# HTML5+
# Native.js 入门指南

**（v0.6）**

DCloud.io

更新时间：2014 年 5 月 15 日

# 一、概述

**Native.js** 技术，简称 **NJS**，是一种将手机操作系统的原生对象转义，映射为 JS 对象，在 JS 里编写原生代码的技术。

如果说 Node.js 把 js 扩展到服务器世界，那么 Native.js 则把 js 扩展到手机 App 的原生世界。

HTML/JS/Css 全部语法只有 7 万多，而原生语法有几十万，Native.js 大幅提升了 HTML5 的能力。

NJS 突破了浏览器的功能限制，也不再需要像 Hybrid 那样由原生语言开发插件才能补足浏览器欠缺的功能。

NJS 编写的代码，最终需要在 HBuilder 里打包发行为 App 安装包，或者在支持 Native.js 技术的浏览器里运行。目前 Native.js 技术不能在普通手机浏览器里直接运行。

- NJS 大幅扩展了 HTML5 的能力范围，原本只有原生或 Hybrid App 的原生插件才能实现的功能如今可以使用纯 JS 实现。
- NJS 大幅提升了 App 开发效率，将 iOS、Android、Web 的 3 个工程师组队才能完成的 App，变为 1 个 web 工程师就搞定。
- NJS 不再需要配置原生开发和编译环境，调试、打包均在 HBuilder 里进行。没有 mac 和 xcode 一样可以开发 iOS 应用。
- NJS 会大幅提升 web 工程师在项目中的主导地位。

## 技术要求

由于 NJS 是直接调用 Native API，需要对 Native API 有一定了解，知道所需要的功能调用了哪些原生 API，能看懂原生代码并参考原生代码修改为 JS 代码。

# 二、开始使用

## 1. 判断平台

Native API 具有平台依赖性，所以需要通过以下方式判断当前的运行平台：

```
function judgePlatform(){
    switch ( plus.os.name ) {
        case "Android":
        // Android 平台: plus.android.*
        break;
        case "iOS":
        // iOS 平台: plus.ios.*
        break;
        default:
        // 其它平台
        break;
    }
}
```

## 2. 类型转换

在 NJS 中调用 Native API 或从 Native API 返回数据到 NJS 时会自动转换数据类型，对应表如下：

| 类型 | Objective-C | Java | JavaScript |
|---|---|---|---|
| 基本数据 | byte/short/int/long/float/double/... | byte/short/int/long/float/double/... | Number |
| 字符 | char | char | String |
| 字符串 | NSString/@"" | String/"" | String |
| 数组 | @[1,2,3]/NSArray | new XXX[] | InstanceObject |
| 类 | @interface | class | ClassObject |
| 对象（实例） | * | * | InstanceObject |
| 空对象 | nil | null | null |
| 其它 | Protocol | Interface | Object(JSON) |

## 3. 其他转换

- Android 原生应用的主 Activity 对象 转为 plus.android.runtimeMainActivity()
  Android 的主 Activity 对象是启动应用时自动创建的，不是代码创建，此时通过 plus.android.runtimeMainActivity()方法获取该 Activity 对象
- Objective-C 方法冒号剔除

[pos setPositionX:(int)x Y:(int)y;] 转为 pos.setPositionXY(x,y);

OC 语法中方法的定义格式为:

“(返回值类型) 函数名: (参数 1 类型) 形参 1 参数 2 名称: (参数 2 类型) 形参 2”

方法的完整名称为: “函数名:参数 2 名称:”。

如:“（void）setPositionX:(int)x Y:(int)y;”，方法的完整名称为“setPositionX:Y:”，调用时语法为：“[pos setPositionX:x Y:y];”。

在 JS 语法中函数名称不能包含“:”字符，所以 OC 对象的方法名映射成 NJS 对象方法名时将其中的“:”字符自动删除，上面方法名映射为“setPositionXY”，在 NJS 调用的语法为：“pos.setPositionXY(x,y);”。

- 文件路径转换

  Web 开发里使用的 image/1.png 是该 web 工程的相对路径，而原生 API 中经常需要使用绝对路径，比如/sdcard/apptest/image/1.png，此时使用这个扩展方法来完成转换：plus.io.convertLocalFileSystemURL("image/1.png")

# 4. 概念

## 类对象

由于 JavaScript 中本身没有类的概念，为了使用 Native API 层的类，在 NJS 中引入了类对象（ClassObject）的概念，用于对 Native 中的类进行操作，如创建类的实例对象、访问类的静态属性、调用类的静态方法等。其原型如下：

```
Interface ClassObject {
    function Object plusGetAttribute( String name );
    function void plusSetAttribute( String name, Object value );
}
```

- 获取类对象

在 iOS 平台我们可以通过 plus.ios.importClass(name)方法导入类对象，参数 name 为类的名称；在 Android 平台我们可以通过 plus.android.importClass(name)方法导入类对象，其参数 name 为类的名称，必须包含完整的命名空间。

**示例：**

```
// iOS 平台导入 NSNotificationCenter 类
var NSNotificationCenter = plus.ios.importClass("NSNotificationCenter");

// Android 平台导入 Intent 类
var Intent = plus.android.importClass("android.content.Intent");
```

获取类对象后，可以通过类对象“.”操作符获取类的静态常量属性、调用类的静态方法，类的静态非常量属性需通过 plusGetAttribute、plusSetAttribute 方法操作。

# 实例对象

在 JavaScript 中，所有对象都是 Object，为了操作 Native 层类的实例对象，在 NJS 中引入了实例对象（InstanceObject）的概念，用于对 Native 中的对象进行操作，如操作对象的属性、调用对象的方法等。其原型如下：

```
Interface InstanceObject {
    function Object plusGetAttribute( String name );
    function void plusSetAttribute( String name, Object value );
}
```

- 获取实例对象

有两种方式获取类的实例对象，一种是调用 Native API 返回值获取，另一种是通过 new 操作符来创建导入的类对象的实例，如下：

```
// iOS 平台导入 NSDictionary 类
var NSDictionary = plus.ios.importClass("NSDictionary");
//  创建 NSDictionary 的实例对象
var ns = new NSDictionary();

// Android 平台导入 Intent 类
var Intent = plus.android.importClass("android.content.Intent");
//  创建 Intent 的实例对象
var intent = new Intent();
```

获取实例对象后，可以通过实例对象“.”操作符获取对象的常量属性、调用对象的成员方法，实例对象的非常量属性则需通过 plusGetAttribute、plusSetAttribute 方法操作。

# 操作对象的属性方法

- 常量属性

获取对象后就可以通过“.”操作符获取对象的常量属性，如果是类对象则获取的是类的静态常量属性，如果是实例对象则获取的是对象的成员常量属性。

- 非常量属性

如果 Native 层对象的属性值在原生环境下被更改，此时使用“.”操作符获取到对应 NJS 对象的属性值就可能不是实时的属性值，而是该 Native 层对象被映射为 NJS 对象那一刻的属性值。

为获取获取 Native 层对象的实时属性值，需调用 NJS 对象的 plusGetAttribute(name)方法，参数 name 为属性的名称，返回值为属性的值。调用 NJS 对象的 plusSetAttribute(name,value)方法设置 Native 层对象的非常量属性值，参数 name 为属性的名称，value 为要设置新的属性值。

**注意：**使用 plusGetAttribute(name)方法也可以获取 Native 层对象的常量属性值，但不如直接使用“.”操作符来获取性能高。

- 方法

获取对象后可以通过“.”操作符直接调用 Native 层方法，如果是类对象调用的是 Native 层类的静态方法，如果是实例对象调用的是 Native 层对象的成员方法。

**注意**：在 iOS 平台由于 JS 语法的原因，Objective-C 方法名称中的“:”字符转成 NJS 对象的方法名称后将会被忽略，因此在 NJS 中调用的方法名需去掉所有“：”字符。

- 类的继承

Objective-C 和 Java 中类如果存在继承自基类，在 NJS 中对应的对象会根据继承关系递归将所有基类的公有方法一一换成 NJS 对象的方法，所有基类的公有属性也可以通过其 plusGetAttribute、plusSetAttribute 方法访问。

# 5. 开始写 NJS

使用 NJS 调用 Native API 非常简单，基本步骤如下：

1. 导入要使用到的类；
2. 创建类的实例对象（或者调用类的静态方法创建）；
3. 调用实例对象的方法；

以下例子使用 NJS 调用 iOS 和 Android 的原生弹出提示框（类似但不同于 js 的 alert）。

## Android

以下代码在 Android 平台展示调用 Native API 显示系统提示框。

首先是 Android 原生 Java 代码，用于比对参考：

```
import android.app.AlertDialog;
//...
// 创建提示框构造对象，Builder 是 AlertDialog 的内部类。参数 this 指代 Android 的主 Activity 对象，该对象启动应用时自动生成
AlertDialog.Builder dlg = new AlertDialog.Builder(this);
// 设置提示框标题
dlg.setTitle("自定义标题");
// 设置提示框内容
dlg.setMessage("使用 NJS 的原生弹出框，可自定义弹出框的标题、按钮");
// 设置提示框按钮
dlg.setPositiveButton("确定(或者其他字符)", null);
// 显示提示框
dlg.show();
//...
```

NJS 代码：

```
/**
 * 在 Android 平台通过 NJS 显示系统提示框
 */
function njsAlertForAndroid(){
```

```
        // 导入 AlertDialog 类
        var AlertDialog = plus.android.importClass("android.app.AlertDialog");
        // 创建提示框构造对象，构造函数需要提供程序全局环境对象，通过plus.android.runtimeMainActivity()方法获取
        var dlg = new AlertDialog.Builder(plus.android.runtimeMainActivity());
        // 设置提示框标题
        dlg.setTitle("自定义标题");
        // 设置提示框内容
        dlg.setMessage("使用 NJS 的原生弹出框，可自定义弹出框的标题、按钮");
        // 设置提示框按钮
        dlg.setPositiveButton("确定(或者其他字符)",null);
        // 显示提示框
        dlg.show();
    }
    //...
```

