从0到1上手 RN 开发

1. 前置基础

在学习 RN 之前，建议对以下技术有基本了解：

- **JavaScript（ES6+）**
  - 变量声明（let、const）、箭头函数、解构赋值、模板字符串、Promise、async/await 等

JavaScript 复制代码

```
1. 变量声明
ES5：只能用 var 声明变量，没有块级作用域。
ES6：新增 let（块级作用域）、const（块级常量声明）。

// ES5
var a = 10;
// ES6
let b = 20;
const c = 30;

2. 字符串模板
ES5：字符串拼接用 +。
ES6：引入模板字符串，用反引号 `，支持内嵌表达式 ${}

// ES5
var msg = "Hello " + name + "!";
// ES6
let msg = `Hello ${name}!`;

3. 箭头函数
ES5：只能用 function 声明普通函数。
ES6：引入箭头函数 () => {}，简化函数表达式并自动绑定this。

// ES5
function add(a, b) { return a + b; }
// ES6
const add = (a, b) => a + b;

4. 类（Class）
ES5：只能用构造函数+原型实现“类”。
ES6：引入了 class 语法和继承 extends。

// ES5
function Person(name) {
  this.name = name;
}
Person.prototype.sayHi = function() { console.log("Hi, " + this.name); };
// ES6
class Person {
```

```
  constructor(name) { this.name = name; }
  sayHi() { console.log(`Hi, ${this.name}`); }
}

5. 解构赋值
ES6：支持数组和对象结构赋值，简化赋值操作。
// ES6
let [a, b] = [1, 2];
let {name, age} = {name: "Tom", age: 18};

6. 扩展运算符/剩余参数（...）
// ES6 展开数组
let arr = [1, 2, 3];
let arr2 = [...arr, 4, 5];
// 剩余参数
function sum(...args) {}

8. 模块化
ES5：没有内置模块系统，一般使用CommonJS、AMD等方案（比如 Node.js 的 require）。
ES6：新增 import 和 export 原生模块化。
// ES6
export function foo() {}

import {foo} from './foo.js';

9. Promise 链式处理异步操作
ES5：没有Promise，需要回调函数处理异步。
ES6：原生支持Promise。
// ES6
function getData() {
  return new Promise((resolve, reject) => {
    // 异步操作比如网络请求
    setTimeout(() => {
      if (success) {
        resolve('OK');
      } else {
        reject('出错了');
      }
    }, 1000);
  });
}

getData()
    .then(res => {
        console.log('成功', res)
    }).catch(err => {
        console.log('失败', err)
    });

10. async/await 是基于 Promise 的语法糖，它让你用“同步形式”来书写异步代码
async function show() {
```

```
  try {
    let res = await getData();
    console.log(res);
  } catch (e) {
    console.log(e);
  }
}

show();
```

ECMAScript 6 入门

- **TypeScript基础**

1. TypeScript 和 JavaScript 主要区别

	JavaScript	TypeScript
类型系统	动态弱类型语言，变量类型在运行时决定，可以随意更改。	静态强类型语言（也可以弱类型），在编译
编译与运行	直接在浏览器/Node.js 中运行，无需编译	需要编译（transpile）成 JavaScript 后才
语法扩展	本身语法较为简单，没有类型相关语法、接口、泛型等。	增加了许多新特性，包括接口（interface）
错误检测	错误往往要在代码执行时才暴露，调试和维护难度更大。	在编译期间发现语法、类型等绝大多数错
兼容性	已被所有主流浏览器、Node.js 等原生支持。	只是开发用语言，最终需要编译成 JavaSc
兼容 JavaScript 代码	无法识别 TypeScript 的类型声明和新语法。	是 JavaScript 的超集，所有合法 JS 代码

