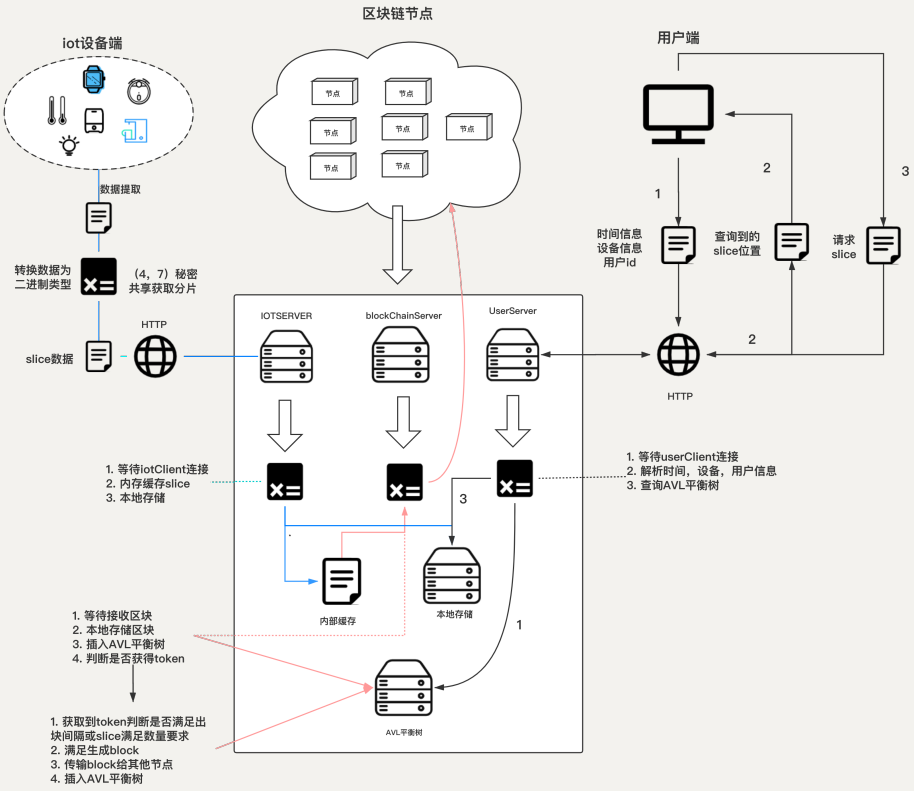
项目中期程序说明文档

整体流程与框架

整体架构图



配置文件读取

将节点以及端口等初始化信息由config.json进行管理，节点启动时首先解析该文件进行初始化操作

```
{
    "p2p_port": 5061,
    "http_port": 5001,
    "node_id": 0,
    "address_book": [
        "http://10.122.196.144:5061",
        "http://10.122.196.144:5062",
        "http://10.122.196.144:5063",
        "http://10.122.196.144:5064",
        "http://10.122.196.144:5065",
        "http://10.122.196.144:5066",
        "http://10.122.196.144:5067"
    ]
}
```

```
func readConfig() {
  data, err := ioutil.ReadFile("./config.json")
  if err != nil {
    panic(err)
  }
  err = json.Unmarshal(data, &config)
  if err != nil {
    panic(err)
  }
}
```

初始化节点

创建节点，初始化httpNodeServer,获取到自己的url，根据backup.txt调用buildTraverser 函数初始化树并读取备份文件，同时校验区块，并插入树中。然后向地址本中的其他节点交互自己的节点id与地址。当节点数满足7个的时候，创建由0号节点创世区块并广播给其他节点，移交token。

```
func initNode() {
  sig = make(chan struct{}, 1)
  node = makeNode(config.P2PPort)
  myURL = node.URL
  log.Printf("I am %s\n", myURL)
  time.Sleep(time.Second * 10)
  updateConnectionInfo(config.NodeId, myURL)
  buildTraverser()
  for nodeId, target := range config.AddressBook {
    if nodeId == config.NodeId {
      continue
    }
    //对地址本中一个节点发送消息，通知自己地址与id
    go func(target string) {
      for {
        log.Printf("Start to send postConnectionInfo to
%s", target)
        buf := new(bytes.Buffer)
        w := multipart.NewWriter(buf)
        _ = w.WriteField("node_id",
strconv.Itoa(config.NodeId))
        _ = w.WriteField("URL", myURL)
        _ = w.Close()
        req, err := http.NewRequest("POST",
target+"/postConnectionInfo", buf)
        if err != nil {
          // TODO: Handle error
          panic(err)
```

```
        }
        req.Header.Set("Content-Type",
w.FormDataContentType())
        client := http.Client{Timeout: RequestTimeout}
        resp, err := client.Do(req)
        if err != nil || resp.StatusCode != 200 {
          log.Printf("Failed to send postConnectionInfo
to %s", target)
          time.Sleep(SendDuration)
          continue
        }
        break
      }
    }(target)
  }
  //收到postConnectionInfo函数通过管道传入的sig后，判定当前上线
节点数并生成初始区块。
  go func() {
    for {
      select {
      case <-sig:
        log.Printf("%d nodes connected", len(urlList))
        if len(urlList) == 7 {
          if config.NodeId == 0 {
            setGenius()
          }
          bigint := big.NewInt(100000)
          initD := DATA{"xxx", "xxx", time.Now(), -1,
"xxxxxxxxxx", "xxxxx", bigint}
          head = &DataNode{initD, nil}
          last = head
          break
        }
      }
    }
  }()
}
```

启动iotServer 与 userServer

启动iotServer 与 userServer 等待接收iot设备的slice和userclient的数据查询请求。

```
func httpServer() {
  //ip := GetOutboundIP()
  ip := "10.122.196.144"
  serverMux := http.NewServeMux()
  serverMux.HandleFunc("/getSlice", getSliceHandler)
  serverMux.HandleFunc("/getIndex", getIndexHandler)
  serverMux.HandleFunc("/postSliceData",
postSliceDataHandler)

  go
http.ListenAndServe(ip+":"+strconv.Itoa(config.HttpPort)
, serverMux)
}
```