**注意：**NJS 代码中创建提示框构造对象要求传入程序全局环境对象，可通过 plus.android.runtimeMainActivity()方法获取应用的主 Activity 对象，它是 HTML5+应用运行期自动创建的程序全局环境对象。

Android 设备上运行效果图：


注：其实 HTML5+规范已经封装过原生提示框消息 API：plus.ui.alert( message, alertCB, title, buttonCapture)。此处 NJS 的示例仅为了开发者方便理解，实际使用时调用 plus.ui.alert 更简单，性能也更高。

# iOS

以下代码在 iOS 平台展示调用 Native API 显示系统提示对话框。

iOS 原生 Objective-C 代码，用于比对参考：

```
#import <UIKit/UIKit.h>
//...
// 创建 UIAlertView 类的实例对象
UIAlertView *view = [UIAlertView alloc];
// 设置提示对话上的内容
[view initWithTitle:@"自定义标题" // 提示框标题
    message:@"使用 NJS 的原生弹出框，可自定义弹出框的标题、按钮" // 提示框上显示的内容
    delegate:nil // 点击提示框后的通知代理对象，nil 类似 js 的 null，意为不设置
    cancelButtonTitle:@"确定(或者其他字符)" // 提示框上取消按钮的文字
```

```
        otherButtonTitles:nil]; // 提示框上其它按钮的文字，设置为 nil 表示不显示
    // 调用 show 方法显示提示对话框，在 OC 中使用[]语法调用对象的方法
    [view show];
    //...
```

NJS 代码：

```
    /**
     * 在 iOS 平台通过 NJS 显示系统提示框
     */
    function njsAlertForiOS(){
        // 导入 UIAlertView 类
        var UIAlertView = plus.ios.importClass("UIAlertView");
        // 创建 UIAlertView 类的实例对象
        var view = new UIAlertView();
        // 设置提示对话上的内容
        view.initWithTitlemessagedelegatecancelButtonTitleotherButtonTitles("自定义标题" // 提示框标题
            , "使用 NJS 的原生弹出框，可自定义弹出框的标题、按钮" // 提示框上显示的内容
            , null // 操作提示框后的通知代理对象，暂不设置
            , "确定(或者其他字符)" // 提示框上取消按钮的文字
            , null ); // 提示框上其它按钮的文字，设置为 null 表示不显示
        // 调用 show 方法显示提示对话框，在 JS 中使用()语法调用对象的方法
        view.show();
    }
    //...
```

**注意：**在 OC 语法中方法的定义格式为:
“(返回值类型) 函数名: (参数 1 类型) 形参 1 参数 2 名称: (参数 2 类型) 形参 2”
方法的完整名称为: “函数名:参数 2 名称:”。
如:“（void）setPositionX:(int)x Y:(int)y;”，方法的完整名称为“setPositionX:Y:”
调用时语法为：“[pos setPositionX:x Y:y];”。
在 JS 语法中函数名称不能包含“:”字符，所以 OC 对象的方法名映射成 NJS 对象方法名时将其中的“:”字符自动删除，上面方法名映射为“setPositionXY”，在 NJS 调用的语法为：“pos.setPositionXY(x,y);”。

iOS 设备上运行效果图：

注：其实 HTML5+规范已经封装过原生提示框消息 API：plus.ui.alert( message, alertCB, title, buttonCapture)。此处 NJS 的示例仅为了开发者方便理解，实际使用时调用 plus.ui.alert 更简单、性能也更高。

在 HBuilder 自带的 Hello H5+模板应用中“Native.JS”（plus/njs.html）页面有完整的源代码，可真机运行查看效果。

# 三、常用 API

## API on Android

为了能更好的理解 NJS 调用 Java Native API，我们在 Android 平台用 Java 实现以下测试类，将在会后面 API 说明中的示例来调用。

文件 NjsHello.java 代码如下：

```
package io.dcloud;

// 定义类 NjsHello
public class NjsHello {
	// 静态常量
	public static final int CTYPE = 1;
	// 静态变量
	public static int count;
	// 成员常量
	public final String BIRTHDAY = "2013-01-13";
	// 成员变量
	String name; //定义属性 name
	NjsHelloEvent observer;
	public void updateName( String newname ) { //定义方法 updateName
		name = newname;
	}
	public void setEventObserver( NjsHelloEvent newobserver ) {
		observer = newobserver;
	}
	public void test() { //定义方法 test
		System.out.printf( "My name is: %s", name );
		observer.onEventInvoked( name );
	}
	public static void testCount() {
		System.out.printf( "Static count is:%d", count );
	}
	static{   // 初始化类的静态变量
		NjsHello.count = 0;
	}
}
```

文件 NjsHelloEvent.java 代码如下：

```
package io.dcloud;

// 定义接口 NjsHelloEvent
public interface NjsHelloEvent {
	public void onEventInvoked( String name );
}
```

注：此 NjsHello 示例仅为了说明原生代码与 NJS 代码之间的映射关系，以下示例代码无法直接在 HBuilder 里真机运行，必须在以后 HBuilder 开放自定义打包后方可把 NjsHello 类打

入 app 并被 NJS 调用。实际使用中，这种需要并非必要，大多数情况可以通过直接写 NJS 代码调用操作系统 API，而无需由原生语言二次封装类供 JS 调用。

# plus.android.importClass

导入 Java 类对象，方法原型如下：

```
ClassObject plus.android.importClass( String classname );
```

导入类对象后，就可以通过“.”操作符直接调用对象（类对象/实例对象）的常量和方法。

- classname：要导入的 Java 类名，必须是完整的命名空间（使用"."分割），如果指定的类名不存在，则导入类失败，返回 null。

**注意：**导入类对象可以方便的使用“.”操作符来调用对象的属性和方法，但也会消耗较多的系统资源。因此导入过多的类对象会影响性能，此时可以使用“高级 API”中提供的方法在不导入类对象的情况下调用 Native API。

**示例：**

1．导入类对象

Java 代码：

```
import io.dcloud.NjsHello;
//...
public class Test {
public static void main( String args[] ) {
    // 创建对象的实例
    NjsHello hello = new NjsHello();
    //...
}
//...
}
```

NJS 代码：

```
// 导入测试类 NjsHello
var NjsHello = plus.android.importClass("io.dcloud.NjsHello");
// 创建 NjsHello 的实例对象
var hello = new NjsHello();
// ...
```

# ClassObject

调用 plus.android.importClass()方法导入类并返回 ClassObject 类对象，通过该类对象，可以创建类的实例对象。在 Java 中类的静态方法会转换成 NJS 类对象的方法，可通过类对象的“.”操作符调用；类的静态常量会转换为 NJS 类对象的属性，可通过类对象的“.”操作符访问；类的静态属性则需通过 NJS 类对象的 plusGetAttribute、plusSetAttribute 方法操作。

**示例：**

1．导入类后获取类的静态常量属性

Java 代码：

```
import io.dcloud.NjsHello;
//...
public class Test {
public static void main( String args[] ) {
    // 获取类的静态常量属性
    int type = NjsHello.CTYPE;
    System.out.printf( "NjsHello Final's value: %d", type );   // 输出“NjsHello Final's value: 1”
    //...
}
//...
}
```

NJS 代码：

```
// 导入测试类 NjsHello
var NjsHello = plus.android.importClass("io.dcloud.NjsHello");
// 获取类的静态常量属性
var type = NjsHello.CTYPE;
console.log( "NjsHello Final's value: "+type ); // 输出“NjsHello Final's value: 1”
// ...
```

2．导入类后调用类的静态方法

Java 代码：

```
import io.dcloud.NjsHello;
//...
public class Test {
public static void main( String args[] ) {
    // 调用类的静态方法
    NjsHello.testCount();
    //...
}
//...
}
```

NJS 代码：

```
// 导入测试类 NjsHello
var NjsHello = plus.android.importClass("io.dcloud.NjsHello");
// 调用类的静态方法
NjsHello.testCount();
// ...
```

## ClassObject.plusGetAttribute

获取类对象的静态属性值，方法原型如下：

```
Object classobject.plusGetAttribute( String name );
```

导入类对象后，就可以调用其 plusGetAttribute 方法获取类的静态属性值。

- name：要获取的静态属性名称，如果指定的属性名称不存在，则获取属性失败，返回 null。

**注意：**如果导入的类对象中存在“plusGetAttribute”同名的静态方法，则必须通过 plus.android.invoke()方法调用。

**示例：**

1．导入类后获取类的静态属性值

Java 代码：

```
import io.dcloud.NjsHello;
//...
public class Test {
public static void main( String args[] ) {
    // 获取类的静态属性
    int count = NjsHello.count;
    System.out.printf( "NjsHello Static's value: %d", count );   // 输出“NjsHello Static's value: 0”
    //...
}
//...
}
```

NJS 代码：

```
// 导入测试类 NjsHello
var NjsHello = plus.android.importClass("io.dcloud.NjsHello");
// 获取类的静态属性
var count = NjsHello.plusGetAttribute( "count" );
console.log( "NjsHello Static's value: "+count ); // 输出“NjsHello Static's value: 0”
// ...
```

## ClassObject.plusSetAttribute

设置类对象的静态属性值，方法原型如下：

```
void classobject.plusSetAttribute( String name, Object value );
```

