实验二十四　ArkWeb

一、实验目的

1. 了解 DevEco Studio 的使用
2. 学习 ArkTS 语言，ArkWeb
3. 编写代码
4. 编译运行
5. 在模拟器上运行

二、实验原理

1. 鸿蒙开发原理
2. ArkTS，ArkUI 开发原理
3. 鸿蒙应用运行原理

三、实验仪器材料

1. 计算机实训室电脑一台
2. DevEco Studio 开发环境及鸿蒙手机模拟器

四、实验步骤

这个实验我们学习 ArkWeb 的使用方法：

https://developer.huawei.com/consumer/cn/doc/harmonyos-guides-V5/web-component-overview-V5

ArkWeb（方舟 Web）提供了 Web 组件，用于在应用程序中显示 Web 页面内容。常见使用场景包括：

- **应用集成 Web 页面**：应用可以在页面中**使用 Web 组件**，**嵌入 Web 页面内容**，以降低开发成本，提升开发、运营效率。
- **浏览器网页浏览场景：浏览器类应用**可以使用 Web 组件，打开三方网页，使用无痕模式浏览 Web 页面，设置广告拦截等。
- **小程序：小程序类宿主应用**可以使用 Web 组件，渲染小程序的页面。

Web 组件为开发者提供了丰富的控制 Web 页面能力。包括：

- **Web 页面加载：声明式加载** Web 页面和**离屏加载** Web 页面等。
- **生命周期管理**：组件生命周期状态变化，通知 Web 页面的加载状态变化等。
- **常用属性与事件**：UserAgent 管理、Cookie 与存储管理、字体与深色模式管理、权限管理等。
- **与应用界面交互**：自定义文本选择菜单、上下文菜单、文件上传界面等与应用界面交互能力。
- **App 通过 JavaScriptProxy，与 Web 页面进行 JavaScript 交互**。
- **安全与隐私**：无痕浏览模式、广告拦截、坚盾守护模式等。
- **维测能力**：Devtools 工具调试能力，使用 crashpad 收集 Web 组件崩溃信息。
- **其他高阶能力**：与原生组件同层渲染、Web 组件的网络托管、Web 组件的媒体播放托管、Web 组件输入框拉起自定义输入法、网页接入密码保险箱等。

我们主要通过实例来学习。

1. 打开 DevEco Studio，点击 Create Project 创建工程

设置项目名称为 WebDemo。

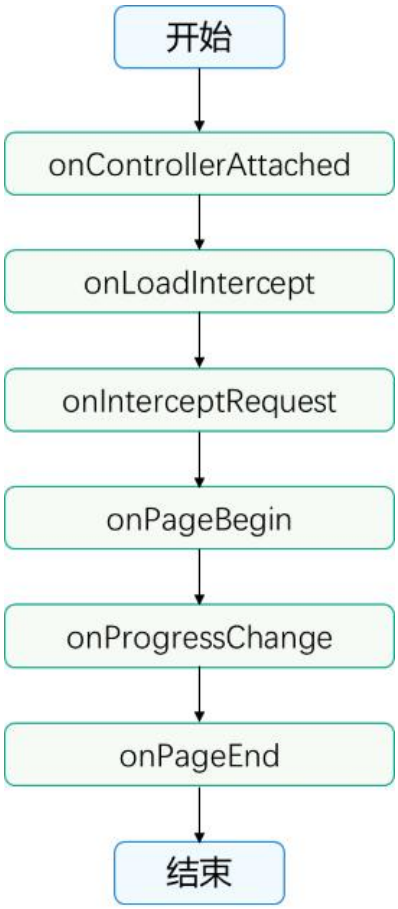
2. Web 组件的生命周期

开发者可以使用 Web 组件加载本地或者在线网页。Web 组件提供了丰富的**组件生命周期回调接口**，通过这些回调接口，开发者可以感知 Web 组件的生命周期状态变化，进行相关的业务处理。

Web 组件的状态主要包括：

- Controller 绑定到 Web 组件
- 网页加载**开始**
- 网页加载**进度**
- 网页加载**结束**
- 页面**即将可见**

对应的 Web 组件网页正常加载过程中的回调事件示意图:



加上页面/组件本身的生命周期函数，Web 组件网页加载的状态说明：

- aboutToAppear 函数：在创建自定义组件的新实例后，在执行其 build 函数前执行。
- onControllerAttached 事件：当 Controller 成功绑定到 Web 组件时触发该回调，且禁止在该事件回调前调用 Web 组件相关的接口，否则会抛出 js-error 异常。
- onLoadIntercept 事件：当 Web 组件加载 url 之前触发该回调，用于判断是否阻止此次访问。默认允许加载。
- onOverrideUrlLoading 事件：当 URL 将要加载到当前 Web 中时，让宿主应用程序有机会获得控制权，回调函数返回 true 将导致当前 Web 中止加载 URL，而返回 false 则会导致 Web 继续照常加载 URL。
- onInterceptRequest 事件：当 Web 组件加载 url 之前触发该回调，用于拦截 url 并返回响应数据。
- onPageBegin 事件：网页开始加载时触发该回调，且只在主 frame（表示一个 HTML 元素，用于展示 HTML 页面的 HTML 元素）触发。如果是 iframe 或者 frameset（用于包含 frame 的 HTML 标签）的内容加载时则不会触发此回调。

- onProgressChange 事件：告知开发者当前页面加载的进度。多 frame 页面或者子 frame 有可能还在继续加载而主 frame 可能已经加载结束，所以在 onPageEnd 事件后依然有可能收到该事件。
- onPageEnd 事件：网页加载完成时触发该回调，且只在主 frame 触发。多 frame 页面有可能同时开始加载，即使主 frame 已经加载结束，子 frame 也有可能才开始或者继续加载中。
- onPageVisible 事件：Web 回调事件。渲染流程中当 HTTP 响应的主体开始加载，新页面即将可见时触发该回调。此时文档加载还处于早期，因此链接的资源比如在线 CSS、在线图片等可能尚不可用。
- onRenderExited 事件：应用渲染进程异常退出时触发该回调，可以在此回调中进行系统资源的释放、数据的保存等操作。如果应用希望异常恢复，需要调用 loadUrl 接口重新加载页面。
- onDisAppear 事件：组件卸载消失时触发此回调。该事件为通用事件，指组件从组件树上卸载时触发的事件。

首先，在 rawfile 中创建一个 index.html 文件：

就是一个基本的 HTML 页面：

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
</head>
```

```
<body>
    <h1>Hello, ArkWeb</h1>
</body>
</html>
```

把 Index.ets 中的 Index 的代码改为：

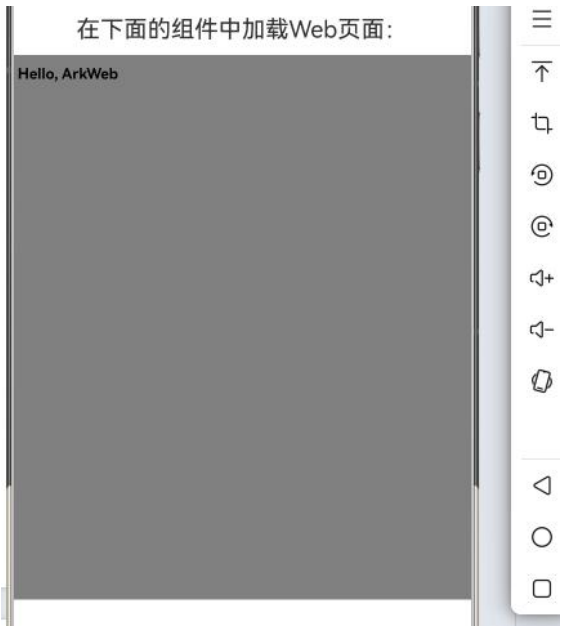
```
struct WebComponent {
  controller: webview.WebviewController = new webview.WebviewController();
  build() {
    Column({space: 10}) {
      Text('在下面的组件中加载 Web 页面：')
        .fontSize(20)
        .margin({top: 30})
      Web({ src: $rawfile('index.html'), controller: this.controller })
        .backgroundColor(Color.Gray)
        .height('60%')
    }.width('100%')
    .height('100%')
    .justifyContent(FlexAlign.Center)
  }
}
```

记得导入需要的包

```
import { webview } from '@kit.ArkWeb';
```

Previewer 是不支持这个组件的：

在模拟器中运行，效果：



可以看到，最基础的使用就是定义一个 webview controller，然后直接使用 Web 组件，设置好 src 和 controller 即可。这里 src 是加载我们之前存在 rawfile 中的 index.html 文件。

我们逐步添加生命周期函数，先添加 aboutToAppear()函数，在 build()前面加上代码：

```
  aboutToAppear(): void {
    try {
      webview.WebviewController.setWebDebuggingAccess(true);
    } catch (error) {
      console.error(`ErrorCode: ${(error as BusinessError).code}, Message: ${(error as BusinessError).message}`);
```

```
    }
  }
```

相应地，文件最前面加上导入 BusinessError 代码：

```
import { BusinessError } from '@kit.BasicServicesKit';
```

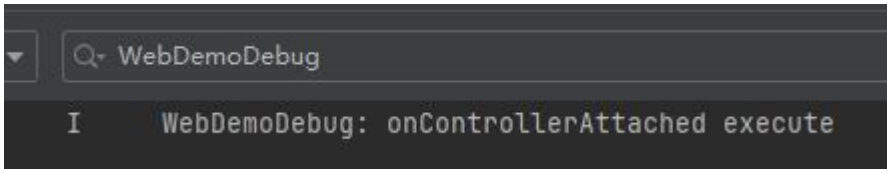
在 aboutToAppear 中，通常把 Debug 开关打开：

```
      webview.WebviewController.setWebDebuggingAccess(true);
```

加上 onControllerAttached: （红色字体）

