Consistent Hash Algorithm and Go Implementation

Kevin Wan · Follow
Published in FAUN—Developer Community 🐾 · 9 min read · Nov 29, 2021

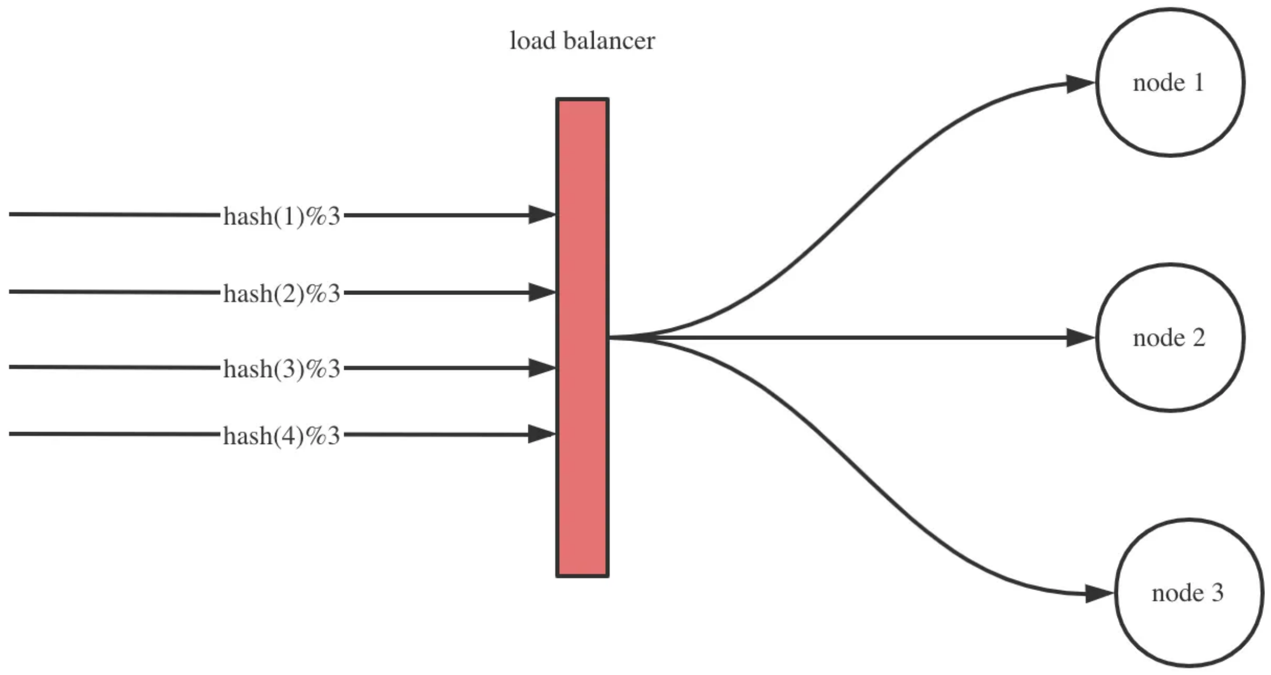
30

Why Consistent Hash is Needed

Hash is the transformation of an input of arbitrary length (also called a pre-image) into an output of fixed length by a hash algorithm, and the output is the hash value. This transformation is a compressed mapping, that is, the space of the hash value is usually much smaller than the space of the input, and different inputs may be hashed into the same output, so it is not possible to determine the unique input value from the hash value. Simply put, it is a function that compresses a message of arbitrary length into a message digest of some fixed length.

In a distributed caching service, node additions and deletions are often required for the service, and what we want is for the node additions and deletions to minimize the update of the data-node mapping relationship.

