Introduction

Cocos Creator 3.8 用户手册

欢迎使用 Cocos Creator 3.8！

Cocos Creator 既是一款高效、轻量、免费开源的跨平台 2D&3D 图形引擎，也是一个实时 2D&3D 数字内容创作平台。拥有 **高性能**、**低功耗**、**流式加载**、**跨平台** 等诸多优点，您可以用它来创作 **游戏**、**车机**、**XR**、**元宇宙** 等领域的项目。

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新手教程，可以帮您快速掌握如何使用 Cocos Creator 以及其相关服务。

您可以将此文档从头读到尾，也可以在有需要的时候用来查阅。

如果您是第一次使用 Cocos Creator，可以从 新手上路 和 示例与教程 开始。

如果您已经熟悉其他引擎如 Unity，您可以查看 Unity 开发者入门 Cocos Creator 快速指南 来快上上手 Cocos Creator。

v3.8 新增功能

- 此版本中增加了 **程序化动画**、**高精度文本**、**全新的自定义渲染管线**、**角色控制器** 等全新特性。
- 完整的更新列表请前往 发布说明
- 采用 Cocos Creator 旧版本的项目升级请参考 升级指南

用户手册主要模块

- 场景制作
- 资源系统
- 脚本指南及事件系统
- 发布跨平台游戏
- 图形渲染
- 2D 渲染
- UI 系统
- 动画系统
- 声音系统
- 物理系统
- 粒子系统
- 缓动系统
- 地形系统
- 资源管理
- 本地化
- 扩展编辑器
- 进阶主题

更多内容

- Cocos 官方论坛 可以提问、查找问题答案、与其他开发者交流
- 案例与教程 可以获得教程和官方示例项目
- Cocos Store 可以获得更多素材、学习案例以及源码

产品线简介

Cocos（雅基软件）多年来不断发展，已经发布了多个与 Cocos Creator 密切相关的产品线。为避免混淆，以下是对这些产品的简要介绍：

- **Cocos Creator 3.x**：发布于 2021 年初，是当前 Cocos Creator 的最新版本，已经过大量商业项目验证。3.x 完全摒弃了 Cocos2d-x 底层，采用全新高性能跨平台 3D 内核，标志着 Cocos Creator 正式发展为全面的泛移动端 3D 游戏引擎。由于 3.x 底层已经完全重写，Cocos Creator 不再被视为 Cocos2d-x 的直接扩展和升级版本。
- **Cocos Creator 2.x**：发布于 2018 年，2023 年停止更新。所有能力已被 Cocos Creator 3.x 继承，因此建议新项目使用最新的 Cocos Creator 3.x。
- **Cocos Creator 3D**：2017 年立项，2019 年底以 Cocos Creator 3D 的身份在中国进行了一年多的小规模测试，后正式合并至 Cocos Creator 3.0。由于已被 Cocos Creator 3.x 替代且不再单独更新，提及 Cocos Creator 3D 时通常指代 Cocos Creator 本身的 3D 能力，而非此特定版本。
- **Cocos2d-x**：发布于 2010 年，2019 年停止更新。这是 Cocos2d 社区最活跃的分支，Cocos Creator 2.x 最初采用的底层运行时便是升级过后的 Cocos2d-x。
- **Cocos**：当 Cocos 作为引擎的名字单独出现时，通常代表 Cocos Creator 3.x，而不是 Cocos2d-x。

经过多年的快速发展，Cocos Creator 3.x 与 Cocos Creator 2.x 在用法上已经有所不同，二者的 API 也不完全兼容。因此，在查阅文档、API 和教程时，请开发者注意辨别目标版本是 2.x 还是 3.x，以免因版本不一致导致错误。

关于 Cocos Creator

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，包含了轻量高效的跨平台游戏引擎，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、iOS、Android、各类"小游戏"、PC 客户端等平台，真正实现一次开发，全平台运行。

产品定位

Cocos Creator 是以内容创作为核心，实现了脚本化、组件化和数据驱动的游戏开发工具。具备了易于上手的内容生产工作流，以及功能强大的开发者工具套件，可用于实现游戏逻辑和高性能游戏效果。

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。

获取帮助和支持

获取帮助和支持

官方媒体

- Cocos 引擎官方微信公众号
  - Cocos 论坛社区
  - Cocos 技术支持

或者参考 如何在 GitHub 上向 Creator 提交代码 来提交你的修改。

Cocos Store

- Cocos Store：各类美术资源、插件、源码、学习DEMO以及实用的解决方案

实用的第三方工具

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

图集生产工具

- TexturePacker

位图字体生产工具

- Glyph Designer
- Hiero
- BMFont (Windows)
- snowb-bmf（来自论坛用户 哈库拉玛塔塔 分享）

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

新手上路

新手上路

欢迎使用 Cocos Creator，在学习使用之前，请先参考 安装和启动 安装好 Cocos Creator。

在安装完编辑器之后，可以通过以下内容来熟悉编辑器，包括如何创建项目、项目结构，以及编辑器界面介绍等：

- 使用 Dashboard
- 项目结构
- 编辑器界面

对编辑器有了一定的熟悉和了解之后，便可以通过简单的示例来熟悉 Cocos Creator 的开发流程：

- Hello world!
- 快速上手：制作第一个游戏

同时 Cocos Creator 还提供了很多的范例和教程，并支持其他第三方工具和资源等。开发者也可以直接反馈问题给 Cocos Creator 开发团队：

- 获取帮助和支持
- 注意事项

安装和启动

安装和启动

Cocos Creator 从 v2.3.2 开始接入了全新的 Dashboard 系统，能够同时对多版本引擎和项目进行统一升级和管理！Cocos Dashboard 将做为 Creator 各引擎统一的下载器和启动入口，方便大家升级和管理多个版本的 Creator。此外还集成了统一的项目管理及创建面板，方便大家同时使用不同版本的引擎开发项目

使用 Dashboard

使用 Dashboard

启动 Cocos Dashboard 并使用 Cocos 开发者帐号登录以后，就会打开 Dashboard 窗口，在这里你可以下载引擎、新建项目、打开已有项目或者获得帮助信息。

Hello World!

Hello World 项目

了解 Cocos Dashboard 以后，我们接下来看看如何创建和打开一个 Hello World 项目。

新建项目

在 Cocos Dashboard 的 **项目** 选项卡中，点击右下角的 **新建** 按钮，进入 **新建项目** 页面。选择 **empty** 项目模板，设置好项目名称和项目路径

项目结构

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，开发者项目文件夹的结构如下：

编辑器界面

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：

场景编辑器

场景编辑器

- 摄像机 Gizmo
- 碰撞器 Gizmo
- 粒子系统 Gizmo

层级管理器

层级管理器

层级管理器 面板上主要包括 **工具栏** 和 **节点列表** 两部分内容，用于展现当前场景中可编辑的节点之间的关系。场景中仍有一些不可见的私有节点，不会在此显示。

你可以单选、多选、创建、复制、移动、删除和重命名节点，任意节点都可创建出子节点，子节点的坐标相对于父级节点，跟随父级节点移动。

资源管理器

资源管理器

资源管理器 面板是用于访问和管理项目资源的重要工作区域。在开始制作游戏时，**导入资源** 通常是必须的步骤。在新建项目时可以使用 **HelloWorld** 模板项目，就可以看到 **资源管理器** 中已经包含了一些基本资源类型。

属性检查器

属性检查器

属性检查器 是我们查看并编辑当前选中节点、节点组件和资源的工作区域。在 **场景编辑器** 或者 **层级管理器** 选中节点，或者在 **资源管理器** 选中资源，就可以在 **属性检查器** 中显示并编辑属性。

控制台

控制台

偏好设置

偏好设置

偏好设置 面板中提供了编辑器的个性化设置，点击编辑器主菜单栏中的 **Cocos Creator/File -> 偏好设置** 即可打开。

偏好设置 由几个不同的分页组成，包括 **通用设置**、**外部程序**、**设备管理器**、**引擎管理器**、**资源数据库**、**控制台**、**属性检查器**、**预览**、**构建发布** 和 **实验室**。修改设置之后 **偏好设置** 面板会自动保存修改。

通用设置

项目设置

项目设置

项目设置 面板通过点击编辑器主菜单栏中的 **项目 -> 项目设置** 即可打开，主要用于设置特定项目的相关配置项。这些设置会保存在项目的 `settings/packages` 文件夹中。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。

项目设置 由几个不同的分页组成，包括 **项目数据**、**Layers**、**物理**、**脚本**、**Macro Config**、**功能裁剪** 和 **纹理压缩**。修改设置之后 **项目设置** 面板会自动保存修改。

项目数据

项目数据 分页主要用于设置默认 Canvas、渲染管线等，只对当前项目生效。

排序图层

主菜单

主菜单

Cocos Creator 顶部的主菜单栏中包括 **文件**、**编辑**、**节点**、**项目**、**面板**、**扩展**、**开发者** 和 **帮助** 8 个菜单项，集成了 Cocos Creator 大部分的功能点。

工具栏

工具栏

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

编辑器布局

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 **Cocos Creator/File -> 布局** 菜单，目前只支持 **默认布局**。在默认布局的基础上，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全局目录下：

- **Windows：** `%USERPROFILE%\.CocosCreator\editor\window.json`
- **macOS：** `$HOME/.CocosCreator/editor/window.json`

调整面板大小

将鼠标悬浮到两个面板之间的边界线上，看到鼠标指针变成

预览调试

项目预览调试

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。

在编辑器中选择预览平台

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。

编辑器预览

术语

术语

本章旨在搜集和整理 Cocos Creator 以及和开发相关的一些术语和缩写，并提供解释。

联系我们

我们会持续更新此文档，如果您发现有未经解释的术语或描述不清的情况请点击屏幕右侧的提交反馈或 获取帮助和支持。

编辑器

术语	描述
Editor	通常指的是 Cocos Creator 的编辑器界面
Gizmo	场景编辑器内的一些控制器，比如用于调整节点的位置、旋转、缩放或者光照等等。

图形

术语	描述
VB, Vertex Buffer	顶点缓存，通常是指的一个模型内，顶点的合集，常用数组描述上，数组内的每个元素为一个复合结构，包括位置（Position）、纹理坐标（Tex-coord)、顶点颜色（Color）或者法线（Normal）等属性
IB, Index Buffer	索引缓存，内部存储的是上述 VB 中的顶点的索引，通过这些索引组合出特定的几何形状， 示例： 如 VB 为 { (0, 0, 0), (1, 0, 0), (1, 1, 0) }，IB 为 [0, 2, 1]， 那么此时绘制出的三角形顶点和顺便如下：{ VB[0], VB[2]，VB[1] }
FPS	帧率，指的是每秒中可以渲染的次数
层级，Layer	指的是在渲染时，节点的分组情况，和 相机的 Visibility 属性 配合可以处理某些节点是否被该相机渲染
多层次细节，LOD	Level of details, 通过不同层级拥有不同细节的模型，来提升渲染效率

光照

术语	描述
GI, Global Illumination	全局光照或者局部光照，指的是在计算机图形学中用于计算和模拟真实光源的一组算法
LDR	低动态范围光，通常指单个颜色的色域在 [0,255]（整数） 之间 或者 [0,1.0]（浮点数） 之间的颜色范围，也就是大多数显示设备可显示的颜色空间
HDR	高动态范围光，指的是颜色色域超出上条所述的颜色空间的颜色，显示中的颜色很可能会超出这个颜色范围，所以在显示设备显示时，需要使用 **色调映射** 将其转化到显示器可识别的区域

2D/UI

术语	描述
2D Object	通过在 **层级管理器** 右键选单内创建的 2D 对象
UI Object	通过在 **层级管理器** 右键选单内创建的 UI 对象，通常是封装了某些用于交互的 UI 元素
Sprite Atlas	图集，通过将一些碎片合并成单张的大图用于提高渲染效率，引擎提供 自动合图 功能

动画

术语	描述
Spine	一款 2D 动画制作软件，其输出的动画文件可以被 Cocos Creator 识别
DragonBone	一款 2D 动画制作软件，其输出的动画文件可以被 Cocos Creator 识别
Animation	动画，引擎内置的动画，无蒙皮功能，支持不同节点可以通过编辑器内的动画功能制作，更多
SkeletalAnimation	骨骼动画，带有蒙皮功能的动画组件，更多
Marionette	支持用户自定义状态图的动画系统，和 SkeletalAnimation 配合可以实现丰富的功能，更多

物理

术语	描述
力	使物体可以线性移动的力
扭矩	转矩，使物体旋转的力

2D 物理

术语	描述
物理后端	引擎封装的物理引擎的类型，Cocos Creator 提供两种物理后端分别是 **内置** 和 **Box2D**
Box2D	Box2D 是有名且高效的 2D 物理引擎
刚体	定义了可受力、速度影响的具有质量的物体，通常有三种类型：运动学、动力学以及静态
关节	Joint 2D 物理关节，用于描述物体之间的物理连接情况
物理材质	用于调整两个物理碰撞体碰撞时的摩擦力、弹力处理情况

3D 物理

术语	描述
物理后端	引擎封装的物理引擎的类型，Cocos Creator 提供多种物理引擎包括： **内置**、**Bullet**、**PhysX** 和 **cannon.js** 等，可以在项目设置中修改以适配不同的平台和情况，**注意**：引擎抹平了大部分物理引擎的差异，但某些物理特定仅在特定的物理后端支持。
刚体	定义了可受力、速度影响的具有质量的物体，通常有三种类型：运动学、动力学以及静态
约束	物理约束，用于描述物体之间的物理连接情况
物理材质	用于调整两个物理碰撞体碰撞时的摩擦力、弹力处理情况
力	使物体可以线性移动的力
扭矩	转矩，使物体旋转的力

资源系统

术语	描述
Bundle， Asset Bundle	一种处理资源的方式，引擎会将一个或者多个资源通过算法整合到单个文件内（通常是某种压缩文件）用于提升存储和下载的效率
Resources	特殊目录，引擎会将该目录打包为特殊的 Bundle，通过 `Resources.load` 等方法加载
Cocos Store	Cocos 的资源商店，开发者可以在上面出售和购买资产、源码或完整的商业游戏

脚本与编程

术语	描述
组件	继承自 `cc.Component` 的脚本，可以挂在在节点上
事件	Cocos Creator 基于 Web 的事件冒泡及捕获标准 实现的事件系统，用于响应硬件输入或者节点间的通信，更多

原生开发

术语	描述
原生	指的是 Cocos Creator 提供的，得以不依赖浏览器，在各桌面原生平台或移动原生平台运行的能力
CMake	一种组织 C ++ 代码的文件格式或者工具

其他功能

术语	描述
XR	Extended Reality。适配 OpenXR 的 Cocos Creator 扩展程序，可以协助您制作增加现实功能，更多
AR	Augmented reality，增强现实。将虚拟世界和现实联系起来的技术，您可以使用 Cocos Creator 实现自己的 AR 应用。更多

Unity 开发者快速入门指南

Unity 开发者入门 Cocos Creator 快速指南

随着游戏平台和渠道的种类越来越多，开发者希望自己的游戏能够一次编写多次发布到不同的平台和渠道，而 Cocos Creator 恰好是很好的满足这一需求。

本文将从一个 Unity 开发者的视角，从以下角度对比，帮助 Unity 开发者迅速上手 Cocos Creator引擎。

- 安装和版本管理
- 编辑器
- 资源工作流
- 脚本以及调试
- 着色器

安装和版本管理

Unity Hub 可以使用来管理 Unity 的编辑器版本、项目以及各种模板。在 Cocos Creator 中，同样您也可以通过 Cocos Dash Board 来管理引擎，项目以及模板。

Unity Hub Cocos Dashboard

快速上手：制作第一个 2D 游戏

快速上手：制作第一个 2D 游戏

平台跳跃类游戏是非常常见且热门的游戏类型，从最初的红白机到现代用复杂的 3D 技术制作，平台跳跃在各种游戏平台上层出不穷。

本节我们将使用 Cocos Creator 的 2D 特性，制作一款简单的平台跳跃类游戏。

搭建环境

下载编辑器

访问 Cocos Creator 官网 下载最新的 Cocos Dashboard 即可以对引擎的版本、项目进行统一的管理，安装完成后打开 Dashboard。

监听触摸事件

监听触摸事件

触摸事件可以通过监听 `Input.EventType.TOUCH_START` 来接收屏幕或者鼠标的事件。

监听方式有两种：

- 通过 `input.on` 的方式监听，这种方式会监听屏幕上所有的触摸
- 通过 `node.on` 的方式监听时，可以监听某个范围内的触摸事件

考虑到我们需要操作角色跳一步或者两步，因此我们选择将屏幕的左边的触摸用于处理跳一步的输入，而右边用于跳两步。

在 UI 层级里，也就是 `UICanvas` 节点下方创建两个空的节点，并分别命名为 LeftTouch 和 RightTouch：

快速上手：制作第一个 3D 游戏

快速上手：制作第一个 3D 游戏

本节我们将向您介绍一些 Cocos Creator 的特定，并如何使用 Cocos Creator 制作一个虽然简单但是完整的跳跃游戏。

下面我们将跟随教程制作一款名叫 **一步两步** 的魔性小游戏。这款游戏考验玩家的反应能力，根据路况选择是要跳一步还是跳两步，“一步两步，一步两步，一步一步似爪牙似魔鬼的步伐”。

可以在 这里 体验一下游戏的完成形态。

添加主角

对于绝大多数游戏来说，都需要一个可以操控的角色。接下来我们将一步一步的制作游戏中的主角：胶囊体先生/女士。

为了方便制作，我们在这里稍微回顾下编辑器内是如何创建节点的：

进阶篇 - 添加阴影、光照和骨骼动画

进阶篇 - 添加阴影、光照和动画

在本节中我们将向您介绍如何完善 快速上手：制作第一个游戏 中制作的原型，如何使用第三方资源比如动画资源等等。

光照和阴影

光影是描述游戏的重要渲染特性，通过光源和阴影，我们可以模拟更加真实的游戏世界，提供更好的沉浸感和代入感。

接下来我们为角色加上简单的影子。

开启阴影

1. 在 **层级管理器** 中点击最顶部的 `Scene` 节点，然后在 **属性检查器** 勾选 `shadows` 中的 Enabled，并修改 **Distance** 和 **Normal** 属性：

示例与教程

示例与教程

示例项目

- **一步两步**（GitHub）：也就是 快速上手 文档里分步讲解制作的游戏。
- **材质展示 Demo**（GitHub）：展示各种引擎内置和自定义材质效果，包括标准 PBR、卡通渲染，也包含皮肤、头发、眼球、树叶等高级材质。
- **渲染管线 Demo**（GitHub）：展示如何使用内置后处理渲染管线以及各种后处理特性的开启效果。
- **物理范例集合**（GitHub）：包含了 2D & 3D 的两个物理测试例工程，也包含一些物理小游戏如吞噬黑洞、简化小车、坠落小球等，介绍了一些基础的功能和使用方法，方便用户结合文档了解物理功能。
- **Marionette 动画系统范例**（GitHub）：全方位展示了 Marionette 动画系统的使用，包含动画状态机、事件处理、姿态图、反向动力学 IK 等功能的展示。
- **UI 展示 Demo**（GitHub| Cocos Store）：各类 UI 组件组合使用的演示 Demo。
- **部分功能范例**（GitHub | Gitee）：包括含了引擎网络、NPM 模块等功能的独立示例。
- **模块展示集合**（GitHub）：引擎各个功能的范例项目，基本涵盖了引擎的大部分功能模块，用户在使用功能时可参考此项目中的用法进行开发。
- **实用解决方案**（GitHub）：官方技术支持团队分享的一系列实用的技术解决方案。
- **弹弹乐 3D**（GitHub | Cocos Store）：用户可通过此项目制作弹弹球游戏。
- **快上车 3D**（GitHub | Cocos Store）：基于物理的游戏制作 demo，用户可通过此项目制作快上车游戏。

  所有 GitHub 上的演示和范例项目都会跟随版本进行更新，主分支对应最新的 Cocos Creator 版本，下载的时候请注意。

教程

游戏开发

- Cocos Creator 零基础教程：3D 跑酷游戏开发
- Cocos Creator 零基础教程：3D 俯视角 RPG 割草游戏
- 鹰击长空 3D射击游戏开发
- 快上车 3D游戏开发

图形渲染

- 图形学与Shader渲染实战系列
- Cocos Creator Shader入门基础系列
- 烘焙|粒子|皮肤|车漆

实用方案分享（由官方技术支持团队提供）

- 【CocosCreator 3.x 技术方案分享】第一期
- 【CocosCreator 3.x 技术方案分享】第二期
- 【CocosCreator 3.x 技术方案分享】第三期
- 【CocosCreator 3.x 技术方案分享】第四期
- 【CocosCreator 3.x 自定义渲染材质方案分享】第一期
- 【CocosCreator 3.x 自定义渲染材质方案分享】第二期
- 【CocosCreator 3.x 自定义渲染材质方案分享】第三期
- Cocos Creator 3.x 学习资料整理

更多

- Cocos 官方论坛：提问、查找答案、与其他开发者交流
- Cocos Store：各类美术资源、插件、源码、学习DEMO
- 哔哩哔哩 — Cocos 官方：不定期直播、以及官方视频教程等持续更新中

升级指南

升级指南

- Cocos Creator 3.0 升级指南
- v3.0 材质升级指南
- v3.1 材质升级指南
- v3.3 动画剪辑数据升级指南
- v3.5 材质升级指南
- 资源分包升级指南
- 资源管理模块升级指南
- v3.5 已构建工程升级指南
- v3.5 已构建工程升级 v3.6 指南
- v3.6 构建模板与 settings.json 升级指南
- Cocos Creator 3.6 材质升级指南
- 粒子系统着色器 v3.5.x 升级到 v3.6.0
- v3.6 之前版本升级 CMake
- v3.8 Android 工程升级

v3.0 升级指南

Cocos Creator 3.0 升级指南

版本介绍

Cocos Creator 3.0 集成了原有 2D 和 3D 两套产品的所有功能，带来了诸多重大更新，将做为 Creator 之后的主力版本。同时 v3.0 还延续了 Cocos 在 2D 品类上轻量高效的优势，并且为 3D 重度游戏提供高效的开发体验。

- **对于 Cocos Creator 2.x**

  为了保障现有的 v2.4 项目平稳过渡，我们会将 v2.4 做为 LTS（长期支持）版本，提供后续 **两年** 的持续更新！在 **2021** 年，v2.4 将继续更新版本，提供缺陷修复和新的 Cocos Creator 小游戏平台支持，保障大家的项目成功上线。在 **2022** 年我们还将为开发者持续提供 v2.4 的关键问题修复，保障已上线的游戏平稳运营！因此：

  - **现有的 v2.x 项目可以安心继续开发，无需强制升级到 v3.0。**
  - **新项目建议使用 v3.0 版本开发**，我们会不断优化 v3.0 的开发体验和运行效率，支撑好 2D、3D 等不同品类的重度游戏顺利上线。

- **对于 Cocos Creator 3D**

  原有的 Cocos Creator 3D 做为 Creator 的分支版本，已经面向中国进行了长达一年的迭代，成功上线了 **星空大决战**、**最强魔斗士** 等重度项目！Cocos Creator 3.0 发布后，Cocos Creator 3D 也将包含在 v3.0 中，现有的 v1.2 项目都可直接升级，因此 Cocos Creator 3D 后续不会再发布独立版本。

Cocos Creator 3.0 使用了面向未来的全新引擎架构，将为引擎带来高性能、面向数据以及负载均衡的渲染器，并且无缝支持 Vulkan & Metal 多后端渲染，未来还会支持移动端 VR/AR 及部分主机平台。

关于 Cocos Creator 3.0 的详细介绍，请移步 官网更新说明。

如何迁移 Cocos Creator 2.x 项目

虽然 **我们不建议开发中的项目，特别是即将上线的项目强升 v3.0**，但是我们仍在 Cocos Creator 3.0 推出了 v2.x 资源导入工具。此工具支持旧项目资源完美导入，以及代码的辅助迁移。

资源导入

开发者只需要点击主菜单中的 **文件 -> 导入 Cocos Creator 2.x 项目**。

---

v3.0 材质升级指南

Cocos Creator 3.0 材质升级指南

本文将详细介绍 Cocos Creator 2.x 的材质升级到 v3.0 的注意事项。

1. 材质系统基础设计简介

1.1 Cocos Creator 的材质系统框架

材质系统自上至下由四个核心类组成，分别是 Material、Effect、Technique 和 Pass，它们的关系可以通过下面的类图来理解：

---

v3.1 材质升级指南

Cocos Creator 3.1 材质升级指南

本文将详细介绍 Cocos Creator 3.0 的材质升级到 v3.1 的注意事项。

1. 着色器升级与变化

1.1 内置头文件变化

原来 v3.0 的标准着色器头文件 `shading-standard` 在 v3.1 改成了 `standard-surface-entry`，可以使 effect 同时兼容 forward 渲染管线和 deferred 渲染管线。

原来 v3.0 的 `cc-fog` 头文件在 v3.1 改成了 `cc-fog-vs/fs`，拆分成了顶点着色器与片元着色器两个部分。

1.2 顶点着色器

- `gl_Position`

  v3.0 的 `VS` 主函数名称 `vert` 在 v3.1 改成了 `main`，并且新增了宏 `gl_Position`，用于给返回值赋值。

```
CCProgram standard-vs %{
    precision highp float;

    // include your headfile

    #include <cc-fog-vs> // 注意这里头文件名称的变化

    // fill in your data here

    void main () {

        // fill in your data here

        gl_Position = fill in your data result;
    }
}%
```

1.3 片元着色器

- `CC_STANDARD_SURFACE_ENTRY()`

  加载标准着色器头文件 `standard-surface-entry`，使用 v3.1 的标准着色器输出函数 `CC_STANDARD_SURFACE_ENTRY()` 替换原有 v3.0 着色器输出的函数 `frag()`。

```
CCProgram standard-fs %{

    // include your headfile

    #include <cc-fog-fs> // 注意这里头文件名称的变化
    #include <standard-surface-entry> // 注意这里标准着色器头文件的名称变化

    // fill in your data here

    void surf (out StandardSurface s) {

        // fill in your data here

    }
    CC_STANDARD_SURFACE_ENTRY() // 标准着色器输出函数
```

```
}%
```

2. Deferred 渲染管线

2.1 Deferred Render Pipeline

v3.1 的材质系统与 v3.0 最大的区别在于 v3.1 支持了 deferred 渲染管线，引擎自带标准的 `standard-surface-entry` 头文件，可以同时支持 forward 渲染管线和 deferred 渲染管线，用法如下：

```
CCEffect %{
    techniques:

    // fill in your data here

        - &deferred
        vert: // your Vertex shader
        frag: // your Fragment shader
        phase: deferred
        propertyIndex: 0
        blendState:
            targets: // 关闭混合
            - blend: false
            - blend: false
            - blend: false
            - blend: false
            properties: // your properties name

    // fill in your data here

}%

// fill in your data here

CCProgram standard-fs %{
    precision highp float;
    #include <cc-global>
    #include <shared-ubos>
    #include <cc-fog-fs> // 注意这里头文件名称的变化。
    #include <standard-surface-entry> // 注意这里标准着色器头文件的名称变化

    // fill in your data here
    void surf (out StandardSurface s) {

        // fill in your data here

    }
    CC_STANDARD_SURFACE_ENTRY() // 标准着色器输出函数
}%

// fill in your data here
```

2.2 渲染管线判断

在头文件 `standard-surface-entry` 会判断选择了哪条渲染管线，光照计算在文件 `shading-standard-additive`

如果判断是 deferred 渲染管线，会先调用 `deferred-lighting` effect 文件，随后调用光照计算文件 `shading-standard-additive`

```
#define CC_STANDARD_SURFACE_ENTRY()
#if CC_FORWARD_ADD
  #include <shading-standard-additive>

  // fill in your data here

#elif CC_PIPELINE_TYPE == CC_PIPELINE_TYPE_FORWARD  // 判断是否为前向渲染管线

  // fill in your data here

#elif CC_PIPELINE_TYPE == CC_PIPELINE_TYPE_DEFERRED  // 判断是否为延迟渲染管线

 // fill in your data here

#endif
```

3. 参数传输升级

v3.0 顶点着色器往片元着色器传递 shadow 参数的宏为 `CCPassShadowParams`，v3.1 则修改为 `CC_TRANSFER_SHADOW`。

v3.1 顶点着色器往片元着色器传输 `FOG` 参数时，直接使用 `CC_TRANSFER_FOG` 宏。

版本对比：

- v3.0

  ```
  v_fog_factor = CC_TRANSFER_FOG(pos);
  CCPassShadowParams(pos);
  ```

- v3.1

  ```
  CC_TRANSFER_FOG(pos);
  CC_TRANSFER_SHADOW(pos);
  ```

---

v3.3 动画剪辑数据升级指南

v3.3 动画剪辑数据升级指南

在 v3.3 中，Creator 大幅度重构了动画剪辑类，摒弃了之前难以操纵、难以理解的接口，引入了轨道、通道等概念，使动画分量编辑变得更容易。更主要的是动画剪辑统一与其他引擎模块使用公共的曲线对象，而非自立门户定义自己的曲线类型。

资源升级

v3.3 之前的动画剪辑资源使用 v3.3 及后续版本打开后会自动升级至新的数据类型，动画效果不变。

但有以下几种边缘情况需要注意：

- 渐变方式

  旧版本的动画剪辑数据中若使用了非线性且非常量的渐变方式时，运行时仍会生效，但在 **动画编辑器** 中无法再编辑，需要手动在曲线编辑器中重新编辑。

- 动画曲线类型

  旧版本的动画曲线都将被转换为相对应的动画属性轨道，包括以下几种曲线类型：

  - `number`
  - `Vec2`、`Vec3`、`Vec4`
  - `Quat`
  - `Color`
  - `Size`

其余类型的曲线将被转换为对象轨道。详情请参考 动画属性轨道。

- 通过模型文件导入的动画数据将不能再通过代码访问

API 更改

旧版动画剪辑对象的以下字段被废弃：

- `times`
- `curves`
- `commonTargets`

考虑到兼容旧版的使用，我们保留了这些 API 的效果。程序化地设置这些字段后，在 **动画运行之前** 都会被正确地转换为新格式。

---

v3.5 材质升级指南

升级指南： Effect 从 v3.4.x 升级到 v3.5.0

宏标记和函数宏

宏标记和功能宏的效果语法已升级，为避免占用标准的 glsl 定义，项目中旧的 Cocos Shader 文件（*.effect）将自动升级，但如果您使用的是没有配套 *.meta 数据的外部 Cocos Shader 文件，或编写新的 Cocos Shader，则必须注意。

- 宏标记的新语法：`#pragma define meta`
- 函数宏的新语法：`#pragma define`

有关详细信息，请参阅 Cocos Shader 语法 - macro-tags。

模型级别的阴影偏移

在 v3.5 中我们支持对模型设置单独阴影偏移值，可以对简单或复杂曲面上的阴影效果进行详细控制。如果您有任何自定义的 effect 文件，您可能需要升级它们使阴影偏移值生效。

解决 **effect** 文件从v3.4.x，升级到 v3.5.0 后，shadowBias 不生效的问题。

> **注意**：如果禁用灯光的阴影贴图，或者没有在顶点着色器上计算 **CC_TRANSFER_SHADOW(pos)** 则 **忽略** 该材质升级。

升级说明

有 **四个元素** 要添加到效果文件中，如下所示：

1. vs out varying 定义

```
#if CC_RECEIVE_SHADOW
    out mediump vec2 v_shadowBias;
#endif
```

2. vs shadow bias 获取

```
#if CC_RECEIVE_SHADOW
    v_shadowBias = CCGetShadowBias();
#endif
```

3. fs in varying 定义

```
#if CC_RECEIVE_SHADOW
    in mediump vec2 v_shadowBias;
#endif
```

4. fs shadow bias 赋值

```
#if CC_RECEIVE_SHADOW
    s.shadowBias = v_shadowBias;
#endif
```

示例（代码片段）

```
// 顶点着色器
CCProgram standard-vs %{
    // 头文件区域
    #include <cc-xxx>
    ...
    #include <cc-xxxx>

    // vs 输出区域
    out vec3 v_xxx;
    ...

    #if CC_RECEIVE_SHADOW
        out mediump vec2 v_shadowBias;
    #endif

    ...
    out vec3 v_xxxx;

    // vs 执行区域
    void main () {
        xxx;
        ...

        #if CC_RECEIVE_SHADOW
            v_shadowBias = CCGetShadowBias();
        #endif

        ...
        xxxx;
    }
}%

// 片元着色器
CCProgram standard-fs %{
    // 头文件区域
    #include <cc-xxx>
    ...
    #include <cc-xxxx>

    // vs 输入区域
    in vec3 v_xxx;
    ...

    #if CC_RECEIVE_SHADOW
        in mediump vec2 v_shadowBias;
    #endif

    ...
    in vec3 v_xxxx;
```

```
    // ps 执行区域
    void surf (out StandardSurface s) {
        xxx;
        ...

        #if CC_RECEIVE_SHADOW
            s.shadowBias = v_shadowBias;
        #endif

        ...
        xxxx;
    }
}%
```

资源分包升级指南

资源分包升级指南

文：Santy-Wang、Xunyi

本文将详细介绍 Cocos Creator 3D 的小游戏子包升级到 Asset Bundle 的注意事项。v2.4 的资源分包与 v3.0 差别不大，无需升级。

在 v2.4 之前，分包加载 功能仅支持各类小游戏平台，如微信小游戏、OPPO 小游戏等。但随着 Creator 的发展，开发者对分包的需求不断增加，例如多平台支持，原有的分包加载已经远远不能满足了。所以，Creator 从 v2.4 开始正式支持功能更为完整的 **Asset Bundle**。

- 对 **美术策划** 而言，项目中的所有资源，例如场景、动画、Prefab 都不需要修改，也不用升级。
- 对 **程序** 而言，影响主要体现在原先代码中使用的 `loader.downloader.loadSubpackage` 需要改为 Asset Manager 中的 `assetManager.loadBundle`。以下将详细介绍这部分内容。

注意：如果你在旧项目中使用了分包功能，也就是在 **属性检查器** 中勾选了 **配置为子包** 选项，那么当项目升级到 v2.4 之后，将自动转变为一个普通文件夹。你可以参考这里进行 Asset Bundle 的配置：

配置 Asset Bundle

需要手动升级的情况

你在自己的代码中使用了 `loader.downloader.loadSubpackage` 来加载分包。

升级步骤

- **备份好旧项目**
- 在 Dashboard 中使用 Cocos Creator v3.0 打开需要升级分包的旧项目，Creator 会对有影响的资源重新导入。第一次导入时会稍微多花一点时间，导入完毕后就会打开编辑器主窗口。然后使用代码编辑器将所有 `loader.downloader.loadSubpackage` 替换为 `assetManager.loadBundle`。

```
// 修改前
loader.downloader.loadSubpackage('sub1', (err) => {
  loader.loadRes('sub1/sprite-frames/background', SpriteFrame);
});

// 修改后
assetManager.loadBundle('sub1', (err, bundle) => {
  // 传入该资源相对 Asset Bundle 根目录的相对路径
  bundle.load('sprite-frames/background/spriteFrame', SpriteFrame);
});
```

注意：加载 Asset Bundle 中的资源需要使用 Asset Bundle 相关的 API，具体请查看 API 文档 Asset Bundle。

Asset Bundle 的使用方式

关于 Asset Bundle 的具体使用方式，请参考文档 Asset Bundle。

资源管理模块升级指南

资源管理模块升级指南

文：Santy-Wang、Xunyi

本文将详细介绍 Cocos Creator 3D 的 loader 升级到 assetManager 时的注意事项。v2.4 的资源管理与 v3.0 差别不大，无需升级。

在 Cocos Creator 2.4 以前，获取和加载资源 是通过 `loader` 模块（包括 `loader.load`、`loader.loadRes`、`loader.loadResDir` 等系列 API）来实现的，`loader` 模块主要用于加载资源。但随着 Creator 的不断发展，开发者对于资源管理的需求不断增加，原来的 `loader` 已无法满足大量的资源管理需求，一个新的资源管理模块呼之欲出。

因此，Creator 在 v2.4 推出了全新的资源管理模块 —— **Asset Manager**。相较之前的 `loader`，Asset Manager 不但提供了更好的加载性能，而且支持 Asset Bundle、预加载资源以及更加方便的资源释放管理。同时 Asset Manager 还拥有强大的扩展性，大大提升开发者的开发效率和使用体验，我们建议所有开发者都进行升级。

为了带来平滑的升级体验，我们仍保留了对 `loader` 相关 API 的兼容。除个别项目使用了无法兼容的特殊用法的 API 必须手动升级外，大部分项目都可以照常运行。之后我们会在时机成熟时才逐渐完全移除对 `loader` 的兼容。如果由于项目周期等原因暂时不方便升级，你可以在确保测试通过的情况下继续保留原来的写法。

目前在使用旧的 API 时，引擎会输出警告并提示升级方法。请你根据警告内容和本文的说明对代码进行调整，升级到新的用法。比较抱歉的是，由于底层经过了升级，我们遗留了个别无法兼容的 API，在运行时会输出错误信息。如果你已经决定好要进行升级，那么请仔细阅读以下内容。

- 对 **美术策划** 而言，项目中的所有资源，例如场景、动画、Prefab 都不需要修改，也不需要升级。
- 对 **程序** 而言，影响主要体现在原先代码中使用的 `loader` 的所有 API，都需要改为 `assetManager` 的 API。以下将详细介绍这部分内容。

注意：因为 v2.4 支持 Asset Bundle，项目中的分包功能也需要进行升级，具体内容请参考 分包升级指南。

需要手动升级的情况

- 你在自己的代码中使用了以 `loader` 开头的 API，比如 `loader.loaderRes`、`loader.loadResDir`、`loader.release` 等。
- 你在自己的代码中使用了以 `AssetLibrary` 开头的 API，比如 `AssetLibrary.loadAsset`。
- 你在自己的代码中使用了 `url` 开头的 API，比如 `url.raw`。
- 你在自己的代码中使用了 `Pipeline`，`LoadingItems` 等类型。
- 你在自己的代码中使用了 `macro.DOWNLOAD_MAX_CONCURRENT` 属性。

升级步骤

- **备份好旧项目**
- 在 Dashboard 中使用 Cocos Creator v3.0 打开需要升级的旧项目，Creator 将对有影响的资源重新导入，第一次导入时会稍微多花一点时间，导入完毕后就会打开编辑器主窗口。此时可能会出现较多的报错或警告信息，别担心，请打开代码编辑工具根据报错或警告信息对代码进行升级。

将 `loader` 相关的 API 替换为 `assetManager` 相关的 API

从 v2.4 开始，不建议使用 `loader`，并且在后续的版本中也会逐渐被彻底移除，请使用新的资源管理模块 `assetManager` 进行替换。

加载相关接口的替换

如果你在自己的代码中使用了 loader.loadRes、loader.loadResArray、loader.loadResDir，请使用 assetManager 中对应的 API 进行替换。可参考下方的替换方式：

- **loader.loadRes**

  resources.load 的参数与 loader.loadRes 完全相同。替换方式如下：

  ```
  // 修改前
  loader.loadRes(...);

  // 修改后
  resources.load(...);
  ```

- **loader.loadResArray**

  assetManager 为了降低学习成本，将 loadResArray 与 load 进行了合并。resources.load 的第一个参数可支持多个路径，所以可以使用 resources.load 进行替换：

  ```
  // 修改前
  loader.loadResArray(...);

  // 修改后
  resources.load(...);
  ```

- **loader.loadResDir**

  resources.loadDir 的参数与 loader.loadResDir 完全相同：

  ```
  // 修改前
  loader.loadResDir(...);

  // 修改后
  resources.loadDir(...);
  ```

  注意：为了简化接口，resources.loadDir 的加载完成回调将 **不再提供** paths 的列表。请避免以下的使用方式：

  ```
  loader.loadResDir('images', Texture2D, (err, assets, paths) => console.log(paths));
  ```

  如果你想要查询 paths 列表，可以使用以下方式：

  ```
  const infos = resources.getDirWithPath('images', Texture2D);
  let paths = infos.map(function (info) {
      return info.path;
  });
  ```

- **loader.load**

  如果你在自己的代码中使用了 loader.load 来加载远程图片或远程音频，为了方便理解，在 assetManager 中将有专门的 API 用于此项工作，如下所示：

  - 加载远程图片

    ```
    // 修改前
    loader.load('http://example.com/remote.jpg', (err, texture) => console.log(texture));

    // 修改后
    assetManager.loadRemote('http://example.com/remote.jpg', (err, texture) => console.log(texture));
    ```

  - 加载远程音频

    ```
    // 修改前
    loader.load('http://example.com/remote.mp3', (err, audioClip) => console.log(audioClip));

    // 修改后
    assetManager.loadRemote('http://example.com/remote.mp3', (err, audioClip) => console.log(audioClip));
    ```

  - 加载远程文本

    ```
    // 修改前
    loader.load('http://example.com/equipment.txt', (err, text) => console.log(text));

    // 修改后
    assetManager.loadRemote('http://example.com/equipment.txt', (err, textAsset) => console.log(textAsset.text));
    ```

注意：

1. 如果你在自己的代码中使用了 loader.downloader.loadSubpackage 来加载分包，请参考 分包升级指南 进行升级。
2. 为了避免产生不必要的错误，loader.onProgress 在 assetManager 中没有对应实现。你可以自己实现全局回调机制，但建议将回调传入到每个加载函数中，避免并发加载时互相干扰。

释放相关接口的替换

如果你在自己的代码中使用了 loader.release、loader.releaseAsset、loader.releaseRes、loader.releaseResDir，请使用 assetManager 中对应的 API 进行替换。可参考下方的替换方式：

- **loader.release**

  loader.release 可用 assetManager.releaseAsset 替换。

  注意：为了避免开发者关注资源中一些晦涩难懂的属性，assetManager.releaseAsset **不再接受** 数组、资源 UUID、资源 URL 进行释放，仅能通过资源本身进行释放。

  ```
  // 修改前
  loader.release(texture);
  // 修改后
  assetManager.releaseAsset(texture);

  // 修改前
  loader.release([texture1, texture2, texture3]);
  // 修改后
  [texture1, texture2, texture3].forEach(t => assetManager.releaseAsset(t));

  // 修改前
  const uuid = texture._uuid;
  loader.release(uuid);
  // 修改后
  assetManager.releaseAsset(texture);

  // 修改前
  const url = texture.url;
  loader.release(url);
  // 修改后
  assetManager.releaseAsset(texture);
  ```

  注意：为了增加易用性，在 assetManager 中释放资源的依赖资源将 **不再需要** 手动获取资源的依赖项，在 assetManager.releaseAsset 内部将会尝试自动去释放相关依赖资源，例如：

  ```
  // 修改前
  const assets = loader.getDependsRecursively(texture);
  loader.release(assets);

  // 修改后
  assetManager.releaseAsset(texture);
  ```

- **loader.releaseAsset**

  loader.releaseAsset 可直接使用 assetManager.releaseAsset 替换：

  ```
  // 修改前
  loader.releaseAsset(texture);
  ```

```
// 修改后
assetManager.releaseAsset(texture);
```

- **loader.releaseRes**

  `loader.releaseRes` 可直接使用 `resources.release` 替换：

```
// 修改前
loader.releaseRes('images/a', Texture2D);

// 修改后
resources.release('images/a', Texture2D);
```

- **loader.releaseAll**

  `loader.releaseAll` 可直接使用 `assetManager.releaseAll` 替换：

```
// 修改前
loader.releaseAll();

// 修改后
assetManager.releaseAll();
```

注意：

1. 出于安全考虑，`loader.releaseResDir` 在 `assetManager` 中没有对应实现，请使用 `assetManager.releaseAsset` 或 `resources.release` 进行单个资源释放。
2. 因为 `assetManager.releaseAsset` 会自动释放依赖资源，所以你不需要再显式调用 `loader.getDependsRecursively`。如果需要查找资源的相关依赖，请参考 `assetManager.dependUtil` 中相关的 API。
3. 出于安全考虑，`assetManager` 仅支持在场景中设置的自动释放，其他的已移除。`assetManager` 中没有实现 `loader.setAutoRelease`、`loader.setAutoReleaseRecursively`、`loader.isAutoRelease` 这几个 API，建议你使用全新的基于引用计数的自动释放机制，详细请参考 资源释放。

扩展相关接口的替换

- **Pipeline**

  如果你的代码中有使用 `loader.insertPipe`、`loader.insertPipeAfter`、`loader.appendPipe`、`loader.addDownloadHandlers`、`loader.addLoadHandlers` 系列 API 对 `loader` 的加载流程做过扩展，或者直接使用了 `loader.assetLoader`、`loader.md5Pipe`、`loader.downloader`、`loader.loader`、`loader.subPackPipe` 中的方法，请使用 `assetManager` 中对应的 API 进行替换。

  因为 `assetManager` 是更通用的模块，不再继承自 `Pipeline`，所以 `assetManager` 不再实现 `loader.insertPipe`、`loader.insertPipeAfter`、`loader.appendPipe`。具体的替换方式如下：

```
// 修改前
const pipe1 = {
  id: 'pipe1',
  handle: (item, done) => {
    let result = doSomething(item.uuid);
    done(null, result);
  }
};

const pipe2 = {
  id: 'pipe2',
  handle: (item, done) => {
    let result = doSomething(item.content);
    done(null, result);
  }
};

loader.insertPipe(pipe1, 1);
loader.appendPipe(pipe2);

// 修改后
function pipe1 (task, done) {
  let output = [];
  for (let i = 0; i < task.input.length; i++) {
    let item = task.input[i];
    item.content = doSomething(item.uuid);
    output.push(item);
  }

  task.output = output;
  done(null);
}

function pipe2 (task, done) {
  let output = [];
  for (let i = 0; i < task.input.length; i++) {
    let item = task.input[i];
    item.content = doSomething(item.content);
    output.push(item);
  }

  task.output = output;
  done(null);
}

assetManager.pipeline.insert(pipe1, 1);
assetManager.pipeline.append(pipe2);
```

  注意：

  1. `assetManager` **不再继承** 自 `Pipeline`，而是 `assetManager` 下拥有的多个 `Pipeline` 实例。详情请参考 管线与任务。
  2. 为了易用性，Pipe 的定义不再需要定义一个拥有 `handle` 方法和 `id` 的对象，只需要一个方法即可。详情请参考 管线与任务。
  3. 为了简化逻辑、提高性能，Pipe 中处理的内容不再是 `item`，而是 `task` 对象。详情请参考 管线与任务。
  4. 为了降低学习成本，`Pipeline` 中不再支持 `insertPipeAfter` 形式的 API，请使用 `insert` 插入指定的位置。

- **addDownloadHandlers、addLoadHandlers**

  出于模块化考虑，`assetManager` 中没有实现 `addDownloadHandlers`、`addLoadHandlers`，请参考以下方式替换：

```
// 修改前
const customHandler = (item, cb) => {
    let result = doSomething(item.url);
    cb(null, result);
};

loader.addDownloadHandlers({png: customHandler});

// 修改后
const customHandler = (url, options, cb) => {
    let result = doSomething(url);
    cb(null, result);
};

assetManager.downloader.register('.png', customHandler);
```

  或者：

```
// 修改前
```

```
const customHandler = (item, cb) => {
    let result = doSomething(item.content);
    cb(null, result);
};

loader.addLoadHandlers({png: customHandler});

// 修改后
const customHandler = (file, options, cb) => {
    let result = doSomething(file);
    cb(null, result);
};

assetManager.parser.register('.png', customHandler);
```

注意：

1. 因为 **下载模块** 与 **解析模块** 都是依靠 **扩展名** 来匹配对应的处理方式，所以调用 `register` 时，传入的第一个参数需要以 . 开头。
2. 出于模块化的考虑，自定义的处理方法将不再传入一个 `item` 对象，而是直接传入与其相关的信息。`downloader` 的自定义处理方法传入的是 **待下载的 URL**，`parser` 传入的则是 **待解析的文件**。具体的内容请参考 下载与解析。
3. 新的拓展机制提供了一个额外的 `options` 参数，可以极大地增加灵活性。但如果你不需要配置引擎内置参数或者自定义参数，可以无视它。具体内容请参考文档 可选参数。

- **downloader，loader，md5Pipe，subPackPipe**

  `loader.downloader` 可由 `assetManager.downloader` 代替，`loader.loader` 可由 `assetManager.parser` 代替。但其中的接口没有完全继承，具体内容请参考文档 下载与解析 或者 API 文档 assetManager.downloader 和 assetManager.parser。

  注意：出于对性能、模块化和易读性的考虑，`loader.assetLoader`、`loader.md5Pipe`、`loader.subPackPipe` 已经被合并到 `assetManager.transformPipeline` 中，你应该避免使用这三个模块中的任何方法与属性。关于 `assetManager.transformPipeline` 的具体内容可参考 管线与任务。

其他更新

`url` 与 `AssetLibrary` 在 **v2.4** 中已经被移除，请避免使用 `url` 与 `AssetLibrary` 中的任何方法和属性。

`Pipeline` 可由 `AssetManager.Pipeline` 进行替换，请参考以下方式进行替换：

```
// 修改前
const pipe1 = {
    id: 'pipe1',
    handle: function (item, cb) {
        let result = doSomething(item);
        cb(null, result);
    }
}

const pipeline = new Pipeline([pipe1]);

// 修改后
function pipe1 (task, cb) {
    task.output = doSomething(task.input);
    cb(null);
}

const pipeline = new AssetManager.Pipeline('test', [pipe1]);
```

注意：`LoadingItem` 在 `assetManager` 中已经不支持，请避免使用这个类型。

为了支持更多加载策略，`macro.DOWNLOAD_MAX_CONCURRENT` 已经从 `macro` 中移除，你可以用以下方式替换：

```
// 修改前
macro.DOWNLOAD_MAX_CONCURRENT = 10;

// 修改后
assetManager.downloader.maxConcurrency = 10;
```

或者

```
// 修改前
macro.DOWNLOAD_MAX_CONCURRENT = 10;

// 修改后（设置预设值）
assetManager.presets['default'].maxConcurrency = 10;
```

具体内容可参考 下载与解析。

---

v3.5 已构建工程升级指南

v3.5 已构建工程升级指南

从 v3.5 开始，Mac 和 Windows 平台的 `AppDelegate` 已移入引擎内部实现，可以通过重载 `AppDelegate` 的方式来兼容之前版本的用法；`game.cpp` 也进行了调整，已有工程需要重新构建进行升级。

工程升级

检查工程目录下 native/engine 目录是否存在。如果存在，需要删除文件夹，删除前需要做好备份（这个目录如果存在，重新构建时不会自动更新）；不存在，则直接构建即可。

自定义代码迁移方法

之前在 AppDelegate 添加的代码，可以通过下文定制平台和 AppDelegate 进行升级；自定义的 game.cpp 可以通过接口更替即可升级。

平台与 AppDelegate 的定制方法

以 **Mac** 为例：

1、自定义AppDelegate（参考文件名：MyAppdelegate.h，MyAppdelegate.mm）

```
@interface MyAppDelegate : NSObject<AppDelegate>
    // 定义需要重写的方法
    - (void)applicationWillResignActive:(UIApplication *)application;
@end

@implementation MyAppDelegate
- (void)applicationWillResignActive:(UIApplication *)application {
    // 注意：调用父类的方法
    [super applicationWillResignActive:application]
}
@end
```

2、自定义平台（参考文件名：CustomMacPlatform.h）

```
#include "platform/BasePlatform.h"
#include "MyAppDelegate.h"

class CustomMacPlatform : public MacPlatform {
public:
   // 重写平台初始化方法
```

```
    int32_t init() override {
        // 调用父类的方法
        return MacPlatform::init();
    }
    // 这里进入 oc 的消息循环，直到程序退出
    int32_t run(int argc, const char** argv) {
         id delegate = [[MyAppDelegate alloc] init];
         NSApplication.sharedApplication.delegate = delegate;
         return NSApplicationMain(argc, argv);
    }
}
```

3、加载自定义平台（参考文件名：main.mm）

```
#include "CustomMacPlatform.h"

int main(int argc, const char * argv[]) {
    CustomMacPlatform platform;
    if (platform.init()) {
        return -1;
    }
    return platform.run(argc, (const char**)argv);
}
```

game.cpp 迁移方法

- 设置js加密秘钥：jsb_set_xxtea_key -> 设置 `_xxteaKey` 成员变量; 或 调用 `setXXTeaKey`
- 设置调试： jsb_enable_debugger -> 设置 `_debuggerInfo` 结构, 或 调用 `setDebugIpAndPort`
- 设置异常回调：setExceptionCallback -> 重写 `handleException` 接口
- 运行自定义脚本：jsb_run_script -> 调用 `runScript`
- 可以通过使用 `engine` 来添加需要监听的事件, -> `getEngine()->addEventCallback(WINDOW_OSEVENT, eventCb);`
- 自定义的游戏 `CustomGame`，需要注册到引擎 `CC_REGISTER_APPLICATION(CustomGame)` 进行加载；
- `game` 继承于 `cc::BaseGame`, 而 `cc::BaseGame` 继承于 `CocosApplication`，因此可以重写部分实现，增加自定义逻辑；

Native 文件修改

- 替换引用的头文件：`#include "cocos/platform/Application.h"` —> `#include "application/ApplicationManager.h"`
- 使用方式变更：`cc::Application::getInstance()->getScheduler()` -> `CC_CURRENT_ENGINE()->getScheduler()`
- 有自定义 jsb 接口的情况：`native_ptr_to_seval` 替换为 `nativevalue_to_se`

Android 升级指南

JAVA 修改

- **game/AppActivity.java** 以及 **game/InstantActivity.java** 的 `onCreate` 方法中删除如下代码：

```
    // Workaround in https://stackoverflow.com/questions/16283079/re-launch-of-activity-on-home-button-but-only-the-first-time/16447508
    if (!isTaskRoot()) {
        // Android launched another instance of the root activity into an existing task
        //  so just quietly finish and go away, dropping the user back into the activity
        //  at the top of the stack (ie: the last state of this task)
        // Don't need to finish it again since it's finished in super.onCreate .
        return;
    }
```

- **app/AndroidManifest.xml** 执行下列操作：
  - 删除 `application` 标签中的下列代码：`android:taskAffinity=""`
  - 在 `application` 标签中增加下列代码：`android:exported="true"`
- **app/build.gradle** 修改下列代码：

```
"${RES_PATH}/assets" -> "${RES_PATH}/data"
```

CMakeLists.txt 修改

- **android/CMakeLists.txt**
  - LIB_NAME 变更为 CC_LIB_NAME
  - PROJ_SOURCES 变更为 CC_PROJ_SOURCES
  - 增加 set(CC_PROJECT_DIR ${CMAKE_CURRENT_LIST_DIR})
  - 增加 set(CC_COMMON_SOURCES)
  - 增加 set(CC_ALL_SOURCES)
  - 删除下列代码：

```
    ${CMAKE_CURRENT_LIST_DIR}/../common/Classes/Game.h
    ${CMAKE_CURRENT_LIST_DIR}/../common/Classes/Game.cpp

    add_library(${LIB_NAME} SHARED ${PROJ_SOURCES})
    target_link_libraries(${LIB_NAME}
        "-Wl,--whole-archive" cocos2d_jni "-Wl,--no-whole-archive"
        cocos2d
    )
    target_include_directories(${LIB_NAME} PRIVATE
        ${CMAKE_CURRENT_LIST_DIR}/../common/Classes
    )
```

  - 增加代码:

```
    cc_android_before_target(${CC_LIB_NAME})
    add_library(${CC_LIB_NAME} SHARED ${CC_ALL_SOURCES})
    # 此处添加用户依赖库 AAA target_link_libraries(${CC_LIB_NAME} AAA)
    # 此处添加用户自定义文件 xxx/include target_include_directories(${CC_LIB_NAME} PRIVATE ${CMAKE_CURRENT_LIST_DIR}/../common/Classes/xxx/include)
    cc_android_after_target(${CC_LIB_NAME})
```

- **common/CMakeLists.txt**
  - cocos2d-x-lite/ 修改为 engine/native/
  - 文件末尾增加代码：

```
    list(APPEND CC_COMMON_SOURCES
        ${CMAKE_CURRENT_LIST_DIR}/Classes/Game.h
        ${CMAKE_CURRENT_LIST_DIR}/Classes/Game.cpp
    )
```

---

v3.6 已构建工程升级指南

v3.5 已构建工程升级 v3.6 指南

本文将详细介绍 Cocos Creator 原生构建工程从 3.5.0 ~ 3.5.x 升级到 3.6 的注意事项。修改仅针对项目工程下的 native 目录。

构建

使用新版本的引擎打开旧版本的工程，待升级完成之后，构建目标平台。为避免升级失败，请先备份好工程，然后依据下列步骤进行升级。

Android 平台

文件修改

- 删除文件：jni/main.cpp
- android/CMakeLists.txt：删除 `${CMAKE_CURRENT_LIST_DIR}/jni/main.cpp`

编译修改

为减少包体大小，更改了 `CMAKE_C_FLAGS_RELEASE`、`CMAKE_CXX_FLAGS_RELEASE` 编译参数 `visibility` 的默认值：从 `default` 改成了 `hidden`。改完后 arm64-v8a 下的引擎动态库可以减少约 3.5M。针对这个修改，若 release 版本 `jni` 出现接口找不到，请先检查接口是否有添加 `JNIEXPORT` 的声明。例如：

- 旧代码

  ```
  void Java_com_google_android_games_paddleboat_GameControllerManager_onMouseConnected
  ```

- 修改后的代码

  ```
  JNIEXPORT void JNICALL Java_com_google_android_games_paddleboat_GameControllerManager_onMouseConnected
  ```

代码修改

- 有自定义 jsb 接口的工程：须删除与 `NonRefNativePtrCreatedByCtorMap` 相关的代码
- JSB 手动绑定的代码需要置空 `_finalize` 函数, 可参考JSB 2.0 绑定教程 C++ 对象的生命周期管理。

  代码示例如下：

  ```
  static bool js_cc_gfx_Size_finalize(se::State& s) // NOLINT(readability-identifier-naming)
  {
      auto iter = se::NonRefNativePtrCreatedByCtorMap::find(SE_THIS_OBJECT<cc::gfx::Size>(s));
      if (iter != se::NonRefNativePtrCreatedByCtorMap::end())
      {
          se::NonRefNativePtrCreatedByCtorMap::erase(iter);
          auto* cobj = SE_THIS_OBJECT<cc::gfx::Size>(s);
          JSB_FREE(cobj);
      }
      return true;
  }
  SE_BIND_FINALIZE_FUNC(js_cc_gfx_Size_finalize)
  ```

  改为

  ```
  static bool js_cc_gfx_Size_finalize(se::State& s) // NOLINT(readability-identifier-naming)
  {
      return true;
  }
  SE_BIND_FINALIZE_FUNC(js_cc_gfx_Size_finalize)
  ```

脚本编写注意事项

由于 Native 引擎的实现和非 Native 的引擎在实现上略有差异，开发者须了解这些差异，现整理如下：

- `Readonly`：在 Native 平台上，为减少内存使用，获取到的属性是新创建的对象。

  关于 `Readonly` 的说明可参考 开发注意事项 - ReaOnly

---

v3.6 构建模板与 settings.json 升级指南

Cocos Creator 3.6.0 构建模板与 settings.json 升级指南

本文将介绍 Cocos Creator 3.6.0 构建模板与 settings.json 升级注意事项。

在 3.6 版本之前，引擎的代码中都包含了一部分无法被剔除的内置 effect 的数据，这部分数据极大的占用了引擎的包体与拖慢了解析引擎代码的时间。对于大部分项目来说是一种资源浪费。

在之前版本中，构建模板中的 `application.ejs` 与 `game.ejs`（某些平台是 `index.ejs`）中包含了不少引擎启动流程的逻辑代码，这部分逻辑代码较为复杂，定制过程中容易出现问题，需要更加易用与稳定的引擎启动流程的定制方案。

此外，在之前版本中，游戏的主要配置数据存储在 `settings.json` 中, 在游戏运行过程中无法访问这部分数据，对于一些插件来说是极不方便的。

针对以上几个问题，我们在 Cocos Creator 3.6 中，为了优化引擎包体，更好地维护引擎的业务代码，同时能够提供更多的定制空间，我们重构了引擎的启动流程并提供了一个新的 `settings` 配置模块 。在重构的过程中，虽然我们希望能尽可能地保证之前版本项目构建模板与 settings.json 的兼容性，但是某些方面还是引入了一些无法兼容的情况。在这些情况下，你需要手动升级项目构建模板与配置文件的定制内容。

- 对 **美术策划** 而言，项目中的所有资源，例如场景、动画和 Prefab 都不需要修改，也不需要升级。
- 对 **程序** 与 **插件开发者** 而言，影响主要体现在原先项目中定制的构建模板和 `settings.json` 需要调整为新的方式。

以下将详细介绍这部分内容。

需要手动升级的情况

- 你在项目中有定制过构建模板中的 `application.ejs`。
- 你在项目中有定制过构建模板中的 `game.ejs` 或 `index.ejs`。
- 你在项目中有定制过生成后的 `settings.json`。

升级步骤

将定制过的 application.ejs 迁移到新的模板上

application.ejs 在 v3.6 版本中改动较大，主要有以下两个方面的改动：

提供了一个更加简单的 Application 类型的定义

```
class Application {
    init (cc: any): Promise<void> | void;
    start(): Promise<void>;
}
```

我们提供了形如以上的 Application 的定义，我们在其中提供了两个生命周期回调：

- `init` 生命周期函数会在引擎代码加载后被调用，引擎模块将作为参数传入，你可以在 `init` 函数中监听引擎启动流程的事件，并在对应事件触发时，执行一些定制逻辑。
- `start` 生命周期函数会在 `init` 函数后被调用，你需要在此函数中以你需要的方式启动并运行引擎。

下面是我们实现的一份最简单的 `application.ejs` 的模板与其对应的解释：

```
let cc;
export class Application {
    constructor () {
```

```
        this.settingsPath = '<%= settingsJsonPath %>'; // settings.json 文件路径，通常由编辑器构建时传入，你也可以指定自己的路径
        this.showFPS = <%= showFPS %>; // 是否打开 profiler，通常由编辑器构建时传入，你也可以指定你需要的值
    }

    init (engine) {
        cc = engine;
        cc.game.onPostBaseInitDelegate.add(this.onPostInitBase.bind(this)); // 监听引擎启动流程事件 onPostBaseInitDelegate
        cc.game.onPostSubsystemInitDelegate.add(this.onPostSystemInit.bind(this)); // 监听引擎启动流程事件 onPostSubsystemInitDelegate
    }

    onPostInitBase () {
        // cc.settings.overrideSettings('assets', 'server', '');
        // 实现一些自定义的逻辑
    }

    onPostSystemInit () {
        // 实现一些自定义的逻辑
    }

    start () {
        return cc.game.init({ // 以需要的参数运行引擎
            debugMode: <%= debugMode %> ? cc.DebugMode.INFO : cc.DebugMode.ERROR,
            settingsPath: this.settingsPath, // 传入 settings.json 路径
            overrideSettings: { // 对配置文件中的部分数据进行覆盖，第二部分会详细介绍这个字段
                // assets: {
                //      preloadBundles: [{ bundle: 'main', version: 'xxx' }],
                // }
                profiling: {
                    showFPS: this.showFPS,
                }
            }
        }).then(() => cc.game.run());
    }
}
```

- 此模板中在 `init` 函数中监听了引擎启动流程中的事件，可以在对应事件触发时执行一些自定义逻辑。 `game.init` 在执行过程中会触发各个阶段的事件，可参考 `game.init` 的 API 说明，其中不同的阶段的事件可以使用不同的引擎能力，触发的事件顺序如下所示：

```
-PreBaseInitEvent，不可以使用任何引擎能力
-基础模块初始化(logging, sys, settings)
-PostBaseInitEvent，可使用基础模块中的能力
-PreInfrastructureInitEvent，可使用基础模块中的能力
-基础设施模块的初始化(assetManager, builtinResMgr, gfxDevice, screen, Layer, macro)
-PostInfrastructureInitEvent，可使用基础模块，基础设施模块中的能力
-PreSubsystemInitEvent，可使用基础模块，基础设施模块中的能力
-子系统模块初始化(animation, physics, tween, ui, middleware 等)
-PostSubsystemInitEvent，可使用基础模块，基础设施模块，子系统模块中的能力
-EngineInitedEvent，可使用基础模块，基础设施模块，子系统模块中的能力
-PreProjectDataInitEvent，可使用基础模块，基础设施模块，子系统模块中的能力
-项目数据初始化(GamePlayScripts, resources, etc)
-PostProjectDataInitEvent，可使用基础模块，基础设施模块，子系统模块，项目数据中的能力
-GameInitedEvent，可使用引擎所有能力
```

- `start` 函数中调用了引擎的初始化与运行。你可以通过在调用 `game.init` 时传入自定义的参数来控制引擎的启动。需要注意的是其中的 `overrideSettings` 字段，此字段可用于覆盖配置文件中的一些配置数据，从而影响引擎的启动。第二部分将详细说到此字段。

新的 `application.ejs` 比之前版本更加简洁和易定制，你可以参考此模板实现你自己的 `application.ejs` 并在其中执行你的自定义逻辑。

引擎启动流程相关的逻辑代码迁移到了引擎中

我们将之前在 application.ejs 中与引擎启动相关的逻辑迁移到了引擎中，包括 settings.json 的加载(`loadSettingsJson`)，js 插件的加载（`loadJsList`），项目 bundle 的加载（`loadAssetBundle`）等等。所以现在你 **无法在 application.ejs 直接修改引擎启动流程**，但你可以通过 `game.init` 中传入的 `overrideSettings` 字段来覆盖配置文件，从而影响启动流程，另外你也可以监听引擎启动过程中的事件，并执行一些自定义逻辑。下面将详细介绍引擎的启动流程定制如何迁移到新的机制上。如果你的项目有对引擎启动流程进行定制，请参考以下升级方法进行迁移：

- 如果你在老的模板中引擎启动的特定阶段插入了自定义的代码逻辑，例如做了如下定制：

```
function start ({
    findCanvas,
}) {
    let settings;
    let cc;
    return Promise.resolve()
        .then(() => topLevelImport('cc'))
        .then(() => customLogic1()) // 自定义逻辑 1
        .then((engine) => {
            cc = engine;
            return loadSettingsJson(cc);
        })
        .then(() => customLogic2()) // 自定义逻辑 2
        .then(() => {
            settings = window._CCSettings;
            return initializeGame(cc, settings, findCanvas)
                .then(() => {
                    if (settings.scriptPackages) {
                        return loadModulePacks(settings.scriptPackages);
                    }
                })
                .then(() => loadJsList(settings.jsList))
                .then(() => loadAssetBundle(settings.hasResourcesBundle, settings.hasStartSceneBundle))
                .then(() => {
                    return cc.game.run(() => onGameStarted(cc, settings));
                });
        }).then(() => customLogic3()); // 自定义逻辑 3
}
```

  你可以在新的 application.ejs 模板中通过上述所说的监听引擎启动流程中的事件来实现自定义逻辑，例如：

```
init (engine) {
    cc = engine;
    cc.game.onPreBaseInitDelegate.add(this.onPreBaseInit.bind(this)); // 监听引擎启动流程事件 onPreBaseInitDelegate
    cc.game.onPostBaseInitDelegate.add(this.onPostBaseInit.bind(this)); // 监听引擎启动流程事件 onPostBaseInitDelegate
    cc.game.onPostProjectInitDelegate.add(this.onPostProjectInit.bind(this)); // 监听引擎启动流程事件 onPostProjectInitDelegate
}

onPreBaseInit () {
    customLogic1(); // 自定义逻辑 1
}

onPostBaseInit () {
    customLogic2() // 自定义逻辑 2
}

onPostProjectInit () {
    customLogic3() // 自定义逻辑 3
}
```

- 如果你在老的模板中定制了物理模块 wasm 的加载，即老模板中的下面这个方法。

```
<% if (hasPhysicsAmmo) { %>
promise = promise
    .then(() => topLevelImport('wait-for-ammo-instantiation'))
    .then(({default: waitForAmmoInstantiation}) => {
        return waitForAmmoInstantiation(fetchWasm(''));
    });
<% } %>
```

  此方法已改为引擎内部使用，如果需要定制，请自定义引擎，并定制引擎目录下的 cocos/physics/bullet/instantiated.ts 内容。

- 如果你在老的模板中定制了 settings.json 的加载, 即老模板中的 `loadSettingsJson` 函数。 此方法已改为引擎内部使用，如果你对这部分存在定制，请分为以下三种情况考虑：
  - 如果你需要定制 `settings.json` 的路径，你可以在调用 `game.init` 方式时传入自定义的 `settingsPath` 路径，例如：

    ```
    game.init({ settingsPath: this.mySettingsPath });
    ```

  - 如果你需要读取 settings.json 中的内容，你可以监听 `game.onPostBaseInitDelegate` 之后的事件，这些事件中你都可以安全的访问引擎中的 `settings` 模块，并通过 `settings` 模块的相关 API 获取对应配置数据，例如：

    ```
    init (engine) {
        cc = engine;
        cc.game.onPostBaseInitDelegate.add(this.onPostBaseInit.bind(this));
    }

    onPostBaseInit () {
        const property = cc.settings.querySettings('MyCustomData', 'MyCustomProperty');
    }
    ```

  - 如果你需要设置配置文件中的内容，你可以通过在调用 `game.init` 方法时传入 `overrideSettings` 字段来覆盖一些配置文件中的数据，也可以在 game 的事件回调中去调用 `settings.overrideSettings` 覆盖配置文件中的数据，从而影响引擎启动流程，例如：

    ```
    start () {
        cc.game.init({
            overrideSettings: {
                'profiling': { 'showFPS': true }
            }
        });
    }
    ```

    或

    ```
    onPostBaseInit () {
        cc.settings.overrideSettings('profiling', 'showFPS', true);
    }
    ```

- 如果你在老的模板中定制了加载 js 插件, 即老模板中的 `loadJsList` 函数。 此方法已改为引擎内部使用，如果你需要修改要加载的 js 文件列表，你可以通过在调用 `game.init` 时传入 `overrideSettings` 的 `plugin` 字段来修改，例如：

  ```
  start () {
      cc.game.init({
          overrideSettings: {
              'plugins': { 'jsList': ['MyCustom.js'] }
          }
      });
  }
  ```

- 如果你在老的模板中定制了加载项目 Bundle， 即老模板中的 `loadAssetBundle` 函数。 此方法已改为引擎内部使用，如果你需要修改要加载的 bundle 列表，你可以通过在调用 `game.init` 时传入 `overrideSettings` 的 `assets` 字段来修改，例如：

  ```
  start () {
      cc.game.init({
          overrideSettings: {
              'assets': { 'preloadBundles': [ 'main', 'resources', 'myBundle' ]}
          }
      });
  }
  ```

- 如果你在老的模板中定制了 macros 与 layer 的初始化， 即老模板中的`initializeGame` 函数。 此方法已改为引擎内部使用，如果你需要修改 macros 与 layer 的列表，你可以通过在调用 `game.init` 时传入 `overrideSettings` 的 `engine` 字段来修改，例如：

  ```
  start () {
      cc.game.init({
          overrideSettings: {
              'engine': {
                  'macros': {},
                  'customLayers': [],
              }
          }
      });
  }
  ```

- 如果你在老的模板中定制了初始场景的加载， 即老模板中的 `onGameStarted` 函数。 此方法已改为引擎内部使用，如果你需要修改初始场景，你可以通过在调用 `game.init` 时传入 `overrideSettings` 的 `launch` 字段来修改，例如：

  ```
  start () {
      cc.game.init({
          overrideSettings: {
              'launch': { 'launchScene': 'MyFirstScene' }
          }
      });
  }
  ```

- 如果你在老的模板中定制了传入 `game.init` 的参数， 即老模板中的 `getGameOptions` 函数。 请参考最新的 `IGameConfig` 接口定义，并在 `game.init` 时传入该参数。

将定制过的 game.ejs 或 index.ejs 迁移到新模板上

game.ejs 与 index.ejs 相较于之前版本没有太大变化，主要变化体现在以下几个点：

- 我们将 `loadJsListFile`, `fetchWasm`, `findCanvas` 等引擎使用的接口迁移到了引擎内部，如果你对这部分内容存在定制，请自定义引擎，并修改引擎目录下的 pal/env 的内容。
- 正如第一部分所说，我们整理了 Application 类型的接口定义，而 game.ejs 与 index.ejs 作为 Application 类型的调用者，所以在 Application 接口调用的地方也发生了变化，比如在某些平台我们改成了如下形式：

  ```
  System.import('<%= applicationJs %>')
  .then(({ Application }) => {
      return new Application();
  }).then((application) => {
      return System.import('cc').then((cc) => {
          return application.init(cc);
      }).then(() => {
          return application.start();
      });
  }).catch((err) => {
      console.error(err.toString() + ', stack: ' + err.stack);
  });
  ```

- 我们将不同平台上用于适配的代码进行了打包，合并成了单个 js 文件以减少包体，所以在运行代码的地方改为只运行一个 js 文件，例如：

  ```
  require('./libs/common/engine/index.js');
  require('./libs/wrapper/engine/index');
  require('./libs/common/cache-manager.js');
  ```

  改为了

  ```
  require('./engine-adapter');
  ```

如果你对 game.ejs 与 index.ejs 存在定制的话，只需要将定制迁移到最新的模板上即可，需要 **注意** 的是如果你的自定义逻辑依赖于 **字符串的匹配** 进行插入，例如：

```
const gameTemplateString = readFileSync('game.ejs', 'utf-8');
gameTemplateString = gameTemplateString.replace('require('./libs/common/cache-manager.js');', 'require('./libs/common/cache-manager.js');\nCustomLogic();\n')
```

则有可能出现匹配字符串失败的地方，你需要在新的模板下进行插入，或者你可以参考第一部分在 application.ejs 中进行定制，新的 application.ejs 模板更加利于定制。

迁移对 settings.json 的定制

我们在 3.6 中增加了 `settings` 模块用于在游戏中对配置数据进行访问，你可以在 settings.json 中增加自定义数据，然后在运行时使用 `settings` 模块进行访问，但为了避免模块之间的配置数据互相影响，我们将之前只有一层的配置数据结构，改为了 `Category` 与 `Property` 的两层结构，下面是具体的变化：

之前版本：

```
interface Settings {
    CocosEngine: string;
    debug: boolean; // 是否是 debug 模式
    designResolution: ISettingsDesignResolution; // 设计分辨率
    jsList: string[]; // js 插件列表
    launchScene: string; // 首场景
    preloadAssets: string[], // 预加载资源
    platform: string; // 平台名称
    renderPipeline: string; // 渲染管线
    physics?: IPhysicsConfig; // 物理相关配置
    exactFitScreen: boolean; // 在 web 下是否让游戏外框对齐到屏幕上

    bundleVers: Record<string, string>; // bundle 版本信息
    subpackages: string[]; // bundle 在小游戏上的子包配置
    remoteBundles: string[]; // 远程 bundle 的列表
    server: string; // 远程服务器地址
    hasResourcesBundle: boolean; // 是否存在 resources 包
    hasStartSceneBundle: boolean; // 是否存在首场景分包

    scriptPackages?: string[]; // 引擎内部使用字段
    splashScreen?: ISplashSetting; // SplashScreen 相关配置

    customJointTextureLayouts?: ICustomJointTextureLayout[]; // JointTextureLayout 相关的配置

    macros?: Record<string, any>; // cc.macros 相关配置
    engineModules: string[]; // 引擎模块，预览时使用
    customLayers: {name: string, bit: number}[]; // 自定义 layer 的相关配置
    orientation?: IOrientation; // 屏幕旋转方向
}
```

新版本的 Settings.json 文件格式：

```
interface Settings {
    CocosEngine: string;
    engine: {
        debug: boolean;
        macros: Record<string, any>;
        customLayers: {name: string, bit: number}[];
        platform: string;
        engineModules?: string[];
        builtinAssets: string[];
    };
    physics?: IPhysicsConfig;
    rendering: {
        renderPipeline: string;
        renderMode?: number;
    };
    assets: {
        server: string;
        remoteBundles: string[];
        bundleVers: Record<string, string>;
        preloadBundles: { bundle: string, version?: string }[];
        importBase?: string;
        nativeBase?: string;
        subpackages: string[];
        preloadAssets: string[];
        jsbDownloaderMaxTasks?: number;
        jsbDownloaderTimeout?: number;
    };
    plugins: {
        jsList: string[];
    };
    scripting: {
        scriptPackages?: string[];
    };
    launch: {
        launchScene: string;
    };
    screen: {
        frameRate?: number;
        exactFitScreen: boolean;
        orientation?: IOrientation;
        designResolution: ISettingsDesignResolution;
    };
    splashScreen?: ISplashSetting;
    animation: {
        customJointTextureLayouts?: ICustomJointTextureLayout[];
    };
    profiling?: {
        showFPS: boolean;
    };
}
```

字段与之前版本几乎保持了一一对应的关系，只是迁移到了某个具体的 `Category` 下。有几个字段略有变化：

- 新增 screen 下的 frameRate 字段用于设置目标更新频率
- 新增 profiling 下的 showFPS 字段用于控制是否开启左下角的统计数据显示
- 移除了原先的 hasResourcesBundle 与 hasStartSceneBundle 字段，改为统一由 assets 下的 preloadBundles 字段控制
- 新增 assets 下的 jsbDownloaderMaxTasks, jsbDownloaderTimeout 字段用于控制原生环境下资源的下载并发数与超时时间。

如果你对 settings.json 中的原有数据字段有所定制的话，你需要参考新的格式，将定制内容迁移到新的格式上。如果你有自定义的配置字段，我们建议你将自定义的配置数据放到一个自定的 `Category` 下，避免与其他模块的数据发生冲突，例如：

```
interface MyCustomSettings extends Settings {
    'customSettings': {
        'someProperty': string;
    };
}
```

以上就是 3.6 版本构建模板与 settings.json 的升级说明，如果在升级过程中遇到问题，欢迎到 Cocos 的官方社区，GitHub 中向我们反馈，感谢！

Cocos Creator 3.6 材质升级指南

升级指南：Effect 从 v3.5.x 升级到 v3.6.0

Chunks 迁移

3.6.0将之前版本 chunks 文件夹中的零散文件分门别类的存放到子文件夹中，书写 chunk 的 #include 时请参考下面的表格：

1、公共函数库

原文件	新路径

原文件	新路径
common	common/common-define
texture-lod	common/texture/texture-lod
packing	common/data/packing
unpack	common/data/unpack
aces	common/color/aces
gamma	common/color/gamma
octahedron-transform	common/math/octahedron-transform
transform	common/math/transform
rect-area-light	common/lighting/rect-area-light

2、Uniform 定义

原文件	新路径
cc-global	builtin/uniforms/cc-global
cc-local	builtin/uniforms/cc-local
cc-forward-light	builtin/uniforms/cc-forward-light
cc-environment	builtin/uniforms/cc-environment
cc-diffusemap	builtin/uniforms/cc-diffusemap
cc-shadow	builtin/uniforms/cc-shadow
cc-world-bound	builtin/uniforms/cc-world-bound

3、通用 Shader 主函数（仅限 legacy shader）

原文件	新路径
outline-vs	legacy/main-functions/outline-vs
outline-fs	legacy/main-functions/outline-fs
general-vs	legacy/main-functions/general-vs

4、引擎功能模块及其他（仅限 legacy shader）

原文件	新路径
cc-fog-base	legacy/fog-base
cc-shadow-map-base	legacy/shadow-map-base
morph	legacy/morph
cc-skinning	legacy/skinning
cc-local-batch	legacy/local-batch
lighting	legacy/lighting
lightingmap-fs	legacy/lightingmap-fs
cc-shadow-map-vs	legacy/shadow-map-vs
cc-shadow-map-fs	legacy/shadow-map-fs
cc-fog-vs	legacy/fog-vs
cc-fog-fs	legacy/fog-fs
lightingmap-vs	legacy/lightingmap-vs
decode	legacy/decode
decode-base	legacy/decode-base
decode-standard	legacy/decode-standard
input	legacy/input
input-standard	legacy/input-standard
output	legacy/output
output-standard	legacy/output-standard
shading-standard	legacy/shading-standard
shading-standard-base	legacy/shading-standard-base
shading-standard-additive	legacy/shading-standard-additive
shading-cluster-additive	legacy/shading-cluster-additive
shading-toon	legacy/shading-toon
standard-surface-entry	legacy/standard-surface-entry

5、仅供内部使用

原文件	新路径
alpha-test	builtin/internal/alpha-test
cc-sprite-common	builtin/internal/sprite-common
cc-sprite-texture	builtin/internal/sprite-texture
embedded-alpha	builtin/internal/embedded-alpha
particle-common	builtin/internal/particle-common

升级指南：粒子从 v3.5.x 升级到 v3.6.0

升级指南：粒子 从 v3.5.x 升级到 v3.6.0

CPU 粒子

v3.6.0 粒子系统新增了 instance 支持，粒子 shader particle-vs-legacy.chunk 需要做如下修改：

原来的 layout：

```
in vec3 a_position; // center position
in vec3 a_texCoord;  // xy:vertex index,z:frame index
in vec3 a_texCoord1; // size
in vec3 a_texCoord2; // rotation
in vec4 a_color;

#if CC_RENDER_MODE == RENDER_MODE_STRETCHED_BILLBOARD
  in vec3 a_color1; // velocity.x, velocity.y, velocity.z, scale
#endif

#if CC_RENDER_MODE == RENDER_MODE_MESH
  in vec3 a_texCoord3;  // mesh vertices
  in vec3 a_normal;     // mesh normal
  in vec4 a_color1;     // mesh color
#endif
```

现在的 layout：

```
in vec3 a_texCoord1;  // size
in vec3 a_texCoord2;  // rotation
in vec4 a_color;

in vec3 a_texCoord;  // xy:vertex index,z:frame index
#if !CC_INSTANCE_PARTICLE
 in vec3 a_position;  // center position
#endif
#if CC_INSTANCE_PARTICLE
 in vec4 a_texCoord4; // xyz:position,z:frame index
#endif

#if CC_RENDER_MODE == RENDER_MODE_STRETCHED_BILLBOARD
  in vec3 a_color1; // velocity.x, velocity.y, velocity.z, scale
#endif

#if CC_RENDER_MODE == RENDER_MODE_MESH
  in vec3 a_texCoord3;  // mesh vertices
  in vec3 a_normal;     // mesh normal
  in vec4 a_color1;     // mesh color
#endif
```

其它 shader 代码需要参考 v3.6.0 的 particle-vs-legacy.chunk 进行修改。

GPU 粒子

GPU 粒子 shader particle-vs-gpu.chunk 需要做如下修改：

原来的 layout：

```
in vec4 a_position_starttime; // center position,particle start time
in vec4 a_size_uv;  // xyz:size, w:uv_0
in vec4 a_rotation_uv;  // xyz:rotation, w:uv_1
in vec4 a_color;
in vec4 a_dir_life;  // xyz:particle start velocity,w:particle lifetime
in float a_rndSeed;

#if CC_RENDER_MODE == RENDER_MODE_MESH
  in vec3 a_texCoord;  // mesh uv
  in vec3 a_texCoord3;  // mesh vertices
  in vec3 a_normal;     // mesh normal
  in vec4 a_color1;     // mesh color
#endif
```

现在的 layout

```
in vec4 a_position_starttime; // center position,particle start time
in vec4 a_color;
in vec4 a_dir_life;  // xyz:particle start velocity,w:particle lifetime
in float a_rndSeed;

#if !CC_INSTANCE_PARTICLE
  in vec4 a_size_uv;  // xyz:size, w:uv_0
  in vec4 a_rotation_uv;  // xyz:rotation, w:uv_1
#endif
#if CC_INSTANCE_PARTICLE
  in vec4 a_size_fid;  // xyz:size, w:fid
  in vec3 a_rotation;  // xyz:rotation
  in vec3 a_uv;
#endif

#if CC_RENDER_MODE == RENDER_MODE_MESH
  in vec3 a_texCoord;   // mesh uv
  in vec3 a_texCoord3;  // mesh vertices
  in vec3 a_normal;     // mesh normal
  in vec4 a_color1;     // mesh color
#endif
```

其它 shader 代码需要参考 v3.6.0 的 particle-vs-gpu.chunk 进行修改。

场景制作

场景制作工作流程

场景是游戏中的环境因素的抽象集合，是创建游戏环境的局部单位，我们可以理解为游戏开发设计人员通过在编辑器中制作一个场景，来表现游戏中的一部分世界内容。

场景资源

场景资源

在 Cocos Creator 3.0 中，游戏场景（Scene）是游戏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。而场景文件本身也作为游戏资源存在，并保存了游戏的大部分信息，也是创作的基础。

> **注意**：请尽量避免多人同时修改同一个场景资源，否则可能会导致冲突，且无法通过 `git` 合并解决冲突。

创建场景

创建场景目前有以下几种方式：

1. 在 **资源管理器** 中右键点击想要放置场景文件的文件夹，然后选择 **创建 -> Scene** 即可。为了使项目具备良好的文件夹目录结构，强烈建议使用该方法创建场景。

节点和组件

节点和组件

Cocos Creator 3.0 的工作流程是以组件式开发为核心的，组件式架构也称作 **实体 — 组件架构**（Entity-Component System），简单来说，就是以组合而非继承的方式进行游戏中各种元素的构建。

在 Cocos Creator 3.0 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。

节点

节点是场景的基础组成单位。节点之间是树状的组织关系，每个节点可以有多个子节点：

坐标系和节点变换

坐标系和节点变换属性

在文档 场景编辑器 和 节点和组件 中，我们介绍了可以通过 **变换工具 Gizmo** 和编辑 **属性检查器** 中节点的属性来变更节点的显示行为。这篇文档我们将会深入了解节点所在场景空间的坐标系，以及节点的 **位置（Position）**、**旋转（Rotation）**、**缩放（Scale）** 三大变换属性的工作原理。

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 3.0 的坐标系，才能理解节点位置的意义。

世界坐标系（World Coordinate）

世界坐标系也叫做绝对坐标系，在 Cocos Creator 3.0 游戏开发中表示场景空间内的统一坐标体系，「世界」用来表示我们的游戏场景。

Creator 3.0 的世界坐标系采用的是笛卡尔右手坐标系，默认 x 向右，y 向上，z 向外，同时使用 -z 轴为正前方朝向。

节点层级和渲染顺序

节点层级和渲染顺序

通过前面的内容，我们了解了通过节点和组件的组合，能够在场景中创建多种元素。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，并按需排列它们的显示顺序。

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。

我们用一个简单的游戏场景来看一下什么是节点树。下图中包括背景图像、一个主角（小球）、标题、跳板、钻石和开始游戏的按钮：

使用场景编辑器搭建场景

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。

使用节点创建菜单快捷添加基本节点类型

当我们开始在场景中添加内容时，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，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，需要展开刚才选中的节点才能看到新添加的节点。

空节点

选择 **创建节点菜单** 中的 **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，也可以用来挂载开发者编写的逻辑和控制组件。另外在下文中我们也会介绍如何通过空节点和组件的组合，创造符合自己特殊要求的控件。

3D 对象

选择 **创建节点菜单** 中的 **创建 3D 对象** 可以创建编辑器自带的一些比较基础的静态模型控件，目前包括立方体、圆柱体、球体、胶囊、圆锥体、圆环体、平面和四方形。若需要创建其他类型的模型，可参考 MeshRenderer 组件。

UI 节点

选择 **创建节点菜单** 中的 **创建 UI** 可以创建 UI 节点。Creator 3.0 的 UI 节点需要其任意上级节点至少得有一个含有 UITransform 组件，在创建时若不符合规则，便会自动添加一个 Canvas 节点作为它的父级。并且每一个 UI 节点本身也会带有 UITransform 组件。

所以 Canvas 节点是 UI 渲染的 **渲染根节点**，所有渲染相关的 UI 节点都要放在 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 节点会根据屏幕大小自动居中显示，所以 Canvas 下的 UI 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 UI 节点位置的脚本更容易编写。

2D 渲染节点

通过 **创建节点菜单** 可以创建像 ParticleSystem（粒子）、Sprite（精灵）、Label（文字）、Mask（遮罩）等由节点和基础渲染组件组成的节点类型。

这里的基础 2D 渲染组件，是无法用其他组件的组合来代替的。需要注意的是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复杂的界面控件，比如下面的 UI 控件节点。

UI 控件节点

从 **创建节点菜单** 中的 **UI** 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，就包括了一个包含 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：

多层次细节

多层次细节

多层次细节（Level Of Details 以下简称 LOD）是大场景开发中常用的一种优化方式。LOD 的核心在于对于远处或者不重要的物体，降低其显示细节，以达到提升渲染效率的目的。

通常的 LOD 的做法是对于某些离屏幕较远或者不重要的物体，使用低模进行代替。

在引擎中，如果要启用 LOD，可以在 **属性检查器** 内选择 **添加组件** 按钮并选择 **LOD Group** 组件。

资源系统

关于资源

本章节将详细介绍 Cocos Creator 中资源的整体工作流程，并对各类资源的使用方法及可能需要注意的地方做出说明。

资源管理器

资源管理器 作为访问管理资源的重要工具，开发者在管理资源时推荐先熟悉资源管理器的使用方法，关于资源管理器的详细介绍可见：资源管理器

资源工作流

资源工作流 通用的资源工作流程包括导入资源、同步资源、定位资源等，详情请参考 资源工作流。

常见资源类型

接下来我们会介绍 Cocos Creator 中的主要资源类型：

- 场景资源
- 图像资源
  - 纹理贴图资源
  - 精灵帧资源
  - 立方体贴图资源
  - 图像资源的自动裁剪
  - 图集资源
  - 渲染纹理
- 预制资源
- 脚本资源
- 字体资源
- 音频资源
- 材质资源
- 模型资源
  - 从第三方工具导出模型资源
  - glTF 模型
- 动画资源
- Spine 骨骼动画资源
- DragonBones 骨骼动画资源
- TiledMap 瓦片图资源
- JSON 资源
- 文本资源

资源管理

运行时资源管理部分的内容请参考 Asset Manager。

资源工作流

资源工作流

导入资源

Creator 提供了三种 **导入资源** 的方式：

- 以新建文件的方式，通过 Cocos Creator 窗口的 **资源管理器** 面板 **创建按钮** 导入资源
- 以复制文件的方式，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，之后打开编辑器或激活编辑器窗口会自动刷新 **资源管理器** 的资源列表，完成导入资源。
- 以拖拽文件的方式，从操作系统的文件管理器中拖拽资源文件到 **资源管理器** 面板的某个文件夹位置，完成导入资源。

资源相关的名称如下：

名称	说明
dataBase	数据库
asset-db	项目资源数据库
internal-db	内置数据库
uuid	唯一标识符
meta	元信息

同步资源

资源管理器 面板中的资源和 **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，都会同步到 **操作系统的文件管理器**，反之亦然。

资源配置信息 .meta 文件

所有资源文件都会在导入时生成一份同名的 .meta 后缀的配置文件 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，是识别一份合法资源的必要因素。

在 **资源管理器** 面板中 .meta 文件是不可见的，对资源的重命名，移动，删除，都会由编辑器自动同步该资源对应的 .meta 文件，以确保配置信息如 **UUID** 等保持不变，即不影响现有的引用。

不推荐直接在 **操作系统的文件管理器** 对资源文件进行操作，如有操作，请同步处理相应的 .meta 文件，如下建议：

- 关闭正在使用的编辑器，避免因为文件锁定或资源名称相同导致更新失败。
- 删除，重命名，移动资源时，请连同 .meta 文件一起删除，重命名，移动。
- 复制资源时如果连同 .meta 文件一起复制，将直接使用复制进来的 .meta 文件，而不是再生成新的 .meta 文件；如果只复制了资源文件，则会生成对应名称的新的 .meta 文件。

Library 中的资源

资源经过导入后会生成一份新的数据存在项目的 **Library** 文件夹里。在 **Library** 里的文件，其结构和资源是面向引擎的，是最终游戏时需要的格式，即机器友好，但对人的阅读不友好。这块。

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。

如何定位资源

一个资源有唯一的 uuid 来定位到该资源，但这种方式不够直观，还有另一种直观的方式：**Database URL** 格式例如 asset-db 对应的协议头是 db://assets，internal-db 对应的协议头是 db://internal。

有文件夹层级的资源格式，如 db://assets/prefabs/fire.prefab

SVN 或 GIT 同步资源

需要注意 .meta 文件的换行符，建议统一下团队成员电脑的换行符风格和规则，避免同步项目资源后打开项目，出现了大量的 .meta 文件修改

图像资源

图像资源

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，目前支持 **JPG**、**PNG**、**BMP**、**TGA**、**HDR**、**WEBBP**、**PSD**、**TIFF** 等格式。

导入图像资源

将图像资源直接拖拽到 **资源管理器** 即可将其导入到项目中，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：

纹理贴图资源

纹理贴图资源（Texture）

纹理贴图资源是一种用于程序采样的资源，如模型上的贴图、精灵上的 UI。当程序渲染 UI 或者模型时，会使用纹理坐标获取纹理颜色，然后填充在模型网格上，再加上光照等等一系列处理便渲染出了整个场景。

纹理贴图资源可由图像资源（ImageAsset）转换而来，图像资源包括一些通用的图像转换格式如 PNG、JPEG 等等。

Texture2D

Texture2D 是纹理贴图资源的一种，通常用于 3D 模型的渲染，如模型材质中的反射贴图、环境光遮罩贴图等等。

在将图像资源 导入 到 Creator 后，即可在 **属性检查器** 面板将其设置为 **texture** 类型，texture 类型便是 Texture2D 纹理资源。

精灵帧资源

精灵帧资源（SpriteFrame）

Cocos Creator 的 SpriteFrame 是 UI 渲染基础图形的容器。其本身管理图像的裁剪和九宫格信息，默认持有一个与其同级的 Texture2D 资源引用。

导入精灵帧资源

使用默认的 资源导入 方式将图像资源导入到项目中，然后在 **属性检查器** 中将图像资源的类型设置为 **sprite-frame**，并点击右上角的绿色打钩按钮保存：

图像资源的自动剪裁

图像资源的自动剪裁

导入图像资源后生成的 SpriteFrame 默认会进行自动剪裁，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。当 SpriteFrame 为自动剪裁时，下图中自动剪裁的相关信息为置灰状态，不可修改：

立方体贴图资源

立方体贴图

TextureCube 为立方体纹理，常用于设置场景的 天空盒。立方体贴图可以通过设置全景图 ImageAsset 为 TextureCube 类型获得，也可以在 Creator 中制作生成。

设置为立方体贴图

在将 ImageAsset 导入 到 Creator 后，即可在 **属性检查器** 面板将其设置为 **texture cube** 类型，设置完成后请点击右上角的绿色打钩按钮，以保存修改。

图集资源

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：

自动图集资源

自动图集资源 (Auto Atlas)

自动图集 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** 中点击左上角的 + 创建按钮，然后选择 **自动图集配置**，即可在 **资源管理器** 中新建一个 **auto-atlas.pac** 资源。

艺术数字资源

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** 中点击右键，然后选择 **创建 -> 艺术数字配置**，或者点击 **资源管理器** 左上角的加号按钮：

预制资源

预制件（Prefab）

预制件用于存储一些可以复用的场景对象，它可以包含节点、组件以及组件上的数据。由预制件生成的实例既可以继承模板的数据，又可以有自己定制化的数据修改。

基础概念

名称	说明	示例
预制件资源	在 **资源管理器** 中的预制件资源，是预制件的序列化文件。	

字体资源

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：

- Glyph Designer
- Hiero
- BMFont (Windows)

