

Golang 常见面试题目解析

交替打印数字和字母

问题描述

使用两个 `goroutine` 交替打印序列，一个 `goroutine` 打印数字，另外一个 `goroutine` 打印字母，最终效果如下：

```
1 12AB34CD56EF78GH910IJ1112KL1314MN15160P1718QR1920ST2122UV2324WX2526YZ2728
```

解题思路

问题很简单，使用 channel 来控制打印的进度。使用两个 channel，来分别控制数字和字母的打印序列，数字打印完成后通过 channel 通知字母打印，字母打印完成后通知数字打印，然后周而复始的工作。

源码参考

```
1 letter,number := make(chan bool),make(chan bool)
2 wait := sync.WaitGroup{}
3 go func() {
4     i := 1
5     for {
6         select {
7             case <-number:
8                 fmt.Println(i)
9                 i++
10                fmt.Println(i)
11                i++
12                letter <- true
13                break
14            default:
15                break
16            }
17        }
18    }()
19    wait.Add(1)
20    go func(wait *sync.WaitGroup) {
21        str := "ABCDEFGHIJKLMNPQRSTUVWXYZ"
22        i := 0
23        for{
```

```

26         select {
27             case <-letter:
28                 if i >= strings.Count(str,"")-1 {
29                     wait.Done()
30                     return
31                 }
32                 fmt.Println(str[i:i+1])
33                 i++
34                 if i >= strings.Count(str,"") {
35                     i = 0
36                 }
37                 fmt.Println(str[i:i+1])
38                 i++
39                 number <- true
40                 break
41             default:
42                 break
43             }
44         }
45     }(&wait)
46     number<-true
47     wait.Wait()

```

源码解析

这里用到了两个 `channel` 负责通知，`letter`负责通知打印字母的goroutine来打印字母，`number`用来通知打印数字的goroutine打印数字。
`wait`用来等待字母打印完成后退出循环。

判断字符串中字符是否全都不同

问题描述

请实现一个算法，确定一个字符串的所有字符【是否全都不同】。这里我们要求【不允许使用额外的储存结构】。给定一个string，请返回一个bool值,true代表所有字符全都不同，false代表存在相同的字符。保证字符串中的字符为【ASCII字符】。字符串的长度小于等于【3000】。

解题思路

这里有几个重点，第一个是 `ASCII字符`， `ASCII字符` 字符一共有256个，其中128个是常用字符，可以在键盘上输入。128之后的是键盘上无法找到的。

然后是全部不同，也就是字符串中的字符没有重复的，再次，不准使用额外的储存结构，且字符串小于等于3000。

如果允许其他额外储存结构，这个题目很好做。如果不允许的话，可以使用golang内置的方式实现。

源码参考

通过 `strings.Count` 函数判断：

```

1 func isUniqueString(s string) bool {
2     if strings.Count(s,"") > 3000{
3         return false

```

```

4 }
5     for _,v := range s {
6         if v > 127 {
7             return false
8         }
9         if strings.Count(s,string(v)) > 1 {
10            return false
11        }
12    }
13    return true
14 }
```

通过 `strings.Index` 和 `strings.LastIndex` 函数判断：

```

1 func isUniqueString2(s string) bool {
2     if strings.Count(s,"") > 3000{
3         return false
4     }
5     for k,v := range s {
6         if v > 127 {
7             return false
8         }
9         if strings.Index(s,string(v)) != k {
10            return false
11        }
12    }
13    return true
14 }
```

源码解析

以上两种方法都可以实现这个算法。

第一个方法使用的是golang内置方法 `strings.Count` ,可以用来判断在一个字符串中包含的另外一个字符串的数量。

第二个方法使用的是golang内置方法 `strings.Index` 和 `strings.LastIndex` , 用来判断指定字符串在另外一个字符串的索引未知，分别是第一次发现位置和最后发现位置。

翻转字符串

问题描述

请实现一个算法，在不使用【额外数据结构和储存空间】的情况下，翻转一个给定的字符串(可以使用单个过程变量)。

给定一个string，请返回一个string，为翻转后的字符串。保证字符串的长度小于等于5000。

解题思路

翻转字符串其实是将一个字符串以中间字符为轴，前后翻转，即将str[len]赋值给str[0]，将str[0] 赋值 str[len]。

源码参考

```

1 func reverString(s string) (string, bool) {
2     str := []rune(s)
```

```
3     l := len(str)
4     if len > 5000 {
5         return s, false
6     }
7     for i := 0; i < len/2; i++ {
8         str[i], str[l-1-i] = str[l-1-i], str[i]
9     }
10    return string(str), true
11 }
```

源码解析

以字符串长度的1/2为轴，前后赋值

判断两个给定的字符串排序后是否一致

问题描述

给定两个字符串，请编写程序，确定其中一个字符串的字符重新排列后，能否变成另一个字符串。这里规定【大小写为不同字符】，且考虑字符串重点空格。给定一个string s1和一个string s2，请返回一个bool，代表两串是否重新排列后可相同。保证两串的长度都小于等于5000。

解题思路

首先要保证字符串长度小于5000。之后只需要一次循环遍历s1中的字符在s2是否存在即可。

源码参考

```
1 func isRegroup(s1,s2 string) bool {
2     sl1 := len([]rune(s1))
3     sl2 := len([]rune(s2))
4     if sl1 > 5000 || sl2 > 5000 || sl1 != sl2{
5         return false
6     }
7     for _,v := range s1 {
8         if strings.Count(s1,string(v)) != strings.Count(s2,string(v)) {
9             return false
10        }
11    }
12    return true
13 }
14 }
```

源码解析

这里还是使用golang内置方法 `strings.Count` 来判断字符是否一致。

字符串替换问题

问题描述

请编写一个方法，将字符串中的空格全部替换为“%20”。假定该字符串有足够的空间存放新增的字符，并且知道字符串的真实长度(小于等于1000)，同时保证字符串由【大小写的英文字母组成】。给定一个string为原始的串，返回替换后的string。

解题思路

两个问题，第一个是只能是英文字母，第二个是替换空格。

源码参考

```
1 func replaceBlank(s string) (string, bool) {
2     if len([]rune(s)) > 1000 {
3         return s, false
4     }
5     for _, v := range s {
6         if string(v) != " " && unicode.IsLetter(v) == false {
7             return s, false
8         }
9     }
10    return strings.Replace(s, " ", "%20", -1), true
11 }
```

源码解析

这里使用了golang内置方法 `unicode.IsLetter` 判断字符是否是字母，之后使用 `strings.Replace` 来替换空格。

机器人坐标问题

问题描述

有一个机器人，给一串指令，L左转 R右转，F前进一步，B后退一步，问最后机器人的坐标，最开始，机器人位于 0 0，方向为正Y。可以输入重复指令n：比如 R2(LF) 这个等于指令 RLFLF。问最后机器人的坐标是多少？

解题思路

这里的一个难点是解析重复指令。主要指令解析成功，计算坐标就简单了。

源码参考

```
1 package main
2 import (
3     "unicode"
4 )
5 const (
6     Left = iota
7     Top
8     Right
9     Bottom
10 )
11 func main() {
12     println(move("R2(LF)", 0, 0, Top))
13 }
14 func move(cmd string, x0 int, y0 int, z0 int) (x, y, z int) {
15     x, y, z = x0, y0, z0
16     repeat := 0
17     repeatCmd := ""
18     for _, s := range cmd {
19         switch {
20             case unicode.IsNumber(s):
21                 repeat = repeat*10 + (int(s) - '0')
22             case s == ')':
23                 repeat--
24         }
25     }
26 }
```

```

27         for i := 0; i < repeat; i++ {
28             x, y, z = move(repeatCmd, x, y, z)
29         }
30         repeat = 0
31         repeatCmd = ""
32     case repeat > 0 && s != '(' && s != ')':
33         repeatCmd = repeatCmd + string(s)
34     case s == 'L':
35         z = (z + 1) % 4
36     case s == 'R':
37         z = (z - 1 + 4) % 4
38     case s == 'F':
39         switch {
40             case z == Left || z == Right:
41                 x = x - z + 1
42             case z == Top || z == Bottom:
43                 y = y - z + 2
44         }
45     case s == 'B':
46         switch {
47             case z == Left || z == Right:
48                 x = x + z - 1
49             case z == Top || z == Bottom:
50                 y = y + z - 2
51         }
52     }
53 }
54 return
55 }
```

源码解析

这里使用三个值表示机器人当前的状况，分别是：x表示x坐标，y表示y坐标，z表示当前方向。L、R命令会改变值z，F、B命令会改变值x、y。值x、y的改变还受当前的z值影响。

如果是重复指令，那么将重复次数和重复的指令存起来递归调用即可。

常见语法题目一

1、下面代码能运行吗？为什么。

```

1 type Param map[string]interface{}
2 type Show struct {
3     Param
4 }
5 func main1() {
6     s := new>Show()
7     s.Param["RMB"] = 10000
8 }
```

```
10 }
```

解析

共发现两个问题：

1. `main` 函数不能加数字。
2. `new` 关键字无法初始化 `Show` 结构体中的 `Param` 属性，所以直接对 `s.Param` 操作会出错。

2、请说出下面代码存在什么问题。

```
1 type student struct {
2     Name string
3 }
4 func zhoujielun(v interface{}) {
5     switch msg := v.(type) {
6         case *student, student:
7             msg.Name
8     }
9 }
10 }
```

解析：

golang中有规定，`switch type` 的 `case T1`，类型列表只有一个，那么 `v := m.(type)` 中的 `v` 的类型就是T1类型。

如果是 `case T1, T2`，类型列表中有多个，那 `v` 的类型还是多对应接口的类型，也就是 `m` 的类型。

所以这里 `msg` 的类型还是 `interface{}`，所以他没有 `Name` 这个字段，编译阶段就会报错。具体解释见：https://golang.org/ref/spec#Type_switches

3、写出打印的结果。

```
1 type People struct {
2     name string `json:"name"`
3 }
4 func main() {
5     js := `{
6         "name":"11"
7     `
8     var p People
9     err := json.Unmarshal([]byte(js), &p)
10    if err != nil {
11        fmt.Println("err: ", err)
12        return
13    }
14    fmt.Println("people: ", p)
15 }
16 }
```

解析：

按照 golang 的语法，小写开头的方法、属性或 `struct` 是私有的，同样，在 `json` 解码或转码的时候也无法上线私有属性的转换。
题目中是无法正常得到 `People` 的 `name` 值的。而且，私有属性 `name` 也不应该加 `json` 的标签。

4、下面的代码是有问题的，请说明原因。

```
1 type People struct {
2     Name string
3 }
4 func (p *People) String() string {
5     return fmt.Sprintf("print: %v", p)
6 }
7 func main() {
8     p := &People{}
9     p.String()
10 }
```

解析：

在golang中 `String() string` 方法实际上是实现了 `String` 的接口的，该接口定义在 `fmt/print.go` 中：

```
1 type Stringer interface {
2     String() string
3 }
```

在使用 `fmt` 包中的打印方法时，如果类型实现了这个接口，会直接调用。而题目中打印 `p` 的时候会直接调用 `p` 实现的 `String()` 方法，然后就产生了循环调用。

5、请找出下面代码的问题所在。

```
1 func main() {
2     ch := make(chan int, 1000)
3     go func() {
4         for i := 0; i < 10; i++ {
5             ch <- i
6         }
7     }()
8     go func() {
9         for {
10             a, ok := <-ch
11             if !ok {
12                 fmt.Println("close")
13                 return
14             }
15         }
16     }()
17 }
```

```
15         fmt.Println("a: ", a)
16     }
17 }()
18 close(ch)
19 fmt.Println("ok")
20 time.Sleep(time.Second * 100)
21 }
```

解析：

在 golang 中 `goroutine` 的调度时间是不确定的，在题目中，第一个写 `channel` 的 `goroutine` 可能还未调用，或已调用但没有写完时直接 `close` 管道，可能导致写失败，既然出现 `panic` 错误。

6、请说明下面代码书写是否正确。

```
1 var value int32
2 func SetValue(delta int32) {
3     for {
4         v := value
5         if atomic.CompareAndSwapInt32(&value, v, (v+delta)) {
6             break
7         }
8     }
9 }
10 }
```

解析：

`atomic.CompareAndSwapInt32` 函数不需要循环调用。

7、下面的程序运行后为什么会爆异常。

```
1 type Project struct{}
2 func (p *Project) deferError() {
3     if err := recover(); err != nil {
4         fmt.Println("recover: ", err)
5     }
6 }
7 func (p *Project) exec(msgchan chan interface{}) {
8     for msg := range msgchan {
9         m := msg.(int)
10        fmt.Println("msg: ", m)
11    }
12 }
13 }
14 func (p *Project) run(msgchan chan interface{}) {
15     for {
16         defer p.deferError()
17         go p.exec(msgchan)
18     }
19 }
```

```

20         time.Sleep(time.Second * 2)
21     }
22 }
23 func (p *Project) Main() {
24     a := make(chan interface{}, 100)
25     go p.run(a)
26     go func() {
27         for {
28             a <- "1"
29             time.Sleep(time.Second)
30         }
31     }()
32     time.Sleep(time.Second * 1000000000000000)
33 }
34 }
35 func main() {
36     p := new(Project)
37     p.Main()
38 }
39 }
```

解析：

有一下几个问题：

1. `time.Sleep` 的参数数值太大，超过了 `1<<63 - 1` 的限制。
2. `defer p.deferError()` 需要在协程开始出调用，否则无法捕获 `panic`。

8、请说出下面代码哪里写错了

```

1 func main() {
2     abc := make(chan int, 1000)
3     for i := 0; i < 10; i++ {
4         abc <- i
5     }
6     go func() {
7         for a := range abc {
8             fmt.Println("a: ", a)
9         }
10    }()
11    close(abc)
12    fmt.Println("close")
13    time.Sleep(time.Second * 100)
14 }
```

解析：

协程可能还未启动，管道就关闭了。

9、请说出下面代码，执行时为什么会报错

```

1 type Student struct {
2     name string
3 }
4 func main() {
5     m := map[string]Student{"people": {"zhoujielun"}}
6     m["people"].name = "wuyanzu"
7 }
8 
```

解析:

map的value本身是不可寻址的，因为map中的值会在内存中移动，并且旧的指针地址在map改变时会变得无效。故如果需要修改map值，可以将 `map` 中的非指针类型 `value`，修改为指针类型，比如使用 `map[string]*Student` .

10、请说出下面的代码存在什么问题？

```

1 type query func(string) string
2 func exec(name string, vs ...query) string {
3     ch := make(chan string)
4     fn := func(i int) {
5         ch <- vs[i](name)
6     }
7     for i, _ := range vs {
8         go fn(i)
9     }
10    return <-ch
11 }
12 }
13 func main() {
14     ret := exec("111", func(n string) string {
15         return n + "func1"
16     }, func(n string) string {
17         return n + "func2"
18     }, func(n string) string {
19         return n + "func3"
20     }, func(n string) string {
21         return n + "func4"
22     })
23     fmt.Println(ret)
24 }
```

解析:

依据4个goroutine的启动后执行效率，很可能打印111func4，但其他的111func*也可能先执行，exec只会返回一条信息。

11、下面这段代码为什么会卡死？

```

1 package main
```

```
3 import (
4     "fmt"
5     "runtime"
6 )
7 func main() {
8     var i byte
9     go func() {
10         for i = 0; i <= 255; i++ {
11             }
12     }()
13     fmt.Println("Dropping mic")
14     // Yield execution to force executing other goroutines
15     runtime.Gosched()
16     runtime.GC()
17     fmt.Println("Done")
18 }
19 }
```

解析：

Golang 中，byte 其实被 alias 到 uint8 上了。所以上面的 for 循环会始终成立，因为 i++ 到 i=255 的时候会溢出， $i \leq 255$ 一定成立。

也即是，for 循环永远无法退出，所以上面的代码其实可以等价于这样：

```
1 go func() {
2     for {}
3 }
```

正在被执行的 goroutine 发生以下情况时让出当前 goroutine 的执行权，并调度后面的 goroutine 执行：

- IO 操作
- Channel 阻塞
- system call
- 运行较长时间

如果一个 goroutine 执行时间太长，scheduler 会在其 G 对象上打上一个标志（preempt），当这个 goroutine 内部发生函数调用的时候，会先主动检查这个标志，如果为 true 则会让出执行权。

main 函数里启动的 goroutine 其实是一个没有 IO 阻塞、没有 Channel 阻塞、没有 system call、没有函数调用的死循环。

也就是，它无法主动让出自己的执行权，即使已经执行很长时间，scheduler 已经标志了 preempt。

而 golang 的 GC 动作是需要所有正在运行 goroutine 都停止后进行的。因此，程序会卡在 `runtime.GC()` 等待所有协程退出。

常见语法题目二

1、写出下面代码输出内容。

```

1 package main
2 import (
3     "fmt"
4 )
5 func main() {
6     defer_call()
7 }
8 func defer_call() {
9     defer func() { fmt.Println("打印前") }()
10    defer func() { fmt.Println("打印中") }()
11    defer func() { fmt.Println("打印后") }()
12    panic("触发异常")
13 }
14
15
16
17 }
```

解析：

`defer` 关键字的实现跟 go 关键字很类似，不同的是它调用的是 `runtime.deferproc` 而不是 `runtime.newproc`。

在 `defer` 出现的地方，插入了指令 `call runtime.deferproc`，然后在函数返回之前的地方，插入指令 `call runtime.deferreturn`。

goroutine 的控制结构中，有一张表记录 `defer`，调用 `runtime.deferproc` 时会将需要 `defer` 的表达式记录在表中，而在调用 `runtime.deferreturn` 的时候，则会依次从 `defer` 表中出栈并执行。

因此，题目最后输出顺序应该是 `defer` 定义顺序的倒序。`panic` 错误并不能终止 `defer` 的执行。

2、以下代码有什么问题，说明原因

```

1 type student struct {
2     Name string
3     Age   int
4 }
5 func pase_student() {
6     m := make(map[string]*student)
7     stus := []student{
8         {Name: "zhou", Age: 24},
9         {Name: "li", Age: 23},
10        {Name: "wang", Age: 22},
11    }
12    for _, stu := range stus {
13        m[stu.Name] = &stu
14    }
15 }
```

解析：

golang 的 `for ... range` 语法中，`stu` 变量会被复用，每次循环会将集合中的值复制给这个变量，因此，会导致最后 `m` 中的 `map` 中储存的都是 `stus` 最后一个 `student`

的值。

3、下面的代码会输出什么，并说明原因

```
1 func main() {
2     runtime.GOMAXPROCS(1)
3     wg := sync.WaitGroup{}
4     wg.Add(20)
5     for i := 0; i < 10; i++ {
6         go func() {
7             fmt.Println("i: ", i)
8             wg.Done()
9         }()
10    }
11    for i := 0; i < 10; i++ {
12        go func(i int) {
13            fmt.Println("i: ", i)
14            wg.Done()
15        }(i)
16    }
17    wg.Wait()
18 }
```

解析：

这个输出结果决定来自于调度器优先调度哪个G。从runtime的源码可以看到，当创建一个G时，会优先放入到下一个调度的 `runnext` 字段上作为下一次优先调度的G。因此，最先输出的是最后创建的G，也就是9。

```
1 func newproc(siz int32, fn *funcval) {
2     argp := add(unsafe.Pointer(&fn), sys.PtrSize)
3     gp := getg()
4     pc := getcallerpc()
5     systemstack(func() {
6         newg := newproc1(fn, argp, siz, gp, pc)
7         _p_ := getg().m.p.ptr()
8         //新创建的G会调用这个方法来决定如何调度
9         runqput(_p_, newg, true)
10        if mainStarted {
11            wakep()
12        }
13    })
14 }
15 ...
16 ...
17 ...
18 if next {
19     retryNext:
20         oldnext := _p_.runnext
21         //当next是true时总会将新进来的G放入下一次调度字段中
22         if !_p_.runnext.cas(oldnext, guintptr(unsafe.Pointer(gp))) {
```

```
24         goto retryNext
25     }
26     if oldnext == 0 {
27         return
28     }
29     // Kick the old runnext out to the regular run queue.
30     gp = oldnext.ptr()
31 }
```

4、下面代码会输出什么？

```
1 type People struct{}
2 func (p *People) ShowA() {
3     fmt.Println("showA")
4     p.ShowB()
5 }
6 func (p *People) ShowB() {
7     fmt.Println("showB")
8 }
9 type Teacher struct {
10     People
11 }
12 func (t *Teacher) ShowB() {
13     fmt.Println("teacher showB")
14 }
15 func main() {
16     t := Teacher{}
17     t.ShowA()
18 }
```

解析：

输出结果为 `showA`、`showB`。golang 语言中没有继承概念，只有组合，也没有虚方法，更没有重载。因此，`*Teacher` 的 `ShowB` 不会覆盖被组合的 `People` 的方法。

5、下面代码会触发异常吗？请详细说明

```
1 func main() {
2     runtime.GOMAXPROCS(1)
3     int_chan := make(chan int, 1)
4     string_chan := make(chan string, 1)
5     int_chan <- 1
6     string_chan <- "hello"
7     select {
8         case value := <-int_chan:
9             fmt.Println(value)
10        case value := <-string_chan:
```

```
11     panic(value)
12 }
13 }
```

解析：

结果是随机执行。golang 在多个 `case` 可读的时候会公平的选中一个执行。

6、下面代码输出什么？

```
1 func calc(index string, a, b int) int {
2     ret := a + b
3     fmt.Println(index, a, b, ret)
4     return ret
5 }
6 func main() {
7     a := 1
8     b := 2
9     defer calc("1", a, calc("10", a, b))
10    a = 0
11    defer calc("2", a, calc("20", a, b))
12    b = 1
13 }
14 }
```

解析：

输出结果为：

```
1 10 1 2 3
2 20 0 2 2
3 2 0 2 2
4 1 1 3 4
```

`defer` 在定义的时候会计算好调用函数的参数，所以会优先输出 `10`、`20` 两个参数。然后根据定义的顺序倒序执行。

7、请写出以下输入内容

```
1 func main() {
2     s := make([]int, 5)
3     s = append(s, 1, 2, 3)
4     fmt.Println(s)
5 }
```

解析：

输出为 `0 0 0 0 1 2 3`。

`make` 在初始化切片时指定了长度，所以追加数据时会从 `[len(s)]` 位置开始填充数据。

8、下面的代码有什么问题？

```
1 type UserAges struct {
2     ages map[string]int
3     sync.Mutex
4 }
5 func (ua *UserAges) Add(name string, age int) {
6     ua.Lock()
7     defer ua.Unlock()
8     ua.ages[name] = age
9 }
10 func (ua *UserAges) Get(name string) int {
11     if age, ok := ua.ages[name]; ok {
12         return age
13     }
14     return -1
15 }
16 }
```

解析：

在执行 Get 方法时可能被 panic。

虽然有使用 sync.Mutex 做写锁，但是 map 是并发读写不安全的。map 属于引用类型，并发读写时多个协程见是通过指针访问同一个地址，即访问共享变量，此时同时读写资源存在竞争关系。会报错误信息：“fatal error: concurrent map read and map write”。

因此，在 Get 中也需要加锁，因为这里只是读，建议使用读写锁 sync.RWMutex。

9、下面的迭代会有什么问题？

```
1 func (set *threadSafeSet) Iter() <-chan interface{} {
2     ch := make(chan interface{})
3     go func() {
4         set.RLock()
5         for elem := range set.s {
6             ch <- elem
7         }
8         close(ch)
9         set.RUnlock()
10    }()
11    return ch
12 }
```

解析：

默认情况下 make 初始化的 channel 是无缓冲的，也就是在迭代写时会阻塞。

10、以下代码能编译过去吗？为什么？

```

1 package main
2 import (
3     "fmt"
4 )
5 type People interface {
6     Speak(string) string
7 }
8 type Student struct{}
9 func (stu *Student) Speak(think string) (talk string) {
10     if think == "bitch" {
11         talk = "You are a good boy"
12     } else {
13         talk = "hi"
14     }
15     return
16 }
17 func main() {
18     var peo People = Student{}
19     think := "bitch"
20     fmt.Println(peo.Speak(think))
21 }
22 }
```

解析:

编译失败, 值类型 `Student{}` 未实现接口 `People` 的方法, 不能定义为 `People` 类型。

在 golang 语言中, `Student` 和 `*Student` 是两种类型, 第一个表示 `Student` 本身, 第二个是指向 `Student` 的指针。

11、以下代码打印出来什么内容, 说出为什么。。。

```

1 package main
2 import (
3     "fmt"
4 )
5 type People interface {
6     Show()
7 }
8 type Student struct{}
9 func (stu *Student) Show() {
10 }
11 func live() People {
12     var stu *Student
13     return stu
14 }
15 func main() {
16     if live() == nil {
17         fmt.Println("AAAAAAA")
18 }
```

```
25     } else {
26         fmt.Println("BBBBBBBB")
27     }
28 }
```

解析：

跟上一题一样，不同的是 `*Student` 的定义后本身没有初始化值，所以 `*Student` 是 `nil` 的，但是 `*Student` 实现了 `People` 接口，接口不为 `nil`。

在 golang 协程和channel配合使用

写代码实现两个 goroutine，其中一个产生随机数并写入到 go channel 中，另外一个从 channel 中读取数字并打印到标准输出。最终输出五个随机数。

解析

这是一道很简单的golang基础题目，实现方法也有很多种，一般想答让面试官满意的答案还是有几点注意的地方。

1. `goroutine` 在golang中式非阻塞的
2. `channel` 无缓冲情况下，读写都是阻塞的，且可以用 `for` 循环来读取数据，当管道关闭后，`for` 退出。
3. golang 中有专用的 `select case` 语法从管道读取数据。

示例代码如下：

```
1 func main() {
2     out := make(chan int)
3     wg := sync.WaitGroup{}
4     wg.Add(2)
5     go func() {
6         defer wg.Done()
7         for i := 0; i < 5; i++ {
8             out <- rand.Intn(5)
9         }
10        close(out)
11    }()
12    go func() {
13        defer wg.Done()
14        for i := range out {
15            fmt.Println(i)
16        }
17    }()
18    wg.Wait()
19 }
```

实现阻塞读且并发安全的map

GO里面MAP如何实现key不存在 get操作等待 直到key存在或者超时，保证并发安全，且需要实现以下接口：

```
1 type sp interface {
```

```

2     Out(key string, val interface{}) //存入key /val, 如果该key读取的
3         goroutine挂起, 则唤醒。此方法不会阻塞, 时刻都可以立即执行并返回
4     Rd(key string, timeout time.Duration) interface{} //读取一个key, 如果
5         key不存在阻塞, 等待key存在或者超时
6 }

```

解析:

看到阻塞协程第一个想到的就是 `channel`，题目中要求并发安全，那么必须用锁，还要实现多个 `goroutine` 读的时候如果值不存在则阻塞，直到写入值，那么每个键值需要有一个阻塞 `goroutine` 的 `channel`。

实现如下：

```

1 type Map struct {
2     c    map[string]*entry
3     rmx *sync.RWMutex
4 }
5 type entry struct {
6     ch      chan struct{}
7     value   interface{}
8     isExist bool
9 }
10 func (m *Map) Out(key string, val interface{}) {
11     m.rmx.Lock()
12     defer m.rmx.Unlock()
13     item, ok := m.c[key]
14     if !ok {
15         m.c[key] = &entry{
16             value: val,
17             isExist: true,
18         }
19     }
20     return
21 }
22     item.value = val
23     if !item.isExist {
24         if item.ch != nil {
25             close(item.ch)
26             item.ch = nil
27         }
28     }
29     return
30 }

```

高并发下的锁与map的读写

场景：在一个高并发的web服务器中，要限制IP的频繁访问。现模拟100个IP同时并发访问服务器，每个IP要重复访问1000次。

每个IP三分钟之内只能访问一次。修改以下代码完成该过程，要求能成功输出 success:100

```

1 package main
2

```

```

3 import (
4     "fmt"
5     "time"
6 )
7
8 type Ban struct {
9     visitIPs map[string]time.Time
10 }
11
12 func NewBan() *Ban {
13     return &Ban{visitIPs: make(map[string]time.Time)}
14 }
15 func (o *Ban) visit(ip string) bool {
16     if _, ok := o.visitIPs[ip]; ok {
17         return true
18     }
19     o.visitIPs[ip] = time.Now()
20     return false
21 }
22 func main() {
23     success := 0
24     ban := NewBan()
25     for i := 0; i < 1000; i++ {
26         for j := 0; j < 100; j++ {
27             go func() {
28                 ip := fmt.Sprintf("192.168.1.%d", j)
29                 if !ban.visit(ip) {
30                     success++
31                 }
32             }()
33         }
34     }
35     fmt.Println("success:", success)
36 }
37 }
```

解析

该问题主要考察了并发情况下map的读写问题，而给出的初始代码，又存在 `for` 循环中启动 `goroutine` 时变量使用问题以及 `goroutine` 执行滞后问题。

因此，首先要保证启动的 `goroutine` 得到的参数是正确的，然后保证 `map` 的并发读写，最后保证三分钟只能访问一次。

多CPU核心下修改 `int` 的值极端情况下会存在不同步情况，因此需要原子性的修改int值。

下面给出的实例代码，是启动了一个协程每分钟检查一下 `map` 中的过期 `ip`，`for` 启动协程时传参。

```

1 package main
2 import (
3     "context"
4     "fmt"
5     "sync"
6 )
```

```

7   "sync/atomic"
8   "time"
9 }
10 type Ban struct {
11   visitIPs map[string]time.Time
12   lock      sync.Mutex
13 }
14 }
15 func NewBan(ctx context.Context) *Ban {
16   o := &Ban{visitIPs: make(map[string]time.Time)}
17   go func() {
18     timer := time.NewTimer(time.Minute * 1)
19     for {
20       select {
21         case <-timer.C:
22           o.lock.Lock()
23           for k, v := range o.visitIPs {
24             if time.Now().Sub(v) >= time.Minute*1 {
25               delete(o.visitIPs, k)
26             }
27           }
28           o.lock.Unlock()
29           timer.Reset(time.Minute * 1)
30         case <-ctx.Done():
31           return
32         }
33     }
34   }()
35   return o
36 }
37 }
38 func (o *Ban) visit(ip string) bool {
39   o.lock.Lock()
40   defer o.lock.Unlock()
41   if _, ok := o.visitIPs[ip]; ok {
42     return true
43   }
44   o.visitIPs[ip] = time.Now()
45   return false
46 }
47 func main() {
48   success := int64(0)
49   ctx, cancel := context.WithCancel(context.Background())
50   defer cancel()
51   ban := NewBan(ctx)
52   wait := &sync.WaitGroup{}
53   wait.Add(1000 * 100)
54   for i := 0; i < 1000; i++ {
55     for j := 0; j < 100; j++ {
56       go func(j int) {
57         defer wait.Done()
58       }()
59     }
60   }

```

```

61         ip := fmt.Sprintf("192.168.1.%d", j)
62         if !ban.visit(ip) {
63             atomic.AddInt64(&success, 1)
64         }
65     }(j)
66 }
67 }
68 wait.Wait()
69 fmt.Println("success:", success)
70
71 }
72 }
```

写出以下逻辑，要求每秒钟调用一次proc并保证程序不退出？

```

1 package main
2 func main() {
3     go func() {
4         // 1 在这里需要你写算法
5         // 2 要求每秒钟调用一次proc函数
6         // 3 要求程序不能退出
7         }()
8     select {}
9 }
10 func proc() {
11     panic("ok")
12 }
```

解析

题目主要考察了两个知识点：

1. 定时执行执行任务
2. 捕获 panic 错误

题目中要求每秒钟执行一次，首先想到的就是 `time.Ticker` 对象，该函数可每秒钟往 `chan` 中放一个 `Time`，正好符合我们的要求。

在 `golang` 中捕获 `panic` 一般会用到 `recover()` 函数。

```

1 package main
2 import (
3     "fmt"
4     "time"
5 )
6 func main() {
7     go func() {
8         // 1 在这里需要你写算法
9         // 2 要求每秒钟调用一次proc函数
10        // 3 要求程序不能退出
11        t := time.NewTicker(time.Second * 1)
12        for {
13            select {
14                case <-t.C:
15                    go func() {
```

```

19         defer func() {
20             if err := recover(); err != nil {
21                 fmt.Println(err)
22             }
23         }()
24     proc()
25         }()
26     }()
27 }()
28 }()
29 select {}
30 }
31 }
32 func proc() {
33     panic("ok")
34 }
35 }
```

为 sync.WaitGroup 中Wait函数支持 WaitTimeout 功能.

```

1 package main
2 import (
3     "fmt"
4     "sync"
5     "time"
6 )
7
8 func main() {
9     wg := sync.WaitGroup{}
10    c := make(chan struct{})
11    for i := 0; i < 10; i++ {
12        wg.Add(1)
13        go func(num int, close <-chan struct{}) {
14            defer wg.Done()
15            <-close
16            fmt.Println(num)
17        }(i, c)
18    }
19    if WaitTimeout(&wg, time.Second*5) {
20        close(c)
21        fmt.Println("timeout exit")
22    }
23    time.Sleep(time.Second * 10)
24 }
25
26 func WaitTimeout(wg *sync.WaitGroup, timeout time.Duration) bool {
27     // 要求手写代码
28     // 要求sync.WaitGroup支持timeout功能
29     // 如果timeout到了超时时间返回true
30     // 如果WaitGroup自然结束返回false
31
32 }
33 }
```

