设计实现简易分布式键值存储系统实验报告

实验目的

本次实验的目的是通过设计并实现一个简易的分布式键值存储系统，学习理解分布式存储系统，并掌握其基本设计和实现方法。分布式存储系统是将数据分布在多台物理服务器或节点上，提供一个统一的逻辑视图以供用户访问的存储系统。其核心目标是提高数据的可用性、可靠性、扩展性和性能，同时实现高效的数据管理与访问。

实验内容

技术原理

分布式哈希表(Distributed Hash Table, DHT)

分布式哈希表是一个基本的结构，被广泛的应用在许多分布式系统中，用于组织动态改变的（分布式）节点集合和透明的资源定位服务，如，DHT 在许多 P2P 系统中作为一个基本结构提供高效透明的资源定位服务。所有节点都同等重要，这使得 DHT 更加的装载平横。而且，在 DHT 中具有一定规律的拓扑结构（环、树、超立方体、网格、蝴蝶形.....），这使得查找更加的高效。虽然各种 DHT 实现的拓扑结构各不相同，但它们至少提供以下两个基本的功能：

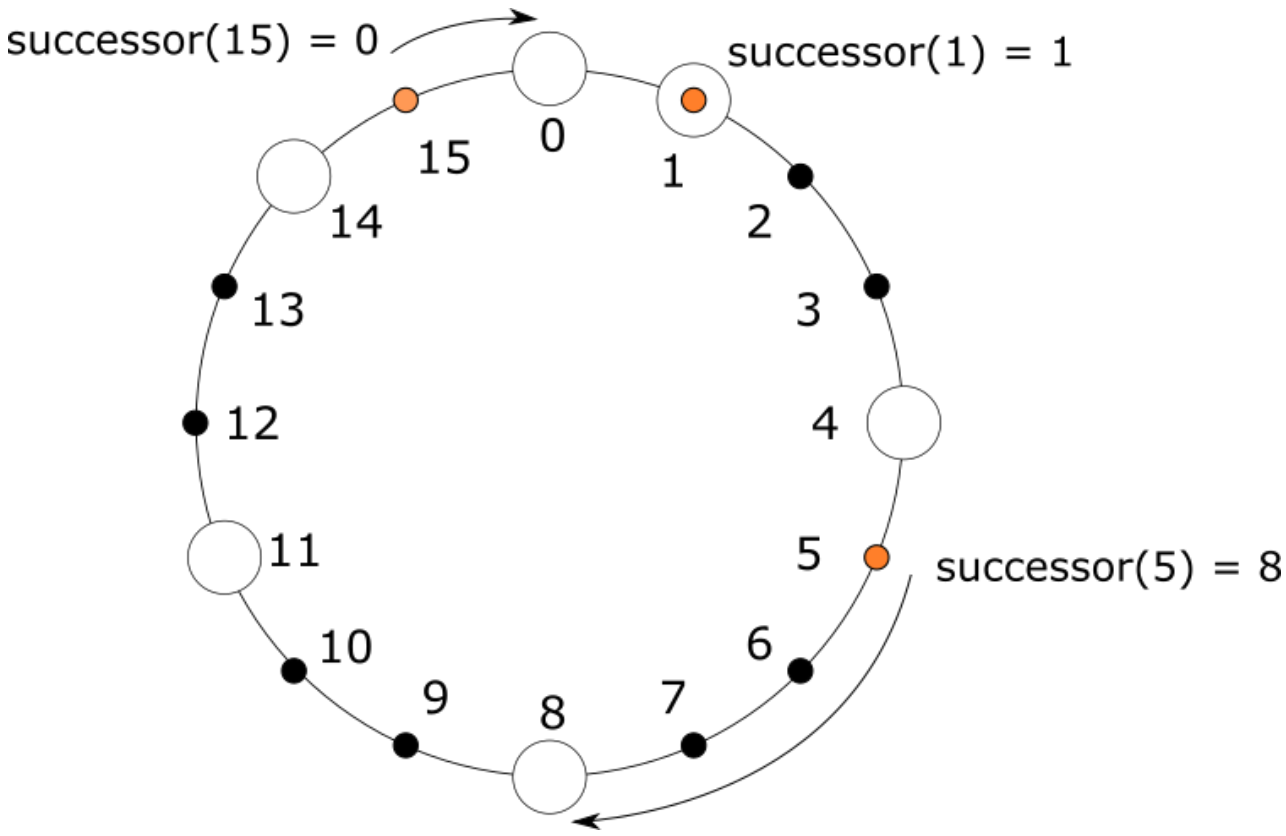
- put(key, value): 插入一个键值对到 DHT 网络中，不需要知道具体的存储地点
- value = get(key)：根据 key 从 DHT 网络中检索相应的值，不需要知道具体的存储地点