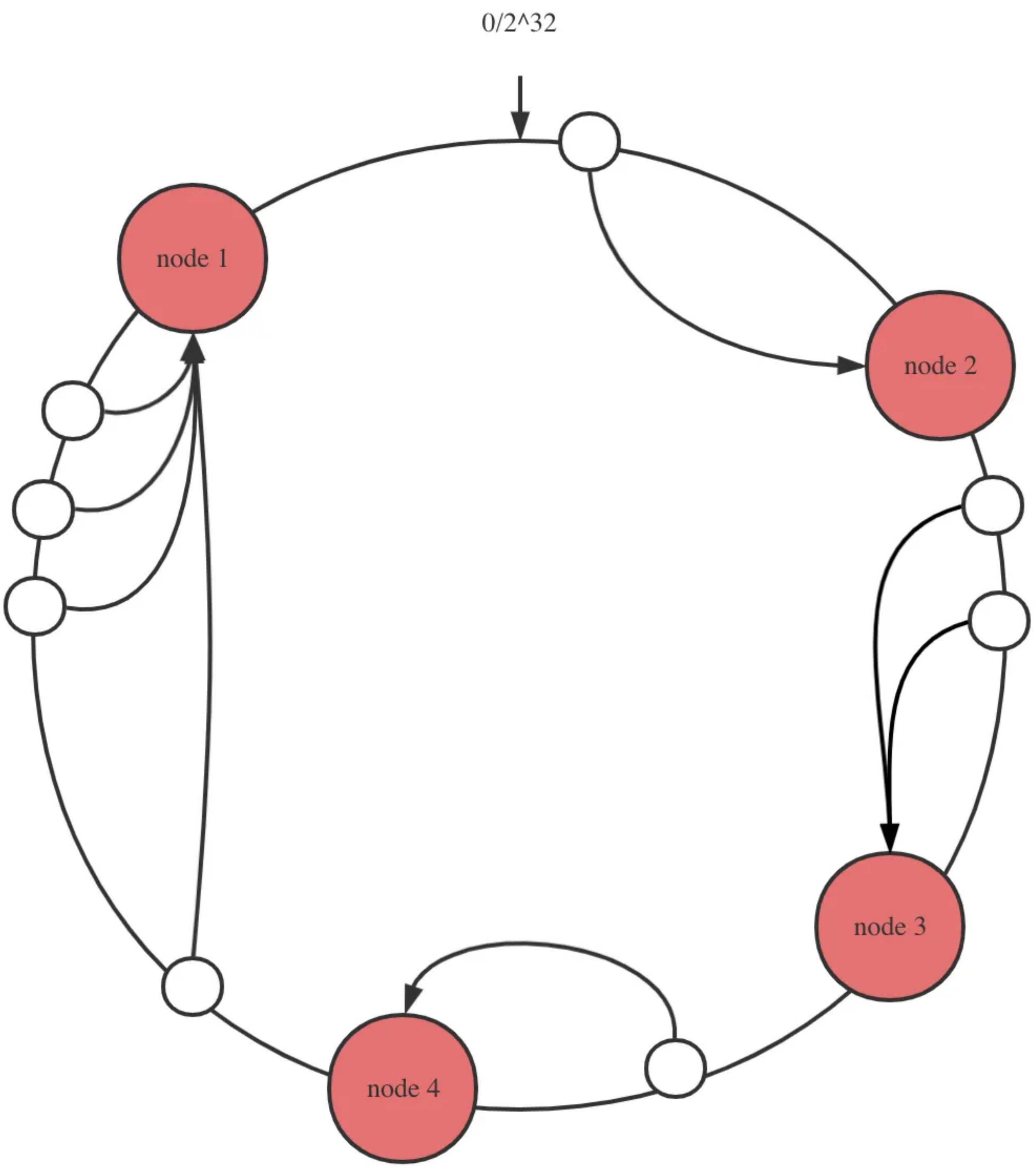
If we use the hash(key)%nodes algorithm as the routing policy:



The disadvantage of hash fetching is that if there are node deletions and additions, the impact on hash(key)%nodes results is too large, causing a large number of requests to fail, resulting in reloading of cached data.

Based on the above drawbacks a new algorithm is proposed: Consistent Hashing. Consistent hashing allows node deletion and addition to affect only a small portion of the data mapping relationship, due to this feature hashing algorithm is also often used in various equalizers to achieve smooth migration of system traffic.

