Cocos2d-x UI ，调度器，帧动画

关于UI

UI

- 所有应用中都有UI
- 游戏中运用更丰富的UI

- 问题：
- UI到底代表什么？
- UI组件具体是怎么工作的？
- ……

UI

- UI含义：
- User Interface（用户界面）缩写

- 包括：
- Label、button、menu、slider、view等

- Cocos2d-x 提供易于使用的UI组件来满足GUI需求

Label标签

Label标签

- Cocos2d-x Label种类：

- LabelBMFont （ Bitmap Font ）
- LabelTTF （ True Type Font ）
- Label System Font （ System Font ）

Label BMFont

- BMFont 位图字体

Label BMFont

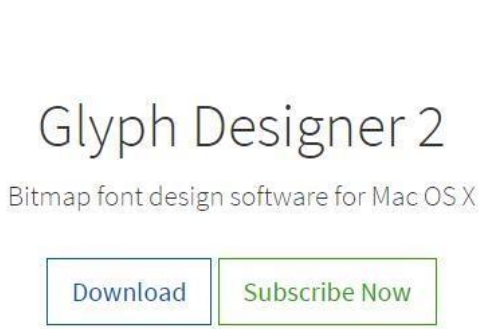
Label BMFont类

- 一个基于位图的字体图集，它允许字符从主图中剪切出来。
- 创建BMFont文本：

  .fnt文件（字体坐标文件）、对应.png图像（字体图像文件），两个文件名字需要一致

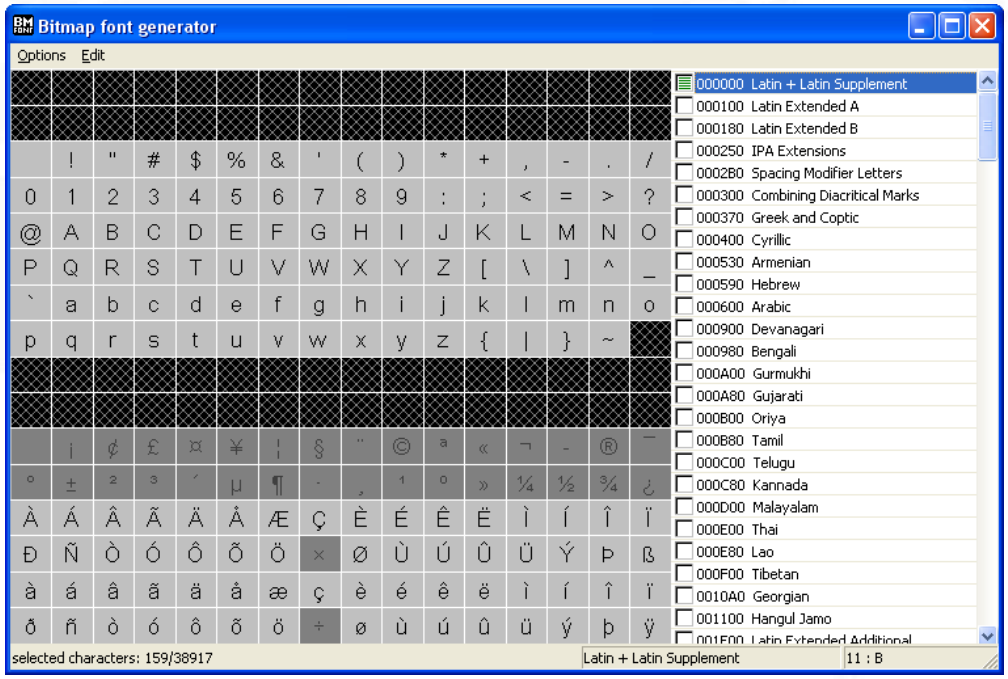
Label BMFont

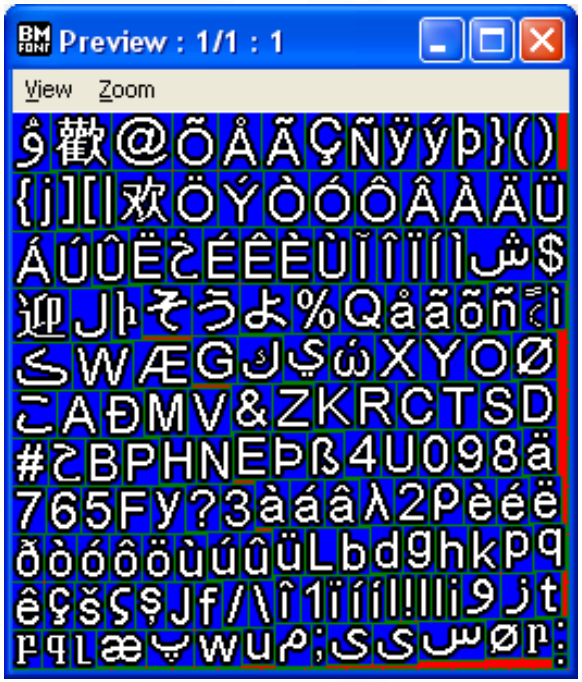
- 创建工具：
  Glyph Designer 2 (Mac)



Label BMFont

- 创建工具：
  Bitmap Font Generator (Windows)