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。

请注意，在导入位图字体之后，需要将 png 文件的类型更改为 sprite-frame，否则位图字体将无法正常使用。

导入后的字体在 **资源管理器** 中显示如下：

音频资源

音频资源（AudioClip）

Cocos Creator 支持导入大多数常见的音频文件格式，将其直接拖拽到 **资源管理器** 面板即可，导入后会在 **资源管理器** 中生成相应的音频资源（AudioClip）。

材质资源

材质资源

材质创建

在 **资源管理器** 面板中点击右键并选择 **创建 -> 材质**：

FBX 智能材质导入

FBX 智能材质导入

FBX 智能材质导入是模型导入器中辅助转换材质的一个功能，它可以将各种 DCC（Digital Content Creation） 工具导出到模型中的部分标准材质直接映射到 Cocos Creator 的内置材质中，尽量还原美术在 DCC 工具中看到的材质效果。该功能为 v3.5.1 新增。

该功能当前支持以下材质：

Software	Phong	PBR
3ds Max	Standard(legacy)	Physical Material
Blender	N/A	Principled BSDF
C4D	Standard	N/A
Maya	Lambert/Blinn/Phong	Standard Surface

以 Maya Standard Surface 为例，材质导入 Cocos Creator 效果对比下表所示：

Maya Viewport **Cocos Creator Viewport**

模型资源

模型资源

目前，Creator 支持 **FBX** 和 **glTF** 两种格式的模型文件。

- FBX（.fbx）：支持 FBX 2020 及更早的文件格式。
- glTF（.gltf、.glb）：支持 glTF 2.0 及更早的文件格式，详情可参考 glTF 模型。

关于如何从第三方工具导出这两种模型文件，请参考 导入从 DCC 工具导出的模型。

模型导入

从外部导入编辑器中后，在 **资源管理器** 中可得到对应的模型资源文件，其目录结构如下：（以 glTF 文件为例，fbx 文件相同）

- 无动画的模型文件结构如下：

从第三方工具导出模型资源

导入从 DCC 工具导出的模型

目前大多数数字内容制作（Digital Content Creation, DCC）工具（3ds Max、Maya、Blender）都能导出 **FBX** 和 **glTF** 这两种格式的模型文件，所以这些工具导出的内容都能在 Cocos Creator 中得到良好的展示。

导出 FBX

因为 DCC 工具的坐标系和游戏引擎的坐标系可能不一致，所以在导出模型时需要进行一些变换才能在引擎中得到想要的结果。例如：Blender 的坐标系为 X 轴朝右，Y 轴朝里，Z 轴朝上，而 Cocos Creator 3.x 的坐标系为 X 轴朝右，Y 轴朝上，Z 轴朝外，所以需要调整旋转才能使得轴向一致。

以下以 Blender 2.8 作为例子，介绍模型的导入流程，首先我们在 Blender 中创建一个模型。

从 3ds Max 中导出 FBX 模型资源

从 3ds Max 中导出 FBX 模型资源

导出步骤

1. 在 3ds Max 选中要导出的模型

从 Maya 中导出 FBX 模型资源

从 Maya 中导出 FBX 模型资源

导出步骤

1. 选中要导出的模型：

---

glTF 模型

glTF 模型

Cocos Creator 支持 glTF 2.0 及更早的文件格式。

URI 解析

Creator 支持 glTF 中指定以下形式的 URI：

- Data URI
- 相对 URI 路径
- 文件 URL
- 文件路径

转换关系

当导入 glTF 模型到 Creator 时，glTF 中的资源将会按照以下关系转换为 Creator 中的资源：

glTF 资源	Cocos Creator 资源
glTF 场景	预制体
glTF 网格	网格
glTF 蒙皮	骨骼
glTF 材质	材质
glTF 贴图	贴图
glTF 图像	图像
glTF 动画	动画剪辑

glTF 场景

导入后，glTF 场景将转换为 Creator 中的预制体资源，glTF 场景中递归包含的节点也将按照相同层级关系一一转换为预制体中的节点。

场景根节点

预制体将使用一个不带任何空间转换信息的节点作为根节点，glTF 场景的所有 根节点 将作为该节点的子节点。

节点转换

glTF 节点中的属性将按照下表中的映射关系转换为预制体节点中的属性：

glTF 节点属性	预制体节点属性
层级关系	层级关系
位移	位置
旋转	旋转
缩放	缩放
矩阵	解压，并分别设置位置、旋转、缩放
网格引用	网格渲染器组件
蒙皮引用	蒙皮网格渲染器组件
初始权重	（蒙皮）网格渲染器组件权重

网格渲染器

若 glTF 节点引用了网格，那么导入后相对应的预制体节点也会添加网格渲染组件（MeshRenderer）。若该 glTF 节点还引用了蒙皮，那么相对应的预制体节点还会添加蒙皮网格渲染组件（SkinnedMeshRenderer）。

（蒙皮）网格渲染组件中的网格、骨骼和材质，都会与转换后的 glTF 网格、蒙皮、材质资源一一对应。

若 glTF 节点指定了初始权重，则转换后的（蒙皮）网格渲染器也将带有此权重。

glTF 网格

导入后，glTF 网格将转换为 Cocos Creator 中的网格资源。

glTF 网格中的所有 基元体 将被一一转换为 Creator 中的子网格。

若 glTF 网格指定了 权重，则相应地，转换后的 Creator 网格中也将存储相应的权重。

glTF 基元体

glTF 基元体的索引数组将一一对应转换为 Cocos Creator 子网格的索引数组。

glTF 基元模式将按照下表中的映射关系转换为 Cocos Creator 基元模式：

glTF 基元模式	Cocos Creator 基元模式
POINTS	gfx.PrimitiveMode.POINT_LIST
LINES	gfx.PrimitiveMode.LINE_LIST
LINE_LOOP	gfx.PrimitiveMode.LINE_LOOP
LINE_STRIP	gfx.PrimitiveMode.LINE_STRIP
TRIANGLES	gfx.PrimitiveMode.TRIANGLE_LIST

glTF 基元模式	Cocos Creator 基元模式
TRIANGLE_STRIP	gfx.PrimitiveMode.TRIANGLE_STRIP
TRIANGLE_FAN	gfx.PrimitiveMode.TRIANGLE_FAN

glTF 顶点属性将转换为 Cocos Creator 顶点属性，属性名称的转换如下表所示：

glTF 顶点属性名称	Cocos Creator 顶点属性名称
POSITION	gfx.AttributeName.ATTR_POSITION
NORMAL	gfx.AttributeName.ATTR_NORMAL
TANGENT	gfx.AttributeName.ATTR_TANGENT
TEXCOORD_0	gfx.AttributeName.ATTR_TEX_COORD
TEXCOORD_1..TEXCOORD_8	gfx.AttributeName.ATTR_TEX_COORD1..gfx.AttributeName.ATTR_TEX_COORD8
COLOR_0	gfx.AttributeName.ATTR_COLOR
COLOR_1..COLOR_2	gfx.AttributeName.ATTR_COLOR1..gfx.AttributeName.ATTR_COLOR2
JOINTS_0	gfx.AttributeName.ATTR_JOINTS
WEIGHTS_0	gfx.AttributeName.ATTR_WEIGHTS

注意：若 glTF 基元体中存在其他 `JOINTS`、`WEIGHTS` 顶点属性，例如 `JOINTS_1`、`WEIGHTS_1`，则意味着此 glTF 网格的顶点可能受到多于 4 根骨骼的影响。

对于每个顶点，所有由 `JOINTS_{}`、`WEIGHTS_{}` 确定的权重信息将按权重值进行排序，取出影响权重最大的四根骨骼作为 `gfx.AttributeName.ATTR_JOINTS` 和 `gfx.AttributeName.ATTR_WEIGHTS`。

glTF 形变目标将被转换为 Cocos Creator 子网格形变数据。

glTF 蒙皮

导入后，glTF 蒙皮将转换为 Cocos Creator 中的骨骼资源。

glTF 材质

导入后，glTF 材质将转换为 Cocos Creator 中的材质资源。

glTF 贴图

导入后，glTF 贴图将转换为 Cocos Creator 中的贴图资源。

glTF 贴图中引用的 glTF 图像将转换为对相应转换后的 Cocos Creator 图像的引用。

glTF 贴图属性将按照下表中的映射关系转换为 Cocos Creator 贴图属性：

glTF 贴图属性	Cocos Creator 贴图属性
放大筛选器	放大筛选器
缩小筛选器	缩小筛选器、Mip Map 筛选器
S 环绕模式	S 环绕模式
T 环绕模式	环绕模式

glTF 贴图放大筛选器将按照下表中的映射关系转换为 Cocos Creator 贴图放大筛选器：

glTF 贴图放大筛选器	Cocos Creator 贴图放大筛选器
NEAREST	TextureBase.Filter.NEAREST
LINEAR	TextureBase.Filter.LINEAR

glTF 贴图缩小筛选器将按照下表中的映射关系转换为 Cocos Creator 贴图缩小筛选器和 Cocos Creator 贴图 Mip Map 筛选器：

glTF 贴图缩小筛选器	Cocos Creator 贴图缩小筛选器	Cocos Creator 贴图 Mip Map 筛选器
NEAREST	TextureBase.Filter.NEAREST	TextureBase.Filter.NONE
LINEAR_MIPMAP_LINEAR	TextureBase.Filter.LINEAR	TextureBase.Filter.NONE
LINEAR_MIPMAP_NEAREST	TextureBase.Filter.NEAREST	TextureBase.Filter.NEAREST
LINEAR	TextureBase.Filter.LINEAR	TextureBase.Filter.NEAREST
NEAREST_MIPMAP_LINEAR	TextureBase.Filter.NEAREST	TextureBase.Filter.LINEAR
NEAREST_MIPMAP_NEAREST	TextureBase.Filter.LINEAR	TextureBase.Filter.LINEAR

glTF 贴图环绕模式将按照下表中的映射关系转换为 Cocos Creator 贴图环绕模式：

glTF 贴图环绕模式	Cocos Creator 贴图环绕模式
CLAMP_TO_EDGE	TextureBase.WrapMode.CLAMP_TO_EDGE
REPEAT	TextureBase.WrapMode.REPEAT
MIRRORED_REPEAT	TextureBase.WrapMode.MIRRORED_REPEAT

glTF 图像

导入后，glTF 图像将转换为 Cocos Creator 中的图像资源。

当 glTF 图像的 URI 是 Data URI 时，图像数据将从 Data URI 中获取。否则，将根据 Cocos Creator 图像位置解析算法 解析并引用外部图像文件，其中 `url` 就是 glTF 图像的 URI，`startDir` 为 glTF 文件所在目录。

glTF 动画

导入后，glTF 动画将转换为 Cocos Creator 动画资源。

程序化创建网格

程序化创建网格

当由 DCC（Digital Content Creation）软件制作或引擎内的地形编辑器制作的模型无法满足需求时，可以通过 API 来创建网格。如需要在运行时创建某种可以生长的蛇、动态编辑模型或实现某些曲面，都可以通过程序化来创建网格。

创建网格

引擎支持两种网格：**静态网格** 和 **动态网格**，适用于不同的场景，开发者可按需使用。

- 静态网格，通过 `utils.MeshUtils.createMesh` 创建，一旦创建成功，网格内的几何体不可编辑的。
- 动态网格：通过 `utils.MeshUtils.createDynamicMesh` 创建，创建成功后，网格内的几何体仍然可以修改。

返回值为 `Mesh` 组件，因此我们方便的将其赋值给 `MeshRenderer` 的 `mesh` 属性，如此即可将其显示在屏幕上。

API

API 请参考 MeshUtils。

范例

Spine 骨骼动画资源

骨骼动画资源（Spine）

Creator 中的骨骼动画资源是由 Spine 编辑器 导出的，目前支持 JSON 和 二进制 两种数据格式。

各 Creator 版本对应支持的 Spine 版本如下所示：

Creator 版本	Spine 版本
v3.0 及以上	v3.8（原生平台不支持特定版本 v3.8.75）
v2.3 及以上	v3.8
v2.2	v3.7
v2.0.8～v2.1	v3.6
v2.0.7 及以下	v2.5

导入骨骼动画资源

骨骼动画所需资源有：

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

DragonBones 骨骼动画资源

骨骼动画资源（DragonBones）

DragonBones 骨骼动画资源是由 DragonBones 编辑器 导出的数据格式（支持 DragonBones v5.6.3 及以下）。

导入 DragonBones 骨骼动画资源

DragonBones 骨骼动画资源包括：

- `.json/.dbbin` 骨骼数据
- `.json` 图集数据
- `.png` 图集纹理

TiledMap 瓦片图资源

瓦片图资源（TiledMap）

瓦片图资源是由 Tiled 编辑器 所导出的数据格式。

Creator 版本	Tiled 版本
v3.0 及以上	v1.4
v2.2 及以上	v1.2.0
v2.1 及以下	v1.0.0

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）

JSON 资源

JSON 资源

Creator 支持使用 Json 文件，通过 资源导入 的方式将其导入到编辑器，所有的 JSON 文件都会导入为 `cc.JsonAsset` 格式的资源。

使用方式

开发者可通过 **编辑器挂载** 和 **代码中动态加载** 两种方式获取 Json 数据。

通过编辑器

首先在 **资源管理器** 中新建一个 TypeScript，脚本内容示例如下：

```
import { _decorator, Component, JsonAsset } from 'cc';
const { ccclass, property } = _decorator;

@ccclass('ItemTemplate')
export class ItemTemplate extends Component {

    // 声明属性 `itemGiftJson` 的类型为 JsonAsset
    @property(JsonAsset)
    itemGiftJson: JsonAsset = null!;

    start () {

        // 获取到 Json 数据
        const jsonData: object = this.itemGiftJson.json!;

    }
}
```

保存脚本内容后回到编辑器，将脚本挂载到相应的节点上，然后将 **资源管理器** 中的 Json 资源拖拽到脚本组件相应的属性框中。例如下图：

文本资源

文本资源

Creator 支持使用文本文件，常见的文本格式如：`.txt`、`.plist`、`.xml`、`.json`、`.yaml`、`.ini`、`.csv`、`.md`。通过 资源导入 的方式将其导入到编辑器，所有的文本文件都会导入为 `cc.TextAsset` 格式的资源。

使用方式

开发者可通过 **编辑器挂载** 和 **代码中动态加载** 两种方式获取文本数据。

通过编辑器

首先在 **资源管理器** 中新建一个 TypeScript，脚本内容示例如下：

```
import { _decorator, Component, TextAsset } from 'cc';
const { ccclass, property } = _decorator;

@ccclass('ItemTemplate')
export class ItemTemplate extends Component {

    // 声明属性 `itemGiftText` 的类型为 TextAsset
    @property(TextAsset)
    itemGiftText: TextAsset = null!;

    start () {

        const data: string = this.itemGiftText.text!;

    }
}
```

保存脚本内容后回到编辑器，将脚本挂载到相应的节点上，然后将 **资源管理器** 中的文本资源拖拽到脚本组件相应的属性框中。例如下图：

脚本指南及事件机制

脚本指南及事件机制

Cocos Creator 脚本用于实现用户定义的（游戏）行为，支持 JavaScript 和 TypeScript 两种编程语言。通过编写脚本组件，并将它挂载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，开发者可以通过声明属性，将脚本中需要调节的变量映射到 **属性检查器** 中，以便策划和美术进行调整。与此同时，也可以通过注册特定的回调函数，来帮助初始化、更新甚至销毁节点。

内容

更多参考

编程语言支持

语言支持

Cocos Creator 支持 **TypeScript** 和 **JavaScript** 两种编程语言。但需要注意的是，**JavaScript** 只支持以 插件脚本 的形式导入使用。

TypeScript

Cocos Creator 支持 TypeScript 4.1.0。在此基础上，做了以下限制：

- `tsconfig.json` 不会被读取。每个项目都隐含着如下选项：

  ```
  {
    "compilerOptions": {
        "target": "ES2015",
        "module": "ES2015",
        "isolatedModules": true,
        "experimentalDecorators": true,
        "moduleResolution": /* Cocos Creator 特定的模块解析算法 */,
        "forceConsistentCasingInFileNames": true,
    }
  }
  ```

  隐含的 `isolatedModules` 选项意味着：

  - 不支持 const enums。
  - 重导出 TypeScript 类型和接口时应该使用 `export type`。例如使用 `export type { Foo } from './foo';` 而不是 `export { Foo } from './foo';`。

- 不支持 `export =` 和 `import =`。
- 命名空间导出的变量必须声明为 `const`，而不是 `var` 或 `let`。
- 同一命名空间的不同声明不会共享作用域，需要显式使用限定符。
- 编译过程中的类型错误将被忽略。

编译时不会读取 `tsconfig.json`，意味着 `tsconfig.json` 的编译选项并不会影响编译。

开发者仍然可以在项目中使用 `tsconfig.json` 以配合 IDE 实现类型检查等功能。为了让 IDE 的 TypeScript 检查功能和 Creator 行为兼容，开发者需要额外注意一些事项，详情可参考 tsconfig。

TypeScript 参考教程

- Cocos Creator 3.0 TypeScript 问题答疑及经验分享
- TypeScript 官方网站
- TypeScript - Classes
- TypeScript - Decorators
- TypeScript - DefinitelyTyped
- X 分钟速成 TypeScript
- TypeScript 源码
- 开发者回避使用 TypeScript 的三个借口 — 以及应当使用 TypeScript 的更有说服力的原因

JavaScript

语言特性

Creator 支持的 JavaScript 语言规范为 ES6。

此外，以下几项更新于 ES6 规范的语言特性或提案仍旧在支持之列：

- 类字段
- Promise 对象
- 可选链操作符 `?.`
- 空值合并操作符 `??`
- 逻辑赋值操作符
  - 逻辑空赋值操作符 `??=`
  - 逻辑与赋值操作符 `&&=`
  - 逻辑或赋值操作符 `||=`
- 全局对象 `globalThis`

以下语言特性同样支持，但需要开启相关的编译选项：

- 异步函数

特别地，Creator 目前支持 **Legacy** 装饰器提案，其具体用法和含义请参考 babel-plugin-proposal-decorators。由于该 提案 仍处于阶段 2，引擎暴露的所有装饰器相关功能接口都在以下划线开头的 `_decorator` 命名空间下。

编译选项

Creator 开放了部分编译选项，这些选项将应用到整个项目。

选项	名称	含义
useDefineForClassFields	符合规范的类字段	当开启时，将使用 `Define` 语义实现类字段，否则将使用 `Set` 语义实现类字段。仅当目标不支持 ES6 类字段时生效。
allowDeclareFields	允许声明类字段	当开启时，在 TypeScript 脚本中将允许使用 `declare` 关键字来声明类字段，并且，当字段未以 `declare` 声明且未指定显式的初始化式时，将依照规范初始化为 `undefined`。

运行环境

从用户的角度来说，Creator 未绑定任何 JavaScript 实现，因此建议开发者严格依照 JavaScript 规范编写脚本，以获取更好的跨平台支持。

举例来说，当希望使用 **全局对象** 时，应当使用标准特性 `globalThis`：

```
globalThis.blahBlah // 任何环境下 globalThis 一定存在
```

而非 `window`、`global`、`self` 或 `this`：

```
typeof window // 可能是 'undefined'
typeof global // 在浏览器环境下可能是 'undefined'
```

再如，Creator 未提供 **CommonJS** 的模块系统，因此以下代码片段会带来问题：

```
const blah = require('./blah-blah'); // 错误，require 是未定义的
module.exports = blah; // 错误 module 是未定义的
```

反之，应使用标准模块语法：

```
import blah from './blah-blah';
export default blah;
```

JavaScript 参考教程

- JavaScript 标准参考教程
- JavaScript 秘密花园
- JavaScript 内存详解 & 分析指南

脚本基础

脚本基础

该部分内容主要介绍脚本的一些基础概念、创建方式、运行环境等：

- 创建脚本
- 配置代码编辑环境
- 脚本运行环境
- 装饰器使用
- 属性参数参考
- 生命周期回调

创建脚本

创建脚本

创建组件脚本

在 Cocos Creator 中，脚本也是资源的一部分。在 **资源管理器** 中创建的脚本，默认是一个 NewComponent 组件，我们称之为组件脚本。可通过以下两种方式创建：

- **资源管理器** 面板空白位置或某个文件夹资源下右击菜单，选择 **Create** > **TypeScript** > **NewComponent**。
- **资源管理器** 左上角的 + 按钮，点击后选择 **TypeScript** > **NewComponent**。

配置代码编辑环境

配置代码编辑环境

在 **偏好设置** 面板中指定了 默认脚本编辑器，便可以在 **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。本篇内容主要以 Visual Studio Code 为例介绍如何配置开发环境。

Visual Studio Code（以下简称 VS Code）是微软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。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、网页调试等功能。

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。

MacOS 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 **VSCodeUserSetup.exe** 按提示完成安装即可运行。

智能提示数据

Cocos Creator 3.x 在创建项目时，项目目录下会自动生成一个 tsconfig.json 文件，里面配置了一个代码提示用的目录文件路径，用 VS Code 打开项目编写代码时便会自动提示 Cocos Creator 引擎 API。若项目升级，引擎 API 也会自动更新。

启动 VS Code 后选择主菜单的 **File -> Open Folder...**，在弹出的对话框中选择项目根目录，也就是 `assets`、`project.json` 所在的路径。然后新建一个脚本，或者打开原有的脚本进行编辑时，就会有语法提示了。

脚本运行环境

运行环境

Cocos Creator 3.0 引擎的 API 都存在模块 `cc` 中，使用标准的 ES6 模块导入语法将其导入：

```
import {
    Component,  // 导入类 Component
    _decorator, // 导入命名空间 _decorator
    Vec3 // 导入类 Vec3
} from 'cc';
import * as modules from 'cc'; // 将整个 Cocos Creator 模块导入为命名空间 Cocos Creator

@_decorator.ccclass("MyComponent")
export class MyComponent extends Component {
    public v = new Vec3();
}
```

保留标识符 cc

注意，由于历史原因，`cc` 是 Cocos Creator 3.0 保留使用的标识符，其行为 **相当于** 在任何模块顶部定义了名为 cc 的对象。因此，开发者不应该将 `cc` 用作任何 **全局对象** 的名称：

```
/* const cc = {}; // 每个 Cocos Creator 脚本都等价于在此处含有隐式定义 */

import * as modules from 'cc'; // 错误：命名空间导入名称 cc 由 Cocos Creator 保留使用

const cc = {
    x: 0
};
console.log(cc.x); // 错误：全局对象名称 cc 由 Cocos Creator 保留使用

function f () {
    const cc = {
        x: 0
    };
    console.log(cc.x); // 正确：cc 可以用作局部对象的名称

    const o = {
        cc: 0
    };
    console.log(o.cc); // 正确：cc 可以用作属性名
}

console.log(cc, typeof cc); // 错误：行为是未定义的
```

装饰器使用

装饰器使用

cc 类

将装饰器 `ccclass` 应用在类上时，此类称为 cc 类。cc 类注入了额外的信息以控制 Cocos Creator 对该类对象的序列化、编辑器对该类对象的展示等。因此，未声明 `ccclass` 的组件类，也无法作为组件添加到节点上。

`ccclass` 装饰器的参数 `name` 指定了 cc 类的名称，cc 类名是 **独一无二** 的，这意味着即便在不同目录下的同名类也是不允许的。当需要获取相应的 cc 类时，可以通过其 cc 类名来查找，例如：

- 序列化。若对象是 cc 类对象，则在序列化时将记录该对象的 cc 类名，反序列化时将根据此名称找到相应的 cc 类进行序列化。
- 当 cc 类是组件类时，`Node` 可以通过组件类的 cc 类名查找该组件。

  注意：类名不应该以 `cc.`、`internal.` 作为前缀，这是 Cocos Creator 的保留类名前缀。

```
@ccclass('Example')
export class Example extends Component {
}
```

组件类装饰器

此类装饰器是只能用来修饰 `Component` 的子类。

executeInEditMode

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。`executeInEditMode` 允许当前组件在编辑器模式下运行，默认值为 `false`。

```
const { ccclass, executeInEditMode } = _decorator;

@ccclass('Example')
@executeInEditMode(true)
export class Example extends Component {
    update (dt: number) {
        // 会在编辑器下每帧执行
    }
}
```

requireComponent

`requireComponent` 参数用来指定当前组件的依赖组件，默认值为 `null`。当组件添加到节点上时，如果依赖的组件不存在，引擎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。

```
const { ccclass, requireComponent } = _decorator;

@ccclass('Example')
@requireComponent(Sprite)
export class Example extends Component {
}
```

executionOrder

`executionOrder` 用来指定脚本生命周期回调的执行优先级。小于 0 的脚本将优先执行，大于 0 的脚本将最后执行。排序方式如下：

- 对于同一节点上的不同组件，数值小的先执行，数值相同的按组件添加先后顺序执行
- 对于不同节点上的同一组件，按节点树排列决定执行的先后顺序

该优先级设定只对 `onLoad`、`onEnable`、`start`、`update` 和 `lateUpdate` 有效，对 `onDisable` 和 `onDestroy` 无效。

```
const { ccclass, executionOrder } = _decorator;

@ccclass('Example')
@executionOrder(3)
export class Example extends Component {
}
```

disallowMultiple

同一节点上只允许添加一个同类型（含子类）的组件，防止逻辑发生冲突，默认值为 false。

```
const { ccclass, disallowMultiple } = _decorator;

@ccclass('Example')
@disallowMultiple(true)
export class Example extends Component {
}
```

menu

`@menu(path)` 用来将当前组件添加到组件菜单中，方便用户查找。

需要注意该菜单是添加在 **属性检查器** 面板中按下添加组件按钮后的下拉框内。

```
const { ccclass, menu } = _decorator;

@ccclass('Example')
@menu('foo/bar')
export class Example extends Component {
}
```

注意：需要在编辑器 **属性检查器** 中展示的属性，属性名开头不应该带 _，否则会识别为 private 属性，private 属性不会在编辑器组件属性面板上显示。

下列代码演示了不同 cc 类型的属性声明：

```
import { _decorator, CCInteger, Node, Enum } from 'cc';
const { ccclass, property, integer, float, type } = _decorator;

enum A {
    c,
    d
}
Enum(A);

@ccclass
class MyClass {
    @property // JavaScript 原始类型，根据默认值自动识别为 Creator 的浮点数类型。
    index = 0;

    @property(Node) // 声明属性 cc 类型为 Node。当属性参数只有 type 时可这么写，等价于 @property({type: Node})
    targetNode: Node | null = null; // 等价于 targetNode: Node = null!;

    // 声明属性 children 的 cc 类型为 Node 数组
    @property({
        type: [Node]
    })
    children: Node[] = [];

    @property({
        type: String,
    }) // 警告：不应该使用构造函数 String。等价于 CCString。也可以选择不声明类型
    text = '';

    @property
    children2 = []; // 未声明 cc 类型，从初始化式的求值结果推断元素为未定义的数组

    @property
    _valueB = 'abc'; // 此处 '_' 开头的属性，只序列化，不会在编辑器属性面板显示

    @property({ type: A })
    accx : A = A.c;
}
```

为了方便，额外提供几种装饰器以快速声明 cc 类型。如果你只需要为属性定义 type 参数，那么可以直接使用下列装饰器替代 `@property`：

装饰器	对应的 property 写法
@type(t)	@property(t)
@integer	@property(CCInteger)
@float	@property(CCFloat)

```
import { _decorator, CCInteger, Node } from 'cc';
const { ccclass, property, integer, float, type } = _decorator;
@ccclass
class MyClass {
    @integer // 声明属性的 cc 类型为整数
    index = 0;

    @type([Node]) // 声明属性 children 的 cc 类型为 Node 数组
    children: Node[] = [];

    @type(String) // 警告：不应该使用构造函数 String。等价于 CCString。也可以选择不声明类型
    text = '';
    // JavaScript 原始类型 `number`、`string`、`boolean` 通常可以不用声明
    // 可以直接写
    @property
    text = '';
}
```

visible

一般情况下，属性是否显示在 **属性检查器** 中取决于属性名是否以 _ 开头。**如果是以 _ 开头，则不显示**。

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

```
@property({ visible: true })
private _num = 0;
```

如果要强制隐藏，可以设置 `visible` 参数为 false：

```
@property({ visible: false })
num = 0;
```

serializable

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属性值保存到场景等资源文件中，之后在加载场景时就会自动还原成设置好的属性值。如果不想序列化，可以设置 `serializable: false`。

```
@property({ serializable: false })
num = 0;
```

override

所有属性都会被子类继承，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：

```
@property({ tooltip: "my id", override: true })
id = "";
```

group

当脚本中定义的属性过多且杂时，可通过 `group` 对属性进行分组、排序，方便管理。同时还支持对组内属性进行分类。

`group` 写法包括以下两种：

- `@property({ group: { name } })`
- `@property({ group: { id, name, displayOrder, style } })`

参数	说明
`id`	分组 ID，`string` 类型，是属性分组组号的唯一标识，默认为 `default`。
`name`	组内属性分类的名称，`string` 类型。
`displayOrder`	对分组进行排序，`number` 类型，数字越小，排序越靠前。默认为 `Infinity`，表示排在最后面。 若存在多个未设置的分组，则以在脚本中声明的先后顺序进行排序
`style`	分组样式，目前支持 **tab** 和 **section** 样式。 默认为 **tab**。

示例脚本如下：

```
import { _decorator, Component, Label, Sprite } from 'cc';
const { ccclass, property } = _decorator;

@ccclass('SayHello')
export class SayHello extends Component {

    // 分组一
    // 组内名为 "bar" 的属性分类，其中包含一个名为 label 的 Label 属性
    @property({ group: { name: 'bar' }, type: Label })
    label: Label = null!;
    // 组内名为 "foo" 的属性分类，其中包含一个名为 sprite 的 Sprite 属性
    @property({ group: { name: 'foo' }, type: Sprite })
    sprite: Sprite = null!;

    // 分组二
    // 组内名为 "bar2" 的属性分类，其中包含名为 label2 的 Label 属性和名为 sprite2 的 Sprite 属性，并且指定排序为 1。
    @property({ group: { name: 'bar2', id: '2', displayOrder: 1 }, type: Label })
    label2: Label = null!;
    @property({ group: { name: 'bar2', id: '2' }, type: Sprite })
    sprite2: Sprite = null!;

}
```

将该脚本挂载到节点上，则在 **属性检查器** 中显示为：

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`。

属性检查器相关参数

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 **属性检查器** 面板中显示或隐藏	boolean	(注1)	详见 visible 参数
displayName	在 **属性检查器** 面板中显示为另一个名字	string	undefined	-
tooltip	在 **属性检查器** 面板中添加属性的 Tooltip	string	undefined	-
multiline	在 **属性检查器** 面板中使用多行文本框	boolean	false	-
readonly	在 **属性检查器** 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 **属性检查器** 面板中显示为滑动条	boolean	false	-
group	在 **属性检查器** 面板中显示为分组，样式默认为 tab { name } 或 { id, name, displayOrder, style }	undefined	详见 group 参数	

序列化相关参数

以下参数不能用于 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
formerlySerializedAs	指定之前序列化所用的字段名	string	undefined	重命名属性时，声明这个参数来兼容之前序列化的数据
editorOnly	在导出项目前剔除该属性	boolean	false	-

其它参数

参数名	说明	类型	默认值	备注
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

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。

生命周期回调

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。开发者只需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，开发者不需要手工调用它们。

目前提供给开发者的生命周期回调函数主要有（按生命周期触发先后排列）：

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

onLoad

组件脚本的初始化阶段，我们提供了 `onLoad` 回调函数。`onLoad` 回调会在节点首次激活时触发，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：

```
import { _decorator, Component, Node, SpriteFrame, find } from 'cc';
const { ccclass, property } = _decorator;

@ccclass("test")
export class test extends Component {
    @property({type:SpriteFrame})
    bulletSprite=null;
    @property({type:Node})
    gun=null;

    _bulletRect=null;

    onLoad(){
        this._bulletRect=this.bulletSprite.getRect();
        this.gun = find('hand/weapon', this.node);
    }
}
```

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。

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。

```
import { _decorator, Component, Node } from 'cc';
const { ccclass, property } = _decorator;

@ccclass("starttest")
export class starttest extends Component {

    private _timer: number = 0.0;

    start () {
        this._timer = 1.0;
    }

    update (deltaTime: number) {
        this._timer += deltaTime;
        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。

```
import { _decorator, Component, Node } from 'cc';
const { ccclass, property } = _decorator;

@ccclass("updatetest")
export class updatetest extends Component {

    update (deltaTime: number) {
        this.node.setPosition(0.0,40.0*deltaTime,0.0);
    }
}
```

lateUpdate

`update` 会在所有动画更新前执行，但如果我们要在动效（如动画、粒子、物理等）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。

```
import { _decorator, Component, Node } from 'cc';
const { ccclass, property } = _decorator;

@ccclass("lateupdatetest")
export class lateupdatetest extends Component {

    lateUpdate (deltaTime: number) {
        this.node.setPosition(0.0,50,0.0);
    }
}
```

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。

开发注意事项

开发注意事项

ReadOnly

由于 `Readonly` 是只读属性，不建议对其进行写操作。因此直接通过 `Readonly` 属性调用接口修改值的方式，不保证在各平台都会生效。

例如，节点的世界坐标系内的位置，是 `Readonly` 的：

```
/**
* @en Position in world coordinate system
* @zh 世界坐标系下的坐标
```

```
*/
get worldPosition(): Readonly<math.Vec3>;
set worldPosition(val: Readonly<math.Vec3>);
```

在原生平台，此处 `add` 的结果不保存到 `worldPosition` 上：

```
this.node.worldPosition.add(xxx); // 结果不会被保存!
```

为避免这种情况，可通过中间变量来传递，代码示例如下：

```
let ret = this.node.worldPosition.add(diff.multiplyScalar(this.speedFactor));
this.node.setWorldPosition(ret);
```

常见属性包括但不限于以下所列的属性：`position`、`rotation`、`scale`、`worldPosition`、`worldRotation`、`worldScale`、`eulerAngles` 以及 `worldMatrix`，都建议采用上述方法使用。

脚本使用

脚本使用

该部分内容主要介绍如何在项目中使用脚本：

访问节点和其他组件

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() {
    let node = this.node;
    node.setPosition(0.0, 0.0, 0.0);
}
```

获得其它组件

如果你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。

```
import { _decorator, Component, Label } from 'cc';
const { ccclass, property } = _decorator;

@ccclass("test")
export class test extends Component {
    private label: any = null

    start() {
        this.label = this.getComponent(Label);
        let text = this.name + 'started';
        // Change the text in Label Component
        this.label.string = text;
    }
}
```

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

```
let 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() {
    console.log( this.node.getComponent(Label) === this.getComponent(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：

```
import { _decorator, Component, Label } from 'cc';
const { ccclass, property } = _decorator;

@ccclass("test")
export class test extends Component {
    private label: any = null;

    start() {
        this.label = this.getComponent(Label);
        if (this.label) {
            this.label.string = "Hello";
        } else {
            console.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 `Node` 的属性：

```
// Cannon.ts

import { _decorator, Component, Node } from 'cc';
const { ccclass, property } = _decorator;

@ccclass("Cannon")
export class Cannon extends Component {
    // 声明 Player 属性
    @property({ type: Node })
    private player = null;
}
```

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 `Node`。这就相当于在其它语言里声明了 `public 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：

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 Node 和 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。

激活/关闭节点

节点默认是激活的，我们可以在代码中设置它的激活状态，方法是设置节点的 `active` 属性：

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

设置 `active` 属性和在编辑器中切换节点的激活、关闭状态，效果是一样的。当一个节点是关闭状态时，它的所有组件都将被禁用。同时，它所有子节点，以及子节点上的组件也会跟着被禁用。要注意的是，子节点被禁用时，并不会改变它们的 `active` 属性，因此当父节点重新激活的时候它们就会回到原来的状态。

也就是说，`active` 表示的其实是该节点 **自身的** 激活状态，而这个节点 **当前** 是否可被激活则取决于它的父节点。并且如果它不在当前场景中，它也无法被激活。我们可以通过节点上的只读属性 `activeInHierarchy` 来判断它当前是否已经激活。

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

若节点原先就处于 **可被激活** 状态，修改 `active` 为 true 就会立即触发激活操作：

- 在场景中重新激活该节点和节点下所有 active 为 true 的子节点
- 该节点和所有子节点上的所有组件都会被启用，它们中的 `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

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

如该节点原先就已经被激活，修改 `active` 为 false 就会立即触发关闭操作：

- 在场景中隐藏该节点和节点下的所有子节点
- 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

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

这两种方法是等价的。

注意：

- 在 v3.0 之前 的版本中`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。v3.0 之后不再需要该参数。
- 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`：返回节点的所有子节点数组。
- `this.node.children.length`：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）

更改节点位置

有以下两种方法：

1. 使用 `setPosition` 方法：
    - `this.node.setPosition(100, 50, 100);`
    - `this.node.setPosition(new Vec3(100, 50, 100));`
2. 设置 `position` 变量：

    ```
    this.node.position = new Vec3(100, 50, 100);
    ```

更改节点旋转

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

或通过欧拉角设置本地旋转：

```
this.node.setRotationFromEuler(90, 90, 90);
```

更改节点缩放

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

常用组件接口

`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(deltaTime: number)：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

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。

以下是一个简单的例子:

```
import { _decorator, Component, Node } from 'cc';
const { ccclass, property } = _decorator;

@ccclass("test")
export class test extends Component {

    start(){
        let node =new Node('box');
        node.setPosition(0,0,-10);
    }
}
```

需要注意的是，在上述的示例中通过 new Node 创建出来的节点并不会主动添加到场景内，直至用户调用 director.getScene().addChild(node) 来添加到场景内或者通过 node.parent = {a valid node} 来作为某个节点的子节点。

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 instantiate 方法完成。使用方法如下：

```
import { _decorator, Component, Node,instantiate, director } from 'cc';
const { ccclass, property } = _decorator;

@ccclass("test")
export class test extends Component {

    @property({type:Node})
    private target: Node = null;

    start(){
        let scene = director.getScene();
        let node = instantiate(this.target);

        scene.addChild(node);
        node.setPosition(0, 0,-10);
    }
}
```

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 instantiate 生成节点。使用方法如下：

```
import { _decorator, Component, Prefab, instantiate, director } from 'cc';
const { ccclass, property } = _decorator;

@ccclass("test")
export class test extends Component {

    @property({type:Prefab})
    private target: Prefab = null;

    start(){
        let scene = director.getScene();
        let node = instantiate(this.target);

        scene.addChild(node);
        node.setPosition(0,0,0);
    }
}
```

需要注意的是，在使用上述的代码时，需将持有该代码的脚本挂在某个节点上，并在 属性检查器 内配置好 target 的值。

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 isValid 判断当前节点是否已经被销毁。

使用方法如下：

```
import { _decorator, Component, Node } from 'cc';
const { ccclass, property } = _decorator;

@ccclass("test")
export class test extends Component {

    @property({type:Node})
    private target: Node = null;

    private positionz: number = -20;

    start(){
        // 5秒后销毁节点
        setTimeout(function () {
            this.target.destroy();
          }.bind(this), 5000);
    }
    update(deltaTime: number){
        console.info(this.target.isValid);
        this.positionz += 1*deltaTime;
        if (this.target.isValid) {
            this.target.setPosition(0.0,0.0,this.positionz);
          }
    }
}
```

destroy 和 removeFromParent 的区别

调用一个节点的 removeFromParent 后，它并不会从内存中释放，因为引擎内部仍会持有它的数据。**因此如果一个节点不再使用，请直接调用它的 destroy 而不是 removeFromParent，否则会导致内存泄漏。**

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。

使用计时器

使用计时器

在 Cocos Creator 中，我们为组件提供了方便的计时器。

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

    ```
    this.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
    let interval = 5;
    // 重复次数
    let repeat = 3;
    // 开始延时
    let delay = 10;
    this.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 + 1** 次。

3. 只执行一次的计时器（快捷方式）

    ```
    this.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++;
    }
    this.schedule(this.callback, 1);
    ```

    注意：组件的计时器调用回调时，会将回调的 `this` 指定为组件本身，因此回调中可以直接使用 `this`。

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我们的 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：Node 不包含计时器相关 API

组件和组件执行顺序

组件和组件执行顺序

所有继承自 Component 的类都称为组件类，其对象称为组件，实现了 Cocos Creator 3.0 EC 系统中的组件概念。

组件类必须是 cc 类。

```
import { Component } from 'cc';

@ccclass("MyComponent")
class MyComponent extends Component {

}
```

组件的创建和销毁

组件的生命周期完全由节点操控。与普通类对象不同，组件不能由构造函数创建：

```
const component = new MyComponent(); // 错误：组件无法由构造函数创建
```

相反地，组件必须由节点来创建，通过如下方法将组件添加到节点上：

```
const myComponent = node.addComponent(MyComponent);
```

当组件不再被需要的时候，可以调用 `node.removeComponent(myComponent)` 移除指定的组件并将其销毁。

```
import { Component } from 'cc';

@ccclass("MyComponent")
class MyComponent extends Component {
    constructor () {
        console.log(this.node.name); // 错误：组件并未附加到节点上
    }

    public printNodeName () {
        console.log(this.node.name);
    }
}

const myComponent = node.addComponent(MyComponent);
myComponent.printNodeName(); // 正确
node.removeComponent(myComponent);
```

```
myComponent.printNodeName(); // 错误：组件已被该节点移除
```

组件执行顺序

使用统一的控制脚本来初始化其他脚本

项目中一般会有一个像 `Game.ts` 这样的脚本作为总的控制脚本，而其余脚本，像 `Configuration.ts`、`GameData.ts` 和 `Menu.ts` 三个组件，如果要在 `Game.ts` 里初始化，那么它们的初始化过程是这样的：

```
// Game.ts
import { _decorator, Component, Node } from 'cc';
const { ccclass, property } = _decorator;

import { Configuration } from './Configuration';
import { GameData } from './GameData';
import { Menu }from './Menu';

@ccclass("Game")
export class Game extends Component {
    private configuration = Configuration;
    private gameData = GameData;
    private menu = Menu;

    onLoad () {
        this.configuration.init();
        this.gameData.init();
        this.menu.init();
    }
}
```

其中在 `Configuration.ts`、`GameData.ts` 和 `Menu.ts` 中需要实现 `init` 方法，并将初始化逻辑放进去。这样就可以保证 Configuration、GameData 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，也可以将分散在每个脚本里的 `update` 替换成自己定义的方法：

```
    //Configuration.ts
    static updateConfig (deltaTime: number) {

    }
```

然后在 `Game.ts` 脚本的 `update` 里调用这些方法：

```
    // Game.ts
    update (deltaTime: number) {
        this.configuration.updateConfig(deltaTime);
        this.gameData.updateData(deltaTime);
        this.menu.updateMenu(deltaTime);
    }
```

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。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，即执行顺序。

假如有两个组件 CompA 和 CompB，它们的内容分别是：

```
// CompA.ts
import { _decorator, Component, Node } from 'cc';
const { ccclass, property } = _decorator;

@ccclass("CompA")
export class CompA extends Component {

    onLoad () {
        console.log('CompA onLoad!');
    }

    start () {
        console.log('CompA start!');
    }

    update (deltaTime: number) {
        console.log('CompA update!');
    }
}


// CompB.ts
import { _decorator, Component, Node } from 'cc';
const { ccclass, property } = _decorator;

@ccclass("CompB")
export class CompB extends Component {

    onLoad () {
        console.log('CompB onLoad!');
    }

    start () {
        console.log('CompB start!');
    }

    update (deltaTime: number) {
        console.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 组件右上角设置菜单里的 `Move Down` 将 CompA 移到 CompB 下面后，输出：

```
CompB onLoad!
CompA onLoad!
CompB start!
CompA start!
CompB update!
CompA update!
```

设置组件执行优先级

如果以上方法还是不能提供所需的控制粒度，还可以直接设置组件的 **executionOrder**。executionOrder 会影响组件生命周期回调执行的优先级。executionOrder 越小，该组件相对其它组件就会越先执行。executionOrder 默认为 0，因此设置为负数的话，就会在其它默认的组件之前执行。设置方法如下：

```
//Configuration.ts
import { _decorator, Component, Node } from 'cc';
const { ccclass, executionOrder } = _decorator;
```

```
@ccclass("Configuration")
@executionOrder(-1)
export class Configuration extends Component {

    onLoad () {
        console.log('Configuration onLoad!');
    }
}

// Menu.ts
import { _decorator, Component, Node } from 'cc';
const { ccclass, executionOrder } = _decorator;

@ccclass("Menu")
@executionOrder(1)
export class Menu extends Component {

    onLoad () {
        console.log('Menu onLoad!');
    }
}
```

通过如上方法设置，Configuration.ts 的 onLoad 会在 Menu.ts 的 onLoad 方法之前执行。

注意： executionOrder 只对 `onLoad`、`onEnable`、`start`、`update` 和 `lateUpdate` 有效，对 `onDisable` 和 `onDestroy` 无效。

加载和切换场景

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：

```
director.loadScene("MyScene");
```

除此之外，从 v2.4 开始 Asset Bundle 还增加了一种新的加载方式：

```
bundle.loadScene('MyScene', function (err, scene) {
    director.runScene(scene);
});
```

Asset Bundle 提供的 `loadScene` 只会加载指定 bundle 中的场景，并不会自动运行场景，还需要使用 `director.runScene` 来运行场景。
`loadScene` 还提供了更多参数来控制加载流程，开发者可以自行控制加载参数或者在加载完场景后做一些处理。

更多关于加载 Asset Bundle 中的场景，可参考文档 Asset Bundle。

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：

```
director.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。 需要注意的是，目标节点必须为位于层级的根节点，否则设置无效。

如果要取消一个节点的常驻属性：

```
director.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。

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：

```
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

`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：

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

之后在合适的时间调用 `loadScene`，就可以真正切换场景。

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

就算预加载还没完成，你也可以直接调用 `director.loadScene`，预加载完成后场景就会启动。

获取和加载资源

获取和加载资源

Cocos Creator 3.0 采用与 Cocos Creator v2.x 统一的资源管理机制，在本篇教程，我们将介绍：

- 资源属性的声明
- 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

资源属性的声明

在 Cocos Creator 中，所有继承自 `Asset` 的类型都统称资源，如 `Texture2D`、`SpriteFrame`、`AnimationClip`、`Prefab` 等。它们的加载是统一并且自动化的，相互依赖的资源能够被自动预加载。

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：

```
//test.ts

import { _decorator, Component, Node, SpriteFrame } from 'cc';
const { ccclass, property } = _decorator;

@ccclass("test")
```

```
export class test extends Component {

    @property({type: SpriteFrame})
    private spriteFrame: SpriteFrame = null;
}
```

如何在属性检查器里设置资源

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。假设我们创建了这样一个脚本：

```
//test.ts
import { _decorator, Component, Node, SpriteFrame, Texture2D } from 'cc';
const { ccclass, property } = _decorator;

@ccclass("test")
export class test extends Component {

    @property({type: Texture2D})
    private texture: Texture2D = null;

    @property({type: SpriteFrame})
    private spriteFrame: SpriteFrame = null;
}
```

将它添加到节点后，在 **属性检查器** 中是这样的：

---

tsconfig 配置

tsconfig

项目中 `tsconfig.json` 的 **绝大多数** 编译选项并不影响 Cocos Creator 对 TypeScript 的编译。因此，你需要小心配置其中的某些选项，以使得 IDE 的检查功能和 Cocos Creator 的编译行为一致。

以下选项不应当显式修改：

- `compilerOptions.target`
- `compilerOptions.module`

例如，若将 `tsconfig.json` 设置为：

```
{
    "compilerOptions": {
        "target": "es5",
        "module": "cjs"
    }
}
```

那么以下脚本代码在（使用 `tsc` 作为检查器的）IDE 中不会引起错误，因为 `compilerOptions.module` 设置为了 `cjs`。

```
const myModule = require("path-to-module");
```

然而 Cocos Creator 隐含的 `compilerOptions.module` 是 `es2015`，因此在运行时可能会提示 "require 未定义" 等错误。

以下脚本代码对于 Cocos Creator 来说是合法的，但 IDE 可能会报告错误。因为 `compilerOptions.target` 设置为了 `es5`，而 `Set` 是 ES6 才引入的。

```
const mySet = new Set();
```

对于其他选项，你可以自由修改。

例如，当你希望禁止你项目中所有 TypeScript 脚本对隐式 `any` 的使用，就可以在 `tsconfig.json` 中将 `compilerOptions.noImplicitAny` 设为 `true`。如此当你用 Visual Studio Code 等 IDE 检查该文件时就会收到相应的错误提示。

对于大多数项目而言，`tsconfig.json` 的某些选项是固定的。例如 `compilerOptions.target`、`compilerOptions.module` 以及 Cocos Creator 的类型声明文件位置等。

由于 `tsc` 的良好设计，`extends` 选项使得 `tsconfig.json` 可以是级联的。Cocos Creator 意识到了这一点，因此固定的 `tsconfig` 选项被放置在 `{项目路径}/tmp/tsconfig.cocos.json` 下，并由 Cocos Creator 管理。

于是，项目根路径下的 `tsconfig` 可以如下配置以共享这些固定选项：

```
{
    extends: './tmp/tsconfig.cocos.json',
    compilerOptions: {
        /* 自定义的 tsconfig 选项 */
    }
}
```

所幸，当你创建新项目时，Creator 将会自动生成这样的 `tsconfig`。

脚本进阶

脚本进阶

在阅读到此章节时，默认您已经对脚本系统较为熟悉，包括装饰器等。否则，请参阅：

- 脚本基础
- 装饰器

实例化

通过脚本定义一个 Foo 类和 Bar 类，Foo 类需要使用 Bar 类定义的属性，此时可以在 Foo 类中将 Bar 类直接 new 出一个对象：

```
class Foo {
    public bar: Bar = null;;
    constructor() {
        this.bar = new Bar();
    }
}
let bar = new Foo();
```

实例方法

实例方法请在原型对象中声明：

```
class Foo {
    public text!: string;
    constructor() {
        this.text = "this is sprite"
    }
    // 声明一个名叫 "print" 的实例方法
    print() {
        console.log(this.text);
    }
```

```
}

let obj = new Foo();
// 调用实例方法
obj.print();
```

判断类型

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

```
class Sub extends Base {

}

let sub = new Sub();
console.log(sub instanceof Sub);  //true
console.log(sub instanceof Base);  //true

let base = new Base();
console.log(base instanceof Sub);  // false
```

静态变量和静态方法

静态变量或静态方法可以用 `static` 声明：

```
class Foo {
    static count = 0;
    static getBounds() {

    }
}
```

静态成员会被子类继承，继承时 Creator 会将父类的静态变量 **浅拷贝** 给子类，因此：

```
class Object {
    static count = 11;
    static range: { w: 100, h: 100 }
}

class Foo extends Object {

}

console.log(Foo.count);    // 结果是 11，因为 count 继承自 Object 类

Foo.range.w = 200;
console.log(Object.range.w);  // 结果是 200，因为 Foo.range 和 Object.range 指向同一个对象
```

如果不需要考虑继承，私有的静态成员也可以直接定义在类的外面：

```
// 局部方法
doLoad(sprite) {
    // ...
};
// 局部变量
let url = "foo.png";

class Sprite {
    public url = '';
    load() {
        this.url = url;
        doLoad(this);
    };
};
```

继承

父构造函数

注意：不论子类是否有定义构造函数，在子类实例化前，父类的构造函数都会被自动调用。

```
class Node {
    name: string;
    constructor() {
        this.name = "node";
    }
}

class Sprite extends Node {
    constructor() {
        super();
        // 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
        console.log(this.name); // "node"
        // 重新设置 this.name
        this.name = "sprite";
    }
}

let obj = new Sprite();
console.log(obj.name); // "sprite"
```

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：

```
class Shape {
    getName() {
        return "shape";
    }
};

class Rect extends Shape {
    getName () {
        return "rect";
    }
};

let obj = new Rect();
console.log(obj.getName());    // "rect"
```

get/set 方法

如果在属性中定义了 get/set，那么在访问属性的时候，就能触发预定义的 get/set 方法。

get

在属性中定义 get 方法：

```
@property
get num() {
    return this._num;
}
```

```
@property
private _num = 0;
```

get 方法可以返回任意类型的值。
定义了 get 方法的属性可以显示在 **属性检查器** 中，可以在代码中直接访问。

```
class Sprite {

    @property
    get width() {
        return this._width;
    }

    @property
    private _width = 0;

    print(){
        console.log(this.width);
    }
};
```

注意：

- 属性定义了 `get` 方法之后就不能被序列化，也就是 `serializable` 参数不可用。例如下面的写法，`width` 属性既不会在编辑器上显示，也不会序列化。

  ```
  get width() {
      return this._width;
  }

  @property({ type: CCInteger, tooltip: "The width of sprite" })
  private _width = 0;
  ```

- 定义了 `get` 方法的属性如果需要在编辑器中显示，需要定义 `property`。例如下面的写法，`width` 属性如果去掉 `@property` 就不会在编辑器上呈现，`_width` 属性会序列化。

  ```
  @property
  get width() {
      return this._width;
  }

  @property({ type: CCInteger, tooltip: "The width of sprite" })
  private _width = 0;
  ```

- 定义了 get 方法的属性本身是只读的，但返回的对象并不是只读的。开发者依然可以通过代码修改对象内部的属性，例如：

  ```
  get num() {
      return this._num;
  }

  @property
  private _num = 0;

  start() {
      console.log(this.num);
  }
  ```

set

在属性中定义 set 方法：

```
set width(value) {
    this._width = value
}

private _width = 0;

start() {
    this.width = 20;
    console.log(this.width);
}
```

set 方法接收一个传入参数，这个参数可以是任意类型。set 方法一般和 get 方法一起使用：

```
@property
get width() {
    return this._width;
}

set width(value) {
    this._width = value;
}

@property
private _width = 0;
```

> **注意：** set 方法不定义属性。

事件系统

事件系统

事件系统是游戏开发过程中需要涉及到交互常用的功能。使用事件系统不仅可以将输入行为（例如：键盘、鼠标、触摸）以事件的形式发送到应用程序，也可以将游戏过程中的发生的，需要其他对象关注的事情通过事件的形式回应。例如：游戏胜利后需要打开结算或者奖励界面。

事件使用

事件需要通过注册获取监听，详情请参考 监听和发射事件。

在事件监听和发射的基础上，Cocos Creator 支持了很多内置事件系统，包括：

- 输入事件系统
- 节点事件系统
- 屏幕事件系统

发射和监听事件

监听和发射事件

Cocos Creator 引擎提供了 `EventTarget` 类，用以实现自定义事件的监听和发射，在使用之前，需要先从 `'cc'` 模块导入，同时需要实例化一个 `EventTarget` 对象。

```
import { EventTarget } from 'cc';
const eventTarget = new EventTarget();
```

> **注意：** 虽然 `Node` 对象也实现了一些 `EventTarget` 的接口，但是我们不再推荐继续通过 `Node` 对象来做自定义事件的监听和发射，因为这样子做不够高效，同时我们也希望 `Node` 对象只监听与 `Node` 相关的事件。

监听事件

监听事件可以通过 `eventTarget.on()` 接口来实现，方法如下：

```
// 该事件监听每次都会触发，需要手动取消注册
eventTarget.on(type, func, target?);
```

其中 `type` 为事件注册字符串，`func` 为执行事件监听的回调，`target` 为事件接收对象。如果 `target` 没有设置，则回调里的 `this` 指向的就是当前执行回调的对象。

值得一提的是，事件监听接口 `on` 的第三个参数 `target`，主要是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
eventTarget.on('foo', function ( event ) {
  this.enabled = false;
}.bind(this));

// 使用第三个参数
eventTarget.on('foo', (event) => {
  this.enabled = false;
}, this);
```

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

取消监听事件

当我们不再关心某个事件时，我们可以使用 `off` 接口关闭对应的监听事件。

`off` 接口的使用方式有以下两种：

```
// 取消对象身上所有注册的该类型的事件
eventTarget.off(type);
// 取消对象身上该类型指定回调指定目标的事件
eventTarget.off(type, func, target);
```

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。

我们推荐的实现方式如下：

```
import { _decorator, Component, EventTarget } from 'cc';
const { ccclass } = _decorator;
const eventTarget = new EventTarget();

@ccclass("Example")
export class Example extends Component {
    onEnable () {
        eventTarget.on('foobar', this._sayHello, this);
    }

    onDisable () {
        eventTarget.off('foobar', this._sayHello, this);
    }

    _sayHello () {
        console.log('Hello World');
    }
}
```

事件发射

发射事件可以通过 `eventTarget.emit()` 接口来实现，方法如下：

```
// 事件发射的时候可以指定事件参数，参数最多只支持 5 个事件参数
eventTarget.emit(type, ...args);
```

事件参数说明

在发射事件时，我们可以在 `emit` 函数的第二个参数开始传递我们的事件参数。同时，在 `on` 注册的回调里，可以获取到对应的事件参数。

```
import { _decorator, Component, EventTarget } from 'cc';
const { ccclass } = _decorator;
const eventTarget = new EventTarget();

@ccclass("Example")
export class Example extends Component {
    onLoad () {
        eventTarget.on('foo', (arg1, arg2, arg3) => {
            console.log(arg1, arg2, arg3);  // print 1, 2, 3
        });
    }

    start () {
        let arg1 = 1, arg2 = 2, arg3 = 3;
        // At most 5 args could be emit.
        eventTarget.emit('foo', arg1, arg2, arg3);
    }
}
```

注意：出于底层事件派发的性能考虑，这里最多只支持传递 5 个事件参数。所以在传参时需要注意控制参数的传递个数。

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，具体的说明及使用方式请参考：

- 输入事件系统
- 节点事件系统

输入事件系统

输入事件系统

`EventTarget` 支持了一套完整的 事件监听和发射机制 。在 Cocos Creator 3.4.0 中，我们支持了 `input` 对象，该对象实现了 `EventTarget` 的事件监听接口，可以通过 `input` 对象监听全局的系统输入事件。而原先的 `systemEvent` 对象则从 v3.4.0 开始废弃了，未来将逐步移除，建议使用 `input` 对象作为替代。

`systemEvent` 和 `input` 二者的差异包括：

- **在类型定义上的差异**
    - `systemEvent` 的触摸事件回调的类型定义是 `(touch: Touch, event: EventTouch) => void`
    - `input` 的触摸事件回调的类型定义是 `(event: EventTouch) => void`
- **在优先级上的差异**
    - `systemEvent` 的事件监听器会被节点的事件监听器拦截

- `input` 对象优先级比节点高，不会被拦截

  注意：我们在 **v3.4.1** 中降低了 `input` 优先级，因此二者从 v3.4.1 开始在优先级上已经不存在差异了。

本篇文档我们将介绍在 Cocos Creator 中对全局输入事件的处理。

全局输入事件是指与节点树不相关的各种输入事件，由 `input` 来统一派发，目前支持了以下几种事件：

- 鼠标事件
- 触摸事件
- 键盘事件
- 设备重力传感事件

定义输入事件

上文提到的输入事件，都可以通过接口 `input.on(type, callback, target)` 注册。可选的 `type` 类型包括：

输入事件	type 类型

鼠标事件	`Input.EventType.MOUSE_DOWN`
	`Input.EventType.MOUSE_MOVE`
	`Input.EventType.MOUSE_UP`
	`Input.EventType.MOUSE_WHEEL`

触摸事件	`Input.EventType.TOUCH_START`
	`Input.EventType.TOUCH_MOVE`
	`Input.EventType.TOUCH_END`
	`Input.EventType.TOUCH_CANCEL`

键盘事件	`Input.EventType.KEY_DOWN`（键盘按下） `Input.EventType.KEY_PRESSING`（键盘持续按下） `Input.EventType.KEY_UP`（键盘释放）
设备重力传感事件	`Input.EventType.DEVICEMOTION`

指针事件

指针事件包括 **鼠标事件** 和 **触摸事件**。

- 事件监听器类型
  - 鼠标事件监听
    - `Input.EventType.MOUSE_DOWN`
    - `Input.EventType.MOUSE_MOVE`
    - `Input.EventType.MOUSE_UP`
    - `Input.EventType.MOUSE_WHEEL`
  - 触摸事件监听
    - `Input.EventType.TOUCH_START`
    - `Input.EventType.TOUCH_MOVE`
    - `Input.EventType.TOUCH_CANCEL`
    - `Input.EventType.TOUCH_END`
- 事件触发后的回调函数
  - 自定义回调函数：`callback(event);`
- 回调参数
  - EventMouse 或 EventTouch

指针事件的使用范例如下：

```
import { _decorator, Component, input, Input, EventTouch } from 'cc';
const { ccclass } = _decorator;

@ccclass("Example")
export class Example extends Component {
    onLoad () {
        input.on(Input.EventType.TOUCH_START, this.onTouchStart, this);
    }

    onDestroy () {
        input.off(Input.EventType.TOUCH_START, this.onTouchStart, this);
    }

    onTouchStart(event: EventTouch) {
        console.log(event.getLocation());  // Location on screen space
        console.log(event.getUILocation());  // Location on UI space
    }
}
```

键盘事件

- 事件监听器类型
  - `Input.EventType.KEY_DOWN`
  - `Input.EventType.KEY_PRESSING`
  - `Input.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
  - EventKeyboard

使用键盘事件的代码示例如下：

```
import { _decorator, Component, input, Input, EventKeyboard, KeyCode } from 'cc';
const { ccclass } = _decorator;

@ccclass("Example")
export class Example extends Component {
    onLoad () {
        input.on(Input.EventType.KEY_DOWN, this.onKeyDown, this);
        input.on(Input.EventType.KEY_UP, this.onKeyUp, this);
    }
```

```
    onDestroy () {
        input.off(Input.EventType.KEY_DOWN, this.onKeyDown, this);
        input.off(Input.EventType.KEY_UP, this.onKeyUp, this);
    }

    onKeyDown (event: EventKeyboard) {
        switch(event.keyCode) {
            case KeyCode.KEY_A:
                console.log('Press a key');
                break;
        }
    }

    onKeyUp (event: EventKeyboard) {
        switch(event.keyCode) {
            case KeyCode.KEY_A:
                console.log('Release a key');
                break;
        }
    }
}
```

设备重力传感事件

- 事件监听器类型
  - `Input.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
  - EventAcceleration

使用设备重力传感事件的代码示例如下：

```
import { _decorator, Component, input, Input, log } from 'cc';
const { ccclass } = _decorator;

@ccclass("Example")
export class Example extends Component {
    onLoad () {
        input.setAccelerometerEnabled(true);
        input.on(Input.EventType.DEVICEMOTION, this.onDeviceMotionEvent, this);
    }

    onDestroy () {
        input.off(Input.EventType.DEVICEMOTION, this.onDeviceMotionEvent, this);
    }

    onDeviceMotionEvent (event: EventAcceleration) {
        log(event.acc.x + "    " + event.acc.y);
    }
}
```

具体使用方法可参考范例 **event**（GitHub | Gitee），其中包含了键盘、重力感应、单点触摸、多点触摸等功能的实现。

3D 物体的触摸检测

3D 物体与 2D UI 节点的触摸检测不同：

- 2D UI 节点只需要通过 `UITransform` 组件提供的尺寸信息和节点的位置信息，就可以实现触摸检测，详情请参考 节点事件系统。
- 3D 物体的触摸检测需要通过射线检测来实现。具体做法是通过渲染 3D 物体的 Camera 到触点的屏幕坐标，生成一条射线，判断射线是否穿过想要检测的对象。具体代码实现如下：

```
import { _decorator, Component, Node, Camera, geometry, input, Input, EventTouch, PhysicsSystem } from 'cc';
const { ccclass, property } = _decorator;

@ccclass("Example")
export class Example extends Component {

    // Specify the camera rendering the target node.
    @property(Camera)
    readonly cameraCom!: Camera;

    @property(Node)
    public targetNode!: Node

    private _ray: geometry.Ray = new geometry.Ray();

    onEnable () {
        input.on(Input.EventType.TOUCH_START, this.onTouchStart, this);
    }

    onDisable () {
        input.off(Input.EventType.TOUCH_START, this.onTouchStart, this);
    }

    onTouchStart(event: EventTouch) {
        const touch = event.touch!;
        this.cameraCom.screenPointToRay(touch.getLocationX(), touch.getLocationY(), this._ray);
        if (PhysicsSystem.instance.raycast(this._ray)) {
            const raycastResults = PhysicsSystem.instance.raycastResults;
            for (let i = 0; i < raycastResults.length; i++) {
                const item = raycastResults[i];
                if (item.collider.node == this.targetNode) {
                    console.log('raycast hit the target node !');
                    break;
                }
            }
        } else {
            console.log('raycast does not hit the target node !');
        }
    }
}
```

节点事件系统

节点事件系统

`input` 对象支持了全局的 输入事件系统，全局输入事件包括鼠标、触摸、键盘和重力传感四种。而 `Node` 也实现了 `EventTarget` 的事件监听接口。在此基础上，我们提供了一些基础的节点相关的系统事件。

本篇文档着重介绍了与 UI 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 UI 相关节点上的，所以被称为节点事件，使用方式如下：

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

注意：我们不推荐直接使用事件的名称字符串注册事件监听了。例如上述代码示例，请不要使用 `node.on('mouse-down', callback, target)` 来注册事件监听。

2D UI 节点上的触摸事件监听依赖于 `UITransform` 组件。如果需要实现对 3D 物体的触摸检测，请参考文档 3D 物体的触摸检测。

鼠标事件类型和事件对象

鼠标事件在 PC 端才会触发，系统提供的事件类型如下：

枚举对象定义	事件触发的时机
Node.EventType.MOUSE_DOWN	当鼠标在目标节点区域按下时触发一次。
Node.EventType.MOUSE_ENTER	当鼠标移入目标节点区域时触发，不论是否按下。
Node.EventType.MOUSE_MOVE	当鼠标在目标节点区域中移动时触发，不论是否按下。
Node.EventType.MOUSE_LEAVE	当鼠标移出目标节点区域时触发，不论是否按下。
Node.EventType.MOUSE_UP	当鼠标从按下状态松开时触发一次。
Node.EventType.MOUSE_WHEEL	当鼠标滚轮滚动时触发。

鼠标事件（`Event.EventMouse`）的重要 API 请参考 鼠标事件 API（Event 标准事件 API 除外）。

触摸事件类型和事件对象

触摸事件在移动端和 PC 端都会触发，开发者若希望更好地在 PC 端进行调试，只需要监听触摸事件即可同时响应移动端的触摸事件和 PC 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	事件触发的时机
Node.EventType.TOUCH_START	当手指触点落在目标节点区域内时。
Node.EventType.TOUCH_MOVE	当手指在屏幕上移动时。
Node.EventType.TOUCH_END	当手指在目标节点区域内离开屏幕时。
Node.EventType.TOUCH_CANCEL	当手指在目标节点区域外离开屏幕时。

触摸事件（`Event.EventTouch`）的重要 API 请参考 触摸事件 API（Event 标准事件 API 除外）。

注意：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。

节点事件派发

Cocos Creator 在 `Node` 上支持了 `dispatchEvent` 接口，通过该接口派发的事件，会进入事件派发阶段。Creator 的事件派发系统是按照 Web 的事件冒泡及捕获标准 实现的，事件在派发之后，会经历下面三个阶段：

- **捕获**：事件从场景根节点，逐级向子节点传递，直到到达目标节点或者在某个节点的响应函数中中断事件传递
- **目标**：事件在目标节点上触发
- **冒泡**：事件由目标节点，逐级向父节点冒泡传递，直到到达根节点或者在某个节点的响应函数中中断事件传递

当调用 `node.dispatchEvent()` 时，意味着 `node` 就是上文提到的目标节点。在事件的传递过程中，我们可以通过调用 `event.propagationStopped = true` 来中断事件传递。

在 v3.0 中，我们移除了 `Event.EventCustom` 类，如果要派发自定义事件，需要先实现一个自定义的事件类，该类继承自 `Event` 类，例如：

```
// Event 由 cc 模块导入
import { Event } from 'cc';

class MyEvent extends Event {
    constructor(name: string, bubbles?: boolean, detail?: any) {
        super(name, bubbles);
        this.detail = detail;
    }
    public detail: any = null;  // 自定义的属性
}
```

屏幕事件系统

屏幕事件系统

简介

正如之前所讨论的，EventTarget 提供了事件监听和发射的功能。Cocos Creator v3.8.0 引入了 screen 对象，它实现了 EventTarget 接口。该对象允许注册全局系统屏幕事件。

支持的事件

以下是当前支持的全局屏幕事件概述：

事件名称	描述	支持平台	支持版本
`window-resize`	监听窗口大小变化	Web, Native, MiniGame	3.8.0
`orientation-change`	监听屏幕方向变化	Web, Native, MiniGame	3.8.3
`fullscreen-change`	监听全屏变化	Web	3.8.0

事件使用示例

```
import { _decorator, Component, screen, macro } from 'cc';
const { ccclass } = _decorator;

@ccclass("Example")
export class Example extends Component {
  onLoad() {
    // Register event listeners with the screen object
    screen.on('window-resize', this.onWindowResize, this);
    screen.on('orientation-change', this.onOrientationChange, this);
    screen.on('fullscreen-change', this.onFullScreenChange, this);
  }

  onDestroy() {
    // Unregister event listeners when the component is destroyed
    screen.off('window-resize', this.onWindowResize, this);
    screen.off('orientation-change', this.onOrientationChange, this);
    screen.off('fullscreen-change', this.onFullScreenChange, this);
  }

  onWindowResize(width: number, height: number) {
    console.log("Window resized:", width, height);
  }

  onOrientationChange(orientation: number) {
    if (orientation === macro.ORIENTATION_LANDSCAPE_LEFT || orientation === macro.ORIENTATION_LANDSCAPE_RIGHT) {
      console.log("Orientation changed to landscape:", orientation);
    } else {
      console.log("Orientation changed to portrait:", orientation);
    }
  }

  onFullScreenChange(width: number, height: number) {
```

```
    console.log("Fullscreen change:", width, height);
  }
}
```

事件 API

全局与节点触摸和鼠标事件 API

鼠标事件 API

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。
getButton	Number	**Event.EventMouse.BUTTON_LEFT** 或 **Event.EventMouse.BUTTON_RIGHT** 或 **Event.EventMouse.BUTTON_MIDDLE**。 |

全局鼠标事件 API

函数名	返回值类型	意义
getLocation	Vec2	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	Vec2	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	Vec2	获取鼠标距离上一次事件移动相对于左下角的距离对象，对象包含 x 和 y 属性。
getDeltaX	Number	获取当前鼠标距离上一次鼠标移动相对于左下角的 X 轴距离。
getDeltaY	Number	获取当前鼠标距离上一次鼠标移动相对于左下角的 Y 轴距离。

节点鼠标事件 API

函数名	返回值类型	意义
getUILocation	Vec2	获取当前鼠标在 UI 窗口内相对于左下角的坐标位置，对象包含 x 和 y 属性。
getUILocationX	Number	获取当前鼠标在 UI 窗口内相对于左下角的 X 轴位置。
getUILocationY	Number	获取当前鼠标在 UI 窗口内相对于左下角的 Y 轴位置。
getUIPreviousLocation	Vec2	获取上一次鼠标在 UI 窗口内相对于左下角的坐标位置，对象包含 x 和 y 属性。
getUIDelta	Vec2	获取鼠标距离上一次事件移动在 UI 坐标系下的距离对象，对象包含 x 和 y 属性。
getUIDeltaX	Number	获取当前鼠标距离上一次鼠标移动在 UI 窗口内相对于左下角的 X 轴距离。
getUIDeltaY	Number	获取当前鼠标距离上一次鼠标移动在 UI 窗口内相对于左下角的 Y 轴距离。

触摸事件 API

API 名	类型	意义
touch	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。

全局触摸事件 API

函数名	返回值类型	意义
getLocation	Vec2	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。
getStartLocation	Vec2	获取触点初始时的位置对象，对象包含 x 和 y 属性。
getPreviousLocation	Vec2	获取触点事件上次触发时的位置对象，对象包含 x 和 y 属性。
getDelta	Vec2	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。
getDeltaX	Number	获取触点距离上一次事件移动的 X 轴距离。
getDeltaY	Number	获取触点距离上一次事件移动的 Y 轴距离。

节点触摸事件 API

函数名	返回值类型	意义
getUILocation	Vec2	获取当前触点在 UI 窗口内相对于左下角的坐标位置，对象包含 x 和 y 属性。
getUILocationX	Number	获取当前触点在 UI 窗口内相对于左下角的 X 轴位置。
getUILocationY	Number	获取当前触点在 UI 窗口内相对于左下角的 Y 轴位置。
getUIStartLocation	Vec2	获取初始触点在 UI 窗口内相对于左下角的位置对象，对象包含 x 和 y 属性。
getUIPreviousLocation	Vec2	获取上一次触点在 UI 窗口内相对于左下角的坐标位置，对象包含 x 和 y 属性。
getUIDelta	Vec2	获取当前触点距离上一次触点移动在 UI 窗口内相对于左下角的距离对象，对象包含 x 和 y 属性。
getUIDeltaX	Number	获取当前触点距离上一次触点移动在 UI 窗口内相对于左下角的 X 轴距离。
getUIDeltaY	Number	获取当前触点距离上一次触点移动在 UI 窗口内相对于左下角的 Y 轴距离。

模块规范与示例

模块规范与示例

所有的代码文件可以大致分为 插件脚本 和 **模块** 两种，该部分内容主要介绍模块相关。

模块 是 TypeScript/JavaScript 代码的一种组织方式，按照模块组织的代码一般又被非正式地称为 **脚本/项目脚本**。在 Cocos Creator 中，除 **插件脚本** 外所有代码都以模块的形式组织，根据来源的不同，大致分为：

- 项目中创建的代码，包括 **组件脚本** 和 **项目类（非组件）脚本**；
- 引擎提供的功能，详情请参考 引擎模块；
- 第三方模块，例如 npm 模块。详情请参考 外部模块使用案例。

Cocos Creator 原生支持并推荐使用 ECMAScript 模块格式（简称 ESM 模块格式）。为了支持对外部模块的使用，Cocos Creator 也在某种限度上支持了 CommonJS 模块格式。关于 Creator 中模块的格式及使用，详情请参考 模块规范。

从 Cocos Creator 3.3 开始，支持导入映射（实验性），详情请参考 导入映射。

模块加载顺序

模块加载顺序如下：

1. 首次导入 Cocos Creator 3.x 的 引擎模块 `"cc"`。
2. 插件脚本：所有插件脚本将按照指定的插件脚本依赖关系顺序执行，不存在依赖关系的插件脚本之间是无序的。详情可参考 插件脚本。

3. 普通脚本：所有普通脚本将被并发导入。导入时将严格遵循由 `import` 确定的引用关系和执行顺序。

引擎模块

引擎模块

引擎通过模块向开发者暴露功能接口，模块以 **ECMAScript** 模块形式存在。

□□ 注意，从 v3.0 开始，将不能通过全局变量 `cc` 访问引擎功能！

功能

模块 '`cc`' 提供了所有引擎功能的访问。模块 '`cc`' 的内容是动态的，其内容和 **项目设置** 中的 **功能裁剪** 设置有关。

引擎日志输出

示例：

```
import { log } from 'cc';
log('Hello world!');
```

构建时常量

引擎模块 '`cc/env`' 暴露了一些构建时的 **常量**，这些常量代表执行环境、调试级别或平台标识等。

由于这些常量都以 `const` 声明，提供了很好的代码优化机会。

执行环境

名称（类型都为 `boolean`）	说明
`BUILD`	是否正在构建后的环境中运行
`PREVIEW`	是否正在预览环境中运行
`EDITOR`	是否正在编辑器环境中运行

调试级别

名称（类型都为 `boolean`）	说明
`DEBUG`	是否处于调试模式。仅当构建时未勾选调试选项的情况下为 `false`，其它情况下都为 `true`
`DEV`	等价于 `DEBUG`/`EDITOR`/`PREVIEW`

平台标识

下表列出的常量表示是否正在 **某一个** 或 **某一类** 平台上运行，常量的类型都是 `boolean`。

名称	代表平台	`MINIGAME` “小游戏”	`RUNTIME_BASED` 基于 Cocos Runtime	`SUPPORT_JIT` 支持 JIT
`HTML5`	Web	□	□	□
`NATIVE`	原生平台	□	□	□
`ALIPAY`	支付宝小游戏	✔□	□	✔□
`BAIDU`	百度小游戏	✔□	□	✔□
`BYTEDANCE`	抖音小游戏	✔□	□	✔□
`WECHAT`	微信小游戏	✔□	□	✔□
`XIAOMI`	小米快游戏	✔□	□	✔□
`COCOSPLAY`	Cocos Play	□	✔□	✔□
`HUAWEI`	华为快游戏	□	✔□	✔□
`OPPO`	OPPO 小游戏	□	✔□	✔□
`VIVO`	vivo 小游戏	□	✔□	✔□

调试模式下的输出

示例如下：

```
import { log } from 'cc';
import { DEV } from 'cc/env';

if (DEV) {
    log('I am in development mode!');
}
```

外部模块使用案例

示例：外部模块使用案例

本章节通过案例讲解如何在 Cocos Creator 项目中使用 npm 模块，如果不知道 npm 该如何获取，请参考 获取 npm 包。

ESM 与 CJS 交互规则

Cocos Creator 3.x 如何使用 npm，最大的问题在于 ESM 与 CJS 模块交互。如果还不了解这两个模块在 Cocos Creator 里是如何定义的，请查看 模块 一节。其实，ESM 和 CJS 模块的交互方式在 Node.js 官方文档 就有提到。在这里我简单的概括以下几点：

- CommonJS 模块由 `module.exports` 导出，在导入 CommonJS 模块时，可以使用 ES 模块默认的导入方式或其对应的 sugar 语法形式导入。

  ```
  import { default as cjs } from 'cjs';
  // 语法糖形式
  import cjsSugar from 'cjs';
  console.log(cjs); // <module.exports>
  console.log(cjs === cjsSugar); // true
  ```

- **ESM** 模块的 `default` 导出指向 **CJS** 模块的 `module.exports`。
- 非 `default` 部分的导出，Node.js 通过静态分析将其作为独立的 ES 模块提供。

接下来看一个代码片段：

```
// foo.js
module.exports = {
    a: 1,
    b: 2,
}

module.exports.c = 3;

// test.mjs
```

```
// default 指向 module.exports
import foo from './foo.js'; // 等价于 import { default as foo } from './foo.js'
console.log(JSON.stringify(foo)); // {"a":1,"b":2,"c":3}

// 导入 foo 模块的所有导出
import * as module_foo from './foo.js'
console.log(JSON.stringify(module_foo)); // {"c":3,"default":{"a":1,"b":2,"c":3}}

import { a } from './foo.js'
console.log(a); // Error: a is not defined

// 根据上方第三点，c 有独立导出
import { c } from './foo.js'
console.log(c); // 3
```

npm 模块使用案例

案例一：protobufjs 使用

首先，需要获取到 protobufjs 包。在项目目录下打开终端，执行 `npm i protobufjs`。如果这个项目属于多人协作，甚至可以把 `protobufjs` 这个包作为依赖写入 `package.json`，通过在上述命名行里加入 `npm install --save protobufjs` 即可利用命令行自动写入到 `package.json` 中。执行完之后，就可以在项目录下的 `node_module` 文件夹里查找到 `protobufjs` 相关文件夹。

模块规范

模块规范

模块格式

本节介绍了 Cocos Creator 如何决定一个模块的格式。

Cocos Creator 引擎提供的所有功能都以 ESM 模块的形式存在，见 引擎模块。

项目资源目录中（一般地，任何资产数据库中）以 `.ts` 作为后缀的文件都视为 ESM 模块。例如 `assets/scripts/foo.ts`。

对于任何其它模块格式，Cocos Creator 选择与 Node.js 类似的规则来 鉴别。具体地，以下文件将被视为 ESM 格式：

- 以 `.mjs` 为后缀的文件；
- 以 `.js` 为后缀的文件，并且与其最相近的父级 `package.json` 文件中包含一个顶级的 `"type"` 字段，其值为 `"module"`。

其余的文件将被视为 CommonJS 模块格式，这包括：

- 以 `.cjs` 为后缀的文件；
- 以 `.js` 为后缀的文件，并且与其最相近的父级 `package.json` 文件中包含一个顶级的 `"type"` 字段，其值为 `"commonjs"`。
- 不在上述条件下的以 `.js` 为后缀的文件。

模块说明符与模块解析

在 ESM 模块中，通过标准的导入导出语句与目标模块进行交互，例如：

```
import { Foo } from './foo';
export { Bar } from './bar';
```

导入导出语句中关键字 `from` 后的字符串称为 **模块说明符**。模块说明符也可作为参数出现在动态导入表达式 `import()` 中。

模块说明符用于指定目标模块，从模块说明符中解析出目标模块 URL 的过程称为 **模块解析**。

Cocos Creator 支持三种模块说明符：

- **相对说明符** 像 `'./foo'`、`'../bar'` 这样以 `'./'` 和 `'../'` 开头的说明符。
- **绝对说明符** 指定了一个 URL 的说明符。例如：`foo:/bar`。
- **裸说明符**（Bare specifier）像 `foo`、`foo/bar` 这样既不是 URL 又不是相对说明符的说明符。

相对说明符

相对说明符以当前模块的 URL 为基础 URL，以相对说明符为输入来解析目标模块的 URL。

例如，对于模块 `项目路径/assets/scripts/utils/foo` 来说，`'./bar'` 将解析为同目录下的 `项目路径/assets/scripts/utils/bar`；`'../baz'` 将解析为上层目录中的 `项目路径/assets/scripts/baz`。

绝对说明符

绝对说明符直接指定了目标模块的 URL。

资产数据库协议

通过 `db://<db-name>/x/y/z.mjs` 这样的 URL 可以访问某个资产数据库中的模块。该 URL 的组成部分如下：

- URL 协议头 `db:` 是固定的。
- URL 主机（host）为 `//<db-name>`。其中`<db-name>` 是资产数据库的标识符，一般情况下，它即是挂载了该资产数据库的插件的标识符。当 `<db-name>` 为 `assets` 时，指代项目资产数据库；为 `internal` 时，指代 Cocos Creator 内置资产数据库。
- URL 路径名是指定了模块相对于资产数据库根目录的路径。

文件协议

Cocos Creator 支持文件协议的 URL，例如 `file:///C:/x/y/z.mjs`。但由于文件 URL 中指定的文件路径是绝对路径，因此很少使用。

> 值得注意的是，在 Node.js 中，一种访问 Node.js 内置模块的方法是通过 `node:` 协议的 URL，例如：`node:fs`。Cocos Creator 会将所有对 Node.js 内置模块的访问请求解析为 `node:` URL 请求，例如 `import fs from 'fs'` 中的 `'fs'` 将解析为 `node:fs`。但 Cocos Creator 并不支持 Node.js 内置模块，也就是说并不支持 `node:` 协议。因此会产生加载错误。当使用 npm 中的模块时，可能会遇到该错误。

裸说明符

目前为止，对于裸说明符，Cocos Creator 将应用 导入映射（实验性质） 和 Node.js 模块解析算法。

> 这就包括了对 npm 模块的解析。

条件性导出

在 Node.js 模块解析算法中，包的条件性导出 特性用于根据一些条件映射包中的子路径。与 Node.js 类似，Cocos Creator 实现了内置条件 `import`、`default`，但未实现条件 `require`、`node`。

开发者可通过编辑器主菜单 **项目 -> 项目设置 -> 脚本** 中的 **导出条件** 项指定 **额外** 的条件，该项默认值为 **browser**，可用 **逗号** 作为分隔符来指定多个额外条件，例如 `browser, bar`。

若 **导出条件** 项使用默认值 `browser`，当某 npm 包 `foo` 的 `package.json` 中包含以下配置时：

```
{
   "exports": {
      ".": {
         "browser": "./dist/browser-main.mjs",
         "import": "./dist/main.mjs"
      }
   }
}
```

`"foo"` 将解析为包中路径为 `dist/browser-main.mjs` 的模块。

> 多玩家框架 Colyseus 中就为 `browser` 条件做了映射配置。

若 **导出条件** 项设置为空，则表示不指定任何额外条件，上例中的 `"foo"` 将解析为包中路径为 `dist/main.mjs` 的模块。

后缀与目录导入

Cocos Creator 对模块说明符中模块的后缀要求更偏向于 Web —— 必须指定后缀并且不支持 Node.js 式的目录导入。然而，基于历史原因和现行的一些限制，TypeScript 模块不允许给出后缀并支持 Node.js 式的目录导入。具体来说：

当目标模块文件的后缀是 `.js`、`.mjs` 时，模块说明符中 **必须指定** 后缀：

```
import './foo.mjs'; // 正确
import './foo'; // 错误：无法找到指定模块
```

Node.js 式的目录导入是不支持的：

```
import './foo/index.mjs'; // 正确
import './foo'; // 错误：无法找到模块。
```

> 这种后缀要求与对目录导入的限制同时应用到了相对说明符和绝对说明符。对于在裸说明符中的要求可参考 Node.js 模块解析算法。

但当目标模块文件的后缀是 `.ts` 时，模块说明符中 **不允许指定** 后缀：

```
import './foo'; // 正确：解析为同目录下的 `foo.ts` 模块
import './foo.ts'; // 错误：无法找到指定模块
```

另一方面，支持 Node.js 式的目录导入：

```
import './foo'; // 正确：解析为 `foo/index.ts` 模块
```

> **注意：**
>
> 1. Cocos Creator 支持 Web 平台。在 Web 平台上实现 Node.js 那样复杂的模块解析算法成本是昂贵的，客户端和服务端之间无法通过频繁的通讯来尝试不同的后缀和文件路径。
> 2. 即使通过一些后处理工具可以实现在构建阶段完成这样的复杂解析，但会造成静态导入解析（通过 `import` 语句）和动态导入解析（通过 `import()` 表达式）算法的不一致。因此在模块解析算法的选择上，我们更偏向于在代码中指定完整的文件路径。
> 3. 但我们却无法完全限制这一点，因为就目前来说，TypeScript 中不允许在说明符中指定后缀为 `.ts`。并且 TypeScript 尚且不支持自动补全特定的目标后缀。在这些限制下，我们很难做到两全其美，但我们仍在观测这些条件在未来是否有好转。

未支持 `browser` 字段

有些 npm 包的清单文件 `package.json` 中记录了 `browser` 字段，例如 JSZip。`browser` 字段用于指定当该包在非 Node.js 环境下特有的模块解析方法，它可使得包中的某些专用于 Node.js 的模块被替换为能够在 Web 中使用的模块。虽然 Cocos Creator **不支持该字段**，但如果对 npm 包有编辑的能力，Cocos Creator 推荐使用 条件化导出 和 子路径导入 来代替 `browser` 字段。

否则，可以以非 npm 的方式使用目标库。例如，将目标库中专为非 Node.js 环境制定的模块复制至项目中，再通过相对路径来导入。

CommonJS 模块解析

在 CommonJS 模块中，Cocos Creator 应用的是 Node.js CommonJS 模块解析算法。

模块格式交互

Cocos Creator 允许在 ESM 模块中导入 CommonJS 模块。

当从 ESM 模块中导入 CommonJS 模块时，CommonJS 模块的 `module.exports` 对象将作为 ESM 模块的默认导出：

```
import { log } from 'cc';

import { default as cjs } from 'cjs';

// 上面导入语句的另一种写法：
import cjsSugar from 'cjs';

log(cjs);
log(cjs === cjsSugar);
// 打印：
//   <module.exports>
//   true
```

CommonJS 模块的 ECMAScript 模块命名空间 表示，是含有一个 `default` 导出的命名空间，其中的 `default` 导出就指向了 CommonJS 模块的 `module.exports` 的值。

该 模块命名空间外来对象 可以通过 `import * as m from 'cjs'` 来观察：

```
import * as m from 'cjs';
console.log(m);
// 打印：
//   [Module] { default: <module.exports> }
```

Cocos Creator ESM 解析算法公示

Cocos Creator 用于解析 ESM 模块说明符的算法由以下的 `CREATOR_ESM_RESOLVE` 方法给出。它返回从当前 URL 解析模块说明符得到的 URL 结果。

在解析算法规范中，引用了Node ESM 解析算法 和 Import Map 解析算法（引用为 `IMPORT_MAP_RESOLVE`）。

解析算法规范

`CREATOR_ESM_RESOLVE(specifier, parentURL)`

1. Let `resolved` be the result of `CREATOR_STD_RESOLVE(specifier, parentURL)`.
2. If both `parentURL` and `resolved` are under project assets directory, then
    1. Let `extensionLessResolved` be the result of `TRY_EXTENSION_LESS_RESOLVE(resolved)`.
        1. If `extensionLessResolved` is not `undefined`, return `extensionLessResolved`.
3. Return `resolved`.

`CREATOR_STD_RESOLVE(specifier, parentURL)`

1. If import map configured, then
    1. Let `resolved` be the result of `IMPORT_MAP_RESOLVE(specifier, parentURL)`, with parsed import map.
    2. If `resolved` is not nil, return `resolved`.
2. return `ESM_RESOLVE(specifier, parentURL)`.

`TRY_EXTENSION_LESS_RESOLVE(url)`

1. If the file at `url` exists, then

    1. Return `url`.
2. Let `baseName` be the portion after the last "/" in pathname of `url`, or whole pathname if it does not contain a "/".
3. If `baseName` is empty, then
    1. Return `undefined`.
4. Let `resolved` be the result URL resolution of "./" concatenated with `baseName` and `.ts`, relative to parentURL.
    1. If the file at `resolved` exists, then
    2. Return `resolved`.
5. Let `resolved` be the result URL resolution of "./" concatenated with `baseName` and `/index.ts`, relative to parentURL.
    1. If the file at `resolved` exists, then
    2. Return `resolved`.
6. Return `undefined`.

导入映射

导入映射（实验性）

Cocos Creator 从 v3.3 开始实验性支持 导入映射（Import maps）。

导入映射控制 TypeScript/JavaScript 的导入行为，尤其是可指定对 裸说明符 的解析。

使用

通过编辑器顶部菜单栏的 **项目 -> 项目设置 -> 脚本** 中的 **导入映射** 项即可指定导入映射文件的路径。设置完成后，导入映射功能开启，使用的导入映射将从指定的文件中读取。

> **注意**：导入映射文件的路径是至关重要的，因为导入映射中的所有相对路径都是相对于导入映射文件本身路径。

别名映射

若有一个模块被项目中所有的模块所使用，而开发者并不希望其他模块以相对路径的方式引用它，而是为它起一个别名，那么便可以选择使用导入映射。

例如，某个模块真实的绝对路径为 `<项目>/assets/lib/foo.ts`，我们希望所有模块可以以 `import {} from 'foo';` 的方式来引用它，操作步骤如下：

首先，在项目目录下创建一个导入映射文件 `import-map.json`：

```
// import-map.json

{
    "imports": {
        "foo": "./assets/lib/foo.ts"
    }
}
```

- `"imports"`：指定应用到所有模块的 **顶级映射**（Top level imports）。
- `"foo"`：指定我们要映射的模块名。
- `"./assets/lib/foo.ts"`：指定如何映射 `"foo"`。`"./assets/lib/foo.ts"` 是相对路径，**导入映射中的所有相对路径都是相对于导入映射文件本身的位置的**，因此 `./assets/lib/foo.ts` 将解析为绝对路径 `<项目>/assets/lib/foo.ts`。

然后在任意模块中使用以下方式引用模块时，`'foo'` 都将解析为模块 `<项目>/assets/lib/foo.ts`：

```
import * as foo from 'foo';
```

目录映射

导入映射还允许映射指定目录下的所有模块。

例如，要映射项目 `assets/lib/bar-1.2.3` 目录下的所有模块，则导入映射的 json 文件如下所示：

```
// import-map.json

{
    "imports": {
        "bar/": "./assets/lib/bar-1.2.3/"
    }
}
```

除了 `"bar/"` 指定的是我们要映射的目录，其余的与 **别名映射** 一致。

这样项目中的模块都能以 `import {} from 'bar/...'` 的形式来引用目录 `bar-1.2.3` 中的模块。

例如：

```
import * as baz from 'bar/baz';
import * as quux from 'bar/qux/quux';
```

`'bar/baz'` 将解析为模块 `<项目>/assets/lib/bar-1.2.3/baz.ts`
`'bar/qux/quux'` 将解析为模块 `<项目>/assets/lib/bar-1.2.3/qux/quux.ts`。

TypeScript 配置

TypeScript 并不支持导入映射，在使用时可能会导致出现找不到模块的报错，所以我们需要通过额外的配置来告诉 TypeScript 类型检查器额外的模块解析信息。

例如上述中的两个例子，可以在项目目录下的 `tsconfig.json` 文件中配置 paths 字段（若没有该字段，可自行补上），如下所示：

```
// tsconfig.json
{
    "compilerOptions": {
        "paths": {
            // 注意：这里的相对路径是相对于 tsconfig.json 所在的路径
            // 由于本例中 tsconfig.json 和 import-map.json 位于同一目录，因此这里的相对路径也相似。
            "foo": ["./assets/lib/foo"],
            "bar/*": ["./assets/lib/bar-1.2.3/*"]
        }
    }
}
```

更多关于导入映射的功能，请参考 导入映射。

支持情况

Cocos Creator 支持 Import Maps Draft Community Group Report, 12 January 2021 中的所有功能。

外部代码支持

外部代码支持

> **注意**：Cocos Creator 3.x 推荐使用模块代替插件脚本的使用！

插件脚本

当脚本资源导入到 **资源管理器** 后，在 **属性检查器** 中设置了 **导入为插件**，此脚本资源便称为 **插件脚本**。插件脚本通常用于引入第三方库。目前仅支持 JavaScript 插件脚本。

跨平台发布

跨平台发布游戏

得益于 Cocos Creator 的双内核引擎架构（C++ 内核用于原生平台，TS 内核用于 Web和小游戏平台），使得使用 Cocos Creator 制作的项目可以在原生平台和 Web 平台和小游戏平台上都运行良好。真正实现一次开发，全平台运行。

发布平台

目前 Cocos Creator 支持发布到以下平台：

- 发布 iOS 应用
- 发布 Android 应用
- 发布 HUAWEI HarmonyOS 应用
- 发布 OpenHarmony 应用
- 发布 Windows 应用
- 发布 macOS 应用
- 发布 Web 移动端 - H5
- 发布 Web 桌面端
- 发布微信小游戏
- 发布淘宝小程序创意互动
- 发布抖音小游戏
- 发布 Facebook 小游戏
- 发布 OPPO 小游戏
- 发布华为快游戏
- 发布 vivo 小游戏
- 发布小米快游戏

准备工作

在项目正常开发，预览效果达到要求的情况下，可以将游戏发布到多个平台。在发布之前需要做的准备工作包括：

- 熟悉构建发布面板
- 了解通用构建选项

进阶

如果对构建流程有了一定程度上的熟悉和了解，就可以自定义构建模板、自定义构建流程等。详情可参考以下文档：

- 构建流程简介与常见错误处理
- 定制项目的构建模版
- 自定义构建流程

开发者还可以通过命令行发布项目，详情请参考 命令行发布项目。

更多资源

- Cocos 技术支持
- Cocos 官方论坛

发布 Android 应用

安卓平台构建打包

本章将说明如何将 Cocos Creator 应用构建并打包到安卓平台。

主要内容

- Android 构建示例
- Android 构建选项介绍
- Android 原生开发环境配置
- 基于反射机制实现 JavaScript 与 Android 系统原生通信
- 使用 JsbBridge 实现 JavaScript 与 Java 通信
- 特性与系统版本

相关阅读

- 构建发布面板
- 通用构建选项介绍
- 原生平台通用构建选项
- 安装配置原生开发环境
- 原生平台 JavaScript 调试
- 构建流程简介与常见错误处理
- 原生平台二次开发指南

Android 构建示例

安卓构建示例

本文将演示 Cocos Creator 项目发布为 Android 应用程序的流程。

请准备一个至少含有一个场景的 Cocos Creator 项目。

Android 构建选项

Android 平台构建选项

Android 平台的构建选项如下：

配置 Android 原生开发环境

Android 原生开发环境配置

下载 Android Studio

开发者可以从 Android Studio 官方网站 下载对应的 IDE。

参考 安装配置原生开发环境 搭建开发环境。

下载并安装 JDK

参考 安装配置原生开发环境 - 下载 Java SDK（JDK）

在终端中输入 `java -version` 查看是否安装成功

---

v3.8 Android 工程升级

v3.8 Android 工程升级

从 v3.8 开始，构建产出的 Android 工程默认支持新版本的 Android Studio（Flamingo | 2022.2.1）。由于 Android Gradle 插件的 要求，开发者需要将 JDK 升级到 17，同时升级 Android Studio 到 Flamingo 版本。

> 如果没有对构建生成的 Android 工程进行配置，可以直接删除 native/engine/android 目录和 build/android 目录，然后重新构建。这样不需要对工程进行一步步的修改升级。但是请注意，这一步操作是有风险的，如对接 SDK 的代码可能会被删除，请谨慎操作。

对于现有的原生 Android 工程，开发者可以参考以下步骤对工程进行升级：

第一步：备份当前工程

在升级之前，我们应该先备份当前的 native 目录，以防万一。比如可以用 Git 保存当前修改.

第二步：升级Gradle插件版本

Gradle 插件版本是 Gradle 与 Android 构建系统之间的接口。因此，在升级 Gradle 之前，我们需要先升级 Gradle 插件版本。在项目的 native/engine/android/build.gradle 文件中，将 classpath 中的 Gradle 插件版本改为 8.0.2。

```
        // jcenter() // keeped as anchor, will be removed soon
    }
    dependencies {
-       classpath 'com.android.tools.build:gradle:4.1.0'
+       classpath 'com.android.tools.build:gradle:8.0.2'

        // NOTE: Do not place your application dependencies here; they belong
        // in the individual module build.gradle files
```

第三步：移除 package 字段

移除 native/engine/android/app/AndroidManifest.xml 文件中的 package 属性。

```
<?xml version="1.0" encoding="utf-8"?>
 <manifest xmlns:android="http://schemas.android.com/apk/res/android"
-    package="com.cocos.test"
     android:installLocation="auto">

     <uses-permission android:name="android.permission.INTERNET"/>
```

在 `native/engine/android/app/build.gradle` 中的修改 applicationId 为 namespace

```
     compileSdkVersion PROP_COMPILE_SDK_VERSION.toInteger()
     buildToolsVersion PROP_BUILD_TOOLS_VERSION
     ndkPath PROP_NDK_PATH
+    namespace APPLICATION_ID

     compileOptions {
         sourceCompatibility JavaVersion.VERSION_1_8
@@ -17,7 +18,6 @@ android {
     }

     defaultConfig {
-        applicationId APPLICATION_ID
         minSdkVersion PROP_MIN_SDK_VERSION
         targetSdkVersion PROP_TARGET_SDK_VERSION
         versionCode 1
```

第四步：升级 Gradle 版本

接下来，我们需要升级 Gradle wrapper 版本。在项目的 `build/android/proj/gradle/wrapper/gradle-wrapper.properties` 文件中，将 distributionUrl 改为 8.0.2，如下所示：

```
distributionUrl=https\://services.gradle.org/distributions/gradle-8.0.2-bin.zip
```

第五步: 更新 Proguard Rules

添加下面的内容到文件 `native/engine/android/app/proguard-rules.pro`

```
# This is generated automatically by the Android Gradle plugin.
-dontwarn android.hardware.BatteryState
-dontwarn android.hardware.lights.Light
-dontwarn android.hardware.lights.LightState$Builder
-dontwarn android.hardware.lights.LightState
-dontwarn android.hardware.lights.LightsManager$LightsSession
-dontwarn android.hardware.lights.LightsManager
-dontwarn android.hardware.lights.LightsRequest$Builder
-dontwarn android.hardware.lights.LightsRequest
-dontwarn android.net.ssl.SSLSockets
-dontwarn android.os.VibratorManager
```

第六步：安装 JDK 17

在 官网 上下载并安装 JDK 17。安装过程中需要注意配置环境变量。

安装完成后，可以通过在命令行输入 `java -version` 来检查是否安装成功。

第七步：升级 Android Studio

在升级 Gradle 之后，我们还需要升级 Android Studio。如果您当前使用的是较老版本的 Android Studio，请先下载最新版本的 Android Studio（Flamingo | 2022.2.1）。下载地址为: https://developer.android.com/studio。

下载完成后，打开 Android Studio 并导入您的项目。Android Studio 会自动检测您的项目所需的 Gradle 版本，并提示您进行升级。按照提示进行操作，即可完成 Android Studio 的升级。

如果在 Android Studio 编译时遇到下面的错误报告：

发布到谷歌 GPG 平台

Google Play Games on PC

Google Play Games （GPG） 是由谷歌开发，可以让您的移动端 APK 在 PC 上发布、游玩的相关技术。

自 v3.8 起，Cocos Creator 将提供发布到 GPG 的支持。这样将有助于您安卓版本游戏在 PC 平台发布以获取更多的增长。

为了顺利在 GPG 上发布，我们建议您先阅读 GPG 官方网站 以便快速介入 GPG SDK。

接入指南

为了让您的应用可以顺利上架，请检查下列的步骤是否都已经设置。

1. 为了让游戏可以在 Windows 上运行（包括 intel 和 AMD 芯片），您需要采用 x86 架构进行构建。在 Cocos Creator 构建时，选中 APP ABI 并勾选 x86_64：

发布和运行

运行和启动

集成 Input SDK

集成 Input SDK

Input SDK提供了一个统一的界面，让玩家在 Google Play Games For PC 上方便地找到游戏的鼠标和键盘绑定，从而提升玩家的游戏体验。

准备工作

如果你要在一个全新的 Android 构建上使用 Input SDK，只需要在构建面板上勾选 Input SDK 选项。

如果这个 Anrdoid 工程是之前构建的，就需要手动在 文件 `native/engine/android/app/build.gradle` 中添加下面的代码：

```
dependencies {
  implementation 'com.google.android.libraries.play.games:inputmapping:1.0.0-beta'
  ...
}
```

添加 InputMapProvider 实现

新建类文件 `MyInputMapProvider.java`。

```
    public class MyInputMapProvider implements InputMappingProvider {
        public enum InputEventIds {
            JUMP,
            LEFT,
            RIGHT,
            USE,
            SPECIAL_JUMP,
            SPECIAL_DUCK,
            CMB_MOVE,
            CMB_DASH,
            CMB_WAYPOINT
        }

        @Override
        public InputMap onProvideInputMap() {
            InputAction jumpInputAction = InputAction.create(
                    getString(R.string.key_jump),
                    InputEventIds.JUMP.ordinal(),
                    InputControls.create(
                            Collections.singletonList(KeyEvent.KEYCODE_SPACE),
                            Collections.emptyList()
                    )
            );
            InputAction leftAction = InputAction.create(
                    getString(R.string.key_Left),
                    InputEventIds.LEFT.ordinal(),
                    InputControls.create(
                            Collections.singletonList(KeyEvent.KEYCODE_DPAD_LEFT),
                            Collections.emptyList()
                    )
            );
            InputAction rightAction = InputAction.create(
                    getString(R.string.key_Right),
                    InputEventIds.RIGHT.ordinal(),
                    InputControls.create(
                            Collections.singletonList(KeyEvent.KEYCODE_DPAD_RIGHT),
                            Collections.emptyList()
                    )
            );

            InputAction cmbMove = InputAction.create(
                    getString(R.string.key_Move),
                    InputEventIds.CMB_MOVE.ordinal(),
                    InputControls.create(
                            Collections.emptyList(),
                            Collections.singletonList(InputControls.MOUSE_RIGHT_CLICK)
                    )
            );
            InputAction cmbDash = InputAction.create(
                    getString(R.string.key_Dash),
                    InputEventIds.CMB_DASH.ordinal(),
                    InputControls.create(
                            Arrays.asList(KeyEvent.KEYCODE_SHIFT_LEFT, KeyEvent.KEYCODE_SPACE),
                            Collections.emptyList()
                    )
            );
            InputAction cmbWaypoint = InputAction.create(
                    getString(R.string.key_Waypoint),
                    InputEventIds.CMB_WAYPOINT.ordinal(),
                    InputControls.create(
                            Collections.singletonList(KeyEvent.KEYCODE_SHIFT_LEFT),
                            Collections.singletonList(InputControls.MOUSE_RIGHT_CLICK)
                    )
            );
            InputGroup movementInputGroup = InputGroup.create(
                    getString(R.string.key_BasicMove),
                    Arrays.asList(jumpInputAction, leftAction, rightAction, cmbMove, cmbDash, cmbWaypoint)
            );
            return InputMap.create(
                    Arrays.asList(movementInputGroup),
                    MouseSettings.create(true, true)
            );
        }
    }
```

添加按键的描述

在 `res\values\strings.xml` 中定义用于国际化的文本，以供前面步骤中使用。

```
<resources>
    ...
    <string name="key_jump">Jump</string>
    <string name="key_Left">Left</string>
    <string name="key_Right">Right</string>
    <string name="key_Move">Move</string>
    <string name="key_Dash">Dash</string>
    <string name="key_Waypoint">Add Waypoint</string>
    <string name="key_BasicMove">Basic Movement</string>
</resources>
```

修改 `AppActivity` 注册 `InputMapping`

`AppActivity.java` 位于 `native\engine\android\app\src\main\cocos\game`，需做如下修改。

```
    ...
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        ...

