

## 参考

Kademlia 详解: [https://blog.csdn.net/qq\\_26720653/article/details/106496916](https://blog.csdn.net/qq_26720653/article/details/106496916)

## 代码

```
package main

import (
    "crypto/rand"
    "crypto/sha1"
    "encoding/base64"
    "encoding/hex"
    "fmt"
    "math/big"
)

type Node struct {
    nodeID string
    buckets [160]*Bucket
    keys    map[string][]byte
}

type Bucket struct {
    ids []string
}

var nodesMap map[string]*Node

var SetNodes []string
var GetNodes []string

func compareGetMin(targetValue, value1, value2 string) string {
    num := new(big.Int)
    num1 := new(big.Int)
    num2 := new(big.Int)
    num.SetString(targetValue, 2)
    num1.SetString(value1, 2)
    num2.SetString(value2, 2)
    //计算出距离
    result1 := new(big.Int)
```

```

    result1.Xor(num, num1)
    result2 := new(big.Int)
    result2.Xor(num, num2)

    if result1.Cmp(result2) < 0 {
        return value1
    } else {
        return value2
    }
}
}

// 生成两个随机数, 0~2 之间
func GetRandom2() (int, int) {
    var nums [2]int
    // 随机生成两个不重复的整数
    for i := range nums {
        num, err := rand.Int(rand.Reader, big.NewInt(3))
        if err != nil {
            // 处理错误
            return -1, -1
        }
        nums[i] = int(num.Int64())
    }
    for nums[0] == nums[1] {
        num, err := rand.Int(rand.Reader, big.NewInt(3))
        if err != nil {
            // 处理错误
            fmt.Println(err)
            return -1, -1
        }
        nums[1] = int(num.Int64())
    }
    return nums[0], nums[1]
}
}

```

/\*\*

## 1. 插入节点

计算插入的节点距离应该加到那个桶

- 如果是距离最近的桶（数组最后一个元素）
  - 查看桶满没有，没满就加入
  - 满了的话就分裂
    - 在数组中加入新的桶，新的桶中永远保存距离最近的 n 个节点，分裂前满的桶装距离远的节点
- 如果是非最近的桶

```

- 桶满就放弃加入
- 桶未就加入
*/

func (s *Node) InsertNode(nodeId string) bool {
    if s.nodeID == nodeId {
        return false
    }
    new_node := Node{nodeID: nodeId}
    result := findBucket(s.nodeID, nodeId)
    if result < 0 {
        return false
    }
    var index int
    index = result
    bucket := s.buckets[index]
    if bucket == nil {
        s.buckets[index] = new(Bucket)
    }
    insertInto(index, new_node.nodeID, s)
    return true
}

func insertInto(index int, newNode string, targetNode *Node) {

    bucket := targetNode.buckets[index]
    for _, v := range bucket.ids {
        if v == newNode {
            return
        }
    }
    //小于三个就更新，满了就不管（简化），事实上要进行心跳监测
    if len(bucket.ids) < 3 {
        bucket.ids = append(bucket.ids, newNode)
    }
}

// 寻找属于第几个桶
func findBucket(selfId, targetId string) int {
    num1 := new(big.Int)
    num2 := new(big.Int)
    num1.SetString(selfId, 2)
    num2.SetString(targetId, 2)

```

```

    result := new(big.Int)
    result.Xor(num1, num2)
    return (160 - len(fmt.Sprintf("%b", result)))
}

// 打印桶中的 id
func (s *Bucket) printBucketContents() {
    for _, v := range s.ids {
        fmt.Printf("nodeID = %s \n", v)
    }
}

// 查看是否有相同的元素
func isDuplicate(binaryStr string, binaryStrs []string) bool {
    // 将二进制字符串转换为大整数类型
    num := new(big.Int)
    num.SetString(binaryStr, 2)

    // 判断这个大整数是否已经存在
    for _, str := range binaryStrs {
        n := new(big.Int)
        n.SetString(str, 2)
        if n.Cmp(num) == 0 {
            return true
        }
    }

    return false
}

func testInsert() {
    var binaryStrs []string
    for len(binaryStrs) < 101 {
        max := new(big.Int).Sub(new(big.Int).Lsh(big.NewInt(1), 160),
big.NewInt(1))
        // 生成一个 160 位的随机二进制字符串
        num, _ := rand.Int(rand.Reader, max)
        binaryStr := fmt.Sprintf("%0160b", num)

        // 检查这个二进制字符串是否已经存在
        if !isDuplicate(binaryStr, binaryStrs) {
            binaryStrs = append(binaryStrs, binaryStr)
        }
    }
}

