notes

发布 *1.0*

2021 年 11 月 25 日

Contents

CHAPTER 1

操作系统

1.1 磁盘

页

1.2 常用 Linux 命令

CHAPTER 2

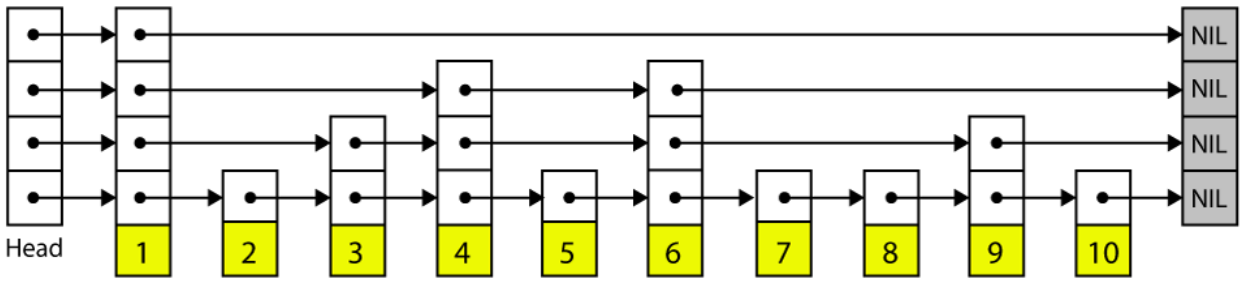
数据结构

2.1 B 树

2.2 B+ 树

2.3 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 3

计算机网络

3.1 TCP

3.2 HTTP

keep-alive: 使用长连接，keep-alive 的值是每次 http 请求完成后的等待时间，超过 keep-alive 的值后会断开 TCP 连接。

CHAPTER 4

存储

4.1 MySQL

4.1.1 常用操作

```
-- mysql 8.0 以上八版本
CREATE DATABASE mydatabase CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci;

-- 低版本
CREATE DATABASE mydatabase CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
```

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;
```

4.1.2 数据迁移

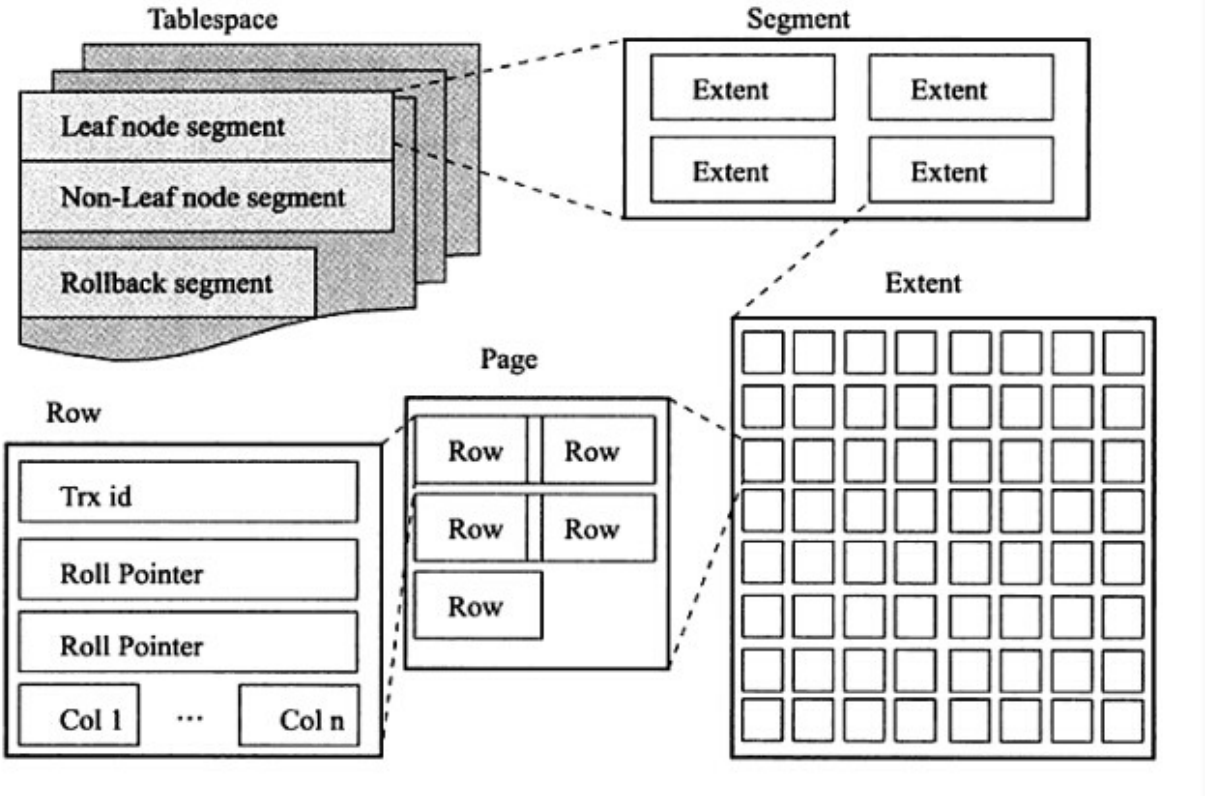
```
# 导出所有数据库
mysqldump -uroot -p123456 --all-databases > all.sql

# 新环境导入
mysql -uroot -p123456 < all.sql

# 导出指定的数据库
mysqldump -uroot -p123456 mydb > mydb.sql

# 导入
mysql -uroot -p123456 mydb < mydb.sql

# 导出指定表的数据
mysqldump -uroot -p123456 mydb mytable > mytable.sql

# 导入
mysql -uroot -p123456 mydb < mytable.sql
```

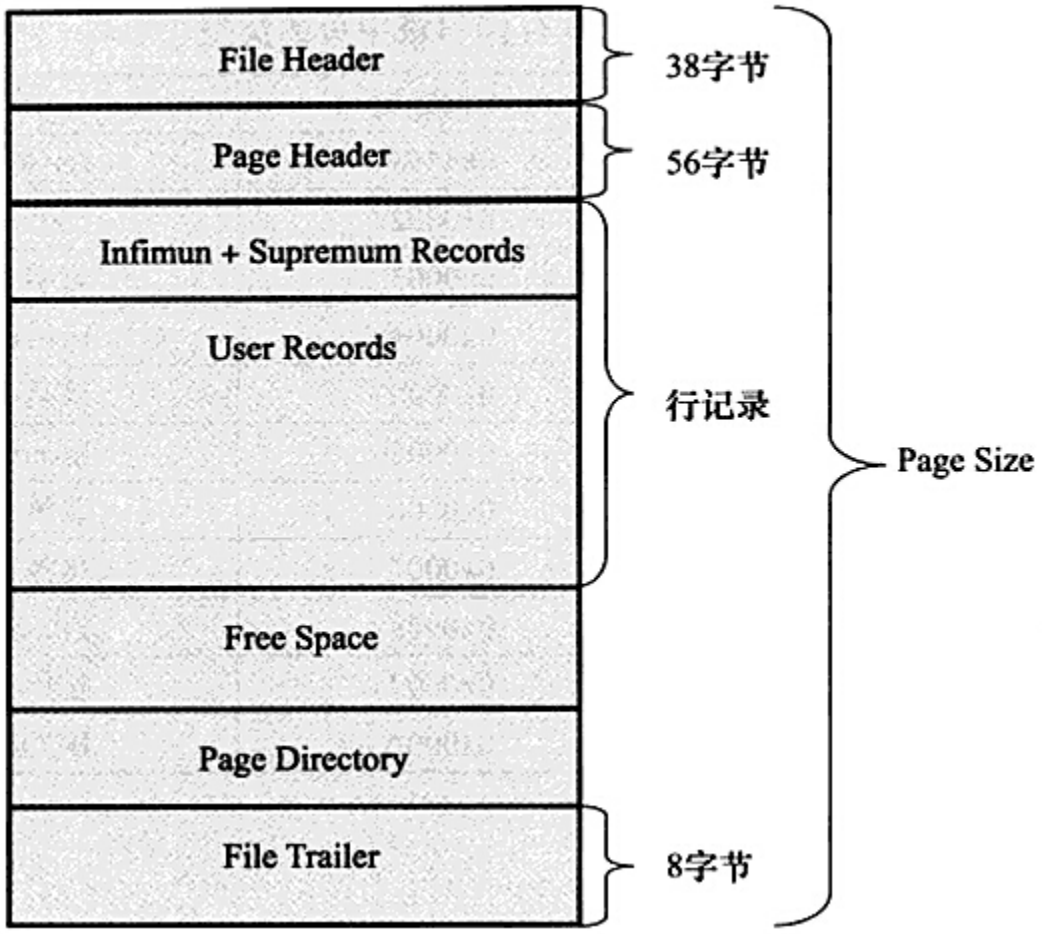
4.1.3 存储引擎

InnoDb

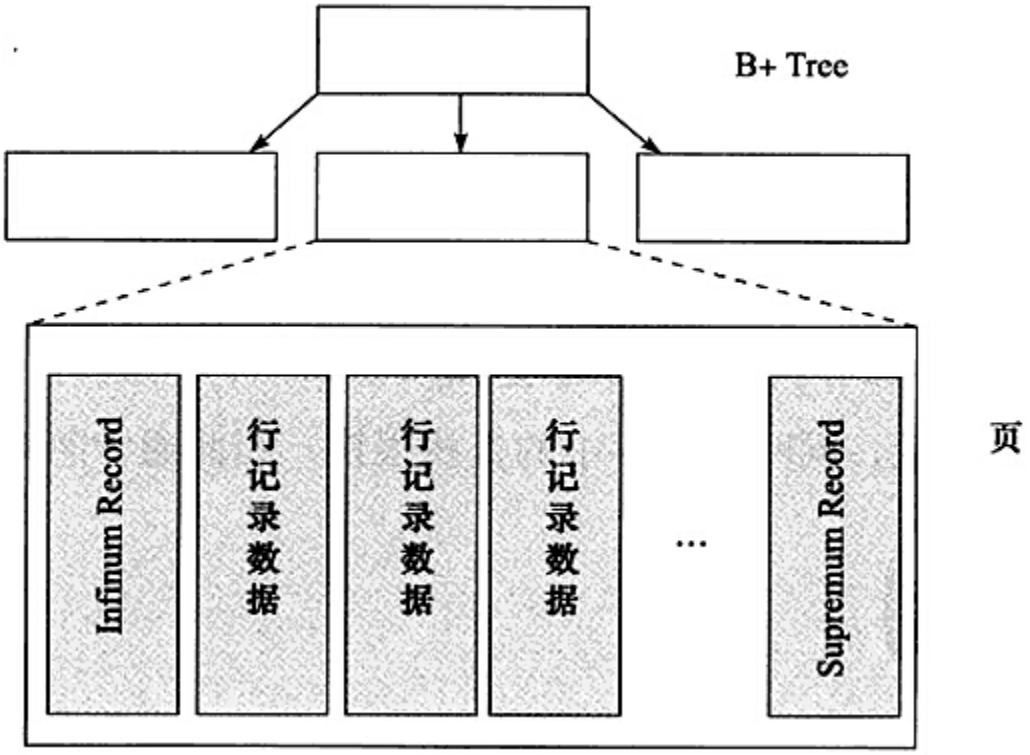
页（Page）是 Innodb 存储引擎用于管理数据的最小磁盘单位。常见的页类型有数据页、Undo 页、系统页、事务数据页等，本文主要分析的是数据页。默认的页大小为 16KB，每个页中至少存储有 2 条或以上的行记录，本文主要分析的是页与行记录的数据结构，有关索引和 B-tree 的部分在后续文章中介绍。

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

4.1.4 索引

4.1.5 事物

事物特性 (ACIC)

1. 原子性 (atomicity)

   一个事物要么全部提交成功，要么全部失败回滚，不能只执行其中的一部分操作。

2. 一致性 (consistency)

   事物的执行不能破坏数据库数据的完整性和一致性，一个事物在执行钱和执行后，数据库都必须处于一致性状态。

3. 隔离性 (isolation)

4. 持久性 (durability)

   一旦事物提交，那么它对数据库的状态的变更就会永久保存到数据库中。

事物隔离级别

数据库事物有四种隔离级别，默认事物级别为可重复读:

1. Read Uncommitted(读未提交)
2. Read Committed(读已提交)
3. Repeatable Read(可重复读)
4. Serializable(可串行化)

   这是最高的隔离级别，它通过强制事物排序，使之不可能相互冲突，从而解决幻读问题。简言之，它是在每个读的数据行上加上共享锁。在这个级别，可能导致大量的超时和锁竞争。

4.1.6 锁

1. 行锁

4.1.7 连接数