        InputMappingClient inputMappingClient = Input.getInputMappingClient(this);
        inputMappingClient.setInputMappingProvider(new MyInputMapProvider());
    }
    ...

    @Override
    protected void onDestroy() {
        InputMappingClient inputMappingClient = Input.getInputMappingClient(this);
        inputMappingClient.clearInputMappingProvider();

        super.onDestroy();
        ...
    }
```

测试效果

在 Google Play Games 提供的 HPE 模拟器中运行，通过 Shift + Tab 调出页面。

发布 iOS 应用

iOS 构建指引

Cocos Creator 支持发布为 iOS 应用，但至少需要一台支持 macOS 的设备以及苹果开发者账号。

主要内容

- iOS 构建示例
- iOS 构建选项介绍
- 基于反射机制实现 JavaScript 与 iOS/macOS 系统原生通信
- 使用 JsbBridge 实现 JavaScript 与 Objective-C 通信
- 特性与系统版本

相关阅读

- 构建发布面板
- 通用构建选项介绍
- 原生平台通用构建选项
- 安装配置原生开发环境
- 原生平台 JavaScript 调试
- 构建流程简介与常见错误处理
- 原生平台二次开发指南

---

iOS 发布示例

iOS 构建示例

本文将演示 Cocos Creator 项目发布为 iOS 应用程序的流程，需要以下准备工作：

- 一台安装了 XCode 的 MacOS 设备
- 一个苹果开发者账号

发布流程

注册开发者账号

首先，需要拥有一个苹果开发者账号，如果没有，请先前往 注册页面 进行注册。

确认 macOS 系统 和 Xcode 版本

Cocos Creator 打包环境要求：

- Xcode 版本为 **11.5** 及以上。
- macOS 版本为 **10.14** 及以上。

> **注意**：默认情况下 AppStore 中对应的 Xcode 与系统匹配，如果要使用特定版本的 Xcode，可前往 Xcode 下载页 下载。

准备测试项目

打开一个已有的项目，或者新建一个测试项目。

构建

---

iOS 构建选项

iOS 平台构建选项

---

iOS 微信小游戏内存与性能优化指南

iOS 微信小游戏内存与性能优化指南

前言

由于微信小游戏架构是基于 C++ 渲染层模拟 Canvas 渲染功能的实现方案，导致在 iOS 端的性能表现一直差强人意。 同样的程序版本，与有 JIT 加速的 Android 比起来，相差甚远。

微信小游戏提供的高性能模式，通过转用微信内部的 Webkit 运行游戏的方式，使得在 iOS 上的小游戏，也能拥有 JIT 能力，大幅度提升运行性能。

从微信小游戏官方文档的水族馆测试中，我们可以看在，同样的场景，在 iPhone11 Pro Max 上，高性能模式下达到了 49 FPS，普通模式下却只有 13 FPS。

发布 HUAWEI HarmonyOS 应用

发布 Huawei HarmonyOS 应用

从 v3.2 开始，Cocos Creator 支持将游戏打包为 HarmonyOS 应用（`.hap`）。

准备工作

- 进入 AppGallery Connect 网站 注册 华为开发者联盟帐号。
- 登录后 创建项目 并 添加 HarmonyOS 应用，**选择平台** 为 **APP（HarmonyOS 应用）**。

发布到 OpenHarmony 应用

发布到 OpenHarmony

自 Cocos Creator v3.8 起，支持发布到 OpenHarmony 平台。

支持情况

引擎版本	OpenHarmony 版本	说明
v3.6.1 ~ v3.7.3	OpenHarmony 3.2 beta	该版本在社区公测，因此无法从 Dashboard 直接下载，开发者请从 社区下载
v3.8	OpenHarmony 3.2，OpenHarmony 4.0 正式版，开发者可以选择从 Dashboard 中直接下载该编辑器版本。	

如果您使用的引擎版本低于 v3.8，我们建议您升级到 v3.8 方便发布正式版。

前言

我们在 Cocos Creator v3.8.0 的基础上成功适配了在 2023 年 4 月 9 日发布的 OpenHarmony 3.2 Release 分支。

鹰击长空：GitHub - cocos/cocos-tutorial-airplane: video tutorial airplane

游戏视频：

如无法观看视频，请点击 下载地址 下载后观看。

准备工作

1. Cocos Creator 下载传送门（版本>=3.8.0）：Cocos Creator
2. OpenHarmany 系统与 SDK 下载传送门：OpenHarmony-3.2-release

发布 macOS 应用

macOS 构建指引

Cocos Creator 支持发布 macOS 桌面应用程序，但至少需要一台支持 macOS 的设备以及苹果开发者账号。

主要内容

- macOS 构建示例
- macOS 构建选项介绍
- 基于反射机制实现 JavaScript 与 iOS/macOS 系统原生通信
- 使用 JsbBridge 实现 JavaScript 与 Objective-C 通信
- 特性与系统版本

相关阅读

- 构建发布面板
- 通用构建选项介绍
- 原生平台通用构建选项
- 安装配置原生开发环境
- 原生平台 JavaScript 调试
- 构建流程简介与常见错误处理
- 原生平台二次开发指南

---

macOS 发布示例

macOS 构建示例

本文将演示 Cocos Creator 项目发布为 macOS 应用程序的流程，需要以下准备工作：

- 一台 安装了 XCode 的 MacOS 设备
- 一个苹果开发者账号

发布流程

注册开发者账号

首先，需要拥有一个苹果开发者账号，如果没有，请先前往 注册页面 进行注册。

确认 macOS 系统 和 Xcode 版本

Cocos Creator 打包环境要求：

- Xcode 版本为 **11.5** 及以上。
- macOS 版本为 **10.14** 及以上。

> **注意**：默认情况下 AppStore 中对应的 Xcode 与系统匹配，如果要使用特定版本的 Xcode，可前往 Xcode 下载页 下载。

准备测试项目

打开一个已有的项目，或者新建一个测试项目

构建

---

macOS 构建选项

Mac 平台构建选项

发布 Windows 应用

Windows 构建指引

Cocos Creator 支持发布 Windows 桌面应用程序，但至少需要一台支持 Windows 的设备。

主要内容

- Windows 构建示例
- Windows 构建选项介绍

相关阅读

- 构建发布面板
- 通用构建选项介绍
- 原生平台通用构建选项
- 安装配置原生开发环境
- 原生平台 JavaScript 调试
- 构建流程简介与常见错误处理
- 原生平台二次开发指南

Windows 构建示例

Windows 构建示例

本文将演示将 Cocos Creator 开发的项目发布为 Windows 应用程序，需要以下准备工作：

- 一台 Windows 电脑
- C++ 开发环境

安装 C++ 编译环境

Windows 下需要安装 Visual Studio 2019/2022。

在安装 Visual Studio 时，请勾选 **使用 C++ 的桌面开发** 和 **使用 C++ 的游戏开发** 两个模块。

> **注意**：在 **使用 C++ 的游戏开发** 模块中有一个 **Cocos** 选项，请勿勾选。

发布流程

准备测试项目

打开一个已有的项目，或者新建一个测试项目

构建

Windows 构建选项

Windows 平台构建选项

Windows 平台的构建选项包括 **渲染后端** 和 **生成平台**。

原生平台发布通用基础

原生平台发布通用基础

Cocos Creator 支持发布到多个平台的原生应用程序：

- 发布 iOS 应用程序
- 发布 Android 应用程序
- 发布 HUAWEI Harmony 应用程序
- 发布 macOS 桌面端应用
- 发布 Windows 桌面端应用

以下是发布到原生平台时可能涉及到的通用知识点：

- 安装配置原生环境
- 构建发布面板
- 通用构建选项介绍
- 原生平台通用构建选项
- 原生平台 JavaScript 调试
- 构建流程简介与常见错误处理

如果要发布到 小游戏 和 Web 平台，请参考对应文档。

如果要对原生平台进行二次开发，请参考：原生开发指南。

如果要对引擎进行定制，请参考：引擎定制工作流。

原生平台通用构建选项

打包发布到原生平台

点击菜单栏的 **项目 -> 构建发布**，打开构建发布面板。

目前可以选择的原生平台包括 Android、iOS、HarmonyOS、Mac 和 Windows 四个。iOS、Mac 只在 macOS 电脑上才能选择，Windows 只在 Windows 电脑上才能选择。

安装配置原生环境

安装配置原生开发环境

Cocos Creator 可以跨平台发布多个平台的原生应用，在使用 Cocos Creator 打包发布到原生平台之前，我们需要先配置好相关的原生开发环境。

本文档提供了特定的开发环境要求，我们对这些指定环境的运行进行了验证和保证。然而，请注意，我们无法确保在除指定环境之外的其他环境下能够完全运行。如果您在非指定环境下遇到任何问题，请不要犹豫，欢迎您到我们的论坛或支持渠道进行反馈。我们的社区将竭诚帮助您解决问题，并收集相关反馈以改进我们的文档和工具。

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。如果不需要发布到 Android 平台或者操作系统上已经有完整的 Android 开发环境，可以跳过这部分内容。

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：

JDK Development Kit 17.0.7 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。

安装后请检查 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
```

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。

如果是 Windows 系统，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。修改完成后 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

下载安装 Android Studio

Cocos Creator 需要使用 **Android Studio Flamingo | 2022.2.1** 作为安卓平台的构建工具，并在 Android Studio 中下载所需的 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，推荐选择主流 API Level 26（8.0）、API Level 28（9.0）等。
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，选择最新的 build tools 版本。
4. 勾选 **Android SDK Platform-Tools** 和 **CMake**，如需安装 Android 支持库，请参考 官方文档 — 支持库设置。
5. 勾选 **NDK**，推荐使用版本为 **r21~23**。
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 编辑器中填写这个 SDK 所在位置。
7. 点击 **OK**，根据提示完成安装。

原生平台 JavaScript 调试

原生平台 JavaScript 调试

游戏发布到原生平台后，由于运行环境不同，可能会出现在浏览器预览时无法重现的 Bug，这时我们就必须直接在原生平台下进行调试。Cocos Creator 可以很方便地对原生平台中的 JavaScript 进行远程调试。

iOS 和 Android 真机调试

如果游戏只有在真机上才能运行，那就必须用真机对打包后的游戏进行调试。调试步骤如下：

- 确保 Android/iOS 设备与 Windows 或者 Mac 在同一个局域网中。注意在调试过程中请勿开启代理，否则可能导致无法正常调试。
- 在 Creator 的 **构建发布** 面板选择 Android/iOS 平台、Debug 模式，构建编译运行工程（iOS 平台建议通过 Xcode 连接真机进行编译运行）。
- 用 Chrome 浏览器打开地址：`devtools://devtools/bundled/js_app.html?v8only=true&ws=设备的本地 IP:43086/00010002-0003-4004-8005-000600070008` 即可进行调试。

集成 Input SDK

集成 Input SDK

Input SDK提供了一个统一的界面，让玩家在 Google Play Games For PC 上方便地找到游戏的鼠标和键盘绑定，从而提升玩家的游戏体验。

准备工作

如果你要在一个全新的 Android 构建上使用 Input SDK，只需要在构建面板上勾选 Input SDK 选项。

如果这个 Anrdoid 工程是之前构建的，就需要手动在 文件 `native/engine/android/app/build.gradle` 中添加下面的代码：

```
dependencies {
  implementation 'com.google.android.libraries.play.games:inputmapping:1.0.0-beta'
  ...
}
```

添加 InputMapProvider 实现

新建类文件 `MyInputMapProvider.java`。

```
    public class MyInputMapProvider implements InputMappingProvider {
        public enum InputEventIds {
            JUMP,
            LEFT,
            RIGHT,
```

```
            USE,
            SPECIAL_JUMP,
            SPECIAL_DUCK,
            CMB_MOVE,
            CMB_DASH,
            CMB_WAYPOINT
        }

        @Override
        public InputMap onProvideInputMap() {
            InputAction jumpInputAction = InputAction.create(
                    getString(R.string.key_jump),
                    InputEventIds.JUMP.ordinal(),
                    InputControls.create(
                            Collections.singletonList(KeyEvent.KEYCODE_SPACE),
                            Collections.emptyList()
                    )
            );
            InputAction leftAction = InputAction.create(
                    getString(R.string.key_Left),
                    InputEventIds.LEFT.ordinal(),
                    InputControls.create(
                            Collections.singletonList(KeyEvent.KEYCODE_DPAD_LEFT),
                            Collections.emptyList()
                    )
            );
            InputAction rightAction = InputAction.create(
                    getString(R.string.key_Right),
                    InputEventIds.RIGHT.ordinal(),
                    InputControls.create(
                            Collections.singletonList(KeyEvent.KEYCODE_DPAD_RIGHT),
                            Collections.emptyList()
                    )
            );

            InputAction cmbMove = InputAction.create(
                    getString(R.string.key_Move),
                    InputEventIds.CMB_MOVE.ordinal(),
                    InputControls.create(
                            Collections.emptyList(),
                            Collections.singletonList(InputControls.MOUSE_RIGHT_CLICK)
                    )
            );
            InputAction cmbDash = InputAction.create(
                    getString(R.string.key_Dash),
                    InputEventIds.CMB_DASH.ordinal(),
                    InputControls.create(
                            Arrays.asList(KeyEvent.KEYCODE_SHIFT_LEFT, KeyEvent.KEYCODE_SPACE),
                            Collections.emptyList()
                    )
            );
            InputAction cmbWaypoint = InputAction.create(
                    getString(R.string.key_Waypoint),
                    InputEventIds.CMB_WAYPOINT.ordinal(),
                    InputControls.create(
                            Collections.singletonList(KeyEvent.KEYCODE_SHIFT_LEFT),
                            Collections.singletonList(InputControls.MOUSE_RIGHT_CLICK)
                    )
            );
            InputGroup movementInputGroup = InputGroup.create(
                    getString(R.string.key_BasicMove),
                    Arrays.asList(jumpInputAction, leftAction, rightAction, cmbMove, cmbDash, cmbWaypoint)
            );
            return InputMap.create(
                    Arrays.asList(movementInputGroup),
                    MouseSettings.create(true, true)
            );
        }
    }
```

添加按键的描述

在 `res\values\strings.xml` 中定义用于国际化的文本，以供前面步骤中使用。

```
<resources>
    ...
    <string name="key_jump">Jump</string>
    <string name="key_Left">Left</string>
    <string name="key_Right">Right</string>
    <string name="key_Move">Move</string>
    <string name="key_Dash">Dash</string>
    <string name="key_Waypoint">Add Waypoint</string>
    <string name="key_BasicMove">Basic Movement</string>
</resources>
```

修改 `AppActivity` 注册 `InputMapping`

`AppActivity.java` 位于 `native\engine\android\app\src\main\cocos\game`，需做如下修改。

```
    ...
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        ...

        InputMappingClient inputMappingClient = Input.getInputMappingClient(this);
        inputMappingClient.setInputMappingProvider(new MyInputMapProvider());
    }
    ...

    @Override
    protected void onDestroy() {
        InputMappingClient inputMappingClient = Input.getInputMappingClient(this);
        inputMappingClient.clearInputMappingProvider();

        super.onDestroy();
        ...
    }
```

测试效果

在 Google Play Games 提供的 HPE 模拟器中运行，通过 Shift + Tab 调出页面。

发布到小游戏平台

发布到小游戏平台

发布到 HUAWEI AppGallery Connect

发布到 HUAWEI AppGallery Connect

Cocos Creator 支持将游戏发布到 **HUAWEI AppGallery Connect**，帮助开发者接入到华为的应用市场。

准备工作

- 进入 AppGallery Connect 后台 并登录，需要先完成 开发者注册，然后再 创建应用。创建应用时，**软件包类型** 选择 **APK**。

发布到支付宝小游戏

发布到支付宝小游戏

环境配置

- 桌面端下载 支付宝小程序开发者工具 并安装。
- 下载 支付宝，并安装到手机设备上。
- 支付宝客户端在 Android 上支持的最低版本为 10.3.70，在 iOS 为 10.3.70。

发布流程

使用 Cocos Creator 打开需要发布的项目工程，在 **构建发布** 面板的 **发布平台** 中选择 **支付宝小游戏**，然后点击 **构建**。

发布到淘宝小游戏

发布到淘宝小游戏

环境配置

- 桌面端下载 淘宝开发者工具 并安装。
- 下载 淘宝，并安装到手机设备上。
- 淘宝客户端在 Android 上支持的最低版本为 10.22.30，在 iOS 为 10.22.30。

发布流程

使用 Cocos Creator 打开需要发布的项目工程，在 **构建发布** 面板的 **发布平台** 中选择 **淘宝小游戏**，然后点击 **构建**。

发布到微信小游戏

发布到微信小游戏

微信小游戏的运行环境是微信小程序环境的扩展，在小程序环境的基础上提供了 WebGL 接口的封装，使得渲染能力和性能有了大幅度提升。不过由于这些接口都是微信团队通过自研的原生实现封装的，所以并不可以等同为浏览器环境。

作为引擎方，为了尽可能简化开发者的工作量，我们为用户完成的主要工作包括：

- 引擎框架适配微信小游戏 API，纯游戏逻辑层面，用户不需要任何额外的修改
- Cocos Creator 编辑器提供了快捷的打包流程，直接发布为微信小游戏，并自动唤起小游戏的开发者工具
- 自动加载远程资源，缓存资源以及缓存资源版本控制

除此之外，小游戏的游戏提交、审核以及发布流程，和小程序是没有区别的，都需要遵守微信团队的要求和标准流程，具体信息可以参考 微信小游戏开发文档。

环境配置

1. 在 微信官方文档 下载微信开发者工具。
2. 在编辑器主菜单的 **Cocos Creator/File -> 偏好设置 -> 外部程序** 中设置微信开发者工具路径。

启用微信小游戏引擎插件

微信小游戏引擎插件使用说明

游戏引擎插件是微信 v7.0.7 新增的一项功能。此插件内置了 Cocos Creator 引擎的官方版本，若玩家首次体验的游戏中启用了此插件，则所有同样启用此插件的游戏，都无需再次下载 Cocos Creator 引擎，只需直接使用公共插件库中的相同版本引擎，或者增量更新引擎即可。

例如，当一个玩家玩过了由 Cocos Creator v2.2.0 开发的 A 游戏，里面已启用了此插件。然后他又玩了同样是 v2.2.0 开发的 B 游戏，如果 B 游戏也启用了此插件，那么就无需重新下载 Cocos Creator 引擎。即使 B 游戏使用的是 v2.2.1 的 Cocos Creator，微信也只需要增量更新引擎两个版本的差异部分。这样就可以大幅减少小游戏的下载量，提升小游戏启动速度 0.5 ~ 2s，获得更好的用户体验。

使用说明

Cocos Creator 只需在 **构建发布** 面板中，勾选 **分离引擎** 选项，然后正常构建发布即可，无需其它人工操作。（此功能仅在编辑器使用内置引擎并且构建时使用 **非调试模式** 时生效）

接入微信 PC 小游戏

接入微信 PC 小游戏

微信 PC 小游戏即支持在微信 PC 版打开微信小游戏。PC 小游戏将具备移动端的大部分能力，包括但不限于虚拟支付、开放数据域、触摸事件等（广告目前暂不支持）。同时 PC 小游戏还支持键盘、鼠标事件及自定义窗口等功能。

Cocos Creator 支持将游戏发布到微信 PC 小游戏，并完成了鼠标、键盘相关接口的适配工作。下面我们来看看，如何通过 Cocos Creator 将游戏发布到微信 PC 小游戏平台。

使用 Cocos Creator 接入微信 PC 小游戏

准备工作

下载并安装最新版本的 微信 PC 版，使用微信开发者工具绑定的微信号登录微信 PC 版。

发布流程

1. 参考 发布到微信小游戏 的流程，将项目工程发布到微信小游戏。
2. 在 **微信开发者工具** 中，点击上方工具栏中的 **预览** 按钮，选择 **自动预览** 选项卡，勾选 **启动 PC 端自动预览**，然后点击 **编译并预览**，即可在微信 PC 版预览并调试小游戏。

发布到抖音小游戏

发布到抖音小游戏

抖音小游戏是基于一种不需要用户下载，点开即玩的全新游戏类型。

小游戏的游戏提交，审核和发布流程等，需要遵守字节官方团队的要求和标准流程，具体信息可以参考 抖音小游戏接入指南。

准备工作

1. 下载 抖音开发者工具 并安装。
2. 参考 抖音小游戏接入指南，在 抖音开发者平台 完成账号注册、登录以及申请小游戏。
3. 小游戏申请通过后，在开发者平台的 **开发管理 -> 开发设置** 中找到 **AppID**。

发布到华为快游戏

发布到华为快游戏

环境配置

- 下载 华为快应用加载器，并安装到华为手机上（建议 Android Phone 6.0 或以上版本）
- PC 端全局安装 nodejs-8.1.4 或以上版本

发布流程

使用 Cocos Creator 打开需要发布的项目工程，从 **菜单栏 -> 项目** 中打开 **构建发布** 面板。在 **构建发布** 面板的 **发布平台** 中选择 **华为快游戏**。

发布到 OPPO 小游戏

发布到 OPPO 小游戏

环境配置

- 下载 OPPO 小游戏调试器，并安装到 OPPO 手机上（建议 Android Phone 6.0 或以上版本）
- 全局安装 nodejs-8.1.4 或以上版本

发布流程

使用 Cocos Creator 打开需要发布的项目工程，从 **菜单栏 -> 项目** 中打开 **构建发布** 面板，**发布平台** 选择 **OPPO 小游戏**。

发布到 vivo 小游戏

发布到 vivo 小游戏

环境配置

- 下载 快应用 & vivo 小游戏调试器 和 vivo 小游戏引擎，并安装到 Android 设备上（建议 Android Phone 6.0 或以上版本）
- 全局安装 nodejs-8.9.0 或以上版本

  > **注意**：安装 nodejs 后，需要注意 npm 源地址是否为 https://registry.npmjs.org/

  ```
  # 查看当前 npm 源地址
  npm config get registry
  # 若不是，重新设置 npm 源地址
  npm config set registry https://registry.npmjs.org/
  ```

- 全局安装 `vivo-minigame/cli`。确定 npm 源地址后，安装 `vivo-minigame/cli`：

  ```
  npm install -g @vivo-minigame/cli
  ```

  若 `vivo-minigame/cli` 安装失败，可能是因为 nodejs 版本过低导致的，请检查 node 版本并升级。

发布流程

使用 Cocos Creator 打开需要发布的项目工程，从 **菜单栏 -> 项目** 中打开 **构建发布** 面板，**发布平台** 选择 **vivo 小游戏**。

发布到小米快游戏

发布到小米快游戏

环境配置

- 桌面端全局安装 Node.js 8.1.4 或以上版本。

- 确保 Node.js 所带的 npm 版本最低是 **5.2.0**。升级 npm 命令如下：

```
# 查看 npm 版本
npm -v
# 若 npm 版本在 5.2.0 以下，可使用以下升级命令升级 npm
npm install npm@latest -g
```

- 下载 小米快游戏调试器和小米快游戏运行环境，并安装到小米手机上（MIUI 8.5.0 或以上版本）。

发布流程

使用 Cocos Creator 打开需要发布的项目工程，从 **菜单栏 -> 项目** 中打开 **构建发布** 面板，**发布平台** 选择 **小米快游戏**。

发布到百度小游戏

发布到百度小游戏

> **注意**：由于相关合约已到期，自 2022 年 11 月 30 日起，将不再支持百度小游戏的发布和构建。

百度小游戏是基于手机百度 app 上的智能小程序进行扩展的小游戏，它不仅提供了强大的游戏能力，还和智能小程序一样，提供了大量的原生接口，比如支付，文件系统，位置，分享等。相当于同时结合了 WEB 易于传播以及 Native 功能丰富的优势。

百度小游戏的运行环境和微信小游戏类似，基本思路也是封装必要的 WEB 接口提供给用户，尽可能追求和 WEB 同样的开发体验。百度小游戏在智能小程序环境的基础上提供了 WebGL 接口的封装，使得渲染能力和性能有了大幅度提升。不过由于这些接口都是百度团队通过自研的原生实现封装的，所以并不可以等同为浏览器环境。

作为引擎方，为了尽可能简化开发者的工作量，我们为用户完成的主要工作包括：

- 引擎框架适配百度小游戏 API，纯游戏逻辑层面，用户不需要任何额外的修改
- Cocos Creator编辑器提供了快捷的打包流程，直接发布为百度小游戏
- 自动加载远程资源，缓存资源以及缓存资源版本控制

具体百度小游戏的申请入驻，开发准备，游戏提交，审核和发布流程可以参考 百度小游戏注册指导文档。

准备工作

- 下载 百度开发者工具 并安装。
- 在手机的应用商店中下载并安装百度应用

- 登录 智能小程序平台，找到 AppID。

开放数据域

开放数据域

目前，**微信**、**百度** 和 **抖音** 小游戏这些平台为了保护其社交关系链数据，增加了 **开放数据域** 的概念，这是一个单独的游戏执行环境。开放数据域中的资源、引擎、程序，都和主游戏（主域）完全隔离，开发者只有在开放数据域中才能通过平台提供的开放接口来访问关系链数据，用于实现一些例如排行榜的功能。

开放数据域目前只支持 Canvas 渲染，在 Cocos Creator 3.0 中，我们废弃了之前的 Canvas Renderer 模块，使用微信团队基于 **XML** + **CSS** 设计研发的一个前端轻量级 Canvas 引擎来替代。并且将该引擎整合进了 Creator 3.0 内置的开放数据域工程模板中，开发者只需要掌握一些基本的前端技能，就能在该模板的基础上定制排行榜功能。

SubContextView 组件说明

由于开放数据域只能在离屏画布 sharedCanvas 上渲染，所以在项目中，需要有一个节点作为渲染开放数据域的容器，并在该节点上添加 **SubContextView** 组件，该组件会将 sharedCanvas 渲染到容器节点上。

SubContextView 组件主要包含 **设计分辨率** 和 **FPS** 两个属性。

小游戏分包

小游戏分包

部分小游戏平台支持分包功能以便对资源、脚本和场景进行划分，包括微信小游戏、百度小游戏、小米快游戏、抖音小游戏、华为快游戏、OPPO 小游戏和 vivo 小游戏。

Cocos Creator 从 v2.4 开始支持 Asset Bundle，开发者可以将需要分包的内容划分成多个 Asset Bundle，这些 Asset Bundle 会被构建成小游戏的分包。在启动游戏时只会下载必要的主包，不会加载这些分包，而是由开发者在游戏过程中手动加载分包，从而有效降低游戏启动的时间。

配置方法

Asset Bundle 是以 **文件夹** 为单位进行配置的。当我们在 **资源管理器** 中选中一个文件夹时，**属性检查器** 中就会出现一个 **配置为 Bundle** 的选项，勾选后会出现如下图的配置项：

发布到 Facebook Instant Games 平台

发布游戏到 Facebook Instant Games

Facebook Instant Games 跟微信小游戏相比，本质上的区别在于 Facebook Instant Games 运行于纯 HTML5 环境。因此它不仅可以在手机上，还可以在桌面浏览器上运行，开发和调试更加便捷。

目前 Cocos Creator 为用户完成的工作包括：

- 集成了 Facebook Instant Games SDK，并且自动进行了初始化，用户能直接调用相关的 API
- 在 Cocos Creator 构建面板中提供一键式打包流程，可直接打包为符合 Facebook Instant Games 技术规范的游戏

用户需要完成的工作：

- 调用 Facebook Instant Games SDK，访问平台相关功能
- 将 Cocos Creator 打包好的版本上传到 Facebook

发布流程

- 使用 Cocos Creator 构建游戏
- 上传到 Facebook 后台
- 测试游戏
- 在 Facebook 中分享你的游戏

使用 Cocos Creator 构建游戏

1. 使用 Cocos Creator 打开需要发布的项目工程，从 **菜单栏 -> 项目** 中打开 **构建发布** 面板。在 **构建发布** 面板的 **发布平台** 中选择 **Facebook Instant Games**。

发布到 Web 平台

发布到 Web 平台

打开主菜单的 **项目 -> 构建发布**，打开 构建发布 面板。

通用构建选项介绍

构建选项介绍

通用构建选项

构建发布 面板中的通用构建参数默认是按照平台区分的，如下：

擦除模块结构（实验性质）

若勾选该项，脚本导入速度更快，但无法使用模块特性，例如 `import.meta`、`import()` 等。

跳过纹理压缩流程

默认为 `false`，若勾选该选项，则构建时会跳过整个纹理压缩流程，以减少构建时间。构建命令为：`skipCompressTexture`。

引擎模块配置

自 3.8.3 起，允许在构建面板上配置部分引擎模块配置，方便针对不同的平台或构建任务选择不同的物理后端等。

命令行发布项目

命令行发布项目

命令行发布项目可以帮助开发者构建自己的自动化构建流程，通过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

例如：构建 web-desktop 平台、Debug 模式

- Mac

```
/Applications/CocosCreator/Creator/3.0.0/CocosCreator.app/Contents/MacOS/CocosCreator --project projectPath
--build "platform=web-desktop;debug=true"
```

- Windows

```
...\CocosCreator.exe --project projectPath --build "platform=web-desktop;debug=true"
```

目前命令行构建除了必填项外，如果不传递一律使用默认值来构建，具体参数默认值请参考下方描述以及平台的参数介绍。

进程退出码

- **32** 构建失败 —— 构建参数不合法
- **34** 构建失败 —— 构建过程出错失败，详情请参考构建日志
- **36** 构建成功

构建参数

- `--project`：必填，指定项目路径
- `--engine`：选填，指定自定义引擎路径
- `--build`：指定构建项目使用的参数

  在 `--build` 后如果没有指定参数，则会使用 Cocos Creator 中 **构建发布** 面板当前的平台、模板等设置来作为默认参数。如果指定了其他参数设置，则会使用指定的参数来覆盖默认参数。可选择的参数有：

  - `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。这个参数可以自己修改也可以直接从构建面板导出，当配置和 configPath 内的配置冲突时，configPath 指定的配置将会被覆盖。
  - `stage` - 指定构建模式，默认为 'build'，可选 'make' | 'build' | 'bundle' 等
  - `logDest` - 指定日志输出路径
  - `includedModules` - 定制引擎打包功能模块，只打包需要的功能模块。具体有哪些功能模块可以参考引擎仓库根目录下 **cc.config.json**（GitHub | Gitee）文件中的 `features` 字段。
  - `outputName` - 构建后生成的发布包文件夹名称。
  - `name` - 游戏名称
  - `platform` - 必填，构建的平台，具体名称参考面板上对应扩展名称即可
  - `buildPath` - 指定构建发布包生成的目录，默认为项目目录下的 `build` 目录。可使用绝对路径或者相对于项目的路径（例如 `project://release`）。从 v3.4.2 开始支持类似 `../` 这样的相对路径。
  - `startScene` - 主场景的 UUID 值（参与构建的场景将使用上一次的编辑器中的构建设置），未指定时将使用参与构建场景的第一个
  - `scenes` - 参与构建的场景信息，未指定时默认为全部场景，具体格式为：`{}`
  - `debug` - 是否为 debug 模式，默认关闭
  - `replaceSplashScreen` - 是否替换插屏，默认关闭
  - `md5Cache` - 是否开启 md5 缓存，默认关闭
  - `mainBundleCompressionType` - 主包压缩类型，具体选项值可参考文档 Asset Bundle — 压缩类型。
  - `mainBundleIsRemote` - 配置主包为远程包
  - `packages` - 各个扩展支持的构建配置参数，需要存放的是对于数据对象的序列化字符串，具体可以参考下文。

Cocos Creator 3.0 各个平台的构建会作为独立的扩展嵌入到 **构建发布** 面板中，因而各个平台的构建参数位置也不同。各个平台的构建参数会配置在 `packages` 字段中，例如：为微信小游戏指定构建参数，配置大体如下：

```
{
    taskName: 'wechatgame',
    packages: {
        wechatgame: {
            appid: '*****',
        }
    }
}
```

之后在构建扩展支持对外开放，其他扩展的配置参数也会通过同样的方式嵌入到 **构建发布** 面板中。具体各个平台的参数字段 **请参照各个平台文档中的参数介绍**，最好是通过 **构建发布** 面板的 **导出** 功能来获取配置参数，更加方便快捷。目前依旧兼容旧版本的参数进行构建，但之后将会移除该兼容处理，请尽快升级配置参数。

命令行执行独立发布 Bundle

1. 打开 Bundle 构建面板，配置好选项后，导出配置(自 3.8.2 起)

定制项目的构建模版

自定义构建模版

Cocos Creator 支持对每个项目分别定制构建模板，只需要在项目路径下添加一个 `build-templates` 目录，里面按照 **平台扩展名称** 划分子目录。在构建结束后，`build-templates/[platform]` 目录下所有的文件都会自动按照对应的目录结构 **复制** 到对应平台构建生成的工程中，这个功能是全平台支持的。具体的 **平台扩展名称** 请参考最下方的 **特殊自定义构建模板平台支持表**。

结构类似：

```
project-folder
 |--assets
 |--build
 |--build-templates
     |--web-mobile
         // 需要添加的文件，如 index.html
         |--index.html
```

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（以平台扩展名称为准）/index.html`。

除此之外，目前构建模板支持的文件类型还包括 **ejs 类型** 和 **json 类型**，这两个类型不会直接拷贝而是会过解析处理，或者数据融合。各平台对这两种模板类型的支持情况，详情请参考下文的 **特殊自定义构建模板平台支持表**。

ejs 类型

随着 Cocos Creator 版本的升级，可能会对构建模板做一些修改和更新，就会导致不同版本构建出来的包内容不完全一样，开发者需要手动同步更新项目中定制的构建模板。例如构建时勾选了 MD5 Cache 选项之后，以 Web 平台的 `index.html` 为例，里面引用的 `css` 文件地址会带有 MD5 Hash 后缀，可能会和原先模板里的不匹配而导致无法使用。
因此为了优化这个问题，Creator 在主菜单的 **项目** 中新增了 **创建项目构建模板** 选项，用于生成对应平台支持的构建模板。

构建流程简介与常见错误处理

构建流程简介与常见问题指南

构建基础结构介绍

构建流程主要包括以下两部分内容：

- **通用构建处理**
- **各平台构建处理**

由于 v3.0 在构建机制上的调整，不同平台的构建处理均以 **构建扩展** 的形式注入 **构建发布** 面板，以及参与构建流程。各平台特有的构建选项也会以展开选项的形式显示在 **构建发布** 面板，开发者可以通过 构建扩展 将自定义的构建选项显示在 **构建发布** 面板上。

引擎编译失败

如果引擎编译失败，请检查安装包是否完整、修改的内置引擎代码是否正确，以及若使用了自定义引擎，路径是否正确等等。

其他报错

如果遇到的错误无法自行解决，请附上 Creator 版本、构建选项配置、构建任务中的构建日志文件以及可复现问题的 Demo 到 论坛 反馈。

图形渲染

图形渲染

Cocos Creator 提供了以下图形功能，用以丰富图形以及加强图形画面的真实性等：

- 渲染管线
- 相机
- 光照和阴影
- 网格
- 纹理
- 材质
- 着色器
- 粒子
- 特效
- 天空盒
- 全局雾
- 几何体渲染器
- 原生调试渲染器

渲染管线

渲染管线概述 (Experimental)

RenderPipeline 用于控制场景的渲染流程，包括光照管理、物体剔除、渲染物体排序、渲染目标切换等。由于每个阶段对于不同项目来说可以有不同的优化处理方式，所以用统一的方法来处理不同类型项目的渲染流程很难达到最优化的结果。可定制化的渲染管线用于对渲染场景中的每个阶段进行更灵活的控制，可以针对不同的项目做更深层次的优化方案。

在可定制化的渲染管线中，可以选择使用引擎内置的渲染管线，内置渲染管线包括 **前向渲染管线** 和 **延迟渲染管线**，引擎默认使用前向渲染管线。详情请参考 内置渲染管线。

内置管线

内置渲染管线

Cocos Creator 3.1 的内置渲染管线包括 **builtin-forward**（前向渲染管线）和 **builtin-deferred**（延迟渲染管线）。渲染管线可通过编辑器主菜单中的 **项目 -> 项目设置 -> 项目数据 -> 渲染管线** 进行设置，设置完成之后 **重启编辑器** 即可生效。

自定义渲染管线（实验性质）

自定义渲染管线

Cocos Creator 3.8 中正式开放了新的 **自定义渲染管线**。

自定义渲染管线 的接口位于 `cocos/core/pipeline/custom/pipeline.ts`

介绍

对于游戏引擎来说，渲染是最重要的功能之一。而引擎的渲染效果，由 **渲染管线**（RenderPipeline）决定，取决于不同的艺术风格、运行平台、硬件设备、渲染技术等等。

如何在引擎中实现各种各样的画面表现，是一个非常复杂的问题。

这需要引擎提供足够的灵活性，让用户可以自由定制渲染管线，以实现各种各样的效果。

Cocos Creator **自定义渲染管线** 能够在不同的平台、不同的硬件设备上，编写最优的渲染管线，以达到最佳的画面表现。

也能够在不同的平台、不同的硬件设备上，编写最通用的渲染管线，以达到最佳的性能表现以及跨平台性。

启用自定义管线

勾选 **自定义渲染管线**。

后期处理

全屏特效后处理流程

全屏特效后处理简称全屏后效、后效等，指的是在相机绘制完成整个画面后，对画面再次进行一次或者多次的图像处理。从而获取更好的画面或者实现某些特殊的效果。

Cocos Creator 在 v3.8 版本新加了全屏特效后处理流程，并且内置了一系列常用的效果。

开启后效流程

开启自定义管线并切换至**前向渲染管线**，如图所示：

PostProcess 组件

PostProcess 组件

Post-Process 组件是 后效 的辅助组件，提供后效的配置。在任何后效组件被添加后，该组件会自动添加到节点上。

属性

BlitScreen 组件

BlitScreen 组件

BlitScreen 组件允许用户定义不同的材质来管理 自定义后效。

属性

自定义后效

自定义后效

自定义后效有两种方式，简单的后效可以直接将后效材质添加到 Blit-Screen 后效组件 上，复杂的后效需要自定义一个后效 pass 。

Blit-Screen 后效组件

参考 设置后效流程 添加 Blit-Screen 后效组件， 将自定义后效材质拖入 **Material** 属性中，Blit-Screen 会按照 **Materials** 数组顺序依次渲染后效材质。

Materials 属性中每个自定义后效材质都支持单独开关，方便开发者管理。

相机

相机

游戏中的相机是用来捕捉场景画面的主要工具。我们通过调节相机相关参数来控制可视范围的大小，在 Cocos Creator 编辑器中相机呈如下表示：

光照

光照

本章主要介绍 Cocos Creator 中光照的工作方式和使用方式。

光照是指光的照射，Creator 中光照的实现模拟了光对真实世界的影响。在场景中添加光源可以使场景产生相应的光照和阴影效果，获得更好的视觉效果。

基于物理的光照

基于物理的光照

Cocos Creator 中采用光学度量单位来描述光源参数。基于光学度量单位，我们可以将光源的相关参数全部转化为真实世界中的物理值。这样，设计人员可根据相关灯光的工业参数以及真实环境的实际物理参数来调节光照强度、颜色、范围等信息，使整体光照效果更加符合真实的自然环境。

光源

光源

光源决定了物体所受到的光照的颜色、色温、强度、方向、以及产生的阴影效果等。目前 Creator 支持的光源类型包括：

- 平行光
- 球面光
- 聚光灯
- 环境光

添加光源

添加光源有以下两种方式：

1. 在 **层级管理器** 中点击左上角的 + 按钮，选择 **光源**，然后根据需要选择光源类型就可以创建一个带有对应类型 **光源组件** 的节点到场景中。

平行光

平行光

平行光又称为方向光（Directional Light），是最常用的一种光源，模拟了无限远处的光源发出的光线，常用于实现太阳光。

球面光

球面光

Cocos Creator 3.x 的球面光与 v2.x 的点光源（Point Light）类似。

球面光会向所有方向均匀地发散光线，接近于蜡烛产生的光线。物体受到的光照强度会随着跟光源距离的增大而减弱，当距离超过设置的光照影响范围，则光照强度为 0。

在实际应用中可用于模拟火把、蜡烛、灯泡等光源，照亮四周一定距离内的环境。

聚光灯

聚光灯

聚光灯 是由一个点向一个方向发射一束锥形光线，类似于手电筒或舞台照明灯产生的光线。与其他光源相比，聚光灯多了 `SpotAngle` 属性，用于调整聚光灯的光照范围。

环境光

环境光

在生活中，错综复杂的光线与凹凸不平的物体表面相互反射，使得整个环境都被照亮，仿佛被一层光均匀笼罩，这个光一般称为 **环境光**，也称为 **漫射环境光**。

因为环境光可以均匀地照亮场景中的所有物体，常用于解决模型背光面全黑的问题，一般需要配合其他类型的光源一起使用。例如场景中只有一个平行光，那么在模型的背光源处会显得非常暗，加入环境光则可以提升模型背部的亮度，显得更加美观。

基于多 Pass 的多光源支持

基于多 Pass 的多光源支持

超着色器（Uber Shader）目前在一些性能受限的平台上仍然是主流方案，但随着硬件性能的增强和画质需求的提高，固定数量的光源再也无法满足实际应用的需求，于是就有了支持多光源的方案 — **多遍绘制**。

下面以 Creator 中默认的光照材质 `default-material.mtl` 为例，介绍如何实现基于多 Pass 的多光源支持。

阴影

阴影

在 3D 世界中，光与影一直都是极其重要的组成部分，它们能够丰富整个环境，质量好的阴影可以达到以假乱真的效果，并且使得整个世界具有立体感。

Creator 3.0 目前支持 **Planar** 和 **ShadowMap** 两种阴影类型。

基于图像的光照

基于图像的光照

光照探针

光照探针

光照探针面板

光照探针面板

通过编辑器顶部菜单，选择 **项目** -> **光照烘焙** -> **光照探针** 可以打开光照探针烘焙面板。

光照烘焙面板 会将 **场景编辑器** 内已布置好的 光照探针 进行烘焙。烘焙后，静态物体的间接光信息将被记录到存储介质中；而所有被配置为 **Movable** 属性的节点将可以获取到更详细和真实可信的间接光光照效果。

属性

反射探针

反射探针

自 v3.7 开始，Cocos Creator 支持反射探针。

反射探针是将选择范围内的反射光，使用烘焙或实时的方式应用到当前场景上，以提高场景光照可信度的组件。

在 **层级管理器** 或顶部菜单上选择 **光源** -> **反射探针** 即可在场景内创建反射探针。

反射探针面板

反射探针面板

通过主菜单上的 **项目** -> **光照烘焙** -> **反射探针** 可以打开反射探针烘焙面板。

反射探针美术工作流

反射探针美术工作流

烘焙反射探针工作流

- 在场景内创建 **反射探针** 节点
- 将需要烘焙反射的节点的 **Mobility** 属性修改为 **Static**

基于图像的光照示例

基于图像的光照示例

在 Cocos Creator 中开发者可通过组合基于图像光照的功能。这些功能包含：

- 通过 天空盒 的烘焙反射卷积图功能以提供更好的环境反射效果
- 通过 光照贴图 将光照信息烘焙到贴图以提高光照性能
- 通过 光照探针/反射探针 检测物体间的反射信息

本文将从艺术资产工作者的工作流演示如何在您的场景中烘焙基于图像的照明。

准备工作

由于光照探针和反射探针都是针对基于物理的光照模型，因此在制作美术资源时，请遵循 PBR 工作流程。

请提前准备好使用以下着色器的材质文件；或通过 导入从 DCC 工具导出的模型 导出，导入到 **资源管理器** 后会自动识别模型内的材质并将其着色器转化为引擎支持的 PBR 着色器。

光照贴图

光照贴图

烘焙系统会对光源稳定的静态物体所受到的光照和阴影等进行预先计算，计算产生的结果存放在一张纹理贴图中，这张贴图我们称之为 **光照贴图**。

生成的光照贴图 Creator 会在运行时自动处理并使用。在光源固定的场景中，使用光照贴图代替实时的光照计算，可以减少资源消耗，从而提高场景运行效率。

光照贴图面板

点击编辑器菜单栏的 **项目 -> 光照贴图**，打开光照贴图面板。面板由 **Scene** 和 **Baked** 两个页面组成。

网格

网格（Meshes）

网格 一般用于绘制 3D 图像。Creator 提供了以下网格渲染器组件来渲染基础网格、蒙皮网格等，从而将模型绘制显示出来：

- MeshRenderer：网格渲染器组件，用于渲染基础的模型网格。
- SkinnedMeshRenderer：蒙皮网格渲染器组件，用于渲染蒙皮模型网格。
- SkinnedMeshBatchRenderer：批量蒙皮网格渲染器组件，用于将同一个骨骼动画组件控制的所有子蒙皮模型合并渲染。

同时，模型若要应用于实际的物理碰撞中，实现类似凹凸不平的路面效果，可以使用网格碰撞组件，会根据模型形状生成碰撞网格。详情请参考 使用网格碰撞。

MeshRenderer 组件

MeshRenderer 组件参考

MeshRenderer（网格渲染器）组件用于显示一个静态的 3D 模型。通过 **Mesh** 属性设置模型网格，通过 **Materials** 属性控制模型的显示外观。

在 **属性检查器** 中点击 **添加组件 -> Mesh -> MeshRenderer** 即可添加 MeshRenderer 组件。

SkinnedMeshRenderer 组件

蒙皮网格渲染器组件（SkinnedMeshRenderer）

蒙皮网格渲染器组件（SkinnedMeshRenderer）主要用于渲染蒙皮模型网格。

导入模型资源 后，若模型网格中带有蒙皮信息，在使用模型时，SkinnedMeshRenderer 组件便会自动添加到模型节点上。

SkinnedMeshBatchRenderer 组件

批量蒙皮网格渲染器组件（SkinnedMeshBatchRenderer）

批量蒙皮网格渲染器组件（SkinnedMeshBatchRenderer）用于将同一个骨骼动画组件控制的所有子蒙皮模型网格合并渲染。

从第三方工具导出模型资源

导入从 DCC 工具导出的模型

目前大多数数字内容制作（Digital Content Creation, DCC）工具（3ds Max、Maya、Blender）都能导出 **FBX** 和 **glTF** 这两种格式的模型文件，所以这些工具导出的内容都能在 Cocos Creator 中得到良好的展示。

导出 FBX

因为 DCC 工具的坐标系和游戏引擎的坐标系可能不一致，所以在导出模型时需要进行一些变换才能在引擎中得到想要的结果。例如：Blender 的坐标系为 X 轴朝右，Y 轴朝里，Z 轴朝上，而 Cocos Creator

3.x 的坐标系为 X 轴朝右，Y 轴朝上，Z 轴朝外，所以需要调整旋转才能使得轴向一致。

以下以 Blender 2.8 作为例子，介绍模型的导入流程，首先我们在 Blender 中创建一个模型。

从 3ds Max 中导出 FBX 模型资源

从 3ds Max 中导出 FBX 模型资源

导出步骤

1. 在 3ds Max 选中要导出的模型

从 Maya 中导出 FBX 模型资源

从 Maya 中导出 FBX 模型资源

导出步骤

1. 选中要导出的模型：

---

glTF 模型

glTF 模型

Cocos Creator 支持 glTF 2.0 及更早的文件格式。

URI 解析

Creator 支持 glTF 中指定以下形式的 URI：

- Data URI
- 相对 URI 路径
- 文件 URL
- 文件路径

转换关系

当导入 glTF 模型到 Creator 时，glTF 中的资源将会按照以下关系转换为 Creator 中的资源：

glTF 资源	Cocos Creator 资源
glTF 场景	预制体
glTF 网格	网格
glTF 蒙皮	骨骼
glTF 材质	材质
glTF 贴图	贴图
glTF 图像	图像
glTF 动画	动画剪辑

glTF 场景

导入后，glTF 场景将转换为 Creator 中的预制体资源，glTF 场景中递归包含的节点也将按照相同层级关系一一转换为预制体中的节点。

场景根节点

预制体将使用一个不带任何空间转换信息的节点作为根节点，glTF 场景的所有 根节点 将作为该节点的子节点。

节点转换

glTF 节点中的属性将按照下表中的映射关系转换为预制体节点中的属性：

glTF 节点属性	预制体节点属性
层级关系	层级关系
位移	位置
旋转	旋转
缩放	缩放
矩阵	解压，并分别设置位置、旋转、缩放
网格引用	网格渲染器组件
蒙皮引用	蒙皮网格渲染器组件
初始权重	（蒙皮）网格渲染器组件权重

网格渲染器

若 glTF 节点引用了网格，那么导入后相对应的预制体节点也会添加网格渲染组件（MeshRenderer）。若该 glTF 节点还引用了蒙皮，那么相对应的预制体节点还会添加蒙皮网格渲染组件（SkinnedMeshRenderer）。

（蒙皮）网格渲染组件中的网格、骨骼和材质，都会与转换后的 glTF 网格、蒙皮、材质资源一一对应。

若 glTF 节点指定了初始权重，则转换后的（蒙皮）网格渲染器也将带有此权重。

glTF 网格

导入后，glTF 网格将转换为 Cocos Creator 中的网格资源。

glTF 网格中的所有 基元体 将被一一转换为 Creator 中的子网格。

若 glTF 网格指定了 权重，则相应地，转换后的 Creator 网格中也将存储相应的权重。

glTF 基元体

glTF 基元体的索引数组将一一对应转换为 Cocos Creator 子网格的索引数组。

glTF 基元模式将按照下表中的映射关系转换为 Cocos Creator 基元模式：

glTF 基元模式	Cocos Creator 基元模式
POINTS	gfx.PrimitiveMode.POINT_LIST
LINES	gfx.PrimitiveMode.LINE_LIST
LINE_LOOP	gfx.PrimitiveMode.LINE_LOOP
LINE_STRIP	gfx.PrimitiveMode.LINE_STRIP
TRIANGLES	gfx.PrimitiveMode.TRIANGLE_LIST
TRIANGLE_STRIP	gfx.PrimitiveMode.TRIANGLE_STRIP
TRIANGLE_FAN	gfx.PrimitiveMode.TRIANGLE_FAN

glTF 顶点属性将转换为 Cocos Creator 顶点属性，属性名称的转换如下表所示：

glTF 顶点属性名称	Cocos Creator 顶点属性名称
POSITION	gfx.AttributeName.ATTR_POSITION
NORMAL	gfx.AttributeName.ATTR_NORMAL
TANGENT	gfx.AttributeName.ATTR_TANGENT
TEXCOORD_0	gfx.AttributeName.ATTR_TEX_COORD
TEXCOORD_1..TEXCOORD_8	gfx.AttributeName.ATTR_TEX_COORD1..gfx.AttributeName.ATTR_TEX_COORD8
COLOR_0	gfx.AttributeName.ATTR_COLOR
COLOR_1..COLOR_2	gfx.AttributeName.ATTR_COLOR1..gfx.AttributeName.ATTR_COLOR2
JOINTS_0	gfx.AttributeName.ATTR_JOINTS
WEIGHTS_0	gfx.AttributeName.ATTR_WEIGHTS

注意：若 glTF 基元体中存在其他 JOINTS、WEIGHTS 顶点属性，例如 JOINTS_1、WEIGHTS_1，则意味着此 glTF 网格的顶点可能受到多于 4 根骨骼的影响。

对于每个顶点，所有由 JOINTS_{}、WEIGHTS_{} 确定的权重信息将按权重值进行排序，取出影响权重最大的四根骨骼作为 gfx.AttributeName.ATTR_JOINTS 和 gfx.AttributeName.ATTR_WEIGHTS。

glTF 形变目标将被转换为 Cocos Creator 子网格形变数据。

glTF 蒙皮

导入后，glTF 蒙皮将转换为 Cocos Creator 中的骨骼资源。

glTF 材质

导入后，glTF 材质将转换为 Cocos Creator 中的材质资源。

glTF 贴图

导入后，glTF 贴图将转换为 Cocos Creator 中的贴图资源。

glTF 贴图中引用的 glTF 图像将转换为对相应转换后的 Cocos Creator 图像的引用。

glTF 贴图属性将按照下表中的映射关系转换为 Cocos Creator 贴图属性：

glTF 贴图属性	Cocos Creator 贴图属性
放大筛选器	放大筛选器
缩小筛选器	缩小筛选器、Mip Map 筛选器
S 环绕模式	S 环绕模式
T 环绕模式	环绕模式

glTF 贴图放大筛选器将按照下表中的映射关系转换为 Cocos Creator 贴图放大筛选器：

glTF 贴图放大筛选器	Cocos Creator 贴图放大筛选器
NEAREST	TextureBase.Filter.NEAREST
LINEAR	TextureBase.Filter.LINEAR

glTF 贴图缩小筛选器将按照下表中的映射关系转换为 Cocos Creator 贴图缩小筛选器和 Cocos Creator 贴图 Mip Map 筛选器：

glTF 贴图缩小筛选器	Cocos Creator 贴图缩小筛选器	Cocos Creator 贴图 Mip Map 筛选器
NEAREST	TextureBase.Filter.NEAREST	TextureBase.Filter.NONE
LINEAR_MIPMAP_LINEAR	TextureBase.Filter.LINEAR	TextureBase.Filter.NONE
LINEAR_MIPMAP_NEAREST	TextureBase.Filter.NEAREST	TextureBase.Filter.NEAREST
LINEAR	TextureBase.Filter.LINEAR	TextureBase.Filter.NEAREST
NEAREST_MIPMAP_LINEAR	TextureBase.Filter.NEAREST	TextureBase.Filter.LINEAR
NEAREST_MIPMAP_NEAREST	TextureBase.Filter.LINEAR	TextureBase.Filter.LINEAR

glTF 贴图环绕模式将按照下表中的映射关系转换为 Cocos Creator 贴图环绕模式：

glTF 贴图环绕模式	Cocos Creator 贴图环绕模式
CLAMP_TO_EDGE	TextureBase.WrapMode.CLAMP_TO_EDGE
REPEAT	TextureBase.WrapMode.REPEAT
MIRRORED_REPEAT	TextureBase.WrapMode.MIRRORED_REPEAT

glTF 图像

导入后，glTF 图像将转换为 Cocos Creator 中的图像资源。

当 glTF 图像的 URI 是 Data URI 时，图像数据将从 Data URI 中获取。否则，将根据 Cocos Creator 图像位置解析算法 解析并引用外部图像文件，其中 url 就是 glTF 图像的 URI，startDir 为 glTF 文件所在目录。

glTF 动画

导入后，glTF 动画将转换为 Cocos Creator 动画资源。

程序化创建网格

程序化创建网格

当由 DCC（Digital Content Creation）软件制作或引擎内的地形编辑器制作的模型无法满足需求时，可以通过 API 来创建网格。如需要在运行时创建某种可以生长的蛇、动态编辑模型或实现某些曲面，都可以通过程序化来创建网格。

创建网格

引擎支持两种网格：**静态网格** 和 **动态网格**，适用于不同的场景，开发者可按需使用。

- 静态网格，通过 utils.MeshUtils.createMesh 创建，一旦创建成功，网格内的几何体不可编辑的。
- 动态网格：通过 utils.MeshUtils.createDynamicMesh 创建，创建成功后，网格内的几何体仍然可以修改。

返回值为 Mesh 组件，因此我们方便的将其赋值给 MeshRenderer 的 mesh 属性，如此即可将其显示在屏幕上。

API

API 请参考 MeshUtils。

范例

顶点动画贴图 （VAT）

顶点动画贴图 （VAT）

顶点动画贴图（Vertex Animation Texture，VAT）通常用于表达刚体破碎、布料、流体等物理模拟动画。在 Houdini 等 PCG 软件中计算好的物理模拟动画可烘焙为 VAT 数据导出，并在实时渲染引擎中回放，以极低的性能消耗来重现复杂的物理效果。

目前支持的 VAT 数据格式为 Houdini 导出的刚体、柔体和流体数据，以及泽森科工 Zeno 的流体数据。

数据导出与使用

导出的 VAT 数据包含 fbx 模型、动画贴图、包含 VAT 材质属性的 json 文件（帧数量和包围盒最大最小值等）三部分。

刚体

1、导出：Houdini 设置好刚体碰撞、破碎等动画后，选择 **VAT2.0 版本，UE 模式，无 paddle，LDR方式** 导出。

2、材质：在 Cocos Creator 中创建一个材质，Effect 选择 **util/dcc/vat/houdini-rigidbody-v2**。

3、参数：查看导出的 json 文件，将 Animation Speed、NumOfFrames、PivotMin/Max、PosMin/Max 等数据填到材质中。

4、贴图：展开导出的 Position 形状贴图，将本体拖入PositionMap，下属的 Sign 和 Alpha 分别拖入 PosSignMap 和 PosAlphaMap（若有）。 Rotation 法线贴图同理。

柔体

1、导出：Houdini 设置好柔体碰撞、破碎等动画后，选择 **VAT3.0 版本，Unity 模式，无 paddle，LDR方式** 导出。

2、材质：在 Cocos Creator 中创建一个材质，Effect 选择 **util/dcc/vat/houdini-softbody-v3**。

3、参数：查看导出的 json 文件，将 Animation Speed、NumOfFrames 等数据填到材质中。

4、贴图：展开导出的 Position 形状贴图，将本体拖入 PositionMap，下属的 Sign 和 Alpha 分别拖入 PosSignMap 和 PosAlphaMap（若有）。 Rotation 法线贴图同理。

流体（Houdini）

1、导出：Houdini 设置好流体动画后，选择 **VAT3.0 版本，Unity 模式，无 paddle，LDR方式** 导出。

2、材质：在 Cocos Creator 中创建一个材质，Effect 选择 **util/dcc/vat/houdini-fluid-v3-liquid**。

3、参数：查看导出的 json 文件，将 Animation Speed、NumOfFrames 等数据填到材质中。

```
- 勾选 Use Lookup Texture，将导出的 lookup 图拖入 LookupMap
- 勾选 Use Lookup Auto Frame，将导出的 lookup 图的分辨率填入 LUTTextureWidth/Height
- 选中导出的 fbx 模型，将其顶点数记录下来填入 FBXVertexCount
```

4、贴图：展开导出的 Position 形状贴图，将本体拖入PositionMap，下属的 Sign 和 Alpha 分别拖入 PosSignMap 和 PosAlphaMap（若有）。 Rotation 法线贴图同理。

流体（Zeno）

1、导出：Zeno 设置好流体动画后，选择 **VAT模式** 导出。

2、材质：在 Cocos Creator 中创建一个材质，Effect 选择 **util/dcc/vat/zeno-fluid-liquid**。

3、参数：查看导出的 json 文件，将 Animation Speed、NumOfFrames 等数据填到材质中。

4、贴图：展开导出的 Position 形状贴图拖入 PositionMap，Rotation 法线贴图拖入 RotationMap。

错误效果调试

刚体

若出现碎块绕圈转等动画异常，请确认导出选项是否指定的 VAT2.0 UE 模式，若选错为 Unity 模式可能导致此情况发生。

流体

若出现面破碎甚至有些面错乱，请检查 NumOfFrames 是否与 DCC 软件中的一致

若整个流体在抖或有穿插的乱面，请确认两张贴图的过滤模式为 Nearest，且一定不能勾选消除透明伪影（fix alpha transparency artifacts）

纹理

纹理（Textures）

纹理是一张可显示的图像，或一段用于计算的中间数据，通过 UV 坐标映射到渲染物体表面，使之效果更为丰富精彩且真实。Creator 中纹理的应用包括以下几种：

- 用于 2D UI 渲染，参考 SpriteFrame。
- 用于 3D 模型渲染，需要在材质中指定 纹理贴图资源，才能将其渲染映射到网格表面。纹理贴图还支持在 导入图像资源 时将其切换为 **立方体贴图** 或 **法线贴图**。
- 用于粒子系统，使粒子表现更丰富。与 3D 模型一样，纹理在粒子系统中的应用也依赖于材质。
- 用于地形渲染，参考 地形系统。

更多内容

- 渲染纹理（RenderTexture）
- 压缩纹理

纹理贴图

纹理贴图资源（Texture）

纹理贴图资源是一种用于程序采样的资源，如模型上的贴图、精灵上的 UI。当程序渲染 UI 或者模型时，会使用纹理坐标获取纹理颜色，然后填充在模型网格上，再加上光照等等一系列处理便渲染出了整个场景。

纹理贴图资源可由图像资源（ImageAsset）转换而来，图像资源包括一些通用的图像转换格式如 PNG、JPEG 等等。

Texture2D

Texture2D 是纹理贴图资源的一种，通常用于 3D 模型的渲染，如模型材质中的反射贴图、环境光遮罩贴图等等。

在将图像资源 导入 到 Creator 后，即可在 **属性检查器** 面板将其设置为 **texture** 类型，texture 类型便是 Texture2D 纹理资源。

立方体贴图

立方体贴图

TextureCube 为立方体纹理，常用于设置场景的 天空盒。立方体贴图可以通过设置全景图 ImageAsset 为 TextureCube 类型获得，也可以在 Creator 中制作生成。

设置为立方体贴图

在将 ImageAsset 导入 到 Creator 后，即可在 **属性检查器** 面板将其设置为 **texture cube** 类型，设置完成后请点击右上角的绿色打钩按钮，以保存修改。

压缩纹理

压缩纹理

Cocos Creator 可以直接在编辑器中设置纹理需要的压缩方式，然后在项目发布时自动对纹理进行压缩。支持同一平台同时导出多种图片格式，引擎将根据不同的设备支持情况自动下载合适的格式。

配置压缩纹理

Cocos Creator 支持导入多种格式的图片（具体见下表），但是在实际游戏运行中，我们不建议使用原始图片作为资源来加载。比如在手机平台上可能只需要原图 80% 或者更少的画质，又或者是没有使用到透明通道的 `.png` 可以将其转换成 `.jpg`，这样可以减少很大一部分图片的存储空间。

图片格式	Android	iOS	Mini Game	Web
PNG	支持	支持	支持	支持
JPG	支持	支持	支持	支持
WEBP	Android 4.0 以上原生支持，其他版本可以使用 解析库 可以使用	解析库	支持	部分支持
PVR	不支持	支持	支持 iOS 设备	支持 iOS 设备
ETC1	支持	不支持	支持 Android 设备	支持 Android 设备
ETC2	部分支持，取决于手机硬件	不支持	不支持	支持部分 Android 设备
ASTC	部分支持	部分支持	不支持（iOS 版微信小游戏 v8.0.3 以上支持）	部分支持

默认情况下 Creator 在构建的时候输出的是原始图片，如果在构建时需要对某一张图片或者自动图集进行压缩，可以在 **资源管理器** 中选中这张图片或图集，然后在 **属性检查器** 中勾选 `useCompressTexture`，再在 `presetId` 中选择图片的纹理压缩预设，设置完成后点击右上角的绿色打钩按钮，即可应用。

自定义构建纹理压缩处理

纹理压缩目前是在构建后生效，编辑器自带了一套处理工具。若需要自定义压缩工具，请参考 自定义纹理压缩。

渲染纹理

渲染纹理资源（Render Texture）

渲染纹理是一张在 GPU 上的纹理。通常我们会把它设置到相机的 **目标纹理** 上，使相机照射的内容通过离屏的 `frambuffer` 绘制到该纹理上。一般可用于制作汽车后视镜，动态阴影等功能。

创建渲染纹理资源

在 **资源管理器** 中点击左上方的 + 按钮，然后选择 **渲染纹理**，即可创建渲染纹理资源：

材质系统

材质系统总览

在脚本中使用材质

程序化使用材质

创建材质

材质（Material）资源可以看成是着色器资源（EffectAsset）在场景中的资源实例。

Creator 支持在 **资源管理器** 中手动 创建材质资源，同时也支持通过 IMaterialInfo 接口在脚本模块中程序化地创建材质。`IMaterialInfo` 的可配置参数包括：

- `effectAsset`/`effectName`：effect 资源引用，指定使用哪个 EffectAsset 所描述的流程进行渲染。（`effectAsset` 和 `effectName` 二者必须选其一）
- `technique`：指定使用 EffectAsset 中的第几个 technique，默认为第 0 个。
- `defines`：宏定义列表，指定开启哪些 预处理宏定义，默认全部关闭。
- `states`：管线状态重载列表，指定对渲染管线状态（深度模板透明混合等）有哪些重载，默认与 effect 声明一致。

创建代码示例：

```
const mat = new Material();
mat.initialize({
    // 通过 effect 名指定材质使用的着色器资源
    effectName: 'pipeline/skybox',
    defines: {
        USE_RGBE_CUBEMAP: true
    }
});
```

使用材质

内置材质

内置材质

Creator 在 **资源管理器** 面板的 **internal/default_materials/** 目录下内置了几种常见的材质，其使用的 Effect 为 内置着色器 ，内置材质的属性都不允许修改。

材质系统类图

材质系统类图

材质系统控制着每个模型最终的着色流程与顺序，在引擎内相关类间结构如下：

着色器

着色器（Cocos Shader）

创建与使用

着色器创建与使用

创建着色器

在 **资源管理器** 面板中点击左上角的 + 号按钮（或者在 Assets 目录下点击右键），在弹出菜单中选择 **着色器（Effect）** 或者 **表面着色器（Surface Shader）**， 便可创建新的着色器资源。

内置着色器

内置着色器

引擎提供了一系列通用的内置着色器，位于编辑器 **资源管理器** 面板的 `internal -> effects` 目录下。双击着色器文件即可在外部 IDE 打开进行查看和编辑（前提是需要在 **偏好设置 -> 外部程序** 中配置 **默认脚本编辑器**）。

Cocos Creator 将内置着色器大致归类为以下几种：

- `internal`：内置引擎功能相关着色器，比如编辑器内用 gizmo，几何体渲染等等。用户通常不需要关注这些。
- `pipeline`：管线特效着色器，包括延迟光照、后效和抗锯齿等。
- `util`：存放一些零散的内置着色器，例如 DCC 材质导入和序列帧动画等。用户通常不需要关注这些。
- `for2d`：2D 渲染相关着色器，如 spine 和 sprite 动画等。
- `particles`：粒子特效相关着色器。
- `advanced`：基于 表面着色器 制作的一些高级材质，如水面、皮肤、头发、玉石等等，引擎会持续迭代。
- 其他的为内置着色器，详情请参考下文说明。

内置管线特效着色器

基于物理的光照模型 PBR

基于物理的光照模型（Physically Based Rendering - PBR）

Cocos Creator 从 v3.0 开始提供了基于物理渲染（PBR）的光照着色器：`builtin-standard.effect`。PBR 根据现实中光线传播原理和能量守恒定律，模拟出近似于真实物理光照的效果。

PBR 的优势在于：

- 真实性：基于物理原理的渲染让最终效果更加逼真
- 一致性：美术制作流程规范化、制作标准统一化
- 复用性：模型材质与光照环境分离，在所有 PBR 项目中均可复用

使用 PBR 制作材质和纹理

在 **资源管理器** 面板中手动创建的材质，默认使用的是 `builtin-standard.effect` 着色器，我们称之为 PBR 材质，PBR 材质使用 PBR 流程中的 Metal/Roughness 工作流。

在使用 PBR 材质进行渲染时，为获得正确的渲染效果，至少需要设置材质的 **固有色（Albedo）**、**粗糙度（Roughness）** 和 **金属度（Metallic）**。这些都可以在材质资源属性面板中进行设置：

卡通渲染

卡通渲染

相对于 真实渲染（Physical Based Rendering - PBR），非真实渲染（Non-Photorealistic Rendering - NPR）通过特性化渲染，实现与真实世界完全不同的美术表现。

卡通渲染（Toon Shading）是非真实渲染的常见效果之一。

通常卡通渲染的内容包含以下几个基础部分：

- 对物体进行描边
- 降低色阶的数量并模拟色阶不连续现象
- 明暗色调分离
- 阴影形状干扰等

无光照

无光照

无光照是最基础的着色模型，这种模型下，引擎的任何光源都无法影响其最终效果，适用于：

- 不受光源影响的物体
- 画面要求不高或性能要求高的场景

在材质的 **Effect** 属性中将着色器切换为 Cocos Creator 内置的无光照着色器（builtin-unlit.effect）时，如下图：

着色器语法

着色器语法

Cocos Creator 中的着色器（Cocos Shader，文件扩展名为 *.effect），是一种基于 YAML 和 GLSL 的单源码嵌入式领域特定语言（single-source embedded domain-specific language），YAML 部分声明流程控制

清单，GLSL 部分声明实际的 Shader 片段，这两部分内容相互补充，共同构成了一个完整的渲染流程描述。

注意：推荐使用 Visual Studio Code 编写 Cocos Shader，并在应用商店中安装 Cocos Effect 扩展，提供编写时的语法高亮提示。

Pass 可选配置参数

Pass 可选配置参数

Pass 中的参数主要分两个部分：

- 开发者可自定义的 **属性检查器** 面板参数 `properties`
- 引擎提供的用于控制渲染管线状态的 `PipelineStates`

Properties

`properties` 用于将 Shader 中定义的 `uniform` 进行别名映射。这个映射可以是某个 `uniform` 的完整映射，也可以是具体某个分量的映射（使用 `target` 参数），代码示例如下：

```
properties:
  albedo: { value: [1, 1, 1, 1] } # uniform vec4 albedo
  roughness: { value: 0.8, target: pbrParams.g } # uniform vec4 pbrParams
  offset: { value: [0, 0], target: tilingOffset.zw } # uniform vec4 tilingOffset
# say there is another uniform, vec4 emissive, that doesn't appear here
# so it will be assigned a default value of [0, 0, 0, 0] and will not appear in the inspector
```

默认情况下，`properties` 中定义的属性参数会暴露并显示在编辑器的 **属性检查器** 面板中，方便进行可视化控制。

如果不想显示在 **属性检查器** 面板上，可在定义属性时加上 `editor: { visible: false }`，代码示例如下：

```
properties:
  factor: { value: 1.0, editor: { visible: false } }
```

在 TypeScript 中可以使用 `Material` 类的 `setProperty` 方法以及 `Pass` 的 `setUniform` 方法进行设置，代码示例如下：

```
mat.setProperty('emissive', Color.GREY); // 直接设置对应的 Uniform 变量
mat.setProperty('albedo', Color.RED);
mat.setProperty('roughness', 0.2); // 仅设置对应的分量
const h = mat.passes[0].getHandle('offset'); // 获取对应的 Uniform 的句柄
mat.passes[0].setUniform(h, new Vec2(0.5, 0.5)); // 使用 'Pass.setUniform' 设置 Uniform 属性
```

注意：在 Shader 中定义的 `uniform` 也可以使用上述代码进行设置，即使没有在 `properties` 中定义。

未指定的 `uniform`，引擎将会在运行时根据自动分析出的数据类型给予默认值。更多关于默认值的内容，请参考下文说明。

为方便声明各 `property` 子属性，可以直接在 `properties` 内声明 `__metadata__` 项，所有 `property` 都会继承它声明的内容，如：

```
properties:
  __metadata__: { editor: { visible: false } }
  a: { value: [1, 1, 0, 0] }
  b: { editor: { type: color } }
  c: { editor: { visible: true } }
```

这样 uniform `a` 和 `b` 已声明的各项参数都不会受到影响，但都不会显示在 **属性检查器** 中（visible 为 false），而 uniform `c` 仍会正常显示。

Property 参数列表

Property 中可配置的参数如下表所示，任何可配置的字段如果和默认值相同都可以省掉。

参数	默认值	可选项	备注
target	**undefined**	undefined	任意有效的 uniform 通道，可指定连续的单个或多个，但不可随机重排
value			详见下文 **Default Values** 部分的介绍
sampler.minFilter	**linear**	none, point, linear, anisotropic	
sampler.magFilter	**linear**	none, point, linear, anisotropic	
sampler.mipFilter	**none**	none, point, linear, anisotropic	
sampler.addressU	**wrap**	wrap, mirror, clamp, border	
sampler.addressV	**wrap**	wrap, mirror, clamp, border	
sampler.addressW	**wrap**	wrap, mirror, clamp, border	
sampler.maxAnisotropy	**16**	16	
sampler.cmpFunc	**never**	never, less, equal, less_equal, greater, not_equal, greater_equal, always	
sampler.borderColor	**[0, 0, 0, 0]**	[0, 0, 0, 0]	
sampler.minLOD	**0**	0	
sampler.maxLOD	**0**	0	如果允许 mipmap 则要根据贴图修改最大 mip 值
sampler.mipLODBias	**0**	0	
editor.displayName	***property name**	*property name	任意字符串
editor.type	**vector**	vector, color	
editor.visible	**true**	true, false	
editor.tooltip	***property name**	*property name	任意字符串
editor.range	**undefined**	undefined, [min, max, [step]]	
editor.deprecated	**false**	true, false	deprecated 标记的数据表示在导入时已更新或在当前版本已废弃，其内容在保存时会被自动删除

Property 默认值

对于 Property 的默认值， Cocos Shader 做出了如下的规定：

类型	默认值	可选项
int		0

类型	默认值	可选项
ivec2		[0, 0]
ivec3		[0, 0, 0]
ivec4		[0, 0, 0, 0]
float		0
vec2		[0, 0]
vec3		[0, 0, 0]
vec4		[0, 0, 0, 0]
sampler2D	**default**	black, grey, white, normal, default
samplerCube	**default-cube**	black-cube, white-cube, default-cube

对于 `defines`：

- boolean 类型默认值为 false。
- number 类型默认值为 0，默认取值范围为 [0, 3]。
- string 类型默认值为 options 数组第一个元素。

PipelineStates

以下为 `PipelineStates` 相关参数，所有参数不区分大小写。

参数名	说明	默认值	备注
switch	指定这个 pass 的执行依赖于哪个 define。可以是任意有效的宏名称，但不应与使用到的 shader 中定义的任何 define 重名	未定义	这个字段默认是不存在的，意味着这个 pass 是无条件执行的
priority	指定这个 pass 的渲染优先级，数值越小渲染优先级越高，取值范围为 **0 ~ 255**	128	可结合四则运算符指定相对值
stage	指定这个 pass 归属于管线的哪个 stage。可以是运行时管线中任何注册的 Stage 名称	**default**	对于默认的 forward 管线，只有 `default` 一个 stage
phase	指定这个 pass 归属于管线的哪个 phase。可以是运行时管线中任何注册的 Phase 名称	**default**	对于默认的 forward 管线，可以是 `default`、`forward-add` 或者 `shadow-caster`
propertyIndex	指定这个 pass 运行时的 uniform 属性数据要和哪个 pass 保持一致，例如 forward add 等 pass 需要和 base pass 一致才能保证正确的渲染效果。可以是任意有效的 pass 索引	未定义	一旦指定了此参数，材质面板上就不会再显示这个 pass 的任何属性
embeddedMacros	指定在这个 pass 的 shader 基础上额外定义的常量宏，可以是一个包含任意宏键值对的对象	未定义	只有当宏定义不同时才能在多个 pass 中使用此参数来复用 shader 资源
properties	Properties 存储着这个 pass 中需要显示在 **属性检查器** 上的可定制的参数		详见上文 **Properties** 部分的内容
migrations	迁移旧的材质数据		详见下文 **Migrations** 部分的介绍
primitive	创建材质顶点数据	**triangle_list**	可选项包括：point_list、line_list、line_strip、line_loop **triangle_list**、triangle_strip、triangle_fan line_list_adjacency、line_strip_adjacency triangle_list_adjacency、triangle_strip_adjacency triangle_patch_adjacency、quad_patch_list、iso_line_list
RasterizerState	光栅化时的可选渲染状态		详见下文 **RasterizerState** 部分的介绍
DepthStencilState	深度和模板缓存的测试与状态		详见下文 **DepthStencilState** 部分的介绍
BlendState	材质混合状态	**false**	详见下文 **BlendState** 部分的介绍

Migrations

一般情况下，在使用材质资源时都希望底层的 effect 接口能始终向前兼容，但有时面对新的需求最好的解决方案依然是含有一定 breaking change 的，这时为了保持项目中已有的材质资源数据不受影响，或者至少能够更平滑地升级，就可以使用 effect 的迁移系统。

在 effect 导入成功后会 **立即更新工程内所有** 依赖于此 effect 的材质资源，对每个材质资源，会尝试寻找所有指定的旧参数数据（包括 **property** 和 **宏定义** 两类），然后将其复制或迁移到新属性中。

> **注意**：使用迁移功能前请一定先备份好项目工程，以免丢失数据！

对于一个现有的 effect，迁移字段声明如下：

```
migrations:
  # macros: # macros follows the same rule as properties, without the component-wise features
  # USE_MIAN_TEXTURE: { formerlySerializedAs: USE_MAIN_TEXTURE }
  properties:
    newFloat: { formerlySerializedAs: oldVec4.w }
```

对于一个依赖于这个 effect，并在对应 pass 中持有属性的材质：

```
{
  "oldVec4": {
    "__type__": "cc.Vec4",
    "x": 1,
    "y": 1,
    "z": 1,
    "w": 0.5
  }
}
```

在 effect 导入成功后，这些数据会被立即转换成：

```
{
  "oldVec4": {
    "__type__": "cc.Vec4",
    "x": 1,
    "y": 1,
    "z": 1,
    "w": 0.5
  },
  "newFloat": 0.5
}
```

在 **编辑器** 内重新编辑并保存这个材质资源后会变成（假设 effect 和 property 数据本身并没有改变）：

```
{
  "newFloat": 0.5
}
```

当然如果希望在导入时就直接删除旧数据，可以再加一条迁移信息来专门指定这点：

```
oldVec4: { removeImmediately: true }
```

这对于在项目有大量旧材质，又能够确定这个属性的数据已经完全冗余时会比较有用。

更多地，注意这里的通道指令只是简单的取 `w` 分量，事实上还可以做任意的 shuffle：

```
newColor: { formerlySerializedAs: someOldColor.yxx }
```

甚至基于某个宏定义：

```
occlusion: { formerlySerializedAs: pbrParams.<OCCLUSION_CHANNEL|z> }
```

这里声明了新的 occlusion 属性会从旧的 `pbrParams` 中获取，而具体的分量取决于 `OCCLUSION_CHANNEL` 宏定义。并且如果材质资源中未定义这个宏，则默认取 `z` 通道。
但如果某个材质在迁移升级前就已经存着 `newFloat` 字段的数据，则不会对其做任何修改，除非指定为强制更新模式：

```
newFloat: { formerlySerializedAs: oldVec4.w! }
```

强制更新模式会强制更新所有材质的属性，无论这个操作是否会覆盖数据。

注意：强制更新操作会在编辑器的每次资源事件中都执行（几乎对应每一次鼠标点击，相对高频），因此只是一个快速测试和调试的手段，一定不要将处于强制更新模式的 effect 提交到版本控制。

再次总结一下为防止数据丢失所设置的相关规则：

- 为避免有效旧数据丢失，只要没有显式指定 `removeImmediately` 规则，就不会在导入时自动删除旧数据；
- 为避免有效的新数据被覆盖，如果没有指定为强制更新模式，对于那些既有旧数据，又有对应的新数据的材质，不会做任何迁移操作。

RasterizerState

参数名	说明	默认值	可选项
isDiscard	引擎预留	**false**	true, false
polygonMode	多边形绘制模式	**fill**	point，line，fill
shadeModel	着色模型	**flat**	flat, gourand
cullMode	光栅化时剔除模式	**back**	front, back, none
isFrontFaceCCW	是否逆时针（CCW）前向	**true**	true，false
depthBias	深度偏移	**0**	
depthBiasSlop	深度偏差斜率	**0**	
depthBiasClamp	深度截断	**0**	
isDepthClip	允许深度剪裁操作 Vulkan 专用	**true**	true, false
isMultisample	是否开启多重采样	**false**	true, false
lineWidth	线宽	1	

DepthStencilState

参数名	说明	默认值	可选项
depthTest	是否开启深度测试	**true**	true,false
depthWrite	是否开启深度缓存写入	**true**	true, false
depthFunc	深度缓存比较方法	**less**	never, less, equal, less_equal, greater, not_equal, greater_equal, always
stencilTestFront	是否开启正面模板缓存测试	**false**	true, false
stencilFuncFront	正面模板缓存比较方法	**always**	never, less, equal, less_equal, greater, not_equal, greater_equal, always
stencilReadMaskFront	正面模板缓存读取掩码	**0xffffffff**	0xffffffff, `[1, 1, 1, 1]`
stencilWriteMaskFront	正面模板缓存写入掩码	**0xffffffff**	0xffffffff, `[1, 1, 1, 1]`
stencilFailOpFront	正面模板缓存测试失败时，如何处理缓冲区的值	**keep**	keep, zero, replace, incr, incr_wrap, decr, decr_wrap, invert
stencilZFailOpFront	正面模板缓存深度测试失败时，如何处理缓冲区的值	**keep**	keep, zero, replace, incr, incr_wrap, decr, decr_wrap, invert
stencilPassOpFront	正面模板缓存测试通过时，如何处理缓冲区的值	**keep**	keep, zero, replace, incr, incr_wrap, decr, decr_wrap, invert
stencilRefFront	正面模板缓存中的比较函数用于比较的值	**1**	1, `[0, 0, 0, 1]`
stencilTestBack	是否开启背面模板缓存测试	**false**	true, false
stencilFuncBack	背面模板缓存比较方法	**always**	never, less, equal, less_equal, greater, not_equal, greater_equal, always
stencilReadMaskBack	背面模板缓存读取掩码	**0xffffffff**	0xffffffff, `[1, 1, 1, 1]`
stencilWriteMaskBack	背面模板缓存写入掩码	**0xffffffff**	0xffffffff, `[1, 1, 1, 1]`
stencilFailOpBack	背面模板缓存测试失败时，如何处理缓冲区的值	**keep**	keep, zero, replace, incr, incr_wrap, decr, decr_wrap, invert
stencilZFailOpBack	背面模板缓存深度测试失败时，如何处理缓冲区的值	**keep**	keep, zero, replace, incr, incr_wrap, decr, decr_wrap, invert
stencilRefBack	背面模板缓存中的比较函数用于比较的值	**1**	1, `[0, 0, 0, 1]`

BlendState

参数名	说明	默认值	可选项
isA2C	是否开启半透明反锯齿（Alpha To Coverage）	**false**	true，false
isIndepend	RGB 和 Alpha 是否分开混合	**false**	true，false
blendColor	指定混合颜色	**0**	0, `[0, 0, 0, 0]`
targets	混合配置，请参考下方的 targets	[]	

Targets

参数名	说明	默认值	可选项
Targets[i].blend	是否开启 **混合**	**false**	true, false
Targets[i].blendEq	指定 **混合源** 和 **混合目标** 的 RGB 的混合方程	**add**	add, sub, rev_sub
Targets[i].blendSrc	指定 **混合源** 的 RGB 混合因子	**one**	one, zero, src_alpha_saturate, src_alpha, one_minus_src_alpha, dst_alpha, one_minus_dst_alpha, src_color, one_minus_src_color, dst_color, one_minus_dst_color, constant_color, one_minus_constant_color, constant_alpha, one_minus_constant_alpha
Targets[i].blendDst	指定 **混合目标** 的 RGB 混合因子	**zero**	one, zero, src_alpha_saturate, src_alpha, one_minus_src_alpha, dst_alpha, one_minus_dst_alpha, src_color, one_minus_src_color, dst_color, one_minus_dst_color, constant_color, one_minus_constant_color, constant_alpha, one_minus_constant_alpha
Targets[i].blendSrcAlpha	指定 **混合源** 的 Alpha 混合因子	**one**	one, zero, src_alpha_saturate, src_alpha, one_minus_src_alpha, dst_alpha, one_minus_dst_alpha, src_color, one_minus_src_color, dst_color, one_minus_dst_color, constant_color, one_minus_constant_color, constant_alpha, one_minus_constant_alpha
Targets[i].blendDstAlpha	指定 **混合目标** 的 Alpha 混合因子	**zero**	one, zero, src_alpha_saturate, src_alpha, one_minus_src_alpha, dst_alpha, one_minus_dst_alpha, src_color, one_minus_src_color, dst_color, one_minus_dst_color, constant_color, one_minus_constant_color, constant_alpha, one_minus_constant_alpha
Targets[i].blendAlphaEq	指定 **混合源** 与 **混合目标** 的 Alpha 混合方法	**add**	add, sub, rev_sub

参数名	说明	默认值	可选项
Targets[i].blendColorMask	指定是否可将 RGB，Alpha 分量写入帧缓存	**all**	all, none, r, g, b, a, rg, rb, ra, gb, ga, ba, rgb, rga, rba, gba
dynamics	可动态更新的管线状态	[]	LINE_WIDTH, DEPTH_BIAS, BLEND_CONSTANTS, DEPTH_BOUNDS, STENCIL_WRITE_MASK, STENCIL_COMPARE_MASK

YAML 101 语法简介

YAML 101

Cocos Shader 使用的是符合 YAML 1.2 标准的解析器，这意味着 Cocos Shader 与 JSON 是完全兼容的，直接使用 JSON 也完全不会有问题，例如：

```
"techniques":
  [{
    "passes":
    [{
      "vert": "skybox-vs",
      "frag": "skybox-fs",
      "rasterizerState":
      {
        "cullMode": "none"
      }
      # ...
    }]
  }]
```

当然这也意味着繁琐的语法，所以 YAML 提供了一些更简洁的数据表示方式：

- 所有的引号和逗号都可以省略

  ```
  key1: 1
  key2: unquoted string
  ```

  注意：冒号后的空格不可省略

- 行首的空格缩进数量代表数据的层级[1]

  ```
  object1:
    key1: false
  object2:
    key2: 3.14
    key3: 0xdeadbeef
    nestedObject:
      key4: 'quoted string'
  ```