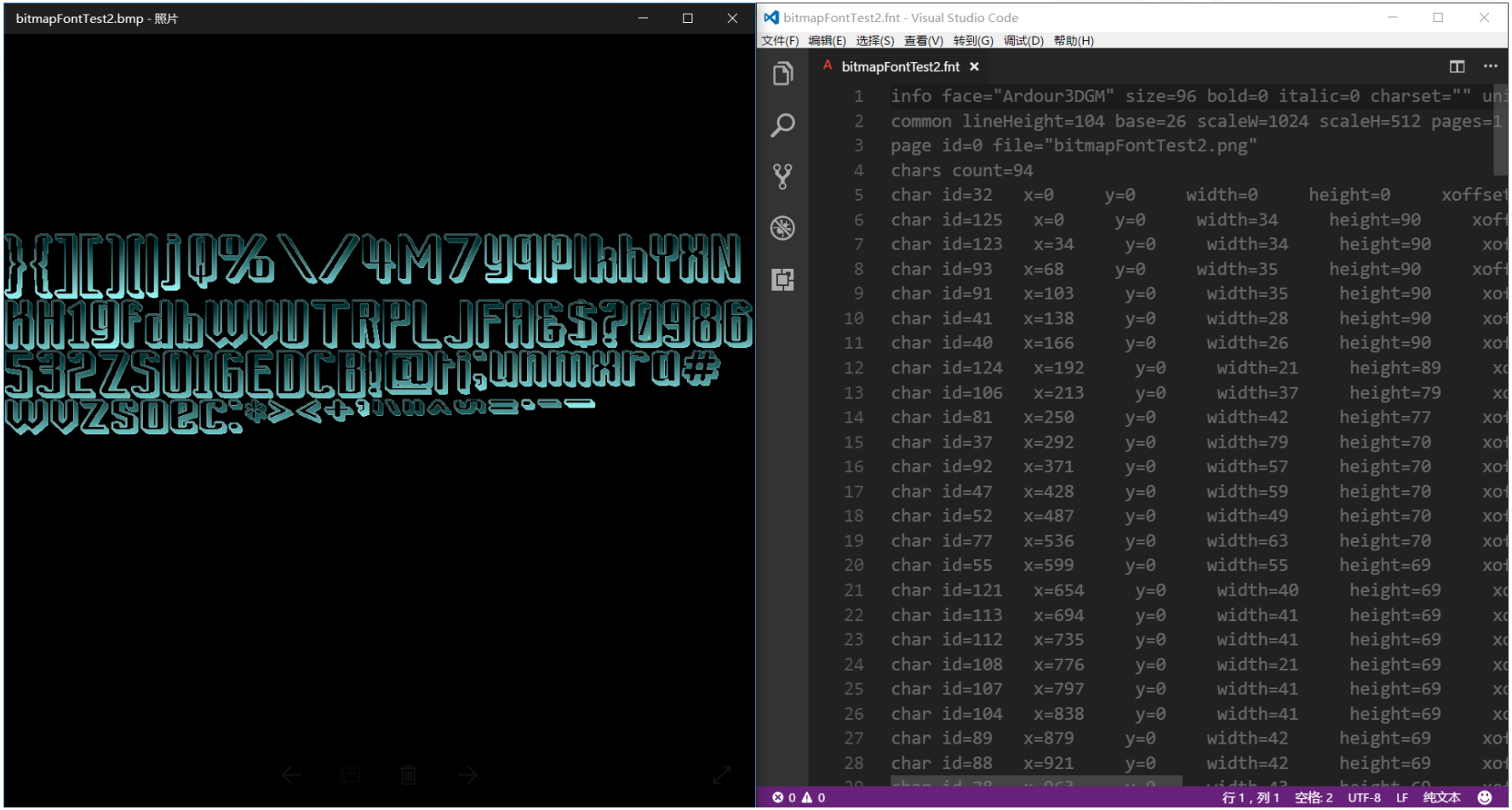
Label BMFont

- 创建一个BMFont的Label：

```
auto label2 = Label::createWithBMFont("fonts/bitmapFontTest2.fnt", "Hello World");
label2->setPosition(Vec2(winSize.width / 2, winSize.height * 0.6f));
addChild(label2);
```

Label BMFont

上图中的bitmapFontTest2.bmp与bitmapFontTest2.fnt



Label TTF

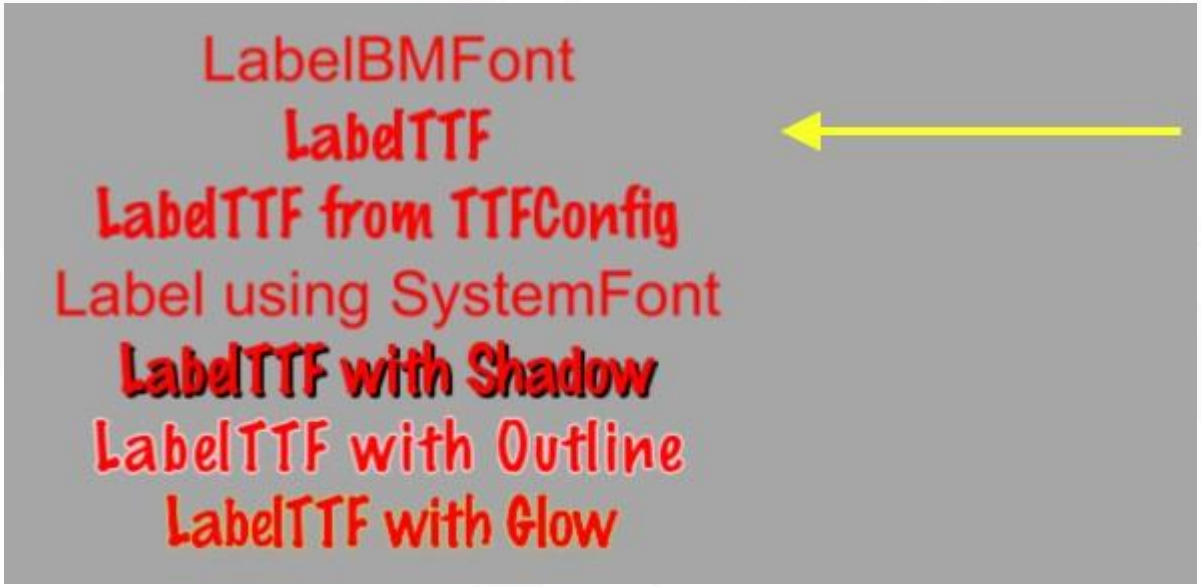
- TTF：True Type Font字体
- 创建需要：
  .ttf格式的字体文件名、文本字符串、字体大小