解析

首先 `sync.WaitGroup` 对象的 `Wait` 函数本身是阻塞的，同时，超时用到的 `time.Timer` 对象也需要阻塞的读。

同时阻塞的两个对象肯定要每个启动一个协程,每个协程去处理一个阻塞，难点在于怎么知道哪个阻塞先完成。

目前我用的方式是声明一个没有缓冲的 `chan`，谁先完成谁优先向管道中写入数据。

```
1 package main
2 import (
3     "fmt"
4     "sync"
5     "time"
6 )
7
8 func main() {
9     wg := sync.WaitGroup{}
10    c := make(chan struct{})
11    for i := 0; i < 10; i++ {
12        wg.Add(1)
13        go func(num int, close <-chan struct{}) {
14            defer wg.Done()
15            <-close
16            fmt.Println(num)
17        }(i, c)
18    }
19
20    if WaitTimeout(&wg, time.Second*5) {
21        close(c)
22        fmt.Println("timeout exit")
23    }
24    time.Sleep(time.Second * 10)
25 }
26
27 func WaitTimeout(wg *sync.WaitGroup, timeout time.Duration) bool {
28     // 要求手写代码
29     // 要求sync.WaitGroup支持timeout功能
30     // 如果timeout到了超时时间返回true
31     // 如果WaitGroup自然结束返回false
32     ch := make(chan bool, 1)
33     go time.AfterFunc(timeout, func() {
34         ch <- true
35     })
36     go func() {
37         wg.Wait()
38         ch <- false
39     }()
40
41     return <- ch
42 }
43
44 }
```

语法找错题

写出以下代码出现的问题

```
1 package main
2 import (
3     "fmt"
4 )
5 func main() {
6     var x string = nil
7     if x == nil {
8         x = "default"
9     }
10    fmt.Println(x)
11 }
```

golang 中字符串是不能赋值 `nil` 的，也不能跟 `nil` 比较。

写出以下打印内容

```
1 package main
2 import "fmt"
3 const (
4     a = iota
5     b = iota
6 )
7 const (
8     name = "menglu"
9     c     = iota
10    d     = iota
11 )
12 func main() {
13     fmt.Println(a)
14     fmt.Println(b)
15     fmt.Println(c)
16     fmt.Println(d)
17 }
```

找出下面代码的问题

```
1 package main
2 import "fmt"
3 type query func(string) string
4 func exec(name string, vs ...query) string {
5     ch := make(chan string)
```

```

7   fn := func(i int) {
8     ch <- vs[i](name)
9   }
10    for i, _ := range vs {
11      go fn(i)
12    }
13    return <-ch
14  }
15
16 func main() {
17   ret := exec("111", func(n string) string {
18     return n + "func1"
19   }, func(n string) string {
20     return n + "func2"
21   }, func(n string) string {
22     return n + "func3"
23   }, func(n string) string {
24     return n + "func4"
25   })
26   fmt.Println(ret)
27 }
```

上面的代码有严重的内存泄漏问题，出错的位置是 `go fn(i)`，实际上代码执行后会启动 4 个协程，但是因为 `ch` 是非缓冲的，只可能有一个协程写入成功。而其他三个协程会一直在后台等待写入。

写出以下打印结果，并解释下为什么这么打印的。

```

1 package main
2 import (
3   "fmt"
4 )
5 func main() {
6   str1 := []string{"a", "b", "c"}
7   str2 := str1[1:]
8   str2[1] = "new"
9   fmt.Println(str1)
10  str2 = append(str2, "z", "x", "y")
11  fmt.Println(str1)
12 }
```

golang 中的切片底层其实使用的是数组。当使用 `str1[1:]` 使，`str2` 和 `str1` 底层共享一个数组，这回导致 `str2[1] = "new"` 语句影响 `str1`。

而 `append` 会导致底层数组扩容，生成新的数组，因此追加数据后的 `str2` 不会影响 `str1`。

但是为什么对 `str2` 复制后影响的确实 `str1` 的第三个元素呢？这是因为切片 `str2` 是从数组的第二个元素开始，`str2` 索引为 1 的元素对应的是 `str1` 索引为 2 的元素。

写出以下打印结果

```
1 package main
2 import (
3     "fmt"
4 )
5 type Student struct {
6     Name string
7 }
8 func main() {
9     fmt.Println(&Student{Name: "menglub"} == &Student{Name: "menglub"})
10    fmt.Println(Student{Name: "menglub"} == Student{Name: "menglub"})
11 }
12 }
```

个人理解：指针类型比较的是指针地址，非指针类型比较的是每个属性的值。

写出以下代码的问题

```
1 package main
2 import (
3     "fmt"
4 )
5 func main() {
6     fmt.Println([...]string{"1"} == [...string{"1"}])
7     fmt.Println([]string{"1"} == []string{"1"})
8 }
```

数组只能与相同纬度长度以及类型的其他数组比较，切片之间不能直接比较。。

下面代码写法有什么问题？

```
1 package main
2 import (
3     "fmt"
4 )
5 type Student struct {
6     Age int
7 }
8 func main() {
9     kv := map[string]Student{"menglub": {Age: 21}}
10    kv["menglub"].Age = 22
11    s := []Student{{Age: 21}}
12    s[0].Age = 22
13    fmt.Println(kv, s)
```

14 }

golang中的 `map` 通过 `key` 获取到的实际上是两个值，第一个是获取到的值，第二个是是否存在该 `key`。因此不能直接通过 `key` 来赋值对象。

golang 并发题目测试

题目来源： [Go并发编程小测验：你能答对几道题？](#)

1 Mutex

```
1 package main
2 import (
3     "fmt"
4     "sync"
5 )
6 var mu sync.Mutex
7 var chain string
8 func main() {
9     chain = "main"
10    A()
11    fmt.Println(chain)
12 }
13 func A() {
14     mu.Lock()
15     defer mu.Unlock()
16     chain = chain + " --> A"
17     B()
18 }
19 func B() {
20     chain = chain + " --> B"
21     C()
22 }
23 func C() {
24     mu.Lock()
25     defer mu.Unlock()
26     chain = chain + " --> C"
27 }
```

- A: 不能编译
- B: 输出 `main --> A --> B --> C`
- C: 输出 `main`
- D: panic

2 RWMutex

```
1 package main
```

```

2 import (
3     "fmt"
4     "sync"
5     "time"
6 )
7 var mu sync.RWMutex
8 var count int
9 func main() {
10     go A()
11     time.Sleep(2 * time.Second)
12     mu.Lock()
13     defer mu.Unlock()
14     count++
15     fmt.Println(count)
16 }
17 func A() {
18     mu.RLock()
19     defer mu.RUnlock()
20     B()
21 }
22 func B() {
23     time.Sleep(5 * time.Second)
24     C()
25 }
26 func C() {
27     mu.RLock()
28     defer mu.RUnlock()
29 }
```

- A: 不能编译
- B: 输出 1
- C: 程序hang住
- D: panic

3 Waitgroup

```

1 package main
2 import (
3     "sync"
4     "time"
5 )
6 func main() {
7     var wg sync.WaitGroup
8     wg.Add(1)
9     go func() {
10         time.Sleep(time.Millisecond)
11         wg.Done()
```

```
12     wg.Add(1)
13 }
14 wg.Wait()
15 }
```

- A: 不能编译
- B: 无输出, 正常退出
- C: 程序hang住
- D: panic

4 双检查实现单例

```
1 package doublecheck
2 import (
3     "sync"
4 )
5 type Once struct {
6     m     sync.Mutex
7     done uint32
8 }
9 func (o *Once) Do(f func()) {
10    if o.done == 1 {
11        return
12    }
13    o.m.Lock()
14    defer o.m.Unlock()
15    if o.done == 0 {
16        o.done = 1
17        f()
18    }
19 }
```

- A: 不能编译
- B: 可以编译, 正确实现了单例
- C: 可以编译, 有并发问题, f函数可能会被执行多次
- D: 可以编译, 但是程序运行会panic

5 Mutex

```
1 package main
2 import (
3     "fmt"
4     "sync"
5 )
6 type MyMutex struct {
7     count int
8 }
```

```

8     sync.Mutex
9 }
10 func main() {
11     var mu MyMutex
12     mu.Lock()
13     var mu2 = mu
14     mu.count++
15     mu.Unlock()
16     mu2.Lock()
17     mu2.count++
18     mu2.Unlock()
19     fmt.Println(mu.count, mu2.count)
20 }
```

- A: 不能编译
- B: 输出 1, 1
- C: 输出 1, 2
- D: panic

6 Pool

```

1 package main
2 import (
3     "bytes"
4     "fmt"
5     "runtime"
6     "sync"
7     "time"
8 )
9 var pool = sync.Pool{New: func() interface{} { return new(bytes.Buffer) }}
10 func main() {
11     go func() {
12         for {
13             processRequest(1 << 28) // 256MiB
14         }
15     }()
16     for i := 0; i < 1000; i++ {
17         go func() {
18             for {
19                 processRequest(1 << 10) // 1KiB
20             }
21         }()
22     }
23     var stats runtime.MemStats
24     for i := 0; ; i++ {
25         runtime.ReadMemStats(&stats)
26         fmt.Printf("Cycle %d: %dB\n", i, stats.Alloc)
```

```

27     time.Sleep(time.Second)
28     runtime.GC()
29 }
30 }
31 func processRequest(size int) {
32     b := pool.Get().(*bytes.Buffer)
33     time.Sleep(500 * time.Millisecond)
34     b.Grow(size)
35     pool.Put(b)
36     time.Sleep(1 * time.Millisecond)
37 }
```

- A: 不能编译
- B: 可以编译, 运行时正常, 内存稳定
- C: 可以编译, 运行时内存可能暴涨
- D: 可以编译, 运行时内存先暴涨, 但是过一会儿会回收掉

7 channel

```

1 package main
2 import (
3     "fmt"
4     "runtime"
5     "time"
6 )
7 func main() {
8     var ch chan int
9     go func() {
10         ch = make(chan int, 1)
11         ch <- 1
12     }()
13     go func(ch chan int) {
14         time.Sleep(time.Second)
15         <-ch
16     }(ch)
17     c := time.Tick(1 * time.Second)
18     for range c {
19         fmt.Printf("#goroutines: %d\n", runtime.NumGoroutine())
20     }
21 }
```

- A: 不能编译
- B: 一段时间后总是输出 #goroutines: 1
- C: 一段时间后总是输出 #goroutines: 2
- D: panic

8 channel

```
1 package main
2 import "fmt"
3 func main() {
4     var ch chan int
5     var count int
6     go func() {
7         ch <- 1
8     }()
9     go func() {
10        count++
11        close(ch)
12    }()
13    <-ch
14    fmt.Println(count)
15 }
```

- A: 不能编译
- B: 输出 1
- C: 输出 0
- D: panic

9 Map

```
1 package main
2 import (
3     "fmt"
4     "sync"
5 )
6 func main() {
7     var m sync.Map
8     m.LoadOrStore("a", 1)
9     m.Delete("a")
10    fmt.Println(m.Len())
11 }
```

- A: 不能编译
- B: 输出 1
- C: 输出 0
- D: panic

10 happens before

```
1 package main
```

```
2 var c = make(chan int)
3 var a int
4 func f() {
5     a = 1
6     <-c
7 }
8 func main() {
9     go f()
10    c <- 0
11    print(a)
12 }
```

- A: 不能编译
- B: 输出 1
- C: 输出 0
- D: panic

答案

1. D

会产生死锁 `panic`，因为 `Mutex` 是互斥锁。

2. D

会产生死锁 `panic`，根据 `sync/rwmutex.go` 中注释可以知道，读写锁当有一个协程在等待写锁时，其他协程是不能获得读锁的，而在 `A` 和 `C` 中同一个调用链中间需要让出读锁，让写锁优先获取，而 `A` 的读锁又要求 `C` 调用完成，因此死锁。

3. D

`WaitGroup` 在调用 `Wait` 之后是不能再调用 `Add` 方法的。

4. C

在多核CPU中，因为CPU缓存会导致多个核心中变量值不同步。

5. D

加锁后复制变量，会将锁的状态也复制，所以 `mu1` 其实是已经加锁状态，再加锁会死锁。

6. C

个人理解，在单核CPU中，内存可能会稳定在 256MB，如果是多核可能会暴涨。

7. C

因为 ch 未初始化，写和读都会阻塞，之后被第一个协程重新赋值，导致写的 ch 都阻塞。

8. D

ch 未有被初始化，关闭时会报错。

9. A

sync.Map 没有 Len 方法。

10. B

c <- 0 会阻塞依赖于 f() 的执行。

记一道字节跳动的算法面试题

题目

这其实是一道变形的链表反转题，大致描述如下 给定一个单链表的头节点 head，实现一个调整单链表的函数，使得每K个节点之间为一组进行逆序，并且从链表的尾部开始组起，头部剩余节点数量不够一组的不需要逆序。（不能使用队列或者栈作为辅助）

例如：

链表：1->2->3->4->5->6->7->8->null, K = 3。那么 6->7->8， 3->4->5， 1->2 各位一组。调整后：1->2->5->4->3->8->7->6->null。其中 1, 2 不调整，因为不够一组。

解析

原文：<https://juejin.im/post/5d4f76325188253b49244dd0>

多协程查询切片问题

题目

假设有一个超长的切片，切片的元素类型为int，切片中的元素为乱序排序。限时5秒，使用多个goroutine查找切片中是否存在给定的值，在查找到目标值或者超时后立刻结束所有goroutine的执行。

比如，切片 `[23,32,78,43,76,65,345,762,.....915,86]`，查找目标值为 345，如果切片中存在，则目标值输出 `"Found it!"` 并立即取消仍在执行查询任务的 goroutine。

如果在超时时间未查到目标值程序，则输出 `"Timeout! Not Found"`，同时立即取消仍在执行的查找任务的 goroutine。

答案: <https://mp.weixin.qq.com/s/GhC2WDw3VHP91DrrFVCnag>

对已经关闭的的chan进行读写，会怎么样？为什么？

题目

对已经关闭的的 chan 进行读写，会怎么样？为什么？

回答

- 读已经关闭的 chan 能一直读到东西，但是读到的内容根据通道内关闭前是否有元素而不同。
 - 如果 chan 关闭前，buffer 内有元素还未读，会正确读到 chan 内的值，且返回的第二个 bool 值（是否读成功）为 true。
 - 如果 chan 关闭前，buffer 内有元素已经被读完，chan 内无值，接下来所有接收的值都会非阻塞直接成功，返回 channel 元素的零值，但是第二个 bool 值一直为 false。
- 写已经关闭的 chan 会 panic

示例

1. 写已经关闭的 chan

```
1 func main(){
2     c := make(chan int,3)
3     close(c)
4     c <- 1
5 }
6 //输出结果
7 panic: send on closed channel
8 goroutine 1 [running]
9 main.main()
10 ...
11 ...
```

- 注意这个 send on closed channel, 待会会提到。

2. 读已经关闭的 chan

```

1 package main
2 import "fmt"
3 func main() {
4     fmt.Println("以下是数值的chan")
5     ci:=make(chan int,3)
6     ci<-1
7     close(ci)
8     num,ok := <- ci
9     fmt.Printf("读chan的协程结束, num=%v, ok=%v\n",num,ok)
10    num1,ok1 := <-ci
11    fmt.Printf("再读chan的协程结束, num=%v, ok=%v\n",num1,ok1)
12    num2,ok2 := <-ci
13    fmt.Printf("再再读chan的协程结束, num=%v, ok=%v\n",num2,ok2)
14
15
16    fmt.Println("以下是字符串chan")
17    cs := make(chan string,3)
18    cs <- "aaa"
19    close(cs)
20    str,ok := <- cs
21    fmt.Printf("读chan的协程结束, str=%v, ok=%v\n",str,ok)
22    str1,ok1 := <-cs
23    fmt.Printf("再读chan的协程结束, str=%v, ok=%v\n",str1,ok1)
24    str2,ok2 := <-cs
25    fmt.Printf("再再读chan的协程结束, str=%v, ok=%v\n",str2,ok2)
26    fmt.Println("以下是结构体chan")
27    type MyStruct struct{
28        Name string
29    }
30    cstruct := make(chan MyStruct,3)
31    cstruct <- MyStruct{Name: "haha"}
32    close(cstruct)
33    stru,ok := <- cstruct
34    fmt.Printf("读chan的协程结束, stru=%v, ok=%v\n",stru,ok)
35    stru1,ok1 := <-cs
36    fmt.Printf("再读chan的协程结束, stru=%v, ok=%v\n",stru1,ok1)
37    stru2,ok2 := <-cs
38    fmt.Printf("再再读chan的协程结束, stru=%v, ok=%v\n",stru2,ok2)
39
40 }
```

输出结果

```

1 以下是数值的chan
2 读chan的协程结束, num=1, ok=true
3 再读chan的协程结束, num=0, ok=false
```

```
4 再再读chan的协程结束, num=0, ok=false
5 以下是字符串chan
6 读chan的协程结束, str=aaa, ok=true
7 再读chan的协程结束, str=, ok=false
8 再再读chan的协程结束, str=, ok=false
9 以下是结构体chan
10 读chan的协程结束, stru={haha}, ok=true
11 再读chan的协程结束, stru=, ok=false
12 再再读chan的协程结束, stru=, ok=false
```

多问一句

1. 为什么写已经关闭的 chan 就会 panic 呢?

```
1 //在 src/runtime/chan.go
2 func chansend(c *hchan,ep unsafe.Pointer,block bool,callerpc uintptr) bool
{
3     //省略其他
4     if c.closed != 0 {
5         unlock(&c.lock)
6         panic(plainError("send on closed channel"))
7     }
8     //省略其他
9 }
```

- 当 `c.closed != 0` 则为通道关闭, 此时执行写, 源码提示直接 `panic`, 输出的内容就是上面提到的 `"send on closed channel"`。

2. 为什么读已关闭的 chan 会一直能读到值?

```
1 func chanrecv(c *hchan,ep unsafe.Pointer,block bool) (selected,received
2 bool) {
3     //省略部分逻辑
4     lock(&c.lock)
5     //当chan被关闭了, 而且缓存为空时
6     //ep 是指 val,ok := <-c 里的val地址
7     if c.closed != 0 && c.qcount == 0 {
8         if receenabled {
9             raceacquire(c.raceaddr())
10        }
11        unlock(&c.lock)
12        //如果接受之的地址不空, 那接收值将获得一个该值类型的零值
13        //typedmemclr 会根据类型清理响应的内存
14        //这就解释了上面代码为什么关闭的chan 会返回对应类型的零值
15    }
```

```
14     if ep != null {
15         typedmemclr(c.elemtype,ep)
16     }
17     //返回两个参数 selected,received
18     // 第二个采纳数就是 val,ok := <- c 里的 ok
19     //也就解释了为什么读关闭的chan会一直返回false
20     return true,false
21 }
22 }
```

- `c.closed != 0 && c.qcount == 0` 指通道已经关闭，且缓存为空的情况下（已经读完了之前写到通道里的值）
- 如果接收值的地址 `ep` 不为空
 - 那接收值将获得是一个该类型的零值
 - `typedmemclr` 会根据类型清理相应地址的内存
 - 这就解释了上面代码为什么关闭的 chan 会返回对应类型的零值

简单聊聊内存逃逸？

问题

知道golang的内存逃逸吗？什么情况下会发生内存逃逸？

回答

golang程序变量会携带有一组校验数据，用来证明它的整个生命周期是否在运行时完全可知。如果变量通过了这些校验，它就可以在栈上分配。否则就说它 逃逸 了，必须在堆上分配。

能引起变量逃逸到堆上的典型情况：

- 在方法内把局部变量指针返回 局部变量原本应该在栈中分配，在栈中回收。但是由于返回时被外部引用，因此其生命周期大于栈，则溢出。
- 发送指针或带有指针的值到 channel 中。在编译时，是没有办法知道哪个 `goroutine` 会在 `channel` 上接收数据。所以编译器没法知道变量什么时候才会被释放。
- 在一个切片上存储指针或带指针的值。一个典型的例子就是 `[]*string`。这会导致切片的内容逃逸。尽管其后面的数组可能是在栈上分配的，但其引用的值一定是在堆上。
- slice 的背后数组被重新分配了，因为 append 时可能会超出其容量(cap)。slice 初始化的地方在编译时是可以知道的，它最开始会在栈上分配。如果切片背后的存储要基于运行时的数据进行扩充，就会在堆上分配。
- 在 interface 类型上调用方法。在 interface 类型上调用方法都是动态调度的——方法的真正实现只能在运行时知道。想像一个 `io.Reader` 类型的变量 `r`，调用 `r.Read(b)` 会使得 `r` 的值和切片 `b` 的背后存储都逃逸掉，所以会在堆上分配。

举例

通过一个例子加深理解，接下来尝试下怎么通过 `go build -gcflags=-m` 查看逃逸的情况。

```
1 package main
2 import "fmt"
3 type A struct {
4     s string
5 }
6 // 这是上面提到的 "在方法内把局部变量指针返回" 的情况
7 func foo(s string) *A {
8     a := new(A)
9     a.s = s
10    return a // 返回局部变量a，在C语言中妥妥野指针，但在go则ok，但a会逃逸到堆
11 }
12 func main() {
13     a := foo("hello")
14     b := a.s + " world"
15     c := b + "!"
16     fmt.Println(c)
17 }
```

执行`go build -gcflags=-m main.go`

```
1 go build -gcflags=-m main.go
2 # command-line-arguments
3 ./main.go:7:6: can inline foo
4 ./main.go:13:10: inlining call to foo
5 ./main.go:16:13: inlining call to fmt.Println
6 /var/folders/45/qx9lfw2s2zzgvhzg3mtzkwc0000gn/T/go-
build409982591/b001/_gomod_.go:6:6: can inline init.0
7 ./main.go:7:10: leaking param: s
8 ./main.go:8:10: new(A) escapes to heap
9 ./main.go:16:13: io.Writer(os.Stdout) escapes to heap
10 ./main.go:16:13: c escapes to heap
11 ./main.go:15:9: b + "!" escapes to heap
12 ./main.go:13:10: main new(A) does not escape
13 ./main.go:14:11: main a.s + " world" does not escape
14 ./main.go:16:13: main []interface {} literal does not escape
15 <autogenerated>:1: os.(*File).close .this does not escape
```

- `./main.go:8:10: new(A) escapes to heap` 说明 `new(A)` 逃逸了,符合上述提到的常见情况中的第一种。
- `./main.go:14:11: main a.s + " world" does not escape` 说明 `b` 变量没有逃逸，因为它只在方法内存在，会在方法结束时被回收。
- `./main.go:15:9: b + "!" escapes to heap` 说明 `c` 变量逃逸，通过 `fmt.Println(a ...interface{})` 打印的变量，都会发生逃逸，感兴趣的朋友可以去查查为什么。

以上操作其实就叫逃逸分析。下篇文章，跟大家聊聊怎么用一个比较trick的方法使变量不逃逸。方便大家在面试官面前秀一波。

原文 <https://mp.weixin.qq.com/s/4YYR1eYFIFsNOaTxL4Q-eQ>

字符串转成byte数组，会发生内存拷贝吗？

问题

字符串转成byte数组，会发生内存拷贝吗？

回答

字符串转成切片，会产生拷贝。严格来说，只要是发生类型强转都会发生内存拷贝。那么问题来了。

频繁的内存拷贝操作听起来对性能不大友好。有没有什么办法可以在字符串转成切片的时候不用发生拷贝呢？

解释

```
1 package main
2 import (
3     "fmt"
4     "reflect"
5     "unsafe"
6 )
7 func main() {
8     a := "aaa"
9     ssh := (*reflect.StringHeader)(unsafe.Pointer(&a))
10    b := (*[]byte)(unsafe.Pointer(&ssh))
11    fmt.Printf("%v", b)
12 }
```

`StringHeader` 是字符串在go的底层结构。

```
1 type StringHeader struct {
2     Data uintptr
3     Len   int
4 }
```

`SliceHeader` 是切片在go的底层结构。

```
1 type SliceHeader struct {
2     Data uintptr
3     Len   int
4     Cap   int
5 }
```

```
5 }
```

那么如果想要在底层转换二者，只需要把 StringHeader 的地址强转成 SliceHeader 就行。那么go有个很强的包叫 unsafe 。

1. `unsafe.Pointer(&a)` 方法可以得到变量a的地址。
2. `(*reflect.StringHeader)(unsafe.Pointer(&a))` 可以把字符串a转成底层结构的形式。
3. `([*]byte)(unsafe.Pointer(&ssh))` 可以把ssh底层结构体转成byte的切片的指针。
4. 再通过 `*` 转为指针指向的实际内容。

Golang 理论

Goroutine调度策略

原文： [第三章 Goroutine调度策略 \(16\)](#)

在调度器概述一节我们提到过，所谓的goroutine调度，是指程序代码按照一定的算法在适当的时候挑选出合适的goroutine并放到CPU上去运行的过程。这句话揭示了调度系统需要解决的三大核心问题：

- 调度时机：什么时候会发生调度？
- 调度策略：使用什么策略来挑选下一个进入运行的goroutine？
- 切换机制：如何把挑选出来的goroutine放到CPU上运行？

对这三大问题的解决构成了调度器的所有工作，因而我们对调度器的分析也必将围绕着它们所展开。

第二章我们已经详细的分析了调度器的初始化以及goroutine的切换机制，本章将重点讨论调度器如何挑选下一个goroutine出来运行的策略问题，而剩下的与调度时机相关的内容我们将在第4~6章进行全面的分析。

再探schedule函数

在讨论main goroutine的调度时我们已经见过schedule函数，因为当时我们的主要关注点在于main goroutine是如何被调度到CPU上运行的，所以并未对schedule函数如何挑选下一个goroutine出来运行做深入的分析，现在是重新回到schedule函数详细分析其调度策略的时候了。

[runtime/proc.go : 2467](#)

```
1 // One round of scheduler: find a runnable goroutine and execute it.
2 // Never returns.
3 func schedule() {
4     _g_ := getg()    // _g_ = m.g0
5     .....
6     var gp *g
7     .....
8
9
10
11
12     if gp == nil {
13         // Check the global runnable queue once in a while to ensure fairness.
14         // Otherwise two goroutines can completely occupy the local runqueue
15         // by constantly respawning each other.
```

```

16     //为了保证调度的公平性，每个工作线程每进行61次调度就需要优先从全局运行队列中
17     //获取goroutine出来运行,
18     //因为如果只调度本地运行队列中的goroutine，则全局运行队列中的goroutine有可能得不到运行
19     if _g_.m.p.ptr().schedtick%61 == 0 && sched.runqsize > 0 {
20         lock(&sched.lock) //所有工作线程都能访问全局运行队列，所以需要加锁
21         gp = globrunqget(_g_.m.p.ptr(), 1) //从全局运行队列中获取1个
22         goroutine
23             unlock(&sched.lock)
24     }
25     if gp == nil {
26         //从与m关联的p的本地运行队列中获取goroutine
27         gp, inheritTime = runqget(_g_.m.p.ptr())
28         if gp != nil && _g_.m.spinning {
29             throw("schedule: spinning with local work")
30         }
31     if gp == nil {
32         //如果从本地运行队列和全局运行队列都没有找到需要运行的goroutine,
33         //则调用findRunnable函数从其它工作线程的运行队列中偷取，如果偷取不到，则
34         //当前工作线程进入睡眠,
35         //直到获取到需要运行的goroutine之后findRunnable函数才会返回。
36         gp, inheritTime = findRunnable() // blocks until work is available
37     }
38     .....
39     //当前运行的是runtime的代码，函数调用栈使用的是g0的栈空间
40     //调用execete切换到gp的代码和栈空间去运行
41     execute(gp, inheritTime)
42 }

```

schedule函数分三步分别从各运行队列中寻找可运行的goroutine:

- 第一步，从全局运行队列中寻找goroutine。为了保证调度的公平性，每个工作线程每经过61次调度就需要优先尝试从全局运行队列中找出一个goroutine来运行，这样才能保证位于全局运行队列中的goroutine得到调度的机会。全局运行队列是所有工作线程都可以访问的，所以在访问它之前需要加锁。
- 第二步，从工作线程本地运行队列中寻找goroutine。如果不需要或不能从全局运行队列中获取到goroutine则从本地运行队列中获取。
- 第三步，从其它工作线程的运行队列中偷取goroutine。如果上一步也没有找到需要运行的goroutine，则调用findRunnable从其他工作线程的运行队列中偷取goroutine，findRunnable函数在偷取之前会再次尝试从全局运行队列和当前线程的本地运行队列中查找需要运行的goroutine。

下面我们先来看如何从全局运行队列中获取goroutine。

从全局运行队列中获取goroutine

从全局运行队列中获取可运行的goroutine是通过globrunqget函数来完成的，该函数的第一个参数是与当前工作线程绑定的p，第二个参数max表示最多可以从全局队列中拿多少个g到当前工作线程的本地运行队列中来。

[runtime/proc.go : 4663](#)

```
1 // Try get a batch of G's from the global runnable queue.
2 // Sched must be locked.
3 func globrunqget(_p_ *p, max int32) *g {
4     if sched.runqsize == 0 { //全局运行队列为空
5         return nil
6     }
7     //根据p的数量平分全局运行队列中的goroutines
8     n := sched.runqsize / gomaxprocs + 1
9     if n > sched.runqsize { //上面计算n的方法可能导致n大于全局运行队列中的
10        goroutine数量
11        n = sched.runqsize
12    }
13    if max > 0 && n > max {
14        n = max //最多取max个goroutine
15    }
16    if n > int32(len(_p_.runq)) / 2 {
17        n = int32(len(_p_.runq)) / 2 //最多只能取本地队列容量的一半
18    }
19    sched.runqsize -= n
20    //直接通过函数返回gp，其它的goroutines通过runqput放入本地运行队列
21    gp := sched.runq.pop() //pop从全局运行队列的队列头取
22    n--
23    for ; n > 0; n-- {
24        gp1 := sched.runq.pop() //从全局运行队列中取出一个goroutine
25        runqput(_p_, gp1, false) //放入本地运行队列
26    }
27    return gp
28 }
29 }
```

globrunqget函数首先会根据全局运行队列中goroutine的数量，函数参数max以及_p_的本地队列的容量计算出到底应该拿多少个goroutine，然后把第一个g结构体对象通过返回值的方式返回给调用函数，其它的则通过runqput函数放入当前工作线程的本地运行队列。这段代码值得一提的是，计算应该从全局运行队列中拿走多少个goroutine时根据p的数量(gomaxprocs)做了负载均衡。

如果没有从全局运行队列中获取到goroutine，那么接下来就在工作线程的本地运行队列中寻找需要运行的goroutine。

从工作线程本地运行队列中获取goroutine

从代码上来看，工作线程的本地运行队列其实分为两个部分，一部分是由p的runq、runqhead和runqtail这三个成员组成的一个无锁循环队列，该队列最多可包含256个

goroutine；另一部分是p的runnext成员，它是一个指向g结构体对象的指针，它最多只包含一个goroutine。

从本地运行队列中寻找goroutine是通过 `runqget` 函数完成的，寻找时，代码首先查看 `runnext` 成员是否为空，如果不为空则返回 `runnext` 所指的goroutine，并把 `runnext` 成员清零，如果 `runnext` 为空，则继续从循环队列中查找goroutine。

[runtime/proc.go : 4825](#)

```
1 // Get g from local runnable queue.
2 // If inheritTime is true, gp should inherit the remaining time in the
3 // current time slice. Otherwise, it should start a new time slice.
4 // Executed only by the owner P.
5 func runqget(_p_ *p) (gp *g, inheritTime bool) {
6     // If there's a runnext, it's the next G to run.
7     //从runnext成员中获取goroutine
8     for {
9         //查看runnext成员是否为空，不为空则返回该goroutine
10        next := _p_.runnext
11        if next == 0 {
12            break
13        }
14        if _p_.runnext.cas(next, 0) {
15            return next.ptr(), true
16        }
17    }
18    //从循环队列中获取goroutine
19    for {
20        h := atomic.LoadAcq(&_p_.runqhead) // load-acquire, synchronize
21        with other consumers
22        t := _p_.runqtail
23        if t == h {
24            return nil, false
25        }
26        gp := _p_.runq[h%uint32(len(_p_.runq))].ptr()
27        if atomic.CasRel(&_p_.runqhead, h, h+1) { // cas-release, commits
28            consume
29            return gp, false
30        }
31    }
}
```

这里首先需要注意的是不管是从 `runnext` 还是从循环队列中拿取goroutine都使用了cas操作，这里的cas操作是必需的，因为可能有其他工作线程此时此刻也正在访问这两个成员，从这里偷取可运行的goroutine。

