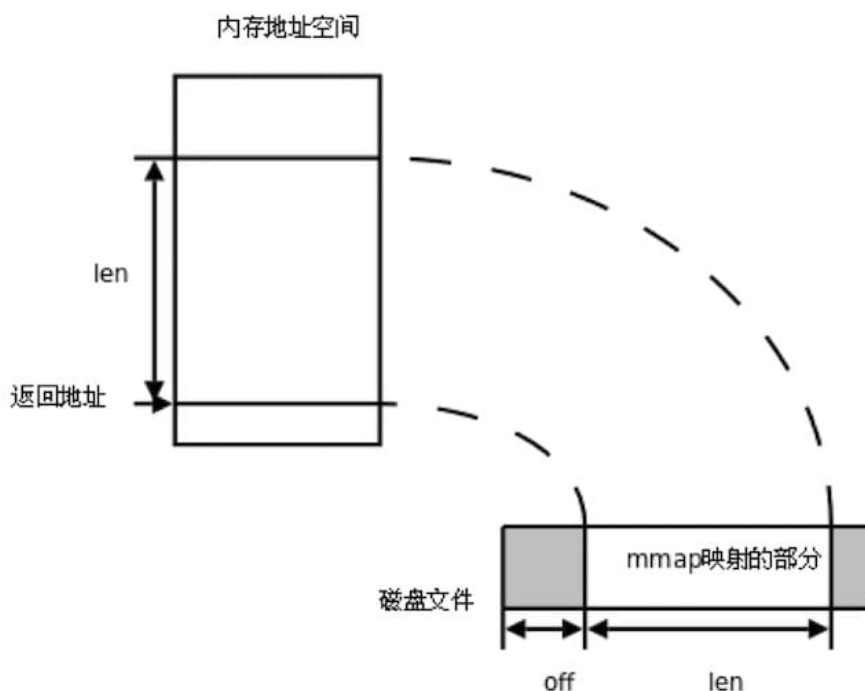


# Go Mmap 文件内存映射简明教程

## Go 简明教程系列文章链接:

- [Go 语言简明教程](#) (Aug 6, 2019)
- [Go Gin 简明教程](#) (Aug 7, 2019)
- [Go2 新特性简明教程](#) (Aug 15, 2019)
- [Go Protobuf 简明教程](#) (Jan 11, 2020)
- [Go RPC & TLS 鉴权简明教程](#) (Jan 13, 2020)
- [Go WebAssembly \(Wasm\) 简明教程](#) (Jan 23, 2020)
- [Go Test 单元测试简明教程](#) (Feb 10, 2020)
- [Go Mock \(gomock\)简明教程](#) (Feb 14, 2020)
- [Go Mmap 文件内存映射简明教程](#) (Apr 20, 2020)
- [Go Context 并发编程简明教程](#) (Apr 20, 2020)



## 1 mmap 简介

In computing, mmap is a POSIX-compliant Unix system call that maps files or devices into memory. It is a method of memory-mapped file I/O.

– [mmap - wikipedia.org](https://en.cppreference.com/w/cpp/string/basic/basic_string_view)

简单理解，mmap 是一种将文件/设备映射到内存的方法，实现文件的磁盘地址和进程虚拟地址空间中一段虚拟地址的一一映射关系。也就是说，可以在某个进程中通过操作这一段映射的内存，实现对文件的读写等操作。修改了这一段内存的内容，文件对应位置的内容也会同步修改，而读取这一段内存的内容，相当于读取文件对应位置的内容。

mmap 另一个非常重要的特性是：减少内存的拷贝次数。在 linux 系统中，文件的读写操作通常通过 read 和 write 这两个系统调用来实现，这个过程会产生频繁的内存拷贝。比如 read 函数就涉及了 2 次内存拷贝：

- 1. 操作系统读取磁盘文件到页缓存；
- 2. 从页缓存将数据拷贝到 read 传递的 buf 中(例如进程中创建的byte数组)。

mmap 只需要一次拷贝。即操作系统读取磁盘文件到页缓存，进程内部直接通过指针方式修改映射的内存。因此 mmap 特别适合读写频繁的场景，既减少了内存拷贝次数，提高效率，又简化了操作。KV数据库 [bbolt](#) 就使用了这个方法持久化数据。

## 2 标准库 mmap

Go 语言标准库 [golang.org/x/exp/mmap](#) 仅实现了 read 操作，后续能否支持 write 操作未知。使用场景非常有限。看一个简单的例子：

从第4个byte开始，读取 tmp.txt 2个byte的内容。

```
1 package main
2
3 import (
4     "fmt"
5     "golang.org/x/exp/mmap"
6 )
7
8 func main() {
9     at, _ := mmap.Open("./tmp.txt")
10    buff := make([]byte, 2)
11    _, _ = at.ReadAt(buff, 4)
12    _ = at.Close()
13    fmt.Println(string(buff))
14 }
```

```
1 $ echo "abcdefg" > tmp.txt
2 $ go run .
3 ef
```

