如何优雅的『搞』小程序之工程流程化

伴随着从2017年1月9日凌晨的夜色，张小龙和他的团队正式发布了微信小程序，瞬间刷爆了业内人士的各种信息流，一个看似『银弹』的产品形态被大家所热捧，一股开发浪潮也随之而来。而近期，微信又做成了两个很大的突破：微信悖逆苹果做小程序市场（应用内搜索已经有『市场性质』）和允许小程序通过社交传播（可以通过朋友圈和聊天传播，二维码可以长按识别），同时让开发者们看到微信小程序发展的信心和前景。总之，不管是『红利』还是『鸡血』，小程序将成为产品布局中重要的一个组成部分。因此，在公司内部，将小程序开发流程统一化、规划化，让小程序开发变得优雅是相当有必要的。

下面，GitChator将以去哪儿网微信小程序为背景，详细对去哪儿网微信小程序工程流程化方案进行剖析，同时与读者们进行交流探讨。

小程序开发问题的转变

读者们一定很奇怪，为什么先聊到『开发问题的转变』？因为做架构、做工程流程化的目的是为了降低开发、测试、发布、运维等一系列环节的成本的同时，解决这些环节之

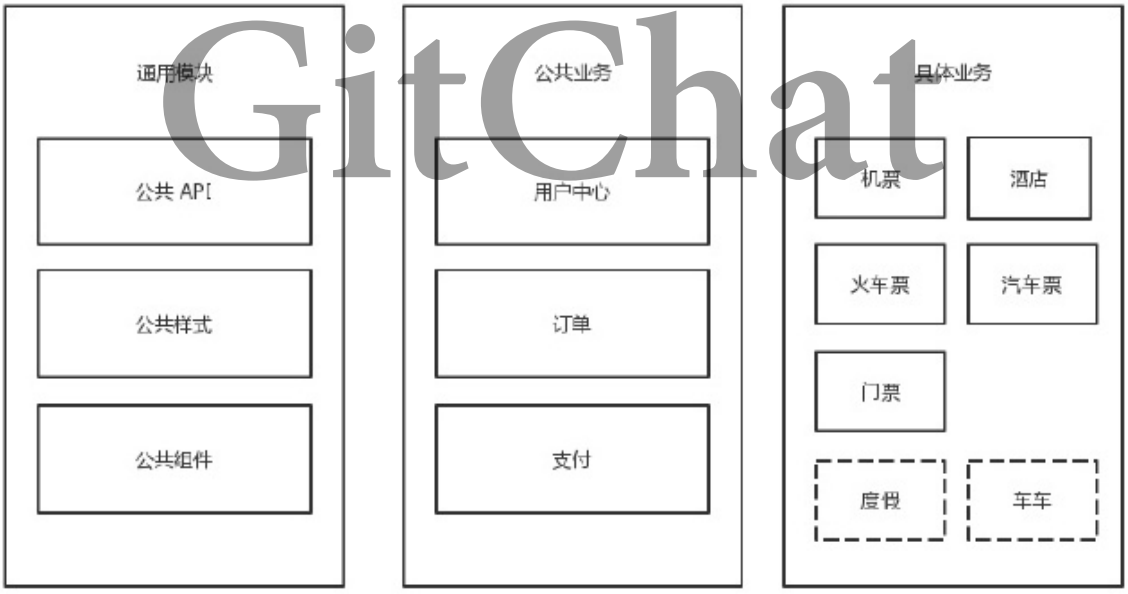
间存在的问题和痛点。所以理清开发小程序的过程中遇到的问题，才能更好的做架构、做工程流程化。

那么，小程序开发急需解决什么问题？而问题又有什么转变呢？

从内测开始到半个月之前，有一个数字非常让人头疼 —— **1024**。这个数字代表着小程序的总 Size，小程序打包后的总体积不能超过 1024 KB （也就是 1 MB）。在这个有限的空间内，放入更多的业务逻辑，这才是当时小程序开发遇到的最大的问题。因此，**压缩** 是当时首当其冲的工作。

当初，公司准备将去哪儿酒店、去哪儿门票和去哪儿交通三个小程序中的五个业务线合并到统一的一个小程序中。笔者所在的团队接到了这个任务，并在业务线配合下，在一个星期的时间里，完成任务上线，而后又在剩余的体积内塞入了两个其他的业务。在当时的情况下，每压缩出 10KB 的体积，都是一件很令人兴奋的事。