导入类对象后，就可以调用其 plusSetAttribute 方法设置类的静态属性值。

- name：要设置的静态属性名称，如果指定的属性名称不存在，则设置属性失败，返回 null。
- value：要设置的属性值，其类型必须与 Native 层类对象的静态属性区配，否则设置操作不生效，将保留以前的值。

**注意：**如果导入的类对象中存在“plusSetAttribute”同名的静态方法，则必须通过 plus.android.invoke()方法调用。

**示例：**

1．导入类后设置类的静态属性值

Java 代码：

```
import io.dcloud.NjsHello;
//...
public class Test {
public static void main( String args[] ) {
    // 设置类的静态属性值
    NjsHello.count = 2;
    System.out.printf( "NjsHello Static's value: %d", NjsHello.count );   // 输出“NjsHello Static's value: 2”
    //...
}
//...
}
```

NJS 代码：

```
// 导入测试类 NjsHello
var NjsHello = plus.android.importClass("io.dcloud.NjsHello");
// 设置类的静态属性值
NjsHello.plusSetAttribute( "count", 2 );
console.log( "NjsHello Static's value: "+NjsHello.plusGetAttribute( "count" ) ); // 输出“NjsHello Static's value: 2”
// ...
```

# InstanceObject

NJS 中实例对象与 Java 中的对象对应，调用 plus.android.importClass()方法导入类后，通过 new 操作符可创建该类的实例对象，或直接调用 plus.android.newObject 方法创建类的实例对象，也可通过调用 Native API 返回实例对象。在 Java 中对象的方法会转换成 NJS 实例对象的方法，可通过实例对象的“.”操作符调用；对象的常量属性会转换 NJS 实例对象的属性，可通过实例对象的“.”操作符访问。对象的非常量属性则必须通过 NJS 实例对象的 plusGetAttribute、plusSetAttribute 方法操作。

**示例：**

1．导入类创建实例对象，调用对象的方法

Java 代码：

```
import io.dcloud.NjsHello;
//...
public class Test {
public static void main( String args[] ) {
    // 创建 NjsHello 的实例对象
    NjsHello hello = new NjsHello();
    // 调用对象的方法
    hello.updateName( "Tester" );
    //...
}
//...
}
```

NJS 代码：

```
// 导入测试类 NjsHello
var NjsHello = plus.android.importClass("io.dcloud.NjsHello");
// 创建 NjsHello 的实例对象
var hello = new NjsHello();
// 调用对象的方法
hello.updateName( "Tester" );
// ...
```

2．导入类创建实例对象，获取对象的常量属性

Java 代码：

```
import io.dcloud.NjsHello;
//...
public class Test {
public static void main( String args[] ) {
    // 创建 NjsHello 的实例对象
    NjsHello hello = new NjsHello();
    // 访问对象的常量属性
    String birthday = hello.BIRTHDAY;
    System.out.printf( "NjsHello Object Final's value: %s", birthday );   // 输出“NjsHello Object Final's value: 2013-01-13”
    //...
}
//...
}
```

NJS 代码：

```
// 导入测试类 NjsHello
var NjsHello = plus.android.importClass("io.dcloud.NjsHello");
// 创建 NjsHello 的实例对象
var hello = new NjsHello();
// 访问对象的常量属性
var birthday = hello.BIRTHDAY;
console.log( "NjsHello Object Final's value: "+birthday ); // 输出“NjsHello Object Final's value: 2013-01-13”
// ...
```

## InstanceObject.plusGetAttribute

获取实例对象的属性值，方法原型如下：

```
Object instancebject.plusGetAttribute( String name );
```

获取实例对象后，就可以调用其 plusGetAttribute 方法获取对象的属性值。

- name：要获取对象的属性名称，如果指定的属性名称不存在，则获取属性失败，返回 null。

**注意：**如果实例对象中存在“plusGetAttribute”同名的方法，则必须通过 plus.android.invoke() 方法调用。

**示例：**

1．导入类创建实例对象，获取对象的属性值

Java 代码：

```
import io.dcloud.NjsHello;
//...
public class Test {
public static void main( String args[] ) {
    // 创建对象的实例
    NjsHello hello = new NjsHello();
    hello.updateName( "Tester" );
    // 获取其 name 属性值
    String name = hello.name;
    System.out.printf( "NjsHello Object's name: %s", name );   // 输出“NjsHello Object's name: Tester”
    //...
}
//...
}
```

NJS 代码：

```
// 导入测试类 NjsHello
var NjsHello = plus.android.importClass("io.dcloud.NjsHello");
// 创建对象的实例
var hello = new NjsHello();
hello.updateName( "Tester" );
// 获取其 name 属性值
var name = hello.plusGetAttribute( "name" );
console.log( "NjsHello Object's name: "+name );   // 输出“NjsHello Object's name: Tester”
// ...
```

## InstanceObject.plusSetAttribute

设置类对象的静态属性值，方法原型如下：

```
void instanceobject.plusSetAttribute( String name, Object value );
```

导入类对象后，就可以调用其 plusSetAttribute 方法设置类的静态属性值。

- name：要设置的静态属性名称，如果指定的属性名称不存在，则设置属性失败，返回 null。
- value：要设置的属性值，其类型必须与 Native 层类对象的静态属性区配，否则设置操作不生效，将保留以前的值。

**注意：**如果导入的类对象中存在“plusSetAttribute”同名的静态方法，则必须通过 plus.android.invoke()方法调用。

**示例：**

1．导入类创建实例对象，设置对象的属性值

Java 代码：

```
import io.dcloud.NjsHello;
```

```
    //...
    public class Test {
    public static void main( String args[] ) {
        // 创建对象的实例
        NjsHello hello = new NjsHello();
        // 设置其 name 属性值
        hello.name = "Tester";
        System.out.printf( "NjsHello Object's name: %s", hello.name );   // 输出“NjsHello Object's name:
Tester”
        //...
    }
    //...
    }
```

NJS 代码：

```
    // 导入测试类 NjsHello
    var Hello = plus.android.importClass("NjsHello");
    // 创建对象的实例
    var hello = new NjsHello();
    // 设置其 name 属性值
    hello.plusSetAttribute( "name", "Tester" );
    console.log( "NjsHello Object's name: "+hello.plusGetAttribute("name") ); // 输出“NjsHello Object's name:
Tester”
    // ...
```

# plus.android.implements

在 Java 中可以通过定义新类并实现 Interface 的接口，并创建出新类对象作为其它接口的参数，在 NJS 中则可快速创建对应的 Interface 对象，方法原型如下：

```
Object plus.android.implements( String name, Object obj );
```

此方法创建 Native 层中 Java 的接口实现对象，作为调用其它 Native API 的参数。

- name：接口的名称，必须是完整的命名空间（使用"."分割），如果不存在此接口，则创建接口实现对象失败，返回 null。
- obj：JSON 对象类型，接口实现方法的定义，JSON 对象中 key 值为接口方法的名称；value 值为 Function，方法参数必须与接口中方法定义的参数区配。

**示例：**

1．Test 类中实现接口 NjsHelloEvent 的方法，并调用 NjsHello 对象的 test 方法触发接口中函数的运行。

Java 代码：

```
    import io.dcloud.NjsHello;
    import io.dcloud.NjsHelloEvent;
    //...

    // Test 类实现 NjsHelloEvent 接口
    public class Test implements NjsHelloEvent {
    public static void main( String args[] ) {
        // 创建对象的实例
```

```
        NjsHello hello = new NjsHello();
        // 调用 updateName 方法
        hello.updateName( "Tester" );
        // 设置监听对象
        hello.setEventObserver( this );
        // 调用 test 方法，触发接口事件
        hello.test(); // 触发 onEventInvoked 函数运行
        //...
    }
    // 实现接口 NjsHelloEvent 的 onEventInvoked 方法
    @Override
    public void onEventInvoked( String name ) {
        System.out.printf( "Invoked Object's name is: %s", name );
    }
    //...
    }
```

NJS 代码：

```
    // 导入测试类 NjsHello
    var NjsHello = plus.android.importClass("io.dcloud.NjsHello");
    // 实现接口“NjsHelloEvent”对象
    var hevent = plus.android.implements( "io.dcloud.NjsHelloEvent", {
        "onEventInvoked":function( name ){
            console.log( "Invoked Object’s name: "+name ); // 输出“Invoked Object’s name: Tester”
        }
    } );
    // 创建对象的实例
    var hello = new NjsHello();
    // 调用 updateName 方法
    hello.updateName( "Tester" );
    // 设置监听对象
    hello.setEventObserver( hevent );
    // 调用 test 方法，触发代理事件
    hello.test(); // 触发上面定义的匿名函数运行
    // ...
```

# plus.android.runtimeMainActivity

获取运行期环境主 Activity 实例对象，方法原型如下：

```
InstanceObject plus.android.runtimeMainActivity();
```

此方法将获取程序的主 Activity 的实例对象，它是 Html5+运行期环境主组件，用于处理与用户交互的各种事件，也是应用程序全局环境 android.app.Activity 的实现对象。android.app.Activity 是一个特殊的类，需要在原生开发环境中注册后才能使用，所以使用 new 操作符创建对象无实际意义。

**示例：**

1．调用 Activity 的 startActivity 方法来拨打电话

Java 代码：

```
import android.app.Activity;
```

```
import android.content.Intent;
import android.net.Uri;

//...

// 获取主 Activity 对象的实例
Activity main = context;
// 创建 Intent
Uri uri = Uri.parse("tel:10086");
Intent call = new Intent("android.intent.action.CALL",uri);
// 调用 startActivity 方法拨打电话
main.startActivity(call);

//...
```

