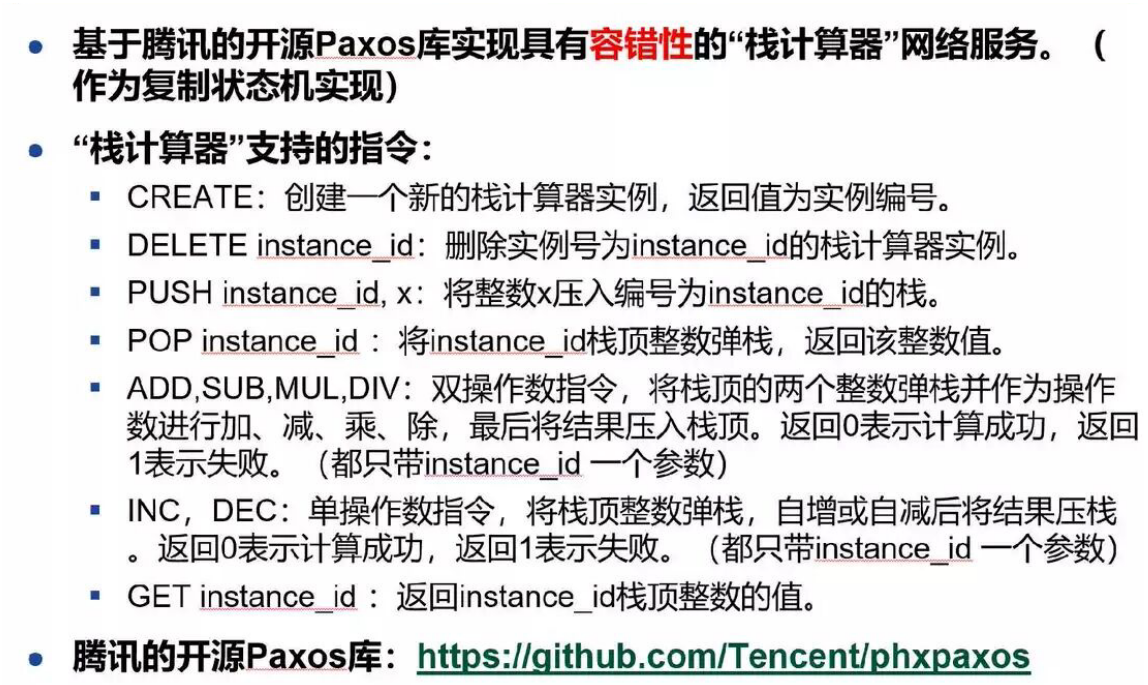
基于腾讯的开源 Paxos 库实现具有容错性的”栈计算器“网络服务

1. 实验要求



2. 实验过程

实验环境搭建：

由于实验需要运行在分布式环境中，因此，本实验在三台 ip 互通的Linux虚拟机中编译构建，服务器环境信息如下：

```
OS: Ubuntu 18.04 bionic
Kernel: x86_64 Linux 4.15.0-112-generic
Shell: zsh 5.4.2
CPU: Intel Xeon E5-2630 v2 @ 8x 2.6GHz
RAM: 7975MiB
gcc version 7.5.0 (Ubuntu 7.5.0-3ubuntu1~18.04)
GNU Make 4.1
```

