go语言基础

effective-go 中文版

<https://github.com/bingohuang/effective-go-zh-en/blob/master/03_Commentary.md>

**Context（跟踪协程的方案）**

通过 select+channel 让协程退出的方式比较优雅，但是如果我们希望做到同时取消很多个协程呢？如果是定时取消协程又该怎么办？这时候 select+channel 的局限性就凸现出来了，即使定义了多个 channel 解决问题，代码逻辑也会非常复杂、难以维护。

要解决这种复杂的协程问题，必须有一种可以跟踪协程的方案，只有跟踪到每个协程，才能更好地控制它们，这种方案就是 Go 语言标准库为我们提供的 Context。

什么是Context？

一个任务会有很多个协程协作完成，一次 HTTP 请求也会触发很多个协程的启动，而这些协程有可能会启动更多的子协程，并且无法预知有多少层协程、每一层有多少个协程。

如果因为某些原因导致任务终止了，HTTP 请求取消了，那么它们启动的协程怎么办？该如何取消呢？因为取消这些协程可以节约内存，提升性能，同时避免不可预料的 Bug。

Context 就是用来简化解决这些问题的，并且是并发安全的。Context 是一个接口，它具备手动、定时、超时发出取消信号、传值等功能，主要用于控制多个协程之间的协作，尤其是取消操作。一旦取消指令下达，那么被 Context 跟踪的这些协程都会收到取消信号，就可以做清理和退出操作。

Context是Go语言中用于控制goroutine生命周期、传递请求范围值和取消信号的标准库组件。它的核心设计思想是通过树形结构传播取消信号和共享数据。

Context 接口只有四个方法

type Context interface {

Deadline() (deadline time.Time, ok bool)

Done() <-chan struct{}

Err() error

Value(key interface{}) interface{}

}

1、Deadline 方法可以获取设置的截止时间，第一个返回值 deadline 是截止时间，到了这个时间点，Context 会自动发起取消请求，第二个返回值 ok 代表是否设置了截止时间。

2、Done 方法返回一个只读的 channel，类型为 struct{}。在协程中，如果该方法返回的 chan 可以读取，则意味着 Context 已经发起了取消信号。通过 Done 方法收到这个信号后，就可以做清理操作，然后退出协程，释放资源。

3、Err 方法返回取消的错误原因，即因为什么原因 Context 被取消。

4、Value 方法获取该 Context 上绑定的值，是一个键值对，所以要通过一个 key 才可以获取对应的值。

Context 接口的四个方法中最常用的就是 Done 方法，它返回一个只读的 channel，用于接收取消信号。当 Context 取消的时候，会关闭这个只读 channel，也就等于发出了取消信号。

Context树

Go 语言提供了函数可以帮助我们生成不同的 Context，通过这些函数可以生成一颗 Context 树，这样 Context 才可以关联起来，父 Context 发出取消信号的时候，子 Context 也会发出，这样就可以控制不同层级的协程退出。

从使用功能上分，有四种实现好的 Context。

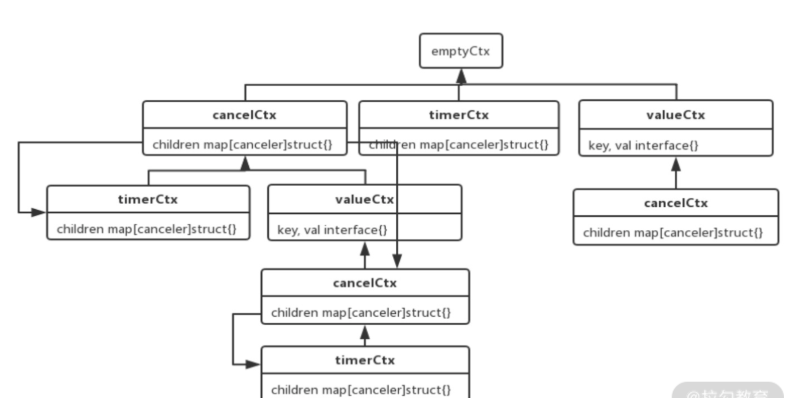
1、空 Context：不可取消，没有截止时间，主要用于 Context 树的根节点。

2、可取消的 Context：用于发出取消信号，当取消的时候，它的子 Context 也会取消。

3、可定时取消的 Context：多了一个定时的功能。

4、值 Context：用于存储一个 key-value 键值对。

从下图 Context 的衍生树可以看到，最顶部的是空 Context，它作为整棵 Context 树的根节点，在 Go 语言中，可以通过 context.Background() 获取一个根节点 Context。



（四种 Context 的衍生树）

有了根节点 Context 后，这颗 Context 树要怎么生成呢？需要使用 Go 语言提供的四个函数。

WithCancel(parent Context)：生成一个可取消的 Context。

WithDeadline(parent Context, d time.Time)：生成一个可定时取消的 Context，参数 d 为定时取消的具体时间。

WithTimeout(parent Context, timeout time.Duration)：生成一个可超时取消的 Context，参数 timeout 用于设置多久后取消

WithValue(parent Context, key, val interface{})：生成一个可携带 key-value 键值对的 Context。

以上四个生成 Context 的函数中，前三个都属于可取消的 Context，它们是一类函数，最后一个是值 Context，用于存储一个 key-value 键值对。

