背景

1. 越来越多的业内人士表示: **「大前端」成为移动开发与 Web 前端的发展趋势，因为终端碎片化和 Serverless** 让这一切看起来更加可信。
    - **终端碎片化**顾名思义就是指终端越来越多样，比如 Apple Watch 手表、智能 TV、VR 眼镜等等。这些终端就和智能手机一样，支持第三方应用的嵌入。**Serverless** 字面意思是无服务架构，实际就是指用新的架构去代替传统服务器。
    - 在这样的大背景下，前端工程师不仅需要**掌握 PC 端和 H5 相关的开发能力，还需要对 Android、iOS**等各种终端的开发具有一定的了解。
    - 总体来看，我们认为前端框架、技术标准在不停演变，新的业务形态、用户终端也在不断涌现，这对于一个行业的发展来说或许是件好事；但对于开发者和团队来说，这意味着居高不下的学习和试错成本。伴随着支付宝小程序、快应用、百度智能小程序等终端的陆续入局，由于成本和效率的问题，业务产品也不太可能为每个终端都单独的进行开发工作。
2. 中国互联网络信息中心（CNNIC）发布的《中国互联网络发展状况统计报告》显示，截至 2018 年 6 月，**我国网民规模达 8.02 亿人，微信月活 10 亿 、支付宝月活 4 亿、百度月活 3.3 亿**；另一方面，**2018 Q3 中国 Android 手机占智能手机整体的比例超过 80%，月活约 6 亿**。
3. BAT 与 Android 成为了中国互联网真正的用户入口。
    - **流量高的入口级别 APP 都希望做平台，成为一个生态平台和互联网流量入口，大量第三方应用的接入，从业务层让公司 APP 关联上更多企业的利益，并且拥有更强的生命力**；
    - **从技术层面可以利用“本地能力接口层”收集大量用户数据，从消费互联网到产业互联网需要大量各行各业基础用户数据线索进行驱动和决策。**
4. 面对入口扩张，主端、独立端、微信小程序、支付宝小程序、百度小程序、Android 厂商联盟快应用，**单一功能在各平台都要重复实现，开发和维护成本成倍增加。迫切需要维护一套代码可以构建多入口的解决方案**。
5. 构建一个抹平小程序端开发差异的解决方案

Serverless

最近看了很多前端前辈写的总结文，最大的体会就是回忆 “前端在这几年到底起到了什么作用”。我们往往会夸大自己的存在感，其实前端存在的意义就是解决人机交互问题，大部分场景下，都是一种景上添花的作用，而不是必须。

回忆你最自豪的工作经历，可能是掌握了 Node 应用的运维知识、前端工程体系建设、研发效能优化、标准规范制定等，但真正对业务起效的部分，恰恰是你觉得写得最不值得一提的业务代码。前端花了太多的时间在周边技术上，而减少了很多对业务、交互的思考。

即便是大公司，也难以招到既熟练使用 Nodejs，又具备丰富运维知识的人，同时还要求他前端技术精湛，对业务理解深刻，鱼和熊掌几乎不可兼得。

Serverless 可以有效解决这个问题，前端同学只需要会写 JS 代码而无需掌握任何运维知识，就可以快速实现自己的整套想法。

趋势分析

多端框架可以大致分为三类：

1. 全包型。代表框架 Flutter，DSL（Dart）：
   - 优点非常明显：性能（的上限）高；各平台渲染结果一致。
   - 缺点也非常明显：需要完全重新学习 DSL（Dart），以及难以适配中国特色的端：小程序。
2. Web 技术型。代表框架是 React Native 和 Weex，DSL（前端框架：JavaScript，CSS）：
   - 这类框架把 Web 技术（JavaScript，CSS）带到移动开发中，自研布局引擎处理 CSS，使用 JavaScript 写业务逻辑，使用流行的前端框架作为 DSL，各端分别使用各自的原生组件渲染。
   - 优点：
     开发迅速
     复用前端生态
     易于学习上手，不管前端后端移动端，多多少少都会一点 JS、CSS
   - 缺点有：
     交互复杂时难以写出高性能的代码，这类框架的设计就必然导致 JS 和 Native 之间需要通信，类似于手势操作这样频繁地触发通信就很可能使得 UI 无法在 16ms 内及时绘制。
     由于没有渲染引擎，使用各端的原生组件渲染，相同代码渲染的一致性没有第一种高。
3. JavaScript 编译型。代表框架 Taro、WePY 、uni-app 、 mpvue 、 chameleon，DSL（JavaScript）：
   - 以这个 DSL 框架为标准在各端分别编译为不同的代码，各端分别有一个运行时框架或兼容组件库保证代码正确运行。
   - 优点：就是小程序，因为第一第二种框架其实除了可以跨系统平台之外，也都能编译运行在浏览器中。