```
      Web({ src: $rawfile('index.html'), controller: this.controller })
        .onControllerAttached(() => {
          // 推荐在此 loadUrl、设置自定义用户代理、注入 JS 对象等
          console.log('WebDemoDebug: onControllerAttached execute')
        })
        .backgroundColor(Color.Gray)
        .height('60%')
```

再次运行，查看 Log：

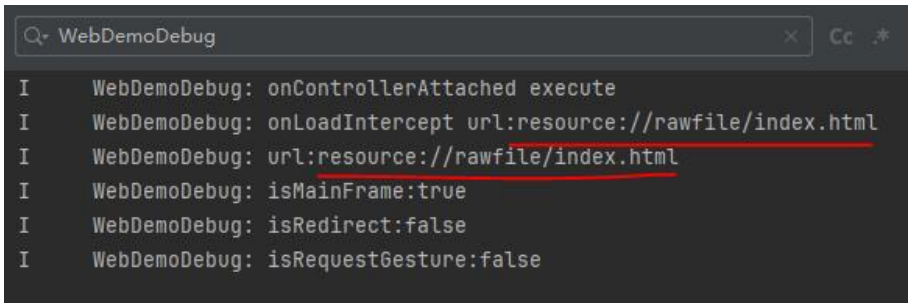


再添加 onLoadIntercept 事件：当 Web 组件加载 url 之前触发该回调，用于判断是否阻止此次访问。默认允许加载，我们让其返回 true，也就是阻止此次访问：

```
        .onLoadIntercept((event) => {
          if (event) {
```

```
            console.log('WebDemoDebug: onLoadIntercept url:' +
event.data.getRequestUrl())
            console.log('WebDemoDebug: url:' + event.data.getRequestUrl())
            console.log('WebDemoDebug: isMainFrame:' + event.data.isMainFrame())
            console.log('WebDemoDebug: isRedirect:' + event.data.isRedirect())
            console.log('WebDemoDebug: isRequestGesture:' +
event.data.isRequestGesture())
        }
        // 返回 true 表示阻止此次加载，否则允许此次加载
        return true
    })
```

模拟器上运行，查看 Log：



可以看到，页面加载被成功拦截了：

我们把后面的 return true 改为 return false:

页面被正常加载。

再添加 onOverrideUrlLoading 事件：当 URL 将要加载到当前 Web 中时，让宿主应用程序有机会获得控制权，回调函数返回 true 将导致当前 Web 中止加载 URL，而返回 false 则会导致 Web 继续照常加载 URL：

```
        .onOverrideUrlLoading((webResourceRequest: WebResourceRequest) => {
          if (webResourceRequest && webResourceRequest.getRequestUrl() ==
"about:blank") {
            return true;
          }
          return false;
        })
```

当用户点击链接或通过 JavaScript 触发页面跳转时，会触发此回调。检查要访问的地址是不是“about:blank”，如果是则会被拦截。

再添加 onInterceptRequest 函数，首先添加 WebResourceResponse，Header 数组和 webData 的定义：（红色字体）

```
struct WebComponent {
  controller: webview.WebviewController = new webview.WebviewController();
  responseWeb: WebResourceResponse = new WebResourceResponse();
  heads: Header[] = new Array();
  @State webData: string = "<!DOCTYPE html>\n" +
```

```
    "<html>\n" +
    "<head>\n" +
    "<title>intercept test</title>\n" +
    "</head>\n" +
    "<body>\n" +
    "<h1>intercept test</h1>\n" +
    "</body>\n" +
    "</html>";

  aboutToAppear(): void {
```

然后添加事件函数：

```
   .onInterceptRequest((event) => {
            if (event) {
              console.log('WebDemoDebug: url:' + event.request.getRequestUrl());
            }
            let head1: Header = {
              headerKey: "Connection",
              headerValue: "keep-alive"
            }
            let head2: Header = {
              headerKey: "Cache-Control",
              headerValue: "no-cache"
```

```
            }
            this.heads.push(head1);
            this.heads.push(head2);
            this.responseWeb.setResponseHeader(this.heads);
            this.responseWeb.setResponseData(this.webData);
            this.responseWeb.setResponseEncoding('utf-8');
            this.responseWeb.setResponseMimeType('text/html');
            this.responseWeb.setResponseCode(200);
            this.responseWeb.setReasonMessage('OK');
            // 返回响应数据则按照响应数据加载，无响应数据则返回 null 表示按照原来的方式加载
            return this.responseWeb;
        })
```

此时的效果：

可以看到，页面被成功拦截，显示这里给出的 responseWeb，也就是 webData 给出的内容。

删除掉这段代码。然后我们一次性添加 onPageBegin, onFirstContentfulPaint, onProgressChange, onPageEnd, onPageVisible, onRenderExited, onDisAppear 事件函数，这些函数基本上都是打印一下 Log:

```
        .onPageBegin((event) => {
```

```
            if (event) {
                console.log('onPageBegin url:' + event.url);
            }
        })
        .onFirstContentfulPaint(event => {
            if (event) {
                console.log("onFirstContentfulPaint:" + "[navigationStartTick]:" +
                event.navigationStartTick + ", [firstContentfulPaintMs]:" +
                event.firstContentfulPaintMs);
            }
        })
        .onProgressChange((event) => {
            if (event) {
                console.log('newProgress:' + event.newProgress);
            }
        })
        .onPageEnd((event) => {
            // 推荐在此事件中执行 JavaScript 脚本
            if (event) {
                console.log('onPageEnd url:' + event.url);
            }
        })
```

```
        .onPageVisible((event) => {
          console.log('onPageVisible url:' + event.url);
        })
        .onRenderExited((event) => {
          if (event) {
            console.log('onRenderExited reason:' + event.renderExitReason);
          }
        })
        .onDisAppear(() => {
          promptAction.showToast({
            message: 'The web is hidden',
            duration: 2000
          })
        })
```

由于最后用到了 promptAction，需要在文件头部导入一下：

```
import { promptAction } from '@kit.ArkUI';
```

再次运行，查看 Log：

```
WebDemoDebug
I     WebDemoDebug: onControllerAttached execute
I     WebDemoDebug: onLoadIntercept url:resource://rawfile/index.html
I     WebDemoDebug: url:resource://rawfile/index.html
I     WebDemoDebug: isMainFrame:true
I     WebDemoDebug: isRedirect:false
I     WebDemoDebug: isRequestGesture:false
I     WebDemoDebug: newProgress:10
I     WebDemoDebug: onPageBegin url:resource://rawfile/index.html
I     WebDemoDebug: newProgress:70
I     WebDemoDebug: onPageEnd url:resource://rawfile/index.html
I     WebDemoDebug: newProgress:100
I     WebDemoDebug: newProgress:100
I     WebDemoDebug: onPageVisible url:resource://rawfile/index.html
```

为了展示 onDisAppear，可以添加一个@State 变量：

```
@State show: boolean = true;
```

然后给 Text 添加一个事件：

```
.onClick(()=>{
    this.show = !this.show;
})
```

然后把整个 Web 组件包在：

```
if(this.show) {}
```

里面。

点击文本，可以看到这个 onDisAppear 被调用了：

Web 组件网页加载的性能指标

网页加载过程中需要关注一些重要的性能指标。例如，FCP(First Contentful Paint)首次内容绘制，

FMP(First Meaningful Paint)首次有效绘制，LCP(Largest Contentful Paint)最大内容绘制等。Web 组件提供了如下接口来通知开发者。

- onFirstContentfulPaint 事件：网页首次内容绘制的回调函数。首次绘制文本、图像、非空白 Canvas 或者 SVG 的时间点。
- onFirstMeaningfulPaint 事件：网页首次有效绘制的回调函数。首次绘制页面主要内容的时间点。
- onLargestContentfulPaint 事件：网页绘制页面最大内容的回调函数。可视区域内容最大的可见元素开始出现在页面上的时间点。

3. 设置基本属性和事件

- 设置 UserAgent

从 API version 11 起，Web 组件基于 ArkWeb 的内核，默认 UserAgent 定义如下：

*Mozilla/5.0 ({**deviceType**}; {**OSName**} {**OSVersion**}) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/114.0.0.0 Safari/537.36 ArkWeb/{**ArkWeb VersionCode**} {**Mobile**}*

字段	含义	备注
deviceType	设备类型	通过系统参数 const.product.devicetype映射得到。
OSName	发行版操作系统名称	通过系统参数 const.product.os.dist.name得到。
OSVersion	发行版操作系统版本	通过系统参数 const.product.os.dist.version解析版本号得到。
ArkWeb VersionCode	ArkWeb版本号	-
Mobile（可选）	是否是手机设备	-

举例：

Mozilla/5.0 (Phone; OpenHarmony5.0) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/114.0.0.0 Safari/537.36 ArkWeb/4.1.6.1 Mobile

1）通过 getUserAgent()接口获取当前默认用户代理

在 module.json5 中添加访问 INTERNET 权限：

```
    "requestPermissions": [
```

```
        {
          "name": "ohos.permission.INTERNET",
          "usedScene": {
            "when": "always"
          }
        }
    ],
```

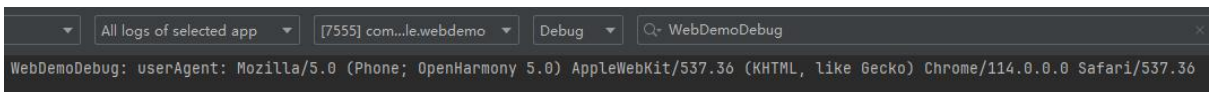
Index.ets 改为：

```
import { webview } from '@kit.ArkWeb';
import { BusinessError } from '@kit.BasicServicesKit';


@Entry
@Component
struct WebComponent {
  controller: webview.WebviewController = new webview.WebviewController();


  build() {
    Column() {
      Button('getUserAgent')
        .onClick(() => {
          try {
            let userAgent = this.controller.getUserAgent();
```

```
                console.log("WebDemoDebug: userAgent: " + userAgent);
            } catch (error) {
                console.error(`ErrorCode: ${(error as BusinessError).code},  Message: ${(error as BusinessError).message}`);
            }
        })
        Web({ src: 'https://www.baidu.com', controller: this.controller })
    }
  }
}
```

模拟器上运行，效果：

点击上方的 getUserAgent 按钮，查看 Log：



2）通过 setCustomUserAgent()接口设置自定义用户代理，覆盖系统的用户代理

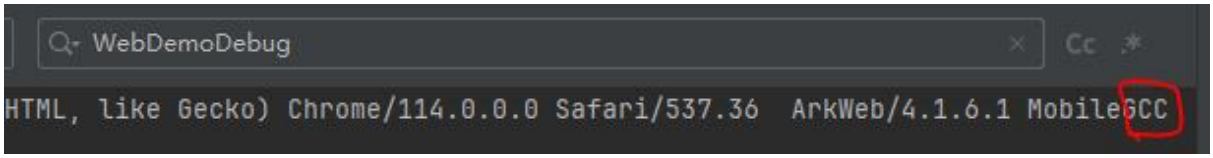
将 Index.ets 代码改为：

