Go 在百亿级分布式文件系统的实践

×

徐桑迪

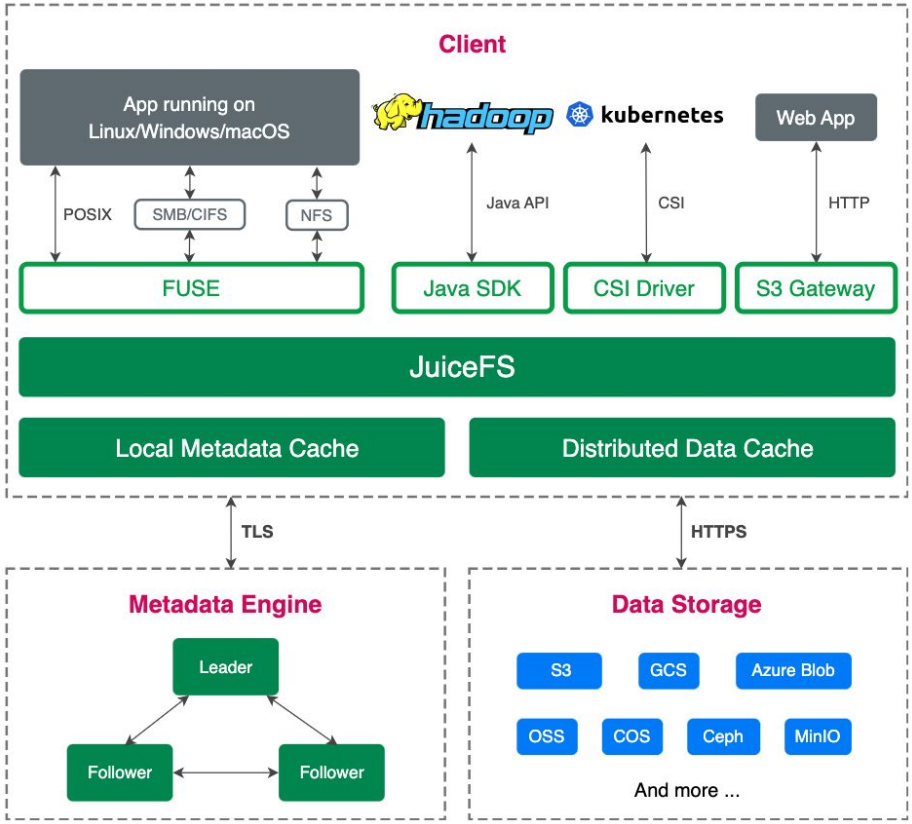
Juicedata 核心系统工程师

目录

第一部分

JuiceFS 简介

JuiceFS 简介



- 为云环境设计的分布式文件系统
- 兼容 POSIX、HDFS 和 S3 协议
- 支持回收站、目录配额、克隆
- 单命名空间支持百亿级文件数
- 高性能、高可靠、高扩展性

第二部分

为什么选择 Go

为什么选择 Go

- 快速开发
  - 多线(协)程:go 关键字, channel 特性
  - 性能分析:go tool pprof 等
  - 故障分析:详细的 stack trace
  - 编译速度快
  - 内存管理:自带GC

为什么选择 Go

- 性能优秀：编译型语言
- 可移植性好：静态编译（第三方库容易有动态依赖）

```
$ apt install musl-tools
$ cat Makefile
ifdef STATIC
     LDFLAGS += -linkmode external -extldflags '-static'
     CC = /usr/bin/musl-gcc
     export CC
endif
...

$ ls -l juicefs*
-rwxr-xr-x 1 root root 84725616 Nov 15 11:37 juicefs
-rwxr-xr-x 1 root root 84728280 Nov 15 11:38 juicefs-static
$ ldd juicefs-static
     not a dynamic executable
```

为什么选择 Go

- 支持多语言 SDK

```
go build -buildmode=c-shared -ldflags="$(LDFLAGS)" -o libjfs.so .
```

  - Java：用 JNI 或 JNR 加载共享库
  - C/C++：`dlopen` 函数加载，或者直接混合编译
  - Python：`ctypes` 模块加载共享库

第三部分

基础内存优化

基础内存优化

- 用定长 struct 代替 string

```
// old
var m map[string]int
k := fmt.Sprintf("%d_%d_%d", id, index, size)

// new
type Key struct {
     id, index, size uint32
}
var m map[Key]int
k := Key{id, index, size}
```

```
type stringStruct struct {
     str unsafe.Pointer
     len int
}
```