- 与BMFont不同：TTF可改变字体显示大小，无需单独字体

Label TTF

- 创建一个TTF的Label：

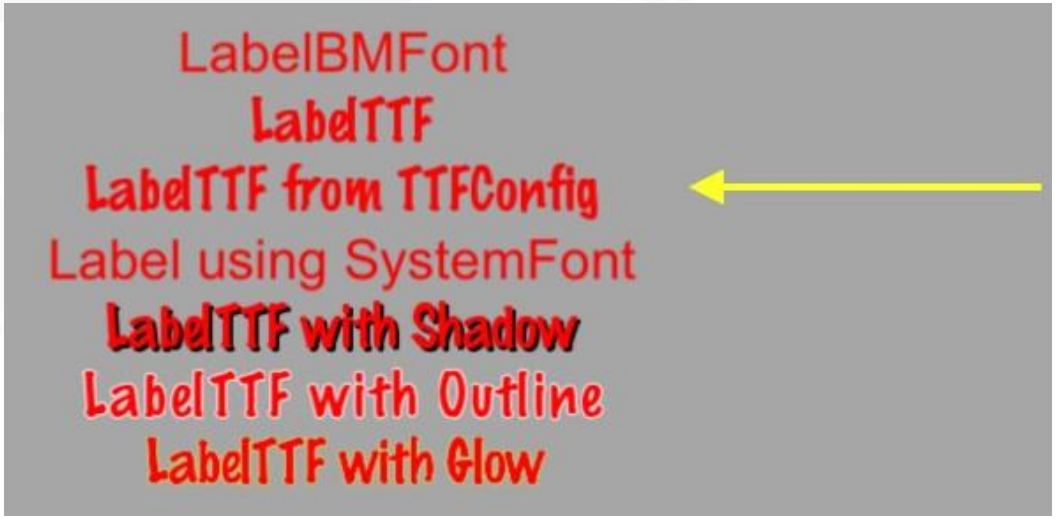
```
auto myLabel = Label::createWithTTF("Your Text", "Marker Felt.ttf", 24);
```



Label TTF

- 创建TTFConfig管理Label属性：

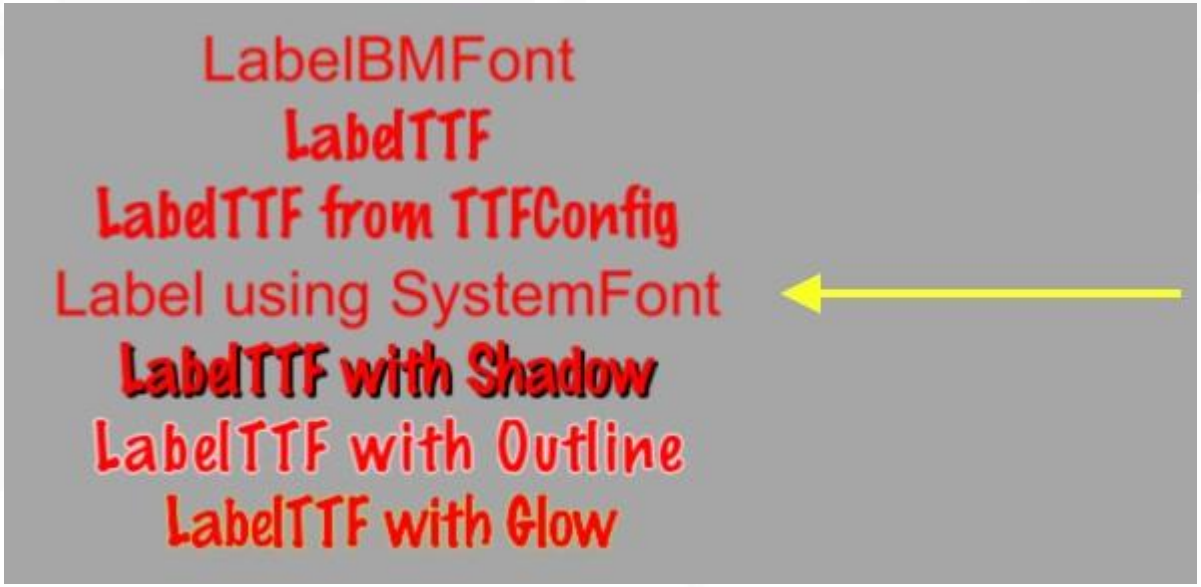
```
// create a TTFConfig files for labels to share
TTFConfig labelConfig;
labelConfig.fontFilePath = "myFont.ttf";
labelConfig.fontSize = 16;
labelConfig.glyphs = GlyphCollection::DYNAMIC;
labelConfig.outlineSize = 0;
labelConfig.customGlyphs = nullptr;
labelConfig.distanceFieldEnabled = false;

// create a TTF Label from the TTFConfig file.
auto myLabel = Label::createWithTTF(labelConfig, "My Label Text");
```



Label SystemFont

- 使用系统默认字体和尺寸的Label
- 创建一个SystemFont的Label：

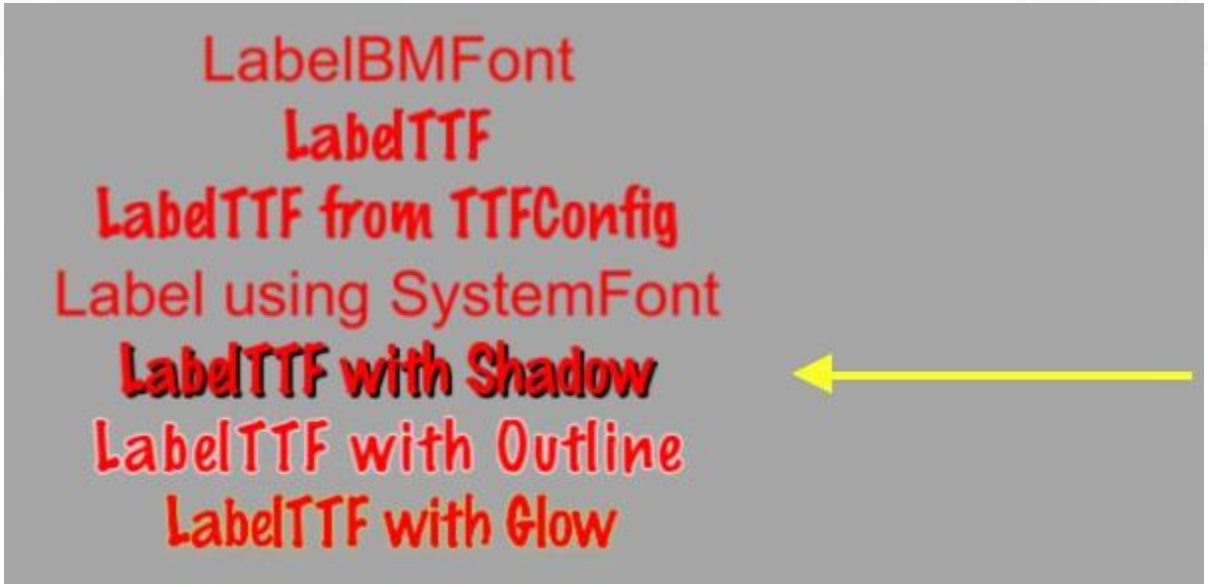
```
auto myLabel = Label::createWithSystemFont("My Label Text", "Arial", 16);
```