Chord算法的基本原理

地址管理

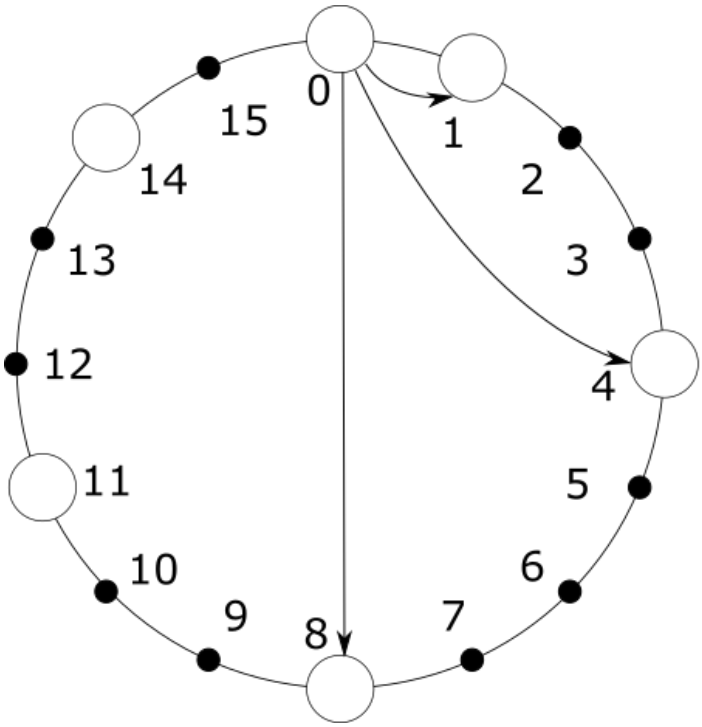
Chord 使用一个 m 位整数作为节点和资源的唯一标识，标识的取值范围为 0 至 $2^m - 1$的整数。Chord 把所有的 ID 排列成一个环，从小到大顺时针排列，首尾相连。为了方便起见，我们称每个 ID 都先于它逆时针方向的 ID，后于它顺时针方向的 ID，每个节点都能在这个环中找到自己的位置。而对于 Key 值为 k 的资源来说，它总是会被第一个 ID 先于或等于 k 的节点负责。我们把负责 k 的节点称为 k 的后继，记作*successor(k)*。

例如，如下图所示，在一个 m = 4 的 Chord 环中，有 ID 分别为 0, 1, 4, 8, 11, 14 的六个节点。Key 为 1 的资源被 ID 为 1 的节点负责，Key 为 5 的资源被 ID 为 8 的节点负责，Key 为 15 的资源被 ID 为 0 的节点负责. 也就是有 *successor*(1)=1，*successor*(5)=8，*successor*(15)=0。



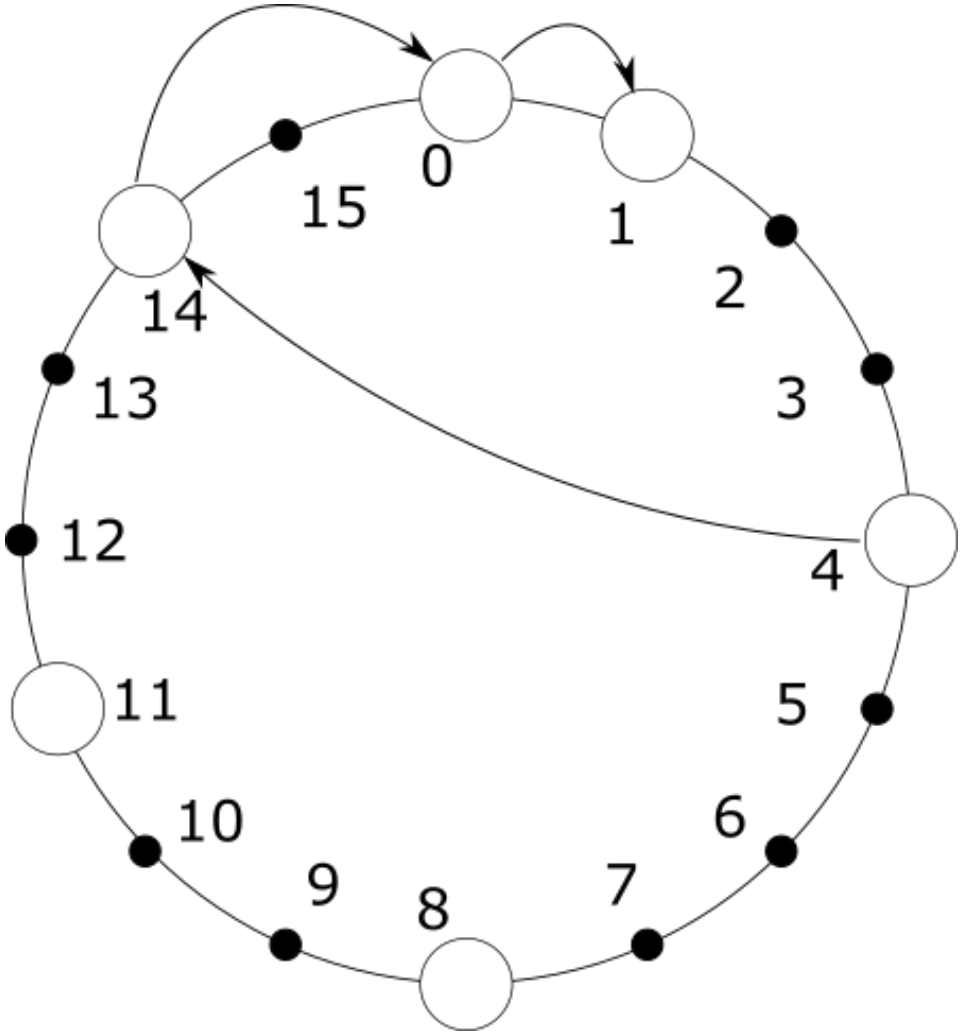
路由算法

在 Chord 算法中, 每个节点都保存一个长度为 m 的路由表, 存储至多 m 个其他节点的信息, 这些信息能让我们联系到这些节点. 假设一个节点的 ID 为 n(以下简称节点 n), 那么它的路由表中第 i 个节点应为第一个 ID 先于或等于 $n+2^{i-1}$的节点, 即 $successor(n+2^{i-1})$, 这里 $1 \leqslant i \leqslant m$。我们把 n 的路由表中的第 i 个节点记作 $n.finger[i]$。显然，路由表中的第一个节点为顺时针方向紧挨着 n 的节点, 我们通常称它为节点 n 的后继节点, 记作 $n.successor$。



i	$n+2^{i-1}$	$successor(n+2^{i-1})$
1	1	1
2	2	4
3	4	4
4	8	8

通过路由表，可以进行快速寻址。假设一个节点 n 需要寻找 Key 值为 k 的资源所在的节点，即寻找 $successor(k)$，它首先判断 k 是否落在区间 $(n, n.successor)$ 由它的后继负责。否则 n 从后向前遍历自己的路由表，直到找到一个 ID 后于 k 的节点，然后把 k 传递给这个节点由它执行上述查找工作，直到找到为止。



