代码随想录项目精讲-Go项目

代码随想录知识星球 项目精讲系列已经发布了：

- 代码随想录项目精讲-论坛项目
- 代码随想录项目精讲-前端项目
- 代码随想录项目精讲-webserver

这次我们正式发布Go项目精讲！

关于go的学习路线以及go的学习资料可以看星球这个帖子。

本篇项目文档，将包含两个Go项目，分别是： 仿极客兔兔分布式缓存、 kv存储 ， 目录如下：

        - 什么是一致性哈希？为什么在项目中要使用它?
        - 如何保证一致性哈希算法的有效性?
        - 一致性哈希算法中的虚拟节点是什么？它们的作用是什么?
        - 虚拟节点怎么实现的？查找目标 key 的过程是怎样的?
    - etcd 相关问题 (如果用到了 etcd 一定要准备相关问题)
        - 为什么要使用 etcd，怎么用的?
        - Raft 算法
        - etcd 是如何保证强一致性的呢
        - etcd分布式锁实现的基础机制是怎样的
        - 能说一说用 etcd 时它处理请求的流程是怎样的吗
- kv存储
    - 前言
    - 所需要的基础知识
    - 相关参考资料
    - 项目的简要介绍
    - 项目模块和结构组织分析
        - mindb
        - storage
        - index
        - datastruct
        - cmd
        - bench
    - 项目面试相关问题
        - 这个项目有什么优点?
        - 介绍一下bitcask模型
        - 了解LSM树么
        - 内存映射（mmap）是什么?
        - 做项目的过程中有没有什么自己的想法?
        - merge操作如何进行？还可以继续优化么?
        - 服务端客户端命令行如何实现的?
        - 项目中用到了跳表，为什么用，怎么实现的?
        - 跳表重点问题（从选择跳表到跳表实现中的细节，如果自己写跳表一定要搞懂）：
            - 刚刚说到了`redis`，为什么`redis`的`zset`用跳表实现而不是红黑树?
            - 跳表索引的动态更新是怎样做的?
            - 跳表的高度控制策略是怎样的?
            - 既然跳表的平衡是随机算法控制的，那如何保证`O(logn)`的复杂度?

仿极客兔兔分布式缓存

前言

该项目来自于极客兔兔七天实现系列中的GeeCache，是一个简单练手项目。
项目原地址：https://geektutu.com/post/geecache.html
在其基础上，可以做一些拓展优化，如：

1. 将单独`lru`算法实现改成多种算法可选（容易实现且lru, lfu都是经典算法，方便顺便考察算法能力）
2. 将`http`通信改为`rpc`通信提高网络通信效率（方便引出rpc相关问题）
3. 细化锁的粒度来提高并发性能
4. 实现热点互备来避免`hot key`频繁请求网络影响性能（groupcache中提出的优化）
5. 加入`etcd`进行分布式节点的监测实现节点的动态管理（能拓展出大量分布式问题，etcd作为常见组件在面试中也很能聊）
6. 加入缓存过期机制，自动清理超时缓存

相关的参考项目

对于一个陌生的项目，首先要将它拉到本地跑起来，现在很多项目都提供了`docker`镜像，也免去了搭建环境的烦恼。然后要去看它的主要功能是什么，针对主要功能的部分去看外层的接口，每次只关注一个小的功能，一层一层往里扒它的具体实现，看几个功能后就会大概清楚项目的目录结构了。对于一些感到困惑的地方，可以去翻`Issue`和`PR`，很可能已经有人有过相同的问题了。

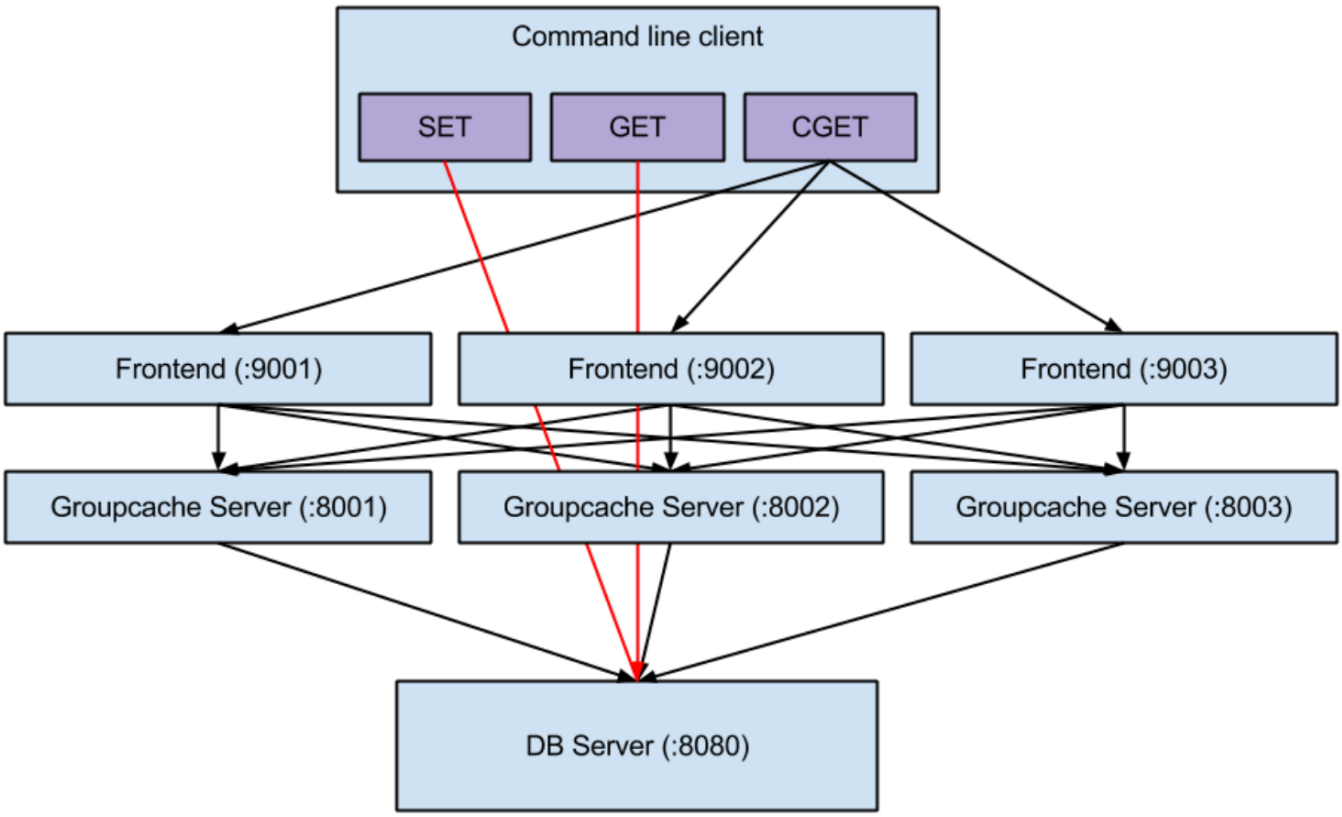
1. peanutcache：使用`grpc`进行节点间通信，用`etcd`实现服务注册和服务发现 https://github.com/peanutzhen/peanutcache
2. maingun-groupcache：提供了数据更新删除的支持和TTL机制 https://github.com/mailgun/groupcache
3. gcache：支持`lru`，`lfu`，`arc`缓存机制同时并发安全 https://github.com/bluele/gcache
4. ccache：针对高并发进行优化 https://github.com/karlseguin/ccache

相关参考资料

1. 一个有趣的分布式缓存实现 — groupcache
2. 从入门到掉坑：Go 内存池/对象池技术介绍
3. groupcache官方文档

项目介绍

这个项目是基于`groupcache`实现的一个分布式缓存，在`groupcache`上做了一些拓展。
项目架构图如下：



主要分为了 Group 模块，缓存模块，分布式一致性d等模块：

Group模块

Group模块是对外提供服务接口的部分，一个Group就是一个缓存空间。其要实现对缓存的增删查方法。

```
// Group 提供命名管理缓存/填充缓存的能力
type Group struct {
  name              string                // 缓存空间的名字
  getter            Getter                // 数据源获取数据
  mainCache         *cache                // 主缓存，并发缓存
  hotCache          *cache                // 热点缓存
  server            Picker                // 用于获取远程节点请求客户端
  flight            *singleflight.Flight // 避免对同一个key多次加载造成缓存击穿
  emptyKeyDuration time.Duration         // getter返回error时对应空值key的过期时间
}
```

