# PBFT算法实验指导书

一、实验课时

4课时

二、实验内容

理解PBFT算法的工作原理和流程，能够使用编程语言实现简单的PBFT算法。

三、实验环境

GO开发环境：go1.20.4

IDE工具：vscode

操作系统：windows11

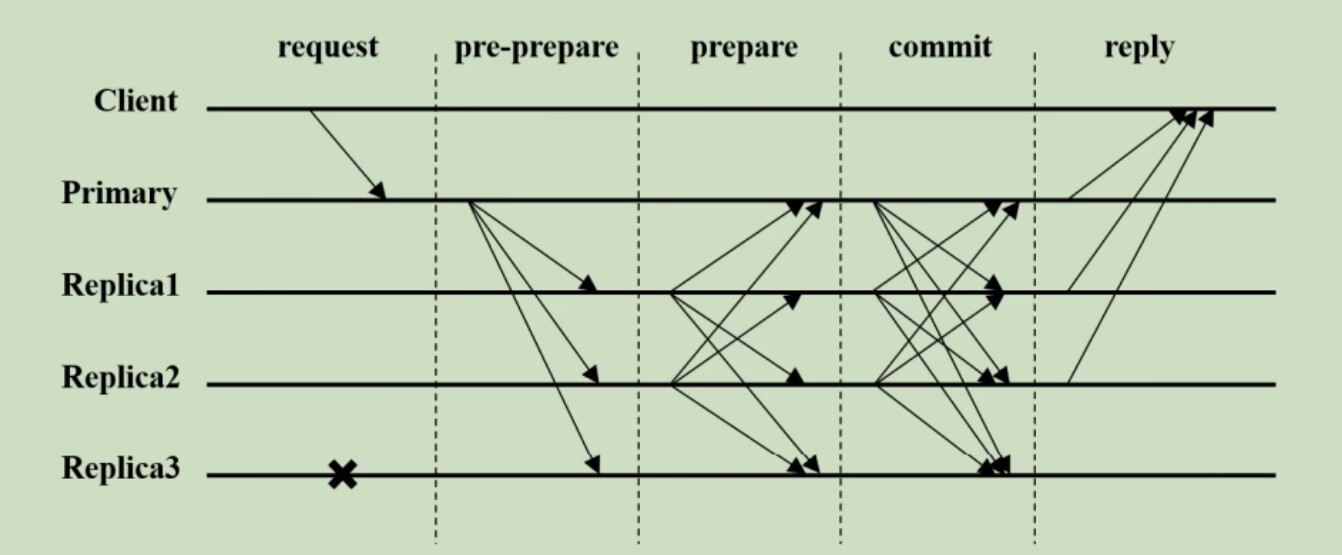
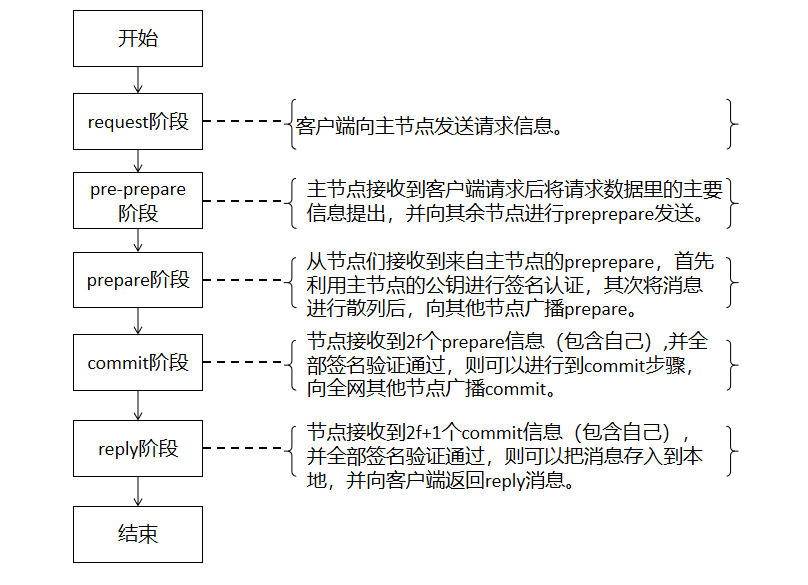
四、算法描述

