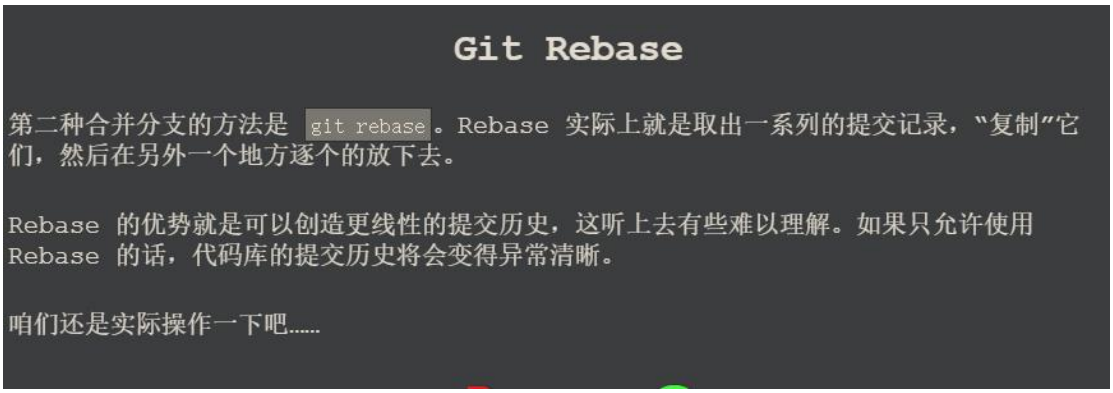
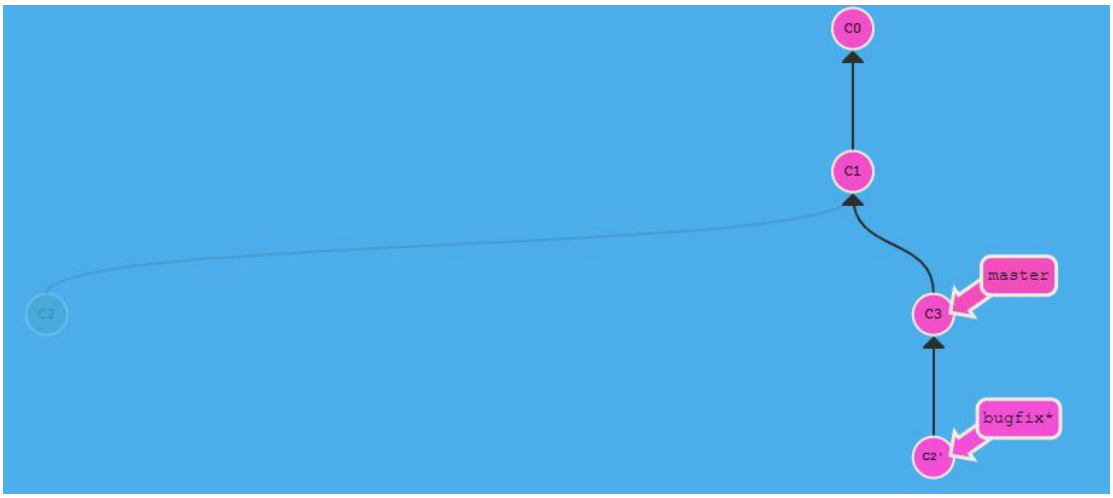
git

git rebase



```
$ git checkout -b bugfix
$ git commit
$ git checkout master
$ git commit
$ git checkout bugfix
$ git rebase master
```



head

HEAD

我们首先看一下 “HEAD”。 HEAD 是一个对当前检出记录的符号引用 —— 也就是指向你正在其基础上进行工作的提交记录。

HEAD 总是指向当前分支上最近一次提交记录。大多数修改提交树的 Git 命令都是从改变 HEAD 的指向开始的。

HEAD 通常情况下是指向分支名的（如 bugFix）。在你提交时，改变了 bugFix 的状态，这一变化通过 HEAD 变得可见。

User 项目练习

1. 数据库设计

名	类型	长度	小数点	不是 null	键
uid	int	11	0	☑	🔑 1
username	varchar	255	0	☐	
password	varchar	255	0	☐	
gender	varchar	255	0	☐	
tel	varchar	255	0	☐	
age	int	3	0	☐	

2. 接口设计

1) 登陆后的基本信息数据结构

```
   struct BasicInfo
       {
           0  optional  long        userId;              // 用户 id
           1  optional  string      token;               // token
           2  optional  string      username;            // 用户名
       };
```

2) 数据响应的基本状态数据结构

```
     struct BasicRsp
       {
           0  optional  int                    iRet;
// 返回码
           1  optional  string                 message;
// 返回信息
       };
```

3) 登录请求的数据结构

```
      struct LoginReq
       {
           0  require  string                 username;
 // 用户名
           1  require  string                 password;
 // 用户密码
       };
```