上图展示了从节点 4 寻找节点 1 的过程。节点 4 的路由表中的节点分别为 8, 11, 14。它首先会从后向前找到路由表中第一个后于 1 的节点为 14，然后就请求 14 帮忙寻找 1。节点 14 的路由表分别为 0, 4, 8；同样地，14 就会请求节点 0 帮忙寻找 1。最终节点 0 找到它的后继即为节点 1。

可以看到，整个查找过程是逐步逼近目标目标节点的。离目标节点越远，跳跃的距离就越长；离目标节点越近，跳跃的距离就越短，越精确。若整个系统有 N 个节点，查找进行的跳跃次数就为 $O(logN)$。

自我组织

节点加入

Chord 系统中的任意一个节点都能对任意 Key k 找到它的后继 $successor(k)$。首先，新节点 n 使用一些算法生成自己的 ID，然后它需要联系系统中的任意一个节点 n'，让他帮忙寻找 $successor(n)$。显然，$successor(n)$即是 n 的后继节点，同时也是它路由表中的第一个节点。接下来它再请求 n'帮忙分别寻找 $successor(n+2^1)$， $successor(n+2^2)$，等等， 直到构建完自己的路由表。

构建完自己的路由表后，n 基本上加入 Chord 了。但这时其他的节点还不知晓它的存在。n 加入后, 有些节点的路由表中本该指向 n 的指针却仍指向 n 的后继。这个时候就需要提醒这些节点更新路由表。哪些节点需要更新路由表呢？我们只需把操作反过来：对所有的 $1 \leqslant i \leqslant m$，找到第一个后于 $n-2^{i-1}$的节点，与寻找后继相同。

最后，新节点 n 还需要取回它负责的所有资源。n 只需联系它的后继取回它后继所拥有的所有 Key 后于或等于 n 的资源即可。

处理并发问题

考虑多个节点同时加入 Chord. 如果使用上述方法, 就有可能导致路由表不准确. 为此, 我们不能在新节点加入的时候一次性更新所有节点的路由表. Chord 算法非常重要的一点就是保证每个节点的后继节点都是准确的. 为了保证这一点, 我们希望每个节点都能和它的后继节点双向通信, 彼此检查. 因此我们给每个节点添加一个属性 $n.predecessor$指向它的前序节点。然后，我们让每个节点定期执行一个操作，称之为 $stabilize$。 在 $stabilize$ 操作中, 每个节点 n 都会向自己的后继节点 s 请求获取 s 的前序节点 x. 然后 n 会检查 x是否更适合当它的后继, 如果是, n 就它自己的后继更新为 x. 同时 n 告诉自己的后继s自己的存在, s 又会检查 n 是否更适合当它的前序, 如果是, s就把自己的前序更新为 n. 这个操作足够简单, 又能够保证 Chord 环的准确性.

当新节点 n 加入 Chord 时, 首先联系已有节点 n' 获取到 $n.successor$. n除了设置好自己的后继之外, 什么都不会做; 此时 n 还没有加入 Chord 环. 这要等到 $stabilize$ 执行之后: 当 n 执行 $stabilize$ 时, 会通知它的后继更新前序为 n; 当 n 真正的前序 p 执行 $stabilize$ 时会把自己的后继修正为 n, 并通知 n 设置前序为 p. 这样 n 就加入到 Chord 环中了. 这样的操作在多个节点同时加入时也是正确有效的.

此外每个节点 n 还会定期修复自己的路由表, 以确保能指向正确的节点. 具体的做法是随机选取路由表中的第 i 个节点, 查找并更新 $n.finger[i]$ 为$successor(n+2^{i-1})$. 由于 n 的后继节点始终是正确的, 所以这个查找操作始终是有效的.

节点失效与离开

一旦有节点失效, 势必会造成某个节点的后继节点失效. 而后继节点的准确性对 Chord 来说至关重要, 它的失效意味着 Chord 环断裂, 可能导致查找失效，$stabilize$操作无法进行. 为了解决这个问题, 一个节点通常会维护多个后继节点, 形成一个后继列表, 它的长度通常是 m. 这样的话, 当一个节点的后继节点失效后, 它会在后继列表中寻找下一个替代的节点. 此外一个节点的失效意味着这个节点上资源的丢失, 因此一个资源除了存储在负责它的节点 n 上之外, 还会被重复地存储在 n 的若干个后继节点上.

节点离开的处理与节点失效相似, 不同的是节点离开时可以做一些额外的操作, 例如通知它周围的节点立即执行 $stabilize$ 操作, 把资源转移到它的后继, 等等。

实验总体思路

计算哈希

在 Chord 中，哈希操作将数据或节点映射到一个范围为 $0 \leq h(k) < 2^m$ 的哈希空间。这个哈希值决定了数据的位置和节点的环上位置。具体来说，`SHA-1` 哈希算法将输入的节点 ID 或键转换为一个固定长度的哈希值，然后根据哈希值将节点或数据映射到一个大小为2^m 的哈希空间中。

```
static HashFunction getHashFunction() {
    if (hashFunction == null) {
        try {
            hashFunction = new HashFunction(MessageDigest.getInstance("SHA-1"));
        } catch (NoSuchAlgorithmException e) {
            throw new RuntimeException("No hash function available!", e);
        }
    }
    return hashFunction;
}
```

