从零开始开发 npm 包

准备

1. node 和 npm(>= 7) 环境
2. 一款你喜欢的编辑器
3. git

开发

一个规范的 npm 包必须符合以下要求：

1. 一个包必须是一个单独的目录；
2. 根目录下必须有 package.json 文件；
3. package.json 文件中必须包含`name`和`version`两个属性，分别代表包的名字和版本号；
4. `main`/`module`字段：表示包的入口。当根目录下有 index.js 文件时可以不必指定，默认采用该文件，否则必须指定；
5. 其他字段大部分是非必填的，极少部分在某些场景是可选的，有些场景必填，对于必填的情况后面会逐步接触；
6. 代码文件。

开始

1. 新建一个文件夹，比如 chaos，作为包的根目录。
2. 进入该文件夹，执行以下命令：

```
npm init
```

该命令会初始化一个 package.json 文件，打开这个文件，可以看到已经初始化了一些字段，包括上面提到的`name`和`version`以及`main`。这个包现在还是空的，下面我们给他加上代码

添加代码

在包的根目录新建文件 index.js，并填入一些内容，然后将这些内容导出，这时候这个包就可以用了：

```
const bigBang = () => {
  return "天地玄黄，宇宙洪荒。";
}

module.exports = bigBang;
```

添加依赖

npm 包的依赖添加方式与我们平常做的项目基本一致，但是有一些特殊情况。

peerDependencies

同等依赖，或者叫同伴依赖，用于指定当前包（也就是你写的包）兼容的宿主版本。用于解决插件与所依赖包不一致的问题。假设我们要开发一个 react 的包 packageA，packageA 理所当然需要依赖于 react。同时我们又需要保证 packageA 与它的宿主项目所依赖的 react 的版本一致，这种情况下我们就需要用到 peerDependencies：

```
{
  "name": "packageA",
  "peerDependencies": {
    "react": "^16.0"
  }
}
```

上面的例子中，我们期望宿主项目的 react 版本是 16。这种写法看起来与直接放在 dependencies 里一样，但是它们的作用是不同的。以上面的场景为例，如果 packageA 把对 react 的依赖写进 dependencies 里，那么如果宿主项目依赖的 react 版本与 packageA 不一致（比如 react18），npm 可能会安装两个不同版本的 react。如果放在 peerDependencies 里，则 npm 会报错。显然第二种情况是我们期望的，因为我们不希望一个项目中出现两个不同版本的 react。

与 peerDependencies 字段配合的还有一个 peerDependenciesMeta 字段。当用户安装你的包时，如果 peerDependencies 中指定的包尚未安装，npm 将发出警告。peerDependenciesMeta 字段用于为 npm 提供有关如何使用对等依赖的更多信息。 具体来说，它允许对等依赖标记为可选：

```
{
  "name": "packageA",
  "peerDependencies": {
    "soy-milk": "1.2",
    "react": "^16.0"
  },
  "peerDependenciesMeta": {
    "soy-milk": {
      "optional": true
    }
  }
}
```

关于 peerDependencies 的官方描述可以在这里找到。

optionalDependencies

这个字段表示依赖是可选的。比如说我们的包引入了一个跟 debug 相关的库，但是因为某些原因(这里先不用管是什么原因)，这个库可能会找不到或者无法安装。这种情况下我们当然希望安装能继续下去，因为 debug 并不影响我们的实际功能。这时就可以使用 optionalDependencies，它的规则与 dependencies 一样，但是优先级比 dependencies 高。另外需要注意的是我们需要自己处理找不到依赖的情形：

```
try {
  var debug = require("debug");
} catch {
  debug = console;
}
debug.log("some message");
```

关于 optionalDependencies 的官方描述可以在这里找到。

本地测试

包开发好后我们要先在本地验证是否可用，就需要做本地测试。现在我们有了开发项目@fh/chaos，还需要一个项目来引入它，这里称为 demo。要在 demo 中引入 chaos，主要有几种方案可供选择，这里选几种不需要第三方工具的进行讲解。