由于本实验基于腾讯 Paxos ，因此需要先克隆 PhxPaxos 仓库到本地，并安装好 Paxos 依赖的第三方库：

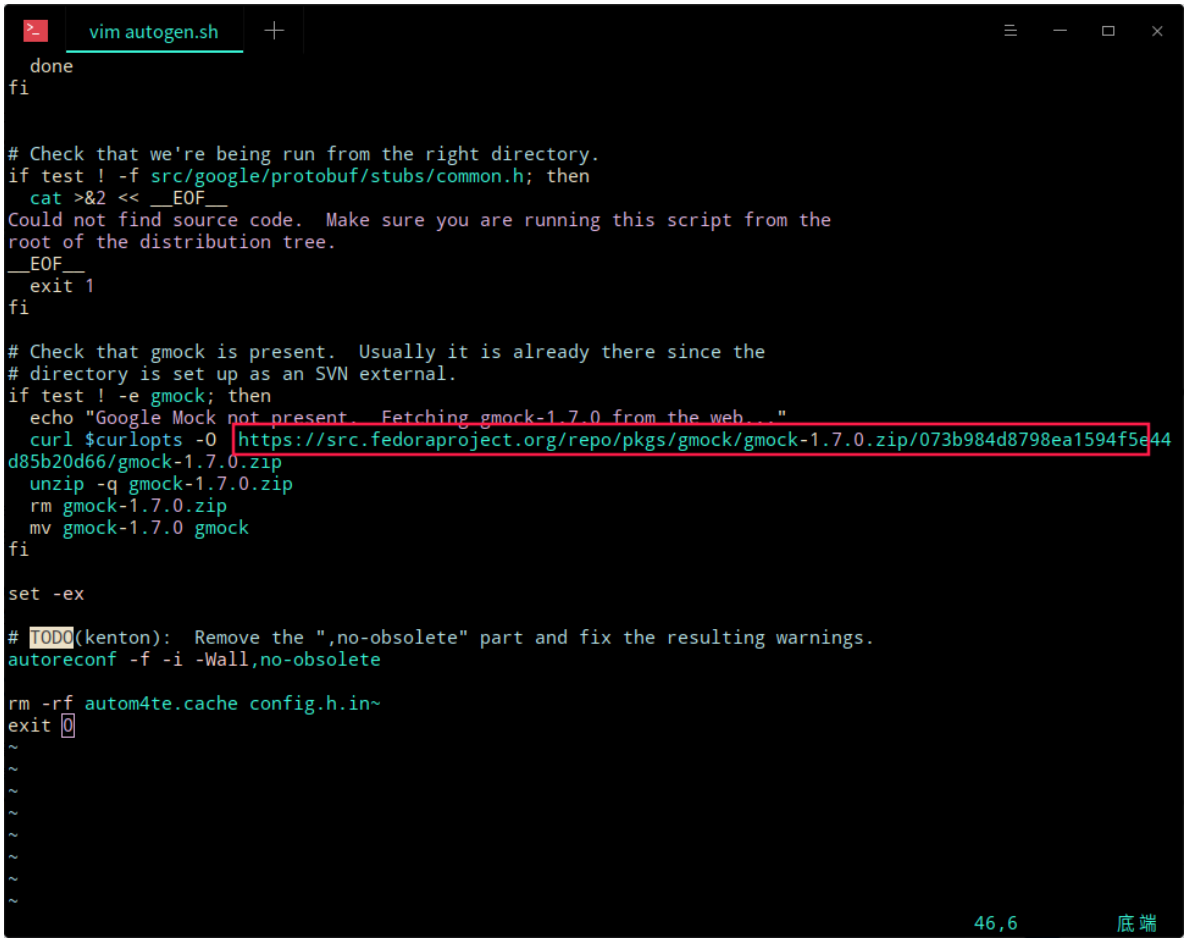
目录	编译对象	内部依赖	第三方库依赖
根目录	libphxpaxos.a	无	protobuf,leveldb
plugin	libphxpaxos_plugin.a	libphxpaxos.a	glog
sample/phxelection	可执行程序	libphxpaxos.a,libphxpaxos_plugin.a	无
sample/phxecho	可执行程序	libphxpaxos.a,libphxpaxos_plugin.a	无
sample/phxkv	可执行程序	libphxpaxos.a,libphxpaxos_plugin.a	grpc
src/ut	单元测试	无	gtest,gmock

