目　录

介绍

转自：https://github.com/aceld/golang

本书针对Golang专题性热门技术深入理解，修养在Golang领域深入话题，脱胎换骨。

主要内容涉及：

- 深入理解GMP全场景分析
- 深入理解GC三色标记与混合写屏障
- Golang技术性能调优

> 作者:刘丹冰Aceld
>
> 

Golang修养必经之路

最常用的调试 golang 的 bug 以及性能问题的实践方法

一、流？I/O操作？阻塞？

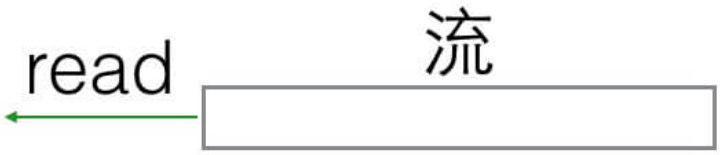
(1) 流

- 可以进行I/O操作的内核对象
- 文件、管道、套接字.....
- 流的入口：文件描述符(fd)

(2) I/O操作

所有对流的读写操作，我们都可以称之为IO操作。

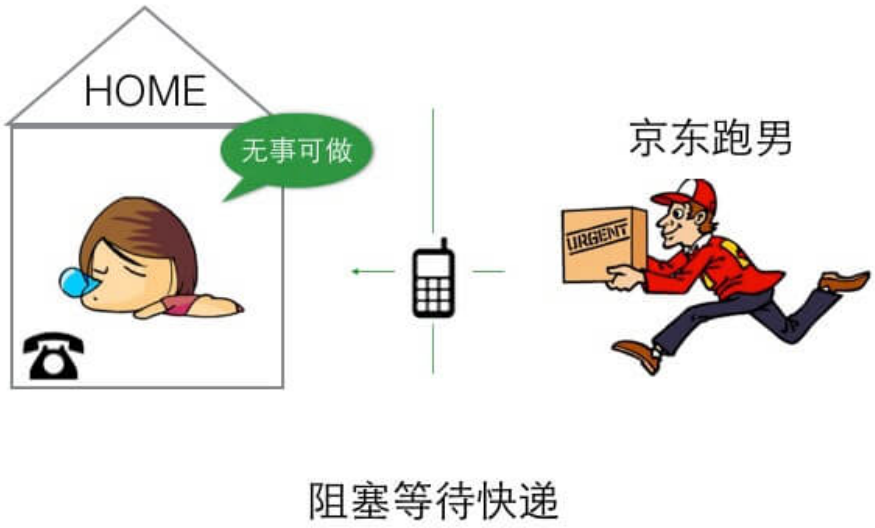
当一个流中， 在没有数据read的时候，或者说在流中已经写满了数据，再write，我们的IO操作就会出现一种现象，就是阻塞现象，如下图。



---

(3) 阻塞

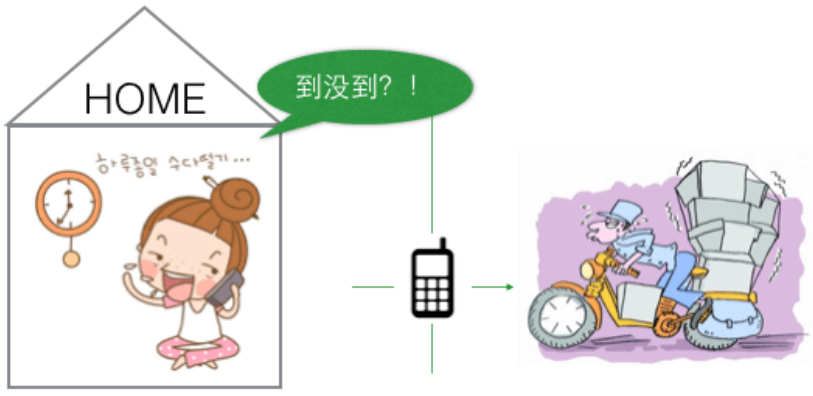
阻塞?



阻塞等待快递

阻塞场景: 你有一份快递，家里有个座机，快递到了主动给你打电话，期间你可以休息。

非阻塞，忙轮询



每隔一分钟催一次

非阻塞，忙轮询场景: 你性子比较急躁， 每分钟就要打电话询问快递小哥一次， 到底有没有到，快递员接你电话要停止运输，这样很耽误快递小哥的运输速度。

- 阻塞等待

空出大脑可以安心睡觉, 不影响快递员工作（不占用CPU宝贵的时间片）。

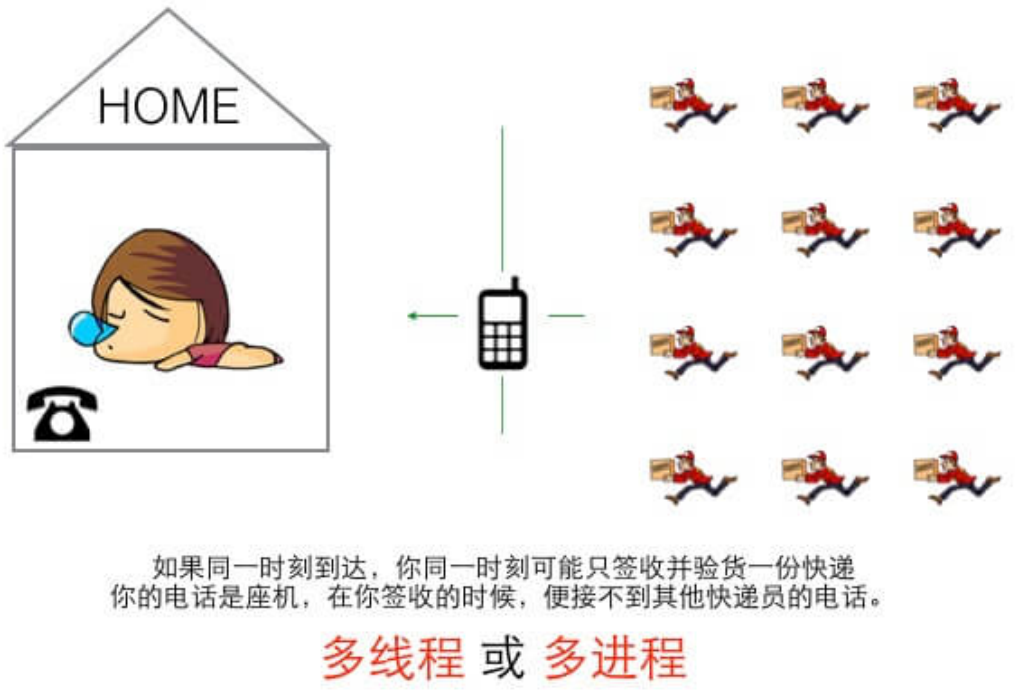
- 非阻塞，忙轮询

浪费时间，浪费电话费，占用快递员时间（占用CPU，系统资源）。

很明显，阻塞等待这种方式，对于通信上是有明显优势的， 那么它有哪些弊端呢？

二、解决阻塞死等待的办法

阻塞死等待的缺点

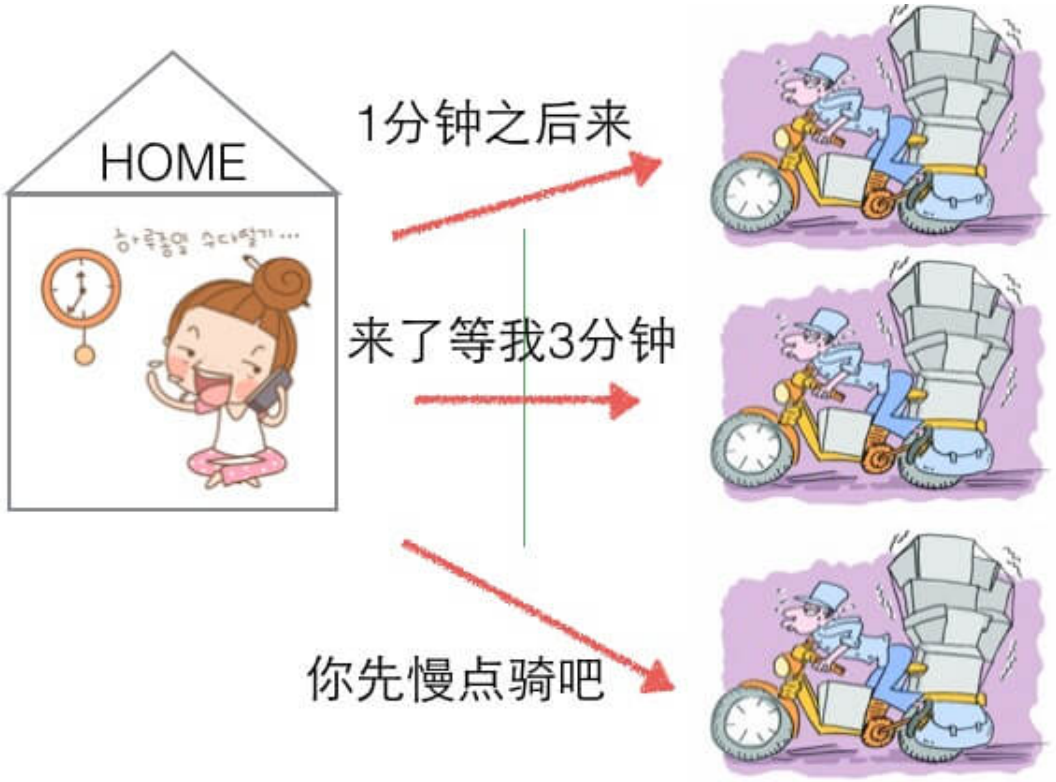


也就是同一时刻，你只能被动的处理一个快递员的签收业务，其他快递员打电话打不进来，只能干瞪眼等待。那么解决这个问题，家里多买N个座机， 但是依然是你一个人接，也处理不过来，需要用影分身术创建都个自己来接电话(采用多线程或者多进程）来处理。

这种方式就是没有多路IO复用的情况的解决方案， 但是在单线程计算机时代(无法影分身)，这简直是灾难。

那么如果我们不借助影分身的方式(多线程/多进程)，该如何解决阻塞死等待的方法呢？

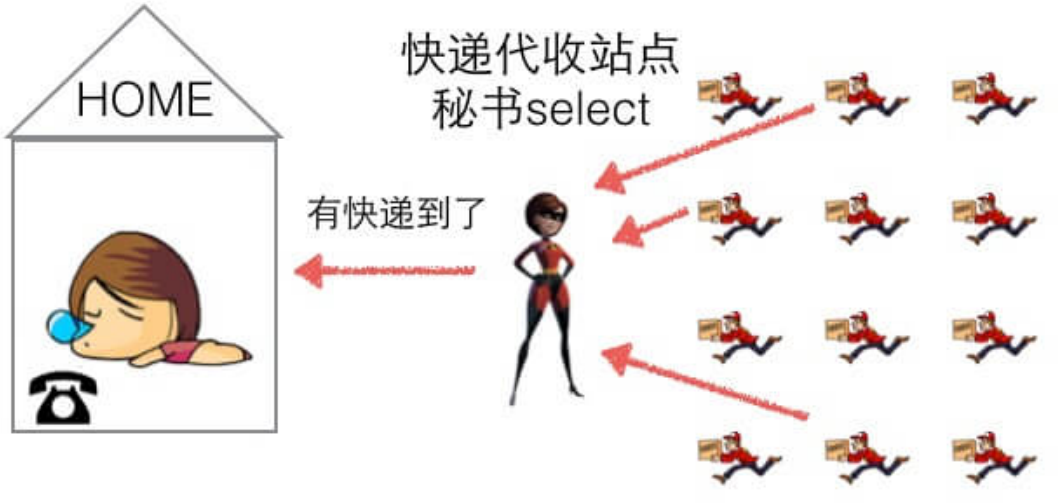
办法一：非阻塞、忙轮询




```
while true {
    for i in 流[] {
        if i has 数据 {
            读 或者 其他处理
        }
    }
}
```

非阻塞忙轮询的方式，可以让用户分别与每个快递员取得联系，宏观上来看，是同时可以与多个快递员沟通(并发效果)、 但是快递员在于用户沟通时耽误前进的速度(浪费CPU)。

办法二：select

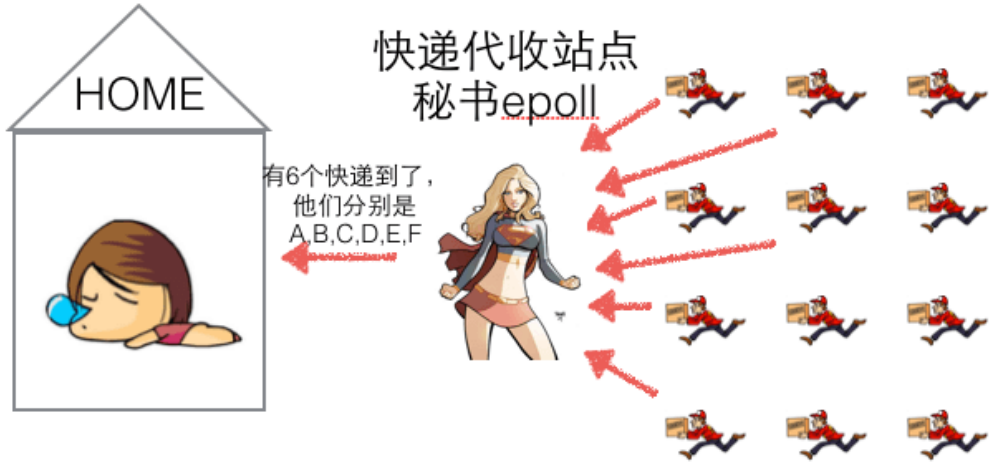


我们可以开设一个代收网点，让快递员全部送到代收点。这个网店管理员叫select。这样我们就可以在家休息了，麻烦的事交给select就好了。当有快递的时候，select负责给我们打电话，期间在家休息睡觉就好了。

但select 代收员比较懒，她记不住快递员的单号，还有快递货物的数量。她只会告诉你快递到了，但是是谁到的，你需要挨个快递员问一遍。

```
while true {
    select(流[]); //阻塞

  //有消息抵达
    for i in 流[] {
        if i has 数据 {
            读 或者 其他处理
        }
    }
}
```

办法三：epoll



epoll的服务态度要比select好很多，在通知我们的时候，不仅告诉我们有几个快递到了，还分别告诉我们是谁谁谁。我们只需要按照epoll给的答复，来询问快递员取快递即可。

```
while true {
    可处理的流[] = epoll_wait(epoll_fd); //阻塞

  //有消息抵达，全部放在 “可处理的流[]”中
    for i in 可处理的流[] {
        读 或者 其他处理
    }
}
```

三、epoll？

- 与select，poll一样，对I/O多路复用的技术
- 只关心“活跃”的链接，无需遍历全部描述符集合
- 能够处理大量的链接请求(系统可以打开的文件数目)

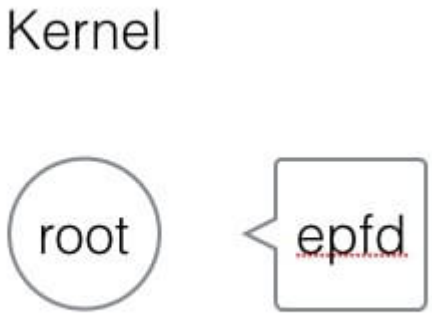
四、epoll的API

(1) 创建EPOLL

```
/**
 * @param size 告诉内核监听的数目
 *
 * @returns 返回一个epoll句柄（即一个文件描述符）
 */
int epoll_create(int size);
```

使用

```
int epfd = epoll_create(1000);
```



创建一个epoll句柄，实际上是在内核空间，建立一个root根节点，这个根节点的关系与epfd相对应。

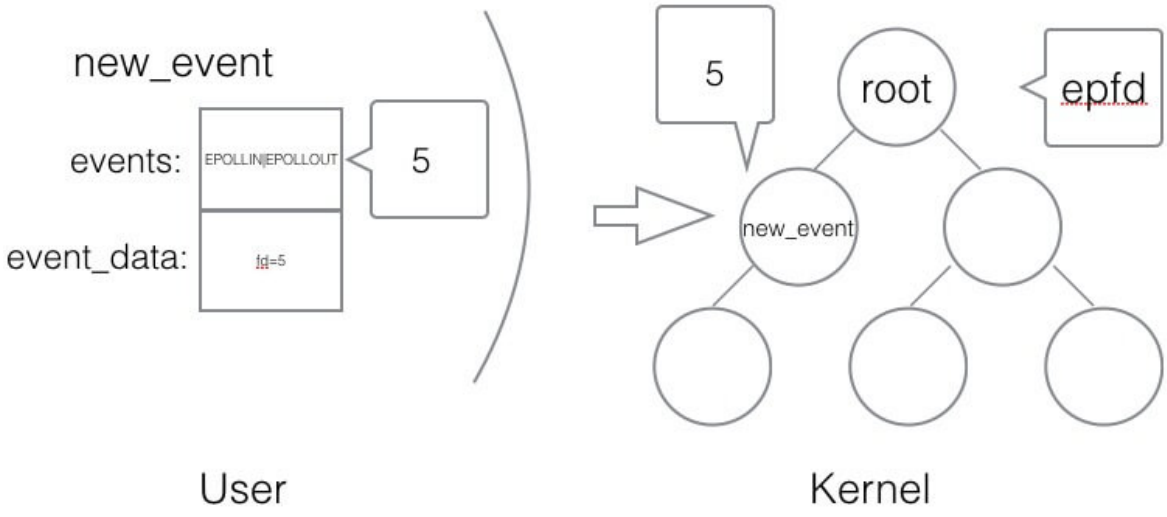
(2) 控制EPOLL

```
/**
* @param epfd 用epoll_create所创建的epoll句柄
* @param op 表示对epoll监控描述符控制的动作
*
* EPOLL_CTL_ADD(注册新的fd到epfd)
* EPOLL_CTL_MOD(修改已经注册的fd的监听事件)
* EPOLL_CTL_DEL(epfd删除一个fd)
*
* @param fd 需要监听的文件描述符
* @param event 告诉内核需要监听的事件
*
* @returns 成功返回0，失败返回-1, errno查看错误信息
*/
int epoll_ctl(int epfd, int op, int fd,
struct epoll_event *event);


struct epoll_event {
    __uint32_t events; /* epoll 事件 */
    epoll_data_t data; /* 用户传递的数据 */
}

/*
 * events : {EPOLLIN, EPOLLOUT, EPOLLPRI,
                        EPOLLHUP, EPOLLET, EPOLLONESHOT}
 */
typedef union epoll_data {
    void *ptr;
    int fd;
    uint32_t u32;
    uint64_t u64;
} epoll_data_t;
```

使用

```
struct epoll_event new_event;

new_event.events = EPOLLIN | EPOLLOUT;
new_event.data.fd = 5;

epoll_ctl(epfd, EPOLL_CTL_ADD, 5, &new_event);
```