```
# 设置最大连接数量
set GLOBAL max_connections=100;

# 查看最大连接数
show variables like 'max_connections';

# 查看已使用的连接数
show global status like 'Max_used_connections';
```

4.2 Mongodb

4.2.1 基本操作

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})
```

增删改查

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

# 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({})
```

更新子文档

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()
```

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)
```

4.2.2 分片

4.3 ElasticSearch

4.3.1 常用操作

```
# 查看线程池排队情况
GET /_cat/thread_pool?v

# 模板列表
GET /_cat/templates
```

4.3.2 搜索

指定节点或者分片查询: https://www.elastic.co/guide/en/elasticsearch/reference/5.6/search-request-preference.html

4.3.3 聚合

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

聚合排序

```
{
   "query": {},
   "aggs": {
       "frame_id": {
           "terms": {
               "field": "frame_id",
               "size": 10,
               "order": {
                   "relation_id": "desc"
               }
           },
           "aggs": {
               "relation_id": {
                   "cardinality": {
                       "field": "relation_id"
                   }
               }
           }
       }
   }
}
```

聚合分区

如果聚合的结果太多，一次返回会非常慢，可以将结果分块，然后每次返回一部分。

```
{
   "size": 0,
   "aggs": {
      "expired_sessions": {
         "terms": {
            "field": "account_id",
            "include": {
               "partition": 0,
               "num_partitions": 20
            },
            "size": 10000,
            "order": {
               "last_access": "asc"
            }
         },
         "aggs": {
            "last_access": {
               "max": {
                  "field": "access_date"
               }
            }
         }
      }
   }
}
```

注解: 如果指定了一个很大的分区数量，而聚合的结果很少，返回的结果是在随机分区，而不是第一个或者最后一个分区。

常规做法:

1. 使用 *cardinality* 统计总的数据，从而计算分区数量;
2. 依次获取每个分区的数据;

4.3.4 分片

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

4.3.5 reindex

4.4 Redis

4.4.1 分布式锁

锁冲突处理

- 直接抛出异常，通知用户；
- sleep 一会再重试；
- 将请求移至延时队列，过一会再试；

4.4.2 延时队列

核心代码:

```
def delay(msg):
    msg.id = str(uuid.uuid4())
    value = json.dumps(msg)
    retry_ts = time.time() + 5  # 5s 后重试
    redis.zadd("delay-queue", retry_ts, value)


def loop():
    while True:
        # 最多取 1 条
        values = redis.zrangebyscore("delay-queue", 0, time.time(), start=0, num=1)
        if not value:
            time.sleep(1)
            continue
        value = values[0]
        success = redis.zrem("delay-queue", value)
        if success:
            msg = json.load(value)
            handle_msg(msg)
```

4.4.3 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
```

4.5 Etcd

4.5.1 etcd 介绍

etcd 官方的定义是:" 一个分布式，可靠的关键数据 kv 存储"。

特性

1. 接口简单，采用标准的 http 协议，json 传输;
2. 数据存储在分层的目录结构;
3. 可监控指定 key 或者目录变化;
4. 支持 SSL 认证;
5. 单机每秒 1000 的写入;
6. 支持 key TTLs 过期;
7. 采用 Raft 算法;

关于 Raft 算法可以参考：http://thesecretlivesofdata.com/raft/

4.5.2 基本语法

术语

1. Node / 节点
2. Member / 成员

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
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
```

4.5.3 认证

```
# 添加 root 用户
etcdctl --endpoints=http://127.0.0.1:2379 user add root

# 开启认证
etcdctl --endpoints=http://127.0.0.1:2379 auth enable

# 关闭认证
etcdctl --endpoints=http://127.0.0.1:2379 --user=root:123456 auth disable

# 创建普通用户
etcdctl --endpoints=http://127.0.0.1:2379 --user=root:123456 user add guest

# 添加角色
```

(下页继续)

(续上页)

```
etcdctl --endpoints=http://127.0.0.1:2379 --user=root:123456 role add normal

# 角色授权
etcdctl --endpoints=http://127.0.0.1:2379 --user=root:123456 role grant-permission --
↪prefix=true normal readwrite /path_name

# 用户绑定角色
etcdctl --endpoints=http://127.0.0.1:2379 --user=root:123456 user grant-role guest normal
```

4.5.4 集群管理

```
# 集群状态
etcdctl member list
```

4.6 HugeGraph

4.6.1 基本操作

propertykeys 支持的数据类型

- DOUBLE
- BYTE
- UUID
- FLOAT
- BLOB
- DATE
- OBJECT
- BOOLEAN
- TEXT
- INT
- LONG

Vertex Id 类型

- AUTOMATIC

- PRIMARY_KEY
- CUSTOMIZE_STRING
- CUSTOMIZE_NUMBER

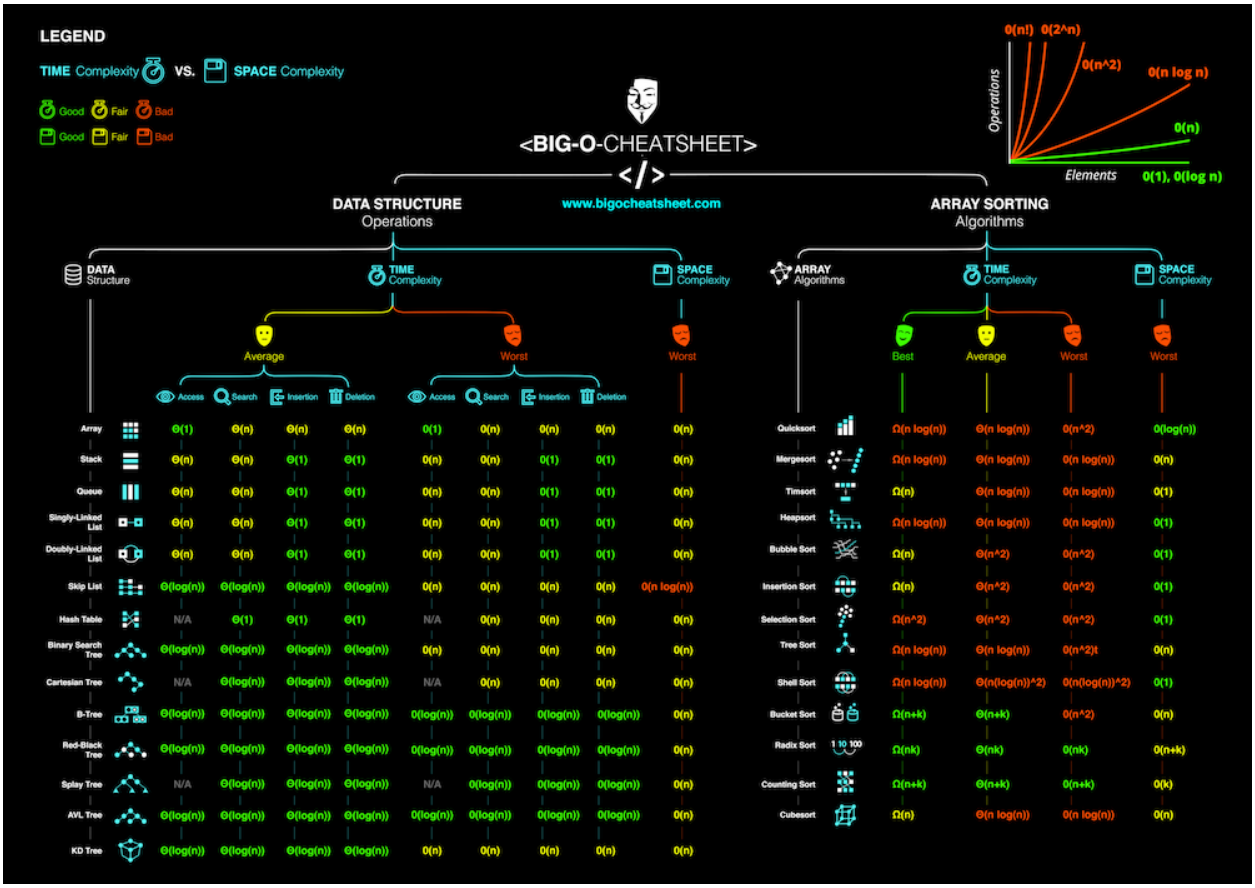
Edgelabels frequency 类型

- SINGLE
- DEFAULT
- MULTIPLE: 必须指定 sort_keys

CHAPTER 5

算法

5.1 复杂度



5.2 随机抽奖算法

题目：

从 1- n 中随机抽取 k 个数。

```
from random import randint

def sample(population, k):
    result = [None] * k
    pool = list(population)
    n = len(population)
    for i in range(k):
        j = randint(0, n-i)
        result[i] = pool[j]
        pool[j] = pool[n-i-1]  # 把末尾的元素移动到抽出的位置

    return result
```

CHAPTER 6

编程语言

6.1 JavaScript

6.1.1 字符串

1. 字符串使用变量

```
`String text ${expression}`
```

6.2 Go

6.2.1 Gin

基本用法

1. 写数据

```
// string
func (c *Context) String(code int, format string, values ...interface{})

// json
func (c *Context) JSON(code int, obj interface{})
```

参数绑定

1. 参数校验

```
// require
type UserParams struct {
    Name     string `json:"name" binding:"required"`
    Password string `json:"password" binding:"required"`
}
```

6.2.2 fasthttp

介绍

项目地址：https://github.com/valyala/fasthttp

fasthttp 不仅是 server 端，也可以当 client 使用。

安装：

```
go get -u github.com/valyala/fasthttp
```

最佳实践

- 不要分配 *[]byte* 缓冲区，尽可能的复用；
- 多使用 *sync.Pool*；

http

1. 请求信息

```
// 请求方法
ctx.Method()

//
ctx.RequestURI()

ctx.Path()

ctx.Host()
```

(下页继续)

(续上页)

```
ctx.QueryArgs()

ctx.UserAgent()

// 连接建立时刻
ctx.ConnTime()

// 请求开始时刻
ctx.Time()

ctx.ConnRequestNum()

ctx.RemoteIP()
```

2. 设置 header

```
ctx.Response.Header.Set("X-My-Header", "my-header-value")
```

3. 设置 cookies

```
var c fasthttp.Cookie
c.SetKey("cookie-name")
c.SetValue("cookie-value")
ctx.Response.Header.SetCookie(&c)
```

router

fasthttp 没有没有提供 router, 也没打算集成进来，官方推荐使用第三方路由。这里我们用 *fasthttp-routing*。

1. 基本用法

```
router := routing.New()
fasthttp.ListenAndServe(":8080", router.HandleRequest)
```

2. Route Groups

```
router := routing.New()
api := router.Group("/api")
api.Use(m1, m2)
api.Get("/users", h1).Post(h2)
api.Put("/users/<id>", h3).Delete(h4)
```

3. router 数据结构

```
type Router struct {
    RouteGroup
    pool              sync.Pool
    routes            map[string]*Route
    stores            map[string]routeStore
    maxParams         int
    notFound          []Handler
    notFoundHandlers []Handler
}

type RouteGroup struct {
    prefix   string
    router   *Router
    handlers []Handler
}
```

源码分析

