矩阵起源 Matrix origin

物理世界的数字化和智能化无处不在 。 矩阵起源（ Matrix Origin ）致力于建设开放的技术开源社区和生态系统、 打造世界级的团队、 并通过业界领先的技术创新和工程能力，实现数据在数字世界中的任意存储和任意计算，帮助用户释放数据的潜力和创新力（ Store Anywhere, Compute Anywhere, Innovate Anywhere ）。

矩阵起源公司成立于2021年，在上海、 深圳、 北京、硅谷等城市设有分支机构。 团队成员由各领域专家组成，在分布式基础架构、 数据库、 大数据及人工智能领域经验丰富。

矩阵起源 Matrix origin

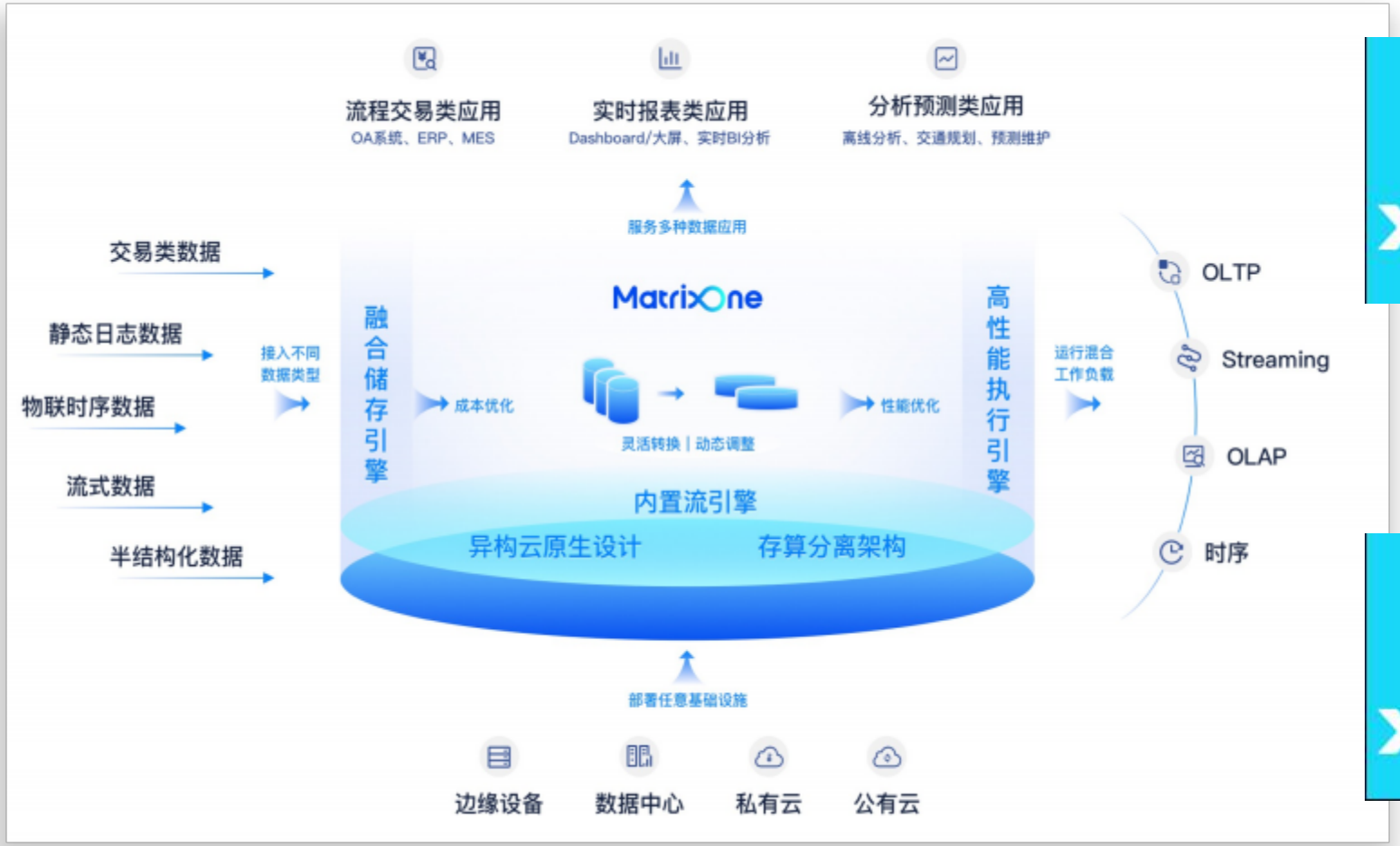
使命

为数字世界提供简捷强大的数据操作系统。

愿景

致力于成为行业领先的数据基础软件公司，帮助所有企业和用户简单、敏捷、高效地拥抱数据价值。

MatrixOne： 立足当下，面向未来的超融合异构数据库



MatrixOne 企业版

企业级超融合异构数据库

稳定高效，专业技术支持

MatrixOne Cloud

全托管云原生数据平台

简单易用， Serverless按量计费

https://github.com/matrixorigin/matrixone

大型项目开发之 Golang 早知道

崔国科

矩阵起源

研发工程师

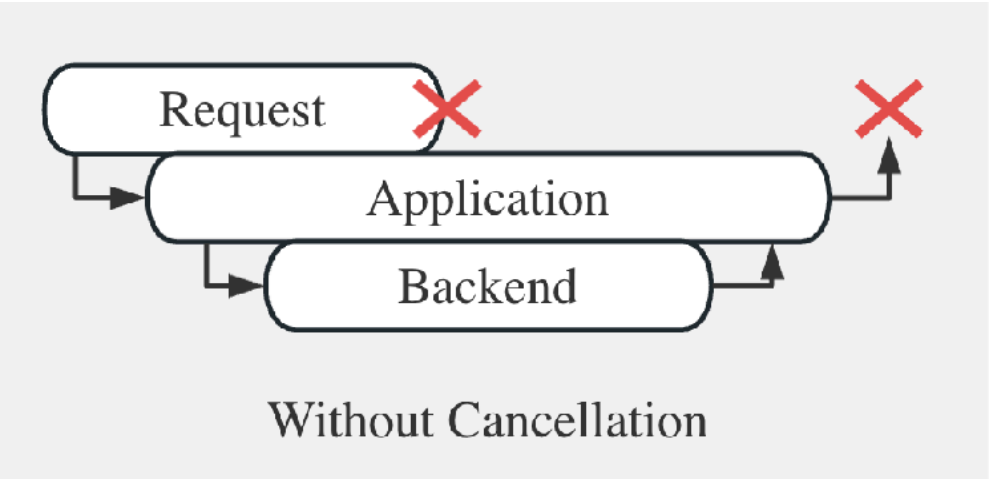
目 录

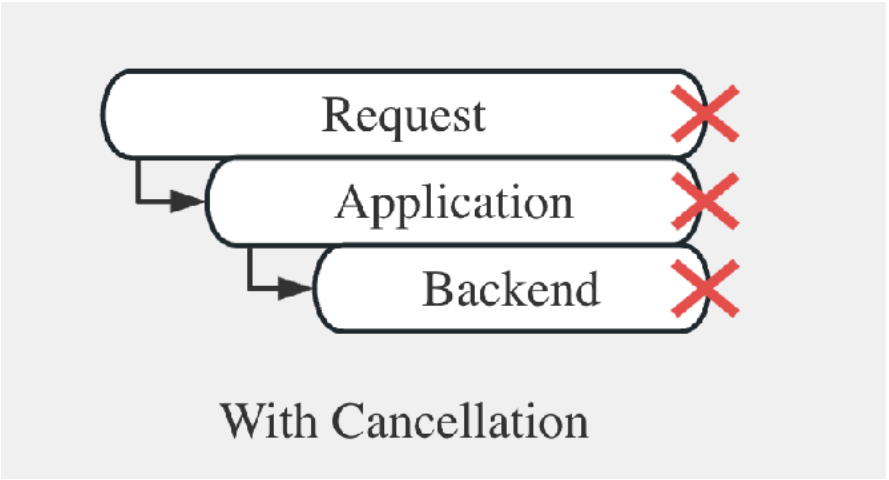
第一部分

Context

1.1 Context Introduction

- 包 context 在 Go 1.7 版本引入的
- 包 context 主要有两个作用：
  - 资源控制：deadline, cancellation
  - 消息传递：request-scoped values between processes





1.2 Context API

```
  // Creating Context
1 func Background() Context
2 func TODO() Context

  // Set Context Deadlines and Timeouts
3 func WithDeadline(parent Context, d time.Time) (Context, CancelFunc)
4 func WithTimeout(parent Context, timeout time.Duration) (Context, CancelFunc)
