GO语言学习笔记

变量：一段或多段用来存储数据的内存，类型决定了变量内存的长度和存储格式。简短模式并不总是重新定义变量，可能是部分退化的赋值操作（P19），前提：最少有一个新变量被定义且必须是同一作用域，err变量可以重复使用。

常量：不使用也不会引起编译错误，在常量组中如不指定类型和初始化值，则与上一行非空常量右值相同（P25）。

枚举：可借助iota标识符实现一组自增常量来实现枚举类型（P25）。

常量与变量的区别：变量在运行期分配存储内存，常量通常会被编译器在预处理阶段直接展开，作为指令数据使用。

引用类型：slice、map、channel，使用make函数创建。

未命名类型：数组、切片、字典、通道等类型与具体元素类型或长度属性有关，可用type为其提供具体名称（P35）。

25个关键字：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| break | default | func | interface | select |
| case | defer | go | map | struct |
| chan | else | goto | package | switch |
| const | fallthrough | if | range | type |
| continue | for | import | return | var |

指针：是一个实体，用来保存地址的整型变量（P42）。在 Go 语言中，通过指向结构体的指针获取结构体中的值时，可以使用点操作符直接访问字段，而无需显式地使用 \* 运算符进行解引用。这是因为 Go 语言在访问指针指向的结构体字段时，会自动解引用指针并访问字段。

函数：结构化变成的最小模块单元，函数只能判断其是否为nil（P59）。形参是指函数定义中的参数，实参则是函数调用时所传递的参数。

变参：本质上是一个切片。只能接受一个到多个同类型参数，且必须放在列表尾部（P66）。

闭包：在其词法上下文中引用了自由变量的函数，或者是函数和其引用的环境的组合体（P71）。

字符串：不可变字节序列，本身是一个复合结构，可以使用len(str)得到字符串的长度（P86）。其默认值为””，允许以索引号访问字节数组，但不能获取元素地址。以切片语法[起始:结束]返回字符串时，其内部依旧指向原字节数组（P87）。使用for循环遍历字符串分为byte和rune两种，其中byte是for i:=0;i<len(s);i++ s[i]。rune则是for I,j := range s。byte是uint8别名通常表示ASCII字符，rune是int32别名，表示Unicode包含ASCII，可以表示中文，使用单引号默认为rune。三者可以直接转换，string(s) byte(s) rune(s)。

数组：长度是类型组成部分，元素类型相同，但长度不同的数组不属于同一类型（P95）。c:=[...]int{5,3:10}结果为[5 0 0 10]。内置len和cap都返回第一维度长度，支持== ！=操作。

指针：指针数组指元素为指针类型的数组，数组指针是取数组变量的地址（P98）。a为数组，p:=&a获得数组a的指针，但p[1]是获取数组第二个元素的值而不是地址。Go数组是值类型，赋值和传参操作都会复制整个数组数据，可以使用指针或切片避免数据复制。

切片：内部通过指针引用底层数组，设定相关属性将数据读写操作限定在指定区域内（P100）。不支持比较操作，可获取元素地址，但不能像数组那样直接用指针访问元素内容，想要获取则需要p:=&s (\*p)[0]或则p1:=&s[0] \*p1+=100。切片只是很小的机构体对象，用来代替数组传参，避免复制开销。make函数允许在运行期动态指定数组长度，绕开了数组类型必须使用编译期常量的限制。

reslice:将切片看做[cap]slice数据源，据此创建新切片对象。不能超出cap，但不受len限制（P105）。新建切片对象依旧指向原底层数组，修改对所有关联切片可见。P106利用reslice实现栈。

append：向切片尾部添加数据，返回新的切片对象。append并未向原数组写入部分数据（P108）。

copy：在两个切片对象间复制数据，允许指向同一底层数组，允许目标区间重叠，最终所复制长度以较短的切片长度为准（P109）。

字典：（哈希表）在创建字典时，不需要指定初始容量，因为字典会根据实际的键值对动态地进行内存分配和管理，是引用类型，使用make函数或初始化表达语句来创建。作为无序键值对集合，字典要求key必须是支持相等运算符（==、！=）的数据，比如数字、字符串、指针、数组、结构体，以及对应接口类型。（P110）delete删除键值对时，不存在的键值返回该值类型的零值不出错。对字典进行迭代，每次返回的键值次序都不相同。len返回当前键值对数量，当使用结构体作为字典的值时，该值会被复制到字典中而不是直接引用，如果想要修改字典中的结构体值，先将其取出修改然后再将修改的值赋值回字典，不能使用m[1].age+=1。正确做法为u:=m[1] u.age+=1 m[1]=u。当value是指针类型时可以使用m[1].age++.

