notes

发布 *1.0*

2020 年 11 月 23 日

Contents

CHAPTER 1

Mongodb

1.1 基本操作

1.1.1 索引操作

```
# 创建索引
db.collection_name.ensureIndex({"key_name":1})

# 创建唯一索引
db.collection_name.ensureIndex({"key_name":1},{"unique":true})

# 复合唯一索引
db.collection_name.ensureIndex({'key1':1, 'key1':1})

# 10s 后自动删除
db.collection_name.ensureIndex({"key_name":1},{expireAfterSeconds:10})
```

1.1.2 增删改查

```
# 查找
db.getCollection('collection_name').find({"provison_state": "success"})
```

(下页继续)

(续上页)

```
# 使用正则表达式
db.getCollection('collection_name').find({"date": /2020-01-/})

# range
db.getCollection('collection_name').find({"date": {"$gt": ISODate("2020-07-18T00:00:00.
↪000Z")}})

# 删除文档
db.getCollection('collection_name').remove({"provison_state": "success", "industry":␣
↪null})

# 删除集合中所有文档
db.getCollection('collection_name').remove({})
```

1.1.3 更新子文档

1.1.4 通过 id 获取时间

mongodb 的 Objectid 由 12 字节构成，分别是:

- a 4-byte value representing the seconds since the Unix epoch,
- a 3-byte machine identifier,
- a 2-byte process id, and
- a 3-byte counter, starting with a random value.

```
ObjectId("567a68517507b377a0a20903").getTimestamp()
```

1.1.5 分页

```
# Page 1
db.getCollection('collection_name').find({}).limit(5)

# Page 2
db.getCollection('collection_name').find({}).skip(5).limit(5)

# Page 3
db.getCollection('collection_name').find({}).skip(10).limit(5)
```

1.2 分片

CHAPTER 2

Redis

2.1 主从同步

```
# 如果主服务设置了密码
config set masterauth <pwd>

# 设置主服务器
SLAVEOF 192.168.1.100 6379

# 取消同步
SLAVEOF NO ONE
```

CHAPTER 3

mysql

3.1 常用操作

3.1.1 Host 授权

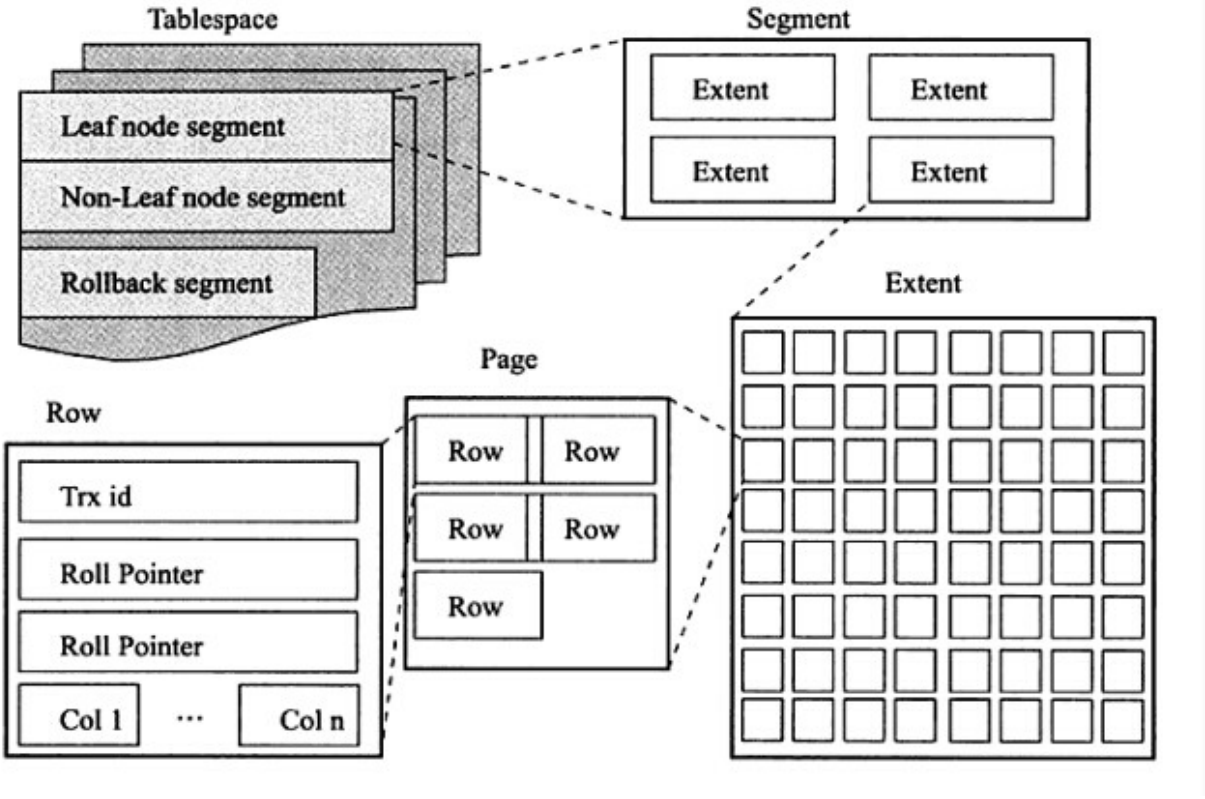
```
-- 通配符设置网段
grant all privileges on <db_name>.* to root@'10.10.10.%' identified by '<pwd>';

grant all privileges on *.* to root@'10.10.10.100' identified by '<pwd>';
flush privileges;
```

