# 1、cli原理

这种工具最初诞生的目的都是为了创建一个模版项目

# 2、大体的流程：

##### 1. 要有一个模版项目

这个比较容易理解，跑 `cli` 最终会生成一个初始化的项目，这其实是定义好的模版项目，社区方案，有些是写在 `cli` 内部的，`vue-cli` 的方式是把模版发布到 `github` 这类平台，然后通过 `download-git-repo` 去远程拉取

[vue-templates](https://github.com/vuejs-templates)

##### 2. 给用户一个友好的交互

一般就是一些互动式的交互，询问一些配置项，然后最终会根据这些配置项去选择对应的模版，并做一些相应的改动

##### 3. 输出到目标目录，安装依赖，运行起来

这个就不多说了

> 而这，单单只是一个脚手架工具，起了一个初始化模版项目而已

现在的 `vue-cli` 是提供完整工具链的工具，几乎包含了所有基于 `vue` 开发涉及的功能

这种方式也是社区目前推崇的，同类型的，比如 `create-react-app`, `angular-cli` 等

### 想清楚这件事儿

现在，我们不谈架构和具体实现，单纯看看这件事儿，大致确定我们的目标（产品需求）

1、肯定还是要初始化一个项目模版的，但是模版要多元化，可配置，可丰富（生态）

2、模版项目里不应该再有乱七八糟的 `webpack`, `rollup`, `babel` 等配置，不需要用户去承担维护，说白了，给用户一个黑盒，用就行（高大上的说法 -- 开箱即用）

3、用户还是有特定的场景需要去配置，比如加个 `webpack` 的 `loader` 处理某些文件等，这种时候不需要去 `eject` （不是黑 `create-react-app`）

4、总结一下，就是要做到用户在使用体感上「傻瓜式」、「零成本」、「易拓展」、「可定制」

我们给上边的要求转换成「程序员语言」

> 不一定是 `vue-cli` 的实现套路，所以，假设我们的 `cli` 工具叫 `蛋蛋`

1、`cli` 命令加个参数，比如这样

```shell

dd create -template=default

dd create -template=typescript

```

然后就通过这个 `template` 拿到参数，然后动态的拼一个远程地址

```js

const remoteRepoUrl = `https://dd.com/lib/template/dd-templates/${template}`;

```

然后，通过去拉代码下来...

> 当然这里顺带就把所有的这种交互式的配置项一把梭封装了，会玩儿的还可以给整成可视化的（`vue-cli` 已支持）

2、模版里面不再保留 `webpack` 这类配置文件，统一抽一个 `service` 模块出来，通过这个 `service` 模块来启动、调试、打包、测试等

这其实也不新鲜，对面的 `react` 不就这么搞的么，所以，这样我们生成的项目，一看 `package.json` 中的命令，就这样了

```js

{

"start": "dd-service start",

"build": "dd-service build",

"eject": "dd-service what fuck?",

}

```

这样的好处是啥，抛开高端用词，本质就是用户不需要去维护这种业务之外的东西，不然还搞 `webpack`，很闹心的，这就交给专门维护 `service` 模块的同学了，对用户来说工作仅仅就是根据需要升级这个包的版本了

但是！

这样也造就了一个问题，这对用户来说，底层的配置项就是一个黑盒了，总有那么三三两两的需求，我确实需要去增加/修改一些配置，肿么办？肿么办？

难道，你要我 `eject`? （真不是黑）

当当当当！（脑补音效）

插件机制啊，还记得我上节课，啥课来着，给大家讲了插件机制吗？

3、好，设计一个插件机制

插件机制现在是运用广泛的一项基础技术，在我们身边的技术栈随处可见

`vue-cli`, `umi`, `babel`, `webpack`, `rollup`...

继续，还是尝试把复杂的东西说简单一点，什么是插件机制？

> 本质上，就是对外暴露出可定制化拓展内部能力的机制

还是装大象的问题，我们也抛开架构和具体实现，康康怎么回事儿？

1、对外，对外，说明外部可注册插件

2、对内，插件是对能力的拓展和升级

3、定制化，说明用户是可以随心所欲的进行能力拓展，但是有边界，边界在于内部暴力出多少能力给外边去操作

![一顿操作](https://p1-jj.byteimg.com/tos-cn-i-t2oaga2asx/gold-user-assets/2018/12/16/167b74a45d1e5f29~tplv-t2oaga2asx-watermark.awebp)

「程序员语言」

1、注册机制

五花八门，挑个好用、你喜欢的就行～

> 还是强调一下，不一定是 `vue-cli`的实现，所以我们的 `cli` 工具还是叫蛋蛋～

```json

// 比如，用户可以提供一个配置文件 dd.config.js

{

"ddPlugins": {

"service": [

// your service

],

"presets": [

// your presets

]

}

}

```

用户可以通过这种方式去引入、并使用某些能力，比较复杂的最好还是分个类，做到物以类聚（单一）

2、提供一套写插件机制的套路

一般来说，插件都是有自己固定写法的，每家厂牌的风格不一样，比如 `webpack` 的插件和 `babel` 的插件，写法就完全不一样

> 还是强调一下，不一定是 `vue-cli`的实现，所以我们的 `cli` 工具还是叫蛋蛋～

```js

// dd.clean.plugin.js

module.exports = (api, options) => {

// just do it! 鸿星尔克

}

```

我们就这样，然后在 `api` 上做文章，说白了就是在这个上边挂一些能力，给外界进行拓展，比如想自定义一个命令 `clean`

```js

const rimraf = require('rimraf');

module.exports = (api, options) => {

api.registerCommand('clean', (...args) => {

rimraf('./dist', (err) => {

if (err) {

console.error(err);

}

})

})

}

```

最后，控制边界，其实就是「约定大于配置」哀莫大于心死～

> api 暴露多少，就只能玩儿多少东西，边界是可控的

剧终～