如何搭建高性能脚手架

前端平台三组　席坤

得物

脚手架脚手架的作用、核心价值

脚手架的作用

- 开发脚手架核心目标：提升前端研发效能

脚手架的核心价值

- 自动化：项目重复代码拷贝/git操作/发布上线操作
- 标准化：创建项目/git flow/发布流程/回滚流程
- 数据化：研发过程系统化、数据化、使得研发过程可量化

脚手架开发难点解析

- 分包：将复杂的系统分开成若干个模块
- 命令注册：du add、du create
- 参数解析
  - *options*全称：*—version*、*—help*
  - *options*简写：*—V*、*—H*
  - 带*params*的*options*：*—path xx*
- 帮助文档

脚手架开发难点解析

- 比如还有很多：
- 命令交互行
- 日志打印
- 命令行文字颜色
- 网络通信：HTTP/WebSocket
- 文件处理
- 等等

原生脚手架开发痛点分析

- 痛点一：重复操作
  - 多package本地link
  - 多package依赖安装
  - 多package单元测试
  - 多package代码提测
  - 多package代码发布
- 痛点二：版本一致性
  - 发布时版本一致性
  - 发布时相互依赖版本升级
- package 越多，管理越复杂

脚手架本地link标准流程

链接本地脚手架

```
cd your-cli-dir
npm link
```

链接本地库文件

```
cd your-lib-dir
npm link
cd your-cli-dir
npm link your-lib
```

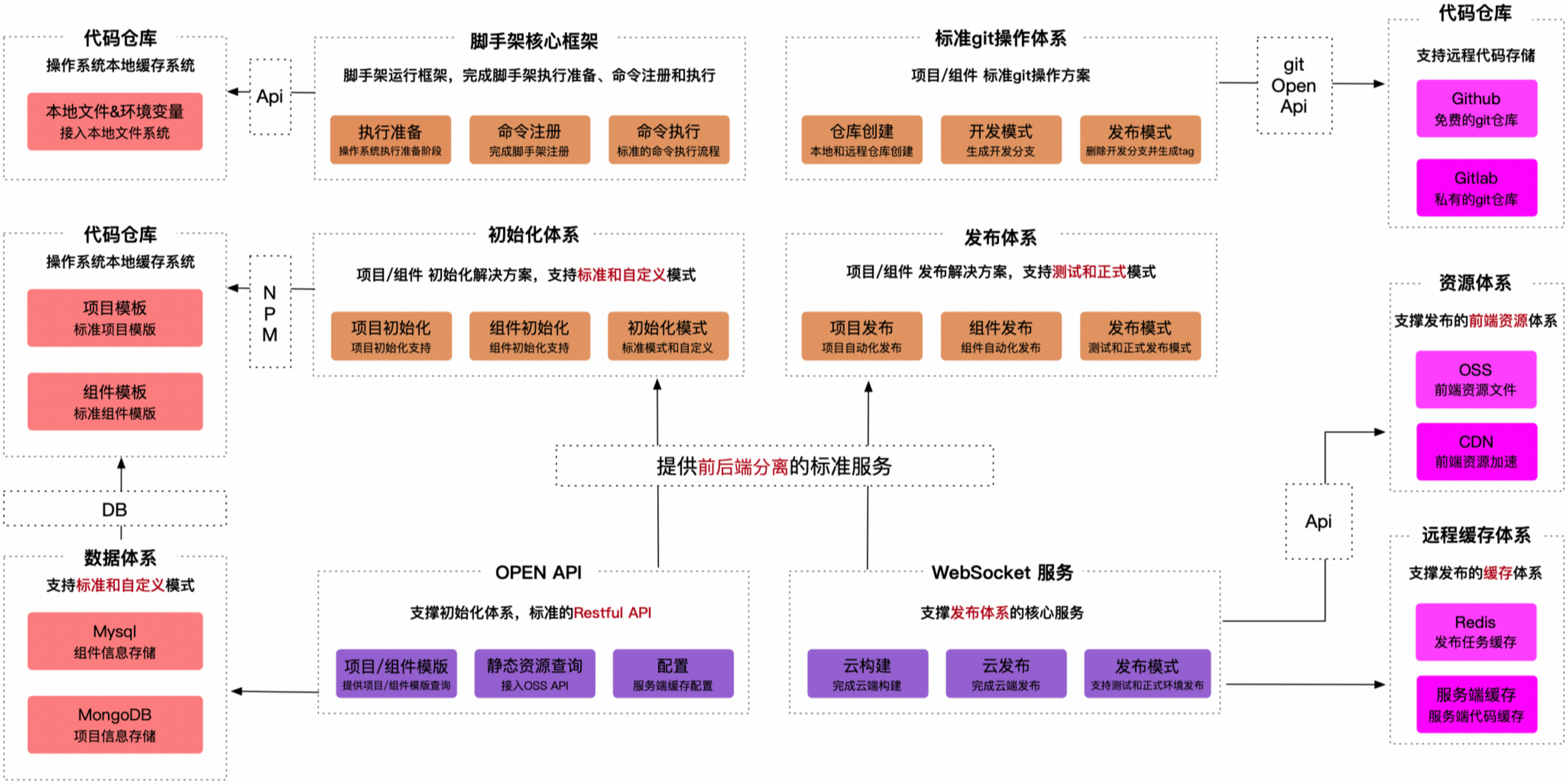
取消链接本地库文件

```
cd your-lib-dir
npm unlink
cd your-cli-dir
# link存在
npm unlink your-lib
# link不存在
rm -rf node_modules
npm install -S your-lib
```