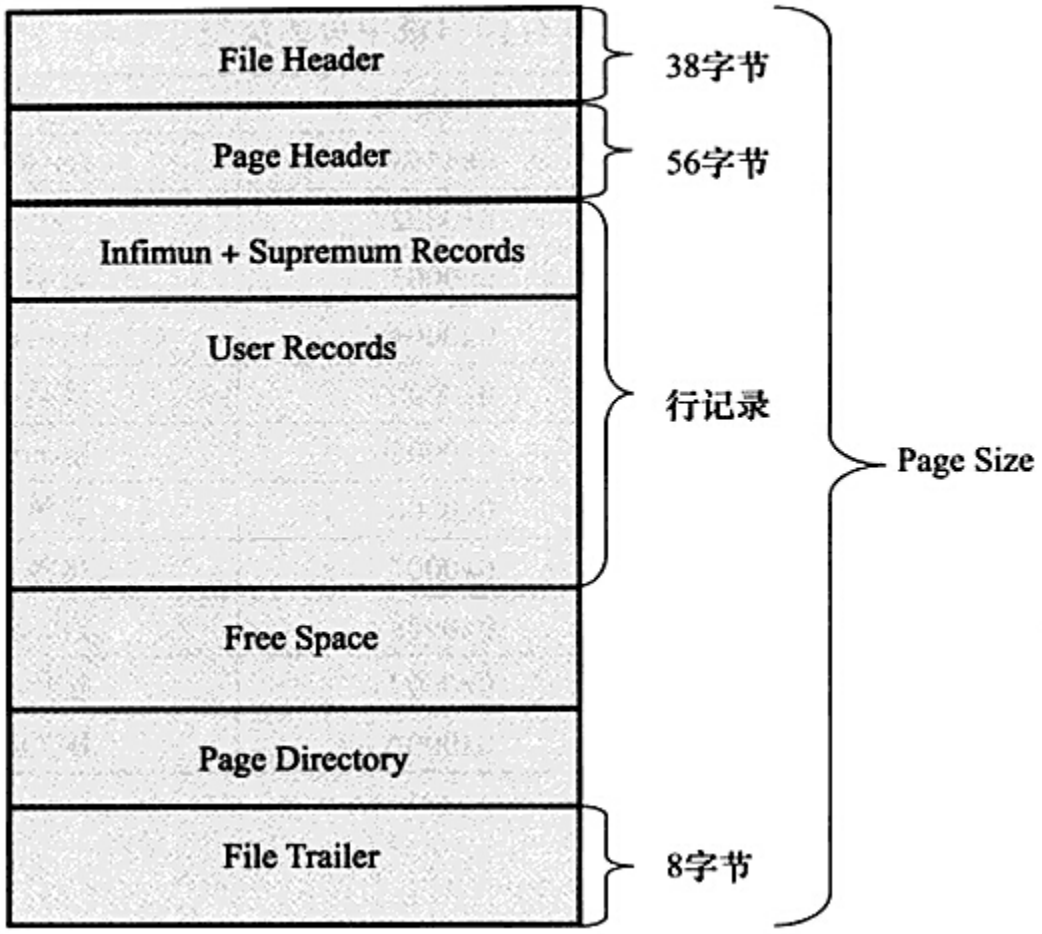
3.2 存储引擎

3.2.1 InnoDb

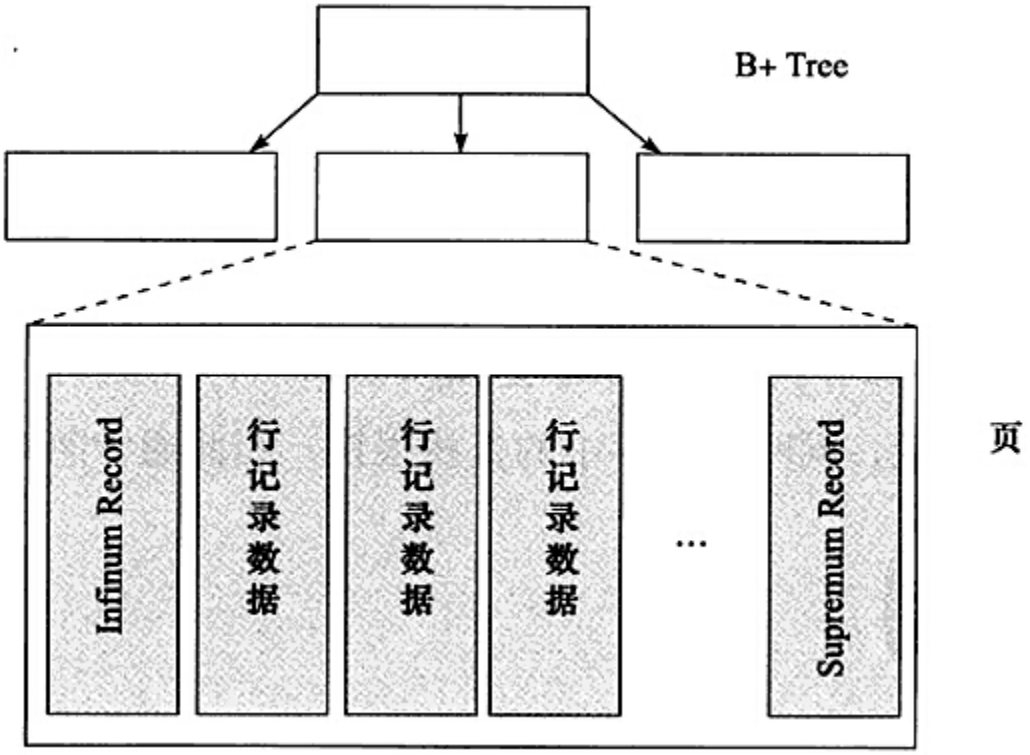
页（Page）是 Innodb 存储引擎用于管理数据的最小磁盘单位。常见的页类型有数据页、Undo 页、系统页、事务数据页等，本文主要分析的是数据页。默认的页大小为 16KB，每个页中至少存储有 2 条或以上的行记录，本文主要分析的是页与行记录的数据结构，有关索引和 B-tree 的部分在后续文章中介绍。

下图是 Innodb 逻辑存储结构图，从上往下依次为：Tablespace、Segment、Extent、Page 以及 Row。本文关注的重点是 Page 和 Row 的数据结构。


Tablespace
Leaf node segment
Non-Leaf node segment
Rollback segment
Segment
Extent
Extent
Extent
Extent
Extent
Page
Row
Row
Row
Row
Row
Row
Trx id
Roll Pointer
Roll Pointer
Col 1
...
Col n



上图为 Page 数据结构，File Header 字段用于记录 Page 的头信息，其中比较重要的是 FIL_PAGE_PREV 和 FIL_PAGE_NEXT 字段，通过这两个字段，我们可以找到该页的上一页和下一页，实际上所有页通过两个字段可以形成一条双向链表。Page Header 字段用于记录 Page 的状态信息。接下来的 Infimum 和 Supremum 是两个伪行记录，Infimum（下确界）记录比该页中任何主键值都要小的值，Supremum（上确界）记录比该页中任何主键值都要大的值，这个伪记录分别构成了页中记录的边界。



User Records 中存放的是实际的数据行记录，具体的行记录结构将在本文的第二节中详细介绍。Free Space 中存放的是空闲空间，被删除的行记录会被记录成空闲空间。Page Directory 记录着与二叉查找相关的信息。File Trailer 存储用于检测数据完整性的校验和等数据。