其次，代码中对 `runqhead` 的操作使用了 `atomic.LoadAcq` 和 `atomic.CasRel`，它们分别提供了 `load-acquire` 和 `cas-release` 语义。

对于 `atomic.LoadAcq` 来说，其语义主要包含如下几条：

- 原子读取，也就是说不管代码运行在哪种平台，保证在读取过程中不会有其它线程对该变量进行写入；

- 位于 `atomic.LoadAcq` 之后的代码，对内存的读取和写入必须在 `atomic.LoadAcq` 读取完成后才能执行，编译器和CPU都不能打乱这个顺序；
- 当前线程执行 `atomic.LoadAcq` 时可以读取到其它线程最近一次通过 `atomic.CasRel` 对同一个变量写入的值，与此同时，位于 `atomic.LoadAcq` 之后的代码，不管读取哪个内存地址中的值，都可以读取到其它线程中位于 `atomic.CasRel` (对同一个变量操作) 之前的代码最近一次对内存的写入。

对于 `atomic.CasRel` 来说，其语义主要包含如下几条：

- 原子的执行比较并交换的操作；
- 位于 `atomic.CasRel` 之前的代码，对内存的读取和写入必须在 `atomic.CasRel` 对内存的写入之前完成，编译器和CPU都不能打乱这个顺序；
- 线程执行 `atomic.CasRel` 完成后其它线程通过 `atomic.LoadAcq` 读取同一个变量可以读到最新的值，与此同时，位于 `atomic.CasRel` 之前的代码对内存写入的值，可以被其它线程中位于 `atomic.LoadAcq` (对同一个变量操作) 之后的代码读取到。

因为可能有多个线程会并发的修改和读取 `runqhead`，以及需要依靠 `runqhead` 的值来读取 `runq` 数组的元素，所以需要使用 `atomic.LoadAcq` 和 `atomic.CasRel` 来保证上述语义。

我们可能会问，为什么读取 `p` 的 `runqtail` 成员不需要使用 `atomic.LoadAcq` 或 `atomic.load`？因为 `runqtail` 不会被其它线程修改，只会被当前工作线程修改，此时没有人修改它，所以也就不需要使用原子相关的操作。

最后，由 `p` 的 `runq`、`runqhead` 和 `runqtail` 这三个成员组成的这个无锁循环队列非常精妙，我们会在后面的章节对这个循环队列进行分析。

CAS操作与ABA问题

我们知道使用 `cas` 操作需要特别注意 ABA 的问题，那么 `runqget` 函数这两个使用 `cas` 的地方会不会有问题呢？答案是这两个地方都不会有 ABA 的问题。原因分析如下：

首先来看对 `runnext` 的 `cas` 操作。只有跟 `_p_` 绑定的当前工作线程才会去修改 `runnext` 为一个非 0 值，其它线程只会把 `runnext` 的值从一个非 0 值修改为 0 值，然而跟 `_p_` 绑定的当前工作线程正在此处执行代码，所以在当前工作线程读取到值 A 之后，不可能有线程修改其值为 B(0) 之后再修改回 A。

再来看对 `runq` 的 `cas` 操作。当前工作线程操作的是 `_p_` 的本地队列，只有跟 `_p_` 绑定在一起的当前工作线程才会因为往该队列里面添加 `goroutine` 而去修改 `runqtail`，而其它工作线程不会往该队列里面添加 `goroutine`，也就不会去修改 `runqtail`，它们只会修改 `runqhead`，所以，当我们这个工作线程从 `runqhead` 读取到值 A 之后，其它工作线程也就不可能修改 `runqhead` 的值为 B 之后再第二次把它修改为值 A (因为 `runqtail` 在这段时间之内不可能被修改，`runqhead` 的值也就无法越过 `runqtail` 再回绕到 A 值)，也就是说，代码从逻辑上已经杜绝了引发 ABA 的条件。

到此，我们已经分析完工作线程从全局运行队列和本地运行队列获取 `goroutine` 的代码，由于篇幅的限制，我们下一节再来分析从其它工作线程的运行队列偷取 `goroutine` 的流程。

goroutine简介

`goroutine` 是 Go 语言实现的用户态线程，主要用来解决操作系统线程太“重”的问题，所谓的太重，主要表现在以下两个方面：

- 创建和切换太重：操作系统线程的创建和切换都需要进入内核，而进入内核所消耗的性能代价比较高，开销较大；
- 内存使用太重：一方面，为了尽量避免极端情况下操作系统线程栈的溢出，内核在创建操作系统线程时默认会为其分配一个较大的栈内存（虚拟地址空间，内核并不会一开始就分配这么多的物理内存），然而在绝大多数情况下，系统线程远远用不了这么多内存，这导致了浪费；另一方面，栈内存空间一旦创建和初始化完成之后其大小就不能再有变化，这决定了在某些特殊场景下系统线程栈还是有溢出的风险。

而相对的，用户态的goroutine则轻量得多：

- goroutine是用户态线程，其创建和切换都在用户代码中完成而无需进入操作系统内核，所以其开销要远远小于系统线程的创建和切换；
- goroutine启动时默认栈大小只有2k，这在多数情况下已经够用了，即使不够用，goroutine的栈也会自动扩大，同时，如果栈太大了过于浪费它还能自动收缩，这样既没有栈溢出的风险，也不会造成栈内存空间的大量浪费。

正是因为Go语言中实现了如此轻量级的线程，才使得我们在Go程序中，可以轻易的创建成千上万甚至上百万的goroutine出来并发的执行任务而不用太担心性能和内存等问题。

注意：为了避免混淆，从现在开始，后面出现的所有线程一词均是指操作系统线程，而goroutine我们不再称之为线程而是直接使用goroutine这个词。

线程模型与调度器

第一章讨论操作系统线程调度的时候我们曾经提到过，goroutine建立在操作系统线程基础之上，它与操作系统线程之间实现了一个多对多(M:N)的两级线程模型。

这里的 M:N 是指M个goroutine运行在N个操作系统线程之上，内核负责对这N个操作系统线程进行调度，而这N个系统线程又负责对这M个goroutine进行调度和运行。

所谓的对goroutine的调度，是指程序代码按照一定的算法在适当的时候挑选出合适的goroutine并放到CPU上去运行的过程，这些负责对goroutine进行调度的程序代码我们称之为goroutine调度器。用极度简化了的伪代码来描述goroutine调度器的工作流程大概是下面这个样子：

```

1 // 程序启动时的初始化代码
2 .....
3 for i := 0; i < N; i++ { // 创建N个操作系统线程执行schedule函数
4     create_os_thread(schedule) // 创建一个操作系统线程执行schedule函数
5 }
6 //schedule函数实现调度逻辑
7 func schedule() {
8     for { //调度循环
9         // 根据某种算法从M个goroutine中找出一个需要运行的goroutine
10        g := find_a_runnable_goroutine_from_M_goroutines()
11        run_g(g) // CPU运行该goroutine，直到需要调度其它goroutine才返回
12        save_status_of_g(g) // 保存goroutine的状态，主要是寄存器的值
13    }
14 }
15 }
```

这段伪代码表达的意思是，程序运行起来之后创建了N个由内核调度的操作系统线程（为了方便描述，我们称这些系统线程为工作线程）去执行schedule函数，而schedule函数在一个调度循环中反复从M个goroutine中挑选出一个需要运行的goroutine并跳转到该goroutine去运行，直到需要调度其它goroutine时才返回到schedule函数中通过save_status_of_g保存刚刚正在运行的goroutine的状态然后再次去寻找下一个goroutine。

需要强调的是，这段伪代码对goroutine的调度代码做了高度的抽象、修改和简化处理，放在这里只是为了帮助我们从宏观上了解goroutine的两级调度模型，具体的实现原理和细节将从本章开始进行全面介绍。

重要的结构体

下面介绍的这些结构体中的字段非常多，牵涉到的细节也很庞杂，光是看这些结构体的定义我们没有必要也无法真正理解它们的用途，所以在里我们只需要大概了解一下就行了，看不懂记不住都没有关系，随着后面对代码逐步深入的分析，我们也必将会对这些结构体有越来越清晰的认识。为了节省篇幅，下面各结构体的定义略去了跟调度器无关的成员。另外，这些结构体的定义全部位于Go语言的源代码路径下的[runtime/runtime2.go](#)文件之中。

stack结构体

stack结构体主要用来记录goroutine所使用的栈的信息，包括栈顶和栈底位置：

```
1 // Stack describes a Go execution stack.  
2 // The bounds of the stack are exactly [lo, hi),  
3 // with no implicit data structures on either side.  
4 // 用于记录goroutine使用的栈的起始和结束位置  
5 type stack struct {  
6     lo uintptr    // 栈顶，指向内存低地址  
7     hi uintptr    // 栈底，指向内存高地址  
8 }
```

gobuf结构体

gobuf结构体用于保存goroutine的调度信息，主要包括CPU的几个寄存器的值：

```
1 type gobuf struct {  
2     // The offsets of sp, pc, and g are known to (hard-coded in) libmach.  
3     //  
4     // ctxt is unusual with respect to GC: it may be a  
5     // heap-allocated funcval, so GC needs to track it, but it  
6     // needs to be set and cleared from assembly, where it's  
7     // difficult to have write barriers. However, ctxt is really a  
8     // saved, live register, and we only ever exchange it between  
9     // the real register and the gobuf. Hence, we treat it as a  
10    // root during stack scanning, which means assembly that saves
```

```

11 // and restores it doesn't need write barriers. It's still
12 // typed as a pointer so that any other writes from Go get
13 // write barriers.
14 sp uintptr // 保存CPU的rsp寄存器的值
15 pc uintptr // 保存CPU的rip寄存器的值
16 g guintptr // 记录当前这个gobuf对象属于哪个goroutine
17 ctxt unsafe.Pointer
18
19 // 保存系统调用的返回值, 因为从系统调用返回之后如果p被其它工作线程抢占,
20 // 则这个goroutine会被放入全局运行队列被其它工作线程调度, 其它线程需要知道系统
21 // 调用的返回值。
22 ret sys.Uintreg
23 lr uintptr
24
25 // 保存CPU的rip寄存器的值
26 bp uintptr // for GOEXPERIMENT=framepointer
}

```

g结构体

g结构体用于代表一个goroutine，该结构体保存了goroutine的所有信息，包括栈，gobuf结构体和其它的一些状态信息：

```

1 // 前文所说的g结构体, 它代表了一个goroutine
2 type g struct {
3     // Stack parameters.
4     // stack describes the actual stack memory: [stack.lo, stack.hi].
5     // stackguard0 is the stack pointer compared in the Go stack growth
6     // prologue.
7     // It is stack.lo+StackGuard normally, but can be StackPreempt to
8     // trigger a preemption.
9     // stackguard1 is the stack pointer compared in the C stack growth
10    // prologue.
11    // It is stack.lo+StackGuard on g0 and gsignal stacks.
12    // It is ~0 on other goroutine stacks, to trigger a call to morestackc
13    // (and crash).
14
15    // 记录该goroutine使用的栈
16    stack stack // offset known to runtime/cgo
17    // 下面两个成员用于栈溢出检查, 实现栈的自动伸缩, 抢占调度也会用到stackguard0
18    stackguard0 uintptr // offset known to liblink
19    stackguard1 uintptr // offset known to liblink
20    .....
21
22    // 此goroutine正在被哪个工作线程执行
23    m *m // current m; offset known to arm liblink
24    // 保存调度信息, 主要是几个寄存器的值
25    sched gobuf
}

```

```

23
24     .....
25     // schedlink字段指向全局运行队列中的下一个g,
26     //所有位于全局运行队列中的g形成一个链表
27     schedlink      guintptr
28
29     .....
30     // 抢占调度标志, 如果需要抢占调度, 设置preempt为true
31     preempt        bool       // preemption signal, duplicates stackguard0
32 = stackpreempt
33     .....
34 }

```

m结构体

m结构体用来代表工作线程, 它保存了m自身使用的栈信息, 当前正在运行的goroutine以及与m绑定的p等信息, 详见下面定义中的注释:

```

1 type m struct {
2     // g0主要用来记录工作线程使用的栈信息, 在执行调度代码时需要使用这个栈
3     // 执行用户goroutine代码时, 使用用户goroutine自己的栈, 调度时会发生栈的切换
4     g0      *g      // goroutine with scheduling stack
5     // 通过TLS实现m结构体对象与工作线程之间的绑定
6     tls     [6]uintptr // thread-local storage (for x86 extern
7 register)
8     mstartfn    func()
9     // 指向工作线程正在运行的goroutine的g结构体对象
10    curg       *g      // current running goroutine
11
12    // 记录与当前工作线程绑定的p结构体对象
13    p         puintptr // attached p for executing go code (nil if not
14 executing go code)
15    nextp     puintptr
16    oldp     puintptr // the p that was attached before executing a
17 syscall
18
19    // spinning状态: 表示当前工作线程正在试图从其它工作线程的本地运行队列偷取
20 goroutine
21    spinning   bool // m is out of work and is actively looking for
22 work
23    blocked   bool // m is blocked on a note
24
25    // 没有goroutine需要运行时, 工作线程睡眠在这个park成员上,
26    // 其它线程通过这个park唤醒该工作线程
27    park      note
28    // 记录所有工作线程的一个链表
29    alllink   *m // on allm
30    schedlink guintptr
31    // Linux平台thread的值就是操作系统线程ID
32
33 }

```

```
29     thread      uintptr // thread handle
30     freelist    *m       // on sched.freem
31     .....
32 }
33 }
```

p结构体

p结构体用于保存工作线程执行go代码时所必需的资源，比如goroutine的运行队列，内存分配用到的缓存等等。

```
1 type p struct {
2     lock mutex
3     status      uint32 // one of pidle/prunning/...
4     link        puintptr
5     schedtick   uint32   // incremented on every scheduler call
6     syscalltick uint32   // incremented on every system call
7     sysmontick  sysmontick // last tick observed by sysmon
8     m           muintptr // back-link to associated m (nil if idle)
9     .....
10
11     // Queue of runnable goroutines. Accessed without lock.
12     //本地goroutine运行队列
13     runqhead uint32 // 队列头
14     runqtail uint32 // 队列尾
15     runq      [256]guintptr //使用数组实现的循环队列
16     // runnext, if non-nil, is a runnable G that was ready'd by
17     // the current G and should be run next instead of what's in
18     // runq if there's time remaining in the running G's time
19     // slice. It will inherit the time left in the current time
20     // slice. If a set of goroutines is locked in a
21     // communicate-and-wait pattern, this schedules that set as a
22     // unit and eliminates the (potentially large) scheduling
23     // latency that otherwise arises from adding the ready'd
24     // goroutines to the end of the run queue.
25     runnext guintptr
26     // Available G's (status == Gdead)
27     gFree struct {
28         gList
29         n int32
30     }
31     .....
32 }
33 }
```

schedt结构体

schedt结构体用来保存调度器的状态信息和goroutine的全局运行队列：

```
1 type schedt struct {
```

```

2 // accessed atomically. keep at top to ensure alignment on 32-bit
systems.

3 goidgen uint64
4 lastpoll uint64
5 lock mutex
6
7 // When increasing nmidle, nmiddlelocked, nmsys, or nmfreed, be
8 // sure to call checkdead().
9
10 // 由空闲的工作线程组成链表
11 midle      muintptr // idle m's waiting for work
12
13 // 空闲的工作线程的数量
14 nmidle     int32    // number of idle m's waiting for work
15 nmiddlelocked int32  // number of locked m's waiting for work
16 mnnext     int64    // number of m's that have been created and next
M ID
17 // 最多只能创建maxmcount个工作线程
18 maxmcount   int32    // maximum number of m's allowed (or die)
19 nmsys       int32    // number of system m's not counted for deadlock
20 nmfreed     int64    // cumulative number of freed m's
21 ngsys uint32 // number of system goroutines; updated atomically
22
23 // 由空闲的p结构体对象组成的链表
24 pidle      puintptr // idle p's
25
26 // 空闲的p结构体对象的数量
27 npidle     uint32
28 nmspinning uint32 // See "Worker thread parking/unparking" comment in
proc.go.

29 // Global runnable queue.
30
31 // goroutine全局运行队列
32 runq      gQueue
33 runqsize int32
34
35 .....
36
37 // Global cache of dead G's.
38 // gFree是所有已经退出的goroutine对应的g结构体对象组成的链表
39 // 用于缓存g结构体对象，避免每次创建goroutine时都重新分配内存
40 gFree struct {
41     lock      mutex
42     stack     gList // Gs with stacks
43     noStack  gList // Gs without stacks
44     n        int32
45 }
46
47 .....
48 }

```

重要的全局变量

```

1 allgs     []*g    // 保存所有的g
2 allm      *m     // 所有的m构成的一个链表，包括下面的m0

```

```
3 allp      []*p    // 保存所有的p, len(allp) == gomaxprocs
4 ncpu       int32   // 系统中cpu核的数量, 程序启动时由runtime代码初始化
5 gomaxprocs int32   // p的最大值, 默认等于ncpu, 但可以通过GOMAXPROCS修改
6 sched     *schedt  // 调度器结构体对象, 记录了调度器的工作状态
7 m0  m      *m      // 代表进程的主线程
8 g0  g      *g      // m0的g0, 也就是m0.g0 = &g0
```

在程序初始化时, 这些全变量都会被初始化为0值, 指针会被初始化为nil指针, 切片初始化为nil切片, int被初始化为数字0, 结构体的所有成员变量按其本类型初始化为其类型的0值。所以程序刚启动时allgs, allm和allp都不包含任何g,m和p。

Redis基础

Redis中的数据结构

原文地址 [Redis 中的数据结构](#)

1. 底层数据结构, 与Redis Value Type之间的关系

对于Redis的使用者来说, Redis作为Key–Value型的内存数据库, 其Value有多种类型.

- String
- Hash
- List
- Set
- ZSet

这些Value的类型, 只是”Redis的用户认为的, Value存储数据的方式”. 而在具体实现上, 各个Type的Value到底如何存储, 这对于Redis的使用者来说是不公开的.

举个栗子: 使用下面的命令创建一个Key–Value

```
1 SET "Hello" "World"
```

对于Redis的使用者来说, Hello这个Key, 对应的Value是String类型, 其值为五个ASCII字符组成的二进制数据. 但具体在底层实现上, 这五个字节是如何存储的, 是不对用户公开的. 即, Value的Type, 只是表象, 具体数据在内存中以何种数据结构存放, 这对于用户来说是不必要的了解的.

Redis对使用者暴露了五种 Value Type, 其底层实现的数据结构有8种, 分别是:

- SDS – simple dynamic string – 支持自动动态扩容的字节数组
- list – 平平无奇的链表
- dict – 使用双哈希表实现的, 支持平滑扩容的字典
- zskiplist – 附加了后向指针的跳跃表
- intset – 用于存储整数数值集合的自有结构
- ziplist – 一种实现上类似于TLV, 但比TLV复杂的, 用于存储任意数据的有序序列的数据结构
- quicklist – 一种以ziplist作为结点的双链表结构, 实现的非常苟

- zipmap – 一种用于在小规模场合使用的轻量级字典结构

而衔接”底层数据结构”与”Value Type”的桥梁的, 则是Redis实现的另外一种数据结构: `redisObject`. Redis中的Key与Value在表层都是一个 `redisObject` 实例, 故该结构有所谓的”类型”, 即是 `ValueType`. 对于每一种 `Value Type` 类型的 `redisObject`, 其底层至少支持两种不同的底层数据结构来实现. 以应对在不同的应用场景中, Redis的运行效率, 或内存占用.

2. 底层数据结构

2.1 SDS – simple dynamic string

这是一种用于存储二进制数据的一种结构, 具有动态扩容的特点. 其实现位于src/sds.h 与src/sds.c中, 其关键定义如下:

```

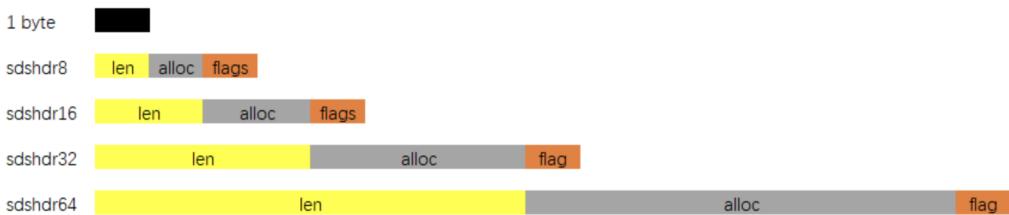
1  typedef char *sds;
2  /* Note: sdshdr5 is never used, we just access the flags byte directly.
3   * However is here to document the layout of type 5 SDS strings. */
4  struct __attribute__((__packed__)) sdshdr5 {
5      unsigned char flags; /* 3 lsb of type, and 5 msb of string length */
6      char buf[];
7  };
8  struct __attribute__((__packed__)) sdshdr8 {
9      uint8_t len; /* used */
10     uint8_t alloc; /* excluding the header and null terminator */
11     unsigned char flags; /* 3 lsb of type, 5 unused bits */
12     char buf[];
13 };
14 struct __attribute__((__packed__)) sdshdr16 {
15     uint16_t len; /* used */
16     uint16_t alloc; /* excluding the header and null terminator */
17     unsigned char flags; /* 3 lsb of type, 5 unused bits */
18     char buf[];
19 };
20 struct __attribute__((__packed__)) sdshdr32 {
21     uint32_t len; /* used */
22     uint32_t alloc; /* excluding the header and null terminator */
23     unsigned char flags; /* 3 lsb of type, 5 unused bits */
24     char buf[];
25 };
26 struct __attribute__((__packed__)) sdshdr64 {
27     uint64_t len; /* used */
28     uint64_t alloc; /* excluding the header and null terminator */
29     unsigned char flags; /* 3 lsb of type, 5 unused bits */
30     char buf[];
31 };
32 
```

SDS的总体概览如下图:



其中sdshdr是头部, buf是真实存储用户数据的地方. 另外注意, 从命名上能看出来, 这个数据结构除了能存储二进制数据, 显然是用于设计作为字符串使用的, 所以在buf中, 用户数据后总跟着一个\0. 即图中 "数据" + "\0" 是为所谓的buf

SDS有五种不同的头部. 其中sdshdr5实际并未使用到. 所以实际上有四种不同的头部, 分别如下:



- len分别以uint8, uint16, uint32, uint64表示用户数据的长度(不包括末尾的\0)
- alloc分别以uint8, uint16, uint32, uint64表示整个SDS, 除过头部与末尾的\0, 剩余的字节数.
- flag始终为一字节, 以低三位标示着头部的类型, 高5位未使用.

当在程序中持有一个SDS实例时, 直接持有的是数据区的头指针, 这样做的用意是: 通过这个指针, 向前偏一个字节, 就能取到flag, 通过判断flag低三位的值, 能迅速判断: 头部的类型, 已用字节数, 总字节数, 剩余字节数. 这也是为什么sds类型即是char *指针类型别名的原因.

创建一个SDS实例有三个接口, 分别是:

```
1 // 创建一个不含数据的sds:  
2 // 头部 3字节 sdshdr8  
3 // 数据区 0字节  
4 // 末尾 \0 占一字符  
5 sds sdseempty(void);  
6 // 带数据创建一个sds:  
7 // 头部 按initlen的值, 选择最小的头部类型  
8 // 数据区 从入参指针init处开始, 拷贝initlen个字节  
9 // 末尾 \0 占一字符  
10 sds sdsnewlen(const void *init, size_t initlen);  
11 // 带数据创建一个sds:  
12 // 头部 按strlen(init)的值, 选择最小的头部类型  
13 // 数据区 入参指向的字符串中的所有字符, 不包括末尾 \0  
14 // 末尾 \0 占一字符  
15 sds sdsnew(const char *init);
```

- 所有创建sds实例的接口, 都不会额外分配预留内存空间

- `sdsnewlen` 用于带二进制数据创建sds实例, `sdsnew`用于带字符串创建sds实例. 接口返回的sds可以直接传入libc中的字符串输出函数中进行操作, 由于无论其中存储的是用户的二进制数据, 还是字符串, 其末尾都带一个\0, 所以至少调用libc中的字符串输出函数是安全的.

在对SDS中的数据进行修改时, 若剩余空间不足, 会调用`sdsMakeRoomFor`函数用于扩容空间, 这是一个很低级的API, 通常情况下不应当由SDS的使用者直接调用. 其实现中核心的几行如下:

```

1 sds sdsMakeRoomFor(sds s, size_t addlen) {
2     ...
3     /* Return ASAP if there is enough space left. */
4     if (avail >= addlen) return s;
5     len = sdslen(s);
6     sh = (char*)s-sdsHdrSize(oldtype);
7     newlen = (len+addlen);
8     if (newlen < SDS_MAX_PREALLOC)
9         newlen *= 2;
10    else
11        newlen += SDS_MAX_PREALLOC;
12    ...
13
14 }
```

可以看到, 在扩充空间时

- 先保证至少有`addlen`可用
- 然后再进一步扩充, 在总体占用空间不超过阈值 `SDS_MAC_PREALLOC` 时, 申请空间再翻一倍. 若总体空间已经超过了阈值, 则步进增长 `SDS_MAC_PREALLOC`. 这个阈值的默认值为 `1024 * 1024`

SDS也提供了接口用于移除所有未使用的内存空间. `sdsRemoveFreeSpace`, 该接口没有间接的被任何SDS其它接口调用, 即默认情况下, SDS不会自动回收预留空间. 在SDS的使用者需要节省内存时, 由使用者自行调用:

```

1 sds sdsRemoveFreeSpace(sds s);
```

总结:

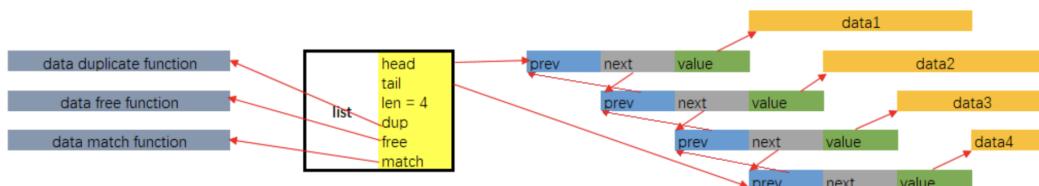
- SDS除了是某些Value Type的底层实现, 也被大量使用在Redis内部, 用于替代C-Style字符串. 所以默认的创建SDS实例接口, 不分配额外的预留空间. 因为多数字符串在程序运行期间是不变的. 而对于变更数据区的API, 其内部则是调用了`sdsMakeRoomFor`, 每一次扩充空间, 都会预留大量的空间. 这样做的考量是: 如果一个SDS实例中的数据被变更了, 那么很有可能会在后续发生多次变更.
- SDS的API内部不负责清除未使用的闲置内存空间, 因为内部API无法判断这样做的合适时机. 即便是在操作数据区的时候导致数据区占用内存减少时, 内部API也不会清除闲置内存空间. 清除闲置内存空间责任应当由SDS的使用者自行担当.
- 用SDS替代C-Style字符串时, 由于其头部额外存储了数据区的长度信息, 所以字符串的求长操作时间复杂度为O(1)

2.2 list

这是普通的链表实现, 链表结点不直接持有数据, 而是通过void *指针来间接的指向数据. 其实现位于 src/adlist.h与src/adlist.c中, 关键定义如下:

```
1 typedef struct listNode {
2     struct listNode *prev;
3     struct listNode *next;
4     void *value;
5 } listNode;
6 typedef struct listIter {
7     listNode *next;
8     int direction;
9 } listIter;
10 typedef struct list {
11     listNode *head;
12     listNode *tail;
13     void *(*dup)(void *ptr);
14     void (*free)(void *ptr);
15     int (*match)(void *ptr, void *key);
16     unsigned long len;
17 } list;
```

其内存布局如下图所示:



这是一个平平无奇的链表的实现. list在Redis除了作为一些Value Type的底层实现外, 还广泛用于Redis的其它功能实现中, 作为一种数据结构工具使用. 在list的实现中, 除了基本的链表定义外, 还额外增加了:

- 迭代器 `listIter` 的定义, 与相关接口的实现.
- 由于list中的链表结点本身并不直接持有数据, 而是通过value字段, 以void *指针的形式间接持有, 所以数据的生命周期并不完全与链表及其结点一致. 这给了list的使用者相当大的灵活性. 比如可以多个结点持有同一份数据的地址. 但与此同时, 在对链表进行销毁, 结点复制以及查找匹配时, 就需要list的使用者将相关的函数指针赋值于 `list.dup`, `list.free`, `list.match`字段.

2.3 dict

dict是Redis底层数据结构中实现最为复杂的一个数据结构, 其功能类似于C++标准库中的`std::unordered_map`, 其实现位于 src/dict.h 与 src/dict.c中, 其关键定义如下:

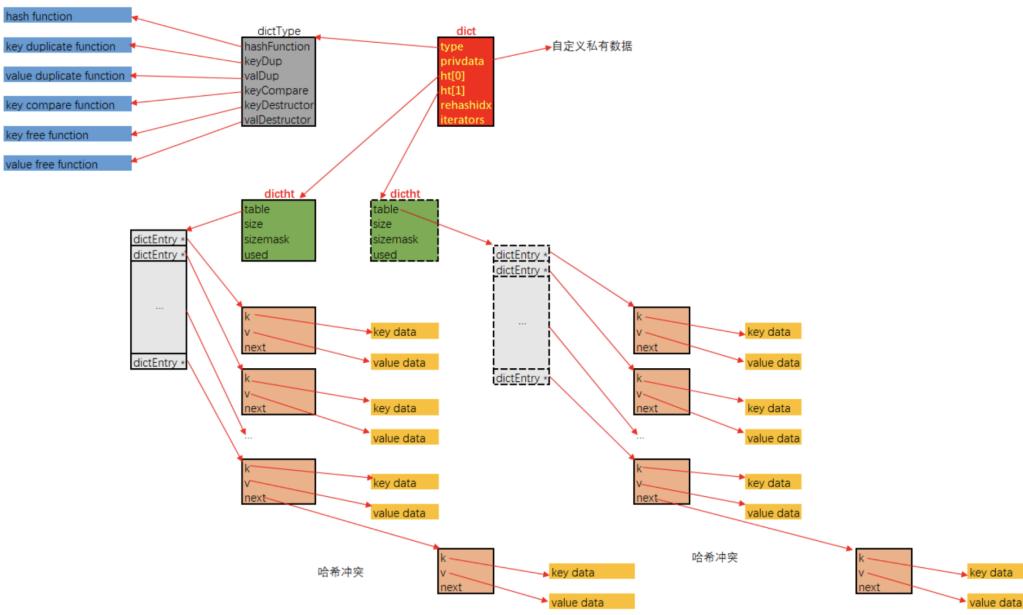
```
1 typedef struct dictEntry {
2     void *key;
3     union {
4         void *val;
```

```

5         uint64_t u64;
6         int64_t s64;
7         double d;
8     } v;
9     struct dictEntry *next;
10 } dictEntry;
11
12 typedef struct dictType {
13     uint64_t (*hashFunction)(const void *key);
14     void *(*keyDup)(void *privdata, const void *key);
15     void *(*valDup)(void *privdata, const void *obj);
16     int (*keyCompare)(void *privdata, const void *key1, const void *key2);
17     void (*keyDestructor)(void *privdata, void *key);
18     void (*valDestructor)(void *privdata, void *obj);
19 } dictType;
20 /* This is our hash table structure. Every dictionary has two of this as
we
21 * implement incremental rehashing, for the old to the new table. */
22
23 typedef struct dictht {
24     dictEntry **table;
25     unsigned long size;
26     unsigned long sizemask;
27     unsigned long used;
28 } dictht;
29
30 typedef struct dict {
31     dictType *type;
32     void *privdata;
33     dictht ht[2];
34     long rehashidx; /* rehashing not in progress if rehashidx == -1 */
35     unsigned long iterators; /* number of iterators currently running */
36 } dict;
37
38 /* If safe is set to 1 this is a safe iterator, that means, you can call
39 * dictAdd, dictFind, and other functions against the dictionary even
while
40 * iterating. Otherwise it is a non safe iterator, and only dictNext()
41 * should be called while iterating. */
42
43 typedef struct dictIterator {
44     dict *d;
45     long index;
46     int table, safe;
47     dictEntry *entry, *nextEntry;
48     /* unsafe iterator fingerprint for misuse detection. */
49     long long fingerprint;
50 } dictIterator;

```

其内存布局如下所示:



- dict 中存储的键值对，是通过 dictEntry 这个结构间接持有的，k 通过指针间接持有键，v 通过指针间接持有值。注意，若值是整数值的话，是直接存储在 v 字段中的，而不是间接持有。同时 next 指针用于指向，在 bucket 索引值冲突时，以链式方式解决冲突，指向同索引的下一个 dictEntry 结构。
- 传统的哈希表实现，是一块连续空间的顺序表，表中元素即是结点。在 dictht.table 中，结点本身是散布在内存中的，顺序表中存储的是 dictEntry 的指针
- 哈希表即是 dictht 结构，其通过 table 字段间接的持有顺序表形式的 bucket，bucket 的容量存储在 size 字段中，为了加速将散列值转化为 bucket 中的数组索引，引入了 sizemask 字段，计算指定键在哈希表中的索引时，执行的操作类似于 dict->type->hashFunction(键) & dict->ht[x].sizemask。从这里也可以看出来，bucket 的容量适宜于为 2 的幂次，这样计算出的索引值能覆盖到所有 bucket 索引位。
- dict 即为字典。其中 type 字段中存储的是本字典使用到的各种函数指针，包括散列函数，键与值的复制函数，释放函数，以及键的比较函数。privdata 是用于存储用户自定义数据。这样，字典的使用者可以最大化的自定义字典的实现，通过自定义各种函数实现，以及可以附带私有数据，保证了字典有很大的调优空间。
- 字典为了支持平滑扩容，定义了 ht[2] 这个数组字段。其用意是这样的：
 - 一般情况下，字典 dict 仅持有一个哈希表 dictht 的实例，即整个字典由一个 bucket 实现。
 - 随着插入操作，bucket 中出现冲突的概率会越来越大，当字典中存储的结点数目，与 bucket 数组长度的比值达到一个阈值(1:1)时，字典为了缓解性能下降，就需要扩容。
 - 扩容的操作是平滑的，即在扩容时，字典会持有两个 dictht 的实例，ht[0] 指向旧哈希表，ht[1] 指向扩容后的新哈希表。平滑扩容的重点在于两个策略：
 - 后续每一次的插入，替换，查找操作，都插入到 ht[1] 指向的哈希表中
 - 每一次插入，替换，查找操作执行时，会将旧表 ht[0] 中的一个 bucket 索引位持有的结点链表，迁移到 ht[1] 中去。迁移的进度保存在 rehashidx 这个字段中。在旧表中由于冲突而被链接在同一索引位上的结点，迁移到新表后，可能会散布在多个新表索引中去。
 - 当迁移完成后，ht[0] 指向的旧表会被释放，之后会将新表的持有权转交给 ht[0]，再重置 ht[1] 指向 NULL。
- 这种平滑扩容的优点有两个：
 - 平滑扩容过程中，所有结点的实际数据，即 dict->ht[0]->table[rehashindex]->k 与 dict->ht[0]->table[rehashindex]->v 分别指向的实际数据，内存地址都不会变化。没有

- 发生键数据与值数据的拷贝或移动, 扩容整个过程仅是各种指针的操作. 速度非常快
- 扩容操作是步进式的, 这保证任何一次插入操作都是顺畅的, dict的使用者是无感知的. 若扩容是一次性的, 当新旧bucket容量特别大时, 迁移所有结点必然会导致耗时陡增.