5 func WithCancel(parent Context) (ctx Context, cancel CancelFunc)
6 func WithCancelCause(parent Context) (ctx Context, cancel CancelCauseFunc)  //
introduced in go1.20

  // Set Request-Scoped Values
7 func WithValue(parent Context, key, val any) Context
```

1.3 Context Example

```
func Stream(ctx context.Context, out chan<- Value) error {
    // Pass a context with a timeout to tell a blocking function
    // that it should abandon its work after the timeout elapses.
1   dctx, cancel := context.WithTimeout(ctx, time.Second * 10)
2   defer cancel()  // releases resources associated with dctx

3   res, err := SlowOperation(dctx)
4   if err != nil {
5     return err
6   }

7   select {
8   case out <- res:  // Read from res; send to out
9   case <-ctx.Done():
10    return ctx.Err()  // Triggered if ctx is cancelled
    }
}
```

1.4 Context Practice

1. 由 WithXXX 函数返回的 context.Context 形成一种派生关系
2. 在 matrixone 演进过程中，一些函数的设计起初并不是 context aware 的，导致了这样一些不足：
    - 无法有效的控制资源，出现了 goroutine 泄漏
    - 传递请求特定的参数时，难以形成一致且简洁的风格
    - 无法兼容基于 context.Context 方案，例如 runtime.trace.Task, OpenTelemetry
    - 后续代码改造往往涉及大量的代码更新
3. 在应用设计和实现阶段，理解并合理应用 context 包是必要的