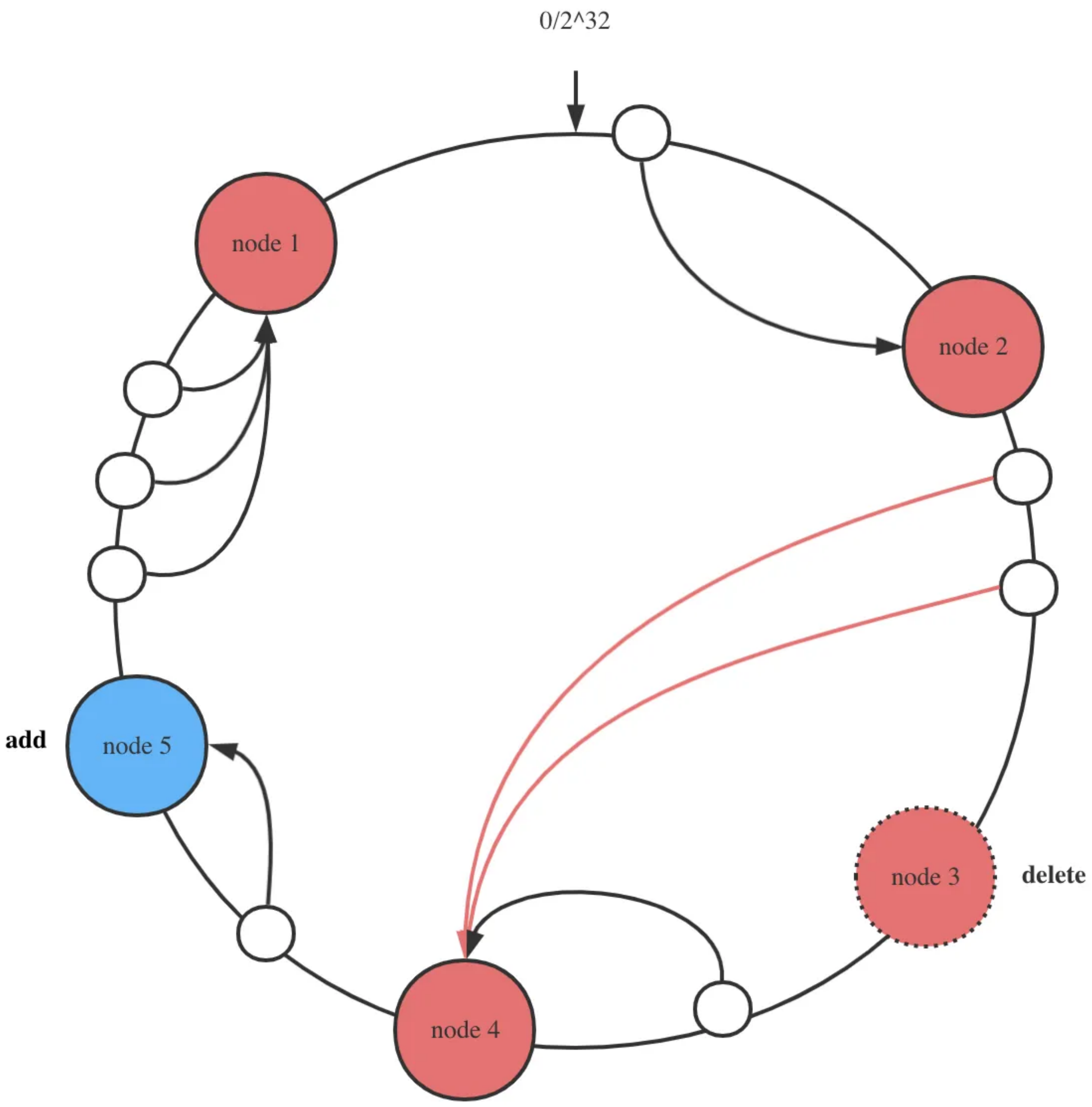
How Consistent Hashing Works



The nodes are first hashed and the hash value is usually in the range of 2^{32}-1. Then we abstract the 2^{32}-1 interval into a ring and map the node's hash value to the ring. When we want to query the target node of the key, we similarly hash the key and then the first node we find clockwise is the target node.

