# Rax介绍&高阶用法

## 1. 课程目标

1. 学习Rax小程序的基本语法、API及组件的使用；
2. 掌握Rax小程序的高阶用法；

## 2. 课程大纲

- Rax基本使用；
- Rax小程序基本介绍；
- Rax小程序 API；
- Rax小程序组件；

## 3. Rax基本使用

> - Rax 语法层面以 React 为标准，可以使用 Hooks、Context 等 80% 以上支持度的 React API
> - 官方配套的研发框架 Rax App，支持 TypeScript、Less/Sass 等基础工程能力，同时支持 MPA、SPA、SSR
> - 支持通过完整的 Rax 语法开发跨支付宝/微信/字节等不同厂商的小程序，同时可降级到 Web
> - 基于 Web 标准支持跨多容器的跨端应用，包含 Web 应用、Flutter 应用（Kraken）、Weex 应用
> - 丰富的跨端生态，比如跨端组件 Fusion Mobile，跨端 API Uni API

- Rax 与 React 的区别是什么?
  - Rax 面向多端设计的，从最初始就引入了 Driver 机制来适配不同端，相比 React 更加轻量，gzip 之后只有 6KB；
- Rax 对于 React 的 API 支持度是怎样的?
  - 不支持 Suspense、lazy API，其他诸如 Hooks、Component 等 API 都支持；

### 3.1. 目录结构

JavaScript 复制代码

```
├── .rax/                          // 运行时生成的临时目录,git不要提交
├── build/                         // 构建产物目录，npm run build 后产物
├── public                         // 本地静态资源
│   └── favicon.png
├── src
│   ├── app.json                   // 路由及页面配置，routes、window等
│   ├── app.ts                     // [小程序|SPA]应用入口
│   ├── miniapp-native/            // [小程序]小程序原生代码
│   ├── components/                // 自定义业务组件
│   ├── pages/                     // 页面
│   ├── models/                    // 应用级数据状态
├── build.json                     // 工程配置
├── package.json
└── tsconfig.json
```

## 3.2. 环境配置

JavaScript 复制代码

```
// 获取不同环境的配置
// src/config.ts
export default {
  // 默认配置
  default: {
    appId: '123',
    baseURL: '/api'
  },
  local: {
    appId: '456',
  },
  daily: {
    appId: '789',
  },
  prod: {
    appId: '101',
  }
}

import { config } from 'rax-app';

console.log(config.appId);
```

## 3.3. 编写组件

支持

- Function Component
- Class Component

JavaScript 复制代码

```
// function component
import { createElement } from 'rax';

function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}

// class component
import { createElement, Component } from 'rax';

class Welcome extends Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

支持类型：

- props
- state
- Fragment
- Hooks：常用Hooks：useState、useEffect etc，具体看下文
- JSX
- JSX+
  - 条件判断：x-if、x-elseif、x-else
  - 循环列表：x-for
  - 单次渲染：x-memo：首次render后值不变不会render
  - 插槽：x-slot
  - 类名绑定：x-class

JavaScript | 复制代码

```
// 条件判断
<View x-if={condition}>Hello</View>
<View x-elseif={anotherCondition}></View>
<View x-else>NothingElse</View>

// x-for
{/* Array or Plain Object*/}
<tag x-for={item in foo}>{item}</tag>

<tag x-for={(item, key) in foo}>{key}: {item}</tag>

// x-memo
<p x-memo>this paragragh {mesasge} content will not change.</p>

// x-slot
<Waterfall>
  <view x-slot:header>header</view>
  <view x-slot:item="props">{props.index}: {props.item}</view>
  <view x-slot:footer>footer</view>
</Waterfall>

<slot name="header" /> // 槽位

// x-class
<div x-class={{ item: true, active: val }} />


```

## 3.4. 样式方案

- 全局样式

统一定义在 src/global.[css|less|scss]，框架会自动引入；

- 组件样式

文件名定义：xxx.module.[css|less|scss]，使用 CSS Modules，避免全局污染

JavaScript 复制代码

```
/** ./pages/Home/index.module.css */
.container {
  background: #fff;
  width: 750rpx;
}

/* 也可通过 CSS Modules 的 :global 语法定义全局样式 */
:global {
  body {
    a {
      color: blue;
    }
  }
}

// ./pages/Home/index.tsx
import styles from './index.module.css';

function Home() {
  return (
    <View className={styles.container}>
      <View>CSS Modules</View>
    </View>
  );
}

// 编译后
<View class="container--1DTudAN">title</View>
```

- 内联样式

在 build.json 里配置了 inlineStyle: true 则说明整个项目使用内联样式

JavaScript 复制代码

```
const myStyle = {
  fontSize: '24px',
  color: '#FF0000'
};

const element = <View style={myStyle}>Hello Rax</View>;
```

支持在使用内联样式方案的项目中局部支持非内联形式

在 build.json 中将 inlineStyle 设置为 { forceEnableCSS: true }

JavaScript | 复制代码

```
import cssModule from './index.module.css';
import styles from './index.css';

function App() {
  // CSS Modules 以 className 的形式使用
  // inlineStyle 以 style 的形式使用
  return <div className={cssModule.header} style={styles.header} />;
}
```

## 3.5. 框架API

通过 rax-app中引入

> import { runApp } from 'rax-app';

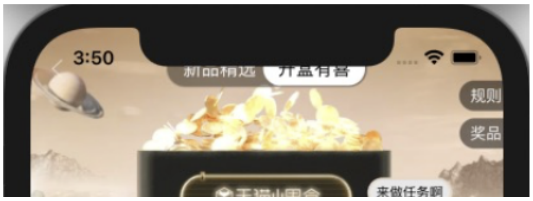
- runApp
- APP_MODE
- ErrorBoundary
- store
- getHistory：获取history实例
- getSearchParams：获取参数
- history：路由实例

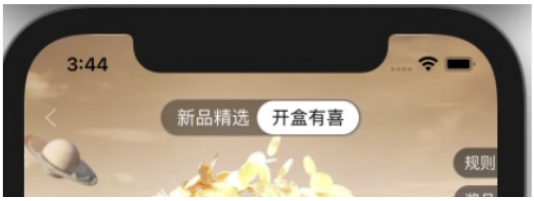
```javascript
import { history } from 'rax-app';

// 用于获取 history 栈里的实体个数
console.log(history.length);

// 用于获取 history 跳转的动作，包含 PUSH、REPLACE 和 POP 三种类型
console.log(history.action);

// 用于获取 location 对象，包含 pathname、search 和 hash
console.log(history.location);

// 用于路由跳转
history.push('/home');

// 用于路由替换
history.replace('/home');

// 用于跳转到上一个路由
history.goBack();
```

## 3.6. 安全区域适配

- 刘海屏适配

为了使页面顶部内容，不被刘海遮挡，可以通过设置容器节点的 padding-top 值来实现，使核心内容整体下移。




获取刘海高度，首先需要设置 viewport-fit ，调整可视窗口的布局方式。当且仅当 viewport-fit 设置为 cover 时，可以进一步设置页面的安全区域范围。

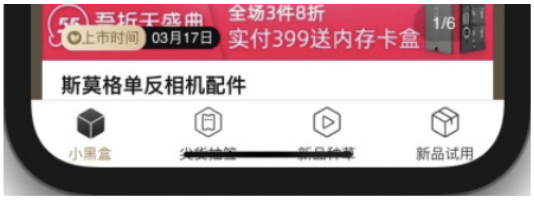
> <meta name="viewport" content="width=device-width, viewport-fit=cover">

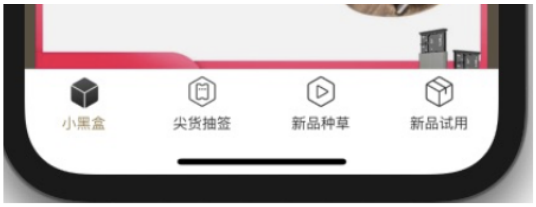
然后，结合 env() 方法，可以获取 safe-area-inset-top 值，并将其作为容器节点的 padding-top 值。