PBFT(Practical Byzantine Fault Tolerance) 是一种分布式系统中的一致性算法，用于解决拜占庭将军问题。PBFT 算法是基于拜占庭容错性的，它可以在有多个节点的情况下，保证数据的一致性。

PBFT 算法的消息传递过程包括四个阶段：准备阶段、传播阶段、确认阶段和终止阶段。在准备阶段，节点准备好消息并等待其他节点响应。在传播阶段，节点将消息传递给其他节点。在确认阶段，收到消息的节点向其他节点发送确认消息。在终止阶段，所有节点完成任务并终止。

1. 实验过程

PBFT算法的流程如下：



数据结构：

// 本地消息池（模拟持久化层），只有确认提交成功后才会存入此池

var localMessagePool = []Message{}

type node struct {

    //节点ID

    nodeID string

    //节点监听地址

    addr string

    //RSA私钥

    rsaPrivKey []byte

    //RSA公钥

    rsaPubKey []byte

}

type pbft struct {

    //节点信息

    node node

    //每笔请求自增序号

    sequenceID int

    //锁

    lock sync.Mutex

    //临时消息池，消息摘要对应消息本体

    messagePool map[string]Request

    //存放收到的prepare数量(至少需要收到并确认2f个)，根据摘要来对应

    prePareConfirmCount map[string]map[string]bool

    //存放收到的commit数量(至少需要收到并确认2f+1个)，根据摘要来对应

    commitConfirmCount map[string]map[string]bool

    //该笔消息是否已进行Commit广播

    isCommitBordcast map[string]bool

    //该笔消息是否已对客户端进行Reply

    isReply map[string]bool

}

1. request阶段:客户端向主节点发送请求信息。

func (p \*pbft) handleClientRequest(content []byte) {

    fmt.Println("主节点已接收到客户端发来的request ...")

    //使用json解析出Request结构体

    r := new(Request)

    err := json.Unmarshal(content, r)

    if err != nil {

        log.Panic(err)

    }

    //添加信息序号

    p.sequenceIDAdd()

    //获取消息摘要

    digest := getDigest(\*r)

    fmt.Println("收到的request消息为: ", r.Message)

    fmt.Println("已将request存入临时消息池")

    //存入临时消息池

    p.messagePool[digest] = \*r

    //主节点对消息摘要进行签名

    digestByte, \_ := hex.DecodeString(digest)

    signInfo := p.RsaSignWithSha256(digestByte, p.node.rsaPrivKey)

    //拼接成PrePrepare，准备发往follower节点

    pp := PrePrepare{\*r, digest, p.sequenceID, signInfo}

    b, err := json.Marshal(pp)

    if err != nil {

        log.Panic(err)

    }

    pause()

    fmt.Println("正在向其他节点进行进行PrePrepare广播 ...")

    fmt.Println("PrePrepare消息内容为: ", pp)

    //进行PrePrepare广播

    p.broadcast(cPrePrepare, b)

    fmt.Println("PrePrepare广播完成")

    pause()

}

1. pre-prepare阶段：主节点N0接收到客户端请求后将请求数据里的主要信息提出，并向其余节点进行preprepare发送。

func (p \*pbft) handlePrePrepare(content []byte) {

    fmt.Println("本节点已接收到主节点发来的PrePrepare ...")

    pause()

    //  //使用json解析出PrePrepare结构体

    pp := new(PrePrepare)

    err := json.Unmarshal(content, pp)

    if err != nil {

        log.Panic(err)

    }

    //获取主节点的公钥，用于数字签名验证

    primaryNodePubKey := p.getPubKey("N0")

    digestByte, \_ := hex.DecodeString(pp.Digest)

    if digest := getDigest(pp.RequestMessage); digest != pp.Digest {

        fmt.Println("信息摘要对不上，拒绝进行prepare广播")

    } else if p.sequenceID+1 != pp.SequenceID {

        fmt.Println("消息序号对不上，拒绝进行prepare广播")

    } else if !p.RsaVerySignWithSha256(digestByte, pp.Sign, primaryNodePubKey) {

        fmt.Println("主节点签名验证失败！,拒绝进行prepare广播")