```

```

node := Node{nodeID: binaryStrs[0]}
fmt.Println("nodeID = ", node.nodeID)
for i, v := range binaryStrs {

    if i == 0 {
        continue
    }
    node.InsertNode(v)
}

for i, v := range node.buckets {
    if v == nil {
        continue
    }
    fmt.Printf("buckets num is = %d \n", i)
    v.printBucketContents()
    fmt.Println("-----")
}
}

func (s *Node) SetValue(key string, value []byte) bool {
    hash := sha1.Sum(value)
    hash_str := hex.EncodeToString(hash[:])
    if key != hash_str {
        return false
    }
    if s.keys[key] != nil {
        return true
    }

    //将内容存入自己的节点中
    s.keys[key] = value

    //获取到最近的桶
    keyInt := new(big.Int)
    str, _ := keyInt.SetString(key, 16)
    keyBinary := fmt.Sprintf("%160b", str)
    result := findBucket(s.nodeID, keyBinary)
    var bucket *Bucket
    bucket = s.buckets[result]
    if bucket == nil {
        return false
    }
}

```

```

//查看找到的新的节点是否为最近的节点，如果是就进行递归，不是的话就停止递归
index1, index2 := checkLen(len(bucket.ids))
var flag1, flag2 bool
if index2 != -1 {
    var maxStr string
    minStr := compareGetMin(keyBinary, bucket.ids[index1],
bucket.ids[index2])
    if minStr == bucket.ids[index1] {
        maxStr = bucket.ids[index2]
    } else {
        maxStr = bucket.ids[index1]
    }
    updateIndex := isUpdated(keyBinary, SetNodes, minStr)
    if updateIndex == -1 {
        return false
    } else {
        SetNodes[updateIndex] = minStr
        flag1 = nodesMap[minStr].SetValue(key, value)
    }
    updateIndexMax := isUpdated(keyBinary, SetNodes, maxStr)
    if updateIndexMax != -1 {
        SetNodes[updateIndexMax] = maxStr
        flag2 = nodesMap[maxStr].SetValue(key, value)
    }
    if flag1 || flag2 {
        return true
    } else {
        return false
    }
} else {
    var flag bool
    updateIndex := isUpdated(keyBinary, SetNodes, bucket.ids[0])
    if updateIndex == -1 {
        return false
    } else {
        SetNodes[updateIndex] = bucket.ids[0]
        flag = nodesMap[bucket.ids[0]].SetValue(key, value)
    }
    if flag {
        return true
    } else {
        return false
    }
}

```

```

    }
}

func (s *Node) GetValue(key string) []byte {
    if s.keys[key] != nil {
        hash := sha1.Sum(s.keys[key])
        hash_str := hex.EncodeToString(hash[:])
        if key != hash_str {
            return nil
        }
        return s.keys[key]
    }

    //获取到最近的桶
    keyInt := new(big.Int)
    str, _ := keyInt.SetString(key, 16)
    keyBinary := fmt.Sprintf("%160b", str)
    result := findBucket(s.nodeID, keyBinary)
    var bucket *Bucket
    bucket = s.buckets[result]
    if bucket == nil {
        return nil
    }

    index1, index2 := checkLen(len(bucket.ids))
    if index2 != -1 {
        var maxStr string
        var value1 []byte
        var value2 []byte
        minStr := compareGetMin(keyBinary, bucket.ids[index1],
bucket.ids[index2])
        if minStr == bucket.ids[index1] {
            maxStr = bucket.ids[index2]
        } else {
            maxStr = bucket.ids[index1]
        }
        updateIndex := isUpdated(keyBinary, GetNodes, minStr)
        if updateIndex == -1 {
            return nil
        } else {
            GetNodes[updateIndex] = minStr
            value1 = nodesMap[minStr].GetValue(key)
        }
    }
}