```JavaScript
.root {
  padding-top: constant(safe-area-inset-top); /* 兼容  iOS < 11.2 */
  padding-top: env(safe-area-inset-top); // /* iOS > 11.2 */
}
// 注意：  在 iOS 11.2 之前的版本，需使用 constant() 方法
```

- 底部小黑条适配

有 tabbar 的应用，iPhone 底部的小黑条常常会挡住 tabbar，影响其可操作区域。和刘海屏适配的原理一致，以 tabbar 为例，小黑条适配可以通过调整 tabbar 的 padding-bottom 值，增加空白区域来实现。




使用示例：

```JavaScript
.tabbar {
  padding-bottom: 0; /* 无小黑条的情况下，无需额外设置  */
  padding-bottom: constant(safe-area-inset-bottom); /* 兼容  iOS < 11.2 */
  padding-bottom: env(safe-area-inset-bottom); // /* iOS > 11.2 */
}
```

## 3.7. 静态资源使用

将文件放入 public 文件夹，webpack不会处理它。而是它将被复制到构建文件夹中；
要引用 public 文件夹中的资源，需要使用名为 process.env.PUBLIC_URL 的特殊变量，这个值会根据工程配中的 publicPath 变化：

```JavaScript
render() {
  // 注意：这是一个 escape hatch，应该谨慎使用！
  // 通常我们建议使用`import`来获取资源的 URL
  return <img src={process.env.PUBLIC_URL + '/img/logo.png'} />;
}
```

通常我们建议从 JS导入 stylesheets，图片和字体存入public文件夹中：

- 你需要在构建输出中具有特定名称的文件，例如 manifest.json；
- 你有数千张图片，需要动态引用它们的路径；
- 你希望在打包代码之外包含一个无需走构建逻辑的小脚本；
- 某些库可能与 webpack 不兼容，只能将其放在 public 中引入；

## 3.8. 代码切割

- dynamic import

使用 import()，webpack 会在编译阶段对引入的资源进行代码切割，即只有当运行时逻辑执行到 import() 调用点时才会加载对应的资源，该函数返回值是 Promise

```JavaScript
import { isWeb } from '@uni/env';

if (isWeb) {
  import('./fetch').then(fetch => {
    fetch('m.taobao.com');
  }).catch(err => {
    console.error('模块引入失败！');
  });
}
```

- rax-use-import

函数式组件提供的Hooks

JavaScript | 复制代码

```
import { createElement } from 'rax';
+import useImport from 'rax-use-import';

export default function App() {
+  const [Bar, error] = useImport(() => import(/* webpackChunkName: "bar" */ './Bar'));
  if (error) {
    return <p>error</p>;
  } else if (Bar) {
    return <Bar />
  } else {
    return <p>loading</p>;
  }
}

```

# 4. Rax小程序基本介绍

> Rax 小程序以运行时方案为基础，支持局部场景使用编译时方案开发组件，充分结合了二者的优势特点，让用户在保证开发效率的大前提下能够针对局部场景进行更高渲染性能的优化。

## 4.1. Rax小程序简介

### 4.1.1. 方案对比

- 编译时方案：
  - 通过 AST 转译 + 运行时垫片模拟 Rax core 的方式，将Rax DSL 1:1 输出为原生小程序代码；
  - 限制较多：
    - JSX 较为灵活，适配工作量巨大，维护成本较高，开发者需要遵循大量的语法约束，否则代码就不能正常编译运行，开发效率难以保证；
    - 需要配合runtime垫片来模拟 Rax 运行 --> Rax 有功能更新时，编译无法得到同步；
    - DOM 和 BOM API 的缺失，Web 上累积的各种前端生态无法复用；
  - 性能较好；
- 运行时方案：
  - 通过底层模拟 DOM 和 BOM API，使开发者可以使用Rax DSL开发；
  - 性能较差，但基本对齐web端生态；

## 4.2. Rax运行

> npm init rax rax-example

```
C:\Users\Admin\Desktop\PTJ\3. 课程培训\1. 前端课程\example\2. 小程序\rax
$ npm init rax rax-example-01
Current registry:  https://registry.npmmirror.com
npm WARN deprecated request-promise@4.2.6: request-promise has been deprecated because it extends the now deprecated req
uest package, see https://github.com/request/request/issues/3142
npm WARN deprecated har-validator@5.1.5: this library is no longer supported
npm WARN deprecated uuid@3.4.0: Please upgrade  to version 7 or higher.  Older versions may use Math.random() in certain
 circumstances, which is known to be problematic.  See https://v8.dev/blog/math-random for details.
npm WARN deprecated request@2.88.2: request has been deprecated, see https://github.com/request/request/issues/3142

changed 250 packages in 9s
? What's your project type? App (Build universal application)
? What's your application type?  小程序跨端应用
? What type of language do you want to use? JavaScript
Creating a new Rax project in C:\Users\Admin\Desktop\PTJ\3. 课程培训\1. 前端课程\example\2. 小程序\rax\rax-example-01
download tarballURL https://registry.npmmirror.com/@rax-materials/scaffolds-app-js/-/scaffolds-app-js-0.4.7.tgz
√ download npm tarball successfully.
clean package.json...
To run your app:
   cd rax-example-01
   npm install
   npm start
We have prepared develop toolkit for you.
See: https://marketplace.visualstudio.com/items?itemName=iceworks-team.iceworks
```

项目中 build.json 中的tagrets

JavaScript 复制代码

```json
{
  "targets": [
    "miniapp",              // 支付宝小程序
    "wechat-miniprogram",   // 微信小程序
    "bytedance-microapp",   // 字节跳动小程序
    "baidu-smartprogram",   // 百度智能小程序
    "kuaishou-miniprogram" // 快手小程序
  ]
}
```

package.json

- start：rax-app start
  - 保留日志；
  - 保留source map
- build：rax-app build
  - 不保留注释；
  - 去除source map；
  - 去除开发工具；

rax-app webpack config github地址：

https://github.com/raxjs/rax-app/tree/master/packages/rax-webpack-config/src

JavaScript | 复制代码

```
// index
export default (options: IOptions) => {
  const config = getBaseConfig(options);
  if (options.mode === 'development') {
    configDev(config);
  } else {
    configBuild(config);
  }
  return config;
};

// webpack.base.ts
export default (options: IOptions) => {
  const config = new Config(); // webpack-chain: 链式配置webpack

  config.mode(options.mode);

  // 尝试按顺序解析这些后缀名。如果有多个文件有相同的名字，但后缀名不同，
  // webpack 会解析列在数组首位的后缀的文件 并跳过其余的后缀
  config.resolve.extensions
    .merge(['.js', '.json', '.jsx', '.ts', '.tsx', '.html']); //
  // webpack loaders
  setWebpackLoaders(config, options);
  // webpack plugins
  setWebpackPlugins(config);

  return config;
};