```
import { webview } from '@kit.ArkWeb';
import { BusinessError } from '@kit.BasicServicesKit';

@Entry
@Component
struct WebComponent {
  controller: webview.WebviewController = new webview.WebviewController();
  @State ua: string = "";

  aboutToAppear(): void {
    webview.once('webInited', () => {
      try {
        // 应用侧用法示例，定制 UserAgent。
        this.ua = this.controller.getUserAgent() + 'GCC';
        this.controller.setCustomUserAgent(this.ua);
        console.log("WebDemoDebug: userAgent: " + this.ua);
      } catch (error) {
        console.error(`ErrorCode: ${(error as BusinessError).code},  Message: ${(error as BusinessError).message}`);
      }
    })
```

```
  }

  build() {
    Column() {
      Web({ src: 'www.baidu.com', controller: this.controller })
    }
  }
}
```

效果：

查看 Log，可以看到’GCC’被加到了最后：



也就是说，用户可以去修改 UserAgent 中的信息，然后通过调用 setCustomUserAgent()接口设置自定义用户代理。

- 管理 Cookie 及数据存储

1）Cookie 管理

Cookie 是网络访问过程中，由**服务端发送给客户端**的一小段数据。客户端可持有该数据，并在后续访问该服务端时，方便服务端快速对客户端身份、状态等进行识别。

当 Cookie SameSite 属性未指定时，默认值为 SameSite=Lax，只在用户导航到 cookie 的源站点时发送 cookie，不会在跨站请求中被发送。Web 组件提供了 **WebCookieManager 类**，用于管理 Web 组件的 Cookie 信息。Cookie 信息保存在应用沙箱路径下 /proc/{pid}/root/data/storage/el2/base/cache/web/Cookiesd 的文件中。

将 Index.ets 改为：

```
import { webview } from '@kit.ArkWeb';
import { BusinessError } from '@kit.BasicServicesKit';

@Entry
@Component
struct WebComponent {
  controller: webview.WebviewController = new webview.WebviewController();

  build() {
    Column({space: 10}) {
      Button('configCookieSync')
        .onClick(() => {
          try {
            webview.WebCookieManager.configCookieSync('https://www.baidu.com',
'username=gcctest');
```

```
            } catch (error) {
                console.error(`ErrorCode: ${(error as BusinessError).code},  Message: ${(error as BusinessError).message}`);
            }
        })
        Button('fetchCookieSync')
          .onClick(() => {
            try {
                let username: string = webview.WebCookieManager.fetchCookieSync('https://www.baidu.com');
                console.log('WebDemoDebug: username in Cookie is: ' + username)
            } catch (error) {
                console.error(`ErrorCode: ${(error as BusinessError).code},  Message: ${(error as BusinessError).message}`);
            }
        })
        Web({ src: 'www.baidu.com', controller: this.controller })
    }
  }
}
```

这里我们尝试了两个函数：configCookieSync 和 fetchCookieSync

效果：

点击 configCookieSync 按钮，然后再点击 fetchCookieSync，查看 Log（可能要拉到后面）：

注意：

系统会自动清理过期的 cookie，对于同名 key 的数据，新数据将会覆盖前一个数据。

为了获取可正常使用的 cookie 值，fetchCookieSync 需传入完整链接。fetchCookieSync 用于获取所有的 cookie 值，每条 cookie 值之间会通过"; "进行分隔，但无法单独获取某一条特定的 cookie 值。

2）缓存与存储管理

在访问网站时，网络资源请求是相对比较耗时的。开发者可以通过 **Cache**、**Dom Storage** 等手段将资源保存到本地，以提升访问同一网站的速度。

Cache

使用 cacheMode()配置页面资源的缓存模式，Web 组件为开发者提供四种缓存模式，分别为：

- Default：优先使用未过期的缓存，如果缓存不存在，则从网络获取。
- None：加载资源使用 cache，如果 cache 中无该资源则从网络中获取。
- Online：加载资源不使用 cache，全部从网络中获取。
- Only ：只从 cache 中加载资源。

Index.ets 代码：

```
import { webview } from '@kit.ArkWeb';
```

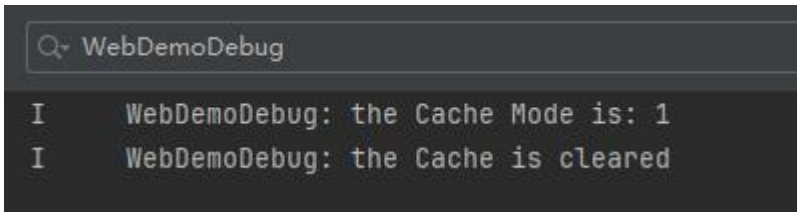
```
import { BusinessError } from '@kit.BasicServicesKit';


@Entry
@Component
struct WebComponent {
  @State mode: CacheMode = CacheMode.None;
  controller: webview.WebviewController = new webview.WebviewController();


  build() {
    Column() {
      Button('removeCache')
        .onClick(() => {
          try {
            // 设置为 true 时同时清除 rom 和 ram 中的缓存，设置为 false 时只清除 ram 中的缓存
            console.log('WebDemoDebug: the Cache Mode is: ' + this.mode.toString())
            this.controller.removeCache(true);
            console.log('WebDemoDebug: the Cache is cleared')
          } catch (error) {
            console.error(`ErrorCode: ${(error as BusinessError).code},  Message: ${(error as BusinessError).message}`);
          }
```

```
        })
        Web({ src: 'www.baidu.com', controller: this.controller })
          .cacheMode(this.mode)
      }
    }
}
```

可以看到：

- 使用 cacheMode()配置页面资源的缓存模式
- 通过 removeCache()接口清除已经缓存的资源

点击 removeCache 按钮，查看 Log：



- 设置深色模式

Web 组件支持对前端页面进行深色模式配置。

通过 **darkMode()接口**可以配置不同的深色模式，默认关闭。当深色模式开启时，Web 将启用媒体查询 prefers-color-scheme 中网页所定义的深色样式，若网页未定义深色样式，则保持原状。如需开启**强制深色模式**，建议配合 **forceDarkAccess()**使用。WebDarkMode.Off 模式表示关闭深色模式。WebDarkMode.On 表示开启深色模式，并且深色模式跟随前端页面。WebDarkMode.Auto 表示开启深

色模式，并且深色模式跟随系统。

先把 index.html 改为：

```
<!-- index.html -->
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width,
                                   initial-scale=1.0,
                                   maximum-scale=1.0,
                                   user-scalable=no">
    <style type="text/css">
        @media (prefers-color-scheme: dark) {
            .contentCss{ background:   #000000; color: white; }
            .hrefCss{ color: #317AF7; }
        }
    </style>
</head>
<body class="contentCss">
<div style="text-align:center">
    <p>Dark mode debug page</p>
</div>
</body>
```

```
</html>
```

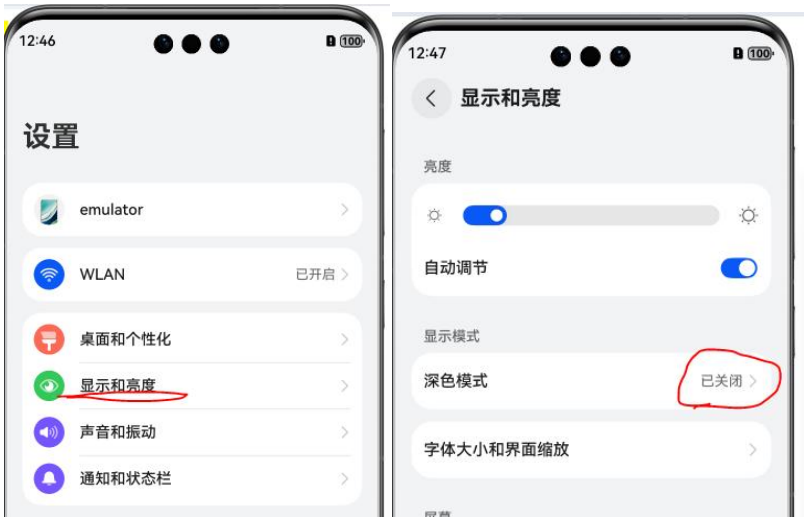
注意 css 文件中的红色字体这个@media 配置。

Index.ets 改为：

```
import { webview } from '@kit.ArkWeb';


@Entry
@Component
struct WebComponent {
  controller: webview.WebviewController = new webview.WebviewController();
  @State mode: WebDarkMode = WebDarkMode.Auto;


  build() {
    Column() {
      Web({ src: $rawfile('index.html'), controller: this.controller })
        .darkMode(this.mode)
    }
  }
}
```

这里配置为 Auto，表示跟随系统。

模拟器上先查看“显示和亮度”：



在已关闭的状态下，运行效果：

还是 Light 模式。

去模拟器的设置里面，将深色模式打开：

然后在模拟器上打开应用（不必再次编译运行）：

可以看到跟随系统模式的效果，现在是 Dark 模式了。

先关掉这个深色模拟。

再尝试通过 forceDarkAccess()接口可将前端页面强制配置深色模式，强制深色模式无法保证所有颜色转换符合预期，且深色模式不跟随前端页面和系统。配置该模式时候，需要将深色模式配置成 WebDarkMode.On。

Index.ets 的代码改为：

```
import { webview } from '@kit.ArkWeb';

@Entry
@Component
struct WebComponent {
  controller: webview.WebviewController = new webview.WebviewController();
  @State mode: WebDarkMode = WebDarkMode.On;
  @State access: boolean = true;

  build() {
    Column() {
      Web({ src: $rawfile('index.html'), controller: this.controller })
        .darkMode(this.mode)
        .forceDarkAccess(this.access)
    }
  }
```

