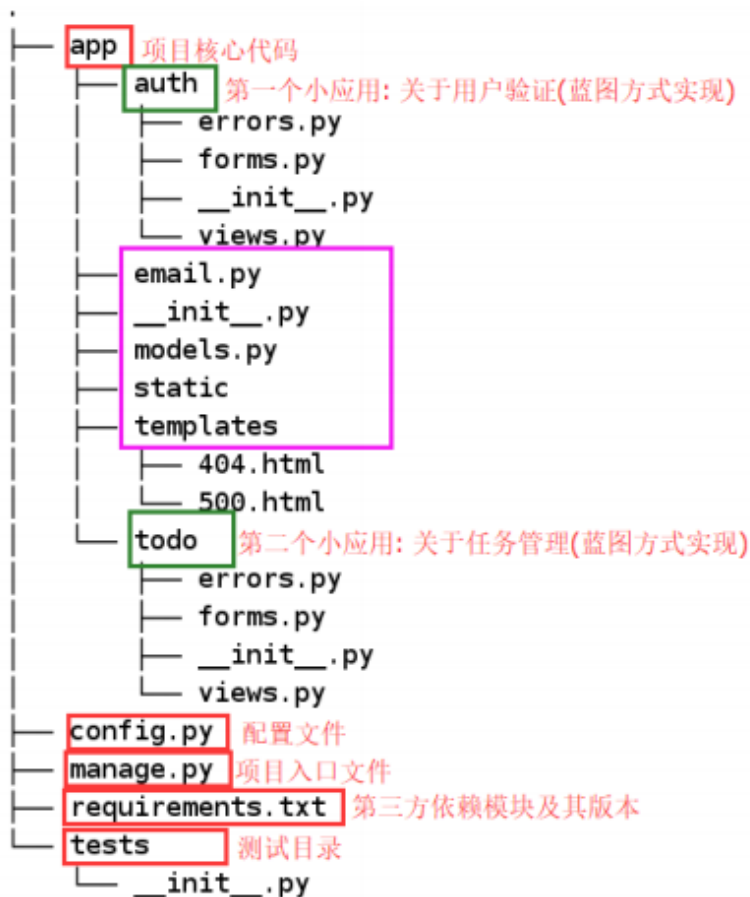


Flask开发大型项目结构

项目结构

多文件 Flask 程序的基本结构, 如下图所示:



- `requirements.txt` 列出了所有依赖包,便于在其他电脑中重新生成相同的虚拟环境;
- `config.py` 存储配置;
- `manage.py` 用于启动程序以及其他的程序任务。

配置文件选项

程序经常需要设定多个配置。一般分为开发、测试和生产环境, 它们使用不同的数据库,不会彼此影响。

```
# config.py
"""
存储配置;
"""

import os
# 获取当前项目的绝对路径;
basedir = os.path.abspath(os.path.dirname(__file__))
```

```

class Config:
    """
    所有配置环境的基类，包含通用配置
    """

    SECRET_KEY = os.environ.get('SECRET_KEY') or 'westos secret key'
    SQLALCHEMY_COMMIT_ON_TEARDOWN = True
    SQLALCHEMY_TRACK_MODIFICATIONS = True
    FLASKY_MAIL_SUBJECT_PREFIX = '[西部开源]'
    FLASKY_MAIL_SENDER = '976131979@qq.com'

    @staticmethod
    def init_app(app):
        pass


class DevelopmentConfig(Config):
    """
    开发环境的配置信息
    """

    # 启用了调试支持，服务器会在代码修改后自动重新载入，并在发生错误时提供一个相当有用的调试
    器。
    DEBUG = True
    MAIL_SERVER = 'smtp.qq.com'
    MAIL_PORT = 587
    MAIL_USE_TLS = True
    MAIL_USERNAME = os.environ.get('MAIL_USERNAME') or '976131979'
    MAIL_PASSWORD = os.environ.get('MAIL_PASSWORD') or '密码'
    SQLALCHEMY_DATABASE_URI = 'sqlite:/// ' + os.path.join(basedir, 'data-
dev.sqlite')


class TestingConfig(Config):
    """
    测试环境的配置信息
    """

    TESTING = True
    SQLALCHEMY_DATABASE_URI = 'sqlite:/// ' + os.path.join(basedir, 'data-
test.sqlite')


class ProductionConfig(Config):
    """
    生产环境的配置信息
    """

    SQLALCHEMY_DATABASE_URI = 'sqlite:/// ' + os.path.join(basedir,
'data.sqlite')


config = {
    'development': DevelopmentConfig,
    'testing': TestingConfig,
    'production': ProductionConfig,
    'default': DevelopmentConfig
}

```

程序工厂函数

- 为什么需要程序工厂函数?

