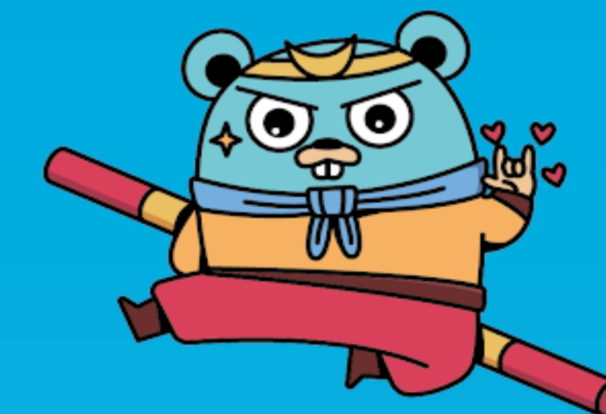


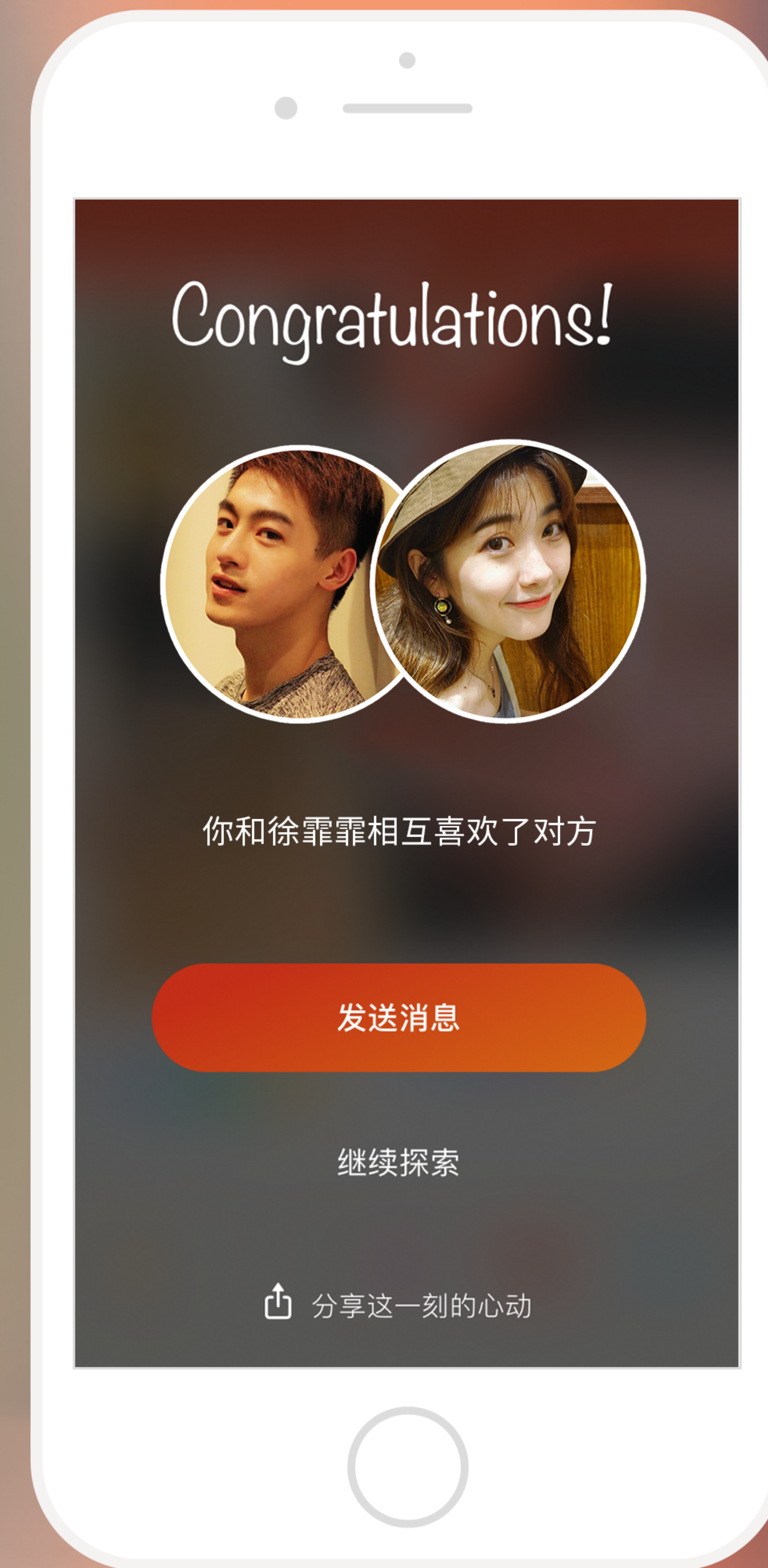
# 基于MINIO的对象存储方案在探探的实践

于乐

探探科技
yule@tantanapp.com

探探 Gopher China 2019

左滑无感 右滑喜欢
操作简单 超快认识人
探探
徐霏霏


互赞开始聊天 零骚扰
全新的相遇方式 不再错过缘分
Congratulations!
你和徐霏霏相互喜欢了对方
发送消息
继续探索
分享这一刻的心动

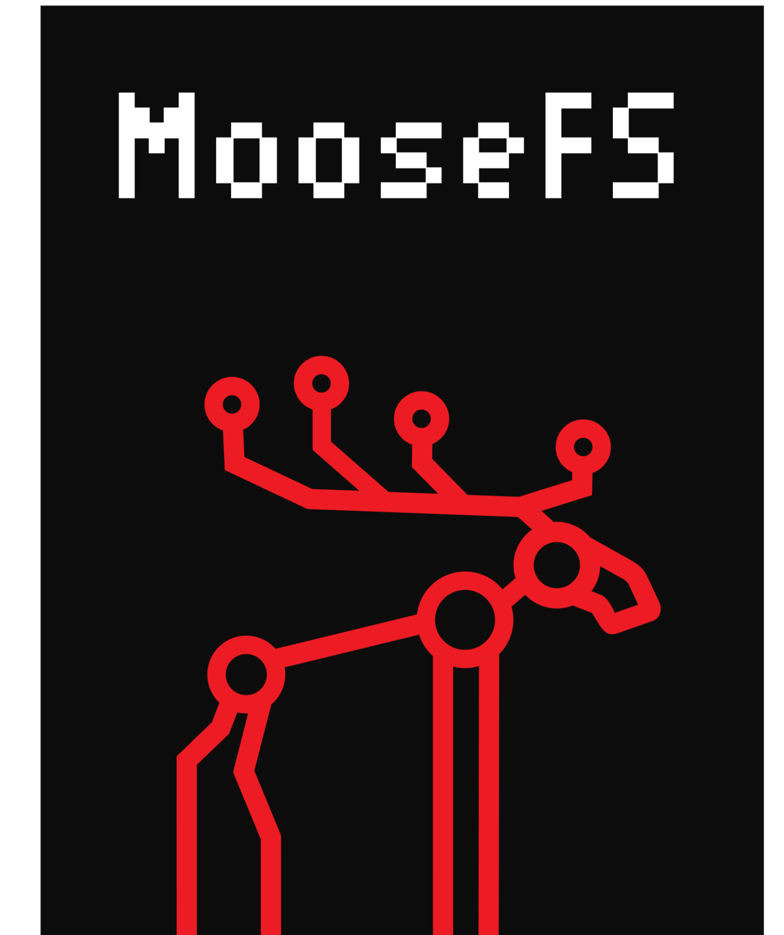
破冰利器 随心畅聊
还有好玩的真心话大冒险
fafa
从前有只丑小鸭，虽然很丑，但是人们发现它
但是味道不错！
输入聊天内容...
经典真心话
私密真心话
拍照
图片/视频
位置

# 存储现状

# 调研目前已有的开源方案

Ambry

FastFS

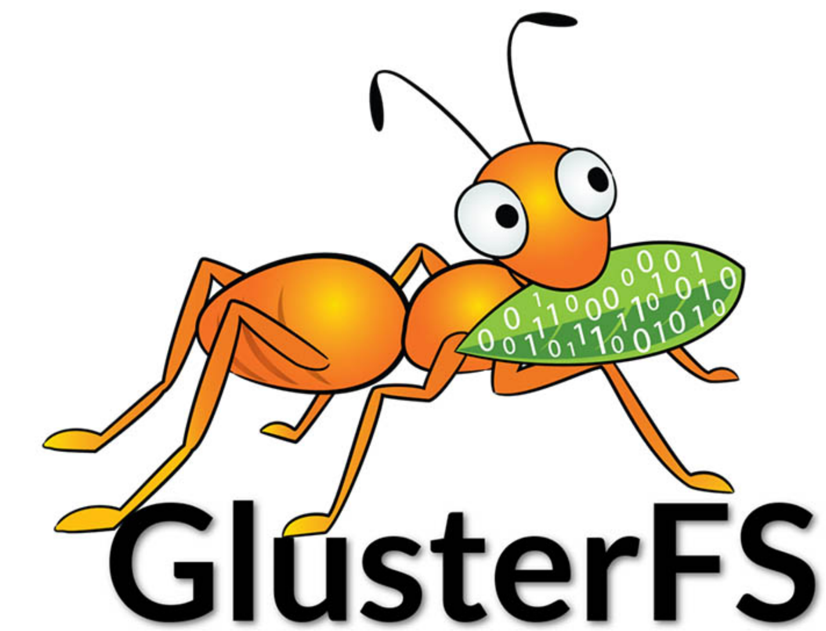


MogileFS

SWIFT

ceph

TFS



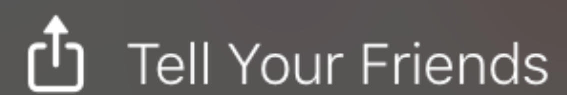
GlusterFS

# Congratulations!

You and MINIO liked each other.