NJS 代码：

```
// 导入 Activity、Intent 类
var Intent = plus.android.importClass("android.content.Intent");
var Uri = plus.android.importClass("android.net.Uri");
// 获取主 Activity 对象的实例
var main = plus.android.runtimeMainActivity();
// 创建 Intent
var uri = Uri.parse("tel:10086");
var call = new Intent("android.intent.action.CALL",uri);
// 调用 startActivity 方法拨打电话
main.startActivity( call );
// ...
```

# API on iOS

为了能更好的理解 NJS 调用 Objective-C Native API，我们在 iOS 平台用 Objective-C 实现以下测试类，将会在后面 API 说明中的示例来调用。

头文件 njshello.h 代码如下：

```
// 定义协议
@protocol NjsHelloEvent <NSObject>
@required
-(void) onEventInvoked:(NSString*)name;
@end

// ------------------------------------------------------------

// 定义类 NjsHello
@interface NjsHello : NSObject {
    NSString *_name;
    id<NjsHelloEvent > _delegate;
}
@property (nonatomic,retain) NSString *name;
@property (nonatomic,retain) id delegate;
-(void)updateName:(NSString*)newname;
-(void)setEventObserver:(id<NjsHelloEvent >)delegate;
```

```
-(void)test;
+(void)testCount;
@end
```

实现文件 njshello.m 源代码如下：

```
#import "njshello.h"

// 实现类 NjsHello
@implementation NjsHello
@synthesize name=_name;
-(void)updateName:(NSString*)newname{
    _name = [newname copy];
}
-(void)setEventObserver:(id<NjsHelloEvent >)delegate{
    _delegate = delegate;
}
-(void)test{
    NSLog("My name is: %@",_name);
    [[self delegate]onEventInvoked:name];
}
-(void)dealloc{
    [_name release];
    [supper dealloc];
}
+(void)testCount{
     NSLog( "Static test count" );
}
@end
```

# plus.ios.importClass

导入 Objective-C 类对象，方法原型如下：

```
ClassObject plus.ios.importClass( String classname );
```

导入类对象后，就可以通过“.”操作符直接调用对象（类对象/实例对象）的常量和方法。通过“.”操作符号调用方法时，不需要使用“:”来分割方法名。

- classname：要导入的 Objective-C 类名，如果指定的类名不存在，则导入类失败，返回 null。

**注意：**导入类对象可以方便的使用“.”操作符来调用对象的属性和方法，但也会消耗较多的系统资源。因此导入过多的类对象会影响性能，此时可以使用“高级 API”中提供的方法在不导入类对象的情况下调用 Native API。

**示例：**

1．导入类并创建实例对象

Objective-C 代码：

```
#import "njshello.h"

int main( int argc, char *argv[] )
```

```
{
    // 创建对象的实例
    NjsHello* hello = [[NjsHello alloc] init];
    // ...
}
// ...
```

NJS 代码：

```
// 导入测试类 NjsHello
var NjsHello = plus.ios.importClass("NjsHello");
// 创建对象的实例
var hello = new NjsHello();
// ...
```

# ClassObject

调用 plus.ios.importClass()方法导入类并返回 ClassObject 类对象，通过该类对象，可以创建类的实例对象。在 Objective-C 中类的静态方法会转换成 NJS 类对象的方法，可通过类对象的“.”操作符调用；

**注意：**由于 Objective-C 中类没有静态变量，而是通过定义全局变量来实现，目前 NJS 中无法访问全局变量的值。对于全局常量，在 NJS 中也无法访问，对于原类型常量可在文档中找到其具体的值，在 JS 代码中直接赋值；对于非原类型常量目前还无法访问。

**示例：**

1．导入类后调用类的静态方法

Objective-C 代码：

```
#import "njshello.h"
// ...
int main( int argc, char *argv[] )
{
    // 调用类的静态方法
    [NjsHello testCount];
    // ...
}
// ...
```

NJS 代码：

```
// 导入测试类 NjsHello
var NjsHello = plus.ios.importClass("NjsHello");
// 调用类的静态方法
NjsHello.testCount();
// ...
```

# InstanceObject

NJS 中实例对象与 Objective-C 中的对象对应，调用 plus.ios.importClass()方法导入类后，通过 new 操作符可创建该类的实例对象，或直接调用 plus.ios.newObject 方法创建类的实例对象，也可通过调用 Native API 返回实例对象。在 Objective-C 中对象的方法会转换成 NJS 实例对象的方法，可通过实例对象的“.”操作符调用；对象的属性则必须通过 NJS 实例对象的 plusGetAttribute、plusSetAttribute 方法操作。

**示例：**

1．导入类创建实例对象，调用对象的方法

Objective-C 代码：

```
#import "njshello.h"

int main( int argc, char *argv[] )
{
    // 创建对象的实例
    NjsHello* hello = [[NjsHello alloc] init];
    // ...
}
// ...
```

NJS 代码：

```
// 导入测试类 NjsHello
var NjsHello = plus.ios.importClass("NjsHello");
// 创建对象的实例
var hello = new NjsHello();
// ...
```

# InstanceObject.plusGetAttribute

获取实例对象的属性值，方法原型如下：

```
Object instancebject.plusGetAttribute( String name );
```

获取实例对象后，就可以调用其 plusGetAttribute 方法获取对象的属性值。

- name：要获取对象的属性名称，如果指定的属性名称不存在，则获取属性失败，返回 null。

**注意：**如果实例对象中存在“plusGetAttribute”同名的方法，则只能通过 plus.ios.invoke()方法调用。

**示例：**

1．导入类创建实例对象，获取对象的属性值

Objective-C 代码：

```
#import "njshello.h"
```

```
int main( int argc, char *argv[] )
{
    // 创建对象的实例
    NjsHello* hello = [[NjsHello alloc] init];
    [hello updateName:@"Tester"];
    // 获取其 name 属性值
    NSString* name = hello.name;
    NSLog("NjsHello Object's name: %@",name);   // 输出“NjsHello Object's name: Tester”
    // ...
}
```

NJS 代码：

```
// 导入测试类 NjsHello
var NjsHello = plus.ios.importClass("NjsHello");
// 创建对象的实例
var hello = new NjsHello();
hello.updateName( "Tester" );
// 获取其 name 属性值
var name = hello.plusGetAttribute( "name" );
console.log( "NjsHello Object’s name: "+name );   // 输出“NjsHello Object’s name: Tester”
// ...
```

## InstanceObject.plusSetAttribute

设置类对象的静态属性值，方法原型如下：

```
void instanceobject.plusSetAttribute( String name, Object value );
```

导入类对象后，就可以调用其 plusSetAttribute 方法设置类的静态属性值。

- name：要设置的静态属性名称，如果指定的属性名称不存在，则设置属性失败，返回 null。
- value：要设置的属性值，其类型必须与 Native 层类对象的静态属性区配，否则设置操作不生效，将保留以前的值。

**注意：**如果导入的类对象中存在“plusSetAttribute”同名的静态方法，则只能通过 plus.android.invoke()方法调用。

**示例：**

1．导入类创建实例对象，设置对象的属性值

Java 代码：

```
#import "njshello.h"

int main( int argc, char *argv[] )
{
    // 创建对象的实例
    NjsHello* hello = [[NjsHello alloc] init];
    // 设置其 name 属性值
    hello.name = @"Tester";
```

```
        NSLog("NjsHello Object's name: %@",hello.name);   // 输出“NjsHello Object's name: Tester”
        // ...
    }
    //...
```

NJS 代码：

```
    // 导入测试类 NjsHello
    var NjsHello = plus.ios.importClass("NjsHello");
    // 创建对象的实例
    var hello = new NjsHello();
    // 设置其 name 属性值
    hello.plusSetAttribute( "name", "Tester" );
    console.log( "NjsHello Object’s name: "+hello.plusGetAttribute("name") ); // 输出“NjsHello Object’s name: Tester”
    // ...
```

# plus.ios.implements

在 Objective-C 中可以通过定义新类并实现 Protocol 的协议，并创建出新类对象作为代理对象，在 NJS 中则可实现协议快速创建代理对象，方法原型如下：

```
Object plus.ios.implements( String name, Object obj );
```

此方法返回一个 NJS 实例对象，映射到 Native 层中的代理对象，其父类为“NSObject”，并且实现 obj 中指定的协议方法。通常作为调用其它 Native API 的参数。

- name：协议的名称，也可以是自定的字符串名称用于定义一个代理。
- obj：JSON 对象类型，代理实现方法的定义，JSON 对象中 key 值为协议中定义的方法名称，必须保留方法名称中的“：”字符；value 值为 Function，方法参数必须与协议中定义方法的参数区配。

**示例：**

1．实现一个代理，并调用 test 方法触发调用代理的方法

Objective-C 代码：

```
    #import "njshello.h"

    // 定义代理类 NjsDelegate
    @interface NjsDelegate: NSObject<NjsHelloEvent> {
        -(void) onEventInvoked:(NSString*)name;
    }
    @end
    // ------------------------------------------------------------
    // 实现代理类 NjsDelegate
    @implementation NjsDelegate
    -(void) onEventInvoked:(NSString*)name{
        NSLog("Invoked Object's name:%@",name);   // 输出“Invoked Object’s name: Tester”
    }
    @end
    // ------------------------------------------------------------
    // 主函数
    int main( int argc, char *argv[] )
```

```
{
    // 创建对象的实例
    NjsHello* hello = [[NjsHello alloc] init];
    // 调用 updateName 方法
    [hello updateName:@"Tester"];
    // 创建代理对象
    NjsDelegate* delegate = [[NjsDelegate alloc] init];
    // 设置监听对象
    [hello setEventObserver:delegate];
    // 调用 test 方法，触发代理事件
    [hello test];  // 触发上面代理对象定义的 onEventInvoked 运行
    // ...
}
```