查找操作

在 Chord 网络中，每个节点都有一个后继节点，后继节点负责管理一部分键值范围。`findSuccessor` 方法的作用就是根据给定的键值，找到该键对应的后继节点。这对于 Chord 的路由操作非常重要。使用 RMI 进行远程调用是 Chord 网络中节点间通信的常见方式。通过 `RMIProxy`，本地节点能够像操作本地对象一样访问远程节点，实现分布式操作。

```
@Override
public Node findSuccessor(ID key) throws CommunicationException {
    this.testConnection();
    try {
       RemoteNodeInfo info = this.remoteNode.findSuccessor(key);
       return new RMIProxy(info, this.localURL);
    } catch (RemoteException e) {
       throw new CommunicationException("Could not connect to "
             + this.nodeURL + "!", e);
    }
}
```

找到最近的前驱节点

查找最近前驱节点的操作实际上就是 `closestPrecedingNode` 方法。`getClosestPrecedingNode(ID key)` 方法用于查找给定 ID（`key`）的最近前驱节点。Chord 协议中的前驱节点是一个指向当前节点前一个节点的引用，该节点对于路由的效率非常重要。

```
final synchronized Node getClosestPrecedingNode(ID key) {

    if (key == null) {
       NullPointerException e = new NullPointerException(
             "ID may not be null!");
       this.logger.error("Null pointer", e);
       throw e;
    }

    Map<ID, Node> foundNodes = new HashMap<ID, Node>();
    // determine closest preceding reference of finger table
```

```
Node closestNodeFT = this.fingerTable.getClosestPrecedingNode(key);
if (closestNodeFT != null) {
   foundNodes.put(closestNodeFT.getNodeID(), closestNodeFT);
}

// determine closest preceding reference of successor list
Node closestNodeSL = this.successorList.getClosestPrecedingNode(key);
if (closestNodeSL != null) {
   foundNodes.put(closestNodeSL.getNodeID(), closestNodeSL);
}

// predecessor is appropriate only if it precedes the given id
Node predecessorIfAppropriate = null;
if (this.predecessor != null
      && key.isInInterval(this.predecessor.getNodeID(), this.localID)) {
   predecessorIfAppropriate = this.predecessor;
   foundNodes.put(this.predecessor.getNodeID(), predecessor);
}
Node closestNode = null;
List<ID> orderedIDList = new ArrayList<ID>(foundNodes.keySet());
orderedIDList.add(key);
int sizeOfList = orderedIDList.size();



Collections.sort(orderedIDList);

int keyIndex = orderedIDList.indexOf(key);

int index = (sizeOfList + (keyIndex - 1)) % sizeOfList;

ID idOfclosestNode = orderedIDList.get(index);
closestNode = foundNodes.get(idOfclosestNode);
if (closestNode == null) {
   throw new NullPointerException("closestNode must not be null!");
}

if (this.logger.isEnabledFor(DEBUG)) {
   this.logger.debug("Closest preceding node of ID "
         + key
         + " at node "
         + this.localID.toString()
         + " is "
         + closestNode.getNodeID()
         + " with closestNodeFT="
         + (closestNodeFT == null ? "null" : ""
               + closestNodeFT.getNodeID())
         + " and closestNodeSL="
         + (closestNodeSL == null ? "null" : ""
               + closestNodeSL.getNodeID())
```

```
                + " and predecessor (only if it precedes given ID)="
                + (predecessorIfAppropriate == null ? "null" : ""
                      + predecessorIfAppropriate.getNodeID()));
    }
    return closestNode;
}
```

该方法通过以下步骤来查找 `key` 的最近前驱节点：

- 查找 `fingerTable` 中的最接近前驱节点。
- 查找 `successorList` 中的最接近前驱节点。
- 如果合适，检查前驱节点。
- 将找到的所有节点按照 ID 排序，并选择距离 `key` 最近的前驱节点。

实验过程

在一个 JVM 中模拟 Chord 网络

基于**oclcal 协议**，在同一个 JVM 中创建网络。通过控制台可以创建多个本地节点。

创建新Chord网络

1. 创建一个名为 `mypeer0` 的节点的新网络；
2. 同时创建两个节点`mypeer1`,`mypeer2`，它们都使用 `mypeer0` 作为引导节点；
3. 创建三个节点，`mypeer3`使用 `mypeer0` 作为引导节点，`mypeer4`,`mypeer5`两个使用 `mypeer1` 作为引导节点。

```
[main] INFO de.uniba.wiai.lspi.util.logging.Log4jLogger - Configuring log4j with 'log4j.properties'.
[main] INFO de.uniba.wiai.lspi.util.logging.Log4jLogger - log4j configured with 'log4j.properties'.
Welcome to Open Chord test environment.
(C) 2004-2008 Distributed and Mobile Systems Group
University of Bamberg

Type 'help' for a list of available commands
Console ready.
oc > create -names mypeer0
Creating new chord network.
oc > create -names mypeer1_mypeer2 -bootstraps mypeer0
Starting node with name 'mypeer1' with bootstrap node 'mypeer0'
Starting node with name 'mypeer2' with bootstrap node 'mypeer0'
oc > create -names mypeer3_mypeer4_mypeer5 -bootstraps mypeer0_mypeer1
Starting node with name 'mypeer3' with bootstrap node 'mypeer0'
Starting node with name 'mypeer4' with bootstrap node 'mypeer1'
Starting node with name 'mypeer5' with bootstrap node 'mypeer1'
```

向Chord DHT中插入条目

1. 节点 'mypeer1'发起插入请求，值 'test' 与键 'test'；
2. 节点 'mypeer2'发起插入请求，值 'test2' 与键 'test2'.

```
oc > insert -node mypeer1 -key test -value test
Value 'test' with key 'test' inserted successfully from node 'mypeer1'.
oc > insert -node mypeer2 -key test2 -value test2
Value 'test2' with key 'test2' inserted successfully from node 'mypeer2'.
```

打印所有节点存储的数据及关联的键

数据被复制了两次，因为属性 `de.uniba.wiai.lspi.chord.service.impl.ChordImpl.successors` 已被设置为 2。

因此，节点 'mypeer2', 'mypeer3', 'mypeer5'共三个节点存储了插入的 `test` 数据；

节点 'mypeer0', 'mypeer2', 'mypeer4'共三个节点存储了插入的 `test2` 数据.