git clone 时加上--recursive参数获取third_party目录下所有的submodule。

```
→ workspace git clone https://github.com/SincereXIA/phxpaxos.git --recursive
正克隆到 'phxpaxos'...
remote: Enumerating objects: 1134, done.
remote: Total 1134 (delta 0), reused 0 (delta 0), pack-reused 1134
接收对象中: 100% (1134/1134), 1.45 MiB | 772.00 KiB/s, 完成.
处理 delta 中: 100% (692/692), 完成.
子模组 'third_party/gflags' (https://github.com/gflags/gflags) 未对路径 'third_party/gflags' 注册
子模组 'third_party/glog' (https://github.com/google/glog) 未对路径 'third_party/glog' 注册
子模组 'third_party/gmock' (https://github.com/google/googlemock) 未对路径 'third_party/gmock' 注册
子模组 'third_party/leveldb' (https://github.com/google/leveldb) 未对路径 'third_party/leveldb' 注册
子模组 'third_party/protobuf' (https://github.com/google/protobuf) 未对路径 'third_party/protobuf' 注册
正克隆到 '/home/zhangjh/workspace/phxpaxos/third_party/gflags'...
remote: Enumerating objects: 23, done.
remote: Counting objects: 100% (23/23), done.
remote: Compressing objects: 100% (22/22), done.
remote: Total 2410 (delta 8), reused 8 (delta 1), pack-reused 2387
接收对象中: 100% (2410/2410), 1.52 MiB | 423.00 KiB/s, 完成.
处理 delta 中: 100% (1410/1410), 完成.
正克隆到 '/home/zhangjh/workspace/phxpaxos/third_party/gmock'...
remote: Enumerating objects: 2280, done.
remote: Total 2280 (delta 0), reused 0 (delta 0), pack-reused 2280
接收对象中: 100% (2280/2280), 1.22 MiB | 405.00 KiB/s, 完成.
处理 delta 中: 100% (1596/1596), 完成.
正克隆到 '/home/zhangjh/workspace/phxpaxos/third_party/leveldb'...
remote: Enumerating objects: 3102, done.
remote: Total 3102 (delta 0), reused 0 (delta 0), pack-reused 3102
接收对象中: 100% (3102/3102), 1.43 MiB | 419.00 KiB/s, 完成.
处理 delta 中: 100% (2189/2189), 完成.
正克隆到 '/home/zhangjh/workspace/phxpaxos/third_party/glog'...
remote: Enumerating objects: 2552, done.
remote: Total 2552 (delta 0), reused 0 (delta 0), pack-reused 2552
接收对象中: 100% (2552/2552), 1.70 MiB | 453.00 KiB/s, 完成.
处理 delta 中: 100% (1772/1772), 完成.
正克隆到 '/home/zhangjh/workspace/phxpaxos/third_party/protobuf'...
remote: Enumerating objects: 16, done.
remote: Counting objects: 100% (16/16), done.
remote: Compressing objects: 100% (16/16), done.
```

在 Paxos 的 third_party 目录中，有 autoinstall.sh 脚本文件，执行该脚本自动对所有依赖的第三方库配置编译参数和编译路径，执行make命令，生成这些依赖的第三方库的动态链接库。

在实际的编译过程中，项目使用的protobuf库版本过低，编译时在线下载的 gmock 库已经失效，手动修改 autogen.sh 文件，替换失效的 gmock 下载路径，完成依赖库的下载。