创建一个用户态的事件，绑定到某个fd上，然后添加到内核中的epoll红黑树中。



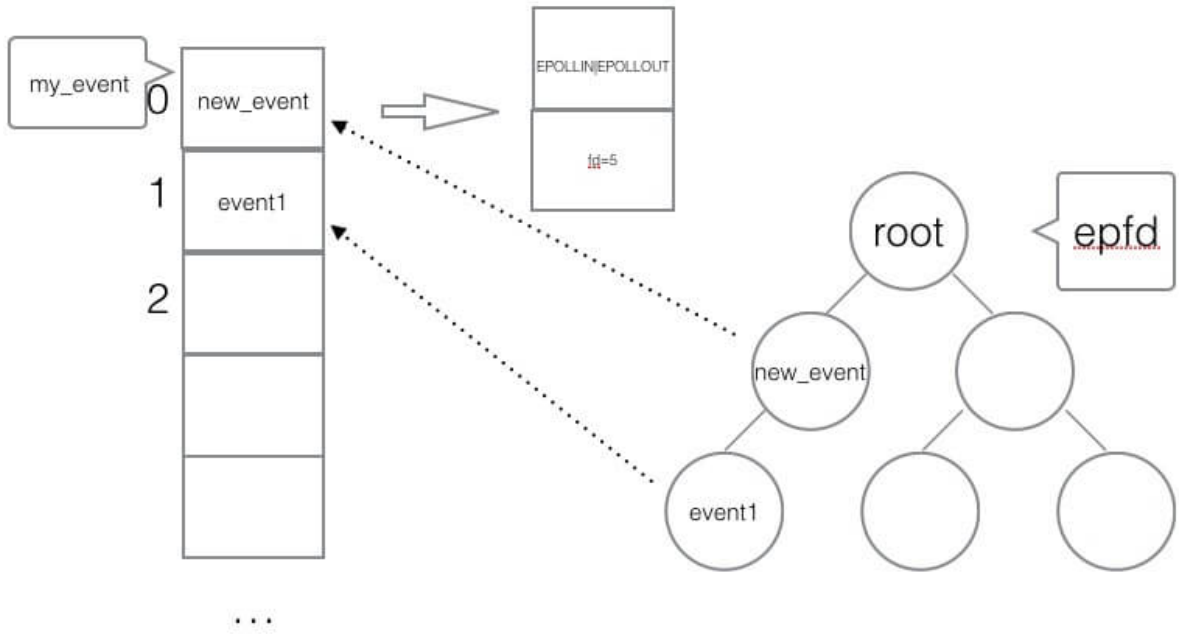
(3) 等待EPOLL

```
/**
*
* @param epfd 用epoll_create所创建的epoll句柄
* @param event 从内核得到的事件集合
* @param maxevents 告知内核这个events有多大,
* 注意: 值 不能大于创建epoll_create()时的size.
* @param timeout 超时时间
* -1: 永久阻塞
* 0: 立即返回，非阻塞
* >0: 指定微秒
*
* @returns 成功: 有多少文件描述符就绪,时间到时返回0
* 失败: -1, errno 查看错误
*/
int epoll_wait(int epfd, struct epoll_event *event,
                            int maxevents, int timeout);
```

使用

```
struct epoll_event my_event[1000];

int event_cnt = epoll_wait(epfd, my_event, 1000, -1);
```

`epoll_wait` 是一个阻塞的状态，如果内核检测到IO的读写响应，会抛给上层的epoll_wait, 返回给用户态一个已经触发的事件队列，同时阻塞返回。开发者可以从队列中取出事件来处理，其中事件里就有绑定的对应fd是哪个(之前添加epoll事件的时候已经绑定)。

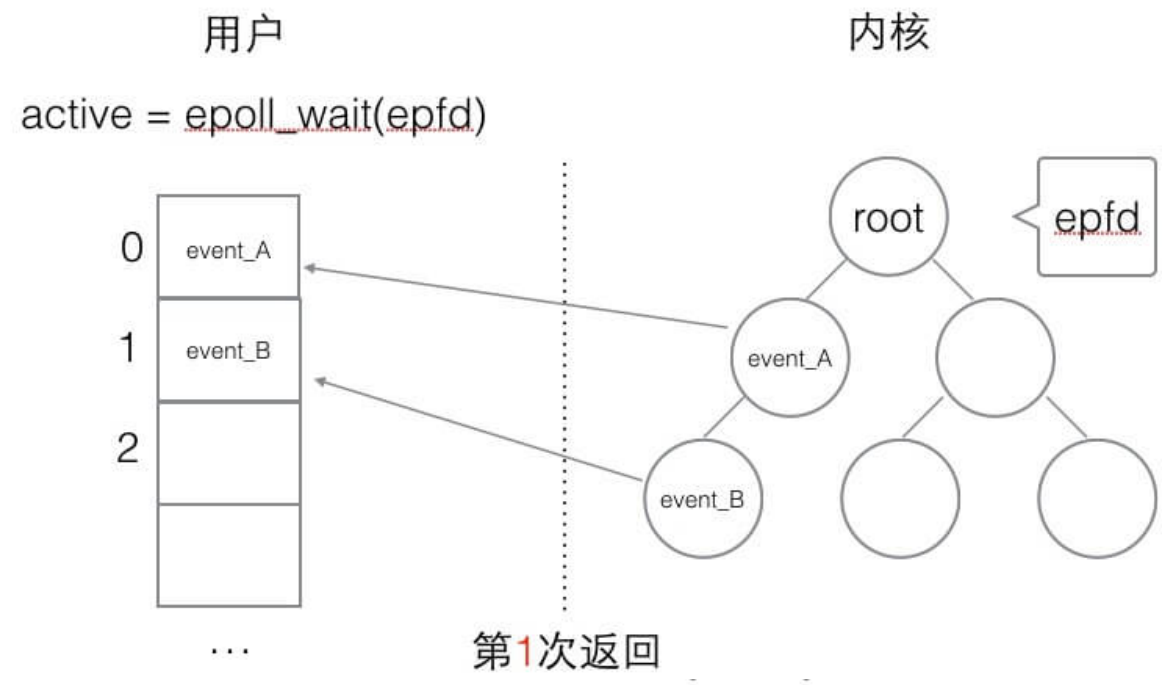


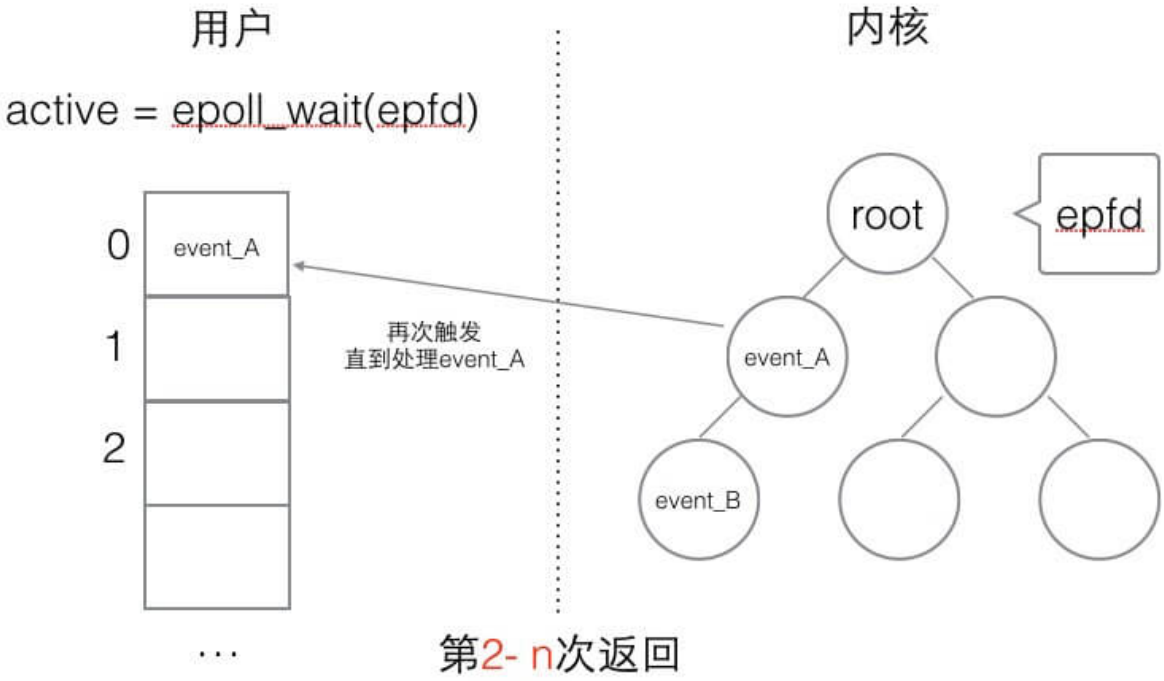
(4) 使用epoll编程主流程骨架

```
int epfd = epoll_crete(1000);

//将 listen_fd 添加进 epoll 中
epoll_ctl(epfd, EPOLL_CTL_ADD, listen_fd,&listen_event);

while (1) {
    //阻塞等待 epoll 中 的fd 触发
    int active_cnt = epoll_wait(epfd, events, 1000, -1);

    for (i = 0 ; i < active_cnt; i++) {
        if (evnets[i].data.fd == listen_fd) {
            //accept. 并且将新accept 的fd 加进epoll中.
        }
        else if (events[i].events & EPOLLIN) {
            //对此fd 进行读操作
        }
        else if (events[i].events & EPOLLOUT) {
            //对此fd 进行写操作
        }
    }
}
```

五、epoll的触发模式

(1) 水平触发

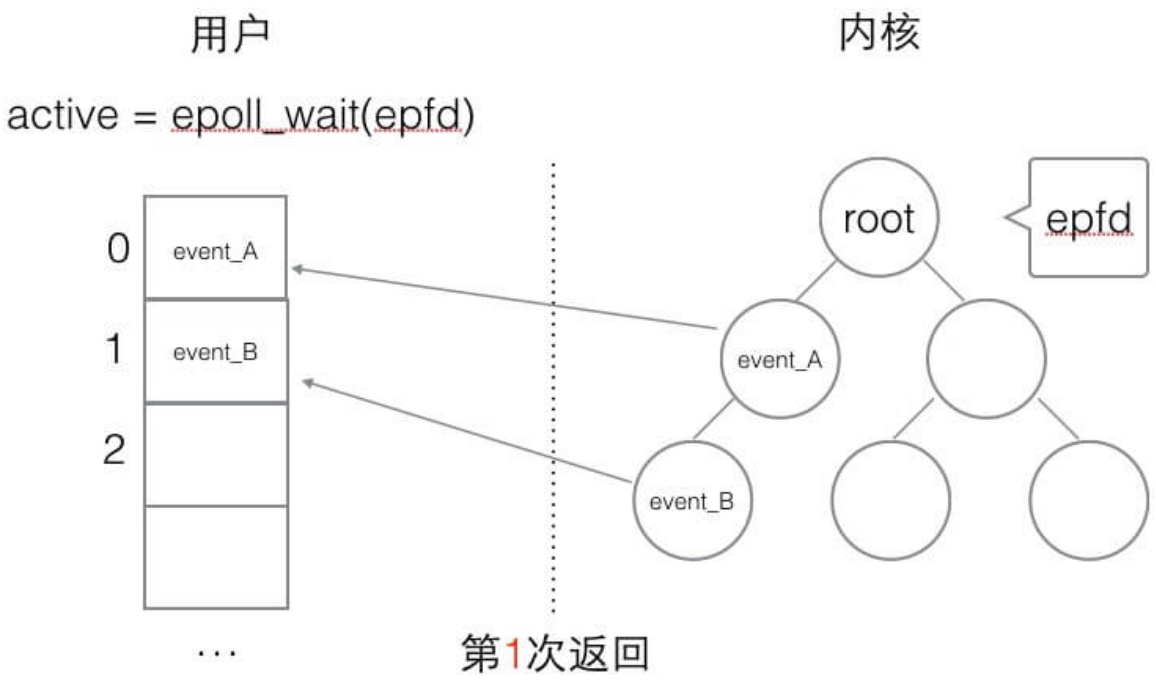


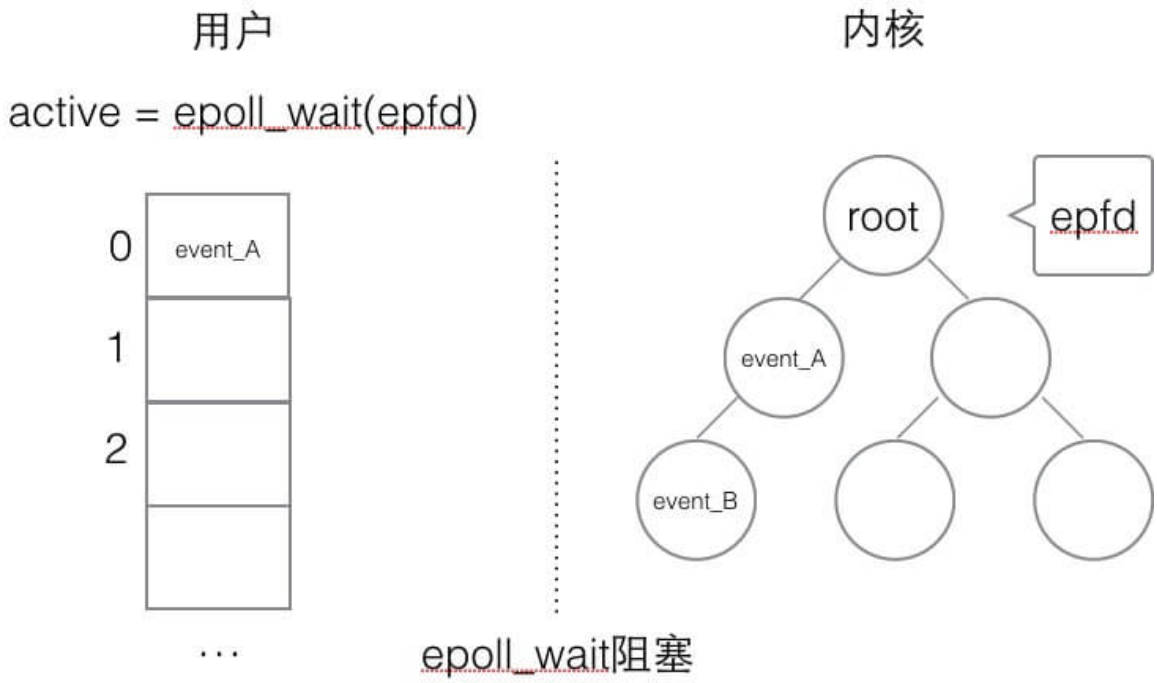


水平触发的主要特点是，如果用户在监听 `epoll` 事件，当内核有事件的时候，会拷贝给用户态事件，但是**如果用户只处理了一次，那么剩下没有处理的会在下一次epoll_wait再次返回该事件**。

这样如果用户永远不处理这个事件，就导致每次都会有该事件从内核到用户的拷贝，耗费性能，但是水平触发相对安全，最起码事件不会丢掉，除非用户处理完毕。

(2) 边缘触发





边缘触发，相对跟水平触发相反，当内核有事件到达， 只会通知用户一次，至于用户处理还是不处理，以后将不会再通知。这样减少了拷贝过程，增加了性能，但是相对来说，如果用户马虎忘记处理，将会产生事件丢的情况。

六、简单的epoll服务器(C语言)

(1) 服务端

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
```

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>

#include <sys/epoll.h>

#define SERVER_PORT (7778)
#define EPOLL_MAX_NUM (2048)
#define BUFFER_MAX_LEN (4096)

char buffer[BUFFER_MAX_LEN];

void str_toupper(char *str)
{
    int i;
    for (i = 0; i < strlen(str); i ++) {
        str[i] = toupper(str[i]);
    }
}

int main(int argc, char **argv)
{
    int listen_fd = 0;
    int client_fd = 0;
    struct sockaddr_in server_addr;
    struct sockaddr_in client_addr;
    socklen_t client_len;

    int epfd = 0;
    struct epoll_event event, *my_events;

    / socket
        listen_fd = socket(AF_INET, SOCK_STREAM, 0);

    // bind
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    server_addr.sin_port = htons(SERVER_PORT);
    bind(listen_fd, (struct sockaddr*)&server_addr, sizeof(server_addr));

    // listen
    listen(listen_fd, 10);

    // epoll create
    epfd = epoll_create(EPOLL_MAX_NUM);
    if (epfd < 0) {
        perror("epoll create");
        goto END;
    }

    // listen_fd -> epoll
    event.events = EPOLLIN;
    event.data.fd = listen_fd;
    if (epoll_ctl(epfd, EPOLL_CTL_ADD, listen_fd, &event) < 0) {
        perror("epoll ctl add listen_fd ");
        goto END;
    }

    my_events = malloc(sizeof(struct epoll_event) * EPOLL_MAX_NUM);
```

```
    while (1) {
        // epoll wait
        int active_fds_cnt = epoll_wait(epfd, my_events, EPOLL_MAX_NUM, -1);
        int i = 0;
        for (i = 0; i < active_fds_cnt; i++) {
            // if fd == listen_fd
            if (my_events[i].data.fd == listen_fd) {
                //accept
                client_fd = accept(listen_fd, (struct sockaddr*)&client_addr, &client_len);
                if (client_fd < 0) {
                    perror("accept");
                    continue;
                }

                char ip[20];
                printf("new connection[%s:%d]\n", inet_ntop(AF_INET, &client_addr.sin_addr, ip, sizeof(ip)),
ntohs(client_addr.sin_port));

                event.events = EPOLLIN | EPOLLET;
                event.data.fd = client_fd;
                epoll_ctl(epfd, EPOLL_CTL_ADD, client_fd, &event);
            }
            else if (my_events[i].events & EPOLLIN) {
                printf("EPOLLIN\n");
                client_fd = my_events[i].data.fd;

                // do read

                buffer[0] = '\0';
                int n = read(client_fd, buffer, 5);
                if (n < 0) {
                    perror("read");
                    continue;
                }
                else if (n == 0) {
                    epoll_ctl(epfd, EPOLL_CTL_DEL, client_fd, &event);
                    close(client_fd);
                }
                else {
                    printf("[read]: %s\n", buffer);
                    buffer[n] = '\0';
#if 1
                    str_toupper(buffer);
                    write(client_fd, buffer, strlen(buffer));
                    printf("[write]: %s\n", buffer);
                    memset(buffer, 0, BUFFER_MAX_LEN);
#endif

                    /*
                       event.events = EPOLLOUT;
                       event.data.fd = client_fd;
                       epoll_ctl(epfd, EPOLL_CTL_MOD, client_fd, &event);
                       */
                }
            }
            else if (my_events[i].events & EPOLLOUT) {
                printf("EPOLLOUT\n");
                /*
                   client_fd = my_events[i].data.fd;
                   str_toupper(buffer);
                   write(client_fd, buffer, strlen(buffer));
                   printf("[write]: %s\n", buffer);
```

```
                    memset(buffer, 0, BUFFER_MAX_LEN);

                    event.events = EPOLLIN;
                    event.data.fd = client_fd;
                    epoll_ctl(epfd, EPOLL_CTL_MOD, client_fd, &event);
                    */
                }
            }
        }

END:
    close(epfd);
    close(listen_fd);
    return 0;
}
```

(2) 客户端

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>

#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <fcntl.h>

#define MAX_LINE (1024)
#define SERVER_PORT (7778)

void setnoblocking(int fd)
{
    int opts = 0;
    opts = fcntl(fd, F_GETFL);
    opts = opts | O_NONBLOCK;
    fcntl(fd, F_SETFL);
}

int main(int argc, char **argv)
{
    int sockfd;
    char recvline[MAX_LINE + 1] = {0};

    struct sockaddr_in server_addr;

    if (argc != 2) {
        fprintf(stderr, "usage ./client <SERVER_IP>\n");
        exit(0);
    }


    // 创建socket
    if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        fprintf(stderr, "socket error");
        exit(0);
    }
```

```
    // server addr 赋值
    bzero(&server_addr, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(SERVER_PORT);

    if (inet_pton(AF_INET, argv[1], &server_addr.sin_addr) <= 0) {
        fprintf(stderr, "inet_pton error for %s", argv[1]);
        exit(0);
    }


    // 链接服务端
    if (connect(sockfd, (struct sockaddr*) &server_addr, sizeof(server_addr)) < 0) {
        perror("connect");
        fprintf(stderr, "connect error\n");
        exit(0);
    }

    setnoblocking(sockfd);

    char input[100];
    int n = 0;
    int count = 0;



    // 不断的从标准输入字符串
    while (fgets(input, 100, stdin) != NULL)
    {
        printf("[send] %s\n", input);
        n = 0;
        // 把输入的字符串发送 到 服务器中去
        n = send(sockfd, input, strlen(input), 0);
        if (n < 0) {
            perror("send");
        }

        n = 0;
        count = 0;


        // 读取 服务器返回的数据
        while (1)
        {
            n = read(sockfd, recvline + count, MAX_LINE);
            if (n == MAX_LINE)
            {
                count += n;
                continue;
            }
            else if (n < 0){
                perror("recv");
                break;
            }
            else {
                count += n;
                recvline[count] = '\0';
                printf("[recv] %s\n", recvline);
                break;
            }
        }
    }
```