```
}
```

可以看到，直接强制就是深色模式了：

- 在新窗口打开页面

Web 组件提供了**在新窗口打开页面**的能力，开发者可以通过 **multiWindowAccess**()接口来**设置是否允许**网页在新窗口打开。当有新窗口打开时，应用侧会在 onWindowNew()接口中收到 Web 组件新窗口事件，开发者需要在此接口事件中，新建窗口来处理 Web 组件窗口请求。

说明

- allowWindowOpenMethod()接口设置为 true 时，前端页面通过 JavaScript 函数调用的方式打开新窗口。
- 如果开发者在 onWindowNew() 接口通知中不需要打开新窗口，需要将 ControllerHandler.setWebController()接口参数设置成 null。

在 rawfile 中新建一个 window.html:

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"/>
    <title>WindowEvent</title>
</head>
<body>
<input type="button" value="新窗口中打开网页" onclick="OpenNewWindow()">
```

```
<script type="text/javascript">
    function OpenNewWindow()
    {
        var txt = '打开的窗口';
        let openedWindow = window.open("about:blank", "",
"location=no,status=no,scrollvars=no");
        openedWindow.document.write("<p>" + "<br><br>" + txt.fontsize(10) +
"</p>");
        openedWindow.focus();
    }
</script>
</body>
</html>
```

注意，这里的 HTML/JS 代码，按钮的点击事件函数中会有一个 window.open（“about.blank”）调用，在浏览器中其实就是要求打开一个新的浏览器窗口。相应地在这里就会触发一个 event，打开 ArkWeb 的新窗口。

Index.ets 代码：

```
import { webview } from '@kit.ArkWeb';

// 在同一 page 页有两个 Web 组件。在 WebComponent 新开窗口时，会跳转到
NewWebViewComp。
@CustomDialog
```

```
struct NewWebViewComp {
  controller?: CustomDialogController;
  webviewController1: webview.WebviewController = new webview.WebviewController();

  build() {
    Column() {
      Web({ src: "", controller: this.webviewController1 })
        .javaScriptAccess(true)
        .multiWindowAccess(false)
        .onWindowExit(() => {
          console.info("NewWebViewComp onWindowExit");
          if (this.controller) {
            this.controller.close();
          }
        })
    }
  }
}

@Entry
@Component
struct WebComponent {
```

```
  controller: webview.WebviewController = new webview.WebviewController();
  dialogController: CustomDialogController | null = null;

  build() {
    Column() {
      Web({ src: $rawfile("window.html"), controller: this.controller })
        .javaScriptAccess(true)
        // 需要使能 multiWindowAccess
        .multiWindowAccess(true)
        .allowWindowOpenMethod(true)
        .onWindowNew((event) => {
          if (this.dialogController) {
            this.dialogController.close()
          }
          let popController: webview.WebviewController = new
webview.WebviewController();
          this.dialogController = new CustomDialogController({
            builder: NewWebViewComp({ webviewController1: popController })
          })
          this.dialogController.open();
          // 将新窗口对应 WebviewController 返回给 Web 内核。
          // 如果不需要打开新窗口请调用 event.handler.setWebController 接口设置成 null。
```

```
            // 若不调用 event.handler.setWebController 接口，会造成 render 进程阻塞。
            event.handler.setWebController(popController);
        })
    }
  }
}
```

这里的代码实际上是创建了一个 CustomDialog 子组件，所谓的打开新的窗口就是打开这个对话框窗口。首先需要打开开关：

```
.javaScriptAccess(true)
.multiWindowAccess(true)
.allowWindowOpenMethod(true)
```

然后通过 onWindowNew 去相应 HTML 文件中的按钮点击事件 event，打开 Dialog 窗口，显示这两句话设定的内容：

```
var txt = '打开的窗口';
openedWindow.document.write("<p>" + "<br><br>" + txt.fontsize(10) + "</p>");
```

效果：

- 管理位置权限

Web 组件提供位置权限管理能力。开发者可以通过 onGeolocationShow()接口对某个网站进行位置权限管理。Web 组件根据接口响应结果，决定是否赋予前端页面权限。

使用获取设备位置功能前请在 module.json5 中添加位置相关权限：

在 string.json 里面添加一个 reason：


```
    {
        "name": "permissionReason",
        "value": "whatever"
    }
```

在 module.json5 中的 requestPermissions 中添加：

```
        {
            "name" : "ohos.permission.LOCATION",
            "reason": "$string:permissionReason",
            "usedScene": {
                "when": "always"
            }
        },
        {
            "name" : "ohos.permission.APPROXIMATELY_LOCATION",
            "reason": "$string:permissionReason",
            "usedScene": {
                "when": "always"
            }
```

```
        },
        {
            "name" : "ohos.permission.LOCATION_IN_BACKGROUND",
            "reason": "$string:permissionReason",
            "usedScene": {
                "when": "always"
            }
        }
```

在 rawfile 中创建一个 getLocation.html:

```
<!DOCTYPE html>
<html>
<body>
<p id="locationInfo" style="font-size: 36px;">位置信息</p>
<button onclick="getLocation()" style="font-size: 36px;">获取位置</button>
<script>
var locationInfo=document.getElementById("locationInfo");
function getLocation(){
  if (navigator.geolocation) {
    <!-- 前端页面访问设备地理位置 -->
    navigator.geolocation.getCurrentPosition(showPosition);
  }
}
```

```
function showPosition(position){
  locationInfo.innerHTML="Latitude: " + position.coords.latitude + "<br />Longitude: " +
position.coords.longitude;
}
</script>
</body>
</html>
```

JS 的代码表示，点击按钮之后，在位置信息那里会显示经纬度信息。

Index.ets 代码：

```
import { webview } from '@kit.ArkWeb';
import { BusinessError } from '@kit.BasicServicesKit';
import { abilityAccessCtrl, common } from '@kit.AbilityKit';

let context = getContext(this) as common.UIAbilityContext;
let atManager = abilityAccessCtrl.createAtManager();

// 向用户请求位置权限设置。
atManager.requestPermissionsFromUser(context,
["ohos.permission.APPROXIMATELY_LOCATION"]).then((data) => {
  console.info('data:' + JSON.stringify(data));
  console.info('data permissions:' + data.permissions);
  console.info('data authResults:' + data.authResults);
```

```
}).catch((error: BusinessError) => {
  console.error(`Failed to request permissions from user. Code is ${error.code}, message is
${error.message}`);
})

@Entry
@Component
struct WebComponent {
  controller: webview.WebviewController = new webview.WebviewController();

  build() {
    Column() {
      Web({ src: $rawfile('getLocation.html'), controller: this.controller })
        .geolocationAccess(true)
        .onGeolocationShow((event) => { // 地理位置权限申请通知
          AlertDialog.show({
            title: '位置权限请求',
            message: '是否允许获取位置信息',
            primaryButton: {
              value: 'cancel',
              action: () => {
                if (event) {
```

```
                    event.geolocation.invoke(event.origin, false, false); // 不允许此站点地理位置权限请求
                }
            }
        },
        secondaryButton: {
            value: 'ok',
            action: () => {
                if (event) {
                    event.geolocation.invoke(event.origin, true, false); // 允许此站点地理位置权限请求
                }
            }
        },
        cancel: () => {
            if (event) {
                event.geolocation.invoke(event.origin, false, false); // 不允许此站点地理位置权限请求
            }
        }
    })
})
```

```
    }
  }
}
```

这段代码首先会调用 atManager.requestPermissionsFromUser 询问用户是否允许应用使用位置权限，

效果：

因为是这个 ohos.permission.APPROXIMATELY_LOCATION，所以是大致位置，原因也用到了我们设置的“whatever”，点击“仅使用期间允许”，之后显示页面：

简单的一个文本加一个按钮。点击“获取位置”按钮：

点击 OK 会触发这段代码：

```
event.geolocation.invoke(event.origin, true, false); // 允许此站点地理位置权限请求
```

此时在位置信息那里，应该是得到位置的信息。

但是模拟器上没有显示，可能有问题。

- 使用隐私模式

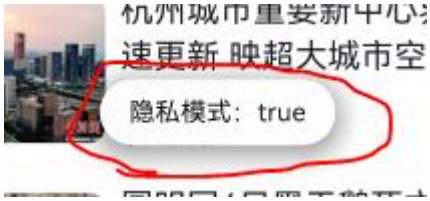
开发者在创建 Web 组件时，可以将可选参数 incognitoMode 设置为 true，来**开启** Web 组件的隐私模式。 当使用隐私模式时，浏览网页时的 Cookie、 Cache Data 等数据**不会保存在本地的持久化文件**，当隐私模式的 Web 组件被销毁时，Cookie、 Cache Data 等数据将不被记录下来。

Index.ets 代码：

```
import { webview } from '@kit.ArkWeb';
import { BusinessError } from '@kit.BasicServicesKit';
import { promptAction } from '@kit.ArkUI';

@Entry
@Component
struct WebComponent {
  controller: webview.WebviewController = new webview.WebviewController();
  @State isPrivate: boolean = true;

  build() {
    Column({space: 10}) {
      Button('isIncognitoMode')
        .onClick(() => {
```

```
            try {
                let result = this.controller.isIncognitoMode();
                promptAction.showToast({
                  message: `隐私模式：${result}`,
                  duration: 2000
                });
            } catch (error) {
                console.error(`ErrorCode: ${(error as BusinessError).code},  Message: ${(error as BusinessError).message}`);
            }
        })
        Web({ src: 'www.baidu.com', controller: this.controller, incognitoMode: this.isPrivate })
    }
  }
}
```

在 Web 组件中，加上 incognitoMode: true 就表示是隐私模式，设为 false 则为非隐私模式。函数 isIncognitoMode 可以得到是否为隐私模式。点击按钮后效果：