等待所有的第三方库编译完成后，开始编译 ibphxpaxos.a，该类库是 Paxos 的核心类库

在PhxPaxos根目录下

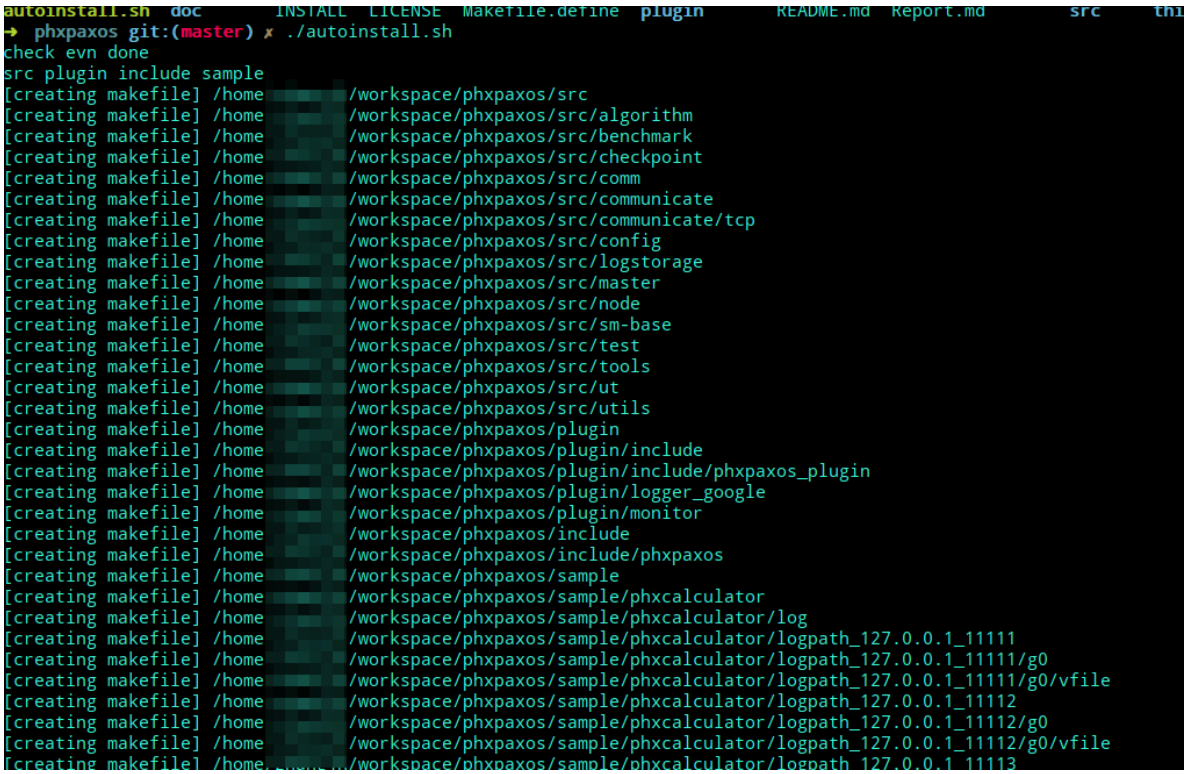
```
./autoinstall.sh
make
make install
```

接着编译 libphxpaxos_plugin.a 该类库负责 Paxos 运行时的日志信息

在plugin目录下

```
make
make install
```

完成编译之后，运行项目根目录下的 `autoinstall.sh`，该脚本自动遍历所有子目录下的 Makefile.define 文件，为每个目录自动生成编译所需要的 Makefile 文件，配置好第三方依赖库的头文件和位置信息。



为实验中三台 Linux 主机全部执行上述操作，至此，实验环境搭建完毕。

状态机的实现

PhxPaxos 提供的服务，本质上是在不同的机器上，维护相同的多个状态机副本，因此需要将项目所需的功能改造成状态机，定义状态机类 PhxCalculatorSM 如下：

```
class PhxCalculatorSM : public phxpaxos::StateMachine{
public:
    PhxCalculatorSM(int smid);

    bool Execute(const int iGroupIdx, const uint64_t llInstanceId,
                 const std::string & sPaxosValue, phxpaxos::SMCtx * poSMCtx);

    const int SMID() const {return smid;}

    int Create();
    int Delete(int instanceId);
    int Push(int instanceId, int x);
    int Pop(int instanceId);
    int Add(int instanceId);
    int Sub(int instanceId);
    int Mul(int instanceId);
    int Div(int instanceId);
    int Inc(int instanceId);
    int Dec(int instanceId);
    int Get(int instanceId);
    int Pop(int instanceId, bool &ret);

private:
    int smid = 0;
    int max_stack_id_now = 0;
    std::map<int, std::stack<int> *> m_instanceDict;

    int Calculate(std::vector<std::string> params);
```

```
    bool isInstanceExist(int instanceId);
};
```

其中Execute为状态机状态转移函数，输入为sPaxosValue， PhxPaxos保证多台机器都会执行相同系列的Execute(sPaxosValue)， 从而获得强一致性。SMID 函数获得状态机的编号，其余函数用于辅助状态机完成状态转换，实现题目要求的功能。