缓存模块

- 缓存模块是数据实际存储的位置，其中实现缓存淘汰算法，过期机制，回调机制等，缓存模块与其他部分是解耦的，因此可以根据不同场景选择不同的缓存淘汰算法（默认lru）。
  groupcache本身的实现中，缓存值只淘汰不更新，也没有超时淘汰机制，通过这样来简化设计，并没有指定缓存的移除操作。
  在本项目的拓展中加了指定键的移除操作，在lru中进行了实现。

增加了键的移除操作后，需要考虑的一个问题是：移除了某个键，每个节点中的热点缓存中如果有该键，也需要删除，因此删除操作需要通知所有节点（etcd实现）。
另外对于缓存来说，超时淘汰有时候也是必要的，因此在缓存中对于每个value，都设计了过期时间。

```
// Cache LRU缓存
type Cache struct {
  capacity  int // 缓存容量
  length    int // 当前缓存大小
  ll        *list.List
  cache     map[string]*list.Element
  onEvicted func(key string, value Value) // 可选，在entry被移除的时候执行
  // .... 根据需求继续拓展
}
```

byteview 模块

byteview是对实际缓存的一层封装，因为实际的缓存值是一个byte切片存储的，而切片的底层是一个指向底层数组的指针，一个记录长度的变量和一个记录容量的变量。
如果获取缓存值时直接返回缓存值的切片，那个切片只是原切片三个变量的拷贝，真正的缓存值就可能被外部恶意修改。
所以用byteView进行一层封装，返回缓存值时的byteView则是一个原切片的深拷贝。

```
type ByteView struct {
  b      []byte
  expire time.Time // 过期时间
}
```

分布式一致性模块

- 分布式一致性模块实现了一致性哈希算法，将机器节点组成哈希环，为每个节点提供了从其他节点获取缓存的能力

```
type Consistence struct {
  hash     Hash           // 哈希函数依赖
  replicas int            // 虚拟节点倍数
  ring     []int          // 哈希环
  hashMap  map[int]string // 虚拟节点hash到真实节点名称的映射
}
```

读流程：

1. 当 Group 模块接收到请求时，当前节点先到本地的主缓存中查找是否有目标值。
2. 如果没有再查看此节点的热点缓存中是否有，如果也没有，就需要到远程伙伴节点去获取
3. 通过一致性哈希计算该`key`的哈希值，找到哈希环上相应的节点，执行该节点的远程调用函数获取值，并把该值加入到当前节点的热点缓存中
4. 如果该key在哈希环中对应的就是当前节点，那么就需要从本地的数据源去获取数据后加载到当前节点的主缓存中

本项目作为`lib`与`app`运行在同一进程中，其既是`server`又是`client`，作为`server`，它与其他`cache`一起组成`hash`环。对于调用者来说，从远程节点请求缓存的过程是透明的，用户像使用本地缓存一样使用`cache`

项目面试相关问题

缓存相关

去设计一个分布式缓存系统要从哪些方面考虑?

1. 首先得考虑本地缓存的问题：
    1. 数据的存储方式：该用什么数据结构存储，是简单的哈希表还是像`redis`一样多种数据类型
    2. 缓存策略：缓存达到上限后如何淘汰，选择`LRU`还是`LFU`等
    3. 过期时间：是否要设置过期时间，过期时间是否要搭配缓存淘汰策略一起使用
    4. 缓存失效：缓存失效或者没有命中该如何处理
    5. 并发访问：并发访问时如何保证系统安全和稳定
2. 其次考虑分布式部署后的问题：
    1. 负载均衡：缓存数据如何均衡地分布在不同节点，系统该如何均衡地处理请求负载
    2. 节点通信：节点之间如何进行通信
    3. 缓存一致性：如果数据在不同节点上均存在，当一个节点的数据发生变化时如何通知其他节点使得它们进行更新
    4. 系统可用性：分布式环境下，会有某个节点发生故障或网络中断的问题，该如何设计容错机制和故障恢复策略

缓存雪崩，击穿，穿透分别是什么，如何应对?

- 缓存雪崩：如果某一时刻有大量缓存过期或者缓存系统发生故障宕机，此时又有大量的用户请求，那么这些请求就会直接访问数据库，从而导致数据库的压力剧增，严重的就会使数据库宕机，导致一系列的连锁反应，这样的情况叫做缓存雪崩。
- 缓存击穿：如果缓存中的某个热点数据过期了，此时大量的请求就会穿过缓存访问到数据库上，数据库很容易就会被高并发的请求击垮，这就是缓存击穿问题

缓存击穿问题可以看作是缓存雪崩的一个子集

- 缓存穿透：缓存穿透是请求数据库中并不存在的数据，因此当这种数据访问时，会先访问缓存而后再访问数据库，但是数据库中也没有，因此并不会更新缓存，后续这类请求再来还是会访问数据库，因此当这类请求大量出现时，数据库就会面临高并发压力。

一般应对措施有：

1. 使用cache集群，保证缓存的高可用（比如`redis`的主从+`sentinel`或`cluster`）
2. 分级多层缓存，比如对于热点数据，使用带过期机制的本地缓存代替一部分缓存系统功能
3. 合理的熔断及限流保护机制，比如`hystrix`熔断，`google`的自适应熔断等，可以非常有效地避免缓存雪崩
4. 合理使用`singleflight`机制

你的项目中如何应对缓存雪崩和缓存击穿问题?

项目实现了`sigleFlight`机制。

- `singleFlight`的原理是：在多个并发请求触发的回调操作里，只有第一个回调方法被执行，其余请求（落在第一个回调方法执行的时间窗口里）阻塞等待第一个回调函数执行完成后直接取结果，以此保证同一时刻只有一个回调方法执行，达到防止缓存击穿的目的

- `singleFlight`常用场景：
  1. 缓存失效时的保护性更新
  2. 防止突增的接口请求对后端服务造成瞬时高负载
- 具体实现：
  利用`mutex`互斥锁和`sync.WaitGroup`机制来实现多`goroutine`的并发控制策略：

1. 用一个类packet表示正在进行或者已经结束的请求：

```
  type packet struct  // 代表正在进行中或者已经结束的请求
  wg  sync.WaitGroup   // 避免重入
  val interface{}
  err error
}
```

2. 用一个map来管理不同key对应的请求，因为该操作并发访问需要加锁所以封装成类

```
type Flight struct   // 管理不同key的请求
  mu     sync.Mutex
  flight map[string]*packet
}
```

3. 一个Fly方法，接收参数为 key 和请求函数，实现：对于这个key，只能同时有一个请求函数在执行

```
// Fly 负责key从数据源进行获取 fn是获取packet的方法
func (f *Flight) Fly(key string, fn func() (interface{}, error)) (interface{}, error) {
  f.mu.Lock()
  if f.flight == nil {
    f.flight = make(map[string]*packet)
  }
  if p, ok := f.flight[key]; ok {  // 如果该key请求正在进行，则等待
    f.mu.Unlock()
    p.wg.Wait() // 等待协程结束
    return p.val, p.err  // 返回目标值
  }
  p := new(packet)
  p.wg.Add(1)   // 发起请求前加锁
  f.flight[key] = p  // 表明该key已经有请求在进行
  f.mu.Unlock()

  p.val, p.err = fn()  // 执行请求
  p.wg.Done()

  f.mu.Lock()
  delete(f.flight, key) // 完成请求 更新Flight
  f.mu.Unlock()

  return p.val, p.err
}
```

调用Fly方法后首先检查map中该key对应的请求是否存在，如果存在就通过waitGroup的wait方法进行阻塞等待，等待完成后直接取走最新值.
如果不存在就调用请求函数，同时要用Add方法通知其他协程，执行完成后更新map中的值，然后进行Done通知其他协程

了解的缓存淘汰策略有哪些?

1. FIFO：先进先出就是每次淘汰最早添加的记录，但是很多记录添加地早也访问的很频繁，因此命中率不高
2. LFU：淘汰缓存中使用频率最低的，LFU认为如果数据过去被访问多次，那么将来被访问的频率也会更高。LFU的实现需要维护一个按照访问次数排序的队列，每次访问后元素的访问次数改变，队列重新排序。

LFU算法的命中率比较高，但缺点是维护每个记录的访问次数对内存的消耗比较高，并且如果数据的访问模式发生改变，LFU需要较长的时间去适应，也就是它受历史访问的影响比较大。在保证高频数据有效性场景下，可选择这类策略

3. LRU：LRU是比较折中的淘汰算法，LRU认为如果数据在最近被访问过，那么将来被访问到的概率也比较高。LRU算法的实现相较于LFU简单很多,只需要维护一个队列,访问到的数据移到队首,每次淘汰队尾即可。

在热点数据场景下较适用，优先保证热点数据的有效性。

4. ARC：ARC 介于 LRU 和 LFU 之间，借助 LRU 和 LFU 基本思想实现，以获得可用缓存的最佳使用

一致性哈希相关问题

什么是一致性哈希？为什么在项目中要使用它?