```
// 这里的 handler 是一个函数
func ListenAndServe(addr string, handler RequestHandler) error {}

type RequestHandler func(ctx *RequestCtx)
```

6.2.3 基础

1. ‘x’ , ‘n’ 是 byte 类型;

6.2.4 类型

1. 类型断言

```
// 其中 i 为 interface{} 类型 T 是要断言的类型。
// 1. 如果 i 是 T 类型, 赋值 i 给 t;
// 2. 如果 i 不是 T 类型, 触发 panic
t := i.(T)

// 1. 如果 i  是 T 类型, 赋值 i 给 T, ok 为 true;
```

(下页继续)

(续上页)

```
// 2. 如果 i 不是 T 类型，赋值 T 类型的零值给 i，ok 为 false;
t, ok := i.(T)
```

2. 获取变量类型

```
// 获取变量类型
v1 := "hello"
// 输出 string，返回的是 Type 类型
fmt.Println(reflect.typeOf(v1))

// 直接打印类型
fmt.Printf("v1 type:%T\n", v1)
```

3. 类型转换

```
// string to int
int, err := strconf.Atoi(string)

// string to int64
int64, err := strconv.ParseInt(string, 10, 64)

// int to string
string := strconv.Itoa(int)

// int64 to string
string := strconv.FormatInt(int64, 10)
```

6.2.5 interface

interface 定义

特点:

1. 函数没有 {}
2. 函数有参数和返回值

```
type Writer interface {
    Write(p []byte) (n int, err error)
}
```

6.2.6 slice

数组和切片是两种不同的数据类型。数组的长度是固定且不可修改的。

数组

```
// 声明，数组声明时，会用零值初始化
var arr [5]int

// 创建并初始化
arr1 := [5]int{1, 2, 3, 4, 5}

// 创建并初始化（自动根据给定值计算数组大小）
arr2 := [...]int{1, 2, 3, 4, 5}

// 指定部分初始化值
// [0, 10, 20, 0, 0]
arr3 := [5]int{1: 10, 2:20}

// 指针数组
var arr4 [5]*int
*arr4[0] = 10

// 指针数组初始化
arr4 := [5]*int{0: new(int), 1:new(int)}

// 下标方式遍历
for i := 0; i < len(arr); i++ {
}

// range 方式遍历
for idx, value := range arr {
}

// 二维数组声明
var arr5 [4][2]int
```

易错点

1. 数组赋值

```go
var arr1 [5]int
arr2 := [5]int{1, 2, 3, 4, 5}

// 这里 arr1 和 arr2 必须是相同类型
// 此时只是把 arr2 的内容复制给 arr1，两个数组都是独立的，并不是把 arr1 指向 arr2
arr1 = arr2

// 如果是指针数组，则执行的内容是一样的
var a [2]*int
b := [2]*int{0: new(int), 1: new(int)}
a = b
*a[0] = 3
// 这里 a[0] 和 b[0] 的地址是相同的
fmt.Println(a[0], b[0])
```

2. 函数参数

```go
var arr [5]int

// 数组作为函数参数是传值，而不是传引用，这一点和 python 有很大区别
func foo1(arr [5]int) {}
foo1(arr)

// 通过引用的方式传参
func foo2(arr *[5]int) {}
foo2(&arr)
```

切片

1. 切片的内存也是连续的，可以使用索引访问；

6.2.7 协程

协程同步

1. sync.WaitGroup

   sync.WaitGroup 只有三个方法，*Add()* 添加计数，*Done()* 计数减一，*Wait()* 阻塞，直到计数为 0。

```go
var wg sync.WaitGroup
```

(下页继续)

(续上页)

```
func task(i int) {
    defer wg.Done()
    fmt.Println(i)
}

func main() {
    for i := 0; i < 10; i++ {
        wg.Add(1)
        go task(i)
    }

    wg.Wait()
}
```

2. 有缓存 channel

   有缓存 channel 不能保证 goroutine 的有序执行。

```
var ch = make(chan int, 10)

func task(i int) {
    fmt.Println(i)
    time.Sleep(time.Second)
    <- ch
}

func main() {
    for i := 0; i < 10; i++ {
        go task(i)
        ch <- i
    }
}
```

3. 无缓存 channel

   无缓存 channel 可以保证 goroutine 的有序执行。

```
var ch = make(chan int)

func task(i int) {
    fmt.Println(i)
    time.Sleep(time.Second)
```

(下页继续)

(续上页)

```
    <- ch
}

func main() {
    for i := 0; i < 10; i++ {
        go task(i)
        ch <- i
    }
}
```

6.2.8 io

```
type Reader interface {
    Read(p []byte) (n int, err error)
}

type Writer interface {
    Write(p []byte) (n int, err error)
}

// 多了偏移量
type ReaderAt interface {
    ReadAt(p []byte, off int64) (n int, err error)
}

type WriterAt interface {
    WriteAt(p []byte, off int64) (n int, err error)
}

type ReaderFrom interface {
    ReadFrom(r Reader) (n int64, err error)
}

type WriterTo interface {
    WriteTo(w Writer) (n int64, err error)
}

type Seeker interface {
    Seek(offset int64, whence int) (ret int64, err error)
```

(下页继续)

(续上页)

```
}

type Closer interface {
    Close() error
}
```

实现了 io.Reader 接口或 io.Writer 接口的类型

- os.File 同时实现了 io.Reader 和 io.Writer
- strings.Reader 实现了 io.Reader
- bufio.Reader/Writer 分别实现了 io.Reader 和 io.Writer
- bytes.Buffer 同时实现了 io.Reader 和 io.Writer
- bytes.Reader 实现了 io.Reader
- compress/gzip.Reader/Writer 分别实现了 io.Reader 和 io.Writer
- crypto/cipher.StreamReader/StreamWriter 分别实现了 io.Reader 和 io.Writer
- crypto/tls.Conn 同时实现了 io.Reader 和 io.Writer
- encoding/csv.Reader/Writer 分别实现了 io.Reader 和 io.Writer
- mime/multipart.Part 实现了 io.Reader
- net/conn 分别实现了 io.Reader 和 io.Writer(Conn 接口定义了 Read/Write)

文件读取

1. 读取小文件

```
func ReadAll(filePath string) ([]byte, error) {

   // f 是 os.File
   f, err := os.Open(filePath)
   if err != nil {
      return nil, err
   }
   return ioutil.ReadAll(f)
}
```

或者使用 ioutil

```
data, err := ioutil.ReadAll(filename)
```

2. 逐行读取

```
func ReadLines(filePath string) error {
  f, err := os.Open(filePath)
  if err != nil {
          fmt.Println(err.Error())
          return err
  }

  defer f.Close()

  reader := bufio.NewReader(f)
  for {
          line, err := reader.ReadBytes('\n')
          fmt.Print(string(line))
          if err != nil {
                  if err == io.EOF {
                          return nil
                  }
                  return err
          }
  }
}
```

6.2.9 http

使用 http.Client

1. GET

```
package main

import (
  "io/ioutil"
  "net/http"
  "time"
)

func Get(url string) string{
    client := &http.Client{}
    resp, err := client.Get(url)
```

(下页继续)

(续上页)

```
    if err != nil {
        panic(err)
    }

    // 读取 body
    defer resp.Body.Close()
    body, err := ioutil.ReadAll(resp.Body)
}
```

常用操作

1. 设置超时时间

```
// 设置 5s 超时时间
client := &http.Client{Timeout: 5*time.Second}
```

2. 设置 header

```
// 需要使用 http.NewRequest
client := &http.Client{}
req, err := http.NewRequest("POST", url, bytes.NewReader(jsonStr))
req.Header.Add("Content-Type", "application/json")
resp, err := client.Do(req)
```

6.2.10 锁

- 互斥锁

```
func main() {
    var mutex sync.Mutex
    count := 0

    for r := 0; r < 50; r++ {
       go func() {
               mutex.Lock()
               count += 1
               mutex.Unlock()
           }()
       }
```

(下页继续)

(续上页)

```
    time.Sleep(time.Second)
    fmt.Println("the count is : ", count)
}
```

- 读写锁
  1. *func (rw *RWMutex) Lock()*
  2. *func (rw *RWMutex) RLock()*
  3. *func (rw *RWMutex) RLocker() Locker*
  4. *func (rw *RWMutex) RUnlock()*
  5. *func (rw *RWMutex) Unlock()*

6.2.11 time

计算日期偏移

AddDate(years int, months int, days int)

```
format := "2006-01-02"

//  2015-05-01
d1, _ := time.Parse(format, "2015-03-31")
fmt.Println(d1.AddDate(0, 2, 0))

// 2015-05-30
d2, _ := time.Parse(format, "2015-04-30")
fmt.Println(d2.AddDate(0, 1, 0))

// 2013-03-01
d3, _ := time.Parse(format, "2012-02-29")
fmt.Println(d3.AddDate(1, 0, 0))
```

6.2.12 context

控制并发的方式

Context 定义

```
type Context interface {
    Deadline() (deadline time.Time, ok bool)

    Done() <-chan struct{}

    Err() error

    Value(key interface{}) interface{}
}
```

有哪些类型的 Context

- Background
- TODO

context 包常用函数

- func WithCancel(parent Context) (ctx Context, cancel CancelFunc)

WaitGroup

6.2.13 字符串操作

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
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
```

(下页继续)

(续上页)

```
var build strings.Builder
build.WriteString("hello")
build.WriteString("world")
s5 := build.String()
```

6.2.14 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"`
   }
   ```

6.2.15 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
```

(下页继续)

(续上页)

```
# 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.

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)
}

func sleep() {
    a := []string{"a", "b", "c", "d"}
    for i := range a {
            fmt.Println(i)
```

(下页继续)

(续上页)

```
    }
    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
```

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)
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

6.2.16 工具使用

标准库文档

https://studygolang.com/pkgdoc

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

6.2.17 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

6.2.18 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。

6.2.19 Go module

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

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。

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
```

配置代理

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

6.2.20 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{})
```

2. interface 设计：

   如果返回的是一个 interface, 想看具体实现代码时比较麻烦；

6.2.21 杂谈

常量目录结构组织

```
  constvar
         constvar.go
```

6.3 Python

6.3.1 GC

6.3.2 pyenv

pyenv 可以帮助你在一台开发机上建立多个版本的 python 环境，并提供方便的切换方法。virtualenv 可以搭建虚拟且独立的 python 环境，可以使每个项目环境与其他项目独立开来，保持环境的干净，解决包冲突问题。

安装

```
git clone https://github.com/pyenv/pyenv.git ~/.pyenv
echo 'export PYENV_ROOT="$HOME/.pyenv"' >> ~/.bash_profile
echo 'export PATH="$PYENV_ROOT/bin:$PATH"' >> ~/.bash_profile
echo -e 'if command -v pyenv 1>/dev/null 2>&1; then\n  eval "$(pyenv init -)"\nfi' >> ~/.
↪bash_profile

# restart shell
exec "$SHELL"
```

安装 python 版本

```
# 安装 python 版本
pyenv install 2.7.10

# centos 需要安装如下包
yum install patch openssl-devel -y
```

pyenv-virtualenv 安装

光有 pyenv 还不够，我们需要结合 virtualenv 来使用，pyenv-virtualenv 就是干这个活的。

```
git clone https://github.com/pyenv/pyenv-virtualenv.git $(pyenv root)/plugins/pyenv-
↪virtualenv
echo 'eval "$(pyenv virtualenv-init -)"' >> ~/.bash_profile

# restart shell
exec "$SHELL"
```

使用

```
# 查看支持哪些版本
pyenv install -l

# 创建项目
pyenv virtualenv 2.7.10 project

# Activate virtualenv
pyenv activate project

# Deactive virtualenv
pyenv deactivate

# Delete existing virtualenv
pyenv uninstall project

# 查看已经安装了哪些 virtualenv
pyenv versions
```

