**周亚男 2020131062 区块链工程**

## 参考

Kademlia详解：https://blog.csdn.net/qq\_26720653/article/details/106496916

## 代码

// 实现一个节点中bucket的节点增删改查

// 实现多个节点之间的查找

/\*

a. 学⽣需要实现Kademlia DHT中的K\_Bucket数据结构，包括桶（Bucket）、节点（Node）等相

关数据结构 。

b. 学⽣应能够正确处理节点的插⼊、删除和更新等操作，根据节点ID将其分配到正确的桶中。

 \*/

/\*

1. K\_Bucket算法实现：

a. 学⽣需要实现Kademlia DHT中的K\_Bucket数据结构，包括桶（Bucket）、节点（Node）等相

关数据结构 。

b. 学⽣应能够正确处理节点的插⼊、删除和更新等操作，根据节点ID将其分配到正确的桶中。

2. 接⼝实现：

需要为K\_Bucket结构提供两个接⼝：

◦ insertNode(nodeId string)：将给定的NodeId插⼊到正确的桶中。

◦ printBucketContents()：打印每个桶中存在的NodeID

\*/

class NodeZ {

  constructor(nodeId) {

    this.nodeId = nodeId;

  }

}

class Bucket {

  constructor(maxCapacity) {

    this.nodes = [];

    this.maxCapacity = maxCapacity;

  }

  insertNode(node) {

    if (this.nodes.length < this.maxCapacity) {

      this.nodes.push(node);

    } else {

      // 桶已满，执行分裂操作

      const splitBucketIndex = this.splitBucket();

      const splitNode = this.nodes[0]; // 分裂节点选择第一个节点

      const splitNodeId = splitNode.nodeId;

      const newBucket = new Bucket(this.maxCapacity);

      newBucket.nodes = this.nodes.splice(0, Math.floor(this.maxCapacity / 2));

      if (splitBucketIndex < this.nodes.length) {

        // 分裂节点的前缀位为0

        const newNode = new NodeZ(""); // 创建一个空的NodeZ对象，nodeId暂时为空

        this.nodes.splice(splitBucketIndex, 0, newNode);

        newBucket.nodes.push(newNode); // 将新的桶插入到新的节点后面

        this.updateNodeIds(); // 更新原桶中节点的nodeId

      } else {

        // 分裂节点的前缀位为1

        const newNode = new NodeZ(""); // 创建一个空的NodeZ对象，nodeId暂时为空

        this.nodes.push(newNode); // 将新的节点添加到原桶末尾

        newBucket.nodes.push(newNode); // 将新的桶插入到新的节点后面

        this.updateNodeIds(); // 更新原桶中节点的nodeId

      }

      // 将分裂节点放入新的桶

      const splitNodeDistance = getDistance(

        splitNodeId,

        newBucket.nodes[0].nodeId

      );

      if (splitNodeDistance === splitBucketIndex) {

        newBucket.nodes.push(splitNode);

      }

      // 重新分配原桶内的节点

      for (let i = 0; i < this.nodes.length; i++) {

        const node = this.nodes[i];

        const distance = getDistance(

          node.nodeId,

          newBucket.nodes[0].nodeId

        );

        if (distance === splitBucketIndex) {

          newBucket.nodes.push(node);

        }

      }

    }

  }

  deleteNode(node) {

    const index = this.nodes.findIndex((n) => n.nodeId === node.nodeId);

    if (index !== -1) {

      this.nodes.splice(index, 1);

    }

  }

  splitBucket() {

    const prefixLength = Math.floor(Math.log2(this.nodes.length));

    return Math.pow(2, prefixLength);

  }

  updateNodeIds() {

    for (let i = 0; i < this.nodes.length; i++) {

      const node = this.nodes[i];

      if (node instanceof NodeZ) {

        node.nodeId = calculateNodeId(i);

      }

    }

  }

}

class K\_Bucket {

  constructor(bucketSize) {

    this.buckets = [new Bucket(Math.pow(2, 160))];

    this.bucketSize = bucketSize;

  }

  insertNode(nodeId) {

    const bucket = this.findBucket(nodeId);

    const newNode = new NodeZ(nodeId);

    const newBucket = new Bucket(this.bucketSize);

    newBucket.insertNode(newNode);

    this.buckets.push(newBucket);

  }

  findBucket(nodeId) {

    // 找到节点ID所在的桶

    const distance = getDistance(nodeId, this.buckets[0].nodes[0].nodeId);

    const bucketIndex = Math.floor(Math.log2(distance));

    return this.buckets[bucketIndex];

  }

  printBucketContents() {

    for (const bucket of this.buckets) {

      for (const node of bucket.nodes) {

        console.log(node.nodeId);

      }

    }

  }

}

function getDistance(nodeId1, nodeId2) {

  // 计算节点之间的距离，可以使用 XOR 运算

  // 将节点ID转换为二进制字符串

  const binaryNodeId1 = hexToBinary(nodeId1);

  const binaryNodeId2 = hexToBinary(nodeId2);

