git提交规范

一：分支管理

- **master 分支**
  - 主分支，永远处于稳定状态，对应当前线上版本
  - 以 tag 标记一个版本，因此在 master 分支上看到的每一个 tag 都应该对应一个线上版本
  - 不允许在该分支直接提交代码
- **develop 分支**
  - 开发分支，包含了项目最新的功能和代码，所有开发都依赖 develop 分支进行
  - 小的改动可以直接在 develop 分支进行，改动较多时切出新的 feature 分支进行

  注： 更好的做法是 develop 分支作为开发的主分支，也不允许直接提交代码。小改动也应该以 feature 分支提 merge request 合并，目的是保证每个改动都经过了强制代码 review，降低代码风险。
- **feature 分支**
  - 功能分支，开发新功能的分支
  - 开发新的功能或者改动较大的调整，从 develop 分支切换出 feature 分支，分支名称为 `feature/xxx`
  - 开发完成后合并回 develop 分支并且删除该 feature/xxx 分支
- **release 分支**
  - 发布分支，新功能合并到 develop 分支，准备发布新版本时使用的分支
  - 当 develop 分支完成功能合并和部分 bug fix，准备发布新版本时，切出一个 release 分支，来做发布前的准备，分支名约定为 `release/xxx`
  - 发布之前发现的 bug 就直接在这个分支上修复，确定准备发版本就合并到 master 分支，完成发布，同时合并到 develop 分支
- **hotfix 分支**
  - 紧急修复线上 bug 分支
  - 当线上版本出现 bug 时，从 master 分支切出一个 `hotfix/xxx` 分支，完成 bug 修复，然后将 `hotfix/xxx` 合并到 master 和 develop 分支(如果此时存在 release 分支，则应该合并到 release 分支)，合并完成后删除该 `hotfix/xxx` 分支

二: 流程

1. 新建dev分支 以组为单位新建 feature/exam feature/course feature/student ...
2. 以人为单位 每个组基于各组的功能新建 feature/exam/list feature/exam/create

3. 功能完整后合并到--> feature/xxx -->
4. 组内开饭完成后 合并到dev分支
5. 分离release 分支 进行测试 -->测试分支命名为 fix/xxx

6. 合并到master分支
7. 在dev merge release

8. 线上有bug master上创建一个 hotfix 修改master的bug 和并到dev分支

三: 提交规范

3.1 Commitizen

Commitizen是一个撰写合格 Commit message 的工具。

安装命令如下。

```
$ npm install -g commitizen
```

然后，在项目目录里，运行下面的命令，使其支持 Angular 的 Commit message 格式。

```
$ commitizen init cz-conventional-changelog --save --save-exact
```

以后，凡是用到 `git commit` 命令，一律改为使用 `git cz` 。这时，就会出现选项，用来生成符合格式的 Commit message。

```
feat:  A new feature  //新的功能

fix:   A bug fix  //修复bug

docs:   Documentation only changes  // 文档改变

style:  Changes that do not affect the meaning of the code (white-space, f
ormatting, missing semi-colons, etc) // 格式化

refactor: A code change that neither fixes a bug nor adds a feature  //既不
修复错误也不添加功能的代码更改，重构

perf:   A code change that improves performance // 优化  减少重绘和回流
```

```
test:   Adding missing tests or correcting existing tests // 添加缺失的测试或纠正现有的测试 单元测试

build:  Changes that affect the build system or external dependencies (example scopes: gulp, broccoli, npm) // 下载了包

ci:    Changes to our CI configuration files and scripts (example scopes: Travis, Circle, BrowserStack, SauceLabs)  // 配置文件 babel.config.js

chore:  Other changes that don't modify src or test files  // 全局属性

revert:  Reverts a previous commit // 回退版本
```

```
? Select the type of change that you're committing: ci:       Changes to our CI configuration files and scripts (example scopes: Travis, Circle, BrowserStack, SauceLabs)
? What is the scope of this change (e.g. component or file name): (press enter to skip) cli
? Write a short, imperative tense description of the change (max 91 chars):
 (16) make syatem full
? Provide a longer description of the change: (press enter to skip)
 xxxxxxxxxxxxxxx
? Are there any breaking changes? Yes
? Describe the breaking changes:
 create a project
? Does this change affect any open issues? No
[master（根提交） ad818a4] ci(cli): make syatem full
 5 files changed, 2634 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 package-lock.json
 create mode 100644 package.json
 create mode 100644 package/admin/1.txt
 create mode 100644 package/server/2.txt
bogon:admin_projecrt sairitsutakara$ git log
commit ad818a451761cd96764b9c52dc26a097955fe5a8 (HEAD -> master)
Author: bingyu123 <cuilibao123@gmail.com>
Date:   Wed Nov 24 12:04:47 2021 +0800

    ci(cli): make syatem full

    xxxxxxxxxxxxxxx

    BREAKING CHANGE: create a project
bogon:admin_projecrt sairitsutakara$
```

```
➜  ng-poopy  master ✗  git add .
➜  ng-poopy  master ✗  git cz

All commit message lines will be cropped at 100 characters.

? Select the type of change that you're committing: (Use arrow keys)
❯ feat:     A new feature
  fix:      A bug fix
  docs:     Documentation only changes
  style:    Changes that do not affect the meaning of the code
            (white-space, formatting, missing semi-colons, etc)
  refactor: A code change that neither fixes a bug or adds a feature
  perf:     A code change that improves performance
  test:     Adding missing tests
  chore:    Changes to the build process or auxiliary tools
            and libraries such as documentation generation
```

3.2 validate-commit-msg

validate-commit-msg 用于检查 Node 项目的 Commit message 是否符合格式。

它的安装是手动的。首先，拷贝下面这个JS文件，放入你的代码库。文件名可以取为 `validate-commit-msg.js` 。

接着，把这个脚本加入 Git 的 hook。下面是在 `package.json` 里面使用 ghooks，把这个脚本加为 `commit-msg` 时运行。

```
"config": {
 "ghooks": {
   "commit-msg": "./validate-commit-msg.js"
 }
}
```

然后，每次 `git commit` 的时候，这个脚本就会自动检查 Commit message 是否合格。如果不合格，就会报错。

```
$ git add -A
$ git commit -m "edit markdown"
INVALID COMMIT MSG: does not match "<type>(<scope>): <subject>" !
was: edit markdown
```

3.3 生成 Change log

如果你的所有 Commit 都符合 Angular 格式，那么发布新版本时， Change log 就可以用脚本自动生成（例1，例2，例3）。

生成的文档包括以下三个部分。

> - New features
> - Bug fixes
> - Breaking changes.

每个部分都会罗列相关的 commit ，并且有指向这些 commit 的链接。当然，生成的文档允许手动修改，所以发布前，你还可以添加其他内容。

conventional-changelog 就是生成 Change log 的工具，运行下面的命令即可。

```
$ npm install -g conventional-changelog
$ cd my-project
$ conventional-changelog -p angular -i CHANGELOG.md -w
```

上面命令不会覆盖以前的 Change log，只会在 `CHANGELOG.md` 的头部加上自从上次发布以来的变动。

如果你想生成所有发布的 Change log，要改为运行下面的命令。

```
$ conventional-changelog -p angular -i CHANGELOG.md -w -r 0
```

为了方便使用，可以将其写入 `package.json` 的 `scripts` 字段。

```
{
"scripts": {
 "changelog": "conventional-changelog -p angular -i CHANGELOG.md -w -
r 0"
}
}
```

以后，直接运行下面的命令即可。

```
$ npm run changelog
```