一致性哈希的主要目的是解决缓存系统的水平扩展性和负载均衡问题。
在传统的缓存系统中，数据的存储位置通常通过将键映射到固定的缓存节点来确定。然而，这种方式在节点的增加或减少时会导致大量的数据迁移，影响性能和可用性。
在本项目中，一致性哈希用于确定缓存数据的分片和分布。每个节点都负责一定范围的哈希键，当需要查询或存储数据时，通过一致性哈希可以快速定位到对应的节点。这种方式有效地避免了数据倾斜和热点问题，并提供了良好的负载均衡和扩展性。

如何保证一致性哈希算法的有效性?

一般来说哈希函数考虑两点：一个是碰撞率，一个是性能。常见的哈希函数有`crc`，`md5`，`sha1`
对于缓存来说，`hash`之后还要再根据节点数量取模，因此哈希函数的碰撞率影响并不大，而是模的大小，即节点数量比较关键，因此引入足够的虚拟节点可以保证哈希算法的命中率。

一致性哈希算法中的虚拟节点是什么？它们的作用是什么?

虚拟节点的作用是在哈希环上增加节点的数量。
通过为每个物理节点创建多个虚拟节点，并将这些虚拟节点均匀地分布在哈希环上，可以更好地平衡数据的负载分布，尤其在节点数量较少的情况下。当数据量不断增长或节点数量不足以满足负载需求时，虚拟节点可以提供更多的灵活性，使得数据能够更加均匀地分布到不同的节点上。
另外其提高系统的故障容忍性。由于虚拟节点将数据分布在环上的不同位置，当物理节点发生故障或不可用时，只需要迁移该物理节点对应的虚拟节点上的数据，而不需要迁移所有数据。这样可以减少数据迁移的成本和影响范围，提高系统的可用性。

虚拟节点怎么实现的？查找目标 key 的过程是怎样的?

1. 用一个`map`来维护每个虚拟节点（虚拟节点哈希值作为key）和它对应的真实节点（真实节点名称作为val）的映射。
2. 查找的过程为：首先计算出key的哈希值，通过二分查找找到哈希环数组中第一个比它大的位置（即环的顺时针）。因为如果位置等于哈希环数组的长度的话应该对应`0`节点，所以将`idx`对哈希环数组长度进行取模。最后返回`map`映射中该节点对应的真实节点即可。

```
// GetPeer 计算key应缓存到的peer
func (c *Consistence) GetPeer(key string) string {
  if len(c.ring) == 0 {
    return ""
  }
  hashValue := int(c.hash([]byte(key)))
  idx := sort.Search(len(c.ring), func(i int) bool { // 二分查找第一个比它哈希值大的节点
    return c.ring[i] >= hashValue
  })
  return c.hashMap[c.ring[idx%len(c.ring)]]
}
```

etcd 相关问题 (如果用到了 etcd 一定要准备相关问题)

为什么要使用 etcd，怎么用的?

在原来的项目中，是无法实现节点的动态添加和移除的，其假定所有的节点都是可用的来简化设计。
而 etcd 是一个高可用强一致性的键值仓库，在很多分布式系统架构中得到了广泛的应用，其最经典的使用场景就是服务发现。服务发现要解决的是分布式系统中最常见的问题之一，即在同一个分布式集群中的进程或服务，要如何才能找到对方并建立连接。本质上来说，服务发现就是想要了解集群中是否有进程在监听`udp`或`tcp`端口，并且通过名字就可以查找和连接。
为了实现节点的动态管理，可加入 etcd 进行分布式节点的监测。

- 引入 etcd 需要实现注册方法和发现方法：

1. 注册方法的实现：先用默认配置通过`New`方法创建一个`etcd`客户端; 用`Grant`方法创建一个租约，默认五秒时间; 把服务节点连同租约一起加到`etcd`中; 设置租约检测，用`keepAlive`得到一个`chan`；最后用`for select`进行`channel`监听：监听停止信号，`context`的`Done`和租约`chan`

```
// Register 注册一个服务至etcd
// 注意 Register将不会return 如果没有error的话
func Register(service string, addr string, stop chan error) error {
  // 创建一个etcd client
  cli, err := clientv3.New(defaultEtcdConfig)
  if err != nil {
    return fmt.Errorf("create etcd client failed: %v", err)
  }
  defer cli.Close()
  // 创建一个租约 配置5秒过期
  resp, err := cli.Grant(context.Background(), 5)
  if err != nil {
```

```
    return fmt.Errorf("create lease failed: %v", err)
  }
  leaseId := resp.ID
  // 注册服务
  err = etcdAdd(cli, leaseId, service, addr)
  if err != nil {
    return fmt.Errorf("add etcd record failed: %v", err)
  }
  // 设置服务租约检测
  ch, err := cli.KeepAlive(context.Background(), leaseId)
  if err != nil {
    return fmt.Errorf("set keepalive failed: %v", err)
  }

  log.Printf("[%s] register service ok\n", addr)
  for {
    select {
    case err := <-stop:
      if err != nil {
        log.Println(err)
      }
      return err
    case <-cli.Ctx().Done():
      log.Println("service closed")
      return nil
    case _, ok := <-ch:
      // 监听租约
      if !ok {
        log.Println("keep alive channel closed")
        _, err := cli.Revoke(context.Background(), leaseId)
        return err
      }
    }
  }
}
```

2. 发现方法的实现：服务端通过etcd的客户端实现的Watch方法监听事件

```
// EtcdDial 向grpc请求一个服务
// 通过提供一个etcd client和service name即可获得Connection
func EtcdDial(c *clientv3.Client, service string) (*grpc.ClientConn, error) {
  etcdResolver, err := resolver.NewBuilder(c)
  if err != nil {
    return nil, err
  }
  return grpc.Dial(
    "etcd:///"+service,
    grpc.WithResolvers(etcdResolver),
    grpc.WithInsecure(),
```

```
    grpc.WithBlock(),
  )
}
```

Raft 算法

1. Raft中一个`Term`是什么意思
   一个`term`指从一次竞选出`leader`到下一次竞选的时间。
   如果`follower`接收不到`leader`的心跳信息，就会结束当前`term`，变为`candidate`继而发起竞选，帮助`leader`故障时集群的恢复。
   如果集群不出现故障，那么一个`term`将无限延续下去。
   投票时候若出现冲突也会直接进入下一个任期重新竞选。
2. Raft状态机是怎样进行切换的
   Raft刚开始运行时，`Noder`默认进入`Follower`状态，等待`Leader`发来心跳信息。
   若等待超时就由`Follower`转换到`Candidate`状态进入下一轮`Term`进行`leader`选举。
   收到集群中多数节点的投票时，该节点就转变为`Leader`状态。
   `Leader`可能因为网络通信的故障，导致别的`Node`发起投票成为新`Term`的`Leader`，此时原来的`Leader`会自动切换成`Follower`。
   `Candidate`如果在等待投票过程中发现已经有竞选成功的`leader`，也会切换为`Follower`。
3. 如何保证最短时间内竞选出Leader，防止竞选冲突
   竞选冲突主要是因为多个节点同时发起选举，导致每个节点的票数都不足以成为`leader`。
   所以在`Candidate`状态下，有一个`times out`值，其是个随机值，在每个`Node`成为`Candidate`后，发起选举的时间受这个随机值的影响，这就造成了一个时间差，使得发起选举的时间能够错开。
   在时间差内，如果`Candidate`节点收到的竞选信息比自己发起的竞选信息的`Term`更大，并且在新一轮想要成为`Leader`的另一个`Candidate`包含了所有的提交数据，那么当前`Candidate`就会投票给另一个`Candidate`
   这样两个措施保证了只有很小的概率会出现竞选冲突.
4. 如何防止别的Candidate在遗漏部分数据的情况下发起投票成为Leader
   Raft选举机制中，使用随机值决定`times out`，第一个超时的`Node`就会增加`Term`发起新一轮投票，一般情况下别的`Node`收到竞选通知就会投票。
   但是如果发现竞选的`Node`在上一个`Term`中保存的已提交数据不完整，`Node`就会拒绝投票给它。
   通过这种机制就可以防止遗漏数据的`Node`成为`Leader`。
5. Raft有节点宕机后会怎样

