Bugly Cocos Plugin 使用指南

一、概述

Bugly Cocos Plugin 是为基于 **Cocos** 引擎的**App**（**iOS/Android**）封装的 **Bugly** 原生(**iOS/Android**) **SDK** 的接口，方便调用 **Bugly** 原生 **SDK**，可用于SDK初始化、设置自定义用户信息、错误等，并自动捕获上报**App**的脚本（**Lua**、**JavaScript**）错误和原生代码（如**Objective-C**、**Java**、**C/C++**等）引发的崩溃信息，提供实时、多维度的异常问题分析服务。

> **准备工作** 如果你尚未注册**App**，请登录 Bugly 网站，注册新App以获取 **Bugly AppID** 。 **注意：** 导出的**Android**和**iOS**项目分别需要注册**两个不同的AppID**。

二、集成步骤

1. 下载 Bugly Cocos Plugin

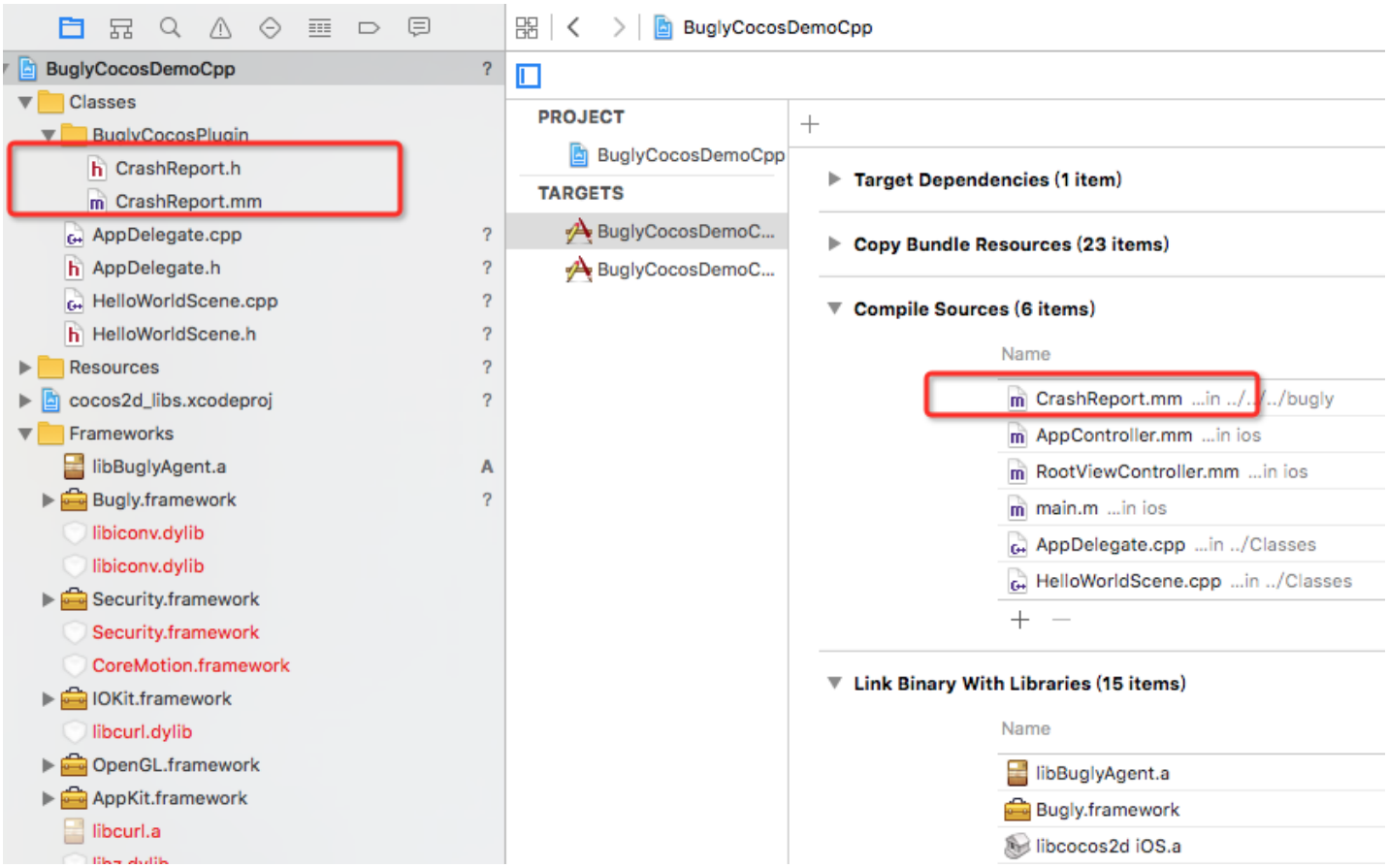
下载 Bugly Cocos Plugin，根据 **Cocos** 项目的开发语言(**c++**、**lua**、**js**)集成插件

