第五章 结构化P2P系统的算法

1. **分散式散列表(DHT)**

DHT(distributed hash table,分布式散列表)，也称分布式哈希表，是一种分布式存储方法。在不需要服务器的情况下，每个客户端负责一个小范围的路由器，并负责存储一小部分数据，从而实现整个DHT网络的寻址和存储。

**什么是哈希（Hash）？**

Hash，即哈希，也称散列，就是把任意长的输入通过散列算法（MD5、SHA-1等），变换成固定长度的输出，该输出就是散列值。这种转换是一种压缩映射，并且是不可逆的。通过这种方法可以把不同空间的输入散列到同一个空间中，方便查询和管理。

**什么是散列表（Hash table）？**

“散列表”是用来存储“键值对”的一种容器。“键值对”即“key/value pairs”，简称“K/V”。散列表的作用就是：如果有了“散列表”，就可以很方便快速地通过 key 来获得 value。

　　举个例子：

　　手机通讯簿可以通俗理解成一个“散列表”。里面的每一条记录都包含“姓名”和“电话号码”。“姓名”相当于“键值对”中的 key，电话号码相当于 value。你可以通过姓名方便地查找出电话号码。

**什么是分布式散列表（Distributed Hash table）？**

“分布式散列表”在概念上类似与传统的“散列表”，差异在于——“传统的散列表” 主要是用于单机上的某个软件中；“分布式散列表”主要是用于分布式系统。

在DHT网络中客户端主机成为节点，存储的数据项称为对象。ID空间是系统的一个标识域，在这个ID空间中所有的标识符都是独一无二的。标识符用来标识节点和对象，每个节点和对象都有唯一的标识符。在DHT系统中，节点标识符可以通过对一个节点的名字(一般是节点的IP地址)进行散列获得；对象标识符可以通过对资源对象的名(文件名)进行散列来获得。

在DHT网络中，每个节点存储的是资源对象的<key,value>对。每个资源对象都有自己的关键字(一般长度不一，格式不一),通过对这些关键字进行散列之后可以得到资源对象的关键字key，而value值就是资源对象真实的存储位置(该资源的在主机中的存储位置或目录)，key与value的关系是(key = f(value))。需要注意：每个key对于的value与节点标识符不是一个东西。Value是资源对象具体的存储位置(深入到主机下的文件夹、文件)，节点标识符仅仅是对主机节点名字(IP地址)的散列（后者是散列值，前者不是散列）。

这样子，整个DHT网络就形成了一张巨大的<key,value>的文件索引散列表。每个节点都按照某种规则只维护一部分散列表（这里的规则依DHT算法的不同而不同）。用户在查询资源对象时，就是将资源对象关键字再次进行散列获取其资源关键字key，通过DHT算法路由获得到该key在整个DHT网络中的真实存储位置value(匹配查询)。

每个节点在加入网络的时候，会对存储在本节点上的资源进行索引并建立索引表，以满足本地资源检索的目的。然后该节点会按照某种预定的规则选择一些节点作为自己的邻居节点，加入到P2P网络中。

当网络中某节点发起查询请求q，它会将q发送给自己的邻居节点。它的邻居节点收到这个查询请求q之后，它会检查本身是否有查询的信息，如果不存在，继续转发该查询请求q，直到返回结果。

当一个节点离开P2P网络时，它会利用DHT算法通知网络内有关的节点，并将自身负责的资源通过算法内部的某种机制让网络中其他节点负责，使得整个网络能够正常地维护。另外，在DHT网络中，还有探测和容错机制，以保证系统的可用性。

1. **DHT功能及实现**

分布式散列表（DHT）的主要功能包括3项：①标识符的生成和管理；②提供重叠网络中的查询定位路由服务；③对提供的服务或文件的信息进行管理。

标识符的生成包括节点标识符ID和资源关键字key的生成。节点加入DHT网络时，将该节点的信息（例如IP）进行散列得到全局唯一的标识符作为节点标识符ID，资源发布到DHT网络时，该资源的信息（例如名称）进行散列得到全局唯一的标识符作为资源关键字key。而标识符的管理指的是节点在动态加入和退出DHT网络时系统的处理方式，具体处理方式根据算法的不同而存在差异。

提供覆盖网络中的查询定位的路由服务是DHT的基本功能，也是现有的DHT架构的热点研究问题。由于整个DHT是按照一定的组织形态存储在不同的网络节点上，某一个节点只负责存储一部分。而当某一个节点需要用到其他节点的资源时就必须在这个由众多DHT节点构成的网络中查询所用到的资源的具体位置。在查询时，资源请求会让节点查询自身所存储的路由表，如果目标节点具体信息在路由表中，则直接完成查询过程。如果没有，则向相应的目标节点(比如邻居节点)发送查询请求，接收到查询消息的节点运行与发起查询的节点相同的查找过程，直到完成目标节点的定位。

1. **DHT的分类**

按照路由复杂度可以将DHT算法分为Muti-hop DHT、O(1)DHT、One-hop DHT三类

* 1. **Muti****-hop DHT**

多跳的DHT算法，主要指在一个N节点的网络中，系统的路由跳数为O(logN)，且每个节点连接到O(logN)个邻居。在该类DHT算法中，节点用于维护节点间信息变化的消耗较低，但在路由查找时时延很大。当系统规模很大时，节点数量可能超过百万，这样的查询效率显然难以满足使用的需要。即使用户能够忍受漫长的时延，查询过程中产生的大量消息也会给网络带来不必要的负荷。而且当节点抖动较强烈时，对网络拓扑的影响也比较大。

此类典型的算法如：Chord、Pastry、CAN、Kademlia

* 1. **O(1)DHT**

平均路由跳数为常数。这类算法的特点是路由复杂度不随网络规模的变化而变化。在该类DHT算法中，为保持一定的路由跳数，需要对数据和节点信息使用相同的机制(如复制机制)，从而导致需要移动资源，系统同时产生了额外的开销，但是相比Multi-hop DHT算法路由的时延减小了很多。