使用原则

Context 是一种非常好的工具，使用它可以很方便地控制取消多个协程。在 Go 语言标准库中也使用了它们，比如 net/http 中使用 Context 取消网络的请求。

要更好地使用 Context，有一些使用原则需要尽可能地遵守。

Context 不要放在结构体中，要以参数的方式传递。

Context 作为函数的参数时，要放在第一位，也就是第一个参数。

要使用 context.Background 函数生成根节点的 Context，也就是最顶层的 Context。

Context 传值要传递必须的值，而且要尽可能地少，不要什么都传。

Context 多协程安全，可以在多个协程中放心使用。

以上原则是规范类的，Go 语言的编译器并不会做这些检查，要靠自己遵守。

总结：

Context 通过 With 系列函数生成 Context 树，把相关的 Context 关联起来，这样就可以统一进行控制。一声令下，关联的 Context 都会发出取消信号，使用这些 Context 的协程就可以收到取消信号，然后清理退出。你在定义函数的时候，如果想让外部给你的函数发取消信号，就可以为这个函数增加一个 Context 参数，让外部的调用者可以通过 Context 进行控制，比如下载一个文件超时退出的需求。

核心原理：

1. 取消机制

cancelCtx的实现关键点：

包含一个done原子值，存储chan struct{}(懒初始化)

使用互斥锁保护children map和err状态

取消时会关闭done channel，递归取消所有子context

type cancelCtx struct {

Context

mu sync.Mutex

done atomic.Value

children map[canceler]struct{}

err error

}

2. 超时机制

timerCtx在cancelCtx基础上增加了：

定时器：到达截止时间后自动触发取消

截止时间：用于提前判断是否已超时

type timerCtx struct {

cancelCtx

timer \*time.Timer

deadline time.Time

}

3. 值传递机制

valueCtx采用链表形式存储键值对：

每次WithValue创建一个新的valueCtx节点

Value查找时沿链表向上查找

查找复杂度O(n)，不适合存储大量数据

type valueCtx struct {

Context

key, val interface{}

}

设计特点

不可变性：Context一旦创建不可修改，只能派生子Context

线程安全：所有方法都保证并发安全

链式传播：取消信号会从父到子传播

轻量级：每个Context只添加必要的功能

性能考量

Context设计非常轻量，创建和取消开销小

Value查找是线性复杂度，不适合高频访问

合理复用Background和TODO作为根节点

避免创建过深的Context链

线程安全机制：

1. cancelCtx的线程安全设计

双重检查锁（Done方法）

使用atomic.Value实现懒加载

互斥锁保护channel的初始化

避免每次访问都加锁

(2) 取消操作的同步

互斥锁保护取消状态

原子操作保证channel关闭安全

递归取消时保持锁持有

2. timerCtx的线程安全

继承cancelCtx的锁机制

定时器回调与手动取消竞态处理

1. valueCtx的线程安全
2. 完全不可变设计
3. value方法无锁实现

**Channel**

Go 语言提供的 channel（通道）解决协程通信的问题。

一个 chan 的操作只有两种：发送和接收。

接收：获取 chan 中的值，操作符为 <- chan。

发送：向 chan 发送值，把值放在 chan 中，操作符为 chan <-。

这里注意发送和接收的操作符，都是 <- ，只不过位置不同。接收的 <- 操作符在 chan 的左侧，发送的 <- 操作符在 chan 的右侧。

无缓冲管道：

容量是 0，不能存储任何数据。所以无缓冲 channel 只起到传输数据的作用，数据并不会在 channel 中做任何停留。这也意味着，无缓冲 channel 的发送和接收操作是同时进行的，它也可以称为同步 channel。

有缓冲管道：

一个有缓冲 channel 具备以下特点：

有缓冲 channel 的内部有一个缓冲队列；

发送操作是向队列的尾部插入元素，如果队列已满，则阻塞等待，直到另一个 goroutine 执行，接收操作释放队列的空间；

接收操作是从队列的头部获取元素并把它从队列中删除，如果队列为空，则阻塞等待，直到另一个 goroutine 执行，发送操作插入新的元素。

单向channel

限制一个 channel 只可以接收但是不能发送，或者限制一个 channel 只能发送但不能接收，这种 channel 称为单向 channel。

单向 channel 的声明也很简单，只需要在声明的时候带上 <- 操作符即可，

onlySend := make(chan<- int)

onlyReceive:=make(<-chan int)

