END` | 'touchend' |
| `cc.Node.EventType.TOUCH_CANCEL` | 'touchcancel' |
| `cc.Node.EventType.MOUSE_DOWN` | 'mousedown' |
| `cc.Node.EventType.MOUSE_ENTER` | 'mouseenter' |
| `cc.Node.EventType.MOUSE_MOVE` | 'mousemove' |
| `cc.Node.EventType.MOUSE_LEAVE` | 'mouseleave' |
| `cc.Node.EventType.MOUSE_UP` | 'mouseup' |
| `cc.Node.EventType.MOUSE_WHEEL` | 'mousewheel' |

自此，开发者可以直接响应节点的各种鼠标和触摸事件，不再需要自己判断触点是否包含在节点中。此外，新的事件系统还支持事件冒泡，假设某个节点上触发了触摸事件，如果事件监听器中不停止冒泡，它的父节点也会触发同样的触摸事件。事件系统的具体使用方式可以参见监听和发射事件文档

## 4. 下一步

以上以非常概括的方式介绍了 Cocos Creator 的一些设计思路，以及从 Cocos2d-x 过渡过来的一些可能的障碍，无法覆盖到所有的知识点，本篇文档目标也不在于此，旨在让 Cocos2d-x 用户更轻松的上手 Cocos Creator。接下来，请继续阅读 Cocos Creator 用户手册，了解完整的工作流程和编程技巧。

# 获取帮助和支持

## 提交问题和反馈

除了本手册里提供的信息，您还可以随时通过下面的渠道获取信息或反馈问题给 Cocos Creator 开发团队：

- Cocos Creator 首页
- QQ群：577848332（4群），548341746（3群已满），428196107（2群已满），246239860 (1群已满)
- 论坛社区

## 演示和范例项目

注意，所有 Github 上的演示和范例项目都会跟随版本进行更新，默认分支对应目前最新的 Cocos Creator 版本，老版本的项目会以 `v0.7` 这样的分支名区分，分支名会和相同版本的 Cocos Creator 对应，下载使用的时候请注意。

- 范例集合：从基本的组件到交互输入，这个项目里包括了 case by case 的功能点用法介绍。
- Star Catcher：也就是 快速上手 文档里分步讲解制作的游戏。
- 腾讯合作开发的21点游戏
- UI 展示 Demo
- Duang Sheep：复制 FlappyBird 的简单游戏，不过主角换成了绵羊。
- 暗黑斩 Cocos Creator 复刻版：由 Veewo Games 独家授权原版暗黑斩资源素材，在 Cocos Creator 里复刻的演示项目
- i18n 游戏多语言支持范例：配套的文档教程是 i18n 游戏多语言支持
- 响应式 UI Demo：展示了可适配任意屏幕尺寸的 UI 系统
- 战斗动画 Demo：使用强力灵活的动画系统制作战斗动画
- 组队界面 Demo：展示使用 Prefab 实现数据和表现分离的动态 UI 构建

## 原生发布代码库

打包发布 Android 平台需要的代码库：

- Android SDK Windows
- Android SDK Mac
- Android NDK Windows 32位
- Android NDK Windows 64位
- Android NDK Mac

## 其他第三方工具和资源

### 代码编辑工具

- VS Code 微软推出的轻量级文本编辑器，支持 Cocos Creator 代码提示和语法高亮
- WebStorm
- Sublime Text
- Atom

### 图集生产工具

- TexturePacker
- Zwoptex

## 位图字体生产工具

- Glyph Designer
- Hiero
- BMFont (Windows)

## 2D 骨骼动画工具

- Spine
- Spriter
- DragonBones

## 粒子特效制作工具

- Particle Designer
- Particle2dx：免费在线工具

## 其他游戏开发资源

- Cocos Store：各类游戏美术资源、扩展工具

# 资源工作流程

## 添加资源

**资源管理器** 提供了三种在项目中添加资源的方式：

- 通过 **创建按钮** 添加资源
- 在操作系统的文件管理器中，将资源文件复制到项目资源文件夹下，之后再打开或激活 Cocos Creator 窗口，完成资源导入。
- 从操作系统的文件管理器中（比如 Windows 的文件资源管理器或 Mac 的 Finder），拖拽资源文件到 **资源管理器** 面板来导入资源

### 从外部导入资源

从操作系统中的其他窗口拖拽文件到 Cocos Creator 窗口中的**资源管理器**面板上，就能够从外部导入资源。该操作会自动复制资源文件到项目资源文件夹下，并完成导入操作。

## 导入和同步资源

**资源管理器** 中的资源和操作系统的文件管理器中看到的项目资源文件夹是同步的，在 **资源管理器** 中对资源的移动、重命名和删除，都会直接在用户的文件系统中对资源文件进行同步修改。同样的，在文件系统中（如 Windows 上的 Explorer 或 Mac 上的 Finder）对添加或删除资源，再次打开或激活 Cocos Creator 程序后，也会对 **资源管理器** 中的资源进行更新。

## 管理资源配置文件（.meta）

所有 `assets` 路径下的资源都会在导入时生成一份 **资源配置文件（.meta）** 这份配置文件提供了该资源在项目中的唯一标识（uuid）以及其他的一些配置信息（如图集中的小图引用，贴图资源的裁剪数据等），非常重要。

在编辑器中管理资源时，meta 文件是不可见的，对资源的任意删除、改名、移动操作，都会由编辑器自动同步相应的 meta 文件，确保 uuid 的引用不会丢失和错乱。

**注意在编辑器外部的文件系统中（Explorer，Finder）对资源文件进行删除、改名、移动时必须同步处理相应的 meta 文件**。资源文件和其对应的 meta 文件应该保持在同一个目录下，而且文件名相同。

### 处理无法匹配的资源配置文件（.meta）

如果您在编辑器外部的文件系统（Explorer，Finder等）中进行了资源文件的移动或重命名，而没有同步移动或重命名 meta 文件时，会导致编辑器将改名或移动的资源当做新的资源导入，可能会出现场景和组件中对该资源（包括脚本）的引用丢失。

在编辑器发现有未同步的资源配置文件时，会弹窗警告用户，并列出所有不匹配的 meta 文件。

这时无法正确匹配的资源配置文件会从项目资源路径（asset）中移除，并自动备份到 `temp` 路径下。

如果您希望恢复这些资源的引用，请将备份的 meta 文件复制到已经移动过的资源文件同一路径下，并保证资源文件和 meta 文件的文件名相同。注意编辑器在处理资源改名和移动时会生成新的 meta 文件，这些新生成的 meta 文件可以在恢复备份的 meta 后安全删除。

# 跨项目导入导出资源

除了导入基础资源外，从 1.5 版本开始编辑器支持将一个项目中的资源和其依赖完整的导出到另一个项目，详情请阅读导入导出资源工作流程。

# 导入其他编辑器项目

现在可以在 Cocos Creator 中导入其他编辑器的项目。具体的说明请参考：导入其他编辑器项目

# 常见资源工作流程

接下来我们会介绍 Cocos Creator 中主要资源类型和相关工作流程：

- 场景资源
- 图像资源
- 图集资源（Atlas）
- 自动图集资源（Auto Atlas）
- 图像资源的自动剪裁
- 预制资源（Prefab）
- 脚本资源
- 字体资源
- 粒子资源
- 声音资源
- 骨骼动画资源（Spine）
- 瓦片图资源（TiledMap）
- 骨骼动画资源（DragonBones）

# 创建和管理场景

## 创建场景

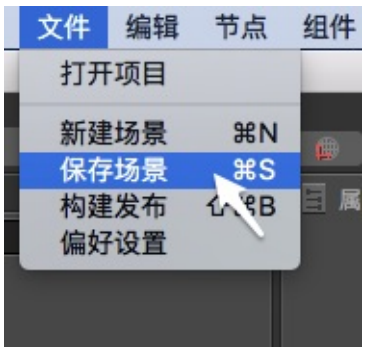
方法一：选择主菜单： `文件/新建场景`


方法二：在 **资源管理器** 中点击创建菜单，创建新场景。


## 保存场景

方法一：使用快捷键 `Ctrl + S` (Windows) 或 `Command + S` (Mac)

方法二：选择主菜单： `文件/保存场景`




## 切换场景

在 **资源管理器** 中，双击需要打开的场景。

# 修改场景资源自动释放策略

如果项目中的场景很多，随着新场景的切换，内存占用就会不断上升。除了使用 `cc.loader.release` 等 API 来精确释放不使用的资源，我们还可以使用场景的自动释放功能。要配置自动释放，可以在 **资源管理器** 中选中所需场景，然后在 **属性检查器** 中设置“自动释放资源”选项，该项默认关闭。
从当前场景切换到下一个场景时，如果当前场景不自动释放资源，则该场景中直接或间接引用到的所有资源（脚本动态加载的不算），**默认**都不主动释放。反之如果启用了自动释放，则这些引用到的资源**默认**都会自动释放。

> 已知问题：粒子系统的 plist 所引用的贴图不会被自动释放。如果要自动释放粒子贴图，请从 plist 中移除贴图信息，改用粒子组件的 Texture 属性来指定贴图。

## 防止特定资源被自动释放

启用了某个场景的资源自动释放后，如果在脚本中保存了对该场景的资源的“特殊引用”，则当场景切换后，由于资源已经被释放，这些引用可能会变成非法的，有可能引起渲染异常等问题。为了让这部分资源在场景切换时不被释放，我们可以使用 cc.loader.setAutoRelease 或者 cc.loader.setAutoReleaseRecursively 来保留这些资源。

> “特殊引用”指的是以全局变量、单例、闭包、“特殊组件”、“动态资源”等形式进行的引用。“特殊组件”是指通过 `cc.game.addPersistRootNode` 方法设置的常驻节点及其子节点上的组件，并且这些组件中包含以字符串 URL 或 UUID，或者以除了数组和字典外的其它容器去保存的资源引用。“动态资源”指的是在脚本中动态创建或动态修改的资源。这些资源如果还引用到场景中的其它资源，则就算动态资源本身不应该释放，其它资源默认还是会被场景自动释放。

# 修改场景加载策略

在 **资源管理器** 中，选中指定场景，可以在 **属性检查器** 中看到“延迟加载资源”选项，该项默认关闭。

## 不延迟加载资源

加载场景时，如果这个选项关闭，则这个场景直接或间接递归依赖的所有资源都将被加载，全部加载完成后才会触发场景切换。

## 延迟加载依赖的资源

加载场景时，如果选项开启，则这个场景直接或间接依赖的所有贴图、粒子和声音都将被延迟到场景切换后才加载，使场景切换速度极大提升。
同时，玩家进入场景后可能会看到一些资源陆续显示出来，并且激活新界面时也可能会看到界面中的元素陆续显示出来，因此这种加载方式更适合网页游戏。
使用这种加载方式后，为了能在场景中更快地显示需要的资源，建议一开始就让场景中暂时不需要显示的渲染组件（如 Sprite）保持非激活状态。

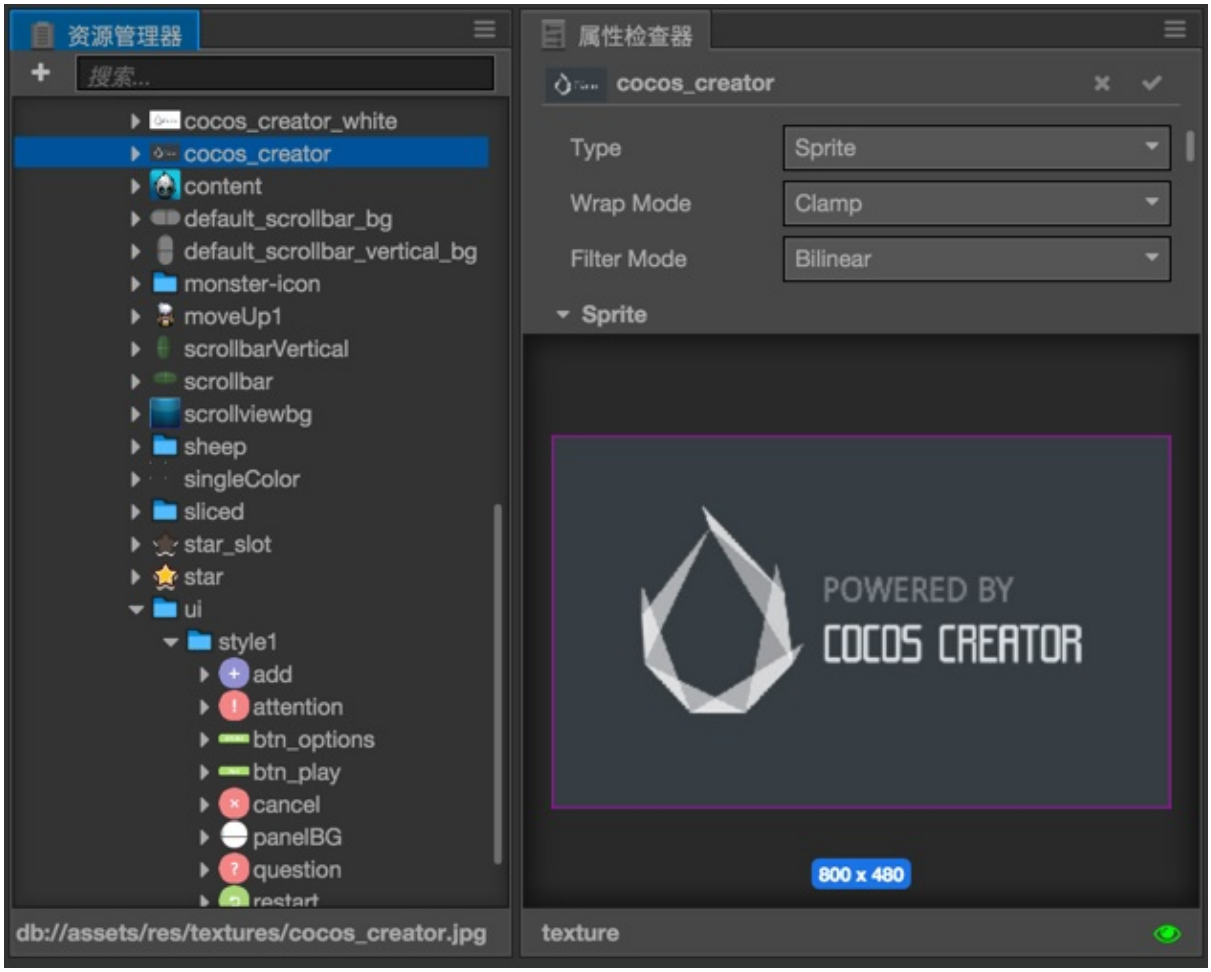
> Spine 和 TiledMap 依赖的资源永远都不会被延迟加载。

# 图像资源（Texture）

图像资源又经常被称作贴图、图片，是游戏中绝大部分图像渲染的数据源。图像资源一般由图像处理软件（比如 Photoshop，Windows 上自带的画图）制作而成并输出成 Cocos Creator 可以使用的文件格式，目前包括 **JPG** 和 **PNG** 两种。

## 导入图像资源

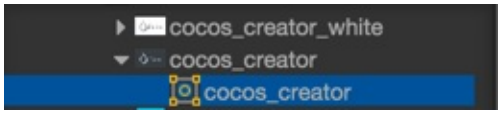
使用默认的资源导入方式就可以将图像资源导入到项目中，之后我们就可以在 **资源管理器** 中看到如下图所示的图像资源。




图像资源在 **资源管理器** 中会以自身图片的缩略图作为图标。在 **资源管理器** 中选中图像资源后，**属性检查器** 下方会显示该图片的缩略图。目前图像资源的属性设置功能还没有完善，请不要在 **属性检查器** 手动修改图像资源的属性设置。

## Texture 和 SpriteFrame 资源类型

在 **资源管理器** 中，图像资源的左边会显示一个和文件夹类似的三角图标，点击就可以展开看到它的子资源（sub asset），每个图像资源导入后编辑器会自动在它下面创建同名的 SpriteFrame 资源。




SpriteFrame 是核心渲染组件 **Sprite** 所使用的资源，设置或替换 **Sprite** 组件中的 `spriteFrame` 属性，就可以切换显示的图像。**Sprite** 组件的设置方式请参考Sprite 组件参考。

为什么会有 SpriteFrame 这种资源？这样的设置是因为除了每个文件产生一个 SpriteFrame 的图像资源（Texture）之外，我们还有包含多个 SpriteFrame 的图集资源（Atlas）类型。参考图集资源（Atlas）文档来了解更多信息。

下面是 Texture 和 SpriteFrame 的 API 接口文档：

- Texture 资源类型
- SpriteFrame 资源类型

# 使用 SpriteFrame

直接将 SpriteFrame 或图像资源从 **资源管理器** 中拖拽到 **层级管理器** 或 **场景编辑器** 中，就可以直接用所选的图像在场景中创建 **Sprite** 节点。

之后可以拖拽其他的 SpriteFrame 或图像资源到该 **Sprite** 组件的 `Sprite Frame` 属性栏中，来切换该 Sprite 显示的图像。

在 **动画编辑器** 中也可以拖拽 SpriteFrame 资源到已创建好的 Sprite Frame 动画轨道上，详见编辑序列帧动画文档。

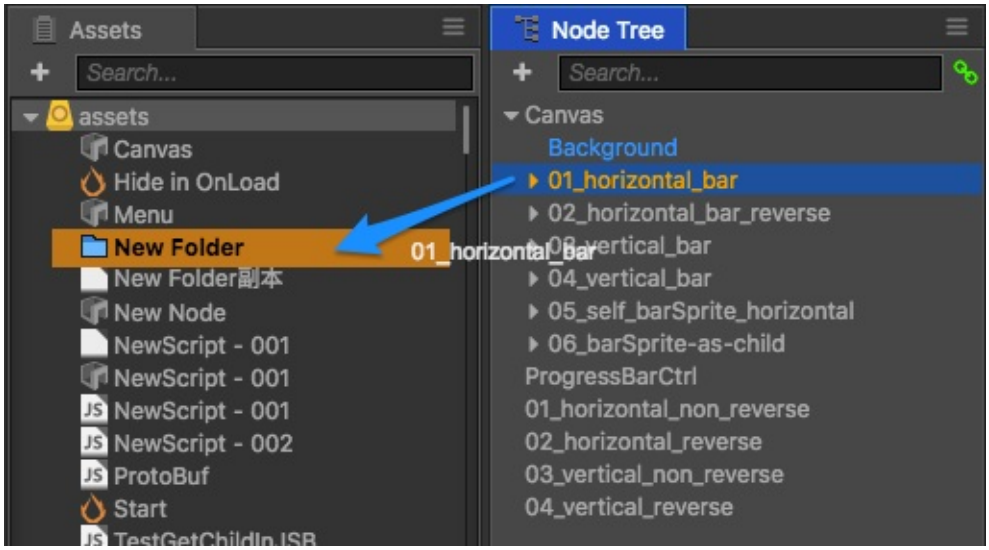
## 性能优化注意事项

使用单独存在的 Texture 作为 Sprite 资源，在预览和发布游戏时，将无法对这些 Sprite 进行批量渲染优化的操作。目前编辑器不支持转换原有的单张 Texture 引用到 Atlas 里的 SpriteFrame 引用，所以在开发正式项目时，应该尽早把需要使用的图片合成 Atlas（图集），并通过 Atlas 里的 SpriteFrame 引用使用。详情请继续阅读下一篇。

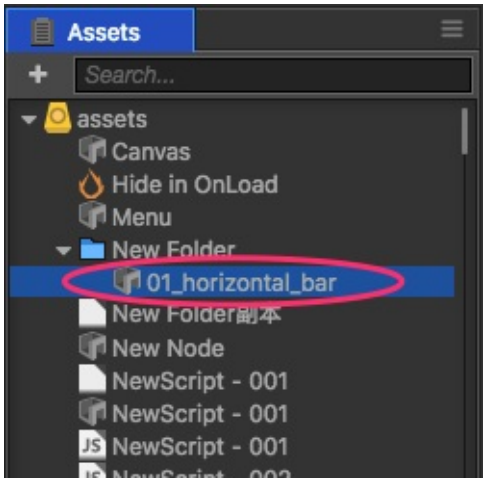
# 预制资源（Prefab）

## 创建预制

在场景中编辑好节点后，直接将节点从 **层级管理器** 拖到 **资源管理器**：

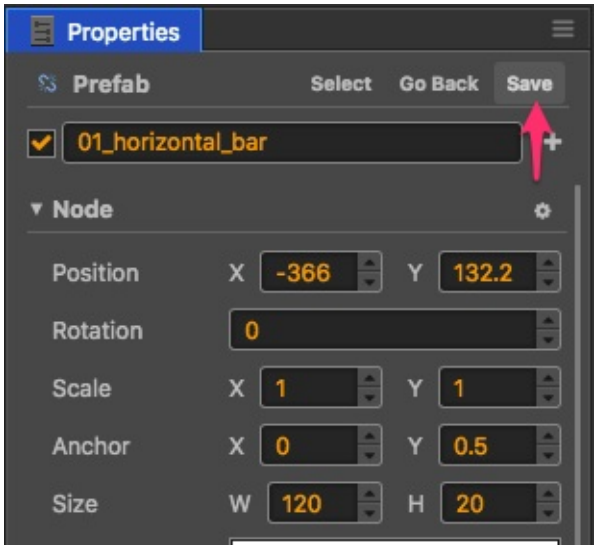



即可创建出一个预制：



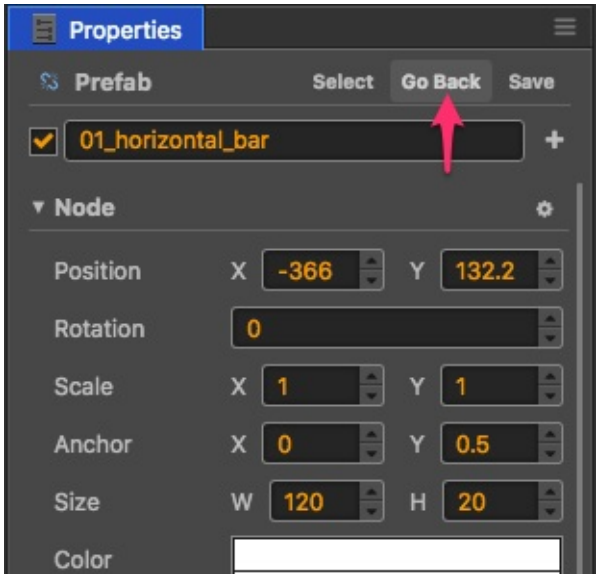

## 保存预制

在场景中修改了预制实例后，在 **属性检查器** 中直接点击 **保存**，即可保存对应的预制资源：




# 还原预制

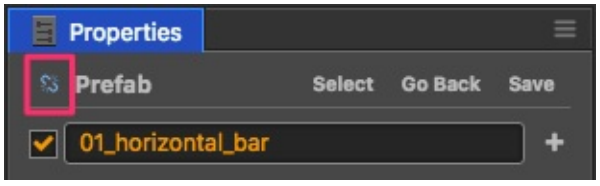
在场景中修改了预制实例后，在 **属性检查器** 中直接点击 **回退**，即可将预制对象还原为资源中的状态：



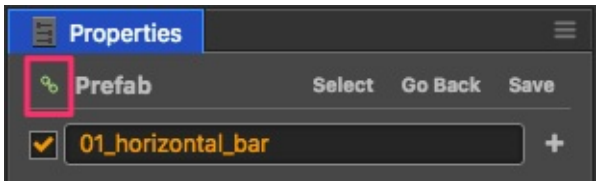

# 自动同步和手动同步

每个场景中的预制实例都可以选择要自动同步和还是手动同步。
设为**手动同步**时，当预制对应的原始资源被修改后，场景中的预制实例不会同步刷新，只有在用户手动还原预制时才会刷新。
设为**自动同步**时，该预制实例会自动和原始资源保持同步。

图中的图标表示当前预制的同步方式，点击图标将会在两种模式之间切换：



上面的图标表示当前预制使用手动同步，点击图标会切换到自动同步：




注意，为了保持引擎的精简，自动同步的预制实例有如下限制：

- 为了便于对各场景实例进行单独定制，场景中的预制根节点自身的 name、active、position 和 rotation 属性不会被自动同步。而其它子节点和所有组件都必须和原始资源保持同步，如果发生修改，编辑器会询问是要撤销修改还是要更新原始资源。
- 自动同步的预制中的组件无法引用该预制外的其它对象，否则编辑器会弹出提示。
- 自动同步的预制外面的组件只能引用该预制的根节点，无法引用组件和子节点，否则编辑器会弹出提示。

> 这些限制都仅影响编辑器操作，运行时不影响。

# 将预制还原成普通节点

从 **资源管理器** 中删除一个预制资源后，你可以将场景中对应的预制实例还原成普通节点。方法是选中预制实例，然后点击菜单 `节点 > 还原成普通节点` 。

# 延迟加载依赖的资源

在 **资源管理器** 中，选中预置资源，可以在 **属性检查器** 中看到“延迟加载资源”选项，该项默认关闭。选中之后，使用 **属性检查器** 关联、loadRes 等方式单独加载预置资源时，将会延迟加载预置所依赖的其它资源，提升部分页游的加载速度。详情请参考场景的延迟加载。

# 图集资源（Atlas）

图集（Atlas）也称作 Sprite Sheet，是游戏开发中常见的一种美术资源。图集是通过专门的工具将多张图片合并成一张大图，并通过 **plist** 等格式的文件索引的资源。可供 Cocos Creator 使用的图集资源由 **plist** 和 **png** 文件组成。下面就是一张图集使用的图片文件：

## 为什么要使用图集资源

在游戏中使用多张图片合成的图集作为美术资源，有以下优势：

- 合成图集时会去除每张图片周围的空白区域，加上可以在整体上实施各种优化算法，合成图集后可以大大减少游戏包体和内存占用
- 多个 Sprite 如果渲染的是来自同一张图集的图片时，这些 Sprite 可以使用同一个渲染批次来处理，大大减少 CPU 的运算时间，提高运行效率。

更形象生动的解释可以观看来自 CodeAndWeb 的教学视频What is a Sprite Sheet（什么是图集）。

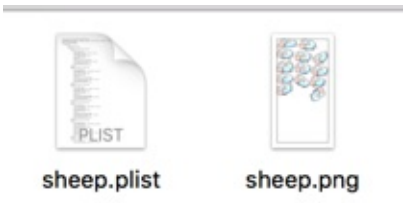
## 制作图集资源

要生成图集，首先您应该准备好一组原始图片：


接下来可以使用专门的软件生成图集，我们推荐的图集制作软件包括：

- TexturePacker
- Zwoptex

使用这些软件生成图集时请选择 cocos2d-x 格式的 plist 文件。最终得到的图集文件是同名的 **plist** 和 **png**。




# 导入图集资源

将上面所示的 **plist** 和 **png** 文件同时拖拽到 **资源管理器** 中，就可以生成可以在编辑器和脚本中使用的图集资源了。

## Atlas 和 SpriteFrame

在图像资源文档中，我们介绍了 Texture 和 SpriteFrame 的关系。导入图集资源后，我们可以看到类型为 `Atlas` 的图集资源可以点击左边的三角图标展开，展开后可以看到图集资源里包含了很多类型为 `SpriteFrame` 的子资源，每个子资源都是可以单独使用和引用的图片。


接下来对于 Sprite Frame 的使用方法就和图像资源中介绍的一样了，请查阅相关文档。

# 碎图转图集和拆分合并图集工作流程

在项目原型阶段或生产初期，美术资源的内容和结构变化都会比较频繁，我们通常会直接使用碎图（也就是多个单独的图片）来搭建场景和制作 UI。在之后为了优化性能和节约包体，需要将碎图合并成图集，或者将图集中的内容进行拆分或合并。

目前我们提供了一个简单的小工具来完成场景中对图片资源引用从碎图或老图集到新图集的重定向。下面介绍工作流程。

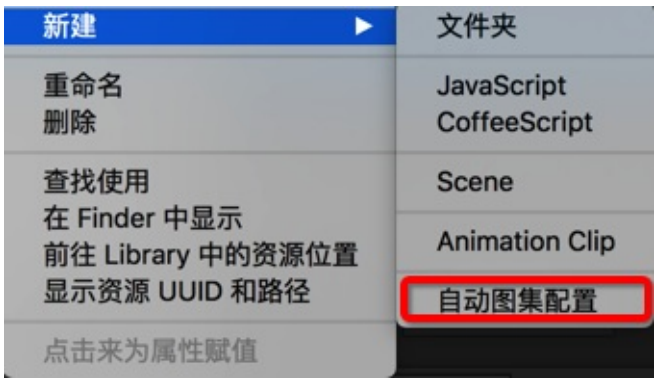
1. 生成新图集：不管是从碎图合并，还是将原来的图集重新拆分或合并，您都需要先使用 TexturePacker 生成完整的新图集。然后将新图集导入到项目资源文件夹中。
2. 双击打开您需要重定向资源引用的场景或 Prefab
3. 点击主菜单的「开发者->在当前场景使用指定图集替换 spriteFrame...」，在打开的对话框里选择您新生成的图集，等待替换操作完成。（如果新图集有多张，应该重复这一步直到所有相关新图集都替换完毕） 4.如果您有多个场景或 prefab，需要重复执行 2-3 步，遍历每个相关的场景或 Prefab
4. 确认所有相关图片资源的引用都已经替换成了新图集后，现在可以删除原有的碎图或旧图集了。

# 自动图集资源 (Auto Atlas)

**自动图集资源** 作为 Cocos Creator 自带的合图功能，可以将指定的一系列碎图打包成一张大图，具体作用和 Texture Packer 的功能很相近。

## 创建自动图集资源

在 **资源管理器** 中右键，可以在如下菜单中找到 **新建 -> 自动图集配置** 的子菜单，点击菜单将会新建一个类似 **AutoAtlas.pac** 的资源。




**自动图集资源** 将会以当前文件夹下的所有 **SpriteFrame** 作为碎图资源，以后会增加其他的选择碎图资源的方式。 如果碎图资源 **SpriteFrame** 有进行配置过，在打包后重新生成的 **SpriteFrame** 将会保留这些配置。

## 配置自动图集资源

在资源管理器中选中一个 **自动图集资源** 后，**属性检查器** 面板将会显示 **自动图集资源** 的所有可配置项。

| 属性 | 功能说明 |
| --- | --- |
| 最大宽度 | 单张图集最大宽度 |
| 最大高度 | 单张图集最大高度 |
| 间距 | 图集中碎图之间的间距 |
| 允许旋转 | 是否允许旋转碎图 |
| 输出大小为正方形 | 是否强制将图集长宽大小设置成正方形 |
| 输出大小为二次幂 | 是否将图集长宽大小设置为二次方倍数 |
| 算法 | 图集打包策略， 可选的策略有 [BestShortSideFit, BestLongSideFit, BestAreaFit, BottomLeftRule, ContactPointRule] |
| 输出格式 | 图集图片生成格式，可选的格式有 [png, jpg, webp] |
| 扩边 | 在碎图的边框为扩展出一像素外框，并复制相邻碎图像素到外框中 |
| 不包含未被引用资源 | 在预览中，此选项不会生效，构建后此选项才会生效 |

配置完成后可以点击 **预览** 按钮来预览打包的结果，按照当前自动图集配置生成的相关结果将会展示在 **属性检查器** 下面的区域。 需要注意的是每次配置过后，需要重新点击 **预览** 才会重新生成预览信息。

结果分为：

- Packed Textures, 显示打包后的图集图片以及图片相关的信息，如果会生成的图片有多张，则会往下在 **属性检查器** 中列出来。
- Unpacked Textures，显示不能打包进图集的碎图资源，造成的原因有可能是这些碎图资源的大小比图集资源的大小还大导致的，这时候可能需要调整下图集的配置或者碎图的大小了。

# 生成图集

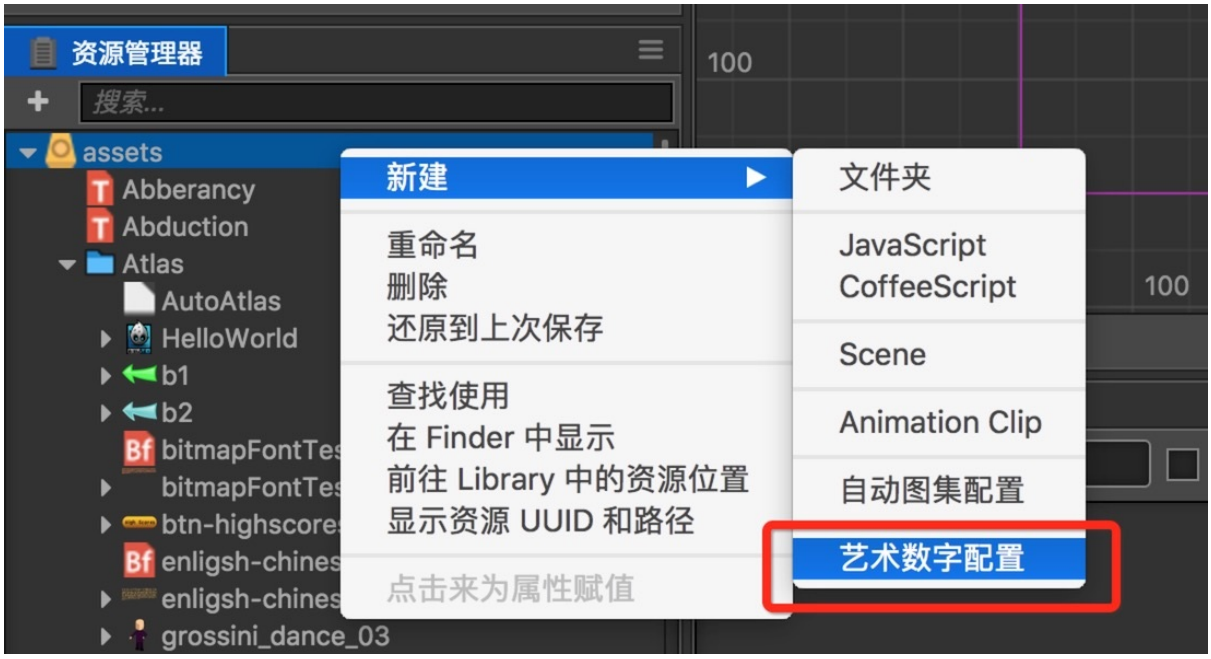
预览项目或者在 Cocos Creator 中使用碎图的时候都是直接使用的碎图资源，在 **构建项目** 这一步才会真正生成图集到项目中。 生成的大图将会放在 **构建目录** 下的 **res/raw-assets** 相对于项目中 assets 目录结构下的对应的目录中，以 **AutoAtlas-xx.png** 结构命名。 生成项目后可以到对应的目录下检查对应的图集资源是否生成成功了。

# 艺术数字资源 (LabelAtlas)

**艺术数字资源** 是一种用户自定义的资源，它可以用来配置艺术数字字体的属性。

## 创建艺术数字资源

在 **资源管理器** 中右键，可以在如下菜单中找到 **新建 -> 艺术数字配置** 的子菜单，点击菜单将会新建一个类似 **LabelAtlas.labelatlas** 的资源。



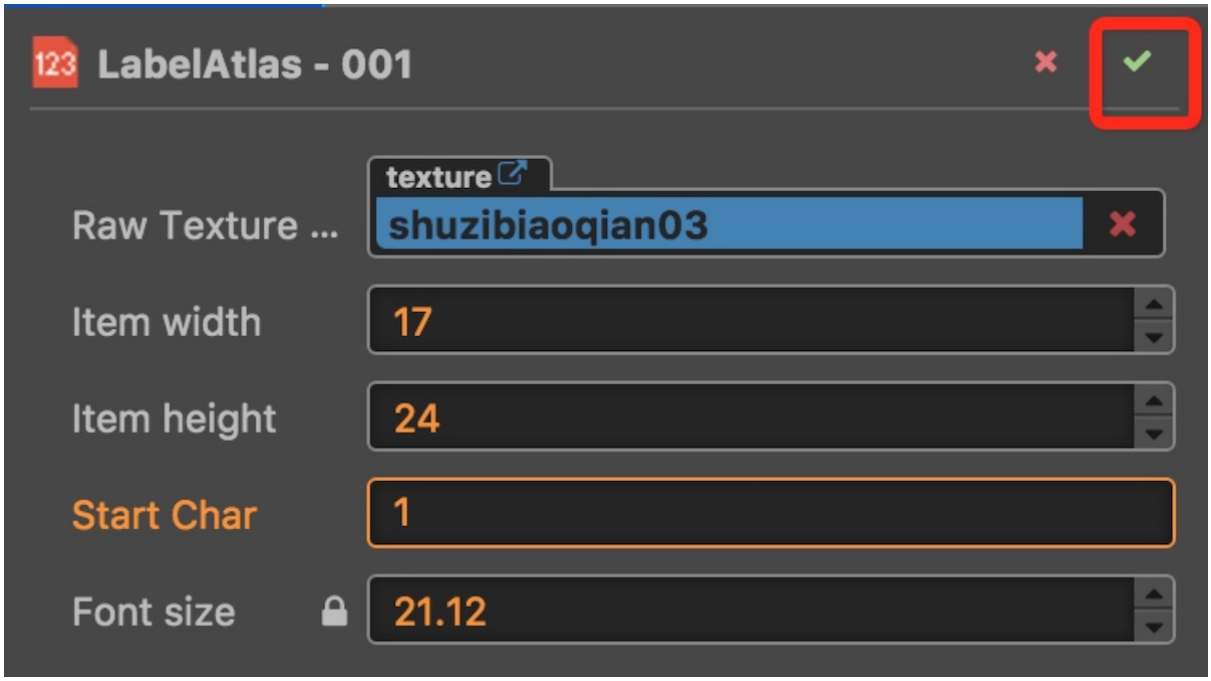

**艺术数字资源** 在使用之前需要进行一些配置，比如关联渲染的图片资源，设置每一个字符的宽高和起始字符信息。

## 配置艺术数字资源

在资源管理器中选中一个 **艺术数字资源** 后，**属性检查器** 面板将会显示 **艺术数字资源** 的所有可配置项。

| 属性 | 功能说明 |
| --- | --- |
| Raw Texture File | 指定渲染图片 |
| Item Width | 指定每一个字符的宽度 |
| Item Height | 指定每一个字符的高度 |
| Start Char | 指定艺术数字字体里面的第一个字符，如果字符是 Space，也需要在这个属性里面输入空格字符 |

配置完成后需要点击 **属性检查器** 右上角的绿色的打勾按钮来保存设置。




## 使用艺术数字资源

使用艺术数字资源非常简单，你只需要新建一个 Label 组件，然后把新建好的艺术数字资源拖到 Label 组件的 Font 属性即可。

# 资源导入导出工作流程

Cocos Creator 是专注于内容创作的游戏开发工具，在游戏开发过程中，对于每个项目该项目专用的程序架构和功能以外，我们还会生产大量的场景、角色、动画和 UI 控件等相对独立的元素。对于一个开发团队来说，很多情况下这些内容元素都是可以在一定程度上重复利用的。

在以场景和 Prefab 为内容组织核心的模式下，1.5版本的 Cocos Creator 内置了场景（.fire)和预制(.prefab)资源的导出和导入工具。

# 资源导出

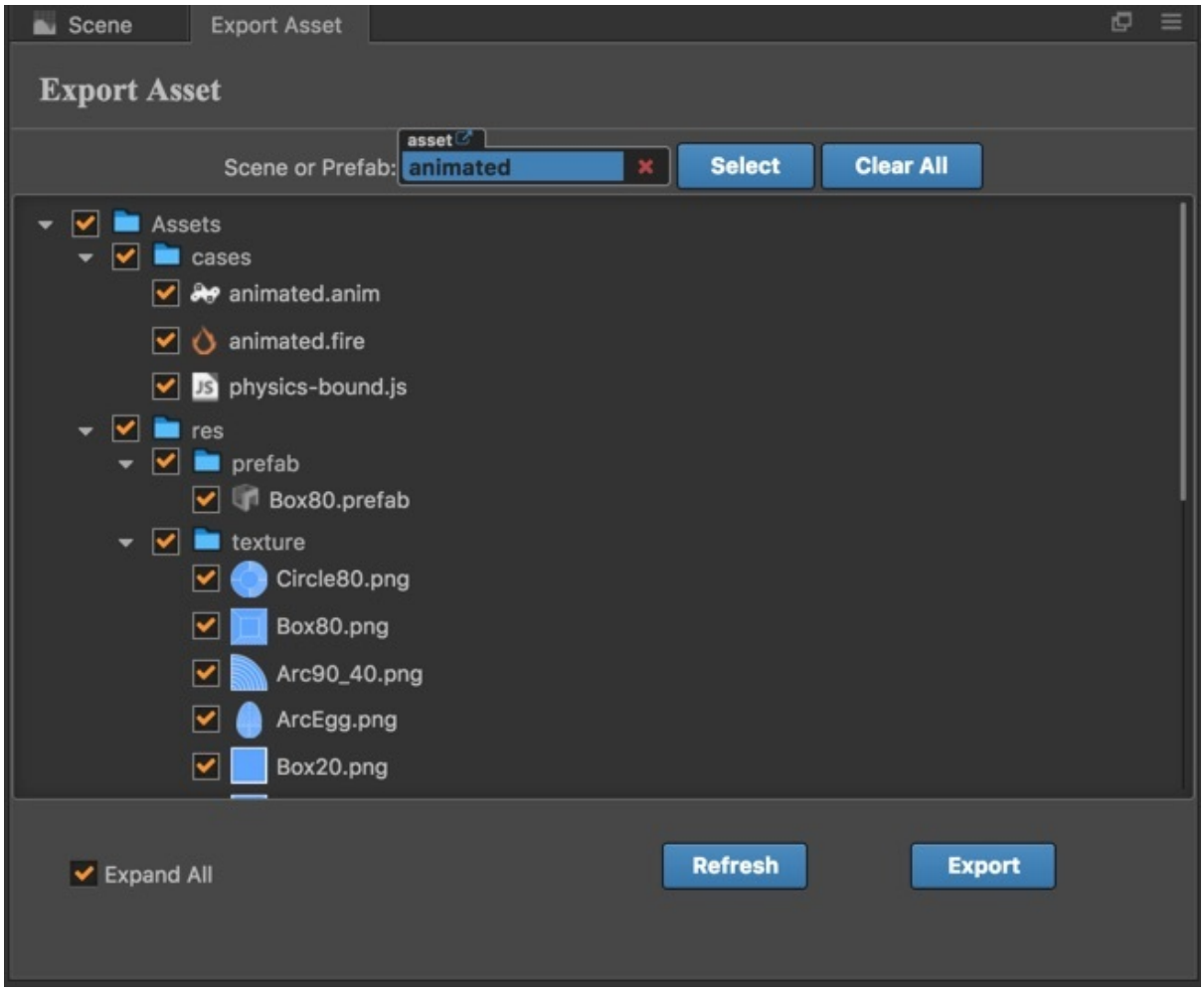
在主菜单选择 「文件->导出资源」，即可打开资源导出工具面板，接下来可以用以下两种方式选择需要导出的资源：

- 将场景或预制文件从 **资源管理器** 中拖拽到导出资源面板的资源栏中
- 点击资源栏右边的「选择」按钮，打开文件选择对话框，并在项目中选取你要导出的资源

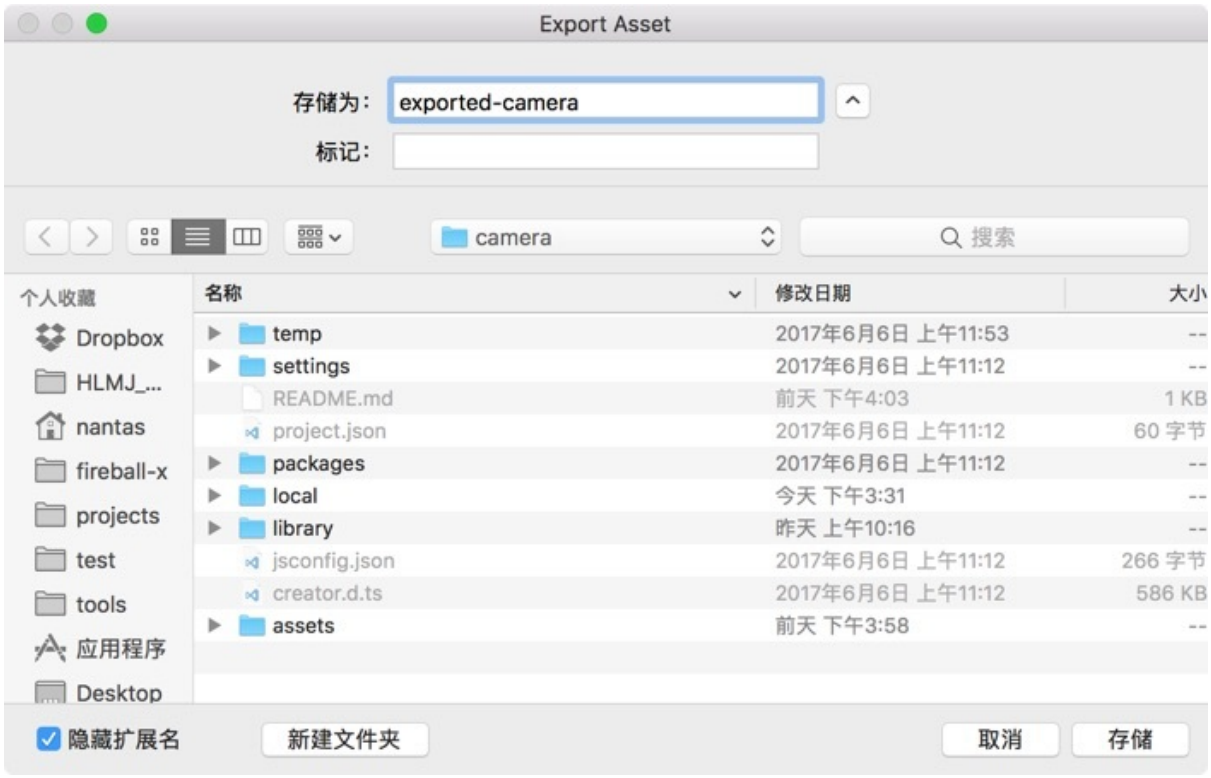
可以选择的资源包括 `.fire` 场景文件和 `.prefab` 预制文件。

## 确认依赖

导出工具会自动检查所选资源的依赖列表并列出在面板里，用户可以手动检查每一项依赖是否必要，并剔除部分依赖的资源。被剔除的资源将不会被导出。



确认完毕后点击 **导出** 按钮，会弹出文件存储对话框，用户需要指定一个文件夹位置和文件名，点击 **存储**，就会生成 `文件名.zip` 的压缩包文件，包含导出的全部资源。




# 资源导入

有了导出的资源包，就可以在新项目中导入这些现成的资源了，在新项目的主菜单里选择 「文件->导入资源」，即可打开资源导入面板。

点击 **Zip 文件路径** 输入框右边的 **选择** 按钮，在文件浏览对话框中选择刚才导出的导出资源压缩包。


导入过程中也会让用户再次确认导入资源依赖，在这时候也可以取消某些资源的勾选来不导入部分资源。

### 设置导入位置

相比导出过程，导入过程中增加了 `导入目标路径` 的设置，用户可以点击旁边的 **选择** 按钮，选择一个项目 `assets` 路径下的某个文件夹作为导入资源的放置位置。由于导出资源时所有资源的路径都是以相对于 `assets` 路径来保存的，导入时如果不希望导入的资源放入 `assets` 根目录下，就可以再指定一层中间目录来隔离不同来源的导入资源。

设置完成后点击 **导入** 按钮，会弹出确认对话框，确认后就会把列出的资源导入到目标路径下。

### 脚本和资源冲突

由于 Creator 项目中的脚本不能同名，当导入的资源包含和当前项目里脚本同名的脚本时，将不会导入同名的脚本。如果出现导入资源的 UUID 和项目中现有资源 UUID 冲突的情况，会自动为导入资源生成新的 UUID，并更新在其他资源里的引用。

## 工作流应用

有了全新的资源导入/导出功能，我们可以进一步根据项目和团队需要扩展工作流，比如：

- 程序和美术分别使用不同的项目进行开发，美术开发好的 UI、角色、动画可以通过导出资源的方式引入到程序负责的主项目中。避免冲突并进一步加强权限管理。
- 一个项目开发完成后，可以将可重用的资源导出并导入到一个公共资源库中，在公共资源库项目里对该资源进行优

化整理后，可以随时再导出到其他项目，大大节约开发成本。
- 将一个较为完整的功能做成场景或预制，并上传资源包到扩展商店，方便社区直接取用。

在此基础上还可以发展出更多样化的工作流程，开发团队可以发挥想象力，并使用扩展插件系统进一步定制导入导出的数据和行为，满足更复杂的需要。

# 图像资源的自动剪裁

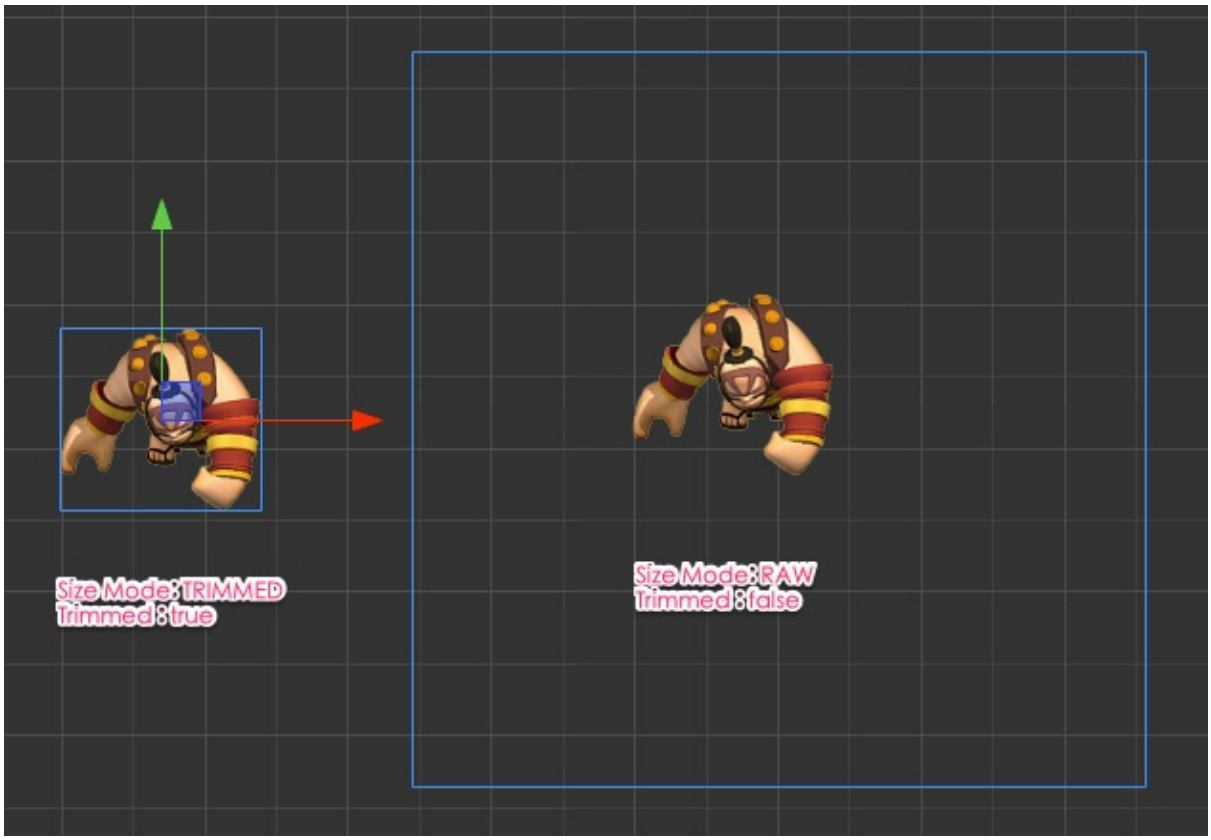
导入图像资源后生成的 SpriteFrame 会进行自动剪裁，去除原始图片周围的透明像素区域。这样我们在使用 SpriteFrame 渲染 Sprite 时，将会获得有效图像更精确的大小。


## Sprite 组件剪裁相关设置详解

和图片裁剪相关的 **Sprite** 组件设置有以下两个：

- `Trim` 勾选后将在渲染 Sprite 图像时去除图像周围的透明像素，我们将看到刚好能把图像包裹住的约束框。取消勾选，Sprite 节点的约束框会包括透明像素的部分。
- `Size Mode` 用来将节点的尺寸设置为原图或原图裁剪透明像素后的大小，通常用于在序列帧动画中保证图像显示为正确的尺寸。有以下几种选择：
  - `TRIMMED` 选择这个选项，会将节点的尺寸（size）设置为原始图片裁剪掉透明像素后的大小。
  - `RAW` 选择这个，会将节点尺寸设置为原始图片包括透明像素的大小。
  - `CUSTOM` 自定义尺寸，用户在使用 **矩形变换工具** 拖拽改变节点的尺寸，或通过修改 `Size` 属性，或在脚本中修改 `width` 或 `height` 后，都会自动将 `Size Mode` 设为 `CUSTOM`。表示用户将自己决定节点的尺寸，而不需要考虑原始图片的大小。

下图中展示了两种常见组合的渲染效果：



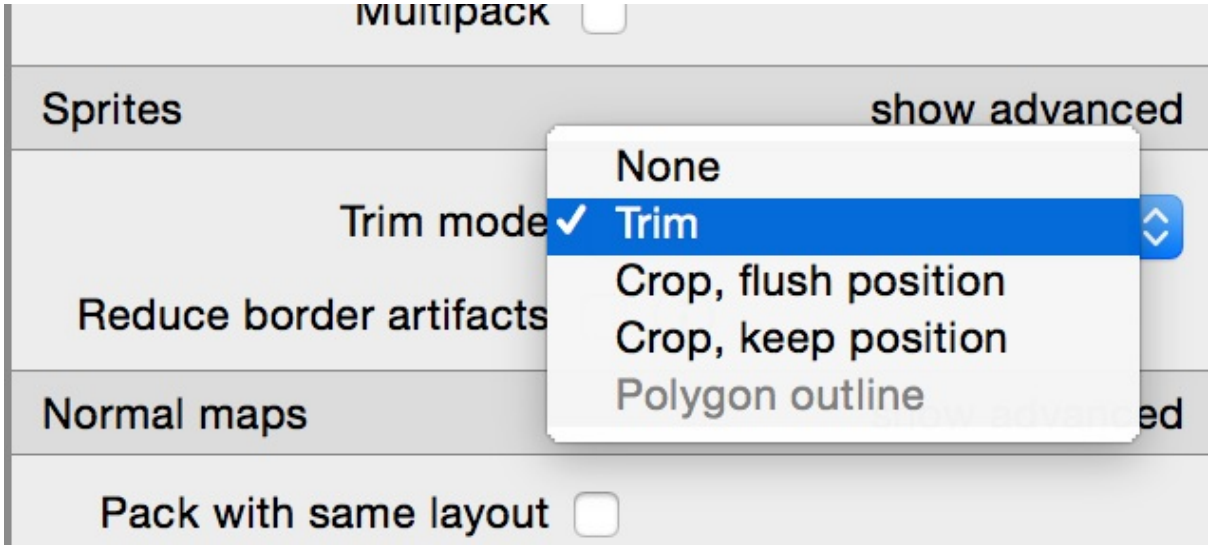

# 自带位置信息的序列帧动画

有很多动画师在绘制序列帧动画时，会使用一张较大的画布，然后将角色在动画中的运动直接通过角色在画布上的位置变化表现出来。在使用这种素材时，我们需要将 **Sprite 组件** 的 `Trim` 设为 `false`，将 `Size Mode` 设为 `RAW`。这样动画在播放每个序列帧时，都将使用原始图片的尺寸，并保留图像周围透明像素的信息，这样才能正确显示绘制在动画中的角色位移。

而 `Trim` 设为 `true`，则是在位移完全由角色位置属性控制的动画中，更推荐使用的方式。

# TexturePacker 设置

在制作序列帧动画时，我们通常会使用TexturePacker这样的工具将序列帧打包成图集，并在导入后通过图集资源下的 `SpriteFrame` 来使用。在 TexturePacker 中输出图集资源时，Sprites 分类下的 **Trim mode** 请选择 `Trim`，一定不要选择 `Crop, flush position`，否则透明像素剪裁信息会丢失，您在使用图集里的资源时也就无法获得原始图片未剪裁的尺寸和偏移信息了。


Multipack
Sprites
show advanced
Trim mode
None
Trim
Crop, flush position
Crop, keep position
Polygon outline
Reduce border artifacts
Normal maps
Pack with same layout

# 脚本资源

脚本用来驱动项目逻辑，实现交互功能。

- 常用操作请参考：创建和使用组件脚本
- 代码编写方法请参考：脚本开发工作流程。

# 字体资源

使用 Cocos Creator 制作的游戏中可以使用三类字体资源：系统字体，动态字体和位图字体。

其中系统字体是通过调用游戏运行平台自带的系统字体来渲染文字，不需要用户在项目中添加任何相关资源。要使用系统字体，请使用Label组件中的 **Use System Font** 属性。

# 导入字体资源

## 动态字体

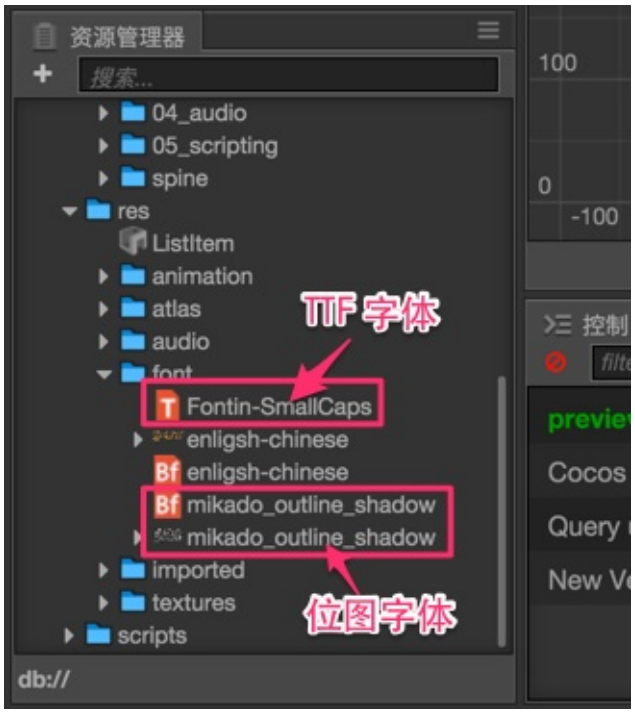
目前 Cocos Creator 支持 **TTF** 格式的动态字体。只要将扩展名为 **TTF** 的字体文件拖拽到 **资源管理器** 中，即可完成字体资源的导入。

## 位图字体

位图字体由 **fnt** 格式的字体文件和一张 **png** 图片组成，fnt 文件提供了对每一个字符小图的索引。这种格式的字体可以由专门的软件生成，请参考位图字体制作工具。

在导入位图字体时，请务必将 fnt 文件和 png 文件同时拖拽到 **资源管理器** 中。

导入后的字体在 **资源管理器** 中显示如下：




**注意** 为了提高资源管理效率，建议将导入的 `fnt` 和 `png` 文件存放在单独的目录下，不要和其他资源混在一起。

# 使用字体资源

字体资源需要通过 Label 组件来渲染，下面是在场景中创建带有 Label 组件的节点的方法。

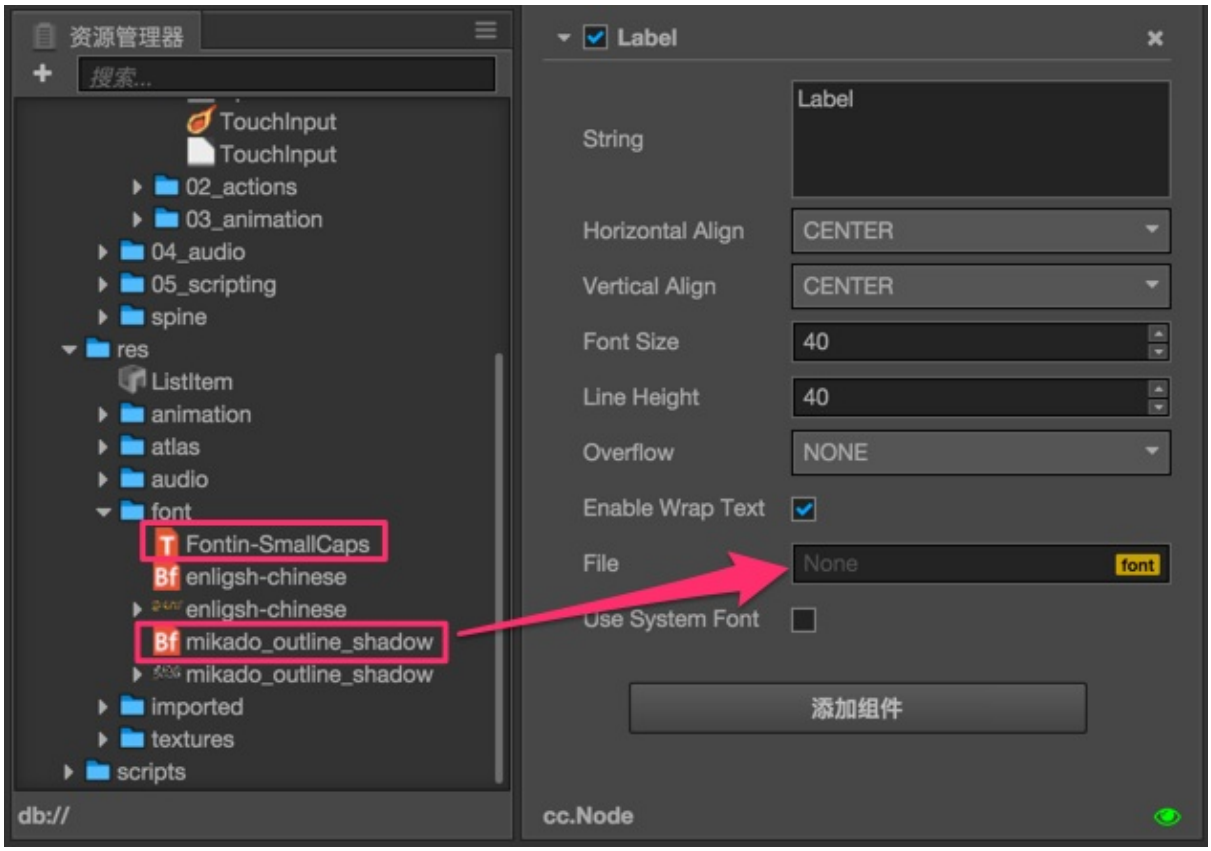
## 使用菜单创建 Label（字体）节点

在 **层级管理器** 中点击左上角的 **创建节点** 按钮，并选择 `创建渲染节点/Label（文字）`，就会在场景中创建出一个带有 **Label** 组件的节点。


您也可以通过主菜单的 `节点/创建渲染节点/Label（文字）` 来完成创建，效果和上面的方法一样。


## 关联字体资源

使用上面方法创建的字体组件默认使用系统字体作为关联的资源，如果想要使用导入到项目中的 TTF 或位图字体，可以将您的字体资源拖拽到创建的 **Label** 组件中的 `File` 属性栏中。




这时场景中的字体会立刻用刚才指定的字体资源进行渲染。您也可以根据项目需要，自由的切换同一个 **Label** 组件的 `File` 属性，来使用 TTF 或位图字体。切换字体文件时，Label 组件的其他属性不受影响。

如果要恢复使用系统字体，可以点击 `Use System Font` 的属性复选框，来清除 `File` 属性中指定的字体文件。

### 拖拽创建 Label（字体）节点

另外一种快捷使用指定资源创建字体节点的方法，是直接从 **资源管理器** 中拖拽字体文件（TTF 或位图字体都可以）到 **层级管理器** 中。和上面用菜单创建的唯一区别，是使用拖拽方式创建的文字节点会自动使用拖拽的字体资源来设置 **Label** 组件的 `File` 属性。

## 位图字体合并渲染

如果位图字体使用的贴图和其他 Sprite 使用的贴图是同一张，而且位图字体和 Sprite 之间没有插入使用其他贴图的渲染对象时，位图字体就可以和 Sprite 合并渲染批次。在放置位图字体资源时，请把 `.fnt` 文件、`.png` 文件和 Sprite 所使用的贴图文件放在一个文件夹下，然后参考 自动图集工作流程 将位图字体的贴图和 Sprite 使用的贴图打包成一个图集，即可在原生和 WebGL 渲染环境下自动享受位图字体合并渲染的性能提升。

详情请参考 BMFont 与 UI 合图自动批处理。

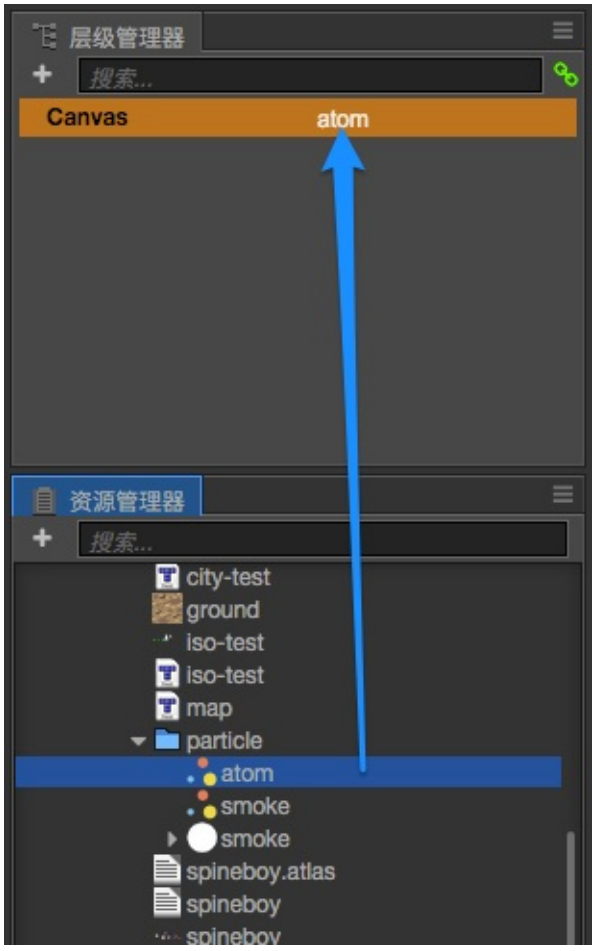
# 粒子资源（ParticleSystem）

## 导入粒子资源

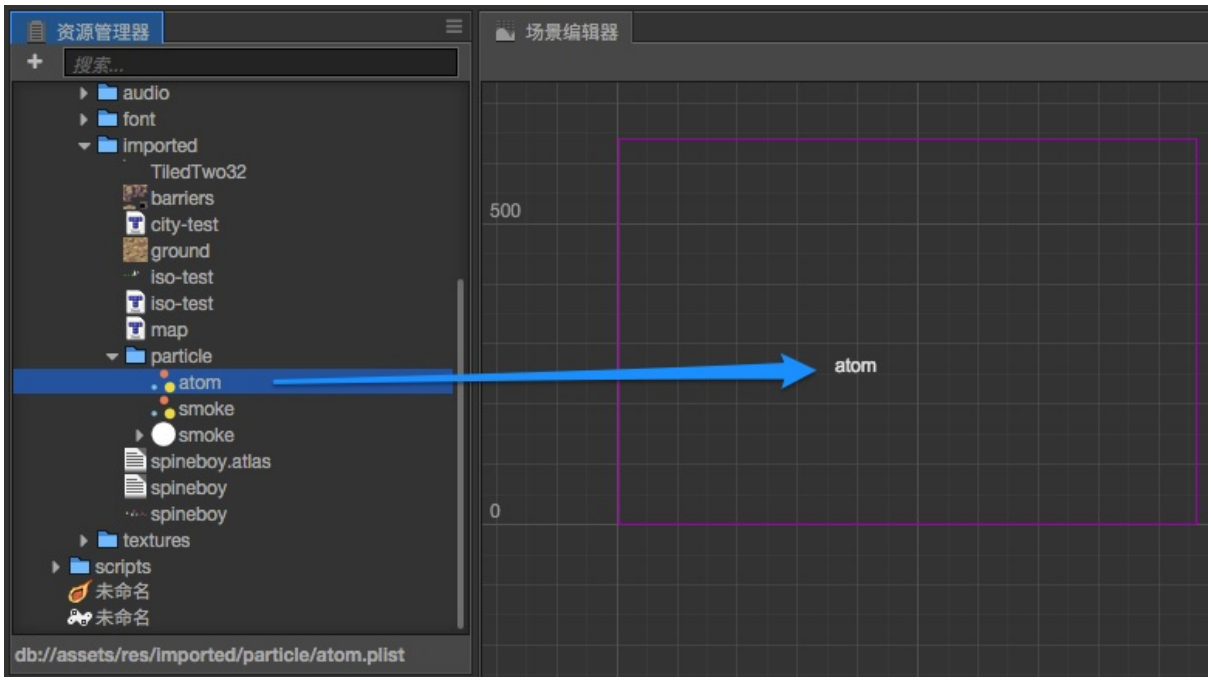
将 Cocos2d 支持的粒子 `.plist` 文件直接放到工程资源目录下。


## 在场景中添加粒子系统

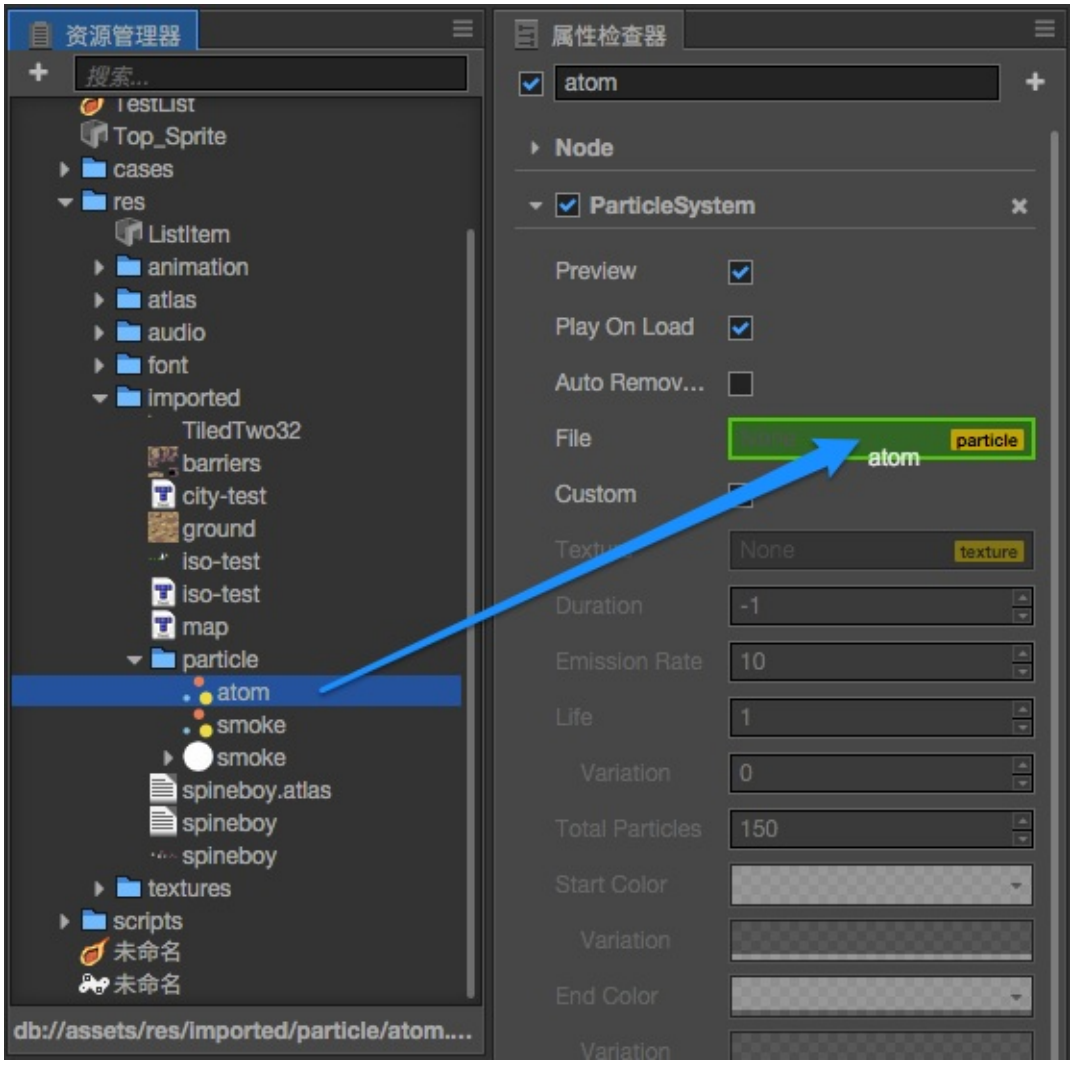
方法一，从 **资源管理器** 里将粒子资源直接拖到 **层级管理器**：



方法二，从 **资源管理器** 里将粒子资源直接拖到 **场景编辑器**：




方法三，在已有节点上添加一个 **粒子系统（ParticleSystem）** 组件，从 **资源管理器** 里将粒子资源直接赋给组件的 `File` 属性：




注意：不支持 `.plist` 文件中的 sourcePosition 属性的导入。

# 在项目中的存放

为了提高资源管理效率，建议将导入的 `plist` 和 `png` （如果有使用贴图）文件存放在单独的目录下，不要和其他资源混在一起。

# 渲染错误解决方法

粒子使用的 png 贴图文件或 base64 格式的内置图片文件可能会有不正确的预乘信息，导致渲染出的粒子不能正确显示透明区域。如果出现这种情况，请手动修改粒子 plist 文件中的 blendFuncSource 属性到下面的值：

```
    <key>blendFuncSource</key>
    <integer>770</integer>
```

# 声音资源

声音资源就是简单的音频文件。

引擎通过各个平台提供的基础接口，播放不同的声音资源来实现游戏内的背景音乐和音效。

## 关于声音的加载模式

**音频的加载方式只影响 Web 上的加载效果** 由于各个 Web 平台实现标准的进度不一致，所以提供了两种声音资源的加载方式。

## WebAudio

通过 WebAudio 方式加载的声音资源，在引擎内是以一个 buffer 的形式缓存的。

这种方式的优点是兼容性好，问题比较少。缺点是占用的内存资源过多。

## DOM Audio

通过生成一个标准的

使用标准的 audio 元素播放声音资源的时候，在某些浏览器上可能遇到一些限制。

比如：每次播放必须是用户操作事件内才允许播放（WebAudio 只要求第一次），只允许播放一个声音资源等。

## 手动选择按某种解析方式加载音频

有时候我们可能不会使用场景的自动加载或是预加载功能，而是希望自己手动控制 cc.load 资源的加载流程。 这个时候我们也是可以通过音频资源的 url 来选择加载的方式。

## 默认方式加载音频

音频默认是使用 webAudio 的方式加载并播放的，只有在不支持的浏览器才会使用 dom 元素加载播放。

```
cc.loader.load(cc.url.raw('resources/background.mp3'), callback);
```

## 强制使用 dom element 加载

1. 在资源管理器内选中一个 audio，属性检查器内会有加载模式的选择
2. 音频在加载过程中，会读取 url 内的 get 参数。其中只需要定义一个 useDom 参数，使其有一个非空的值，这样在 audioDownloader 中，就会强制使用 DOM element 的方式加载播放这个音频。

   ```
   cc.loader.load(cc.url.raw('resources/background.mp3?useDom=1'), callback);
   ```

需要注意的是，如果使用 dom element 加载的音频，在 cc.load 的 cache 中，缓存的 url 也会带有 ?useDom=1 **建议不要直接填写资源的 url** 尽量在脚本内定义一个 AudioClip，然后从编辑器内定义。

参考：

- 音频播放

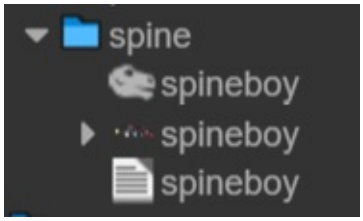
# 骨骼动画资源（Spine）

骨骼动画资源是由 Spine® 所导出的数据格式。

## 导入骨骼动画资源

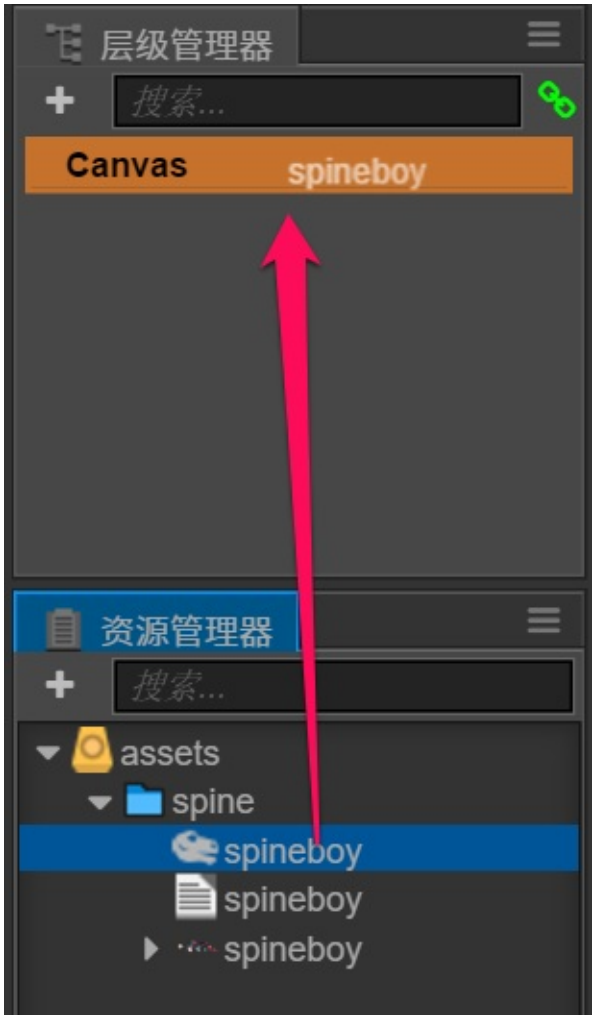
骨骼动画所需资源有：

- .json 骨骼数据
- .png 图集纹理
- .txt/.atlas 图集数据

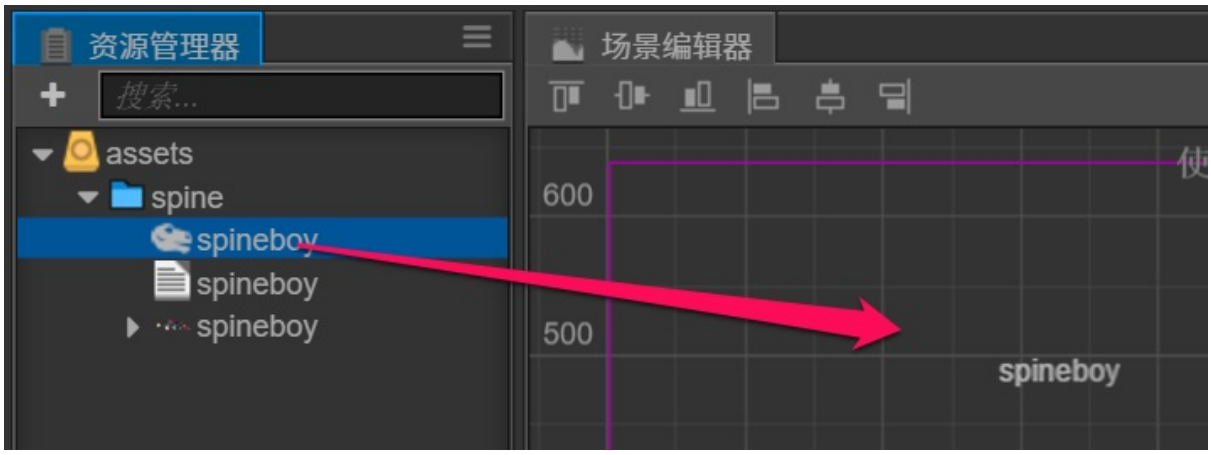



## 创建骨骼动画资源

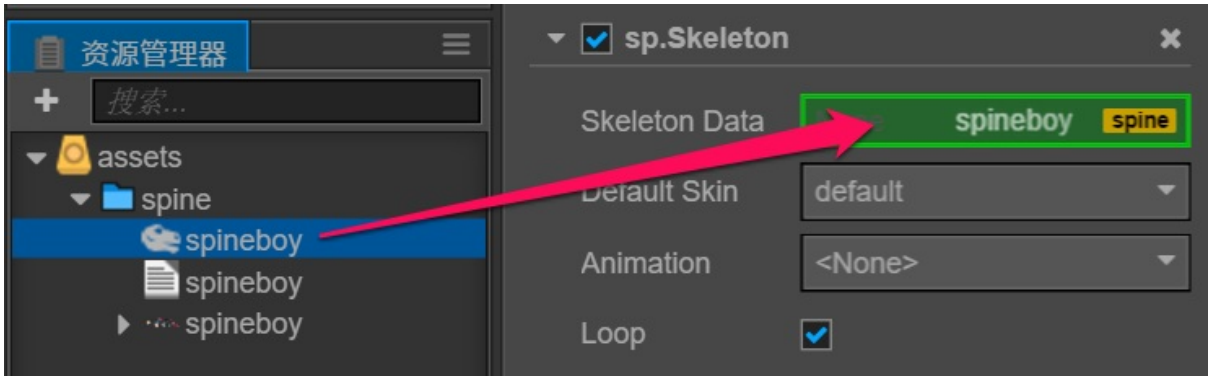
第一种方式：从 **资源管理器** 里将骨骼动画资源拖动到层级管理器中:




第二种方式：从 **资源管理器** 里将骨骼动画资源拖动到场景中:




第三种方式：从 **资源管理器** 里将骨骼动画资源拖动到已创建 Spine 组件中 Skeleton Data 属性中：


资源管理器
搜索...
assets
spine
spineboy
spineboy
spineboy
sp.Skeleton
Skeleton Data
spineboy
spine
Default Skin
default
Animation
<None>
Loop

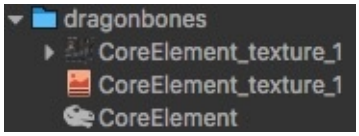
# 骨骼动画资源（DragonBones）

DragonBones 骨骼动画资源是由 DragonBones 编辑导出的数据格式。

## 导入 DragonBones 骨骼动画资源

DragonBones 骨骼动画资源有：

- .json 骨骼数据
- .json 图集数据
- .png 图集纹理

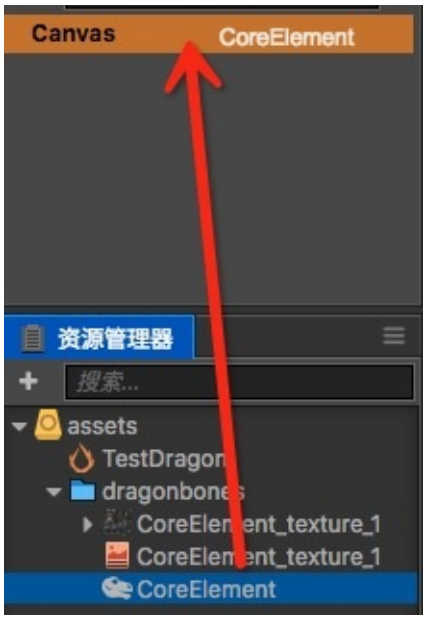



## 创建骨骼动画资源

在场景中使用 DragonBones 骨骼动画资源需要两个步骤：

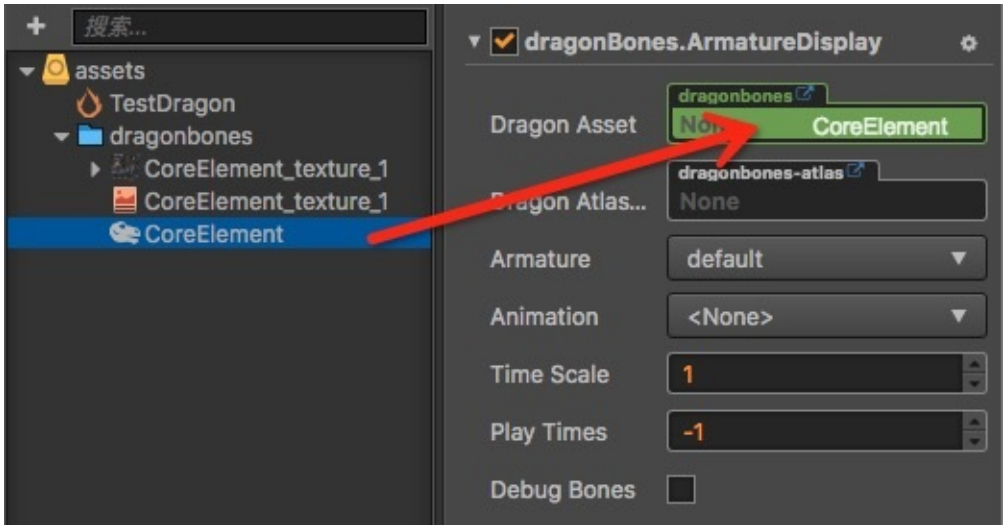
**1. 创建节点并添加 DragonBones 组件，可以通过三种方式实现：**

第一种方式：从 **资源管理器** 里将骨骼动画资源拖动到 **层级管理器** 中:



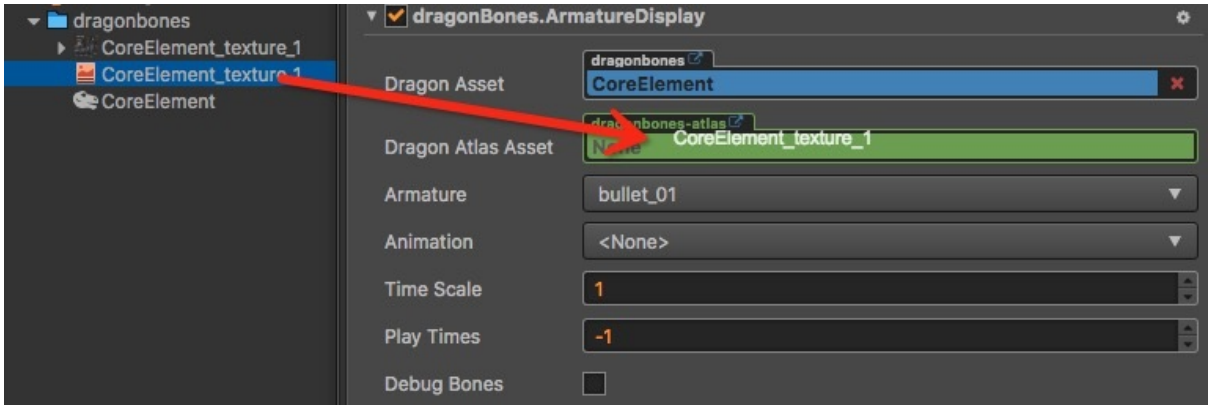

第二种方式：从 **资源管理器** 里将骨骼动画资源拖动到 **场景** 中:


第三种方式：从 **资源管理器** 里将骨骼动画资源拖动到已创建 DragonBones 组件的 Dragon Asset 属性中：




**2. 为 DragonBones 组件设置图集数据**

从 **资源管理器** 里将图集数据拖动到 DragonBones 组件的 Dragon Atlas Asset 属性中：




# 在项目中的存放

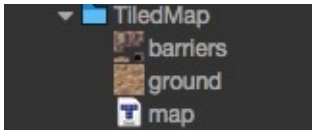
为了提高资源管理效率，建议将导入的资源文件存放在单独的目录下，不要和其他资源混在一起。

# 瓦片图资源（TiledMap）

## 导入地图资源

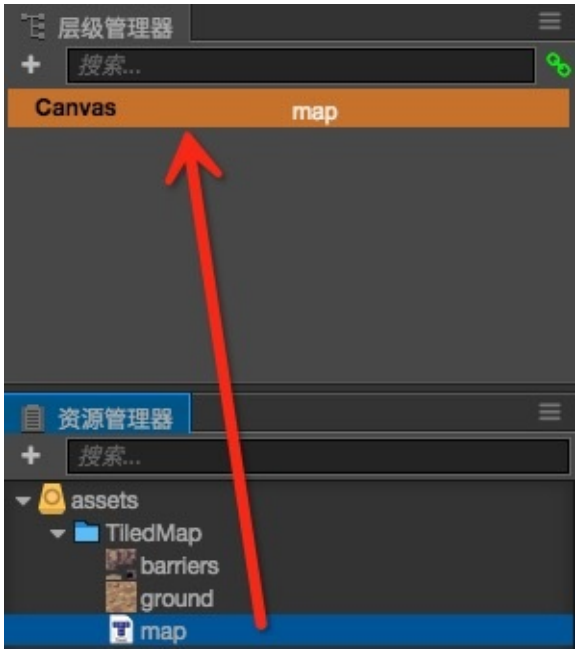
地图所需资源有：

- .tmx 地图数据
- .png 图集纹理
- .tsx tileset 数据配置文件（部分 tmx 文件需要）

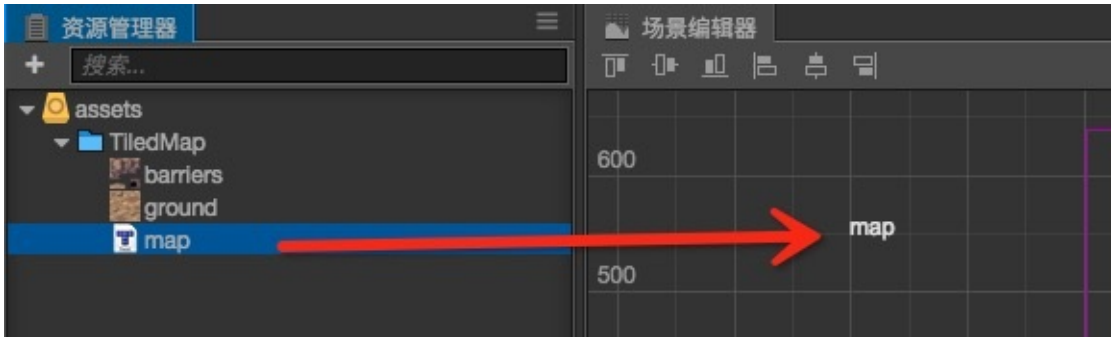



## 创建瓦片图资源

第一种方式：从 **资源管理器** 里将地图资源拖动到层级管理器中:




第二种方式：从 **资源管理器** 里将地图资源拖动到场景中:




第三种方式：从 **资源管理器** 里将地图资源拖动到已创建 TiledMap 组件的 Tmx File 属性中：


# 在项目中的存放

为了提高资源管理效率，建议将导入的 `tmx`，`tsx` 和 `png` 文件存放在单独的目录下，不要和其他资源混在一起。

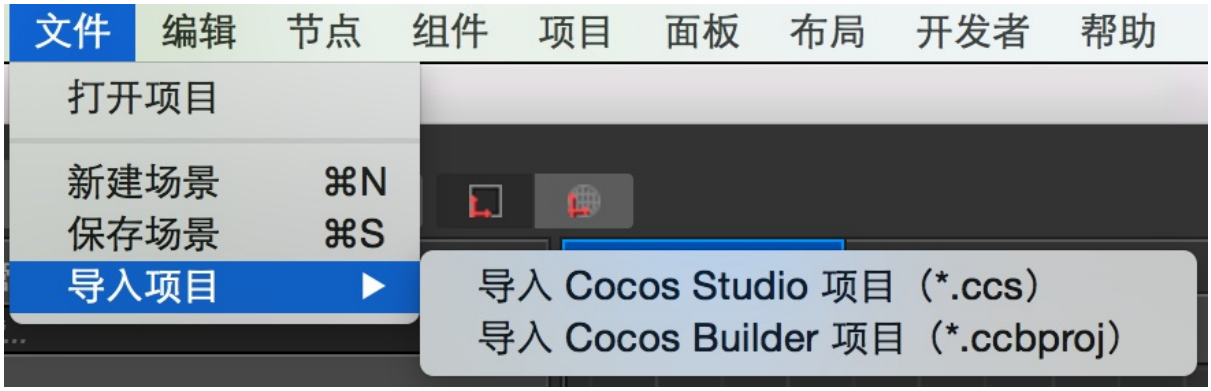
# 导入其他编辑器项目

## 简介

通过 Cocos Creator 主菜单中 `文件->导入项目` 的子菜单，可以导入其他编辑器中的项目。目前支持导入的编辑器项目有：

- Cocos Studio 项目（*.ccs 文件）
- Cocos Builder 项目（*.ccbproj 文件）

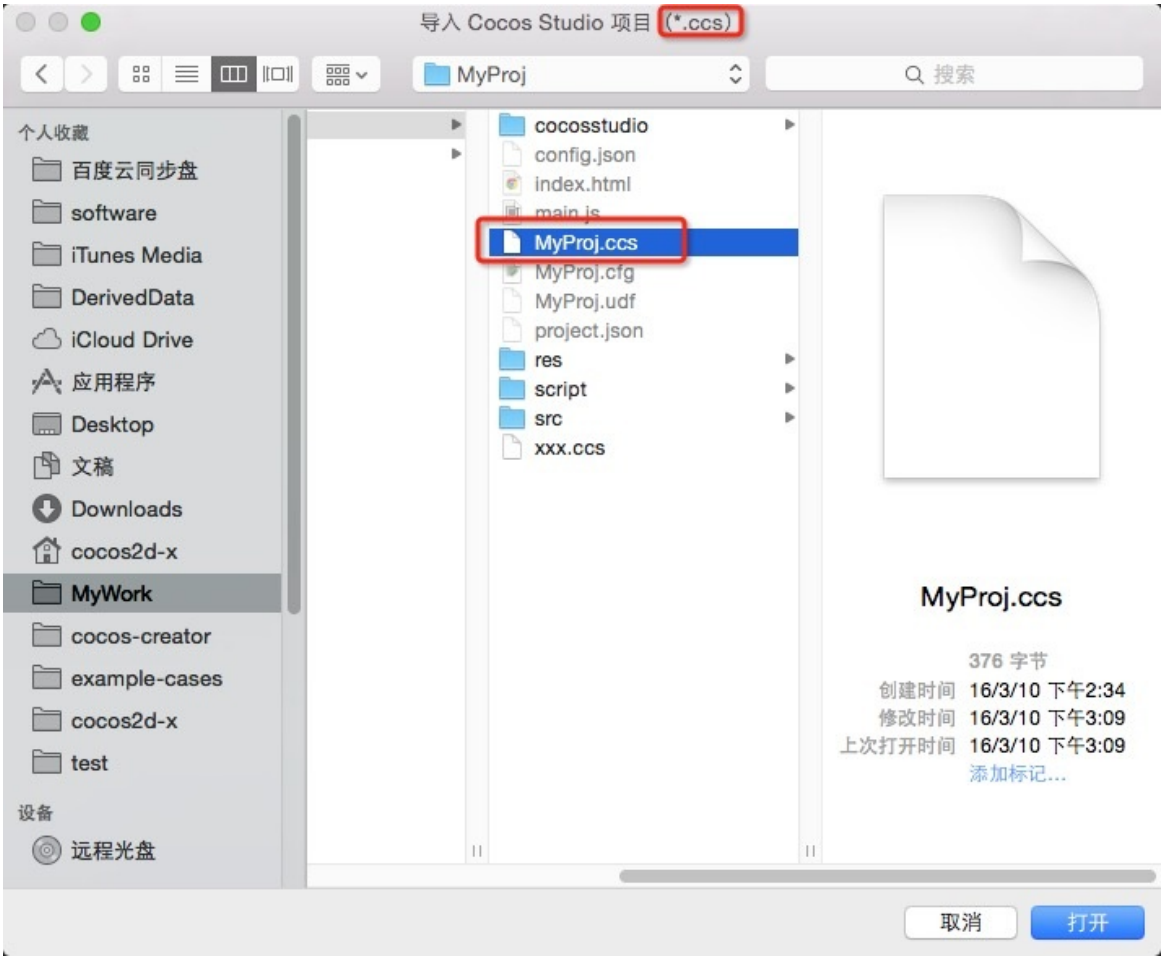
相应的菜单项如图：



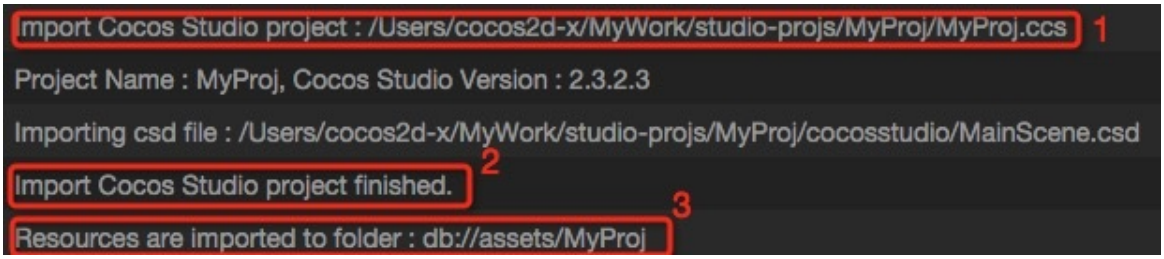

## 操作步骤说明

导入其他编辑器项目的操作步骤如下：

1. 点击相应的菜单，打开文件选择对话框。
2. 选择对应扩展名的文件，即可开始导入。如图：




3. 导入过程中 Cocos Creator 的控制台窗口会持续输出 log 以显示当前的导入进度。如下图：




输出的 log 说明：

- 首先输出导入的工程文件全路径。
- 中间持续输出正在导入的文件。
- 当导入完成时，会输出 `Import XXX project finished.` 的 log。（其中 XXX 为项目所用编辑器名称）
- 最后输出资源导入后所在的 url。

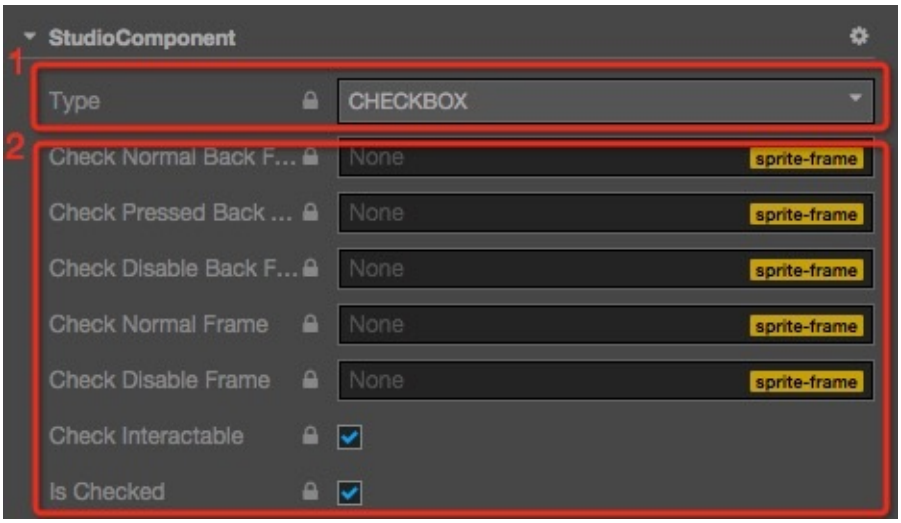
**备注：** 导入项目所需时间长短取决于项目大小。在导入过程中请勿在 Cocos Creator 中执行其他操作，耐心等待导入完成。

# Cocos Studio 项目导入说明

## 实现方案

- Cocos Studio 工程中的 csd 文件分为三类：
  - Scene ---- 导入为场景文件（.fire）

    - Layer ---- 导入为 prefab
    - Node ---- 导入为 prefab
- csd 文件中记录的节点帧动画数据，导入为 anim 文件。
- 导入后的目录结构：
    - Cocos Studio 工程导入后存放在 assets 目录下一个独立的文件夹中（文件夹以 Cocos Studio 工程名命名）。
    - Cocos Studio 工程中使用的资源文件以相同的目录结构导入到 Cocos Creator 工程中。
    - csd 文件中的帧动画数据存放在一个子目录中，子目录命名为 [csd文件名]_action
- 目前 Cocos Studio 中部分控件在 Cocos Creator 中暂不支持。这些控件在导入过程中，为相应的节点添加 StudioComponent 组件，并将控件数据保存在 StudioComponent 组件中。不支持的控件类型有：
    - 复选框
    - 艺术数字
    - 滑动条
    - 列表容器
    - 翻页容器
- StudioComponent 组件数据如下图：




其中 1 为组件的类型，对应于 Studio 中的控件；2 为控件数据，不同的类型显示相应的数据。

## 目前无法支持的情况

- 不支持骨骼动画数据的导入
- 不支持 csi 文件的导入（对应的图片以碎图的形式导入，而不是合图）
- 不支持节点的 SkewX 与 SkewY 属性以及相应的动画
- Particle 不支持 Blend Function 属性，同时 Sprite 和 Particle 的动画编辑中也不支持 Blend Function 属性的动画

## 特别说明

Cocos Studio 项目导入功能是基于 Cocos Studio 3.10 版本进行开发与测试的。如果要导入旧版本的项目，建议先使用 Cocos Studio 3.10 版本打开项目。这样可以将项目升级到对应版本，然后进行导入操作。

# Cocos Builder 项目导入说明

## 实现方案

- 所有的 ccb 文件都导入为 prefab。
- ccb 文件中的帧动画数据，导入为 anim 文件。
- 导入后的目录结构参考 Cocos Studio 项目导入的实现。

## 目前无法支持的情况

- CCControlButton 可以设置不同状态下文本的颜色，Cocos Creator 不支持
- 不支持渐变色的 Layer 节点
- 不支持节点的 skew 属性以及动画

