Garbage Colletion垃圾回收

> 什么是垃圾回收：
>
> 对**没用的资源**进行**自动回收**
>
> 不回收容易引发**内存泄漏**问题

一、web项目性能监控

1、使用`ginpprof`

```
package main

import (
  "github.com/DeanThompson/ginpprof"
  "github.com/gin-gonic/gin"
)

func main() {
  router := gin.Default()
  ginpprof.Wrap(router)
  router.Run()
}

```

2、使用`pprof`

- 只需要在`main.go`中引入：`_"net/http/pprof"`(gin框架专用`"github.com/DeanThompson/ginpprof"`)
- 访问：`127.0.0.1:8080/debug/pprof`

```
/debug/pprof/profile:访问这个链接会自动进行 CPU_profiling,持续30s，并生成一个文件供下载。
/debug/pprof/block： Goroutine阻塞事件的记录。默认每发生一次阻塞事件时取样一次。
/debug/pprof/goroutines： 活跃Goroutine的信息的记录。仅在获取时取样一次。
/debug/pprof/heap： 堆内存分配情况的记录。默认每分配512K字节时取样一次。
/debug/pprof/mutex： 查看争用互斥锁的持有者。
/debug/pprof/threadcreate： 系统线程创建情况的记录。仅在获取时取样一次。
```

二、单个的go文件如果查看GC(Garbage Collection)

> **触发条件**：
>
> 1、超过**内存**大小的阈值
>
> ```
> 比如一次回收完毕之后，内存的使用量为5M，那么下次回收的实际则是内存分配达到10M的时候
> ```
>
> 2、达到**定时时间**
>
> ```
> 如果一直达不到内存大小，这个时候，GC就会被定时时间触发，默认2min触发一次，保证资源的回收
> ```

1、设置环境变量

```
set GODEBUG=gctrace=1
go run main.go
```

或者Mac下:

```
GODEBUG=gctrace=1 go run main.go
```

2、术语

> - `mark`：标记阶段。
> - `markTermination`：标记结束阶段。
> - `mutator assit`：辅助GC，是指在GC过程中mutator线程会并发运行，而mutator assist机制会协助GC做一部分的工作。
> - `heaplive`：在Go的内存管理中，span是内存页的基本单元，每页大小为8kb，同时Go会根据对象的大小不同而分配不同的页数的span，而heaplive就代表着所有的span的总大小。
> - `dedicated/fractional/idle`:在标记阶段会分为三种不同的mark worker模式，分别是dedicated、fractional和idle,它们代表着不同的专注程度，其中的dedica模式最专注，是完整的GC回收行为，fractional只会干部分的GC行为，idle最轻松。
> - `P`:指处理器

3、含义

`gc 2 @0.008s 3%: 0.002+0.84+0.006 ms clock, 0.018+0.068/1.1/0.029+0.055 ms cpu, 9->9->8 MB, 10 MB goal, 8 P`

- `gc 2`: GC执行次数的**编号**，每次叠加。 **第2次GC。**

- `@0.008s`:自程序启动之后到当前的具体秒数。 **当前程序启动后的0.008s。**
- `3%`:自程序启动以来在GC中花费的时间百分比。 **程序启动后到现在共花费3%的时间在GC上。**
- `0.002+0.84+0.006 ms clock`:GC的标记工作共使用的CPU时间占总CPU时间的百分比。
  - `0.002`表示单个P在`mark`阶段的`STW`[1]时间。
  - `0.84`表示所有P在`mark concurrent`(并发标记)所使用的时间。
  - `0.006`表示单个P在`markTermination`阶段的`STW`时间
- `0.018+0.068/1.1/0.029+0.055 ms cpu`:
  - `0.018`:表示整个进程在`mark`阶段`STW`停顿的时间。
  - `0.068/1.1/0.029`:0表示`mutator assist`占用的时间，`1.1`表示`dedicated +fractional`占用的时间，`0.029`表示`idle`占用的时间。
- `9->9->8 MB`:
  - `9`:表示开始`mark`阶段前的`heap_live`大小。
  - `9`:表示开始`markTermination`阶段前的`heap_live`大小。
  - `8`:表示被标记对象的大小。
- `10 MB goal`:表示下一次出发GC回收的阈值是10MB。
- `8P`:本次GC一共涉及多少个P