```

```

updateIndexMax := isUpdated(keyBinary, GetNodes, maxStr)
if updateIndexMax != -1 {
    GetNodes[updateIndexMax] = maxStr
    value2 = nodesMap[maxStr].GetValue(key)
}

if value1 == nil && value2 == nil {
    return nil
} else if value1 != nil && value2 != nil {
    hash1 := sha1.Sum(value1)
    hash1_str := hex.EncodeToString(hash1[:])
    hash2 := sha1.Sum(value2)
    hash2_str := hex.EncodeToString(hash2[:])
    if key != hash1_str && key != hash2_str {
        return nil
    } else {
        if key != string(hash1[:]) {
            return value2
        }
        return value1
    }
} else {
    var hash_str string
    var value []byte
    if value1 == nil {
        hash := sha1.Sum(value2)
        hash_str = hex.EncodeToString(hash[:])
        value = value2
    } else {
        hash := sha1.Sum(value1)
        hash_str = hex.EncodeToString(hash[:])
        value = value1
    }
    if key != hash_str {
        return nil
    }
    return value
}
} else {
    updateIndex := isUpdated(key, GetNodes, bucket.ids[0])
    if updateIndex == -1 {
        return nil
    }
}

```



```

        GetNodes[updateIndex] = bucket.ids[0]
        value := nodesMap[bucket.ids[0]].GetValue(key)
        if value == nil {
            return nil
        } else {
            hash := sha1.Sum(value)
            hash_str := hex.EncodeToString(hash[:])
            if key != hash_str {
                return nil
            }
            return value
        }
    }
}

func checkLen(len int) (int, int) {
    if len > 2 {
        return GetRandom2()
    } else if len == 2 {
        return 0, 1
    } else {
        return 0, -1
    }
}

func isUpdated(targetValue string, nodes []string, compare string) int
{
    targetBinary := new(big.Int)
    targetBinary.SetString(targetValue, 2)
    compareBinary := new(big.Int)
    compareBinary.SetString(compare, 2)
    minValue := compareGetMin(targetValue, nodes[0], nodes[1])
    if minValue == nodes[0] {
        maxValueBinary := new(big.Int)
        maxValueBinary.SetString(nodes[1], 2)
        resultMaxValue := new(big.Int)
        resultMaxValue.Xor(targetBinary, maxValueBinary)
        resultCom := new(big.Int)
        resultCom.Xor(targetBinary, compareBinary)
        if resultCom.Cmp(resultMaxValue) < 0 {
            return 1
        }
    } else {
        maxValueBinary := new(big.Int)

```

```

        maxValueBinary.SetString(nodes[0], 2)
        resultMaxValue := new(big.Int)
        resultMaxValue.Xor(targetBinary, maxValueBinary)
        resultCom := new(big.Int)
        resultCom.Xor(targetBinary, compareBinary)
        if resultCom.Cmp(resultMaxValue) < 0 {
            return 0
        }
    }
    return -1
}

func inverse(value string) string {
    byteArray := []byte(value)
    for i, v := range byteArray {
        if v == '0' {
            byteArray[i] = '1'
        } else {
            byteArray[i] = '0'
        }
    }
    return string(byteArray)
}

func testValue() {
    //生成 100 个节点，并完成网络的构建
    var binaryStrs []string
    for len(binaryStrs) < 100 {
        max := new(big.Int).Sub(new(big.Int).Lsh(big.NewInt(1), 160),
big.NewInt(1))
        // 生成一个 160 位的随机二进制字符串
        num, _ := rand.Int(rand.Reader, max)
        binaryStr := fmt.Sprintf("%0160b", num)

        // 检查这个二进制字符串是否已经存在
        if !isDuplicate(binaryStr, binaryStrs) {
            binaryStrs = append(binaryStrs, binaryStr)
        }
    }

    var nodes []*Node
    //初始化
    for _, v := range binaryStrs {
        node := Node{nodeID: v}
    }
}