```
    return 0;
}
```

Golang的协程调度器原理及GMP设计思想

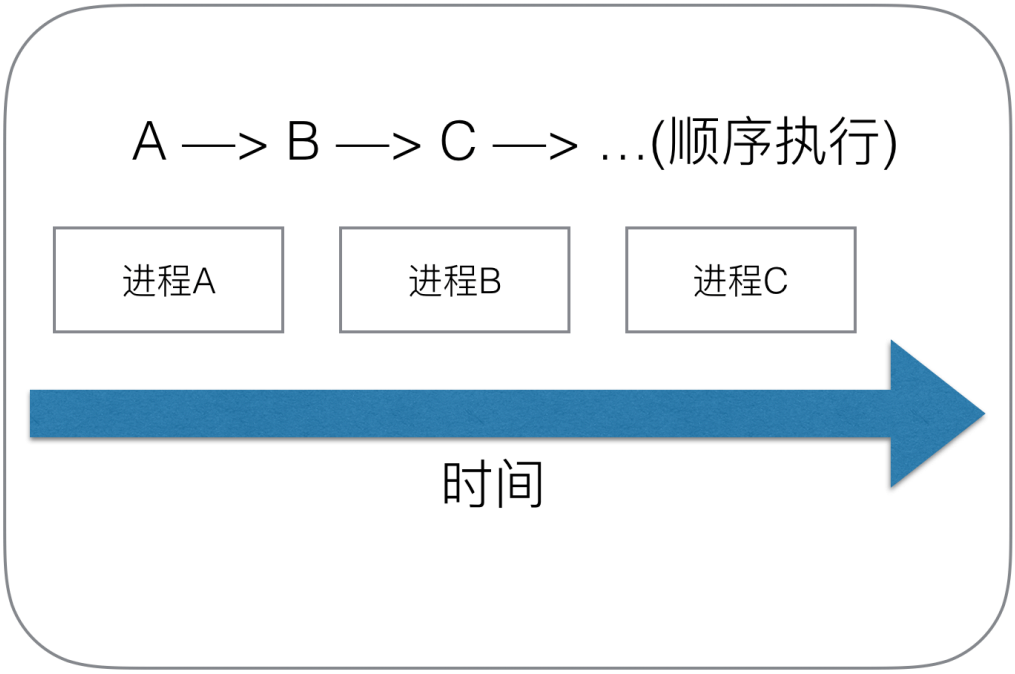
> 本节为**重点**章节
> 本章节含视频版:

一、Golang“调度器”的由来?

(1) 单进程时代不需要调度器

我们知道，一切的软件都是跑在操作系统上，真正用来干活(计算)的是CPU。早期的操作系统每个程序就是一个进程，知道一个程序运行完，才能进行下一个进程，就是“单进程时代”

一切的程序只能串行发生。



早期单进程操作系统

早期的单进程操作系统，面临2个问题：

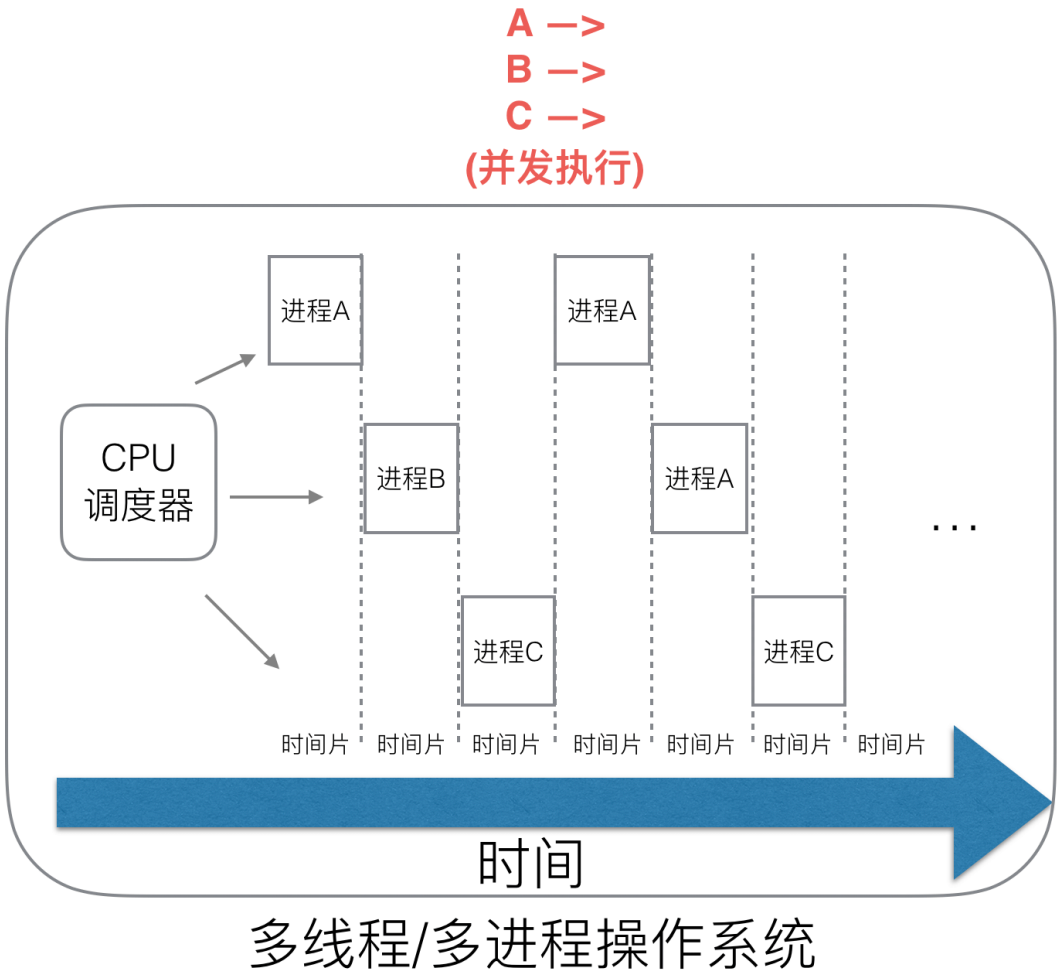
1.单一的执行流程，计算机只能一个任务一个任务处理。

2.进程阻塞所带来的CPU时间浪费。

那么能不能有多个进程来宏观一起来执行多个任务呢？

后来操作系统就具有了**最早的并发能力：多进程并发**，当一个进程阻塞的时候，切换到另外等待执行的进程，这样就能尽量把CPU利用起来，CPU就不浪费了。

(2)多进程/线程时代有了调度器需求

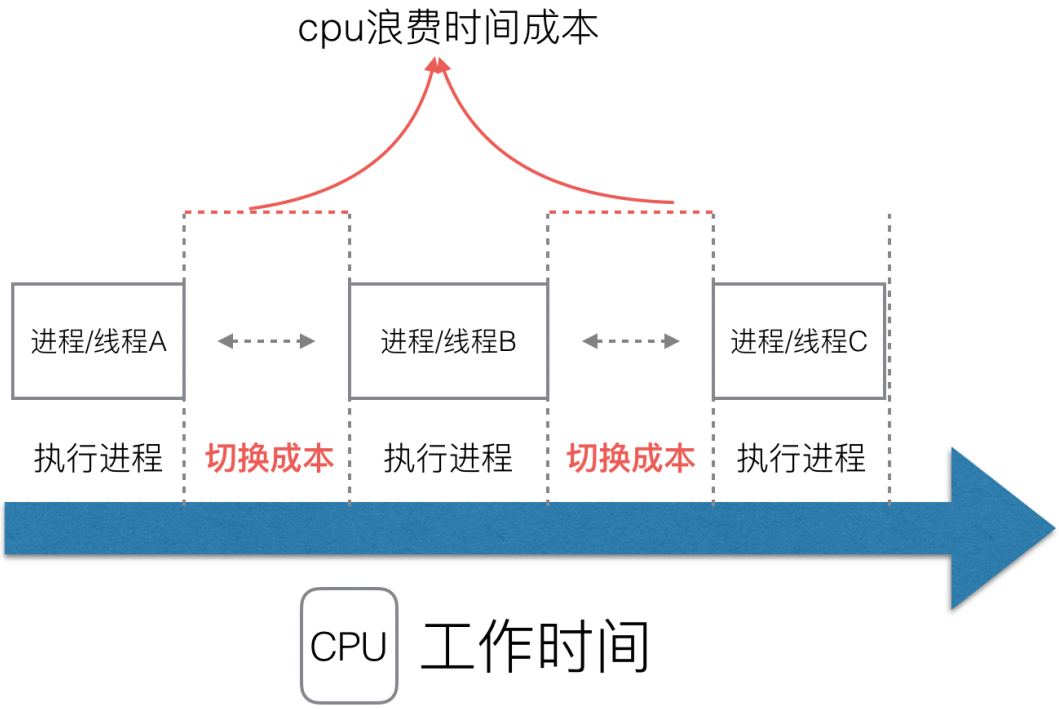


在多进程/多线程的操作系统中，就解决了阻塞的问题，因为一个进程阻塞cpu可以立刻切换到其他进程中去执行，而且调度cpu的算法可以保证在运行的进程都可以被分配到cpu的运行时间片。这样从宏观来看，似乎多个进程是在同时被运行。

但新的问题就又出现了，进程拥有太多的资源，进程的创建、切换、销毁，都会占用很长的时间，CPU虽然利用起来了，但如果进程过多，CPU有很大的一部分都被用来进行进程调度了。

怎么才能提高CPU的利用率呢？

但是对于Linux操作系统来讲，cpu对进程的态度和线程的态度是一样的。



很明显，CPU调度切换的是进程和线程。尽管线程看起来很美好，但实际上多线程开发设计会变得更加复杂，要考虑很多同步竞争等问题，如锁、竞争冲突等。

(3)协程来提高CPU利用率

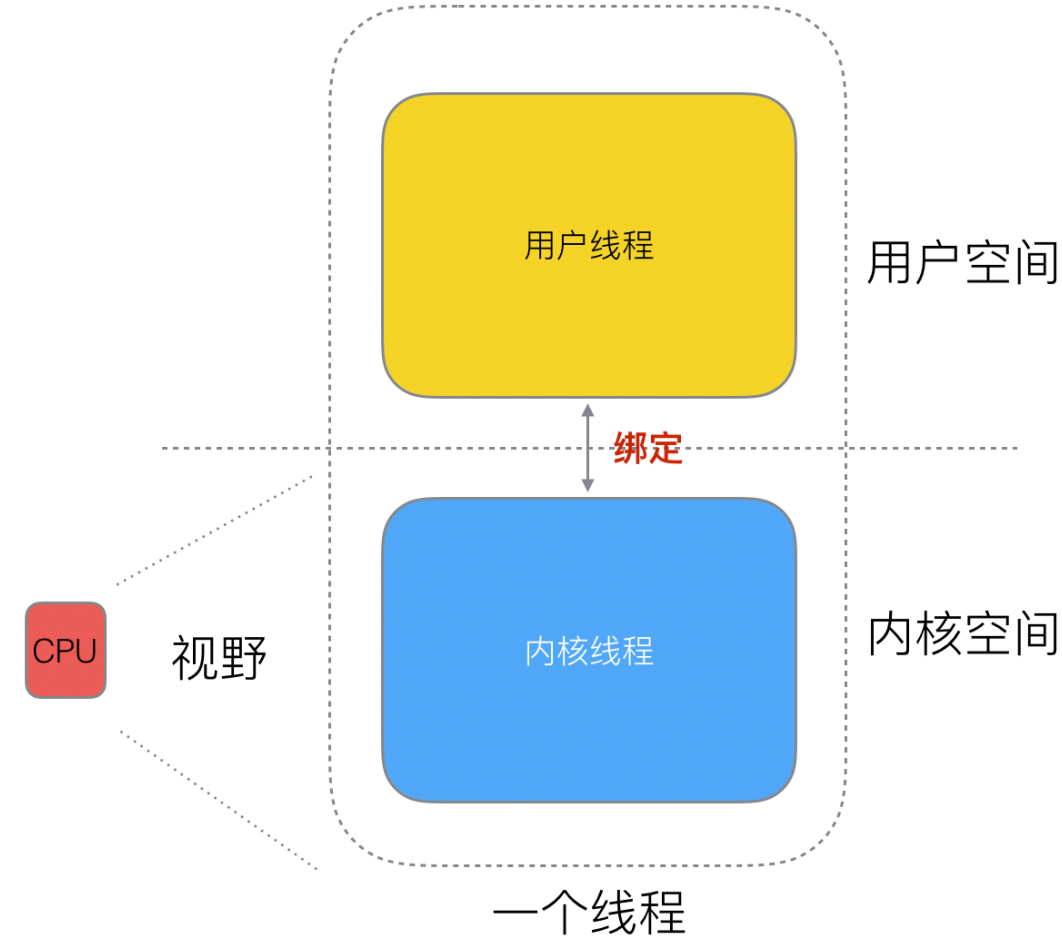
多进程、多线程已经提高了系统的并发能力，但是在当今互联网高并发场景下，为每个任务都创建一个线程是不现实的，因为会消耗大量的内存(进程虚拟内存会占用4GB[32位操作系统], 而线程也要大约4MB)。

大量的进程/线程出现了新的问题

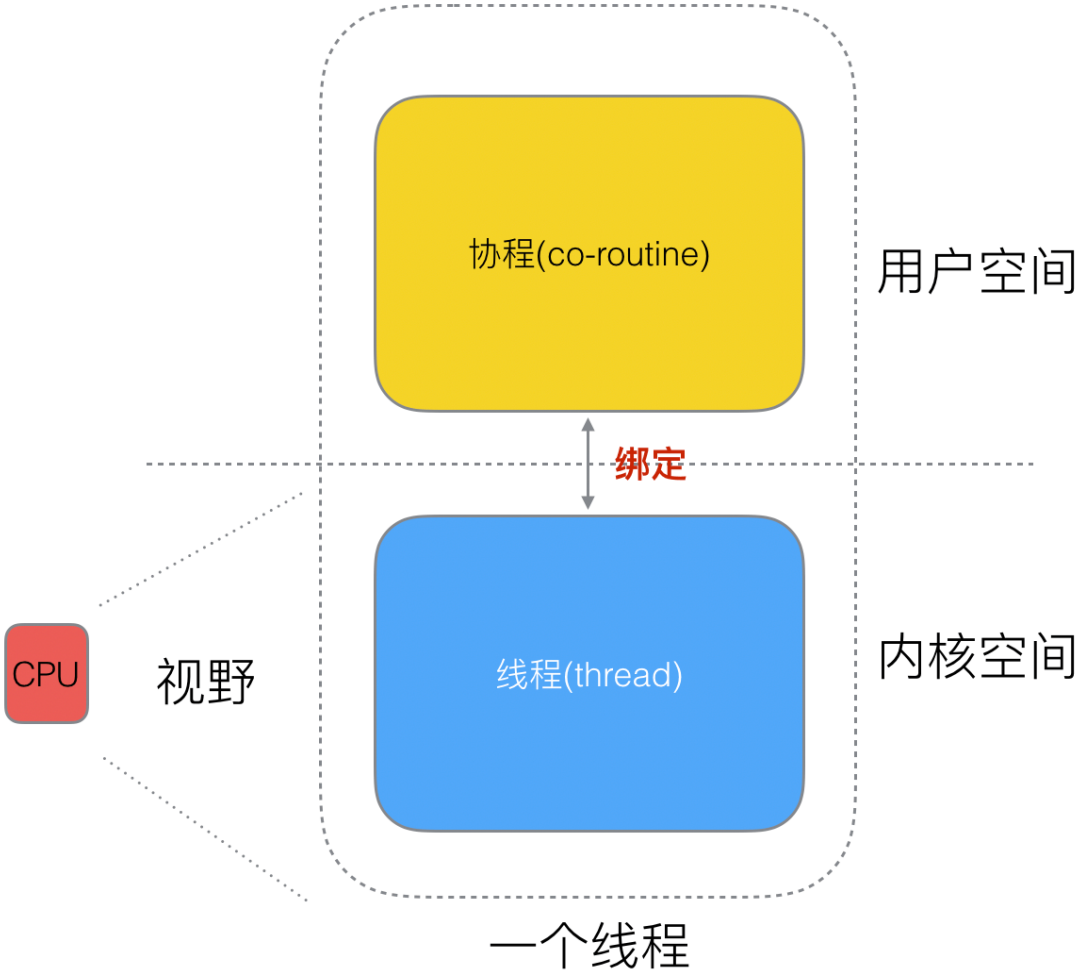
- 高内存占用
- 调度的高消耗CPU

好了，然后工程师们就发现，其实一个线程分为“内核态“线程和”用户态“线程。

一个“用户态线程”必须要绑定一个“内核态线程”，但是CPU并不知道有“用户态线程”的存在，它只知道它运行的是一个“内核态线程”(Linux的PCB进程控制块)。



这样，我们再去细化去分类一下，内核线程依然叫“线程(thread)”，用户线程叫“协程(co-routine)”.



看到这里，我们就要开脑洞了，既然一个协程(co-routine)可以绑定一个线程(thread)，那么能不能多个协程(co-routine)绑定一个或者多个线程(thread)上呢。

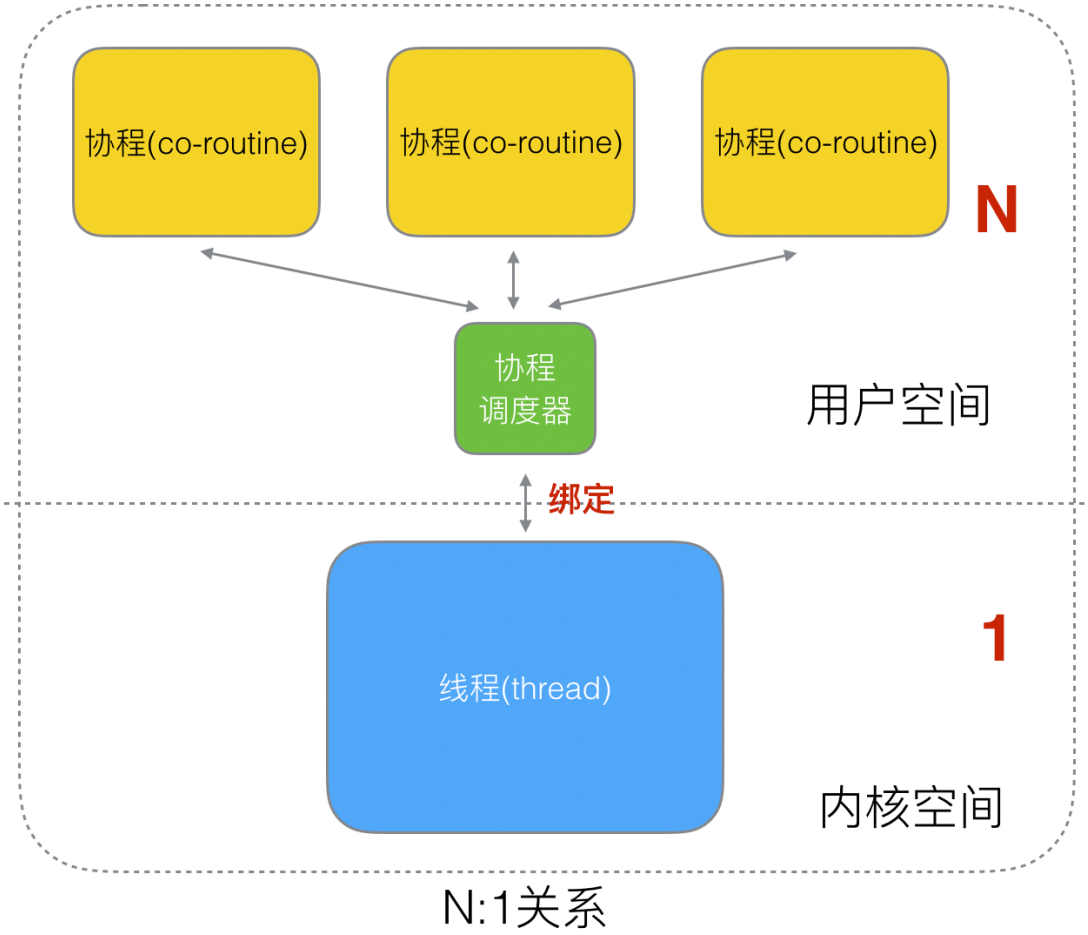
之后，我们就看到了有3中协程和线程的映射关系：

> **N:1关系**

N个协程绑定1个线程，优点就是**协程在用户态线程即完成切换，不会陷入到内核态，这种切换非常的轻量快速**。但也有很大的缺点，1个进程的所有协程都绑定在1个线程上

缺点：

- 某个程序用不了硬件的多核加速能力
- 一旦某协程阻塞，造成线程阻塞，本进程的其他协程都无法执行了，根本就没有并发的能力了。



> **1:1 关系**

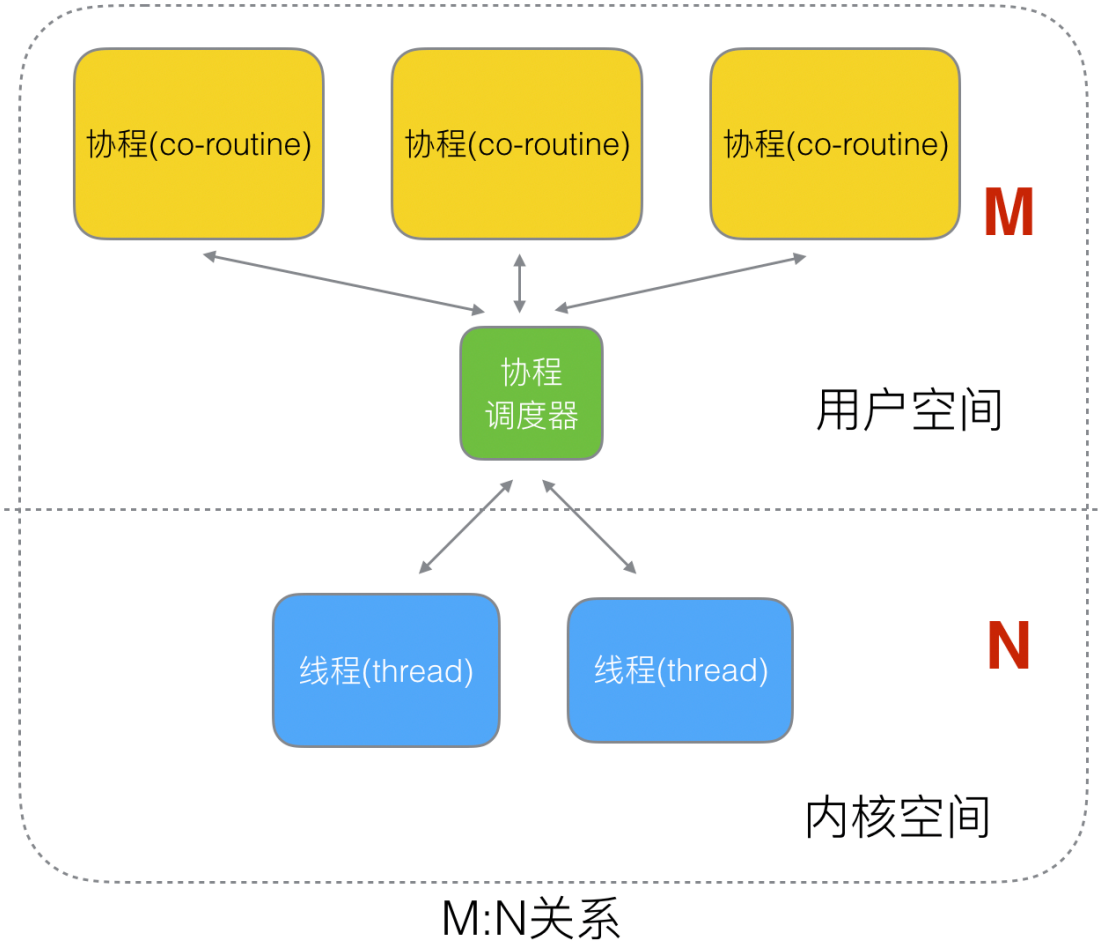
1个协程绑定1个线程，这种最容易实现。协程的调度都由CPU完成了，不存在N:1缺点，

缺点：

- 协程的创建、删除和切换的代价都由CPU完成，有点略显昂贵了。

> **M:N关系**

M个协程绑定1个线程，是N:1和1:1类型的结合，克服了以上2种模型的缺点，但实现起来最为复杂。



M:N关系

协程跟线程是有区别的，线程由CPU调度是抢占式的，**协程由用户态调度是协作式的**，一个协程让出CPU后，才执行下一个协程。

(4)Go语言的协程goroutine

Go为了提供更容易使用的并发方法，使用了goroutine和channel。goroutine来自协程的概念，让一组可复用的函数运行在一组线程之上，即使有协程阻塞，该线程的其他协程也可以被 `runtime` 调度，转移到其他可运行的线程上。最关键的是，程序员看不到这些底层的细节，这就降低了编程的难度，提供了更容易的并发。

Go中，协程被称为goroutine，它非常轻量，一个goroutine只占几KB，并且这几KB就足够goroutine运行完，这就能在有限的内存空间内支持大量goroutine，支持了更多的并发。虽然一个goroutine的栈只占几KB，但实际是可伸缩的，如果需要更多内容， `runtime` 会自动为goroutine分配。

Goroutine特点：

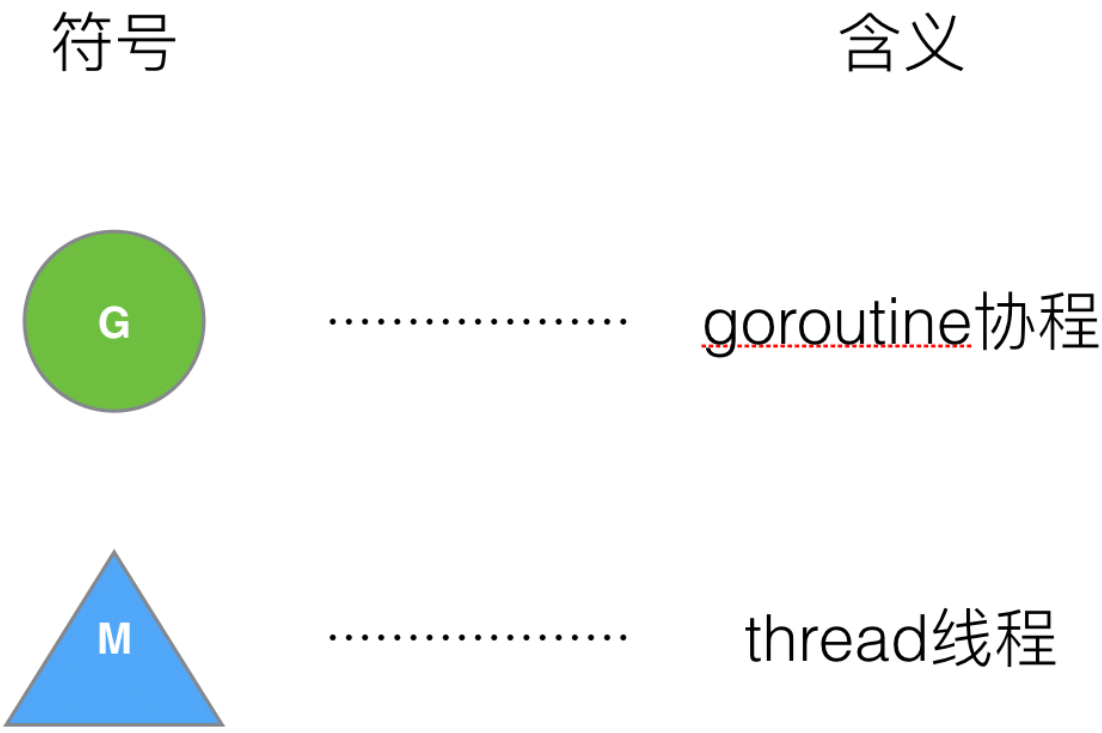
- 占用内存更小（几kb）
- 调度更灵活(runtime调度)

(5)被废弃的goroutine调度器

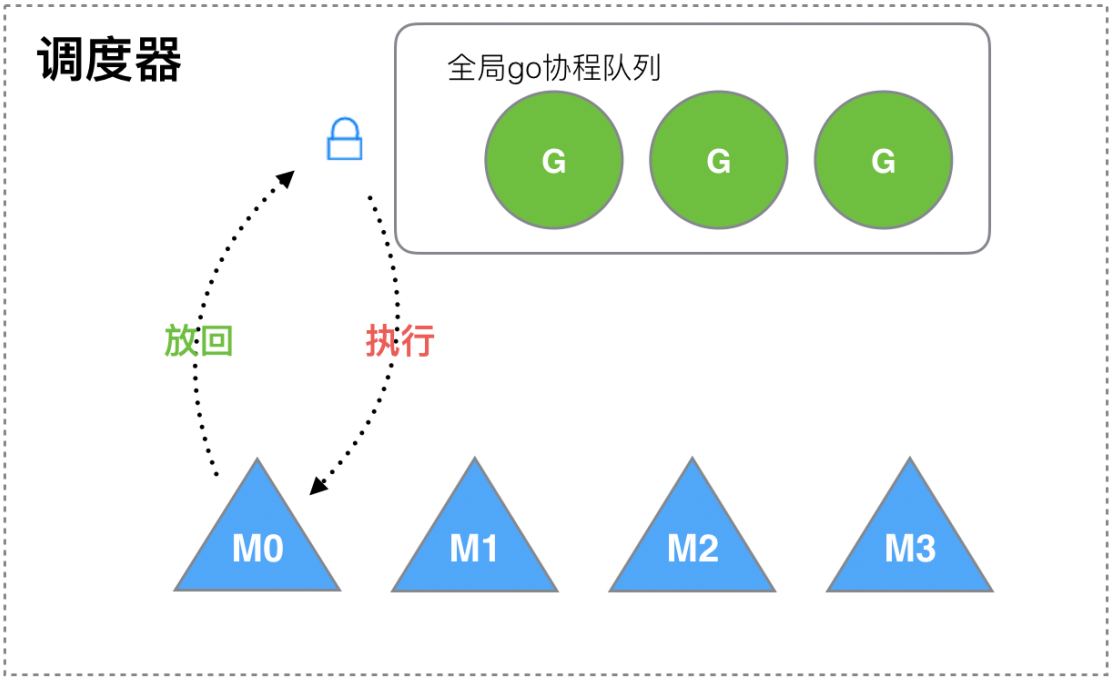
好了，既然我们知道了协程和线程的关系，那么最关键的一点就是调度协程的调度器的实现了。

Go目前使用的调度器是2012年重新设计的，因为之前的调度器性能存在问题，所以使用4年就被废弃了，那么我们先来分析一下被废弃的调度器是如何运作的？

> 大部分文章都是会用G来表示Goroutine，用M来表示线程，那么我们也会用这种表达的对应关系。



下面我们来看看被废弃的golang调度器是如何实现的？



M想要执行、放回G都必须访问全局G队列，并且M有多个，即多线程访问同一资源需要加锁进行保证互斥/同步，所以全局G队列是有互斥锁进行保护的。

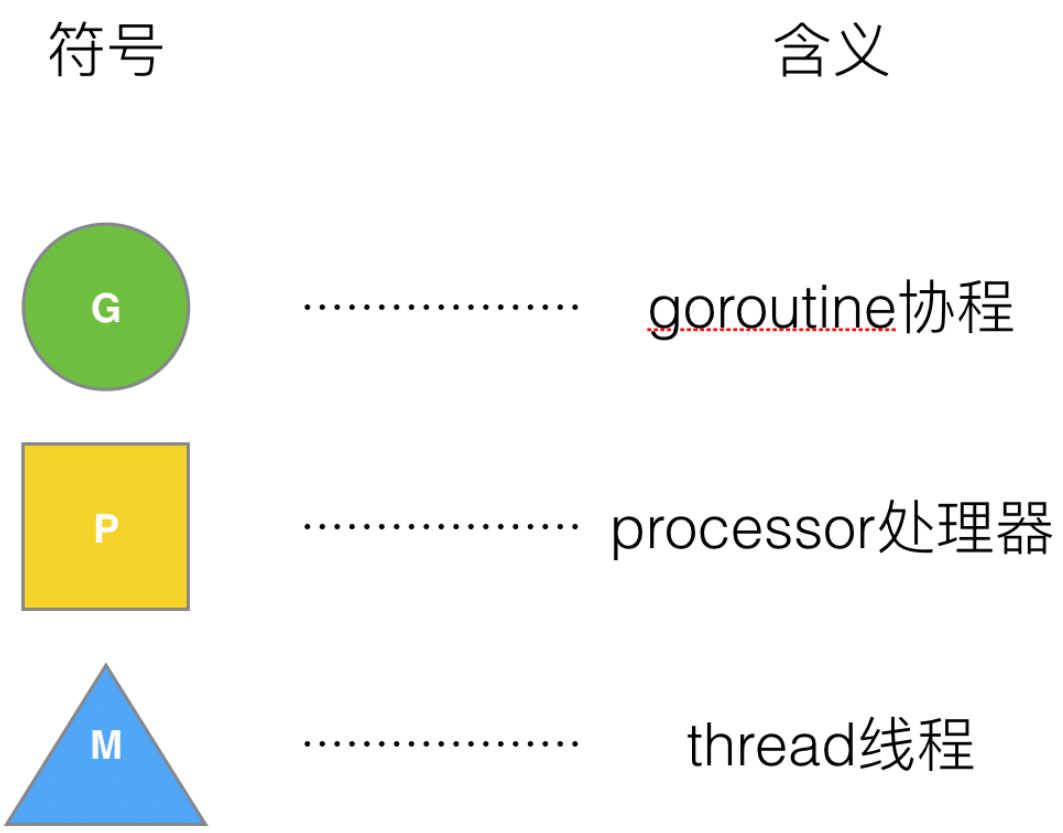
老调度器有几个缺点：