安全：在迭代期间删除或新增键值是安全的。

性能：字典对象本身就是指针包装，传参时无须再次取地址。在创建时预先准备足够空间有助于提升性能，减少扩张时的内存分配和重新哈希操作。

结构体：将多个不同类型命名字段序列打包成一个复合类型，字段名必须唯一（P118）。在对struct初始化时，可以使用键值对方式赋值（一部分值）。只有所有字段类型全部支持时才能做相等操作。可以使用指针直接操作结构字段，例如type user struct{name string age int} p:=&user{name:”Tom”,age:20} p.name=”Mary” p.age++。

空结构：（struct{}）指没有字段的结构体类型（P120），无论是自身还是作为数组元素类型（var b [10]struct{}创建长度为10的数组，每个元素都是空结构体struct{}{}，与var b [10]struct{}{}没有区别），长度都为零。空结构可作为通道元素类型，用于事件通知。

匿名字段：没有名字仅有类型的字段，也被称为嵌入字段或嵌入类型（P122）。

字段标签：不是注释，是用来对字段进行描述的元数据（P125），不属于数据成员，是类型的组成部分，例如type user struct{name string `昵称`}。

方法：方法是与对象实例绑定的特殊函数，用于维护和展示对象的自身状态。对象是内敛的，每个实例都有各自不同的独立特征，以属性和方法来暴露对外通信接口。普通函数则专注于算法流程，通过接受参数来完成特定逻辑运算，并返回最终结果。方法是有关联状态的，函数通常没有。可以为当前包以及除接口和指针以外的任何类型定义方法。方法可以看作特殊的函数，那么接受类型可以是基础类型和指针，这关系到调用时对象实例是否被复制。不能使用多级指针调用方法。

调用对象实例是否被复制只看方法接受时是指针还是普通类型，也就是看方法type (p Person) 还是type(p \*Person)。即便是p=&a,p.value() <-前面的即（p Person）那么仍属于是复制。因此对于receiver的选择：要修改实例状态用\*T，无需修改状态的小对象或固定值用T。大对象用\*T以减少复制成本。

匿名字段：（P133）

接口：代表一种调用契约，是多个方法声明的集合。只要目标类型方法集内包含接口声明的全部方法，就被视为实现了该接口，无需做显示声明（P141）。如果接口没有任何方法声明，就是一个空接口（interface{}），可被赋值为任何类型的对象。接口变量默认值为nil，可以嵌入其他接口，目标类型方法集中必须拥有包含嵌入接口方法在内的全部方法才算实现该接口。

只有指针类型的接受者方法才能实现对应接口中的方法，如果接口中的方法集合中包含了指针类型的接收者方法，那么实现该接口的类型必须是指针类型，而不是值类型。例子:type tester interface{test() string()} type data struct{} func(\*data) test(){} func(data) string() var d data var t tester = &d(重点在这，因为实现方法中存在\*data，所以必须使用&d)。(\*data) 类型的方法集合包含了 tester 接口中的方法，也就是说 (\*data) 类型实现了 tester 接口。因此，为了使 data 类型实现 tester 接口，你需要使用 &d 将 data 类型的值取地址，得到 \*data 类型的指针。

执行机制：接口使用itab的结构存储运行期所需的相关类型信息，将对象赋值给接口变量时，会复制该对象，解决方法：将对象指针赋值给接口，那么接口内存储的就是 指针的复制品。例子：type data struct{x int} d :=data{100} var t interface{}=&d t.(\*data).x=200, 接口类型的变量只能访问其所包含的方法，而不能直接访问其包含的字段如果。我们想要访问接口类型所包含的具体结构体的字段，需要使用类型断言将其转换为具体的结构体类型，然后再访问字段。所以，在这个代码中，我们需要使用 t.(\*data).x，通过类型断言将接口类型 t 转换为 \*data 类型的指针，然后访问 \*data 类型的指针的字段 x。