> 下载包目录结构说明：
>
> - CocosPlugin
>   - **bugly** - 支持**c++**的接口封装
>   - **bugly/lua** - 支持**lua**的接口封装
>   - **bugly/js** - 支持**js**的接口封装
>   - **agent/** - 原生SDK(iOS/Android)的接口封装
> - BuglySDK - 原生SDK(iOS/Android)

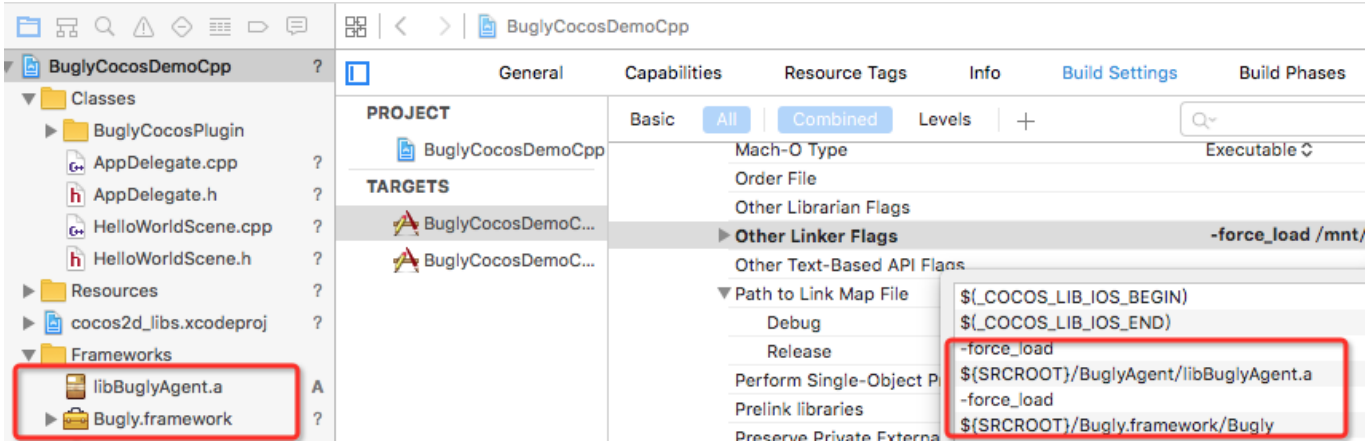
2. 集成 Bugly Cocos Plugin

iOS 工程配置

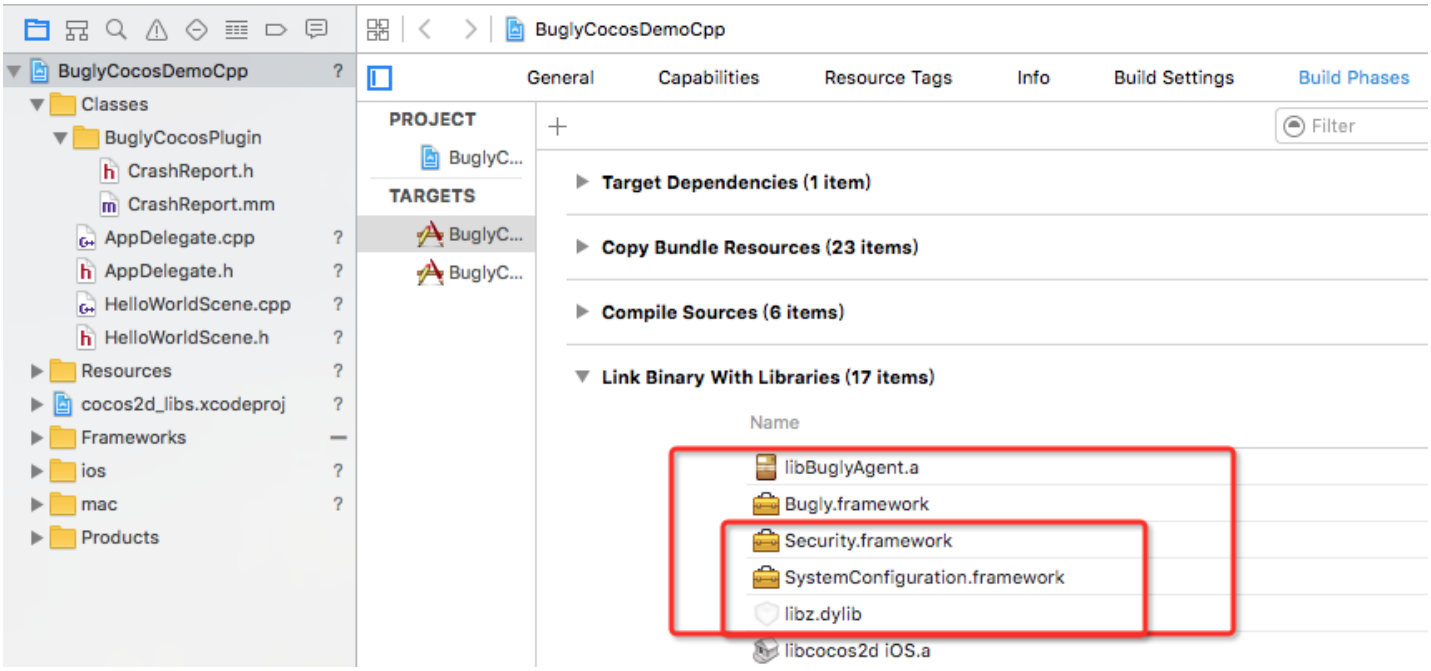
- 打开 `proj.ios_mac` 目录的 **Xcode** 工程，将 `bugly` 目录的头文件和源文件添加到工程中，并将源文件添加到指定 **Target** 的 `Compile Sources` 中，如图



- 将 `agent/iOS` 目录的 **libBuglyAgent.a** 和 `BuglySDK/iOS` 目录下的 **Bugly.framework** 添加到 **Xcode** 工程中，并在 `Build Settings` 的 `Other Linker Flags` 配置中添加 `-force_load` 标记，设置 **libBuglyAgent.a** 和 **Bugly.framework** 的路径，如图



- 检查 **Xcode** 工程的 `Build Phases` 的 `Link Binary With Libraries` 配置，确认已经添加依赖的动态库，如图