在单个文件中开发程序很方便,但却有个很大的缺点,因为程序在全局作用域中创建,所以无法动态修改配置。运行脚本时,程序实例已经创建,再修改配置为时已晚。这一点对单元测试尤其重要,因为有时为了提高测试覆盖度,必须在**不同的配置环境中**运行程序。

这个问题的解决方法是**延迟创建程序实例**,把创建过程移到可显式调用的工厂函数中。这种方法不仅可以给脚本留出配置程序的时间,还能够创建多个程序实例。

- 如何实现程序工厂函数?

创建扩展类时不向构造函数传入参数, 在之前创建的扩展对象上调用 `init_app()` 可以完成初始化过程。

不使用程序工厂函数:

```
app = Flask(__name__)
bootstrap = Bootstrap(app)
mail = Mail(app)
```

使用程序工厂函数:

```
bootstrap = Bootstrap()
mail = Mail()

def create_app():
    app = Flask(__name__)
    bootstrap.init_app(app)
    mail.init_app(app)
    return app
```

- `app/__init__.py` 文件详细代码如下:

```
"""
程序工厂函数，延迟创建程序实例
"""

from flask import Flask
from flask_bootstrap import Bootstrap
from flask_mail import Mail
from flask_sqlalchemy import SQLAlchemy
from config import config

bootstrap = Bootstrap()
mail = Mail()
db = SQLAlchemy()

def create_app(config_name='development'):
    """
    默认创建开发环境的app对象
    """
    app = Flask(__name__)
    app.config.from_object(config[config_name])
    config[config_name].init_app(app)
    bootstrap.init_app(app)
```

```
mail.init_app(app)
db.init_app(app)

# 附加路由和自定义的错误页面
# .....后续还需完善， 补充视图和错误页面

return app
```

蓝图: 组件化开发

什么是蓝图?

Flask蓝图提供了模块化管理程序路由的功能，使程序结构清晰、简单易懂。

假如说我们要为某所学校的每个人建立一份档案，一个很自然的优化方式就是这些档案如果能分类管理，就是说假如分为老师、学生、后勤人员等类别，那么后续查找和管理这些档案就方便清晰许多。Flask的蓝图就提供了类似“分类”的功能。

为什么使用蓝图?

- 将不同的功能模块化
- 构建大型应用
- 优化项目结构
- 增强可读性,易于维护

应用蓝图三部曲?

- 蓝图的创建: `app/auth/__init__.py`

```
# 'auth' 是蓝图的名称
# __name__ 是蓝图所在路径
auth =Blueprint('auth',__name__)
from . import views
```

- 蓝图对象上注册路由,指定静态文件夹,注册模版过滤器: `app/auth/views.py`

```
@auth.route('/')
def auth_home():
    return 'auth_home'
```

- 注册蓝图对象 `app/__init__.py`

```
# url_prefix: 指定访问该蓝图中定义的视图函数时需要添加的前缀， 没有指定则不加；
app.register_blueprint(auth,url_prefix='/auth')
```

- 访问网址 `http://127.0.0.1:5000/auth/` 可以查看到 `auth_home` 的内容。

任务清单蓝图的应用

auth 蓝图

```

app
├── auth
│   ├── errors.py
│   ├── forms.py
│   ├── __init__.py
│   └── views.py
└──

```

- 蓝图的创建: `app/auth/__init__.py`

```

from flask import Blueprint

# 实例化一个 Blueprint 类对象可以创建蓝本，指定蓝本的名字和蓝本所在的包或模块
auth = Blueprint('auth', __name__)
# 把路由和错误处理程序与蓝本关联，一定要写在最后，防止循环导入依赖；
from . import views, errors

```

- 蓝图对象上注册路由,指定静态文件夹,注册模版过滤器: `app/auth/views.py`

```

from . import auth

@auth.route('/login')
def login():
    return 'login'

@auth.route('/logout')
def logout():
    return 'logout'

```

- 注册蓝图对象 `app/__init__.py`

```

def create_app(config_name='development'):
    """
    默认创建开发环境的app对象
    """
    # .....

    # 附加路由和自定义的错误页面
    from app.auth import auth as auth_bp
    app.register_blueprint(auth_bp) # 注册蓝本

    return app