IOT设备部分

整体思路

暂时利用go和本地测试文件模拟iot设备。读取文件内容后利用(4,7)秘密共享算法对文件进行分割并发送给7个node节点。

数据结构定义

```
type DATA struct {
  DeviceId string //IOT设备号
  UserId   string //用户ID
  SavedAt  time.Time //slice存储时间
  Serial   int //slice序号, 0-6
  Hash     string //整体hash
  StoreOn  string //slice存储的节点号
  ModNum   *big.Int //秘密共享的模数
}
```

IOT连接初始化

初始化iotserver连接表

```
func initHttpServerTable() {
  httpServerTable = append(httpServerTable,
"http://10.122.196.144:5001",
"http://10.122.196.144:5002",
"http://10.122.196.144:5003",
"http://10.122.196.144:5004",
"http://10.122.196.144:5005",
"http://10.122.196.144:5006",
"http://10.122.196.144:5007")
}
```

秘密共享与切片生成

这部分用于将文件利用秘密共享算法进行处理，生成slice。
详见：秘密共享部分

数据传输模块

利用http协议将经过秘密共享算法得到的slice分别分发给7个节点

```
func postSliceDataClient(slice []byte, data []byte,
serverURL string) {
  postURL := serverURL + "/postSliceData"
  //contentType:="multipart/form-data"
  //准备两个bytes的reader 以二进制文件流形式输入
  sliceReader := bytes.NewReader(slice)
  dataReader := bytes.NewReader(data)

  body := new(bytes.Buffer)
  w := multipart.NewWriter(body)

  slicePart, err := w.CreateFormFile("slice", "slice")
  _, err = io.Copy(slicePart, sliceReader)
  if err != nil {
    fmt.Println(err)
  }

  dataPart, err := w.CreateFormFile("data", "data")
  _, err = io.Copy(dataPart, dataReader)
  if err != nil {
    fmt.Println(err)
  }

  w.Close()
```

```
  req, err := http.NewRequest("POST", postURL, body)
  if err != nil {
    fmt.Println(err)
    return
  }
  req.Header.Set("Content-Type",
w.FormDataContentType())

  client := http.Client{Timeout: time.Second * 3}
  response, err := client.Do(req)
  if err != nil {
    fmt.Println("Post error:", err)
    return
  }
  defer response.Body.Close()
  fmt.Println(response.Status)

}
```

区块链部分

区块结构

将区块链以json格式存储在backup.txt中，内存里只存放上一个区块的相关信息。

slice链定义如下

```
var head *DataNode
var last *DataNode

type DataNode struct {
  data DATA
  next *DataNode
}
```

区块头定义如下

```
type Block struct {
  Index              int //区块的序号
  SavedAt            time.Time //区块生成时间
  Hash               string //整体hash
  PrevHash           string //上一块的hash
  BlockGenerator     int //生成当前区块的服务器编号
  NextBlockGenerator int //下一个生成区块的服务器编号
  Data               []DATA //区块体，即数据部分
}
```

DATA定义如下

```
type DATA struct {
  DeviceId string //IOT设备号
  UserId   string //用户ID
  SavedAt  time.Time //slice存储时间
  Serial   int //slice序号，0-6
  Hash     string //整体hash
  StoreOn  string //slice存储的节点号
  ModNum   *big.Int //秘密共享的模数
}
```

区块结构如下


```
{
    "Index": 4,
    "SavedAt": "2020-12-08T23:50:13.023177+08:00",
    "Hash":
"ff2508dbcfa723db9d2fd0ee96a21796937b527c35334dcf00549f5
259e35329",
    "PrevHash":
"7de796d78aa0b24b0f31fc6504487a0bd01ea2527799d18dfe89281
214fbeb54",
    "BlockGenerator": 3,
    "NextBlockGenerator": 4,
    "Data": [
        {
            "DeviceId": "000a43b",
            "UserId": "0000f42e",
            "SavedAt": "2020-12-
08T23:48:55.5100521+08:00",
            "Serial": 4,
            "Hash":
"2300792d02a985ed7f938216a17d0617d2db3043d25236ea8ae26d8
a868eb7ef",
```

```
            "StoreOn": "http://10.122.196.144:5004",
            "ModNum": 100000000000000000000007
        }
        ...
    ]
}
```

区块的接收、生成与校验

- handleBlock
  接收上层函数传来的DATA数组，生成区块头并上链、广播。

```
func handleBlock(test []DATA) {

  mutex.Lock()
  prevBlock = getPrevBlock()
  newBlock := generateBlock(prevBlock, test)
  if isBlockValid(newBlock, prevBlock) {
    storeBlock(newBlock)
    fmt.Println("BlockGenerator !!!!!!")
    fmt.Println(config.NodeId)
    fmt.Println(newBlock.BlockGenerator)
    tmpData, _ := json.Marshal(newBlock)
    broadcastBlock(tmpData)
  }
  mutex.Unlock()
}
```

- getPrevBlock
  从backup.txt中读取最后一个区块（最后一行）的信息，转成json格式并返回。

```
func getPrevBlock() Block {

  file, err := os.Open("backup.txt")
  if err != nil {
    log.Fatal(err)
  }
  defer file.Close()
  br := bufio.NewReader(file)

  var lastLine []byte
  for {
    currentLine, _, c := br.ReadLine()
    if c == io.EOF {
      var newBlock Block
```

```
      err1 := json.Unmarshal(lastLine, &newBlock)
      if err1 != nil {
        fmt.Println(err)
      }
      return newBlock
    }
    lastLine = currentLine
  }
}
```

- generateBlock
  利用上个区块的hash生成新区块，并将Data填入新的区块中。

```
func generateBlock(oldBlock Block, Data []DATA) Block {

  var newBlock Block

  t := time.Now()

  newBlock.Index = oldBlock.Index + 1
  newBlock.SavedAt = t
  newBlock.PrevHash = oldBlock.Hash
  //genius
  if oldBlock.Hash == "" {
    newBlock.BlockGenerator = 0
    newBlock.NextBlockGenerator = 1
  } else {
    newBlock.BlockGenerator = (oldBlock.BlockGenerator +
1) % 7
    newBlock.NextBlockGenerator =
(oldBlock.BlockGenerator + 2) % 7
  }

  //newBlock.Data = Data
  for _, data := range Data {
    if data != (DATA{}) {
      newBlock.Data = append(newBlock.Data, data)
    }
  }

  newBlock.Hash = calculateHash(newBlock)
  return newBlock
}
```