Label效果与排版

- Label特效： 阴影、轮廓、光晕等
- 创建一个带阴影的Label：

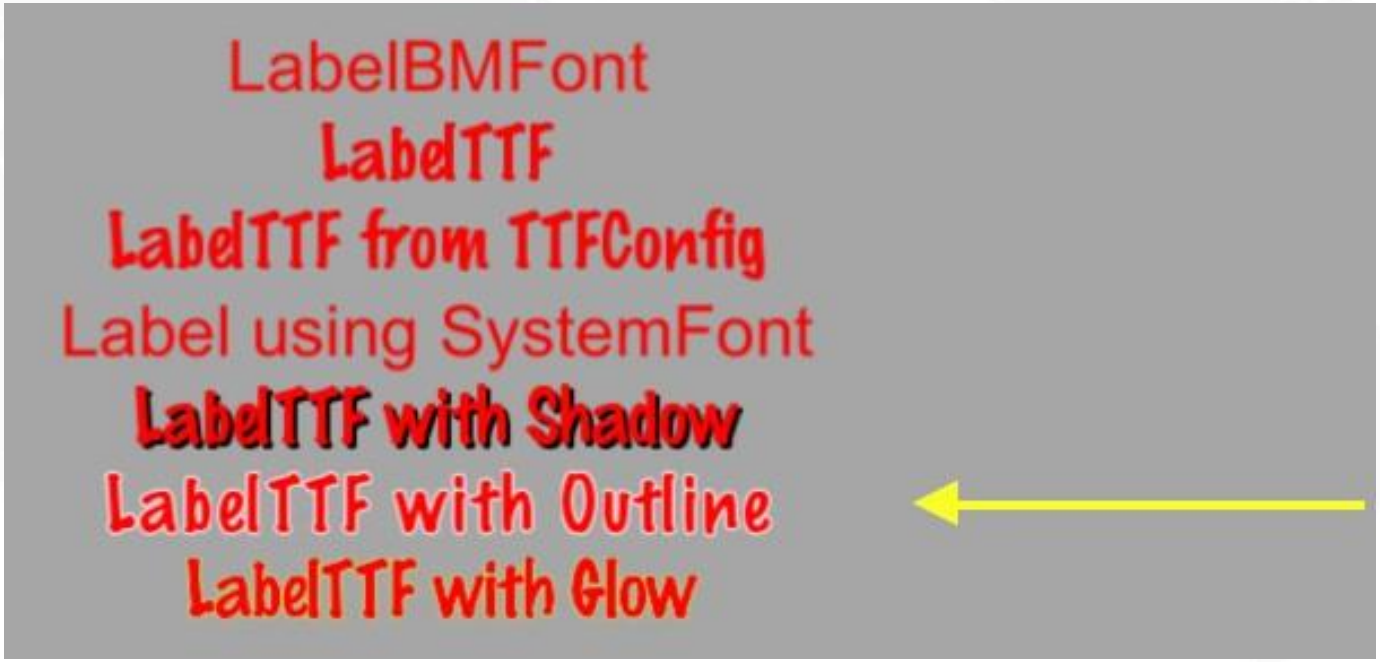
```
auto myLabel = Label::createWithTTF("myFont.ttf", "My Label Text", 16);

// shadow effect is supported by all Label types
myLabel->enableShadow();
```



Label效果与排版

- 创建一个带轮廓的Label：

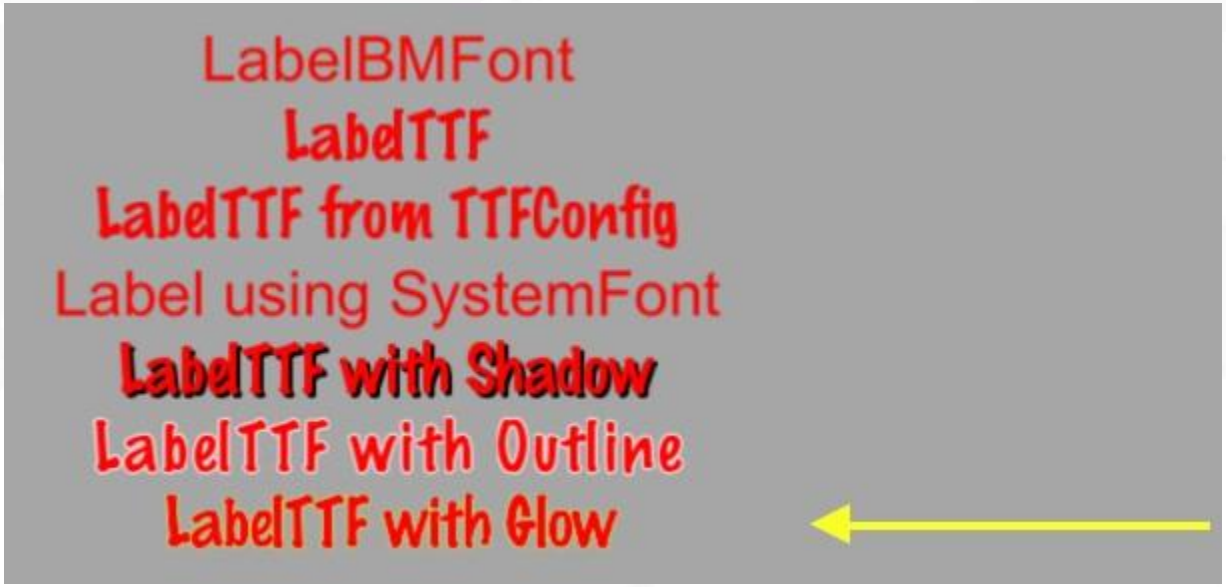
```
auto myLabel = Label::createWithTTF("myFont.ttf", "My Label Text", 16);

// outline effect is TTF only, specify the outline color desired
myLabel->enableOutline(Color4B::WHITE, 1));
```