Lerna简介

Learn 是一个优化基于 git + npm 的多 pageage 项目管理的工具，
像 bable、vue-cli、create-react-app 都使用Learn进行管理

脚手架架构图



脚手架拆包策略

核心流程：core

命令:commands

初始化

发布

清除缓存

模型层:models

Command命令

Project项目

Component组件

Npm模块

git仓库

支撑模块:utils

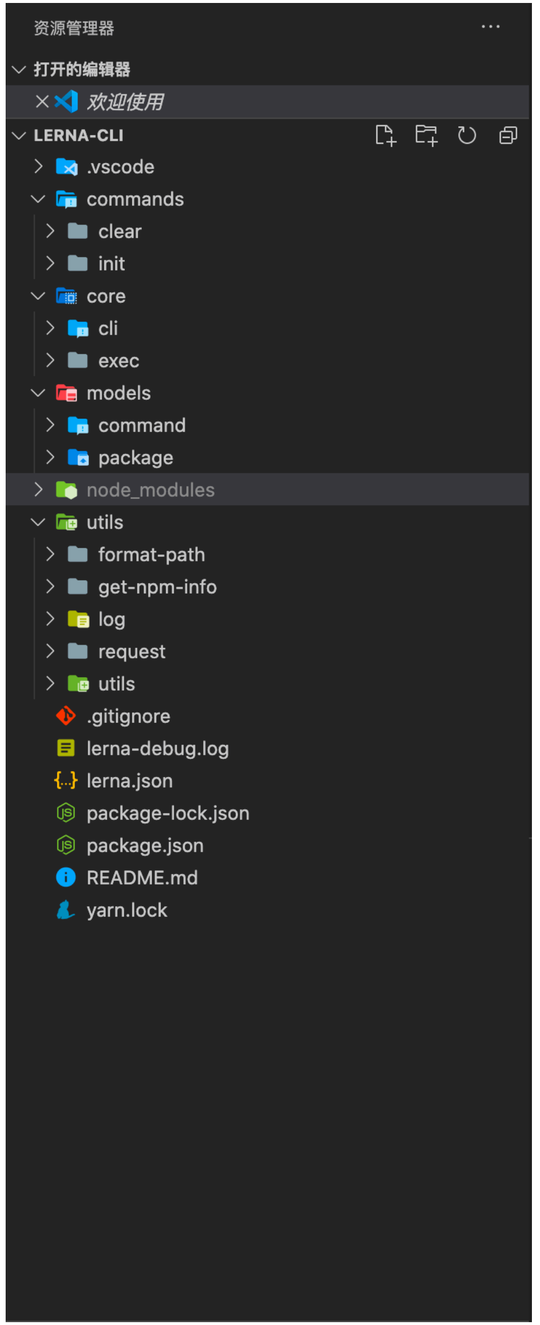
Git操作

云构建

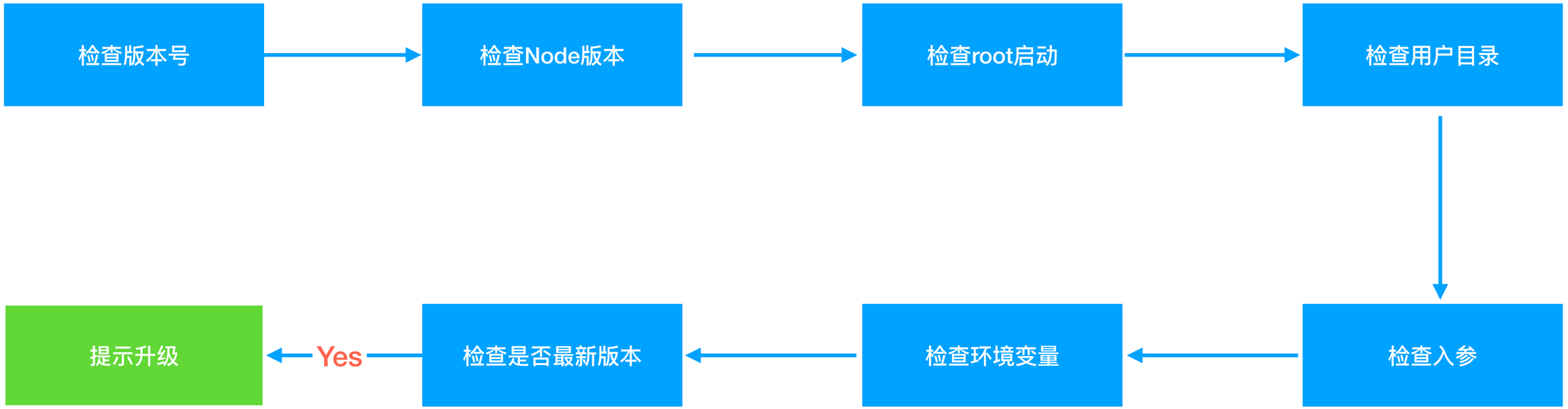
工具方法

API请求

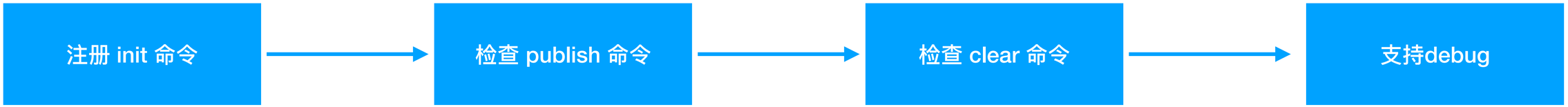
Git API



准备执行阶段



命令注册流程



Commads高级定制

自定义Help信息

```
const program = new commander.Command();