- isBlockValid

检验新生成区块的合法性

- 新区块的序号是否是上一个区块的序号递增1
- 新区块中存放的上一块的hash是否正确
- 新区块的内容是否被篡改

```
func isBlockValid(newBlock, oldBlock Block) bool {
  if oldBlock.Index+1 != newBlock.Index {
    return false
  }

  if oldBlock.Hash != newBlock.PrevHash {
    return false
  }

  if calculateHash(newBlock) != newBlock.Hash {
    return false
  }

  return true
}
```

- storeBlock
  将生成的区块转成byte数组存入backup.txt，并插入检索树。

```
func storeBlock(newBlock Block) {
  fd, _ := os.OpenFile("backup.txt",
os.O_RDWR|os.O_CREATE|os.O_APPEND, 0755)
  tmpData, _ := json.Marshal(newBlock)
  InsertBlock(&newBlock)
  fd.Write(tmpData)
  fd.Write([]byte("\n"))
  fd.Close()
}
```

用户端部分

总体思路

暂时利用go代码模拟user部分，输入起始时间与时间间隔，利用http向随机一个节点请求这个时间段属于该用户的slice切片的所有存放位置，然后用户根据得到的存放位置逐一请求各个节点以得到分片并利用秘密共享算法进行数据恢复。

slice获取部分

首先利用http 设置起始、终止时间、iotID和userID向服务器发送获取时间段内slice的所有存放位置，根据存放位置去依次请求slice，当请求到4个且可以恢复后即完成，否则继续请求其他组合直到可以完成恢复，若都无法完成则认为数据已无法恢复。

```
func userGetIndex(startTime time.Time, endTime
time.Time, iotID string, userID string) {
  //1.处理请求参数
  params := url.Values{}
  params.Set("startTime",
strconv.FormatInt(startTime.Unix(), 10))
  params.Set("endTime",
strconv.FormatInt(endTime.Unix(), 10))
  params.Set("iotID", iotID)
  params.Set("userID", userID)

  //2.设置请求URL
  rawUrl := serverURL + "/getIndex"
  reqURL, err := url.ParseRequestURI(rawUrl)
  if err != nil {
    log.Fatalf("url.ParseRequestURI()函数执行错误,错误
为:%v\n", err)
    return
  }

  //3.整合请求URL和参数
  //Encode方法将请求参数编码为url编码格式
("bar=baz&foo=quux"),编码时会以键字母进行排序。
  reqURL.RawQuery = params.Encode()

  //4.发送HTTP请求
  //说明: reqURL.String() String将URL重构为一个合法URL字符
串。
  //eg: http://127.0.0.1:5000/getIndex?
endTime=1606086515&iotID=aabbccdd&startTime=1606072115&u
serID=11223344
  resp, err := http.Get(reqURL.String())
  if err != nil {
    fmt.Printf("http.Get()函数执行错误,错误为:%v\n", err)
    return
  }
  defer resp.Body.Close()

  //5.一次性读取响应的所有内容
  body, err := ioutil.ReadAll(resp.Body)
  if err != nil {
```

```
    fmt.Printf("ioutil.ReadAll()函数执行出错,错误为:%v\n",
err)
    return
  }
  //body是[]byte，这里仅转成16进制输出
  //fmt.Println(hex.Dump(body))

  var indexs []DATA
  err = json.Unmarshal(body, &indexs)
  for _, tmp := range indexs {
    fmt.Println(tmp)
  }
  if err != nil {
    log.Println("json解析出错，错误为：", err)
    return
  }

  for i := 0; i < len(indexs); i += 7 {
    sliceGet := 0
    slices := make([]*big.Int, 4)
    choice := make([]int64, 4)
    for j := 0; j < 7; j++ {
      temp, _ := userGetSlice(indexs[i+j].StoreOn,
indexs[i+j].Hash)
      if temp != nil {
        tempCiphertext := new(big.Int)
        tempCiphertext.SetString(string(temp), 10)
        fmt.Println(tempCiphertext)
        slices[sliceGet] = tempCiphertext
        choice[sliceGet] = int64(indexs[i+j].Serial) - 1
        sliceGet++
      }
      if sliceGet == 4 {
        fmt.Println(indexs[i+j].ModNum)
        _, err = recoverMessage(slices,
indexs[i+j].ModNum, choice)
        if err != nil {
          log.Println("还原失败，将全部get然后尝试组合恢复。")
          var indexForFailGet []DATA
          for k := 0; k < 7; k++ {
            indexForFailGet[k] = indexs[i+k]
          }
          decryptFailGet(indexForFailGet)
          continue
        }
        break
      }
```

```
    }
  }

}
```

本部分为用户在获取slice的位置后用于调用获取slice数据

```
//用户端使用get方法发起请求，获取slice
func userGetSlice(addr, hash string) ([]byte, error) {
  params := url.Values{}
  params.Set("hash", hash)

  rawUrl := addr + "/getSlice"
  reqURL, err := url.ParseRequestURI(rawUrl)
  if err != nil {
    log.Printf("url.ParseRequestURI()函数执行错误,错误
为:%v\n", err)
    return nil, err
  }

  reqURL.RawQuery = params.Encode()
  resp, err := http.Get(reqURL.String())
  if err != nil {
    log.Printf("http.Get()函数执行错误,错误为:%v\n", err)
    return nil, err
  }
  defer resp.Body.Close()

  slice, err := ioutil.ReadAll(resp.Body)
  if err != nil {
    log.Printf("ioutil.ReadAll()函数执行出错,错误为:%v\n",
err)
    return nil, err
  }
  if len(slice) == 0 {
    log.Println("slice's len == 0, return nil.")
    return nil, err
  }
  //slice是[]byte，这里仅转成16进制输出

  fmt.Println(hex.Dump(slice))
  return slice, nil
}
```