在 NJS 中不需要创建新的类对象，调用 plus.ios.implements 实现协议接口即可创建出代理对象，代码如下：

```
// 导入测试类 NjsHello
var NjsHello = plus.ios.importClass("NjsHello");
// 实现协议“NjsHelloEvent”的代理
var hevent = plus.ios.implements( "NjsHelloEvent", {
    "onEventInvoked":function( name ){
        console.log( "Invoked Object's name: "+name ); // 输出“Invoked Object's name: Tester”
    }
} );
// 调用 updateName 方法
hello.updateName( "Tester" );
// 设置监听对象
hello.setEventObserver( hevent );
// 调用 test 方法，触发代理事件
hello.test(); // 触发上面代理对象定义的匿名函数运行
// ...
```

# plus.ios.deleteObject

释放 NJS 中实例对象中映射的 Native 对象，方法原型如下：

```
void plus.ios.deleteObject( Object obj );
```

NJS 中所有的实例对象（InstanceObject）都可以通过此方法释放，会将 Native 层的对象使用的资源进行释放。

- obj：要释放的实例对象，如果 obj 对象不是有效的实例对象，则不执行对象的是否资源操作。

**注意：**此方法是可选的，如果不调用此方法释放实例对象，则在页面关闭时会自动释放所有对象；若对象占用较多的系统资源，则在业务逻辑处理完成时应该主动调用此方法释放资源，以提到程序的运行效率。

**示例：**

1．创建实例对象使用完成后，显式操作销毁对象

Objective-C 代码：

```
#import "njshello.h"

int main( int argc, char *argv[] )
{
    // 创建对象的实例
    NjsHello* hello = [[NjsHello alloc] init];
    // 调用 updateName 方法
    [hello updateName:@"Tester"];
    // ...
    // 使用完后销毁对象的实例
    [hello release];
}
```

NJS 代码：

```
// 导入测试类 NjsHello
var NjsHello = plus.ios.importClass("NjsHello");
// 创建对象的实例
var hello = new NjsHello();
// 调用 updateName 方法
hello.updateName( "Tester" );
// ...
// 使用完后销毁对象的实例
plus.ios.deleteObject( hello );
```

# 四、完整业务演示

## Android

在 Android 手机桌面上创建快捷方式图标，这是原本只有原生程序才能实现的功能。即使使用 Hybrid 方案，也需要原生工程师来配合写插件。

下面我们演示如何直接使用 js 在 Android 手机桌面创建快捷方式，在 HelloH5+应用中 Native.JS 页面中“Shortcut (Android)”可以查看运行效果。

这段代码是使用原生 Java 实现的创建快捷方式的代码，用于参考比对：

```
import android.app.Activity;
import android.content.Intent;
import android.graphics.BitmapFactory;
import android.graphics.Bitmap;

// 创建桌面快捷方式
void createShortcut(){
    // 获取主 Activity
    Activity main = this;
    // 创建快捷方式意图
    Intent shortcut = new Intent("com.android.launcher.action.INSTALL_SHORTCUT");
    // 设置快捷方式的名称
    shortcut.putExtra(Intent.EXTRA_SHORTCUT_NAME, "HelloH5+");
    // 设置不可重复创建
    shortcut.putExtra("duplicate",false);
    // 设置快捷方式图标
    Bitmap bitmap = BitmapFactory.decodeFile("/sdcard/icon.png");
    shortcut.putExtra(Intent.EXTRA_SHORTCUT_ICON, bitmap);
    // 设置快捷方式启动执行动作
    Intent action = new Intent(Intent.ACTION_MAIN);
    action.setComponent( main.getComponentName() );
    shortcut.putExtra( Intent.EXTRA_SHORTCUT_INTENT, action );
    // 广播创建快捷方式
    main.sendBroadcast(shortcut);
}
```

使用 NJS 实现时首先导入需要使用到的 android.content.Intent 、android.graphics.BitmapFactory 类，按照 Java 代码中的方法对应转换成 JavaScript 代码。

其中快捷方式图标是通过解析本地 png 文件进行设置，在 JavaScript 中需要使用 plus.io.* API 转换成本地路径传递给 Native API，完整代码如下：

```
var Intent=null,BitmapFactory=null;
var main=null;
document.addEventListener( "plusready", function() {//"plusready"事件触发时执行 plus 对象的方法
    // ...
    if ( plus.os.name == "Android" ) {
        // 导入要用到的类对象
        Intent = plus.android.importClass("android.content.Intent");
        BitmapFactory = plus.android.importClass("android.graphics.BitmapFactory");
        // 获取主 Activity
        main = plus.android.runtimeMainActivity();
```

```
            }
        }, false);
        /**
         * 创建桌面快捷方式
         */
        function createShortcut(){
            // 创建快捷方式意图
            var shortcut = new Intent("com.android.launcher.action.INSTALL_SHORTCUT");
            // 设置快捷方式的名称
            shortcut.putExtra(Intent.EXTRA_SHORTCUT_NAME, "测试快捷方式");
            // 设置不可重复创建
            shortcut.putExtra("duplicate",false);
            // 设置快捷方式图标
            var iconPath = plus.io.convertLocalFileSystemURL("/icon.png"); // 将相对路径资源转换成系统绝对路径
            var bitmap = BitmapFactory.decodeFile(iconPath);
            shortcut.putExtra(Intent.EXTRA_SHORTCUT_ICON,bitmap);
            // 设置快捷方式启动执行动作
            var action = new Intent(Intent.ACTION_MAIN);
            action.setComponent(main.getComponentName());
            shortcut.putExtra(Intent.EXTRA_SHORTCUT_INTENT,action);
            // 广播创建快捷方式
            main.sendBroadcast(shortcut);
            console.log( "桌面快捷方式已创建完成！" );
        }
```

**注意**：在真机运行时需要添加 Android 权限才能在桌面创建快捷方式，在 HBuilder 的工程中双击“manifest.json”文件，切换到“代码视图”中添加权限“<uses-permission android:name="com.android.launcher.permission.INSTALL_SHORTCUT"/>”，如下图所示：


# iOS

在 iOS 手机上登录 game center，一个游戏中心服务，这是原本只有原生程序才能实现的功能。即使使用 Hybrid 方案，也需要原生工程师来配合写插件。

下面我们演示如何直接使用 js 在 iOS 手机上登录 game center，在 HelloH5+应用中 Native.JS 页面中的“Game Center (iOS)”可以查看运行效果。

注意手机未开通 game center 则无法登陆，请先点击 iOS 自带的 game center 进行配置。

这段代码是使用原生 Objective-C 实现的登录 game center 的代码，用于参考比对。原生 Objective-C 代码的头文件 Test.h 中代码如下：

```
@interface Test: NSObject
// 游戏玩家登录状态监听函数
- (void)authenticationChanged:(NSNotification*)notification;
// 获取游戏玩家状态信息
- (void)playerInformation:(GKPlayer *)player;
// 登录到游戏中心
- (void)loginGamecenter;
// 停止监听登录游戏状态变化
- (void)logoutGamecenter;
@end
```

实现文件 Test.m 中代码如下：

```
@implementation Test
// 游戏玩家登录状态监听函数
- (void)authenticationChanged:(NSNotification*)notification
{
    // 获取游戏玩家共享实例对象
    GKLocalPlayer *player = notification.object;
    if ( player.isAuthenticated ) {
        // 玩家已登录认证，获取玩家信息
        [self playerInformation:player];
    } else {
        // 玩家未登录认证，提示用户登录
        NSLog(@"请登录！");
    }
    // 释放使用的对象
    [player release];
}
// 获取游戏玩家状态信息
- (void)playerInformation:(GKPlayer *)player
{
    // 获取游戏玩家的名称
    NSLog(@"Name: %@",player.displayName);
}

// 登录到游戏中心
- (void)loginGamecenter
{
    // 监听用户登录状态变更事件
    NSNotificationCenter *nc = [NSNotificationCenter defaultCenter];
    [nc addObserver:self
            selector:@selector(authenticationChanged)
                name:@"GKPlayerAuthenticationDidChangeNotificationName"
              object:nil];
    // 获取游戏玩家共享实例对象
    GKLocalPlayer *localplayer = [GKLocalPlayer localPlayer];
    // 判断游戏玩家是否已经登录认证
    if ( localplayer.isAuthenticated ) {
        // 玩家已登录认证，获取玩家信息
        [self playerInformation:localplayer];
    } else {
        // 玩家未登录认证，发起认证请求
        [localplayer authenticateWithCompletionHandler:nil];
        NSLog(@"登录中...");
    }
```

```
        // 释放使用的对象
        [localplayer release];
        [nc release];
    }

    // 停止监听登录游戏状态变化
    - (void)logoutGamecenter
    {
        // 取消监听用户登录状态变化
        NSNotificationCenter *nc = [NSNotificationCenter defaultCenter];
        [nc removeObserver:self
                      name:@"GKPlayerAuthenticationDidChangeNotificationName"
                    object:nil];
        // 释放使用的对象
        [nc release];
    }
    @end
```

使用 NJS 实现时可以按照 Objective-C 代码中的方法对应转换成 JavaScript 代码，最关键的代码是 loginGamecenter 方法中对用户登录状态的监听，需调用 NSNotificationCenter 对象的“addObserver:selector:name:object”方法，

1．addObserver:后要求传入一个实例对象用于查找 selector 参数中指定的方法，在 Objective-C 中通常将对象自身（self）传入，但在 NJS 中没有此概念，因此需使用 plus.ios.implements 方法来创建一个新的对象：

```
var delegate = plus.ios.implements("NSObject",{"authenticationChanged:":authenticationChanged});
```