- 宕机的是`Follower`：如果剩余可用节点数据超过一半，集群可以几乎没有影响地正常工作
- 宕机的是`Leader`：`Follower`就会收不到心跳信息而超时，从而发起新一轮`Term`的选举，然后选出新的`Leader`，继续为集群提供服务

注意：etcd目前没有自动调整整个集群总结点数量的机制，即如果没有人为调整，`etcd`中宕机后的`Node`仍会被计算到总节点数量中，投票时仍以这个总量的一半为参考。

6. 为什么Raft算法在确定可用节点数量时不需要考虑拜占庭将军问题
   因为Raft算法中假设所有的Node都是诚实的。
   Raft算法中`Node`在竞选时要告诉别的`Node`自身的`Term`编号以及前一轮`Term`最终结束时的`Index`值，这些数据都是准确的，不存在欺骗的可能，其他`Node`根据这些值决定是否投票。
   另外，Raft限制只有`Leader`到`Follower`这样的数据流向保证数据一致不会出错。

etcd 是如何保证强一致性的呢

etcd 使用 Raft 协议来保证强一致性，包括两个过程：leader选举和主从数据同步。

- leader选举：
  1. 集群初始化时，每个节点都是`follower`，都维护一个的计时器，如果计时器时间到了还没有收到`leader`的消息，自己就会变成`candidate`，竞选`leader`
  2. 当`follower`一定时间内没有收到来自主节点的心跳，则认为主节点宕机，会将自己变为`candidate`，并发起选举，当收到包括自己在内超过半数的节点投票后，选举成功；当票数不足或者选举超时则选举失败，若本轮未选出主节点则马上进行新一轮选举
  3. 集群中存在至多一个主节点，通过心跳机制与其他节点同步数据
  4. `candidate`节点收到来自主节点的信息后，会立即终止选举进入`follower`角色，为了避免陷入选主失败循环，每个节点未收到心跳后发起选举的时间是一定范围内的随机值，这样来避免两个节点同时发起选举
- 主从数据同步：
  1. `client`连接`follower`或者`leader`，如果是读请求，`follower`也可以处理，如果是写请求，则连接到`follower`后还需要转发给`leader`处理
  2. `leader`接收到`client`的写请求后，将该请求转为`entries`（批量的entry），写入到自己的日志中，得到在日志中的`index`，将该`entries`发送给所有的`follower`
  3. `follower`接收到`leader`的`AppendEntriesRPC`请求后，会将`leader`传过来的`entries`写入到文件中（并不立即刷盘），然后向`leader`回复确认，`leader`收到过半的确认后就认为可以提交了，会应用到自己的状态机中，然后更新`commitIndex`，回复`client`
  4. 在下一次`leader`发送给`follower`的心跳中，会将`leader`的`commitIndex`发送给`follower`，`follower`发现`commitIndex`更新了则也将`commitIndex`之前的日志都进行提交和应用到状态机中
- 在`leader`收到数据操作的请求时，先不着急更新本地缓存（数据是持久化在磁盘上的），而是生成对应的`log`，然后把生成`log`的请求广播给所有的`follower`，每个`follower`在收到请求之后听从`leader`的命令，也写入`log`，返回确认。`leader`收到过半的确认之后进行二次提交，正式写入数据（持久化），然后告诉`follower`，让他们也持久化

etcd分布式锁实现的基础机制是怎样的

- `Lease`机制
  租约机制，etcd 可以为存储的`key-value`对设置租约，当租约到期，`key-value`将失效删除
  同时也支持续约，通过客户端可以在租约到期之前续约
  `Lease`机制可以保证分布式锁的安全性，为锁对应的`key`配置租约，即使锁的持有者因故障而不能主动释放锁，锁也会因为租约到期而自动释放
- `Revision`机制
  每个key都带有一个`Revision`号，其是全局唯一的，通过`Revision`号的大小可以知道写操作的顺序
  在实现分布式锁时，多个客户端同时抢锁，根据`Revision`号大小依次获得锁，可以避免羊群效应
- `Prefix`机制
  例如，一个名为`/etcd/lock`的锁，两个争抢它的客户端进行写操作， 实际写入的`key`分别为：`key1="/etcd/lock/UUID1"`，`key2="/etcd/lock/UUID2"`。
  其中，`UUID`表示全局唯一的`ID`，确保两个`key`的唯一性。

写操作都会成功，但返回的 `Revision` 不一样， 那么，如何判断谁获得了锁呢？通过前缀 `/etcd/lock` 查询，返回包含两个 `key-value` 对的的 `KeyValue` 列表， 同时也包含它们的 `Revision` ，通过`Revision` 大小，客户端可以判断自己是否获得锁。

能说一说用 etcd 时它处理请求的流程是怎样的吗

- etcd主要分为四部分：
    1. http server：用于处理用户发送的`api` 请求以及其他 etcd 节点的同步与心跳信息请求
    2. store：用于处理 etcd 支持的各类功能的事务，包括数据索引，节点状态变更，监控反馈，事件处理执行等，是 etcd 对用户提供的大多数`api` 功能的具体实现
    3. raft：强一致性算法的实现
    4. wal：预写日志，除了在内存中存有所有数据的状态以及节点的索引外，etcd 就通过 `wal` 进行持久化存储；其中`snapshot` 是为了防止数据过多而进行的状态快照， `entry` 表示存储的具体日志内容
- 通常一个用户请求发过来会经由`http server` 转发给`store` 进行具体的事务处理，如果涉及到节点的修改，则交给`raft` 模块进行状态变更，日志记录，然后再同步给别的 etcd 节点以确认数据提交，最后进行数据的提交，再次同步

kv存储

前言

本项目来自于开源项目rosedb，是根据博主roseduan大佬早期在b站录制的视频思路思路来的，之后roseduan对项目做了多次重构，但主体思路是不变的，可以在本项目的基础上去看rosedb的源码，会清晰很多
项目源地址：https://github.com/roseduan/rosedb

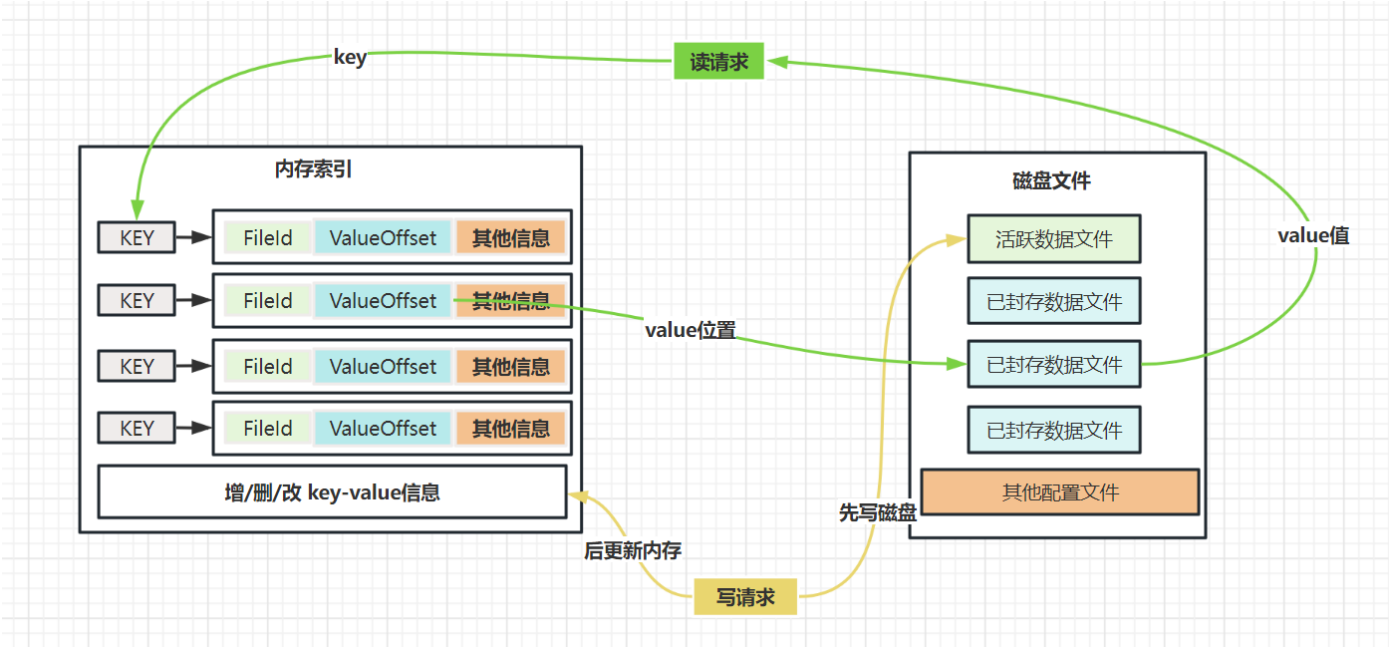
所需要的基础知识

熟悉go语言的基础语法，数据结构，对操作系统和数据库有一定了解

相关参考资料

1. 项目owner录制的系列讲解视频：使用 Go 写一个数据库-1基本结构
2. 项目owner在go夜读的视频分享：详解 rosedb 及存储模型【Go 夜读】
3. bitcask分析视频：bitcask模型上半部分：bitcask实现原理及源码剖析
4. rosedb分析视频：bitcask模型下半部分：类bitcask(rosedb、nutsdb)实现原理及源码剖析