// webpack.dev.ts
import * as TimeFixPlugin from 'time-fix-plugin';

export default (config) => {
  // custom stat output by stats.toJson() calls
  // 在使用 Node.js API 时，此选项无效；
  // 你需要将统计配置项传递给 stats.toString() 和 stats.toJson() 调用；
  // state: none 不输出bundle信息
  config.stats('none');

  // set source map,
https://webpack.js.org/configuration/devtool/#devtool
  // build速度: ok，rebuild速度： slow，且sourcemap为转移后的sourcemap
  config.devtool('cheap-module-source-map');

  // fix: https://github.com/webpack/watchpack/issues/25
```

```
    // 解决某些bug
    config.plugin('TimeFixPlugin').use(TimeFixPlugin);
  };


  // webpack.build.ts
  import * as TerserPlugin from '@builder/pack/deps/terser-webpack-plugin';
  import isWebpack4 from './isWebpack4';

  export default (config) => {
    // disable devtool of mode prod build
    config.devtool(false);

    // 压缩JS文件 terser
    let terserPluginOptions = {
      parallel: true, // 使用多进程并发运行以提高构建速度
      extractComments: false, // 是否将注释剥离到单独的文件中, boolean是否启动
      terserOptions: {
        output: {
          // 只有ascII码生效，转义字符串和正则表达式中的 Unicode 字符，
          // 非ascII码指令无效
          ascii_only: true,
          // 保留包含 "@license", "@copyright", "@preserve 等关键字的JSDoc的comments;
          comments: 'some',
          // 优化output的样式
          beautify: false,
        },
        mangle: true, // 压缩变量名
      },
    };

    let safeParser;
    let CssMinimizerPlugin;

    if (isWebpack4) {
      terserPluginOptions = {
        ...terserPluginOptions,
        // @ts-ignore
        cache: true, // 缓存地址：node_modules/.cache/terser-webpack-plugin
      };
      // Safe parser
      safeParser = require('@builder/rax-pack/deps/postcss-safe-parser');
      // css minimizer plugin
      CssMinimizerPlugin = require('@builder/rax-pack/deps/css-minimizer-webpack-plugin');
    } else {
```

```
90        // Safe parser
91        safeParser = require('@builder/pack/deps/postcss-safe-parser');
92        // css minimizer plugin
93        CssMinimizerPlugin = require('@builder/pack/deps/css-minimizer-
      webpack-plugin');
94      }
95
96      // uglify js file
97      config.optimization
98        .minimizer('TerserPlugin')
99        .use(TerserPlugin, [terserPluginOptions]);
100
101     // optimize css file
102     config.optimization
103       .minimizer('CssMinimizerPlugin')
104       .use(CssMinimizerPlugin, [{
105         parallel: false, // 使用多进程并发执行，提升构建速度
106         minimizerOptions: {
107           preset: [ // 移除所有注释
108             'default',
109             {
110               discardComments: { removeAll: true },
111             },
112           ],
113           processorOptions: {
114             parser: safeParser,  // 配置cssnano的processorOptions属性， 压缩
      postCSS
115           },
116         },
117       }]);
118   };
119
```

## 4.3. 入口文件

src/app.js（TS为app.tsx）为文件入口

JavaScript | 复制代码

```javascript
import { runApp } from 'rax-app';

const appConfig = {
  app: {
    // 可选，自定义添加 Provider
    addProvider: ({ children }) => {
      return <ConfigProvider>{children}</ConfigProvider>;
    },

    // 可选，开启默认的 ErrorBoundary 行为，默认值为 false
    errorBoundary: true,

    // 可选，自定义错误边界的 fallback UI
    ErrorBoundaryFallback: <div>渲染错误</div>,

    // 可选，自定义错误的处理事件
    onErrorBoundaryHandler: (error, componentStack) {
      // Do something with the error
    },

    // 可选，小程序应用生命周期
    onLaunch() {},
    onShow() {},
    onHide() {},
    onError() {},
    onShareAppMessage() {},
    ......
  },
  // 不建议在此创建store的getInitialStates
  store: {
    initialStates: {},
    getInitialStates: (initialData) => {}
  }
};

runApp(appConfig);
```

## 4.4. 应用配置

在src/app.json 中，对window，tabbar配置，且小程序中的路由存在于routes中

- path：指定页面对应的路由地址
- source： 指定页面组件地址，必须写成 pages/[PAGE_NAME]/index 格式，暂不支持嵌套式路

由；

- targets：指定页面需要构建的端，默认为 build.json 所配置的 targets 的值；
- window：指定该页面的窗体表现，可以覆盖全局的窗口设置；
- tabBar：如果应用是一个多 tab 应用（底部栏可以切换页面），可以指定 tab 栏及切换时显示的对应页面；

JavaScript | 复制代码

```
{
  "routes": [
    {
      "path": "/",
      "source": "pages/Home/index",
      "window": {
        "barButtonTheme": "default"
      }
    },
    {
      "path": "/about",
      "source": "pages/About/index",
      "window": {
        "barButtonTheme": "light"
      }
    }
  ],
  "window": {
    "title": "Rax App 1.0",
  },
   "tabBar": {
    "textColor": "#999",
    "selectedColor": "#666",
    "backgroundColor": "#f8f8f8",
    "items": [
      {
        "text": "home",
        "pageName": "/",
        "icon": "https://gw.alicdn.com/tfs/TB1ypSMTcfpK1RjSZFOXXa6nFXa-144-144.png",
        "activeIcon": "https://gw.alicdn.com/tfs/TB1NBiCTgHqK1RjSZFPXXcwapXa-144-144.png"
      }
    ]
  }
}
```

Tips：原生小程序中，app.json（debug、networkTimeout）的其他字段都可以在此配置

## 4.5. 工程配置

在build.json中，其中可以指定小程序的配置项

- buildType：小程序引擎，默认位运行时，编译时为 compile；
- nativeConfig：即微信小程序的project.config.json；
- subPackages：是否分包加载；
- runtimeDependencies：在运行时引擎下，使用rax工程的多包npm时，使用编译时还是运行时实现；
- nativePackage：配置原生小程序自定义组件及原生页面所用到的 npm 依赖

JavaScript | 复制代码

```
{
  "targets": ["miniapp", "wechat-miniprogram"],
  "miniapp": {}, // 支付宝小程序语法配置
  // 微信小程序语法配置
  "wechat-miniprogram": {
    "nativeConfig": {
      "appId": YOUR_APP_ID,
      "miniprogramRoot": "build/wechat-miniprogram"
    },
    "subPackages": {
      "shareMemory": true // 共享运行时内存
    },
  "nativePackage": {
    // 自动安装 Rax 项目中的原生小程序自定义组件及原生页面所用到的 npm 依赖
    // 不配置dependencies 字段且 autoInstall 未设置为 false时
    // 会默认安装所有依赖，可作为性能优化
    "autoInstall": true,
    "dependencies": {
      // 指定安装依赖，格式同package.json 中的 dependencies 字段一致
      "mini-ali-ui": "^1.3.4"
    }
  }
}
```

Tips：

- 当运行时项目中使用到由 Rax 组件工程产出的多端组件 npm 包时，不配置 runtimeDependencies， 默认采用其编译时的代码实现。此时需要将其配置到 nativePackage.dependencies，否则产物中将报找不到该组件的问题；

- nativePackage.dependencies 中配置的原生小程序 npm 依赖依然需要在 Rax 项目根目录的 package.json 中配置并安装；

## 4.6. 页面生命周期及事件处理

移除了部分原生小程序事件相关Hooks：

1. useHistory、useReachBottom等只能作为单纯API而不是Hooks使用；
2. 性能与可扩展性较弱，每当小程序新支持一个事件，都需要去开发一个对应功能的 API，并且在页面初始化之初就监听所有事件 --> 不合理的；

新增API：

| | |
|---|---|
| registerNativeEventListeners(Component, [...eventName]) | 注册该页面需要监听的所有事件，第一个参数为页面级组件 |
| addNativeEventListener(eventName, callback) | 开始监听某个事件并执行回调函数 |
| removeNativeEventListener(eventName, callback) | 移除某个事件的回调函数 |

JavaScript | 复制代码