属性 `de.uniba.wiai.lspi.chord.data.ID.number.of.displayed.bytes` 被设置为 1，因此数据项 `test` 的键的第一个字节以十六进制显示。

```
oc > entries
Node mypeer4: Entries:
  key = 10 9F 4B 3C , value = [( key = 10 9F 4B 3C , value = test2)]

Node mypeer1: Entries:

Node mypeer0: Entries:
  key = 10 9F 4B 3C , value = [( key = 10 9F 4B 3C , value = test2)]

Node mypeer3: Entries:
  key = A9 4A 8F E5 , value = [( key = A9 4A 8F E5 , value = test)]

Node mypeer2: Entries:
  key = 10 9F 4B 3C , value = [( key = 10 9F 4B 3C , value = test2)]
  key = A9 4A 8F E5 , value = [( key = A9 4A 8F E5 , value = test)]

Node mypeer5: Entries:
  key = A9 4A 8F E5 , value = [( key = A9 4A 8F E5 , value = test)]
```

打印特定节点条目

```
oc > entries -node mypeer1
Retrieving node mypeer1
Entries:

oc > entries -node mypeer5
Retrieving node mypeer5
Entries:
  key = A9 4A 8F E5 , value = [( key = A9 4A 8F E5 , value = test)]
```

检索指定键的数据

1. 从节点 'mypeer1' 检索与键 'test' 关联的值：'test'
2. 从节点 'mypeer2' 检索与键 'test2' 关联的值：'test2'
3. 从节点 'mypeer2' 检索与键 'test' 关联的值：'test'

```
oc > retrieve -node mypeer1 -key test
Values associated with key 'test':
test
Values retrieved from node 'mypeer1'
oc > retrieve -node mypeer2 -key test2
Values associated with key 'test2':
test2
Values retrieved from node 'mypeer2'
oc > retrieve -node mypeer2 -key test
Values associated with key 'test':
test
Values retrieved from node 'mypeer2'
```

删除先前插入的数据

1. 从节点 'mypeer1' 删除值 'test' 与键 'test'；
2. 打印删除后存在的键和值.

```
oc > remove -node mypeer1 -key test -value test
Value 'test' with key 'test' removed successfully from node 'mypeer1'.
oc > entries
Node mypeer4: Entries:
  key = 10 9F 4B 3C , value = [( key = 10 9F 4B 3C , value = test2)]

Node mypeer1: Entries:

Node mypeer0: Entries:
  key = 10 9F 4B 3C , value = [( key = 10 9F 4B 3C , value = test2)]

Node mypeer3: Entries:

Node mypeer2: Entries:
  key = 10 9F 4B 3C , value = [( key = 10 9F 4B 3C , value = test2)]

Node mypeer5: Entries:
```

打印出指定节点在其指针表中的引用

打印节点 mypeer3的指针表中节点的标识符，以及每个标识符后面的数字范围，表示该节点在指针表中对应的索引范围。

- **节点信息**：
  - 节点ID：`DB 99 60 FD`
  - 节点URL：`oclocal://mypeer3/`

```
Node: DB 99 60 FD , oclocal://mypeer3/
```

- **指针表（Finger Table）**：
  - **E5 29 3F 1C , oclocal://mypeer1/ (0-155)**：第一条记录，表示 `mypeer1` 节点负责处理标识符在 0 到 155 范围内的请求。即，`mypeer3` 会通过 `mypeer1` 来找到位于这个 ID 范围内的其他节点。
  - **19 3F CC C3 , oclocal://mypeer0/ (156-157)**：第二条记录，表示 `mypeer0` 节点负责处理标识符在 156 到 157 范围内的请求。

- **21 EC 58 4C , oclocal://mypeer4/ (158)**：第三条记录，表示 `mypeer4` 节点负责处理标识符为 158 的请求。
- **BE 1B BE 51 , oclocal://mypeer2/ (159)**：最后一条记录，表示 `mypeer2` 节点负责处理标识符为 159 的请求。

- **后继节点列表（Successor List）**：
  后继节点是 Chord 网络中紧随当前节点的节点，指示了数据的存储和查找顺序。
  - `mypeer3` 的后继节点 `mypeer1` 的标识符为 `E5 29 3F 1C`
  - `mypeer3` 的第二个后继节点 `mypeer0` 的标识符为 `19 3F CC C3`

```
Successor List:
  E5 29 3F 1C , oclocal://mypeer1/
  19 3F CC C3 , oclocal://mypeer0/
```

- **前驱节点（Predecessor）**：
  前驱节点是指在 Chord 环中排名紧靠当前节点之前的节点。
  `mypeer3` 的前驱节点是 `mypeer5`，其标识符为 `D9 DF 97 CF`

```
Predecessor: D9 DF 97 CF , oclocal://mypeer5/
```

```
oc > refs -node mypeer3
Retrieving node mypeer3
Node: DB 99 60 FD , oclocal://mypeer3/
Finger table:
  E5 29 3F 1C , oclocal://mypeer1/ (0-155)
  19 3F CC C3 , oclocal://mypeer0/ (156-157)
  21 EC 58 4C , oclocal://mypeer4/ (158)
  BE 1B BE 51 , oclocal://mypeer2/ (159)
Successor List:
  E5 29 3F 1C , oclocal://mypeer1/
  19 3F CC C3 , oclocal://mypeer0/
Predecessor: D9 DF 97 CF , oclocal://mypeer5/
```