6.3.3 FAQ

- */root/.pyenv/plugins/python-build/bin/python-build: line 1326: patch: command not found*

```
yum install patch -y
```

- mac os zlib 问题

  现象如下：

```
Last 10 log lines:
File "/private/var/folders/qq/cxqjcr296h7bvhl4nqbzrsn00000gn/T/python-build.
↪20190419143439.41015/Python-3.5.3/Lib/ensurepip/__main__.py", line 4, in <module>
  ensurepip._main()
File "/private/var/folders/qq/cxqjcr296h7bvhl4nqbzrsn00000gn/T/python-build.
↪20190419143439.41015/Python-3.5.3/Lib/ensurepip/__init__.py", line 209, in _main
  default_pip=args.default_pip,
File "/private/var/folders/qq/cxqjcr296h7bvhl4nqbzrsn00000gn/T/python-build.
↪20190419143439.41015/Python-3.5.3/Lib/ensurepip/__init__.py", line 116, in␣
↪bootstrap
  _run_pip(args + [p[0] for p in _PROJECTS], additional_paths)
File "/private/var/folders/qq/cxqjcr296h7bvhl4nqbzrsn00000gn/T/python-build.
↪20190419143439.41015/Python-3.5.3/Lib/ensurepip/__init__.py", line 40, in _run_pip
  import pip
zipimport.ZipImportError: can't decompress data; zlib not available
```

  解决办法：

```
brew install zlib
export LDFLAGS="-L/usr/local/opt/zlib/lib"
export CPPFLAGS="-I/usr/local/opt/zlib/include"
export PKG_CONFIG_PATH="/usr/local/opt/zlib/lib/pkgconfig"
```

6.3.4 pytest

技巧

1. 类全局共享

   有时候在 Test 类中需要用到共同的变量，直接写到构造函数或 class 是无法共享的，我们需要借助其它方式：

```python
@pytest.fixture(name="sample_manager", scope="class")
def sample_manager_fixture():
    class SampleManager:
        def __init__(self):
            self.last_value = 0

    return SampleManager()


class TestSample:
    testcases = [("name1", 1), ("name2", 2), ("name3", 3), ("name4", 4)]

    def test_order(self, sample_manager):
        print(sample_manager.last_value)

    @pytest.mark.parametrize(('testname', 'testInput'), testcases)
    def test_run(self, testname, testInput, sample_manager):
        if sample_manager.last_value >= 10:
            sample_manager.last_value += random.randint(1, 10)
        else:
            sample_manager.last_value += 5

        print(sample_manager.last_value)
```

6.4 Shell

6.4.1 sed

匹配空格

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

行范围

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

# 前两行
```

(下页继续)

(续上页)

```
sed -n '1,2'p file.txt

# 去掉第一行
sed -n '2,$'p file.txt
```

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
```

6.4.2 awk

```
# 求和
awk '{sum += $1};END {print sum}'

# 字符串转 int
awk '{print int($1)}'
```

6.4.3 输出颜色

```
RED='\033[31m'
GREEN='\033[32m'
BLUE='\033[36m'
NC='\033[0m'

# 需要加 -e 参数
echo -e "${RED}hello world!${NC}"
```

6.4.4 系统

6.4.5 进程分析

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

CHAPTER 7

前端开发

7.1 vue

7.1.1 cli

官方文档：https://cli.vuejs.org/guide/creating-a-project.html

7.2 iview

CHAPTER 8

云计算

8.1 K8s

8.1.1 Deployment

8.1.2 ConfigMap

8.1.3 Volume

- EmptyDir

  两个容器使用同一个 EmptyDir

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 1
  template:
    metadata:
```

(下页继续)

(续上页)

```
    labels:
      app: nginx
  spec:
    containers:
    - name: api
      image: nginx:latest
      imagePullPolicy: Always
      ports:
      - containerPort: 80
      volumeMounts:
      - name: log-data
        mountPath: /var/log/nginx

    - name: filebeat
      image: filebeat:latest
      imagePullPolicy: Always
      volumeMounts:
      - name: log-data
        mountPath: /var/log/nginx

    volumes:
    - name: log-data
      emptyDir: {}
```

8.1.4 节点管理

节点操作

```
# 驱赶节点上所有 pod
kubectl drain 192.168.100.102 --delete-local-data --ignore-daemonsets

# 删除节点
kubectl delete node

# 生成加入集群命令 (master 上执行)
kubeadm token create --print-join-command
```

label 操作

```
# 显示 label
kubectl get nodes --show-labels

# 添加 label
kubectl label nodes <node-name> <label-key>=<label-value>

# 删除 label
kubectl label nodes <node-name> <label-key>-

# 修改 label
kubectl label nodes <node-name> <label-key>=<label-value> --overwrite
```

使用 label 调度

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
      nodeSelector:
        label_name: label_value
```

8.1.5 用户认证

生成 dashboard 登录配置文件

```
# 查看 token
kubectl -n kube-system describe secret $(kubectl -n kube-system get secret | grep␣
↪kubernetes-dashboard-admin-token | awk '{print $1}')

# 设置登录地址
```

(下页继续)

(续上页)

```
kubectl config set-cluster kubernetes --server=<master ip>:6443 --kubeconfig=<filename>

# 这里的 scret 参数需要替换成上面获取到的登陆的 token 值
kubectl config set-credentials dashboard --token="<your token>" --kubeconfig=<filename>

kubectl config set-context dashboard@kubernetes --cluster=kubernetes --user=dashboard --
↪kubeconfig=<filename>
kubectl config use-context dashboard@kubernetes  --kubeconfig=<filename>
```

8.1.6 DNS

CoreDNS 策略

- None：表示空的 DNS 设置，这种方式一般用于想要自定义 DNS 配置的场景，而且，往往需要和 dnsConfig 配合一起使用达到自定义 DNS 的目的。
- Default：有人说 Default 的方式，是使用宿主机的方式，这种说法并不准确。这种方式其实是让 kubelet 来决定使用何种 DNS 策略。而 kubelet 默认的方式，就是使用宿主机的 /etc/resolv.conf，但是 kubelet 是可以灵活来配置使用什么文件来进行 DNS 策略的，我们完全可以使用 kubelet 的参数：–resolv-conf=/etc/resolv.conf 来决定您的 DNS 解析文件地址。
- ClusterFirst：这种方式表示 Pod 内的 DNS 使用集群中配置的 DNS 服务，简单来说，就是使用 Kubernetes 中 kubedns 或 coredns 服务进行域名解析。如果解析不成功，才会使用宿主机的 DNS 配置进行解析。
- ClusterFirstWithHostNet: 优先使用宿主机的 DNS 配置进行解析

如果未明确指定 dnsPolicy，则默认使用 **ClusterFirst**

- 如果将 dnsPolicy 设置为 “Default”，则名称解析配置将从运行 pod 的工作节点继承。
- 如果将 dnsPolicy 设置为 “ClusterFirst”，则 DNS 查询将发送到 kube-dns 服务。对于以配置的集群域后缀为根的域的查询将由 kube-dns 服务应答。所有其他查询（例如，www.kubernetes.io）将被转发到从节点继承的上游名称服务器。在此功能之前，通常通过使用自定义解析程序替换上游 DNS 来引入存根域。但是，这导致自定义解析程序本身成为 DNS 解析的关键路径，其中可伸缩性和可用性问题可能导致群集丢失 DNS 功能。此特性允许用户在不接管整个解析路径的情况下引入自定义解析。

如果某个工作负载不需要使用集群内的 coredns，可以使用 kubectl 命令或 API 将此策略设置为 dnsPolicy: Default。

配置方式

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
```

(下页继续)

(续上页)

```
  name: nginx
  labels:
    app: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:latest
      dnsPolicy: ClusterFirstWithHostNet
```

options 设置

```
dnsConfig:
  options:
    - name: timeout
      value: '2'
    - name: ndots
      value: '5'
    - name: single-request-reopen
```

8.1.7 创建 harbor 认证

```
kubectl create secret docker-registry harbor --namespace=ns --docker-server=https://your.
↪harbor.cn --docker-username=username --docker-password=password
```

8.1.8 集群配置

修改 node-port 端口范围：

1. 在 */etc/kubernetes/manifests/kube-apiserver.yaml* 文件中增加 *service-node-port-range* 参数；
2. systemctl restart kubelet；

```
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    component: kube-apiserver
    tier: control-plane
  name: kube-apiserver
  namespace: kube-system
spec:
  containers:
  - command:
    - kube-apiserver
    - --advertise-address=192.168.0.254
    - --allow-privileged=true
    - --authorization-mode=Node,RBAC
    - --client-ca-file=/etc/kubernetes/pki/ca.crt
    - --enable-admission-plugins=NodeRestriction
    - --enable-bootstrap-token-auth=true
    - --etcd-cafile=/etc/kubernetes/pki/etcd/ca.crt
    - --etcd-certfile=/etc/kubernetes/pki/apiserver-etcd-client.crt
    - --etcd-keyfile=/etc/kubernetes/pki/apiserver-etcd-client.key
    - --etcd-servers=https://127.0.0.1:2379
    - --insecure-port=0
    - --kubelet-client-certificate=/etc/kubernetes/pki/apiserver-kubelet-client.crt
    - --kubelet-client-key=/etc/kubernetes/pki/apiserver-kubelet-client.key
    - --kubelet-preferred-address-types=InternalIP,ExternalIP,Hostname
    - --proxy-client-cert-file=/etc/kubernetes/pki/front-proxy-client.crt
    - --proxy-client-key-file=/etc/kubernetes/pki/front-proxy-client.key
    - --requestheader-allowed-names=front-proxy-client
    - --requestheader-client-ca-file=/etc/kubernetes/pki/front-proxy-ca.crt
    - --requestheader-extra-headers-prefix=X-Remote-Extra-
    - --requestheader-group-headers=X-Remote-Group
    - --requestheader-username-headers=X-Remote-User
    - --secure-port=6443
    - --service-account-key-file=/etc/kubernetes/pki/sa.pub
    - --service-cluster-ip-range=10.1.0.0/16
    - --service-node-port-range=1-65535
```

8.2 Helm

8.2.1 介绍

helm3 移出了 Tiller。

8.2.2 repo

```
# 添加 repo
helm repo add my-helm-repo https://xxx/artifactory/my-helm-virtual --username <username>␣
↪--password <password>

# 更新 repo
helm repo update
```

注解: helm repo 的信息是存在 *$HOME/.helm* 目录下的，所以切换用户后，helm repo 信息是看不到的。

8.2.3 Charts

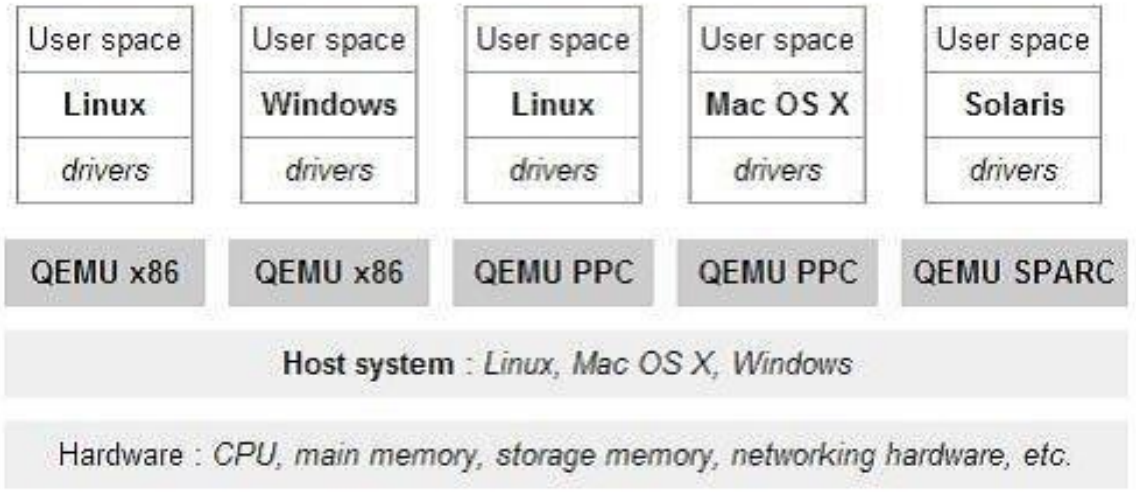
```
# 创建 Chart
helm create myapp