2. TypeScript 主要特性

TypeScript 复制代码

```
1. 静态类型（Static Typing）
TypeScript 支持类型注解，允许开发者为变量、函数参数及返回值指定类型。静态类型帮助在编译期间发现错误，提升代码的可靠性和可维护性。
let age: number = 25;
function greet(name: string): void {
  console.log(`Hello, ${name}`);
}

2. 类型推断（Type Inference）
即使没有显式指定类型，TypeScript 也能根据变量的初始值自动推断其类型。
let message = "Hello"; // 推断为 string 类型

3. 接口（Interface）
用于定义对象的结构和类型约束，促进面向接口编程。
interface Person {
  name: string;
  age: number;
}
```

```
4. 类（Class）和面向对象编程（OOP）
TypeScript 扩展了 JavaScript 的类，支持修饰符（public、private、protected）、继承、抽象类等特性。
class Animal {
  constructor(public name: string) {}
  move() {
    console.log(`${this.name} is moving`);
  }
}

5. 枚举（Enum）
枚举类型用于定义一组命名常量，便于代码的可读性和可维护性。
enum Direction {
  Up,
  Down,
  Left,
  Right,
}

6. 泛型（Generics）
泛型使函数、类和接口在多种类型下复用，增强灵活性和类型安全。
function identity<T>(arg: T): T {
  return arg;
}

7. 类型别名（Type Aliases）
通过 type 关键字为类型命名，提高可读性。
type Point = { x: number; y: number };

8. 联合类型（Union Types）和交叉类型（Intersection Types）
支持变量可以是多种类型（联合），或者同时具备多种类型特性（交叉）。
let value: string | number;
type Admin = Employee & User;

9. 元组（Tuple）
元组允许定义已知数量和类型的数组。
let tuple: [string, number] = ["Tom", 25];

10. 类型守卫（Type Guards）和类型保护（Type Narrowing）
根据运行时检查缩小类型范围，提高类型安全。
function printId(id: number | string) {
  if (typeof id === "string") {
    console.log(id.toUpperCase());
  } else {
    console.log(id);
  }
}
```

TypeScript 教程

- **React 基础**

React 在 React Native 上是“能力移植+原生适配”，不支持 HTML、CSS、DOM 这些 Web 环境原生能力，你要用 RN 组件和样式语法实现界面。

React 的标准能力：

TypeScript 复制代码

```
“基于组件的声明式 UI 构建、响应式数据流和高可组合性”，拥有 props/state/hook/生命周期/事件/条件渲染等核心机制。

1. 组件化开发（函数组件、类组件）
函数组件：
function Hello(props) {
  return <Text>Hello, {props.name}</Text>;
}

类组件：
import React, { Component } from 'react';
import { Text } from 'react-native';

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

2. JSX(TSX) 语法
<View>
  <Text>这是 JS(TS) + XML 的混合写法</Text>
</View>

3. Props 属性传递
function Welcome(props) {
  return <Text>Welcome, {props.user}</Text>;
}
// 使用
<Welcome user="小明" />

4. State（状态）管理
函数组件：
import React, { useState } from 'react';
import { Button, Text, View } from 'react-native';

function Counter() {
  const [count, setCount] = useState(0);
  return (
    <View>
      <Text>{count}</Text>
      <Button title="加1" onPress={() => setCount(count + 1)} />
    </View>
  );
```

```
}

类组件：
class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
  }
  render() {
    return (
      <View>
        <Text>{this.state.count}</Text>
        <Button title="加1" onPress={() => this.setState({ count: this.state.count + 1
})} />
      </View>
    );
  }
}

5. 单向数据流
function Parent() {
  const [msg, setMsg] = useState('');
  return <Child onSend={text => setMsg(text)} />;
}

function Child({ onSend }) {
  return <Button title="发送内容" onPress={() => onSend("hello")} />;
}

6. 声明式事件机制
<Button title="点我" onPress={() => alert('按钮点击了！')} />
<TextInput onChangeText={value => console.log(value)} />

7. 生命周期管理 & Hook
类组件：
class Demo extends React.Component {
  componentDidMount() {
    console.log('组件挂载完成');
  }
  componentWillUnmount() {
    console.log('组件卸载');
  }
  render() {
    return <Text>Demo</Text>;
  }
}

函数组件 Hook：
import React, { useEffect } from 'react';
function Demo() {
  useEffect(() => {
```

```
    console.log('组件挂载');
    return () => {
      console.log('组件卸载');
    };
  }, []);
  return <Text>Demo</Text>;
}

8. Context（上下文）
const MyContext = React.createContext('默认值');

function Parent() {
  return (
    <MyContext.Provider value="来自父组件">
      <Child />
    </MyContext.Provider>
  );
}

function Child() {
  const value = React.useContext(MyContext);
  return <Text>{value}</Text>;
}

9. 条件渲染 & 列表渲染
条件渲染：
import { View, Text, Button } from 'react-native';
function AuthStatus() {
  const [isLoggedIn, setIsLoggedIn] = useState(false);
  if (isLoggedIn) {
    return (
      <View>
        <Text>Welcome User!</Text>
        <Button title="Logout" onPress={() => setIsLoggedIn(false)} />
      </View>
    );
  } else {
    return (
      <View>
        <Text>Please sign in</Text>
        <Button title="Login" onPress={() => setIsLoggedIn(true)} />
      </View>
    );
  }
}

FlatList（列表渲染）：
import { FlatList, Text } from 'react-native';
const data = [{ key: 'A' }, { key: 'B' }];
<FlatList
  data={data}
```

```
145    renderItem={({item}) => <Text>{item.key}</Text>}
146  />
147
```