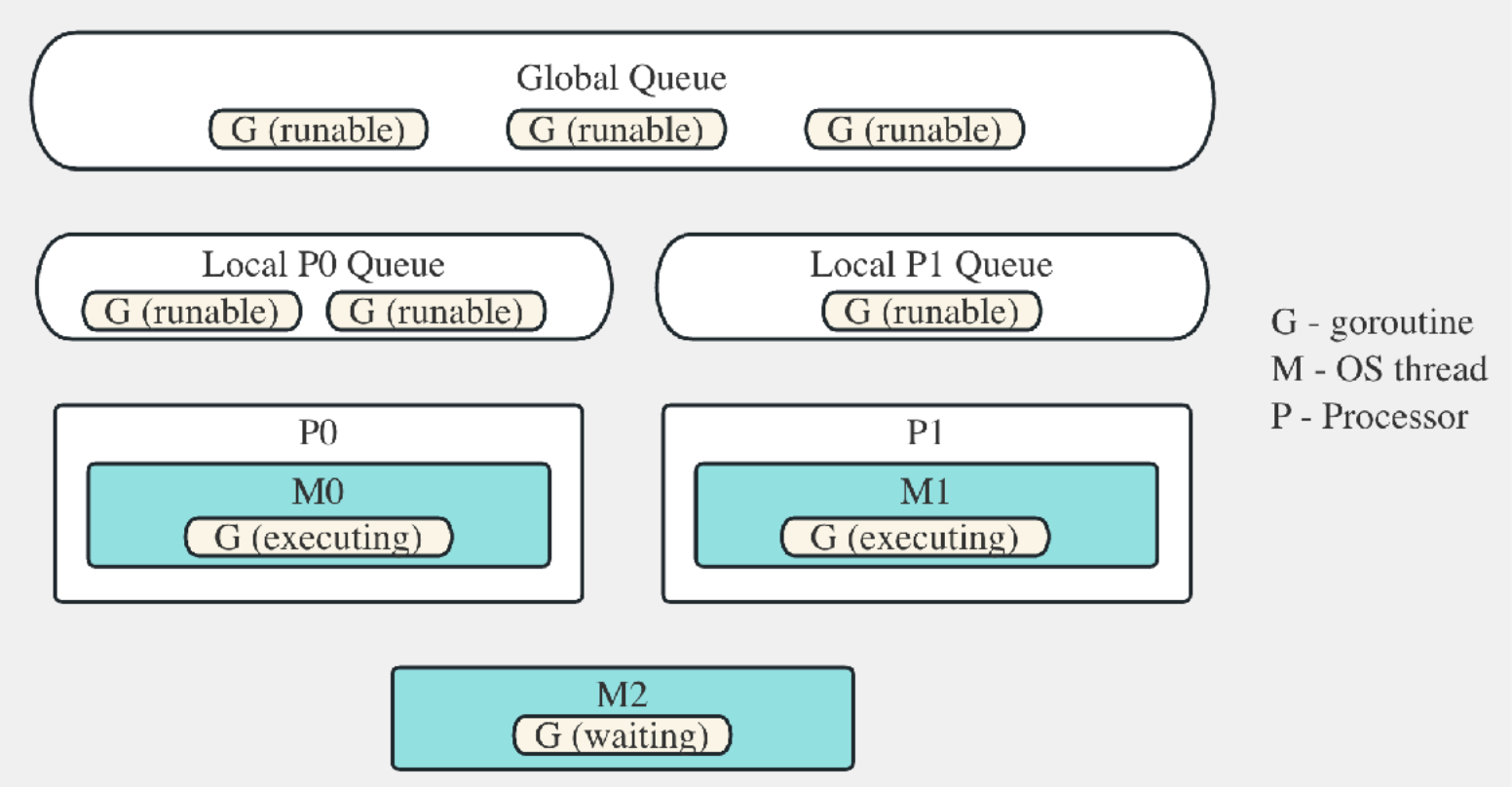
   良好的实现应当形成层次分明的 context.Context 结构

第二部分

Goroutine

2.1 Goroutine GMP

相比 OS Thread，goroutine 是轻量的，但这并不意味着 goroutine 可以不受限制：



- GMP 调度模型决定了过多的 goroutine 会降低应用执行效率

2.1 Goroutine Resource

相比 OS Thread，goroutine 是轻量的，但这并不意味着 goroutine 可以不受限制：

```
type g struct {
    // Stack parameters.
    // stack describes the actual stack memory: [stack.lo, stack.hi).
    // stackguard0 is the stack pointer compared in the Go stack growth prologue.
    // It is stack.lo+StackGuard normally, but can be StackPreempt to trigger a preemption.
    // stackguard1 is the stack pointer compared in the C stack growth prologue.
    // It is stack.lo+StackGuard on g0 and gsignal stacks.
    // It is ~0 on other goroutine stacks, to trigger a call to morestackc (and crash).
    stack       stack   // offset known to runtime/cgo
    stackguard0 uintptr // offset known to liblink
    stackguard1 uintptr // offset known to liblink

    _panic    *_panic // innermost panic - offset known to liblink
    _defer    *_defer // innermost defer
    m         *m      // current m; offset known to arm liblink
```

- goroutine 需要消耗内存资源：
  - stack - 最小 2 KB，随着 goroutine 执行情况动态扩/缩容

2.1 Goroutine Issue 1

在 matrixone 演进过程中，出现了这样一些与 goroutine 相关的问题：

- goroutine 泄漏

```
38565 goroutine 30196 [chan send, 61 minutes]:
38566 github.com/matrixorigin/matrixone/pkg/vm/engine/disttae.
38567     /go/src/github.com/matrixorigin/matrixone/pkg/vm/engine/disttae/...
38568 created by github.com/matrixorigin/matrixone/pkg/vm/engine/disttae.
38569     /go/src/github.com/matrixorigin/matrixone/pkg/vm/engine/disttae/...
```