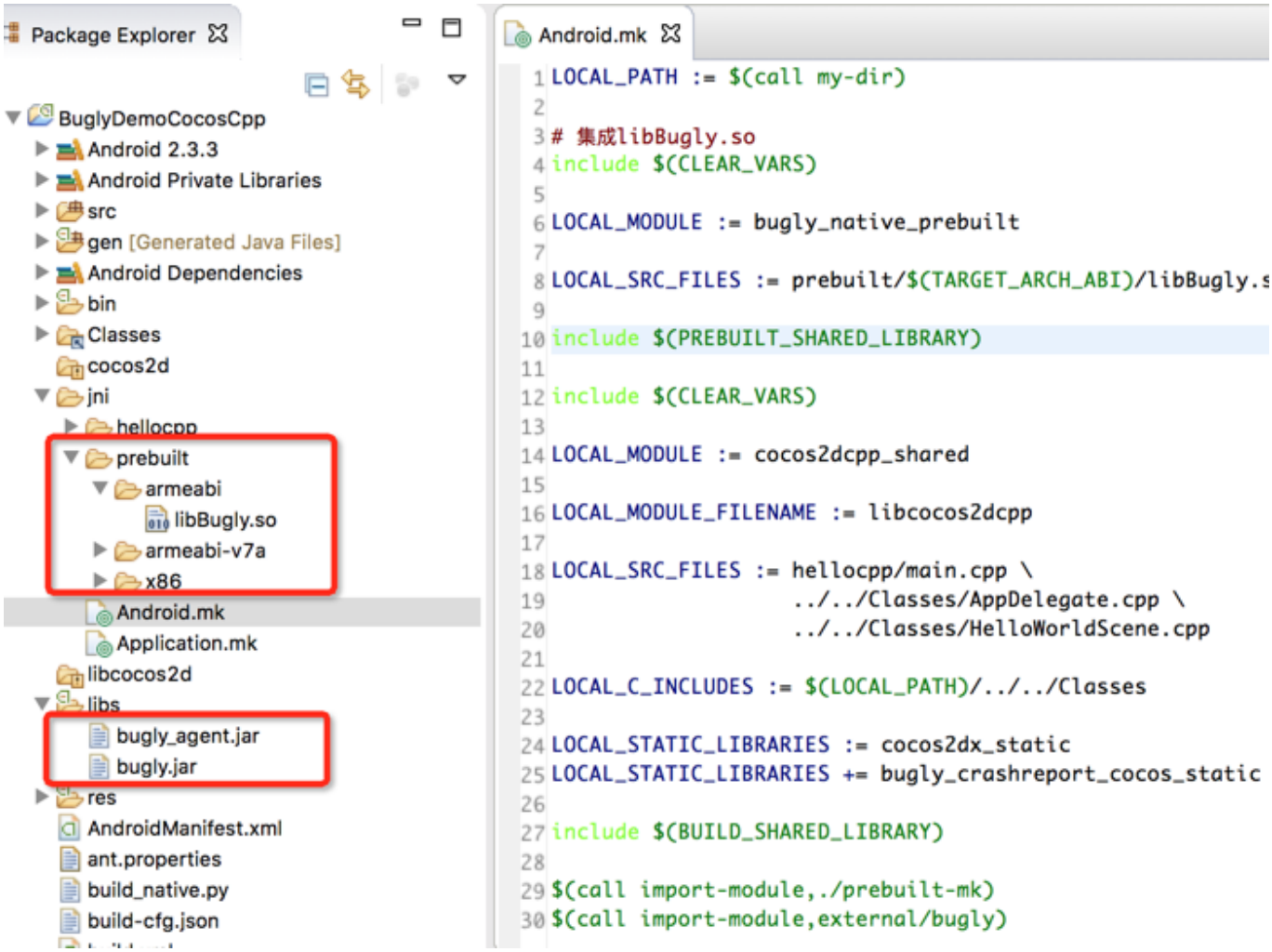


Android 工程配置

- 将 `bugly` 目录拷贝到项目编译使用的 **Cocos** 引擎(源码或框架)目录的 `external` 子目录，即

```
{cocos2d-x}/external/bugly
```

- 打开 `proj.android` (或 `proj.android_studio`) 目录的 **Android** 工程，将 `agent/Android` 目录下 **.jar** 和 `BuglySDK/Android` 目录下的 **.jar** 添加到工程的 `libs` 目录；将 `BuglySDK/Android` 目录下的 **libBugly.so** 添加到工程的 `jni/prebuilt` 目录，如图



- 修改 `proj.android/jni/Android.mk` 文件，添加如下配置：

```
# 集成libBugly.so, 添加在LOCAL_PATH := $(call my-dir)行之后
include $(CLEAR_VARS)

LOCAL_MODULE := bugly_native_prebuilt

LOCAL_SRC_FILES := prebuilt/$(TARGET_ARCH_ABI)/libBugly.so

include $(PREBUILT_SHARED_LIBRARY)
# ————————

# 引用bugly/Android.mk定义的Module, 在LOCAL_STATIC_LIBRARIES := xxx 行之后添加
LOCAL_STATIC_LIBRARIES += bugly_crashreport_cocos_static

# 添加在末行
$(call import-module,external/bugly)
```

示例：

```
LOCAL_PATH := $(call my-dir)

# --- libBugly.so ---
include $(CLEAR_VARS)

LOCAL_MODULE := bugly_native_prebuilt
# 可在Application.mk添加APP_ABI := armeabi armeabi-v7a 指定集成对应架构的.so文件
LOCAL_SRC_FILES := prebuilt/$(TARGET_ARCH_ABI)/libBugly.so

include $(PREBUILT_SHARED_LIBRARY)
# --- end ---

include $(CLEAR_VARS)

LOCAL_MODULE := cocos2dcpp_shared
LOCAL_MODULE_FILENAME := libcocos2dcpp

LOCAL_SRC_FILES := hellocpp/main.cpp \
                   ../../Classes/AppDelegate.cpp \
                   ../../Classes/HelloWorldScene.cpp

LOCAL_C_INCLUDES := $(LOCAL_PATH)/../../Classes

LOCAL_STATIC_LIBRARIES := cocos2dx_static
LOCAL_STATIC_LIBRARIES += bugly_crashreport_cocos_static

include $(BUILD_SHARED_LIBRARY)

$(call import-module,./prebuilt-mk)
$(call import-module,external/bugly)
```