类型转换：可将接口变量还原为原始类型，或用来判断是否实现了某个更具体的接口类型。

技巧：让编译器检查，确保类型实现了指定接口（P151）。

并发：逻辑上具备同时处理多个任务的能力（P153）。并行：物理上在同一时刻执行多个并发任务。多线程或多进程是并行的基本条件，但单线程也可用协程做到并发。通常情况下用多进程来实现分布式和负载平衡，减轻单进程垃圾回收压力；用多线程（LWP）抢夺更多的处理器资源；用协程来提高处理器时间片利用率。

只需在函数调用前添加go关键字即可创建并发任务，注意是函数调用，所以必须提供相应的参数。关键字go并非执行并发操作，而是创建一个并发任务单元。新建任务被放置在系统队列中，等待调度器安排合适系统线程去获取执行权。当前流程不会阻塞，不会等待该任务启动，且运行时也不保证并发任务的执行次序。每个任务单元除保存函数指针，调用参数外，还会分配执行所需的栈内存空间。

Wait:进程退出时不会等待并发任务结束，可用通道（channel）阻塞，然后发出退出信号（P155）。如果要等待多个任务结束，使用sync.WaitGroup。通过设定计数器，让每个goroutine在退出前递减，直至归零时解除阻塞（P155）。

goroutine任务无法设置优先级，无法获取编号。暂停，释放线程去执行其他任务。当前任务被放回队列，等待下次调度时恢复执行（P160）。

通道：是显式的，要求操作双方必须知道数据类型和具体通道，并不关心另一端操作者身份和数量。可如果另一端未准备妥当，或消息未能及时处理时，会阻塞当前端（P163）。从底层实现上讲，通道只是一个队列。同步模式下，发送和接受双方配对，然后直接复制数据给对方。如匹配失败，则置入等待队列，直到另一方出现后才被唤醒。异步模式抢夺的规则是数据缓冲槽。发送方要求有空槽可供写入，而接受方则要求有缓冲数据可读。需求不符时，同样加入等待队列，直到有另一方写入数据或腾出空槽后被唤醒。

同步模式必须有配对操作的goroutine出现，否则会一直阻塞。而异步模式在缓冲区未满或数据未读完前，不会阻塞。多数时候，异步通道有助于提升性能，减少排队阻塞。缓冲区大小仅是内部属性，不属于类型组成部分。通道变量本身就是指针，可用相等操作符判断是否为同一对象或nil。内置函数cap和len返回缓冲区大小和当前已缓冲数量；而对同步通道则都返回0，据此可判断通道是同步还是异步。

收发：ch:=make(chan int) ch<-1 x:=<-c，还可以用ok-idom或range模式处理数据。例如：x,ok:=<-ch if !ok{//判断通道是否关闭}。对于循环接受数据，range模式更简洁，例如:for x:=range c{println(x)}，及时用close函数关闭通道引发结束通知，否则可能会导致死锁。向已关闭通道发送数据，引发panic。从已关闭接受数据，返回已缓冲数据或零值。无论收发，nil通道都会阻塞。

单向：通道默认是双向的，并不区分发送和接收端。可以限制收发操作的方向来获得更严谨的操作逻辑。例如：c:=make(chan int) var send chan<- int = c var recv <-chan int = c，close无法用于接收端。

选择：如要同时处理多个通道，可选用select语句。它会随机选择一个可用通道做收发操作（P170）。即便是同一通道，也会随机选择case执行。当所有通道都不可用时，select会执行default语句。如此可避开select阻塞，可以用default处理一些默认逻辑，比如当前通道已满，生成新的缓存通道。

模式：通常使用工厂方法将goroutine和通道绑定（P176）。

性能：将发往通道的数据打包，减少传输次数，可有效提升性能。从实现上来说，通道队列依旧使用锁同步机制，单次获取更多数据，可改善因频繁加锁造成的性能问题（P180）。

资源泄露：通道可能引发goroutine leak，确切的说，是指goroutine处于发送或接受阻塞状态，但一直未被唤醒。垃圾回收并不收集此类资源，导致它们会在等待队列里长久休眠，形成资源泄露。