- 以 **连字符 - 空格** 开头，表示数组元素

  ```
  - 42
  - "double-quoted string"
  - arrayElement3:
      key1: punctuations? sure.
      key2: you can even have {}s as long as they are not the first character
      key3: { nested1: 'but no unquoted string allowed inside brackets', nested2: 'also notice the comma is back too' }
  ```

综合以上几点，本文开头的内容就可以简化成下面这样：

```
techniques:
- passes:
  - vert: skybox-vs
    frag: skybox-fs
    rasterizerState:
      cullMode: none
    # ...
```

另一个非常有用的 YAML 特性是数据间的引用与继承。

- **引用**

  ```
  object1: &o1
    key1: value1
  object2:
    key2: value2
    key3: *o1
  ```

  这个数据解析出来是这样的：

  ```
  {
    "object1": {
      "key1": "value1"
    },
    "object2": {
      "key2": "value2",
      "key3": {
        "key1": "value1"
      }
    }
  }
  ```

- **继承**

  ```
  object1: &o1
    key1: value1
    key2: value2
  object2:
    <<: *o1
    key3: value3
  ```

  这个数据解析出来是这样的：

  ```
  {
    "object1": {
      "key1": "value1",
      "key2": "value2"
    },
    "object2": {
      "key1": "value1",
      "key2": "value2",
      "key3": "value3"
    }
  }
  ```

对应到 Cocos Shader 中，比如多个 Pass 拥有相同的 `properties` 内容时（或更多其他需要复用数据的情况），可以很方便地复用数据，例如：

```
techniques:
- passes:
  - # pass 1 specifications...
    properties: &props # declare once...
      p1: { value: [ 1, 1, 1, 1 ] }
      p2: { sampler: { mipFilter: linear } }
      p3: { inspector: { type: color } }
  - # pass 2 specifications...
    properties: *props # reference anywhere
```

最后需要注意的是：在 Cocos Shader 文件中使用 YAML 声明的所有流程，都要确保是在 **CCEffect** 语法块内：

```
CCEffect %{
  # YAML starts here
}%
```

若有疑问可复制代码示例到任何 在线 YAML JSON 转换器 观察生成的数据。

参考链接

- https://en.wikipedia.org/wiki/YAML
- https://yaml.org/spec/1.2/spec.html

[1]. 标准 YAML 并不支持制表符，但在解析 Cocos Shader 内容时，我们会先尝试把其中所有的制表符替换为 2 个空格，以避免偶然插入制表符带来的麻烦。但请尽量避免插入制表符以确保编译无误。↩

GLSL 语法简介

GLSL 语法简介

GLSL 是为图形计算量身定制的用于编写着色器的语言，它包含一些针对向量和矩阵操作的特性，使渲染管线具有可编程性。本章主要介绍在编写 Shader 时常用的一些语法，包括以下几个方面：

- 变量
- 语句
- 限定符
- 预处理宏定义

变量

变量及变量类型

变量类型	说明	Cocos Shader 中的默认值	Cocos Shader 中的可选项
bool	布尔型标量数据类型	false	无
int/ivec2/ivec3/ivec4	包含 1/2/3/4 个整型向量	0/[0, 0]/[0, 0, 0]/[0, 0, 0, 0]	无
float/vec2/vec3/vec4	包含 1，2，3，4 个浮点型向量	0/[0, 0]/[0, 0, 0]/[0, 0, 0, 0]	无
sampler2D	表示 2D 纹理	**default**	black、grey、white、normal、default
samplerCube	表示立方体纹理	**default-cube**	black-cube、white-cube、default-cube
mat[2..3]	表示 2x2 和 3x3 的矩阵	不可用	
mat4	表示 4x4 的矩阵	[1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1]	

标量

构造标量的方式和 C 语言一致：

```
float floatValue = 1.0;
bool booleanValue = false;
```

向量

构造向量时的规则如下：

- 若向向量构造器提供了一个标量，则向量的所有值都会设定为该标量值
- 若提供多个标量值或向量，则从左到右使用提供的值赋值。前提是标量或向量的数量之和要等于向量构造器的数量

```
vec4 myVec4 = vec4(1.0);              // myVec4 = {1.0, 1.0, 1.0, 1.0}
vec2 myVec2 = vec2(0.5, 0.5);         // myVec2 = {0.5, 0.5}
vec4 newVec4 = vec4(1.0, 1.0, myVec2);// newVec4 = {1.0, 1.0, 0.5, 0.5}
```

向量可以通过 `r, g, b, a` 或 `x, y, z, w` 进行访问，也可以同时访问多个角标：

```
vec4 myVec4_0 = vec4(1.0); // myVec4_0 = { 1.0, 1.0, 1.0, 1.0 }
vec4 myVec4 = vec4(1.0, 2.0, 3.0, 4.0); // myVec4 = { 1.0, 2.0, 3.0, 4.0 }
float x = myVec4.x;       // x = 1.0;
vec3 myVec3_0 = myVec4.xyz; // myVec3_0 = { 1.0, 2.0, 3.0 }
vec3 myVec3_1 = myVec4.rgb; // myVec3_1 = { 1.0, 2.0, 3.0 }
vec3 myVec3_2 = myVec4.zyx; // myVec3_2 = { 3.0, 2.0, 1.0 }
vec3 myVec3_3 = myVec4.xxx; // myVec3_3 = { 1.0, 1.0, 1.0 }
```

矩阵

在 GLSL 内可构造 mat[2..4] 来表示 2 阶到 4 阶的矩阵。

矩阵构造有如下的规则：

- 若只为矩阵构造器提供了一个标量，则该值会构造矩阵对角线上的值
- 矩阵可以由多个向量构造
- 矩阵可以由单个标量从左到右进行构造

```
mat4 marixt4x4 = mat4(1.0); // marixt4x4 = { 1.0, 0.0, 0.0, 0.0,
                            //                 0.0, 1.0, 0.0, 0.0
                            //                 0.0, 0.0, 1.0, 0.0
                            //                 0.0, 0.0, 0.0, 1.0 }

vec2 col1 = vec2(1.0, 0.0);
vec2 col2 = vec2(1.0, 0.0);

mat2 matrix2x2 = mat2(coll1, col2);

// GLSL 是列矩阵存储，因此构造时，构造器会按照列顺序进行填充
mat3 matrix3x3 = mat3(0.0, 0.0, 0.0,   // 第一列
                      0.0, 0.0, 0.0,   // 第二列
                      0.0, 0.0, 0.0);  // 第三列
```

注意：为避免 implicit padding，引擎规定若要使用 Uniform 限定符的矩阵，必须是 4 阶矩阵，2 阶和 3 阶的矩阵不可作为 Uniform 变量。

矩阵的访问：

矩阵可以通过索引访问不同的列：

```
mat2 matrix2x2 = mat2(0.0, 0.0, 0.0, 0.0);
vec4 myVec4 = vec4(matrix2x2[0], matrix2x2[1]);
vec2 myVec2 = matrix2x2[0];

// 访问第一列的第一个元素
float value = matrix2x2[0][0];
matrix2x2[1][1] = 2.0;
```

结构体

结构体的形成和 C 语言类似，可由不同数据类型聚合而成：

```
struct myStruct
{
  vec4 position;
  vec4 color;
  vec2 uv;
};
```

构造结构体的代码示例如下：

```
myStruct structVar = myStruct(vec4(0.0, 0.0,0.0,0.0), vec4(1.0, 1.0, 1.0, 1.0), vec2(0.5, 0.5));
```

结构体支持赋值（=）和比较（==，!=）运算符，但要求两个结构体拥有相同的类型且组件分量（component-wise）都必须相同。

数组

数组的用法和 C 语言类似，规则如下：

- 数组必须声明长度
- 数组不能在声明的同时初始化
- 数组必须由常量表达式初始化
- 数组不能用 `const` 修饰
- 不支持多维数组

数组声明和初始化的代码示例如下：

```
float array[4];
for(int i =0; i < 4; i ++)
{
    array[i] = 0.0;
}
```

语句

控制流程

GLSL 支持标准的 C/C++ 控制流程，包括：

- `if-else/switch-case`
- `for/while/do-while`
- `break/continue/return`
- 没有 `goto`，若要跳出可使用 `discard`。该语句仅在片元着色器下有效，需要注意的是使用该语句会导致管线放弃当前片元，不会写入帧缓存

`if-else` 的用法和 C 语言一致，代码示例如下：

```
if(v_uvMode >= 3.0) {
    i.uv = v_uv0 * v_uvSizeOffset.xy + v_uvSizeOffset.zw;
} else if (v_uvMode >= 2.0) {
  i.uv = fract(v_uv0) * v_uvSizeOffset.xy + v_uvSizeOffset.zw;
} else if (v_uvMode >= 1.0) {
  i.uv = evalSlicedUV(v_uv0) * v_uvSizeOffset.xy + v_uvSizeOffset.zw;
} else {
  i.uv = v_uv0;
}
```

在 GLSL 中，循环变量必须是常量或者编译时已知，代码示例如下：

```
const float value = 10.;
for(float i = 0.0; i < value; i ++){
    ...
}
```

错误示例：

```
float value = 10.;
// 错误，value 不为常量
for(float i =0.0; i < value; i ++){
    ...
}
```

函数

GLSL 的函数由 **返回值**、**函数名** 和 **参数** 构成，其中返回值和函数名是必须的。若无返回值，需要使用 `void` 代替。

> **注意**：GLSL 的函数不能递归。

代码示例如下：

```
void scaleMatrix (inout mat4 m, float s){
  m[0].xyz *= s;
  m[1].xyz *= s;
  m[2].xyz *= s;
}
```

限定符

存储限定符

存储限定符用于描述变量在管线中的作用。

限定符	说明
<none:default>	无限定符或者使用 `default`，常用语局部变量，函数参数
const	编译时为常量或作为参数时只读
attribute	应用程序和顶点着色器间通信，用于确定顶点格式
uniform	应用程序和着色器之间交互数据。在顶点着色器和片元着色器中保持一致
varying	顶点着色器传输给片元着色器的插值

uniform

在一个渲染过程内声明的 `uniform` 不能重复。例如在顶点着色器中定义了变量 `variableA`，`variableA` 也会存在于片元着色器且值相同，那么也就是 `variableA` 不能在片元着色器中再次定义。

引擎不支持离散声明的 `uniform` 变量，必须使用 UBO 并保持内存对齐，以避免 implicit padding。

varying

`varying` 是由顶点着色器输出并传输给片元着色器的变量。在管线的作用下，变量值并不会和顶点着色器输出的保持一致，而是由管线进行插值，这就可能会出现顶点输出的法线没有归一化的情况。此时需要手动归一化，代码示例如下：

```
// 归一化法线
vec3 normal = normalize(v_normal);
```

参数限定符

GLSL 中函数的参数限定符包括以下几种：

限定符	说明
<none:in>	缺省限定符，和 C 语言的值传递类似，指明传入的参数传递的是值，函数内不会修改传入的值
inout	类似于 C 语言的引用，参数的值会传入函数并返回函数内修改的值
out	参数的值不会传入函数，由函数内部修改并返回修改后的值

精度限定符

GLSL 引入了精度限定符，用于指定整型或浮点型变量的精度。精度限定符可使着色器的编写者明确定义着色器变量计算时使用的精度。在 在 Shader 头部声明的精度应用于整个 Shader，是所有基于浮点型的变量的默认精度，同时也可以定义单个变量的精度。在 Shader 中如果没有指定默认精度，则所有的整型和浮点型变量都采用高精度计算。

GLSL 支持的精度限定符包括以下几种：

限定符	说明
highp	高精度。 浮点型精度范围为 $[-2^{62}, 2^{62}]$ 整型精度范围为 $[-2^{16}, 2^{16}]$。
mediump	中精度。 浮点型精度范围为 $[-2^{14}, 2^{14}]$ 整型精度范围为 $[-2^{10}, 2^{10}]$。
lowp	低精度。 浮点型精度范围为 $[-2^{8}, 2^{8}]$ 整型精度范围为 $[-2^{8}, 2^{8}]$。

代码示例如下：

```
highp mat4 cc_matWorld;
mediump vec2 dir;
lowp vec4 cc_shadowColor;
```

预处理宏定义

GLSL 允许定义和 C 语言类似的宏定义。

预处理宏定义允许着色器定义多样化的动态分支，确定最终的渲染效果。

在 GLSL 中使用预处理宏定义的代码示例如下：

```
#define
#undef
#if
#ifdef

#ifndef
#else
#elif
#endif
```

下方代码示例若 `USE_VERTEX_COLOR` 条件为真，则声明一个名为 `v_color` 的四维向量：

```
#if USE_VERTEX_COLOR
  in vec4 v_color;
#endif
```

预处理宏定义和引擎的交互部分可参考： 预处理宏定义

> **注意**：在引擎中，材质的预处理宏定义在材质初始化完成后便不能修改。如果需要修改，请使用 `Material.initialize` 或 `Material.reset` 方法。代码示例可参考：程序化使用材质

预处理宏定义

预处理宏定义

为了更好地管理代码内容，Cocos Shader 提供了预处理宏机制，它有几个特性：

1. 不同组合的宏会生成不同的代码
2. 生成的代码无冗余、执行高效
3. 使用过的宏定义会显示在材质面板上，方便调试
4. 以 `CC_` 开头的宏不会显示在材质面板上

以默认的 Surface Shader 为例，当它被材质使用时，你在材质的 **属性检查器** 中，可以看到如下图所示的宏开关：

着色器片段（Chunk）

着色器片段（Chunk）

着色器片段（Chunk）是一种跨文件代码引用机制，使着色器代码片段可以在不同的文件之间进行复用。

着色器片段的语法基于 **GLSL 300 ES**，在资源加载时会进行预编译，生成目标 Shader 代码。

创建着色器片段

在 **资源管理器** 面板中点击右键，选择 **创建 -> 着色器片段（Chunk）**：

内置全局 Uniform

Cocos Shader 内置全局 Uniform

要在 Cocos Shader 中使用内置变量 Uniform，需要包含对应的着色器片段（Chunk）即可，如下代码所示：

```
//local uniforms
#include <builtin/uniforms/cc-local>
//global uniforms
#include <builtin/uniforms/cc-global>
```

以下是常用内置变量，按所在着色器片段进行分组，列表如下：

cc-local.chunk

Name	Type Info

Name	Type	Info
cc_matWorld	mat4	模型空间转世界空间矩阵
cc_matWorldIT	mat4	模型空间转世界空间逆转置矩阵

cc-global.chunk

Name	Type	Info
cc_time	vec4	x：游戏运行时间（秒） y：帧时间（秒） z：游戏运行帧数 w：未使用
cc_screenSize	vec4	xy：屏幕尺寸 zw：屏幕尺寸倒数
cc_screenScale	vec4	xy：屏幕缩放 zw：屏幕缩放倒数
cc_nativeSize	vec4	xy：实际着色缓冲的尺寸 zw：实际着色缓冲的尺寸倒数
cc_matView	mat4	视图矩阵
cc_matViewInv	mat4	视图逆矩阵
cc_matProj	mat4	投影矩阵
cc_matProjInv	mat4	投影逆矩阵
cc_matViewProj	mat4	视图投影矩阵
cc_matViewProjInv	mat4	视图投影逆矩阵
cc_cameraPos	vec4	xyz：相机位置 w：combineSignY
cc_exposure	vec4	x：相机曝光 y：相机曝光倒数 z：是否启用 HDR w：HDR 转 LDR 缩放参数
cc_mainLitDir	vec4	xyz：主方向光源方向 w：是否启用阴影
cc_mainLitColor	vec4	xyz：主方向光颜色 w：主方向光强度
cc_ambientSky	vec4	xyz：天空颜色 w：亮度
cc_ambientGround	vec4	xyz：地面反射光颜色 w：环境贴图 Mipmap 等级

cc-environment.chunk

Name	Type	Info
cc_environment	samplerCube	IBL 环境贴图

cc-forward-light.chunk

Name	Type	Info
cc_sphereLitPos[MAX_LIGHTS]	vec4	xyz：球面光位置
cc_sphereLitSizeRange[MAX_LIGHTS]	vec4	x：球光尺寸 y：球光范围
cc_sphereLitColor[MAX_LIGHTS]	vec4	xyz：球光颜色 w：球光强度
cc_spotLitPos[MAX_LIGHTS]	vec4	xyz：聚光灯位置
cc_spotLitSizeRangeAngle[MAX_LIGHTS]	vec4	x：聚光灯尺寸 y：聚光灯范围 z：聚光灯角度
cc_spotLitDir[MAX_LIGHTS]	vec4	xyz：聚光灯方向
cc_spotLitColor[MAX_LIGHTS]	vec4	xyz：聚光灯颜色 w：聚光灯强度

cc-shadow.chunk

Name	Type	Info
cc_matLightPlaneProj	mat4	平面阴影的变换矩阵
cc_shadowColor	vec4	阴影颜色

完整常量列表，请参考 **internal/builtin/uniforms/** 目录。

公共函数库

公共函数库

可以在 **资源管理器/internal/chunks/common/** 文件夹下找到不同分类的函数库头文件。

公共库中的函数不依赖任何内部数据，可以当作工具函数直接使用。

使用示例

```
#include <common/color/aces>
#include <common/data/packing>
```

目录与功能应表

文件夹名	函数用途	主要文件
color	色彩相关功能（颜色空间、tone-mapping 等）	aces, gamma
data	数据相关功能（压缩解压缩等）	packing, unpack
debug	Debug View 相关功能	
effect	场景特效相关功能（水、雾等）	fog
lighting	光照相关功能（bxdf、反射、衰减、烘焙等）	brdf, bxdf, light-map
math	数学库（坐标变换、数值判定和运算等）	coordinates, transform
mesh	模型相关功能（材质转换、模型动画等）	material, vat-animation
shadow	阴影相关功能（pcf、pcss 等）	native-pcf

文件夹名	函数用途	主要文件
texture	贴图相关功能（采样、mip 计算等）	cubemap, texture-lod

Surface Shader 内部已经自动包含了常用的公共函数头文件，不需要再 include。

前向渲染与延迟渲染 Shader 执行流程

前向渲染与延迟渲染 Shader 执行流程

Cocos Creator 引擎支持 前向渲染和延迟渲染。因此，在 Shader 架构上，也要为这两种渲染流程做兼容，并且让用户感知不到。

内置的 Legacy Shader 都是 PBR 材质，它们在渲染时都遵守以下流程：

前向渲染

1. 调用 vs
2. 调用 fs -> surf -> 光照计算

延迟渲染

Buffer 阶段

1. 调用 vs
2. 调用 fs -> surf -> GBuffer

Lighting 阶段

1. 从 GBuffer 还原 StandardSurface 信息
2. 光照计算

可以看出，对于 PBR 材质来说，不管是前向渲染还是延迟渲染，用户能够控制的只有 vs 和 surf 函数。

这就统一了架构，使用户写的 Shader 可以不用修改就运行在 前向渲染管线 和 延迟渲染管线中。

表面着色器 - Surface Shader

表面着色器（Surface Shader）

Surface Shader 使用高效、统一的渲染流程，让用户以简洁的代码创建表面材质信息，指定用于组合的光照和着色模型。

相比于 传统着色器（Legacy Shader），它的优点是更易书写和维护，有更好的版本兼容性，也更不容易产生渲染错误。并且可以从统一流程中获取很多公共特性，如统一的全场景光照和渲染调试功能等。

在 Cocos Creator v3.7.2 中，Surface Shader 已作为默认的 builtin-standard，用户可以在创建菜单查看。原来的 builtin-standard 相关的着色流程，统一归类到了 internal/effects/legacy/ 目录。

由于基于统一的着色流程，使用 Surface Shader 的代价是无法对光照和着色运算进行大幅度的定制。如有极为特殊的需求，建议使用 Legacy Shader。

Surface Shader 相关内容列表：

内置 Surface Shader 导读

内置 Surface Shader 导读

Cocos Creator 3.7.2 版本开始， builtin-standard.effect 使用 Surface Shader 架构实现。

本文以 builtin-standard.effect 作为典型案例，讲解 Surface Shader 细节。

你可以属性 Surface Shader 结构定义、语法细节以及渲染流程。

下面的内容，建议配合 internal/effects/builtin-standard.effect 一起阅读。

基本结构

Surface Shader 代码通常由几个部分组成：

- 信息描述（`CCEffect`）：描述此 Shader 的技术、渲染过程组成部分，以及每个渲染过程使用的 Shader、渲染状态、属性等。
- 共享常量（`Shared UBOs`）：把 vs 和 fs 都需要用到的 uniforms 定义在一起，方便管理。
- 宏映射（`Macro Remapping`）：处理一些宏定义，以及映射一些内部宏，使其可以显示到材质面板上。
- 函数（`Surface Functions`）：用于声明表面材质信息相关的 Surface 函数。
- 组装器（`Shader Assembly`）：用于组装每个顶点着色器（Vertex Shader）和片元着色器（Fragment Shader）的代码模块。

可前往 Surface Shader 基本结构 了解更多详情。

CCEffect

一个物体被渲染到屏幕上，需要以下信息：

- 模型数据（顶点、UV、法线等）
- 光照信息
- 世界空间（旋转、平移、缩放）信息
- 绘制过程（材质）
- 渲染状态（材质）

- 纹理（材质）
- Uniform（材质）
- Shader 代码（材质）

其中模型数据、光照信息、世界空间信息与材质无关，而 纹理、Uniform、渲染状态、Shader 代码、绘制过程 则属于材质信息。

CCEffect 则描述了以上材质相关信息，并且与引擎渲染管线共同完成一个模型的渲染流程。

渲染技术（technique）

内置的 Surface Shader 实现了 `opaque` 和 `transparent` 两种渲染技术，前者用于渲染非透明物体，后者用于渲染半透明物体。

渲染过程（passes）

内置的 Surface Shader 的每一个技术，只有一个 pass，且均为 PBR。

抛开其余细节，我们可以看到整个 Surface Shader 的信息描述如下：

```
CCEffect %{
  techniques:
  - name: opaque
    passes:
    - vert: standard-vs
      frag: standard-fs
      properties: &props
        ...

  - name: transparent
    passes:
    - vert: standard-vs
      frag: standard-fs
      ...
      properties: *props
}%
```

Shader 入口（vert 与 frag）

`opaque` 和 `transparent` 在渲染效果上完全一致，差异仅仅是渲染状态。

可以看到，它们使用同样的 vert 和 frag 入口。

```
- vert: standard-vs
  frag: standard-fs
```

属性（properties）

由于 `opaque` 和 `transparent` 在渲染效果上完全一致，Shader 代码也是用的同一样，所以涉及到的属性也就完全一样。

所有渲染过程中用到的属性集中放在了 properties 字段里。关于属性的语法，可以查看 Pass 可选配置参数

配置复用

在属性配置中，可以看到， opaque 的属性定义为 `properties: &props`，而 `transparent` 的属性定义为 `properties: *props`。

这是一个属性命名和复用机制。

`properties: &props` 的含义是为当前 `properties` 起名为 `props`。

`properties: *props` 的含义是使用名为 `props` 的属性块填充当前 `properties`。

上面的配置结果为：`transparent` 直接使用 `opaque` 的 `properties` 字段。

用于特定阶段（phase）

默认情况下， Surface Shader 会参与到场景渲染阶段。 但也有一些特殊的场合，比如 阴影，反射探针烘焙 等。

针对这类需求，我们可以添加特定的渲染过程（pass），并标记参与的阶段（phase）来实现目的。

当 Cocos 引擎的渲染器在执行渲染时，会获取到材质中相应 phase 的 pass，用于渲染。 如果没有，表示此物体不参与这个阶段。

如 Surface Shader 中的：

- forward-add：用于附加光照阶段，当物体受主光源以外的光源影响时，会调用这个。
- shadow-caster：用于阴影图渲染阶段
- reflect-map：用于环境反射探针烘焙阶段

```
- &forward-add
    vert: standard-vs
    frag: standard-fs
    phase: forward-add
    ...
- &shadow-caster
    vert: shadow-caster-vs
    frag: shadow-caster-fs
    phase: shadow-caster
    ...
- &reflect-map
    vert: standard-vs
    frag: reflect-map-fs
    phase: reflect-map
    ...
```

如上面代码所示，phase 用于标记此 pass 的参与阶段。 而 `&forward-add`，`&shadow-caster`，`&reflect-map` 则是给这个 pass 起了一个名字，方便后面的技术对它进行复用。

比如，`transparent` 就直接复用了 `opaque` 的 `forward-add` 和 `shadow-caster` pass。

```
- name: transparent
    passes:
    - vert: standard-vs
        frag: standard-fs
        ...
        properties: *props
    - *forward-add
    - *shadow-caster
```

渲染状态

如开头所说，为了完成一个模型的渲染，除了定义好渲染过程，所需要的属性以外，还需要配合渲染状态。

渲染状态主要涉及 模板测试、深度测试、光栅器状态、透明混合等。

同样的渲染流程、属性、Shader 代码，配合不同的渲染状态，就可以实现不同的效果。

```
//深度、模板测试
depthStencilState:
    depthFunc: equal
    depthTest: true
    depthWrite: false
```

```
//透明混合状态
blendState:
    targets:
    - blend: true
        blendSrc: one
        blendDst: one
        blendSrcAlpha: zero
        blendDstAlpha: one
//光栅器状态
rasterizerState:
    cullMode: front
```

渲染状态会有一套默认值，在有需要的时候进行修改即可。

比如 `opaque` 与 `transparent` 就只有渲染状态的区别。

内嵌宏

```
    embeddedMacros: { CC_FORCE_FORWARD_SHADING: true }
```

有时候，我们想为某个pass单独开启或者关闭一些宏，可以使用 `embeddedMacros` 字段进行。

代码引用机制

Surface Shader 提供了两种代码块引用机制：头文件和CCProgram。详情请查看 include 机制。

共享常量

许多常量是 vs 和 fs 都会用到的，或者多个 technique 和 pass 都需要用到的，定义在一起方便管理。

共享常量本质上还是属于 Shader 代码片段，使用 GLSL 来编写。

```
CCProgram shared-ubos %{
  uniform Constants {
    vec4 tilingOffset;
    vec4 albedo;
    vec4 albedoScaleAndCutoff;
    vec4 pbrParams;
    vec4 emissive;
    vec4 emissiveScaleParam;
    vec4 anisotropyParam;
  };
}%
```

在后面的组装环节，只需要 `#include <shared-ubos>` 即可。

宏映射

关于宏映射详细信息，请参考 宏定义与重映射。

在 内置的 Surface Shader 中，使用 `CCProgram macro-remapping` 片段来组织所的宏映射工作，方便管理。

可以看到，在内置的 Surface Shader 中，使用 `#pragma define-meta` 将许多内置的宏重定向到了面板上。

```
CCProgram macro-remapping %{
  // ui displayed macros
  #pragma define-meta HAS_SECOND_UV
  #pragma define-meta USE_TWOSIDE
  #pragma define-meta IS_ANISOTROPY
  #pragma define-meta USE_VERTEX_COLOR

  #define CC_SURFACES_USE_SECOND_UV HAS_SECOND_UV
  #define CC_SURFACES_USE_TWO_SIDED USE_TWOSIDE
  #define CC_SURFACES_LIGHTING_ANISOTROPIC IS_ANISOTROPY
  #define CC_SURFACES_USE_VERTEX_COLOR USE_VERTEX_COLOR

  // depend on UI macros
#if IS_ANISOTROPY || USE_NORMAL_MAP
  #define CC_SURFACES_USE_TANGENT_SPACE 1
#endif

  // functionality for each effect
  #define CC_SURFACES_LIGHTING_ANISOTROPIC_ENVCONVOLUTION_COUNT 31
}%
```

Surface 函数段

在 Surface Shader 中，定义了两个 CCProgram 用于处理具体的Shader计算。

- CCProgram surface-vertex：用于处理 vs 相关计算
- CCProgram surface-fragment：用于处理 fs 相关计算

CCProgram surface-vertex

内置的 vs 流程基本上能满足 Surface Shader 的 vs 需求，导致 surface-vertex 非常简单，只做了少量的特殊处理。

我们以第二套 UV 的处理函数为例。

它先定义了 `CC_SURFACES_VERTEX_MODIFY_UV` 宏，然后实现了 `SurfacesVertexModifyUV` 方法。

```
#define CC_SURFACES_VERTEX_MODIFY_UV
void SurfacesVertexModifyUV(inout SurfacesStandardVertexIntermediate In)
{
    In.texCoord = In.texCoord * tilingOffset.xy + tilingOffset.zw;
    #if CC_SURFACES_USE_SECOND_UV
    In.texCoord1 = In.texCoord1 * tilingOffset.xy + tilingOffset.zw;
    #endif
}
```

这就是 Surface Shader 的核心机制，可以通过宏定义改写内部函数，在不修改内部源码的情况下实现特定的渲染需求。

具体的机制请参考使用宏定义实现函数替换 和 Surface Shader 内置可替换函数列表。

CCProgram surface-fragment

surface-fragment 主要实现了 PBR 计算时需要的表面材质信息填充。

宏开关

```
#if USE_ALBEDO_MAP
    uniform sampler2D albedoMap;
    #pragma define-meta ALBEDO_UV options([v_uv, v_uv1])
#endif
```

可以看到，在内置的 Surface Shader 中，所有的贴图，都被宏定义包裹起来，这样的好处就是可以根据需求关闭对应的宏，以提升性能。

材质面板可选择的宏

`#pragma define-meta` + 名称 + `options([item0,item1,....])` 可以定义一个供用户选择的宏。

以下面代码为例：

```
#pragma define-meta ALBEDO_UV options([v_uv, v_uv1])
```

材质面板上， ALBEDO_UV 会出现下拉选择框，Shader 编译时，会以用户选择值为准。

比如，用户如果选择了 `v_uv1`，这条语句编译出来的最终结果为：

```
#define ALBEDO_UV v_uv1
```

```
#if USE_ALPHA_TEST
    #pragma define-meta ALPHA_TEST_CHANNEL options([a, r])
#endif
```

ALPHA_TEST_CHANNEL 也是如此，默认使用 `a` 通道，但也可以选择 r 通道。

PBR 通道

```
#pragma define OCCLUSION_CHANNEL          r
#pragma define ROUGHNESS_CHANNEL          g
#pragma define METALLIC_CHANNEL           b
#pragma define SPECULAR_INTENSITY_CHANNEL a
```

Surface Shader 使用一张图作为 PBR 贴图，根据定义就可以知道，PBR 贴图各通道的含义：

- r通道：环境遮蔽
- r通道：粗糙度
- b通道：金属度
- a通道：高光强度

具体实现

与 surface-vertex 一样， surface-fragment 中也通过函数替换方式，实现 PBR 参数填充。

想要了解更多具体的机制请参考以下文章：

- 使用宏定义实现函数替换
- Surface Shader 内置可替换函数列表
- Surface Shader 执行流程

Shader 组装

上面提到的几个 CCProgram：

- shared-ubos
- macro-remapping
- surface-vertex
- surface-fragment

只是一些实现 Surface Shader 必要的组成部分，想要实现一个完整的 Surface Shader，还需要将这些部分，配合Surface Shader 内置的其它模块进行组装。

具体的组装机制请查看：Surface Shader 组装。

最后组装出来的 CCProgram，才是 CCEffect 部分引用的内容。

- CCProgram standard-vs
- CCProgram shadow-caster-vs
- CCProgram standard-fs
- CCProgram shadow-caster-fs
- CCProgram reflect-map-fs

Surface Shader 基本结构

Surface Shader 结构

一个典型的 Surface Shader 通常由四个主要部分构成：

1. CCEffect
2. 共享常量声明
3. 宏映射
4. 主体功能函数
5. Shader组装器：用于将上面4个部分与内置的Surface Shader 功能进行级任

1、CCEffect

CCEffect 用于描述 Surface Shader 的 techniques, pass , 属性以及渲染状态等信息。材质会根据 CCEffect 中的描述生成默认值，以及在材质面板上显示。

具体内容请参考 Cocos Shader 语法。

2、共享常量声明

共享常量声明会将所有 pass， vs 和 fs 都需要用到的常量写在一起，简化 Shader 编写。

这是一个可选项，但对于复杂的 Shader 来说，建议使用。以内置 Surface Shader 为例：

```
CCProgram shared-ubos %{
    uniform Constants {
        vec4 titlingOffset;
        ...
    }
}%
```

> 注意：并不一定非要叫 `shared-ubos` 和 `Constants`，只要方便记忆即可。

3、宏映射

Surface Shader 提供了大量的内置宏，这些宏默认是不显示在面板上的，如果想让这些宏显示在面板上，就需要重新映射它们的名字。

并且，你可以将一些宏集中放在这个代码片段中，方便管理，以内置 Surface Shader 为例：

```
CCProgram macro-remapping %{
    // ui displayed macros
    #pragma define-meta HAS_SECOND_UV
    #pragma define-meta USE_TWOSIDE
```

```
    ...
    #define CC_SURFACES_USE_SECOND_UV HAS_SECOND_UV
    #define CC_SURFACES_USE_TWO_SIDED USE_TWOSIDE
}%
```

更多详情请参考 宏定义与重映射。

注意：并不一定非要叫 `macro-remapping` 只要方便记忆即可。

4、函数块

Surface Shader 统一了渲染流程，同时也暴露了许函数可供用户自定义渲染细节。

为了方便管理，我们一般至少需要声明两个 CCProgram 代码片段，用于 vs 和 fs。

以内置的 Surface Shader 为例：

```
CCProgram surface-vertex %{
    ...
}%

CCProgram surface-fragment %{
    ...
}%
```

`surface-vertex` 用于 vs 相关函数的处理，`surface-fragment` 用于 fs 相关函数的处理。

在这两个代码片段中，通过宏定义机制，替换内部函数。也可以增加自己的 vs 到 fs 的输入。

详情请参考 使用宏定义实现函数替换， Vertex Shader 的输入值 和 Fragment Shader 的输入值。

5、Shader 组装器

Surface Shader 中，最后一个部分，是 Shader 的组装。

我们以内置的 Surface Shader 为例：

```
CCProgram standard-vs %{
    ...
}%

CCProgram shadow-caster-vs %{
    ...
}%

CCProgram standard-fs %{
    ...
}%

CCProgram shadow-caster-fs %{
    ...
}%
```

值得说明的是，上面三个部分（共享常量声明、宏映射 主体功能函数）的代码片段，可以有零个或者多个。最后根据需求组装出最后的 Shader， 供 Surface Shader 开头部分的 CCEffect 引用。

详情请参考 Surface Shader 组装。

Surface Shader 执行流程

Surface Shader 执行流程

Surface Shader 统一了着色流程，同时为 vs 和 fs 提供了大量的自定义函数，大家可以根据自己的需求，重写相关函数。

请参考 Surface Shader 内置函数 和 使用宏定义实现函数替换。

本文主要目的在于帮助开发者熟悉 Surface Shader 执行流程，弄清楚各函数调用时机。

函数入口

我们先看一下内置的 Surface Shader 文件的 CCEffect 部分：

```
CCEffect %{
  techniques:
  - name: opaque
    passes:
    - vert: standard-vs
      frag: standard-fs
    ...
}%
```

可以看到， 每一个 pass 的 vert 和 frag 没有指定专有的入口函数，这就表示，它所用到的 vs 和 fs 的入口函数为 main。

从 Surface Shader 结构 中，我们可以了解到， 在 Surface Shader 组装 环节，每一个 Surface Shader 会根据不同的 渲染用途，引入不同的头文件。 这个头文件就是我们的入口函数。

VS 主函数

以内置 Surface Shader 的 standard-vs 为例：

```
CCProgram standard-vs %{
    #include <shading-entries/main-functions/render-to-scene/vs>
}%
```

可以看到，它引入的是 render-to-scene 下面的 vs.chunk。

打开 render-to-scene/vs.chunk，可以看到，它只有一个 main 函数。代码和注释如下：

```
void main()
{
    //声明一个表面材质数据结构
    SurfacesStandardVertexIntermediate In;

    //获取顶点基本数据
    CCSurfacesVertexInput(In);
    //获取顶点动画数据
    CCSurfacesVertexAnimation(In);
    // ===== 本地坐标相关 ====
    //处理本地坐标，用户可用宏替换
    In.position.xyz = SurfacesVertexModifyLocalPos(In);
    //处理本地法线，用户可用宏替换
    In.normal.xyz = SurfacesVertexModifyLocalNormal(In);
    //处理本地切线，用户可用宏替换
    #if CC_SURFACES_USE_TANGENT_SPACE
        In.tangent = SurfacesVertexModifyLocalTangent(In);
    #endif
    //进一步处理自定义本地数据，用户可用宏替换
    SurfacesVertexModifyLocalSharedData(In);
```

```
    //==== 世界坐标相关======
    //进行世界坐标转换，并填充数据结构
    CCSurfacesVertexWorldTransform(In);
    //额外的处理世界坐标函数，用户可用宏替换
    In.worldPos = SurfacesVertexModifyWorldPos(In);

    //投影空间坐标
    In.clipPos = cc_matProj * cc_matView * vec4(In.worldPos, 1.0);
    //进一步处理投影空间坐标，用户可用宏替换
    In.clipPos = SurfacesVertexModifyClipPos(In);

    //其它一些数据变换
    vec3 viewDirect = normalize(cc_cameraPos.xyz - In.worldPos);
    In.worldNormal.w = dot(In.worldNormal.xyz, viewDirect) < 0.0 ? -1.0 : 1.0;
    //进一步处理世界法线，用户可用宏替换
    In.worldNormal.xyz = SurfacesVertexModifyWorldNormal(In);

    //进一步处理UV，用户可用宏替换
    SurfacesVertexModifyUV(In);
    //进一步处理自定义数据，用户可用宏替换
    SurfacesVertexModifySharedData(In);

    //其它变换
    //UV
    CCSurfacesVertexTransformUV(In);
    //雾效
    CCSurfacesVertexTransferFog(In);
    //阴影
    CCSurfacesVertexTransferShadow(In);
    //光照贴图UV
    CCSurfacesVertexTransferLightMapUV(In);

    //最终输出
    CCSurfacesVertexOutput(In);
}
```

FS 主函数

同样的，我们以内置 Surface Shader 的 standard-fs 为例：

```
CCProgram standard-fs %{
  #include <shading-entries/main-functions/render-to-scene/fs>
}%
```

render-to-scene/fs.chunk 文件内容如下：

```
#if (CC_PIPELINE_TYPE == CC_PIPELINE_TYPE_FORWARD || CC_FORCE_FORWARD_SHADING)
  #include <shading-entries/main-functions/render-to-scene/pipeline/forward-fs>
#elif CC_PIPELINE_TYPE == CC_PIPELINE_TYPE_DEFERRED
  #include <shading-entries/main-functions/render-to-scene/pipeline/deferred-fs>
#endif
```

可以看到它区分了 forward 和 deferred 管线。

forward-fs

```
//定义颜色输出目标
layout(location = 0) out vec4 fragColorX;

void main()  {
    #if CC_DISABLE_STRUCTURE_IN_FRAGMENT_SHADER
    //获取基本颜色和透明度，用户可用宏替换
    vec4 color = SurfacesFragmentModifyBaseColorAndTransparency();
    #else
    //获取表面材质数据
    SurfacesMaterialData surfaceData;
    CCSurfacesFragmentGetMaterialData(surfaceData);

    //计算阴影参数
    vec2 shadowBias = vec2(0.0);
    ...


    //计算雾效参数
    #if !CC_FORWARD_ADD
        float fogFactor = 1.0;
    #endif

    //计算光照
    LightingResult lightingResult;
    CCSurfacesLighting(lightingResult, surfaceData, shadowBias);


    //渲染调试相关
    ...

    //像素着色计算
    vec4 color = CCSurfacesShading(surfaceData, lightingResult);

    ...

    //颜色输出
    #if CC_USE_RGBE_OUTPUT
        fragColorX = packRGBE(color.rgb); // for reflection-map
        return;
    #endif

    //HDR，LinearToSRGB 等最终运算
    #if CC_USE_HDR
        #if CC_USE_DEBUG_VIEW == CC_SURFACES_DEBUG_VIEW_COMPOSITE_AND_MISC && CC_SURFACES_ENABLE_DEBUG_VIEW
        if (IS_DEBUG_VIEW_COMPOSITE_ENABLE_TONE_MAPPING)
        #endif
        color.rgb = ACESToneMap(color.rgb);
    #endif
    #if CC_USE_DEBUG_VIEW == CC_SURFACES_DEBUG_VIEW_COMPOSITE_AND_MISC
        if (IS_DEBUG_VIEW_COMPOSITE_ENABLE_GAMMA_CORRECTION)
    #endif
    color.rgb = LinearToSRGB(color.rgb);

    #if !CC_FORWARD_ADD && CC_USE_FOG != CC_FOG_NONE
        CC_APPLY_FOG_BASE(color, fogFactor);
    #endif

    fragColorX = CCSurfacesDebugDisplayInvalidNumber(color);
}
```

deferred-fs

延迟渲染由于分成两个阶段：GBuffer 和 Lighting。

在 GBuffer 阶段主要是填充各个渲染目标，只需要收集对应的材质表面数据即可，代码如下：

```
//GBuffer 0, 1, 2
layout(location = 0) out vec4 fragColor0;
layout(location = 1) out vec4 fragColor1;
layout(location = 2) out vec4 fragColor2;
```

```
void main () {
    //收集表面材质数据
    SurfacesMaterialData surfaceData;
    CCSurfacesFragmentGetMaterialData(surfaceData);

    //填充 GBuffer
    fragColor0 = CCSurfacesDeferredOutput0(surfaceData);
    fragColor1 = CCSurfacesDeferredOutput1(surfaceData);
    fragColor2 = CCSurfacesDeferredOutput2(surfaceData);

    //调试渲染相关
    #if CC_USE_DEBUG_VIEW == CC_SURFACES_DEBUG_VIEW_SINGLE && CC_SURFACES_ENABLE_DEBUG_VIEW
        vec4 debugColor = vec4(0.0, 0.0, 0.0, 1.0);
        CCSurfacesDebugViewMeshData(debugColor);
        CCSurfacesDebugViewSurfaceData(debugColor, surfaceData);
        if (IS_DEBUG_VIEW_ENABLE_WITH_CAMERA) {
        fragColor0 = debugColor;
        }
    #endif
}
```

延迟渲染的 Lighting 阶段，是受引擎渲染管线控制的，会统一使用 GBuffer 进行光照计算，可参考 internal/effects/deferred-lighting.effect。

其余的渲染用途主函数同理，可以到 internal/chunks/shading-entries/ 目录下查看。

> 提示：可以被替换的代码，都以 Surface###Modify### 方式命名。

---

include 机制

include 机制

头文件

Surface Shader 中，包含了多个光照模型，如 standard，toon 等。

同时，我们拥有不同的渲染用途，如 render-to-scene， render-to-shadowmap 等。

不同的光照模型拥有不同的表面材质数据结构和光照计算函数，不同的渲染用途之间，流程上会有细微的差别。

如果为不同的光照模型和不同的渲染用途编写独立的代码，会使整个架构很难维护。

因此，在 Surface Shader 中，我们将不同光照模型的方法和结构体，以相同的名字定义在不同的头文件中，**再使用 include 进行组合**。

这样就可以很容易实现，在同样的流程下，通过切换头文件实现不同的光照模型。

比如，在下面的代码中，我们通过 **#include** 引入 **SurfacesMaterialData** 的定义，函数中使用的结构体，就是 PBR 表面材质数据结构体。

```
//PBR 表面材质
#include <surfaces/data-structures/standard>
#define CC_SURFACES_FRAGMENT_MODIFY_SHARED_DATA
void SurfacesFragmentModifySharedData(inout SurfacesMaterialData surfaceData)
{
    // set user-defined data to surfaceData
}
```

如果我们更换它的头文件：

```
//toom 表面材质
#include <surfaces/data-structures/toon>
#define CC_SURFACES_FRAGMENT_MODIFY_SHARED_DATA
void SurfacesFragmentModifySharedData(inout SurfacesMaterialData surfaceData)
{
    // set user-defined data to surfaceData
}
```

此时的 SurfacesMaterialData 使用的就是 surfaces/data-structures/toon 中定义的结构体。

CCProgram

在 Surface Shader 中，CCProgram 包含的内容就是纯粹的 GLSL 代码内容。 我们可以将 Shader 分割到不同的 CCProgram 里形成代码片段，然后通过 `include` 进行引用。

以下面代码为例：

```
CCProgram shared-ubos %{
    ...
}%
CCProgram macro-remampping %{
    ...
}%

CCProgram vs %{
    #include<shared-ubos>
    #include<macro-remapping>
}%

CCProgram fs %{
    #include<shared-ubos>
    #include<macro-remapping>
}%
```

> 注意：使用 `include` 引用 `CCProgram` 定义的代码片段时，仅限本文件内。

宏定义与重映射

宏定义与重映射

Surface Shader 内部计算时会用到一些宏开关，这些宏以 `CC_SURFACES_` 开头。

> 注意：以 `CC_SURFACES_` 开头的宏不会出现在材质面板上。

以下是完整的宏列表：

宏名	类型	含义
CC_SURFACES_USE_VERTEX_COLOR	BOOL	是否使用顶点色
CC_SURFACES_USE_SECOND_UV	BOOL	是否使用2uv
CC_SURFACES_USE_TWO_SIDED	BOOL	是否使用双面法线，用于双面光照
CC_SURFACES_USE_TANGENT_SPACE	BOOL	是否使用切空间（使用法线图或各向异性时必须开启）
CC_SURFACES_TRANSFER_LOCAL_POS	BOOL	是否在 FS 中访问模型空间坐标
CC_SURFACES_LIGHTING_ANISOTROPIC	BOOL	是否开启各向异性材质
CC_SURFACES_LIGHTING_ANISOTROPIC_ENVCONVOLUTION_COUNT	UINT	各向异性环境光卷积采样数，为 0 表示关闭卷积计算，仅当各向异性开启时有效
CC_SURFACES_LIGHTING_USE_FRESNEL	BOOL	是否通过相对折射率 ior 计算菲涅耳系数

宏名	类型	含义
CC_SURFACES_LIGHTING_TRANSMIT_DIFFUSE	BOOL	是否开启背面穿透漫射光（如头发、叶片、耳朵等）
CC_SURFACES_LIGHTING_TRANSMIT_SPECULAR	BOOL	是否开启背面穿透高光（如水面、玻璃折射等）
CC_SURFACES_LIGHTING_TRT	BOOL	是否开启透射后内部镜面反射出的光线（如头发材质等）
CC_SURFACES_LIGHTING_TT	BOOL	是否开启透射后内部漫反射出的光线（用于头发材质）
CC_SURFACES_USE_REFLECTION_DENOISE	BOOL	是否开启环境反射除噪，仅 legacy 兼容模式下生效
CC_SURFACES_USE_LEGACY_COMPATIBLE_LIGHTING	BOOL	是否开启 legacy 兼容光照模式，可使渲染效果和 legacy/standard.effect 完全一致，便于升级

注意： 如果未定义这些宏，系统内部会自动定义为默认值 0；

搜索 `CCProgram macro-remapping` 一段，可以看到内容有如下三部分组成：

使用宏定义实现函数替换

使用宏定义实现函数替换

有时候需要制作一些公用的函数供不同的需求场景使用， 以降低代码量和维护成本。

但流程里的某些函数，我们希望在某些场合替换为特定版本，以满足特定需求。

可以通过宏定义机制来完成。

```
//example.chunk
#ifndef CC_USER_MODIFY_SOMETHING
void ModifySomething(){
    //do something here
}
#endif

void Before(){
    //do something here
}

void After(){
    //do something here
}

void myFunc(){
    Before();
    ModifySomething();
    After();
}
```

可以看到，在上面的代码中，myFunc 拥有完整的调用流程。如果想修改 ModifySomething 的实现，只需要在 #include example.chunk 之前，定义 CC_USER_MODIFY_SOMETHING 宏，并实现自己的 ModifySomething 函数即可。

```
#define CC_USER_MODIFY_SOMETHING
void ModifySomething(){
    //do what you want
}

#include <example.chunk>
```

注意，重载函数定义要放在 include 前面。

这个机制在 Surface Shader 系统中被广泛应用，比如前面提到的 lighting-models/includes/common 中的 SurfacesFragmentModifySharedData 函数。

```
// user-defined-common-surface.chunk:
// base surface function
#ifndef CC_SURFACES_FRAGMENT_MODIFY_SHARED_DATA
#define CC_SURFACES_FRAGMENT_MODIFY_SHARED_DATA
void SurfacesFragmentModifySharedData(inout SurfacesMaterialData surfaceData)
{
    ..................
}
#endif

// effect
// this function needs overriding
#define CC_SURFACES_FRAGMENT_MODIFY_SHARED_DATA
void SurfacesFragmentModifySharedData(inout SurfacesMaterialData surfaceData)
{
    ..............
}
// base functions should place after override functions
#include <user-defined-common-surface.chunk>
```

可替换的内置函数

Surface Shader 可替换的内置函数

Surface Shader 统一了着色流程，同时向用户提供了大量的自定义函数，大家可以根据自己的需求，利用宏机制，重写相关函数。

1、原理

Surface Shader 提供的自定义函数，在内部都有一个默认版本，并且在适合的时候被调用。可以参考 Surface Shader 执行流程了解详情。

这些函数通常以 `Surfaces+Shader类型名+Modify+属性` 方式全名，比如：

- SurfacesVertexModifyLocalPos
- SurfacesVertexModifyLocalNormal
- SurfacesVertexModifyLocalTangent

所有函数可以在 `internal/chunks/surfaces/default-functions/` 中查看。

如果你想替换某函数的实现，可以通过预定义该函数对应的宏来完成。

比如，可以预先定义 `CC_SURFACES_VERTEX_MODIFY_WORLD_POS` 宏，让 Surface Shader 使用你定义的函数来计算世界坐标，示例代码如下：

```
#define CC_SURFACES_VERTEX_MODIFY_WORLD_POS
vec3 SurfacesVertexModifyWorldPos(in SurfacesStandardVertexIntermediate In)
{
  vec3 worldPos = In.worldPos;
  worldPos.x += sin(cc_time.x * worldPos.z);
  worldPos.y += cos(cc_time.x * worldPos.z);
  return worldPos;
}
```

如果对 Surface Shader 中的函数替换机制不熟悉，可以先参考 使用宏定义实现函数替换。

用这种方式的好处是可以方便地扩展多种不同的材质数据结构、光照模型和渲染用途，并且不用对内置的 Surface Shader 流程进行修改。

2、VS 对应的常用函数列表

可用宏替换的 VS 内置函数定义在：internal/chunks/surfaces/default-functions/common-vs.chunk 文件中。

在 VS 中的内置函数参数均为 `SurfacesStandardVertexIntermediate` 结构体，存放的是 VS 输入输出的数据。用户无需关心具体的顶点输入输出流程处理，只需要聚焦到某个数据的修改即可。

预先定义宏	对应的函数定义	对应的材质模型	功能说明
CC_SURFACES_VERTEX_MODIFY_LOCAL_POS	vec3 SurfacesVertexModifyLocalPos	Common	返回修改后的模型空间坐标
CC_SURFACES_VERTEX_MODIFY_LOCAL_NORMAL	vec3 SurfacesVertexModifyLocalNormal	Common	返回修改后的模型空间法线
CC_SURFACES_VERTEX_MODIFY_LOCAL_TANGENT	vec4 SurfacesVertexModifyLocalTangent	Common	返回修改后的模型空间切线和镜像法线标记
CC_SURFACES_VERTEX_MODIFY_LOCAL_SHARED_DATA	void SurfacesVertexModifyLocalSharedData	Common	如果某些贴图和计算需要在多个材质节点中使用，可在此函数中进行，在世界变换前调用，直接修改 SurfaceStandardVertexIntermediate 结构体内的三个Local参数
CC_SURFACES_VERTEX_MODIFY_WORLD_POS	vec3 SurfacesVertexModifyWorldPos	Common	返回修改后的世界空间坐标（世界空间动画）
CC_SURFACES_VERTEX_MODIFY_CLIP_POS	vec4 SurfacesVertexModifyClipPos	Common	返回修改后的剪裁（NDC）空间坐标（通常用于修改深度）
CC_SURFACES_VERTEX_MODIFY_UV	void SurfacesVertexModifyUV	Common	修改结构体内的 UV0 和 UV1 （使用 tiling 等）
CC_SURFACES_VERTEX_MODIFY_WORLD_NORMAL	vec3 SurfacesVertexModifyWorldNormal	Common	返回修改后的世界空间法线（世界空间动画）
CC*SURFACES_VERTEX_MODIFY*SHARED_DATA	void SurfacesVertexModify SharedData	Common	如果某些贴图和计算需要在多个材质节点中使用，可在此函数中进行，直接修改 SurfaceStandardVertexIntermediate 结构体内的参数，减少性能耗费

3、FS 对应的常用函数列表

FS 由 PBR 和 Toon 两个部分组成，分别在下面两个文件中：

- internal/chunks/surfaces/default-functions/standard-fs.chunk
- internal/chunks/surfaces/default-functions/toon-vs.chunk

FS 中的函数，大部分为无参函数，用户需要结合 FS 输入值 来做处理。对于一些特殊用途的函数，也提供了对应的参数。具体属于哪种情况请参考函数定义。

预先定义宏	对应的函数定义	对应的材质模型	功能说明
CC*SURFACES_FRAGMENT_MODIFY* BASECOLOR_AND_TRANSPARENCY	vec4 SurfacesFragmentModify BaseColorAndTransparency	Common	返回修改后的基础色（rgb 通道）和透明值（a 通道）
CC_SURFACES_FRAGMENT_ALPHA_CLIP_ONLY	vec4 SurfacesFragmentModify AlphaClipOnly	Common	不需要获取颜色仅处理透贴的Pass中使用。如渲染到阴影图等，不重载此函数可能导致阴影没有透贴效果
CC*SURFACES_FRAGMENT_MODIFY* WORLD_NORMAL	vec3 SurfacesFragmentModify WorldNormal	Common	返回修改后的像素法线（通常是法线贴图）
CC*SURFACES_FRAGMENT_MODIFY*SHARED_DATA	void SurfacesFragmentModify SharedData	Common	若某些贴图和计算需要在多个材质节点中使用，可在此函数中进行，直接修改 Surface 结构体内的参数，减少性能耗费，类似legacy shader中的surf()函数。**需要在定义函数前 include 必要的头文件**
CC*SURFACES_FRAGMENT_MODIFY* WORLD_TANGENT_AND_BINORMAL	void SurfacesFragmentModify WorldTangentAndBinormal	Standard PBR	修改 Surface 结构体内的世界切空间向量
CC*SURFACES_FRAGMENT_MODIFY*EMISSIVE	vec3 SurfacesFragmentModify Emissive	Standard PBR	返回修改后的自发光颜色
CC*SURFACES_FRAGMENT_MODIFY*PBRPARAMS	vec4 SurfacesFragmentModify PBRParams	Standard PBR	返回修改后的 PBR 参数（ao, roughness, metallic, specularIntensity）
CC*SURFACES_FRAGMENT_MODIFY* ANISOTROPY_PARAMS	vec4 SurfacesFragmentModify AnisotropyParams	Standard PBR	返回修改后的各向异性参数（rotation, shape, unused, unused）
CC*SURFACES_FRAGMENT_MODIFY* BASECOLOR_AND_TOONSHADE	void SurfacesFragmentModify BaseColorAndToonShade	Toon	修改卡通渲染基础色
CC*SURFACES_FRAGMENT_MODIFY* TOON_STEP_AND_FEATHER	vec4 SurfacesFragmentModify ToonStepAndFeather	Toon	返回修改后的参数
CC*SURFACES_FRAGMENT_MODIFY* TOON_SHADOW_COVER	vec4 SurfacesFragmentModify ToonShadowCover	Toon	返回修改后的参数
CC*SURFACES_FRAGMENT_MODIFY* TOON_SPECULAR	vec4 SurfacesFragmentModify ToonSpecular	Toon	返回修改后的参数
CC_SURFACES_LIGHTING_MODIFY_FINAL_RESULT	void SurfacesLightingModifyFinalResult	Common	自定义光照模型，可以在之前计算的光照结果上再次修改，比如添加轮廓光等。**需要在定义函数前 include 必要的头文件**

渲染用途

渲染用途

默认情况下，Surface Shader 都是以输出到屏幕上，显示场景为主。

但有时候，我们也有一些特殊需要，比如:

- 渲染为阴影贴图（ShadowMap）
- 渲染为环境反射图（Reflection Probe）

不同的渲染用途，有不同的渲染流程和细节，因此需要特殊处理。

Surface Shader 框架中，预定义了常见的不同用途的流程，在 **资源管理器/internal/chunks/shading-entries/main-functions/** 目录下可以找到。

以下是内置的渲染用途：

常用的渲染用途	文件位置	备注
渲染到场景（默认）	render-to-scene	
渲染到阴影贴图	render-to-shadowmap	
渲染到环境贴图	render-to-reflectmap	可选
渲染卡通描边	misc/silhouette-edge	
渲染天空	misc/sky	
后期处理或通用计算 Pass	misc/quad	引擎预留

只需要在 Surface Shader 组装 环节引用对应的头文件，就可以完成渲染流程。

比如，在 internal/effects/builtin-standard.effect 中，我们可以看到应用案例：

```
CCProgram standard-vs %{
    ...
    #include <shading-entries/main-functions/render-to-scene/vs>
}%

CCProgram shadow-caster-vs %{
    ...
    #include <shading-entries/main-functions/render-to-shadowmap/vs>
```

```
}%

CCPogram standard-fs %{
    ...
    #include <shading-entries/main-functions/render-to-scene/fs>
}%

CCProgram shadow-caster-fs %{
    ...
    #include <shading-entries/main-functions/render-to-shadowmap/fs>
}%

CCProgram reflect-map-fs %{
    ...
    #include <shading-entries/main-functions/render-to-shadowmap/fs>
}%
```

光照模型

光照模型

光照模型用于说明物体表面与是如何对光线产生影响和作用的。

目前支持的光照模型如下：

光照模型名称	说明
standard	PBR 光照，支持 GGX BRDF 分布的各向同性与各向异性光照，支持卷积环境光照
toon	简单的卡通光照，阶梯状的光照效果

与光照模型相关的内置 Shader 函数放在了 internal/chunks/lighting-models/includes/ 目录下，在 Surface Shader 组装 时，通过 include 引入对应文件，就能完成光照计算。

表面材质数据结构

表面材质数据结构

什么是表面材质数据

Surface Shader 中，根据不同光照模型的需求，提供了对应的表面材质数据结构 **SurfaceMaterialData** 。

表面材质数据结构体定义了一系列用于计算物体表面最终颜色的物理参数，如 反照率、粗糙度等。

> 注意：材质数据模型与光照模型必须关联使用

表面材质数据类型

材质数据类型	说明
standard	粗糙度和金属性描述的标准 PBR 材质，和 SP、Blender、Maya 等软件中的材质节点类似
toon	简单的卡通材质，有多种 shade 颜色处理

如何使用

对应的表面材质数据结构存放在 **internal/chunks/data-structures/** 目录下。

只需要 include 对应的表面材质数据结构头文件，就可以使用不同类别的数据结构。

```
//PBR 表面材质
#include <surfaces/data-structures/standard>
// toon 表面材质
//#include <surfaces/data-structures/toon>
#define CC_SURFACES_FRAGMENT_MODIFY_SHARED_DATA
void SurfacesFragmentModifySharedData(inout SurfacesMaterialData surfaceData)
{
    // set user-defined data to surfaceData
}
```

着色器类别

Shader 类型

渲染是由不同的着色器来完成的，有处理顶点的、有处理像素的、有用于通用计算的。

在 Surface Shader 架构中，为了良好的可读性和可维护性，不同的 Shader 类型会有一个约定的代码标识， 如下表所示：

着色器阶段	对应的 Surface Shader 代码标识
Vertex Shader	vs
Fragment Shader	fs
Computer Shader	cs

你可以在内置的 effect 和 chunk 文件中发现许多文件以 xxxx-vs 或者 xxxx-fs 命名。

而在 effect 和 chunk 文件中， 也有许多类似 `CCProgram xxx-vs %{}%` 和 `CCProgram xxx-fs %{}%` 的代码片段定义。

比如：

```
CCProgram standard-vs %{
    //...
}%

CCProgram standard-fs %{
    //...
}%
```

用户在编写自己的 Shader 时，最好也遵守这个约定，以维持源码的可读性与维护性。

组装器

Surface Shader 组装

Shader片段组装器

在内置的 Surface Shader 文件中，你会看到下面的代码片段：

```
CCProgram standard-vs %{
    //includes
}%

CCProgram shadow-caster-vs %{
    //includes
}%

CCProgram standard-fs %{
    //includes
}%

CCProgram shadow-caster-fs %{
    //includes
}%

CCProgram reflect-map-fs %{
    //includes
}%
```

这类以 xxx-vs, xxx-fs 命名的 CCProgram 代码片段，就是我们的组装器。

在这些代码片段中，我们使用 #include 关键字，根据需要，引入不同模块头的文件，按顺序组装每个Shader。

include 两种方式

在这些 include 中，你可以看到如下两种情况：

```
//include from CCProgram
#include <macro-remapping>

//include from file
#include <surfaces/effect-macros/common-macros>
```

我们可以引入一个外部 chunk 文件，也可以通过名字，引入一个先前定义好的 CCProgram，比如 macro-remapping。

主要部分

以 `standard-fs` 为例，我们可以看到整个 Fragment Shader 的组装分为以下 6 个部分。

1、宏

首先需要包含必要的内部宏映射和通用宏定义。

宏映射使用在 Macro Remapping 一段中描述的自定义 CCProgram 代码块或 chunk 文件。

接下来需要包含通用宏定义文件 `common-macros`，如下所示：

```
#include <macro-remapping>
#include <surfaces/effect-macros/common-macros>
```

对于一些特殊渲染用途的 Shader 而言，建议直接包含对应渲染用途的宏定义文件，以 ShadowMap 为例：

```
CCProgram shadow-caster-fs %{
    ...
    #include <surfaces/effect-macros/render-to-shadowmap>
    ...
}%
```

2、通用头文件

根据 **当前的 Shader 类型（Shader Stage）** 来选择对应的通用头文件，如下所示：

```
//Vertex Shader
CCProgram standard-vs %{
    ...
    #include <surfaces/includes/common-vs>
    ...
}%

//Fragment Shader
CCProgram standard-fs %{
    ...
    #include <surfaces/includes/common-fs>
    ...
}%
```

3、Surface Shader 主体

这个部分是 Surface Shader 中的主体部分。

比如外部常量 uniforms，shared-ubos 代码块。

以及 Surface Shader 中，用户可以控制的主体函数，比如内置着色器里的 surface-vertex 和 surface-fragment 代码段。

如下所示：

```
CCProgram standard-fs %{
    ...
    #include <shared-ubos>
    #include <surface-fragment>
    ...
}%
```

4、光照模型

此部分为**可选项**，Vertex Shader ，以及渲染到 ShadowMap 时，不需要。

这个部分的作用，是使用 **光照模型名称** 来选择对应的头文件，如下所示：

```
//Standard PBR Lighting
#include <lighting-models/includes/standard>

//Toon Lighting
#include <lighting-models/includes/toon>
```

5、表面材质数据结构

此部分为**可选项**，渲染到 ShadowMap 时，不需要。

选择与光照模型对应的表面材质数据结构，如下所示：

```
//Vertex Shader
//Standard
#include <surfaces/includes/standard-fs>
//Toon
#include <surfaces/includes/toon-fs>
```

```
//Fragment Shader
//Standard
#include <surfaces/includes/standard-fs>
//Toon
#include <surfaces/includes/toon-fs>
```

6、主函数

使用**渲染用途名称 + Shader 类型（Shader Stage）** 来选择对应的主函数头文件，如下图所示：

```
//standard-vs:
#include <shading-entries/main-functions/render-to-scene/vs>

//shadow-caster-vs:
#include <shading-entries/main-functions/render-to-shadowmap/vs>

//standard-fs:
#include <shading-entries/main-functions/render-to-scene/fs>

//shadow-caster-fs:
#include <shading-entries/main-functions/render-to-shadowmap/fs>
```

更多信息，可参考 内置 Surface Shader 导读。

VS 输入

Vertex Shader 的输入值

参数列表

Vertex Shader 输入值都在 `SurfacesStandardVertexIntermediate` 结构体中，作为函数参数传入。

Vertex Shader 输入值	类型	使用时需要开启的宏	含义
position	vec4	N/A	本地坐标
normal	vec3	N/A	本地法线
tangent	vec4	CC_SURFACES_USE_TANGENT_SPACE	本地切线和镜像法线标记
color	vec4	CC_SURFACES_USE_VERTEX_COLOR	顶点颜色
texCoord	vec2	N/A	UV0
texCoord1	vec2	CC_SURFACES_USE_SECOND_UV	UV1
clipPos	vec4	N/A	投影坐标
worldPos	vec3	N/A	世界坐标
worldNormal	vec4	N/A	世界法线和双面材质标记
worldTangent	vec3	CC_SURFACES_USE_TANGENT_SPACE	世界切线
worldBinormal	vec3	CC_SURFACES_USE_TANGENT_SPACE	世界副法线

宏开关

当需要使用带有宏开关的输入参数时，需要在 macro-remapping 代码段中开启相应的宏，示例代码如下：

```
CCProgram macro-remapping %{
    ...
    //使用第二套 UV
    #define CC_SURFACES_USE_SECOND_UV 1
    //使用世界副法线
    #define CC_SURFACES_USE_TANGENT_SPACE 1
    ...
}
```

使用示例

在任何带 SurfacesStandardVertexIntermediate 参数的函数中，都可以直接访问到相关参数，以 SurfacesVertexModifyWorldPos 函数为例：

```
#define CC_SURFACES_VERTEX_MODIFY_WORLD_POS
vec3 SurfacesVertexModifyWorldPos(in SurfacesStandardVertexIntermediate In)
{
    vec3 worldPos = In.worldPos;
    worldPos.x += sin(cc_time.x * worldPos.z);
    worldPos.y += cos(cc_time.x * worldPos.z);
    return worldPos;
}
```

FS 输入

Fragment Shader 的输入值

内置输入列表

Vertex Shader 向 Fragment Shader 传递了许多常用变量，列表如下：

Fragment Shader 输入值	类型	使用时需要开启的宏	含义
FSInput_worldPos	vec3	N/A	World Position 世界坐标
FSInput_worldNormal	vec3	N/A	World Normal 世界法线
FSInput_faceSideSign	float	N/A	Two Side Sign 物理正反面标记，可用于双面材质
FSInput_texcoord	vec2	N/A	UV0
FSInput_texcoord1	vec2	N/A	UV1
FSInput_vertexColor	vec4	N/A	Vertex Color 顶点颜色
FSInput_worldTangent	vec3	N/A	World Tangent 世界切线
FSInput_mirrorNormal	float	N/A	Mirror Normal Sign 镜像法线标记
FSInput_localPos	vec4	CC_SURFACES_TRANSFER_LOCAL_POS	Local Position 局部坐标
FSInput_clipPos	vec4	CC_SURFACES_TRANSFER_CLIP_POS	Clip Position 投影/裁切空间坐标

宏开关

当需要使用带有宏开关的输入参数时，需要在 macro-remapping 代码段中开启相应的宏，示例代码如下：

```
CCProgram macro-remapping %{
    ...
    //使用 FSInput_localPos
    #define CC_SURFACES_TRANSFER_LOCAL_POS 1
    //使用 FSInput_clipPos
    #define CC_SURFACES_TRANSFER_CLIP_POS 1
    ...
}
```

使用说明

在任意函数中直接调用即可。

自定义传递值

在制作一些特殊效果时，需要 Vertex Shader 向 Fragment Shader 传递更多信息，此时需要我们自定义传递变量。

新增一个自定义的传递变量非常简单，我们以新增一个 testVec3 为例。

首先，在 Vertex Shader 中声明一个带 out 标记的变量，示例代码如下：

```
CCProgram surface-vertex %{
    ...
    out vec3 testVec3;
    ...
}
```

再在 Fragment Shader 声明对应的带 in 标记的变量即可，示例代码如下：

```
CCProgram surface-fragment %{
    ...
    in vec3 testVec3;
    ...
}
```

接下来就可以在 Fragment Shader 的代码中使用 testVec3 了。

自定义 Surface Shader

自定义表面着色器

虽然 Surface Shader 提供了大多数场景材质都能适配的光照模型，但其功能还是较为固定的。

有时候，**用户需要使用完全定制化的光照计算和色彩计算**，比如说：一些特殊的、风格化的材质，需要轮廓光、额外的补光、非真实的环境照明等等。

针对这类极为特殊的情况，Surface Shader 也提供了自定义能力。

但需要注意的是，由于干预了表面材质数据和光照计算过程，渲染效果可能会出现意料之外的效果。

1、自定义 VS 输出与 FS 输入

我们可以在 VS 阶段新定义一个传递变量之后，在某个 Surface 函数中计算并输出该变量值。

在 FS 阶段定义一个同名变量之后在某个 Surface 函数中获取并使用该变量值。

详情请参考 Fragment Shader 的输入参数：**自定义传递值**。

2、自定义材质信息

在 VS 函数块中添加如下代码：

```
//PBR 光照模型
#include <surfaces/data-structures/standard>
// toon 光照模型
//#include <surfaces/data-structures/toon>
#define CC_SURFACES_FRAGMENT_MODIFY_SHARED_DATA
void SurfacesFragmentModifySharedData(inout SurfacesMaterialData surfaceData)
{
    // set user-defined data to surfaceData
}
```

函数开头的 `#include` 用于决定使用的**材质数据结构名称**，根据不同的 include 文件, 会采用不同的 `SurfacesMaterialData` 结构体。

具体内容，可以查看 **internal/chunks/surfaces/data-structures/** 目录下的 **standard.chunk** 和 **toon.chunk**

当定义了 `CC_SURFACES_FRAGMENT_MODIFY_SHARED_DATA` 宏后，Shader编译器会选择你自己写的 `SurfacesFragmentModifySharedData` 来替换默认的函数。

此函数会在 vs 阶段被调用，具体可以查看 **internal/chunks/shading-entries/main-functions/** 目录下的： **render-to-scene/vs.chunk** 和 **render-to-shadowmap/vs.chunk** 文件。

在这个函数中，我们可以直接修改 surfaceData 里的属性，为光照阶段做准备。

自定义光照计算结果

有了上面自定义的 SurfacesMaterialData，我们还需要配合光照阶段，才能实现我们想要的计算效果。

在 FS 中，添加下面的代码：

```
#include <lighting-models/includes/common>
#define CC_SURFACES_LIGHTING_MODIFY_FINAL_RESULT
void SurfacesLightingModifyFinalResult(inout LightingResult result, in LightingIntermediateData lightingData, in SurfacesMaterialData surfaceData, in LightingMiscData miscData)
{
    // use surfaceData and lightingData for customizing lighting result
}
```

这个函数会在 fs.chunk 中被调用。

可以看到函数有四个参数：

- LightingIntermediateData：计算光照时需要的信息，如法线、视线方向、视距等等
- SurfacesMaterialData：颜色、世界空间法线、PBR参数等信息
- LightingMiscData：光源类型、位置、方向、颜色、强度等
- LightingResult：用于返回光照结果，如 diffuse, specular, shadow,ao 等等。

在这个函数中，可以利用光照和材质参数，计算出光照结果，并放入 result 中。

对于局部光源（点光、聚光灯等）而言，此函数会逐光源执行。也就是说，如果物体受 6 个光源影响，这个函数会被调用 6 次。

如果希望 在重载函数内可以直接调用现成的内置光照模块函数，可以将 lighting-models/includes/common 改为对应光照模型使用的头文件。

比如，如果想要在函数中使用 PBR 光照模型内置的光照函数，可以包含 lighting-models/includes/standard 头文件。

在这个头文件中，会包含 lighting-models/model-functions/standard 头文件。

PBR光照相关的内置函数都在这里，直接调用即可。

更多自定义

如果上面的自定义机制还不能满足需求，建议参考 chunks/shading-entries 构建自己的 main 函数，以控制整个着色流程和计算细节。

渲染调式功能

渲染调试功能

注意：只有使用 Surface Shader 框架的材质，内置的渲染调试功能才可以生效。

通过在编辑器的场景预览窗口右上角按钮选择对应的调试模式即可同屏查看模型、材质、光照及其他计算数据，在渲染效果异常的时候可以快速定位问题。

传统着色器 - Legacy Shader

传统着色器 Legacy Shader

相对于传统着色器而言，表面着色器让流程更统一，暴露给用户的细节更少，因此从 3.7.2 开始，Surface Shader 作为默认的 builtin-standard 出现。

但传统着色器（Legacy Shader) 与 表面着色器（Surface Shader）各有优缺：

类型	优点	缺点
Legacy Shader	在面对特殊需求时更加灵活	暴露过多细节给用户，引擎升级时不易维护
Surface Shader	统一的着色流程，无需关注细节；引擎升级时用户层代码更易维护	需要理解整个实现机制才能掌握；能够自定义的功能有限

另外，引擎内置的 builtin-unlit 依然使用了部分 legacy shader 库。

掌握 Legacy Shader 也有助于理解更多实现细节。

- 内置 Legacy Shader 导读
- Legacy Shader 主要函数与结构体

内置 Legacy Shader 导读

内置 Legacy Shader 导读

Legacy Shader 相关的源码，有两个目录：

- internal/chunks/legacy/
- internal/effects/legacy/

在 `chunks/legacy/` 目录中，存放的是一些公共函数，如解码器、雾效、输入、输出、阴影、骨骼蒙皮等等。

Legacy Shader 和 Surface Shader 都会调用 internal/chunks/builtin/ 和 internal/chunks/common/ 提供的函数。

在 `effects/legacy/` 目录中，提供了三个内置的 Legacy Shader：

- standard： 标准材质
- terrain：用于地形渲染
- toon：用于卡通渲染

基本结构

Legacy Shader 代码通常由几个部分组成：

- 信息描述（`CCEffect`）：描述此 Shader 的技术、渲染过程组成部分，以及每个渲染过程使用的 Shader、渲染状态、属性等。
- 共享常量（`Shared UBOs`）：把 vs 和 fs 都需要用到的 uniforms 定义在一起，方便管理。
- 主体函数（`Shader Body`）：用于实现具体的 Shader 主体。

Legacy Shader 中的 CCEffect 和 共享常量部分与 Surface Shader 一致，可前往 内置 Surface Shader 导读 了解详情。

着色函数

为了更好地理解渲染流程，请先查看 前向渲染与延迟渲染 Shader 执行流程。

standard(PBR)

在 legacy/standard.effect 中，定义了着色相关的Shader代码：

```
CCProgram standard-vs %{
    //...
    void main(){
        StandardVertInput In;
        CCVertInput(In);
        //...
        gl_Position = cc_matProj * (cc_matView * matWorld) * In.position;
    }
}%

CCProgram standard-fs %{
    //...
    void surf(out StandardSurface s){
        //s.albedo = ...
        //s.occlusion = ...
        //s.roughness = ...
        //s.metallic = ...
        //s.specularIntensity = ...
        //s.normal = ...
    }
    CC_STANDARD_SURFACE_ENTRY()
}%
```

可以看到，在 vs 中，直接使用了 main 函数作为入口，而在 fs 中，只有一个 surf 函数。

这是因为 `CC_STANDARD_SURFACE_ENTRY` 宏展开后，就是 main 函数，这个 main 函数会调用 surf 函数。

terrain

terrain 使用 StandardSurface 作为材质表面数据结构，使用 `CC_STANDARD_SURFACE_ENTRY` 作为入口。这就说明，terrain 的渲染流程与光照计算和 standard 完全一致。

只是由于地形采用的是多层纹理混合，所以 terrain 使用的纹理以及 surf 函数实现细节与 standard 的有较大区别。

toon

在 legacy/toon.effect 中，我们可以看到：

```
CCProgram toon-vs %{
    //...
    void main(){
        StandardVertInput In;
        CCVertInput(In);
        //...
```

```
        gl_Position = cc_matProj * (cc_matView * matWorld) * In.position;
    }
}%

CCProgram toon-fs %{
    //...
    void surf(out ToonSurface s){
        //s.baseStep = ...
        //s.baseFeather = ...
        //s.shadeStep = ...
        //s.shadeFeather = ...
        //s.shadowCover = ...
    }

    void frag(){
        ToonSurface s; surf(s);
        vec4 color = CCToonShading(s);
        return CCFragOutput(color);
    }
}%
```

toon 最大的特征是在 CCEffect 中，多定义了一个 outline pass，outline pass 的代码在 chunks/legacy/main-functions/outline-vs(fs) 中。

toon 材质表面数据结构为 ToonSurface，与 standard 使用的不一样。 在 frag 函数中可以看到， toon 的光照计算使用了专门的 CCToonShading 函数。

并且，toon 自己定义了一个 frag 入口函数，未使用 `CC_STANDARD_SURFACE_ENTRY` 宏。这也意味着，toon 是不支持延迟渲染的。

shadow-caster

可以看到，standard，terrain，toon 都有关于 shadow 的代码片段：

```
CCProgram shadow-caster-vs %{
    //...
}%

CCProgram shadow-caster-fs %{
    //...
}%
```

这个套 vs/fs 用于阴影贴图生成，引擎渲染管线会在阴影贴图生成阶段，查找 phase 为 shadow-add 的 pass 进行绘制。

Legacy Shader 主要函数与结构体

Legacy Shader 主要函数与结构体

CCVertInput[1]

- 为对接骨骼动画与数据解压流程，我们提供了 `CCVertInput` 工具函数，它有 `general` 和 `standard` 两个版本，内容如下：

  ```
  // 位于 'input.chunk' 的通用顶点着色器输入
  #define CCVertInput(position) \
    CCDecode(position);         \
    #if CC_USE_MORPH            \
      applyMorph(position);     \
    #endif                      \
    #if CC_USE_SKINNING         \
      CCSkin(position);         \
    #endif                      \
    #pragma // 空 'pragma' 技巧，在编译时消除尾随分号

  // 位于 'input-standard.chunk' 的标准顶点着色器输入
  #define CCVertInput(In) \
    CCDecode(In);         \
    #if CC_USE_MORPH      \
      applyMorph(In);     \
    #endif                \
    #if CC_USE_SKINNING   \
      CCSkin(In);         \
    #endif                \
    #pragma // 空 'pragma' 技巧，在编译时消除尾随分号
  ```

- 如果只需要获取 **顶点位置信息**，可以使用 **general** 版本，那么顶点着色器函数开头的代码示例如下：

  ```
  #include <legacy/input>
  vec4 vert () {
    vec3 position;
    CCVertInput(position);
    // ... 对位置信息做自定义操作
  }
  ```

  如果还需要法线等信息，可使用 `standard` 版本，像下面这样写：

  ```
  #include <legacy/input-standard>
  vec4 vert () {
    StandardVertInput In;
    CCVertInput(In);
    // ... 此时 'In.position' 初始化完毕，并且可以在顶点着色器中使用
  }
  ```

上面的示例代码中，`StandardVertInput` 对象 `In` 会返回模型空间的顶点位置（`position`）、法线（`normal`）和切空间（`tangent`）信息，并对骨骼动画模型做完蒙皮计算。

`StandardVertInput` 结构体的定义如下：

```
struct StandardVertInput {
  highp vec4 position;
  vec3 normal;
  vec4 tangent;
};
```

> **注意**：引用头文件后，不要在 Shader 内重复声明这些 attributes（`a_position`、`a_normal`、`a_tangent` 等）。对于其他顶点数据（如 uv 等）还是需要声明 attributes 后再使用。

如果要对接引擎动态 Mesh 合批和几何体实例化（GPU Instancing），需要包含 `cc-local-batch` 头文件，通过 `CCGetWorldMatrix` 工具函数获取世界矩阵，示例如下：

```
mat4 matWorld;
CCGetWorldMatrix(matWorld);

mat4 matWorld, matWorldIT;
CCGetWorldMatrixFull(matWorld, matWorldIT);
```

更多细节，请参考 Cocos Shader 内置全局 Uniform。

CCFragOutput

Cocos Shader 提供了 `CCFragOutput` 工具函数用以简化片元着色器的输出，可用于直接返回片元着色器所需要的值，代码示例如下：

```
#include <legacy/output>
vec4 frag () {
  vec4 o = vec4(0.0);
  // ... 编写片元着实代码
  return CCFragOutput(o);
```

```
}
```

`CCFragOutput` 会根据管线状态来决定是否需要做 `ToneMap` 转码处理，这样中间的颜色计算就不必区分当前渲染管线是否为 HDR 流程。

代码示例如下：

```
vec4 CCFragOutput (vec4 color) {
  #if CC_USE_HDR
    color.rgb = ACESToneMap(color.rgb);
  #endif
  color.rgb = LinearToSRGB(color.rgb);
  return color;
}
```

特别注意：

如果采用 `CCFragOutput` 作为片元输出，中间的颜色运算必须转到 `Linear` 空间，因为 `CCFragOutput` 认为传入的参数是在 `Linear` 空间的，总是会进行 `LinearToSRGB` 转码。

`CCFragOutput` 函数一般不需要自己实现，它只起到与渲染管线对接的作用，且对于这种含有光照计算的输出，因为计算结果已经在 HDR 范围，所以应该包含 `output-standard` 而非 `output` 头文件。

如需包含标准的 PBR 光照计算，可使用 `StandardSurface` 结构体与函数 `CCStandardShadingBase` 一起构成 PBR 着色流程。

`StandardSurface` 结构体内容如下：

```
struct StandardSurface {
  // albedo
  vec4 albedo;
  // these two need to be in the same coordinate system
  vec3 position;
  vec3 normal;
  // emissive
  vec3 emissive;
  // light map
  vec3 lightmap;
  float lightmap_test;
  // PBR params
  float roughness;
  float metallic;
  float occlusion;
};
```

代码示例如下：

```
#include <legacy/shading-standard-base>
#include <legacy/output-standard>
void surf (out StandardSurface s) {
  // fill in your data here
}
vec4 frag () {
  StandardSurface s; surf(s);
  vec4 color = CCStandardShadingBase(s);
  return CCFragOutput(color);
}
```

也可以参考 `builtin-standard.effect` 中，使用 `surf` 函数与 `CC_STANDARD_SURFACE_ENTRY()` 宏组合。

`CC_STANDARD_SURFACE_ENTRY()` 是一个 wrapper，会根据渲染状态，利用 `surf` 函数构建出一个可用于片元的 `main` 函数，代码示例如下：

```
CCProgram shader-fs %{
  #include <legacy/standard-surface-entry>

  void surf (out StandardSurface s) {
    // fill in your data here
  }

  CC_STANDARD_SURFACE_ENTRY()
}%
```

StandardSurface

StandardSurface 为 PBR 材质信息结构体，记录了一个像素需要进行光照计算的表面信息。 Lagecy Shader 中的 surf 函数主要用于填充，它会在光照阶段被使用。

```
struct StandardSurface {
    // albedo
    vec4 albedo;
    // these two need to be in the same coordinate system
    HIGHP_VALUE_STRUCT_DEFINE(vec3, position);
    vec3 normal;
    // emissive
    vec3 emissive;
    // light map
    vec3 lightmap;
    float lightmap_test;
    // PBR params
    float roughness;
    float metallic;
    float occlusion;
    float specularIntensity;

    #if CC_RECEIVE_SHADOW
      vec2 shadowBias;
    #endif
};
```

ToonSurface

```
struct ToonSurface {
    vec4 baseColor;
    vec4 specular;
    // these two need to be in the same coordinate system
    HIGHP_VALUE_STRUCT_DEFINE(vec3, position);
    vec3 normal;
    // shading params
    vec3 shade1;
    vec3 shade2;
    vec3 emissive;
    float baseStep;
    float baseFeather;
    float shadeStep;
    float shadeFeather;
    float shadowCover;

    #if CC_RECEIVE_SHADOW
      vec2 shadowBias;
    #endif
};
```

和 StandardSurface 一样，ToonSurface 也是用于记录像素表面信息，只是它是卡通材质专用的结构体。

[1]. 不包含粒子、Sprite、后期效果等不基于 Mesh 渲染的 Shader。↩

自定义着色器

自定义着色器

若内置的着色器无法满足需求，可通过自定义着色器实现特例化渲染。

自定义着色器有以下两种方式：

1. 参考 着色器资源 创建新的着色器。
2. 基于内置着色器。将 **资源管理器** 面板中 **internal/effects/** 目录下相应的内置着色器拷贝到 **assets** 目录下，然后对其进行自定义。

准备工作

由于着色器是使用 YAML 作为流程控制，GLSL 作为着色器语言，因此在自定义着色器之前需要对这些知识有一定程度上的熟悉和了解：

对于不熟悉的开发者，我们也准备了一些简单的介绍：

- GLSL 语法简介
- YAML 语法简介

本节将以自定义 2D 着色器和 3D 着色器为例，详细介绍自定义流程，具体内容请参考：

- 3D 着色器：RimLight
- 2D 着色器：Gradient

2D 精灵着色器：Gradient

2D 精灵着色器：Gradient

默认的情况下，UI 和 2D 组件会使用引擎内置的着色器，这些着色器放在 **资源管理器** 面板的 **internal -> effects** 目录下，可查看 内置着色器 来了解其作用。

对于任何持有 CustomMaterial 属性的 UI 和 2D 组件，都可在 **属性检查器** 内通过该属性的下拉框选择或者从 **资源管理器** 内拖拽实现自定义材质。

引擎规定 UI 组件的自定义材质只能有一个。

3D 着色器：RimLight

3D 着色器：RimLight

本文将通过实现一个 RimLight 效果，来演示如何编写一个在 Cocos 可用于 3D 模型渲染的 Cocos Shader。

RimLight 也称为“内发光”/“轮廓光”/“边缘光”（本文统一使用边缘光），是一种通过使物体的边缘发出高亮，让物体更加生动的技术。

RimLight 是菲涅尔现象[1]的一种应用，通过计算物体法线和视角方向的夹角的大小，调整发光的位置和颜色，是一种简单且高效的提升渲染效果的着色器。

在边缘光的计算中，视线和法线的夹角越大，则边缘光越明显。

皮肤材质

皮肤材质

在游戏中皮肤如果使用普通的 BRDF 材质往往表现不够红润和通透，因为现实中皮肤结构比较复杂，我们需要引用 Separable Subsurface Scattering 可分离次表面散射，还原皮肤的 SSS 效果，并使表现层次更丰富。

使用皮肤材质

1. 使用皮肤材质需要开启自定义管线。详细操作步骤请参考自定义管线的功能开启。
2. 使用皮肤材质需要一些额外的宏定义配置。在 **项目** 中选中 **项目设置**，然后点击 **引擎管理器** 的 **宏配置** 侧边选项。
    - 勾选 **ENABLE_FLOAT_OUTPUT** 选项。如果不勾选，引擎每次启动也会自动检测是否有皮肤材质并更改此选项以保障正确的效果，但会产生警告信息。

自定义几何体实例化属性

自定义几何体实例化属性

通过 **几何体实例化** 特性（GPU Instancing）可使 GPU 批量绘制模型相同且材质相同的渲染对象。如果我们想在不打破这一特性的情况下单独修改某个对象的显示效果，就需要通过自定义几何体实例化属性。

我们以新增一个颜色属性为例。

定义变量

实例化属性需要单独定义，并且处于 `USE_INSTANCING` 宏定义之下，否则会出现编译错误。

```
#if USE_INSTANCING // when instancing is enabled
  #pragma format(RGBA8) // normalized unsigned byte
  in vec4 a_instanced_color;
#endif
```

用 vs 传递

虽然 a_instanced_color 仅用在 fs 中修改物体颜色，但几何体实例化属性属于顶点属性，只能在 vs 中被访问。 因此需要在 vs 中声明一个输出到 fs 的变量。 代码示例如下：

```
CCProgram vs %{
  #if USE_INSTANCING
    out vec4 instancedColor;
  #endif

  vec4 vert(){
      ...
      #if USE_INSTANCING
        instancedColor = a_instanced_color;
      #endif
      ...
  }
}%
```

在 fs 中使用

通过在 fs 声明对应的 in 变量，获取到 vs 中传递过来的几何体实例化属性，实现想要的功能。

```
CCProgram fs %{
  #if USE_INSTANCING
    in vec4 instancedColor;
  #endif

  vec4 frag(){
      ...
      vec4 o = mainColor;
      #if USE_INSTANCING
        o *= instancedColor;
      #endif
      ...
  }
}%
```

在脚本中设置属性

几何体实例化属性属于具体的渲染对象实例，无法通过材质属性面板设置，只能通过模型组件 MeshRenderer 上的 setInstancedAttribute 方法进行设置。 示例代码如下：

```
const comp = node.getComponent(MeshRenderer);
comp.setInstancedAttribute('a_instanced_color', [100, 150, 200, 255]);
```

注意事项

有以下几点需要注意：

1. `#pragma format(RGBA8)` 用于指定此属性的具体数据格式，参数可以为引擎 `GFXFormat` 中的任意枚举名[1]；如未声明则默认为 RGBA32F 类型。
2. 所有实例化属性都是从利用顶点着色器（vs）的 attribute 输入，如果要在片元着色器（fs）中使用，需要先在 vs 中声明，再传递给 fs。
3. 请确保代码在所有分支都能正常执行，无论 `USE_INSTANCING` 是否启用。
4. 实例化属性的值在运行时会初始化为 0。
5. 如果在 **MeshRenderer** 组件上更换了材质，那么所有的实例化属性值都会被重置，需要重新设置。

[1]. WebGL 1.0 平台下不支持整型 attributes，如项目需要发布到此平台，应使用默认浮点类型。↩

UBO 内存布局策略

UBO 内存布局策略

Cocos Shader 规定，所有非 sampler 类型的 uniform 都应以 UBO（Uniform Buffer Object/Uniform Block）形式声明。

以内置着色器 `builtin-standard.effect` 为例，其 uniform block 声明如下：

```
uniform Constants {
    vec4 tilingOffset;
    vec4 albedo;
    vec4 albedoScaleAndCutoff;
    vec4 pbrParams;
    vec4 miscParams;
    vec4 emissive;
    vec4 emissiveScaleParam;
  };
```

并且所有的 UBO 应当遵守以下规则：

1. 不应出现 vec3 成员；
2. 对数组类型成员，每个元素 size 不能小于 vec4；
3. 不允许任何会引入 padding 的成员声明顺序。

Cocos Shader 在编译时会对上述规则进行检查，以便在导入错误（implicit padding 相关）时及时提醒修改。

这可能听起来有些过分严格，但背后有非常务实的考量：
首先，UBO 是渲染管线内要做到高效数据复用的唯一基本单位，离散声明已不是一个选项；
其次，WebGL2 的 UBO 只支持 std140 布局，它遵守一套比较原始的 padding 规则[1]：

- 所有 vec3 成员都会补齐至 vec4：

```
uniform ControversialType {
  vec3 v3_1; // offset 0, length 16 [IMPLICIT PADDING!]
}; // total of 16 bytes
```

- 任意长度小于 vec4 类型的数组和结构体，都会将元素补齐至 vec4：

```
uniform ProblematicArrays {
  float f4_1[4]; // offset 0, stride 16, length 64 [IMPLICIT PADDING!]
}; // total of 64 bytes
```

- 所有成员在 UBO 内的实际偏移都会按自身所占字节数对齐[2]：

```
uniform IncorrectUBOOrder {
  float f1_1; // offset 0, length 4 (aligned to 4 bytes)
  vec2 v2; // offset 8, length 8 (aligned to 8 bytes) [IMPLICIT PADDING!]
  float f1_2; // offset 16, length 4 (aligned to 4 bytes)
}; // total of 32 bytes

uniform CorrectUBOOrder {
  float f1_1; // offset 0, length 4 (aligned to 4 bytes)
  float f1_2; // offset 4, length 4 (aligned to 4 bytes)
  vec2 v2; // offset 8, length 8 (aligned to 8 bytes)
}; // total of 16 bytes
```

这意味着大量的空间浪费，且某些设备的驱动实现也并不完全符合此标准[3]，因此目前 Cocos Shader 选择限制这部分功能的使用，以帮助排除一部分非常隐晦的运行时问题。

再次提醒：uniform 的类型与 inspector 的显示和运行时参数赋值时的程序接口可以不直接对应，通过 property target 机制，可以独立编辑任意 uniform 的具体分量。

[1]. OpenGL 4.5, Section 7.6.2.2, page 137 ↩

[2]. 注意在示例代码中，UBO IncorrectUBOOrder 的总长度为 32 字节，实际上这个数据到今天也依然是平台相关的，看起来是由于 GLSL 标准的疏忽，更多相关讨论可以参考 这里。↩

[3]. **Interface Block - OpenGL Wiki**：https://www.khronos.org/opengl/wiki/Interface_Block_(GLSL)#Memory_layout ↩

WebGL 1.0 向下兼容支持

WebGL 1.0 向下兼容支持

由于 WebGL 1.0 仅支持 GLSL 100 标准语法，因此在 Cocos Shader 编译时会提供 GLSL 300 ES 转 GLSL 100 的向下兼容代码（fallback shader），开发者基本不需关心这层变化。

需要注意的是目前的自动向下兼容策略仅支持一些基本的格式转换，如果使用了 GLSL 300 ES 独有的函数（例如 texelFetch、textureGrad）或一些特有的扩展（extensions），推荐根据 __VERSION__ 宏定义判断 GLSL 版本，自行实现更稳定精确的向下兼容，代码示例如下：

```
#if __VERSION__ < 300
#ifdef GL_EXT_shader_texture_lod
  vec4 color = textureCubeLodEXT(envmap, R, roughness);
#else
  vec4 color = textureCube(envmap, R);
#endif
#else
  vec4 color = textureLod(envmap, R, roughness);
#endif
```

Cocos Shader 在编译时会解析所有已经是常量的宏控制流，生成不同版本的 GLSL Shader 代码。

VSCode 着色器插件

VSCode 着色器插件（实验性）

在 VSCode 应用商店中搜索扩展 Cocos Effect 并安装，即可启用 .effect 文件的简易语法着色。

如果打开的 .effect 是 3.7.3 及之后版本引擎中的内置文件，或使用此版本编辑器打开过的工程中的文件，会自动新增如下功能：

1、自动补全和语法着色

引擎内置函数、宏和全局变量的自动补全、语法着色等功能。

输入关键字会有列表提示，按上下键选择，同时会显示提示信息，按回车可以自动补全

鼠标移动到补全后的关键字上会有悬停提示信息，包括注释和所在文件位置等

> **注意**：由于是实验性功能，此功能目前仅支持部分引擎内置常用代码的感应，不支持用户代码自动感应。

2、语法检查及错误跳转

当前窗口打开一个 .effect 文件，然后在查看菜单--命令面板（Ctrl+Shift+P）....选择**Cocos Effect: compile effect**即可调用工具自动编译当前文件，并给出对应的错误提示，**在提示窗口红色错误的文件路径上按住 Ctrl 点击鼠标左键即可跳转到对应代码行**。

在此项右侧有一个小的齿轮图标，点击还可以添加快捷键绑定，此后只要在 .effect 文件中按快捷键即可调用语法检查器，完整操作流程见下图。

> **注意**：由于是实验性功能，此功能目前只支持 Windows 系统。

计算着色器

计算着色器

计算着色器（Compute Shader，以下简称 CS）是 GPU 上执行通用计算任务的编程模型。Cocos CS 继承 GLSL 语法与内置变量，添加方式与光栅化 Shader 相同以 effect 形式呈现，**仅支持在自定义管线**中使用。Compute Shader 无光栅化过程，输入输出均为内存数据。通过多线程完成并行处理，处理大量数据时非常高效。