基础内存优化

- 使用 sync.Pool 循环使用内存

```
pool := sync.Pool{
     New: func() interface{} {
          buf := make([]byte, 1<<22)
          return &buf
     },
}

buf := pool.Get().(*[]byte)
// do some work
pool.Put(buf)
```

基础内存优化

- 维护引用计数，强化 sync.Pool

```
type Page struct {
     Data   []byte
     size   int
     refs   int32
     parent *Page
}

func (p *Page) Acquire() {
     atomic.AddInt32(&p.refs, 1)
}

func (p *Page) Release() {
     if atomic.AddInt32(&p.refs, -1) == 0 {
          pool.Put(&p.Data)
          ...
     }
}
```

基础内存优化

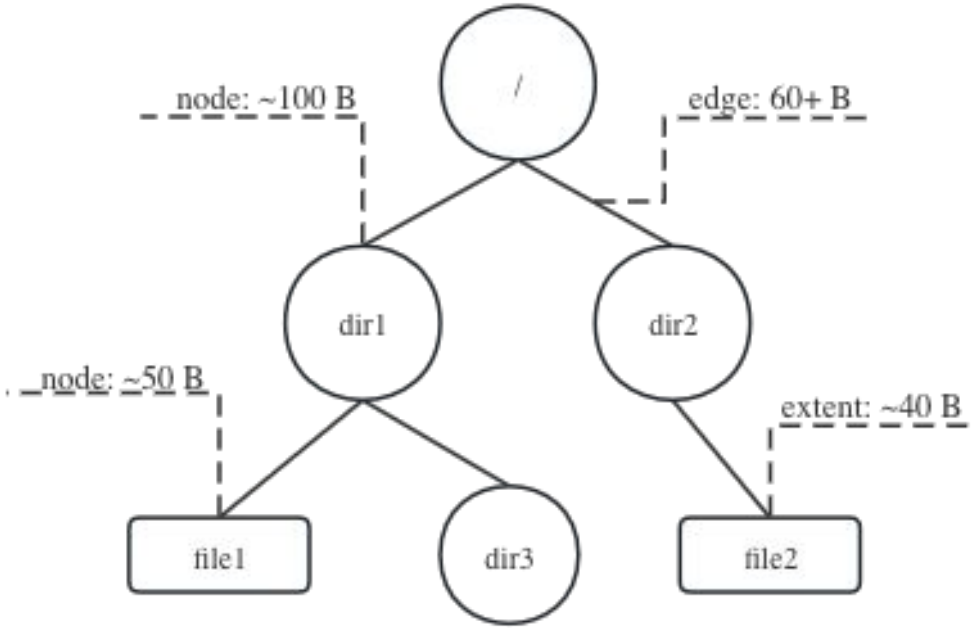
- 维护引用计数，强化 sync.Pool

```
func (p *Page) Slice(off, len int) *Page {
     p.Acquire()
     np := NewPage(p.Data[off : off+len])
     np.parent = p
     return np
}



func (s *storage) asyncUpload (key string, p *Page) {
     p.Acquire()
     go func() {
          defer p.Release()
          errs <- s.Put(key, p)
     }()
}
```