# 场景制作工作流程

- 节点和组件
- 坐标系和变换
- 管理节点层级和显示顺序
- 使用场景编辑器搭建场景图像

在学习了上述基本的场景设置和搭建流程后，我们接下来会分三个部分分别介绍组成场景的各类元素，他们分别是：

- 图像和渲染元素
- UI 系统和控件
- 动画系统

请在完成本章后继续了解这些系统和元素。

---

继续前往 节点和组件 说明文档。

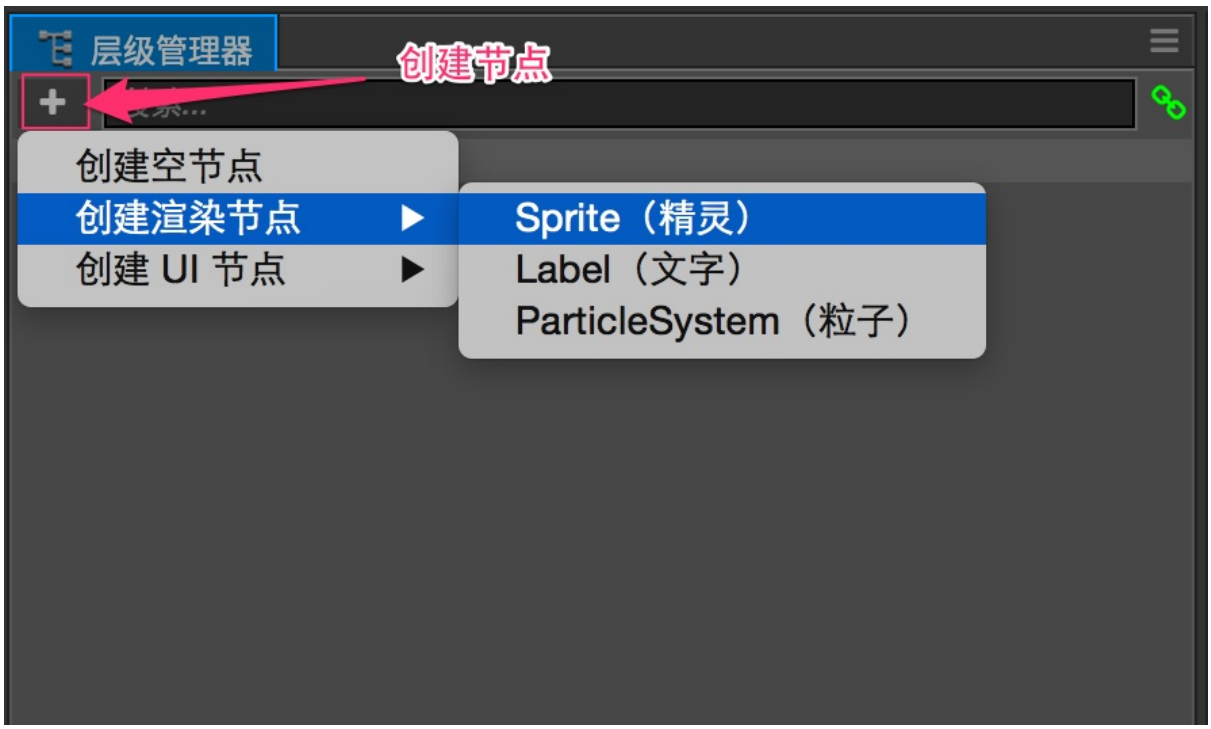
# 节点和组件

Cocos Creator 的工作流程是以组件式开发为核心的，组件式架构也称作 **组件-实体系统**（或 Entity-Component System），简单的说，就是以组合而非继承的方式进行实体的构建。

在 Cocos Creator 中，**节点（Node）**是承载组件的实体，我们通过将具有各种功能的 **组件（Component）** 挂载到节点上，来让节点具有各式各样的表现和功能。接下来我们看看如何在场景中创建节点和添加组件。

## 创建节点（Node）

要最快速的获得一个具有特定功能的节点，可以通过 **层级管理器** 左上角的 **创建节点** 按钮。我们以创建一个最简单的 Sprite（精灵）节点为例，点击 **创建节点** 按钮后选择 `创建渲染节点/Sprite（精灵）`：

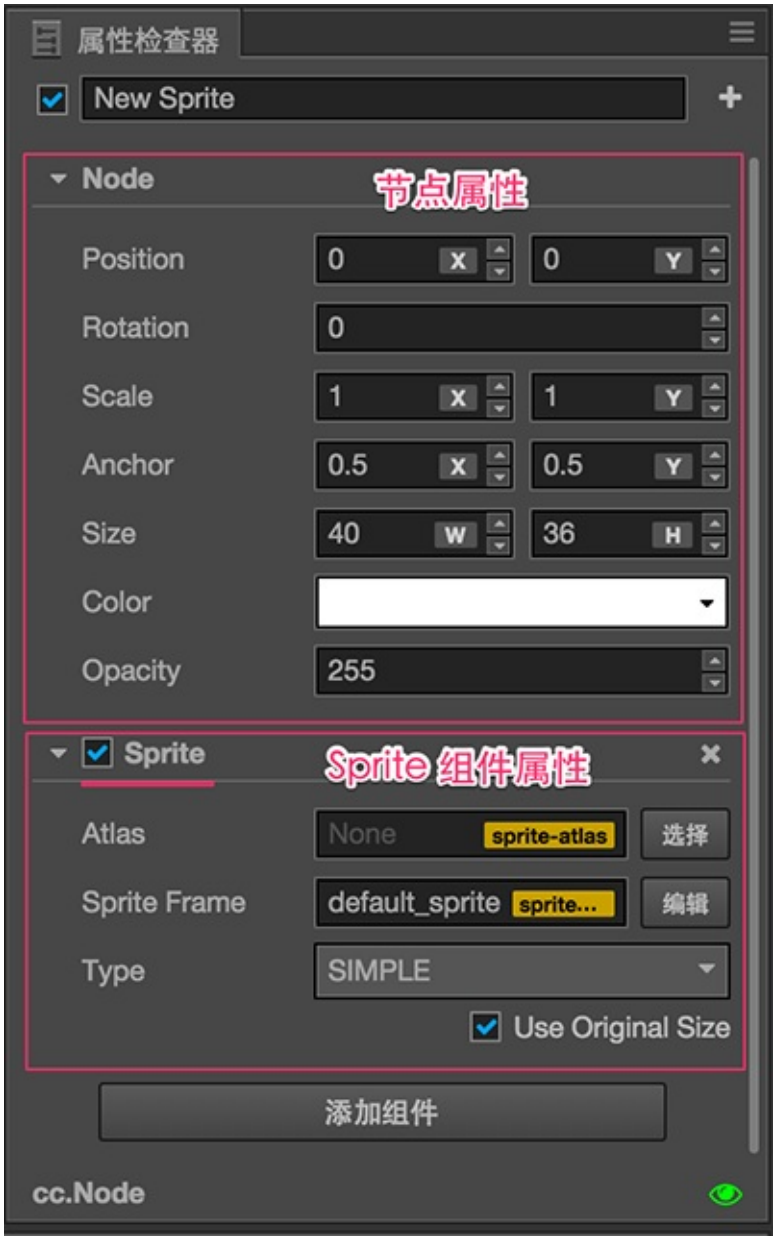



之后我们就可以在 **场景编辑器** 和 **层级管理器** 中看到新添加的 Sprite 节点了。新节点命名为 `New Sprite`，表示这是一个主要由 Sprite 组件负责提供功能的节点。您也可以尝试再次点击 **创建节点** 按钮，选择其他的节点类型，可以看到他们的命名和表现会有所不同。

## 组件（Component）

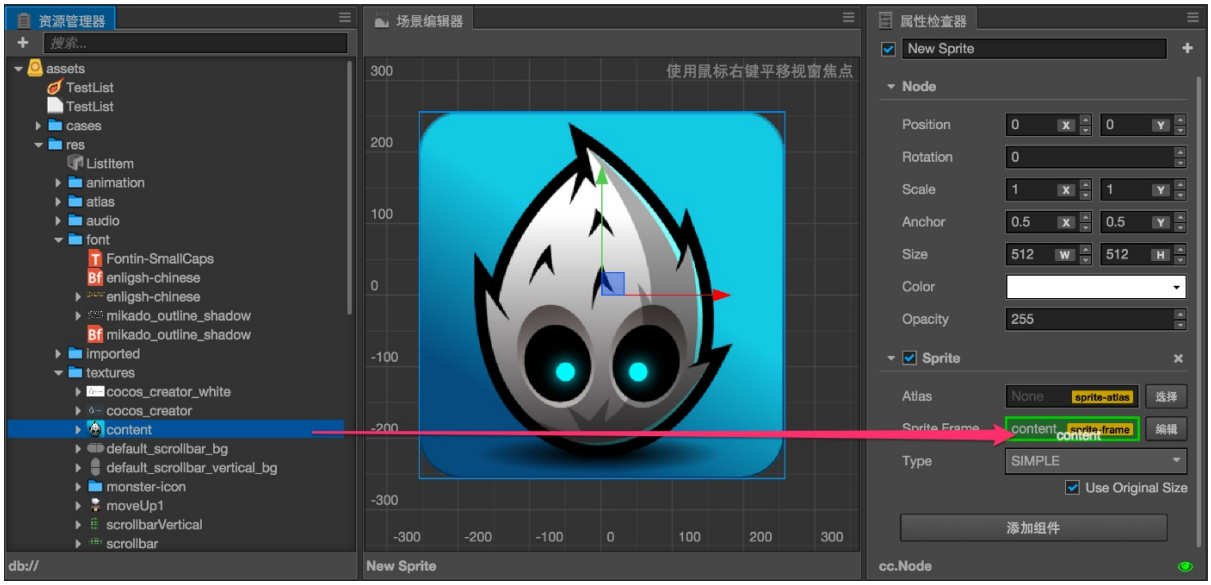
### Sprite 组件

我们有了一些节点，现在我们就来看看什么是组件，以及组件和节点的关系。选中我们刚才创建的 `New Sprite` 节点，可以看到 **属性检查器** 中的显示：




**属性检查器** 中以 `Node` 标题开始的部分就是节点的属性，节点属性包括了节点的位置、旋转、缩放、尺寸等变换信息和锚点、颜色、不透明度等其他信息。我们将在 搭建场景图像 部分进行详细介绍。

接下来以 `Sprite` 标题开始的部分就是 Sprite 组件的属性，在 2D 游戏中，Sprite 组件负责游戏中绝大部分图像的渲染。Sprite 组件最主要的属性就是 `Sprite Frame`，我们可以在这个属性指定 Sprite 在游戏中渲染的图像文件。让我们试试从 **资源管理器** 中拖拽任意一张图片资源到 **属性检查器** 的 `Sprite Frame` 属性中：

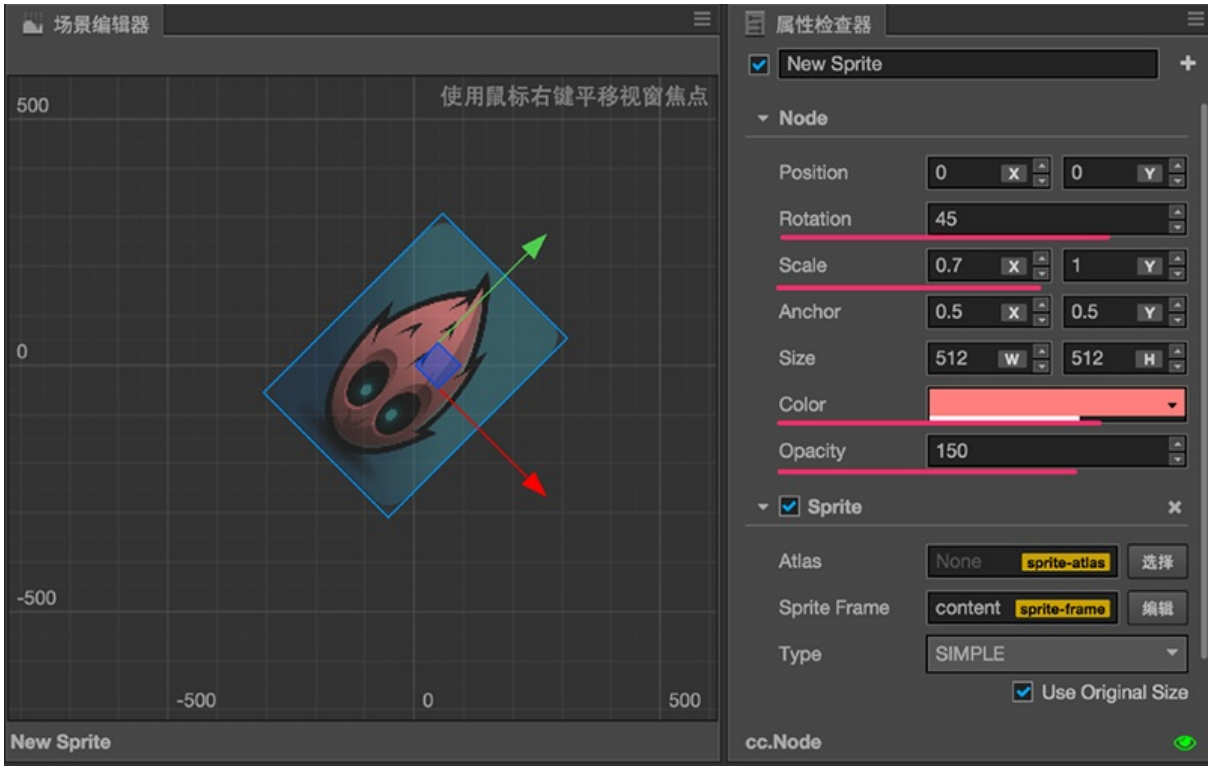



可以看到刚才的默认 Sprite 图片变成了我们指定的图片，这就是 Sprite 组件的作用：渲染图片。

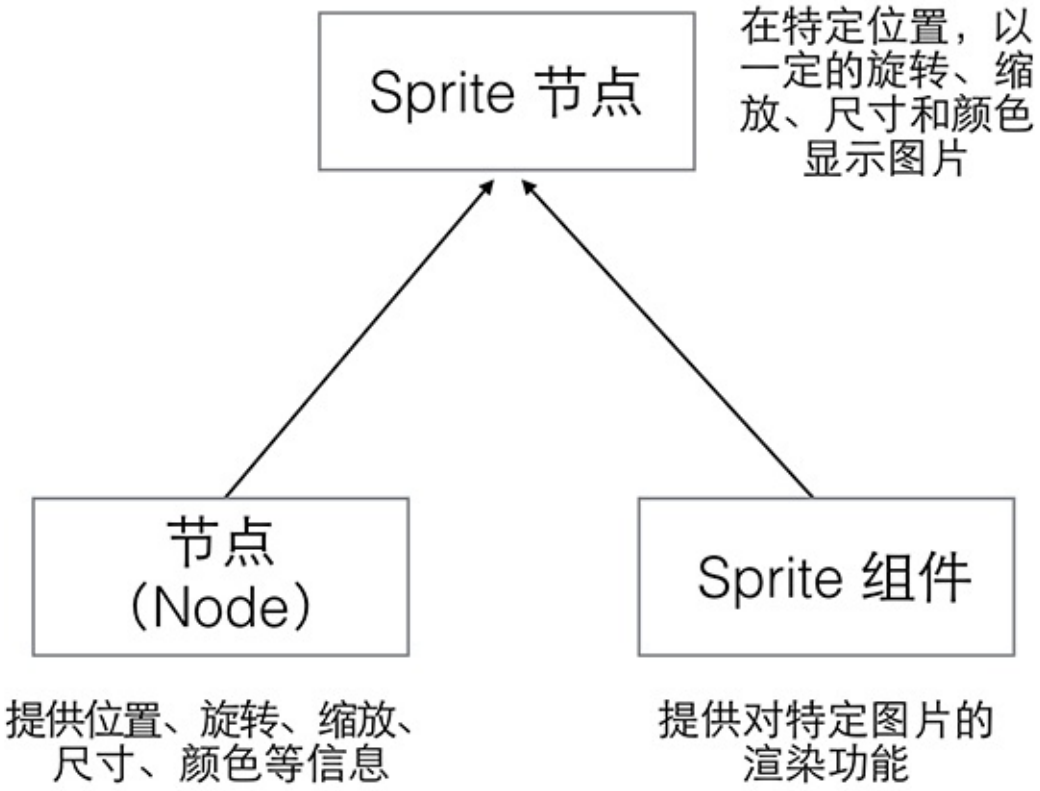
> 组件上设置好的任何资源，如这里的 SpriteFrame，都会在场景加载时自动同时加载好。你也可以在自定义的组件中声明需要设置和自动加载的资源类型，详见 获取和加载资源。

## 节点属性对 Sprite 组件的影响

节点和 Sprite 组件进行组合之后，就可以通过修改节点属性来控制图片渲染的方式，您可以按照下图中红线标记属性的设置对您的节点进行调整，可以看到图片的旋转、比例、颜色和不透明度都发生了变化：




我们前面提到了组件式的结构是以组合方式来实现功能的扩展的，下图中就展示了节点和 Sprite 组件的组合。




## 节点颜色（Color）和不透明度（Opacity）属性

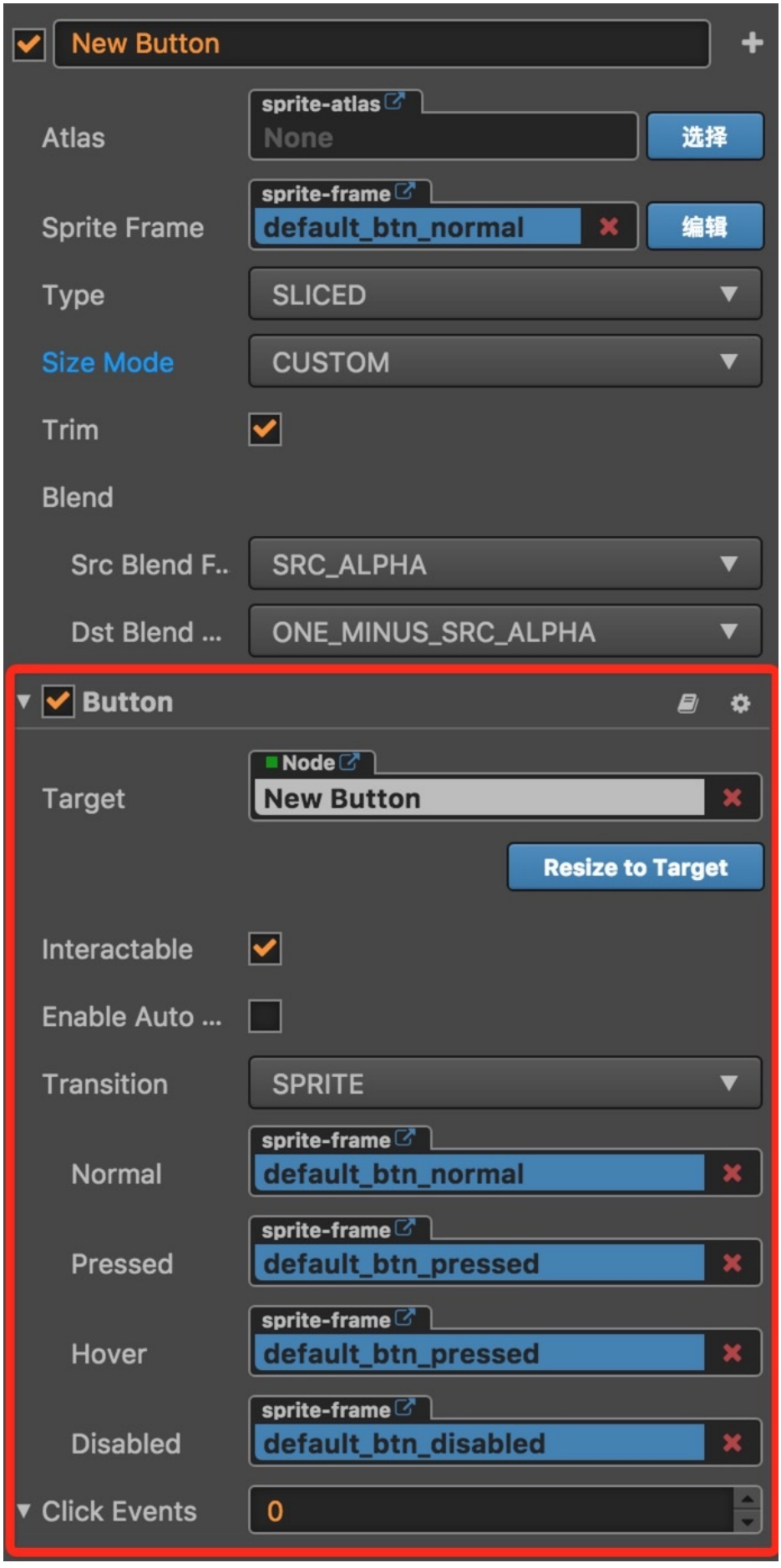
上图中节点的 **颜色（Color）** 属性和 **不透明度（Opacity）** 属性直接影响了 Sprite 组件对图片的渲染。颜色和不透明度同样会影响 文字（Label） 这样的渲染组件的显示。

这两个属性会和渲染组件本身的渲染内容进行相乘，来决定每个像素渲染时的颜色和不透明度。此外不透明度（Opacity）属性还会作用于子节点，可以通过修改父节点的 `Opacity` 轻松实现一组节点内容的淡入淡出效果。

## 添加其他组件

在一个节点上可以添加多个组件，来为节点添加更多功能。在上面的例子中，我们可以继续选中 `New Sprite` 这个节点，点击 **属性检查器** 面板下面的 **添加组件** 按钮，选择 `添加 UI 组件/Button` 来添加一个 Button（按钮）组件。

之后按照下图对 Button 组件的属性进行设置（具体的颜色属性可以根据爱好自由设置）：


New Button
Atlas
sprite-atlas
None
选择
Sprite Frame
sprite-frame
default_btn_normal
编辑
Type
SLICED
Size Mode
CUSTOM
Trim
Blend
Src Blend F..
SRC_ALPHA
Dst Blend ...
ONE_MINUS_SRC_ALPHA
Button
Target
Node
New Button
Resize to Target
Interactable
Enable Auto ...
Transition
SPRITE
Normal
sprite-frame
default_btn_normal
Pressed
sprite-frame
default_btn_pressed
Hover
sprite-frame
default_btn_pressed
Disabled
sprite-frame
default_btn_disabled
Click Events
0


接下来点击工具栏上面的 **运行预览** 按钮 ，并在浏览器运行窗口中将鼠标悬停在图片上，可以看到图片的颜色发生变化，也就是我们为节点添加的 Button 组件行为生效了！

# 小结

上面的例子里，我们先是将 Sprite 组件和节点组合，有了可以指定渲染图片资源的场景图像，接下来我们通过修改节点属性，能够对这个图像进行缩放和颜色等不同方式的显示。现在我们又为这个节点添加了 Button 组件，让这个节点具有了根据按钮的不同状态（普通、悬停、按下等）的行为。这就是 Cocos Creator 中组件式开发的工作流程，我们可以用这样的方式将不同功能组合在一个节点上，实现如主角的移动攻击控制、背景图像的自动卷动、UI 元素的排版和交互功能等等复杂目标。

值得注意的是，一个节点上只能添加一个渲染组件，渲染组件包括 **Sprite（精灵）**， **Label（文字）**，**Particle（粒子）**等。

# 参考阅读

- 理解组件-实体系统 - i_dovelemon的博客

# 坐标系和节点变换属性

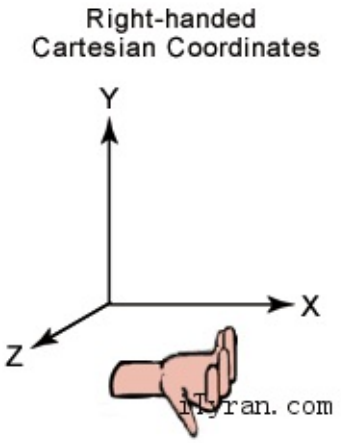
在 场景编辑器 和 节点和组件 文档中，我们介绍了可以通过 **变换工具** 和编辑 **属性检查器** 中节点的属性来变更节点的显示行为。这一节我们将会深入了解节点所在场景空间的坐标系，以及节点位置（Position）、旋转（Rotation）、缩放（Scale）、尺寸（Size）四大变换属性的工作原理。

## Cocos Creator 坐标系

我们已经知道可以为节点设置位置属性，那么一个有着特定位置属性的节点在游戏运行时将会呈现在屏幕上的什么位置呢？就好像日常生活的地图上有了经度和纬度才能进行卫星定位，我们也要先了解 Cocos Creator 的坐标系，才能理解节点位置的意义。

### 笛卡尔坐标系

Cocos Creator 的坐标系和 cocos2d-x 引擎坐标系完全一致，而 cocos2d-x 和 OpenGL 坐标系相同，都是起源于笛卡尔坐标系。笛卡尔坐标系中定义右手系原点在左下角，x 向右，y 向上，z 向外，我们使用的坐标系就是笛卡尔右手系。

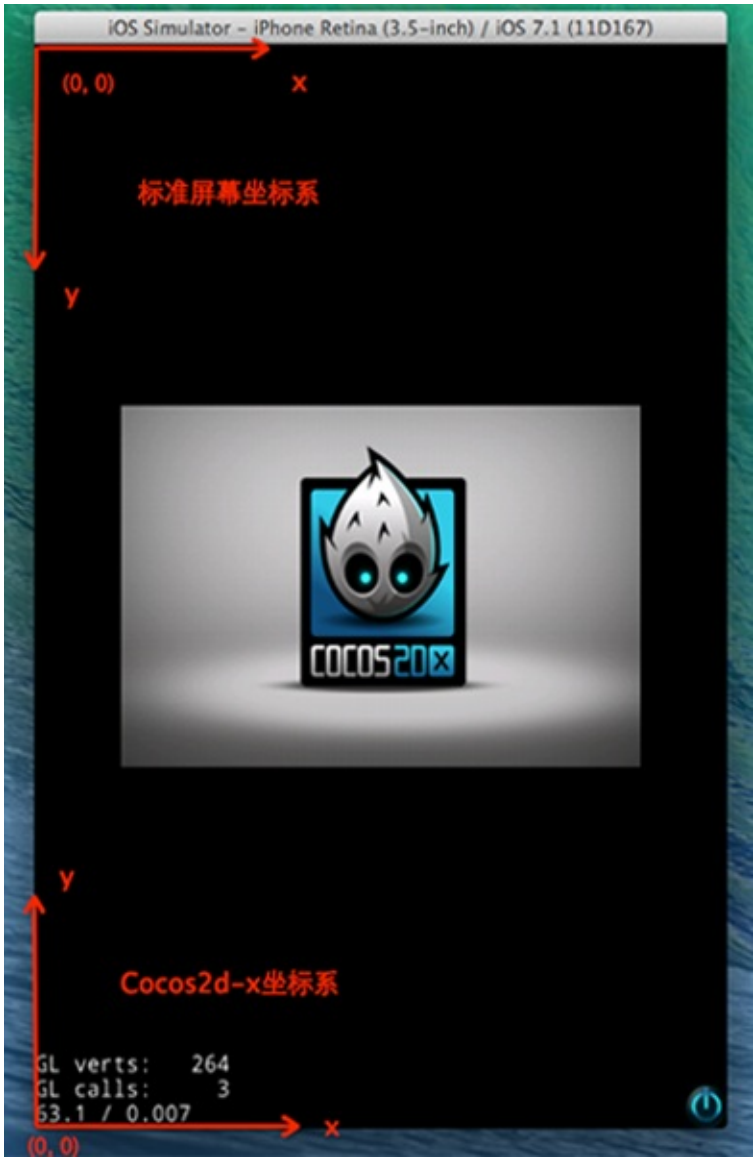



### 屏幕坐标系和 cocos2d-x 坐标系

标准屏幕坐标系使用和 OpenGL 不同的坐标系，和 cocos2d-x 坐标系有很大区别。

在 iOS, Android, Windows Phone 等平台用原生 SDK 开发应用时使用的是标准屏幕坐标系，原点为屏幕左上角，x 向右，y 向下。

Cocos2d-x 坐标系和 OpenGL 坐标系一样，原点为屏幕左下角，x 向右，y 向上。




## 世界坐标系（World Coordinate）和本地坐标系（Local Coordinate）

世界坐标系也叫做绝对坐标系，在 Cocos Creator 游戏开发中表示场景空间内的统一坐标体系，「世界」就用来表示我们的游戏场景。

本地坐标系也叫相对坐标系，是和节点相关联的坐标系。每个节点都有独立的坐标系，当节点移动或改变方向时，和该节点关联的坐标系将随之移动或改变方向。

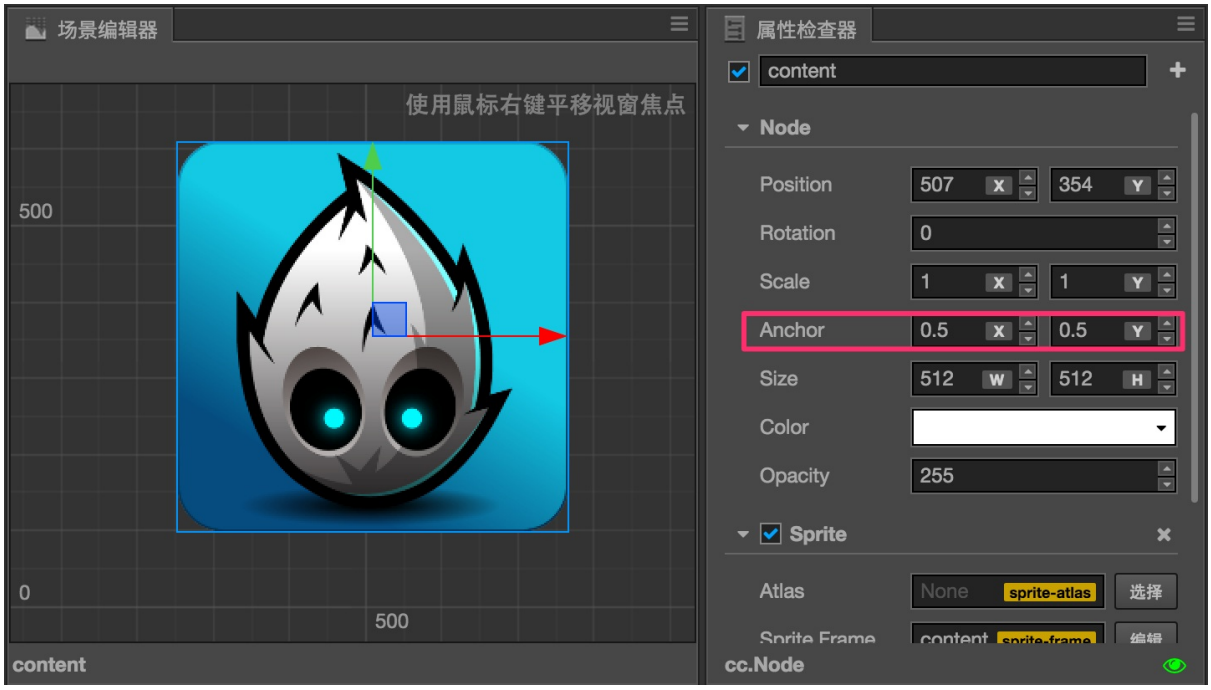
Cocos Creator 中的 **节点（Node）** 之间可以有父子关系的层级结构，我们修改节点的 **位置（Position）** 属性设定的节点位置是该节点相对于父节点的 **本地坐标系** 而非世界坐标系。最后在绘制整个场景时 Cocos Creator 会把这些节点的本地坐标映射成世界坐标系坐标。

要确定每个节点坐标系的作用方式，我们还需要了解 **锚点** 的概念。

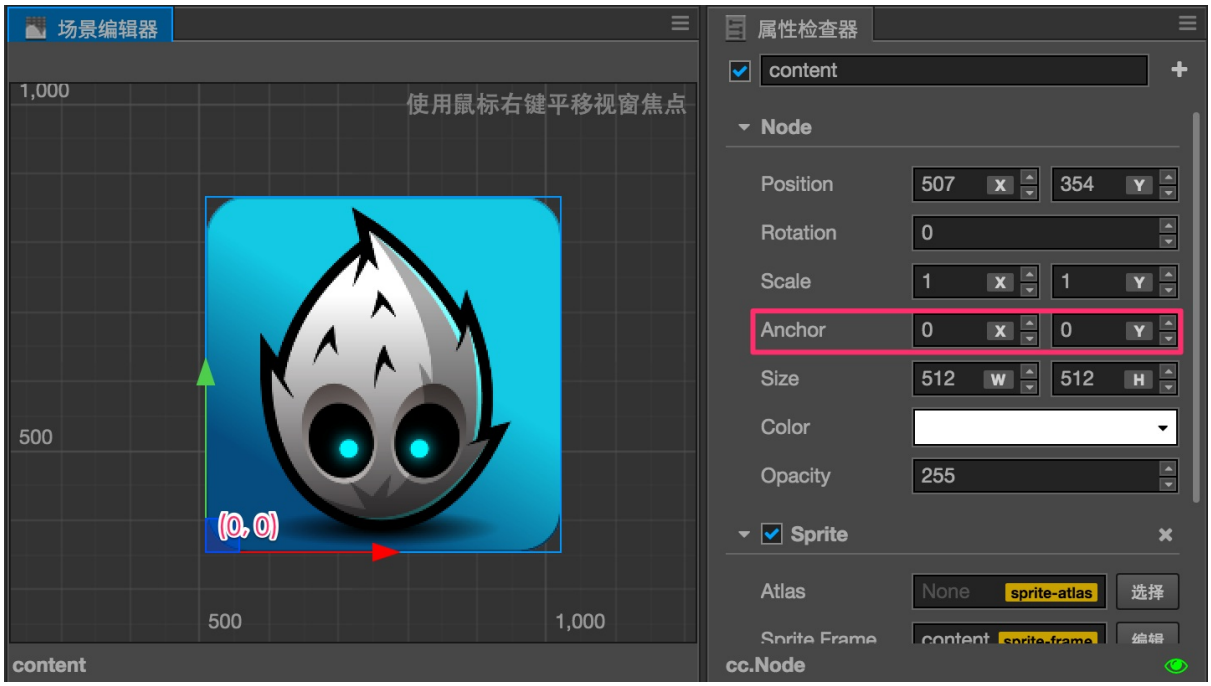
## 锚点（Anchor）

**锚点（Anchor）** 是节点的另一个重要属性，它决定了节点以自身约束框中的哪一个点作为整个节点的位置。我们选中节点后看到变换工具出现的位置就是节点的锚点位置。

锚点由 `anchorX` 和 `anchorY` 两个值表示，他们是通过节点尺寸计算锚点位置的乘数因子，范围都是 `0 ~ 1` 之间。`(0.5, 0.5)` 表示锚点位于节点长度乘 0.5 和宽度乘 0.5 的地方，即节点的中心。




锚点属性设为 `(0, 0)` 时，锚点位于节点本地坐标系的初始原点位置，也就是节点约束框的左下角。




## 子节点的本地坐标系

锚点位置确定后，所有子节点就会以 **锚点所在位置** 作为坐标系原点，注意这个行为和 cocos2d-x 引擎中的默认行为不同，是 Cocos Creator 坐标系的特色！

假设场景中节点的结构如下图所示：


当我们的场景中包含不同层级的节点时，我们按照以下的流程确定每个节点在世界坐标系下的位置：

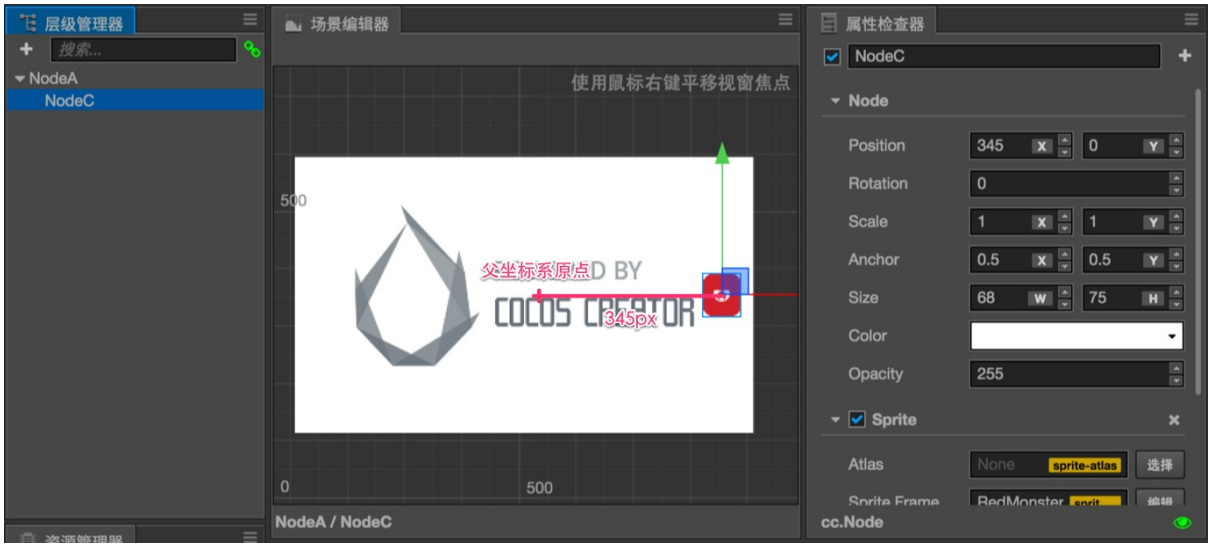
1. 从场景根级别开始处理每个节点，上图中 `NodeA` 就是一个根级别节点。首先根据 NodeA 的 **位置（Position）** 属性和 **锚点（Anchor）** 属性，在世界坐标系中确定 NodeA 的显示位置和坐标系原点位置（和锚点位置一致）。
2. 接下来处理 NodeA 的所有直接子节点，也就是上图中 `NodeB` 以及和 NodeB 平级的节点。根据 NodeB 的位置和锚点属性，在 NodeA 的本地坐标系中确定 NodeB 在场景空间中的位置和坐标系原点位置。
3. 之后不管有多少级节点，都继续按照层级高低依次处理，每个节点都使用父节点的坐标系和自身位置锚点属性来确定在场景空间中的位置。

# 变换属性

除了上面介绍过的 **锚点（Anchor）** 之外，节点还包括四个主要的变换属性，下面我们依次介绍。


## 位置（Position）

**位置（Position）** 由 `x` 和 `y` 两个属性组成，分别规定了节点在当前坐标系 x 轴和 y 轴上的坐标。



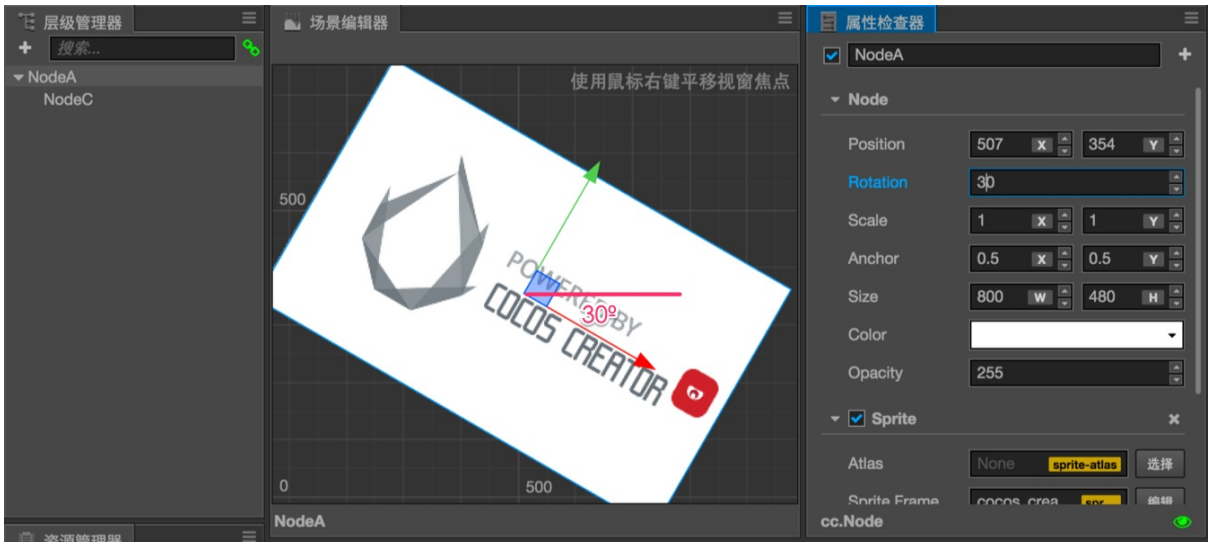

上图中节点 `NodeA` 位于场景根级别，他的位置是 `(507, 354)`（可以参考 **场景编辑器** 背景的刻度显示），其子节点 `NodeC` 的位置是 `(345, 0)`，可以看到子节点的位置是以父节点锚点为基准来偏移的。

位置属性的默认值是 `(0, 0)`，也就是说，新添加节点时，节点总会出现在父节点的坐标系原点位置。Cocos Creator 中节点的默认位置为 `(0, 0)`，默认锚点设为 `(0.5, 0.5)`。这样子节点会默认出现在父节点的中心位置，在制作 UI 或组合玩家角色时都能够对所有内容一览无余。

在场景编辑器中，可以随时使用 移动变换工具 来修改节点位置。

## 旋转（Rotation）

**旋转（Rotation）** 是另外一个会对节点本地坐标系产生影响的重要属性，旋转属性只有一个值，表示节点当前的旋转角度。角度值为正时，节点顺时针旋转，角度值为负时，节点逆时针旋转。

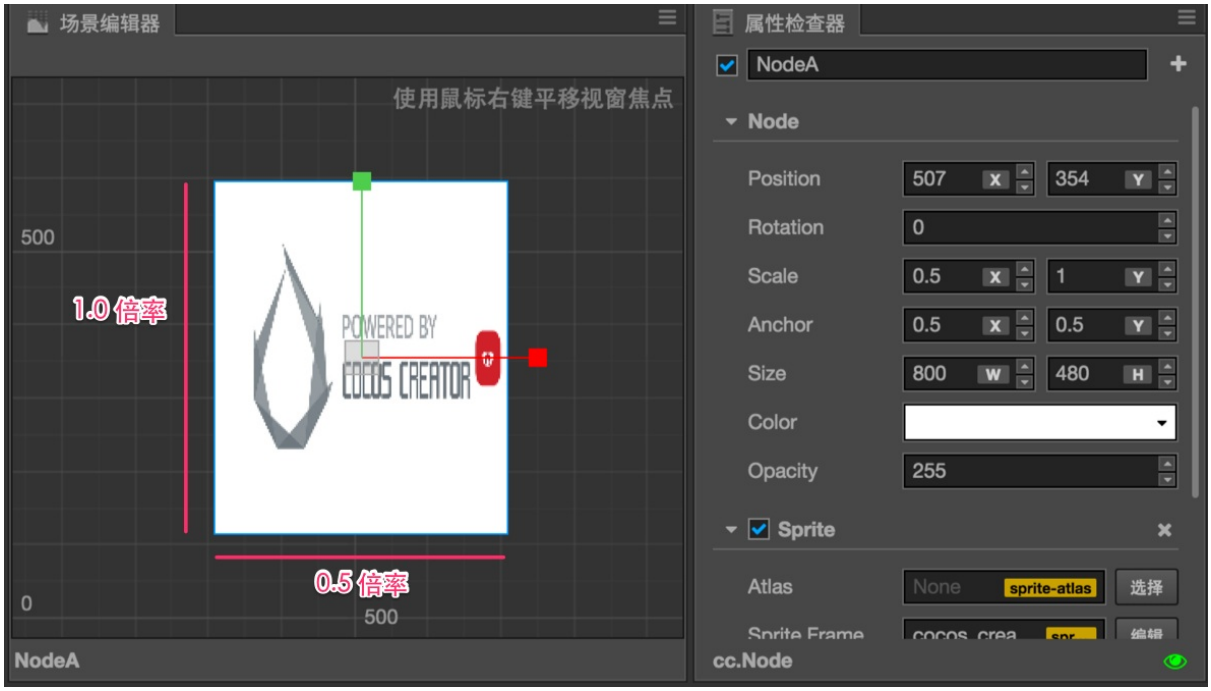



上图所示的节点层级关系和前一张图相同，只是节点 `NodeA` 的 **旋转（Rotation）** 属性设为了 `30` 度，可以看到除了 `NodeA` 本身顺时针旋转了 30 度之外，其子节点 `NodeC` 也以 `NodeA` 的锚点为中心，顺时针旋转了 30 度。

在场景编辑器中，可以随时使用 旋转变换工具 来修改节点旋转。

## 缩放（Scale）

**缩放（Scale）** 属性也是一组乘数因子，由 `scaleX` 和 `scaleY` 两个值组成，分别表示节点在 x 轴和 y 轴的缩放倍率。




上图中节点 `NodeA` 的缩放属性设为 `(0.5, 1.0)`，也就是将该节点在 x 轴方向缩小到原来的 0.5 倍，y 轴保持不变。可以看到子节点 `NodeC` 图像也在 x 轴方向上进行了压缩，所以缩放属性会影响所有子节点。

子节点上设置的缩放属性会和父节点叠加作用，子节点的子节点会将每一层级的缩放属性全部相乘来获得在世界坐标系下显示的缩放倍率，这一点和位置、旋转其实是一致的，只不过位置和旋转属性是相加作用，只有缩放属性是相乘，作用表现的更明显。

缩放属性是叠加在位置、尺寸等属性上作用的，修改缩放属性时，节点的位置和尺寸不会变化，但显示节点图像时会先将位置和尺寸等属性和缩放相乘，得出的数值才是节点显示的真实位置和大小。

在场景编辑器中，可以随时使用 缩放变换工具 来修改节点缩放。

## 尺寸（Size）

**尺寸（Size）** 属性由 `Width`（宽度）和 `Height`（高度）两个值组成，用来规定节点的约束框大小。对于 Sprite 节点来说，约束框的大小也就相当于显示图像的大小。

因此尺寸属性很容易和缩放属性混淆，两者都会影响 Sprite 图像的大小，但他们是通过不同的方式来影响图像实际显示大小的。尺寸属性和位置、锚点一起，规定了节点四个顶点所在位置，并由此决定由四个顶点约束的图像显示的范围。尺寸属性在渲染 九宫格图像（Sliced Sprite） 时有至关重要的作用。

而缩放属性是在尺寸数值的基础上进行相乘，得到节点经过缩放后的宽度和高度。可以说在决定图像大小时，尺寸是基础，缩放是变量。另外尺寸属性不会直接影响子节点的尺寸（但可以通过 对齐挂件（Widget）间接影响），这一点和缩放属性有很大区别。

在场景编辑器中，可以随时使用 矩形变换工具 来修改节点尺寸。

# 管理节点层级和显示顺序

通过前面的内容，我们了解了通过节点和组件的组合，能够在场景中创建各种图像、文字和交互元素。当场景中的元素越来越多时，我们就需要通过节点层级来将节点按照逻辑功能归类，并按需要排列他们的显示顺序。

## 了解层级管理器

创建和编辑节点时，**场景编辑器** 可以展示直观的可视化场景元素。而节点之间的层级关系则需要使用 **层级管理器** 来检查和操作。请先阅读层级管理器面板介绍，来掌握 **层级管理器** 的使用方法。

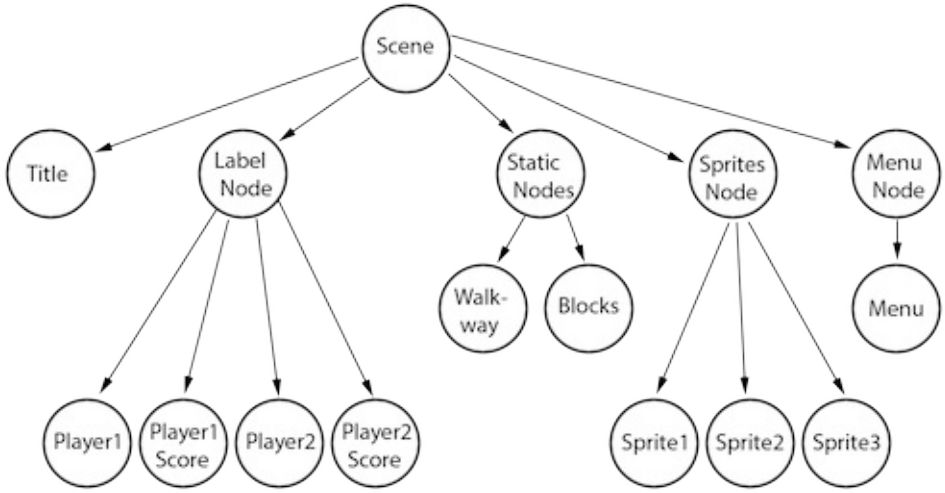
## 节点树

通过层级管理器或运行时脚本的操作，建立的节点之间的完整逻辑关系，就叫做节点树。

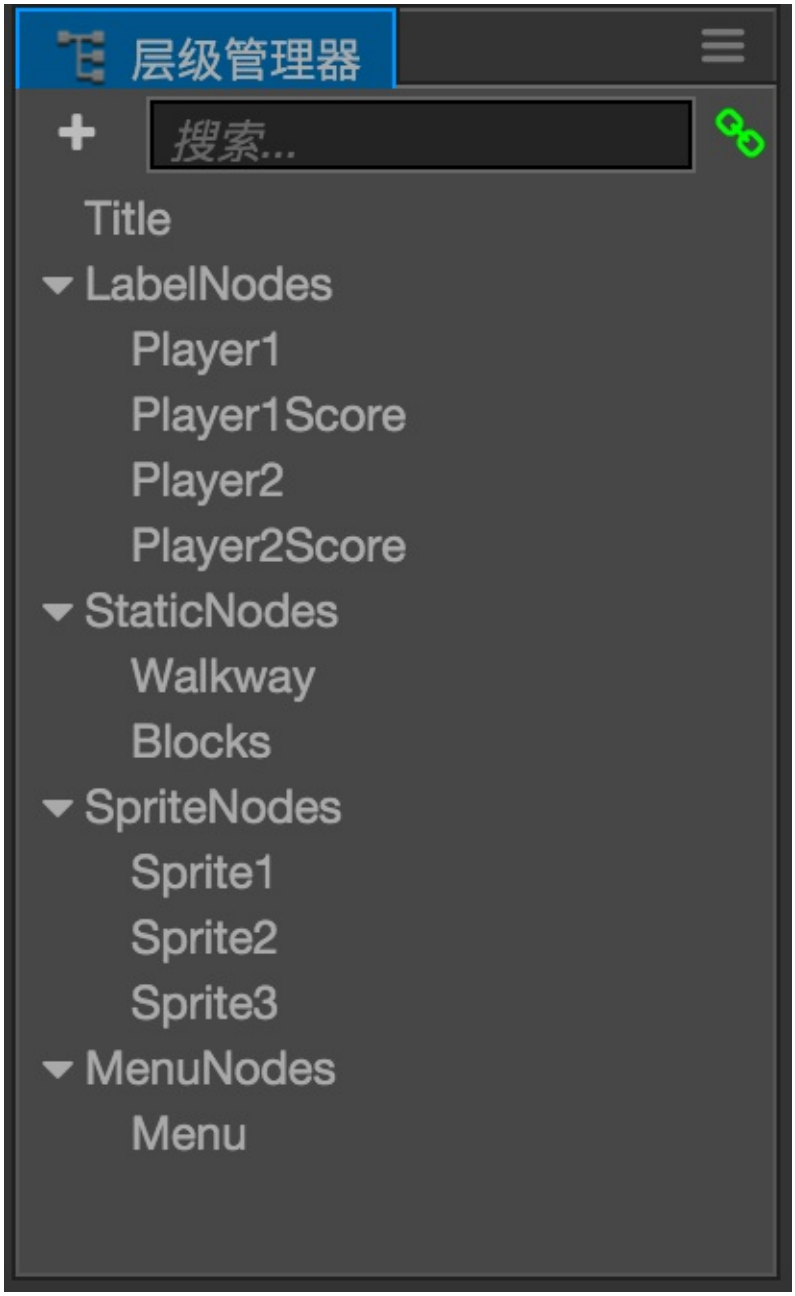
我们用来自 Cocos2d-x Programmer Guide 的一组图片来展示什么是节点树：

下面是一个简单的游戏场景，包括背景图像，三个角色，标题文字，分数文字和开始游戏的按钮：


每个视觉元素都是一个节点，通常我们不会把所有节点平铺在场景上，而是会按照一定的分类和次序组织成如下图所示的节点树：




节点树中由箭头连接的两个节点之间就是父子关系，我们把显示在上面的叫做父节点，下面的叫子节点。在 **层级管理器** 中，上面的节点树就会是这个样子：

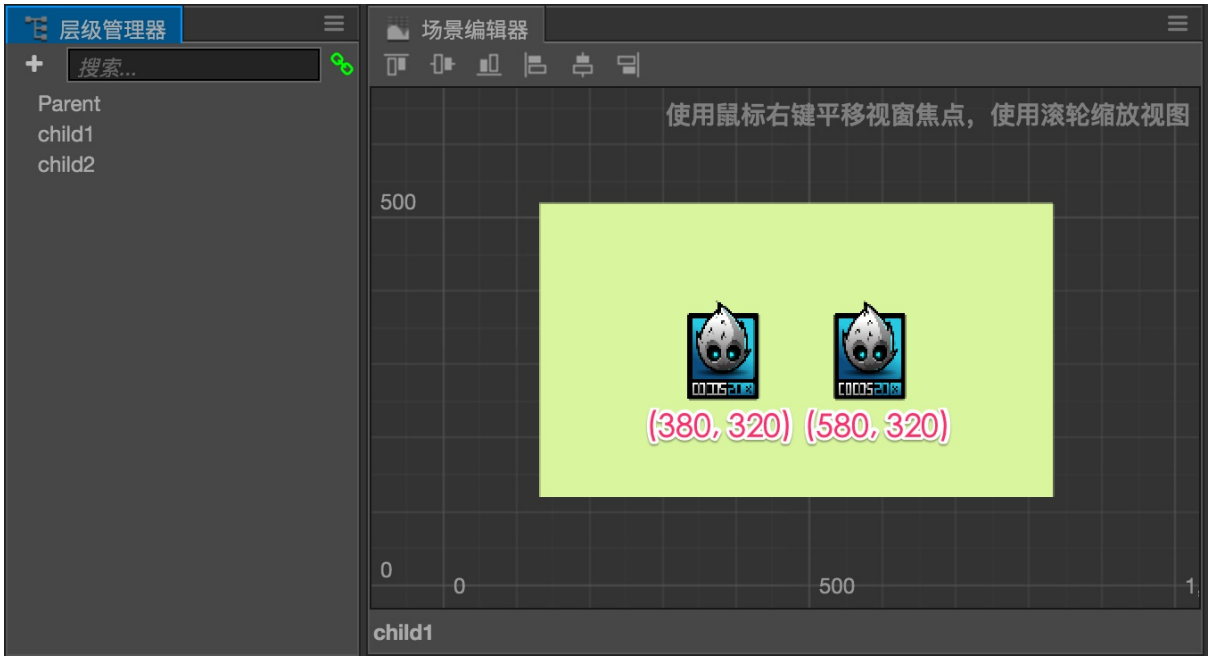



这就是如果依照类别创建几个父节点，然后把同类节点放在一个父节点下构建出的节点树。在实际游戏项目中我们还可以根据需要用其他各种方式组织节点树，下面我们就来仔细看看节点树和节点父子关系的实际作用，以及我们如何组织节点树的策略。

## 节点本地坐标系

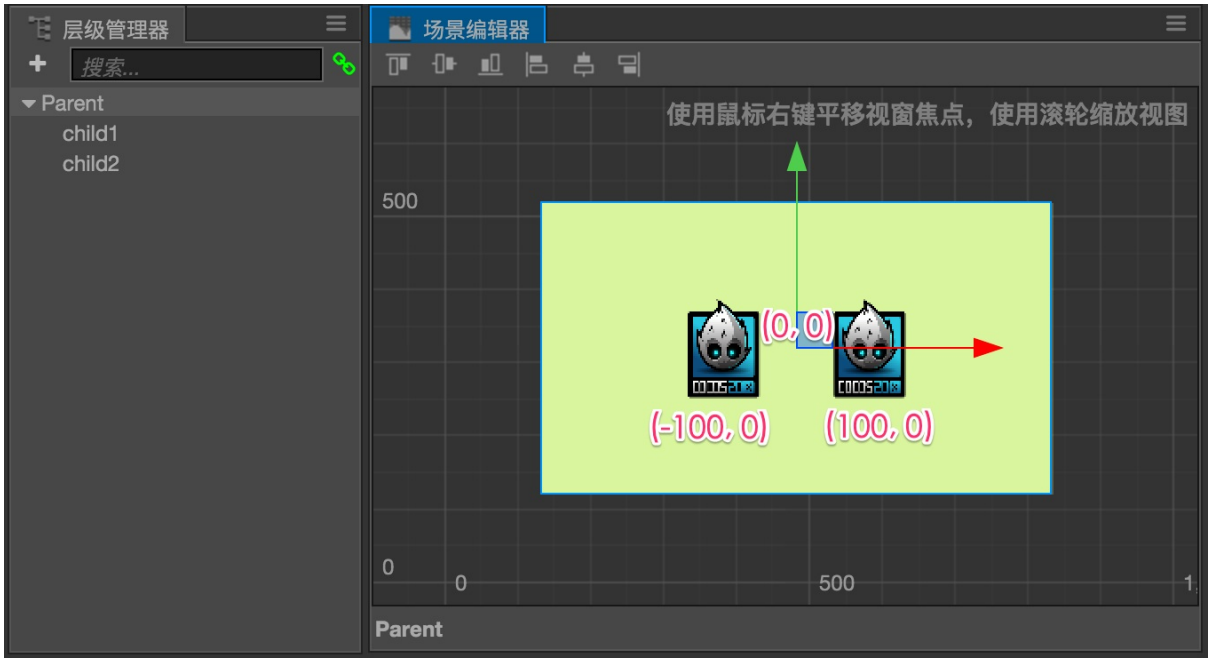
在前一节的本地坐标系相关介绍中，我们了解了节点父子关系的重要作用之一就是让我们能够在 **本地坐标系** 下使用子节点的变换属性。

我们知道世界坐标系的原点是屏幕左下角，如果我们场景中所有节点都是平行排列，当我们需要将两个节点放在背景节点上面比较靠近中间的位置时，我们可以看到节点的 **位置** 属性如下图所示：



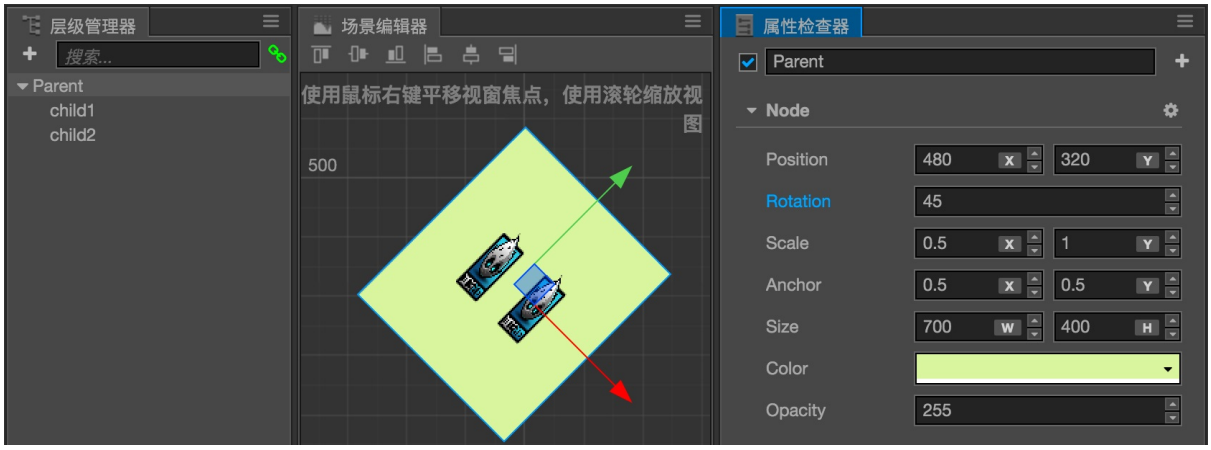

由于两个主要节点和背景节点没有任何关系，因此他们的位置都是在世界坐标系下的数值，基本上没有规律，当我们需要让节点在背景范围内移动时，要计算出节点新的位置也需要动一番脑筋。

下面我们再来看看借助节点父子关系和本地坐标系，我们把两个主要节点拖拽到 `Parent` 节点下面作为子节点，这时两个节点的 **位置** 属性会变成怎样：




由于 `Parent` 节点的锚点属性是 `(0.5, 0.5)`，也就是以中心点作为本地坐标系原点，所以靠近父物体中心摆放的两个子节点的位置现在变成了 `(-100, 0)` 和 `(100, 0)`，使用本地坐标系的位置信息能够直观的反应两个子节点的摆放逻辑，也就是「靠近背景中心左右对称摆放」。这样的工作方式能够更直接的体现设计师在搭建场景时的想法，在后续让节点在背景范围内运动的过程中，也更容易获得边界范围，比如父节点最右边的本地坐标，就是 `(parentNode.width/2, 0)`。

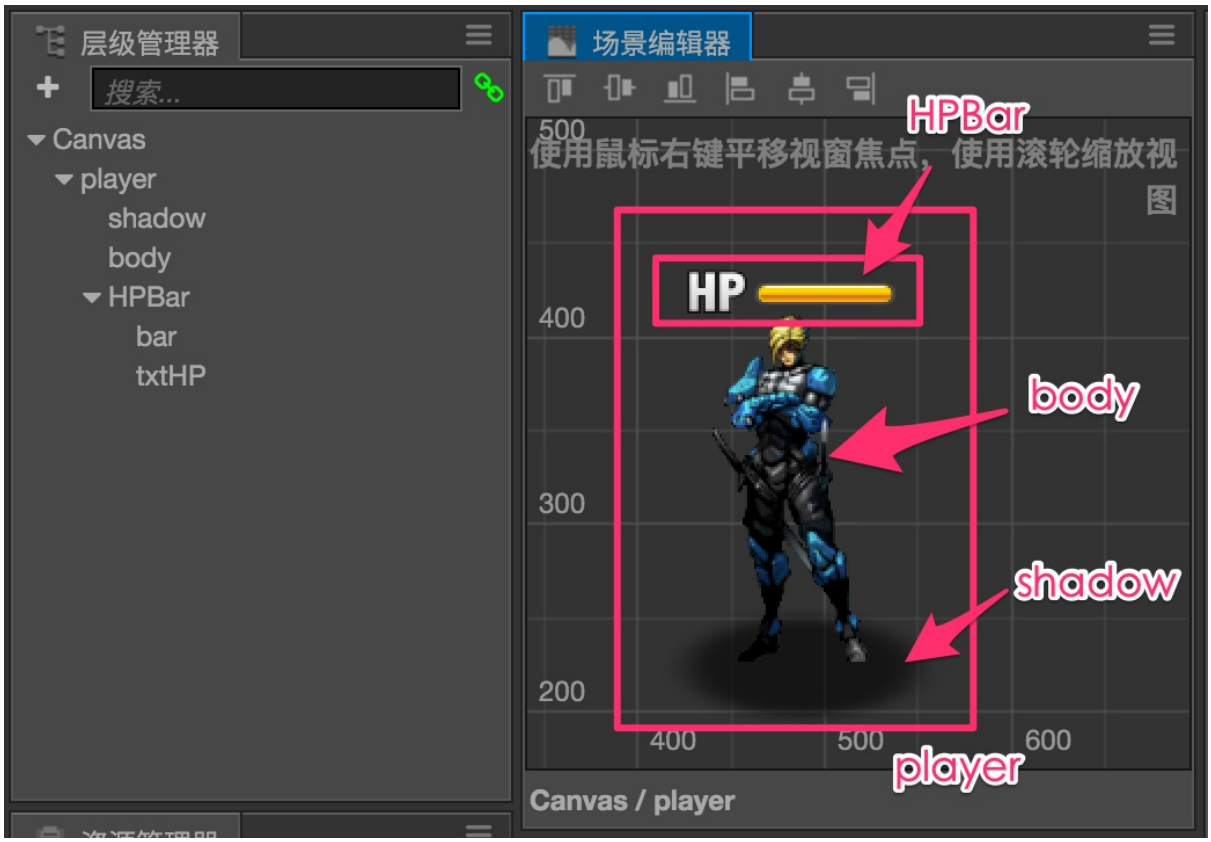
另外当需要将一组节点作为一个整体进行移动、缩放、旋转时，节点的父子关系也可以让我们只关心父节点的变换操作，而不需要再去对子节点进行一一的遍历和计算。下图就是我们把上面例子里的父节点进行旋转和缩放的结果，可以看到子节点像印在父节点上一样，和父节点一起进行变换。




我们经常会遇到由很多节点组合成的复杂角色，游戏控制这些角色互动时，就需要这种基于父节点整体变换的功能。下面我们就来看看都有哪些基于逻辑关系的节点树管理方式。

## 管理节点逻辑关系

在游戏中经常需要控制复杂的玩家角色，这种角色通常不会只由单个节点组成，我们看看下面这张图里的英雄角色，就由三个不同的部分组成。




我们将英雄角色的 Sprite 图像显示和帧动画组件放在 `body` 节点上，然后需要跟随角色移动的阴影 Sprite 单独拿出来作为 `shadow` 节点。最后把负责生命值显示的进度条作为一组独立功能的节点，形成自己的迷你节点树 `HPBar`。

上面的例子就是典型的根据逻辑需要来组织节点关系，我们可以根据游戏逻辑操作英雄角色节点的动画播放、左右翻转；根据角色当前血量访问 `HPBar` 节点来更新生命值显示；最后他们共同的父节点 `player` 用于控制角色的移动，并且可以作为一个整体被添加到其他场景节点中。

## 管理节点渲染顺序

在上面的例子中，我们可以注意一下 `body` 和 `shadow` 节点的排列顺序，在 **层级管理器** 中会按照节点排列顺序依次渲染，也就是显示在列表上面的节点会被下面的节点遮盖住。`body` 节点在列表里出现在下面，因此实际渲染时会挡住 `shadow` 节点。

我们可以看到父节点永远是出现在子节点上面的，因此子节点永远都会遮盖住父节点，这点需要特别注意。这也是我们为什么必须把英雄角色 Sprite 单独分离出来作为 `body` 节点的原因，因为如果英雄的 Sprite 处在 `player` 节点上，我们就无法使英雄图像挡住他脚下的阴影了。

# 性能考虑

注意，虽然前面我们说父节点可以用来组织逻辑关系甚至是当做承载子节点的容器，但节点数量过多时，场景加载速度会受影响，因此在制作场景时应该避免出现大量无意义的节点，应该尽可能合并相同功能的节点。

# 使用场景编辑器搭建场景图像

本文将介绍使用 **场景编辑器** 创建和编辑场景图像时的工作流程和技巧。

# 使用 Canvas 作为渲染根节点

在开始添加节点之前，我们先简单了解一下新建场景后默认存在的 Canvas 节点的作用，以及我们如何从这里开始场景的搭建。

Canvas 节点是我们推荐大家使用的 **渲染根节点**，这个的意思就是希望大家将所有渲染相关的节点都放在 Canvas 下面，这样做有以下好处：

- Canvas 能提供多分辨率自适应的缩放功能，以 Canvas 作为渲染根节点能够保证我们制作的场景在更大或更小的屏幕上都保持较好的图像效果，详见多分辨率适配方案相关文档。
- Canvas 的默认锚点位置是 `(0.5, 0.5)`，加上 Canvas 节点会根据屏幕大小自动居中显示，所以 Canvas 下的节点会以屏幕中心作为坐标系的原点。根据我们的经验，这样的设置会简化场景和 UI 的设置（比如让按钮元素的文字默认出现在按钮节点的正中），也能让控制节点位置的脚本更容易编写。

如果您不希望使用我们默认配置的几种屏幕适配方案，或者不希望以屏幕中心作为坐标系原点，也可以直接删除 Canvas 节点，并使用您偏好的设置策略。

## 逻辑节点的归属

除了有具体图像渲染任务的节点之外，我们还会有一部分节点只负责挂载脚本，执行逻辑，不包含任何渲染相关内容。通常我们将这些节点放置在场景根层级，和 Canvas 节点并列，如下图所示：


可以看到除了 Canvas 下的背景、菜单、玩家角色等节点之外，我们还将包含有游戏主逻辑组件的 `Game` 节点放在了和 Canvas 平行的位置上，方便协作的时候其他开发者能够第一时间找到游戏逻辑和进行相关的数据绑定。

# 使用节点创建菜单快捷添加基本节点类型

当我们需要开始为场景添加内容时，一般会先从 **层级管理器** 的 **创建节点菜单** 开始，也就是点击左上角的 `+` 按钮弹出的菜单。这个菜单的内容和主菜单中 `节点` 菜单里的内容一致，都可以从几个简单的节点分类中选择我们需要的基础节点类型并添加到场景中。

添加节点时，在 **层级管理器** 中选中的节点将成为新建节点的父节点，如果你选中了一个折叠显示的节点然后通过菜单添加了新节点，记得要展开刚才选中的节点才能看到新加的节点哦！

## 空节点

选择 **创建节点菜单** 里的 `创建空节点` 就能够创建一个不包含任何组件的节点。空节点可以作为组织其他节点的容器，也可以用来挂载用户编写的逻辑和控制组件。另外在下文中我们也会介绍通过空节点和组件的组合，创造符合自己特殊要求的渲染控件的用法。

## 渲染节点

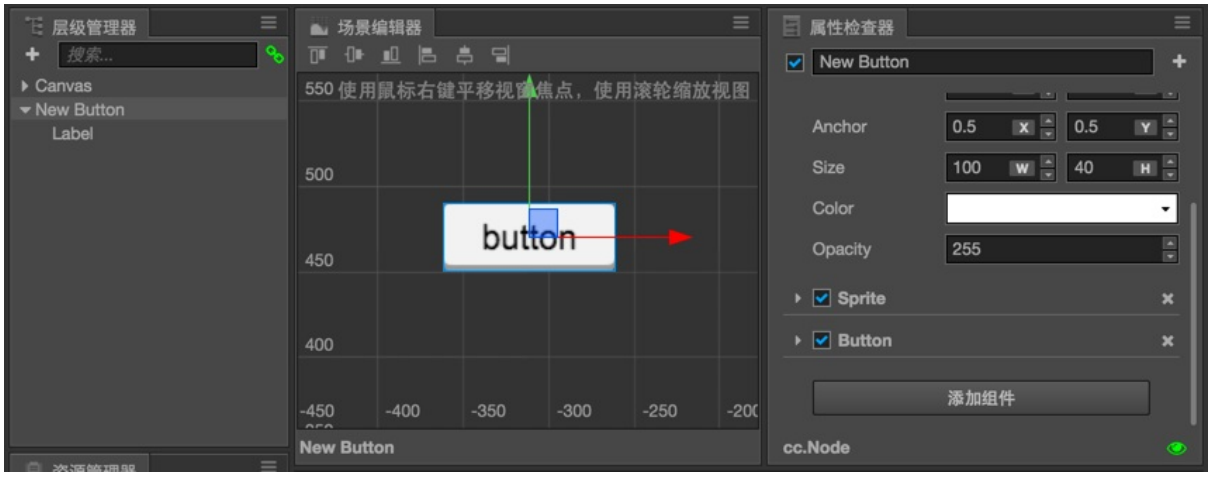
**创建节点菜单** 里下一个类别是 `创建渲染节点`，这里我们能找到像 Sprite（精灵）、Label（文字）、ParticleSystem（粒子）、Tilemap（瓦片图）等由节点和基础渲染组件组成的节点类型。

这里的基础渲染组件，是无法用其他组件的组合来代替的，因此单独归为 **渲染** 类别。要注意每个节点上只能添加一个渲染组件，重复添加会导致报错。但是可以通过将不同渲染节点组合起来的方式实现更复杂的界面控件，比如下面 UI 类中的很多控件节点。

## UI 节点

从 **创建节点菜单** 中的 `创建 UI 节点` 类别里可以创建包括 Button（按钮）、Widget（对齐挂件）、Layout（布局）、ScrollView（滚动视图）、EditBox（输入框）等节点在内的常用 UI 控件。

UI 节点大部分都是由渲染节点组合而成的，比如我们通过菜单创建的 Button 节点，就包含了一个包含 Button + Sprite 组件的按钮背景节点，加上一个包含 Label 组件的标签节点：



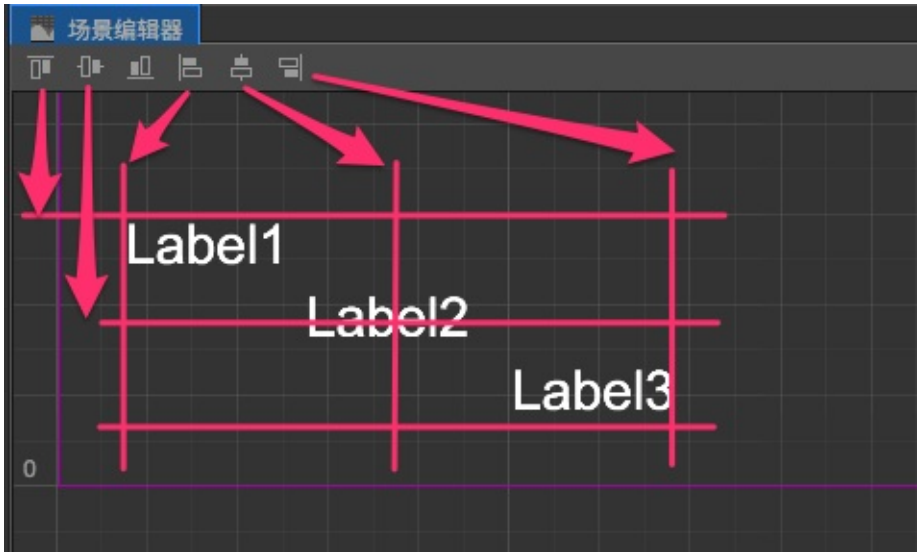

使用菜单创建基础类型的节点，是快速向场景中添加内容的推荐方法，之后我们还可以根据需要对使用菜单创建的节点进行编辑，创造我们需要的组合。

# 提高场景制作效率的技巧

- **层级管理器** 里选中一个节点，然后按 `Cmd/Ctrl + F` 就可以在 **场景编辑器** 里聚焦这个节点。
- 选中一个节点后按 `Cmd/Ctrl + D` 会在该节点相同位置复制一个同样的节点，当我们需要快速制作多个类似节点时可以用这个命令提高效率。
- 在 **场景编辑器** 里要选中多个节点，可以按住 `Cmd/Ctrl` 键依次点击你想要选中的节点，在 **层级管理器** 里也是一样的操作方式。
- **场景编辑器** 中将鼠标悬停在一个节点上（即使是空节点），会显示该节点的名称和约束框大小，这时点击就会选中当前显示名称的节点。在复杂的场景中选节点之前先悬停一会，可以大大提高选择成功率。

## 对齐节点

**场景编辑器** 左上角有一排按钮可以用来在选中多个节点时对齐这些节点，具体的对齐规则如下：




假设三个 Label 节点都已经选中，从左到右的对齐按钮会依次将这些节点：

- 按照最靠近上面的边界对齐
- 按照整体的水平中线对齐
- 按照最靠近下面的边界对齐
- 按照最靠近左边的边界对齐
- 按照整体的垂直中线对齐
- 按照最靠近右边的边界对齐

要注意对齐操作不管是一开始测定左右边界和中线还是之后将每个节点对齐时的参照，都是节点约束框的中心或某条边界，而不是节点的位置坐标。比如下图中我们将三个宽度不同的 Label 节点向右对齐后，得到下图中三个节点约束框的右边界对齐的情况，而不是让三个节点位置里的 `x` 坐标变成一致。

# 脚本开发工作流程

## 简介

Cocos Creator 的脚本主要是通过扩展组件来进行开发的。目前 Cocos Creator 支持 JavaScript 和 CoffeeScript 两种脚本语言。通过编写脚本组件，并将它赋予到场景节点中来驱动场景中的物体。

在组件脚本的编写过程中，你可以通过声明属性，将脚本中需要调节的变量映射到 **属性检查器**（Properties） 中，供策划和美术调整。于此同时，你也可以通过注册特定的回调函数，来帮助你初始化，更新甚至销毁节点。

## 内容



## 更多参考

- JavaScript 标准支持
- 推荐编码规范
- SizeProvider

继续前往 创建和使用组件脚本 开始阅读。

# 创建和使用组件脚本

## 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) {
    },
});
```

## 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。但我们建议用户根据自己的需求，选择自己喜爱的文本工具（如：Vim, Sublime Text, Web Storm, VSCode...）进行脚本编辑。

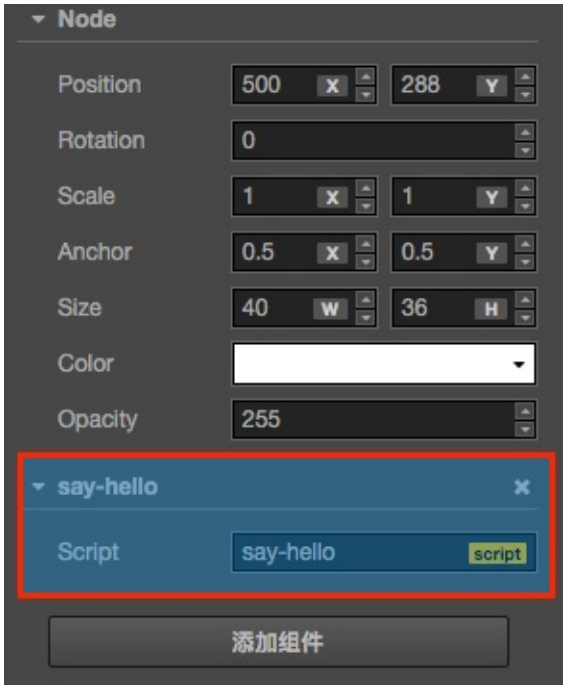
通过双击脚本资源，可以直接打开内置的 Code Editor 编辑。如果用户需要使用外部工具，请先到 偏好设置 中进行设置。

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。

## 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。

---

继续前往 使用 cc.Class 声明类型。

# 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 **CCClass**。

## 定义 CCClass

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。

## 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();
```

## 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
```

## 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
    }
});
```

## 实例方法

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

# 继承

使用 `extends` 实现继承：

```
// 父类
var Shape = cc.Class();

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

## 父构造函数

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 Rect = cc.Class({
    extends: Shape
});

var Square = cc.Class({
    extends: Rect,
    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"。

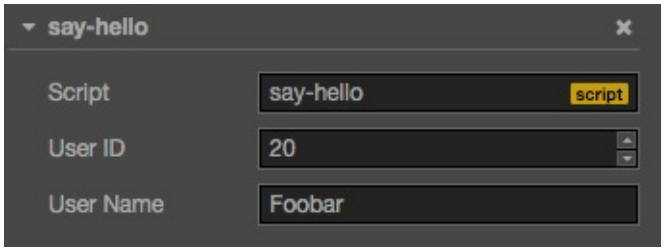
# 声明属性

通过在组件脚本中声明属性，我们可以将脚本组件中的字段可视化地展示在 **属性检查器** 中，从而方便地在场景中调整属性值。

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：




在 Cocos Creator 中，我们提供两种形式的属性声明方法：

## 简单声明

在多数情况下，我们都可以使用简单声明。

- 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。

## 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 进阶参考：type 参数
- **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

## 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
    },
}
```

## 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。

---

继续前往 访问节点和其他组件 或 CCClass 进阶参考。

# 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

# 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;
}
```

# 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';

    // Change the text in Label Component
    label.string = text;
}
```

你也可以为 `getComponent` 传入一个类名。

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

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 rotate = this.getComponent("SinRotate");
```

在节点上也有一个 `getComponent` 方法，它们的作用是一样的：

```
start: function () {
    cc.log( this.node.getComponent(cc.Label) === this.getComponent(cc.Label) );  // true
}
```

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?");
        }
    }
```

# 获得其它节点及其组件

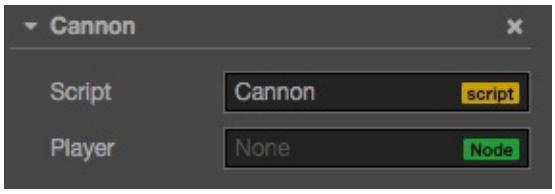
仅仅能访问节点自己的组件通常是不够的，脚本通常还需要进行多个节点之间的交互。例如，一门自动瞄准玩家的大炮，就需要不断获取玩家的最新位置。Cocos Creator 提供了一些不同的方法来获得其它节点或组件。

## 利用属性检查器设置节点

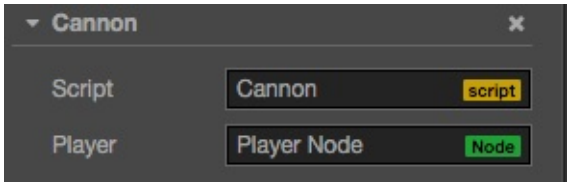
最直接的方式就是在 **属性检查器** 中设置你需要的对象。以节点为例，这只需要在脚本中声明一个 type 为 `cc.Node` 的属性：

```
// Cannon.js

cc.Class({
    extends: cc.Component,
    properties: {
        // 声明 player 属性
        player: {
            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: {
        // 声明 player 属性
        player: {
            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: {
        // 声明 player 属性，这次直接是组件类型
        player: {
            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 () {
        var cannons = this.node.children;
```

```
        // ...
    }
});
```

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");
```

## 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，即使要用也最好保证全局变量只读。

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。

```
// Globals.js, this file can have any name

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

由于所有脚本都强制声明为 "use strict"，因此定义全局变量时的 `window.` 不可省略。
接着你可以在合适的地方直接访问并初始化 `Global`：

```
// Back.js

cc.Class({
    extends: cc.Component,

    onLoad: function () {
        Global.backNode = this.node;
        Global.backLabel = this.getComponent(cc.Label);
    }
});
```

初始化后，你就能在任何地方访问到 `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';
        Global.backLabel.string = text;
    }
});
```

> 访问全局变量时，如果变量未定义将会抛出异常。
> 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。

## 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 exports 的对象。

```
// Back.js

// this feels more safe since you know where the object comes from
var Global = require("Global");

cc.Class({
    extends: cc.Component,

    onLoad: function () {
        Global.backNode = this.node;
        Global.backLabel = this.getComponent(cc.Label);
    }
});
```

```
// AnyScript.js

// this feels more safe since you know where the object comes from
var Global = 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;
    }
});
```

更详细内容，请参考 模块化。

---

继续前往 常用节点和组件接口。

# 常用节点和组件接口

在通过 访问节点和组件 介绍的方法获取到节点或组件实例后，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。

## 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，当该节点的所有父节点都激活，将意味着：

- 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，也就是不会再执行这些组件中的 `update` 中的代码
- 这些组件上如果有 `onDisable` 方法，这些方法将被执行

`this.node.active = true;`

该操作会激活节点，当该节点的所有父节点都激活，将意味着：

- 在场景中重新激活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：

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

或

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

这两种方法是等价的。

注意：

- `removeFromParent` 通常需要传入一个 `false`，否则默认会清空节点上绑定的事件和 action 等。
- 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。

# 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;
```

使用 `setPosition` 方法：

```
this.node.setPosition(100, 50);
this.node.setPosition(cc.v2(100, 50));
```

设置 `position` 变量：

```
this.node.position = cc.v2(100, 50);
```

以上两种用法等价。

## 更改节点旋转

```
this.node.rotation = 90;
```

或

```
this.node.setRotation(90);
```

## 更改节点缩放

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

或

```
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

```
this.node.setContentSize(100, 100);
this.node.setContentSize(cc.v2(100, 100));
```

或

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

以上两种方式等价。

## 更改节点锚点位置

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

或

```
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。

# 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;
```

# 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（节点添加到节点树时）执行
- `start()`： 会在该组件第一次 `update` 之前执行，通常用于需要在所有组件的 `onLoad` 初始化完毕后执行的逻辑

---

更多组件成员方法请继续参考 生命周期回调 文档。

# 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。用户只要定义特定的回调函数，Creator 就会在特定的时期自动执行相关脚本，用户不需要手工调用它们。

目前提供给用户的声明周期回调函数主要有：

- 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` 回调会在组件首次激活时触发，比如所在的场景被载入，或者所在节点被激活的情况下。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。onLoad 总是会在任何 start 方法调用前执行，这能用于安排脚本的初始化顺序。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。

```
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，或者所在节点的 `active` 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，或者所在节点的 `active` 属性从 `true` 变为 `false` 时，会激活 `onDisable` 回调。

# onDestroy

当组件或者所在节点调用了 `destroy()`，则会调用 `onDestroy` 回调，并在当帧结束时统一回收组件。

---

继续前往 创建和销毁节点。

# 创建和销毁节点

## 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。

以下是一个简单的例子:

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

  properties: {
    sprite: {
      default: null,
      type: cc.SpriteFrame,
    },
  },

  start: function () {
    var node = new cc.Node('Sprite');
    var sp = node.addComponent(cc.Sprite);

    sp.spriteFrame = this.sprite;
    node.parent = this.node;
  },
});
```

## 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);
  },
});
```

## 创建预制节点

和克隆已有节点相似，你可以设置一个预制（Prefab）并通过 `cc.instantiate` 生成节点。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);
  },
});
```

# 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被移除，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 () {
    // 5 秒后销毁目标节点
    setTimeout(function () {
      this.target.destroy();
    }.bind(this), 5000);
  },

  update: function (dt) {
    if (cc.isValid(this.target)) {
      this.target.rotation += dt * 10.0;
    }
  },
});
```

## destroy 和 removeFromParent 的区别

调用一个节点的 `removeFromParent` 后，它不一定就能完全从内存中释放，因为有可能由于一些逻辑上的问题，导致程序中仍然引用到了这个对象。因此如果一个节点不再使用了，请直接调用它的 `destroy` 而不是 `removeFromParent`。`destroy` 不但会激活组件上的 `onDestroy`，还会降低内存泄露的几率，同时减轻内存泄露时的后果。

总之，如果一个节点不再使用，`destroy` 就对了，不需要 `removeFromParent` 也不需要设置 `parent` 为 `null` 哈。

---

继续前往 资源管理/加载和切换场景 说明文档。

# 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");
```

# 通过常驻节点进行场景资源管理和参数传递

引擎同时只会运行一个场景，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);
```

需要注意的是上面的 API 并不会立即销毁指定节点，只是将节点还原为可在场景切换时销毁的节点。

## 使用全局变量

除此之外，简单的数值类数据传递也可以使用全局变量的方式进行，详见通过全局变量访问。

# 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。

# 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。那就可以预先使用 `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`，就可以真正切换场景。

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

就算预加载还没完成，你也可以直接调用 `cc.director.loadScene`，预加载完成后场景就会启动。实战例子可以参考 21点演示项目

**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：

```
// 请不要再使用下面的方法预加载场景！
cc.loader.loadRes('MyScene.fire', function(err, res) {
    cc.director.runScene(res.scene);
});
```

---

继续前往 获取和加载资源 说明文档。

# 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
- 动态加载资源
- 加载远程资源和设备资源
- 资源的依赖和释放

# 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**。

## 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。

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
        },

    }
});
```

## Raw Asset

Cocos2d 的一些旧 API 并没有使用上面提到的 Asset 对象，而是直接用 URL 字符串指代资源。为了兼容这些 API，我们把这类资源叫做 "Raw Asset"。图片（ `cc.Texture2D` ），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。

当你在脚本里声明一个类型是 `cc.Texture2D` 的 Raw Asset，一开始可能会想这样定义：

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

        textureURL: {
            default: null,
            type: cc.Texture2D
        }
```

```
    }
});
```

这样写的问题在于，在代码中 `textureURL` 实际上是一个字符串，而不是 `cc.Texture2D` 的实例。为了不混淆 type 的语义，在 CCClass 中声明 Raw Asset 的属性时，要用 `url: cc.Texture2D` 而不是 `type: cc.Texture2D` 。

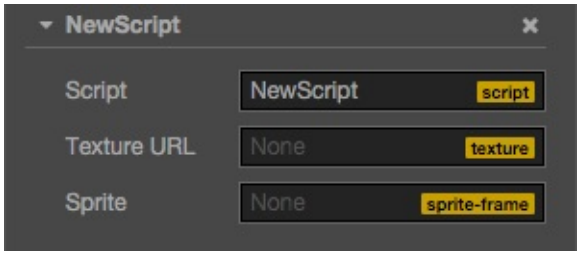
```
cc.Class({
    extends: cc.Component,
    properties: {

        textureURL: {
            default: "",
            url: cc.Texture2D
        }

    }
});
```

# 如何在属性检查器里设置资源

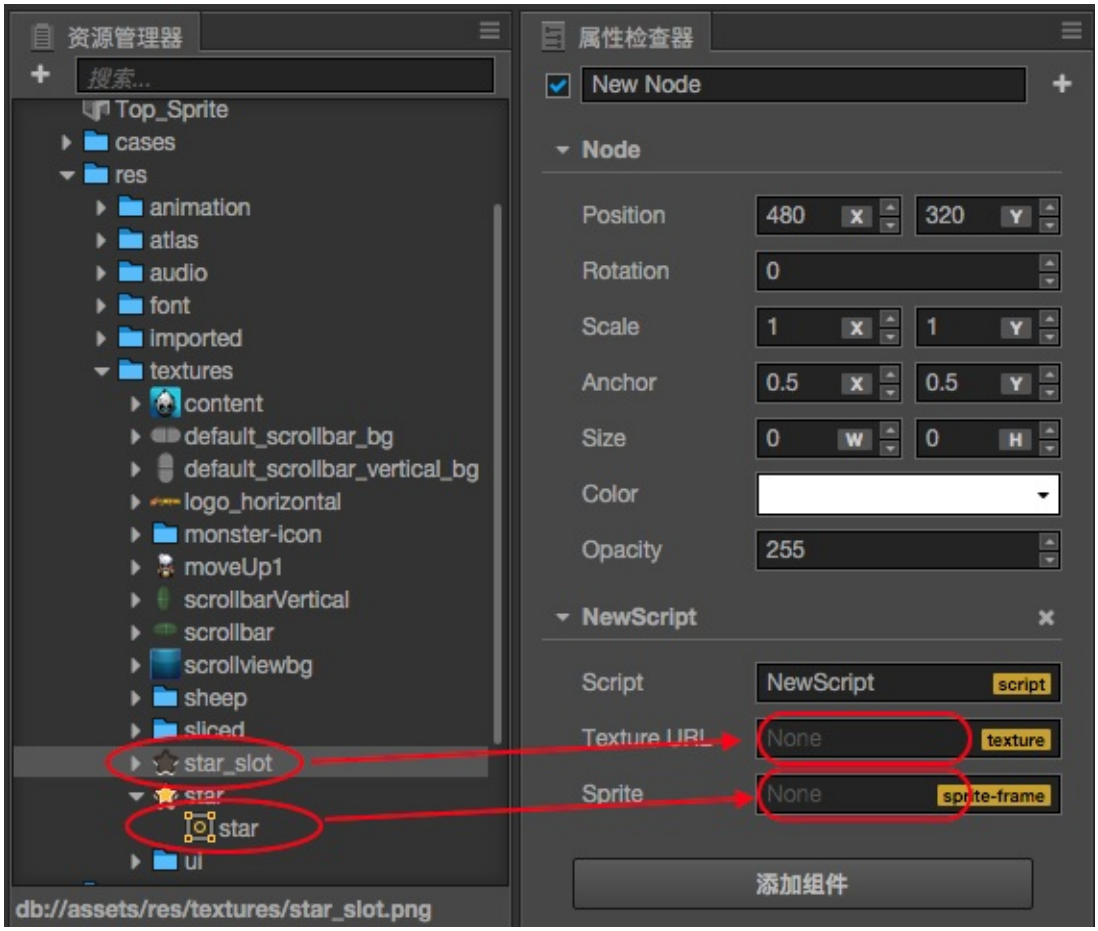
不论是 Asset 还是 Raw Asset，只要在脚本中定义好类型，就能直接在 **属性检查器** 很方便地设置资源。假设我们有这样一个组件：

```
// NewScript.js

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

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

    }
});
```

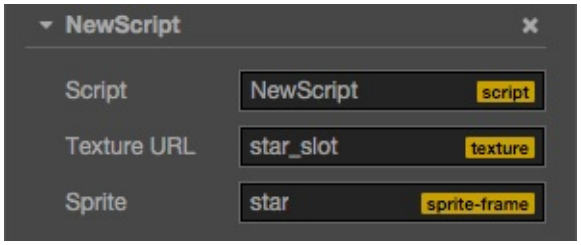
将它添加到场景后，**属性检查器** 里是这样的：




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




结果如下：



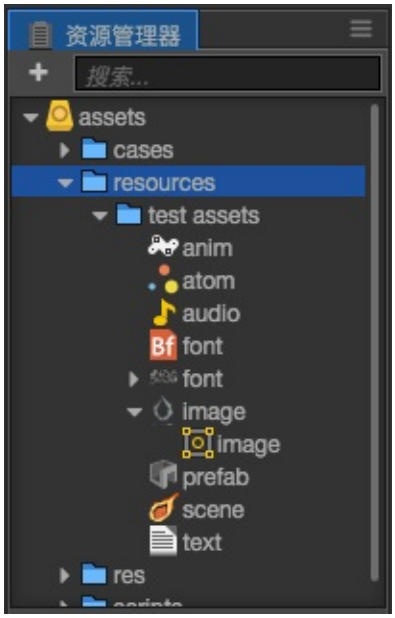

这样就能在脚本里直接拿到设置好的资源：

```
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。

# 动态加载

动态加载资源要注意两点，一是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。如果一份资源不需要由脚本**直接**动态加载，那么千万不要放在 `resources` 文件夹里。

第二个要注意的是 Creator 相比之前的 Cocos2d-html5，资源动态加载的时都是**异步**的，需要在回调函数中获得载入的资源。这么做是因为 Creator 除了场景关联的资源，没有另外的资源预加载列表，动态加载的资源是真正的动态加载。

## 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
var self = this;
cc.loader.loadRes("test assets/anim", function (err, clip) {
    self.node.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
var self = this;
cc.loader.loadRes("test assets/image", cc.SpriteFrame, function (err, spriteFrame) {
    self.node.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` 可以传入和 `loadRes` 相同的路径和类型参数。

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

此外，你也可以使用 `cc.loader.releaseAsset` 来释放特定的 Asset 实例。

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

## 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");

// 改用 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);
```

## 资源批量加载

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

// 加载 sheep.plist 图集中的所有 SpriteFrame
cc.loader.loadResDir("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', cc.SpriteAtlas, function (err, atlas) {...}) 获得的则是整个 SpriteAtlas 对象。
});
```

# 加载远程资源和设备资源

在目前的 Cocos Creator 中，我们支持加载远程贴图资源，这对于加载用户头像等需要向服务器请求的贴图很友好，需要注意的是，这需要开发者直接调用 `cc.loader.load` 。同时，如果用户用其他方式下载了资源到本地设备存储中，也需要用同样的 API 来加载，上文中的 `loadRes` 等 API 只适用于应用包内的资源和热更新的本地资源。下面是这个 API 的用法：

```
// 远程 url 带图片后缀名
var remoteUrl = "http://unknown.org/someres.png";
cc.loader.load(remoteUrl, function (err, texture) {
    // Use texture to create sprite frame
});

// 远程 url 不带图片后缀名，此时必须指定远程图片文件的类型
remoteUrl = "http://unknown.org/emoji?id=124982374";
cc.loader.load({url: remoteUrl, type: 'png'}, function () {
    // Use texture to create sprite frame
});

// 用绝对路径加载设备存储内的资源，比如相册
var absolutePath = "/dara/data/some/path/to/image.png"
cc.loader.load(absolutePath, function () {
    // Use texture to create sprite frame
});
```

目前的此类手动资源加载还有一些限制，对用户影响比较大的是：

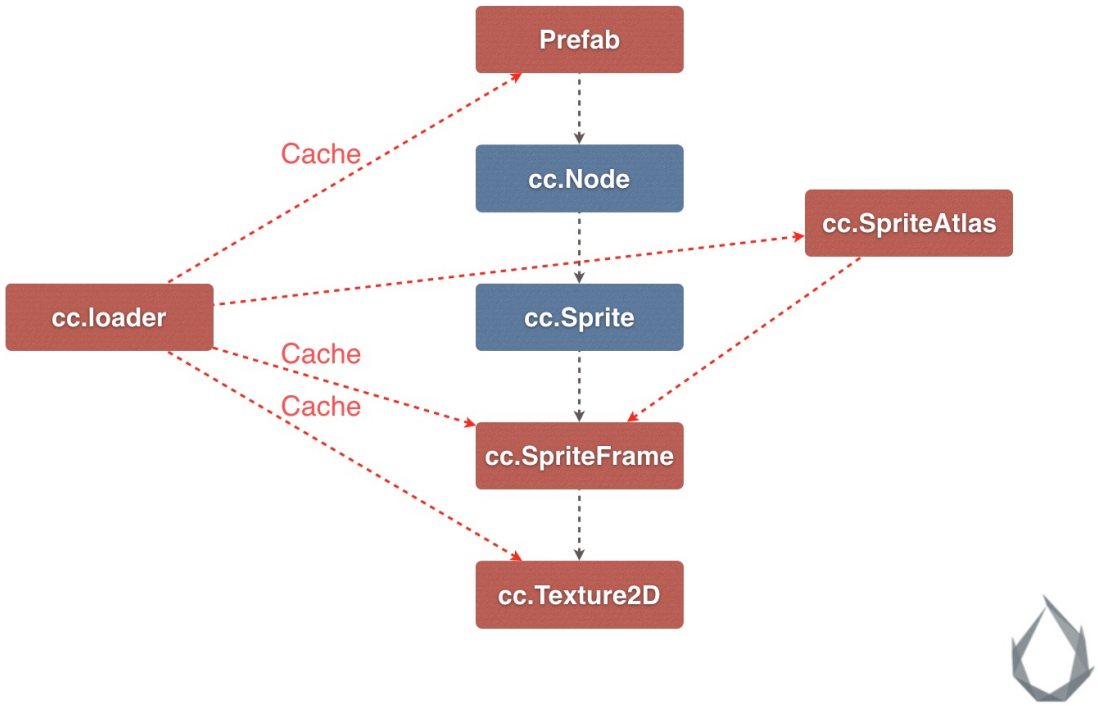
1. 远程加载不支持图片文件以外类型的资源（已在 1.5 / 1.6 支持计划中）
2. 这种加载方式只支持 raw asset 资源类型，不支持 SpriteFrame、SpriteAtlas、Tilemap 等资源的直接加载和解析（需要后续版本中的 Assets Bundle 支持）
3. Web 端的远程加载受到浏览器的 CORS 跨域策略限制，如果对方服务器禁止跨域访问，那么会加载失败，而且在 WebGL 渲染模式下，即便对方服务器允许 http 请求成功之后也无法渲染，这是 WebGL 的安全策略的限制

# 资源的依赖和释放

在加载完资源之后，所有的资源都会临时被缓存到 `cc.loader` 中，以避免重复加载资源时发送无意义的 http 请求，当然，缓存的内容都会占用内存，有些资源可能用户不再需要了，想要释放它们，这里介绍一下在做资源释放时需要注意的事项。

**首先最为重要的一点就是：资源之间是互相依赖的。**

比如下图，Prefab 资源中的 Node 包含 Sprite 组件，Sprite 组件依赖于 SpriteFrame，SpriteFrame 资源依赖于 Texture 资源，而 Prefab，SpriteFrame 和 Texture 资源都被 cc.loader 缓存起来了。这样做的好处是，有可能有另一个 SpriteAtlas 资源依赖于同样的一个 SpriteFrame 和 Texture，那么当你手动加载这个 SpriteAtlas 的时候，就不需要再重新请求贴图资源了，cc.loader 会自动使用缓存中的资源。




在搞明白资源的相互引用之后，资源释放的问题也就呼之欲出了，当你选择释放一个 Prefab 时，我们是不会自动释放它依赖的其他资源的，因为有可能这些依赖资源还有其他的用处。所以用户在释放资源时经常会问我们，为什么我都把资源释放了，内存占用还是居高不下？原因就是真正占用内存的贴图等基础资源并不会随着你释放 Prefab 或者 SpriteAtlas 而被释放。

**接下来要介绍问题的另一个核心：JavaScript 中无法跟踪对象引用。**

在 JavaScript 这种脚本语言中，由于其弱类型特性，以及为了代码的便利，往往是不包含内存管理功能的，所有对象的内存都由垃圾回收机制来管理。这就导致 JS 层逻辑永远不知道一个对象会在什么时候被释放，这意味着引擎无法通过类似引用计数的机制来管理外部对象对资源的引用，也无法严谨得统计资源是否不再被需要了。基于以上的原因，目前 cc.loader 的设计实际上是依赖于用户根据游戏逻辑管理资源，用户可以决定在某一时刻不再需要某些资源以及它依赖的资源，立即将它们在 cc.loader 中的缓存释放。也可以选择在释放依赖资源的时候，防止部分共享资源被释放。下面是一个简单的示例：

```
// 直接释放某个贴图
cc.loader.release(texture);
// 释放一个 prefab 以及所有它依赖的资源
var deps = cc.loader.getDependsRecursively('prefabs/sample');
cc.loader.release(deps);
// 如果在这个 prefab 中有一些和场景其他部分共享的资源，你不希望它们被释放，有两种方法：
// 1. 显式声明禁止某个资源的自动释放
cc.loader.setAutoRelease(texture2d, false);
// 2. 将这个资源从依赖列表中删除
var deps = cc.loader.getDependsRecursively('prefabs/sample');
var index = deps.indexOf(texture2d._uuid);
if (index !== -1)
    deps.splice(index, 1);
cc.loader.release(deps);
```

**最后一个值得关注的要点：JavaScript 的垃圾回收是延迟的。**

想象一种情况，当你释放了 cc.loader 对某个资源的引用之后，由于考虑不周的原因，游戏逻辑再次请求了这个资源。此时垃圾回收还没有开始（垃圾回收的时机不可控），或者你的游戏逻辑某处，仍然持有一个对于这个旧资源的引用，那么意味着这个资源还存在内存中，但是 cc.loader 已经访问不到了，所以会重新加载它。这造成这个资源在内存中有两份同样的拷贝，浪费了内存。如果只是一个资源还好，但是如果类似的资源很多，甚至不止一次被重复加载，这对于内存的压力是有可能很高的。如果观察到游戏使用的内存曲线有这样的异常，请仔细检查游戏逻辑，是否存在泄漏，如果没有的话，垃圾回收机制是会正常回收这些内存的。

以上就是管理资源依赖和释放时需要注意的细节，这部分的功能和 API 设计还没有完全定案，我们还是希望尽力给大家带来尽可能方便的引擎 API，所以后续也会尝试一些其他的办法提升友好度，届时会更新这篇文档。

---

继续前往 事件系统/发射和监听事件 说明文档。

# 监听和发射事件

## 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：

```
// 使用函数绑定
this.node.on('mousedown', function ( event ) {
  this.enabled = false;
}.bind(this));

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

# 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',
    });
  },
});
```

# 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) );
```

如果我们希望在 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：

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

# 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 |
| --- | --- | --- |
| | | |

| | | |
|---|---|---|
| type | String | 事件的类型（事件名） |
| target | cc.Node | 接收到事件的原始对象 |
| currentTarget | cc.Node | 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 |
| getType | Funciton | 获取事件的类型 |
| stopPropagation | Function | 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 |
| stopPropagationImmediate | Function | 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 |
| getCurrentTarget | Function | 获取当前接收到事件的目标节点 |
| detail | Function | 自定义事件的信息（属于 cc.Event.EventCustom ） |
| setUserData | Function | 设置自定义事件的信息（属于 cc.Event.EventCustom ） |
| getUserData | Function | 获取自定义事件的信息（属于 cc.Event.EventCustom ） |