通过本地路径

从 2.0.0 版本开始，npm 支持本地路径。本地路径可以使用 `npm install -S` 或 `npm install --save` 保存，支持下面的任意一种形式：

```
../foo/bar
~/foo/bar
./foo/bar
/foo/bar
```

这种情况下它们会被规范化为相对路径(大概意思是如果你输入的是绝对路径，会被 npm 修改为相对路径)并添加到 package.json 中：

```
{
  "name": "baz",
  "dependencies": {
    "@fh/chaos": "file:../chaos"
  }
}
```

需要注意的是，这种情况下在 demo 中运行`npm install`时，chaos 中的依赖不会自动安装，我们需要在 chaos 中手动安装。

通过 npm link 命令

`npm link`命令的作用是将模块创建成本地依赖包，它的详细文档见官网，使用方式如下：

在我们项目根目录下，执行指令

```
npm link
```

进入 demo 项目根目录，执行指令

```
npm link @fh/chaos
```

修改 package.json，在 dependencies 中添加依赖(此时还不能用 npm install)：

```
{
  "dependencies": {
    "@fh/chaos": "^1.0.0"
  }
}
```

接下来就可以在 demo 中使用 chaos 中的 api 了。

测试没问题后，需要在 demo 的根目录下使用`npm unlink package-name`命令解除项目与本地 npm 包的关联：

```
npm unlink @fh/chaos
```

使用这种方式与上面的本地路径存在同样的问题，即 chaos 的依赖不会自动安装。同时因为 dependencies 中指向的是 npm 上的版本，而此时 npm 上并没有我们的包，运行 `npm install` 会报 404 错误。

workspace

前面两种方案虽然都能满足我们的要求，但是多少有些问题。我们秉承一个理念：**包作者需要像包的用户那样使用自己的包**。为了达到这个目标，在 npm 7 发布之前，我们需要借助第三方工具。从 7 开始 npm 原生支持 workspace，我们就可以利用这一特性来实现。Workspace 这一概念很常见，并非 npm 的原创。事实上就连在 node 生态落地也是 npm 迫于其他其他包管理器的竞争压力才实现的。

在 npm 中，workspace 指一组支持从单个顶级根包中管理本地文件系统中多个包的功能。这组功能优化了从本地文件系统处理链接包的更加简化的工作流程。它可以避免我们手动自动执行 `npm link` 命令，而是在 `npm install` 的时候自动将 workspaces 下面

的合法包添加到 node_modules 中。更多关于 workspace 的详细内容，可以查阅官方文档，这里演示如何利用它来解决我们的问题。

首先，我们需要新建一个 git 仓库并创建 package.json 文件：

```
git init fh-chaos && cd fh-chaos
npm init
```

添加 workspace：

```
npm init -w ./packages/foo
```

上面命令的意思是，新建一个项目（一个项目就是一个 workspace） foo，这个项目放在目录 packages 下。这时候我们的项目结构是这样的：

```
fh-chaos
|  package-lock.json
|  package.json
|
├─node_modules
|  |  .package-lock.json
|  |
|  └─foo
|           package.json
|
└─packages
    └─foo
            package.json
```

可以看到 npm 自动创建了缺失的文件夹，并且与之前创建项目的过程相比，文件夹下多了 node_modules 目录，并且该目录下面多了对 foo 的引用（快捷方式）！打开根目录下的 package.json 文件，我们发现里面有个 workspaces 字段，它的内容指向了我们刚才创建的项目 foo。关于这个字段的描述可以查阅官方文档。

现在我们知道了怎么使用命令来获得对 workspace 的支持，这些操作也可以手动进行。

首先，依然需要创建 git 仓库和 package.json 文件。然后打开 package.json 文件，在根节点下添加 workspaces 字段，里面的内容是各个包的路径，例如：

```
{ "workspaces": ["packages/*", "demo"] }
```