program.helpInformation = function () {
    return ''
}
```

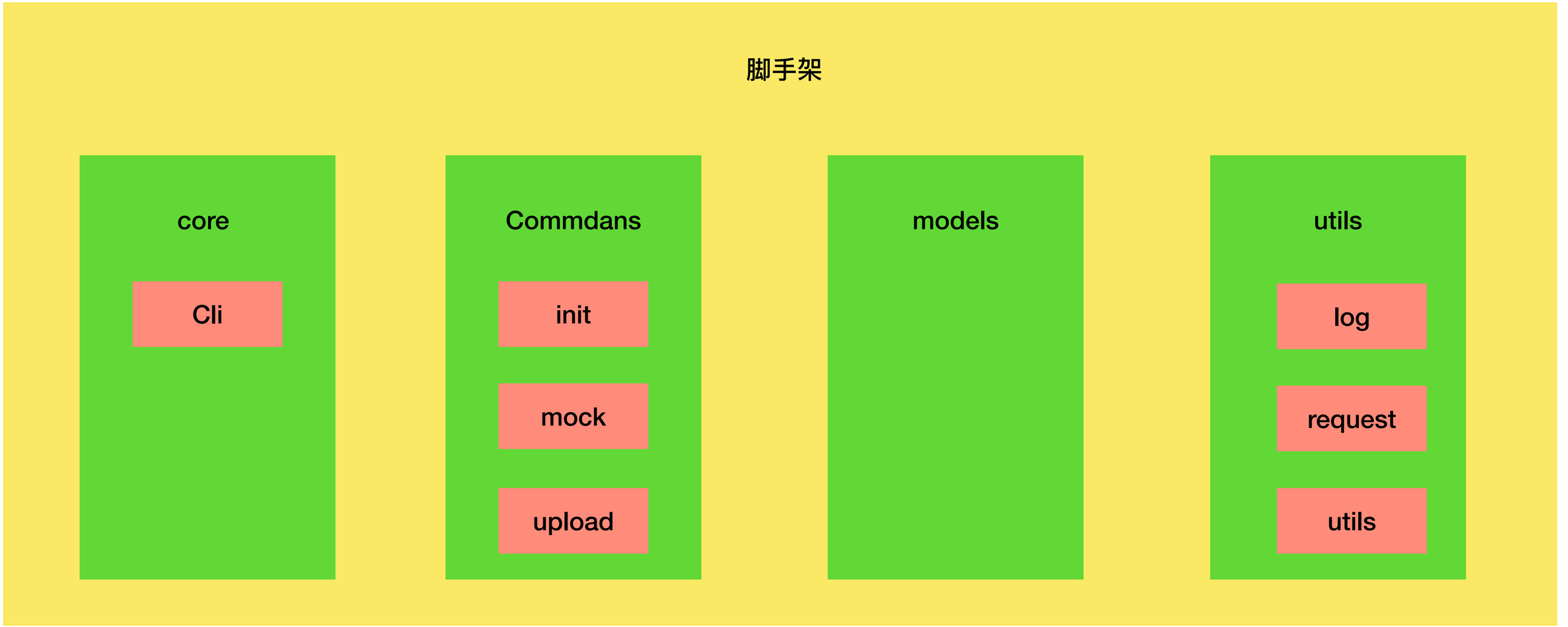
实现debug模式

```
program.on('option:debug', function () {
    if (program.debug) {
        process.env.LOG_LEVEL = 'verbose';
    }
    console.log(process.env.LOG_LEVEL);
})
```

对未知命令进行监控

```
program.on('command:*', function (obj) {
    console.error('未知的命令' + obj[0]);
    const allCommands = program.commands.map(cmd => cmd.name())
    console.log(allCommands)
})
```