文件恢复部分

该部分用于将从server端请求到的slice恢复。
具体恢复操作详见: 秘密恢复

文件存储与检索系统部分

总体思路

检索部分选择使用AVL平衡树作为快速检索分片信息区间的数据结构。

检索树

采用AVL平衡树，使用开源的gods库中的AVL实现。
排序key采用设备号DeviceId作为第一关键字，保存时间SaveAt作为第二关键字，分片号Serial作为第三关键字。原因是查询时会指定特定的物联网设备，所以同一物联网设备的分片存储在邻近区间。同时，我们假定物联网设备同一时间戳只会产生一次数据，所以保存时间实际是一次数据的标识。由于会产生7个分片，为了让AVL平衡树区分这7个分片，使用了分片号作为第三关键字。

- 恢复检索树
  区块链部分会将系统所有区块备份至backup.txt文件中。
  在系统启动时，会调用 buildTraverser 函数初始化树并读取备份文件，同时校验区块，并插入树中。

```
func nextBlock() *Block {
  line, _, err := reader.ReadLine()
  if err != nil {
    return nil
  }
  var block Block
  err = json.Unmarshal(line, &block)
  if err != nil {
    panic(err)
  }
  if lastBlock != nil && !isBlockValid(block,
*lastBlock) {
    panic(errors.New("invalid block"))
  }
  lastBlock = &block
  return &block
}

func buildTraverser() {
  initTraverser()
  var err error
```

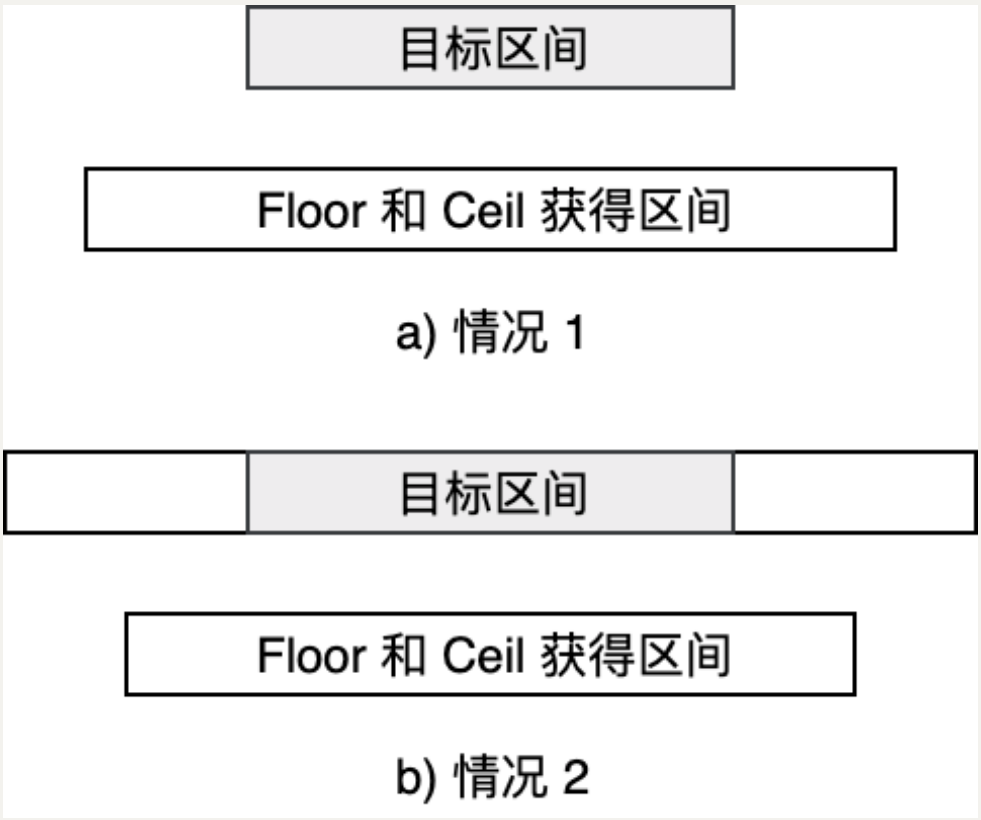
```
  backupFile, err = os.OpenFile(BackupFilePath,
os.O_RDONLY, os.FileMode(0644))
  if err != nil {
    log.Print(err)
    return
  }
  reader = bufio.NewReader(backupFile)
  for {
    block := nextBlock()
    if block == nil {
      break
    }
    InsertBlock(block)
  }
}
```

- 插入区块
  调用AVL平衡树的Put方法，遍历给定区块的数据Data，构建节点关键词TreeKey结构，插入分片信息。

```
func InsertBlock(block *Block) {
  fmt.Println(block.Data)
  for _, data := range block.Data {
    if data.Hash == "" {
      break
    }
    tree.Put(TreeKey{DeviceId: data.DeviceId, SavedAt:
data.SavedAt, Serial: data.Serial}, data)
  }
}
```

- 查询
  查询函数需要给定待查询的物联网设备id以及查询的起止时间。
  使用AVL平衡树的Floor和Ceil方法，查询开始时间的下限和结束时间的上限。
  处理Floor和Ceil的查询结果floor和ceiling，遵循以下三个步骤：

1. 若其中没有找到要求节点，即查询区间端点值不存在，树中最小的节点大于floor或者最大的节点小于ceiling，可能树的最小或者最大节点在当前查询区间内，所以分别取Left和Right节点，即树中存储的最小和最大两个节点；
2. 若当前节点floor和ceiling的设备号不是要查询的设备号或者是查询的设备号但时间小于/大于查询的时间，即查询区间端点值不存在，获得了查询区间的开区间，所以取当前节点的后一个/前一个节点；
3. 若最后得到的节点为空，或者设备号不是要查询的设备号，则表明查询区间内没有任何分片，返回空数组。



最后将查询到的区间中的分片信息取出，并返回。

```
func GetData(deviceId string, startTime time.Time,
endTime time.Time) []DATA {
  if tree.Size() == 0 {
    return []DATA{}
  }
  floor, found := tree.Floor(TreeKey{DeviceId: deviceId,
SavedAt: startTime, Serial: 1})
  if !found {
    floor = tree.Left()
  }
  if floor.Key.(TreeKey).DeviceId != deviceId ||
floor.Key.(TreeKey).SavedAt.Before(startTime) {
    floor = floor.Next()
  }
  if floor == nil || floor.Key.(TreeKey).DeviceId !=
deviceId {
    return []DATA{}
  }
  ceiling, found := tree.Ceiling(TreeKey{DeviceId:
deviceId, SavedAt: endTime, Serial: MaxSerial})
  if !found {
    ceiling = tree.Right()
  }
  if ceiling.Key.(TreeKey).DeviceId != deviceId ||
ceiling.Key.(TreeKey).SavedAt.After(endTime) {
    ceiling = ceiling.Prev()
```

```
  }
  if ceiling == nil || ceiling.Key.(TreeKey).DeviceId !=
deviceId {
    return []DATA{}
  }
  var data []DATA
  for {
    data = append(data, floor.Value.(DATA))
    if floor == ceiling {
      break
    }
    floor = floor.Next()
  }
  return data
}
```