# 检查
helm lint myapp

# 打包
helm package myapp

# 上传文件到 helm repo (artifactory 仓库)
curl -u username:password -T myapp-0.1.0.tgz "https://xxx/artifactory/my-helm-release/
↪myapp-0.1.0.tgz"

# 安装 Chart
helm install <name> <package>

# 删除 Chart
helm uninstall <name>
```

8.3 Openstack

8.3.1 QEMU

什么是 qemu

qemu 官方的定义是: *QEMU is a generic and open source machine emulator and virtualizer.* 简单来说 qemu 就是用软件来模拟计算机的各种硬件, 使 guest os 认为自己和硬件直接打交道，其实是和 qemu 模拟的硬件交互。qemu 会将指令翻译给 host 执行。所有指令通过 qemu 翻译后执行性能会比较差。qemu 的架构如下图所示:



什么是 KVM

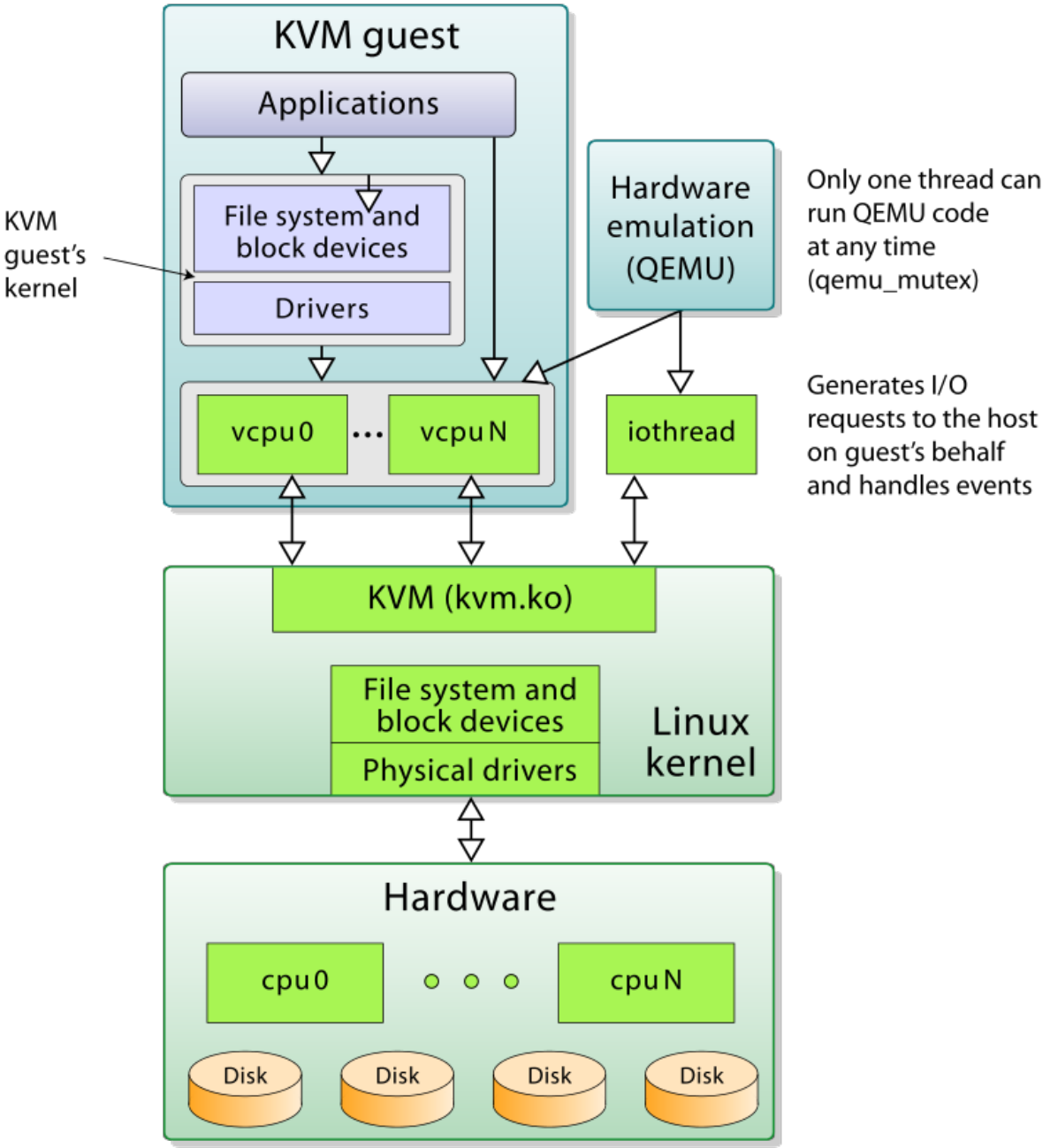
KVM 实际上是 linux 内核提供的虚拟化架构，可以将内核直接充当 hypervisor 来使用。KVM 需要处理器硬件本身支持虚拟化扩展，例如 intel VT 和 AMD AMD-V 技术。

KVM 包含一个内核模块 kvm.ko 用来实现核心虚拟化功能，以及一个和处理器强相关的模块如 kvm-intel.ko 或 kvm-amd.ko。KVM 本身不实现任何模拟，仅仅是暴露了一个/dev/kvm 接口，这个接口可被宿主机用来主要负责 vCPU 的创建，虚拟内存的地址空间分配，vCPU 寄存器的读写以及 vCPU 的运行。有了 KVM 以后，guest os 的 CPU 指令不用再经过 QEMU 来转译便可直接运行，大大提高了运行速度。但 KVM 的 kvm.ko 本身只提供了 CPU 和内存的虚拟化，所以它必须结合 QEMU 才能构成一个完整的虚拟化技术，也就是下面要介绍的技术。

什么是 QEMU-KVM

KVM 负责 cpu 虚拟化 + 内存虚拟化，实现了 cpu 和内存的虚拟化，但 kvm 并不能模拟其他设备，还必须有个运行在用户空间的工具才行。KVM 的开发者选择了比较成熟的开源虚拟化软件 QEMU 来作为这个工

具，QEMU 模拟 IO 设备（网卡，磁盘等），对其进行了修改，最后形成了 QEMU-KVM。



镜像基本操作

```
# 创建镜像
$ qemu-img create -f <format> <filename> <size>

# 查看镜像信息
```

(下页继续)

(续上页)

```
$ qemu-img info <filename>
```

格式转换

```
$ qemu-img convert -c -f <fmt> -O <out_fmt> -o <options> <fname> <out_fname>
```

扩容

```
$ qemu-img resize test.img +2G
```

qemu-img 快照

```
# 创建快照
$ qemu-img snapshot -c first_snapshot /var/lib/test.img

# 查询快照
$ qemu-img snapshot -l /var/lib/test.img
Snapshot list:
ID        TAG                 VM SIZE                DATE       VM CLOCK
1         first_snapshot            0 2017-07-11 09:30:40   00:00:00.000

# 使用快照
$ qemu-img snapshot -a 1 /var/lib/test.img

# 删除快照
$ qemu-img snapshot -d 1 /var/lib/test.img
```

qemu 镜像修改

有时候当我们的 qemu 镜像系统挂了或者是没有密码时，我们可以挂载 qemu 镜像，然后对镜像进行修改和文件备份。操作步骤如下：

- 挂载 qcow2

```
modprobe nbd max_part=8
qemu-nbd -c /dev/nbd0 vdisk01.img
mount /dev/nbd0p1 /mnt/
```

- 挂载 lvm 分区 qcow2 镜像

```
vgscan
vgchange -ay
mount /dev/VolGroupName/LogVolName /mnt/
```

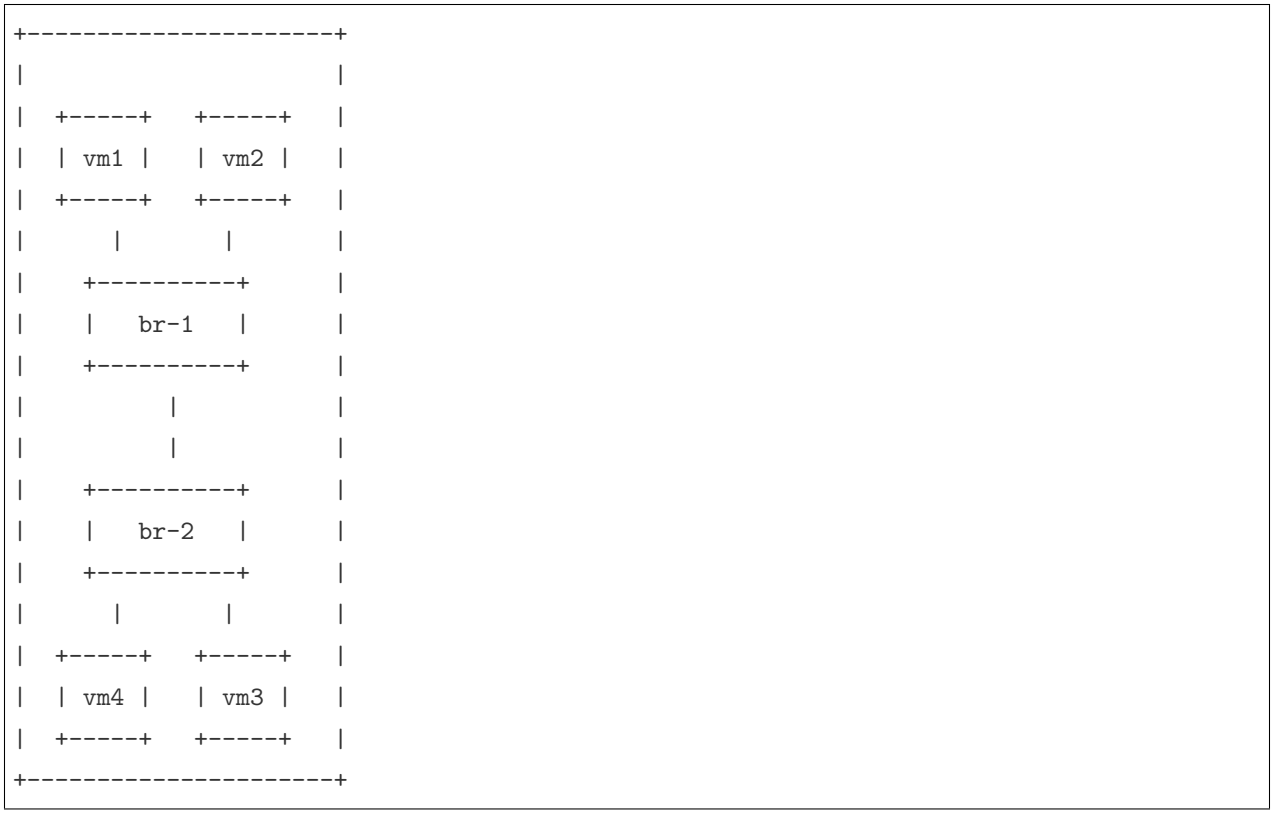
- 卸载 qcow2

```
umount /mnt/
vgchange -an VolGroupName
killall qemu-nbd
```

8.3.2 OVS

连接网桥

例如现有两个 ovs 桥 br-1 和 br-2, 这两个桥上都挂有虚机，如果不做处理，两个桥之间的虚机是不通的，如下图所示，要在 vm2 中访问 vm4, 需要把两个桥打通，或者有一个机器同时接到两个桥。



创建 peer 口：