除了字典本身的实现外, 其中还顺带实现了一个迭代器, 这个迭代器中有字段safe以标示该迭代器是”安全迭代器”还是”非安全迭代器”, 所谓的安全与否, 指的是这种场景: 设想在运行迭代器的过程中, 字典正处于平滑扩容的过程中. 在平滑扩容的过程中时, 旧表一个索引位上的, 由冲突而链起来的多个结点, 迁移到新表后, 可能会散布到新表的多个索引位上. 且新的索引位的值可能比旧的索引位要低.

遍历操作的重点是, 保证在迭代器遍历操作开始时, 字典中持有的所有结点, 都会被遍历到. 而若在遍历过程中, 一个未遍历的结点, 从旧表迁移到新表后, 索引值减小了, 那么就可能会导致这个结点在遍历过程中被遗漏.

所以, 所谓的”安全”迭代器, 其在内部实现时: 在迭代过程中, 若字典正处于平滑扩容过程, 则暂停结点迁移, 直至迭代器运行结束. 这样虽然不能保证在迭代过程中插入的结点会被遍历到, 但至少保证在迭代起始时, 字典中持有的所有结点都会被遍历到.

这也是为什么dict结构中有一个iterators字段的原因: 该字段记录了运行于该字典上的安全迭代器的数目. 若该数目不为0, 字典是不会继续进行结点迁移平滑扩容的.

下面是字典的扩容操作中的核心代码, 我们以插入操作引起的扩容为例:

先是插入操作的外部逻辑:

1. 如果插入时, 字典正处于平滑扩容过程中, 那么无论本次插入是否成功, 先迁移一个bucket索引中的结点至新表
2. 在计算新插入结点键的bucket索引值时, 内部会探测哈希表是否需要扩容(若当前不在平滑扩容过程中)

```

1 int dictAdd(dict *d, void *key, void *val)
2 {
3     dictEntry *entry = dictAddRaw(d, key, NULL);           // 调用dictAddRaw
4     if (!entry) return DICT_ERR;
5     dictSetVal(d, entry, val);
6     return DICT_OK;
7 }
8
9 dictEntry *dictAddRaw(dict *d, void *key, dictEntry **existing)
10 {
11     long index;
12     dictEntry *entry;
13     dictht *ht;
14
15     if (dictIsRehashing(d)) _dictRehashStep(d); // 若在平滑扩容过程中, 先步进
16     // 迁移一个bucket索引
17
18     /* Get the index of the new element, or -1 if
19      * the element already exists. */
20
21     // 在计算键在bucket中的索引值时, 内部会检查是否需要扩容
22     if ((index = _dictKeyIndex(d, key, dictHashKey(d, key), existing)) ==
23         -1)
24         return NULL;
25
26     /* Allocate the memory and store the new entry.
27      * Insert the element in top, with the assumption that in a database
28      * system it is more likely that recently added entries are accessed

```

```

28     * more frequently. */
29     ht = dictIsRehashing(d) ? &d->ht[1] : &d->ht[0];
30     entry = zmalloc(sizeof(*entry));
31     entry->next = ht->table[index];
32     ht->table[index] = entry;
33     ht->used++;
34     /* Set the hash entry fields. */
35     dictSetKey(d, entry, key);
36     return entry;
37 }
38 }
```

下面是计算bucket索引值的函数, 内部会探测该哈希表是否需要扩容, 如果需要扩容(结点数目与bucket数组长度比例达到1:1), 就使字典进入平滑扩容过程:

```

1 static long _dictKeyIndex(dict *d, const void *key, uint64_t hash,
2 dictEntry **existing)
3 {
4     unsigned long idx, table;
5     dictEntry *he;
6     if (existing) *existing = NULL;
7     /* Expand the hash table if needed */
8     if (_dictExpandIfNeeded(d) == DICT_ERR) // 探测是否需要扩容, 如果需要, 则
开始扩容
9         return -1;
10    for (table = 0; table <= 1; table++) {
11        idx = hash & d->ht[table].sizemask;
12        /* Search if this slot does not already contain the given key */
13        he = d->ht[table].table[idx];
14        while(he) {
15            if (key==he->key || dictCompareKeys(d, key, he->key)) {
16                if (existing) *existing = he;
17                return -1;
18            }
19            he = he->next;
20        }
21        if (!dictIsRehashing(d)) break;
22    }
23    return idx;
24 }
25 /* Expand the hash table if needed */
26 static int _dictExpandIfNeeded(dict *d)
27 {
28     /* Incremental rehashing already in progress. Return. */
29     if (dictIsRehashing(d)) return DICT_OK; // 如果正在扩容过程中, 则什么也不
做
30     /* If the hash table is empty expand it to the initial size. */
31     // 若字典中本无元素, 则初始化字典, 初始化时的bucket数组长度为4
32     if (d->ht[0].size == 0) return dictExpand(d, DICT_HT_INITIAL_SIZE);
33     /* If we reached the 1:1 ratio, and we are allowed to resize the hash
 * table (global setting) or we should avoid it but the ratio between
34     * table (global setting) or we should avoid it but the ratio between
35     * table (global setting) or we should avoid it but the ratio between
36     * table (global setting) or we should avoid it but the ratio between
37     * table (global setting) or we should avoid it but the ratio between
38 }
```

```

38     * elements/buckets is over the "safe" threshold, we resize doubling
39     * the number of buckets. */
40 // 若字典中元素的个数与bucket数组长度比值大于1:1时，则调用dictExpand进入平滑
41 // 扩容状态
42     if (d->ht[0].used >= d->ht[0].size &&
43         (dict_can_resize ||
44          d->ht[0].used/d->ht[0].size > dict_force_resize_ratio))
45     {
46         return dictExpand(d, d->ht[0].used*2);
47     }
48     return DICT_OK;
49 }
50 int dictExpand(dict *d, unsigned long size)
51 {
52     dictht n; /* the new hash table */ // 新建一个dictht结构
53     unsigned long realsize = _dictNextPower(size);
54     /* the size is invalid if it is smaller than the number of
55      * elements already inside the hash table */
56     if (dictIsRehashing(d) || d->ht[0].used > size)
57         return DICT_ERR;
58     /* Rehashing to the same table size is not useful. */
59     if (realsize == d->ht[0].size) return DICT_ERR;
60     /* Allocate the new hash table and initialize all pointers to NULL */
61     n.size = realsize;
62     n.sizemask = realsize-1;
63     n.table = zcalloc(realsize*sizeof(dictEntry*)); // 初始化dictht下的
64     // table, 即bucket数组
65     n.used = 0;
66     /* Is this the first initialization? If so it's not really a rehashing
67      * we just set the first hash table so that it can accept keys. */
68     // 若是新字典初始化，直接把dictht结构挂在ht[0]中
69     if (d->ht[0].table == NULL) {
70         d->ht[0] = n;
71         return DICT_OK;
72     }
73     // 否则，把新dictht结构挂在ht[1]中，并开启平滑扩容(置rehashidx为0，字典处于
74     // 非扩容状态时，该字段值为-1)
75     /* Prepare a second hash table for incremental rehashing */
76     d->ht[1] = n;
77     d->rehashidx = 0;
78     return DICT_OK;
79 }

```

下面是平滑扩容的实现:

```

1 static void _dictRehashStep(dict *d) {
2     // 若字典上还运行着安全迭代器，则不迁移结点
3     // 否则每次迁移一个旧bucket索引上的所有结点
4     if (d->iterators == 0) dictRehash(d,1);
5 }

```

```

5   int dictRehash(dict *d, int n) {
6       int empty_visits = n*10; /* Max number of empty buckets to visit. */
7       if (!dictIsRehashing(d)) return 0;
8       while(n-- && d->ht[0].used != 0) {
9           dictEntry *de, *nextde;
10          /* Note that rehashidx can't overflow as we are sure there are
11             more
12             * elements because ht[0].used != 0 */
13          assert(d->ht[0].size > (unsigned long)d->rehashidx);
14          // 在旧bucket中，找到下一个非空的索引位
15          while(d->ht[0].table[d->rehashidx] == NULL) {
16              d->rehashidx++;
17              if (--empty_visits == 0) return 1;
18          }
19          // 取出该索引位上的结点链表
20          de = d->ht[0].table[d->rehashidx];
21          /* Move all the keys in this bucket from the old to the new hash
22             HT */
23          // 把所有结点迁移到新bucket中去
24          while(de) {
25              uint64_t h;
26              nextde = de->next;
27              /* Get the index in the new hash table */
28              h = dictHashKey(d, de->key) & d->ht[1].sizemask;
29              de->next = d->ht[1].table[h];
30              d->ht[1].table[h] = de;
31              d->ht[0].used--;
32              d->ht[1].used++;
33              de = nextde;
34          }
35          d->ht[0].table[d->rehashidx] = NULL;
36          d->rehashidx++;
37      }
38      /* Check if we already rehashed the whole table... */
39      // 检查是否旧表中的所有结点都被迁移到了新表
40      // 如果是，则置先释放原旧bucket数组，再置ht[1]为ht[0]
41      // 最后再置rehashidx=-1，以示字典不处于平滑扩容状态
42      if (d->ht[0].used == 0) {
43          zfree(d->ht[0].table);
44          d->ht[0] = d->ht[1];
45          _dictReset(&d->ht[1]);
46          d->rehashidx = -1;
47          return 0;
48      }
49      /* More to rehash... */
50      return 1;
51  }

```

总结:

字典的实现很复杂,主要是实现了平滑扩容逻辑 用户数据均是以指针形式间接由 dictEntry结构持有,故在平滑扩容过程中,不涉及用户数据的拷贝 有安全迭代器可用, 安全迭代器保证, 在迭代起始时, 字典中的所有结点, 都会被迭代到, 即使在迭代过程中对字典有插入操作 字典内部使用的默认散列函数其实也非常有讲究, 不过限于篇幅, 这里不展开讲. 并且字典的实现给了使用者非常大的灵活性(dictType结构与dict.privdata字段), 对于一些特定场合使用的键数据, 用户可以自行选择更高效更特定化的散列函数

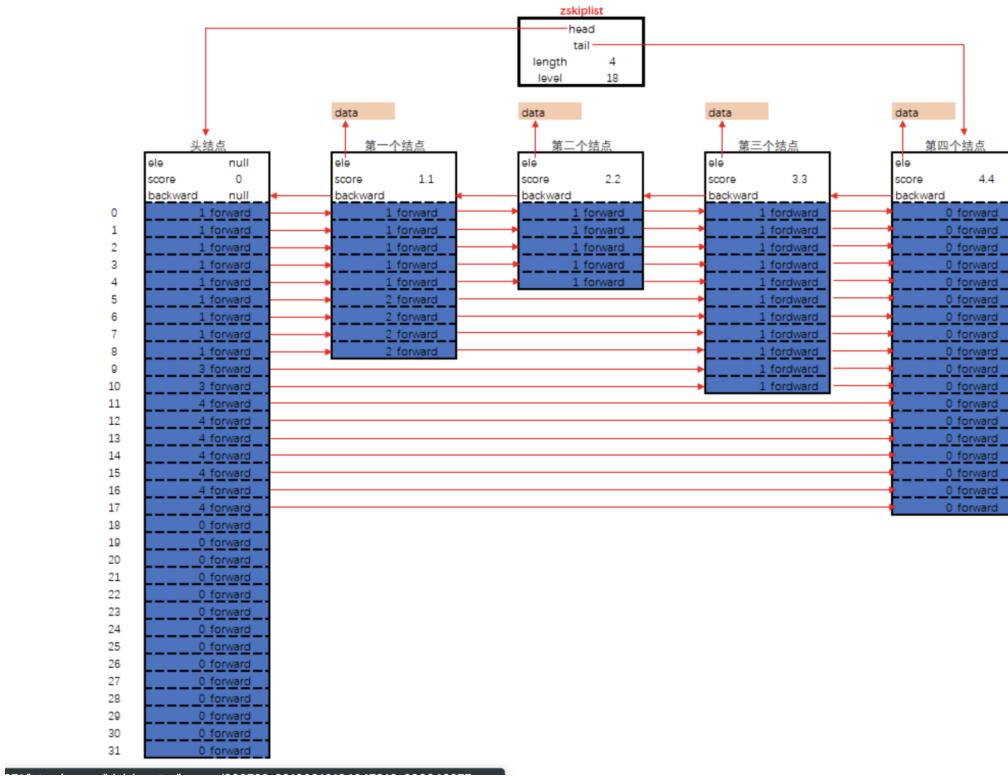
2.4 zskiplist

zskiplist是Redis实现的一种特殊的跳跃表. 跳跃表是一种基于线性表实现简单的搜索结构, 其最大的特点就是: 实现简单, 性能能逼近各种搜索树结构. 血统纯正的跳跃表的介绍在维基百科中即可查阅. 在Redis中, 在原版跳跃表的基础上, 进行了一些小改动, 即是现在要介绍的zskiplist结构.

其定义在src/server.h中, 如下:

```
1  /* ZSETs use a specialized version of Skiplists */
2  typedef struct zskiplistNode {
3      sds ele;
4      double score;
5      struct zskiplistNode *backward;
6      struct zskiplistLevel {
7          struct zskiplistNode *forward;
8          unsigned int span;
9      } level[];
10 } zskiplistNode;
11 typedef struct zskiplist {
12     struct zskiplistNode *header, *tail;
13     unsigned long length;
14     int level;
15 } zskiplist;
```

其内存布局如下图:



zskiplist的核心设计要点为:

1. 头结点不持有任何数据, 且其level[]的长度为32
2. 每个结点, 除了持有数据的ele字段, 还有一个字段score, 其标示着结点的得分, 结点之间凭借得分来判断先后顺序, 跳跃表中的结点按结点的得分升序排列.
3. 每个结点持有一个backward指针, 这是原版跳跃表中所没有的. 该指针指向结点的前一个紧邻结点.
4. 每个结点中最多持有32个zskiplistLevel结构. 实际数量在结点创建时, 按幂次定律随机生成(不超过32). 每个zskiplistLevel中有两个字段.
5. forward字段指向比自己得分高的某个结点(不一定是紧邻的), 并且, 若当前zskiplistNode实例在level[]中的索引为X, 则其forward字段指向的结点, 其level[]字段的容量至少是X+1. 这也是上图中, 为什么forward指针总是画的水平的原因.
6. span字段代表forward字段指向的结点, 距离当前结点的距离. 紧邻的两个结点之间的距离定义为1.
7. zskiplist中持有字段level, 用以记录所有结点(除过头结点外), level[]数组最长的长度. 跳跃表主要用于, 在给定一个分值的情况下, 查找与该分值最接近的结点. 搜索时, 伪代码如下:

```

1 int level = zskiplist->level - 1;
2 zskiplistNode p = zskiplist->head;
3 while(1 && p)
4 {
5     zskiplistNode q = (p->level)[level]->forward;
6     if(q->score > 分值)
7     {
8         if(level > 0)
9         {
10             level--;
11         }
12     }

```

```

13     else
14     {
15         return :
16             q为整个跳跃表中，分值大于指定分值的第一个结点
17             q->backward为整个跳跃表中，分值小于或等于指定分值的最后一个结点
18     }
19 }
20 else
21 {
22     p = q;
23 }
24 }
```

跳跃表的实现比较简单, 最复杂的操作即是插入与删除结点, 需要仔细处理邻近结点的所有level[]中的所有zskiplistLevel结点中的forward与span的值的变更.

另外, 关于新创建的结点, 其 level[] 数组长度的随机算法, 在接口zsllInsert的实现中, 核心代码片断如下:

```

1 zskiplistNode *zslInsert(zskiplist *zsl, double score, sds ele) {
2 //...
3     level = zslRandomLevel();    // 随机生成新结点的, level[]数组的长度
4     if (level > zsl->level) {
5         // 若生成的新结点的level[]数组的长度比当前表中所有结点的level[]的长度都大
6         // 那么头结点中需要新增几个指向该结点的指针
7         // 并刷新ziplist中的level字段
8         for (i = zsl->level; i < level; i++) {
9             rank[i] = 0;
10            update[i] = zsl->header;
11            update[i]->level[i].span = zsl->length;
12        }
13        zsl->level = level;
14    }
15    x = zslCreateNode(level,score,ele); // 创建新结点
16    //... 执行插入操作
17 }
18 // 按幂次定律生成小于32的随机数的函数
19 // 宏 ZSKIPLIST_MAXLEVEL 的定义为32, 宏 ZSKIPLIST_P 被设定为 0.25
20 // 即
21 //      level == 1的概率为 75%
22 //      level == 2的概率为 75% * 25%
23 //      level == 3的概率为 75% * 25% * 25%
24 //      ...
25 //      level == 31的概率为 0.75 * 0.25^30
26 //      而
27 //      level == 32的概率为 0.75 * sum(i = 31 ~ +INF){ 0.25^i }
28 int zslRandomLevel(void) {
29     int level = 1;
30     while ((random()&0xFFFF) < (ZSKIPLIST_P * 0xFFFF))
31         level += 1;
```

```
34     return (level<ZSKIPLIST_MAXLEVEL) ? level : ZSKIPLIST_MAXLEVEL;
35 }
```

2.5 intset

这是一个用于存储有序的整数的数据结构，也底层数据结构中最简单的一个，其定义与实现在src/intset.h与src/intset.c中，关键定义如下：

```
1 typedef struct intset {
2     uint32_t encoding;
3     uint32_t length;
4     int8_t contents[];
5 } intset;
6 #define INTSET_ENC_INT16 (sizeof(int16_t))
7 #define INTSET_ENC_INT32 (sizeof(int32_t))
8 #define INTSET_ENC_INT64 (sizeof(int64_t))
```

intset结构中的encoding的取值有三个，分别是宏INTSET_ENC_INT16, INTSET_ENC_INT32, INTSET_ENC_INT64。length代表其中存储的整数的个数，contents指向实际存储数值的连续内存区域。其内存布局如下图所示：

- intset中各



字段，包括contents中存储的数值，都是以主机序(小端字节序)存储的。这意味着Redis若运行在PPC这样的大端字节序的机器上时，存取数据都会有额外的字节序转换开销

- 当encoding == INTSET_ENC_INT16时，contents中以int16_t的形式存储着数值。类似的，当encoding == INTSET_ENC_INT32时，contents中以int32_t的形式存储着数值。
- 但凡有一个数值元素的值超过了int32_t的取值范围，整个intset都要进行升级，即所有的数值都需要以int64_t的形式存储。显然升级的开销是很大的。
- intset中的数值是以升序排列存储的，插入与删除的复杂度均为O(n)。查找使用二分法，复杂度为O(log_2(n))
- intset的代码实现中，不预留空间，即每一次插入操作都会调用zrealloc接口重新分配内存。每一次删除也会调用zrealloc接口缩减占用的内存。省是省了，但内存操作的时间开销上升了。
- intset的编码方式一经升级，不会再降级。

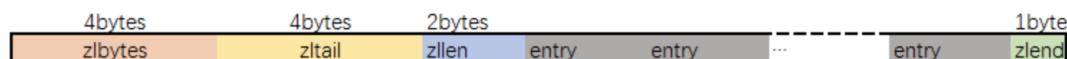
总之，intset适合于如下数据的存储：

- 所有数据都位于一个稳定的取值范围内。比如均位于int16_t或int32_t的取值范围内
- 数据稳定，插入删除操作不频繁。能接受O(lgn)级别的查找开销

2.6 ziplist

ziplist是Redis底层数据结构中, 最苟的一个结构. 它的设计宗旨就是: 省内存, 从牙缝里省内存. 设计思路和TLV一致, 但为了从牙缝里节省内存, 做了很多额外工作.

ziplist的内存布局与intset一样: 就是一块连续的内存空间. 但区域划分比较复杂, 概览如下图:



- 和intset一样, ziplist中的所有值都是以小端序存储的
- zlbytes字段的类型是uint32_t, 这个字段中存储的是整个ziplist所占用的内存的字节数
- zltail字段的类型是uint32_t, 它指的是ziplist中最后一个entry的偏移量. 用于快速定位最后一个entry, 以快速完成pop等操作
- zllen字段的类型是uint16_t, 它指的是整个ziplist中entry的数量. 这个值只占16位, 所以蛋疼的地方就来了: 如果ziplist中entry的数目小于65535, 那么该字段中存储的就是实际entry的值. 若等于或超过65535, 那么该字段的值固定为65535, 但实际数量需要一个个entry的去遍历所有entry才能得到.
- zlend是一个终止字节, 其值为全F, 即0xff. ziplist保证任何情况下, 一个entry的首字节都不会是255

在画图展示entry的内存布局之前, 先讲一下entry中都存储了哪些信息:

- 每个entry中存储了它前一个entry所占用的字节数. 这样支持ziplist反向遍历.
- 每个entry用单独的一块区域, 存储着当前结点的类型: 所谓的类型, 包括当前结点存储的数据是什么(二进制, 还是数值), 如何编码(如果是数值, 数值如何存储, 如果是二进制数据, 二进制数据的长度)
- 最后就是真实的数据了

entry的内存布局如下所示:



`prevlen` 即是”前一个entry所占用的字节数”, 它本身是一个变长字段, 规约如下:

- 若前一个entry占用的字节数小于 254, 则prevlen字段占一字节
- 若前一个entry占用的字节数等于或大于 254, 则prevlen字段占五字节: 第一个字节值为 254, 即0xfe, 另外四个字节, 以uint32_t存储着值.

`encoding` 字段的规约就复杂了许多

- 若数据是二进制数据, 且二进制数据长度小于64字节(不包括64), 那么encoding占一字节. 在这一字节中, 高两位值固定为0, 低六位值以无符号整数的形式存储着二进制数据的长度. 即 00xxxxxxxx, 其中低六位bitxxxxxxxx是用二进制保存的数据长度.
- 若数据是二进制数据, 且二进制数据长度大于或等于64字节, 但小于16384(不包括16384)字节, 那么encoding占用两个字节. 在这两个字节16位中, 第一个字节的高两位固定为01, 剩余的14个位, 以小端序无符号整数的形式存储着二进制数据的长度, 即 01xxxxxxxx, yyyy-yyyy, 其中yyyy-yyyy是高八位, xxxxxxxx是低六位.

- 若数据是二进制数据, 且二进制数据的长度大于或等于16384字节, 但小于 $2^{32}-1$ 字节, 则encoding占用五个字节. 第一个字节是固定值10000000, 剩余四个字节, 按小端序uint32_t的形式存储着二进制数据的长度. 这也是ziplist能存储的二进制数据的最大长度, 超过 $2^{32}-1$ 字节的二进制数据, ziplist无法存储.
 - 若数据是整数值, 则encoding和data的规约如下:
 - 首先, 所有存储数值的entry, 其encoding都仅占用一个字节. 并且最高两位均是11
 - 若数值取值范围位于[0, 12]中, 则encoding和data挤在同一个字节中. 即为1111 0001~1111 1101, 高四位是固定值, 低四位的值从0001至1101, 分别代表 0 ~ 12这十五个数值
 - 若数值取值范围位于[-128, -1] [13, 127]中, 则encoding == 0b 1111 1110. 数值存储在紧邻的下一个字节, 以int8_t形式编码
 - 若数值取值范围位于[-32768, -129] [128, 32767]中, 则encoding == 0b 1100 0000. 数值存储在紧邻的后两个字节中, 以小端序int16_t形式编码
 - 若数值取值范围位于[-8388608, -32769] [32768, 8388607]中, 则encoding == 0b 1111 0000. 数值存储在紧邻的后三个字节中, 以小端序存储, 占用三个字节.
 - 若数值取值范围位于[- 2^{31} , -8388609] [8388608, $2^{31}-1$]中, 则encoding == 0b 1101 0000. 数值存储在紧邻的后四个字节中, 以小端序int32_t形式编码
 - 若数值取值均不在上述范围, 但位于int64_t所能表达的范围内, 则encoding == 0b 1110 0000, 数值存储在紧邻的后八个字节中, 以小端序int64_t形式编码

在大规模数值存储中, ziplist几乎不浪费内存空间, 其苟的程序到达了字节级别, 甚至对于[0, 12]区间的数值, 连data里的那一个字节也要省下来. 显然, ziplist是一种特别节省内存的数据结构, 但它的缺点也十分明显:

- 和intset一样, ziplist也不预留内存空间, 并且在移除结点后, 也是立即缩容, 这代表每次写操作都会进行内存分配操作.
 - ziplist最蛋疼的一个问题是: 结点如果扩容, 导致结点占用的内存增长, 并且超过254字节的话, 可能会导致链式反应: 其后一个结点的entry.prevlen需要从一字节扩容至五字节. 最坏情况下, 第一个结点的扩容, 会导致整个ziplist表中的后续所有结点的entry.prevlen字段扩容. 虽然这个内存重分配的操作依然只会发生一次, 但代码中的时间复杂度是 $O(N)$ 级别, 因为链式扩容只能一步一步的计算. 但这种情况的概率十分的小, 一般情况下链式扩容能连锁反映五六次就很不幸了. 之所以说这是一个蛋疼问题, 是因为, 这样的坏场景下, 其实时间复杂度并不高: 依次计算每个entry新的空间占用, 也就是 $O(N)$, 总体占用计算出来后, 只执行一次内存重分配, 与对应的memmove操作, 就可以了. 蛋疼说的是: 代码特别难写, 难读. 下面放一段处理插入结点时处理链式反应的代码片断, 大家自行感受一下:

```
1 unsigned char *_ziplistInsert(unsigned char *zl, unsigned char *p,
2     unsigned char *s, unsigned int slen) {
3     size_t curlen = intrev32ifbe(ZIPLIST_BYTES(zl)), reqlen;
4     unsigned int prevlensize, prevlen = 0;
5     size_t offset;
6     int nextdiff = 0;
7     unsigned char encoding = 0;
8     long long value = 123456789; /* initialized to avoid warning. Using a
9     value
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
229
230
231
232
233
234
235
236
237
238
239
239
240
241
242
243
244
245
246
247
247
248
249
249
250
251
252
253
254
255
256
256
257
258
259
259
260
261
262
263
264
264
265
266
266
267
268
268
269
269
270
270
271
271
272
272
273
273
274
274
275
275
276
276
277
277
278
278
279
279
280
280
281
281
282
282
283
283
284
284
285
285
286
286
287
287
288
288
289
289
290
290
291
291
292
292
293
293
294
294
295
295
296
296
297
297
298
298
299
299
300
300
301
301
302
302
303
303
304
304
305
305
306
306
307
307
308
308
309
309
310
310
311
311
312
312
313
313
314
314
315
315
316
316
317
317
318
318
319
319
320
320
321
321
322
322
323
323
324
324
325
325
326
326
327
327
328
328
329
329
330
330
331
331
332
332
333
333
334
334
335
335
336
336
337
337
338
338
339
339
340
340
341
341
342
342
343
343
344
344
345
345
346
346
347
347
348
348
349
349
350
350
351
351
352
352
353
353
354
354
355
355
356
356
357
357
358
358
359
359
360
360
361
361
362
362
363
363
364
364
365
365
366
366
367
367
368
368
369
369
370
370
371
371
372
372
373
373
374
374
375
375
376
376
377
377
378
378
379
379
380
380
381
381
382
382
383
383
384
384
385
385
386
386
387
387
388
388
389
389
390
390
391
391
392
392
393
393
394
394
395
395
396
396
397
397
398
398
399
399
400
400
401
401
402
402
403
403
404
404
405
405
406
406
407
407
408
408
409
409
410
410
411
411
412
412
413
413
414
414
415
415
416
416
417
417
418
418
419
419
420
420
421
421
422
422
423
423
424
424
425
425
426
426
427
427
428
428
429
429
430
430
431
431
432
432
433
433
434
434
435
435
436
436
437
437
438
438
439
439
440
440
```

```

10    zlentry tail;
11    /* Find out prevlen for the entry that is inserted. */
12    if (p[0] != ZIP_END) {
13        ZIP_DECODE_PREVLEN(p, prevlensize, prevlen);
14    } else {
15        unsigned char *ptail = ZIPLIST_ENTRY_TAIL(zl);
16        if (ptail[0] != ZIP_END) {
17            prevlen = zipRawEntryLength(ptail);
18        }
19    }
20}
21/* See if the entry can be encoded */
22if (zipTryEncoding(s,slen,&value,&encoding)) {
23    /* 'encoding' is set to the appropriate integer encoding */
24    reqlen = zipIntSize(encoding);
25} else {
26    /* 'encoding' is untouched, however zipStoreEntryEncoding will use
the
27        * string length to figure out how to encode it. */
28    reqlen = slen;
29}
30/*
31 * We need space for both the length of the previous entry and
32 * the length of the payload. */
33reqlen += zipStorePrevEntryLength(NULL,prevlen);
34reqlen += zipStoreEntryEncoding(NULL,encoding,slen);
35/* When the insert position is not equal to the tail, we need to
36 * make sure that the next entry can hold this entry's length in
37 * its prevlen field. */
38int forcelarge = 0;
39nextdiff = (p[0] != ZIP_END) ? zipPrevLenByteDiff(p,reqlen) : 0;
40if (nextdiff == -4 && reqlen < 4) {
41    nextdiff = 0;
42    forcelarge = 1;
43}
44/* Store offset because a realloc may change the address of zl. */
45offset = p-zl;
46zl = ziplistResize(zl,curlen+reqlen+nextdiff);
47p = zl+offset;
48/* Apply memory move when necessary and update tail offset. */
49if (p[0] != ZIP_END) {
50    /* Subtract one because of the ZIP-END bytes */
51    memmove(p+reqlen,p-nextdiff,curlen-offset-1+nextdiff);
52    /* Encode this entry's raw length in the next entry. */
53    if (forcelarge)
54        zipStorePrevEntryLengthLarge(p+reqlen,reqlen);
55    else
56        zipStorePrevEntryLength(p+reqlen,reqlen);
57    /* Update offset for tail */
58    ZIPLIST_TAIL_OFFSET(zl) =
59        intrev32ifbe(intrev32ifbe(ZIPLIST_TAIL_OFFSET(zl))+reqlen);
60}
61}
62}
63}
64}

```

```

65     /* When the tail contains more than one entry, we need to take
66      * "nextdiff" in account as well. Otherwise, a change in the
67      * size of prevlen doesn't have an effect on the *tail* offset. */
68     zipEntry(p+reqlen, &tail);
69     if (p[reqlen+tail.headersize+tail.len] != ZIP_END) {
70         ZIPLIST_TAIL_OFFSET(zl) =
71
72         intrev32ifbe(intrev32ifbe(ZIPLIST_TAIL_OFFSET(zl))+nextdiff);
73     }
74 } else {
75     /* This element will be the new tail. */
76     ZIPLIST_TAIL_OFFSET(zl) = intrev32ifbe(p-zl);
77 }
78 /* When nextdiff != 0, the raw length of the next entry has changed,
79 so
80     * we need to cascade the update throughout the ziplist */
81 if (nextdiff != 0) {
82     offset = p-zl;
83     zl = __ziplistCascadeUpdate(zl,p+reqlen);
84     p = zl+offset;
85 }
86 /* Write the entry */
87 p += zipStorePrevEntryLength(p,prevlen);
88 p += zipStoreEntryEncoding(p,encoding,slen);
89 if (ZIP_IS_STR(encoding)) {
90     memcpy(p,s,slen);
91 } else {
92     zipSaveInteger(p,value,encoding);
93 }
94 ZIPLIST_INCR_LENGTH(zl,1);
95 return zl;
96 }
97 unsigned char *__ziplistCascadeUpdate(unsigned char *zl, unsigned char *p)
{
100    size_t curlen = intrev32ifbe(ZIPLIST_BYTES(zl)), rawlen, rawlensize;
101    size_t offset, noffset, extra;
102    unsigned char *np;
103    zlentry cur, next;
104    while (p[0] != ZIP_END) {
105        zipEntry(p, &cur);
106        rawlen = cur.headersize + cur.len;
107        rawlensize = zipStorePrevEntryLength(NULL,rawlen);
108        /* Abort if there is no next entry. */
109        if (p[rawlen] == ZIP_END) break;
110        zipEntry(p+rawlen, &next);
111        /* Abort when "prevlen" has not changed. */
112        if (next.prevrawlen == rawlen) break;
113        if (next.prevrawlensize < rawlensize) {
114            /* The "prevlen" field of "next" needs more bytes to hold

```

```

119         * the raw length of "cur". */
120         offset = p-zl;
121         extra = rawlensize-next.prevrawlensize;
122         zl = zipListResize(zl,curlen+extra);
123         p = zl+offset;
124         /* Current pointer and offset for next element. */
125         np = p+rawlen;
126         noffset = np-zl;
127         /* Update tail offset when next element is not the tail
element. */
128         if ((zl+intrev32ifbe(ZIPLIST_TAIL_OFFSET(zl))) != np) {
129             ZIPLIST_TAIL_OFFSET(zl) =
130
131             intrev32ifbe(intrev32ifbe(ZIPLIST_TAIL_OFFSET(zl))+extra);
132         }
133         /* Move the tail to the back. */
134         memmove(np+rawlensize,
135                 np+next.prevrawlensize,
136                 curlen-noffset-next.prevrawlensize-1);
137         zipStorePrevEntryLength(np,rawlen);
138         /* Advance the cursor */
139         p += rawlen;
140         curlen += extra;
141     } else {
142         if (next.prevrawlensize > rawlensize) {
143             /* This would result in shrinking, which we want to avoid.
144             * So, set "rawlen" in the available bytes. */
145             zipStorePrevEntryLengthLarge(p+rawlen,rawlen);
146         } else {
147             zipStorePrevEntryLength(p+rawlen,rawlen);
148         }
149         /* Stop here, as the raw length of "next" has not changed. */
150         break;
151     }
152 }
153 return zl;
154 }
```

这种代码的特点就是: 最好由作者去维护, 最好一次性写对. 因为读起来真的费劲, 改起来也很费劲.

