1. 需要注意的是 { 不能单独放在一行，所以以下代码在运行时会产生错误：

```
func main()
{ // 错误，{ 不能在单独的行上
    fmt.Println("Hello, World!")
}
```

2. Go 语言的字符串连接可以通过 + 实现

```
func main() {
    fmt.Println("Google" + "Runoob")
}
```

以上实例输出结果为：

```
GoogleRunoob
```

3. Go 语言中使用 fmt.Sprintf 或 fmt.Printf 格式化字符串并赋值给新串：

- Sprintf 根据格式化参数生成格式化的字符串并返回该字符串。
- Printf 根据格式化参数生成格式化的字符串并写入标准输出。

```
package main
import "fmt"
func main() {
   // %d 表示整型数字，%s 表示字符串
    var stockcode=123
    var enddate="2020-12-31"
    var url="Code=%d&endDate=%s"
    var target_url=fmt.Sprintf(url,stockcode,enddate)
    fmt.Println(target_url)
}
```

输出结果为：

```
Code=123&endDate=2020-12-31
```

4. Go 语言按类别有以下几种数据类型：

序号	类型和描述
1	**布尔型** 布尔型的值只可以是常量 true 或者 false。一个简单的例子：var b bool = true。
2	**数字类型** 整型 int 和浮点型 float32、float64，Go 语言支持整型和浮点型数字，并且支持复数，其中位的运算采用补码。
3	**字符串类型:** 字符串就是一串固定长度的字符连接起来的字符序列。Go 的字符串是由单个字节连接起来的。Go 语言的字符串的字节使用 UTF-8 编码标识 Unicode 文本。
4	**派生类型:** (a) 指针类型（Pointer） (b) 数组类型、切片类型、Map 类型 (c) 结构化类型(struct)、接口类型（interface） (d) 函数类型、Channel 类型

Go 也有基于架构的类型，例如：int、uint 和 uintptr。

序号	类型和描述
1	**uint8**：无符号 8 位整型 (0 到 255)
2	**uint16**：无符号 16 位整型 (0 到 65535)
3	**uint32**：无符号 32 位整型 (0 到 4294967295)
4	**uint64**：无符号 64 位整型 (0 到 18446744073709551615)
5	**int8**：有符号 8 位整型 (-128 到 127)
6	**int16**：有符号 16 位整型 (-32768 到 32767)
7	**int32**：有符号 32 位整型 (-2147483648 到 2147483647)
8	**int64**：有符号 64 位整型 (-9223372036854775808 到 9223372036854775807)

序号	类型和描述
1	**float32**：IEEE-754 32 位浮点型数
2	**float64**：IEEE-754 64 位浮点型数
3	**complex64**：32 位实数和虚数
4	**complex128**：64 位实数和虚数

序号	类型和描述
1	**byte**：类似 uint8
2	**rune**：类似 int32
3	**uint**：32 或 64 位
4	**int**：与 uint 一样大小
5	**uintptr**：无符号整型，用于存放一个指针

5. 声明变量的一般形式是使用 var 关键字：

```
var identifier type
```

可以一次声明多个变量：

```
var identifier1, identifier2 type
```

第二种，根据值自行判定变量类型。

```
var v_name = value
```

第三种，不声明变量类型，直接使用 := 声明变量，格式：

```
v_name := value
```

第四种，多变量声明

```
//类型相同多个变量，非全局变量
var vname1, vname2, vname3 type
vname1, vname2, vname3 = v1, v2, v3

var vname1, vname2, vname3 = v1, v2, v3 // 和 python 很像,不需要显示声明类型，自动推断
vname1, vname2, vname3 := v1, v2, v3 // 出现在 := 左侧的变量不应该是已经被声明过的，否则会导致编译错误
// 这种因式分解关键字的写法一般用于声明全局变量
var (
    vname1 v_type1
    vname2 v_type2
)
```

代码：

```
func main() {
    var a string = "Runoob"
    var b, c int = 1, 2
    // 声明一个变量并初始化
    var d = "RUNOOB"
}
```

6. 值类型和引用类型

当使用等号 = 将一个变量的值赋值给另一个变量时，如：j = i，实际上是在内存中将 i 的值进行了拷贝。可以通

过 &i 来获取变量 i 的内存地址，例如：0xf840000040（每次的地址都可能不一样）。**值类型变量的值存储在堆中。**更复杂的数据通常会需要使用多个字，这些数据一般使用引用类型保存。一个引用类型的变量 r1 存储的是 r1 的值所在的内存地址（数字），或内存地址中第一个字所在的位置。