```
# ovs-vsctl add-br br-1
# ovs-vsctl add-br br-2

# ovs-vsctl add-port br-1 patch-br2 -- set Interface patch-br2 type=internal
# ovs-vsctl add-port br-2 patch-br1 -- set Interface patch-br1 type=internal

# ovs-vsctl set interface patch-br2 options:peer=patch-br1
# ovs-vsctl set interface patch-br1 options:peer=patch-br2

# ovs-vsctl show
c0618d27-6364-4e3f-9e38-f5c520575954
    Bridge "br-1"
        Port "br-1"
            Interface "br-1"
                type: internal
        Port "patch-br2"
            Interface "patch-br2"
                type: internal
                options: {peer="patch-br1"}
    Bridge "br-2"
        Port "br-2"
            Interface "br-2"
                type: internal
        Port "patch-br1"
            Interface "patch-br1"
                type: internal
                options: {peer="patch-br2"}
```

VLAN 设置

```
# ovs-vsctl set port {port} vlan_mode=access
# ovs-vsctl set port {port} tag={segmentation_id}

# ovs-vsctl clear port {port} tag
# ovs-vsctl clear port {port} trunks
# ovs-vsctl clear port {port} vlan_mode
```

查看 interface 信息

```
# ovs-vsctl list interface tapaf2a9c21-d7
_uuid               : 7b32826b-dcf6-4e69-8b2c-0c8f7da946a0
admin_state         : down
bfd                 : {}
bfd_status          : {}
cfm_fault           : []
cfm_fault_status    : []
cfm_flap_count      : []
cfm_health          : []
cfm_mpid            : []
cfm_remote_mpids    : []
cfm_remote_opstate  : []
duplex              : []
error               : []
external_ids        : {attached-mac="fa:16:3e:d6:f8:96", iface-id="af2a9c21-d7c3-485b-
↪a869-af701ab53ba2", iface-status=active}
ifindex             : 0
ingress_policing_burst: 0
ingress_policing_rate: 0
lacp_current        : []
link_resets         : 0
link_speed          : []
link_state          : down
lldp                : {}
mac                 : []
mac_in_use          : []
mtu                 : []
mtu_request         : []
name                : "tapaf2a9c21-d7"
ofport              : 6
ofport_request      : []
options             : {}
other_config        : {}
statistics          : {collisions=0, rx_bytes=488, rx_crc_err=0, rx_dropped=0, rx_
↪errors=0, rx_frame_err=0, rx_over_err=0, rx_packets=5, tx_bytes=438, tx_dropped=0, tx_
↪errors=0, tx_packets=5}
status              : {driver_name=openvswitch}
type                : internal
```

查看流表

```
# ovs-ofctl dump-flows brbm
```

8.3.3 ovn

ovn-controller

ovs-vswitchd 的 Openflow 控制器来控制流量的转发。ovn-controller 是一种分布式 SDN 控制器。

ovn-northd

neutron & ovn

使用 neutron 创建的网络在 ovn 上以 switch 形式存在

```
# ovn-nbctl show
switch e90a7858-68d5-49bd-90f6-844863ecd511 (neutron-4f362a66-81ff-4784-8297-
↪6a411c68c58b) (aka net1)
    port 0c30be97-9757-4d1b-91a1-7428f22e051b
        type: localport
        addresses: ["fa:16:3e:b1:83:1b 192.168.1.2"]
    port 699211f0-dbd5-42d1-9097-ebaf104f1893
        addresses: ["fa:16:3e:5b:b7:c7 192.168.1.239"]
    port provnet-4f362a66-81ff-4784-8297-6a411c68c58b
        type: localnet
        tag: 100
        addresses: ["unknown"]
```

neutron 创建出的 port 在对应 OVN 网络的 switch 上以 port 形式存在，在 OVS 上以 port 形式存在

查看 bridge_mapping 信息

```
# ovs-vsctl get Open_vSwitch . external-ids:ovn-bridge-mappings
"extnet:br-ex,physnet1:br-data"
```

8.3.4 Virsh

virsh 是 libvirt 的 cli 工具，通过调用 libvirt 接口来控制虚拟机。

- 常用命令

```
# 查看虚机列表
$ virsh list
Id    Name                           State
----------------------------------------------------
 3     dev_test                       running

# 查看网络
$ virsh net-list

# dumpxml
$ virsh dumpxml <id>

# 查看 vnc 端口号
$ virsh vncdisplay <id>

# 增加网卡
$ virsh attach-interface --domain vm1 --type bridge --source br1
```

8.3.5 OpenStack 命令行速查表

认证 (keystone)

```
# 列出所有的用户
openstack user list

# 列出认证服务目录
openstack catalog list

# AZ
openstack availability zone list
```

计算 (nova)

```
# 列出规格类型
openstack flavor list

# 创建 flavor
openstack flavor create --ram 512 --disk 1 --vcpus 1 m1.tiny

# 列出实例，核实实例状态
```

(下页继续)

(续上页)

```
openstack server list

# 删除实例
openstack server delete bad544b4-46df-4b0a-9067-a1c680b687c9

# 显示实例详细信息
openstack server show NAME

# 查看云主机的控制台日志
openstack console log show MyFirstInstance

# 指定 user-data
openstack server create --user-data userdata.txt --image cirros-qcow2 --flavor m1.tiny␣
↪MyUserdataInstance2
```

网络 (neutron)

```
# 网络列表
openstack network list

# 创建安全组
neutron security-group-rule-create --direction ingress --ethertype IPv4 my_sg

# 创建 VLAN 网络
neutron net-create net1 --shared --provider:physical_network physnet1 --provider:network_
↪type vlan --provider:segmentation_id 16
```

镜像 (glance)

```
# 镜像列表
openstack image list

# 显示进度
glance image-create --name windows7 --visibility public --disk-format qcow2 --container-
↪format bare --file win7.qcow2 --progress

# 设置 VGA
glance image-create --name centos8 --visibility public --disk-format qcow2 --container-
↪format bare --file centos8.qcow --property hw_video_model=vga --progress
```

卷 (cinder)

```
# 卷列表
openstack volume list
```

对象存储 (swift)

```
# 列出容器
swift list

# 展示账户，容器以及对象的信息
swift stat CONTAINER
```

swift-ring-builder

```
# 必须到 ring 对应的目录下执行才行
cd /etc/swift
```

8.3.6 Neutron

防火墙

几个概念

- rule
- policy
- firewall

```
# openstack firewall group rule create --name test_rule_icmp_deny --action deny --
↪protocol icmp
+--------------------------+--------------------------------------+
| Field                    | Value                                |
+--------------------------+--------------------------------------+
| Action                   | deny                                 |
| Description              |                                      |
| Destination IP Address   | None                                 |
| Destination Port         | None                                 |
| Enabled                  | True                                 |
| ID                       | d7b3ffb2-3702-4208-888a-8dd9cfd78906 |
| IP Version               | 4                                    |
| Name                     | test_rule_icmp_deny                  |
| Project                  | 1fac5da33e6c48ffb990b6da2ec40020     |
| Protocol                 | icmp                                 |
```

(下页继续)

(续上页)

```
| Shared                | False                                |
| Source IP Address     | None                                 |
| Source Port           | None                                 |
| firewall_policy_id    | None                                 |
| project_id            | 1fac5da33e6c48ffb990b6da2ec40020     |
+-----------------------+--------------------------------------+
```

```
# openstack firewall group policy create --firewall-rule d7b3ffb2-3702-4208-888a-
↪8dd9cfd78906 icmp_policy
+----------------+-------------------------------------------+
| Field          | Value                                     |
+----------------+-------------------------------------------+
| Audited        | False                                     |
| Description    |                                           |
| Firewall Rules | [u'd7b3ffb2-3702-4208-888a-8dd9cfd78906'] |
| ID             | dd9f661a-9d61-447a-b32a-b0b2e0d744ca      |
| Name           | icmp_policy                               |
| Project        | 1fac5da33e6c48ffb990b6da2ec40020          |
| Shared         | False                                     |
| project_id     | 1fac5da33e6c48ffb990b6da2ec40020          |
+----------------+-------------------------------------------+
```

```
# openstack firewall group create --name icmp_test --ingress-firewall-policy dd9f661a-
↪9d61-447a-b32a-b0b2e0d744ca
+------------------+--------------------------------------+
| Field            | Value                                |
+------------------+--------------------------------------+
| Description      |                                      |
| Egress Policy ID | None                                 |
| ID               | e969c06a-bbc8-4b2a-88aa-beac16b94c7f |
| Ingress Policy ID | dd9f661a-9d61-447a-b32a-b0b2e0d744ca |
| Name             | icmp_test                            |
| Ports            | []                                   |
| Project          | 1fac5da33e6c48ffb990b6da2ec40020     |
| Shared           | False                                |
| State            | UP                                   |
| Status           | INACTIVE                             |
| project_id       | 1fac5da33e6c48ffb990b6da2ec40020     |
+------------------+--------------------------------------+
```

Neutron CLI

网络

```
# 创建 vlan 网络
neutron net-create <name> --shared --provider:physical_network physnet1  --
↪provider:network_type vlan --provider:segmentation_id 16
```

8.3.7 Swift

Rings

Storage Policies

- Account: 提供账号信息管理，定义 Container 的 namespace
- Container:
- Object: 存储数据内容

8.3.8 cloud-base

常用配置

1. 配置获取 metadata 的重试次数和间隔

   修改 *C:Program FilesCloudbase SolutionsCloudbase-Initconfcloudbase-init.con* 文件:

8.4 Virtualbox

8.4.1 Cli

```
# 显示所有虚拟机
vboxmanage list vms

# 开启虚拟机嵌套
vboxmanage modifyvm "Ubuntu 20.04 Server" --nested-hw-virt on
```

8.5 交换机

8.5.1 基础知识

交换机 VLAN 三种模式

- Access 模式
  - 端口接收报文：收到报文，判断是否有 VLAN 信息，如果没有则打上端口的 PVID()，并进行交换转发，否则丢弃（即使和缺省 VLAN 不一致）；
  - 端口发送报文：将报文的 VLAN 信息剥离，直接发送出去；
- Trunk 模式
  - 端口接收报文：收到报文，判断是否有 VLAN 信息，如果没有则打上端口的 PVID()，并进行交换转发，如果有
  - 端口发送报文：比较端口的 PVID 和将要发送报文的 VLAN 信息，如果两者相等则剥离 VLAN 信息并发送，如果不等，则直接发送；
- Hybrid 模式
  - 端口接收报文：收到报文，判断是否有 VLAN 信息，如果没有则打上端口的 PVID()，并进行交换转发，如果有则判断该 hybrid 端口是否允许该 VLAN 的数据进入，如果可以则转发，否则丢弃（此时端口的 untag 配置是不用考虑的，untag 配置只对发送报文起作用）
  - 端口发送报文：
    1. 判断该 VLAN 在本端口的属性（查看该端口对应哪些 VLAN 是 untag, 哪些 VLAN 是 tag）
    2. 如果是 untag 则剥离 VLAN 信息再发送，如果是 tag 则直接发送；

8.5.2 S5130

配置 vlan

```
system-view

# 创建 vlan
vlan 2

# 批量创建 vlan
vlan 100 to 200

# 加入端口
port GE1/0/1
```

(下页继续)

(续上页)

```
quit

# 配置 vlan 地址
interface vlan-interface 2
ip address 10.1.2.2 24
quite


# trunk 模式配置
interface GE1/0/4
port link-type trunk
# 支持 vlan2, vlan3, vlan4, vlan2000
port trunk permit vlan 2 to 4 2000
```

vlan 删除端口

```
interface GE1/0/1
no vlan 2
```

查看某个 vlan 信息

```
[root]show vlan 2
 VLAN ID: 2
 VLAN type: Static
 Route interface: Configured
 IPv4 address: 10.95.85.100
 IPv4 subnet mask: 255.255.255.0
 Description: VLAN 0002
 Name: VLAN 0002
 Tagged ports:
    Bridge-Aggregation48
 Untagged ports:
    GigabitEthernet1/0/7
```

8.5.3 LLDP 查询网络拓扑

1. 安装工具包