    } else {

        //序号赋值

        p.sequenceID = pp.SequenceID

        //将信息存入临时消息池

        fmt.Println("已将消息存入临时节点池")

        p.messagePool[pp.Digest] = pp.RequestMessage

        //节点使用私钥对其签名

        sign := p.RsaSignWithSha256(digestByte, p.node.rsaPrivKey)

        //拼接成Prepare

        pre := Prepare{pp.Digest, pp.SequenceID, p.node.nodeID, sign}

        bPre, err := json.Marshal(pre)

        if err != nil {

            log.Panic(err)

        }

        //进行准备阶段的广播

        fmt.Println("正在进行Prepare广播 ...")

        fmt.Println("广播的Prepare消息内容为: ", pre)

        p.broadcast(cPrepare, bPre)

        fmt.Println("Prepare广播完成")

        reader := bufio.NewReader(os.Stdin)

        fmt.Print("Press enter to continue...")

        \_, \_ = reader.ReadString('\n')

    }

}

1. prepare阶段：从节点们接收到来自主节点的preprepare，首先利用主节点的公钥进行签名认证，其次将消息进行散列（消息摘要，以便缩小信息在网络中的传输大小）后，向其他节点广播prepare。

func (p \*pbft) handlePrepare(content []byte) {

    //使用json解析出Prepare结构体

    pre := new(Prepare)

    err := json.Unmarshal(content, pre)

    if err != nil {

        log.Panic(err)

    }

    fmt.Printf("本节点已接收到%s节点发来的Prepare ... \n", pre.NodeID)

    //获取消息源节点的公钥，用于数字签名验证

    MessageNodePubKey := p.getPubKey(pre.NodeID)

    digestByte, \_ := hex.DecodeString(pre.Digest)

    if \_, ok := p.messagePool[pre.Digest]; !ok {

        fmt.Println("当前临时消息池无此摘要，拒绝执行commit广播")

    } else if p.sequenceID != pre.SequenceID {

        fmt.Println("消息序号对不上，拒绝执行commit广播")

    } else if !p.RsaVerySignWithSha256(digestByte, pre.Sign, MessageNodePubKey) {

        fmt.Println("节点签名验证失败！,拒绝执行commit广播")

    } else {

        p.setPrePareConfirmMap(pre.Digest, pre.NodeID, true)

        count := 0

        for range p.prePareConfirmCount[pre.Digest] {

            count++

        }

        //因为主节点不会发送Prepare，所以不包含自己

        specifiedCount := 0

        if p.node.nodeID == "N0" {

            specifiedCount = nodeCount / 3 \* 2

        } else {

            specifiedCount = (nodeCount / 3 \* 2) - 1

        }

        //如果节点至少收到了2f个prepare的消息（包括自己）,并且没有进行过commit广播，则进行commit广播

        p.lock.Lock()

        //获取消息源节点的公钥，用于数字签名验证

        if count >= specifiedCount && !p.isCommitBordcast[pre.Digest] {

            pause()

            fmt.Println("本节点已收到至少2f个节点(包括本地节点)发来的Prepare信息，内容为： ", pre)

            //节点使用私钥对其签名

            sign := p.RsaSignWithSha256(digestByte, p.node.rsaPrivKey)

            c := Commit{pre.Digest, pre.SequenceID, p.node.nodeID, sign}

            bc, err := json.Marshal(c)

            if err != nil {

                log.Panic(err)

            }

            //进行提交信息的广播

            fmt.Println("正在进行commit广播 ...")

            fmt.Println("广播的commit消息内容为: ", bc)

            p.broadcast(cCommit, bc)

            p.isCommitBordcast[pre.Digest] = true

            fmt.Println("commit广播完成")

        }

        p.lock.Unlock()

        pause()

    }

}

1. commit阶段：节点接收到2f个prepare信息（包含自己）,并全部签名验证通过，则可以进行到commit步骤，向全网其他节点广播commit。

func (p \*pbft) handleCommit(content []byte) {

    //使用json解析出Commit结构体

    c := new(Commit)

    err := json.Unmarshal(content, c)

    if err != nil {

        log.Panic(err)

    }

    fmt.Printf("本节点已接收到%s节点发来的Commit ... \n", c.NodeID)