```javascript
import { createElement, useEffect } from 'rax';
import View from 'rax-view';
import Text from 'rax-text';
import { isMiniApp } from 'universal-env';
import { registerNativeEventListeners, addNativeEventListener,
removeNativeEventListener } from 'rax-app';


function Index() {
  function handlePageReachBottom() {}
  useEffect(() => {
    if(isMiniApp) {
      // 开始监听 onReachBottom 事件
      addNativeEventListener('onReachBottom', handlePageReachBottom);
    }
    return () => {
      if (isMiniApp) {
      // 移除onReachBottom 事件的监听器
      removeNativeEventListener('onReachBottom', handlePageReachBottom);
    }
    }
  }, []);

  return (
    <View>
      <Text>1</Text>
    </View>
  );
}

if (isMiniApp) {
  registerNativeEventListeners(Index, ['onReachBottom']);
}

export default Index;
```

注意：

1. onShow事件建议：
    a. Function component：使用usePageShow(cb) ，cb在渲染后执行；
    b. Class component：使用 onShow()，会在constructor后调用；
    c. addNativeEventListener 监听需要在useEffect外调用，也要在useEffect cb清除监听

JavaScript 复制代码

```
import { createElement, useEffect } from 'rax';
import View from 'rax-view';
import Text from 'rax-text';
import { registerNativeEventListeners, addNativeEventListener,
removeNativeEventListener } from 'rax-app';


function Index() {
  function handlePageShow() {}
  addNativeEventListener('onShow', handlePageShow);
  useEffect(() => {
    return () => {
      removeNativeEventListener('onShow', handlePageShow);
    }
  });
  return (
    <View>
      <Text>1</Text>
    </View>
  );
}

registerNativeEventListeners(Index, ['onShow']);

export default Index;
```

## 4.7. 组件

在微信等小程序端通过 bind 前缀绑定事件，在 JSX 中需要处理为 on 前缀，并遵循驼峰式命名规则，如上面 bindgetphonenumber 处理为 onGetPhoneNumber；

使用小程序原生组件

1. npm安装
    a. 配置nativePackage 的 dependencies，不然会安装所有依赖；
    b. 开发者个人npm包
        i. 配置在 package.json 中的miniappConfig，main指向原生小程序入口

```
JavaScript | 复制代码

├── README.md
├── lib
│   └── miniapp
│       ├── index.acss
│       ├── index.axml
│       ├── index.js
│       └── index.json
├── package.json // miniappConfig 字段的值为 { "main": "lib/miniapp/index" }

// 引入方式：import Button from 'your-custom-component'
```

c. 第三方npm包

i. 不使用miniappConfig，使用 `import Title from 'mini-ali-ui/es/title/index'` 方式引入

2. 源码拷贝到本地

a. 拷贝到 `src/miniapp-native` 下，使用 `import Button from '../../miniapp-native/Button/index'` （而非 `import Button from '@src/miniapp-native'` ）

## 4.8. API

支持直接在对应环境下使用对应api

```
JavaScript | 复制代码

import { isMiniApp, isWeChatMiniProgram } from '@uni/env';

function scan() {
  if (isWeChatMiniProgram) {
    wx.scanCode();
  } else if (isMiniApp) {
    my.scan();
  }
}
```

## 4.9. 状态管理

全局状态管理：

- useReducer、useContext etc；--> 不建议

- 提供store

JavaScript | 复制代码

```
src
├── models                // 全局状态
|   ├── counter.ts
|   └── user.ts
├── app.tsx
└── store.ts

// src/models/user.ts
export const delay = (time) => new Promise((resolve) => setTimeout(() =>
resolve(), time));

export default {
  // 定义 model 的初始 state
  state: {
    name: '',
    id: ''
  },

  // 定义改变该模型状态的纯函数
  reducers: {
    update (prevState, payload) {
      return {
        ...prevState,
        ...payload,
      };
    },
  },

  // 定义处理该模型副作用的函数
  effects: (dispatch) => ({
    async updateUserInfo () {
      await delay(1000);
      dispatch.user.update({
        name: 'taobao',
        id: '123',
      });
    },
  }),
};

// src/store.ts
import { createStore } from 'rax-app';
import user from './models/user';

const store = createStore({ user });
```

```
45
46  export default store;
47
48  // 引用全局状态
49  import store from '@/store';
50
51  const HomePage = () => {
52    const [userState, userDispatchers] = store.useModel('user');
53
54    return (
55      <>
56        <span>{userState.id}</span>
57        <span>{userState.name}</span>
58      </>
59    );
60  }
```

## 4.10 使用编译时组件

1. 源码方式开发
    a. npm init 运行时，npm 安装 jsx2mp-runtime；
    b. src下创建miniapp-compiled，此目录下使用编译时编译；
    c. 在src/miniapp-compiled 下新建 index.jsx 文件，将所有编译时组件引入后进行导出；
    d. 引入时使用相对路径，不要用解构；
    e. 限制：
        i. 无法使用全局状态管理方案，其只拥有从父组件获取 props 执行渲染，并通过事件与父组件通信的能力；
        ii. 编译时组件在形态上等同于原生自定义组件，其依赖的 npm 包建议用户配置在 nativePackage 中，必须将 jsx2mp-runtime 加入 nativePackage.dependencies 中；

JavaScript | 复制代码

```
├── src
│   ├── miniapp-compiled
│   │   ├── CompiledComp1.jsx
│   │   ├── CompiledComp2.jsx
│   │   ├── CompiledComp3
│   │   │   └── index.jsx
│   │   └── index.jsx
│   └── pages
│       └── Home
│           ├── index.css
│           └── index.jsx
```

```jsx
// src/miniapp-compiled/index.jsx
import CompiledComp1 from './CompiledComp1';
import CompiledComp2 from './CompiledComp2';
import CompiledComp3 from './CompiledComp3';

export {
  CompiledComp1,
  CompiledComp2,
  CompiledComp3
}

// 引入时
import CompiledComp1 from '../miniapp-compiled/CompiledComp1';  // 正确
import { CompiledComp1 } from '../miniapp-compiled'; // 错误
```

2. npm方式开发
   a. Rax组件方式开发小程序默认采用编译时编译组件，可以直接在运行时项目中使用；
   b. 需要将该组件添加到 package.json 依赖中，还需要将其添加到 build.json 中 nativePackage.dependencies 字段内；
   c. 在微信小程序中使用：
      i. 需要在 Rax 组件的 package.json 中加入 "miniprogram": "." 兼容 npm 构建机制；
      ii. 在运行时项目中将 jsx2mp-runtime 添加到 package.json 和 build.json 里的 nativePackage.dependencies ；
      iii. 编译完成后，用户需要在微信 IDE 中进行构建 npm 的操作；
   d. 有兴趣同学可以参考Rax小程序组件开发：https://rax.js.org/docs/guide/miniapp-com-dev

## 4.11. 引用小程序原生页面

JavaScript 复制代码