```
yum install lldpd
systemctl start lldpd
```

2. 查询 lldp 信息

```
# lldpcli show neighbors
-------------------------------------------------------------------------------
LLDP neighbors:
-------------------------------------------------------------------------------
Interface:    eth0, via: LLDP, RID: 1, Time: 0 day, 00:01:58
  Chassis:
    ChassisID:    mac 00:00:00:00:00:00
    SysName:      xxx
    SysDescr:     H3C Comware Platform Software, Software Version 7.1.070, Release␣
↪1312
                  H3C S5130-54C-HI
                  Copyright (c) 2004-2019 New H3C Technologies Co., Ltd. All rights␣
↪reserved.
    MgmtIP:       10.95.85.1
    Capability:   Bridge, on
    Capability:   Router, on
  Port:
    PortID:       ifname GigabitEthernet1/0/25
    PortDescr:    GigabitEthernet1/0/25 Interface
    TTL:          121
```

FAQ

1. intel X710 lldp agent 禁用

```
# 查看 driver 是否是 i40e
ethtool -i eth0 | grep "driver"

# disable lldp agent, 注意 0000:01:00.2 是 pci 号, 可以先 cat, 里面会显示网卡的名字
echo lldp stop > /sys/kernel/debug/i40e/0000:01:00.2/command
```

8.6 Docker

8.6.1 资源清理

1. 占用空间查看

```
docker system df
```

CHAPTER 9

消息队列

9.1 如何选择消息队列

9.1.1 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；

9.1.2 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；

9.1.3 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；

9.2 rabbitmq

9.2.1 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。

9.3 kafka

9.3.1 Topic

topic 相关操作

```
# 创建 topic
./kafka-topics.sh --create --zookeeper localhost:2181/kafka --replication-factor 1 --
↪partitions 1 --topic test

# 列出所有 topic
./kafka-topics.sh --zookeeper localhost:2181/kafka --list

# 查看 topic
./kafka-topics.sh --zookeeper localhost:2181 --describe --topic test

# 增加分区数量 (分区只能增加不能减少)
./kafka-topics.sh --zookeeper localhost:2181 --alter --topic test --partitions 4

# 删除 topic
./kafka-topics.sh --zookeeper localhost:2181 --delete --topic test
```

9.3.2 Group

group 相关操作

```
# 列出所有 consumer group
./kafka-consumer-groups.sh --bootstrap-server localhost:9092 --list

# 某个 consumer group 信息
./kafka-consumer-groups.sh --bootstrap-server localhost:9092 --describe --group group_id_
↪qa

# 将某个 group 的 topic 重置到 earliest
./kafka-consumer-groups.sh --bootstrap-server localhost:9092 --group example --topic␣
↪test --execute --reset-offsets --to-earliest
```

9.3.3 Message

```
# 发送消息
./kafka-console-producer.sh --broker-list localhost:9092 --topic test

# 消费消息
./kafka-console-consumer.sh  --bootstrap-server localhost:9092 --topic test --from-
↪beginning

# 消费一条消息
./kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic test --max-
↪messages 1

# 指定 group 消费一条消息
./kafka-console-consumer.sh --bootstrap-server localhost:9092 --group g_group --topic␣
↪test --max-messages 1
```

9.3.4 丢消息

丢消息的情况

- 生产者发送消息。丢消息的原因主要是生产者一般发送消息到 kafka 都是异步的，所以有可能失败之后没有处理。解决方案一般是进行失败重试，或者设置发消息的方法设置成同步。
- 消费者消费消息。丢消息的原因主要是消费者拉取消息后会设置 autocommit offset，但是消费者拉取消息后因为某些原因宕机后没有处理这个数据导致丢消息。解决方案可以先关掉 autocommit offset，等待业务处理完后再提交。但是这个方案要注意不要让消费者消费过长让 kafka 超时踢出消费者组导致 rebalance
- kafka 中 topic 及 replicate 的主从同步。

注解: 前提知识：kafka 的高可用方案是把一个 topic 在多台 kafka 实例上做了副本 replicate （当然也分了 leader 和 follower，用的是 zk 管理，新版本的 kafka 也去除了 zk 管理的依赖）。这时一般的读写都是从 leader 上操作的，当 leader 接收了一个消息，没有来得及同步到 follwer 前 broker 挂掉了，会导致丢消息。

解决方案

- 全同步。可配置一个参数 acks = all，也就是类似于 Mysql 主从复制中的全同步，等到所有 topic 副本都同步完才返回给生产者发送成功。

- 半同步。可配置一个参数 mini.sync.replicas。也就是 Mysql 的半同步，可以设置同步到多少个副本就返回给生产者发送成功
- 新增副本个数，增大同步到的概率。可配置一个参数 replication.factor，也就是增大副本的个数，因为副本越多，同步到的概率就越大，但会导致数据冗余。

9.4 celery

9.4.1 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",
```

(下页继续)

(续上页)

```
    "children": [],
    "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。

9.4.2 task

调用异步任务的方式，用下例子说明:

```
@app.task
def add(x, y):
    return x + y
```

- task.delay()

  适合简单的 task 调用，eg:

```
add.delay(1, 2)
```

- task.apply_async()

  可以加控制参数

- app.send_task()

CHAPTER 10

OpenResty

10.1 Nginx

10.1.1 常用配置

1. 查看 nginx 安装模块

10.2 lua

- access_by_lua
- access_by_lua_block
- content_by_lua
- content_by_lua_file

设置 header: ngx.req.set_header(header_name, value) # 清除 header: ngx.req.clear_header(header)

清除所有的 headers_M = {}

```
for header, _ in pairs(ngx.req.get_headers()) do
    ngx.req.clear_header(header)
end
```

CHAPTER 11

监控

11.1 prometheus

11.1.1 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

11.1.2 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

11.1.3 疑问

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

11.2 Go 使用 prometheus

```
import (
    "github.com/prometheus/client_golang/prometheus"
)

counter := prometheus.NewCounter(prometheus.CounterOpts{
    Name:       "rpc_durations_seconds",
            Help:       "RPC latency distributions.",
            Objectives: map[string]string{0.5: 0.05, 0.9: 0.01, 0.99: 0.001},
})
```

CHAPTER 12

CI/CD

12.1 Jenkins

12.1.1 Pipeline

完整的文档参考：https://www.jenkins.io/zh/doc/

- 触发其它 job

```
// 等待
stage("step1") {
   steps {
        build job: 'job1', parameters: []
    }
}

// 加参数，不等待
stage("step1") {
     steps {
         build job: 'job1', parameters: [string(name: 'Name', value: 'Baz2')],␣
↪wait: false
     }
}
```

(下页继续)

(续上页)

```
// 串行触发多个 job
stage("step1") {
     steps {
         build job: 'job1', parameters: []
         build job: 'job2', parameters: []
     }
}

// 并行触发多个 job, 并等待完成
stage('step1') {
   def jobs = [:]
   jobs[0] = {build job: 'job1', parameters: [string(name: 'Name', value: param)],␣
↪quietPeriod: 2}
   jobs[1] = {build job: 'job2', parameters: [string(name: 'Name', value: param)],␣
↪quietPeriod: 2}
   parallel jobs
}
```

- 指定分支和文件改变条件

```
stage("step1") {
    when{
        environment name: 'GIT_BRANCH', value: 'origin/test'
        anyOf {
            changeset 'go.mod'
            changeset 'go.sum'
            changeset 'docker/Dockerfile'
        }
}
```

- 失败发送邮件

```
pipeline {
    post {
        failure {
            emailext(
                subject: "Jenkins build is ${currentBuild.result}: ${env.JOB_NAME} #
↪${env.BUILD_NUMBER}",
                mimeType: "text/html",
                body: """<p>Jenkins build is ${currentBuild.result}: ${env.JOB_NAME}
↪ #${env.BUILD_NUMBER}:</p>
```

(下页继续)

(续上页)

```
                        <p>Check console output at <a href="${env.BUILD_URL}console
↪">${env.JOB_NAME} #${env.BUILD_NUMBER}</a></p>""",
                recipientProviders: [[$class: 'CulpritsRecipientProvider'],
                                     [$class: 'DevelopersRecipientProvider'],
                                     [$class: 'RequesterRecipientProvider']]
            )
        }
    }
}
```

- 控制 job 每次只有一个在运行

```
pipeline {
    options {
        disableConcurrentBuilds()
    }
}
```

- 构建参数

```
pipeline {
    agent any
    parameters {
        string(name: 'PERSON', defaultValue: 'Mr Jenkins', description: 'Who should␣
↪I say hello to?')
        text(name: 'BIOGRAPHY', defaultValue: '', description: 'Enter some␣
↪information about the person')
        booleanParam(name: 'TOGGLE', defaultValue: true, description: 'Toggle this␣
↪value')
        choice(name: 'CHOICE', choices: ['One', 'Two', 'Three'], description: 'Pick␣
↪something')
        password(name: 'PASSWORD', defaultValue: 'SECRET', description: 'Enter a␣
↪password')
    }

    stages {
        stage("parameters test") {
            steps {
                sh """
                    echo "${params.PERSON}"
                """
```

(下页继续)

(续上页)

```
                }
            }
        }
    }
}
```

- 指定 agent

```
// 指定所有 agent
pipeline {
   agent any
}

// 固定 agent
pipeline {
   agent {
       label "slave"
   }
}
```

- stage 失败了继续执行

```
pipeline {
   agent any
   stages {
       stage('1') {
           steps {
               sh 'exit 0'
           }
       }
       stage('2') {
           steps {
               catchError(buildResult: 'SUCCESS', stageResult: 'FAILURE') {
                   sh "exit 1"
               }
           }
       }
       stage('3') {
           steps {
               sh 'exit 0'
           }
```

(下页继续)

(续上页)

```
        }
    }
}
```

- 按条件触发

```
// 根据分支和文件修改来控制
stage("step1") {
    when {
        anyOf {
            environment name: 'GIT_BRANCH', value: 'origin/master'
            changeset 'file_a'
            changeset 'file_b'
        }
    }
}
```

- 发布单元测试结果

```
// 这里使用的是 html publish 插件
// 需要在 jenkins 上执行下： System.setProperty("hudson.model.
↪DirectoryBrowserSupport.CSP", "")
// 否则 chrome 浏览器会禁用 css 和 js
post {
    always {
        publishHTML (target : [allowMissing: false,
            alwaysLinkToLastBuild: true,
            keepAll: true,
            reportDir: 'htmlcov',
            reportFiles: 'index.html',
            reportName: 'Code Coverage',
            reportTitles: 'Code Coverage'])
    }
}
```

12.1.2 Plugins

插件离线下载地址：https://updates.jenkins-ci.org/download/plugins/

设置国内镜像源