1. 创建、销毁、调度G都需要每个M获取锁，这就形成了**激烈的锁竞争**。
2. M转移G会造成**延迟和额外的系统负载**。比如当G中包含创建新协程的时候，M创建了G’，为了继续执行G，需要把G’交给M’执行，也造成了**很差的局部性**，因为G’和G是相关的，最好放在M上执行，而不是其他M’。
3. 系统调用(CPU在M之间的切换)导致频繁的线程阻塞和取消阻塞操作增加了系统开销。

二、Goroutine调度器的GMP模型的设计思想

面对之前调度器的问题，Go设计了新的调度器。

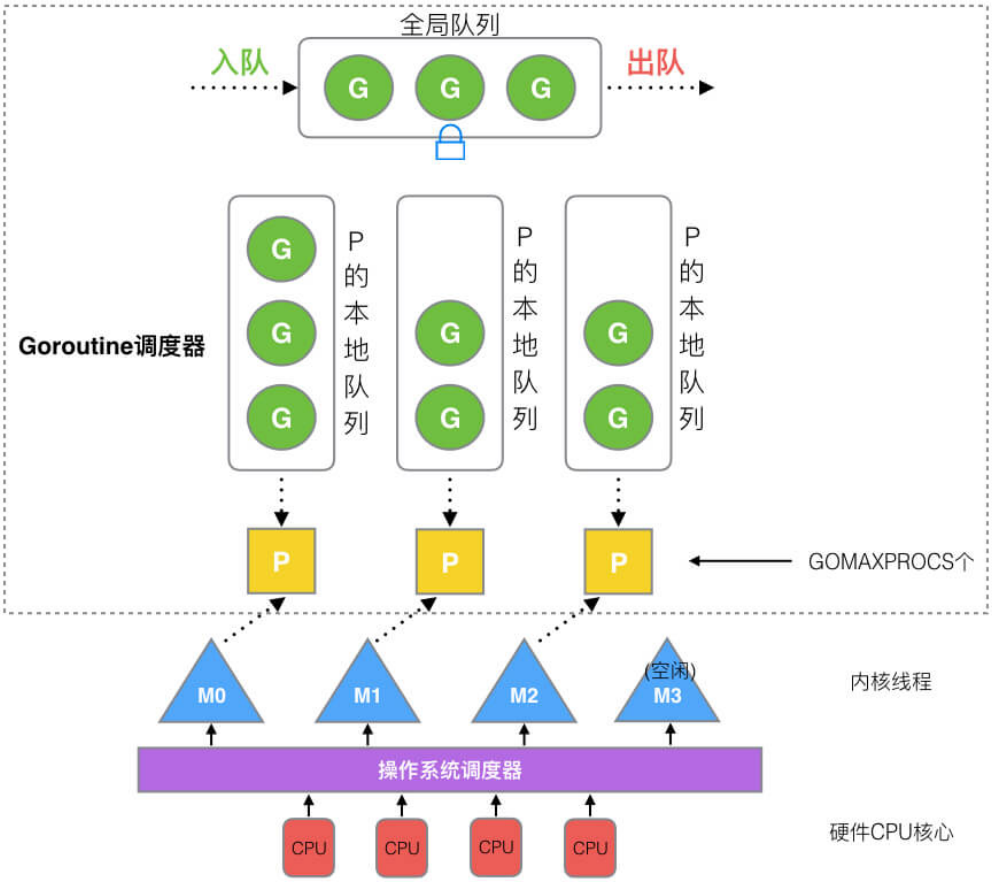
在新调度器中，出列M(thread)和G(goroutine)，又引进了P(Processor)。



Processor，它包含了运行goroutine的资源，如果线程想运行goroutine，必须先获取P，P中还包含了可运行的G队列。

(1)GMP模型

在Go中，**线程是运行goroutine的实体，调度器的功能是把可运行的goroutine分配到工作线程上**。



1. **全局队列**（Global Queue）：存放等待运行的G。
2. **P的本地队列**：同全局队列类似，存放的也是等待运行的G，存的数量有限，不超过256个。新建G'时，G'优先加入到P的本地队列，如果队列满了，则会把本地队列中一半的G移动到全局队列。
3. **P列表**：所有的P都在程序启动时创建，并保存在数组中，最多有`GOMAXPROCS`(可配置)个。
4. **M**：线程想运行任务就得获取P，从P的本地队列获取G，P队列为空时，M也会尝试从全局队列**拿**一批G放到P的本地队列，或从其他P的本地队列**偷**一半放到自己P的本地队列。M运行G，G执行之后，M会从P获取下一个G，不断重复下去。

Goroutine调度器和OS调度器是通过M结合起来的，每个M都代表了1个内核线程，OS调度器负责把内核线程分配到CPU的核上执行。

> **有关P和M的个数问题**

1、P的数量：

- 由启动时环境变量`$GOMAXPROCS`或者是由`runtime`的方法`GOMAXPROCS()`决定。这意味着在程序执行的任意时刻都只有`$GOMAXPROCS`个goroutine在同时运行。

2、M的数量:

- go语言本身的限制：go程序启动时，会设置M的最大数量，默认10000.但是内核很难支持这么多的线程数，所以这个限制可以忽略。
- runtime/debug中的SetMaxThreads函数，设置M的最大数量
- 一个M阻塞了，会创建新的M。

M与P的数量没有绝对关系，一个M阻塞，P就会去创建或者切换另一个M，所以，即使P的默认数量是1，也有可能会创建很多个M出来。

> **P和M何时会被创建**

1、P何时创建：在确定了P的最大数量n后，运行时系统会根据这个数量创建n个P。

2、M何时创建：没有足够的M来关联P并运行其中的可运行的G。比如所有的M此时都阻塞住了，而P中还有很多就绪任务，就会去寻找空闲的M，而没有空闲的，就会去创建新的M。

(2)调度器的设计策略

复用线程：避免频繁的创建、销毁线程，而是对线程的复用。

1）work stealing机制

当本线程无可运行的G时，尝试从其他线程绑定的P偷取G，而不是销毁线程。

2）hand off机制

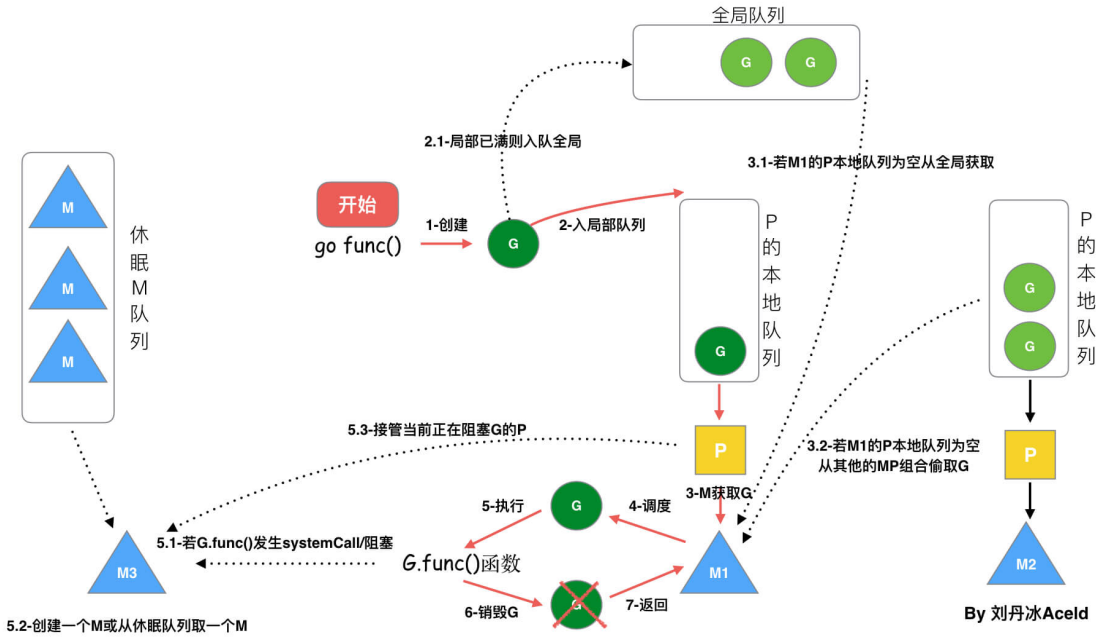
当本线程因为G进行系统调用阻塞时，线程释放绑定的P，把P转移给其他空闲的线程执行。

利用并行：`GOMAXPROCS` 设置P的数量，最多有 `GOMAXPROCS` 个线程分布在多个CPU上同时运行。 `GOMAXPROCS` 也限制了并发的程度，比如 `GOMAXPROCS = 核数/2` ，则最多利用了一半的CPU核进行并行。

抢占：在coroutine中要等待一个协程主动让出CPU才执行下一个协程，在Go中，一个goroutine最多占用CPU 10ms，防止其他goroutine被饿死，这就是goroutine不同于coroutine的一个地方。

全局G队列：在新的调度器中依然有全局G队列，但功能已经被弱化了，当M执行work stealing从其他P偷不到G时，它可以从全局G队列获取G。

(3) go func() 调度流程



从上图我们可以分析出几个结论：

1、我们通过 go func()来创建一个goroutine；

2、有两个存储G的队列，一个是局部调度器P的本地队列、一个是全局G队列。新创建的G会先保存在P的本地队列中，如果P的本地队列已经满了就会保存在全局的队列中；

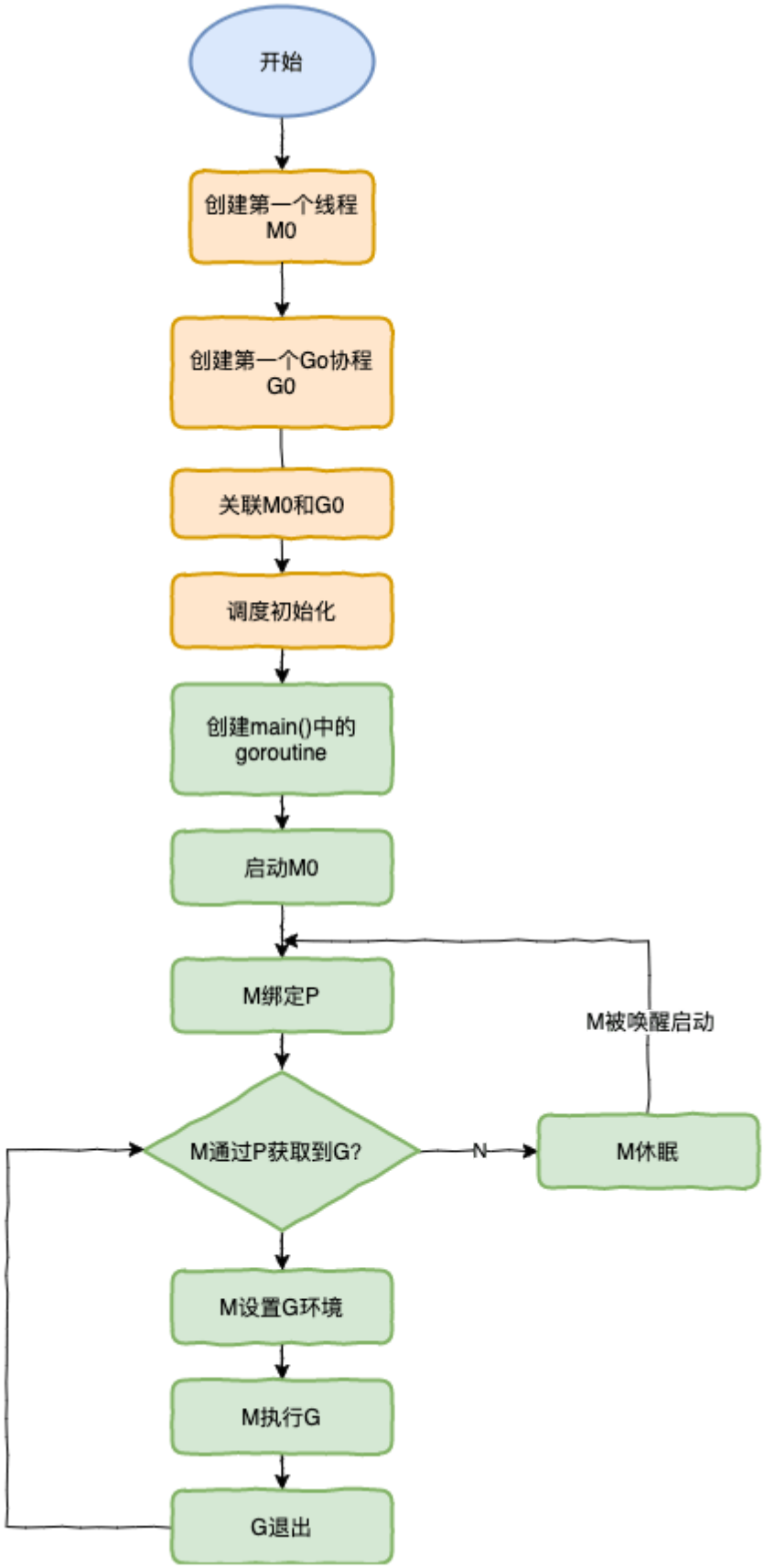
3、G只能运行在M中，一个M必须持有一个P，M与P是1：1的关系。M会从P的本地队列弹出一个可执行状态的G来执行，如果P的本地队列为空，就会想其他的MP组合偷取一个可执行的G来执行；

4、一个M调度G执行的过程是一个循环机制；

5、当M执行某一个G时候如果发生了syscall或则其余阻塞操作，M会阻塞，如果当前有一些G在执行，runtime会把这个线程M从P中摘除(detach)，然后再创建一个新的操作系统的线程(如果有空闲的线程可用就复用空闲线程)来服务于这个P；

6、当M系统调用结束时候，这个G会尝试获取一个空闲的P执行，并放入到这个P的本地队列。如果获取不到P，那么这个线程M变成休眠状态， 加入到空闲线程中，然后这个G会被放入全局队列中。

(4)调度器的生命周期


开始
创建第一个线程 M0
创建第一个Go协程 G0
关联M0和G0
调度初始化
创建main()中的 goroutine
启动M0
M绑定P
M通过P获取到G?
N
M休眠
M被唤醒启动
M设置G环境
M执行G
G退出

特殊的M0和G0

M0

`M0` 是启动程序后的编号为0的主线程，这个M对应的实例会在全局变量runtime.m0中，不需要在heap上分配，M0负责执行初始化操作和启动第一个G， 在之后M0就和其他的M一样了。

G0

`G0` 是每次启动一个M都会第一个创建的gourtine，G0仅用于负责调度的G，G0不指向任何可执行的函数, 每个M都会有一个自己的G0。在调度或系统调用时会使用G0的栈空间, 全局变量的G0是M0的G0。

我们来跟踪一段代码