同步：通道并非取代锁，它们各自有不同的使用场景。通道倾向于解决逻辑层次的并发处理架构，而锁用来保护局部范围内的数据安全（P183）。标准库sync提供了互斥和读写锁，另有原子操作等。将Mutex作为匿名字段时，相关方法必须实现为pointer-receiver（指针接收器），否则会因复制导致锁机制失效。Mutex不支持递归锁，即便是在同一goroutine下也会导致死锁

工作空间：由src、bin、pkg三个目录组成，通常需要将空间路径添加到GOPATH环境变量列表中，以便相关工具能正常工作。包括子包在内的所有源码文件都保存在src目录下，至于bin和pkg主要影响go install/get命令，它们会将编译结果（可执行文件或静态库）安装到这两个目录下，以实现增量编译。

反射：可以在运行期探知对象的类型信息和内存结构，反射还是实现元编程的重要手段。反射操作所需的全部信息都源自接口变量，接口变量除存储自身类型外，还会保存实际对象的类型数据（P200）。例子（获取变量的类型）：type X int var a X = 100 t:=reflect.TypeOf(a) t.Name() t.Kind()为X int。a := reflect.ArrayOf(10, reflect.TypeOf(byte(0)))该代码创建长度为10的字节数组的反射类型。m := reflect.MapOf(reflect.TypeOf(""), reflect.TypeOf(0))该代码创建一个字符串到整数的映射的反射类型。运用上述两函数，可以在运行时动态地创建数组和映射的反射类型，对于反射和代码生成等场景有用。方法Elem返回指针、数组、切片、字典或通道的基类型，例如fmt.Println(reflect.TypeOf(map[string]int{}).Elem())结果为int。

值：和Type获取类型信息不同，Value专注于对象实例数据读写。接口变量会复制对象，且是unaddressable的，所以想要修改目标对象，就必须使用指针。例如：reflect.ValueOf(a)，relect.ValueOf(&a).Elem()

方法：（P210），按In列表准备好所需参数即可。

构建：反射库提供了内置函数make和new的对应操作，例如MakeFunc。可用它实现通用模版，适应不同数据类型（P212）。

性能：如对性能要求高，那么须谨慎使用反射。

单元测试：用来测试逻辑算法是否符合预期，承担这监控代码质量（P216）。

testing：测试代码必须放在当前包以“\_test.go”结尾的文件中，测试函数以Test为名称前缀，测试命令（go test）忽略以“\_”或“.”开头的测试文件，正常编译操作会忽略测试文件。标准库testing提供了专门类型T来控制测试结果和行为（相信见P217，例如Fail Failnow Log）。

t.Parallel()可以利用多核并行优势，缩短测试时间。当你运行多个测试函数时，测试框架会自动检测哪些测试函数可以并行执行，并尽可能地在多个 goroutine 中同时执行它们。常用测试阐述见P218，例如-args（命令行参数） -v（输出详细信息）等配合go test使用。

table driven：单元测试代码一样要写的简洁优雅，可以用一种类似数据表的模式来批量输入条件并以此对比结果（P219）。这种方式将测试数据和测试逻辑分离，更便于维护，使用t.Errorf()是为了让真个表全部测试完成，以便知道那个组条件出了问题。

test main:需要为测试用例提供初始化和清理操作，但testing没有setuo/teardown机制。解决方法是定义一个名为TestMain的函数，go test会改为执行该函数，而不再是具体的测试用例（P220）。

example：例代码最大用途不是测试，而是导入到GoDoc等工具生成的帮助文档中，它通过比对输出结果和内部output注释是否一致来判断是否成功。

性能测试：函数以Benchmark为前缀，同样报错在\*\_test.go文件里（P221）。测试工具默认不会执行性能测试，须使用bench参数，它通过逐步调用B.N值，反复执行测试函数，直到能获取准确的测量结果。某些耗时的目标，默认循环次数过少，去平均值不足以准确计量性能。可用benchtime设定最小测试时间来增加循环次数，以便返回更准确的结果。

timer：如果在测试函数中要执行一些额外操作，那么应该临时阻止计时器工作。例如：

b.ResetTimer() 重置 b.StopTimer() 暂停 b.StartTimer() 恢复。

memory：性能测试关心的不仅仅是执行时间，还包括堆上的内存分配，因为内存分配和垃圾回收的相关操作也应记入消耗成本。

代码覆盖率：度量测试自身完整性和有效性的一种手段。通过覆盖率值，可以分析出测试代码的编写质量。检测它是否提供了足够的测试条件，是否执行了足够的函数、语句、分支和代码行等，以此来量化测试本身，让白盒测试真正起到应有的质量保障作用（P224）。