    //获取消息源节点的公钥，用于数字签名验证

    MessageNodePubKey := p.getPubKey(c.NodeID)

    digestByte, \_ := hex.DecodeString(c.Digest)

    if \_, ok := p.prePareConfirmCount[c.Digest]; !ok {

        fmt.Println("当前prepare池无此摘要，拒绝将信息持久化到本地消息池")

    } else if p.sequenceID != c.SequenceID {

        fmt.Println("消息序号对不上，拒绝将信息持久化到本地消息池")

    } else if !p.RsaVerySignWithSha256(digestByte, c.Sign, MessageNodePubKey) {

        fmt.Println("节点签名验证失败！,拒绝将信息持久化到本地消息池")

    } else {

        p.setCommitConfirmMap(c.Digest, c.NodeID, true)

        count := 0

        for range p.commitConfirmCount[c.Digest] {

            count++

        }

        //如果节点至少收到了2f+1个commit消息（包括自己）,并且节点没有回复过,并且已进行过commit广播，则提交信息至本地消息池，并reply成功标志至客户端！

        p.lock.Lock()

        if count >= nodeCount/3\*2 && !p.isReply[c.Digest] && p.isCommitBordcast[c.Digest] {

            fmt.Println("本节点已收到至少2f + 1 个节点(包括本地节点)发来的Commit信息 ...")

            //将消息信息，提交到本地消息池中！

            localMessagePool = append(localMessagePool, p.messagePool[c.Digest].Message)

            info := ""

            if p.node.nodeID != "N0" {

                info = p.node.nodeID + "节点已将msgid:" + strconv.Itoa(p.messagePool[c.Digest].ID) + "存入本地消息池中,消息内容为：" + p.messagePool[c.Digest].Content

            } else {

                info = "主节点已将msgid:" + strconv.Itoa(p.messagePool[c.Digest].ID) + "存入本地消息池中,消息内容为：" + p.messagePool[c.Digest].Content

            }

            fmt.Println(info)

            fmt.Println("正在reply客户端 ...")

            tcpDial([]byte(info), p.messagePool[c.Digest].ClientAddr)

            p.isReply[c.Digest] = true

            fmt.Println("reply完毕")

        }

        p.lock.Unlock()

    }

}

1. reply阶段：节点接收到2f+1个commit信息（包含自己），并全部签名验证通过，则可以把消息存入到本地，并向客户端返回reply消息。

fmt.Println(info)

            fmt.Println("正在reply客户端 ...")

            tcpDial([]byte(info), p.messagePool[c.Digest].ClientAddr)

            p.isReply[c.Digest] = true

            fmt.Println("reply完毕")