典型算法：Kelips、Beehive等

* 1. **One-hop DHT**

每个节点将存储全局的路由表，从而实现单跳查询。为了确保路由的正确性，每个节点维持的路由表再节点信息变化时都需要及时进行维护。在动态环境中，节点的加入和退出是很频繁的，因此会导致路由维护的代价消耗很大。

此类算法也是O(1)DHT算法的一种，但是是比较特殊的一种。

1. **DHT算法—Chord**

在Chord算法中，通过使用散列函数(Hash)，将每个节点的信息进行散列得到唯一的节点标识符（Node ID）,同时对每一个资源的信息（资源名）进行散列得到唯一的key值。对于每一个资源对象的关键字key，Chord选择将关键字存储到相应的节点上，由这个节点来存储该对象的<key,value>对。而Chord中关键字key与节点的存储映射关系是将关键字key存储在节点标识符（Node ID）等于或大于key值(Node ID ≥ key)的第一个节点上。当需要查找资源时，采用同样的方法，先通过散列运算得到资源的key值，然后根据key与节点的映射关系，可以快速定位到存储资源的节点上。

在Chord系统中，节点并不需要知道网络中其他所有节点的信息，而仅需要保存部分节点的路由信息，通过与其他相邻节点进行通信从而完成路由查询。

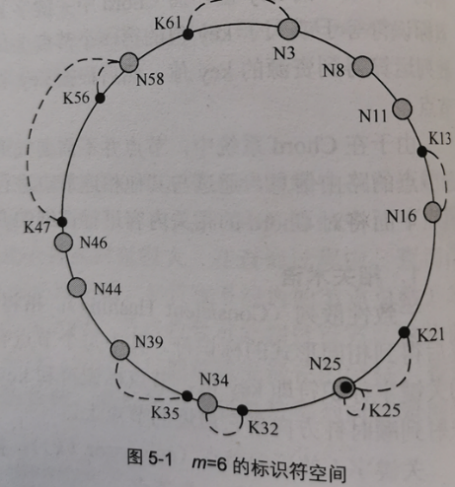
* 1. **相关术语**
* 一致性散列（Consisten Hashing）：是指将节点信息（如IP地址）和资源信息经过散列运算后得到相同形式的标识符（Node ID）。其中每个节点有其唯一的节点标识符，每个资源内容有其唯一的关键字标识符key值(key值由其相应的value散列得到)，节点标识符和key值位于同一值域。在Chord算法中，将关键字key映射到顺时针方向离他最近的节点上。
* 关键字k的后向节点（successor（k））：指的是存储关键字k的节点，该节点的节点标识符（Node ID）是等于或大于k的第一个节点
* 关键字k的前向节点（predecessor（k））：指的是在Chord环中关键字k的逆时针方向上的第一个节点
* 节点的后向节点（successor）：一般指某个节点的后向节点，即Chord环中某节点在顺时针方向上的第一个节点
* 节点的前向节点（predecessor）：Chord环中某节点在逆时针方向上的第一个节点
* 后向节点列表（successor-list）：一般指某个节点的r个后向节点组成的列表，即Chord环中的某个节点在顺时针方向上最近的r个连续的节点组成的列表。
  1. **标识符空间**

每个参与P2P网络的节点信息通过散列函数（典型的如SHA-1）映射成为m bit的唯一的节点标识符。同时，Chord采用同一个散列函数，把每个资源的信息(value)映射为m bit的关键字标识符，即key值。由此得到的节点标识符和关键字标识符在数值上都是[0,2^(m-1)]区间内的整数。比如IP地址为192.168.0.5的节点经过SHA-1散列运算后得到的标识符为123，而“LetItBe”散列运算后的key值为60。将所有的节点和资源按照它们的标识符值从小到大按顺时针方向排列并形成一个环形的标识符域，称为标识符空间，也成为了Chord标识符环。

在散列运算时，散列函数Hash将输入参数（如节点名字、资源名称）进行映射，生成一个高度随机的数值。因此节点标识符和关键字标识符的分布不是连续的，在Chord环上体现为，有些位置是存在节点，而有些位置为空。同时，由于节点和关键字共用一个标识符空间，标识符长度m必须足够长，这样才能保证节点或资源的标识符的唯一性。

Chord系统中，每个关键字都保存在它的后向节点中，也可以说，每个节点只负责保存以他为后向节点的关键字的信息。由此实现了信息的分布式存储。

下图是一个Chord标识符环，假设标识符比特数为m = 6，则标识符空间大小为2^6=64，即标识符的取值范围是[0,63]中的整数。给定的资源有8个，首先对资源进行散列运算得到关键字标识符，用K表示，分别为K13、K21、K25、K32、K35、K47、K56、K61；节点有10个，用N表示对节点信息进行散列运算得到的节点标识符，分别为N3、N11、N16、N25、N34、N39、N44、N46、N58。



标识符按顺时针方向从小到大的顺序组织起来形成一个环。关键字标识符K13在Chord环顺时针方向的下一个节点（即后向节点）为N16，则由N16保存K13的信息；同理，在顺时针方向上的节点标识符数值大于21的第一个节点为N25,由N25保存K21的信息。特殊的，K25的后向节点正是与它的关键字标识符在数值上相等的N25,K25的信息由N25存储；由于Chord的环形结构，K61的后向节点为N3。

* 1. **Chord路由**

在分布式环境中，在最简单的情况下每个节点只需要知道它在Chord标识符环上的后向节点是谁即可进行定位操作。我们可以设想，一个最简单的查询应该是这样一个过程：当在节点n上查找一个资源时，首先将资源关键字进行散列得到key值，然后判断该key值是否在节点n与其后向节点（successor）之间。如果是，返回当前节点的后向节点作为查询结果；如果不是，则将查询信息发送给节点n的后向节点，再在n的后向节点与它的后向节点上进行相同的操作，一直往下找到保存该关键字的节点。

以上的这种方法虽然只要保证后向节点信息正确就可以保证查找的正确性，但是查询速度比较慢，可能要遍历网络中所有的节点才能找到目标。因此，为了缩短查找过程，Chord中的每个节点都需要维护额外的路由信息，每次都将查询请求转发到节点所知的离请求关键字最近的节点上，实现关键字的快速定位。