- 编辑AndroidManifest.xml文件，添加如下权限声明:

```
<uses-permission android:name="android.permission.READ_PHONE_STATE" />
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE" />
<uses-permission android:name="android.permission.READ_LOGS" />
```

初始化

- 在 `Classes/AppDeleagate.cpp` 文件的 `AppDelegate::applicationDidFinishLaunching()` 方法中调用接口 `CrashReport::initCrashReport(const char* appId, bool debug)` 进行初始化

示例代码：

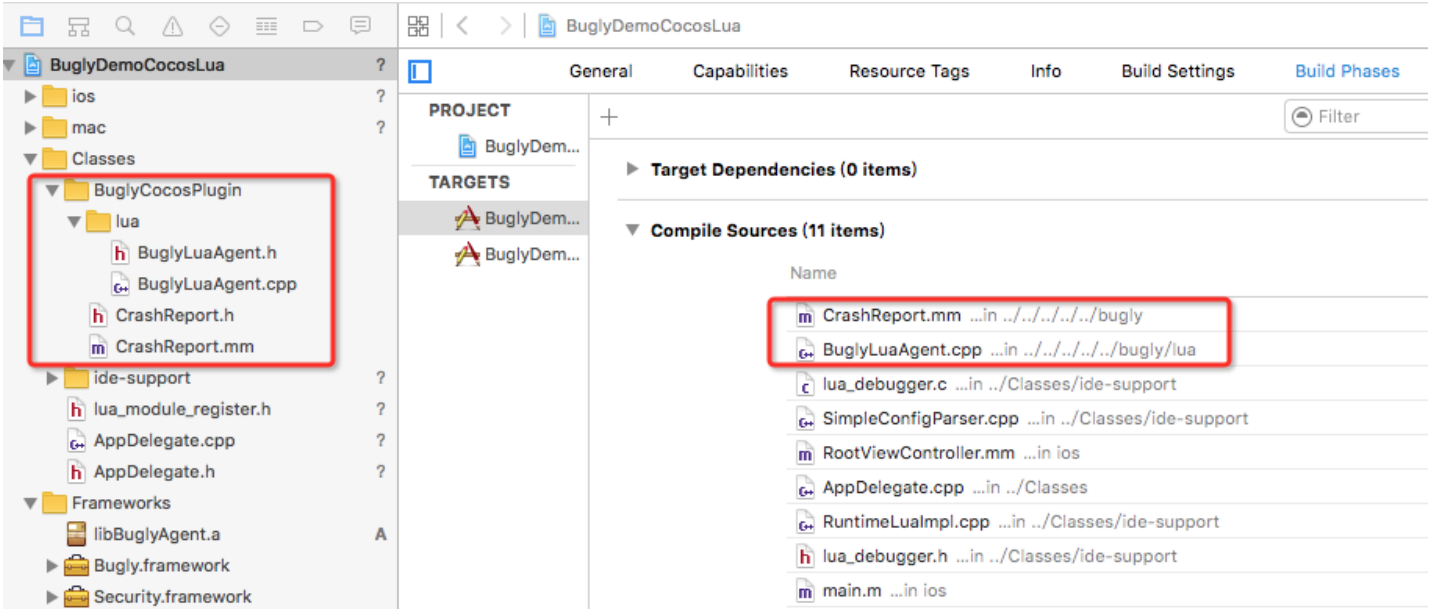
```
// 导入头文件
#if (CC_TARGET_PLATFORM == CC_PLATFORM_ANDROID)
#include "bugly/CrashReport.h"
#elif (CC_TARGET_PLATFORM == CC_PLATFORM_IOS)
#include "CrashReport.h"
#endif
```

```
bool AppDelegate::applicationDidFinishLaunching() {
    // Init the Bugly
    CrashReport::initCrashReport("Your AppID", false);

    // initialize director
    auto director = Director::getInstance();
    // ...

    return true;
}
```

3. 支持Lua脚本错误捕获 `仅限Cocos lua工程可用`

iOS 工程配置

- 打开 `proj.ios_mac` 目录的 **Xcode** 工程，将 `bugly/lua` 目录下头文件和源文件添加到工程中，并将源文件添加到 **Target** 的 `Compile Sources` 中，如图



Android 工程配置

- 修改 `proj.android/jni/Android.mk` 文件，添加如下配置：

```
# 引入bugly/lua/Android.mk定义的Module
LOCAL_STATIC_LIBRARIES += bugly_agent_cocos_static_lua

# 添加在最后一行
$(call import-module,external/bugly/lua)
```