切片的接收与缓存

切片缓存链的结构定义如下

```
var head *DataNode
var last *DataNode

type DataNode struct {
  data DATA
  next *DataNode
}
```

切片缓存链利用单链表的形式，定义了头指针与尾指针以动态缓存与读取由iot设备发送且未上链的slicedata数据

```
//接受data和slice，由iotServer调用
func receiveSlice(newSlice []byte, newData []byte) {
  //接收信息以及slice
  var dataBuf DATA
  err := json.Unmarshal(newData, &dataBuf)
  if err != nil {
    log.Println(err)
  }
  StoreSlice(newSlice, dataBuf)
  tempDataNode := &DataNode{
    data: dataBuf,
    next: nil,
  }
  if last != nil {
```

```
    last.next = tempDataNode
  }
  last = tempDataNode
}
```

切片的本地存储与读取

对iot传输的slice数据进行本地存储。

对文件定义了如下的文件结构:

- 文件名
  data结构的哈希值+.slc
- 文件头

```
var (
  SLICEHEAD = []byte{
    0x53, 0x4c, 0x43, 0x01,
  }
)
```

- 文件内容
  slice内容的16进制形式存储
- slice存储
  根据传入的slice与data结构进行存储

```
func StoreSlice(slice []byte, data DATA) bool {
  sliceLen := len(slice)
  filename := data.Hash
```

```
  print(filename)
  file, err := os.OpenFile("./slice/"+filename+".slc",
os.O_WRONLY|os.O_CREATE, os.ModePerm)
  defer file.Close()
  if err != nil {
    log.Panic(err)
    return false
  }
  if sliceLen > 0 {
    file.Write(SLICEHEAD)
    file.Write(IntToBytes(sliceLen))
    file.Write(slice)
    return true
  }
  return true
}
```

- slice读取
  根据传入的data的hash值进行文件查询

```
func ExtractSlice(hash string) []byte {
  filePath := "./slice/" + hash + ".slc"
  file, err := os.OpenFile(filePath, os.O_RDONLY,
os.ModePerm)
  if err != nil {
    log.Panic(err)
  }
  defer file.Close()
  fileinfo, _ := file.Stat()
  filesize := fileinfo.Size()
  buffer := make([]byte, filesize)

  _, err = file.Read(buffer)
  if err != nil {
    log.Panic(err)
  }
  sliceLen := BytesToInt(buffer[4:8][:])
  slice := buffer[8:][:]
  if sliceLen == len(slice) {
    return slice
  }
  return nil
}
```

秘密共享

总体思路

本项目采用(4,7)-秘密共享方案。构造一个以范德蒙德行列式为系数矩阵的单模线性同余方程组，消息被按长度等分成四份作为方程组的四个未知数。分片的过程是求线性代数方程组的右侧常数列向量；解密的过程则是根据其中的四个方程求解四元一次方程组。

秘密分片部分被IOT设备调用；秘密恢复部分被用户端调用。

秘密分片

- 初始化
  首先初始化系数矩阵为

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 \\ 1 & 4 & 8 & 16 \\ 1 & 8 & 27 & 64 \\ 1 & 32 & 243 & 1024 \\ 1 & 64 & 729 & 4096 \end{bmatrix}$$

```
tempCoefficient := [7][4]int64{
    {1, 1, 1, 1},
    {1, 2, 3, 4},
    {1, 4, 9, 16},
    {1, 8, 27, 64},
    {1, 16, 81, 256},
    {1, 32, 243, 1024},
    {1, 64, 729, 4096}}

  coefficient := make([][]*big.Int, 7)
  for i := 0; i < 7; i++ {
    coefficient[i] = make([]*big.Int, 4)
  }

  for i := 0; i < 7; i++ {
    for j := 0; j < 4; j++ {
      coefficient[i][j] =
new(big.Int).SetInt64(tempCoefficient[i][j])
    }
  }
```

然后读取文件内容，返回其二进制字符串。

```
func myRead(path string) string {
  // 读文件得到[]byte格式
  bytesFormat := readToBytes(path)
  // 将[]byte转为big.int
  intFormat := new(big.Int)
  intFormat.SetBytes(bytesFormat)
  // 将big.int转为二进制字符串
  binStr := fmt.Sprintf("%b", intFormat)
  return binStr
}
```

接下来将得到的文件内容和系数矩阵作为参数，调用encrypt函数进行分片。

```
ciphertext, p := encrypt(coefficient, messageStr)
```

- 分片
  encrypt函数对消息进行分片，返回长度为7的分片数组以及加解密过程中需要的模数p，分片以及模数p均设置为 big.int 类型。

首先计算模数p，我们将模数p设置为大于 \(2^{L/4}\)的最小素数。

```
modNum := big.NewInt(1) // 模数p
for i := 0; i < L; i++ {
    modNum.Mul(modNum, baseNum)
}

one := big.NewInt(1)
two := big.NewInt(2)
p := modNum
p.Add(p, one)
for {
    if p.ProbablyPrime(4) {
        break
    }
    p.Add(p, two)
}
fmt.Println("p =", p)
```

接着将messageStr按长度尽可能平均分成4份，分成四个大整数。为了防止类似"0111110101"这样数据的高位0的丢失，我们在每个数据的最高位人为增加一个"1"，解密的时候再去掉。

```
// 将明文消息分成四份
var splitMessage [4]*big.Int
temp := 0
for i := 0; i < 4; i++ {
    tmpInt := big.NewInt(0)
    //在每段 splitMessage 前加一个1，防止下面计算的时候最高位的
0丢失
    tmpStr := "1" + messageStr[temp:temp+splitLen[i]]
    tmpInt, _ = tmpInt.SetString(tmpStr, 2)
    splitMessage[i] = tmpInt
    temp += splitLen[i]
}
```