而现在不同了，微信将小程序的 Size 增加到 **2MB**，也就是 2048 KB，足足 **翻了一倍**。 由于有之前 1024 的经验，翻倍后，在相同的压缩逻辑下，已经足够满足绝大多数 App 的要求。例如，将去哪儿旅行 App 上的主要业务的主要流程都放进微信小程序内，应该是没问题的。

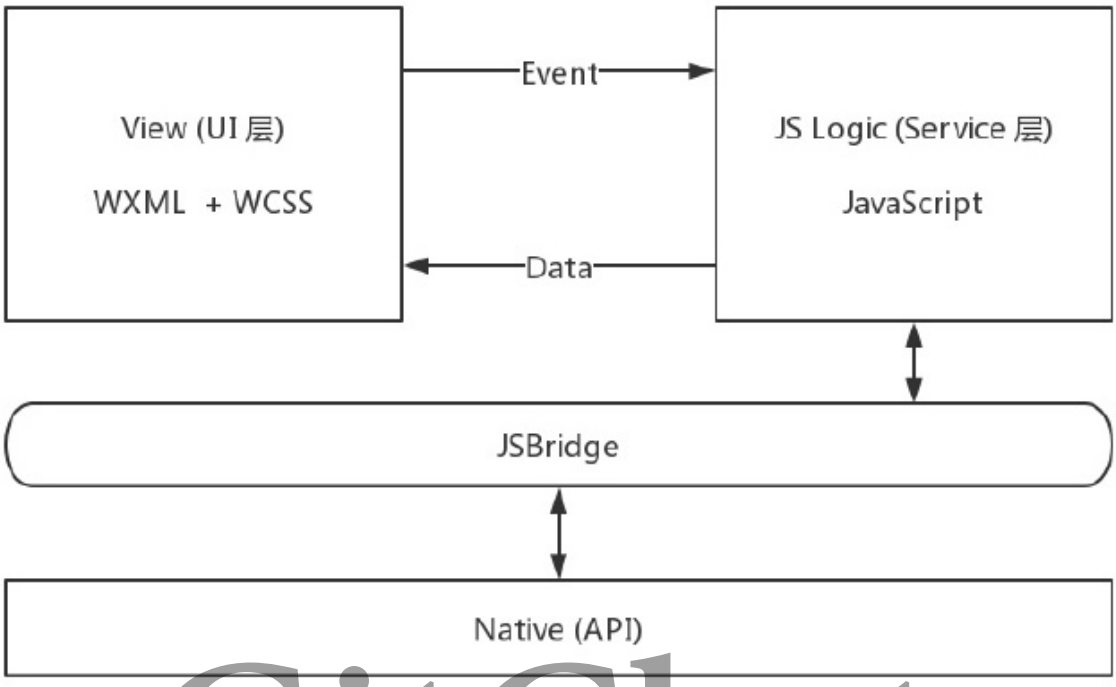


所以，在压缩问题并不突出明显后，**如何更好的管控小程序的代码**，**如何做好业务隔离**，**如何分配业务资源的配比**，这些问题将会成为现在以及以后核心要解决的问题。当然，压缩也是工程流程化的一部分，只是优先级降低而已。

最后，提出一个问题：据上文所述，**可以将去哪儿旅行 App 上的主要业务的主要流程都放进微信小程序内**，但是在灵活的小程序应用场景下，这样真的好吗？真的适合小程序的场景吗？这也是工程流程化要考虑和完善的一个问题。

小程序的架构模式

对于小程序的架构，用一句话总结就是『类 React Native 的 MVC Web UI 架构』。小程序的架构思想与 React Native 类似，都是以组件化的方式和 MVC 的模式将 UI 层和 Service 层分离，在保证 UI 层的显示效率的同时，也保持了 Service 层的一致性；而与 React Native 最大的不同是：React Native 使用的是 Native 的 UI，而微信使用的 Web UI。



相比来说，Native 的 UI 性能更好，但也更依赖系统和 App 本身的 Native，不易维护和进行热更新；而 Web UI 性能和 Native UI 有一定的差距，但是依赖少，易维护并且容易实现热更新。

小程序的架构就先简单介绍到这里，大家可以从其他的 GitChat 文章里了解更多。这次 GitChat 主要要阐述的是小程序的工程流程化问题，而为了实现工程流程化，在简单了解小程序架构的同时，也必须先了解基于这套架构的一些技术细节，例如代码结构、构建方法、调试发布方式等。

下面简单列举一些比较重要的点。

代码结构

首先，在工程内有三个公共入口文件：

- `app.json` ：配置文件，配置路由列表、程序信息等。
- `app.js` ：公共入口文件，小程序启动时的 Init 逻辑。
- `app.wxss` ：公共样式文件，公共样式用于每个视图 View 中。