```
package main

import "fmt"

func main() {
    fmt.Println("Hello world")
}
```

接下来我们来针对上面的代码对调度器里面的结构做一个分析。

也会经历如上图所示的过程：

1. runtime创建最初的线程m0和goroutine g0，并把2者关联。
2. 调度器初始化：初始化m0、栈、垃圾回收，以及创建和初始化由GOMAXPROCS个P构成的P列表。
3. 示例代码中的main函数是 `main.main` ， `runtime` 中也有1个main函数——`runtime.main` ，代码经过编译后， `runtime.main` 会调用 `main.main` ，程序启动时会为 `runtime.main` 创建goroutine，称它为main goroutine吧，然后把main goroutine加入到P的本地队列。
4. 启动m0，m0已经绑定了P，会从P的本地队列获取G，获取到main goroutine。
5. G拥有栈，M根据G中的栈信息和调度信息设置运行环境
6. M运行G
7. G退出，再次回到M获取可运行的G，这样重复下去，直到 `main.main` 退出， `runtime.main` 执行Defer和Panic处理，或调用 `runtime.exit` 退出程序。

调度器的生命周期几乎占满了一个Go程序的一生， `runtime.main` 的goroutine执行之前都是为调度器做准备工作， `runtime.main` 的goroutine运行，才是调度器的真正开始，直到 `runtime.main` 结束而结束。

(5)可视化GMP编程

有2种方式可以查看一个程序的GMP的数据。

方式1：go tool trace

trace记录了运行时的信息，能提供可视化的Web页面。

简单测试代码：main函数创建trace，trace会运行在单独的goroutine中，然后main打印”Hello World”退出。

> trace.go

```
package main

import (
    "os"
    "fmt"
    "runtime/trace"
)
```

```
func main() {

    //创建trace文件
    f, err := os.Create("trace.out")
    if err != nil {
        panic(err)
    }

    defer f.Close()

    //启动trace goroutine
    err = trace.Start(f)
    if err != nil {
        panic(err)
    }
    defer trace.Stop()

    //main
    fmt.Println("Hello World")
}
```

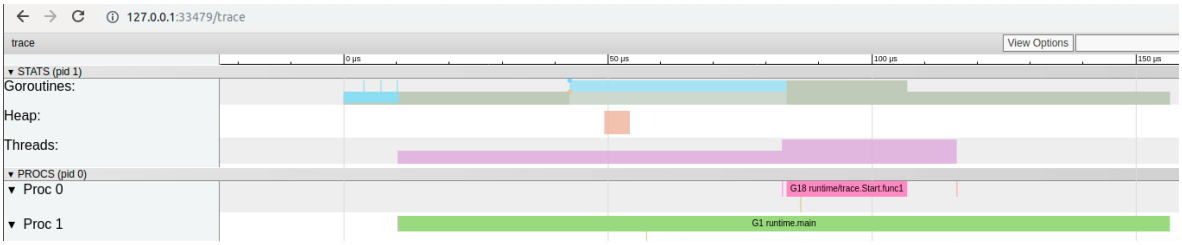
运行程序

```
$ go run trace.go
Hello World
```

会得到一个 `trace.out` 文件，然后我们可以用一个工具打开，来分析这个文件。

```
$ go tool trace trace.out
2020/02/23 10:44:11 Parsing trace...
2020/02/23 10:44:11 Splitting trace...
2020/02/23 10:44:11 Opening browser. Trace viewer is listening on http://127.0.0.1:33479
```

我们可以通过浏览器打开 `http://127.0.0.1:33479` 网址，点击 `view trace` 能够看见可视化的调度流程。



G信息

点击Goroutines那一行可视化的数据条，我们会看到一些详细的信息。

3 items selected. | Counter Samples (3)

Counter	Series	Time	Value	
Goroutines	GCWaiting	0.042808	0	G0
Goroutines	Runnable	0.042808	1	
Goroutines	Running	0.042808	1	G1

一共有两个G在程序中，一个是特殊的G0，是每个M必须有的一个初始化的G，这个我们不必讨论。

其中G1应该就是main goroutine(执行main函数的协程)，在一段时间内处于可运行和运行的状态。

M信息

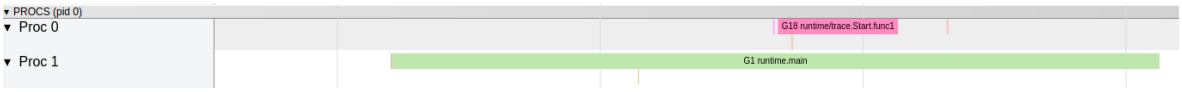
点击Threads那一行可视化的数据条，我们会看到一些详细的信息。

2 items selected. | Counter Samples (2)

Counter	Series	Time	Value
Threads	InSyscall	0.010201	0
Threads	Running	0.010201	1

一共有两个M在程序中，一个是特殊的M0，用于初始化使用，这个我们不必讨论。

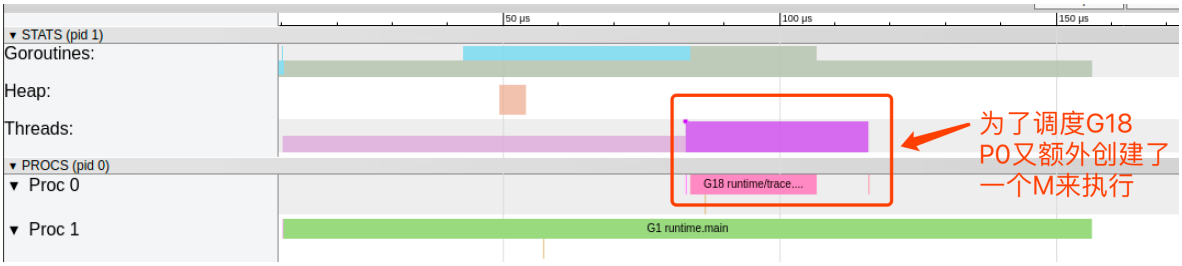
P信息



G1中调用了 `main.main` ，创建了 `trace goroutine g18` 。G1运行在P1上，G18运行在P0上。

这里有两个P，我们知道，一个P必须绑定一个M才能调度G。

我们在来看看上面的M信息。



我们会发现，确实G18在P0上被运行的时候，确实在Threads行多了一个M的数据，点击查看如下：

2 items selected. | Counter Samples (2)

Counter	Series	Time	Value
Threads	InSyscall	0.083032	0
Threads	Running	0.083032	2

多了一个M2应该就是P0为了执行G18而动态创建的M2.

方式2：Debug trace

```
package main

import (
    "fmt"
    "time"
)

func main() {
```

```
    for i := 0; i < 5; i++ {
        time.Sleep(time.Second)
        fmt.Println("Hello World")
    }
}
```

编译