行格式

Innodb 存储引擎提供了两种格式的行记录：Compact 和 Redundant。

COMPACT 行格式

Redundant 行记录

3.3 索引

3.4 锁

CHAPTER 4

Elasticsearch

4.1 聚合

```
# 按 key 排序
"total_use": {
    "terms": {
        "field": "total_use",
        "size": 100,
        "order": {
            "_term": "desc"
        }
    }
}
```

4.2 分片

4.2.1 分片大小设计规则

按照 20-50G 一个分片划分比较合适。

1. 查看分片分布情况

   http://localhost:9200/test_index/_search_shards

4.3 reindex

CHAPTER 5

操作系统

5.1 磁盘

页

CHAPTER 6

K8s

6.1 deployment

CHAPTER 7

Etcd

7.1 基本语法

7.1.1 术语

1. Node / 节点
2. Member / 成员

7.1.2 语法

这里全部用 V3 版本

```
# 设置 api 版本
export ETCDCTL_API=3

# 设置 key
etcdctl put /monitor_services/thalassa/uri/relay_exist 0.5

# 获取 key 对应的值
etcdctl get /monitor_services/thalassa/uri/relay_exist

# 只打印值，不打印 key
```

(下页继续)

(续上页)

```
etcdctl get /monitor_services/thalassa/uri/relay_exist --print-value-only

# 匹配 key 列表
etcdctl  get /monitor_services/thalassa/uri/ --prefix --keys-only

# 监控 key 是否发生变化
etcdctl watch foo

# 监控匹配 key
etcdctl watch --prefix foo
```

7.2 集群管理

```
# 集群状态
etcdctl member list
```

CHAPTER 8

消息队列

8.1 如何选择消息队列

8.1.1 rabbitmq

优点

1. 开源，流行；
2. 有 Exchange 模块，支持非常灵活的路由配置；
3. 支持的编程语言很多；

缺点

1. rabbitmq 对消息积压的支持并不好，在它的设计理念里面，消息队列是一个管道，大量消息积压会导致性能急剧下降；
2. 性能比较差，每秒可以处理几万到十几万的消息；
3. 使用 Erlang 编写，比较小众；

8.1.2 RocketMQ

优点

1. 性能比 rabbitmq 高一个数量级，每秒大概能处理几十万条消息；

缺点

1. 国产消息队列，知名度比较低；

8.1.3 Kafka

Kafka 与周边生态系统的兼容性是最好的，没有之一；

Kafka 并不太适合在线业务场景；

优点

1. 性能比 rabbitmq 高一个数量级，每秒大概能处理几十万条消息；

8.2 rabbitmq

8.2.1 消息确认机制

1. 确认消息是否发送到 broker;
2. 确认消息是否成功消费;

RabbitMQ 为我们提供了两种方式：

通过 AMQP 事务机制实现，这也是 AMQP 协议层面提供的解决方案；通过将 channel 设置成 confirm 模式来实现；

事务机制

RabbitMQ 中与事务机制有关的方法有三个：txSelect(), txCommit() 以及 txRollback(), txSelect 用于将当前 channel 设置成 transaction 模式，txCommit 用于提交事务，txRollback 用于回滚事务，在通过 txSelect 开启事务之后，我们便可以发布消息给 broker 代理服务器了，如果 txCommit 提交成功了，则消息一定到达了 broker 了，如果在 txCommit 执行之前 broker 异常崩溃或者由于其他原因抛出异常，这个时候我们便可以捕获异常通过 txRollback 回滚事务了。

Confirm 模式

使用事物可以确认消息是否真的到达 broker, 但是会影响系统的吞吐量，使用 Confirm 可以解决这一问题。

8.3 kafka

8.4 celery

8.4.1 配置

从 4.0 版本开始，celery 使用小写下划线连接方式命令配置项；

1. 保存 task 执行状态和结果

   配置 *result_backend*, 保存的结果格式如下：

   正常时的消息

```
{
    "status": "SUCCESS",
    "result": 10,
    "traceback": null,
    "children": [],
    "task_id": "20fb6fb0-0ef2-4a2f-9517-2fbb5e41e443",
    "date_done": "2020-09-05T07:40:18.085679"
}
```