痛点分析



痛点分析

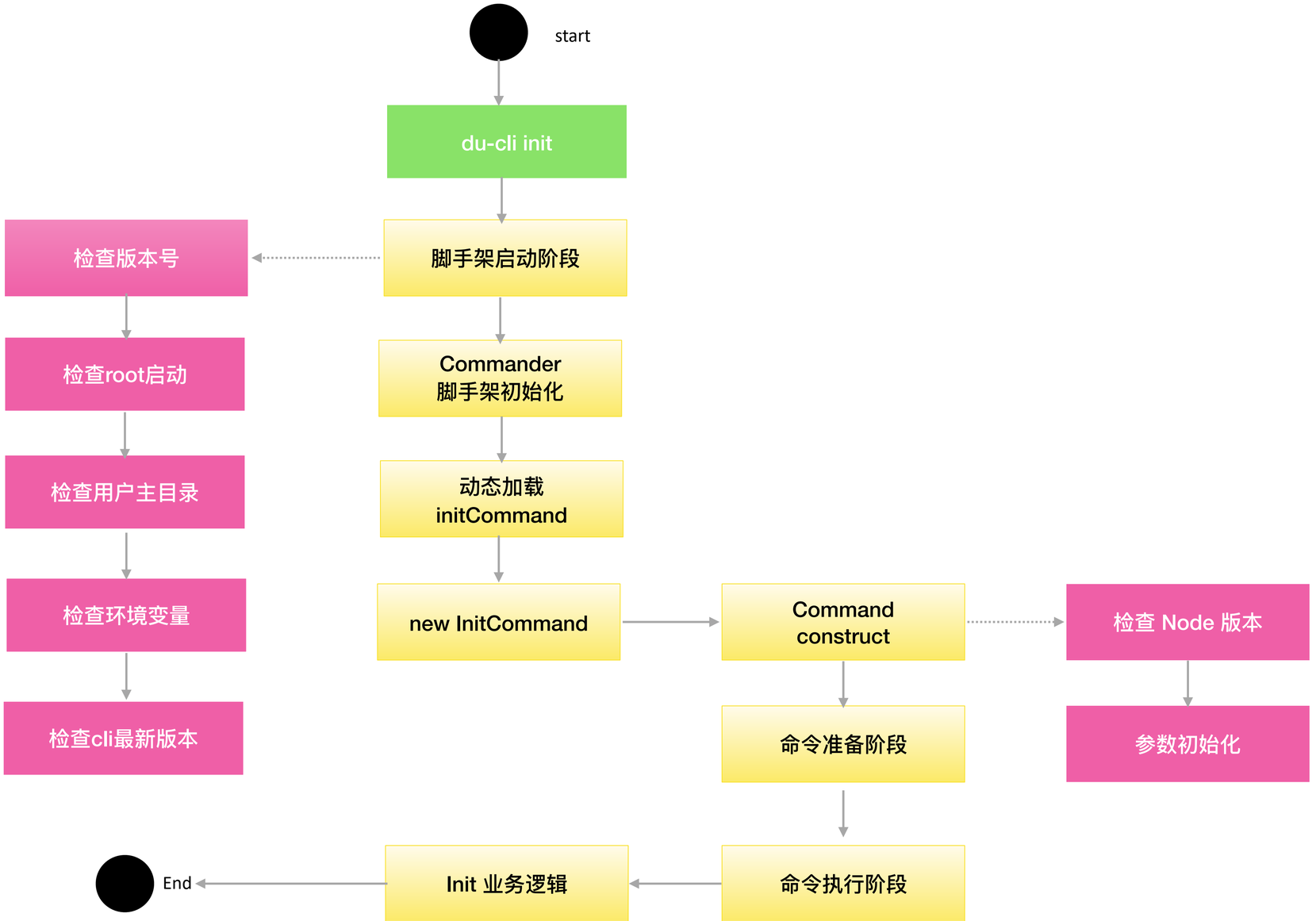
1. Cli 安装速度慢、所有 package 都集中在 cli 里，因此当命令过多时，会减慢 cli 速度