Fig 4.3: Reference types and assignment

7. iota 用法

iota 在 const 关键字出现时将被重置为 0(const 内部的第一行之前)，const 中每新增一行常量声明将使 iota 计数一次(iota 可理解为 const 语句块中的行索引)。**iota 可以被用作枚举值**：

```
const (
    a = iota
    b = iota
    c = iota
)
```

第一个 iota 等于 0，每当 iota 在新的一行被使用时，它的值都会自动加 1；所以 a=0, b=1, c=2 可以简写为如下形式：

```
const (
    a = iota
    b
    C
)
```

实例

```
package main
import "fmt"

func main() {
    const (
            a = iota   //0
            b          //1
            c          //2
            d = "ha"   //独立值，iota += 1
            e          //"ha"   iota += 1
            f = 100    //iota +=1
            g          //100  iota +=1
            h = iota   //7,恢复计数
            i          //8
    )
    fmt.Println(a,b,c,d,e,f,g,h,i)
}
```

以上实例运行结果为：

```
0 1 2 ha ha 100 100 7 8
```

8. 指针案例

```
func main() {
   var a = 4
   var ptr *int
   ptr = &a


   fmt.Println(ptr)  // ptr 所指变量的地址（0xc00001e0a8）
   fmt.Println(*ptr) // ptr 所指变量（4）
   fmt.Println(&a)   // ptr 所指变量的地址（0xc00001e0a8）
   fmt.Println(&ptr) // ptr 的地址（0xc00000a028）
}
```

9. Switch 语句

switch 默认情况下 case 最后自带 break 语句，匹配成功后就不会执行其他 case，如果我们需要执行后面的 case，可以使用 fallthrough 。

Go 编程语言中 switch 语句的语法如下：

```
switch var1 {
   case val1:
      ...
   case val2:
      ...
   default:
      ...
}
```

变量 var1 可以是任何类型，而 val1 和 val2 是同类型的任意值。类型不被局限于常量或整数，但必须是相同的类型；或者最终结果为相同类型的表达式。

可以同时测试多个可能符合条件的值，使用逗号分割它们，例如：case val1, val2, val3。

```
    switch marks {
      case 90: grade = "A"
      case 80: grade = "B"
      case 50,60,70 : grade = "C"
      default: grade = "D"
   }
```

switch 语句还可以被用于 **type-switch 来判断某个 interface 变量中实际存储的变量类型**。

Type Switch 语法格式如下：

```
switch x.(type){
   case type:
     statement(s);
   case type:
     statement(s);
   /* 你可以定义任意个数的 case */
   default: /* 可选 */
     statement(s);
}
```

实例

```
package main
import "fmt"
```

```
func main() {
    var x interface{}

    switch i := x.(type) {
        case nil:
            fmt.Printf(" x 的类型 :%T",i)
        case int:
            fmt.Printf("x 是 int 型")
        case float64:
            fmt.Printf("x 是 float64 型")
        case func(int) float64:
            fmt.Printf("x 是 func(int) 型")
        case bool, string:
            fmt.Printf("x 是 bool 或 string 型" )
        default:
            fmt.Printf("未知型")
    }
}
```

10. Fallthrough

使用 **fallthrough 会强制执行后面的 case 语句**，fallthrough **不会判断下一条 case 的表达式结果是否为 true**。

```
package main
import "fmt"
func main() {
    switch {
    case false:
            fmt.Println("1、case 条件语句为 false")
            fallthrough
    case true:
            fmt.Println("2、case 条件语句为 true")
            fallthrough
    case false:
            fmt.Println("3、case 条件语句为 false")
            fallthrough
    case true:
            fmt.Println("4、case 条件语句为 true")
    case false:
            fmt.Println("5、case 条件语句为 false")
            fallthrough
    default:
            fmt.Println("6、默认 case")
    }
}
```

以上代码执行结果为：

```
2、case 条件语句为 true
3、case 条件语句为 false
4、case 条件语句为 true
```

从以上代码输出的结果可以看出：switch 从第一个判断表达式为 true 的 case 开始执行，如果 case 带有 fallthrough，程序会继续执行下一条 case，且它不会去判断下一个 case 的表达式是否为 true。