## 程序执行过程：

首先切换到项目根路径，分别运行下面两段代码进行初始化：

 go mod init pbft

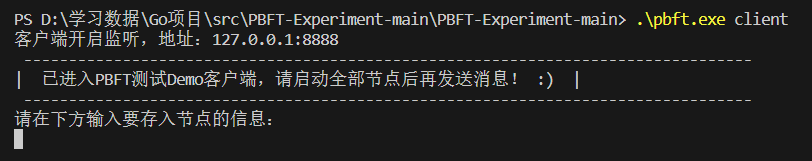
 go build -o pbft.exe

开启五个端口（一个客户端，四个节点）

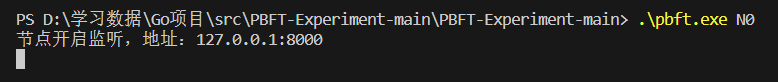
客户端执行 .\pbft.exe client

其他四个节点依次执行 .\pbft.exe N0  .\pbft.exe N1  .\pbft.exe N2  .\pbft.exe N3

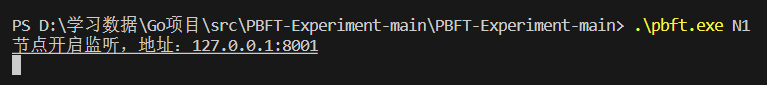
客户端运行



主节点运行



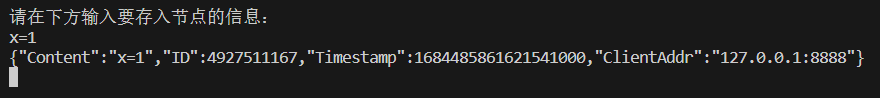
三个从节点运行：



1.系统无拜占庭节点，均正常运行

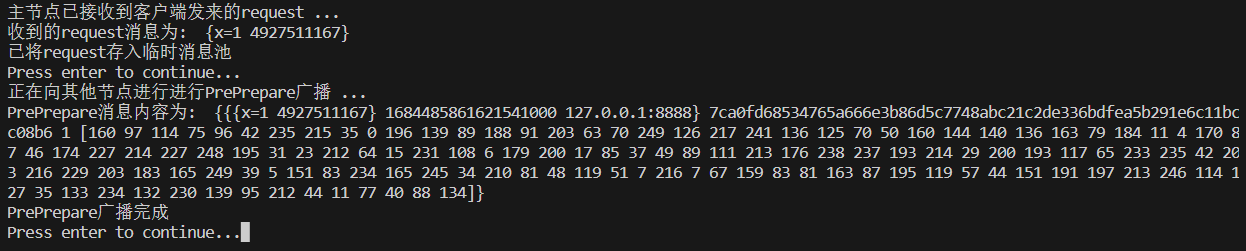
1.1 request阶段：

客户端运行：