Effect 定义方式

定义方式同光栅化 Shader，在计算着色器下配置 PipelineState 无意义。

```
CCEffect %{
  techniques:
  - name: opaque
    passes:
    - compute: compute-main           // 定义 cs 入口
      pass: user-compute              // 定义 layout
      properties: &props
        mainTexture: { value: grey }  // 注册着色器材质面板资源属性
}%
CCProgram compute-main %{
  precision highp float;
  precision mediump image2D;
  layout(local_size_x = 8, local_size_y = 4, local_size_z = 1) in;

  #pragma rate mainTexture batch
  uniform sampler2D mainTexture;

  #pragma rate outputImage pass
  layout (rgba8) writeonly uniform image2D outputImage;

  void main () {
    imageStore(outputImage, ivec2(gl_GlobalInvocationID.xy), vec4(1, 0, 0, 1));
  }
}%
```

如要查看更多语法，请移步 着色器语法

输入输出

CS 输入输出包含内置输入变量与着色器资源。

内置输入包含以下部分，与 GLSL 语义相同：

```
in uvec3 gl_NumWorkGroups;
in uvec3 gl_WorkGroupID;
in uvec3 gl_LocalInvocationID;
in uvec3 gl_GlobalInvocationID;
in uint  gl_LocalInvocationIndex;
layout(local_size_x = X, local_size_y = Y, local_size_z = Z) in;
```

着色器资源包含：

- UniformBuffer
- StorageBuffer
- ImageSampler
- StorageImage
- SubpassInput

CS 无内置输出，可通过 StorageBuffer / Image 输出。

着色器资源

CS 目前支持 **PerPass**，**PerBatch** 两种频率的资源绑定，如下所示：

```
#pragma rate mainTexture batch
uniform sampler2D mainTexture;
```

```
#pragma rate outputImage pass
layout (rgba8) writeonly uniform image2D outputImage;
```

其中 PerPass 资源可以定义为需要管线跟踪处理同步问题的资源，PerBatch 通常为常量数据、静态纹理，可通过 Material 进行修改绑定。

PerBatch mainTexture 在 properties 中绑定后，可在材质面板中配置。

PerPass outputImage 需要在管线中声明并由 ComputePass 引用，并由 RenderGraph 管理数据的读写同步以及 ImageLayout，详见后文示例。

管线集成

Cocos 自定义管线添加 CS 过程分为三步：

1. 添加 Compute Pass，其中 `passName` 为当前 Pass 的 LayoutName，需要与 Effect 中的 pass 字段对应。

   ```
   const csBuilder = pipeline.addComputePass('passName');
   ```

2. 声明、引用资源，并关联资源访问类型与 Shader 资源槽位。

   ```
   const csOutput = 'cs_output';
   if (!pipeline.containsResource(csOutput)) {
      pipeline.addStorageTexture(csOutput,
          gfx.Format.RGBA8,
          width, height,
          rendering.ResourceResidency.MANAGED);
   } else {
      pipeline.updateStorageTexture(csOutput,
          width, height,
          gfx.Format.RGBA8);
   }

   csBuilder.addStorageImage(csOutput,  // 资源名
      rendering.AccessType.WRITE,      // 内存访问类型
      'outputImage');                  // 着色器资源名
   ```

3. 添加 Dispatch 调用实例，设置 Dispatch 参数，绑定材质

   ```
   csBuild.addQueue().addDispatch(x, y, z, rtMat);
   ```

平台支持

特性支持

	WebGL	WebGL2	Vulkan	Metal	GLES3	GLES2
支持情况	N	N	Y	Y	Y(3.1)	N

可通过 `device.hasFeature(gfx.Feature.COMPUTE_SHADER)` 查询。

约束限制

- `maxComputeSharedMemorySize`: 本地共享内存的最大字节数。
- `maxComputeWorkGroupInvocations`: 一个 WorkGroup 内的最大调用次数，即 Local WorkGroup 体积
- `maxComputeWorkGroupSize`: Local WorkGroup 三维数组限制
- `maxComputeWorkGroupCount`: Dispatch 三维数组限制

可通过 `device.capabilities` 查询。

平台差异

Cocos Creator 会将 Cocos CS 转化为平台对应版本的 glsl shader，因此为了能够保证各个平台的兼容性，需要尽量满足所有平台的限制要求，包括：

1. Vulkan 与 GLES 要求显式标明 Storage Image 的 Format 标识符，详细参考 GLSE 语法标准。
2. GLES 要求显式标明 Stroage 资源的 Memory 标识符，目前仅支持 readonly 与 writeonly。此外需要显式标识默认精度。

优化建议

1. 进行屏幕空间图像后处理时，可优先考虑 Fragment Shader。避免 RenderTarget 结果中途切换 CS 进行写操作。
2. 避免使用较大的工作组，尤其在使用 shared memory 情况下，每个 WorkGroup 大小建议不超过 64。

示例

下文展示了一个通过 ComputePass 实现的单个球 1 rpp 的简单光线追踪演示代码，使用到了 UniformBuffer，ImageSampler 与 StorageImage，其中 Effect Pass 声明部分如下：

```
techniques:
- name: opaque
  passes:
  - compute: compute-main
    pass: user-ray-tracing
    properties: &props
      mainTexture: { value: grey }
```

`compute-main` 实现部分如下：

```
precision highp float;
precision mediump image2D;

layout(local_size_x = 8, local_size_y = 4, local_size_z = 1) in;

#pragma rate tex batch
uniform sampler2D tex;

#pragma rate constants pass
uniform constants {
  mat4 projectInverse;
};

#pragma rate outputImage pass
layout (rgba8) writeonly uniform image2D outputImage;

void main () {
  vec3 spherePos = vec3(0, 0, -5);
  vec3 lightPos = vec3(1, 1, -3);
  vec3 camPos = vec3(0, 0, 0);
  float sphereRadius = 1.0;

  vec4 color = vec4(0, 0, 0, 0);

  ivec2 screen = imageSize(outputImage);
  ivec2 coords = ivec2(gl_GlobalInvocationID.x, gl_GlobalInvocationID.y);
  vec2 uv = vec2(float(coords.x) / float(screen.x), float(coords.y) / float(screen.y));

  vec4 ndc = vec4(uv * 2.0 - vec2(1.0), 1.0, 1.0);
  vec4 pos = projectInverse * ndc;
  vec3 camD = vec3(pos.xyz / pos.w);
  vec3 rayL = normalize(camD - camPos);

  vec3 dirS = spherePos - camPos;
  vec3 rayS = normalize(dirS);
```

```
  float lenS = length(dirS);

  float dotLS = dot(rayL, rayS);
  float angle = acos(dotLS);
  float projDist = lenS * sin(angle);

  if (projDist < sphereRadius) {
    // intersection
    vec3 rayI = rayL * (lenS * dotLS - sqrt(sphereRadius * sphereRadius - projDist * projDist));

    vec3 N = normalize(rayI - dirS);
    vec3 L = normalize(lightPos - rayI);
    color = vec4(vec3(max(dot(N, L), 0.05)), 1.0);
  }

  imageStore(outputImage, coords, color);
}
```

管线 API 调用如下：

```
export function buildRayTracingComputePass(
    camera: renderer.scene.Camera,
    pipeline: rendering.Pipeline) {
    // 获取屏幕长宽
    const area = getRenderArea(camera,
        camera.window.width,
        camera.window.height);
    const width = area.width;
    const height = area.height;

    // 声明 RT Storage Image 资源
 const csOutput = 'rt_output';
    if (!pipeline.containsResource(csOutput)) {
        pipeline.addStorageTexture(csOutput,
            gfx.Format.RGBA8,
            width, height,
            rendering.ResourceResidency.MANAGED);
    } else {
        pipeline.updateStorageTexture(csOutput,
            width, height,
            gfx.Format.RGBA8);
    }

    // 声明 Compute Pass, layout 需要与 Effect 中 pass 字段保持一致
 const cs = pipeline.addComputePass('user-ray-tracing');
    // 更新相机投影参数
    cs.setMat4('projectInverse', camera.matProjInv);
    // 声明当前 Compute Pass 对 Storage Image 引用
    cs.addStorageImage(csOutput, rendering.AccessType.WRITE, 'outputImage');
    // 添加 Dispatch 参数，绑定 Material
    cs.addQueue()
        .addDispatch(width / 8, height / 4, 1, rtMat);
    // 返回当前 Image 资源名，用于后续 Post Processing 处理
    return csOutput;
}
```

Compute Pass 需要用户完成对 PerPass 资源的更新与绑定，PerBatch 的资源会由材质系统完成绑定，最终经过 FullScreen Blit 到屏幕的效果:

渲染排序

渲染排序组件

对于大部分渲染情况来说，默认的排序已经满足需求。但是实际上由于半透明物体的特殊性，我们可能需要手动对这些物体进行排序。对于这种情况，可以使用 **渲染排序组件**。

在 **属性检查器** 内点击 **添加组件** 按钮选择 **Sorting** 即可添加。

特效组件

特效组件

特效组件包括 **广告牌** 和 **线段组件**，可在 **属性检查器** 中点击 **添加组件 -> Effects** 进行添加。

具体的说明及使用请参考：

- 广告牌
- 线段组件

广告牌

Billboard 组件

Billboard 组件用于渲染一个始终面向摄像机的方块。

线段组件

Line 组件

Line 组件用于渲染 3D 场景中给定的点连成的线段。Line 组件渲染的线段是有宽度的，并且总是面向摄像机，这与 billboard 组件相似。

属性与说明

天空盒

天空盒

游戏中的天空盒是一个包裹整个场景的立方体，可以很好地渲染并展示整个场景环境，在基于 PBR 的工作流中天空盒也可以贡献非常重要的 IBL 环境光照。

开启天空盒

在 **层级管理器** 中选中场景根节点，然后在 **属性检查器** 的 **Skybox** 组件中勾选 **Enabled** 属性即可开启天空盒。

全局雾

全局雾

全局雾用于在游戏中模拟室外环境中的雾效果。在游戏中除了用于雾效表现外，还可以用于隐藏摄像机远剪切平面外的模型来提高渲染的性能。

全局雾的类型目前包括 **线性雾**、**指数雾**、**指数平方雾**、**层雾** 四种，详情请参考下文 **全局雾类型** 部分的内容。

开启全局雾

在 **层级管理器** 中选中场景根节点，然后在 **属性检查器** 的 **Fog** 组件中勾选 **Enabled** 属性即可开启全局雾。

几何渲染器

几何渲染器（Geometry-Renderer）

几何渲染器是引擎提供的一种批量渲染各种几何体的功能接口，主要用于调试（比如显示物体的包围盒）及 Cocos Creator 的 gizmo 批量显示。

几何渲染器的效果展示如图：

调试渲染器（Debug-Renderer）

调试渲染器（Debug-Renderer）

调试渲染器是引擎提供的一种批量渲染屏幕文字的功能接口，主要用于调试，输出任意的文字调试信息到屏幕上。目前仅支持原生平台。

其效果图如下所示：

2D 对象

2D 对象概述

区别于 3D 模型对象，我们将不涉及模型的图片渲染体统称为 2D 渲染对象。2D 渲染对象的处理在底层的数据提交上与 3D 模型存在差异，其遵循自己的规则做出了一些针对性的调整以实现更好的效率表现和使用体验。

2D 对象渲染结构说明

2D 渲染对象的收集采用树状结构，RenderRoot 节点（带有 RenderRoot2D 组件的节点）为 2D 对象数据收集的入口节点，所有的 2D渲染对象需在 RenderRoot 节点下才可以被渲染。由于 Canvas 组件本身继承 RenderRoot2D 组件，所以 Canvas 组件也可以作为数据收集的入口。2D 渲染节点必须带有 UITransform 组件作为渲染顶点数据、点击或者对齐策略等功能生效的必要条件。

2D 渲染也可以支持对模型进行渲染，唯一的条件是带有网格渲染器组件（例如 `MeshRenderer/SkinnedMeshRenderer`）的节点必须添加 **UI/UIMeshRenderer** 组件才可以和 UI 在相同的管线上进行渲染。

2D 渲染流程如下：

2D 渲染

2D 渲染

引擎中所有不拥有的 model 的渲染对象都为 2D 渲染对象。与 3D 对象不同，2D 对象本身不拥有 model 信息，其顶点信息是由 UITransform 组件的 Rect 信息持有并由引擎创建的，且本身没有厚度。由于引擎的设计要求，2D 渲染对象需要为 RenderRoot 节点（带有 RenderRoot2D 组件的节点）的子节点才能完成数据的收集操作。

所以 2D 渲染对象的渲染要求有两点：

1. 自身带有 UITransform 组件
2. 需要为带有 RenderRoot2D/Canvas 组件节点的子节点

2D 渲染对象可见性说明

由于 2D 渲染对象在 Camera 的可见性判断上和 3D 渲染节点并无区别，所以用户需要自己控制节点的 layer 属性并设置 Camera 的 Visibility 来配合进行分组渲染，如果场景中出现多个相机的情况，错误的 layer 设置导致节点重复渲染或不渲染。

这里请 3D 1.2 版本升级的用户注意，我们纠正了之前的 Canvas 只会渲染其子节点的行为，目前需要用户自己管理节点的 layer 和相机的 Visibility，之前使用了多 Canvas 渲染的用户可能会需要对项目做出调整以达到更合理的场景结构。

2D 渲染组件

本身拥有渲染能力的组件我们称为 2D 渲染组件，包括：

- Sprite 组件参考
- Label 组件参考
- Mask 组件参考
- Graphics 组件参考
- RichText 组件参考
- UIStaticBatch 组件参考
- TiledMap 组件参考
- TiledTile 组件参考
- Spine（骨骼动画）Skeleton 组件参考
- DragonBones（龙骨）ArmatureDisplay 组件参考
- MotionStreak 组件参考

组件添加方式

我们在编辑器内置了一些 2D 渲染组件，在创建了 RenderRoot 节点之后，即可在此节点下创建带有 2D 渲染组件的节点：

渲染排序规则

渲染排序说明

2D 渲染节点排序

2D 渲染节点可分为在 Canvas 下的节点和不在 Canvas 下的节点两种：

- 在 Canvas 下的节点可参考下文 **UI 节点排序** 部分的内容。
- 不在 Canvas 下的节点，用户可选择通过 自定义材质 来开启深度检测实现和 3D 物体的遮挡显示，开启后会按照物体的 Z 轴坐标进行遮挡渲染（可参考范例 **2d-rendering-in-3d**（GitHub | Gitee）。

UI 节点排序

UI 节点特指在 Canvas 节点下的 UI 节点，节点的混合是严格按照节点树进行排序的。UI 的渲染排序采用的是一个深度优先的排序方式，节点树的排序方式即为最终渲染数据提交的顺序。所以用户可以通过设置节点的 siblingIndex 来改变节点在父节点下的顺序，从而改变渲染顺序。

举个例子：

2D 渲染组件合批说明

2D 渲染组件合批规则说明

合批条件说明

2D 渲染组件合批需要满足以下几点条件：

条件	说明
节点的 Layer 相同	由于 Layer 与节点是否渲染相关，因此不同的 Layer 之间不能进行合批。
使用的材质相同	**材质相同是合批的必要要求。** 由于 Creator 使用的是 **材质实例化** 的机制，所以当用户在设置了材质的 `uniform` 之后，材质会进行实例化，实例化之后的材质是无法进行合批的。 如果用户在自定义材质中设置了 `uniform`（对应组件无法进行合批），在 `uniform` 值使用完毕后想要该组件参与合批，那么可通过 `CustomMaterial` 接口将材质资源重新赋值给组件即可。
渲染组件上所添加的材质的 `BlendState` 和 `DepthStencilState` 属性设置相同	`DepthStencilState` 属性用于控制组件的深度检测和模板缓冲，是由引擎自动控制（用于 Mask 的效果实现）的，一般来说用户不需关心该属性的设置。
渲染组件的顶点信息要在同一个 **buffer** 中上传（v3.4.1 新增）	一般情况下顶点信息是由引擎统一进行分配和管理的，用户无需关心。若想要了解更多相关信息，请参考下文 **MeshBuffer 合批说明** 部分的内容。
贴图源以及贴图采样相同	一般情况下该条件是影响合批最主要的因素，尤其对于 Sprite 和 Label 来说，贴图很容易产生差别导致无法合批。Creator 提供了部分方法用于更好地实现合批，详情请参考下文 **合批方法说明** 部分的内容。

合批方法说明

结合以上的合批条件说明，我们可以通过一些方法来实现更好的合批方法。需要额外说明的是，2D 渲染组件的渲染数据采集使用的是基于 **节点树** 的渲染方式，而有部分组件无法合批，且会打断其他组件合批，需要用户进行分模块管理节点树布局，以达到更好的合批效果。无法合批的组件包括：

- 内置组件 Mask、Graphics 和 UIMeshRenderer 组件由于材质不同和数据组织方式的差异，无法与其他组件合批；
- TiledMap、Spine 和 DragonBones 这三个中间件组件则是遵循自己的内部合批机制。

对于 Sprite 和 Label 组件来说，因为贴图很容易产生差别，导致无法合批。因此我们提供了以下方法可以更好地实现合批，用户可以根据需要参考使用：

- 对于 Sprite 组件，我们提供了 静态合图 和 动态合图 两种合批方案，通过将图片纹理合并，即可在其他条件满足的情况下进行合批。
- 对于 Label 组件，我们提供了 Bitmap 的缓存方法，通过将 Label 的纹理合图，即可实现 Sprite 和 Label 组件的合批，但需要注意的是，使用 Bitmap 缓存方式的 Label 不可频繁变动文字内容。

一般来说，通过控制材质和节点树状态，然后配合合图方法，便能够达到较好的合批效果。

MeshBuffer 合批说明

由于合批要求渲染对象的顶点在同一个 MeshBuffer 中，而以下几种情况则会造成 MeshBuffer 切换。

v3.4.1 之前

场景中要绘制的顶点数量超过了 MeshBuffer 所能容纳的最大顶点数量（65535 个）。

v3.4.1 之后

由于我们在 v3.4.1 中采用了新的渲染数据提交设计，所以需要注意以下几点：

1. **项目设置 -> Macro Configurations** 面板中的 **BATCHER2D_MEM_INCREMENT** 的值会影响每个 MeshBuffer 可容纳的最大顶点数量，当这个值越大，同一个 MeshBuffer 可容纳的 2D 渲染对象也就越多。但与此同时，内存增加的幅度也会越大，请用户结合自身项目规模酌情设置大小。
2. **BATCHER2D_MEM_INCREMENT** 的单位为 **KB**，与可容纳的顶点数量之间的转换关系如下：

   引擎内置标准的顶点格式为 vfmtPosUvColor，在引擎中的定义为：

   ```
   export const vfmtPosUvColor = [
   // RGB32F 表示 3 个 32 位的 float
   new Attribute(AttributeName.ATTR_POSITION, Format.RGB32F),
   // RG32F 表示 2 个 32 位的 float
   new Attribute(AttributeName.ATTR_TEX_COORD, Format.RG32F),
   // RGBA32F 表示 4 个 32 位的 float
   new Attribute(AttributeName.ATTR_COLOR, Format.RGBA32F),
   ];
   ```

   由此我们可以得到每个顶点占用 9 个 `float` 值，每个顶点占用的空间为：`1 * 9 * 4 / 1024 (KB)`，其中：

   - **1** 表示顶点数；
   - **9** 表示 `vfmtPosUvColor` 条件下每个顶点占用的 `float` 值；
   - **4** 表示每个 `float` 占用的字节数；
   - **1024** 表示字节与 KB 转换单位。

   因此 **BATCHER2D_MEM_INCREMENT** 的默认值 144KB，表示可容纳 `144 * 1024 / (9 * 4) = 4096` 个标准格式的顶点。需要注意的是：同一 MeshBuffer 容纳的最大顶点数不可超过 **65535** 个，即 **BATCHER2D_MEM_INCREMENT** 的最大值不可大于 `65535 * 9 * 4 / 1024 ≈ 2303.96 (KB)`。
3. 目前 2D 渲染使用的核心为 **static VB**，其主要内容包括：
   - **VB 固定** 伴随着组件的整个生命周期而存在，而决定渲染顺序的 IB 则放弃缓存，每帧进行录制。由于组件的 VB 信息多而复杂，直接保存下来可针对组件的内存段进行操作，而且保存下来可以避免无用的更新。
   - IB 每帧填充。相对于 VB 来说，IB 的结构简单、数量较少，并且会直接决定渲染顺序，所以在每帧遍历时重新填充 IB 消耗较小，且无需管理复杂的缓存机制。
   - 由于 VB 固定，所以在整个组件的生命周期中，VB 会在最开始就分配好，直到组件销毁，所以在组件加载时，便会向预先分配好的 MeshBuffer 中申请好想要使用的 VB，然后在销毁时归还。
   - 当 MeshBuffer 已经无法分配出组件需要的 VB 时，系统便会新创建一个大小为 **BATCHER2D_MEM_INCREMENT** 的 MeshBuffer 来继续分配 VB。

2D 渲染对象自定义材质

2D 渲染对象自定义材质

2D 渲染对象的自定义材质是拓展 2D 渲染对象表现和提升 2D 渲染对象自身能力的最佳实践，可以通过自定义材质实现溶解、外发光等酷炫的渲染效果。

v3.0 的 2D 渲染组件大部分都支持使用自定义材质，其使用界面如下图（以 Sprite 组件为例）：

2D 渲染组件

2D 渲染组件介绍

Sprite 组件参考

Sprite 组件参考

Sprite（精灵）是 2D/3D 游戏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。

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：

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，该属性只有选择了 `RADIAL` 填充类型才能修改。决定了扇形填充时会环绕 Sprite 上的哪个点。

Label 组件参考

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、BMFont 字体或艺术数字。另外，Label 还具有排版功能。

Label 排版

属性	功能说明
CLAMP	文字尺寸不会根据 Content Size 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 Content Size 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 Content Size 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 Content Size 的大小。 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 Content Size 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。

文本缓存类型（Cache Mode）

类型	功能说明
NONE	默认值，Label 中的整段文本将生成一张位图。
BITMAP	选择后，Label 中的整段文本仍将生成一张位图，但是会尽量参与 动态合图。只要满足动态合图的要求，就会和动态合图中的其它 Sprite 或者 Label 合并 Draw Call。由于动态合图会占用更多内存，**该模式只能用于文本不常更新的 Label**。此模式在节点安排合理的情况下可大幅降低 Draw Call，请酌情选择使用
CHAR	原理类似 BMFont，Label 将以“字”为单位将文本缓存到全局共享的位图中，相同字体样式和字号的每个字符将在全局共享一份缓存。能支持文本的频繁修改，对性能和内存最友好。不过目前该模式还存在如下限制，我们将在后续的版本中进行优化： 1. **该模式只能用于字体样式和字号（通过记录字体的 fontSize、fontFamily、color、outline 为关键信息，以此进行字符的重复使用，其他有使用特殊自定义文本格式的需要注意）固定，并且不会频繁出现巨量未使用过的字符的 Label**。这是为了节约缓存，因为全局共享的位图尺寸为 **1024 * 1024**，只有场景切换时才会清除，一旦位图被占满后新出现的字符将无法渲染。 2. Overflow 不支持 SHRINK。 3. 不能参与动态合图（同样启用 CHAR 模式的多个 Label 在渲染顺序不被打断的情况下仍然能合并 Draw Call） 4. 目前暂不支持 **IsBold**、**IsItalic** 和 **IsUnderline** 属性。

注意：Cache Mode 对所有平台都有优化效果。

Mask 组件参考

Mask（遮罩）组件参考

Mask 用于规定子节点可渲染的范围，默认带有 Mask 组件的节点会使用该节点的约束框（也就是 **属性检查器** 中 Node 组件的 **ContentSize** 规定的范围）创建一个矩形渲染遮罩，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。

Graphics 组件参考

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。

RichText 组件参考

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。

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>` param 当点击事件触发时，可以在回调函数的第二个参数获取该数值 `<on click="handler" param="test"> click me! </on>` 依赖 click 事件 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。

img 标签的可选属性

为了更好地排版，我们为 img 标签类型提供了可选属性，你可以使用 `width` 及 `height` 来指定 SpriteFrame 的大小，这将允许该图片可以大于或是小于行高（但此设定不会改变行高）。

当你改变了 SpriteFrame 的高度或宽度后，或许会需要使用 `align` 来调整该图片在行中的对齐方式。

属性	描述	示例	注意事项
height	指定 SpriteFrame 的渲染高度，大小值必须为整数	`<img src='foo' height=50 />`	如果你只使用了高度属性，该 SpriteFrame 会自动计算宽度以保持图片比例
width	指定 SpriteFrame 的渲染宽度，大小值必须为整数	`<img src='foo' width=50 />`	你可以同时使用高度及宽度属性 `<img src='foo' width=20 height=30 />`
align	指定 SpriteFrame 在行中的对齐方式，值必须为 `bottom`、`top` 或 `center`	`<img src='foo' align=center />`	预设对齐方式为 bottom

为了支持定制图片排版，我们还提供了 `offset` 属性，用于微调 SpriteFrame 在 RichText 中的位置。设置 `offset` 时需注意属性值必须为整数，并且如设置不当将导致图片与文字重叠。

offset 属性	示例	描述	注意事项
Y	`<img src='foo' offset=5 />`	指定 SpriteFrame 的 y 轴加上 5	当 offset 只设定一个值的时候，它代表 y 轴的偏移
Y	`<img src='foo' offset=-5 />`	指定 SpriteFrame 的 y 轴减少 5	你可以设定负整数来减少 y 轴
X, Y	`<img src='foo' offset=6,-5 />`	指定 SpriteFrame 的 x 轴加上 6，y 轴减少 5	偏移属性的值只能包含 `0-9`、`-` 和 `,` 字符

标签嵌套

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>
```

文本缓存类型（Cache Mode）

由于富文本组件是由多个 Label 节点拼装而成，所以对于复杂的富文本，drawcall 数量也会比较高，因此引擎为富文本组件提供了 Label 组件的文本缓存类型设置，来适当减少 drawcall 的增加。对于每种缓存类型的具体说明，参照 Label 组件的文本缓存类型

类型	功能说明
NONE	默认值，对富文本所拆分创建的每个 Label 节点，设置其 CacheMode 为 NONE 类型，即将每个 Label 的整段文本生成一张位图并单独进行渲染。
BITMAP	选择后，对富文本所拆分创建的每个 Label 节点，设置其 CacheMode 为 BITMAP 类型，即将每个 Label 的整段文本生成一张位图，并将该位图添加到动态图集中，再依据动态图集进行合并渲染。
CHAR	选择后，对富文本所拆分创建的每个 Label 节点，设置其 CacheMode 为 CHAR 类型，即将每个 Label 的文本以“字”为单位缓存到全局共享的位图中，相同字体样式和字号的每个字符将在全局共享一份缓存。

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。

示例

事件的用法

我们以 `on` 标签作为示例，来说明下事件的用法。请在项目中新建任意一个带有 RichText 组件的节点。并创建一个自定义脚本，挂载在 RichText 组件上。

UIStaticBatch 组件参考

UIStaticBatch 组件参考

注意：我们在 v3.4.1 中采用了新的渲染合批策略，普通动态合批组件在性能上会得到有效提升，因此不再建议使用 UIStaticBatch 组件进行手动管理。

UI 静态合批组件是一个提升 UI 渲染性能的组件，脚本在初始化当前帧渲染的过程中会收集该 UI 节点树下的所有渲染数据（除了模型、Mask 和 Graphics），存储为一个静态的 IA 渲染数据。并在后续的渲染流程中使用固定数据进行渲染，不再遍历其节点树，此后的坐标变换将不再生效。当你需要修改静态数据的时候，可以调用 markAsDirty 接口来重新触发渲染数据收集标记。

遮罩的组件接口请参考 UIStaticBatch API。

关于使用可以参考范例 **UIStaticBatch**（GitHub | Gitee）。

通过脚本代码开启静态合批

```
import { _decorator, Component } from 'cc';
const { ccclass, property } = _decorator;

@ccclass("example")
export class example extends Component {
    start(){
        const uiStatic = this.node.getComponent(UIStaticBatch);
        // 选择你要开始静态合批的时机，调用此接口开始静态合批
        uiStatic.markAsDirty();
    }
}
```

注意事项

使用该组件有以下几点需要注意：

1. 不要频繁触发静态合批，因为会清空原先存储的 IA 数据重新采集，会有一定性能和内存损耗。
2. 不适用于子节点树中包含 Mask、Graphics 和 Model 的情况。
3. 对于节点树不会有任何改变的节点（例如 2D 地图），在 **开始静态合批之后** 即可将所有子节点删除，以得到最好的性能和内存表现。

Spine Skeleton 组件参考

Spine Skeleton 组件参考

Spine Skeleton 组件支持 Spine 官方工具导出的数据格式，并对 Spine（骨骼动画）资源进行渲染和播放。

DragonBones ArmatureDisplay 组件参考

DragonBones ArmatureDisplay 组件参考

ArmatureDisplay 组件可以对 DragonBones（龙骨）资源进行渲染和播放。

TiledMap 组件参考

TiledMap 组件参考

TiledMap（地图）用于在游戏中显示 TMX 格式的地图。

TiledTile 组件参考

TiledTile 组件参考

TiledTile 组件可以单独对某一个地图块进行操作。

MotionStreak

MotionStreak（拖尾）组件参考

MotionStreak（拖尾）是运动轨迹，用于在游戏对象的运动轨迹上实现拖尾渐隐效果。

UI 系统

UI 系统

本章将介绍 Cocos Creator 中强大而灵活的 UI（用户界面）系统，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，以及支持多种排版布局方式的 UI 界面。

UI 入门

在引擎中界定 UI 和 2D 渲染对象的区别主要在于适配和交互，所有的 UI 需要在 Canvas 节点下，以做出适配行为，而 Canvas 组件本身继承自 `RenderRoot2D` 组件，所以也可以作为数据收集的入口。

UI 是游戏开发的必要交互部分，一般游戏上的按钮、文字、背景等都是通过 UI 来制作的。在开始制作一款 UI 时，首先需要确定当前设计的内容显示区域大小（设计分辨率），可以在菜单栏的 **项目 -> 项目设置 -> 项目数据** 面板中设置：

UI 组件

UI 组件

一些常用的 UI 控件可通过添加节点的方式来创建。在 **层级管理器** 中点击左上角的 + 创建节点按钮，然后选择 **UI** 来创建所需的 UI 节点，相应的 UI 组件便会自动挂载到节点上：

Canvas 组件参考

Canvas（画布）组件参考

UITransform 组件参考

UI 变换组件

定义了 UI 上的矩形信息，包括矩形的尺寸和锚点位置。开发者可以通过该组件任意地操作矩形的大小、位置。一般用于渲染、点击事件的计算、界面布局以及屏幕适配等。

点击 **属性检查器** 下面的 **添加组件** 按钮，然后选择 **UI/UITransform** 即可添加 UITransform 组件到节点上。

UITransform 脚本接口请参考 UITransform API。

UITransform 属性介绍

属性	功能说明
ContentSize	UI 矩形内容尺寸
AnchorPoint	UI 矩形锚点位置

通过脚本代码修改节点尺寸和锚点

```
import { _decorator, Component, Node, UITransform } from 'cc';
const { ccclass, property } = _decorator;

@ccclass('Example')
export class Example extends Component {

    start () {
        const uiTransform = this.getComponent(UITransform);
        // 方法一
        uiTransform.setContentSize(200, 120);
        uiTransform.setAnchorPoint(0, 0.5);

        // 方法二
        uiTransform.width = 200;
        uiTransform.height = 120;
        uiTransform.anchorX = 0;
        uiTransform.anchorY = 0.5;
    }
}
```

priority 属性的弃用说明

我们在 3.1 版本中弃用了 UITransform 组件的 priority 属性，用户可以通过使用 setSiblingIndex 来设置节点树的顺序来进行调整渲染顺序。

关于 priority 属性的移除及推荐使用的 SiblingIndex 属性的说明： 由于表意不明以及与引擎中其他属性的命名冲突，UITransform 组件上的 priority 属性在 v3.1 已被废弃。priority 属性的设计之初是为用户提供节点树排序的快捷方式，本身并无其他用途，和 priority 所表达的“优先级”也不相关，并且设置之后实际上还是通过更改节点树的顺序来调整渲染顺序。

在移除了 `priority` 属性之后，用户可以用 `setSiblingIndex` 方法来替换使用，`setSiblingIndex` 方法通过影响节点中的 `siblingIndex` 属性来调整节点树的顺序。不同之处在于，`priority` 属性存在默认值，而 node 的 `siblingIndex` 属性实际上就是这个节点在父节点中的位置，所以在节点树发生变化之后，节点的 `siblingIndex` 属性数值就会产生变化。这就要求在使用 `setSiblingIndex` 方法的时候，需要知道节点在父节点中的相对位置并做出控制，才能够获得预期的结果。

需要注意的是，不能直接将 `siblingIndex` 属性等同于 `priority`（已废弃）属性来理解使用，它们的意义是不同的，改变 `siblingIndex` 属性需要理解并清楚其代表的是在父节点下的位置，且在节点树变化时会发生变化，并且只能通过 `setSiblingIndex` 方法来修改 `siblingIndex` 属性。

考虑到节点快捷排序的需求，我们会在之后的版本中提供更方便快捷的接口供用户排列节点使用。

Widget 组件参考

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。对齐方案详细说明请参考 对齐方案

Button 组件参考

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。

Layout 组件参考

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。

EditBox 组件参考

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。

ScrollView 组件参考

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。

ScrollBar 组件参考

ScrollBar 组件参考

ScrollBar 允许用户通过拖动滑块来滚动一张图片，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。

ProgressBar 组件参考

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。

LabelOutline 组件参考

LabelOutline 组件参考

LabelOutline 组件会为所在节点的 Label 添加描边效果，不支持 BMFont 字体。

LabelShadow 组件参考

LabelShadow 组件参考

LabelShadow 组件可以为 Label 组件添加阴影效果。

Toggle 组件参考

Toggle 组件参考

Toggle 是一个 CheckBox，当它和 ToggleContainer 一起使用的时候，可以变成 RadioButton。

ToggleContainer 组件参考

ToggleContainer 组件参考

Slider 组件参考

Slider 组件参考

Slider 是一个滑动器组件。

PageView 组件参考

PageView 组件参考

PageView 是一种页面视图容器.

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

PageViewIndicator 设置

PageViewIndicator 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。

PageView 事件

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，都是通过代码添加。首先需要构造一个 `EventHandler` 对象，然后设置好对应的 `target`、`component`、`handler` 和 `customEventData` 参数。

```
import { _decorator, Component, Event, Node, PageView, EventHandler } from 'cc';
const { ccclass, property } = _decorator;

@ccclass("example")
export class example extends Component {
    onLoad(){
        const pageChangedEventHandler = new EventHandler();
        pageChangedEventHandler.target = this.node; // 这个 node 节点是你的事件处理代码组件所属的节点
        pageChangedEventHandler.component = 'example'; // 这个是脚本类名
        pageChangedEventHandler.handler = 'callback';
        pageChangedEventHandler.customEventData = 'foobar';

        const page = this.node.getComponent(PageView);
        page.clickEvents.push(pageChangedEventHandler);
    }

    callback(event: Event, customEventData: string){
        // 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
        const node = event.target as Node;
        const pageview = node.getComponent(PageView);
        console.log(customEventData); // 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：
import { _decorator, Component, Event, Node, PageView } from 'cc';
const { ccclass, property } = _decorator;

@ccclass("example")
export class example extends Component {
    onLoad(){
        this.pageView.node.on('page-turning', this.callback, this);
    }

    callback(pageView: PageView) {
        // 回调的参数是 pageView 组件
        // 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 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。

UIMeshRenderer 组件参考

UIMeshRenderer 组件参考

UIMeshRenderer 是一个将 3D 模型从 3D 渲染管线转换到 2D 渲染管线的带有转换功能的渲染组件。该组件支持 3D 模型和粒子在 UI 上的显示，没有这个组件，即使模型和粒子节点在 UI 里也不会被渲染。

> **注意：**若 3D 模型无法在 UI 场景中正常显示，请尝试放大模型倍数。

该组件的添加方式是在 **层级管理器** 中选中带有或继承自 MeshRenderer 组件的节点，然后点击 **属性检查器** 下方的 **添加组件** 按钮，选择 **UI-> UIMeshRenderer** 即可。而粒子则是添加到粒子节点上。通常结构如下所示：

UICoordinateTracker 组件参考

UICoordinateTracker 组件参考

UI 坐标跟踪映射组件是在 UI 上执行坐标转换以及模拟透视相机下 3D 物体近大远小效果。通过事件的方式将转换后的坐标以及物体在视口下的占比返回。适用于 3D 人物血条以及姓名条之类功能。

UICoordinateTracker 属性

属性	功能说明
Target	目标对象。需要转换到哪一个 UI 节点下。
Camera	照射相机。
UseScale	是否启用缩放。在透视相机下，根据 3D 节点坐标与相机的距离，调整映射后物体的缩放比例，实现近大远小效果。需要结合 distance 使用。
Distance	距相机多少距离为正常显示计算大小。根据模型在相机下的照射效果调整最佳位置，以该位置为分界线计算在视口占比。
SyncEvents	映射数据事件。回调的第一个参数是映射后的本地坐标，第二个是距相机距离比。

具体的使用方法可参考范例 **UI 展示 Demo**（GitHub | Gitee）中的 rocker 场景。

UIOpacity 组件参考

UIOpacity（透明度设置）组件参考

该组件会为节点记录一个透明度修改标识用来影响到后续的渲染节点。一般用于非渲染节点，如果作用在渲染节点上会形成透明度叠加现象。渲染节点可以通过设置 color 的 alpha 通道来设置透明度。

> **注意**：该组件仅对根节点上有 Canvas 或 RenderRoot2D 组件的节点生效。

使用方法如下：

BlockInputEvents 组件参考

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。

WebView 组件参考

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。由于不同平台对于 WebView 组件的授权、API、控制方式都不同，还没有形成统一的标准，所以目前只支持 Web、iOS 和 Android 平台。

VideoPlayer 组件参考

VideoPlayer 组件参考

VideoPlayer 是一种视频播放组件，可通过该组件播放本地和远程视频。

播放本地视频：

SafeArea 组件参考

SafeArea 组件参考

该组件会将所在节点的布局适配到 iPhone X 等异形屏手机的安全区域内，可适配 Android 和 iOS 设备，通常用于 UI 交互区域的顶层节点。

UI 实践指南

UI 实践指南

多分辨率适配方案

多分辨率适配方案

Cocos Creator 3.0 在整体设计上沿用了 Cocos Creator 2.x 一套资源适配多种分辨率屏幕的方案。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）** 组件添加给 UI 节点，能够根据需要将元素对齐目标节点（默认是父节点）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 `800 x 480` 和 `1280 x 720` 两种屏幕分辨率，或 iOS 设备中 `1136 x 640` 和 `960 x 640` 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，会如何进行适配呢？

假设我们的设计分辨率为 `800 x 480`，美术制作了一个同样分辨率大小的背景图像。

对齐策略

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. 在 **层级管理器** 中创建 2D 对象时会默认自动创建一个 Canvas 节点作为其父节点，这些元素节点都需要放在 Canvas 节点下
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 40px，下边距设为 30px。

文字排版

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 UITransform 组件所拥有的尺寸信息（`contentSize`），也就是约束框（Bounding Box）所规定的范围。

自动布局容器

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。

接下来说到的布局类型，节点结构都如下图：

制作动态生成内容的列表

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 脚本开发指南 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.ts`，并添加如下的属性：

```
@ccclass('Item')
export class Item {
    @property
    id = 0;
    @property
    itemName = '';
    @property
    itemPrice = 0;
    @property(SpriteFrame)
    iconSF: SpriteFrame | null = null;
}

@ccclass
export class ItemList extends Component {
    @property([Item])
    items: Item[] = [];
    @property(Prefab)
    itemPrefab: Prefab | null = null;

    onLoad() {
        for (let i = 0; i < this.items.length; ++i) {
            const item = instantiate(this.itemPrefab);
            const data = this.items[i];
            this.node.addChild(item);
            item.getComponent('ItemTemplate').init(data);
        }
    }
}
```

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 `Component`，因此它不是一个组件，但可以被组件使用。关于声明自定义类的更多内容，请查阅 装饰器 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.ts` 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。

制作可任意拉伸的 UI 图像

制作可任意拉伸的 UI 图像

UI 系统核心的设计原则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。

安卓大屏幕适配

安卓大屏幕适配

动画系统

动画系统

Cocos Creator 内置了通用的动画系统用以实现基于关键帧的动画。除了支持标准的位移、旋转、缩放动画和帧动画之外，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细腻的各种动态效果。

动画剪辑

动画剪辑

动画剪辑（Animation Clip）是一份动画的声明数据，即包含动画数据的资源，是动画系统的核心之一。将动画剪辑挂载到 动画组件 上，就能够将这份动画数据应用到动画组件所在的节点上。

目前 Creator 支持从外部导入动画，或者直接在 Creator 内部创建一个全新的动画剪辑。

Creator 内部创建的动画

通过 **动画编辑器** 可以直接创建全新的动画剪辑，并进行编辑和预览，详情请参考 使用动画编辑器。

也可以通过脚本创建，详情请参考 程序化编辑动画剪辑。

外部导入的骨骼动画

外部导入的动画大概包括以下几种：

1. 第三方美术工具生产的骨骼动画
2. 模型导入后附带的骨骼动画

   带动画的模型导入后，会同时导入模型中包含的动画。这个动画和内部新建资源的使用方式是一样的，骨骼动画的裁剪可以参考 模型资源的动画模块介绍。

Creator 从 v3.4 开始引入了全新的 Marionette 动画系统，实现了由状态机控制的骨骼动画流程，具体内容请参考 Marionette 动画系统。

更多关于骨骼动画的设置等，详情请参考 骨骼动画。

> **注意**：外部导入的骨骼动画不支持在 **动画编辑器** 中查看和编辑，各节点也是锁住状态，只能在外部美术工具中进行编辑。

动画组件参考

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。

使用动画编辑器

动画编辑器

Creator 支持在 **动画编辑器** 中直接创建、编辑和预览动画剪辑，除了可以对节点基本属性进行动画化，还支持对材质和部分组件的属性进行动画化，并且可以通过调用 动画事件 的事件函数来充实动画剪辑。

在使用 **动画编辑器** 制作动画之前需要先为节点添加 **动画组件**，并为动画组件挂载 **动画剪辑** 后才可以进行编辑。详情请参考 创建 Animation 组件和动画剪辑。

创建 Animation 组件和动画剪辑

创建动画组件和动画剪辑

在使用 **动画编辑器** 制作动画之前需要先在 **层级管理器** 或者 **场景编辑器** 中选择要添加动画的节点，然后添加 动画组件，并在组件上挂载 **动画剪辑（Animation Clip）**，便可以编辑动画数据，编辑后的动画数据会保存在当前的动画剪辑中。没有挂载 Clip 的节点是无法编辑动画数据的。

如果当前选中节点没有动画组件，则 **动画编辑器** 的界面上会显示 **添加 Animation 组件** 按钮，点击即可在 **属性检查器** 上添加 **动画组件**。

动画编辑器面板介绍

熟悉动画编辑器

动画编辑器 面板用于编辑和预览当前所选节点的动画剪辑。编辑动画数据或者相关属性时，鼠标焦点离开后会立即生效。

Cocos Creator 的默认布局中已经包含了 **动画编辑器**，也可以通过点击编辑器上方主菜单的 **面板 -> 动画 -> 动画编辑器** 打开 **动画编辑器**。

面板介绍

动画编辑器 面板可以划分为以下几个主要部分：

编辑动画剪辑

编辑动画剪辑

在节点的动画组件上挂载了动画剪辑后，点击 **进入动画编辑模式** 或者使用快捷键 `Ctrl`/`Cmd` + `E` 进入动画编辑模式，便可以在动画剪辑中添加关键帧数据，以此实现节点的动画化。在编辑动画剪辑之前请先 熟悉动画编辑器。

一个动画剪辑内可能包含了多个节点（节点及其子节点），每个节点上可挂载多个动画属性。通过对节点进行移动、旋转、缩放等操作，便会在当前选中节点相对应的动画属性上添加关键帧，动画属性上添加的所有关键帧在对应的动画属性中显示为线性轨迹的清单模式，我们可以称之为动画曲线。

创建动画曲线

在添加关键帧之前需要先了解一下动画属性，动画属性包括了节点自有的 `position`、`rotation`、`scale` 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`。

点击 **属性列表** 区域右上角的 + 按钮即可根据需要添加动画属性，根据节点类型的不同，可添加的动画属性也有所不同。已添加的动画属性则为置灰状态，不可重复添加。

关键帧编辑视图

关键帧编辑视图

节点轨道显示区域与属性轨道显示区域，默认均为关键帧编辑视图的显示方式。在这种视图下，可以更方便的编辑整体的关键帧排布、添加以及删除关键帧。下面介绍在关键帧编辑视图下，支持的各种

关键帧操作方法，了解这些方法技巧可以更快更方便地编辑动画剪辑。

选中关键帧

选中的关键帧会由蓝色变成白色，包括以下几种：

- 单击动画属性轨道上的关键帧即可选中
- 双击关键帧则会在选中关键帧的同时将时间控制线移动到当前关键帧所在位置

- 单击节点在动画时间轴中的关键帧，即可同时选中节点的各个动画属性在同一位置上的所有关键帧。

曲线编辑视图

曲线编辑视图

在创建了基本的动画剪辑之后，我们可以**通过编辑动画曲线来控制关键帧之间如何随着时间推移而变化**。动画曲线实际上是由当前属性轨道上各个关键帧连接而成的线性轨迹，每个关键帧是这条曲线路径上的控制点。曲线编辑视图下，可以更直观的看到关键帧值的变化，方便更加细腻的调节关键帧数值变化。

点击属性列表边上的切换视图按钮

曲线编辑器

曲线编辑器

曲线编辑器主要用于编辑关键帧之间变化的曲线轨迹，在编辑器中的 动画 和 粒子 都使用了曲线编辑器来编辑关键帧曲线。

每一条曲线上的关键点，**横坐标** 表示关键帧时间/帧数，**纵坐标** 表示当前曲线属性在对应时间上的值。曲线编辑器支持同时编辑多条曲线，但在粒子的使用场景内目前只支持单条曲线编辑。

缩放移动曲线显示区域

- 直接在曲线显示区域内滚动鼠标滚轮，即可同时放大或者缩小横纵时间轴的显示比例。
- 按住鼠标右键不放拖动也可以直接平移当前的显示区域范围。

除此之外还可以结合快捷键来单独控制横纵时间轴的平移缩放：

- 按住 `Shift` 不放后滚动鼠标滚轮，可以向左或向右 **平移** 曲线视图区域;
- 按住 `Ctrl/Cmd + Shift` 不放后滚动鼠标滚轮，可以向左或向右 **缩放** 曲线视图区域;
- 按住 `Alt/Option` 不放后滚动鼠标滚轮，可以向上或向下 **平移** 曲线视图区域;
- 按住 `Ctrl+Alt / Cmd+Option` 不放后滚动鼠标滚轮，可以向上或向下 **缩放** 曲线视图区域;

编辑曲线轨迹

添加关键帧

1. 当只显示一条曲线轨迹时，可以在任意空白处右键点击，然后在弹出的菜单中选择 **新建关键帧** 即可。

   以粒子编辑器中的曲线为例：

辅助曲线编辑视图

辅助曲线编辑视图

辅助曲线是 Cocos Creator v3.8 新增的动画辅助功能，您可以阅读 辅助曲线 来了解辅助曲线的基础概念。

如果要启用辅助曲线，请在 **偏好设置** 中的 **实验室** 分页中启用 **动画辅助曲线**。

添加动画事件

添加动画事件

通过在动画时间轴的指定帧调用 **动画事件** 函数可以更好地充实动画剪辑。在动画时间轴某一帧上添加 **事件帧** 后，动画系统将会在动画执行到该帧时，根据事件帧中设置的触发函数名称去匹配动画根节点中对应的函数方法并执行。

若要通过脚本添加动画事件，详情请参考 帧事件。

添加事件帧

在 **动画编辑器** 中添加事件帧包括以下两种方式：

1. 将时间控制线拖动到需要添加事件帧的位置，然后点击 **菜单工具栏** 中的

程序化编辑动画剪辑

程序化编辑动画剪辑

> **注意**：从 v3.3 开始，动画剪辑接口经历了较大的变动，详情可参考 动画剪辑数据升级指南。

Creator 除了支持在 **动画编辑器** 中 创建动画剪辑，还可以通过脚本模块程序化地创建动画剪辑，例如：

```
import { animation, AnimationClip, Vec3 } from "cc";

const animationClip = new AnimationClip();
animationClip.duration = 1.0; // 整个动画剪辑的周期

const track  = new animation.VectorTrack(); // 创建一个向量轨道
track.componentsCount = 3; // 使用向量轨道的前三条通道
track.path = new animation.TrackPath().toHierarchy('Foo').toProperty('position'); // 指定轨道路径，即指定目标对象为 "Foo" 子节点的 "position" 属性
const [x, y, z] = track.channels(); // x, y, z 是前三条通道
x.curve.assignSorted([ // 为 x 通道的曲线添加关键帧
    [0.4, ({ value: 0.4 })],
    [0.6, ({ value: 0.6 })],
    [0.8, ({ value: 0.8 })],
]);

// 如果关键帧的组织是 [时间, 向量] 数组，可以利用解构语法赋值每一条通道曲线。
const vec3KeyFrames = [
    [0.4, new Vec3(1.0, 2.0, 3.0)],
    [0.6, new Vec3(1.0, 2.0, 3.0)],
    [0.8, new Vec3(1.0, 2.0, 3.0)],
```

```
] as [number, Vec3][];
x.curve.assignSorted(vec3KeyFrames.map(([time, vec3]) => [time, { value: vec3.x }]));
y.curve.assignSorted(vec3KeyFrames.map(([time, vec3]) => [time, { value: vec3.y }]));
z.curve.assignSorted(vec3KeyFrames.map(([time, vec3]) => [time, { value: vec3.z }]));

// 最后将轨道添加到动画剪辑以应用
animationClip.addTrack(track);
```

具体的说明请查看下文介绍。

动画属性轨道

动画剪辑中的任一节点支持添加多条 **动画属性轨道**，动画属性轨道由类 `animation.Track` 表示，描述了某一对象上的某一动画属性随着时间推移而发生的变化，并规定了如何将其应用到目标对象上。

动画属性轨道根据下文中介绍的 轨道类型 的不同可包含一至多条通道，一般情况下一条动画属性轨道对应一条通道，除了复合轨道，例如 `position`，有 `X`、`Y`、`Z` 三条通道。每条通道都含有一条曲线，曲线是 **可编辑的最小单元**，若动画属性轨道上未添加关键帧，则曲线为空曲线。

根据曲线类型的不同，通道包括以下几种：

- **实数通道**，含有一条实数曲线 `RealCurve`
- **四元数通道**，含有一条四元数曲线 `QuatCurve`
- **对象通道**，含有一条动画曲线 `ObjectCurve`

动画剪辑运行时，每条属性轨道都将绑定到某个对象或某个对象的动画属性上，并通过赋值对象属性或下文介绍的 **值代理** 来产生动画效果。

轨道类型

动画属性轨道的类型决定了轨道包含多少条通道（曲线），以及每条通道（曲线）的类型和含义，Creator 提供了以下类型的轨道：

轨道类型	类	产生的值	说明
数值轨道	`animation.RealTrack`	`number`	数值轨道产生 JavaScript 数值，包含 **一条** 实数通道。
向量轨道（2/3/4 维）	`animation.VectorTrack`	`Vec2`/`Vec3`/`Vec4`	向量轨道包括 2/3/4 维向量，值分别由向量类 `Vec2`、`Vec3`、`Vec4` 表示。向量轨道的维度是通过 `componentsCount` 字段获取和设置的，需要注意的是向量轨道共包含 **4 条** 实数通道，但运行时仅会使用前 `componentsCount` 条通道。
四元数轨道	`animation.QuatTrack`	`Quat`	四元数轨道（对应节点上的 `rotation` 属性）产生四元数，值由 `Quat` 表示。四元数轨道仅包含 **一条** 四元数通道，这意味着四元数的各个分量属性不可单独编辑，但大多数情况下分量属性的单独编辑也是没有意义的。
颜色轨道	`animation.ColorTrack`	`Color`	颜色轨道产生颜色值，值由 `Color` 表示。颜色轨道包含 **4 条** 实数通道，分别对应于红色、绿色、蓝色、透明度，且范围在 [0-255] 内。各个通道采样后产生的颜色值，将按照类 `Color` 规定的方式将实数转换为整数颜色分量值。
尺寸轨道	`animation.SizeTrack`	`Size`	尺寸轨道产生尺寸值，值由 `Size` 表示。尺寸轨道包含 **两条** 实数通道，分别对应于尺寸的宽度和高度。
对象轨道	`animation.ObjectTrack`	任意值	对象轨道产生任意类型的值，仅包含 **一条** 对象曲线。对象轨道产生的值即是对象曲线产生的值。

轨道路径

每个动画属性轨道都记录了一个路径，称为 **轨道路径**，由类 `animation.TrackPath` 表示。轨道路径指定了在 **运行时** 如何从当前节点对象寻址到目标对象，因为寻址是在运行时完成的，这种特性使得动画剪辑可以复用到多个对象上。

轨道路径由多个子路径组成，每个子路径都指定了如何从上一级路径的寻址结果寻址到另一个对象，最后一个子路径寻址到的结果就是目标对象。类似文件路径用于定位文件夹或文件，而轨道路径用于定位目标对象。

通过下表中 `animation.TrackPath` 的方法可根据目标对象类型添加子路径，以及获取、判断子路径的类型：

目标对象类型	添加子路径方法	获取解析子路径类型	判别子路径类型
对象属性	`toProperty()`	`parsePropertyAt()`	`isPropertyAt()`
数组元素	`toElement()`	`parseElementAt()`	`isElementAt()`
节点的子节点	`toHierarchy()`	`parseHierarchyAt()`	`isHierarchy()`
节点上的组件	`toComponent()`	`parseComponentAt()`	`isComponentAt()`

以下代码片段演示了如何指定轨道路径：

```
function specifyTrackPath(track: animation.TrackPath) {
    const { path } = track;
    path                                    // 从当前节点对象寻址到目标对象
        .toHierarchy('path/to/children')    // 目标对象为当前节点的 “path/to/child” 子节点
        .toComponent('MyComponent')         // 目标对象为 “path/to/child” 子节点的 “MyComponent” 组件
        .toProperty('myProperty')           // 目标对象为 “MyComponent” 组件上的 “myProperty” 属性
        .toElement(1)                       // 目标对象为 “myProperty” 属性中的第二个数组元素
        ;
}
```

轨道路径中的子路径可以任意组合，只要它们具有正确的含义，但以下几种情况的轨道路径是无效的：

1. 空路径
2. 路径的末尾不是属性或数组元素，且未设置 **值代理**（参考下文介绍）
3. 对象属性、数组元素、节点的子节点、节点上的组件不存在时

对于无效的路径，运行时，此条轨道会被忽略并给出警告。

值代理

在轨道路径定位到目标对象后，若最后定位到的是一个 **属性**，默认情况下 Creator 将通过对该属性赋值以完成动画。

但在某些情况下，对象可能并没有提供“属性设置”接口，就不能通过赋值来完成设置。例如，材质对象是通过 `Material.prototype.setProperty(name, value)` 来改变其材质属性的值，并没有提供“属性设置”接口，这时候便可以通过在轨道指定 **值代理**，自定义赋值给目标对象。

要创建值代理，需要实现 `animation.ValueProxyFactory` 接口，代码示例如下：

```
class SetMaterialPropertyValueProxyFactory {
    /*
     * 材质属性名称。
     */
    private _propertyName: string;

    constructor (propertyName: string) {
        this._propertyName = propertyName;
    }

    /**
     * 需要实现该接口。‘target’ 是轨道路径的解析结果。
     * 返回的结果应实现值代理接口 ‘animation.ValueProxy’。
     */
    public forTarget (target: unknown): animation.ValueProxy {
        // 一个好的实现方法这里应该指定 'target' 一定是材质对象
        // asserts(target instanceof Material);
        const material = target as Material;
        return {
            set: (value) => {
                // ‘value’ 是轨道产生的值
                material.setProperty(this._propertyName, value);
            },
        };
    }
}
```

然后我们便可以设置一个能修改材质属性的动画属性轨道，代码示例如下：

```
import { MeshRenderer, animation } from 'cc';

function setupMaterialPropertyTrack(track: animation.TrackPath) {
    // 先设置轨道路径，指定目标对象为材质
    track.path
        .toHierarchy('path/to/children')
        .toComponent(MeshRenderer)
        .toProperty('materials')
        .toElement(1)
        ;

    // 应用值代理
    track.valueProxy = new SetMaterialPropertyValueProxyFactory('mainColor');
}
```

因为动画是可重用的，它可以绑定到多个对象上，Creator 支持 **不同对象应由不同的值代理**，所以 `animation.Track` 的 `valueProxy` 字段是 `animation.ValueProxyFactory` 而不是 `animation.ValueProxy`。另一方面，实现可以在 `forTarget` 这一层面做些优化。

注意：此例仅为阐述值代理的创建和使用，Creator 本身提供了用于设置材质属性（`Uniform`）的值代理工厂：`animation.UniformProxyFactory`。

循环模式

动画剪辑通过 `AnimationClip.wrapMode` 可以设置不同的循环模式。以下列出了几种常用的循环模式：

`AnimationClip.wrapMode` 说明	
`WrapMode.Normal`	播放到结尾后停止
`WrapMode.Loop`	循环播放
`WrapMode.PingPong`	从动画开头播放到结尾后，从结尾开始反向播放到开头，如此循环往复

更多循环模式，详情请参考 API WrapMode 以及文档 循环模式与循环次数。

外来动画

有些动画数据并不由轨道表示，但它以另一种形式存在于动画剪辑中，并在运行时产生动画效果。这部分动画数据称为外来动画（Exotic Animation）。外来动画旨在于让 Creator 更高效地存储和计算一些复杂的动画。

用户无法访问和编辑外来动画。由编辑器从模型中导入的骨骼动画就存储在外来动画中。

骨骼动画

骨骼动画

骨骼动画是一种常见但类型特殊的动画。

骨骼动画组件

导入带有动画文件的 模型资源 后，若模型网格中带有蒙皮信息，在使用模型时，SkeletalAnimation 组件便会自动添加到模型节点上。

或者在 **层级管理器** 中选中需要添加骨骼动画组件的节点，然后在 **属性检查器** 中选择 **添加组件 -> Animation -> SkeletalAnimation** 即可。

骨骼贴图布局设置

骨骼贴图布局设置

要确保 骨骼动画 也能够完全正确地参与 动态 Instancing，需要用户手动指定每张骨骼贴图的数据分配方式。

比如一个场景中要绘制大量相同的人物角色，每个角色可能在走/跳/攻击。如果希望一个 Drawcall 就能够正确完成所有角色的绘制，一个重要的前提条件是 **这三个动画（走、跳、攻击）的数据都储存在同一张骨骼贴图内**。

目前在默认的 预烘焙骨骼动画模式 下，骨骼贴图已经做到全局自动复用，但每张贴图的大小和它们各储存哪些动画是不可预知的。如果不做任何处理直接开启蒙皮模型的 instancing 的话，最终的运行时效果可能会出现有的动画效果正确，有的动画效果完全错乱，并且完全无法预测。

为此打开 **项目设置** 找到 **骨骼贴图布局设置** 分页，用于手动指定每张骨骼贴图中要存储哪些骨骼的哪些动画信息。

程序化控制动画

程序化控制动画

动画组件

动画组件管理了一组动画状态，用于控制各动画的播放、暂停、继续、停止、切换等。动画组件会为每一个动画剪辑都创建相应的 动画状态 对象，动画状态用于控制需要在对象上使用的动画剪辑。

在动画组件中，动画状态是通过名称来标识的，每个动画状态的默认名称就是其动画剪辑的名称。

在脚本中为节点添加动画组件的方式如下：

```
import { Animation, Node } from 'cc';

function (node: Node) {
    const animationComponent = node.addComponent(Animation);
}
```

动画的播放与切换

播放动画

动画组件通过 play() 控制指定动画的播放，例如：

```
// 播放动画状态 'idle'
animationComponent.play('idle');
```

使用 `play` 播放动画时若未指定具体动画，并且设置了 `defaultClip`，则会播放 defaultClip 动画。若动画组件的 `playOnLoad` 也设置为 `true`，则动画组件将在第一次运行时自动播放 `defaultClip` 的内容。

```
// 未指定播放的动画，并且设置了 defaultClip 的话，则会播放 defaultClip 动画
animationComponent.play();
```

切换动画

使用 `play` 接口播放一个动画时，如果此时还有其他的动画正在播放，则会立即停止其他动画的播放。这种切换是非常突兀的，在某些情况下，我们希望这种切换是“淡入淡出”的效果，那么便可以使用 crossFade()，在指定的周期内平滑地完成切换。例如：

```
// 播放动画状态 'walk'
```

```
animationComponent.play('walk');

/* ... */

// 在 0.3 秒内平滑地从走的动画切换为跑的动画
animationComponent.crossFade('run', 0.3);
```

`crossFade()` 的这种淡入淡出机制使得同一时刻可能有不止一个动画状态在播放。因此，动画组件没有 **当前动画** 的概念。

即便如此，动画组件仍提供了 `pause()`、`resume()`、`stop()` 方法，这些方法在暂停、继续以及停止正在播放的所有动画状态的同时，也暂停、继续以及停止动画的切换。

关于动画组件更多相关的控制接口，详情请参考 类 `Animation`。

动画状态

动画组件只提供了一些简单的控制函数，大部分情况下是足够和易于使用的，但若想要得到更多的动画信息以及动画控制接口，需要使用 动画状态。

帧事件

动画编辑器支持可视化编辑 事件帧，也可以直接在脚本里添加帧事件。

`AnimationClip` 的 `events` 包含了此动画所有的帧事件，每个帧事件都具有以下属性：

```
{
    frame: number;
    func: string;
    params: any[];
}
```

- `frame`：表示事件触发的时间点，单位为秒。例如 `0.618` 就表示当动画到达第 0.618 秒时将触发事件。时间轴刻度单位之间的转换，详情请参考 时间轴的刻度单位显示。
- `func`：表示事件触发时回调的函数名称。事件触发时，动画系统会搜索 **动画根节点中的所有组件**，若组件中有实现动画事件 `func` 中指定的函数，便会对其进行调用，并传入 `params` 中的参数。

例如，在动画时间轴的第 0.5s 添加了一个事件帧：

动画状态

动画状态

动画剪辑仅描述某一类对象的动画数据，例如角色的跑、走、跳等，但并未绑定具体要执行动画的对象。动画状态便是用于控制在某个对象上使用的动画剪辑，类似于播放机，除了提供动画组件也有的简单的控制函数外，还提供了更多的动画信息以及动画控制接口，允许对动画播放进行调速、设置循环模式等控制。一个动画剪辑可以同时被多个动画状态使用。

动画状态由 类 `AnimationState` 管理。

设置播放速度

首先可以通过 getState() 获取动画状态：

```
// 获取动画组件
const animationComponent = node.getComponent(Animation);
// 获取动画组件上的动画剪辑
const [ idleClip, runClip ] = animationComponent.clips;

// 获取 idleClip 的动画状态
const idleState = animationComponent.getState(idleClip.name);
```

然后设置动画播放的速度：

```
// 以二倍速播放 idleClip 动画
animationComponent.getState('idle').speed = 2.0; // speed 值越大速度越快，值越小则速度越慢
```

动画状态也提供了 `play()`、`pause()`、`resume()`、`stop()` 等用于播放控制的方法，详情可参考下文 **播放状态** 部分的内容。

播放时间

动画状态记录了动画的 **累计播放时间**。初始时累计播放时间为 0。当动画自然播放时，时间会不断累计。例如，当动画循环播放时，刚好第二次循环完毕后，累计播放时间将为 **动画周期 * 2**。

任意时刻动画所处的播放位置称为 **进度时间**，因此进度时间总是在 `[0, 动画周期]` 范围内。

- **累计播放时间** 由 `AnimationState` 的 time 字段获取，并且可以显式设置。
- **进度时间** 由 `AnimationState` 的 current 字段获取，是 **只读** 的。

动画播放的循环模式与循环次数决定了累计播放至某一时间时动画的进度时间，不管 **累计播放时间** 因为时间的推移而增加还是因为直接设置而更改，**进度时间** 都会相应发生改变。

循环模式与循环次数

动画可以播放到结尾就停止，或者一直循环播放，或者也可以先播放到结尾再从结尾播放到开头如此循环，这些统称为循环模式，由枚举 `AnimationClip.WrapMode` 表示，包括以下几种：

循环模式	说明
`AnimationClip.WrapMode.Normal`	从开头播放至结尾后停止。
`AnimationClip.WrapMode.Loop`	不断地从开头播放至结尾。
`AnimationClip.WrapMode.PingPong`	从开头播放至结尾后，再从结尾反向播放至开头，如此循环往复。

除此之外，上表中的每种循环模式还存在对应的 **反向** 循环模式：

循环模式	说明
`AnimationClip.WrapMode.Reverse`	从结尾播放至开头后停止。
`AnimationClip.WrapMode.LoopReverse`	不断地从结尾播放至开头。
`AnimationClip.WrapMode.PingPongReverse`	从结尾播放至开头后，再从开头反向播放至结尾，如此循环往复。

动画状态的初始循环模式将从动画剪辑中读取。需要改变动画状态的循环模式时，简单地设置动画状态的 `wrapMode` 字段即可。

> **注意**：设置循环模式时会重置动画状态的 **累计播放时间**。

除 `AnimationClip.WrapMode.Normal` 和其对应的 `AnimationClip.WrapMode.Reverse` 外（它们可以理解为单次循环），其余的循环模式执行的都是无限次循环。无限次循环需要与 `AnimationState` 的 `repeatCount` 配合使用才能达到效果，并且可以通过 `repeatCount` 字段来设置和获取循环的次数。

当动画循环模式为：

- 单次循环模式：`repeatCount` 将被设置为 **1**。
- 无限次循环模式：`repeatCount` 将被设置为 `Number.Infinity`，即无限循环。

> **注意**：设置循环次数应该在设置循环模式之后进行，因为重新设置循环模式时会重置循环次数。

播放控制

动画状态提供了以下几种方法用于控制动画的播放、暂停、恢复和停止：

方法	说明
`play()`	重置播放时间为 0 并开始播放动画。
`pause()`	暂停动画。
`resume()`	从当前时间开始继续播放动画。
`stop()`	停止播放动画。

也可以通过以下字段查询动画的播放状态：

字段（只读）	说明
`isPlaying`	动画是否处于播放状态。
`isPaused`	动画是否处于暂停状态。
`isMotionless`	动画是否处于暂停状态或者已被停止。

播放控制与播放状态之间的关系如下图所示：

变形动画

变形动画（Morph）

嵌入播放器

嵌入播放器（实验性功能）

嵌入播放器的作用是允许在动画剪辑的制作时，可以同时绑定粒子特效或其他动画等，用于满足在播放动画同时播放动效的功能。可以用于实现挥动武器会出现刀光，或者脚踩在地面上会出现烟尘等功能。

在动画窗口展开 **播放器轨道列表** 即可观察整个嵌套系统：

Marionette 动画系统

Marionette 动画系统

Cocos Creator 3.4 引入了一个全新的 Marionette 动画系统，通过状态机控制对象的骨骼动画，实现了自动化、可复用的动画流程。

为了跟 v3.4 之前的动画系统区分，我们将新的动画系统称为木偶（Marionette）动画系统，称 v3.4 之前使用的动画系统为旧式动画系统。两种动画系统都可以正常使用，但不支持同时使用。主要的区别在于：

- 旧式动画系统：以动画组件、动画状态为核心，手动简单控制动画剪辑的播放暂停等。动画剪辑支持使用通过编辑器创建的 Animation Clip 和外部导入的骨骼动画（`.fbx` 、 `.gltf` 和 `.glb`）。
- Marionette 动画系统：以动画控制器组件、动画图为核心，按照事先搭建好的动画图，通过状态机自动控制动画剪辑的播放和切换等。动画剪辑只支持外部导入的骨骼动画（`.fbx` 、 `.gltf` 和 `.glb`）。

内容

Marionette 动画系统主要内容包括：

- 动画图资源
- 动画控制器组件参考
- 动画图面板
- 动画图层级
- 动画状态机
- 状态过渡
- 动画遮罩资源
- 动画图变体
- 程序化动画

名词解释

Marionette 动画系统相关功能名词的说明如下：

功能名词	说明
动画图资源	用于存储对象的整个动画流程数据，可直接在 **资源管理器** 面板创建。详情请参考 动画图资源。
动画控制器组件	引用动画图资源并将其应用于对象。详情请参考 动画控制器组件参考。
动画图面板	当准备好对象所需的骨骼动画后，便可通过动画图面板将其组合成完整的动画流程。具体的操作请参考 动画图面板。
状态	对象所处的一种播放特定动画剪辑的动作，例如待机、行走、移动、攻击等。 该状态与旧式动画系统中，动画组件为每一个动画剪辑创建的 动画状态 不同。
状态过渡	大多数情况下，一个对象会拥有多个状态，并按照一定的需求逻辑在各个状态之间切换，这种切换便称为 状态过渡。例如角色在行走时触发了死亡条件，行走状态便会切换到死亡状态。
动画状态机	用于可视化地管理控制某个对象上各个状态及状态之间的过渡，可视为一个流程图。详情请参考 动画状态机。

状态及状态过渡在动画图面板中是以图形的方式显示，例如下图，方块表示状态，箭头表示状态之间的过渡：

动画图资源

动画图资源

动画图资源用于存储对象的整个动画流程数据，通过状态机描述动画的流程，目前一个动画图只支持一个状态机。

创建

在 **资源管理器** 点击左上方的 + 按钮，然后选择 **动画图（Animation Graph）**：

动画控制器组件参考

动画控制器（Animation Controller）组件参考

动画控制器组件用于将 动画图资源 应用到对象上。

在 **层级管理器** 中选中需要应用动画图的节点，然后在 **属性检查器** 中点击下方的 **添加组件 -> Animation -> Animation Controller**，即可添加一个动画控制器组件到节点上。

动画图面板

动画图面板

动画图面板用于查看和编辑动画图资源。当准备好对象所需的骨骼动画后，便可在 **动画图面板** 中将其组装结合成完整的动画流程。

打开面板

在 **资源管理器** 中选中动画图资源，然后在 **属性检查器** 中点击 **编辑**：

层级（Layers）

该分页主要用于创建、查看和编辑动画图的层级，每个层级分别由一个状态机控制，点击 **Layers** 按钮可以进入层级编辑状态。

在 v3.5.0 中，我们增加了新的机制 **下一帧重置（Reset on next frame）**，在勾选后，触发器会在动画更新后立即重置。

动画图层级

动画图层级

动画图可以有多个层级，层级之间同时运行，其上播放的动画效果可以根据层级的配置进行混合。

创建层级

动画状态机

动画状态机

我们把对象所处的播放某种特定动画的动作，例如待机、移动、奔跑、攻击等，称为 **状态**。
一般情况下，一个对象拥有多个状态，并按照一定的逻辑顺序在状态之间切换，以执行不同的动作，这种切换在动画图中称为 过渡。发生过渡时需要满足的条件称为 **过渡条件**。

而动画状态机则是用于管理和控制对象上各个状态及状态之间的过渡，类似于流程图，我们可以直接在 动画图面板 中对其进行可视化编辑。目前一个动画图包含一个状态机，当状态机位于动画图中的某个状态时，便会播放该状态对应的动画。那么按照事先搭建好的动画流程图，通过状态机便可自动控制骨骼动画的播放和切换等。

状态基础

状态机中的状态除了用于控制动画播放的状态，还包括了 **入口**、**出口**、**任意**、**子状态机** 这几个特殊的状态，因为其本身并不承载动画，只是为了标志状态机开始、结束等特殊用途，因此为了与状态区分我们暂且称之为“伪状态”。

子状态机 是嵌套在状态机中的，用于降解复杂的动画逻辑，具体可查看下文 **子状态机** 部分的内容。一般我们将动画图最上层的状态机称为 **顶层状态机**，类似于场景中的根节点。

入口（Entry）、**出口**（Exit）、**任意**（Any）则是默认存在于状态机中的固定单元，不可删除。在打开动画图编辑状态机时便可在网格布局区域中看到，需要注意的是 **出口** 只存在于 **子状态机** 中。

状态机事件绑定

状态机事件绑定用于在状态机发生特定事件时进行相关的处理。

每个事件绑定需要指定一个 **方法名称**，当事件发生时，动画控制器所在结点上的所有组件的同名方法将被调用。

状态相关事件

当选中状态对象时，可在属性检查器中绑定该状态的进入、退出事件。

状态过渡

状态过渡

过渡 代表着两个状态之间的转换，根据过渡发生源头的不同，我们可以将其分为：

- **普通过渡**：过渡发生的源头为伪状态 **入口** 或者 **子状态机**。

动画遮罩资源

动画遮罩资源

动画遮罩是一种资源（Asset），用在动画图层级中，用于屏蔽指定骨骼上的动画效果。

常见的应用场景如上下半身分开播放动画，假如要制作一个在射击时行走的动画，可以通过动画遮罩对射击动画的上半身和行走动画的下半身进行混合来实现。

创建

在 **资源管理器** 点击左上方的 + 按钮，然后选择 **动画遮罩（Animation Mask）**：

动画图变体

动画图变体

如果两个动画图他们的动画逻辑相同但对应的动画剪辑不同，重新制作一个动画图会带来额外的维护负担。通过使用动画图变体，可以更好的解决此问题。

动画图变体是一种资源，在 **资源管理器** 内右键创建一个新的动画图变体资源：

程序式动画

程序式动画（实验性）

> **注意**：程序式动画是实验性功能，请谨慎使用。Cocos Creator 期待你的反馈。

Cocos Creator 3.8 开始实验性地支持程序式动画效果，使得开发者有能力实现一些复杂的骨骼动画。

内容

- 在 程序式动画简介 中，介绍了关于程序式动画的起因和基础。
- 程序式动画目前作为 Cocos Creator 实验性功能，默认关闭，启用程序式动画功能 中讲述了如何对其进行开启。
- 姿态图 中介绍了程序式动画的核心构件——姿态图。
- 加性动画 是一种复用动画资产的手段。
- 除姿态图外，辅助曲线 也是实现程序式动画的有效手段。
- 姿态暂存 用于实现姿态图的复用。

程序式动画

程序式动画简介

在了解程序化动画之前，我们需要先了解动画中所指的“姿势（Pose）”，相信下面的图中的姿势您都有见过：

启用程序式动画功能

启用程序式动画功能

程序式动画目前为实验性功能，如要启用，请在 **偏好设置** 中找到 **实验室** 功能并开启下列选项：

- 动画辅助曲线（**Animation auxiliary curve**）
- 开启姿态图功能（**Enable Pose-Express Function**）

姿态图

姿态图（实验性）

姿态图 是表达了姿态的产生和转变的结点图，是实现程序式动画的核心构件。

姿态图依附在动画图的其它构件中：

- 状态机中，每个 程序式姿态（状态）包含一张姿态图。
- 每个 姿态暂存 中包含一张姿态图。

本节将讲述姿态图的结构。

在开始了解程序式动画之前，我们建议您阅读和 Marionette 动画系统 和动画图相关的内容。

值及类型

广泛来说，姿态图描述了值的流动。每个值都有类型，姿态图支持的值的类型如下：

类型	含义
姿态对象	角色姿态。
浮点数	浮点数值。
整数	整数值。
三维向量	三维向量。
四元数	四元数。

其中，姿态对象无法凭空产生和指定，仅由姿态结点产生（见下文）。

结点

姿态图节点视图

姿态图节点视图

创建节点

创建节点前需要先在动画图中创建一个姿势图，之后通过鼠标双击进入到姿势图的可视化编辑界面内。

在姿势图内点击鼠标右键，在弹出菜单中的姿势图节点（Post Nodes）单击鼠标既可以创建姿势图节点。

姿态结点

姿态结点

动画图中可以创建姿态图并对姿态图进行编辑，编辑后的结果将作为动画图资产的一部分存储在动画图内部。 您可以参考动画图的用法来创建、删除和修改姿态图。

本章我们将向您介绍姿态图的概念以及用法。

姿态结点输出姿态。引擎提供了以下作用的姿态结点：

- 播放或采样动画
- 混合姿态
- 选择姿态
- 修改姿态
- 状态机
- 使用暂存姿态

可视化姿态图节点编辑器器的内容请参考 姿态图节点视图。

内容

- 姿态图节点操作

混合姿态

混合姿态

姿态图提供了几种用于混合姿态的结点。

这些结点的混合都在本地空间中进行。

混合姿态可以在姿态图中点击鼠标右键选择 **姿态节点** -> **混合** 来创建。

修改姿态

修改姿态

姿态图提供了几种用于修改姿态的结点。这些结点都接受一个输入姿态，并输出修改后的姿态。

输入	类型	含义
姿态	姿态	要修改的姿态。

姿态节点的操作请参考 姿态图节点视图。

应用变换

播放或采样动画

播放或采样动画

姿态图提供了两种从动画中读取姿态的结点：播放动画结点和采样动画结点。

在下文，“动作”是指动画剪辑或动画混合。

播放动画

音频系统

音频系统

音乐是游戏中不可或缺的一部分，好的音乐能让游戏更加真实、富有沉浸感。Cocos Creator 的音频系统支持导入并播放大多数常见的音频文件格式，具体内容请参考：

- 音频资源
- AudioSource 组件参考
- AudioSource 播放示例
- 兼容性说明

AudioSource 组件参考

AudioSource 组件参考

AudioSource 组件用于控制音乐和音效的播放。

全局音频管理器示例

音频播放管理器示例

由于 Cocos Creator 3.x 移除了 v2.x `cc.audioEngine` 系列的 API，统一使用 AudioSource 控制音频播放。

但在实际项目开发中，我们仍然需要一个方便随时调用的音频播放管理器，可参考或直接使用以下代码：

```
//AudioMgr.ts
import { Node, AudioSource, AudioClip, resources, director } from 'cc';
/**
 * @en
 * this is a sington class for audio play, can be easily called from anywhere in you project.
 * @zh
 * 这是一个用于播放音频的单件类，可以很方便地在项目的任何地方调用。
 */
export class AudioMgr {
    private static _inst: AudioMgr;
    public static get inst(): AudioMgr {
        if (this._inst == null) {
            this._inst = new AudioMgr();
        }
        return this._inst;
    }

    private _audioSource: AudioSource;
    constructor() {
        //@en create a node as audioMgr
        //@zh 创建一个节点作为 audioMgr
        let audioMgr = new Node();
        audioMgr.name = '__audioMgr__';

        //@en add to the scene.
        //@zh 添加节点到场景
        director.getScene().addChild(audioMgr);

        //@en make it as a persistent node, so it won't be destroied when scene change.
        //@zh 标记为常驻节点，这样场景切换的时候就不会被销毁了
        director.addPersistRootNode(audioMgr);

        //@en add AudioSource componnrnt to play audios.
        //@zh 添加 AudioSource 组件，用于播放音频。
        this._audioSource = audioMgr.addComponent(AudioSource);
    }

    public get audioSource() {
        return this._audioSource;
    }

    /**
```

```
     * @en
     * play short audio, such as strikes,explosions
     * @zh
     * 播放短音频,比如 打击音效，爆炸音效等
     * @param sound clip or url for the audio
     * @param volume
     */
    playOneShot(sound: AudioClip | string, volume: number = 1.0) {
        if (sound instanceof AudioClip) {
            this._audioSource.playOneShot(sound, volume);
        }
        else {
            resources.load(sound, (err, clip: AudioClip) => {
                if (err) {
                    console.log(err);
                }
                else {
                    this._audioSource.playOneShot(clip, volume);
                }
            });
        }
    }

    /**
     * @en
     * play long audio, such as the bg music
     * @zh
     * 播放长音频，比如 背景音乐
     * @param sound clip or url for the sound
     * @param volume
     */
    play(sound: AudioClip | string, volume: number = 1.0) {
        if (sound instanceof AudioClip) {
            this._audioSource.stop();
            this._audioSource.clip = sound;
            this._audioSource.play();
            this.audioSource.volume = volume;
        }
        else {
            resources.load(sound, (err, clip: AudioClip) => {
                if (err) {
                    console.log(err);
                }
                else {
                    this._audioSource.stop();
                    this._audioSource.clip = clip;
                    this._audioSource.play();
                    this.audioSource.volume = volume;
                }
            });
        }
    }

    /**
     * stop the audio play
     */
    stop() {
        this._audioSource.stop();
    }

    /**
     * pause the audio play
     */
    pause() {
        this._audioSource.pause();
    }

    /**
     * resume the audio play
     */
    resume(){
        this._audioSource.play();
    }
}
```

兼容性说明

兼容性说明

Web 平台音频资源的加载模式

Web 平台上的音频资源比较特别，因为 Web 标准支持以两种不同的方式加载音频资源，分别是：

- Web Audio：提供相对更加现代化的声音控制接口，在引擎内是以一个 audio buffer 的形式缓存的。这种方式的优点是兼容性好，问题比较少。
- DOM Audio：通过生成一个标准的 audio 元素来播放音频资源，在引擎内缓存的就是这个 audio 元素。使用标准的 audio 元素播放音频资源时，在某些浏览器上可能会遇到一些兼容性问题，比如 iOS 上的浏览器不支持调整音量大小，所有 volume 相关属性将不会生效。

目前 Cocos Creator 默认以 Web Audio 的方式加载音频资源，但如果检测到当前浏览器不支持加载 Web Audio，则会切换使用 DOM Audio 的方式加载音频。

如果项目需要强制通过 DOM Audio 的方式加载音频资源，请使用以下方式动态加载：

```
assetManager.loadRemote('http://example.com/background.mp3', {
    audioLoadMode: AudioClip.AudioType.DOM_AUDIO
}, (err, clip: AudioClip) => {
    if(err){
        console.log(err);
    }
});
```

物理系统

物理系统

Cocos Creator 的物理系统提供了高效的组件化工作流程和便捷的使用方法。目前支持刚体、碰撞组件、触发和碰撞事件、物理材质、射线检测等等特性。

2D 物理

2D 物理简介

Cocos Creator 支持内置的轻量 Builtin 物理系统和强大的 Box2D 物理系统。Builtin 物理系统只提供了碰撞检测的功能，对于物理计算较为简单的情况，我们推荐使用 Builtin 物理模块，这样可以避免加载庞大的 Box2D 物理模块并构建物理世界的运行时开销。而 Box2D 物理模块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。

你可以根据需要来选择适合自己的物理模块，通过编辑器主菜单中的 **项目 -> 项目设置 -> 功能裁剪** 切换物理模块的使用。

2D 物理系统

2D 物理系统

物理系统隐藏了大部分物理模块（Box2D 和 Builtin 模块）实现细节（比如创建刚体，同步刚体信息到节点中等）。

可以通过物理系统访问一些物理模块常用的功能，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

物理系统默认是开启的，代码如下：

```
PhysicsSystem2D.instance.enable = 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，可以通过组合这些信息来绘制相关的内容。

```
PhysicsSystem2D.instance.debugDrawFlags = EPhysics2DDrawFlags.Aabb |
    EPhysics2DDrawFlags.Pair |
    EPhysics2DDrawFlags.CenterOfMass |
    EPhysics2DDrawFlags.Joint |
    EPhysics2DDrawFlags.Shape;
```

设置绘制标志位为 **EPhysics2DDrawFlags.None**，即可以关闭绘制。

```
PhysicsSystem2D.instance.debugDrawFlags = EPhysics2DDrawFlags.None;
```

物理单位到世界坐标系单位的转换

一般物理模块（Box2D）都是使用 **米 - 千克 - 秒（MKS）** 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 **世界坐标系中的单位**（简称世界单位）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世界单位上的相互转换。

一般情况下我们把这个比率设置为 32，这个值可以通过 `PHYSICS_2D_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世界单位进行转换，用户访问和设置的都是进行 2d 游戏开发中所熟悉的世界单位。

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, -10)` 米/秒^2，按照上面描述的转换规则，即 `(0, -320)` 世界单位/秒^2。

如果希望重力加速度为 0，可以这样设置：

```
PhysicsSystem2D.instance.gravity = v2();
```

如果希望修改重力加速度为其他值，比如每秒加速降落 20m/s，那么可以这样设置：

```
PhysicsSystem2D.instance.gravity = v2(0, -20 * PHYSICS_2D_PTM_RATIO);
```

设置物理步长

物理系统是按照一个固定的步长来更新物理世界的，默认的步长是 `1/60`。但是有的游戏可能会不希望按照这么高的频率来更新物理世界，毕竟这个操作是比较消耗时间的，可以通过降低步长来达到这个效果。代码示例如下：

```
const system = PhysicsSystem2D.instance;

// 物理步长，默认 fixedTimeStep 是 1/60
system.fixedTimeStep = 1/30;

// 每次更新物理系统处理速度的迭代次数，默认为 10
system.velocityIterations = 8;

// 每次更新物理系统处理位置的迭代次数，默认为 10
system.positionIterations = 8;
```

注意：降低物理步长和各个属性的迭代次数，都会降低物理的检测频率，所以会更有可能发生刚体穿透的情况，使用时需要考虑到这个情况。

通过 物理配置 也可以在 **项目设置** -> **物理** 内修改重力和物理步长。

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。例如，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方便用户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。

```
const collider = PhysicsSystem2D.instance.testPoint(point);
```

矩形测试

矩形测试将测试世界坐标系下指定的一个矩形，如果一个碰撞体的包围盒与这个矩形有重叠部分，则这个碰撞体会被添加到返回列表中。

```
const colliderList = PhysicsSystem2D.instance.testAABB(rect);
```

射线测试

Box2D 物理模块（Builtin 模块没有）提供了射线检测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。

```
const results = PhysicsSystem2D.instance.raycast(p1, p2, type, mask);

for (const i = 0; i < results.length; i++) {
    const result = results[i];
    const collider = result.collider;
    const point = result.point;
    const normal = result.normal;
    const fraction = result.fraction;
}
```

射线检测的第三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。Cocos Creator 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。

- ERaycast2DType.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。

- ERaycast2DType.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。

- ERaycast2DType.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。

- ERaycast2DType.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。

2D 刚体组件

2D 刚体

刚体是组成物理世界的基本对象，可以将刚体想象成一个你不能看到（绘制）也不能摸到（碰撞）的且不能变形的物体。

由于 Builtin 2D 物理系统只带有碰撞检测的功能，所以刚体对于 Builtin 2D 物理系统是不生效的，本篇设置只对 Box 2D 物理系统产生作用。

添加刚体

点击 **属性检查器** 的 **添加组件** 按钮，输入 Rigidbody2D 即可以添加 2D 刚体组件。

2D 碰撞体

2D 碰撞组件

添加碰撞组件

目前引擎支持三种不同的碰撞组件： **盒碰撞组件（BoxCollider2D）**、**圆形碰撞组件（CircleCollider2D）** 和 **多边形碰撞组件（PolygonCollider2D）**。在 **属性检查器** 上点击 **添加组件** 按钮，输入碰撞组件的名称即可添加。

2D 碰撞回调

2D 碰撞回调

当物体在场景中移动并碰撞到其它物体时，物理引擎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. **Box2D** 物理模块需要先在 Rigidbody 中 **开启碰撞监听**，才会有相应的回调产生。开启方法为，在 **Rigidbody2D** 的 **属性检查器** 勾选 **EnabledContactListener** 属性，如下图所示：

2D 物理关节

2D 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创建一个真实有趣的场景。

注意：

1. 关节组件在 Builtin 2D 物理模块中是无效的。
2. 关节组件都需要搭配 刚体 才可以正确运行。如下图所示：

3D 物理系统

3D 物理系统

3D 物理简介

设置物理引擎

设置物理引擎

点击编辑器上方菜单栏的 **项目 -> 项目设置 -> 功能裁剪**，在 **3D -> 物理系统** 中可以根据需要选择适合项目的物理引擎进行开发，也可以在开发过程中随时切换。目前 Creator 支持的物理引擎包括 **Bullet（ammo.js）**、**cannon.js**、**PhysX** 和 **builtin**，默认使用 **Bullet（ammo.js）**。

物理系统配置

物理系统配置

物理系统模块（PhysicsSystem）用于管理整个物理系统，负责同步物理元素、触发物理事件和调度物理世界的迭代。

物理配置

有两种办法可以配置物理系统，一种是在编辑器中配置，另一种是通过代码配置。

通过物理配置面板

通过 **项目设置 -> 物理配置** 可以对物理系统进行相关配置。

分组和掩码

分组和掩码

分组和掩码是物体之间能否进行物理碰撞检测的必要条件。分组可以简单的理解为一个碰撞对象所处的分组，掩码可以简单的理解为该碰撞对象需要与哪些分组对象进行碰撞。

碰撞检测原理

在 Cocos Creator 中是否进行碰撞检测采用的是二进制 按位操作，通过将分组值与掩码值进行“与”运算，从而判断是否满足条件。对象间允许进行碰撞检测的计算方式如下：

```
(GroupA & MaskB) && (GroupB & MaskA)
```

从这个公式中可以看出，分组 A 需要满足在分组 B 的掩码列表里并且分组 B 也需要满足在分组 A 的掩码列表里，这样的两个对象之间就能进行碰撞检测。如何将二进制操作与允许进行碰撞检测计算公式结合起来，这是下面需要了解的部分。但在这之前，需要用户在 **项目设置 -> 物理 -> 碰撞矩阵** 处先配置 碰撞分组。

物理组件

物理组件

Cocos Creator 目前为用户提供的物理组件分为以下几种：

- 碰撞组件

  碰撞组件用于定义碰撞体形状。可在编辑器环境下查看其形状，运行时不可见。其形状可以根据需求设置，无需和对象网格完全相同。

- 刚体组件

  刚体是组成物理世界的基本对象，它可以使游戏对象的运动方式受物理控制。

- 恒力组件

  恒力组件是一个工具组件，依赖于刚体，会在每帧对一个刚体施加给定的力和扭矩。

- 约束组件

  约束组件依赖于刚体组件，可将某个对象的位置、方向或比例约束到其他对象。

碰撞体

碰撞组件与基础属性

碰撞组件可用于定义需要进行物理碰撞的物体形状，不同的几何形状拥有不同的属性。碰撞体通常分为以下几种：

1. 基础碰撞体。常见的包含 盒、球、圆柱、圆锥、胶囊 碰撞体。
2. 复合碰撞体。可以通过在一个节点身上添加一个或多个基础碰撞体，简易模拟游戏对象形状，同时保持较低的性能开销。
3. 网格碰撞体。根据物体网格信息生成碰撞体，完全的贴合网格。
4. 单纯形碰撞体。提供点、线、三角面、四面体碰撞。
5. 平面碰撞体。可以代表无限平面或半空间。这个形状只能用于静态的、非移动的物体。
6. 地形碰撞体。一种用于凹地形的特殊支持。

   注意：在某些（如 Bullet）物理后端中，由于计算精度的原因，应该避免使用比例很高的尺寸，这里建议低于1000。如某个盒碰撞器，其 **Size** 属性的 Y 值为 40 而 Z 值为 0.01，此时他们的 Y、Z 的比例超过了 1000，此时可能会出现浮点数计算不准确的问题。

添加碰撞组件

这里以获取 **BoxCollider** 盒碰撞器组件为例。

通过编辑器添加

1. 新建一个 3D 对象 Cube，在 **资源管理器** 中点击左上角的 + 创建按钮，然后选择 **创建 -> 3D 对象 -> Cube 立方体**。

刚体

刚体组件

刚体是组成物理世界的基本对象，它可以使游戏对象的运动方式受物理控制。例如：刚体可以使游戏对象受重力影响做自由下落，也可以在力和扭矩的作用下，让游戏对象模拟真实世界的物理现象。

添加刚体

通过编辑器添加

点击 **属性检查器** 下方的 **添加组件 -> Physics -> RigidBody**，即可添加刚体组件到节点上。

恒力组件

恒力组件

恒力组件是一个工具组件，依赖于刚体组件，每帧都会对一个刚体施加给定的力和扭矩。

约束

约束

在物理引擎中，**约束** 用于模拟物体间的连接情况，如连杆、绳子、弹簧或者布娃娃等。

约束依赖 刚体组件，若节点无刚体组件，则添加约束时，引擎会自动添加刚体组件。

> **注意**：目前的约束仅在物理引擎选择为 Bullet、PhysX 或 Cannon.js 的情况下生效。

铰链约束 HingeConstraint

通过铰链约束，将连接物体的运动约束在某一个轴上，这种约束在模拟门的合页或电机转动等情形下非常有用。

物理材质

物理材质

物理材质是一种资源，它记录了物体的物理属性，这些信息用来计算碰撞物体受到的摩擦力和弹力等。

创建物理材质

在编辑器内创建

在 **属性检查器** 内右键任意空白处或点击 + 号间都可以创建物理材质：

物理事件

物理事件

触发器与碰撞器

射线检测

射线和线段检测

本文将说明如何通过射线和线段对物理世界内的碰撞体进行检测。

射线检测

射线检测

射线检测是对一条射线和另一个形状进行 **相交性判断**，如下图所示。

几何投射检测

几何投射检测

Cocos Creator 从 v3.8 开始，支持扫掠功能。

几何投射检测，会沿着指定的射线，发射不同的几何体，就像使用某个几何体沿着射线扫过一个区域，因此又称扫掠。扫掠会对几何体扫过的物理世界区域内的碰撞体进行检查，并返回特定的结果。

和 `rayCast` 射线检测发射的射线不同，`sweep` 允许物理引擎投射不同的几何体，并返回特定的碰撞信息。

目前引擎提供以下 **盒形**、**球形** 以及 **胶囊体** 扫掠。

方法

- 盒形：
  - `sweepBox`：沿着给定的射线投射一个盒，并返回所有命中的碰撞体
  - `sweepBoxClosest`：沿着给定的射线投射一个盒，并返回最近命中的碰撞体
  - 参数说明：
    - worldRay：geometry.Ray：世界空间下的一条射线
    - halfExtent：IVec3Like 盒体的一半尺寸，三维矢量的 xyz 各代表盒在每个轴上的大小的一半
    - orientation：IQuatLike： 盒体的方向
    - mask: number：掩码，默认为 0xffffffff，请参考 分组和掩码 以及 射线检测
    - maxDistance: number 最大检测距离，默认为 10000000，目前请勿传入 Infinity 或 Number.MAX_VALUE
    - queryTrigger：boolean 默认为 true，是否检测触发器
- 球形
  - `sweepSphere`：沿着给定的射线投射一个球形，并返回所有命中的碰撞体
  - `sweepSphereClosest`：沿着给定的射线投射一个球形，并返回最近命中的碰撞体
  - 参数说明：
    - worldRay：geometry.Ray：世界空间下的一条射线
    - radius：number 球体的半径
    - mask：number, 掩码，默认为 0xffffffff，请参考 分组和掩码 以及 射线检测
    - maxDistance：number 最大检测距离，默认为 10000000，目前请勿传入 Infinity 或 Number.MAX_VALUE
    - queryTrigger：boolean 默认为 true，是否检测触发器
- 胶囊体
  - sweepCapsule：沿着给定的射线投射一个胶囊体，并返回所有命中的碰撞体
  - sweepCapsuleClosest：沿着给定的射线投射一个胶囊体，并返回最近命中的碰撞体
  - 参数说明：
    - worldRay：geometry.Ray：世界空间下的一条射线
    - radius：number：胶囊体的半径
    - height：number：胶囊体末端两个半球圆心的距离
    - orientation：IQuatLike： 胶囊体的朝向

- mask：number, 掩码，默认为 0xffffffff，请参考 分组和掩码 以及 射线检测
- maxDistance：number 最大检测距离，默认为 10000000，目前请勿传入 Infinity 或 Number.MAX_VALUE
- queryTrigger：boolean 默认为 true，是否检测触发器

详细说明请参考 API。

返回值

函数的返回值为 boolean，用于确定表示是否有检测到碰撞。为保证扫掠的性能，扫掠方法的结果都被存储在 `PhysicsSystem` 内。使用时请先通过 `sweepCastClosestResult/sweepCastResults` 获取到检测结果，下次扫掠后，之前的结果可能会被覆盖或失效，因此建议在获取到结果之后，将结果内的内容提取出来使用。

- `sweepBoxClosest`、`sweepSphereClosest` 以及 `sweepCapsuleClosest` 方法的结果被保存在 `PhysicsSystem.instance.sweepCastResults` 内，其类型为 `PhysicsRayResult`，代码示例如下