最后按下面的公式计算七个分片，并将分片数组ciphertext和模数p返回。其中\(s_1-s_7\)是分片，\(x_1-x_4\)是被分成四份的原消息。

$$
\begin{aligned}
s_1 &= 1^0 * x_1 + 2^0 * x_2 + 3^0 * x_3 + 4^0 * x_4 \ mod\, p \\
s_2 &= 1^1 * x_1 + 2^1 * x_2 + 3^1 * x_3 + 4^1 * x_4 \ mod\, p \\
s_3 &= 1^2 * x_1 + 2^2 * x_2 + 3^2 * x_3 + 4^2 * x_4 \ mod\, p \\
s_4 &= 1^3 * x_1 + 2^3 * x_2 + 3^3 * x_3 + 4^3 * x_4 \ mod\, p \\
s_5 &= 1^4 * x_1 + 2^4 * x_2 + 3^4 * x_3 + 4^4 * x_4 \ mod\, p \\
s_6 &= 1^5 * x_1 + 2^5 * x_2 + 3^5 * x_3 + 4^5 * x_4 \ mod\, p \\
s_7 &= 1^6 * x_1 + 2^6 * x_2 + 3^6 * x_3 + 4^6 * x_4 \ mod\, p
\end{aligned}
$$

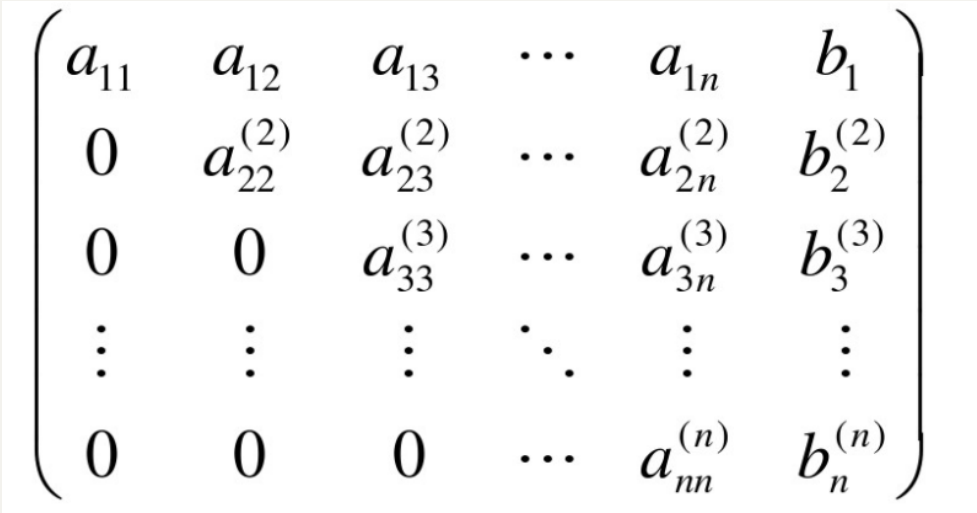
```
ciphertext := make([]*big.Int, 7)
for i := 0; i < 7; i++ {
    var temp1 *big.Int
    temp1 = new(big.Int)
    ciphertext[i] = big.NewInt(0)
    for j := 0; j < 4; j++ {
        temp1 = big.NewInt(0)
        temp1.Mul(coefficient[i][j], splitMessage[j])
        ciphertext[i].Add(ciphertext[i], temp1)
    }
    ciphertext[i].Mod(ciphertext[i], p)
}
```

秘密恢复

recoverMessage函数需要存有4个分片的数组、模数p以及每个分片对应的序号（一个choice数组）三个数据作为参数，恢复原消息并将其写入指定文件。

- 恢复消息
  解密的主要代码位于solve.go中，使用列主元消去法解方程。

首先将系数矩阵的增广矩阵整理成如下所示的上三角矩阵。



其中calculate(a,b,p)函数的功能是返回大整数\((a*(b^{-1}\ mod\ p))mod\ p\)，这个运算被多次用到所以封装成了函数。

```
for i := 0; i < n-1; i++ {
    //求第i列的主元素并调整顺序
    acol := make([]int64, n-i)
    for icol := i; icol < n; icol++ {
        acol[icol-i] = atemp[icol][i].Int64()
    }
    _, ii, _ := MaxAbs(acol)
    if ii+i != i {
        temp0 = atemp[ii+i]
        atemp[ii+i] = atemp[i]
        atemp[i] = temp0
        temp1 = btemp[ii+i]
        btemp[ii+i] = btemp[i]
        btemp[i] = temp1
    }

    //列消去
    for j := i + 1; j < n; j++ {
        mul := calculate(atemp[j][i], atemp[i][i], p)
        for k := i; k < n; k++ {
            var t1 *big.Int
            t1 = new(big.Int)
            t1.Mul(mul, atemp[i][k])
            t1.Sub(atemp[j][k], t1)
            atemp[j][k].Mod(t1, p)
        }
        var t2 *big.Int
        t2 = new(big.Int)
        t2.Mul(mul, btemp[i])
        t2.Sub(btemp[j], t2)
        btemp[j].Mod(t2, p)
```

```
    }
}
```

整理成上三角阵之后就可以从最后一行向上回代方程的解了。

```
//回代
sol[n-1] = calculate(btemp[n-1], atemp[n-1][n-1], p)
for i := n - 2; i >= 0; i-- {
    temp1 = big.NewInt(0)
    for j := i + 1; j < n; j++ {
        var temp2 *big.Int
        temp2 = new(big.Int)
        temp2.Mul(atemp[i][j], sol[j])
        temp1.Add(temp1, temp2)
    }
    sol[i] = calculate(temp1.Sub(btemp[i], temp1),
atemp[i][i], p)
}
```

最后将加密时在高位添加的"1"去掉，并将解方程得到的\(x_1-x_4\)拼在一起恢复出原消息的二进制字符串。

```
for i := 0; i < 4; i++ {
    // 将加密时高位添加的1去掉
    binaryStr := fmt.Sprintf("%b", sol2[i])
    recoveredMessage += binaryStr[1:]
}
```