React 教程

2. 环境搭建

- 安装 Node.js，建议使用 nvm 管理
- 安装 npm(pnpm 或者 yarn) 和 watchman
- 项目初始化：Expo 是 React Native 的增强工具套件，旨在简化移动开发的整个生命周期

npx create-expo-app@latest

经典结构

npx create-expo-app my-app -t expo-template-blank

- 运行项目

npx expo start

- 预构建原生项目代码

npx expo prebuild

- 安装依赖

npm install

- 配置 TypeScript 环境
  - 安装 TypeScript 及类型声明包

npm install --save-dev typescript @types/react @types/react-native

  - 自动生成 TypeScript 配置

npx tsc --init

- 开发工具
  - Cursor（强烈推荐）：ESLint 、Prettier、GitLens
  - VSCode
- 工程目录

Plain Text 复制代码

```
rn-project/
├── node_modules/           # npm/yarn 依赖
├── android/                # Android 原生代码
├── ios/                    # iOS 原生代码
├── src/                    # 主代码目录
├── .eslintrc               # ESLint(静态代码分析工具) 配置
├── .prettierrc             # 代码格式化配置
├── hammer.config.js        # 项目工程自动化(构建、发布、热更新、自动化CI/CD)
├── babel.config.js         # 设置 js 代码“转码规则”
├── metro.config.js         # metro 打包配置
```

```
├── app.json                # 应用元数据配置
├── tsconfig.json           # ts编译和类型检查配置
├── package.json            # 项目依赖和脚本
├── package-lock.json       # 锁定依赖版本
├── index.ts                # 项目入口文件
└── App.tsx                 # 首个页面
```

4. 基础学习

1. **常用组件**
   - 容器型

组件	说明
View	基础布局容器，类似 ，支持嵌套任意组件。
ImageBackground	用于设置背景图并包裹子内容容器
ScrollView	可滚动的容器，适合包裹需要滚动的内容。
TouchableOpacity	可触摸的容器，用于包裹需要点击反馈的组件（如按钮）。
KeyboardAvoidingView	自动调整布局避免键盘遮挡的容器。
SectionList	分组列表容器，通过 renderItem 和 renderSectionHeader 渲染内容。
FlatList	高性能列表容器，通过 renderItem 动态渲染子项。
Animated.View	支持动画的容器或组件，可包裹子组件实现动画效果。

   - 自闭合组件(无法嵌套子组件)

组件	说明
Text	文本组件，但**可嵌套其他 <Text> 组件**（仅限文本类内容）。
Image	图片组件
TextInput	输入框，内容通过 value 属性控制。
Button	按钮，标题通过 title 属性设置。
Switch	开关，状态通过 value 控制。
StatusBar	状态栏控制组件，无需也无法包裹子组件。