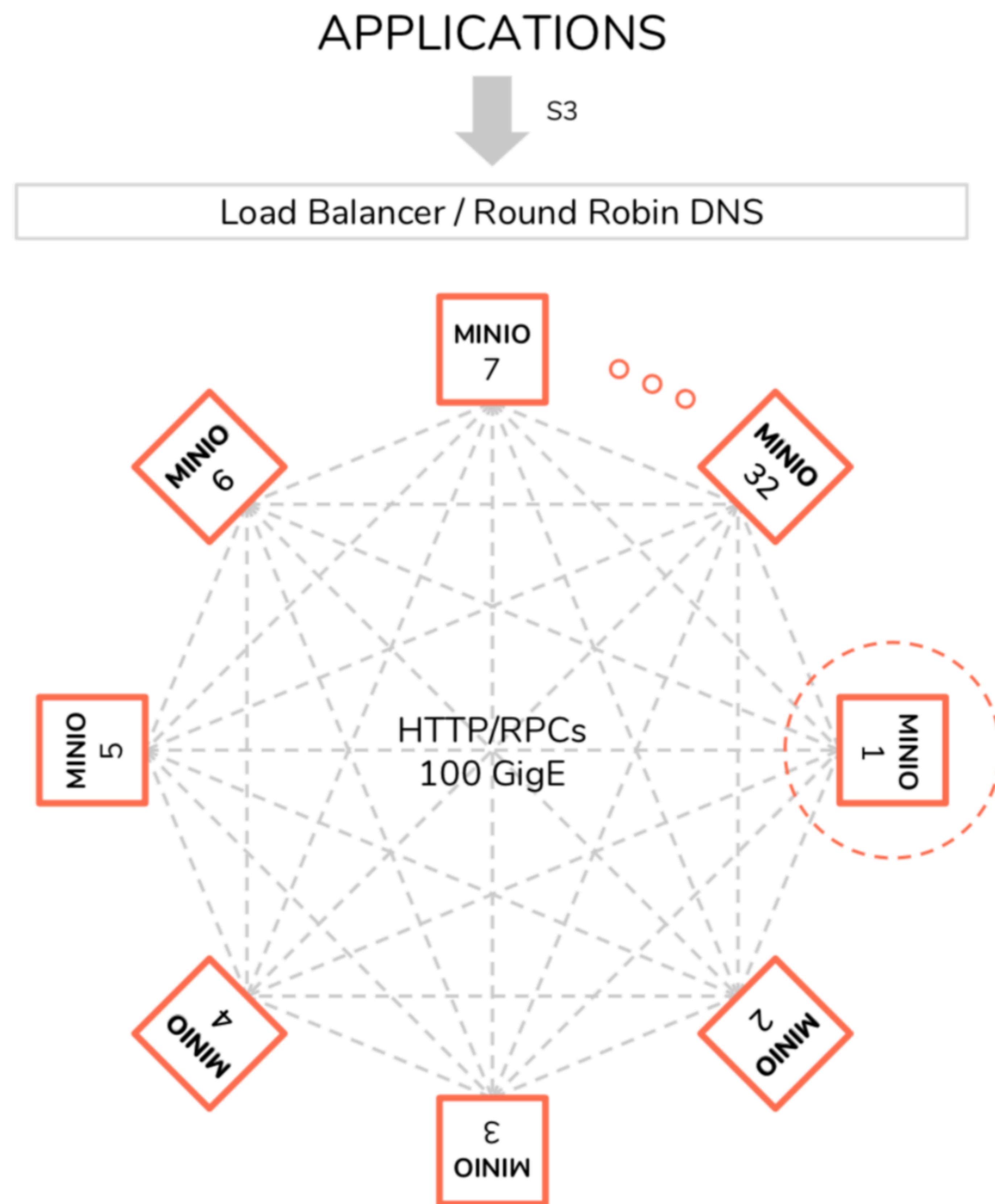
Let's Chat

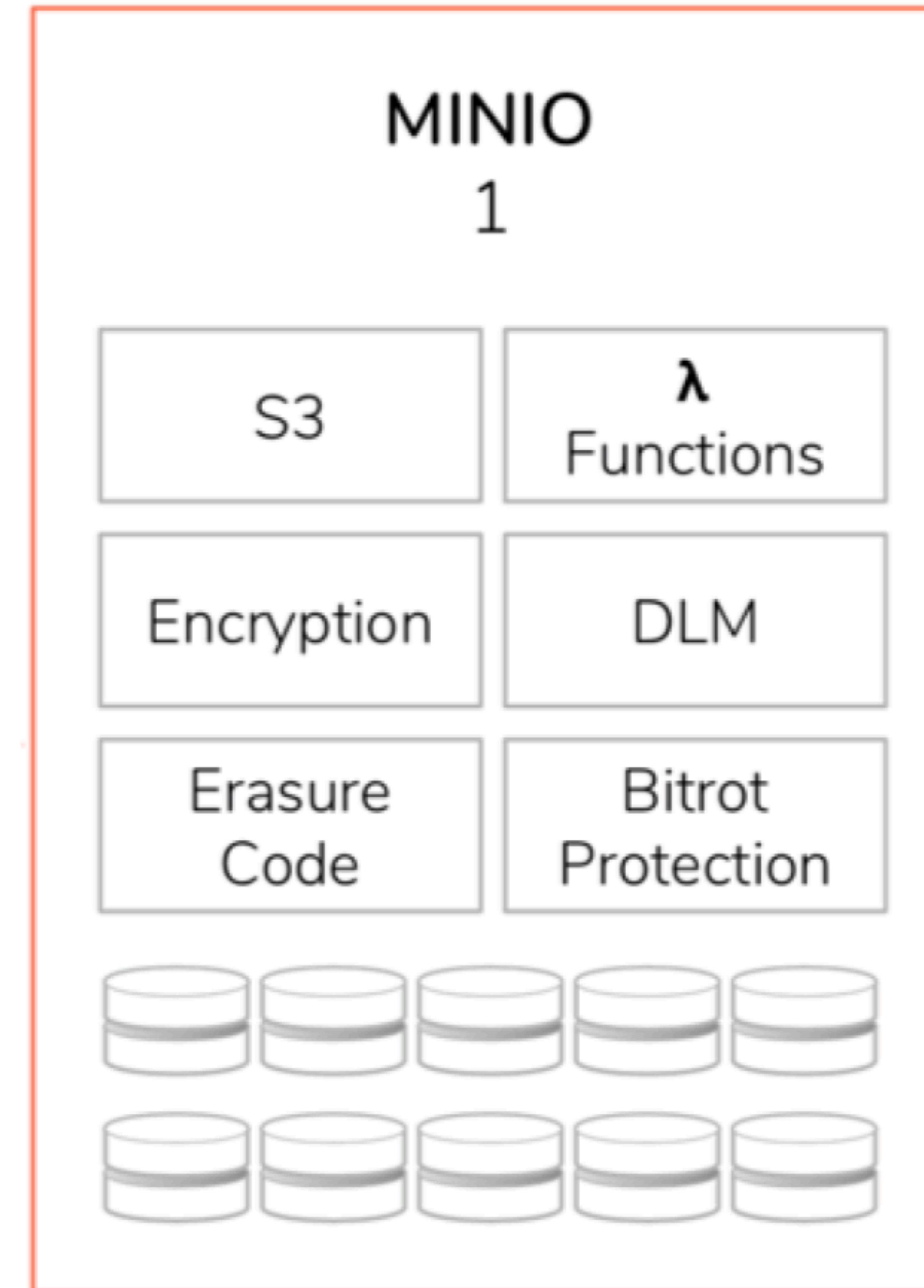
Keep Swiping



# 初探 MINIO

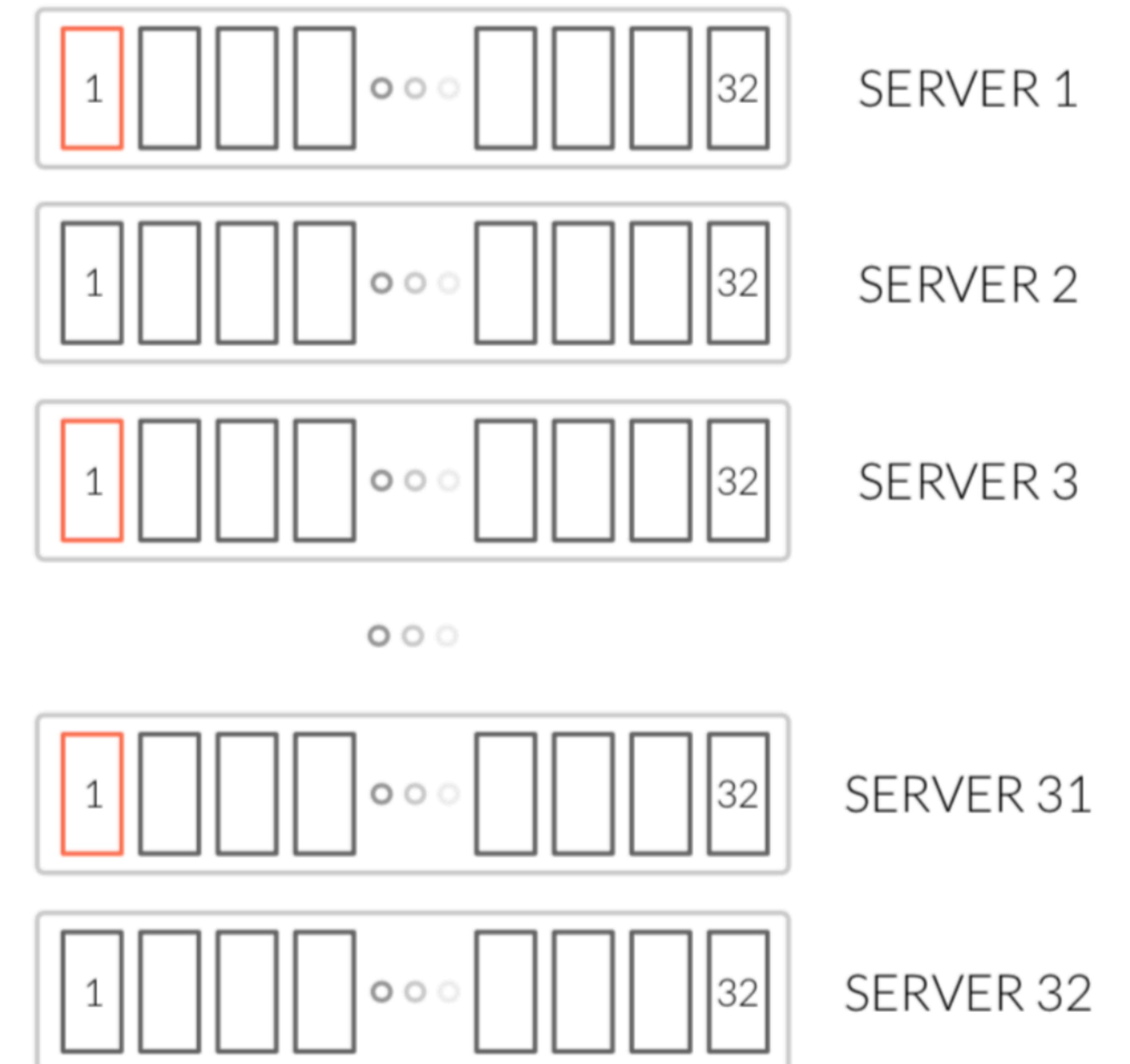
# MINIO 架构



# 数据分布以及可靠性

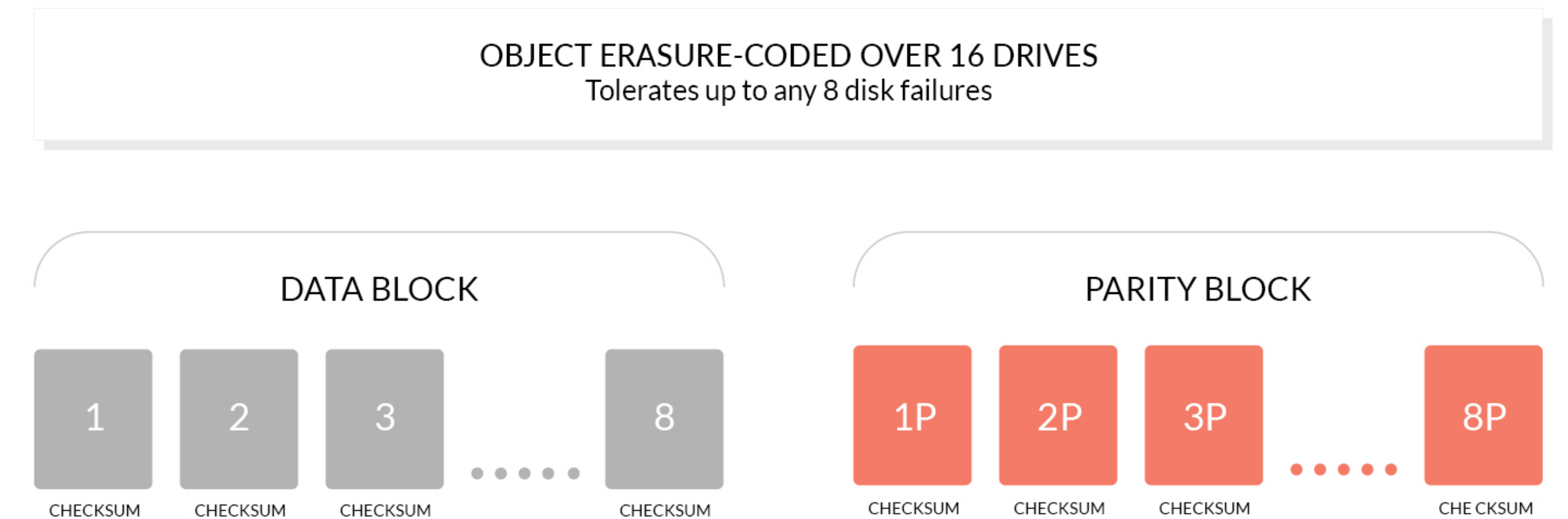
- Drive, 可以简单理解为一块硬盘
- Set，一组 Drive 的集合
  - 一个对象存储在一个 Set 上
  - 一个集群划分为多个 Set
  - 一个 Set 包含的 Drive 数量是固定的
    - 默认由系统根据集群规模自动计算得出
    - MINIO_ERASURE_SET_DRIVE_COUNT
  - 一个 SET 中的 Drive 尽可能分布在不同节点上



# 数据编码

Erasure Code, a mathematical algorithm to reconstruct missing or corrupted data.