2.1 Goroutine Issue 2

在 matrixone 演进过程中，出现了这样一些与 goroutine 相关的问题：

- 部分模块未限制 goroutine 规模，过多的 goroutine 导致了明显的调度延迟

```
Goroutines:
github.com/lni/dragonboat/v4/internal/tan.(*LogDB).sequentialSaveState.func1 N=6963
github.com/fagongzi/goetty/v2.(*server).doStart.func2.1 N=8
runtime.gcBgMarkWorker N=10
github.com/matrixorigin/matrixone/pkg/common/stopper.(*Stopper).doRunCancelableTask.func1 N=40
github.com/matrixorigin/matrixone/pkg/util/export.(*MOCollector).doExport N=8
github.com/matrixorigin/matrixone/pkg/util/export.(*MOCollector).doGenerate N=8
github.com/lni/goutils/syncutil.(*Stopper).runWorker.func1 N=36
```

2.1 Goroutine Practice 1

在应用设计和实现阶段，关于 goroutine 的使用，应当注意的是：

- 尽量避免直接用关键字 go 启动 goroutine，可通过统一构建的工具函数启动

  例如 github.com/cockroachdb/cockroach/pkg/util/stop.(*Stopper).RunAsyncTask

```
func (s *Stopper) RunAsyncTask(
  ctx context.Context, taskName string, f func(context.Context),
) error {
  return s.RunAsyncTaskEx(ctx,
    TaskOpts{
      TaskName:   taskName,
      SpanOpt:    FollowsFromSpan,
      Sem:        nil,
      WaitForSem: false,
    },
  f)
}
```

2.1 Goroutine Practice 2

在应用设计和实现阶段，关于 goroutine 的使用，应当注意的是：

- 启动 goroutine 时候，考虑退出条件，例如通过 context.Context 来通知 goroutine 退出

```
1  go func(ctx context.Context, notifyChan <-chan struct{}) {
2    for {
3      select {
4      case <-ctx.Done(): // 通过 ctx.Done() 来退出 goroutine
5        return
6      case <-notifyChan:
7        doSomeTask(ctx)
8      }
9    }
10 }(ctx, notify)

// 基于 context.Context 来通知 goroutine 退出意味着额外的 goroutine
// 更直接的方式是基于 done channel 来通知 goroutine 退出
```

第三部分

Abstraction

3.1 Introduction

Golang 提供的抽象工具：

- Interface
- Generics

借助 interface 或者 generics，Golang 程序员可以编写类型无关的代码

3.2 Interface Introduction

interface 定义了对象的行为：不同的对象实现同一接口，可以表现出相同的行为。

```
package io

// Read reads data from data source into `p`
type Reader interface {
    Read(p []byte) (n int, err error)
}
```

基于 interface 编写类型无关代码，可以带来这样一些好处：

1. 提高代码复用，基于行为组织代码，而不用受限于某个具体的类型
2. 基于 mock，便于实现单元测试

3.2 Interface Issue 1

在 matrixone 演进过程中，出现了这样一些与 interface 有关的问题：

- 过度使用 interface，即使实现这个接口的具体类型只有一个

过度使用 interface 意味着过早的抽象：

- 基于 interface 的抽象会牺牲代码的可读性
- 并且 interface 不是“免费”的

```
Don’t design with interfaces, discover them. - Rob Pike
```

3.2 Interface Practice 1

通常碰到下面这些情况时，我们考虑使用 interface：

- 当多个对象表现出相同的行为时
- 为了解除耦合，便于负责不同模块的小组之间并行工作
- 作为函数参数类型，限制该函数对参数所执行的操作

```
package io

func ReadAll(r Reader) ([]byte, error)
```