4) 登录响应的数据结构

```
       struct LoginRsp
       {
           0  optional  int                    iRet;
// 返回码
```

```
            1  optional  string                  message;
// 返回信息
        };
```

5) 分页请求的数据结构

```
      struct PageInfoReq
        {
            0  optional  BasicInfo               basicInfo;
// 基本信息
            1  optional  long                    uid;
 // 查询 ID
            2  optional  int                     page = 0;
// 当前页数 默认:0
            3  optional  int                     pageNum = 10;
// 每页显示条数 默认:10
        };
```

6) 根据 id 查询信息的数据结构

```
     struct QueryIdReq
        {
            0  optional  BasicInfo               basicInfo;
// 基本信息
            1  require   long                    uid;
// 用户 id
        };
```

7) 用户基本信息数据结构

```
        struct UserInfo
        {
            0  optional  long                    uid;
// 用户 id
            1  optional  string                  username;
 // 用户姓名
            2  optional  string                  password;
 // 用户密码
            3  optional  string                  gender;
// 用户性别
            4  optional  string                  tel;
// 用户电话
            5  optional  int                     age;
// 用户年龄
        };
```

8) 用户详细信息响应数据结构

```
        struct UserInfoRsp
        {
```

```
            0  optional  int                     iRet;
// 返回码
            1  optional  string                  message;
// 返回信息
            2  optional  UserInfo                userinfo;
 // 用户信息
        };
```

9) 用户列表数据结构

```
     struct UserInfoList
        {
            0  optional  int                     total;
// 总条数
            1  optional  vector<UserInfo>        list;
// 用户列表
        };
```

10) 用户列表响应数据结构

```
        struct UserListRsp
        {
            0  optional  int                     iRet;
// 返回码
            1  optional  string                  message;
// 返回信息
            2  optional  UserInfoList            data;
// 用户信息列表

        };
```

11) 保存用户请求数据结构

```
     struct SaveUserInfoReq
        {
            0   optional  BasicInfo              basicInfo;
 // 基本信息
            1   require  UserInfo               userinfo;
// 用户信息
        };
```

12) 保持用户响应数据结构

```
    struct SaveUserInfoRsp
        {
            0  optional  int                     iRet;
// 返回码
            1  optional  string                  message;
// 返回信息
```

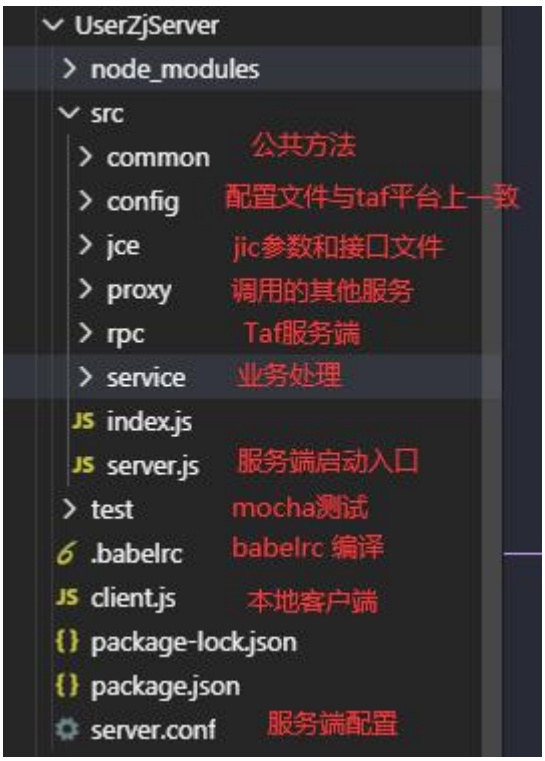
```
            2  optional  int                    uid;
 // 返回 id
        };
```

13)　　接口方法

```
    // h5 用户登录
            int login(LoginReq stReq, out LoginRsp stRsp);
            // h5 获取用户列表
            int getUserList(PageInfoReq stReq, out UserListRsp stRsp);
            // h5 获取用户详细信息
            int getUserDetail(QueryIdReq stReq, out UserInfoRsp stRsp);


               // h5 用户登录
            int loginForAd(LoginReq stReq, out LoginRsp stRsp);
            // admin 获取用户列表
            int getUserListForAd(PageInfoReq stReq, out UserListRsp stRs
p);
            // admin 删除用户
            int delUser(QueryIdReq stReq, out BasicRsp stRsp);
            // admin 添加/保存用户
            int saveUser(SaveUserInfoReq stReq, out SaveUserInfoRsp stRs
p);
            // h5 获取用户详细信息
            int getUserDetailForAd(QueryIdReq stReq, out UserInfoRsp stR
sp);
```

3. 项目结构搭建



4. 接口的实现（举一个例子，其他参照这个格式）

```
   import { Test } from './User';
   import UserService from '../service/User';
   import { reflectRun } from '../common';
   Test.UserImp.prototype.getUserDetailForAd = async function (current,
 stReq, stRsp) {
       await reflectRun(current, stReq, stRsp, UserService.getUserDetai
lForAd);
   }
```