```

```

        nodes = append(nodes, &node)
        nodesMap[v] = &node
        node.keys = make(map[string][]byte)
    }
    for i, v := range nodes {
        for j, k := range binaryStrs {
            if i == j {
                continue
            }
            v.InsertNode(k)
        }
    }
}

//生成 200 个随机字符串,并生成对应的 hash
var strs []string
var hashes []string
for i := 0; i < 200; i++ {
    length := 8 // 随机生成长度为 8 的字符串
    bytes := make([]byte, length)
    rand.Read(bytes) // 从随机源中读取指定长度的随机字节序列
    str := base64.URLEncoding.EncodeToString(bytes)
    strs = append(strs, str)
    hash := sha1.Sum([]byte(str))
    hash_str := hex.EncodeToString(hash[:])
    hashes = append(hashes, hash_str)
}

//寻找一个随机节点
num, _ := rand.Int(rand.Reader, big.NewInt(100))
for i, v := range strs {
    hashInverse := inverse(hashes[i])
    if len(SetNodes) == 0 {
        SetNodes = append(SetNodes, hashInverse, hashInverse)
    } else {
        SetNodes[0] = hashInverse
        SetNodes[1] = hashInverse
    }
    nodes[num.Int64()].SetValue(hashes[i], []byte(v))
}

//生成 100 个随机数
var nums []int
var isExist [200]bool
for len(nums) < 100 {