关闭指定节点

1. 关闭节点'mypeer1'，关闭时它们会通知其前驱和后继节点；

> `mypeer1` 节点被关闭，理论上它应该从 `mypeer3` 的 **指针表**（Finger Table）和 **后继节点列表**（Successor List）中被移除。然而，再次执行 `refs –node mypeer3` 命令时，`mypeer3` 的指针表和后继节点列表没有发生变化。
>
> - 原因：在 Chord 协议中，指针表（Finger Table），后继节点列表（Successor List）并不是实时动态更新的。等待一个新的网络操作来触发更新。

```
oc > shutdown -names mypeer1
Node with name oclocal://mypeer1/ left.
oc > refs -node mypeer3
Retrieving node mypeer3
Node: DB 99 60 FD , oclocal://mypeer3/
Finger table:
  E5 29 3F 1C , oclocal://mypeer1/ (0-155)
  19 3F CC C3 , oclocal://mypeer0/ (156-157)
  21 EC 58 4C , oclocal://mypeer4/ (158)
  BE 1B BE 51 , oclocal://mypeer2/ (159)
Successor List:
  E5 29 3F 1C , oclocal://mypeer1/
  19 3F CC C3 , oclocal://mypeer0/
Predecessor: D9 DF 97 CF , oclocal://mypeer5/
```

2. 采用`retrieve`指令触发更新，发现`mypeer3`不能再查找到键`test`；
   再次通过`refs`指令查看，发现`mypeer1`已从`mypeer3`的指针列表和后继节点列表中删除。

```
oc > retrieve -node mypeer3 -key test
Values associated with key 'test':

Values retrieved from node 'mypeer3'
oc > refs -node mypeer3
Retrieving node mypeer3
Node: DB 99 60 FD , oclocal://mypeer3/
Finger table:
  19 3F CC C3 , oclocal://mypeer0/ (0-157)
  21 EC 58 4C , oclocal://mypeer4/ (158)
  BE 1B BE 51 , oclocal://mypeer2/ (159)
Successor List:
  19 3F CC C3 , oclocal://mypeer0/
  21 EC 58 4C , oclocal://mypeer4/
Predecessor: D9 DF 97 CF , oclocal://mypeer5/
```

节点崩溃

1. 使节点'mypeer3'崩溃，崩溃的节点不会通知其前驱和后继节点。

```
oc > crash -names mypeer3
Crashing node oclocal://mypeer3/.
Node with name oclocal://mypeer3/ crashed.
```

显示当前正在运行的节点列表

- 按照节点在 Chord 环中的顺序显示节点列表：
  - 节点 mypeer0，ID = 19
  - 节点 mypeer4，ID = 21
  - 节点 mypeer5，ID = D9

- 节点 mypeer1，ID = E5

```
oc > show
Node list in the order as nodes are located on chord ring:
Node mypeer0 with id 19 3F CC C3
Node mypeer4 with id 21 EC 58 4C
Node mypeer2 with id BE 1B BE 51
Node mypeer5 with id D9 DF 97 CF
oc > show -count
No. of nodes currently running 4
```

连接到并使用远程网络

控制台还可以创建一个使用 `ocsocket` 通信协议的网络。为此，控制台中会实例化一个使用 `ocsocket` 协议的单一 Open Chord 节点。此时，所有操作（例如 `insert`）都通过 `Chord` 接口的同步方法进行。

创建或加入网络

- **joinN**：在 JVM 中创建一个单一的 Open Chord 节点。如果未提供任何参数，则节点会创建一个新的网络，并在 Open Chord 的标准端口（4242）上监听。

1. 创建网络：

```
oc > joinN
Creating new chord overlay network!
URL of created chord node ocsocket://10.192.15.99/.
oc >
```

2. 加入网络：
   - `port` 参数用于指定不同于标准端口的端口号
   - `bootstrap` 参数需要指定一个引导节点的主机和端口

```
oc > joinN -port 8080 -bootstrap 10.192.15.99:4242
Trying to join chord network with boostrap URL ocsocket://10.192.15.99:4242/
URL of created chord node ocsocket://10.192.15.99:8080/.
oc >
```

将一个字符串插入到 DHT

- **insertN**：需要两个参数：`key` 和 `value`，这两个参数都是字符串。`key` 用于定义键，而 `value` 用于定义插入到 DHT 中的字符串

```
oc > insertN -key test -value test
```

检索与提供的键相关的所有数据

- **retrieveN**：需要一个 `key` 参数。

```
oc > retrieveN -key test
Values associated with key 'test':
test
```

从 DHT 中删除先前插入的数据

- **removeN**：参数与 `insertN` 相同

```
oc > removeN -key test -value test
Value 'test' with key 'test' removed.
```

显示当前在控制台 JVM 中运行的节点的指针表、后继节点列表和前驱节点

- **refsN**

```
oc > refsN
Node: 0C
Finger table:
7D (0-158)
Successor List:
7D
Predecessor: 7D
```

显示当前在控制台 JVM 中运行的 Open Chord 节点存储的数据

- **entriesN**

```
oc > entriesN
Entries:
key = A9 , value = [( key = A9 , value = test)]
```