Execute 状态转移函数的实现如下：

```
bool PhxCalculatorSM::Execute(const int iGroupIdx, const uint64_t llInstanceId,
const std::string &sPaxosValue,
                              phxpaxos::SMCtx *poSMCtx){
    if (sPaxosValue.empty()) {
        return true;
    }
    cout <<"[COMMAND: "  << sPaxosValue << "]  ";
    std::vector<std::string> params;
    SplitString(sPaxosValue, params, " "); // 分割用空格隔开的参数
    transform(params[0].begin(), params[0].end(), params[0].begin(), ::toupper);
// 不区分大小写

    cout << Calculate(params) << endl;

    //only commiter node have SMCtx.
    if (poSMCtx != nullptr && poSMCtx->m_pCtx != nullptr)
    {
        PhxCalculatorSMCtx * poPhxEchoSMCtx = (PhxCalculatorSMCtx *)poSMCtx-
>m_pCtx;
        poPhxEchoSMCtx->iExecuteRet = 0;
        poPhxEchoSMCtx->sEchoRespValue = string("Finish");
    }

    return true;
}
```

该函数接受用户输入的指令，如 `PUSH 2 1` ，从中提取出参数，并交给 Calcuate 函数进行执行，Calculate 函数的实现如下：

```
int PhxCalculatorSM::Calculate(vector<string> params) {
    vector<pair<string, fp>> menu {
      pair<string, fp>("DELETE", &PhxCalculatorSM::Delete),
      pair<string, fp>("POP", &PhxCalculatorSM::Pop),
      pair<string, fp>("ADD", &PhxCalculatorSM::Add),
      pair<string, fp>("SUB", &PhxCalculatorSM::Sub),
      pair<string, fp>("MUL", &PhxCalculatorSM::Mul),
      pair<string, fp>("DIV", &PhxCalculatorSM::Div),
      pair<string, fp>("INC", &PhxCalculatorSM::Inc),
      pair<string, fp>("DEC", &PhxCalculatorSM::Dec),
      pair<string, fp>("GET", &PhxCalculatorSM::Get),

    };

    if (params.empty()) {
        //only commiter node have SMCtx.
        printf("EMPTY COMMAND\n");
```

```
        return 0;
    }
    if (params[0] == "CREATE") {
        int id = Create();
        printf("create new instance: %d\n", id);
    }else if (params[0] == "PUSH") {
        if (params.size() < 3) {
            printf("WRONG COMMAND\n");
            return -1;
        }
        int instanceId = atoi(params[1].c_str());
        int num = atoi(params[2].c_str());
        Push(instanceId, num);
        printf("PUSH %d to instance: %d\n", num, instanceId);
    }else {
        if (params.size() < 2) {
            cout << "WRONE COMMAND" << endl;
            return -1;
        }
        int instanceId = atoi(params[1].c_str());
        for (auto m : menu) {
            if (m.first == params[0]) {
                int rs =  (this->*(m.second))(instanceId);
                return rs;
            }
        }
        printf("COMMAND NOT FOUND\n");
        return -1;
    }
    return 0;
}
```

由于实现的功能较为复杂，需要处理的指令较多，因此函数先对所有支持的功能，使用键值对的方式，将指令名称和对应的函数指针保存在 menu 这个 vector 中，之后对数组进行遍历，就可以判断出需要执行的函数。避免了大量的 if else 判定分支。

以Push 为例，栈计算器的具体实现如下：

```
int PhxCalculatorSM::Push(int instanceId, int x) {
    if (!isInstanceExist(instanceId)) {
        return -1;
    }
    auto instance = m_instanceDict[instanceId];
    instance->push(x);
    return 0;
}
```

函数首先读取用户输入的实例ID，判断该实例是否存在，实例采用 Map 的方式进行存储：

```
std::map<int, std::stack<int> *> m_instanceDict;
```

若该实例存在，则通过下标的方式取出该实例，调用 stack 的 push 方法，实现 Push 一个整数入栈。

以相同的方法，实现剩余的全部十条指令，完成了状态机的实现。

构建 Paxos Server：

定义 PhxCalculatorServer 类，实现 Paxos 服务器，该类的定义如下：

```
class PhxCalculatorServer {
public:

    PhxCalculatorServer(const phxpaxos::NodeInfo &oMyNode, const
phxpaxos::NodeInfoList &vecNodeList);

    ~PhxCalculatorServer();

    int RunPaxos();

    int RunCommand(std::string &command, std::string &result);

    int MakeLogStoragePath(std::string &sLogStoragePath);

    phxpaxos::NodeInfo m_oMyNode; // 标识本机的IP/Port信息
    phxpaxos::NodeInfoList m_vecNodeList; // 标识集群信息

    phxpaxos::Node * m_poPaxosNode; // 本次需要运行的 PhxPaxos 示例指针
    PhxCalculatorSM m_oCalculatorSM; // 状态机
};
```