生态比拼

开发工具

结果： `uni-app > Taro > chameleon > WePY、mpvue`

	chameleon	mpvue	Taro	uni-app	WePY
DSL	类 Vue	Vue	React	Vue	类 Vue
IDE/图形化开发工具	无	无	无	有	无
语法校验工具	自研	无	ESLint 规则	IDE 支持	无
TypeScript	无	有	有	有	有
Typing/自动补全	无	API	API + JSX	IDE 支持	无
样式	sass, less, stylus	sass, less, stylus	sass, less, stylus, CSS Modules	sass, less, stylus	sass, less, stylus

多端支持度

结果： `chameleon > Taro、uni-app > mpvue、WePY`

	chameleon	mpvue	Taro	uni-app	WePY
移动端容器	Weex	无	React Native	Weex	无
移动端增强	chameleon SDK	无	Expo	nvue	无
多端编译方式	自研多态协议	环境变量条件编译	环境变量 + 文件条件编译	自研条件编译语法	环境变量条件编译
H5 兼容 API	自研多态协议	无	有	有	无
跨端组件库	有	无	有	有	无
小程序支持	微信、百度、支付宝	微信、百度、支付宝、字节跳动	微信、百度、支付宝、字节跳动	微信、百度、支付宝、字节跳动	微信、百度、支付宝

组件库/工具库/demo

结果： `WePY > uni-app 、taro > mpvue > chameleon`

	chameleon	mpvue	Taro	uni-app	WePY
自研组件库	有	无	有	有	无
第三方组件	较少	丰富	较丰富	丰富	丰富
第三方工具库	较少	较丰富	较丰富	丰富	丰富
Demo	较少	较丰富	较丰富	较丰富	丰富
状态管理工具	Vuex	Vuex	Redux/MobX/Dva	Vuex	Redux
转换微信小程序工具	无	无	有	无	无

接入成本

结果： `Taro > mpvue 、 uni-app > WePY > chameleon`

	chameleon	mpvue	Taro	uni-app	WePY
移动端容器	Weex	无	React Native	Weex	无
移动端增强	chameleon SDK	无	Expo	nvue	无
多端编译方式	自研多态协议	环境变量条件编译	环境变量 + 文件条件编译	自研条件编译语法	环境变量条件编译
H5 兼容 API	自研多态协议	无	有	有	无
跨端组件库	有	无	有	有	无
小程序支持	微信、百度、支付宝	微信、百度、支付宝、字节跳动	微信、百度、支付宝、字节跳动	微信、百度、支付宝、字节跳动	微信、百度、支付宝

流行度

结果： `uni-app > Taro、WePY、mpvue > chameleon`

	chameleon	mpvue	Taro	uni-app	WePY
GitHub Star	3843	16281	16084	2921	16779
GitHub Issue/PR	68/8	1369/104	2026/368	215/18	1662/441
NPM + CNPM 下载量 [1]	241/周	1971/周	4332/周	N/A	976/周
案例	较少	丰富	丰富	非常丰富	丰富
开发者人数 [2]	~1000 人	~ 5000 人	~ 5000 人	10000+	~ 5000 人
自建开发者社区	无	无	无	有	无

[1] 统计上周各框架 CLI 工具下载量，uni-app 可用 IDE 工具直接开发

[2] 统计框架及相关生态微信/QQ 群的总人数

生态对比图表

		chameleon	mpvue	Taro	uni-app	WePY
开发工具	DSL	类 Vue	Vue	React	Vue	类 Vue
	IDE/图形化开发工具	无	无	无	有	无
	语法校验工具	自研	无	ESLint 规则	IDE 支持	无
	TypeScript	无	有	有	有	有
	Typing/自动补全	无	API	API + JSX	IDE 支持	无
	样式	sass, less, stylus	sass, less, stylus	sass, less, stylus, CSS Modules	sass, less, stylus	sass, less, stylus
多端支持	小程序支持	微信、百度、支付宝	微信、百度、支付宝、字节跳动	微信、百度、支付宝、字节跳动	微信、百度、支付宝、字节跳动	微信、百度、支付宝
	移动端容器	Weex	无	React Native	Weex	无
	移动端增强	chameleon SDK	无	Expo	nvue	无
	多端编译方式	自研多态协议	环境变量条件编译	环境变量 + 文件条件编译	自研条件编译语法	环境变量条件编译
	H5 兼容 API	自研多态协议	无	有	有	无
	跨端组件库	有	无	有	有	无
组件库/工具库/demo	第三方组件	较少	丰富	较丰富	丰富	丰富
	第三方工具库	较少	较丰富	较丰富	丰富	丰富
	Demo	较少	较丰富	较丰富	较丰富	丰富
	状态管理工具	Vuex	Vuex	Redux/MobX/Dva	Vuex	Redux
	转换微信小程序工具	无	无	有	无	有
	自研组件库	有	无	有	有	无
流行度	GitHub Star	3843	16281	16084	2921	16779
	GitHub Issue/PR	68/8	1369/104	2026/368	215/18	1662/441
	NPM + CNPM 下载量 [1]	241/周	1971/周	4332/周	N/A	976/周
	案例	较少	丰富	丰富	非常丰富	丰富
	开发者人数 [2]	~1000 人	~ 5000 人	~ 5000 人	10000+	~ 5000 人
	自建开发者社区	无	无	无	有	无