2. 灵活行差：init 命令只能使用 @aotu-cli/init 包，对于集团而言，每个部门的 init 可能各不相同，需要实现 init 命令动态化

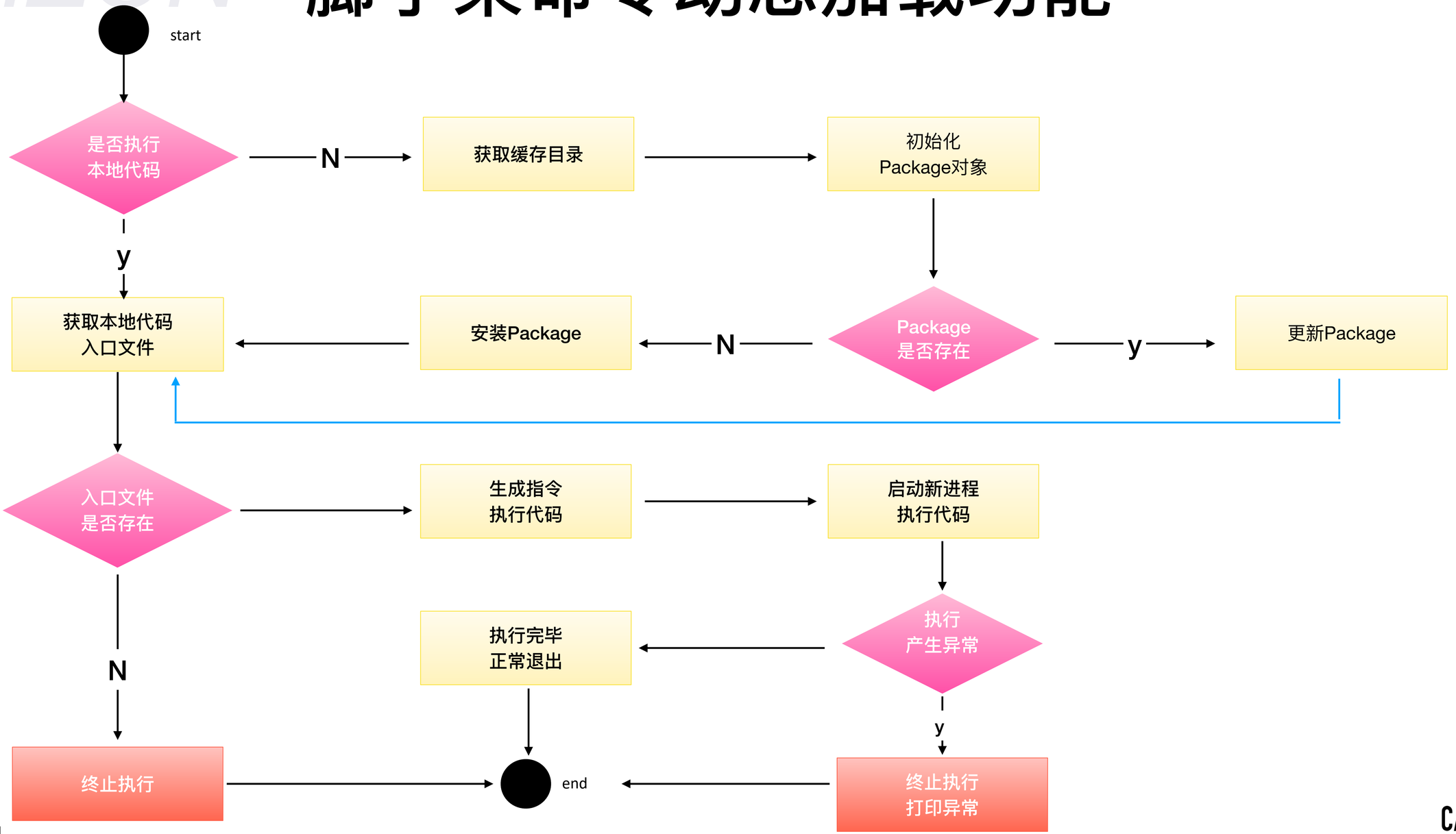
- 团队A使用 @du/init 作为初始化模块
- 团队B使用 @du/my-init 作为初始化模块
- 团队B使用 @du/your-init 作为初始化模块

这时候要求我们动态加载 init 命令，这将增加架构的复杂度、但是大大提升脚手架的可扩展性，将脚手架和业务逻辑解藕

脚手架架构优化



脚手架命令动态加载功能



Package

- 可以封装一个 Package 类提供 install、updat、getRootFilePath 获取文件入口等方法

- 利用 npminstall 库进行动态安装、指定到规定的缓存目录

```
class Package {
    constructor(options) {
        // Package的目标路径
        this.targetPath = options.targetPath;
        // 缓存Package的路径
        this.storeDir = options.storeDir;
        // Package的name
        this.packageName = options.packageName;
        // Package的version
        this.packageVersion = options.packageVersion;
        // package的缓存目录前缀
        this.cacheFilePathPrefix = this.packageName.
replace("/", "_");
    }

    // 判断当前Package是否存在
    async exists() { }

    // 安装Package
    async install() {
        await this.prepare();
        return npminstall({
            root: this.targetPath,
            storePath: this.storeDir,
            registry: getDefaultRegistry(),
            pkgs: [
                {
                    name: this.packageName,
                    version: this.packageVersion,
                },
            ],
        });
    }

    async update() { }

    // 获取入口文件的路径
    getRootFilePath() { }
}

module.exports = Package;
```

如何生成指令

- Node 默认可以加载 .js / .json / .node
- 如果不是这三种、Node默认会当作 .js 来执行
- Node 默认执行方法 Node xx.js
- 也可以通过 Node -e “string” （字符串）