隐私模式提供了一系列接口，用于操作地理位置、Cookie 以及 Cache Data。

以 Cookie 为例，Index.ets 改为：

```
import { webview } from '@kit.ArkWeb';
import { BusinessError } from '@kit.BasicServicesKit';

@Entry
@Component
struct WebComponent {
  controller: webview.WebviewController = new webview.WebviewController();

  build() {
    Column({space: 10}) {
      Button('configCookieSync')
        .onClick(() => {
          try {
            // configCookieSync 第三个参数表示获取隐私模式（true）或非隐私模式（false）下，对应 url 的 cookies。
            webview.WebCookieManager.configCookieSync('https://www.baidu.com', 'a=b', true);
          } catch (error) {
            console.error(`ErrorCode: ${(error as BusinessError).code},  Message: ${(error as BusinessError).message}`);
          }
```

```
        })

      Button('existCookie')
        .onClick(() => {
          // existCookie 参数表示隐私模式（true）或非隐私模式（false）下，查询是否存在cookies。
          let result = webview.WebCookieManager.existCookie(true);
          console.log("WebDemoDebug exist Cookie result: " + result);
        })

      Button('fetchCookieSync')
        .onClick(() => {
          try {
            // fetchCookieSync 第二个参数表示获取隐私模式（true）或非隐私模式（false）下，webview 的内存 cookies。
            let value = webview.WebCookieManager.fetchCookieSync('https://www.baidu.com', true);
            console.log("WebDemoDebug: fetchCookieSync cookie = " + value);
          } catch (error) {
            console.error(`ErrorCode: ${(error as BusinessError).code},  Message: ${(error as BusinessError).message}`);
          }
```

```
        })

        Button('clearAllCookiesSync')
          .onClick(() => {
            // clearAllCookiesSync 参数表示清除隐私模式（true）或非隐私模式（false）下，webview 的所有内存 cookies。
            webview.WebCookieManager.clearAllCookiesSync(true);
          })

        Web({ src: 'www.baidu.com', controller: this.controller, incognitoMode: true })
      }
    }
}
```

效果：

四个按钮，分别调用 configCookieSync，existCookie，fetchCookieSync，clearAllCookiesSync，这些函数调用的时候都需要去设置是否是隐私模式。

- 使用运动和方向传感器

Web 组件可以通过 W3C 标准协议接口对接**运动和方向相关的传感器**。开发者在使用该功能中的**加速度、陀螺仪及设备运动事件**接口时，需在配置文件中声明相应的**传感器权限**。

https://developer.huawei.com/consumer/cn/doc/harmonyos-guides-V5/web-sensor-V5

这里的例子仅仅给出了获取加速度权限和陀螺仪权限的代码，大家可以自行尝试。

4. Web 渲染和布局

- Web 组件渲染模式

Web 组件支持两种渲染模式。

- **异步渲染**模式（默认）

  renderMode: RenderMode.ASYNC_RENDER

  异步渲染模式下，Web 组件作为图形 surface 节点，独立送显。**建议在仅由 Web 组件构成的应用页面中使用此模式**，有更好的性能和更低的功耗表现。

- **同步渲染**模式

  renderMode: RenderMode.SYNC_RENDER

  同步渲染模式下，Web 组件作为图形 canvas 节点，Web 渲染跟随系统组件一起送显。可以渲染更长 Web 组件内容，但会消耗更多的性能资源。

规格与约束

异步渲染模式

- Web 组件的宽高最大规格不超过 7,680px（物理像素），超过则会导致白屏。
- 不支持动态切换模式。

同步渲染模式

- Web 组件的宽高最大规格不超过 500,000px（物理像素），超过则会导致白屏。
- 不支持 DSS 合成。
- 不支持动态切换模式。

- Web 组件大小自适应页面内容布局

使用 Web 组件大小自适应页面内容布局模式 **layoutMode(WebLayoutMode.FIT_CONTENT)**时，能使 Web 组件的大小根据页面内容自适应变化。

适用于 Web 组件需要根据网页高度撑开，与其他原生组件一起滚动的场景，如：

- 浏览长文章。Web 组件同一布局层级有其他原生组件，如评论区、工具栏等。
- 长页面首页。Web 组件同一布局层级有其他原生组件，如宫格菜单。

规格与约束

- 建议配置渲染模式为**同步渲染**模式，避免因为组件大小超出限制导致异常场景（白屏，布局错误）。
- 建议配置**过滚动模式为关闭**状态。当过滚动模式开启时，当用户在 Web 界面上滑动到边缘时，Web 会通过弹性动画弹回界面，会与 Scroll 组件的回弹相互冲突，影响体验。
- 键盘避让属性配置为 RESIZE_CONTENT 时，该避让模式不生效。
- **不支持对页面进行缩放**。
- **不支持通过 Web 组件的 height 属性修改组件高度**。
- **仅支持**根据页面内容自适应组件**高度，不支持自适应宽度**。

在 rawfile 中创建一个 longtext.html 文件：

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, user-scalable=no">
    <title>Fit-Content</title>
</head>
<body>
<div>
    <div><h2 id="使用场景">使用场景</h2>
```

ArkWeb（方舟 Web）提供了 Web 组件，用于在应用程序中显示 Web 页面内容。常见使用场景包括：

 应用集成 Web 页面：应用可以在页面中使用 Web 组件，嵌入 Web 页面内容，以降低开发成本，提升开发、运营效率。
 浏览器网页浏览场景：浏览器类应用可以使用 Web 组件，打开三方网页，使用无痕模式浏览 Web 页面，设置广告拦截等。
 小程序：小程序类宿主应用可以使用 Web 组件，渲染小程序的页面。

<h2 id="能力范围">能力范围</h2>
Web 组件为开发者提供了丰富的控制 Web 页面能力。包括：

 Web 页面加载：声明式加载 Web 页面和离屏加载 Web 页面等。
 生命周期管理：组件生命周期状态变化，通知 Web 页面的加载状态变化等。

 常用属性与事件：UserAgent 管理、Cookie 与存储管理、字体与深色模式管理、权限管理等。

与应用界面交互：自定义文本选择菜单、上下文菜单、文件上传界面等与应用界面交互能力。

 App 通过 JavaScriptProxy，与 Web 页面进行 JavaScript 交互。

 安全与隐私：无痕浏览模式、广告拦截、坚盾守护模式等。

 维测能力：Devtools 工具调试能力，使用 crashpad 收集 Web 组件崩溃信息。

其他高阶能力：与原生组件同层渲染、Web 组件的网络托管、Web 组件的媒体播放托管、Web 组件输入框拉起自定义输入法、等。

<h2 id="约束与限制">约束与限制</h2>

Web 内核版本：ArkWeb 基于谷歌 Chromium 内核开发，使用的 Chromium 版本为 M114。

```
            </ul>
        </div>
    </div>
    </body>
    </html>
```

Index.ets 代码：

```
import { webview } from '@kit.ArkWeb';

@Entry
@Component
struct WebHeightPage {
  private webviewController: WebviewController = new webview.WebviewController()
  private scroller: Scroller = new Scroller()

  build() {
    Navigation() {
      Column() {
        Scroll(this.scroller) {
          Column() {
            Web({
              src: $rawfile("longtext.html"),
              controller: this.webviewController,
```

```
                renderMode: RenderMode.SYNC_RENDER // 设置为同步渲染模式
            })
                .layoutMode(WebLayoutMode.FIT_CONTENT) // 设置为 Web 组件大小自适应
页面内容
                .overScrollMode(OverScrollMode.NEVER) // 设置过滚动模式为关闭状态
            Text("评论区")
                .fontSize(28)
                .fontColor("#FF0F0F")
                .height(100)
                .width("100%")
                .backgroundColor("#f89f0f")
        }
    }


  }
  .title("标题栏")
 }
}
```

效果：

5. 应用侧与前端页面相互调用及数据通道

- 应用侧调用前端页面函数

应用侧可以通过 runJavaScript()和 runJavaScriptExt()方法调用前端页面的 JavaScript 相关函数。

runJavaScript()和 runJavaScriptExt()在参数类型上有些差异。runJavaScriptExt()入参类型不仅支持 string 还支持 ArrayBuffer（从文件中获取 JavaScript 脚本数据），另外可以通过 AsyncCallback 的方式获取执行结果。

在 rawfile 中创建一个 appcallweb.html:

```
<!DOCTYPE html>
<html>
<body>
<button type="button" onclick="callArkTS()" style="font-size: 36px;">Click Me!</button>
<h1 id="text">这是一个测试信息，默认字体为黑色，调用 runJavaScript 方法后字体为绿色，调用 runJavaScriptCodePassed 方法后字体为红色</h1>
<script>
```

```
    // 调用有参函数时实现。
    var param = "param: JavaScript Hello World!";
    function htmlTest(param) {
        document.getElementById('text').style.color = 'green';
        console.log(param);
    }
    // 调用无参函数时实现。
    function htmlTest() {
        document.getElementById('text').style.color = 'green';
    }
    // Click Me！触发前端页面 callArkTS()函数执行 JavaScript 传递的代码。
    function callArkTS() {
        changeColor();
    }
</script>
</body>
</html>
```

注意，这里的 JS 代码中有三个函数:

- htmlTest(param)带参数的，虽然这个参数没什么用处。里面是设置了文本的颜色属性为绿色

- htmlTest()不带参数，同样，是设置了文本的颜色属性为绿色

- callArkTs()里面是调用应用侧传递过来的 changeColor 函数。按钮点击的时候这个函数被调用。

Index.ets 代码：

```
import { webview } from '@kit.ArkWeb';
```

```
@Entry
@Component
struct WebComponent {
  webviewController: webview.WebviewController = new webview.WebviewController();

  aboutToAppear() {
    // 配置 Web 开启调试模式
    webview.WebviewController.setWebDebuggingAccess(true);
  }

  build() {
    Column() {
      Button('runJavaScript')
        .onClick(() => {
          // 前端页面函数无参时，将 param 删除。
          this.webviewController.runJavaScript('htmlTest(param)');
        })
      Button('runJavaScriptCodePassed')
        .onClick(() => {
          // 传递 runJavaScript 侧代码方法。
```