* 数据采集于 2019 年 3 月 11 日，以各框架目前稳定版特性为标准

[1] 统计上周各框架 CLI 工具下载量，uni-app 可用 IDE 工具直接开发

[2] 统计框架及相关生态微信/QQ 群的总人数

演示：Taro 框架应用及脚手架

Taro 框架应用及原理分析

编译原理

代码到底是什么

其实代码跟 JSON 一样，是一种结构化的文本数据格式。在这里我们要仅仅抓着两个特点——“文本”和“结构化”。

- 代码的第一个特点是文本，那意味着我们所有对字符串的拼接、截取或者替换等所有操作，都可以应用在代码上面。很多程序员虽然都能对各类文本的读写了如指掌，但大家好像都没有意识到代码文件，也可以是那个可以读写、修改的文件之一。

- 代码的第二个特点则是结构化。不知道大家能不能理解，代码里面除了字面量意外，其他部分都只是标识结构而并不具有实际意义，赋予这些结构意义是解释器如何和执行这段代码。这个特点就是要求我们在看待代码的时候，要在脑中形成一种结构，而不再是一行一行的字符串。

真正有意识地把代码文件当成文本文件以后，我们就能把代码从此拉下“神坛”，可以让大家能够像思考文本一样思考代码。

不管是任意语言的代码，其实它们都有两个共同点：

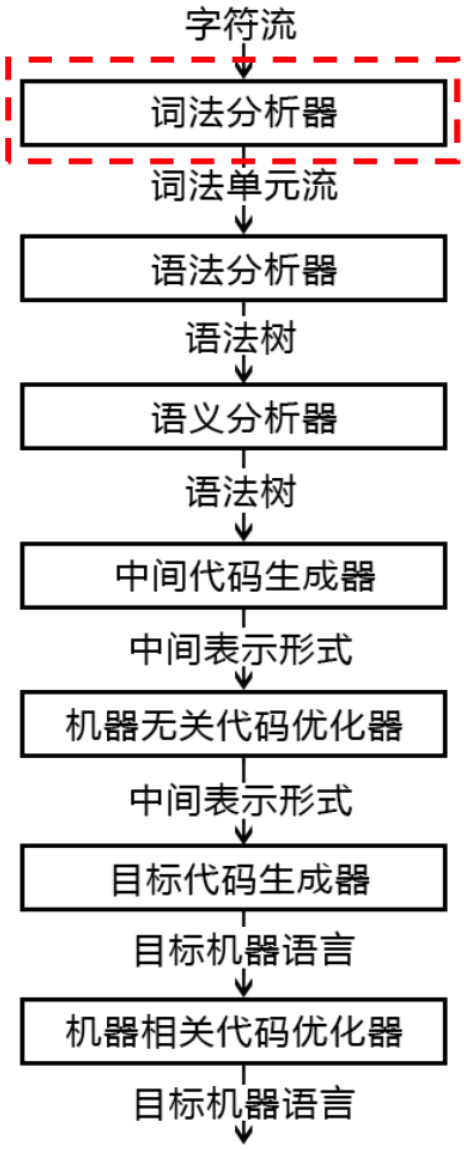
1. 它们都是由字符串构成的文本
2. 它们都要遵循自己的语言规范

编译器前端技术 —— Parser。

- 编译器前段就在干一件事，把代码这个结构化的文本文件解析成我们计算机可以理解的数据结构 —— 抽象语法树（AST）。
- 解析代码一般分成两个步骤，第一个步骤是词法分析，将文本的代码转化成一个个 Token。看到这里的大家应该都有一些正则表达式的基础吧，在解析代码的过程中，我们需要用正则来分词做词法分析。
- 解析代码的第二个步骤是语法分析，语法分析是将我们上面词法分析出的 Token 转化成 AST。语法分析我们要学习上下文无关文法（CFG），并且可以用 BNF 这个表示。CFG 比正则表达能力更强，强在 CFG 能表达递归结构，常见的递归结构有表达式和代码块。
- Parser 在编译原理里面是难点但却不是重点，所以在这一部分大家觉得复杂的算法完全可以跳过，不建议浪费太多时间。Parser 都是可以根据正则和 CFG 自动生成的，并不需要自己手写。所以这部分主要目的是学好的是正则和 CFG，那些复杂的算法学起来意义很小。
- parser 并不是编译器，它甚至不属于编译里很重要的东西。