11. Select 语句

select 语句只能用于通道操作，每个 case 必须是一个通道操作，要么是发送要么是接收。

select 语句会监听所有指定的通道上的操作，一旦其中一个通道准备好就会执行相应的代码块。

如果多个通道都准备好，那么 select 语句会随机选择一个通道执行。如果所有通道都没有准备好，那么执行 default 块中的代码。

```
select {
  case <- channel1:
    // 执行的代码
  case value := <- channel2:
    // 执行的代码
  case channel3 <- value:
    // 执行的代码

    // 你可以定义任意数量的 case

  default:
    // 所有通道都没有准备好，执行的代码
}
```

以下描述了 select 语句的语法：

- 每个 case 都必须是一个通道
- 所有 channel 表达式都会被求值
- 所有被发送的表达式都会被求值
- 如果任意某个通道可以进行，它就执行，其他被忽略。
- 如果有多个 case 都可以运行，select 会随机公平地选出一个执行，其他不会执行。

  否则：
  - 如果有 default 子句，则执行该语句。
  - 如果没有 default 子句，select 将阻塞，直到某个通道可以运行；Go 不会重新对 channel 或值进行求值。

```
package main
import (
    "fmt"
    "time"
)
func main() {
    c1 := make(chan string)
    c2 := make(chan string)

    go func() {
        time.Sleep(1 * time.Second)
        c1 <- "one"
    }()
    go func() {
        time.Sleep(2 * time.Second)
        c2 <- "two"
    }()

    for i := 0; i < 2; i++ {
```

```
        select {
        case msg1 := <-c1:
            fmt.Println("received", msg1)
        case msg2 := <-c2:
            fmt.Println("received", msg2)
        }
    }
}
```

12. 函数作为实参

Go 语言可以很灵活的创建函数，并作为另外一个函数的实参。以下实例中我们在定义的函数中初始化一个变量，该函数仅仅是为了使用内置函数 **math.sqrt()**，实例为：

```
package main
import (
   "fmt"
   "math"
)
func main(){
   /* 声明函数变量 */
   getSquareRoot := func(x float64) float64 {
      return math.Sqrt(x)
   }
   /* 使用函数 */
   fmt.Println(getSquareRoot(9))
}
```

以上代码执行结果为：

```
3
```

13. 闭包

```
package main
import "fmt"

func getSequence() func() int {
   i:=0
   return func() int {
      i+=1
     return i
   }
}

func main(){
   /* nextNumber 为一个函数，函数 i 为 0 */
   nextNumber := getSequence()

   /* 调用 nextNumber 函数，i 变量自增 1 并返回 */
   fmt.Println(nextNumber())
   fmt.Println(nextNumber())
```

```
    fmt.Println(nextNumber())

    /* 创建新的函数 nextNumber1，并查看结果 */
    nextNumber1 := getSequence()
    fmt.Println(nextNumber1())
    fmt.Println(nextNumber1())
}
```

以上代码执行结果为：

```
1 2 3 1 2
```

以下实例我们定义了多个匿名函数，并展示了如何将匿名函数赋值给变量、在函数内部使用匿名函数以及将匿名函数作为参数传递给其他函数。

```
package main
import "fmt"

func main() {
    // 定义一个匿名函数并将其赋值给变量 add
    add := func(a, b int) int {
        return a + b
    }

    // 调用匿名函数
    result := add(3, 5)
    fmt.Println("3 + 5 =", result)

    // 在函数内部使用匿名函数
    multiply := func(x, y int) int {
        return x * y
    }

    product := multiply(4, 6)
    fmt.Println("4 * 6 =", product)

    // 将匿名函数作为参数传递给其他函数
    calculate := func(operation func(int, int) int, x, y int) int {
        return operation(x, y)
    }

    sum := calculate(add, 2, 8)
    fmt.Println("2 + 8 =", sum)

    // 也可以直接在函数调用中定义匿名函数
    difference := calculate(func(a, b int) int {
        return a - b
    }, 10, 4)
    fmt.Println("10 - 4 =", difference)
}
```

以上代码执行结果为：

```
3 + 5 = 8
4 * 6 = 24
2 + 8 = 10
10 - 4 = 6
```

14. 方法

Go 语言中同时有函数和方法。一个方法就是一个包含了接受者的函数，接受者可以是命名类型或者结构体类型的一个值或者是一个指针。所有给定类型的方法属于该类型的方法集。语法格式如下：

```
func (variable_name variable_data_type) function_name() [return_type]{
   /* 函数体 */
}
```