第四部分

深度内存优化

深度内存优化



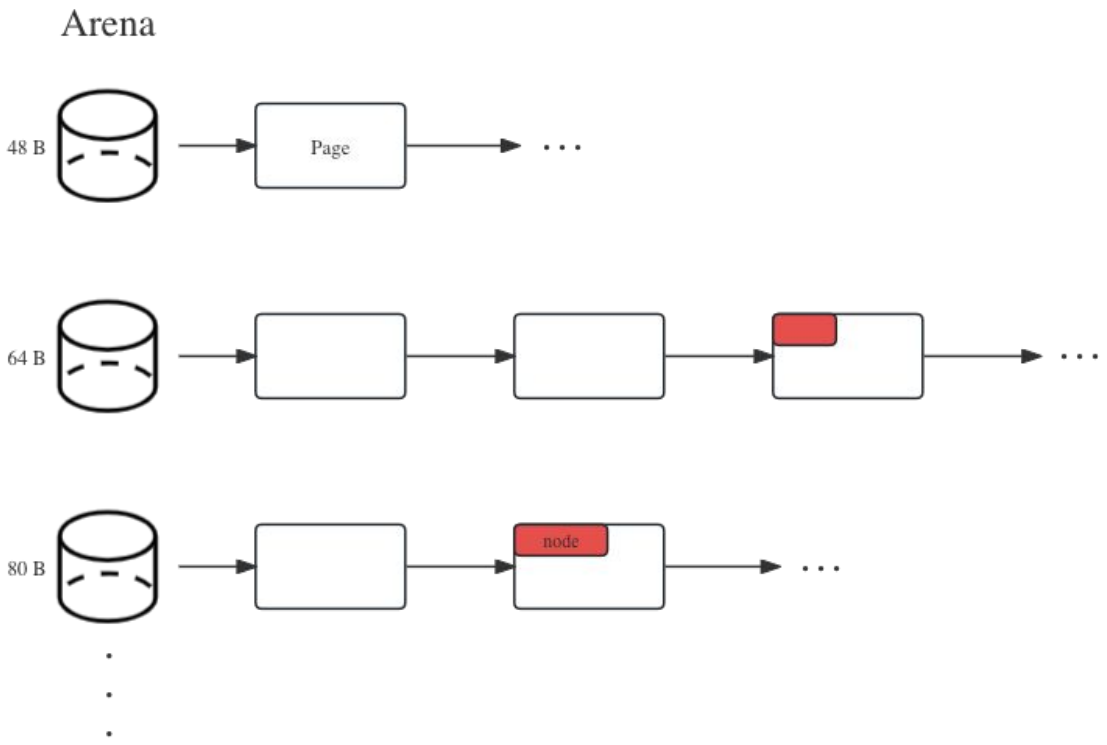
文件系统元数据服务进程：

- 占用近百 GiB 内存
- 缓存尽可能多的文件（十亿级）
- 高速处理元数据请求（百微秒）

深度内存优化

- 自主管理小对象的分配
  - GC 全局能看到的指针要少
  - GC 递归扫描的深度要小

自主管理小对象分配



自主管理小对象分配

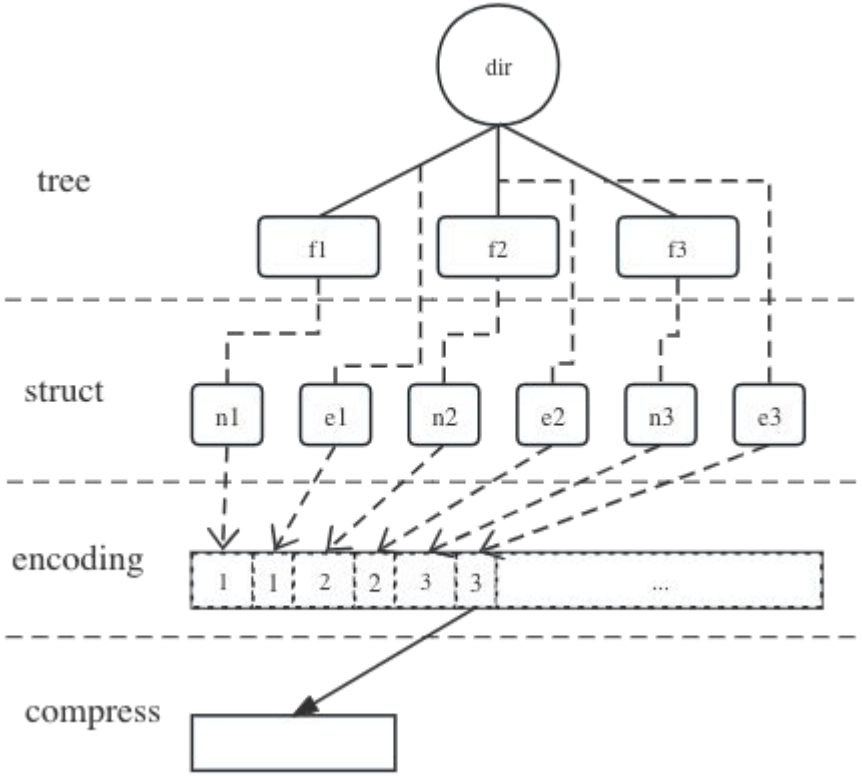
```
var slabs = make(map[uintptr][]byte)

p := pagePool.Get().(*[]byte) // 128 KiB
ptr := unsafe.Pointer(&(*p)[0])
arena.AddPage(uintptr(ptr))

// hold this page
slabs[uintptr(ptr)] = *p
```

```
func (a *arena) Alloc(size int) unsafe.Pointer {...}

size := nodeSizes[type]
n := (*node)(nodeArena.Alloc(size))


// var nodeMap map[uint32, uintptr]
nodeMap[n.id] = uintptr(unsafe.Pointer(n)))
```