编译原理分析

编译器的结构



http://www.doc88.com/p-3971550434706.html
https://max.book118.com/html/2017/0620/116914353.shtm

Babel 模块

选择使用 Babel，主要有以下几个原因：

- Babel 可以解析还没有进入 ECMAScript 规范的语法。例如装饰器这样的提案，虽然现在没有进入标准但是已经广泛使用有一段时间了；
- Babel 提供插件机制解析 TypeScript、Flow、JSX 这样的 JavaScript 超集，不必单独处理这些语言；
- Babel 拥有庞大的生态，有非常多的文档和样例代码可供参考；
- 除去 parser 本身，Babel 还提供各种方便的工具库可以优化、生成、调试代码。

Babel 是一个通用的多功能的 JavaScript 编译器，更确切地说是源码到源码的编译器，通常也叫做转换编译器（transpiler）。 意思是说你为 Babel 提供一些 JavaScript 代码，Babel 更改这些代码，然后返回给你新生成的代码。

> 静态分析是在不需要执行代码的前提下对代码进行分析的处理过程（执行代码的同时进行代码分析即是动态分析）。 静态分析的目的是多种多样的， 它可用于语法检查、编译、代码高亮、代码转换、优化和压缩等等场景。

Babel 实际上是一组模块的集合，拥有庞大的生态。Taro 项目的代码编译部分就是基于 Babel 的以下模块实现的：

1. Babylon - Babel 的解析器。它可以把一段符合规范的 JavaScript 代码输出成一个符合 Esprima 规范的 AST。大部分 parser 生成的 AST 数据结构都遵循 Esprima 规范，包括 ESLint 的 parser ESTree。这就意味着我们熟悉了 Esprima 规范的 AST 数据结构还能去写 ESLint 插件。

```
import * as babylon from "babylon";

const code = `n * n`;

babylon.parse(code);
```

```
- expression: BinaryExpression {
    type: "BinaryExpression"
    start: 3
    end: 8
  + loc: {start, end}
  - left: Identifier {
      type: "Identifier"
      start: 3
      end: 4
    + loc: {start, end, identifierName}
      name: "n"
    }
    operator: "*"
  - right: Identifier = $node {
      type: "Identifier"
      start: 7
      end: 8
    + loc: {start, end, identifierName}
      name: "n"
    }
  }
```

2. Babel-traverse - babel-traverse 可以遍历由 Babylon 生成的抽象语法树，并把抽象语法树的各个节点从拓扑数据结构转化成一颗路径（Path）树，Path 表示两个节点之间连接的响应式（Reactive）对象，它拥有添加、删除、替换节点等方法。当你调用这些修改树的方法之后，路径信息也会被更新。除此之外，Path 还提供了一些操作作用域（Scope） 和标识符绑定（Identifier Binding） 的方法可以去做处理一些更精细复杂的需求。可以说 babel-traverse 是使用 Babel 作为编译器最核心的模块。

```
import * as babylon from "@babel/parser";
import traverse from "babel-traverse";

const code = `function square(n) {
  return n * n;
}`;

const ast = babylon.parse(code);

traverse(ast, {
  enter(path) {
    if (path.node.type === "Identifier" && path.node.name === "n") {
      path.node.name = "x";
    }
  }
});
```

3. Babel-types - 一个用于 AST 节点的 Lodash 式工具库， 它包含了构造、验证以及变换 AST 节点的方法。 该工具库包含考虑周到的工具方法，对编写处理 AST 逻辑非常有用。

```
import traverse from "babel-traverse";
import * as t from "babel-types";

traverse(ast, {
  enter(path) {
    if (t.isIdentifier(path.node, { name: "n" })) {
      path.node.name = "x";
    }
  }
});
```

4. Babel-generator - Babel 的代码生成器，它读取 AST 并将其转换为代码和源码映射（sourcemaps）。
5. Babel-template - 另一个虽然很小但却非常有用的模块。 它能让你编写字符串形式且带有占位符的代码来代替手动编码， 尤其是生成大规模 AST 的时候。 在计算机科学中，这种能力被称为准引用（quasiquotes）。