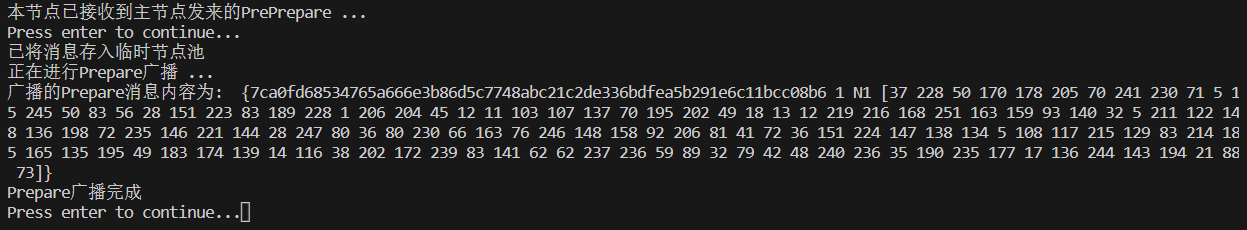
1.2 pre-prepare阶段：

主节点运行：

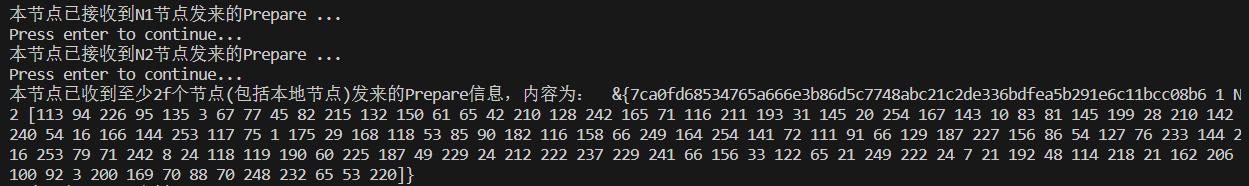


1.3 prepare阶段

三个从节点运行

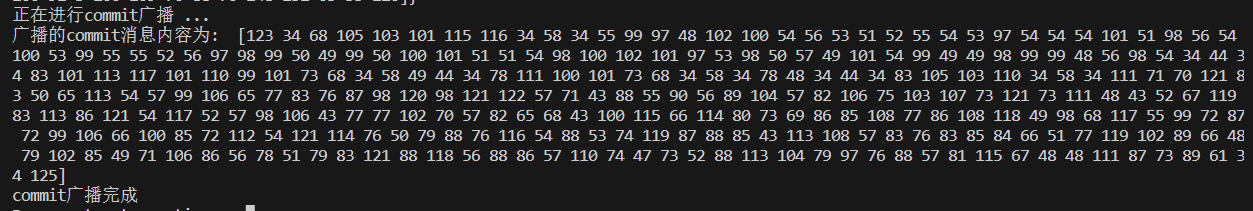


主节点运行

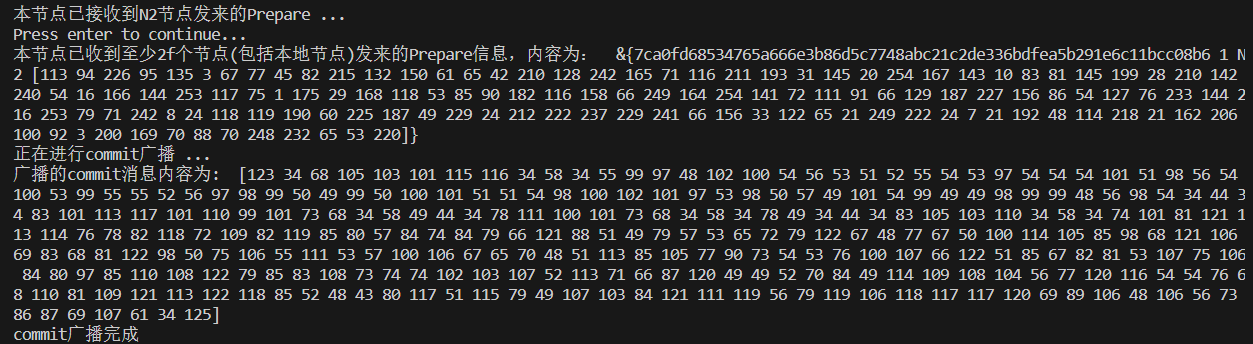


1.4 commit阶段

主节点运行：

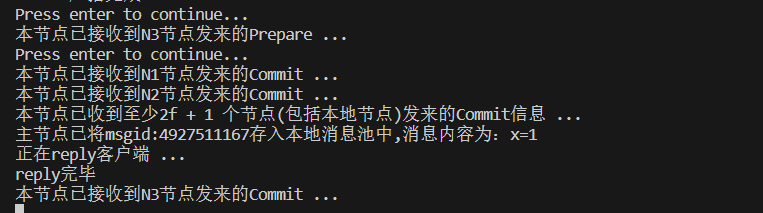


三个从节点运行：

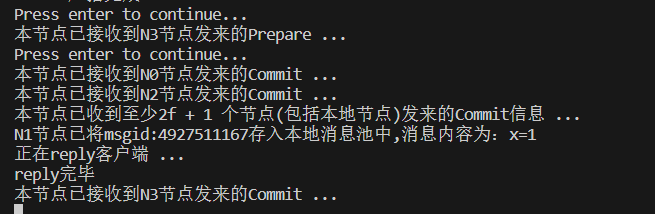


1.5 reply阶段

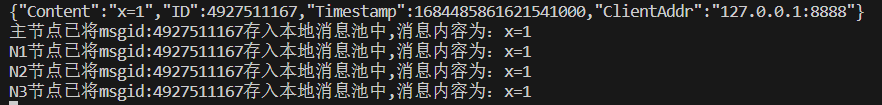
主节点运行：



三个从节点运行：



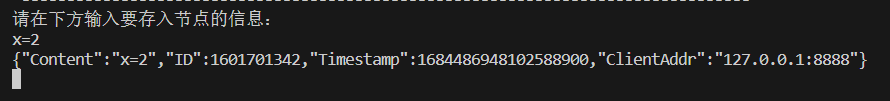
客户端收到消息：