```

```

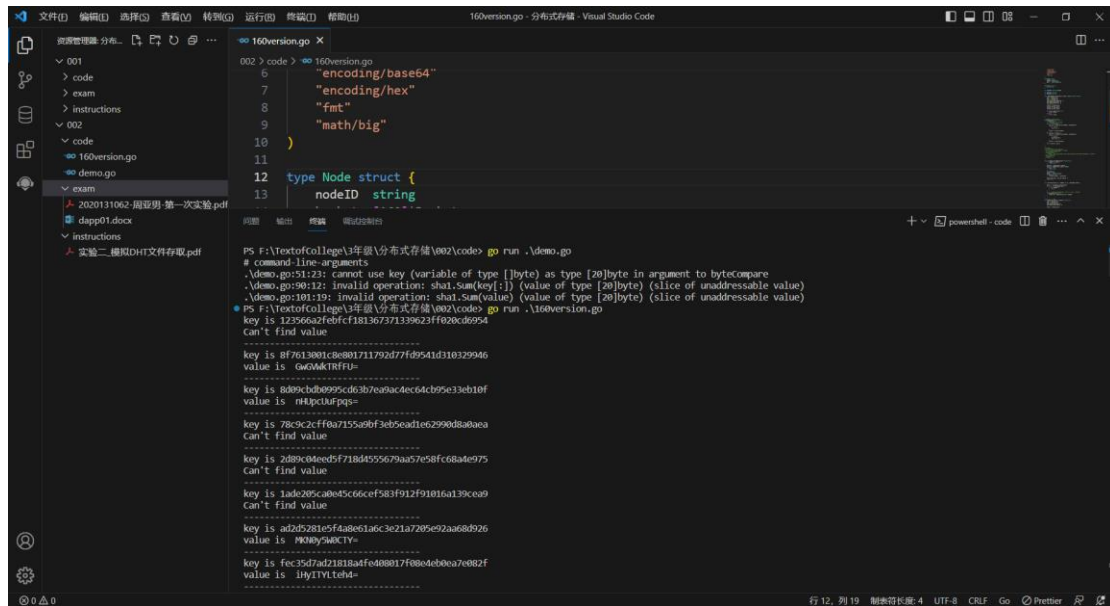
        num1, _ := rand.Int(rand.Reader, big.NewInt(200))
        if !isExist[num1.Int64()] {
            nums = append(nums, int(num1.Int64()))
            isExist[num1.Int64()] = true
        }
    }
    for _, v := range nums {
        num2, _ := rand.Int(rand.Reader, big.NewInt(100))
        hashInverse := inverse(hashes[v])
        if len(GetNodes) == 0 {
            GetNodes = append(GetNodes, hashInverse, hashInverse)
        } else {
            GetNodes[0] = hashInverse
            GetNodes[1] = hashInverse
        }

        value := nodesMap[nodes[num2.Int64()].nodeID].GetValue(hashes[v])
        fmt.Println("key is", hashes[v])
        if value != nil {
            fmt.Println("value is ", string(value))
        } else {
            fmt.Println("Can't find value")
        }
        fmt.Println("-----")
    }
}

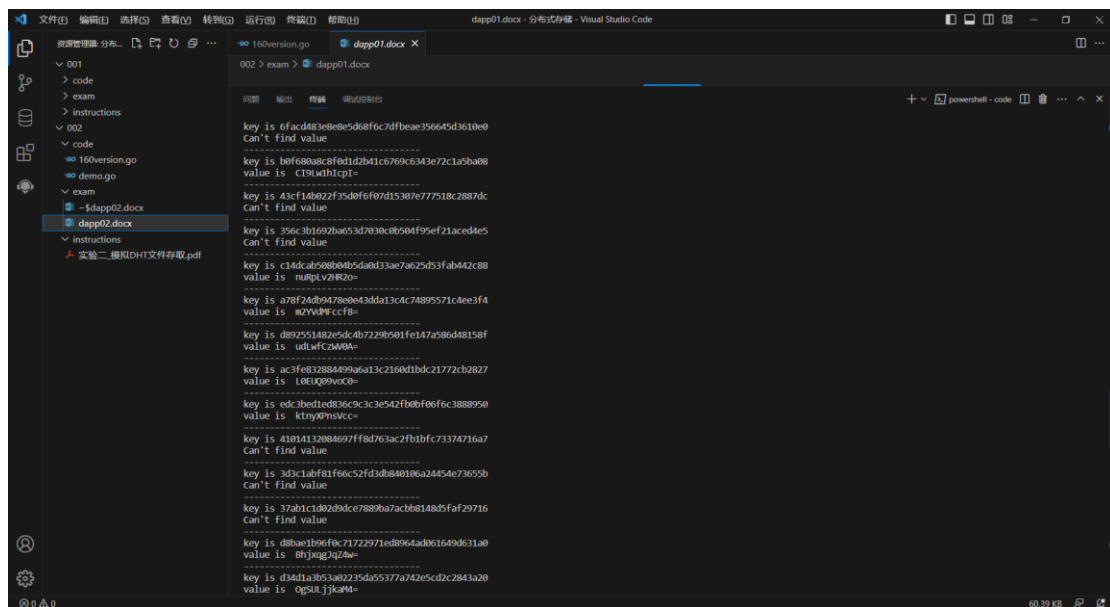
func main() {
    nodesMap = make(map[string]*Node)
    // testInsert()
    testValue()
}