同样的，对于每个视图 View 都存在与其对应的入口文件，假设此 View 为 `page` ，那么

- `page.json` ：此视图 View 的配置。
- `page.js` : 此视图的脚本逻辑。
- `page.wxss` : 此视图的样式。

其他的，不论是 JavaScript 脚本还是 Wxml 模板和 Wxss 样式，应被入口文件 `require/import` 使用。

构建方法

在小程序开发者工具的 Sources 面板，查看 JavaScript 脚本，会发现：项目中所有的 JavaScript 都会被 **同步加载** ，不管是否被 `require` 。

每个脚本都会被套上如下代码：

```
define("some.js", function(require, module){
    // 原本的代码
});
```

这种加载方式类似AMD，但是跟标准的AMD又有些不同，缺少了依赖部分的声明。

而对于 Wxml 和 Wxss 文件，则被开发者工具自动转换为 JavaScript 后加载，其中：

- Wxss ：主要处理的是 import 逻辑，然后生成的 Css，通过脚本的形式插入页面使用。
- Wxml ：和 React Naitive 的 JSX 类似，被编译成 createElement 类似的形式。

预览发布方式

预览、调试、打包上传都集成在微信提供的开发者工具中，而发布体验版和发布线上则在微信小程序的管理后台中。打包上传、发布的逻辑，都需要人工操作。

关于开发者工具的一些细节

关于小程序的打包、预览、上传的流程，都包含在开发者工具中，所以想要更深入得了解小程序的流程机制，必须要从开发者工具入手。

经过简单研究，发现开发者工具是基于 `NW.js` 构建的，基于 Node 和 Webkit 构建的应用程序。对于前端来说，这是一件令人感到幸福的事，直接可以通过读源码来了解它的逻辑。

代码在哪里？ 在 MacOS 系统中，源码比较好找：右键开发者工具『Show Contents』（显示包内容），就能在 `Resources/app.nw/` 下找到相应的源码，完成路径如下：`/Applications/wechatwebdevtools.app/Contents/Resources/app.nw/` 。

源码都是压缩过后的 JavaScript 脚本，可以使用 js-beautity 进行格式化，以便于阅读。

```
// 在源码目录的 app 目录下执行
find . -type f -name '*.js' -exec js-beautify -r -s 2 -p -f '{}'
\;
```

一些技巧：

- 在资源目录下：app/dist/app.js 的第 37 行 window.addEventListener("resize", function() {}) 之前，加入 nw.Window.get().showDevTools();。之后每次打开微信开发者工具时，会自动启动针对『开发者工具』的开发者工具，并可以通过它调试微信的开发者工具。
- 在打印日志时，不要用 console.log，请使用 global.contentWindow.console.log。这样，才能输出到上面所说的开发者工具的开发者工具的控制台里。（NW.js 的 Node JS Context 和 Webkit JS Context 是分开的，JavaScript 脚本运行在 Node 的 JS Context 中，因此，打印其实打印在 Node 的输出中，并不在 Webkit 的开发者工具的控制台中。global.contentWindow 获取的是 Webkit 的 JS Context 里的 Window）

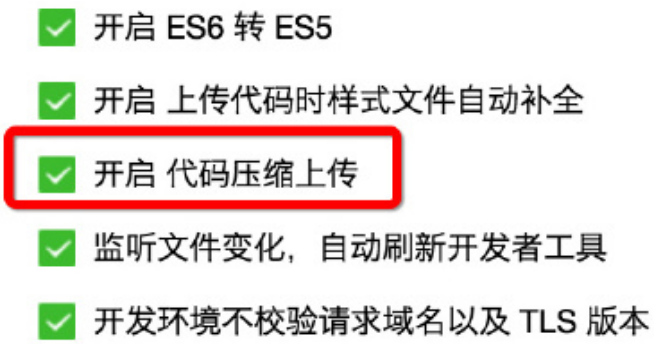
使用这两点技巧，读者们可以优雅地去阅读微信开发者工具的源码了。

工程流程化方案

上面，我们简单了解了小程序的一些背景知识和架构模式，下面，GitChator 将从 **问题 -> 解决方案** 的方式，来说明这套工程流程化方案。

Size 问题 -> 压缩工具

还是从老问题压缩说起吧。虽然微信对 Size 的限制从 1024 变成了 2048，但是终有一天，代码会增长到超过 2048，而且，**Size 的大小会影响用户加载的速度，包括下载最新版本代码的速度和小程序初始化的速度**，所以压缩是一直有必要的。