示例:

```
LOCAL_PATH := $(call my-dir)

# --- 引用 libBugly.so ---
include $(CLEAR_VARS)

LOCAL_MODULE := bugly_native_prebuilt
# 可在Application.mk添加APP_ABI := armeabi armeabi-v7a 指定集成对应架构的.so文件
LOCAL_SRC_FILES := prebuilt/$(TARGET_ARCH_ABI)/libBugly.so

include $(PREBUILT_SHARED_LIBRARY)
# --- end ---

include $(CLEAR_VARS)

LOCAL_MODULE := cocos2dlua_shared

LOCAL_MODULE_FILENAME := libcocos2dlua

FILE_LIST := hellolua/main.cpp
FILE_LIST += $(wildcard $(LOCAL_PATH)/../../Classes/*.cpp)
FILE_LIST += $(wildcard $(LOCAL_PATH)/../../Classes/ide-support/*.cpp)
FILE_LIST += $(wildcard $(LOCAL_PATH)/../../Classes/ide-support/*.c)

LOCAL_SRC_FILES := $(FILE_LIST:$(LOCAL_PATH)/%=%)

LOCAL_C_INCLUDES := $(LOCAL_PATH)/../../Classes

LOCAL_STATIC_LIBRARIES := cocos2d_lua_static
LOCAL_STATIC_LIBRARIES += cocos2d_simulator_static
# 引入bugly/Android.mk定义的Module
LOCAL_STATIC_LIBRARIES += bugly_crashreport_cocos_static
# 引入bugly/lua/Android.mk定义的Module
LOCAL_STATIC_LIBRARIES += bugly_agent_cocos_static_lua

include $(BUILD_SHARED_LIBRARY)

$(call import-module,scripting/lua-bindings/proj.android/prebuilt-mk)
$(call import-module,tools/simulator/libsimulator/proj.android/prebuilt-mk)

$(call import-module,external/bugly)
$(call import-module,external/bugly/lua)
```

注册Lua脚本错误的监听回调函数

- 在 `Classes/AppDeleagate.cpp` 文件的 `AppDelegate::applicationDidFinishLaunching()` 方法中调用接口

`BuglyLuaAgent::registerLuaExceptionHandler(...)` 进行初始化

示例代码：

```
bool AppDelegate::applicationDidFinishLaunching()
{
    //
    CrashReport::initCrashReport("Your AppID", false);

    // set default FPS
    Director::getInstance()->setAnimationInterval(1.0 / 60.0f);
    // register lua module
    auto engine = LuaEngine::getInstance();
    ScriptEngineManager::getInstance()->setScriptEngine(engine);

    // register lua exception handler
    BuglyLuaAgent::registerLuaExceptionHandler(engine);

#if (COCOS2D_DEBUG > 0) && (CC_CODE_IDE_DEBUG_SUPPORT > 0)
    // NOTE:Please don't remove this call if you want to debug with Cocos Code ID
E
    auto runtimeEngine = RuntimeEngine::getInstance();
    runtimeEngine->addRuntime(RuntimeLuaImpl::create(), kRuntimeEngineLua);
    runtimeEngine->start();
#else
    if (engine->executeScriptFile("src/main.lua"))
    {
        return false;
    }
#endif

    return true;
}
```

- 在 **Lua** 脚本的__G__***TRACKBACK***__回调函数中，调用 **Lua** 脚本错误上报的接口 `buglyReportLuaException`