```
$ go build trace2.go
```

通过Debug方式运行

```
$ GODEBUG=schedtrace=1000 ./trace2
SCHED 0ms: gomaxprocs=2 idleprocs=0 threads=4 spinningthreads=1 idlethreads=1 runqueue=0 [0 0]
Hello World
SCHED 1003ms: gomaxprocs=2 idleprocs=2 threads=4 spinningthreads=0 idlethreads=2 runqueue=0 [0 0]
Hello World
SCHED 2014ms: gomaxprocs=2 idleprocs=2 threads=4 spinningthreads=0 idlethreads=2 runqueue=0 [0 0]
Hello World
SCHED 3015ms: gomaxprocs=2 idleprocs=2 threads=4 spinningthreads=0 idlethreads=2 runqueue=0 [0 0]
Hello World
SCHED 4023ms: gomaxprocs=2 idleprocs=2 threads=4 spinningthreads=0 idlethreads=2 runqueue=0 [0 0]
Hello World
```

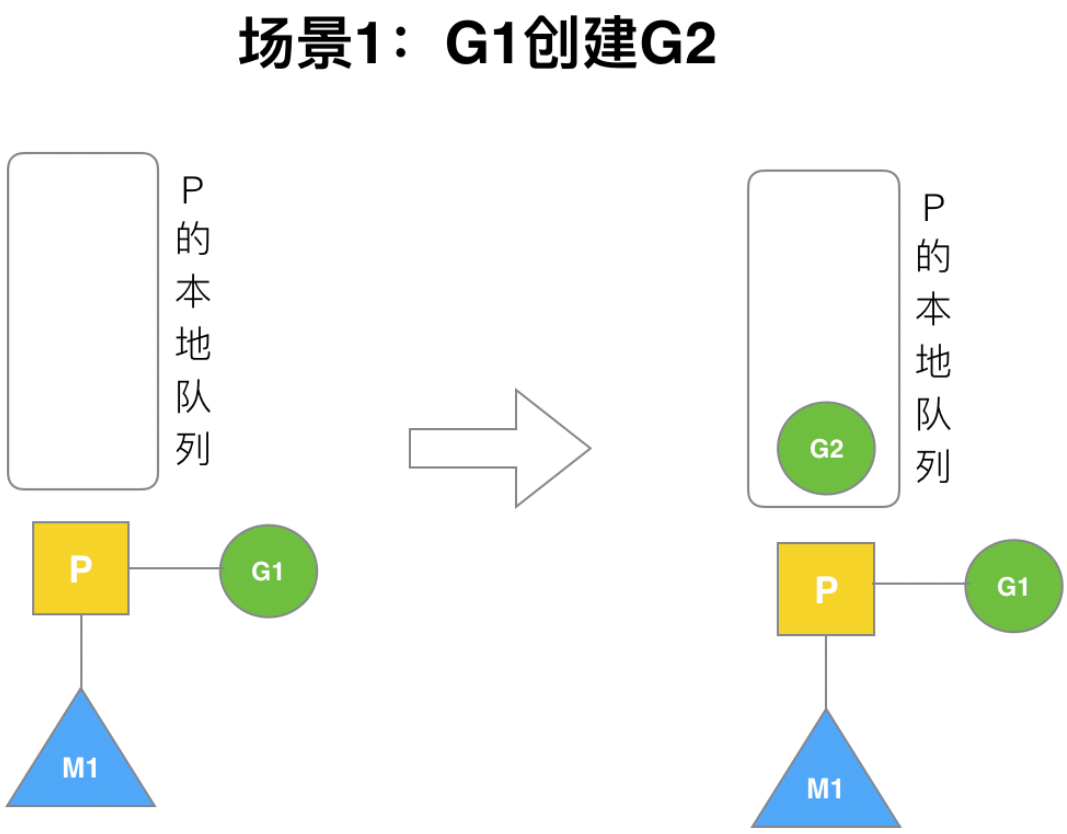
- `SCHED`：调试信息输出标志字符串，代表本行是goroutine调度器的输出；
- `0ms`：即从程序启动到输出这行日志的时间；
- `gomaxprocs`: P的数量，本例有2个P, 因为默认的P的属性是和cpu核心数量默认一致，当然也可以通过GOMAXPROCS来设置；
- `idleprocs`: 处于idle状态的P的数量；通过gomaxprocs和idleprocs的差值，我们就可知道执行go代码的P的数量；
- t`hreads: os threads/M`的数量，包含scheduler使用的m数量，加上runtime自用的类似sysmon这样的thread的数量；
- `spinningthreads`: 处于自旋状态的os thread数量；
- `idlethread`: 处于idle状态的os thread的数量；
- `runqueue=0`： Scheduler全局队列中G的数量；
- `[0 0]`: 分别为2个P的local queue中的G的数量。

下一篇，我们来继续详细的分析GMP调度原理的一些场景问题。

三、Go调度器调度场景过程全解析

(1)场景1

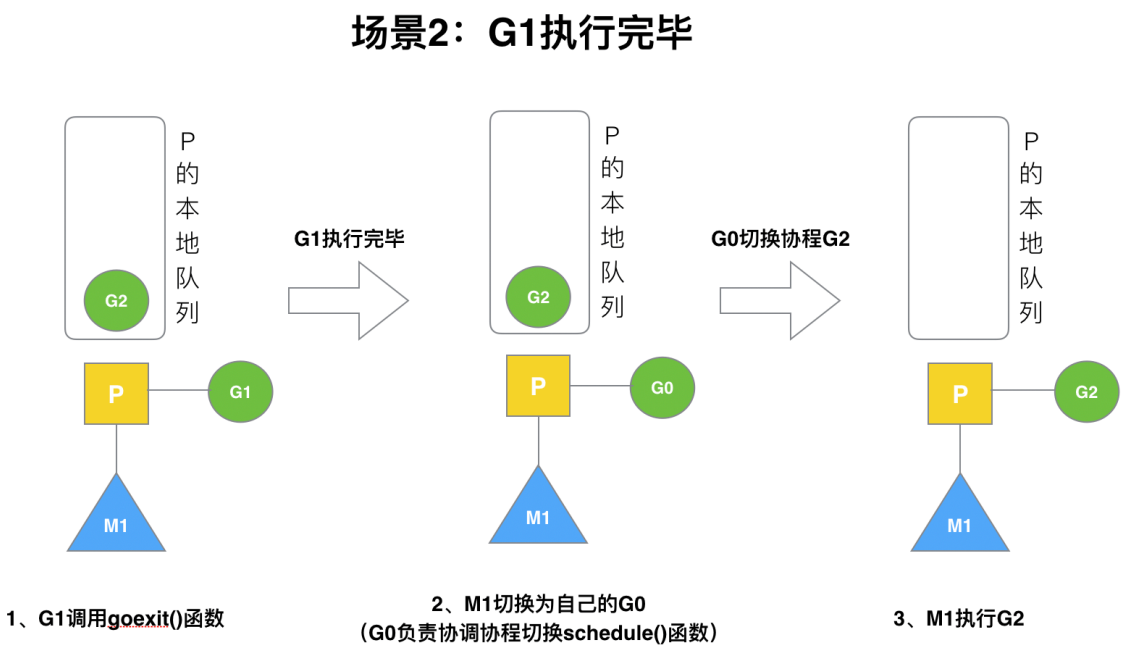
P拥有G1，M1获取P后开始运行G1，G1使用 `go func()` 创建了G2，为了局部性G2优先加入到P1的本地队列。



(2)场景2

G1运行完成后(函数：`goexit`)，M上运行的goroutine切换为G0，G0负责调度时协程的切换（函数：`schedule`）。从P的本地队列取G2，从G0切换到G2，并开始运行G2(函数：`execute`)。实现了线程M1的复

用。

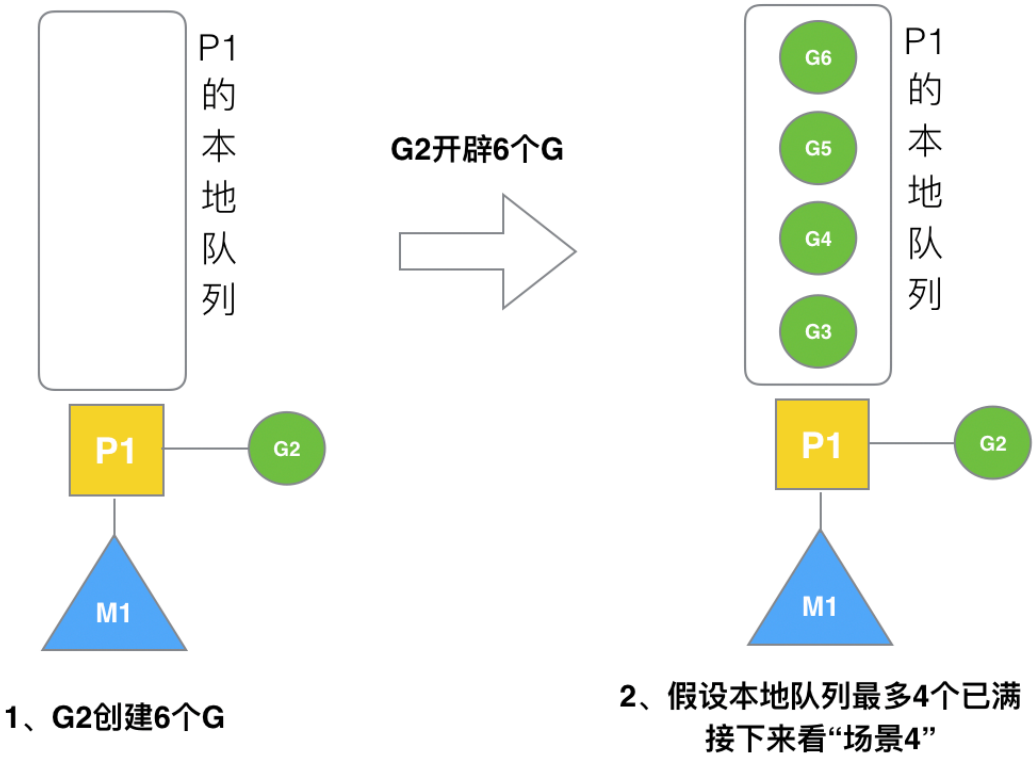

场景2：G1执行完毕
P
的
本
地
队
列
G2
P
G1
M1
G1执行完毕
P
的
本
地
队
列
G2
P
G0
M1
G0切换协程G2
P
的
本
地
队
列
P
G2
M1
1、G1调用goexit()函数
2、M1切换为自己的G0
（G0负责协调协程切换schedule()函数）
3、M1执行G2

---

(3)场景3

假设每个P的本地队列只能存3个G。G2要创建了6个G，前3个G（G3, G4, G5）已经加入p1的本地队列，p1本地队列满了。

场景3：G2开辟过多的G

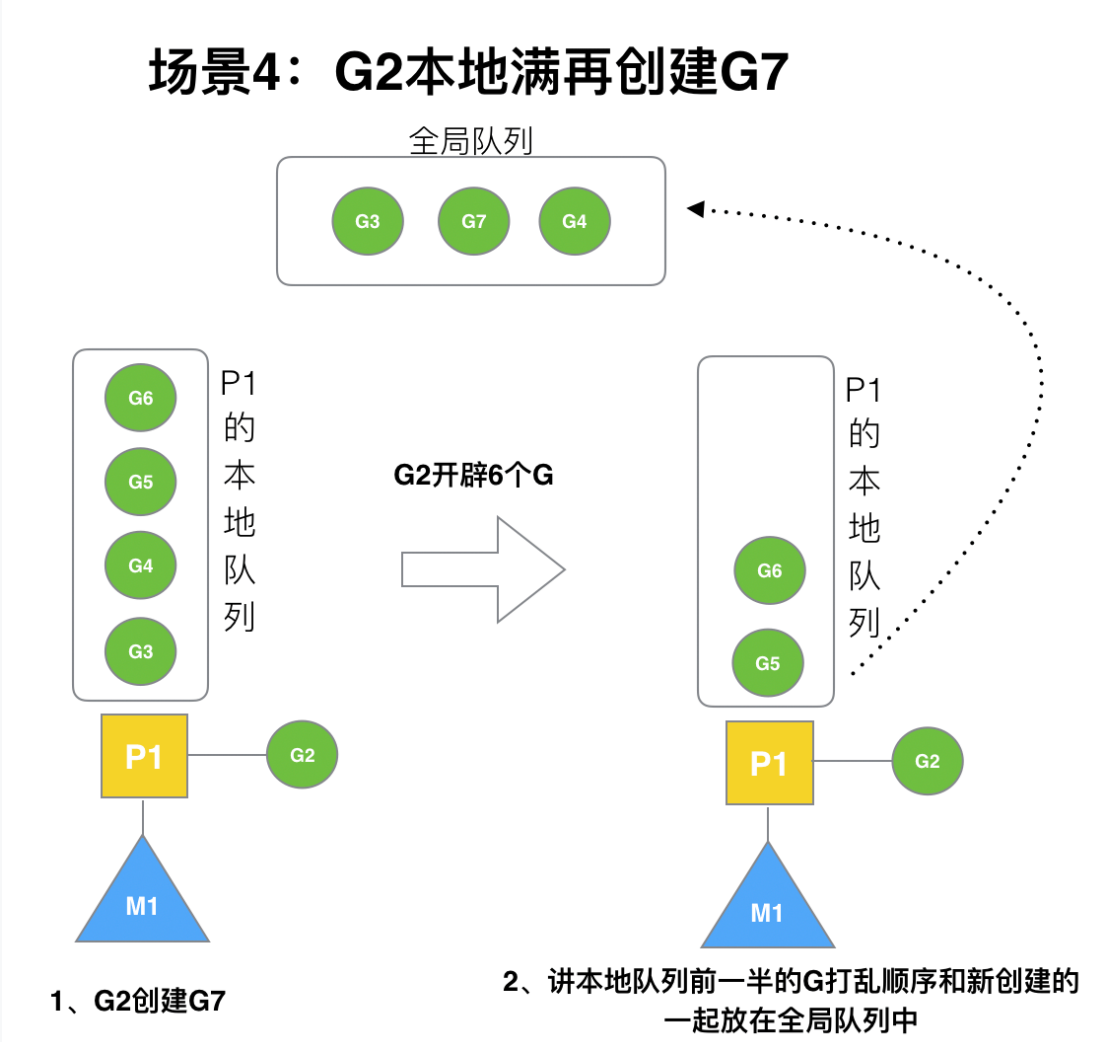


---

(4)场景4

G2在创建G7的时候，发现P1的本地队列已满，需要执行**负载均衡**(把P1中本地队列中前一半的G，还有新创建G**转移**到全局队列)

（实现中并不一定是新的G，如果G是G2之后就执行的，会被保存在本地队列，利用某个老的G替换新G加入全局队列）

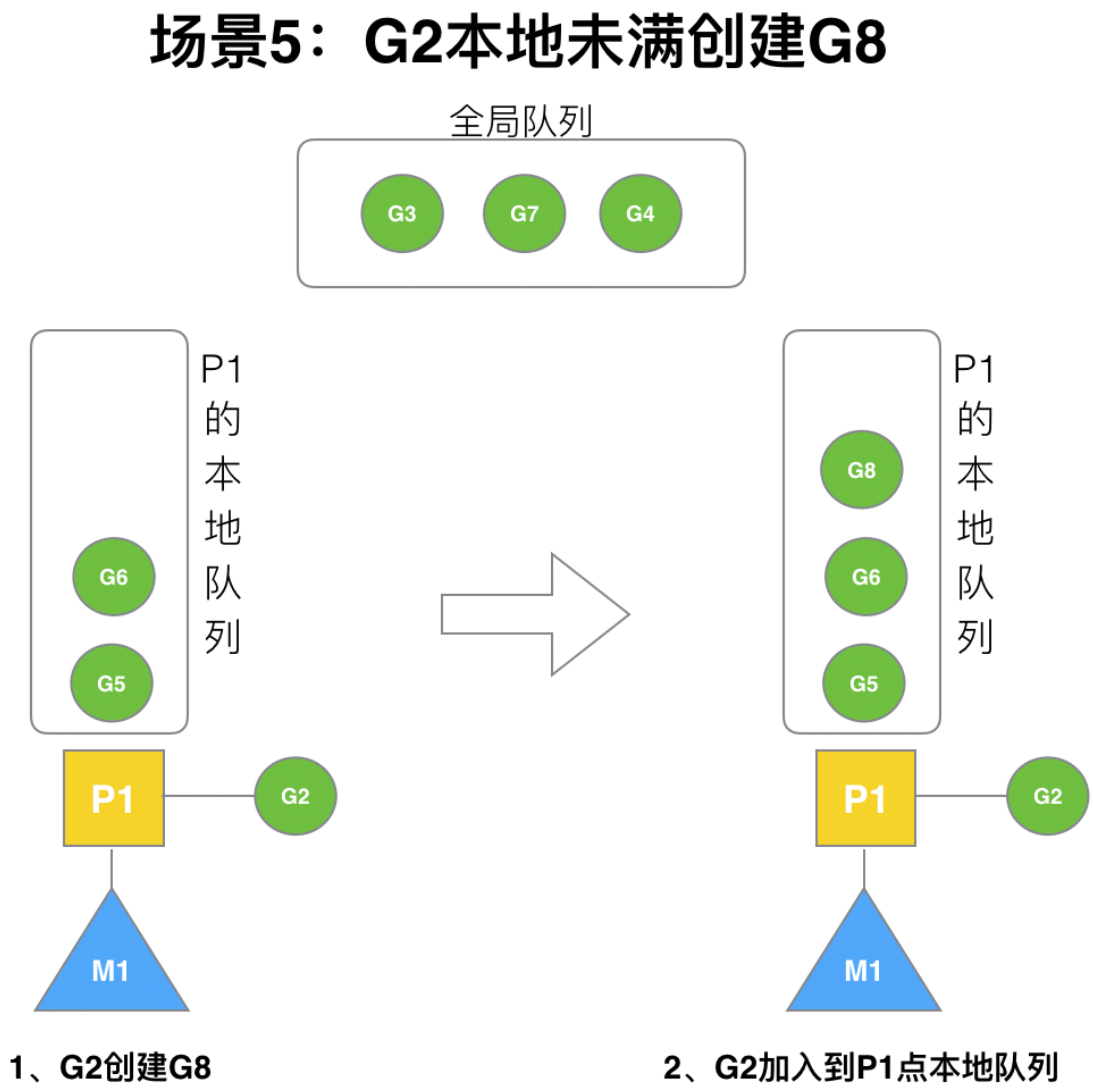


这些G被转移到全局队列时，会被打乱顺序。所以G3,G4,G7被转移到全局队列。

---

(5)场景5

G2创建G8时，P1的本地队列未满，所以G8会被加入到P1的本地队列。

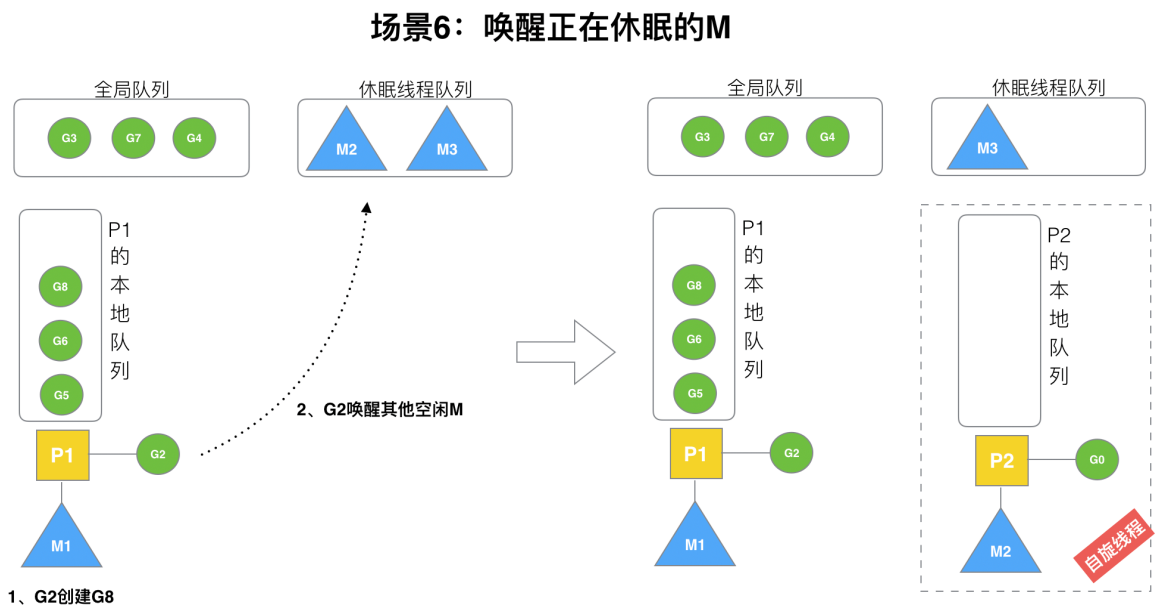


G8加入到P1点本地队列的原因还是因为P1此时在与M1绑定，而G2此时是M1在执行。所以G2创建的新的G会优先放置到自己的M绑定的P上。

---

(6)场景6

规定：**在创建G时，运行的G会尝试唤醒其他空闲的P和M组合去执行。**



假定G2唤醒了M2，M2绑定了P2，并运行G0，但P2本地队列没有G，M2此时为自旋线程（**没有G但为运行状态的线程，不断寻找G**）。

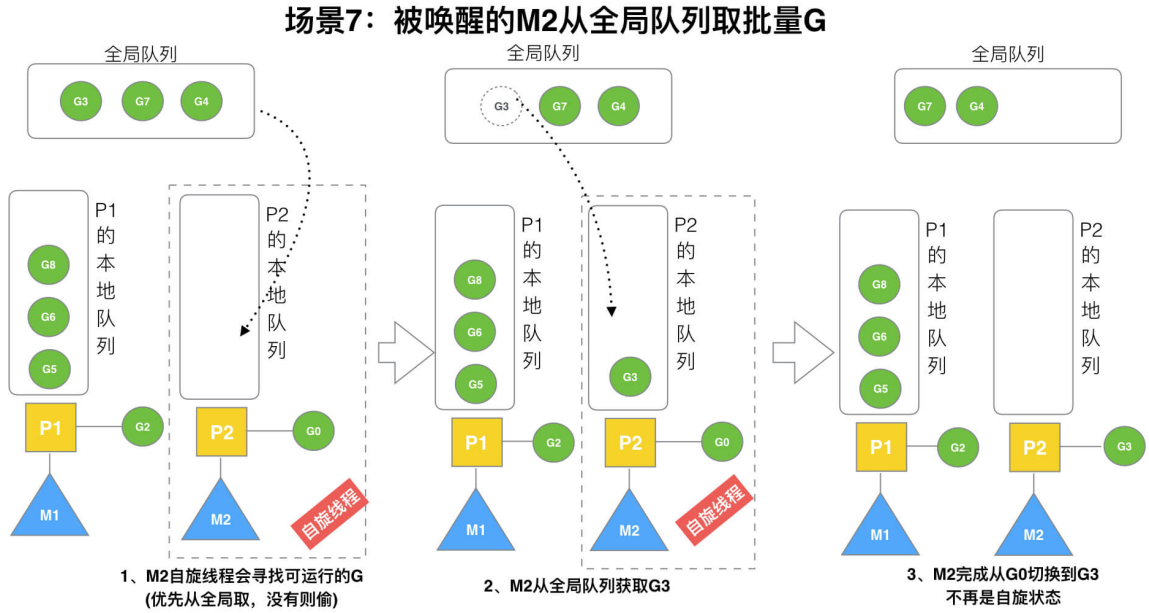
---

(7)场景7

M2尝试从全局队列(简称“GQ”)取一批G放到P2的本地队列（函数：`findrunnable()`）。M2从全局队列取的G数量符合下面的公式：