2.7 quicklist

如果说ziplist是整个Redis中为了节省内存, 而写的最苟的数据结构, 那么称quicklist就是在最苟的基础上, 再苟了一层. 这个结构是Redis在3.2版本后新加的, 在3.2版本之前, 我们可以讲, dict是最复杂的底层数据结构, ziplist是最苟的底层数据结构. 在3.2版本之后, 这两个记录被双双刷新了.

这是一种，以ziplist为结点的，双端链表结构。宏观上，quicklist是一个链表，微观上，链表中的每个结点都是一个ziplist。

它的定义与实现分别在src/quicklist.h与src/quicklist.c中，其中关键定义如下：

```
1  /* Node, quicklist, and Iterator are the only data structures used
2   * currently. */
3  /* quicklistNode is a 32 byte struct describing a ziplist for a quicklist.
4   * We use bit fields keep the quicklistNode at 32 bytes.
5   * count: 16 bits, max 65536 (max zl bytes is 65k, so max count actually <
6   * 32k).
7   * encoding: 2 bits, RAW=1, LZF=2.
8   * container: 2 bits, NONE=1, ZIPLIST=2.
9   * recompress: 1 bit, bool, true if node is temporarry decompressed for
10  usage.
11  * attempted_compress: 1 bit, boolean, used for verifying during testing.
12  * extra: 12 bits, free for future use; pads out the remainder of 32 bits
13  */
14  typedef struct quicklistNode {
15      struct quicklistNode *prev;
16      struct quicklistNode *next;
17      unsigned char *zl;
18      unsigned int sz;           /* ziplist size in bytes */
19      unsigned int count : 16;    /* count of items in ziplist */
20      unsigned int encoding : 2;  /* RAW==1 or LZF==2 */
21      unsigned int container : 2; /* NONE==1 or ZIPLIST==2 */
22      unsigned int recompress : 1; /* was this node previous compressed? */
23      unsigned int attempted_compress : 1; /* node can't compress; too small
24      */
25      unsigned int extra : 10; /* more bits to steal for future usage */
26  } quicklistNode;
27  /* quicklistLZF is a 4+N byte struct holding 'sz' followed by
28  'compressed'.
29  * 'sz' is byte length of 'compressed' field.
30  * 'compressed' is LZF data with total (compressed) length 'sz'
31  * NOTE: uncompressed length is stored in quicklistNode->sz.
32  * When quicklistNode->zl is compressed, node->zl points to a quicklistLZF
33  */
34  typedef struct quicklistLZF {
35      unsigned int sz; /* LZF size in bytes*/
36      char compressed[];
37  } quicklistLZF;
38  /* quicklist is a 40 byte struct (on 64-bit systems) describing a
39  quicklist.
40  * 'count' is the number of total entries.
41  * 'len' is the number of quicklist nodes.
42  * 'compress' is: -1 if compression disabled, otherwise it's the number
43  *                 of quicklistNodes to leave uncompressed at ends of
44  quicklist.
45  * 'fill' is the user-requested (or default) fill factor. */
```

```

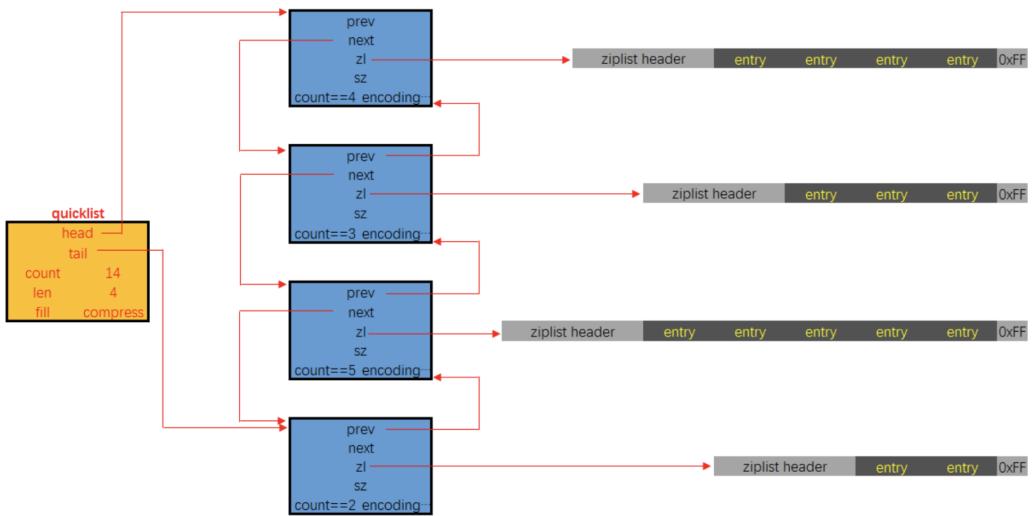
40  typedef struct quicklist {
41      quicklistNode *head;
42      quicklistNode *tail;
43      unsigned long count;           /* total count of all entries in all
44      ziplists */
45      unsigned long len;            /* number of quicklistNodes */
46      int fill : 16;                /* fill factor for individual nodes */
47      unsigned int compress : 16;   /* depth of end nodes not to
compress;0=off */
48  } quicklist;
49  typedef struct quicklistIter {
50      const quicklist *quicklist;
51      quicklistNode *current;
52      unsigned char *zi;
53      long offset; /* offset in current ziplist */
54      int direction;
55  } quicklistIter;
56  typedef struct quicklistEntry {
57      const quicklist *quicklist;
58      quicklistNode *node;
59      unsigned char *zi;
60      unsigned char *value;
61      long long longval;
62      unsigned int sz;
63      int offset;
64  } quicklistEntry;

```

这里定义了五个结构体:

- quicklistNode, 宏观上, quicklist是一个链表, 这个结构描述的就是链表中的结点. 它通过zi字段持有底层的ziplist. 简单来讲, 它描述了一个ziplist实例
- quicklistLZF, ziplist是一段连续的内存, 用LZ4算法压缩后, 就可以包装成一个quicklistLZF结构. 是否压缩quicklist中的每个ziplist实例是一个可配置项. 若这个配置项是开启的, 那么quicklistNode.zi字段指向的就不是一个ziplist实例, 而是一个压缩后的quicklistLZF实例
- quicklist. 这就是一个双链表的定义. head, tail分别指向头尾指针. len代表链表中的结点. count指的是整个quicklist中的所有ziplist中的entry的数目. fill字段影响着每个链表结点中ziplist的最大占用空间, compress影响着是否要对每个ziplist以LZ4算法进行进一步压缩以更节省内存空间.
- quicklistIter是一个迭代器
- quicklistEntry是对ziplist中的entry概念的封装. quicklist作为一个封装良好的数据结构, 不希望使用者感知到其内部的实现, 所以需要把ziplist.entry的概念重新包装一下.

quicklist的内存布局图如下所示:



下面是有关quicklist的更多额外信息:

quicklist.fill的值影响着每个链表结点中, ziplist的长度.

1. 当数值为负数时, 代表以字节数限制单个ziplist的最大长度. 具体为:
 - a. -1 不超过4kb
 - b. -2 不超过 8kb
 - c. -3 不超过 16kb
 - d. -4 不超过 32kb
 - e. -5 不超过 64kb
 - f. 当数值为正数时, 代表以entry数目限制单个ziplist的长度. 值即为数目. 由于该字段仅占16位, 所以以entry数目限制ziplist的容量时, 最大值为 2^{15} 个
2. quicklist.compress的值影响着quicklistNode.zl字段指向的是原生的ziplist, 还是经过压缩包装后的quicklistLZF
 - a. 0 表示不压缩, zl字段直接指向ziplist
 - b. 1 表示quicklist的链表头尾结点不压缩, 其余结点的zl字段指向的是经过压缩后的quicklist LZF
 - c. 2 表示quicklist的链表头两个, 与末两个结点不压缩, 其余结点的zl字段指向的是经过压缩后的quicklist LZF
 - d. 以此类推, 最大值为 2^{16}
3. quicklistNode.encoding字段, 以指示本链表结点所持有的ziplist是否经过了压缩. 1代表未压缩, 持有的是原生的ziplist, 2代表压缩过
4. quicklistNode.container字段指示的是每个链表结点所持有的数据类型是什么. 默认的实现是ziplist, 对应的该字段的值是2, 目前Redis没有提供其它实现. 所以实际上, 该字段的值恒为2
5. quicklistNode.recompress字段指示的是当前结点所持有的ziplist是否经过了解压. 如果该字段为1即代表之前被解压过, 且需要在下一次操作时重新压缩.

`quicklist` 的具体实现代码篇幅很长, 这里就不贴代码片断了, 从内存布局上也能看出来, 由于每个结点持有的ziplist是有上限长度的, 所以在与操作时要考虑的分支情况比较多. 想想都蛋疼.

quicklist有自己的优点, 也有缺点, 对于使用者来说, 其使用体验类似于线性数据结构, list作为最传统的双链表, 结点通过指针持有数据, 指针字段会耗费大量内存. ziplist解决了耗费内存这个问题. 但引入了新的问题: 每次写操作整个ziplist的内存都需要重分配.

quicklist在两者之间做了一个平衡. 并且使用者可以通过自定义quicklist.fill, 根据实际业务情况, 经验主义调参.

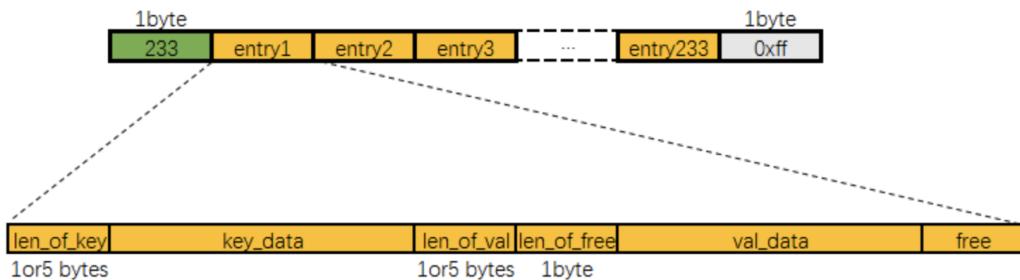
2.8 zipmap

dict作为字典结构, 优点很多, 扩展性强悍, 支持平滑扩容等等, 但对于字典中的键值均为二进制数据, 且长度都很小时, dict的中的一坨指针会浪费不少内存, 因此Redis又实现了一个轻量级的字典, 即为zipmap.

zipmap适合使用的场合是:

- 键值对量不大, 单个键, 单个值长度小
- 键值均是二进制数据, 而不是复合结构或复杂结构. dict支持各种嵌套, 字典本身并不持有数据, 而仅持有数据的指针. 但zipmap是直接持有数据的.

zipmap的定义与实现在src/zipmap.h与src/zipmap.c两个文件中, 其定义与实现均未定义任何struct结构体, 因为zipmap的内存布局就是一块连续的内存空间. 其内存布局如下所示:



- zipmap起始的第一个字节存储的是zipmap中键值对的个数. 如果键值对的个数大于254的话, 那么这个字节的值就是固定值254, 真实的键值对个数需要遍历才能获得.
- zipmap的最后一个字节是固定值0xFF
- zipmap中的每一个键值对, 称为一个entry, 其内存占用如上图, 分别六部分:
 - len_of_key, 一字节或五字节. 存储的是键的二进制长度. 如果长度小于254, 则用1字节存储, 否则用五个字节存储, 第一个字节的值固定为0xFE, 后四个字节以小端序uint32_t类型存储着键的二进制长度.
 - key_data为键的数据
 - len_of_val, 一字节或五字节, 存储的是值的二进制长度. 编码方式同len_of_key
 - len_of_free, 固定值1字节, 存储的是entry中未使用的空间的字节数. 未使用的空间即为图中的free, 它一般是由于键值对中的值被替换发生的. 比如, 键值对hello <-> word被修改为hello <-> w后, 就空了四个字节的闲置空间
 - val_data, 为值的数据
 - free, 为闲置空间. 由于len_of_free的值最大只能是254, 所以如果值的变更导致闲置空间大于254的话, zipmap就会回收内存空间.

Redis中内存淘汰算法实现

Redis的 `maxmemory` 支持的内存淘汰机制使得其成为一种有效的缓存方案, 成为memcached的有效替代方案。

当内存达到 `maxmemory` 后, Redis会按照 `maxmemory-policy` 启动淘汰策略。

Redis 3.0中已有淘汰机制:

- `noeviction`

- allkeys-lru
- volatile-lru
- allkeys-random
- volatile-random
- volatile-ttl

maxmemory-policy	含义	特性
noeviction	不淘汰	内存超限后写命令会返回错误(如OOM, del命令除外)
allkeys-lru	所有key的LRU机制在	所有key中按照最近最少使用LRU原则剔除key, 释放空间
volatile-lru	易失key的LRU	仅以设置过期时间key范围内的LRU(如均为设置过期时间, 则不会淘汰)
allkeys-random	所有key随机淘汰	一视同仁, 随机
volatile-random	易失Key的随机	仅设置过期时间key范围内的随机
volatile-ttl	易失key的TTL淘汰	按最小TTL的key优先淘汰

其中LRU(less recently used)经典淘汰算法在Redis实现中有一定优化设计, 来保证内存占用与实际效果的平衡, 这也体现了工程应用是空间与时间的平衡性。

PS: 值得注意的, 在主从复制模式Replication下, 从节点达到maxmemory时不会有任何异常日志信息, 但现象为增量数据无法同步至从节点。

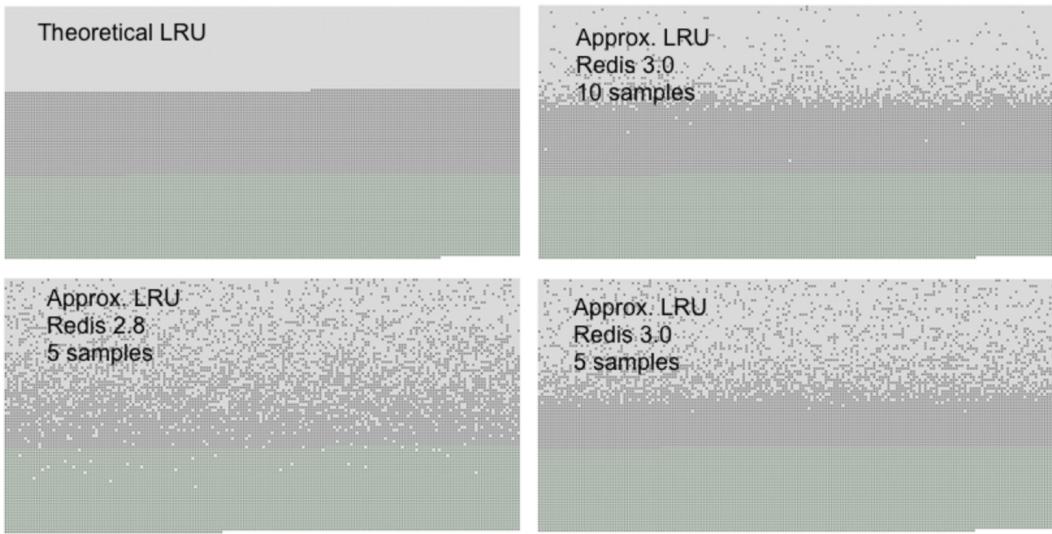
Redis 3.0中近似LRU算法

Redis中LRU是近似LRU实现, 并不能取出理想LRU理论中最佳淘汰Key, 而是通过从小部分采样后的样本中淘汰局部LRU键。

Redis 3.0中近似LRU算法通过增加待淘汰元素池的方式进一步优化, 最终实现与精确LRU非常接近的表现。

精确LRU会占用较大内存记录历史状态, 而近似LRU则用较小内存支出实现近似效果。

以下是理论LRU和近似LRU的效果对比:



- 按时间顺序接入不同键，此时最早写入也就是最佳淘汰键
- 浅灰色区域：被淘汰的键
- 灰色区域：未被淘汰的键
- 绿色区域：新增写入的键

总结图中展示规律，

- 图1Theoretical LRU符合预期：最早写入键逐步被淘汰
- 图2Approx LRU Redis 3.0 10 samples：Redis 3.0中近似LRU算法(采样值为10)
- 图3Approx LRU Redis 2.8 5 samples：Redis 2.8中近似LRU算法(采样值为5)
- 图4Approx LRU Redis 3.0 5 samples：Redis 3.0中近似LRU算法(采样值为5)

结论：

- 通过图4和图3对比：得出相同采样值下，3.0比2.8的LRU淘汰机制更接近理论LRU
- 通过图4和图2对比：得出增加采样值，在3.0中将进一步改善LRU淘汰效果逼近理论LRU
- 对比图2和图1：在3.0中采样值为10时，效果非常接近理论LRU

采样值设置通过maxmemory-samples指定，可通过CONFIG SET maxmemory-samples 动态设置，也可启动配置中指定maxmemory-samples

源码解析

```

1 int freeMemoryIfNeeded(void){
2     while (mem_freed < mem_tofree) {
3         if (server.maxmemory_policy == REDIS_MAXMEMORY_NO_EVICTION)
4             return REDIS_ERR; /* We need to free memory, but policy forbids.
5 */
6         if (server.maxmemory_policy == REDIS_MAXMEMORY_ALLKEYS_LRU ||
7             server.maxmemory_policy == REDIS_MAXMEMORY_ALLKEYS_RANDOM)
8             {.....}
9         /* volatile-random and allkeys-random policy */
10        if (server.maxmemory_policy == REDIS_MAXMEMORY_ALLKEYS_RANDOM ||
11            server.maxmemory_policy ==
12            REDIS_MAXMEMORY_VOLATILE_RANDOM)
13                {.....}
14        /* volatile-lru and allkeys-lru policy */
15        else if (server.maxmemory_policy == REDIS_MAXMEMORY_ALLKEYS_LRU ||
16

```

```

15         server.maxmemory_policy == REDIS_MAXMEMORY_VOLATILE_LRU)
16     {
17         // 淘汰池函数
18         evictionPoolPopulate(dict, db->dict, db->eviction_pool);
19         while(bestkey == NULL) {
20             evictionPoolPopulate(dict, db->dict, db->eviction_pool);
21             // 从后向前逐一淘汰
22             for (k = REDIS_EVICT_POOL_SIZE-1; k >= 0; k--) {
23                 if (pool[k].key == NULL) continue;
24                 de = dictFind(dict, pool[k].key); // 定位目标
25                 /* Remove the entry from the pool. */
26                 sdsfree(pool[k].key);
27                 /* Shift all elements on its right to left. */
28                 memmove(pool+k, pool+k+1,
29                         sizeof(pool[0])*(REDIS_EVICT_POOL_SIZE-k-1));
30                 /* Clear the element on the right which is empty
31                  * since we shifted one position to the left. */
32                 pool[REDIS_EVICT_POOL_SIZE-1].key = NULL;
33                 pool[REDIS_EVICT_POOL_SIZE-1].idle = 0;
34                 /* If the key exists, is our pick. Otherwise it is
35                  * a ghost and we need to try the next element. */
36                 if (de) {
37                     bestkey = dictGetKey(de); // 确定删除键
38                     break;
39                 } else {
40                     /* Ghost... */
41                     continue;
42                 }
43             }
44         }
45     }
46 }
47 }
48 /* volatile-ttl */
49 else if (server.maxmemory_policy == EDIS_MAXMEMORY_VOLATILE_TTL)
{.....}
50     // 最终选定待删除键bestkey
51     if (bestkey) {
52         long long delta;
53         robj *keyobj = createStringObject(bestkey, sdslenbestkey)); // 目标对象
54         propagateExpire(db, keyobj);
55         latencyStartMonitor(eviction_latency); // 延迟监控开始
56         dbDelete(db, keyobj); // 从db删除对象
57         latencyEndMonitor(eviction_latency); // 延迟监控结束
58         latencyAddSampleIfNeeded("eviction-del", iction_latency); // 延迟采样
59         latencyRemoveNestedEvent(latency, eviction_latency);
60         delta -= (long long) zmalloc_used_memory();
61         mem_freed += delta; // 释放内存计数
62         server.stat_evictedkeys++; // 淘汰key计数, info中可见

```

```
64         notifyKeyspaceEvent(REDIS_NOTIFY_EVICTED, "evicted", keyobj,
65     db->id); // 事件通知
66     decrRefCount(keyobj); // 引用计数更新
67     keys_freed++;
68     // 避免删除较多键导致的主从延迟，在循环内同步
69     if (slaves) flushSlavesOutputBuffers();
70 }
71 }
```

Redis 4.0中新的LFU算法

从Redis4.0开始，新增LFU淘汰机制，提供更好缓存命中率。LFU(Least Frequently Used)通过记录键使用频率来定位最可能淘汰的键。

对比LRU与LFU的差别：

- 在LRU中，某个键很少被访问，但在刚刚被访问后其被淘汰概率很低，从而出现这类异常持续存在的缓存；相对的，其他可能被访问的键会被淘汰
- 而LFU中，按访问频次淘汰最少被访问的键

Redis 4.0中新增两种LFU淘汰机制：

- volatile-lfu：设置过期时间的键按LFU淘汰
- allkeys-lfu：所有键按LFU淘汰

LFU使用Morris counters计数器占用少量位数来评估每个对象的访问频率，并随时间更新计数器。此机制实现与近似LRU中采样类似。但与LRU不同，LFU提供明确参数来指定计数更新频率。

- lfu-log-factor：0–255之间，饱和因子，值越小代表饱和速度越快
- lfu-decay-time：衰减周期，单位分钟，计数器衰减的分钟数

这两个因子形成一种平衡，通过少量访问 VS 多次访问 的评价标准最终形成对键重要性的评判。

原文：<http://fivezh.github.io/2019/01/10/Redis-LRU-algorithm/>

Redis中内存淘汰算法实现

Redis的 `maxmemory` 支持的内存淘汰机制使得其成为一种有效的缓存方案，成为 memcached的有效替代方案。

当内存达到 `maxmemory` 后，Redis会按照 `maxmemory-policy` 启动淘汰策略。

Redis 3.0中已有淘汰机制：

- noeviction
- allkeys-lru
- volatile-lru
- allkeys-random
- volatile-random
- volatile-ttl

maxmemory-policy	含义	特性
noeviction	不淘汰	内存超限后写命令会返回错误(如OOM, del命令除外)
allkeys-lru	所有key的LRU机制在	所有key中按照最近最少使用LRU原则剔除key, 释放空间
volatile-lru	易失key的LRU	仅以设置过期时间key范围内的LRU(如均为设置过期时间, 则不会淘汰)
allkeys-random	所有key随机淘汰	一视同仁, 随机
volatile-random	易失Key的随机	仅设置过期时间key范围内的随机
volatile-ttl	易失key的TTL淘汰	按最小TTL的key优先淘汰

其中LRU(less recently used)经典淘汰算法在Redis实现中有一定优化设计, 来保证内存占用与实际效果的平衡, 这也体现了工程应用是空间与时间的平衡性。

PS: 值得注意的, 在主从复制模式Replication下, 从节点达到maxmemory时不会有任何异常日志信息, 但现象为增量数据无法同步至从节点。

MySQL相关

1. MySQL 索引使用有哪些注意事项呢?

可以从三个维度回答这个问题: 索引哪些情况会失效, 索引不适合哪些场景, 索引规则

索引哪些情况会失效

- 查询条件包含or, 可能导致索引失效
- 如何字段类型是字符串, where时一定用引号括起来, 否则索引失效
- like通配符可能导致索引失效。
- 联合索引, 查询时的条件列不是联合索引中的第一个列, 索引失效。
- 在索引列上使用mysql的内置函数, 索引失效。
- 对索引列运算 (如, +、-、*、/) , 索引失效。
- 索引字段上使用 (! = 或者 < >, not in) 时, 可能会导致索引失效。
- 索引字段上使用is null, is not null, 可能导致索引失效。
- 左连接查询或者右连接查询查询关联的字段编码格式不一样, 可能导致索引失效。
- mysql估计使用全表扫描要比使用索引快, 则不使用索引。

索引不适合哪些场景

- 数据量少的不适合加索引

- 更新比较频繁的也不适合加索引
- 区分度低的字段不适合加索引（如性别）

索引的一些潜规则

- 覆盖索引
- 回表
- 索引数据结构（B+树）
- 最左前缀原则
- 索引下推

2. MySQL 遇到过死锁问题吗，你是如何解决的？

我排查死锁的一般步骤是酱紫的：

- 查看死锁日志`show engine innodb status;`
- 找出死锁Sql
- 分析sql加锁情况
- 模拟死锁案发
- 分析死锁日志
- 分析死锁结果

3. 日常工作中你是怎么优化SQL的？

可以从这几个维度回答这个问题：

- 加索引
- 避免返回不必要的数据
- 适当分批量进行
- 优化sql结构
- 分库分表
- 读写分离

4. 说说分库与分表的设计

分库分表方案，分库分表中间件，分库分表可能遇到的问题

分库分表方案：

- 水平分库：以字段为依据，按照一定策略（hash、range等），将一个库中的数据拆分到多个库中。

- 水平分表：以字段为依据，按照一定策略（hash、range等），将一个表中的数据拆分到多个表中。
- 垂直分库：以表为依据，按照业务归属不同，将不同的表拆分到不同的库中。
- 垂直分表：以字段为依据，按照字段的活跃性，将表中字段拆到不同的表（主表和扩展表）中。

常用的分库分表中间件：

- sharding-jdbc（当当）
- Mycat
- TDDL（淘宝）
- Oceanus(58同城数据库中间件)
- vitess（谷歌开发的数据库中间件）
- Atlas(Qihoo 360)

分库分表可能遇到的问题

- 事务问题：需要用分布式事务啦
- 跨节点Join的问题：解决这一问题可以分两次查询实现
- 跨节点的count,order by,group by以及聚合函数问题：分别在各个节点上得到结果后在应用程序端进行合并。
- 数据迁移，容量规划，扩容等问题
- ID问题：数据库被切分后，不能再依赖数据库自身的主键生成机制啦，最简单可以考虑UUID
- 跨分片的排序分页问题（后台加大pagesize处理？）

5. InnoDB与MyISAM的区别

- InnoDB支持事务，MyISAM不支持事务
- InnoDB支持外键，MyISAM不支持外键
- InnoDB 支持 MVCC(多版本并发控制)，MyISAM 不支持
- `select count(*) from table` 时，MyISAM更快，因为它有一个变量保存了整个表的总行数，可以直接读取，InnoDB就需要全表扫描。
- Innodb不支持全文索引，而MyISAM支持全文索引（5.7以后的InnoDB也支持全文索引）
- InnoDB支持表、行级锁，而MyISAM支持表级锁。
- InnoDB表必须有主键，而MyISAM可以没有主键
- Innodb表需要更多的内存和存储，而MyISAM可被压缩，存储空间较小，。
- Innodb按主键大小有序插入，MyISAM记录插入顺序是，按记录插入顺序保存。
- InnoDB 存储引擎提供了具有提交、回滚、崩溃恢复能力的事务安全，与 MyISAM 比 InnoDB 写的效率差一些，并且会占用更多的磁盘空间以保留数据和索引

- InnoDB 属于索引组织表，使用共享表空间和多表空间储存数据。MyISAM 用 `.frm`、`.MYD`、`.MTI` 来储存表定义，数据和索引。

6. 数据库索引的原理，为什么要用 B+树，为什么不用二叉树？

可以从几个维度去看这个问题，查询是否够快，效率是否稳定，存储数据多少，以及查找磁盘次数，为什么不是二叉树，为什么不是平衡二叉树，为什么不是B树，而偏偏是B+树呢？

为什么不一般二叉树？

如果二叉树特殊化为一个链表，相当于全表扫描。平衡二叉树相比于二叉查找树来说，查找效率更稳定，总体的查找速度也更快。

为什么不平衡二叉树呢？

我们知道，在内存比在磁盘的数据，查询效率快得多。如果树这种数据结构作为索引，那我们每查找一次数据就需要从磁盘中读取一个节点，也就是我们说的一个磁盘块，但是平衡二叉树可是每个节点只存储一个键值和数据的，如果是B树，可以存储更多的节点数据，树的高度也会降低，因此读取磁盘的次数就降下来啦，查询效率就快啦。

那为什么不B树而是B+树呢？

- 1) B+树非叶子节点上是不存储数据的，仅存储键值，而B树节点中不仅存储键值，也会存储数据。innodb中页的默认大小是16KB，如果不存储数据，那么就会存储更多的键值，相应的树的阶数（节点的子节点树）就会更大，树就会更矮更胖，如此一来我们查找数据进行磁盘的IO次数会再次减少，数据查询的效率也会更快。
- 2) B+树索引的所有数据均存储在叶子节点，而且数据是按照顺序排列的，链表连着的。那么B+树使得范围查找，排序查找，分组查找以及去重查找变得异常简单。

7. 聚集索引与非聚集索引的区别

- 一个表中只能拥有一个聚集索引，而非聚集索引一个表可以存在多个。
- 聚集索引，索引中键值的逻辑顺序决定了表中相应行的物理顺序；非聚集索引，索引中索引的逻辑顺序与磁盘上行的物理存储顺序不同。
- 索引是通过二叉树的数据结构来描述的，我们可以这么理解聚簇索引：索引的叶节点就是数据节点。而非聚簇索引的叶节点仍然是索引节点，只不过有一个指针指向对应的数据块。
- 聚集索引：物理存储按照索引排序；非聚集索引：物理存储不按照索引排序；何时使用聚集索引或非聚集索引？

动作描述	使用聚集索引	使用非聚集索引
列经常被分组排序	应	应
返回某范围内的数据	应	不应
一个或极少不同值	不应	不应
小数目的不同值	应	不应
大数目的不同值	不应	应
频繁更新的列	不应	应
外键列	应	应
主键列	应	应
频繁修改索引列	不应	应