- 将一个对象编码成若干个数据块和跟校验块
- Reed-Solomon
- 低冗余，高可靠

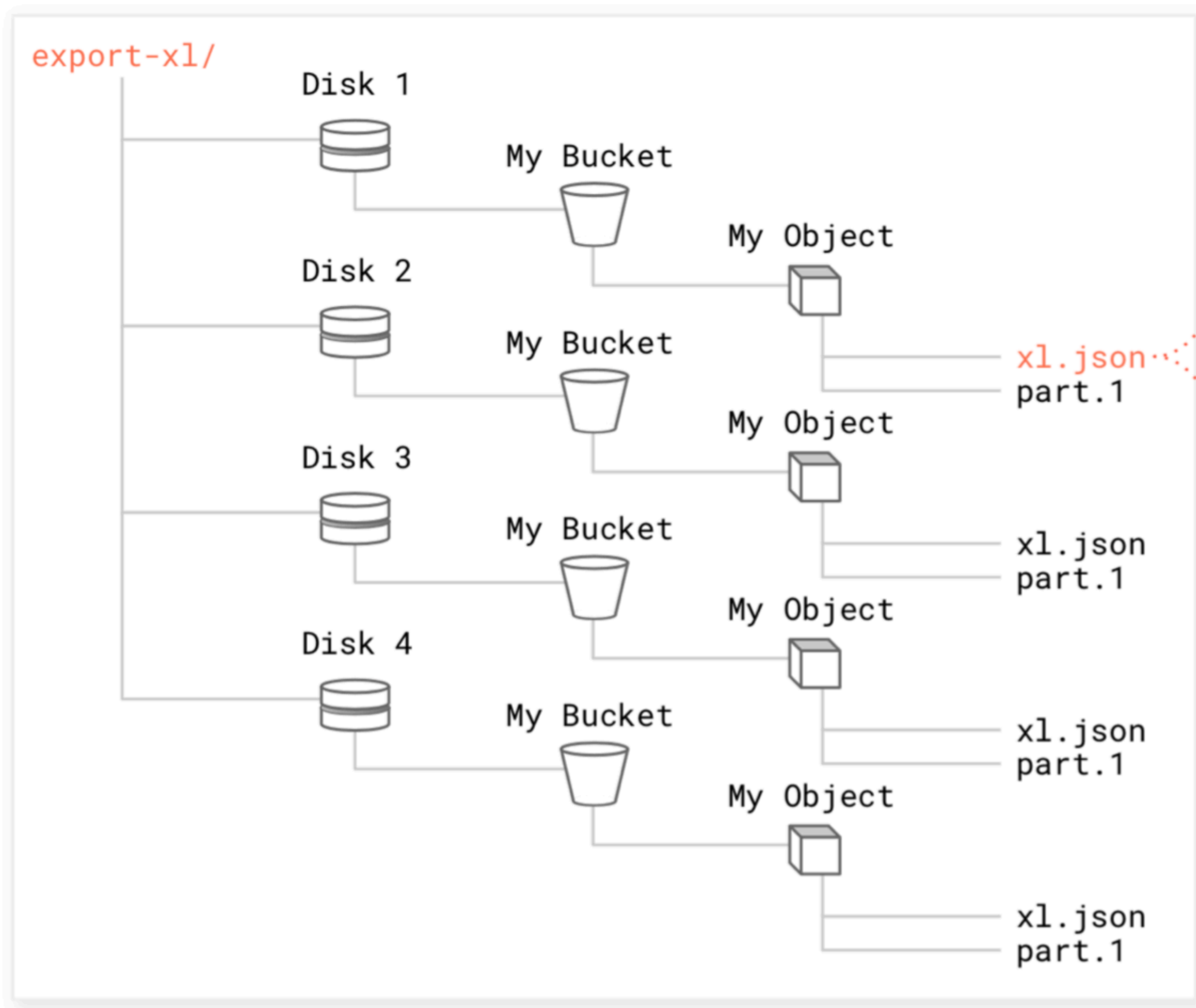



Bit Rot Protection

- HighwayHash

# MINIO

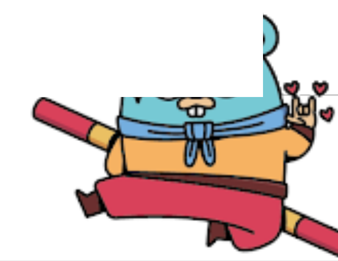
## Object Erasure Code



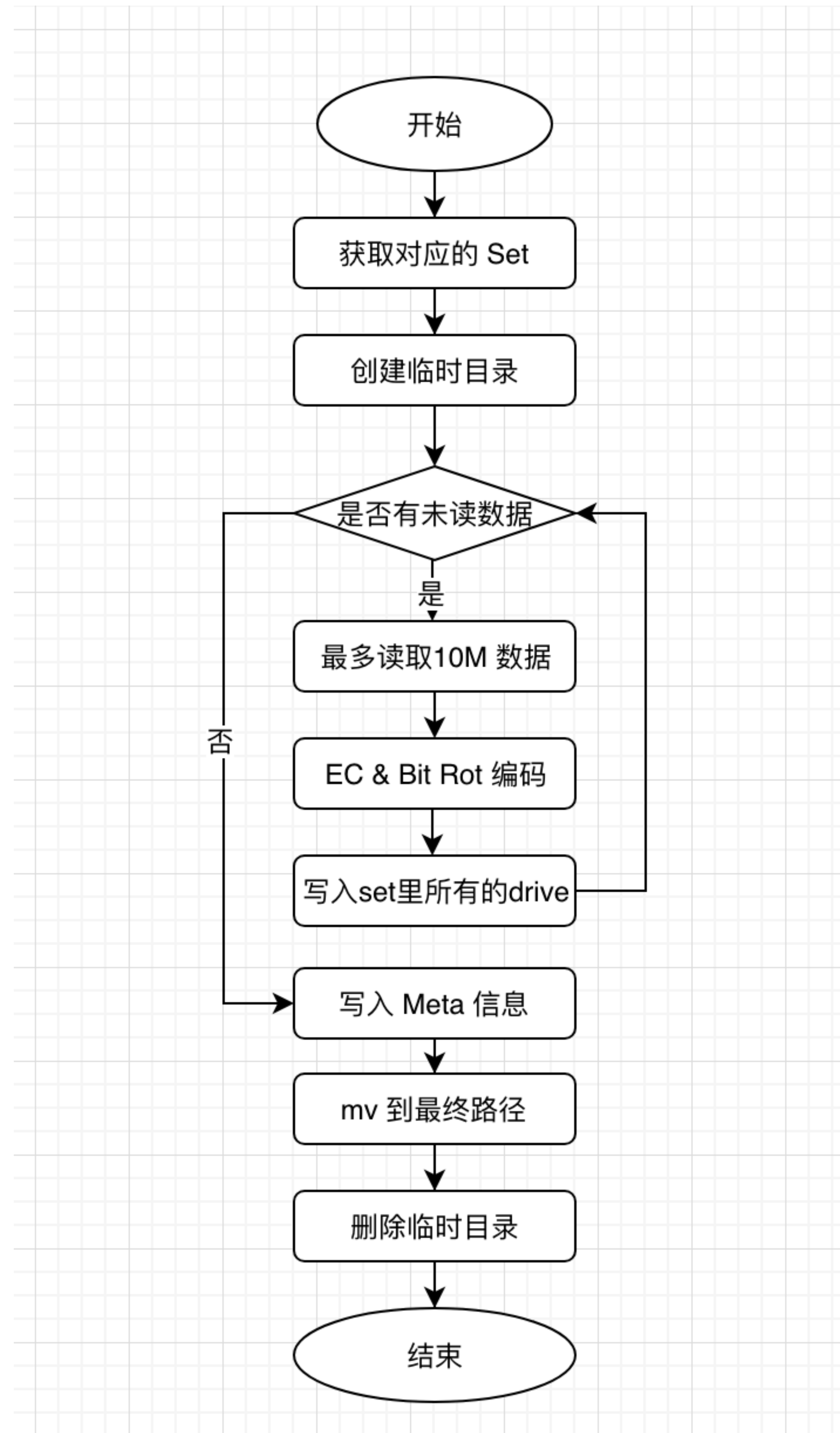


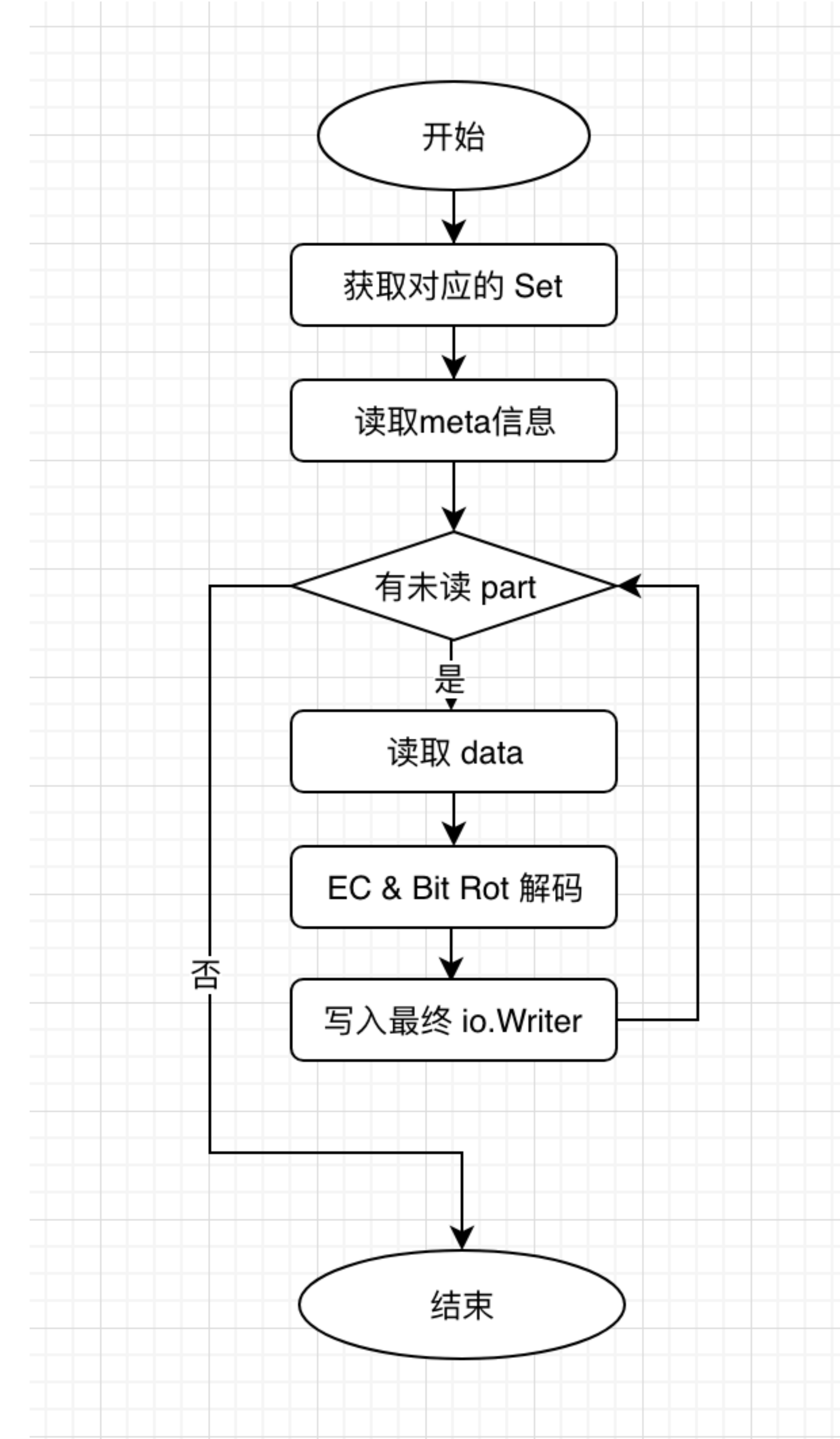
```
{
  "version": "1.0.1",
  "format": "xl",
  "stat": {
     "size": 2286,
     "modTime": "2017-12-02T00:24:20.975968336Z",
},
  "erasure": {
     "algorithm": "klauspost/reedsolomon/vandermonde",
     "data": 2,
     "parity": 2,
     "blockSize": 10485760,
     "index": 2,
     "distribution": "[
      2,
      3,
      4,
      1
    ],
     "checksum": [
      {
        "name": "part.1",
        "algorithm": "blake2b",
        "hash": "c24fa0451fd85a3a482c...b672b7f"
      }
    ]
  },
   "minio": {
     "release": "DEVELOPMENT.GOGET"
  },
   "meta": {
     "content-type": "application/octet-stream",
     "etag": "c1d217c52d44c9eab00e81496b2b91b6"
  },
   "parts": [
    {
       "number": 1,
       "name": "part.1",
       "etag": "",
       "size": 2286
    }
  ]
}
```



# 上传下载流程



# IO 实现细节

```go
var onlineDisks []StorageAPI
// ...
writers := make([]io.Writer, len(onlineDisks))
 for i, disk := range onlineDisks {
     writers[i] = newBitrotWriter(
         disk,minioMetaTmpBucket,tempErasureObj,
         erasure.ShardFileSize(curPartSize),
         DefaultBitrotAlgorithm,
         erasure.ShardSize())
 }
 // ...
n, erasureErr := erasure.Encode(
    ctx, curPartReader, writers, buffer, erasure.dataBlocks+1)
 // ...
```

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

# 巧用 channel

```go
readTriggerCh := make(chan bool, len(p.readers))
for i := 0; i < p.dataBlocks; i++ {
    readTriggerCh <- true
}
for readTrigger := range readTriggerCh {
    canDecode := p.canDecode(newBuf)
    if canDecode {
        break
    }
    if !readTrigger {
        continue
    }
    wg.Add(1)
    go func(i int) {
        defer wg.Done()
        disk := p.readers[i]
        if disk == nil {
            readTriggerCh <- true
            return
        }
        _, err := disk.ReadAt(p.buf[i], p.offset)
        if err != nil {
            readTriggerCh <- true
            return
        }
        // ...
        readTriggerCh <- false
    }(readerIndex)
    readerIndex++
}
wg.Wait()
```