```
n = min(len(GQ)/GOMAXPROCS + 1, len(GQ/2))
```

至少从全局队列取1个g，但每次不要从全局队列移动太多的g到p本地队列，给其他p留点。这是**从全局队列到P本地队列的负载均衡**。

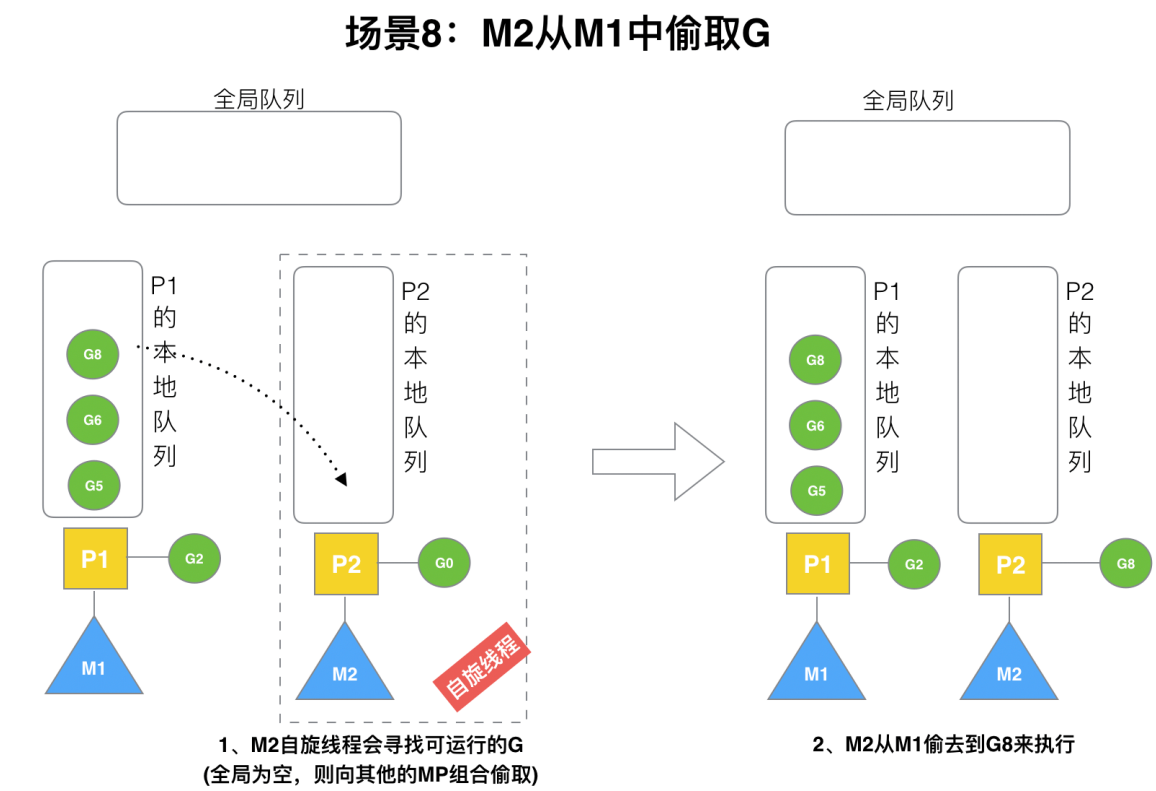


假定我们场景中一共有4个P（GOMAXPROCS设置为4，那么我们允许最多就能用4个P来供M使用）。所以M2只从能从全局队列取1个G（即G3）移动P2本地队列，然后完成从G0到G3的切换，运行G3。

---

(8)场景8

假设G2一直在M1上运行，经过2轮后，M2已经把G7、G4从全局队列获取到了P2的本地队列并完成运行，全局队列和P2的本地队列都空了,如场景8图的左半部分。

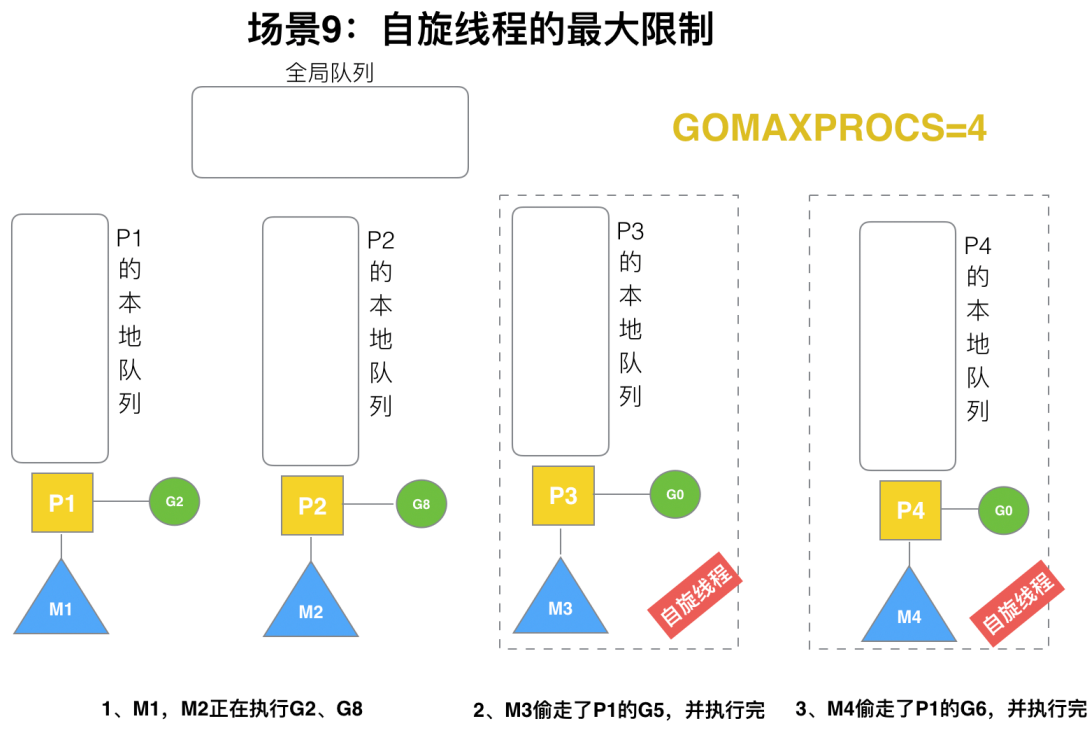


全局队列已经没有G，那m就要执行work stealing(偷取)：从其他有G的P哪里偷取一半G过来，放到自己的P本地队列。P2从P1的本地队列尾部取一半的G，本例中一半则只有1个G8，放到P2的本地队列并执行。

---

(9)场景9

G1本地队列G5、G6已经被其他M偷走并运行完成，当前M1和M2分别在运行G2和G8，M3和M4没有goroutine可以运行，M3和M4处于**自旋状态**，它们不断寻找goroutine。

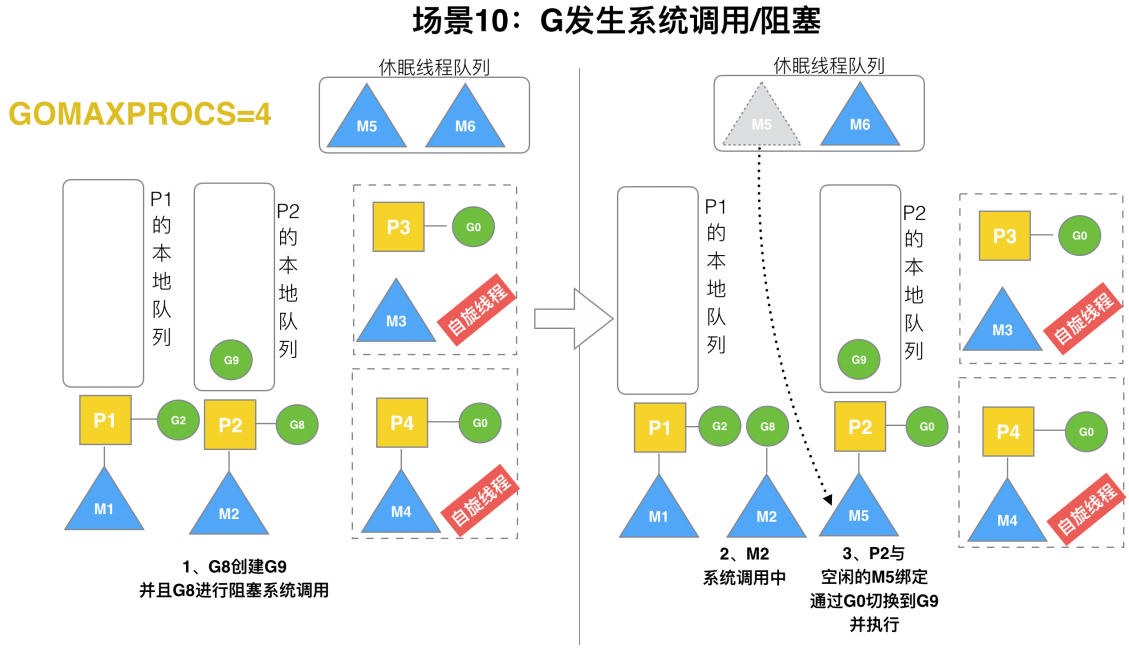


为什么要让m3和m4自旋，自旋本质是在运行，线程在运行却没有执行G，就变成了浪费CPU. 为什么不销毁现场，来节约CPU资源。因为创建和销毁CPU也会浪费时间，我们**希望当有新goroutine创建时，立刻能有M运行它**，如果销毁再新建就增加了时延，降低了效率。当然也考虑了过多的自旋线程是浪费CPU，所以系统中最多有 `GOMAXPROCS` 个自旋的线程(当前例子中的 `GOMAXPROCS` =4，所以一共4个P)，多余的没事做线程会让他们休眠。

---

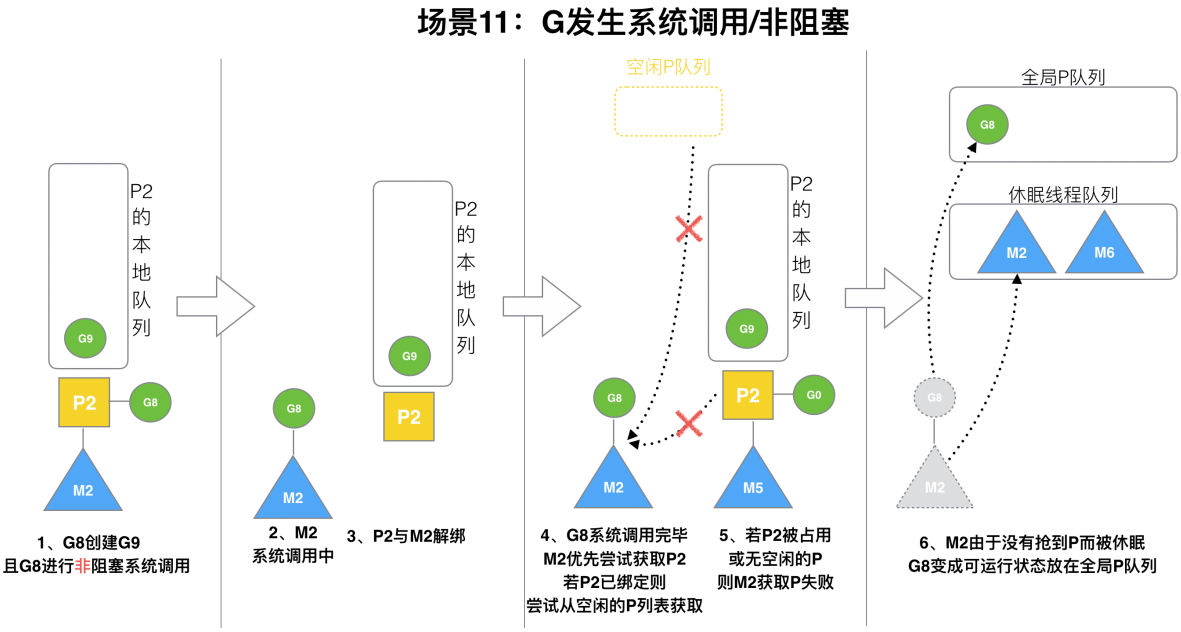
(10)场景10

假定当前除了M3和M4为自旋线程，还有M5和M6为空闲的线程(没有得到P的绑定，注意我们这里最多就只能够存在4个P，所以P的数量应该永远是M>=P, 大部分都是M在抢占需要运行的P)，G8创建了G9，G8进行了**阻塞的系统调用**，M2和P2立即解绑，P2会执行以下判断：如果P2本地队列有G、全局队列有G或有空闲的M，P2都会立马唤醒1个M和它绑定，否则P2则会加入到空闲P列表，等待M来获取可用的p。本场景中，P2本地队列有G9，可以和其他空闲的线程M5绑定。



(11)场景11

G8创建了G9，假如G8进行了**非阻塞系统调用**。



M2和P2会解绑，但M2会记住P2，然后G8和M2进入**系统调用**状态。当G8和M2退出系统调用时，会尝试获取P2，如果无法获取，则获取空闲的P，如果依然没有，G8会被记为可运行状态，并加入到全局队列,M2因为没有P的绑定而变成休眠状态(长时间休眠等待GC回收销毁)。

四、小结

总结，Go调度器很轻量也很简单，足以撑起goroutine的调度工作，并且让Go具有了原生（强大）并发的能力。**Go调度本质是把大量的goroutine分配到少量线程上去执行，并利用多核并行，实现更强大的并发。**

五、思维导图笔记

Golang中逃逸现象, 变量“何时栈?何时堆?”

一、C/C++报错?Golang通过?

我们先看一段代码

```
package main

func foo(arg_val int)(*int) {

    var foo_val int = 11;
    return &foo_val;
}

func main() {

    main_val := foo(666)

    println(*main_val)
}
```

编译运行

```
$ go run pro_1.go
11
```

竟然没有报错!

了解C/C++的小伙伴应该知道,这种情况是一定不允许的,因为 外部函数使用了子函数的局部变量, 理论来说,子函数的 `foo_val` 的声明周期早就销毁了才对,如下面的C/C++代码

```
#include <stdio.h>

int *foo(int arg_val) {

    int foo_val = 11;

    return &foo_val;
}

int main()
{
    int *main_val = foo(666);

    printf("%d\n", *main_val);
}
```

编译

```
$ gcc pro_1.c
pro_1.c: In function ‘foo’:
pro_1.c:7:12: warning: function returns address of local variable [-Wreturn-local-addr]
     return &foo_val;
            ^~~~~~~~
```

出了一个警告,不管他,再运行

```
$ ./a.out
段错误（核心已转储）
```

程序崩溃.

如上C/C++编译器明确给出了警告，foo把一个局部变量的地址返回了；反而高大上的go没有给出任何警告，难道是go编译器识别不出这个问题吗？

二、Golang编译器得逃逸分析

go语言编译器会自动决定把一个变量放在栈还是放在堆，编译器会做**逃逸分析(escape analysis)，当发现变量的作用域没有跑出函数范围，就可以在栈上，反之则必须分配在堆**。
go语言声称这样可以释放程序员关于内存的使用限制，更多的让程序员关注于程序功能逻辑本身。

我们再看如下代码:

```
package main

func foo(arg_val int) (*int) {

    var foo_val1 int = 11;
    var foo_val2 int = 12;
    var foo_val3 int = 13;
    var foo_val4 int = 14;
    var foo_val5 int = 15;


    //此处循环是防止go编译器将foo优化成inline(内联函数)
    //如果是内联函数，main调用foo将是原地展开，所以foo_val1-5相当于main作用域的变量
    //即使foo_val3发生逃逸，地址与其他也是连续的
    for i := 0; i < 5; i++ {
        println(&arg_val, &foo_val1, &foo_val2, &foo_val3, &foo_val4, &foo_val5)
    }

    //返回foo_val3给main函数
    return &foo_val3;
}


func main() {
    main_val := foo(666)

    println(*main_val, main_val)
}
```

我们运行一下

```
$ go run pro_2.go
0xc000030758 0xc000030738 0xc000030730 0xc000082000 0xc000030728 0xc000030720
0xc000030758 0xc000030738 0xc000030730 0xc000082000 0xc000030728 0xc000030720
0xc000030758 0xc000030738 0xc000030730 0xc000082000 0xc000030728 0xc000030720
0xc000030758 0xc000030738 0xc000030730 0xc000082000 0xc000030728 0xc000030720
0xc000030758 0xc000030738 0xc000030730 0xc000082000 0xc000030728 0xc000030720
13 0xc000082000
```

我们能看到 `foo_val3` 是返回给main的局部变量, 其中他的地址应该是 `0xc000082000` ,很明显与其他的foo_val1、2、3、4不是连续的.

我们用 `go tool compile` 测试一下

```
$ go tool compile -m pro_2.go
pro_2.go:24:6: can inline main
pro_2.go:7:9: moved to heap: foo_val3
```

果然,在编译的时候, `foo_val3` 具有被编译器判定为逃逸变量, 将 `foo_val3` 放在堆中开辟.

我们在用汇编证实一下:

```
$ go tool compile -S pro_2.go > pro_2.S
```

打开pro_2.S文件, 搜索 `runtime.newobject` 关键字

```
...
16      0x0021 00033 (pro_2.go:5)   PCDATA  $0, $0
17      0x0021 00033 (pro_2.go:5)   PCDATA  $1, $0
18      0x0021 00033 (pro_2.go:5)   MOVQ    $11, "".foo_val1+48(SP)
19      0x002a 00042 (pro_2.go:6)   MOVQ    $12, "".foo_val2+40(SP)
20      0x0033 00051 (pro_2.go:7)   PCDATA  $0, $1
21      0x0033 00051 (pro_2.go:7)   LEAQ    type.int(SB), AX
22      0x003a 00058 (pro_2.go:7)   PCDATA  $0, $0
23      0x003a 00058 (pro_2.go:7)   MOVQ    AX, (SP)
24      0x003e 00062 (pro_2.go:7)   CALL    runtime.newobject(SB)  //foo_val3是被new出来的
25      0x0043 00067 (pro_2.go:7)   PCDATA  $0, $1
26      0x0043 00067 (pro_2.go:7)   MOVQ    8(SP), AX
27      0x0048 00072 (pro_2.go:7)   PCDATA  $1, $1
28      0x0048 00072 (pro_2.go:7)   MOVQ    AX, "".&foo_val3+56(SP)
29      0x004d 00077 (pro_2.go:7)   MOVQ    $13, (AX)
30      0x0054 00084 (pro_2.go:8)   MOVQ    $14, "".foo_val4+32(SP)
31      0x005d 00093 (pro_2.go:9)   MOVQ    $15, "".foo_val5+24(SP)
32      0x0066 00102 (pro_2.go:9)   XORL    CX, CX
33      0x0068 00104 (pro_2.go:15)  JMP 252
...
```

看出来, foo_val3是被runtime.newobject()在堆空间开辟的, 而不是像其他几个是基于地址偏移的开辟的栈空间.

三、new的变量在栈还是堆?

那么对于new出来的变量,是一定在heap中开辟的吗,我们来看看

```
package main

func foo(arg_val int) (*int) {

    var foo_val1 * int = new(int);
    var foo_val2 * int = new(int);
    var foo_val3 * int = new(int);
    var foo_val4 * int = new(int);
    var foo_val5 * int = new(int);


    //此处循环是防止go编译器将foo优化成inline(内联函数)
    //如果是内联函数, main调用foo将是原地展开, 所以foo_val1-5相当于main作用域的变量
```

```
    //即使foo_val3发生逃逸，地址与其他也是连续的
    for i := 0; i < 5; i++ {
        println(arg_val, foo_val1, foo_val2, foo_val3, foo_val4, foo_val5)
    }

    //返回foo_val3给main函数
    return foo_val3;
}


func main() {
    main_val := foo(666)

    println(*main_val, main_val)
}
```

我们将foo_val1-5全部用new的方式来开辟, 编译运行看结果

```
$ go run pro_3.go
666 0xc000030728 0xc000030720 0xc00001a0e0 0xc000030738 0xc000030730
666 0xc000030728 0xc000030720 0xc00001a0e0 0xc000030738 0xc000030730
666 0xc000030728 0xc000030720 0xc00001a0e0 0xc000030738 0xc000030730
666 0xc000030728 0xc000030720 0xc00001a0e0 0xc000030738 0xc000030730
666 0xc000030728 0xc000030720 0xc00001a0e0 0xc000030738 0xc000030730
0 0xc00001a0e0
```

很明显, `foo_val3` 的地址 `0xc00001a0e0` 依然与其他的不是连续的. 依然具备逃逸行为.

四、结论

Golang中一个函数内局部变量，不管是不是动态new出来的，它会被分配在堆还是栈，是由编译器做逃逸分析之后做出的决定。

按理来说, 人家go的设计者明明就不希望开发者管这些,但是面试官就偏偏找这种问题问? 醉了也是.

Golang中make与new有何区别

一、前言

本文主要给大家介绍了Go语言中函数 `new` 与 `make` 的使用和区别，关于Go语言中 `new` 和 `make` 是内建的两个函数，主要用来创建分配类型内存。在我们定义生成变量的时候，可能会觉得有点迷惑，其实他们的规则很简单，下面我们就通过一些示例说明他们的区别和使用。

二、变量的声明

```
var i int
var s string
```

变量的声明我们可以通过var关键字，然后就可以在程序中使用。当我们不指定变量的默认值时，这些变量的默认值是他们的零值，比如int类型的零值是0,string类型的零值是 `""` ，引用类型的零值是 `nil` 。

对于例子中的两种类型的声明，我们可以直接使用，对其进行赋值输出。但是如果我们换成指针类型呢？

> test1.go

```
package main

import (
 "fmt"
)

func main() {
   var i *int
   *i=10
   fmt.Println(*i)
}
```

```
$ go run test1.go
panic: runtime error: invalid memory address or nil pointer dereference
[signal SIGSEGV: segmentation violation code=0x1 addr=0x0 pc=0x4849df]

goroutine 1 [running]:
main.main()
    /home/itheima/go/src/golang_deeper/make_new/t
```

从这个提示中可以看出，对于引用类型的变量，我们不光要声明它，还要为它分配内容空间，否则我们的值放在哪里去呢？这就是上面错误提示的原因。

对于值类型的声明不需要，是因为已经默认帮我们分配好了。

要分配内存，就引出来今天的 `new` 和 `make` 。

三、new

对于上面的问题我们如何解决呢？既然我们知道了没有为其分配内存，那么我们使用new分配一个吧。

```
func main() {

    var i *int
    i=new(int)
    *i=10
    fmt.Println(*i)

}
```

现在再运行程序，完美PASS，打印10。现在让我们看下new这个内置的函数。

```
// The new built-in function allocates memory. The first argument is a type,
// not a value, and the value returned is a pointer to a newly
// allocated zero value of that type.
func new(Type) *Type
```

它只接受一个参数，这个参数是一个类型，分配好内存后，返回一个指向该类型内存地址的指针。同时请注意它同时把分配的内存置为零，也就是类型的零值。

我们的例子中，如果没有 `*i=10` ，那么打印的就是0。这里体现不出来new函数这种内存置为零的好处，我们再看一个例子。

> test2.go