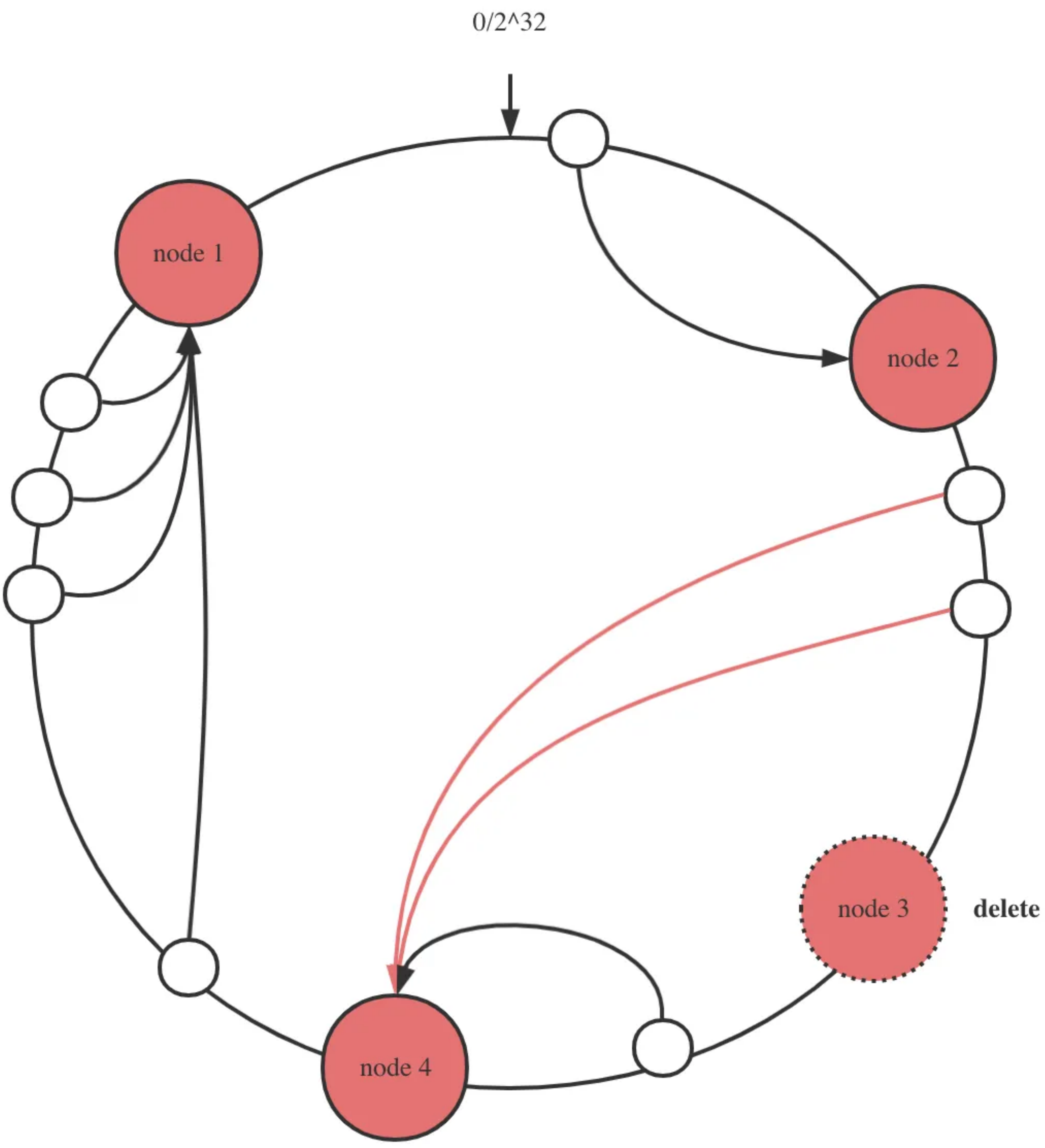
According to the principle, we analyze the effect of node addition and deletion on the data range.

Node addition



only affects the data between the added node and the previous node (the first node found counterclockwise by the added node).