8. limit 1000000 加载很慢的话，你是怎么解决的呢？

方案一：如果id是连续的，可以这样，返回上次查询的最大记录(偏移量)，再往下limit

```
1 select id, name from employee where id>1000000 limit 10.
```

方案二：在业务允许的情况下限制页数：

建议跟业务讨论，有没有必要查这么后的分页啦。因为绝大多数用户都不会往后翻太多页。

方案三：order by + 索引 (id为索引)

```
1 select id, name from employee order by id limit 1000000, 10
2 SELECT a.* FROM employee a, (select id from employee where 条件 LIMIT
1000000,10 ) b where a.id=b.id
```

方案四：利用延迟关联或者子查询优化超多分页场景。（先快速定位需要获取的id段，然后再关联）

9. 如何选择合适的分布式主键方案呢？

- 数据库自增长序列或字段。
- UUID。
- Redis生成ID
- Twitter的snowflake算法
- 利用zookeeper生成唯一ID
- MongoDB的ObjectId

10. 事务的隔离级别有哪些？MySQL的默认隔离级别是什么？

- 读未提交 (Read Uncommitted)

- 读已提交 (Read Committed)
- 可重复读 (Repeatable Read)
- 串行化 (Serializable)

Mysql默认的事务隔离级别是可重复读(Repeatable Read)

11. 什么是幻读，脏读，不可重复读呢？

- 事务A、B交替执行，事务A被事务B干扰到了，因为事务A读取到事务B未提交的数据,这就是脏读
- 在一个事务范围内，两个相同的查询，读取同一条记录，却返回了不同的数据，这就是不可重复读。
- 事务A查询一个范围的结果集，另一个并发事务B往这个范围中插入/删除了数据，并静悄悄地提交，然后事务A再次查询相同的范围，两次读取得到的结果集不一样了，这就是幻读。

12. 在高并发情况下，如何做到安全的修改同一行数据？

要安全的修改同一行数据，就要保证一个线程在修改时其它线程无法更新这行记录。一般有悲观锁和乐观锁两种方案~

使用悲观锁

悲观锁思想就是，当前线程要进来修改数据时，别的线程都得拒之门外~ 比如，可以使用select...for update ~

```
1 select * from User where name='jay' for update
```

以上这条sql语句会锁定了User表中所有符合检索条件 (name='jay') 的记录。本次事务提交之前，别的线程都无法修改这些记录。

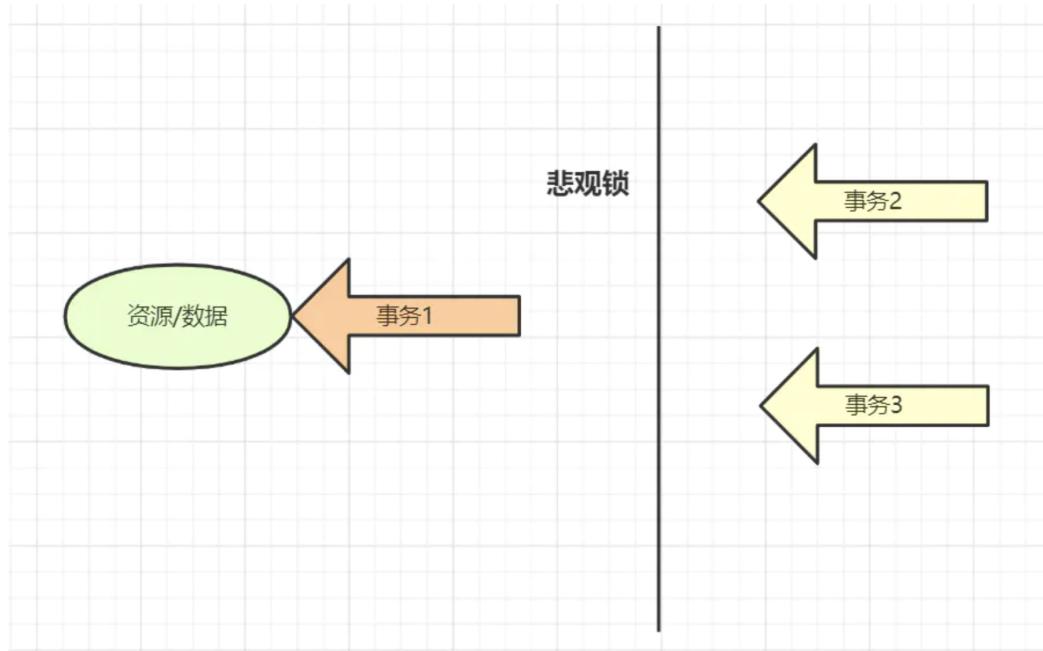
使用乐观锁

乐观锁思想就是，有线程过来，先放过去修改，如果看到别的线程没修改过，就可以修改成功，如果别的线程修改过，就修改失败或者重试。实现方式：乐观锁一般会使用版本号机制或CAS算法实现。

13. 数据库的乐观锁和悲观锁。

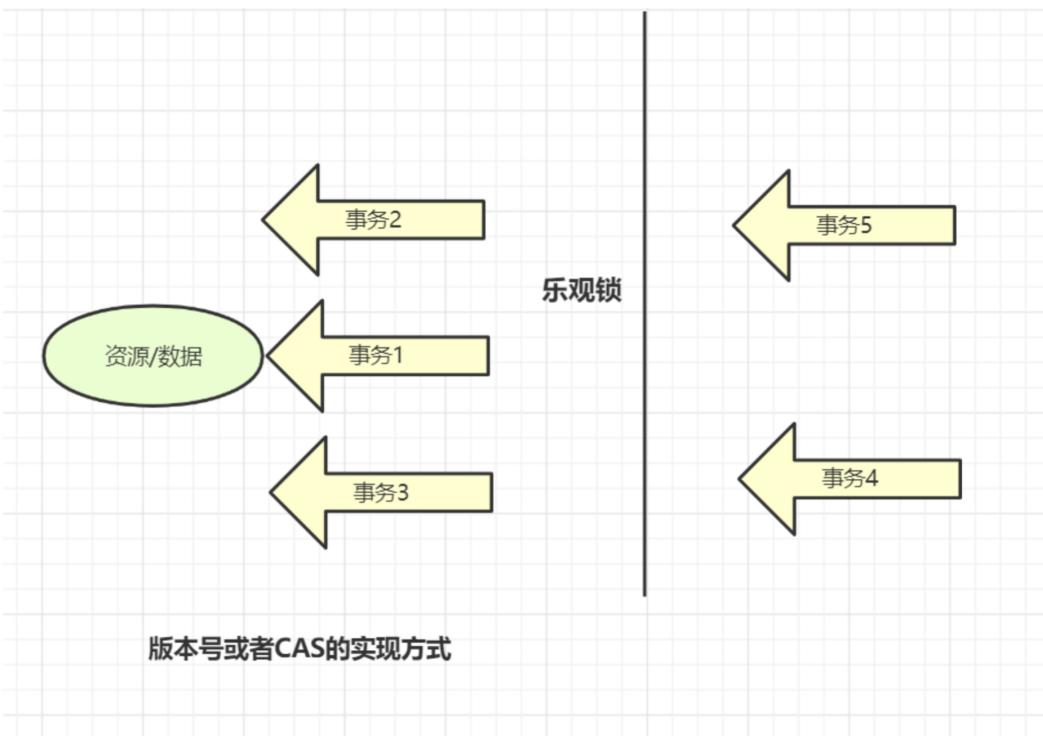
悲观锁：

悲观锁她专一且缺乏安全感了，她的心只属于当前事务，每时每刻都担心着它心爱的数据可能被别的事务修改，所以一个事务拥有（获得）悲观锁后，其他任何事务都不能对数据进行修改啦，只能等待锁被释放才可以执行。



乐观锁：

乐观锁的“乐观情绪”体现在，它认为数据的变动不会太频繁。因此，它允许多个事务同时对数据进行变动。实现方式：乐观锁一般会使用版本号机制或CAS算法实现。



14. SQL优化的一般步骤是什么，怎么看执行计划（explain），如何理解其中各个字段的含义。

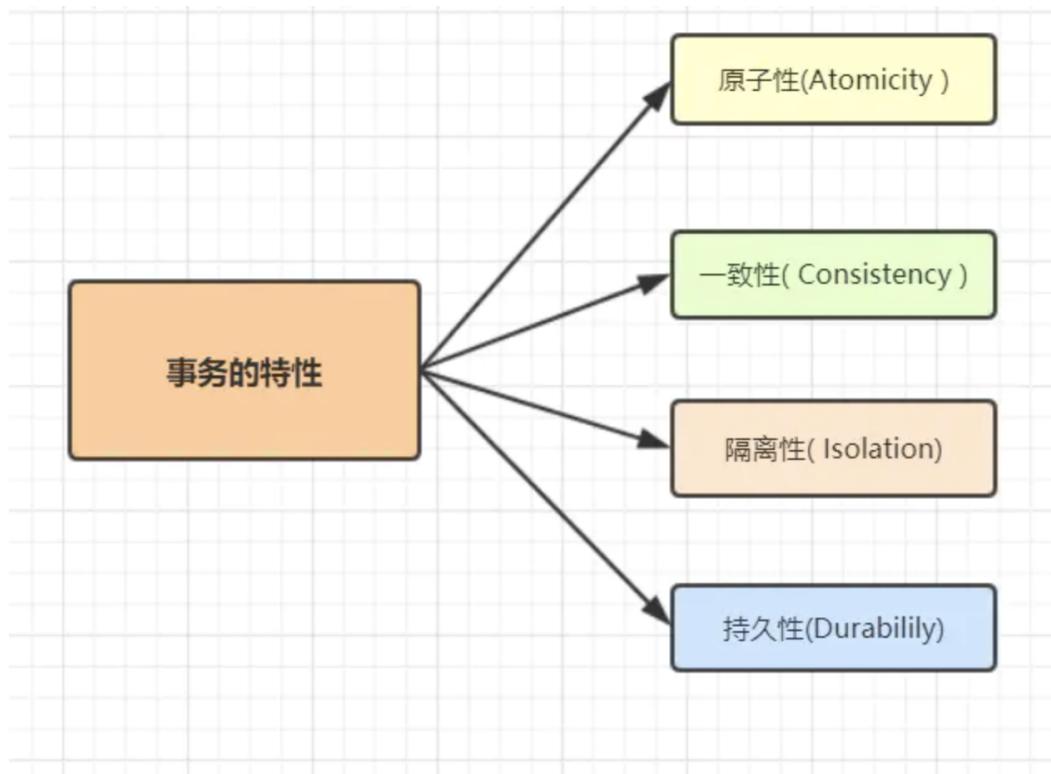
- `show status` 命令了解各种 sql 的执行频率
- 通过慢查询日志定位那些执行效率较低的 sql 语句
- `explain` 分析低效 sql 的执行计划（这点非常重要，日常开发中用它分析Sql，会大大降低Sql导致的线上事故）

15. select for update有什么含义，会锁表还是锁行还是其他。

select for update 含义

select查询语句是不会加锁的，但是select for update除了有查询的作用外，还会加锁呢，而且它是悲观锁哦。至于加了是行锁还是表锁，这就要看是不是用了索引/主键啦。没用索引/主键的话就是表锁，否则就是行锁。

16. MySQL事务得四大特性以及实现原理



- 原子性： 事务作为一个整体被执行，包含在其中的对数据库的操作要么全部被执行，要么都不执行。
- 一致性： 指在事务开始之前和事务结束以后，数据不会被破坏，假如A账户给B账户转10块钱，不管成功与否，A和B的总金额是不变的。

- 隔离性：多个事务并发访问时，事务之间是相互隔离的，即一个事务不影响其它事务运行效果。简言之，就是事务之间是进水不犯河水的。
- 持久性：表示事务完成以后，该事务对数据库所作的操作更改，将持久地保存在数据库之中。

事务ACID特性的实现思想

- 原子性：是使用 undo log来实现的，如果事务执行过程中出错或者用户执行了 rollback，系统通过undo log日志返回事务开始的状态。
- 持久性：使用 redo log来实现，只要redo log日志持久化了，当系统崩溃，即可通过redo log把数据恢复。
- 隔离性：通过锁以及MVCC,使事务相互隔离开。
- 一致性：通过回滚、恢复，以及并发情况下的隔离性，从而实现一致性。

17. 如果某个表有近千万数据，CRUD比较慢，如何优化。

分库分表

某个表有近千万数据，可以考虑优化表结构，分表（水平分表，垂直分表），当然，你这样回答，需要准备好面试官问你的分库分表相关问题呀，如

- 分表方案（水平分表，垂直分表，切分规则hash等）
- 分库分表中间件（Mycat，sharding-jdbc等）
- 分库分表一些问题（事务问题？跨节点Join的问题）
- 解决方案（分布式事务等）

索引优化

除了分库分表，优化表结构，当然还有所以索引优化等方案~

18. 如何写sql能够有效的使用到复合索引。

复合索引，也叫组合索引，用户可以在多个列上建立索引，这种索引叫做复合索引。当我们创建一个组合索引的时候，如(k1,k2,k3)，相当于创建了 (k1) 、(k1,k2) 和 (k1,k2,k3) 三个索引，这就是最左匹配原则。

```
1 select * from table where k1=A AND k2=B AND k3=D
```

有关于复合索引，我们需要关注查询Sql条件的顺序，确保最左匹配原则有效，同时可以删除不必要的冗余索引。

19. mysql中in 和exists的区别。

这个，跟一下demo来看更刺激吧，啊哈哈

假设表A表示某企业的员工表，表B表示部门表，查询所有部门的所有员工，很容易有以下SQL：

```
1 select * from A where deptId in (select deptId from B);
```

这样写等价于：

先查询部门表B select deptId from B 再由部门deptId，查询A的员工 select * from A where A.deptId = B.deptId

可以抽象成这样的一个循环：

```
1 List<> resultSet ;
2 for(int i=0;i<B.length;i++) {
3     for(int j=0;j<A.length;j++) {
4         if(A[i].id==B[j].id) {
5             resultSet.add(A[i]);
6             break;
7         }
8     }
9 }
```

显然，除了使用in，我们也可以用exists实现一样的查询功能，如下：

```
1 select * from A where exists (select 1 from B where A.deptId = B.deptId);
```

因为exists查询的理解就是，先执行主查询，获得数据后，再放到子查询中做条件验证，根据验证结果（true或者false），来决定主查询的数据结果是否得意保留。

那么，这样写就等价于：

select * from A,先从A表做循环 select * from B where A.deptId = B.deptId,再从B表做循环.

同理，可以抽象成这样一个循环：

```
1 List<> resultSet ;
2 for(int i=0;i<A.length;i++) {
3     for(int j=0;j<B.length;j++) {
4         if(A[i].deptId==B[j].deptId) {
5             resultSet.add(A[i]);
6             break;
7         }
8     }
9 }
```

数据库最费劲的就是跟程序链接释放。假设链接了两次，每次做上百万次的数据集查询，查完就走，这样就只做了两次；相反建立了上百万次链接，申请链接释放反复重复，这样系统就受不了了。即mysql优化原则，就是小表驱动大表，小的数据集驱动大的数据集，从而让性能更优。因此，我们要选择最外层循环小的，也就是，如果B的数

据量小于A，适合使用in，如果B的数据量大于A，即适合选择exists，这就是in和exists的区别。

20. 数据库自增主键可能遇到什么问题。

使用自增主键对数据库做分库分表，可能出现诸如主键重复等的问题。解决方案的话，简单点的话可以考虑使用UUID哈自增主键会产生表锁，从而引发问题 自增主键可能用完问题。

21. MVCC熟悉吗，它的底层原理？

MVCC,多版本并发控制,它是通过读取历史版本的数据，来降低并发事务冲突，从而提高并发性能的一种机制。

MVCC需要关注这几个知识点：

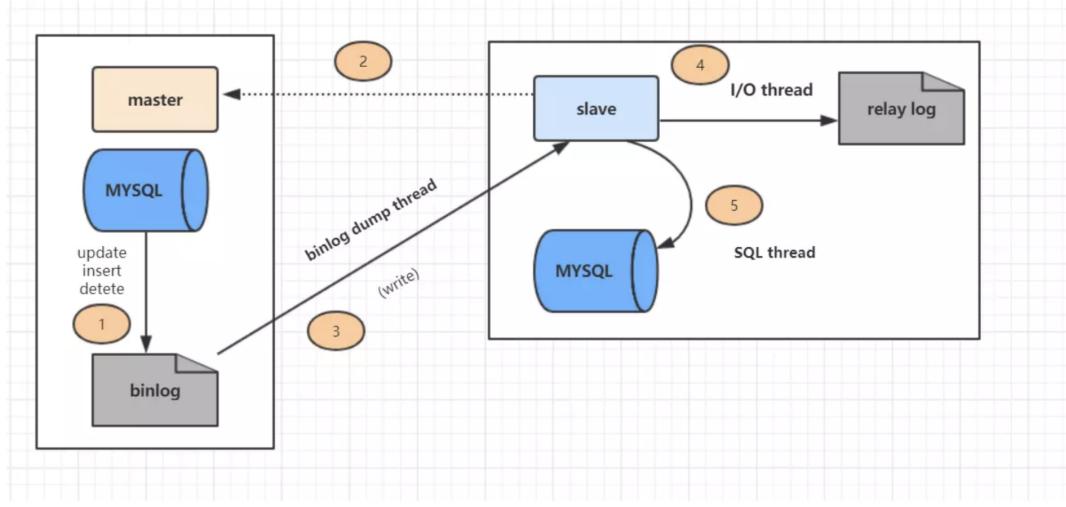
- 事务版本号
- 表的隐藏列
- undo log
- read view

22. 数据库中间件了解过吗， sharding jdbc, mycat?

sharding-jdbc目前是基于jdbc驱动，无需额外的proxy，因此也无需关注proxy本身的高可用。Mycat 是基于 Proxy，它复写了 MySQL 协议，将 Mycat Server 伪装成一个 MySQL 数据库，而 Sharding-JDBC 是基于 JDBC 接口的扩展，是以 jar 包的形式提供轻量级服务的。

23. MYSQL的主从延迟，你怎么解决？

嘻嘻，先复习一下主从复制原理吧，如图：



主从复制分了五个步骤进行：

- 步骤一：主库的更新事件(update、insert、delete)被写到binlog
- 步骤二：从库发起连接，连接到主库。
- 步骤三：此时主库创建一个binlog dump thread，把binlog的内容发送到从库。
- 步骤四：从库启动之后，创建一个I/O线程，读取主库传过来的binlog内容并写入到relay log
- 步骤五：还会创建一个SQL线程，从relay log里面读取内容，从 Exec_Master_Log_Pos位置开始执行读取到的更新事件，将更新内容写入到slave 的db

主从同步延迟的原因

一个服务器开放N个链接给客户端来连接的，这样有会有大并发的更新操作，但是从服务器的里面读取binlog的线程仅有一个，当某个SQL在从服务器上执行的时间稍长或者由于某个SQL要进行锁表就会导致，主服务器的SQL大量积压，未被同步到从服务器里。这就导致了主从不一致，也就是主从延迟。

主从同步延迟的解决办法

- 主服务器要负责更新操作，对安全性的要求比从服务器要高，所以有些设置参数可以修改，比如sync_binlog=1, innodb_flush_log_at_trx_commit = 1 之类的设置等。
- 选择更好的硬件设备作为slave。
- 把一台从服务器当度作为备份使用，而不提供查询，那边他的负载下来了，执行 relay log 里面的SQL效率自然就高了。
- 增加从服务器喽，这个目的还是分散读的压力，从而降低服务器负载。

24. 说一下大表查询的优化方案

- 优化schema、sql语句+索引；

- 可以考虑加缓存，memcached, redis，或者JVM本地缓存；
- 主从复制，读写分离；
- 分库分表；

25. 什么是数据库连接池?为什么需要数据库连接池呢?

连接池基本原理：

数据库连接池原理：在内部对象池中，维护一定数量的数据库连接，并对外暴露数据库连接的获取和返回方法。

应用程序和数据库建立连接的过程：

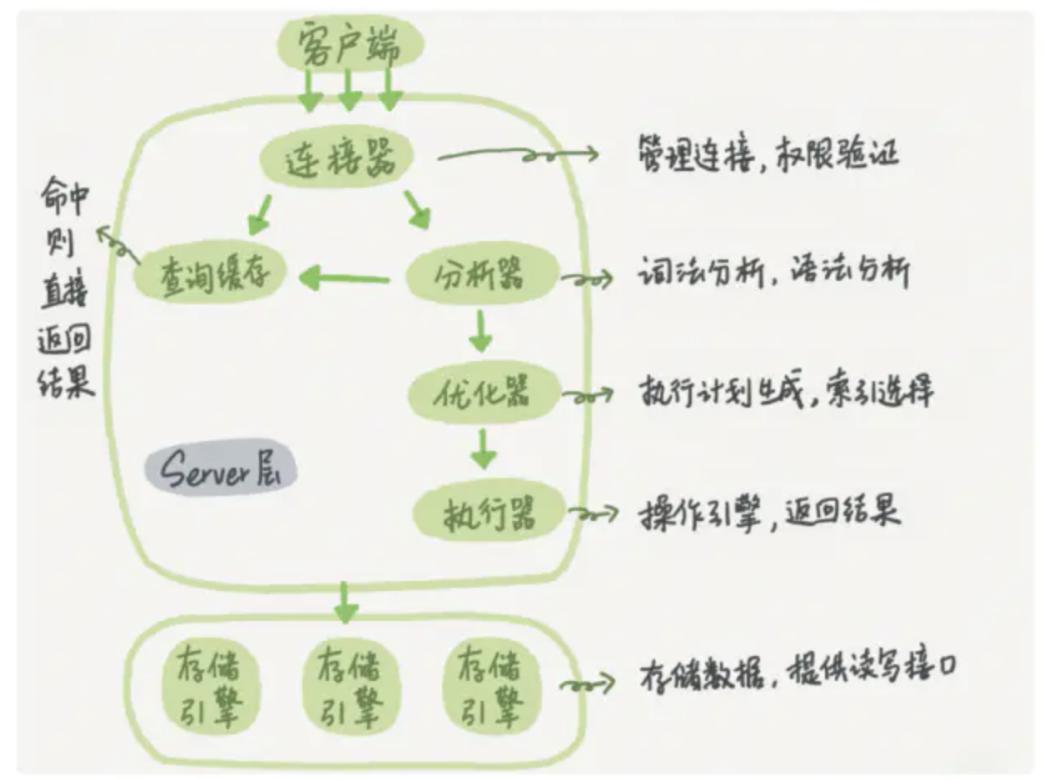
- 通过TCP协议的三次握手和数据库服务器建立连接
- 发送数据库用户账号密码，等待数据库验证用户身份
- 完成身份验证后，系统可以提交SQL语句到数据库执行
- 把连接关闭，TCP四次挥手告别。

数据库连接池好处：

- 资源重用 (连接复用)
- 更快的系统响应速度
- 新的资源分配手段 统一的连接管理，避免数据库连接泄漏

26. 一条SQL语句在MySQL中如何执行的?

先看一下Mysql的逻辑架构图吧~



查询语句:

- 先检查该语句是否有权限
- 如果没有权限，直接返回错误信息
- 如果有权限，在 MySQL8.0 版本以前，会先查询缓存。
- 如果没有缓存，分析器进行词法分析，提取 sql 语句 select 等的关键元素。然后判断 sql 语句是否有语法错误，比如关键词是否正确等等。
- 优化器进行确定执行方案
- 进行权限校验，如果没有权限就直接返回错误信息，如果有权限就会调用数据库引擎接口，返回执行结果。

27. InnoDB引擎中的索引策略，了解过吗？

- 覆盖索引
- 最左前缀原则
- 索引下推
- 索引下推优化是 MySQL 5.6 引入的，可以在索引遍历过程中，对索引中包含的字段先做判断，直接过滤掉不满足条件的记录，减少回表次数。

28. 数据库存储日期格式时，如何考虑时区转换问题？

- datetime类型适合用来记录数据的原始的创建时间，修改记录中其他字段的值，datetime字段的值不会改变，除非手动修改它。

- timestamp类型适合用来记录数据的最后修改时间，只要修改了记录中其他字段的值，timestamp字段的值都会被自动更新。

29. 一条sql执行过长的时间，你如何优化，从哪些方面入手？

- 查看是否涉及多表和子查询，优化Sql结构，如去除冗余字段，是否可拆表等
- 优化索引结构，看是否可以适当添加索引
- 数量大的表，可以考虑进行分离/分表（如交易流水表）
- 数据库主从分离，读写分离
- explain分析sql语句，查看执行计划，优化sql
- 查看mysql执行日志，分析是否有其他方面的问题

30. MYSQL数据库服务器性能分析的方法命令有哪些？

- Show status, 一些值得监控的变量值：

Bytes_received和Bytes_sent 和服务器之间来往的流量。 Com_*服务器正在执行的命令。 Created_*在查询执行期限间创建的临时表和文件。 Handler_*存储引擎操作。 Select_*不同类型的联接执行计划。 Sort_*几种排序信息。
- Show profiles 是MySQL用来分析当前会话SQL语句执行的资源消耗情况

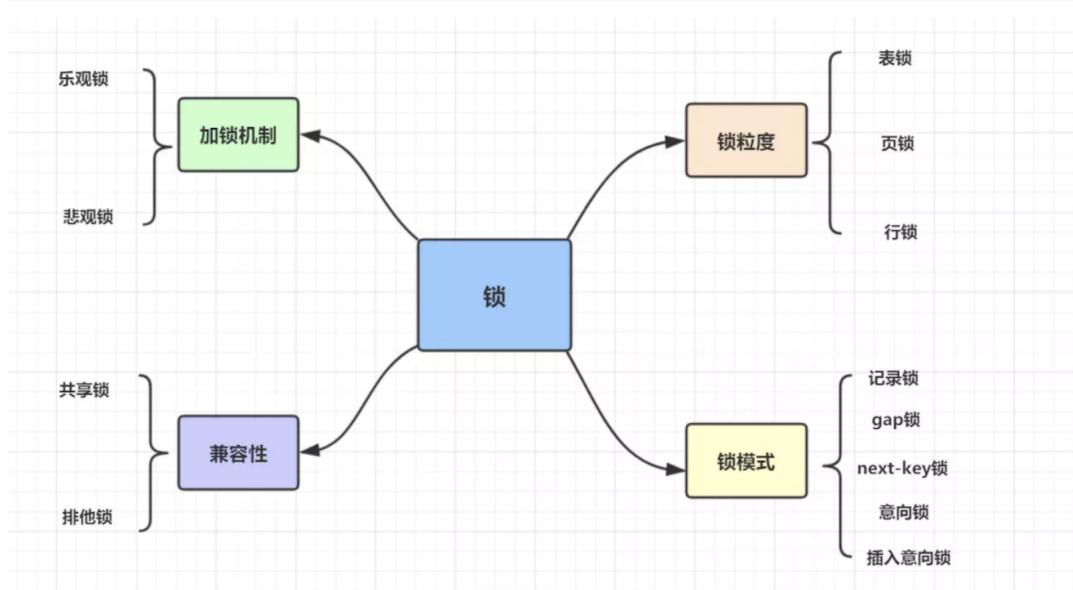
31. Blob和text有什么区别？

- Blob用于存储二进制数据，而Text用于存储大字符串。
- Blob值被视为二进制字符串（字节字符串），它们没有字符集，并且排序和比较基于列值中的字节的数值。
- text值被视为非二进制字符串（字符字符串）。它们有一个字符集，并根据字符集的排序规则对值进行排序和比较。

32. mysql里记录货币用什么字段类型比较好？

- 货币在数据库中MySQL常用Decimal和Numric类型表示，这两种类型被MySQL实现为同样的类型。他们被用于保存与金钱有关的数据。
- salary DECIMAL(9,2)，9(precision)代表将被用于存储值的总的小数位数，而2(scale)代表将被用于存储小数点后的位数。存储在salary列中的值的范围是从-9999999.99到9999999.99。
- DECIMAL和NUMERIC值作为字符串存储，而不是作为二进制浮点数，以便保存那些值的小数精度。

33. MySQL中有哪几种锁，列举一下？



如果按锁粒度划分，有以下3种：

- 表锁：开销小，加锁快；锁定力度大，发生锁冲突概率高，并发度最低；不会出现死锁。
- 行锁：开销大，加锁慢；会出现死锁；锁定粒度小，发生锁冲突的概率低，并发度高。
- 页锁：开销和加锁速度介于表锁和行锁之间；会出现死锁；锁定粒度介于表锁和行锁之间，并发度一般

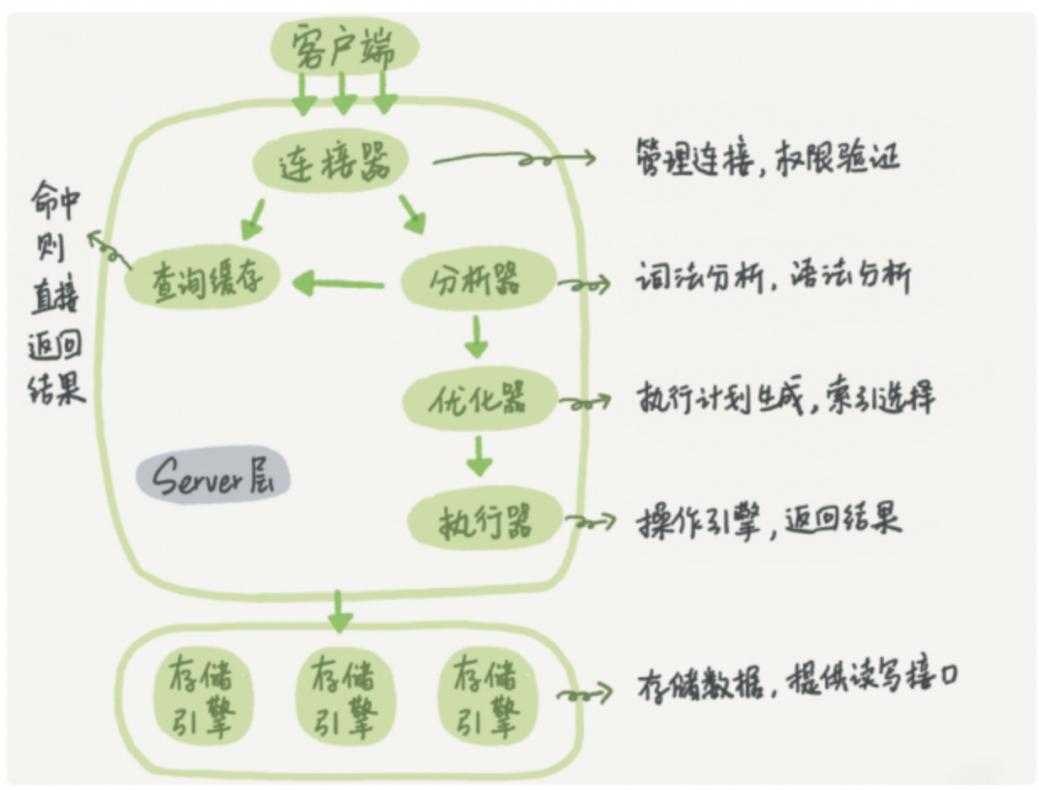
34. Hash索引和B+树区别是什么？你在设计索引是怎么抉择的？

- B+树可以进行范围查询，Hash索引不能。
- B+树支持联合索引的最左侧原则，Hash索引不支持。
- B+树支持order by排序，Hash索引不支持。
- Hash索引在等值查询上比B+树效率更高。
- B+树使用like进行模糊查询的时候，like后面（比如%开头）的话可以起到优化的作用，Hash索引根本无法进行模糊查询。

35. MySQL 的内连接、左连接、右连接有什么区别？

- Inner join 内连接，在两张表进行连接查询时，只保留两张表中完全匹配的结果集
- left join 在两张表进行连接查询时，会返回左表所有的行，即使在右表中没有匹配的记录。
- right join 在两张表进行连接查询时，会返回右表所有的行，即使在左表中没有匹配的记录。

36. 说说MySQL 的基础架构图



MySQL 逻辑架构图主要分三层：

- 第一层负责连接处理，授权认证，安全等等
- 第二层负责编译并优化SQL
- 第三层是存储引擎。

37. 什么是内连接、外连接、交叉连接、笛卡尔积呢？

- 内连接 (inner join)：取得两张表中满足存在连接匹配关系的记录。
- 外连接 (outer join)：取得两张表中满足存在连接匹配关系的记录，以及某张表（或两张表）中不满足匹配关系的记录。
- 交叉连接 (cross join)：显示两张表所有记录一一对应，没有匹配关系进行筛选，也被称为：笛卡尔积。

38. 说一下数据库的三大范式

- 第一范式：数据表中的每一列（每个字段）都不可以再拆分。
- 第二范式：在第一范式的基础上，分主键列完全依赖于主键，而不能是依赖于主键的一部分。
- 第三范式：在满足第二范式的基础上，表中的非主键只依赖于主键，而不依赖于其他非主键。