- 写入文件
  将messageStr先转换成bigInt，再化成bytes数组写入文件。

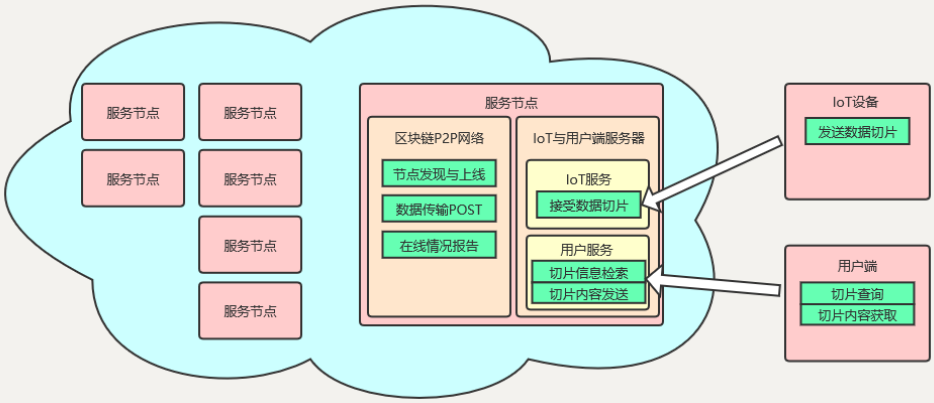
```
recoveredMessageBigInt := new(big.Int)
recoveredMessageBigInt.SetString(recoveredMessage, 2)
_ = ioutil.WriteFile("./recoverMessage.txt",
recoveredMessageBigInt.Bytes(), 0777)
fmt.Println(recoveredMessageBigInt.String())
```

网络传输部分

总体思路

网络传输层的总体思路是使用**HTTP服务**实现P2P网络连接以及与用户端和IoT设备的连接。

架构示意图如下：



图中展示的全部网络连接均基于HTTP协议。在节点之间还有更进一层的应用层协议，HTTP仅负责提供数据传输功能以及简单的响应功能；对于IoT设备和用户端就是纯粹的HTTP服务。

区块链P2P网络

- node.go 节点类及发送数据的功能

```
type Node struct {
  ip         string
  port       int
  URL        string
  P2PServer *http.ServeMux
}
```

此为node节点的对象声明，包括IP地址、端口、URL和httpServer对象。其中：

- URL应当为node的唯一标识，由IP地址和端口组成。
- 对于节点之间，应当IP不同、端口相同。
- httpServer对象的相关方法在p2pHttpServer.go中完成。

```
func makeNode(port int) *Node {
    //......
}
```

初始化节点时调用。参数仅需要端口。内部包括本机的互联网IP地址的获取，node的创建，处理函数的绑定和http服务器启用。

```
//Node的方法，调用发送实现和进行错误处理。进一步有了签名还可以做加密等工作。
func (e *Node) sendData(targetURL string, data []byte) bool {
  //......
  err := e.sendP2PMessage(targetURL, data)
  //......
}

//Node的方法，以POST方法实现发送区块信息。
func (e *Node) sendP2PMessage(serverURL string, data []byte) error {
  //......

  contentPart, err := w.CreateFormFile("block", "block")
  _, err = io.Copy(contentPart, packReader)
  if err != nil {
    return err
  }
  w.Close()

  req, err := http.NewRequest("POST", postURL, body)
  if err != nil {
    return err
  }
  req.Header.Set("Content-Type", w.FormDataContentType())

  //......
}
```

节点发送数据的主要方法。对于上层来说，可以直接调用sendData，传入目标URL参数和[]byte字节流即可发送。内部具体实现是使用POST方法中的multipart/form-data来完成表单内容构建和发送的。

- p2pHttpServer.go 定义node中http服务器的handler函数

```
func (e *Node) NewP2PProtocol() {
  //......
}
```

注册handler函数。在makeNode函数中调用。

```
//http_P2P post方法传送block的handler
```

```
func postHttpP2PHandler(w http.ResponseWriter, r
*http.Request) {
  //......
  if r.MultipartForm.File != nil {
    pack = parseMultipartFormFileForP2P(r)
  }
    //......
  resolveFunc(w, r, pack)
}

// parse form file, 解析表单中二进制的数据。
func parseMultipartFormFileForP2P(r *http.Request)
[]byte {
  //......
  var b bytes.Buffer
  _, _ = io.Copy(&b, formFile)
  log.Printf("     formfile: content=[%s]\n",
strings.TrimSuffix(b.String(), "\n"))
  return b.Bytes()
}
```

处理收到的p2p数据。首先使用parseMultipartFormFileForP2P函数解析得到[]byte字节流。然后调用resolveFunc函数来完成解析处理。

```
//用于接收其他节点传输过来的上线信息。
func postConnectionInfo(w http.ResponseWriter, r
*http.Request) {
  //......
  _nodeId := r.Form.Get("node_id")
  //......
  URL := r.Form.Get("URL")
  updateConnectionInfo(nodeId, URL)
  if sig != nil {
    sig <- struct{}{}
  }
  w.WriteHeader(http.StatusOK)
}
```

处理收到的上线信息。获取表单中的node_id和URL，然后更新本地在线状态并向sig管道中传入信号，调用主函数中对当前在线节点数的判断。

- main.go 通知上线和激活创世区块生成相关内容

```
func initNode() {
  //......
  for nodeId, target := range config.AddressBook {
    if nodeId == config.NodeId {
      continue
    }
    //对地址本中一个节点发送消息，通知自己地址与id
    go func(target string) {
      for {
        //......
        _ = w.WriteField("node_id",
strconv.Itoa(config.NodeId))
        _ = w.WriteField("URL", myURL)
        _ = w.Close()
        req, err := http.NewRequest("POST",
target+"/postConnectionInfo", buf)
        //......
        break
      }
    }(target)
  }
  //收到postConnectionInfo函数通过管道传入的sig后，判定当前上线
节点数并生成初始区块。
  go func() {
    for {
      select {
      case <-sig:
        log.Printf("%d nodes connected", len(urlList))
        if len(urlList) == 7 {
          //......
          break
        }
      }
    }
  }()
}
```