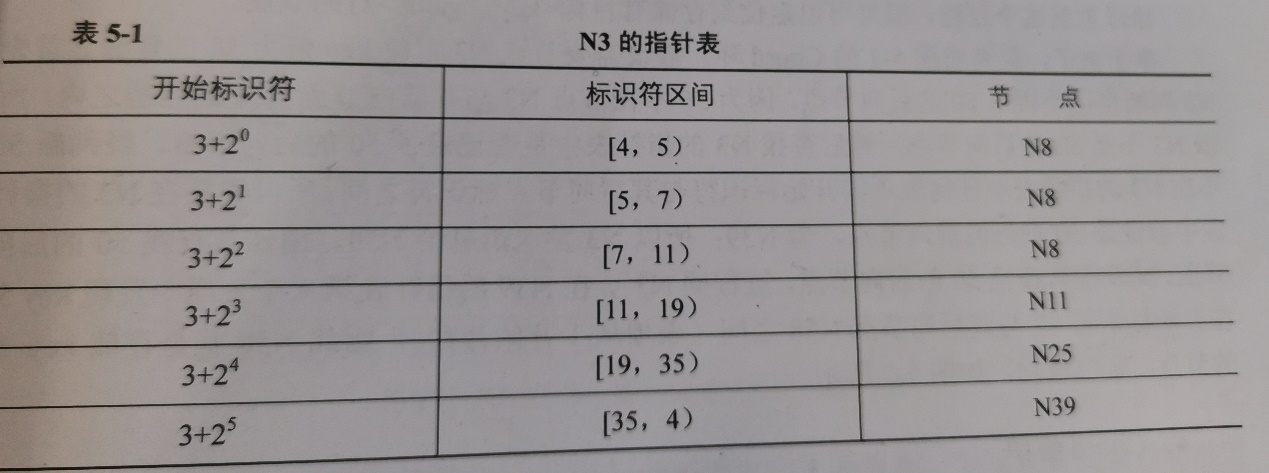
下面详细介绍Chord节点维护的路由信息和资源查找路由过程。

* + 1. **节点维护的信息**

同样的，关键字标识符和节点标识符都用m bit表示。每个节点n需要维护3个信息：前向节点（predecessor）；后向节点列表（successor-list）；一张最多有m条信息的指针表（finger table）。其中前向节点和路由表信息在路由过程中发挥了重要作用，而后向节点列表用于维护Chord环的稳定。

节点n的指针表包括m个表项，每个表项包含3个内容：开始标识符（start）、标识符区间（interval）、节点（node）。节点n的指针表中的第i个表项是这样构造的：开始标识符，为（n+）mod；标识符区间，为第i项与第i+1项的开始标识符之间的区域（包含第i项的开始标识符，但不包括第i+1项的开始标识符）；节点，为该表项的开始标识符在Chord环中顺时针方向上的下一个节点，性质类似于后向节点，表示为s=successor（n+），s称为节点n的第i个指针。这里1≤i≤m，且所有项都必须对取模运算。

仍以图5-1为例，给出的节点N3的典型指针表如下表所示。整个指针表共有6个表项。第一个表项的开始标识符为3+=4；标识符区间为[4,5） ;节点是以4为标识符的后向节点，为N8。



指针表的这种结构有两个重要特征。第一，每个节点都只需要存储少量其他节点的信息，且对在Chord环上离自己较近的节点的信息了解得比离它远的节点更少。第二，单个节点的指针表没有存储足够的信息来直接决定任意一个标识符的后向节点。例如，节点N3无法依靠自己存储的信息来直接得到标识符为13的后向节点，因为其后向节点N16没有出现在节点N3的指针表里。

当一个节点n并不知道关键字标识符k的后向节点该怎么办？如果节点n能够在自己的指针表中找到比自己的节点标识符更加靠近k的节点，那么这个节点会比自己更加了解k的信息。根据这一特性，n将查找它的指针表，找到节点标识符大于关键字标识符的第一个节点j，然后向j询问key k的后向节点是谁，通过重复这样的操作，最终能够找到k的后向节点。

* + 1. **资源查找过程**

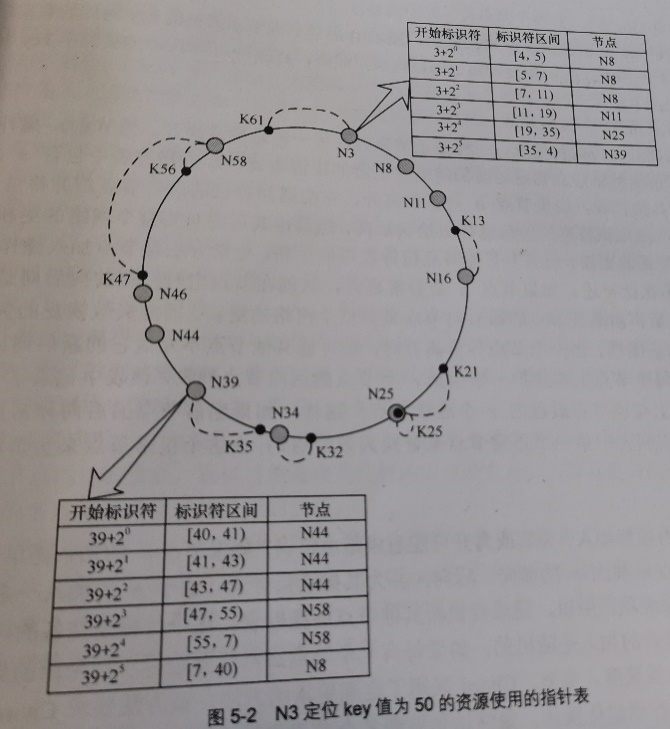
当一个节点n收到查询某个资源的请求时，首先对这个资源的信息进行散列运算，得到关键字标识符key，假设为k，然后在P2P网络上查找k的后向节点，找到后向节点就相当于找到了资源。由于每个节点都维护了其前向节点、路由表信息，节点在查找k的后向节点时，分为以下几个步骤：