下面定义一个结构体类型和该类型的一个方法：

实例

```
package main

import (
    "fmt"
)

/* 定义结构体 */
type Circle struct {
  radius float64
}

func main() {
  var c1 Circle
  c1.radius = 10.00
  fmt.Println("圆的面积 = ", c1.getArea())
}

//该 method 属于 Circle 类型对象中的方法
func (c Circle) getArea() float64 {
  //c.radius 即为 Circle 类型对象中的属性
  return 3.14 * c.radius * c.radius
}
```

以上代码执行结果为：

```
圆的面积 =  314
```

15. delete() 函数用于删除集合的元素，参数为 map 和其对应的 key。

```
package main
import "fmt"
func main() {
        /* 创建map */
        countryCapitalMap := map[string]string{"France": "Paris", "Italy": "Rome", "Japan": "Tokyo"}
        /*删除元素*/
        delete(countryCapitalMap, "France")
}
```

16. 字符串类型转化

整型 -> 字符串：`newString := strconv.FormatInt(num, 10)`

字符串 -> 整型：`num, _ := strconv.ParseInt(name, 10, 64)`

17. 接口类型转换

接口类型转换有两种情况：**类型断言**和**类型转换**。

类型断言用于将接口类型转换为指定类型，其语法为：

```
value.(type) 或者
value.(T)
```

其中 value 是接口类型的变量，type 或 T 是要转换成的类型。如果类型断言成功，它将返回转换后的值和一个布尔值，表示转换是否成功。

```go
package main
import "fmt"

func main() {
    var i interface{} = "Hello, World"
    str, ok := i.(string)
    if ok {
        fmt.Printf("'%s' is a string\n", str)
    } else {
        fmt.Println("conversion failed")
    }
}
```

18. 接口

Go 语言提供了另外一种数据类型即接口，它把所有的具有共性的方法定义在一起，任何其他类型只要实现了这些方法就是实现了这个接口。接口可以让我们将不同的类型绑定到一组公共的方法上，从而实现多态和灵活的设计。

Go 语言中的接口是隐式实现的，也就是说，如果一个类型实现了一个接口定义的**所有方法**，那么它就自动地实现了该接口。因此，我们可以通过将接口作为参数来实现对不同类型的调用，从而实现多态。

```go
/* 定义接口 */
type interface_name interface {
   method_name1 [return_type]
   method_name2 [return_type]
   ...
   method_namen [return_type]
}

/* 定义结构体 */
type struct_name struct {
   /* variables */
}

/* 实现接口方法 */
func (struct_name_variable struct_name) method_name1() [return_type] {
   /* 方法实现 */
}
```

```
...
func (struct_name_variable struct_name) method_namen() [return_type] {
   /* 方法实现*/
}
```

19. Go 并发-Goroutine

Go 语言支持并发，我们只需要**通过 go 关键字来开启** goroutine 即可。goroutine 是轻量级线程，goroutine 的调度是由 Golang 运行时进行管理的。goroutine 语法格式：

```
go 函数名( 参数列表 )
```

或者匿名函数写法如下：

```
go func() {

}()
```

代码示例：

```
func testGoroutine() {
    go func(str string) {
        for i := 0; i < 5; i++ {
            time.Sleep(100 * time.Millisecond)
            fmt.Println(str)
        }
    }("A")

    go func(str string) {
        for i := 0; i < 5; i++ {
            time.Sleep(100 * time.Millisecond)
            fmt.Println(str)
        }
    }("B")
    time.Sleep(1000 * time.Millisecond)
}
```

20. Go 并发-通道 Channel

通道（channel）是**用来传递数据**的一个数据结构。通道可用于**两个 goroutine 之间**通过传递一个指定类型的值来同步运行和通讯。操作符 **<-** 用于指定通道的方向，发送或接收。如果未指定方向，则为双向通道。

```
ch <- v    // 把 v 发送到通道 ch
v := <-ch  // 从 ch 接收数据，并把值赋给 v
```

声明一个通道很简单，我们使用 chan 关键字即可，通道在使用前必须先创建：

```
ch := make(chan int)
```

注意：默认情况下，通道是不带缓冲区的。发送端发送数据，同时必须有接收端相应的接收数据。

以下实例通过两个 goroutine 来计算数字之和，在 goroutine 完成计算后，它会计算两个结果的和：

```
package main
import "fmt"

func sum(s []int, c chan int) {
    sum := 0
    for _, v := range s {
        sum += v
```

```
        time.Sleep(100 * time.Millisecond)
    }
    c <- sum // 把 sum 发送到通道 c
}

func main() {
    s := []int{7, 2, 8, -9, 4, 0}

    c := make(chan int)
    go sum(s[:len(s)/2], c)
    go sum(s[len(s)/2:], c)
    x, y := <-c, <-c // 从通道 c 中接收

    fmt.Println(x, y, x+y)
}
```