  // 计算 XOR 运算结果

  let distance = "";

  for (let i = 0; i < binaryNodeId1.length; i++) {

    if (binaryNodeId1.charAt(i) === binaryNodeId2.charAt(i)) {

      distance += "0";

    } else {

      distance += "1";

    }

  }

  // 将二进制字符串转换为十进制数值

  return parseInt(distance, 2);

}

function hexToBinary(hex) {

  let binary = "";

  for (let i = 0; i < hex.length; i++) {

    const value = parseInt(hex.charAt(i), 16);

    binary += value.toString(2).padStart(4, "0");

  }

  return binary;

}

function calculateNodeId(bucketIndex) {

  // 根据桶索引计算节点ID

  return bucketIndex.toString(16);

}

//测试：

//插入节点：

const kBucket = new K\_Bucket(3);

kBucket.buckets[0].insertNode(new NodeZ("node0"));

kBucket.insertNode("node1");

kBucket.insertNode("node2");

kBucket.insertNode("node3");

kBucket.insertNode("node4");

kBucket.insertNode("node5");

console.log("---Bucket Contents:---");

kBucket.printBucketContents();

// 插入相同的节点多次：

const kBucket2 = new K\_Bucket(3);

kBucket2.buckets[0].insertNode(new NodeZ("node0"));

kBucket2.insertNode("node1");

kBucket2.insertNode("node1");

kBucket2.insertNode("node1");

console.log("---Bucket Contents:---");

kBucket2.printBucketContents();

// 删除节点：

const kBucket3 = new K\_Bucket(3);

kBucket3.buckets[0].insertNode(new NodeZ("node0"));

kBucket3.insertNode("node1");

kBucket3.insertNode("node2");

kBucket3.insertNode("node3");

console.log("---删除前Bucket Contents:---");

kBucket3.printBucketContents();

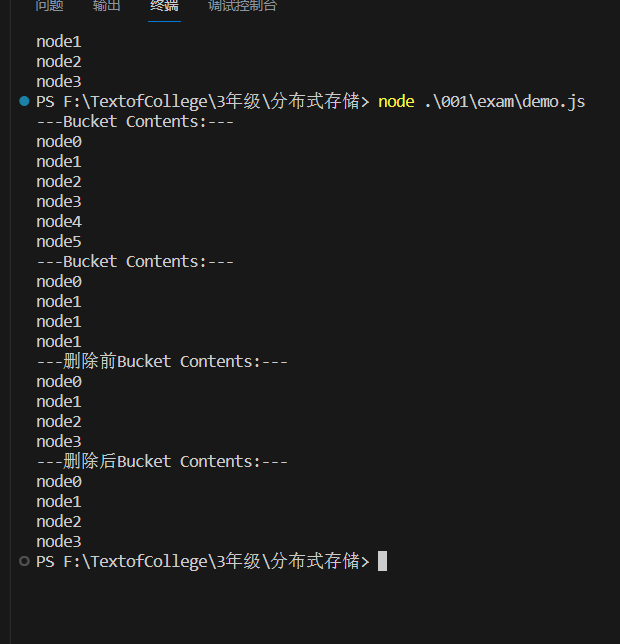
const nodeToDelete = new NodeZ("node3");

kBucket3.buckets[1].deleteNode(nodeToDelete);

console.log("---删除后Bucket Contents:---");

kBucket3.printBucketContents();

## 运行结果



## 实验思考

1. 桶的初始化：

在实现 K\_Bucket 类时，首先要在构造函数中为每个桶创建一个实例。我们可以使用 new Array(k) 来创建具有 k 个元素的数组，并在循环中使用 new Bucket() 构造这些桶。

1. 节点的插入：

当插入一个节点时，需要根据该节点与当前节点 nodeId 的距离找到正确的桶。我们可以定义一个 getDistance 方法来计算两个节点之间的距离，并使用 Math.floor(Math.log2(distance)) 计算出要插入的桶的索引。接下来，调用相应桶的 insertNode方法将节点添加到桶中即可。

1. 节点位置的移动：

在 insertNode方法中，如果要添加的节点已经存在于桶中，则需要将它移到桶的末尾。可以使用 findBucket 方法找到节点在数组中的索引，然后删除该节点将其添加到数组的末尾。

1. 桶的分裂：

当某个桶的节点数量达到一定阈值时，需要将该桶分裂成两个新桶。但是，由于我们只考虑插入节点的情况，而不考虑删除节点的情况，这个问题变得较为复杂。因此，在实现 K\_Bucket 算法时，通常可以通过忽略该情况来简化代码。

## 问题及解决方案

**问题 1**：如何计算两个节点之间的距离？

解决方案：可以将节点的 ID 转换成二进制并计算异或结果，最后得出两个节点的 Hamming 距离。详见 getDistance 方法的实现。

**问题 2**：如何手动创建一个 Node 实例？

解决方案：可以使用 new NodeZ() 创建一个带有给定 id实例。详见 NodeZ类的实现