当前节点以有序的方式离开 DHT

- **leaveN**

```
oc > leaveN
Leaving network.
```

```
(C) 2004-2008 Distributed and Mobile Systems Group
University of Bamberg

Type 'help' for a list of available commands
Console ready.
oc > joinN -port 8080 -bootstrap 10.192.15.99:4242
Trying to join chord network with boostrap URL ocsocket://10.192.15.99:4242/
URL of created chord node ocsocket://10.192.15.99:8080/.
oc > insertN -key test -value test
oc > retrieveN -key test
Values associated with key 'test':
test
oc > removeN -key test -value test
Value 'test' with key 'test' removed.
oc > refsN
Node: 74 FA 45 77 , ocsocket://10.192.15.99:8080/
Finger table:
  0D F4 8B AB , ocsocket://10.192.15.99:4242/ (0-159)
Successor List:
  0D F4 8B AB , ocsocket://10.192.15.99:4242/
Predecessor: 0D F4 8B AB , ocsocket://10.192.15.99:4242/
oc > entriesN
Entries:

oc > leaveN
Leaving network.
oc >
```

实验结果

修改项目源码中各参数设置

参数的含义

- `log4j.properties.file=log4j.properties`：指定log4j使用的属性文件名称，以确定控制日志记录的行为，如日志级别、输出格式和日志文件的滚动策略。
- `de.uniba.wiai.lspi.chord.data.ID.number.of.displayed.bytes=4`：设置显示的ID的字节数。
- `de.uniba.wiai.lspi.chord.data.ID.displayed.representation=2`：设置显示ID时的表示方式。0代表二进制，1代表十进制，2代表十六进制。
- `de.uniba.wiai.lspi.chord.service.impl.ChordImpl.successors=2`：设置每个节点在Chord环中维护的后继节点（`successor`）数量。
- `de.uniba.wiai.lspi.chord.service.impl.ChordImpl.AsyncThread.no=10`：设置异步执行任务的线程数量。
- `de.uniba.wiai.lspi.chord.service.impl.ChordImpl.StabilizeTask.start=12`：设置稳定任务（`StabilizeTask`）的开始时间
- `de.uniba.wiai.lspi.chord.service.impl.ChordImpl.StabilizeTask.interval=12`：设置稳定任务的执行间隔。
- `de.uniba.wiai.lspi.chord.service.impl.ChordImpl.FixFingerTask.start=0`：设置修复手指表任务（`FixFingerTask`）的开始时间。
- `de.uniba.wiai.lspi.chord.service.impl.ChordImpl.FixFingerTask.interval=12`：设置修复手指表任务的执行间隔。
- `de.uniba.wiai.lspi.chord.service.impl.ChordImpl.CheckPredecessorTask.start=6`：设置检查前驱节点任务（`CheckPredecessorTask`）的开始时间。
- `de.uniba.wiai.lspi.chord.service.impl.ChordImpl.CheckPredecessorTask.interval=12`：设置检查前驱节点任务的执行间隔。
- `de.uniba.wiai.lspi.chord.com.socket.InvocationThread.corepoolsize=10`：设置处理入站请求的线程池的核心线程数。
- `de.uniba.wiai.lspi.chord.com.socket.InvocationThread.maxpoolsize=50`：设置处理入站请求的线程池的最大线程数。
- `de.uniba.wiai.lspi.chord.com.socket.InvocationThread.keepalivetime=20`：设置线程池中空闲线程的存活时间。

修改参数

修改维护后继节点数量，修改id表示方式：

```
# Representation chosen when displaying IDs. 0 = binary, 1 = decimal, 2 = hexadecimal
de.uniba.wiai.lspi.chord.data.ID.displayed.representation=1

#Number of successors. Must be greater or equal to 1.
de.uniba.wiai.lspi.chord.service.impl.ChordImpl.successors=5
```

修改后的展示效果（维护5个后继节点，十进制表示）：

```
oc > create-names mypeer3_mypeer4_mypeer5-bootstraps mypeer0_mypeer1
Starting node with name 'mypeer3' with bootstrap node 'mypeer0'
Starting node with name 'mypeer4' with bootstrap node 'mypeer1'
Starting node with name 'mypeer5' with bootstrap node 'mypeer1'
oc > insert-node mypeer1-key test-value test
Created URL: oclocal://mypeer1/
Created Key and Value objects.
Found node: mypeer1,node id:229 41 63 28 hash code:
Value 'test' with key 'test' inserted successfully from node 'mypeer1'.
oc > successors-node mypeer1
Successor List:
LocalID:229 41 63 28
  25 63 204 195 , oclocal://mypeer0/
  33 236 88 76 , oclocal://mypeer4/
  190 27 190 81 , oclocal://mypeer2/
  217 223 151 207 , oclocal://mypeer5/
  219 153 96 253 , oclocal://mypeer3/

oc >
```

补充更加详细的输出

insert

- 创建URL的详细信息：在尝试创建一个`URL`对象后，添加代码来打印出创建的`URL`。
- 创建Key和Value对象的确认信息：在创建用于插入操作的`Key`和`Value`对象后，添加代码确认这些对象已经被成功创建。
- 节点查找的详细信息：在通过`Registry`查找节点后，如果节点存在，添加代码来打印出找到的节点的主机名和节点ID。这为用户提供了关于网络中节点身份的详细信息，包括它的网络位置和唯一标识符。

```
URL url = null;
    try {
      url = new URL(URL.KNOWN_PROTOCOLS.get(URL.LOCAL_PROTOCOL) + "://" + node + "/");
      //add more information
      this.out.println("Created URL: " + url);
    } catch (MalformedURLException e1) {
      throw new ConsoleException(e1.getMessage());
    }

    Key keyObject = new Key(key);
    Value valueObject = new Value(value);
    //add more information
    this.out.println("Created Key and Value objects.");
```

```
    ThreadEndpoint ep = Registry.getRegistryInstance().lookup(url);
    if (ep == null) {
      this.out.println("Node '" + node + "' does not exist!");
      return;
    }
    //add more information
    this.out.println("Found node: " + ep.getURL().getHost()+",node
id:"+ep.getNodeID()+"hash code:");
```