三、垃圾回收常用方法

1、引用计数(reference counting)

原理：在每个对象内部维护一个整数值，叫做这个对象的引用计数，当对象被引用时引用计数加一，当对象不被引用时引用计数减一。当引用计数为0时，自动销毁对象。

注意：

- 回收的是内存空间
- 链式引用都要跟着更新计数

缺陷：

- 不能解决循环引用的问题；
- 频繁更新引用计数降低了性能

```go
package main

import "fmt"

func main() {
  var name string
  name = "hello"
  var new_name *string = &name
  fmt.Println(&name) //0x14000096210

  fmt.Println(&new_name) //0x140000b4018

```

```
13
14      *new_name = ""
15      fmt.Println(&new_name)//0x140000b4018
16
17
18      new_name = nil
19      fmt.Println(&new_name)//0x140000b4018
20
21  }
22
```

2、标记—清除法(mark&sweep)(Go1.5 以前用的)

- **标记**：从程序的根节点开始，递归遍历所有对象，将能变遍历的对象打上标记；
- **清除**：将所有为标记的对象当作垃圾销毁；
- 缺点：
    - 在标记时，必须暂停整个程序，遍历递归，会消耗很多时间。

3、分代收集(Generational Garbage Collection)(Java、.Net等使用这种)

分代收集是传统Mark-Sweep的一个改进

原理：

- 新对象放入第0代
- 当内存用量超过一个较小的阈值时，触发0代收集
- 当第0代幸存的对象(未被收集)放入第1代
- 只有当内存用量超过一个较高代阈值时，才会出发1代收集
- 2代同理

4、三色标记算法(Go 1.5、Go 1.6)