```
// 目录
├── build.json
├── package.json
└── src
    ├── app.js
    ├── app.json
    ├── components
    │   └── Logo
    │       ├── index.css
    │       └── index.jsx
    ├── pages
    │   ├── About // 小程序原生
    │   │   ├── index.axml
    │   │   ├── index.css
    │   │   ├── index.js
    │   │   └── index.json
    │   └── Home
    │       ├── index.css
    │       └── index.jsx
    └── miniapp-native // 小程序原生
        └── List
            ├── index.axml
            ├── index.css
            ├── index.js
            └── index.json

// 小程序src/app.json
{
  "routes": [
    {
      "path": "/",
      "source": "pages/Home/index"
    },
    {
      "path": "/about",
      "source": "pages/About/index",
      "targets" ["miniapp"] // 声明
    }
  ],
  "window": {
    "title": "Rax App"
  }
}
```

Tips：

- 原生页面使用到的原生自定义组件（如上面项目中的 List 组件）必须放置于 src/miniapp-native 文件夹中，否则无法使用；
- 原生小程序页面建议放在 src/miniapp-native 文件夹中，而不是上面 src/pages/About 的做法；

## 4.12. 编写原生app.js

JavaScript 复制代码

```javascript
// 自 miniapp-render v2.7.0 开始，用户在 runApp 中编写的 app 的生命周期
// 将全部与小程序原生的 App 生命周期对齐

// src/app.ts
import { runApp } from 'rax-app';
runApp({
  // 可以在此处实现对应声明周期
  app: {
    onLaunch() {
      this.__age = 20; // this 即为 app 实例，可直接挂载变量
      console.log('on launch');
    },
    onShow() {
      console.log('on show');
    }
  }
})

// src/miniapp-native 新建app.js，必须module.export出app.js
// src/miniapp-native/app.js
console.log('我是自主编写的业务逻辑');
const originalPage = Page;
Page = function(options) {
  console.log('Page 已被劫持');
  originalPage(options);
}

const nativeAppConfig = {
  __age: 20,
  onLaunch() {
    console.log('on launch');
  },
  onShow() {
    console.log('on show');
  }
};
module.exports = nativeAppConfig;
```

## 4.13. 分包加载

JavaScript 复制代码

```
// 目录结构
src
  └── pages
  |   ├── Home               // 主包
  |   |   ├── Foo
  |   |   ├── Bar
  |   |   ├── models
  |   |   |   ├── Foo.ts
  |   |   |   └── Bar.ts
  |   |   ├── store.ts
  |   |   ├── app.json
  |   |   └── app.ts
  |   ├── About              // 子包
  |   |   ├── pages/index.ts
  |   |   └── app.ts
  |   |   └── app.json
  |----app.json
  |----app.js
```

1. 在build.json中配置

JavaScript 复制代码

```
{
  "targets": ["miniapp"],
  "miniapp": {
    "subPackages": true
  }
}
```

2. 分包入口设置

设置分包后，src/app.js 失效，src/app.json routes设置为分包入口

JavaScript 复制代码

```json
{
  "routes": [
    {
      "source": "pages/Home/app",
      "miniappMain": true // 主包入口
    },
    {
      "source": "pages/About/app"
    }
  ],
  "window": {
    "title": "Rax app"
  },
  "tabBar": {
    "textColor": "#999",
    "selectedColor": "#666",
    "backgroundColor": "#f8f8f8",
    "items": [
      {
        "text": "首页",
        "pagePath": "pages/Home/models/Foo", // 路径修改为具体路径
        "icon": "https://gw.alicdn.com/tfs/TB1vGsVqiDsXe8jSZR0XXXK6FXa-200-200.png",
        "activeIcon": "https://gw.alicdn.com/tfs/TB1EBamvz39YK4jSZPcXXXrUFXa-200-200.png"
      },
      {
        "text": "关于",
        "pagePath": "pages/Home/models/Bar",
        "icon": "https://gw.alicdn.com/tfs/TB1PceUq5pE_u4jSZKbXXbCUVXa-200-200.png",
        "activeIcon": "https://gw.alicdn.com/tfs/TB1BZTsqmslXu8jSZFuXXXg7FXa-200-200.png"
      },
    ]
  }
}
```

3. 分包入口代码

在分包下创建app.js作为入口，必须引入app.json并执行ruApp;

JavaScript 复制代码

```
import { runApp } from 'rax-app';
// 引入 app.json
import staticConfig from './app.json';

runApp({
  app: {
    onShow() {
      console.log('app show...');
    },
    onHide() {
      console.log('app hide...');
    },
  }
}, staticConfig);
```

4. 分包配置

JavaScript 复制代码

```
{
  "routes": [{
    "path": "/about",
    "source": "pages/index",
    "miniappPreloadRule": {
      "network": "wifi",
      "packages": ["pages/About"]
    }
  }]
}
```

5. 分包间共享内存

公共模块变为单例共享同一引用

```json
{
  "targets": ["miniapp"],
  "miniapp": {
    "subPackages": {
      "shareMemory": true
    }
  }
}
```

## 4.14. 原生小程序工程配置

在build.json 中的nativeConfig配置

```json
{
  "targets": ["miniapp", "wechat-miniprogram"],
  "wechat-miniprogram": {
    "nativeConfig": {
      "appId": YOUR_APP_ID, // 微信小程序必须要appId
      "miniprogramRoot": "build/wechat-miniprogram"
    },
    "subPackages": {
      "shareMemory": true
    },
    "runtimeDependencies": ["@ali/comp1", "/^raxcomp/"],
    "nativePackage": {
      "autoInstall": true,
      "dependencies": {
        "mini-ali-ui": "^1.3.4"
      }
    }
  }
}
```

## 4.1.5. 性能优化

1. 局部场景引入编译时组件（双引擎结合）
    a. 在局部性能要求较高的场景下（例如长列表），将部分组件（例如长列表中的循环项）采用编译时方案开发；

2. 代码逻辑优化
    a. 多使用 memo 等以减少不必要的子组件的重复渲染；
3. 高频组件分级
    a. 针对 view/image/text 等高频使用的组件，Rax 会在运行时根据其是否绑定 props、events 为其自动分级，以映射到对应的模板上，减少无用的 props、events 绑定，提升交互性能。因此，在编写代码时请注意只为该类组件绑定必需的 props 和 events；
4. 模板属性及事件配置
    a. 运行时方案中，我们会遍历所有内置组件，并将其输出至模板文件中，在小程序运行起来后根据实时的 setData 的数据递归迭代模板以生成完整的 DOM 结构。因此组件的 template 会预先绑定上所有的事件和属性，根据运行时的数据来进行属性传递和事件触发；
    b. 删除属性及事件：支持删除props和events

JavaScript | 复制代码

```
{
  "targets": [
    "wechat-miniprogram"
  ],
  "wechat-miniprogram": {
    "template": {
      "view": {
        "delete": {
          "props": ["hover-class", "role"],
          "events": ["TransitionEnd", "FirstAppear", "TouchMove"]
        }
      },
      "cover-view": {
        "delete": {
          "props": ["scroll-top"]
        }
      }
    }
  }
}
```

    c. 添加组件属性：只支持属性