Node deletion

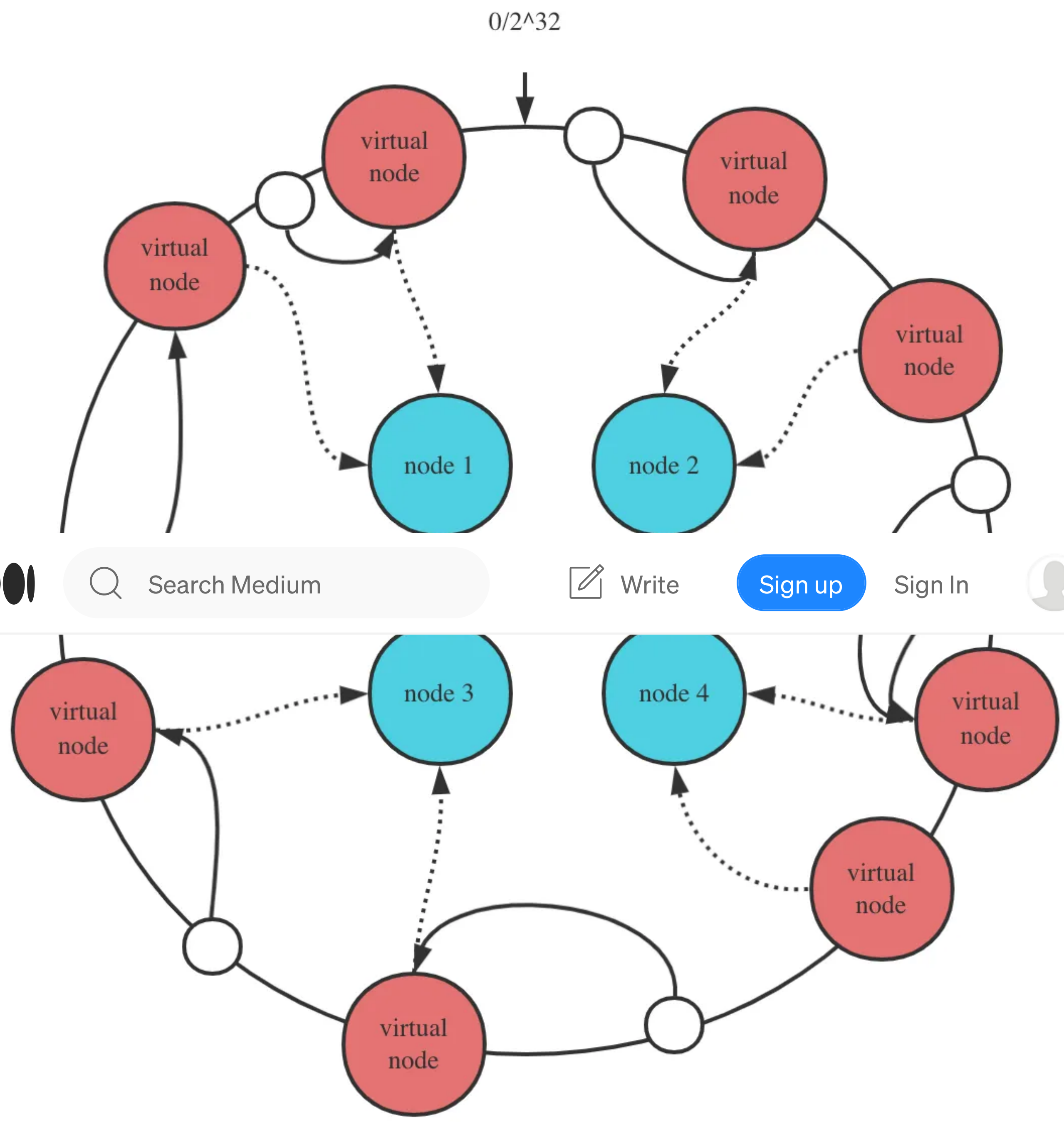


only affects the data between the deleted node and the previous node (the first node found counterclockwise by the deleted node).

Is that the end of it? No, imagine if the number of nodes on the ring is very small, then it is very likely that the data distribution will be unbalanced, essentially because the interval distribution on the ring is too coarse in granularity.