完整的 API 列表可以参考 cc.Event 及其子类的 API 文档。

# 系统内置事件

以上是通用的事件监听和发射规则，在 Cocos Creator 中，我们默认支持了一些系统内置事件，可以参考我们后续的文档来查看如何使用：

- 鼠标、触摸：可参考节点系统事件文档
- 键盘、重力感应：可参考全局系统事件文档

# 节点系统事件

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节点相关的系统事件，这篇文档将介绍这些事件的使用方式。

Cocos Creator 支持的系统事件包含鼠标、触摸、键盘、重力传感四种，其中本章节重点介绍与节点树相关联的鼠标和触摸事件，这些事件是被直接触发在相关节点上的，所以被称为节点系统事件。与之对应的，键盘和重力传感事件被称为全局系统事件，细节可以参考全局系统事件文档。

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);
```

# 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 |
| --- | --- | --- |
| `cc.Node.EventType.MOUSE_DOWN` | 'mousedown' | 当鼠标在目标节点区域按下时触发一次 |
| `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 |
| `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` |

# 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开屏幕时 |
| `cc.Node.EventType.TOUCH_CANCEL` | 'touchcancel' | 当手指在目标节点区域外离开屏幕时 |

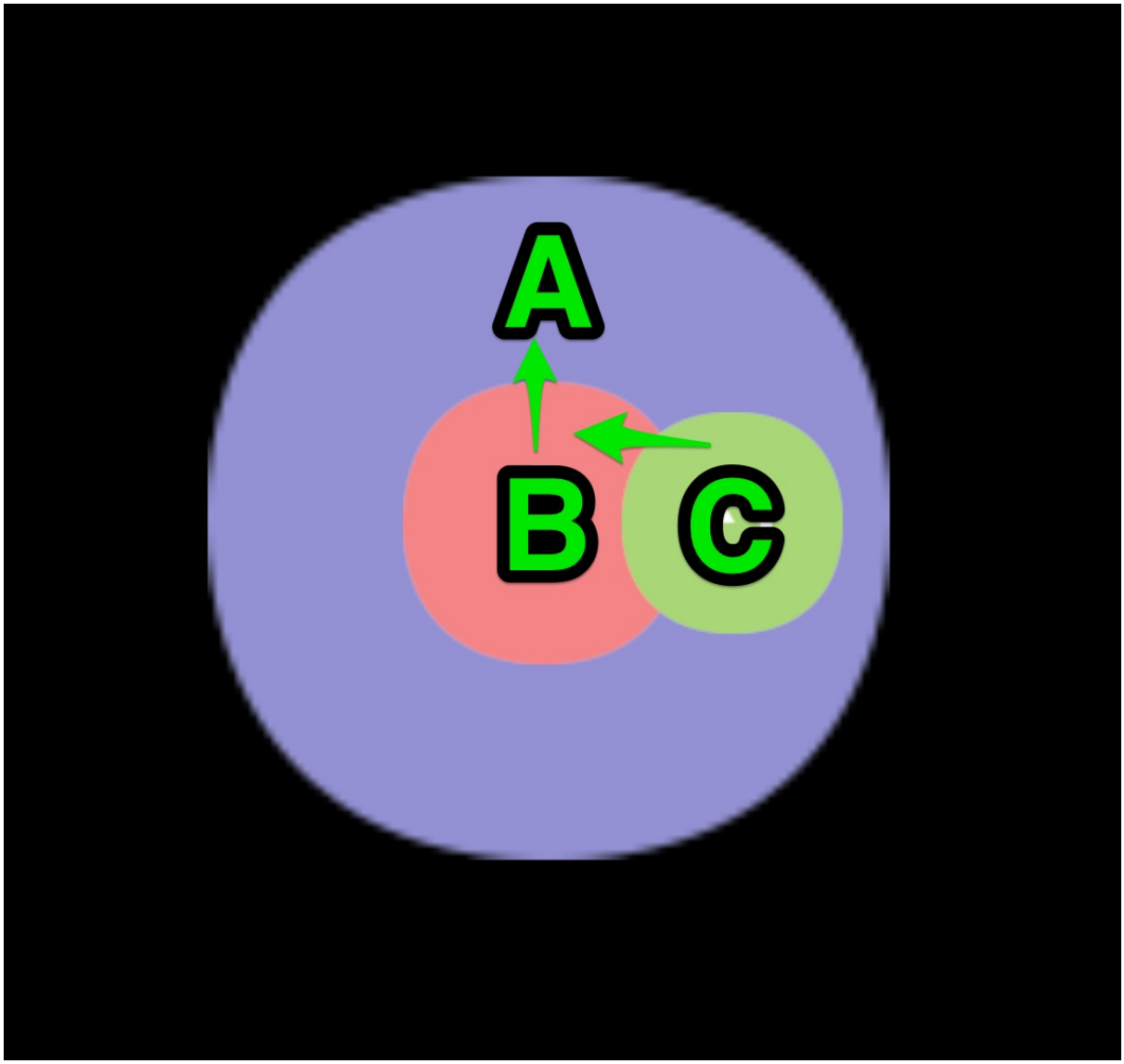
触摸事件（`cc.Event.EventTouch`）的重要 API 如下（`cc.Event` 标准事件 API 之外）：

| 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。

# 触摸事件冒泡

触摸事件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，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。

## `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 |

# 全局系统事件

本篇教程，我们将介绍 Cocos Creator 的全局系统事件。

全局系统事件是指与节点树不相关的各种全局事件，由 `cc.systemEvent` 来统一派发，目前支持了以下几种事件：

- 键盘事件
- 设备重力传感事件

除此之外，鼠标事件与触摸事件请参考节点系统事件

**注意：目前已经不建议直接使用 cc.eventManager 来注册任何事件，cc.eventManager 的用法也不保证持续性，有可能随时被修改**

# 如何定义输入事件

键盘、设备重力传感器此类全局事件是通过函数 `cc.systemEvent.on(type, callback, target)` 注册的。

可选的 `type` 类型有:

1. cc.SystemEvent.EventType.KEY_DOWN (键盘按下)
2. cc.SystemEvent.EventType.KEY_UP (键盘释放)
3. cc.SystemEvent.EventType.DEVICEMOTION (设备重力传感)

## 键盘事件

- 事件监听器类型： `cc.SystemEvent.EventType.KEY_DOWN` 和 `cc.SystemEvent.EventType.KEY_UP`
- 事件触发后的回调函数：
  - 自定义回调函数：callback(event);
- 回调参数：
  - KeyCode: API 传送门
  - Event：API 传送门

```
cc.Class({
    extends: cc.Component,
    onLoad: function () {
        // add key down and key up event
        cc.systemEvent.on(cc.SystemEvent.EventType.KEY_DOWN, this.onKeyDown, this);
        cc.systemEvent.on(cc.SystemEvent.EventType.KEY_UP, this.onKeyUp, this);
    },

    onDestroy () {
        cc.systemEvent.off(cc.SystemEvent.EventType.KEY_DOWN, this.onKeyDown, this);
        cc.systemEvent.off(cc.SystemEvent.EventType.KEY_UP, this.onKeyUp, this);
    },

    onKeyDown: function (event) {
        switch(event.keyCode) {
            case cc.KEY.a:
                console.log('Press a key');
                break;
        }
    },

    onKeyUp: function (event) {
        switch(event.keyCode) {
            case cc.KEY.a:
                console.log('release a key');
```

```
                break;
        }
    }
});
```

## 设备重力传感事件

- 事件监听器类型：`cc.SystemEvent.EventType.DEVICEMOTION`
- 事件触发后的回调函数：
  - 自定义回调函数：callback(event);;
- 回调参数：
  - Event：API 传送门

```
cc.Class({
    extends: cc.Component,
    onLoad () {
        // open Accelerometer
        cc.inputManager.setAccelerometerEnabled(true);
        cc.systemEvent.on(cc.SystemEvent.EventType.DEVICEMOTION, this.onDeviceMotionEvent, this);
    },

    onDestroy () {
        cc.systemEvent.off(cc.SystemEvent.EventType.DEVICEMOTION, this.onDeviceMotionEvent, this);
    },

    onDeviceMotionEvent (event) {
        cc.log(event.acc.x + "    " + event.acc.y);
    },
});
```

大家可以也去看 官方范例 `cases03_gameplay/01_player_control` 目录下的完整范例（这里包含了，键盘，重力感应，单点触摸，多点触摸的范例）。

# 在 Cocos Creator 中使用动作系统

## 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。

## 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);
// 执行动作
node.runAction(action);
// 停止一个动作
node.stopAction(action);
// 停止所有动作
node.stopAllActions();
```

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

## 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）

### 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`。

### 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, 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);
```

## 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 CCClass 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
```

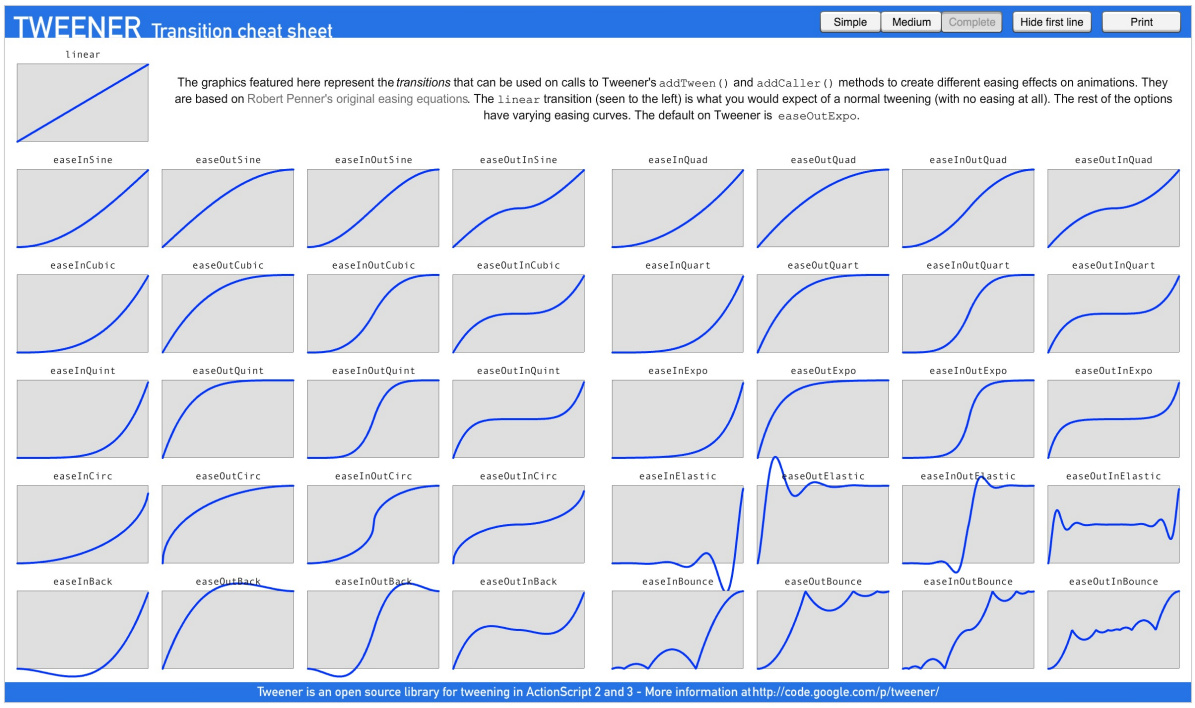
注意: 在 cc.callFunc 中不应该停止自身动作，由于动作是不能被立即删除，如果在动作回调中暂停自身动作会引发一系列遍历问题，导致更严重的 bug。

## 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：




图片源自 *http://hosted.zeh.com.br/tweener/docs/en-us/*

# 具体动作 API 参考

接下来请参考动作系统 API 列表来了解有哪些动作系统接口可以使用。

# 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。

在这些动作类型的文档中，开发者可以了解到各个动作类型的基本 API。

## 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 |
| 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 |
| 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 |
| 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 |

# 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。

也许有人会认为 `setTimeout` 和 `setInterval` 就足够了，开发者当然可以使用这两个函数，不过我们更推荐使用计时器，因为它更加强大灵活，和组件也结合得更好！

下面来看看它的具体使用方式：

1. 开始一个计时器

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

   上面这个计时器将每隔 5s 执行一次。

2. 更灵活的计时器

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

3. 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。

4. 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。

## 注意：`cc.Node` 不包含计时器相关 API

# 脚本执行顺序

更完善的脚本执行顺序控制将在 1.5 中添加，目前请使用下面的原则控制脚本执行顺序：

## 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。

## 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);
    }
```

## 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，可以通过组件在 **属性检查器** 里的排列顺序来控制。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!');
    },
});
```

组件顺序 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!
```

---

继续前往 网络接口。

# 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

当然，在 Web 平台，浏览器原生就支持这两个接口，之所以说 Cocos Creator 支持，是因为在发布原生版本时，用户使用这两个网络接口的代码也是可以运行的。也就是遵循 Cocos 一直秉承的 “一套代码，多平台运行” 原则。

# 使用方法

1. 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。

2. 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。

# 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
3. 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，因此 Cocos 在原生环境中自带了 SocketIO 实现。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
   }
   ```

4. 将 SocketIO 脚本文件设为 插件脚本，这样在组件中直接使用 window.io 就能访问到 SocketIO
5. 在组件中使用 SocketIO，可以参考 SocketIO 官方网站查询 API 和文档等

# 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。

# 对象池的概念

对象池就是一组可回收的节点对象，我们通过创建 `cc.NodePool` 的实例来初始化一种节点的对象池。通常当我们有多个 prefab 需要实例化时，应该为每个 prefab 创建一个 `cc.NodePool` 实例。 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，调用对象池实例的 `put(node)` 方法，传入需要销毁的节点实例，对象池会自动完成把节点从场景节点树中移除的操作，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.NodePool` 的详细 API 说明，请参考 cc.NodePool API 文档。

# 流程介绍

下面是使用对象池的一般工作流程

## 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。

## 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 () {
    this.enemyPool = new cc.NodePool();
    let initCount = 5;
    for (let i = 0; i < initCount; ++i) {
        let enemy = cc.instantiate(this.enemyPrefab); // 创建节点
        this.enemyPool.put(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。

## 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;
    if (this.enemyPool.size() > 0) { // 通过 size 接口判断对象池中是否有空闲的对象
        enemy = this.enemyPool.get();
    } 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
    enemy.getComponent('Enemy').init(); //接下来就可以调用 enemy 身上的脚本进行初始化
}
```

安全使用对象池的要点就是在 `get` 获取对象之前，永远都要先用 `size` 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！另一个选择是直接调用 `get`，如果对象池里没有可用的节点，会返回 `null`，在这一步进行判断也可以。

### 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 应该是一个 cc.Node
    this.enemyPool.put(enemy); // 和初始化时的方法一样，将节点放进对象池，这个方法会同时调用节点的 removeFromParent
}
```

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。

## 使用组件来处理回收和复用的事件

使用构造函数创建对象池时，可以指定一个组件类型或名称，作为挂载在节点上用于处理节点回收和复用事件的组件。假如我们有一组可点击的菜单项需要做成对象池，每个菜单项上有一个 `MenuItem.js` 组件：

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

    onLoad: function () {
        this.node.selected = false;
        this.node.on(cc.Node.EventType.TOUCH_END, this.onSelect.bind(this), this.node);
    },

    unuse: function () {
        this.node.off(cc.Node.EventType.TOUCH_END, this.onSelect.bind(this), this.node);
    },

    reuse: function () {
        this.node.on(cc.Node.EventType.TOUCH_END, this.onSelect.bind(this), this.node);
    }
});
```

在创建对象池时可以用：

```
let menuItemPool = new cc.NodePool('MenuItem');
```

这样当使用 `menuItemPool.get()` 获取节点后，就会调用 `MenuItem` 里的 `reuse` 方法，完成点击事件的注册。当使用 `menuItemPool.put(menuItemNode)` 回收节点后，会调用 `MenuItem` 里的 `unuse` 方法，完成点击事件的反注册。

另外 `cc.NodePool.get()` 可以传入任意数量类型的参数，这些参数会被原样传递给 `reuse` 方法：

```
// BulletManager.js
let myBulletPool = new cc.NodePool('Bullet'); //创建子弹对象池
...
let newBullet = myBulletPool.get(this); // 传入 manager 的实例，用于之后在子弹脚本中回收子弹


// Bullet.js

reuse (bulletManager) {
    this.bulletManager = bulletManager; // get 中传入的管理类实例
}

hit () {
    // ...
    this.bulletManager.put(this.node); // 通过之前传入的管理类实例回收子弹
}
```

# 清除对象池

如果对象池中的节点不再被需要，我们可以手动清空对象池，销毁其中缓存的所有节点：

```
myPool.clear(); // 调用这个方法就可以清空对象池
```

当对象池实例不再被任何地方引用时，引擎的垃圾回收系统会自动对对象池中的节点进行销毁和回收。但这个过程的时间点不可控，另外如果其中的节点有被其他地方所引用，也可能会导致内存泄露，所以最好在切换场景或其他不再需要对象池的时候手动调用 `clear` 方法来清空缓存节点。

# 使用 cc.NodePool 的优势

`cc.NodePool` 除了可以创建多个对象池实例，同一个 prefab 也可以创建多个对象池，每个对象池中用不同参数进行初始化，大大增强了灵活性；此外 `cc.NodePool` 针对节点事件注册系统进行了优化，用户可以根据自己的需要自由的在节点回收和复用的生命周期里进行事件的注册和反注册。

而之前的 `cc.pool` 接口是一个单例，无法正确处理节点回收和复用时的事件注册。不再推荐使用。

对象池的基本功能其实非常简单，就是使用数组来保存已经创建的节点实例列表。如果有其他更复杂的需求，你也可以参考 暗黑斩 Demo 中的 PoolMng 脚本 来实现自己的对象池。

# 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：

- Java 和 Python 中的 `import`
- C# 中的 `using`
- C/C++ 中的 `include`
- 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 Creator 编译为原生的 JavaScript，可直接在浏览器中运行。

# 引用模块

## require

除了 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。

## 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，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。

# 定义模块

## 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，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。

## 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 `cfg` 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"。

> `module.exports` 的默认值：
> 当你的 `module.exports` 没有任何定义时，Creator 会自动优先将 `exports` 设置为脚本中定义的 Component。如果脚本没定义 Component 但是定义了别的类型的 CCClass，则自动把 `exports` 设为定义的 CCClass。

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。

# 更多示例

## 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。

```
// foobar.js:

module.exports = {
    FOO: function () {
        this.type = "foo";
    },
    bar: "bar"
};
```

```
// test.js:

var foobar = require("foobar");
var foo = new foobar.FOO();
cc.log(foo.type);      // "foo"
cc.log(foobar.bar);    // "bar"
```

## 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：

```
// foobar.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
```

# 循环引用

请参考属性延迟定义

---

继续前往 插件脚本。

# 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”。

- 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，一般简单了解即可：

| 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 |
| 变量声明 | 脚本内声明的局部变量不会暴露到全局 | 发布后，脚本内不在任何函数内声明的局部变量都会暴露成全局变量。编辑器下则和普通脚本相同。 |
| 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 |

| | | |
|---|---|---|
| 脚本导入项目时 | 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 |

勾选“导入为插件”后，还能够进一步在 **属性检查器** 设置这个插件脚本什么时候才会生效：

| 选项 | 影响平台 | 备注 |
|---|---|---|
| 允许 Web 平台加载 | 浏览器、网页预览、编辑器 | 默认启用，禁用时会连带“允许编辑器加载”一起禁用 |
| 允许编辑器加载 | 编辑器 | 默认禁用，如果编辑器中的其它普通脚本加载过程中会依赖当前脚本，则需要手动开启这个选项。<br>开启后，脚本内不在任何函数内声明的局部变量**不会**暴露成全局变量，所以全局变量需要用 `window.abc = 0` 的方式定义才能生效。 |
| 允许 Native 平台加载 | 原生平台、模拟器预览 | 默认启用 |

# 脚本加载顺序

脚本加载顺序如下：

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。

# 注意事项

- **如果插件包含了多个脚本，则需要把插件用到的所有脚本合并为单个的 js 文件**
  以 Async（https://github.com/caolan/async）为例，这个库包含了非常多的零散的源文件，如果把所有源文件都放到项目里，则每个源文件都要设置一次“导入为插件”，并且 Creator 无法保证这些源文件之间的加载顺序，很容易

报错。所以我们要找到插件作者提供的预编译好的单个脚本，例如 async.js 或 async.min.js，这样的文件可以直接用浏览器加载运行，不需要做额外的编译操作，一般可以直接放入 Creator 中使用。如果插件作者没提供打包好的版本，通常也会在文档中说明如何编译出浏览器可执行的脚本，照着操作就行。

- **如果插件还依赖于其它插件，也需要把多个插件合并为单个 js 文件**
  以 protobuf.js 为例，这个库还依赖于 bytebuffer.js，但是插件作者并没有提供整合好的独立运行版本。我们可以先下载到这两个库各自编译后的两个文件 protobuf.js 和 bytebuffer.js，然后使用文本编辑器或类似 `cat` 这样的命令行工具将这两个脚本拼合成一个新的脚本 protobuf_all.js。然后就能在 Creator 中直接使用这个 protobuf_all.js 了。
- **不支持插件主动加载其它脚本**
  以 lzma 插件为例，这个插件默认提供的 `lzma.js` 脚本会通过浏览器的 Worker 加载另一个工作者脚本，目前 Creator 不支持这样的额外加载。解决方式是 单独使用 `lzma_worker.js` 就好。其它像是内部采用 `document.createElement("script")` 自行加载依赖项的插件，也需要做类似处理才能导入 Creator。

# 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
window.DEFAULT_IP = "192.168.1.1";

// 定义组件开关
window.ENABLE_NET_DEBUGGER = true;

// 定义引擎 API 缩写（仅适用于构造函数）
window.V2 = cc.Vec2;
```

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,
        start: function () {
            // 在开始后就移除该组件
            this.destroy();
        }
    });
}
```

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。

请注意：游戏脱离编辑器运行时，插件脚本将直接运行在全局作用域，脚本内不在任何函数内的局部变量都会暴露成全局变量，请小心因此引发的全局变量污染。

> 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。

---

继续前往 JavaScript 快速入门 或者返回 脚本开发。

# JavaScript 快速入门

> 本文改编自 A JavaScript Primer For Meteor

## 概述

本文以介绍 JavaScript 为主，初学者掌握本文的内容后，将能够对 JavaScript 有大体了解，并且满足 Cocos Creator 的开发需求。

JavaScript是一门充满争议的编程语言：它以 Java 命名，但实际上和 Java 毫无关系。JavaScript 的创造只用了10天时间，但在20年时间里却发展成世界上最流行的 Web 开发语言。如果为 JavaScript 今日的地位和流行程度找一个原因，那毫无疑问是容易上手的语言特性。当然，精通 JavaScript 是一项艰巨的任务，但学会足够开发 Web 应用和游戏的知识却很简单，如果你已经有了一定编程基础，熟悉 JavaScript 语言特性不会花费你多长时间。

另外，在使用 Cocos Creator 开发游戏时你大多数情况下都会重复使用一些固有的模式。根据帕雷托法则（也叫二八定律），掌握一门语言的20%就足够你应付80%以上的情况了。现在就让我们来花最短的时间学习足够的 JavaScript 知识，以便我们开始使用 Cocos Creator 开发游戏。

## 边读边尝试

如果你能看到这篇文章，那么你已经具备了全功能的 JavaScript 开发环境——我说的就是你正在使用的浏览器！

在本页面中读到的所有例子，你都可以把他们输入到浏览器的控制台里并查看运行结果，如果你不清楚怎么做，可以阅读如何在不同浏览器中打开控制台的指南。

准备好了吗？让我们开始学习 JavaScript 吧！

## 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。

## 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);
```

我们看到函数声明也和变量声明一样遵从 `var something = somethingElse` 的模式。因为在 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
```

# 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!)
}
```

# If

JavaScript 中条件判断语句 `if` 是这样用的：

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

# If/Else

`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();
}
```

# 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
```

# 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; // Jare Guo
myProfile.friends[1].name; // Nantas
```

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。

# 匿名函数

我们之前试过了用变量声明的语法来定义函数：

```
myFunction = function (myArgument) {
    // do something
}
```

再复习一下将函数作为参数传入其他函数调用中的用法：

```
square = function (a) {
    return a * a;
}
applyOperation = function (f, a) {
    return f(a);
}
applyOperation(square, 10); // 100
```

我们还见识了 JavaScript 的语法是多么喜欢偷懒，所以我们就可以用这样的方式代替上面的多个函数声明：

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

我们这次并没有声明 `square` 函数，并将 `square` 作为参数传递，而是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。

# 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：

```
var n = 5;
n.double().square(); //100
```

# This

`this` 可能是 JavaScript 中最难以理解和掌握的概念了。

简单地说， `this` 关键字能让你访问正在处理的对象：就像变色龙一样， `this` 也会随着执行环境的变化而变化。

解释 `this` 的原理是很复杂的，不妨让我们使用两种工具来帮助我们在实践中理解 `this` 的值：

首先是最普通又最常用的 `console.log()` ，它能够将对象的信息输出到浏览器的控制台里。在每个函数体开始的地方加入一个 `console.log()` ，确保我们了解当时运行环境下正在处理的对象是什么。

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

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;
    // do something
}
```

乍一看好像这样子并没有什么作用，实际上它允许你安全的使用 `myObject` 这个变量来指代最初执行函数的对象，而不用担心在后面的代码中 `this` 会变成其他东西。

关于 JavaScript 里 `this` 的详细原理说明，请参考这篇文章 this 的值到底是什么？一次说清楚。

# 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
```

# 代码风格

最后，下面这些代码风格上的规则能帮助我们写出更清晰明确的代码：

- 使用驼峰命名法：定义 `myRandomVariable` 这样的变量名，而不是 `my_random_variable`
- 在每一行结束时写一个 `;`，尽管在 JavaScript 里行尾的 `;` 是可以忽略的
- 在每个关键字前后都加上空格，如 `a = b + 1`，而不是 `a=b+1`

# 组合我们学到的知识

以上基础的 JavaScript 语法知识已经介绍完了，下面我们来看看能否理解实际的 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。

# 继续学习

这篇简短的教程从任何角度上说都无法代替系统的 JavaScript 学习，但这里介绍的几种语法模式已经能够帮助你理解绝大部分 Cocos Creator 文档和教程中的代码了，至少从语法上完全可以理解。

如果你像我一样喜欢通过实践学习，那么现在就可以开始跟随教程和文档学习在 Cocos Creator 中开发游戏了！

# JavaScript Resources

以下是 JavaScript 的一些入门教程:

- JavaScript标准参考教程
- JavaScript秘密花园

---

继续前往 CCClass 进阶参考。

# 使用 TypeScript 脚本

TypeScript 是一种由微软开发的自由和开源的编程语言。它是 JavaScript 的一个严格超集，并添加了可选的静态类型和基于类的面向对象编程。TypeScript 的设计目标是开发大型应用，然后转译成 JavaScript 运行。由于 TypeScript 是 JavaScript 的超集，任何现有的 JavaScript 程序都是合法的 TypeScript 程序。

关于 TypeScript 的详细使用方法，请访问 TypeScript官方网站。

# TypeScript 和 Cocos Creator

Cocos Creator 的很多用户之前是使用其他强类型语言（如 C++/C#）来编写游戏的，因此在使用 Cocos Creator 的时候也希望能够使用强类型语言来增强项目在较大规模团队中的表现。

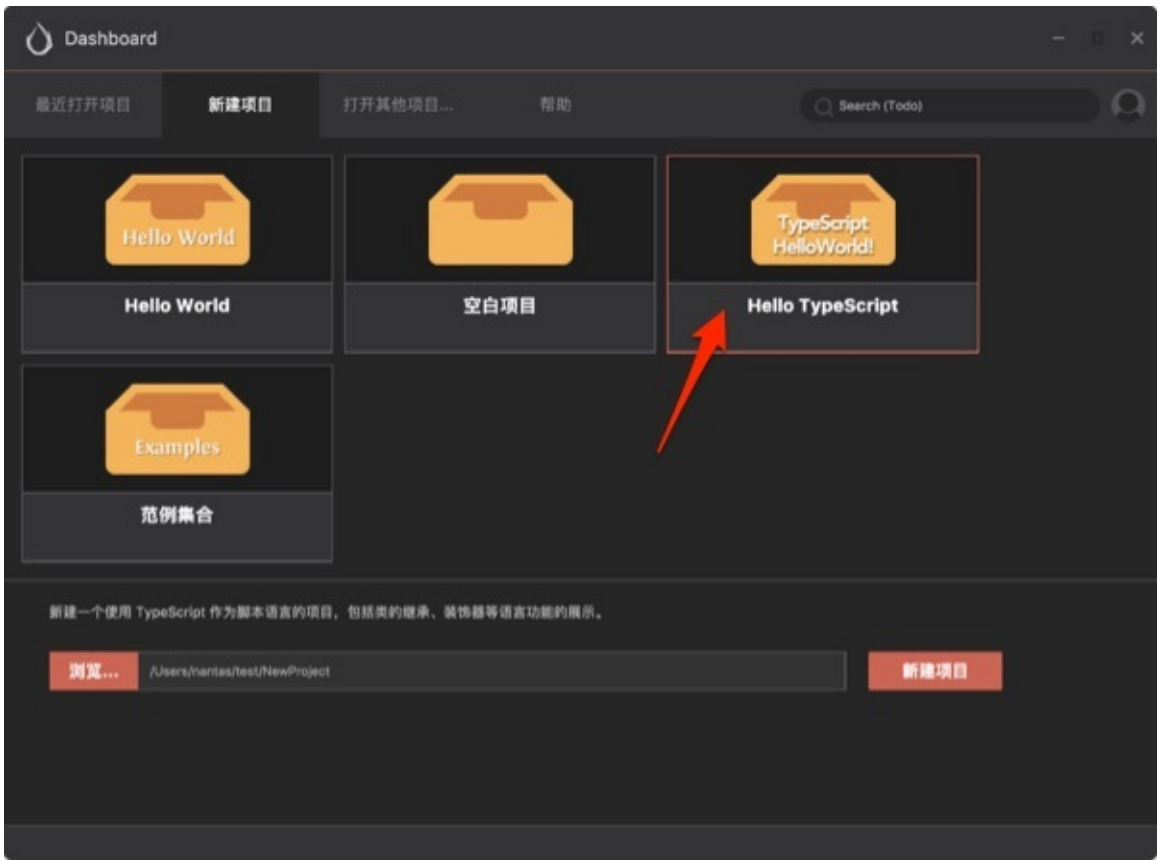
从 v1.5 版本开始 Cocos Creator 支持在项目中使用 TypeScript 编写脚本，用户的源码可以完全使用 TypeScript，或者 TypeScript 和 JavaScript 混合使用。

和其他 JavaScript 脚本一样，项目 `assets` 目录下的 TypeScript 脚本（.ts 文件）在创建或修改后激活编辑器，就会被编译成兼容浏览器标准的 ES5 JavaScript 脚本。编译后的脚本存放在项目下的 `library` （还包括其他资源）目录。

# 使用准备

## 在新项目中使用 TypeScript

新建项目时，从项目模板中选择 **HelloWorld TypeScript**，即可创建一个包括 TypeScript 相关设置和基本组件的 HelloWorld 项目。



在编辑 TypeScript 脚本时，我们推荐使用微软推出的 VS Code 作为代码编辑器，VS Code 具有完善的 TypeScript 语言支持功能。

### 在已有项目中添加 TypeScript 设置

如果希望在原有项目中添加 TypeScript 脚本，并获得 VS Code 等 IDE 的完整支持，需要执行主菜单的 `开发者 -> VS Code 工作流 -> 更新 VS Code 智能提示数据` 和 `开发者 -> VS Code 工作流 -> 添加 TypeScript 项目配置`，来添加 `creator.d.ts` 和 `tsconfig.json` 文件到你的项目根目录中。`creator.d.ts` 声明了引擎的所有 API，用于支持 VS Code 的智能提示。`tsconfig.json` 用于设置 TypeScript 项目环境，您可以参考官方的 tsconfig.json 说明 进行定制。

### 在项目中创建 TypeScript 脚本

和创建 JavaScript 脚本一样，你可以直接在文本编辑器里新建 `.ts` 文件，或通过编辑器的 **资源管理器** 的创建菜单，右键点击一个文件夹，并选择 `新建 -> TypeScript`。

## 使用 TypeScript 声明 CCClass

在 TypeScript 中 class 的声明方式 和 ES6 Class 相似。但为了编辑器能够正确解析 **属性检查器** 里显示的各类属性，我们还需要使用引擎内置的一些装饰器，来将普通的 class 声明成 CCClass。这和目前将 JavaScript 中的 ES6 Class 声明为 CCClass 的方法类似。关于装饰器的更多信息请参考 TypeScript decorator。

下面是一个基本的 TypeScript 声明组件的实例：

```
const {ccclass, property} = cc._decorator; // 从 cc._decorator 命名空间中引入 ccclass 和 property 两个装饰器

@ccclass // 使用装饰器声明 CCClass
export default class NewClass extends cc.Component { // ES6 Class 声明语法，继承 cc.Component

    @property(cc.Label)     // 使用 property 装饰器声明属性，括号里是属性类型，装饰器里的类型声明主要用于编辑器展示
    label: cc.Label = null; // 这里是 TypeScript 用来声明变量类型的写法，冒号后面是属性类型，等号后面是默认值

    // 也可以使用完整属性定义格式
    @property({
        visible: false
    })
    text: string = 'hello';

    // 成员方法
    onLoad() {
        // init logic
    }
}
```

装饰器使用 `@` 字符开头作为标记，装饰器主要用于编辑器对组件和属性的识别，而 TypeScript 语法中的类型声明 `myVar: Type` 则允许 VS Code 编码时自动识别变量类型并提示其成员。

### 更多属性类型声明方法

声明值类型

```
@property({
    type: cc.Integer
})
myInteger = 1;

@property
myNumber = 0;
```

```
@property
myText = "";

@property(cc.Node)
myNode: cc.Node = null;

@property
myOffset = new cc.Vec2(100, 100);
```

声明数组

```
@property([cc.Node])
public myNodes: cc.Node[] = [];

@property([cc.Color])
public myColors: cc.Color[] = [];
```

声明 getset

```
@property
_width = 100;

@property
get width () {
    return this._width;
}

@property
set width (value) {
    cc.log('width changed');
    return this._width = value;
}
```

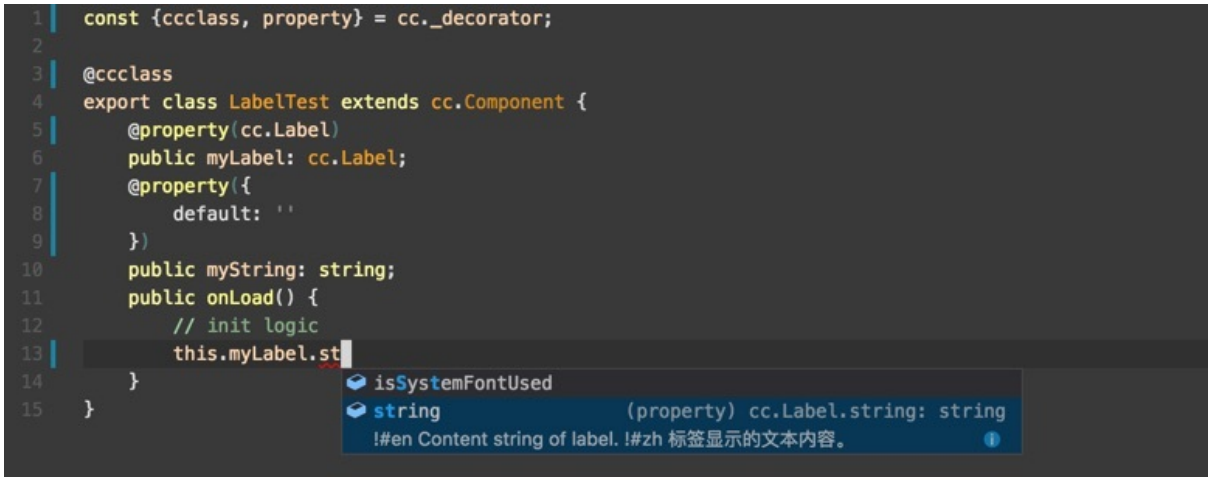
注意：TypeScript 的 public, private 修饰符不影响成员在 **属性检查器** 中的默认可见性，默认的可见性仍然取决于成员变量名是否以下划线开头。

# 完善的智能提示功能

按照 **使用准备** 里描述的方式创建项目或添加配置后，在 VS Code 里打开项目，就可以享受完善的代码智能提示功能了。

## 组件本身的属性成员

只要输入 `this.` 就会自动提示组件本身的其他成员，输入 `this.member.` 可以继续提示该成员的属性或方法




## 提示其他组件属性和方法

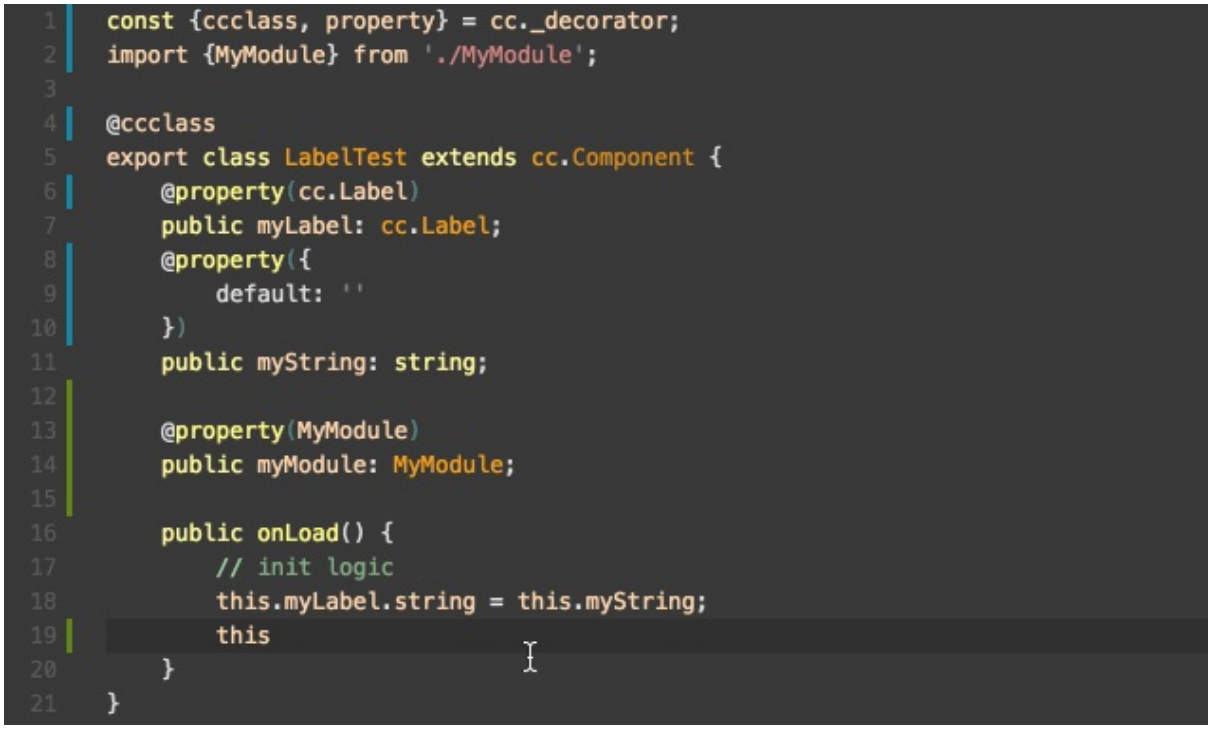
首先我们声明一个组件：

```
// MyModule.ts
const {ccclass, property} = cc._decorator;

@ccclass
export class MyModule extends cc.Component {
    @property(cc.String)
    myName : string = "";

    @property(cc.Node)
    myNode: cc.Node = null;
}
```

然后在其他组件中 import MyModule, 并且声明一个 `MyModule` 类型的成员变量：

```
// MyUser.ts
const {ccclass, property} = cc._decorator;
import {MyModule} from './MyModule';

@ccclass
export class MyUser extends cc.Component {
    @property(MyModule)
    public myModule: MyModule = null;

    public onLoad() {
        // init logic
        this.myModule.myName = 'John';
    }
}
```

输入 `this.myModule.` 时，就可以提示我们在 `MyModule.ts` 中声明的属性了。



```
const {ccclass, property} = cc._decorator;
import {MyModule} from './MyModule';

@ccclass
export class LabelTest extends cc.Component {
    @property(cc.Label)
    public myLabel: cc.Label;
    @property({
        default: ''
    })
    public myString: string;

    @property(MyModule)
    public myModule: MyModule;

    public onLoad() {
        // init logic
        this.myLabel.string = this.myString;
        this
    }
}
```

# 使用命名空间

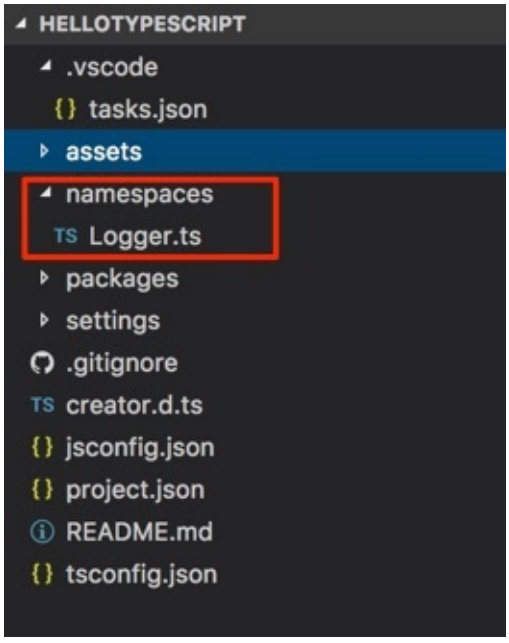
在 TypeScript 里，命名空间是位于全局命名空间下的一个普通的带有名字的JavaScript对象。通常用于在使用全局变量时为变量加入命名空间限制，避免污染全局空间。命名空间和模块化是完全不同的概念，命名空间无法导出或引用，仅用来提供通过命名空间访问的全局变量和方法。关于命名空间和模块化更详细的解释请参阅官方文档 命名空间和模块。

Creator 中默认所有 assets 目录下的脚本都会进行编译，自动为每个脚本生成模块化封装，以便脚本之间可以通过 `import` 或 `require` 相互引用。当希望把一个脚本中的变量和方法放置在全局命名空间，而不是放在某个模块中时，我们需要选中这个脚本资源，并在 **属性检查器** 里设置该脚本 `导入为插件` 。设为插件的脚本将不会进行模块化封装，也不会进行自动编译。

所以对于包含命名空间的 TypeScript 脚本来说，我们既不能将这些脚本编译并进行模块化封装，也不能将其设为插件脚本（会导致 TS 文件不被编译成 JS）。如果需要使用命名空间，我们需要使用特定的工作流程。

## 命名空间工作流程

1. 在项目的根目录下（assets 目录外），新建一个文件夹用于存放我们所有包含命名空间的 ts 脚本，比如

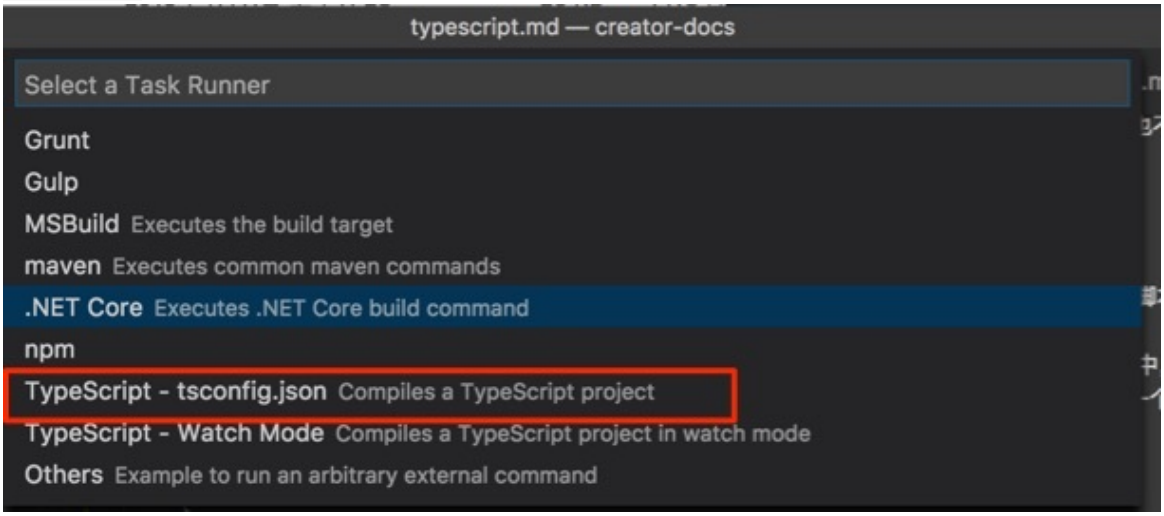



`namespaces` 。

2. 修改 `tsconfig.json` 文件，将刚创建的 `namespace` 文件夹加入到 `include` 字段中，表示我们将会通过 VSCode 编译这部分文件。
3. 在 `tsconfig.json` 的 `compilerOptions` 字段中，加入 `outFile` 字段，并设置一个 `assets` 文件夹下的文件路径。通过这些设置，我们会将所有 `namespace` 目录下的 ts 文件编译到 `assets` 目录下的某个 js 文件中。

```
{
    "compilerOptions": {
        "module": "commonjs",
        "lib": [ "dom", "es5", "es2015.promise" ],
        "target": "es5",
        "outFile": "./assets/Script/Lib/namespace.js",
        "experimentalDecorators": true
    },
    "include": [
        "namespaces"
    ]
}
```

4. 按下 `Ctrl/Cmd + Shift + P` ，在 Command Palette 里输入 `task` ，并选择 `Tasks: Configure Task Runner` 。在弹出的对话框里选择 `TypeScript - tsconfig` 。这将在 `.vscode` 文件夹下新建一个 `tasks.json` 配置文件，并配置根据 `tsconfig.json` 来编译项目中指定的 ts 脚本的任务。




5. 现在你可以在 `namespace` 目录下书写包含命名空间的 ts 脚本了，编程完成后按下 `Ctrl/Cmd + Shift + B` 触发默认构

建任务，就会将 `namespace` 里的脚本内容编译到 `assets` 目录下的指定文件里。 每次修改 `namespace` 中的脚本后，都应该执行构建任务来更新编译后的文件。

6. 回到 Creator 编辑器，在资源管理器里选中刚生成的 namespace 脚本 `namespace.js`，在属性检查里设置 「导入为插件」。避免编辑器对该脚本进行进一步的编译封装。

这就是在 Creator 里使用 TypeScript 命名空间的完整工作流程。

# 更新引擎接口声明数据

Creator 每个新版本都会更新引擎接口声明，建议升级了 Creator 后，通过主菜单的 `开发者 -> VS Code 工作流 -> 更新 VS Code 智能提示数据` 来更新已有项目的 `creator.d.ts` 文件。

---

Cocos Creator 中对 TypeScript 的支持参考了很多 Creator TypeScript Boilerplate 项目的设置和做法，在此特别感谢。另外这个项目中也包含了很多关于使用 TypeScript 项目的工作流程和高级功能，可供参考。

# CCClass 进阶参考

相比其它 JavaScript 的类型系统，CCClass 的特别之处在于功能强大，能够灵活的定义丰富的元数据。CCClass 的技术细节比较丰富，你可以在开发过程中慢慢熟悉。本文档将列举它的详细用法，阅读前需要先掌握 使用 cc.Class 声明类型。

## 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：包含“成员变量”和“成员方法”。
- 静态成员：包含“静态变量”和“类方法”。
- 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。

## 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

## 通过 ctor 定义

CCClass 的构造函数使用 `ctor` 定义，为了保证反序列化能始终正确运行， `ctor` **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。

## 通过 `__ctor__` 定义

`__ctor__` 和 `ctor` 一样，但是允许定义构造参数，并且不会自动调用父构造函数，因此用户可以自行调用父构造函数。 `__ctor__` 不是标准的构造函数定义方式，如果没有特殊需要请一律使用 `ctor` 定义。

# 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。

# 成员

## 实例变量

在构造函数中定义的实例变量不能被序列化，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();
// 调用实例方法
obj.print();
```

## 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，因为 count 继承自 Object 类

Sprite.range.w = 200;
cc.log(Object.range.w);  // 结果是 200，因为 Sprite.range 和 Object.range 指向同一个对象
```

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：

```
// 局部方法
function doLoad (sprite) {
    // ...
};
// 局部变量
var url = "foo.png";

var Sprite = cc.Class({
    load: function () {
        this.url = url;
        doLoad(this);
    };
});
```

# 继承

## 父构造函数

请注意，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，所以 this.name 已经被初始化过了
        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 Node = cc.Class({
    ctor: function () {
        this.name = "node";
```

```
    }
});
var Sprite = cc.Class({
    extends: Node,
    ctor: function () {
        Node.call(this);        // 别这么干!
        this._super();          // 也别这么干!

        this.name = "sprite";
    }
});
```

> 在一些很特殊的情况下，父构造函数接受的参数可能和子构造函数无法兼容。这时开发者就只能自己手动调用父构造函数并且传入需要的参数，这时应该将构造函数定义在 `__ctor__` 中。

## 重写

所有成员方法都是虚方法，子类方法可以直接重写父类方法：

```
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，父类被重写的方法并不会被 CCClass 自动调用，如果你要调用的话：

方法一：使用 CCClass 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 Shape = cc.Class({
    getName: function () {
        return "shape";
    }
});
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` 来实现继承。

# 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（序列化）的值。

## 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
2. `null` 或 `undefined`
3. 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(),
        }
    }
```

4. 空数组 `[]` 或空对象 `{}`
5. 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，序列化后就会将编辑器中设置好的值保存到场景等资源文件中，并且在加载场景时自动还原之前设置好的值。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 属性参数。

## 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

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

```
var Game = cc.Class({
    properties: {
        item: {
            default: null,
            type: Item
        }
    }
});

module.exports = Game;
```

- Item.js

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

var Item = cc.Class({
    properties: {
        game: {
            default: null,
            type: Game
        }
    }
});

module.exports = Item;
```

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

```
var Game = cc.Class({
    properties: () => ({
        item: {
            default: null,
            type: require("Item")
        }
    })
});

module.exports = Game;
```

- Item.js

```
var Item = cc.Class({
    properties: () => ({
        game: {
            default: null,
            type: require("Game")
        }
    })
});

module.exports = Item;
```

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")
        }
    };
}
```

# 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。

## get

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！
```

## set

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。

# 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，禁止同类型（含子类）的组件再添加到同一个节点，
    // 防止逻辑发生冲突。
    //
    // 值类型：Boolean
    // 默认值：false
    disallowMultiple: false,

    // 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: "",
  }
});
```

继续前往 属性参数参考。

# 属性参数

> 属性参数用来给已定义的属性附加元数据，类似于脚本语言的 Decorator 或者 C# 的 Attribute。

## 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 | |
| min | 限定数值在编辑器中输入的最小值 | number | undefined | |
| max | 限定数值在编辑器中输入的最大值 | number | undefined | |
| step | 指定数值在编辑器中调节的步长 | number | undefined | |
| range | 一次性设置 min, max, step | [min, max, step] | undefined | step 值可选 |
| slide | 在 **属性检视器** 面板中显示为滑动条 | boolean | false | |

## 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 | 当属性被赋值时触发指定方法 | `function (oldValue) {}` | undefined | 需要定义 default 属性<br>并且不能用于数组<br>不支持 ES6 定义方式 |
| override | 当重写父类属性时需要定义该参数为 true | boolean | false | 详见 override 参数 |

| | | | | |
|---|---|---|---|---|
| animatable | 该属性是否能被动画修改 | boolean | true | |

**注1:** visible 的默认值取决于属性名。当属性名以下划线 `_` 开头时，默认隐藏，否则默认显示。

---

继续前往 动画系统 或者返回 脚本开发。

# 跨平台发布游戏



---

继续前往 发布到 Web 平台 开始了解跨平台发布游戏的工作流程。

# 发布到 Web 平台

打开主菜单的 `项目/构建发布`，打开构建发布窗口。

Cocos Creator 提供了两种 Web 平台的页面模板，可以通过 **发布平台** 的下拉菜单选择 `Web Mobile` 或 `Web Desktop`，他们的区别主要在于 `Web Mobile` 会默认将游戏视图撑满整个浏览器窗口，而 `Web Desktop` 允许在发布时指定一个游戏视图的分辨率，而且之后游戏视图也不会随着浏览器窗口大小变化而变化。

## 发布路径

通过在 **发布路径** 输入框输入路径或通过 `...` 浏览按钮直接选择，我们可以为游戏指定一个发布路径，后续的多平台发布都会在这个发布路径中的子文件夹中创建资源或工程。

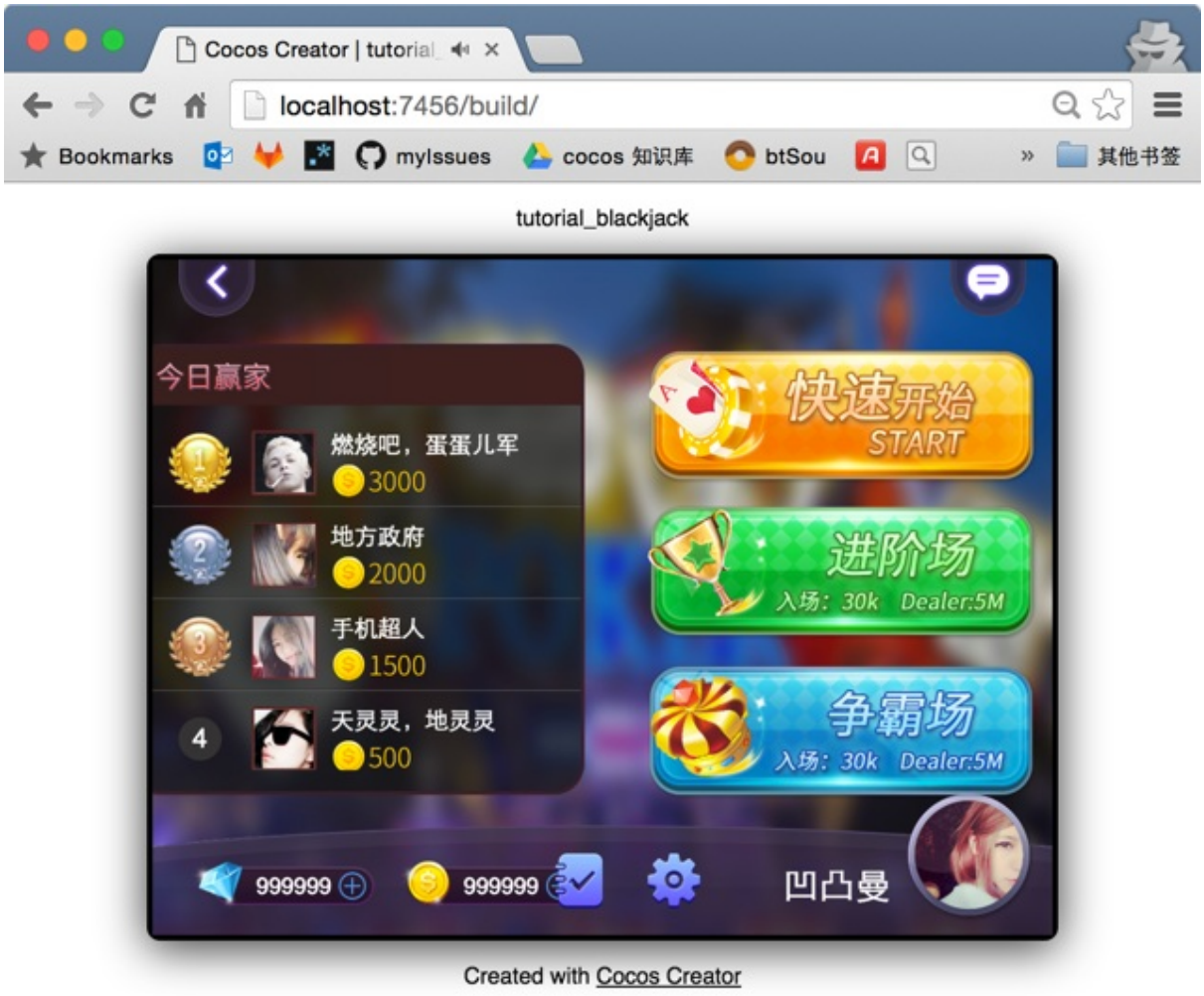
默认的发布路径在项目文件夹下的 `build` 文件夹中，如果您使用 git、svn 等版本控制系统，可以将 `build` 文件夹在版本控制中忽略。

## 构建和预览

Web 平台的构建非常简单，就是把游戏项目和资源库中的脚本和资源进行压缩后复制到指定的发布路径中。如果需要调试，也可以开启 **调试模式** 和 **Sourcemap** 的选项，这样构建出的版本会保留 sourcemap。

点击 **构建** 按钮，开始 Web 平台版本构建。面板上方会出现一个进度条，当进度条达到 100% 完成度时，构建就完成了。

接下来可以点击 **运行预览** 按钮，在浏览器中打开构建后的游戏版本进行预览和调试。




上图所示就是 Web Desktop 模式的预览，可以看到游戏视图是固定分辨率的，不会占满整个浏览器窗口。

### 浏览器兼容性

Cocos Creator 开发过程中测试的桌面浏览器包括： Chrome，Firefox（火狐），IE11 其他浏览器只要内核版本够高也可以正常使用，对部分浏览器来说请勿开启 IE6 兼容模式。

移动设备上测试的浏览器包括：Safari (iOS)，Chrome，QQ 浏览器，UC 浏览器，百度浏览器，微信内置 Webview。

## Retina 设置

在脚本中通过 `cc.view.enableRetina(true)` 可以控制是否使用高分辨率，构建到 Web 平台时默认会开启 Retina 显示，在不支持 WebGL 渲染的 Android 平台上，开启 Retina 会对帧率造成较大影响。

## 发布到 Web 服务器

要在互联网上发布或分享您的游戏，只要点击 **发布路径** 旁边的 **打开** 按钮，打开发布路径之后，将构建出的 `web-mobile` 或 `web-desktop` 文件夹里的内容整个复制到您的 Web 服务器上就可以通过相应的地址访问了。

关于 Web 服务器的架设，可以自行搜索 Apache、Nginx、IIS、Express 等相关解决方案。

要了解如何发布游戏到原生平台，请继续前往 安装配置原生开发环境 说明文档。

# 安装配置原生开发环境

除了内置的 Web 版游戏发布功能外，Cocos Creator 使用基于 cocos2d-x 引擎的 JSB 技术实现跨平台发布原生应用。在使用 Cocos Creator 打包发布到原生平台之前，我们需要先配置好 cocos2d-x 相关的开发环境。

# Android 平台相关依赖

要发布到 Android 平台，需要安装以下全部开发环境依赖。

如果您没有发布到 Android 平台的计划，或您的操作系统上已经有完整的 Android 开发环境，可以跳过这个部分。

## Android SDK 10 依赖

从 v1.2.2 开始，默认的 Android 项目模板将指定 `android-10` sdk platform 版本作为默认的 target，详情可见 Pull Request Use API Level 10。

如果编译 Android 工程时遇到 '未找到 android-10' 之类的报错，可以通过下文介绍的方式下载 Android SDK API Level 10。

如果需要更改 target 的 API Level，可以修改原生引擎目录下 `cocos/platform/android/java/project.properties` 文件中的

```
target=android-10
```

将 `android-10` 修改为其他您需要的 API Level。

## 下载安装 Android Studio

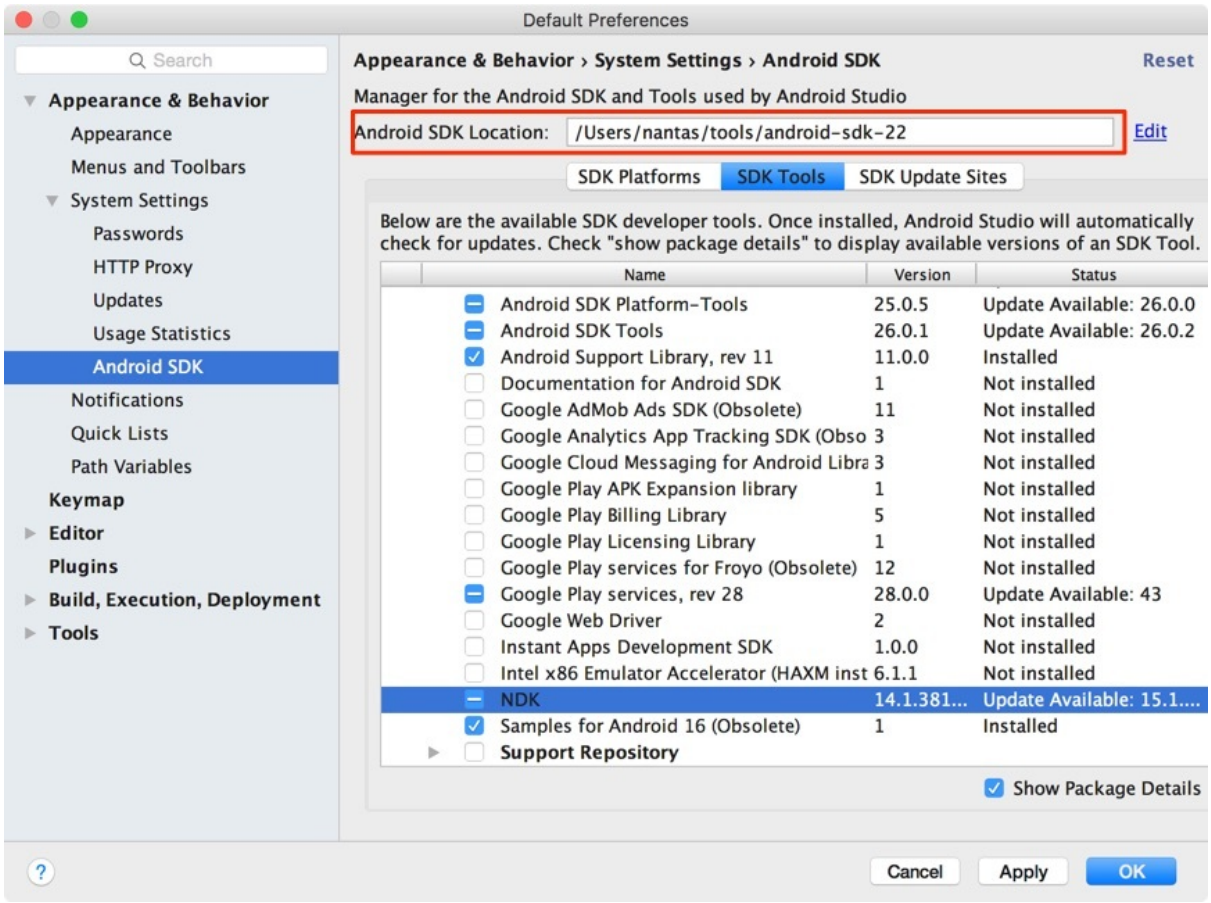
从 v1.5 开始，我们支持最新版本的 Android Studio 和配套的构建工具，推荐使用 Android Studio 作为安卓平台的构建工具，并在 Android Studio 里下载所需的 SDK 和 NDK 包。首先请 安装 Android Studio。

## 下载发布 Android 平台所需的 SDK 和 NDK

安装 Android Studio 完成后，参考官方文档，打开 SDK Manager：

SDK Manager 使用说明

1. 在 SDK Platforms 分页栏，勾选你希望安装的 API Level，也就是支持安卓系统的版本，推荐选择最低兼容的 API Level 10（2.3.3) 和最主流的 API Level 17 (4.2) 以及 API Level 22 (5.1)。
2. 在 SDK Tools 分页栏，首先勾选右下角的 `Show package details`，显示分版本的工具选择。
3. 在 `Android SDK Build-Tools` 里，选择 25 以上的 build tools 版本。
4. 勾选 `Android SDK Platform-Tools`，`Android SDK Tools` 和 `Android Support Library`
5. 勾选 `NDK`，确保版本在 14 以上。
6. 记住窗口上方所示的 Android SDK Location 指示的目录，稍后我们需要在 Cocos Creator 里填写这个 SDK 所在位置。
7. 点击 `OK`，根据提示完成安装。




### 下载 Java SDK （JDK）

编译 Android 工程需要本地电脑上有完整的 Java SDK 工具，请到以下地址下载：

Java SE Development Kit 8 Downloads

下载时注意选择和本机匹配的操作系统和架构，下载完成后运行安装程序即可。

**注意**：安装完成后请在命令行中确认 `java` 命令是有效的，否则请参考下文手动添加 `java` 可执行文件所在目录到您的环境变量中。

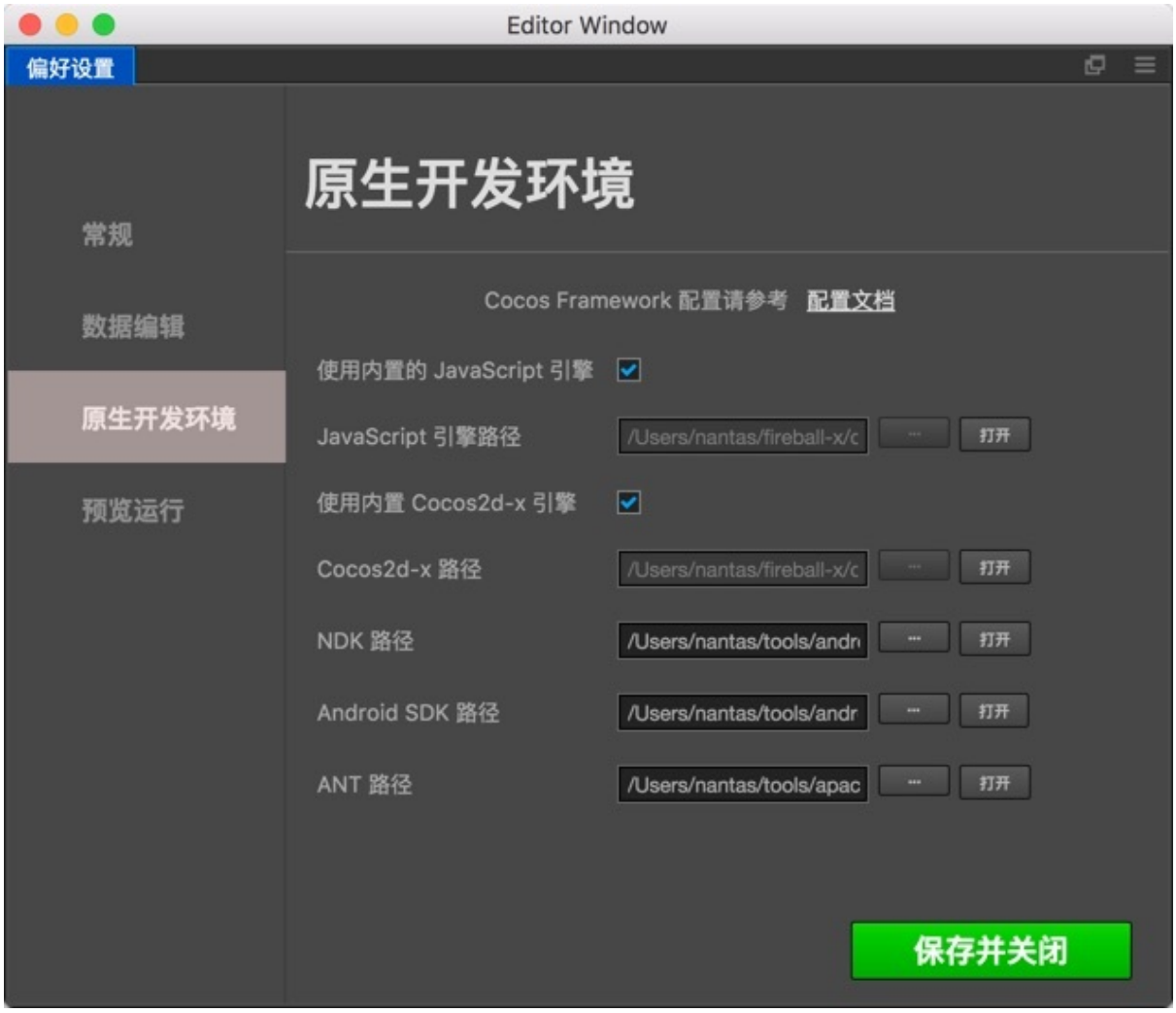
## 安装 C++ 编译环境

Cocos2d-x 自带的编译工具 Cocos Console 需要以下运行环境：

- Python 2.7.5+，下载页，注意不要下载 Python 3.x 版本。
- Windows 下需要安装 Visual Studio 2015 或 2017 社区版，下载页
- Mac 下需要安装 Xcode 和命令行工具，下载页

## 配置原生发布环境路径

下载安装好开发环境依赖后，让我们回到 Cocos Creator 中配置构建发布原生平台的环境路径。在主菜单中选择 `文件/偏好设置`，打开偏好设置窗口：




我们在这里需要配置以下三个路径：

- **Android SDK**，选择刚才在 SDK Manager 中记下的 `Android SDK Location` 路径，不需要编译 Android 平台的话这里可以跳过
- **NDK**，选择 `Android SDK Location` 路径下的 `ndk-bundle` 文件夹，不需要编译 Android 平台的话这里可以跳过
- **ANT**（如使用 Android Studio，这一步可以跳过），请选择下载并解压完成的 Apache Ant 路径，选定的路径中应该包括一个名叫 `ant` 的可执行文件。

配置完成后点击 **保存** 按钮，保存并关闭窗口。

**注意**：这里的配置会在编译 **原生工程** 的时候生效。如果没有生效（一些 Mac 机器有可能出现这个情况），可能需要您尝试到 **系统环境变量** 设置这些值：COCOS_CONSOLE_ROOT, ANT_ROOT, NDK_ROOT, ANDROID_SDK_ROOT。

# 注意事项

由于在公测版中收到了很多原生打包的问题反馈，这里补充一些可能的问题原因。

1. 检查路径

   在偏好设置中设置好的路径必须确保是正确的，比如：ant 路径需要设置到 ant 安装目录内的 bin 目录下，NDK 是其根目录，而 Android SDK 的目录下应该包含 build-tools、platforms 等文件夹。

2. 检查 Xcode 和 Visual Studio

打包 Mac 版本和 iOS 版本需要 Xcode 支持。打包 Windows 版本需要安装 Visual Studio，同时在安装 Visual Studio 时，默认并没有勾选 C++ 编译组件。如果没有安装，则需要重新安装并选择 C++ 相关编译组件。

3. 检查 NDK 版本

   请使用 NDK r10c 以上的版本，推荐 r10e。

4. Windows 平台需要检查 JAVA_HOME 环境变量

   如果使用 Windows 平台，请确认你的环境变量中包含 JAVA_HOME，可以通过右键点击我的电脑，选择属性，打开高级选项卡中来查看和修改环境变量。Windows 平台可能需要重启电脑才会生效。

   参考如何设置或更改 JAVA 系统环境变量?

5. 检查 JAVA 环境

   在 Mac 终端或者 Windows 命令行工具中输入下面代码来查看：

   ```
   java -version
   ```

   如果显示为 JAVA SE 则没有问题，如果系统中使用的是 JRE，则需要安装 JAVA SE 运行环境。

6. 包名问题

   检查构建发布面板中的包名，包含空格，`-` 等都是非法的包名。

7. 不使用 Android Studio

   如果您使用 Cocos Creator v1.5 以前的版本，或由于某些原因无法使用 Android Studio，请安装 Eclipse 并使用旧的流程下载 SDK 和 NDK

   从以下链接下载和操作系统一致的 Android SDK 和 NDK：

   - Android SDK Windows
   - Android SDK Mac
   - Android NDK Windows 32位
   - Android NDK Windows 64位
   - Android NDK Mac

     下载之后解压到任意位置，我们之后需要设置 Android SDK 和 NDK 的路径，请记住以上文件的解压位置。

     下载 Apache Ant 是一种用来构建软件的 Java 程序库和可执行文件。我们在构建 Android 平台项目时需要这个软件的支持。

     前往 Apache Ant 的下载链接：

     Apache Ant 下载

     选择稳定版的 `.zip` 压缩包并下载，下载完成后解压到任意目录，之后我们在进行设置时需要选择这个目录。

8. Android 6.0 SDK 的支持问题

   Android 6.0 SDK 去除了 Cocos2d-x 依赖的 HttpClient 库，所以会导致 Cocos Creator v1.5 以前的版本编译失败。旧版本用户的解决方案是：

   - 找到 Android SDK 目录下的 HttpClient 库：`platforms/android-23/optional/org.apache.http.legacy.jar` 。
   - 如果使用源码引擎模版，需要拷贝到原生编译目录下的 `jsb/frameworks/cocos2d-x/cocos/platform/android/java/libs/` 目录下。如果使用预编译库引擎模版，需要拷贝到原生编译目录下的 `jsb/frameworks/runtime-src/proj.android/jars/` 目录下。
   - 重新编译。

9. Android 编译成功，但运行时提示 `dlopen failed: cannot locate symbol "xxxx" referenced by "libcocos2djs.so"...`

> 请检查 NDK 和 Android SDK 的架构和版本是否和测试用的 Android 系统相对应，另外可以尝试使用本文章下载的 NDK 和 Android SDK 来测试。

最后，如果依然打包失败，可以尝试创建一个标准的 Cocos2d-x 工程，并尝试编译，如果 Cocos2d-x 工程可以编译，而 Cocos Creator 无法打包，请将 bug 通过论坛反馈给我们。

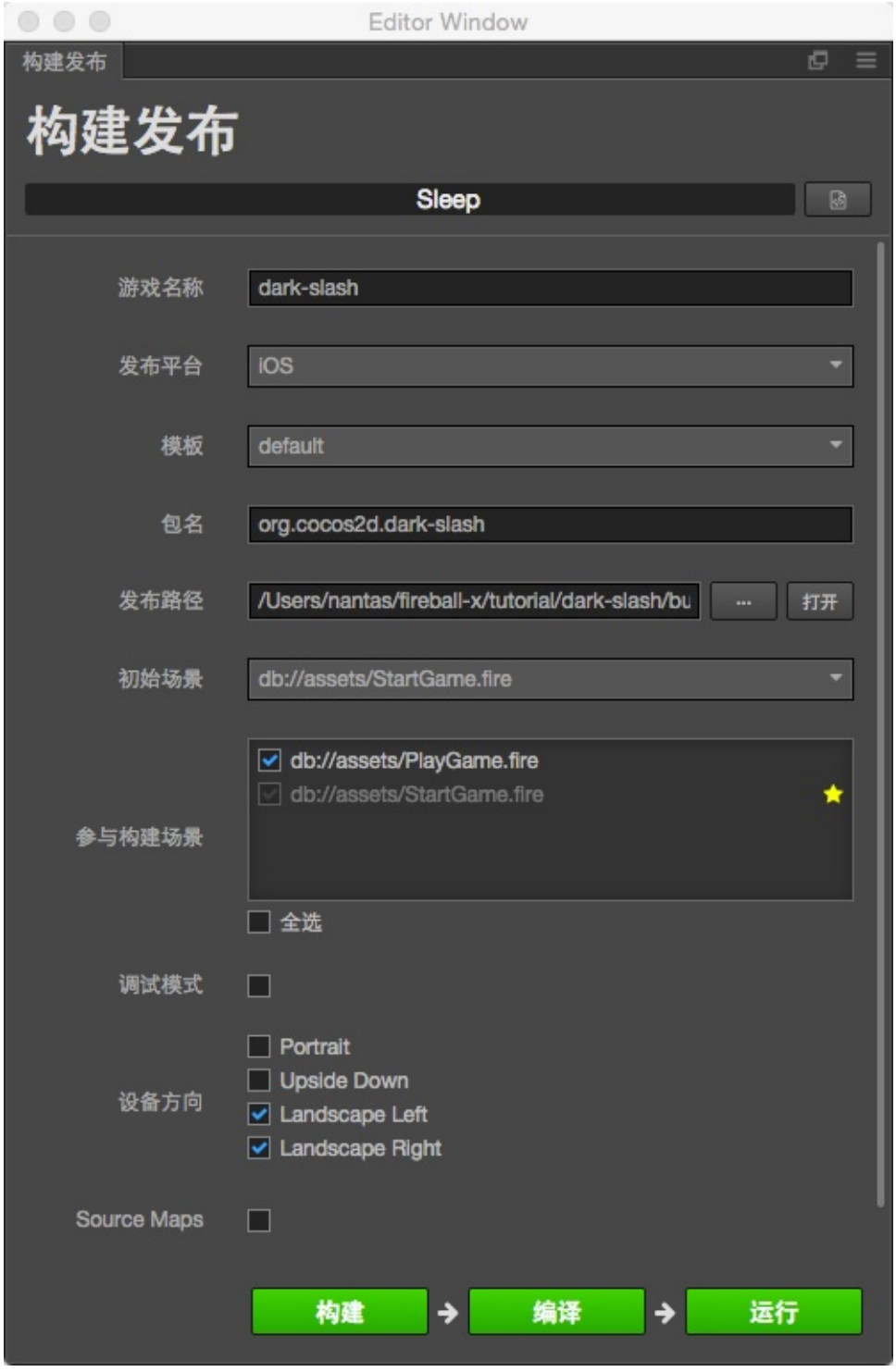
---

现在您已经完成了全部原生开发环境的配置，接下来请继续前往 打包发布原生平台 说明文档。

# 打包发布原生平台

打开主菜单的 `项目/构建发布`，打开构建发布窗口。

目前可以选择的原生平台包括 Android, iOS, Mac, Windows 四个，其中发布到 Mac 和 Windows 的选项只能在相应的操作系统中才会出现。

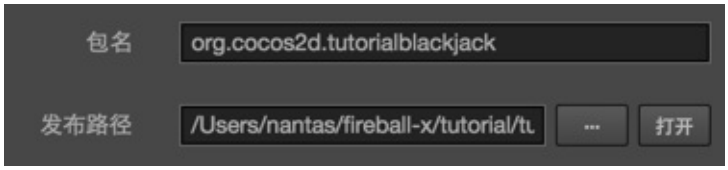


# 设置包名（Package Name）

选择一个原生平台后，构建发布窗口中会显示 **包名** 的输入框，在这里请输入您游戏的包名（也称作 Package Name 或 Bundle ID），通常以产品网站 url 倒序排列，如 `com.mycompany.myproduct` 。

**注意**：

- 包名中只能包含数字、字母和下划线，此外包名最后一部分必须以字母开头，不能以下划线或数字开头。
- 如果使用的 xcode 的版本低于 7.2(目前测试到 7.2 以上版本不会有问题，其他版本不能保证)，那么包名包含下划线的话也可能会编译失败。




# 选择源码或预编译库模板

接下来在 **模板** 下拉菜单里，我们可以从引擎模板中选择一个，下面是可用的三种选项：

- default，使用默认的 cocos2d-x 源码版引擎构建项目
- binary，使用预编译好的 cocos2d-x 库构建项目
- link，与 default 模板不同的是，link 模板不会拷贝 cocos2d-x 源码到构建目录下，而是使用共享的 cocos2d-x 源码。这样可以有效减少构建目录占用空间，以及对 cocos2d-x 源码的修改可以得到共享。

## 源码引擎和预编译库

在 v3.10 之后，cocos2d-x 引擎中现在包括源码引擎和预编译引擎（各个平台原生支持的二进制库格式）。他们适用的范围是：

- 源码引擎初次构建和编译某个工程时需要很长的时间编译 C++ 代码，视电脑配置而定，这个时间可能在 20 分钟到 1 小时。对于同一个项目，已经编译过一次之后，下次再编译需要的时间会大大缩短。
- 源码引擎构建出的工程，使用原生开发环境编译和运行（如 Eclipse，Xcode 等 IDE），是可以进行调试和错误捕获的。
- 预编译引擎库，由于事先已经把 cocos2d-x 引擎编译完成了，构建项目时将直接使用编译完的库，可以将编译时间大大缩短。
- 但是预编译引擎库不包含源码，因此也无法在原生工程中进行调试。

目前 Cocos Creator 安装目录下已经包含了自带的 cocos2d-x 源码引擎（v3.10+），如果您希望使用预编译引擎库进行日常构建和编译，可以通过主菜单的 `开发者/编译 cocos2d-x 预编译库` 来进行手动编译。

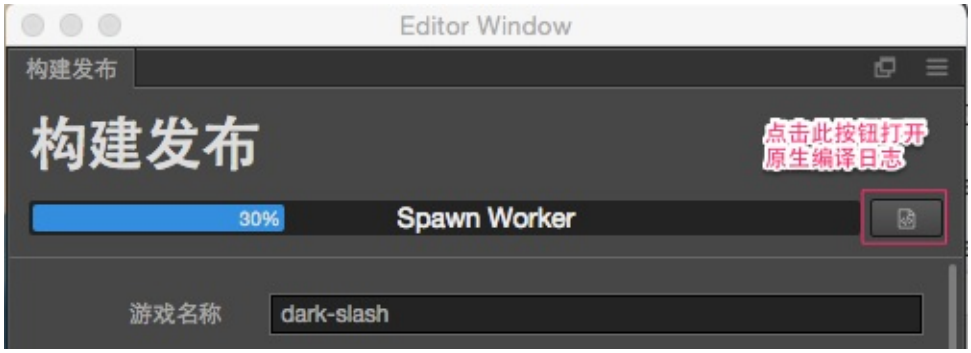
编译预编译库的过程会比较长，与此同时会占用大量的 CPU 资源，建议在不需要工作的时候再进行。另外编译的过程是在一个独立的进程中通过 Cocos Console 命令行进行的，如果您想了解编译的进度，请点击 **控制台** 面板的日志按钮，并选择 **Cocos Console 日志** 来打开相应的日志文件。

cocos console log

编译完成后会在 **控制台** 输出成功的信息，之后您就可以在 **构建发布** 界面的 **模板** 选项里选择 `binary` 来使用预编译库构建和编译原生项目了。

# 构建原生工程

选择发布平台，设置了包名和初始场景后，就可以开始构建了，点击右下角的 **构建** 按钮，开始构建流程。




编译脚本和打包资源时会在窗口上方显示进度条，进度条到达 100% 后请继续等待 **控制台** 面板中的工程构建结束，成功的话会显示如下所示的日志：

```
Built to "/myProject/tutorial-blackjack/build/tutorial-blackjack" successfully
```

构建结束后，我们得到的是一个标准的 cocos2d-x 工程，和使用 Cocos Console 新建的工程有同样的结构。接下来我们可以选择通过 Cocos Creator 编辑器的进程进行编译，以及运行桌面预览，或手动在相应平台的 IDE 中打开构建好的原生工程，进行进一步的预览、调试和发布。

# 通过编辑器编译和预览

点击下方的 **编译** 按钮，进入编译流程，如果模板选择了 `default` 的源码版引擎，这个编译的过程将会花费非常久的时间。编译成功后会提示

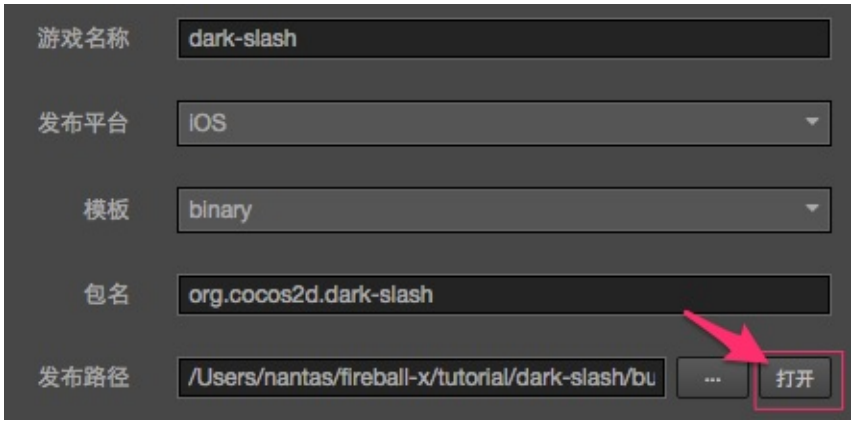
```
Compile native project successfully
```

接下来就可以点击右下角的 **运行** 按钮，通过默认方式预览原生平台的游戏。


其中 Mac/iOS/Windows 平台会使用 Cocos Simulator 模拟器在桌面运行预览，Android 平台必须通过 USB 连接真机，并且在真机上开启 USB 调试后才可以运行预览。

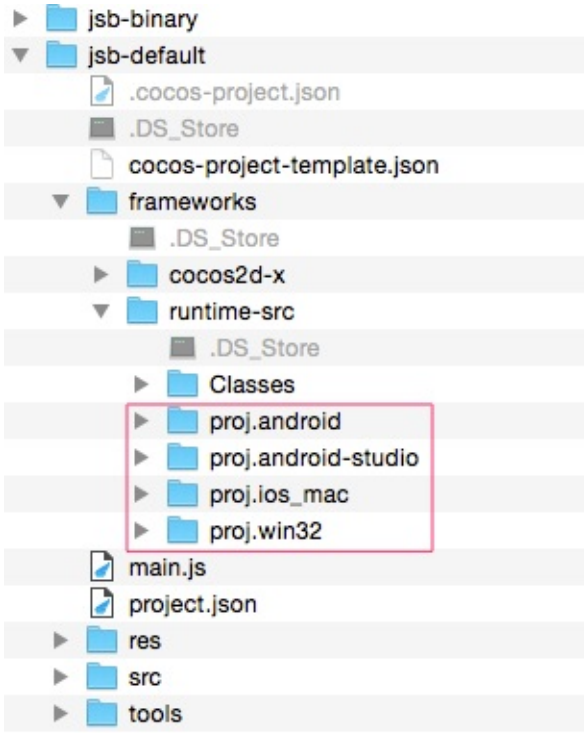
点击运行后，视平台不同可能还会继续进行一部分编译工作，请耐心等待或通过日志文件查看进展。

# 使用原生工程




点击发布路径旁边的 **打开** 按钮，就会在操作系统的文件管理器中打开构建发布路径。

这个路径中的 `jsb-default` 或 `jsb-binary` （根据选择模板不同）里就包含了所有原生构建工程。




图中红框所示的就是不同原生平台的工程，接下来您只要使用原生平台对应的 IDE （如 Xcode、Eclipse、Android Studio、Visual Studio）打开这些工程，就可以进行进一步的编译、预览、发布操作了。关于原生平台 IDE 的使用请搜索相关信息，这里就不再赘述了。

---

要了解如何在原生平台上调试，请继续前往 原生平台调试 说明文档。

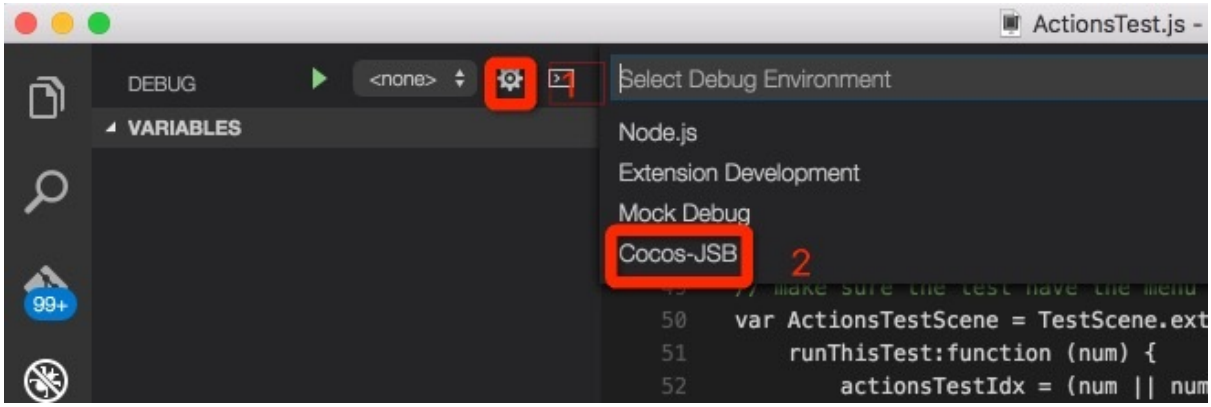
# 原生平台调试

目前使用 Cocos Creator 进行游戏开发时，游戏脚本以 JavaScript 形式编写，运行在原生平台时需要经过 JSB 绑定，然后通过原生 C++ 引擎代码运行，由于运行平台不同，可能会出现和 Web 浏览器运行效果不同的地方和错误。我们可以借助 VS Code 的调试功能来调试我们的 JSB 游戏程序。

## 安装 Cocos-Debug VS Code 插件

在 Cocos Creator 的开发者菜单中，选择 VS Code 工作流，再选择安装 VS Code 扩展插件，插件就会被自动安装到用户文件夹的 `.vscode/extensions` 路径下，然后启动 VS Code。

安装插件成功后，在 VS Code 左侧 tab 中选择第四个 **Debug**，然后可以点击上面的齿轮选择 `Cocos-JSB` 调试方案。




如果安装插件不成功，可以查看 VS Code 官网的插件安装指南文档。

## 模拟器调试

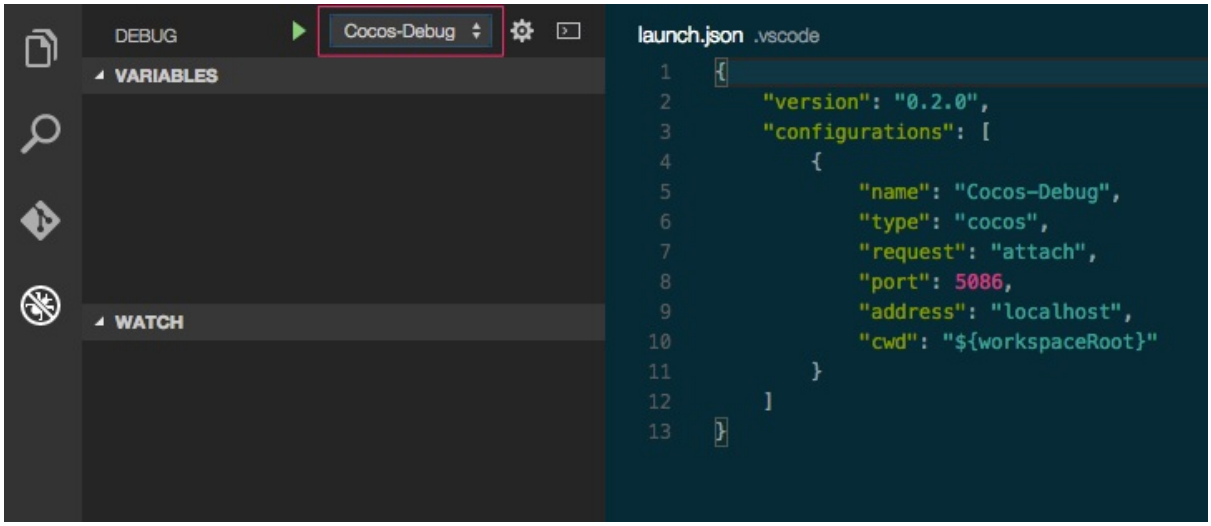
首先在编辑器工具栏正上方选择使用 **模拟器（调试）** 作为预览平台，然后点击编辑器中的 **运行预览** 按钮在模拟器中运行游戏。

然后找到模拟器中的项目资源路径，这个路径会在 Cocos Creator 安装路径中，根据操作系统的不同，路径有一些区别：

- Windows：`CocosCreator/resources/cocos2d-x/simulator/win32`
- Mac：`CocosCreator.app/Contents/Resources/cocos2d-x/simulator/mac/Simulator.app/Contents/Resources`

在 Mac 平台由于 VS Code 无法通过浏览选择应用程序包（.app）之内的路径，可能需要先打开 VS Code，关闭所有已开启的文件夹，然后将 Finder 里的上述路径拖拽到 VS Code 中。

然后就可以在打开的工程中的 `src/project.dev.js` 文件中设置断点并进行调试了。先设置好断点，然后确保模拟器已经在运行的情况下，VS Code 中切换到 Debug 页面，在下拉菜单选中 `Cocos-Debug`，然后点击绿色三角按钮开启调试进程。




现在可以进行正常的 JSB 程序调试了。

关于 VS Code 调试功能的使用，请参考VS Code 调试指南文档。

### Windows socket 链接错误解决方法

在 Windows 上启动调试器时可能会链接失败并提示：

```
ar: attachRequest: retry socket.connect
```

可以尝试点击 Debug 页面的齿轮按钮，并在 `launch.json` 配置文件中修改 `address` 字段为：

```
"address": "::1"
```

原因是在 Windows 上部分链接会默认使用 IPv6，因此我们需要填写 IPv6 格式的本地默认 ip 地址。

## 原生工程调试

构建发布出原生工程以后，对包括 iOS、Android、Windows、Mac 等原生平台都可以进行调试。首先要用 VS Code 打开构建发布出的原生工程，一般在

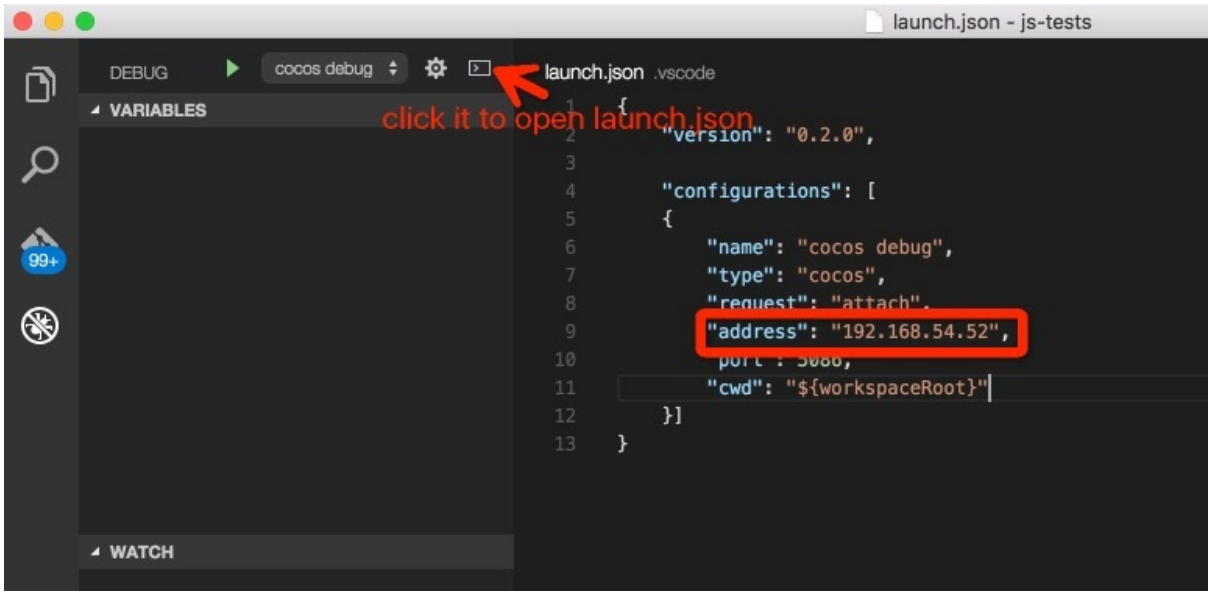
`myProject/build/myProject/jsb-default` 或 `myProject/build/myProject/jsb-binary`，根据构建工程时选择的引擎模板而定。

在真机或桌面模拟器上运行后，流程就和模拟器调试类似了，请在 `jsb-default/src/project.dev.js` 中设置断点和调试。下面是移动设备调试时的注意事项。

### iOS 调试

要调试 iOS 设备：

- 通过 USB 连接线将设备和电脑链接起来
- 通过 Xcode 在设备上运行游戏
- 配置 VS Code 中 cocos-debug 调试方案的 `launch.json` 文件，将 `address` 字段改为您 iOS 设备的 IP 地址




## Android 调试

要调试 Android 设备：

- 通过 USB 连接线将设备和电脑链接起来
- Adb 在设备上运行游戏（可以通过 **构建发布** 面板，在构建和编译完成后点击运行）
- 使用 adb 命令转发调试端口： `adb forward tcp:5086 tcp:5086`

# 使用限制

VS Code 只能挂载到运行中的 cocos2d-x JSB 程序，才能开始调试，因此目前不能在启动时就停在断点处。以后我们会改善这一点，目前可以在游戏中手动设置启动按钮来回避这个问题。

# 命令行发布项目

通过命令行发布项目可以帮助大家构建自己的自动化构建流程，大家可以修改命令行的参数来达到不同的构建需求。

# 命令行发布参考

- Mac - `/Applications/CocosCreator.app/Contents/MacOS/CocosCreator --path projectPath --build "platform=android;debug=true"`
- Windows - `CocosCreator/CocosCreator.exe --path projectPath --build "platform=android;debug=true"`

如果希望在构建完原生项目后自动开始编译的话，可以使用 `autoCompile` 参数

- `--build "autoCompile=true"`

也可以自己开始编译项目， `--compile` 命令的参数和 `--build` 命令的参数一致

- `--compile "platform=android;debug=true"`

# 构建参数

- `path` - 指定项目路径
- `build` - 指定构建项目使用的参数。这里会使用 Creator 中构建面板当前的参数来作为默认构建参数，如果指定了其他参数，则会使用指定的参数来覆盖默认参数。
- `compile` - 指定编译项目使用的参数。这里会使用 Creator 中构建面板当前的参数来作为默认构建参数，如果指定了其他参数，则会使用指定的参数来覆盖默认参数。

---

`--build` 和 `--compile` 可选择的参数有：

- `excludedModules` - engine 中需要排除的模块，模块可以从 这里 查找到
- `title` - 项目名
- `platform` - 构建的平台 [web-mobile, web-desktop, android, win32, ios, mac, runtime]
- `buildPath` - 构建目录
- `startScene` - 主场景的 uuid 值
- `debug` - 是否为 debug 模式
- `previewWidth` - web desktop 窗口宽度
- `previewHeight` - web desktop 窗口高度
- `sourceMaps` - 是否需要加入 source maps
- `webOrientation` - web mobile 平台下的旋转选项 [landscape, portrait, auto]
- `renderMode` - 设置渲染类型
  - 0 - 由引擎自动选择.
  - 1 - 强制使用 Canvas 渲染模式
  - 2 - 强制使用 Webgl 渲染模式，但是在移动浏览器中这个选项会被忽略
- `inlineSpriteFrames` - 是否内联所有 SpriteFrame
- `mergeStartScene` - 是否合并初始场景依赖的所有 JSON
- `optimizeHotUpdate` - 是否将图集中的全部 SpriteFrame 合并到同一个包中
- `packageName` - 包名

- `vsVersion` - 设置使用的 visual studio 版本，只在 windows 上有用，可选的选项有 [auto, 2013, 2015, 2017]
- `useDebugKeystore` - 是否使用 debug keystore
- `keystorePath` - keystore 路径
- `keystorePassword` - keystore 密码
- `keystoreAlias` - keystore 别名
- `keystoreAliasPassword` - keystore 别名密码
- `orientation` - native mobile 平台下的旋转选项 [portrait, upsideDown, landscapeLeft, landscapeRight]
  因为这是一个 object, 所以定义会特殊一些。
  - orientation={'landscapeLeft': true} 或
  - orientation={'landscapeLeft': true, 'portrait': true}
- `template` - native 平台下的模板选项 [default, link, binary]
- `apiLevel` - 设置编译 android 使用的 api 版本
- `appABIs` - 设置 android 需要支持的 cpu 类型，可以选择一个或多个选项 [armeabi, armeabi-v7a, arm64-v8a, x86]
  因为这是一个数组类型，数据类型需要像这样定义，注意选项需要用引号括起来
  - appABIs=['armeabi', 'armeabi-v7a']
- `androidStudio` - 是否使用 android studio 来编译 android 项目
- `includeAnySDK` - web 平台下是否加入 AnySDK 代码
- `oauthLoginServer` - AnySDK 验证登陆服务器
- `appKey` - AnySDK App Key
- `appSecret` - AnySDK App Secret
- `privateKey` - AnySDK Private Key
- `includeEruda` - 是否在 web 平台下插入 Eruda 调试插件
- `autoCompile` - 是否在构建完成后自动进行编译项目。默认为 **否**。
- `configPath` - 参数文件路径。如果定义了这个字段，那么构建时将会按照 `json` 文件格式来加载这个数据，并作为构建参数

# 定制项目构建模板

Creator 支持对每个项目分别定制构建模板，只需要在需要定制的项目路径下添加一个 `build-templates` 目录，里面按照平台路径划分子目录，然后里面的所有文件在构建结束后都会自动按照对应的目录结构复制到构建出的工程里。

结构类似：

```
project-folder
 |--assets
 |--build
 |--build-templates
      |--web-mobile
            |--index.html
      |--jsb-binary
            |--main.js
      |--jsb-default
            |--main.js
```

这样如果当前构建的平台是 `web-mobile` 的话，那么 `build-templates/web-mobile/index.html` 就会在构建后被拷贝到 `build/web-mobile/index.html` 。

# 图像和渲染

# 基本图像渲染

包括以下渲染组件：

- Sprite 组件参考
- Label 组件参考
- Mask 组件参考

# 外部资源渲染

包括以下渲染组件：

- ParticleSystem 组件参考
- TiledMap 组件参考
- Spine 组件参考
- DragonBones 组件参考
- VideoPlayer 组件参考
- WebView 组件参

# Camera 摄像机

摄像机组件在制作卷轴或是其他需要移动屏幕的游戏时比较有用，在没有摄像机组件的情况下，卷轴游戏都是通过移动场景节点或是游戏根节点来实现的，这样将会导致大量节点的矩阵都需要重新计算，效率自然会有所降低，而使用摄像机是直接将摄像机的矩阵信息在渲染阶段统一处理，将会比移动节点来移动屏幕更加高效。

摄像机组件提供了两个属性来供用户设置：

- targets - 指定摄像机会拍摄哪些节点，即摄像机会影响哪些节点。
- zoomRatio - 指定摄像机的缩放比例，值越大显示的图像越大。

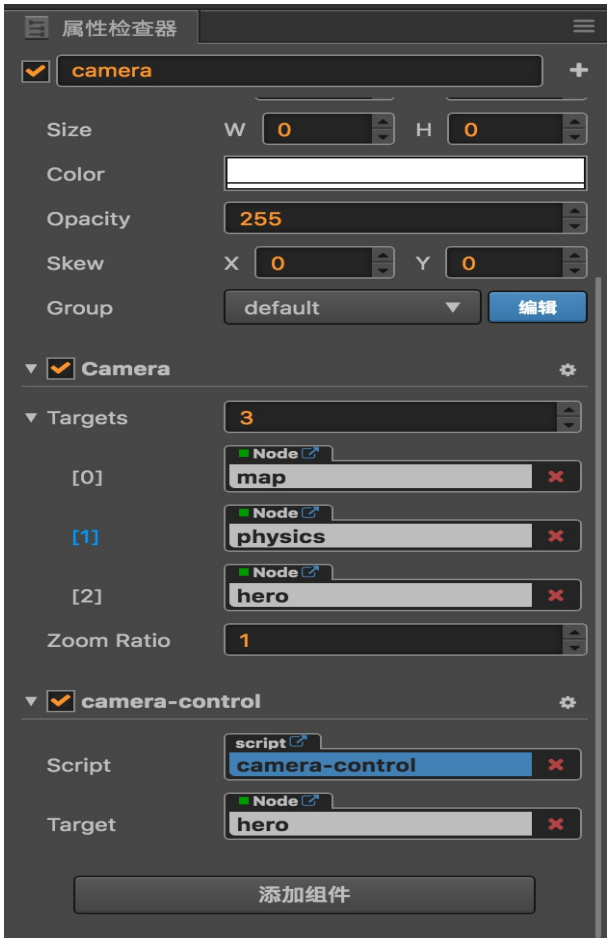
摄像机组件将会随着他依附的节点进行移动，可以想象成我们举着摄像机跟随者摄像机的节点移动，而这个摄像机只会拍摄他的 targets 目标，摄像机组件拍摄的范围即是设备屏幕大小。