组件详细用法

2. **样式**

- StyleSheet 创建样式

TSX 复制代码

```tsx
import { StyleSheet, View, Text } from 'react-native';

function App() {
  return (
    <View style={styles.container}>
      <Text style={styles.title}>Hello StyleSheet!</Text>
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#F5FCFF',
  },
  title: {
    fontSize: 20,
    fontWeight: 'bold',
    color: '#333',
  },
});

```

- Flexbox 布局

类别	属性	说明	示例值
Flex 容器	flex	定义元素的伸缩比例（权重），默认 0（不伸缩）。	flex: 1
	flexDirection	主轴方向（子元素排列方向），默认 'column'。	'row' / 'column'
	flexWrap	子元素是否换行，默认 'nowrap'。	'wrap' / 'nowrap'
	justifyContent	子元素在主轴上的对齐方式。	'flex-start' / 'center' / 'spa
	alignItems	子元素在交叉轴上的对齐方式。	'stretch' / 'center' / 'flex-s
	alignContent	多行（或多列）本身在交叉轴上的整体分布	'stretch' / 'center' / 'flex-s
尺寸	width / height	绝对尺寸（数值）或相对尺寸（百分比）。	100 / '80%'
	minWidth / maxWidth	最小/最大宽度限制。	200 / '90%'
间距	margin	外边距（可单独设置 marginTop 等）。	10 / { vertical: 5, horizonta
	padding	内边距（可单独设置 paddingLeft 等）。	20 / { top: 10, bottom: 5 }

定位	position	定位类型，默认 'relative'。	'absolute' / 'relative'
	top / bottom	绝对定位时距离父容器顶/底部的距离。	10 / '20%'
	left / right	绝对定位时距离父容器左/右侧的距离。	0 / 'auto'
对齐	alignSelf	覆盖父容器的 alignItems，定义单个子元素的交叉轴对齐方式。	'flex-end' / 'center'
	textAlign	文本对齐方式（仅适用于 Text 组件）。	'center' / 'justify'
其他	zIndex	层级叠加顺序（数值越大越靠前）。	2
	overflow	内容超出容器时的处理方式。	'hidden' / 'visible'
	transform	元素变形（平移、旋转、缩放等）。	[{ translateX: 10 }, { scale

官方文档

5. 高级内容

- **Hooks 深入（useEffect、useRef 等）：**在 React 中，以 use 开头命名的函数被称为 Hook。

Hook	作用简介	简单示例
useState	状态管理	const [count, setCount] =
useEffect	副作用（生命周期）处理，例如数据请求、监听等	useEffect(()=>{...}, [deps]
useRef	保存可变引用（如获取子组件 ref、定时器 id）	const inputRef = useRef(n
useMemo	结果缓存优化，避免重复计算	useMemo(()=>exp, [deps]
useCallback	缓存函数的引用，避免不必要的函数“重新创建”，避免不必要的子组件重渲染	useCallback(fn, [deps])
useContext	跨组件层级共享数据/配置	const theme = useContex

  - Hook 规则
    - 只在顶层调用 Hook
    - 仅在 React 函数中调用 Hook

- **与原生模块交互通信（Linking、Native Modules）**

小红书业务具体介绍

- **动画（Animated）**

JSX 复制代码

```jsx
import React, { useRef, useEffect } from 'react';
import { Animated, View, Button } from 'react-native';

```

```
export default function FadeInViewExample() {
  const fadeAnim = useRef(new Animated.Value(0)).current; // 初始透明度 0
  useEffect(() => {
    Animated.timing(
      fadeAnim,
      {
        toValue: 1, // 最终透明度 1
        duration: 2000,
        useNativeDriver: true, // RN 0.62+ 建议总是写上
      },
    ).start();
  }, [fadeAnim]);

  return (
    <Animated.View  // 使用 Animated.View 包裹要做动画的元素
      style={{ opacity: fadeAnim }}>
      <View style={{ width: 200, height: 200, backgroundColor: 'pink' }} />
    </Animated.View>
  );
}

```