深度内存优化

- 结构体打包序列化和压缩
  - 文件系统天然具有组织结构(目录树)
  - 访问请求通常具有局部性

结构体序列化和压缩

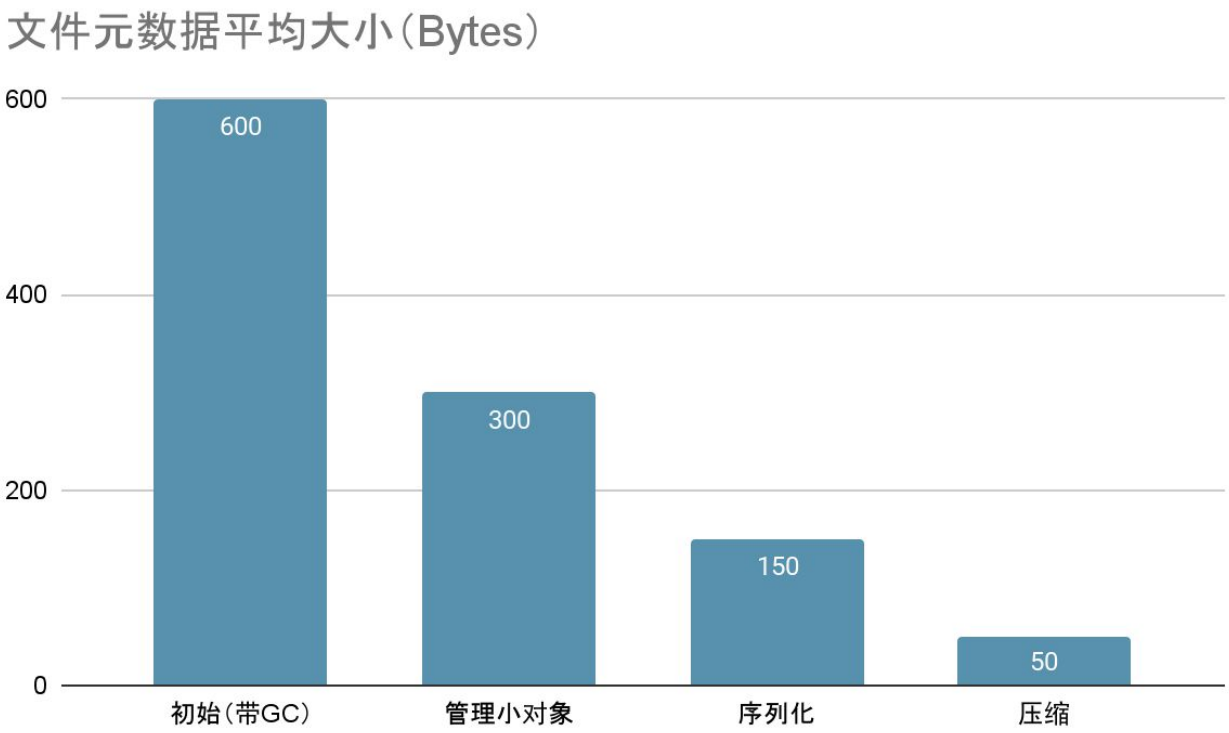

dir
tree
f1
f2
f3
struct
n1
e1
n2
e2
n3
e3
encoding
1
1
2
2
3
3
...
compress

深度内存优化

- 动态定义参数值的意义（类似 union）

```
type fnode struct {
     ...
     // extents []uint64
     extent uintptr
     num    uint32
}

// return list of the extent IDs
func (f *fnode) getExtents() []uint64 {
     if f.num == 0 {
          return nil
     } else if f.num == 1 { // extent is the real ID
          return unsafe.Slice((*uint64)(unsafe.Pointer(&f.extent)), 1)
     } else { // extent is the address of the ID array
          return unsafe.Slice((*uint64)(unsafe.Pointer(f.extent)), f.num)
     }
}
```

深度内存优化


文件元数据平均大小(Bytes)
600
400
200
0
600
300
150
50
初始(带GC)
管理小对象
序列化
压缩

优化结果

单个 MDS 进程：

- 30 GiB 内存管理约 3 亿个文件
- 数十万 ops，亚毫秒级延迟
- 多分区水平扩展

最大的文件系统（生产环境）：

- 十几个元数据节点，约 80 个分区
- 200+ 亿文件，约 10 PiB 容量

第五部分

内存快照持久化

内存快照持久化

- 通过进程 fork 实现内存快照
  - 多线程应用容易发生死锁
    - 核心数据结构由单线程无锁访问
  - Go runtime 也有可能导致死锁
    - 父进程在 fork 前做好检查和准备，减少子进程内存分配
    - 父进程监控子进程状态，超时则主动 kill

谢谢！

Website: https://juicefs.com

https://github.com/juicedata/juicefs