How to solve it? Isn't the granularity too coarse? Then add more nodes, which leads to the concept of virtual nodes of consistent hashing, the role of virtual nodes is to make the interval distribution of nodes on the ring finer granularity.

A real node corresponds to multiple virtual nodes, the hash value of the virtual node is mapped to the ring, and the target node of the query key we first query the virtual node and then find the real node.



code implementation

Based on the above consistent hashing principle, we can refine the core functions of consistent hashing.

1. add nodes
2. delete nodes
3. query nodes

Let's define the interface.

```
ConsistentHash interface {
    Add(node Node)
    Get(key Node) Node
    Remove(node Node)
}
```

In reality, different nodes may have different service capabilities due to hardware differences, so we want to specify weights when adding nodes. This is reflected in the consistency hash where the so-called weight means that we want the target node of the key to have a higher probability of being hit, and a real node with a higher number of virtual nodes means a higher probability of being hit.

In the interface definition, we can add two methods: support for adding nodes by specifying the number of virtual nodes, and support for adding nodes by weight. In essence, both of them will eventually reflect the difference in probability distribution due to the different number of virtual nodes.

When specifying the weight: actual number of virtual nodes = configured virtual nodes * weight/100

```
ConsistentHash interface {
    Add(node Node)
    AddWithReplicas(node Node, replicas int)
    AddWithWeight(node Node, weight int)
    Get(key Node) Node
    Remove(node Node)
}
```

Next, consider a few engineering implementation issues.

How are the virtual nodes stored?

It's simple, just store it in a list (slice).

virtual node — real node relationship storage

map works well.

clockwise query of the first virtual node how to achieve

Let the list of virtual nodes be ordered, dichotomously find the first index larger than hash(key), list[index] can be.

There is a small probability of conflict when hashing virtual nodes, how to deal with it?

The conflict means that the virtual node will correspond to more than one real node, the value in map stores the array of real nodes, and the nodes are modulo when the target node of the key is queried.

How to generate virtual nodes

Configure replicas based on the number of virtual nodes, loop the replicas and append i bytes in turn for hash calculation.

go-zero source code explained

`core/hash/consistenthash.go`

It took a day to read through the go-zero source code consistency hash source code, which is really well written, with all kinds of details taken into account.

The hash function go-zero uses is `MurmurHash3`, GitHub: https://github.com/spaolacci/murmur3

go-zero does not have an interface definition, it does not matter, look directly at the structure `ConsistentHash`.

```
// Func defines the hash method.
// Hash function
Func func(data []byte) uint64

// A ConsistentHash is a ring hash implementation.
// Consistent hash
ConsistentHash struct {
    // Hash function
    hashFunc Func
    // Determine the number of virtual nodes of the node
    replicas int
    // list of virtual nodes
    keys []uint64
    // Mapping of virtual nodes to physical nodes
    ring map[uint64][]interface{}
    // Physical node map, quickly determine if a node exists
    nodes map[string]lang.PlaceholderType
    // Read and write lock
    lock sync.RWMutex
}
```

Hashing of keys and virtual nodes

Convert key to string before hashing

```
// can be interpreted as a serialization method to determine the
node string value
// In case of a hash conflict, the key needs to be hashed again
// In order to reduce the probability of conflict, a prime time
is appended to reduce the probability of conflict
func innerRepr(v interface{}) string {
   return fmt.Sprintf("%d:%v", prime, v)
}

// can be interpreted as a serialization method that determines
the value of the node string
// Wouldn't it be better to have node force String() to be
implemented?
func repr(node interface{}) string {
   return mapping.Repr(node)
}
```

Here the `mapping.Repr` will determine the `fmt.Stringer` interface and if it matches, it will call its `String` method. The `go-zero` code is as follows.

```
// Repr returns the string representation of v.
func Repr(v interface{}) string {
    if v == nil {
        return ""
    }

    // if func (v *Type) String() string, we can't use Elem()
    switch vt := v.(type) {
    case fmt.Stringer:
        return vt.String()
    }

    val := reflect.ValueOf(v)
    if val.Kind() == reflect.Ptr && !val.IsNil() {
        val = val.Elem()
    }

    return reprOfValue(val)
}
```