项目的简要介绍



- 基本功能：读取数据和写入数据；支持多种数据结构；支持过期时间；支持服务端和客户端；支持磁盘空间回收；支持单独存磁盘和内存磁盘双存两种模式
- 设计理念：基于bitcask模型，日志形式存储，磁盘中的日志文件本身作为数据库，内存中存储日志文件索引。相比于redis这种内存数据库，稍微损失了一点性能而获得了数倍于内存的存储空间
- 写操作：增删改全部为在磁盘上新增一条记录，即全部为顺序写，不需要磁盘寻址，每个记录有固定的格式。磁盘写完之后更新内存中的相关索引
- 读操作：在内存中通过 key 定位到目标数据的相关记录，从该记录中读取到实际数据在磁盘中的具体位置

项目模块和结构组织分析

mindb

该文件为项目的入口，相当于`main`文件。在其中进行了**错误类型**，**配置文件**，**数据库结构**等的类型定义。

- 数据库结构为：

- `main`文件主要方法为：
  1. `Open`：根据既定配置，打开一个数据库实例
  2. `Reopen`：重启数据库实例
  3. `Close`：关闭数据库实例，并保存相关配置
  4. `Sync`：将数据持久化到磁盘
  5. `Reclaim`：重新组织磁盘中的数据，回收磁盘空间
  6. `Backup`：复制当前数据库实例的目录，用于备份

storage

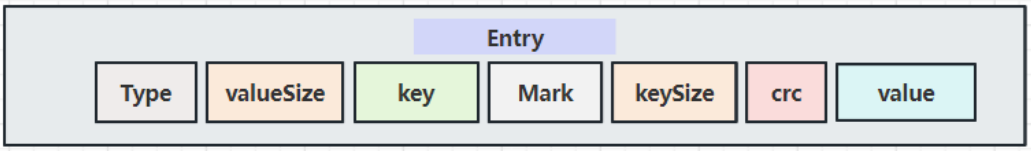
该目录下存放数据库的数据结构文件，包含类型定义和相关方法。

1. `entry`

`entry`是向数据库写操作时的数据组织格式，一条写记录即为一个`entry`。该文件中定义了对`entry`的创建和编解码方法等

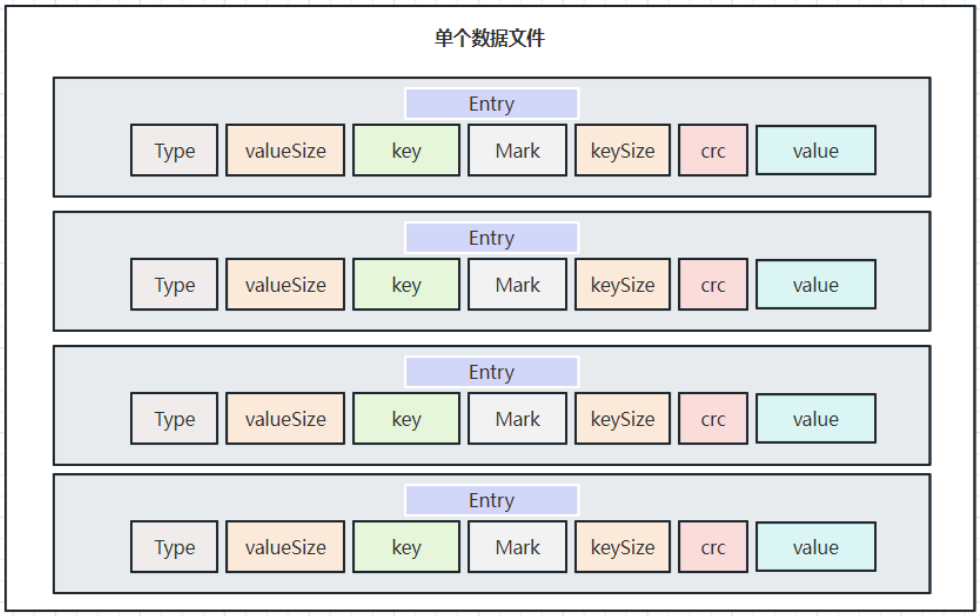
```
type (
  // Entry 数据entry定义
  Entry struct {
    Meta   *Meta
    Type   uint16 //数据类型
    Mark   uint16 //数据操作类型
    crc32 uint32 //校验和
  }

  // Meta meta 数据
  Meta struct {
    Key        []byte
    Value      []byte
```

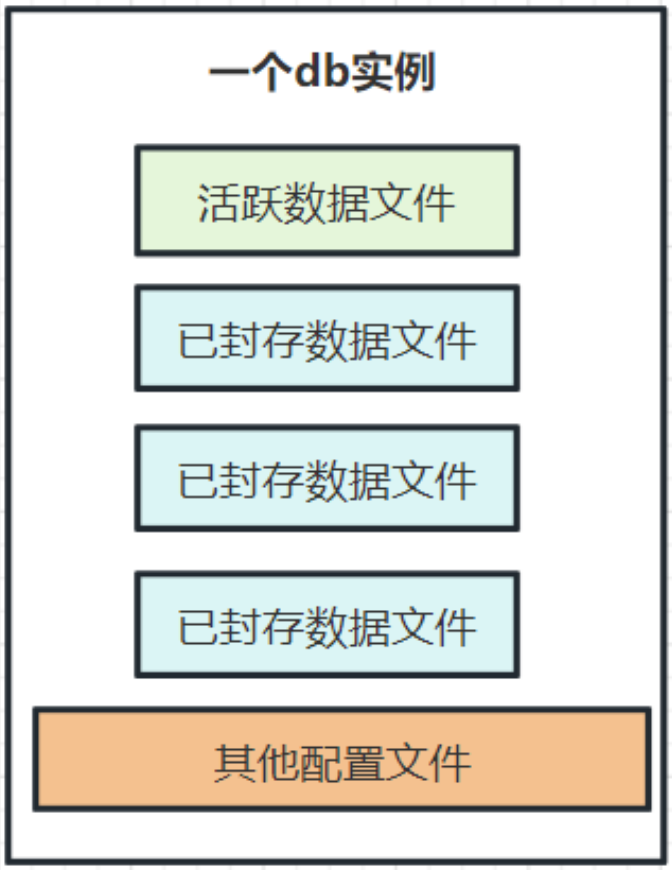
```
    Extra     []byte //操作Entry所需的额外信息
    KeySize   uint32
    ValueSize uint32
    ExtraSize uint32
  }
)
```



2. `db_file`

`db_file`就是一个个的数据文件，包括活跃数据文件和封存数据文件。结构关系如下图所示：





该文件中定义了数据文件的新建，读取，写入，加载等方法。

```
type DBFile struct {
  Id     uint32
  path   string
  File   *os.File
  mmap   mmap.MMap
  Offset int64
  method FileRWMethod
}
```

3. 过期字典

```
type ExpiresValue struct {
  Key      []byte
  KeySize  uint32
  Deadline uint64
}
```

index

包括跳表的实现和通用内存索引`indexer`，内存索引结构如下：

```
type Indexer struct {
  Meta      *storage.Meta //元数据信息
  FileId    uint32        //存储数据的文件id
  EntrySize uint32        //数据条目(Entry)的大小
  Offset    int64         //Entry数据的查询起始位置
}
```

datastruct

数据结构实现目录，包括`hash` `list` `set` `zset` 这几种结构的实现

cmd

客户端和服务端的实现，主要提供访问接口方法

bench

相关的bench_test

项目面试相关问题

这个项目有什么优点?