## 实例

我们用一个场景实例来解释摄像机组件怎么使用。

假设我们在做一个物理游戏，需要 physics 节点，tiled map 做背景，hero 做主角，我们的摄像机需要跟随 hero 来移动。


这里我们还新建了一个 camera 节点来作为摄像机的载体，使用一个单独的节点作为摄像机节点会更灵活，当然我们也可以直接将摄像机组件添加到 hero 节点上，但是这样摄像机的位置就只能和 hero 节点重叠在一起了，不能做到缓慢跟随之类的效果。




在这里摄像机组件添加了三个节点到 targets 上，即我们需要摄像机拍摄这三个节点。并且我们还添加了一个 **camera-control** 组件，这个组件的作用主要是移动 camera 节点跟随 hero 节点。

实例可在 物理系统示例 中的 tiled 示例中找到。

**注意**：

当我们使用摄像机时，如果使用到了物理系统或者碰撞系统这些会内置渲染节点的系统，需要调用相关的 api 将他们的渲染节点也添加到摄像机上。

```
cc.director.getPhysicsManager().attachDebugDrawToCamera(camera);
cc.director.getCollisionManager().attachDebugDrawToCamera(camera);
```

# 绘图系统

本章将详细介绍 Cocos Creator 的绘画组件接口的使用。

# 绘图接口

## 路径

| 方法 | 功能说明 |
| --- | --- |
| moveTo (x, y) | 把路径移动到画布中的指定点，不创建线条 |
| lineTo (x, y) | 添加一个新点，然后在画布中创建从该点到最后指定点的线条 |
| bezierCurveTo (c1x, c1y, c2x, c2y, x, y) | 创建三次方贝塞尔曲线 |
| quadraticCurveTo (cx, cy, x, y) | 创建二次贝塞尔曲线 |
| arc (cx, cy, r, a0, a1, counterclockwise) | 创建弧/曲线（用于创建圆形或部分圆） |
| ellipse (cx, cy, rx, ry) | 创建椭圆 |
| circle (cx, cy, r) | 创建圆形 |
| rect (x, y, w, h) | 创建矩形 |
| close () | 创建从当前点回到起始点的路径 |
| stroke () | 绘制已定义的路径 |
| fill () | 填充当前绘图（路径） |
| clear () | 清除所有路径 |

## 颜色，样式

| 属性 | 功能说明 |
| --- | --- |
| lineCap | 设置或返回线条的结束端点样式 |
| lineJoin | 设置或返回两条线相交时，所创建的拐角类型 |
| lineWidth | 设置或返回当前的线条宽度 |
| miterLimit | 设置或返回最大斜接长度 |
| strokeColor | 设置或返回笔触的颜色 |
| fillColor | 设置或返回填充绘画的颜色 |

# 第三方库

绘图组件的 api 是参考的 Canvas 的绘图接口，而市面上已经有很多基于 Canvas 绘图接口实现的绘图库，比如 paper.js, raphael.js。 利用这些基础绘图接口和市面上的这些绘画库，我们也可以在绘图组件上扩展出很多更高级的库。

这里列举了一些基于绘图组件扩展的第三方高级绘图库和相关 demo 。

## ccc.raphael

- github：https://github.com/2youyou2/ccc.raphael
- demo：https://github.com/2youyou2/raphael-example
- 特性（持续更新中）
  - 线条变形



  - 线条虚线



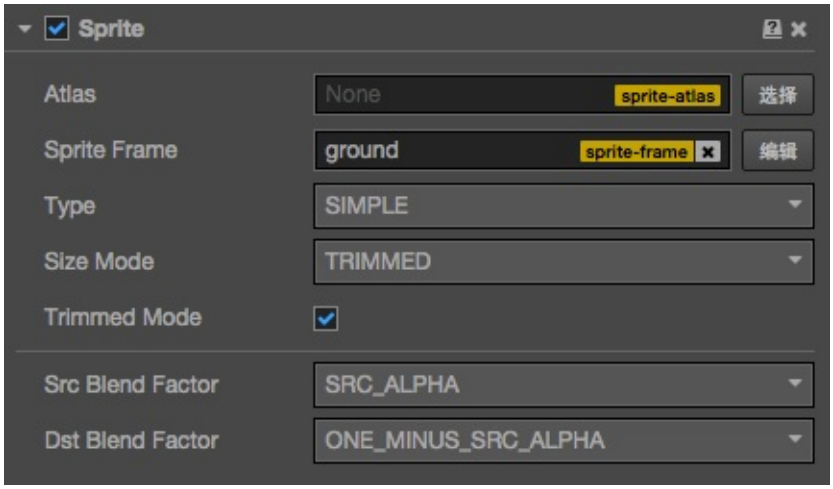
  - 简化路径



  - 读取 svg

# Sprite 组件参考

Sprite（精灵）是 2D 游戏中最常见的显示图像的方式，在节点上添加 Sprite 组件，就可以在场景中显示项目资源中的图片。




点击**属性检查器**下面的 `添加组件` 按钮，然后从 `添加渲染组件` 中选择 `Sprite`，即可添加 Sprite 组件到节点上。

脚本接口请参考Sprite API。

# Sprite 属性

| 属性 | 功能说明 |
| --- | --- |
| Atlas | Sprite 显示图片资源所属的 Atlas 图集资源 |
| Sprite Frame | 渲染 Sprite 使用的 SpriteFrame 图片资源 |
| Type | 渲染模式，包括普通（Simple）、九宫格（Sliced）、平铺（Tiled）和填充（Filled）渲染四种模式 |
| Size Mode | 指定 Sprite 的尺寸， `Trimmed` 会使用原始图片资源裁剪透明像素后的尺寸； `Raw` 会使用原始图片未经裁剪的尺寸；当用户手动修改过 `size` 属性后， `Size Mode` 会被自动设置为 `Custom`，除非再次指定为前两种尺寸。 |
| Trimmed Mode | 是否渲染原始图像周围的透明像素区域，详情请参考图像资源的自动剪裁。 |
| Src Blend Factor | 当前图像混合模式 |
| Dst Blend Factor | 背景图像混合模式，和上面的属性共同作用，可以将前景和背景 Sprite 用不同的方式混合渲染，效果预览可以参考 glBlendFunc Tool |

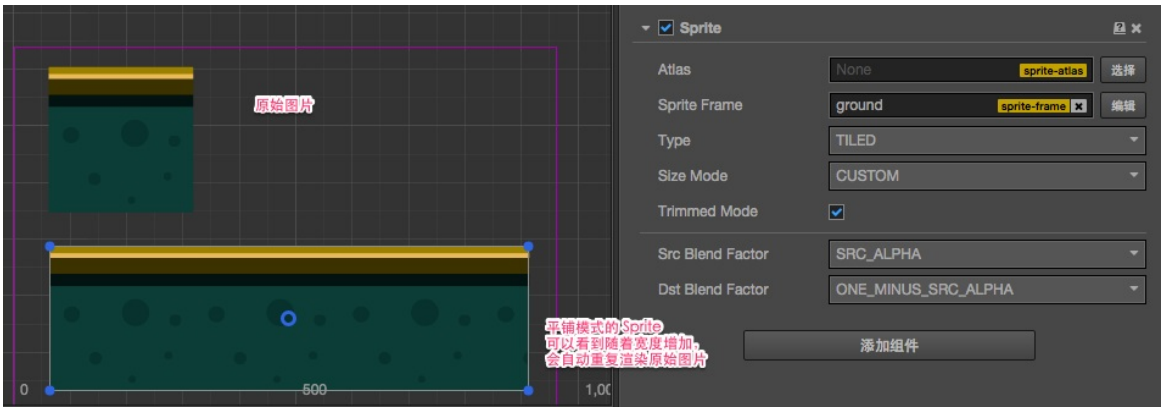
添加 Sprite 组件之后，通过从**资源管理器**中拖拽 Texture 或 SpriteFrame 类型的资源到 `Sprite Frame` 属性引用中，就可以通过 Sprite 组件显示资源图像。

如果拖拽的 SpriteFrame 资源是包含在一个 Atlas 图集资源中的，那么 Sprite 的 `Atlas` 属性也会被一起设置。之后可以点击 `Atlas` 属性旁边的**选择**按钮来从该 Atlas 中挑选另外一个 SpriteFrame 指定给 Sprite。

# 渲染模式

Sprite 组件支持四种渲染模式：

- 普通模式（Simple）：按照原始图片资源样子渲染 Sprite，一般在这个模式下我们不会手动修改节点的尺寸，来保证场景中显示的图像和美术人员生产的图片比例一致。
- 九宫格模式（Sliced）：图像将被分割成九宫格，并按照一定规则进行缩放以适应可随意设置的尺寸( `size` )。通常用于 UI 元素，或将可以无限放大而不影响图像质量的图片制作成九宫格图来节省游戏资源空间。详细信息请阅读使用 Sprite 编辑器制作九宫格图像一节。
- 平铺模式（Tiled）：当 Sprite 的尺寸增大时，图像不会被拉伸，而是会按照原始图片的大小不断重复，就像平铺瓦片一样将原始图片铺满整个 Sprite 规定的大小。




- 填充模式（Filled）：根据原点和填充模式的设置，按照一定的方向和比例绘制原始图片的一部分。经常用于进度条的动态展示。

## 填充模式（Filled）

`Type` 属性选择填充模式后，会出现一组新的属性可供配置，让我们依次介绍他们的作用。

| 属性 | 功能说明 |
| --- | --- |
| Fill Type | 填充类型选择，有 `HORIZONTAL` （横向填充）、`VERTICAL` （纵向填充）和 `RADIAL` （扇形填充）三种。 |
| Fill Start | 填充起始位置的标准化数值（从 0 ~ 1，表示填充总量的百分比），选择横向填充时， `Fill Start` 设为 0，就会从图像最左边开始填充 |
| Fill Range | 填充范围的标准化数值（同样从 0 ~ 1），设为 1，就会填充最多整个原始图像的范围。 |
| Fill Center | 填充中心点，只有选择了 `RADIAL` 类型才会出现这个属性。决定扇形填充时会环绕 Sprite 上的哪个点，坐标系和设置 Anchor 锚点 时是一样的。 |

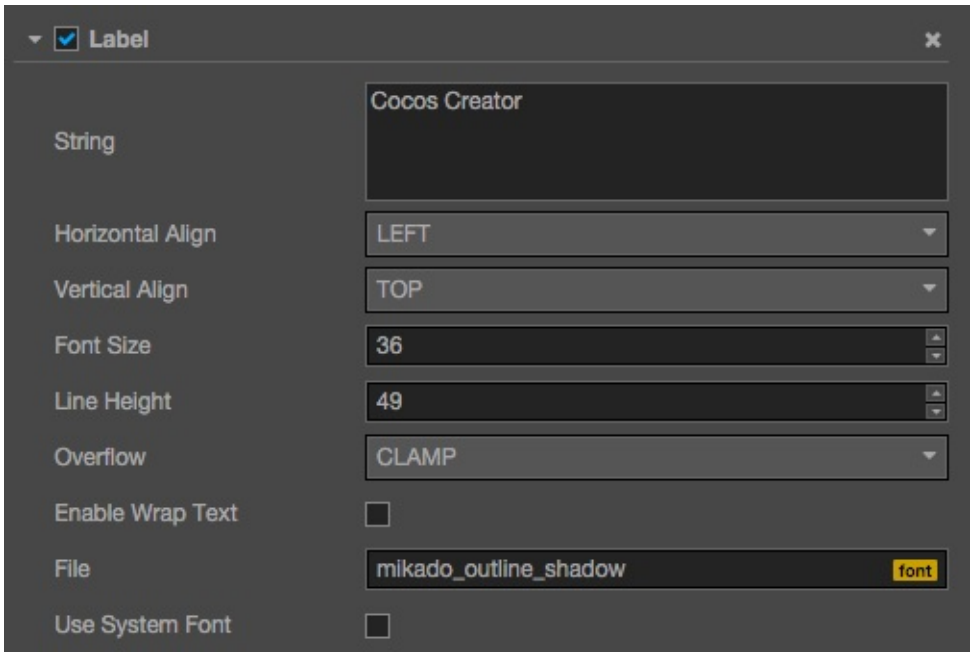
## Fill Range 填充范围补充说明

在 `HORIZONTAL` 和 `VERTICAL` 这两种填充类型下， `Fill Start` 设置的数值将影响填充总量，如果 `Fill Start` 设为 0.5，那么即使 `Fill Range` 设为 1.0，实际填充的范围也仍然只有 Sprite 总大小的一半。

而 `RADIAL` 类型中 `Fill Start` 只决定开始填充的方向， `Fill Start` 为 0 时，从 x 轴正方向开始填充， `Fill Range` 决定填充总量，值为 1 时将填充整个圆形。 `Fill Range` 为正值时逆时针填充，为负值时顺时针填充。

# Label 组件参考

Label 组件用来显示一段文字，文字可以是系统字体，TrueType 字体或者 BMFont 字体和艺术数字，另外，Label 还具有排版功能。


点击**属性检查器**下面的 添加组件 按钮，然后从 添加渲染组件 中选择 `Label`，即可添加 Label 组件到节点上。

文字的脚本接口请参考Label API。

# Label 属性

| 属性 | 功能说明 |
| --- | --- |
| String | 文本内容字符串。 |
| Horizontal Align | 文本的水平对齐方式。可选值有 LEFT，CENTER 和 RIGHT。 |
| Vertical Align | 文本的垂直对齐方式。可选值有 TOP，CENTER 和 BOTTOM。 |
| Font Size | 文本字体大小。 |
| Line Height | 文本的行高。 |
| Overflow | 文本的排版方式，目前支持 CLAMP，SHRINK 和 RESIZE_HEIGHT。详情见Label 排版。 |
| Enable Wrap Text | 是否开启文本换行。 |
| SpacingX | 文本字符之间的间距 （只有 BMFont 字体可以设置） |
| Font | 指定文本渲染需要的字体文件，如果使用系统字体，则此属性可以为空。 |
| Use System Font | 布尔值，是否使用系统字体。 |

# Label 排版

| 属性 | 功能说明 |
| --- | --- |
| CLAMP | 文字尺寸不会根据 Bounding Box 的大小进行缩放，Wrap Text 关闭的情况下，按照正常文字排列，超出 Bounding Box 的部分将不会显示。Wrap Text 开启的情况下，会试图将本行超出范围的文字换行到下一行。如果纵向空间也不够时，也会隐藏无法完整显示的文字。 |
| SHRINK | 文字尺寸会根据 Bounding Box 大小进行自动缩放（不会自动放大，最大显示 Font Size 规定的尺寸），Wrap Text 开启时，当宽度不足时会优先将文字换到下一行，如果换行后还无法完整显示，则会将文字进行自动适配 Bounding Box 的大小。如果 Wrap Text 关闭时，则直接按照当前文字进行排版，如果超出边界则会进行自动缩放。 |
| RESIZE_HEIGHT | 文本的 Bounding Box 会根据文字排版进行适配，这个状态下用户无法手动修改文本的高度，文本的高度由内部算法自动计算出来。 |

# 详细说明

Label 组件可以通过往 `属性检查器` 里的 `File` 属性拖拽 TTF 字体文件和 BMFont 字体文件来修改渲染的字体类型。如果不想继续使用字体文件，可以通过勾选 `Use System Font` 来重新启用系统字体。

使用艺术数字字体需要创建艺术数字资源，参考链接中的文档设置好艺术数字资源的属性之后，就可以像使用 BMFont 资源一样来使用艺术数字了。

## BMFont 与 UI 合图自动批处理

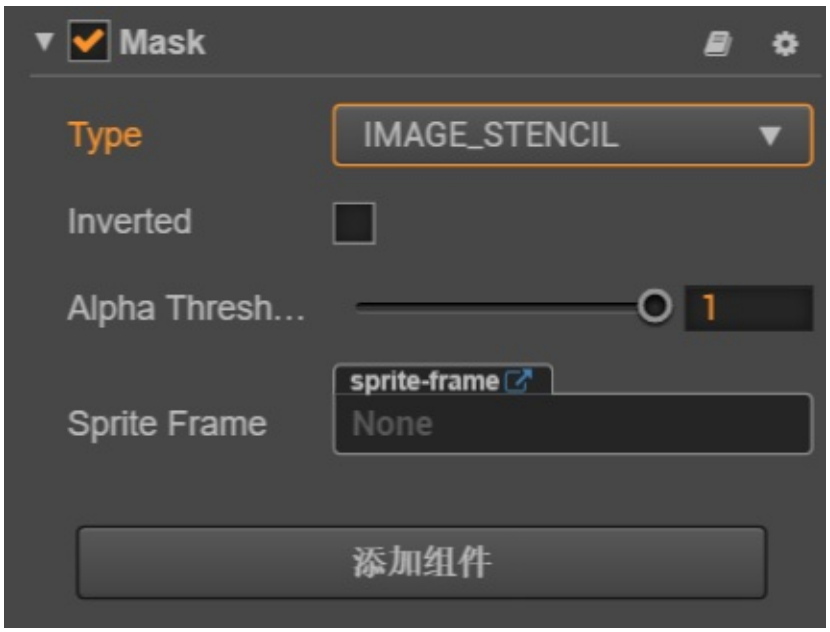
从 Creator 1.4 版本开始， BMFont 支持与 UI 一起合图进行批量渲染。 理论上，如果你的游戏 UI 没有使用系统字体或者 TTF 字体，并且所有的 UI 图片资源都可以合在一张图上，那么 UI 是可以只用一个 Draw Call 来完成的。 更多关于 BMFont 与 UI 合图自动批处理的内容，请参考 BMFont 与 UI 合图自动批处理

# Mask（遮罩）组件参考

Mask 用于规定子节点可渲染的范围，带有 Mask 的组件的节点会使用该节点的约束框（也就是 `size` 规定的范围）创建一个渲染遮罩，该节点的所有子节点都会依据这个遮罩进行裁剪，遮罩范围外的将不会渲染。


点击 **属性检查器** 下面的 `添加组件` 按钮，然后从 `添加渲染组件` 中选择 `Mask`，即可添加 Mask 组件到节点上。注意该组件不能添加到有其他渲染组件的节点上，如 `Sprite`、`Label` 等。

遮罩的脚本接口请参考Mask API。




## Mask 属性

| 属性 | 功能说明 |
| --- | --- |
| Type | 遮罩类型 Type API |
| Inverted | 布尔值，反向遮罩 |
| AlphaThreshold | 浮点数，只有当模板的像素的 alpha 大于 alphaThreshold 时，才会绘制内容，该数值 0 ~ 1 之间的浮点数，1 表示因此禁用 alpha 只在遮罩 IMAGE_STENCIL 类型可用 |
| SpriteFrame | 遮罩所需要的贴图，只在遮罩 IMAGE_STENCIL 类型可用 |
| Segements | 椭圆遮罩的曲线细分数，只在遮罩 ELLIPSE 类型可用 |

## 详细说明

给节点添加 Mask 组件之后，所有在该节点下的子节点在渲染的时候都会受 Mask 影响。

# MotionStreak（拖尾）组件参考

MotionStreak（拖尾）是 运动条纹，用于游戏对象的运动轨迹上实现拖尾渐隐效果。


点击 **属性检查器** 下面的 `添加组件` 按钮，然后从 `添加其他组件` 中选择 `MotionStreak`，即可添加 MotionStreak 组件到节点上。

拖尾的脚本接口请参考MotionStreak API。

## MotionStreak 属性

| 属性 | 功能说明 |
| --- | --- |
| fadeTime | 拖尾的渐隐时间,以秒为单位。 |
| minSeg | 拖尾之间最小距离。 |
| stroke | 拖尾的宽度。 |
| texture | 拖尾的贴图。 |
| fastMode | 是否启用了快速模式。当启用快速模式，新的点会被更快地添加，但精度较低。 |

# Particle System 组件参考

## 概述

该组件是用来读取 粒子资源 数据，并且对其进行一系列例如播放，暂时，销毁等操作。。

## 创建方式

编辑器创建：

点击**属性检查器**下面的 `添加组件` 按钮，然后从 `添加渲染组件` 中选择 `ParticleSystem`，即可添加 Particle System 组件到节点上。

脚本创建：

```
// 创建一个节点
var node = new cc.Node();
// 并将节点添加到场景中
c.director.getScene().addChild(node);
// 并添加粒子组件到 Node 上
var particleSystem = node.adComponent(cc.ParticleSystem);
// 接下去就可以对 particleSystem 这个对象进行一系列操作了
```

Particle System 的脚本接口请参考 Particle System API。

## Particle System 属性

| 属性 | 功能说明 |
| --- | --- |
| Preview | 布尔值，在编辑器模式下预览粒子，启用后选中粒子时，粒子将自动播放 |
| PlayOnLoad | 布尔值，如果设置为 true 运行时会自动发射粒子 |
| AutoRemoveOnFinish | 布尔值，粒子播放完毕后自动销毁所在的节点 |
| File | Plist 格式的粒子配置文件 |
| Custom | 布尔值，是否自定义粒子属性 |
| Texture | 粒子贴图 |
| Duration | 发射器生存时间，单位秒，-1 表示持续发射 |
| EmissionRate | 每秒发射的粒子数目 |
| Life | 粒子的运行时间 |
| LifeVar | 粒子的运行时间变化范围 |
| ParticleCount | 当前播放的粒子数量 |
| StartColor | 粒子初始颜色 |
| StartColorVar | 粒子初始颜色变化范围 |
| EndColor | 粒子结束颜色 |
| EndColorVar | 粒子结束颜色变化范围 |

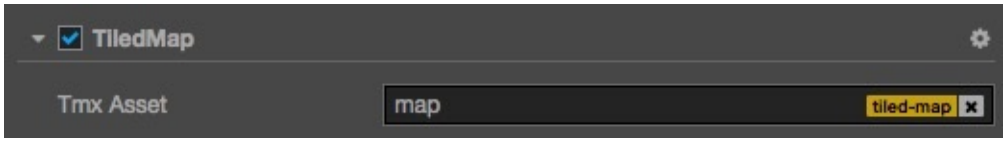
| | |
|---|---|
| Angle | 粒子角度 |
| AngleVar | 粒子角度变化范围 |
| StartSize | 粒子的初始大小 |
| StartSizeVar | 粒子的初始大小变化范围 |
| EndSize | 粒子结束大小 |
| EndSizeVar | 粒子结束大小的变化范围 |
| StartSpin | 粒子开始自旋角度 |
| StartSpinVar | 粒子开始自旋角度变化范围 |
| EndSpin | 粒子结束自旋角度 |
| EndSpinVar | 粒子结束自旋角度变化范围 |
| SourcePos | 发射器位置 |
| PosVar | 发射器位置的变化范围。（横向和纵向） |
| PositionType | 粒子位置类型 PositionType API |
| EmitterMode | 发射器类型 EmitterMode API |
| Gravity | 重力 |
| Speed | 速度 |
| SpeedVar | 速度变化范围 |
| TangentialAccel | 每个粒子的切向加速度，即垂直于重力方向的加速度，只有在重力模式下可用 |
| TangentialAccelVar | 每个粒子的切向加速度变化范围 |
| RadialAccel | 粒子径向加速度，即平行于重力方向的加速度，只有在重力模式下可用 |
| RadialAccelVar | 粒子径向加速度变化范围 |
| RotationIsDir | 每个粒子的旋转是否等于其方向，只有在重力模式下可用 |
| StartRadius | 初始半径，表示粒子出生时相对发射器的距离，只有在半径模式下可用 |
| StartRadiusVar | 初始半径变化范围 |
| EndRadius | 结束半径，只有在半径模式下可用 |
| EndRadiusVar | 结束半径变化范围 |
| RotatePerS | 粒子每秒围绕起始点的旋转角度，只有在半径模式下可用 |
| RotatePerSVar | 粒子每秒围绕起始点的旋转角度变化范围 |
| rotatePerSVar | 每个粒子的旋转是否等于其方向，只有在重力模式下可用 |
| rotatePerSVar | 每个粒子的旋转是否等于其方向，只有在重力模式下可用 |
| SrcBlendFactor | 指定原图混合模式 BlendFactor API |
| DstBlendFactor | 指定目标的混合模式 BlendFactor API |

# 注意事项

目前由于在 Canvas 中进行的每个粒子纹理 Color 渲染时很耗性能，所以建议在 Canvas 渲染模式下粒子数量不要太多尽量保持在 200 个以内，否则会导致运行时非常卡

# TiledMap 组件参考

TiledMap（地图）用于在游戏中显示 TMX 格式的地图。在节点上添加 TiledMap 组件，然后给该组件设置 tmxAsset 属性，就可以在场景中显示相应的地图了。




点击**属性检查器**下面的 `添加组件` 按钮，然后从 `添加渲染组件` 中选择 `TiledMap`，即可添加 TiledMap 组件到节点上。

瓦片图的脚本接口请参考TiledMap API。

# TiledMap 属性

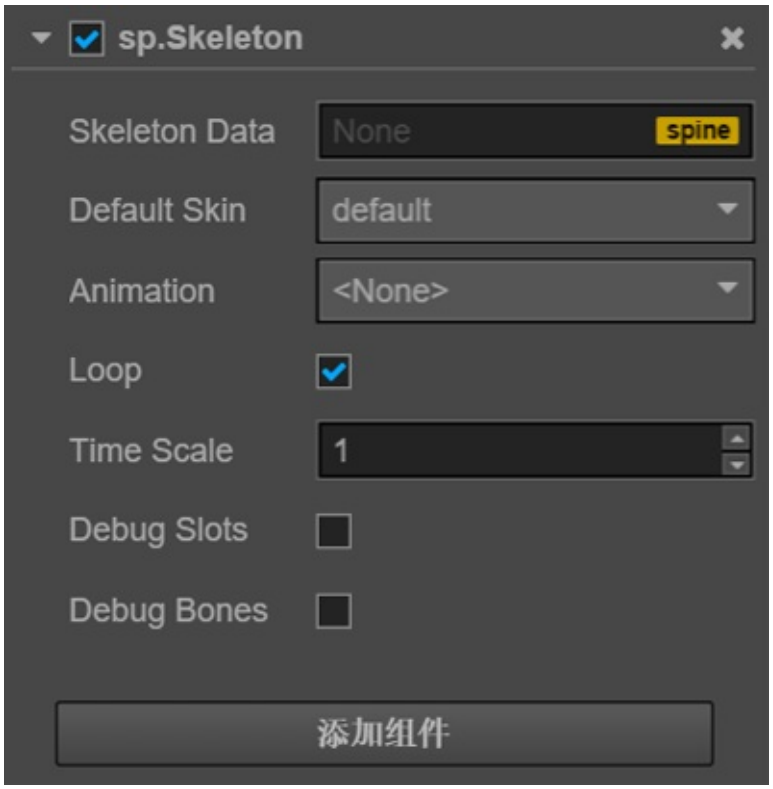
| 属性 | 功能说明 |
| --- | --- |
| tmxAsset | 指定 tmx 格式的地图资源。 |

# 详细说明

- 添加 TiledMap 组件之后，通过从**资源管理器**中拖拽一个 TiledMap 格式的资源到 tmxAsset 属性上就可以在场景中看到地图的显示了。
- TiledMap 组件会在节点中添加与地图中的 Layer 对应的节点。这些节点都添加了 TiledLayer 组件。**请勿删除这些 Layer 节点中的 TiledLayer 组件**。
- 在之前版本的 TiledMap 组件中，只能在 `mapLoaded` 的回调中使用 TiledMap 组件。而新版本的 TiledMap 组件加载机制进行了调整，不再支持 `mapLoaded` 回调，在 `start` 函数中就可以正常使用 TiledMap 组件了。

# Spine 组件参考

Spine 组件对骨骼动画（Spine）资源，进行渲染和播放。




点击**属性检查器**下面的添加组件按钮，然后从添加其他组件中选择 Spine Skeleton，即可添加 Spine 组件到节点上。

Spine 的脚本接口请参考Skeleton API。

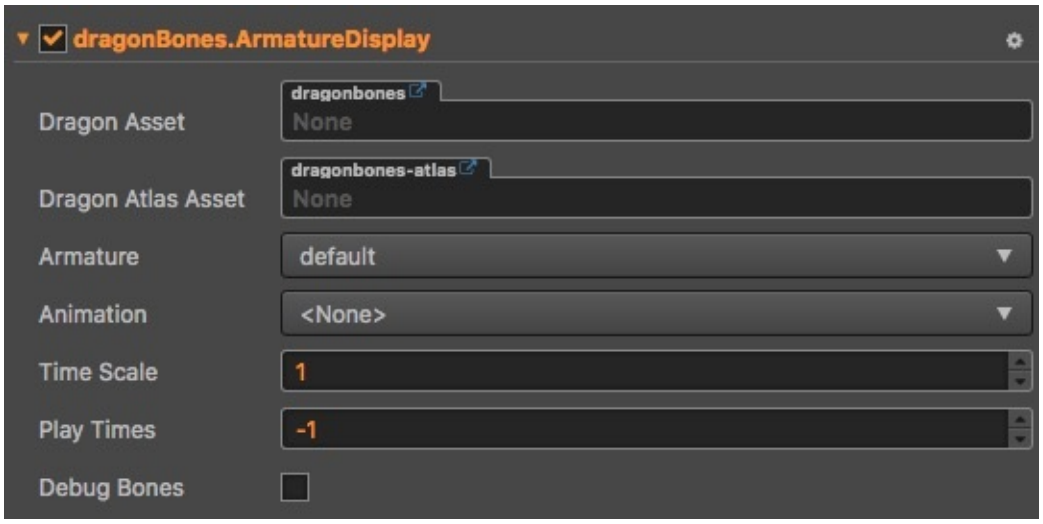
# Spine 属性

| 属性 | 功能说明 |
| --- | --- |
| Skeleton Data | Spine 导出后的 `.json` 文件 |
| Default Skin | 默认的皮肤 |
| Animation | 当前播放的动画 |
| Loop | 动画是否循环 |
| Premultiplied Alpha | 图片是否使用预乘，默认为 True。<br>当图片的透明区域出现色块时需要关闭该选项，<br>当图片的半透明区域颜色变黑时需要启用该选项。 |
| Time Scale | 播放速度 |
| Debug Slots | 显示图片边框 |
| Debug Bones | 显示骨骼 |

> 注意：当使用 Spine 组件时，Node 节点上 `Anchor` 与 `Size` 是无效的。

# DragonBones 组件参考

DragonBones 组件对骨骼动画（DragonBones）资源，进行渲染和播放。




点击**属性检查器**下面的添加组件按钮，然后从添加渲染组件中选择 DragonBones，即可添加 DragonBones 组件到节点上。

DragonBones 组件在脚本中的操作请参考 example cases 中的 DragonBones 测试例。

# DragonBones 属性

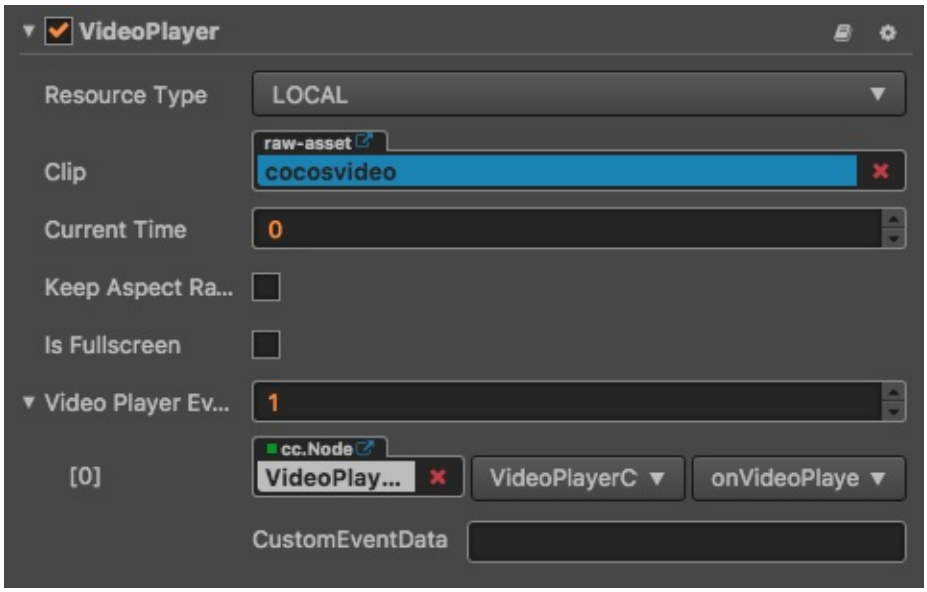
| 属性 | 功能说明 |
| --- | --- |
| Dragon Asset | DragonBones 导出后的骨骼数据（.json）文件 |
| Dragon Atlas Asset | DragonBones 导出后的图集数据（.json）文件 |
| Armature | 当前使用的 Armature 名称 |
| Animation | 当前播放的动画名称 |
| Time Scale | 播放速度 |
| Play Times | 动画播放次数（-1 表示使用配置文件中的默认值；0 表示无限循环；>0 表示循环次数） |
| Debug Bones | 显示骨骼 |

> 注意：当使用 DragonBones 组件时，Node 节点上 `Anchor` 与 `Size` 是无效的。

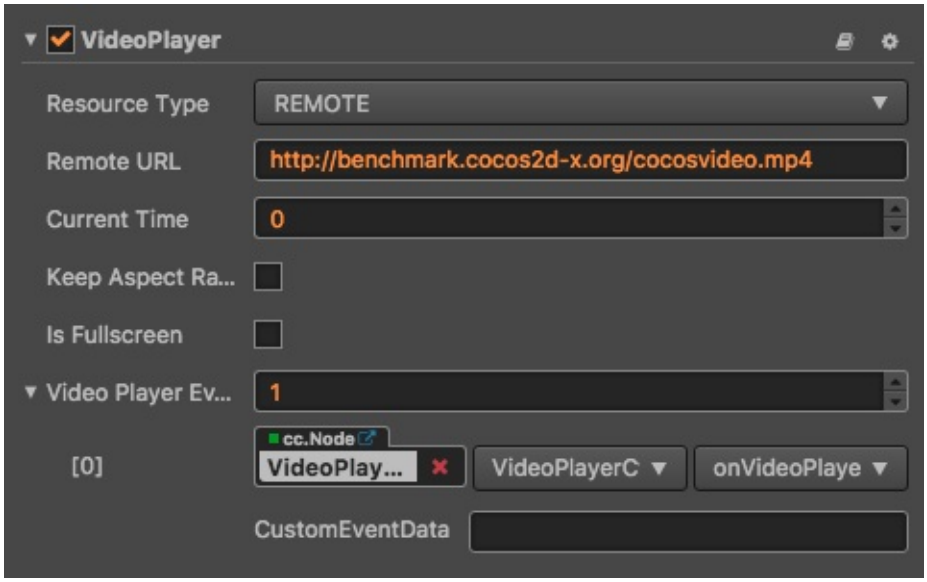
# VideoPlayer 组件参考

VideoPlayer 是一种视频播放组件，该组件让你可以轻松地播放本地和远程视频。

播放本地视频：




播放远程视频：




点击**属性检查器**下面的 `添加组件` 按钮，然后从 `添加 UI 组件` 中选择 `VideoPlayer`，即可添加 VideoPlayer 组件到节点上。

VideoPlayer 的脚本接口请参考 VideoPlayer API。

# VideoPlayer 属性

| 属性 | 功能说明 |
| --- | --- |
| Resource Type | 视频来源的类型，目前支持本地（LOCAL）视频和远程（REMOTE）视频 |

| | |
|---|---|
| Clip | 当 Resource Type 为 LOCAL 时显示的字段，指代一个本地视频的路径 |
| Remote URL | 当 Resource Type 为 REMOTE 时显示的字段，指代一个远程视频的路径 |
| Current Time | 指定视频播放时的当前时间点 |
| Keep Aspect Ratio | 是否保持视频原来的宽高比 |
| Is Fullscreen | 是否全屏播放视频 |
| Video Player Event | 视频播放回调函数，该回调函数会在特定情况被触发，比如播放中，暂时，停止和完成播放。详情见 `VideoPlayer 事件` 章节 |

# VideoPlayer 事件

## VideoPlayerEvent 事件

| 属性 | 功能说明 |
|---|---|
| Target | 带有脚本组件的节点。 |
| Component | 脚本组件名称。 |
| Handler | 指定一个回调函数，当视频开始播放后，暂停时或者结束时都会调用该函数，该函数会传一个事件类型参数进来。详情见 `Video Player Event 类型` 章节 |
| CustomEventData | 用户指定任意的字符串作为事件回调的最后一个参数传入。 |

## 事件回调参数

| 名称 | 功能说明 |
|---|---|
| PLAYING | 表示视频正在播放中。 |
| PAUSED | 表示视频暂停播放。 |
| STOPPED | 表示视频已经停止播放。 |
| COMPLETED | 表示视频播放完成。 |
| META_LOADED | 表示视频的元信息已加载完成，你可以调用 getDuration 来获取视频总时长。 |
| CLICKED | 表示视频被用户点击了。 |
| READY_TO_PLAY | 表示视频准备好了，可以开始播放了。 |

注意：在 iOS 平台上面，全屏模式下面点击视频无法发送 CLICKED 事件，如果需要让 iOS 全屏播放并正确接受 CLICKED 事件， 可以使用 Widget 组件把视频控件撑满，详情请参考引擎自带的 Example-cases 测试例。

# 详细说明

目前此组件只支持Web（PC 和手机）、iOS 和 Android 平台，Mac 和 Windows 平台暂时还不支持，如果在场景中使用此组件， 那么在 PC 的模拟器里面预览的时候可能看不到效果。

此控件支持的视频格式由所运行系统的视频播放器决定，为了让所有支持的平台都能正确播放视频，推荐使用 mp4 格式的视频。

## 通过脚本代码添加回调

## 方法一

这种方法添加的事件回调和使用编辑器添加的事件回调是一样的，通过代码添加， 你需要首先构造一个 `cc.Component.EventHandler` 对象，然后设置好对应的 target, component, handler 和 customEventData 参数。

```
var videoPlayerEventHandler = new cc.Component.EventHandler();
videoPlayerEventHandler.target = this.node; //这个 node 节点是你的事件处理代码组件所属的节点
videoPlayerEventHandler.component = "cc.MyComponent"
videoPlayerEventHandler.handler = "callback";
videoPlayerEventHandler.customEventData = "foobar";

videoPlayer.videoPlayerEvent.push(videoPlayerEventHandler);

//here is your component file
cc.Class({
    name: 'cc.MyComponent'
    extends: cc.Component,

    properties: {
    },

    //注意参数的顺序和类型是固定的
    callback: function(videoplayer, eventType, customEventData) {
        //这里 videoplayer 是一个 VideoPlayer 组件对象实例
        // 这里的 eventType === cc.VideoPlayer.EventType enum 里面的值
        //这里的 customEventData 参数就等于你之前设置的 "foobar"
    }
});
```

## 方法二

通过 `videoplayer.node.on('ready-to-play', ...)` 的方式来添加

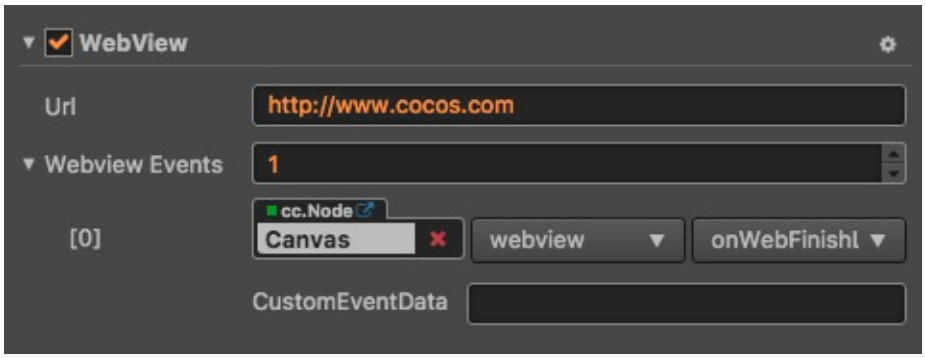
```
//假设我们在一个组件的 onLoad 方法里面添加事件处理回调，在 callback 函数中进行事件处理：

cc.Class({
    extends: cc.Component,


    properties: {
       videoplayer: cc.VideoPlayer
    },

    onLoad: function () {
       this.videoplayer.node.on('ready-to-play', this.callback, this);
    },

    callback: function (event) {
       //这里的 event 是一个 EventCustom 对象，你可以通过 event.detail 获取 VideoPlayer 组件
       var videoplayer = event.detail;
       //do whatever you want with videoplayer
       //另外，注意这种方式注册的事件，也无法传递 customEventData
    }
});
```

同样的，你也可以注册 'meta-loaded', 'clicked' , 'playing' 等事件，这些事件的回调函数的参数与 'read-to-play' 的参数一致。

关于完整的 VideoPlayer 的事件列表，可以参考 VideoPlayer 的 API 文档。

# WebView 组件参考

WebView 是一种显示网页的组件，该组件让你可以在游戏里面集成一个小的浏览器。




点击**属性检查器**下面的 `添加组件` 按钮，然后从 `添加 UI 组件` 中选择 `WebView`，即可添加 WebView 组件到节点上。

WebView 的脚本接口请参考 WebView API。

# WebView 属性

| 属性 | 功能说明 |
| --- | --- |
| Url | 指定一个 URL 地址，这个地址以 HTTP 或者 HTTPS 开头，请填写一个有效的 URL 地址。 |
| WebViewEvents | WebView 的回调事件，当 webview 在加载网页过程中，加载网页结束后或者加载网页出错时会调用此函数。 |

# WebView 事件

## WebViewEvents 事件

| 属性 | 功能说明 |
| --- | --- |
| Target | 带有脚本组件的节点。 |
| Component | 脚本组件名称。 |
| Handler | 指定一个回调函数，当网页加载过程中、加载完成后或者加载出错时会被调用，该函数会传一个事件类型参数进来。详情见 `WebView 事件回调参数` 章节 |
| CustomEventData | 用户指定任意的字符串作为事件回调的最后一个参数传入。 |

## WebView 事件回调参数

| 名称 | 功能说明 |
| --- | --- |
| LOADING | 表示网页正在加载过程中。 |
| LOADED | 表示网页加载已经完毕。 |
| ERROR | 表示网页加载出错了。 |

# 详细说明

目前此组件只支持Web（PC 和手机）、iOS 和 Android 平台，Mac 和 Windows 平台暂时还不支持，如果在场景中使用此组件， 那么在 PC 的模拟器里面预览的时候可能看不到效果。

此控件暂时不支持加载指定 HTML 文件或者执行 Javascript 脚本。

## 通过脚本代码添加回调

### 方法一

这种方法添加的事件回调和使用编辑器添加的事件回调是一样的，通过代码添加， 你需要首先构造一个 `cc.Component.EventHandler` 对象，然后设置好对应的 target, component, handler 和 customEventData 参数。

```
var webviewEventHandler = new cc.Component.EventHandler();
webviewEventHandler.target = this.node; //这个 node 节点是你的事件处理代码组件所属的节点
webviewEventHandler.component = "cc.MyComponent"
webviewEventHandler.handler = "callback";
webviewEventHandler.customEventData = "foobar";

webview.webviewEvents.push(webviewEventHandler);

//here is your component file
cc.Class({
    name: 'cc.MyComponent'
    extends: cc.Component,

    properties: {
    },

    //注意参数的顺序和类型是固定的
    callback: function(webview, eventType, customEventData) {
        //这里 webview 是一个 WebView 组件对象实例
        // 这里的 eventType === cc.WebView.EventType enum 里面的值
        //这里的 customEventData 参数就等于你之前设置的 "foobar"
    }
});
```

### 方法二

通过 `webview.node.on('loaded', ...)` 的方式来添加

```
//假设我们在一个组件的 onLoad 方法里面添加事件处理回调，在 callback 函数中进行事件处理：

cc.Class({
    extends: cc.Component,


    properties: {
       webview: cc.WebView
    },

    onLoad: function () {
       this.webview.node.on('loaded', this.callback, this);
    },

    callback: function (event) {
       //这里的 event 是一个 EventCustom 对象，你可以通过 event.detail 获取 WebView 组件
       var webview = event.detail;
       //do whatever you want with webview
       //另外，注意这种方式注册的事件，也无法传递 customEventData
    }
```

```
});
```

同样的，你也可以注册 'loading', 'error' 事件，这些事件的回调函数的参数与 'loaded' 的参数一致。

# Graphics 组件参考

Graphics 组件提供了一系列绘画接口，这些接口参考了 canvas 的绘画接口来进行实现。

## 路径

| 方法 | 功能说明 |
| --- | --- |
| moveTo (x, y) | 把路径移动到画布中的指定点，不创建线条 |
| lineTo (x, y) | 添加一个新点，然后在画布中创建从该点到最后指定点的线条 |
| bezierCurveTo (c1x, c1y, c2x, c2y, x, y) | 创建三次方贝塞尔曲线 |
| quadraticCurveTo (cx, cy, x, y) | 创建二次贝塞尔曲线 |
| arc (cx, cy, r, a0, a1, counterclockwise) | 创建弧/曲线（用于创建圆形或部分圆） |
| ellipse (cx, cy, rx, ry) | 创建椭圆 |
| circle (cx, cy, r) | 创建圆形 |
| rect (x, y, w, h) | 创建矩形 |
| close () | 创建从当前点回到起始点的路径 |
| stroke () | 绘制已定义的路径 |
| fill () | 填充当前绘图（路径） |
| clear () | 清楚所有路径 |

## 颜色，样式

| 属性 | 功能说明 |
| --- | --- |
| lineCap | 设置或返回线条的结束端点样式 |
| lineJoin | 设置或返回两条线相交时，所创建的拐角类型 |
| lineWidth | 设置或返回当前的线条宽度 |
| miterLimit | 设置或返回最大斜接长度 |
| strokeColor | 设置或返回笔触的颜色 |
| fillColor | 设置或返回填充绘画的颜色 |

更多关于 **Graphics** 的信息请前往 绘画

# UI 系统

本章将介绍 Cocos Creator 中强大而灵活的 UI （用户界面）系统，我们将通过组合不同 UI 组件，来生产能够适配多种分辨率屏幕的、通过数据动态生成和更新显示内容、支持多种排版布局方式的 UI 界面。


- 使用 Sliced Sprite 制作 UI 图像
- 多分辨率适配方案
- 对齐策略
- 文字排版
- 常用 UI 控件
- 自动布局容器
- 制作动态生成内容的列表

---

继续前往 制作可任意拉伸的 UI 图像 说明文档。

# 多分辨率适配方案

Cocos Creator 在设计之初就致力于解决一套资源适配多种分辨率屏幕的问题。简单概括来说，我们通过以下几个部分完成多分辨率适配解决方案：

- **Canvas（画布）**组件随时获得设备屏幕的实际分辨率并对场景中所有渲染元素进行适当的缩放。
- **Widget（对齐挂件）**放置在渲染元素上，能够根据需要将元素对齐父节点的不同参考位置。
- **Label（文字）**组件内置了提供各种动态文字排版模式的功能，当文字的约束框由于 Widget 对齐要求发生变化时，文字会根据需要呈现完美的排版效果。
- **Sliced Sprite（九宫格精灵图）**则提供了可任意指定尺寸的图像，同样可以满足各式各样的对齐要求，在任何屏幕分辨率上都显示高精度的图像。

接下来我们首先了解设计分辨率、屏幕分辨率的概念，才能理解 **Canvas（画布）**组件的缩放作用。

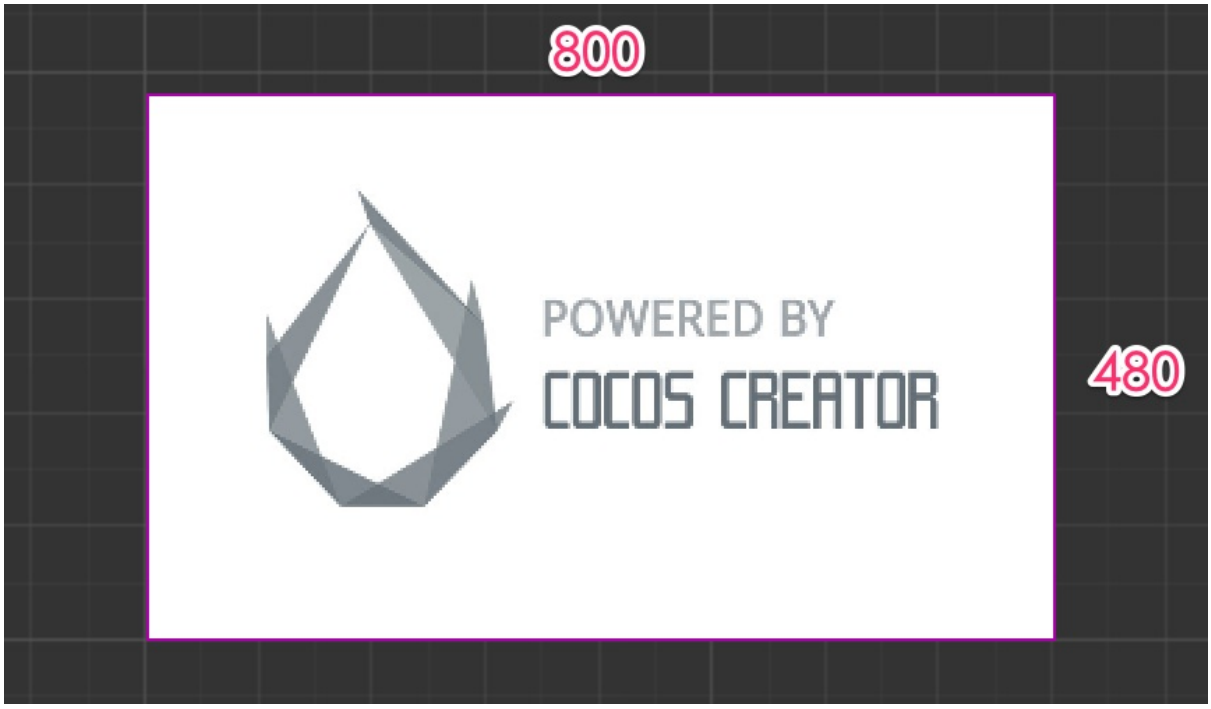
## 设计分辨率和屏幕分辨率

**设计分辨率** 是内容生产者在制作场景时使用的分辨率蓝本，而 **屏幕分辨率** 是游戏在设备上运行时的实际屏幕显示分辨率。

通常设计分辨率会采用市场目标群体中使用率最高的设备的屏幕分辨率，比如目前安卓设备中 `800x480` 和 `1280x720` 两种屏幕分辨率，或 iOS 设备中 `1136x640` 和 `960x640` 两种屏幕分辨率。这样当美术或策划使用设计分辨率设置好场景后，就可以自动适配最主要的目标人群设备。

那么当设计分辨率和屏幕分辨率出现差异时，Cocos Creator 会如何进行适配呢?

假设我们的设计分辨率为 `800x480`， 美术制作了一个同样分辨率大小的背景图像。

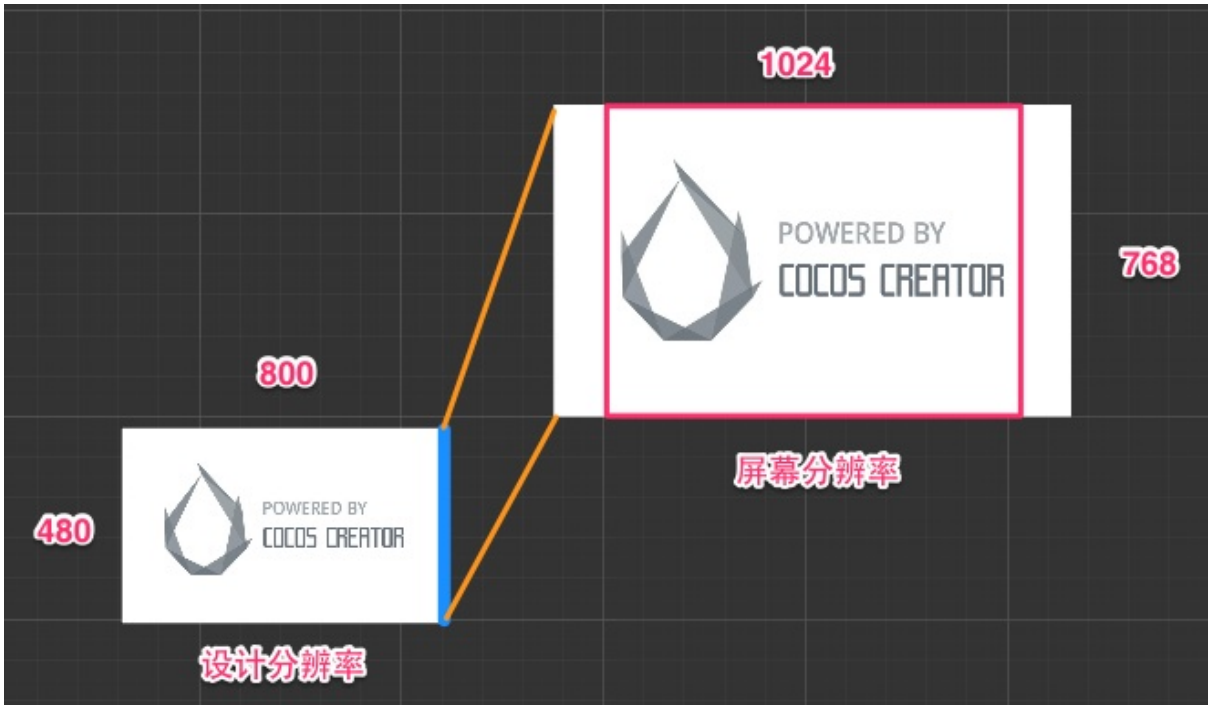



### 设计分辨率和屏幕分辨率宽高比相同

在屏幕分辨率的宽高比和设计分辨率相同时，假如屏幕分辨率是 `1600x960`，正好将背景图像放大 1600/800 = **2 倍** 就可以完美适配屏幕。这是最简单的情况，这里不再赘述。

## 设计分辨率宽高比大于屏幕分辨率，适配高度避免黑边

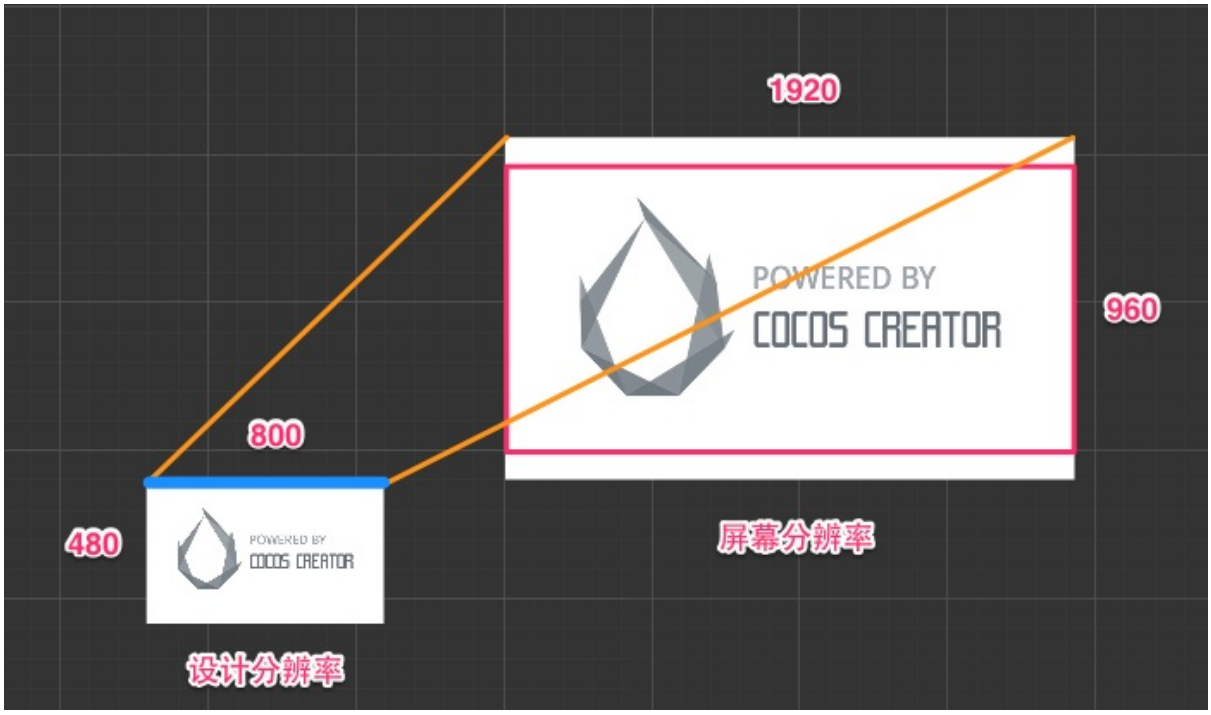
假设屏幕分辨率是 `1024x768`，在下图中以红色方框表示设备屏幕可见区域。我们使用 Canvas 组件提供的的 **适配高度**（`Fit Height`）模式，将设计分辨率的高度自动撑满屏幕高度，也就是将场景图像放大到 768/480 = **1.6 倍**。




这是设计分辨率宽高比大于屏幕分辨率时比较理想的适配模式，如上图所示，虽然屏幕两边会裁剪掉一部分背景图，但能够保证屏幕可见区域内不出现任何穿帮或黑边。之后可以通过 Widget（对齐挂件）调整 UI 元素的位置，来保证 UI 元素出现在屏幕可见区域里，我们在下一节 对齐策略 中将会详细介绍。

## 设计分辨率宽高比小于屏幕分辨率，适配宽度避免黑边

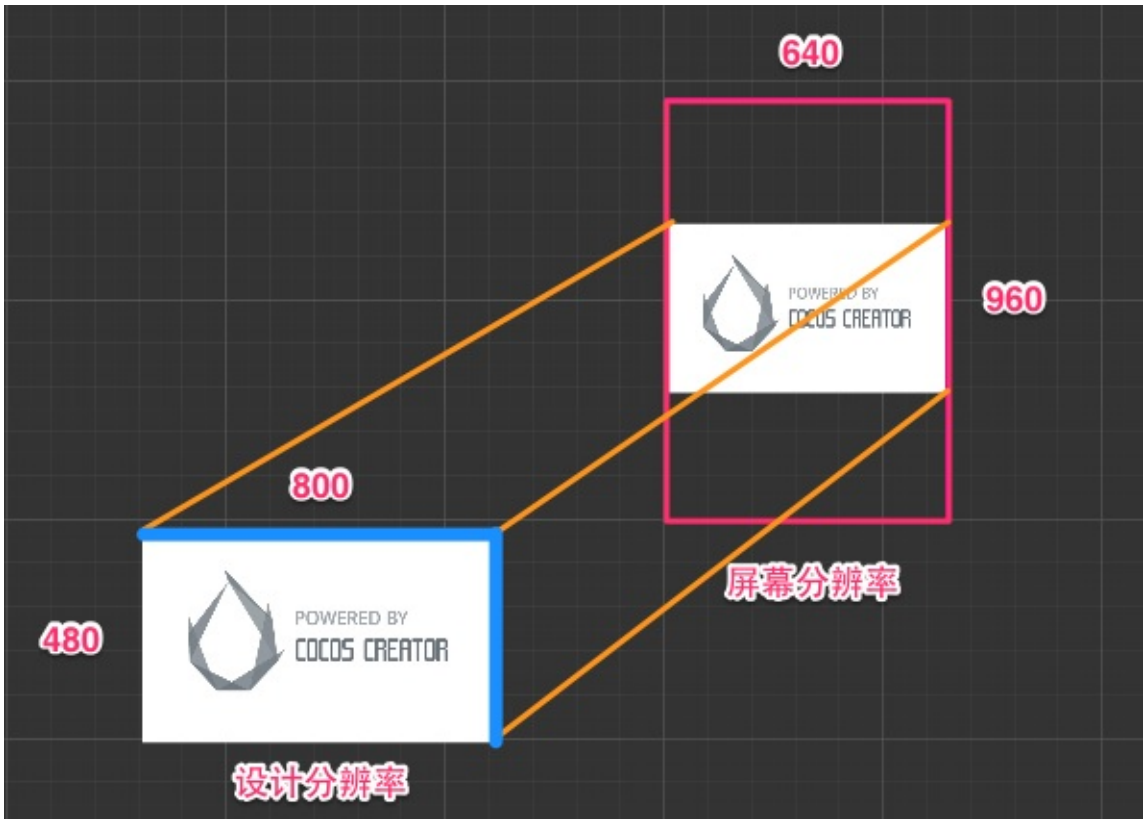
假设屏幕分辨率是 `1920x960`，同样在下图中以红色方框表示设备屏幕可见区域。我们使用 Canvas 组件提供的 **适配宽度**（`Fit Width`）模式，将设计分辨率的宽度自动撑满屏幕宽度，也就是将场景放大 1920/800 = **2.4 倍**。




在设计分辨率宽高比较小时，使用这种模式会裁剪掉屏幕上下一部分背景图。

## 不管屏幕宽高比如何，完整显示设计分辨率中的所有内容，允许出现黑边

最后一个例子，我们屏幕分辨率假设为 `640 x 960` 的竖屏，如果要确保背景图像完整的在屏幕中显示，需要同时开启 Canvas 组件中的 **适配高度** 和 **适配宽度**，这时场景图像的缩放比例是按照屏幕分辨率中较小的一维来计算的，在下图的例子中，由于屏幕宽高比小于 1，就会以宽度为准计算缩放倍率，即 640/800 = **0.8 倍**。



在这种显示模式下，屏幕上可能会出现黑边，或超出设计分辨率的场景图像（穿帮）。尽管一般情况下开发者会尽量避免黑边，但如果需要确保设计分辨率范围的所有内容都显示在屏幕上，也可以采用这种模式。

### 根据屏幕宽高比，自动选择适配宽度或适配高度

如果对于屏幕周围可能被剪裁的内容没有严格要求，也可以不开启 Canvas 组件中任何适配模式，这时会根据屏幕宽高比自动选择 **适配高度** 或 **适配宽度** 来避免黑边。也就是说，设计分辨率宽高比大于屏幕分辨率时，会自动适配高度（上面第一张图）；设计分辨率宽高比小于屏幕分辨率时，会自动适配宽度（上面第二张图）。

### Canvas 组件不提供分别缩放 x 和 y 轴缩放率，会使图像变形拉伸的适配模式

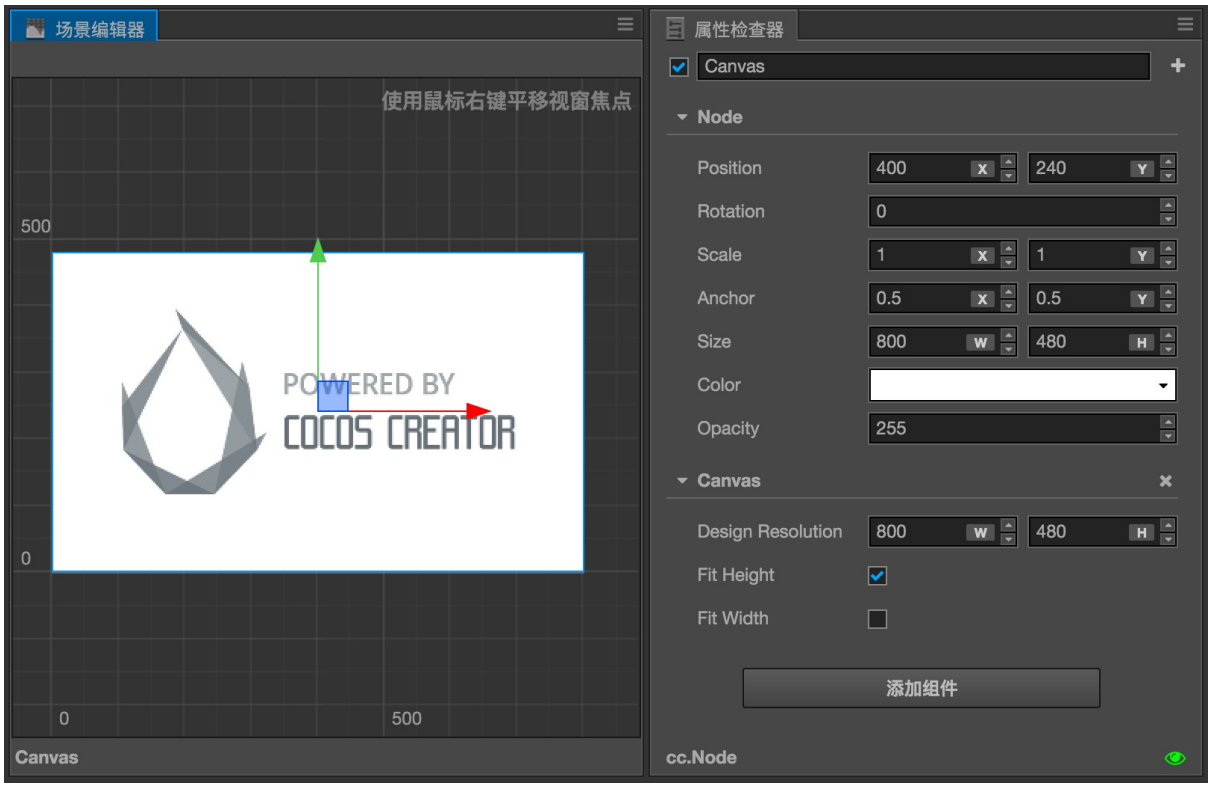
在 Cocos 引擎中，也存在称为 `ExactFit` 的适配模式，这种模式没有黑边，也不会裁剪设计分辨率范围内的图像。但是代价是场景图像的 x 和 y 方向的缩放倍率不同，图像会产生形变拉伸。

如果不介意图像形变，可以在不使用 Canvas 组件的情况下，使用引擎 API 来实现适配效果，详情可以参考 Cocos2d-JS 的屏幕适配方案。

## 在场景中使用 Canvas 组件

新建场景时，会自动在场景根节点上添加一个包含 Canvas 组件的节点。在 Canvas 组件上就可以设置上文中提到的选项：

- **设计分辨率**（`Design Resolution`）
- **适配高度**（`Fit Height`）
- **适配宽度**（`Fit Width`）




将 Canvas 节点作为所有图像渲染节点的根节点，这些节点就都可以自动享受 Canvas 智能适配多分辨率的缩放效果了。

### 编辑场景时 Canvas 的特性

在编辑场景时，Canvas 节点的尺寸（ `Size` ）属性会保持和 **设计分辨率** 一致，不能手动更改。

位置（ `Position` ）属性会保持在 `(width/2, height/2)` ，也就是和设计分辨率相同大小的屏幕中心。

由于锚点（ `Anchor` ）属性的默认值会设置为 `(0.5, 0.5)` ，Canvas 会保持在屏幕中心位置，并且 Canvas 的子节点会以屏幕中心作为坐标系原点，这一点和 Cocos 引擎中的习惯不同，请格外注意。

## 运行时 Canvas 的特性

除了上述特性外，运行时 Canvas 组件还会有以下属性变化：

- **缩放**（ `Scale` ）：根据前文中描述的缩放倍率计算原则，将计算后的缩放倍率赋值给 `Scale` 属性。
- **尺寸**（ `Size` ）：在无黑边的模式中，Canvas 的 `Size` 属性会和屏幕分辨率保持一致。在有黑边的模式中，Canvas 的 `Size` 会保持设计分辨率不变。

由于运行时 Canvas 可以准确获得屏幕可见区域的尺寸，我们就可以根据这个尺寸来设置 UI 元素的对齐策略，来保证 UI 元素都能在屏幕可见区域正确显示。

# 对齐策略

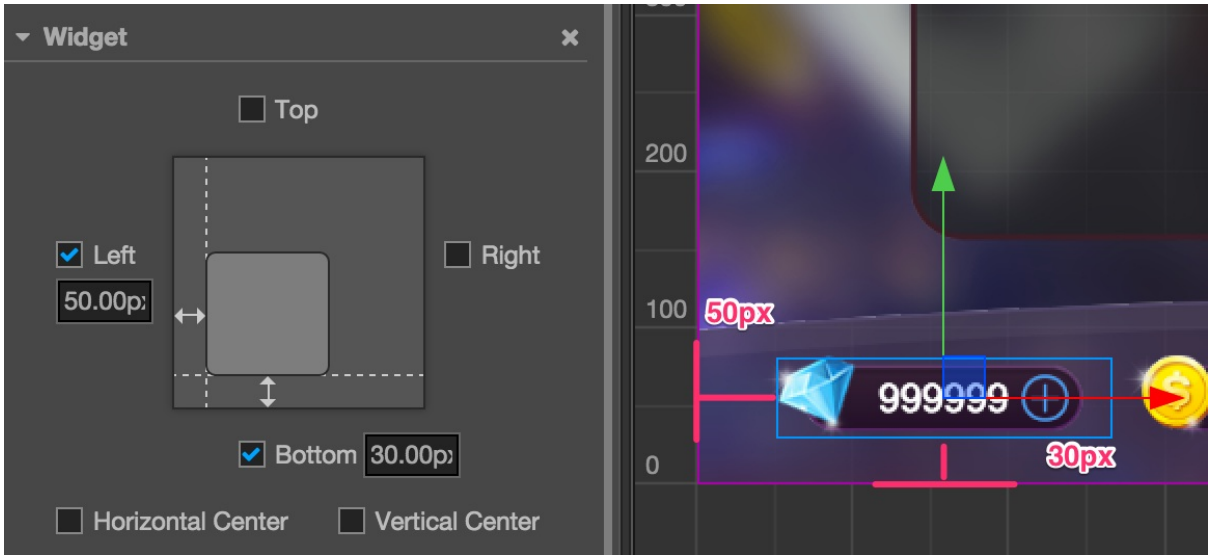
要实现完美的多分辨率适配效果，UI 元素按照设计分辨率中规定的位置呈现是不够的，当屏幕宽度和高度发生变化时，UI 元素要能够智能感知屏幕边界的位置，才能保证出现在屏幕可见范围内，并且分布在合适的位置。我们通过 **Widget（对齐挂件）** 来实现这种效果。

下面我们根据要对齐元素的类别来划分不同的对齐工作流：

## 需要贴边对齐的按钮和小元素

对于暂停菜单、游戏金币这一类面积较小的元素，通常只需要贴着屏幕边对齐就可以了。这时只要几个简单的步骤：

1. 把这些元素在 **层级管理器** 中设为 Canvas 节点的子节点
2. 在元素节点上添加 Widget 组件
3. 以对齐左下角为例，开启 `Left` 和 `Bottom` 的对齐。
4. 然后设置好节点和屏幕边缘的距离，下图中左边距设为 50px，右边距设为 30px。



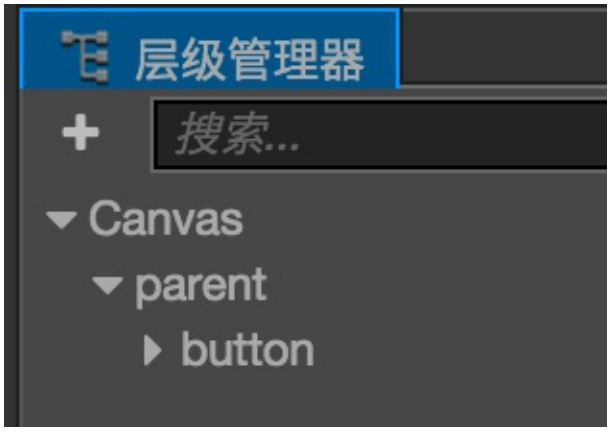

这样设置好 Widget 组件后，不管实际屏幕分辨率是多少，这个节点元素都会保持在屏幕左下角，而且节点约束框左边和屏幕左边距离保持 50px，节点约束框下边和屏幕下边距离保持 30px。

注意 Widget 组件提供的对齐距离是参照子节点和父节点相同方向的约束框边界的。比如上面例子里勾选了 `Left` 对齐左边，那么子节点约束框左边和父节点（也就是 Canvas 节点，约束框永远等于屏幕大小）约束框左边的距离就是我们设置的 50px。

## 嵌套对齐元素

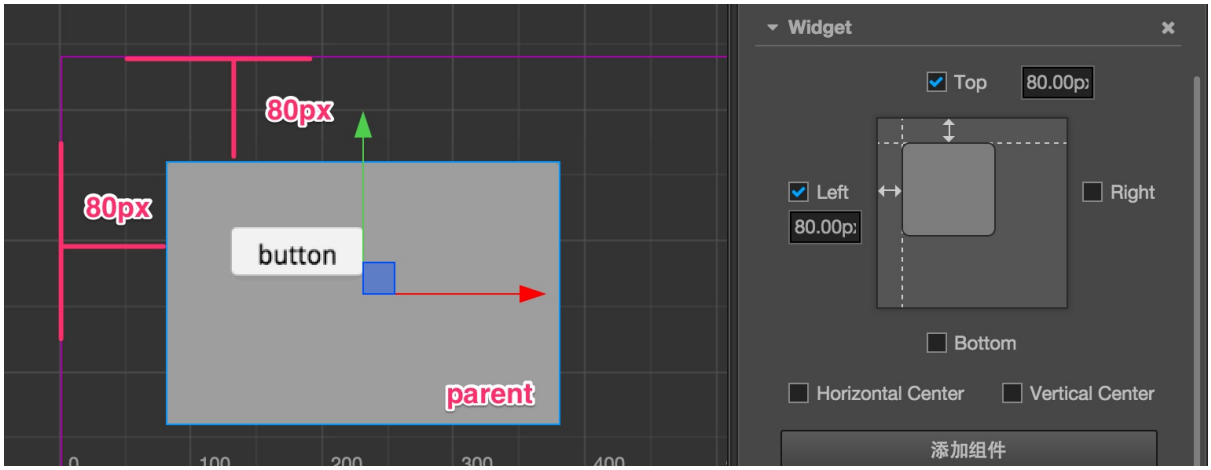
上面介绍了对齐屏幕边缘的做法，由于 Widget 默认的对齐参照物是父节点，所以我们也可以添加不同的节点层级，并且让每一级节点都使用自动对齐的功能。

我们下面用一个简单的例子来说明，假设我们有这样的节点层级关系：

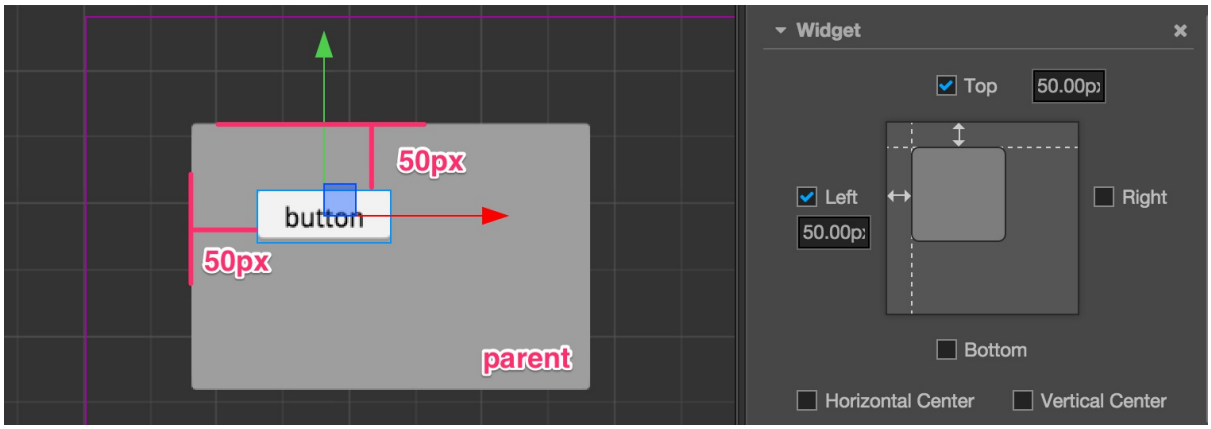



其中 `parent` 是一个面板， `button` 是一个按钮。我们可以分别为这两个节点添加 Widget 组件，并且分别设置对齐距离。

对于 `parent` 节点来说，对齐 `Canvas` 节点的左上角，距离都是 80px：



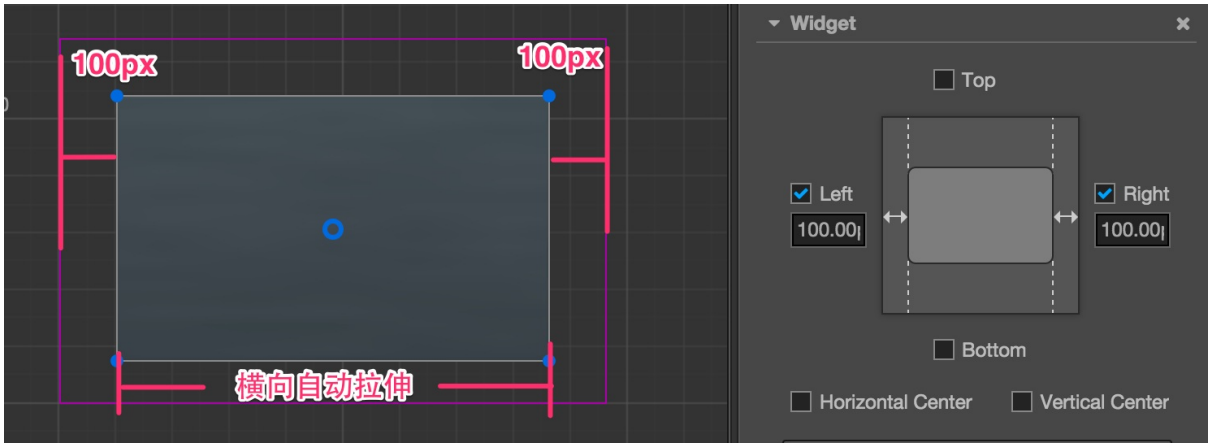

对于 `button` 节点来说，对齐 `parent` 节点的左上角，距离都是 50px：




依照这样的工作流程，就可以将 UI 元素按照显示区域或功能进行分组，并且不同级别的元素都可以按照设计进行对齐。

## 根据对齐需要自动缩放节点尺寸

以上我们展示的例子里，并没有同时对齐在同一轴向相反方向的两个边，如果我们要做一个占满整个屏幕宽度的面板，就可以同时勾选 `Left` 和 `Right` 对齐开关：

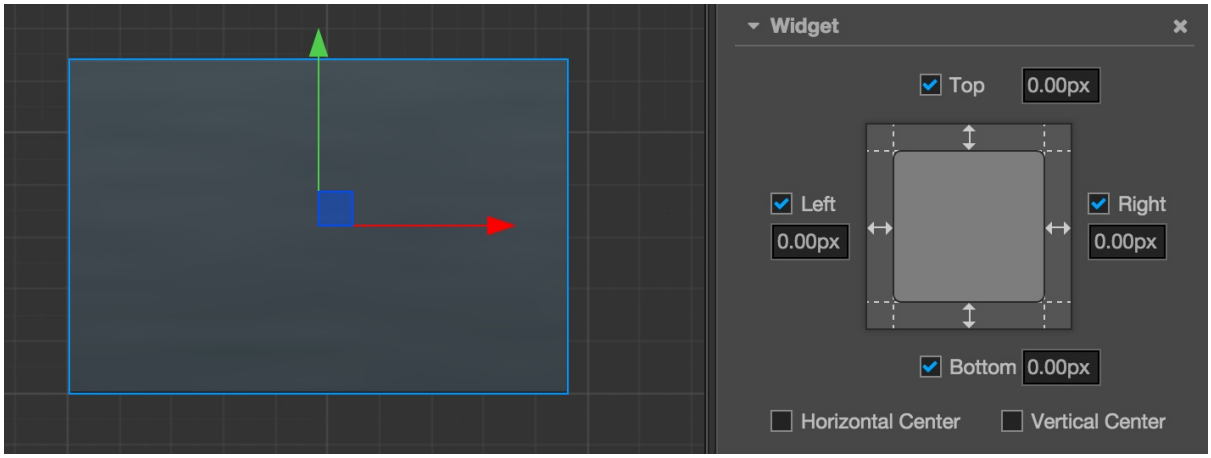



当同时勾选相反的两个方向的对齐开关时，Widget 就获得了根据对齐需要修改节点尺寸（`Size`）的能力，上图中我们勾选了左右两个方向并设置了边距，Widget 就会根据父节点的宽度来动态设置节点的 `Width` 属性，表现出来就是不管在多宽的屏幕上，我们的面板距离屏幕左右两边的距离永远保持 100px。

### 制作和屏幕大小保持一致的节点

利用自动缩放节点的特性，我们可以通过设置节点的 Widget 组件，使节点的尺寸和屏幕大小保持一致，这样我们就不需要把所有需要对齐屏幕边缘的 UI 元素都放在 Canvas 节点下，而是可以根据功能和逻辑的需要结组。

要制作这样的节点，首先要保证该节点的父节点尺寸能够保持和屏幕大小一致，Canvas 节点就是一个最好的选择。接下来按照下图的方式设置该节点的 Widget 组件：




就可以在运行时时刻保持该节点和 Canvas 节点的尺寸完全一致，也就是和屏幕大小一致。经过这样设置的节点，其子节点也可以使用同样的设置来传递屏幕实际尺寸。

注意，由于 Canvas 节点本身就有保持和屏幕大小一致的功能，因此不需要在 Canvas 上添加 Widget 组件。

## 设置百分比对齐距离

Widget 组件上开启某个方向的对齐之后，除了指定以像素为单位的边距以外，我们还可以输入百分比数值，这样 Widget 会以父节点相应轴向的宽度或高度乘以输入的百分比，计算出实际的边距值。

还是看看实际的例子，我们还是以一个直接放在 Canvas 下的子节点为例，我们希望这个节点面板保持在屏幕右侧，并且总是占据 60% 的屏幕总高度。那么按照下图所示设置 Widget 组件就可以实现这个效果：


Widget 在对齐方向开启时输入边距值时，可以按照需要混合像素单位和百分比单位的使用。比如在需要对齐屏幕中心线的 `Left` 方向输入 `50%`，在需要对齐屏幕边缘的 `Right` 方向输入 `20px`，最后计算子节点位置和尺寸时，所有的边距都会先根据父节点的尺寸换算成像素距离，然后再进行摆放。

利用百分比对齐距离，我们可以制作出根据屏幕大小无限缩放的 UI 元素，发挥你的想象力，一套资源适配数千种安卓机型不是问题！

# 运行时每帧更新对齐和优化策略

Widget 组件一般用于场景在目标设备上初始化时定位每个元素的位置，但一旦场景初始化完毕，很多时候我们就不需要 Widget 组件再进行对齐了。这里有个重要的属性 `alignOnce` 用于确保 Widget 组件只在初始化时执行对齐定位的逻辑，在运行时不再消耗时间来进行对齐。

`alignOnce` 如果被选中，在组件初始化时执行过一次对齐定位后，就会通过将 Widget 组件的 `enabled` 属性设为 `false` 来关闭之后每帧自动更新来避免重复定位。如果需要在运行时实时定位，你需要手动将 `alignOnce` 属性关闭（置为 `false`），或者在运行时需要进行每帧更新对齐时手动遍历需要对齐的 Widget 并将他们的 `enabled` 属性设为 `true`。

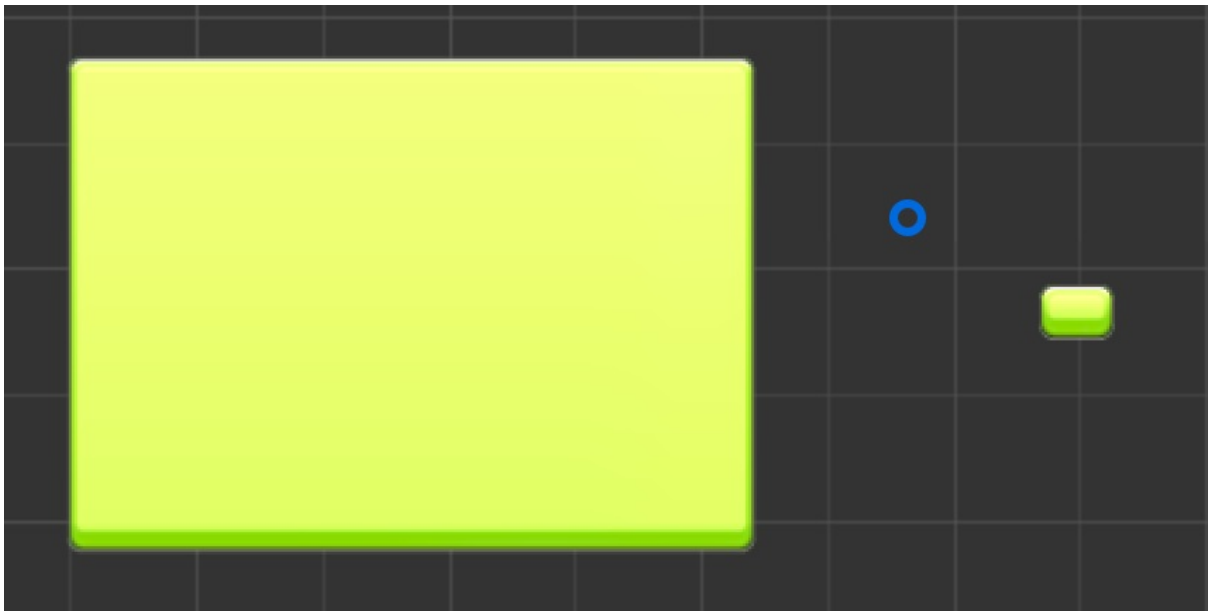
对于有很多 UI 元素的场景，确保 Widget 组件的 `alignOnce` 选项打开，可以大幅提高场景运行性能。

# 对齐组件对节点位置、尺寸的限制

通过 **Widget** 组件开启一个或多个对齐参考后，节点的位置（position）和尺寸（width，height）属性可能会被限制，不能通过 API 或动画系统自由修改。如果需要在运行时修改对齐节点的位置或尺寸，请参考Widget 组件参考：对节点位置、尺寸的限制相关内容。

# 制作可任意拉伸的 UI 图像

Cocos Creator 的 UI 系统核心的设计原则就是能够自动适应各种不同的设备屏幕尺寸，因此我们在制作 UI 时需要正确设置每个控件元素的尺寸（size），并且让每个控件元素的尺寸能够根据设备屏幕的尺寸进行自动的拉伸适配。为了实现这一点，就需要使用九宫格格式的图像来渲染这些元素。这样即使使用很小的原始图片也能生成覆盖整个屏幕的背景图像，一方面节约游戏包体空间，另一方面能够灵活适配不同的排版需要。



上图右边为原始贴图大小的显示，左边是选择 Sliced 模式并放大 `size` 属性后的显示效果。

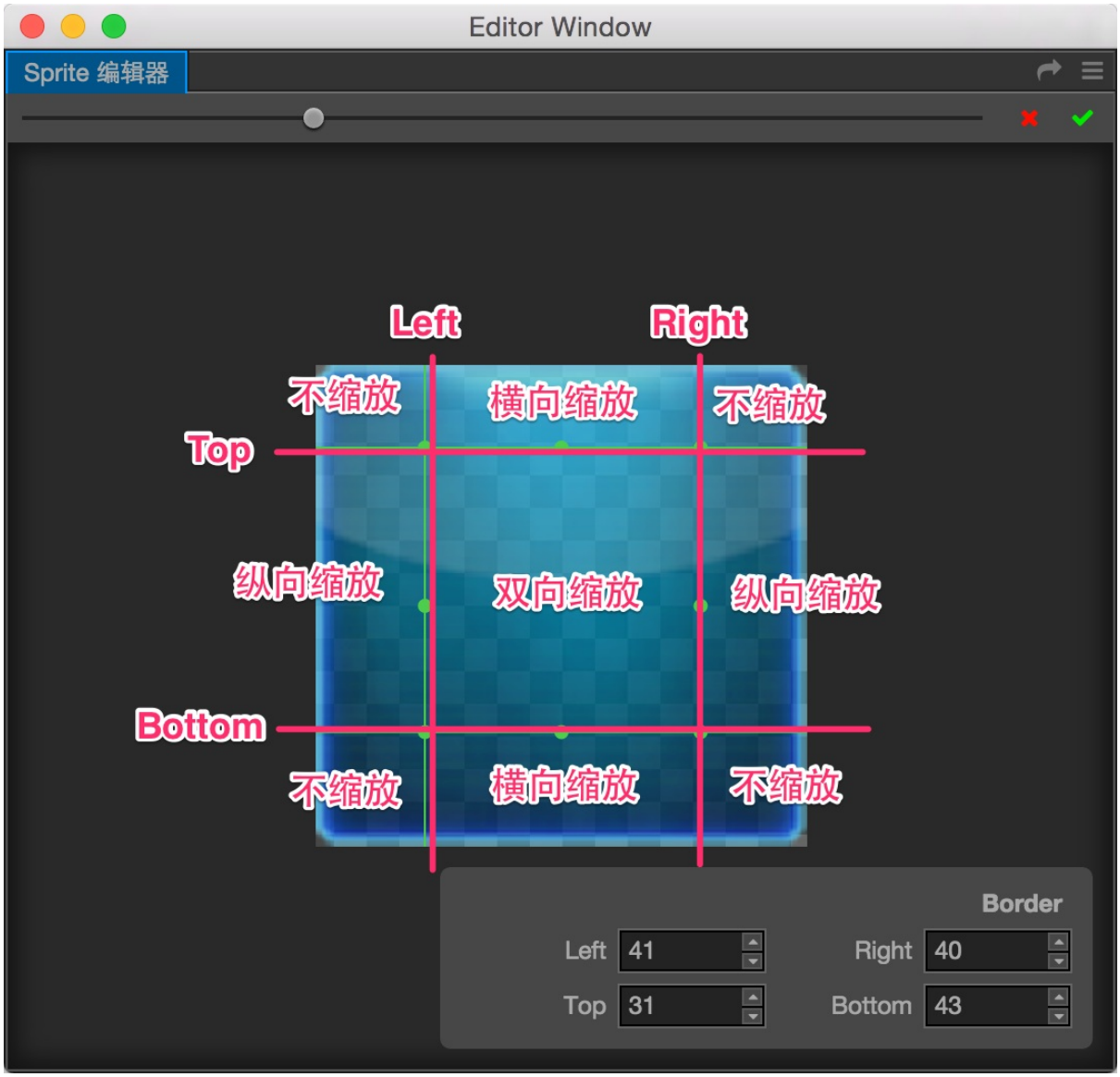
## 编辑图像资源的九宫格切分

要使用可以无限放大的九宫格图像效果，我们需要先对图像资源进行九宫格切分。有两种方式可以打开 **Sprite 编辑器** 来编辑图像资源：

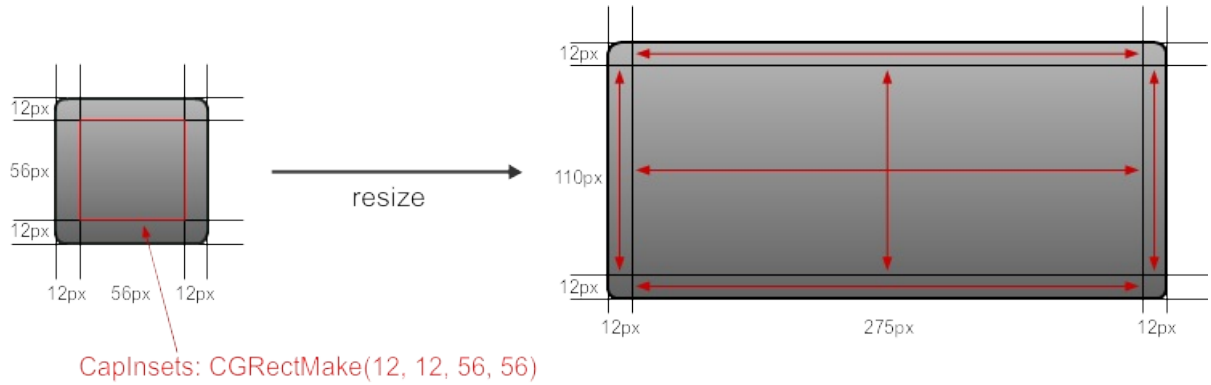
- 在**资源管理器**中选中图像资源，然后点击**属性检查器**最下面的**编辑**按钮。如果您的窗口高度不够，可能需要向下滚动**属性检查器**才能看到下面的按钮。
- 在**场景编辑器**中选中想要九宫格化的图像节点，然后在**属性检查器**的 Sprite 组件里，找到并按下 `Sprite Frame` 属性右侧的**编辑**按钮。

打开**Sprite 编辑器**以后，可以看到图像周围有一圈绿色的线条，表示当前九宫格分割线的位置。将鼠标移动到分割线上，可以看到光标形状改变了，这时候就可以按下并拖拽鼠标来更改分割线的位置。

我们分别拖动上下左右四条分割线，将图像切分成九宫格，九个区域在 Sprite 尺寸（ `size` ）变化时会应用不同的缩放策略，见下图：




而下图中描述了不同区域缩放时的示意（图片来自Yannick Loriot的博客）：




完成切分后别忘记点击**Sprite 编辑器**右上角的绿色对勾来保存对资源的修改。

## 设置 Sprite 组件使用 Sliced 模式

准备好九宫格切分的资源后，就可以修改 Sprite 的显示模式并通过修改 `size` 来制作可任意指定尺寸的 UI 元素了。

1. 首先选中场景中的 Sprite 节点，将 Sprite 的 `Type` 属性设为 `Sliced` 。
2. 然后通过矩形变换工具拖拽控制点使节点的 `size` 属性变大。您也可以直接在 **属性检查器** 中输入数值来修改 `size` 属性。如果图像资源是用九宫格的形式生产的，那么不管 Sprite 如何放大，都不会产生模糊或变形。

# 注意事项

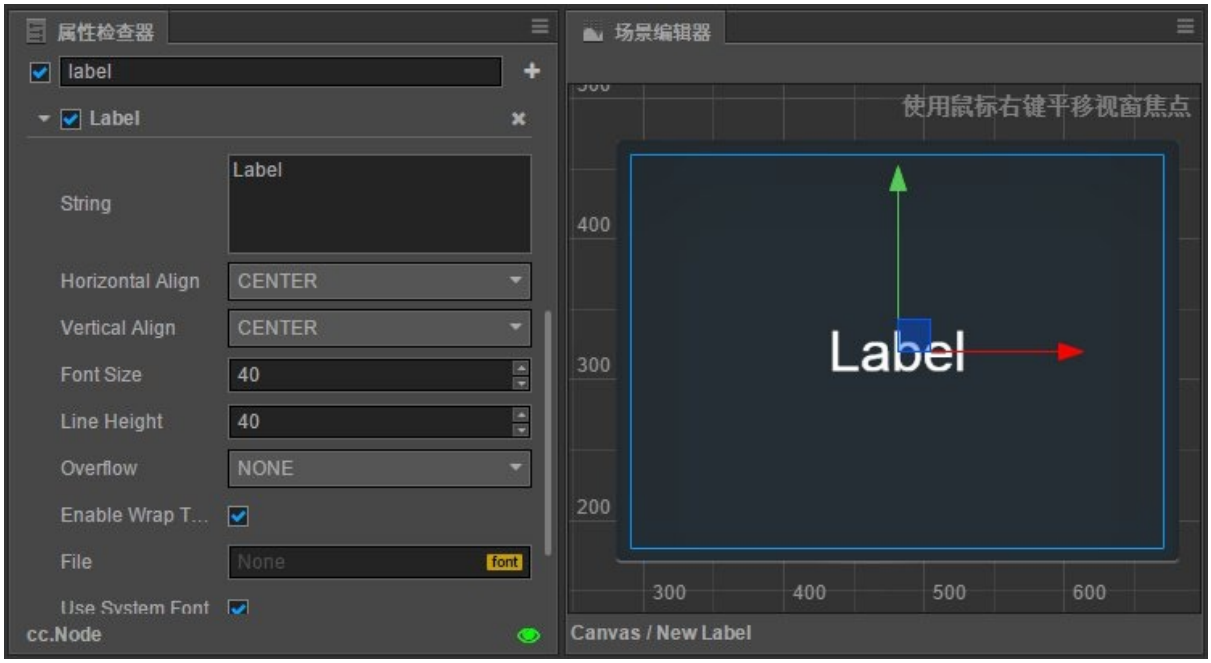
在使用**矩形变换工具**或直接修改 Sliced Sprite 的 `size` 属性时，注意 `size` 属性值不能为负数，否则不能以 Sliced 模式正常显示。

# 文字排版

**文字组件（Label）** 是核心渲染组件之一，您需要了解如何设置文字的排版，才能在 UI 系统进行多分辨率适配和对齐设置时显示完美的效果。

## 文字在约束框中对齐

和其他渲染组件一样，Label 组件的排版也是基于节点尺寸（`Size`），也就是约束框（Bounding Box）所规定的范围的。




上图所示就是 Label 渲染的文字在蓝色约束框内显示的效果。Label 中以下的属性决定了文字在约束框中显示的位置：

- `Horizontal Align`（水平对齐）：文字在约束框中水平方向的对齐准线，可以从 Left、Right、Center 三种位置中选择。
- `Vertical Align`（垂直对齐）：文字在约束框中垂直方向的对齐准线，可以从 Top、Bottom、Center 三种位置中选择。

上图中水平和垂直方向的对齐位置都设为了 `Center`，可以看到文字出现在约束框的正中心。您可以将以上两个属性修改为其他组合，文字会根据设置出现在约束框的四边或四个角的位置。

## 文字尺寸和行高

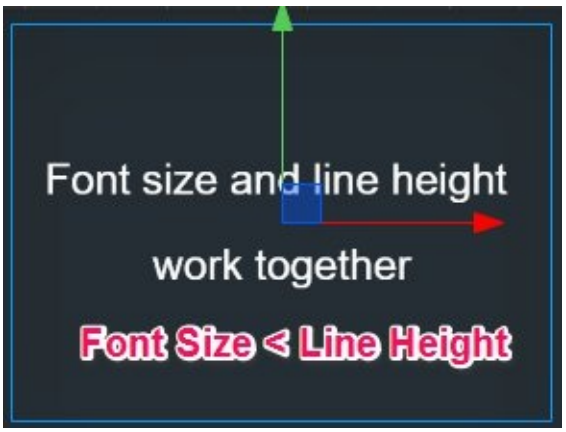
`Font Size`（文字尺寸）决定了文字的显示大小，单位是 Point （也称作“磅”），是大多数图像制作和文字处理软件中通用的字体大小单位。对于动态字体来说，`Font Size` 可以无损放大，但位图字体在将 `Font Size` 设置为超过字体标定的字号大小时，显示会变得越来越模糊。

`Line Height`（行高）决定了文字在多行显示时每行文字占据的空间高度，单位同样是 Point。多行文字显示可以通过两种方式实现：

- 在 `String` 属性中输入文字时，手动输入回车或换行符
- 开启 `Enable Wrap Text`（换行）属性，下文会详细介绍

文字尺寸和行高的关系：

- 如果 `Font Size` 和 `Line Height` 设为相同数值，文字正好占据一行大部分的空间高度。
- 如果 `Font Size` 小于 `Line Height`，多行文字之间间隔会加大
- 如果 `Font Size` 大于 `Line Height`，多行文字之间间隔会缩小，甚至出现文字相互重叠的情况。

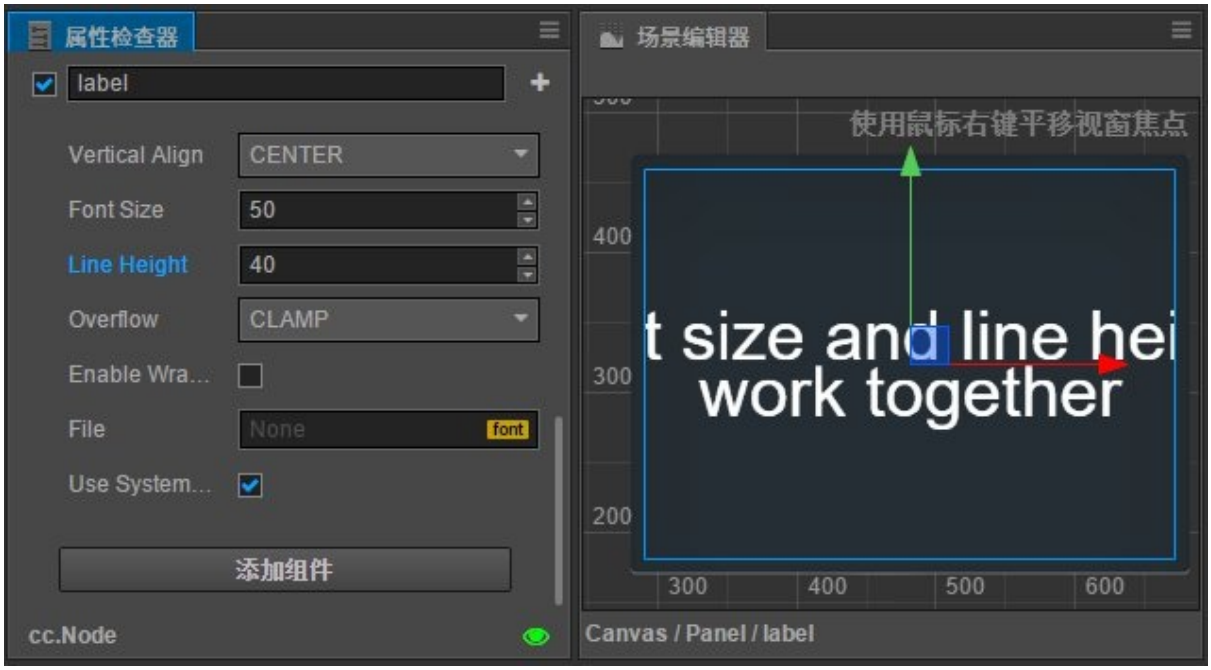





# 排版模式（Overflow）

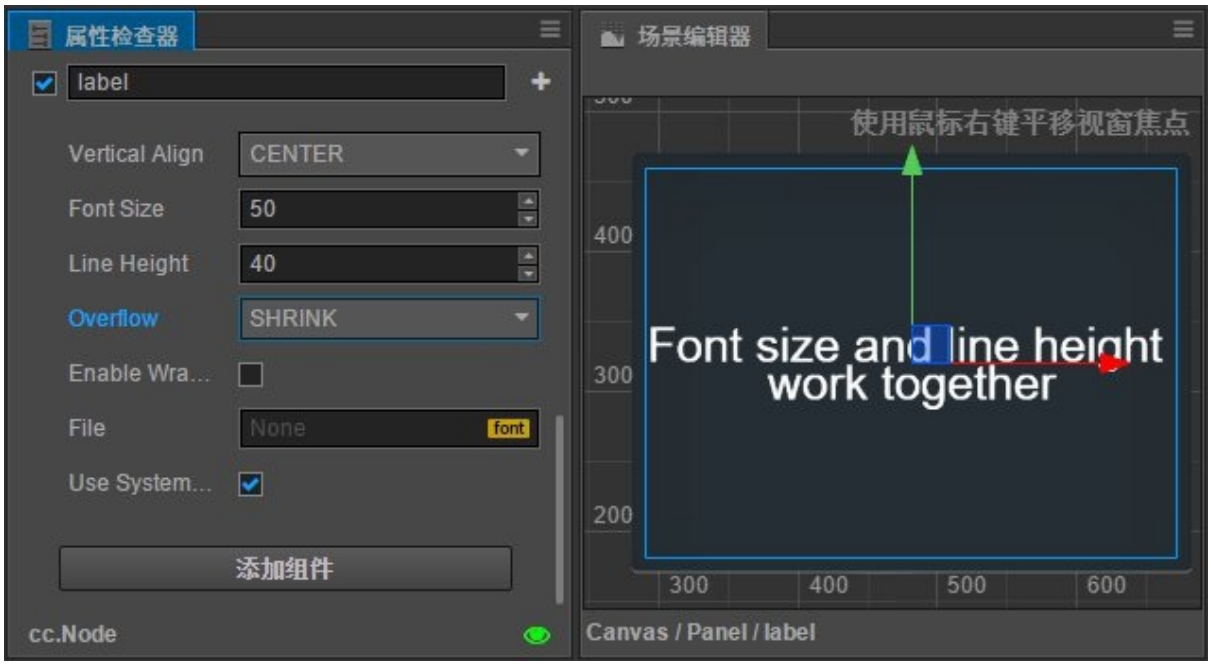
**Overflow（排版模式）** 属性，决定了文字内容增加时，如何在约束框的范围内排布。