三色标记是传统的Mark-Sweep的一个改进，它是一个并发的GC算法。

让系统的GC暂停时间能够被预测

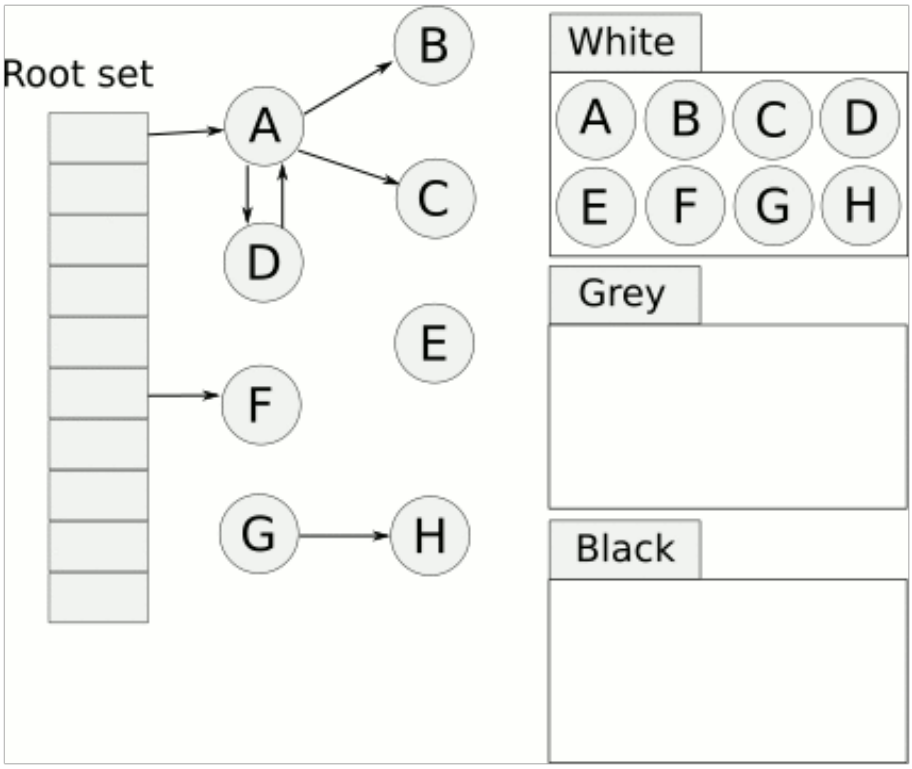
通过三个阶段确定要清楚的对象有哪些

根节点对象：全局变量喝函数栈里的对象

原理：

- 首先创建三个集合：白、灰、黑；
    - 白色对象（可能死亡）：未被回收器访问到的对象。在回收开始阶段，所有对象均为白色，当回收结束后，白色对象均不可达。

    - 灰色对象（波面）：已被回收器访问到的对象，但回收器需要对其中的一个或多个指针进行扫描，因为他们可能还指向白色对象。
    - 黑色对象（确定存活）：已被回收器访问到的对象，其中所有字段都已被扫描，黑色对象中任何一个指针都不可能直接指向白色对象。
- 将所有对象放入白色集合中；
- 然后从根节点开始遍历所有对象（注意这里并不递归遍历，只遍历一次），把遍历到的对象从白色集合放入灰色集合；
- 之后遍历灰色集合，将灰色对象***引用的对象***从白色集合放入灰色集合，之后将此灰色对象放入黑色集合(这里指的是灰色对象引用到的所有对象，包括灰色节点间接引用的那些对象，没有自节点的，也会被移动到黑色集合当中)；
- 重复上一步，直到灰色无任何对象；
- 通过write-barrier（屏障)检测对象有变化，重复以上操作，直到灰色对象中没有对象
- 收集所有白色对象（垃圾），还在白色集合中的意味着已经找不到该对象在哪儿了，不可能再被重新引用。



一次垃圾回收♻️完成后，将黑色集合编程白色集合，会进一步gc操作，依次循环。

注意：

- 如果没有STW，在标记的时候对象被引用了，会出现对象丢失现象，可以使用写屏障实现类似STW的效果，尽可能缩短STW的时间

缺陷：

- 垃圾产生的速度会大于垃圾收集的速度，这样会导致程序中的垃圾越来越多，无法被收集掉。

四、写屏障、

> 原理：当某一个对象被标黑，此时这个对象又被其他对象引用，就把引用这个对象的对象变灰入队，比如a对象被标黑，a又被b引用，那就把b标灰

> 强三色不变性：黑色对象不会指向白色对象，只会指向灰色对象或者黑色对象

> 弱三色不变性：黑色对象指向的白色对象必须包含一条从灰色对象经由多个白色对象的可达路径

1、插入写屏障（使满足强三色不变性）

> - 如果两个对象之间新建立引用，那么引用指向的对象就会被标记为灰色以满足强三色不变性
> - 标记其对应对象为灰色状态，这样就不存在黑色对象引用白色对象的情况了，满足强三色不变性

2、删除写屏障（使满足弱三色不变性）

> 一个灰色对象指向一个白色对象的引用被删除，那么在删除之前写屏障检测到内存变化，就会把白色对象变灰

3、混合屏障

> 插入写屏障和删除写屏障

五、GC调优

1、减少对象的分配，合理重复利用

2、避免string和[]byte转化

`string和[]byte转化的时候，底层数据结构会进行复制，导致gc效率变低`

3、尽量少使用`+`拼接字符串

`针对它的每一个操作都会创建一个新的string，可以用string.Join代替`

4、尽量避免频繁创建对象(`&abc{}`,`new(abc{})`,`make()`)

5、程序开发阶段关注请求响应的时间，每个功能优化一点时间，很多功能就很客观了。

---

1. STW 可以是Stop The World的缩写，也可以是Start The World的缩写。通常意义上指的是从Stop The World到Start The World这一段时间间隔。垃圾回收过程中为了保证准确性、防止无止境的内存增长等问题而不可避免的需要停止赋值器进一步操作对象图以完成垃圾回收。STW时间越长，对用户代码造成的影响越大。 ↩