Label效果与排版

- 创建一个带光晕的Label：

```
auto myLabel = Label::createWithTTF("myFont.ttf", "My Label Text", 16);

// glow effect is TTF only, specify the glow color desired.
myLabel->enableGlow(Color4B::YELLOW);
```



Menu菜单、MenuItem菜单项

Menu菜单由什么组成

- Menu菜单是游戏的导航
- 包含：播放、退出、设置、关于等
- 可点击的按钮形式

- 创建一个空Menu：

```
auto myMenu = Menu::create();
```

菜单选项和添加到菜单

- MenuItem：Menu的核心
- 三个要素：
- 正常状态、被选择状态、一个回调

- 注意：
  回调发生在Menuitem被选择的时候

菜单选项和添加到菜单

- 创建含单MenuItem的Menu

```
// creating a menu with a single item

// create a menu item by specifying images
auto closeItem = MenuItemImage::create("CloseNormal.png", "CloseSelected.png",
CC_CALLBACK_1(HelloWorld::menuCloseCallback, this));

auto menu = Menu::create(closeItem, NULL);
this->addChild(menu, 1);
```

- 第一个参数是正常状态的menuItem图片
- 第二个参数是选择状态的menuItem图片
- 第三个参数是点击Item的回调函数

菜单选项和添加到菜单

- Vector创建MenuItem对象：

```
// creating a Menu from a Vector of items
Vector<MenuItem*> MenuItems;

auto closeItem = MenuItemImage::create("CloseNormal.png", "CloseSelected.png",
CC_CALLBACK_1(HelloWorld::menuCloseCallback, this));

MenuItems.pushBack(closeItem);

/* repeat for as many menu items as needed */

auto menu = Menu::createWithArray(MenuItems);
this->addChild(menu, 1);
```

Lambda作为菜单回调

- Lambda函数——内联函数的函数

- 一个简单的Lambda函数：

```
// create a simple Hello World lambda
auto func = [] () { cout << "Hello World"; };

// now call it someplace in code
func();
```

Lambda作为菜单回调

- 使用lambda作为Action函数：

```
auto action1 = CallFunc::create([&](){
    std::cout << "using a Lambda callback" << std::endl;
});
```

- 使用lambda创建一个std::function：

```
std::function<void()> myFunction = []()
{
    std::cout << "From myFunction()" << std::endl;
};
auto action2 = CallFunc::create(myFunction);
```

Lambda作为菜单回调

- 使用lambda作为MenuItem回调：

```
auto closeItem = MenuItemImage::create("CloseNormal.png", "CloseSelected.png",
[&](Ref* sender){
    // your code here
});
```

GUI控件和容器

综述

- 新的GUI模块是基于GUI控件的框架，最开始设计是用于Cocos Studio中
- 父类继承自ProtectedNode的ui::Widget
- GUI两部分：Widget组件、Containers容器

Layout（布局）

- 所有容器的父类，继承自Widget
- 用于陈列子空间和剪裁
- 陈列元素：LayoutManager、LayoutParameter和Margin类
- HBox → 水平陈列
- VBox → 垂直陈列
- RelativedBox → 相对陈列
- ScrolView、ListView、PageView指定容器

Widgets组件

- GUI对象

- 包括：Button（按钮）、CheckBox（复选框）、LoadingBar（进度条）、Slider（滑动条）、ImageView（图像显示控件）、Text（文本）等

Button按钮

- 拦截触摸事件，点击按钮会调用一个预定义的回调函数
- 继承自ui:Wiget
- 设置按钮标题、图像、其他属性
- 三个状态：正常、被选择、无效（外观根据状态改变）

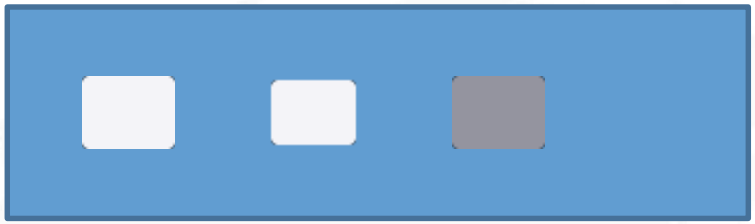
Button按钮

- 创建一个Button：

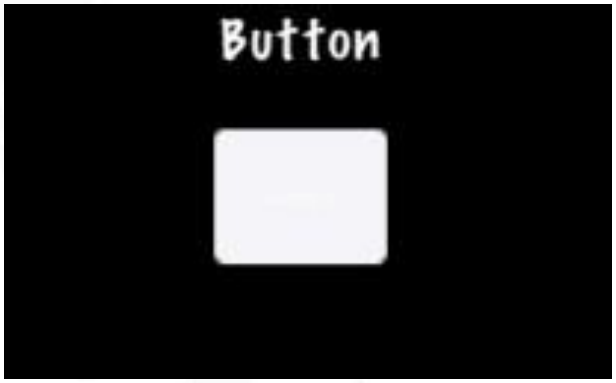
```
#include "ui/CocosGUI.h"

auto button = Button::create("normal_image.png", "selected_image.png", "disabled_image.png");

button->setTitleText("Button Text");

button->addTouchEventListener([&](Ref* sender, Widget::TouchEventType type){
        switch (type)
        {
                case ui::Widget::TouchEventType::BEGAN:
                        break;
                case ui::Widget::TouchEventType::ENDED:
                        std::cout << "Button 1 clicked" << std::endl;
                        break;
                default:
                        break;
        }
});

this->addChild(button);
```