39. mysql有关权限的表有哪几个呢？

MySQL服务器通过权限表来控制用户对数据库的访问，权限表存放在mysql数据库里，由mysql_install_db脚本初始化。这些权限表分别user, db, table_priv, columns_priv和host。

- user权限表：记录允许连接到服务器的用户帐号信息，里面的权限是全局级的。
- db权限表：记录各个帐号在各个数据库上的操作权限。
- table_priv权限表：记录数据表级的操作权限。
- columns_priv权限表：记录数据列级的操作权限。
- host权限表：配合db权限表对给定主机上数据库级操作权限作更细致的控制。这个权限表不受GRANT和REVOKE语句的影响。

40. Mysql的binlog有几种录入格式？分别有什么区别？

有三种格式哈，statement, row和mixed。

- statement，每一条会修改数据的sql都会记录在binlog中。不需要记录每一行的变化，减少了binlog日志量，节约了IO，提高性能。由于sql的执行是有上下文的，因此在保存的时候需要保存相关的信息，同时还有一些使用了函数之类的语句无法被记录复制。
- row，不记录sql语句上下文相关信息，仅保存哪条记录被修改。记录单元为每一行的改动，基本是可以全部记下来但是由于很多操作，会导致大量行的改动(比如alter table)，因此这种模式的文件保存的信息太多，日志量太大。
- mixed，一种折中的方案，普通操作使用statement记录，当无法使用statement的时候使用row。

41. InnoDB引擎的4大特性，了解过吗

- 插入缓冲 (insert buffer)
- 二次写(double write)
- 自适应哈希索引(ahi)
- 预读(read ahead)

42. 索引有哪些优缺点？

优点：

- 唯一索引可以保证数据库表中每一行的数据的唯一性
- 索引可以加快数据查询速度，减少查询时间

缺点：

- 创建索引和维护索引要耗费时间
- 索引需要占物理空间，除了数据表占用数据空间之外，每一个索引还要占用一定的物理空间

- 以表中的数据进行增、删、改的时候，索引也要动态的维护。

43. 索引有哪几种类型？

- 主键索引：数据列不允许重复，不允许为NULL，一个表只能有一个主键。
- 唯一索引：数据列不允许重复，允许为NULL值，一个表允许多个列创建唯一索引。
- 普通索引：基本的索引类型，没有唯一性的限制，允许为NULL值。
- 全文索引：是目前搜索引擎使用的一种关键技术，对文本的内容进行分词、搜索。
- 覆盖索引：查询列要被所建的索引覆盖，不必读取数据行
- 组合索引：多列值组成一个索引，用于组合搜索，效率大于索引合并

44. 创建索引有什么原则呢？

- 最左前缀匹配原则
- 频繁作为查询条件的字段才去创建索引
- 频繁更新的字段不适合创建索引
- 索引列不能参与计算，不能有函数操作
- 优先考虑扩展索引，而不是新建索引，避免不必要的索引
- 在order by或者group by子句中，创建索引需要注意顺序
- 区分度低的数据列不适合做索引列(如性别)
- 定义有外键的数据列一定要建立索引。
- 对于定义为text、image数据类型的列不要建立索引。
- 删除不再使用或者很少使用的索引

45. 创建索引的三种方式

在执行CREATE TABLE时创建索引

```
1 CREATE TABLE `employee` (
2   `id` int(11) NOT NULL,
3   `name` varchar(255) DEFAULT NULL,
4   `age` int(11) DEFAULT NULL,
5   `date` datetime DEFAULT NULL,
6   `sex` int(1)
7   PRIMARY KEY (`id`),
8   KEY `idx_name` (`name`) USING BTREE
9 ) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

使用ALTER TABLE命令添加索引

```
1 ALTER TABLE table_name ADD INDEX index_name (column);
```

使用CREATE INDEX命令创建

```
1 CREATE INDEX index_name ON table_name (column);
```

46. 百万级别或以上的数据，你是如何删除的？

- 我们想要删除百万数据的时候可以先删除索引
- 然后批量删除其中无用数据
- 删完完成后重新创建索引。

47. 什么是最左前缀原则？什么是最左匹配原则？

- 最左前缀原则，就是最左优先，在创建多列索引时，要根据业务需求，where子句中使用最频繁的一列放在最左边。
- 当我们创建一个组合索引的时候，如(k1,k2,k3)，相当于创建了 (k1) 、(k1,k2) 和 (k1,k2,k3) 三个索引，这就是最左匹配原则。.

48. B树和B+树的区别，数据库为什么使用B+树而不是B树？

- 在B树中，键和值即存放在内部节点又存放在叶子节点；在B+树中，内部节点只存键，叶子节点则同时存放键和值。
- B+树的叶子节点有一条链相连，而B树的叶子节点各自独立的。
- B+树索引的所有数据均存储在叶子节点，而且数据是按照顺序排列的，链表连着的。那么B+树使得范围查找，排序查找，分组查找以及去重查找变得异常简单。.
- B+树非叶子节点上是不存储数据的，仅存储键值，而B树节点中不仅存储键值，也会存储数据。innodb中页的默认大小是16KB，如果不存储数据，那么就会存储更多的键值，相应的树的阶数（节点的子节点树）就会更大，树就会更矮更胖，如此一来我们查找数据进行磁盘的IO次数会再次减少，数据查询的效率也会更快。

49. 覆盖索引、回表等这些，了解过吗？

- 覆盖索引：查询列要被所建的索引覆盖，不必从数据表中读取，换句话说查询列要被所使用的索引覆盖。
- 回表：二级索引无法直接查询所有列的数据，所以通过二级索引查询到聚簇索引后，再查询到想要的数据，这种通过二级索引查询出来的过程，就叫做回表。

50. B+树在满足聚簇索引和覆盖索引的时候不需要回表查询数据？

- 在B+树的索引中，叶子节点可能存储了当前的key值，也可能存储了当前的key值以及整行的数据，这就是聚簇索引和非聚簇索引。在InnoDB中，只有主键索引是聚簇索引，如果没有主键，则挑选一个唯一键建立聚簇索引。如果没有唯一键，则隐式生成一个键来建立聚簇索引。

- 当查询使用聚簇索引时，在对应的叶子节点，可以获取到整行数据，因此不用再次进行回表查询。

51. 何时使用聚簇索引与非聚簇索引

动作描述	使用聚集索引	使用非聚集索引
列经常被分组排序	应	应
返回某范围内的数据	应	不应
一个或极少不同值	不应	不应
小数目的不同值	应	不应
大数目的不同值	不应	应
频繁更新的列	不应	应
外键列	应	应
主键列	应	应
频繁修改索引列	不应	应

52. 非聚簇索引一定会回表查询吗？

不一定，如果查询语句的字段全部命中了索引，那么就不必再进行回表查询（哈哈，覆盖索引就是这么回事）。

举个简单的例子，假设我们在学生表的上建立了索引，那么当进行 `select age from student where age < 20` 的查询时，在索引的叶子节点上，已经包含了age信息，不会再进行回表查询。

53. 组合索引是什么？为什么需要注意组合索引中的顺序？

组合索引，用户可以在多个列上建立索引，这种索引叫做组合索引。因为InnoDB引擎中的索引策略的最左原则，所以需要注意组合索引中的顺序。

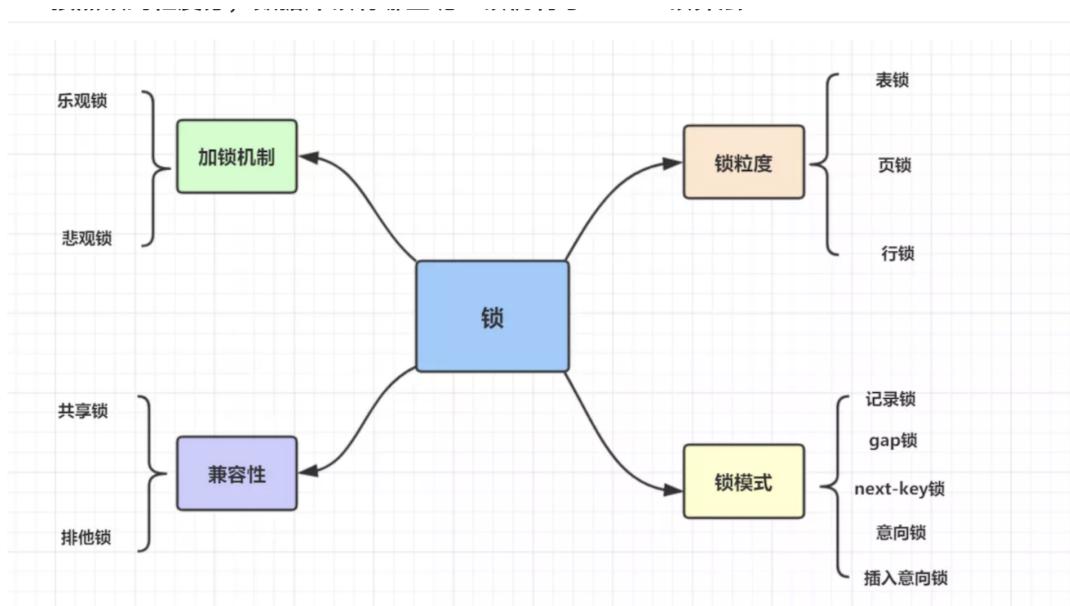
54. 什么是数据库事务？

数据库事务（简称：事务），是数据库管理系统执行过程中的一个逻辑单位，由一个有限的数据库操作序列构成，这些操作要么全部执行，要么全部不执行，是一个不可分割的工作单位。

55. 隔离级别与锁的关系

回答这个问题，可以先阐述四种隔离级别，再阐述它们的实现原理。隔离级别就是依赖锁和MVCC实现的。

56. 按照锁的粒度分，数据库锁有哪些呢？锁机制与InnoDB锁算法



- 按锁粒度分有：表锁，页锁，行锁
- 按锁机制分有：乐观锁，悲观锁

57. 从锁的类别角度讲，MySQL都有哪些锁呢？

从锁的类别上来讲，有共享锁和排他锁。

- 共享锁:** 又叫做读锁。当用户要进行数据的读取时，对数据加上共享锁。共享锁可以同时加上多个。
- 排他锁:** 又叫做写锁。当用户要进行数据的写入时，对数据加上排他锁。排他锁只可以加一个，他和其他的排他锁，共享锁都相斥。

锁兼容性如下：

兼容性	S	X
S	兼容	不兼容
X	不兼容	不兼容

58. MySQL中InnoDB引擎的行锁是怎么实现的？

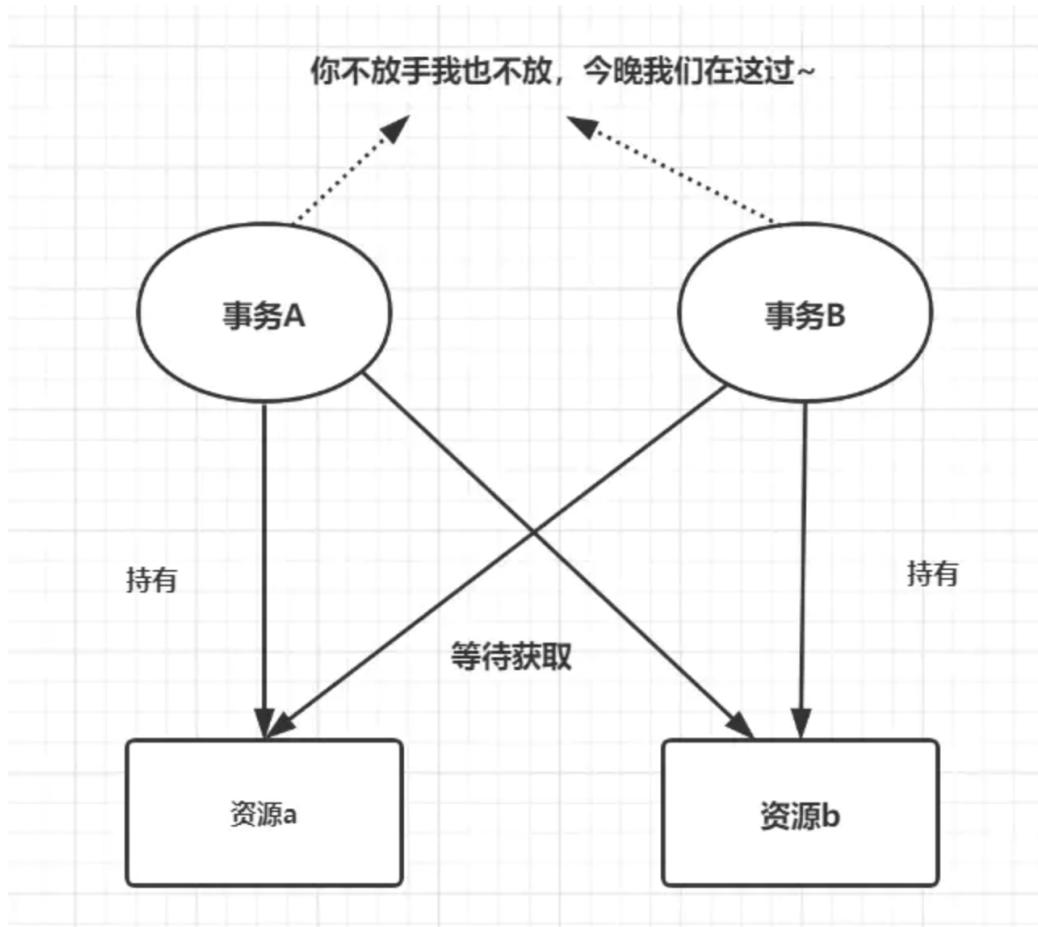
基于索引来完成行锁的。

```
1 select * from t where id = 666 for update;
```

for update 可以根据条件来完成行锁锁定，并且 id 是有索引键的列，如果 id 不是索引键那么InnoDB将实行表锁。

59. 什么是死锁？怎么解决？

死锁是指两个或多个事务在同一资源上相互占用，并请求锁定对方的资源，从而导致恶性循环的现象。看图形象一点，如下：



死锁有四个必要条件：互斥条件，请求和保持条件，环路等待条件，不剥夺条件。解决死锁思路，一般就是切断环路，尽量避免并发形成环路。

- 如果不同程序会并发存取多个表，尽量约定以相同的顺序访问表，可以大大降低死锁机会。
- 在同一个事务中，尽可能做到一次锁定所需要的所有资源，减少死锁产生概率；
- 对于非常容易产生死锁的业务部分，可以尝试使用升级锁定颗粒度，通过表级锁定来减少死锁产生的概率；
- 如果业务处理不好可以用分布式事务锁或者使用乐观锁
- 死锁与索引密不可分，解决索引问题，需要合理优化你的索引，

60. 为什么要使用视图？什么是视图？

为什么要使用视图？

为了提高复杂SQL语句的复用性和表操作的安全性，MySQL数据库管理系统提供了视图特性。

什么是视图？

视图是一个虚拟的表，是一个表中的数据经过某种筛选后的显示方式，视图由一个预定的查询select语句组成。

61. 视图有哪些特点？哪些使用场景？

视图特点：

- 视图的列可以来自不同的表，是表的抽象和在逻辑意义上建立的新关系。
- 视图是由基本表(实表)产生的表(虚表)。
- 视图的建立和删除不影响基本表。
- 对视图内容的更新(添加，删除和修改)直接影响基本表。
- 当视图来自多个基本表时，不允许添加和删除数据。

视图用途：简化sql查询，提高开发效率，兼容老的表结构。

视图的常见使用场景：

- 重用SQL语句；
- 简化复杂的SQL操作。
- 使用表的组成部分而不是整个表；
- 保护数据
- 更改数据格式和表示。视图可返回与底层表的表示和格式不同的数据。

62. 视图的优点，缺点，讲一下？

- 查询简单化。视图能简化用户的操作
- 数据安全性。视图使用户能以多种角度看待同一数据，能够对机密数据提供安全保护
- 逻辑数据独立性。视图对重构数据库提供了一定程度的逻辑独立性

63. count(1)、count(*) 与 count(列名) 的区别?

- count(*)包括了所有的列，相当于行数，在统计结果的时候，不会忽略列值为NULL
- count(1)包括了忽略所有列，用1代表代码行，在统计结果的时候，不会忽略列值为NULL
- count(列名)只包括列名那一列，在统计结果的时候，会忽略列值为空（这里的空不是只空字符串或者0，而是表示null）的计数，即某个字段值为NULL时，不统计。

64. 什么是游标?

游标提供了一种对从表中检索出的数据进行操作的灵活手段，就本质而言，游标实际上是一种能从包括多条数据记录的结果集中每次提取一条记录的机制。

65. 什么是存储过程? 有哪些优缺点?

存储过程，就是一些编译好了的SQL语句，这些SQL语句代码像一个方法一样实现一些功能（对单表或多表的增删改查），然后给这些代码块取一个名字，在用到这个功能的时候调用即可。

优点：

- 存储过程是一个预编译的代码块，执行效率比较高
- 存储过程在服务器端运行，减少客户端的压力
- 允许模块化程序设计，只需要创建一次过程，以后在程序中就可以调用该过程任意次，类似方法的复用 -一个存储过程替代大量T_SQL语句，可以降低网络通信量，提高通信速率
- 可以一定程度上确保数据安全

缺点：

- 调试麻烦
- 可移植性不灵活
- 重新编译问题

66. 什么是触发器? 触发器的使用场景有哪些?

触发器，指一段代码，当触发某个事件时，自动执行这些代码。

使用场景：

- 可以通过数据库中的相关表实现级联更改。
- 实时监控某张表中的某个字段的更改而需要做出相应的处理。
- 例如可以生成某些业务的编号。
- 注意不要滥用，否则会造成数据库及应用程序的维护困难。

67. MySQL中都有哪些触发器?

MySQL 数据库中有六种触发器：

- Before Insert
- After Insert
- Before Update
- After Update
- Before Delete
- After Delete

68. 超键、候选键、主键、外键分别是什么？

- 超键：在关系模式中，能唯一标识元组的属性集称为超键。
- 候选键：是最小超键，即没有冗余元素的超键。
- 主键：数据库表中对储存数据对象予以唯一和完整标识的数据列或属性的组合。一个数据列只能有一个主键，且主键的取值不能缺失，即不能为空值（Null）。
- 外键：在一个表中存在的另一个表的主键称此表的外键。。

69. SQL 约束有哪几种呢？

- NOT NULL: 约束字段的内容一定不能为NULL。
- UNIQUE: 约束字段唯一性，一个表允许有多个 Unique 约束。
- PRIMARY KEY: 约束字段唯一，不可重复，一个表只允许存在一个。
- FOREIGN KEY: 用于预防破坏表之间连接的动作，也能防止非法数据插入外键。
- CHECK: 用于控制字段的值范围。

70. 谈谈六种关联查询，使用场景。

- 交叉连接
- 内连接
- 外连接
- 联合查询
- 全连接
- 交叉连接

71. varchar(50)中50的涵义

字段最多存放 50 个字符 如 varchar(50) 和 varchar(200) 存储 "jay" 字符串所占空间是一样的，后者在排序时会消耗更多内存

72. mysql中int(20)和char(20)以及varchar(20)的区别

- int(20) 表示字段是int类型，显示长度是 20
- char(20)表示字段是固定长度字符串， 长度为 20
- varchar(20) 表示字段是可变长度字符串， 长度为 20

73. drop、delete与truncate的区别

	delete	truncate	drop
类型	DML	DDL	DDL
回滚	可回滚	不可回滚	不可回滚
删除内容	表结构还在， 删除表的全部或者一部分数据行	表结构还在， 删除表中的所有数据	从数据库中删除表， 所有的数据行， 索引和权限也会被删除
删除速度	删除速度慢， 逐行删除	删除速度快	删除速度最快

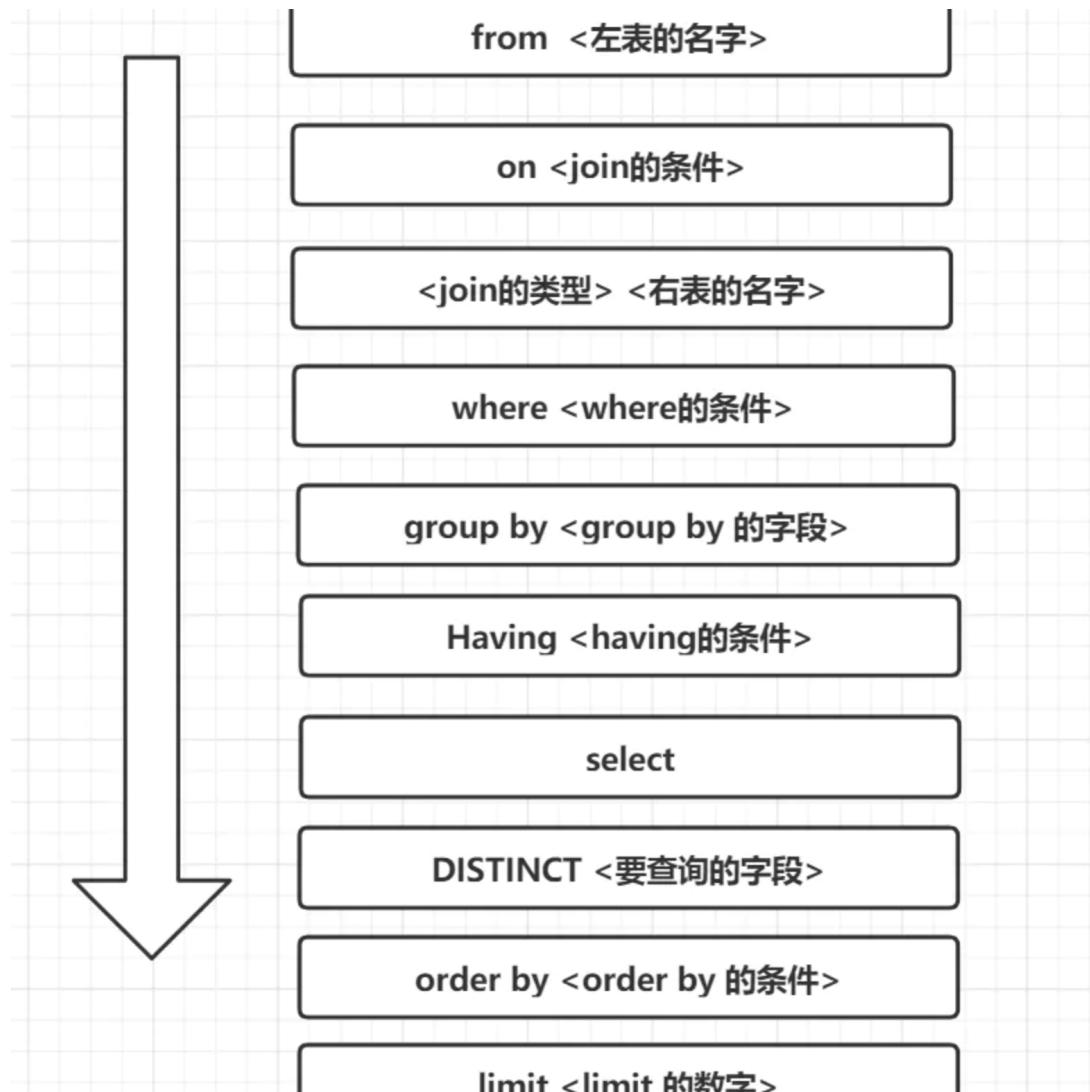
74. UNION与UNION ALL的区别?

- Union: 对两个结果集进行并集操作， 不包括重复行， 同时进行默认规则的排序；
- Union All: 对两个结果集进行并集操作， 包括重复行， 不进行排序；
- UNION的效率高于 UNION ALL

75. SQL的生命周期?

- 服务器与数据库建立连接
- 数据库进程拿到请求sql
- 解析并生成执行计划， 执行
- 读取数据到内存，并进行逻辑处理
- 通过步骤一的连接， 发送结果到客户端
- 关掉连接， 释放资源

76. 一条Sql的执行顺序?



77. 列值为NULL时，查询是否会用到索引？

列值为NULL也是可以走索引的。计划对列进行索引，应尽量避免把它设置为可空，因为这会让 MySQL 难以优化引用了可空列的查询，同时增加了引擎的复杂度。

78. 关心过业务系统里面的sql耗时吗？统计过慢查询吗？对慢查询都怎么优化过？

- 我们平时写SQL时，都要养成用explain分析的习惯。
- 慢查询的统计，运维会定期统计给我们。

优化慢查询：

- 分析语句，是否加载了不必要的字段/数据。
- 分析SQL执行句话，是否命中索引等。
- 如果SQL很复杂，优化SQL结构
- 如果表数据量太大，考虑分表

79. 主键使用自增ID还是UUID，为什么？

如果是单机的话，选择自增ID；如果是分布式系统，优先考虑UUID吧，但还是最好自己公司有一套分布式唯一ID生产方案吧。

- 自增ID：数据存储空间小，查询效率高。但是如果数据量过大，会超出自增长的值范围，多库合并，也有可能有问题。
- uuid：适合大量数据的插入和更新操作，但是它无序的，插入数据效率慢，占用空间大。

80. mysql自增主键用完了怎么办？

自增主键一般用int类型，一般达不到最大值，可以考虑提前分库分表的。

自增ID用完后 一直都是最大值 如果标识了主键 则主键冲突

81. 字段为什么要定义为not null？

null值会占用更多的字节，并且null有很多坑的。

82. 如果要存储用户的密码散列，应该使用什么字段进行存储？

密码散列，盐，用户身份证号等固定长度的字符串，应该使用char而不是varchar来存储，这样可以节省空间且提高检索效率。

83. Mysql驱动程序是什么？

这个jar包：mysql-connector-java-5.1.18.jar Mysql驱动程序主要帮助编程语言与MySQL服务端进行通信，如连接、传输数据、关闭等。

84. 如何优化长难的查询语句？有实战过吗？

- 将一个大的查询分为多个小的相同的查询
- 减少冗余记录的查询。
- 一个复杂查询可以考虑拆成多个简单查询
- 分解关联查询，让缓存的效率更高。

85. 优化特定类型的查询语句

平时积累吧：

- 比如使用select 具体字段代替 select *

- 使用count(*) 而不是count(列名)
- 在不影响业务的情况下，使用缓存
- explain 分析你的SQL

86. MySQL数据库cpu飙升的话，要怎么处理呢？

排查过程：

- 使用top 命令观察，确定是mysqld导致还是其他原因。
- 如果是mysqld导致的，show processlist，查看session情况，确定是不是有消耗资源的sql在运行。
- 找出消耗高的 sql，看看执行计划是否准确，索引是否缺失，数据量是否太大。

处理：

- kill 掉这些线程(同时观察 cpu 使用率是否下降)，
- 进行相应的调整(比如说加索引、改 sql、改内存参数)
- 重新跑这些 SQL。

其他情况：

也有可能是每个 sql 消耗资源并不多，但是突然之间，有大量的 session 连进来导致 cpu 飙升，这种情况就需要跟应用一起来分析为何连接数会激增，再做出相应的调整，比如说限制连接数等

87. 读写分离常见方案？

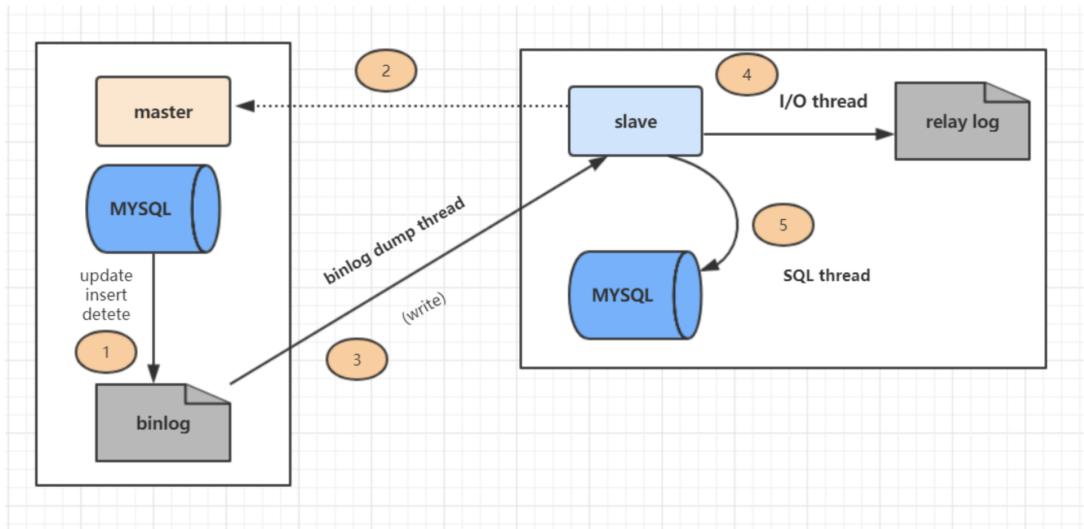
- 应用程序根据业务逻辑来判断，增删改等写操作命令发给主库，查询命令发给备库。
- 利用中间件来做代理，负责对数据库的请求识别出读还是写，并分发到不同的数据库中。 (如：amoeba, mysql-proxy)

88. MySQL的复制原理以及流程

主从复制原理，简言之，就三步曲，如下：

- 主数据库有个bin-log二进制文件，纪录了所有增删改Sql语句。 (binlog线程)
- 从数据库把主数据库的bin-log文件的sql语句复制过来。 (io线程)
- 从数据库的relay-log重做日志文件中再执行一次这些sql语句。 (Sql执行线程)

如下图所示：



上图主从复制分了五个步骤进行：

- 步骤一：主库的更新事件(update、insert、delete)被写到binlog
- 步骤二：从库发起连接，连接到主库。
- 步骤三：此时主库创建一个binlog dump thread，把binlog的内容发送到从库。
- 步骤四：从库启动之后，创建一个I/O线程，读取主库传过来的binlog内容并写入到relay log
- 步骤五：还会创建一个SQL线程，从relay log里面读取内容，从 Exec_Master_Log_Pos位置开始执行读取到的更新事件，将更新内容写入到slave 的db

89. MySQL中DATETIME和TIMESTAMP的区别

存储精度都为秒

区别：

- DATETIME 的日期范围是 1001——9999 年；TIMESTAMP 的时间范围是 1970 ——2038 年
- DATETIME 存储时间与时区无关；TIMESTAMP 存储时间与时区有关，显示的值也依赖于时区
- DATETIME 的存储空间为 8 字节；TIMESTAMP 的存储空间为 4 字节
- DATETIME 的默认值为 null；TIMESTAMP 的字段默认不为空(not null)，默认值为当前时间(CURRENT_TIMESTAMP)

90. InnoDB的事务实现原理？

- 原子性：是使用 undo log来实现的，如果事务执行过程中出错或者用户执行了 rollback，系统通过undo log日志返回事务开始的状态。
- 持久性：使用 redo log来实现，只要redo log日志持久化了，当系统崩溃，即可通过redo log把数据恢复。
- 隔离性：通过锁以及MVCC,使事务相互隔离开。
- 一致性：通过回滚、恢复，以及并发情况下的隔离性，从而实现一致性。

91. 谈谈MySQL的Explain

Explain 执行计划包含字段信息如下：分别是 id、select_type、table、partitions、type、possible_keys、key、key_len、ref、rows、filtered、Extra 等12个字段。 我们重点关注的是type，它的属性排序如下：

```
1 system > const > eq_ref > ref > ref_or_null >
2 index_merge > unique_subquery > index_subquery >
3 range > index > ALL
```

92. InnoDB的事务与日志的实现方式

有多少种日志

- innodb两种日志redo和undo。

日志的存放形式

- redo：在页修改的时候，先写到 redo log buffer 里面，然后写到 redo log 的文件系统缓存里面(fwrite)，然后再同步到磁盘文件（fsync）。
- Undo：在 MySQL5.5 之前，undo 只能存放在 ibdata文件里面，5.6 之后，可以通过设置 innodb_undo_tablespaces 参数把 undo log 存放在 ibdata之外。

事务是如何通过日志来实现的

- 因为事务在修改页时，要先记 undo，在记 undo 之前要记 undo 的 redo，然后修改数据页，再记数据页修改的 redo。Redo（其中包括 undo 的修改）一定要比数据页先持久化到磁盘。
- 当事务需要回滚时，因为有 undo，可以把数据页回滚到前镜像的状态，崩溃恢复时，如果 redo log 中事务没有对应的 commit 记录，那么需要用 undo 把该事务的修改回滚到事务开始之前。
- 如果有 commit 记录，就用 redo 前滚到该事务完成时并提交掉。

93. MySQL中TEXT数据类型的最大长度

- TINYTEXT: 256 bytes
- TEXT: 65,535 bytes(64kb)
- MEDIUMTEXT: 16,777,215 bytes(16MB)

- LONGTEXT: 4,294,967,295 bytes(4GB)

94. 500台db，在最快时间之内重启。

- 可以使用批量 ssh 工具 pssh 来对需要重启的机器执行重启命令。
- 也可以使用 salt (前提是客户端有安装 salt) 或者 ansible (ansible 只需要 ssh 免登通了就行) 等多线程工具同时操作多台服务

95. 你是如何监控你们的数据库的？你们的慢日志都是怎么查询的？

监控的工具有很多，例如zabbix, lepus, 我这里用的是lepus

96. 你是否做过主从一致性校验，如果有，怎么做的，如果没有，你打算怎么做？

主从一致性校验有多种工具 例如checksum、mysqldiff、pt-table-checksum等

97. 你们数据库是否支持emoji表情存储，如果不支持，如何操作？

更换字符集utf8-->utf8mb4

98. MySQL如何获取当前日期？

SELECT CURRENT_DATE();

99. 一个6亿的表a，一个3亿的表b，通过外键tid关联，你如何最快的查询出满足条件的第50000到第50200中的这200条数据记录。

1、如果A表TID是自增长,并且是连续的,B表的ID为索引