虽然开发者工具，已经支持了代码压缩上传，但是 GitChator 觉得它是个『假的压缩选项』。因为在阅读开发者工具的源码逻辑之后，发现它的压缩，只是将 JavaScript 用 Uglify 进行混淆压缩。而对 Wxml 、 Wxss 没有进行任何压缩处理。同时，对资源路径中的无用文件也没有做处理。因此，我们要做的有关压缩的事情还是很多的。例如：

合并 JavaScript 并压缩

将所有的 JavaScript 脚本使用合并成一个文件，这样会使 **脚本压缩效率变高**（例如 `require` 的长路径没了）、**混淆性越大**（代码的目录结构没了）。虽然 JavaScript 脚本被包装成类 AMD 的形式，但是使用时是 **同步加载同步执行** 的，因此将多个 JavaScript 脚本合并成一个并不影响小程序加载或切换视图的效率，反而因为减少了 IO 次数，提升了加载效率。（现在 GitChator 暂时使用的是 WebPack，而使用 Rollup 会让代码更小，执行效率更快）可是要如何做呢？

首先是，入口文件选取：正向上面代码结构所说，小程序有一个统一的入口是 `app.js` ，而每个视图 View 都有自己的入口 `page.js` ，将这些所有入口 `require` 到一个统一的入口文件中，进行打包，这样会得到一个拥有 JavaScript 脚本逻辑的 `bundle.js` 。

其次，要对入口文件进行一定的修改，让代码被合并成一个文件后，仍能正常运行。

对于视图 View 的入口，必须存在 `Page(PageOption)` ，这样的视图注册逻辑，我们只需通过正则或者 AST 将其改为自定义的注册（Register）方法 `global.__p('/path/to/page.js', pageIndex, PageOption)` 即可。

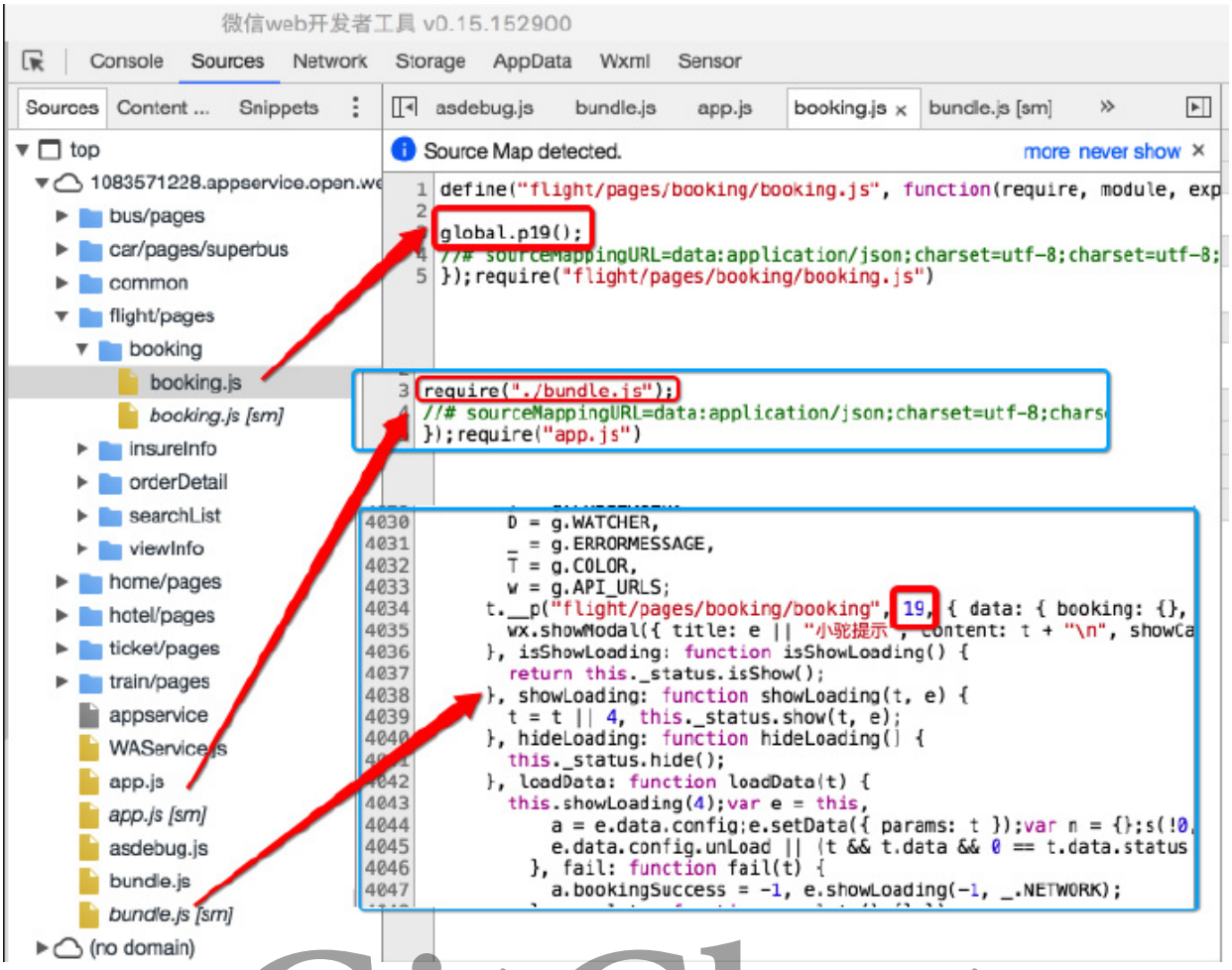
实现如下：

```
global.__p = pageName, index, pageOpt) => {
    // 其他相关逻辑
    global['p' + index] = () => {
        Page(pageOpt || {});
    };
};
```