3.2 Interface Issue 2

在 matrixone 项目中，还有这样一个 interface 相关的问题：

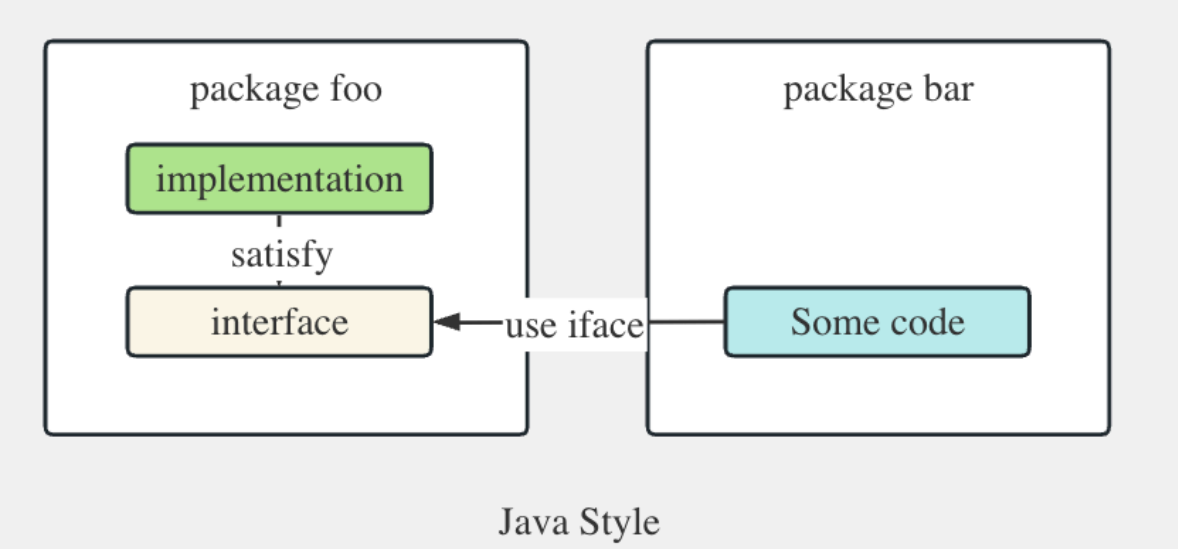
- 延续 Java 风格，总是将接口同具体实现组织在一起

延续 Java 风格意味着无法灵活控制 interface 的粒度：

- 对于某些具体实现，即便某些方法不是必要的，也不得不实现

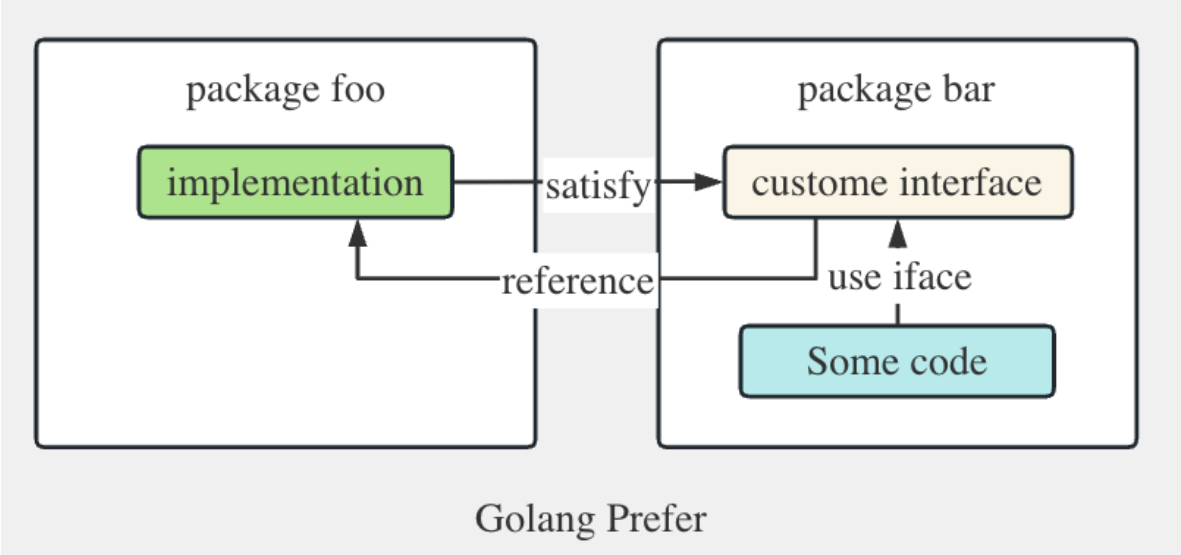
```
func (m *blockStream) DeleteCache(cacheID uint64) {
    panic("not implement")
}
func (m *blockStream) GetCache(cacheID uint64) (morpc.MessageCache, error) {
    panic("not implement")
}
```