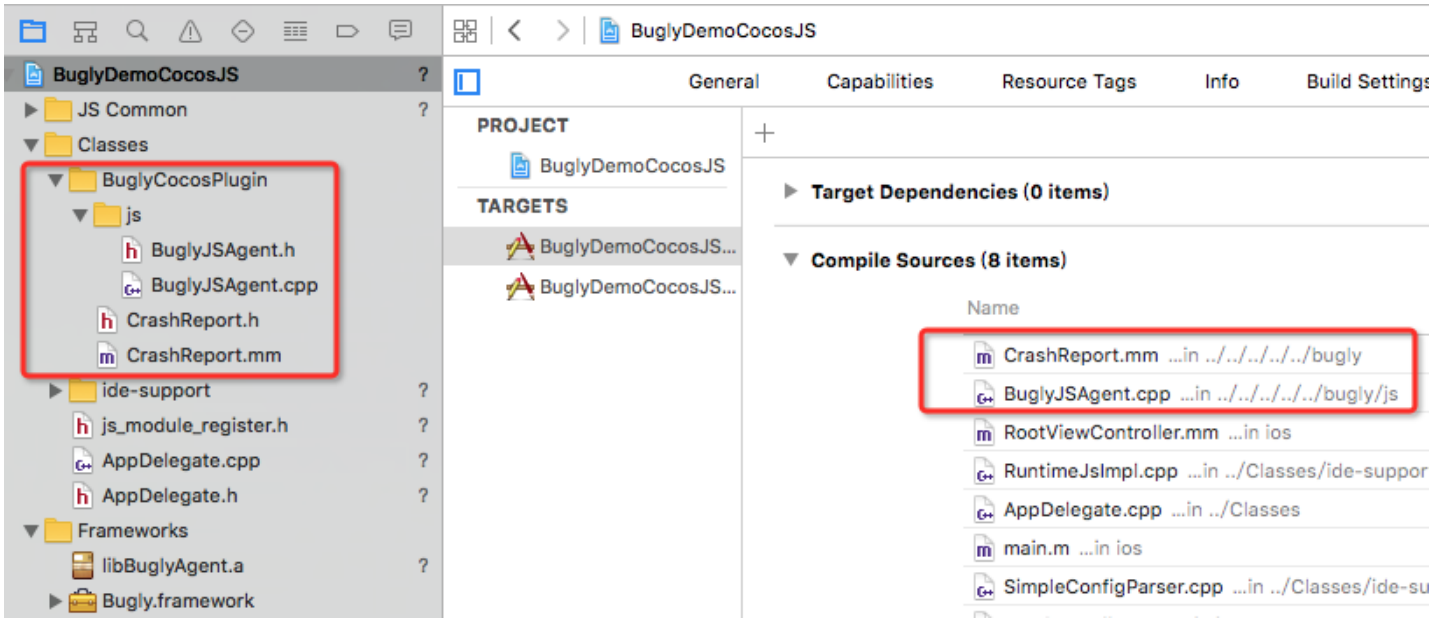
示例代码：

```
__G__TRACKBACK__ = function(msg)
     -- record the message
    local message = msg;

     -- auto genretated
    local msg = debug.traceback(msg, 3)
    print(msg)

    -- report lua exception
    buglyReportLuaException(tostring(message), debug.traceback())

    return msg
end
```

4. 支持JavaScript脚本错误捕获 `仅限Cocos js工程可用`

iOS 工程配置

- 打开 `proj.ios_mac` 目录的 **Xcode** 工程，将 `bugly/js` 目录下头文件和源文件添加到工程中，并将源文件添加到指定 **Target** 的 `Compile Sources` 中，如图



Android 工程配置

- 修改 `proj.android/jni/Android.mk` 文件，添加如下配置：

```
# 引入bugly/lua/Android.mk定义的Module
LOCAL_STATIC_LIBRARIES += bugly_agent_cocos_static_js

# 添加在最后一行
$(call import-module,external/bugly/js)
```

示例：

```
LOCAL_PATH := $(call my-dir)

# --- 引用 libBugly.so ---
include $(CLEAR_VARS)

LOCAL_MODULE := bugly_native_prebuilt
# 可在Application.mk添加APP_ABI := armeabi armeabi-v7a 指定集成对应架构的.so文件
LOCAL_SRC_FILES := prebuilt/$(TARGET_ARCH_ABI)/libBugly.so

include $(PREBUILT_SHARED_LIBRARY)
# --- end ---

include $(CLEAR_VARS)

LOCAL_MODULE := cocos2djs_shared

LOCAL_MODULE_FILENAME := libcocos2djs

LOCAL_SRC_FILES := \
../../Classes/AppDelegate.cpp \
../../Classes/ide-support/SimpleConfigParser.cpp \
../../Classes/ide-support/RuntimeJsImpl.cpp \
hellojavascript/main.cpp

LOCAL_C_INCLUDES := $(LOCAL_PATH)/../../Classes

LOCAL_STATIC_LIBRARIES := cocos2d_js_static
LOCAL_STATIC_LIBRARIES += cocos2d_simulator_static

# 引入bugly/Android.mk定义的Module
LOCAL_STATIC_LIBRARIES += bugly_crashreport_cocos_static
# 引入bugly/js/Android.mk定义的Module
LOCAL_STATIC_LIBRARIES += bugly_agent_cocos_static_js

include $(BUILD_SHARED_LIBRARY)

$(call import-module,scripting/js-bindings/proj.android/prebuilt-mk)
$(call import-module,tools/simulator/libsimulator/proj.android/prebuilt-mk)

$(call import-module,external/bugly)
$(call import-module,external/bugly/js)
```