这样在视图 View 的入口文件中，只需要 `global.pXX()` 一段代码即可。

对于小程序的入口 `app.js` ，将原本的内容直接改为 `require('bundle.js')` 即可，因为它并不像视图入口和代码路径有关。

最终，形式如下：



压缩其他文件

其他文件主要包括 Wxml 文件、 Wxss 文件和 JSON 文件。平时前端开发中，对 HTML 和 CSS 的压缩，主要是 **去除的空白字符、换行** 以及 **删除注释。**

对于 Wxml 直接两个正则即可：

- `/<!--((.|\n|\r)*?)-->/gm` ：去除注释。
- `/\"\n\s*/g` : 去除换行。

对与 Wxss，直接用 `uglifycss` 即可；对于 JSON，直接 `JSON.stringify(JSON.parse(...))`。

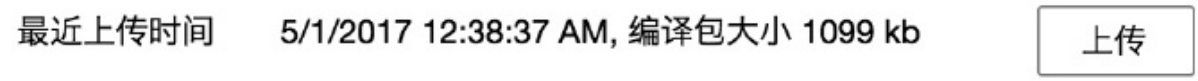
这里，有些读者会可能提出两个疑问：

- 1、空白字符、换行能有多少，减不了多少吧？
- 2、开发者工具为什么不做对这些文件的压缩？

关于第一个问题，一个约 1000 KB，空白字符和换行大概有 10KB。在有上限的情况之下，10 KB 也是要珍惜的。

关于第二个问题，GitChator 认为微信开发者工具的开发者觉得没有必要去做。上文中提过，Wxml 和 Wxss 都会被转义成 JavaScript 脚本，在此过程中，不管 Wxml 和 Wxss 是否

被压缩，它们的转化结果是相同的。因此，压缩与否，对于最终产物是没有影响的（最终产物指在服务器二次打包后的结果，也是用户真正使用的）。但是，Size 是以本地打包上传的内容进行计算的，不进行此步压缩，会使微信服务端判定的 Size 增大。



删除无用文件

删除无用的 JavaScript 文件（因为已经打包为 `bundle.js` 了，无需其他非入口文件了），删除没被 `import` 的 Wxss 和 Wxml 文件，删除空目录等等。

代码级优化

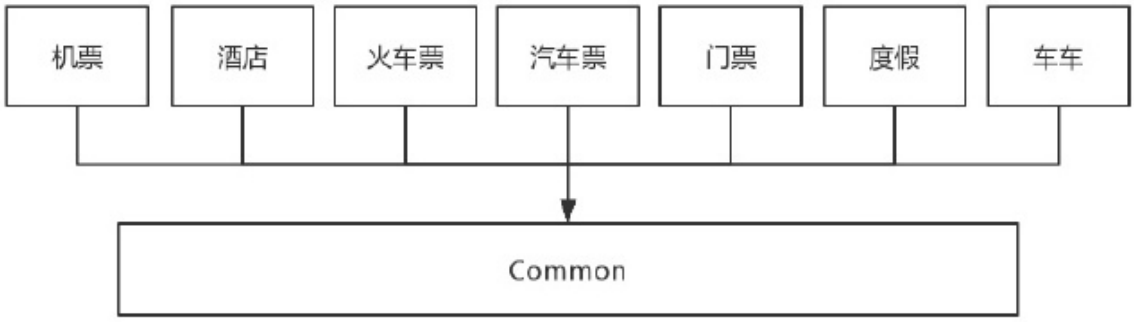
除了利用工具进行压缩，在编写代码时，也可以通过一些方法来减小体积，在这里简单列几点：