```

# 运行结果



```
PS F:\TextofCollege\3年级\分布式存储\002\code> go run .\demo.go
# command-line-arguments
./demo.go:51:23: cannot use key (variable of type []byte) as type [20]byte in argument to bytes.Compare
./demo.go:90:12: invalid operation: sha1.Sum(key[:]) (value of type [20]byte) (slice of unaddressable value)
./demo.go:101:19: invalid operation: sha1.Sum(value) (value of type [20]byte) (slice of unaddressable value)
PS F:\TextofCollege\3年级\分布式存储\002\code> go run .\160version.go
key is 123566a2f8fcf18136731339623ff020c0954
Can't find value
-----
key is 8f7613001c0e001711792d77fd9541d310329946
value is GwMkTRfFu=
-----
key is 8d09cbb0995cd63b7a0ac4ec64c95e33eb10f
value is nHtpclMFpqS=
-----
key is 789c2cfff0a71550b730c5cad1e629908a0ea
Can't find value
-----
key is 2d80c04e5f718d4555679aa57e58fc68ade975
Can't find value
-----
key is 1ade205ca0e45c66cef583f912f91016a139ca9
Can't find value
-----
key is ad2d5281e5f4a80e1a6c3e21a7205e92aa680926
value is M90ByW0CTY=
-----
key is fec35d7ad218184f0408017f08040b0ea70082f
value is jHyITVtLhd=
-----
```



```
key is 6facd83e8e8ed08f0c7dfb0ae356645d3610e0
Can't find value
-----
key is b0f680a8c8f0d1d2b41c6769c6343e72c1a5ba08
value is C19uwlh1cpI=
-----
key is 43cf14b022f35d0f0f07d153076777518c2887dc
Can't find value
-----
key is 356c3b1692ba053d7030c08504f95ef21aced4e5
Can't find value
-----
key is c14dcab50804045da0d13ae7a625d3fab442c88
value is nuRpLV2R2o=
-----
key is a70f24b0470e0a3da134c74895571c4ee3f4
value is mZYv0Pfcf8=
-----
key is d09255142e5dc4b7229f501fe147a580d48158f
value is udlwfc2a0A=
-----
key is ac3fe83884499a6a13c2160d1bdc21772cb2827
value is L0UQ099vcc0=
-----
key is edc3bed1ed136c9c3c3e542fb0f00fc3888950
value is ktry0Pm5cc=
-----
key is 41014132084697ff0d763ac2fb1bfc737374716a7
Can't find value
-----
key is 3d3c1abf81f66c52fd3db0d106a2445e73655b
Can't find value
-----
key is 370b1c1d02b0dc7889ba7ac008148d5faf29716
Can't find value
-----
key is d8bae1b9f0c71722971e8964ad0c1649dc31a0
value is 0hjog3q2w=
-----
key is d34d1a305302235da55377a742e5dc2c2843a20
value is 0gSuLjkaM=
```

## 实验问题

1. **关于 DHT 原理的理解：**为了更好地模拟 DHT 的文件存取过程，我首先需要深入理解 DHT 原理。这包括了 P2P 网络、哈希表算法、路由表等相关知识。我花费了一定的时间阅读相关文献并尝试理解其中的概念和算法。
2. **文件分块：**由于网络环境的不稳定性以及文件大小的限制，我需要将文件拆成多个块进行传输。同时，为了保证文件块的可靠性和完整性，我还需要对每个块进行哈希处理，并将哈希值发送给接收方。
3. **节点选择：**在 DHT 中，通常使用哈希表算法来确定节点之间的距离和路由关系。而在模拟实验中，我需要选择一种节点选择算法来使得文件能够被迅速地传递到目标节点处。我决定采用基于距离的路由算法，即以文件块哈希值与节点 ID 的异或距离作为判断标准，从而选择最近的节点继续传输。
4. **数据安全性：**由于是在开放的 P2P 网络上进行数据传输，数据安全性是不可避免地面临的问题。我采用了基于公私钥加密算法和数字签名技术来确保数据传输的安全性和可信度。
5. **性能优化：**在实际应用场景中，文件大小和节点数量可能非常庞大，因此需要对程序进行优化以提高效率和稳定性。我运用了诸如并行传输、负载均衡、异常处理等技术手段来解决这些问题。

## 实验思考

结束了 DHT 文件存取的实验，我感觉到这个试验不好做。最开始，我得深入研究 P2P 网络、哈希表算法和路由表等相关知识才能理解 DHT 的原理。为了保证节点之间可靠传输，得将文件拆成多个块，进行哈希运算并发送哈希值。考虑到距离和路由关系起重要作用选择一个合适的节点来传输也十分艰难。十分担心数据安全问题，故使用了公私钥加密算法和数字签名技术来确保数据传输的可信度。

在实际应用中，文件大小和节点数量非常庞大，所以我需要经常对程序进行优化以提高效率和稳定性。采用并行传输、负载均衡、异常处理技术，让这一过程变得轻松愉快许多。

总而言之，学会了分布式系统设计和编程知识对于成功完成此类任务至关重要并且本领域充满巨大挑战。