注册JavaScript脚本错误的监听回调函数

- 在 `Classes/AppDeleagate.cpp` 文件的

`AppDelegate::applicationDidFinishLaunching()` 方法中调用接口 `BuglyJSAgent::registerJSExceptionHandler(JSContext * cx)` 进行初始化，如果需要在JavaScript脚本中调用接口方法，则需要调用接口 `ScriptingCore::getInstance()->addRegisterCallback` 注册回调 `BuglyJSAgent::registerJSFunctions`

示例代码：

```
bool AppDelegate::applicationDidFinishLaunching()
{
    //
    CrashReport::initCrashReport("Your AppID", false);

    // initialize director
    auto director = Director::getInstance();

    js_module_register();

    // js function register before start()
    ScriptingCore::getInstance()->addRegisterCallback(BuglyJSAgent::registerJSFunc
tions);

    ScriptingCore* sc = ScriptingCore::getInstance();
    sc->start();
    sc->runScript("script/jsb_boot.js");

    // js exception handler before runScript
 BuglyJSAgent::registerJSExceptionHandler(ScriptingCore::getInstance()->getGlobal
Context());

    ScriptingCore::getInstance()->runScript("main.js");
#endif

    return true;
}
```

三、接口说明

C++ 接口

- **void initCrashReport(const char* appId, bool debug)**

> 初始化SDK，并设置是否开启SDK的调试模式(调试模式会打印较多的日志信息，请在发布版本务必设置为NO)

- **void initCrashReport(const char* appId, bool debug, CrashReport::CRLogLevel level)**

  > 初始化SDK，设置Debug模式，并设置自定义日志上报的日志级别(只控制iOS SDK的上报)

- **void setUserId(const char* userId)**

  > 设置用户唯一标识

- **void setAppChannel(const char* channel)**

  > 设置App的渠道，初始化方法之前调用设置有效

- **void setAppVersion(const char* version)**

  > 设置App的版本，初始化方法之前调用设置有效

- **void reportException(int category, const char* type, const char* msg, const char* traceback)**

  > 上报自定义异常

- **void setTag(int tag)**

  > 设置自定义标签

- **void addUserValue(const char* key, const char* value)**

  > 设置用户自定义数据

- **void removeUserValue(const char* key)**

  > 删除用户自定义数据

- **void log(CrashReport::CRLogLevel level, const char* tag, const char * fmt, ...)**

  > 自定义日志打印接口

Lua 接口

- **buglySetUserId(string id)**

  > 设置用户id，您在页面的异常详情会显示具体用户的id

- **buglySetTag(int tag)**

  > 设置当前场景的“TAG”，int类型（bugly页面可备注该值含义）。
  >
  > 一次运行过程中只有一个TAG，以上报异常时最后TAG为准。例如登录时设置tag=1，进入副本设置tag=2

- **buglyAddUserValue(string key, string value)**

  > 上报一些自定义的Key-Value键值对：
  >
  > 1）最多10对，超出的会被忽略
  >
  > 2）每个key限长50字节，value限长200字节
  >
  > 3）key限制为字母 + 数字

- **buglyLog(int level, string tag, string log)**

  > 记录开发者自定义的关键信息日志。该日志会随异常信息一起上报。
  >
  > 注：Log最大长度约10K，超长会保留最近内容。建议每条Log长度控制在200字节以内。

JavaScript 接口

- **buglySetUserId(string id)**

  > 设置用户id，您在页面的异常详情会显示具体用户的id

- **buglySetTag(int tag)**

  > 设置当前场景的“TAG”，int类型（bugly页面可备注该值含义）。
  >
  > 一次运行过程中只有一个TAG，以上报异常时最后TAG为准。例如登录时设置tag=1，进入副本设置tag=2

- **buglyAddUserValue(string key, string value)**

  > 上报一些自定义的Key-Value键值对：
  >
  > 1）最多10对，超出的会被忽略
  >
  > 2）每个key限长50字节，value限长200字节
  >
  > 3）key限制为字母 + 数字

- **buglyLog(int level, string tag, string log)**

  > 记录开发者自定义的关键信息日志。该日志会随异常信息一起上报。
  >
  > 注：Log最大长度约10K，超长会保留最近内容。建议每条Log长度控制在200字节以内。