原理：

底层数据结构：

type hchan struct {

qcount uint // 当前队列中元素数量

dataqsiz uint // 环形队列大小（固定长度）

buf unsafe.Pointer // 指向环形队列的指针

elemsize uint16 // 每个元素的大小

closed uint32 // 关闭标志（0未关闭，1已关闭）

elemtype \*\_type // 元素类型信息（用于GC等）

sendx uint // 发送位置索引（环形队列）

recvx uint // 接收位置索引（环形队列）

recvq waitq // 阻塞的接收者队列（sudog链表）

sendq waitq // 阻塞的发送者队列（sudog链表）

lock mutex // 互斥锁（非Go的sync.Mutex）

}

核心工作原理：

1. 环形缓冲区机制

有缓冲channel使用环形队列（循环数组）存储数据

sendx和recvx分别指向下一次发送/接收的位置

当索引到达数组末尾时自动回绕到开头

2. 阻塞/唤醒机制

当缓冲区满时，发送者goroutine被包装为sudog加入sendq并阻塞

当缓冲区空时，接收者goroutine被包装为sudog加入recvq并阻塞

当有对应操作时（如新接收者到来），从等待队列唤醒最早的goroutine

3. 直接传递优化

无缓冲channel或缓冲区满时的特殊情况：

如果发送时发现有接收者阻塞，直接拷贝数据到接收者

如果接收时发现有发送者阻塞，直接从发送者获取数据

避免不必要的缓冲区操作

线程安全实现机制：

1、互斥锁保护

每个hchan都内置一个lock mutex（专门优化的轻量级互斥锁）

所有channel操作（发送/接收/关闭）都必须先获取该锁

锁保护以下关键字段：

缓冲区状态（qcount, sendx, recvx）

等待队列（recvq, sendq）

关闭标志（closed）

2. 原子操作

对closed标志的读写使用原子操作（避免加锁）

某些状态检查使用原子操作优化

3. 等待队列管理

使用sudog结构将goroutine与等待队列关联

等待队列操作都在锁保护下进行

goroutine阻塞/唤醒通过gopark和goready实现

4. 内存可见性保证

channel操作建立happens-before关系：

第n个发送操作 happens-before 第n个接收完成

channel的关闭 happens-before 所有接收完成

通过内存屏障保证多核CPU下的可见性

四、特殊场景优化

1. 无竞争场景优化

当没有等待goroutine且缓冲区未满时，快速路径处理

减少锁占用时间

2. select语句优化

编译器会将select中的channel操作转换为特定函数调用

实现非阻塞检查和随机选择

3. 零值channel处理

对nil channel的操作会永久阻塞（通过显式检查实现）

五、性能关键点

锁粒度：单个channel一个锁，不同channel无竞争

内存拷贝：所有数据传递都涉及内存拷贝（包括直接传递）

调度开销：goroutine阻塞/唤醒需要调度器介入

缓存友好性：环形缓冲区设计提高CPU缓存命中率

**Error、Defer、Panic、Recover**

一 Error错误处理机制

底层实现原理

1. 接口设计

type error interface {

Error() string

}

任何实现Error() string方法的类型都是error类型

标准库errors.New()返回的是\*errorString指针类型

2、错误链

fmt.Errorf使用%w创建包装错误(wrapError)

Unwrap()方法实现错误解包

errors.Is/As实现错误类型检查和提取

1. 性能优化

避免频繁创建相同错误的实例

简单错误使用值类型，复杂错误使用指针类型

1. Defer延迟调用机制

底层实现原理

数据结构：

每个goroutine有一个\_defer链表

后进先出(LIFO)执行顺序

编译器处理：

// 编译器将defer语句转换为

d := newdefer()

d.fn = func

d.link = gp.\_defer

gp.\_defer = d

关键优化：

开放编码优化(open-coded defer)：对简单defer直接插入调用

堆分配优化：减少小defer的内存分配

执行时机：

函数返回前

发生panic时

1. Panic异常抛出机制

底层实现原理

1. 数据结构

type \_panic struct {

argp unsafe.Pointer

arg interface{} // panic参数

link \*\_panic // 嵌套panic链表

recovered bool // 是否被recover

aborted bool // 是否被终止

}

执行流程：

创建panic对象并加入goroutine的panic链表

执行当前函数的所有defer

如果没有被recover，打印堆栈并终止程序

性能特点：

比错误处理开销大很多

不应用于常规错误处理

1. Recover恢复机制

底层实现原理

工作条件：

必须在defer函数中调用

只能恢复当前goroutine的panic

实现机制：

func recover() interface{} {

gp := getg()

if gp.\_panic == nil || gp.\_panic.goexit {

return nil

}

gp.\_panic.recovered = true

return gp.\_panic.arg

}

恢复流程：

标记panic为已恢复

阻止panic继续向上传播

正常执行剩余的defer

1. 四者协同工作原理
2. 典型panic-recover模式：

defer func() {

if err := recover(); err != nil {

// 处理panic

}

}()

执行顺序：

defer按照LIFO顺序执行

recover必须位于可能发生panic的defer中

panic会执行所有已注册的defer

错误处理最佳实践：

常规错误使用error返回值

真正不可恢复的错误使用panic

recover只用于最外层保护性恢复

六、性能对比与选型

机制 开销级别 适用场景

error 低 常规错误处理

defer 中 资源清理、锁释放等

panic 高 真正不可恢复的严重错误

recover 高 顶层保护性恢复

Sync包中的元素

1. sync.Mutex, sync.RWMutex

mutex:互斥锁可以被用来保护一个临界区或者一组相关临界区。我们可以通过它来保证，在同一时刻只有一个goroutine处于该临界区之内。

使用互斥锁的注意事项如下：

不要重复锁定互斥锁；

不要忘记解锁互斥锁，必要时使用defer语句；

不要对尚未锁定或者已解锁的互斥锁解锁；

不要在多个函数之间直接传递互斥锁。

一个读写锁中实际上包含了两个锁，即：读锁和写锁。sync.RWMutex类型中的Lock方法和Unlock方法分别用于对写锁进行锁定和解锁，而它的RLock方法和RUnlock方法则分别用于对读锁进行锁定和解锁。