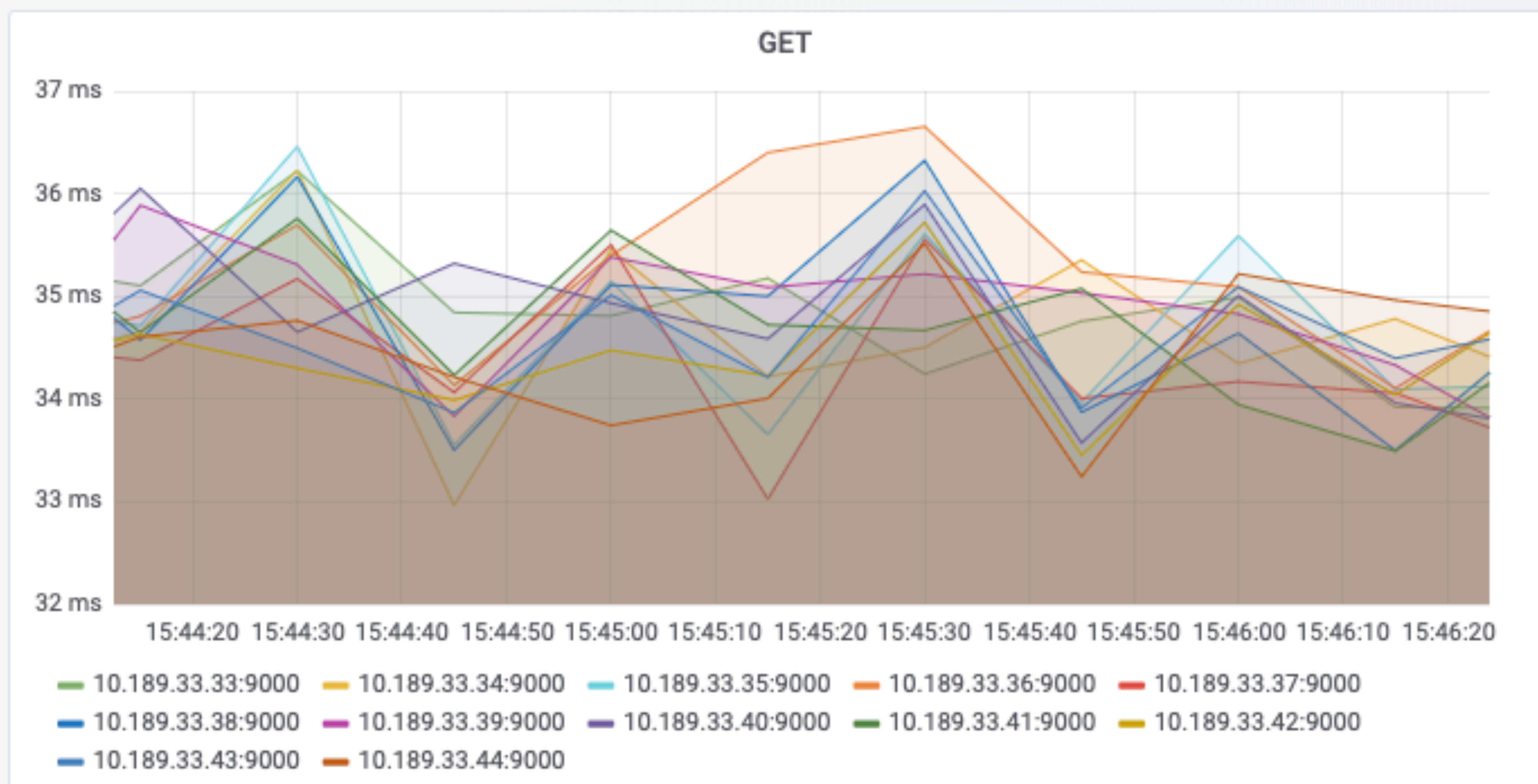
# 如何部署

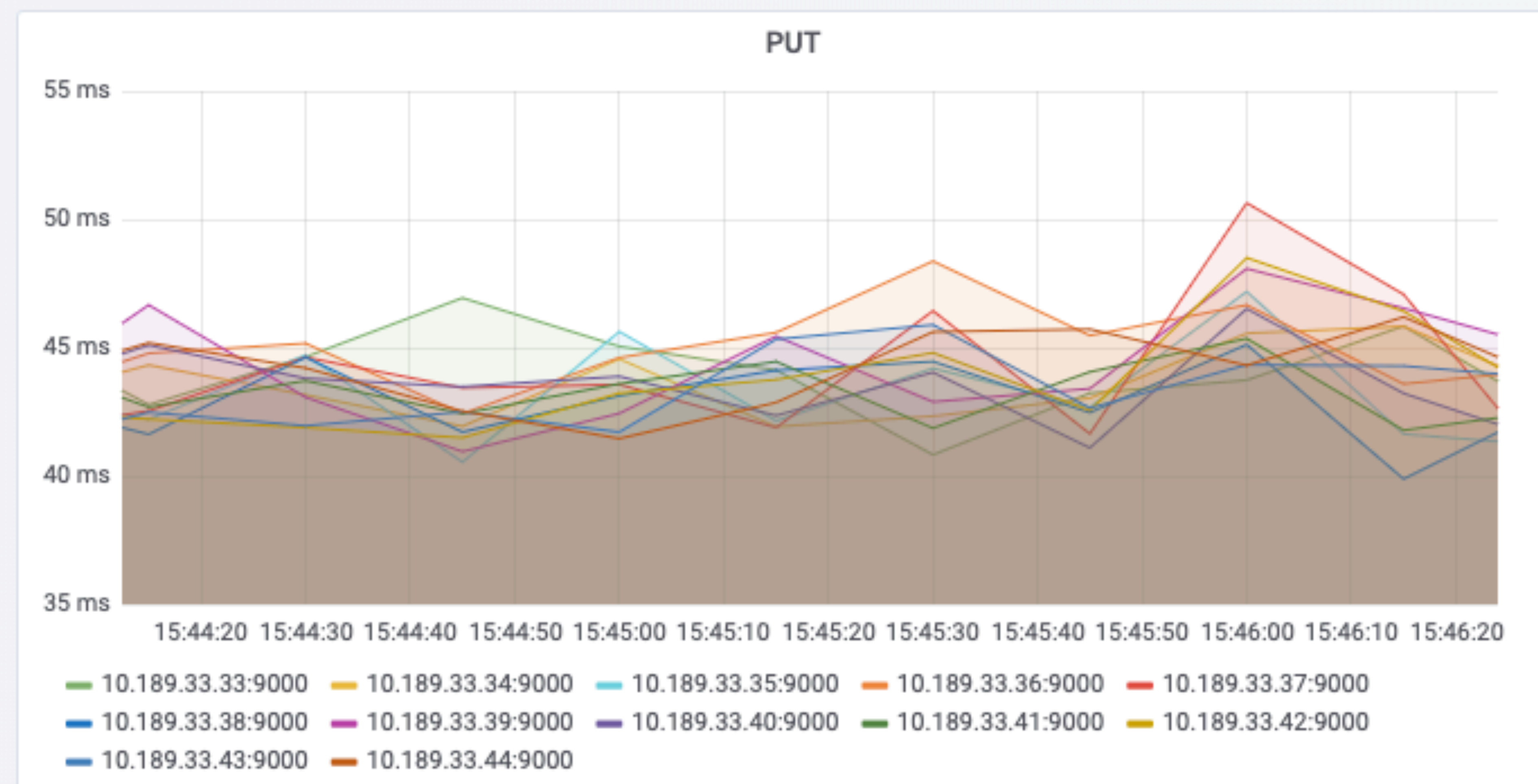
```
# 启动一个12个节点的 minio 集群，每个节点有12个drive
minio server http://10.3.1.{1…12}/data{1…12}
```

# 性能压测

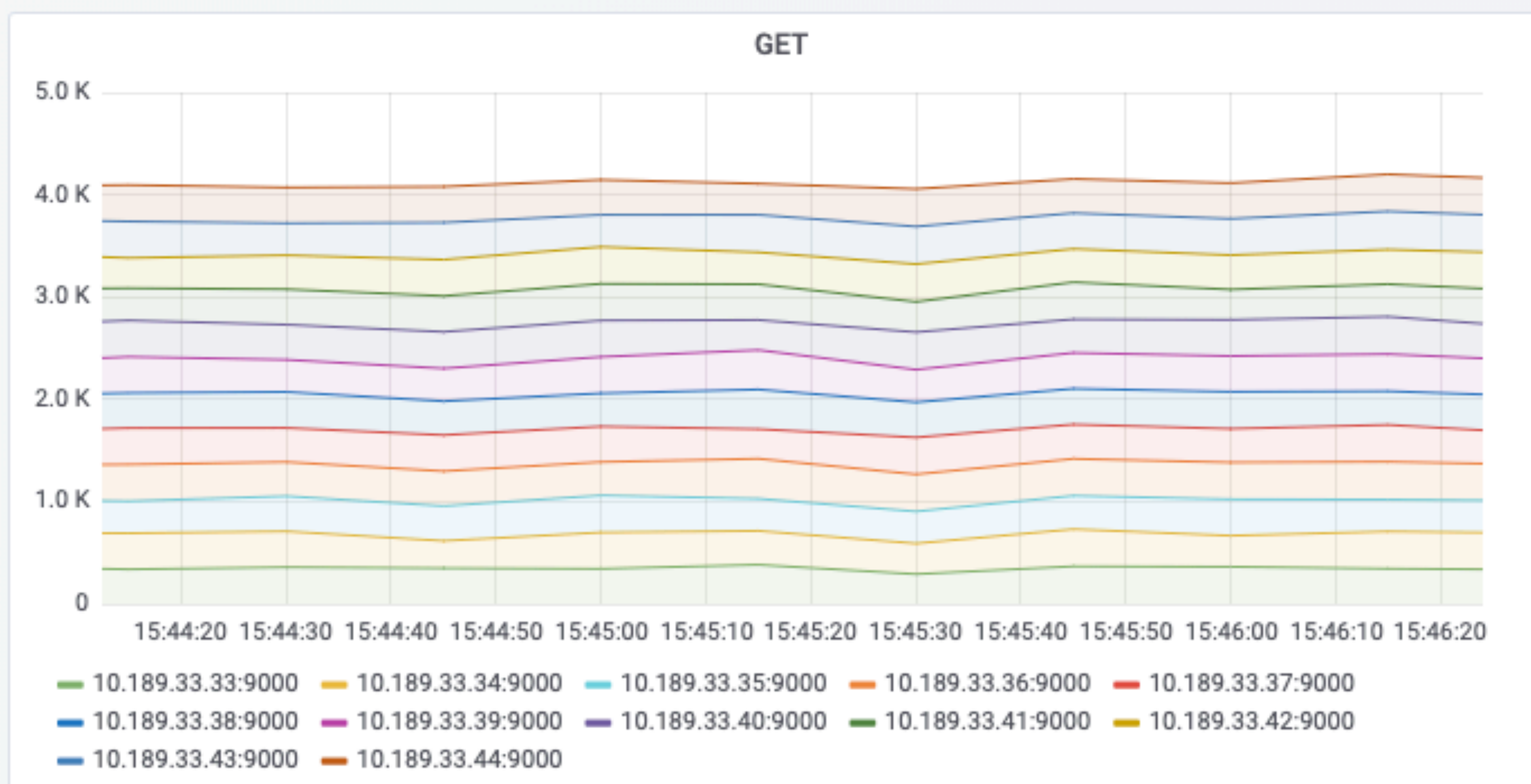
- CosBench
- 参考 Minio 官方压测方案
- 不同大小，不同并发，不同读写比例组合测试