5. 处理接口里的业务逻辑

```
    getUserDetailForAd: async req => {
      const {uid} = req;
      if(!uid) throw error.INPUT_ERROR();
      let getDetailSql = `select * from user_info where uid='${uid}';`
;
      console.log(getDetailSql);
```

```
      const [results, fields] = await conn.query(getDetailSql);
      const { username, password, gender, tel, age} = results[0];
            return {userinfo: {
              uid,
              username,
              password,
              gender,
              tel,
              age
            }};
    }
};
```

6. 数据库连接

```
// 引入
const mysql = require('mysql2');
// 创建数据库连接
// 使用异步
const conn  = mysql.createConnection({
    host: '192.168.6.170',
    user: 'root',
    password:'root',
    database: 'test_zzj',
    charset:'utf8',
  });
export default conn.promise();
```

7. taf 返回数据的处理

```
const OK_RET = 0;
const OK_RSP = { iRet: OK_RET, message: 'ok' };

const returnRet = (current, stRsp, ret) => {
  console.log(ret);
  stRsp.readFromObject(_.assign({}, OK_RSP, ret));
  console.log(stRsp);
  current.sendResponse(OK_RET, stRsp);
};
```

```
const returnErr = (current, stRsp, err) => {
  console.log('xxxxxxxxxxxxxxx');
```

```
    logger.exception.error(err);
    stRsp.readFromObject(err.err);
    current.sendResponse(OK_RET, stRsp);
  };
  const reflectRun = async (current, stReq, stRsp, fn) => {
    try {
      const req = stReq.toObject();
      const ret = await fn(req);
      returnRet(current, stRsp, ret);
    } catch (err) {
      returnErr(current, stRsp, err);
    }
  };
```

8. 配置 taf 服务

```
  <taf>
      <application>
          <server>
              app=Test
              server=UserZjServer
              <Test.UserZjServer.UserZjObjAdapter>
                  allow
                  endpoint=tcp -h 127.0.0.1 -p 14001 -t 60000
                  protocol=taf
                  servant=Test.UserZjServer.UserZjObj
              </Test.UserZjServer.UserZjObjAdapter>
          </server>
          <client>
              modulename=Test.UserZjServer
          </client>
      </application>
  </taf>
```

9. taf 服务启动文件

```
  const svr = new Taf.server();
  const startSer = () => {
    svr.initialize(process.env.TAF_CONFIG || './server.conf', function
(server) {
      console.log('-------------------------------------------------
-----');
```

```
        const servantName = `${server.Application}.${server.ServerName}.
UserZjObj`;
        server.addServant(Test.UserImp, servantName);
        logger.data.info(`start servant: ${servantName}, ${process.pid}`)
;
      });

      svr.start();
    };
    const startup = async () => {
      try {
        await startSer();
      } catch (error) {
        process.exit(-1);
      }
    };
    process.on('uncaughtException', err => {
      logger.exception.error('uncaughtException', err);
    });
    process.on('unhandledRejection', (reason, p) => {
      logger.exception.error('unhandledRejection', reason, p);
    });
    startup();
```

10. 编写客户端

```
    let servant = 'Test.UserZjServer.UserZjObj';
    if (!process.env.TAF_CONFIG) {
        servant += '@tcp -h 127.0.0.1 -p 14001 -t 60000';
    }

    const prx = TAF.stringToProxy(Test.UserProxy, servant);
```

```
    const stReq = new Test.QueryIdReq();
    stReq.readFromObject(
        {
            uid: 1,
        }
    );
    prx.getUserDetailForAd(stReq).then(function (ret) {
        console.log('### helloWorld ok ###', ret.response.arguments.stRs
p.toObject());
    }, function (ret) {
```

```
    console.log('### helloWorld error ###', ret.response);
});
```

11. 启动服务端和客户端获取数据

1. 配置 package.json

```
"scripts": {
  "start": "nodemon  --exec babel-node -w src src/server.js",
  "test": "mocha --compile js:babel-register -r babel-polyfill --timeout 3500"
},
```

2. 启动服务端

```
PS D:\zijianzhang\whup\study\07tafUser\UserZjServer> npm start

> UserZjServer@1.0.0 start D:\zijianzhang\whup\study\07tafUser\UserZjServer
> nodemon  --exec babel-node -w src src/server.js

[nodemon] 2.0.1
[nodemon] to restart at any time, enter `rs`
[nodemon] watching dir(s): src\**\*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `babel-node src/server.js`
```

3. 启动客户端

```
PS D:\zijianzhang\whup\study\07tafUser\UserZjServer> node client.js
```

```
### helloWorld ok ### {
  iRet: 0,
  message: 'ok',
  userinfo: {
    uid: 1,
    username: '22',
    password: '22',
    gender: '22',
    tel: '22',
    age: 22
  }
}
```