另外，对于同一个读写锁来说有如下规则。

在写锁已被锁定的情况下再试图锁定写锁，会阻塞当前的goroutine。

在写锁已被锁定的情况下试图锁定读锁，也会阻塞当前的goroutine。

在读锁已被锁定的情况下试图锁定写锁，同样会阻塞当前的goroutine。

在读锁已被锁定的情况下再试图锁定读锁，并不会阻塞当前的goroutine。

sync.Mutex互斥锁

底层数据结构

type Mutex struct {

state int32 // 锁状态标志位

sema uint32 // 信号量(用于阻塞/唤醒goroutine)

}

2. 状态位解析（state）

最低3位分别表示：

mutexLocked (1 << 0)：锁是否被持有

mutexWoken (1 << 1)：是否有被唤醒的goroutine

mutexStarving (1 << 2)：是否处于饥饿模式

剩余29位表示：等待锁的goroutine数量

3. 核心工作流程

加锁(Lock)过程：

快速路径：CAS原子操作尝试直接获取锁

慢速路径：

自旋等待(最多4次)

计算期望状态

更新状态并可能进入饥饿模式

通过信号量阻塞当前goroutine

解锁(Unlock)过程：

原子操作释放锁标志

如果有等待者：

正常模式：唤醒一个等待者

饥饿模式：直接将锁交给队列头部的等待者

4. 两种模式

正常模式：竞争锁的goroutine先自旋尝试，后加入队列

饥饿模式：当某个goroutine等待超过1ms后触发，锁直接交给等待队列

5. 关键优化点

自旋尝试减少上下文切换

饥饿模式防止长尾延迟

内存对齐减少false sharing

二、sync.RWMutex 读写锁

1. 底层数据结构

go

type RWMutex struct {

w Mutex // 互斥锁(用于写操作)

writerSem uint32 // 写等待信号量

readerSem uint32 // 读等待信号量

readerCount int32 // 当前读操作数量(或负数表示有写等待)

readerWait int32 // 写操作需要等待的读操作数量

}

2. 核心工作原理

读锁(RLock)：

1、原子增加readerCount

2、如果readerCount < 0(表示有写等待)：

阻塞在读信号量上

读解锁(RUnlock)：

1、原子减少readerCount

2、如果readerCount < 0(表示有写等待)：

减少readerWait

如果readerWait == 0，唤醒写等待者

写锁(Lock)：

1、获取互斥锁w

2、原子减少readerCount(使其为负，标记有写等待)

3、等待现有读操作完成(readerWait计数)

4、如果有其他写操作，阻塞在写信号量上

写解锁(Unlock)：

1、恢复readerCount为正数

2、唤醒所有阻塞的读操作

3、释放互斥锁w

3. 关键特性

写优先：当有写等待时，新读操作会被阻塞

递归禁止：同一个goroutine不能连续获取读锁

内存屏障：保证操作的内存可见性

三、实现对比

特性 Mutex RWMutex

适用场景 独占访问 读多写少

底层实现 状态位+信号量 Mutex+双信号量+计数器

性能特点 写操作高效 并发读性能高

内存占用 8字节 24字节

特殊模式 饥饿模式 写优先模式

四、底层源码关键点

Mutex关键函数：

sync\_runtime\_canSpin：判断能否自旋

sync\_runtime\_doSpin：执行自旋

runtime\_SemacquireMutex：获取信号量

runtime\_Semrelease：释放信号量

RWMutex关键机制：

readerCount双重用途：

0：当前读锁数量

<0：有写锁等待(值为-r-waitCount)

写锁必须等待所有先前的读锁释放

五、使用注意事项

Mutex：

不可重入(同一goroutine重复Lock会导致死锁)

解锁未锁定的Mutex会panic

适合保护简短临界区

RWMutex：

写锁会阻塞新的读锁

读锁升级写锁会导致死锁

适合读多写少且读操作耗时的场景

这两种锁的实现充分考虑了现代多核CPU的特性，通过自旋、状态压缩和模式切换等优化，在保证正确性的同时提供了高性能的并发控制能力。

**sync.map**

1. 设计背景

标准map在并发场景下需要外部加锁，而sync.Map专为以下两种场景设计：

读多写少的并发访问模式

键值对一旦写入就很少修改，但需要频繁读取

1. 核心数据结构

type Map struct {

mu Mutex // 保护dirty的互斥锁

read atomic.Value // 只读数据(readOnly结构)

dirty map[interface{}]\*entry // 可写数据(加锁访问)

misses int // 读dirty的计数

}

type readOnly struct {

m map[interface{}]\*entry

amended bool // dirty是否包含m中没有的key

}

type entry struct {

p unsafe.Pointer // \*interface{}

}

三、底层工作原理

1. 双存储机制

read：原子访问的无锁读取区

dirty：需要加锁访问的写入区

2. 读写流程

读操作(Load)：

1、首先原子读取read

2、如果read中存在且有效，直接返回(无锁)

3、如果read中没有且amended=true，加锁后访问dirty

4、记录misses计数，触发潜在的数据提升

写操作(Store)：

1、先检查read中是否存在该key

存在且未被删除：尝试原子更新(无锁)

2、加锁后操作dirty

初始化dirty(如果需要)

更新或添加新条目

3、更新read的amended标记

删除操作(Delete)：

1、先尝试从read中标记删除(原子操作)