```
            this.webviewController.runJavaScript(`function
changeColor(){document.getElementById('text').style.color = 'red'}`);
        })
        Web({ src: $rawfile('appcallweb.html'), controller: this.webviewController })
    }
  }
}
```

这里有两个按钮，一个按钮是运行 runJavaScript('htmlTest(param)')，参数就是 HTML 中的 JS 函数，带参数的那个。另外一个同样是调用 runJavaScript，但是往里面放了一个字符串：

function changeColor(){document.getElementById('text').style.color = 'red'}

其实这个就是相当于往 JS 中添加了一个函数 changeColor()的定义。

运行效果：

点击 runJavaScript 按钮：

点击 runJavaScriptCodePassed 按钮，然后再点击文本上方的小按钮：

达成了期望的效果。

- 前端页面调用应用侧函数

开发者使用 Web 组件**将应用侧代码注册到前端页面**中，注册完成之后，**前端页面中使用注册的对象名称就可以调用应用侧的函数**，实现在前端页面中调用应用侧方法。

注册应用侧代码**有两种方式**，一种在 **Web 组件初始化调用**，使用 **javaScriptProxy**()接口。另外一种**在 Web 组件初始化完成后调用**，使用 **registerJavaScriptProxy**()接口。

在 rawfile 中创建一个 webcallapp.html：

```
<!DOCTYPE html>
<html>
<body>
<button type="button" onclick="callArkTS()" style="font-size: 50px">Click Me!</button>
<p id="demo" style="font-size: 50px"></p>
<script>
    function callArkTS() {
        let str = testObjName.test();
```

```
            document.getElementById("demo").innerHTML = str;
            console.info('ArkTS Hello World! :' + str);
        }
</script>
</body>
</html>
```

在这里我们还不知道这个 testObjName.test()做了什么，返回什么。

先尝试一下 javaScriptProxy()接口，Index.ets 改为：

```
import { webview } from '@kit.ArkWeb';

class testClass {
  constructor() {
  }

  test(): string {
      return 'ArkTS Hello World!';
  }
}

@Entry
@Component
```

```
struct WebComponent {
  webviewController: webview.WebviewController = new webview.WebviewController();
  // 声明需要注册的对象
  @State testObj: testClass = new testClass();

  build() {
    Column() {
      // Web 组件加载本地 index.html 页面
      Web({ src: $rawfile('webcallapp.html'), controller: this.webviewController})
        // 将对象注入到 web 端
        .javaScriptProxy({
          object: this.testObj,
          name: "testObjName",
          methodList: ["test"],
          controller: this.webviewController,
          // 可选参数
          asyncMethodList: [],
          permission:
'{"javascriptProxyPermission":{"urlPermissionList":[{"scheme":"resource","host":"rawfile","po
rt":"","path":""},' +
```

```
'{"scheme":"e","host":"f","port":"g","path":"h"}],"methodList":[{"methodName":"test","urlPe
rmissionList":' +
'[{"scheme":"https","host":"xxx.com","port":"","path":""},{"scheme":"resource","host":"rawfil
e","port":"","path":""}]},' +
'{"methodName":"test11","urlPermissionList":[{"scheme":"q","host":"r","port":"","path":"t"},'
+
'{"scheme":"u","host":"v","port":"","path":""}]}]}}'
})
}
}
}
```

可以看到，核心的点在：

```
.javaScriptProxy({
object: this.testObj,
name: "testObjName",
methodList: ["test"],
```

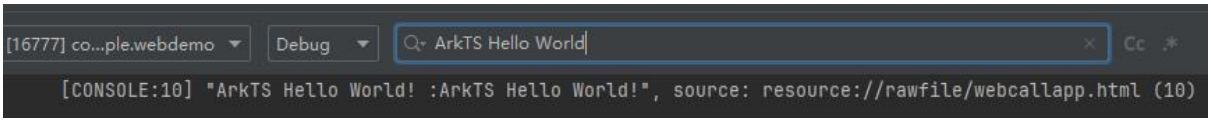
我们把名称为 testObjName，实例为 this.testObj，方法中包括了 test 的参数传给了 javaScriptProxy，这样在 HTML 的 JS 代码中：

```
let str = testObjName.test();
```

就知道该怎样执行了。

点击 Click Me 按钮：

查看 Log：



要注意这里的 Permission，展开 JSON：

```
{
  "javascriptProxyPermission": {
    "urlPermissionList": [          // Object 级权限，如果匹配，所有 Method 都授权
      {
        "scheme": "resource",      // 精确匹配，不能为空
        "host": "rawfile",         // 精确匹配，不能为空
        "port": "",                // 精确匹配，为空不检查
        "path": ""                 // 前缀匹配，为空不检查
      },
      {
        "scheme": "https",         // 精确匹配，不能为空
```

```
        "host": "xxx.com",          // 精确匹配，不能为空
        "port": "8080",             // 精确匹配，为空不检查
        "path": "a/b/c"             // 前缀匹配，为空不检查
      }
    ],
    "methodList": [
      {
        "methodName": "test",
        "urlPermissionList": [     // Method 级权限
          {
            "scheme": "https",     // 精确匹配，不能为空
            "host": "xxx.com",     // 精确匹配，不能为空
            "port": "",            // 精确匹配，为空不检查
            "path": ""             // 前缀匹配，为空不检查
          },
          {
            "scheme": "resource",// 精确匹配，不能为空
            "host": "rawfile",     // 精确匹配，不能为空
            "port": "",            // 精确匹配，为空不检查
            "path": ""             // 前缀匹配，为空不检查
          }
        ]
```

```
        },
        {
            "methodName": "test11",
            "urlPermissionList": [     // Method 级权限
                {
                    "scheme": "q",          // 精确匹配，不能为空
                    "host": "r",            // 精确匹配，不能为空
                    "port": "",             // 精确匹配，为空不检查
                    "path": "t"             // 前缀匹配，为空不检查
                },
                {
                    "scheme": "u",          // 精确匹配，不能为空
                    "host": "v",            // 精确匹配，不能为空
                    "port": "",             // 精确匹配，为空不检查
                    "path": ""              // 前缀匹配，为空不检查
                }
            ]
        }
    ]
  }
}
```

我们这里能够执行，是因为在 urlPermissionList 中，有

```
        {
```

```
            "scheme": "resource",       // 精确匹配，不能为空
            "host": "rawfile",          // 精确匹配，不能为空
            "port": "",                 // 精确匹配，为空不检查
            "path": ""                  // 前缀匹配，为空不检查
        },
```

而我们的 webcallapp.html 文件就在这里。

再尝试一下应用侧使用 registerJavaScriptProxy()接口注册。Index.ets 改为：

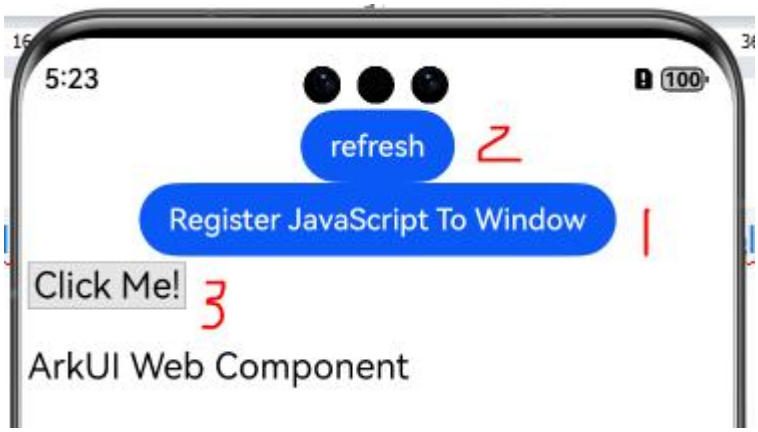
```
import { webview } from '@kit.ArkWeb';
import { BusinessError } from '@kit.BasicServicesKit';

class testClass {
  constructor() {
  }

  test(): string {
    return "ArkUI Web Component";
  }

  toString(): void {
    console.log('Web Component toString');
  }
}
```

```
@Entry
@Component
struct Index {
  webviewController: webview.WebviewController = new webview.WebviewController();
  @State testObj: testClass = new testClass();

  build() {
    Column() {
      Button('refresh')
        .onClick(() => {
          try {
            this.webviewController.refresh();
          } catch (error) {
            console.error(`ErrorCode: ${(error as BusinessError).code},  Message: ${(error as BusinessError).message}`);
          }
        })
      Button('Register JavaScript To Window')
        .onClick(() => {
          try {
            this.webviewController.registerJavaScriptProxy(this.testObj,  "testObjName", ["test", "toString"],
```

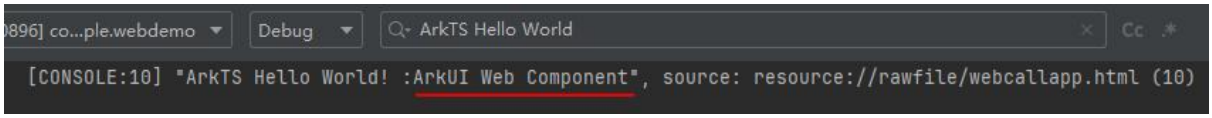
```
                    // 可选参数, asyncMethodList
                    [],
                    // 可选参数, permission
                    '{"javascriptProxyPermission":{"urlPermissionList":[{"scheme":"resource","host":"rawfile","port":"","path":""},' +
                    '{"scheme":"e","host":"f","port":"g","path":"h"}],"methodList":[{"methodName":"test","urlPermissionList":' +
                    '[{"scheme":"https","host":"xxx.com","port":"","path":""},{"scheme":"resource","host":"rawfile","port":"","path":""}]},' +
                    '{"methodName":"test11","urlPermissionList":[{"scheme":"q","host":"r","port":"","path":"t"},' +
                        '{"scheme":"u","host":"v","port":"","path":""}]}]}}'
                );
            } catch (error) {
                console.error(`ErrorCode: ${(error as BusinessError).code},  Message: ${(error as BusinessError).message}`);
            }
        })
```

```
      Web({ src: $rawfile('webcallapp.html'), controller: this.webviewController })
    }
  }
}
```

效果：（先点击 Register 按钮，再 refresh，再 Click Me）



查看 Log：



- 建立应用侧与前端页面数据通道

前端页面和应用侧之间可以用 createWebMessagePorts()接口创建消息端口来实现两端的通信。

在下面的示例中，应用侧页面中通过 createWebMessagePorts 方法创建消息端口，再把其中一个端口通过 postMessage()接口发送到前端页面，便可以在前端页面和应用侧之间互相发送消息。

在 rawfile 中创建一个 communication.html：