```
const result = PhysicsSystem.instance.sweepCastClosestResult;
```

- `sweepBox`、`sweepSphere` 和 `sweepCapsule` 方法的结果存储在 `PhysicsSystem.instance.sweepCastResults` 内，其为 `PhysicsRayResult` 类型的数组。代码示例如下

```
const results = PhysicsSystem.instance.sweepCastResults;

for (let i = 0; i < results.length; i++) {
    const result = results[i];
    ...
}
```

`PhysicsRayResult` 的描述如下：

- hitPoint: vec3 在世界坐标系下的击中点
- distance: number 击中点到射线原点的距离
- collider: Collider 击中的碰撞盒
- hitNormal: vec3 在世界坐标系下击中面的法线

API 参考 PhysicsRayResult。

示例

以盒型扫掠为例，使用方法如下：

```
if (PhysicsSystem.instance.sweepSphereClosest(this._ray, this._sphereRadius * this._scale, this._mask, this._maxDistance, this._queryTrigger)) {
    const result = PhysicsSystem.instance.sweepCastClosestResult;
}
```

更多示例请参考 GIT。

下载示例后，打开 **case-physics-sweep** 场景，运行即可查看扫掠的结果。

连续碰撞检测

连续碰撞检测

连续碰撞检测（简称 **CCD**）是一种用于避免高速运动的物体在离散运动时出现穿透现象而导致碰撞数据不精确的技术，可用于实现类似高速运动的子弹与物体发生碰撞时，不会直接穿透物体。

开启 CCD

CCD 默认为禁用状态，若要启用，将物体 RigidBody 组件的 `useCCD` 属性设置为 `true` 即可：

```
const rigidBody = this.getComponent(RigidBody);
rigidBody.useCCD = true;
```

解决线性穿透问题

穿透是由于物体离散运动而导致的现象，三维物体的运动可以拆分为 **移动** 和 **旋转**。Creator 目前仅支持解决 **线性穿透** 问题，指的便是因物体离散的 **移动** 而导致的穿透现象。

穿透现象通常发生在高速运动的物体上，例如子弹类的物体。这是由于计算机的模拟都是基于离散化形式，当物体运动速度过快时，就会导致单次积分的能量过大，便可能导致物体穿越了本应该碰撞的另一物体，如下图所示：

角色控制器

角色控制器

自 v3.8 开始，Cocos Creator 提供角色控制器。

角色控制可以为您的游戏添加简单易用的角色控制功能。

添加角色控制器

Cocos Creator 提供两种角色控制器类型：盒控制器和胶囊控制器。他们都继承自 `CharacterController`。

需要注意的是，角色控制器仅支持 **Bullet** 以及 **PhysX** 物理后端。请在编辑器顶部菜单 **项目** -> **项目设置** 中的 **功能剪裁** 分页中找到 **物理系统**，并将物理后端修改为 **Bullet Based Physics System** 或 **PhysX Based Physics System**（默认的物理后端是 Bullet）。

物理应用案例

物理应用案例

射箭案例

1. 用基础形状组合十字架——复合形状

   下图中用两个盒形状组合一个十字架，节点上所有的碰撞体组合成了一个十字形状，这是实现带有凹面形状最基础的方法：

粒子系统

粒子系统

粒子系统是游戏引擎特效表现的基础，它可以用于模拟的火、烟、水、云、雪、落叶等自然现象，也可用于模拟发光轨迹、速度线等抽象视觉效果。

2D 粒子

ParticleSystem2D 组件参考

2D 粒子组件（ParticleSystem2D）用于读取粒子资源数据，并对其进行一系列例如播放、暂停、销毁等操作。粒子资源支持 `plist` 文件和图片，这两个资源建议放在同一个文件夹下。

3D 粒子

3D 粒子系统

粒子系统是游戏引擎特效表现的基础，它可以用于模拟的火、烟、水、云、雪、落叶等自然现象，也可用于模拟发光轨迹、速度线等抽象视觉效果。

基本结构

粒子系统的基本单元是粒子，一个粒子一般具有位置、大小、颜色、速度、加速度、生命周期等属性。在每一帧中，粒子系统一般会执行如下步骤：

1. 产生新的粒子，并初始化
2. 删除超过生命周期的粒子
3. 更新粒子的动态属性
4. 渲染所有有效的粒子

粒子系统一般由以下几个部分组成：

1. 发射器，用于创建粒子，并初始化粒子的属性
2. 影响器，用于更新粒子的属性
3. 渲染器，渲染粒子
4. 粒子类，存储粒子的属性
5. 粒子系统类，管理上面的模块

添加 3D 粒子系统

在编辑器中添加粒子系统组件有以下两种方式：

1. 在 **层级管理器** 中选中节点，然后点击右侧 **属性检查器** 面板上的 **添加组件** 按钮添加粒子，如下图：

粒子系统模块

粒子系统功能介绍

Cocos Creator 的 **粒子系统** 存储了粒子发射的初始状态以及粒子发射后的状态更新子模块。

粒子系统模块

Cocos Creator 粒子系统操作面板如下：

主模块

主模块（ParticleSystem）

粒子系统主模块用于存储在 **属性检查器** 中显示的所有数据，管理粒子生成、播放、更新，以及销毁。

发射器模块

发射器模块（ShapeModule）

发射器模块主要用于设置粒子发射器形状、粒子发射方向和速度。

速度模块

速度模块（VelocityOvertimeModule）

速度模块用于控制粒子在生命周期内的速度。

加速度模块

加速度模块（ForceOvertimeModule）

通过该模块可对粒子进行加速，以模拟类似风的效果。

大小模块

大小模块（SizeOvertimeModule）

大小模块用于设置粒子在其生命周期内的大小，从而实现类似大小不一的火焰和雪花等粒子效果。

旋转模块

旋转模块（RotationOvertimeModule）

旋转模块用于设置粒子运行时在移动中旋转，可用于模拟类似下落的雪花这类随机旋转特效。

颜色模块

颜色模块（ColorOvertimeModule）

颜色模块用于设置粒子颜色及粒子在生命周期内的颜色变化。

贴图动画模块

贴图动画模块（TextureAnimationModule）

贴图动画模块用于将 渲染模块 中 **ParticleMaterial** 属性指定的贴图纹理作为动画帧进行动态播放，用于实现类似下图中的效果：

限速模块

限速模块（LimitVelocityOvertimeModule）

限速模块用于设置粒子的速度在生命周期内逐渐减缓。

拖尾模块

轨迹模块（TrailModule）

轨迹模块用于在粒子尾部添加一个轨迹效果，实现类似下图中的拖尾效果：

渲染模块

渲染模块（Renderer）

渲染模块用于生成粒子渲染所需要的数据。

粒子属性编辑

粒子属性编辑

Creator 提供了 **粒子控制面板**、**曲线编辑器** 和 **渐变色编辑器**，用于控制粒子播放，以及编辑粒子属性等。

详情请参考：

- 粒子控制面板
- 曲线编辑器
- 渐变色编辑器

控制面板

粒子控制面板

在 **层级管理器** 选中粒子节点时，**粒子控制面板** 便会默认显示在 **场景编辑器** 右下角：

粒子曲线编辑器

粒子曲线编辑器

粒子曲线编辑器 支持设置粒子系统中部分属性随时间变化的曲线。属性默认都是以常量（Constant）的形式显示，点击属性框右侧的

渐变色编辑器

渐变色编辑器

渐变色编辑器 支持设置粒子系统中部分属性颜色随时间发生渐变。属性默认都是以固定颜色（Color）的形式显示，点击属性框右侧的

缓动系统

缓动系统

缓动接口

缓动接口

Tween 属性和接口说明

接口说明

接口	功能说明
tag	为当前缓动添加一个数值类型（`number`）的标签
to	添加一个对属性进行 **绝对值** 计算的间隔动作
by	添加一个对属性进行 **相对值** 计算的间隔动作
set	添加一个 **直接设置目标属性** 的瞬时动作
delay	添加一个 **延迟时间** 的瞬时动作
call	添加一个 **调用回调** 的瞬时动作
target	添加一个 **直接设置缓动目标** 的瞬时动作
union	**将上下文的缓动动作打包成一个**
then	**插入一个 Tween 到缓动队列中**
repeat	**执行几次**（此前为重复几次，请及时适配）
repeatForever	**一直重复执行**
sequence	**添加一个顺序执行的缓动**
parallel	**添加一个同时进行的缓动**

接口	功能说明
start	启动缓动
stop	停止缓动
clone	克隆缓动
show	启用节点链上的渲染，缓动目标需要为 Node
hide	禁用节点链上的渲染，缓动目标需要为 Node
removeSelf	将节点移出场景树，缓动目标需要为 Node

静态接口说明

这些方法为 `Tween` 的静态方法，调用方式示例：

```
Tween.stopAll()
Tween.stopAllByTag(0);
Tween.stopAllByTarget(this.node);
```

接口	功能说明
stopAll	停止所有缓动 该接口会移除底层所有已注册的缓动动画 **注意**：该方法会影响所有对象
stopAllByTag	停止所有指定标签的缓动 该接口将移除通过 **tag** 方法指定的所有缓动 可通过指定第二个参数 `target?: object` 来指定是否仅移除该对象上带有某个标签的缓动
stopAllByTarget	停止所有指定对象的缓动

工具函数说明

接口	功能说明
tween	这是一个工具函数，帮助实例化 `Tween` 类 **注意**：该方法并非 `Tween` 类的成员，开发者也可自行调用 `new Tween<T>(target:T)` 的方式实例化缓动。

示例

这里以一个的 `to` 缓动动画作为示例演示缓动的用法：

```
let tweenDuration : number = 1.0;                                   // 缓动的时长
tween(this.node.position).to( tweenDuration, new Vec3(0, 10, 0),    // 这里以node的位置信息坐标缓动的目标
    {                                                               // ITweenOption 的接口实现：
    onUpdate : (target:Vec3, ratio:number)=>{                       // onUpdate 接受当前缓动的进度
        this.node.position = target;                                // 将缓动系统计算出的结果赋予 node 的位置
    }
}).start();                                                         // 调用 start 方法，开启缓动
```

更多示例可查看 缓动示例

一些限制

为了降低更新 `Node Transform` 信息的频率，`Node` 内部维护了一个 `dirty` 状态，只有在调用了可能会改变 `Node Transform` 信息的接口，才会将 `dirty` 置为需要更新的状态。

但目前的接口存在一定的限制，例如：通过 `this.node.position` 获取到的 `position` 是一个通用的 `Vec3`。

当执行 `this.node.position.x = 1` 这段代码的时候，只执行了 `position` 的 `getter`，并没有执行 `position` 的 `setter`。由于 `dirty` 并没有更新，便会导致渲染时使用的节点的 `Transform` 信息没有更新。

目前，我们也不支持这样的调用，而是鼓励使用 `setPosition` 或 `position` 的 `setter`，如下所示：

```
let _pos = new Vec3(0, 1, 0);
this.node.position = _pos;      // 这里将通过 position 的 setter
this.node.setPosition(_pos);    // 这里将通过接口 setPosition
```

缓动函数

缓动函数

引擎实现了一系列不同类型的缓动函数，通过这些缓动函数，可以实现不同的实时动画效果。这些缓动函数主要用于 `Tween.to` 和 `Tween.by` 这两个接口中。

内置缓动函数

目前引擎提供的缓动函数如下所示：

```
export type TweenEasing =
'linear'    | 'smooth'     | 'fade'         | 'constant'     |
'quadIn'    | 'quadOut'    | 'quadInOut'    | 'quadOutIn'    |
'cubicIn'   | 'cubicOut'   | 'cubicInOut'   | 'cubicOutIn'   |
'quartIn'   | 'quartOut'   | 'quartInOut'   | 'quartOutIn'   |
'quintIn'   | 'quintOut'   | 'quintInOut'   | 'quintOutIn'   |
'sineIn'    | 'sineOut'    | 'sineInOut'    | 'sineOutIn'    |
'expoIn'    | 'expoOut'    | 'expoInOut'    | 'expoOutIn'    |
'circIn'    | 'circOut'    | 'circInOut'    | 'circOutIn'    |
'elasticIn' | 'elasticOut' | 'elasticInOut' | 'elasticOutIn' |
'backIn'    | 'backOut'    | 'backInOut'    | 'backOutIn'    |
'bounceIn'  | 'bounceOut'  | 'bounceInOut'  | 'bounceOutIn';
```

其缓动效果可参考下图：

缓动示例

缓动示例

本文将主要介绍 Cosos Creator 缓动中常见的一些用法和接口。

构造缓动

通过 `tween` 方法或使用 `new Tween<T>(target: T)` 都可以构造缓动。

> **注意**：‘tween’ 是引擎提供的工具方法，并非 ‘Tween’ 的成员，请注意区分。关于这点可以参考接口说明： 缓动接口。

链式 API

大部分和动作相关的接口都会返回 `this` 或者一个新的 `Tween` 对象，因此可以方便的使用链式调用来进行组合：

```
tween()
    .target(this.node)
    .to(1.0, { position: new Vec3(0, 10, 0) })
    .by(1.0, { position: new Vec3(0, -10, 0) })
```

```
    .delay(1.0)
    .by(1.0, { position: new Vec3(0, -10, 0) })
    .start()
```

to，by 简单示例

这里演示了如何使用一个 to 类型的缓动绑定节点的位置信息并将其位置沿 Y 轴偏移 10 个单位：

```
let tweenDuration : number = 1.0;                                   // 缓动的时长
tween(this.node.position)
    .to( tweenDuration, new Vec3(0, 10, 0), {                       // to 接口表示节点的绝对值
    onUpdate : (target:Vec3, ratio:number)=>{                       // 实现 ITweenOption 的 onUpdate 回调，接受当前缓动的进度
        this.node.position = target;                                // 将缓动系统计算出的结果赋予 node 的位置
    }
}).start();                                                         // 调用 start 方法，开启缓动
```

绑定不同对象

开发中使用 Node 作为绑定目标的情景会更多一些，代码示例如下：

```
let  quat : Quat = new Quat();
Quat.fromEuler(quat, 0, 90, 0);
tween(this.node)
    .to(tweenDuration, {
        position: new Vec3(0, 10, 0),                   // 位置缓动
        scale: new Vec3(1.2, 3, 1),                     // 缩放缓动
        rotation:quat }                                 // 旋转缓动
    )
    .start();                                           // 调用 start 方法，开启缓动
```

实际上缓动可以绑定任意对象，代码示例如下：

```
class BindTarget{
    color : Color
}

let sprite : Sprite = this.node.getComponent(Sprite) ;
let bindTarget : BindTarget = new BindTarget();
bindTarget.color = Color.BLACK;
tween(bindTarget)
    .by( 1.0, { color: Color.RED }, {
        onUpdate(tar:BindTarget){
            sprite.color = tar.color;  // 设置精灵的为 BindTarget 内的颜色
        }
})
.start()
```

常见示例

多个动作

通常来说，一个缓动可由一个或多个 **动作** 组成，Tween 维护了一个由多个 **动作** 组成的数据结构用于管理当前缓动内的所有动作。

下面代码演示了将物体的位置沿 Y 轴移动 10 个单位后，沿 -Y 轴移动 10 个单位。

```
let tweenDuration : number = 1.0;
tween(this.node.position)
.to( tweenDuration, new Vec3(0, 10, 0), {
    onUpdate : (target:Vec3, ratio:number)=>{
        this.node.position = target;
    }
})
.to( tweenDuration, new Vec3(0, -10, 0), {
    onUpdate : (target:Vec3, ratio:number)=>{
        this.node.position = target;
    }
}) // 此时 Tween 内的动作数量为 2
```

多个缓动也可使用 union、squence、parallel 接口来组织。通过提前创建好一些固定的缓动，并使用 union、squence、parallel 来组合他们从而减少代码的编写。

整合多个缓动

union 方法会将当前所有的动作合并为一整个，代码示例如下：

```
let tweenDuration : number = 1.0;
tween(this.node)
    .to(tweenDuration, { position:new Vec3(0, 10, 0) })  // 这里以 node 为缓动的目标
    .to(tweenDuration, { position:new Vec3(0, -10, 0) }) // 此时 Tween 内的动作数量为 2
    .union()                                             // 这里会将上述的两个缓动整合成一个，此时 Tween 内的动作数量为 1
    .start();                                            // 调用 start 方法，开启缓动
```

缓动队列

sequence 会将传入的缓动转化为队列形式并加入到当前的缓动内，代码示例如下：

```
let tweenDuration: number = 1.0;
let t1 = tween(this.node)
    .to(tweenDuration, { position: new Vec3(0, 10, 0) })

let t2 = tween(this.node)
    .to(tweenDuration, { position: new Vec3(0, -10, 0) })

tween(this.node).sequence(t1, t2).start(); // 将 t1 和 t2 两个缓动加入到新的缓动队列内
```

同时执行多个缓动

parallel 会将传入的缓动转化为并行形式并加入到当前的缓动内，代码示例如下：

```
let tweenDuration: number = 1.0;
let t1 = tween(this.node)
    .to(tweenDuration, { position: new Vec3(0, 10, 0) })

let t2 = tween(this.node)
    .to(tweenDuration, { position: new Vec3(0, -10, 0) })

tween(this.node).parallel(t1, t2).start(); // 将 t1 和 t2 转化为并行的缓动并加入当前的缓动
```

插入缓动

then 接口允许传入新的缓动，并将该缓动整合后添加到当前缓动的动作内，代码示例如下：

```
let tweenAfter = tween(this.node)
    .to(1.0, { position: new Vec3(0, -10, 0) })

tween(this.node)
    .by(1.0, { position: new Vec3(0, 10, 0) })
    .then(tweenAfter)
    .start();
```

延迟执行

`delay` 会在 **当前** 的动作 **后** 添加一个延时。

注意在下列代码示例中，`delay` 位置不同会造成完全不同的结果：

- 延迟 1 秒后，开始进行运动，并连续运动两次。

```
let tweenDuration: number = 1.0;
tween(this.node)
    .delay(1.0)
    .to(tweenDuration, { position: new Vec3(0, 10, 0) })
    .to(tweenDuration, { position: new Vec3(0, -10, 0) })
    .start()
```

- 在第一次运动后，会延迟 1 秒再做第二次运动。

```
let tweenDuration: number = 1.0;
tween(this.node)
    .to(tweenDuration, { position: new Vec3(0, 10, 0) })
    .delay(1.0)
    .to(tweenDuration, { position: new Vec3(0, -10, 0) })
    .start()
```

重复执行

接口 `repeat` 可以为缓动添加一个重复次数，若 `embedTween` 参数为空，则会使用当前缓动的最后一个动作作为参数。

这意味着，如果当前缓动由多个缓动组成，则只会重复 **最后一个**，请注意下面的示例：

```
let tweenDuration: number = 1.0;
tween(this.node)
    .to(tweenDuration, { position: new Vec3(0, 10, 0) })
    .by(tweenDuration, { position: new Vec3(0, -10, 0) })
    .repeat(3) // 注意这里会重复 by 这个缓动 3 次
    .start()
```

若第二个参数 `embedTween` 不为空，则会重复嵌入的缓动，代码示例如下：

```
let tweenDuration: number = 1.0;
let embedTween = tween(this.node)
    .by(tweenDuration, { position: new Vec3(0, -10, 0) })

tween(this.node)
    .to(tweenDuration, { position: new Vec3(0, 10, 0) })
    .repeat(3, embedTween)  // 这里会重复 embedTween
    .start()
```

`repeatForever` 接口和 `repeat` 类似，但是会变为永久重复。

节点相关的缓动

节点相关的方法只适用于 `target` 是 `Node` 的情况。

显示和隐藏节点

`show` 和 `hide` 接口可以控制绑定节点的显示和隐藏，下面示例中，节点会被隐藏并在延迟 1 秒后显示。

```
tween(this.node)
    .hide()
    .delay(1.0)
    .show()
    .start();
```

删除节点

该方法会产生一个 **删除节点的动作**，该动作会将传入的节点从场景树内删除。

在下面的示例中，节点会在延迟 1 秒后从场景内删除。

```
tween(this.node)
    .delay(1.0)
    .removeSelf()
    .start()
```

添加回调动作

`call` 接口允许给缓动添加一个回调动作，该接口在处理某些异步逻辑时非常有用，示例如下：

```
tween(this.node)
    .to(1.0, { position: new Vec3(0, 10, 0)})
    // to 动作完成后会调用该方法
    .call( ()=>{
        console.log("call");
    })
    .start()
```

设置目标属性

通过 `set` 可以设置目标的属性。下面的示例会在延迟 1 秒后将节点设置在 [0, 100, 0] 的位置。

```
tween(this.node)
    .delay(1.0)
    .set({ position: new Vec3(0, 100, 0) })
    .start();
```

也可以同时设置多个不同的属性，代码示例如下：

```
tween(this.node)
    // 同时设置节点的位置，缩放和旋转
    .set({ position: new Vec3(0, 100, 0), scale: new Vec3(2, 2, 2), rotation: Quat.IDENTITY } )
    .start();
```

复制缓动

`clone` 方法可将当前的缓动复制到目标参数上，注意在复制时源缓动和目前缓动的绑定对象要类型一致，即 `new Tween<T>(target: T)` 中的 `T` 需要类型一致。代码示例如下：

```
let srcTween = tween(this.node).delay(1.0).by(1.0, { position: new Vec3(0, 10, 0) })
// 将 'srcTween' 复制到名为 Cone 的节点上
srcTween.clone(find("Cone")).start();
```

销毁

自动销毁

当缓动目标为 `Node` 时，将会监听其销毁事件进行缓动的自动销毁，调用 `target` 方法也会自动更新监听。

手动销毁

大部分缓动在最后一个动作完毕后，都会对自身进行销毁，但是对于未能正确销毁的缓动， 如 `repeatForever` 在切换场景后，会一直驻留在内存中。需要手动调用销毁接口进行销毁。

如果要停止并销毁缓动，有下列的方法：

- 成员 `stop` 接口，销毁该缓动，代码示例如下：

```
let t = tween(this.node.position)
.to( 1.0, new Vec3(0, 10, 0), {
    onUpdate : (target:Vec3, ratio:number)=>{
        this.node.position = target;
    }
})
t.stop();
```

- 使用静态接口 `stopAll`、`stopAllByTag` 和 `stopAllByTarget` 销毁所有或特定缓动，代码示例如下：

```
Tween.stopAll() // 销毁所有缓动

Tween.stopAllByTag(0); // 销毁所有以 0 为标签的缓动

Tween.stopAllByTarget(this.node); // 销毁该节点上的所有缓动
```

注意：切换场景时记得停止相应的缓动。

地形系统

地形系统

地形系统以一种高效的方式来展示大自然的山川地貌。开发者可以很方便的使用画刷来雕刻出盆地、山脉、峡谷、平原等地貌。

资源管理

Asset Manager 概述

文：Santy-Wang、Xunyi

在游戏的开发过程中，一般需要使用到大量的图片、音频等资源来丰富整个游戏内容，而大量的资源就会带来管理上的困难。所以 Creator 提供了 **Asset Manager** 资源管理模块来帮助开发者管理其资源的使用，大大提升开发效率和使用体验。

Asset Manager 是 Creator 在 v2.4 新推出的资源管理器，用于替代之前的 `loader`。新的 Asset Manager 资源管理模块具备加载资源、查找资源、销毁资源、缓存资源、Asset Bundle 等功能，相比之前的 `loader` 拥有更好的性能，更易用的 API，以及更强的扩展性。所有函数和方法可通过 `assetManager` 进行访问，所有类型和枚举可通过 `AssetManager` 命名空间进行访问。

注意：为了带来平滑的升级体验，我们会在一段时间内保留对 `loader` 的兼容，但还是建议新项目统一使用 **Asset Manager**。

你可以参考以下文章升级：

- loader 升级 assetManager 指南
- 子包升级 Asset Bundle 指南

加载资源

动态加载资源

除了在编辑场景时，可以将资源应用到对应组件上，Creator 还支持在游戏运行过程中动态加载资源并进行设置。而动态加载资源 Asset Manager 提供了以下两种的方式：

1. 通过将资源放在 resources 目录下，并配合 `resources.load` 等 API 来实现动态加载。
2. 开发者可以自己规划资源制作为 Asset Bundle，再通过 Asset Bundle 的 `load` 系列 API 进行资源的加载。例如：

```
resources.load('images/background/spriteFrame', SpriteFrame, (err, asset) => {
  this.getComponent(Sprite).spriteFrame = asset;
});
```

相关的 API 列表如下：

类型	支持	加载	释放	预加载	获取	查询资源信息
单个资源	Asset Bundle	load	release	preload	get	getInfoWithPath
文件夹	Asset Bundle	loadDir	releaseAsset	preloadDir	N/A	getDirWithPath
场景	Asset Bundle	loadScene	N/A	preloadScene	N/A	getSceneInfo
单个资源	`resources`	load	release	preload	get	getInfoWithPath
文件夹	`resources`	loadDir	releaseAsset	preloadDir	N/A	getDirWithPath
远程	Asset Manager	loadRemote	releaseAsset	N/A	N/A	N/A

相关文档可参考：

- 动态加载资源

所有加载到的资源都会被缓存在 `assetManager` 中。

预加载

为了减少下载的延迟，`assetManager` 和 Asset Bundle 中不但提供了加载资源的接口，每一个加载接口还提供了对应的预加载版本。开发者可在游戏中进行预加载工作，然后在真正需要时完成加载。预加载只会下载必要的资源，不会进行反序列化和初始化工作，所以性能消耗更小，适合在游戏过程中使用。

```
start () {
    resources.preload('images/background/spriteFrame', SpriteFrame);
    setTimeOut(this.loadAsset.bind(this), 10000);
}

loadAsset () {
    resources.load('images/background/spriteFrame', SpriteFrame, (err, asset) => {
        this.getComponent(Sprite).spriteFrame = asset;
    });
}
```

关于预加载的更多内容请参考 预加载与加载。

Asset Bundle

开发者可以将自己的场景、资源、代码划分成多个 Asset Bundle，并在运行时动态加载资源，从而实现资源的模块化，以便在需要时加载对应资源。例如：

```
assetManager.loadBundle('testBundle', function (err, bundle) {
    bundle.load('textures/background', (err, asset) => {
        // ...
    });
});
```

更多关于 Asset Bundle 的介绍请参考 bundle。

释放资源

Asset Manager 提供了更为方便的资源释放机制，在释放资源时开发者只需要关注该资源本身而不再需要关注其依赖资源。引擎会尝试对其依赖资源根据引用数量进行释放，以减少用户管理资源释放的复杂度。例如：

```
resources.load('prefabs/enemy', Prefab, function (err, asset) {
    assetManager.releaseAsset(asset);
});
```

Creator 还提供了引用计数机制来帮助开发者控制资源的引用和释放。例如：

- 当需要持有资源时，请调用 `addRef` 来增加引用，确保该资源不会被其他引用到的地方自动释放。

  ```
  resources.load('textures/armor/texture', Texture2D, function (err, texture) {
      texture.addRef();
      this.texture = texture;
  });
  ```

- 当不再需要持有该资源时，请调用 `decRef` 来减少引用，`decRef` 还将根据引用计数尝试自动释放。

  ```
  this.texture.decRef();
  this.texture = null;
  ```

更多详细内容请参考文档 资源释放。

缓存管理器

在某些平台上，比如微信小游戏，因为存在文件系统，所以可以利用文件系统对一些远程资源进行缓存。此时需要一个缓存管理器来管理所有缓存资源，例如缓存资源、清除缓存资源、修改缓存周期等。从 v2.4 开始，Creator 在所有存在文件系统的平台上都提供了缓存管理器，以便对缓存进行增删改查操作。例如：

```
// 获取某个资源的缓存
assetManager.cacheManager.getCache('http://example.com/bundle1/import/9a/9aswe123-dsqw-12xe-123xqawe12.json');

// 清除某个资源的缓存
assetManager.cacheManager.removeCache('http://example.com/bundle1/import/9a/9aswe123-dsqw-12xe-123xqawe12.json');
```

更多缓存管理器的介绍请参考 缓存管理器。

可选参数

`assetManager` 和 Asset Bundle 的部分接口都额外提供了 `options` 参数，可以极大地增加灵活性以及扩展空间。`options` 中除了可以配置 Creator 内置的参数之外，还可以自定义任意参数，这些参数将提供给下载器、解析器以及加载管线。

```
bundle.loadScene('test', { priority: 3 }, callback);
```

更多关于 `options` 的内容可参考文档 可选参数。

如果不需要配置引擎内置参数或者自定义参数来扩展引擎功能，可以无视它，直接使用更简单的 API 接口，比如 `resources.load`。

加载管线

为了更方便地自定义资源加载流程，Asset Manager 底层使用了名为 **管线与任务**、**下载与解析** 的机制来完成资源的加载工作，极大地增加了灵活性和定制性。如果需要自定义加载管线或自定义管线，可以参考：

- 管线与任务
- 下载与解析

更多参考

- Asset Bundle
- 资源释放
- 下载与解析
- 加载与预加载
- 缓存管理器
- 可选参数
- 管线与任务

---

loader 升级 assetManager 指南

资源管理模块升级指南

> 文：Santy-Wang、Xunyi
>
> 本文将详细介绍 Cocos Creator 3D 的 loader 升级到 assetManager 时的注意事项。v2.4 的资源管理与 v3.0 差别不大，无需升级。

在 Cocos Creator 2.4 以前，获取和加载资源 是通过 `loader` 模块（包括 `loader.load`、`loader.loadRes`、`loader.loadResDir` 等系列 API）来实现的，`loader` 模块主要用于加载资源。但随着 Creator 的不断发展，开发者对于资源管理的需求不断增加，原来的 `loader` 已无法满足大量的资源管理需求，一个新的资源管理模块呼之欲出。

因此，Creator 在 v2.4 推出了全新的资源管理模块 —— **Asset Manager**。相较之前的 `loader`，Asset Manager 不但提供了更好的加载性能，而且支持 Asset Bundle、预加载资源以及更加方便的资源释放管理。同时 Asset Manager 还拥有强大的扩展性，大大提升开发者的开发效率和使用体验，我们建议所有开发者都进行升级。

为了带来平滑的升级体验，我们仍保留了对 `loader` 相关 API 的兼容。除个别项目使用了无法兼容的特殊用法的 API 必须手动升级外，大部分项目都可以照常运行。之后我们会在时机成熟时才逐渐完全移除对 `loader` 的兼容。如果由于项目周期等原因暂时不方便升级，你可以在确保测试通过的情况下继续保留原来的写法。

目前在使用旧的 API 时，引擎会输出警告并提示升级方法。请你根据警告内容和本文的说明对代码进行调整，升级到新的用法。比较抱歉的是，由于底层经过了升级，我们遗留了个别无法兼容的 API，在运行时会输出错误信息。如果你已经决定好要进行升级，那么请仔细阅读以下内容。

- 对 **美术策划** 而言，项目中的所有资源，例如场景、动画、Prefab 都不需要修改，也不需要升级。
- 对 **程序** 而言，影响主要体现在原先代码中使用的 `loader` 的所有 API，都需要改为 `assetManager` 的 API。以下将详细介绍这部分内容。

注意：因为 v2.4 支持 Asset Bundle，项目中的分包功能也需要进行升级，具体内容请参考 分包升级指南。

需要手动升级的情况

- 你在自己的代码中使用了以 `loader` 开头的 API，比如 `loader.loaderRes`、`loader.loadResDir`、`loader.release` 等。
- 你在自己的代码中使用了以 `AssetLibrary` 开头的 API，比如 `AssetLibrary.loadAsset`。
- 你在自己的代码中使用了 `url` 开头的 API，比如 `url.raw`。
- 你在自己的代码中使用了 `Pipeline`，`LoadingItems` 等类型。
- 你在自己的代码中使用了 `macro.DOWNLOAD_MAX_CONCURRENT` 属性。

升级步骤

- **备份好旧项目**
- 在 Dashboard 中使用 Cocos Creator v3.0 打开需要升级的旧项目，Creator 将对有影响的资源重新导入，第一次导入时会稍微多花一点时间，导入完毕后就会打开编辑器主窗口。此时可能会出现较多的报错或警告信息，别担心，请打开代码编辑工具根据报错或警告信息对代码进行升级。

将 `loader` 相关的 API 替换为 `assetManager` 相关的 API

从 v2.4 开始，不建议使用 `loader`，并且在后续的版本中也会逐渐被彻底移除，请使用新的资源管理模块 `assetManager` 进行替换。

加载相关接口的替换

如果你在自己的代码中使用了 `loader.loadRes`、`loader.loadResArray`、`loader.loadResDir`，请使用 `assetManager` 中对应的 API 进行替换。可参考下方的替换方式：

- **loader.loadRes**

  `resources.load` 的参数与 `loader.loadRes` 完全相同。替换方式如下：

  ```
  // 修改前
  loader.loadRes(...);

  // 修改后
  resources.load(...);
  ```

- **loader.loadResArray**

  `assetManager` 为了降低学习成本，将 `loadResArray` 与 `load` 进行了合并。`resources.load` 的第一个参数可支持多个路径，所以可以使用 `resources.load` 进行替换：

  ```
  // 修改前
  loader.loadResArray(...);

  // 修改后
  resources.load(...);
  ```

- **loader.loadResDir**

  `resources.loadDir` 的参数与 `loader.loadResDir` 完全相同：

  ```
  // 修改前
  loader.loadResDir(...);

  // 修改后
  resources.loadDir(...);
  ```

  注意：为了简化接口，`resources.loadDir` 的加载完成回调将 **不再提供** `paths` 的列表。请避免以下的使用方式：

  ```
  loader.loadResDir('images', Texture2D, (err, assets, paths) => console.log(paths));
  ```

  如果你想要查询 `paths` 列表，可以使用以下方式：

  ```
  const infos = resources.getDirWithPath('images', Texture2D);
  let paths = infos.map(function (info) {
      return info.path;
  });
  ```

- **loader.load**

  如果你在自己的代码中使用了 `loader.load` 来加载远程图片或远程音频，为了方便理解，在 `assetManager` 中将有专门的 API 用于此项工作，如下所示：

  - 加载远程图片

    ```
    // 修改前
    loader.load('http://example.com/remote.jpg', (err, texture) => console.log(texture));

    // 修改后
    assetManager.loadRemote('http://example.com/remote.jpg', (err, texture) => console.log(texture));
    ```

  - 加载远程音频

    ```
    // 修改前
    loader.load('http://example.com/remote.mp3', (err, audioClip) => console.log(audioClip));

    // 修改后
    assetManager.loadRemote('http://example.com/remote.mp3', (err, audioClip) => console.log(audioClip));
    ```

  - 加载远程文本

    ```
    // 修改前
    loader.load('http://example.com/equipment.txt', (err, text) => console.log(text));

    // 修改后
    assetManager.loadRemote('http://example.com/equipment.txt', (err, textAsset) => console.log(textAsset.text));
    ```

注意：

1. 如果你在自己的代码中使用了 `loader.downloader.loadSubpackage` 来加载分包，请参考 分包升级指南 进行升级。
2. 为了避免产生不必要的错误，`loader.onProgress` 在 `assetManager` 中没有对应实现。你可以自己实现全局回调机制，但建议将回调传入到每个加载函数中，避免并发加载时互相干扰。

释放相关接口的替换

如果你在自己的代码中使用了 `loader.release`、`loader.releaseAsset`、`loader.releaseRes`、`loader.releaseResDir`，请使用 `assetManager` 中对应的 API 进行替换。可参考下方的替换方式：

- **loader.release**

  `loader.release` 可用 `assetManager.releaseAsset` 替换。

  注意：为了避免开发者关注资源中一些晦涩难懂的属性，`assetManager.releaseAsset` **不再接受** 数组、资源 UUID、资源 URL 进行释放，仅能通过资源本身进行释放。

  ```
  // 修改前
  loader.release(texture);
  // 修改后
  assetManager.releaseAsset(texture);

  // 修改前
  loader.release([texture1, texture2, texture3]);
  // 修改后
  [texture1, texture2, texture3].forEach(t => assetManager.releaseAsset(t));

  // 修改前
  const uuid = texture._uuid;
  loader.release(uuid);
  // 修改后
  assetManager.releaseAsset(texture);

  // 修改前
  const url = texture.url;
  loader.release(url);
  // 修改后
  assetManager.releaseAsset(texture);
  ```

  注意：为了增加易用性，在 `assetManager` 中释放资源的依赖资源将 **不再需要** 手动获取资源的依赖项，在 `assetManager.releaseAsset` 内部将会尝试自动去释放相关依赖资源，例如：

  ```
  // 修改前
  const assets = loader.getDependsRecursively(texture);
  loader.release(assets);

  // 修改后
  assetManager.releaseAsset(texture);
  ```

- **loader.releaseAsset**

  `loader.releaseAsset` 可直接使用 `assetManager.releaseAsset` 替换：

```
// 修改前
loader.releaseAsset(texture);

// 修改后
assetManager.releaseAsset(texture);
```

- **loader.releaseRes**

  `loader.releaseRes` 可直接使用 `resources.release` 替换：

```
// 修改前
loader.releaseRes('images/a', Texture2D);

// 修改后
resources.release('images/a', Texture2D);
```

- **loader.releaseAll**

  `loader.releaseAll` 可直接使用 `assetManager.releaseAll` 替换：

```
// 修改前
loader.releaseAll();

// 修改后
assetManager.releaseAll();
```

注意：

1. 出于安全考虑，`loader.releaseResDir` 在 `assetManager` 中没有对应实现，请使用 `assetManager.releaseAsset` 或 `resources.release` 进行单个资源释放。
2. 因为 `assetManager.releaseAsset` 会自动释放依赖资源，所以你不需要再显式调用 `loader.getDependsRecursively`。如果需要查找资源的相关依赖，请参考 `assetManager.dependUtil` 中相关的 API。
3. 出于安全考虑，`assetManager` 仅支持在场景中设置的自动释放，其他的已移除。`assetManager` 中没有实现 `loader.setAutoRelease`、`loader.setAutoReleaseRecursively`、`loader.isAutoRelease` 这几个 API，建议你使用全新的基于引用计数的自动释放机制，详细请参考 资源释放。

扩展相关接口的替换

- **Pipeline**

  如果你的代码中有使用 `loader.insertPipe`、`loader.insertPipeAfter`、`loader.appendPipe`、`loader.addDownloadHandlers`、`loader.addLoadHandlers` 系列 API 对 `loader` 的加载流程做过扩展，或者直接使用了 `loader.assetLoader`、`loader.md5Pipe`、`loader.downloader`、`loader.loader`、`loader.subPackPipe` 中的方法，请使用 `assetManager` 中对应的 API 进行替换。

  因为 `assetManager` 是更通用的模块，不再继承自 `Pipeline`，所以 `assetManager` 不再实现 `loader.insertPipe`、`loader.insertPipeAfter`、`loader.appendPipe`。具体的替换方式如下：

```
// 修改前
const pipe1 = {
  id: 'pipe1',
  handle: (item, done) => {
    let result = doSomething(item.uuid);
    done(null, result);
  }
};

const pipe2 = {
  id: 'pipe2',
  handle: (item, done) => {
    let result = doSomething(item.content);
    done(null, result);
  }
};

loader.insertPipe(pipe1, 1);
loader.appendPipe(pipe2);

// 修改后
function pipe1 (task, done) {
  let output = [];
  for (let i = 0; i < task.input.length; i++) {
    let item = task.input[i];
    item.content = doSomething(item.uuid);
    output.push(item);
  }

  task.output = output;
  done(null);
}

function pipe2 (task, done) {
  let output = [];
  for (let i = 0; i < task.input.length; i++) {
    let item = task.input[i];
    item.content = doSomething(item.content);
    output.push(item);
  }

  task.output = output;
  done(null);
}

assetManager.pipeline.insert(pipe1, 1);
assetManager.pipeline.append(pipe2);
```

  注意：

  1. `assetManager` **不再继承** 自 `Pipeline`，而是 `assetManager` 下拥有的多个 `Pipeline` 实例。详情请参考 管线与任务。
  2. 为了易用性，Pipe 的定义不再需要定义一个拥有 `handle` 方法和 `id` 的对象，只需要一个方法即可。详情请参考 管线与任务。
  3. 为了简化逻辑、提高性能，Pipe 中处理的内容不再是 `item`，而是 `task` 对象。详情请参考 管线与任务。
  4. 为了降低学习成本，`Pipeline` 中不再支持 `insertPipeAfter` 形式的 API，请使用 `insert` 插入指定的位置。

- **addDownloadHandlers、addLoadHandlers**

  出于模块化考虑，`assetManager` 中没有实现 `addDownloadHandlers`、`addLoadHandlers`，请参考以下方式替换：

```
// 修改前
const customHandler = (item, cb) => {
    let result = doSomething(item.url);
    cb(null, result);
};

loader.addDownloadHandlers({png: customHandler});

// 修改后
const customHandler = (url, options, cb) => {
    let result = doSomething(url);
    cb(null, result);
};

assetManager.downloader.register('.png', customHandler);
```

  或者：

```
// 修改前
const customHandler = (item, cb) => {
    let result = doSomething(item.content);
    cb(null, result);
};

loader.addLoadHandlers({png: customHandler});

// 修改后
const customHandler = (file, options, cb) => {
    let result = doSomething(file);
    cb(null, result);
};

assetManager.parser.register('.png', customHandler);
```

注意：

1. 因为 **下载模块** 与 **解析模块** 都是依靠 **扩展名** 来匹配对应的处理方式，所以调用 `register` 时，传入的第一个参数需要以 . 开头。
2. 出于模块化的考虑，自定义的处理方法将不再传入一个 `item` 对象，而是直接传入与其相关的信息。`downloader` 的自定义处理方法传入的是 **待下载的 URL**，`parser` 传入的则是 **待解析的文件**。具体的内容请参考 下载与解析。
3. 新的拓展机制提供了一个额外的 `options` 参数，可以极大地增加灵活性。但如果你不需要配置引擎内置参数或者自定义参数，可以无视它。具体内容请参考文档 可选参数。

- **downloader，loader，md5Pipe，subPackPipe**

  `loader.downloader` 可由 `assetManager.downloader` 代替，`loader.loader` 可由 `assetManager.parser` 代替。但其中的接口没有完全继承，具体内容请参考文档 下载与解析 或者 API 文档 assetManager.downloader 和 assetManager.parser。

  注意：出于对性能、模块化和易读性的考虑，`loader.assetLoader`、`loader.md5Pipe`、`loader.subPackPipe` 已经被合并到 `assetManager.transformPipeline` 中，你应该避免使用这三个模块中的任何方法与属性。关于 `assetManager.transformPipeline` 的具体内容可参考 管线与任务。

其他更新

`url` 与 `AssetLibrary` 在 **v2.4** 中已经被移除，请避免使用 `url` 与 `AssetLibrary` 中的任何方法和属性。

`Pipeline` 可由 `AssetManager.Pipeline` 进行替换，请参考以下方式进行替换：

```
// 修改前
const pipe1 = {
    id: 'pipe1',
    handle: function (item, cb) {
        let result = doSomething(item);
        cb(null, result);
    }
}

const pipeline = new Pipeline([pipe1]);

// 修改后
function pipe1 (task, cb) {
    task.output = doSomething(task.input);
    cb(null);
}

const pipeline = new AssetManager.Pipeline('test', [pipe1]);
```

注意：`LoadingItem` 在 `assetManager` 中已经不支持，请避免使用这个类型。

为了支持更多加载策略，`macro.DOWNLOAD_MAX_CONCURRENT` 已经从 `macro` 中移除，你可以用以下方式替换：

```
// 修改前
macro.DOWNLOAD_MAX_CONCURRENT = 10;

// 修改后
assetManager.downloader.maxConcurrency = 10;
```

或者

```
// 修改前
macro.DOWNLOAD_MAX_CONCURRENT = 10;

// 修改后（设置预设值）
assetManager.presets['default'].maxConcurrency = 10;
```

具体内容可参考 下载与解析。

子包升级 Asset Bundle 指南

资源分包升级指南

文：Santy-Wang、Xunyi

本文将详细介绍 Cocos Creator 3D 的小游戏子包升级到 Asset Bundle 的注意事项。v2.4 的资源分包与 v3.0 差别不大，无需升级。

在 v2.4 之前，分包加载 功能仅支持各类小游戏平台，如微信小游戏、OPPO 小游戏等。但随着 Creator 的发展，开发者对分包的需求不断增加，例如多平台支持，原有的分包加载已经远远不能满足了。所以，Creator 从 v2.4 开始正式支持功能更为完整的 **Asset Bundle**。

- 对 **美术策划** 而言，项目中的所有资源，例如场景、动画、Prefab 都不需要修改，也不用升级。
- 对 **程序** 而言，影响主要体现在原先代码中使用的 `loader.downloader.loadSubpackage` 需要改为 Asset Manager 中的 `assetManager.loadBundle`。以下将详细介绍这部分内容。

注意：如果你在旧项目中使用了分包功能，也就是在 **属性检查器** 中勾选了 **配置为子包** 选项，那么当项目升级到 v2.4 之后，将自动转变为一个普通文件夹。你可以参考这里进行 Asset Bundle 的配置：

配置 Asset Bundle

需要手动升级的情况

你在自己的代码中使用了 `loader.downloader.loadSubpackage` 来加载分包。

升级步骤

- **备份好旧项目**
- 在 Dashboard 中使用 Cocos Creator v3.0 打开需要升级分包的旧项目，Creator 会对有影响的资源重新导入。第一次导入时会稍微多花一点时间，导入完毕后就会打开编辑器主窗口。然后使用代码编辑器将所有 `loader.downloader.loadSubpackage` 替换为 `assetManager.loadBundle`。

```
// 修改前
loader.downloader.loadSubpackage('sub1', (err) => {
  loader.loadRes('sub1/sprite-frames/background', SpriteFrame);
});

// 修改后
assetManager.loadBundle('sub1', (err, bundle) => {
  // 传入该资源相对 Asset Bundle 根目录的相对路径
```

```
    bundle.load('sprite-frames/background/spriteFrame', SpriteFrame);
});
```

注意：加载 Asset Bundle 中的资源需要使用 Asset Bundle 相关的 API，具体请查看 API 文档 Asset Bundle。

Asset Bundle 的使用方式

关于 Asset Bundle 的具体使用方式，请参考文档 Asset Bundle。

资源加载

加载资源

动态加载 resources

通常我们会把项目中需要动态加载的资源放在 `resources` 目录下，配合 `resources.load` 等接口动态加载。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
resources.load("test_assets/prefab", Prefab, (err, prefab) => {
    const newNode = instantiate(prefab);
    this.node.addChild(newNode);
});

// 加载 AnimationClip
resources.load("test_assets/anim", AnimationClip, (err, clip) => {s
    this.node.getComponent(Animation).addClip(clip, "anim");
});
```

- 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` 文件夹需要在 **assets 根目录** 下手动创建。如下所示：

Asset Bundle

Asset Bundle 介绍

文：Santy-Wang、Xunyi

从 v2.4 开始，Creator 正式支持 Asset Bundle 功能。Asset Bundle 作为资源模块化工具，允许开发者按照项目需求将贴图、脚本、场景等资源划分在多个 Asset Bundle 中，然后在游戏运行过程中，按照需求去加载不同的 Asset Bundle，以减少启动时需要加载的资源数量，从而减少首次下载和加载游戏时所需的时间。

Asset Bundle 可以按需求放置在不同地方，比如可以放在远程服务器、本地、或者小游戏平台的分包中。

从 v3.8 开始，Bundle 的配置方案被转移到 **项目设置** -> **Bundle 配置** 分页内。您可以点击 **属性检查器** 上的 **编辑按钮** 或者通过 **项目** 菜单打开 **项目设置** 对 Bundle 进行配置。

内置 Asset Bundle

项目中除了自定义的 Asset Bundle 外，Creator 还有内置的 3 个 Asset Bundle。与其他自定义 Asset Bundle 一样，内置 Asset Bundle 也可以根据不同平台进行配置。

资源释放

资源释放

文：Santy-Wang、Xunyi

Asset Manager 中提供了资源释放模块，用于管理资源的释放。

在资源加载完成后，会被临时缓存到 `assetManager` 中，以便下次复用。但是这也会造成内存和显存的持续增长，所以有些资源如果不需要用到，可以通过 **自动释放** 或者 **手动释放** 的方式进行释放。释放资源将会销毁资源的所有内部属性，比如渲染层的相关数据，并移出缓存，从而释放内存和显存（对纹理而言）。

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 assetManager 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，assetManager 会自动使用缓存中的资源。

下载与解析

下载与解析

文：Santy-Wang、Xunyi

Asset Manager 底层使用了多条加载管线来加载和解析资源，每条管线中都使用了 `downloader` 和 `parser` 模块，也就是下载器和解析器。开发者可以通过 `assetManager.downloader` 和 `assetManager.parser` 来访问。

下载器

下载器是一个全局单例，包括 **下载重试**、**下载优先级排序** 和 **下载并发数限制** 等功能。

下载重试

下载器如果下载资源失败，会自动重试下载，开发者可以通过 `maxRetryCount` 和 `retryInterval` 属性来设置重试下载的相关参数。

- `maxRetryCount` 属性用于设置重试下载的最大次数，默认 3 次。若不需要重试下载，可设置为 0，则下载失败时会立即返回错误。

  ```
  assetManager.downloader.maxRetryCount = 0;
  ```

- `retryInterval` 属性用于设置重试下载的间隔时间，默认 2000 ms。若设置为 4000 ms，则下载失败时会先等待 4000 ms，然后再重新下载。

  ```
  assetManager.downloader.retryInterval = 4000;
  ```

下载优先级

Creator 开放了四个下载优先级，下载器将会按照优先级 **从大到小** 的顺序来下载资源。

资源	优先级	说明
脚本或 Asset Bundle	2	优先级最高
场景资源	1	包括场景中的所有资源，确保场景能够快速加载
开发者手动加载的资源	0	
预加载资源	-1	优先级最低，因为预加载更多是提前加载资源，时间要求相对较为宽松

开发者也可以通过可选参数 `priority` 传入一个优先级来覆盖默认设置，从而控制加载顺序。详情可参考下方的“通过可选参数设置”部分。

设置下载并发数

开发者可以通过 `maxConcurrency` 和 `maxRequestsPerFrame` 来设置下载器的最大下载并发数等限制。

- `maxConcurrency` 用于设置下载的最大并发连接数，若当前连接数超过限制，将会进入等待队列。

  ```
  assetManager.downloader.maxConcurrency = 10;
  ```

- `maxRequestsPerFrame` 用于设置每帧发起的最大请求数，从而均摊发起请求的 CPU 开销，避免单帧过于卡顿。如果此帧发起的连接数已经达到上限，将延迟到下一帧发起请求。

  ```
  assetManager.downloader.maxRequestsPerFrame = 6;
  ```

另外，`downloader` 中使用了一个 `jsb.Downloader` 类的实例，用于在 **原生平台** 从服务器上下载资源。`jsb.Downloader` 与 Web 的 XMLHttpRequest 类似。目前 `jsb.Downloader` 类的实例的下载并发数限制默认为 **32**，超时时长默认为 **30s**，如果需要修改默认值，可以在 `main.js` 中修改：

```
// main.js
assetManager.init({
    bundleVers: settings.bundleVers,
    remoteBundles: settings.remoteBundles,
    server: settings.server,
    jsbDownloaderMaxTasks: 32, // 最大并发数
    jsbDownloaderTimeout: 60 // 超时时长
});
```

解析器

解析器用于将文件解析为引擎可识别的资源，开发者可以通过 `assetManager.parser` 来访问。

通过可选参数设置

在下载器和解析器中的设置都是全局设置，若开发者需要单独设置某个资源，可以通过 **可选参数** 传入专有设置来覆盖全局设置，例如：

```
assetManager.loadAny({'path': 'test'}, {priority: 2, maxRetryCount: 1, maxConcurrency: 10}, callback);
```

具体内容可参考文档 可选参数。

预设

Creator 预先对正常加载、预加载、场景加载、Asset Bundle 加载、远程资源加载、脚本加载这 6 种加载情况的下载/解析参数做了预设，其中预加载因为性能考虑，所以限制较大，最大并发数更小。如下所示：

```
{
    'default': {
        priority: 0,
    },

    'preload': {
        maxConcurrency: 2,
        maxRequestsPerFrame: 2,
        priority: -1,
    },

    'scene': {
        maxConcurrency: 8,
        maxRequestsPerFrame: 8,
        priority: 1,
    },

    'bundle': {
        maxConcurrency: 8,
        maxRequestsPerFrame: 8,
        priority: 2,
    },

    'remote': {
        maxRetryCount: 4
    },

    'script': {
        priority: 2
    }
}
```

开发者可以通过 `assetManager.presets` 对每种预设进行修改，使用时需要传入预设的名称来访问对应的参数项。

```
// 修改预加载的预设优先级为 1
let preset = assetManager.presets.preload;
preset.priority = 1;
```

也可以增加自定义预设，并通过可选参数 `preset` 传入。

```
// 自定义预设，并通过可选参数 preset 传入
assetManager.presets.mypreset = {maxConcurrency: 10, maxRequestsPerFrame: 6};
assetManager.loadAny({'path': 'test'}, {preset: 'mypreset'}, callback);
```

注意：通过可选参数、预设以及下载器/解析器本身，均能设置下载和解析过程中的相关参数（例如下载并发数、重试次数等）。当使用多种方式设置了同一个参数时，引擎会按照 **可选参数 > 预设 > 下载器/解析器** 的先后顺序进行选取使用。也就是说，如果引擎在可选参数中找不到相关设置时，会去预设中查找，如果预设中也找不到时，才会去下载器/解析器中查找。

自定义处理方法

下载器和解析器都拥有一张注册表，在使用 `downloader` 或 `parser` 时，下载器和解析器会根据传入的后缀名称去注册表中查找对应的下载方式和解析方式，并将参数传入对应的处理方式之中。当开发者需要修改目前格式的处理方式，或者在项目中增加一个自定义格式时，可以通过注册自定义的处理方式来实现扩展引擎。下载器与解析器都提供了 `register` 接口用于注册处理方法，使用方式如下：

```
assetManager.downloader.register('.myformat', function (url, options, callback) {
    // 下载对应资源
    ......
});

assetManager.parser.register('.myformat', function (file, options, callback) {
    // 解析下载完成的文件
    ......
});
```

自定义的处理方法需要接收三个参数：

- 第一个参数为处理对象，在下载器中是 url，在解析器中是文件。
- 第二个参数是可选参数，可选参数可以在调用加载接口时指定。
- 第三个参数是完成回调，当注册完成处理方法时，需要调用该函数，并传入错误信息或结果。

在注册了处理方法之后，当下载器/解析器遇到带相同扩展名的请求时，会使用对应的处理方式，这些自定义的处理方式将供全局所有加载管线使用。

```
assetManager.loadAny({'url': 'http://example.com/myAsset.myformat'}, callback);
```

需要注意的是，处理方法可以接收传入的可选参数，开发者可以利用可选参数实现自定义扩展，具体内容可查看文档 可选参数。

加载与预加载

加载与预加载

文：Santy-Wang，Xunyi

为了尽可能缩短下载时间，很多游戏都会使用预加载。Asset Manager 中的大部分加载接口包括 `load`、`loadDir`、`loadScene` 都有其对应的预加载版本。加载接口与预加载接口所用的参数是完全一样的，两者的区别在于：

1. 预加载只会下载资源，不会对资源进行解析和初始化操作。
2. 预加载在加载过程中会受到更多限制，例如最大下载并发数会更小。
3. 预加载的下载优先级更低，当多个资源在等待下载时，预加载的资源会放在最后下载。
4. 因为预加载没有做任何解析操作，所以当所有的预加载完成时，不会返回任何可用资源。

相比 Creator v2.4 以前的版本，以上优化手段充分降低了预加载的性能损耗，确保了游戏体验顺畅。开发者可以充分利用游戏过程中的网络带宽缩短后续资源的加载时间。

因为预加载没有去解析资源，所以需要在预加载完成后配合加载接口进行资源的解析和初始化，来完成资源加载。例如：

```
resources.preload('images/background/spriteFrame', SpriteFrame);

// wait for while
resources.load('images/background/spriteFrame', SpriteFrame, function (err, spriteFrame) {
    spriteFrame.addRef();
    self.getComponent(Sprite).spriteFrame = spriteFrame;
});
```

注意：加载不需要等到预加载完成后再调用，开发者可以在任何时候进行加载。正常加载接口会直接复用预加载过程中已经下载好的内容，缩短加载时间。

缓存管理器

缓存管理器

文：Santy-Wang、Xunyi

在 Web 平台，资源下载完成之后，缓存是由浏览器进行管理，而不是引擎。
而在某些非 Web 平台，比如微信小游戏，这类平台具备文件系统，可以利用文件系统对一些远程资源进行缓存，但并没有实现资源的缓存机制。此时需要由引擎实现一套缓存机制用于管理从网络上下载下来的资源，包括缓存资源、清除缓存资源、查询缓存资源等功能。

从 v2.4 开始，Creator 在所有存在文件系统的平台上都提供了缓存管理器，以便对缓存进行增删改查操作，开发者可以通过 `assetManager.cacheManager` 进行访问。

资源的下载、缓存及版本管理

引擎下载资源的逻辑如下：

1. 判断资源是否在游戏包内，如果在则直接使用；
2. 如果不在则查询资源是否在本地缓存中，如果在则直接使用；
3. 如果不在则查询资源是否在临时目录中，如果在则直接使用（原生平台没有临时目录，跳过该步骤）；
4. 如果不在就从远程服务器下载资源，资源下载到临时目录后直接使用（原生平台是将资源下载到缓存目录）；
5. 后台缓慢地将临时目录中的资源保存到本地缓存目录中，以便再次访问时使用（原生平台跳过该步骤）；
6. 当缓存空间占满后资源会保存失败，此时会使用 LRU 算法删除比较久远的资源（原生平台的缓存空间没有大小限制，跳过该步骤，开发者可以手动调用清理）。

对小游戏平台来说，一旦缓存空间占满，所有需要下载的资源都无法保存，只能使用下载保存在临时目录中的资源。而当退出小游戏时，所有的临时目录都会被清理，再次运行游戏时，这些资源又会被再次下载，如此循环往复。

> **注意**：缓存空间超出限制导致文件保存失败的问题不会在微信小游戏的 **微信开发者工具** 上出现，因为微信开发者工具没有限制缓存大小，所以测试缓存时需要在真实的微信环境中进行测试。

当开启引擎的 **md5Cache** 功能后，文件的 URL 会随着文件内容的改变而改变，这样当游戏发布新版本后，旧版本的资源在缓存中就自然失效了，只能从服务器请求新的资源，也就达到了版本控制的效果。

上传资源到远程服务器

当包体过大时，需要将资源上传到远程服务器，请将资源所在的 Asset Bundle 配置为远程包。接下来我们以微信小游戏为例，来看一下具体的操作步骤：

1. 合理分配资源，将需要模块化管理的资源文件夹（例如 `resources` 文件夹）配置为 Asset Bundle，并勾选 **配置为远程包**，具体可参考文档 配置 Asset Bundle。

可选参数

可选参数

文：Santy-Wang，Xunyi

为了增加灵活性和可扩展性，Asset Manager 中大部分的加载接口包括 `assetManager.loadAny` 和 `assetManager.preloadAny` 都提供了 `options` 参数。`options` 除了可以配置 Creator 的内置参数，还可以自定义任意参数用于扩展引擎功能。如果开发者不需要配置引擎内置参数或者扩展引擎功能，可以无视它，直接使用更简单的 API 接口，比如 `resources.load`。

目前 `options` 中引擎已使用的参数包括：

`uuid`, `url`, `path`, `dir`, `scene`, `type`, `priority`, `preset`, `audioLoadMode`, `ext`, `bundle`, `onFileProgress`, `maxConcurrency`, `maxRequestsPerFrame`, `maxRetryCount`, `version`, `xhrResponseType`, `xhrWithCredentials`, `xhrMimeType`, `xhrTimeout`, `xhrHeader`, `reloadAsset`, `cacheAsset`, `cacheEnabled`

注意：请 **不要** 使用以上字段作为自定义参数的名称，避免与引擎功能发生冲突。

扩展引擎

开发者可以通过在 管线 和 自定义处理方法 中使用可选参数来扩展引擎的加载功能。

```
// 扩展管线
assetManager.pipeline.insert(function (task, done) {
    const input = task.input;
    for (let i = 0; i < input.length; i++) {
        if (input[i].options.myParam === 'important') {
            console.log(input[i].url);
        }
    }
    task.output = task.input;
    done();
}, 1);

assetManager.loadAny({'path': 'images/background'}, {'myParam': 'important'}, callback);

// 注册处理方法
assetManager.downloader.register('.myformat', function (url, options, callback) {
    // 下载对应资源
    const img = new Image();
    if (options.isCrossOrigin) {
        img.crossOrigin = 'anonymous';
```

```
    }

    img.onload = function () {
        callback(null, img);
    };

    img.onerror = function () {
        callback(new Error('download failed'), null);
    };

    img.src = url;

});

assetManager.parser.register('.myformat', function (file, options, callback) {
    // 解析下载完成的文件
    callback(null, file);
});

assetManager.loadAny({'url': 'http://example.com/myAsset.myformat'}, {isCrossOrigin: true}, callback);
```

通过可选参数，再结合管线和自定义处理方法，引擎可以获得极大的扩展度。Asset Bundle 可以看做是使用可选参数扩展的第一个实例。

管线与任务

管线与任务

文：Santy-Wang，Xunyi

本文适用于对加载流程有定制需求的进阶开发者

为了更方便地修改或者扩展引擎资源加载流程，Asset Manager 底层使用了名为 **管线与任务** 和 **下载与解析** 的机制对资源进行加载，本篇内容主要介绍 **管线与任务**。

虽然在 v2.4 之前的 `loader` 已经开始使用管线的概念来进行资源加载，但是在 Asset Manager 中，我们对管线进行了重构，使得逻辑更加清晰，也更容易扩展。开发者可以扩展现有管线，也可以使用引擎提供的类 `AssetManager.Pipeline` 来自定义管线。

管线

管线 可以理解为一系列过程的串联组合，当一个请求经过管线时，会被管线的各个阶段依次进行处理，最后输出处理后的结果。如下图所示：

资源管理注意事项 - meta 文件

资源管理注意事项 --- meta 文件

本文全文转载自 微信公众号：奎特尔星球，转载前已获得作者授权 作者：ShawnZhang

Cocos Creator 会为 assets 目录下的每一个文件和目录生成一个同名的 meta 文件，相信大家一定不会太陌生。理解 Creator 生成 meta 文件的作用和机理，能帮助您和您的团队解决在多人开发时常会遇到的资源冲突、文件丢失、组件属性丢失等问题。那 meta 文件是做什么用的呢？下面我们来了解一下。

多语言（L10N）

多语言本地化（L10N）

多语言本地化（以下简称 L10N 或本地化）是 Cocos Creator 3.6 推出的功能，该功能整合了第三方译文服务商的翻译服务，同时运行将文本、音频和图片等资源的本地化功能整合到引擎内，并支持一键发布到不同语言。

L10N 是单词 Localization 的首字母以及尾字母的缩写，10 代表 Localization 中间有 10 个字母。

L10N 总览

在引擎顶部菜单中选择 **面板** -> **本地化编辑器** 中即可打开本地化编辑器面板。

首次启动时，用户需手动启用 L10N 功能：

译文服务商

译文服务商

译文服务商是第三方软件商，引擎通过抹平他们之间 API 的差异将其整合到一起。通常开发者需要注册服务商的账号并开启相应的 API 才可以启动自动翻译的功能。

若您没有对应的开发者账号也无需担心，L10N 支持手动翻译。

收集并统计

收集并统计

收集功能会将项目内文本、Typescript 脚本、场景资源、预制体、视频、引擎和图片等文件搜集起来，并允许开发者进行本地化配置。

语言编译

语言编译

L10nLabel 组件

L10nLabel 组件

L10nLabel 是可以根据内容进行定制化翻译的组件。配合文本组件使用，可对文本组件的内容进行翻译。

示例

脚本使用示例

导入

导入示例：

```
import l10n from 'db://localization-editor/core/L10nManager'
```

- 描述：l10n以api的方式提供了在代码中翻译文本的能力

动态切换语言

代码示例如下：

```
l10n.changeLanguage('zh-Hans-CN')
```

参数类型请参考 查看 BCP47 Language Tag 以获得更多信息。

> **注意**:在调用此方法后，会自动重启游戏，请务必做好数据持久化工作。

- 接口定义： `t(key: L10nKey, options?: StandardOption): L10nValue`

根据键获取 L10N 的值

```
console.log(l10n.t('this_is_apple'))
// 这是一个苹果
```

此处可以获取到以 `this_is_apple` 为键的当前语言的值。

查询某个键是否存在

代码示例如下：

```
console.log(l10n.exists('test_key'))
```

获取当前的语言

代码示例如下：

```
console.log(l10n.currentLanguage)
// 'zh-Hans-CN'
```

返回当前语言的 BCP47 Language Tag。

获取所有可用语言

代码示例如下：

```
console.log(l10n.languages)
// ['zh-Hans-CN', 'en-US']
```

返回当前可用的语言的 BCP47 Language Tag 数组。

获取语言的方向

绝大多数语言都遵循从左到右的阅读习惯，但某些语言却例外比如阿拉伯语，此方法可以得知所传入语言的 `TextInfoDirection`

```
console.log(l10n.direction('ar'))
// 'rtl'
```

更多详细的 API 描述请参考 Localization Editor Api

API

Localization Editor Api

快速开始

核心功能 `l10n`

l10n 提供了核心翻译功能以及 ICU 功能，同时也提供的切换语言的功能。

我们会将切换后的目标语言存储于 `localStorage` 中，同时也会自动重启项目运行时，并在下次启动时读取 `localStorage` 配置以完成整个语言切换流程。

> **注意**：因此我们希望用户在切换语言之前务必处理好数据持久化工作。

导入 `l10n` 模块

localization-editor 所提供的所有 API 都将从 db://localization-editor/l10n 进行具名导入，导入示例如下：

```
import { l10n } from 'db://localization-editor/l10n'
```

使用翻译api

```
// 任意 component 组件代码中

// l10n 是 localization 的核心功能
import { l10n } from 'db://localization-editor/l10n'
import { _decorator, Label, Component } from 'cc';

@ccclass('SomeComponent')
class SomeComponent extends Component {
    // ......
    someMethod() {
        // 将返回 this_is_an_apple 所对应文案
        const text = l10n.t("this_is_an_apple")
    }
    // ......
}
```

API详细说明

- 类:L10nManager

- 接口:ResourceList
- 接口:ResourceBundle
- 接口:ResourceData
- 接口:ResourceItem

- 接口:FallbackLanguageObjectList
- 接口:L10nOptions
- 接口:StandardOption

- 枚举:L10nListenEvent

- 别名:L10nKey
- 别名:L10nValue
- 别名:TextInfoDirection
- 别名:FallbackLanguage

L10nManager

导入示例：

```
import { L10nManager } from 'db://localization-editor/l10n'
```

描述：

通常我们不建议您自行使用或构造该类型。

而我们提供了l10n作为全局单例以使用翻译功能。

索引

构造函数

- L10nManager **private**

全局变量

l10n

定义:`const l10n: L10nManager`

静态属性

LOCAL_STORAGE_LANGUAGE_KEY

定义:`static LOCAL_STORAGE_LANGUAGE_KEY: string`

描述:当调用changeLanguage切换游戏语言时，将使用`localStorage`存储所切换的目标语言标记，并且使用LOCAL_STORAGE_LANGUAGE_KEY作为`localStorage`的key

备注：

默认值 localization-editor/language

实例方法

config

定义:`config(options: L10nOptions): void`

描述:用于配置l10n的某些设置，探索更多选项可以查看L10nOptions

用例:

```
l10n.config({
    // 用于在默认语言没有找到相应翻译时，以该值进行补充显示
    fallbackLanguage: 'zh-Hans-CN',
    // 如果不喜欢LOCAL_STORAGE_LANGUAGE_KEY的默认值，可以在此修改，但是需要确保在changeLanguage之前
    localStorageLanguageKey: 'localization-editor/langauge'
})
```

---

changeLanguage

定义:`changeLanguage(language: Intl.BCP47LanguageTag): void`

描述:用于动态切换语言，请查看BCP47 Language Tag以获得更多信息

用例:

```
l10n.changeLanguage('zh-Hans-CN')
```

> **注意**:在调用此方法后，会自动重启游戏，请务必做好数据持久化工作。

---

t

定义:`t(key: L10nKey, options?: StandardOption): L10nValue`

描述:根据传入的L10nKey，返回当前语言数据中所对应的L10nValue，探索更多选项可以查看StandardOption

用例:

```
console.log(l10n.t('this_is_apple'))
// 这是一个苹果
```

> 注意:语言数据需要配合 Localization Editor 扩展在编译后生成。

exists

定义:`exists(key: L10nKey): boolean`

描述:返回是否存在key

用例:

```
console.log(l10n.exists('test_key'))
```

---

currentLanguage

定义:`get currentLanguage(): Intl.BCP47LanguageTag`

描述:返回当前语言的BCP47 Language Tag

用例:

```
console.log(l10n.currentLanguage)
// 'zh-Hans-CN'
```

---

languages

定义:`get languages(): readonly Intl.BCP47LanguageTag[]`

描述:返回当前可用语言的BCP47 Language Tag数组，可利用该方法作为切换语言下拉框的数据源

用例:

```
console.log(l10n.languages)
// ['zh-Hans-CN', 'en-US']
```

---

direction

定义:`direction(language?: Intl.BCP47LanguageTag): TextInfoDirection`

描述:绝大多数语言都尊崇从左到右的阅读习惯，但某些语言却例外比如阿拉伯语，此方法可以得知所传入语言的TextInfoDirection

用例:

```
console.log(l10n.direction('ar'))
// 'rtl'
```

---

on

定义:`on(event: L10nListenEvent, callback: (...args: any[]) => void)`

描述:用于注册l10n的L10nListenEvent事件回调，比如`languageChanged`

用例:

```
l10n.on(L10nListenEvent.languageChanged, (...args: any[]) => {
    //在切换语言后的一些操作，某些数据可以放在这里持久化，之后便会重启整个游戏场景
})
```

---

off

定义:`off(event: L10nListenEvent, callback: (...args: any[]) => void)`

描述:用于反注册l10n的L10nListenEvent事件回调

请务必使on与off成对出现，确保正确的销毁无用数据

别名

别名	原类型
L10nKey	string
L10nValue	string
TextInfoDirection	'ltr' / 'rtl'
FallbackLanguage	string / readonly string[] / FallbackLanguageObjectList / ((language: Intl.BCP47LanguageTag) => string / readonly string[] / FallbackLanguageObjectList

---

接口

L10nOptions

函数/变量名	类型	可选
fallbackLanguage	false / FallbackLanguage	是
localStorageLanguageKey	string	是
beforeTranslate	(key: L10nKey) => L10nValue	是
afterTranslate	(key: L10nKey) => L10nValue	是
returnNull	boolean	是
returnEmptyString	boolean	是

ResourceList

函数/变量名	类型	可选
defaultLanguage	Intl.BCP47LanguageTag	是
fallbackLanguage	Intl.BCP47LanguageTag	是
languages	Intl.BCP47LanguageTag[]	否

ResourceBundle

函数/变量名	类型	可选
[language: Intl.BCP47LanguageTag]	ResourceData	否

ResourceData

函数/变量名	类型	可选
[namespace: string]	ResourceItem	否

ResourceItem

函数/变量名	类型	可选
[key: string]	any	否

FallbackLanguageObjectList

函数/变量名	类型	可选
[language: string]	readonly string[]	否

StandardOption

函数/变量名	类型	可选
count	number	是
defaultValue	L10nValue	是
language	Intl.BCP47LanguageTag	是
fallbackLanguage	FallbackLanguage	是

枚举

L10nListenEvent

函数/变量名	类型
languageChanged	languageChanged
onMissingKey	missingKey

EXCEL 导入示例

EXCEL 导入示例

在开始本篇之前，请开发者准备好支持 L10N 的引擎（v3.6以及以上），并创建一个空的项目，我们将通过该示例演示如何在项目中使用 EXCEL 文件作为多语言的数据文件。

> 其他文件类型如 PO、CSV，其使用流程与 EXCEL 类似。

准备工作

- 首先打开 Dashboard 创建任意空项目

XR

Cocos CreatorXR 介绍

Cocos CreatorXR 是基于 Cocos Creator 和 Cocos Engine 打造的一款 XR 内容创作工具。底层通过支持 OpenXR 标准协议来抹平不同 XR 设备之间的差异，可以一站式对创作内容进行开发并发布到不同的 XR 设备中而无需去适配不同设备的 I/O 项；中层封装了一系列不同功能的 XR 中间件来提供 XR 内容创作支持，并支持用户自定义扩展组件内容；上层基于 Cocos Creator 面板扩展出多种形式的 XR 功能菜单和组件样式，为用户提供更为便利的内容创作界面。

版本历史

版本历史

V1.2.1

新增：

- 移动端和Web端AR降级功能，支持更多设备。
- Pico设备支持FFR、透视功能。
- 透视纹理新增静态纹理类型。

V1.2.0

新增：

- iOS 支持基础模式光照估计，华为安卓平台支持基础和 HDR 模式光照估计。
- 支持将 XR 应用构建发布至 WebXR 平台，会话支持 inline、immersive-vr、immersive-ar 三种模式。
- 屏幕手势交互器新增行为控制功能，可自由选择控制交互物的手势行为。
- 屏幕手势交互器组件放置功能新增放置位置的计算方式，可以基于屏幕交互器固定距离来放置内容。
- 新增合成层（Composition Layer）功能，支持 Overlay、Underlay 两种渲染方式。
- 新增透视（Pass Through）功能，支持 OpenXR 标准接口开启一体机视频透视功能，配合合成层可调整透视视频流的渲染方式（Overlay/Underlay）。
- 新增 XR Web Preview 功能，支持在一体机端浏览 web 内容。
- XR 视频播放器支持解析播放 3D 视频资源。
- 支持空间音频，允许从各个方向渲染音频，模拟声音在物理世界中的表现，增加沉浸感。
- 新增静态注视点渲染（FFR）调节，支持使用 OpenXR 标准接口开启 FFR 并调节渲染等级。
- Snapdragon Spaces SDK 支持更新到 0.11.1，并新增 RGB Camera 、Meshing 特性的支持。

修复：

- 屏幕手势交互的效果优化。
- 华为 VR Glass 的手柄震动时间 0 无效问题。
- 转换 AR Camera 报错问题修复。
- Android Target API 高版本兼容性问题。

v1.1.1

新增：

- 适配 Cocos Creator 3.7.2。

v1.1.0

新增：

- 新增 AR 应用开发模块，可发布 AR 应用至 AREngine、ARCore、ARKit 和 Qualcomm Spaces 平台，提供自动化行为编辑组件来支持快速创建和体验 AR 内容，支持开启设备追踪（AR Camera）/平面追踪/图像追踪/锚点/Mesh 等 AR 特性。
- 新增非缓冲式手柄控制器震动反馈，在 cc.InteractorEvent 的 Haptic Event 中选择想要开启震动的事件类型并调整震动参数。
- XRUI 一键转换功能，传统 2D UI 可以一键转化为的带有空间属性的 XR UI。
- 凝视交互器，根据头戴设备的注视中心位置进行交互行为。
- XR 视频播放器，针对 XR 设备优化了视频渲染管线并支持切换展示窗口、180度、360度多风格的视频。可以满足用户在 3D 场景中浏览全景视频或动态材质的需要。
- XR 内容预览新增无线串流方式，在 Web 浏览器中预览 XR 项目并同步所有来自 XR 设备的信号，无需打包应用至设备即可快速完整地体验所有 XR 项目内容。
- 屏幕手势交互器，使用屏幕手势操作 AR 虚拟对象。
- 新增 Rokid Air 设备支持手机作为 3DOF 空鼠的功能。

修复：

- Huawei VR Glass（6Dof套装）对 6Dof 输入的完整支持；
- Huawei VR Glass配套的 6Dof 手柄在触摸摇杆时也产生随机信号的处理问题；

v1.0.1

- 支持发布 XR 应用至 Rokid Air、HuaweiVR、Meta Quest/Quest2、Pico Neo3/Pico4、Monado 五类设备。
- 提供设备映射、交互、虚拟移动、XR UI 模块化组件支持 XR 内容创作。
- 支持使用 Web 浏览器预览 XR 内容，可用键鼠操作模拟头显和手柄控制器设备。

架构

架构

插件架构是指插件的系统结构。作为 Cocos CreatorXR 的用户，您可以通过了解插件架构的组成元素、它们的职责以及它们如何相互组合联系，来快速完整的构建 XR 应用。

内置资源与预制体

内置资源与预制体

在 Cocos Creator 扩展管理器中 **开启XR扩展** 之后就可以允许在编辑器中使用传统创建对象的方式创建 XR 对象。

在 层级管理器 右键选择 **创建 -> XR**，右侧会出现当前可以创建的所有 XR 预制体。选择想要实例化生成的对象即可在场景中创建出来。

XR 组件

XR组件

Cocos CreatorXR 通过组件的组合封装为实体赋能，实体根据其不同特性又被不同的功能系统所管理。所以编辑器中所有 XR 相关的功能底层都是由封装好的特殊 XR 组件驱动的。

Cocos CreatorXR 的功能组件主要由 5 部分构成：

- 设备映射
- 交互组件
- 交互限制组件
- 虚拟移动组件
- XR UI

开启了 xr-plugin 扩展之后，想要给场景中的对象添加 XR 相关的功能组件可以在 **属性检查器** 中点击 **添加组件** 按钮，在出现的组件列表中找到 **XR** 分类，选择 XR 分类下的想要添加的 XR 组件类别再找到类别下的对应组件即可。

设备映射组件

此类组件主要用以同步现实世界中物理设备和虚拟世界中的代理节点之间的 I/O 信息。确保在用户在 XR 设备的使用和虚拟世界中的反馈一致。

主要包括以下组件：

TrackingOrigin

追踪原点代理组件。

预览

预览

为了方便开发者在项目开发过程中实时调试，快速验证一些传统的功能逻辑来提高开发效率，Cocos CreatorXR 基于 Cocos Creator 的 Web Preview 功能开发了适用于 XR 项目的预览功能。

操作说明

在 xr-plugin 的资源库中找到 XR Simulator，将其拖拽至场景中。

XR 视频播放器

XR 视频播放器

XR 头戴设备相较于传统的显示器拥有更为多样化的视频展示方式，结合设备自身的多轴向定位特性和双屏渲染画面，可以满足用户在 3D 场景中浏览全景视频或动态材质的需要。Cocos CreatorXR v1.1.0 提供了通用化的 XR 视频播放器，针对 XR 设备优化了视频渲染管线并支持切换展示窗口、180 度、360 度多风格的视频。同时，播放器还提供了交互功能辅助您进行播放控制，您只需要添加或替换视频资源即可完成简易的视频播放功能的内容开发，简化创作步骤，降低开发门槛。

创建视频播放器，请在层级管理器右键 **创建 -> XR -> XR Video Player**。

XR 网页浏览器

XR 网页浏览器

在XR中，网页浏览器可以让用户在虚拟现实环境中访问和浏览网页。用户可以设备控制器与网页界面进行交互，如选择链接、滚动页面等等操作，以增强用户的沉浸感和体验。

XR 网页浏览器功能

属性	描述
Content	指定用于渲染网页内容的MeshRenderer对象。
Url	网页链接。

使用 XR 网页浏览器

层级管理器右键 **创建 > XR > XR Webview**。

默认创建一个带有 cc.MeshRenderer 组件的节点作为子节点 Webview Content。

XR 空间音频

XR 空间音频

XR 空间音频是虚拟现实领域中的一项重要技术，它可以模拟现实世界中的音频环境，让用户在虚拟现实环境中获得更加真实的听觉体验。基于头戴显示器的声音跟踪技术可以通过跟踪用户的头部运动来模拟现实世界中的音频环境。当用户在虚拟现实环境中移动头部时，系统可以根据用户的头部运动和位置来调整音频的位置和方向，从而模拟出现实世界中的音频环境。

XR 空间音频功能

属性	描述
Clip	引用需要挂载的音频文件。
Loop	是否循环播放音频。
Play On Wake	是否在启动项目时自动播放音频。
Volume	音量大小。
Distance Rolloff Model	音量根据距离效应进行衰减的模型。

XR 空间音频使用

选择想要添加音频的节点对象，在属性管理器中添加组件：**XR > Extra > XRSpatialAudioSource**

XR 合成层

XR 合成层

在 XR 应用开发中，合成层（Composition Layer）是一种常用的技术，通常应用在混合现实场景，将虚拟现实场景和真实世界场景进行混合，合成层将根据用户设定的 layer 深度将不同的 layer 分别渲染到不同的图层中，然后将这些图层进行合成，形成一个完整的XR场景。同时，通过调整图层的透明度和深度，能达到虚拟对象与真实世界对象完美融合的效果。Composition Layer 技术可以实现高质量的XR渲染效果，为 XR 应用的开发和体验提供了很大的帮助和支持。

合成层功能

属性	说明
Layer Setting	合成层效果设置。
--Type	合成层的类型： Overlay：将纹理呈现在 Eye Buffer 前面。 Underlay：将纹理呈现在 Eye Buffer 后面。
--Shape	提供两种种形状的合成层： Quad：具有四个顶点的平面纹理，通常用来显示场景中的文本或信息。 Cylinder具有柱面弧度的圆柱形纹理，通常用于显示曲面 UI 界面。
--Redius	选择 Cylinder 时出现此项，设置曲面半径。
--CentralAngle	选择 Cylinder 时出现此项，设置中心角大小。
--Depth	定义合成层在场景中的顺序。数值越小，越靠近 Eye Buffer。
TextureSetting	材质效果设置。
--Texture Type	设置纹理类型： Static：若需要渲染静态内容，需使用静态纹理。 Dynamic：若要将动态内容渲染至该合成层，即为合成层每帧更新纹理，则需要使用动态纹理。例如使用普通摄像机生成的 RenderTexture 图像。
--Static Texture	绑定静态纹理资源。
--Camera	用于绑定合成层要获取动态纹理的相机。
--Width	设置相机 RT 的宽。
--Height	设置相机 RT 的高。

注意：

1. 合成层功能对接 OpenXR 的核心 API 扩展，适用于所有对接 OpenXR 标准的设备。
2. 必须将摄像机放置在圆柱内切球内。如果摄像机接近内切球表面，合成层将显示异常。

使用合成层

目前合成层功能可以渲染动态纹理，主要应用于渲染 Camera 所采集的画面。

以下案例实现一个 Overlay 表现的镜子对象，可以反射 XR 角色的动作表现。

配置步骤

先在场景中创建完整的 XR 代理节点用于设备追踪。并绑定简单的头显/手柄模型。

透视

透视

透视（Pass Through）的实现方式是通过 XR 设备的摄像头捕捉现实世界的场景，再将其传输到显示器上，让用户可以看到现实世界的场景。在虚拟现实环境中，用户无法感知现实世界的环境，容易发生碰撞或者其他安全问题，使用 Pass Through 技术可以让用户感知现实世界的环境，不但可以更加安全地在虚拟现实环境中行动，还可以增强虚拟现实的沉浸感。

透视功能

XR 扩展提供了专门用于渲染透视图像的层，同时使用合成层技术控制透视图像与虚拟场景的融合显示关系。

属性	说明
Placement	指定 Pass Through Layer 的合成方式。
Depth	设置深度来指定 Pass Through在Composition Layer 的排序。
Opacity	设置 Pass Through 图像的不透明度。

注意：透视功能对接 OpenXR 的非核心扩展API。当前版本只对接了 Meta Quest 系列设备。

开启透视

调整 **XR HMD** 节点 Camera 组件的 Clear Flags 为 SOLID_COLOR，Clear Color 的不透明度调为 0。

AR

AR

Cocos CreatorXR 扩展允许您创建跨平台的增强现实（AR）应用程序。您可以给场景中的节点添加相应的 AR 功能组件来选择要启用哪些 AR 特性而无需关心平台的 AR SDK 之间的差异，当您在 AR 设备上构建和运行应用程序时，Cocos Creator 引擎底层的 AR 模块（AR Module）会调用设备原生的 AR SDK 以启用 AR 功能，因此您可以只进行一次开发并发布到多个 AR 平台。

版本要求

编辑器请使用 Cocos Creator v3.7.1 或更高的版本。

插件请使用 xr-plugin v1.1.0 或更高的版本。

AR 相机

AR 相机

和头戴显示器一样，场景中为了能够抽象表式移动端设备带有 AR 能力的摄像机，XR 插件使用 AR Camera 组件封装一系列属性来映射物理设备的摄像头 AR 功能。

AR Manager

AR Manager

Cocos CreatorXR 的 AR 模块提供了一个全局管理器，用于收集当前项目使用到的 AR 特性并进行管理，特性管理器中每个特性的属性都是全局属性，调整参数会修改设备相关的或者项目全局的功能。cc.ARManager 默认挂载在 XR Agent 节点上，当您创建 AR 自动化行为节点时，ARManager 会收集此节点到其对应的特性列表中，方便后续管理和维护所有特性节点。

当前版本针对以支持的 AR 特性提供了对应的全局功能属性：

平面追踪特性

当您在场景中创建一个或多个 Plane Tracking 节点，AR Manager 中的 Configuration 会新增 Plane Feature 属性。您可以调整特性下的各项参数，或定位到对应的特性节点。

AR 自动化行为编辑

AR 自动化行为编辑

AR 场景中，虚拟物体与现实实体间总是存在未知的依赖关系，如果能够清晰方便的描述现实实体的条件特征并针对此种条件执行匹配的行为，可以很大程度上简化开发者处理复杂的 AR 功能特性而专注于编写项目核心逻辑。Cocos CreatorXR 提供了 AR 自动化行为编辑组件，将常用的物理特征和逻辑行为抽象成元素供开发者自由搭配，图形化的操作极大程度降低了 AR 应用的开发成本和开发门槛。

每种特性的自动化行为编辑组件都有其独特的特征库和行为库，以下介绍了当前版本支持自动化行为的所有 AR 特性：

平面追踪

在编辑器的层级管理器中右键 **创建 -> XR -> Plane Tracking**，创建平面代理节点，此节点可用于描述物理世界中的某一个平面实体。

AR 交互

AR 交互

AR 交互主要由 cc.ScreenTouchInteractor 组件驱动，该组件将触摸事件转换为点击、拖拽和捏合等手势，交互器将这些手势传递给可以交互的虚拟交互物，完成手势对应触发的行为。

手势交互

AR 手势交互器组件将屏幕触摸转换为手势。Cocos Creator 的输入系统将手势信号传递给交互物，然后交互物响应手势事件发生变换行为。交互物能发生交互行为的前提是必须绑定 **cc.Selectable** 组件，关于此组件的属性描述详见交互组件 Selectable。

想要使用 **屏幕手势交互器**，在层级管理器中右键创建 **XR -> Screen Touch Interactor** 。

AR 降级

AR 降级

为了解决市面上部分安卓手机不支持AR功能以及WebXR浏览器和设备覆盖率不高的问题，Cocos CreatorXR提供了一系列解决方案：

1. 移动设备原生端：引擎内部通过集成设备Sensor数据、相机视频流数据和 **OpenCV** 的图像识别库，来降级原生AR SDK的平面和图像追踪能力的表现效果。
2. Web端：引擎内部通过集成浏览器API和 **MindAR** 图像追踪算法库，来降级WebXR的平面和图像追踪能力的表现效果。

这些方案允许AR应用在不带有AR SDK或不支持WebXR的设备上使用，但是表现效果会有所降低。

想了解设备是否支持AR，移动设备原生AR SDK支持情况的相关内容请参考各平台 AR SDK 的设备支持情况；WebXR支持情况的相关内容请参考 WebXR 的浏览器支持情况。

降级功能说明

原生端

平台

Android

系统

OS版本在Android8.0及以上

平面追踪特性降级

1. 开启特性降级功能：选择一个Plane Tracking节点，开启cc.ARPlaneTracking下的Enable auto fallback。

快速部署指南

Cocos CreatorXR 介绍

Cocos CreatorXR 是基于 Cocos Creator 和 Cocos Engine 打造的一款 XR 内容创作工具。底层通过支持 OpenXR 标准协议来抹平不同 XR 设备之间的差异，可以一站式对创作内容进行开发并发布到不同的 XR 设备中而无需去适配不同设备的 I/O 项；中层封装了一系列不同功能的 XR 中间件来提供 XR 内容创作支持，并支持用户自定义扩展组件内容；上层基于 Cocos Creator 面板扩展出多种形式的 XR 功能菜单和组件样式，为用户提供更为便利的内容创作界面。

VR 项目创建

VR 项目创建

Cocos CreatorXR 支持用户使用以下几种方式快速创建 VR 项目。

> **注意**：创建 XR 项目时务必保证编辑器版本 >= 3.6.1。

使用模板新建 VR 项目：在 Cocos Dashboard 中新建项目时，选择 v3.6.1 及以上的编辑器（若需要体验完整功能，引擎请选择v3.7.1及以上的版本），选择 Empty(VR) 模板创建。

VR 项目构建与发布

VR 项目构建与发布

当项目开发完成后，需要将项目打包到对应的平台上，在菜单栏中选中 **项目** -> **构建发布** 打开构建发布面板，通过 **发布平台** 属性下的下拉框中选择目标平台：

AR 项目创建

AR 项目创建

参照以下步骤完成对项目的 AR 相关特性配置。

以下提供三种方法，可以任选一种来配置扩展或直接打开内置 AR 项目。

方法一：将 xr-plugin 应用到项目

在 Cocos Store 中搜索 xr-plugin，获取扩展并安装，具体安装说明请参考 说明。

安装完毕后将扩展添加至对应工程。

这种方式适合为存量 3D 项目做 XR 模式迁移。

AR 项目构建与发布

AR 项目构建与发布

完成 AR 应用的项目设置并完成项目开发之后，即可打包 AR 应用。点击 **菜单栏 -> 项目 -> 构建发布**。

ARCore、AREngine

针对于安卓和华为平台的手机发布 AR 应用，新建构建任务，平台选择 **安卓**。

WebXR 项目配置

WebXR 项目配置

WebXR 项目的创建和普通XR项目创建流程保持一致。

若需要创建沉浸式虚拟现实（VR）体验的工程，可以参考 VR 项目创建。

若需要创建沉浸式增强现实（AR）体验的工程，可以参考 AR 项目创建。

部署完毕之后需要为 **XR Agent** 节点添加 **cc.WebXRSessionController** 组件，组件位置为：**XR > Device > WebXRSessionController**。

根据需要选择默认 Session Mode：

- IMMERSIVE_AR：Session 将独占访问沉浸式 XR 设备，渲染的内容将与现实世界的环境混合在一起。
- IMMERSIVE_VR：Session 对场景的渲染不会被被覆盖或融入现实环境。
- INLINE：3D 内容输出在标准 HTML 文档的元素上下文中内联显示，而不会占据整个视觉空间。inline session 既可以在单目渲染的设备呈现，也可以在双目立体渲染的设备中呈现；而且不关心设备是否可进行位姿追踪。inline session 不需要特殊的设备，在任何提供 WebXR API支持的 用户代理 上都可以使用。

WebXR 项目构建与发布

WebXR 项目构建与发布

完成 WebXR 应用的 项目设置 并完成项目开发之后，即可打包 WebXR 应用。点击 **菜单栏 -> 项目 -> 构建发布**。

配置 WebXR 构建属性

构建平台选择 **Web 移动端**。

原生开发

原生开发

Cocos Creator 支持发布为移动端原生应用、移动端网页、移动端小游戏、桌面端原生应用、桌面端网页等。

本章将包含一些在原生端应用开发中可能会遇到的议题。

内容

原生平台二次开发指南

原生平台二次开发指南

如果你需要为增加第三方 SDK 库，或者增删 C++，OC，JAVA 代码文件，以下内容可以帮你更好地理解。

原生项目目录

当点击 **构建** 按钮后，会生成三个原生平台相关的文件夹。

公共目录

公共目录位置：`native/engine/common`

此目录用于存放共公内容，如引擎库配置，以及一些所有平台都会用上的第三方库。

> 这个目录下的代码多数是由 C/C++ 编写。

原生平台目录

平台目录名字规则：native/engine/当前构建的平台名称

这个目录用于存放对应平台相关的信息，比如：

- native/engine/android
- native/engine/ios
- native/engine/win64
- native/engine/mac

> win64 用于 windows, 目前已不再支持 win32, 仅支持 win64 应用程序发布。

项目目录

项目目录名字规则：`build/当前构建的平台名称`

这个目录，包含的是最终生成的原生工程，用于编译、调试和发布。如：

- build/android
- build/ios
- build/windows
- build/mac

每一次构建时，引擎会将公共目录和原生平台目录，以及 Cocos Creator 项目中的资源、脚本等结合在一起，生成项目目录。

项目目录中的代码和相关配置引用原生平台目录下的文件，在 IDE 中改动对应的部分，平台目录下的文件也会做对就修改。

例外：`native/engine/ios/info.plist` 与 `native/engine/mac/info.plist` 文件由于CMake的机制，使用的是复制方式。 如果要对 `info.plist` 进行修改，则需要注意。

项目目录包含下内容：

- `assets`：`data` 目录的软链，用于兼容各平台
- `data`：Cocos Creator 项目中的资源和脚本生成的内容
- `proj`：存放当前构建的原生平台工程，可用于对应平台的 IDE（如 Xcode，Android Studio 等） 执行编译、调试和发布。
- `cocos.compile.config.json`：本次构建的采用的构建选项配置

原生项目定制开发

因为项目需要，有时候我们需要修改、增删原生平台相关的源代码，或者引入第三方SDK，修改项目配置等。这些工作，我们称为项目定制开发。

下面我们就分类说明，不同情况下的需求，如何操作。

修改引擎代码

请参考 引擎定制工作流程。

修改项目代码

如果需要修改项目相关的代码，只需要找到对应文件进行修改即可，修改完即可编译，不需要额外配置。

增删项目代码文件

增删代码文件，将会涉及到编译配置，而不同的语言和平台有所差异，下面将分类说明。

增删 C++ 文件

如果需要增加和删除项目相关的 `C++` 文件，需要做对应的 `CMakeList.txt` 修改。

如果增删的是 `native/engine/common/` 目录下的代码，则需要修改 `native/engine/common/CMakeLists.txt`

```
list(APPEND CC_COMMON_SOURCES
    ${CMAKE_CURRENT_LIST_DIR}/Classes/Game.h
    ${CMAKE_CURRENT_LIST_DIR}/Classes/Game.cpp
)
```