Button按钮

- 需要资源：



- 显示呈现：



CheckBox复选框

- 多重选择
- 三个状态：正常、被选择、不可选
- 存储：0、1

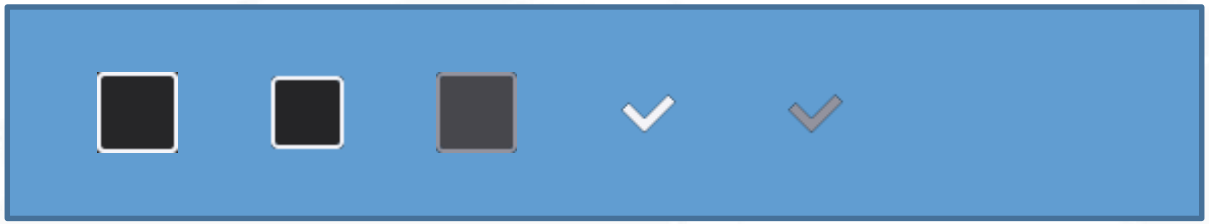
CheckBox复选框

- 创建一个CheckBox：

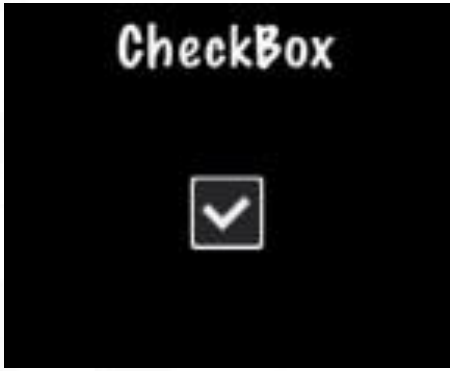
```
#include "ui/CocosGUI.h"

auto checkbox = CheckBox::create("check_box_normal.png",
                                 "check_box_normal_press.png",
                                 "check_box_active.png",
                                 "check_box_normal_disable.png",
                                 "check_box_active_disable.png");

checkbox->addTouchEventListener([&](Ref* sender, Widget::TouchEventType type){
        switch (type)
        {
                case ui::Widget::TouchEventType::BEGAN:
                        break;
                case ui::Widget::TouchEventType::ENDED:
                        std::cout << "checkbox 1 clicked" << std::endl;
                        break;
                default:
                        break;
        }
});

this->addChild(checkbox);
```

CheckBox复选框

- 需要资源：



- 显示呈现：



LoadingBar进度条

- 显示操作进程，状态条

- 创建一个LoadingBar：

```
#include "ui/CocosGUI.h"

auto loadingBar = LoadingBar::create("LoadingBarFile.png");

// set the direction of the loading bars progress
loadingBar->setDirection(LoadingBar::Direction::RIGHT);

this->addChild(loadingBar);
```

LoadingBar进度条

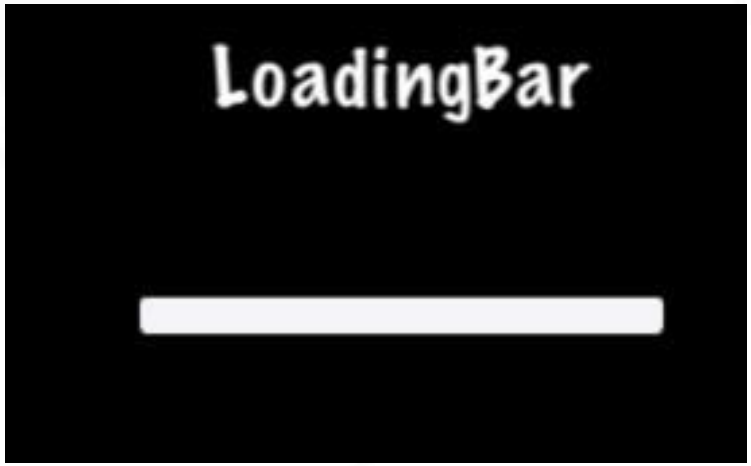
- 百分比控制LoadingBar显示：

```
#include "ui/CocosGUI.h"

auto loadingBar = LoadingBar::create("LoadingBarFile.png");
loadingBar->setDirection(LoadingBar::Direction::RIGHT);

// something happened, change the percentage of the loading bar
loadingBar->setPercent(25);

// more things happened, change the percentage again.
loadingBar->setPercent(35);

this->addChild(loadingBar);
```