```
<!DOCTYPE html>
<html>
```

```
<head>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>WebView Message Port Demo</title>
</head>
<body>
<h1>WebView Message Port Demo</h1>
<div>
    <input type="button" value="SendToEts"
onclick="PostMsgToEts(msgFromJS.value);"/><br/>
    <input id="msgFromJS" type="text" value="send this message from HTML to
ets"/><br/>
</div>
<p class="output">display received message send from ets</p>
</body>
<script>
    var h5Port;
    var output = document.querySelector('.output');
    window.addEventListener('message', function (event) {
        if (event.data === '__init_port__') {
            if (event.ports[0] !== null) {
                h5Port = event.ports[0]; // 1. 保存从应用侧发送过来的端口。
                h5Port.onmessage = function (event) {
```

```
                    // 2. 接收 ets 侧发送过来的消息。
                    var msg = 'Got message from ets:';
                    var result = event.data;
                    if (typeof(result) === 'string') {
                        console.info(`received string message from html5, string is: ${result}`);
                        msg = msg + result;
                    } else if (typeof(result) === 'object') {
                        if (result instanceof ArrayBuffer) {
                            console.info(`received arraybuffer from html5, length is:
${result.byteLength}`);
                            msg = msg + 'length is ' + result.byteLength;
                        } else {
                            console.info('not support');
                        }
                    } else {
                        console.info('not support');
                    }
                    output.innerHTML = msg;
                }
            }
        }
    })
```

```
    // 3. 使用 h5Port 向应用侧发送消息。
    function PostMsgToEts(data) {
        if (h5Port) {
            h5Port.postMessage(data);
        } else {
            console.error('h5Port is null, Please initialize first');
        }
    }
</script>
</html>
```

Index.ets 代码：

```
import { webview } from '@kit.ArkWeb';
import { BusinessError } from '@kit.BasicServicesKit';

@Entry
@Component
struct WebComponent {
  controller: webview.WebviewController = new webview.WebviewController();
  ports: webview.WebMessagePort[] = [];
  @State sendFromEts: string = 'Send this message from ets to HTML';
  @State receivedFromHtml: string = 'Display received message send from HTML';
```

```
  build() {
    Column() {
      // 展示接收到的来自 HTML 的内容
      Text(this.receivedFromHtml)
      // 输入框的内容发送到 HTML
      TextInput({ placeholder: 'Send this message from ets to HTML' })
        .onChange((value: string) => {
          this.sendFromEts = value;
        })

      // 该内容可以放在 onPageEnd 生命周期中调用。
      Button('postMessage')
        .onClick(() => {
          try {
            // 1、创建两个消息端口。
            this.ports = this.controller.createWebMessagePorts();
            // 2、在应用侧的消息端口(如端口 1)上注册回调事件。
            this.ports[1].onMessageEvent((result: webview.WebMessage) => {
              let msg = 'Got msg from HTML:';
              if (typeof (result) === 'string') {
                console.info(`received string message from html5, string is: ${result}`);
                msg = msg + result;
```

```
                } else if (typeof (result) === 'object') {
                    if (result instanceof ArrayBuffer) {
                        console.info(`received arraybuffer from html5, length is: ${result.byteLength}`);
                        msg = msg + 'length is ' + result.byteLength;
                    } else {
                        console.info('not support');
                    }
                } else {
                    console.info('not support');
                }
                this.receivedFromHtml = msg;
            })
            // 3、将另一个消息端口(如端口 0)发送到 HTML 侧，由 HTML 侧保存并使用。
            this.controller.postMessage('__init_port__', [this.ports[0]], '*');
        } catch (error) {
            console.error(`ErrorCode: ${(error as BusinessError).code}, Message: ${(error as BusinessError).message}`);
        }
    })

    // 4、使用应用侧的端口给另一个已经发送到 html 的端口发送消息。
```

```
        Button('SendDataToHTML')
          .onClick(() => {
            try {
              if (this.ports && this.ports[1]) {
                this.ports[1].postMessageEvent(this.sendFromEts);
              } else {
                console.error(`ports is null, Please initialize first`);
              }
            } catch (error) {
              console.error(`ErrorCode: ${(error as BusinessError).code},  Message: ${(error as BusinessError).message}`);
            }
          })
        Web({ src: $rawfile('communication.html'), controller: this.controller })
      }
    }
}
```

核心的代码，第一步创建通信端口：

```
this.ports = this.controller.createWebMessagePorts();
```

创建了两个端口，ports[0]给 HTML 侧使用：

```
// 3、将另一个消息端口(如端口 0)发送到 HTML 侧，由 HTML 侧保存并使用。
this.controller.postMessage('__init_port__', [this.ports[0]], '*');
```

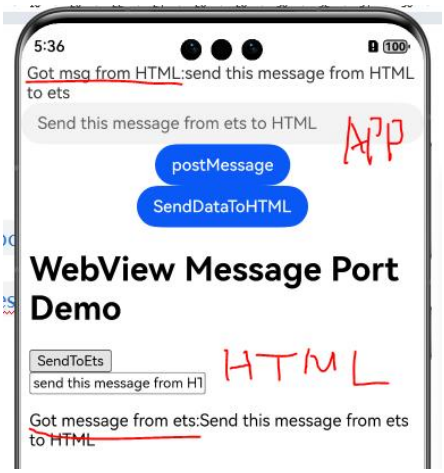
而 ports[1]留给应用侧自己使用：

```
// 2、在应用侧的消息端口(如端口 1)上注册回调事件。
this.ports[1].onMessageEvent((result: webview.WebMessage) => {
```

以及：

```
        // 4、使用应用侧的端口给另一个已经发送到 html 的端口发送消息。
        Button('SendDataToHTML')
          .onClick(() => {
            try {
              if (this.ports && this.ports[1]) {
                this.ports[1].postMessageEvent(this.sendFromEts);
```

效果：



APP 侧，收到 Web 侧发过来的消息，也可以发送消息到 Web 侧，Web 侧能够成功接收。

6. 管理网页加载与浏览记录

- 使用 Web 组件加载页面

页面加载是 Web 组件的基本功能。根据页面加载数据来源可以分为三种常用场景，包括**加载网络页面、加载本地页面、加载 HTML 格式的富文本数据**。页面加载过程中，若涉及网络资源获取，请在 module.json5 中配置网络访问权限（我们前面已经加上了）。

开发者可以在 Web 组件**创建时，指定默认加载的网络页面** 。在默认页面加载完成后，如果开发者

需要变更此 Web 组件显示的网络页面，可以通过**调用 loadUrl()接口**加载指定的网页。Web 组件的**第一个参数变量 src 不能通过状态变量（例如：@State）动态更改地址**，如需更改，请通过 loadUrl()重新加载。

Index.ets 代码：

```
import { webview } from '@kit.ArkWeb';
import { BusinessError } from '@kit.BasicServicesKit';

@Entry
@Component
struct WebComponent {
  controller: webview.WebviewController = new webview.WebviewController();

  build() {
    Column() {
      Button('loadUrl')
        .onClick(() => {
          try {
            // 点击按钮时，通过 loadUrl，跳转到 www.example1.com
            this.controller.loadUrl('www.sina.com.cn');
          } catch (error) {
            console.error(`ErrorCode: ${(error as BusinessError).code},  Message: ${(error as BusinessError).message}`);
```

```
        }
      })
      // 组件创建时，加载 www.example.com
      Web({ src: 'www.baidu.com', controller: this.controller })
    }
  }
}
```

代码的目的是，刚开始 Web 组件加载 www.baidu.com，然后点击按钮之后，加载 www.sina.com.cn。

效果：

达到预期目的。

对于本地页面，将本地页面文件放在应用的 rawfile 目录下，开发者可以在 Web 组件创建的时候指定默认加载的本地页面 ，并且加载完成后可通过调用 loadUrl()接口变更当前 Web 组件的页面。

加载本地 html 文件时引用本地 css 样式文件可以通过下面方法实现。

```
<link rel="stylesheet" href="resource://rawfile/xxx.css">
<link  rel="stylesheet"  href="file:///data/storage/el2/base/haps/entry/cache/xxx.css">//
加载沙箱路径下的本地 css 文件。
```

加载 HTML 格式的文本数据

Web 组件可以通过**loadData()接口**实现**加载 HTML 格式的文本数据**。当开发者不需要加载整个页面，只需要显示一些页面片段时，可通过此功能来快速加载页面，当加载大量 html 文件时，需设置第四个参数 baseUrl 为"data"。

Index.ets 代码：

```
import { webview } from '@kit.ArkWeb';
import { BusinessError } from '@kit.BasicServicesKit';

@Entry
@Component
struct WebComponent {
  controller: webview.WebviewController = new webview.WebviewController();

  build() {
    Column() {
      Button('loadData')
        .onClick(() => {
          try {
            // 点击按钮时，通过 loadData，加载 HTML 格式的文本数据
            this.controller.loadData(
              "<html><body
bgcolor=\"white\">Source:<pre>source</pre></body></html>",
```

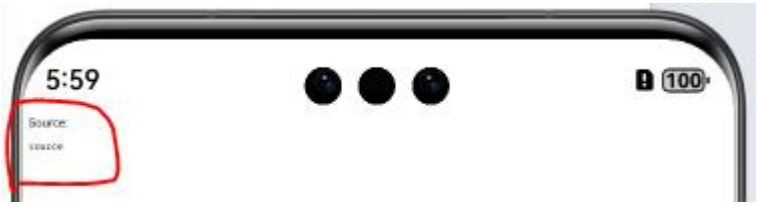
```
                "text/html",
                "UTF-8"
            );
        } catch (error) {
            console.error(`ErrorCode: ${(error as BusinessError).code},  Message: ${(error as BusinessError).message}`);
        }
      })
      // 组件创建时，加载 www.baidu.com
      Web({ src: 'www.baidu.com', controller: this.controller })
    }
  }
}
```

效果：

Web 组件可以通过 data url 方式直接加载 HTML 字符串，Index.ets 代码：

```
import { webview } from '@kit.ArkWeb';