Node多进程child_process

异步

- exec
- execFile
- fork
- spawn

同步

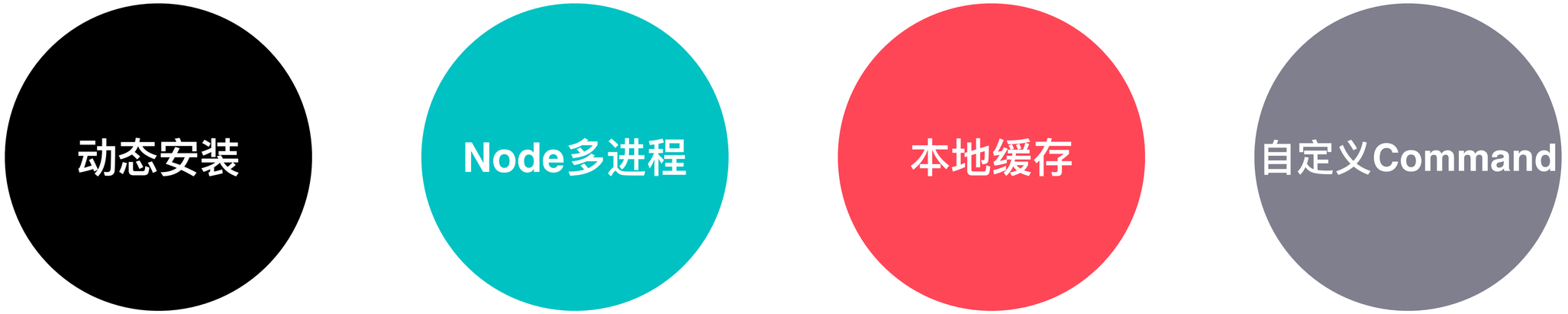
- execSync
- execFileSync
- spawnSync

动态执行

```
// 在node子进程中调用
const code = `require('${rootFile}').call(null, ${JSON.stringify(args)})`;
const child = spawn("node", ["-e", code], {
  cwd: process.cwd(),
  stdio: "inherit",
});
child.on("error", (e) => {
  log.error(e.message);
  process.exit(1);
});
child.on("exit", (e) => {
  log.verbose("命令执行成功:" + e);
  process.exit(e);
});
```