## 截断（Clamp）




截断模式下，文字首先按照对齐模式和尺寸的要求进行渲染，而超出约束框的部分会被隐藏（截断）。

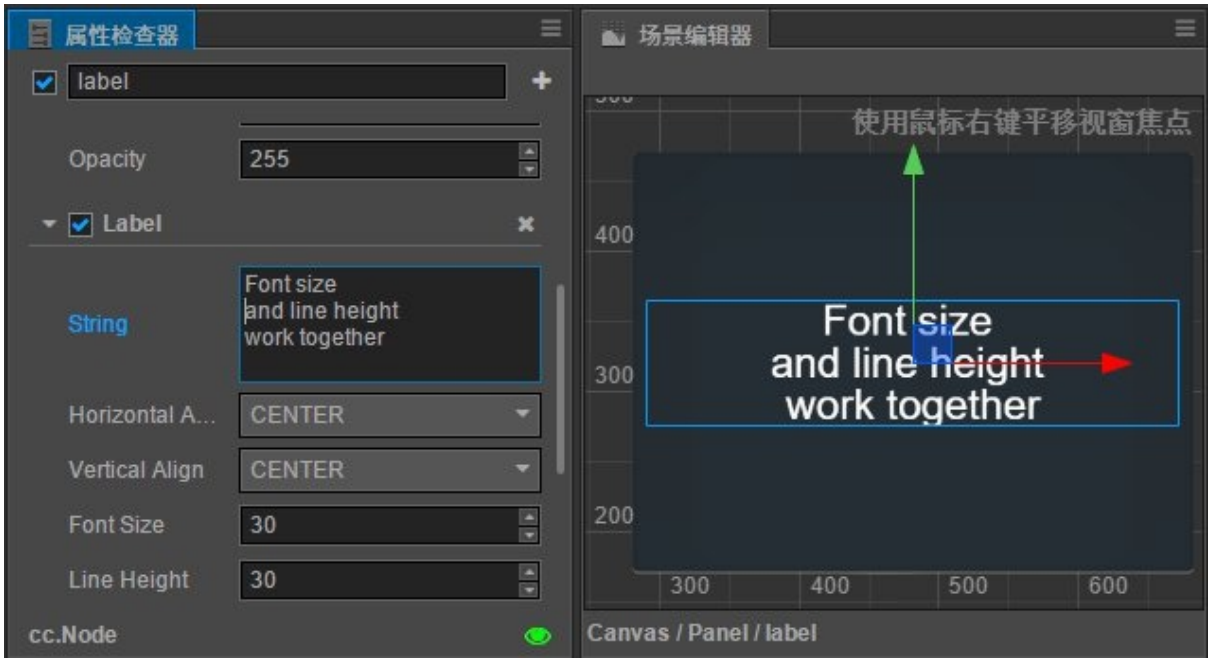
## 自动缩小（Shrink）




自动缩小模式下，如果文字按照原定尺寸渲染会超出约束框时，会自动缩小文字尺寸以显示全部文字。

注意，自动缩小模式不会放大文字来适应约束框。

## 自动适应高度 （Resize Height）




自动适应高度模式会保证文字的约束框贴合文字的高度，不管文字有多少行。这个模式非常适合显示内容量不固定的大段文字，配合 ScrollView 组件可以在任意 UI 区域中显示无限量的文字内容。

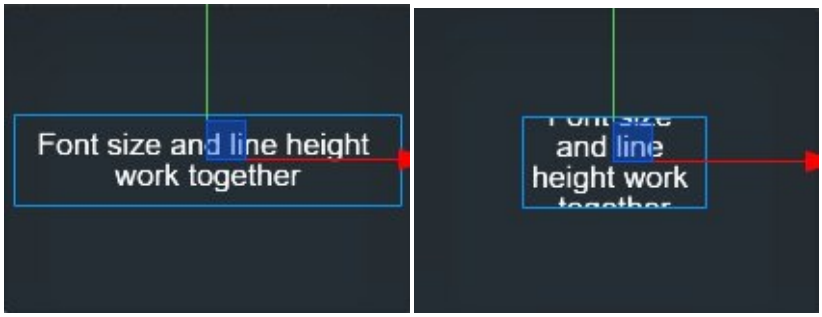
# 自动换行

Label 组件中的 `Enable Wrap Text` （自动换行）属性，可以切换文字的自动换行开关。在自动换行开启的状态下，不需要在输入文字时手动输入回车或换行符，文字也会根据约束框的宽度自动换行。

## 截断模式自动换行

截断模式开启自动换行后，会优先在约束框允许的范围内换行排列文字，如果换行之后仍无法显示全部文字时才发生截断。

以下两幅图都是在 `Clamp` + `Enable Wrap Text` 开启情况下的，区别在于文字约束框的宽度不同：




在约束框宽度从左图变化到右图的过程中，文字将不断调整换行，从 2 行逐渐变为 4 行，最后由于约束框高度不足而产生了截断显示。

## 自动缩小模式自动换行

和截断模式类似，自动缩小模式下文字超出约束框宽度时也会优先试图换行，在约束框宽度和长度都已经完全排满的情况下才会自动缩小文字以适应约束框。


### 自动适应高度

自动适应高度模式中，自动换行属性是强制开启的。

### 中文自动换行

中文自动换行的行为和英文不同，英文是以单词为单位进行换行的，必须有空格才能作为换行调整的最小单位。中文是以字为单位进行换行，每个字都可以单独调整换行。

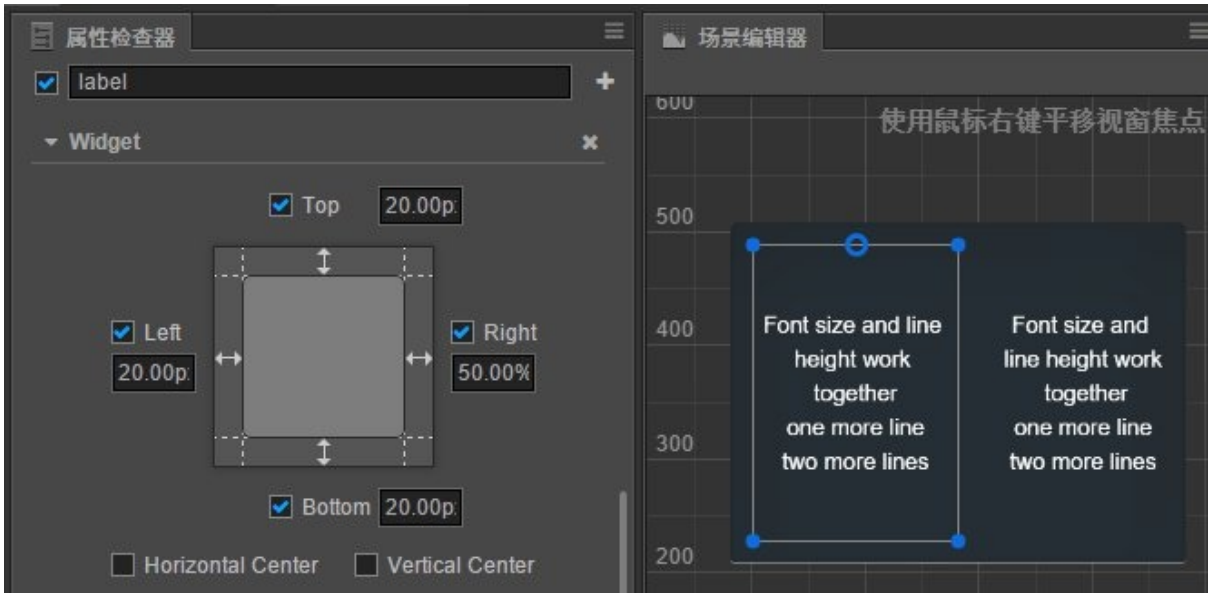
## 文字节点的锚点

文字节点的锚点和文字在约束框中的对齐模式是需要区分的两个概念。在需要靠文字内容将约束框撑大的排版模式中（如 `Resize Height`），要正确设置锚点位置，才能让约束框向我们期望的方向延伸。

例如，如果希望文字约束框向下延伸，需要将锚点（`Anchor`）的 `y` 属性设为 `1`。


## 文字配合对齐挂件（Widget）

在 Label 组件所在节点上添加一个 **Widget （对齐挂件）** 组件，就可以让文字节点相对于父节点进行各式各样的排版。




上图中我们在背景节点上添加了两个 Label 子节点，分别为他们添加 Widget 组件后，设置左边文字 Widget 的 `Right` 属性为 `50%`，右边文字 Widget 的 `Left` 属性为 `60%`，就可以实现图中所示的多列布局式文字。

而且通过 Widget 上设置边距，加上文字本身的排版模式，可以让我们在不需要具体微调文字约束框大小的情况下轻松实现灵活美观的文字排版。

# 查看组件参考

关于 Label 组件的属性，也可以查阅Label 组件参考文档。

# UI 动画

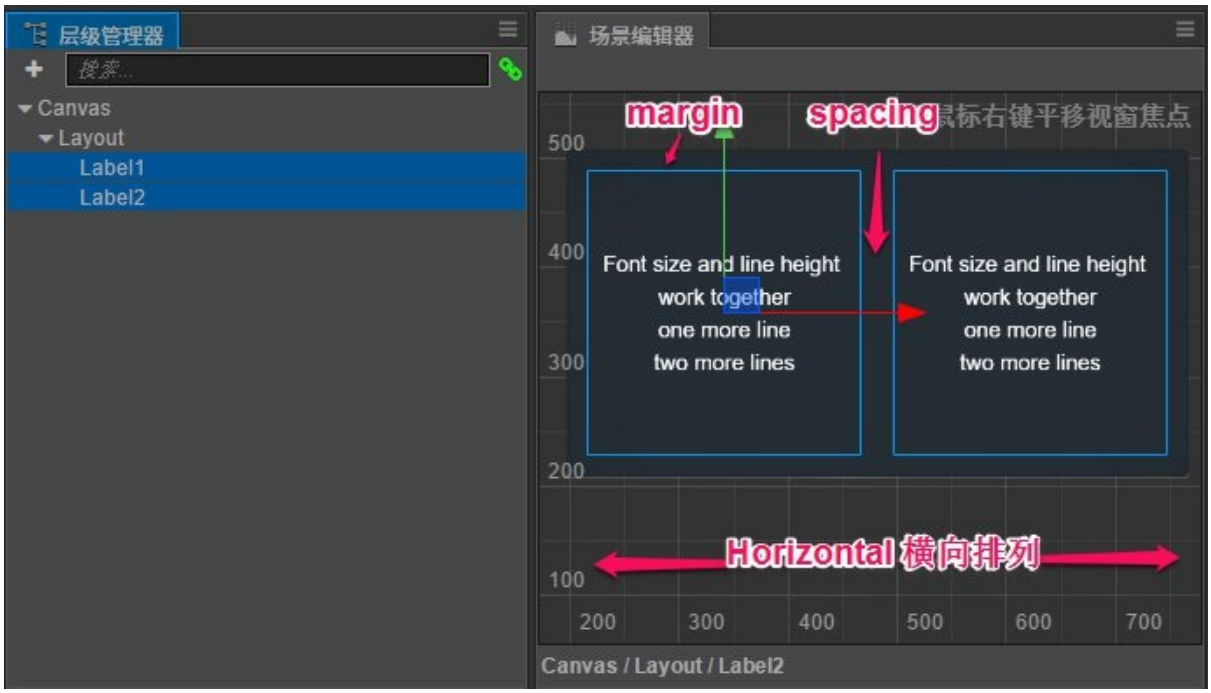
# 自动布局容器

Layout（自动布局）组件可以挂载在任何节点上，将节点变成一个有自动布局功能的容器。所谓自动布局容器，就是能够自动将子节点按照一定规律排列，并可以根据节点内容的约束框总和调整自身尺寸的容器型节点。

## 布局模式（Layout Type）

自动布局组件有几种基本的布局模式，可以通过 `Layout Type` 属性进行设置，包括以下几种。

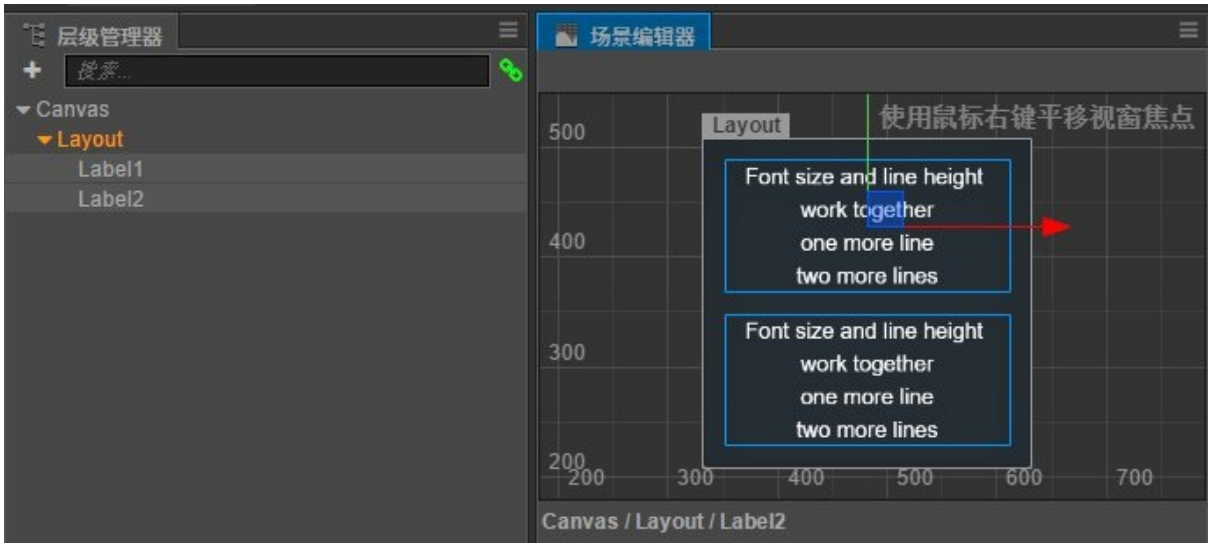
### 水平布局（Horizontal)




`Layout Type` 设为 `Horizontal` 时，所有子节点都会自动横向排列，并根据子节点的宽度（`Width`）总和设置 Layout 节点的宽度。上图中 Layout 包括的两个 Label 节点就自动被横向排列。

水平布局模式下，Layout 组件不会干涉节点在 y 轴上的位置或高度属性，子节点甚至可以放置在 Layout 节点的约束框高度范围之外。如果需要子节点在 y 轴向上对齐，可以在子节点上添加 Widget 组件，并开启 Top 或 Bottom 的对齐模式。

### 垂直布局（Vertical）




`Layout Type` 设为 `Vertical` 时，所有子节点都会自动纵向排列，并根据子节点的高度（ `Height` ）总和设置 Layout 节点的高度。

垂直布局模式下，Layout 组件也不会修改节点在 x 轴的位置或宽度属性，子节点需要添加 Widget 并开启 Left 或 Right 对齐模式才能规整的排列。

# 节点排列方向

Layout 排列子节点时，是以子节点在 **层级管理器** 中显示顺序为基准，加上 `Vertical Direction` 或 `Horizontal Direction` 属性设置的排列方向来排列的。

## 水平排列方向（Horizontal Direction）

可以设置 `Left to Right` 或 `Right to Left` 两种方向，前者会按照节点在 **层级管理器** 中显示顺序从左到右排列；后者会按照节点显示从右到左排列。

## 垂直排列方向（Vertical Direction）

可以设置 `Top to Bottom` 或 `Bottom to Top` 两种方向。前者会按照节点在 **层级管理器** 中显示顺序从上到下排列；后者会按照节点显示从下到上排列。

# 其他布局模式还在持续添加中

我们会在下个版本的文档中更新这部分内容。

其他 Layout 组件的属性请查阅Layout 组件参考文档。

# 制作动态生成内容的列表

UI 界面只有静态页面内容是不够的，我们会遇到很多需要由一组数据动态生成多个元素组成的 UI 面板，比如选人界面、物品栏、选择关卡等等。

# 准备数据

以物品栏为例，我们要动态生成一个物品，大概需要这样的一组数据：

- 物品 id
- 图标 id，我们可以在另一张资源表中建立图标 id 到对应 spriteFrame 的索引
- 物品名称
- 出售价格
- ...

下面我们将会结合脚本介绍如何定义和使用数据，如果您对 Cocos Creator 的脚本系统还不熟悉，可以先从 脚本开发工作流程一章开始学习。

## 自定义数据类

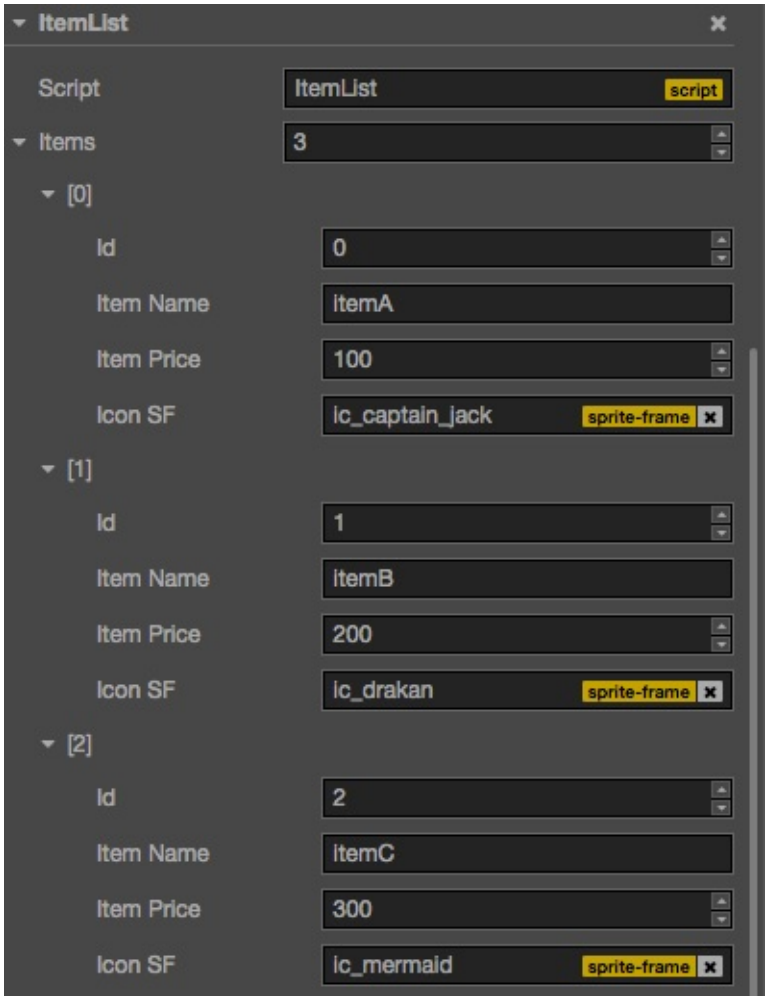
对于大多数游戏来说，这些数据通常都来自于服务器或本地的数据库，现在我们为了展示流程，暂时把数据存在列表组件里就可以了。您可以新建一个脚本 `ItemList.js`，并添加如下的属性：

```
var Item = cc.Class({
    name: 'Item',
    properties: {
        id: 0,
        itemName: '',
        itemPrice: 0,
        iconSF: cc.SpriteFrame
    }
});

cc.Class({
    extends: cc.Component,
    properties: {
        items: {
            default: [],
            type: Item
        }
    },
});
```

上面脚本的前半部分我们声明了一个叫做 `Item` 的数据类，用来存放我们展示物品需要的各种数据。注意这个类并没有继承 `cc.Component`，因此他不是一个组件，但可以被组件使用。关于声明自定义类的更多内容，请查阅自定义 Class文档。

下半部分是正常的组件声明方式，这个组件中只有一个 `items` 属性，上面的声明方式将会给我们一个由 `Item` 类组成的数组，我们可以在 **属性检查器** 中为每个 `Item` 元素设置数据。

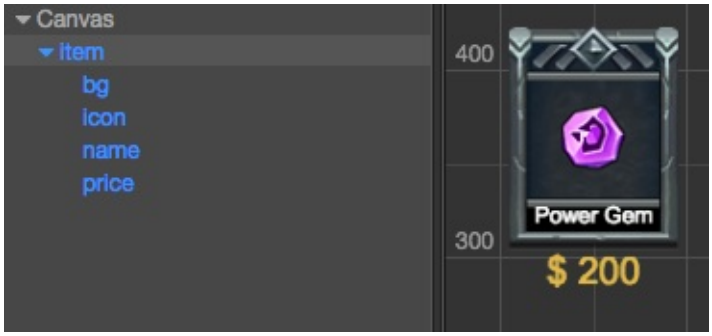
新建一个节点并将 `ItemList.js` 添加上去，我们可以在 **属性检查器** 里找到 `Items` 属性，要开始创建数据，需要先将数组的容量设为大于 0 的值。让我们将容量设为 3，并将每个元素的数据如下图设置。




这样我们最基本的数据就准备好了，如果您在制作有很多内容的游戏，请务必使用 excel、数据库等更专业的系统来管理您的数据，将外部数据格式转化为 Cocos Creator 可以使用的 JavaScript 和 JSON 格式都非常容易。

# 制作表现：Prefab 模板

接下来我们还需要一个可以在运行时用来实例化每个物品的模板资源 —— Prefab 预制。这个 Prefab 的结构如下图所示



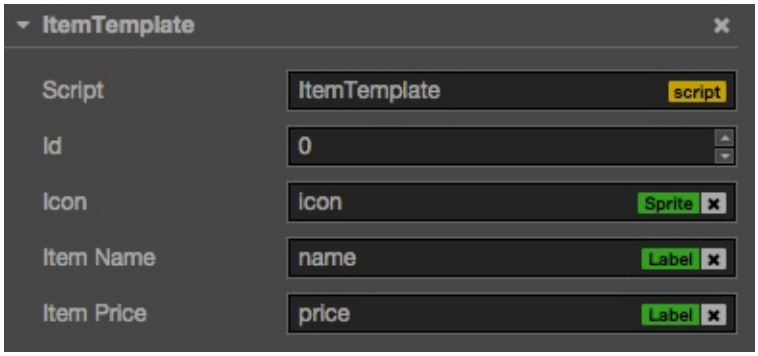

`icon`，`name`，`price` 节点之后就会用来展示图标、物品名称和价格的数据。

## 模板组件绑定

您在拼装 Prefab 时可以根据自己的需要自由发挥，上图中展示的仅仅是一个结构的例子。有了物品的模板结构，接下来我们需要一个组件脚本来完成节点结构的绑定。新建一个 `ItemTemplate.js` 的脚本，并将其添加到刚才制作的模板节点上。该脚本内容如下：

```
cc.Class({
    extends: cc.Component,
    properties: {
        id: 0,
        icon: cc.Sprite,
        itemName: cc.Label,
        itemPrice: cc.Label
    }
});
```

接下来将对应的节点拖拽到该组件的各个属性上：




注意 `id` 这个属性我们会直接通过数据赋值，不需要绑定节点。

### 通过数据更新模板表现

接下来我们需要继续修改 `ItemTemplate.js`，为其添加接受数据后进行处理的逻辑。在上述脚本后面加入以下内容：

```
    properties: {
    ...
    },
    // data: {id,iconSF,itemName,itemPrice}
    init: function (data) {
        this.id = data.id;
        this.icon.spriteFrame = data.iconSF;
        this.itemName.string = data.itemName;
        this.itemPrice.string = data.itemPrice;
    }
```

`init` 方法接受一个数据对象，并使用这个对象里的数据更新各个负责表现组件的相应属性。现在我们可以将 `Item` 节点保存成一个 Prefab 了，这就是我们物品的模板。

## 根据数据生成列表内容

现在让我们回到 `ItemList.js` 脚本，接下来要添加的是物品模板 Prefab 的引用，以及动态生成列表的逻辑。

```
    properties: {
        //...
        itemPrefab: cc.Prefab
    },

    onLoad () {
        for (var i = 0; i < this.items.length; ++i) {
```

```
            var item = cc.instantiate(this.itemPrefab);
            var data = this.items[i];
            this.node.addChild(item);
            item.getComponent('ItemTemplate').init({
                id: data.id,
                itemName: data.itemName,
                itemPrice: data.itemPrice,
                iconSF: data.iconSF
            });
        }
    }
```

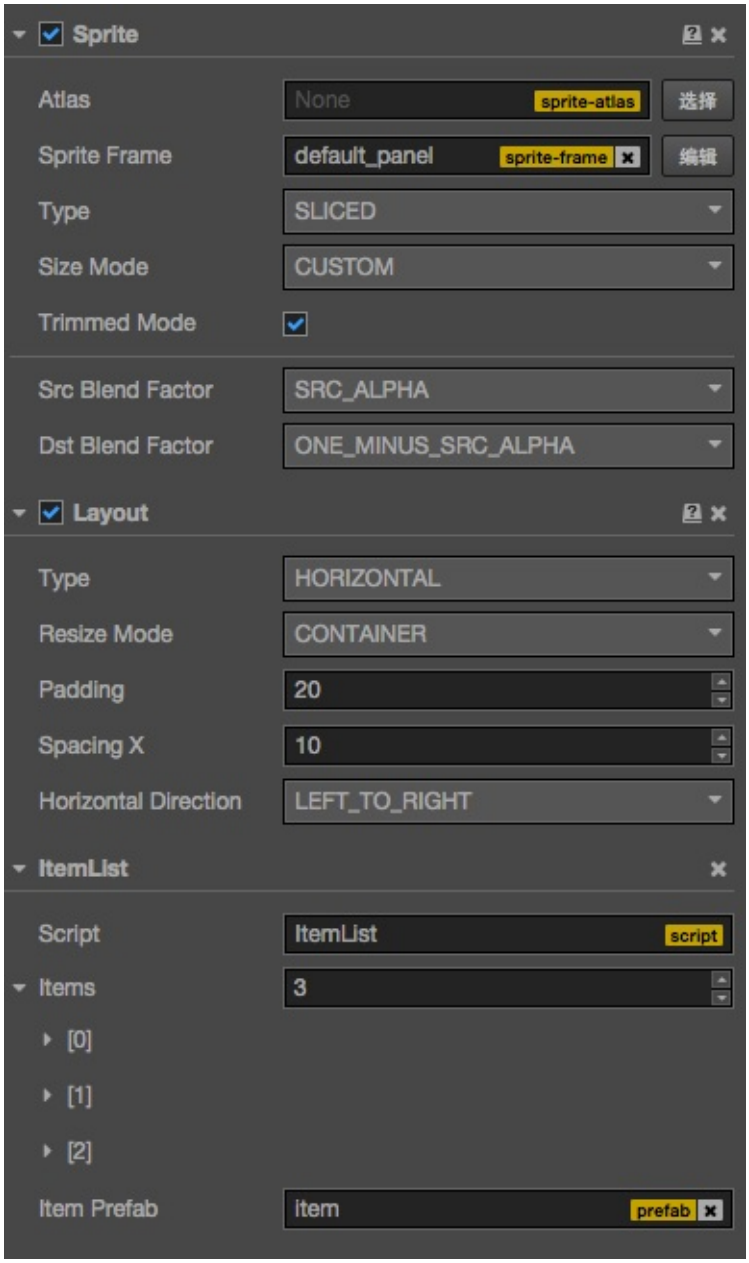
在 `onLoad` 回调方法里，我们依次遍历 `items` 里存储的每个数据，以 `itemPrefab` 为模板生成新节点并添加到 `ItemList.js` 所在节点上。之后调用 `ItemTemplate.js` 里的 `init` 方法，更新每个节点的表现。

现在我们可以为 `ItemList.js` 所在的节点添加一个 **Layout** 组件，通过 **属性检查器** 下方的 `添加组件/添加 UI 组件/Layout`，然后设置 **Layout** 组件的以下属性：

- `Type`：`HORIZONTAL`
- `Resize Mode`：`CONTAINER`

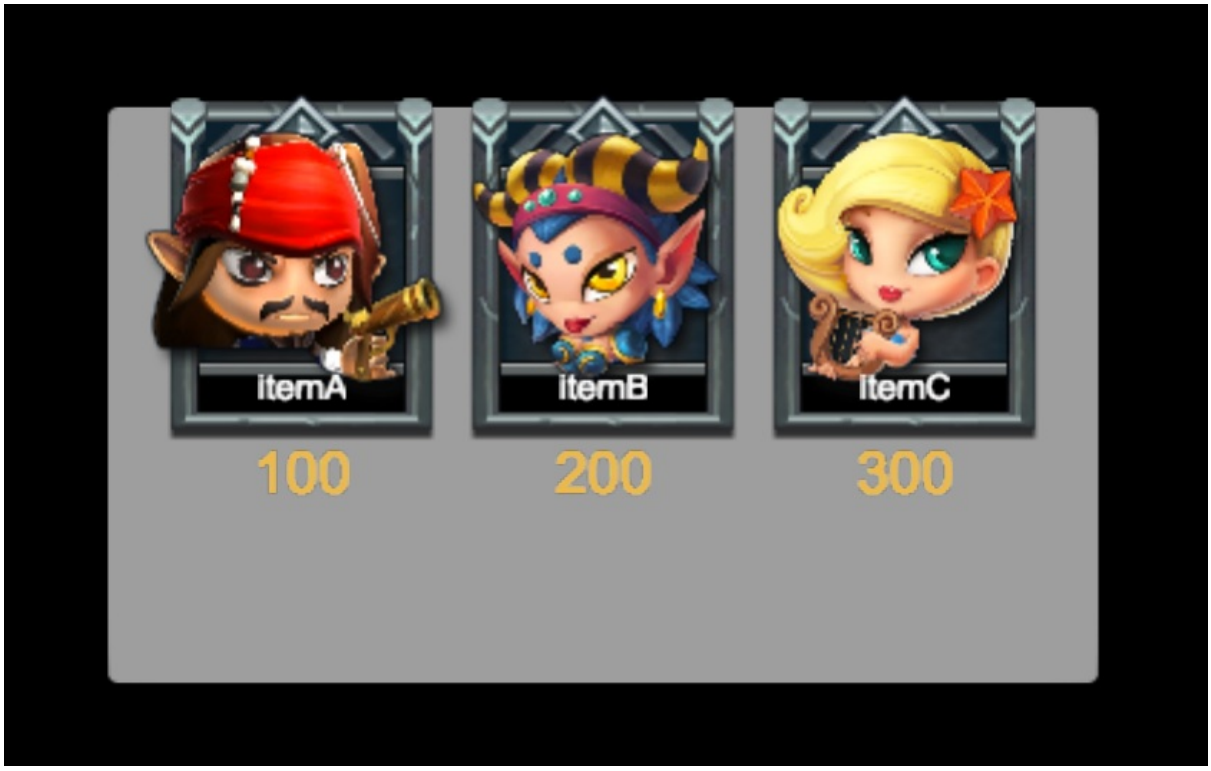
别忘了把 `item` Prefab 拖拽到 `ItemList` 组件的 `itemPrefab` 属性里。您还可以为这个节点添加一个 **Sprite** 组件，作为列表的背景。

完成后的 `itemList` 节点属性如下：




# 预览效果

最后运行预览，可以看到类似这样的效果（具体效果和您制作的物品模板，以及输入的数据有关）：



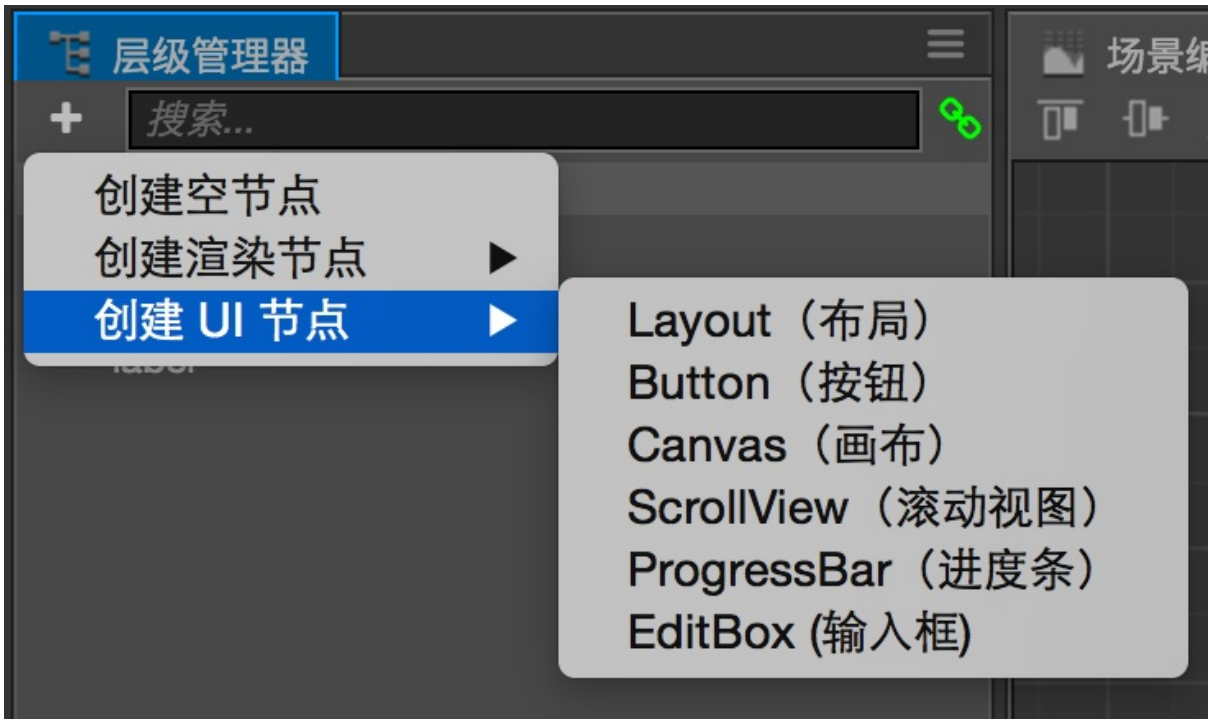

注意前面步骤中添加 **Layout** 组件并不是必须的，**Layout** 能够帮助您自动排列列表中的节点元素，但您也可以用脚本程序来控制节点的排列。我们通常还会配合 **ScrollView** 滚动视图组件一起使用，以便在有限的空间内展示大量内容。可以配合自动布局和滚动视图一起学习。

# 常用 UI 控件

本篇文档将介绍 UI 系统中常用的非核心控件，使用核心渲染组件和对齐策略，这些控件将构成我们游戏中 UI 的大部分交互部分。您将会了解以下 UI 控件的用法：

- ScrollView（滚动视图）、ScrollBar（滚动条）和 Mask（遮罩）
- Button（按钮）
- ProgressBar（进度条）
- EditBox（输入框）

以下介绍的 UI 控件都可以通过 **层级管理器** 左上角创建节点菜单中的 `创建 UI 节点` 子菜单来创建。




## ScrollView

ScrollView 是一个典型的组合型控件，通常由以下节点组成：

### ScrollView 根节点

这个节点上包含 ScrollView 组件，组件属性的详细说明可以查阅 ScrollView 组件参考

### content （内容）节点

content 节点用来承载将会在滚动视图中显示的内容，这个节点的约束框通常会远远大于 ScrollView 根节点的约束框，也只有在 content 节点比 ScrollView 节点大时，视图才能有效的滚动。

content 节点可以包括任意数量的子节点，配合 Layout 组件，可以确保 content 节点的约束框等于所有子节点约束框的总和。

### Mask（遮罩）节点

ScrollView 中的遮罩是可选的，但我们通常都希望能够隐藏 content 中超出 ScrollView 约束框范围的内容。

Mask 组件能够隐藏自身约束框范围外的子节点内容，注意 **Mask** 属于渲染组件，因此不能和其他渲染组件（如 Sprite，Label 等）共存于同一个节点上，我们需要额外的一个节点专门用来放置 Mask，否则 ScrollView 将无法设置用于背景的 **Sprite** 组件。

### ScrollBar（滚动条）节点

滚动条也是可选的，在有鼠标的设备上，我们可以通过滚动条提供鼠标拖拽快速滚动的功能。而在移动设备上，滚动条通常只用于指示内容的总量和当前显示范围。

我们可以同时设置横向和纵向两个滚动条，每个滚动条节点都包含一个 **ScrollBar** 组件。滚动条节点也可以包括子节点，来同时显示滚动条的前景和背景。详细的属性设置请查阅 ScrollBar 组件参考。

另外值得注意的是，ScrollBar 的 **handle** 部分的尺寸是可变的，推荐使用 **Sliced** （九宫格）模式的 Sprite 作为 ScrollBar 的 handle。

## Button（按钮）

通过 **层级管理器** 菜单创建的 **Button** 节点，由带有 **Button** 组件的父节点和一个带有 **Label** 组件的子节点组成。Button 父节点提供交互功能和按钮背景图显示，Label 子节点提供按钮上标签文字的渲染。

您可以根据美术风格和设计需要，将 Label 节点删除或者替换成需要的其他图标 Sprite。

关于 **Button** 组件的详细属性说明可以查阅 Button 组件参考。

### Transition 说明

Button 的 `Transition` 属性用于设置当按钮处在普通（Normal）、按下（Pressed）、悬停（Hover）、禁用（Disabled）四种状态下 `Target` 属性引用的背景图节点的表现。可以从以下三种模式中选择：

- `NONE` （无 Transtion），这个模式下按钮不会自动响应交互事件来改变自身外观，但您可以在按钮上加入自定义的脚本来精确控制交互的表现行为。
- `COLOR` （颜色变化），选择这个模式后，能看到四种状态属性并可以为每一种状态设置一个颜色叠加，在按钮转换到对应状态时，设置的状态颜色会和按钮背景图的颜色进行相乘作为展示的颜色。这个模式还允许通过 `Duration` 属性设置颜色变化过程的时间长度，实现颜色渐变的效果。
- `SPRITE` （图片切换），选择这个模式后，可以为四种状态分别指定一个 `SpriteFrame` 图片资源，当对应的状态被激活后，按钮背景图就会被替换为对应的图片资源。要注意如果设置了 `Normal` 状态的图片资源，按钮背景的 Sprite 属性中的 `SpriteFrame` 会被覆盖。

### Click Events 点击事件

`Click Events` 属性是一个数组类型，将它的容量改为 `1` 或更多就可以为按钮按下（鼠标或触摸）事件添加一个或多个响应回调方法。新建 `Click Events` 后，就可以拖拽响应回调方法所在组件的节点到 `Click Event` 的 `Target` 属性上，然后选择节点上的一个组件，并从列表中选择组件里的某个方法作为回调方法。

Button 上的点击事件是为了方便设计师在制作 UI 界面时可以自行指定按钮功能而设置的，要让按钮按照自定义的方式响应更多样化的事件，可以参考 系统内置事件文档，手动在按钮节点上监听这些交互事件并做出处理。

## ProgressBar（进度条）

进度条是由 **ProgressBar** 组件驱动一个 **Sprite** 节点的属性来实现根据设置的数值显示不同长度或角度的进度。ProgressBar 有三种基本工作模式（由 `Mode` 属性设置）：

- HORIZONTAL 水平进度条
- VERTICAL 垂直进度条
- FILLED 填充进度条

其他的基本属性设置请查阅ProgressBar 组件参考。

### 水平和垂直模式（HORIZONTAL & VERTICAL）

模式选择 `HORIZONTAL` 或 `VERTICAL` 时，进度条通过修改 `Bar Sprite` 引用节点的尺寸（ `width` 或 `height` 属性），来改变进度条显示的长度。在这两种模式下 `Bar Sprite` 推荐使用 `Sliced` 九宫格显示模式，这样在节点尺寸产生拉伸的情况下仍能保持高质量的图像渲染结果。

在这两种模式下， `Total Length` 属性的单位是像素，用来指定进度条在 100% 的状态下（ `Progress` 属性值为 1） `Bar Sprite` 的长度。这个属性保证我们在编辑场景时可以自由设置 `Progress` 为小于 1 的值，而 `Bar Sprite` 总是能够记录我们希望的总长度。

### 填充模式（FILLED）

和上面两种模式不同，填充模式下的进度条会通过按照一定百分比剪裁 `Bar Sprite` 引用节点来显示不同进度，因此我们需要对 `Bar Sprite` 引用的 **Sprite** 组件进行特定的设置。首先将该 Sprite 的 `Type` 属性设置为 `FILLED`，然后可以选择一个填充方向（HORIZONTAL、VERTICAL、RADIAL），详情请查阅 Sprite 填充模式参考文档。

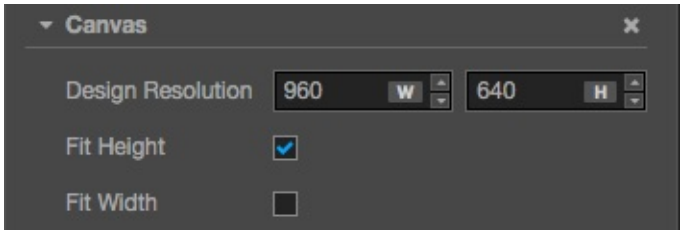
要注意进度条在选择了填充模式后， `Total Length` 的单位变成了百分比小数，取值范围从 0 ~ 1，设置的 `Total Length` 数值会同步到 `Bar Sprite` 的 `Fill Range` 属性，使之保持一致。下图显示了填充模式进度条当 `Bar Sprite` 的 `Fill Type` 设置为 `RADIAL` 时，不同的 `Total Length` 对显示的影响。


## EditBox（输入框）

输入框的使用方式比较直接，请直接参考 EditBox 组件参考 文档里每个属性的说明进行设置即可。

# Canvas（画布）组件参考

**Canvas（画布）**组件能够随时获得设备屏幕的实际分辨率并对场景中所有渲染元素进行适当的缩放。场景中的 Canvas 同时只能有一个，建议所有 UI 和可渲染元素都设置为 Canvas 的子节点。



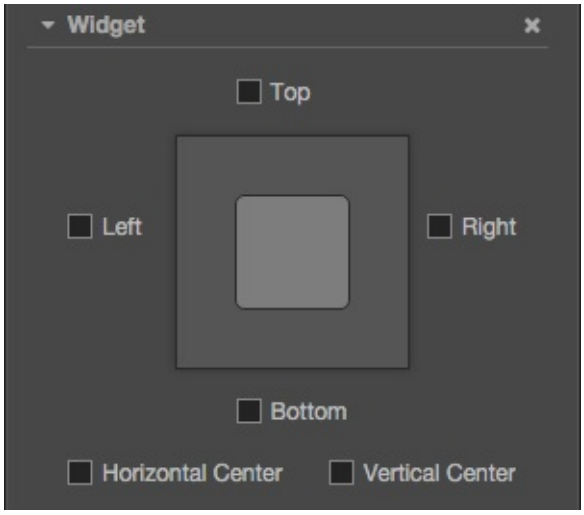

## 选项

| 选项 | 说明 |
| --- | --- |
| Design Resolution | 设计分辨率（内容生产者在制作场景时使用的分辨率蓝本） |
| Fit Height | 适配高度（设计分辨率的高度自动撑满屏幕高度） |
| Fit Width | 适配宽度（设计分辨率的宽度自动撑满屏幕宽度） |

画布的脚本接口请参考Canvas API。

# Widget 组件参考

Widget(对齐挂件) 是一个很常用的 UI 布局组件。它能使当前节点自动对齐到父物体的任意位置，或者约束尺寸，让你的游戏可以方便地适配不同的分辨率。




对齐挂件的脚本接口请参考Widget API。

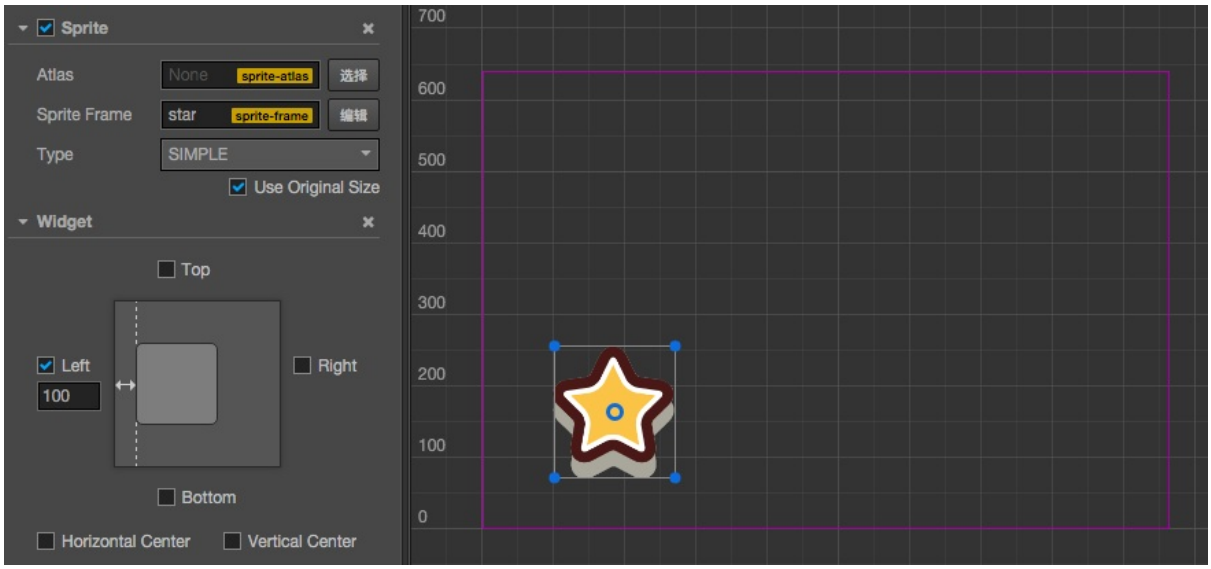
## 选项

| 选项 | 说明 | 备注 |
| --- | --- | --- |
| Top | 对齐上边界 | 选中后，将在旁边显示一个输入框，用于设定当前节点的上边界和父物体的上边界之间的距离。 |
| Bottom | 对齐下边界 | 选中后，将在旁边显示一个输入框，用于设定当前节点的下边界和父物体的下边界之间的距离。 |
| Left | 对齐左边界 | 选中后，将在旁边显示一个输入框，用于设定当前节点的左边界和父物体的左边界之间的距离。 |
| Right | 对齐右边界 | 选中后，将在旁边显示一个输入框，用于设定当前节点的右边界和父物体的右边界之间的距离。 |
| HorizontalCenter | 水平方向居中 | |
| VerticalCenter | 竖直方向居中 | |
| Target | 对齐目标 | 指定对齐参照的节点，当这里未指定目标时会使用直接父级节点作为对齐目标 |
| AlignOnce | 默认为 `true`，是否仅在组件初始化时进行一次对齐，设为 `false` 时，每帧都会对当前 Widget 组件执行对齐逻辑（对性能有较大损耗！）。 | |

## 对齐边界

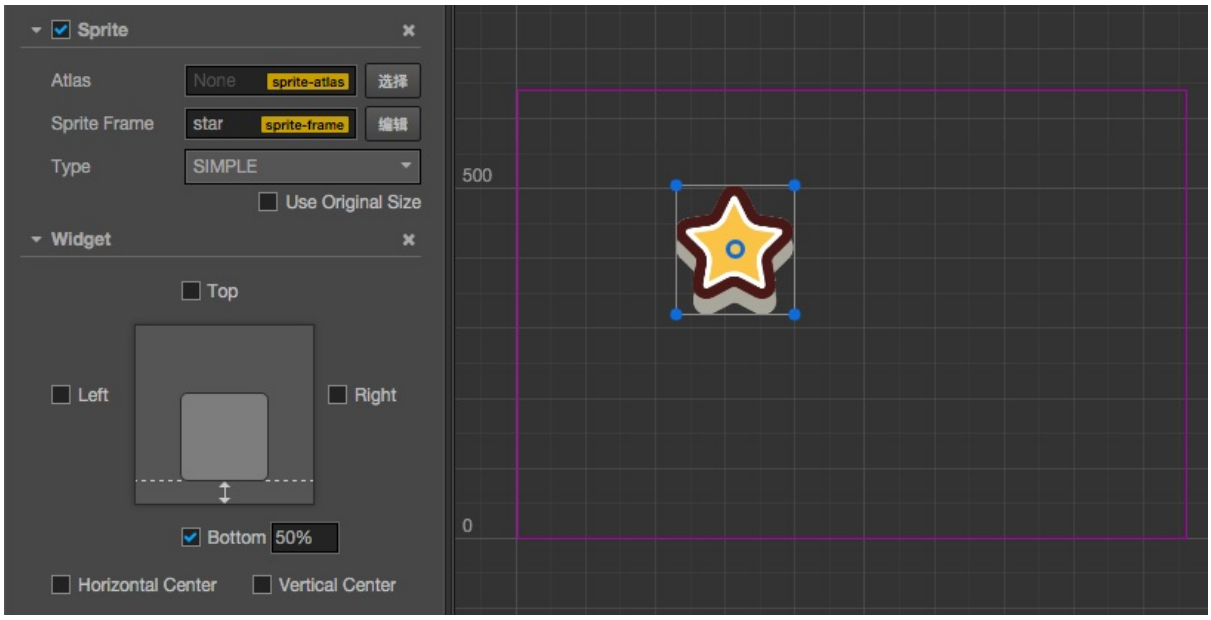
我们可以在 Canvas 下面放置一个 Widget，然后做如下一些测试：

## 左对齐，左边界距离 100 px：

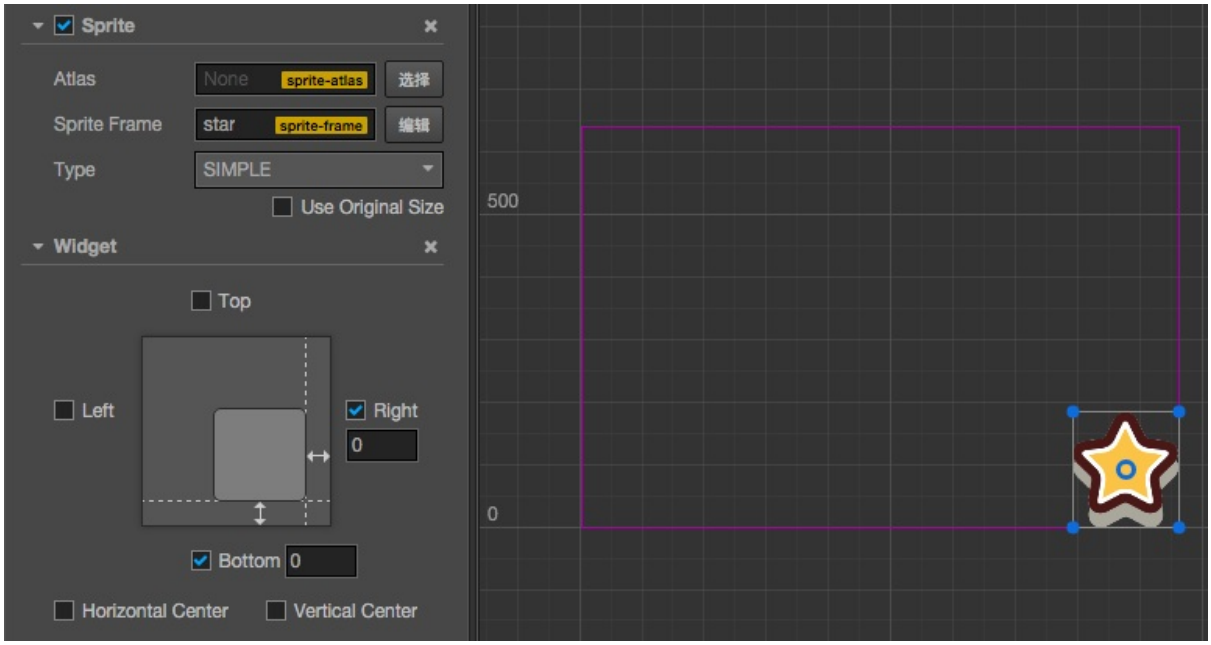



## 下对齐，左边界距离 50%：

百分比将以父物体的宽或高作为基准

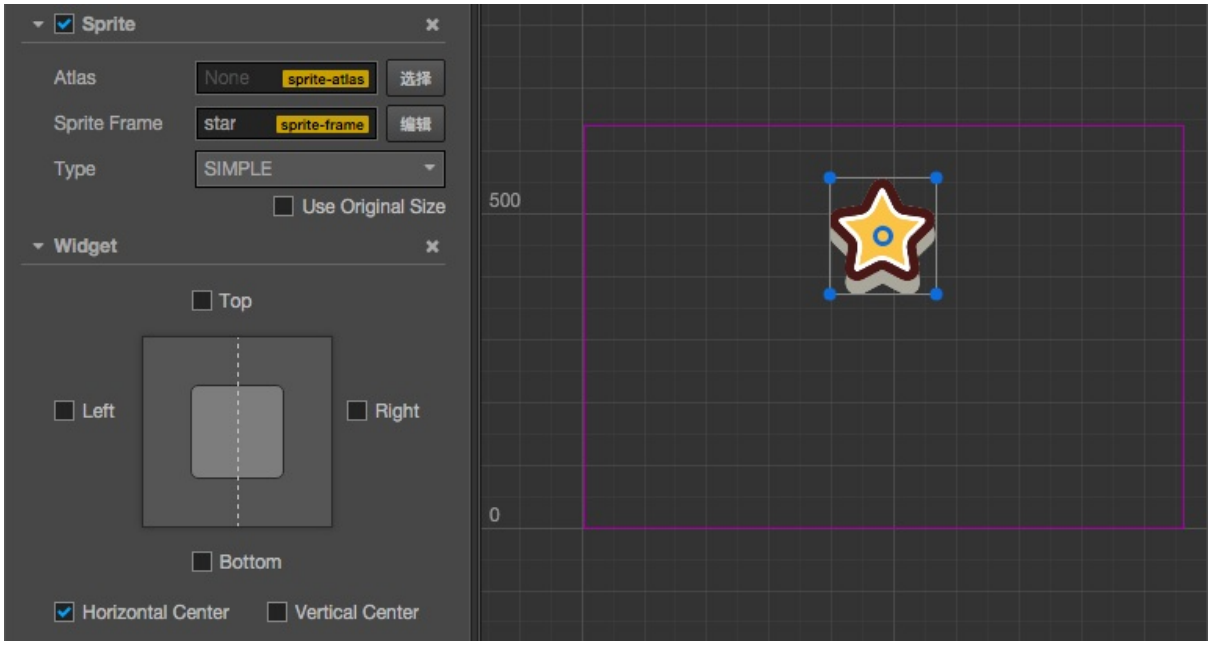



## 右下对齐，边界距离 0 px：

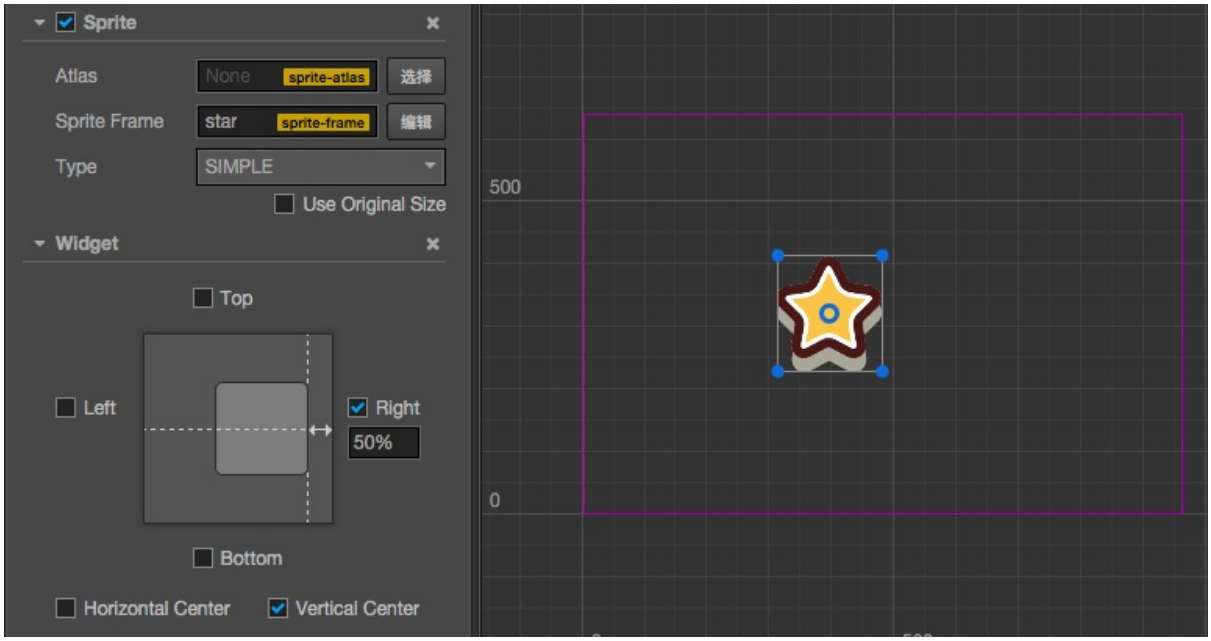



## 居中对齐

**水平方向居中：**



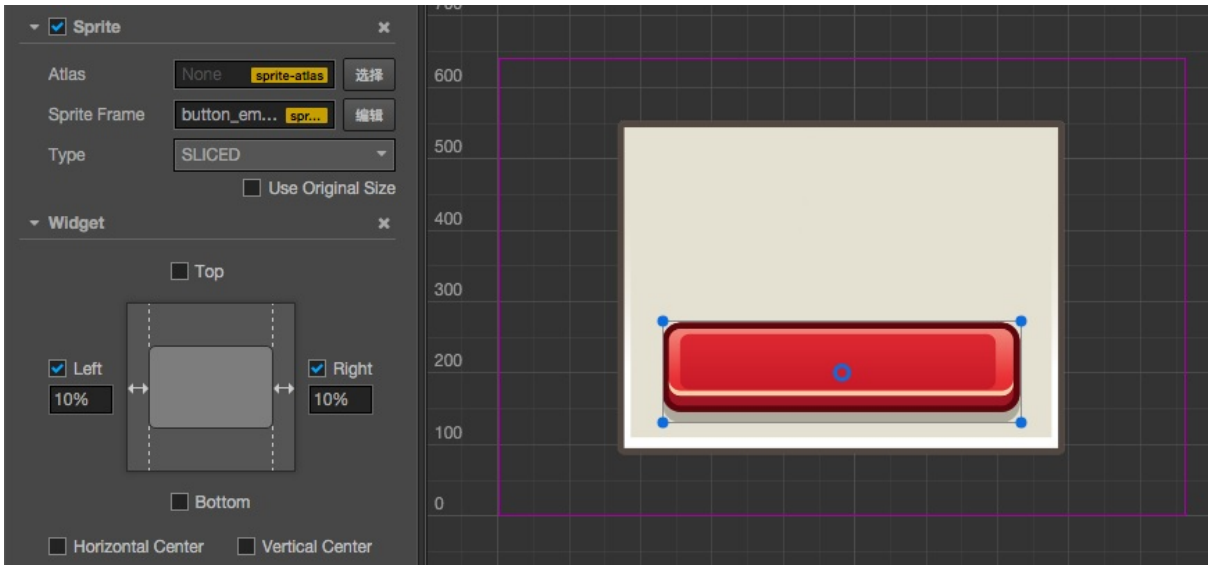

**竖直方向居中，并且右边界距离 50%：**



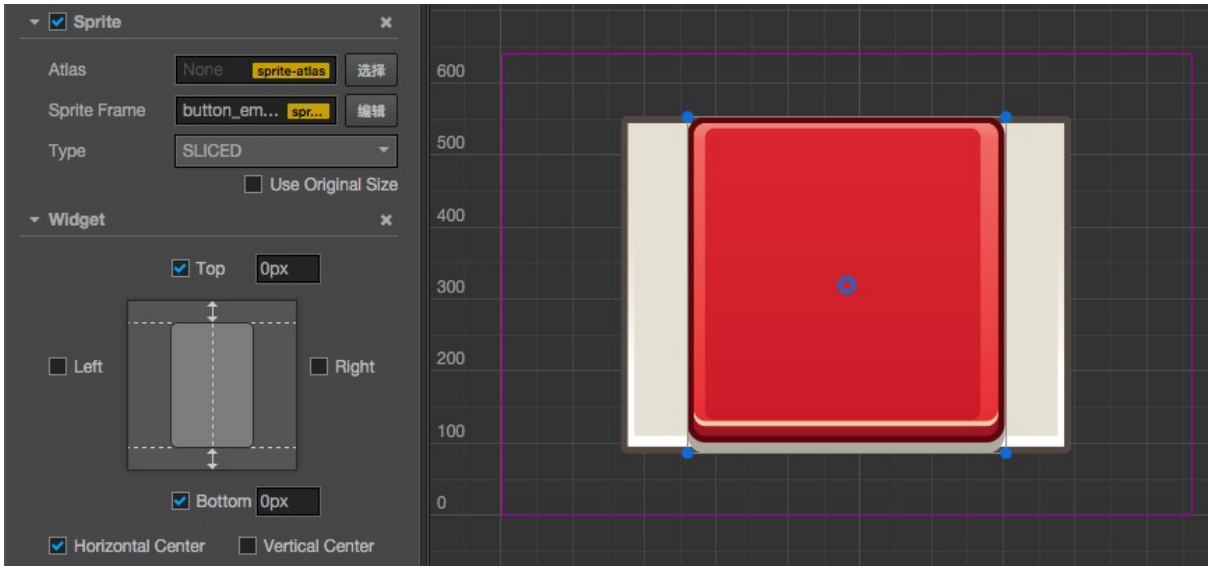

# 约束尺寸

如果左右同时对齐，或者上下同时对齐，那么在相应方向上的尺寸就会被拉伸。 下面演示一下，在场景中放置两个矩形 Sprite，大的作为对话框背景，小的作为对话框上的按钮。按钮节点作为对话框的子节点，并且按钮设置成 Sliced 模式以便展示拉伸效果。

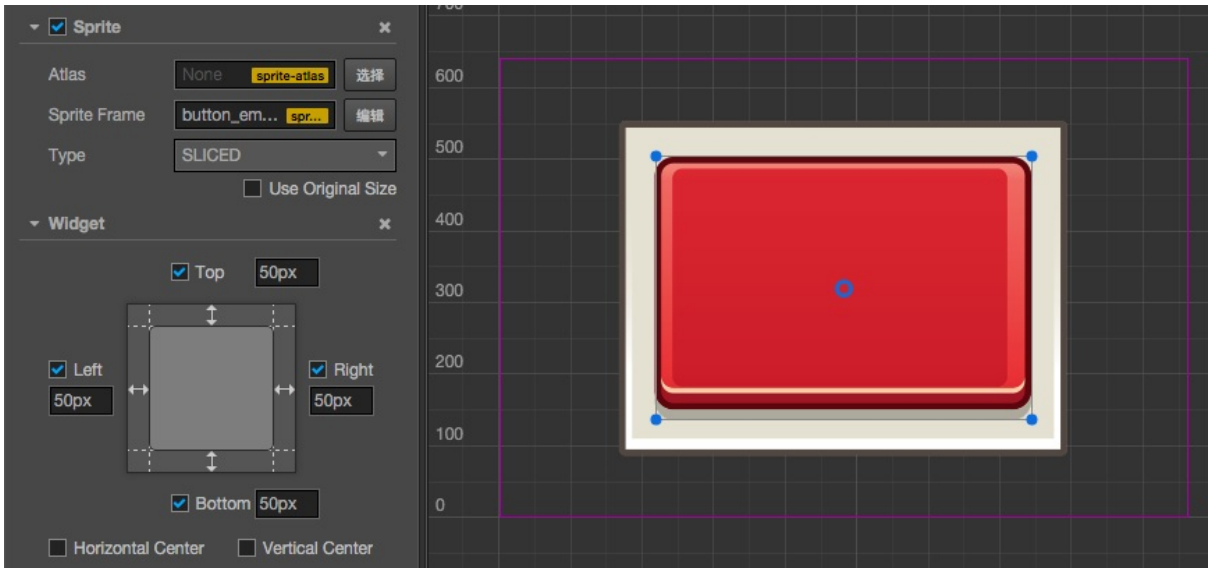
### 宽度拉伸，左右边距 10%：




### 高度拉伸，上下边距 0，同时水平居中：




### 水平和竖直同时拉伸，边距 50 px：




## 对节点位置、尺寸的限制

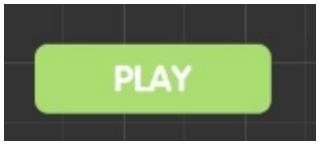
如果 `alignOnce` 属性设为 `false` 时，会在运行时每帧都按照设置的对齐策略进行对齐，组件所在节点的位置（position）和尺寸（width，height）属性可能会被限制，不能通过 API 或动画系统自由修改。这是因为通过 Widget 对齐是在每帧的最后阶段进行处理的，因此对 Widget 组件中已经设置了对齐的相关属性进行设置，最后都会被 Widget 组件本身的更新所重置。

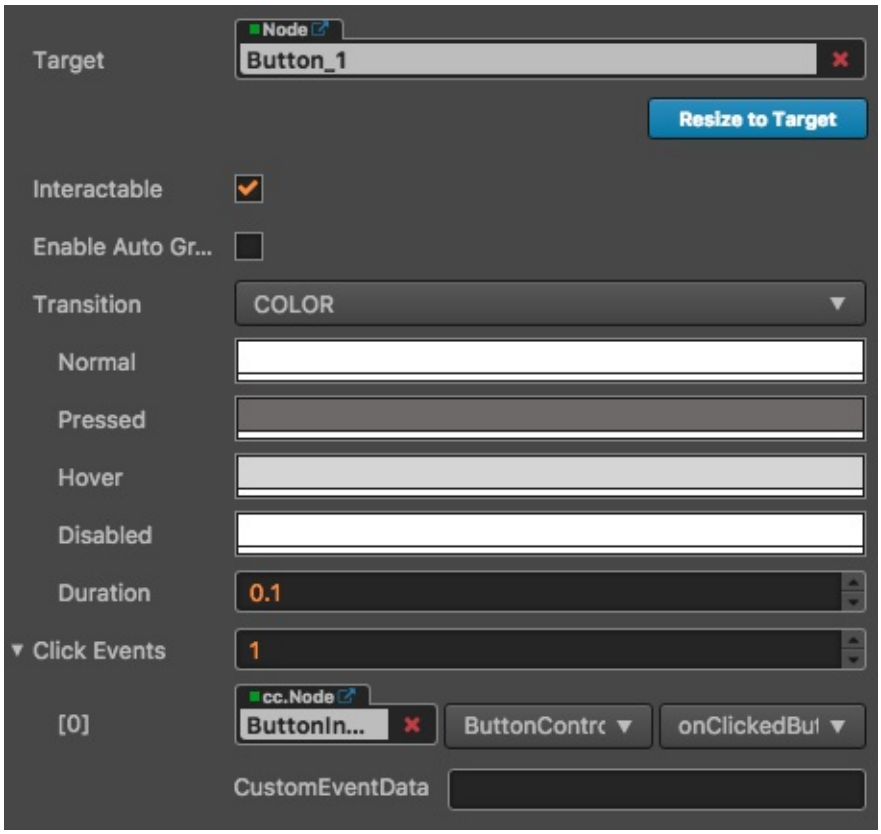
如果需要同时满足对齐策略和可以在运行时改变位置和尺寸的需要，可以通过以下两种方式实现：

1. 确保 **Widget** 组件的 `alignOnce` 属性处于选中状态，该属性只会负责在组件初始化（onEnable）时进行一次对齐，而不会每帧再进行一次对齐。可以在初始化时自动完成对齐，然后就可以通过 API 或动画系统对 UI 进行移动变换了。
2. 通过调用 **Widget** 组件的对齐边距 API，包括 `top`，`bottom`，`left`，`right`，直接修改 Widget 所在节点的位置或某一轴向的拉伸。这些属性也可以在动画编辑器中添加相应关键帧，保证对齐的同时实现各种丰富的 UI 动画。

# Button（按钮）组件参考

Button 组件可以响应用户的点击操作，当用户点击 Button 时，Button 自身会有状态变化。另外，Button 还可以让用户在完成点击操作后响应一个自定义的行为。




点击**属性检查器**下面的 `添加组件` 按钮，然后从 `添加 UI 组件` 中选择 `Button`，即可添加 Button 组件到节点上。

按钮的脚本接口请参考Button API。

# Button 属性

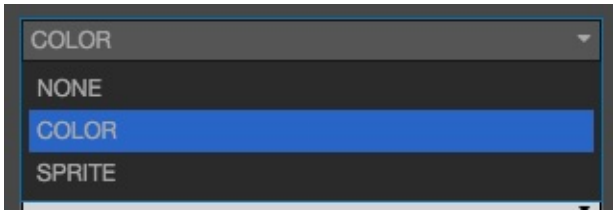
| 属性 | 功能说明 |
| --- | --- |
| Target | Node 类型，当 Button 发生 Transition 的时候，会相应地修改 Target 节点的 SpriteFrame，颜色或者 Scale。 |
| interactable | 布尔类型，设为 false 时，则 Button 组件进入禁用状态。 |
| enableAutoGrayEffect | 布尔类型，当设置为 true 的时候，如果 button 的 interactable 属性为 false，则 button 的 sprite Target 会使用内置 shader 变灰。 |
| Transition | 枚举类型，包括 NONE, COLOR，SPRITE 和 SCALE。每种类型对应不同的 Transition 设置。详情见 `Button Transition` 章节。 |
| Click Event | 列表类型，默认为空，用户添加的每一个事件由节点引用，组件名称和一个响应函数 |

| | |
|---|---|
| Click Event | 组成。详情见 `Button 事件` 章节 |

**注意**：当 Transition 为 SPRITE 且 disabledSprite 属性有关联一个 spriteFrame 的时候，此时不会使用内置 shader 来变灰

## Button Transition

Button 的 Transition 用来指定当用户点击 Button 时的状态表现。目前主要有 NONE，COLOR，SPRITE 和 SCALE。




## Color Transition


| 属性 | 功能说明 |
|---|---|
| Normal | Button 在 Normal 状态下的颜色。 |
| Pressed | Button 在 Pressed 状态下的颜色。 |
| Hover | Button 在 Hover 状态下的颜色。 |
| Disabled | Button 在 Disabled 状态下的颜色。 |
| Duration | Button 状态切换需要的时间间隔。 |

## Sprite Transition

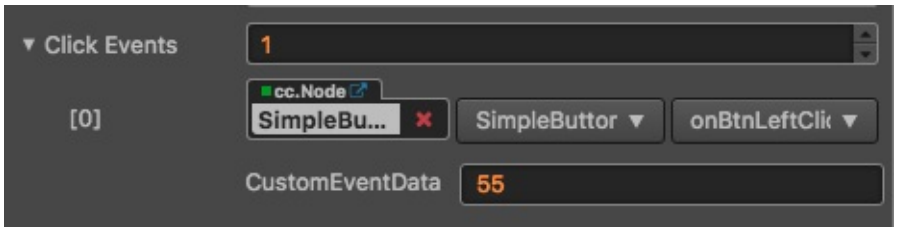

| 属性 | 功能说明 |
|---|---|
| Normal | Button 在 Normal 状态下的 SpriteFrame。 |
| | |

| | |
|---|---|
| Hover | Button 在 Hover 状态下的 SpriteFrame。 |
| Disabled | Button 在 Disabled 状态下的 SpriteFrame。 |

## Scale Transition


| 属性 | 功能 |
|---|---|
| Duration | Button 状态切换需要的时间间隔。 |
| ZoomScale | 当用户点击按钮后，按钮会缩放到一个值，这个值等于 Button 原始 scale * zoomScale，zoomScale 可以为负数 |

## Button 事件




| 属性 | 功能说明 |
|---|---|
| Target | 带有脚本组件的节点。 |
| Component | 脚本组件名称。 |
| Handler | 指定一个回调函数，当用户点击 Button 并释放时会触发此函数。 |
| CustomEventData | 用户指定任意的字符串作为事件回调的最后一个参数传入。 |

## 详细说明

Button 目前只支持 Click 事件，即当用户点击并释放 Button 时才会触发相应的回调函数。

## 通过脚本代码添加回调

#### 方法一

这种方法添加的事件回调和使用编辑器添加的事件回调是一样的，通过代码添加， 你需要首先构造一个 `cc.Component.EventHandler` 对象，然后设置好对应的 target, component, handler 和 customEventData 参数。

```
//here is your component file, file name = MyComponent.js
cc.Class({
    extends: cc.Component,
    properties: {},

    onLoad: function () {
        var clickEventHandler = new cc.Component.EventHandler();
        clickEventHandler.target = this.node; //这个 node 节点是你的事件处理代码组件所属的节点
        clickEventHandler.component = "MyComponent";//这个是代码文件名
```

```
        clickEventHandler.target = this.node; //这个 node 节点是你的事件处理代码组件所属的节点
        clickEventHandler.component = "MyComponent";//这个是代码文件名
        clickEventHandler.handler = "callback";
        clickEventHandler.customEventData = "foobar";

        var button = node.getComponent(cc.Button);
        button.clickEvents.push(clickEventHandler);
    },

    callback: function (event, customEventData) {
        //这里 event 是一个 Touch Event 对象，你可以通过 event.target 取到事件的发送节点
        var node = event.target;
        var button = node.getComponent(cc.Button);
        //这里的 customEventData 参数就等于你之前设置的 "foobar"
    }
});
```

### 方法二

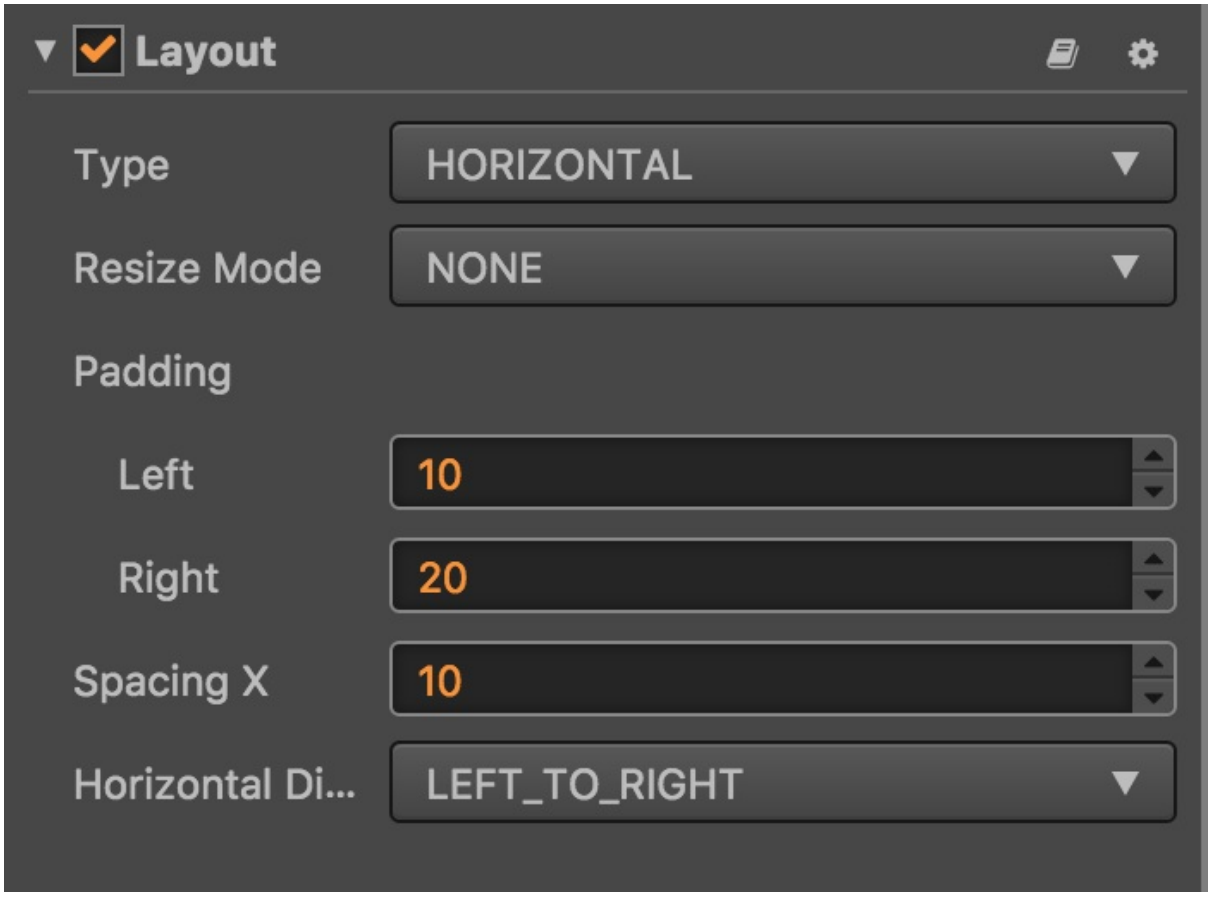
通过 `button.node.on('click', ...)` 的方式来添加，这是一种非常简便的方式，但是该方式有一定的局限性，在事件回调里面无法 获得当前点击按钮的屏幕坐标点。

```
//假设我们在一个组件的 onLoad 方法里面添加事件处理回调，在 callback 函数中进行事件处理：

cc.Class({
    extends: cc.Component,


    properties: {
       button: cc.Button
    },

    onLoad: function () {
       this.button.node.on('click', this.callback, this);
    },

    callback: function (event) {
       //这里的 event 是一个 EventCustom 对象，你可以通过 event.detail 获取 Button 组件
       var button = event.detail;
       //do whatever you want with button
       //另外，注意这种方式注册的事件，也无法传递 customEventData
    }
});
```

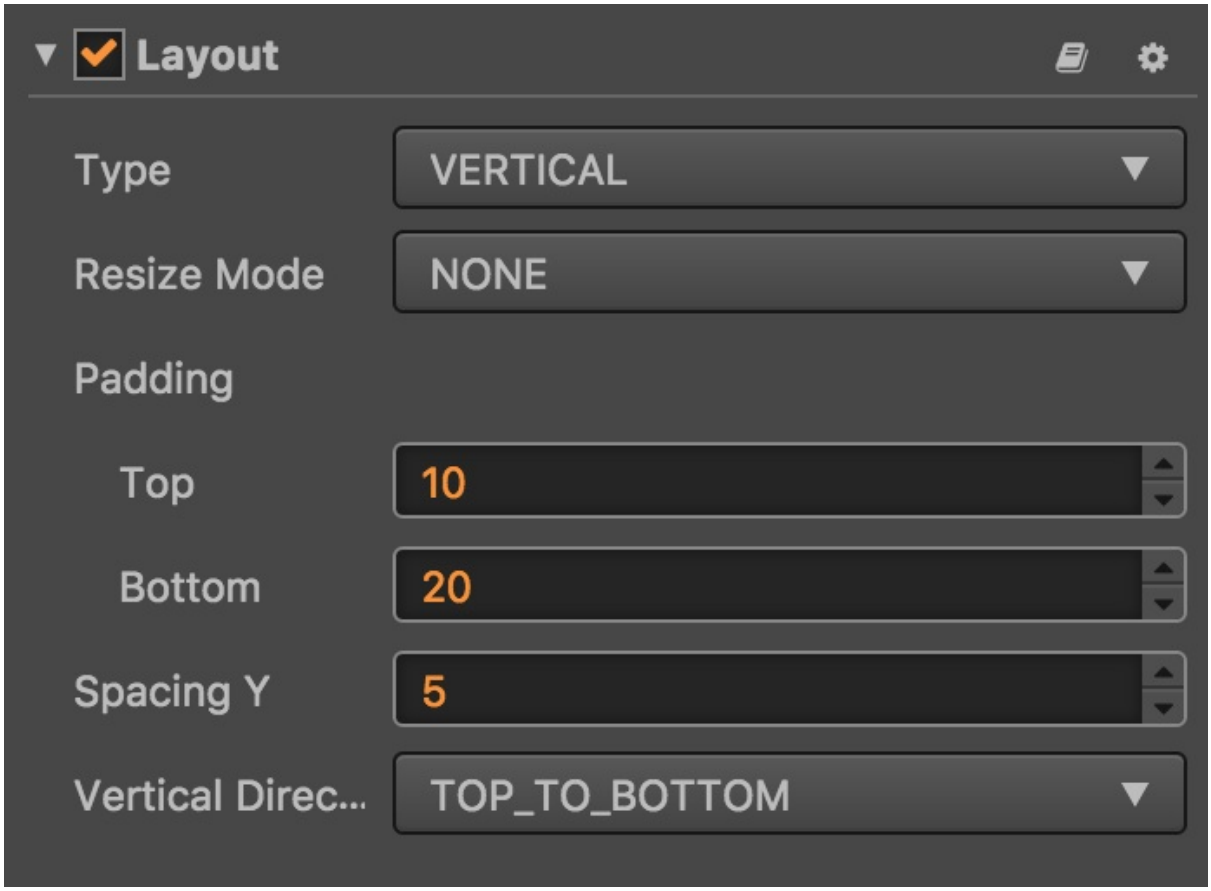
# Layout 组件参考

Layout 是一种容器组件，容器能够开启自动布局功能，自动按照规范排列所有子物体，方便用户制作列表、翻页等功能。

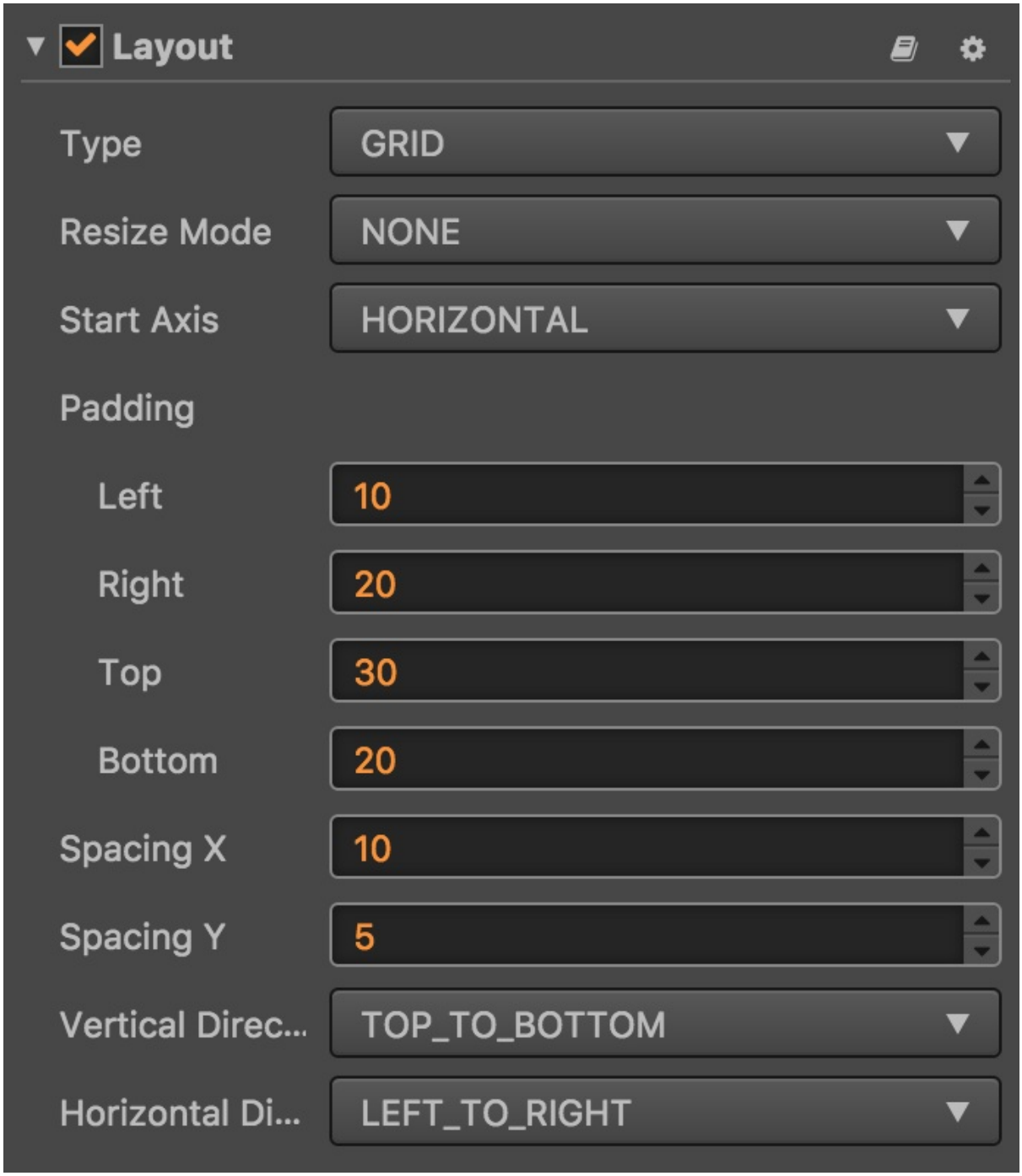
- 水平布局容器




- 垂直布局容器


Layout
Type
VERTICAL
Resize Mode
NONE
Padding
Top
10
Bottom
20
Spacing Y
5
Vertical Direc...
TOP_TO_BOTTOM

- 网格布局容器




点击**属性检查器**下面的 `添加组件` 按钮，然后从 `添加 UI 组件` 中选择 `Layout`，即可添加 Layout 组件到节点上。

布局的脚本接口请参考Layout API。

## Layout 属性

| 属性 | 功能说明 |
| --- | --- |
| Type | 布局类型，支持 NONE, HORIZONTAL，VERTICAL 和 GRID。 |
| ResizeMode | 缩放模式，支持 NONE，CHIDREN 和 CONTAINER。 |
| PaddingLeft | 排版时，子物体相对于容器左边框的距离。 |
| | |

| | |
|---|---|
| PaddingRight | 排版时，子物体相对于容器右边框的距离。 |
| PaddingTop | 排版时，子物体相对于容器上边框的距离。 |
| PaddingBottom | 排版时，子物体相对于容器下边框的距离。 |
| SpacingX | 水平排版时，子物体与子物体在水平方向上的间距。NONE 模式无此属性。 |
| SpacingY | 垂直排版时，子物体与子物体在垂直方向上的间距。NONE 模式无此属性。 |
| Horizontal Direction | 指定水平排版时，第一个子节点从容器的左边还是右边开始布局。当容器为 Grid 类型时，此属性和 Start Axis 属性一起决定 Grid 布局元素的起始水平排列方向。 |
| Vertical Direction | 指定垂直排版时，第一个子节点从容器的上面还是下面开始布局。当容器为 Grid 类型时，此属性和 Start Axis 属性一起决定 Grid 布局元素的起始垂直排列方向。 |
| Cell Size | 此属性只在 Grid 布局时存在，指定网格容器里面排版元素的大小。 |
| Start Axis | 此属性只在 Grid 布局时存在，指定网格容器里面元素排版指定的起始方向轴。 |

# 详细说明

添加 Layout 组件之后，默认的布局类型是 NONE，它表示容器不会修改子物体的大小和位置，当用户手动摆放子物体时，容器会以能够容纳所有子物体的最小矩形区域作为自身的大小。

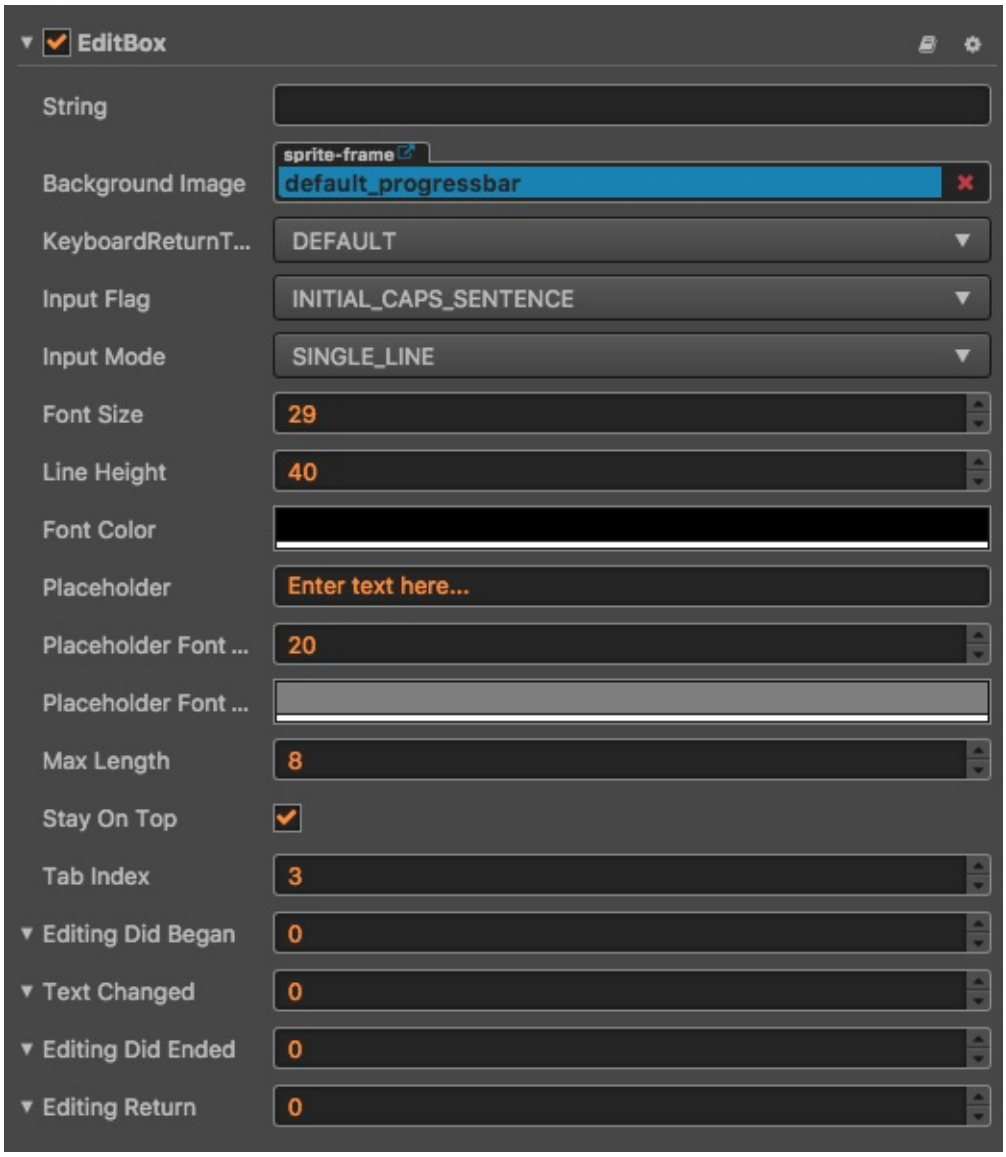
通过修改**属性检查器**里面的 `Type` 可以切换布局容器的类型，可以切换成水平，垂直或者网格布局。

另外，所有的容器均支持 ResizeMode（NONE 容器只支持 NONE 和 CONTAINER）。

- 当 ResizeMode 设置为 NONE 时，子物体和容器的大小变化互不影响。
- 设置为 CHILDREN 则子物体大小会随着容器的大小而变化。
- 设置为 CONTAINER 则容器的大小会随着子物体的大小变化。

# EditBox 组件参考

EditBox 是一种文本输入组件，该组件让你可以轻松获取用户输入的文本。




点击**属性检查器**下面的 `添加组件` 按钮，然后从 `添加 UI 组件` 中选择 `EditBox`，即可添加 EditBox 组件到节点上。

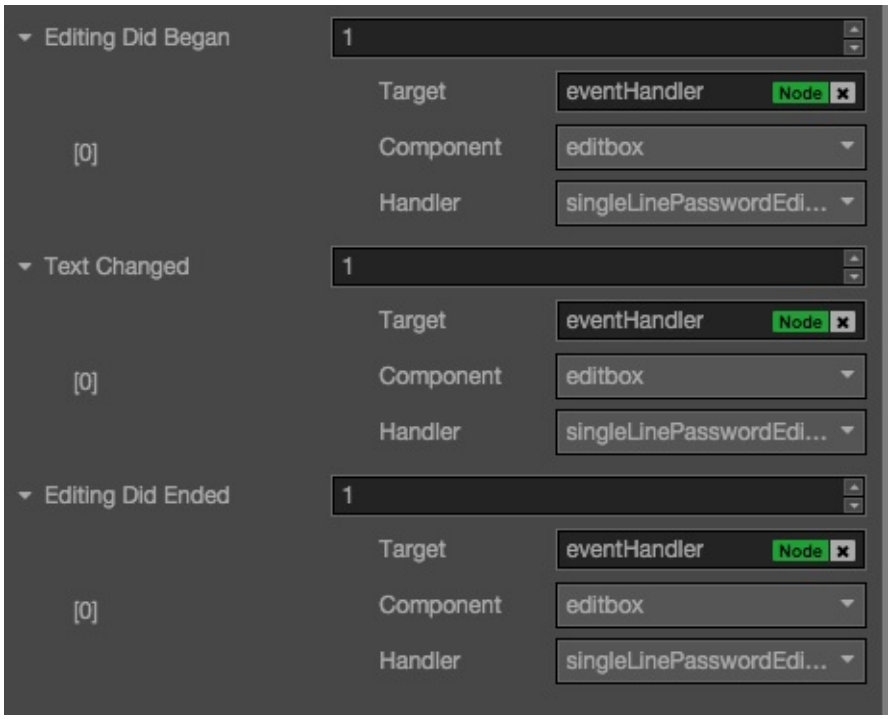
EditBox 的脚本接口请参考EditBox API。

# EditBox 属性

| 属性 | 功能说明 |
| --- | --- |
| String | 输入框的初始输入内容，如果为空则会显示占位符的文本 |
| Background Image | 输入框的背景图片 |
| Keyboard Return Type | 指定移动设备上面回车按钮的样式 |
| Input Flag | 指定输入标识：可以指定输入方式为密码或者单词首字母大写 |

| | |
|---|---|
| Input Mode | 指定输入模式: ANY 表示多行输入，其它都是单行输入，移动平台上还可以指定键盘样式。 |
| Font Size | 输入框文本的字体大小 |
| StayOnTop | 输入框总是可见，并且永远在游戏视图的上面 |
| TabIndex | 修改 DOM 输入元素的 tabIndex，这个属性只有在 Web 上面修改有意义。 |
| Line Height | 输入框文本的行高 |
| Font Color | 输入框文本的颜色 |
| Placeholder | 输入框占位符的文本内容 |
| Placeholder Font Size | 输入框占位符的字体大小 |
| Placeholder Font Color | 输入框占位符的字体颜色 |
| Max Length | 输入框最大允许输入的字符个数 |

# EditBox 事件




## Editing Did Began 事件

| 属性 | 功能说明 |
|---|---|
| Target | 带有脚本组件的节点。 |
| Component | 脚本组件名称。 |
| Handler | 指定一个回调函数，当用户开始输入文本的时候会调用该函数 |
| CustomEventData | 用户指定任意的字符串作为事件回调的最后一个参数传入。 |

说明：该事件在用户点击输入框获取焦点的时候被触发。

### Text Changed 事件

| 属性 | 功能说明 |
| --- | --- |
| Target | 带有脚本组件的节点。 |
| Component | 脚本组件名称。 |
| Handler | 指定一个回调函数，当用户正在输入文本的时候会调用该函数 |
| CustomEventData | 用户指定任意的字符串作为事件回调的最后一个参数传入。 |

说明：该事件在用户每一次输入文字变化的时候被触发。

### Editing Did Ended 事件

| 属性 | 功能说明 |
| --- | --- |
| Target | 带有脚本组件的节点。 |
| Component | 脚本组件名称。 |
| Handler | 指定一个回调函数，当用户输入文本结束时会调用该函数。 |
| CustomEventData | 用户指定任意的字符串作为事件回调的最后一个参数传入。 |

说明：在单行模式下面，一般是在用户按下回车或者点击屏幕输入框以外的地方调用该函数。 如果是多行输入，一般是在用户点击屏幕输入框以外的地方调用该函数。

### Editing Return 事件

| 属性 | 功能说明 |
| --- | --- |
| Target | 带有脚本组件的节点。 |
| Component | 脚本组件名称。 |
| Handler | 指定一个回调函数，当用户输入文本按下回车键时会调用该函数。 |
| CustomEventData | 用户指定任意的字符串作为事件回调的最后一个参数传入。 |

说明：该事件在用户按下回车键的时候被触发, 如果是单行输入框，按回车键还会使输入框失去焦点。

## 详细说明

- Keyboard Return Type 特指在移动设备上面进行输入的时候，弹出的虚拟键盘上面的回车键样式。
- 如果需要输入密码，则需要把 Input Flag 设置为 password，同时 Input Mode 必须是 Any 之外的选择，一般选择 Single Line。
- 如果要输入多行，可以把 Input Mode 设置为 Any。
- 背景图片支持九宫格缩放

注意：如果在 iframe 里面使用，最好把 `stayOnTop` 属性设置为 true

### 通过脚本代码添加回调

**方法一**

这种方法添加的事件回调和使用编辑器添加的事件回调是一样的，通过代码添加， 你需要首先构造一个 `cc.Component.EventHandler` 对象，然后设置好对应的 target, component, handler 和 customEventData 参数。

```
var editboxEventHandler = new cc.Component.EventHandler();
editboxEventHandler.target = this.node; //这个 node 节点是你的事件处理代码组件所属的节点
editboxEventHandler.component = "cc.MyComponent"
editboxEventHandler.handler = "onEditDidBegan";
editboxEventHandler.customEventData = "foobar";

editbox.editingDidBegan.push(editboxEventHandler);
// 你也可以通过类似的方式来注册其它回调函数
//editbox.editingDidEnded.push(editboxEventHandler);
//editbox.textChanged.push(editboxEventHandler);
//editbox.editingReturn.push(editboxEventHandler);


//here is your component file
cc.Class({
    name: 'cc.MyComponent'
    extends: cc.Component,

    properties: {
    },

    onEditDidBegan: function(editbox, customEventData) {
        //这里 editbox 是一个 cc.EditBox 对象
        //这里的 customEventData 参数就等于你之前设置的 "foobar"
    },
    //假设这个回调是给 editingDidEnded 事件的
    onEditDidEnded: function(editbox, customEventData) {
        //这里 editbox 是一个 cc.EditBox 对象
        //这里的 customEventData 参数就等于你之前设置的 "foobar"
    }
    //假设这个回调是给 textChanged 事件的
    onTextChanged: function(text, editbox, customEventData) {
        //这里的 text 表示 修改完后的 EditBox 的文本内容
        //这里 editbox 是一个 cc.EditBox 对象
        //这里的 customEventData 参数就等于你之前设置的 "foobar"
    }
    //假设这个回调是给 editingReturn 事件的
    onEditingReturn: function(editbox,  customEventData) {
        //这里 editbox 是一个 cc.EditBox 对象
        //这里的 customEventData 参数就等于你之前设置的 "foobar"
    }
});
```

### 方法二

通过 `editbox.node.on('editing-did-began', ...)` 的方式来添加

```
//假设我们在一个组件的 onLoad 方法里面添加事件处理回调，在 callback 函数中进行事件处理：

cc.Class({
    extends: cc.Component,


    properties: {
       editbox: cc.EditBox
    },

    onLoad: function () {
       this.editbox.node.on('editbox', this.callback, this);
    },

    callback: function (event) {
       //这里的 event 是一个 EventCustom 对象，你可以通过 event.detail 获取 EditBox 组件
```