- 提炼公共组件、公共 API。
- 使用 ES6 时，尽量不使用依赖 `Runtime/Polyfill` 的语法，例如 `import` 和 `class`。
- 图标使用 Iconfont。
- 等等…

多业务并行开发问题 -> 以模块化的方式进行业务代码隔离

随着小程序的诸多限制放开，越来越多的产品想来分此一杯羹，导致像去哪儿网这样多业务的公司，一个小程序里承接的业务也会逐渐增多，开发人员的数量也会直线上升（一个小程序的开发者最多 30 人）。而此时，如果所有业务的开发者还在同一个项目里开发的话，那么这个项目将会非常难以管理。因此，**以模块化的方式进行业务代码隔离**势在必行。

对于模块的划分，很显然应该 **按照业务来划分** ，每个业务自己是单独的模块，并且 **相互之间不能存在依赖关系**， **能并且只能依赖公共模块**。就如下图所示。

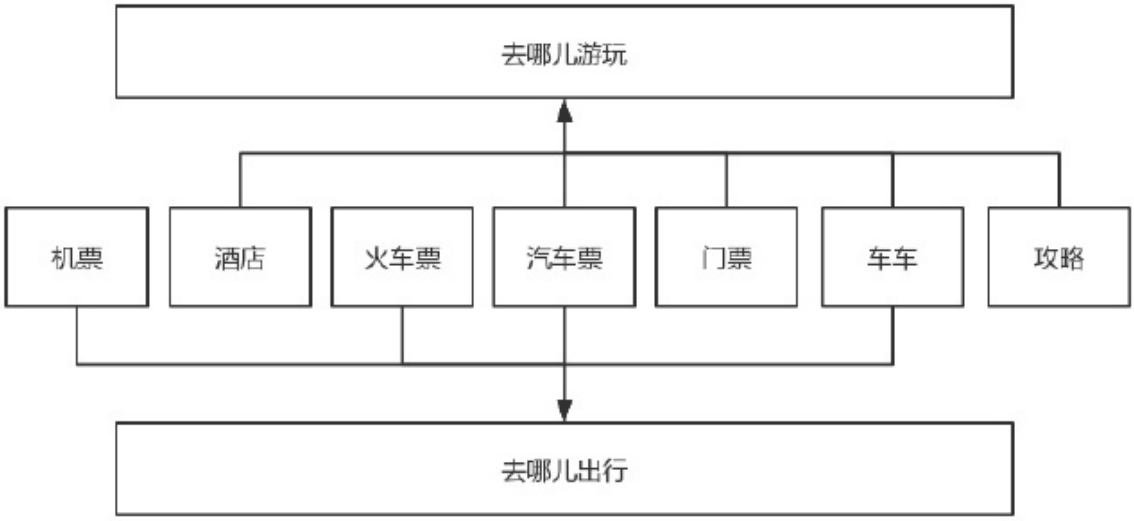


这样实现带来最大的好处是 **可插拔**，也就是可以随时将任意一个模块在不影响其他业务的情况下进行修改或删除。

上文中提出过一个问题，**将去哪儿旅行 App 上的主要业务的主要流程都放进微信小程序内，真的适合小程序的场景吗？**。这是问题虽然是一个纯产品上的问题，但是 **可插拔** 的特性，会让这个问题 **变得容易解决**，产品同学可以很容易地以业务模块为单位增减业务。假设一个业务取得的收益很小，产品同学可以立即在不影响其他业务的前提下，将这个业务下线，并将其的空间用于接入新业务或分配给其他线上业务。

同时，业务在开发时，本地只需要存在 Common 模块就能运行，降低了开发和调试的成本。

可插拔的模块化不仅仅可以让模块变更的成本降低，同时也让 **以相同模块构建多个小程序** 的成本降低，增加小程序产品的 **灵活性**，根据场景构建出最适合的 **即开即用** 的小程序。