@Entry
@Component
struct WebComponent {
  controller: webview.WebviewController = new webview.WebviewController();
  htmlStr: string = "data:text/html, <html><body bgcolor=\"white\">Source:<pre>source</pre></body></html>";

  build() {
    Column() {
      // 组件创建时，加载 htmlStr
      Web({ src: this.htmlStr, controller: this.controller })
    }
  }
}
```

效果：



- 管理页面跳转及浏览记录导航

历史记录导航

在前端页面点击网页中的链接时，Web 组件默认会自动打开并加载目标网址。当前端页面替换为新

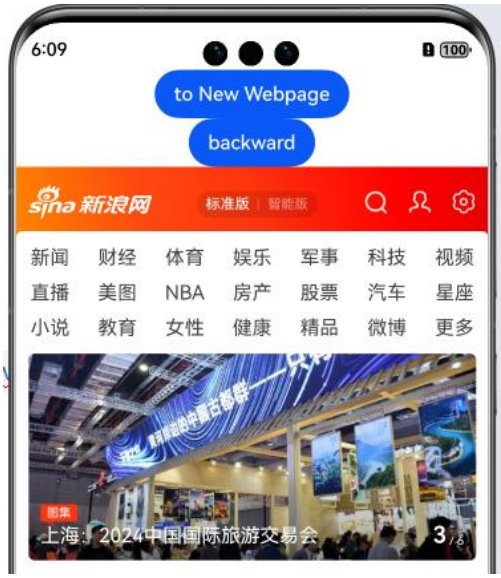
的加载链接时，会**自动记录已经访问的网页地址**。可以通过 **forward()和 backward()接口向前/向后**浏览上一个/下一个历史记录。

Index.ets 代码：

```
import { webview } from '@kit.ArkWeb';
import { BusinessError } from '@kit.BasicServicesKit';

@Entry
@Component
struct WebComponent {
  webviewController: webview.WebviewController = new webview.WebviewController();

  build() {
    Column() {
      Button('to New Webpage')
        .onClick(() => {
          try {
            // 点击按钮时，通过 loadUrl，跳转到 www.example1.com
            this.webviewController.loadUrl('www.sina.com.cn');
          } catch (error) {
            console.error(`ErrorCode: ${(error as BusinessError).code},  Message: ${(error as BusinessError).message}`);
          }
```

```
        })
        Button('backward')
          .onClick(() => {
            if (this.webviewController.accessBackward()) {
              this.webviewController.backward();
            }
          })
        Web({ src: 'https://www.baidu.com/', controller: this.webviewController })
      }
    }
}
```

这里首先加载百度网站，然后点击按钮 to New Webpage 去到新浪，此时调用 accessBackward()会返回 true，然后调用 backward()可以返回到百度，效果：



页面跳转

当点击网页中的链接需要跳转到应用内其他页面时，可以通过使用 Web 组件的 **onLoadIntercept()接口**来实现。

在 rawfile 中创建一个 route.html:

```
<!DOCTYPE html>
<html>
<body>
  <div>
      <a href="native://pages/ProfilePage" style="font-size: 60px">个人中心</a>
  </div>
</body>
</html>
```

在 pages 目录下创建一个 ProfilePage（注意要用新建 Page 的方式）：

```
@Entry
@Component
struct ProfilePage {
  @State message: string = 'My Personal Profile';

  build() {
    Column() {
      Text(this.message)
        .fontSize(20)
    }
    .width('100%')
```

```
    .height('100%')
    .justifyContent(FlexAlign.Center)
  }
}
```

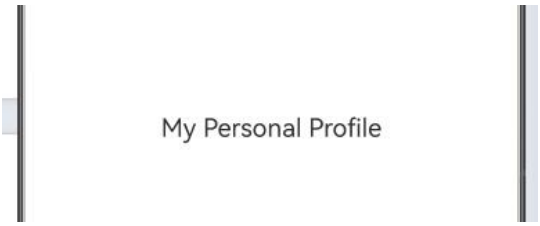
Index.ets 代码：

```
import { webview } from '@kit.ArkWeb';
import { router } from '@kit.ArkUI';

@Entry
@Component
struct WebComponent {
  webviewController: webview.WebviewController = new webview.WebviewController();

  build() {
    Column() {
      // 资源文件 route.html 存放路径 src/main/resources/rawfile
      Web({ src: $rawfile('route.html'), controller: this.webviewController })
        .onLoadIntercept((event) => {
          if (event) {
            let url: string = event.data.getRequestUrl();
            if (url.indexOf('native://') === 0) {
              // 跳转其他界面
```

```
                router.pushUrl({ url: url.substring(9) });
                return true;
            }
        }
        return false;
    })
  }
}
```

效果：

点击之后：



跨应用跳转

Web 组件可以实现点击前端页面超链接跳转到其他应用，在下面的示例中，点击 call.html 前端页面中的超链接，跳转到电话应用的拨号界面。

在 rawfile 中创建一个 call.html:

```
<!DOCTYPE html>
<html>
```

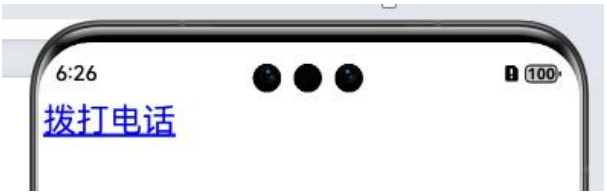
```
<body>
<div>
    <a href="tel://133 0224 3125" style="font-size: 60px">拨打电话</a>
</div>
</body>
</html>
```

Index.ets 改为：

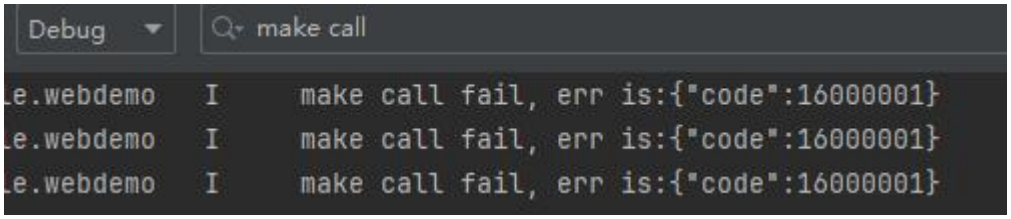
```
import { webview } from '@kit.ArkWeb';
import { call } from '@kit.TelephonyKit';

@Entry
@Component
struct WebComponent {
  webviewController: webview.WebviewController = new webview.WebviewController();

  build() {
    Column() {
      Web({ src: $rawfile('call.html'), controller: this.webviewController })
        .onLoadIntercept((event) => {
          if (event) {
            let url: string = event.data.getRequestUrl();
            // 判断链接是否为拨号链接
```

```
                if (url.indexOf('tel://') === 0) {
                    // 跳转拨号界面
                    call.makeCall(url.substring(6), (err) => {
                        if (!err) {
                            console.info('make call succeeded.');
                        } else {
                            console.info('make call fail, err is:' + JSON.stringify(err));
                        }
                    });
                    return true;
                }
            }
            return false;
        })
    }
  }
}
```

效果：



点击发现无法拉起电话拨号：

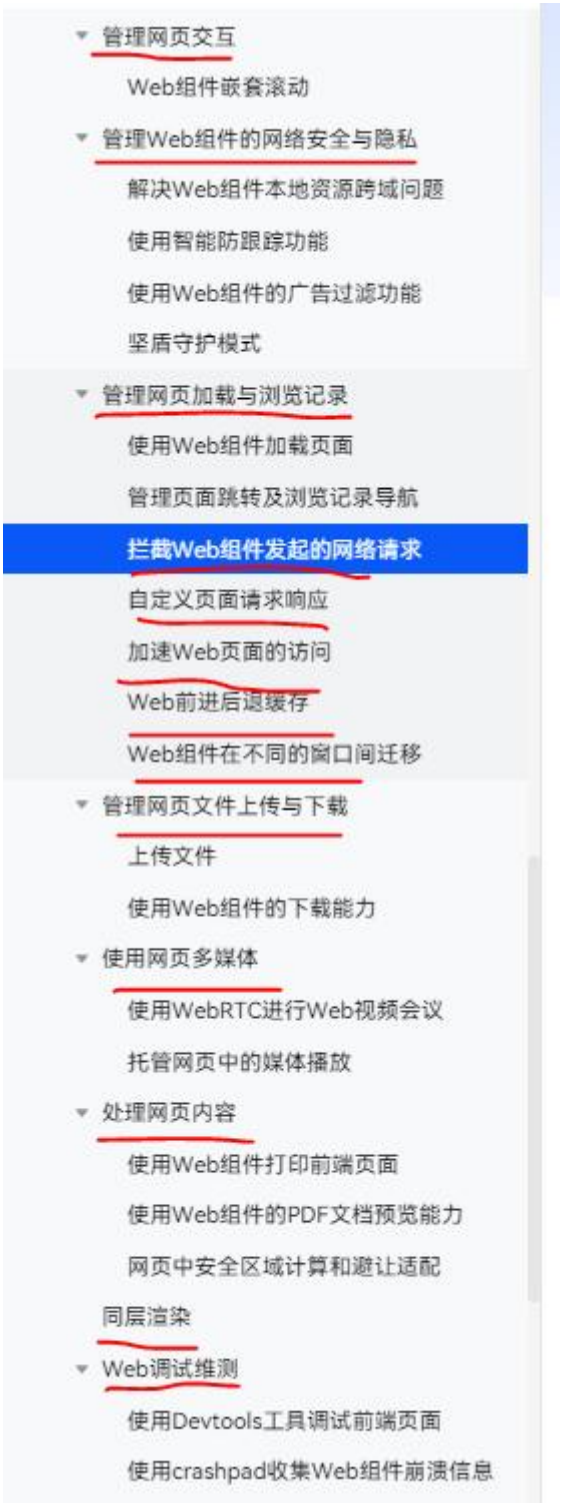


错误码是 16000001，有空再去查原因。至少可以看到尝试去拉起电话拨号了。

ArkWeb 还有很多的内容，需要大家自己去看文档：

文档入口：

https://developer.huawei.com/consumer/cn/doc/harmonyos-guides-V5/arkweb-V5



五、实验注意事项

1. 注意教师的操作演示。

2. 学生机与教师机内网连通，能接收和提交实验结果。

3.按实验要求输入测试数据，并查看输出结果是否符合实验结果。

六、思考题

1.通过这个实验，你学到了什么？