2、如果read中没有，加锁操作dirty

四、锁优化策略

1. 读写分离优化

读优先：绝大多数读操作无需加锁

延迟加锁：只有read miss时才需要获取锁

2. 原子操作替代锁

read的访问通过atomic.Value实现原子读

entry值的更新通过atomic.CompareAndSwapPointer

3. 动态数据迁移

misses计数触发：当read miss次数超过dirty大小时，将dirty提升为read

延迟初始化：dirty在第一次需要时才从read拷贝创建

4、标记删除技术

func (e \*entry) delete() (hadValue bool) {

for {

p := atomic.LoadPointer(&e.p)

if p == nil || p == expunged {

return false

}

if atomic.CompareAndSwapPointer(&e.p, p, nil) {

return true

}

}

}

使用nil和expunged特殊标记表示删除状态

避免立即操作dirty带来的锁竞争

五、性能优化点

无锁读路径：read的原子访问避免锁竞争

写合并：连续写入同一key只触发一次dirty操作

空间换时间：维护两份数据减少锁持有时间

懒惰处理：dirty的初始化和提升按需进行

七、使用注意事项

不适合频繁写入：大量写入会导致性能下降

类型安全：需要自行保证类型安全

Range操作：不保证遍历时并发的修改可见性

内存开销：长期存储大量不使用的key会导致内存浪费

**sync.pool**

sync.Pool类型可以被称为临时对象池，它的值可以被用来存储临时的对象。sync.Pool类型属于结构体类型，它的值在被真正使用之后，就不应该再被复制了。

这里的“临时对象”的意思是：不需要持久使用的某一类值。这类值对于程序来说可有可无，但如果有的话会明显更好。它们的创建和销毁可以在任何时候发生，并且完全不会影响到程序的功能。

sync.Pool类型只有两个方法——Put和Get。Put用于在当前的池中存放临时对象，它接受一个interface{}类型的参数；而Get则被用于从当前的池中获取临时对象，它会返回一个interface{}类型的值。

更具体地说，这个类型的Get方法可能会从当前的池中删除掉任何一个值，然后把这个值作为结果返回。如果此时当前的池中没有任何值，那么这个方法就会使用当前池的New字段创建一个新值，并直接将其返回。

sync.Pool类型的New字段代表着创建临时对象的函数。它的类型是没有参数但有唯一结果的函数类型，即：func() interface{}。

在临时对象池的内部，有一个多层的数据结构支撑着对临时对象的存储。它的顶层是本地池列表，其中包含了与某个P对应的那些本地池，并且其长度与P的数量总是相同的。

在每个本地池中，都包含一个私有的临时对象和一个共享的临时对象列表。前者只能被其对应的P所关联的那个goroutine中的代码访问到，而后者却没有这个约束。从另一个角度讲，前者用于临时对象的快速存取，而后者则用于临时对象的池内共享。

正因为有了这样的数据结构，临时对象池才能够有效地分散存储压力和性能压力。同时，又因为临时对象池的Get方法对这个数据结构的妙用，才使得其中的临时对象能够被高效地利用。比如，该方法有时候会从其他的本地池的共享临时对象列表中，“偷取”一个临时对象。

这样的内部结构和存取方式，让临时对象池成为了一个特点鲜明的同步工具。它存储的临时对象都应该是拥有较长生命周期的值，并且，这些值不应该被某个goroutine中的代码长期的持有和使用。

因此，临时对象池非常适合用作针对某种数据的缓存。从某种角度讲，临时对象池可以帮助程序实现可伸缩性，这也正是它的最大价值。

Go语言sync.Pool底层核心原理与优化

## 一、核心设计目标

sync.Pool设计用于\*\*减少GC压力\*\*和\*\*提高对象复用率\*\*，特别适合存储临时对象池的场景。其核心思想是：

1. 无锁获取（线程安全）

2. 自动清理（GC感知）

3. 对象复用（减少分配）

## 二、底层数据结构

```go

type Pool struct {

noCopy noCopy

local unsafe.Pointer // 本地P的poolLocal数组指针

localSize uintptr // 本地数组大小

victim unsafe.Pointer // 上一轮GC幸存的对象

victimSize uintptr // 幸存对象数量

New func() interface{} // 创建新对象的函数

}

// 每个P对应的本地池

type poolLocal struct {

poolLocalInternal

// 缓存行填充，避免false sharing

pad [128 - unsafe.Sizeof(poolLocalInternal{})%128]byte

}

type poolLocalInternal struct {

private interface{} // 只能被当前P使用

shared poolChain // 本地P可共享的双端队列

}