如上面代码所示，找到对应位置，添加或者删除自己的源码即可。

如果增删的是平台相关的 `C++` 代码文件，则需要修改 `native/engine/平台名称/CMakeLists.txt`。参考下面代码：

```
include(${CC_PROJECT_DIR}/../common/CMakeLists.txt)

//在这个位置添加自己的 C++ 文件
list(APPEND CC_PROJ_SOURCES
    ${CMAKE_CURRENT_LIST_DIR}/MyTest2.hpp
    ${CMAKE_CURRENT_LIST_DIR}/MyTest2.cpp
)
```

增删 OC 文件

Cocos Creator 生成的 iOS/macOS 原生工程中，Objective-C 文件的管理方式，与 C++ 完全一致，参考上面的内容即可。

增删 Java 文件

Java 语言本身是基于路径的包管理机制，增删 `JAVA` 文件不需要做特殊处理。

引入第三方 C++/OC 库

如果引入的库是由 C++/OC 编写而成，则根据情况将 SDK 放入 `native/engine/common/` 或者 `native/engine/平台名称/` 目录下，并修改对应目录下的 `CMakeList.txt`。

OC 库只能放在平台目录，不能放在 `native/engine/common/`，否则会导致在其他原生平台出现编译错误。

大部分 C++ SDK 也提供了自己的 `CMakeList.txt`，直接通过 `include` 的方式集成就行。

关于 CMake 的配置，可以参考项目中已有的 `CMakeList.txt` 进行修改。更多关于 CMake 的使用详情，可参考 CMake 使用简介。

引入 Jar 库

如果引入的库是 Android 平台特有的库，直接放到对应的 `native/engine/android/` 目录，配置 `native/android/build.gradle` 即可。

脚本与原生通信

新写的原生方法，或者新引入的原生 SDK，如果想要导出到脚本层使用，可以采用以下几种方案。

使用 JsbBridge

如果需要调用一些简单，非高频的函数，可以使用 `JsbBridge` 机制进行调用。

- 使用 JsbBridge 实现 JavaScript 与 Java 通信
- 使用 JsbBridge 实现 JavaScript 与 Objective-C 通信

JSB 自动绑定

对于需要高频调用，或者批量导出 API 到脚本层的接口，建议使用 JSB 自动绑定 机制实现脚本与原生交互。

基于语言反射机制

基于 Java 和 OC 语言反射机制的通信，也可以很方便实现脚本与原生的交互，但由于 iOS 的审核规则越来越严格，iOS 上使用反射机制有审核失败的风险。

- 基于反射机制实现 JavaScript 与 Android 系统原生通信
- 基于反射机制实现 JavaScript 与 iOS/macOS 系统原生通信

源码版本管理

如果你的团队使用源码版本管理软件进行多人协同工作，`native` 目录需要全部加入到源码版本管理。

所有的项目定制化工作都应该尽量放到 `native` 目录，这样 `build` 目录就可以随时被删除，它不需要加入到源代码版本管理。

对于一些特殊的项目需求，无法在 `native` 目录下完成，则需要改动 `build` 目录下的内容，此时应该根据需求将对应的文件夹加入管理。

Java 原生反射机制

基于反射机制实现 JavaScript 与 Android 系统原生通信

JavaScript 调用 Java 静态方法

使用 Cocos 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。它的定义如下：

```
import { native } from 'cc';
var o = native.reflection.callStaticMethod(className, methodName, methodSignature, parameters...)
```

- className：类名
- methodName：方法名
- methodSignature：方法签名
- parameters：参数列表

接下来，我们以 `com.cocos.game` 包下面的 `Test` 类为例，来具体说明。

```
// package "com.cocos.game";

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;
    }
}
```

className

`className` 需要包含包名信息，如果要调用上面的 Test 类中的静态方法，`className` 应该为 "com/cocos/game/Test"。

> **注意**：这里必须是斜线 `/`，而不是在 Java 代码中的 `.`。

methodName

`methodName` 就是方法本来的名字，例如要调用 `sum` 方法的话，methodName 传入的就是 "sum"。

methodSignature

由于 Java 支持函数重载功能，方法签名用于告诉反射系统对应的参数类型和返回值类型，以确定唯一的方法。

它的格式为：**（参数类型）返回值类型**。

目前 Cocos Creator 中支持的 Java 类型签名有以下 4 种：

Java 类型	签名
int	I
float	F
boolean	Z
String	Ljava/lang/String;

> **注意**：String 类型的签名为 `Ljava/lang/String;`，不要漏掉了最后的 `;`。

下面是一些案例

- `()V` 表示没有参数，没有返回值
- `(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
- `(ILjava/lang/String;F)Ljava/lang/String;` 表示参数类型为一个 int，一个 String 和一个 float，返回值类型为 String 的方法

parameters

传递的参数与签名匹配即可，支持 number、bool 和 string。

使用示例

接下来我们看几个 Test 类中的静态方法的调用示例：

```
if(sys.os == sys.OS.ANDROID && sys.isNative){
    // 调用 hello 方法
    native.reflection.callStaticMethod("com/cocos/game/Test", "hello", "(Ljava/lang/String;)V", "this is a message from JavaScript");

    // 调用第一个 sum 方法
    var result = native.reflection.callStaticMethod("com/cocos/game/Test", "sum", "(II)I", 3, 7);
    log(result); // 10

    // 调用第二个 sum 方法
    var result = native.reflection.callStaticMethod("com/cocos/game/Test", "sum", "(I)I", 3);
    log(result); // 5
}
```

`sys.isNative` 用于判断是否为原生平台，`sys.os` 用于判断当前运行系统。由于各平台通信机制不同，建议先判断再处理。

运行后，可以在 **控制台** 中看到相应的输出结果。

Java 调用 JavaScript

除了 JavaScript 调用 Java，引擎也提供了 Java 调用 JavaScript 的机制。

通过引擎提供的 `CocosJavascriptJavaBridge.evalString` 方法可以执行 JavaScript 代码。需要注意的是，由于 JavaScript 相关代码会在 GL 线程中执行，我们需要利用 `CocosHelper.runOnGameThread` 来确保线程是正确的。

接下来，我们给刚才的 Alert 对话框增加一个按钮，并在它的响应函数中执行一段 JavaScript 代码。

```
alertDialog.setButton("OK", new DialogInterface.OnClickListener() {
    public void onClick(DialogInterface dialog, int which) {
        // 一定要在 GL 线程中执行
        CocosHelper.runOnGameThread(new Runnable() {
            @Override
            public void run() {
                CocosJavascriptJavaBridge.evalString("cc.log(\"Javascript Java bridge!\")");
            }
        });
    }
});
```

调用全局函数

我们可以在脚本中通过如下代码新增一个全局函数：

```
window.callByNative = function(){
  //to do
}
```

`window` 是 Cocos 引擎脚本环境中的全局对象，如果要让一个变量、函数、对象或者类全局可见，需要将它作为 `window` 的属性。可以使用 `window.变量名` 或者 `变量名` 进行访问。

然后像下面这样调用:

```
CocosHelper.runOnGameThread(new Runnable() {
    @Override
    public void run() {
        CocosJavascriptJavaBridge.evalString("window.callByNative()");
    }
});
```

或者：

```
CocosHelper.runOnGameThread(new Runnable() {
    @Override
    public void run() {
        CocosJavascriptJavaBridge.evalString("callByNative()");
    }
});
```

调用类的静态函数

假如在 TypeScript 脚本中有一个类具有如下静态函数：

```
export class NativeAPI{
  public static callByNative(){
    //to do
  }
}
//将 NativeAPI 注册为全局类，否则无法在 Java 中被调用
window.NativeAPI = NativeAPI;
```

我们可以像这样调用：

```
CocosHelper.runOnGameThread(new Runnable() {
    @Override
    public void run() {
        CocosJavascriptJavaBridge.evalString("NativeAPI.callByNative()");
    }
});
```

调用单例函数

如果脚本代码中，有实现可以全局访问的单例对象

```
export class NativeAPIMgr{
  private static _inst:NativeAPIMgr;

  public static get inst():NativeAPIMgr{
    if(!this._inst){
      this._inst = new NativeAPIMgr();
    }
    return this._inst;
  }

  public static callByNative(){
    //to do
  }
}

//将 NativeAPIMgr 注册为全局类，否则无法在 Java 中被调用
window.NativeAPIMgr = NativeAPIMgr;
```

我们可以像下面这样调用：

```
CocosHelper.runOnGameThread(new Runnable() {
    @Override
    public void run() {
```

```
        CocosJavascriptJavaBridge.evalString("NativeAPIMgr.inst.callByNative()");
    }
});
```

参数传递

以上几种 Java 调用 JS 的方式，均支持参数传递，但参数只支持 string, number 和 bool 三种基础类型。

我们以全局函数为例：

```
window.callByNative = function(a:string, b:number, c:bool){
  //to do
}
```

可像这样调用：

```
CocosHelper.runOnGameThread(new Runnable() {
    @Override
    public void run() {
        CocosJavascriptJavaBridge.evalString("window.callByNative('test',1,true)");
    }
});
```

在 C++ 代码中调用 JavaScript

如果要在 C++ 中调用 `evalString`，我们可以参考下面的方式，确保 `evalString` 在 JavaScript 引擎所在的线程被执行：

```
CC_CURRENT_ENGINE()->getScheduler()->performFunctionInCocosThread([=]() {
    se::ScriptEngine::getInstance()->evalString(script.c_str());
});
```

线程安全

可以看到，上面的代码中，使用了 `CocosHelper.runOnGameThread` 和 `CC_CURRENT_ENGINE()->getScheduler()->performFunctionInCocosThread`。这是为了代码在执行时处于正确的线程，详情请参考：线程安全。

Objective-C 原生反射机制

基于反射机制实现 JavaScript 与 iOS/macOS 系统原生通信

JavaScript 调用 Objective-C 代码

在 Cocos Creator 中提供了依靠语言反射机制的跨语言通信方式，使 JavaScript 可以直接调用 Objective-C 函数的方法。 方法原型示例如下：

```
var result = native.reflection.callStaticMethod(className, methodName, arg1, arg2, .....);
```

在 `native.reflection.callStaticMethod` 方法中，我们通过传入 Objective-C 的类名、方法名、参数就可以直接调用 Objective-C 的静态方法，并且可以获得 Objective-C 方法的返回值。

> **注意**：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 以及 Weex。

为了降低提审不通过的风险，建议参考 使用 JsbBridge 实现 JavaScript 与 Objective-C 通信。

类名与静态方法

参数中的类名不需要路径，只需要传入 Objective-C 中的类名即可。比如你在工程目录下的任意文件中新建一个类 `NativeOcClass`，只要你将它引入工程即可。

再次强调一下，仅支持 JavaScript 调用 Objective-C 中类的静态方法。

带参方法

方法名比较需要注意。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，需要将它的 **:** 也带上。

有参方法定义示例：

```
import <Foundation/Foundation.h>
@interface NativeOcClass : NSObject
+(BOOL)callNativeUIWithTitle:(NSString *) title andContent:(NSString *)content;
@end
```

有参方法调用示例：

```
if(sys.isNative && (sys.os == sys.OS.IOS || sys.os == sys.OS.OSX)){
    var ret = native.reflection.callStaticMethod("NativeOcClass",
                                                 "callNativeUIWithTitle:andContent:",
                                                 "cocos2d-js",
                                                 "Yes! you call a Native UI from Reflection");
}
```

`sys.isNative` 用于判断是否为原生平台，`sys.os` 用于判断当前运行系统。由于各平台通信机制不同，建议先判断再处理。

> **注意**：此时的方法名是 `callNativeUIWithTitle:andContent:`，不要漏掉了冒号 **:** 。

无参方法

如果是没有参数的函数，那么就不需要 **:**。如下面代码中的方法名是 `callNativeWithReturnString`，由于没有参数，就不需要 **:**，跟 Objective-C 的 method 写法一致。

无参方法定义示例：

```
+(NSString *)callNativeWithReturnString;
```

无参方法调用示例：

```
var ret = native.reflection.callStaticMethod("NativeOcClass",
                                             "callNativeWithReturnString");
```

返回值

这里是这个方法在 Objective-C 的实现，可以看到是弹出了一个原生对话框。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:@"Cancel" otherButtonTitles:@"OK", nil];
  [alertView show];
  return true;
}
```

此时，你就可以在 `ret` 中接收到从 Objective-C 传回的返回值（true）了。

参数类型转换

在 Objective-C 的实现中，如果方法的参数需要使用 float、int 和 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，传入两个浮点数，然后计算它们的合并返回，我们使用 NSNumber 作为参数类型，而不是 int 和 float。

```
+(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以下类型：

- `int`
- `float`
- `bool`
- `string`

其余的类型暂时不支持。

如果对如何在项目中新增 `Objective-C` 文件不熟悉，可参考 原生平台二次开发指南。

Objective-C 执行 JavaScript 代码

在 Cocos Creator 项目中，我们也可以通过 `evalString` 方法，在 C++ 或者 Objective-C 中执行 JavaScript 代码。

调用示例如下：

```
CC_CURRENT_ENGINE()->getScheduler()->performFunctionInCocosThread([=](){
    se::ScriptEngine::getInstance()->evalString(script.c_str());
});
```

> **注意**：除非确定当前线程是 **主线程**，否则都需要使用 `performFunctionInCocosThread` 方法将函数分发到主线程中去执行。

调用全局函数

我们可以在脚本中通过如下代码新增一个全局函数：

```
window.callByNative = function(){
  //to do
}
```

> `window` 是 Cocos 引擎脚本环境中的全局对象，如果要让一个变量、函数、对象或者类全局可见，需要将它作为 `window` 的属性。可以使用 `window.变量名` 或者 `变量名` 进行访问。

然后像下面这样调用:

```
CC_CURRENT_ENGINE()->getScheduler()->performFunctionInCocosThread([=](){
    se::ScriptEngine::getInstance()->evalString("window.callByNative()");
});
```

或者：

```
CC_CURRENT_ENGINE()->getScheduler()->performFunctionInCocosThread([=](){
    se::ScriptEngine::getInstance()->evalString("callByNative()");
});
```

调用类的静态函数

假如在 TypeScript 脚本中有一个类具有如下静态函数：

```
export class NativeAPI{
  public static callByNative(){
    //to do
  }
}
// 将 NativeAPI 注册为全局类，否则无法在 OC 中被调用
window.NativeAPI = NativeAPI;
```

我们可以像这样调用：

```
CC_CURRENT_ENGINE()->getScheduler()->performFunctionInCocosThread([=](){
    se::ScriptEngine::getInstance()->evalString("NativeAPI.callByNative()");
});
```

调用单例函数

如果脚本代码中，有实现可以全局访问的单例对象

```
export class NativeAPIMgr{
  private static _inst:NativeAPIMgr;

  public static get inst():NativeAPIMgr{
    if(!this._inst){
      this._inst = new NativeAPIMgr();
    }
    return this._inst;
  }

  public static callByNative(){
    //to do
  }
}

// 将 NativeAPIMgr 注册为全局类，否则无法在 OC 中被调用
window.NativeAPIMgr = NativeAPIMgr;
```

我们可以像下面这样调用：

```
CC_CURRENT_ENGINE()->getScheduler()->performFunctionInCocosThread([=](){
    se::ScriptEngine::getInstance()->evalString("NativeAPIMgr.inst.callByNative()");
});
```

参数传递

以上几种 OC 调用JS的方式，均支持参数传递，但参数只支持 string, number 和 bool 三种基础类型。

我们以全局函数为例：

```
window.callByNative = function(a:string, b:number, c:bool){
  //to do
}
```

可像这样调用：

```
CC_CURRENT_ENGINE()->getScheduler()->performFunctionInCocosThread([=](){
    se::ScriptEngine::getInstance()->evalString("window.callByNative('test',1,true)");
});
```

线程安全

可以看到，上面的代码中，使用了 `CC_CURRENT_ENGINE()->getScheduler()->performFunctionInCocosThread`。这是为了代码在执行时处于正确的线程，详情请参考：线程安全。

JsbBridge JS 与 JAVA 通信

使用 JsbBridge 实现 JavaScript 与 Java 通信

背景

基于反射机制实现 JavaScript 与 Android 系统原生通信 的方法中，我们不仅需要严格声明包名和函数签名，还需要严格校对参数数量以确保正常运行，步骤较为复杂。

因此我们额外提供了另外一种方法，用于简化脚本层到原生层的调用。这是一种通道，或者说是一个桥梁，我们将其命名为 `JsbBridge`，意为通过 `JSB` 绑定作为沟通脚本和原生 APP 的桥梁。

> **注意**：两种方式都是可以正常使用的，开发者可以根据实际需要选择使用。

调用机制

JavaScript 接口介绍

在脚本层只有 `sendToNative` 和 `onNative` 两个接口，定义如下：

```
// JavaScript
export namespace bridge{
    /**
     * Send to native with at least one argument.
     */
    export function sendToNative(arg0: string, arg1?: string): void;
    /**
     * Save your own callback controller with a JavaScript function,
     * Use 'jsb.bridge.onNative = (arg0: String, arg1: String | null)=>{...}'
     * @param args : received from native
     */
    export function onNative(arg0: string, arg1?: string | null): void;
}
```

见名知义，`sendToNative` 用于调用原生层的代码，而 `onNative` 用于响应原生层的调用。

使用时需要注意以下几点：

- 由于现在这个功能还在实验阶段，所以只支持 `string` 的传输，如果需要传输包含多种参数的对象，请考虑将其转化为 `Json` 形式进行传输，并在不同层级解析。
- `onNative` 同一时间只会记录一个函数，当再次 `set` 该属性时会覆盖原先的 `onNative` 方法。
- `sendToScript` 方法是单向通信，不会关心下层的返回情况，也不会告知 `JavaScript` 操作成功或者失败。开发者需要自行处理操作情况。

Java 接口介绍

在 `JAVA` 中，也有两对应的接口， `sendToScript` 和 `onScript`，定义如下：

```
// JAVA
public class JsbBridge {
    public interface ICallback{
        /**
         * Applies this callback to the given argument.
         *
         * @param arg0 as input
         * @param arg1 as input
         */
        void onScript(String arg0, String arg1);
    }
    /** Add a callback which you would like to apply
     * @param f ICallback, the method which will be actually applied. multiple calls will override
     * */
    public static void setCallback(ICallback f);
    /**
     * Java dispatch Js event, use native c++ code
     * @param arg0 input values
     */
    public static void sendToScript(String arg0, String arg1);
    public static void sendToScript(String arg0);
}
```

其中 `sendToScript` 用于调用脚本层代码，而 `onScript` 用于响应脚本层的调用。

我们需要实现 `ICallback` 接口，并且使用 `setCallback` 注册，来响应 `onScript` 的具体行为。

基本使用

JavaScript 触发 Java 的方法

假设我们用 Java 写了一个打开广告的接口，当玩家点击打开广告的按钮时，应该由 JavaScript 调用对应的接口，触发打开广告的操作。

我们需要先实现一个 ICallback 接口，用于响应操作，并利用 `JsbBridge.setCallback` 注册，代码如下：

```
JsbBridge.setCallback(new JsbBridge.ICallback() {
    @Override
    public void onScript(String arg0, String arg1) {
        //TO DO
        if(arg0.equals("open_ad")){
            //call openAd method.
        }
    }
});
```

> 实际项目中，上面的代码一般会在 `AppActivity.java` 的 `onCreated` 中直接或者间接被调用，以确保能够响应所有来自脚本层的调用。

在 JavaScript 脚本中，我们就可以像下面一样调用：

```
import { native } from 'cc'
public static onclick(){
    native.bridge.sendToNative('open_ad', defaultAdUrl);
}
```

JAVA 触发 JavaScript 的方法

假设我们的广告播放完成后，需要通知 JavaScript 层，可以像下面这样操作。

首先，需要在 JavaScript 使用 `onNative` 响应事件：

```
native.bridge.onNative = (arg0:string, arg1: string):void=>{
    if(arg0 == 'ad_close'){
        if(arg1 == "finished") {
            //ad playback completed.
        }
        else{
            //ad cancel.
        }
    }
    return;
```

```
}
```

实际项目中，可以将以面代码写在一个程序启动时就要加载的脚本脚本的 onload 函数中，以确保尽早监听来自原生层的事件。

然后，在 `Java` 中，用如下代码调用：

```
JsbBridge.sendToScript("ad_close", "finished");
```

通过上述操作，便可以通知 JavaScript 广告的播放结果了。

最佳实践

JsbBridge 提供了 arg0 和 arg1 两个 string 类型的参数用于传递信息，可以根据不同的需求进行分配。

1. arg0 和 arg1 均用于参数

如果通信需求比较简单，不需要进行分类处理，则可以将 arg0 和 arg1 用作参数传递。

2. arg0 用于分类标记, arg1 用于参数

如果通信需求相对复杂，可以使用 arg0 作为分类标记，根据 arg0 来分类处理， arg1 用于参数传递

3. arg0 用于分类标记， arg1 作为 JSON 字符串

对于特别复杂的需求， 单纯的 string 类型参数无法满足，此时可以将需要传递的对象通过 `JSON.stringfy` 转化为字符串，再通过 arg1 进行传递。 使用时，再利用 `JSON.parse` 还原为对象，做后续的处理。

由于涉及到 JSON 的序列化和反序列化操作，这种使用方式不建议高频调用。

线程安全

注意，如果相关代码涉及到原生 UI 的部分，就需要考虑线程安全问题，详情请参考：线程安全。

示例工程：简单的多事件调用

Cocos Creator 提供了 **native-script-bridge**（GitHub | Gitee）范例，开发者可根据需要自行下载以参考使用。

JsbBridge JS 与 Objective-C 通信

使用 JsbBridge 实现 JavaScript 与 Objective-C 通信

背景

基于反射机制实现 JavaScript 与 iOS/macOS 系统原生通信 的方法中，我们不仅需要严格声明包名和函数签名，还需要严格校对参数数量以确保正常运行，步骤较为复杂。

因此我们额外提供了另外一种方法，用于简化脚本层到原生层的调用。这是一种通道，或者说是一个桥梁，我们将其命名为 `JsbBridge`，意为通过 `JSB` 绑定作为沟通脚本和原生 APP 的桥梁。

注意：两种方式都是可以正常使用的，开发者可以根据实际需要选择使用。

调用机制

JavaScript 接口介绍

在脚本层只有 `sendToNative` 和 `onNative` 两个接口，定义如下：

```
// JavaScript
export namespace bridge{
    /**
     * Send to native with at least one argument.
     */
    export function sendToNative(arg0: string, arg1?: string): void;
    /**
     * Save your own callback controller with a JavaScript function,
     * Use 'jsb.bridge.onNative = (arg0: String, arg1: String | null)=>{...}'
     * @param args : received from native
     */
    export function onNative(arg0: string, arg1?: string | null): void;
}
```

见名知义，`sendToNative` 用于调用原生层的代码，而 `onNative` 用于响应原生层的调用。

使用时需要注意以下几点：

- 由于现在这个功能还在实验阶段，所以只支持 `string` 的传输，如果需要传输包含多种参数的对象，请考虑将其转化为 `Json` 形式进行传输，并在不同层级解析。
- `onNative` 同一时间只会记录一个函数，当再次 `set` 该属性时会覆盖原先的 `onNative` 方法。
- `sendToScript` 方法是单向通信，不会关心下层的返回情况，也不会告知 `JavaScript` 操作成功或者失败。开发者需要自行处理操作情况。

Objective-C 接口介绍

在 `Objective-C` 中，也有两对应的接口， `sendToScript` 和 `callByScript`，定义如下：

```
//Objective-c
typedef void (^ICallback)(NSString*, NSString*);

@interface JsbBridge : NSObject

+(instancetype)sharedInstance;
-(bool)setCallback:(ICallback)cb;
-(bool)callByScript:(NSString*)arg0 arg1:(NSString*)arg1;
-(void)sendToScript:(NSString*)arg0 arg1:(NSString*)arg1;
-(void)sendToScript:(NSString*)arg0;

@end
```

其中 `sendToScript` 用于调用脚本层代码，而 `callByScript` 用于响应脚本层的调用。

我们需要实现 `ICallback` 接口，并且使用 `setCallback` 注册，来响应 `callByScript` 的具体行为。

基本使用

JavaScript 触发 Objective-C 的回调

假设我们用 Objective-C 写了一个打开广告的接口，当玩家点击打开广告的按钮时，应该由 JavaScript 调用对应的 Objective-C 接口，触发打开广告的操作。

我们需要先实现一个 ICallback 接口，用于响应操作，然后通过 `setCallback` 方法，注册到 `JsbBridge`。

Objective-C 代码如下：

```
#include "platform/apple/JsbBridge.h"
```

```
static ICallback cb = ^void (NSString* _arg0, MSString* _arg1){
    if([_arg0 isEqual:@"open_ad"]){
        //open Ad
    }
};

JsbBridge* m = [JsbBridge sharedInstance];
[m setCallback:cb];
```

在 JavaScript 脚本中，我们就可以像下面一样调用：

```
import { native } from 'cc'
public static onclick(){
    native.bridge.sendToNative('open_ad', defaultAdUrl);
}
```

Objective-C 触发 JavaScript 的回调

假设我们的广告播放完成后，需要通知 JavaScript 层，可以像下面这样操作。

首先，需要在 JavaScript 使用 `onNative` 响应事件：

```
native.bridge.onNative = (arg0:string, arg1: string):void=>{
    if(arg0 == 'ad_close'){
        if(arg1 == "finished") {
            //ad playback completed.
        }
        else{
            //ad cancel.
        }
    }
    return;
}
```

然后，在 `Objective-C` 中，用如下代码调用：

```
#include "platform/apple/JsbBridge.h"

JsbBridge* m = [JsbBridge sharedInstance];
[m sendToScript:@"ad_close" arg1:@"finished"];
```

通过上述操作，便可以通知 JavaScript 广告的播放结果了。

最佳实践

JsbBridge 提供了 arg0 和 arg1 两个 string 类型的参数用于传递信息，可以根据不同的需求进行分配。

1. arg0 和 arg1 均用于参数

如果通信需求比较简单，不需要进行分类处理，则可以将 arg0 和 arg1 用作参数传递。

2. arg0 用于分类标记, arg1 用于参数

如果通信需求相对复杂，可以使用 arg0 作为分类标记，根据 arg0 来分类处理， arg1 用于参数传递

3. arg0 用于分类标记， arg1 作为 JSON 字符串

对于特别复杂的需求， 单纯的 string 类型参数无法满足，此时可以将需要传递的对象通过 `JSON.stringfy` 转化为字符串，再通过 arg1 进行传递。 使用时，再利用 `JSON.parse` 还原为对象，做后续的处理。

> 由于涉及到 JSON 的序列化和反序列化操作，这种使用方式不建议高频调用。

线程安全

注意，如果相关代码涉及到原生 UI 的部分，就需要考虑线程安全问题，详情请参考：线程安全。

示例工程：简单的多事件调用

Cocos Creator 提供了 **native-script-bridge**（GitHub | Gitee）范例，开发者可根据需要自行下载以参考使用。

JsbBridgeWrapper 基于原生反射机制的事件处理

JsbBridgeWrapper 基于原生反射机制的事件处理

> **注意**：在 v3.6 之后，jsb 模块将会逐步废弃，接口将会迁移到 cc 命名空间下的 native 模块。

JsbBridgeWrapper 是基于事件机制，用于 JS 层与原生层进行通信的接口。

建立于 JsbBridge 上的事件分发机制

`JsbBridgeWrapper` 是封装在 `JsbBridge` 之上的事件派发机制，相对于 `JsbBridge` 而言它更方便易用。开发者不需要手动去实现一套消息收发机制就可以进行多事件的触发。但它不具有多线程稳定性或者是 100% 安全。如果遇到复杂需求场景，仍然建议自己实现对应的事件派发。

JsbBridgeWrapper 接口介绍

```
/**
 * Listener for jsbBridgeWrapper's event.
 * It takes one argument as string which is transferred by jsbBridge.
 */
export type OnNativeEventListener = (arg: string) => void;
export namespace jsbBridgeWrapper {
    /** If there's no event registered, the wrapper will create one  */
    export function addNativeEventListener(eventName: string, listener: OnNativeEventListener);
    /**
     * Dispatch the event registered on Objective-C, Java etc.
     * No return value in JS to tell you if it works.
     */
    export function dispatchEventToNative(eventName: string, arg?: string);
    /**
     * Remove all listeners relative.
     */
    export function removeAllListenersForEvent(eventName: string);
    /**
     * Remove the listener specified
     */
    export function removeNativeEventListener(eventName: string, listener: OnNativeEventListener);
    /**
     * Remove all events, use it carefully!
     */
    export function removeAllListeners();
}
```

`OnNativeEventListener` 是实际注册的 **回调（callback）** 类型，为了防止因为类型不匹配导致的低级错误，因此使用显示声明该类型。`addNativeEventListener` 中的第二个参数即为传入的 callback。当然也可以使用匿名函数代替。代码示例如下：

```
import { native } from 'cc'
// 当事件 “A” 触发时， ‘this.A’ 方法会被调用
native.jsbBridgeWrapper.addNativeEventListener("A", (usr: string) => {
    this.A(usr);
});
```

注意：这里是为了防止 this 指向不明确，所以使用匿名函数封装一层作用域。

JsbBridgeWrapper 接口说明

addNativeEventListener

增加一个事件监听。

参数：

- eventName: string 事件名称
- listener: OnNativeEventListener 回调函数

dispatchEventToNative

派发一个事件到原生层。

参数：

- eventName: string 事件名称
- arg?: string 参数

removeNativeEventListener

删除事件监听。

参数：

- eventName: string 事件名称
- listener: OnNativeEventListener 要删除的回调函数

removeAllListeners

删除所有的事件监听。

原生平台 JsbBridgeWrapper 的实现

`JsbBridgeWrapper` 在不同平台有不同的实现，开发者可以通过下列方式进行查看：

- 在 Objective-C 端，可查看 JsbBridgeWrapper.h：

```
//In Objective-C
typedef void (^OnScriptEventListener)(NSString*);

@interface JsbBridgeWrapper : NSObject
/**
 * Get the instance of JsbBridgetWrapper
 */
+ (instancetype)sharedInstance;
/**
 * Add a listener to specified event, if the event does not exist, the wrapper will create one. Concurrent listener will be ignored
 */
- (void)addScriptEventListener:(NSString*)eventName listener:(OnScriptEventListener)listener;
/**
 * Remove listener for specified event, concurrent event will be deleted. Return false only if the event does not exist
 */
- (bool)removeScriptEventListener:(NSString*)eventName listener:(OnScriptEventListener)listener;
/**
 * Remove all listener for event specified.
 */
- (void)removeAllListenersForEvent:(NSString*)eventName;
/**
 * Remove all event registered. Use it carefully!
 */
- (void)removeAllListeners;
/**
 * Dispatch the event with argument, the event should be registered in javascript, or other script language in future.
 */
- (void)dispatchEventToScript:(NSString*)eventName arg:(NSString*)arg;
/**
 * Dispatch the event which is registered in javascript, or other script language in future.
 */
- (void)dispatchEventToScript:(NSString*)eventName;
@end
```

- 安卓可查看 JsbBridgeWrapper.java：

```
// In JAVA
public class JsbBridgeWrapper {
    public interface OnScriptEventListener {
        void onScriptEvent(String arg);
    }
    /**
     * Add a listener to specified event, if the event does not exist, the wrapper will create one. Concurrent listener will be ignored
     */
    public void addScriptEventListener(String eventName, OnScriptEventListener listener);
    /**
     * Remove listener for specified event, concurrent event will be deleted. Return false only if the event does not exist
     */
    public boolean removeScriptEventListener(String eventName, OnScriptEventListener listener);
    /**
     * Remove all listener for event specified.
     */
    public void removeAllListenersForEvent(String eventName);
    /**
     * Remove all event registered. Use it carefully!
     */
    public void removeAllListeners() {
        this.eventMap.clear();
    }
    /**
     * Dispatch the event with argument, the event should be registered in javascript, or other script language in future.
     */
    public void dispatchEventToScript(String eventName, String arg);
    /**
     * Dispatch the event which is registered in javascript, or other script language in future.
     */
    public void dispatchEventToScript(String eventName);
}
```

- Huawei HarmonyOS 也可以通过 JsbBridgeWrapper.java 查看其实现方式。

使用 JsbBridgeWrapper

常见的需求如数据存放在原生层，当需要将数据取至 JS 层时，可以通过 JsbBridgeWrapper 实现。

下文通过一个示例说明，如何通过原生的回调结果改变 label 内容，当原生层的事件被触发时，将目标文本字符回传给 JS 层。

注册 JS 事件

JS 层需要首先注册一个 `changeLabelContent` 事件监听。

```
public changeLabelContent(user: string): void {
    console.log("Hello " + user + " I'm K");
    this.labelForContent!.string = "Hello " + user + " ! I'm K";
}
native.jsbBridgeWrapper.addNativeEventListener("changeLabelContent", (usr: string) => {
        this.changeLabelContent(usr);
});
```

当 JS 层的 `changeLabelContent` 事件被触发时，标签的内容会变成对应的字符串组合。接下来需要处理原生的事件注册。

原生事件注册与派发

- 在 Objective-C 端使用下列代码：

```
// Objective-C
JsbBridgeWrapper* m = [JsbBridgeWrapper sharedInstance];
OnScriptEventListener requestLabelContent = ^void(NSString* arg){
    JsbBridgeWrapper* m = [JsbBridgeWrapper sharedInstance];
    [m dispatchEventToScript:@"changeLabelContent" arg:@"Charlotte"];
};
[m addScriptEventListener:@"requestLabelContent" listener:requestLabelContent];
```

- 在 JAVA 端使用如下代码：

```
// JAVA
JsbBridgeWrapper jbw = JsbBridgeWrapper.getInstance();
jbw.addScriptEventListener("requestLabelContent", arg ->{
    System.out.print("@JAVA: here is the argument transport in" + arg);
    jbw.dispatchEventToScript("changeLabelContent","Charlotte");
});
```

  注意：JAVA 可以通过匿名函数的方法来实现 interface 的需求，此处写法简化。

这里原生的返回值被设置成固定字符，但开发者可以根据需求实现异步亦或是延后的字符赋值，时机并非固定。简而言之，当原生收到 `requestLabelContent` 的事件时，原生将会反过来触发 JS 层的 `changeLabelContent` 的事件，并将字符作为事件触发的传参。

在场景中派发事件

最后一步，我们在场景中添加一个按钮和对应的事件。

```
// 按钮点击事件 SAY HELLO
public sayHelloBtn() {
    native.jsbBridgeWrapper.dispatchEventToNative("requestLabelContent");
}
```

最终的效果和 JsbBridge 的测试例效果相同。点击 `SAY HELLO` 按钮，第一行的内容会改变为打过招呼的信息，否则即为失败。

使用 `JsbBridgeWrapper` 模块时，开发者不需要自己去维护多余的机制，只需要关心是否正确注册和取消注册即可。

JSB 2.0 使用指南

JSB 2.0 绑定教程

抽象层

架构

JSB 手动绑定

使用 JSB 手动绑定

> 本文转载自 腾讯在线教育部技术博客
> 作者：晋中望（xepherjin）

背景

一直以来，ABCmouse 项目中的整体 JS/Native 通信调用结构都是基于 `callStaticMethod <-> evalString` 的方式。通过 `callStaticMethod` 方法我们可以通过反射机制直接在 JavaScript 中调用 `Java/Objective-C` 的静态方法。而通过 `evalString` 方式，则可以执行 JS 代码，这样便可以进行双端通信。

JSB 自动绑定

使用 JSB 自动绑定

> 本文转载自 腾讯在线教育部技术博客
> 作者：张鑫（kevinxzhang）

尽管 Creator 提供了 `native.reflection.callStaticMethod` 方式支持从 ts 端直接调用 Native 端（Android/iOS/Mac）的接口，但是经过大量实践发现此接口在大量频繁调用情况下性能很低下，尤其是在 Android 端，比如调用 Native 端实现的打印 log 的接口，而且会容易引起一些 native crash，例如 `local reference table overflow` 等问题。纵观 Cocos 原生代码的实现，基本所有的接口方法的实现都是基于 JSB 的方式来实现，所以此文主要讲解下 JSB 的自动绑定逻辑，帮助大家能快速实现 `callStaticMethod` 到 JSB 的改造过程。

背景

对于用过 Cocos Creator（为了方便后文直接简称 CC）的人来说，`jsb.reflection.callStaticMethod` 这个方法肯定不陌生，其提供了我们从 ts 端调用 Native 端的能力，例如我们要调用 Native 实现的 log 打印和持久化的接口，就可以很方便的在 JavaScript 中按照如下的操作调用即可：

```
import {NATIVE} from 'cc/env';

if (NATIVE && sys.os == sys.OS.IOS) {
    msg = this.buffer_string + '\n[cclog][' + clock + '][' + tag + ']' + msg;
    native.reflection.callStaticMethod("ABCLogService", "log:module:level:", msg, 'cclog', level);
    return;
} else if (NATIVE && sys.os == sys.OS.ANDROID) {
    msg = this.buffer_string + '\n[cclog][' + clock + '][' + tag + ']' + msg;
    native.reflection.callStaticMethod("com/example/test/CommonUtils", "log", "(ILjava/lang/String;Ljava/lang/String;)V", level, 'cclog', msg);
    return;
}
```

尽管使用很简单，一行代码就能实现跨平台调用，稍微看下其实现就可以知道，在 C++ 层 Android 端是通过 jni 的方式实现的，IOS 端是通过运行时的方式动态调用，但是为了兼顾通用性和支持所有的方

法，Android 端没有对 jni 相关对象做缓存机制，就会导致短时间大量调用时出现很严重的性能问题，之前我们遇到的比较多的情况就是在下载器中打印 log，某些应用场景短时间内触发大量的下载操作，就会出现 `local reference table overflow` 的 crash，甚至在低端机上导致界面卡顿无法加载出来的问题。

修复此问题就需要针对 log 调用进行 JSB 的改造，同时还要加上 jni 的相关缓存机制，优化性能。jSB 绑定说白了就是 C++ 和脚本层之间进行对象的转换，并转发脚本层函数调用到 C++ 层的过程。

JSB 绑定通常有 **手动绑定** 和 **自动绑定** 两种方式。手动绑定方式可以参考 使用 JSB 手动绑定。

- 手动绑定方式优点是灵活，可定制型强；缺点就是全部代码要自己书写，尤其是在 ts 类型跟 c++ 类型转换上，稍有不慎容易导致内存泄漏，某些指针或者对象没有释放。
- 自动绑定方式则会帮你省了很多麻烦，直接通过一个脚本一键生成相关的代码，后续如果有新增或者改动，也只需要重新执行一次脚本即可。所以自动绑定对于不需要进行强定制，需要快速完成 JSB的情况来说就再适合不过了。下面就一步步说明下如何实现自动绑定 JSB。

环境配置和自动绑定展示

环境配置

自动绑定，说简单点，其实就只要执行一个 python 脚本即可自动生成对应的 `.cpp`、`.h` 文件。所以首先要保证电脑有 python 运行环境，这里以 Mac 上安装为例来讲解。

1. 安装 python 3.0 版本，从 python 官网下载安装包：

   https://www.python.org/downloads/release/python-398/

2. 通过 pip3 安装 python 的一些依赖库：

   ```
   sudo pip3 install pyyaml==5.4.1
   sudo pip3 install Cheetah3
   ```

3. 安装 NDK，涉及到 c++ 肯定这个是必不可少的，建议安装 Android NDK r21e 版本，然后在 `~/.bash_profile` 中设置 `PYTHON_ROOT` 和 `NDK_ROOT` 这两个环境变量，因为在后面执行的 python 文件里面就会直接用到这两个环境变量：

   ```
   export NDK_ROOT=/Users/kevin/android-ndk-r21e
   export PYTHON_BIN=python3
   ```

Windows 下直接参考上面需要安装的模块直接安装就好了，最后也要记得配置环境变量。

自动绑定展示

这里演示的是 cocos 引擎下面也即 **generated/cocos/bindings/auto** 目录下的文件（如下图所示）是怎么自动生成的：

Swig

简介

从 Cocos Creator 3.7.0 开始，我们将生成 JS 绑定代码的方式从 bindings-generator 改为 Swig。 Swig 通过解析与 C++ 兼容的接口定义语言 (IDL) 的方式来生成胶水代码，此方式有较多好处 。 关于我们为什么改用 Swig，可以参考此 issue。

为引擎模块生成绑定代码

在 3.8 及以上的版本中，开发者不再需要手动触发绑定代码的生成，因为 CMake 会自动在编译过程中调用 SWIG 命令来生成绑定代码。如果生成失败, 请注意终端的输出日志。

需要注意: 如果为引擎添加或删除了 `.i` 文件，你需要重新生成工程来确保更新对 `.i` 文件的引用，并生成更新的绑定代码。

为开发者的项目生成绑定代码

- 确保你已经安装了 NodeJS，版本号大于或等于 v8.9.4
- 打开终端 (macOS / Linux) 或者命令提示符 (Windows),
- 创建一个用于存放自动绑定胶水代码的目录，例如： `/Users/abc/my-project/native/engine/common/Classes/bindings/auto`
- 写一个 JS 配置文件
  - 创建一个 JS 配置文件, 路径例如： `/Users/abc/my-project/tools/swig-config/swig-config.js` ，内容为：

    ```
    'use strict';
    const path = require('path');

    // 开发者自己的模块定义配置
    // configList 是必须的
    const configList = [
      [ 'your_module_interface_0.i', 'jsb_your_module_interface_0_auto.cpp' ],
      [ 'your_module_interface_1.i', 'jsb_your_module_interface_1_auto.cpp' ],
      // ......
    ];

    const projectRoot = path.resolve(path.join(__dirname, '..', '..'));
    // interfaceDir 是可选的
    const interfacesDir = path.join(projectRoot, 'tools', 'swig-config');
    // bindingsOutDir 是可选的
    const bindingsOutDir = path.join(projectRoot, 'native', 'engine', 'common', 'Classes', 'bindings', 'auto');

    module.exports = {
      interfacesDir, // 可选参数, 如果没有指定, configList 中的路径必须为绝对路径或者相对于当前 swig-config.js 的相对路径
      bindingsOutDir, // 可选参数, 如果没有指定, configList 中的路径必须为绝对路径或者相对于当前 swig-config.js 的相对路径
      configList // 必填参数
    };
    ```

  - 执行如下命令

    ```
    # 如果当前终端或者命令提示符所在的目录不是在 '/Users/abc/my-project/tools/swig-config'
    $ node < 引擎的根目录 >/native/tools/swig-config/genbindings.js -c /Users/abc/my-project/tools/swig-config/swig-config.js

    # 如果你已经在 '/Users/abc/my-project/tools/swig-config' 目录, 你执行命令的时候可以不需要带上 -c 参数, 例如:
    $ cd /Users/abc/my-project/tools/swig-config
    $ node < 引擎的根目录 >/native/tools/swig-config/genbindings.js
    ```

Swig 接口定义文件

- 在引擎的 `engine/native/tools/swig-config` 目录下有一个 swig-interface-template.i 模版文件。你可以拷贝其到自己的工程目录下并重命名。此模版文件中包含一些注释用于展示如何在 .i 文件中配置你的模块。你也可以参考在 `engine/native/tools/swig-config` 目录下引擎内部的 .i 文件，例如：参考 `scene.i` 或者 `assets.i` 来快速上手。
- 如果你使用 `Visual Studio Code`, 你可以安装 `Hong-She Liang` 开发的 `SWIG Language` 扩展，其可用于 .i 文件的语法高亮。
- 关于编写 .i 文件的更多详细信息，建议参考下面 教程 章节。

教程

请访问 在 Cocos Creator 中的 Swig 工作流教程, 其包含如何一步一步地为引擎内的新模块或用户工程模块配置绑定。

Swig 示例

在 Cocos Creator 中的 Swig 工作流教程

如何为引擎内的新模块添加绑定

添加一个新模块的接口文件

- 添加一个新模块的接口文件到 `native/tools/swig-config` 目录, 例如： `new-engine-module.i`
- 拷贝 swig-interface-template.i 文件中的内容到 new-engine-module.i
- 添加必要的配置，可以参考 `native/tools/swig-config` 目录下现有的 .i 文件配置，或者参考下面的章节内容。

修改 `engine/native/cocos/CMakeLists.txt`

```
######## auto
cocos_source_files(
    NO_WERROR   NO_UBUILD   ${SWIG_OUTPUT}/jsb_cocos_auto.cpp # 添加此行
                            ${SWIG_OUTPUT}/jsb_cocos-auto.h # 添加此行
    NO_WERROR   NO_UBUILD   ${SWIG_OUTPUT}/jsb_cocos_auto.cpp
                            ${SWIG_OUTPUT}/jsb_cocos_auto.h
    ......
```

为脚本引擎注册新的模块

打开 `jsb_module_register.cpp`，做如下修改

```
......
#if CC_USE_PHYSICS_PHYSX
    #include "cocos/bindings/auto/jsb_physics_auto.h"
#endif
#include "cocos/bindings/auto/jsb_new_engine_module_auto.h" // 添加此行

bool jsb_register_all_modules() {
    se::ScriptEngine *se = se::ScriptEngine::getInstance();
    ......
    se->addRegisterCallback(register_all_my_new_engine_module); // 添加此行

    se->addAfterCleanupHook([]() {
        cc::DeferredReleasePool::clear();
        JSBClassType::cleanup();
    });
    return true;
}
```

如何为开发者的项目绑定一个新模块

假定我们已经有一个 Cocos Creator 的工程，其位于 `/Users/james/NewProject` 目录下。

打开 Cocos Creator 的构建面板，构建出一个原生平台的工程，会生成 `/Users/james/NewProject/native` 目录。

绑定一个简单的类

创建一个简单类

创建一个头文件，其位于 `/Users/james/NewProject/native/engine/Classes/MyObject.h`, 其内容为：

```
// MyObject.h
#pragma once
#include "cocos/cocos.h"
namespace my_ns {
class MyObject {
public:
    MyObject() = default;
    MyObject(int a, bool b) {}
    virtual ~MyObject() = default;
    void print() {
        CC_LOG_DEBUG("==> a: %d, b: %d\n", _a, (int)_b);
    }

    float publicFloatProperty{1.23F};
private:
    int _a{100};
    bool _b{true};
};
} // namespace my_ns {
```

编写一个 Swig 接口文件

创建一个名称为 `my-module.i` 的接口文件，其位于 `/Users/james/NewProject/tools/swig-config`目录下。

```
// my-module.i
%module(target_namespace="my_ns") my_module

// %insert(header_file) %{ ... }%} 代码块中的内容最终会被原封不动地插入到生成的头文件(.h)开头的地方
%insert(header_file) %{
#pragma once
#include "bindings/jswrapper/SeApi.h"
#include "bindings/manual/jsb_conversions.h"

#include "MyObject.h" // 添加这行，%include 指令表示让 swig 解析此文件，并且为此文件中的类生成绑定代码。
%}

// %{ ... %} 代码块中的内容最终会被原封不动地插入到生成的源文件(.cpp)开头的地方
%{
#include "bindings/auto/jsb_my_module_auto.h"
%}

%include "MyObject.h"
```

编写一个 Swig 配置文件（swig-config.js）

创建一个名为 swig-config.js 的文件，例如： `/Users/james/NewProject/tools/swig-config`目录下。

```
// swig-config.js
'use strict';
const path = require('path');
const configList = [
    [ 'my-module.i', 'jsb_my_module_auto.cpp' ],
];

const projectRoot = path.resolve(path.join(__dirname, '..', '..'));
const interfacesDir = path.join(projectRoot, 'tools', 'swig-config');
const bindingsOutDir = path.join(projectRoot, 'native', 'engine', 'common', 'bindings', 'auto');
// includeDirs 意思是 swig 执行时候使用的头文件搜索路径
const includeDirs = [
    path.join(projectRoot, 'native', 'engine', 'common', 'Classes'),
];

module.exports = {
    interfacesDir,
    bindingsOutDir,
    includeDirs,
    configList
```

```
};
```

为项目生成自动绑定文件

```
cd /Users/james/NewProject/tools/swig-config
node < 引擎根目录 >/native/tools/swig-config/genbindings.js -c swig-config.js
```

如果成功，包含自动绑定代码的 `jsb_my_module_auto.cpp/.h` 两个文件将被创建到 `/Users/james/NewProject/native/engine/bindings/auto` 目录下。

修改项目的 CMakeLists.txt 文件

- 打开 `/Users/james/NewProject/native/engine/common/CMakeLists.txt`, 添加 `MyObject.h` 和自动绑定代码文件

```
include(${COCOS_X_PATH}/CMakeLists.txt)

list(APPEND CC_COMMON_SOURCES
    ${CMAKE_CURRENT_LIST_DIR}/Classes/Game.h
    ${CMAKE_CURRENT_LIST_DIR}/Classes/Game.cpp
    ############### 添加下面几行 ###############
    ${CMAKE_CURRENT_LIST_DIR}/Classes/MyObject.h
    ${CMAKE_CURRENT_LIST_DIR}/bindings/auto/jsb_my_module_auto.h
    ${CMAKE_CURRENT_LIST_DIR}/bindings/auto/jsb_my_module_auto.cpp
    ########################################################
)
```

- 修改 `/Users/james/NewProject/native/engine/mac/CMakeLists.txt`

```
cmake_minimum_required(VERSION 3.8)
# ......
cc_mac_before_target(${EXECUTABLE_NAME})
add_executable(${EXECUTABLE_NAME} ${CC_ALL_SOURCES})
############### 添加下面几行 ##############
target_include_directories(${EXECUTABLE_NAME} PRIVATE
    ${CC_PROJECT_DIR}/../common
)
########################################################
cc_mac_after_target(${EXECUTABLE_NAME})
```

打开项目工程

macOS: `/Users/james/NewProject/build/mac/proj/NewProject.xcodeproj`

Windows: `< 一个存放项目的目录 >/NewProject/build/win64/proj/NewProject.sln`

为脚本引擎注册新的模块

修改 `Game.cpp`

```
#include "Game.h"
#include "bindings/auto/jsb_my_module_auto.h" // 添加此行
//......
int Game::init() {
  // ......
  se::ScriptEngine::getInstance()->addRegisterCallback(register_all_my_module); // 添加此行
  BaseGame::init();
  return 0;
}
// ......
```

测试绑定

- 在项目的根目录下添加一个 `my-module.d.ts` 文件，使 TS 编译器识别我们的绑定类型

```
// my-module.d.ts
declare namespace my_ns {
class MyObject {
    constructor();
    constructor(a: number, b: number);

    publicFloatProperty : number;
    print() : void;
}
}
```

- 修改 `/Users/james/NewProject/temp/tsconfig.cocos.json` 文件

```
{
  "$schema": "https://json.schemastore.org/tsconfig",
  "compilerOptions": {
    "target": "ES2015",
    "module": "ES2015",
    "strict": true,
    "types": [
      "./temp/declarations/cc.custom-macro",
      "./temp/declarations/jsb",
      "./temp/declarations/cc",
      "./temp/declarations/cc.env",
      "./my-module" // 添加这行
    ],
    // ......
    "forceConsistentCasingInFileNames": true
  }
}
```

- 在 Cocos Creator 中打开 NewProject 项目, 在场景中添加一个立方体，并且添加一个脚本组件到这个立方体上，脚本的内容是：

```
import { _decorator, Component } from 'cc';
const { ccclass } = _decorator;

@ccclass('MyComponent')
export class MyComponent extends Component {
    start() {
        const myObj = new my_ns.MyObject();
        myObj.print(); // 调用原生的 print 方法
        console.log(`==> myObj.publicFloatProperty: ${myObj.publicFloatProperty}`); // 获取原生中定义的属性值
    }
}
```

- 在 Xcode 或者 Visual Studio 中运行项目, 如果成功，可以看到如下日志输出

```
17:31:44 [DEBUG]: ==> a: 100, b: 1
17:31:44 [DEBUG]: D/ JS: ==> myObj.publicFloatProperty: 1.2300000190734863
```

本节总结

在此节中，我们学会了如何使用 Swig 工具绑定一个简单的 C++ 类，并把它的公有方法与属性导出到 JS 中。从下一节开始，我们将更多地关注如何使用 Swig 的一些特性来满足各式各样的 JS 绑定需求，例如：

- 如何使用 %import 指令导入头文件依赖？
- 如何忽略绑定某些特殊的类、方法或属性？
- 如何重命名类、方法或属性？

- 如何将 C++ 的 getter 和 setter 函数绑定为 JS 属性？
- 如何配置 C++ 模块宏

导入头文件依赖

假定我们让 MyObject 类继承于 MyRef 类。但是我们并不想绑定 MyRef 类型。

```
// MyRef.h
#pragma once
namespace my_ns {
class MyRef  {
public:
    MyRef() = default;
    virtual ~MyRef() = default;
    void addRef() { _ref++; }
    void release() { --_ref; }
private:
    unsigned int _ref{0};
};
} // namespace my_ns {
```

```
// MyObject.h
#pragma once
#include "cocos/cocos.h"
#include "MyRef.h"
namespace my_ns {
// MyObject 继承于 MyRef
class MyObject : public MyRef {
public:
    MyObject() = default;
    MyObject(int a, bool b) {}
    virtual ~MyObject() = default;
    void print() {
        CC_LOG_DEBUG("==> a: %d, b: %d\n", _a, (int)_b);
    }

    float publicFloatProperty{1.23F};
private:
    int _a{100};
    bool _b{true};
};
} // namespace my_ns {
```

当 Swig 解析 MyObject.h 的时候, 它并不知道 `MyRef` 是什么, 因此它会在终端输出一个警告信息。

```
.../Classes/MyObject.h:7: Warning 401: Nothing known about base class 'MyRef'. Ignored.
```

要解决此警告也容易，我们只需要使用 `%import` 指令让 Swig 知道 MyRef 类的存在即可。

```
// ......
// Insert code at the beginning of generated source file (.cpp)
%{
#include "bindings/auto/jsb_my_module_auto.h"
%}

%import "MyRef.h" // 添加此行
%include "MyObject.h"
```

尽管 Swig 不再报错了，但是生成的代码却无法编译通过，会出现如下报错：

CMake 使用简介

CMake 使用简介

CMake 介绍

CMake 是一个非常强大的构建工具，可以大大简化软件的编译过程，提高开发效率。Cocos Creator 在各个原生平台都使用了 CMake。以下是一些快速了解 CMake 的优点：

- 使用 CMakeLists.txt 文件描述整个项目的构建过程，而不是像其他构建工具一样使用脚本文件。
- 是跨平台的，可以在 Windows、Linux、macOS 等操作系统上运行。
- 可以自动生成 Makefile、Visual Studio 等 IDE 的工程文件，从而简化了软件的编译过程。
- 可以轻松的管理依赖库，将代码组织成模块等。
- 支持多种编程语言，包括 C、C++、Fortran、Java、Python 等。

虽然 CMake 是一个非常强大的构建工具，但是它也有一些缺点，比如语法比较复杂，需要一定的学习成本。

开发者可以学习 CMake 的语法并添加自己的模块，以便在构建过程中执行特定的任务。例如，他们可以定义自己的预处理器宏或编译器选项，以便在构建期间执行自定义操作。另外，他们还可以编写脚本来修改工程文件，以便在不同的平台上进行不同的配置。详情可参考 二次开发。

如果您想深入学习 CMake，可以阅读本教程的其他章节，了解如何安装 CMake、创建 CMakeLists.txt 文件、指定编译器、添加源文件、添加依赖库和构建项目等。由于篇幅限制，本文无法对其进行详细介绍，例如如何使用 FindPackage、如何设置构建类型、如何安装和测试等内容。为了更好地掌握它，开发者需要查阅其他文档并进行更多的实践。

安装

为方便开发者，Cocos Creator 内部集成了 cmake 程序，构建流程会使用它来完成。因此，一般情况下开发者不需要手动安装 cmake。

如果开发者希望编辑器使用设备上的 cmake，则可以通过编辑相关的配置完成。

如果开发者想要在命令行中使用 cmake，可以前往官网下载。在 Mac 平台上，也可以使用 Homebrew 进行安装，执行以下命令进行安装：

```
brew install cmake
```

快速开始

`CMakeLists.txt` 文件是 CMake 的核心文件，用于描述整个项目的构建过程。使用该文件可以方便地管理项目的构建和编译过程。其中包含了一系列命令和变量，用于指定项目名称、版本号、源文件、依赖库等信息，以及指定编译器、编译选项等参数。

下面是一个简单的 CMake helloworld工程的例子。

首先，创建一个名为 `CMakeLists.txt` 的文件。在此文件中，添加以下内容：

```
# CMake 版本
cmake_minimum_required(VERSION 3.10)

# 项目名称，指定语言为 C++
project(helloworld CXX)

# 可执行文件
add_executable(helloworld main.cpp)
```

然后，在项目的根目录下创建一个名为 `main.cpp` 的文件，并添加以下内容：

```
#include <iostream>

int main() {
    std::cout << "Hello, world!" << std::endl;
    return 0;
```

```
}
```

最后，在项目的根目录下创建一个名为 `build` 的目录，并在其中执行以下命令：

```
# 在 build 目录下生成默认的工程文件。如果已经安装了 Visual Studio，则默认为 Visual Studio 工程；在 Mac 下默认为 Makefile 工程。通过指定 -G 可以设置工程文件的类型，比如 -GXcode。
cmake -B build -S .
# 生成可执行文件
cmake --build build
```

执行完这些命令后，将在 `build` 目录中生成可执行文件 `helloworld`。运行该文件，将输出 "Hello, world!"。

这里用到的两个命令 project 和 add_executable

`project` 是 CMake 中的一个命令，用于指定项目名称、版本号、语言等信息，其语法如下：

```
project(project_name [version] [LANGUAGES languages...])
```

其中，`project_name` 用于指定项目的名称，`version` 用于指定项目的版本号，`languages` 用于指定项目所使用的编程语言。如果不指定 `version` 或 `languages` 参数，则可以省略它们。例如：

```
project(MyProject)
```

这个命令将设置项目名称为 `MyProject`，不指定版本号和编程语言。

`add_executable` 用于添加可执行文件的构建规则，其语法如下：

```
add_executable(executable_name [source1] [source2] ...)
```

其中，`executable_name` 用于指定可执行文件的名称，`source1`、`source2` 等参数用于指定源文件的名称。例如：

```
add_executable(MyProject main.cpp)
```

这个命令将设置可执行文件名称为 `MyProject`，并将 `main.cpp` 文件作为源文件添加到项目中。

其他常用 CMake 命令

message

`message()` 命令用于在 CMake 运行时向用户显示消息。它接受一个或多个参数，作为要显示的消息。例如：

```
message("Hello, world!")
```

这个命令将在 CMake 运行时向用户显示 "Hello, world!" 消息。

你也可以在 `message()` 命令中使用变量和表达式。例如：

```
set(SRC_FILES main.cpp)
message("Source files: ${SRC_FILES}")
```

这个命令将在 CMake 运行时向用户显示 "Source files: main.cpp" 消息。

`message()` 命令还可以用于输出调试信息。例如：

```
if (DEBUG)
  message("Debug mode enabled")
endif ()
```

这个命令将在 CMake 运行时检查变量 `DEBUG` 是否为真，如果为真，则向用户输出 "Debug mode enabled" 消息。

`message` 命令还有其他用途，例如：

- 输出警告信息：`message(WARNING "This is a warning message")`
- 输出错误信息：`message(FATAL_ERROR "This is an error message")`
- 输出调试信息：`message(STATUS "This is a status message")`

set

`set()` 命令主要用于创建或修改变量。该命令至少接受两个参数：变量名和值。例如，你可以使用 `set(SRC_FILES main.cpp)` 来设置变量 `SRC_FILES` 的值为 `main.cpp`。如果你想要为变量设置多个值（比如列表），你可以在命令中添加更多参数，如 `set(SRC_FILES main.cpp util.cpp)`。如果你想要读取变量的值，可以使用 `${}` 语法，如 `message(${SRC_FILES})`。

可以使用 `set` 命令向列表变量中添加元素。具体来说，可以使用 `set(SRC_FILES ${SRC_FILES} util.cpp)` 命令将 `util.cpp` 添加到 `SRC_FILES` 列表的末尾。其中，`${SRC_FILES}` 表示取出 `SRC_FILES` 变量的当前值。这个命令还可以使用其他的 `set` 命令选项，如 `CACHE` 和 `APPEND` 等。

list

`list()` 命令用于处理列表类型的变量。它可以接受多种子命令，如 `APPEND`（在列表尾部添加元素）、`INSERT`（在指定位置插入元素）、`REMOVE_ITEM`（删除指定的元素）等。例如，`list(APPEND SRC_FILES util.cpp)` 命令会将 `util.cpp` 添加到 `SRC_FILES` 列表的末尾。

add_library

`add_library` 命令用于定义一个库目标。它至少需要两个参数：库的名称和源文件。如果你只提供了一个源文件，那么 CMake 将创建一个由这个源文件构建的库。例如，`add_library(MyLib main.cpp)`。如果你有多个源文件，你可以将它们全部放到 `add_library()` 命令中，例如 `add_library(MyLib main.cpp util.cpp)`。

CMake支持创建静态库和动态库。默认情况下，`add_library()` 命令会创建一个静态库。如果你想要创建一个动态库，你需要在命令中添加 `SHARED` 参数，例如：`add_library(MyLib SHARED main.cpp)`。

如果你想要同时创建静态库和动态库，你可以将它们都列出来，例如：`add_library(MyLibStatic STATIC main.cpp)` 和 `add_library(MyLibShared SHARED main.cpp)`。

静态库是指在编译时链接到可执行文件中的库，而动态库是指在运行时加载的库。静态库通常只包含可执行文件所需的代码，因此它们比较小。动态库通常包含更多的代码和数据，因为它们需要在运行时执行。动态库的优点是可以在不重新编译可执行文件的情况下更新库。它们也可以在多个可执行文件之间共享，从而节省磁盘空间。

使用 `add_library()` 命令时，你可以指定库的名称和类型（静态库或动态库），以及要包含的源文件和头文件。例如，`add_library(MyLib STATIC main.cpp)` 命令将 `main.cpp` 源文件添加到名为 `MyLib` 的静态库中。

find_library

命令用于查找并定位系统上的库文件。你需要提供一个变量名（用于存储找到的库的路径）和库的名称。例如，`find_library(MY_LIB NAMES MyLib)`。在这个例子中，CMake 会在系统的库路径中搜索名为 `MyLib` 的库。如果找到了，`MY_LIB` 变量的值将会被设置为该库的全路径。

可以使用 `find_library` 命令来查找系统上的库文件。你需要提供一个变量名（用于存储找到的库的路径）和库的名称。

例如，假设你想要查找名为 `libexample` 的库，它在 `/usr/local/lib` 目录中。 这里的绝对路径可以使用`${CMAKE_CURRENT_LIST_DIR}`等变量转化为相对路径。你可以在 `CMakeLists.txt` 文件中添加以下命令：

```
find_library(EXAMPLE_LIB libexample /usr/local/lib)
```

这个命令会将 `libexample` 库的路径保存到变量 `EXAMPLE_LIB` 中。如果找不到该库，`EXAMPLE_LIB` 变量的值将会是空的。

在使用 `find_library` 命令时，你可以指定库的名称、路径、版本和语言。例如，`find_library(EXAMPLE_LIB NAMES example PATHS /usr/local/lib VERSION 1.0 LANGUAGES CXX)` 命令将查找名为 `example`、版本为 `1.0`、语言为 `C++` 的库，并将其路径保存到 `EXAMPLE_LIB` 变量中。

如果你想要查找多个库，可以在命令中添加多个库名称。例如，`find_library(LIB1 NAMES lib1 lib1.a PATHS /usr/local/lib)` 命令将查找名为 `lib1` 或 `lib1.a` 的库，并将其路径保存到 `LIB1` 变量中。

注意，在使用 `find_library` 命令时，你需要确保库文件的名称、路径、版本和语言与你的项目相匹配。否则，你的项目可能无法正确地链接到库文件。

target_link_libraries

`target_link_libraries()` 命令用于将指定的库链接到目标。这个命令至少需要两个参数：目标名称和库名称。例如，`target_link_libraries(MyApp MyLib)`。这个命令将 `MyLib` 库链接到 `MyApp` 目标。

这意味着 MyApp 在构建时会使用 MyLib。

target_include_directories

target_include_directories() 命令用于为指定的目标添加包含目录。这个命令需要至少两个参数：目标名称和要添加的目录。例如，target_include_directories(MyApp PRIVATE include/)。这个命令将 include/ 目录添加到 MyApp 目标的包含目录中。这意味着编译 MyApp 时，编译器会在 include/ 目录中查找头文件。

target_compile_options

```
message(STATUS "my custom debug info")
```

target_compile_options() 命令用于为指定的目标设置编译选项。这个命令至少需要两个参数：目标名称和编译选项。例如，target_compile_options(MyApp PRIVATE -Wall)。这个命令将 -Wall 选项添加到 MyApp 的编译选项中。这意味着 MyApp 在编译时会启用所有的警告（这是 -Wall 选项的作用）。

常见的任务

添加源文件

可以使用以下命令添加源文件：

```
add_executable(MyProject main.cpp math/vec3.cpp math/vec4.cpp)
```

在这个示例中，MyProject 的源文件包括 **main.cpp**、 math/vec3.cpp 和 math/vec4.cpp . 如果有更多的源文件,只需要往这个列表添加它们.

添加依赖库

可以使用以下命令添加依赖库：

```
target_link_libraries(MyProject MyLibrary)
```

下面的例子中, find_library() 命令将在 libs 目录下查找名为 libexample 的静态库, 并将其路径保存到变量 LIBS 中. target_link_libraries() 命令将这个库链接到 MyProject 目标.

```
find_library(LIBS libexample libs PATHS ${CMAKE_CURRENT_LIST_DIR}/libs/android/${ANDROID_ABI})

add_executable(MyProject main.cpp)
target_link_libraries(MyProject ${LIBS})

# 添加头文件的搜索路径
target_include_directories(MyProject  PUBLIC ${CMAKE_CURRENT_LIST_DIR}/libs/include)
```

CMake变量

CMake内置了一些以CMAKE_开头的变量，方便与环境交互。这些变量的使用可以使你的CMakeLists.txt文件更加简洁、易于维护。比如，CMAKE_CURRENT_LIST_DIR 变量用于存储当前处理的CMakeLists.txt文件所在的目录的路径。在CMakeLists.txt文件中使用此变量的示例如下：

```
add_library(MyLibrary STATIC ${CMAKE_CURRENT_LIST_DIR}/src/my_library.cpp)
```

上述示例中，我们使用CMAKE_CURRENT_LIST_DIR变量指定源文件的路径。同样，CMAKE_BINARY_DIR变量用于存储二进制文件的根目录的路径。在CMakeLists.txt文件中使用此变量的示例如下：

```
set(EXECUTABLE_OUTPUT_PATH ${CMAKE_BINARY_DIR}/bin)
```

这里，CMAKE_BINARY_DIR变量用于指定可执行文件输出的根目录。在编译项目时，可执行文件将被输出到${CMAKE_BINARY_DIR}/bin目录中。

请注意，${CMAKE_BINARY_DIR}和${CMAKE_CURRENT_BINARY_DIR}变量之间的区别。${CMAKE_BINARY_DIR}是指二进制文件的根目录，而${CMAKE_CURRENT_BINARY_DIR}是指当前处理的CMakeLists.txt文件的二进制目录。

此外，其他常用的变量包括但不限于：

- CMAKE_SOURCE_DIR: CMakeLists.txt文件所在的目录
- CMAKE_CURRENT_SOURCE_DIR: 当前处理的CMakeLists.txt所在的目录
- CMAKE_BINARY_DIR: 二进制文件的根目录
- CMAKE_CURRENT_BINARY_DIR: 当前处理的CMakeLists.txt的二进制目录
- CMAKE_INSTALL_PREFIX: 安装目录的根目录
- CMAKE_MODULE_PATH: CMake模块的根目录
- CMAKE_BUILD_TYPE: 编译类型
- CMAKE_CXX_FLAGS: C++编译器选项

在 Cocos 中使用 CMake

Android 在编译 C++ 代码使用了 cmake，这是原生支持的. 我们会通过 gradle 去配置参数和调用 cmake 命名生成/编译/打包 C++ 代码。对于其他的原生平台，我们会通过构建插件调用对于的 cmake 命令去生成工程文件。在Windows 上的 Visual Studio 工程，Mac 上的 Xcode 工程. 后续的开发就只需通过 IDE 去完成.

由于 CMake 的特性，可能会因为不同的开发环境和配置而产生差异，因此不建议共享生成的工程文件。另外，对生成工程的修改很容易被后续生成的工程覆盖。相反，应该将 CMakeLists.txt 文件包含在项目中，并在每个开发环境中使用 CMake 来生成相应的工程文件。所有对工程的修改都应该以 CMake 指令的方式写入到 CMakeLists.txt 中。

在使用 CMake 和 Xcode cocoapods 时，可能会出现一些问题。主要的问题是，CMake 生成的 Xcode 工程文件与 cocoapods 集成不兼容。这是因为 cocoapods 使用了自己的方式来管理 Xcode 项目文件，而 CMake 生成的工程文件没有考虑这一点。这可能会导致一些问题，如编译错误、链接错误, 修改被覆盖等等。

为了解决这个问题，我们可以在构建 Mac/iOS 平台时勾选 “跳过 Xcode 工程更新” 选项。这样做的意思是，后续引擎或工程的 CMake 配置更新不会同步到 Xcode 工程。勾选该选项后，就可以与 CocoaPods 协作，以普通的 Xcode 工程形式进行构建。

共享生成的 Xcode 工程文件

CMake 生成的 Xcode 工程记录了依赖路径，这些路径来自于以下几个方面：

- Xcode 的安装路径
- Cocos Creator 版本和安装路径
- 工程的版本和安装路径
- 工程文件所在的路径

通过在不同设备上使用相同的目录结构，Xcode 可以实现不同设备间的共享。

另一个做法是修改 Xcode 工程文件内部引用的路径，这个方法比较 hack，这里不做详细介绍。

和 Xcode 不同，Android Studio 使用 CMake 是直接将 CMakeLists.txt 作为配置文件进行编译，而不是生成工程文件。因此，CMake 生成的 Android 平台的 native 库在不同设备上的表现应该是一致的。此外，Android Studio 的 Gradle 插件会自动处理依赖关系，因此不需要像 Xcode 那样手动管理依赖目录。

目录结构

当选择某个原生平台进行构建时，项目目录 native\engine 目录下会生成 当前构建的平台名称 文件夹（例如 android），以及 common 文件夹。CMake 在第一次运行时将会在这两个目录下分别生成 CMakeLists.txt 文件，作用各不相同：

- 当前构建的平台名称 文件夹：CMakeLists.txt 主要用于配置对应的构建平台。以 Android 平台为例：

原生引擎内存泄漏检测系统

原生引擎内存泄漏检测系统

原生引擎使用 C++ 语言开发，为了方便游戏和引擎开发者快速查找内存泄漏，Creator 从 v3.4 开始提供了 **内存泄漏检测系统**。

相对其他内存泄漏查找工具，Cocos Creator 内置的内存泄漏检测工具有以下优点：

- **跨平台**：支持 Windows/Android/Mac/iOS 平台。
- **易用性**：无需下载额外的工具以及进行复杂的配置。支持输出内存泄漏处的堆栈信息，方便快速定位泄漏。
- **一致性**：各平台的使用流程几乎一致，都是从原生平台对应的 IDE 中启动游戏 -> 运行一段时间 -> 关闭游戏 -> 查看 IDE 输出日志。
- **实时性**：查找泄露过程中的游戏帧率虽有下降，但仍然保持实时运行帧率。
- **精确性**：理论上零漏报，零误报。

使用步骤

1. 内存泄漏检测系统默认是关闭的。若要开启，需要将引擎目录下 `engine/native/cocos/base/Config.h` 文件中宏 `USE_MEMORY_LEAK_DETECTOR` 的值修改为 **1**。

   ```
   #ifndef USE_MEMORY_LEAK_DETECTOR
       #define USE_MEMORY_LEAK_DETECTOR 1
   #endif
   ```

2. 由于实现机制的不同，Android 平台上需要额外执行以下步骤：

   在项目目录的 `native/engine/android/CMakeLists.txt` 文件中添加一行代码 `set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -finstrument-functions")`，如下所示：

   ```
   set(CC_PROJ_SOURCES)
   set(CC_COMMON_SOURCES)
   set(CC_ALL_SOURCES)

   set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -finstrument-functions")
   ```

3. 从原生平台对应的 IDE（如 Visual Studio、Android Studio、Xcode）启动游戏，运行一段时间后关闭游戏，若存在内存泄漏，则此时会在 IDE 的输出窗口中输出内存泄漏的详细信息。

   - **Windows 平台**

原生场景剔除

原生场景剔除

Creator 从 v3.4.0 开始支持原生场景剔除，包括 **八叉树场景剔除** 和 **遮挡查询剔除**，仅对原生平台生效。

八叉树场景剔除

一般情况下，引擎剔除不在视锥（摄像机的可见范围）内的模型，是通过逐个检测模型的包围盒是否在视锥内，速度较慢。若开启八叉树场景剔除，则通过八叉树可以快速剔除不在视锥内的模型。

该功能默认关闭，若需要开启，在 **层级管理器** 中选中场景根节点 **Scene**，然后在 **属性检查器** 中便可看到 **Octree Scene Culling** 项，勾选 **Enabled** 即可：

原生性能剖析器

性能剖析器（Profiler）

性能剖析器是用于性能分析和统计的工具，目前仅限原生平台。

默认统计数据

- 性能剖析器的效果展示如图：

原生插件

原生插件

> 注意：对 3.6.2 的原生插件支持有问题，请升级到 3.6.3 或更高版本。

原生插件是编辑器插件的一部分。开发者通过原生插件调用脚本绑定接口（如 sebind）可以扩展 JS 脚本调用 C++ 的接口的能力，对解决脚本的性能瓶颈和复用现有代码库都非常有利。

和现有插件系统的关系

原生插件能独立于编辑器插件存在，用户通过拷贝到指定目录就可以使用原生插件。

同时，原生插件也作为现有编辑器插件系统的补充，扩展游戏运行时的能力。利用编辑插件的能力实现对原生插件管理， 如：下载/开关/版本升级等功能。

插件的结构

每个插件的根目录下都有一个插件的描述文件 `cc_plugin.json`，这是一个标准的 JSON 文件。

在构建原生工程的时候，构建系统会从工程的 `extensions` 和 `native` 目录中去递归查找这所有的 `cc_plugin.json` 文件，以定位原生插件。一旦在目录中找到 `cc_plugins.json`，就不会再查找子目录中的内容。

安装依赖

在少数未安装编辑器的环境下，需要安装 NodeJS 8.0 以上的版本，以支持插件配置解析。开发者可以将 NodeJS 并设置环境变量 `PATH`，也可以在 `CMakeLists.txt` 中通过设置 `NODE_EXECUTABLE` 指定。

也可以设置环境变量 `NODE_EXECUTABLE` 为 node 的完整路径。3.6.2 开始，如果 CMake 仍然定位不到 nodejs，可以在 `native/engine/common/localCfg.cmake` 中直接设置 `NODE_EXECUTABLE`。

目录结构示例

```
├── cc_plugin.json
├── android
│   ├── arm64-v8a
│   ├── armeabi-v7a
│   ├── x86
│   └── x86_64
├── ios
│   ├── include
│   └── lib
├── mac
│   ├── include
│   └── lib
└── windows
    ├── include
    └── lib
```

文件 `cc_plugin.json` 是提供了加载插件所必须的信息，是原生插件的标识。每一个支持的原生平台对应一个目录，目录中至少包含一个 `<PackageName>-Config.cmake` 文件。构建系统会子使用 CMake 的 `find_package` 机制定位或链接到所需的库文件。

如果插件中存在可跨平台的源文件或 CMake 配置, 可以将这些文件合并到顶层目录。详情请参考 示例工程。

描述文件 `cc_plugin.json` 格式

```
{
    "name": string;             // 必填：插件名称
    "version": string;          // 必填：插件版本
    "engine-version":string;    // 必填：对应引擎版本的区间
    "author": string;           // 必填：插件作者
    "description": string;      // 必填：插件描述
    "platforms":string[];       // 可选：支持的平台列表，不填默认支持所有原生平台。包括 windows, android, mac, ios
    "disabled":true;         // 可选：禁用插件
    "disable-by-platforms":string[];    //可选：指定平台禁用插件
    "modules": [{                        // 必填：插件包含的库，
        "target":string;                // 必填：对应 `find_package` 名称，需和 `CC_PLUGIN_ENTRY` 的首参数保持一致
        "depends": string|string[];     // 可选：依赖其他 module 名称
        "platforms":string[];           // 可选：重新限定支持的原生平台
    }]

}
```

`engine-version` 可以指定版区间和排除指定版本，代码示例如下：

```
"engine-version": ">=3.3 <= 3.6.0 !3.5.2|| 4.x"
```

文件示例

```
{
    "name":"hello-cocos-demo",
    "version":"0.1.0",
    "author":"demo group",
    "engine-version":">=3.6.0",
    "description": "demo project",
    "modules":[
        {
            "target":"hello_cocos_glue"
        }
    ],
    "platforms":["windows", "android", "mac", "iOS"]
}
```

安装原生插件

开发者可以从 Store 下载并启用包含原生插件的编辑器扩展，同时完成插件的安装到 `extensions` 目录。也可以从论坛获取插件压缩包，手动解压到 `native/` 目录或其子目录。

如果想关闭插件，或者仅在特地平台关闭，可以修改 cc_plugin.json 中的 `disabled` 和 `disable-by-platforms` 字段。

> **注意**：原生插件要求 CMake 为 3.12+，Android 需要 指定 CMake 版本 为 3.18.1. 其他平台使用编辑器内建的 CMake，可不必指定版本号。

创建原生插件

Cocos 原生工程使用 CMake 管理，原生插件会通过 find_package 的搜索路径/目录来进行管理，因此只要目录符合 CMake find_package 的搜索规则，插件就能正确加载。所以，原生插件的开发过程，就是提供 CMake 配置和相关的资源，以及编写 cc_plugin.json 的过程。相关示例请参考 原生插件构建工程示例。

原生插件创建范例

原生插件创建范例

如果想在原生项目中使用第三方原生库，则可以按照本文的步骤进行。

本文需要对原生工程的编译生成有一定了解，开发者可以通过 CMake 官网 了解。 我们也准备了范例工程 GitHub 以供参考。

创建原生插件

插件开发工程 Windows 配置

示例中，我们将引入 hello_cocos.lib 作为 windows 平台上的插件，引入引擎并使其支持在 TS/JS 中使用。其他平台将使用 hello_cocos.a 为例，如果要使用其他库，请提前编译到对应平台。

- 使用 Cocos Creator 3.6+ 创建一个工程

  启动 CocosCreator，在指定目录执行 `创建空工程`。

原生引擎跨语言调用优化

原生引擎跨语言调用优化

前言

在 Cocos Creator 3.6.0 版本的原生实现上，我们提高了原生（CPP）层级，主要体现在节点树（Scene、Node）、资产（Asset 及其子类）、材质系统（Material、Pass、ProgramLib）、3D 渲染器 （包含 Model、SubModel）、2D 渲染器（Batch2D、RenderEntity）都在 CPP 中实现，并通过绑定技术暴露给 JS 层使用。

众所周知，在 Cocos Creator 3.x 中，开发者只能使用 TS 脚本开发游戏业务逻辑。尽管我们将更多的引擎代码在 CPP 中实现，但是开发者并无法直接使用 CPP 语言进行游戏业务逻辑的开发，因此我们通过`脚本引擎封装层(Script Engine Wrapper)`的 `API`，简称 `SE API`，将这些 CPP 层实现的类型绑定并暴露给 JS 中，暴露到 JS 中的接口保持跟 Web 环境中一致。

原生层级的上移最直接的好处就是：在原生平台上引擎代码的执行性能得到提升，特别是不支持 JIT 的平台（比如：iOS）尤为明显。但在 3.6.0 正式版发布前，我们也面临了一系列由于`原生层级提升`带来的副作用，最主要的方面就是 JSB 调用（JS <-> CPP 语言之间的交互）次数比之前版本多了不少，而这直接导致`原生层级提升`带来的收益被抵消甚至性能相比之前版本（ 3.5 ）变得更差。本文将介绍一些降低 `JSB 调用`的优化方式，若开发者自己的 CPP 代码中也存在类似的 JSB 调用过多的问题，希望本文也能提供一些优化思路。

共享内存

对 Node 中需要频繁同步的属性，我们使用 CPP 与 JS 共享内存的方式，避免 JSB 调用。为了更方便地共享 CPP 的内存到 JS 中，我们封装了 bindings::NativeMemorySharedToScriptActor 辅助类。

bindings/utils/BindingUtils.h

```
namespace cc::bindings {

class NativeMemorySharedToScriptActor final {
public:
    NativeMemorySharedToScriptActor() = default;
    ~NativeMemorySharedToScriptActor();

    void initialize(void *ptr, uint32_t byteLength);
    void destroy();

    inline se::Object *getSharedArrayBufferObject() const { return _sharedArrayBufferObject; }

private:
    se::Object *_sharedArrayBufferObject{nullptr};

    CC_DISALLOW_COPY_MOVE_ASSIGN(NativeMemorySharedToScriptActor)
};

} // namespace cc::bindings
```

bindings/utils/BindingUtils.h

```
#include "bindings/utils/BindingUtils.h"
#include "bindings/jswrapper/SeApi.h"

namespace cc::bindings {

NativeMemorySharedToScriptActor::~NativeMemorySharedToScriptActor() {
    destroy();
}

void NativeMemorySharedToScriptActor::initialize(void* ptr, uint32_t byteLength) {
    CC_ASSERT_NULL(_sharedArrayBufferObject);
    // The callback of freeing buffer is empty since the memory is managed in native,
    // the external array buffer just holds a reference to the memory.
    _sharedArrayBufferObject = se::Object::createExternalArrayBufferObject(ptr, byteLength, [](void* /*contents*/, size_t /*byteLength*/, void* /*userData*/) {});
    // Root 此对象，防止对象被 GC，当 Actor 对象销毁的时候会调用 destroy 函数，其内部会 unroot ArrayBuffer 对象
    _sharedArrayBufferObject->root();
}

void NativeMemorySharedToScriptActor::destroy() {
    if (_sharedArrayBufferObject != nullptr) {
        _sharedArrayBufferObject->unroot();
        _sharedArrayBufferObject->decRef();
        _sharedArrayBufferObject = nullptr;
    }
}

} // namespace cc::bindings
```

bindings/jswrapper/v8/Object.h

```
using BufferContentsFreeFunc = void (*)(void *contents, size_t byteLength, void *userData);
static Object *createExternalArrayBufferObject(void *contents, size_t byteLength, BufferContentsFreeFunc freeFunc, void *freeUserData = nullptr);
```

bindings/jswrapper/v8/Object.cpp

```
/* static */
Object *Object::createExternalArrayBufferObject(void *contents, size_t byteLength, BufferContentsFreeFunc freeFunc, void *freeUserData /* = nullptr*/) {
    Object *obj = nullptr;
    std::shared_ptr<v8::BackingStore> backingStore = v8::ArrayBuffer::NewBackingStore(contents, byteLength, freeFunc, freeUserData);
    v8::Local<v8::ArrayBuffer> jsobj = v8::ArrayBuffer::New(__isolate, backingStore);

    if (!jsobj.IsEmpty()) {
        obj = Object::_createJSObject(nullptr, jsobj);
    }
    return obj;
}
```

分析以上代码可知，其实 NativeMemorySharedToScriptActor 最终是调用 v8::ArrayBuffer::NewBackingStore 和 v8::ArrayBuffer::New 函数创建了一个 External 类型的 ArrayBuffer，取名为 External 的原因是其内存不在 V8 内部分配和管理，即其内存完全交由 V8 的上层管理，当 ArrayBuffer 对象被 GC 的后，freeFunc 回调函数会被触发。由于在 Node 中需要共享的内存是 Node 中的若干连续属性，内存的创建和释放完全交由 Node 自己维护，而 CPP Node 实例的销毁也是由 GC 控制的，因此 NativeMemorySharedToScriptActor::initialize 内部调用 se::Object::createExternalArrayBufferObject 的时候，传递了一个**空实现**的回调函数。

Node.h

```
class Node : public CCObject {
    ......
    inline se::Object *_getSharedArrayBufferObject() const { return _sharedMemoryActor.getSharedArrayBufferObject(); } // NOLINT
    ......
    bindings::NativeMemorySharedToScriptActor _sharedMemoryActor;
    ......
        // Shared memory with JS
    // NOTE: TypeArray created in node.jsb.ts _ctor should have the same memory layout
    uint32_t _eventMask{0};                                     // Uint32: 0
    uint32_t _layer{static_cast<uint32_t>(Layers::LayerList::DEFAULT)}; // Uint32: 1
    uint32_t _transformFlags{0};                                // Uint32: 2
    index_t _siblingIndex{0};                                   // Int32: 0
    uint8_t _activeInHierarchy{0};                              // Uint8: 0
    uint8_t _active{1};                                         // Uint8: 1
    uint8_t _isStatic{0};                                       // Uint8: 2
    uint8_t _padding{0};                                        // Uint8: 3
    ......
};
```

Node.cpp

```
Node::Node(const ccstd::string &name) {
#define NODE_SHARED_MEMORY_BYTE_LENGTH (20)
    static_assert(offsetof(Node, _padding) + sizeof(_padding) - offsetof(Node, _eventMask) == NODE_SHARED_MEMORY_BYTE_LENGTH, "Wrong shared memory size");
    _sharedMemoryActor.initialize(&_eventMask, NODE_SHARED_MEMORY_BYTE_LENGTH);
#undef NODE_SHARED_MEMORY_BYTE_LENGTH

    _id = idGenerator.getNewId();
    if (name.empty()) {
        _name.append("New Node");
    } else {
        _name = name;
    }
}
```

Node 构造函数中调用 `_sharedMemoryActor.initialize(&_eventMask, NODE_SHARED_MEMORY_BYTE_LENGTH);` 将 _eventMask 属性开始的 20 字节设置为共享内存。

node.jsb.ts

> **注意**：所有的 .jsb.ts 结尾的文件最终在打包的时候会替换对应不带 .jsb 的文件，比如这里 node.jsb.ts 会替换掉 node.ts，具体可以查看引擎根目录下的 cc.config.json 文件，有对应的 overrides 字段：*"cocos/scene-graph/node.ts"*:"cocos/scene-graph/node.jsb.ts",

```
// JS 的 _ctor 回调函数会在 JS 的 var node = new Node(); 流程的最后阶段被触发，即 CPP Node 对象创建之后，因此 _getSharedArrayBufferObject 绑定函数返回的 ArrayBuffer 一定存在。
nodeProto._ctor = function (name?: string) {
    ......
    // 通过 _getSharedArrayBufferObject 绑定方法，取得 CPP 共享给 JS 的 ArrayBuffer 对象
    const sharedArrayBuffer = this._getSharedArrayBufferObject();
    // Uint32Array with 3 elements, offset from the start: eventMask, layer, dirtyFlags
    this._sharedUint32Arr = new Uint32Array(sharedArrayBuffer, 0, 3);
    // Int32Array with 1 element, offset from 12th bytes: siblingIndex
    this._sharedInt32Arr = new Int32Array(sharedArrayBuffer, 12, 1);
    // Uint8Array with 3 elements, offset from 16th bytes: activeInHierarchy, active, static
    this._sharedUint8Arr = new Uint8Array(sharedArrayBuffer, 16, 3);
    //

    this._sharedUint32Arr[1] = Layers.Enum.DEFAULT; // this._sharedUint32Arr[1] is layer
    ......
};
```

采用共享内存的方式，也意味着我们无法通过 JSB 绑定函数的方式把要设置的值从 JS 传递到 CPP 中。因此我们需要在 .jsb.ts 中定义对应的 getter / setter 函数，其内部通过直接操作 TypedArray 的方式修改共享内存。

```
Object.defineProperty(nodeProto, 'activeInHierarchy', {
    configurable: true,
    enumerable: true,
    get (): Readonly<Boolean> {
        return this._sharedUint8Arr[0] != 0; // Uint8, 0: activeInHierarchy
    },
    set (v) {
        this._sharedUint8Arr[0] = (v ? 1 : 0); // Uint8, 0: activeInHierarchy
    },
});

Object.defineProperty(nodeProto, '_activeInHierarchy', {
```

```
    configurable: true,
    enumerable: true,
    get (): Readonly<Boolean> {
        return this._sharedUint8Arr[0] != 0; // Uint8, 0: activeInHierarchy
    },
    set (v) {
        this._sharedUint8Arr[0] = (v ? 1 : 0); // Uint8, 0: activeInHierarchy
    },
});

Object.defineProperty(nodeProto, 'layer', {
    configurable: true,
    enumerable: true,
    get () {
        return this._sharedUint32Arr[1]; // Uint32, 1: layer
    },
    set (v) {
        this._sharedUint32Arr[1] = v; // Uint32, 1: layer
        if (this._uiProps && this._uiProps.uiComp) {
            this._uiProps.uiComp.setNodeDirty();
            this._uiProps.uiComp.markForUpdateRenderData();
        }
        this.emit(NodeEventType.LAYER_CHANGED, v);
    },
});

Object.defineProperty(nodeProto, '_layer', {
    configurable: true,
    enumerable: true,
    get () {
        return this._sharedUint32Arr[1]; // Uint32, 1: layer
    },
    set (v) {
        this._sharedUint32Arr[1] = v; // Uint32, 1: layer
    },
});

Object.defineProperty(nodeProto, '_eventMask', {
    configurable: true,
    enumerable: true,
    get () {
        return this._sharedUint32Arr[0]; // Uint32, 0: eventMask
    },
    set (v) {
        this._sharedUint32Arr[0] = v; // Uint32, 0: eventMask
    },
});

Object.defineProperty(nodeProto, '_siblingIndex', {
    configurable: true,
    enumerable: true,
    get () {
        return this._sharedInt32Arr[0]; // Int32, 0: siblingIndex
    },
    set (v) {
        this._sharedInt32Arr[0] = v; // Int32, 0: siblingIndex
    },
});

nodeProto.getSiblingIndex = function getSiblingIndex() {
    return this._sharedInt32Arr[0]; // Int32, 0: siblingIndex
};

Object.defineProperty(nodeProto, '_transformFlags', {
    configurable: true,
    enumerable: true,
    get () {
        return this._sharedUint32Arr[2]; // Uint32, 2: _transformFlags
    },
    set (v) {
        this._sharedUint32Arr[2] = v; // Uint32, 2: _transformFlags
    },
});

Object.defineProperty(nodeProto, '_active', {
    configurable: true,
    enumerable: true,
    get (): Readonly<Boolean> {
        return this._sharedUint8Arr[1] != 0; // Uint8, 1: active
    },
    set (v) {
        this._sharedUint8Arr[1] = (v ? 1 : 0); // Uint8, 1: active
    },
});

Object.defineProperty(nodeProto, 'active', {
    configurable: true,
    enumerable: true,
    get (): Readonly<Boolean> {
        return this._sharedUint8Arr[1] != 0; // Uint8, 1: active
    },
    set (v) {
        this.setActive(!!v);
    },
});

Object.defineProperty(nodeProto, '_static', {
    configurable: true,
    enumerable: true,
    get (): Readonly<Boolean> {
        return this._sharedUint8Arr[2] != 0;
    },
    set (v) {
        this._sharedUint8Arr[2] = (v ? 1 : 0);
    },
});
```

性能对比

扩展编辑器

扩展编辑器

本章主要介绍一些编辑器的扩展功能，包括以下内容：

- 扩展管理器面板
- 入门示例-菜单
- 入门示例-面板
- 入门示例-扩展间通信
- 扩展改名
- 安装与分享
- 上架扩展到商店
- 增强已有功能

- 扩展系统详解
- Editor 接口说明

扩展管理器面板

扩展管理器面板

扩展管理器 用于管理编辑器内的扩展。点击 Cocos Creator 顶部菜单栏中的 **扩展 -> 扩展管理器** 即可打开：

扩展模板与编译构建

扩展模板与编译构建

本文将详细介绍如何通过 Creator 提供的扩展模板创建一个带有面板的扩展并使用。

在编辑器主菜单上选择 **扩展 -> 创建扩展** 菜单，即可打开新建面板。

入门示例-菜单

入门示例-菜单

本文将演示如何创建一个 Cocos Creator 扩展，本文将包含以下知识点：

- 创建扩展
- 新增菜单
- 菜单消息

创建并安装扩展

在编辑器的菜单栏中找到 **扩展 -> 创建扩展** 菜单，如下图所示：

入门示例-面板

入门示例-面板

在文档 入门示例-菜单 中讲解了怎么创建一个最简单的扩展，接下来我们看看如何创建一个面板并与之通信。

通过模板创建

在 Cocos Creator 中创建面板最快捷的方式是通过 **包含面板的扩展模板** 创建，如下图所示：

入门示例-扩展间通信

入门示例-扩展间通信

在前面两篇文档 入门示例-菜单 和 入门示例-面板 中，我们介绍了：

- 怎么创建扩展
- 怎么在扩展中定义菜单
- 怎么在扩展中定义消息
- 怎么在扩展中定义面板

本文主要演示两个扩展之间如何通信，将涉及到三个话题：

- 如何打开另一个扩展的面板
- 如何向另一个扩展发送消息
- 如何发送和监听广播消息

打开另一个扩展的面板

有时候我们需要在自己写的扩展中打开另一个扩展，接下来我们就试着对 **入门示例-菜单** 中的扩展示例进行改造，使它可以打开 **入门示例-面板**。

修改后的 `package.json` 如下：

```
{
    "package_version": 2,
    "version": "1.0.0",
    "name": "hello-world",
    ...
    "contributions": {
        "menu": [
            {
                "path": "Develop/HelloWorld",
                "label": "test",
                "message": "log"
            },
            {
                "path": "Develop/HelloWorld",
                "label": "open other",
                "message": "open-other"
            }
        ],
        "messages": {
            "log": {
                "methods": [
                    "log"
                ]
            },
            "open-other": {
                "methods": [
                    "openOther"
                ]
            }
        }
    }
}
```

我们修改了 `contributions.menu`，新增了 `open other` 菜单项，并且把此扩展的菜单都放到了 Develop/HelloWorld 下。 刷新扩展后，可以在顶部菜单栏找到如下图所示的菜单内容：

扩展改名

扩展改名

修改显示名称

如果想对扩展改名，只需修改 `package.json` 文件中的 `name` 字段即可。代码示例如下：

```
// "name": "simple-1649426645745"
"name": "hello-world"
```