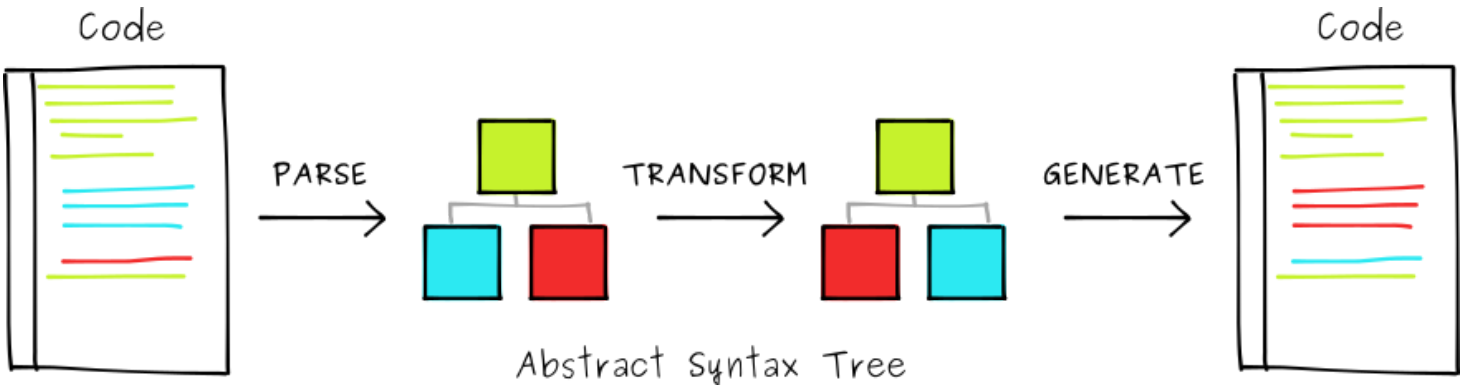
解析页面 Config 配置源码

```
// 1. babel-traverse方法，  遍历和更新节点
traverse(ast, {
  ClassProperty(astPath) {
    // 遍历类的属性声明
    const node = astPath.node;
    if (node.key.name === "config") {
      // 类的属性名为 config
      configObj = traverseObjectNode(node);
      astPath.remove(); // 将该方法移除掉
    }
  }
});

// 2. 遍历，解析为 JSON 对象
function traverseObjectNode(node, obj) {
  if (node.type === "ClassProperty" || node.type === "ObjectProperty") {
    const properties = node.value.properties;
    obj = {};
    properties.forEach((p, index) => {
      obj[p.key.name] = traverseObjectNode(p.value);
    });
    return obj;
  }
  if (node.type === "ObjectExpression") {
    const properties = node.properties;
    obj = {};
    properties.forEach((p, index) => {
      // const t = require('babel-types')  AST 节点的 Lodash 式工具库
      const key = t.isIdentifier(p.key) ? p.key.name : p.key.value;
      obj[key] = traverseObjectNode(p.value);
    });
    return obj;
  }
  if (node.type === "ArrayExpression") {
    return node.elements.map(item => traverseObjectNode(item));
  }
  if (node.type === "NullLiteral") {
    return null;
  }
  return node.value;
}

// 3. 写入对应目录的 *.json 文件
fs.writeFileSync(outputPageJSONPath, JSON.stringify(configObj, null, 2));
```

Taro 原理

编译工作流与抽象语法树（AST）

Taro 的核心部分就是将代码编译成其他端（H5、小程序、React Native 等）代码。一般来说，将一种结构化语言的代码编译成另一种类似的结构化语言的代码包括以下几个步骤：



首先是 Parse，将代码解析（Parse）成抽象语法树（Abstract Syntex Tree），然后对 AST 进行遍历（traverse）和替换(replace)（这对于前端来说其实并不陌生，可以类比 DOM 树的操作），最后是生成（generate），根据新的 AST 生成编译后的代码。