RunPaxos 方法对 Paxos 服务器进行初始化，设置好本机IP/PORT信息以及所有机器的信息。设置并初始化刚才实现的状态机。设置好日志函数，这里使用了plugin目录的glog日志方法。最后，调用Node::RunNode(oOptions, m_poPaxosNode)，传入参数选项，如果运行成功，函数返回0， 并且m_poPaxosNode指向这个运行中的PhxPaxos实例。这样PhxPaxos实例就运行成功了。

当读入用户输入之后，调用 PhxCalculatorServer 类的 RunCommand 命令进行处理，将用户输入的 command 提交到状态机进行运行：

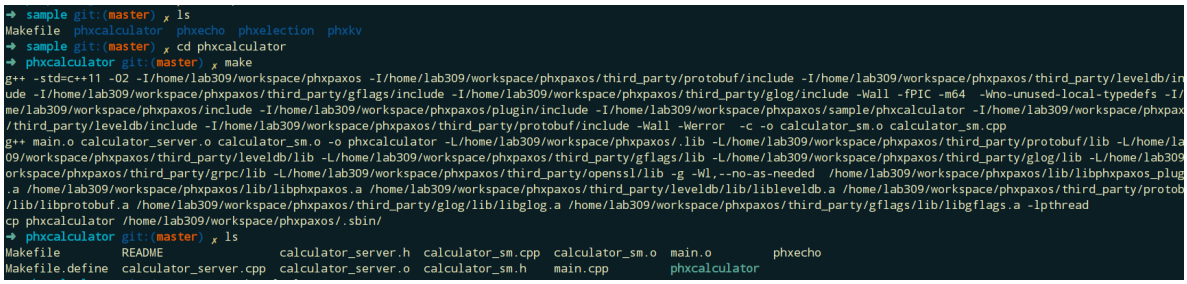
```
int PhxCalculatorServer::RunCommand(string &command, string &result) {
    SMCtx oCtx;
    PhxCalculatorSMCtx calculatorSmCtx; // 上下文

    int smid = 1; // 当前选择的状态机
    oCtx.m_iSMID = smid;
    oCtx.m_pCtx = (void *)&calculatorSmCtx;  // 将自定义上下文的指针送往状态机上下文中

    uint64_t llInstanceId = 1; // 实例编号
    int ret = m_poPaxosNode->Propose(0, command, llInstanceId, &oCtx);
    if (ret != 0) {
        printf("paxos propose fail, ret %d\n", ret);
        return ret;
    }
    if (calculatorSmCtx.iExecuteRet != 0)
    {
        printf("Calculator SM excute fail, %d \n", calculatorSmCtx.iExecuteRet);
        return  calculatorSmCtx.iExecuteRet;
    }
    result = calculatorSmCtx.sEchoRespValue.c_str();
    return 0;
}
```

配置编译脚本

编写 `Makefile.define` 文件如下：

```
allobject=phxcalculator

PHXCALCULATOR_OBJ= main.o calculator_server.o calculator_sm.o

PHXCALCULATOR_LIB=

PHXCALCULATOR_SYS_LIB=$(PHXPAXOS_LIB_PATH)/libphxpaxos_plugin.a
$(PHXPAXOS_LIB_PATH)/libphxpaxos.a $(LEVELDB_LIB_PATH)/libleveldb.a
$(PROTOBUF_LIB_PATH)/libprotobuf.a $(GLOG_LIB_PATH)/libglog.a
$(GFLAGS_LIB_PATH)/libgflags.a -lpthread

PHXCALCULATOR_INCS=$(SRC_BASE_PATH)/sample/phxcalculator
 $(PHXPAXOS_PLUGIN_PATH) $(PHXPAXOS_INCLUDE_PATH) $(LEVELDB_INCLUDE_PATH)
$(PROTOBUF_INCLUDE_PATH)

PHXCALCULATOR_EXTRA_CPPFLAGS=-Wall -Werror
```