3.2 Interface Practice 2



Java Style

延续 Java 风格意味着无法灵活控制 interface 的粒度



Golang Prefer

与 Java 通过关键字 *implements* 显式地标记接口实现不同，Golang 接口的实现是隐式的

3.2 Interface Practice 2

大多数情况下，interface 应该出现在使用侧，而非实现侧

但这并非绝对，例如标准库中 io.Reader, gob.BinaryMarshaler 是预先定义好的

并非定义在使用侧，不过这种预先定义的接口粒度通常较小

3.3 Generics Introduction

介绍完 interface，我们看下 Golang 1.18 引入的 Generics

3.3 Generics Introduction

泛型可以用来编写模版代码以适应不同的类型。

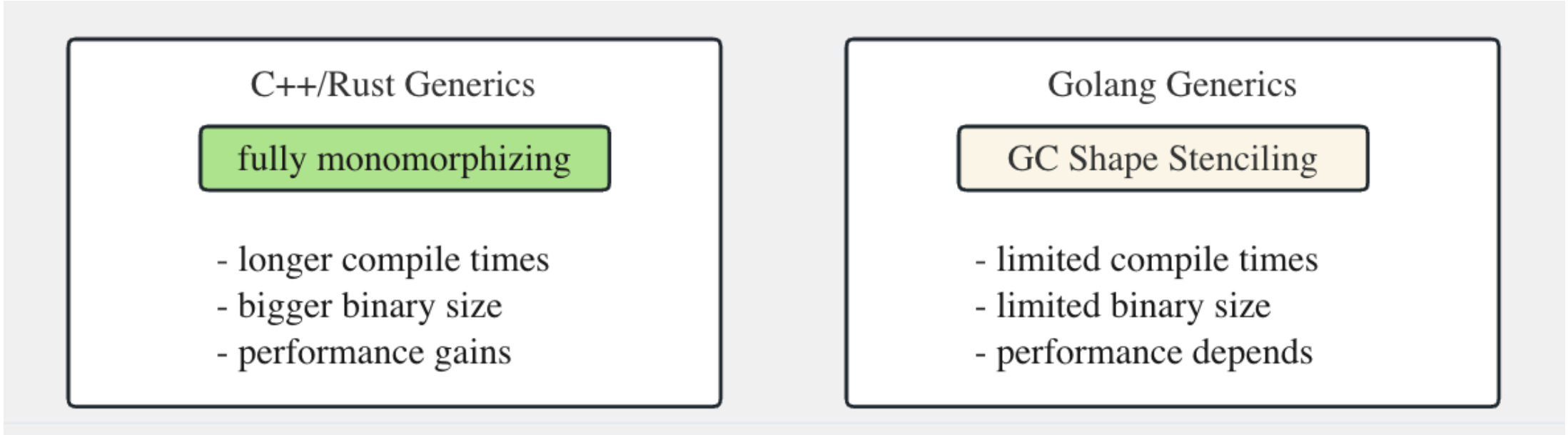
在工程实践中，我们经常碰到这样的情况：同一份逻辑可以应用到多个不同的类型：

- 不管是 int 类型还是 uint 类型的数据集，都可以应用相同的排序算法
- 不管链表节点上保存什么类型的数据，链表本身的操作都是一致的

类似的情况，同样的算法或者逻辑，难道要为每种类型都实现一遍吗?

有了泛型，相同的逻辑，我们就不需要维护多份代码，代码可维护性从而得到了大大提高。

3.3 Generics Implementation



Golang Generics Design

3.3 Generics Practice 1

Golang Generics 可以带来软件工程方面的收益：

- 减少对样板代码的 copy-paste
- 提高开发效率，有利于提高代码的可维护性

但是 Golang Generics 不一定能带来程序运行效率上的提升：

- 对于性能敏感的代码，需要通过严谨的性能测试来评估泛型给程序性能带来的影响
- 对性能敏感的代码，建立必要的性能测试集，

  评估 Golang 版本演进可能带来的泛型性能方面的变化

```
As Go developers, we have lived without them for more than a decade.
```

3.3 Generics Practice 2

Golang Generics 和 interface 有其各自的应用场景：

- generics 适用于类型无关的算法或者数据结构，例如链表、BTree
- 当我们依赖函数参数的行为时，考虑使用 interface

```
// In this case, using generics won't bring any value to our code.
// make the w argument an io.Writer directly.
func foo[T io.Writer](w T) {
  b := getBytes()
  _, _ = w.Write(b)
}
```

第四部分

Performance

4.1 Introduction

如果不限定范围，“性能”这个概念本身可以覆盖比较大的范畴：

- 程序开发是否高效
- 程序维护是否容易
- 程序运行是否效率

这一部分主要从程序运行效率的角度讨论性能

4.2 Memory Escape Introduction

```go
package main

//go:noinline
func makeBuffer() []byte {
    return make([]byte, 1024)
}

func main() {
    buf := makeBuffer()
    for i := range buf {
        buf[i] = buf[i] + 1
    }
}
```

```
$ go build -gcflags="-m" escape.go
```

示例代码中函数 makeBuffer 返回的内存位于函数栈上。

在 C 语言中，这是一段有问题的代码，会导致未定义的行为：

- clang 13.0.0 编译器会给出警告 -Wreturn-stack-address

在 Go 语言中，这样的写法是允许的。Go 编译器会执行 escape analysis：

当它发现一段内存不能放置在函数栈上时，会将这段内存放置在堆内存上

4.2 Memory Escape Cases

此外，还存在其他一些情况会触发内存的"逃逸"：

- 全局变量，因为它们可能被多个 goroutine 并发访问
- 局部变量过大，无法放在函数栈上
- 本地变量的大小在编译时未知

```
// if n is unknown when compiling
// then `s` would be placed on heap memory
s := make([]int, n)
```

- 对 slice 的 append 操作触发了其底层数组重新分配
- 通过 channel 传送指针

```
type Hello struct { name string }
ch := make(chan *Hello, 1)
ch <- &Hello{ name: "world"}
```

- 通过 channel 传送的结构体中持有指针

```
type Hello struct { name *string }
ch := make(chan *Hello, 1)
name := "world"
ch <- Hello{ name: &name }
```

4.2 Memory Escape Practice

在应用设计和实现阶段，注意避免因内存逃逸导致的不必要的动态内存分配：

- 保持合理的函数签名设计，自下向上返回栈内存会触发内存“逃逸”

  例如 github.com/cockroachdb/cockroach/pkg/util/encoding.EncodeUint32Ascending

```
// EncodeUint32Ascending encodes the uint32 value using a big-endian 4 byte
// representation. The bytes are appended to the supplied buffer and
// the final buffer is returned.
1 func EncodeUint32Ascending(b []byte, v uint32) []byte {
2   return append(b, byte(v>>24), byte(v>>16), byte(v>>8), byte(v))
3 }
```

4.3 Interface Introduction

介绍完 Golang 内存逃逸，我们重新看下 interface 类型

4.3 Interface Introduction

Golang 中 接口实现 为一个"胖"指针：

- 一个指向函数指针表（类似于C++ 中的虚函数表）
- 一个指向实际的数据

```
type iface struct {
  tab  *itab
  data unsafe.Pointer
}
```

4.3 Interface Benchmark

```go
package interfaces

import (
        "testing"
)

var global interface{}

func BenchmarkInterface(b *testing.B) {
        var local interface{}
        for i := 0; i < b.N; i++ {
                // assign value to interface{}
                local = calculate(i)
        }
        global = local
}
```

```go
// values is bigger than single machine word.
type values struct {
        value  int
        double int
        triple int
}

func calculate(i int) values {
        return values{
                value:  i,
                double: i * 2,
                triple: i * 3,
        }
}
```