2.系统存在一个拜占庭节点，其余两个从节点均正常运行

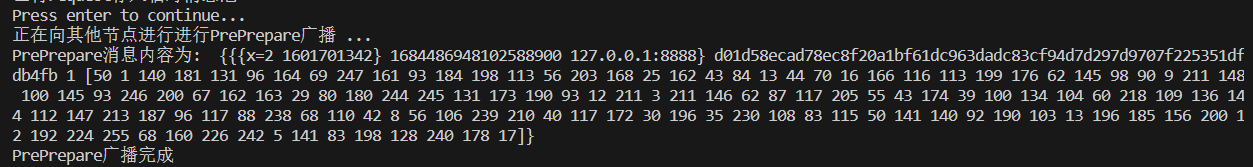
2.1 request阶段：

客户端运行：



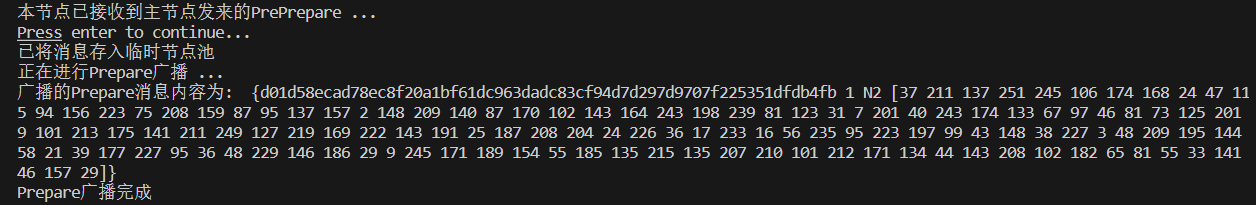
2.2 pre-prepare阶段：

主节点运行：

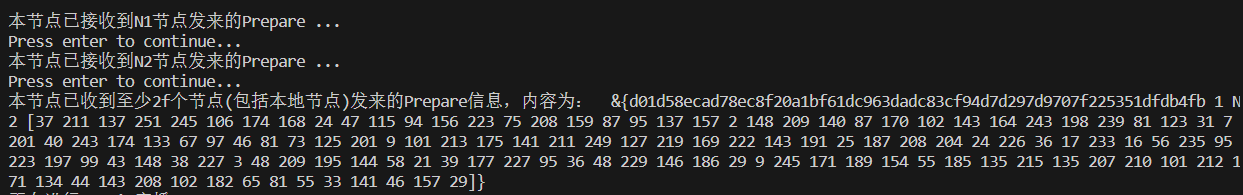


2.3 prepare阶段

三个从节点运行

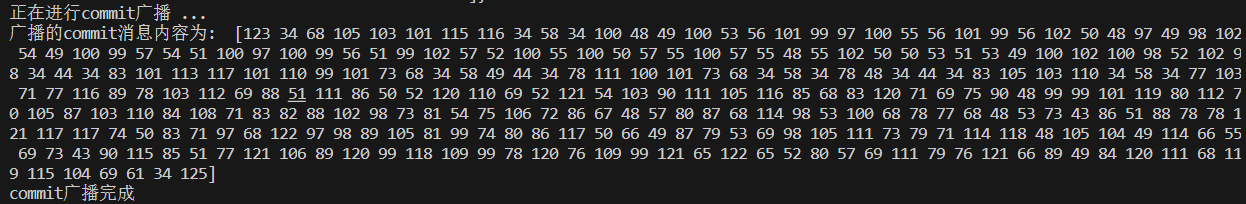


主节点运行

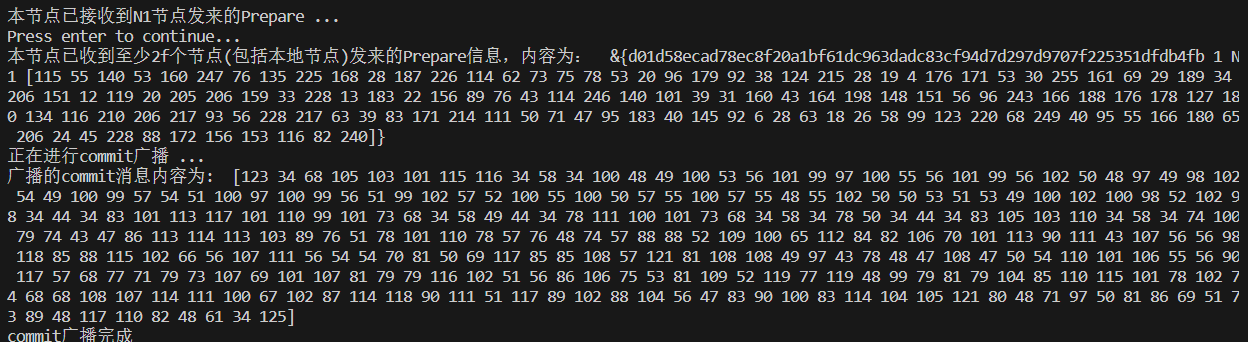


2.4 commit阶段

主节点运行：

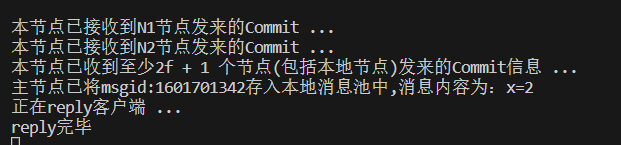


三个从节点运行：

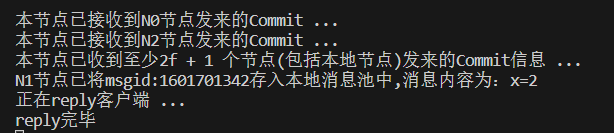


2.5 reply阶段

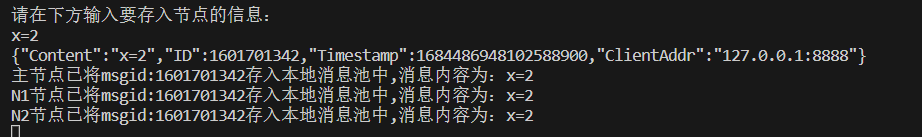