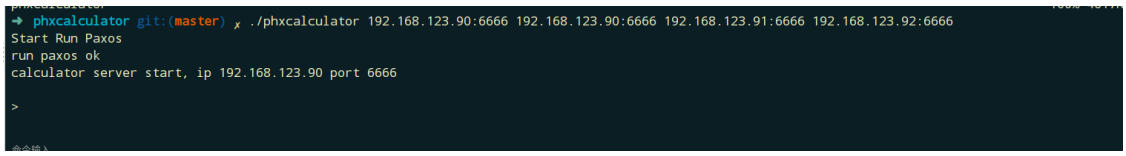
之后，执行工程根目录下的 autoinstall.sh 生成 Makefile 文件，运行make命令完成编译：



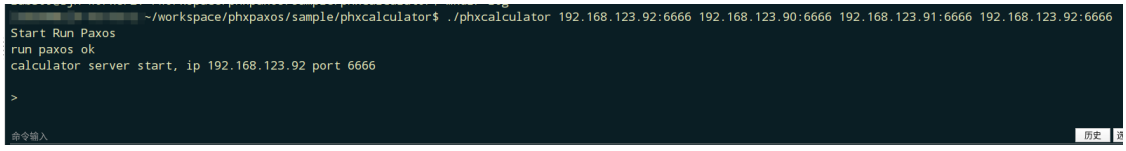
3. 实验结果

- 三台服务器分别运行分布式栈计算器：

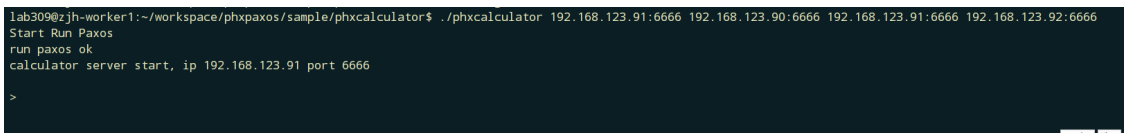
  第一台服务器执行命令：`./phxcalculator 192.168.123.90:6666 192.168.123.90:6666 192.168.123.91:6666 192.168.123.92:6666`

  第二台服务器执行命令：`./phxcalculator 192.168.123.92:6666 192.168.123.90:6666 192.168.123.91:6666 192.168.123.92:6666`

  第三台服务器执行命令：`./phxcalculator 192.168.123.91:6666 192.168.123.90:6666 192.168.123.91:6666 192.168.123.92:6666`

- 输入指令，创建栈实例，并进行运算：

在用户输入端的交互显示：

```
> create
[COMMAND: create]  create new instance: 1
0
Finish

> create
[COMMAND: create ]  create new instance: 2
0
Finish

> push 2 3
[COMMAND: push 2 3]  PUSH 3 to instance: 2
0
Finish

> push 2 7
[COMMAND: push 2 7]  PUSH 7 to instance: 2
0
Finish

> mul 2
[COMMAND: mul 2]  0
Finish

> get 2
[COMMAND: get 2]  [RESULT] 21
Finish

> pop 2
[COMMAND: pop 2]  21
Finish

>
```

在其它机器上的同步运算：

```
[redacted]~/workspace/phxpaxos/sample/phxcalculator$ ./phxcalculator
Start Run Paxos
run paxos ok
calculator server start, ip 192.168.123.91 port 6666

> [COMMAND: create]  create new instance: 1
0
[COMMAND: create ]  create new instance: 2
0
[COMMAND: push 2 3]  PUSH 3 to instance: 2
0
[COMMAND: push 2 7]  PUSH 7 to instance: 2
0
[COMMAND: mul 2]  0
[COMMAND: get 2]  [RESULT] 21
[COMMAND: pop 2]  21
```

通过实验，可以验证，成功构建出来一个基于 Paxos 的分布式栈计算器，使得单机的计算状态拓展到了多机器，保证了计算的一致性，当少量节点失效时，仍然能继续完成运算，实现了自动容灾的特性。