【前端工程化】为什么大厂都要开发自己的脚手架?

每个初入大厂的前端同学，在真正投入开发之前的第一件事可能就是熟悉公司的脚手架，从那一刻起，你就抛弃了熟悉的cra、vue-cli、vite等，成为了合格的大厂人（误）

大厂是不是为了冲绩效所以才会开发脚手架呢？每当新技术出现，作为热爱技术的前端人我们难道就不想折腾折腾让自己的项目“现代”起来?

我们可以从预开发环节 ⇒ 开发环节 ⇒ 构建环节 ⇒ 预提交环节 ⇒ 发布环节，看看脚手架到底做了些什么。

本篇会做一些代码的示意，但不会真正的写一个脚手架出来。

预开发环节

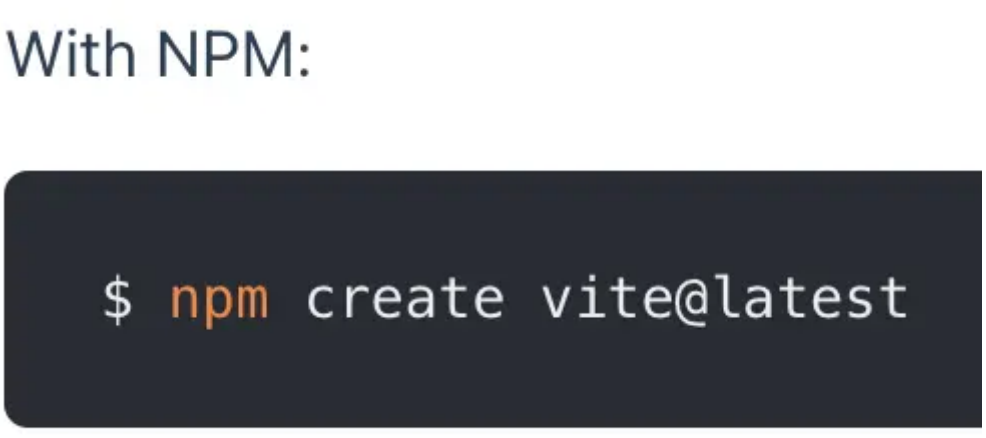
简化命令与配置

在没有脚手架的时候，我们是怎么配置一个项目的呢？我们需要在项目中创建wekpack.config.js / rollup.config.js文件，基于开发和生产环境做各自的配置。

如果用到了一些别的工具，如esbuild等，又得增加一个别的配置文件，而像eslint、prettier、jest这些也都有各自的配置文件。而具体使用的时候，各个工具又各自需要各自的命令。

每次都记这一堆命令和配置对开发者来说是一件非常麻烦的事情，这也是脚手架对开发者来说最大的意义，它能够很大程度上简化我们需要的命令和配置。

而我们是怎么使用脚手架的呢？拿vite为例，我们可以通过简单的命令就创建出一个vite的项目。



输入命令行，进行一些选择就可以创建一个项目，不需要你去写一大堆的配置文件，直接`npm run dev`就能跑起来；同时，直接`npm run build`就可以打包成生产环境的代码。如果是大厂内部的

框架，甚至还可以集成大厂内部的部署环境，跑一个 `npm run preview` 或者 `npm run deploy` 就可以预览 / 部署。

如果我们要实现这样的效果，首先我们需要在脚手架项目中预定义一些模板（类似react、react-ts、vue、vue-ts等），其次，我们要能在命令行中运行bin来进行项目创建，可以看看vite的源码，在create-vite的package.json中

```
"bin": {
    "create-vite": "index.js",
    "cva": "index.js"
}
```

bin指定create-vite路径，而在index.js中则是主要做了命令行参数读取、拉取最新模板的操作，有兴趣的同学可以自己阅读一下。

同时，脚手架往往提供了自己的config文件，比如vite就提供了vite.config来统一配置内部所使用到的工具。这要怎么实现呢？简单来说就是在vite的bin中读取了vite.config文件内容，从而构建出vite config对象，再根据这个vite config对象生成对应的rollup config对象然后调用rollup。

相应vite可以参考github.com/vitejs/vite··· 中resolveConfig这一段

```
export async function resolveConfig(
  inlineConfig: InlineConfig,
  command: 'build' | 'serve',
  defaultMode = 'development'
): Promise<ResolvedConfig> {
  let config = inlineConfig
  let configFileDependencies: string[] = []
  let mode = inlineConfig.mode || defaultMode

  // some dependencies e.g. @vue/compiler-* relies on NODE_ENV for getting
  // production-specific behavior, so set it here even though we haven't
  // resolve the final mode yet
  if (mode === 'production') {
    process.env.NODE_ENV = 'production'
  }
  ...
```

【以反战为名，百万周下载量 node-ipc 包作者进行供应链投毒】

开始有人以为只是个恶作剧，但事情并非如此简单。有开发者在对代码进行测试处理后发现，node-ipc 包的作者 RIAEvangelist 在投毒。他起初提交的是一段恶意攻击代码：如果主机的 IP 地址来自俄罗斯或白俄罗斯，该代码将对其文件进行攻击，将文件全部替换成 ❤。该作者是个反战人士，还特意新建了一个 peacenotwar 仓库来