1. **写性能优秀**：对于所有写入，mindb都是直接在磁盘顺序写，这样可以减少磁盘的寻道时间，实现高吞吐量的写入
2. **读性能尚可**：读操作需要进行一次磁盘IO，且大概率是随机IO，但其可以通过文件系统提供的缓存做一些优化，读过的数据会加速访问。同时mindb提供了内存访问模式，可以在启动数据库的时候进行配置，将数据`value`直接记录在内存中，大大加快读的速度
3. **支持多种数据结构**：在`bitcask`模型中，内存中的数据结构是哈希表，虽然可以达到O(1)的性能，但是不能支持复杂查询。在mindb中支持了`string`,`hash`,`list`,`set`,`zset`五种数据类型，可操作的空间大大增加。
4. **支持过期时间**：在内存中维护一个`map`，记录每个`key`和其对应的过期时间(如果存在的话)，过期时间信息会持久化到一个特定的文件中，启动数据库时加载到内存
5. **实现服务端和客户端**：除了可以直接内嵌使用，还支持命令行操作

介绍一下bitcask模型

`bitcask`是一个日志形式存储，顺序写的存储模型，像是简化版的`LSM`树模型。
一个`bitcask`实例就是一个系统上的文件夹，数据文件在磁盘上顺序写入，文件大小到达限制后，会将其封存变为只读文件，然后另开新的文件进行写入操作，每次只能有一个文件进行写入操作，称为`activc文件`。
数据文件中一条记录称为一个`entry`，包括`key的大小`，`value的大小`，`key和value值`还有`这条记录的操作类型`以及`校验和`等。
更新操作和删除操作也是追加一条记录到数据文件中。
而在内存中维护一个索引，`bitcask`论文中是用的哈希表，每次操作先写文件后更新内存。
读取操作就直接读内存中的索引，索引记录了目标所在的文件位置，然后到磁盘中去进行查找读取。
因为数据文件只写入不删除，所以会有很多冗余，因此`bitcask`有`merge操作`，即遍历所有的数据文件，只保留有效记录。
因为日志型的数据文件本身是无序的，因此查询操作必须依靠内存索引进行，启动数据库时就必须加载索引信息。
`bitcask`使得可以装下数倍于内存的数据，因为内存只需要保存`key`即可。

了解LSM树么

LSM 树的全称为结构化合并树，其设计目标是提供比传统的 B+ 树更好的写性能。
LSM 树将对数据的修改增量保持在内存中，达到指定的大小限制后将这些修改操作批量写入磁盘（由此提升了写性能），是一种基于硬盘的数据结构。
在 LSM 树中，通过将磁盘的随机写转化为顺序写来提高写性能，而付出的代价就是牺牲部分读性能和写放大问题（B+树同样有写放大的问题）。
LSM 相比 B+ 树能提高写性能的本质原因是：外存的随机读写都要慢于顺序读写，无论磁盘还是 SSD。
磁盘中会定期做`merge`操作，合并成一棵大树，以优化读性能。LSM树的优势在于有效地规避了磁盘随机写入问题，但读取时可能需要访问较多的磁盘文件。

内存映射（mmap）是什么?

`mmap`是一种将文件映射到内存的方法，实现文件的磁盘地址和进程虚拟地址空间中的一段虚拟地址的一一映射关系。
也就是说可以在某个进程中通过操作这一段映射的内存，实现对文件的读写等操作，修改了这一段内存的内容，文件对应位置的内容也会同步修改，而获取这一段内存的内容，也相当于读取文件对应位置的内容。
`mmap`一个重要的特性是减少内存的拷贝次数，文件的读写操作一般通过`read`和`write`两个系统调用来实现，这个过程会产生内存拷贝，如`read`函数就涉及到两次内存拷贝：

1. 操作系统读取磁盘文件到页缓存

2. 从页缓存将数据拷贝到用户空间（如`read`的`buf数组`）

而`mmap`只需要一次拷贝，即操作系统读取磁盘文件到页缓存，进程内部可以直接通过指针的方式修改映射的内存，因此`mmap`适合读写频繁的场景，既减少了内存拷贝次数提高效率，又简化了操作。

代码中用的是go的一个开源库`mmap-go`

做项目的过程中有没有什么自己的想法?

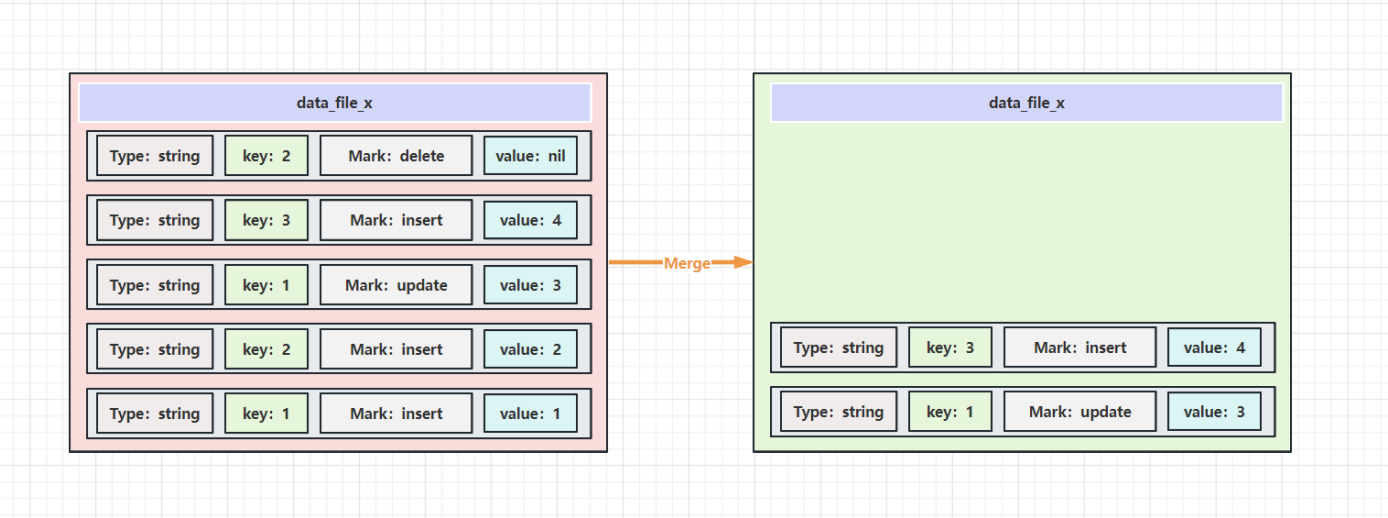
- 在增加`List`等数据类型时对索引的权衡

首先通用索引`indexer`的结构是磁盘对应记录的位置，包括`文件id`，`offset偏移`，`entry大小`，也可选实际`value值`
这样的一个结构是单独的可操作的，其就是一个数据结构中最小的单位。
对于`string`类型来说，一个`indexer`和一个`string`是一一对应的，对`string`的操作就可以直接通过`indexer`反应出来。
而像`List`，`Hash`，`Set`这些结构，它们本身是一些数据的容器，它们的应用场景决定了它们是一定要对其保存的数据做操作的。
比如`List`要`push`和`pop`，`Set`要算交集并集。
如果将其作为一个`indexer`通用索引，首先是每次操作都得将整个结构取出来然后操作完再写一条新的记录回去而造成没必要的空间和性能损耗，其次将一个`List`或者是`Set`这样的结构写到磁盘，它们的数据如何组织也是一个比较麻烦的事情，更可能与原来的`entry`的组织结构发生冲突。
因此最终选择像 redis 一样在内存中直接保留`List`，`Hash`这样的结构值，把`entry`结构由原来的直接记录最新值更改为记录每次操作的类型和本次写入的具体的值，如对`List`进行一个`LPush`操作，`entry`就只记录这次`Lpush`的值，然后更新内存中的值，就相当于全都是`key value`都在内存中的模式。
这样一定程度上牺牲了内存空间，但是也获得了更丰富的操作类型和灵活的使用场景。
如果不需要使用这些数据类型，就只使用简单的`string`类型`kv`存储，那么`string`单独支持仅有`key`在内存中也完全满足使用需求。使用方式可以完全由用户决定。
最后`List`类的数据结构定义如下：

```
type (
  // List list索引结构
  List struct {
    // record 保存list结构的kv映射
    record Record

    // values 保存list中的value值，用以辅助判断某个值是否存在该list
    values map[string]map[string]struct{}
  }

  // Record list record to save
  Record map[string]*list.List
)
```