```

todo 蓝图

```
— todo
  ├── errors.py
  ├── forms.py
  ├── __init__.py
  └── views.py
```

- 蓝图的创建: `app/todo/__init__.py`

```
from flask import Blueprint

# 实例化一个 Blueprint 类对象可以创建蓝本，指定蓝本的名字和蓝本所在的包或模块
todo = Blueprint('todo', __name__)
# 把路由和错误处理程序与蓝本关联，一定要写在最后，防止循环导入依赖；
from . import views, errors
```

- 蓝图对象上注册路由,指定静态文件夹,注册模版过滤器: `app/todo/views.py`

```
from . import todo

@todo.route('/add/')
def add():
    return 'todo add'

@todo.route('/delete/')
def delete():
    return 'todo delete'
```

- 注册蓝图对象 `app/__init__.py`

```
def create_app(config_name='development'):
    """
    默认创建开发环境的app对象
    """
    # .....

    # 附加路由和自定义的错误页面
    from app.auth import auth as auth_bp
    app.register_blueprint(auth_bp) # 注册蓝本
    from app.todo import todo as todo_bp
    app.register_blueprint(todo_bp, ) # 注册蓝本

    return app
```

启动脚本

顶级文件夹中的 `manage.py` 文件用于启动程序。

```
"""
用于启动程序以及其他的程序任务。
"""
```

```

from flask_migrate import Migrate, MigrateCommand
from flask_script import Manager, Shell

from app import create_app, db

app = create_app('default')
manager = Manager(app)
migrate = Migrate(app, db)

def make_shell_context():
    return dict(app=app, db=db)

# 初始化 Flask-Script、Flask-Migrate 和为 Python shell 定义的上下文。
manager.add_command("shell", Shell(make_context=make_shell_context))
manager.add_command('db', MigrateCommand)
if __name__ == '__main__':
    manager.run()

```

基于 Unix 的操作系统中可以通过下面命令执行脚本:

```

# python3 manage.py runserver --help 获取详细使用参数
python3 manage.py runserver

```

效果如下:

```

[kiosk@foundation0 Todolist]$ python3 manage.py runserver
* Serving Flask app "app" (lazy loading)
* Environment: production
  WARNING: Do not use the development server in a production environment.
  Use a production WSGI server instead.
* Debug mode: on
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 367-866-036

```

依次访问网址, 如果访问可以显示相关信息, 则成功。

- `http://127.0.0.1:5000/login`
- `http://127.0.0.1:5000/logout`
- `http://127.0.0.1:5000/todo/add/`
- `http://127.0.0.1:5000/todo/delete/`

依赖包文件

程序中必须包含一个 `requirements.txt` 文件,用于记录所有依赖包及其精确的版本号。

```

pip freeze > requirements.txt

```

创建一个和当前环境相同的虚拟环境,并在其上运行以下命令:

```

pip install -r requirements.txt

```

单元测试

什么是单元测试？

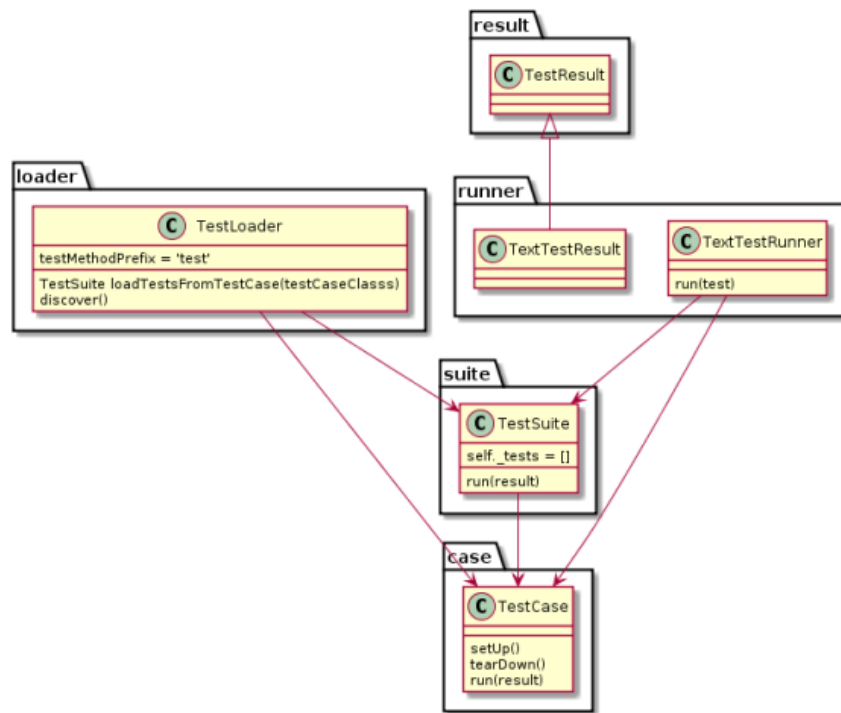
单元测试也称之为“模块测试”，是对程序设计中的最小单元——函数进行测试的一种方法，所谓测试，就是验证我们自己编写的方法能不能够得到正确的结果，即用方法得到的结果与真实结果进行比对，这就称之为测试。