   异常时的消息

```
{
    "status": "FAILURE",
    "result": {
            "exc_type": "ZeroDivisionError",
            "exc_message": [
                    "division by zero"
            ],
            "exc_module": "builtins"
    },
    "traceback": "Traceback (most recent call last):\n  File \"/Users/sealee/.pyenv/
↪versions/3.8.1/envs/tianhe/lib/python3.8/site-packages/celery/app/trace.py\",␣
↪line 385, in trace_task\n    R = retval = fun(*args, **kwargs)\n  File \"/Users/
↪sealee/.pyenv/versions/3.8.1/envs/tianhe/lib/python3.8/site-packages/celery/app/
↪trace.py\", line 648, in __protected_call__\n    return self.run(*args,␣
↪**kwargs)\n  File \"/Users/sealee/code/mq/consumer.py\", line 13, in add\n    1/␣
↪0\nZeroDivisionError: division by zero\n",
    "children": [],
```

(下页继续)

(续上页)

```
    "task_id": "b7364eda-bbd4-4dc7-89ba-16589dff8401",
    "date_done": "2020-09-05T08:07:12.214929"
}
```

如果同时在 task 里配置了 *ignore_result=True* , 则不会保存结果到对应的 backend.

2. 消费配置

   通过配置项 *broker_transport_options* 可以配置消费参数;

```
broker_transport_options = {
    'max_retries': 5  #  最大尝试次数
}
```

3. 消费完之后再 Acknowledged

   celery 默认 ACK 是当一个任务执行后，立刻发送 Acknowledged 信号，标记该任务已经被执行。但是异常中断时，该任务不会被重新分发。可以通过配置 *task_acks_late* 让任务执行完成后再发送 Acknowledged. 这样可以保证不丢消息，但最好保证消费是幂等的，不然会影响结果。

4. 读多条消息

   默认情况下，celery worker 一次会读取 4 条消息，可以通过 *worker_prefetch_multiplier* 配置，如果不希望一次读多条，设置为 1，如果设置为 0，则 worker 一次会读取尽可能多的消息。

CHAPTER 9

shell

9.1 sed

9.1.1 匹配空格

```
sed -i 's/key[[:space:]]*=[[:space:]]*value/key=new_value/' file
```

9.1.2 行范围

```
# 匹配行到最后一行
sed -n '/Installed Packages/,$'p file.txt

# 前两行
sed -n '1,2'p file.txt

# 去掉第一行
sed -n '2,$'p file.txt
```

9.1.3 模糊匹配

```
# 替换 *.iso 为 test.iso, 注意引号的区别
sed -i "s/\\(.*\\)iso/test.iso/" vm.xml
sed -i 's/\(.*\)iso/test.iso/' vm.xml
```

9.2 输出颜色

```
RED='\033[0;31m'
NC='\033[0m'
echo "${RED}hello world!${NC}"
```

9.3 系统

9.4 进程分析

```
# 查看僵尸进程
ps -A -ostat,ppid,pid,cmd |grep -e '^[Zz]'
```

CHAPTER 10

Go

10.1 context

控制并发的方式

WaitGroup

10.2 字符串操作

10.2.1 字符串拼接

```
// 直接相加
s1 := "hello" + "world"

// 格式化
s2 := fmt.Sprintf("%s%s", "hello", "world")

// strings.Join
var strList []string = []string{"hello", "world"}
s3 := strings.Join(strList, "")

// buffer.WriteString
```

(下页继续)

(续上页)

```
var bt bytes.Buffer
bt.WriteString("hello")
bt.WriteString("world")
s4 := bt.String()

// strings.Builder, 和 WriteString 差不多，不过官方推荐这种方式
var build strings.Builder
build.WriteString("hello")
build.WriteString("world")
s5 := build.String()
```

10.3 json 处理

> **警告:** 如果序列化成 JSON，只有大写开头的变量才会被序列化。

eg: 下面的例子中，age 字段不会被序列化。

```
type Student struct {
    Name string `json:"name"`
    age  int    `json:"age"`
}
```

1. 针对可有可无的字段

   如果有些字段不一定存在，可以使用 *omitempty* 注解，但是不能区分零值和是否存在, eg:

   ```
   type Domain struct {
       Hosts  []string `json:"hosts"`
       TaskID string   `json:"task_id,omitempty"`
   }
   ```

10.4 pprof 火焰图

pprof 可以用来统计 cpu 和内存的使用情况。如果应用是 api 或者服务类型的，使用 *net/http/pprof* 库，如果是单次运行的，使用 *runtime/pprof* 工具。

火焰图依赖 graphviz 工具，安装方式如下：

```
# for macos
brew install graphviz

# for unbunt
apt install graphviz -y

# for centos
yum install graphviz -y
```

注解: go 1.11 开始已经支持火焰图了，如果是之前的版本，可以使用 go-torch.