如果使用 os.File 操作，代码几乎是一样的，os.File 还支持写操作 WriteAt：

```
1 package main
```

```
2
3  import (
4      "fmt"
5      "os"
6  )
7
8  func main() {
9      f, _ := os.OpenFile("tmp.txt", os.O_CREATE|os.O_RDWR, 0644)
10     _, _ = f.WriteAt([]byte("abcdefg"), 0)
11
12     buff := make([]byte, 2)
13     _, _ = f.ReadAt(buff, 4)
14     _ = f.Close()
15     fmt.Println(string(buff))
16 }
```

### 3 mmap(linux)

如果要支持 write 操作，那么就需要直接调用 mmap 的系统调用来实现了。Linux 和 Windows 都支持 mmap，但接口有所不同。对于 linux 系统，mmap 方法定义如下：

```
1  func Mmap(fd int, offset int64, length int, prot int, flags int) (data []byte, err error)
```

每个参数的含义分别是：

- 1 - fd: 待映射的文件描述符。
- 2 - offset: 映射到内存区域的起始位置，0 表示由内核指定内存地址。
- 3 - length: 要映射的内存区域的大小。
- 4 - prot: 内存保护标志位，可以通过或运算符`|`组合
  - 5 - PROT\_EXEC // 页内容可以被执行
  - 6 - PROT\_READ // 页内容可以被读取
  - 7 - PROT\_WRITE // 页可以被写入
  - 8 - PROT\_NONE // 页不可访问
- 9 - flags: 映射对象的类型，常用的是以下两类
  - 10 - MAP\_SHARED // 共享映射，写入数据会复制回文件，与映射该文件的其他进程共享。
  - 11 - MAP\_PRIVATE // 建立一个写入时拷贝的私有映射，写入数据不影响原文件。

首先定义2个常量和数据类型Demo：

```
1  const defaultMaxFileSize = 1 << 30 // 假设文件最大为 1G
2  const defaultMemMapSize = 128 * (1 << 20) // 假设映射的内存大小为 128M
3
```

```
4  type Demo struct {
5      file      *os.File
6      data      *[defaultMaxFileSize]byte
7      dataRef []byte
8  }
9
10 func _assert(condition bool, msg string, v ...interface{}) {
11     if !condition {
12         panic(fmt.Sprintf(msg, v...))
13     }
14 }
```

- 内存有换页机制，映射的物理内存可以远小于文件。
- Demo结构体由3个字段构成，file 即文件描述符，data 是映射内存的起始地址，dataRef 用于后续取消映射。

定义 mmap, grow, ummap 三个方法：

```
1  func (demo *Demo) mmap() {
2      b, err := syscall.Mmap(int(demo.file.Fd()), 0, defaultMemMapSize, syscall.PROT_WRITE
3      _assert(err == nil, "failed to mmap", err)
4      demo.dataRef = b
5      demo.data = (*[defaultMaxFileSize]byte)(unsafe.Pointer(&b[0]))
6  }
7
8  func (demo *Demo) grow(size int64) {
9      if info, _ := demo.file.Stat(); info.Size() >= size {
10         return
11     }
12     _assert(demo.file.Truncate(size) == nil, "failed to truncate")
13 }
14
15 func (demo *Demo) munmap() {
16     _assert(syscall.Munmap(demo.dataRef) == nil, "failed to munmap")
17     demo.data = nil
18     demo.dataRef = nil
19 }
```

- mmap 传入的内存保护标志位为 `syscall.PROT_WRITE|syscall.PROT_READ`，即可读可写，映射类型为 `syscall.MAP_SHARED`，即对内存的修改会同步到文件。
- `syscall.Mmap` 返回的是一个切片对象，需要从该切片中获取到内存的起始地址，并转换为可操作的 byte 数组，byte数组的长度为 `defaultMaxFileSize`。

- grow 用于修改文件的大小，Linux 不允许操作超过文件大小之外的内存地址。例如文件大小为 4K，可访问的地址是 data[0~4095]，如果访问 data[10000] 会报错。
- munmap 用于取消映射。

在文件中写入 hello, geektutu!

```
1 func main() {
2     _ = os.Remove("tmp.txt")
3     f, _ := os.OpenFile("tmp.txt", os.O_CREATE|os.O_RDWR, 0644)
4     demo := &Demo{file: f}
5     demo.grow(1)
6     demo.mmap()
7     defer demo.munmap()
8
9     msg := "hello geektutu!"
10
11     demo.grow(int64(len(msg) * 2))
12     for i, v := range msg {
13         demo.data[2*i] = byte(v)
14         demo.data[2*i+1] = byte(' ')
15     }
16 }
```

- 在调用 mmap 之前，调用了 grow(1)，因为在 mmap 中使用 &b[0] 获取到映射内存的起始地址，所以文件大小至少为 1 byte。
- 接下来，便是通过直接操作 demo.data，修改文件内容了。