诸如上图所示，用不同的模块构建出不同场景的小程序。

- 去哪儿出行：交通类产品，包括去哪儿所有的交通类业务模块。
- 去哪儿游玩：场景是本地游，因此包括与本地游相关的业务模块。

当然，这只是举例，具体情况视产品场景而定，但是，这种构建方式，GitChator 个人觉得比较适合小程序这种灵活的场景。而大而全的功能，还是交给 App 吧。

这里就出现另一个问题：**如何有效的管理模块。**

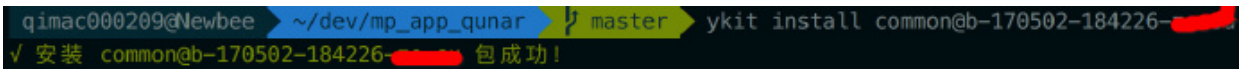
因为一个模块，被多个小程序项目引用，所以它的版本化是非常重要的，否则有可能出现，为了满足一个小程序项目的需求进行改动后，在另一个小程序内出现问题。

因此，GitChator 基于公司通用的前端开发工具集 YKit 实现插件，配合 公司的代码仓库 GitLab 和 公司的打包平台 Jenkins，实现了一套模块版本的管理逻辑。

本地版本查询:

```
qimac000209@Newbee ~/dev/mp_app_qunar master ykit list
Δ 本地模块列表如下:
> bus : 0.1.0
> car : 0.1.2
> common : b-170502-184226-[redacted]
> flight : 0.1.0
> home_qunar : 0.1.5
> hotel : 0.1.2
> ticket : 0.1.3
> train : 0.1.3
> vacation : 0.1.2
```

本地安装：

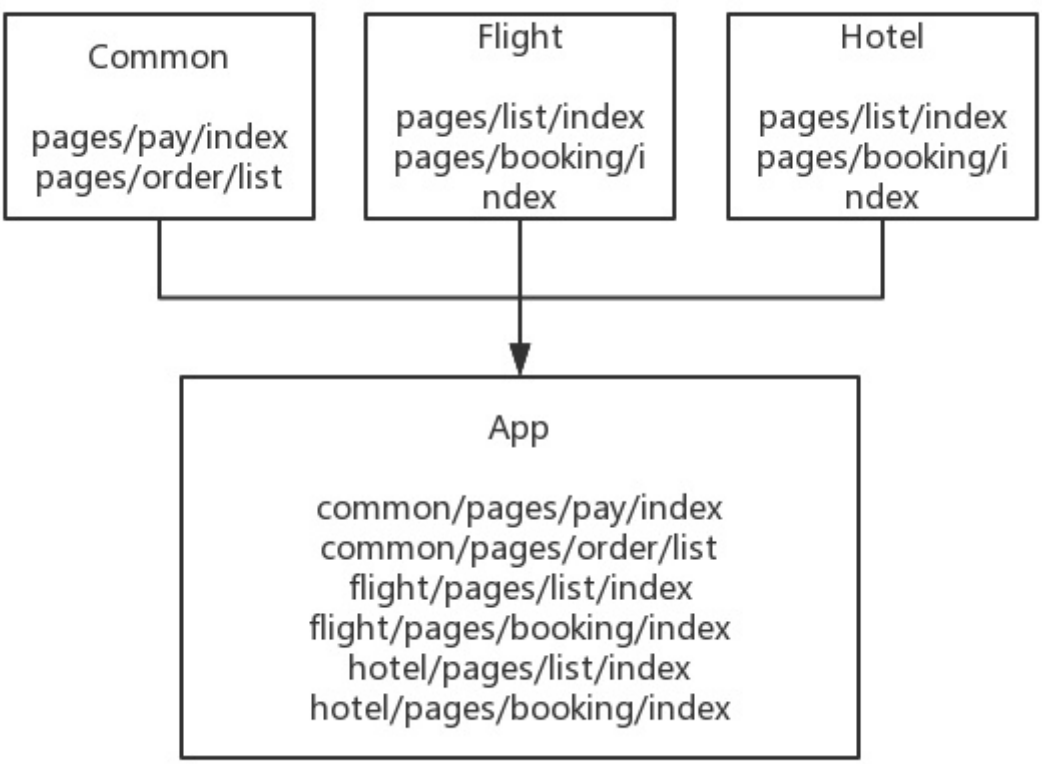


查询页面：

当然，也可以用私有 NPM 仓库实现模块管理，GitChator 主要是为了和公司发布系统打通，才选择这样实现。

在实现模块化的过程中，有两个地方需要注意：

路由：业务模块已经隔离，而路由配置在统一的入口处，这样会影响模块的插拔。因此，在每个模块的根目录下，添加各自的 `app.json` 配置，用于配置模块自己的路由，然后在工具打包的时候，将各模块的路由进行合并 Merge。