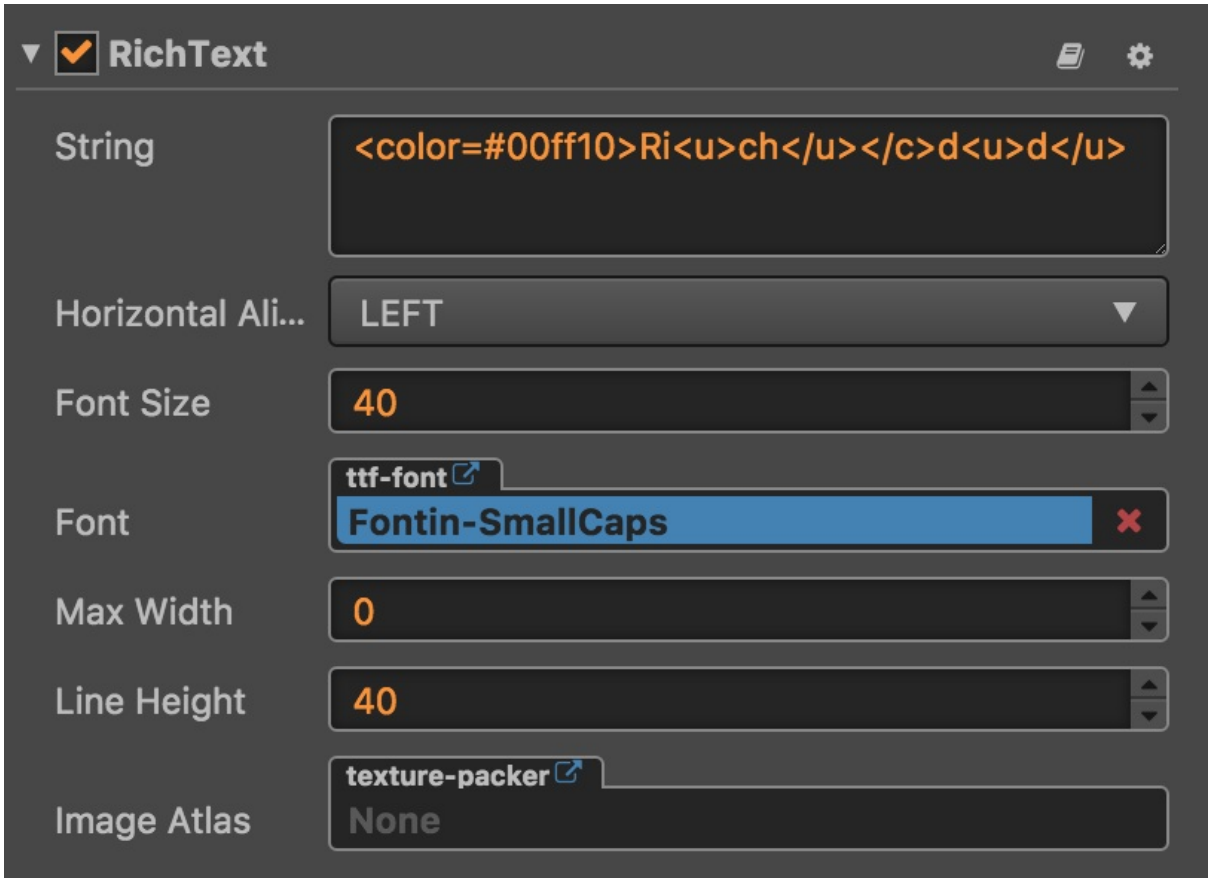
```
        var editbox = event.detail;
        //do whatever you want with the editbox
    }
});
```

同样的，你也可以注册 'editing-did-ended'，'text-changed' 和 'editing-return' 事件，这些事件的回调函数的参数与 'editing-did-began' 的参数一致。

# RichText 组件参考

RichText 组件用来显示一段带有不同样式效果的文字，你可以通过一些简单的 BBCode 标签来设置文字的样式。 目前支持的样式有：颜色(color)，字体大小(size)，字体描边(outline)，加粗(b)，斜体(i)，下划线(u)，换行(br)，图片(img)和点击事件(on)，并且不同的 BBCode 标签是可以支持相互嵌套的。

更多关于 BBCode 标签的内容，请参考本文档的 `BBCode 标签格式说明` 小节。




点击**属性检查器**下面的 `添加组件` 按钮，然后从 `添加渲染组件` 中选择 `RichText`，即可添加 RichText 组件到节点上。

富文本的脚本接口请参考 RichText API。

# RichText 属性

| 属性 | 功能说明 |
| --- | --- |
| String | 富文本的内容字符串, 你可以在里面使用 BBCode 来指定特定文本的样式 |
| Horizontal Align | 水平对齐方式 |
| Font Size | 字体大小, 单位是 point （注意，该字段不会影响 BBCode 里面设置的字体大小） |
| Font | 富文本定制字体，所有的 label 片断都会使用这个定制的 TTF 字体 |
| Line Height | 字体行高, 单位是 point |
| Max Width | 富文本的最大宽度, 传 0 的话意味着必须手动换行. |
| | |

| | |
|---|---|
| Image Atlas | 对于 img 标签里面的 src 属性名称，都需要在 imageAtlas 里面找到一个有效的 spriteFrame，否则 img tag 会判定为无效。 |

# BBCode 标签格式

## 基本格式

目前支持的标签类型有：size，color, b, i, u, img 和 on，分别用来定制字体大小，字体颜色, 加粗，斜体，下划线，图片和点击事件。 每一个标签都有一个起始标签和一个结束标签，起始标签的名字和属性格式必要符合要求，且全部为小写。 结束标签的名字不做任何检查，只需要满足结束标签的定义即可。

下面分别是应用 size 和 color 标签的一个例子：

```
<color=green>你好</color>，<size=50>Creator</>
```

## 支持标签

注意：所有的 tag 名称必须是小写，且属性值是用=号赋值

| 名称 | 描述 | 示例 | 注意事项 |
|---|---|---|---|
| color | 指定字体渲染颜色，颜色值可以是内置颜色，比如 white, black 等，也可以使用 16 进制颜色值，比如#ff0000 表示红色 | `<color=#ff0000>Red Text</color>` | 内置颜色值参考 cc.Color |
| size | 指定字体渲染大小，大小值必须是一个整数 | `<size=30>enlarge me</size>` | Size 值必须使用等号赋值 |
| outline | 设置文本的描边颜色和描边宽度 | `<outline color=red width=4>A label with outline</outline>` | 如果你没有指定描边的颜色或者宽度的话，那么默认的颜色是白色(#ffffff),默认的宽度是 1 |
| b | 指定使用粗体来渲染 | `<b>This text will be rendered as bold</b>` | 名字必须是小写，且不能写成 bold |
| i | 指定使用斜体来渲染 | `<i>This text will be rendered as italic</i>` | 名字必须是小写，且不能写成 italic |
| u | 给文本添加下划线 | `<u>This text will have a underline</u>` | 名字必须是小写，且不能写成 underline |
| on | 指定一个点击事件处理函数，当点击该 Tag 所在文本内容时，会调用该事件响应函数 | `<on click="handler"> click me! </on>` | 除了 on 标签可以添加 click 属性，color 和 size 标签也可以添加，比如 `<size=10 click="handler2">click me</size>` |
| br | 插入一个空行 | `<br/>` | 注意： `<br></br>` 和 `<br>` 都是不支持的。 |
| img | 给富文本添加图文混排功能，img 的 src 属性必须是 ImageAtlas 图集里面的一个有效的 spriteframe 名称 | `<img src='emoji1' click='handler' />` | 注意: 只有 `<img src='foo' click='bar' />` 这种写法是有效的。如果你指定一张很大的图片，那么该图片创建出来的精灵会被等比缩放，缩放的值等于富文本的行高除以精灵的高度。 |

标签与标签是支持嵌套的，且嵌套规则跟 HTML 是一样的。比如下面的嵌套标签设置一个文本的渲染大小为 30，且颜色为绿色。

```
<size=30><color=green>I'm green></color></size>
```

也可以实现为:

```
<color=green><size=30>I'm green</size></color>
```

# 详细说明

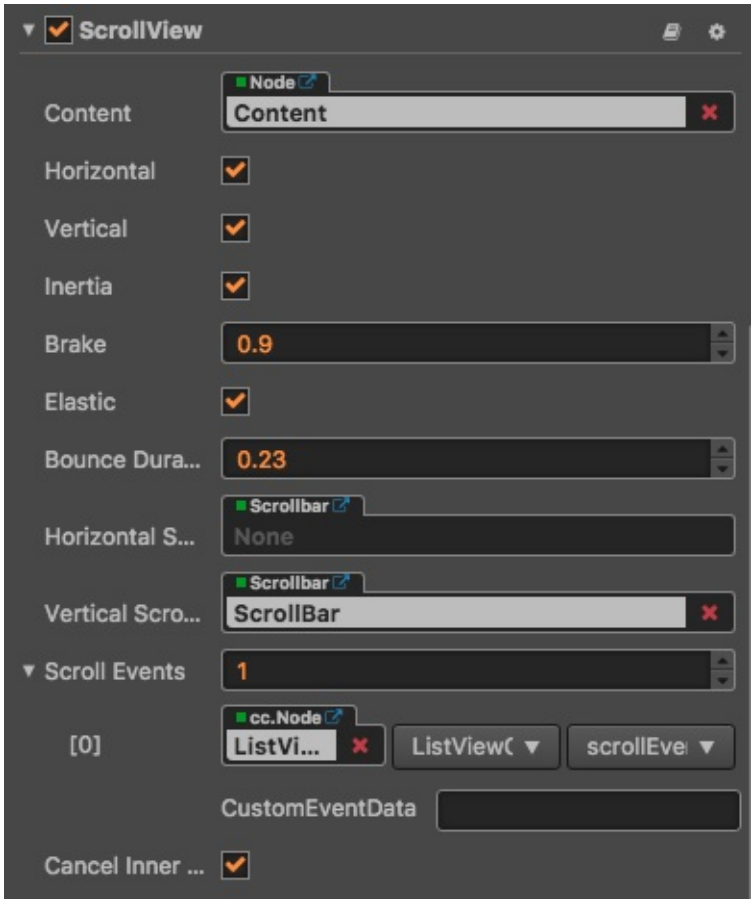
富文本组件全部由 JS 层实现，采用底层的 Label 节点拼装而成，并且在上层做排版逻辑。 这意味着，你新建一个复杂的富文本，底层可能有十几个 label 节点，而这些 label 节点都是采用系统字体渲染的，

所以，一般情况下，你不应该在游戏的主循环里面频繁地修改富文本的文本内容, 这可能会导致性能比较低。

另外，如果能不使用富文本组件，就尽量使用普通的文本组件，并且 BMFont 的效率是最高的。

# ScrollView 组件参考

ScrollView 是一种带滚动功能的容器，它提供一种方式可以在有限的显示区域内浏览更多的内容。通常 ScrollView 会与 `Mask` 组件配合使用，同时也可以添加 `ScrollBar` 组件来显示浏览内容的位置。




点击**属性检查器**下面的 `添加组件` 按钮，然后从 `添加 UI 组件` 中选择 `ScrollView`，即可添加 ScrollView 组件到节点上。

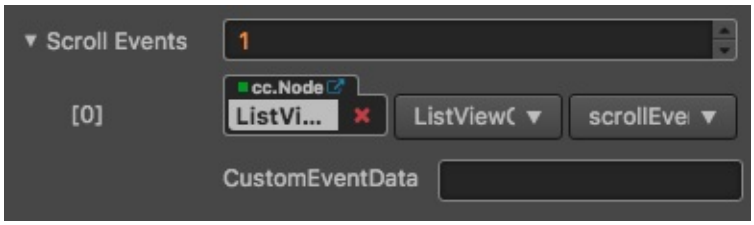
滚动视图的脚本接口请参考ScrollView API。

# ScrollView 属性

| 属性 | 功能说明 |
| --- | --- |
| content | 它是一个节点引用，用来创建 ScrollView 的可滚动内容，通常这可能是一个包含一张巨大图片的节点。 |
| | |

| | |
|---|---|
| Horizontal | 布尔值，是否允许横向滚动。 |
| Vertical | 布尔值，是否允许纵向滚动。 |
| Inertia | 滚动的时候是否有加速度。 |
| Brake | 浮点数，滚动之后的减速系数。取值范围是 0-1，如果是 1 则立马停止滚动，如果是 0，则会一直滚动到 content 的边界。 |
| Elastic | 布尔值，是否回弹。 |
| Bounce Duration | 浮点数，回弹所需要的时间。取值范围是 0-10。 |
| Horizontal ScrollBar | 它是一个节点引用，用来创建一个滚动条来显示 content 在水平方向上的位置。 |
| Vertical ScrollBar | 它是一个节点引用，用来创建一个滚动条来显示 content 在垂直方向上的位置 |
| ScrollView Events | 列表类型，默认为空，用户添加的每一个事件由节点引用，组件名称和一个响应函数组成。详情见 'Scrollview 事件' 章节 |
| CancelInnerEvents | 如果这个属性被设置为 true，那么滚动行为会取消子节点上注册的触摸事件，默认被设置为 true。 |

# ScrollView 事件




| 属性 | 功能说明 |
|---|---|
| Target | 带有脚本组件的节点。 |
| Component | 脚本组件名称。 |
| Handler | 指定一个回调函数，当 ScrollView 的事件发生的时候会调用此函数。 |
| CustomEventData | 用户指定任意的字符串作为事件回调的最后一个参数传入。 |

Scrollview 的事件回调有两个参数，第一个参数是 ScrollView 本身，第二个参数是 ScrollView 的事件类型。

# 详细说明

ScrollView 组件必须有指定的 content 节点才能起作用，通过指定滚动方向和 content 节点在此方向上的长度来计算滚动时的位置信息，Content 节点也可以通过 `UIWidget` 设置自动 resize。

通常一个 ScrollView 的节点树如下图：

这里的 Viewport 用来定义一个可以显示的滚动区域，所以通常 Mask 组件会被添加到 Viewport 上。可以滚动的内容可以直接放到 content 节点或者添加节 content 的子节点上。

## ScrollBar 设置

ScrollBar 是可选的，你可以选择只设置水平或者垂直 ScrollBar，当然也可以两者都设置。

建立关联可以通过在**层级管理器**里面拖拽一个带有 ScrollBar 组件的节点到 ScrollView 的相应字段完成。

## 通过脚本代码添加回调

### 方法一

这种方法添加的事件回调和使用编辑器添加的事件回调是一样的，通过代码添加， 你需要首先构造一个 `cc.Component.EventHandler` 对象，然后设置好对应的 target, component, handler 和 customEventData 参数。

```
//here is your component file, file name = MyComponent.js
cc.Class({
    extends: cc.Component,
    properties: {},

    onLoad: function () {
        var scrollViewEventHandler = new cc.Component.EventHandler();
        scrollViewEventHandler.target = this.node; //这个 node 节点是你的事件处理代码组件所属的节点
        scrollViewEventHandler.component = "MyComponent";//这个是代码文件名
        scrollViewEventHandler.handler = "callback";
        scrollViewEventHandler.customEventData = "foobar";

        var scrollview = node.getComponent(cc.ScrollView);
        scrollview.scrollEvents.push(scrollViewEventHandler);
    },

    //注意参数的顺序和类型是固定的
    callback: function (scrollview, eventType, customEventData) {
        //这里 scrollview 是一个 Scrollview 组件对象实例
        //这里的 eventType === cc.ScrollView.EventType enum 里面的值
        //这里的 customEventData 参数就等于你之前设置的 "foobar"
    }
});
```

### 方法二

通过 `scrollview.node.on('scroll-to-top', ...)` 的方式来添加

```
//假设我们在一个组件的 onLoad 方法里面添加事件处理回调，在 callback 函数中进行事件处理：

cc.Class({
    extends: cc.Component,


    properties: {
       scrollview: cc.ScrollView
    },

    onLoad: function () {
       this.scrollview.node.on('scroll-to-top', this.callback, this);
    },

    callback: function (event) {
       //这里的 event 是一个 EventCustom 对象，你可以通过 event.detail 获取 ScrollView 组件
       var scrollview = event.detail;
```

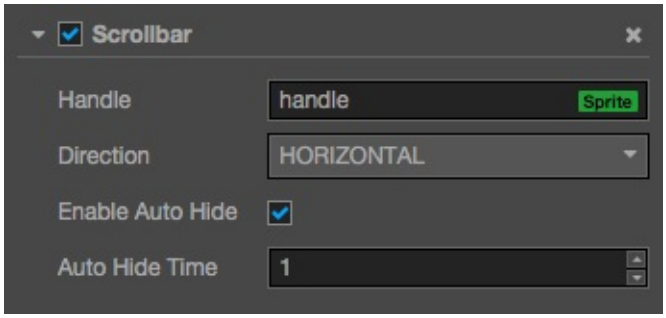
```
        //do whatever you want with scrollview
        //另外，注意这种方式注册的事件，也无法传递 customEventData
    }
});
```

同样的，你也可以注册 'scrolling', 'touch-up' , 'scrolling' 等事件，这些事件的回调函数的参数与 'scroll-to-top' 的参数一致。

关于完整的 ScrollView 的事件列表，可以参考 ScrollView 的 API 文档。

# ScrollBar 组件参考

ScrollBar 允许用户通过拖动滑块来滚动一张图片（公测版本暂不支持），它与 `Slider` 组件有点类似，但是它主要是用于滚动而 Slider 则用来设置数值。




点击**属性检查器**下面的 `添加组件` 按钮，然后从 `添加 UI 组件` 中选择 `ScrollBar`，即可添加 ScrollBar 组件到节点上。

滚动条的脚本接口请参考ScrollBar API。

# ScrollBar 属性

| 属性 | 功能说明 |
| --- | --- |
| Handle | ScrollBar 前景图片，它的长度/宽度会根据 ScrollView 的 content 的大小和实际显示区域的大小来计算。 |
| Direction | 滚动方向，目前包含水平和垂直两个方向。 |
| Enable Auto Hide | 是否开启自动隐藏，如果开启了，那么在 ScrollBar 显示后的 `Auto Hide Time` 时间内会自动消失。 |
| Auto Hide Time | 自动隐藏时间，需要配合设置 `Enable Auto Hide` |

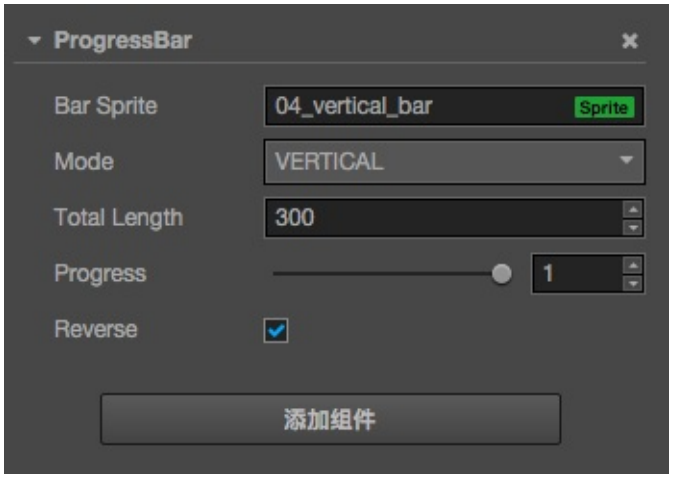
# 详细说明

ScrollBar 一般不会单独使用，它需要与 `ScrollView` 配合使用，另外 ScrollBar 需要指定一个 `Sprite` 组件，即属性面板里面的 `Handle` 。

通常我们还会给 ScrollBar 指定一张背景图片，用来指示整个 ScrollBar的长度或者度宽。

# ProgressBar 组件参考

ProgressBar（进度条）经常被用于在游戏中显示某个操作的进度，在节点上添加 ProgressBar 组件，然后给该组件关联一个 Bar Sprite 就可以在场景中控制 Bar Sprite 来显示进度了。




点击**属性检查器**下面的 `添加组件` 按钮，然后从 `添加 UI 组件` 中选择 `ProgressBar`，即可添加 ProgressBar 组件到节点上。

进度条的脚本接口请参考ProgressBar API。

# ProgressBar 属性

| 属性 | 功能说明 |
| --- | --- |
| Bar Sprite | 进度条渲染所需要的 Sprite 组件，可以通过拖拽一个带有 `Sprite` 组件的节点到该属性上来建立关联。 |
| Mode | 支持 `HORIZONTAL`（水平）、`VERTICAL`（垂直）和 `FILLED`（填充）三种模式，可以通过配合 `reverse` 属性来改变起始方向。 |
| Total Length | 当进度条为 100%时 Bar Sprite 的总长度/总宽度。在 `FILLED` 模式下 `Total Length` 表示取 Bar Sprite 总显示范围的百分比，取值范围从 0 ~ 1。 |
| Progress | 浮点，取值范围是 0~1，不允许输入之外的数值。 |
| Reverse | 布尔值，默认的填充方向是从左至右/从下到上，开启后变成从右到左/从上到下。 |

# 详细说明

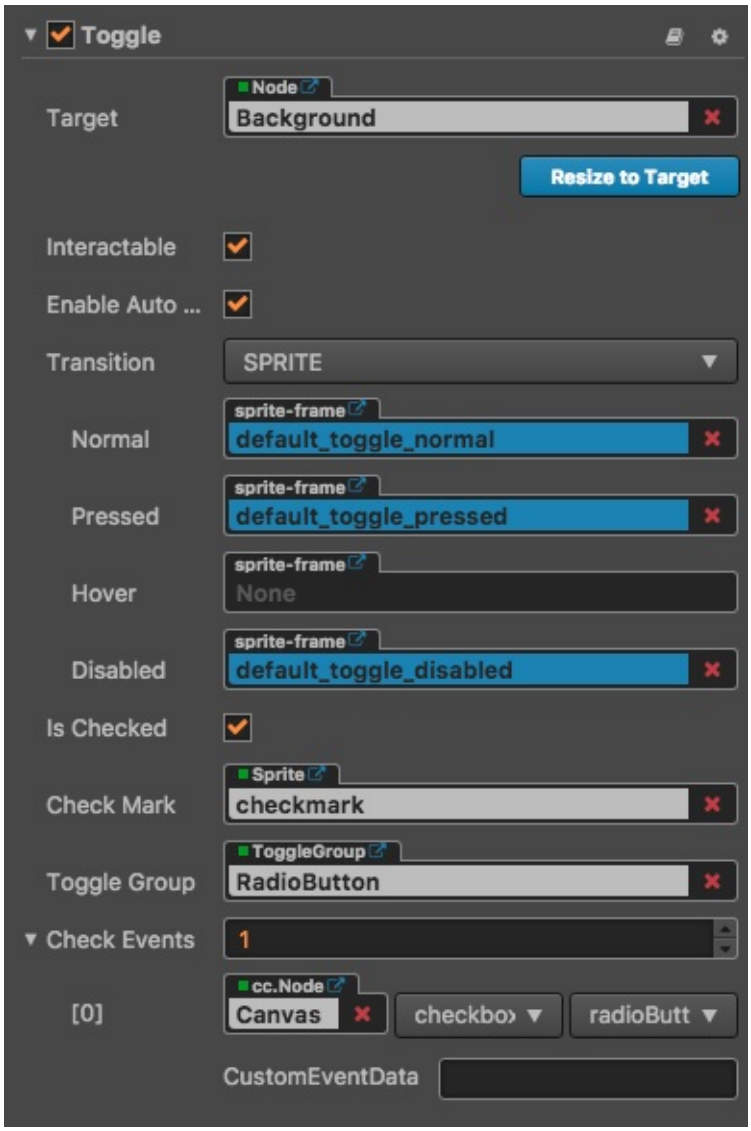
添加 ProgressBar 组件之后，通过从**层级管理器**中拖拽一个带有 `Sprite` 组件的节点到 Bar Sprite 属性上，此时便可以通过拖动 progress 滑块来控制进度条的显示了。

Bar Sprite 可以是自身节点，子节点，或者任何一个带有 `Sprite` 组件的节点。另外，Bar Sprite 可以自由选择 Simple、Sliced 和 Filled 渲染模式。

进度条的模式选择 `FILLED` 的情况下，Bar Sprite 的 `Type` 也需要设置为 `FILLED`，否则会报警告。详细使用说明请查阅 ProgressBar UI 控件介绍。

# Toggle 组件参考

Toggle 是一个 CheckBox，当它和 ToggleGroup 一起使用的时候，可以变成 RadioButton。




点击 **属性检查器** 下面的 `添加组件` 按钮，然后从 `添加 UI 组件` 中选择 `Toggle`，即可添加 Toggle 组件到节点上。

Toggle 的脚本接口请参考Toggle API。

# Toggle 属性

| 属性 | 功能说明 |
| --- | --- |
| isChecked | 布尔类型，如果这个设置为 true，则 check mark 组件会处于 enabled 状态，否则处于 disabled 状态。 |
| checkMark | cc.Sprite 类型，Toggle 处于选中状态时显示的图片 |
| toggleGroup | cc.ToggleGroup 类型， Toggle 所属的 ToggleGroup，这个属性是可选的。如果这个属性为 null，则 Toggle 是一个 CheckBox，否则，Toggle 是一个 RadioButton。 |
| | |

| | |
|---|---|
| Check Events | 列表类型，默认为空，用户添加的每一个事件由节点引用，组件名称和一个响应函数组成。详情见 Toggle 事件 章节 |

注意：因为 Toggle 继承至 Button，所以关于 Toggle 的 Button 相关属性的详细说明和用法请参考 Button 组件对应的章节，这里就不再赘述了。

# Toggle 事件

| 属性 | 功能说明 |
|---|---|
| Target | 带有脚本组件的节点。 |
| Component | 脚本组件名称。 |
| Handler | 指定一个回调函数，当 Toggle 的事件发生的时候会调用此函数。 |
| CustomEventData | 用户指定任意的字符串作为事件回调的最后一个参数传入。 |

Toggle 的事件回调有二个参数，第一个参数是 Toggle 本身, 第二个参数是 customEventData。

# 详细说明

Toggle 组件的节点树一般为：


这里注意的是，checkMark 组件所在的节点需要放在 background 节点的上面。

## 通过脚本代码添加回调

## 方法一

这种方法添加的事件回调和使用编辑器添加的事件回调是一样的，通过代码添加， 你需要首先构造一个 `cc.Component.EventHandler` 对象，然后设置好对应的 target, component, handler 和 customEventData 参数。

```
var checkEventHandler = new cc.Component.EventHandler();
checkEventHandler.target = this.node; //这个 node 节点是你的事件处理代码组件所属的节点
checkEventHandler.component = "cc.MyComponent"
checkEventHandler.handler = "callback";
checkEventHandler.customEventData = "foobar";

toggle.checkEvents.push(checkEventHandler);

//here is your component file
cc.Class({
    name: 'cc.MyComponent'
    extends: cc.Component,

    properties: {
    },

    callback: function(toggle, customEventData) {
        //这里 toggle 是 事件发出的 Toggle 组件
```

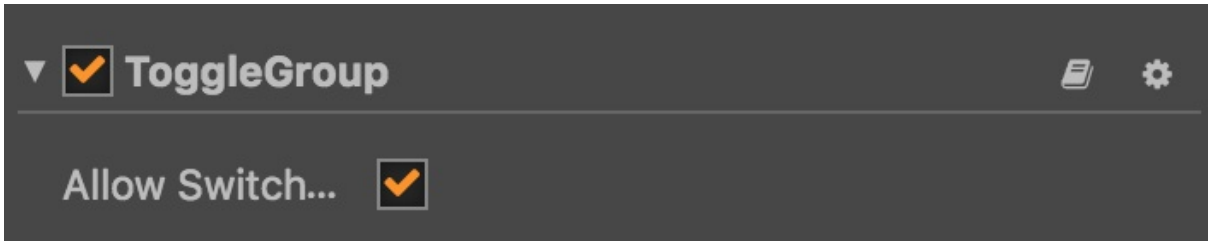
```
        //这里的 customEventData 参数就等于你之前设置的 "foobar"
    }
});
```

## 方法二

通过 `toggle.node.on('toggle', ...)` 的方式来添加

```
//假设我们在一个组件的 onLoad 方法里面添加事件处理回调，在 callback 函数中进行事件处理：

cc.Class({
    extends: cc.Component,


    properties: {
       toggle: cc.Toggle
    },

    onLoad: function () {
       this.toggle.node.on('toggle', this.callback, this);
    },

    callback: function (event) {
       //这里的 event 是一个 EventCustom 对象，你可以通过 event.detail 获取 Toggle 组件
       var toggle = event.detail;
       //do whatever you want with toggle
    }
});
```

# ToggleGroup 组件参考




ToggleGroup 不是一个可见的 UI 组件，它可以用来修改一组 Toggle 组件的行为。当一组 Toggle 属于同一个 ToggleGroup 的时候， 任何时候只能有一个 Toggle 处于选中状态。

点击**属性检查器**下面的 `添加组件` 按钮，然后从 `添加 UI 组件` 中选择 `ToggleGroup`，即可添加 ToggleGroup 组件到节点上。

ToggleGroup 的脚本接口请参考ToggleGroup API。

## ToggleGroup 属性

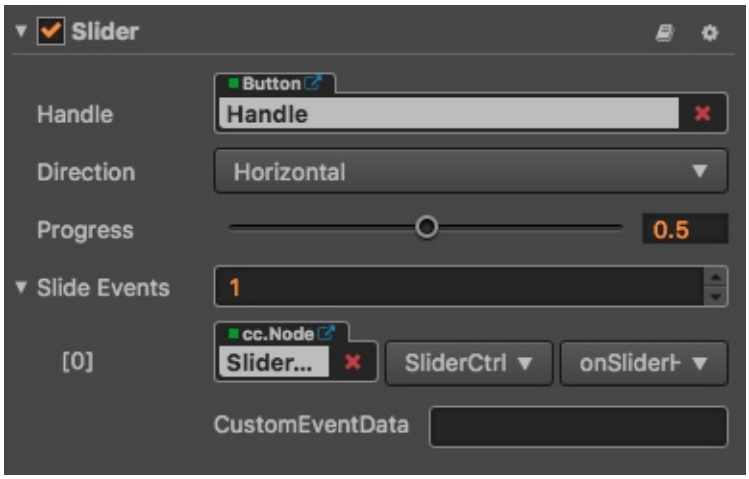
| 属性 | 功能说明 |
| --- | --- |
| allowSwitchOff | 如果这个设置为 true， 那么 toggle 按钮在被点击的时候可以反复地被选中和未选中。 |

## 详细说明

ToggleGroup 一般不会单独使用，它需要与 `Toggle` 配合使用来实现 RadioButton 的单选效果。

# Slider 组件参考

Slider 是一个滑动器组件。



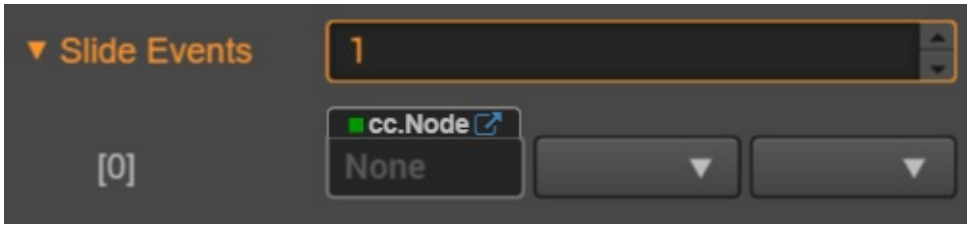

点击**属性检查器**下面的 `添加组件` 按钮，然后从 `添加 UI 组件` 中选择 `Slider`，即可添加 Slider 组件到节点上。

滑动器的脚本接口请参考Slider API。

## Slider 属性

| 属性 | 功能说明 |
| --- | --- |
| handle | 滑块按钮部件，可以通过该按钮进行滑动调节 Slider 数值大小 |
| direction | 滑动器的方向，分为横向和竖向 |
| progress | 当前进度值，该数值的区间是 0-1 之间 |
| slideEvents | 滑动器组件事件回调函数 |

## Slider 事件




| 属性 | 功能说明 |
| --- | --- |
| Target | 带有脚本组件的节点。 |
| Component | 脚本组件名称。 |

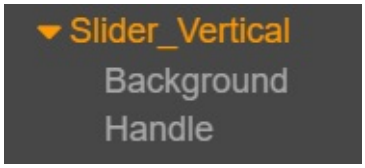
| | |
|---|---|
| Handler | 指定一个回调函数，当 Slider 的事件发生的时候会调用此函数。 |
| CustomEventData | 用户指定任意的字符串作为事件回调的最后一个参数传入。 |

Slider 的事件回调有两个参数，第一个参数是 Slider 本身，第二个参数是 CustomEventData

# 详细说明

Slider 通常用于调节数值的 UI（例如音量调节），它主要的部件一个滑块按钮，该部件用于用户交互，通过该部件可进行调节 Slider 的数值大小。

通常一个 Slider 的节点树如下图：




## 通过脚本代码添加回调

### 方法一

这种方法添加的事件回调和使用编辑器添加的事件回调是一样的，通过代码添加， 你需要首先构造一个 `cc.Component.EventHandler` 对象，然后设置好对应的 target, component, handler 和 customEventData 参数。

```
var sliderEventHandler = new cc.Component.EventHandler();
sliderEventHandler.target = this.node; //这个 node 节点是你的事件处理代码组件所属的节点
sliderEventHandler.component = "cc.MyComponent"
sliderEventHandler.handler = "callback";
sliderEventHandler.customEventData = "foobar";

slider.slideEvents.push(sliderEventHandler);

//here is your component file
cc.Class({
    name: 'cc.MyComponent'
    extends: cc.Component,

    properties: {
    },

    callback: function(slider, customEventData) {
        //这里 slider 是一个 cc.Slider 对象
        //这里的 customEventData 参数就等于你之前设置的 "foobar"
    }
});
```

### 方法二

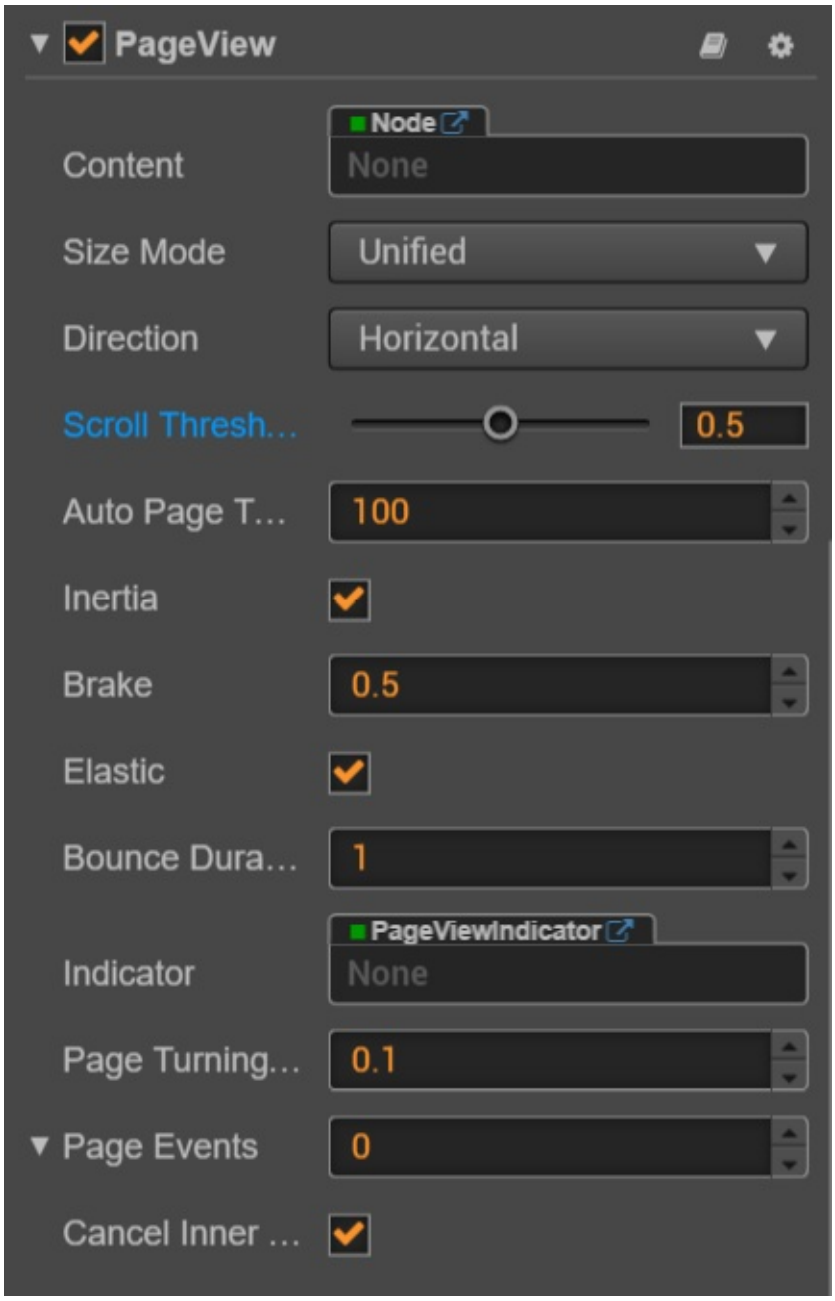
通过 `slider.node.on('slide', ...)` 的方式来添加

```
//假设我们在一个组件的 onLoad 方法里面添加事件处理回调，在 callback 函数中进行事件处理：

cc.Class({
    extends: cc.Component,


    properties: {
       slider: cc.Slider
    },
```

```
    onLoad: function () {
        this.slider.node.on('slide', this.callback, this);
    },

    callback: function (event) {
        //这里的 event 是一个 EventCustom 对象，你可以通过 event.detail 获取 Slider 组件
        var slider = event.detail;
        //do whatever you want with the slider
    }
});
```

# PageView 组件参考

PageView 是一种页面视图容器.




点击**属性检查器**下面的 `添加组件` 按钮，然后从 `添加 UI 组件` 中选择 `PageView`，即可添加 PageView 组件到节点上。

页面视图的脚本接口请参考 PageView API。

# PageView 属性

| 属性 | 功能说明 |
| --- | --- |
| SizeMode | 页面视图中每个页面大小类型，目前有 Unified 和 Free 类型 [SizeMove API](../api/enums/PageView.SizeMode.html) |

| | |
|---|---|
| Content | 它是一个节点引用，用来创建 PageView 的可滚动内容 |
| Direction | 页面视图滚动方向 |
| ScrollThreshold | 滚动临界值，默认单位百分比，当拖拽超出该数值时，松开会自动滚动下一页，小于时则还原 |
| AutoPageTurningThreshold | 快速滑动翻页临界值，当用户快速滑动时，会根据滑动开始和结束的距离与时间计算出一个速度值，该值与此临界值相比较，如果大于临界值，则进行自动翻页 |
| Inertia | 否开启滚动惯性 |
| Brake | 开启惯性后，在用户停止触摸后滚动多快停止，0表示永不停止，1表示立刻停止 |
| Elastic | 布尔值，是否回弹 |
| Bounce Duration | 浮点数，回弹所需要的时间。取值范围是 0-10 |
| Indicator | 页面视图指示器组件 |
| PageTurningEventTiming | 设置 PageView PageTurning 事件的发送时机 |
| PageEvents | 数组，滚动视图的事件回调函数 |
| CancelInnerEvents | 布尔值，是否在滚动行为时取消子节点上注册的触摸事件 |

# PageView 事件

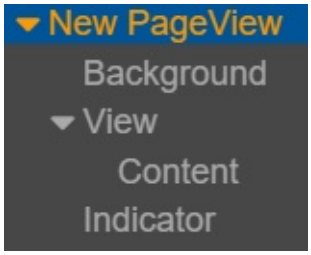

| 属性 | 功能说明 |
|---|---|
| Target | 带有脚本组件的节点 |
| Component | 脚本组件名称 |
| Handler | 指定一个回调函数，当 PageView 的事件发生的时候会调用此函数 |
| CustomEventData | 用户指定任意的字符串作为事件回调的最后一个参数传入 |

PageView 的事件回调有两个参数，第一个参数是 PageView 本身，第二个参数是 PageView 的事件类型。

# 详细说明

PageView 组件必须有指定的 content 节点才能起作用，content 中的每个子节点为一个单独页面，该每个页面的大小为 PageView 节点的大小，操作效果分为 2 种：第一种：缓慢滑动，通过拖拽视图中的页面到达指定的 ScrollThreshold 数值（该数值是页面大小的百分比）以后松开会自动滑动到下一页，第二种：快速滑动，快速的向一个方向进行拖动，自动滑倒下一页，每次滑动最多只能一页。

通常一个 PageView 的节点树如下图：




# CCPageViewIndicator 设置

CCPageViewIndicator 是可选的，该组件是用来显示页面的个数和标记当前显示在哪一页。

建立关联可以通过在**层级管理器**里面拖拽一个带有 PageViewIndicator 组件的节点到 PageView 的相应字段完成。

## 通过脚本代码添加回调

### 方法一

这种方法添加的事件回调和使用编辑器添加的事件回调是一样的，通过代码添加， 你需要首先构造一个 `cc.Component.EventHandler` 对象，然后设置好对应的 target, component, handler 和 customEventData 参数。

```
var pageViewEventHandler = new cc.Component.EventHandler();
pageViewEventHandler.target = this.node; // 这个是你的事件处理代码组件所属的节点
pageViewEventHandler.component = "cc.MyComponent"
pageViewEventHandler.handler = "callback";
pageViewEventHandler.customEventData = "foobar";

pageView.pageEvents.push(pageViewEventHandler);

//here is your component file
cc.Class({
    name: 'cc.MyComponent'

    extends: cc.Component,

    properties: {
    },

    // 注意参数的顺序和类型是固定的
    callback: function(pageView, eventType, customEventData) {
        // 这里 pageView 是一个 PageView 组件对象实例
        // 这里的 eventType === cc.PageView.EventType.PAGE_TURNING
        // 这里的 customEventData 参数就等于你之前设置的 "foobar"
    }
});
```

### 方法二

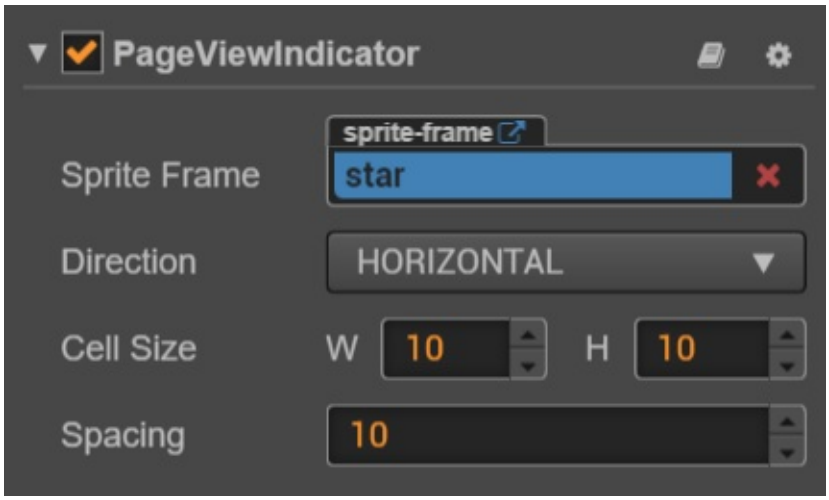
通过 `pageView.node.on('page-turning', ...)` 的方式来添加

```
// 假设我们在一个组件的 onLoad 方法里面添加事件处理回调，在 callback 函数中进行事件处理：

cc.Class({
    extends: cc.Component,

    properties: {
       pageView: cc.PageView
    },

    onLoad: function () {
       this.pageView.node.on('click', this.callback, this);
```

```
    },

    callback: function (event) {
        // 这里的 event 是一个 EventCustom 对象，你可以通过 event.detail 获取 PageView 组件
        var pageView = event.detail;
        // 另外，注意这种方式注册的事件，也无法传递 customEventData
    }
});
```

# PageviewIndicator 组件参考

PageviewIndicator 用于显示 PageView 当前的页面数量和标记当前所在的页面。




点击**属性检查器**下面的 `添加组件` 按钮，然后从 `添加 UI 组件` 中选择 `PageviewIndicator`，即可添加 PageviewIndicator 组件到节点上。

PageviewIndicator 的脚本接口请参考 PageviewIndicator API。

# PageviewIndicator 属性

| 属性 | 功能说明 |
| --- | --- |
| spriteFrame | 每个页面标记显示的图片 |
| direction | 页面标记摆放方向，分别为 水平方向 和 垂直方向 |
| cellSize | 每个页面标记的大小 |
| spacing | 每个页面标记之间的边距 |

# 详细说明

PageviewIndicator 一般不会单独使用，它需要与 `PageView` 配合使用，可以通过相关属性，来进行创建相对应页面的数量的标记，当你滑动到某个页面的时，PageviewIndicator 就会高亮它对应的标记。

# BlockInputEvents 组件参考

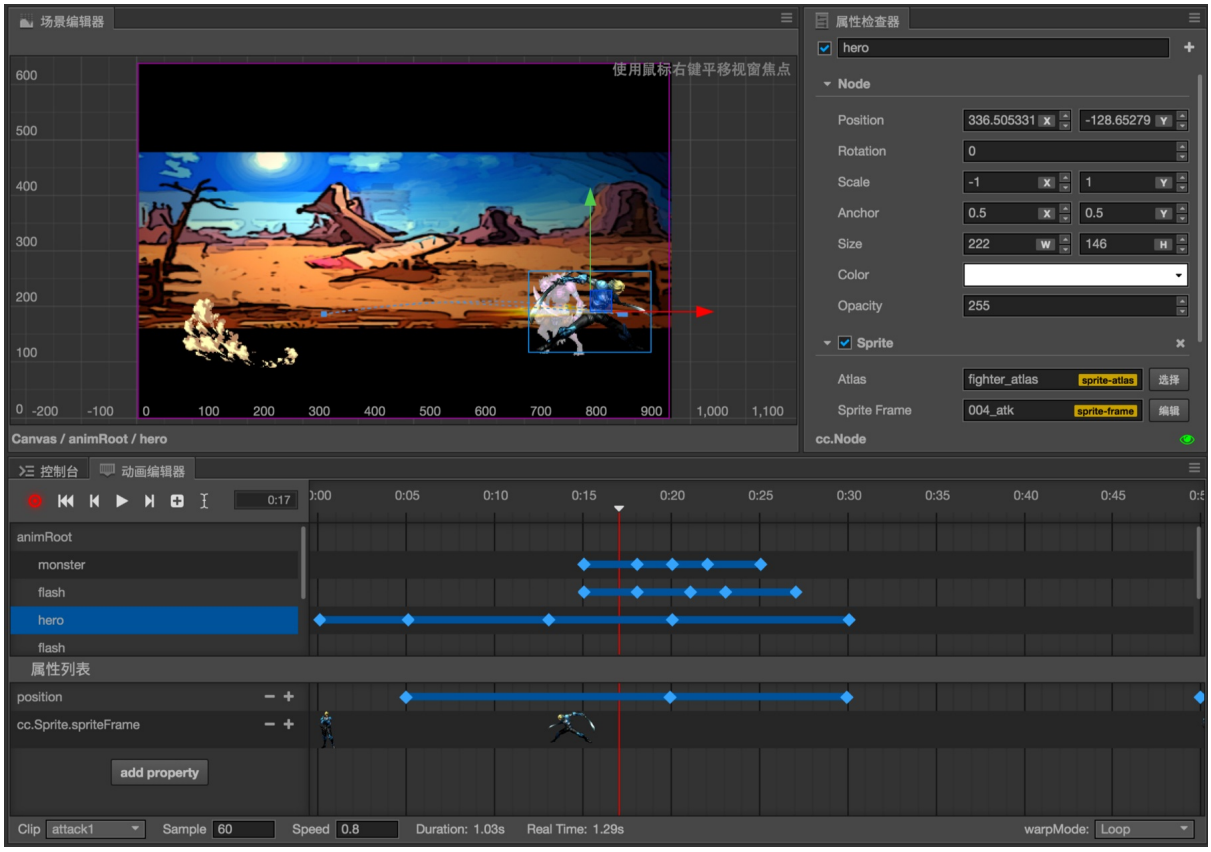
BlockInputEvents 组件将拦截所属节点 bounding box 内的所有输入事件（鼠标和触摸），防止输入穿透到下层节点，一般用于上层 UI 的背景。

当我们制作一个弹出式的 UI 对话框时，对话框的背景默认不会截获事件。也就是说虽然它的背景挡住了游戏场景，但是在背景上点击或触摸时，下面被遮住的游戏元素仍然会响应点击事件。这时我们只要在背景所在的节点上添加这个组件，就能避免这种情况。

该组件没有任何 API 接口，直接添加到场景即可生效。

# 动画系统

本章将介绍 Cocos Creator 的动画系统，除了标准的位移、旋转、缩放动画和序列帧动画以外，这套动画系统还支持任意组件属性和用户自定义属性的驱动，再加上可任意编辑的时间曲线和创新的移动轨迹编辑功能，能够让内容生产人员不写一行代码就制作出复杂而细腻的各种动态效果。




- 关于 Animation
- 创建 Animation 组件和动画剪辑
- 编辑动画曲线
- 编辑序列帧动画
- 编辑时间曲线
- 添加动画事件
- 使用脚本控制动画

从关于 Animation开始了解。

# 关于 Animation

## Animation 组件

之前我们了解了 Cocos Creator 是组件式的结构。那么 Animation 也不例外，它也是节点上的一个组件。

## Clip 动画剪辑

动画剪辑就是一份动画的声明数据，我们将它挂载到 Animation 组件上，就能够将这份动画数据应用到节点上。

### 节点数据的索引方式

数据中索引节点的方式是以挂载 Animation 组件的节点为根节点的相对路径。 所以在同个父节点下的同名节点，只能够产生一份动画数据，并且只能应用到第一个同名节点上。

### clip 文件的参数

**sample**： 定义当前动画数据每秒的帧率，默认为 60，这个参数会影响时间轴上每两个整数秒刻度之间的帧数量（也就是两秒之内有多少格）。

**speed**： 当前动画的播放速度，默认为 1

**duration**： 当动画播放速度为 1 的时候，动画的持续时间

**real time**： 动画从开始播放到结束，真正持续的时间

**wrap mode**： 循环模式

## 动画编辑模式

动画在普通模式下是不允许编辑的，只有在动画编辑模式下，才能够编辑动画文件。但是在编辑模式下，无法对节点进行 增加 / 删除 / 改名 操作。

打开编辑模式：

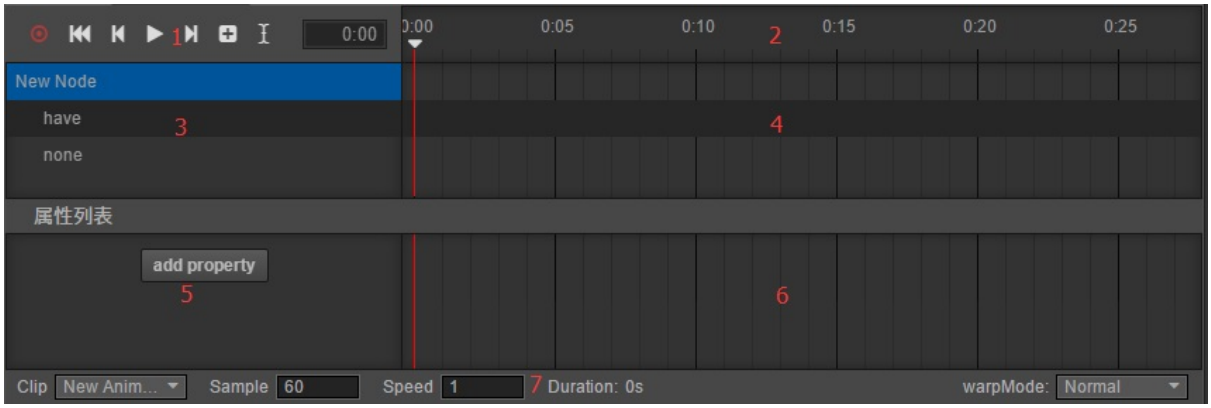
- 选中一个包含 Animation 组件，并且包含有一个以上 clip 文件的节点。然后在动画编辑器左上角点击唯一的按钮。

退出编辑模式：

- 点击动画编辑器上点击左上角的编辑按钮，或者在场景编辑器左上角的关闭按钮

## 熟悉动画编辑器

动画编辑器一共可以划分为7个部分。




1. 常用按钮区域，这里负责显示一些常用功能按钮，从左到右依次为：开关录制状态、返回第一帧、上一帧、播放/暂停、下一帧、新建动画剪辑、插入动画事件
2. 时间轴与事件，这里主要是显示时间轴，添加的自定义事件也会在这里显示。
3. 层级管理（节点树），当前动画剪辑可以影响到的节点数据。
4. 节点内动画帧的预览区域，这里主要是显示各个节点上的所有帧的预览时间轴。
5. 属性列表，显示当前选中的节点在选中的动画剪辑中已经包含了的属性列表。
6. 关键帧，每个属性相对应的帧都会显示在这里
7. 动画剪辑的基本属性，选择动画剪辑后，基本数据都在这里显示以及更改。

### 时间轴的刻度单位表示方式

时间轴上刻度的表示法是 `1:05` 。该数值由两部分组成，冒号前面的是表示当前秒数，冒号后面的表示在当前这一秒里的第几帧。

`1:05` 表示该刻度在时间轴上位于从动画开始经过了 1 秒又 5 帧 的时间。

因为动画帧率（sample）可以随时调整，因此同一个刻度表示的时间点也会随着帧率变化而有所不同。

- 当帧率为 30 时， `1:05` 表示动画开始后 1 + 5/30 = 1.1667 秒。
- 当帧率为 10 时， `1:05` 表示动画开始后 1 + 5/10 = 1.5 秒。

虽然当前刻度表示的时间点会随着帧率变化，但一旦在一个位置添加了关键帧，该关键帧所在的总帧数是不会改变的，假如我们在帧率 30 时向 `1:05` 刻度上添加了关键帧，该关键帧位于动画开始后总第 35 帧。之后把帧率修改为 10，该关键帧仍然处在动画开始后第 35 帧，而此时关键帧所在位置的刻度读数为 `3:05` 。换算成时间以后正好是之前的 3 倍。

## 基本操作

### 更改时间轴缩放比例

在操作中如果觉得动画编辑器显示的范围太小，需要按比例缩小，让更多的关键帧显示到编辑器内怎么办?

- 在图中2、4、6区域内滚动鼠标滚轮，可以放大，或者缩小时间轴的显示比例。

### 移动显示区域

如果想看动画编辑器右侧超出编辑器被隐藏的关键帧或是左侧被隐藏的关键帧，这时候就需要移动显示区域：

- 按住键盘 `空格` 按键，并且在图中2、4、6区域内按下鼠标左键，向左右拖拽即可。

- 在图中2、4、6区域内按下鼠标中键拖拽。

### 更改当前选中的时间轴节点

- 在时间轴（图中2）区域内点击任意位置或者拖拽，都可以更改当前的时间节点。
- 在图中4区域内拖拽标示的红线即可。

### 播放/暂停动画

- 在图示1中点击播放按钮，按钮会自动变更为暂停，再次点击则是暂停。
- 播放状态下，保存场景等操作会终止播放。

### 修改 clip 属性

- 在图示7区域，修改对应的属性，在输入框失去焦点的时候就会更新到实际的 clip 数据中。

## 快捷键

- left：向前移动一帧，如果已经在第 0 帧，则忽略当前操作
- right：向后移动一帧
- up：跳转到上一个关键帧
- down：跳转到下一个关键帧
- delete：删除当前所选中的关键帧
- k：正向的播放动画，抬起后停止
- j：反向播放动画，抬起后停止
- ctrl / cmd + left：跳转到第 0 帧
- ctrl / cmd + right：跳转到有效的最后一帧

# 创建Animation组件和动画剪辑

## 创建 Animation 组件

在每个节点上，我们都可以添加不同的组件。如果我们想在这个节点上创建动画，也必须为它新建一个 Animation 组件。 创建的方法有两种：

- 选中相应的节点，在属性检查器中点击右上方的 `+`，或者下方的 `添加组件`，在其他组件中选择 Animation。
- 打开动画编辑器，然后在层级管理器中选中需要添加动画的节点，在动画编辑器中点击 `添加 Animation 组件` 按钮。


## 创建与挂载动画剪辑

现在我们的节点上已经有了 Animation 组件了，但是还没有相应的动画剪辑数据，动画剪辑也有两种创建方式：

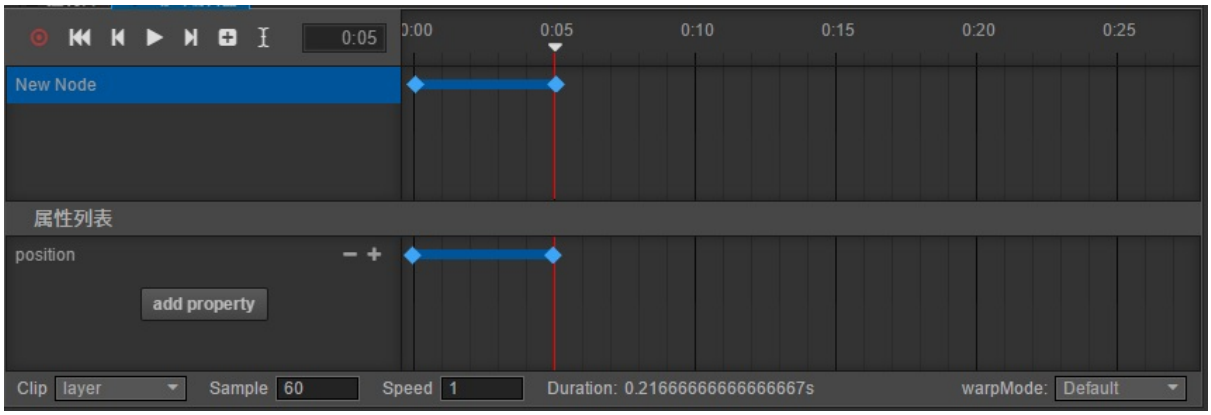
- 在资源管理器中点击左上方的 `+`，或者右键空白区域，选择 Animation Clip，这时候会在管理器中创建一个名为 'New AnimationClip' 的剪辑文件。 单单创建还是不够的，我们再次在层级管理器中点选刚刚的节点，在属性检查器中找到 Animation，这时候的 Clips 显示的是 0，我们将它改成 1。 然后将刚刚在资源管理器中创建的 'New AnimationClip'，拖入刚刚出现的 `animation-clip选择框` 内。
- 如果 Animation 组件中还没有添加动画剪辑文件，则可以在动画编辑器中直接点击 `新建 AnimationClip` 按钮，根据弹出的窗口创建一个新的动画剪辑文件。 需要注意的是，如果选择覆盖已有的剪辑文件，被覆盖的文件内容会被清空。


至此我们已经完成了动画制作之前的准备工作，下一步就是要创建动画曲线了。

# 编辑动画序列

我们刚刚已经在节点上挂载了动画剪辑，现在我们可以在动画剪辑中创建一些动画曲线了。

我们首先了解一下动画属性，动画属性包括了节点自有的 `position` , `rotation` 等属性，也包含了组件Component中自定义的属性。 组件包含的属性前会加上组件的名字，比如 `cc.Sprite.spriteFrame` 。 比如下图的 position 那条就是属性轨道，而对应的蓝色棱形就是动画帧。




# 添加一个新的属性轨道

常规的添加方式，我们需要先选中节点，然后在属性区域点击 `add property` 。 弹出菜单中，会将可以添加的所有属性罗列出来，选中想要添加的属性，就会对应新增一个轨道。

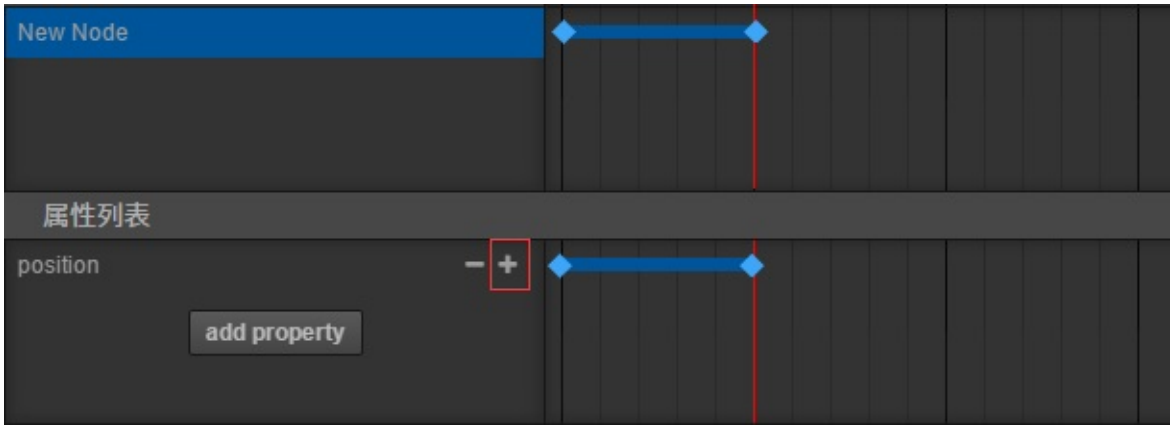
也可以在编辑模式下直接更改节点的对应轨道的属性 - 例如直接在场景编辑器中拖动当前选中的节点， `position` 轨道上就会在当前的时间上增加一个关键帧。 需要注意的是，如果更改的属性轨道不存在，则会忽略此次的操作，所以如果想要修改后自动插入关键帧，需要预先创建好属性轨道。

# 删除一个属性轨道

右键点击属性列表中的属性，在弹出菜单中选择 `delete` 选项，选中后对应的属性就会从动画数据中删除。

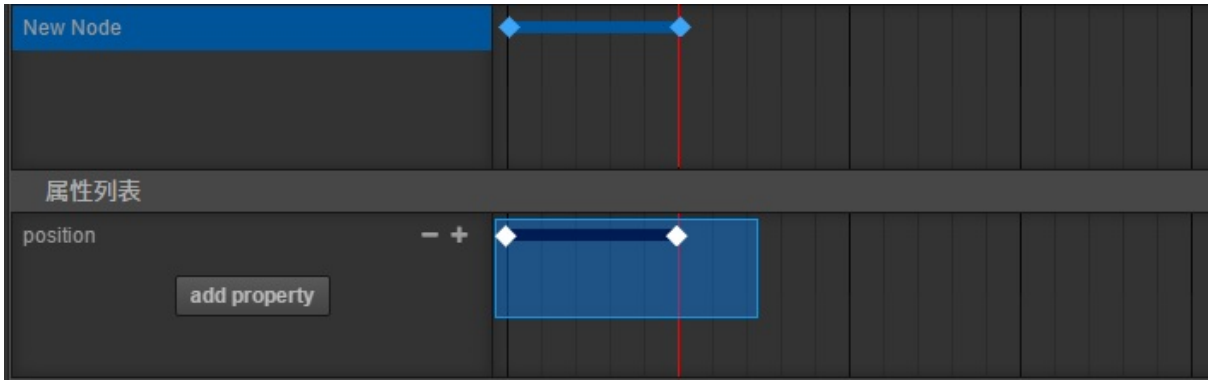
# 添加动画帧

刚刚我们说到在录制状态下直接更改对应属性可以自动添加对应的属性和帧。 也可以直接在属性列表中点击对应属性右侧的 `+` 号，这样会在当前选中的时间点上增加一帧。




## 选择动画帧

点击我们创建的序列帧后序列帧会显示成选中状态，此时关键帧由蓝变白，如果需要多选，可以按住ctrl再次选择其他序列帧。或者直接在属性区域拖拽框选。




## 移动动画帧

此时我们将鼠标移动到任意一个选中的关键帧（蓝色棱形）或序列帧（包括关键帧和关键帧之间的连线）上，鼠标会显示出左右箭头，这时候按下鼠标左键就可以拖拽所有被选中的节点了。

## 更改动画帧

点击需要修改的动画帧，此时时间轴上选中的帧也会跳到这一帧，然后确保打开了录制状态，直接在属性检查器内修改对应的属性即可。

## 删除动画帧

选中序列帧后，点击属性区域的 `-`，此时当前属性被选中的序列帧会被删除。或者直接按下键盘上的 delete 按键，则所有被选中的节点都会被删除。

# 编辑序列帧动画

我们刚刚了解了属性帧的操作，现在来看看具体怎么创建一个帧动画。

## 为节点新增Sprite组件

首先我们需要让节点正常显示纹理，所以需要为节点增加Sprite组件。 选中节点后在属性检查器中通过 **添加组件** 按钮，选择 `添加渲染组件->Sprite` 。

## 在属性列表中添加 cc.Sprite.spriteFrame

节点可以正常显示纹理后，还需要为纹理创建一个动画轨道。 在动画编辑器中点击 `add property` ，然后选择 `cc.Sprite.spriteFrame`

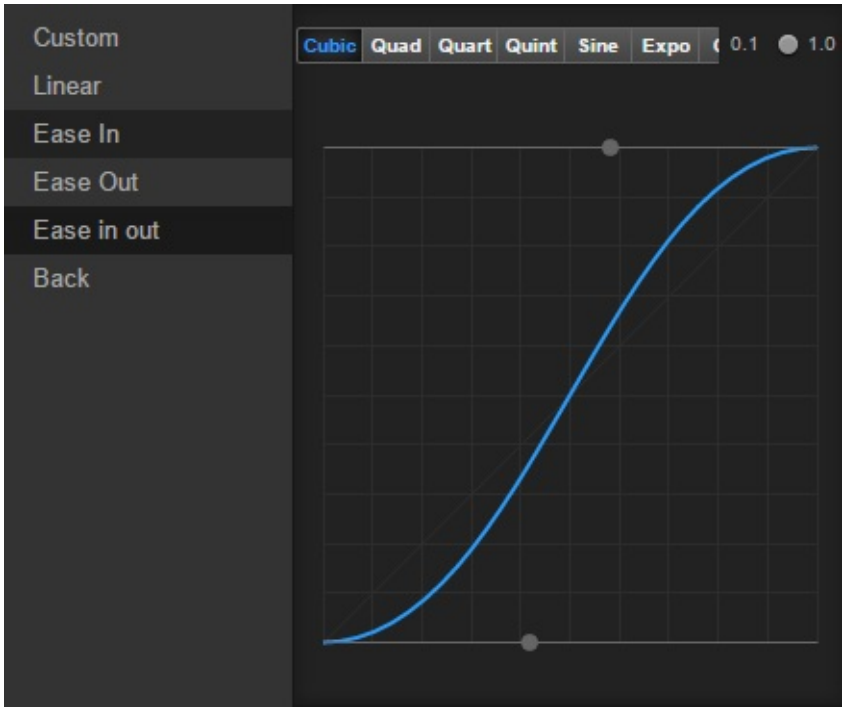
## 添加帧

从资源管理器中，将纹理拖拽到属性帧区域，放在 `cc.Sprite.spriteFrame` 轨道上。 再将下一帧需要显示的纹理拖到指定位置，然后点击播放就可以预览刚刚创建的动画了。

# 编辑时间曲线

我们已经创建了基本的动画了。 但有时候我们会需要在两帧之间实现EaseInOut等缓动效果，那么在动画编辑器中怎么实现呢?

我们首先需要在一条轨道上创建两个不相等的帧，比如在position上创建两帧，从 0,0 到 100,100。 这时候两帧之间会出现一根连接线（连接俩关键帧之间的蓝色线段），双击连接线，则可以打开时间曲线编辑器。




# 使用预设曲线

我们在曲线编辑器左侧可以选择预设的各种效果。比如说 Ease In 等。 选中后右侧上方还会出现一些预设的参数，可以根据需求选择。

# 自定义曲线

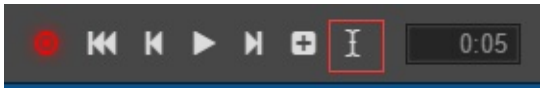
有时候预设的不能够满足动画需求，我们也可以自己修改曲线。 右侧下方预览图内，有两个灰色的控制点，拖拽控制点可以更改曲线的轨迹。 如果控制点需要拖出视野外，则可以使用鼠标滚轮或者右上角的小比例尺缩放预览图，支持的比例从 0.1 到 1。

# 添加动画事件

在游戏中，经常需要在动画结束或者某一帧的特定时刻，执行一些函数方法。那么在动画编辑器中怎么实现呢?

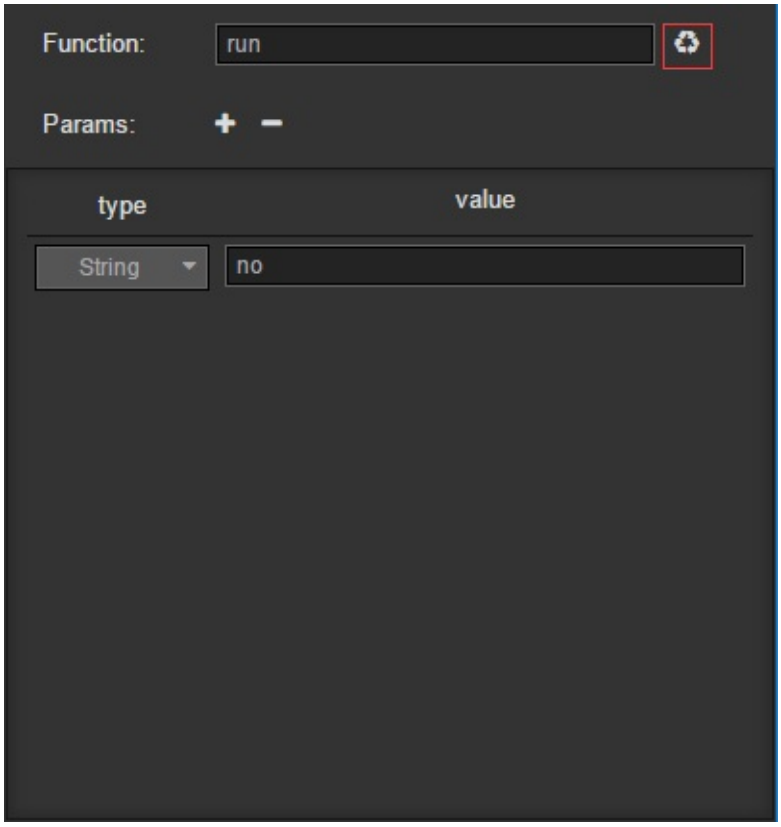
## 添加事件

首先选中某个位置，然后点击按钮区域最左侧的按钮（add event），这时候在时间轴上会出现一个白色的矩形，这就是我们添加的事件。




## 删除事件

双击刚刚出现的白色矩形，打开事件编辑器后点击 function 后面的回收图标，会提示是否删除这个 event，点击确认则删除。




也可以在动画编辑器中右键点击 event，选择delete。

## 指定事件触发函数以及传入参数

双击刚刚出现的白色矩形，可以打开事件编辑器，在编辑器内，我们可以手动输入需要出发的function名字，触发的时候会根据这个函数名，去各个组件内匹配相应的方法。

如果需要添加传入的参数，则在 Params 旁点击 `+` 或者 `-` ，只支持Boolean，String，Number三种类型的参数。

# 使用脚本控制动画

## Animation 组件

Animation 组件提供了一些常用的动画控制函数，如果只是需要简单的控制动画，可以通过获取节点的 Animation 组件来做一些操作。

### 播放

```
var anim = this.getComponent(cc.Animation);

// 如果没有指定播放哪个动画，并且有设置 defaultClip 的话，则会播放 defaultClip 动画
anim.play();

// 指定播放 test 动画
anim.play('test');

// 指定从 1s 开始播放 test 动画
anim.play('test', 1);

// 使用 play 接口播放一个动画时，如果还有其他的动画正在播放，则会先停止其他动画
anim.play('test2');
```

Animation 对一个动画进行播放的时候会判断这个动画之前的播放状态来进行下一步操作。 如果动画处于：

- **停止** 状态，则 Animation 会直接重新播放这个动画
- **暂停** 状态，则 Animation 会恢复动画的播放，并从当前时间继续播放下去
- **播放** 状态，则 Animation 会先停止这个动画，再重新播放动画

```
var anim = this.getComponent(cc.Animation);

// 播放第一个动画
anim.playAdditive('position-anim');

// 播放第二个动画
// 使用 playAdditive 播放动画时，不会停止其他动画的播放。如果还有其他动画正在播放，则同时会有多个动画进行播放
anim.playAdditive('rotation-anim');
```

Animation 是支持同时播放多个动画的，播放不同的动画并不会影响其他的动画的播放状态，这对于做一些复合动画比较有帮助。

### 暂停 恢复 停止

```
var anim = this.getComponent(cc.Animation);

anim.play('test');

// 指定暂停 test 动画
anim.pause('test');

// 暂停所有动画
// anim.pause();

// 指定恢复 test 动画
anim.resume('test');
```

```
// 恢复所有动画
// anim.resume();

// 指定停止 test 动画
anim.stop('test');

// 停止所有动画
// anim.stop();
```

**暂停**，**恢复**， **停止** 几个函数的调用比较接近。

**暂停** 会暂时停止动画的播放，当 **恢复** 动画的时候，动画会继续从当前时间往下播放。 而 **停止** 则会终止动画的播放，再对这个动画进行播放的时候会重新从开始播放动画。

### 设置动画的当前时间

```
var anim = this.getComponent(cc.Animation);

anim.play('test');

// 设置 test 动画的当前播放时间为 1s
anim.setCurrentTime(1, 'test');

// 设置所有动画的当前播放时间为 1s
// anim.setCurrentTime(1);
```

你可以在任何时候对动画设置当前时间，但是动画不会立刻根据设置的时间进行状态的更改，需要在下一个动画的 **update** 中才会根据这个时间重新计算播放状态。

## AnimationState

**Animation** 只提供了一些简单的控制函数，希望得到更多的动画信息和控制的话，需要使用到 **AnimationState**。

### AnimationState 是什么?

如果说 **AnimationClip** 作为动画数据的承载，那么 **AnimationState** 则是 **AnimationClip** 在运行时的实例，它将动画数据解析为方便程序中做计算的数值。 **Animation** 在播放一个 **AnimationClip** 的时候，会将 **AnimationClip** 解析成 **AnimationState**。 **Animation** 的播放状态实际都是由 **AnimationState** 来计算的，包括动画是否循环，怎么循环，播放速度 等。

### 获取 AnimationState

```
var anim = this.getComponent(cc.Animation);
// play 会返回关联的 AnimationState
var animState = anim.play('test');

// 或是直接获取
var animState = anim.getAnimationState('test');
```

### 获取动画信息

```
var anim = this.getComponent(cc.Animation);
var animState = anim.play('test');

// 获取动画关联的clip
var clip = animState.clip;
```

```
// 获取动画的名字
var name = animState.name;

// 获取动画的播放速度
var speed = animState.speed;

// 获取动画的播放总时长
var duration = animState.duration;

// 获取动画的播放时间
var time = animState.time;

// 获取动画的重复次数
var repeatCount = animState.repeatCount;

// 获取动画的循环模式
var wrapMode = animState.wrapMode

// 获取动画是否正在播放
var playing = animState.isPlaying;

// 获取动画是否已经暂停
var paused = animState.isPaused;

// 获取动画的帧率
var frameRate = animState.frameRate;
```

从 **AnimationState** 中可以获取到所有动画的信息，你可以利用这些信息来判断需要做哪些事情。

## 设置动画播放速度

```
var anim = this.getComponent(cc.Animation);
var animState = anim.play('test');

// 使动画播放速度加速
animState.speed = 2;

// 使动画播放速度减速
animState.speed = 0.5;
```

**speed** 值越大速度越快，值越小则速度越慢

## 设置动画 循环模式 与 循环次数

```
var anim = this.getComponent(cc.Animation);
var animState = anim.play('test');

// 设置循环模式为 Normal
animState.wrapMode = cc.WrapMode.Normal;

// 设置循环模式为 Loop
animState.wrapMode = cc.WrapMode.Loop;

// 设置动画循环次数为2次
animState.repeatCount = 2;

// 设置动画循环次数为无限次
animState.repeatCount = Infinity;
```

**AnimationState** 允许动态设置循环模式，目前提供了多种循环模式，这些循环模式可以从 **cc.WrapMode** 中获取到。如果动画的循环类型为 **Loop** 类型的话，需要与 **repeatCount** 配合使用才能达到效果。 默认在解析动画剪辑的时候，如果动画循环类型为：

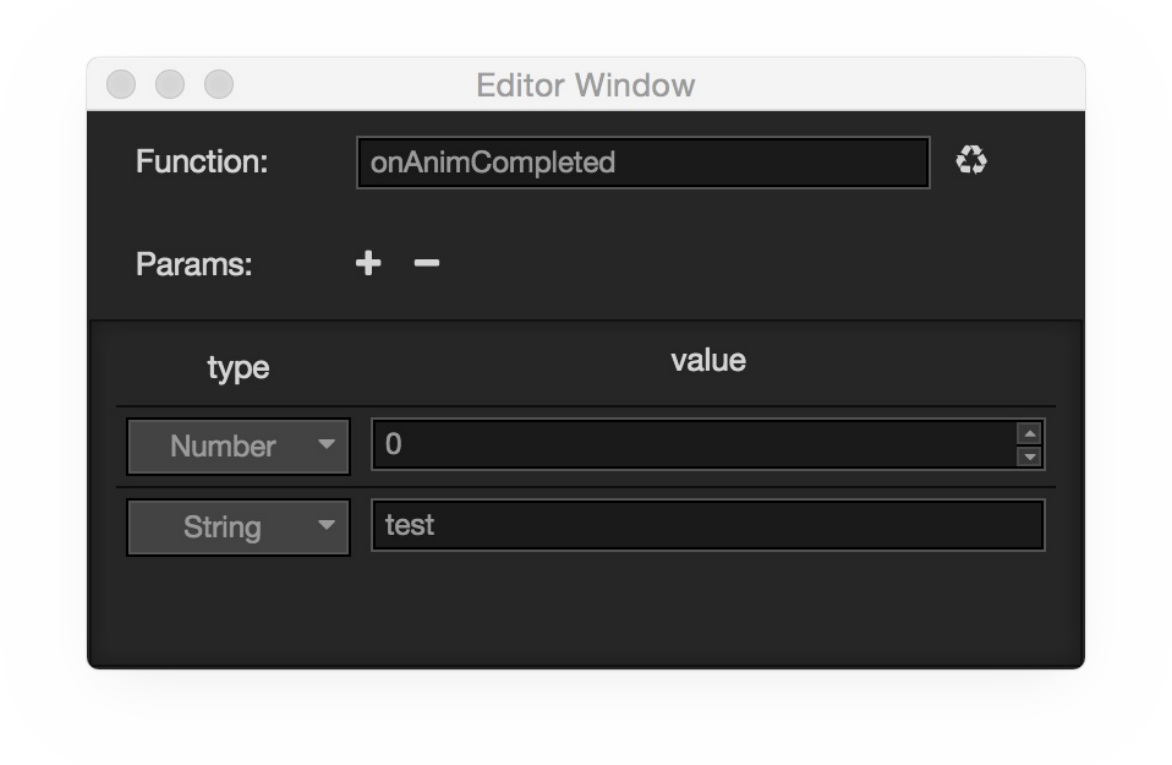
- **Loop** 类型，**repeatCount** 将被设置为 **Infinity**, 即无限循环
- **Normal** 类型，**repeatCount** 将被设置为 1

# 动画事件

在动画编辑器里支持可视化编辑帧事件 (如何编辑请参考 这里 )，在脚本里书写动画事件的回调也非常简单。 动画事件的回调其实就是一个普通的函数，在动画编辑器里添加的帧事件会映射到动画根节点的组件上。

## 实例：

假设在动画的结尾添加了一个帧事件，如下图：




那么在脚本中可以这么写：

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

    onAnimCompleted: function (num, string) {
        console.log('onAnimCompleted: param1[%s], param2[%s]', num, string);
    }
});
```

将上面的组件加到动画的 **根节点** 上，当动画播放到结尾时，动画系统会自动调用脚本中的 `onAnimCompleted` 函数。 动画系统会搜索动画根节点中的所有组件，如果组件中有实现动画事件中指定的函数的话，就会对它进行调用，并传入事件中填的参数。

# 注册动画回调

除了动画编辑器中的帧事件提供了回调外，动画系统还提供了动态注册回调事件的方式。
目前支持的回调事件有：

- play : 开始播放时
- stop : 停止播放时
- pause : 暂停播放时
- resume : 恢复播放时
- lastframe : 假如动画循环次数大于 1，当动画播放到最后一帧时
- finished : 动画播放完成时

当在 `cc.Animation` 注册了一个回调函数后，它会在播放一个动画时，对相应的 `cc.AnimationState` 注册这个回调，在 `cc.AnimationState` 停止播放时，对 `cc.AnimationState` 取消注册这个回调。

`cc.AnimationState` 其实才是动画回调的发送方，如果希望对单个 `cc.AnimationState` 注册回调的话，那么可以获取到这个 `cc.AnimationState` 再单独对它进行注册。

## 实例

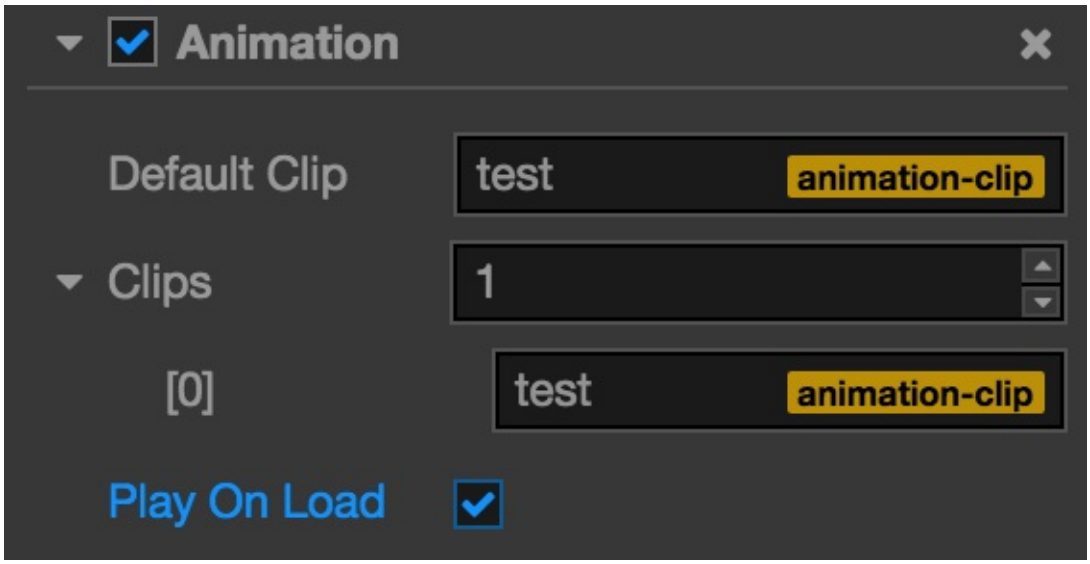
```
var animation = this.node.getComponent(cc.Animation);

// 注册
animation.on('play',      this.onPlay,        this);
animation.on('stop',      this.onStop,        this);
animation.on('lastframe', this.onLastFrame,   this);
animation.on('finished',  this.onFinished,    this);
animation.on('pause',     this.onPause,       this);
animation.on('resume',    this.onResume,      this);

// 取消注册
animation.off('play',      this.onPlay,        this);
animation.off('stop',      this.onStop,        this);
animation.off('lastframe', this.onLastFrame,   this);
animation.off('finished',  this.onFinished,    this);
animation.off('pause',     this.onPause,       this);
animation.off('resume',    this.onResume,      this);

// 对单个 cc.AnimationState 注册回调
var anim1 = animation.getAnimationState('anim1');
anim1.on('lastframe',    this.onLastFrame,      this);
```

# 动态创建 Animation Clip

```
    var animation = this.node.getComponent(cc.Animation);
    // frames 这是一个 SpriteFrame 的数组.
    var clip = cc.AnimationClip.createWithSpriteFrames(frames, 17);
    clip.name = "anim_run";
    clip.wrapMode = cc.WrapMode.Loop;

    // 添加帧事件
    clip.events.push({
        frame: 1,               // 准确的时间，以秒为单位。这里表示将在动画播放到 1s 时触发事件
        func: "frameEvent",     // 回调函数名称
        params: [1, "hello"]    // 回调参数
    });

    animation.addClip(clip);
    animation.play('anim_run');
```

# Animation（动画）组件参考

**Animation（动画）** 组件可以以动画方式驱动所在节点和子节点上的节点和组件属性，包括用户自定义脚本中的属性。




点击 **属性检查器** 下面的 `添加组件` 按钮，然后从 `添加其他组件` 中选择 `Animation`，即可添加 **Animation（动画）** 组件到节点上。

动画的脚本接口请参考Animation API。

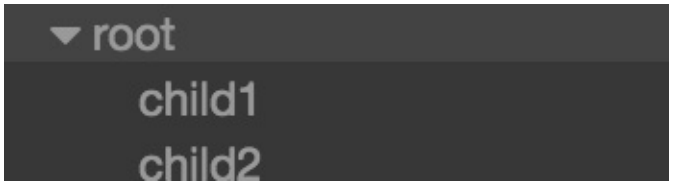
## Animation 属性

| 属性 | 功能说明 |
| --- | --- |
| Default Clip | 默认的动画剪辑，如果这一项设置了值，并且 **Play On Load** 也为true，那么动画会在加载完成后自动播放 **Default Clip** 的内容 |
| Clips | 列表类型，默认为空，在这里面添加的 **AnimationClip** 会反映到 **动画编辑器**里，用户可以在 **动画编辑器** 里编辑 **Clips** 的内容 |
| Play On Load | 布尔类型，是否在动画加载完成后自动播放 **Default Clip** 的内容 |

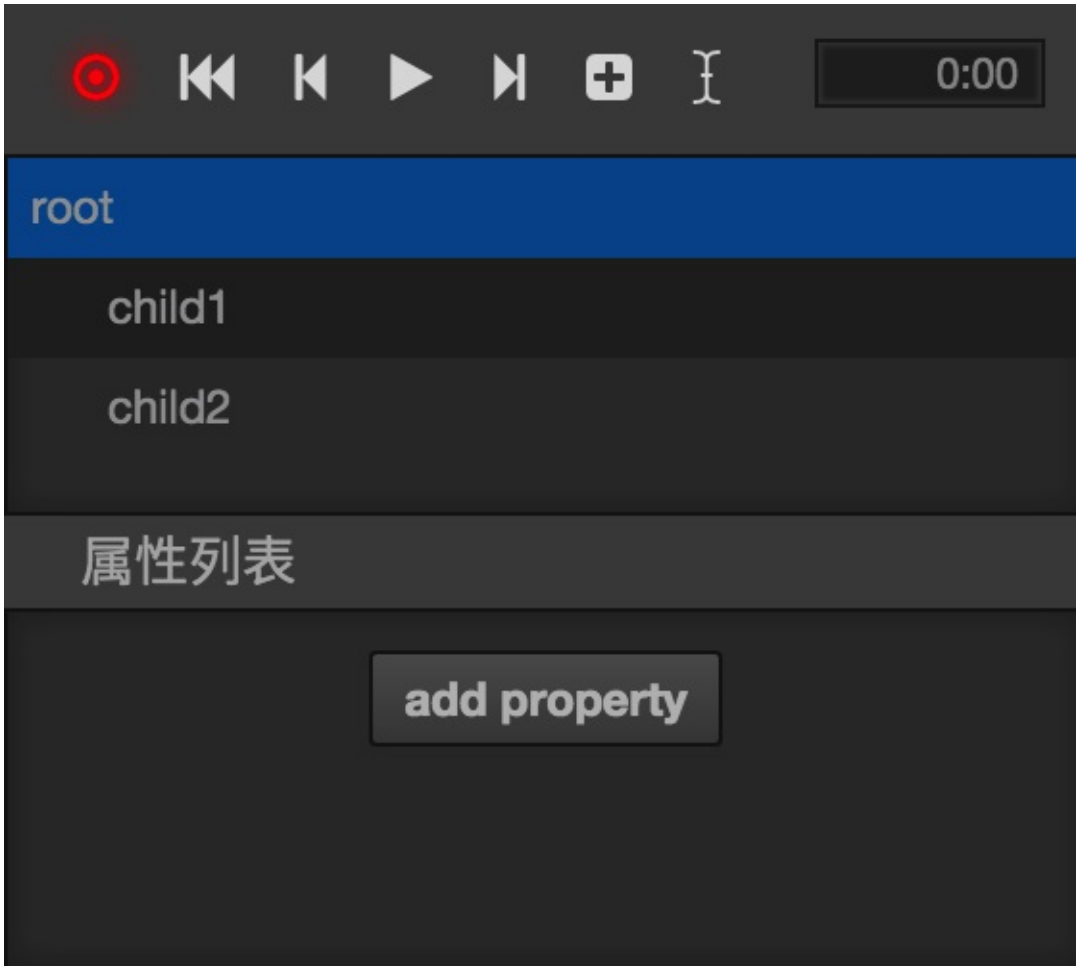
## 详细说明

如果一个动画需要包含多个节点，那么一般会新建一个节点来作为动画的 **根节点**，将 **Animation （动画）**组件添加到这个 **根节点**上，然后这个根节点下的其他子节点都会自动进入到这个动画中。

假如添加了如下所示的节点树：



那么在动画编辑器中的层级就会显示为：




更多关于 **Animation （动画）**的信息请前往 动画系统

# 物理系统

Creator 里的物理系统包括两个部分：

- 碰撞组件
- Box2D 物理引擎

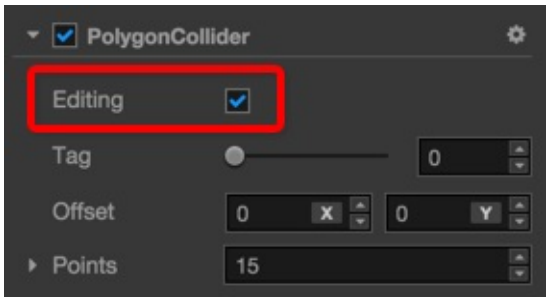
对于物理计算较为简单的情况，我们推荐用户直接使用碰撞组件，这样可以避免加载物理引擎并构建物理世界的运行时开销。而物理引擎提供了更完善的交互接口和刚体、关节等已经预设好的组件。可以根据需要来选择适合自己的物理系统。

# 碰撞系统

本章将介绍 Cocos Creator 的碰撞系统，目前 Cocos Creator 内置了一个简单易用的碰撞检测系统，支持 **圆形**，**矩形** 以及 **多边形** 相互间的碰撞检测。
下面会分成几个小节来介绍碰撞系统的细节内容。

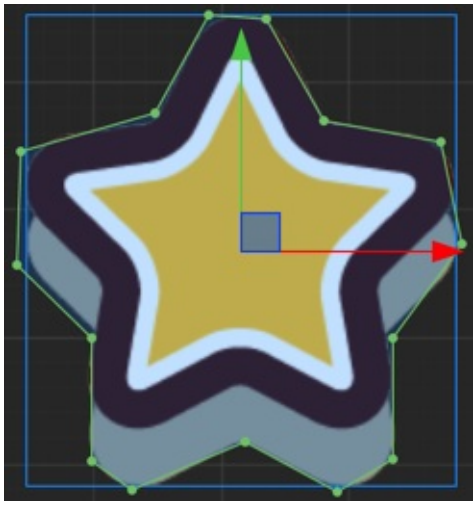
# 编辑碰撞组件

当添加了一个碰撞组件后，可以通过点击 **inspector** 中的 **editing** 来开启碰撞组件的编辑，如下图。



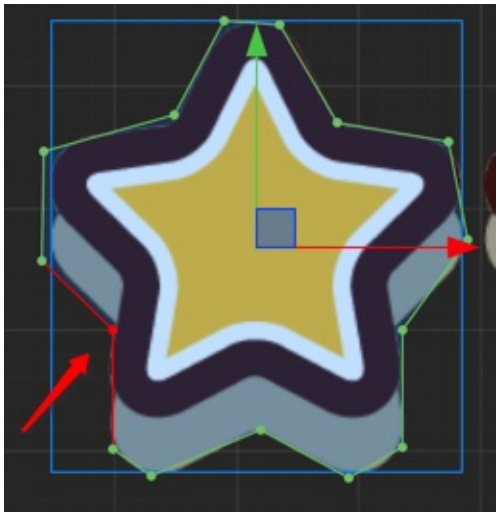

# 多边形碰撞组件

如果编辑的是 **多边形碰撞组件** 的话，则会出现类似下图所示的 **多边形编辑区域**
区域中的这些点都是可以拖动的，拖动的结果会反映到 **多边形碰撞组件** 的 **points** 属性中。



当鼠标移动到两点连成的线段上时，鼠标指针会变成 **添加** 样式，这时点击鼠标左键会在这个地方添加一个点到 **多边形碰撞组件** 中。

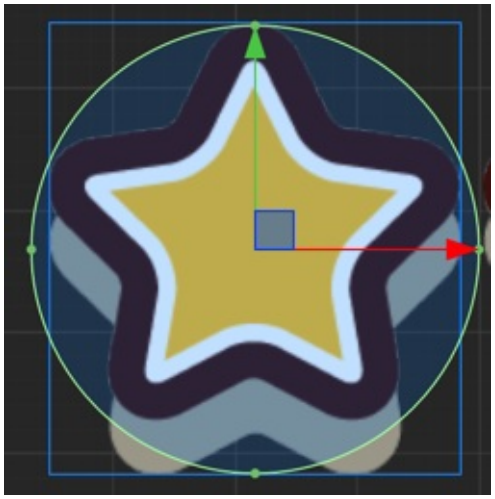
当按住 **ctrl** 或者 **command** 键时，移动鼠标到多边形顶点上，会发现顶点以及连接的两条线条变成红色，这时候点击鼠标左键将会删除 **多边形碰撞组件** 中的这个点。



在 CocosCreator 1.5 中，多边形碰撞组件中添加了一个 **Regenerate Points** 的功能，这个功能可以根据组件依附的节点上的 **Sprite** 组件的贴图的像素点来自动生成相应轮廓的顶点。
**Threshold** 指明生成贴图轮廓顶点间的最小距离，值越大则生成的点越少，可根据需求进行调节。

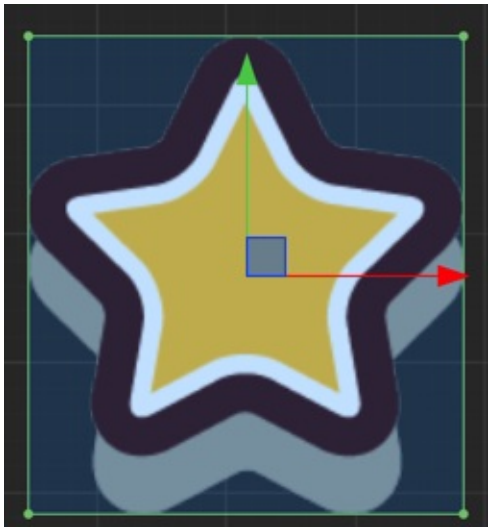

# 圆形碰撞组件

如果编辑的是 **圆形碰撞组件** 的话，则会出现类似下图所示的 **圆形编辑区域**

当鼠标悬浮在 **圆形编辑区域** 的边缘线上时，边缘线会变亮，这时点击鼠标左键拖动将可以修改 **圆形碰撞组件** 的半径大小。 

# 矩形碰撞组件

如果编辑的是 **矩形碰撞组件** 的话，则会出现类似下图所示的 **矩形编辑区域**



当鼠标悬浮在 **矩形碰撞区域** 的顶点上时，点击鼠标左键拖拽可以同时修改 **矩形碰撞组件** 的长宽； 当鼠标悬浮在 **矩形碰撞区域** 的边缘线上时，点击鼠标左键拖拽将修改 **矩形碰撞组件** 的长或宽中的一个方向。

按住 **shift** 键拖拽时，在拖拽过程中将会保持按下鼠标那一刻的 **长宽比例**， 按住 **alt** 建拖拽时，在拖拽过程中将会保持 **矩形中心点位置** 不变。

# 修改碰撞组件偏移量

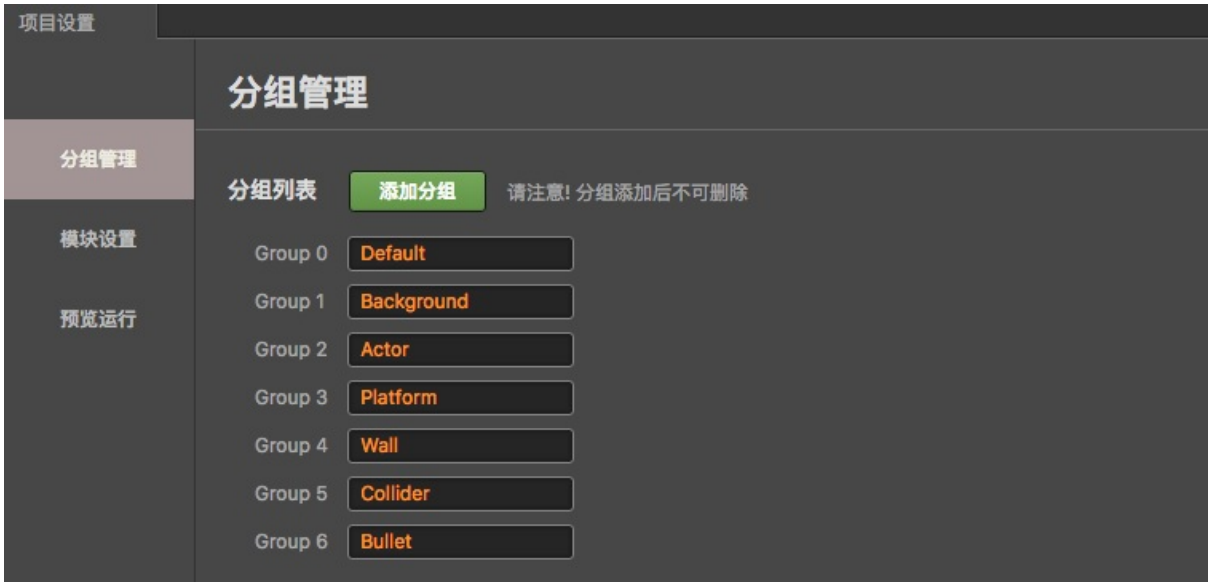
在所有的碰撞组件编辑中，都可以在各自的 **碰撞中心区域** 点击鼠标左键拖拽来快速编辑碰撞组件的 **偏移量**

# 碰撞分组管理

## 分组管理

分组管理，需要打开 **项目设置** 面板进行设置，位置为 菜单栏 => 项目 => 项目设置。

打开 **项目设置** 面板后，在 **分组管理** 一栏可以看到 **分组列表** 的配置项，如下图

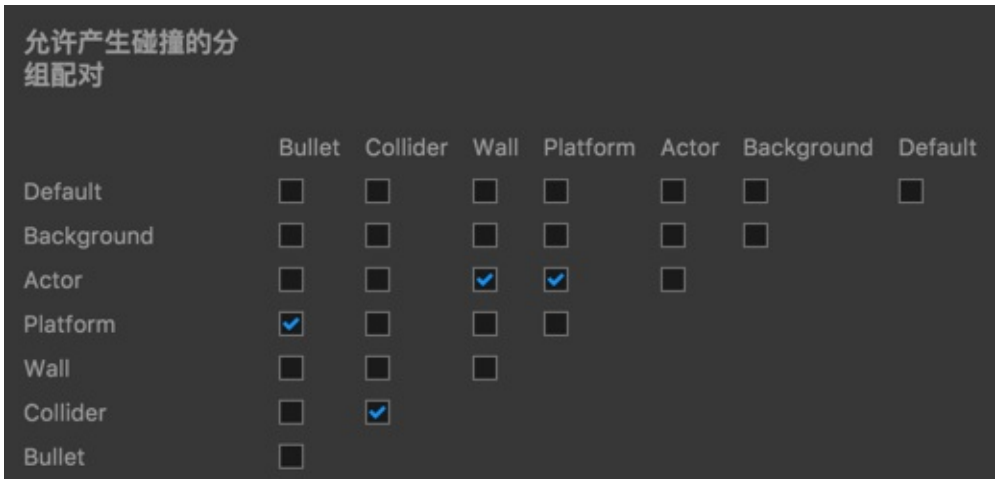



点击 **添加分组** 按钮后即可添加一个新的分组，默认会有一个 **Default** 分组。

**需要注意的是：分组添加后是不可以删除的，不过你可以任意修改分组的名字**

## 碰撞分组配对

在 **分组列表** 下面可以进行 **碰撞分组配对** 表的管理，如下图




这张表里面的行与列分别列出了 **分组列表** 里面的项，**分组列表** 里的修改将会实时映射到这张表里。
你可以在这张表里面配置哪一个分组可以其他的分组进行碰撞检测，假设 **a行 b列** 被勾选上，那么表示 **a行** 上的分组将会与 **b列** 上的分组进行碰撞检测

根据上面的规则，在这张表里产生的碰撞对有：

Platform - Bullet
Collider - Collider
Actor - Wall
Actor - Platform

# 碰撞系统脚本控制

Cocos Creator 中内置了一个简单易用的碰撞检测系统，他会根据添加的碰撞组件进行碰撞检测。
当一个碰撞组件被启用时，这个碰撞组件会被自动添加到碰撞检测系统中，并搜索能够与他进行碰撞的其他已添加的碰撞组件来生成一个碰撞对。
需要注意的是，一个节点上的碰撞组件，无论如何都是不会相互进行碰撞检测的。

# 碰撞检测系统的使用

## 碰撞系统接口

获取碰撞检测系统

```
var manager = cc.director.getCollisionManager();
```

默认碰撞检测系统是禁用的，如果需要使用则需要以下方法开启碰撞检测系统

```
manager.enabled = true;
```

默认碰撞检测系统的 debug 绘制是禁用的，如果需要使用则需要以下方法开启 debug 绘制

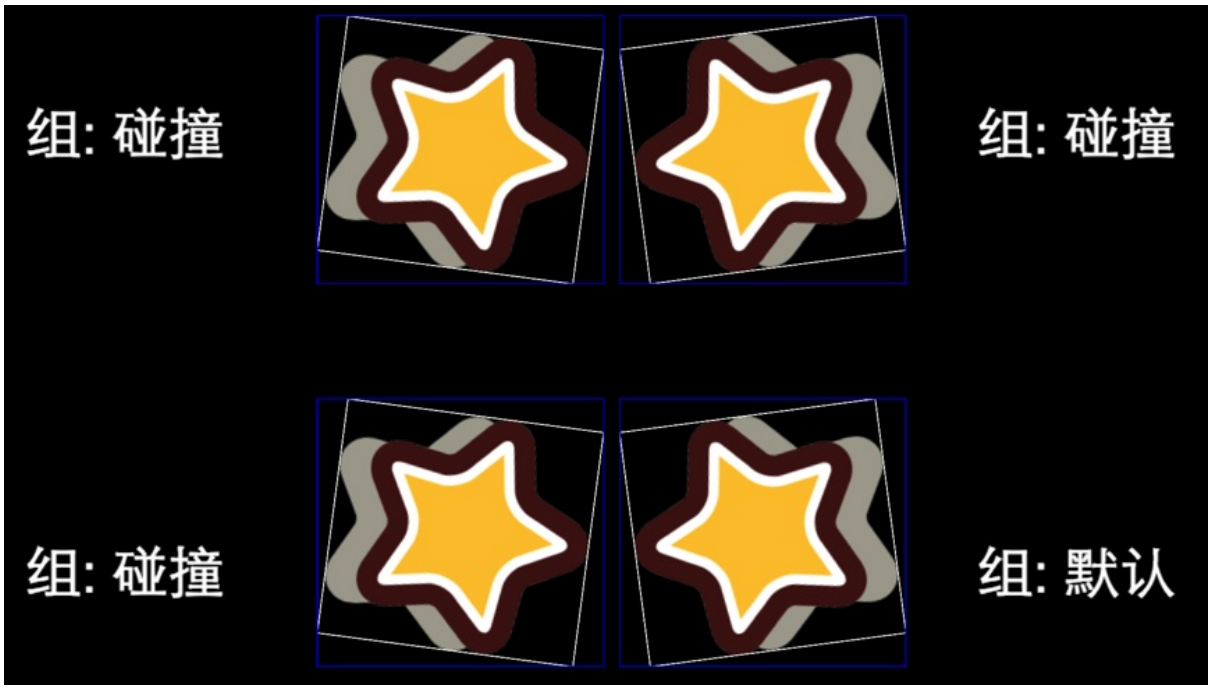
```
manager.enabledDebugDraw = true;
```

开启后在运行时可显示 **碰撞组件** 的 **碰撞检测范围**，如下图


如果还希望显示碰撞组件的包围盒，那么可以通过以下接口来进行设置