```JavaScript
{
  "targets": [
    "miniapp"
  ],
  "miniapp": {
    "template": {
      "cover-view": {
        "add": {
          "props": [
            { "name": "test1", "default": "123" },
            { "name": "test2", "default": 123 }
            { "name": "test3", "default": false }
          ]
        }
      }
    }
  }
}
```

d. 删除无用template：针对不用的原生内置组件

```JavaScript
{
  "targets": [
    "miniapp"
  ],
  "miniapp": {
    "template": {
      "delete": ["canvas", "progress"]
    }
  }
}
```

# 5. Rax小程序API

## 5.1. 核心API

### 5.1.1. DOM

1. render

> 在container 里通过指定的 Driver, 渲染一个 Rax 元素，并返回该根组件的实例；
> 如果提供了可选的回调函数，该回调将在组件被渲染或更新之后被执行；

- element：任意需要渲染的 Rax 组件或字符串
- container：任意指定 DOM 渲染容器
- options：
  - driver : 指定 Driver，包含：DriverDom、DriverWeex、DriverUniversal
  - hydrate: 指定是否开启 hydrate 渲染模式，默认为 false
    - 最大程度的复用容器节点中已有的元素：SEO、SSR
- callback：传入回调函数，将在组件被渲染或更新之后被执行

JavaScript | 复制代码

```
render(element, container, options, [callback])

import { render } from 'rax';
import DriverDom from 'driver-dom';

const HelloMessage = function (props) {
  return <h1>{props.name}</h1>
};
render(<HelloMessage name="world" />, document.body, { driver: DriverDom
})
```

2. hydrate

> 最大程度的复用容器节点中已有的元素

- element：任意需要渲染的 Rax 组件或字符串
- container：为任意指定 DOM 渲染容器
- callback：传入回调函数，将在组件被渲染或更新之后被执行

JavaScript 复制代码

```
hydrate(element, container, [callback])

import hydrate from 'rax-hydrate';

// MyComponent.jsx
function MyComponent(props) {
  return <h1>Hello world</h1>;
}

hydrate(<MyComponent />, document.body);
```

3. createPortal

> 提供了一种将子元素渲染到存在于 DOM 组件层次结构之外的 DOM 节点中。

- child 是任何可渲染的 Rax 子元素，例如一个元素，字符串或 Fragment；
- container 是一个 DOM 元素；

JavaScript 复制代码

```
import createPortal from 'rax-create-portal';

const Portal = ({ children }) => {
  const el = useRef(document.createElement('div'));
  useEffect(() => {
    document.body.appendChild(el.current);
    return () => {
      el.current.parentElement.removeChild(el.current);
    };
  }, []);

// 无须再创建一个新的节点，它只是把 children 组件渲染到 `el.current` 中。
  return createPortal(children, el.current);
};

function App() {
  return <div>
    <Portal>
      <text>Hello Rax</text>
    </Portal>
  </div>
}
```

4. unmountComponentAtNode

> 卸载通过 render 函数渲染的组件

- container 为需要卸载的 DOM 元素；

JavaScript | 复制代码

```
unmountComponentAtNode(container)

import {createElement, render, useRef, useEffect } from 'rax';
import unmountComponentAtNode from 'rax-unmount-component-at-node';
import View from 'rax-view';
import Text from 'rax-text';

function App() {
  const ref = useRef(null);
  useEffect(() => {
    const result = unmountComponentAtNode(ref.current);
  });
  return <View>
  <Text ref={ref}>Hello Rax!</Text>
  </View>;
}
```

5. findDOMNode

> 通过ref获取真正的 DOM 元素，以便对 DOM 节点进行操作

- component 参数可以是 Rax 元素或者 DOM，均可返回真实 DOM 节点。

JavaScript 复制代码

```
findDOMNode(component)

import {createElement, render, useRef, useEffect } from 'rax';
import findDOMNode from 'rax-find-dom-node';
import View from 'rax-view';

function App() {
  const ref = useRef(null);
  useEffect(() => {
  const dom = findDOMNode(ref.current);
  });
  return <View ref={ref} ></View>;
}

```

6. setNativeProps

> 通过 setNativeProps 可以直接更改原生组件的属性来更新组件状态，避免多次render

JavaScript 复制代码

```
setNativeProps(node, props?)


import { createElement, Component, render, useRef } from 'rax';
import View from 'rax-view';
import Text from 'rax-text';
import setNativeProps from 'rax-set-native-props';

function App() {
  const textRef = useRef(null);

  function updateStyle() {
    setNativeProps(textRef.current, {
      style: {
        color: '#dddddd'
      }
    });
  }

  return <View>
  <Text ref={textRef} >setNativeProps</Text>
    <View onClick={updateStyle}>
      <Text >修改文字样式</Text>
    </View>
  </View>
}
```

7. getElementById

> 高效查找特定元素的方法

- id：为指定id

JavaScript 复制代码

```
getElementById(id)


import { createElement, Component, render } from 'rax';
import View from 'rax-view';
import Text from 'rax-text';
import getElementById from 'rax-get-element-by-id';

function App() {

  function focus() {
    getElementById('input').focus();
  }
  return <View>
      <Input id="input" />
      <Text onClick={focus}>input focus</Text>
    </View>
}
```

## 5.1.2. Element

1. createElement

> 用于创建并返回指定类型的 Rax 元素

- type：类型参数为标签名字符串或 Rax 元素；
- props：标签的属性；
- children 从第三个参数开始为元素的子节点，有多少个子节点就创建多少个；

JavaScript 复制代码

```
createElement(type, [props], [...children])

import { createElement } from 'rax';

createElement(
  'div',
  { id: 'foo' },
  createElement('p', null, 'hello world')
);
```

2. cloneElement

> 以 Rax 元素为模板克隆并返回新的 Rax 元素，将传入的 props 与原始元素的 props 浅层合并后返回新元素的 props。新的子元素将取代现有的子元素，而来自原始元素的 key 和 ref 将被保留。

- element：Rax 元素；
- props：标签的属性；
- children ： 从第三个参数开始为元素的子节点，有多少个子节点就创建多少个；

JavaScript 复制代码

```
cloneElement(element, [props], [...children])

import cloneElement from 'rax-clone-element';
import View from 'rax-view';
import Text from 'rax-text';

function Hello({ name }) {
  const Banner = <Text>Hello Rax!</Text>;
  return cloneElement(Banner);
}


function App() {
  return <View>
    <Hello></Hello>
  </View>
}
```

3. isValidElement

> 用于判断传入的对象是否为有效 Rax 元素，返回值为 true 或 false

- object：为 Rax 元素，校验通过返回 true，校验失败返回 false

JavaScript 复制代码

```
isValidElement(object)


import isValidElement from 'rax-is-valid-element';

const Hello = <h1>Hello </h1>;
const App = () => <div>{Hello}</div>;
console.log('Hello is valid? ', isValidElement(Hello));
// => true
console.log('App is valid? ',  isValidElement(App));
// => false
```

4. createFactory

> 通过工厂方法创建 Rax 组件实例，该方法就是对 createElement() 的封装；

- type：为类型参数；

JavaScript 复制代码

```
createFactory(type)

import { createElement } from 'rax';
import createFactory from 'rax-create-factory';

// createFactory(type);
const factory = createFactory('li');
const li1 = factory(null /** props */, 'Hello Rax!'/** children */);
const li2 = factory(null /** props */, 'Hello Rax!'/** children */);
createElement('ul', null, li1, li2);
```

5. Children

> Children 提供了用于处理 props.children 不透明数据结构的实用方法

JavaScript | 复制代码