Adding a node

The final call is to specify the virtual node add node method

```
// expand operation to add physical nodes
func (h *ConsistentHash) Add(node interface{}) {
    h.AddWithReplicas(node, h.replicas)
}
```

Adding a node — specifying weights

The final call is also to specify the virtual node add node method

```
// Add nodes by weight
// Calculate the method factor by weight, which ultimately
controls the number of virtual nodes
// The higher the weight, the higher the number of virtual nodes
func (h *ConsistentHash) AddWithWeight(node interface{}, weight
int) {
    replicas := h.replicas * weight / TopWeight
    h.AddWithReplicas(node, replicas)
}
```

Add nodes — specify the number of virtual nodes

```
// Scaling operation to add physical nodes
func (h *ConsistentHash) AddWithReplicas(node interface{},
replicas int) {
    // Support repeatable additions
    // Perform the delete operation first
    h.Remove(node)
    // cannot exceed the upper limit of the magnification factor
```

```go
    if replicas > h.replicas {
        replicas = h.replicas
    }
    // node key
    nodeRepr := repr(node)
    h.lock.Lock()
    defer h.lock.Unlock()
    // Add node map mapping
    h.addNode(nodeRepr)
    for i := 0; i < replicas; i++ {
        // Create virtual node
        hash := h.hashFunc([]byte(nodeRepr + strconv.Itoa(i)))
        // add virtual nodes
        h.keys = append(h.keys, hash)
        // map virtual nodes - real nodes
        // Note that hashFunc may have hash conflicts, so append
operation is used
        // The virtual node-real node mapping is actually an
array
        // A virtual node may correspond to more than one real
node, but the probability is very small
        h.ring[hash] = append(h.ring[hash], node)
    }
    // Sorting
    // Later, we'll use a dichotomous lookup of the virtual nodes
    sort.Slice(h.keys, func(i, j int) bool {
        return h.keys[i] < h.keys[j]
    })
}
```

Delete nodes

```go
// Remove the physical node
func (h *ConsistentHash) Remove(node interface{}) {
    // string of the node
    nodeRepr := repr(node)
    // concurrently safe
    h.lock.Lock()
    defer h.lock.Unlock()
    // Node does not exist
    if !h.containsNode(nodeRepr) {
        return
    }
    // Remove the virtual node mapping
    for i := 0; i < h.replicas; i++ {
        // Calculate the hash value
        hash := h.hashFunc([]byte(nodeRepr + strconv.Itoa(i)))
        // Dichotomous search to the first virtual node
        index := sort.Search(len(h.keys), func(i int) bool {
            return h.keys[i] >= hash
        })
        // Slice and delete the corresponding element
        if index < len(h.keys) && h.keys[index] == hash {
            // locate the element before the index of the slice
            // move the element after index (index+1) forward to
cover index
            h.keys = append(h.keys[:index], h.keys[index+1:]...)
        }
        // Virtual node removal mapping
        h.removeRingNode(hash, nodeRepr)
    }
    // Remove the real node
    h.removeNode(nodeRepr)
}

// Remove the virtual-real node mapping relationship
// hash - the virtual node
// nodeRepr - the real node
func (h *ConsistentHash) removeRingNode(hash uint64, nodeRepr
string) {
    // map should be checked when used
    if nodes, ok := h.ring[hash]; ok {
        // Create a new empty slice, with the same capacity as
nodes
        newNodes := nodes[:0]
        // Iterate over nodes
        for _, x := range nodes {
            // if serialized values are not the same, x is
another node
            // cannot be deleted
            if repr(x) ! = nodeRepr {
                newNodes = append(newNodes, x)
            }
        }
        // rebind the mapping relationship if the remaining nodes
are not empty
        if len(newNodes) > 0 {
            h.ring[hash] = newNodes
        } else {
            // otherwise just delete
            delete(h.ring, hash)
        }
    }
}
```