```
manager.enabledDrawBoundingBox = true;
```

结果如下图




## 碰撞系统回调

当碰撞系统检测到有碰撞产生时，将会以回调的方式通知使用者，如果产生碰撞的碰撞组件依附的节点下挂的脚本中有实现以下函数，则会自动调用以下函数，并传入相关的参数。

```
/**
 * 当碰撞产生的时候调用
 * @param  {Collider} other 产生碰撞的另一个碰撞组件
 * @param  {Collider} self  产生碰撞的自身的碰撞组件
 */
onCollisionEnter: function (other, self) {
    console.log('on collision enter');

    // 碰撞系统会计算出碰撞组件在世界坐标系下的相关的值，并放到 world 这个属性里面
    var world = self.world;

    // 碰撞组件的 aabb 碰撞框
    var aabb = world.aabb;

    // 上一次计算的碰撞组件的 aabb 碰撞框
    var preAabb = world.preAabb;

    // 碰撞框的世界矩阵
    var t = world.transform;

    // 以下属性为圆形碰撞组件特有属性
    var r = world.radius;
    var p = world.position;

    // 以下属性为 矩形 和 多边形 碰撞组件特有属性
    var ps = world.points;
},
```

```
/**
 * 当碰撞产生后，碰撞结束前的情况下，每次计算碰撞结果后调用
 * @param  {Collider} other 产生碰撞的另一个碰撞组件
 * @param  {Collider} self  产生碰撞的自身的碰撞组件
```

```
 */
onCollisionStay: function (other, self) {
    console.log('on collision stay');
},
```

```
/**
 * 当碰撞结束后调用
 * @param  {Collider} other 产生碰撞的另一个碰撞组件
 * @param  {Collider} self  产生碰撞的自身的碰撞组件
 */
onCollisionExit: function (other, self) {
    console.log('on collision exit');
}
```

## 点击测试

```
cc.eventManager.addListener({
    event: cc.EventListener.TOUCH_ONE_BY_ONE,
    onTouchBegan: (touch, event) => {
        var touchLoc = touch.getLocation();

        // 获取多边形碰撞组件的世界坐标系下的点来进行点击测试
        // 如果是其他类型的碰撞组件，也可以在 cc.Intersection 中找到相应的测试函数
        if (cc.Intersection.pointInPolygon(touchLoc, this.polygonCollider.world.points)) {
            this.title.string = 'Hit';
        }
        else {
            this.title.string = 'Not hit';
        }

        return true;
    },
}, this.node);
```

更多的范例可以到 github 上查看

# Collider 组件参考

点击 **属性检查器** 下面的 `添加组件` 按钮，然后从 `添加碰撞组件` 中选择需要的 **Collider** 组件，即可添加 **Collider** 组件到节点上。

# Collider 组件属性

| 属性 | 功能说明 |
| --- | --- |
| tag | 标签。当一个节点上有多个碰撞组件时，在发生碰撞后，可以使用此标签来判断是节点上的哪个碰撞组件被碰撞了。 |
| editing | 是否编辑此碰撞组件，只在编辑器中有效 |

# 详细说明

一个节点上可以挂多个碰撞组件，这些碰撞组件之间可以是不同类型的碰撞组件。

碰撞组件目前包括了 **Polygon（多边形）**，**Circle（圆形）**，**Box（矩形）** 这几种碰撞组件，这些组件都继承自 **Collider** 组件，所以 **Collider** 组件的属性他们也都享有。

## Polygon（多边形） 碰撞组件属性

| 属性 | 功能说明 |
| --- | --- |
| offset | 组件相对于节点的 **偏移量**。 |
| points | 组件的 **顶点数组**。 |

## Circle（圆形） 碰撞组件属性

| 属性 | 功能说明 |
| --- | --- |
| offset | 组件相对于节点的 **偏移量**。 |
| radius | 组件的 **半径**。 |

## Box（矩形） 碰撞组件属性

| 属性 | 功能说明 |
| --- | --- |
| offset | 组件相对于节点的 **偏移量**。 |
| size | 组件的 **长宽**。 |

更多关于 **Collider** 的信息请前往 碰撞系统

# 物理系统

- 物理系统
- 刚体组件
- 碰撞组件
- 碰撞回调
- 关节组件

更多使用方法请到 物理系统示例 查阅

# 物理系统

物理系统将 box2d 作为内部物理系统，并且隐藏了大部分 box2d 实现细节（比如创建刚体，同步刚体信息到节点中等）。 你可以通过物理系统访问一些 box2d 常用的功能，比如点击测试，射线测试，设置测试信息等。

# 物理系统相关设置

## 开启物理系统

物理系统默认是关闭的，如果需要使用物理系统，那么首先需要做的事情就是开启物理系统，否则你在编辑器里做的所有物理编辑都不会产生任何效果。

```
cc.director.getPhysicsManager().enabled = true;
```

## 绘制物理调试信息

物理系统默认是不绘制任何调试信息的，如果需要绘制调试信息，请使用 **debugDrawFlags** 。 物理系统提供了各种各样的调试信息，你可以通过组合这些信息来绘制相关的内容。

```
cc.director.getPhysicsManager().debugDrawFlags = cc.PhysicsManager.DrawBits.e_aabbBit |
    cc.PhysicsManager.DrawBits.e_pairBit |
    cc.PhysicsManager.DrawBits.e_centerOfMassBit |
    cc.PhysicsManager.DrawBits.e_jointBit |
    cc.PhysicsManager.DrawBits.e_shapeBit
    ;
```

设置绘制标志位为 **0**，即可以关闭绘制。

```
cc.director.getPhysicsManager().debugDrawFlags = 0;
```

## 物理单位到像素单位的转换

box2d 使用 **米-千克-秒(MDS)** 单位制，box2d 在这样的单位制下运算的表现是最佳的。 但是我们在 2D 游戏运算中一般使用 **像素** 来作为长度单位制，所以我们需要一个比率来进行物理单位到像素单位上的相互转换。 一般情况下我们把这个比率设置为 32，这个值可以通过 `cc.PhysicsManager.PTM_RATIO` 获取，并且这个值是只读的。 通常用户是不需要关心这个值的，物理系统内部会自动对物理单位与像素单位进行转换，用户访问和设置的都是进行 2d 游戏开发中所熟悉的像素单位。

## 设置物理重力

重力是物理表现中非常重要的一点，大部分物理游戏都会使用到重力这一物理特性。 默认的重力加速度是 (0, -320) 像素/秒^2，按照上面描述的转换规则，即 (0, -10) 米/秒^2。

如果希望重力加速度为 0，可以这样设置：

```
cc.director.getPhysicsManager().gravity = cc.v2();
```

如果希望修改重力加速度为其他值，比如每秒加速降落 640 像素，那么可以这样设置：

```
cc.director.getPhysicsManager().gravity = cc.v2(0, -640);
```

# 查询物体

通常你可能想知道在给定的场景中都有哪些实体。 比如如果一个炸弹爆炸了，在范围内的物体都会受到伤害，或者在策略类游戏中，可能会希望让用户选择一个范围内的单位进行拖动。

物理系统提供了几个方法来高效快速地查找某个区域中有哪些物体，每种方法通过不同的方式来检测物体，基本满足游戏所需。

## 点测试

点测试将测试是否有碰撞体会包含一个世界坐标系下的点，如果测试成功，则会返回一个包含这个点的碰撞体。注意，如果有多个碰撞体同时满足条件，下面的接口只会返回一个随机的结果。

```
var collider = cc.director.getPhysicsManager().testPoint(point);
```

## 矩形测试

矩形测试将测试指定的一个世界坐标系下的矩形，如果一个碰撞体的包围盒与这个矩形有重叠部分，则这个碰撞体会给添加到返回列表中。

```
var colliderList = cc.director.getPhysicsManager().testAABB(rect);
```

## 射线测试

射线检测用来检测给定的线段穿过哪些碰撞体，我们还可以获取到碰撞体在线段穿过碰撞体的那个点的法线向量和其他一些有用的信息。

```
var results = cc.director.getPhysicsManager().rayCast(p1, p2, type);

for (var i = 0; i < results.length; i++) {
    var result = results[i];
    var collider = result.collider;
    var point = result.point;
    var normal = result.normal;
    var fraction = result.fraction;
}
```

射线检测的最后一个参数指定检测的类型，射线检测支持四种类型。 这是因为 box2d 的射线检测不是从射线起始点最近的物体开始检测的，所以检测结果不能保证结果是按照物体距离射线起始点远近来排序的。 CocosCreator 物理系统将根据射线检测传入的检测类型来决定是否对 box2d 检测结果进行排序，这个类型会影响到最后返回给用户的结果。

- cc.RayCastType.Any
  检测射线路径上任意的碰撞体，一旦检测到任何碰撞体，将立刻结束检测其他的碰撞体，最快。
- cc.RayCastType.Closest
  检测射线路径上最近的碰撞体，这是射线检测的默认值，稍慢。
- cc.RayCastType.All
  检测射线路径上的所有碰撞体，**检测到的结果顺序不是固定的。在这种检测类型下一个碰撞体可能会返回多个结果**，这是因为 box2d 是通过检测夹具(fixture)来进行物体检测的，而一个碰撞体中可能由多个夹具(fixture)组成的，慢。更多细节可到 物理碰撞组件 查看。

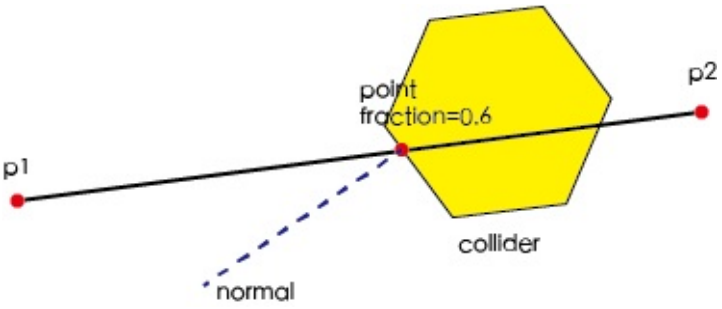
- cc.RayCastType.AllClosest
  检测射线路径上所有碰撞体，但是会对返回值进行删选，只返回每一个碰撞体距离射线起始点最近的那个点的相关信息，最慢。

## 射线检测的结果

射线检测的结果包含了许多有用的信息，你可以根据实际情况来选择如何使用这些信息。

- collider
  指定射线穿过的是哪一个碰撞体。
- point
  指定射线与穿过的碰撞体在哪一点相交。
- normal
  指定碰撞体在相交点的表面的法线向量。
- fraction
  指定相交点在射线上的分数。

可以通过下面这张图更好的理解射线检测的结果。



# 刚体

刚体是组成物理世界的基本对象，你可以将刚体想象成一个你不能看到（绘制）也不能摸到（碰撞）的带有属性的物体。

# 刚体属性

## 质量

刚体的质量是通过碰撞组件的 **密度** 与 大小 自动计算得到的。
当你需要计算物体应该受到多大的力时可能需要使用到这个属性。

```
var mass = rigidbody.getMass();
```

## 移动速度

```
// 获取移动速度
var velocity = rigidbody.linearVelocity;
// 设置移动速度
rigidbody.linearVelocity = velocity;
```

移动速度衰减系数，可以用来模拟空气摩擦力等效果，它会使现有速度越来越慢。

```
// 获取移动速度衰减系数
var damping = rigidbody.linearDamping;
// 设置移动速度衰减系数
rigidbody.linearDamping = damping;
```

有些时候可能会希望获取刚体上某个点的移动速度，比如一个盒子旋转着往前飞，碰到了墙，这时候可能会希望获取盒子在发生碰撞的点的速度，可以通过 `getLinearVelocityFromWorldPoint` 来获取。

```
var velocity = rigidbody.getLinearVelocityFromWorldPoint(worldPoint);
```

或者传入一个 cc.Vec2 对象作为第二个参数来接收返回值，这样你可以使用你的缓存对象来接收这个值，避免创建过多的对象来提高效率。

**刚体的 get 方法都提供了 out 参数来接收函数返回值。**

```
var velocity = cc.v2();
rigidbody.getLinearVelocityFromWorldPoint(worldPoint, velocity);
```

## 旋转速度

```
// 获取旋转速度
var velocity = rigidbody.angularVelocity;
// 设置旋转速度
rigidbody.angularVelocity = velocity;
```

旋转速度衰减系数，与移动衰减系数相同。

```
// 获取旋转速度衰减系数
var velocity = rigidbody.angularDamping;
// 设置旋转速度衰减系数
rigidbody.angularDamping = velocity;
```

## 旋转，位移与缩放

旋转，位移与缩放是游戏开发中最常用的功能，几乎每个节点都会对这些属性进行设置。而在物理系统中，系统会自动对节点的这些属性与 box2d 中对应属性进行同步。

有几点信息需要大家**注意**：

1. box2d 中只有旋转和位移，并没有缩放，所以如果设置节点的缩放属性时，会重新构建这个刚体依赖的全部碰撞体。一个有效避免这种情况发生的方式是将渲染的节点作为刚体节点的子节点，缩放只对这个渲染节点作缩放，尽量避免对刚体节点进行直接缩放。
2. 每个物理时间步之后会把所有刚体信息同步到对应节点上去，而处于性能考虑，节点的信息只有在用户对节点相关属性进行显示设置时才会同步到刚体上，并且刚体只会监视他所在的节点，即如果修改了节点的父节点的旋转位移是不会同步这些信息的。

## 固定旋转

做平台跳跃游戏时通常都不会希望主角的旋转属性也被加入到物理模拟中，因为这样会导致主角在移动过程中东倒西歪的，这时可以设置刚体的 `fixedRotation` 属性。

```
rigidbody.fixedRotation = true;
```

## 开启碰撞监听

只有开启了刚体的碰撞监听，刚体发生碰撞时才会回调到对应的组件上。

```
rigidbody.enabledContactListener = true;
```

# 刚体类型

box2d 原本的刚体类型是三种：**Static**, **Dynamic**, **Kinematic**。在 CocosCreator 里多添加了一个类型：**Animated**。
Animated 是从 Kinematic 类型衍生出来的，一般的刚体类型修改 **旋转** 或 **位移** 属性时，都是直接设置的属性，而 Animated 会根据当前 旋转或位移 属性与目标 旋转或位移 属性计算出所需的速度，并且赋值到对应的 移动或旋转 速度上。
添加 Animated 类型主要是防止对刚体做动画时可能出现的奇怪现象，比如 穿透 等。

- cc.RigidBodyType.Static
  静态刚体，零质量，零速度，即不会受到重力或速度影响，但是可以设置他的位置来进行移动。
- cc.RigidBodyType.Dynamic
  动态刚体，有质量，可以设置速度，会受到重力影响。
- cc.RigidBodyType.Kinematic
  运动刚体，零质量，可以设置速度，不会受到重力的影响，但是可以设置速度来进行移动。
- cc.RigidBodyType.Animated
  动画刚体，在上面已经提到过，从 Kinematic 衍生的类型，主要用于刚体与动画编辑结合使用。

# 刚体方法

## 获取或转换旋转位移属性

使用这些 api 来获取世界坐标系下的旋转位移会比通过节点来获取相关属性更快，因为节点中还需要通过矩阵运算来得到结果，而这些 api 是直接得到结果的。

## 获取刚体世界坐标值

```
// 直接获取返回值
var out = rigidbody.getWorldPosition();

// 或者通过参数来接收返回值
out = cc.v2();
rigidbody.getWorldPosition(out);
```

## 获取刚体世界旋转值

```
var rotation = rigidbody.getWorldRotation();
```

## 局部坐标与世界坐标转换

```
// 世界坐标转换到局部坐标
var localPoint = rigidbody.getLocalPoint(worldPoint);
// 或者
localPoint = cc.v2();
rigidbody.getLocalPoint(worldPoint, localPoint);
```

```
// 局部坐标转换到世界坐标
var worldPoint = rigidbody.getWorldPoint(localPoint);
// 或者
worldPoint = cc.v2();
rigidbody.getLocalPoint(localPoint, worldPoint);
```

```
// 局部向量转换为世界向量
var worldVector = rigidbody.getWorldVector(localVector);
// 或者
worldVector = cc.v2();
rigidbody.getWorldVector(localVector, worldVector);
```

```
var localVector = rigidbody.getLocalVector(worldVector);
// 或者
localVector = cc.v2();
rigidbody.getLocalVector(worldVector, localVector);
```

## 获取刚体质心

当对一个刚体进行力的施加时，一般会选择刚体的质心作为施加力的作用点，这样能保证力不会影响到旋转值。

```
// 获取本地坐标系下的质心
var localCenter = rigidbody.getLocalCenter();

// 或者通过参数来接收返回值
```

```
localCenter = cc.v2();
rigidbody.getLocalCenter(localCenter);

// 获取世界坐标系下的质心
var worldCenter = rigidbody.getWorldCenter();

// 或者通过参数来接收返回值
worldCenter = cc.v2();
rigidbody.getWorldCenter(worldCenter);
```

## 力与冲量

移动一个物体有两种方式，可以施加一个力或者冲量到这个物体上。力会随着时间慢慢修改物体的速度，而冲量会立即修改物体的速度。 当然你也可以直接修改物体的位置，只是这看起来不像真实的物理，你应该尽量去使用力或者冲量来移动刚体，这会减少可能带来的奇怪问题。

```
// 施加一个力到刚体上指定的点上，这个点是世界坐标系下的一个点
rigidbody.applyForce(force, point);

// 或者直接施加力到刚体的质心上
rigidbody.applyForceToCenter(force);

// 施加一个冲量到刚体上指定的点上，这个点是世界坐标系下的一个点
rigidbody.applyLinearImpulse(impulse, point);
```

力与冲量也可以只对旋转轴产生影响，这样的力叫做扭矩。

```
// 施加扭矩到刚体上，因为只影响旋转轴，所以不再需要指定一个点
rigidbody.applyTorque(torque);

// 施加旋转轴上的冲量到刚体上
rigidbody.applyAngularImpulse(impulse);
```

## 其他

有些时候需要获取刚体在某一点上的速度时，可以通过 `getLinearVelocityFromWorldPoint` 来获取，比如当物体碰撞到一个平台时，需要根据物体碰撞点的速度来判断物体相对于平台是从上方碰撞的还是下方碰撞的。

```
rigidbody.getLinearVelocityFromWorldPoint(worldPoint);
```

# 物理碰撞组件

**物理碰撞组件** 继承自 **碰撞组件** ，编辑和设置 **物理碰撞组件** 的方法和 编辑碰撞组件 是基本一致的。

# 物理碰撞组件属性

- sensor - 指明碰撞体是否为传感器类型，传感器类型的碰撞体会产生碰撞回调，但是不会发生物理碰撞效果。
- density - 碰撞体的密度，用于刚体的质量计算
- friction - 碰撞体摩擦力，碰撞体接触时的运动会受到摩擦力影响
- restitution - 碰撞体的弹性系数，指明碰撞体碰撞时是否会受到弹力影响

## 物理碰撞组件内部细节

物理碰撞组件内部是由 box2d 的 b2Fixture 组成的，由于 box2d 内部的一些限制，一个多边形物理碰撞组件可能会由多个 b2Fixture 组成。

这些情况为：

1. 当多边形物理碰撞组件的顶点组成的形状为凹边形时，物理系统会自动将这些顶点分割为多个凸边形。
2. 当多边形物理碰撞组件的顶点数多于 `b2.maxPolygonVertices` (一般为 8) 时，物理系统会自动将这些顶点分割为多个凸边形。

一般情况下这些细节是不需要关心的，但是当使用射线检测并且检测类型为 `cc.RayCastType.All` 时，一个碰撞体就可能会检测到多个碰撞点，原因即是检测到了多个 b2Fixture。

# 碰撞回调

当物体在场景中移动并碰撞到其他物体时，box2d 会处理大部分必要的碰撞检测，我们一般不需要关心这些情况。但是制作物理游戏最主要的点是有些情况下物体碰撞后应该发生些什么，比如角色碰到怪物后会死亡，或者球在地上弹动时应该产生声音等。

我们需要一个方式来获取到这些碰撞信息，物理引擎提供的方式是在碰撞发生时产生回调，在回调里我们可以根据产生碰撞的两个碰撞体的类型信息来判断需要作出什么样的动作。

**注意**

1. 需要先在 rigidbody 中 **开启碰撞监听**，才会有相应的回调产生。
2. 回调中的信息在物理引擎都是以缓存的形式存在的，所以信息只有在这个回调中才是有用的，不要在你的脚本里缓存这些信息，但你可以复制这些信息到你自己的缓存中来使用。
3. 在回调中创建的物理物体，比如刚体，关节等，这些不会立刻就创建出 box2d 对应的物体，会在整个物理系统更新完成后再进行这些物体的创建。

# 定义回调函数

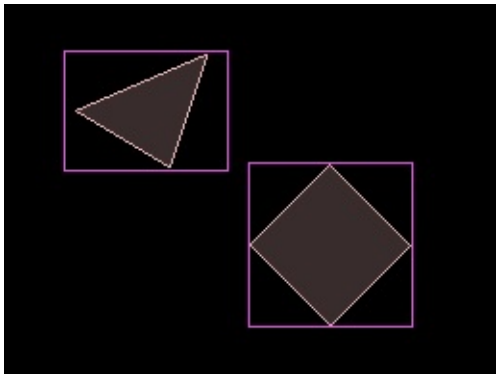
定义一个碰撞回调函数很简单，只需要在刚体所在的节点上挂一个脚本，脚本中添加上你需要的回调函数即可。

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

    // 只在两个碰撞体开始接触时被调用一次
    onBeginContact: function (contact, selfCollider, otherCollider) {
    },

    // 只在两个碰撞体结束接触时被调用一次
    onEndContact: function (contact, selfCollider, otherCollider) {
    },

    // 每次将要处理碰撞体接触逻辑时被调用
    onPreSolve: function (contact, selfCollider, otherCollider) {
    },

    // 每次处理完碰撞体接触逻辑时被调用
    onPostSolve: function (contact, selfCollider, otherCollider) {
    }
});
```

在上面的代码示例中，我们添加了所有的碰撞回调函数到这个脚本中，一共有四个类型的回调函数，每个回调函数都有三个参数。每种回调函数的作用如注释所示，你可以根据自己的需求来实现相应的回调函数。

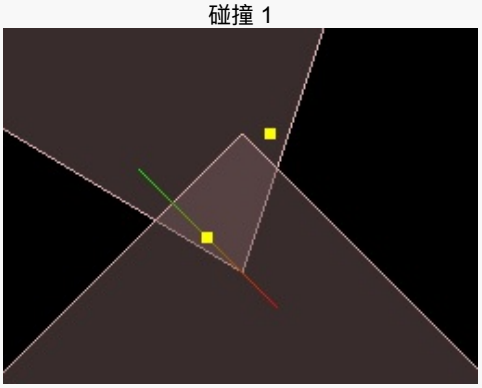
# 回调的顺序

我们可以通过拆分一个简单的示例的碰撞过程来描述碰撞回调函数的回调顺序和回调的时机，假设有两个刚体正相互移动，三角形往右运动，方块往左运动，即将碰撞到了一起。

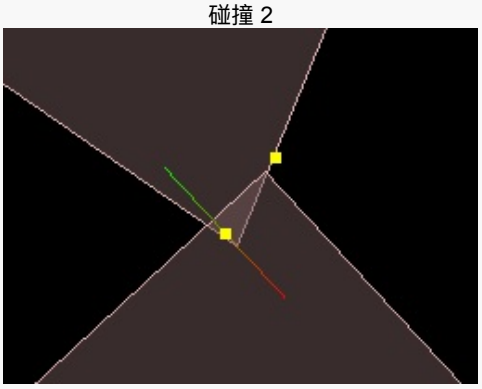


碰撞的过程

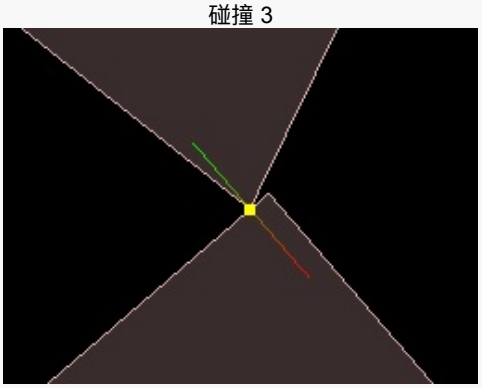
碰撞 1



碰撞 2



碰撞 3



当两个碰撞体相互覆盖时，box2d 默认的行为是给每个碰撞体一个冲量去把他们分开，但是这个行为不一定能在一个时间步内完成。 像这里显示的一样，示例中的碰撞体会在三个时间步内相互覆盖直到“反弹”完成并且他们相互分离。
在这个时间里我们可以定制我们想要的行为，**onPreSolve** 会在每次物理引擎处理碰撞前回调，我们 可以在这个回调里修改碰撞信息，而 **onPostSolve** 会在处理完成这次碰撞后回调，我们可以在这个回调中获取到物理引擎计算出的碰撞的冲量信息。
下面给出的输出信息能使我们更清楚回调的顺序。

```
    ...
Step
Step
BeginContact
PreSolve
PostSolve
Step
PreSolve
PostSolve
Step
PreSolve
PostSolve
Step
EndContact
Step
Step
...
```

# 回调的参数

回调的参数包含了所有的碰撞接触信息，每个回调函数都提供了三个参数 : **contact**, **selfCollider**, **otherCollider**。

**selfCollider** 和 **otherCollider** 很容易理解，如名字所示，**selfCollider** 指的是回调脚本的节点上的碰撞体，**ohterCollider** 指的是发生碰撞的另一个碰撞体。

最主要的信息都包含在 **contact** 中，这是一个 **cc.PhysicsContact** 类型的实例，可以在 api 文档中找到相关的 api 。contact 中比较常用的信息就是碰撞的位置和法向量，contact 内部是按照刚体的本地坐标来存储信息的，而我们一般需要的是世界坐标系下的信息，我们可以通过 `contact.getWorldManifold` 来获取这些信息。

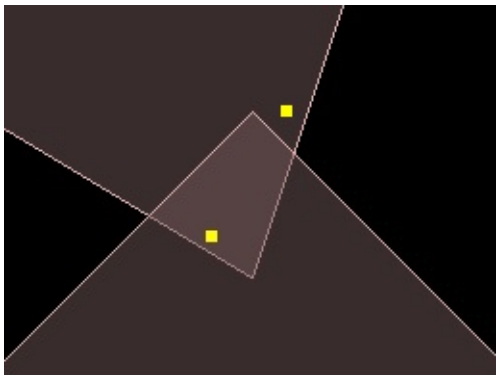
## worldManifold

```
var worldManifold = contact.getWorldManifold();
var points = worldManifold.points;
var normal = worldManifold.normal;
```

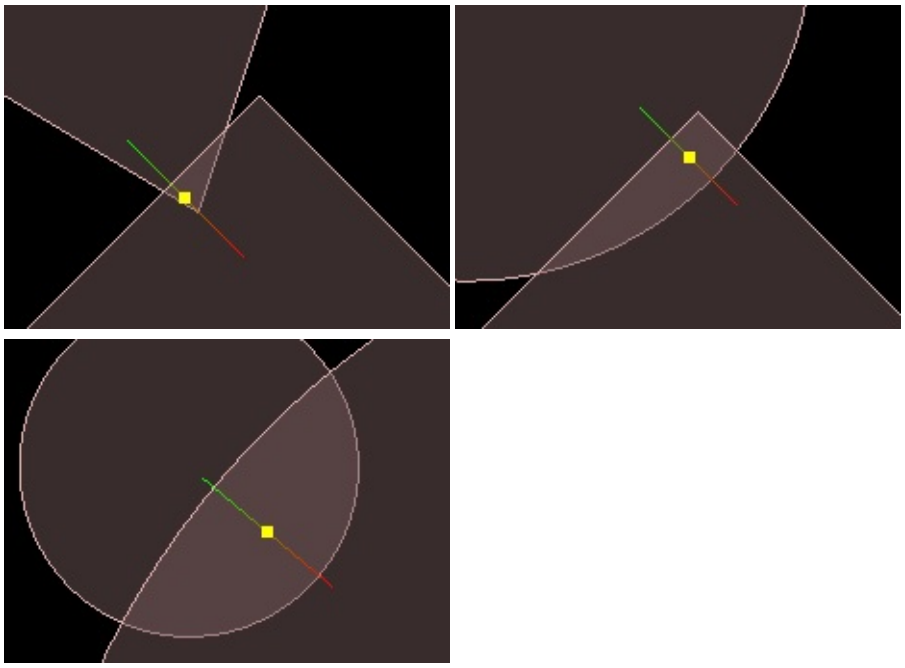
`worldManifold` 具有以下成员：

## points

碰撞点数组，他们不一定会精确的在碰撞体碰撞的地方上，如下图所示（除非你将刚体设置为子弹类型，但是会比较耗性能），但实际上这些点在使用上一般都是够用的。

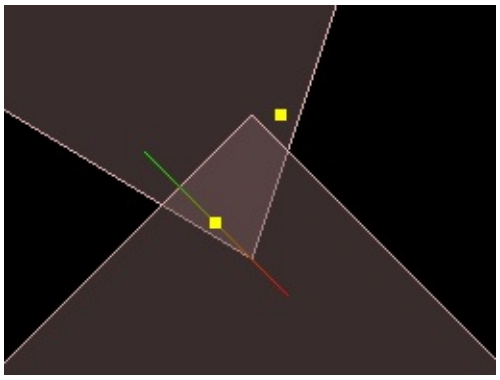


**注意**：
不是每一个碰撞都会有两个碰撞点，在模拟的更多的情况下只会产生一个碰撞点，下面列举一些其他的碰撞示例。



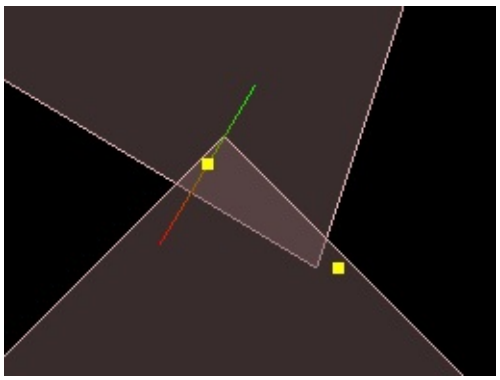
- normal

碰撞点上的法向量，由自身碰撞体指向对方碰撞体，指明解决碰撞最快的方向。



在图中所示的线条即碰撞点上的法向量，在这个碰撞中，解决碰撞最快的途径是添加冲量将三角形往左上推，将方块往右下推。需要注意的是这里的法向量只是一个方向，并不带有位置属性，也不会连接到这些碰撞点中的任何一个。

你还需要明白的是**碰撞法向量并不是碰撞体碰撞的角度**，他只会指明可以解决两个碰撞体相互覆盖这一问题最短的方向。比如上面的例子中如果三角形移动得更快一点，覆盖的情形像下图所示的话：



那么最短的方式将会是把三角形往右上推，所以使用法向量来作为碰撞角度不是一个好主意。如果你希望知道碰撞的真正的方向，可以使用下面的方式：

```
var vel1 = triangleBody.getLinearVelocityFromWorldPoint( worldManifold.points[0] );
var vel2 = squareBody.getLinearVelocityFromWorldPoint( worldManifold.points[0] );
var relativeVelocity = vel1.sub(vel2);
```

这个代码可以获取到两个碰撞体相互碰撞时在碰撞点上的相对速度。

## 禁用 contact

```
contact.disabled = true;
```

禁用掉 contact 会使物理引擎在计算碰撞时会忽略掉这次碰撞，禁用将会持续到碰撞完成，除非在其他回调中再将这个 contact 启用。

或者如果你只想在本次物理处理步骤中禁用 contact ，可以使用 `disabledOnce` 。

```
contact.disabledOnce = true;
```

## 修改 contact 信息

前面有提到我们在 **onPreSolve** 中修改 contact 的信息，因为 **onPreSolve** 是在物理引擎处理碰撞信息前回调的，所以对碰撞信息的修改会影响到后面的碰撞计算。

```
// 修改碰撞体间的摩擦力
contact.setFriction(friction);

// 修改碰撞体间的弹性系数
contact.setRestitution(restitution);
```

注意这些修改只会在本次物理处理步骤中生效。

# 关节组件

物理系统包含了一系列用于链接两个刚体的关节组件。关节组件可以用来模拟真实世界物体间的交互，比如铰链，活塞，绳子，轮子，滑轮，机动车，链条等。学习如何使用关节组件可以有效地帮助创建一个真实有趣的场景。

目前物理系统中提供了以下可用的关节组件：

- Revolute Joint - 旋转关节，可以看做一个铰链或者钉，刚体会围绕一个共同点来旋转。
- Distance Joint - 距离关节，关节两端的刚体的锚点会保持在一个固定的距离。
- Prismatic Joint - 棱柱关节，两个刚体位置间的角度是固定的，他们只能在一个指定的轴上滑动。
- Weld Joint - 焊接关节，根据两个物体的初始角度将两个物体上的两个点绑定在一起。
- Wheel Joint - 轮子关节，由 Revolute 和 Prismatic 组合成的关节，用于模拟机动车车轮。
- Rope Joint - 绳子关节，将关节两端的刚体约束在一个最大范围内。
- Motor Joint - 马达关节，控制两个刚体间的相对运动。

# 关节的共同属性

虽然每种关节都有不同的表现，但是他们都有一些共同的属性。

- connectedBody - 关节链接的另一端的刚体
- anchor - 关节本端链接的刚体的锚点
- connectedAnchor - 关节另一端链接的刚体的锚点
- collideConnected - 关节两端的刚体是否能够互相碰撞

每个关节都需要链接上两个刚体才能够发挥他的功能，我们把和关节挂在同一节点下的刚体视为关节的本端，把 **connectedBody** 视为另一端的刚体。

通常情况下，每个刚体会围绕自身周围的位置来设定此点。 根据关节组件类型的不同，此点决定了物体的旋转中心，或者是用来保持一定距离的坐标点，等等。

**collideConnected** 属性允许你决定关节两端的刚体是否需要继续遵循常规的碰撞规则。 比如如果你现在准备制作一个布娃娃，你可能会希望大腿和小腿能够部分重合，然后在膝盖处链接到一起，那么就需要设置 **collideConnected** 属性为 false 。 如果你正在制作一个升降机，你可能希望升降机平台和地板能够碰撞，那么就需要设置 **collideConnected** 属性为 true。

# 音乐和音效

- 音频播放
- AudioSource 组件参考

# 音频播放

- 音频的加载方式请参考：声音资源

# 使用 AudioSource 播放

1. 创建一个空节点
2. 在这个空节点上，添加一个 '其他组件' - 'AudioSource'
3. 在脚本上预设好 AudioSource，并且根据实际需求，完善脚本的对外接口，如下：

```
cc.Class({
 properties: {
     audioSource: {
         type: cc.AudioSource,
         default: null
     },
 },
 play: function () {
     this.audioSource.play();
 },
 pause: function () {
     this.audioSource.pause();
 },
});
```

# 使用 AudioEngine 播放

1. 在脚本内定义一个 audioClip 资源对象，如下示例中 properties 对象内。
2. 直接使用 cc.audioEngine.play(audio, loop, volume); 播放。如下示例中 onLoad 中。

```
cc.Class({
    properties: {
        audio: {
            url: cc.AudioClip,
            default: null
        }
    },

    onLoad: function () {
        this.current = cc.audioEngine.play(this.audio, false, 1);
    },

    onDestroy: function () {
        cc.audioEngine.stop(this.current);
    }
});
```

AudioEngine 播放的时候，需要注意这里的传入的是一个完整的 url（与 res 路径稍有不同）。 所以我们不建议在 play 接口内直接填写音频的 url 地址，而是希望大家先定义一个 AudioClip，然后在编辑器内将音频拖拽过来。

# AudioSource 组件参考

## 属性

| 属性 | 说明 |
| --- | --- |
| Clip | 音频资源 |
| Volume | 音量大小 |
| Mute | 是否静音 |
| Loop | 是否循环 |
| Play on load | 加载完成是否立即播放 |
| preload | 是否在未播放的时候预先加载 |

更多音频接口的脚本接口请参考AudioSource API。

# 扩展编辑器

Cocos Creator 提供了一系列方法来让用户定制和扩展编辑器的功能。这些扩展以包（package）的形式进行加载。用户通过将自己或第三方开发的扩展包安装到正确的路径进行扩展的加载，根据扩展功能的不同，有时可能会要求用户手动刷新窗口或者重新启动编辑器来完成扩展包的初始化。

Cocos Creator 的扩展包沿用了 Node.js 社区的包设计方式，通过 `package.json` 描述文件来定义扩展包的内容和注册信息。

## 基本工作流程

- 你的第一个扩展包
- 安装与分享
- IPC （进程间通讯）简介
- 入口程序
- 扩展包工作流程模式
- 扩展主菜单
- 扩展编辑器面板
- 进程间通讯工作流程
- 定义一份简单的面板窗口
- 多语言化
- 工作路径和常用 URL
- 提交插件到商店

## 管理项目中的场景和资源

- 调用引擎 API 和项目脚本
- 管理项目资源

## 编写和扩展面板 UI

- 编写面板界面
- 掌握 UI Kit
- 扩展 UI Kit
- 界面排版
- 在面板中使用 Vue

## 扩展属性面板

- 扩展 Inspector
- 扩展 Property

## 扩展 Gizmos

- 自定义 Gizmo
- 自定义 Gizmo 进阶

# 其他

- 测试你的扩展包

# API

- AssetDB
  - AssetDB API Main
  - AssetDB API Renderer
- Editor

# 参考

- package.json 字段参考
- 主菜单字段参考
- 面板字段参考
- 面板定义参考
- 自定义界面元素定义参考
- 常用 ipc 消息参考
- meta 文件参考
- UI Kit
  - 控件
    - ui-button
    - ui-checkbox
    - ui-color
    - ui-input
    - ui-num-input
    - ui-select
    - ui-slider
    - ui-text-area
  - 特殊控件
    - ui-asset
    - ui-node
  - 控件容器
    - ui-box-container
    - ui-prop
    - ui-section
  - 视图元素
    - ui-hint
    - ui-loader
    - ui-markdown
    - ui-progress

# 你的第一个扩展包

本文会教你如何创建一个简单的 Cocos Creator 扩展包，并且向你介绍一些扩展包中的基本概念。通过学习，你将会创建一个扩展包，并在主菜单中建立一个菜单项，并通过该菜单项在主进程中执行一条扩展指令。

## 创建并安装扩展包

创建一个空文件夹命名为 “hello-world”，并在该文件夹中创建 `main.js` 和 `package.json` 两个文本文件。该扩展包的结构大致如下：

```
hello-world
  |--main.js
  |--package.json
```

将该文件夹放入到 `~/.CocosCreator/packages`（Windows 用户为 `C:\Users\${你的用户名}\.CocosCreator\packages`），或者放入到 `${你的项目路径}/packages` 文件夹下即可完成扩展包的安装。

## 定义你的包描述文件：package.json

每个包都需要一份 `package.json` 文件去描述他的用途，这样 Cocos Creator 编辑器才能知道这个包要扩展什么，从而正确加载。值得一提的是，虽然 `package.json` 在很多字段上的定义和 node.js 的 npm-package 相似，他们仍然是为不同的产品服务的，所以从 npm 社区中下载的包，并不能直接放入到 Cocos Creator 中变成插件。

我们在这里做一份简单的 `package.json`：

```
{
  "name": "hello-world",
  "version": "0.0.1",
  "description": "一份简单的扩展包",
  "author": "Cocos Creator",
  "main": "main.js",
  "main-menu": {
    "Packages/Hello World": {
      "message": "hello-world:say-hello"
    }
  }
}
```

解释：

- `name` String - 定义了包的名字，包的名字是全局唯一的，他关系到你今后在官网服务器上登录时的名字。
- `version` String - 版本号，我们推荐使用 semver 格式管理你的包版本。
- `description` String（可选） - 一句话描述你的包是做什么的。
- `author` String（可选） - 扩展包的作者
- `main` String (可选) - 入口程序
- `main-menu` Object (可选) - 主菜单定义

## 入口程序

当你定义好你的描述文件以后，接下来就要书写你的入口程序 `main.js` 了。定义如下：

```
'use strict';

module.exports = {
  load () {
    // 当 package 被正确加载的时候执行
  },

  unload () {
    // 当 package 被正确卸载的时候执行
  },

  messages: {
    'say-hello' () {
      Editor.log('Hello World!');
    }
  },
};
```

这份入口程序会在 Cocos Creator 的主进程中被加载，在加载成功后，他会调用入口程序中的 `load` 函数。并且会将定义在 `messages` 字段中的函数注册成 IPC 消息。更多关于入口函数中的消息注册，以及 IPC 消息的内容，我们会在 IPC 简介 中讲解。

这里我们只要明白，入口函数中的 `messages` 字段中的函数，将会在主进程的 IPC 监听模块中，注册一份消息，其格式为 `${扩展包名}:${函数名}`，并将对应的函数作为 IPC 响应的函数。

# 运行扩展包程序

现在你可以打开你的 Cocos Creator，你将会发现你的主菜单中多出了一份 `Packages` 的菜单，点击 `Packages` 菜单中的 `Hello World` 将会发送一个消息 “hello-world:say-hello” 给我们的扩展包的 `main.js`， 它会在 Creator 的控制台中打印出 “Hello World” 的日志信息。

恭喜你完成了第一个简单的编辑器扩展工具。

# 安装与分享

当你启动 Cocos Creator 并且打开了一个指定项目后，Cocos Creator 会开始搜索并加载扩展包。Cocos Creator 有两个扩展包搜索路径，“全局扩展包路径”和“项目扩展包路径”。

## 从扩展商店获取扩展插件

点击主菜单的 `扩展/扩展商店`，即可打开扩展商店。

在扩展商店里可以搜索或浏览不同类别的插件，Cocos Creator 的编辑器扩展插件会归类到 `Creator扩展` 里。

## 全局扩展包路径

当你打算将你的扩展应用到所有的 Cocos Creator 项目中，你可以选择将扩展包安装到全局扩展包路径中。根据你的目标平台，全局扩展包路径将会放在：

- **Windows** `%USERPROFILE%\.CocosCreator\packages`
- **Mac** `$HOME/.CocosCreator/packages`

## 项目扩展包路径

有时候我们只希望某些扩展功能被应用于指定项目中，这个时候我们可以将扩展包安装到项目的扩展包存放路径上。项目的扩展包路径为 `$你的项目地址/packages` 。

## 扩展包包名和目录名

扩展包的包名 ( `package.json` 声明中的 `name` 字段的内容) 最好和扩展包所在路径的路径名一致，比如前面第一个扩展包 `hello-world`，如果放在项目扩展包路径中，其路径结构应该是：

```
MyProject
 |--assets
 |--packages
    |--hello-world
       |--package.json
       |--main.js
```

这样我们在编写扩展包逻辑时可以更方便的获取到扩展包所在路径下的文件 url，减少出错的可能。

## 开发扩展包时的实时改动监控

在开发扩展包的过程中，编辑器进程会对扩展包里的脚本内容进行监控，当有脚本内容发生变化时，会自动对扩展包进行重新载入。文件监控的规则可以在 `package.json` 中定制，详情请阅读 package.json字段：reload 。

# IPC 简介

在学习 Cocos Creator 的插件编写之前，我们先要理解 Cocos Creator 插件开发中的重要一环，进程间通信（IPC）。Cocos Creator 的编辑器是基于 GitHub 开发的 Electron 内核。Electron 是一个集成了 Node.js 和 Chromimu 的跨平台开发框架。

在 Electron 的架构中，一份应用程序由主进程和渲染进程组成，其主进程负责管理平台相关的调度，如窗口的开启关闭，菜单选项，基础对话框等等。而每一个新开启的窗口就是一个独立的渲染进程。在 Electron 中，每个进程独立享有自己的 JavaScript 内容，彼此之间无法直接访问。当我们需要在进程之间传递数据时，就需要使用进程间通信（IPC）。 你可以通过阅读 Electron's introduction document 更深入的理解 Electron 中的主进程和渲染进程的关系。简单点说，Electron 的主进程相当于一个 Node.js 服务端程序，而每一个窗口（渲染进程）则相当于一份客户端网页程序。

Cocos Creator 沿用了 Electron 的主进程和渲染进程的设计结构，在 Creator 启动前，我们将许多服务如：资源数据库，脚本编译器，预览服务器，打包工具等在主进程中开启，之后我们在主窗口也就是渲染进程中启动编辑器的界面操作部分。当界面操作需要使用主进程的模块时，我们就会进行进程间的通信（IPC）来完成请求。

## 进程间通信（IPC）

前面我们已经说到了两个进程之间的 JavaScript 内容是相互独立的，必须靠进程间通信的方式来交换数据。进程间通信实际上就是在一个进程中发消息，然后在另外一个进程中监听消息的过程。 Electron 为我们提供了进程间通信对应的模块 ipcMain 和 ipcRenderer 来帮助我们完成这个任务。由于这两个模块仅完成了非常基本的通信功能，并不能满足编辑器，插件面板与主进程之间的通信需求，所以 Cocos Creator 在这之上又进行了封装，扩展了进程间消息收发的方法，方便插件开发者和编辑器开发者制作更多复杂情景。

## IPC 消息命名规范

IPC 的消息名就是一个字符串，对于消息的发送者，可以自由的进行消息的命名。但是我们希望开发人员在 Cocos Creator 中定制消息时遵守一定的规范，防止混乱。 消息的命名规范为：

```
'module-name:action-name'
// or
'package-name:action-name'
```

你可以在扩展程序中定义自己的的 IPC 消息，也可以监听编辑器内置的各个 IPC 消息，请参考编辑器内置的 IPC 消息列表。

## 扩展包中的进程关系

Cocos Creator 的扩展包也分为主进程和渲染进程两个部分。我们在上一篇文章中已经介绍了扩展包的入口程序，它是一份跑在主进程的脚本程序（就是前面声明过的入口程序 `main.js`，下一节会对 入口程序 进行详细介绍），如果你的扩展包需要用户界面，那么还需要一个或多个渲染进程 （会在后面的 扩展编辑器面板 介绍）。

# 入口程序

每一个插件都可以指定一份入口程序，入口程序是在 **主进程** 中被执行的。通常我们会在入口程序中：

- 初始化扩展包
- 执行后台操作程序（文件I/O，服务器逻辑）
- 调用 Cocos Creator 主进程中的方法
- 管理扩展面板的开启和关闭，以及响应主菜单和其他面板发送来的 IPC 消息

这里有一份入口程序的最简单样例：

```
'use strict';

module.exports = {
  load () {
    console.log('package loaded');
  },

  unload () {
    console.log('package unloaded');
  },
};
```

# 生命周期回调

## load

当扩展包正确载入后，将会执行用户入口程序中的 load 函数。我们可以在这里做一些关于扩展包本身的初始化操作。

## unload

当扩展包卸载进行到最后阶段，将会执行用户入口程序中的 unload 函数。我们可以在这里做一些扩展包卸载前的清理操作。

## 加载和卸载注意事项

Cocos Creator 支持在编辑器运行时动态的添加和删除扩展包，所以要注意如果扩展包依赖编辑器其他模块的特定工作状态时，必须在 `load` 和 `unload` 里进行妥善处理。如果插件的动态加载和卸载导致其他模块工作异常时，扩展包的用户总是可以选择关闭编辑器后重新启动。

# IPC 消息注册

在入口程序中添加 `messages` 字段，可以让扩展包在加载的时候进行主进程的 IPC 消息注册。样例如下：

```
'use strict';

module.exports = {
  messages {
    'foo-bar' ( event ) { console.log('hello foobar'); },
    'scene:saved' ( event ) { console.log('scene saved!'); },
  },
};
```

通过上面的例子，我们可以看到注册的 IPC 消息接受两种格式：

## 短命名消息

短命名消息是指消息名不带 `:` 的消息。这些消息将被视为该扩展包内的消息，在注册阶段，实际注册是，会被写成 `${你的扩展包名}:${消息名}` 。以上面的代码为例，假设我们的扩展包名字为 `simple-demo`，那么 `foo-bar` 这个消息实际注册时，将会扩展成 `simple-demo:foo-bar` 。

实际应用中，我们就可以通过 `Editor.sendToPackage` 函数发送 IPC 消息到主进程的指定扩展包的注册函数中。

```
Editor.sendToPackage('simple-demo', 'foo-bar');
```

当然，我们还有更多的消息发送策略，我们会在后续的章节中进行详细介绍。

## 全名消息

全名消息就是指消息名中带有 `:` 的消息命名方式。通常我们书写全名消息是为了监听其他扩展包，或者其他模块的 IPC 消息。通过全名消息很清晰地知道这份消息是从哪个扩展包中发出的，也更好的避免了消息冲突的问题。

如上面的例子，我们可以清楚的了解到，“scene:saved” 这个消息是从 scene 这个内置扩展中发送的。

# 扩展包工作流程模式

在设计和开发扩展包时，我们总是希望扩展包在我们给予一定的输入时，完成特定的工作并返回结果。这个过程可以由以下几种工作模式来完成：

## 入口程序完成全部工作

如果我们的插件不需要任何用户输入，而且只要一次性的执行一些主进程逻辑，我们可以将所有工作放在 `main.js` 的 `load` 生命周期回调里：

```
// main.js
module.exports = {
  load () {
    let fs = require('fs');
    let path = require('path');
    // 插件加载后在项目根目录自动创建指定文件夹
    fs.mkdirSync(Path.join(Editor.projectPath, 'myNewFolder'));
    Editor.success('New folder created!');
  }
}
```

如果你的插件会自动完成工作，别忘记通过 `Editor.log`，`Editor.success` 接口（上述接口可以在 Console API 查看详情），来告诉用户刚刚完成了哪些工作。

示例中使用到的 `Editor.projectPath` 接口会返回当前打开项目的绝对路径，详情可以在 Editor API 中找到。

这种工作模式的更推荐的变体是将执行工作的逻辑放在菜单命令后触发，如 第一个扩展包 文档所示，我们在 `package.json` 里定义了 `main-menu` 字段和选择菜单项后触发的 IPC 消息，之后就可以在入口程序里监听这个消息并开始实际的业务逻辑：

```
  messages: {
    'start' () {
      //开始工作!
    }
  }
```

关于菜单命令的声明，请参考 扩展主菜单。

## 入口程序和编辑器面板配合实现复杂交互功能

入口程序除了可以在主进程执行 Node.js 所有标准接口以外，还可以打开编辑器面板、窗口，并通过 IPC 消息在主进程的入口程序和渲染进程的编辑器面板间进行通讯，通过编辑器面板和用户进行复杂的交互，并在相关的进程中完成业务逻辑的处理。

要通过入口程序打开一个编辑器面板：

```
  messages: {
    'open' () {
      // open entry panel registered in package.json
      Editor.Panel.open('myPackage');
    }
  }
```

其中 `myPackage` 是面板的 ID，在单面板的扩展程序中，这个面板 ID 和扩展包名是一致的。用户可以通过 `package.json` 里的 `panel` 字段声明自定义的编辑器面板。我们会在下一节的 扩展编辑器面板 文档中进行详细介绍。

启动面板后，在主进程和面板渲染进程间就可以通过 `Editor.Ipc.sendToPanel`， `Editor.Ipc.sendToMain` 等方法来进行进程间通讯，我们会在后面的文章中进行详细介绍。

# 插件只提供组件和资源

由于 Cocos Creator 本身采用的组件系统就有很高的扩展性和复用性，所以一些运行时相关的功能可以通过单纯开发和扩展组件的形式完成，而扩展包可以作为这些组件和相关资源（如 Prefab、贴图、动画等）的载体。通过扩展包声明字段 `runtime-resource` 可以将扩展包目录下的某个文件夹映射到项目路径下，并正确参与构建和编译等流程：

```
//package.json
  "runtime-resource": {
    "path": "path/to/runtime-resource",
    "name": "shared-resource"
  }
```

上述声明会将 `projectPath/packages/myPackage/path/to/runtime-resource` 路径下的全部资源都映射到资源管理器中，显示为 `[myPackage]-[shared-resource]`。

这个路径下的内容（包括组件和其他资源）可以由项目中的其他场景、组件引用。使用这个工作流程，开发者可以将常用的控件、游戏架构以插件形式封装在一起，并在多个项目之间共享。

更多信息请阅读 runtime-resource 字段参考。

# 扩展主菜单

Cocos Creator 的主菜单是可以自由扩展的。扩展方法为在 `package.json` 文件中的 `main-menu` 字段里，加入自己的菜单路径和菜单设置选项。下面是一份主菜单的配置样例：

```
{
  "main-menu": {
    "Examples/FooBar/Foo": {
      "message": "my-package:foo"
    },
    "Examples/FooBar/Bar": {
      "message": "my-package:bar"
    }
  }
}
```

在样例中，我们通过配置菜单路径，在主菜单 "Example" > "Foobar" 里加入了 "Foo" 和 "Bar" 两个菜单选项。当我们点击这个菜单项，他会发送定义在其 `message` 字段中的 IPC 消息到主进程中。比如我们点击 "Foo" 将会发送 `my-package:foo` 这个消息。

# 菜单路径

在主菜单中注册中，其键值（Key）需要是一份菜单路径。菜单路径是 posix 路径格式，使用 `/` 作为分割符。在主菜单注册过程中，Cocos Creator 会根据路径一级一级往下寻找子菜单项，如果在寻找路径中没有找到对应的子菜单，就会在对应的路径中进行创建，直到最后一级菜单，将被当做菜单项加入。

在注册过程中也会遇到一些出错的情况：

## 注册的菜单项已经存在

这种情况多发生在多个扩展包之间，当你的扩展包和其他用户的扩展包的菜单注册路径相同时，就会发生该冲突。

## 注册菜单项的父级菜单已经被其他菜单项注册，其类型不是一个子菜单（submenu）

这个情况和上一种情况类似，我们可以用以下这个例子来说明：

```
{
  "main-menu": {
    "Examples/FooBar": {
      "message": "my-package:foo"
    },
    "Examples/FooBar/Bar": {
      "message": "my-package:bar"
    }
  }
}
```

在这个例子中，我们先在主菜单中注册了一份菜单路径 “Examples/Foobar”，这之后我们有注册了 “Examples/Foobar/Bar”，而第二个菜单路径的注册要求 Foobar 的类型为一个分级子菜单（submenu），然而由于上一次的注册已经将 Foobar 的类型定义为菜单选项（menu-item），从而导致了注册失败。

# 菜单选项

在上面的例子中，我们已经使用了 `message` 菜单选项。菜单注册过程中还有许多其他的可选项，例如：icon、accelerator、type 等。更多选项，请阅读主菜单字段参考。

# 插件专用菜单分类

为了避免用户安装多个插件时，每个插件随意注册菜单项，降低可用性，我们推荐所有编辑器扩展插件的菜单都放在统一的菜单分类里，并以插件包名对不同插件的菜单项进行划分。

插件专用的菜单分类路径是 `i18n:MAIN_MENU.package`，在中文语言环境会显示为一级菜单 `插件`，接下来的二级菜单路径名应该是插件的包名，最后的三级路径是具体的菜单项功能，如：

```
i18n:MAIN_MENU.package/FooBar/Bar
```

在中文环境的编辑器下就会显示如 `插件/FooBar/Bar` 这样的菜单。

`i18n:MAIN_MENU.package` 是多语言专用的路径表示方法，详情请见 扩展包多语言化 文档。

# 扩展编辑器面板

Cocos Creator 允许用户定义一份面板窗口做编辑器的 UI 交互。

## 定义方法

在插件的 package.json 文件中定义 `panel` 字段如下:

```
{
  "name": "simple-package",
  "panel": {
    "main": "panel/index.js",
    "type": "dockable",
    "title": "Simple Panel",
    "width": 400,
    "height": 300
  }
}
```

目前编辑器扩展系统支持每个插件注册一个面板，面板的信息通过 `panel` 字段对应的对象来申明。其中 `main` 字段用来标记面板的入口程序，和整个扩展包的入口程序概念类似， `panel.main` 字段指定的文件路径相当于扩展包在渲染进程的入口。

另外值得注意的是 `type` 字段规定了面板的基本类型:

- `dockable`： 可停靠面板，打开该面板后，可以通过拖拽面板标签到编辑器里，实现扩展面板嵌入到编辑器中。下面我们介绍的面板入口程序都是按照可停靠面板的要求声明的。
- `simple`： 简单 Web 面板，不可停靠到编辑器主窗口，相当于一份通用的 HTML 前端页面。详细情况请见 定义简单面板。

其他面板定义的说明请参考 面板字段参考。

## 定义入口程序

要定义一份面板的入口程序，我们需要通过 `Editor.Panel.extend()` 函数来注册面板。如以下代码:

```
// panel/index.js
Editor.Panel.extend({
  style: `
    :host { margin: 5px; }
    h2 { color: #f90; }
  `,

  template: `
    <h2>标准面板</h2>
    <ui-button id="btn">点击</ui-button>
    <hr />
    <div>状态: <span id="label">--</span></div>
  `,

  $: {
    btn: '#btn',
    label: '#label',
  },

  ready () {
    this.$btn.addEventListener('confirm', () => {
```

```
      this.$label.innerText = '你好';
      setTimeout(() => {
        this.$label.innerText = '--';
      }, 500);
    });
  },
});
```

`Editor.Panel.extend()` 接口传入的参数是一个包括特定字段的对象，用来描述整个面板的外观和功能。

在这份对象代码中，我们定义了面板的样式（style）和模板（template），并通过定义选择器 `$` 获得面板元素，最后在 ready 初始化回调函数中中对面板元素的事件进行注册和处理。

在完成了上述操作后，我们就可以通过在主进程（入口程序）调用 `Editor.Panel.open('simple-package')` 激活我们的面板窗口。 关于 `Editor.Panel` 接口的用法请参考 Panel API。

更多关于面板定义对象字段的说明，请阅读面板定义参考。

# 在主菜单中添加打开面板选项

为了方便我们打开窗口，通常我们会将打开窗口的方法注册到主菜单中，并通过发消息给我们的插件主进程代码来完成。 要做到这些事情，我们需要在我们的 package.json 中注册主进程入口函数和主菜单选项:

```
{
  "name": "simple-package",
  "main": "main.js",
  "main-menu": {
    "Panel/Simple Panel": {
      "message": "simple-package:open"
    }
  },
  "panel": {
    "main": "panel/index.js",
    "type": "dockable",
    "title": "Simple Panel",
    "width": 400,
    "height": 300
  }
}
```

在主进程函数中，我们做如下定义：

```
'use strict';

module.exports = {
  load () {
  },

  unload () {
  },

  messages: {
    open() {
      Editor.Panel.open('simple-package');
    },
  },
};
```

一切顺利的话，你将可以通过主菜单，打开如下的面板：


更多关于在 `package.json` 文件中注册面板时的字段描述，请阅读面板字段参考。

# 窗口面板与主进程交互

通常我们需要在窗口面板中设置一些 UI，然后通过发送 IPC 消息将任务交给主进程处理。这里我们可以通过 `Editor.Ipc` 模块来完成。在我们上面定义的 index.js 中，我们可以通过在 `ready()` 函数中处理 按钮消息来达成。

```
this.$btn.addEventListener('confirm', () => {
  Editor.Ipc.sendToMain('simple-package:say-hello', 'Hello, this is simple panel');
});
```

当你点击按钮时，他将会给插件主进程发送 'say-hello' 消息，并附带对应的参数。你可以用任何你能想得到的前端 技术编辑你的窗口界面，还可以结合 Electron 的 内置 node 在窗口内 require 你希望的 node 模块，完成 任何你希望做的操作。

---

更全面和详细的主进程和面板之间的 IPC 通讯交互方法，请继续阅读 进程间通讯工作流程。

# 进程间通讯（IPC）工作流程

关于扩展包进程间通讯（以下简称 IPC）的基本概念，请先阅读 IPC 简介。

我们前面介绍了主进程中的 入口程序 和渲染进程中的 面板程序 的基本声明方法和交互方式，接下来我们将结合实际需求介绍两种进程间通讯的详细工作流程。

本节提及的所有相关 API 均可查询 Editor.Ipc 主进程 API 和 Editor.Ipc 渲染进程 API。

# 发送消息

## 主进程向面板发送消息

在主进程中，主要使用

```
Editor.Ipc.sendToPanel('panelID', 'message' [, ...args, callback, timeout)
```

接口向特定面板发送消息。对于目前支持的单面板插件来说：

- `panelID` 面板 ID，对于单面板扩展包来说，面板 ID 就是插件的包名，如 `foobar`
- `message` 是 IPC 消息的全名，如 `foobar:do-some-work`，我们推荐在定义 IPC 消息名时使用 `-` 来连接单词，而不是使用驼峰或下划线。
- **可选** `args`，从第三个参数开始，可以定义数量不定的多个传参，用于传递更具体的信息到面板进程。
- **可选** `callback`，在传参后面可以添加回调方法，在面板进程中接受到 IPC 消息后可以选择向主进程发送回调，并通过 callback 回调方法进行处理。回调方法的参数第一个是 error（如果没有错误则传入 `null`），之后才是传参。
- **可选** `timeout`，回调超时，只能配合回调方法一起使用，如果规定了超时，在消息发送后的一定时间内没有接到回调方法，就会触发超时错误。如果不指定超时，则默认的超时设置是 5000 毫秒。

## 面板向主进程发送消息

```
Editor.Ipc.sendToMain('message', [, ...args, callback, timeout])
```

和 `sendToPanel` 相比，除了缺少第一个面板 ID 参数之外，其他参数的意义和用法完全相同。

## 其他消息发送方法

主进程对面板和面板对主进程是最常见的两种消息发送方式，但实际上 IPC 不局限于不同的两类进程之间，我们可以把消息发送方式做以下归类：

- 任意进程对主进程 `Editor.Ipc.sendToMain`
- 任意进程对面板 `Editor.Ipc.sendToPanel`
- 任意进程对编辑器主窗口（也就是对主窗口里的所有渲染进程广播） `Editor.Ipc.sendToMainWin`
- 任意进程对所有窗口（对包括弹出窗口在内的所有窗口渲染进程广播） `Editor.Ipc.sendToWins`
- 任意进程对所有进程广播 `Editor.Ipc.sendToAll`

上述方法在两种进程里写法都是一致的，只要注意消息接收的对象是在渲染进程还是主进程，并选择对应的方法即可。详细的接口用法请参考上文的描述和本文最上面的 IPC 接口文档链接。

# 接收消息

要在主进程或渲染进程中接受 IPC 消息，最简单的办法是在声明对象的 `messages` 字段中注册以 IPC 消息为名的消息处理方法。

## 面板渲染进程消息监听

```
//packages/foobar/panel/index.js
Editor.Panel.extends({
    //...
    messages: {
        'my-message': function (event, ...args) {
            //do some work
        }
    }
});
```

## 主进程消息监听

```
//packages/foobar/main.js
module.exports = {
    //...
    messages: {
        'my-message': function (event, ...args) {
            //do some work
        }
    }
}
```

注册监听消息时，我们使用的消息名是省略了扩展包名的短命名，上述消息短名 `my-message` 在发送时应该是 `Editor.sendToPanel('foobar:my-message')` 和 `Editor.sendToMain('foobar:my-messages')` 。

可以看到主进程和渲染进程中监听 IPC 消息的函数声明方式是一致的，传入的第一个参数是一个 `event` 对象，我们可以通过这个对象发送回调。

## 其他消息监听方式

除了在 `messages` 字段内注册之外还可以使用 Electron 的 Ipc 消息接口来监听，形式上更灵活：

渲染进程中：

```
require('electron').ipcRenderer.on('foobar:message', function(event, args) {});
```

主进程中：

```
require('electron').ipcMain.on('foobar:message', function(event, args) {});
```

关于 Electron 的 IPC 接口可以参考 Electron API: ipcMain Electron API: ipcRenderer。

# 向消息来源发送回调

假如我们从主进程发送了一个消息：

```
//packages/foobar/main.js
Editor.Ipc.sendToPanel('foobar', 'greeting', 'How are you?', function (error, answer) {
    Editor.log(answer);
});
```

在面板监听消息的方法中，我们可以使用 `event.reply` 来发送回调：

```
//packages/foobar/panel/index.js
Editor.Panel.extends({
    //...
    messages: {
        'greeting': function (event, question) {
            console.log(question); //How are you?
            if (event.reply) {
                //if no error, the first argument should be null
                event.reply(null, 'Fine, thank you!');
            }
        }
    }
});
```

注意 `event.reply` 第一个参数是报错，没有错误时应该传入 `null`，此外建议总是检查 `event.reply` 是否存在，如果发送消息时参数中不包含回调方法，则 `event.reply` 的检查将返回 `undefined`，这种情况下调用 `event.reply` 会产生错误。

## 处理超时

发送消息时的最后一个参数是超时时限，单位是毫秒，如果未指定超时时限，则使用默认的 5000 ms 超时限制。

如果要取消超时限制，最后一次参数应该传入 `-1`，这种情况下应该靠其他逻辑保证回调必将触发。

从消息发送开始，在超过规定的时限后仍然没有接到消息监听方法中返回的回调的话，就会收到系统发送的超时错误回调：

```
Editor.Ipc.sendToMain('foobar:greeting', function (error, answer) {
  if ( error.code === 'ETIMEOUT' ) { //check the error code to confirm a timeout
    Editor.error('Timeout for ipc message foobar:greeting');
    return;
  }
  Editor.log(answer);
});
```

# 定义简单面板

有时候我们希望扩展的面板并不用停靠在主窗口中，而是希望他是一个独立窗口，利用标准的 HTML 页面载入 方式展现。这个时候我们可以考虑使用 `simple` 类型的面板窗口。

# 定义方法

在插件的 package.json 文件中定义 `panel` 字段如下:

```
{
  "name": "simple-package",
  "panel": {
    "main": "panel/index.html",
    "type": "simple",
    "title": "Simple Panel",
    "width": 400,
    "height": 300
  }
}
```

通过定义 `panel` 字段，并申明面板 `type` 为 `simple` 我们即可获得该份面板窗口。通过定义 `main` 字段我们可以为我们的面板窗口置顶一份入口 html。

和 扩展编辑器面板 中介绍的面板定义的区别在于， `type` 使用 `simple` ，而 `main` 索引的是 html 文件而不是 javascript 文件。

接下来我们可以定义一份简单的 html 入口，就和我们写任何网页一样:

```
<html>
  <head>
    <title>Simple Panel Window</title>
    <meta charset="utf-8">
    <style>
      body {
        margin: 10px;
      }

      h1 {
        color: #f90
      }
    </style>
  </head>

  <body>
    <h1>A simple panel window</h1>
    <button id="btn">Send Message</button>

    <script>
      let btn = document.getElementById('btn');
      btn.addEventListener('click', () => {
        Editor.log('on button clicked!');
      });
    </script>
  </body>
</html>
```

在完成了上述操作后，我们就可以通过 `Editor.Panel.open('simple-package')` 激活我们的窗口。

使用简单窗口更接近于纯粹的网页编程，也更适合那些将已有 Web APP 移植或整合到 Cocos Creator 中的情况。

# 多语言化 (i18n)

编辑器扩展系统中内置的多语言方案允许你配置多份语言的键值映射，并根据编辑器当前的语言设置在插件界面显示不同语言的文字。

要启用多语言功能（以下简称 i18n），请在扩展包的目录下新建一个名叫 `i18n` 的文件夹，并为每种语言添加一个相应的 JavaScript 文件，作为键值映射数据。数据文件名应该和语言的代号一致，如 `en.js` 对应英语映射数据。

下面是键值映射数据源的例子：

```
// en.js
module.exports = {
  'search': 'Search',
  'edit': 'Edit',
};

// zh.js
module.exports = {
  'search': '搜索',
  'edit': '编辑',
};
```

该文件将会将包含的映射注册到 `i18n` 的全局表里以扩展包名命名的部分里，假设包名为 `foobar`，则对应搜索文字的完整 key 为 `foobar.search`。

# 显示对应语言的文本

## 在脚本中使用

在 JavaScript 脚本中，你可以通过 `Editor.T` 接口获取当前语言对应的翻译后的文本：

```
// NOTE: my package name is "foobar"
Editor.T('foobar.search');
```

在编辑器面板的模板定义文件里，也可以直接使用这个接口：

```
// NOTE: my package name is "foobar"
Editor.Panel.extend({
  template: `
    <div class="btn">${Editor.T('foobar.edit')}</div>
  `
});
```

## 在菜单项中使用

在扩展包的 `package.json` 中注册菜单路径时支持 i18n 格式的路径，该类路径以 `i18n:${key}` 的形式表示。我们可以写 `i18n:MAIN_MENU.package.title/foobar/i18n:foobar.edit`，Cocos Creator 会帮助我们查找对应的 i18n 字符串并进行替换。

其中 `MAIN_MENU.package.title` 是 Cocos Creator 内置主菜单里的一级菜单名，为了方便大家使用插件，请尽量将扩展包的菜单注册到这个一级菜单下面，上述菜单路径变成用户实际看到的菜单路径就是 `扩展/foobar/编辑`。

# 工作路径和常用 URL

## 访问项目路径

- `Editor.projectPath` （主进程）当前在编辑器打开项目的根目录绝对路径。

## 自定义协议 URL

由于主进程和渲染进程在路径查询上有复杂的差异，我们引入了一些自定义协议的 URL 来方便的访问各个不同模块和路径的文件

- `db://` 在 管理项目资源 中介绍过，这个协议会映射到项目根目录，可以直接写 `db://assets/script/MyScript.js` 来获取特定的项目文件。注意在运行编辑器时的插件加载阶段，不能使用 `Editor.url('db://')`，这个阶段还没有初始化项目路径，要在编辑器窗口初始化完毕，项目文件全部加载后才能使用。
- `packages://` 映射到项目本地的插件目录 `packages` 和全局的插件目录 `$HOME/.CocosCreator/packages`，也就是说在这两个目录下的任何扩展包和其中的文件都可以通过这个协议索引，如 `packages://foobar/package.json` 就表示 `foobar` 这个扩展包中的配置文件。
- `unpack://` 访问 Cocos Creator 安装目录下的开源内容，包括
  - `unpack://engine` JavaScript 引擎路径
  - `unpack://cocos2d-x` C++ 引擎路径
  - `unpack://simulator` 模拟器路径

要将这些自定义协议 URL 转换为绝对路径，使用 `Editor.url()` 接口。

### 声明面板时使用独立的 HTML 和 CSS 文件

使用 `Editor.url` 配合插件路径，我们就可以在声明面板的时候读取其他文件里包含的 HTML 和 CSS 定义，如：

```
var Fs = require('fs');
Editor.Panel.extend({
  // css style for panel
  style: Fs.readFileSync(Editor.url('packages://foobar/panel/index.css', 'utf8')),

  // html template for panel
  template: Fs.readFileSync(Editor.url('packages://foobar/panel/index.html', 'utf8')),
  //...
});
```

# 提交插件到商店

Cocos Creator 内置了插件商店，可供用户浏览、下载和自动安装第三方插件。插件商店的使用方法可见 安装与分享。要分享或贩卖编辑器扩展包，需要将插件提交到插件商店，下面是提交流程。

# 打包插件

假设开发完成的插件目录结构如下：

```
foobar
    |--panel
    |    |--index.js
    |--package.json
    |--main.js
```

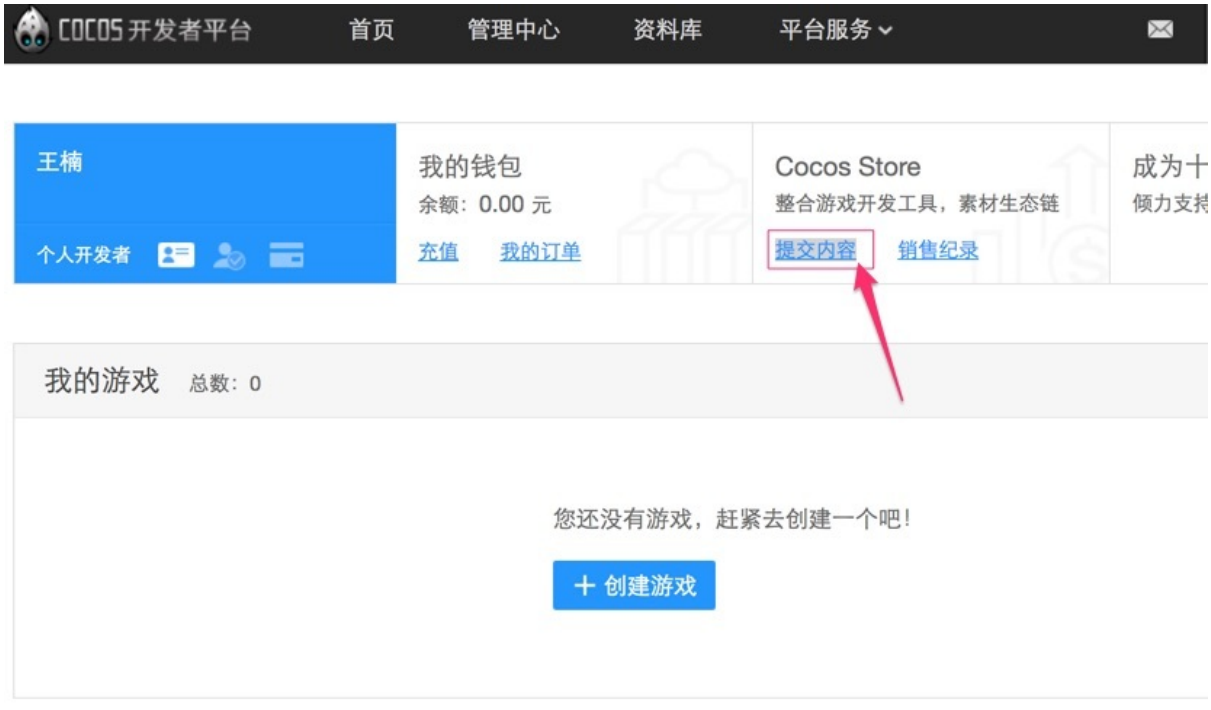
插件开发者需要将 `foobar` 文件夹打包成 `foobar.zip` 文件。

## NPM 第三方库

目前扩展包安装系统中没有包括安装 NPM 等包括管理系统的工作流程，因此使用了第三方库的扩展包应该将 `node_modules` 等文件夹也一起加入到 zip 包中。

# 登入开发者平台

- 访问 Cocos 开发者平台 并登录
- 进入 管理中心
- 在 Cocos Store 下面点击 提交内容



# 创建插件

进入 Cocos Store 页面后，点击右上方的 `提交内容` 按钮，进入插件提交页面。

- 首先为插件取一个中文名称
- 选择类别为 `Creator插件`
- 点击创建，进入信息完善界面

# 完善插件信息

这里需要填写插件信息表单，其中需要注意的有：

- **版本号** 书写规范请遵守 semver 规范
- **价格** 单位是 RMB，如果免费请填写 `0`
- **描述** 请填写基本功能和使用方法
- **下载地址** 可以有两种方式提供插件下载：1. 自己上传到可公开下载的网盘或使用 github 的下载链接 2. 手动上传到 Cocos Store，上传后会自动生成下载链接
- **图标** 正方形图标，推荐大小 512x512
- **截图** 最多上传5张，尺寸为 640x960 或者 960x640
- **支持链接** 填写插件首页的地址，或论坛讨论帖地址，方便用户获取帮助和技术支持

完成填写后点击 `提交审核` 即可，插件商店的管理人员会在审核插件内容和信息后：

- 如果没有问题，会将插件上架
- 如果有问题需要修改，会通过 Cocos 开发者平台的注册邮箱和提交者取得联系

# 调用引擎 API 和项目脚本

在插件中可以声明一个特殊的脚本文件（场景脚本），该脚本和项目中的脚本 （`assets` 目录下的脚本）具有相同的环境，也就是说在这个脚本里可以调用引擎 API 和其他项目脚本，实现：

- 遍历场景中的节点，获取或改动数据
- 调用项目中的其他脚本完成工作

## 注册场景脚本

首先在 `package.json` 里添加 `scene-script` 字段，该字段的值是一个脚本文件的路径，相对于扩展包目录：

```
    "name": "foobar",
    "scene-script": "scene-walker.js"
```

该路径将指向 `packages/foobar/scene-walker.js`，接下来我们看看如何编写场景脚本。

## 编写场景脚本

`scene-walker.js` 需要用这样的形式定义：

```
module.exports = {
    'get-canvas-children': function (event) {
        var canvas = cc.find('Canvas');
        Editor.log('children length : ' + canvas.children.length);

        if (event.reply) {
            event.reply(canvas.children.length);
        }
    }
};
```

可以看到场景脚本由一个或多个 IPC 消息监听方法组成，收到相应的 IPC 消息后，我们在函数体内可以使用包括全部引擎 API 和用户组件脚本里声明的方法和属性。

## 从扩展包中向场景脚本发送消息

接下来在扩展包程序的主进程和渲染进程中，都可以使用下面的接口来向 `scene-walker.js` 发送消息（假设扩展包名是 `foobar`）：

```
Editor.Scene.callSceneScript('foobar', 'get-canvas-children', function (err, length) {
    console.log(`get-canvas-children callback :  length - ${length}`);
});
```

这样就可以在扩展包中获取到场景里的 `Canvas` 根节点有多少子节点，当然还可以用来对场景节点进行更多的查询和操作。

在发送消息时 `callSceneScript` 接受的参数输入和其他 IPC 消息发送接口一致，也可以指定更多传参和 timeout 超时时限。详情请看 IPC 工作流程。

# 在场景脚本中引用模块和插件脚本

除了通过 `cc.find` 在场景脚本中获取特定节点，并操作该节点和挂载的组件以外，我们还可以引用项目中的非组件模块，或者通过全局变量访问插件脚本。

## 引用模块

```
//MyModule.js
module.exports = {
    init: function () {
        //do initialization work
    }
}

//scene-walker.js
module.exports = {
    'init-module': function (event) {
        var myModule = cc.require('MyModule');
        myModule.init();
    }
};
```

注意，要使用和项目脚本相同的模块引用机制，在场景脚本里必须使用 `cc.require` 的写法。

## 引用插件脚本

直接使用 `window.globalVar` 来访问插件脚本里声明的全局变量和方法即可。

# 管理项目资源

## 管理场景

### 保存当前场景

在上一节 调用引擎 API 和项目脚本 中我们介绍了通过场景脚本访问引擎 API 和用户项目脚本的方法，在对场景数据进行修改后可以使用以下接口保存当前场景。

`_Scene.save()`

其中 `_Scene` 是一个特殊的单例，用来控制场景编辑器里加载的场景实例。

### 加载其他场景

我们的扩展包可能需要遍历多个场景并依次操作和保存，要加载新场景，请使用

```
_Scene.loadSceneByUuid(uuid, function(error) {
    //do more work
});
```

传入的参数是场景资源的 uuid，可以通过下面介绍的资源管理器接口来获取。

## 资源 URL 和 UUID 的映射

在 Cocos Creator 编辑器和扩展中，资源的 url 由形如

`db://assets/path/to/scene.fire`

这样的形式表示。其中 `db` 是 AssetDB 的简称。 项目中 `assets` 路径下的全部资源都会被 AssetDB 导入到资源库（library）中，并可以通过 uuid 来引用。

在扩展包的主进程中 url 和 uuid 之间可以互相转化：

- `Editor.assetdb.urlToUuid(url)`
- `Editor.assetdb.uuidToUrl(uuid)`

此外如果希望直接使用资源在本地文件系统中的绝对路径，也可以使用 `fspathToUuid` 和 `uuidToFspath` 接口，其中 `fspath` 就表示绝对路径。

## 管理资源

### 导入资源

要将新资源导入到项目中，可以使用以下接口

```
//main process
Editor.assetdb.import(['/User/user/foo.js', '/User/user/bar.js'], 'db://assets/foobar', function ( err, results ) {
    results.forEach(function ( result ) {
    // result.uuid
    // result.parentUuid
    // result.url
    // result.path
```

```
    // result.type
    });
});


//renderer process
Editor.assetdb.import( [
    '/file/to/import/01.png',
    '/file/to/import/02.png',
    '/file/to/import/03.png',
], 'db://assets/foobar', callback);
```

## 创建资源

使用扩展包管理资源的一个常见误区，就是当扩展包需要创建新资源时直接使用了 Node.js 的 fs 模块，这样即使创建文件到了 `assets` 目录，也无法自动被资源管理器导入。正确的工作流程应该是使用 `create` 接口来创建资源。

```
//main process or renderer process
Editor.assetdb.create( 'db://assets/foo/bar.js', data, function ( err, results ) {
    results.forEach(function ( result ) {
    // result.uuid
    // result.parentUuid
    // result.url
    // result.path
    // result.type
    });
});
```

传入的 `data` 就是该资源文件内容的字符串。在创建完成后会自动进行该资源的导入操作，回调成功后就可以在资源管理器中看到该资源了。

## 保存已有资源

要使用新的数据替换原有资源内容，可以使用以下接口

```
//main process or renderer process
Editor.assetdb.saveExists( 'db://assets/foo/bar.js', data, function ( err, meta ) {
    // do something
});
```

如果要在保存前检查资源是否存在，可以使用

```
//main process
Editor.assetdb.exists(url); //return true or false
```

在渲染进程，如果给定了一个目标 url，如果该 url 指向的资源不存在则创建，资源存在则保存新数据的话，可以使用

```
//renderer process
Editor.assetdb.createOrSave( 'db://assets/foo/bar/foobar.js', data, callback);
```

## 刷新资源

当资源文件在 `assets` 中已经修改，而由于某种原因没有进行重新导入的情况下，会出现 `assets` 里的资源数据和数据库里展示的资源数据不一致的情况（如果使用 `fs` 模块直接操作文件内容就会出现），可以通过手动调用资源刷新接口来重新导入资源

```
//main process or renderer process
```

```
Editor.assetdb.refresh('db://assets/foo/bar/', function (err, results) {});
```

## 移动和删除资源

由于资源导入后会生成对应的 `meta` 文件，所以单独删除和移动资源文件本身都会造成数据库中数据一致性受损，推荐使用专门的 AssetDB 接口来完成这些工作

```
Editor.assetdb.move(srcUrl, destUrl);
Editor.assetdb.delete([url1, url2]);
```

关于这些接口的详情，请查阅 AssetDB API Main 和 AssetDB API Renderer。

# 编写面板界面

Cocos Creator 的面板界面使用 HTML5 标准编写。你可以为界面指定 HTML 模板和 CSS 样式，然后对 界面元素绑定消息编写逻辑和交互代码。如果你之前有过前端页面编程经验，那么这些内容对你来说再熟悉不过。 而没有前端编程经验的开发者也不必太担心，通过本节的学习，将可以让你在短时间内掌握 Creator 面板界面 的编写技巧。

## 定制你的模板

通常在开始编写界面之前，我们总是希望能够在界面中看见点什么。我们可以通过面板定义函数的 `template` 和 `style` 选项来稍微在面板界面上绘制点东西。

一般我们会选择绘制一些区块用于规划界面布局，我们可以写以下代码：

```
Editor.Panel.extend({
  style: `
    .wrapper {
      box-sizing: border-box;
      border: 2px solid white;
      font-size: 20px;
      font-weight: bold;
    }

    .top {
      height: 20%;
      border-color: red;
    }

    .middle {
      height: 60%;
      border-color: green;
    }

    .bottom {
      height: 20%;
      border-color: blue;
    }
  `,

  template: `
    <div class="wrapper top">
      Top
    </div>

    <div class="wrapper middle">
      Middle
    </div>

    <div class="wrapper bottom">
      Bottom
    </div>
  `,
});
```

通过以上代码，我们获得了一个如下的界面效果：


## 界面排版

界面排版是通过在 `style` 中书写 CSS 来完成的。在上面的例子中，我们已经对界面做了简单的排版。如果 对 CSS 不熟悉，推荐大家可以阅读 W3 School 的 CSS 教程 来加强。

在界面排版过程中，有时候我们希望更好的表达元素之间的布局关系，比如我们喜欢 Top 和 Bottom 元素的高度固定为 30px， 而 Middle 元素的高度则撑满剩余空间。这个时候我们就可以使用 CSS Flex 布局来制作。

我们可以这么修改 `style` 部分：

```
Editor.Panel.extend({
  style: `
    :host {
      display: flex;
      flex-direction: column;
    }

    .wrapper {
      box-sizing: border-box;
      border: 2px solid white;
      font-size: 20px;
      font-weight: bold;
    }

    .top {
      height: 30px;
      border-color: red;
```

```
    }

    .middle {
      flex: 1;
      border-color: green;
    }

    .bottom {
      height: 30px;
      border-color: blue;
    }
  `
});
```

由于 CSS Flex 布局语法有些复杂，为了方便大家使用，Cocos Creator 对这部分进行了重新包装，关于这部分的详细介绍，请阅读 界面排版。

# 添加 UI 元素

规划好布局后，我们就可以考虑加入界面元素来完成界面功能。通常，熟悉前端编程的开发人员会想到一些常用的 界面元素，如 `<button>`， `<input>` 等等。这些元素当然是可以直接被使用，但是我们强烈推荐大家使用 Cocos Creator 的内置 UI Kit 元素。这些内置元素都是以 `ui-` 开头，例如 `<ui-button>`， `<ui-input>`。 Cocos Creator 提供了非常丰富的内置元素，开发人员可以通过 掌握 UI Kit 章节获得更详细 的了解。

内置元素不但在样式上经过细致的调整，同时也统一了消息发送规则并且能够更好的处理 focus 等系统事件。

让我们稍微丰富一下我们上面的面板：

```
Editor.Panel.extend({
  style: `
    :host {
      display: flex;
      flex-direction: column;
      margin: 5px;
    }

    .top {
      height: 30px;
    }

    .middle {
      flex: 1;
      overflow: auto;
    }

    .bottom {
      height: 30px;
    }
  `,

  template: `
    <div class="top">
      Mark Down 预览工具
    </div>

    <div class="middle layout vertical">
      <ui-text-area resize-v value="请编写你的 Markdown 文档"></ui-text-area>
      <ui-markdown class="flex-1"></ui-markdown>
    </div>

    <div class="bottom layout horizontal end-justified">
      <ui-button class="green">预览</ui-button>
    </div>
```

```
  `,
});
```

如果一切正常，你将会看到如下界面：


## 为 UI 元素添加逻辑交互

最后让我们通过标准的事件处理代码来完成面板的逻辑部分。假设我们需要在每次点击预览按钮后，都会将 text-area 中输入的 Markdown 文档，渲染并显示在下方。我们可以做如下代码操作：

```
Editor.Panel.extend({
  // ...

  $: {
    txt: 'ui-text-area',
    mkd: 'ui-markdown',
    btn: 'ui-button',
  },

  ready () {
    this.$btn.addEventListener('confirm', () => {
      this.$mkd.value = this.$txt.value;
    });

    // init
    this.$mkd.value = this.$txt.value;
  },
});
```

这里我们通过 `$` 选择器，预先索引了我们需要的 ui 元素。在利用 HTML 标准 API `addEventListener` 为元素添加事件。对于内置 UI Kit 元素，每个 UI 元素都拥有一组标准事件，他们分别是：`cancel`，`change` 和 `confirm`。同时，多数 UI 元素都会携带 `value` 属性，记录元素内相关的值信息。

希望通过本节的代码示例，能够启发你进行面板界面开发的工作。当然，要灵活运用面板界面，还是需要深入学习和掌握 HTML5 标准。

# 掌握 UI Kit

Cocos Creator 为开发者提供了非常丰富的界面元素，帮助开发者快速的开发面板界面。 于此同时， 我们还为开发者提供了控件预览面板，方便开发者在使用控件时，查看控件的各种属性和这些属性对应的效果。 要打开控件预览窗口，仅需要在主菜单中选择 **开发者 / UI Kit Preview**。

目前界面元素包括：

## 控件

- ui-button
- ui-checkbox
- ui-color
- ui-input
- ui-num-input
- ui-select
- ui-slider
- ui-text-area

## 特殊控件

- ui-asset
- ui-node

## 控件容器

- ui-box-container
- ui-prop
- ui-section

## 视图元素

- ui-hint
- ui-loader
- ui-markdown
- ui-progress

# 扩展 UI Kit

如果发现 Cocos Creator 内置的界面元素仍然满足不了你的需求，也不必太担心，你可以通过自定义元素 来对 UI Kit 进行扩展。

UI Kit 的扩展是基于 HTML5 的 Custom Elements 标准。

通过 `Editor.UI.registerElement(tagName, prototype)` 我们可以很轻松的注册自定义元素。这里 有一个简单的范例。

```
Editor.UI.registerElement('foobar-label', {
  template: `
    <div class="text">Foobar</div>
  `,

  style: `
    .text {
      color: black;
      padding: 2px 5px;
      border-radius: 3px;
      background: #09f;
    }
  `
});
```

当你完成上面的代码，并在 Panel 中正确 require 后，你就可以在编辑器中使用 `<foobar-label>` 这个元素。

更多关于自定义元素定义的选项，请阅读自定义界面元素定义参考。

# 内容分布

有时候我们会在自定义元素内加入内容，为了能够让自定义元素正确处理内容，需要在模板中通过 `<content>` 标签加以说明。 这个过程我们称作“内容分布”。

拿上面的例子来说，假设我们希望 `<foobar-label>` 不只是显示 Foobar ，而是根据我们加入的内容进行显示，例如：

```
<foobar-label>Hello World</foobar-label>
```

这个时候就需要使用内容分发功能。我们可以对之前的范例做一个小小的更改：

```
template: `
  <div class="text">
    <content></content>
  </div>
`
```

通过使用 `<content>` 标签告诉样板，我们希望将用户内容放置在这个地方。

# 内容分布选择器

有时候自定义元素的内容不止是文字，而是一些复合元素，我们在做内容分发的时候，希望有些元素在某些标签下，有些元素位于另外的标签中。这个时候就可以考虑使用内容分发选择器。考虑如下样例：

```
<foobar-label>
  <div class="title">Hello World</div>
```

```
    <div class="body">This is Foobar</div>
  </foobar-label>
```

如果我们希望将 `.title` 和 `.body` 元素内容区分对待，我们可以做如下代码：

```
  template: `
    <div class="text title">
      <content select=".title"></content>
    </div>
    <div class="text body">
      <content select=".body"></content>
    </div>
  `
```

通过 `<content>` 标签中加入 `select` 属性，我们可以利用选择器来分布内容。

# 界面排版

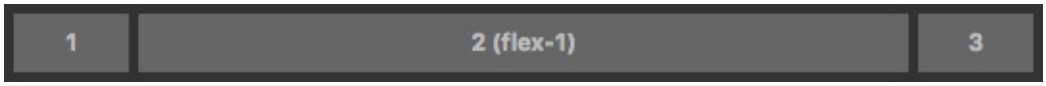
Cocos Creator 的界面排版是通过在 `style` 中书写 CSS 来完成的。如果对 CSS 不熟悉， 推荐大家可以阅读 W3 School 的 CSS 教程 来加强。

然而普通的 CSS 排版并不适合界面元素，为此，CSS 最新标准中加入了 CSS Flex 布局。 通过 Flex 布局，我们可以很轻易的对界面元素进行横排和纵排。为了方便开发者使用 CSS Flex，Cocos Creator 也对他进行了一些封装。本章节主要就是介绍 Cocos Creator 中的界面排版方法。

## 横排和纵排

**layout horizontal**

```
<div class="layout horizontal">
  <div>1</div>
  <div class="flex-1">2 (flex-1)</div>
  <div>3</div>
</div>
```




**layout vertical**

```
<div class="layout vertical">
  <div>1</div>
  <div class="flex-1">2 (flex-1)</div>
  <div>3</div>
</div>
```


## 对其元素

我们在横排纵排时，会希望对所有子元素进行对其操作。可以通过 `start`， `center` 和 `end` 来进行 子元素的对其操作。

对于横排元素，他们分别代表上，中，下对其。对于纵排元素，他们分别代表左，中，右对其。

让我们以横排为例，来看一组例子：

```
<div class="layout horizontal start">
  <div>1</div>
  <div>2</div>
  <div>3</div>
```

```
</div>
<div class="layout horizontal center">
  <div>1</div>
  <div>2</div>
  <div>3</div>
</div>
<div class="layout horizontal end">
  <div>1</div>
  <div>2</div>
  <div>3</div>
</div>
```


有时候，我们需要对排版容器中的某个元素进行对其调整，这个时候就可以通过 `self-` 关键字来操作。在 Cocos Creator 中，我们提供了： `self-start` ， `self-center` ， `self-end` 和 `self-stretch`

让我们以横排为例，来看看这么做的效果：

```
<div class="layout horizontal">
  <div class="self-start">self-start</div>
  <div class="self-center">self-center</div>
  <div class="self-end">self-end</div>
  <div class="self-stretch">self-stretch</div>
</div>
```


# 元素分布

元素分布主要描述元素在排版方向上如何分布。比如所有元素都从排版容器的左边开始排，或者从右边，或者根据 元素大小散步在排版容器中。

我们提供了： `justified` ， `around-justified` ， `start-justified` ， `center-justified` 和 `end-justified` 。

还是以横排为例，来看一组例子：

```
<div class="layout horizontal justified">
  <div>1</div>
  <div>2</div>
  <div>3</div>
</div>
```

```
<div class="layout horizontal around-justified">
  <div>1</div>
  <div>2</div>
  <div>3</div>
</div>
  ...
  ...
```


# 元素自适应

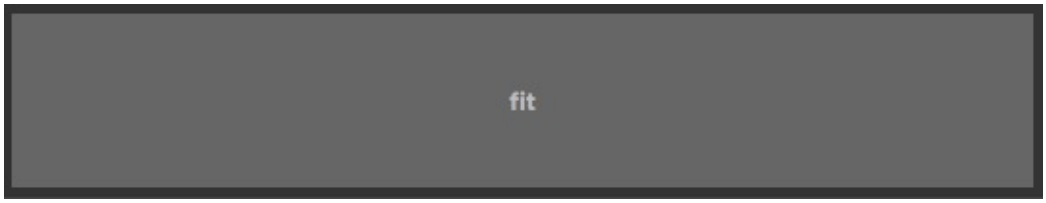
有些时候我们希望元素撑满布局的剩余控件。我们可以通过在布局容器的子元素中使用 `flex-1`， `flex-2`， ..... `flex-12` 来操作。

来看一组例子：

```
<div class="layout horizontal">
  <div class="flex-1">flex-1</div>
  <div class="flex-2">flex-2</div>
  <div class="flex-3">flex-3</div>
</div>
<div class="layout horizontal">
  <div class="flex-none">flex-none</div>
  <div class="flex-1">flex-1</div>
  <div class="flex-none">flex-none</div>
</div>
  ...
  ...
```


还有些时候我们希望元素本身就撑满容器的整个空间。这个时候就可以考虑使用 `fit` 这个 class。方法和效果如下：

```
<div class="wrapper">
  <div class="fit">fit</div>
</div>
```


fit

# 在面板中使用 Vue

如果你已经掌握了 编写面板界面 这章中的界面编写方法，你或许会觉得这样 编写界面有些繁琐。是否能够使用一些前端界面框架来提升界面编写效率呢？答案是肯定的。Cocos Creator 支持任何界面框架如 Vue，React， Polymer 等等。

在测试过程中，我们发现 Vue 非常符合 Cocos Creator 的整体设计思路，所以 我们重点介绍一下如何在 Cocos Creator 中使用 Vue 编写面板界面。

# 部署 Vue

事实上你不用做任何准备工作，Cocos Creator 的面板窗口在打开时就会默认加载 Vue。

# 初始化 Vue 面板

我们可以在 `ready()` 函数中初始化 Vue 面板。初始化方式如下：

```
ready () {
  new window.Vue({
    el: this.shadowRoot,
  });
}
```

通过传入 `panel-frame` 的 shadow root 元素，我们可以让 Vue 在该元素节点下生成一份 vm。让我们来 看一个更详细的使用例子：

```
Editor.Panel.extend({
  style: `
    :host {
      margin: 10px;
    }
  `,

  template: `
    <h2>A Simple Vue Panel</h2>

    <input v-model="message">
    <p>Input Value = <span>{{message}}</span></p>
  `,

  ready () {
    new window.Vue({
      el: this.shadowRoot,
      data: {
        message: 'Hello World',
      },
    });
  },
});
```

# 数据绑定

我们可以在面板的 `template` 关键字中，定义 Vue 的数据绑定规则。然后通过在 Vue 定义的 `data` 关键字 中写入绑定数据来完成整个操作。

具体例子如下：

```
Editor.Panel.extend({
  template: `
    <ui-button>{{txtOK}}</ui-button>
    <ui-button v-if="showCancel">{{txtCancel}}</ui-button>
    <ui-input v-for="item in items" value="{{item.message}}"></ui-input>
  `,

  ready () {
    new window.Vue({
      el: this.shadowRoot,
      data: {
        txtOK: 'OK',
        txtCancel: 'Cancel',
        showCancel: false,
        items: [
          { message: 'Foo' },
          { message: 'Bar' },
        ]
      },
    });
  },
});
```

# 事件绑定

除了使用数据绑定，我们还可以通过 Vue 的 `@` 方式来将事件和方法绑定在一起。值得注意的是，绑定的 方法必须定义在 Vue 定义中的 `methods` 关键字中。

具体例子如下：

```
Editor.Panel.extend({
  template: `
    <ui-button @confirm="onConfirm">Click Me</ui-button>
  `,

  ready () {
    new window.Vue({
      el: this.shadowRoot,
      methods: {
        onConfirm ( event ) {
          event.stopPropagation();
          console.log('On Confirm!');
        },
      },
    });
  },
});
```

# 扩展 Inspector

Inspector 是在 **属性检查器** 里展示的组件控件面板，有时候我们需要对自己书写的 Component 定义一份 Inspector，用自定义的方式对他进行显示。这个时候 我们可以考虑对 Inspector 进行扩展。

扩展的步骤如下：

1. 在 Component 中注明自定义 Inspector 的入口文件
2. 创建自定义 Inspector 的扩展包
3. 在扩展包中编写自定义 Inspector 的入口文件

## 在 Component 中注明自定义 Inspector 的入口文件

首先我们需要定义一份 Component 脚本，并且为这个脚本注明使用自定义 Inspector，方法如下：

```
cc.Class({
  name: 'Foobar',
  extends: cc.Component,
  editor: {
    inspector: 'packages://foobar/inspector.js',
  },
  properties: {
    foo: 'Foo',
    bar: 'Bar'
  },
});
```

**注意1:** 这里我们定义了一个 `editor` 字段，并在该字段中定义了 `inspector` 入口文件。

**注意2:** 在 `inspector` 中我们采用 `packages://` 协议定义入口文件路径。在 Cocos Creator 中会对 `packages://` 协议做特殊处理，会将 `packages://` 协议后面的分路径名当做扩展包名字进行搜索， 并根据搜索结果将整个协议替换成扩展包的路径做后续搜索。

## 创建自定义 Inspector 的扩展包

和我们创建一份扩展包没有任何区别，你可以按照 你的第一个扩展包 中的 方式创建一份。这里我们假设我们的扩展包名为 foobar。

注意，在创建完扩展包后，你需要重启一下 Cocos Creator 以便让他正确读入该扩展包。

## 在扩展包中编写自定义 Inspector 的入口文件

接下来我们就可以在 `foobar` 包中定义 `inspector.js`：

```
Vue.component('foobar-inspector', {
  template: `
    <ui-prop v-prop="target.foo"></ui-prop>
    <ui-prop v-prop="target.bar"></ui-prop>
  `,

  props: {
    target: {
      twoWay: true,
      type: Object,
```

```
    },
  },
});
```

Cocos Creator 的 Inspector 扩展使用了 Vue。这里我们通过定义一份 Vue 的组件，并在组件中定义 `props`，使得其包含 `target` 数据来完成整个 Inspector 的数据定义。

该 `target` 就是我们的 `Foobar` Class 在 Inspector 中对应的实例。

# 关于 target

上一小节中提到的 `target` 实例是经过 Inspector 处理过的 target。其内部包含了对属性的加工处理。 在使用的时候，我们不能简单的认为 `target.foo` 就代表 foo 的值。如果你去查看 `target.foo` 你会 发现他是一个 Object 而不是我们在最开始定义的那样一个 'Foo' 的字符串。该份 Object 中包含了 `attrs`， `type`， `value` 等信息。其中的 `value` 才是我们真正的值。

这么做的目的是为了让 Inspector 可以更好的获得数据可视化的各方面信息。例如当你定义了 cc.Class 的属性为：

```
properties: {
  foo: {
    value: 'Foo',
    readonly: true
  }
}
```

这个时候在 Inspector 中的 target 里反应出的信息为 `target.foo.value` 为 'Foo'， `target.foo.attrs.readonly` 为 `true`。这些信息有助于帮助你创建多变的界面组合。

# 关于属性绑定

由于这些信息非常繁琐，Cocos Creator 也对 Vue 的 directive 做了一定的扩展。目前我们扩展了 `v-prop`， `v-value` `v-readonly` 和 `v-disable`。

当你还是想利用 Cocos Creator 的默认方案显示数据段时，你可以使用 `v-prop` 配合 `<ui-prop>` 控件做绑定，如：

```
<ui-prop v-prop="target.foo"></ui-prop>
```

当你想使用 `<ui-xxx>` 的原生控件时，你可以使用 `v-value` 来做数据绑定，如：

```
<ui-input v-value="target.foo.value"></ui-input>
```

当你想对控件应用 readonly 或者 disable 行为时，请使用 `v-readonly` 和 `v-disable`。如：

```
<ui-button v-readonly="target.foo.attrs.readonly">Foo</ui-button>
<ui-button v-disable="target.bar.value">Bar</ui-button>
```

# 自定义 Gizmo

目前 Gizmo 使用 svg.js 作为操作工具, 具体 svg.js 的 api 可以参考 http://documentup.com/wout/svg.js

# 创建自定义 Gizmo

这里演示创建一个简单的跟着节点移动并缩放的圆

```
// 定义一个简单的 component，并命名为 CustomComponent
cc.Class({
    extends: cc.Component,

    properties: {
        radius: 100
    },
});
```

```
class CustomGizmo extends Editor.Gizmo {
    init () {
        // 初始化一些参数
    }

    onCreateRoot () {
        // 创建 svg 根节点的回调，可以在这里创建你的 svg 工具
        // this._root 可以获取到 Editor.Gizmo 创建的 svg 根节点

        // 实例：

        // 创建一个 svg 工具
        // group 函数文档 : http://documentup.com/wout/svg.js#groups
        this._tool = this._root.group();

        // 画一个的圆
        // circle 函数文档 : http://documentup.com/wout/svg.js#circle
        let circle = this._tool.circle();

        // 为 tool 定义一个绘画函数，可以为其他名字
        this._tool.plot = (radius, position) => {
            this._tool.move(position.x, position.y);
            circle.radius(radius);
        };
    }

    onUpdate () {
        // 在这个函数内更新 svg 工具

        // 获取 gizmo 依附的组件
        let target = this.target;

        // 获取 gizmo 依附的节点
        let node = this.node;

        // 获取组件半径
        let radius = target.radius;

        // 获取节点世界坐标
        let worldPosition = node.convertToWorldSpaceAR(cc.p(0, 0));

        // 转换世界坐标到 svg view 上
        // svg view 的坐标体系和节点坐标体系不太一样，这里使用内置函数来转换坐标
        let viewPosition = this.worldToPixel(worldPosition);
```

```
        // 对齐坐标，防止 svg 因为精度问题产生抖动
        let p = Editor.GizmosUtils.snapPixelWihVec2( viewPosition );

        // 获取世界坐标下圆半径
        let worldPosition2 = node.convertToWorldSpaceAR(cc.p(radius, 0));
        let worldRadius = worldPosition.sub(worldPosition2).mag();
        worldRadius = Editor.GizmosUtils.snapPixel(worldRadius);

        // 移动 svg 工具到坐标
        this._tool.plot(worldRadius, p);
    }

// 如果需要自定义 Gizmo 显示的时机，重写 visible 函数即可
//    visible () {
//        return this.selecting || this.editing;
//    }

// Gizmo 创建在哪个 Layer 中 : foreground, scene, background
// 默认创建在 scene Layer
//    layer () {
//        return 'scene';
//    }

// 如果 Gizmo 需要参加 点击测试，重写 rectHitTest 函数即可
//    rectHitTest (rect, testRectContains) {
//        return false;
//    }
}

module.exports = CustomGizmo;
```

# 注册自定义 Gizmo

在你的自定义 **package** 里的 **package.json** 中定义 **gizmos** 字段，并注册上你的自定义 Gizmo

```
"gizmos": {
    "CustomComponent": "packages://custom-gizmo/custom-gizmo.js"
}
```

**CustomComponent**： Component 名字
**packages://custom-gizmo/custom-gizmo.js**： CustomGizmo 路径

这样就将 CustomGizmo 注册到 CustomComponent 上了，当添加一个 CustomComponent 到节点上并选择这个节点时，就可以看到这个 gizmo 了。

请阅读下一篇 自定义 Gizmo 进阶

更多 Gizmo Api 请参考 Gizmo Api 更多 Gizmo 实例请参考 Gizmo 实例

# 自定义 Gizmo 进阶

上一篇 讲了如何自定义一个跟随节点移动并缩放的 Gizmo，这篇我们将实现一个可以编辑的 Gizmo

```
let ToolType = {
    None: 0,
    Side: 1,
    Center: 2
};


class CustomGizmo extends Editor.Gizmo {
    init () {
        // 初始化一些参数
    }

    onCreateMoveCallbacks () {
        // 创建 gizmo 操作回调

        // 申明一些局部变量
        let startOffset;        // 按下鼠标时记录的园偏移量
        let startRadius;        // 按下鼠标时记录的圆半径
        let pressx, pressy;     // 按下鼠标时记录的鼠标位置

        return {
            /**
             * 在 gizmo 上按下鼠标时触发
             * @param x 按下点的 x 坐标
             * @param y 按下点的 y 坐标
             * @param event mousedown dom event
             */
            start: (x, y, event) => {
                startRadius = this.target.radius;
                startOffset = this.target.offset;
                pressx = x;
                pressy = y;
            },

            /**
             * 在 gizmo 上按下鼠标移动时触发
             * @param dx 鼠标移动的 x 位移
             * @param dy 鼠标移动的 y 位移
             * @param event mousedown dom event
             */
            update: (dx, dy, event, type) => {
                // 获取 gizmo 依附的节点
                let node = this.node;

                // 获取 gizmo 依附的组件
                let target = this.target;

                if (type === ToolType.Center) {
                    // 计算新的偏移量
                    let t = cc.affineTransformClone( node.getWorldToNodeTransform() );
                    t.tx = t.ty = 0;

                    let d = cc.v2(cc.pointApplyAffineTransform(dx, dy, t)).add(startOffset);
                    target.offset = d;
                    this.adjustValue(target, 'offset');
                }
                else {
                    // 转换坐标点到节点下
                    let position = node.convertToNodeSpaceAR(cc.v2(pressx + dx, pressy + dy));
                    // 计算 radius
                    target.radius = position.sub(startOffset).mag();
```

```
                    // 防止 radius 小数点位数过多
                    this.adjustValue(target, 'radius');
                }
            },

            /**
             * 在 gizmo 抬起鼠标时触发
             * @param event mousedown dom event
             */
            end: (updated, event) => {
            }
        };
    }

    onCreateRoot () {
        // 创建 svg 根节点的回调，可以在这里创建你的 svg 工具
        // this._root 可以获取到 Editor.Gizmo 创建的 svg 根节点

        // 实例：

        // 创建一个 svg 工具
        // group 函数文档 : http://documentup.com/wout/svg.js#groups
        this._tool = this._root.group();

        // 创建中心拖拽区域，用于操作 offset 属性
        let dragArea = this._tool.circle()
            // 设置 fill 样式
            .fill( { color: 'rgba(0,128,255,0.2)' } )
            // 设置点击区域，这里设置的是根据 fill 模式点击
            .style( 'pointer-events', 'fill' )
            // 设置鼠标样式
            .style( 'cursor', 'move' )
            ;

        // 注册监听鼠标移动事件的 svg 元素
        // ToolType.Center 是自定义的参数，会在移动回调中按照参数的形式传递到移动回调中，方便区别当前回调是哪一个 svg 元素产生的回
调。
        // {cursor: 'move'} 指定移动时的鼠标类型
        this.registerMoveSvg( dragArea, ToolType.Center, {cursor: 'move'} );

        // 创建边缘拖拽区域，用于操作 radius 属性
        let circle = this._tool.circle()
            // 设置stroke 样式
            .stroke( { color: '#7fc97a', width: 2 } )
            // 设置点击区域，这里设置的是根据 stroke 模式点击
            .style( 'pointer-events', 'stroke' )
            // 设置鼠标样式
            .style( 'cursor', 'pointer' )

        this.registerMoveSvg( circle, ToolType.Side, {cursor: 'pointer'} );

        // 为 tool 定义一个绘画函数，方便在 onUpdate 中更新 svg 的绘制。
        this._tool.plot = (radius, position) => {
            this._tool.move(position.x, position.y);
            dragArea.radius(radius);
            circle.radius(radius);
        };
    }

    onUpdate () {
        // 更新 svg 工具

        // 获取 gizmo 依附的组件
        let target = this.target;

        // 获取 gizmo 依附的节点
        let node = this.node;

        // 获取节点世界坐标
```

```
        let position = node.convertToWorldSpaceAR(target.offset);

        // 转换世界坐标到 svg view 上
        // svg view 的坐标体系和节点坐标体系不太一样，这里使用内置函数来转换坐标
        position = this.worldToPixel(position);

        // 对齐坐标，防止 svg 因为精度问题产生抖动
        position = Editor.GizmosUtils.snapPixelWihVec2( position );

        // 获取世界坐标下圆半径
        let p1 = node.convertToWorldSpaceAR(cc.p(target.radius, 0));
        let p2 = node.convertToWorldSpaceAR(cc.p(0, 0));
        let radius = p1.sub(p2).mag();

        // 对齐坐标，防止 svg 因为精度问题产生抖动
        radius = Editor.GizmosUtils.snapPixel(radius);

        // 移动 svg 工具到坐标
        this._tool.plot(radius, position);
    }
}

module.exports = CustomGizmo;
```

更多 Gizmo Api 请参考 Gizmo Api 更多 Gizmo 实例请参考 Gizmo 实例

# 测试你的扩展包

努力更新中...