第一个参数“NSObject”表示对象的类型，第二个参数中的 JSON 对象表明对象拥有的方法，“authenticationChanged”方法是 delegate 对象的方法。

2．selector:后要传入一个类函数指针，在 Objective-C 中通过“@selector”指令可选择函数指针，在 NJS 中则需使用 plus.ios.newObject 方法来创建一个函数对象：

```
plus.ios.newObject("@selector","authenticationChanged:")
```

第一个参数需固定值为“@selector”，表示创建的是类函数指针对象，第二个参数。

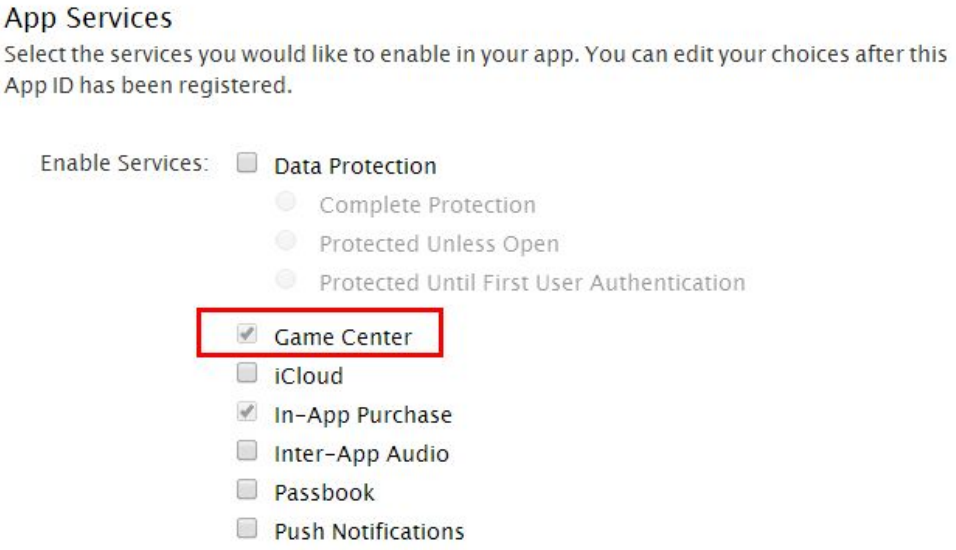
在"plusready"事件中导入 GKLocalPlayer 和 NSNotificationCenter 类，并调用登录方法 longinGamecenter()，完整 JavaScript 代码如下：

```
// 处理"plusready"事件
var bLogin=false;
document.addEventListener( "plusready", function() {
    // ...
    if ( plus.os.name == "iOS" ) {
        GKLocalPlayer    = plus.ios.importClass("GKLocalPlayer");
        NSNotificationCenter = plus.ios.importClass("NSNotificationCenter");
        longinGamecenter();
    } else {
        alert("欢迎您");
        bLogin = true;
        setTimeout( function(){
            plus.ui.toast( "此平台不支持 Game Center 功能！" );
        }, 500 );
    }
}, false);

var GKLocalPlayer=null,NSNotificationCenter=null;
var delegate=null;
```

```
// 游戏玩家登录状态监听函数
function authenticationChanged( notification ){
	// 获取游戏玩家共享实例对象
	var player = notification.plusGetAttribute("object");
	if ( player.plusGetAttribute("isAuthenticated") ) {
		// 玩家已登录认证，获取玩家信息
		playerInformation(player);
		bLogin = true;
	} else {
		// 玩家未登录认证，提示用户登录
		alert("请登录");
		bLogin = false;
	}
	// 释放使用的对象
	plus.ios.deleteObject(player);
}

// 获取游戏玩家状态信息
function playerInformation( player ){
	var name = player.plusGetAttribute("displayName");
	alert( name+" 已登录！ " );
}

// 登录到游戏中心
function longinGamecenter(){
	if ( bLogin ){
		return;
	}
	// 监听用户登录状态变更事件
	var nc = NSNotificationCenter.defaultCenter();
	delegate = plus.ios.implements("NSObject",{"authenticationChanged:":authenticationChanged});
	nc.addObserverselectornameobject(delegate,
	 plus.ios.newObject("@selector","authenticationChanged:"),
	 "GKPlayerAuthenticationDidChangeNotificationName",
	 null);
	// 获取游戏玩家共享实例对象
	var localplayer = GKLocalPlayer.localPlayer();
	// 判断游戏玩家是否已经登录认证
	if ( localplayer.isAuthenticated() ) {		// localplayer.plusGetAttribute("isAuthenticated")
		// 玩家已登录认证，获取玩家信息
		playerInformation( localplayer );
		bLogin = true;
	} else {
		// 玩家未登录认证，发起认证请求
		localplayer.authenticateWithCompletionHandler(null);
		alert( "登录中..." );
	}
	// 释放使用的对象
	plus.ios.deleteObject(localplayer);
	plus.ios.deleteObject(nc);
}

// 停止监听登录游戏状态变化
function stopGamecenterObserver()
{
	// 取消监听用户登录状态变化
	var nc = NSNotificationCenter.defaultCenter();
```

```
        nc.removeObservernameobject(delegate,"GKPlayerAuthenticationDidChangeNotificationName",null);
        plus.ios.deleteObject(nc);
        plus.ios.deleteObject(delegate);
        delegate = null;
    }
```

**注意**：正式发布提交到 AppStore 时，在配置苹果开发者网站上配置 App ID 需要选中“Game Center”服务：




# 五、开发注意和建议用途

Native.js 的运行性能仍然不比纯原生应用；JS 与 Native 之间的数据交换效率并不如在 js 内部的数据交换效率；基于如上原因，有几点开发建议：

- 以标准 web 代码为主，当遇到 web 能力不足的时候，调用 Native.js。
- 以标准 web 代码为主，当遇到 web 性能不足的时候，需要分析，

  if ((原生进行运算的效率-js 与原生通讯的损耗)>纯 web 的效率){

  使用 Native.js

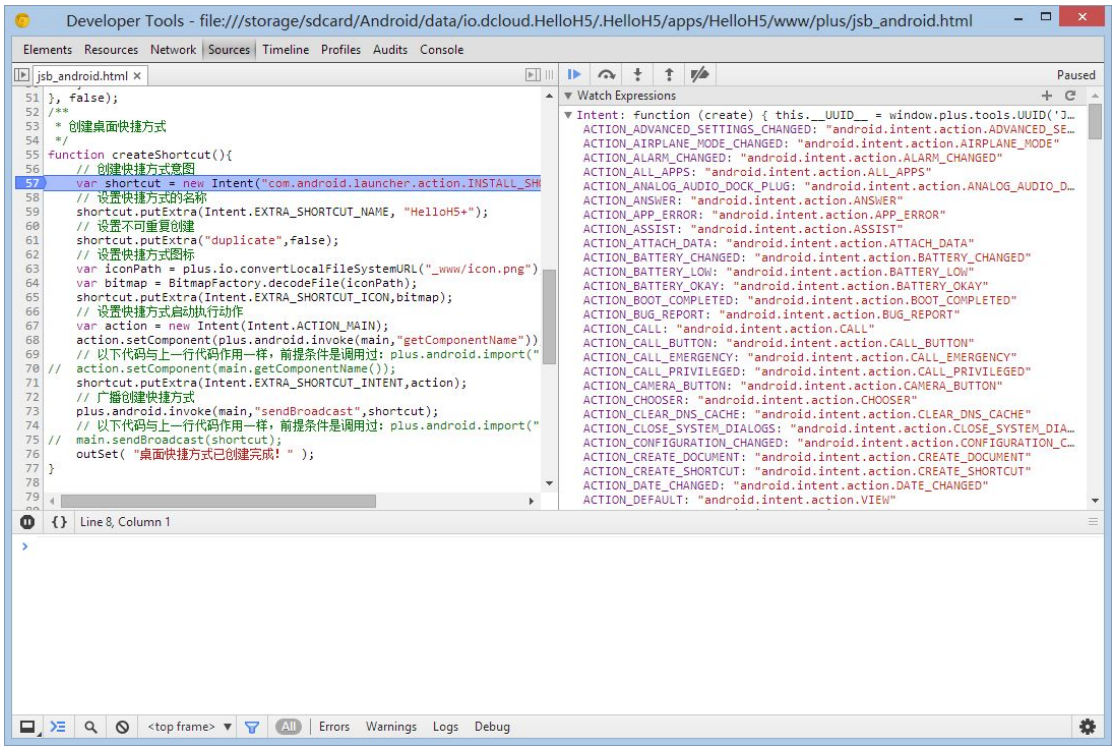
  }else{

  还应该使用纯 js

  }
- 应避免把程序设计为在短时间内并发触发 Native.js 代码

# 六、调试

使用 safari 和 chrome 的控制台调试 HBuilder 的 5+App 时，一样可以调试 NJS 对象，即可以在浏览器控制台中查看各种原生对象的属性和方法，如下图所示，57 行设了断点，watch 了 Intent 对象，并在右边展开了该对象的所有属性方法：




关于如何在浏览器控制台调试 HBuilder 的 5+App，请参考 HBuilder 的 5+App 开发入门教程。

# 七、开发资源

iOS 官方在线文档：https://developer.apple.com/library/ios/navigation/

Android 官方在线文档：https://developer.android.com/reference/packages.html

# 八、高级 API

有前述的常用 API，已经可以完成各项业务开发。此处补充的高级 API，是在熟悉 NJS 后，为了提升性能而使用的 API。高级 API 无法直接用“.”操作符使用原生对象的方法，在 debug 时也无法 watch 原生对象，但高级 API 性能高于常规 API。