```
Children.map(children, function[(thisArg)])

import Children from 'rax-children';

function Hello({ children }) {
  return Children.map(children, (child, i) => {
    if ( i < 1 ) return;
    return child;
  });
}

function App() {
  return <Hello>
    <Text>Hello</Text>
    <Text>Rax</Text>
  </Hello>
}

// 与 Children.map() 类似，但它不返回处理后的节点数组，仅遍历节点，多用于处理数据。
Children.forEach(children, function[(thisArg)])

function Hello({ children }) {
  Children.forEach(children, (child, i) => {
    if ( i < 1 ) {
      child.props.children = 'Hello' + child.props.children
    };
  });
  return children
}

function App() {
  return <Hello>
    <Text>World</Text>
    <Text>Rax</Text>
  </Hello>
}

// 返回 children 中的元素总数量，等同于通过 map 或 forEach 调用回调函数的次数。
Children.count(children)

function Hello({ children }) {
  const count = Children.count(children);
  return children
}

```

```jsx
function App() {
  return <Hello>
    <Text>World</Text>
    <Text>Rax</Text>
  </Hello>
}

// 验证 children 是否只有一个子节点（一个 React 元素），如果有则返回它，否则此方法会抛出错误
Children.only(children)

function Hello({ children }) {
  const only = Children.only(children);
  return only ? children : null;
}

function App() {
  return <Hello>
    <Text>World</Text>
  </Hello>
}

// 将 children 这个复杂的数据结构以数组的方式扁平展开并返回
Children.toArray(children)

function Hello({ children }) {
  const [state, setState] = useState('Rax');
  return Children.toArray(children).filter(child => child.props.children === state);
}

function App() {
  return <Hello>
    <Text>Rax</Text>
    <Text>World</Text>
  </Hello>
}
```

Rax 的核心是组件。你可以像嵌套 html 标签那样嵌套 Rax 组件，因为它类似于标记，使得编写 jsx 变得很容易。因为我们使用的是 javascript，我们可以改变 props.children。我们可以给他们传递特殊的属性，来决定是否渲染他们并且可以按照我们的意愿去操作他们。 Children 提供了用于处理 props.children 不透明数据结构的实用方法。从本质上来讲，props.children 可以是任何的类型，比如数组、函数、对象等等。

### 5.1.3. Component

1. memo

> 在函数组件， 如果你的函数组件在给定相同 props 的情况下渲染相同的结果

JavaScript | 复制代码

```
import { memo, useState } from 'rax';
import View from 'rax-view';
import Text from 'rax-text';

const useUpdate = () => {
  const [, setState] = useState(0);
  return () => setState(num => num + 1);
};

const HelloMemo = memo((props) => {
  console.log('memo-render');
  return <Text>{props.children}</Text>
});

const HelloNormal = (props) => {
  console.log('normal-render');
  return <Text>{props.children}</Text>
}

function App() {
  console.log('render')
  const update = useUpdate();
  const [state, setState] = useState(1);
  return <View>
    <HelloNormal>Rax</HelloNormal> // 不会rerender
    <HelloMemo>Hello</HelloMemo>   // 会rerender
    <View>{state}</View>
    <View onClick={() => setState(num => num + 1)}>Update</View>
  </View>
}
```

## 5.1.4. Hooks

支持的Hooks包括：

- useState
- useEffect
- useLayoutEffect
- useContext

- useRef
- useCallback
- useMemo
- useReducer

### 5.1.5. Refs

1. createRef

> 创建一个能够通过 ref 属性附加到 Rax 元素的 ref。当你需要访问节点时，可以通过 ref.current 得到

JavaScript | 复制代码

```
import { createRef, useEffect } from 'rax';

function App() {
  const inputRef = createRef();
  useEffect(() => {
    inputRef.current.focus();
  }, [inputRef.current]);

  return <input type="text" ref={inputRef} />;
}
```

2. forwardRef

> Ref转发，会创建一个 Rax 组件，这个组件能够将其接受的 ref 属性转发到其组件树下的另一个组件中。

JavaScript 复制代码

```
import { forwardRef } from 'rax';
import View from 'rax-view';
import Text from 'rax-text';

const MyInput = forwardRef((props, ref) => (
  <input type="text" ref={ref}></input>
));

function App() {
  const ref = createRef();

  const focus = () => {
    ref.current.focus();
  };

  return <View>
    <MyInput ref={ref}></MyInput>
    <View onClick={focus}>Click</View>
  </View>
}
```

### 5.1.6. Fragment

> 用于减少不必要嵌套的组件

JavaScript 复制代码

```
<Fragment>
  <header>A heading</header>
  <footer>A footer</footer>
</Fragment>

// JSX 也提供了短语法，小程序中暂未支持
<>
  <header>A heading</header>
  <footer>A footer</footer>
</>
```

### 5.1.7. Context

> 创建一个 Context 对象。当 Rax 渲染一个订阅了这个 Context 对象的组件，这个组件会从组件树中离自身最近的那个匹配的 Provider 中读取到当前的 context 值。

不建议，如果有需要可以使用strore

JavaScript 复制代码

```
import { createContext } from 'rax';
const MyContext = createContext(defaultValue);
```

### 5.1.8. PropTypes

> 要在组件的 props 上进行类型检查，你只需配置特定的 propTypes 属性 --> 建议使用TS

JavaScript 复制代码

```
import PropTypes from 'prop-types';

MyComponent.propTypes = {
  // 你可以将属性声明为 JS 原生类型，默认情况下
  // 这些属性都是可选的。
  optionalArray: PropTypes.array,
  optionalBool: PropTypes.bool,
  optionalFunc: PropTypes.func,
  optionalNumber: PropTypes.number,
  optionalObject: PropTypes.object,
  optionalString: PropTypes.string,
  optionalSymbol: PropTypes.symbol,

  // 任何可被渲染的元素（包括数字、字符串、元素或数组）
  // （或 Fragment）也包含这些类型。
  optionalNode: PropTypes.node
};
```

### 5.1.9. version

> 获取当前Rax core版本

JavaScript | 复制代码