性能监控：通过捕获或监控相关执行状态数据，就可定位引发问题的原因，从而有针对性改进算法（P226）。捕获方式有两种：首先，在测试时输出并保存相关数据，进行初期评估。其次，在运行阶段通过Web接口获取实时数据，分析一段时间内的健康状况。还可以用自定义计数器（expvar）提供更多与逻辑相关的参考数据。

工具：

|  |  |  |
| --- | --- | --- |
| go build | 每次都会重新编译出标准库外所有依赖包 | P231 |
| go install | 与build参数相同，编译结果安装到bin、pkg，支持增量编译 | P232 |
| go get | 将第三方包下载到GOPATH列表的第一工作空间 | P232 |
| go env | 显示全部或指定环境参数 | P233 |
| go clean | 清理工作目录，删除编译和安装遗留的目标文件 | P233 |

交叉编译：在一个平台上编译出其他平台所需的可执行文件。GO实现自举后，交叉编译变得方便，只需使用GOOS、GOARCH环境变量指定目标平台和架构就行（P235）。比如在Windows上使用Go语言交叉编译生成Linux可执行文件的步骤：set GOOS=linux set GOARCH= amd64 go build 将生成的文件拷贝到linux系统中，并在linux上运行它。

预处理：用go generate命令扫描源码文件，找出所有“go:generate”注释，提取其中的命令并执行（P239）。命令必须放在.go源文件中，以”//go:generate”开头，每个文件可以有多条generate命令。

内存分配（P255）

概述：每次从操作系统申请一大块内存（1MB，例如），以减少系统调用。将申请到的大块内存按照特定大小预先切成小块，构成链表。为对象分配内存时，只需从大小合适的链表提取一个小块即可。回收对象内存时，将该小块内存重新归还到原链表，以便复用。如闲置内存过多，则尝试归还部分内存给操作系统，降低整体开销。

内存块：分配器将其管理的内存块分为两种：1、span，由多个地址连续的页（page）组成的大块内存。2、object，将span按特定大小切分成多个小块，每个小块可存储一个对象。span面向内部管理，object面向对象分配。分配器初始化时，会构建对照表存储大小和规格的对应关系，包括用来切分的span页数。

管理组件：（P257）采用tcmalloc的架构。分配器由三种组件组成：1、cache，每个运行期工作线程都会绑定一个cache，用于无锁object分配。2、central，为所有cache提供切分好的后备span资源。3、heap，管理闲置span，需要时向操作系统申请新内存。

分配流程：1、计算待分配对象对应的规格（size class）。2、从cahce.alloc数组找到规格相同的span。3、从span.freelist链表提取可用object。4、如span.freelist为空，从central获取新span。5、如central.nonempty为空，从heap.free/freelarge获取，并切分成object链表。6、如heap没有大小合适的闲置span，向操作系统申请新内存块。

释放流程：1、将标记为可回收的object交还给所属的span.freelist。2、该span被放回central，可供任意cache重新获取使用。3、如span已收回全部object，则将其交还给heap，以便重新切分复用。4、定期扫描heap里长时间闲置的span，释放其占用的内存。

初始化：（P259）因为内存分配器和垃圾回收算法都很依赖连续地址，所以在初始化阶段，预先保留了很大的一段虚拟地址空间（保留地址空间，并不会分配内存）。该空间被划分成三部分：spans、bitmap、areana。用三个数组组成一个高性能内存管理结构。使用arena地址向操作系统申请内存，位图bitmap为每个对象提供4bit标记位，用以保存指针、GC（垃圾回收）等信息。创建span时，按页填充对应spans空间，在回收object时，只需将其他地址按页对齐后就可以找到所属span，分配器还用此访问相邻span做合并操作。区域相关属性保存在heap里。

初始化工作：1、创建对象规格大小对照表。2、计算相关区域大小，并尝试从某个指定位置开始保留地址空间。3、在heap里保存区域信息，包括起始位置和大小。4、初始化heap其他属性。