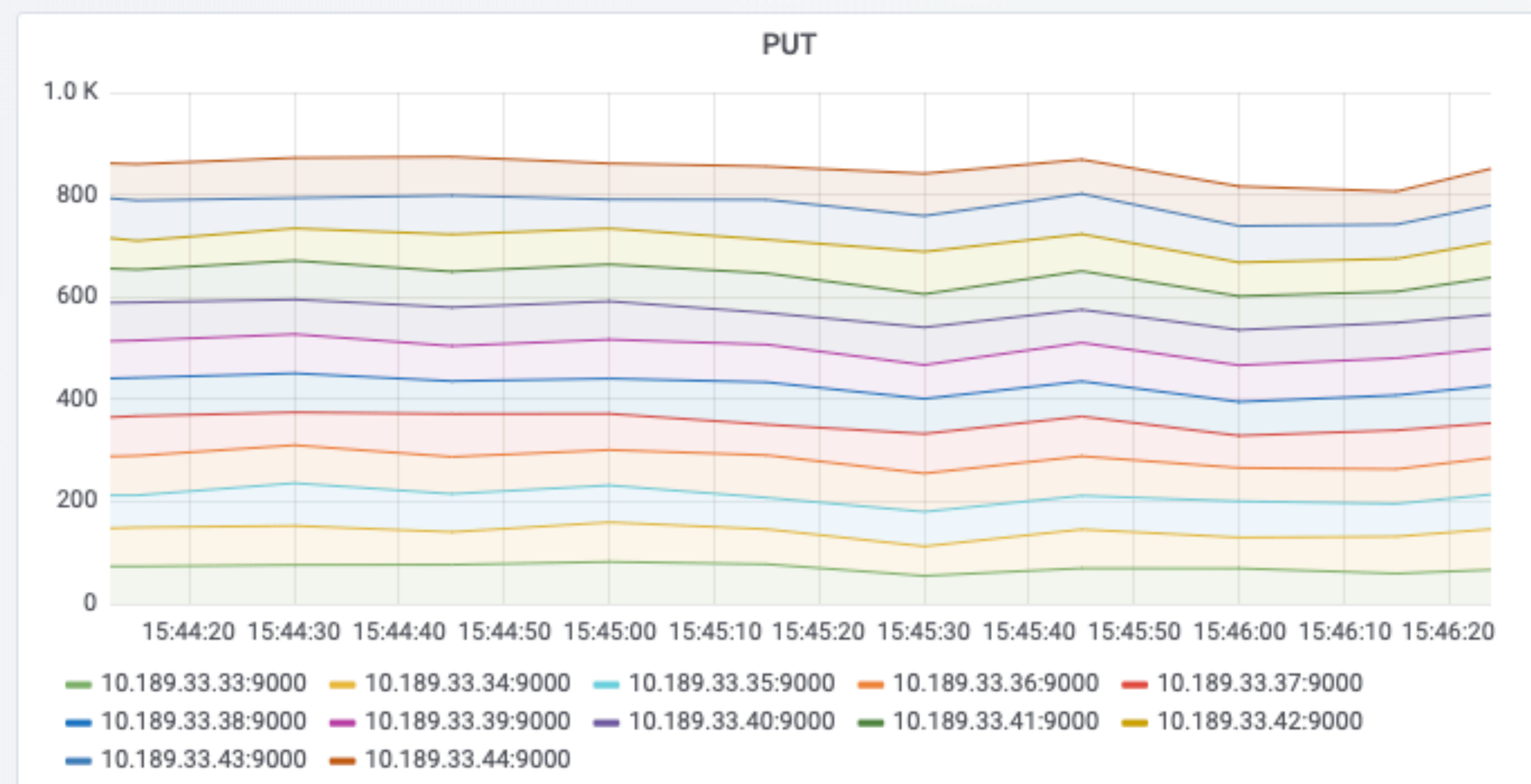
## HTTP Request Average Duration




## HTTP Requests Count per second

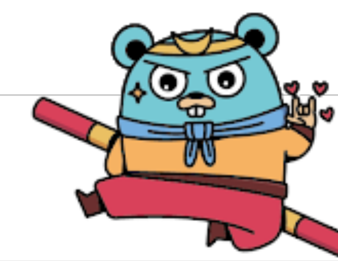



12 个节点， 每个节点 12 块盘，混合读写 64KB

# 多集群扩容





# MINIO Limitation

- 集群搭建后之后不允许扩容
- 大集群最大节点数为 32
    - Dsync 分布式锁

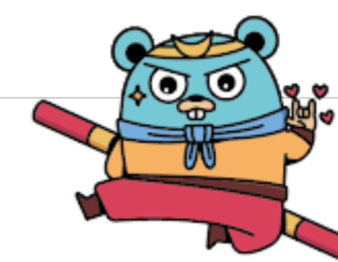
"We can certainly do single large namespace, but then taking one cluster down can potentially be a large downtime and a very complex deployment. This is exactly what we did with GlusterFS for 10yrs. This why most storage solutions never scale beyond few PBs.

*Minio's fundamental design is to avoid such practical issues in production and make managing clusters easier. Failures are common in real world scenarios and when it's a gigantic pool, it is very hard to know what is going on.*

*You can avoid these details in your application, if you are clever about it on how you lookup URLs.*"

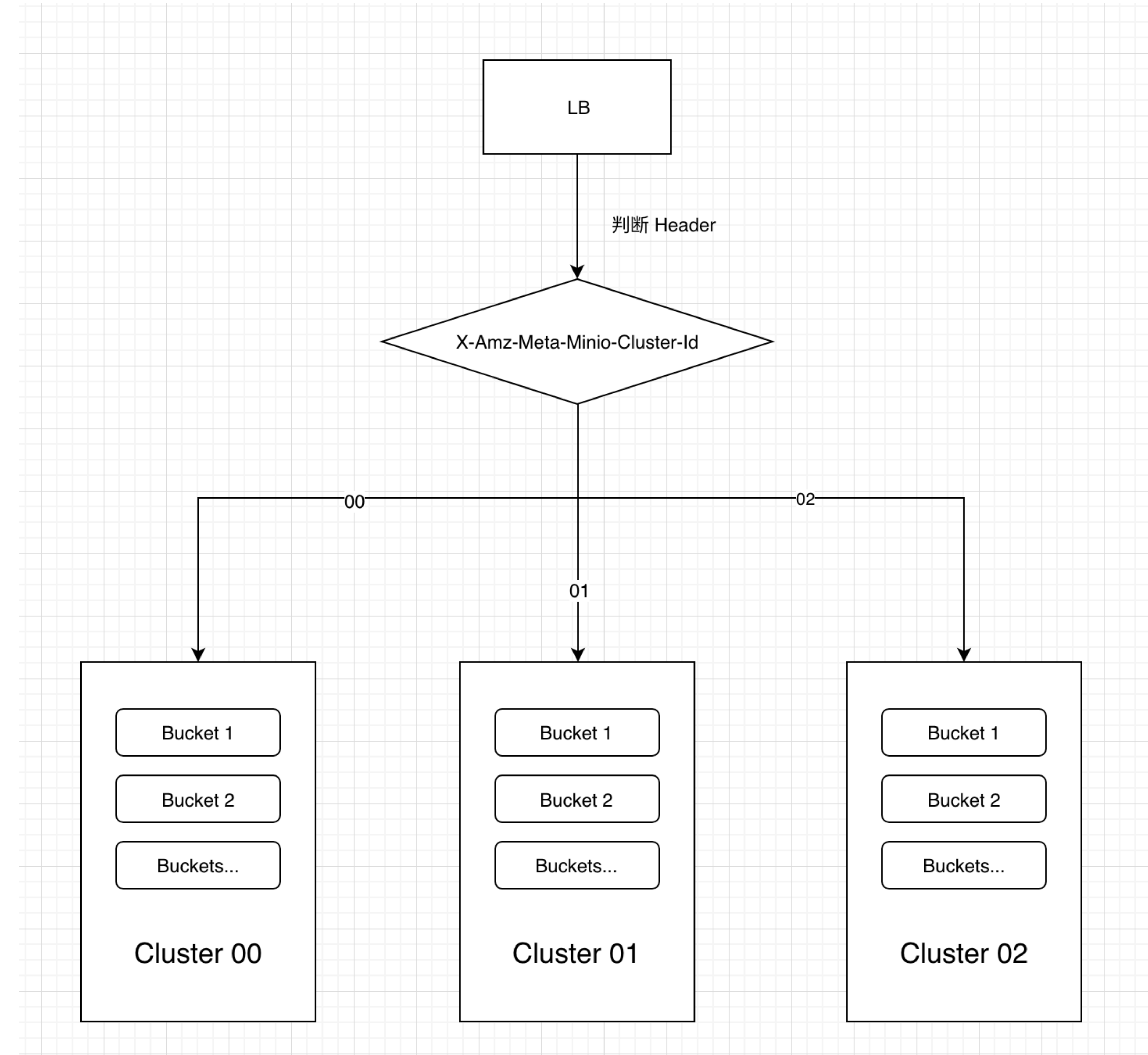
—— *Harshavardhana*





# 扩容方案

- 可同时向多个集群写入
- 集群容量到达阈值，从写集群中剔除，变成只读
- 读取的时候根据编码到文件名上的集群信息，从而路由到正确的集群

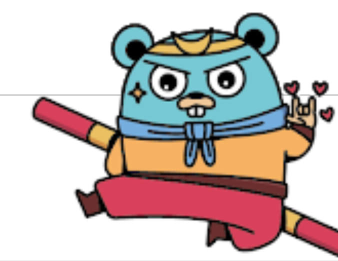


# 万事俱备，只欠东风

- 筹划数据迁移
- 准备接入服务
- 采购机器

# 生活并不是一帆风顺的





# 实际流量测试

- 100ms
- 200ms
- 500ms
- 报警!

# S3 下载响应时间

S3 GET latency

90 ms
80 ms
70 ms
60 ms
50 ms
40 ms
30 ms
20 ms
10 ms

4/19 00:00 | 4/19 12:00 | 4/20 00:00 | 4/20 12:00 | 4/21 00:00 | 4/21 12:00 | 4/22 00:00 | 4/22 12:00 | 4/23 00:00 | 4/23 12:00 | 4/24 00:00 | 4/24 12:00 | 4/25 00:00 | 4/25 12:00

2019-04-23 18:14:44
image - original - default - all: 38 ms
image - unoriginal - default - all: 26 ms

# S3 上传响应时间

S3 PUT latency

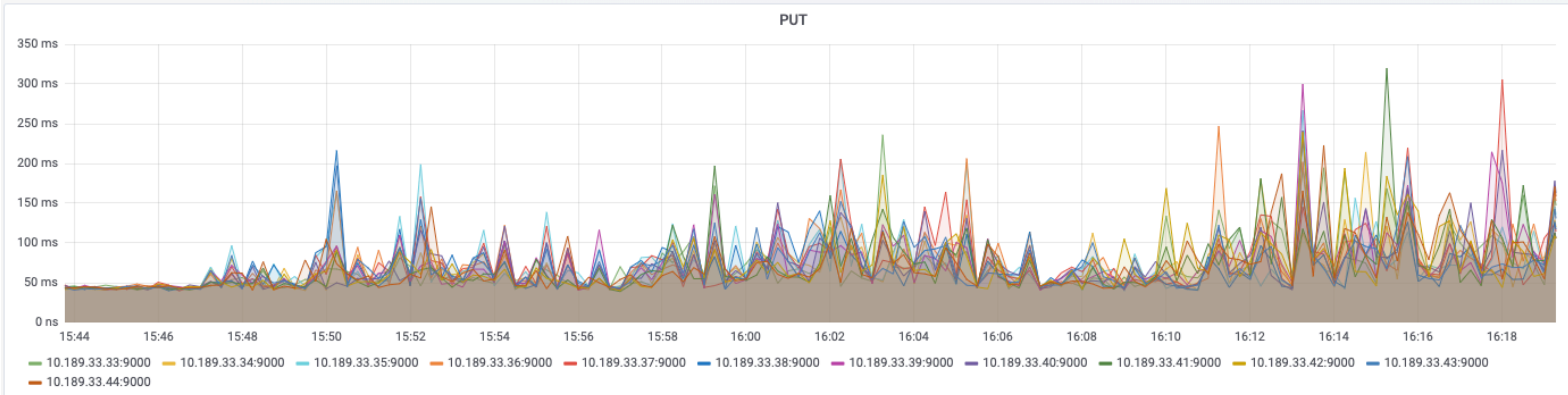
90 ms
80 ms
70 ms
60 ms
50 ms
40 ms
30 ms
20 ms
10 ms
0 ns

4/19 00:00 4/19 12:00 4/20 00:00 4/20 12:00 4/21 00:00 4/21 12:00 4/22 00:00 4/22 12:00 4/23 00:00 4/23 12:00 4/24 00:00 4/24 12:00 4/25 00:00 4/25 12:00