Query node

```go
// find the nearest virtual node based on v clockwise
// then find the real node by virtual node mapping
func (h *ConsistentHash) Get(v interface{}) (interface{}, bool) {
    h.lock.RLock()
    defer h.lock.RUnlock()
    // No physical node currently
    if len(h.ring) == 0 {
        return nil, false
    }
    // Calculate the hash value
    hash := h.hashFunc([]byte(repr(v)))
    // Dichotomous lookup
    // because the virtual nodes are reordered each time a node
is added
    // so the first node queried is our target node
    // remainder will give us a circular list effect, finding
nodes clockwise
    index := sort.Search(len(h.keys), func(i int) bool {
        return h.keys[i] >= hash
    }) % len(h.keys)
    // virtual nodes -> physical nodes mapping
    nodes := h.ring[h.keys[index]]
    switch len(nodes) {
    // No real nodes exist
    case 0:
        return nil, false
    // only one real node, return nil, false
    case 1:
        return nodes[0], true
    // The presence of multiple real nodes means that there is a
hash conflict
    default:
        // At this point we re-hash v
        // We get a new index by taking the remainder of the
nodes length
        innerIndex := h.hashFunc([]byte(innerRepr(v)))
        pos := int(innerIndex % uint64(len(nodes)))
        return nodes[pos], true
    }
}
```

GitHub Project

https://github.com/zeromicro/go-zero

Welcome to use `go-zero` and **star** to support us!

Join FAUN: Website 💻|Podcast 🎙️|Twitter 🐦|Facebook 👥|Instagram 📷|Facebook Group 🗣️|Linkedin Group 💬| Slack 📱 |Cloud Native News 📰|More.

If this post was helpful, please click the clap 👏 button below a few times to show your support for the author 👇

Go Golang Microservices Hash Architecture

Written by Kevin Wan

344 Followers · Writer for FAUN—Developer Community 🐾

github.com/zeromicro/go-zero

Follow

More from Kevin Wan and FAUN—Developer Community 🐾

Kevin Wan in CodeX

Understanding and implementing consistent hash...

In go-zero's distributed caching implementation, we used consistent hash...

4 min read · Sep 21, 2021

173

Open Data Ana... in FAUN—Developer Commu...

18 Sites to Host Your Backend Code for Free

In today's world, developers have a wide range of choices when it comes to backen...

6 min read · Apr 11

771 10

Ioannis Moust... in FAUN—Developer Commun...

AWS Solutions Architect Professional Cheat Sheet

This material was gathered while preparing for the AWS Solutions Architect...

· 59 min read · Jun 12

118

Kevin Wan in Dev Genius

Implementing Go stream API

What is Stream Processing

15 min read · Jan 3, 2022

52 3

See all from Kevin Wan

See all from FAUN—Developer Community

Recommended from Medium

Isuru Harischandra in Towards Dev

Interfaces in Go: A Guide to Implementing and Using...

Unlock the Power of Interfaces in Go: A Complete Guide to Implementing and...

6 min read · May 26

16

Nyoman Frastyawan in Stackademic

Secure File Transfer Made Easy: Connect to SFTP Servers with...

Intro

3 min read · Aug 28

Lists

General Coding Knowledge

20 stories · 283 saves

Stories to Help You Grow as a Software Developer

19 stories · 330 saves

Now in AI: Handpicked by Better Programming

266 stories · 120 saves

New_Reading_List

174 stories · 93 saves

Ethiraj Srinivasan in Bootcamp

The world of Rate Limit Algorithms

Rate limit algorithm is a mechanism used to control the rate of requests or messages…

✦ · 14 min read · Mar 9

141

Mike Norgate

Unlocking Go Slice Performance: Navigating sync.Pool for…

When dealing with large slices, a common suggestion is to utilize sync.Pool in order t…

5 min read · 6 days ago

18

Rustem Kamalov in ITNEXT

Best regexp alternative for Go

Benchmarks and Plots

9 min read · 5 days ago

24

Dmitry Kruglov in Better Programming

The Architecture of a Modern Startup

Hype wave, pragmatic evidence vs the need to move fast

16 min read · Nov 8, 2022

5.4K 48

See more recommendations