上面的格式只是我个人的习惯，你也可以将 demo 也放到 packages 目录下，或者将所有的包都放在根目录下，你喜欢就好，只需要确保路径是正确的。

接下来，将 demo 项目移动到根目录下，将 chaos 项目移动到 packages 目录下，现在项目结构是这样的：

```
fh-chaos
├─demo
│
│  .gitignore
│  package.json
│
└─packages
    └─chaos
```

现在你就可以进入到 demo 里面，然后像在其他地方一样执行`npm install @fh/chaos --save`命令并正常使用它了！

到目前为止一切顺利。但是，如果你的 demo 是使用一些现代工具（比如 vite）创建的，然后你像下面这样用，项目会报错: Uncaught SyntaxError: The requested module 'xxx/chaos/index.js' does not provide an export named 'default'

```
import bigBang from "@fh/chaos";
const world = bigBang();
```

在解决这个问题之前，我们首先需要对 JS 模块规范有一定的了解。

JS 模块规范

模块化产生的背景

远古时期，我们都使用 script 标签来引入 js，但当一个页面引入的 js 文件越来越多时，就产生了几个难以避免的问题：

1. 全局变量污染
2. 变量重名
3. js 之间的依赖关系复杂，无法保证顺序

为了解决这写问题，模块化规范应运而生。这里我们仅描述几种常见的模块规范，以及它们之间的区别。

CommonJS

CommonJS 是一个项目，其目标是为 JavaScript 在网页浏览器之外创建模块约定，它是随着 Node.js 一起出现的。相当长时间以来 JavaScript 都是没有模块化的概念的，直到 Node.js 的出现，把 JavaScript 带到服务端后，面对文件系统、网络、操作系统等等复杂的业务场景，模块化就变得不可或缺。由此可见 CommonJS 是服务于服务端的，虽然它的载体是前端语言 JavaScript，但是它不是前端。

在 CommonJS 中，一个文件就是一个模块，有自己的作用域。在一个文件里面定义的变量、函数、类，都是私有的，对其他文件不可见：

```
// a.js
var name = "张三";
var age = 20;
```

上面的代码中，a.js 是 Node.js 应用中的一个模块，里面申明的变量 name 和 age 是 a.js 私有的，其它文件都访问不到。CommonJS 规范还规定，每个模块内部有两个变量可以使用：require 和 module。require 用来加载某个模块，module 代表当前模块，是一个对象，保存了当前模块的信息。exports 是 module 上的一个属性，保存了当前模块要导出的接口或者变量，使用 require 加载的某个模块获取到的值就是那个模块使用 exports 导出的值：

```
// a.js
var name = "张三";
var age = 20;
module.exports.name = name;
module.exports.getAge = function () {
  return age;
};

//b.js
var a = require("a.js");
console.log(a.name); // '张三'
console.log(a.getAge()); // 20
```

到这里，我们就能知道前面我们在项目 chaos 中写的代码采用的是 CommonJS 规范。关于 CommonJS 模块就介绍到这里，更多详细内容可以查阅官网

AMD(Asynchronous Module Definition)

CommonJS 采用的是同步加载方式，而前端天然适合异步，于是有了 AMD(异步模块定义)。它是由加载器 RequireJS 定义的。

AMD 定义了两个函数 `define` 和 `require`，通过 `define` 定义模块，`require` 加载模块：

```
define(id?, dependencies?, factory);
require([module], callback);
```

引入的模块和回调函数不是同步的，所以浏览器不会因为引入的模块加载不成功而假死。

AMD 规范实践最广泛的前端框架应该是是 AngularJS1.x 版本，随着 React 和 Vue 的出现，现在已很少见到了。

UMD(Universal Module Definition)

UMD 是一种通用模块定义规范，严格上说它不能算是一种规范，因为它没有模块定义和调用。它的作用在于提供其他模块(AMD 和 CommonJS)的结合体，使得模块可以被不同的模块规范调用，通俗点讲就是浏览器和node都可以用。

项目地址

ES Module

不论是 CommonJS，还是 AMD 或者 UMD 或者其他，本质上都是社区的开发者们制定的模块加载方案，并不是语言层面的标准。从 ES6 开始，在语言标准的层面上，实现了模块化功能，而且实现得相当简单，完全可以取代 CommonJS 和 CMD、AMD 规范，成为浏览器和服务器通用的模块解决方案，这也是我们现在最常用的模块方案。