```
1 select * from a,b where a.tid = b.id and a.tid>500000 limit 200;
```

2、如果A表的TID不是连续的,那么就需要使用覆盖索引.TID要么是主键,要么是辅助索引,B表ID也需要有索引。

```
1 select * from b , (select tid from a limit 50000,200) a where b.id = a .tid;
```

100. Mysql一条SQL加锁分析

一条SQL加锁，可以分9种情况进行：

- 组合一：id列是主键，RC隔离级别
- 组合二：id列是二级唯一索引，RC隔离级别
- 组合三：id列是二级非唯一索引，RC隔离级别
- 组合四：id列上没有索引，RC隔离级别
- 组合五：id列是主键，RR隔离级别
- 组合六：id列是二级唯一索引，RR隔离级别
- 组合七：id列是二级非唯一索引，RR隔离级别
- 组合八：id列上没有索引，RR隔离级别
- 组合九：Serializable隔离级别

作者：Jay_huaxiao 链接：

<https://juejin.im/post/5ec15ab9f265da7bc60e1910>

面试必备算法

实现 Sunday 匹配

01、实现 strStr()

题目：实现 strStr()

实现 strStr() 函数。给定一个 haystack 字符串和一个 needle 字符串，在 haystack 字符串中找出 needle 字符串出现的第一个位置（从0开始）。如果不存在，则返回 -1。

示例 1：

输入: haystack = "hello", needle = "ll" 输出: 2

示例 2：

输入: haystack = "aaaaa", needle = "bba" 输出: -1

说明：

当 needle 是空字符串时，我们应当返回什么值呢？这是一个在面试中很好的问题。

对于本题而言，当 needle 是空字符串时我们应当返回 0。这与C语言的 strstr() 以及 Java的 indexOf() 定义相符。

02、Sunday 匹配

Sunday 算法是 Daniel M.Sunday 于1990年提出的字符串模式匹配。其核心思想是：在匹配过程中，模式串发现不匹配时，算法能跳过尽可能多的字符以进行下一步的匹配，从而提高了匹配效率。

因为该问是字符串匹配篇第一讲，所以先普及几个概念：

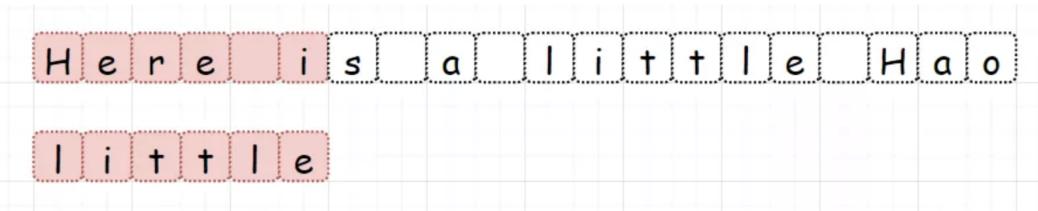
- 串：串是字符串的简称
- 空串：长度为零的串称为空串
- 主串：包含子串的串相应地称为主串
- 子串：串中任意个连续字符组成的子序列称为该串的子串
- 模式串：子串的定位运算又称为串的模式匹配，是一种求子串第一个字符在主串中序号的运算。被匹配的主串称为目标串，子串称为模式串。

了解这些基本概念，回到这个算法。Sunday匹配不是说这人在周末发现了这个算法，而是这人名字叫星期天（可能父母总加班，所以起了这么个名）。听起来牛叉的不得了，其实是个啥意思：

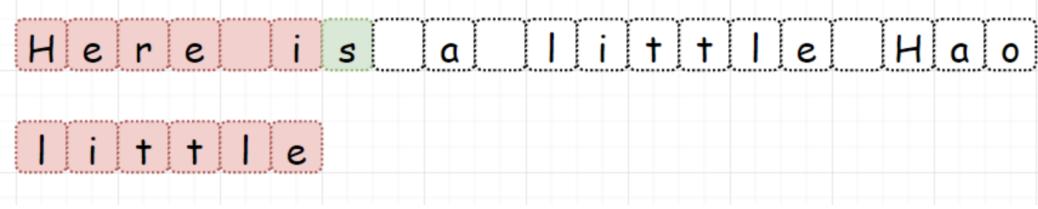
假若我们的目标串为：Here is a little Hao

模式串为：little

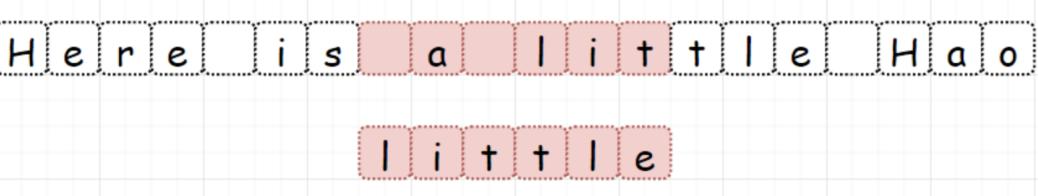
一般来讲，字符串匹配算法第一步，都是把目标串和模式串对齐。不管是KMP，BM，SUNDAY都是这样。



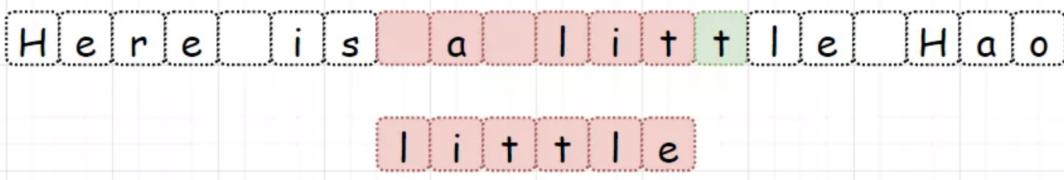
而对于SUNDAY算法，我们从头部开始比较，一旦发现不匹配，直接找到主串中位于模式串后面的第一个字符，即下面绿色的“s”。（这里说明一下，为什么是找模式串后面的第一个字符。在把模式串和目标串对齐后，如果发现不匹配，那肯定需要移动模式串。问题是需要移动多少步。各字符串匹配算法之间的差别也来自于这个地方，对于KMP，是建立部分匹配表来计算。BM，是反向比较计算移动量。对于SUNDAY，就是找到模式串后的第一个字符。因为，无论模式串移动多少步，模式串后的第一个字符都要参与下一次比较，也就是这里的“s”）



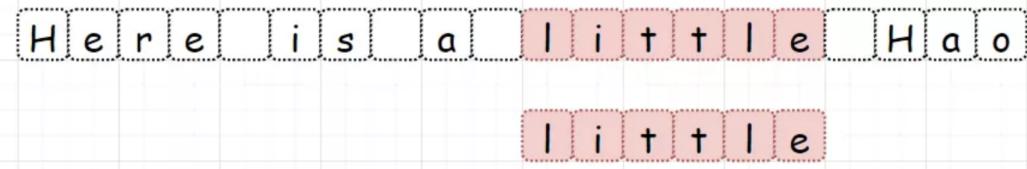
找到了模式串后的第一个字符“s”，接下来该怎么做？我们需要查看模式串中是否包含这个元素，如果不包含那就可以跳过一大片，从该字符的下一个字符开始比较。



因为仍然不匹配（空格和），我们继续重复上面的过程。找到模式串的下一个元素：t



现在有意思了，我们发现 t 被包含于模式串中，并且 t 出现在模式串倒数第3个。所以我们把模式串向前移动3个单位：



有内味了，我们发现竟然匹配成功了，是不是很神奇？证明的过程今天暂且不谈（后面我会出一个算法证明篇，来证明之前讲过的一些算法。我需要你做的是，掌握上面这些！）

捞干货，这个过程里我们做了一些什么：

- 对齐目标串和模式串，从前向后匹配
- 关注主串中位于模式串后面的第一个元素（核心）
- 如果关注的字符没有在子串中出现则直接跳过
- 否则开始移动模式串，移动位数 = 子串长度 - 该字符最右出现的位置(以0开始)

03、算法应用

自然是把这个算法应用到我们的题目中咯...

根据分析，得出代码：（给一个保证你能看的懂的Golang版本的）

```

1 func strStrSunday(haystack, needle string) int {
2     //先判断两个字符串的合法性
3     if len(haystack) < len(needle) {
4         return -1
5     }
6     if haystack == needle {
7         return 0
8     }
9     //定义最终位置的索引
10    index := -1
11    i := 0
12    //定义目标匹配索引
13    needleIndex := 0
14    for i < len(haystack) {
15        //逐字节判断是否相等
16        if haystack[i] == needle[needleIndex] {
17            //只有当index为-1时，说明是首次匹配到字符
18            if index == -1 {

```

```
19         index = i
20     }
21     //主串索引和模式串索引都自增
22     i++
23     needleIndex++
24     //判断是否完成匹配
25     if needleIndex >= len(needle) {
26         break
27     }
28     continue
29 }
30 //走到这里说明没有匹配成功，将匹配目标索引置为默认
31 index = -1
32 //计算主串需要移动的位置
33 i = i + len(needle) - needleIndex
34 //如果主串索引大于了主串实际长度则返回
35 if i >= len(haystack) {
36     return index
37 }
38 //计算下一个字符在模式串最右的位置
39 offset := 1
40 for j := len(needle) - 1; j > 0; j-- {
41     if haystack[i] == needle[j] {
42         offset = j
43         break
44     }
45 }
46 //将主串的索引左移指定长度，使当前的字符和模式串中最右匹配到的字符串对齐
47 i = i - offset
48 //将模式串的索引重置
49 needleIndex = 0
50 }
51 return index
52 }
```

反转字符串(301)

01、题目分析

第344题：反转字符串

编写一个函数，其作用是将输入的字符串反转过来。输入字符串以字符数组 `char[]` 的形式给出。不要给另外的数组分配额外的空间，你必须原地修改输入数组、使用 $O(1)$ 的额外空间解决这一问题。

示例 1：

输入：["h", "e", "l", "l", "o"] 输出：["o", "l", "l", "e", "h"]

示例 2：

输入：["H", "a", "n", "n", "a", "h"] 输出：["h", "a", "n", "n", "a", "H"]

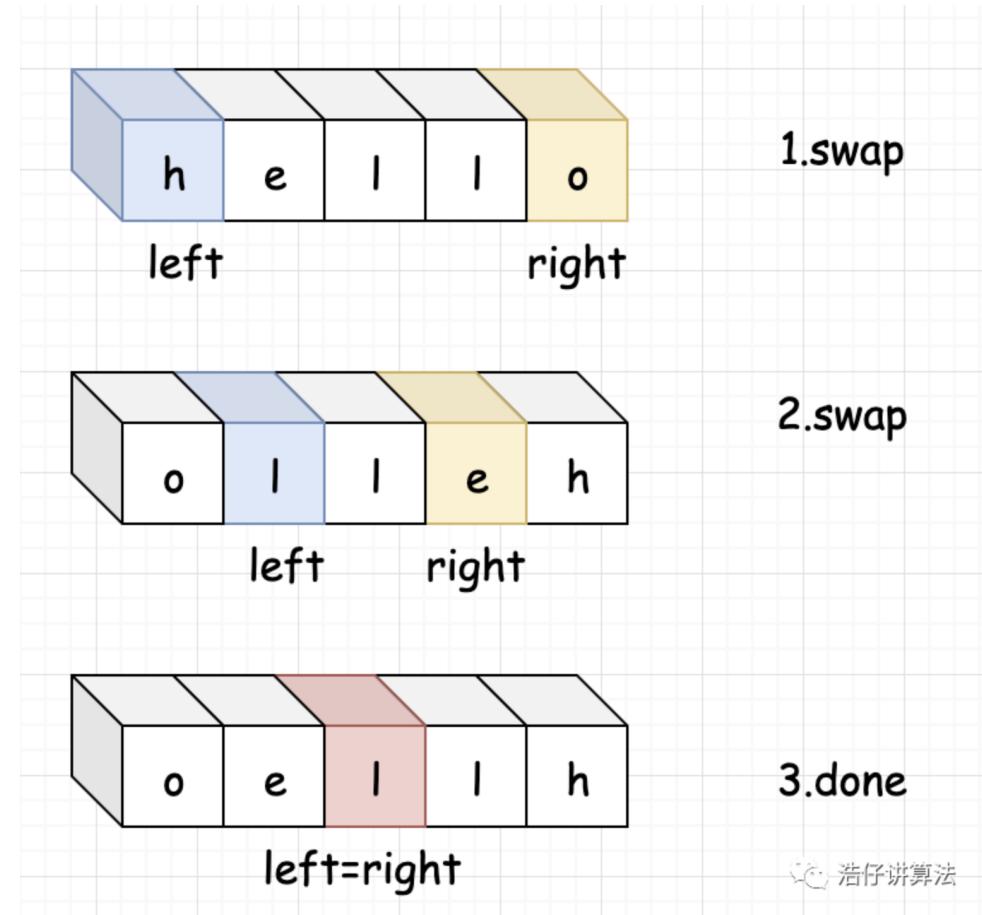
02、题目图解

这是一道相当简单的经典题目，直接上题解：使用双指针进行反转字符串。

假设输入字符串为 ["h", "e", "l", "l", "o"]

- 定义 left 和 right 分别指向首元素和尾元素
- 当 $\text{left} < \text{right}$ ，进行交换。
- 交换完毕， $\text{left}++, \text{right}--$
- 直至 $\text{left} == \text{right}$

具体过程如下图所示：



03、Go语言示例

根据以上分析，我们可以得到下面的题解：

```
1 func Reverse(s []byte) {  
2     right := len(s) - 1
```

```
3     left := 0
4
5     for left < right {
6         s[left], s[right] = s[right], s[left]
7         left++
8         right--
9     }
10 }
```

原文

反转字符串(301)

字符串中的第一个唯一字符

01、题目分析

第387题：字符串中的第一个唯一字符

给定一个字符串，找到它的第一个不重复的字符，并返回它的索引。如果不存在，则返回 -1。案例：

s = "leetcode" 返回 0.

s = "loveleetcode", 返回 2.

注意事项：您可以假定该字符串只包含小写字母。

常考题目，建议自行思考 1–2 分钟先～

02、题目图解

题目不难，直接进行分析。由于字母共有 26 个，所以我们可以声明一个 26 个长度的数组（该种方法在本类题型很常用）因为字符串中字母可能是重复的，所以我们可以先进行第一次遍历，在数组中记录每个字母的最后一次出现的所在索引。然后再通过一次循环，比较各个字母第一次出现的索引是否为最后一次的索引。如果是，我们就找到了我们的目标，如果不是我们将其设为 -1（标示该元素非目标元素）如果第二次遍历最终没有找到目标，直接返回 -1 即可。

图解如下：

people

第一次：

p	0	p	0	p	0	p	3
		e	1	e	1	e	5
				o	2	o	2
						l	4
						

第二次：

p	-1	p	-1	p	-1
e	5	e	-1	e	-1
o	2	o	2	o	2
l	4	l	4	l	4

0所在的元素位置
即为目标元素

注：上图省略了未出现的元素

03、GO语言示例

根据以上分析，可以得到代码如下：

```
1 func firstUniqueChar(s string) int {
2     var arr [26]int
3     //第一次遍历计算所有字符出现的最后位置
4     for i, k := range s {
5         //因为26个字母从a开始，减去a则索引会从0开始
6         arr[k-'a'] = i
7     }
8     for i, k := range s {
9         if arr[k-'a'] == i {
10             return i
11         }
12     }
13     return -1
14 }
```

验证回文串

01、题目示例

见微知著，发现一组数据很有趣，分享给大家。leetcode 第一题通过次数为 993,335，第二题通过次数为 396,160，第三题通过次数为 69,508。我想说什么，请自己悟。

第125题：验证回文串

给定一个字符串，验证它是否是回文串，只考虑字母和数字字符，可以忽略字母的大小写。

说明：本题中，我们将空字符串定义为有效的回文串。

示例 1: 输入: "A man, a plan, a canal: Panama" 输出: true

示例 2: 输入: "race a car" 输出: false

02、图解教程

经典题目，你需要像掌握反转字符串一样掌握本题。

首先，我想确保你知道什么是回文串。“回文串”是一个正读和反读都一样的字符串，比如“level”或者“noon”等等就是回文串。

对于字符串中可能存在的其他字符，可以通过正则替换，但是正则替换会增加程序运行复杂度，下面给出的是在判断过程中忽略其他字符：

```
1 func isPalindrome(s string) bool {
2     if s == "" {
3         return false
4     }
5     s = strings.ToLower(s)
6     if len(s) == 2 {
7         return s[0] == s[1]
8     }
9     left := 0
10    right := len(s) - 1
11    for left < right {
12        //忽略除字母和数字之外的字符
13        if !((s[left] >= 'a' && s[left] <= 'z') || (s[left] >= '0' &&
14        s[left] <= '9')) {
15            left++
16            continue
17        }
18        if !((s[right] >= 'a' && s[right] <= 'z') || (s[right] >= '0' &&
19        s[right] <= '9')) {
20            right--
21            continue
22        }
23        if s[left] != s[right] {
24            return false
25        }
26        left++
27        right--
```

```
27     return true  
28 }
```

滑动窗口最大值

01、题目分析

第239题：滑动窗口最大值

给定一个数组 `nums`, 有一个大小为 `k` 的滑动窗口从数组的最左侧移动到数组的最右侧。你只可以看到在滑动窗口内的 `k` 个数字。滑动窗口每次只向右移动一位。返回滑动窗口中的最大值。

返回滑动窗口中的最大值所构成的数组。

示例:

输入: `nums = [1,3,-1,-3,5,3,6,7]`, 和 `k = 3` 输出: `[3,3,5,5,6,7]` 解释:

滑动窗口的位置 最大值

```
[1 3 -1] -3 5 3 6 7 3  
1 [3 -1 -3] 5 3 6 7 3  
1 3 [-1 -3 5] 3 6 7 5  
1 3 -1 [-3 5 3] 6 7 5  
1 3 -1 -3 [5 3 6] 7 6  
1 3 -1 -3 5 [3 6 7] 7
```

02、题目分析

本题对于题目没有太多需要额外说明的，应该都能理解，直接进行分析。我们很容易想到，可以通过遍历所有的滑动窗口，找到每一个窗口的最大值，来进行暴力求解。那一共有多少个滑动窗口呢，小学题目，可以得到共有 $L-k+1$ 个窗口。

假设 `nums = [1,3,-1,-3,5,3,6,7]`，和 `k = 3`，窗口数为 `6`：

1	3	-1	-3	5	3	6	7
1	3	-1	-3	5	3	6	7
1	3	-1	-3	5	3	6	7
1	3	-1	-3	5	3	6	7
1	3	-1	-3	5	3	6	7
1	3	-1	-3	5	3	6	7

根据分析，直接完成代码：

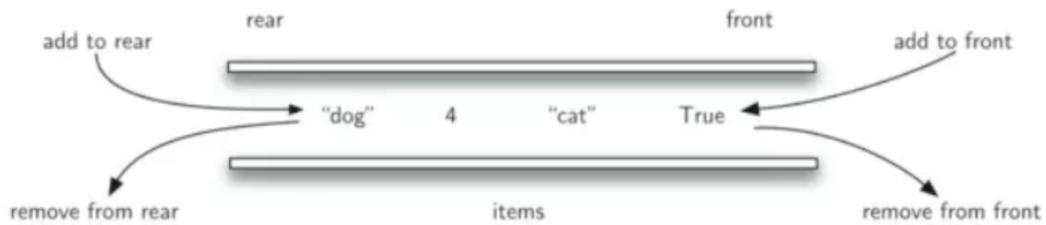
```

1 func maxSlidingWindow(nums []int, k int) []int {
2     l1 := len(nums)
3     index := 0
4     ret := make([]int, 0)
5     for index < l1 {
6         m := nums[index]
7         if index > l1 - k {
8             break
9         }
10        for j := index + 1; j < index + k; j++ {
11            if m < nums[j] {
12                m = nums[j]
13            }
14        }
15        ret = append(ret,m)
16        index++
17    }
18    return ret
19 }
```

03、线性题解

这里不卖关子，其实这道题比较经典，我们可以采用队列，DP，堆等方式进行求解，所有思路的主要源头应该都是在窗口滑动的过程中，如何更快的完成查找最大值的过程。但是最典型的解法还是使用双端队列。具体怎么来求解，一起看一下。

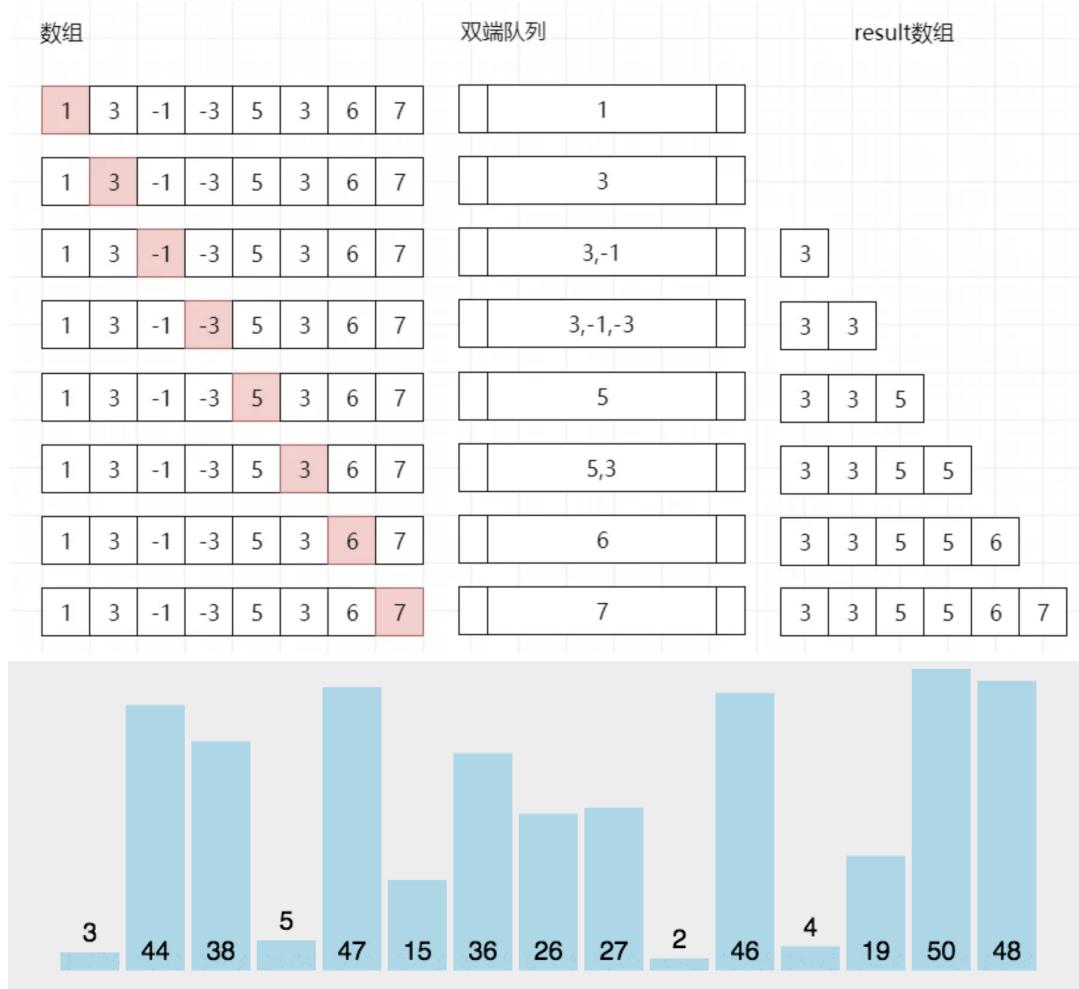
首先，我们了解一下，什么是双端队列：是一种具有队列和栈的性质的数据结构。双端队列中的元素可以从两端弹出或者插入。

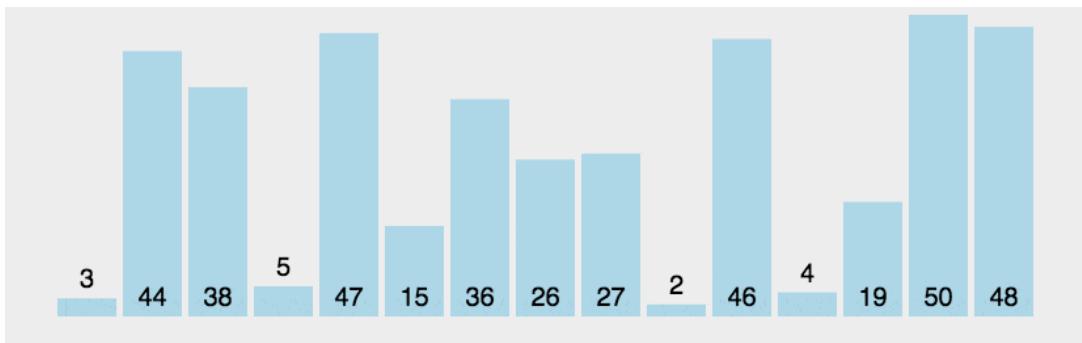


我们可以利用双端队列来实现一个窗口，目的是让该窗口可以做到张弛有度（汉语博大精深，也就是长度动态变化。其实用游标或者其他解法的目的都是一样的，就是去维护一个可变长的窗口）

然后我们再做一件事，只要遍历该数组，同时**在双端队列的头去维护当前窗口的最大值（在遍历过程中，发现当前元素比队列中的元素大，就将原来队列中的元素祭天），在整个遍历的过程中我们再记录下每一个窗口的最大值到结果数组中。**最终结果数组就是我们想要的，整体图解如下。

假设 `nums = [1,3,-1,-3,5,3,6,7]`， 和 `k = 3`：





根据分析，得出代码：

```

1 func maxSlidingWindow2(nums []int, k int) []int {
2     ret := make([]int,0)
3     if len(nums) == 0 {
4         return ret
5     }
6     var queue []int
7     for i := range nums {
8         for i > 0 && (len(queue) > 0) && nums[i] > queue[len(queue)-1] {
9             //将比当前元素小的元素祭天
10            queue = queue[:len(queue)-1]
11        }
12        //将当前元素放入queue中
13        queue = append(queue, nums[i])
14        if i >= k && nums[i-k] == queue[0] {
15            //维护队列，保证其头元素为当前窗口最大值
16            queue = queue[1:]
17        }
18        if i >= k-1 {
19            //放入结果数组
20            ret = append(ret, queue[0])
21        }
22    }
23    return ret
24 }
```

排序算法

冒泡排序

冒泡排序（Bubble Sort）也是一种简单直观的排序算法。它重复地走访过要排序的数列，一次比较两个元素，如果他们的顺序错误就把他们交换过来。走访数列的工作是重复地进行直到没有再需要交换，也就是说该数列已经排序完成。这个算法的名字由来是因为越小的元素会经由交换慢慢“浮”到数列的顶端。

作为最简单的排序算法之一，冒泡排序给我的感觉就像 Abandon 在单词书里出现的感觉一样，每次都在第一页第一位，所以最熟悉。冒泡排序还有一种优化算法，就是立一

个 flag，当在一趟序列遍历中元素没有发生交换，则证明该序列已经有序。但这种改进对于提升性能来说并没有什么太大作用。

1. 算法步骤

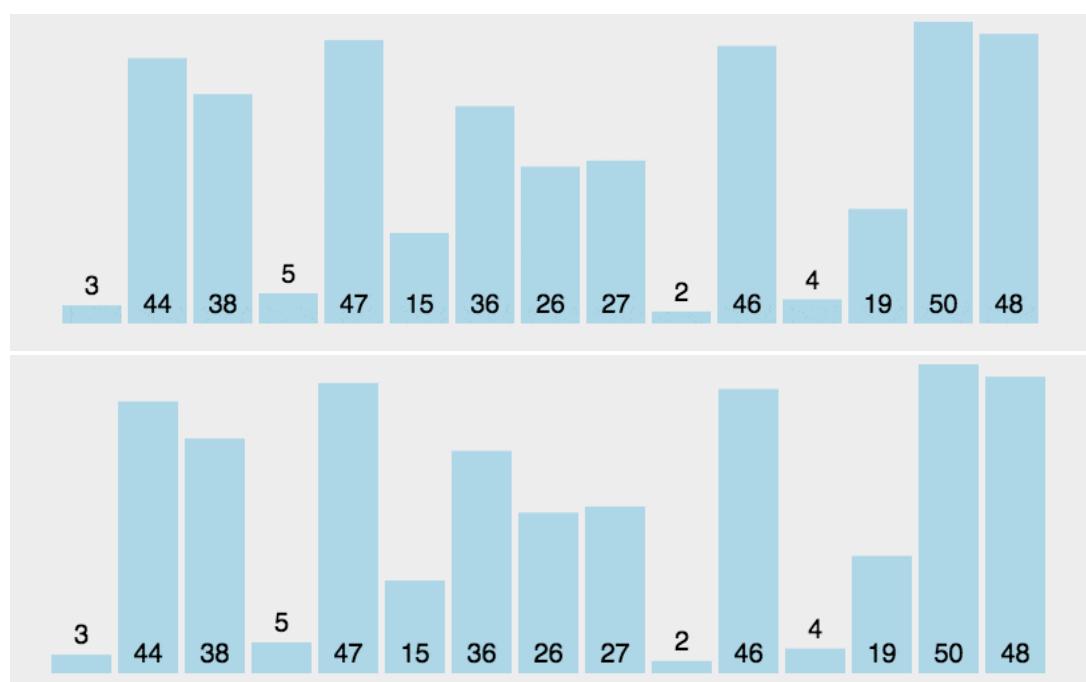
比较相邻的元素。如果第一个比第二个大，就交换他们两个。

对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对。这步做完后，最后的元素会是最大的数。

针对所有的元素重复以上的步骤，除了最后一个。

持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较。

2. 动图演示



3. 最慢和最快

正序时最快，反序时最慢

Golang实现

```
1 func bubbleSort(arr []int) []int {
2     if len(arr) == 0 {
3         return arr
```

```

4 }
5     for i := 0; i < len(arr); i++ {
6         for j := 0; j < len(arr); j++ {
7             if arr[i] > arr[j] {
8                 arr[j], arr[i] = arr[i], arr[j]
9             }
10        }
11    }
12    return arr
13 }
```

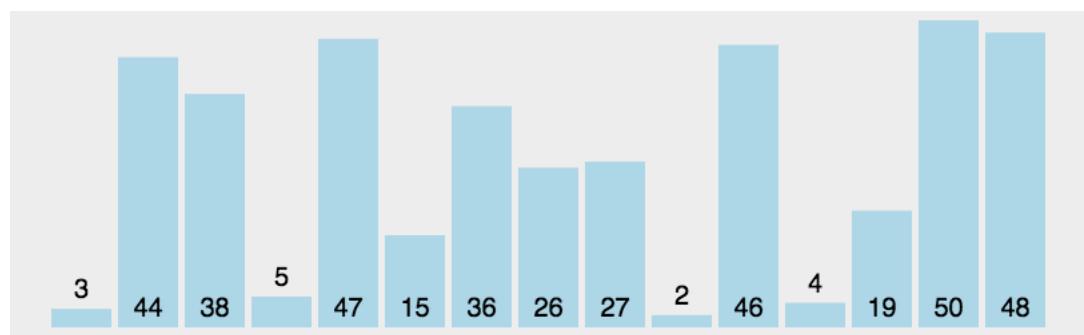
选择排序

选择排序是一种简单直观的排序算法，无论什么数据进去都是 $O(n^2)$ 的时间复杂度。所以用到它的时候，数据规模越小越好。唯一的好处可能就是不占用额外的内存空间了吧。

1. 算法步骤

- 首先在未排序序列中找到最小（大）元素，存放到排序序列的起始位置
- 再从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾。
- 重复第二步，直到所有元素均排序完毕。

2. 动图演示



3. Go 代码实现

```

1 func selectionSort(arr []int) []int {
2     l := len(arr)
3     if l == 0 {
4         return arr
5     }
6     for i := 0; i < l; i++ {
7         min := i
8         for j := i + 1; j < l; j++ {
```

```
9     if arr[j] < arr[min] {  
10         min = j  
11     }  
12 }  
13 arr[i],arr[min] = arr[min],arr[i]  
14 }  
15 return arr  
16 }
```