https://juejin.im/post/5ba346a7f265da0ad13b78bd

demo：

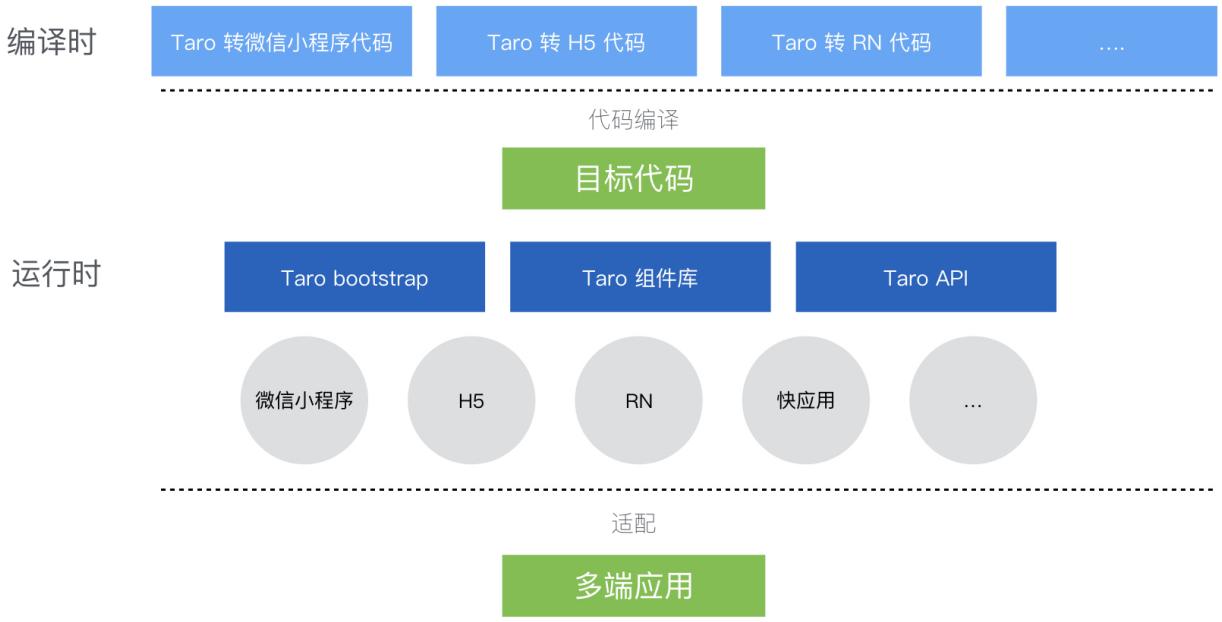
```
import Taro, { Component } from "@tarojs/taro";
import { View, Text } from "@tarojs/components";

class Home extends Component {
  config = {
    navigationBarTitleText: "首页"
  };

  state = {
    numbers: [1, 2, 3, 4, 5]
  };

  handleClick = () => {
    this.props.onTest();
  };

  render() {
    const oddNumbers = this.state.numbers.filter(number => number & 2);
    return (
      <ScrollView className="home" scrollTop={false}>
        奇数:
        {oddNumbers.map(number => (
          <Text onClick={this.handleClick}>{number}</Text>
        ))}
        偶数:
        {numbers.map(
          number =>
            number % 2 === 0 && <Text onClick={this.handleClick}>{number}</Text>
        )}
      </ScrollView>
    );
  }
}
```

设计思路

Taro 的结构主要分两个方面：运行时和编译时。运行时负责把编译后到代码运行在本不能运行的对应环境中，你可以把 Taro 运行时理解为前端开发当中 polyfill(Polyfill 你可以理解为“腻子”，就是装修的时候，可以把缺损的地方填充抹平)。

设计思想

02



taro编译时

1. config => JSON。
2. state => data。
3. 函数 handleClick 我们交给运行时处理。
4. 和 React 一样，Taro 每次更新都会调用 render 函数，但和 React 不同的是：
   - React 的 render 是一个创建虚拟 DOM 的方法
   - Taro 的 render 会被重命名为 _createData，它是一个创建数据的方法：在 JSX 使用过的数据都在这里被创建最后放到小程序 Page 或 Component 工厂方法中的 data 。最终我们的 render 方法会被编译为：

```
_createData() {
this.__state = arguments[0] || this.state || {};
this.__props = arguments[1] || this.props || {};

const oddNumbers = this.__state.numbers.filter(number => number & 2);
Object.assign(this.__state, {
    oddNumbers: oddNumbers
});
return this.__state;
}
```

5. WXML 和 JSX

   在 Taro 里 render 的所有 JSX 元素都会在 JavaScript 文件中被移除，它们最终将会编译成小程序的 WXML：

- 首先我们可以先看 `<View className='home'>`，它在 AST 中是一个 JSXElement，它的结构和我们定义 Element 类型差不多。我们先将 JSXElement 的 ScrollView 从驼峰式的 JSX 命名转化为短横线（kebab case）风格，className 和 scrollTop 的值分别代表了 JSXAttribute 值的两种类型：StringLiteral 和 JSXExpressionContainer，className 是简单的 StringLiteral 处理起来很方便，scrollTop 处理起来稍微麻烦点，我们需要用两个花括号 {} 把内容包起来。
  接下来我们再思考一下每一个 JSXElement 出现的位置，你可以发现其实它的父元素只有几种可能性：return、循环、条件（逻辑）表达式。而在上一篇文章中我们提到，babel-traverse 遍历的 AST 类型是响应式的——也就是说只要我们按照 JSXElement 父元素类型的顺序穷举处理这几种可能性，把各种可能性大结果应用到 JSX 元素之后删除掉原来的表达式，最后就可以把一个复杂的 JSX 表达式转换为一个简单的 WXML 数据结构。
- `oddNumbers.map(number => <Text onClick={this.handleClick}>{number}</Text>)`
  Text 的父元素是一个 map 函数（CallExpression），我们可以把函数的 callee: oddNumbers 作为 wx:for 的值，并把它放到 state 中，匿名函数的第一个参数是 wx:for-item 的值，函数的第二个参数应该是 wx:for-index 的值。
  对于 onClick 而言，在 Taro 中 on 开头的元素参数都是事件，所以我们只要把 this. 去掉即可。Text 元素的 children 是一个 JSXExpressionContainer，我们按照之前的处理方式处理即可。最后这行我们生成出来的数据结构应该是这样：

```
{
  "type": "element",
  "tagName": "text",
  "attributes": [
    { "bindtap": "handleClick" },
    { "wx:for": "{{oddNumbers}}" },
    { "wx:for-item": "number" }
  ],
  "children": [{ "type": "text", "content": "{{number}}" }]
}
```