ES 模块与 CommonJS

Node.js 从 8.5 版本开始支持 ES 模块，从 12.0.0 版本开始，通过 package.json 的 `"type"` 字段添加对使用 .js 文件扩展名的 ES 模块的支持。

扩展名问题

除了.js外，有时候我们会看到.cjs和.mjs这两种扩展名。Node.js 的 ES 模块加载器接受且仅接受这三种扩展名。

在包中，package.json 的"type"字段定义了 Node.js 应该如何解释 .js 文件。 如果 package.json 文件没有"type"字段，或者字段值为"commonjs"，则 .js 文件将被视为 CommonJS。如果"type"字段值为"module"，则 .js 文件将被视为 ES 模块。

以.mjs结尾的文件总是加载为 ES 模块，而不管最近的父级 package.json。

以.cjs结尾的文件总是加载为 CommonJS 模块，而不管最近的父级 package.json。

.mjs和.cjs可用于在同一个包中混合类型：

- 在 "type": "module" 包中，可以指示 Node.js 通过将特定文件命名为 .cjs 扩展名将其解释为 CommonJS（因为 .js 和 .mjs 文件都被视为 "module" 包中的 ES 模块）。
- 在 "type": "commonjs" 包中，可以指示 Node.js 将特定文件解释为 ES 模块，方法是将其命名为 .mjs 扩展名（因为 .js 和 .cjs 文件在 "commonjs" 包中都被视为 CommonJS）。

注意，.mjs和.cjs在 typescript 中不支持。

双模块包

双模块包是指在保重同时包含 CommonJS 和 ES 模块的源码。其中 package.json 中的"main"字段指定 CommonJS 入口点，"module"字段指定 ES 模块入口点。这种做法使 Node.js 能够运行 CommonJS 入口点，而构建工具（例如打包器）使用 ES 模块入口点，因为 Node.js 忽略（并且仍然忽略）顶层"module"字段（关于"module"字段，可以参考这篇文章）。

Pure ES 包

顾名思义，Pure ES 包是指仅包含 ES 模块源码的包。虽然 EM 模块作为面向未来的 JavaScript 模块化标准，已经足以应对现代模块化开发的大部分场景，但是仍然有不少包是采用 CommonJS 规范。而 ES 和 CommonJS 两者相互并不兼容（现在有了一些互通方案），可见这个概念是比较激进的。

关于 Pure ES 包和双模块包，社区有不少争论，这些不属于本次讨论范围，这里仅介绍相关概念。

解决问题

现在回到前面的问题。我们的代码使用的是 CommonJS，而现代工具创建的项目都是基于 ES 模块的，它无法与 CommonJS 兼容，因此我们只需要将我们的项目代码改成 ES 即可：

```
// index.js
// module.exports = bigBang;
export default bigBang;
```

编译

目前为止，一个基本的包已经开发完成，但是很多情况下做到这一步还不够，在某些场景下甚至无法使用。比较常见的有下面几种场景：

1. 兼容性问题
2. 采用了某些框架特定的语法或特性（比如 jsx 语法）
3. 需要在模块规范之间转换，比如从 ES 模块转成 CommonJS

比如我们的 chaos 包里面用到了箭头函数，但是旧版本的浏览器是无法识别的（chrome 从版本 45 开始支持箭头函数）。要解决这个问题，就需要用到编译工具对代码进行编译。

从传统意义上来说，JavaScript 代码是不需要编译的，因为它本身是解释型语言，而浏览器内置了它的解释器和运行环境。但是这里我们依然能套用【编译】这个概念：利用编译程序从源语言编写的源程序产生目标程序，只不过这里的原程序和目标程序都是 JavaScript 语言的程序。