虽然导入类对象（plus.android.importClass 和 plus.ios.importClass）后，可以方便的通过“.”操作符来访问对象的常量、调用对象的方法，但导入类对象也需要消耗较多的系统资源，所以在实际开发时应该尽可能的减少导入类对象，以提高程序效率。可以参考以下依据进行判断：

1. 如导入的类特别复杂，继承自很多基类，方法和属性特别多则考虑不导入类；
2. 对导入类是否需要频繁操作，若导入类仅是为了实例化，并作为调用其它 API 的参数，则不应该导入类；
3. 在同一页面中是否导入了很多类？如果导入太多则需要考虑减少导入类的数目；

如果我们不导入类对象则无法通过 new 操作符实例化类对象，这时可通过 plus.ios.newObject()、plus.android.newObject()方法来创建实例对象，如下：

```
// iOS 平台创建 NSDictionary 的实例对象
var ns = plus.ios.newObject( "NSDictionary" );

// Android 平台创建 Intent 的实例对象
var intent = plus.android.newObject( "android.content.Intent" );
```

# API on Android

## plus.android.newObject

不导入类对象直接创建类的实例对象，方法原型如下：

```
InstanceObject plus.android.newObject( String classname, Object...args );
```

此方法对 Native 层中对类进行实例化操作，创建一个类的实体并返回 NJS 层的实例对象。相比导入类对象后使用 new 操作符创建对象效率要高。

- classname：要创建实例对象的类名，类名必须是完整的命名空间，使用“.”分隔符（如“android.app.AlertDialog”），如果需要创建内部类对象需要使用“$”分割符（如“android.app.AlertDialog$Builder”）。如果指定的类名不存在，则创建对象失败，返回 null。
- args：调用类构造函数的参数，其类型和数目必须与 Native 层 Java 类构造函数区配，否则无法创建类对象，将返回 null。

**注意：**由于没有导入类对象，所以通过此方法创建的实例对象无法通过“.”操作符直接调用对象的方法，而必须使用 plus.android.invoke 方法来调用。

**示例：**

1．不导入类创建实例对象

Java 代码：

```
import io.dcloud.NjsHello;
//...
public class Test {
public static void main( String args[] ) {
```

```
        // 创建对象的实例
        NjsHello hello = new NjsHello();
        //...
    }
    //...
    }
```

NJS 代码：

```
    // 不调用 plus.android.importClass("io.dcloud.NjsHello")导入类 NjsHello
    // 创建对象的实例
    var hello = plus.android.newObject( "io.dcloud.NjsHello" );
    // ...
```

# plus.android.getAttribute

不导入类对象，则无法通过类对象并访问类的静态属性，需调用以下方法获取类的静态属性值，方法原型如下：

```
Object plus.android.getAttribute( String|Object obj, String name );
```

此方法也可以获取类对象或实例对象的属性值，如果是类对象获取的则是类的静态属性，如果是实例对象则获取的是对象的非静态属性。

- obj：若是 String 类型，表示要获取静态属性值的类名，类名必须是完整的命名空间（使用"."分割）；若是 ClassObject 类型，表示要获取静态属性的类对象；若是 InstanceObject 类型，表示要获取属性值的实例对象。
- name：要获取的属性名称，如果指定的属性名称不存在，则获取属性失败，返回 null。

**注意：**同样导入类对象后也可以调用此方法，obj 参数类型为 ClassObject 时，其作用与 ClassObject.plusSetAttribute 方法一致。obj 参数类型为 InstanceObject 时，其作用与 InstanceObject.plusSetAttribute 方法一致。

**示例：**

1．不导入类对象获取类的静态常量属性

Java 代码：

```
    import io.dcloud.NjsHello;
    //...
    public class Test {
    public static void main( String args[] ) {
        // 获取类的静态常量属性
        int type = NjsHello.CTYPE;
        System.out.printf( "NjsHello Final's value: %d", type );   // 输出“NjsHello Final's value: 1”
        // 获取类的静态属性
        int count = NjsHello.count;
        System.out.printf( "NjsHello Static's value: %d", count );   // 输出“NjsHello Static's value: 0”
        //...
    }
    //...
```

```
}
```

NJS 代码：

```
// 不调用 plus.android.importClass("io.dcloud.NjsHello")导入类 NjsHello
// 访问类的静态常量属性
var type = plus.android.getAttribute( "NjsHello", "CTYPE" );
console.log( "NjsHello Final's value: "+type ); // 输出“NjsHello Final's value: 1”
// ...
```

2．不导入类对象，创建实例对象，并获取其 name 属性值

Java 代码：

```
import io.dcloud.NjsHello;
//...
public class Test {
public static void main( String args[] ) {
    // 创建对象的实例
    NjsHello hello = new NjsHello();
    // 获取其 name 属性值
    String name = hello.name;
    System.out.printf( "NjsHello Object's name: %s", name );   // 输出“NjsHello Object's name: Tester”
    //...
}
//...
}
```

NJS 代码：

```
// 不调用 plus.android.importClass("io.dcloud.NjsHello")导入类 NjsHello
// 创建对象的实例
var hello = plus.android.newObject( "io.dcloud.NjsHello" );
// 获取其 name 属性值
var name = plus.android.getAttribute( hello, "name" );
console.log( "NjsHello Object's name: "+name );   // 输出“NjsHello Object's name: Tester”
// ...
```

# plus.android.setAttribute

若没有导入类对象，则无法通过类对象设置类的静态属性值，需调用以下方法设置类的静态属性值，方法原型如下：

```
void plus.android.setAttribute( String|Object obj, String name, Object value );
```

此方法也可以设置类对象或实例对象的属性值，如果是类对象设置的则是类的静态属性，如果是实例对象则设置的是对象的非静态属性。

- obj：若是 String 类型，表示要设置静态属性值的类名，类名必须是完整的命名空间（使用"."分割）；若是 ClassObject 类型，表示要设置静态属性的类对象；若是 InstanceObject 类型，表示要设置属性值的实例对象。
- name：要设置的属性名称，如果指定的属性名称不存在，则设置属性失败，返回 null。

- value：要设置的属性值，其类型必须与 Native 层 obj 对象的属性区配，否则设置操作不生效，将保留以前的值。

**注意：**同样导入类对象后也可以调用此方法，obj 参数类型为 ClassObject 时，其作用与 ClassObject.plusSetAttribute 方法一致。obj 参数类型为 InstanceObject 时，其作用与 InstanceObject.plusSetAttribute 方法一致。

**示例：**

1．不导入类对象设置类的静态属性值

Java 代码：

```
import io.dcloud.NjsHello;
//...
public class Test {
public static void main( String args[] ) {
    // 设置类的静态属性值
    NjsHello.count = 2;
    System.out.printf( "NjsHello Static's value: %d", NjsHello.count );   // 输出“NjsHello Static's value: 2”
    //...
}
//...
}
```

NJS 代码：

```
// 不调用 plus.android.importClass("io.dcloud.NjsHello")导入类 NjsHello
// 设置类的静态属性值
plus.android.setAttribute( "io.dcloud.NjsHello", "count", 2 );
console.log( "NjsHello Static's value: "+plus.android.getAttribute("NjsHello","count") ); // 输出“NjsHello Static's value: 2”
// ...
```

2．不导入类对象，创建实例对象，并设置其 name 属性值

Java 代码：

```
import io.dcloud.NjsHello;
//...
public class Test {
public static void main( String args[] ) {
    // 创建对象的实例
    NjsHello hello = new NjsHello();
    // 设置其 name 属性值
    hello.name = "Tester";
    System.out.printf( "NjsHello Object's name: %s", hello.name );   // 输出“NjsHello Object's name: Tester”
    //...
}
//...
}
```

NJS 代码：

```
// 不调用 plus.android.importClass("io.dcloud.NjsHello")导入类 NjsHello
// 创建对象的实例
var hello = plus.android.newObject( "io.dcloud.NjsHello" );
```

```
    // 设置其 name 属性值
    plus.android.setAttribute( hello, "name", "Tester" );
    console.log( "NjsHello Object's name: "+hello.plusGetAttribute("name") ); // 输出“NjsHello Object's name: Tester”
    // ...
```

# plus.android.invoke

若没有导入类对象，则无法通过实例对象的“.”操作符调用其成员方法，需通过以下方法调用实例对象的成员方法，方法原型如下：

```
Object plus.android.invoke( String|Object obj, String name, Object... args );
```

此方法也可以调用类对象或实例对象的方法，如果是类对象则调用的是类的静态方法，如果是实例对象则调用的是对象的普通成员方法。函数返回值是调用 Native 层方法运行后的返回值，Native 对象的方法无返回值则返回 undefined。

- obj：若是 String 类型，表示要调用静态方法的类名，类名必须包含完整的包名；若是 ClassObject 类型，表示要调用静态方法的类对象；若是 InstanceObject 类型，表示要调用成员方法的实例对象。
- name：要调用的方法名称，如果指定的方法不存在，则调用方法失败，返回值为 null。
- args：调用方法的参数，其类型和数目必须与 Native 层对象方法的函数区配，否则无法调用对象的方法，将返回 null。

**注意：**同样导入类对象后也可以调用此方法，其作用与通过类对象或实例对象的“.”操作符调用方法作用一致。

**示例：**

1．不导入类对象，调用类的静态方法

Java 代码：

```
import io.dcloud.NjsHello;
//...
public class Test {
public static void main( String args[] ) {
    // 调用类的静态方法
    NjsHello.testCount();
    //...
}
//...
}
```

NJS 代码：

```
// 不调用 plus.android.importClass("io.dcloud.NjsHello")导入类 NjsHello
// 调用类的静态方法
plus.android.invoke( "io.dcloud.NjsHello", "testCount" );
// ...
```