```

## 三、核心工作原理

### 1. 多级存储结构

- \*\*private\*\*（每个P独占）：快速存取路径，完全无锁

- \*\*shared\*\*（每个P共享）：无锁队列（CAS操作）

- \*\*victim cache\*\*：GC幸存对象缓存

### 2. 对象获取流程（Get）

1. 先尝试从当前P的private获取（无锁）

2. 如果private为空，从当前P的shared队列头部获取（CAS乐观锁）

3. 如果当前P没有，则从其他P的shared队列尾部"偷取"（CAS乐观锁）

4. 如果所有池都为空，调用New函数创建新对象

### 3. 对象放回流程（Put）

1. 优先放回当前P的private（如果private为空）

2. 如果private非空，放入当前P的shared队列头部（CAS操作）

### 4. GC处理机制

1. 每次GC时会清空所有primary缓存

2. 将primary缓存转移到victim缓存

3. 只有经过两轮GC都存活的对象才会被真正回收

## 四、线程安全实现

### 1. 无锁设计优化

- \*\*private字段\*\*：每个P独占，完全无竞争

- \*\*shared队列\*\*：使用CAS实现的无锁队列（poolChain）

- \*\*victim缓存\*\*：只在GC时由系统线程操作

### 2. 消除false sharing

- 通过`pad`字段填充cache line（128字节对齐）

- 确保不同P的poolLocal不在同一缓存行

### 3. 工作窃取(Work Stealing)机制

- 当本地P无对象时，可以从其他P的shared队列尾部窃取

- 减少锁竞争，提高并发性能

## 五、性能优化点

1. \*\*层级缓存\*\*：

- private → shared → victim → new

- 90%以上的操作只需要访问private（完全无锁）

2. \*\*GC友好设计\*\*：

- 对象最多存活两轮GC周期

- 自动调整缓存大小，避免内存泄漏

3. \*\*无锁队列\*\*：

- shared使用atomic实现的无锁双向队列

- 头部操作无竞争（当前P专用）

- 尾部窃取使用CAS

4. \*\*缓存行优化\*\*：

- 每个poolLocal占满整个缓存行

- 避免多核CPU的伪共享问题

## 六、与普通对象分配对比

| 特性 | sync.Pool | 直接new/分配 |

|---------------|-------------------------|-----------------------|

| 分配开销 | 可能为零（复用） | 每次都要分配 |

| GC压力 | 显著降低 | 正常压力 |

| 线程安全 | 是 | 否 |

| 对象生命周期 | 自动管理（两轮GC） | 由引用计数决定 |

| 适用场景 | 高频创建/销毁的临时对象 | 长期存活的对象 |

## 七、使用注意事项

1. \*\*对象清空\*\*：从Pool获取的对象可能是复用的，必须重置状态

2. \*\*不保证保留\*\*：Pool中的对象随时可能被GC回收

3. \*\*适合场景\*\*：对象分配成本高且生命周期短

4. \*\*大小限制\*\*：没有内置大小限制，可能占用较多内存

5. \*\*类型安全\*\*：需要自行保证存储/取出类型一致

## 八、底层关键代码

```go

// runtime包中的实际实现

func poolGet(p \*Pool) interface{} {

// 获取当前P的poolLocal

l := p.pin()

// 先尝试private

if x := l.private; x != nil {

l.private = nil

return x

}

// 再尝试shared队列

if x := l.shared.popHead(); x != nil {

return x

}

// 最后尝试从其他P窃取

return p.getSlow()

}

func (p \*Pool) put(x interface{}) {

if x == nil {

return

}

l := p.pin()

if l.private == nil {

l.private = x

} else {

l.shared.pushHead(x)

}

}

```

sync.Pool通过这种精巧的无锁设计和多级缓存机制，在保证线程安全的同时，为特定场景提供了极高性能的对象复用能力，是Go语言高性能并发编程的重要组件之一。

sync.waitGroup

场景：让其他的goroutine在运行结束之前，都向这个通道发送一个元素值，并且，让主goroutine在最后从这个通道中接收元素值，接收的次数需要与其他的goroutine的数量相同。

比通道更加适合实现这种一对多的goroutine协作流程。

WaitGroup类型拥有三个指针方法：Add、Done和Wait。你可以想象该类型中有一个计数器，它的默认值是0。我们可以通过调用该类型值的Add方法来增加，或者减少这个计数器的值。

一般情况下，我会用这个方法来记录需要等待的goroutine的数量。相对应的，这个类型的Done方法，用于对其所属值中计数器的值进行减一操作。我们可以在需要等待的goroutine中，通过defer语句调用它。

而此类型的Wait方法的功能是，阻塞当前的goroutine，直到其所属值中的计数器归零。如果在该方法被调用的时候，那个计数器的值就是0，那么它将不会做任何事情。

注意事项：

不要把增加其计数器值的操作和调用其Wait方法的代码，放在不同的goroutine中执行。换句话说，要杜绝对同一个WaitGroup值的两种操作的并发执行。

sync.once

Once类型的Do方法只接受一个参数，这个参数的类型必须是func()，即：无参数声明和结果声明的函数。

该方法的功能并不是对每一种参数函数都只执行一次，而是只执行“首次被调用时传入的”那个函数，并且之后不会再执行任何参数函数。

所以，如果你有多个只需要执行一次的函数，那么就应该为它们中的每一个都分配一个sync.Once类型的值（以下简称Once值）。

Once类型中还有一个名叫done的uint32类型的字段。它的作用是记录其所属值的Do方法被调用的次数。不过，该字段的值只可能是0或者1。一旦Do方法的首次调用完成，它的值就会从0变为1。

既然done字段的值不是0就是1，那为什么还要使用需要四个字节的uint32类型呢？

原因很简单，因为对它的操作必须是“原子”的。Do方法在一开始就会通过调用atomic.LoadUint32函数来获取该字段的值，并且一旦发现该值为1，就会直接返回。这也初步保证了“Do方法，只会执行首次被调用时传入的函数”。

不过，单凭这样一个判断的保证是不够的。因为，如果有两个goroutine都调用了同一个新的Once值的Do方法，并且几乎同时执行到了其中的这个条件判断代码，那么它们就都会因判断结果为false，而继续执行Do方法中剩余的代码。