1. 判断这个k值是否落在n与n的前向节点之间，如果是，则节点n就是该资源的后向节点，否则转②
2. 查找节点n的指针表，在指针表中找到k值所属的标识符区间，在判断这个key值是否在该表项的开始标识符和其后向节点之间，如果在，那么这个表项中的后向节点就是存储目标<key,value>对的节点，查找结束，否则转③
3. 在指针表中查找k的前向节点，即逆时针方向上离k最近的节点。首先在节点n的指针表的节点一列中查询节点标识符最大但是不超过k的节点，该节点是根据节点n的指针表确定的k的前向节点，并发送查询请求消息给这个前向节点，通知其查找k值的后向节点，转①。

通过重复这个过程，最终可以定位到存储有目标<key,value>对的节点。

举例，仍考虑图5-1的Chord环。假设需要节点N3定位key值为50的资源。

首先，N3判断自己是否是50的后向节点，因为50不在节点N3与其前向节点N58的范围之内，所以N3不是50的后向节点；然后查找N3的指针表中是否记录了50的后向节点，经判断50不在N3的指针表中任何表项的开始标识符与其后向节点标识符之间；所以需要在N3的指针表中查找距离50最近的前向节点，为N39；所以N3发送消息给N39，通知其查找50的后向节点。N39继续查找50的后向节点，过程同N3，在N39的第4个表项中发现key 50 在开始标识符47与其后向节点N58之间，因此N58就是存储key 50 的节点，定位成功，如下图所示。



* 1. **Chord维护**

在动态网络中，节点可能随时加入或离开，为了保证网络中能够准确地对资源进行定位，Chord需要保证两个不变的量：第一，每个节点都有正确的后向节点；第二，每个key值都需要由正确的后向节点维护。第三，为了实现快速查询，指针表也必须保证正确。因此，Chord维护的主要目标就是保证这三者的正确性。

Chord环中的每个节点都需要维护节点的前向节点信息，一个作用是帮助路由查询，另一个重要作用就是它可以简化节点的加入和离开机制。

* + 1. **节点加入**

在Chord中，新节点n加入时必须完成以下3个步骤：

1. 初始化节点n的前向节点、后向节点列表、指针表信息
2. 更新已经存在的其他相关节点维护的信息
3. 转移关键字信息。其他节点将所有关键字键值应归属于n的关键字key转移到节点n上，这些key对应的数据项应该由n来维护。

具体来说，当一个新节点n加入Chord环时，首先需要确定Chord环上的一个已经存在的节点n’，通过节点n’对n进行初始化并将其加入到自己的网络中，具体过程如下：

（1）初始化节点n的前向节点、后向节点列表、指针表信息。节点n’首先会查询获得节点n的后向节点，记为节点s；获取后向节点s的当前前向节点作为n的前向节点；向s请求它的后向节点列表，将s后向列表的前（r-1）项和s作为节点n的后向节点列表成员；构造n的指针表

（2）节点n加入以后，更新已经存在的其他相关节点维护的信息。相关节点包括n的前向、后向、逆时针方向的前r个节点以及那些在节点n加入以后必须更改路由表的节点。

更新n的前向、后向节点的过程为：修改n的后向节点的前向为n，修改n的前向节点的后向为n。

更新n的前r个节点的后向节点列表的过程为：将原后向节点列表中保存的从s开始的所有节点向后移动一位，移出的一位从后向节点列表中删除，节点n作为后向节点列表中的一项放在原节点s的位置上。

更新当n加入以后必须改变的路由表节点的过程较为复杂。当且仅当以下两个条件都成立时，节点n将成为节点p的第i项中的后向节点：（a）p先于n至少，即p+≤n；（b）p的第i项，即开始标识符为（p+），它的后向节点为n。将满足这两个条件的p的相应项更新为n。

很高的概率下，一个节点加入网络所需要更新的节点数是O(logN)，找到并更新这些节点需耗时O((logN)^2)。

（3）转移关键字信息。节点n加入以后，原本由n的后向节点s维护的key对应的数据项应该由n来维护，因此节点n需要和自己的后向节点联系，将应该属于n的关键字key转移到节点n上。

* + 1. **节点离开**

节点的离开分为两种情况，一种是正常离开，另一种是异常离开。当节点n离开之后，为了保证在系统重新建立稳定之前仍能进行正常的资源查询，那些指针表中包含n的节点需要更新其指针表。

如果节点n是正常离开的，它会通知自己的后向节点，并将自己负责存储的那一段标识符空间的信息拷贝给其后向节点，然后由其后向节点负责整个网络的更新，包括了前向后向关系的更新、网络上其他节点指针表项的更新，更新的方法是节点加入操作时的逆向过程。

如果节点n是异常离开的，这种情况只能在节点定时检查其前后向节点是否存在时发现，然后再由离开节点的原后向节点负责整个网络的更新。因此失效恢复的关键点就在于确保后向的正确性。当一个节点异常离开时，很可能其他节点不知道它的新的后向节点将会是谁，为了避免这种情况，每个节点上都会维护一张包括r个节点的后向节点列表，该表中记录了Chord环顺时针方向上与该节点最近的r个连续节点。这样，如果当前节点的后向节点异常退出，该节点就用后向列表中第一个正常节点来替换失效节点，并由这个正常节点来负责网络的更新（因为它是异常退出节点的原后向节点）。

* + 1. **稳定化**

Chord节点的频繁加入、失效或离开可能造成路由错误。如果节点n的加入是单一的，n可以通过已知节点n’找到n的前向、后向，并为其构造指针表。由于n的加入一定会影响到某些节点的路由表项，所以，随后要更新其他节点的指针表。这个过程是完备的，但是在网络环境中，节点的加入时随机的，如果每有一个节点加入就需要进行一次全面的更新，系统的消耗将很大，影响查找的速度。为此，Chord采用定期更新的方法。该方法要求Chord环中的节点必须周期性进行稳定化操作，最终达到更新各个节点指针表中错误表项的目的。