输出结果为：

```
-5 17 12
```

通道缓冲区

通道可以设置缓冲区，通过 make 的第二个参数指定缓冲区大小：

```
ch := make(chan int, 100)
```

带缓冲区的通道允许发送端的数据发送和接收端的数据获取处于**异步状态**，就是说发送端发送的数据可以放在缓冲区里面，可以等待接收端去获取数据，而不是立刻需要接收端去获取数据。

不过由于缓冲区的大小是有限的，所以还是必须有接收端来接收数据的，否则缓冲区一满，数据发送端就无法再发送数据了。

注意：如果通道不带缓冲，发送方会阻塞直到接收方从通道中接收了值。如果通道带缓冲，发送方则会阻塞直到发送的值被拷贝到缓冲区内；如果缓冲区已满，则意味着需要等待直到某个接收方获取到一个值。接收方在有值可以接收之前会一直阻塞。

```
package main
import "fmt"

func main() {
    // 这里我们定义了一个可以存储整数类型的带缓冲通道
    // 缓冲区大小为2
    ch := make(chan int, 2)
    // 因为 ch 是带缓冲的通道，我们可以同时发送两个数据
    // 而不用立刻需要去同步读取数据
    ch <- 1
    ch <- 2
    // 获取这两个数据
    fmt.Println(<-ch)
    fmt.Println(<-ch)
}
```

执行输出结果为：

```
12
```

Go 遍历通道与关闭通道

Go 通过 range 关键字来实现遍历读取到的数据，类似于与数组或切片。格式如下：

```
v, ok := <-ch
```

如果通道接收不到数据后 ok 就为 false，这时通道就可以使用 close() 函数来关闭。

```
package main
import (
    "fmt"
)

func fibonacci(n int, c chan int) {
    x, y := 0, 1
    for i := 0; i < n; i++ {
        c <- x
        x, y = y, x+y
    }
    close(c)
}

func main() {
    c := make(chan int, 10)
    go fibonacci(cap(c), c)
    // range 函数遍历每个从通道接收到的数据，因为 c 在发送完 10 个数据之后就关闭了通道，
    // 所以这里我们 range 函数在接收到 10 个数据之后就结束了。如果上面的 c 通道不关闭，
    // 那么 range 函数就不会结束，从而在接收第 11 个数据的时候就阻塞了。
    for i := range c {
        fmt.Println(i)
    }
}
```

21. WaitGroup 同步锁

有一种业务场景是你需要知道所有的协程是否已执行完成他们的任务。这个和只需要随机选择一个条件为 true 的 select 不同，他需要你满足所有的条件都是 true 才可以激活主线程继续执行。这里的条件指的是非阻塞的通道操作。

`WaitGroup` 是一个带着计数器的结构体，这个计数器可以追踪到有多少协程创建，有多少协程完成了其工作。当计数器为 0 的时候说明所有协程都完成了其工作。

```
func service(wg *sync.WaitGroup, instance int) {
    time.Sleep(2 * time.Second)
    fmt.Println("Service called on instance",instance)
    wg.Done() //协程数-1
}

func main() {
    fmt.Println("main started")
    var wg sync.WaitGroup
    for i:=1;i<= 3; i++{
        wg.Add(1)
        go service(&wg,i)
```

```
    }
    wg.Wait()//阻塞
    fmt.Println("main stop")
}
```

22. Mutex 互斥锁

在 Go 中，互斥数据结构（map）由 sync 包提供。在 Go 中，多协程去操作一个值都可能会引起竞态条件。我们需要在操作数据之前使用 mutex.Lock() 去锁定它，一旦我们完成操作，比如上面提到的 i = i + 1，我们就可以使用 mutext.Unlock() 方法解锁。

```
func worker(wg *sync.WaitGroup,m *sync.Mutex) {
    m.Lock()
    i = i+1
    m.Unlock()
    wg.Done()
}

func main() {
    fmt.Println("main started")
    var wg sync.WaitGroup
    var m sync.Mutex
    for i:=0;i<1000;i++{
        wg.Add(1)
        go worker(&wg,&m)
    }
    wg.Wait()
    fmt.Println("main stop",i)
}
```