```
import { version } from 'rax';
console.log('version: ', version);
// ==> version: 1.0.4
```

### 5.1.10. 其他扩展Hooks

| 名称 | 描述 |
| --- | --- |
| useMountedState | 获取当前mounted状态 |
| useMounted | 组件mounted回调 |
| useUnmount | 组件unmounted回调 |
| useInterval | setInterval基础上，在组件mount前clearInterval避免错误发送 |
| useOnceEffect | effect执行一次 |
| useTimeout | 似useInterval，在组件mount前clearTimeout避免错误发送 |
| useAsyncEffect | 用于异步的effect |
| useOnceAsyncEffect | 异步effect执行一次 |
| usePromise | 适用于promise场景 |
| useFetch | 适用于fetch请求场景 |
| useImport | 动态引入组件 |
| useCountDown | 返回倒计时时分秒等信息 |

建议有兴趣同学参考：https://rax.js.org/docs/api/useMountedState

## 5.2. Uni API

基于Rax 跨端开发的能力，推出了 Uni API，抹平了Web、Weex、多端小程序的差异；

### 5.2.1. 基础

| 名称 | 描述 |
| --- | --- |
| env | 判断和获取运行时环境（如isWeb、isWeex、isMiniApp etc） |
| canIUse | 判断 API 是否可用 |
| request | 用于发起网络请求 注意：此 API 不支持 promise 调用 |
| unitTool | 工具库，px2rpx, rpx2px |

### 5.2.2. 应用

| 名称 | 描述 |
| --- | --- |
| getApp | 获取全局唯一应用实例 |
| getCurrentPages | 获取当前页面栈。数组中第一个元素为首页，最后一个元素为当前页面 |
| getLaunchOptionsSync | 获取小程序启动时的参数 |
| unitTool | 工具库，px2rpx, rpx2px |
| onError | 错误事件的监听 |
| onUnhandledRejection | 错误事件的监听的promise拒绝事件 |
| offError | 取消错误事件的监听 |
| onUnhandledRejection | 取消错误事件的监听的promise拒绝事件 |

### 5.2.3. 设备

| 名称 | 描述 |
| --- | --- |
| scan | 获取全局调用扫一扫功能的 API |
| getClipboard | 获取当前页面栈获取系统剪贴板的内容 |
| setClipboard | 设置系统剪贴板的内容 |
| systemInfo | 获取系统信息 |
| makePhoneCall | 拨打电话 |

### 5.2.4. 文件

| 名称 | 描述 |
| --- | --- |
| download | 下载资源到本地 |
| getInfo | 下载资源信息 |
| getSavedInfo | 获取保存的文件 |
| openDocument | 在新页面打开预览 |
| removedSaved | 移除保存的文件 |
| upload | 上传文件到服务器 |

### 5.2.5. 界面

| 名称 | 描述 |
| --- | --- |
| alert | 警告框 |
| confirm | 模态对话框 |
| getScrollOffset | 获取元素移动位置信息 |
| getBoundingClientRect | 获取元素getBoundingClientRect |
| getMenuButtonBoundingClientRect | 解获取菜单按钮（右上角胶囊按钮）的布局位置信息 |
| showLoading/hideLoading | 显示/关闭 loading 提示框 |
| showToast/hideToast | 显示/关闭 Toast 提示框 |
| showTabBar/hideTabBar | 显示/关闭 tabBar |
| actionSheet | 显示操作菜单 |
| intersectionObserver | 用于推断某些节点是否可以被用户看见、有多大比例可以被用户看见 |
| onPullDownRefresh | 开启下拉刷新 |
| startPullDownRefresh/stopPullDownRefresh | 开始/关闭下拉刷新 |
| createTransition | 创建一个过渡动画 |
| createAnimation | 创建一个过渡动画实例 |
| setNavigationBarColor | 设置页面导航条颜色 |
| setNavigationBarTitle | 设置页面导航条文案 |
| pageScrollTo | 滚动到制定位置 |

### 5.2.6. 多媒体

| 名称 | 描述 |
| --- | --- |
| chooseImage | 本地选择相册或拍照 |
| chooseMedia | 拍摄或从手机相册中选择图片或视频 |
| chooseVideo | 拍摄视频或从手机相册中选视频 |
| compressImage | 压缩图片 |
| createAudioContext/createVideoContext | 创建音视频 |
| getImageInfo | 获取照片信息 |
| previewImage | 全屏预览图片 |
| saveImage | 保存图片到本地相册 |
| RecorderManager | 获取录音管理器 |

### 5.2.7. 路由

| 名称 | 描述 |
| --- | --- |
| navigate | go、replace、back、push、relaunch、switchTab |

### 5.2.8. 缓存

| 名称 | 描述 |
| --- | --- |
| getStorage | 从本地缓存中异步获取指定 key 的内容 |
| getStorageSync | 从本地缓存中同步获取指定 key 的内容 |
| removeStorage | 从本地缓存中异步移除指定 key |
| removeStorageSync | 从本地缓存中同步移除指定 key |
| setStorage | 将数据异步存储在本地缓存中指定的 key 中。会覆盖掉原来该 key 对应的内容 |
| setStorageSync | 将数据同步存储在本地缓存中指定的 key 中。会覆盖掉原来该 key 对应的内容 |

建议有兴趣同学参考：https://rax.js.org/docs/api/about

# 6. Rax小程序组件

## 6.1. 基础元件

即通用的universal组件，支持Web、各种小程序；

- 属性

| 属性名 | 类型 | 描述 |
| --- | --- | --- |
| style | object | 为元素设置内联样式 |
| className | string | 类选择器，允许以一种独立于文档元素的方式来指定样式 |

- 事件

| 事件名 | 类型 | 描述 |
| --- | --- | --- |
| onClick | function | 当组件被点击时触发的事件 |

支持的基础元件包括：

| 基础元件 | 描述 |
| --- | --- |
| Text | 用于显示文本，在 web 中实际上是一个 span 标签而非 p 标签 |
| View | 最基础的组件，它支持 Flexbox、touch handling 等功能，并且可以任意嵌套，像 web 中的 div 。 支持任意自定义属性的透传 |
| TextInput | TextInput 是唤起用户输入的基础组件；<br>当定义 multiline 输入多行文字时其功能相当于 textarea； |
| Link | • Link 是基础的链接组件，同 a 标签。它带有默认样式 textDecoration: 'none'。 在浏览器中，同我们熟悉的 a 标签，使用 Link 标签并不能新开一个 webview ，它只是在当前的 webview 中做页面的跳转<br>• 小程序场景使用 navigator 标签 |
| Icon | 图标组件<br>• source.uri：图片型icon的url或iconfont的url<br>• fontFamily：iconfont的字体<br>• source.codePoint：iconfont的码点 |
| Image | 展示图片 |
| Video | 视频播放组件 |
| ScrollView | 包装了滚动操作的组件。需要一个确定的高度来保证 ScrollView 的正常展现 |
| Waterfall | 实现瀑布流容器 |
| Portal | 提供了“传送”能力，可以将任意 RaxNode 渲染至根节点（body），见example |
| Embed | 内嵌内容容器，在 weex 下为 <web> 实现，在 web 下为 <iframe> <embed> 实现，小程序中实现为<webview> |
| Countdown | 倒计时组件 |
| Swiper | 轮播组件 |
| Modal | 弹出遮罩层的能力，为 Alert, Confirm 等对话框组件提供底层能力 |

JavaScript 复制代码

```
Example:
import { createElement, render, Fragment } from "rax";
import View from "rax-view";
import Text from "rax-text";
import Portal from "rax-portal";

const Demo = (props) => {
  return (
    <Fragment>
      <View>
        <Text>Demo content</Text>
      </View>
      <Portal>
        <View>
          <Text>Portal content</Text>
        </View>
      </Portal>
      <Portal container={document.body}> // 此元素渲染之document.body上
        <View>
          <Text>Portal with custom container content</Text>
        </View>
      </Portal>
    </Fragment>
  );
};

render(<Demo />);
```

## 6.2. 基础组件

使用Fusion Mobile作为移动端的组件库

使用方式

JavaScript 复制代码

```
// 安装依赖
npm install @alifd/meet -S
npm i build-plugin-fusion-mobile -D // 工程插件

// build.json配置
{
  "targets": [
    "web",
    "wechat-miniprogram"
  ],
  "plugins": [
    "@ali/build-plugin-rax-app-def",
+    "build-plugin-fusion-mobile"
  ]
}

// 业务中引入组件
import { createElement } from 'rax';
+ import { Button } from '@alifd/meet';
+ import '@alifd/meet/es/core/index.css';

export default function Home() {
  return (
    <>
+      <Button type="primary">Hello World</Button>
    </>
  );
}
```

具体组件参考：https://rax.js.org/docs/components/meet-about