- `numbers.map(number => number % 2 === 0 && <Text onClick={this.handleClick}>{number}</Text>)`
  由于 wx:if 和 wx:for 同时作用于一个元素可能会出现问题，所以我们应该生成一个 block 元素，把 wx:if 挂载到 block 元素，原元素则全部作为 children 传入 block 元素中。这时 babel-traverse 会检测到新的元素 block，它的父元素是一个 map 循环函数，因此我们可以按照第一个循环表达式的处理方法来处理这个表达式。

taro 小程序运行时

编译生成好的代码仍然不能直接运行在小程序环境里。

> 在小程序里，程序的功能及配置与页面和组件差异较大，因此运行时提供了两个方法 createApp 和 createComponent 来分别创建程序和组件（页面）。createApp 的实现非常简单，本章我们主要介绍 createComponent 做的工作。

createComponent 方法主要做了这样几件事情：

- 将组件的 state 转换成小程序组件配置对象的 data
- 将组件的生命周期对应到小程序组件的生命周期
- 将组件的事件处理函数对应到小程序的事件处理函数

组件 state 转换

其实在 Taro（React） 组件里，除了组件的 state，JSX 里还可以访问 props 、render 函数里定义的值、以及任何作用域上的成员。而在小程序中，与模板绑定的数据均来自对应页面（或组件）的 data 。因此 JSX 模板里访问到的数据都会对应到小程序组件的 data 上。

生命周期

> 小程序的页面除了渲染过程的生命周期外，还有一些类似于 onPullDownRefresh 、 onReachBottom 等功能性的回调方法也放到了生命周期回调函数里。这些功能性的回调函数，Taro 未做处理，直接保留了下来。

componentWillMount
在微信小程序中这一生命周期方法对应页面的onLoad或入口文件app中的onLaunch

componentDidMount
在微信小程序中这一生命周期方法对应页面的onReady或入口文件app中的onLaunch，在 componentWillMount后执行

componentDidShow
在微信小程序中这一生命周期方法对应 onShow

componentDidHide
在微信小程序中这一生命周期方法对应 onHide

componentDidCatchError
错误监听函数，在微信小程序中这一生命周期方法对应 onError

componentDidNotFound
页面不存在监听函数，在微信小程序中这一生命周期方法对应 onPageNotFound

shouldComponentUpdate
页面是否需要更新

componentWillUpdate
页面即将更新

componentDidUpdate
页面更新完毕

componentWillUnmount
页面退出，在微信小程序中这一生命周期方法对应 onUnload

在小程序中 ，页面还有一些专属的方法成员，如下：

1. onPullDownRefresh： 页面相关事件处理函数–监听用户下拉动作
2. onReachBottom： 页面上拉触底事件的处理函数
3. onShareAppMessage： 用户点击右上角转发
4. onPageScroll： 页面滚动触发事件的处理函数
5. onTabItemTap： 当前是 tab 页时，点击 tab 时触发

6. componentWillPreload： 预加载，只在微信小程序中可用

Taro 组件的 setState 行为最终会对应到小程序的 setData。Taro 引入了如 nextTick ，编译时识别模板中用到的数据，在 setData 前进行数据差异比较等方式来提高 setState 的性能。

如上图，组件调用 setState 方法之后，并不会立刻执行组件更新逻辑，而是会将最新的 state 暂存入一个数组中，等 nextTick 回调时才会计算最新的 state 进行组件更新。这样即使连续多次的调用 setState 并不会触发多次的视图更新。在小程序中 nextTick 是这么实现的：

```
const nextTick = (fn, ...args) => {
  fn = typeof fn === 'function' ? fn.bind(null, ...args) : fn
  const timerFunc = wx.nextTick ? wx.nextTick : setTimeout
  timerFunc(fn)
}
```

除了计算出最新的组件 state ， 在组件状态更新过程里还会调用前面提到过的 _createData 方法，得到最终小程序组件的 data，并调用小程序的 setData 方法来进行组件的更新。

事件处理函数对应

在小程序的组件里，事件响应函数需要配置在 methods 字段里。而在 JSX 里，事件是这样绑定的：

`<View onClick={this.handleClick}></View>` 编译的过程会将 JSX 转换成小程序模板： `<view bindclick="handleClick"></view>`

在 createComponent 方法里，会将事件响应函数 handleClick 添加到 methods 字段中，并且在响应函数里调用真正的 this.handleClick 方法。