当已存在的节点n进行稳定化操作时，它首先向Chord网络中任意一个已知节点n’查询获得n的后向节点，然后请求n的后向节点返回它的前向节点p。如果p=n，则表明n和它的后向互相承认作为对方的前向和后向；如果p位于n和n的后向节点之间，则表明p是刚刚加入到网络中的，节点n将其后向节点更新为p，p将以n作为其前向节点。然后采用节点加入时的方法对网络中其他节点维护的信息进行更新。

* 1. **Chord算法特点**

在一个节点数量为N的稳定的Chord网络中，每个节点需要维护O(logN)个其他节点的路由信息，路由复杂度为O(logN)。当节点加入或离开网络时，Chord需要传递O((logN)^2)条消息来更新路由消息。

除了以上路由性能上的特点，Chord具有以下优势：

负载平衡：Chord采用分布式散列函数，所有节点以同等的概率分担系统负荷，从而避免了某些节点负载过大的问题，此算法本身就保证了负载的平衡性。

分布性：Chord是一个全分布式系统，所有节点完全平等。这一特点不仅使Chord系统具有很好的健壮性，而且更适合于松散的P2P应用系统。

可扩展性：Chord系统的路由开销随着系统规模（节点总数为N）的增加以O(logN)的速度增长，因此即使是很大规模的系统也是切实可行的。且获得这种可扩展性不需要进行任何参数的调整。

有效性：Chord能够根据节点的加入或离开自动地更新路由表，即使系统状态在不断变化，仍能够确保节点查询可靠地运行。

灵活的命名：Chord没有对关键字设置任何的限制，也就是说Chord的关键字是扁平的。在应用程序给关键字设置自己所需要的名字时具有很大的灵活性

同时，Chord也存在一些问题：

1. Chord网络的绕路（Detouring）问题。Chord在构建网络时没有考虑节点的实际物理拓扑结构，忽视了覆盖网络与底层网络的差异，可能导致在实际网络中相距很近的两个节点通过散列映射在覆盖网络上相距很远。这就是Chord网络的绕路
2. 未考虑节点的性能差异。Chord网络不加区分地赋予网络中所有节点同样的责任，这可能导致高性能的节点没有充分发挥其能力，而相对性能较差的节点却担负着无法担负的责任。
3. 指针表信息冗余。由于Chord算法构造指针表的方法和Chord环形结构的特点，使得信息冗余在指针表中不可避免。显然，在Chord节点的指针表中，随着i的增大，标识符区间范围成2倍增长。因此当i较小时，标识符区间也较小，其中的节点数可能很少甚至没有，导致节点指针表的前几行的表项内容重复。
4. **DHT算法—Kademila**
   1. **概述**

Kademlia是一种点对点分布式哈希表（DHT），它在容易出错的环境中也具有可证明的一致性和性能。使用一种基于异或指标的拓扑结构来路由查询和定位节点，这简化了算法并有助于证明。该拓扑结构有一个特点：每次消息交换都能够传递或强化有效信息。系统利用这些信息进行并发的异步查询，可以容忍节点故障，并且故障不会导致用户超时。

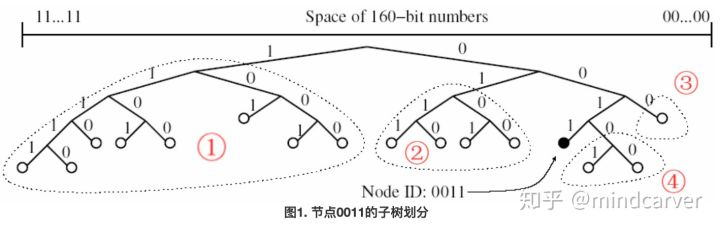
* 1. **节点状态**

节点的基本属性包括如下：

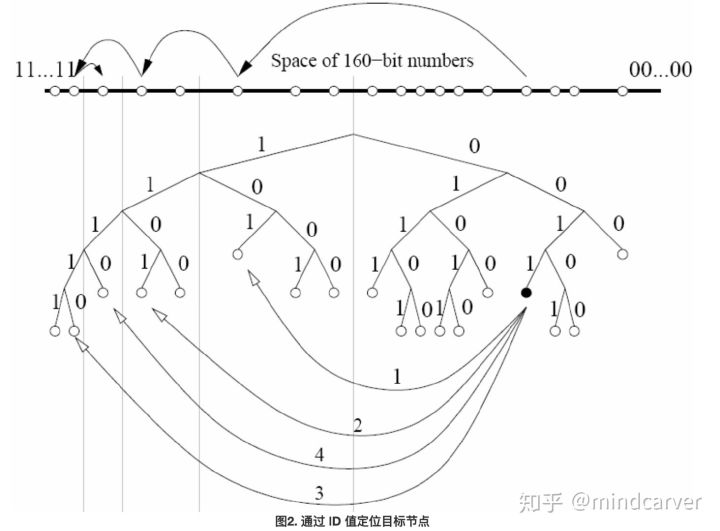
* 节点ID，Node ID
* 节点IP地址与端口号

在 Kad 网络中，所有节点都被当作一颗二叉树的叶子，并且每一个节点的位置都由其 ID 值的最短前缀唯一的确定。

对于任意一个节点，都可以把这颗二叉树分解为一系列连续的，不包含自己的子树。最高层的子树，由整颗树不包含自己的树的另一半组成；下一层子树由剩下部分不包含自己的一半组成；依此类推，直到分割完整颗树。图 1 就展示了节点0011如何进行子树的划分：



虚线包含的部分就是各子树，由上到下各层的前缀分别为0，01，000，0010。Kad 协议确保每个节点知道由其分割的各子树的至少一个节点，只要这些子树非空。在这个前提下，每个节点都可以通过ID值来找到任何一个节点。这个路由的过程是通过所谓的 XOR（异或）距离得到的。图 2 就演示了节点0011如何通过连续查询来找到节点1110的。节点0011通过在逐步底层的子树间不断学习并查询最佳节点，获得了越来越接近的节点，最终收敛到目标节点上。