23. GC 算法-三色标记法

目前主流的垃圾回收算法有两类，分别是**追踪式垃圾回收算法（Tracing garbage collection）**和引用计数法（ Reference counting ）。而三色标记法是属于追踪式垃圾回收算法的一种。

追踪式算法的核心思想是判断一个对象是否可达，因为一旦这个对象不可达就可以立刻被 GC 回收了。

那么如何判断一个对象是否可达呢？

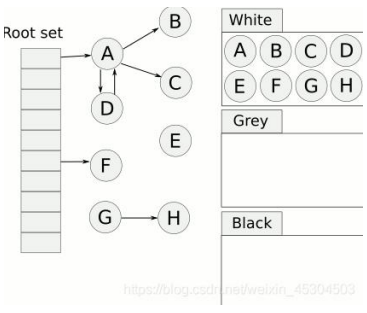
分为两步：

- 第一步，找出所有的全局变量和当前函数栈里的变量，标记为可达。
- 第二步，从已经标记的数据开始，进一步标记它们可访问的变量，周而复始，这一过程也叫传递闭包。

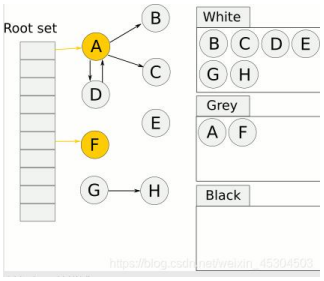
三色标记算法是对标记阶段的改进，原理如下：

1. 整个进程空间里申请每个对象占据的内存可以视为一个图，**初始状态下每个内存对象都是白色标记**。

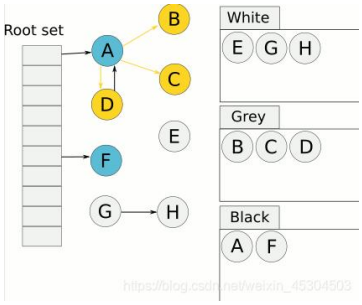
2. 先 stop the world，将**扫描任务作为多个并发的 goroutine 立即入队给调度器**，进而被 **CPU 处理**，第一轮先扫描所有可达的内存对象，标记为灰色放入队列，放入待处理队列。然后从根节点开始遍历所有对象（注意这里并不递归遍历），



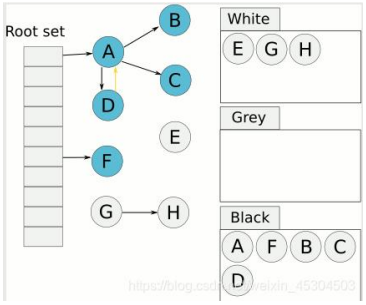
把遍历到的对象从白色集合放入灰色集合。因为 root set 指向了 A、F，所以从根结点开始遍历的是 A、F，所以是把 A、F 放到灰色集合中。



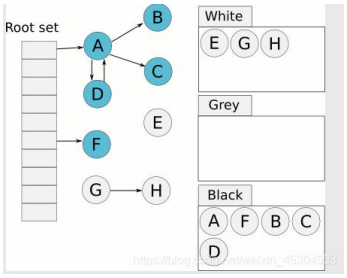
3. 从队列取出灰色对象，将其引用对象标记为灰色放入队列，自身标记为黑色。我们可以发现这个 A 指向了 B，C，D 所以也就是把 BCD 放到灰色中，把 A 放到黑色中，而 F 没有指任何的对象，所以直接放到黑色中。



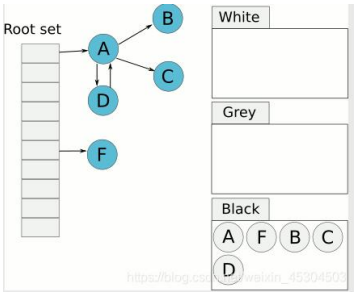
4. 重复 3，直到灰色对象队列为空。因为 D 指向了 A 所以 D 也放到了黑色中，而 B 和 C 能放到黑色集合中的道理和 F 一样，已经没有了可指向的对象了。



5. 通过 write-barrier 检测对象有无变化，重复以上操作。由于这个 EGH 并没有和 RootSet 有直接或是间接的关系，所以就会被清除。



6. 此时白色对象即为垃圾，进行回收。



所以可以看出这里的情况，只要是和 root set 根集合直接相关的对象或是间接相关的对象都不会被清楚。只有不相关的才会被回收。