10.4.1 runtime/pprof

先看看 *runtime/pprof* 的使用方式，示例代码如下：

```
package main

import (
    "flag"
    "log"
    "os"
    "runtime"
    "runtime/pprof"
    "time"
)

var cpuprofile = flag.String("cpuprofile", "", "write cpu profile to `file`")
var memprofile = flag.String("memprofile", "", "write memory profile to `file`")

func count() {
    sum := 0
    for i := 0; i < 1000000; i++ {
            sum += i
            for j := 0; j < 10000; j++ {
                    sum -= j
            }
    }
    fmt.Println(sum)
```

(下页继续)

(续上页)

```
}

func sleep() {
    a := []string{"a", "b", "c", "d"}
    for i := range a {
            fmt.Println(i)
    }
    time.Sleep(time.Second * 5)
}
func main() {
    flag.Parse()
    if *cpuprofile != "" {
            f, err := os.Create(*cpuprofile)
            if err != nil {
                    log.Fatal("could not create CPU profile: ", err)
            }
            defer f.Close() // error handling omitted for example
            if err := pprof.StartCPUProfile(f); err != nil {
                    log.Fatal("could not start CPU profile: ", err)
            }
            defer pprof.StopCPUProfile()
    }

    if *memprofile != "" {
            f, err := os.Create(*memprofile)
            if err != nil {
                    log.Fatal("could not create memory profile: ", err)
            }
            defer f.Close() // error handling omitted for example
            runtime.GC()    // get up-to-date statistics
            if err := pprof.WriteHeapProfile(f); err != nil {
                    log.Fatal("could not write memory profile: ", err)
            }
    }

    count()
    sleep()
}
```

生成 pprof 文件

```
# 生成二进制文件
go build main.go

# 生成 prof 文件
go run main.go -cpuprofile cpu.prof -memprofile mem.prof
```

web 查看

```
go tool pprof -http=":8081" main cpu.prof
```

10.4.2 net/http/pprof

示例代码:

```
package main

import (
    "fmt"
    "log"
    "net/http"
    _ "net/http/pprof"
)

func main() {
    go func() {
            log.Println(http.ListenAndServe(":8080", nil))
    }()

    // 占用 cpu
    for {
            fmt.Println("hello")
    }
}
```

如果是默认的 *http.DefaultServeMux*, 只需要加一行 import 就行: _ "*net/http/pprof*",

如果你使用自定义的 Mux, 则需要手动注册一些路由规则:

```
r.HandleFunc("/debug/pprof/", pprof.Index)
r.HandleFunc("/debug/pprof/cmdline", pprof.Cmdline)
r.HandleFunc("/debug/pprof/profile", pprof.Profile)
```

(下页继续)

(续上页)

```
r.HandleFunc("/debug/pprof/symbol", pprof.Symbol)
r.HandleFunc("/debug/pprof/trace", pprof.Trace)
```

1. 编译 *go build main.go*
2. 运行 server *go run main.go*
3. 访问 *http://localhost:8080/debug/pprof/*
4. 点击 *profile* 会下载随后 30s 的 cpu profile 文件;
5. web 查看 go tool pprof -http=" :8081" main profile

10.5 工具使用

10.5.1 标准库文档

https://studygolang.com/pkgdoc

10.5.2 pkg.go.dev

pkg.go.dev 是有关 Go 软件包和模块的信息资源中心。以前看包的文档需要在 godoc.org 上看，现在 pkg.go.dev 也提供 Go 文档功能。并且它懂 go module，提供相关软件包以前版本的信息!

> **警告:** 如果一个 git 库里有多个子项目，路径需要精确到子项目，否则看不到 api Doc!
>
> eg: github.com/prometheus/client_golang 有 api 和 prometheus，如果要查看 prometheus，需要搜索: github.com/prometheus/client_golang/prometheus

10.6 常用工具库

1. mongodb
   - https://www.mongodb.com/blog/post/mongodb-go-driver-tutorial
   - https://github.com/qiniu/qmgok (七牛云开源产品，做了一些封装，项目初期，还不是很稳定)
2. mysql
3. redis
4. elasticsearch
5. GUI

- https://github.com/fyne-io/fyne

10.7 go 陷阱

1. make slice 时指定 len;

   eg:

   ```
   // 指定 len 和 cap 都为 10
   var docs []string = make([]string, 10, 10)
   ```

   这个时候 docs 已经有十个已经初始化的值了，进行 append 操作时，不会删除原来的值，正确做法是指定 len 为 0，cap 为我们希望的大小。

10.8 Go module

Go 的依赖管理比较混乱，在 Go 1.11 版本引入了 Go Modules.