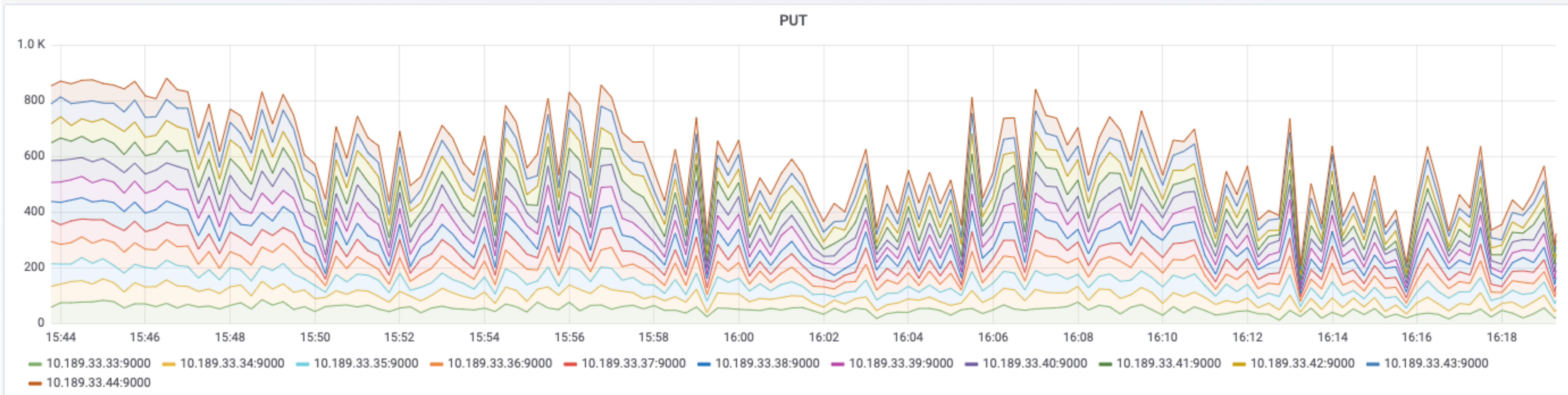
2019-04-23 19:34:44
image - original - default - all: 62 ms
image - unoriginal - default - all: 38 ms

# Minio 上传性能

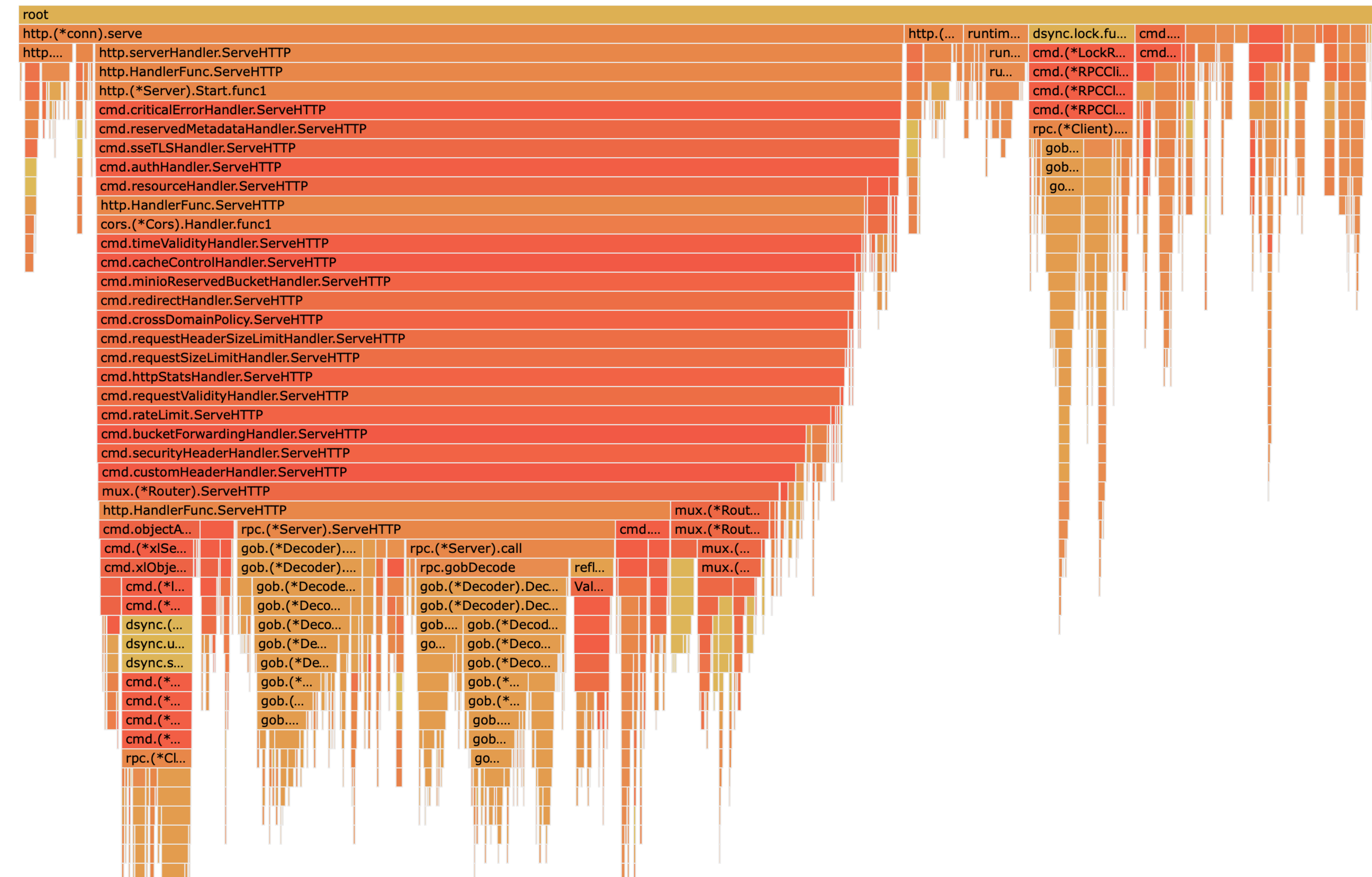
HTTP Request Average Duration




HTTP Requests Count per second



# MINIO Profile




```
# 开始采集整个集群指定 profile 类型的数据，类型可选 'cpu', 'mem', 'block', 'mutex', ‘trace'
mc admin profile start -type cpu
# 结束采集，自动下载采集结果文件
mc admin profile stop
# 使用 google pprof 工具进行可视化展示
pprof -http=0.0.0.0:2222 ./profiling-10.189.33.60\:9000.pprof
```

# 杀手锏

**PUT-getLock**

3.5 ms
3.0 ms
2.5 ms
2.0 ms
1.5 ms
1.0 ms

22:00 00:00 02:00 04:00 06:00 08:00 10:00 12:00 14:00

10.189.33.45:9000 10.189.33.46:9000 10.189.33.47:9000 10.189.33.48:9000 10.189.33.49:9000
10.189.33.50:9000 10.189.33.51:9000 10.189.33.52:9000 10.189.33.53:9000 10.189.33.54:9000
10.189.33.55:9000 10.189.33.56:9000

**posix.StatFile**

80 ms
60 ms
40 ms
20 ms
0 ns

22:00 00:00 02:00 04:00 06:00 08:00 10:00 12:00 14:00

10.189.33.45:9000 10.189.33.46:9000 10.189.33.47:9000 10.189.33.48:9000 10.189.33.49:9000
10.189.33.50:9000 10.189.33.51:9000 10.189.33.52:9000 10.189.33.53:9000 10.189.33.54:9000
10.189.33.55:9000 10.189.33.56:9000

**posix.CreateFile**

35 ms
30 ms
25 ms
20 ms
15 ms
10 ms
5 ms
0 ns

22:00 00:00 02:00 04:00 06:00 08:00 10:00 12:00 14:00

10.189.33.45:9000 10.189.33.46:9000 10.189.33.47:9000 10.189.33.48:9000 10.189.33.49:9000
10.189.33.50:9000 10.189.33.51:9000 10.189.33.52:9000 10.189.33.53:9000 10.189.33.54:9000
10.189.33.55:9000 10.189.33.56:9000

**posix.RenameFile**

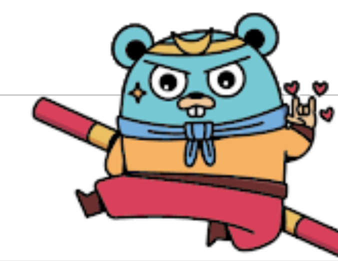
25 ms
20 ms
15 ms
10 ms
5 ms
0 ns

22:00 00:00 02:00 04:00 06:00 08:00 10:00 12:00 14:00

10.189.33.45:9000 10.189.33.46:9000 10.189.33.47:9000 10.189.33.48:9000 10.189.33.49:9000
10.189.33.50:9000 10.189.33.51:9000 10.189.33.52:9000 10.189.33.53:9000 10.189.33.54:9000
10.189.33.55:9000 10.189.33.56:9000

# 全面了解IO





# 全面了解 IO

- IO 流程
- Buffered IO vs Direct IO
- Page Cache
- IO Scheduler
- 磁盘性能
- 参数调优

# The Linux Storage Stack Diagram

version 4.10, 2017-03-10
outlines the Linux storage stack as of Kernel version 4.10

Fibre Channel over Ethernet
iSCSI
Fibre Channel
FireWire
USB
Virtual Host

LIO
tcm_fc
iscsi_target_mod
tcm_qla2xxx
sbp_target
tcm_usb_gadget
tcm_vhost
target_core_mod
target_core_file
target_core_iblock
target_core_pscsi
target_core_user

Applications (processes)
mmap
(anonymous pages)
malloc
vfs_writev, vfs_readv, ...
read(2)
write(2)
open(2)
stat(2)
chmod(2)
...

Direct I/O
(O_DIRECT)

VFS
Block-based FS
ext2 ext3 ext4 xfs
btrfs ifs iso9660
gfs ocfs ...
Network FS
NFS coda
smbfs ...
ceph
Pseudo FS
proc sysfs
pipefs futexfs
usbfs ...
Special purpose FS
tmpfs ramfs
devtmpfs
Stackable FS
ecryptfs overlayfs unionfs
FUSE
userspace (e.g. sshfs)
network

Page cache

stackable
(optional)
Devices on top of "normal" block devices
drbd LVM
device mapper
dm-crypt dm-mirror
dm-cache dm-thin
dm-raid dm-delay
mdraid
bcache
...

BIOs (block I/Os)
BIOs (block I/Os)
userspace

struct bio
- sector on disk
- sector cnt
- bio_vec cnt
- bio_vec index
- bio_vec list

BIOs
BIOs
BIOs

Block Layer
I/O scheduler
Maps BIOs to requests
noop
cfq
deadline
Hardware dispatch queue
blkmq
multi queue
Software queues
Hardware dispatch queues

hooked in device drivers
(they hook in like stacked devices do)

Request based drivers
Request based drivers
BIO based drivers

Request-based device mapper targets
dm-multipath

sysfs
(transport attributes)

Transport classes
scsi_transport_fc
scsi_transport_sas
scsi_transport_...

SCSI mid layer
scsi-mq
SCSI upper level drivers
/dev/sda /dev/sd* ...
/dev/sr* /dev/st*

/dev/zram* /dev/rbd* /dev/mmcblk*p* /dev/nullb* /dev/vd* /dev/rssd* /dev/skd*
/dev/ubiblock* /dev/nbd* /dev/loop* /dev/nvme*n* /dev/rsxx*
zram ubi rbd nbd mmc loop null_blk virtio_blk mtip32xx nvme skd rsxx
memory
network

SCSI low level drivers
libata megaraid_sas qla2xxx pm8001 iscsi_tcp virtio_scsi ufs ...
ahci ata_piix ...
aacraid lpfc mpt3sas vmw_pvscsi
network