Babel

谈到 JavaScript 编译器，无论如何也绕不开 Babel ，甚至有些人把它当作 JavaScript 编译器的代名词。它的主要目标是将 ECMAScript 2015+ 代码转换为 JavaScript 向后兼容版本的代码（当然还可以做一些其他的事情）。更多关于 Babel 的内容可以查阅官网，网络上也有大量的相关教程和文章，这里仅演示如何使用 Babel 解决 chaos 的兼容性问题。

1. 在 chaos 根目录下新建 src 目录，并将 index.js 文件移入该目录

2. 安装依赖包：

```
npm install --save-dev @babel/core @babel/cli @babel/preset-env
```

3. 在 chaos 的根目录下创建名为 babel.config.json 的配置文件，并填入以下内容：

```
{
  "presets": [
    [
      "@babel/preset-env",
      {
        "targets": {
          "chrome": "40"
        },
        "useBuiltIns": "usage",
        "corejs": "3.6.5",
        "modules": false
      }
    ]
  ]
}
```

4. 在 package.json 的"scripts"字段下新增"build"命令：

```
{
  "scripts": {
    "build": "babel src --out-dir lib"
  }
}
```

5. 修改 package.json 的"main"字段为"lib/index.js"

现在，我们运行命令就可以将 src 目录下的代码编译到 lib 目录下：

```
npm run build
```

SWC

尽管 Babel 已经非常成熟稳健了，但由于先天不足，在 JavaScript 项目日益复杂的今天，它的性能越来越跟不上需求，于是 SWC 就出现了。SWC 是一个基于 Rust 开发

的，按照官方的说法，SWC 在单线程情况下比 Babel 块 20 倍，四核下要快 70 倍。

相比于 Babel，它的优点就是快，但是缺点也很明显，生态相对较弱，甚至官方文档上很多内容都是直接链接到别处。尽管如此，也足以满足绝大多数场景了，目前 Next.js 就是使用 SWC 作为编译工具。

SWC 在项目中的用法与 Babel 类似，具体可以参考官方文档，另外 glory 中的 core、layout 和 query 也是用的 SWC，这里就不演示了。

tsc

严格来说 tsc 并不是 JavaScript 编译器，它是 TypeScript 附带的命令行工具。但是它确实能编译 JavaScript，并且有意思的是即使是编译 JavaScript 代码，也可以做类型推导并生成.d.ts 文件，只不过很少有人这么做。

创建双模块包

前面提到了双模块包的概念，也看到了如何使用 Babel 编译我们的代码。在文件 babel.config.json 中有一个选项：`"modules": false`，这个选项表示是否将 ES 模块转换成其他模块，创建双模块包就是利用这个选项。可以通过不同的配置文件，或者自己编写编译脚本来设置这个选项的值来编译成不同模块。关于这个选项的详细说明可以查阅官网

打包

绝大多数场景下我们的 npm 包是不需要打包的。但是如果我们需要将代码保密做混淆，或者需要支持 CDN，可能就需要这一步操作。还有一种情况是如果我们希望使用打包工具的强大的生态系统以方便我们的开发，也可以使用这类工具。

最常见的打包工具是 webpack，它与 Babel 一样在前端开发界应该是无人不知。它功能强大，却也因笨重难用而饱受诟病，所以有了一些替代品，近年来最有名的应该是 rollup。关于它们之间的对比这里不展开，一个认同度比较高的观点是 rollup 更适合组件库的打包。这里不演示它们的用法，如果有需求可以查阅它们的官网，或者在网络上搜索相关教程。

webpack Rollup

优化

至此，我们绝大部分工作已经完成了，但依然有一些很重要的事情要做。

单元测试

一个库要保证质量，单元测试是必不可少的。这里以测试框架 jest 为例：

```
npm install --save-dev jest
```

在项目根目录下新建文件 index.test.js，并加入以下内容：

```
const bigBang = require("./index");

test("一斧开天", () => {
  expect(bigBang().toBe("天地玄黄，宇宙洪荒。"));
});
```

然后修改在 package.json 中的 test 指令：

```
{
  "scripts": {
    "test": "jest"
  }
}
```