App.onLaunch: 业务模块有些逻辑，需要在 App 的 onLaunch 时执行。因此，在每个模块的根目录下，添加各自的 `app.js` 脚本，完善相应的，然后在工具打包时，将脚本注入到相应的位置。


```
module.exports = () => {
    onLaunch: () => {
        // TODO Something
    }
};
```

最后，可插拔的模块化对于 **模块 Size 的计算** 也是非常重要的。

```
模块的 Size = 小程序总 Size - 拔去此模块时小程序的 Szie
```

这样的计算方式，是十分可靠的，有利于小程序对每个模块进行配额分配。

人工发布问题 -> 自动化发布系统集成

一般比较成熟的公司，都会有自己的发布系统，用于项目的发布，在降低人工成本的同时，规避人工操作的风险。而微信小程序后台并没有提供相应的接口 API 进行此类操作，因此，需要自己实现一套上传发布逻辑。

这个逻辑主要有两点：**小程序打包** 和 **各种扫码验证**。

对于小程序打包，直接从微信开发者工具内，将小程序打包的逻辑抽离出来即可。

```
// wxBuild 为开发者工具资源下 app/dist/weapp/commit/build.js
// wxPack 为开发者工具资源下 app/dist/weapp/commit/pack.js
wxBuild({
    projectpath: 'path/to/source', // 小程序项目路径
    es6: true,
    postcss: true,
    minified: true
}, {
    noCompile: true
}, (err, data) => {
    wxPack(data, 'path/to/result.wx', (err, data) => {
        // data 为产出物的路径
    });
});
```

对于扫码验证，其实是模拟用户的行为。Gitchator 实现的逻辑是，将二维码打印在发布日志的 Job 里，管理员使用微信进行扫码即可。下面是发布日志的截图。（主要用到 phantom）。

由于篇幅有限，具体的细节可以私下和 GitChator 交流。不过 GitChator 觉得，微信的同学，应该会在今后提供相应的上传发布接口，自动化发布，其实是大多数公司的刚需。

打包发布完成，发给项目相关人，一封统计性邮件，也是非常有必要的。

【微信小程序】小程序体验版发布提醒 b-170419-161904-[redacted]

小程序管家
Wednesday, 19 April 2017 at 4:24 PM
To: [redacted]
Cc: [redacted]

去哪儿旅行（YMFE）(工程地址)

版本	标签	环境参数		用量(kb)	结余(kb)	配额(kb)
v1.1.9	b-170419-161904-[redacted]	prod		1092.85	955.15	2048
体验版二维码		模块	版本	-	-	-
		common	0.1.4	260.51	19.49	280
		home_qunar	0.1.4			
		bus	0.1.0	108.94	91.06	200
		car	0.1.0	73.3	106.70	180
		flight	0.1.0	127.1	92.90	220
		hotel	0.1.2	131.54	88.46	220
		ticket	0.1.2	128.67	71.33	200
		train	0.1.2	158.65	81.35	240
		vacation	0.1.1	104.16	75.84	180
		查看更多版本信息				

总结

除去上面描述的几点，我们还做了诸如 **环境参数配置**、**项目参数注入**、**无埋点统计** 等通用逻辑，来解决并完善开发中所遇到的问题。

总结起来，去哪儿的小程序工程流程化主要包括三部分：**本地工具**、**模块仓库** 以及 **发布系统关联**。

- 本地工具：包括项目初始化、打包压缩、模块编译、环境参数配置、项目参数注入、无埋点统计等等功能，主要解决开发过程中所遇到的问题和痛点。
- 模块仓库：模块化方案的基础，管理模块的版本。保证一个小程序使用多个模块和一个模块被多个小程序引用时的正确性。
- 发布系统关联：避免人工操作的繁琐和易错，自动化完成打包、上传、发布等流程。

结语

这篇文章，GitChator 从微信小程序的问题背景和架构模式出发，到优雅地使用工具，结合着发布系统，进行低成本、低风险、灵活度高的开发等各个方面，对去哪儿网在小程序的工程流程化上的探索与实践进行的阐述，希望对各位有一定的帮助。

写在最后

对于文中所提到的工具，GitChator 将在尽快将其公开。因为工具逻辑中，有一些和公司业务、发布系统耦合得比较紧密，所以需要一定的时间进行完善。大家可以关注 GitChat 所在团队的博客 ymfe.tech 和 GitHub YMFE 关注最新进度。