如何实现单元测试？

python中有特别多的单元测试框架和工具，`unittest`，`testtools`，`subunit`，`coverage`，`testrepository`，`nose`，`mox`，`mock`，`fixtures`，`discover` 等等，先不说如何写单元测试，光是怎么运行单元测试就有N多种方法。`unittest`，作为标准python中的一个模块，是其它框架和工具的基础。

`unittest` 核心概念及关系

- `TestCase` 的实例就是一个测试用例。什么是测试用例呢？就是一个完整的测试流程，包括测试前准备环境的搭建(`setUp`)，执行测试代码(`run`)，以及测试后环境的还原(`tearDown`)。单元测试(`unit test`)的本质也就在这里，一个测试用例是一个完整的测试单元，通过运行这个测试单元，可以对某一个问题进行验证。
- `TestSuite` 是多个测试用例的集合，`TestSuite` 也可以嵌套 `TestSuite`。
- `TestLoader` 是用来加载 `TestCase` 到 `TestSuite` 中的，其中有几个 `loadTestsFrom__()` 方法，就是从各个地方寻找 `TestCase`，创建它们的实例，然后add到 `TestSuite` 中，再返回一个 `TestSuite` 实例。
- `TextTestRunner` 是用来执行测试用例的，其中的`run(test)`会执行 `TestSuite/TestCase` 中的 `run(result)`方法。
- `TextTestResult` 保存测试的结果，包括运行了多少测试用例，成功了多少，失败了多少等信息。



测试范例

```
# tests/test_number.py

import random
import unittest
"""
单独执行测试用例: python3 -m unittest test_number.py
"""

class TestSequenceFunctions(unittest.TestCase):
    """
    setUp() 和 tearDown() 方法分别在各测试前后运行,并且名字以 test_ 开头的函数都作为测试
    执行。
    """
    def setUp(self):
        self.seq = list(range(10))

    def test_shuffle(self):
        # make sure the shuffled sequence does not lose any elements
        random.shuffle(self.seq)
        self.seq.sort()
        self.assertEqual(self.seq, list(range(10)))

        # should raise an exception for an immutable sequence
        self.assertRaises(TypeError, random.shuffle, (1, 2, 3))

    def test_choice(self):
        element = random.choice(self.seq)
        self.assertTrue(element in self.seq)

    def test_sample(self):
```

```

with self.assertRaises(ValueError):
    random.sample(self.seq, 20)
for element in random.sample(self.seq, 5):
    self.assertTrue(element in self.seq)

def tearDown(self):
    del self.seq

```

- 单独执行测试用例:

```
python3 -m unittest test_number.py
```

- 运行结果如下:

```

[kiosk@foundation0 tests]$ python3 -m unittest test_number.py
..F
=====
FAIL: test_shuffle (test_number.TestSequenceFunctions)
[kiosk@foundation0 tests]$ python3 -m unittest test_number.py
...
-----
Ran 3 tests in 0.000s
OK

```

运行正确

任务清单单元测试的应用

第一个测试确保程序实例存在。第二个测试确保程序在测试配置中运行。若想把 tests 文件夹作为包使用,需要添加 tests/__init__.py 文件,不过这个文件可以为空,因为 unittest 包会扫描所有模块并查找测试。

```

# tests/test_basics.py

import unittest
from flask import current_app
from app import create_app, db

class BasicsTestCase(unittest.TestCase):
    """
    setUp() 和 tearDown() 方法分别在各测试前后运行,并且名字以 test_ 开头的函数都作为测试执行。
    """

    def setUp(self):
        """
        在测试前创建一个测试环境。
        1). 使用测试配置创建程序
        2). 激活上下文, 确保能在测试中使用 current_app
        3). 创建一个全新的数据库,以备不时之需。
        :return:
        """
        self.app = create_app('testing')
        self.app_context = self.app.app_context()