需要说明的是:只有第一步查询的节点101，是节点0011已经知道的，后面各步查询的节点，都是由上一步查询返回的更接近目标的节点，这是一个递归操作的过程。

* 1. **节点距离**

Kad 网络中每个节点都有一个 160 bit 的 ID 值作为标志符，Key 也是一个 160 bit 的标志符(在系统中使用160bit来对对象进行标识是为了减小标识符或键值相同的可能性)，每一个加入 Kad 网络的计算机都会在 160 bit 的 key 空间被分配一个节点 ID（node ID）值（可以认为 ID 是随机产生的）， <key,value> 对的数据就存放在 ID 值“最”接近 key 值的节点上。

判断两个节点 x,y 的距离远近是基于数学上的异或的二进制运算， d(x,y)=x⊕y ，既对应位相同时结果为0，不同时结果为1。例如：

010101

XOR 110001

----------

100100

则这两个节点的距离为 32+4=36 。显然，高位上数值的差异对结果的影响更大。对于异或操作，有如下一些数学性质：

两个节点间的距离是随机的

节点与自身的距离是0

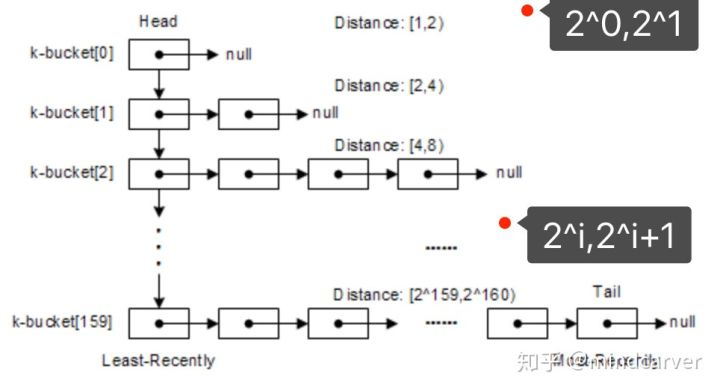
对称性:A 到 B 的距离和 B 到 A 的距离相等

三角不等:distance(A,B)+distance(B,C) <= distance(A,C)

对于任意给定的节点 x 和距离 Δ≥0 ，总会存在一个精确的节点 y ，使得 d(x,y)=Δ 。另外，单向性也确保了对于同一个 key 值的所有查询都会逐步收敛到同一个路径上，而不管查询的起始节点位置如何。

* 1. **K桶**
     1. **K桶的概念**

Kad 的路由表是通过一些称之为 K 桶的表格构造起来的。每一个节点一般包含有160个K桶，第i个K桶存储了距离本节点距离为的节点的属性信息(IP地址，UDP端口号，节点ID)。并且每个 K 桶内部节点的存放位置是根据最后联系时间进行排列的，最近联系的节点在K桶的尾部(Tail)，最早联系的节点在K桶的头部(Head)。一个节点的全部 K 桶列表如下图所示：



Kad规定：每个桶都有不超过 k 个的数据项。当 i 值很小时，K 桶通常是空的（也就是说没有足够多的节点，比如当 i = 0 时，就最多可能只有1项）；而当 i 值很大时，其对应 K 桶的项数又很可能会超过 k 个（当然，覆盖距离范围越广，存在较多节点的可能性也就越大），这里 k 是为平衡系统性能和网络负载而设置的一个常数，但必须是偶数，比如 k = 20。在 BitTorrent 的实现中，取值为 k = 8。

由于每个 K 桶覆盖距离的范围呈指数关系增长，这就形成了离自己近的节点的信息多，离自己远的节点的信息少(因为i值大的K桶最多也就只能存储k条消息，多余的距离比较远的节点可能没地方存储)，从而可以保证路由查询过程是收敛。因为是用指数方式划分区间，经过证明，对于一个有 N 个节点的 Kad 网络，最多只需要经过 logN 步查询，就可以准确定位到目标节点。

* + 1. **K桶的更新机制**

当接收者节点 x 收到一个 PRC 消息时，发送者 y 的 IP 地址就被用来更新对应的 K 桶，具体步骤如下：

1. 计算自己和发送者的距离： d(x,y)=x⊕y ，注意：x 和 y 是 ID 值，不是 IP 地址
2. 通过距离 d 选择对应的 K 桶进行更新操作
3. 如果 y 的 IP 地址已经存在于这个 K 桶中，则把对应项移到该 K 桶的尾部(按联系时间排序)
4. 如果 y 的 IP 地址不存在于该 K 桶中，有以下两种可能性：
5. k桶中节点的数量少于k个(未满)，将节点A添加到k桶的底部。
6. 如果k桶已经满了，向k桶头部节点z发送ping

* 如果 z 没有响应，则从 K 桶中移除 z 的信息，并把 y 的信息插入队列尾部
* 如果 z 有响应，则把 z 的信息移到队列尾部，同时忽略 y 的信息

K 桶的更新机制非常高效的实现了一种把最近看到的节点更新的策略，除非在线节点一直未从 K 桶中移出过。也就是说在线时间长的节点具有较高的可能性继续保留在 K 桶列表中。所以，通过把在线时间长的节点留在 K 桶里，Kad 就明显增加 K 桶中的节点在下一时间段仍然在线的概率，这对应 Kad 网络的稳定性和减少网络维护成本（不需要频繁构建节点的路由表）带来很大好处。

这种机制的另一个好处是能在**一定程度上防御 DOS 攻击**，因为只有当老节点失效后，Kad 才会更新 K 桶的信息，这就避免了通过新节点的加入来泛洪路由信息。

为了防止 K 桶老化，所有在一定时间之内无更新操作的 K 桶，都会分别从自己的 K 桶中随机选择一些节点执行 RPC\_PING 操作。

上述这些 K 桶机制使 Kad 缓和了流量瓶颈（所有节点不会同时进行大量的更新操作），同时也能对节点的失效进行迅速响应。

* 1. **协议消息**

Kademlia 协议包括四种远程 RPC 操作：PING、STORE、FIND\_NODE、FIND\_VALUE。

1. PING 操作的作用是探测一个节点，用以判断其是否仍然在线。

2. STORE 操作的作用是通知一个节点存储一个 <key,value> 对，以便以后查询需要。

3. FIND\_NODE 操作使用一个 160 bit 的 ID 作为参数。本操作的接受者返回它所知道的更接近目标 ID 的 K 个节点的 (IP address, UDP port, Node ID) 信息。这些节点的信息可以是从一个单独的 K 桶获得，也可以从多个 K 桶获得（如果最接近目标 ID 的 K 桶未满）。不管是哪种情况，接受者都将返回 K 个节点的信息给操作发起者。但如果接受者所有 K 桶的节点信息加起来也没有 K 个，则它会返回全部节点的信息给发起者。