# package.json 字段参考

## name (String)

你的扩展包名字。扩展包名字是全局唯一的，他关系到你今后在官网服务器上登录时的名字。

## version (String)

版本号，我们推荐使用 semver 格式管理你的包版本。

## description (String)

一句话描述你的扩展包是用来做什么的。

## author (String)

注明扩展包的作者，可以是你的名字，团队的名字或者公司的名字。

## main (String)

入口函数文件。通常我们会在包中存放一个 main.js 文件作为入口函数文件。你也可以在入口函数文件放在扩展包中的其他位置例如：`main/index.js`，只要在 `main` 字段中正确书写你的文件的相对路径即可。

## main-menu (Object)

主菜单注册，主菜单注册的键值（Key）是一段菜单路径，注册信息为一个对象，关于注册信息可详细阅读主菜单字段参考。

其中菜单路径为一份 posix 格式的路径，菜单会根据路径中的名字，依次注册到主菜单中。例如：“My Package/Preview/Foo Bar” 将会在主菜单中寻找 My Package > Preview 这个菜单路径，如果寻找过程中没有发现对应的菜单项，将会自动添加。最终 Cocos Creator 会将 “Foo Bar” 添加到对应的路径中，如图：


## panel (Object)

面板注册，面板注册的键值（Key）是一个以 `panel` 开头的字符串，字符串后面可跟上后缀名用于多面板的注册。 注册完的面板，将会生成以 `${包名}${面板后缀名}` 为组合的面板 ID，如果没有后缀名（通常我们如果注册一个面板就不会带后缀）， 则面板 ID 直接等于插件包的名字。

关于多面板注册，这里我们提供了一个简单的例子，在 `package.json` 中：

```
{
  "name": "simple-package",
  "panel": {
    "main": "panel/index.js",
    "type": "dockable",
    "title": "Simple Panel",
    "width": 400,
```

```
    "height": 300
  },
  "panel-02": {
    "main": "panel-02/index.js",
    "type": "dockable",
    "title": "Simple Panel 02",
    "width": 400,
    "height": 300
  },
}
```

这样注册完的面板，将会生成两份面板 ID 分别为： `simple-package` 和 `simple-package-02` .

关于面板注册信息可详细阅读面板字段参考。

## reload (Object)

可以通过 `reload` 字段定制扩展包自动重载的文件监控规则，未做声明时的默认规则如下：

```
"reload": {
  "test": [ "test/**/*", "tests/**/*" ],
  "renderer": [ "renderer/**/*", "panel/**/*" ],
  "ignore": [],
  "main": []
}
```

## runtime-resource (Object)

插件通过在 `package.json` 文件中配置 `runtime-resource` 字段来 mount runtime 资源到资源管理器中。配置的格式如下：

```
"runtime-resource": {
  "path": "path/to/runtime-resource",
  "name": "runtime-res-name"
}
```

最终在资源管理器中由插件 mount 的文件夹名称为 `[packageName]-[runtime-resource.name]` 。且插件导入的资源文件夹为只读的。

需要注意的是，通过配置 `runtime-resource` 字段将扩展包中的文件夹 mount 到项目资源后，本身就具备自动同步的功能，也就是对扩展包中的 `runtime-resource` 进行的修改会自动同步到项目资源中并触发编译等流程，所以应该将 `runtime-resource` 里 `path` 字段指向的路径添加到 `package.json` 中的 `reload.ignore` 中，否则会引起插件的重复加载：

```
"runtime-resource": {
  "path": "my-components",
  "name": "components"
},
"reload": {
  "ignore": ["my-components/**/*"]
}
```

## scene-script (String)

`scene-script` 字段用于声明一个扩展包内的脚本，在该脚本中可以使用引擎 API，并访问当前场景中的节点和组件。

声明形式如下：

```
"scene-script": "scene-walker.js"
```

该字段的值是一个脚本文件的路径，相对于扩展包目录。详细的用法和工作流程请阅读 调用引擎 API 和项目脚本。

# 主菜单字段参考

## message (String)

IPC 消息字段。当点击菜单后，将会发送该字段中的消息到主进程中。

## command (String)

command 和 message 功能相似，但是他不是向主进程中发送 IPC 消息，而是直接在主进程中寻找并运行你给出的全局函数。例如：

```
{
  "Examples/Say Hello": {
    "command": "Editor.log",
    "params": ["Hello World!"]
  }
}
```

值得注意的是，message 和 command 字段不能共存，因为他们都关系到点击行为。

## params (Array)

消息参数，你可以通过定义数组，并填写数组元素使得发送的消息带有参数。

## accelerator (String)

快捷键注册字段。你可以在这里定义你的菜单快捷键，具体的定义规则，请参考 Accelerator。

## icon (String)

图标文件的相对路径，通过指定 icon 文件，你可以为你的菜单选项加上一份图标。

## visible (Boolean)

控制菜单是否隐藏。

## enabled (Boolean)

控制菜单是否可点击。

# 面板字段参考

## main (String)

面板入口文件，可以是 js 文件也可以是 html 文件。取决于面板类型（type）。如果面板类型为 `simple` 则读入的为 html 文件。否则读入 js 文件。

## type (String)

面板类型。可选类型有：

- dockable：该面板为标准的编辑器面板，可以自由的在编辑器内停靠。
- float：该面板为浮动面板，不能停靠在编辑器中。
- fixed-size：该面板和浮动面板功能相似，不同之处在于他的窗口大小是固定的。
- quick：该面板和浮动面功能相似，不同之处在于当他失去焦点后将会自动关闭。
- simple：简单面板拥有独立窗口，通过读取用户自定义的 html 文件展示界面。

## title (String)

面板标题，如果面板类型为 `dockable`，面板标题将会显示在 Tab 上。

## icon (String)

面板 Tab 上显示的图标

## resizable (Boolean)

设置面板在独立窗口状态下时是否可以改变大小

## width (Integer)

设置面板窗口的初始宽度

## height (Integer)

设置面板窗口的初始高度

## min-width (Integer)

设置面板窗口的最小宽度

## min-height (Integer)

设置面板窗口的最小高度

## max-width (Integer)

设置面板窗口的最大宽度

## max-height (Integer)

设置面板窗口的最大高度

# 面板定义参考

> 一份简单的面板定义

```
Editor.Panel.extend({
  style: `
    :host { margin: 5px; }
    h2 { color: #f90; }
  `,

  template: `
    <h2>Hello World!</h2>
  `,

  ready () {
    Editor.log('Hello World!');
  },
});
```

# 选项

## 属性

### style

使用 CSS 定义你的样式。在 Panel 定义中，CSS 样式被定义在 Shadow DOM 中， 所以他遵循 Shadow DOM 的样式定义规范。

使用 `:host` 来表示 Panel 本身的样式。而在 Panel 中的其他样式则遵循 CSS 选择器的书写规则。 由于采用 Shadow DOM，我们定义在面板内的样式和其他面板样式以及全局样式是隔离开的，所以不必担心 CSS 命名污染的问题。

更多的 Shadow DOM 的介绍可以参考：

- http://www.html5rocks.com/zh/tutorials/webcomponents/shadowdom/
- http://www.html5rocks.com/zh/tutorials/webcomponents/shadowdom-201/
- http://www.html5rocks.com/zh/tutorials/webcomponents/shadowdom-301/

### template

使用 HTML 标记语言定义你的面板的 UI 元素。

### listeners

通过定义一个 Object 将 DOM 事件绑定到自定义函数上。这个 Object 的 Key 就是 DOM 事件名，Value 则是函数本身。样例如下：

```
Editor.Panel.extend({
  // ...

  listeners: {
    mousedown ( event ) {
      event.stopPropagation();
      Editor.log('on mousedown');
    },
```

```
    'panel-resize' ( event ) {
      event.stopPropagation();
      Editor.log('on panel resize');
    }
  }
});
```

## messages

通过定义一个 Object 将 IPC 消息绑定到自定义函数上。这个 Object 的 Key 就是 IPC 消息名，Value 则是函数本身。样例如下：

```
Editor.Panel.extend({
  // ...

  messages: {
    'foobar:say-hello' ( event ) {
      Editor.log(`Hello ${foobar}`);
    },
  }
});
```

## behaviors

`behaviors` 为一个数组，behaviors 会将数组中的元素通过 `mixin` 的方式融合到 Panel 本身。这 为 Panel 中实现行为共享提供了比较方便的途径。

使用方法如下：

```
  // foobar.js
  module.exports = {
    sayHello () {
      Editor.log('Hello Foobar');
    }
  }
```

```
const Foobar = Editor.require('packages://foobar/foobar.js')

Editor.Panel.extend({
  behaviors: [ Foobar ],

  ready () {
    this.sayHello();
  },
});
```

## dependencies

`dependencies` 为一个数组，其内部元素为 url 或 文件路径。 有时候我们会需要用到一些第三方库，通过 指定 dependencies 会在 Panel 初始化前，就先将第三方库读入。

使用方法如下：

```
Editor.Panel.extend({
  dependencies: [
    'packages://foobar/index.js',
  ],
});
```

## $

`$` 为一个 Object，他可以通过 CSS 选择器的语法，将模板中的元素映射成 `$` 变量方便用户使用。假设 我们有以下代码：

```
Editor.Panel.extend({
  template: `
    <div class="foo"></div>
    <div class="bar"></div>
  `,

  $: {
    foo: '.foo',
    bar: '.bar',
  }

  ready () {
    this.$foo.innerText = 'Foo';
    this.$bar.innerText = 'Bar';
  },
});
```

我们可以看到，通过选择器，我们得到 `$foo` 和 `$bar` 两个元素。方便了我们在初始化过程中对其进行进一步的操作。

## 函数

## ready ()

当 Panel 被正确读入，模板实例化并且样式也被正确加入后，将会调动 ready 函数。

## run ( argv )

当 Panel 第一次载入 ready 结束后，或者当 Panel 被 `Editor.Panel.open` 调用激活。 其中，通过 `Editor.Panel.open` 的方式调用是可以传递参数 argv 给 `run` 函数。

## close ()

当 Panel 被关闭退出前被调用。通过返回值决定 Panel 是否真正被关闭。如果返回为 true 则关闭 Panel。

## DOM 事件

## panel-show

当 Panel 显示的时候发送

## panel-hide

当 Panel 隐藏的时候发送

## panel-resize

当 Panel 大小改变的时候发送

## panel-cut

当操作系统剪切行为触发时发送

## panel-copy

当操作系统复制行为触发时发送

## panel-paste

当操作系统粘贴行为触发时发送

# 自定义界面元素定义参考

> 一份简单的元素定义

```
Editor.UI.registerElement('foobar-label', {
  template: `
    <div class="text">Foobar</div>
  `,

  style: `
    .text {
      color: black;
      padding: 2px 5px;
      border-radius: 3px;
      background: #09f;
    }
  `
});
```

自定义元素是基于 HTML5 的 Element 和 Custom Elements 标准。

## 选项

### 属性

### style

使用 CSS 定义你的样式。同 面板定义参考。

### template

使用 HTML 标记语言定义你的面板的 UI 元素。

### listeners

通过定义一个 Object 将 DOM 事件绑定到自定义函数上。这个 Object 的 Key 就是 DOM 事件名，Value 则是函数本身。同 面板定义参考。

### behaviors

`behaviors` 为一个数组，behaviors 会将数组中的元素通过 `mixin` 的方式融合到自定义元素本身。 目前内置的 behaviors 有：

- Editor.UI.Focusable
- Editor.UI.Disabled
- Editor.UI.Readonly
- Editor.UI.Droppable
- Editor.UI.ButtonState
- Editor.UI.InputState

### $

`$` 为一个 Object，他可以通过 CSS 选择器的语法，将模板中的元素映射成 `$` 变量方便用户使用。 同 面板定义参考。

## 函数

### ready ()

当自定义元素被正确创建后，将会调动 ready 函数。

### factoryImpl (arg1, arg2, ...)

他相当于构建函数，可以让你的自定义元素在构建的时候传递参数。例如：

```
let FoobarLabel = Editor.UI.registerElement('foobar-label', {
  template: `
    <div class="text"></div>
  `,

  $: {
    text: '.text'
  },

  factoryImpl ( text ) {
    this.$text.innerText = text;
  },
});

// 当定义完 factoryImpl，你可以就可以通过如下方法实例化创建
let el = new FoobarLabel('Hello World');
document.body.appendChild(el);
```

### attributeChangedCallback ( name, oldVal, newVal )

当 element 的 html attribute 发生更改时（如调用了 `setAttribute()` 函数）触发。

# 常用 IPC 消息

## 内置插件广播出来的消息

Creator 内置的一些组件，或者插件，在某些操作下，会向所有插件广播一些消息，通知所有插件出现的变动以及更改。

### 文件系統

1. asset-db:assets-created

   新建文件的时候，assetDB 会发送这个事件。

2. asset-db:assets-moved

   项目文件夹内如果有文件被移动，则会发送这个事件。

3. asset-db:assets-deleted

   当一个文件被删除的时候，会发送这个事件。

4. asset-db:asset-changed

   如果文件被修改，则会发送这个事件。

5. asset-db:script-import-failed

   当一个脚本在导入时出现错误，会发送这个事件通知。

### 场景

1. scene:enter-prefab-edit-mode

   场景进入 prefab 编辑状态的时候会发送这个消息

2. scene:saved

   当场景保存后，会发送这个消息

3. scene:reloading

   当场景因为特殊原因刷新的时候，会发送这个消息

4. scene:ready

   场景准备完毕发送的消息

### 编译

1. editor:build-start

   编译开始的消息

2. editor:build-finished

   编译完成的消息

3. builder:state-changed

编译状态更新时，发送的消息

4. builder:query-build-options

   查看构建的选项

# 内置插件 Panel 内监听的消息

## scene:new-scene

在编辑器内打开一个新的场景。

```
Editor.Ipc.sendToPanel('scene', 'scene:new-scene');
```

## scene:play-on-device

使用界面上当前选中的预览设备来进行预览。

```
Editor.Ipc.sendToPanel('scene', 'scene:play-on-device');
```

## scene:query-hierarchy

查询编辑器内当前打开场景里的 hierarchy 数据。

```
Editor.Ipc.sendToPanel('scene', 'scene:query-hierarchy', (error, sceneID, hierarchy) => {
    if (error)
        return Editor.error(error);
    // hierarchy
});
```

## scene:query-nodes-by-comp-name

传入一个 Component 名字，返回场景内含有这个组件的节点数组。

```
Editor.Ipc.sendToPanel('scene', 'scene:query-nodes-by-comp-name', 'cc.Sprite', (error, nodes) => {
    if (error)
        return Editor.error(error);
    // nodes
});
```

## scene:query-node

发送一个节点 id，查询这个节点的 dump 数据。dump 数据是一个字符串，需要使用 JSON 手动转成 Object 使用。

```
Editor.Ipc.sendToPanel('scene', 'scene:query-node', '9608cbWFmVIM7m6hasLXYV7', (error, dump) => {
    if (error)
        return Editor.error(error);
    // JSON.parse(dump);
});
```

## scene:query-node-info

传入一个节点或者组件的 id 与一个类型，返回查询的节点的基本信息。

```
Editor.Ipc.sendToPanel('scene', 'scene:query-node-info', '9608cbWFmVIM7m6hasLXYV7', 'cc.Node', (error, info) => {
    if (error)
        return Editor.error(error);
    // info
});
```

## scene:query-node-functions

传入一个节点 id，返回这个节点上所有组件内的函数

```
Editor.Ipc.sendToPanel('scene', 'scene:query-node-functions', '9608cbWFmVIM7m6hasLXYV7', (error, functions) => {
    if (error) {
        return Editor.error(error);
    }
    // functions
});
```

## scene:query-animation-node

传入一个节点 id，根据这个节点查找最近的动画根节点。并返回这个节点的 dump 数据。

```
Editor.Ipc.sendToPanel('scene', 'scene:query-animation-node', '9608cbWFmVIM7m6hasLXYV7', (error, dump) => {
    if (error) {
        return Editor.error(error);
    }
    // dump
});
```

# 进阶主题

- 热更新管理器
  - 热更新范例教程
  - 热更新管理器文档
- C++/Lua 引擎支持
- i18n 游戏多语言支持
- 存储和读取用户数据
- 引擎定制工作流程
- 脏矩形优化
- Java 原生反射机制
- Objective-c 原生反射机制
- BMFont 与 UI自动批处理

---

继续前往 动态热更新 开始了解扩展编辑器的工作流程。

# C++ and Lua Support

For now C++ and Lua support in Cocos Creator is provided as an extension package.

Please checkout Creator support for cocos2d-x for download and guide.

# 资源热更新教程

## 前言

之所以这篇文档的标题为教程，是因为目前 Cocos Creator 资源热更新的工作流还没有彻底集成到编辑器中，不过引擎本身对于热更新的支持是完备的，所以借助一些外围脚本和一些额外的工作就可以达成。

本篇文档的范例工程可以从 Github 仓库获取。


## 使用场景和设计思路

资源热更新的使用场景相信游戏开发者都非常熟悉，对于已发布的游戏，在游戏内通过从服务器动态下载新的游戏内容，来时刻保持玩家对游戏的新鲜感，是保持一款游戏长盛不衰非常重要的手段。当然热更新还有一些其他的用途，不过在此不再深入讨论，我们下面将主要讨论 Cocos Creator 对热更新支持的原理和手段。

Cocos Creator 中的热更新主要源于 Cocos 引擎中的 AssetsManager 模块对热更新的支持。它有个非常重要的特点：

**服务端和本地均保存完整版本的游戏资源**，热更新过程中通过比较服务端和本地版本的差异来决定更新哪些内容。这样即可天然支持跨版本更新，比如本地版本为 A，远程版本是 C，则直接更新 A 和 C 之间的差异，并不需要生成 A 到 B 和 B 到 C 的更新包，依次更新。所以，在这种设计思路下，新版本的文件以离散的方式保存在服务端，更新时以文件为单位下载。

除此之外，由于 WEB 版本可以通过服务器直接进行版本更新，所以资源热更新只适用于原生发布版本。
AssetsManager 类也只在 jsb 命名空间下，在使用的时候需要注意判断运行环境。

## Manifest 文件

对于不同版本的文件级差异，AssetsManager 中使用 Manifest 文件来进行版本比对。本地和远端的 Manifest 文件分别标示了本地和远端的当前版本包含的文件列表和文件版本，这样就可以通过比对每个文件的版本来确定需要更新的文件列表。

Manifest 文件中包含以下几个重要信息：

1. 远程资源包的根路径
2. 远程 Manifest 文件地址
3. 远程 Version 文件地址（非必需）
4. 主版本号
5. 文件列表：以文件路径来索引，包含文件版本信息，一般推荐用文件的 md5 校验码来作为版本号
6. 搜索路径列表

其中 Version 文件内容是 Manifest 文件内容的一部分，不包含文件列表。由于 Manifest 文件可能比较大，每次检查更新的时候都完整下载的话可能影响体验，所以开发者可以额外提供一个非常小的 Version 文件。AssetsManager 会首先检查 Version 文件提供的主版本号来判断是否需要继续下载 Manifest 文件并更新。

# 在 Cocos Creator 项目中支持热更新

在这篇教程中，将提出一种针对 Cocos Creator 项目可行的热更新方案，我们也在 cocos2d-x 的中开放了 Downloader 的 JavaScript 接口，用户可以自由开发自己的热更新方案。

在开始详细讲解之前，开发者可以看一下 Cocos Creator 发布原生版本后的目录结构，这个目录结构和 Cocos2d-x JS 项目的目录是完全一致的。以前没有接触过 Cocos2d-x 的用户可以参考项目结构文档。对于 Cocos Creator 来说，所有 JS 脚本将会打包到 src 目录中，其他 Assets 资源将会被导出到 res 目录。

基于这样的项目结构，本篇教程中的热更新思路很简单：

1. 基于原生打包目录中的 res 和 src 目录生成本地 Manifest 文件。
2. 创建一个热更新组件来负责热更新逻辑。
3. 游戏发布后，若需要更新版本，则生成一套远程版本资源，包含 res 目录、src 目录和 Manifest 文件，将远程版本部署到服务端。
4. 当热更新组件检测到服务端 Manifest 版本不一致时，就会开始热更新

教程所使用的范例工程是基于 21 点范例修改而来的，为了展示热更新的过程，将工程中的 table 场景（牌桌场景）删除，设为 1.0.0 版本。并在 `remote-assets` 目录中保存带有 table 场景的完整版本，设为 1.1.0 版本。游戏开始时会检查远程是否有版本更新，如果发现远程版本则提示用户更新，更新完成后，用户重新进入游戏即可进入牌桌场景。


**注意**，项目中包含的 `remove-assets` 为 debug 模式，开发者在测试的时候必须使用 debug 模式构建项目才有效，否则 release 模式的 jsc 文件优先级会高于 `remove-assets` 中的资源而导致脚本失效。

## 使用 Version Generator 来生成 Manifest 文件

在范例工程中，我们提供了一个 version_generator.js 文件，这是一个用于生成 Manfiest 文件的 NodeJS 脚本。使用方式如下：

```
> node version_generator.js -v 1.0.0 -u http://your-server-address/tutorial-hot-update/remote-assets/ -s native/package/ -d assets/
```

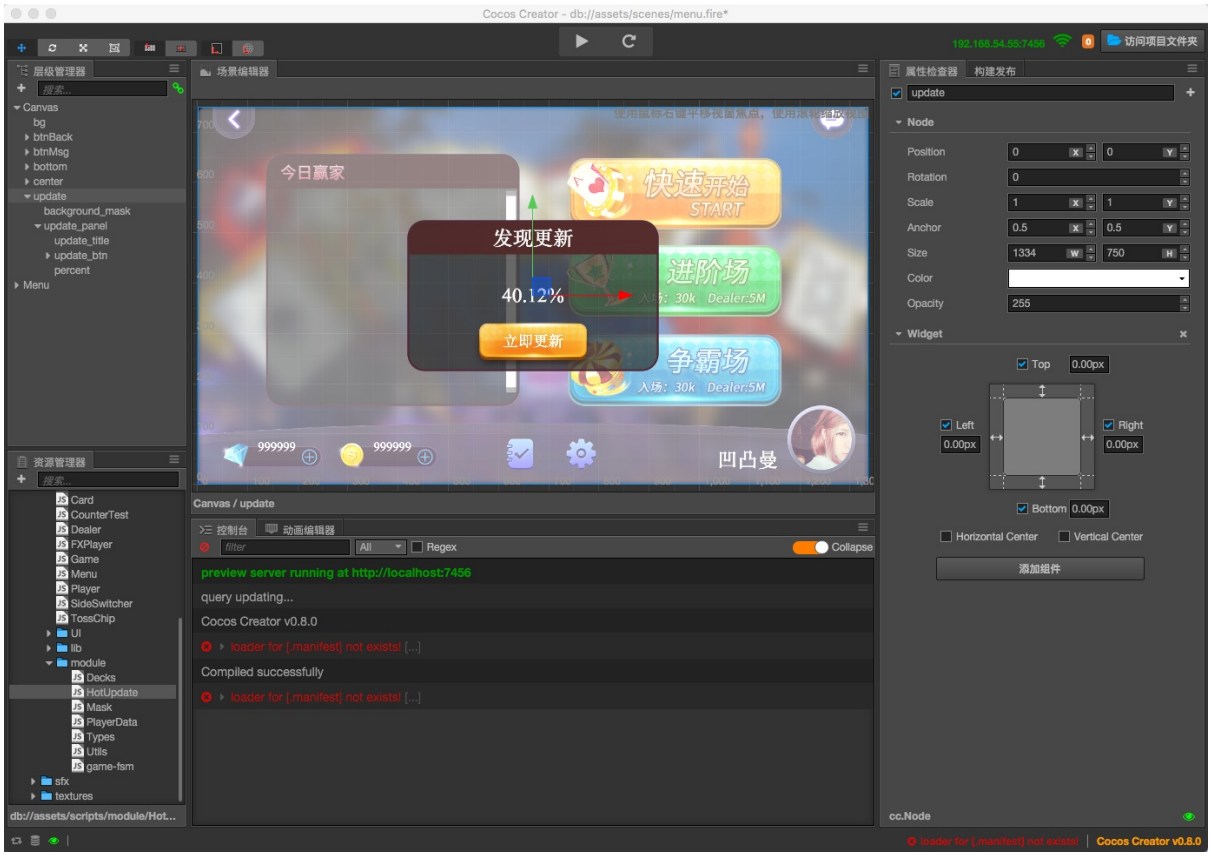
下面是参数说明：

- `-v` 指定 Manifest 文件的主版本号。
- `-u` 指定服务器远程包的地址，这个地址需要和最初发布版本中 Manifest 文件的远程包地址一致，否则无法检测到更新。
- `-s` 本地原生打包版本的目录相对路径。
- `-d` 保存 Manifest 文件的地址。

## 热更新组件

在范例工程中，热更新组件的实现位于 `assets/scripts/module/HotUpdate.js` 中，开发者可以参考这种实现，也可以自由得按自己的需求修改。

除此之外，范例工程中还搭配了一个 `Canvas/update` 节点用于提示更新和显示更新进度供参考。




## 部署远程服务器

为了让游戏可以检测到远程版本，可以在本机上模拟一个远程服务器，搭建服务器的方案多种多样（比如 Python SimpleHTTPServer），这里不做讨论，开发者可以使用自己习惯的方式。搭建成功后，访问远程包和 Manifest 文件的地址与范例工程中不同，所以需要修改以下几个地方来让游戏可以成功找到远程包：

1. `assets/project.manifest`：游戏的本地 Manifest 文件中的 `packageUrl`、`remoteManifestUrl` 和 `remoteVersionUrl`
2. `remote-assets/project.manifest`：远程包的 Manifest 文件中的 `packageUrl`、`remoteManifestUrl` 和 `remoteVersionUrl`
3. `remote-assets/version.manifest`：远程包的 Version 文件中的 `packageUrl`、`remoteManifestUrl` 和 `remoteVersionUrl`

## 打包原生版本

下载完成范例工程后，可以用 Cocos Creator 直接打开这个工程。打开 `构建发布` 面板，构建原生版本，建议使用 Windows / Mac 来测试。

构建成功原生版本之后，打开原生发布包的地址，给 `main.js` 附加上搜索路径设置的逻辑：

```
// 在 main.js 的开头添加如下代码
if (cc.sys.isNative) {
    var hotUpdateSearchPaths = cc.sys.localStorage.getItem('HotUpdateSearchPaths');
    if (hotUpdateSearchPaths) {
        jsb.fileUtils.setSearchPaths(JSON.parse(hotUpdateSearchPaths));
    }
}
```

或者直接使用项目仓库根目录下的 `main.js` 覆盖原生打包文件夹内的 `main.js`。注意，每次使用 Cocos Creator 构建后，都需要重新修改 `main.js`。

这一步是必须要做的原因是，热更新的本质是用远程下载的文件取代原始游戏包中的文件。Cocos2d-x 的搜索路径恰好满足这个需求，它可以用来指定远程包的下载地址作为默认的搜索路径，这样游戏运行过程中就会使用下载好的远程版本。另外，这里搜索路径是在上一次更新的过程中使用 `cc.sys.localStorage`（它符合 WEB 标准的 Local Storage API）固化保存在用户机器上，`HotUpdateSearchPaths` 这个键值是在 `HotUpdate.js` 中指定的，保存和读取过程使用的名字必须匹配。

此外，打开工程过程中如果遇到这个警告可以忽略：`loader for [.manifest] not exists!`。

### 运行范例工程

如果一切正常，此时运行原生版本的范例工程，就会发现检测到新版本，提示更新，更新之后会自动重启游戏，此时可进入 table 场景。


## 结语

以上介绍的是目前一种可能的热更新方案，Cocos Creator 在未来版本中提供更成熟的热更新方案，直接集成到编辑器中。当然，也会提供底层 Downloader API 来允许用户自由实现自己的热更新方案，并通过插件机制在编辑器中搭建完整可视化的工作流。这篇教程和范例工程提供给大家参考，也鼓励开发者针对自己的工作流进行定制。如果有问题和交流也欢迎反馈到论坛中。

## Next Step

1. 热更新管理器文档

# 热更新管理器 AssetsManager

这篇文档将全面覆盖热更新管理器 AssetsManager 的设计思路，技术细节以及使用方式。由于热更新机制的需求对于开发者来说可能各不相同，在维护过程中开发者也提出了各个层面的各种问题，说明开发者需要充分了解热更新机制的细节才能够定制出符合自己需要的工作流。所以这篇文档比较长，也尽力循序渐进得介绍热更新机制，但是并不会介绍过多使用层面的代码，对于想要先了解具体如何使用热更新机制来更新自己游戏的开发者，可以先尝试我们提供的一个简易教程。

## 资源热更新简介

资源热更新是为游戏运行时动态更新资源而设计的，这里的资源可以是图片，音频甚至游戏逻辑。在游戏漫长的运营维护过程中，你将可以上传新的资源到你的服务器，让你的游戏跟踪远程服务器上的修改，自动下载新的资源到用户的设备上。就这样，全新的设计，新的游玩体验甚至全新的游戏内容都将立刻被推送到你的用户手上。重要的是，你不需要针对各个渠道去重新打包你的应用并经历痛苦的应用更新审核！

资源热更新管理器经历过三个重要的阶段：

1. 在 Cocos2d-JS v3.0 中初次设计并实现。
2. 在 Cocos2d-x v3.9 中升级了 Downloader 和多线程并发实现。
3. 在 Cocos Creator v1.4.0 和 Cocos2d-x v3.15 中经过一次重大重构，系统性解决了热更新过程中的 bug。

所以请配合使用最新版本的引擎来使用，这篇文档也是基于最后一次重构来编写的。

## 设计目标和基本原理

热更新机制本质上是从服务器下载需要的资源到本地，并且可以执行新的游戏逻辑，让新资源可以被游戏所使用。这意味着两个最为核心的目标：下载新资源，覆盖使用新逻辑和资源。同时，由于热更新机制最初在 Cocos2d-JS 中设计，我们考虑了什么样的热更新机制才更适合 Cocos 的 JavaScript 用户群。最终我们决定使用类似 Web 网页的更新模式来更新游戏内容，我们先看一下 Web 的更新模式：

1. Web 页面在服务端保存完整的页面内容
2. 浏览器请求到一个网页之后会在本地缓存它的资源
3. 当浏览器重新请求这个网页的时候会查询服务器版本的最后修改时间（Last-Modified）或者是唯一标识（Etag），如果不同则下载新的文件来更新缓存，否则继续使用缓存

浏览器的缓存机制远比上面描述的要复杂，不过基本思路我们已经有了，那么对于游戏资源来说，也可以在资源服务器上保存一份完整的资源，客户端更新时与服务端进行比对，下载有差异的文件并替换缓存。无差异的部分继续使用包内版本或是缓存文件。这样我们更新游戏需要的就是：

1. 服务端保存最新版本的完整资源（开发者可以随时更新服务器）
2. 客户端发送请求和服务端版本进行比对获得差异列表
3. 从服务端下载所有新版本中有改动的资源文件
4. 用新资源覆盖旧缓存以及应用包内的文件

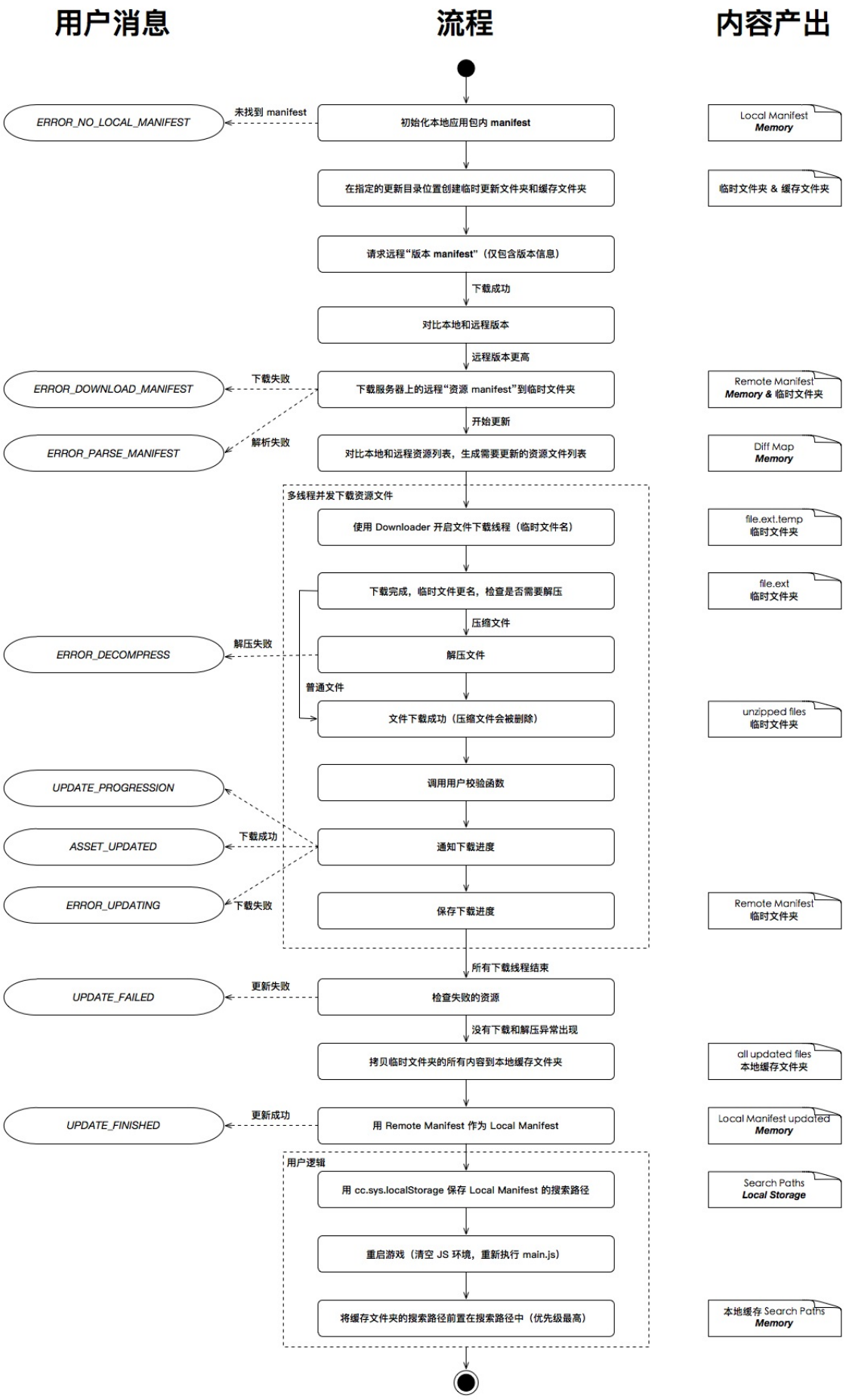
这就是整个热更新流程的设计思路，当然里面还有非常多具体的细节，后面会结合实际流程进行梳理。这里需要特别指出的是：

**Cocos 默认的热更新机制并不是基于补丁包更新的机制，传统的热更新经常对多个版本之间分别生成补丁包，按顺序下载补丁包并更新到最新版本。Cocos 的热更新机制通过直接比较最新版本和本地版本的差异来生成差异列表并更新。**这样即可天然支持跨版本更新，比如本地版本为 A，远程版本是 C，则直接更新 A 和 C 之间的差异，并不需要生

成 A 到 B 和 B 到 C 的更新包，依次更新。所以，在这种设计思路下，新版本的文件以离散的方式保存在服务端，更新时以文件为单位下载。

# 热更新基本流程

在理解了上面基本的设计思路之后，我们来看一次典型的热更新流程。我们使用 manfiest 资源描述文件来描述本地或远程包含的资源列表及资源版本，manifest 文件的定义会在后面详述。运行环境假定为用户安装好 app 后，第一次检查到服务端的版本更新：


用户消息
流程
内容产出
ERROR_NO_LOCAL_MANIFEST
未找到 manifest
初始化本地应用包内 manifest
Local Manifest
Memory
在指定的更新目录位置创建临时更新文件夹和缓存文件夹
临时文件夹 & 缓存文件夹
请求远程“版本 manifest”（仅包含版本信息）
下载成功
对比本地和远程版本
远程版本更高
ERROR_DOWNLOAD_MANIFEST
下载失败
下载服务器上的远程“资源 manifest”到临时文件夹
Remote Manifest
Memory & 临时文件夹
开始更新
ERROR_PARSE_MANIFEST
解析失败
对比本地和远程资源列表，生成需要更新的资源文件列表
Diff Map
Memory
多线程并发下载资源文件
使用 Downloader 开启文件下载线程（临时文件名）
file.ext.temp
临时文件夹
下载完成，临时文件更名，检查是否需要解压
file.ext
临时文件夹
压缩文件
ERROR_DECOMPRESS
解压失败
解压文件
普通文件
文件下载成功（压缩文件会被删除）
unzipped files
临时文件夹
调用用户校验函数
UPDATE_PROGRESSION
ASSET_UPDATED
下载成功
通知下载进度
ERROR_UPDATING
下载失败
保存下载进度
Remote Manifest
临时文件夹
所有下载线程结束
UPDATE_FAILED
更新失败
检查失败的资源
没有下载和解压异常出现
拷贝临时文件夹的所有内容到本地缓存文件夹
all updated files
本地缓存文件夹
UPDATE_FINISHED
更新成功
用 Remote Manifest 作为 Local Manifest
Local Manifest updated
Memory
用户逻辑
用 cc.sys.localStorage 保存 Local Manifest 的搜索路径
Search Paths
Local Storage
重启游戏（清空 JS 环境，重新执行 main.js）
将缓存文件夹的搜索路径前置在搜索路径中（优先级最高）
本地缓存 Search Paths
Memory

上图分为三个部分，中间是热更新的流程，左边是更新过程中 AssetsManager 向用户发送的消息，右边则是各个步骤产出的中间结果，其中粗体字表示中间结果所在的位置，比如内存中、临时文件夹中或者是缓存文件夹。

相信看完这张图还是有很多疑问，下面会从细节上来解析各个步骤中需要注意或者不容易理解的地方。

# 技术细节解析

## Manifest 格式

Manifest 格式是我们用来比较本地和远程资源差异的一种 json 格式，其中保存了主版本信息、引擎版本信息、资源列表及资源信息等：

```
{
    "packageUrl" :          远程资源的本地缓存根路径
    "remoteVersionUrl" :    [可选项] 远程版本文件的路径，用来判断服务器端是否有新版本的资源
    "remoteManifestUrl" :   远程资源 Manifest 文件的路径，包含版本信息以及所有资源信息
    "version" :             资源的版本
    "engineVersion" :       引擎版本
    "assets" :              所有资源列表
        "key" :             资源的相对路径（相对于资源根目录）
        "md5" :             md5 值代表资源文件的版本信息
        "compressed" :      [可选项] 如果值为 true，文件被下载后会自动被解压，目前仅支持 zip 压缩格式
        "size" :            [可选项] 文件的字节尺寸，用于快速获取进度信息
    "searchPaths" :         需要添加到 FileUtils 中的搜索路径列表
}
```

Manifest 文件可以通过 Cocos Creator 的热更新范例中的 Version Generator 脚本来自动生成。

这里需要注意的是，remote 信息（包括 packageUrl、remoteVersionUrl、remoteManifestUrl）是该 manifest 所指向远程包信息，也就是说，当这个 manifest 成为本地包或者缓存 manifest 之后，它们才有意义（偷偷透露个小秘密，更新版本时更改远程包地址也是一种玩法呢）。另外，md5 信息可以不是文件的 md5 码，也可以是某个版本号，这完全是由用户决定的，本地和远程 manifest 对比时，只要 md5 信息不同，我们就认为这个文件有改动。

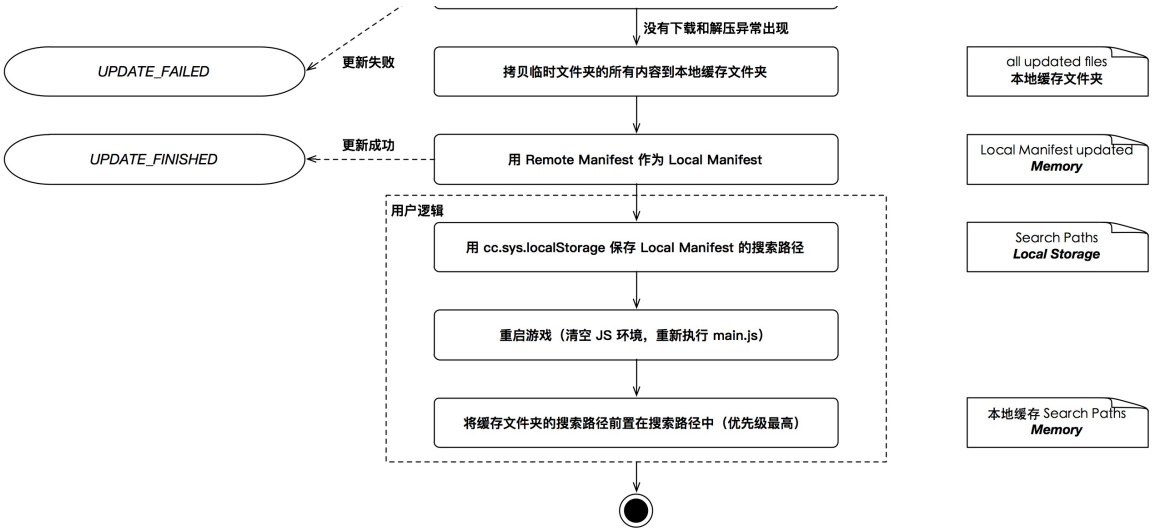
## 工程资源和游戏包内资源的区别

大家在创建一个 Cocos Creator 工程的时候，可以看到它的目录下有 assets 目录，里面保存了你的场景、脚本、prefab 等，对应编辑器中的 assets 面板。但是这些工程资源并不等同于打包后的资源，在使用构建面板构建原生版本时，我们会在构建目录下找到 res 和 src 文件夹，这两个文件夹内保存的才是真正让游戏运行起来的游戏包内资源。其中 src 包含所有脚本， res 包含所有资源。

所以我们的资源热更新自然应该更新构建出来的资源，而不是工程的 assets 目录。

## 包内资源、本地缓存资源和临时资源

在开发者的游戏安装到用户的手机上时，它的游戏是以 .ipa（iOS）或者 .apk（Android）形式存在的，这种应用包在安装后，它的内容是无法被修改或者添加的，应用包内的任何资源都会一直存在。所以热更新机制中，我们只能更新本地缓存到手机的可写目录下（应用存储空间或者 SD 卡指定目录），并通过 FileUtils 的搜索路径机制完成本地缓存对包内资源的覆盖。同时为了保障更新的可靠性，我们在更新过程中会首先将新版本资源放到一个临时文件夹中，只有当本次更新正常完成，才会替换到本地缓存文件夹内。如果中途中断更新或者更新失败，此时的失败版本都不会污染现有的本地缓存。这一步骤在上一章节的流程图中有详细介绍：




在长期多次更新的情况下，本地缓存会一直被替换为最新的版本，而应用包只有等到用户在应用商店中更新到新版本才会被修改。

## 进度信息

在前面章节的大图中，可以看到热更新管理器有发送 UPDATE_PROGRESSION 消息给用户。目前版本中，用户接收到的进度信息包含字节级进度和文件级进度（百分比数值）：

```
function updateCb (event) {
    switch (event.getEventCode())
    {
        case jsb.EventAssetsManager.UPDATE_PROGRESSION:
            cc.log("Byte progression : " + event.getPercent() / 100);
            cc.log("File progression : " + event.getPercentByFile() / 100);
            break;
    }
}
```

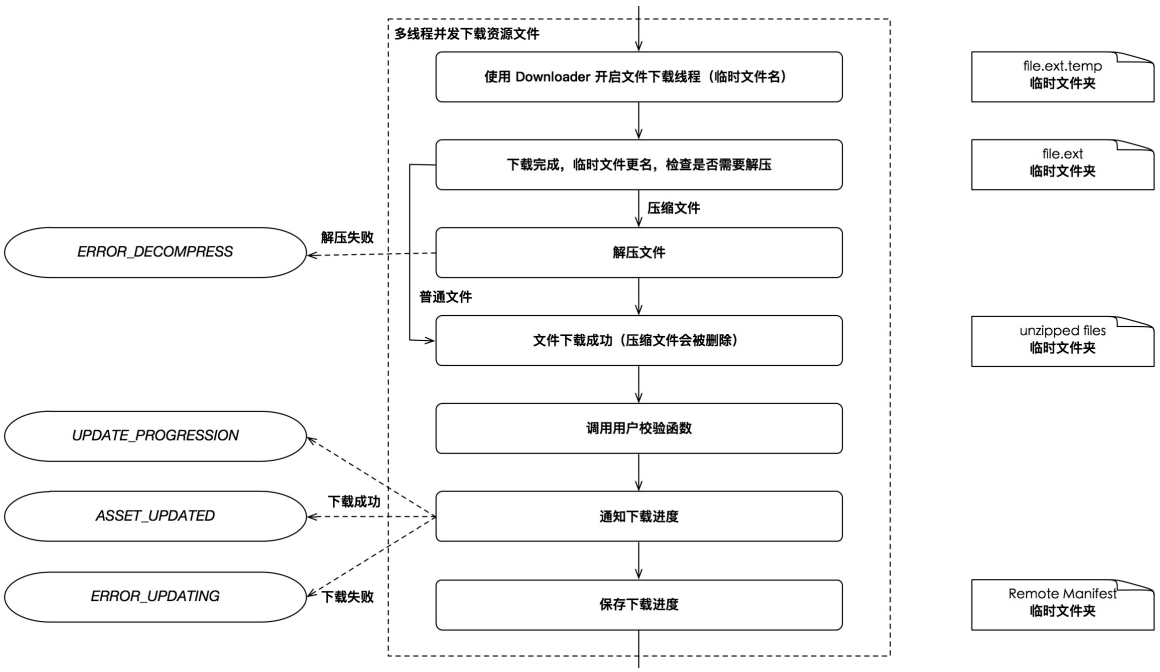
我们会在 Cocos Creator 1.5 中添加更多信息：

1. 字节级进度（百分比）
2. 文件级进度（百分比）
3. 已接收到的字节数
4. 总字节数
5. 已接收到的文件数
6. 总文件数

## 断点续传

肯定有开发者会问，如果在更新过程中网络中断会怎么样？答案是热更新管理器支持断点续传，并且同时支持文件级别和字节级别的断点续传。

那么具体是怎么做的呢？首先我们使用 Manifest 文件来标识每个资源的状态，比如未开始、下载中、下载成功，在热更新过程中，文件下载完成会被标识到内存的 Manifest 中，当下载完成的文件数量每到一个进度节点（默认以 10% 为一个节点）都会将内存中的 Manifest 序列化并保存到临时文件夹中。具体的步骤展示在流程图多线程并发下载资源部分：




在中断之后，再次启动热更新流程时，会去检查临时文件夹中是否有未完成的更新，校验版本是否和远程匹配后，则直接使用临时文件夹中的 Manifest 作为 Remote Manifest 继续更新。此时，对于下载状态为已完成的，不会重新下载，对于下载中的文件，会尝试发送续传请求给服务器（服务器需要支持 `Accept-Ranges`，否则从头开始下载）。

## 控制并发

Cocos Creator v1.4 和 Cocos2d-x v3.15 开始，热更新管理器添加了控制下载并发数量的 API，使用方式如下：

```
assetsManager.setMaxConcurrentTask(10);
```

## 版本对比函数

热更新流程中很重要的步骤是比较客户端和服务端的版本，默认情况下只有当服务端主版本比客户端主版本更新时才会去更新。引擎中实现了一个版本对比的函数，它的最初版本使用了最简单的字符串比较而为人诟病，比如会出现 1.9 > 1.10 的情况。在 Cocos Creator v1.4 和 Cocos2d-x v3.15 之后，我们升级为支持 x.x.x.x 四个序列版本的对比函数（x 为纯数字），不符合这种版本号模式的情况下会继续使用字符串比较函数。

除此之外，我们还允许用户使用自己的版本对比函数，使用方法如下：

```
// versionA 和 versionB 都是字符串类型
assetsManager.setVersionCompareHandle(function (versionA, versionB) {
    var sub = parseFloat(versionA) - parseFloat(versionB);
    // 当返回值大于 0 时，versionA > versionB
    // 当返回值等于 0 时，versionA = versionB
    // 当返回值小于 0 时，versionA < versionB
    return sub;
});
```

## 下载后文件校验

由于下载过程中仍然有小概率可能由于网络原因或其他网络库的问题导致下载的文件内容有问题，所以我们提供了用户文件校验接口，在文件下载完成后热更新管理器会调用这个接口（用户实现的情况下），如果返回 true 表示文件正常，返回 false 表示文件有问题。

```
assetsManager.setVerifyCallback(function (filePath, asset) {
    var md5 = calculateMD5(filePath);
    if (md5 === asset.md5)
        return true;
    else
        return false;
});
```

由于 Manifest 中的资源版本建议使用 md5 码，那么在校验函数中计算下载文件的 md5 码去和 asset 的 md5 码比对即可判断文件是否正常。除了 md5 信息之外，asset 对象还包含下面的属性：

1. md5： md5 码
2. path： 服务器端相对路径
3. compressed： 是否为压缩文件
4. size： 文件尺寸
5. downloadState： 下载状态，包含 UNSTARTED、DOWNLOADING、SUCCESSED、UNMARKED

## 错误处理和失败重试

在流程图的左侧，大家应该注意到了不少的用户消息，这些用户消息都是可以通过热更新的事件监听器来获得通知的，具体可以参考范例中热更新组件的实现。流程图标识了所有错误信息的触发时机和原因，开发者可以根据自己的系统设计来做出相应的处理。

最重要的就是当下载过程中出现异常，比如下载失败、解压失败、校验失败，最后都会触发 UPDATE_FAILED 事件，此时热更新管理器中记录了所有失败的资源列表，开发者可以通过很简单的方式进行失败资源的下载重试：

```
assetsManager.downloadFailedAssets();
```

这个接口调用之后，会重新进入热更新流程，仅下载之前失败的资源，整个流程是和正常的热更新流程一致的。

## 重启的必要性

在论坛中经常会有开发者提问，能不能够不重启就直接启用热更新下来的资源？答案是，对。

有两个原因，第一是更新下来的脚本需要干净的 JS 环境才能正常运行。第二是场景配置，AssetsLibrary 中的配置都需要更新到最新才能够正常加载场景和资源。

1. JS 脚本的刷新

   在热更新完成后，游戏中的所有脚本实际上已经执行过，所有的类、组件、对象都已经存在 JS context 中了，此时如果不重启直接加载脚本，同名的类和对象的确会被覆盖，但是已经用旧的类创建的对象是一直存在的，而被直接覆盖的全局对象在运行过程中修改的状态也全部丢失了。试想一下旧版本的对象和新版本的对象在一起打架的场景，一定很壮观。我还没有提对内存造成的额外开销呢。

2. 资源配置的刷新

   在 Cocos2d-x/JS 中，的确可以做到不重启直接启用新的贴图、字体、音效等资源，但是这点在 Cocos Creator 中并不成立，原因在于 Cocos Creator 的资源也依赖于配置，场景依赖于 settings.js 中的场景列表，而 raw assets 依赖于 settings.js 中的 raw assets 列表。如果 settings.js 没有重新执行，并被 main.js 和 AssetsLibrary 重新读取，那么游戏中是加载不到新的场景和资源的。

上面是热更新后必须要重启的原因，不过如何启用新的资源呢？那就需要依赖 Cocos 引擎的搜索路径机制了，Cocos 中所有文件的读取都是通过 FileUtils 来读取的，而 FileUtils 会按照搜索路径的优先级顺序查找文件。那么解决方案就很简单了，只要我们将热更新的缓存目录添加到搜索路径中，并且前置，那么就会优先搜索到缓存目录中的资源。下面是示例代码：

```
if (cc.sys.isNative) {
    // 创建 AssetsManager
    var assetsManager = new jsb.AssetsManager(manifestUrl, storagePath);
    // 初始化后的 AssetsManager 的 local manifest 就是缓存目录中的 manifest
    var hotUpdateSearchPaths = assetsManager.getLocalManifest().getSearchPaths();
    // 默认的搜索路径
    var searchPaths = jsb.fileUtils.getSearchPaths();

    // hotUpdateSearchPaths 会前置在 searchPaths 数组的开头
    Array.prototype.unshift.apply(searchPaths, hotUpdateSearchPaths);

    jsb.fileUtils.setSearchPaths(searchPaths);
}
```

值得一提的是，这段代码必须放在 main.js 中 require 其他脚本之前执行，否则还是会加载到应用包内的脚本。

# 进阶主题

上面的章节相信已经覆盖了热更新管理器的大部分实现和使用细节，应该可以解答开发者的大多数疑问，不过在一些特殊的应用场景下，可能需要一些特殊的技巧来避免热更新引发问题。

## 迭代升级

对于游戏开发者来说，热更新是比较频繁的需求，从一个大版本升级到另一个大版本的过程中，中间可能会发布多个热更新版本。那么下面两个问题是开发者比较关心的：

1. 在本地缓存覆盖过程中会发生什么问题

   当用户环境中已经包含一个本地缓存版本时，热更新管理器会比较缓存版本和应用包内版本，使用较新的版本作为本地版本。如果此时远程版本有更新，热更新管理器在更新过程中，按照正常流程会使用临时文件夹来下载远程版本。当远程版本更新成功后，临时文件夹的内容会被复制到本地缓存文件夹中，如果有同名文件则被覆盖，最后删除临时文件夹。注意，这个过程中并不会删除本地缓存中的原始文件，因为这些文件仍然可能是有效的，只是它们没有在这次版本中被修改。

   所以理论上小版本的持续热更新不会遇到什么问题。

2. 游戏大版本更新过程中，应该怎么处理

   在游戏包更新过程中，可能开发者会想要彻底清理一次本地的热更新缓存。那么有很多种做法，比如可以记录当前的游戏版本，检查 cc.sys.localStorage 中保存的版本是否匹配，如果不匹配则可以做一次清理操作：

   ```
   // 之前版本保存在 local Storage 中的版本号，如果没有认为是旧版本
   var previousVersion = parseFloat( cc.sys.localStorage.getItem('currentVersion') );
   // game.currentVersion 为该版本的常量
   if (previousVersion < game.currentVersion) {
       // 热更新的储存路径，如果旧版本中有多个，可能需要记录在列表中，全部清理
       jsb.fileUtils.removeDirectory(storagePath);
   }
   ```

## 更新引擎

升级游戏使用的引擎版本是可能会对热更新产生巨大影响的一个决定，开发者们可能注意过在原生项目中存在 src/jsb_polyfill.js 文件，这个文件是 JS 引擎编译出来的，包含了对 C++ 引擎的一些接口封装和 Entity Component 层的代码。在不同版本的引擎中，它的代码会产生比较大的差异，而 C++ 底层也会随之发生一些改变。这种情况下，如果游戏包内的 C++ 引擎版本和 src/jsb_polyfill.js 的引擎版本不一致，就可能导致严重的问题，甚至游戏完全无法运行。

建议更新引擎之后，尽量推送大版本到应用商店，如果不想要更新大版本，请一定要仔细完成各个旧版本更新新版本的测试。

# i18n 游戏多语言支持

游戏多语言支持是通过 Cocos Creator 编辑器扩展插件实现的，这个插件实现了 Label 和 Sprite 组件的多语言国际化（i18n）。

注意，多语言国际化和本地化的区别是，国际化需要软件里包括多种语言的文本和图片数据，并根据用户所用设备的默认语言或菜单选择来进行实时切换。而本地化是在发布软件时针对某一特定语言的版本定制文本和图片内容。

本插件是多语言支持插件，因此不包括构建项目时去除一部分多语言数据的功能。

# 插件安装方法

## 通过扩展商店安装

请参考 扩展编辑器:安装与分享 文档。

## 手动安装

请先下载多语言支持插件包下载。

## 范例项目

下载 i18n多语言支持范例项目 并打开，确保 i18n 插件已经加载，然后就可以对照实际项目来测试多语言功能了。

# 语言配置

首先从主菜单打开 i18n 面板： `插件->i18n` 。

然后需要创建包含多语言翻译数据的 JSON 文件（为了方便使用采用 .js 格式存储）：

- 在 `Manage Languages` 部分的 `New Language ID` 输入框里输入新增语言的 ID，如 `zh` （代表中文）， `en` （代表英文）等。
- 输入 ID 后点击 `Create` 按钮，会在相关的语言选择菜单里增加一种语言，并且会在项目的 `resources/i18n` 目录下创建对应语言的翻译数据模板，如 `resources/i18n/zh.js`

接下来在 i18n 面板的 `Preview Language` 部分的下拉菜单里就可以选择编辑器里预览时的语言了。

# 本地化 Label 文本

## 添加 Localize 组件

i18n 插件提供了两种组件分别用于配合 Label 和 Sprite 来显示多语言内容。

我们从 Label 开始，我们可以在场景或 prefab 中任何 Label 组件所在的节点上添加 `i18n/LocalizedLabel` 组件。这个组件只需要输入翻译数据索引的 dataID 就可以根据当前语言来更新 Label 的字符串显示。

下面我们来介绍如何配置 dataID。

## 翻译数据

插件创建的翻译数据模板是这样的：

```
// zh.js

if (!window.i18n) window.i18n = {};
window.i18n.zh={
// write your key value pairs here
    "label_text": {
        "hello": "你好！",
        "bye": "再见！"
    }
};
```

其中 `window.i18n.zh` 全局变量的写法让我们可以在脚本中随时访问到这些数据，而不需要进行异步的加载。

在大括号里面的内容是用户需要添加的翻译键值对，我们使用了 AirBnb 公司开发的 Polyglot 库来进行国际化的字符串查找，翻译键值对支持对象嵌套、参数传递和动态修改数据等功能，非常强大。更多用法请阅读上面链接里的文档。

如果像上面例子里一样设置我们的翻译数据，那么就会生成如下的键值对：

- "label_text.hello" : "你好！"
- "label_text.bye" : "再见！"

## 查看效果

接下来我们只要在 LocalizedLabel 组件的 `dataID` 属性里写入 `label_text.hello`，其所在节点上的 Label 组件就会显示 `你好！` 文字。

运行时如果需要修改 Label 渲染的文字，也请对 `LocalizedLabel.dataID` 进行赋值，而不要直接更新 `Label.string`。

当需要预览其他语言的显示效果时，打开 i18n 面板，并切换 `Preview Language` 里的语言，场景中的 Label 显示就会自动更新。

## 运行时设置语言

游戏运行时可以根据用户操作系统语言或菜单选择来设置语言，在获取到需要使用的语言 ID 后，需要用以下的代码来进行初始化：

```
const i18n = require('LanguageData');
i18n.init('zh'); // languageID should be equal to the one we input in New Language ID input field
```

需要在之后动态切换语言时也可以调用 `i18n.init()`。

注意运行时必须保证 `i18n.init(language)` 在包含有 LocalizedLabel 组件的场景加载前执行，否则将会因为组件上无法加载到数据而报错。

## 脚本中使用翻译键值对获取字符串

除了和 LocalizedLabel 配合使用解决场景中静态 Label 的多语言问题，`LanguageData` 模块还可以单独在脚本中使用，提供运行时的翻译：

```
const i18n = require('LanguageData');
i18n.init('en');
let myGreeting = i18n.t('label_text.hello');
cc.log(myGreeting); // Hello!
```

# 本地化 Sprite 图片

## 添加 LocalizedSprite 组件

首先在场景或 prefab 中任何 Sprite 组件所在的节点上添加 `i18n/LocalizedSprite` 组件。该组件需要我们手动添加一组语言 id 和 SpriteFrame 的映射，就可以在编辑器预览和运行时显示正确语言的图片了。

## 添加语言图片映射

负责承载语言到贴图映射的属性 `spriteFrameSet` 是一个数组，我们可以像操作其他数组属性一样来添加新的映射

- 首先设置数组的大小，要和语言种类相等
- 为每一项里的 `language` 属性填入对应语言的 id，如 `en` 或 `zh`
- 将语言对应的贴图（或 SpriteFrame）拖拽到 `spriteFrame` 属性里。

完成设置后，点击下面的 `Refresh` 按钮，就可以在场景中看到效果了。

和 `LocalizedLabel` 一样，当我们在 i18n 面板设更改了预览语言时，当前场景里所有的 `LocalizedSprite` 也会自动刷新，显示当前语言对应的图片。

# 存储和读取用户数据

我们在游戏中通常需要存储用户数据，如音乐开关、显示语言等，如果是单机游戏还需要存储玩家存档。 Cocos Creator 中我们使用 `cc.sys.localStorage` 接口来进行用户数据存储和读取的操作。

`cc.sys.localStorage` 接口是按照 Web Storage API 来实现的，在 Web 平台运行时会直接调用 Web Storage API，在原生平台上会调用 sqlite 的方法来存储数据。一般用户不需要关心内部的实现。

配合本篇文档可以参考 数据存储范例。

## 存储数据

`cc.sys.localStorage.setItem(key, value)`

上面的方法需要两个参数，用来索引的字符串键值 `key`，和要保存的字符串数据 `value`。

假如我们要保存玩家持有的金钱数，假设键值为 `gold`：

`cc.sys.localStorage.setItem('gold', 100);`

对于复杂的对象数据，我们可以通过将对象序列化为 JSON 后保存：

```
userData = {
    name: 'Tracer',
    level: 1,
    gold: 100
};

cc.sys.localStorage.setItem('userData', JSON.stringify(userData));
```

## 读取数据

`cc.sys.localStorage.getItem(key)`

和 `setItem` 相对应， `getItem` 方法只要一个键值参数就可以取出我们之前保存的值了。对于上文中储存的用户数据：

```
var userData = JSON.parse(cc.sys.localStorage.getItem('userData'));
```

## 移除键值对

当我们不再需要一个存储条目时，可以通过下面的接口将其移除：

`cc.sys.localStorage.removeItem(key)`

## 数据加密

对于单机游戏来说，对玩家存档进行加密可以延缓游戏被破解的时间。要加密存储数据，只要在将数据通过 `JSON.stringify` 转化为字符串后调用你选中的加密算法进行处理，再将加密结果传入 `setItem` 接口即可。

您可以搜索并选择一个适用的加密算法和第三方库，比如 encryptjs， 将下载好的库文件放入你的项目，存储时：

```
var encrypt=require('encryptjs');
var secretkey= 'open_sesame'; // 加密密钥
```

```
var dataString = JSON.stringify(userData);
var encrypted = encrypt.encrypt(dataString,secretkey,256);

cc.sys.localStorage.setItem('userData', encrypted);
```

读取时：

```
var cipherText = cc.sys.localStorage.getItem('userData');
var userData=JSON.parse(encrypt.decrypt(cipherText,secretkey,256));
```

**注意** 数据加密不能保证对用户档案的完全掌控，如果您需要确保游戏存档不被破解，请使用服务器进行数据存取。

# 引擎定制工作流程

Cocos Creator 的引擎部分包括 JavaScript 和 C++ 两个部分。全部都在 github 上开源。地址在：

- Creator-JS 引擎：https://github.com/cocos-creator/engine
- Cocos2d-x 引擎：https://github.com/cocos-creator/cocos2d-x-lite

我们建议您通过 github 的 fork 工作流程来维护自己定制的仓库，具体操作方式请阅读 github help: Fork A Repo。关于更多 github 相关工作流程请参考 github help。

# 定制 JavaScript 引擎

如果您仅需要定制 Web 版游戏的引擎功能，或只需要修改纯 JavaScript 层逻辑（如 UI 系统，动画系统），那么您只要按照下面的流程修改 JS 引擎就可以了。

## 获取 JS 引擎

首先您需要从 github 上 clone Creator-JS 引擎的原始（地址见上文）或 fork 后的版本。根据不同的 Creator 版本，还需要 checkout 不同的分支，例如 Creator 1.1.2 对应的是引擎的 v1.1 分支。下载后存放到任意本地路径，在命令行中进入此路径。

## 安装编译依赖

```
# 安装 gulp 构建工具
npm install -g gulp
# 在命令行中进入引擎路径
npm install
```

## 进行修改然后编译

接下来您可以定制引擎修改了，修改之后请在命令行中执行：

```
gulp build
```

来编译将引擎源码编译到 `bin` 目录下。

## 在 Cocos Creator 中使用定制版引擎

通过 **偏好设置** 面板的 原生开发环境 分页设置。设置使用您本地定制后的 JS 引擎路径。

# 定制 Cocos2d-x 引擎

如果您需要定制渲染和原生接口相关的引擎功能，在修改 JS 引擎的基础上，还需要同步修改 Cocos2d-x 的 C++ 引擎。注意 Cocos Creator 使用的 Cocos2d-x 引擎是专门定制的，需要从上文中指定的 github 仓库下载。

和 JS 引擎类似，C++ 引擎在使用前也请确认当前所在分支，对于 Cocos Creator v1.2.0 版本请使用 `v1.2` 分支。

## 初始化

下载或克隆好引擎仓库后，在命令行进入引擎路径然后执行：

```
# 安装编译依赖
npm install
# 下载依赖包，需要提前配置好 python
python download-deps.py
# 同步子 repo，需要提前配置好 git
git submodule update --init
```

## 在 Cocos Creator 中使用定制版引擎

通过 **偏好设置** 面板的 原生开发环境 分页设置。设置使用您本地定制后的 Cocos2d-x 引擎路径。

## 修改引擎

接下来您可以对 Cocos2d-x 引擎进行定制了，由于只有在 **构建发布** 过程中才会编译代码，所以修改引擎后可以直接打开 **构建发布** 面板，选择 `default` 项目模板进行构建和编译。

## 编译预编译库和模拟器

如果想在 **构建发布** 面板中使用 `binary` 预编译库模板加速编译过程，就需要在 Cocos2d-x 引擎路径下执行：

```
# 通过 cocos console 生成预编译库
gulp gen-libs
```

要在模拟器中预览您的引擎修改，需要执行以下命令来重新编译模拟器

```
# 通过 cocos console 生成模拟器
gulp gen-simulator
gulp update-simulator-config
```

# JSB 绑定流程

如果您需要在 JavaScript 引擎和 C++ 引擎同步修改内容，应该完成 JSB 绑定。请参考以下指导文章：

- Cocos 中的脚本绑定
- Cocos 中的自动绑定

# 脏矩形优化

## 脏矩形对 Canvas 渲染的优化情况

在canvas模式下，游戏的渲染需要采用大量的drawImage来实现，渲染性能较低。对于2D的游戏来说，大部分场景中的不少元素都处于静态，屏幕上只有部分区域有显示变化，只有这部分区域才需要更新渲染。

脏矩形的机制能够在canvas模式下，只渲染屏幕上变化的区域，因此能大大降低引擎的drawCall数量，提升渲染性能。

以demoUI为例，在开启脏矩形的情况下，整个渲染drawcall数据从366降低到28。


渲染元素能使用脏矩形的前提条件是能够让渲染器得到自己在屏幕的渲染区域，也就是包围盒AABB，它受局部坐标系下的包围盒(localBB or localBoundingBox)和坐标变换(world Tranform)的影响。默认情况下localBB通过contentSize来取得，用户也可以在canvasRenderCommand中实现getLocalBB接口，返回localBB。

## 脏矩形的开关和调试

在 Creator 中，脏矩形默认处于开启状态。开启和关闭脏矩形的API是

```
//enable dirty region
cc.renderer.enableDirtyRegion(true);
//disable dirty region
cc.renderer.enableDirtyRegion(false);
//check dirty region enable state
```

```
var isEnabled = cc.renderer.isDirtyRegionEnabled();
```

当屏幕上有大量的物体都在运动时，使用脏矩形的效率会更低，因此引擎提供Threshold机制, 在大量物体运动的情况下，渲染自动切换到原始渲染逻辑。默认情况下，threshold是10。

```
//set dirty region threshold
cc.renderer.setDirtyRegionCountThreshold(threshold);
```

脏矩形的调试:

```
//debug dirty region or not
cc.renderer._debugDirtyRegion = true;
cc.renderer._debugDirtyRegion = false;
```

# 脏矩形的渲染组件兼容性情况

目前脏矩形对渲染组件的兼容性情况：

- 完全兼容的组件：Sprite，Label，Mask
- 部分兼容组件：Particle，TileMap，Spine
- 待兼容组件：Graphics

除此之外，如果一个自定义渲染组件没有contentSize并且没有实现getLocalBB接口，其与脏矩形也会不兼容。

# 已知脏矩形优化对浏览器的不兼容情况

UC浏览器，IE浏览器与脏矩形机制不兼容，这两个浏览器上脏矩形已经被禁用。

# BMFont 与 UI 合图自动批处理

## 前言

**本篇文档基于 Cocos Creator v1.4.0 完成**

UI 界面里面经常会使用一些 Label 组件来进行描述性的说明，而这些 Label 组件会打断引擎本身的自动批处理功能，导致 UI 界面的 Draw Call 非常的高。 由于 BMFont 和艺术数字也是使用图片进行渲染，所以我们在 1.4 版本添加了 BMFont 和艺术数字与其它 UI 图片进行合图渲染的功能。

使用这个功能非常的简单，你只需要参考 自动图集资源 新建一个自动图集资源配置，然后把所有你希望进行合图的 UI 图片， BMFont 和艺术数字都拖到自动图集资源所在的目录即可。如果你希望最后 UI 的 Draw call 数量尽可能低，你需要保证合图的数量尽可能的少。最好是使用一张合图就可以 容纳所有的 UI 元素，这样理论上，如果你不使用系统字体和 TTF 字体，你的 UI 界面的 Draw call 数量可以达到 1。

## 注意事项

1. 由于 creator 的合图功能是在项目导出的时候进行的，所以需要对发布的项目进行合图批次渲染功能测试。
2. 在导入 BMFont 的资源的时候，需要把 .fnt 和相应的 png 图片放在同一个目录下面。
3. LabelAtlas 底层渲染采用的跟 BMFont 一样的机制，所以也可以和 BMFont 及其它 UI 元素一起合图来实现批次渲染。

# 如何在 Android 平台上使用 JavaScript 直接调用 Java 方法

使用 Creator 打包的安卓原生应用中，我们可以通过反射机制直接在 JavaScript 中调用 Java 的静态方法。它的使用方法很简单：

```
var o = jsb.reflection.callStaticMethod(className, methodName, methodSignature, parameters...)
```

在 `callStaticMethod` 方法中，我们通过传入 Java 的类名，方法名，方法签名，参数就可以直接调用 Java 的静态方法，并且可以获得 Java 方法的返回值。下面介绍的类名和方法签名可能会有一点奇怪，但是 Java 的规范就是如此的。

## 类名

参数中的类名必须是包含 Java 包路径的完整类名，例如我们在 `org.cocos2dx.javascript` 这个包下面写了一个 `Test` 类：

```
package org.cocos2dx.javascript;

public class Test {

    public static void hello(String msg){
        System.out.println(msg);
    }

    public static int sum(int a, int b){
        return a + b;
    }

    public static int sum(int a){
        return a + 2;
    }

}
```

那么这个 Test 类的完整类名应该是 `org/cocos2dx/javascript/Test`，注意这里必须是斜线 `/`，而不是在Java代码中我们习惯的点 `.`。

## 方法名

方法名很简单，就是方法本来的名字，例如sum方法的名字就是 `sum`。

## 方法签名

方法签名稍微有一点复杂，最简单的方法签名是 `()V`，它表示一个没有参数没有返回值的方法。其他一些例子：

- `(I)V` 表示参数为一个int，没有返回值的方法
- `(I)I` 表示参数为一个int，返回值为int的方法
- `(IF)Z` 表示参数为一个int和一个float，返回值为boolean的方法

现在有一些理解了吧，括号内的符号表示参数类型，括号后面的符号表示返回值类型。因为 Java 是允许函数重载的，可以有多个方法名相同但是参数返回值不同的方法，方法签名正是用来帮助区分这些相同名字的方法的。

目前 Cocos Creator 中支持的 Java 类型签名有下面 4 种：

| Java类型 | 签名 |
| --- | --- |
| int | I |
| float | F |
| boolean | Z |
| String | Ljava/lang/String; |

# 参数

参数可以是 0 个或任意多个，直接使用 JS 中的 number，bool 和 string 就可以。

# 使用示例

我们将会调用上面的Test类中的静态方法：

```
// 调用hello方法
jsb.reflection.callStaticMethod("org/cocos2dx/javascript/Test", "hello", "(Ljava/lang/String;)V", "this is a message from
js");

// 调用第一个sum方法
var result = jsb.reflection.callStaticMethod("org/cocos2dx/javascript/Test", "sum", "(II)I", 3, 7);
cc.log(result); //10

// 调用第二个sum方法
var result = jsb.reflection.callStaticMethod("org/cocos2dx/javascript/Test", "sum", "(I)I", 3);
cc.log(result); //5
```

在你的控制台会有正确的输出的，这很简单吧。

# 注意

另外有一点需要注意的就是，在 Android 应用中，cocos 引擎的渲染和 JS 的逻辑是在 GL 线程中进行的，而 Android 本身的 UI 更新是在 App 的 UI 线程进行的，所以如果我们在 JS 中调用的 Java 方法有任何刷新 UI 的操作，都需要在 UI 线程进行。

例如，在下面的例子中我们会调用一个Java方法，它弹出一个 Android 的 Alert 对话框。

```
// 给我们熟悉的 AppActivity 类稍微加点东西
public class AppActivity extends Cocos2dxActivity {

    private static AppActivity app = null;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        app = this;
    }

    public static void showAlertDialog(final String title,final String message) {

        // 这里一定要使用 runOnUiThread
```

```
        app.runOnUiThread(new Runnable() {
            @Override
            public void run() {
                AlertDialog alertDialog = new AlertDialog.Builder(app).create();
                alertDialog.setTitle(title);
                alertDialog.setMessage(message);
                alertDialog.setIcon(R.drawable.icon);
                alertDialog.show();
            }
        });
    }
}
```

然后我们在 JS 中调用

```
jsb.reflection.callStaticMethod("org/cocos2dx/javascript/AppActivity", "showAlertDialog", "(Ljava/lang/String;Ljava/lang/S
tring;)V", "title", "hahahahha");
```

这样调用你就可以看到一个 Android 原生的 Alert 对话框了。

# 再加点料

现在我们可以从 JS 调用 Java 了，那么能不能反过来？当然可以！ 在你的项目中包含 Cocos2dxJavascriptJavaBridge，这个类有一个 `evalString` 方法可以执行 JS 代码，它位于 `frameworks\js-bindings\bindings\manual\platform\android\java\src\org\cocos2dx\lib` 文件夹下。我们将会给刚才的 Alert 对话框增加一个按钮，并在它的响应中执行 JS。和上面的情况相反，这次执行 JS 代码必须在 GL 线程中进行。

```
alertDialog.setButton("OK", new DialogInterface.OnClickListener() {
    public void onClick(DialogInterface dialog, int which) {
        // 一定要在 GL 线程中执行
        app.runOnGLThread(new Runnable() {
            @Override
            public void run() {
                Cocos2dxJavascriptJavaBridge.evalString("cc.log(\"Javascript Java bridge!\")");
            }
        });
    }
});
```

这样在点击 OK 按钮后，你应该可以在控制台看到正确的输出。evalString 可以执行任何 JS 代码，并且它可以访问到你在 JS 代码中的对象。

# 如何在 iOS 平台上使用 Javascript 直接调用 Objective-C 方法

使用 Creator 打包的 iOS / Mac 原生应用中，我们也提供了在 iOS 和 Mac 上 JavaScript 通过原生语言的反射机制直接调用 Objective-C 函数的方法，示例代码如下：

```
var result = jsb.reflection.callStaticMethod(className, methodNmae, arg1, arg2, .....);
```

在 `jsb.reflection.callStaticMethod` 方法中，我们通过传入 OC 的类名，方法名，参数就可以直接调用 OC 的静态方法，并且可以获得 OC 方法的返回值。注意仅仅支持调用可访问类的静态方法。

**警告**：苹果 App Store 在 2017 年 3 月对部分应用发出了警告，原因是使用了一些有风险的方法，其中 `respondsToSelector:` 和 `performSelector:` 是反射机制使用的核心 API，在使用时请谨慎关注苹果官方对此的态度发展，相关讨论：JSPatch、React-Native、Weex

## 类

- 参数中的类名，只需要传入 OC 中的类名即可，与 Java 不同，类名并不需要路径。比如你在工程底下新建一个类 `NativeOcClass`，只要你将他引入工程，那么他的类名就是 `NativeOcClass`，你并不需要传入它的路径。

```
import <Foundation/Foundation.h>
@interface NativeOcClass : NSObject
+(BOOL)callNativeUIWithTitle:(NSString *) title andContent:(NSString *)content;
@end
```

## 方法

- JS 到 OC 的反射仅支持 OC 中类的静态方法。
- 方法名比较要需要注意，我们需要传入完整的方法名，特别是当某个方法带有参数的时候，你需要将他的**:**也带上。根据上面的例子。此时的方法名字是 **callNativeUIWithTitle:andContent:**，不要漏掉了他们之间的：。
- 如果是没有参数的函数，那么他就不需要 **:**，如下代码，他的方法名是 `callNativeWithReturnString`，由于没有参数，他不需要 **:**，跟 OC 的 method 写法一致。

```
+(NSString *)callNativeWithReturnString;
```

## 使用示例

- 下面的示例代码将调用上面 `NativeOcClass` 的方法，在 JS 层我们只需要这样调用：

```
var ret = jsb.reflection.callStaticMethod("NativeOcClass",
                                          "callNativeUIWithTitle:andContent:",
                                          "cocos2d-js",
                                          "Yes! you call a Native UI from Reflection");
```

- 这里是这个方法在 OC 的实现，可以看到是弹出一个原生对话框。并把 `title` 和 `content` 设置成你传入的参数，并返回一个 boolean 类型的返回值。

```
+(BOOL)callNativeUIWithTitle:(NSString *) title andContent:(NSString *)content{
```

```
    UIAlertView *alertView = [[UIAlertView alloc] initWithTitle:title message:content delegate:self cancelButtonTitle:@"Ca
ncel" otherButtonTitles:@"OK", nil];
    [alertView show];
    return true;
}
```

- 此时，你就可以在 `ret` 中接收到从 OC 传回的返回值（true）了。

# 注意

在 OC 的实现中，如果方法的参数需要使用 float、int、bool 的，请使用如下类型进行转换：

- **float，int 请使用NSNumber类型**
- **bool 请使用 BOOL 类型**
- 例如下面代码，我们传入 2 个浮点数，然后计算他们的合并返回，我们使用 NSNumber 而不是 int、float去作为参数类型。

```
+(float) addTwoNumber:(NSNumber *)num1 and:(NSNumber *)num2{
    float result = [num1 floatValue]+[num2 floatValue];
    return result;
}
```

- 目前参数和返回值支持 **int, float, bool, string**，其余的类型暂时不支持。

# 第三方SDK集成

- AnySDK
- SDKBox

# AnySDK

**AnySDK**为 CP 提供一套第三方 SDK 接入解决方案，整个接入过程，不改变任何 SDK 的功能、特性、参数等，对于最终玩家而言是完全透明无感知的。 目的是让 CP 商能有更多时间更专注于游戏本身的品质，所有 SDK 的接入工作统统交给我们吧。第三方 SDK 包括了渠道SDK、用户系统、支付系统、广告系统、统计系统、分享系统等等。

# AnySDK Framework

**AnySDK Framework**为 AnySDK 客户端框架，它为开发者提供了统一的 SDK 接口。开发者只需要调用 AnySDK Framework 的接口，即可实现多 SDK 的接入。

详情请阅读 AnySDK Framework 一节。

# Cocos Creator 接入 AnySDK 教程

详情请阅读 Integrate AnySDK 一节。

# AnySDK Framework

Cocos Creator 内置 Cocos 引擎中包含了 AnySDK Framework 资源。即开发者构建发布出的平台工程已经包含了 AnySDK Framework。该章节介绍了如何选择性使用 AnySDK。

# 使用 AnySDK

## 原生

- 开发者可根据 AnySDK 官方文档 直接接入 AnySDK 相关接口

## H5

- 构建时勾选 AnySDK 选项
- 开发者可根据 AnySDK H5 接入文档 直接接入 AnySDK 相关接口

# 无需使用 AnySDK

开发者若不需要使用 AnySDK，目前只支持手动删除相关文件，删除步骤如下:

- 删除 `frameworks/runtime-src/Classes` 下的

```
jsb_anysdk_basic_conversions.cpp
manualanysdkbindings.cpp
jsb_anysdk_protocols_auto.cpp
SDKManager.cpp
jsb_anysdk_basic_conversions.h
manualanysdkbindings.hpp
jsb_anysdk_protocols_auto.hpp
SDKManager.h
```

- 删除 `main.js` 下的

```
// anysdk scripts
if (cc.sys.isNative && cc.sys.isMobile) {
  jsList = jsList.concat(['jsb_anysdk.js', 'jsb_anysdk_constants.js']);
}
```

在需要定制的项目路径下添加一个 build-templates 目录，里面按照平台路径划分子目录， 将删除代码的 `main.js` 拷贝在子目录下 结构类似：

```
project-folder
 |--assets
 |--build
 |--build-templates
      |--web-mobile
            |--main.js
      |--jsb-binary
            |--main.js
      |--jsb-default
            |--main.js
```

- Eclipse 工程
  - 删除 libs 下的 `libPluginProtocol.jar` 文件
  - 删除 res 下的

```
drawable/plugin_btn_close.png
drawable/plugin_ui_ad.png
values-en/plugin_string.xml
values/plugin_string.xml
layout/plugin_ads.xml
layout/plugin_login.xml
```

  - 删除 jni 下的 Android.mk 中 `LOCAL_WHOLE_STATIC_LIBRARIES := PluginProtocolStatic`
  - 删除 jni 下的 Android.mk 中

```
LOCAL_SRC_FILES :=  ../../Classes/SDKManager.cpp \
                    ../../Classes/jsb_anysdk_basic_conversions.cpp \
                    ../../Classes/manualanysdkbindings.cpp \
                    ../../Classes/jsb_anysdk_protocols_auto.cpp
```

  - 删除 jni 下的 Application.mk 宏定义 `APP_CPPFLAGS := -DPACKAGE_AS`
  - 修改 `src/org/cocos2dx/javascript/SDKWrapper.java` 文件中 `private final static boolean PACKAGE_AS = true;` , `true` 修改为 `false`
- Android Studio 工程
  - 删除 libs 下的 `libPluginProtocol.jar`
  - 删除 res 下的

```
mipmap/plugin_btn_close.png
mipmap/plugin_ui_ad.png
values-en/plugin_string.xml
values/plugin_string.xml
layout/plugin_ads.xml
layout/plugin_login.xml
```

  - 删除 jni 下的 Android.mk 中 `LOCAL_WHOLE_STATIC_LIBRARIES := PluginProtocolStatic`
  - 删除 jni 下的 Android.mk 中

```
LOCAL_SRC_FILES :=  ../../Classes/SDKManager.cpp \
                    ../../Classes/jsb_anysdk_basic_conversions.cpp \
                    ../../Classes/manualanysdkbindings.cpp \
                    ../../Classes/jsb_anysdk_protocols_auto.cpp
```

  - 删除 jni 下的 Application.mk 宏定义 `APP_CPPFLAGS := -DPACKAGE_AS`
  - 修改 `src/org/cocos2dx/javascript/SDKWrapper.java` 文件中 `private final static boolean PACKAGE_AS = true;` , `true` 修改为 `false`
- Xcode 工程
  - 删除 `libPluginProtocol.a` 库
  - Xcode 删除 `libPluginProtocol.a` 引用
  - Xcode 删除 `Classes` 下的引用

```
jsb_anysdk_basic_conversions.cpp
manualanysdkbindings.cpp
jsb_anysdk_protocols_auto.cpp
SDKManager.cpp
jsb_anysdk_basic_conversions.h
manualanysdkbindings.hpp
jsb_anysdk_protocols_auto.hpp
```

    - 删除预编译宏 `PACKAGE_AS`
- Web 工程
    - 找到 index.html 文件删除

```
<script charset="utf-8" id="protocols" type="text/javascript">
    var protocols = document.createElement("script");
    protocols.onload = function () {
        anysdk.agentManager.init();
        anysdk.agentManager.loadAllPlugins(function (code, msg) {
          });
    };
    protocols.src = "http://statics.h5.anysdk.com/protocols/protocols.js";
    document.body.appendChild(protocols);
</script>
```

# 删除 AnySDK 后仍需使用

- 使用 Cocos Console 调用命令 `cocos package import -b anysdk -p project-path --anysdk`

# 更新 AnySDK Framework

- 使用 Cocos Console 调用命令 `cocos package update -p project-path --anysdk` 即可实现更新

# 接入常见问题

- 渠道包出现闪退现象
    - 产生原因: 构建出的 Cocos 工程中 `frameworks/runtime-src/Classes/SDKManager.cpp` 的 `loadAllPlugins` 方法已经调用了 `init` 方法，用户在 JS 层调用 `init` 方法无法生效。
    - 解决方案: 用户无需再 JS 层调用 'init' 方法，需使用构建出的 Cocos 工程中 `frameworks/runtime-src/Classes/SDKManager.cpp` 的 `loadAllPlugins` 方法 `init` 方法，传递 appKey、appSecret、privateKey、oauthLoginServer

# Cocos Creator 接入 AnySDK

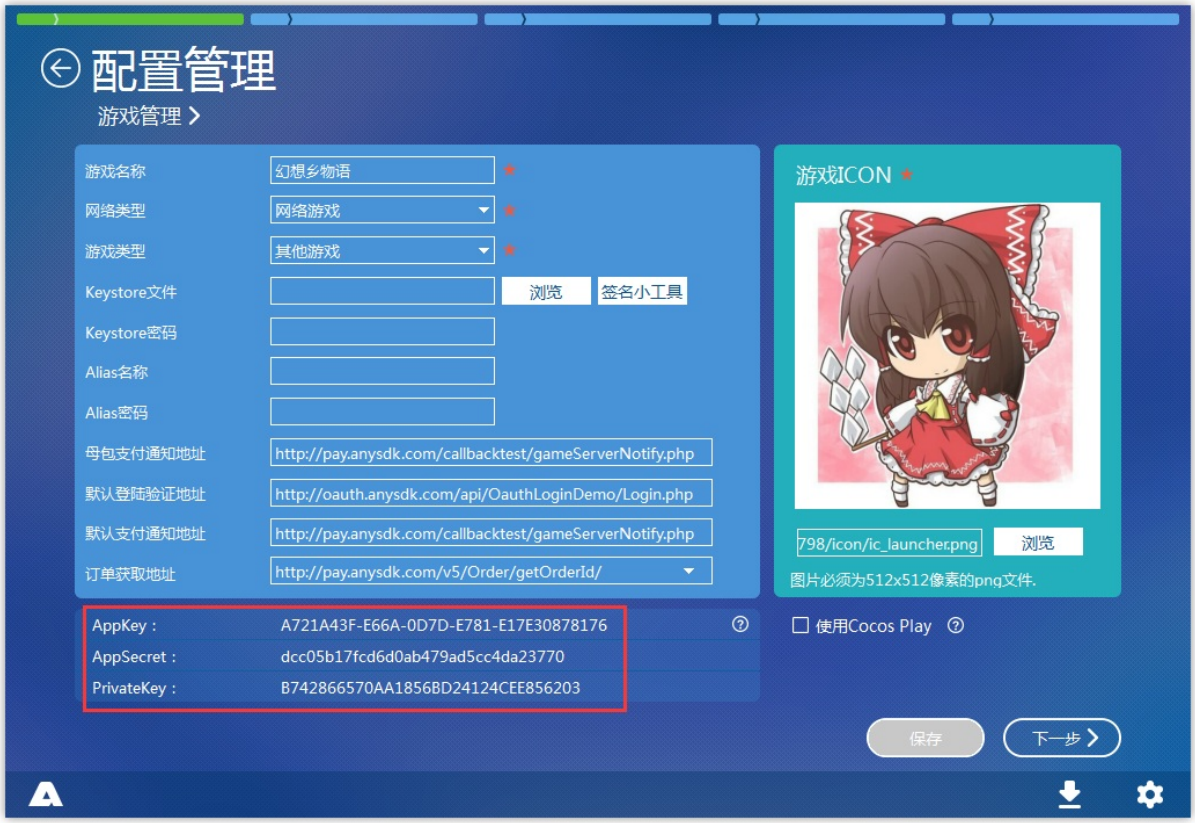
## 概述

从 Creator 1.2 版本起，构建项目的时候就会自动集成 AnySDK 框架，本篇介绍如何接入 AnySDK。更多关于 AnySDK 的资料可以查看 官方wiki。

## 创建游戏

由于目前 AnySDK for Creator 的插件尚未发布，需要从 官网 下载独立的 AnySDK 客户端来进行创建游戏和打包。登陆 AnySDK 客户端，创建新游戏，得到三个参数。

关于 AnySDK 客户端更多的介绍可以参考 客户端使用手册。



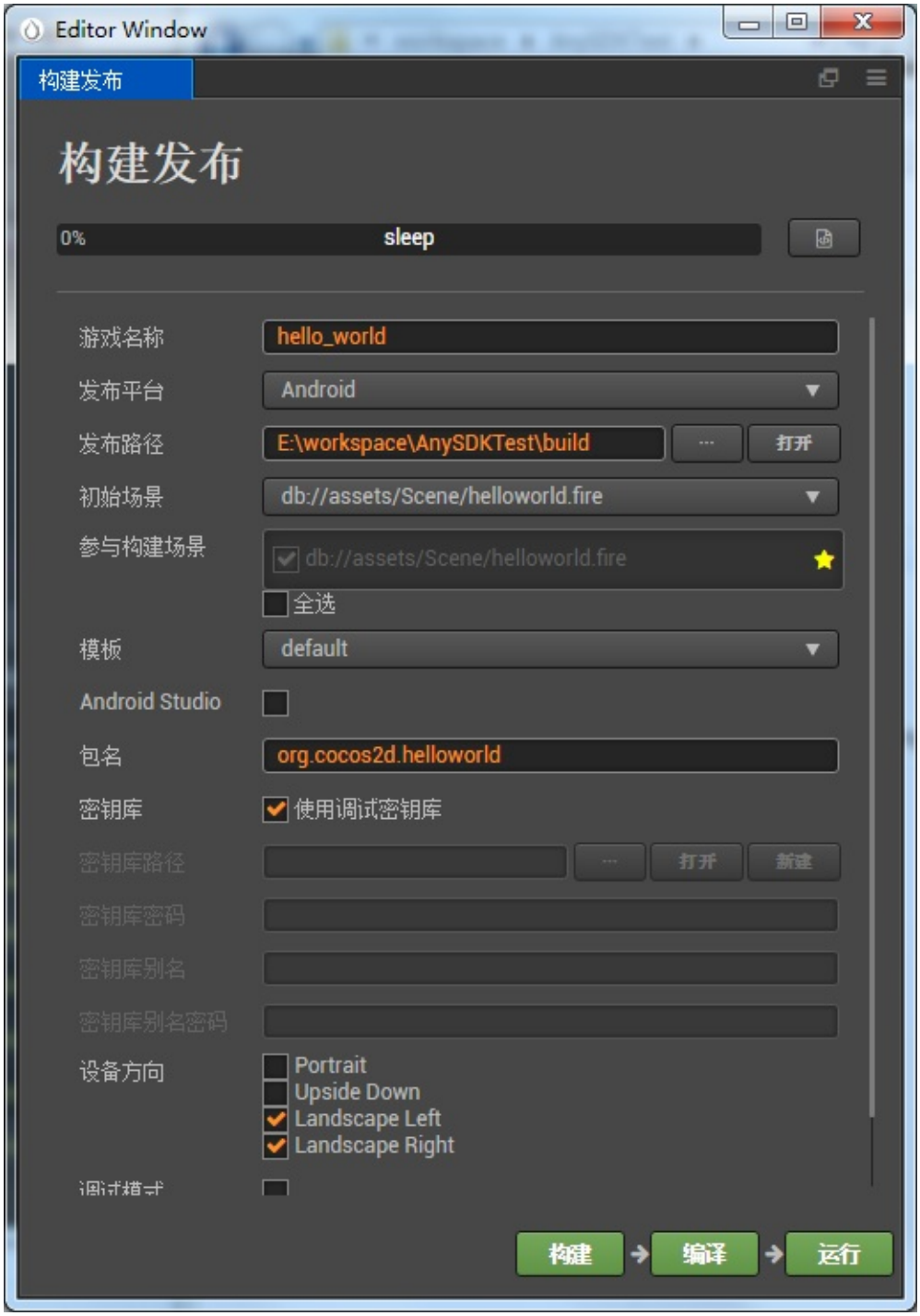

## 服务端接入

如果游戏接入用户和支付的话，需要游戏服务端处理登陆验证和支付验签相关逻辑，参考以下两篇文档进行接入。
统一登陆验证
订单支付通知

## 客户端接入

### 构建项目

点击菜单 **项目** -> **构建发布**，在该界面构建出 Cosos工 程，构建出来的 Cosos 工程已经自动集成好了 AnySDK。




## 调用初始化接口

用户无需手动调用初始化接口，需使用构建出的 Cocos 工程中 `frameworks/runtime-src/Classes/SDKManager.cpp` 的 `loadAllPlugins` 方法 `init` 方法，传递 appKey、appSecret、privateKey、oauthLoginServer

```
//frameworks/runtime-src/Classes/SDKManager.cpp
std::string oauthLoginServer = "OAUTH_LOGIN_SERVER";
std::string appKey = "APP_KEY";
std::string appSecret = "APP_SERCRET";
std::string privateKey = "PRIVATE_KEY";

AgentManager* pAgent = AgentManager::getInstance();
```

```
pAgent->init(appKey,appSecret,privateKey,oauthLoginServer);

//Initialize plug-ins, including SDKs.
pAgent->loadAllPlugins();
```

`appKey` 、 `appSecret` 、 `privateKey` 填写 AnySDK 客户端创建游戏后生成的参数， `oauthLoginServer` 填写游戏服务端用于登陆验证的地址（如不接入用户则随便填写）。

PS： `init` 初始化传的参数如果和打包的游戏的参数不一致，会导致渠道包运行的时候就强制退出。

### 调用各系统接口

根据游戏需求，参考以下文档来调用各个系统的接口。

- 用户系统
- 支付系统
- 统计系统
- 分享系统
- 广告系统
- 推送系统
- 崩溃分析系统
- 广告追踪系统

## 打包

Android：编译生成游戏的 apk，该 apk 将作为 AnySDK 客户端打包用的母包。 iOS：Xcode 工程直接作为 AnySDK 客户端打包用的母工程。 H5：无需打包，只需 AnySDK 客户端添加渠道配置。

在 AnySDK 客户端里添加渠道以及自己所需要接入的 SDK，配置好 SDK 的参数（需要自己上 SDK 的后台申请参数），选择游戏母包即可进行打包。

# 备注

目前 H5 只支持渠道 SDK 接入，并且必须使用 AnySDK 企业版，如有需求可联系 AnySDK 商务。

陈燕淑　商务主管
触控厦门 | AnySDK 项目组
QQ：173732820
Mob：13950013330 （微信）
E-mail：chenys@anysdk.com、yanshu.chen@chukong-inc.com
网址：www.anysdk.com
Add：厦门市观音山商务中心7号楼1302

# SDKBox

SDKBOX 是免费的让移动游戏开发人员简单轻松集成第三方 SDK 的工具，主要面向海外的各种平台和服务。所有的服务都经过严格的测试与第三方平台官方认证。请访问 SDKBox 的主页 获取更多的信息。

在 Cocos Creator 中使用 SDKBox 的工作流程和详细指南请参阅 SDKBox 官方文档。