LoadingBar进度条

- 需要资源：

- 显示呈现：



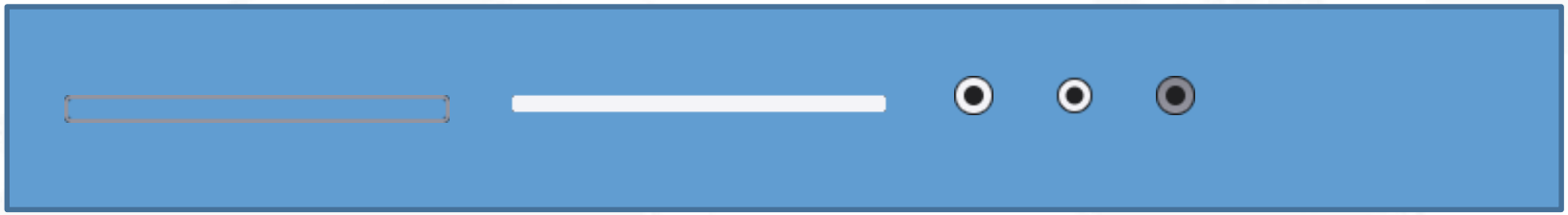
Slider滑动条

- 允许用户通过移动一个指标来设定值
- 创建一个Slider：

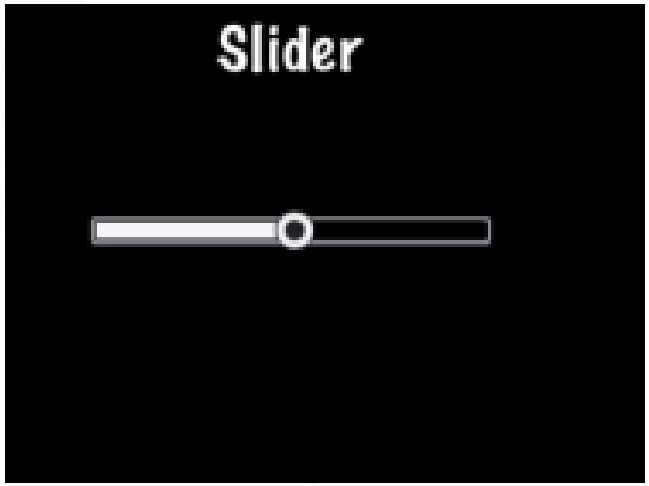
```
#include "ui/CocosGUI.h"

auto slider = Slider::create();
slider->loadBarTexture("Slider_Back.png"); // what the slider looks like
slider->loadSlidBallTextures("SliderNode_Normal.png", "SliderNode_Press.png", "SliderNode_Disable.png");
slider->loadProgressBarTexture("Slider_PressBar.png");

slider->addTouchEventListener([&](Ref* sender, Widget::TouchEventType type){
        switch (type)
        {
                case ui::Widget::TouchEventType::BEGAN:
                        break;
                case ui::Widget::TouchEventType::ENDED:
                        std::cout << "slider moved" << std::endl;
                        break;
                default:
                        break;
        }
});

this->addChild(slider);
```

Slider滑动条

- 需要资源：



- 显示呈现：



TextField文本框

- 用于输入文本
- 创建一个简单TextField：

```
#include "ui/CocosGUI.h"

auto textField = TextField::create("","Arial",30);

textField->addTouchEventListener([&](Ref* sender, Widget::TouchEventType type){
                std::cout << "editing a TextField" << std::endl;
});

this->addChild(textField);
```

TextField文本框

- 支持触摸事件、对焦、百分比定位和内容大小百分比
- 创建一个带参数TextField：

```
#include "ui/CocosGUI.h"

auto textField = TextField::create("","Arial",30);

// make this TextField password enabled
textField->setPasswordEnabled(true);

// set the maximum number of characters the user can enter for this TextField
textField->setMaxLength(10);

textField->addTouchEventListener([&](Ref* sender, Widget::TouchEventType type){
                std::cout << "editing a TextField" << std::endl;
});

this->addChild(textField);
```

TextField文本框

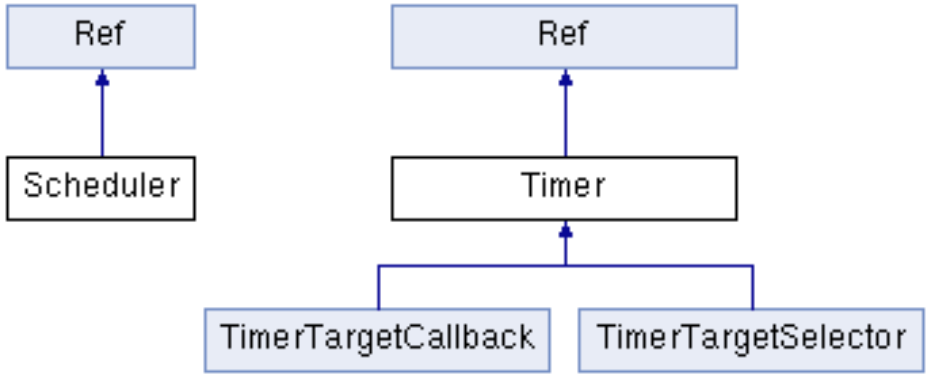
- 一般显示：

- 输入时：

调度器Scheduler

调度器Scheduler

- 继承关系



调度器Scheduler

- 原理介绍：

- 为游戏提供定时事件和定时调用服务
- 所有Node对象都知道如何调度和取消调度事件

- 使用调度器好处：
  - 每当Node不再可见或被移出，调度器停止
  - 引擎暂停，调度器停止；引擎重新开始，调度器继续启动
  - 引擎封装对应多平台的调度器，使用时不需关心其所设定的定时对象的销毁、停止、崩溃风险

调度器Scheduler

- 基本用法：

  处理对应随时间变化的逻辑判断

- 三种调度器：

- 默认调度器：schedulerUpdate()
- 自定义调度器：schedule(SEL_SCHEDULE selector, float interval, unsigned int repeat, float delay)
- 单次调度器：scheduleOnce(SEL_SCHEDULE selector, float delay)

调度器Scheduler

- 默认调度器：schedulerUpdate()
- 场合：
  使用Node刷新事件Update方法，每帧绘制前调用一次
- Node默认不启用Update事件，需要重载Update方法
- 取消方法：
  unschedulerUpdate()停止调度器

调度器Scheduler

- 默认调度器：schedulerUpdate()

- 示例：

HelloWorldScene.h

```
void  update( float  dt) override;
```

HelloWorldScene.cpp

```
bool  HelloWorld::init()
{
    ...
    scheduleUpdate();
    return  true ;
}

void  HelloWorld::update( float  dt)
{
    log ( "update" );
}
```

调度器Scheduler

- 默认调度器：schedulerUpdate()

- 示例输出：

```
cocos2d: update
cocos2d: update
cocos2d: update
cocos2d: update
```

调度器Scheduler

- 自定义调度器：scheduler

- 场合：
  不需要频繁的进行逻辑检测 → 提高游戏性能

- 条件：
  自定义时间间隔 > 2帧，一般时间在0.1秒+

- 取消方法：
  unschedule(SEL_SCHEDULE selector, float delay)