4. FIND\_VALUE 操作和 FIND\_NODE 操作类似，不同的是它只需要返回一个节点的 (IP address, UDP port, Node ID) 信息。如果本操作的接受者收到同一个 key 的 STORE 操作，则会直接返回存储的 value 值。

为了防止伪造地址，在所有 RPC 操作中，接受者都需要响应一个随机的 160 bit 的 ID 值。另外，为了确信发送者的网络地址，PING 操作还可以附带在接受者的 RPC 回复信息中（**在上述 4种操作中 接受者回复 发送者时，可以携带上 接受者对 发送者的 PING, 以此校验 发送者是否还健在**）。

* 1. **路由查找**

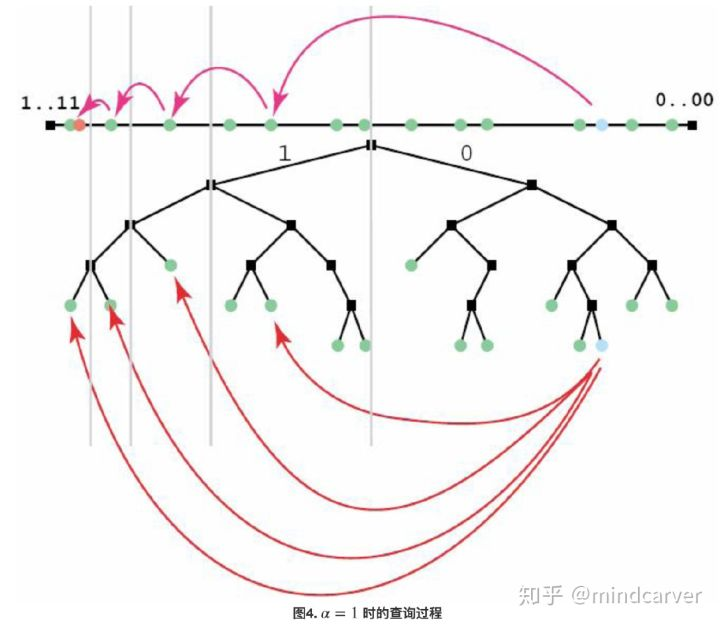
Kad 技术的最大特点之一就是能够提供快速的节点查找机制，并且还可以通过参数进行查找速度的调节。

假如节点 x 要查找 ID 值为 t 的节点，Kad 按照如下递归操作步骤进行路由查找：

1. 计算到 t 的距离： d(x,y)=x⊕y
2. 从 x 的第 [] 个 K 桶中取出 α 个节点的信息（“[ ]”是取整符号），同时进行 FIND\_NODE 操作。如果这个 K 桶中的信息少于 α 个，则从附近多个桶中选择距离最接近 d 的总共 α 个节点。
3. 对接受到查询操作的每个节点，如果发现自己就是 t，则回答自己是最接近 t 的；否则测量自己和 t 的距离，并从自己对应的 K 桶中选择 α 个节点(距离t最近的)的信息给 x。
4. X 对新接受到的每个节点都再次执行 FIND\_NODE 操作，此过程不断重复执行，直到每一个分支都有节点响应自己是最接近 t的(如果t真实存在于网络中，最后的一次结果必然是只有t本身对x进行了回复)。
5. 通过上述查找操作，x 得到了 k 个最接近 t 的节点信息。

注意：这里用“最接近”这个说法，是因为 ID 值为 t 的节点不一定存在网络中，也就是说 t 没有分配给任何一台电脑。

这里 α 也是为系统优化而设立的一个参数，就像 K 一样。在 BitTorrent 实现中，取值为 α=3 。当 α=1 时，查询过程就类似于 Chord 的逐跳查询过程，如图 4。



整个路由查询过程是递归操作的，其过程可用数学公式表示为：

|  |
| --- |
| N0=x (即查询操作的发起者) N1=find ⎯noden0(t) N2=find ⎯noden1(t) ... ... Nl=find ⎯nodenl−1(t) |

这个递归过程一直持续到 Nl=t ，或者 Nl 的路由表中没有任何关于 t 的信息，即查询失败。由于每次查询都能从更接近 t 的 K 桶中获取信息，这样的机制保证了每一次递归操作都能够至少获得距离减半（或距离减少 1 bit）的效果，从而保证整个查询过程的收敛速度为 O(logN) ，这里 N 为网络全部节点的数量。

当节点 x 要查询 <key,value> 对时，和查找节点的操作类似，x 选择 k 个 ID 值最接近 key 值的节点，执行 FIND\_VALUE 操作，并对每一个返回的新节点重复执行 FIND\_VALUE 操作，直到某个节点返回 value 值。

一旦 FIND\_VALUE 操作成功执行，则 <key,value>会被缓存在没有返回 value 值的最接近的节点上。这样下一次查询相同的 key 时就会更加快速的得到结果。通过这样的方式，热门 <key,value>的缓存范围就逐步扩大，使系统具有极佳的响应速度( 缓存的**cache 可以存活24小时**，但是目标节点上的内容会**每1小时**向其他最近节点重新发布使得数据的超时时间得以刷新，而远离目标节点的节点的数据存活时间当然就可能不会被重新发布到，所以也就是数据缓存的超时时间和节点的距离成反比)

* 1. **数据的存储**

存放 <key,value> 对数据的过程为：

1. 发起者首先定位 k 个 ID 值最接近 key 的节点
2. 发起者对这 k 个节点发起 STORE 操作
3. 执行 STORE 操作的 k 个节点每小时重发布自己所有的 <key,value> 对数据
4. 为了限制失效信息，如果在接收不到更新消息的情况下，相应的 <key,value>数据会在初始发布24小时后过期