4.3 Interface Benchmark

```
$ go test -run none -bench Interface -benchmem -memprofile mem.out
goos: darwin
goarch: arm64
pkg: github.com/cnutshell/go-pearls/memory/interfaces
BenchmarkInterface-8    101292834     11.80 ns/op    24 B/op      1 allocs/op
PASS
ok      github.com/cnutshell/go-pearls/memory/interfaces        2.759s
$ go tool pprof -alloc_space -flat mem.out
```

```
2.31GB     2.31GB (flat, cum) 99.89% of Total
      .          .      7:var global interface{}
      .          .      8:
      .          .      9:func BenchmarkInterface(b *testing.B) {
      .          .     10:    var local interface{}
      .          .     11:    for i := 0; i < b.N; i++ {
2.31GB     2.31GB     12:            local = calculate(i) // assign value to interface{}
      .          .     13:    }
      .          .     14:    global = local
      .          .     15:}
      .          .     16:
      .          .     17:// values is bigger than single machine word.
```

4.3 Interface Note

向接口类型的变量赋值，可能会触发内存“逃逸”，导致动态内存分配。

基于接口实现抽象，主要存在这些问题：

- 丢失了类型信息，程序行为从编译阶段转移到运行阶段；
- 程序运行阶段不可避免地需要执行类型转换、类型断言或者反射等等操作；
- 为接口类型的变量赋值可能会触发内存逃逸；
- Golang 中，基于接口的函数调用，其实际的调用开销为：

  指针解引用（确定方法地址）+ 函数执行开销

  编译器无法对其执行内联优化，内联优化后进一步的优化也变得不可能；

4.3 Interface Practice

对于项目开发而言，关于接口的使用，需要注意的是：

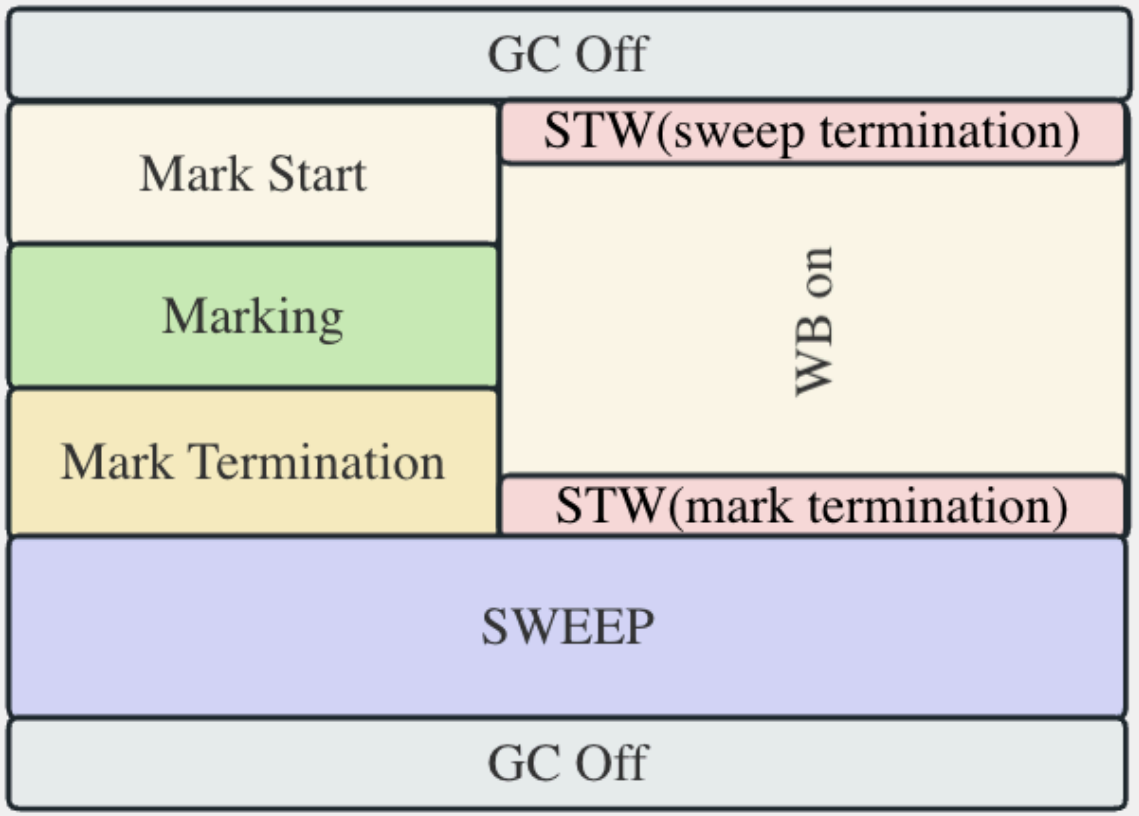
- 基于接口类型的抽象不是免费的，避免过早的抽象；
- 对于处在 hot path 上的数据结构或者函数，谨慎地使用接口类型变量；
- 将接口类型改为范型类型，是避免额外内存分配，优化程序性能的一个手段；

4.4 GC

前面我们了解到，Golang 编译器执行 escape analysis 后，根据需要数据可能被“搬”到堆内存上。

这里简单地介绍下 Golang 的 GC，从而了解写 Golang 代码时为什么应该尽量避免“额外的”内存分配。

4.4 GC Introduction



GC Disabled

Write Barrier on

Collection pointer from global and goroutine stack, follow pointers and mark objects.