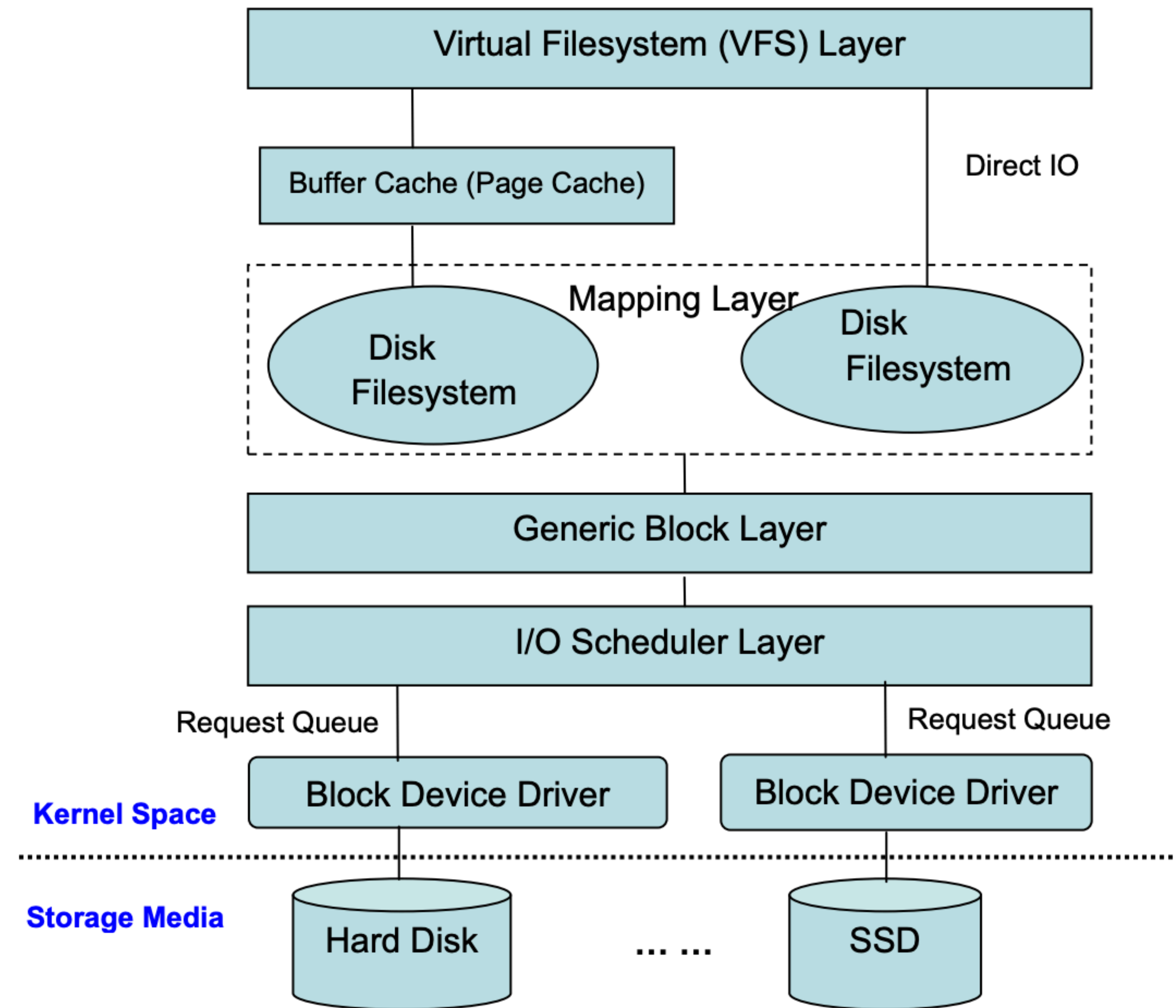
Physical devices
HDD SSD DVD drive LSI RAID Qlogic HBA PMC-Sierra HBA
Adaptec RAID Emulex HBA LSI 12Gbs SAS HBA
para-virtualized SCSI
virtio_pci
VMware's para-virtualized SCSI
mobile device flash memory
SD-/MMC-Card
Micron PCIe card
nvme device
stec device
IBM flash adapter

The Linux Storage Stack Diagram
http://www.thomas-krenn.com/en/wiki/Linux_Storage_Stack_Diagram
Created by Werner Fischer and Georg Schönberger
License: CC-BY-SA 3.0, see http://creativecommons.org/licenses/by-sa/3.0/

THOMAS KRENN®

ina 2019 · 北京

# Block IO 流程

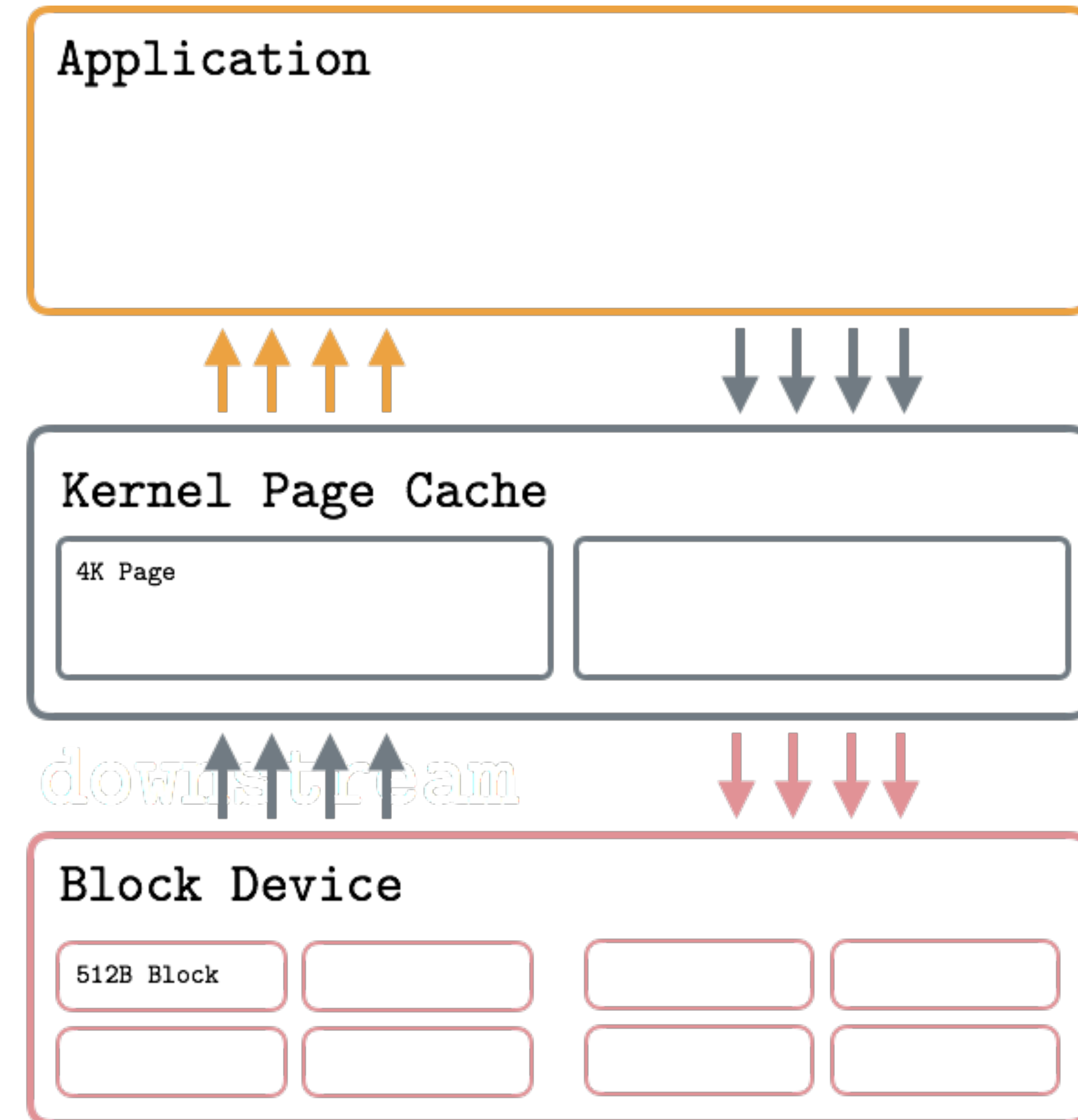



http://www.ilinuxkernel.com/files/Linux.Generic.Block.Layer.pdf

# Buffered IO vs Direct IO

- Buffered IO
  - read (LRU)
  - flush or writeback

- Direct IO
  - O_DIRECT
  - Block Alignment
  - WAL(write ahead log)




https://medium.com/databasss/on-disk-io-part-1-flavours-of-io-8e1ace1de017

# Page Cache

- **Temporal Locality**, recently accessed pages will be accessed again at some point in nearest future.
- **Spatial Locality**, the elements physically located nearby have a good chance of being located close to each other.

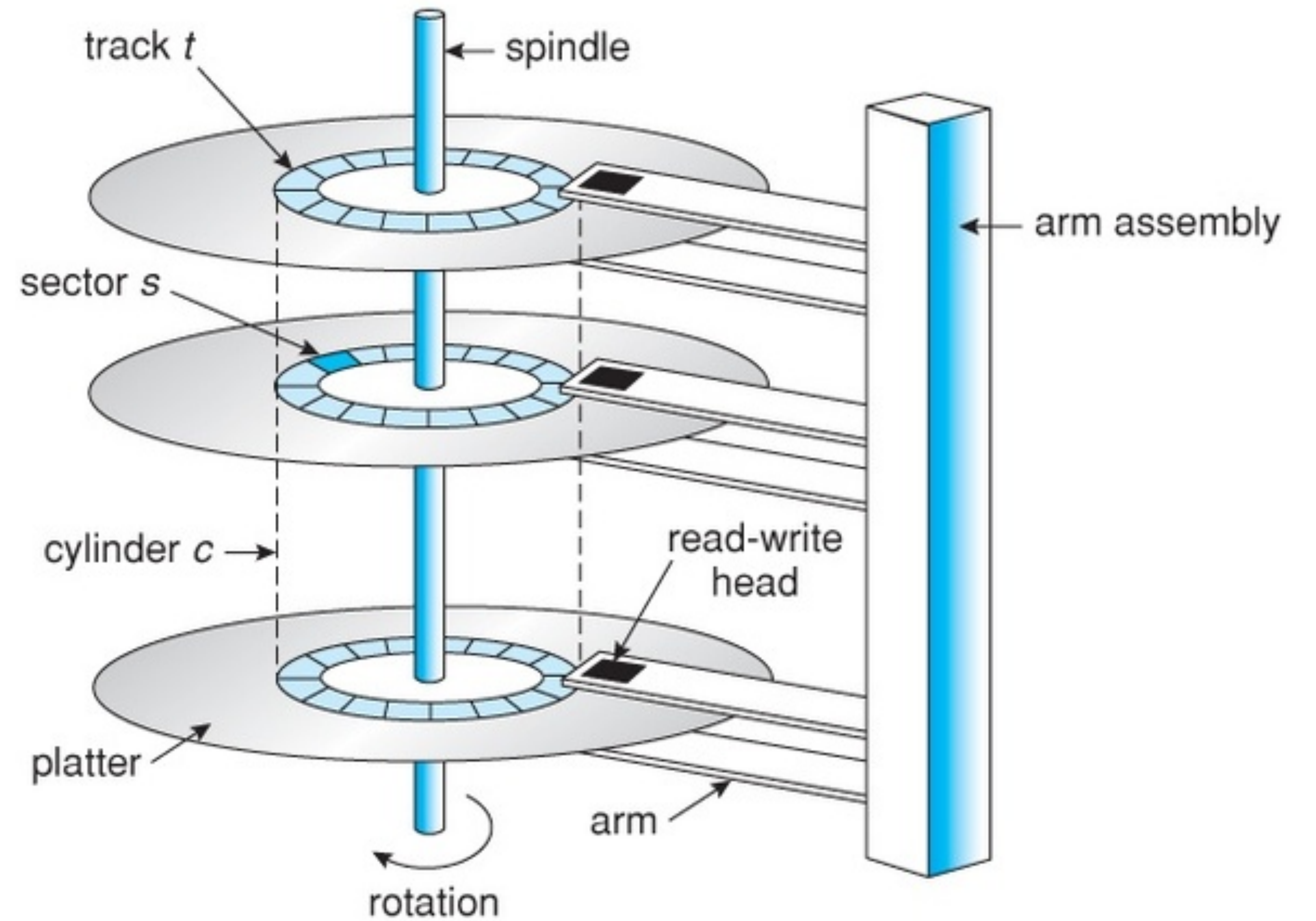
# IO Scheduler

- IO Queue
- elevator: Deadline, CFQ, Noop
- how to

```
# 查看 sda 的 scheduler
cat /sys/block/sda/queue/scheduler
# 设置 sdb 的 scheduler 为 noop 算法
echo noop > /sys/block/sdb/queue/scheduler
```