调度器Scheduler

- 自定义调度器：scheduler

- 示例：

HelloWorldScene.h

```
void  updateCustom( float  dt);
```

HelloWorldScene.cpp

```
bool HelloWorld::init()
{
    //...
        schedule(schedule_selector(HelloWorld::updateCustom), 1.0f, kRepeatForever, 0);
    return true;
}

void HelloWorld::updateCustom(float dt)
{
    log("Custom");
}
```

调度器Scheduler

- 自定义调度器：scheduler

- 示例结果：（每隔1秒输出）

```
cocos2d: Custom
cocos2d: Custom
cocos2d: Custom
cocos2d: Custom
cocos2d: Custom
```

调度器Scheduler

- 自定义调度器：scheduler
- scheduler(SEL_SCHEDULE selector, float interval, unsigned int repeat, float delay)

- 参数： 1、selector即为你要添加的事件函数 2、interval为事件触发时间间隔

3、repeat为触发一次事件后还会触发的次数，默认值为kRepeatForever，表示无限触发次数

4、delay表示第一次触发之前的延时

调度器Scheduler

- 单次调度器：schedulerOnce

- 场合：
  只想进行一次逻辑检测，只触发一次

- 取消方法：
  unschedule(SEL_SCHEDULE selector, float delay)

调度器Scheduler

- 单次调度器：schedulerOnce

- 示例：

HelloWorldScene.h

```
void  updateOnce( float  dt);
```

HelloWorldScene.cpp

```
bool  HelloWorld::init()
{
     ...
     scheduleOnce(schedule_selector(HelloWorld::updateOnce), 0.1f);
     return  true ;
}

void  HelloWorld::updateOnce( float  dt)
{
     log ( "Once" );
}
```

调度器Scheduler

- 单次调度器：schedulerOnce

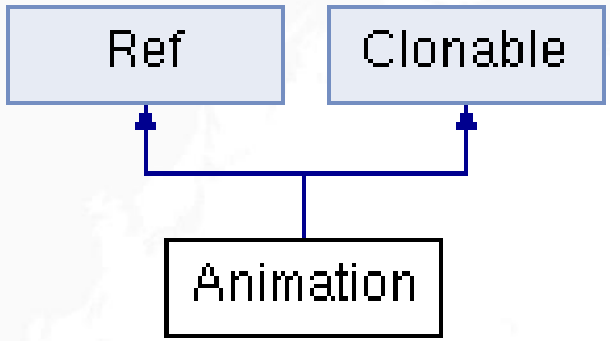
- 示例结果：

```
cocos2d: Once
```

序列帧动画

序列帧动画

- 继承关系



序列帧动画

- 原理：

- 使用Animation类描述一个动画
- 精灵显示动画的动作是一个Animate对象
- 动画动作Animate是精灵显示动画的动作，由动画对象创建，由精灵执行

序列帧动画

- 创建方法：

- 手动添加序列帧到Animation类
- 使用文件初始化Animation类

序列帧动画

- 手动添加：

- 每一帧精灵有序添加到Animation
- 设置每帧播放时间
- 通过setRestoreOriginFrame设置是否在动画播放结束后恢复到第一帧
- 创建完Animation实例后，需创建Animate实例来播放序列帧动画

序列帧动画

- 手动添加：

- 示例：

```
auto animation = Animation::create();
for (int i = 1; i<15; i++)
{
    char szName[100] = { 0 };
    sprintf(szName, "Images/grossini_dance_%d.png", i);
    animation->addSpriteFrameWithFile(szName);
}
// should last 2.8 seconds. And there are 14 frames.
animation->setDelayPerUnit(2.8f / 14.0f);
animation->setRestoreOriginalFrame(true);

auto action = Animate::create(animation);
_grossini->runAction(Sequence::create(action, action->reverse(), NULL));
```

序列帧动画

- 手动添加：

- 创建Animation实例时，会使用接口：

- addSpriteFrame，添加精灵帧到Animation实例
- setDelayUnits，设置每一帧持续时间，以秒为单位
- setRestoreOriginalFrame，设置是否在动画播放结束后恢复到第一帧
- clone，克隆一个该Animation实例

序列帧动画

- 文件添加：

- AnimationCache，加载xml/plist
  （plist文件里保存了组成动画的相关信息）

- 使用AnimationCache类时用到的接口：
- addAnimationsWithFile，添加动画文件到缓存，plist文件
- getAnimation，从缓存中获取动画对象
- getInstance，获取动画缓存实例对象

序列帧动画

- 文件添加：

- 示例：

```
auto cache = AnimationCache::getInstance();
cache->addAnimationsWithFile("animations/animations-2.plist");
auto animation2 = cache->getAnimation("dance_1");

auto action2 = Animate::create(animation2);
_tamara->runAction(Sequence::create(action2, action2->reverse(), NULL));
```

- 注意：
  3.0开始，Cocos2d-x使用getInstance来获取单例实例

序列帧动画

- 动画缓存：AnimationCache

- 每次创建需要加载，按序添加，创建动作 → 繁琐
- 使用频率比较高 → 缓存

- 接口：
- static AnimationCache* getInstance()，全局共享的单例
- void addAnimation(Animation *animation, const std::string& name)，添加一个动画到缓存
- void addAnimationsWithFile(const std::string& plist)，添加动画文件到缓存
- void removeAnimation(const std::string& name)，移除一个指定的动画
- Animation* getAnimation(const std::string& name)，获得事先存入的动画

序列帧动画

- 动画缓存：AnimationCache

- 建议：在内存警告时我们应该加入如下的清理缓存操作：

```
void releaseCaches()
{

    AnimationCache::destroyInstance();
    SpriteFrameCache::getInstance()->removeUnusedSpriteFrames();
    TextureCache::getInstance()->removeUnusedTextures();
}
```

- 注意清理顺序：
  动画缓存 → 精灵帧缓存 → 纹理缓存

谢谢！

Contact us:
商务邮箱：edu@chukong-inc.com
触控开发者平台：www.cocos.com