2．不导入类对象，创建实例对象，并调用其 updateNmae 方法

Java 代码：

```
import io.dcloud.NjsHello;
//...

public class Test {
public static void main( String args[] ) {
    // 创建对象的实例
    NjsHello hello = new NjsHello();
    // 调用 updateName 方法
    hello.updateName( "Tester" );
    System.out.printf( "NjsHello Object's name: %s", name );   // 输出“NjsHello Object's name: Tester”
    //...
}
//...
}
```

NJS 代码：

```
// 不调用 plus.android.importClass("io.dcloud.NjsHello")导入类 NjsHello
// 创建对象的实例
var hello = plus.android.newObject( "io.dcloud.NjsHello" );
// 调用 updateName 方法
plus.android.invoke( hello, "updateName", "Tester" );
console.log( "NjsHello Object's name: "+hello.getAttribute("name") ); // 输出“NjsHello Object's name:
Tester”
// ...
```

# API on iOS

## plus.ios.newObject

不导入类对象直接创建类的实例对象，方法原型如下：

```
InstanceObject plus.ios.newObject( String classname, Object..args );
```

此方法会在 Native 层中对类进行实例化操作，创建一个类的实体并返回 NJS 层的类实例对象。相比导入类对象后使用 new 操作符创建对象效率要高。

- classname：要创建实例对象的类名，如果指定的类名不存在，则创建对象失败，返回 null。
- args：调用类构造函数的参数，其类型和数目必须与 Native 层对象构造函数区配，否则无法创建类对象，将返回 null。

**注意：**由于没有导入类对象，所以通过此方法创建的实例对象无法通过“.”操作符直接调用对象的方法，而必须使用 plus.ios.invoke 方法来调用。classname 参数值为“@selector”表示需要创建一个函数指针对象，与 Objective-C 中的@selector 指令功能相似，args 参数为函数的名称，此时函数的名称需要包含“:”字符。

**示例：**

1．不导入类创建实例对象

Objective-C 代码：

```
#import "njshello.h"

int main( int argc, char *argv[] )
{
    // 创建对象的实例
    NjsHello* hello = [[NjsHello alloc] init];
    // ...
}
```

NJS 代码：

```
// 未导入“NjsHello”类
// 创建对象的实例
var hello = plus.ios.newObject( "NjsHello" );
// ...
```

# plus.ios.invoke

若没有导入类对象，则无法通过实例对象的“.”操作符调用其成员方法，需通过以下方法调用实例对象的成员方法，方法原型如下：

```
Object plus.ios.invoke( String|Object obj, String name, Object... args );
```

此方法也可以调用类对象或实例对象的方法，如果是类对象则调用的是类的静态方法，如果是实例对象则调用的是对象的普通成员方法。函数返回值是调用 Native 层方法运行后的返回值，Native 对象的方法无返回值则返回 undefined。

- obj：若是 String 类型，表示要调用静态方法的类名，类名必须包含完整的包名；若是 ClassObject 类型，表示要调用静态方法的类对象；若是 InstanceObject 类型，表示要调用成员方法的实例对象。
- name：要调用的方法名称，必须保留方法名称中的“：”字符，如果指定的方法不存在，则调用方法失败，返回值为 null。
- args：调用方法的参数，其类型和数目必须与 Native 层对象方法的函数区配，否则无法调用对象的方法，将返回 null。

**注意：**同样导入类对象后也可以调用此方法，其作用与通过类对象或实例对象的“.”操作符调用方法作用一致。

**示例：**

1．不导入类创建实例对象，并调用 updateName 方法

Objective-C 代码：

```
#import "njshello.h"

int main( int argc, char *argv[] )
{
    // 创建对象的实例
```

```
        NjsHello* hello = [[NjsHello alloc] init];
        // 调用 updateName 方法
        [hello updateName:@"Tester"];

        NSLog("NjsHello Object's name: %@",hello.name);   // 输出“NjsHello Object's name: Tester”
        // ...
    }
```

NJS 代码：

```
    // 未导入“NjsHello”类
    // 创建对象的实例
    var hello = plus.ios.newObject( "NjsHello" );
    // 调用 updateName 方法
    plus.ios.invoke( hello, "updateName", "Tester" );
    console.log( "NjsHello Object's name: "+hello.getAttribute("name") ); // 输出“NjsHello Object's name:
Tester”
    // ...
```

# 九、性能优化

## 调整代码结构优化

前面章节中我们介绍如何通过 NJS 调用 Native API 来显示系统提示框，在真机运行时会发现第一次调用时会有 0.5s 左右的延时，再次调用则不会延时。这是因为 NJS 中导入类对象操作会花费较长的时间，再次调用时由于类对象已经导入过，会能很快执行完毕。因此可以调整代码结构进行优化，在页面打开后触发的“plusready”事件中进行类对象的导入操作，从而避免第一次调用的延时。

Android 平台调整 NJS 代码结构如下：

```
    // 保存 Android 导入对象和全局环境对象
    var AlertDialog=null,mainActivity=null;
    // H5+事件处理
    document.addEventListener("plusready",function(){
        switch ( plus.os.name ) {
            case "Android":
            // 程序全局环境对象，内部自动导入 Activity 类
            mainActivity = plus.android.runtimeMainActivity();
            // 导入 AlertDialog 类
            AlertDialog = plus.android.importClass("android.app.AlertDialog");
            break;
            default:
            break;
        }
    },false);
    //...
    /**
     * 在 Android 平台通过 NJS 显示系统提示框
     */
    function njsAlertForAndroid(){
```

```
        // 创建提示框构造对象，构造函数需要提供程序全局环境对象，通过plus.android.runtimeMainActivity()方法获取
        var dlg = new AlertDialog.Builder(mainActivity);
        // 设置提示框标题
        dlg.setTitle("自定义标题");
        // 设置提示框内容
        dlg.setMessage("使用 NJS 的原生弹出框，可自定义弹出框的标题、按钮");
        // 设置提示框按钮
        dlg.setPositiveButton("确定(或者其他字符)",null);
        // 显示提示框
        dlg.show();
    }
    //...
```

iOS 平台调整 NJS 代码结构如下：

```
    // 保存 iOS 平台导入的类对象
    var UIAlertView=null;
    // H5+事件处理
    document.addEventListener("plusready",function(){
        switch ( plus.os.name ) {
            case "iOS":
            // 导入 UIAlertView 类
            UIAlertView = plus.ios.importClass("UIAlertView");
            break;
            default:
            break;
        }
    },false);
    //...
    /**
     * 在 iOS 平台通过 NJS 显示系统提示框
     */
    function njsAlertForiOS(){
        // 创建 UIAlertView 类的实例对象
        var view = new UIAlertView();
        // 设置提示对话上的内容
        view.initWithTitlemessagedelegatecancelButtonTitleotherButtonTitles("自定义标题" // 提示框标题
            , "使用 NJS 的原生弹出框，可自定义弹出框的标题、按钮" // 提示框上显示的内容
            , null // 操作提示框后的通知代理对象，暂不设置
            , "确定(或者其他字符)" // 提示框上取消按钮的文字
            , null ); // 提示框上其它按钮的文字，设置为 null 表示不显示
        // 调用 show 方法显示提示对话框
        view.show();
    }
    //...
```

# 使用高级 API 优化

前面章节中我们提到导入类对象会消耗较多的系统资源，导入过多的类对象会影响性能。在高级 API 中提供一组接口可以在不导入类对象的情况下调用 Native API，从而提升代码运行性能。

Android 平台使用高级 API 优化代码如下：

```
// 保存 Android 导入对象和全局环境对象
var mainActivity=null;
// H5+事件处理
document.addEventListener("plusready",function(){
    switch ( plus.os.name ) {
        case "Android":
        // 程序全局环境对象，内部自动导入 Activity 类
        mainActivity = plus.android.runtimeMainActivity();
        break;
        default:
        break;
    }
},false);
//...
/**
 * 在 Android 平台通过 NJS 显示系统提示框
 */
function njsAlertForAndroid(){
    // 由于 Builder 类是 android.app.AlertDialog 类的内部类，这里需要使用$符号分割
    var dlg = plus.android.newObject("android.app.AlertDialog$Builder",mainActivity);
    // 设置提示框标题
    plus.android.invoke(dlg,"setTitle","自定义标题");
    // 设置提示框内容
    plus.android.invoke(dlg,"setMessage","使用 NJS 的原生弹出框，可自定义弹出框的标题、按钮");
    // 设置提示框按钮
    plus.android.invoke(dlg,"setPositiveButton","确定(或者其他字符)",null);
    // 显示提示框
    plus.android.invoke(dlg,"show");
}
//...
```

iOS 平台使用高级 API 优化代码如下：

```
/**
 * 在 iOS 平台通过 NJS 显示系统提示框
 */
function njsAlertForiOS(){
    // 创建 UIAlertView 类的实例对象
    var view = plus.ios.newObject("UIAlertView");
    // 设置提示对话上的内容，这里的方法名称中必须包含':'字符
    plus.ios.invoke(view,"initWithTitle:message:delegate:cancelButtonTitle:otherButtonTitles:"
        ,"自定义标题" // 提示框标题
        , "使用 NJS 的原生弹出框，可自定义弹出框的标题、按钮" // 提示框上显示的内容
        , null // 操作提示框后的通知代理对象，暂不设置
        , "确定(或者其他字符)" // 提示框上取消按钮的文字
        , null ); // 提示框上其它按钮的文字，设置为 null 表示不显示
    // 调用 show 方法显示提示对话框，在 JS 中使用()语法调用对象的方法
    plus.ios.invoke(view,"show");
}
//...
```