```

```

# Binds the app context to the current context.
self.app_context.push()
db.create_all()

def tearDown(self):
    db.session.remove()
    db.drop_all()
    # Pops the app context
    self.app_context.pop()

def test_app_exists(self):
    """
    测试当前app是否存在?
    """
    self.assertFalse(current_app is None)

def test_app_is_testing(self):
    """
    测试当前app是否为测试环境?
    """
    self.assertTrue(current_app.config['TESTING'])

```

为了运行单元测试,你可以在 `manage.py` 脚本中添加一个自定义命令。

```

# manage.py

# manager.command 修饰器让自定义命令变得简单。修饰函数名就是命令名,函数的文档字符串会显示在帮助消息中。
@manager.command
def test():
    """Run the unit tests."""
    import unittest
    tests = unittest.TestLoader().discover('tests')
    unittest.TextTestRunner(verbosity=2).run(tests)

```

单元测试可使用下面的命令运行:

```
python manage.py test
```

运行效果如下:

```

[kiosk@foundation0 Todolist]$ python3 manage.py test
test_app_exists (test_basics.BasicsTestCase) ... ok
test_app_is_testing (test_basics.BasicsTestCase) ... ok
test_choice (test_number.TestSequenceFunctions) ... ok
test_sample (test_number.TestSequenceFunctions) ... ok
test_shuffle (test_number.TestSequenceFunctions) ... ok

```

```
-----
Ran 5 tests in 0.032s
```

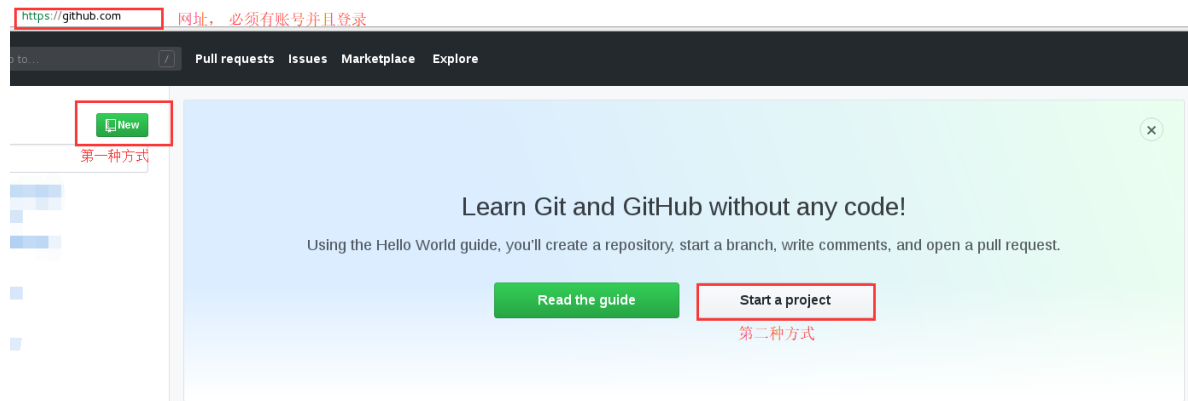
```
OK 测试用例全部通过
```

项目需求文档

文件 `README.md`, 建议包含项目介绍, 项目功能, 项目技术栈和项目最终演示效果。

项目与Github

- Github上新建一个仓库 Repositories




- 填写相关 Repositories 仓库信息

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository](#).

Owner **Repository name ***

 lvah / TodoList 仓库名称

Great repository names are short and memorable. Need inspiration? How about **supreme-chainsaw**?

Description (optional) 仓库描述, 尽量详细

任务清单管理系统采用 B/S 架构, 基于 Linux 平台开发。采用轻量级的Web服务器 Nginx, 其后端实现建议采

☒ **Public** 公有仓库, 所有人可以访问
Anyone can see this repository. You choose who can commit.

☐ **Private**
You choose who can see and commit to this repository.

Skip this step if you're importing an existing repository.

☐ **Initialize this repository with a README** 是否初始化产生文件README, 项目中已经编写, 这里不需要初始化
This will let you immediately clone the repository to your computer.

Add .gitignore: **None** | Add a license: **None** ⓘ

Create repository 点击即可创建

出现如下页面信息, 提示如何提交项目到 Github 上:

Quick setup — If you've done this kind of thing before

or **HTTPS** **SSH** <https://github.com/lvah/ToDoList.git>



Get started by [creating a new file](#) or [uploading an existing file](#). We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#).