```
https://mirrors.tuna.tsinghua.edu.cn/jenkins/updates/update-center.json
http://mirror.xmission.com/jenkins/updates/update-center.json
```

实用插件推荐

1. Allure Jenkins Plugin

12.2 Ansible

解决 ssh 首次登录认证问题

在脚本执行目录下添加 *ansible.cfg* 文件，内容如下：

12.3 Gitlab

设置 gitlab 状态

updateGitlabCommitStatus

- pending
- running
- canceled
- success
- failed

12.4 Prometheus

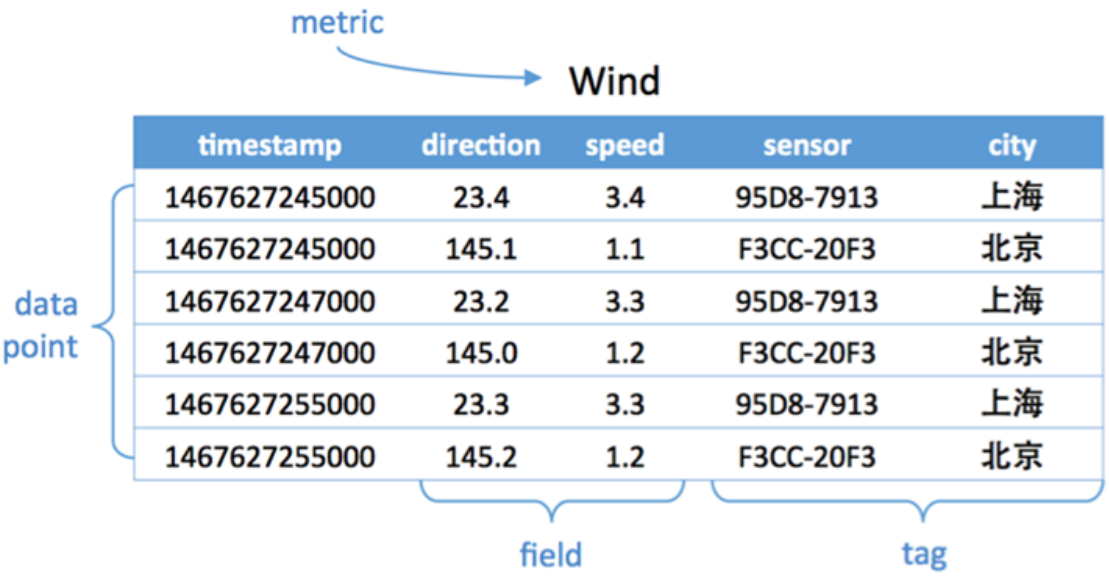
12.4.1 时序数据库

时序数据是基于时间的一系列的数据。在有时间的坐标中将这些数据点连成线，往过去看可以做成多纬度报表，揭示其趋势性、规律性、异常性; 往未来看可以做大数据分析，机器学习，实现预测和预警。

时序数据库就是存放时序数据的数据库，并且需要支持时序数据的快速写入、持久化、多纬度的聚合查询等基本功能。

对比传统数据库仅仅记录了数据的当前值，时序数据库则记录了所有的历史数据。同时时序数据的查询也总是会带上时间作为过滤条件。

下面用一张图来说明时序数据库



概念：

- metric: 度量，相当于关系型数据库中的 table。
- data point: 数据点，相当于关系型数据库中的 row。
- timestamp：时间戳，代表数据点产生的时间。
- field: 度量下的不同字段。比如位置这个度量具有经度和纬度两个 field。一般情况下存放的是会随着时间戳的变化而变化的数据。
- tag: 标签，或者附加信息。一般存放的是并不随着时间戳变化的属性信息。timestamp 加上所有的 tags 可以认为是 table 的 primary key。

如上图，度量为 Wind，每一个数据点都具有一个 timestamp，两个 field：direction 和 speed，两个 tag：sensor、city。它的第一行和第三行，存放的都是 sensor 号码为 95D8-7913 的设备，属性城市是上海。随着时间的变化，风向和风速都发生了改变，风向从 23.4 变成 23.2; 而风速从 3.4 变成了 3.3。

参考文献：

1. https://www.sohu.com/a/237660940_130419

12.4.2 数据存储

Prometheus 内置了一个本地的时序数据库，同时也支持配置远端数据库。

本地存储

远程存储集成

12.4.3 PromQL

PromQL (Prometheus Query Language) 是 Prometheus 自己开发的数据查询 DSL 语言，语言表现力非常丰富，内置函数很多，在日常数据可视化以及 rule 告警中都会使用到它。

查询结果类型

PromQL 查询结果主要有 3 种类型：

- 瞬时数据 (Instant vector): 包含一组时序，每个时序只有一个点，例如：http_requests_total
- 区间数据 (Range vector): 包含一组时序，每个时序有多个点，例如：http_requests_total[5m]
- 纯量数据 (Scalar): 纯量只有一个数字，没有时序，例如：count(http_requests_total)

查询条件

Prometheus 存储的是时序数据，而它的时序是由名字和一组标签构成的，其实名字也可以写出标签的形式，例如 http_requests_total 等价于 *{name=" http_requests_total" }*。

操作符

Prometheus 查询语句中，支持常见的各种表达式操作符，例如

1. 算术运算符

   支持的算术运算符有 +，-，*，/，%，^, 例如 http_requests_total * 2 表示将 http_requests_total 所有数据 double 一倍。

2. 比较运算符

   支持的比较运算符有 ==，!=，>，<，>=，<=，例如 http_requests_total > 100 表示 http_requests_total 结果中大于 100 的数据。

3. 逻辑运算符

   支持的逻辑运算符有 and，or，unless, 例如 http_requests_total == 5 or http_requests_total == 2 表示 http_requests_total 结果中等于 5 或者 2 的数据。

4. 聚合运算符

   sum，min，max，avg，stddev，stdvar，count，count_values，bottomk，topk，quantile，例如 max(http_requests_total) 表示 http_requests_total 结果中最大的数据。

内置函数

Prometheus 内置不少函数，方便查询以及数据格式化，例如将结果由浮点数转为整数的 floor 和 ceil，

```
floor(avg(http_requests_total{code="200"}))
ceil(avg(http_requests_total{code="200"}))
```

```
sum(rate(http_server_requests_seconds_count{job="presell-pro", uri="/helloWorld"}[5m]))␣
↪by (uri)
```

12.4.4 内置函数

- ans()
- absent()
- absent_over_time()
- ceil()
- changes()
- clamp()
- clamp_max()
- clamp_min()
- day_of_month()
- day_of_week()
- days_in_month()
- delta()
- deriv()
- exp()
- floor()
- histogram_quantile()
- holt_winters()
- hour()
- idelta()
- increase()
- irate()

- label_join()
- label_replace()
- ln()
- log2()
- log10()
- minute()
- month()
- predict_linear()
- rate()

  rate(v range-vector) 计算区间数据每秒的平均值。

  eg: 获取最近 5min 的 QPS

  rate 应该只用于 *Counter* 类型的数据。
- resets()
- round()
- scalar()
- sgn()
- sort()

  sort(v instant-vector) 按递增顺序排
- sort_desc()

  逆序排
- sqrt()
- time()
- timestamp()
- vector()
- year()
- aggregation>_over_time()

12.4.5 Metric

指标类型

Prometheus 里存储的数据都是 *float64*，可以把数据分为如下四大类：

1. Counter: 用于计数，这个值会一直增加，不会减少，可用于请求次数、任务完成数、错误发生次数。
2. Gauge: 一般的数值，可大可小，例如：温度变化、内存使用变化。
3. Histogram
4. Summary

12.4.6 Pushgateway

正常情况下，prometheus 是拉模式，prometheus server 从各个数据源拉取 metrics 信息。pushgateway 是我们将数据主动 push 到 Pushgateway，然后 prometheus 从 pushgateway 拉取数据。

12.5 Granfa

CHAPTER 13

软件架构

13.1 软件架构的基本概念

13.1.1 1. 什么是架构

架构就是对系统中的实体以及实体之间的关系所进行的抽象描述，是一些列的决策。

13.1.2 2. 什么是架构图

CHAPTER 14

分布式系统

14.1 CAP

14.2 共识算法

常见的共识算法

- Paxos
- Raft

CHAPTER 15

项目管理

15.1 jira

15.1.1 看板

1. 配置看板筛选器

15.1.2 筛选器

1. 常见筛选器

```
# 待我确认关闭
project = XXX AND issuetype = Bug AND reporter = currentUser() AND status in ("Check
↪", Fixed) ORDER BY created DESC

# 我汇报的所有问题（处理中）
project = XXX AND issuetype = Bug AND reporter = currentUser() AND statusCategory =
↪"In Progress" ORDER BY created DESC

# 所有分配给我的问题（处理中）
project = XXX AND issuetype = Bug AND assignee = currentUser() AND statusCategory =
↪"In Progress" ORDER BY created DESC
```

（下页继续）

(续上页)

```
# 所有未完成 BUG
project = XXX AND issuetype = Bug AND statusCategory = "In Progress" ORDER BY␣
↪created DESC
```

CHAPTER 16

Sphinx

16.1 语法

请参考 reference。

16.2 自定义样式

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

```
.wy-nav-content {
    max-width: 1400px !important;
}
```

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

CHAPTER 17

工具

17.1 VSCode

有用的快捷键

```
# 打开命令面板
 A

# 打开文件面板
P

# 隐藏目录
B

# workbench.action.toggleActivityBarVisibility
```

推荐插件

1. Font Switcher
2. Back & Forth

CHAPTER 18

面试

18.1 编程语言

18.1.1 python

1. 多重继承，如果基类有相同方法，从左向右寻找；
2. python 中如何管理内存，python 有一个私有堆内存来放置所有对象和数据结构；
3. 谈一下 GC;
4. python 协程和 golang 协程有什么区别;
5. 什么是上下文管理器；
6. 什么是闭包；
7. 深拷贝和浅拷贝区别；
8. 数组和元组的区别；// (1, 2), [1, 2] 占用内存大小相同吗?
9. type 作用;
10. **列表反置；**
    - a. a[::-1]
    - a.reverse() # 修改原数组；
11. **判断变量类型**

- type
- isinstance

12. **如何判断两个对象相同**
    - is
    - == # __eq__

18.1.2 go

1. 解释下 GPM;
2. Slice 和数组的区别（切片是指向数组的指针）;
3. 怎么实现同步（waitGroup）;
4. 什么是 Context;
5. defer 的执行顺序;
6. 怎么进行异常处理;
7. channel 是什么?
8. 有那些方式安全读写共享变量（Mutex)?
9. 无缓冲 chan 的发送和接收是否同步?
10. JSON 标准库对 nil slice 和空 slice 的处理是一致的吗
11. init() 函数是什么时候执行的
12. 2 个 interface 可以比较吗 ?
13. = 和:= 的区别
14. Go 支持默认参数或可选参数吗

18.1.3 shell

1. 单引号和双引号区别;
2. 如何只输出一个文件第十行;
3. 系统磁盘满了，如何找出哪个目录或文件占用空间比较大;

18.2 计算机网络

1. HTTP 长连接和短连接的区别;
2. HTTP 常用状态码：204, 202 是什么意思?

3. 谈谈什么是多路 IO 复用，以及常用的方式，select, poll, epoll 之间的区别

18.3 数据库

redis

1. **常用数据结构**
    - string
    - list
    - set
    - sorted set
    - hash
2. 怎么实现分布式锁
3. 持久化有哪些方式
4. 淘汰策略

18.4 数据结构

18.5 消息队列

1. 消息队列的作用；
2. kafka 是推模式还是拉模式；
3. kafka 经常 reblance 如何解决；
4. **kafka 为什么快？**
    - Cache Filesystem Cache PageCache 缓存
    - 顺序写 由于现代的操作系统提供了预读和写技术，磁盘的顺序写大多数情况下比随机写内存还要快。
    - Zero-copy 零拷技术减少拷贝次数
    - Batching of Messages 批量量处理。合并小的请求，然后以流的方式进行交互，直顶网络上限
    - Pull 拉模式 使用拉模式进行消息的获取消费，与消费端处理能力相符。
5. 如何保证重复消息只消费一次；

18.6 操作系统

18.6.1 Linux

1. 请求超时，如何分析；
2. 文件权限，解释 777
3. lsattr
4. 进程查看
5. 进程堆栈分析

系统/服务日志

1. 服务日志查看
2. messages
3. journactl
4. dmesg

网络

1. IO 查看
2. 端口查找
3. TCP 统计

18.7 CI/CD

1. Jenkins 怎么统一管理 job;
2. 怎么编写 gitlab pipline

18.8 工具

1. git stash 怎么用；
2. git cherry-pick 怎么用;

18.9 监控告警

1. prometheus 怎么统计 QPS, nginx + 多 api 代理

18.10 算法

1. 抽奖算法，1-100 中随机抽一个;

18.11 VUE

1. VUE 的生命周期及理解
2. v-if 和 v-show 的区别
3. vue 组件的通信
4. computed 和 watch 的用法和区别?