在这个条件判断之后，Do方法会立即锁定其所属值中的那个sync.Mutex类型的字段m。然后，它会在临界区中再次检查done字段的值，并且仅在条件满足时，才会去调用参数函数，以及用原子操作把done的值变为1。

Do方法在功能方面的两个特点。

第一个特点，由于Do方法只会在参数函数执行结束之后把done字段的值变为1，因此，如果参数函数的执行需要很长时间或者根本就不会结束（比如执行一些守护任务），那么就有可能会导致相关goroutine的同时阻塞。

例如，有多个goroutine并发地调用了同一个Once值的Do方法，并且传入的函数都会一直执行而不结束。那么，这些goroutine就都会因调用了这个Do方法而阻塞。因为，除了那个抢先执行了参数函数的goroutine之外，其他的goroutine都会被阻塞在锁定该Once值的互斥锁m的那行代码上。

第二个特点，Do方法在参数函数执行结束后，对done字段的赋值用的是原子操作，并且，这一操作是被挂在defer语句中的。因此，不论参数函数的执行会以怎样的方式结束，done字段的值都会变为1。

也就是说，即使这个参数函数没有执行成功（比如引发了一个panic），我们也无法使用同一个Once值重新执行它了。所以，如果你需要为参数函数的执行设定重试机制，那么就要考虑Once值的适时替换问题。

在很多时候，我们需要依据Do方法的这两个特点来设计与之相关的流程，以避免不必要的程序阻塞和功能缺失。

# Go语言sync.WaitGroup与sync.Once核心原理与对比

## 一、sync.WaitGroup

### 1. 核心原理

\*\*数据结构\*\*：

```go

type WaitGroup struct {

noCopy noCopy // 防止复制的标记

state1 [3]uint32 // 存储计数器和waiter计数

}

```

\*\*底层实现\*\*：

- 使用32位原子操作维护两个计数器：

- \*\*counter\*\*：当前未完成的任务数（高32位）

- \*\*waiter\*\*：等待的goroutine数（低32位）

- 通过信号量（semaphore）实现阻塞和唤醒

\*\*关键方法\*\*：

- `Add(delta int)`：原子操作修改计数器

- `Done()`：等价于`Add(-1)`

- `Wait()`：检查计数器，若不为0则阻塞

### 2. 适用场景

- \*\*并行任务等待\*\*：等待一组goroutine完成工作

- \*\*批量任务同步\*\*：主goroutine等待所有子任务完成

- \*\*服务优雅退出\*\*：等待所有处理中的请求完成

### 3. 使用示例

```go

var wg sync.WaitGroup

for i := 0; i < 10; i++ {

wg.Add(1)

go func() {

defer wg.Done()

// 执行任务

}()

}

wg.Wait() // 等待所有goroutine完成

```

## 二、sync.Once

### 1. 核心原理

\*\*数据结构\*\*：

```go

type Once struct {

done uint32 // 执行标志

m Mutex // 互斥锁

}

```

\*\*底层实现\*\*：

- 使用`done`标志位（原子访问）和互斥锁双重检查

- 通过`atomic.CompareAndSwap`实现无竞争快速路径

\*\*关键方法\*\*：

- `Do(f func())`：

1. 原子加载done标志

2. 若未执行过，获取锁

3. \*\*双重检查\*\*done标志

4. 执行函数并原子存储done标志

### 2. 适用场景

- \*\*单例初始化\*\*：确保初始化代码只执行一次

- \*\*延迟加载\*\*：资源首次使用时初始化

- \*\*全局配置\*\*：保证配置只加载一次

### 3. 使用示例

```go

var (

config map[string]string

once sync.Once

)

func LoadConfig() map[string]string {

once.Do(func() {

// 只会执行一次

config = readConfig()

})

return config

}

```

## 三、核心区别对比

| 特性 | sync.WaitGroup | sync.Once |

|---------------|-----------------------------------|-------------------------------|