主节点运行：



三个从节点运行：



客户端收到消息：

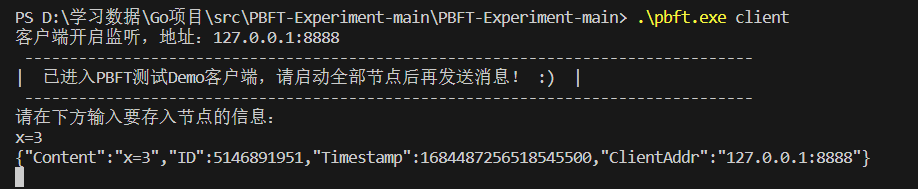


**可以看到，客户端依然会接收到reply，因为根据公式 n >= 3f+1 ，就算宕机一个节点，系统依然能顺利运行**

3.系统存在两个拜占庭节点，剩下一个从节点正常运行

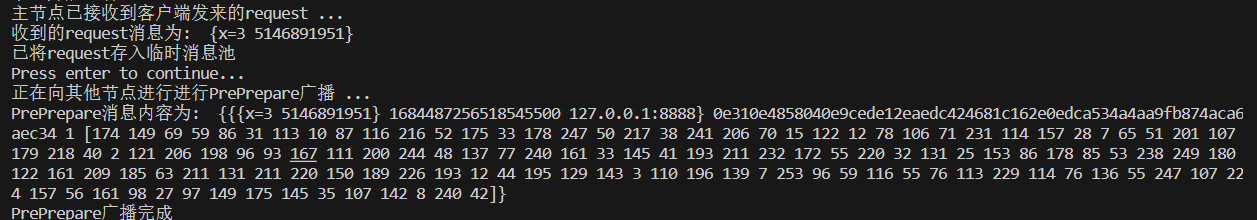
3.1 request阶段：

客户端运行：



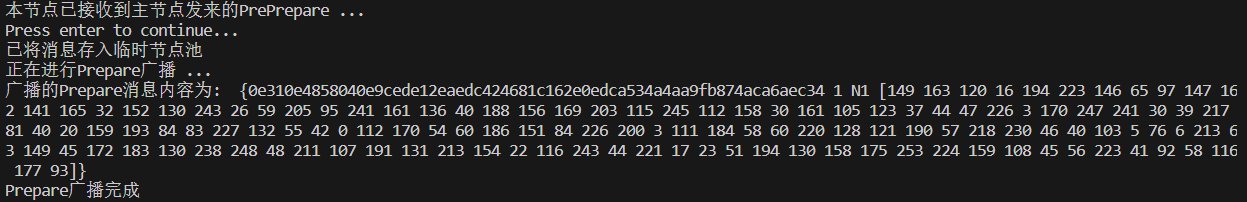
3.2 pre-prepare阶段：

主节点运行：

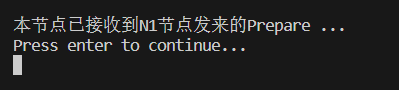


3.3 prepare阶段

三个从节点运行



主节点运行



**可以看到，关闭两个节点后，故障节点已经超出了pbft的允许数量，消息进行到Prepare阶段由于接收不到满足数量的信息，固系统不再进行commit确认,客户端也接收不到reply**