# 磁盘

- 顺序读写快
- 随机读写慢




https://qph.fs.quoracdn.net/main-qimg-f0bf80ac2007d1f39e8851a5e25b870c

# 回顾两个问题

- 之前的压测结果中，为什么刚开始的一段时间响应时间特别低?
- 为什么之后响应时间变高？并且波动幅度很大

# 参数调优

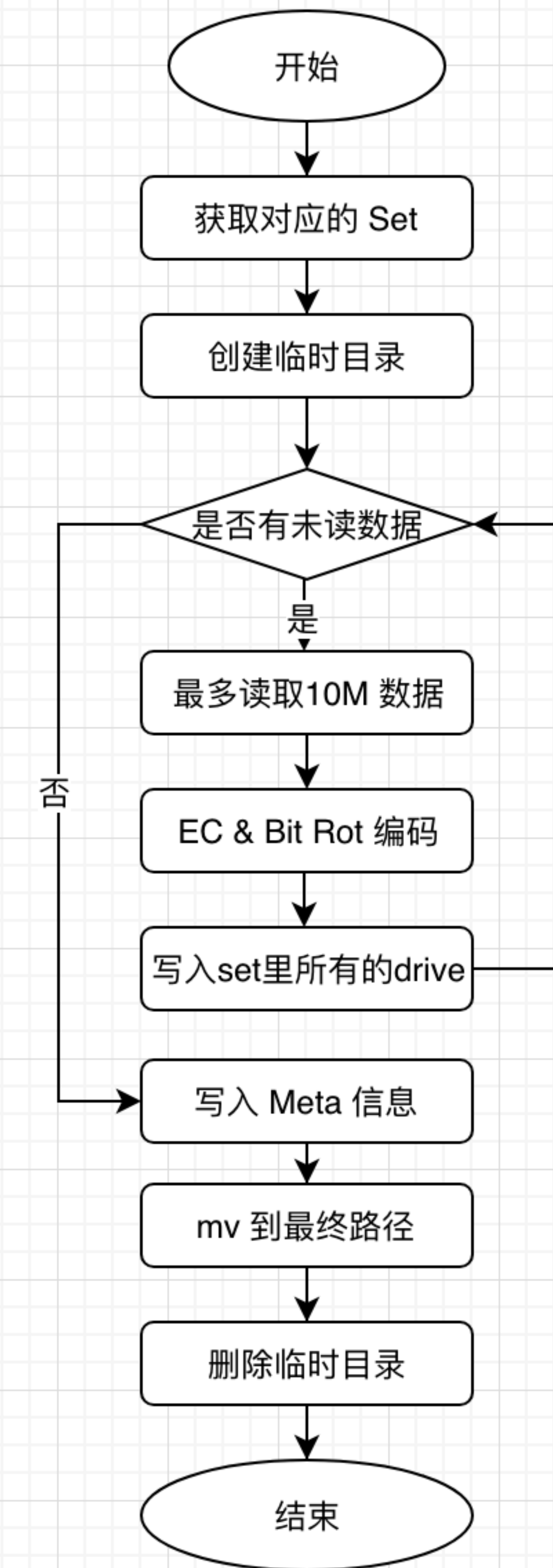
- Virtual Memory
  - /proc/sys/vm/dirty_ratio
  - /proc/sys/vm/dirty_background_ratio
  - /proc/sys/vm/dirty_writeback_centisecs
  - /proc/sys/vm/dirty_expire_centisecs

- IO
  - cat /sys/block/sda/queue/scheduler
  - /sys/block/sda/queue/read_ahead_kb
  - nr_requests
  - max_sectors_kb

# 深度优化 Minio

# 再回顾 MINIO 写流程

- 随机写
- 写次数放大



# 优化方案 — 小文件合并大文件

- fallocate 预分配空间
- 顺序写入
- 小文件的索引（volume id, offset, size）存 leveldb

# FIO 测试效果

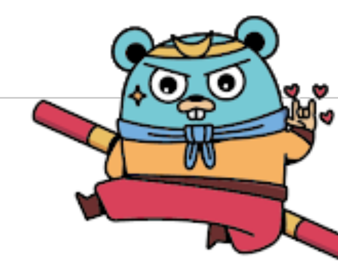
```
# 测试随机写性能
fio -name sequentialwrite -rw=write -bs=10k -runtime=100 -iodepth 128   \

    -ioengine libaio   -direct=1 --size=200G --randrepeat=1 \

    --directory=/data10/fio --nrfiles=1 --openfiles=1  -group_reporting

# 测试顺序写性能
fio  -name randomwrite -rw=randwrite -bs=10k -runtime=100 -iodepth 128  \

     -ioengine libaio   -direct=1 --size=200G --randrepeat=1 \

     --directory=/data10/fio --nrfiles=1 --openfiles=1  --group_reporting
```

# 再谈 Golang IO

- 不同 IO 介质之间数据互传
  - req.Body
  - *os.File
  - io.Copy

```go
func (v *Volume) WriteAll(key string, size int64, r io.Reader) error {
    // ...
    v.lock.Lock()
    defer v.lock.Unlock()
    _, err := io.Copy(v.file, r)
    if err != nil {
        return err
    }
    // ...
    return nil
}
```





# 性能压测

## HTTP Request Average Duration

PUT

150 ms
100 ms
50 ms
0 ns

13:30 13:40 13:50 14:00 14:10 14:20 14:30 14:40 14:50 15:00 15:10 15:20 15:30 15:40 15:50 16:00 16:10 16:20 16:30 16:40 16:50 17:00 17:10 17:20 17:30

10.189.33.57:9000 10.189.33.58:9000 10.189.33.59:9000 10.189.33.60:9000 10.189.33.61:9000 10.189.33.62:9000 10.189.33.63:9000 10.189.33.64:9000 10.189.33.65:9000 10.189.33.66:9000 10.189.33.67:9000
10.189.33.68:9000

## HTTP Requests Count per second

PUT

6.0 K
5.0 K
4.0 K
3.0 K
2.0 K
1.0 K
0

13:30 13:40 13:50 14:00 14:10 14:20 14:30 14:40 14:50 15:00 15:10 15:20 15:30 15:40 15:50 16:00 16:10 16:20 16:30 16:40 16:50 17:00 17:10 17:20 17:30

10.189.33.57:9000 10.189.33.58:9000 10.189.33.59:9000 10.189.33.60:9000 10.189.33.61:9000 10.189.33.62:9000 10.189.33.63:9000 10.189.33.64:9000 10.189.33.65:9000 10.189.33.66:9000 10.189.33.67:9000
10.189.33.68:9000

12 个节点， 每个节点 12 块盘，写 64KB

# 测试读性能

分析业务场景

- 20% 数据为刚写入，大部分在 Page Cache 中
- 80% 数据随机分布在硬盘上

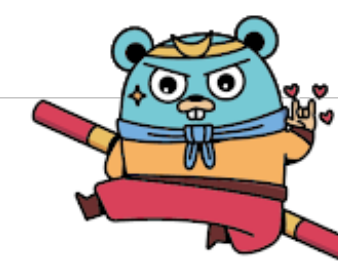
制定测试方案

- 利用 MINIO 消息机制，将所上传的对象的列表写入 kafka
- 写入一段时间之后，再根据 kafka 中的文件列表进行读压测

你猜性能如何?

性能高的离谱！



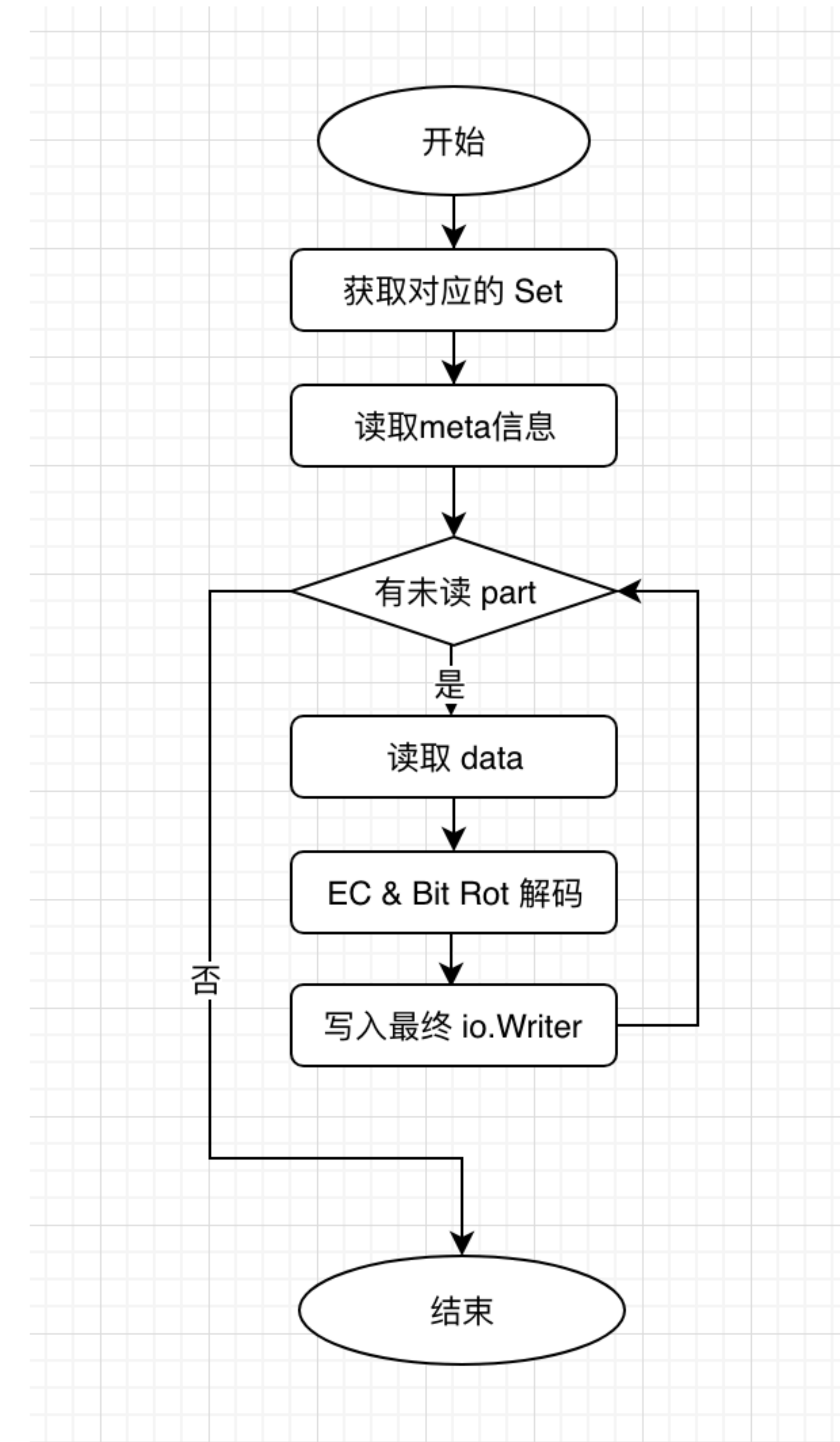


# 事实本该如此

- 因为数据顺序写入，其文件列表发到kafka的时候也是顺序的，所以命中了顺序读的策略
- 加入随机因子之后，读性能急剧下降，符合预期

# 再回顾 MINIO 读流程

- 从 N 个节点上读取 meta 信息
- 至少从 N/2 个节点读取 data
- 每一次读取的时间包括两个部分
  - 从 levelDB 读取索引
  - 从大文件中读取对应的数据
- 并发读的最终时间取决于最长的一个



# 优化方案

- 将 Meta 信息压缩后直接存入 levelDB, 减少一组 IO
- 将 levelDB 存在性能更高的 SSD
- 降低单个 Set 的 Drive 数量

# 压测结果

HTTP Request Average Duration

GET

75 ms
70 ms
65 ms
60 ms
55 ms
50 ms
45 ms
40 ms

09:00 09:30 10:00 10:30 11:00

10.189.33.57:9000 10.189.33.58:9000 10.189.33.59:9000 10.189.33.60:9000 10.189.33.61:9000
10.189.33.62:9000 10.189.33.63:9000 10.189.33.64:9000 10.189.33.65:9000 10.189.33.66:9000
10.189.33.67:9000 10.189.33.68:9000

PUT

300 ms
250 ms
200 ms
150 ms
100 ms
50 ms
0 ns

09:00 09:30 10:00 10:30 11:00

10.189.33.57:9000 10.189.33.58:9000 10.189.33.59:9000 10.189.33.60:9000 10.189.33.61:9000
10.189.33.62:9000 10.189.33.63:9000 10.189.33.64:9000 10.189.33.65:9000 10.189.33.66:9000
10.189.33.67:9000 10.189.33.68:9000

HTTP Requests Count per second

GET

3.5 K
3.0 K
2.5 K
2.0 K
1.5 K
1.0 K
500
0

09:00 09:30 10:00 10:30 11:00

10.189.33.57:9000 10.189.33.58:9000 10.189.33.59:9000 10.189.33.60:9000 10.189.33.61:9000
10.189.33.62:9000 10.189.33.63:9000 10.189.33.64:9000 10.189.33.65:9000 10.189.33.66:9000
10.189.33.67:9000 10.189.33.68:9000

PUT

1.5 K
1.0 K
500
0

09:00 09:30 10:00 10:30 11:00

10.189.33.57:9000 10.189.33.58:9000 10.189.33.59:9000 10.189.33.60:9000 10.189.33.61:9000
10.189.33.62:9000 10.189.33.63:9000 10.189.33.64:9000 10.189.33.65:9000 10.189.33.66:9000
10.189.33.67:9000 10.189.33.68:9000

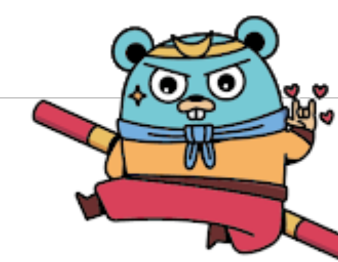
12 个节点， 每个节点 12 块盘，混合读写 64KB

# 路漫漫其修远兮

- 如何解决尖峰问题
- 长时间压测，确保集群容量很大以后仍然可以保持不错的性能
- levelDB 是否适合这个场景?

# 总结





# 总结

- 再回顾 MINIO
- 磨刀不误砍柴工

# 谢谢