merge操作如何进行？还可以继续优化么？

`merge`要重新组织磁盘中的数据，回收磁盘空间，当回收的时候整个数据库操作会被阻塞，因此要选择一个合适的时机进行这个操作，比如在关闭数据库前。



1. 遍历五种类型的已封存的文件信息（以`map`形式存在），判断是否有哪种类型的文件数量超过了阈值，如果都没有超过则不需要回收。
2. 回收时先新建临时目录，用于创建新的数据文件，然后进行加锁，避免产生其他数据操作。
3. 因为有五种不同类型的数据文件，而这五种文件也互不干扰，因此可以分别用五个`goroutine`来执行回收操作。用`sync.Map`创建并发安全的临时封存文件索引，之后遍历每种数据类型，以下以`string`类型为例；
4. 先判断当前类型的文件数量是否达到了阈值，如果没有达到则不整理该类型文件。然后遍历当前类型的所有封存文件，遍历每个文件中的`entry`记录，来判断当前的`entry`是否有效
5. 将所有有效的entry放到一个新的entry数组中后，将这些有效的entry重新写入到一批数据文件中，最后因为磁盘中文件的位置发生了变更，因此也要更新内存索引中记录的文件信息，再更新数据库配置即完成全部操作

```
// Reclaim 重新组织磁盘中的数据，回收磁盘空间，回收过程中数据库会阻塞，无法使用
func (db *MinDB) Reclaim() (err error) {

  var reclaimable bool                        // 是否需要回收空间的flag
  for _, archFiles := range db.archFiles { // 遍历所有类型的已封存文件信息
    if len(archFiles) >= db.config.ReclaimThreshold { // 如果某类型的已封存文件数量已经达到了配置的阈值，则可以回收
      reclaimable = true
      break
    }
  }
  if !reclaimable { // 如果所有类型的文件数量都不够阈值，则没必要回收，退出
    return ErrReclaimUnreached
  }

  //新建临时目录，用于暂存新的数据文件
  reclaimPath := db.config.DirPath + reclaimPath
  if err := os.MkdirAll(reclaimPath, os.ModePerm); err != nil {
    return err
  }
```

```
  defer os.RemoveAll(reclaimPath)

  db.mu.Lock() // 回收操作需要加锁，避免有其他数据操作
  defer db.mu.Unlock()

  // 用goroutine处理不同类型的文件
  newArchivedFiles := sync.Map{} // 新的封存文件索引
  reclaimedTypes := sync.Map{}
  wg := sync.WaitGroup{}
  wg.Add(5)
  for i := 0; i < 5; i++ { // dType由const表示,0到4分别表示几种数据类型
    go func(dType uint16) { // 开一个goroutine处理当前类型的文件
      defer func() { // 用defer来做最后的wg.Done()操作
        wg.Done()
      }()

      if len(db.archFiles[dType]) < db.config.ReclaimThreshold { // 如果当前类型的封存文件数量没有达到阈值就不回收此类型
        newArchivedFiles.Store(dType, db.archFiles[dType]) // 注意不回收也要将此类型加入到新的封存文件索引中
        return
      }

      var (
        df        *storage.DBFile
        fileId    uint32
        archFiles = make(map[uint32]*storage.DBFile)
      )

      for _, file := range db.archFiles[dType] { // 遍历当前类型的所有封存文件，key为id，value为文件信息
        var offset int64 = 0
        var reclaimEntries []*storage.Entry // 用一个Entry数组来记录新的有效的entry

        // 读取db中所有当前类型文件，找出有效的entry
        for {
          if e, err := file.Read(offset); err == nil { // 通过offset值去读取文件中的entry，同时offset更新
            if db.validEntry(e, offset, file.Id) { // 判断当前entry是否有效
              reclaimEntries = append(reclaimEntries, e) // 如果有效就将此条entry加入到新的entry数组中
            }
            offset += int64(e.Size()) // 更新offset
          } else { // 如果读取到了文件末尾，就退出
            if err == io.EOF {
              break
            }
            log.Fatalf("err occurred when read the entry: %+v", err)
            return
```

```
            }
          }

          // 将找出来的有效的entry重新写入到新的一批数据文件中
          for _, entry := range reclaimEntries {
            if df == nil || int64(entry.Size())+df.Offset > db.config.BlockSize {
              // 如果df未指向某个文件或者是当前文件将要满了，就新建一个文件
              df, err = storage.NewDBFile(reclaimPath, fileId, db.config.RwMethod,
db.config.BlockSize, dType)
              if err != nil {
                log.Fatalf("err occurred when create new db file: %+v", err)
                return
              }
              archFiles[fileId] = df // 将文件id和文件进行映射缓存
              fileId += 1
            }
            // 对当前文件进行entry的写入
            if err = df.Write(entry); err != nil {
              log.Fatalf("err occurred when write the entry: %+v", err)
              return
            }

            // 因为磁盘中文件的位置发生了变更，因此索引中记录的文件信息也要更新
            if dType == String {
              item := db.strIndex.idxList.Get(entry.Meta.Key)
              idx := item.Value().(*index.Indexer)
              idx.Offset = df.Offset - int64(entry.Size()) // 更新offset
              idx.FileId = fileId                          // 更新文件id
              db.strIndex.idxList.Put(idx.Meta.Key, idx)
            }
          }
        }
        reclaimedTypes.Store(dType, struct{}{})  // 更新merge类型映射
        newArchivedFiles.Store(dType, archFiles) // 更新新的类型与文件组映射
      }(uint16(i))
    }
    wg.Wait()

    // 转移封存文件组
    dbArchivedFiles := make(ArchivedFiles) // 新建封存文件组
    for i := 0; i < 5; i++ {               // 遍历所有类型的新文件组
      dType := uint16(i)
      value, ok := newArchivedFiles.Load(dType) // 从sync.Map中取出文件组转移到新建文件组中
      if !ok {
        log.Printf("one type of data(%d) is missed after reclaiming.", dType)
        return
      }
      dbArchivedFiles[dType] = value.(map[uint32]*storage.DBFile)
    }
```

```
  // 删除掉旧的文件
  for dataType, files := range db.archFiles {
    // 通过回收类型这个map来判断当前类型是否被回收整理过，如果是则删除旧的
    if _, exist := reclaimedTypes.Load(dataType); exist {
      for _, f := range files {
        _ = os.Remove(f.File.Name())
      }
    }
  }

  // 将新的数据文件进行更名
  for dataType, files := range dbArchivedFiles {
    if _, exist := reclaimedTypes.Load(dataType); exist {
      for _, f := range files {
        name := storage.PathSeparator +
fmt.Sprintf(storage.DBFileFormatNames[dataType], f.Id)
        os.Rename(reclaimPath+name, db.config.DirPath+name)
      }
    }
  }

  // 更新数据库配置
  db.archFiles = dbArchivedFiles
  return

}
```

- 继续优化merge操作，可以考虑的是：

bitcask模型中的回收策略是取出全部文件的`Entry`，并将有效的`Entry`重新写入到新的数据文件中去。在数据量较大的情况下，这势必会对磁盘造成较大的压力。
但是每个文件中冗余的数据量分布可能是不均匀的，如果一个数据文件中并没有太多的冗余数据，那么将其取出并扫描重新写入到新的数据文件中就是巨大的无用功。
考虑到这点，所以可以增加对单个文件的`merge`操作来进一步优化。

服务端客户端命令行如何实现的?

对于服务端：

1. 通过`flag`包解析配置信息，如果没有指定配置就加载默认的
2. 新建一个`server`服务，启动一个`goroutine`去处理`server`
3. 用一个`signal`类型的`chan`去监听中断事件，如果服务端服务过程中有对应的中断事件发生，就停止服务
4. 处理`server`的逻辑就是`tcp`服务监听，通过`for select`结构来进行处理客户端连接和逻辑，`server`结构中有一个空`struct`类型的`chan`，如果`chan`收到消息，处理逻辑就停止返回

```
func main() {
  flag.Parse() // 解析配置
```

```
  //set the config
  var cfg mindb.Config
  if *config == "" {
    log.Println("no config set, using the default config.")
    cfg = mindb.DefaultConfig()
  } else {
    c, err := newConfigFromFile(*config)
    if err != nil {
      log.Printf("load config err : %+v\n", err)
      return
    }
    cfg = *c
  }

  if *dirPath == "" {
    log.Println("no dir path set, using the os tmp dir.")
  } else {
    cfg.DirPath = *dirPath
  }

  // 监听中断事件
  sig := make(chan os.Signal, 1)
  signal.Notify(sig, os.Interrupt, os.Kill, syscall.SIGHUP,
    syscall.SIGINT, syscall.SIGTERM, syscall.SIGQUIT)

  server, err := cmd.NewServer(cfg) // 新建一个server
  if err != nil {
    log.Printf("create mindb server err: %+v\n", err)
    return
  }
  go server.Listen(cfg.Addr) // 启动一个goroutine处理server

  <-sig
  server.Stop()
  log.Println("mindb is ready to exit, bye...")
}
```