操作演示

核心方案



脚手架创建功能架构

准备阶段

1.确保项目的安装环境
2.确认项目的安装信息

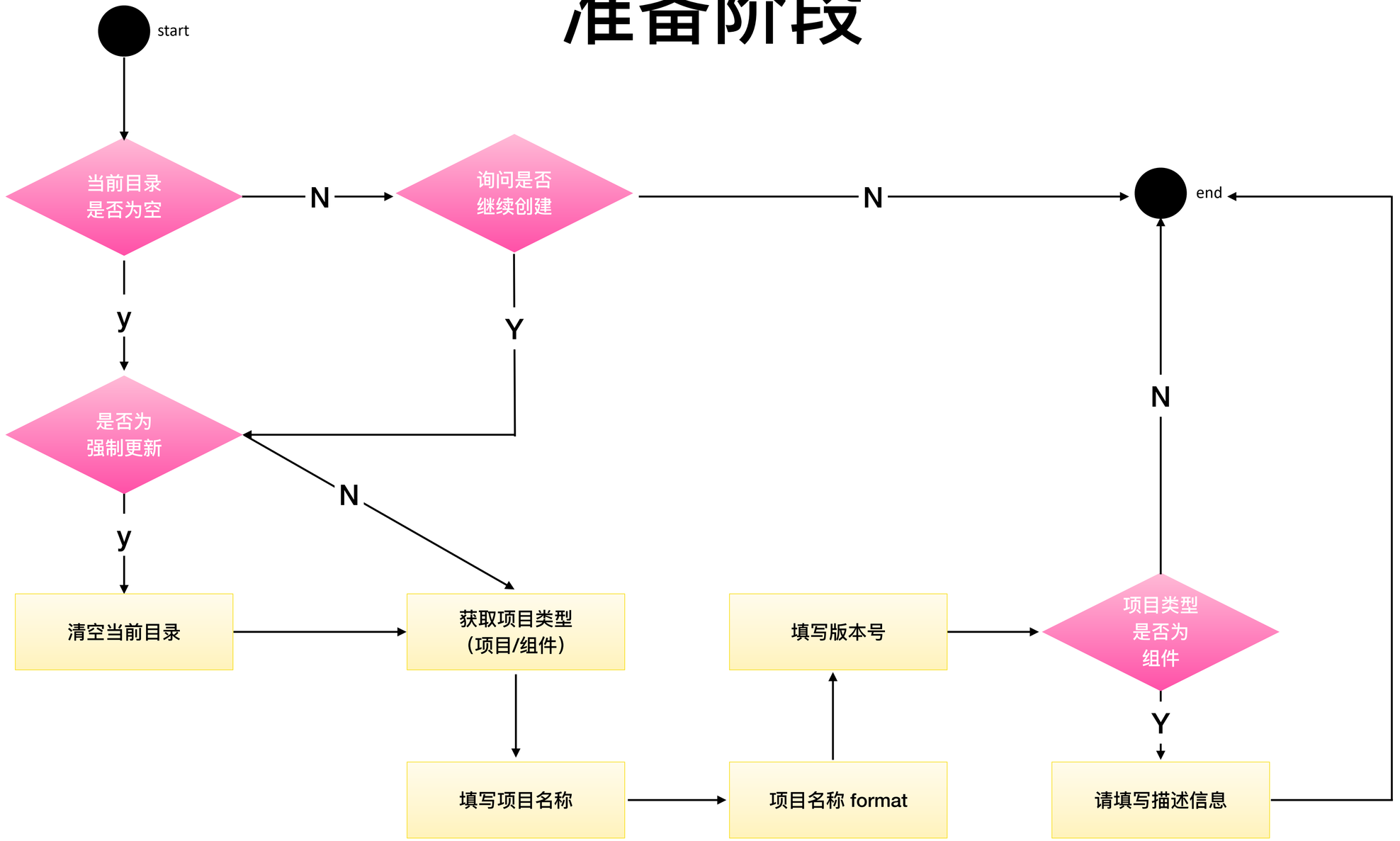
下载模版

利用前面封装好的 Package 类来实现

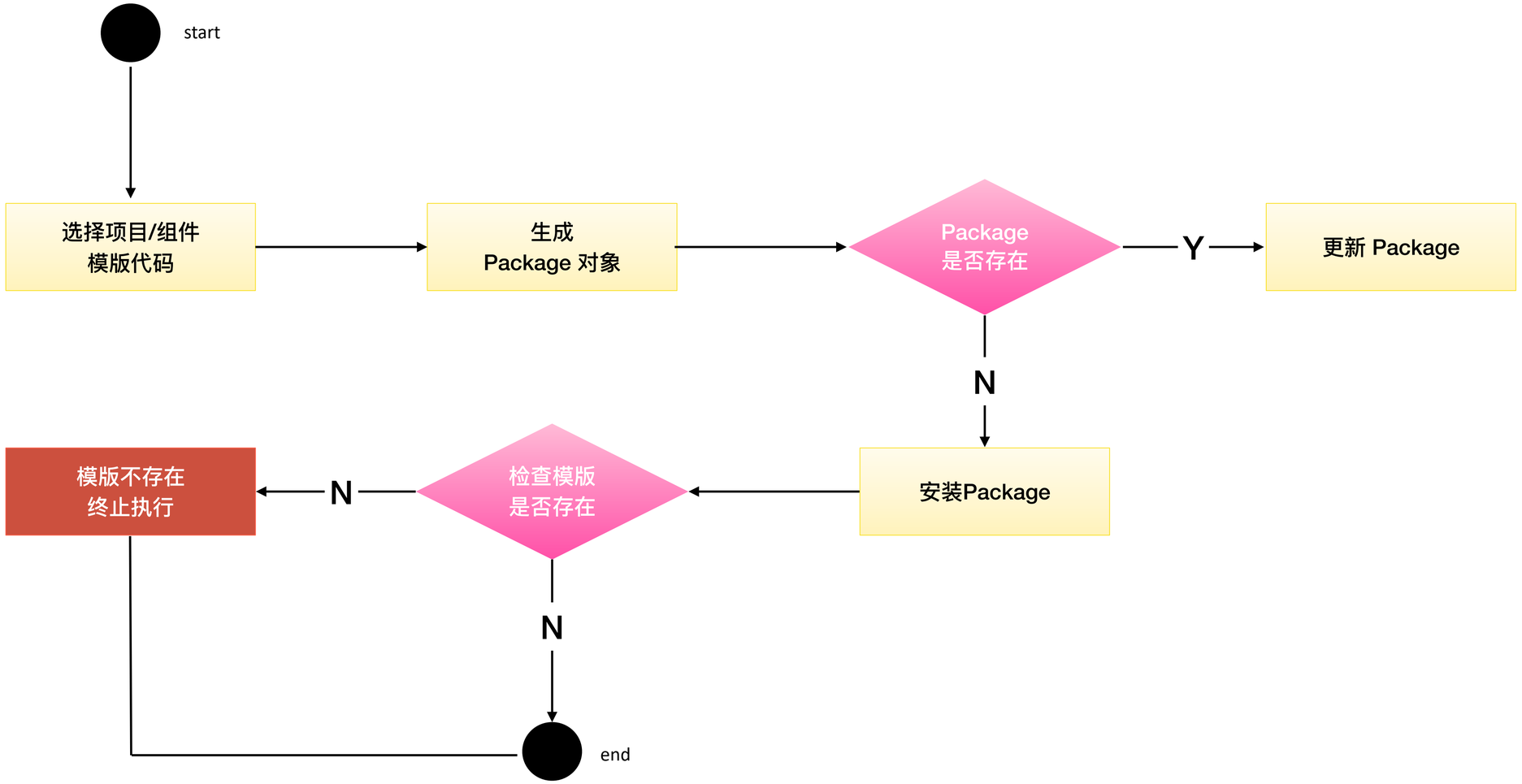
安装模版

1.标准模式，通过 ejs 实现模版渲染、并自动安装依赖并启动项目
2.自定义模式下，将允许用户主动去实现模版的安装过程和后续启动过程

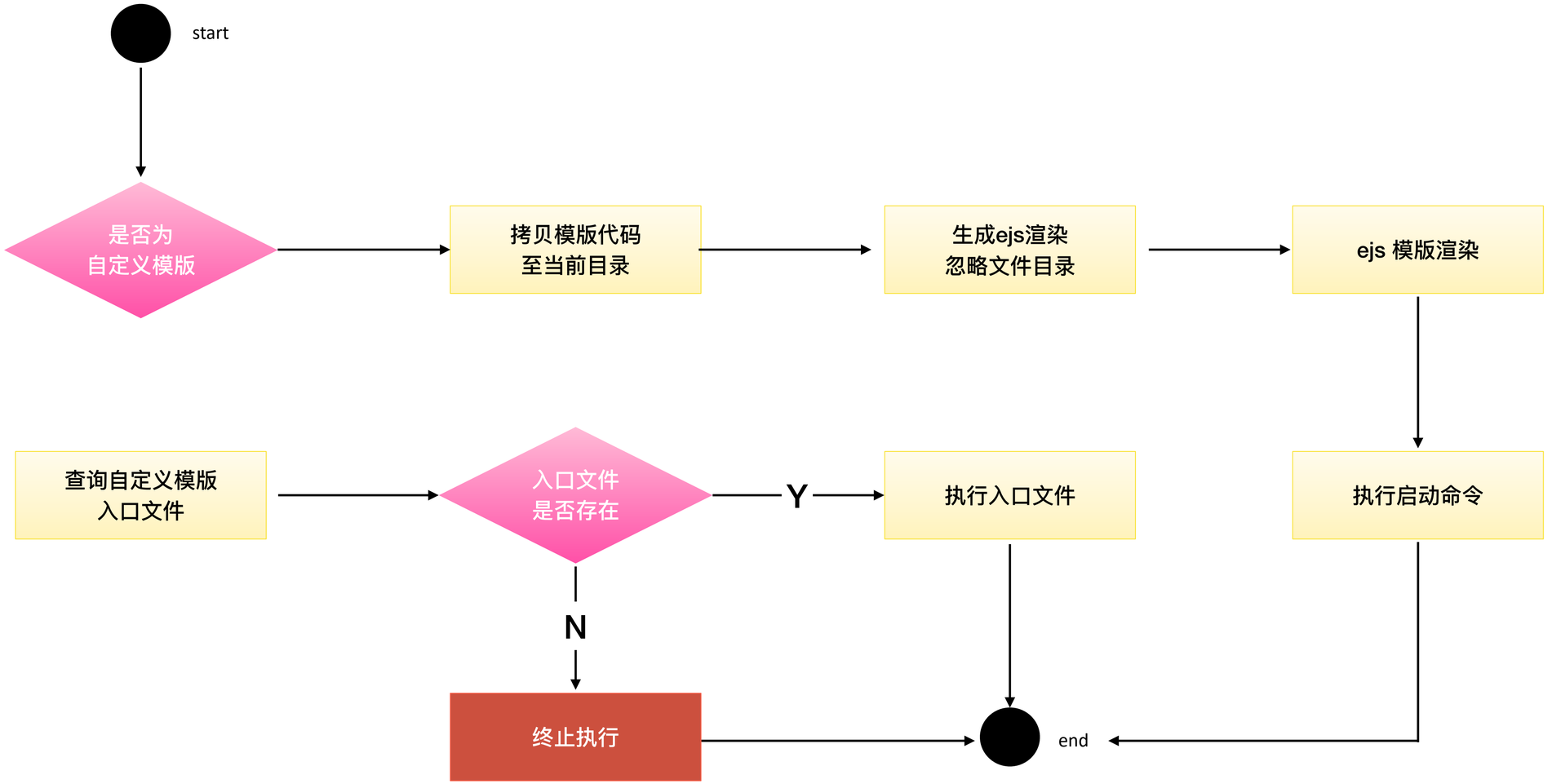
准备阶段



下载模版



安装模版



模版字段

- name
- npmName
- version
- type
- installCommand
- startCommand
- tag
- Ignore
- more…

ejs动态渲染

- 简单的模版标签 <% %>
- 自定义分隔符(例如<??>代替<% %>)
- 引入模版片段
- 同时支持服务端和浏览器JS环境
- 模版静态缓存
- 详见

总结

集成更多的工具

- Mock Cli
- Compression Cli
- Upload Cli
- More 。。。

思考🤔

- lerna有哪些缺点
- 流程还可以如何优化
- 从中学到了什么

谢谢