| \*\*设计目的\*\* | 多goroutine等待同步 | 单次初始化保证 |

| \*\*底层实现\*\* | 原子计数器+信号量 | 双重检查锁+原子标志 |

| \*\*线程安全\*\* | 完全线程安全 | 完全线程安全 |

| \*\*执行次数\*\* | Add/Done可多次调用 | Do方法只执行一次 |

| \*\*阻塞行为\*\* | Wait()会阻塞直到计数器归零 | 首次Do()会阻塞，后续立即返回 |

| \*\*内存占用\*\* | 12字节（含noCopy） | 8字节（含noCopy） |

| \*\*典型场景\*\* | 并行任务等待 | 单例初始化 |

## 四、实现细节对比

### 1. 内存布局

- \*\*WaitGroup\*\*：

- 使用`state1 [3]uint32`巧妙处理32/64位对齐

- 在64位系统上，counter和waiter可以原子操作

- \*\*Once\*\*：

- 简单的`done`标志位+互斥锁组合

- 内存紧凑（仅占8字节）

### 2. 性能关键点

- \*\*WaitGroup\*\*：

- Add/Done使用原子操作（无锁）

- Wait仅在counter>0时才会阻塞

- \*\*Once\*\*：

- 快速路径：原子加载done标志（无锁）

- 慢速路径：双重检查锁模式

### 3. 特殊注意事项

- \*\*WaitGroup\*\*：

- 计数器不能为负（会panic）

- Add调用必须在Wait之前

- \*\*Once\*\*：

- 即使f panic，done标志也会被设置

- 不能重置状态（一旦执行永久标记）

## 五、选择建议

1. \*\*使用WaitGroup当\*\*：

- 需要等待多个并行操作完成

- 任务数量动态可变

- 需要阻塞等待结果

2. \*\*使用Once当\*\*：

- 需要确保某段代码只执行一次

- 初始化成本高的资源

- 需要线程安全的延迟加载

## 六、底层源码关键片段

\*\*WaitGroup.Add\*\*实现节选：

```go

func (wg \*WaitGroup) Add(delta int) {

state := atomic.AddUint64(statep, uint64(delta)<<32

if int32(state) < 0 {

panic("sync: negative WaitGroup counter")

}

if state>>32 == 0 { // counter归零

for i := 0; i < int(v); i++ {

runtime\_Semrelease(semap, false)

}

}

}

```

\*\*Once.Do\*\*实现节选：

```go

func (o \*Once) Do(f func()) {

if atomic.LoadUint32(&o.done) == 0 {

o.doSlow(f)

}

}

func (o \*Once) doSlow(f func()) {

o.m.Lock()

defer o.m.Unlock()

if o.done == 0 {

defer atomic.StoreUint32(&o.done, 1)

f()

}

}

```

两种同步原语虽然简单，但其精巧的实现体现了Go并发模型的设计哲学：通过最少的同步机制提供最大的并发性能。理解它们的底层原理有助于在适当场景选择最佳工具。

**singleflight**

一、Singleflight是什么？

Singleflight是Go官方提供的一个并发原语（位于golang.org/x/sync/singleflight），用于抑制重复的函数调用。它的核心功能是当多个goroutine同时调用同一个函数时，Singleflight能确保对于相同的调用键（key），只有一个实际执行会进行，其他并发调用会等待并共享相同的结果。

二、核心适用场景

1、缓存击穿防护：

当缓存失效时，防止大量请求同时穿透到数据库

典型案例：热点key突然失效导致的雪崩效应

2、重复计算抑制：

避免对计算密集型函数的重复计算

例如：复杂的配置解析或模板编译

3、下游服务保护：

防止对相同数据的重复API调用

特别适用于响应较慢的外部服务接口

4、全局状态同步：

多个goroutine需要获取相同初始化状态时

例如：应用启动时的全局配置加载

1. 底层实现原理
2. 核心数据结构

type Group struct {

mu sync.Mutex // 保护callMap的互斥锁

m map[string]\*call // 正在进行的调用记录

}

type call struct {

wg sync.WaitGroup // 等待此调用的goroutine

val interface{} // 函数返回值

err error // 函数返回错误

forgotten bool // 是否已丢弃结果

dups int // 重复调用计数

chans []chan<- Result // 用于DoChan的通道

}

1. 关键工作流程（以Do方法为例）
2. 加锁检查：

g.mu.Lock()

if g.m == nil {

g.m = make(map[string]\*call)

}

if c, ok := g.m[key]; ok {

c.dups++

g.mu.Unlock()

c.wg.Wait() // 等待已有调用完成

return c.val, c.err

}

1. 创建调用记录

c := new(call)

c.wg.Add(1)

g.m[key] = c

g.mu.Unlock()

3、执行实际函数

c.val, c.err = fn()

c.wg.Done()

4、清理阶段

g.mu.Lock()

if !c.forgotten {

delete(g.m, key)

}

g.mu.Unlock()

3. 三种调用方式

方法 返回值 特点

Do (val, err) 同步阻塞等待

DoChan <-chan Result 异步返回channel接收结果

Forget - 主动丢弃已缓存的结果

四、性能优化特点

锁粒度优化：

使用细粒度锁（仅保护map操作）

实际函数执行时不持有锁

内存高效：

调用结束后立即删除记录

通过Forget可提前释放资源

并发控制：

使用WaitGroup实现等待通知

避免重复计算的同时减少goroutine阻塞

灵活的结果传递：

支持同步和异步两种结果获取方式

错误结果也会被共享

六、与类似机制对比

特性 Singleflight sync.Once Mutex+缓存

去重粒度 按key区分 全局唯一 需自行实现

动态性 每次调用独立处理 只执行一次 可灵活控制

错误处理 共享错误结果 不处理错误 需自行处理

适用场景 动态key的重复抑制 初始化场景 通用同步场景

七、注意事项

函数副作用：确保被包装的函数可以安全地多次执行（虽然实际只执行一次）

超时控制：长期阻塞的函数会影响所有等待者，需结合context使用

内存泄漏：对于不常用的key应及时Forget

结果类型：Do返回interface{}需要类型断言

Singleflight通过这种精巧的设计，在特定场景下能显著降低系统负载，是构建高并发服务的有效工具之一。理解其原理有助于在缓存系统、API网关等中间件中合理应用。

典型使用事例

var group singleflight.Group

func GetData(key string) (string, error) {

// 先查缓存

if val, ok := cache.Get(key); ok {

return val, nil

}

// 使用Singleflight查询DB

val, err, \_ := group.Do(key, func() (interface{}, error) {

return db.Query(key) // 实际数据库查询

})

if err == nil {

cache.Set(key, val)

}

return val.(string), err

}

异步版本

func AsyncGetData(key string) <-chan singleflight.Result {

ch := group.DoChan(key, func() (interface{}, error) {

return db.Query(key)

})

return ch

}