Write Barrier off

Reclaim unmarked objects as needed, Adjust gc pacing

Reclaim and repeat

Golang GC Mechanics

4.4 GC Note

如果我们的代码中存在大量"额外"的堆内存分配，尤其是在代码关键路径上，

对于性能的负面影响是非常大的：

- 首先，堆内存的分配本身就是相对耗时的操作
- 其次，大量"额外"的堆内存分配意味着额外的 GC 过程，STW 会进一步影响程序执行效率
- 极端情况下，短时间内大量的堆内存分配，可能会直接触发 OOM，GC 甚至都没有执行的机会

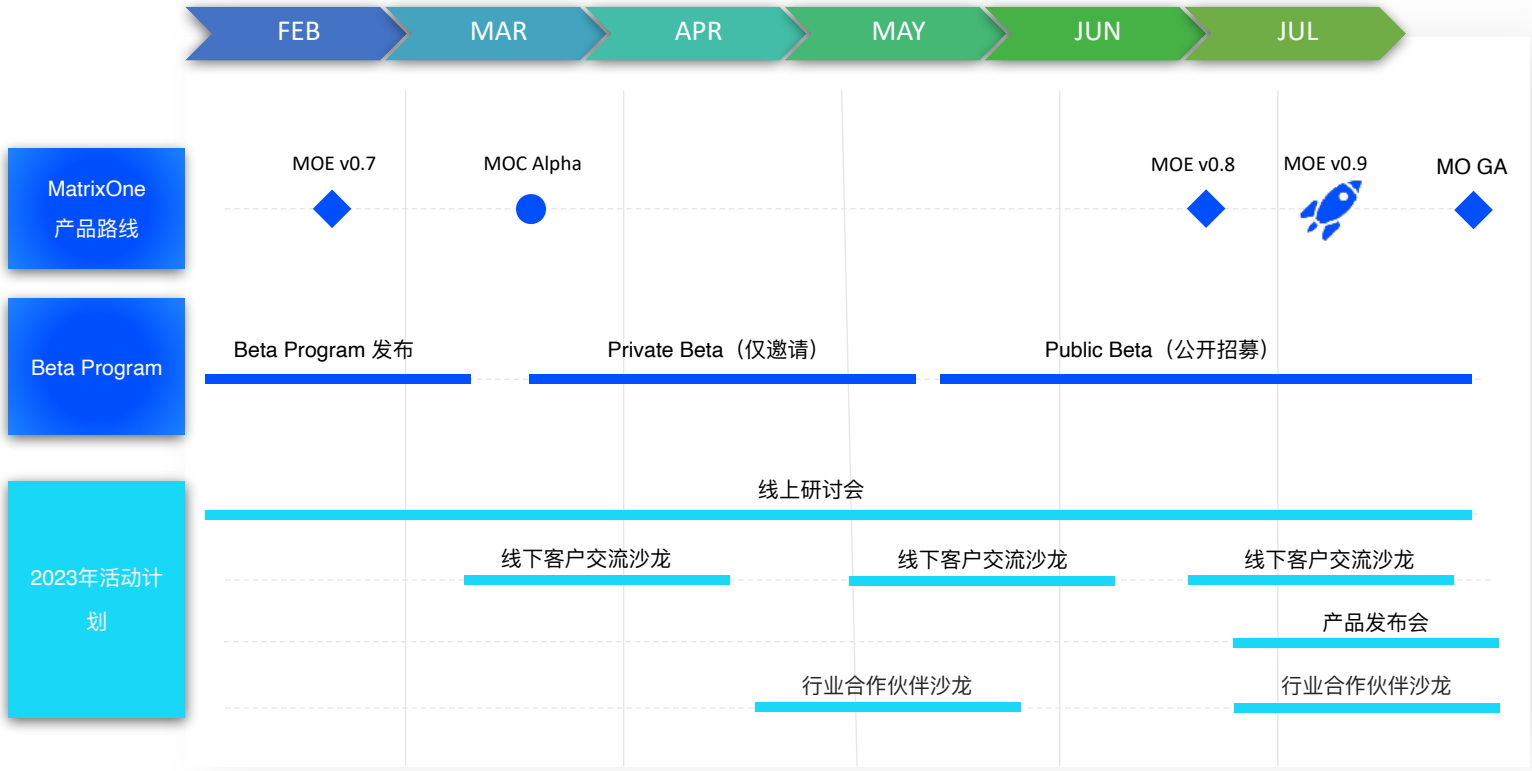
认为 Golang 有 GC 在，写起代码来便洋洋洒洒，这是很常见的谬误

Thanks and FAQ

欢迎加入 MatrixOne Beta Program 用户体验计划

MatrixOne Beta Program 是矩阵起源全新推出的，与客户、用户一起持续提升 MatrixOne 产品和性能体验优化的计划。

- 新功能内测权益
- 产品设计参与权益
- 新功能本地环境优先测试权益
- 开发过程的直接发言权益
- 专家端到端专业支持权益



加入 MatrixOne Beta Program

- Step1: 扫描下方小程序码提交注册
- Step2: MO架构师将会通过邮件的方式进行初步联系和沟通
- Step3: 加入 Beta Program 社区，开始您和 MatrixOne 的旅程

矩阵起源 Matrix origin

加入 MatrixOne 社区群，获取 PPT

www.matrixorigin.cn