想象一下如果你是个毛子这个时候应该有多绝望哈哈，公司有自己的脚手架可以限定死版本，减少某个包的作者一时兴起或者被攻击者入侵发布了有问题的新版本，而我们又在这时新起项目或者升级包导致业务挂掉的情况。

这里说的是减少，因为总有库不是被包括在脚手架里的，这也是lock文件的存在意义。

开发环节

提供模板

脚手架可提供多套模板供用户选择，类似纯js、纯ts、vue3+ts。同时，对于一些网页需要的通用配置元素，如favicon，title，preconnect link可以通过config的方式提供给用户配置。

也可以先提供monorepo壳子的模板，在其中再进行项目的初始化。

约定式路由

在项目中，可以约定一个名为pages的文件夹，pages下的文件夹中的index默认为一个个page。在pages中如果给出404、500这样的index，也可以方便地做一些做页面错误catch的处理；并且可以在脚手架中默认集成动态引入组件。

在需要做单页面维度的权限校验的场景下，这时又分两种提示方式：第一种是不该进的路由用户无法进入，第二种是可以进入，但是给一个申请权限的弱提示。如果是想要第一种提示方式的话，约定式路由需要进入页面后校验没有权限再跳出，体验上会稍差一些。

集合微前端

现在的single-spa、qiankun等微前端框架都非常流行，脚手架可以提供一套有微前端框架的B端模板，直接用于给开发者生成。如果是在大公司内部，往往对这些微前端框架本身也有一定的封装，形成公司自己的微前端框架，但对于脚手架来说，做的事情还是一样的，就是把微前端框架的部分封装起来。在开发者使用的时候，开发者甚至不需要知道页面是通过微前端的方式来加载的，就用普通写组件的方式来写就好。

提供可插拔的功能插件（权限、埋点、sentry等

脚手架可提供一些插件，供开发者选择是否接入（插件和上part提到的feature flag各自的侧重点是：通过feature flag提供的功能更多与构建相关，而插件更多为开发以及一些附加平台功能）。像是埋点、sentry都比较简单好理解，我们这里来聊一下权限。

大厂尤其是内部系统很多都是使用单点登录进行登录的，而如果访问接口没有权限一般会是401的错误码，所以其实可以在权限插件中做统一的无权限的登录跳转。

权限插件还可以做再细致的权限校验，我们可以结合menu封装，对每一个页面进行业务逻辑上的可见的设置。

网络请求层面封装

脚手架一般会根据当前的运行环境进行网络请求方面的配置，比如dev、test、pre-release、production等，因为不同环境的请求baseURL、超时时间会不同。

同时，脚手架还可以提供基于接口定义生成接口定义代码的功能，通过后端提供的swagger/thrift文件直接生成接口请求代码。

prettier、eslint

每个大厂都会有自己对prettier和eslint的要求，将prettier和eslint的配置可以统一收口在脚手架中，形成一套代码风格。

构建环节

构建打包支持

脚手架往往提供开箱即用的构建打包工具支持，对于前端常见的文件类型提供了默认配置，一般包括：

1. ts、js
2. css / less / sass / stylus / postcss / css module / tailwind支持，有的脚手架甚至提供了默认的换肤配置
3. 图片、svg（包括导入为src或组件直接渲染）
4. wasm等等

避免重复造轮子

这里的重复造轮子更多指的是性能优化层面，首先脚手架一般都有持续迭代的同学，当新技术出来时（比如vite、esbuild、swc等），可以先进行一波升级并推出beta版本或者提供出一个feature flag，减少真正产出业务的同学重复踩坑的可能。

这里提供一个swc的feature flag供大家参考：

```
/**
 * config sample
 * swc: { jsc: { parser: { syntax: "typescript", tsx: true } } }
 */
if (options?.swc) {
  return {
    test: /\.(t|j)sx?$/,
    use: {
      loader: "swc-loader",
      options:
        typeof options.swc === "object"
          ? options.swc
```

```
          : transformTsConfigToSwcConfig(),
    },
    exclude: /node_modules/,
  };
} else {
  return {
    test: /\.(t|j)sx?$/,
    use: {
      loader: "babel-loader",
    },
    exclude: /node_modules/,
  };
}
```

预提交环节

commit hook

脚手架中可以集成git commit之前触发的钩子，钩子中主要可以做：

1. commit message的check，例如项目是monorepo结构，那commit message应该是feat/fix/chore……(project name): xxx改动，可以采用commitlint - Lint commit messages
2. 代码质量的初步校验，eslint、prettier的强校验

这里不建议在commit hook中做过于重的校验，因为commit时希望hook能够较快执行完，让用户还是可以尽快提交。更全的校验还是放在CI阶段执行，如unit test等。

发布环节

push结合CI

在预提交环节，我们提到了用commit hook做初步校验，但是hook总是有方法绕过的，所以在CI时还是要做double-check。

一般脚手架会提供一个ci yml文件来灵活的控制构建，在push代码可以进行CI，一旦CI有问题后续的合入或者CD都不可再继续。

发正式版本可以在本地用脚手架publish，也可以在画面CI时直接提供打正式包的选项。