分配：（P265）为对象分配内存须区分是在栈上还是在堆上完成。编辑器有责任尽可能使用寄存器和栈存储对象，有助于提升性能，减少垃圾回收的压力。不是说用new函数就一定会分配在堆上，即便是相同的源码也会有不同的结果。在编辑器没有内联时，使用new就会在堆上分配，因为需要在两个栈间传递对象，因此会在堆上分配而不是返回一个失效栈帧里的数据。而当内联后，它实际上就变成main栈帧内的局部变量，无须去堆上操作。

内存分配：大对象直接从heap获取span。小对象从cache.alloc[sizeclass].freelist获取object。微小对象组合使用cache.tiny object。

资源不足时扩张：在heap里闲置的span不会被垃圾回收期关注，但central里的span却有可能正在被清理。从central里获取span时，优先取用已有资源。只有当现有资源都无法满足时，才去heap获取span，并重新切分成object链表。从heap获取span的算法核心是找到大小最合适的块，首先从页数相同的链表查找，找不到再从页数更多的链表提取，直至超大块或申请新块。如返回更大的span为避免浪费，会将多余部分切出来重新放回heap链表。同时来尝试合并相邻的闲置span空间，减少碎片。freelarge是简单链表，只能遍历选出最小、地址最靠前的块。

回收：（P279）内存回收的源头是垃圾清理操作。说回收而非释放，是因为整个内存分配器的核心是内存复用，不再使用的内存会被放回合适位置，等下次分配时再次使用，只有当空闲内存资源过多时，才会考虑释放。基于效率考虑，回收操作自然不会直接盯着单个对象，而是以span为基础单位。通过比对bitmap里的扫描标记，逐步将object收归原span，最终上交central或heap复用。

遍历span，将收集到的不可达object合并到freelist链表，如该span已收回全部object，那么就将这块完全自由的内存还给heap，以便后续复用。无论是向操作系统申请内存，还是清理回收内存，只要往heap里放span，都会尝试合并左右相邻的闲置span，以构成更大的自由块。

释放：（P283）在运行时入口函数main.main里，会专门启动一个监控任务sysmon，它每隔一段时间就会检查heap里的闲置内存块。遍历free、freelarge里的所有span，如闲置时间超过阈值，则释放其关联的物理内存。系统调用madvise告知操作系统某段内存暂时不使用，建议内核收回对物理内存。只是建议，是否回收有内核决定，如物理内存资源充足，该建议可能被忽略，以避免无所谓的损耗，而当再次使用该内存块时，会引发缺页异常，内核会自动重新关联物理内存页。分配器面对的是虚拟内存，所以在地址空间充足的情况下，根本无须放弃这段虚拟内存，无须收回mspan等管理对象，这也是arena能线性扩张的根本原因。

其他：（P285）从运行时的角度，整个进程内的对象可分为两类：1、从arena区域分配的用户对象；2、运行时自身运行和管理所需的对象，比如管理arena内存片段的maspan，提供无锁分配的mcache等。管理对象的声明周期不复杂，且类型和长度都相对固定，所以算法策略不复杂，相对较长的生命周期也不适合占用arena区域，否则会导致更多碎片。为此，运行时专门设计了FixAlloc固定分配器来管理对象分配内存。当运行时再初始化heap时，一共构建了4种固定分配器。分配算法优先从复用链表获取内存，只在获取失败或剩余空间不足时才获取新内存块。

垃圾回收：（P191）基本特征“非分代、非紧缩、写屏障、并发标记清理”。

三色标记和写屏障：这是让标记和用户代码并发的基本保障，基本原理：起初所有对象都是白色；扫描找出所有可达对象（能够通过程序中的引用链直接或间接访问到的对象，换句话说指程序执行期间仍然可被访问和使用的对象），标记为灰色，放入待处理队列；从队列提取灰色对象，将其引用对象标记为灰色放入队列，自身标记为黑色；写屏障监视对象内存修改，重新标色或放回队列。当完成全部扫描和标记工作后，剩余的不是白色就是黑色，分别代表对待回收和活跃对象，清理操作只需将白色对象内存回收即可。

控制器：控制器全程参与并发回收任务，记录相关状态数据，动态调整运行策略，影响并发标记单元的工作模式和数量，平衡CPU资源占用。当回收结束时，参与next\_gc回收阈值设置，调整垃圾回收触发频率。