- for循环与go func(target string)，对地址本中全部节点并行地、不断发送自己的在线状态，直到全部收到OK的响应为止。
- go func()，等待postConnectionInfo函数从sig管道传过来的信息，判定是否全部上线并生成初始区块。

区块链节点通信协议

本部分通过对需要发送和接收到的消息进行打包和解包，定义了如下包结构


```
type Pack struct {
  nodeID, packetID, typeID, length int
  content                          []byte
}
```

nodeID: 1-7
packetID: 留用
TypeID:

TYPEID	解释
100	心跳包
101	心跳响应
200	收到广播的区块
201	收到宕机之后的区块
400	广播某节点无法连接通知
401	节点重启，请求当前已有区块之后的区块链。

对pack的数据结构转换为byte数组并返回

```
func Enpack(pack Pack) []byte {
  message := IntToBytes(pack.nodeID)
  var temp = append(append(append(append(message,
IntToBytes(pack.packetID)...),
IntToBytes(pack.typeID)...),
IntToBytes(pack.length)...), pack.content...)
  return temp
}
```

对接收到的字节流进行拆包，解析为pack类型数据

```
func Unpack(message []byte) Pack {
  var pack Pack
  pack.nodeID = ByteToInt(message[0:4])
  pack.packetID = ByteToInt(message[4:8])
  pack.typeID = ByteToInt(message[8:12])
  pack.length = ByteToInt(message[12:16])
  pack.content = message[16:]
  return pack
}
```

心跳包的发送函数

```
func heartBeat(typeID int) {
  var hiPack Pack
  hiPack.nodeID = nodeID
  hiPack.packetID = 1
  hiPack.typeID = typeID
  hiPack.length = 16
  heartPack := Enpack(hiPack)
  //调用发送
  for nodeId, URL := range urlList {
    if nodeId == config.NodeId {
      continue
    } else {
      sendResult := node.sendData(URL, heartPack)
      if !sendResult {
        //TODO:心跳
      }

    }
  }
}
```

广播区块函数，由区块链部分调用，负责生成区块包并发送

```
func broadcastBlock(block []byte) {
  var blockPack Pack
  blockPack.nodeID = nodeID
  blockPack.packetID = 1
  blockPack.typeID = 200
  blockPack.content = block
  blockPack.length = 16 + blockPack.length
  pack := Enpack(blockPack)
  for nodeId, URL := range urlList {
    if nodeId == config.NodeId {
```

```
      continue
    } else {
      node.sendData(URL, pack)
    }
  }
}
```

宕机的节点重新请求宕机后的所有区块。

```
func requestAllBlocks() {
  var blockPack Pack
  blockPack.nodeID = nodeID
  blockPack.packetID = 1
  blockPack.typeID = 401
  blockPack.length = 16 + blockPack.length
  Enpack(blockPack)
}
```

对接收到解析后的包分发给不同的处理函数

```
func resolveFunc(w http.ResponseWriter, r *http.Request,
bPack []byte) {
  var receivedPack Pack
  receivedPack = Unpack(bPack)
  switch receivedPack.typeID {
  //hello
  case 100:
    println("receice heartbeat from: ",
receivedPack.nodeID)
    w.WriteHeader(http.StatusOK)
  //hello response
  case 101:
    println("heartbeat response")
  case 200:
    println("receive")
    block := Block{}
    temp := receivedPack.content
    json.Unmarshal(temp, &block)
    storeBlock(block)
    w.WriteHeader(http.StatusOK)
    r.Body.Close()
    if config.NodeId == block.Nextblockgenerator {
      getToken()
    }
```

```
  case 201:
    println("receive blocks after crash")
    //TODO:宕机恢复
  case 400:
    println("node unreachable")
    //TODO:节点不可达，更新list
  case 401:
    println("node restart")
    //TODO:节点重启，恢复
  }
}
```

IoT与用户服务器

这部分HTTP服务由一个监听与P2P不同端口的HTTP服务器完成。

```
func httpServer() {
  //......
}
```

初始化HTTP服务器并绑定handler函数。

```
//get方法检索获取切片位置和检索号的handler
func getIndexHandler(w http.ResponseWriter, r
*http.Request) {
  //......
  data := GetData(params.Get("iotID"),
time.Unix(startTime, 0), time.Unix(endTime, 0))
  fmt.Println(data)
  j, _ := json.Marshal(data)
  _, err := w.Write(j)
  //......
}
```

接收来自用户端对数据信息的索引查询。使用get方法，参数包括：

- 开始时间

- 结束时间
- IoT设备的ID

之后以json字节流[]byte的形式返回给用户端。

```
//get方法获得获取切片具体信息的handler
func getSliceHandler(w http.ResponseWriter, r
*http.Request) {
  defer r.Body.Close()
  log.Println("Receive get slice request from ",
r.RemoteAddr)

  params := r.URL.Query() //params的类型Values，就是
map[string][]string
  slice := ExtractSlice(params.Get("hash"))
  _, err := w.Write(slice)
  if err != nil {
    log.Println(err)
  }
}
```

接收来自用户端对于数据信息的获取。使用get方法，参数包括：

- hash，是切片信息的摘要值。内容来自索引查询中的获取值。

之后根据hash值提取保存在本地的切片文件并以[]byte返回给用户端。

```
//post方法发送multipart/form-data.
//收到并读出[]byte的slice和data后，会调用receiveSlice方法
func postSliceDataHandler(w http.ResponseWriter, r
*http.Request) {
  //......
  if r.MultipartForm.File != nil {
    slice, data = parseMultipartFormFile(r)
  }
    //......
  receiveSlice(slice, data)
  w.WriteHeader(http.StatusOK)
}

// parse form file，解析表单中二进制的数据
func parseMultipartFormFile(r *http.Request) (slice
[]byte, data []byte) {
```

```
  var formNameParams = [2]string{"data", "slice"}
  for _, formName := range formNameParams {
    //......
    _, _ = io.Copy(&b, formFile)
    //......
    if formName == "data" {
      data = b.Bytes()
    } else if formName == "slice" {
      slice = b.Bytes()
    }
  }
  return
}
```

处理**IoT**发来的数据。主要是使用 `parseMultipartFormFile` 函数来解析表单中的multipart/form-data，分别处理data数据信息和slice切片内容，然后调用 `receiveSlice` 方法保存。