在编译过程中，会提取模板中绑定过的方法，并存到组件的 $events 字段里，这样在运行时就可以只将用到的事件响应函数配置到小程序组件的 methods 字段中。

在运行时通过 processEvent 这个方法来处理事件的对应，省略掉处理过程，就是这样的：

```
function processEvent (eventHandlerName, obj) {
  obj[eventHandlerName] = function (event) {
    // ...
        scope[eventHandlerName].apply(callScope, realArgs)
  }
}
```

这个方法的核心作用就是解析出事件响应函数执行时真正的作用域 callScope 以及传入的参数。在 JSX 里，我们可以像下面这样通过 bind 传入参数：

```
<View onClick={this.handleClick.bind(this, arga, argb)}></View>
```

对 API 进行 Promise 化的处理

Taro 对小程序的所有 API 进行了一个分类整理，将其中的异步 API 做了一层 Promise 化的封装。例如，wx.getStorage经过下面的处理对应到Taro.getStorage(此处代码作示例用，与实际源代码不尽相同)：

```
Taro['getStorage'] = options => {
  let obj = Object.assign({}, options)
  const p = new Promise((resolve, reject) => {
        ['fail', 'success', 'complete'].forEach((k) => {
          obj[k] = (res) => {
            options[k] && options[k](res)
            if (k === 'success') {
                  resolve(res)
            } else if (k === 'fail') {
                  reject(res)
            }
          }
        })
        wx['getStorage'](obj)
  })
  return p
}
```

就可以这么调用了：

```
// 小程序的调用方式
Taro.getStorage({
  key: 'test',
  success() {

  }
})
// 在 Taro 里也可以这样调用
Taro.getStorage({
  key: 'test'
}).then(() => {
  // success
})
```

- Taro 小程序运行时是如何配合编译过程，抹平了状态、事件绑定以及生命周期的差异，使得 Taro 组件运行在小程序环境中。
- 通过运行时对原生 API 进行扩展，实现了诸如事件绑定时通过 bind 传递参数、通过 Promise 的方式调用原生 API 等特性。
- 本章节参考 Taro 源码
  @tarojs/taro-weapp，微信小程序运行时
  @tarojs/taro-swan，百度小程序运行时
  @tarojs/taro-alipay，支付宝小程序运行时

taro H5运行时

组件实现

我们不妨捋一捋小程序和 Web 开发在这些组件上的差异：

image-20180903170556961

作为开发者，你第一反应或许会尝试在编译阶段下功夫，尝试直接使用效果类似的 Web 组件替代：用div替代view，用img替代image，以此类推。

API 适配

除了组件，小程序下有一些 API 也是 Web 开发中所不具备的。比如小程序框架内置的wx.request/wx.getStorage等API；但在 Web 开发中，我们使用的是fetch/localStorage等内置的函数或者对象。

image-20180903170610928

小程序的 API 实现是个巨大的黑盒，我们仅仅知道如何使用它，使用它会得到什么结果，但对它内部的实现一无所知。

如何让 Web 端也能使用小程序框架中提供的这些功能？既然已经知道这个黑盒的入参出参情况，那我们自己打造一个黑盒就好了。

换句话说，我们依然通过运行时框架来实现这些 Web 端不存在的能力。

路由

作为小程序的一大能力，小程序框架中以栈的形式维护了当前所有的页面，由框架统一管理。

实现形式上，我们参考react-router：监听页面路径变化，再触发 UI 更新。这是React的精髓之一，单向数据流。

@tarojs/router包中包含了一个轻量的history实现。history中维护了一个栈，用来记录页面历史的变化。对历史记录的监听，依赖两个事件：hashchange和popstate。

/* 示例代码 */

```
window.addEventListener('hashchange', () => {});
window.addEventListener('popstate', () => {})
```

对于使用 Hash 模式的页面路由，每次页面跳转都会依次触发popstate和hashchange事件。由于在popstate的回调中可以取到当前页面的 state，我们选择它作为主要跳转逻辑的容器。

三种操作：push, pop，还有replace。

H5 转换与优化升级

Taro H5 端在构建过程中，使用 webpack 作为构建的核心。在 webpack 中使用 treeshaking 功能有几个需要注意的地方：

如果是 npm 模块，需要package.json中存在sideEffects字段，并且准确配置了存在副作用的源代码。

必须使用 ES6 模块语法。由于诸如babel-preset-env之类的 babel 预配置包默认会对代码的模块机制进行改写，还需要将modules设置为false，将模块解析的工作直接交给 webpack。

需要工作在 webpack 的production模式下。

https://juejin.im/post/5c7dec9cf265da2ddc3c9a7d#heading-10

踩坑

一次编写,到处运行终是梦,一次运行,各种填坑才是真。