辅助回收：某些时候，对象分配速度可能远快于后台标记，这回引发一系列恶果，比如堆恶性扩张，甚至让垃圾回收永远无法完成。此时，让用户代码线程参与后台回收标记就非常有必要，在为对象分配堆内存时，通过相关策略去执行一定限度的回收操作，平衡分配和回收操作，让进程处于良性状态。

初始化：（P293）重点是设置gcpercent和next\_gc阈值。

启动：在为对象分配堆内存后，mallocgc函数会检查垃圾回收触发条件，并依照相关状态启动或参与辅助回收。垃圾回收默认以全并发模式运行，但可以用环境变量或参数禁用并发标记和并发清理。GC goroutine一直循环，直到符合触发条件时被唤醒。

标记：（P300）并发标记分为两个步骤：1、扫描：遍历相关内存区域，依照指针标记找出灰色可达对象，加入队列。2、标记：将灰色对象从队列取出，将其引用对象标记为灰色，自身标记为黑色。

扫描：扫描函数gcscan\_m启动时，用户代码和MarkWorker都在运行。parfor是一个并行任务框架，其功能就是将任务平分，让多个线程各领一份并发任务。为保证整个任务组能尽快完成，他允许从执行较慢的线程偷取任务。所有扫描过程，最终通过scanblock比对bitmap区域信息找出合法指针，将其目标当做灰色可达对象添加到待处理队列。

标记：并发标记由多个MarkWorker goroutine共同完成，它们在回收任务开始前被绑定到P，然后进入休眠状态，直到被调度器唤醒。处理灰色对象时，无需知道其真实大小，只当作内存分配器提供的object块即可。按指针类型长度对齐，配合bitmap标记进行遍历，就可找出所有引用成员，将其作为灰色对象压入队列，当前对象自然成为黑色对象，从队列移除。

清理：所有未被标记的白色对象都不再被引用，可简单的将其内存回收。并发清理同样由一个专门的goroutine完成，它在runtime.main调用gcenable时被创建。并发清理本质是一个死循环，被唤醒后开始执行清理任务。通过遍历所有span对象，出发内存分配器的回收操作。任务完成后，再次休眠等待下次任务。

监控：（P314）模拟场景：服务重启，海量客户端重新接入，瞬间分配大量对象，这会将垃圾回收的触发条件next\_gc推到一个很大值，而当服务正常后，因活跃对象远小于该阈值，造成垃圾回收久久无法触发，服务进程内就会有大量白色对象无法被回收，造成隐形内存泄露，同样的情形也可能是因为某个算法短期内大量使用临时对象造成的。监控服务sysmon每隔2分钟就会检查一次垃圾回收状态，如超出2分钟未曾触发，俺就强制执行。

并发调度：（P326）1、语句go func()创建G。2、放入P本地队列。3、唤醒或新建M执行任务。4、进入调度循环schedule。5、竭力获取待执行G任务并执行。6、清理现场，重新进入调度循环。

Processor（简称P），其作用类似CPU核，用来控制可同时并发执行的任务数量。每个工作线程都必须绑定一个有效P才被允许执行任务，否则只能休眠，直到有空闲P时被唤醒。P还为线程提供执行资源，例如对象分配内存、本地任务队列等。线程独享所绑定的P资源，可在无锁状态下执行高效操作。

基本上，进程内的一切都以goroutine（简称G）方式运行，包括运行时相关服务，以及main.main入口函数。需要指出，G并非执行体，它仅仅保存并发任务状态，为任务执行提供所需栈内存空间。G任务创建后被放置在P本地队列或全局队列，等待工作线程调度执行。

实际执行体是系统线程（简称M），它和P绑定，以调度循环方式不停执行G并发任务。M通过修改寄存器，将执行栈指向G自带的栈内存，并在此空间内分配堆栈帧，执行任务函数，当需要中途切换时，只要将相关寄存器保存回G空间即可维持状态，任何都可据此恢复执行。线程仅负责执行，不在持有状态，这是并发任务跨线程调度，实现多路复用的根本所在。

初始化：调度器初始化函数schedinit，除去内存分配、垃圾回收等操作外，针对自身初始化有MaxMcount、GOMAXPROCS

任务：（P332）编译器会将go func(…)语句翻译成newproc调用，但这中间发生了什么。

后面包括通道、延迟、析构和缓存池，都是源码看不懂了，后面需要的时候再看吧。