最后执行测试指令，jest 就会输出测试结果：

```
npm test

> chaos@1.0.0 test
> jest

 PASS  ./index.test.js
  √ 开天辟地！ (4 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
```

```
Time:        1.215 s, estimated 2 s
Ran all test suites.
```

关于该框架的详细内容，可以参考官网。需要注意的是这个框架默认使用 CommonJS，如果要支持 ES 模块，需要一些额外的配置，可以查看这里。

添加类型

添加类型有两种常见的方案：

1. 使用 TypeScript 编写，然后编译出声明文件
2. 单独编写声明文件

如果你开发的是 react 组件库，还有第三种选择：借助于 prop-types 这个库。这里我们不讨论这个场景，只看前两种。

使用 TypeScript 并不一定会生成声明文件，这取决于所采用的编译器以及相关配置。常用的便以及依然是我们前面列出来的三个：

- Babel：Babel 本身是用来处理 JavaScript 代码的，要处理 TypeScript 就需要借助插件。不过官方已经为我们准备好了：@babel/preset-typescript。需要注意的是 Babel 在处理 TypeScript 代码的时候不做类型检查，也不会生成`.d.ts`文件，并且部分 TypeScript 语法 Babel 不支持（const enum 和 namespace）。
- SWC：SWC 不需要插件就能编译 TypeScript 代码，但在编译 TypeScript 代码的时候不会生成`.d.ts`文件，从这点看，SWC 似乎除了快一无是处，但天下武功，唯快不破。
- tsc：tsc 是 TypeScript 官方的编译工具，但是除了编译 TypeScript 之外，它的其他功能不如 Babel 那般强大，并且会全量引入 core-js。

它们之间的区别不止这些，但仅从上面的简单对比可以看出，编译器的选择似乎是件比较困难的事情，这正应了那句老话：软件开发没有银弹。对于它们之间的权衡网络上有不少意见，比如在用 Babel 编译之前先用 tsc 做类型检查，再比如把类型检查放到开发阶段，直接依赖编辑器的错误提示等等。但不论如何，我们应用的过程中根据实际需求选择即可。glory 的 screw 和 spare 使用了 TypeScript 开发，并选用 tsc 作为编译器。

如果使用 Babel 或者 SWC 作为编译器，或者直接编写 JavaScript 代码，或者使用了 tsc，并且将`declaration`选项设为`false`，就需要手写声明文件。

类型识别

申明文件的使用有两种方式：一种是跟源码放在一起，另一种是单独做成一个包。

如果选择第一种方式，我们需要在 package.json 中添加"types"字段（注意不是前提到过的"type"），它的值指向类型文件的入口（类似于"main"字段）：

```
{
  "types": "types/index.d.ts"
}
```

注意，这个字段并非 npm 自带的，而是为了 typescript 服务的。

如果要单独做成一个包，则需要用户将包安装到 node_modules/@types 目录下。@types 是统一由 DefinitelyTyped 管理的。要将声明文件发布到 @types 下，就需要给 DefinitelyTyped 创建一个 pull-request，其中包含了类型声明文件，测试代码，以及 tsconfig.json 等，现在不建议这么做，即使做了也不一定会成功。

eslint 和 prettier

略

代码规范

Git 自带的 hook 大家应该都有了解，要自己编写 hook 可能会比较麻烦。这里介绍两个工具可以很方便的帮我们完成这件工作，并且让 hook 可以在项目成员之间共享。

husky

git hooks 是本地的，不会被同步到 git 仓库里。为了保证每个人的本地仓库都能执行预设的 git hooks，于是就有了 husky，它可以帮我们非常轻松地创建 git hooks。

lint-staged

有了 hook，我们就可以在提交代码之前执行 lint 等格式化的操作。但是随着项目越来越大，全量格式化的耗时越来越长，这就导致提交代码的效率变得非常低。假设现有的代码都是经过 lint 的，然后在提交的时候只需要对有变动的代码做 lint，那么提交速度就会大大提高，于是就有了 lint-staged