另外，为了保证数据发布、搜寻的一致性，规定在任何时候，当节点 w 发现新节点 u 比 w 上的某些 <key,value> 对数据更接近，则 w 把这些 <key,value> 对数据复制到 u 上，但是并不会从 w 上删除。

* 1. **节点的加入和离开**

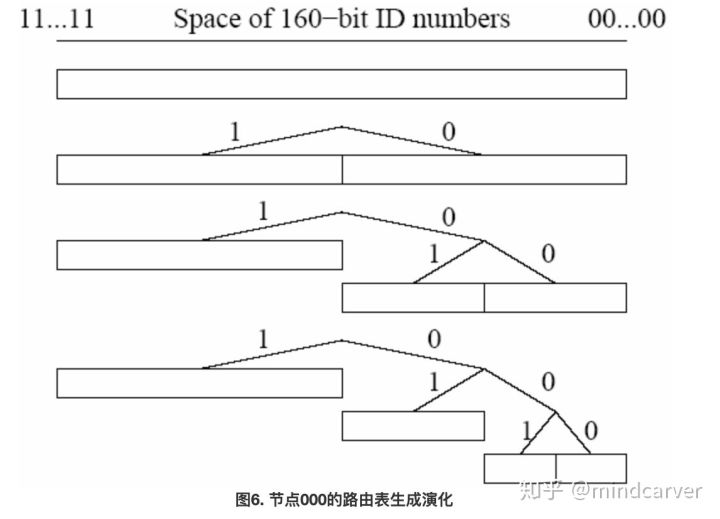
如果节点 u 要想加入 Kad 网络，它必须要和一个已经在 Kad 网络的节点，比如 w，取得联系。u 首先把 w 插入自己适当的 K 桶中，然后对自己的节点 ID 执行一次 FIND\_NODE 操作 (向 w 发布 查找 u 的 FIND\_NODE 请求)，然后根据接收到的信息更新自己的 K 桶内容。通过对自己邻近节点由近及远的逐步查询，u 完成了仍然是空的 K 桶信息的构建，同时也把自己的信息发布到其他节点的 K 桶中。

节点 u 为例，其路由表的生成过程为：

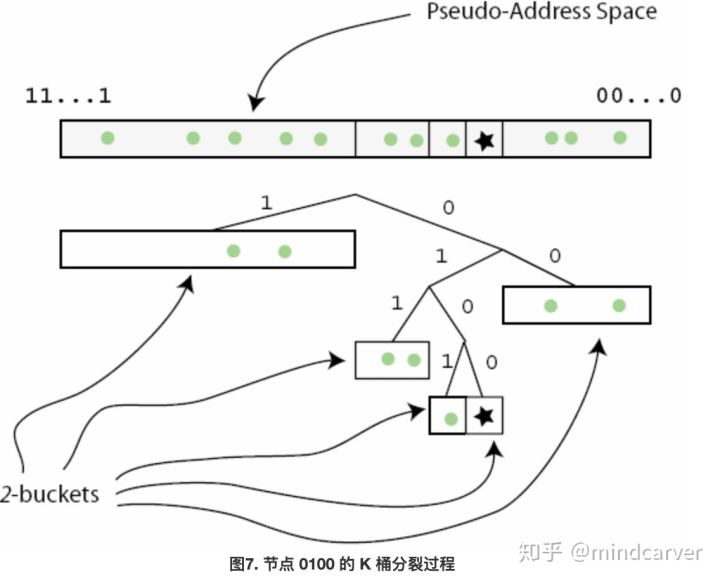
* 1. 最初，u 的路由表为一个单个的 K 桶(此时节点只有一个K桶)，覆盖了整个 160 bit ID 空间，如图 6 最上面的路由表；
  2. 当学习到新的节点信息后，则 u 会尝试把新节点的信息，根据其前缀值插入到对应的 K 桶中。
  3. 如果该 K 桶没有满，则新节点直接插入到这个 K 桶中；
  4. 如果该 K 桶已经满了，分为以下两种情况：

1. 如果该 K 桶覆盖范围包含了节点 u 的 ID的前缀，则把该 K 桶分裂为两个大小相同的新 K 桶，并对原 K 桶内的节点信息按照新的 K 桶前缀值进行重新分配。
2. 如果该 K 桶覆盖范围没有包含节点 u 的 ID的前缀，则直接丢弃该新节点信息

5. 上述过程不断重复，最终会形成表 1 结构的路由表。达到距离近的节点的信息多，距离远的节点的信息少的结果，保证了路由查询过程能快速收敛。



在图 7 中，演示了当覆盖范围包含自己 ID 值(所谓覆盖自己，就是指节点与该K桶的ID值的前几位是完全相同的)的 K 桶是如何逐步分裂的。



当 K 桶 010 (其前三位与节点0100是完全相等的)满了之后，由于其覆盖范围包含了节点 0100 的 ID，故该 K 桶分裂为两个新的 K 桶：0101 和 0100，原 K 桶 010 的信息会根据其其前缀值重新分布到这两个新的 K 桶中。注意，这里并没有使用 160 bit 的 ID 值表示法，只是为了方便原理的演示，实际 Kad 网络中的 ID 值都是 160 bit 的

**节点离开 Kad 网络不需要发布任何信息**，Kademlia 协议的目标之一就是能够弹性工作在任意节点随时失效的情况下。为此，Kad 要求每个节点必须周期性 【一般是： 每小时】 的发布全部自己存放的<key,value>数据，并把这些数据缓存在自己的 k 个最近邻居处，这样存放在失效节点的数据会很快被更新到其他新节点上。所以有节点离开了，那么就离开了，而且节点中的k-桶刷新机制也能保证会把已经不在线的节点信息从自己本地k-桶中移除。

k-桶刷新机制下，节点会对那些在过去一小时之内没有收到任何节点查询操作的k-桶执行刷新操作(BT协议实现规定为15min)。具体实现是从节点的某k-桶中选择一个随机的节点信息，并对该节点ID执行一次FIND\_NODE操作。