10.8.1 启用 go modules

启用条件：

1. go 版本大于 v1.11
2. 设置 GO111MODULE 环境变量

要使用 go module 首先要设置 GO111MODULE=on，GO111MODULE 有三个值，off、on、auto，off 和 on 即关闭和开启，auto 则会根据当前目录下是否有 go.mod 文件来判断是否使用 modules 功能。无论使用哪种模式，module 功能默认不在 GOPATH 目录下查找依赖文件，所以使用 modules 功能时请设置好代理。

10.8.2 使用

```
export GO111MODULE=on

# 初始化
go mod init github.com/you/hello

# go build 会将项目的依赖添加到 go.mod 中
go build
```

10.8.3 配置代理

```
export GOPROXY=https://mirrors.aliyun.com/goproxy/
```

10.9 不喜欢 go 的点

1. 不允许存在未使用的变量和包；

   这一点有点烦，特别是在代码开发调试阶段，虽然可以通过注释来规避，但总是看着不太爽。另外 go 的这个检查并不是非常准确。

   eg: 下面的代码定义了 findOptions 变量，实际并没有使用，编译阶段也不会报错。

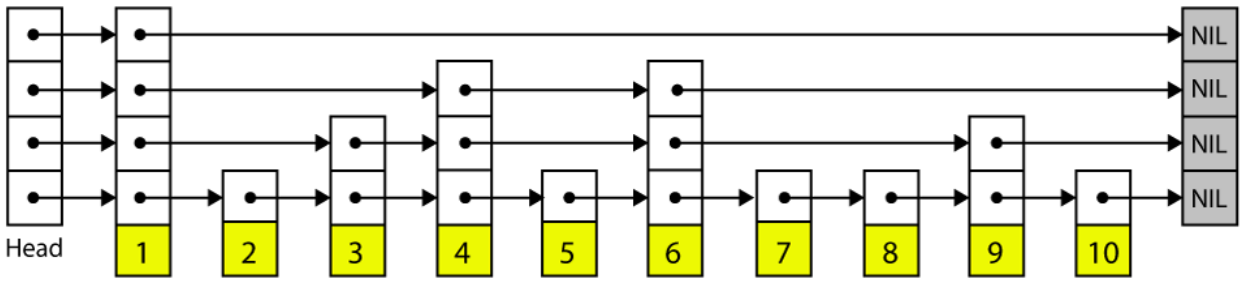
   ```
   findOptions := options.Find()
   findOptions.SetLimit(2)
   cur, err := collection.Find(context.TODO(), bson.D{})
   ```

CHAPTER 11

数据结构

11.1 跳表

跳表的本质是在链表的基础上建立多级索引，如下图所示：



跳表操作：

1. 插入
2. 删除
3. 查找
4. 查找一个区间的元素
5. 输出有序序列

为啥 redis 使用跳表，不使用红黑树:

1. 跳表操作时间复杂度和红黑树相同;

2. 跳表代码实现更易读;

3. 跳表区间查找效率更高

CHAPTER 12

OpenResty

CHAPTER 13

监控

13.1 prometheus

13.1.1 使用

prometheus 是拉模式的，如果要向 prometheus 发送数据，可以先发送到 pushgateway，然后再配置 prometheus 拉取 pushgateway 的数据。

https://github.com/prometheus/pushgateway

13.1.2 metrics

数据类型

1. Counter

   Counter 是计数器，单调递增的，只有服务重启时才会清零。

2. Guage
3. Historgram
4. Summary

13.1.3 疑问

1. 服务 down 机重启 Counter 会重新计数；
2. 起多个进程，Counter 错乱；

CHAPTER 14

Sphinx

14.1 语法

请参考 reference。

14.2 自定义样式

1. 创建样式文件 *source/_static/css/style.css*;

```
.wy-nav-content {
    max-width: 1200px !important;
}
```

2. 创建模板 *source/_templates/layout.html*;