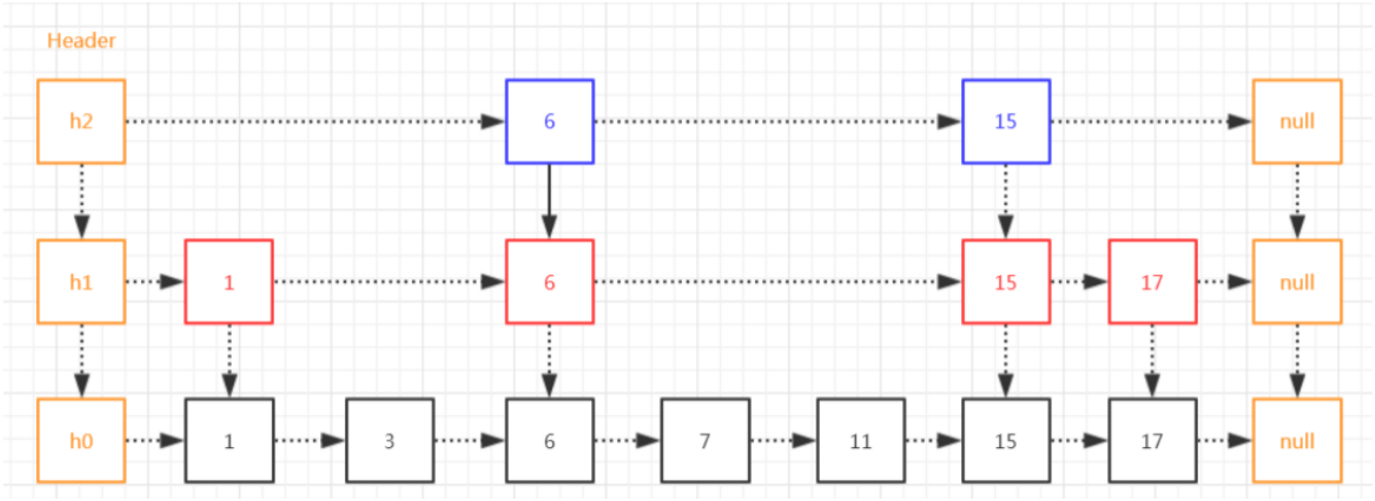
对于客户端：

客户端向服务端请求连接，返回响应即可。

其中命令行的实现主要是用了一个开源组件`liner`

项目中用到了跳表，为什么用，怎么实现的?

在`bitcask`论文中内存中的索引结构是哈希表，其对于复杂查询的需求难以支持。参考其他数据库项目，`LevelDB``Redis`等很多项目都用到了跳表，其查找，插入，删除速度快且实现简单，所以本项目也选择用跳表来支撑复杂的读写需求。



跳表的实现方式很多，这里给出一种跳表的实现方式：

1. 类型定义：

```
const (
  //跳表索引最大层数，可根据实际情况进行调整
  maxLevel    int      = 18
  probability float64 = 1 / math.E
)
type (
  // Node 跳表节点
  Node struct {
    next []*Element // next指针，分层存储
  }

  // Element 跳表存储元素
  Element struct {
    Node
    key   []byte      // 存储 key
    value interface{} // 存储 value
  }

  // SkipList 跳表定义
  SkipList struct {
    Node
    maxLevel       int          // 最大层数
    Len            int          // 跳表长度
    randSource     rand.Source // 随机数生成
    probability    float64
    probTable      []float64
    prevNodesCache []*Node
  }
)
```

2. `put` 方法

```
// Put 存储一个元素至跳表中，如果key已经存在，则会更新其对应的value
// 因此此跳表的实现暂不支持相同的key
func (t *SkipList) Put(key []byte, value interface{}) *Element {
  var element *Element
  prev := t.backNodes(key) // 找出key节点在每一层索引应该放的位置的前一个节点

  if element = prev[0].next[0]; element != nil && bytes.Compare(element.key, key) <= 0
{
    element.value = value // 如果key和prev的下一个节点的key相等，说明该key已存在，更新value返回
即可
    return element
  }

  element = &Element{
    Node: Node{
      next: make([]*Element, t.randomLevel()), // 初始化ele的next索引层
    },
    key:   key,
    value: value,
  }
  // 当前key应该插入的位置已经确定，就在prev的下一个位置
  for i := range element.next { // 遍历ele的所有索引层，建立节点前后联系
    element.next[i] = prev[i].next[i]
    prev[i].next[i] = element
  }

  t.Len++
  return element
}
// 找到key对应的前一个节点索引的信息，即key节点在每一层索引的前一个节点
func (t *SkipList) backNodes(key []byte) []*Node {
  var prev = &t.Node
  var next *Element

  prevs := t.prevNodesCache

  for i := t.maxLevel - 1; i >= 0; i-- { // 从最高层索引开始遍历
    next = prev.next[i] // 当前节点在第i层索引上的下一个节点

    for next != nil && bytes.Compare(key, next.key) > 0 { // 如果目标节点的key比next节点的
key大
      prev = &next.Node   // 将prev放到next节点的位置上
      next = next.next[i] // next通过当前层的索引跳到下一个位置
    } // 循环跳出后，key节点应位于pre和next之间

    prevs[i] = prev // 将当前的prev节点缓存到跳表中的对应层上
  } // 到下一层继续寻找
```

```
  return prevs
}
```

3. get 方法

```
// Get 根据 key 查找对应的 Element 元素
//未找到则返回nil
func (t *SkipList) Get(key []byte) *Element {
  var prev = &t.Node
  var next *Element

  for i := t.maxLevel - 1; i >= 0; i-- {
    next = prev.next[i]

    for next != nil && bytes.Compare(key, next.key) > 0 {
      prev = &next.Node
      next = next.next[i]
    }
  }

  if next != nil && bytes.Compare(next.key, key) <= 0 {
    return next
  }

  return nil
}
```

4. 随机索引层数

```
// 生成索引随机层数
func (t *SkipList) randomLevel() (level int) {
  r := float64(t.randSource.Int63()) / (1 << 63) // 生成一个[0, 1)的概率值

  level = 1
  for level < t.maxLevel && r < t.probTable[level] { // 找到第一个prob小于 r 的层数
    level++
  }
  return
}

func probabilityTable(probability float64, maxLevel int) (table []float64) {
  for i := 1; i <= maxLevel; i++ { // 将每一层的prob值设置为p的(层数减一)次方
    prob := math.Pow(probability, float64(i-1))
    table = append(table, prob)
  }
  return table
}
```

跳表重点问题（从选择跳表到跳表实现中的细节，如果自己写跳表一定要搞懂）：

刚刚说到了`redis`，为什么`redis`的`zset`用跳表实现而不是红黑树?

主要原因是平衡性和复杂性之间的权衡。
跳表在搜索方面具有较好的性能，其查找时间复杂度为O(log n)，相比之下，红黑树是一种自平衡的二叉搜索树，它通过保持一些特定的性质来保持平衡。虽然红黑树在平衡性方面表现出色，但它的实现比较复杂，需要更多的代码来维护平衡和旋转操作。这增加了实现和维护的复杂性，并且可能导致更多的存储空间占用。
而跳表在实现上相对简单，容易理解和实现，并且不需要复杂的旋转操作来保持平衡。因此，Redis选择使用跳表来实现有序集合（ZSet），以在保持良好的性能的同时简化实现过程。

跳表索引的动态更新是怎样做的?

跳表索引的动态更新通过插入和删除元素时的逐层遍历和更新操作来实现。

跳表的高度控制策略是怎样的?

跳表的高度控制策略通常是基于概率的方法，以平衡查询效率和空间占用之间的权衡。
在插入新节点时，为该节点分配一个随机的高度，每个节点有一定的概率获得一个更高的高度，例如有50%的概率获得1个索引层级，25%的概率获得2个索引层级，以此类推。
随机化的高度控制策略可以在一定程度上保持跳表的平衡性。通过随机分配高度，节点在跳表中的分布相对均匀，减少了不平衡的可能性，并且避免了频繁的调整索引层级的开销。

既然跳表的平衡是随机算法控制的，那如何保证`O(logn)`的复杂度?

跳表中每个节点在不同层级上的分布是基于一定的概率，由于节点在各个层级上的分布是以指数递减的，跳表的高度也不会无限增长。在一个大致均衡的跳表中，节点的高度平均为O(log n)，其中 n 是跳表中的节点数量。
在最坏情况下，例如节点高度递增的情况下，操作的时间复杂度可能会达到O(n)。然而，跳表的随机性质和节点高度的概率分布使得这种最坏情况的发生概率相对较低，因此平均时间复杂度仍然保持在O(log n)。