效果：

```
oc >  insert-node mypeer1-key test-value test
Created URL: oclocal://mypeer1/
Created Key and Value objects.
Found node: mypeer1,node id:E5 29 3F 1C hash code:5385945
Value 'test' with key 'test' inserted successfully from node 'mypeer1'.
oc >
```

```
Node nodeI = null;

    ID nodeID = ep.getNodeID();
    HashFunction hashFunction = HashFunction.getHashFunction();
    ID valueId = hashFunction.createID(value.getBytes());
    this.out.println("value="+value+",valueID="+valueId);
    int compareResult = nodeID.compareTo(valueId);


    Map eps = Registry.getRegistryInstance().lookupAll();
    ID maxID = null;
    Node maxNode = null;
    if (eps.size() != 0) {
      Iterator valueIterator = eps.values().iterator();
      while (valueIterator.hasNext()) {
        ThreadEndpoint threadEndpoint = (ThreadEndpoint) valueIterator.next();
        if (maxID ==null){
          maxID = threadEndpoint.getNodeID();
          maxNode = threadEndpoint.getNode();
        }else {
          int result = threadEndpoint.getNodeID().compareTo(maxID);
          if (result > 0){
            maxID = threadEndpoint.getNodeID();
            maxNode = threadEndpoint.getNode();
          }
        }
      }
      this.out.println("max id node:"+maxNode.getNodeURL().getHost()+",id:"+maxID);
    }
```

```
    if (maxID.compareTo(valueId) < 0 ||
maxNode.getSuccessor().getNodeID().compareTo(valueId) > 0) {
      nodeI =
Registry.getRegistryInstance().lookup(maxNode.getSuccessor().getNodeURL()).getNode();
    } else {
      if (compareResult > 0) {
        this.out.println("Comparison result with node: " +
ep.getNode().getNodeURL().getHost() + " is: " + compareResult + ", finding
predecessor");
        ThreadEndpoint tempep =
Registry.getRegistryInstance().lookup(ep.getNode().getPredecessor().getNodeURL());
        nodeI = preNode(tempep.getNode(), valueId);
        Node sucNode = nodeI.getSuccessor();
        nodeI = Registry.getRegistryInstance().lookup(sucNode.getNodeURL()).getNode();
      } else if (compareResult < 0) {
        this.out.println("Comparison result with node: " +
ep.getNode().getNodeURL().getHost() + " is: " + compareResult + ", finding successor");
        ThreadEndpoint tempep =
Registry.getRegistryInstance().lookup(ep.getNode().getPredecessor().getNodeURL());
        nodeI = sucNode(tempep.getNode(), valueId);
      } else if (compareResult == 0) {
        this.out.println("Comparison result: " + compareResult);
        nodeI = ep.getNode();
      }
    }

    this.out.println("Found node: " + nodeI.getNodeURL().getHost());
```

show

修改以获得更多的节点信息：

```
if (eps.size() != 0) {
      Iterator valueIterator = eps.values().iterator();
      ID[] ids = new ID[eps.size()];
      int index = 0;
      while (valueIterator.hasNext()) {
        ThreadEndpoint ep = (ThreadEndpoint) valueIterator.next();
        ids[index] = ep.getNodeID();
        temp.put(ids[index], ep);
        index++;
      }
      Arrays.sort(ids);
      this.out
          .println("Node list in the order as nodes are located on chord ring: ");
      for (int i = 0; i < ids.length; i++) {
        ThreadEndpoint ep = temp.get(ids[i]);
        this.out.println("Node ID : "+ep.getNodeID()+", Host : " +
ep.getURL().getHost() + " Port : " + ids[i]);
```

```
      }
    } else {
      this.out.println("No nodes running.");
    }
```

效果如下：

```
oc > show
Node list in the order as nodes are located on chord ring:
Node ID: 25 63 204 195 , Host: mypeer0, Port: -1
Node ID: 33 236 88 76 , Host: mypeer4, Port: -1
Node ID: 190 27 190 81 , Host: mypeer2, Port: -1
Node ID: 217 223 151 207 , Host: mypeer5, Port: -1
Node ID: 219 153 96 253 , Host: mypeer3, Port: -1
Node ID: 229 41 63 28 , Host: mypeer1, Port: -1
```

增加`ShowEntriesNetworkLocalNode`

于在Chord网络的本地节点上显示所有存储的条目。但由于无法连接到Chord网络，所以目前没有展示效果。

```
public class ShowEntriesNetworkLocalNode extends Command {
    /**
     * The name of this {@link Command}.
     */
    public static final String COMMAND_NAME = "entriesNL";

    public ShowEntriesNetworkLocalNode(Object[] toCommand, PrintStream out) {
        super(toCommand, out);
    }

    @Override
    public void exec() throws ConsoleException {
        Chord chord = ((RemoteChordNetworkAccess)
this.toCommand[1]).getChordInstance();
        Map<ID, Set<Entry>> entries = ((ChordImpl)chord).getEntries();

        for(java.util.Map.Entry<ID, Set<Entry>> e : entries.entrySet()) {
            this.out.println(e.getValue());
        }
    }

    @Override
    public void printOutHelp() {

    }

    @Override
    public String getCommandName() {
```

```
        return COMMAND_NAME;
    }

}
```

体会和建议

在设计实现简易分布式键值存储系统的过程中，深刻认识到分布式系统在数据一致性、故障容错、网络分区等问题上的复杂性。通过对系统的设计与实现，对分布式存储系统的工作原理以及它如何保证数据的一致性、可用性和容错性有了更为直观和深入的理解。

通过本次实验，对分布式键值存储系统的架构进行了分析，深入理解了如何通过分布式哈希表（DHT）来实现数据的分布和查找。通过模拟实现系统中的数据存储、读取、复制与容错机制，掌握如何使用一致性哈希来均衡负载并有效地处理节点的动态变化。不仅加深了对分布式存储系统理论的理解，也提高了在实际编码过程中解决问题的能力。