像上面一样将 `name` 字段改为 "hello-world" 并在扩展管理器面板刷新扩展，就可以看到扩展的名称改成了 **hello-world**。

安装与分享

安装与分享

安装位置

Cocos Creator 在启动项目的过程中会搜索并加载 **项目** 路径下的扩展。

项目

扩展存放的地址为：

- `${你的项目地址}/extensions`

安装扩展

可以通过三种方式获得扩展：

- 其他开发者打包分享，请参考下文 打包扩展。
- 从 **Dashboard->商城** 下载。

上架扩展到资源商店

上架扩展到资源商店

Cocos Creator 内置了 **扩展商店**，可供用户浏览、下载和自动安装官方或者第三方扩展、资源。同时用户也可以将自己开发的扩展、美术素材、音乐音效等资源提交到扩展商店以便分享或者售卖。下面我们以提交扩展为例来讲解具体的提交流程。

增强已有功能

增强已有的功能

Cocos Creator 支持各个扩展间互相提供数据（`contributions`）。

我们在编写一个扩展的时候，可以查询编辑器内已有功能是否提供了对外接收 `contributions` 的功能。如果对应功能提供该功能，则能够在编写扩展的时候使用这些功能。

contributions 数据定义

`contributions` 功能，统一在 `package.json` 里的 `contributions` 字段中定义，如下所示：

```
{
    "name": "hello-world",
    "contributions": {
        "builder":{ ... },
        "assets":{ ... },
        "profile": { ... },
        "scene": { ... },
        "menu": [ ... ],
        "inspector":{ ... },
        "messages": { ... },
        "shortcuts": { ... },
        "preferences": { ... },
        "project": { ... }
    },
}
```

字段说明

`contributions` 提供了与编辑器各功能系统交互的能力，主要涉及到的功能如下：

- `builder` - 自定义构建流程，详细信息请参考文档 自定义构建流程。
- `assets` - 增强资源管理器面板，详细信息请参考文档 增强资源管理器面板。
- `profile` - 定义扩展需要用到的配置，详细信息请参看文档 配置系统。
- `scene` - 在扩展中编写需要和引擎、项目脚本交互的脚本，详细信息请参看文档 调用引擎 API 和项目脚本。
- `inspector` - 自定义 **属性检查器** 面板，详细信息请参看文档 自定义属性检查器面板。
- `menu` - 定义扩展需要新增的菜单信息，详细信息请参看文档 自定义主菜单。
- `messages` - 定义扩展需要用到的消息列表，详细信息请参看文档 自定义消息。
- `shortcuts` - 定义扩展需要用到的快捷键，详细信息请参看文档 自定义快捷键。
- `preferences` - 自定义偏好设置，详细信息请参看文档 自定义偏好设置面板。
- `project` - 自定义项目设置，详细信息请参看文档 自定义项目设置面板。

自定义主菜单

自定义主菜单

编辑器顶部有一栏主菜单，在扩展内可以方便的在这个菜单栏添加自己的菜单。

注册菜单

当扩展需要添加菜单的时候，只需要填写 `contributions.menu` 对象。例如我们在 "扩展" 菜单里增加一个菜单项，可以修改 `package.json`，代码示例如下：

```
{
    // package.json
    "name": "hello-world",
    "contributions": {
        "messages": {
            "open-panel": {
                "methods": ["openPanel"]
            }
        },
        "menu": [
            {
                "path": "i18n:menu.extension",
                "label": "Open Hello World",
                "icon": "./static/icon.png",
                "message": "open-panel"
            }
        ]
    }
}
```

上面的配置信息将会在编辑器的 "扩展" 菜单里新增一个 "Open Hello World" 菜单，点击这个菜单后将会按照 message 配置发送一条 `open-panel` 消息给当前扩展，若当前扩展配置了这个消息的监听以及对应的 `openPanel` 处理函数，将会被触发。

关于消息的定义请参考文档 自定义消息。

下面我们来看看 `menu` 对象中各字段的意义：

path

类型 {string} 必填

填写格式为：[顶部已有菜单路径][/路径1][/路径2]，以下写法都是合理的：

- `i18n:menu.extension` - 以扩展菜单作为父菜单
- `i18n:menu.extension/Hello World` - 在扩展菜单中添加一个 `Hello World` 菜单项作为父菜单
- `MyMenu` - 在顶部菜单栏添加一个 `MyMenu` 菜单作为父菜单
- `MyMenu/Hello World` - 在顶部菜单栏添加一个 `MyMenu`，并再添加一个 `Hello World` 菜单项作为父菜单

顶部菜单栏中，预设的菜单有：

- i18n:menu.project - “项目” 菜单
- i18n:menu.node - “节点” 菜单
- i18n:menu.panel - “面板” 菜单
- i18n:menu.extension - “扩展” 菜单
- i18n:menu.develop - “开发者” 菜单

label

类型 {string} 必填

菜单项目的名称，支持 i18nkey 语法。

icon

类型 {string} 可选

菜单的图标相对路径，扩展使用的素材一般放在名为 `static` 的文件夹下面，若不存在则新建一个。

message

类型 {string} 可选

菜单点击后触发的消息，此消息需要在 `contributions.messsages` 中先定义。

自定义消息

自定义消息

在 Cocos Creator 编辑器架构中，所有的交互都是通过消息通信实现的，本文将讲解如何自定义一条消息，并调用这条消息。

定义一条消息（监听消息）

只有在 `package.json` 文件的 `contributions.messages` 字段里定义过的消息才能被使用。消息的定义如下所示：

```
{
    "name": "hello-world",
    "contributions": {
        "messages": {
            "test-messasge": {
                "public": false,
                "description": "",
                "doc": "",
                "methods": []
            }
        }
    }
}
```

`test-messasge` 为消息名称，下面我们逐一讲解每个属性的含义。

public

类型 {string} 可选

是否对外显示这条消息，如果为 true，则会在 **开发者->消息列表** 面板显示这条消息的基本信息。

description

类型 {string} 可选

消息摘要信息，如果 public 为 true，则会在消息管理面板显示，支持 i18nkey 语法。

doc

类型 {string} 可选

消息文档说明，如果 public 为 true，则会在消息管理面板显示，支持 i18nkey 语法。

这个文档使用 markdown 格式撰写并渲染。

methods

类型 {string[]} 可选

消息触发的方法队列。

这是一个字符串数组，字符串为扩展或者面板上的方法（methods）。 如果是触发扩展主程序的方法，则直接定义 `methodName`，如果要触发扩展里定义的面板上的方法，则要填写 `panelName.methodName`。

下面的示例中，`package-message` 将触发扩展主程序中的 `receiveMessage` 方法，`panel-message` 将触发 `test-panel` 面板中的 `receiveMessage` 方法。

```
{
    "name": "hello-world",
    "panels": {
        "test-panel": {
            "title": "HelloWorld",
            "main": "./dist/panel/index.js"
        }
    },
    "contributions": {
        "messages": {
            "package-message": {
                "public": true,
                "description": "Test Message: send to extension main.js",
                "doc": "Unable to find inheritance data. Please check the specified source for any missing or incorrect information.\nLine breaks are also supported.\n- options {a
                "methods": [
                    "receiveMessage"
                ]
            },
            "panel-message": {
                "methods": [
                    "test-panel.receiveMessage"
                ]
            },
            "hello-world:ready": {
                "public": true,
                "description": "Test Broadcast Message"
            }
        }
    }
}
```

操作当前场景

操作当前场景

在一些时候我们需要去操作当前场景里的一些数据，比如修改节点位置、增删节点、增删组件等。

这时候我们可以通过消息去调用场景已经开放的一些接口，详情参考 消息系统。

但如果有一些定制操作的时候，全部使用开放的接口拼接会比较困难，或者步骤非常多。这时候我们可以在扩展中可以定义一个特殊的 **场景脚本** 文件，该脚本会和项目中 `assets\` 目录下的脚本处于同一运行进程，具有相同的运行环境，需要注意的是场景脚本只运行在场景进程，属于编辑器能力。

在 **场景脚本** 里可以调用引擎 `API` 和其他项目脚本，通过这个特性我们可以实现：

- 查询、遍历场景中的节点，获取或修改节点数据
- 调用节点上的引擎组件相关函数，完成工作

注册场景脚本

在 `package.json` 中定义 `contributions.scene.script` 字段，该字段的值是一个相对于扩展包根目录的脚本文件路径，如下所示：

```
{
    "contributions": {
        "scene": {
            "script": "./dist/scene.js"
        }
    }
}
```

场景脚本模板

在 `src` 目录下新建一个 `scene.ts`，编写如下代码：

```
export function load() { };
export function unload() { };
export const methods = { };
```

`load` - 模块加载的时候触发的函数

`unload` - 模块卸载的时候触发的函数

`methods` - 模块内定义的方法，可用于响应外部消息

通过场景调用引擎 API

接下来，我们通过对主摄像机进行旋转，来演示场景脚本如何调用引擎 API。

为了调用引擎 API，我们需要在 `scene.ts` 开头加入引擎脚本的搜索路径，并编写相应的代码，最终代码如下所示：

```
import { join } from 'path';

// 临时在当前模块增加编辑器内的模块为搜索路径，为了能够正常 require 到 cc 模块，后续版本将优化调用方式
module.paths.push(join(Editor.App.path, 'node_modules'));

// 当前版本需要在 module.paths 修改后才能正常使用 cc 模块
// 并且如果希望正常显示 cc 的定义，需要手动将 engine 文件夹里的 cc.d.ts 添加到插件的 tsconfig 里
// 当前版本的 cc 定义文件可以在当前项目的 temp/declarations/cc.d.ts 找到
import { director } from 'cc';

export function load() { };

export function unload() { };

export const methods = {
    /**
     * 旋转 Main Camera 的角度
     * 这个函数可以是一个异步（async）函数，函数的返回值，将会返回给触发执行函数的消息，详看下面的示例
     * @param num
     * @returns {boolean}
     */
    rotateCamera(num: number) {
        // TS 定义是确定的，但是触发这个函数的消息所发送的参数可能是非法的
        // 所以这里严谨的话，还是需要一些基础的容错处理
        if (typeof num !== 'number') {
```

```
            num = parseFloat(num + '');
        }
        if (isNaN(num)) {
            num = 10;
        }

        // 通过引擎的方法，进行一些基础处理
        const mainCamera = director.getScene().getChildByName('Main Camera');
        if (mainCamera) {
            let euler = new Vec3();
            euler.set(mainCamera.eulerAngles.x, mainCamera.eulerAngles.y + num, mainCamera.eulerAngles.z);
            mainCamera.setRotationFromEuler(euler);
            return true;
        }
        return false;
    },
};
```

上面的代码中，我们定义了一个 `rotateCamera` 方法，此方法每执行一次，就会让主摄像机绕 `Y` 轴旋转传入的 num 或者 `10` 度。

在其他扩展脚本中，我们可以通过消息使用刚刚定义的 `rotateCamera` 函数，代码如下：

```
// 这个定义在生成的扩展的根目录的 @types 文件夹里
// 所有的扩展公开定义都按照规范 @types/packages/{ExtensionName}/@types/public 存放
// 这里引入 Scene 扩展内针对场景脚本的定义
import type { ExecuteSceneScriptMethodOptions } from '../@types/packages/scene/@types/public';

// 这里直接找到插件的 package.json 定义，拿到里面的扩展名字
import packageJSON from '../package.json';

const options: ExecuteSceneScriptMethodOptions = {
    name: packageJSON.name,
    method: 'rotateCamera',
    args: [10],
};

// result: true/false
const result = await Editor.Message.request('scene', 'execute-scene-script', options);
```

`ExecuteSceneScriptMethodOptions` 的属性定义如下：

- name - `scene.ts` 所在的扩展包名，如果是本扩展中可以使用 `packageJSON.name`
- method：`scene.ts` 中定义的方法
- args： 参数，可选，任意可序列化数据，将会原封不动的传递给执行函数

由于扩展间通信实现是基于 Electron 的底层跨进程 IPC 机制，传输的数据均会被序列化为JSON。所以传输的数据不可以包含原生对象，否则可能导致进程崩溃或者内存暴涨。如上面代码中的 `options.args` 参数和场景脚本方法的返回值，建议只传输纯 `JSON` 对象。

增强资源管理器面板

增强资源管理器面板

为了能更好的理解本篇文档内容，在继续阅读本文档之前，推荐大家先阅读 Cocos Creator 扩展编辑器 文档，了解扩展开发相关知识。

自定义右击菜单

若要自定义右击菜单，请先参考 创建并安装扩展 新建一个扩展，在扩展的 `package.json`文件中，通过定义 `contributions.assets.menu` 字段，即可对 **资源管理器** 面板的右击菜单显示事件进行监听，可以实现菜单的追加，如下所示：

```
// package.json
{
  "contributions": {
    "assets": {
      "menu": {
        "methods": "./dist/assets-menu.js",
        "createMenu": "onCreateMenu",
        "assetMenu": "onAssetMenu",
        "dbMenu": "onDBMenu",
        "panelMenu": "onPanelMenu"
      }
    }
  }
}
```

各字段含义：

- `methods` - 菜单事件处理函数（示例中的 `on***Menu` 系列）所在的脚本相对路径
- `createMenu` - **创建资源** 菜单显示时触发的事件，有两个触发时机:
  - 点击资源管理器面板左上角的 + 按钮
  - 资源菜单中的 **新建** 菜单项被选中时
- `dbMenu` - 右击资源数据库根节点 `assets` 时触发的事件
- `assetMenu` - 右击普通资源节点或目录时触发的事件
- `panelMenu` - 右击资源管理面板空白区域时触发的事件

生成 `./dist/assets-menu.js` 文件的 `assets-menu.ts` 内容如下：

```
export function onCreateMenu(assetInfo: AssetInfo) {
  return [
    {
      label: 'i18n:extend-assets-demo.menu.createAsset',
      click() {
        if (!assetInfo) {
          console.log('get create command from header menu');
        } else {
          console.log('get create command, the detail of diretory asset is:');
          console.log(assetInfo);
        }
      },
    },
  ];
};

export function onAssetMenu(assetInfo: AssetInfo) {
  return [
    {
      label: 'i18n:extend-assets-demo.menu.assetCommandParent',
      submenu: [
        {
          label: 'i18n:extend-assets-demo.menu.assetCommand1',
          enabled: assetInfo.isDirectory,
          click() {
            console.log('get it');
            console.log(assetInfo);
          },
        },
        {
          label: 'i18n:extend-assets-demo.menu.assetCommand2',
          enabled: !assetInfo.isDirectory,
          click() {
            console.log('yes, you clicked');
```

```
            console.log(assetInfo);
          },
        },
      ],
    },
  ];
};
```

assets-menu.ts 中 on****Menu(assetInfo:AssetInfo):MenuItem[] 系列函数参数和返回值说明如下：

- 参数 AssetInfo：{}
  - displayName：string - 资源用于显示的名字
  - extends：string[] - 可选，继承类
  - importer：string - 导入器名字
  - isDirectory：boolean - 是否是文件夹
  - imported：boolean - 是否导入完成
  - invalid：boolean - 是否导入失败
  - name：string - 资源名字
  - file：string - 资源文件所在的磁盘绝对路径
  - readonly：boolean - 是否只读
  - type：string - 资源类型
  - url：string - db:// 开头的资源地址
  - uuid：string - 资源 ID
- 返回值 [MenuItem]
  - type：string - 可选，可选项分别为 normal、separator、submenu、checkbox 或 radio
  - label：string - 可选，显示的文本
  - sublabel：string - 可选，显示的二级文本
  - submenu：MenuItem[] - 可选，子项菜单
  - click：function - 可选，点击事件
  - enabled：boolean - 可选，是否可用，不可用会有置灰样式
  - visible：boolean - 可选，是否显示
  - accelerator：string - 可选，显示快捷键
  - checked：boolean - 可选，当 type 为 checkbox / radio 时是否选中

    更多属性可参考 electron menu-item 的数据格式。

示例中以 i18n: 开始的字符串，需要配置多语言相关内容，请参考多语言系统（i18n）。

最终实现效果如下图所示：

自定义资源数据库

自定义资源数据库

所有项目内的资源文件都是通过资源数据库进行管理，其中项目内的 assets 目录存放的是当前项目的资源，引擎仓库里 editor/assets 里存放的是引擎内置的资源（如：常见的图片、脚本等）。

当我们在扩展内使用了资源时，需要将扩展内的资源文件夹注册到资源数据库里，并在扩展发布时将资源随着扩展一起发布。

通过本文我们将学会如何注册一个资源文件夹，并在脚本里使用资源。

注册配置

资源注册需要在 contributions 中使用 asset-db 字段进行配置，如下所示：

```
{
    "name": "test-package",
    "contributions": {
        "asset-db": {
            "mount": {
                "path": "./assets",
                "readonly": true
            }
        }
    }
}
```

上面的示例中，我们把扩展 test-package 根目录下的 assets 文件夹注册到了资源数据库中。

脚本资源

在 test-package/assets/ 目录下创建一个脚本 foo.ts，内容如下：

```
/// foo.ts
import { _decorator, Component, Node } from 'cc';
const { ccclass, property } = _decorator;

@ccclass('Foo')
export class Foo extends Component {
    start () {
        console.log('foo');
    }
}
```

为了使用 cc 的定义，我们需要拷贝 {项目目录}\temp\declarations 的定义文件到扩展根目录下。

由于 foo.ts 只是作资源使用，不属于扩展源码，所以我们需要在 tsconfig.json 中加入 exclude 配置进行排除，否则会出现编译错误。

```
{
    "compilerOptions": {
        ...
    },
    "exclude": ["./assets"]
}
```

注意：扩展中的脚本资源可以在 Cocos Creator 工程中编写并测试完成后，再复制到扩展的 assets 目录。

其他资源

图片、文本、字体等资源直接放入 assets 目录下即可。

使用扩展中的资源

刷新扩展，可以在 Cocos Creator 编辑器的 **资源管理器** 窗口中看到新增了一个 test-package 资源包，如下图所示：

自定义属性检查器面板

自定义属性检查器面板

有自定义属性检查器需求的开发者建议先参考文档 通过修饰器定义属性，若该文档满足需求，建议优先使用该文档中的方法。

属性检查器 作为 Cocos Creator 里显示当前选中状态的模块，为大家提供了一些基础的扩展能力。

在 **属性检查器** 里，定义了两个层级的数据:

1. 选中的物体主类型
2. 渲染主类型内容时，内容里包含的子数据类型

当在 **层级管理器/资源管理器** 中选中一个 **节点/资源** 时，Cocos Creator 会将物体被选中的消息进行广播。当 **属性检查器** 接收到消息后，会检查被选中的物体的类型，例如选中的是节点，那么类型便是 `node`。

针对两种类型，允许注册两种渲染器：

1. 主类型渲染器
2. 主类型渲染器接收数据开始渲染的时候，允许附带子类型渲染器

示例里选中的是 `node`，`node` 上携带了多个 `component`，所以主类型就是 `node`，而子类型则是 `component`。

在 **属性检查器** 收到选中物体的广播消息后，首先确定类型，而后 **属性检查器** 会将收到的数据（物体的 uuid），传递给 `node` 渲染器，将渲染权限完全移交。

而在 `node` 渲染器里，会根据自身的实现，渲染到每个 `组件` 的时候，将该区域的渲染权限交给 `子类型渲染器`。大部分情况下，我们不需要关注这些。我们先来看看几种常用的的自定义方式吧。

自定义 Component 渲染

默认提供的组件渲染器有时候并不能满足我们的需求，这时候我们就需要自定义一个组件的渲染方式。

首先在项目内新建一个脚本组件 `CustomLabelComponent.ts`，并添加一个字符串属性，内容如下:

```
import { _decorator, Component, Node } from 'cc';
const { ccclass, property } = _decorator;

@ccclass('CustomLabelComponent')
export class CustomLabelComponent extends Component {
    @property
    label = '';

    start () {

    }
}
```

将组件拖到节点上，这时候能看到组件上有一个输入框。

新建一个扩展，在扩展的 `package.json` 里注册如下的 `contributions.inspector` 信息：

```
{
    "contributions": {
        "inspector": {
            "section": {
                "node": {
                    "CustomLabelComponent": "./dist/contributions/inspector/comp-label.js"
                }
            }
        }
    }
}
```

自动渲染

编写一个 `src/contributions/inspector/comp-label.ts` 文件，内容如下：

```
'use strict';

type Selector<$> = { $: Record<keyof $, any | null> }

export const template = `
<ui-prop type="dump" class="test"></ui-prop>
`;

export const $ = {
    test: '.test',
};

export function update(this: Selector<typeof $> & typeof methods, dump: any) {
    // 使用 ui-porp 自动渲染，设置 prop 的 type 为 dump
    // render 传入一个 dump 数据，能够自动渲染出对应的界面
    // 自动渲染的界面修改后，能够自动提交数据
    this.$.test.render(dump.value.label);
}
export function ready(this: Selector<typeof $> & typeof methods) {}
```

编译并刷新扩展后，我们可以发现 `CustomLabelComponent` 组件的渲染被接管了。

> **注意**：每一个 `ui-prop` 对应一条属性，若要显示多条属性需要定义多个 `ui-prop`。

手动渲染

上面的自动渲染示例中，我们使用了类型为 `dump` 的特殊的 `ui-prop` 进行渲染数据提交，它使我们可以快捷的接管组件的渲染，但若面临一些极端情况却很难处理一些细节问题，此时可以切换到手动渲染模式，代码如下所示：

```
'use strict';

type Selector<$> = { $: Record<keyof $, any | null> }

export const template = `
<!-- 帮忙提交数据的元素 -->
<ui-prop type="dump" class="test"></ui-prop>
<!-- 实际渲染的元素 -->
<ui-label class="label"></ui-label>
<ui-input class="test-input"></ui-input>
`;

export const $ = {
    label:'.label',
    test: '.test',
    testInput: '.test-input',
};

type PanelThis = Selector<typeof $> & { dump: any };

export function update(this: PanelThis, dump: any) {
    // 缓存 dump 数据，请挂在 this 上，否则多开的时候可能出现问题
    this.dump = dump;
    // 将 dump 数据传递给帮忙提交数据的 prop 元素
    this.$.test.dump = dump.value.label;
    // 更新负责输入和显示的 input 元素上的数据
    this.$.testInput.value = dump.value.label.value;
    this.$.label.value = dump.value.label.name;
```

```
}
export function ready(this: PanelThis) {
    // 监听 input 上的提交事件，当 input 提交数据的时候，更新 dump 数据，并使用 prop 发送 change-dump 事件
    this.$.testInput.addEventListener('confirm', () => {
        this.dump.value.label.value = this.$.testInput.value;
        this.$.test.dispatch('change-dump');
    });
}
```

手动渲染模式下依然要通过一个类型为 `dump` 的 `ui-prop` 元素进行数据提交。但 `template` 中的 `html` 布局就是一个完全可以自由掌控的内容，可根据需求制定出十分复杂的显示方式。

自定义 Asset 渲染

当在资源管理窗口（Assets）中选中文件时，属性检查器窗口会显示当前选中文件的重要属性，如果默认显示的信息不满足要求，我们也可以自定义渲染内容。

在 `package.json` 中添加 `contributions.section.asset` 字段，并定义对应资源类型的自定义渲染脚本，如下所示：

```
{
    "contributions": {
        "inspector": {
            "section": {
                "asset": {
                    "effect": "./dist/contributions/inspector/asset-effect.js"
                }
            }
        }
    }
}
```

`effect` 表示我们要对 Cocos Shader（*.effect） 文件类型的资源属性面板自定义渲染。常见的资源文件类型如下：

- `scene` - 场景文件
- `typescript` - TypeScript 脚本文件
- `prefab` - 预制体文件
- `fbx` - FBX 文件
- `material` - 材质文件
- `directory` - 文件夹
- `image` - 图片文件

可通过查看文件对应的 `*.meta` 中的 `importer` 字段获取该文件的类型定义。

接下来，在扩展目录下新建一个 `src/contributions/inspector/asset-effect.ts` 脚本文件，并编写如下代码：

```
'use strict';

interface Asset {
    displayName: string;
    file: string;
    imported: boolean;
    importer: string;
    invalid: boolean;
    isDirectory: boolean;
    library: {
        [extname: string]: string;
    };
    name: string;
    url: string;
    uuid: string;
    visible: boolean;
    subAssets: {
        [id: string]: Asset;
    };
}

interface Meta {
    files: string[];
    imported: boolean;
    importer: string;
    subMetas: {
        [id: string]: Meta;
    };
    userData: {
        [key: string]: any;
    };
    uuid: string;
    ver: string;
}

type Selector<$> = { $: Record<keyof $, any | null> } & { dispatch(str: string): void, assetList: Asset[], metaList: Meta[] };

export const $ = {
    'test': '.test',
};

export const template = `
<ui-prop>
    <ui-label slot="label">Test</ui-label>
    <ui-checkbox slot="content" class="test"></ui-checkbox>
</ui-prop>
`;

type PanelThis = Selector<typeof $>;

export function update(this: PanelThis, assetList: Asset[], metaList: Meta[]) {
    this.assetList = assetList;
    this.metaList = metaList;
    this.$.test.value = metaList[0].userData.test || false;
};

export function ready(this: PanelThis) {
    this.$.test.addEventListener('confirm', () => {
        this.metaList.forEach((meta: any) => {
            // 修改对应的 meta 里的数据
            meta.userData.test = !!this.$.test.value;
        });
        // 修改后手动发送事件通知，资源面板是修改资源的 meta 文件，不是修改 dump 数据，所以发送的事件和组件属性修改不一样
        this.dispatch('change');
    });
};

export function close(his: PanelThis, ) {
    // TODO something
};
```

编译、刷新扩展后，回到 Cocos Creator 编辑器界面，选中某个 `Cocos Shader` 文件，可以在属性检查器底部发现，多了一个 **Test 复选框**。

> **注意：**多个扩展注册的数据是并存的。如果一个 **组件/资源** 已经有了自定义渲染器，那么再次注册的自定义渲染器都会附加在后面。如果一个 **组件/资源** 没有内置自定义渲染器，使用的是默认的渲染器，那么当扩展注册自定义渲染器的时候，会完全接管渲染内容。

自定义构建流程

自定义构建流程

对构建流程进行自定义的前提是需要对构建发布的整体流程有所了解，不熟悉的开发者建议先阅读 构建流程简介与常见问题指南。

为了能更好的理解本篇文档内容，在继续阅读本文档之前，推荐大家先阅读 Cocos Creator 扩展编辑器 文档，了解扩展开发相关知识。

为方便书写，本文中我们约定将用于自定义构建流程的扩展简称为： **构建扩展**

构建扩展模板

Cocos Creator 提供了快捷方式生成 **构建扩展模板**：点击 **项目** -> **新建构建扩展包** 菜单即可生成 `cocos-build-template`，如下图所示：

自定义项目设置面板

自定义项目设置面板

项目设置是存放和项目开发相关配置的地方，例如引擎模块、文件后处理配置等，对应配置的修改变动会记录在项目目录的 settings 目录下。

> **注意**：settings 目录默认是进入项目版本管理的，在扩展自定义项目设置时，请不要记录一些临时数据或者是存在系统差异的数据配置，以免不同设备的用户在同步数据时遇到问题。

面板介绍

项目设置 里存放的是和项目运行相关的配置，这部分配置存放在项目目录下的 `settings` 文件夹中，需要纳入版本管理，多人共享配置，否则可能导致不同机器上运行不一致的问题。

可在顶部菜单栏中找到 **项目** -> **项目设置** 菜单，如下图所示：

自定义偏好设置面板

自定义偏好设置面板

偏好设置，通常来说存放的是开发者个性化定制的设置，或一些绝对路径的配置，这些信息可能和当前的硬件绑定，如环境配置、一些开发工具的绝对路径。因此通常来说不需要通过 git 同步给该项目其他开发者。

在自定义偏好设置面板之前，开发者应该提前设计好是将配置存放在项目设置内，还是偏好设置内。

并且在偏好设置面板里，还能选择存放在项目还是存放在全局。存放在项目的配置只对当前项目生效，如果存放在全局，则会影响所有编辑器。

偏好设置面板简介

在顶部菜单栏可找到 **Cocos Creator/File -> 偏好设置** 菜单，如下图所示：

快捷键

自定义快捷键

编辑器内的快捷键由 "快捷键管理器" 统一管理。每一个快捷键可以绑定一个消息，当快捷键按下的时候，会触发绑定的消息。

定义快捷键

定义快捷键需要在 `package.json` 的 `contributions.shortcuts` 字段中进行，如下所示：

```
// package.json
{
    "name": "hello-world",
    "panels": {
        "default": {
            "main": "./panel.js"
        }
    },
    "contributions": {
        "messages": {
            "undo": {
                "title": "i18n:hello.messages.undo.title",
                "methods": ["say-undo"]
            }
        },
        "shortcuts": [
            {
                "message": "undo",
                "when": "panelName === 'hello-world'",
                "win": "ctrl+z",
                "mac": "cmd+z",
            }
        ]
    }
}
```

本示例中，我们定义了一个 **撤销** 操作的快捷键，在 Windows 系统下是 `CTRL + Z`，在 macOS 系统下是 `CMD + Z`。

当对应快捷键被按下时，会发送 `undo` 消息。

> **注意**：此消息需要在 `contributions.messages` 里面先定义好，详请请参考文档 自定义消息。

参数说明

下面我们来看看 `contributions.shortcuts` 各参数的具体说明。

message

类型 {string} 必填

快捷键绑定的消息，当这个快捷键被触发时，会发送此消息。快捷键按下的消息只能发送给当前扩展。

when

类型 {string} 可选

在某些条件下才会触发这个快捷键。

`"when": "panelName === 'hello-world'"` 表示当获得焦点的面板名称为 `hello-world` 时，按下快捷键才会发送 `message` 消息。

win

类型 {string} 必填

在 Windows 平台上，监听的按键。

mac

类型 {string} 必填

在 macOS 上，监听的按键。

扩展系统详解

扩展系统详解

编辑器内提供了许多功能，这里主要介绍一些书写扩展时常用的功能。

基础结构

基础结构

在编写扩展之前，我们首先需要了解一下 Cocos Creator 内，扩展的基础结构。

Electron

Cocos Creator 编辑器是基于 GitHub 的 Electron 内核开发。

Electron 是一个集成了 Node.js 和 Google Chromium 的跨平台开发框架。

多进程机制

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。每个进程独立享有自己的 JavaScript 内容，且彼此之间无法直接访问。当需要在进程之间传递数据时，需要使用进程间通信（IPC）机制。

相关功能可以通过阅读 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 编辑器沿用了 Electron 的主进程和渲染进程的结构设计。所以扩展在编辑器内启动并运行的时候，扩展定义的 main 其实是在主进程启动，而 panels 定义的面板，则在渲染进程启动。进程结构简要概括如下：

扩展包定义

扩展包的定义

扩展包需要在 `package.json` 文件里预先定义好所有功能以及一些基础信息，如下所示：

```
{
    "package_version": 2,
    "version": "1.0.0",
    "name": "first-panel",
    "tilte": "i18n:first-panel.title",
    "description": "i18n:first-panel.description",
    "author": "Cocos Creator",
    "editor": ">=3.4.2",
    "main": "./dist/main.js",
    "dependencies": { ... },
    "devDependencies": { ... },
    "panels": { ... },
    "contributions": {
    },
    "scripts": {
        "build": "tsc -b",
        "watch": "tsc -w"
    }
}
```

package_version

类型 {number} 必填

扩展的版本号，用于提交扩展的版本校验，以及扩展自身的一些升级，数据迁移作为对比的依据。

version

类型 {string} 必填

扩展的版本号，主要用于显示，如需进行逻辑判断，推荐使用 `package_version`。

name

类型 {string} 必填

扩展的名称，这个名字需要和扩展文件夹一一对应。

title

类型 {string} 可选

扩展的显示标题，当配置了 `title` 时，在需要展示扩展名的地方会优先采用 `title`，支持 多语言（i18n） 配置。

description

类型 {string} 可选

扩展的描述，简单概括一下扩展的功能。支持 多语言（i18n） 的多语言语法。

author

类型 {string} 可选

扩展作者的名字，将会显示在 "扩展管理器" 内。

editor

类型 {string} 可选

描述扩展支持的编辑器版本，符合 `semver` 语义化版本控制规范。

main

类型 {string} 可选

一个 js 文件的相对路径，定义功能入口文件，当扩展启动的时候，就会执行 `main` 字段指向的 js 文件，并根据流程触发或执行对应的方法。

panels

类型 {[name: string]: PanelInfo} 可选

扩展内定义的面板信息。可以使用 `Editor.Panel.open('hello-world.list');` 打开定义好的面板。详细信息请参看 面板系统。

contributions

类型 {[name: string]: any} 可选

`contributions` 提供了与编辑器各功能系统交互的能力，更多信息请参看文档 增强已有功能。

scripts

类型 {[name: string]: any} 必填

扩展可执行的命令行。

消息系统

消息系统

Cocos Creator 中有许多独立运行进程，且这些进程间是相互隔离的。在编辑器内需要与其他功能进行交互的时候，需要通过 "消息机制" 进行交互。

编辑器里的"消息系统"是 IPC（进程间通信）的功能扩展封装。这个系统承担起了整个编辑器内通讯交互的重担。

更多关于多进程构架和跨进程通信的介绍请参考文档 基础结构.

消息类型

Cocos Creator 系统内的消息有两种类型：

1. 普通消息：主动发送某条消息到某个功能（扩展）
2. 广播消息：某个功能（扩展）完成了一个操作后向所有人发送通知，告知操作已经完成

普通消息

可以理解成一种对外的接口，例如引擎的 **场景编辑器** 模块已经定义好了一个用于查询节点的 `query-node` 消息，如下所示：

```
{
    "name": "scene",
    "contributions": {
        "messages": {
            "query-node": {
                "methods": ["queryNode"]
            }
        }
    }
}
```

关于如何自定义消息以及消息各字段的含义，请参考文档 自定义消息。

当我们在自己编写的扩展中想要查询场景节点时，就可以使用这个消息来完成，如下所示:

```
const info = await Editor.Message.request('scene', 'query-node', uuid);
```

这种消息类似一种远程调用 (RPC), 拿到的 `info` 对象就是实际查询的节点上的部分数据。

> **注意：** 由于是远程调用，`request` 是不会立即返回的，因此需要使用 `await` 将异步转为同步。

普通消息的命名规范

请使用 **小写** 单词，并且不能包含特殊字符，单词间以 **-** 连接。如 `open-panel`、`text-changed`。

广播消息

广播消息是某一个功能内的操作完成后，对外进行的一种通知。

比如，**场景编辑器** 在启动一个场景后，需要通知所有人 "场景" 已经启动完毕，**场景编辑器** 发送广播消息使用的是如下代码：

```
Editor.Message.broadcast('scene:ready', sceneUUID);
```

接收广播消息

若一个扩展想要接收 `scene:ready` 消息，则需要在 `package.json` 里先定义，如下所示：

```
{
    "name": "hello-world",
    "contributions": {
        "messages": {
            "scene:ready": {
                "methods": ["initData"]
            }
        }
    }
}
```

每当场景准备就绪后，广播的 `scene:ready` 消息就会触发 "hello-world" 扩展里的 `initData` 方法。

发送广播消息

若一个扩展想要发送广播消息，也需要在 `package.json` 里先定义。

比如，"hello-world" 在准备好数据后，同样可以向外广播一条消息，以方便其他扩展与之配合。如下所示：

```
{
    "name": "hello-world",
    "contributions": {
        "messages": {
            "scene:ready": {
                "methods": ["initData"]
            },
            "hello-world:ready": {
                "public": true,
                "description": "hello-world ready notification."
            }
        }
    }
}
```

在适当的时机，"hello-world" 扩展内调用如下代码即可广播给所有人：

```
Editor.Message.broadcast('hello-world:ready');
```

> **注意**：广播消息可以没有 `methods`，当一条消息没有 methods 的时候，不会触发处理函数，我们可以给他增加 public 字段，让它出现在 **消息列表面板**。如上面的定义所示，表示 `hello-world:ready` 没有触发函数，但是会在 **消息列表面板** 上显示。

广播消息的命名规范

格式为 `packageName:actionName`，以下命名都是合法的：

- scene:ready
- scene:query-node
- hello-world:ready
- hello-world:data-loaded

加上 `packageName` 可以防止命名出现冲突，在 `package.json` 中定义消息的时候也能够更加直观的看到监听的是哪一个扩展的哪个广播消息（动作）。

查看消息列表

编辑器内的功能以及扩展对外开放的消息列表，可以通过 **开发者 -> 消息管理** 面板查看。详细定义规则请参考文档 自定义消息。

在代码中发送消息

`send` 方法只发送消息，并不会等待返回。如果不需要返回数据，且不关心是否执行完成，请使用这个方法。

```
Editor.Message.send(pkgName, message, ...args);
```

`request` 方法返回一个 promise 对象，这个 promise 会接收消息处理后返回的数据。

```
await Editor.Message.request(pkgName, message, ...args);
```

`broadcast` 方法只发送，并且发送给所有监听对应消息的功能扩展。

```
Editor.Message.broadcast(`${pkgName}:${actionName}`, ...args);
```

查看公开消息列表

在编辑器的顶部菜单栏中找到 **开发者** -> **消息列表**，可以打开消息管理面板。

多语言系统（i18n）

多语言系统（i18n）

什么是 i18n

i18n（其来源是英文单词 internationalization 的首末字符 i 和 n，18 为中间的字符数）是“国际化”的简称。

在资讯领域，i18n 指让产品（出版物，软件，硬件等）无需做太多改变就能够适应不同的语言和地区的需要。

在程序开发领域，i18n 则是指在不修改内部代码的情况下，能根据不同语言及地区显示相应的本地化内容。

Cocos Creator 扩展系统中内置的多语言方案（i18n）允许扩展配置多份语言的 **键值映射**，并根据编辑器当前的语言设置在扩展里使用不同语言的字符串。

> **注意**：本文将介绍如何赋予 **扩展** 多语言的能力，如需在工程内添加运行时的多语言能力，请参考：
>
> 1. 多语言本地化（L10N）
> 2. i18n 游戏多语言支持

i18n 文件夹

要启用多语言功能（以下简称 i18n），需要在扩展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，`zh.js`对应中文映射数据。如下图所示：

配置系统

配置系统

Cocos Creator 扩展提供了一套配置管理机制，用于在扩展保存用户设置和数据。

配置类型

配置类型有两种：

1. 编辑器配置（editor）
2. 项目配置（project）

编辑器配置

编辑器配置用于存放一些编辑器相关的用户设置和数据，分成三个优先级，从高到低今次为：

```
local -> global -> default
```

在进行配置数据获取时，会优先采用 `local` 中的配置项，若 `local` 中无对应配置项，则会采用 `global` 中的配置项，若 `global` 中也找不到对应配置项，则会采用默认的 `default` 配置。

项目配置

项目配置用于存放一些和项目相关的用户设置和数据，分成两优先级，从高到低为：

`local -> default`

和 **编辑器配置** 的规则类似，在进行配置数据获取时，会优先采用 `local` 中的配置项，若 `local` 中无对应配置项，则会采用默认的 `default` 配置。

注册配置

若要使用配置系统，需要在扩展定义文件 `package.json` 的 `contributions` 字段中定义 `profile` 相关信息，如下所示：

```
{
    "name": "hello-world",
    "contributions": {
        "profile": {
            "editor": {
                "test.a": {
                    "default": 0,
                    "message": "editorTestAChanged",
                    "label": "测试编辑器配置"
                }
            },
            "project": {
                "test.a": {
                    "default": 1,
                    "message": "projectTestAChanged",
                    "label": "测试项目配置"
                }
            }
        }
    },
}
```

contributions.profile 相关的字段释义如下：

- `editor`:{} - 编辑器配置
- `project`:{} - 项目配置
- `test.a`：{} - key 为 test.a 的配置项
- `default`:any - 此配置项的默认值，可选参数
- `message`:string - 此配置项被修改时会触发此消息，可选参数
- `label`:string - 在可以显示配置的地方，可能会显示这个描述。支持 i18n:key 格式，可选参数

`profile` 相关的 TypeScript 接口定义如下：

```
interface ProfileInfo {
    editor: { [ key: string ]: ProfileItem };
    project: { [ key: string ]: ProfileItem };
}

interface ProfileItem {
    // 配置的默认数据
    default: any;
    // 配置更改后，会自动发送这个消息进行通知
    message: string;
    // 简单的描述配置信息的作用，支持 i18n:key 语法
    label: string;
}
```

读取与修改配置

导入扩展定义

```
import packageJSON from '../package.json';
```

`Editor.Profile.getConfig` 最后一个参数为空的情况，会进行 优先级 匹配。

若指定了获取位置（`local` 、 `global` 、`default` 三者之一），则会返回对应的值。如下所示，获取到的 `local` 和 `global` 为 `undefined` 是因为未对其进行设置。

```
await Editor.Profile.getConfig(packageJSON.name, 'test.a'); // 0
await Editor.Profile.getConfig(packageJSON.name, 'test.a', 'local'); // undefined
await Editor.Profile.getConfig(packageJSON.name, 'test.a', 'global'); // undefined
```

修改编辑器配置

用以下代码修改配置后再调用 `getConfig` 可以看到对应变化。

```
await Editor.Profile.setConfig(packageJSON.name, 'test.a', 1);
await Editor.Profile.setConfig(packageJSON.name, 'test.a', 'local', 2);
await Editor.Profile.setConfig(packageJSON.name, 'test.a', 'global', 3);
```

读取项目配置

`Editor.Profile.getProject` 最后一个参数为空的情况，会进行 优先级 匹配。

若指定了获取位置（`local` 、`default` 二者之一），则会返回对应的值。如下所示，获取到的 `local` 为 `undefined` 是因为未对其进行设置。

```
await Editor.Profile.getProject(packageJSON.name, 'test.a'); // 1
await Editor.Profile.getProject(packageJSON.name, 'test.a', 'local'); // undefined
```

修改项目配置

用以下代码修改配置后再调用 `getProject` 可以看到对应变化。

```
await Editor.Profile.setProject(packageJSON.name, 'test.a', 1);
await Editor.Profile.setProject(packageJSON.name, 'test.a', 'local', 2);
```

存储路径

编辑器配置存储路径

层级	路径
local	`{projectPath}/profiles/v2/extensions/{extensionName}.json`
global(mac)	`Users/{name}/.CocosCreator/profiles/v2/extensions/{extensionName}.json`
global(window)	`c/Users/{name}/.CocosCreator/profiles/v2/extensions/{extensionName}.json`
default	`{extensionPath}/package.json`

项目配置存储路径

层级	路径
local	`{projectPath}/settings/v2/extensions/{extensionName}.json`
default	`{extensionPath}/package.json`

面板系统

面板系统

扩展默认情况下是没有界面显示的，如果一个扩展需要实现界面交互，就需要使用到面板系统相关功能。

面板的定义

在 `package.json` 里可以在 `panels` 字段定义一个或者多个面板，如下所示：

```
{
    "name": "hello-world",
    "panels": {
        "default": {
            "title": "world panel",
            "type": "dockable",
            "main": "./dist/panels/default",
            "icon": "./static/default.png"
        },
        "list": {
            "title": "world list",
            "type": "simple",
            "main": "./dist/panels/list",
            "icon": "./static/list.png",

            "flags": {},
            "size": {}
        }
    }
}
```

我们定义了两个面板：`defualt` 和 `list`。`default` 为默认面板，当不指名特定面板的时候，它就做为默认操作对象。

面板各字段含义如下：

- `title`：string - 面板标题，支持 i18n:key，必填
- `main`：string - 面板源码相对目录，必填
- `icon`：string - 面板图标相对目录，必填
- `type`：string - 面板类型（dockable | simple），可选
- `flags`：{} - 标记，可选
    - resizable - 是否可以改变大小，默认 true，可选
    - save - 是否需要保存，默认 false，可选
    - alwaysOnTop - 是否保持顶层显示，默认 flase，可选
- `size`：{} - 大小信息，可选
    - min-width：Number - 最小宽度，可选
    - min-height：Number - 最小高度，可选
    - width：Number - 面板默认宽度，可选
    - height：Number - 面板默认高度，可选

编写面板

在扩展根目录下分别建立两个文件 `src/panels/default/index.ts` 和 `src/panels/list/index.ts`，并将下面的最简面板模板代码分别贴到两个 `index.ts` 文件中：

```
module.exports = Editor.Panel.define({
    listeners: {
        show() { console.log('show'); },
        hide() { console.log('hide'); },
    },
    template: '<div>Hello</div>',
    style: 'div { color: yellow; }',
    $: {
        elem: 'div',
    },
    methods: {

    },
    ready() {

    },
    beforeClose() { },
    close() { },
});
```

`listeners` - 面板的一些事件监听 `template` - 面板的 HTML 布局文件 `style` - 面板的 css 文件 `$` - 全局选择器，用于快速访问一些元素 `methods` - 此面板对外的方法接口 `ready` - 当面板被打开时会调用 `beforeClose` - 面板在关闭前会调用 `close` - 面板在关闭后会调用

显示面板

可以使用 `Editor.Panel.open` 方法打开任何面板（本扩展自己的面板以及其他扩展的面板）。

假设扩展为 `hello-world`，那以下两种方式都可以打开默认面板：

```
// Editor.Panel.open('hello-world.defualt');
Editor.Panel.open('hello-world');
```

打开其它面板的方式为：

```
// Editor.Panel.open('{extension-name}.panelName');
Editor.Panel.open('hello-world.list');
```

通信交互

Cocos Creator 扩展系统基于 Electron 的多进程方式构建，每一个扩展是一个独立的进程，扩展中的每一个面板，也是一个独立的进程。 因此，扩展与面板之间、面板与面板的交互只能通过进程间通信（IPC）实现。 详细信息请参考文档 消息系统。

面板向外发送消息

由于面板关闭后，进程也会退出，因此我们通常将扩展作为内存数据的载体。面板中需要用到的数据和逻辑接口一般会从扩展主进程里获取。

如果想 **查询** 和 **设置** 位于扩展主进程中的数据，假设扩展中定义了 `queryData` 和 `saveData` 两个消息，我们可以这样使用：

```
const data = await Editor.Message.request(packageName, 'queryData', dataName);
await Editor.Message.request(packageName, 'saveData', dataName,dataValue);
```

如果想广播通知整个扩展系统，则可以利用 **广播消息** 机制实现，详细信息请参考文档 消息系统。

面板接收消息

```
// package.json
{
"contributions": {
        "messages": {
            "log": {
                "methods": ["log"]
            }
        }
    }
}
```

上面的消息定义了一个 log 消息，并由此扩展的主进程中的 log 方法处理。接下来我们稍作修改，使消息的接收方变成面板：

```
// package.json
{
"contributions": {
        "messages": {
            "log": {
                "methods": ["default.log"]
            }
        }
    }
}
```

`default.log` 使得消息接收方变成了 `default` 面板，只需要在面板中实现一个 `log` 方法，就能够顺利处理此消息了。

更好的面板资源组织方式

在上面的最简面板模板中，我们有两行面板显示相关的代码：

```
module.exports = Editor.Panel.define({
    ...
    template: '<div>Hello</div>',
    style: 'div { color: yellow; }',
    ...
});
```

大部分情况下，面板布局不可能这么简单。如果继续将复杂的 HTML 布局和 css 样式写在这里，代码将变得不可维护。可以参考文档 入门示例-面板 中创建的项目，我们可以将 `html` 和 `css` 代码分离为独立的文件，放入 `static` 文件夹中。

最终形成的面板模板代码如下所示：

```
import { readFileSync } from 'fs-extra';
import { join } from 'path';

module.exports = Editor.Panel.define({
    listeners: {
        show() { console.log('show'); },
        hide() { console.log('hide'); },
    },
    template: readFileSync(join(__dirname, '../../../static/template/default/index.html'), 'utf-8'),
    style: readFileSync(join(__dirname, '../../../static/style/default/index.css'), 'utf-8'),
    $: {
        app: '#app',
    },
    methods: {

    },
    ready() {

    },
    beforeClose() { },
    close() { },
});
```

更多面板实用详情请参考 入门示例-面板。

UI 组件

UI 组件

UI 组件面板

为了方便布局，编辑器内提供了许多预设的 UI 组件。

1、找到编辑器顶部主菜单中的 **开发者 -> UI 组件** 查看。

Editor 接口说明

Editor API 说明

本文介绍编辑器提供的接口的说明

- App
- Clipboard
- Dialog
- I18n
- Logger
- Message
- Network
- Package
- Panel
- Profile
- Project
- Selection
- Utils

App

App

Creator 基本信息

变量

dev

• **dev**: `boolean`

是否是开发模式

```
Editor.App.dev;  // true
```

home

• **home**: `string`

Creator 的主目录，存放一些临时文件和配置信息

```
Editor.App.home;  // "C:\\Users\\Administrator\\.CocosCreator_Develop"
```

path

• **path**: `string`

Creator 程序所在文件夹

```
Editor.App.path;  // "D:\\Program\\CocosEditor\\Creator\\3.4.0\\resources\\app.asar"
```

temp

• **temp**: `string`

获取当前编辑器的临时缓存目录

```
Editor.App.temp;  // "C:\\Users\\ADMINI~1\\AppData\\Local\\Temp\\CocosCreator\\3.4.0"
```

userAgent

• **userAgent**: `string`

Creator 使用的用户代理信息

```
const UA = Editor.App.userAgent;  // "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) CocosCreator/3.4.0 Chrome/91.0.4472.106 Electron/13.1.4 Saf
```

version

• **version**: `string`

Creator 版本号

```
const version = Editor.App.version;  // "3.4.0"
```

函数

quit

▸ **quit**(): `void`

正常退出 Creator，会逐个询问所有扩展，当所有扩展都允许关闭后，才会开始关闭流程

```
Editor.App.quit();
```

Clipboard

Clipboard

系统剪切板，当调用剪切板接口的时候，会修改系统剪切板内容。请谨慎使用

接口说明

```
export type ICopyType = 'image' | 'text' | 'files' | string;
```

函数

clear

▸ **clear**(): `void`

清空剪贴板

```
Editor.Clipboard.clear();
```

has

▸ **has**(type: `ICopyType`): `boolean`

判断当前剪贴板内容是否含有指定类型的数据

请求参数

Name	Type	Description
type	ICopyType	剪贴板内容的类型

返回结果

```
boolean
```

```
const res = Editor.Clipboard.has('files');  // false
```

read

▸ **read**(type: `ICopyType`): `any`

获取剪贴板内容

请求参数

Name	Type	Description
type	ICopyType	剪贴板内容的类型

```
const textRes = Editor.Clipboard.read('text');  // 'your copy text'
const filesRes = Editor.Clipboard.read('files');  // []
```

write

▸ **write**(type: `ICopyType`, value: `string | FileList`): `boolean`

写入剪贴板内容，请勿频繁调用，这个操作会覆盖系统剪切板，频繁调用可能会影响用户正常操作

请求参数

Name	Type	Description
type	ICopyType	剪贴板内容的类型
value	string\ FileList	复制到剪贴板中的内容

返回结果

```
boolean
```

```
Editor.Clipboard.write('text', 'you can test other type');
```

Dialog

Dialog

对话框弹窗，当在扩展进程使用的时候，弹出普通弹窗，如果在面板进程使用，则会弹出模态弹窗

通用接口说明

MessageDialogOptions

Name	Type	Description
title?	string	标题，在部分可能会被隐藏
detail?	string	详细描述
buttons?	string array	在弹窗上新增一个或者多个按钮
default?	number	打开弹窗时默认选中的按钮
cancel?	number	当弹窗关闭时，默认触发的按钮
checkboxLabel?	number	在弹窗上附加一个可选项
checkboxChecked?	number	弹窗上附加的可选项默认值

SelectDialogOptions

Name	Type	Description
title?	string	标题，在部分可能会被隐藏
button?	string	弹窗按钮上的文字
path?	string	默认打开位置
type?	`directory	file` 是选择文件夹还是文件
multi?	boolean	是否可以多选
filters?	object array	过滤弹窗可选择的对象
extensions?	string array	设置可选择的文件扩展名，例如 'png'

SaveDialogReturnValue

Name	Type	Description
canceled	boolean	用户是否取消
filePath	string	选择的文件路径

OpenDialogReturnValue

Name	Type	Description
canceled	boolean	用户是否取消
filePaths	string array	选择的文件路径

filters 示例

```
{
  filters: [
    { name: 'Images', extensions: ['jpg', 'png', 'gif'] },
    { name: 'Movies', extensions: ['mkv', 'avi', 'mp4'] },
    { name: 'Custom File Type', extensions: ['as'] },
    { name: 'All Files', extensions: ['*'] }
  ]
}
```

函数

info

▸ **info**(message: `string`, options?: `MessageDialogOptions`): Promise<`MessageBoxReturnValue`>

普通信息弹窗

请求参数

Name	Type	Description
message	string	显示的消息
options?	MessageDialogOptions	信息弹窗可选参数

返回结果

Promise<`MessageBoxReturnValue`>

```
const result = await Editor.Dialog.info('Dialog Message', {
    buttons: ['confirm', 'cancel'],
    title: 'Dialog Title',
});
if (0 == result.response) {
    // ... confirm event
} else {
    // ... cancel event
}
```

warn

▸ **warn**(message: `string`, options?: `MessageDialogOptions`): Promise<`MessageBoxReturnValue`>

警告弹窗

请求参数

Name	Type	Description
message	string	警告信息
options?	MessageDialogOptions	警告弹窗可选参数

返回结果

Promise<`MessageBoxReturnValue`>

```
await Editor.Dialog.warn('Warn Message');
```

error

▸ **error**(message: `string`, options?: `MessageDialogOptions`): Promise<`MessageBoxReturnValue`>

错误弹窗

请求参数

Name	Type	Description
message	string	错误信息
options?	MessageDialogOptions	错误弹窗可选参数

返回结果

Promise<`MessageBoxReturnValue`>

```
await Editor.Dialog.error('error content', {
    title: 'options-title'
});
```

save

▸ **save**(options?: `MessageDialogOptions`): Promise<`SaveDialogReturnValue`>

保存文件弹窗，保存文件时只能选择文件夹，且无法多选，相关参数不会生效

请求参数

Name	Type	Description
options?	SelectDialogOptions	保存文件窗口参数

返回结果

Promise<`SaveDialogReturnValue`>

```
const result = await Editor.Dialog.save({
    path: Editor.Project.path,
    title: 'Save Title',
    filters: [
        { name: 'Package', extensions: ['zip'] },
    ],
});
if (!result.filePath) {
    return;
}
```

select

▸ **select**(options?: `SelectDialogOptions`): Promise<`OpenDialogReturnValue`>

选择文件弹窗

请求参数

Name	Type	Description
options?	SelectDialogOptions	选择弹窗参数

返回结果

Promise<`OpenDialogReturnValue`>

```
const result = await Editor.Dialog.select({
    title: 'Select Title',
    path: aEditor.Project.path,
    filters: [{ name: 'Package', extensions: ['zip'] }],
});
if (result.filePaths && result.filePaths[0]) {
    return result.filePaths[0];
} else {
    return '';
}
```

更多说明请参考 Electron 官方文档

I18n

I18n

本地化翻译，通过在扩展中注册对应的数据，可以使用 i18n 进行翻译

接口说明

```
export type I18nMap = {
    [key: string]: I18nMap | string;
};
```

函数

getLanguage

▸ **getLanguage**(): `any`

获取当前的语言

返回结果

value	Type	Description
zh	string	中文
en	string	English

```
const langeage = Editor.I18n.getLanguage(); // "zh"
```

select

▸ **select**(language: `string`): `any`

选择一种翻译语言

请求参数

Name	Type	Description
language	string	选择当前使用的语言

```
Editor.I18n.select('zh');
```

t

▸ **t**(key: string, obj?: {[key: string]: string}): any

传入 key，翻译成当前语言 允许翻译变量 {a}，传入的第二个参数 obj 内定义 a

请求参数

Name	Type	Description
key	string	用于翻译的 key 值
obj?	Object	翻译字段内如果有 {key} 等可以在这里传入替换字段

```
/*
* zh.js 文件定义的翻译映射数据
* showUuid: '复制并打印 UUID'
* cancelSearchType: '取消搜索类型 {type}，默认 搜索名称或 UUID',
*/

Editor.I18n.t('hierarchy.menu.showUuid');  // '复制并打印 UUID'
Editor.I18n.t('hierarchy.menu.cancelSearchType', { type: 'UUID' });  // '取消搜索类型 UUID，默认 搜索名称或 UUID'
```

Logger

Logger

Creator 日志管理

函数

clear

▸ **clear**(): any

清空所有的日志

```
Editor.Logger.clear();
```

query

▸ **query**(): any

查询所有日志

```
const list = await Editor.Logger.query();
```

Message

Message

Creaotr 消息系统，消息在 Creator 非常重要，几乎所有的操作和数据传递都是通过消息进行的。

具体扩展支持调用的消息可以在菜单栏 开发者->消息列表 查看。

函数

send

▸ **send**(name: string, message: string, ...args: any[]): void

发送一个消息，没有返回

请求参数

Name	Type	Description
name	string	目标扩展的名字
message	string	触发消息的名字
...args	any[]	消息需要的参数

```
Editor.Message.send('builder', 'open-devtools');
```

request

▸ **request**(name: string, message: string, ...args: any[]): Promise<any>

发送一个消息，并等待返回

请求参数

Name	Type	Description
name	string	目标扩展的名字
message	string	触发消息的名字
...args	any[]	消息需要的参数

返回结果

Promise<any>

```
const sceneDirty = await Editor.Message.request('scene', 'query-dirty');  // false
```

broadcast

▸ **broadcast**(message: string, ...args: any[]): void

广播一个消息

我们约定每个扩展自己发出的广播消息，将以 [扩展名]:xxx 的形式命名，希望后续其他各个扩展也可以遵守这样的规范

请求参数

Name	Type	Description
message	string	消息的名字
...args	any[]	消息附加的参数

```
Editor.Message.broadcast('console:update-log-level', []);
```

addBroadcastListener（废弃）

▸ **addBroadcastListener**(message: `string`, func: `Function`): `any`

废弃警告，请通过扩展监听广播消息 新增一个广播消息监听器，不监听的时候，需要主动取消监听

请求参数

Name	Type	Description
message	string	消息名
func	Function	处理函数

```
Editor.Message.addBroadcastListener('console:logsUpdate', () => {});
```

removeBroadcastListener（废弃）

▸ **removeBroadcastListener**(message: `string`, func: `Function`): `any`

废弃警告，请通过扩展监听广播消息 新增一个广播消息监听器

请求参数

Name	Type	Description
message	string	消息名
func	Function	处理函数

```
Editor.Message.removeBroadcastListener('console:logsUpdate', () => {});
```

Network

Network

Creator 网络工具函数

函数

get

▸ **get**(url: `string`, data?: `Object`): Promise<`Buffer`>

Get 方式请求某个服务器数据

请求参数

Name	Type	Description
url	string	请求的 url
data?	Object	请求时带上的数据

返回结果

Promise<`Buffer`>

```
network.get(RUNTIME_REQUEST_URL).then((ret: any) => {
    ret = ret.toString();
}).catch((e: any) => {
    console.error('error', e);
});
```

post

▸ **post**(url: `string`, data?: `Object`): Promise<`Buffer`>

Post 方式请求某个服务器数据

请求参数

Name	Type	Description
url	string	请求的 url
data?	Object	请求时带上的数据

返回结果

Promise<`Buffer`>

```
let res: Buffer = await Editor.Network.post('https://creator-api.cocos.com/api/session/token', {
    ip: '127.0.0.1',
    client_type: 1
});
```

portIsOccupied

▸ **portIsOccupied**(port: `number`): Promise<`boolean`>

检查一个端口是否被占用

请求参数

Name	Type	Description
port	number	端口号

返回结果

Promise<`boolean`>

```
const isOccupied = await Editor.Network.portIsOccupied(8000);  // false
```

queryIPList

▸ **queryIPList**(): `string[]`

查询当前电脑的 ip 列表

返回结果

`string[]`

```
const ipList = Editor.Network.queryIPList();  // ["127.0.0.1", "192.168.52.154"]
```

testConnectServer

▸ **testConnectServer**(): Promise<`boolean`>

测试是否可以联通 passport.cocos.com 服务器

返回结果

Promise<`boolean`>

```
const res = await Editor.Network.testConnectServer();  // true
```

testHost

▸ **testHost**(ip: `string`): Promise<`boolean`>

测试是否可以联通某一台主机

请求参数

Name	Type	Description
ip	`string`	ip 地址

返回结果

Promise<`boolean`>

```
const res = await Editor.Network.testHost('127.0.0.1');  // true
```

Package

Package

扩展管理器

接口说明

```
interface GetPackageOptions {
    name?: string;
    debug?: boolean;
    path?: string;
    enable?: boolean;
    invalid?: boolean;
}

interface EditorInterfacePackageInfo {
    debug: boolean;
    invalid: boolean;
    enable: boolean;
    name: string;
    path: string;
    version: string;
    info: PackageJson;
}

type PathType = 'home' | 'data' | 'temp';
```

函数

disable

▸ **disable**(path: `string`): `any`

关闭一个扩展

请求参数

Name	Type	Description
path	`string`	扩展所在路径
options	`any`	关闭时带上的配置

```
Editor.Package.disable('D:\\Program\\CocosEditor\\Creator\\3.4.0\\resources\\app.asar\\builtin\\assets', {});
```

enable

▸ **enable**(path: `string`): `any`

启动一个扩展

请求参数

Name	Type	Description
path	`string`	扩展所在路径

```
Editor.Package.enable('D:\\Program\\CocosEditor\\Creator\\3.4.0\\resources\\app.asar\\builtin\\assets', {});
```

getPackages

▸ **getPackages**(options?: `GetPackageOptions`): `EditorInterfacePackageInfo[]`

查询扩展列表

请求参数

Name	Type	Description
options?	`GetPackageOptions`	查询条件

返回结果

`EditorInterfacePackageInfo[]`

```
const pkgs = Editor.Package.getPackages({ enable: true });
```

getPath

▸ **getPath**(extensionName: `string`, type?: `PathType`): `any`

获取一个扩展的几个预制目录地址

请求参数

Name	Type	Description
`extensionName`	`string`	扩展的名字
`type?`	`PathType`	地址类型（temp 临时目录，data 需要同步的数据目录，不传则返回现在打开的扩展路径）

```
const path = Editor.Package.getPath('menu');  // "D:\\Program\\CocosEditor\\Creator\\3.4.0\\resources\\app.asar\\builtin\\menu"
```

Panel

Panel

面板管理器

函数

open

▸ **open**(name: `string`, ...args: `any[]`): `any`

传入面板名字，打开一个面板

请求参数

Name	Type	Description
`name`	`string`	面板名称
`...args`	`any[]`	打开面板时传递的参数

```
Editor.Panel.open('console');
```

close

▸ **close**(name: `string`): `any`

传入面板名字，关闭同名的面板

请求参数

Name	Type	Description
`name`	`string`	面板名称

```
Editor.Panel.close('package-asset.import');
```

focus

▸ **focus**(name: `string`): `any`

将焦点传递给找到的第一个同名面板

请求参数

Name	Type	Description
`name`	`string`	面板名称

```
Editor.Panel.focus('assets');
```

has

▸ **has**(name: `string`): Promise<`boolean`>

检查面板是否已经打开

请求参数

Name	Type	Description
`name`	`string`	面板名称

返回结果

Promise<`boolean`>

```
const res = await Editor.Panel.has('package-asset.import');
```

define

▸ **define**(options: `Options`): `PanelObject`

定义一个面板，如果我们用 typescript 书写面板内容，ready 等生命周期函数内无法解析出正确的 this 对象，所以 Creator 里提供了一个 define 函数。

这个函数传入一个 PanelObject，返回一个 PanelObject，并不进行逻辑处理。但在这个函数传入的 PanelObject 上，能够正常识别出 this 对象。

请求参数

Name	Type	Description
`options`	`Options`<`Selector`, `M`, `U`>	面板的配置

```
module.exports = Editor.Panel.define({
    template: '<div id="app"></div>',
    $: {
        app: '#app',
    },
    methods: {
        init() { },
    },
    async ready() {
        new App({
            el: this.$.app,
        });
    },
});
```

Profile

Profile

配置

接口说明

```
type preferencesProtocol = 'default' | 'global' | 'local';
```

```
type projectProtocol = 'default' | 'project';
```

函数

getConfig

▸ **getConfig**(name: `string`, key?: `string`, type?: `preferencesProtocol`): Promise<`any`>

读取扩展配置

请求参数

Name	Type	Description
name	string	扩展名
key?	string	配置路径
type?	preferencesProtocol	配置的类型，选填

返回结果

Promise<`any`>

```
const value = await Editor.Profile.getConfig('asset-db', 'autoScan');
```

setConfig

▸ **setConfig**(name: `string`, key: `string`, value: `any`, type?: `preferencesProtocol`): Promise<`void`>

设置扩展配置

请求参数

Name	Type	Description
name	string	扩展名
key	string	配置路径
value	any	配置的值
type?	preferencesProtocol	配置的类型，选填

返回结果

Promise<`void`>

```
await Editor.Profile.setConfig('package-asset', 'import-path', dirname(result.filePaths[0]));
await Editor.Profile.setConfig('reference-image', 'show', true);
```

removeConfig

▸ **removeConfig**(name: `string`, key: `string`, type?: `preferencesProtocol`): Promise<`void`>

删除某个扩展配置

请求参数

Name	Type	Description
name	string	扩展名
key	string	配置路径
type?	preferencesProtocol	配置的类型，选填

返回结果

Promise<`void`>

```
await Editor.Profile.removeConfig('device', 'enable', 'global');
```

getProject

▸ **getProject**(`name`:string, `key?`:string, `type?`:projectProtocol) : `Promise`<any`>

读取扩展内的项目配置

请求参数

Name	Type	Description
name	string	扩展名
key?	string	配置路径
type?	projectProtocol	配置的类型，选填

返回结果

Promise<`any`>

```
const engineModules = await Editor.Profile.getProject('engine', 'modules.includeModules');
```

setProject

▸ **setProject**(name: `string`, key: `string`, value: `any`, type?: `preferencesProtocol`): Promise<`void`>

设置扩展内的项目配置

请求参数

Name	Type	Description
name	string	扩展名
key	string	配置路径
value	any	配置的值

Name	Type	Description
type?	projectProtocol	配置的类型，选填

返回结果

Promise<void>

```
await Editor.Profile.setProject('node-library', 'custom', {});
```

removeProject

▸ **removeProject**(name: string, key: string, type?: projectProtocol): Promise<void>

删除扩展内的项目配置

请求参数

Name	Type	Description
name	string	扩展名
key	string	配置路径
type?	projectProtocol	配置的类型，选填

返回结果

Promise<void>

```
await Editor.Profile.removeProject('engine', 'modules.includeModules');
```

getTemp

▸ **getTemp**(name: string, key?: string): Promise<any>

读取扩展配置

请求参数

Name	Type	Description
name	string	扩展名
key?	string	配置路径，选填

返回结果

Promise<any>

```
const state = await Editor.Profile.getTemp('assets', 'state');
```

setTemp

▸ **setTemp**(name: string, key: string, value: any): Promise<void>

设置扩展配置

请求参数

Name	Type	Description
name	string	扩展名
key	string	配置路径
value	any	配置的值

返回结果

Promise<void>

```
Editor.Profile.setTemp('assets', 'state', {});
```

removeTemp

▸ **removeTemp**(name: string, key: string): Promise<void>

删除某个扩展配置

请求参数

Name	Type	Description
name	string	扩展名
key	string	配置路径

返回结果

Promise<void>

```
await Editor.Profile.removeTemp('assets', 'state');
```

migrateProject

▸ **migrateProject**(pkgName: string, profileVersion: string, profileData: any): void

迁移扩展某个版本的项目配置数据到编辑器最新版本

请求参数

Name	Type	Description
pkgName	string	扩展名
profileVersion	string	要迁移的扩展版本号
profileData	any	迁移的数据

```
await Editor.Profile.migrateProject('builder', '1.2.1', buildJson);
```

migrateGlobal

▸ **migrateGlobal**(pkgName: string, profileVersion: string, profileData: any): void

迁移扩展某个版本的全局配置数据到编辑器最新版本

请求参数

Name	Type	Description

Name	Type	Description
pkgName	string	扩展名
profileVersion	string	要迁移的扩展版本号
profileData	any	迁移的数据

```
// const buildJson = { xxx };
await Editor.Profile.migrateGlobal('builder', '1.2.1', buildJson);
```

migrateLocal

▸ **migrateLocal**(pkgName: `string`, profileVersion: `string`, profileData: `any`): `void`

迁移扩展某个版本的本地配置数据到编辑器最新版本

请求参数

Name	Type	Description
pkgName	string	扩展名
profileVersion	string	要迁移的扩展版本号
profileData	any	迁移的数据

```
await Editor.Profile.migrateLocal('builder', '1.2.1', buildJson);
```

Project

Project

当前打开的项目基本信息

变量

name

• **name**: `string`

当前项目名称(取自 package.json)

```
const projectName = Editor.Project.name; // "Hello World 3.4.0"
```

path

• **path**: `string`

当前项目路径

```
const projectPath = Editor.Project.path;  // "E:\\workSpace\\Hello World 3.4.0"
```

tmpDir

• **tmpDir**: `string`

当前项目临时文件夹

```
const projectTmpDir = Editor.Project.tmpDir;  // "E:\\workSpace\\Hello World 3.4.0\\temp"
```

uuid

• **uuid**: `string`

当前项目 uuid

```
const projectUUID = Editor.Project.uuid;  // "7aa7c089-8e53-4611-8689-98b69ab28e22"
```

Selection

Selection

Creator 选择管理器，编辑器内所有的选中物体，都通过这个管理器进行管理

选中物体需要记录 "类型" 和 "ID" 两个属性

函数

clear

▸ **clear**(type: `string`): `any`

清空一个类型的所有选中元素

如果有元素被取消选中，会发送 selection:unselect 广播消息

请求参数

Name	Type	Description
type	string	选中的类型

```
Editor.Selection.clear('asset');
```

hover

▸ **hover**(type: `string`, uuid?: `string`): `any`

悬停触碰了某个元素

会发出 selection:hover 的广播消息

请求参数

Name	Type	Description
type	string	选中的类型
uuid?	string	元素的 uuid

```
Editor.Selection.hover('asset', '7bf9df40-4bc9-4e25-8cb0-9a500f949102');
```

select

▸ **select**(type: `string`, uuid:`string | string[]`): `any`

选中一个或者一组元素

当没有选中的元素变成选中状态时，会发出 selection:select 的广播消息

请求参数

Name	Type	Description
type	string	选中的类型
uuid	`string	string[]`	元素的 uuid

```
Editor.Selection.select('asset', '7bf9df40-4bc9-4e25-8cb0-9a500f949102');
```

unselect

▸ **unselect**(type: `string`, uuid:`string | string[]`): `any`

取消一个或者一组元素的选中状态

当元素被取消选中的时候，会发送 selection:unslect 广播消息

请求参数

Name	Type	Description
type	string	选中的类型
uuid	`string	string[]`	元素的 uuid

```
Editor.Selection.unselect('asset', '7bf9df40-4bc9-4e25-8cb0-9a500f949102');
```

getSelected

▸ **getSelected**(type: `string`): `string[]`

获取一个类型选中的所有元素数组

请求参数

Name	Type	Description
type	string	选中的类型

返回结果

```
string[]
```

```
const uuids = Editor.Selection.getSelected('asset');  // ["b0a4abb1-db32-49c3-9e09-a45b922a2024"]
```

getLastSelected

▸ **getLastSelected**(type: `string`): `string`

获取某个类型内，最后选中的元素

请求参数

Name	Type	Description
type	string	选中的类型

返回结果

```
string
```

```
const elem = Editor.Selection.getLastSelected('asset');  // "b0a4abb1-db32-49c3-9e09-a45b922a2024"
```

getLastSelectedType

▸ **getLastSelectedType**(): `string`

获取最后选中的元素的类型

返回结果

```
string
```

```
const type = Editor.Selection.getLastSelectedType();  // "asset"
```

Utils

Utils

Creator 内置的一些工具函数

File

接口说明

```
interface UnzipOptions {
    peel?: boolean;
}
```

函数

copy

▸ **copy**(source: `string`, target: `string`): `void`

复制一个文件到另一个位置

请求参数

Name	Type	Description
source	string	
target	string	

```
Editor.Utils.File.copy('C:\\CocosCreatorWorkSpace\\HelloWorld', 'E:\\CocosCreatorWorkSpace\\HelloWorld');
```

getName

▸ **getName**(file: `string`): `string`

初始化一个可用的文件名，返回可用名称的文件路径 遇到重名时会自动递增直到获取到一个不会冲突的可用文件地址

Parameters

Name	Type	Description
file	string	初始文件路径

返回结果

`string`

```
const newFileName = Editor.Utils.File.getName('E:\\CocosCreatorWorkSpace\\HelloWorld');
```

unzip

▸ **unzip**(zip: `string`, target: `string`, options?: `UnzipOptions`): Promise<`void`>

解压文件夹

请求参数

Name	Type	Description
zip	string	
target	string	
options?	UnzipOptions	

返回结果

Promise<`void`>

```
await Editor.Utils.File.unzip('E:\\repositories\\utils.zip', 'E:\\CocosCreatorWorkSpace\\HelloWorld');
```

Math

函数

add

▸ **add**(arg1: `string | number`, arg2: `string | number`): `number`

加法函数 入参：函数内部转化时会先转字符串再转数值，因而传入字符串或 number 均可 返回值：arg1 加上 arg2 的精确结果

请求参数

Name	Type	Description
arg1	string | number	
arg2	string | number	

返回结果

`number`

```
const res = Editor.Utils.Math.add('123', 12.12);  // 135.12
```

clamp

▸ **clamp**(val: `number`, min: `number`, max: `number`): `any`

取给定边界范围的值

请求参数

Name	Type	Description
val	number	
min	number	
max	number	

```
const res = Editor.Utils.Math.clamp(100, 1, 99);  // 99
```

clamp01

▸ **clamp01**(val: `number`): `number`

取给 0 - 1 范围内的值

请求参数

Name	Type	Description
val	number	

返回结果

`number`

```
const res = Editor.Utils.Math.clamp01(0.5);  // 0.5
```

sub

▸ **sub**(arg1: `string | number`, arg2: `string | number`): `number`

减法函数 入参：函数内部转化时会先转字符串再转数值，因而传入字符串或number均可 返回值：arg1 减 arg2的精确结果

请求参数

Name	Type	Description
arg1	string | number	
arg2	string | number	

返回结果

`number`

```
const res = Editor.Utils.Math.sub('123', 12.12);  // 110.88
```

toFixed

▸ **toFixed**(val: `number`, num: `number`): `number`

保留小数点

请求参数

Name	Type	Description
val	number	
num	number	

返回结果

`number`

```
const res = Editor.Utils.Math.toFixed(12.1294, 2);  // 12.13
```

Path

变量

delimiter

• **delimiter**: `";" | ":"`

sep

• **sep**: `"\\" | "/"`

函数

basename

▸ **basename**(p: `string`, ext?: `string`): `string`

返回路径的最后一部分

请求参数

Name	Type	Description
p	string	路径
ext?	string	文件扩展名，可选

返回结果

`string`

```
const fileName = Editor.Utils.Path.basename('E:\\CocosCreatorWorkSpace\\HelloWorld\\package.json', '.json');  // package
```

basenameNoExt

▸ **basenameNoExt**(path: `string`): `string`

返回一个不含扩展名的文件名

请求参数

Name	Type	Description
path	string	

返回结果

`string`

```
const fileName = Editor.Utils.Path.basenameNoExt('E:\\CocosCreatorWorkSpace\\HelloWorld\\package.json');  // package
```

contains

▸ **contains**(pathA: `string`, pathB: `string`): `boolean`

判断路径 pathA 是否包含 pathB

请求参数

Name	Type	Description
pathA	string	
pathB	string	

返回结果

`boolean`

```
const res = Editor.Utils.Path.contains('foo/bar', 'foo/bar/foobar');  // true
const res = Editor.Utils.Path.contains('foo/bar', 'foo/bar');  // true
const res = Editor.Utils.Path.contains('foo/bar/foobar', 'foo/bar');  // false
const res = Editor.Utils.Path.contains('foo/bar/foobar', 'foobar/bar/foo');  // false
```

dirname

▸ **dirname**(p: `string`): `string`

返回路径的目录名

请求参数

Name	Type	Description
p	string	

返回结果

`string`

```
const dirname = Editor.Utils.Path.dirname('E:\\CocosCreatorWorkSpace\\HelloWorld\\package.json');
// "E:\\CocosCreatorWorkSpace\\HelloWorld"
```

extname

▸ **extname**(p: `string`): `string`

返回路径的扩展名

请求参数

Name	Type	Description
p	string	

返回结果

`string`

```
const extname = Editor.Utils.Path.extname('E:\\CocosCreatorWorkSpace\\HelloWorld\\package.json');  // .json
```

format

▸ **format**(p: `FormatInputPathObject`): `string`

根据对象数据返回路径字符串

请求参数

Name	Type	Description
p	FormatInputPathObject	

返回结果

`string`

```
const path = Editor.Utils.Path.format({
  root: '/ignored',
  dir: '/home/user/dir',
  base: 'file.txt'
});  // "/home/user/dir\\file.txt"
```

isAbsolute

▸ **isAbsolute**(p: `string`): `boolean`

判断路径是否为绝对路径

请求参数

Name	Type	Description
p	string	

返回结果

`boolean`

```
const res = Editor.Utils.Path.isAbsolute('E:\\CocosCreatorWorkSpace\\HelloWorld\\package.json');  // true
```

join

▸ **join**(...paths: `string[]`): `string`

使用特定于平台的分隔符作为定界符将所有给定的 `path` 片段连接在一起，然后规范化生成的路径

请求参数

Name	Type	Description
...paths	string[]	

返回结果

`string`

```
const path = Editor.Utils.Path.join('/foo', 'bar', 'abc/def', 'g');  // "\\foo\\bar\\abc\\def\\g"
```

normalize

▸ **normalize**(path: `string`): `string`

格式化路径 如果是 Windows 平台，需要将盘符转成小写进行判断

请求参数

Name	Type	Description
path	string	

返回结果

`string`

```
const path = Editor.Utils.Path.normalize('/foo/bar//abc/def/g');  // "\\foo\\bar\\abc\\def\\g"
```

parse

▸ **parse**(p: `string`): `ParsedPath`

返回一个对象，表示路径的重要信息

请求参数

Name	Type	Description
p	string	

返回结果

`ParsedPath`

```
const res = Editor.Utils.Path.parse('/foo/bar//abc/def/g.txt');
```

relative

▸ **relative**(from: `string`, to: `string`): `string`

根据当前工作目录返回从 `from` 到 `to` 的相对路径

请求参数

Name	Type	Description
from	string	

Name	Type	Description
to	string	

返回结果

`string`

```
const relativePath = Editor.Utils.Path.relative('C:\\program\\test\\aaa', 'C:\\program\\impl\\bbb');  // "..\\..\\impl\\bbb"
```

resolve

▸ **resolve**(...pathSegments: `string[]`): `string`

将路径或路径片段的序列解析为绝对路径

请求参数

Name	Type	Description
...pathSegments	string[]	

返回结果

`string`

```
const path = Editor.Utils.Path.resolve('/foo/bar', './abc');  // "D:\\foo\\bar\\abc"
```

slash

▸ **slash**(path: `string`): `string`

将 \ 统一换成 /

请求参数

Name	Type	Description
path	string	

返回结果

`string`

```
const path = Editor.Utils.Path.slash('\\foo\\bar');  // "/foo/bar"
```

stripExt

▸ **stripExt**(path: `string`): `string`

删除一个路径的扩展名

请求参数

Name	Type	Description
path	string	

返回结果

`string`

```
const path = Editor.Utils.Path.stripExt('E:\\HelloWorld\\package.json');  // "E:\\HelloWorld\\package"
```

stripSep

▸ **stripSep**(path: `string`): `string`

去除路径最后的斜杆，返回一个不带斜杆的路径

请求参数

Name	Type	Description
path	string	

返回结果

`string`

```
const path = Editor.Utils.Path.stripSep('E:\\HelloWorld\\package.json\\');  // "E:\\HelloWorld\\package.json"
```

Url

函数

getDocUrl

▸ **getDocUrl**(relativeUrl: `string`, type?: `"manual" | "api"`): `string`

快捷获取文档路径

请求参数

Name	Type	Description
relativeUrl	string	
type?	`"manual" | "api"`	

返回结果

`string`

```
const url = Editor.Utils.Url.getDocUrl('publish/publish-bytedance.html');
// "https://docs.cocos.com/creator/3.4/manual/zh/publish/publish-bytedance.html"
```

UUID

函数

compressUUID

▸ **compressUUID**(uuid: `string`, min: `boolean`): `string`

压缩 UUID

请求参数

Name	Type	Description
uuid	string	
min	boolean	

返回结果

`string`

```
const uuid = Editor.Utils.UUID.compressUUID('7bf9df40-4bc9-4e25-8cb0-9a500f949102');  // "7bf9d9AS8lOJYywmlAPlJEC"
```

decompressUUID

▸ **decompressUUID**(str: `string`): `string`

解压 UUID

请求参数

Name	Type	Description
str	string	

返回结果

`string`

```
const uuid = Editor.Utils.UUID.decompressUUID('7bf9d9AS8lOJYywmlAPlJEC');  // "7bf9df40-4bc9-4e25-8cb0-9a500f949102"
```

generate

▸ **generate**(): `string`

生成一个新的 uuid

返回结果

`string`

```
const uuid = Editor.Utils.UUID.generate();
```

isUUID

▸ **isUUID**(str: `string`): `string`

检查输入字符串是否是 UUID

请求参数

Name	Type	Description
str	string	

返回结果

`string`

```
const isUUID = Editor.Utils.UUID.isUUID('7bf9d9AS8lOJYywmlAPlJEC');  // true
```

进阶主题

进阶主题

如何向 Cocos 提交代码

如何向 Cocos 提交代码

Cocos Creator 是一个开源引擎，连同范例、文档都是开源的。
在你开发游戏的过程中，当发现了引擎、文档或者范例不够完善的地方，如果仅仅是向官方团队提出建议，官方团队可能会因为人力资源的紧张而无法及时跟进。在此我们欢迎所有用户主动向我们提交 PR，帮助 Cocos 越做越好。引擎有 Bug？提 PR！范例难看？提 PR！API 注释不清晰？提 PR！文档有错别字？提 PR！想要把你的宝贵修改贡献给游戏社区？提 PR！以下几个是目前官方比较常用的开源仓库，这些仓库都可以提交 PR。

- **Cocos 引擎官方仓库：** GitHub | Gitee

下面让我们来看一下，如何从零开始在 GitHub 上向 Cocos 提交代码。

注册一个 GitHub 账号

打开 GitHub 网站 注册账号。若之前已经有注册过，那直接登录就可以了。

环境配置

安装 Git

首先先确认电脑是否已经安装 Git，命令行输入 `git`，安装过则会输出以下内容：

存储和读取用户数据

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 `localStorage` 接口来进行用户数据存储和读取的操作。

`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**（GitHub | Gitee）。

存储数据

```
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`：

```
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
};

sys.localStorage.setItem('userData', JSON.stringify(userData));
```

读取数据

```
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(sys.localStorage.getItem('userData'));
```

存储数据 和 **读取数据** 的完整示例代码参考如下：

```
import { _decorator, Component, sys } from 'cc';
const { ccclass, property } = _decorator;

const coinKey: string = 'gold'
const userDataKey = 'userData';

interface UserData{
    name:string
    level:number
    gold:number
}

@ccclass('LocalStorageExample')
export class LocalStorageExample extends Component {
    start() {
        this.saveCoin(100);
        this.loadCoin();
        this.saveUserData();
        this.loadUserData();
    }

    saveCoin(value: number) {
        sys.localStorage.setItem(coinKey, value.toString());
    }
    loadCoin() {
        const value = sys.localStorage.getItem(coinKey);
        if (value) {
            console.log(`${coinKey} = ${value}`)
        } else {
            console.log(`${coinKey} is not exist`);
        }
    }

    saveUserData(){
        const userData = { name:"Tracer", level:1, gold:100}
        const jsonStr = JSON.stringify(userData);
        sys.localStorage.setItem(userDataKey,jsonStr);
    }

    loadUserData(){
        const jsonStr = sys.localStorage.getItem(userDataKey);
        const userData = JSON.parse(jsonStr) as UserData;;
        console.log(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：

```
sys.localStorage.removeItem(key)
```

清空数据

当我们不再需要已存储的用户数据时，可以通过下面的接口将其清空：

```
sys.localStorage.clear()
```

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：

```
import encrypt from 'encryptjs';
var secretkey = 'open_sesame'; // 加密密钥
var dataString = JSON.stringify(userData);
var encrypted = encrypt.encrypt(dataString,secretkey,256);

sys.localStorage.setItem('userData', encrypted);
```

读取时：

```
var cipherText = 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

引擎定制工作流程

Cocos Creator 拥有两套引擎内核，C++ 内核 和 TypeScript 内核。C++ 内核用于原生平台，TypeScript 内核用于 Web 和小游戏平台。 在引擎内核之上，是用 TypeScript 编写的引擎框架层，用以统一两套内核的差异，让开发更便捷。

网页预览定制工作流程

网页预览定制工作流程

自定义预览模板

预览支持自定义模板方便用户自定义需要的预览效果，自定义的预览模板可以放置在项目目录的 `preview-template` 文件夹中。或者点击编辑器主菜单中的 **项目 -> 生成预览模板** 就可以在项目目录下创建一个最新的预览模板。编辑器中的预览也是使用模板来注入最新的项目数据，预览时将会查找该目录下的 index 文件，如果存在就是要该文件作为预览的模板。

`preview-template` 文件夹的结构类似：

```
project-folder
 |--assets
 |--build
 |--preview-template
     // 必须的入口文件
     |--index.ejs
     // 其他文件可根据想要实现的预览效果进行添加
```

开始自定义网页预览，需要注意的是，预览模板中存在一些预览菜单项以及预览调试工具等内容，所以在增删一些模板语法的内容时要稍加注意，如果随意修改可能会导致预览模板不可用。建议使用 ejs 注入的内容都保留，然后在此基础上添加需要的内容即可。另外，假如 `index.html` 与 `index.ejs` 共存时，**`index.html` 将会替代 `index.ejs`** 成为预览的页面内容。

使用示例

以下示例可以在 GitHub | Gitee 查找到。

1. 点击编辑器主菜单中的 **项目 -> 生成预览模板**，**控制台** 便会输出“预览模板生成成功”的提示，并显示预览模板的生成路径。
2. 添加需要使用的脚本如 `test.js`，其中 `<%- include(cocosTemplate, {}) %>` 中包含的是默认的启动游戏逻辑，添加的脚本可以根据需要在游戏逻辑启动前/后来决定存放的位置。下面的 `test.js` 是在游戏启动后加载。
    - 打开 `index.ejs` 修改如下：

      ```
      <html>
          ...
          <body>
              ...
              <%- include(cocosTemplate, {}) %> // 游戏启动处理逻辑
              <script src="/test.js"></script> // 新增脚本
          </body>
      </html>
      ```

    - `test.js` 放置在页面内标识的相对路径（只能在 `preview-template` 文件夹中）

      ```
      |--preview-template
              |--index.ejs
              |--test.js
      ```

---

i18n 游戏多语言支持

i18n 游戏多语言支持

Cocos Creator 从 3.1 版本起，提供了 i18n 游戏多语言扩展插件，目前支持 Label 和 Sprite 组件，可在 Cocos Store 中下载使用。

> **注意**：从 v3.6 起，引擎已内置 本地化（L10N） 提供功能更丰富、强大的多语言功能。如果您使用的是 v3.6 之后的版本我们建议您使用该功能。

什么是 i18n

i18n 是 **国际化** 的简称，来源是英文单词 **internationalization** 的首末字符 **i** 和 **n**，18 为单词中间的字符数。在资讯领域，国际化（i18n）可以让产品无需做大的改变，就能够满足不同语言和地区的需要。对程序来说，在不修改内部代码的情况下，就能根据不同语言及地区切换相应的语言界面。如今全球化的时代，国际化尤为重要，因为产品的潜在用户可能来自世界的各个角落。

通常与 i18n 相关的还有 L10n（“本地化”的简称），多语言国际化和本地化的区别在于：

- **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进行实时切换。
- **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。

> **注意**：i18n 游戏多语言扩展插件不提供构建项目时的多语言资源管理，若面向不同地区发布项目，需要移除一部分多语言资源时，请手动处理。

安装

1. 点击 Creator 顶部菜单栏中的 **扩展 -> 商城** 进入 **Cocos Store**，查找 i18n 多语言化扩展插件 或者直接在上方搜索框中搜索：

动态合图

动态合图

降低 DrawCall 是提升游戏渲染效率一个非常直接有效的办法，而两个 DrawCall 是否可以合并为一个 DrawCall 的其中一个重要因素就是这两个 DrawCall 是否使用了同一张贴图。

Cocos Creator 提供了 **动态合图**（Dynamic Atlas）的功能，它能在项目运行时动态地将贴图合并到一张大贴图中。当渲染一张贴图的时候，动态合图系统会自动检测这张贴图是否已经被合并到了图集（图片集合）中，如果没有，并且此贴图又符合动态合图的条件，就会将此贴图合并到图集中。

动态合图是按照 **渲染顺序** 来选取要将哪些贴图合并到一张大图中的，这样就能确保相邻的 DrawCall 能合并为一个 DrawCall（又称“合批”）。

启用、禁用动态合图

Cocos Creator 在初始化过程中，会根据不同的平台设置不同的 CLEANUP_IMAGE_CACHE 参数，当禁用 `CLEANUP_IMAGE_CACHE` 时，动态合图就会默认开启。
启用动态合图会占用额外的内存，不同平台占用的内存大小不一样。目前在小游戏和原生平台上默认会禁用动态合图，但如果你的项目内存空间仍有富余的话建议开启。

若希望强制开启动态合图，请在代码中加入：

```
macro.CLEANUP_IMAGE_CACHE = false;
dynamicAtlasManager.enabled = true;
```

> **注意**：这些代码请写在项目脚本中的最外层，不要写在 `onLoad/start` 等类函数中，才能确保在项目加载过程中即时生效。否则如果在部分贴图缓存已经释放的情况下才启用动态图集，可能会导致报错。

若希望强制禁用动态合图，可以直接通过代码控制：

```
dynamicAtlasManager.enabled = false;
```

贴图限制

动态合图系统限制了能够进行合图的贴图大小，默认只有贴图宽高都小于 **512** 的贴图才可以进入到动态合图系统。开发者可以根据需求修改这个限制：

```
dynamicAtlasManager.maxFrameSize = 512;
```

详情可在 API 文档中查找 `DynamicAtlasManager` 进行参考。

热更新范例教程

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 | Gitee 获取（master 分支）。

热更新管理器

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！

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 manifest 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：

HTTP 请求

HTTP 请求

在某些情况下，可能需要向网络后端请求一些数据，在 Cocos Creator 里面可以通过 `fetch` 方法，`fetch` 方法是 JavaScript 的一部分：

```
declare function fetch(input: RequestInfo | URL, init?: RequestInit): Promise<Response>;
```

搭建服务

首先通过自身熟悉的后端语言搭建一个简易的 http 服务，非后端开发人员可忽略本段。本示例中采用 golang 搭建，代码示例如下：

```
package main

import (
"fmt"
"log"
"net/http"
)

func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "http request received")
}

func main() {
    http.HandleFunc("/", handler)
    log.Fatal(http.ListenAndServe(":8080", nil))
}
```

上述代码会搭建一个监听在 8080 端口的网络服务器。

创建 Http 请求

在 Cocos Creator 里面通过 `fetch` 方法向服务器请求数据，此处以 `GET` 方法为例，并以文本格式返回服务器的数据，代码示例如下：

```
import { _decorator, Component } from 'cc';
const { ccclass, property } = _decorator;

@ccclass('HttpTest')
export class HttpTest extends Component {
    start() {

        fetch("http://127.0.0.1:8080").then((response: Response) => {
```

```
            return response.text()
        }).then((value) => {
            console.log(value);
        })
    }
}
```

运行场景后会打印如下日志：

```
> http request received.
```

也可通过 `respoonse.json()` 来获取 JSON 格式的返回。

您可以通过 MDN Web Doc 社区 来查看更多详细的信息。

WebSocket

WebSocket 简介

通常在开发网络应用或者游戏时，出于数据安全或其他目标，需要将用户的数据存放在网络后端服务端中，而浏览器一般来说只支持 HTTP 协议。

HTTP（HyperText Transfer Protocol）协议[1]是基于 TCP 协议的应用层传输协议，使用方法是通过客户端（网页浏览器或通过 API）向服务端请求数据，服务端将数据存储在 HTTP 协议的响应内，返回给客户端。

WebSocket 客户端

WebSocket 客户端

原生环境支持 Web 标准的 WebSocket 接口。

差异

在 v3.5 版本之前，Android 和 Windows 上的 WebSocket API 使用 libwebsockets 实现。需要开发者通过第三个参数指定 CA 文件路径：

```
this.wsInstance = new WebSocket('wss://echo.websocket.org', [], caURL);
```

详情请参考 示例代码。可访问 下载 CA 证书 下载对应的证书。

在 v3.5 版本之后, Android 上不再强制要求提供证书。

WebSocket 服务器

使用 WebSocket 服务器

开发者可以在游戏进程中启动一个 **WebSocket 服务器**，提供 **RPC 接口**。通过完善和调用这些 **RPC 接口**，开发者能够对游戏进程内部状态进行监控，增加对游戏进程状态的管理能力。

如何启用

WebSocket 服务器 默认是剔除的。若要启用，需要在编辑器顶部菜单栏中 **项目设置 -> 功能裁剪** 中勾选**WebSocket Server** 配置。

如何调用 WebSocket 服务器接口

可参考下方实例代码：

```
// 在原生平台的 Release 模式下或者在 Web /微信小游戏等平台中，WebSocketServer 可能没有定义
if (typeof WebSocketServer === "undefined") {
    console.error("WebSocketServer is not enabled!");
    return;
}

let s = new WebSocketServer();
s.onconnection = function (conn) {
    conn.onmessage = function (data) {
        conn.send(data, (err) => {});
    }
    conn.onclose = function () {
        console.log("connection gone!");
    };
};

s.onclose = function () {
  console.log("server is closed!");
}
s.listen(8080, (err) => {
   if (!err);
   console.log("server booted!");
});
```

API

接口定义如下：

```
/**
 * 服务器对象
 */
class WebSocketServer {
    /**
     * 关闭服务
     */
    close(cb?: WsCallback): void;
    /**
     * 监听并启动服务
     */
    listen(port: number, cb?: WsCallback): void;
    /**
     * 处理新的请求
     */
    set onconnection(cb: (client: WebSocketServerConnection) => void);
    /**
     * 设置服务器关闭回调
     */
    set onclose(cb: WsCallback);
    /**
     * 获取所有的连接对象
     */
    get connections(): WebSocketServerConnection[];
}

/**
 * 服务器中客户端的连接对象
 */
```

```
class WebSocketServerConnection {
    /**
     * 关闭连接
     */
    close(code?: number, reason?: string): void;
    /**
     * 发送数据
     */
    send(data: string|ArrayBuffer, cb?: WsCallback): void;

    set ontext(cb: (data: string) => void);
    set onbinary(cb: (data: ArrayBuffer) => void);
    set onmessage(cb: (data: string|ArrayBuffer) => void);
    set onconnect(cb: () => void;);
    set onclose(cb: WsCallback);
    set onerror(cb: WsCallback);

    get readyState(): number;
}

interface WsCallback {
    (err?: string): void;
}
```

注意: `ondata` 回调已经在 v3.7.0 版本废弃, 请使用 `onmessage` 代替.

参考链接

接口设计参考了 nodejs-websocket。