运行：

```
1 $ go run .
2 $ cat tmp.txt
3 h e l l o   g e e k t u t u !
```

## 4 mmap(Windows)

相对于 Linux，Windows 上 mmap 的使用要复杂一些。

```
1 func (demo *Demo) mmap() {
2     h, err := syscall.CreateFileMapping(syscall.Handle(demo.file.Fd()), nil, syscall.PA
3     _assert(h != 0, "failed to map", err)
4
5     addr, err := syscall.MapViewOfFile(h, syscall.FILE_MAP_WRITE, 0, 0, uintptr(default
6     _assert(addr != 0, "MapViewOfFile failed", err)
7 }
```

```
8     err = syscall.CloseHandle(syscall.Handle(h));
9     _assert(err == nil, "CloseHandle failed")
10
11     // Convert to a byte array.
12     demo.data = (*[defaultMaxFileSize]byte)(unsafe.Pointer(addr))
13 }
14
15 func (demo *Demo) munmap() {
16     addr := (uintptr)(unsafe.Pointer(&demo.data[0]))
17     _assert(syscall.UnmapViewOfFile(addr) == nil, "failed to munmap")
18 }
```

- 需要 `CreateFileMapping` 和 `MapViewOfFile` 两步才能完成内存映射。`MapViewOfFile` 返回映射成功的内存地址，因此可以直接将该地址转换成 `byte` 数组。
- Windows 对文件的大小没有要求，直接操作内存 `data`，文件大小会自动发生改变。

使用时无需关注文件的大小。

```
1 func main() {
2     _ = os.Remove("tmp.txt")
3     f, _ := os.OpenFile("tmp.txt", os.O_CREATE|os.O_RDWR, 0644)
4     demo := &Demo{file: f}
5     demo.mmap()
6     defer demo.munmap()
7
8     msg := "hello geektutu!"
9     for i, v := range msg {
10         demo.data[2*i] = byte(v)
11         demo.data[2*i+1] = byte(' ')
12     }
13 }
```

```
1 $ go run .
2 $ cat .\tmp.txt
3 h e l l o   g e e k t u t u !
```

## 附 参考

- [edsrzf/mmap-go - github.com](https://github.com/edsrzf/mmap-go)
- [golang 官方文档 syscall - golang.org](https://golang.org)

专题: Go 简明教程

本文发表于 2020-04-20, 最后修改于 2021-02-06。

本站永久域名「[geektutu.com](https://geektutu.com)」, 也可搜索「极客兔兔」找到我。

上一篇 « 动手写ORM框架 - GeeORM第七天 数据库迁移(Migrate)

下一篇 » Go Context 并发编程简明教程

赞赏支持



推荐阅读

动手写RPC框架 - GeeRPC第七天 服务发现与注册中心(registry)  
发表于2020-10-08, 阅读约37分钟


动手写ORM框架 - GeeORM第四天 链式操作与更新删除  
发表于2020-03-08, 阅读约24分钟

TensorFlow 2 / 2.0 中文文档  
发表于2019-07-09, 阅读约11分钟

- #关于我 (9)   #Go (48)   #百宝箱 (2)   #Cheat Sheet (1)   #Go语言高性能编程 (20)   #友链 (1)   #Pandas (3)
- #机器学习 (9)   #TensorFlow (9)   #mnist (5)   #Python (10)   #强化学习 (3)   #OpenAI gym (4)   #DQN (1)
- #Q-Learning (1)   #CNN (1)   #TensorFlow 2 (10)   #官方文档 (10)   #Rust (1)

2 条评论

未登录用户 ▾



说点什么

① 支持 Markdown 语法

使用 GitHub 登录

预览

bxclib2 发表于 3 个月前

↩



感谢您的教程。

```
func (demo *Demo) mmap() {
    b, err := syscall.Mmap(int(demo.file.Fd()), 0, defaultMemMapSize, syscall.PROT_READ|syscall.PROT_WRITE, syscall.MAP_SHARED)
    _assert(err == nil, "failed to mmap", err)
    demo.dataRef = b
    demo.data = (*[defaultMaxFileSize]byte)(unsafe.Pointer(&b[0]))
}
```

大佬可不可以讲下dataRef的作用？



geektutu 发表于 3 个月前



文中有提到，dataRef 保留地址，用于后续取消映射。  
syscall.Munmap(demo.dataRef)。


## Go语言动手写Web框架 - Gee第六天 模板 (HTML Template)

11 评论 • 24天前

 **liron-li** —— urlPattern := path.Join(relativePath, "/\*filepath") 应该是

## Go语言动手写Web框架 - Gee第五天 中间件 Middleware

18 评论 • 14天前

 **YoshieraHuang** —— 你好，大佬写的文章很受用。有个问题想问一下，在使用循环调用


## Go语言动手写Web框架 - Gee第二天 上下文 Context

19 评论 • 1天前

 **DiDiDaDiDiDa** —— 感谢分享~

## Go 语言笔试面试题(基础语法)

6 评论 • 7天前

 **zzhaolei** —— ##### 2. Q13 如何判断 2 个字符串切片 (slice) 是相等的？针对这个中的`b = b[:len(a)]`，使用`go 1.15.6`编译，已经没

Gitalk Plus

© 2021 - 极客兔兔 - 沪ICP备18001798号-1

Powered by [Hexo](#) | Theme [Geektutu](#) 

👁 996642 🗄 4340