这两个工具配合基本可以满足我们绝大多数场景下的需求了，它们的集成方式可以在各自的文档上找到。

文档

最重要的文档就是项目根目录下的 readme.md 文件。不论是 npm 上的项目主页，还是 git，都会以这个文件作为项目说明。

对于一个简单的包，可能只需要写 readme 就够了，但是复杂的就需要单独的文档了，市面上生成文档的工具比较多，如JSDoc、ESDoc等。另外还有一些静态网站生成器，比较出名的有 VuePress。如果你在开发 React 组件库，那么可以考虑使用React Styleguidist，或者dumi，Ant Design 的官网就是用它做的。

发布

配置项补充

发布之前，package.json 里还有一些配置需要补充。比如 `"keywords"`，用于在 npm 上搜索。再比如假设包只能在特定平台上运行（例如 linux），则需要配置 `"os"` 字段。其他还有一些字段以及它们的作用，可以在官网上找到，这里就不一一列举了。

版本号

版本号主要由四部分组成：

- major：代表主版本号，通常在需要提交不能向下兼容的情况下对该版本号进行升级
- minor：代表次版本号，通常在新增功能时才对该版本号进行升级
- patch：代表修复版本号，升级该版本号通常代表修复一些 bug，但没有新增功能或者存在不向下兼容的功能
- prerelease：带有该版本号的包通常表示在测试阶段，尚未稳定，通常不建议用户安装。

其中，prerelease 部分可以分为以下三类：

- alpha: 代表内部测试版，会有很多 Bug，是比 beta 更早的版本，一般不建议对外发布

- beta: 相对 alpha 版本已有了很大的改进，但还是存在一些缺陷，需要经过多次测试来进一步消除
- rc：Release Candidate 顾名思义就是正式发布的候选版本。和 Beta 版最大的差别在于 Beta 阶段会一直加入新的功能，但是到了 RC 版本，几乎就不会加入新的功能了，而主要着重于除错! RC 版本是最终发放给用户的最接近正式版的版本，发行后改正 bug 就是正式版了，就是正式版之前的最后一个测试版

例如：

1.1.0-alpha.1

1.1.0-beta.1

1.1.0-rc.1

版本号可以手动修改，也可以使用工具自动修改。

npm 的 version 命令可以修改 package.json 的 version 版本：

```
npm version [<newversion> | major | minor | patch | premajor | preminor | prepatch
| prerelease | from-git]
```

发布到 npm

发布到 npm 之前，我们必须有一个账号，如果没有则需要去官网注册。

如果是第一发布，则要先登录：

```
npm login
```

登录完成后执行 publish 就可以了：

```
npm publish
```

有几个细节需要注意：

1. 如果之前有改过源，则要切换回官方源
2. 第一次发布，最好去官网查一下是否有重名的，重名会导致发布不成功

3. 注册后需要邮箱验证，未验证也会导致发布失败

发布到其他地方

npm 包不一定非要发布到 npm 上别人才可以使用。除了从 npm 仓库导入外，npm 支持四种导入方式：

1. URL
2. git
3. GitHub
4. 本地路径

它们的用法可以查阅官方文档。glory 就是发布在公司的 git 上，然后通过 git 导入的。

其他

Monorepo

Monorepo 是一种项目代码管理方式，指单个仓库中管理多个项目，有助于简化代码共享、版本控制、构建和部署等方面的复杂性，并提供更好的可重用性和协作性。事实上前面讲本地测试的时候我们已经接触到这一概念了，但是 npm 自带的 workspace 功能较弱，对于一些复杂的场景，我们可能需要一些其他第三方工具。glory 使用了 lerna。更多关于 Monorepo 的详细介绍以及一些选型建议可以参考这篇文章。

CDN

如果要支持从 CDN 导入，则我们的包必须支持 UMD 模块。

unpkg

unpkg 是一个内容源自 npm 的全球快速 CDN。npm 包要发布到 unpkg 上非常简单，只需要下面几个步骤即可：

1. 添加 dist 目录，同时添加到.gitignore
2. 编译代码生成 umd 格式的文件到 dist 目录中
3. 在 package.json 的 `"files"` 字段里添加`"dist"`