...or create a new repository on the command line

```
echo "# ToDoList" >> README.md
git init  初始化项目目录为Git仓库
git add README.md
git commit -m "first commit"
git remote add origin https://github.com/lvah/ToDoList.git
git push -u origin master
```



...or push an existing repository from the command line

```
git remote add origin https://github.com/lvah/ToDoList.git
git push -u origin master
```



- 初始化项目为Git仓库，将项目文件添加到暂存区，提交到本地仓库，最终上传至远程仓库 **Github**。

初始化项目为Git仓库

```
git init
```

Initialized empty Git repository in
/home/kiosk/Desktop/201905python/ToDoList/.git/

将所有项目文件添加到暂存区

```
git add *
```

提交到本地仓库

```
git commit -m "Flask任务清单管理系统(一): 大型项目结构化管理"
```

添加一个新的远程仓库，第一次需要，后面的不需要添加。

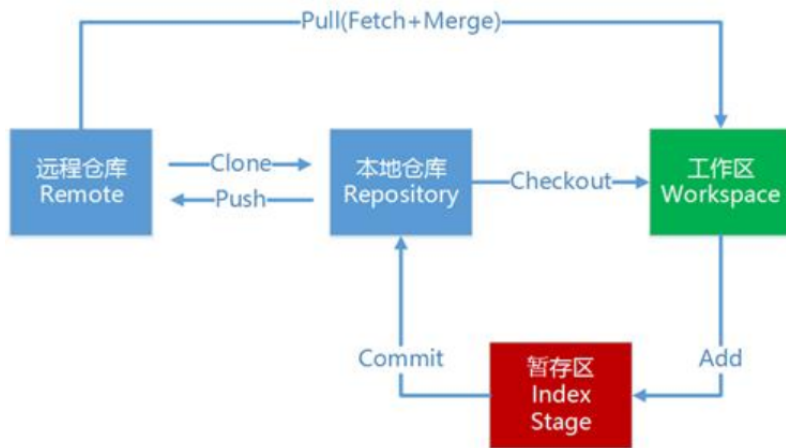
```
git remote add origin https://github.com/lvah/ToDoList.git
```

上传项目至远程仓库`Github`

```
git push -u origin master
```

```
[kiosk@foundation0 ToDoList]$ git remote add origin https://github.com/lvah/ToDoList.git
[kiosk@foundation0 ToDoList]$ git push -u origin master
Username for 'https://github.com': lvah
Password for 'https://lvah@github.com':
Counting objects: 37, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (35/35), done.
Writing objects: 100% (37/37), 11.70 KiB | 0 bytes/s, done.
Total 37 (delta 3), reused 0 (delta 0)
remote: Resolving deltas: 100% (3/3), done. 成功
To https://github.com/lvah/ToDoList.git
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
```

Git常用命令流程图



Git命令快速查找手册

Git 常用命令速查表		master :默认开发分支	Head :默认开发分支
		origin :默认远程版本库	Head^ :Head 的父提交
创建版本库		分支与标签	
\$ git clone <url>		\$ git branch	#显示所有本地分支
\$ git init		\$ git checkout <branch/tag>	#切换到指定分支或标签
		\$ git branch <new-branch>	#创建新分支
		\$ git branch -d <branch>	#删除本地分支
		\$ git tag	#列出所有本地标签
		\$ git tag <tagname>	#基于最新提交创建标签
		\$ git tag -d <tagname>	#删除标签
修改和提交		合并与衍合	
\$ git status		\$ git merge <branch>	#合并指定分支到当前分支
\$ git diff		\$ git rebase <branch>	#衍合指定分支到当前分支
\$ git add .			
\$ git add <file>			
\$ git mv <old> <new>			
\$ git rm <file>			
\$ git rm --cached <file>			
\$ git commit -m "commit message"			
\$ git commit --amend			
查看提交历史		远程操作	
\$ git log		\$ git remote -v	#查看远程版本库信息
\$ git log -p <file>		\$ git remote show <remote>	#查看指定远程版本库信息
\$ git blame <file>		\$ git remote add <remote> <url>	#添加远程版本库
		\$ git fetch <remote>	#从远程库获取代码
		\$ git pull <remote> <branch>	#下载代码及快速合并
		\$ git push <remote> <branch>	#上传代码及快速合并
		\$ git push <remote> :<branch/tag-name>	#删除远程分支或标签
		\$ git push --tags	#上传所有标签
撤销			
\$ git reset --hard HEAD			
\$ git checkout HEAD <file>			
\$ git revert <commit>			

总结

项目 Github 地址: <https://github.com/lvah/ToDoList>