```
package main

import (
    "fmt"
    "sync"
)

type user struct {
    lock sync.Mutex
    name string
    age int
}

func main() {

    u := new(user) //默认给u分配到内存全部为0

    u.lock.Lock()  //可以直接使用，因为lock为0,是开锁状态
    u.name = "张三"
    u.lock.Unlock()

    fmt.Println(u)
}
```

运行

```
$ go run test2.go
&{{0 0} 张三 0}
```

示例中的user类型中的lock字段我不用初始化，直接可以拿来用，不会有无效内存引用异常，因为它已经被零值了。

这就是new，它返回的永远是类型的指针，指向分配类型的内存地址。

四、make

make也是用于内存分配的，但是和new不同。

它只用于

- chan
- map
- slice

的内存创建，而且它返回的类型就是这三个类型本身，而不是他们的指针类型，因为这三种类型就是引用类型，所以就没有必要返回他们的指针了。

注意，因为这三种类型是引用类型，所以必须得初始化，但是不是置为零值，这个和new是不一样的。

```
func make(t Type, size ...IntegerType) Type
```

从函数声明中可以看到，返回的还是该类型。

五、make与new的异同

相同

- 堆空间分配

不同

make: 只用于slice、map以及channel的初始化， 无可替代

new: 用于类型内存分配(初始化值为0)， 不常用

> new不常用
>
> 所以有new这个内置函数，可以给我们分配一块内存让我们使用，但是现实的编码中，它是不常用的。我们通常都是采用短语句声明以及结构体的字面量达到我们的目的，比如：
>
> ```
> i := 0
> u := user{}
> ```

> make 无可替代
>
> 我们在使用slice、map以及channel的时候，还是要使用make进行初始化，然后才才可以对他们进行操作。

Golang三色标记+混合写屏障GC模式全分析

> 本节为**重点**章节

> 本章节含视频版:

垃圾回收(Garbage Collection，简称GC)是编程语言中提供的自动的内存管理机制，自动释放不需要的对象，让出存储器资源，无需程序员手动执行。

Golang中的垃圾回收主要应用三色标记法，GC过程和其他用户goroutine可并发运行，但需要一定时间的**STW(stop the world)**，STW的过程中，CPU不执行用户代码，全部用于垃圾回收，这个过程的影响很大，Golang进行了多次的迭代优化来解决这个问题。

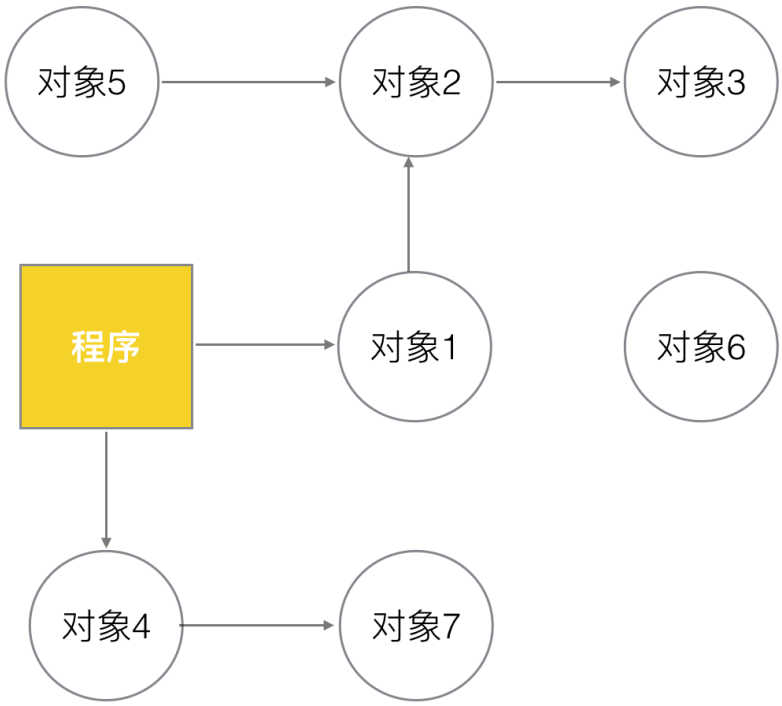
一、Go V1.3之前的标记-清除(mark and sweep)算法

此算法主要有两个主要的步骤：

- 标记(Mark phase)
- 清除(Sweep phase)

第一步，暂停程序业务逻辑, 找出不可达的对象，然后做上标记。第二步，回收标记好的对象。

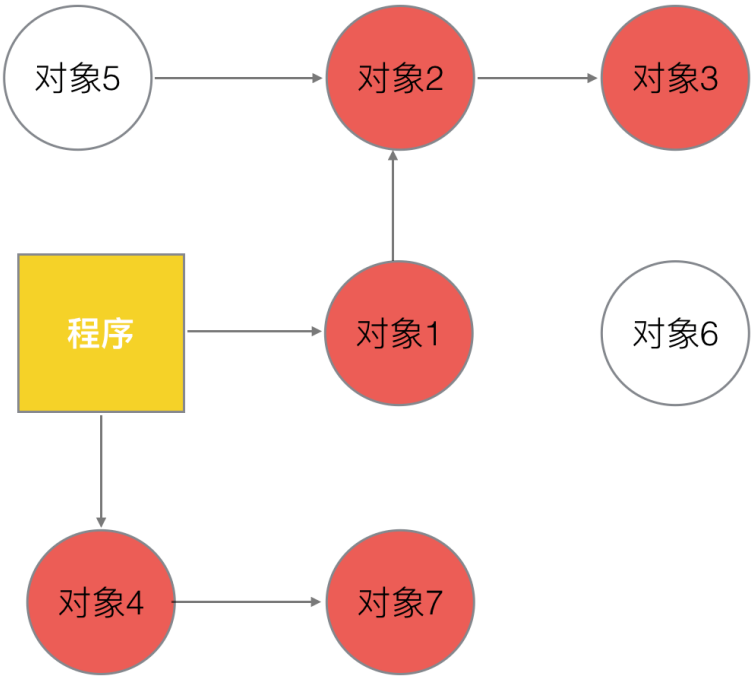
操作非常简单，但是有一点需要额外注意：mark and sweep算法在执行的时候，需要程序暂停！即 `STW(stop the world)` 。也就是说，这段时间程序会卡在哪儿。



(1)、程序与对象的可达关系

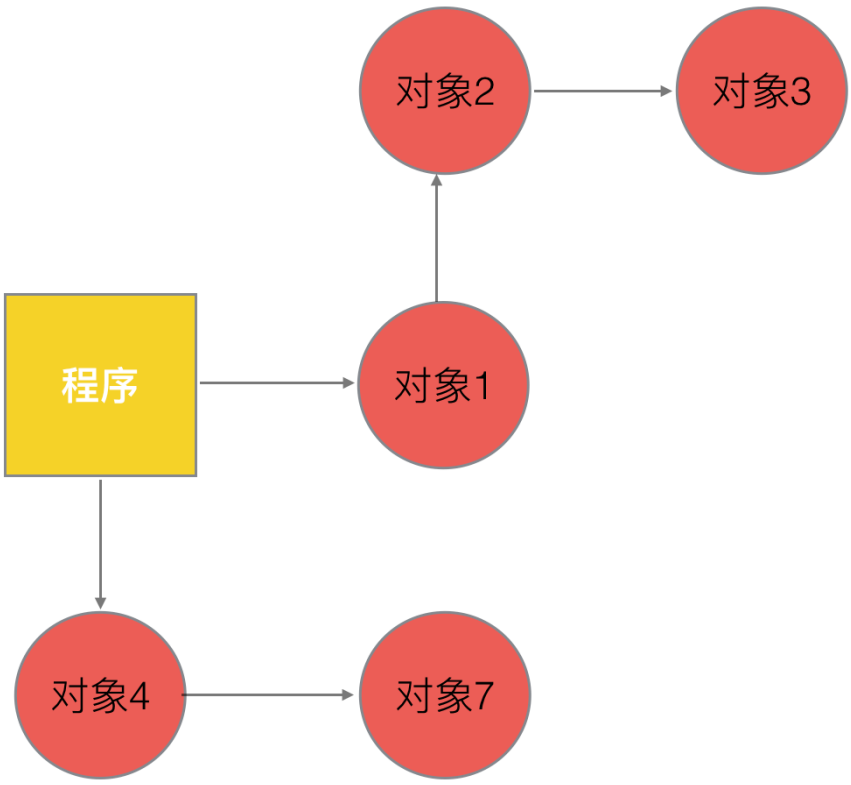
目前程序的可达对象有1-2-3, 4-7 等五个对象

第二步, 开始标记，程序找出它所有可达的对象，并做上标记。如下图所示：



(2)、1-2-3, 4-7 等五个对象可达,做上标记

第三步, 标记完了之后，然后开始清除未标记的对象. 结果如下.



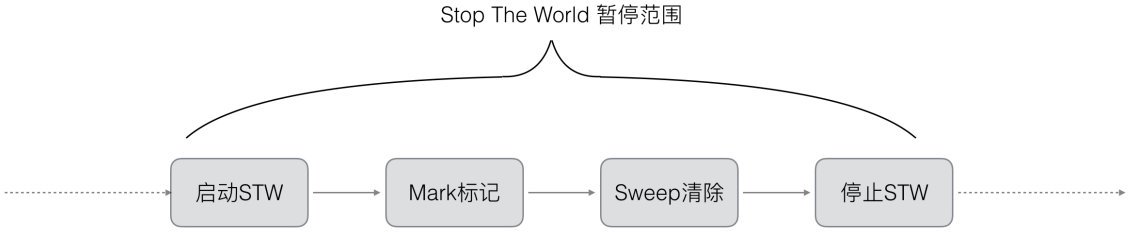
(3)、对象5, 6不可达,被GC所清除

第四步, 停止暂停，让程序继续跑。然后循环重复这个过程，直到process程序生命周期结束。

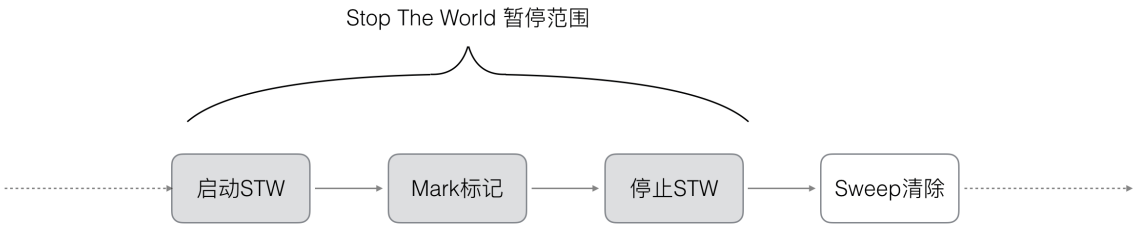
二、标记-清扫(mark and sweep)的缺点

- STW，stop the world；让程序暂停，程序出现卡顿 **(重要问题)**。
- 标记需要扫描整个heap
- 清除数据会产生heap碎片

所以Go V1.3版本之前就是以上来实施的, 流程是



Go V1.3 做了简单的优化,将STW提前, 减少STW暂停的时间范围.如下所示



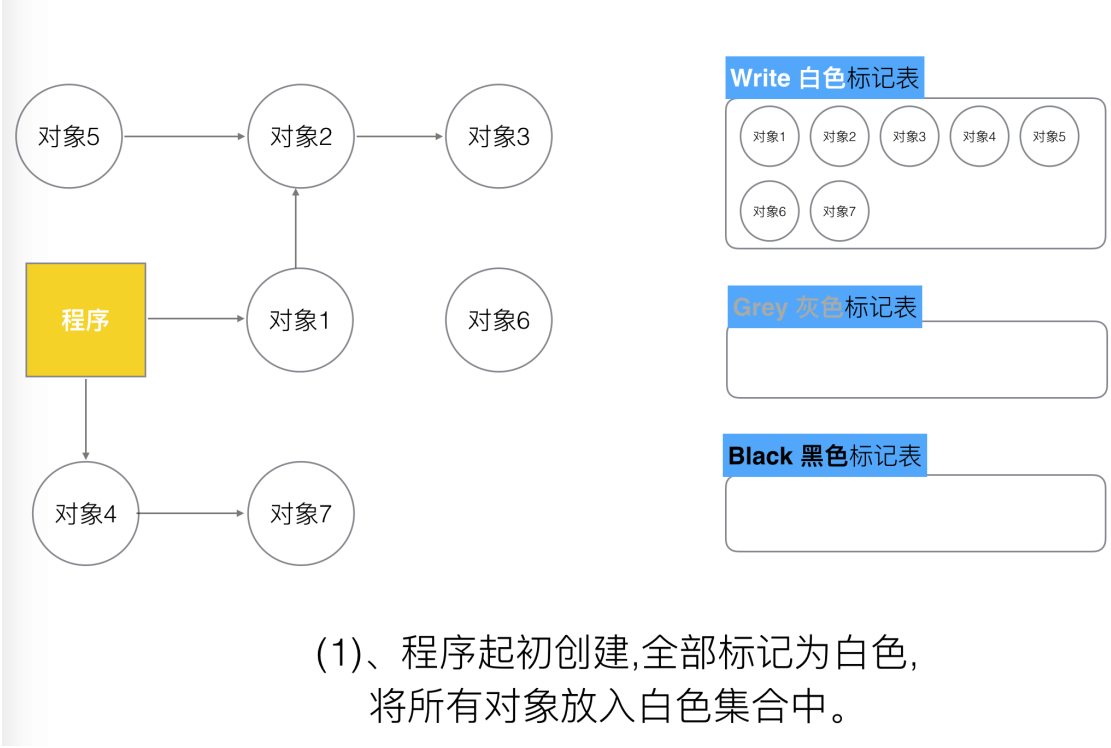
这里面最重要的问题就是：mark-and-sweep 算法会暂停整个程序 。

Go是如何面对并这个问题的呢？接下来G V1.5版本 就用**三色并发标记法**来优化这个问题.

三、Go V1.5的三色并发标记法

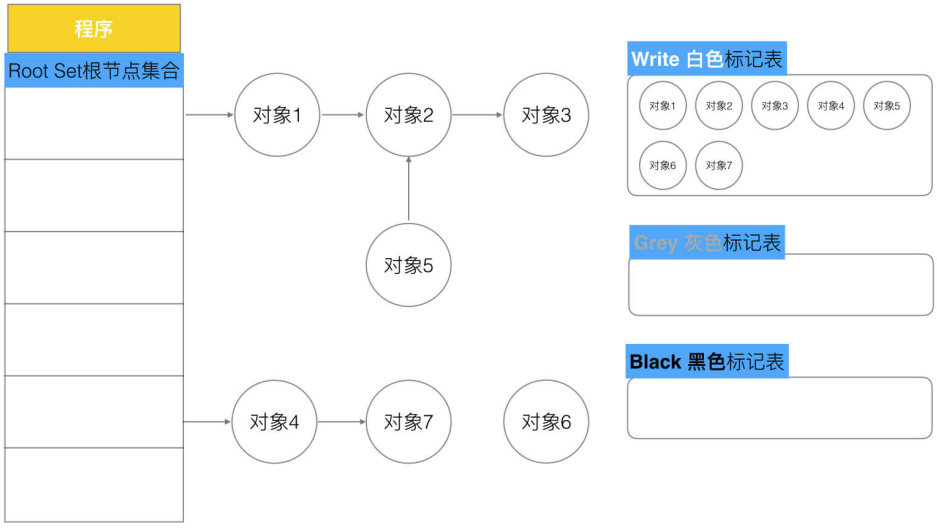
三色标记法 实际上就是通过三个阶段的标记来确定清楚的对象都有哪些. 我们来看一下具体的过程.

第一步，就是只要是新创建的对象,默认的颜色都是标记为“白色”.



(1)、程序起初创建,全部标记为白色,
将所有对象放入白色集合中。

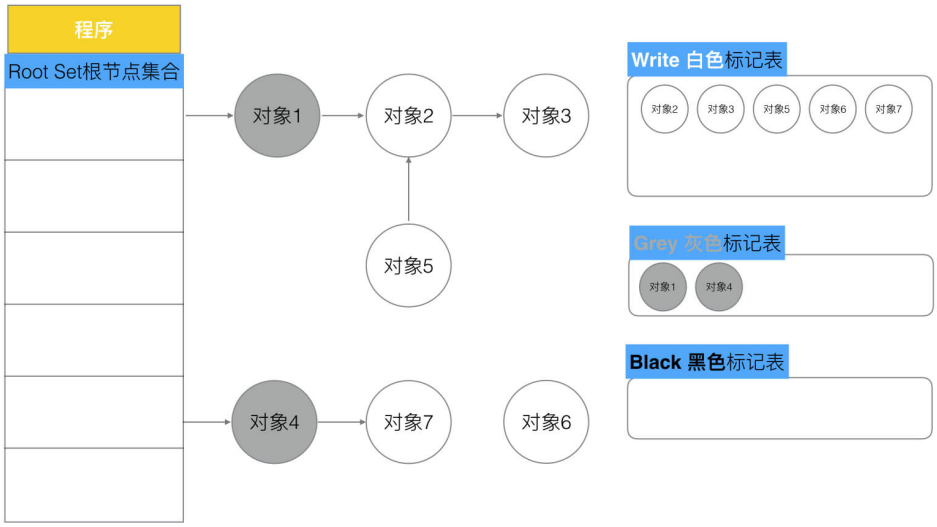
这里面需要注意的是，所谓“程序”，则是一些对象的跟节点集合.



(2)、将程序的根节点集合展开的形式

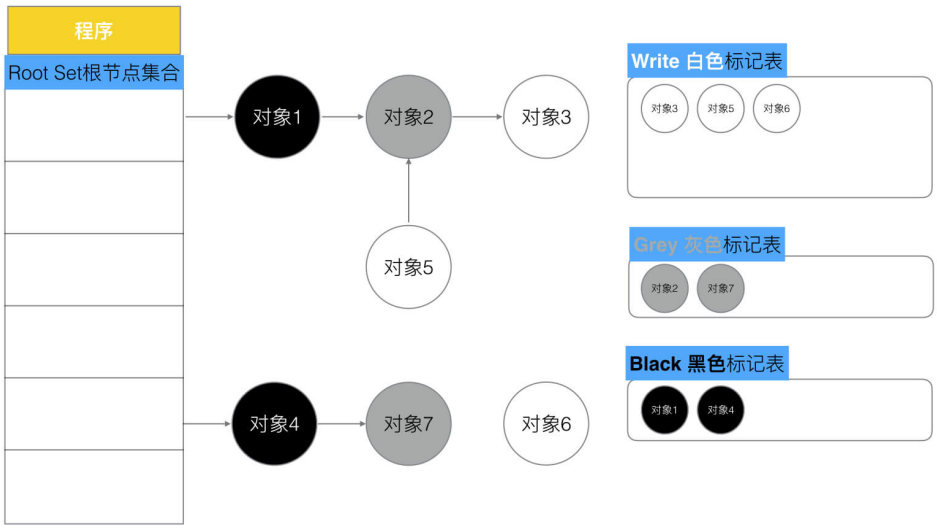
所以上图,可以转换如下的方式来表示.

第二步，每次GC回收开始，然后从根节点开始遍历所有对象，把遍历到的对象从白色集合放入“灰色”集合。



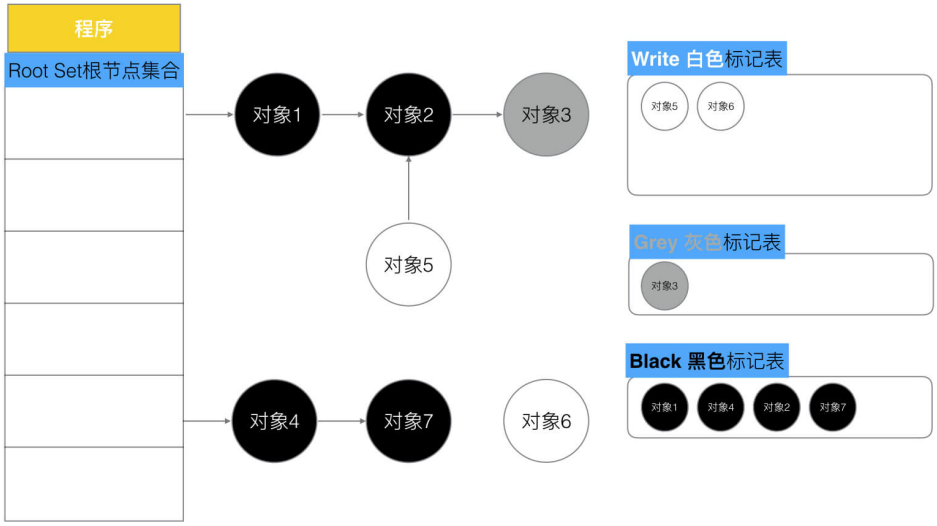
遍历Root Set (非递归形式,只遍历一次).得到灰色节点.

第三步，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

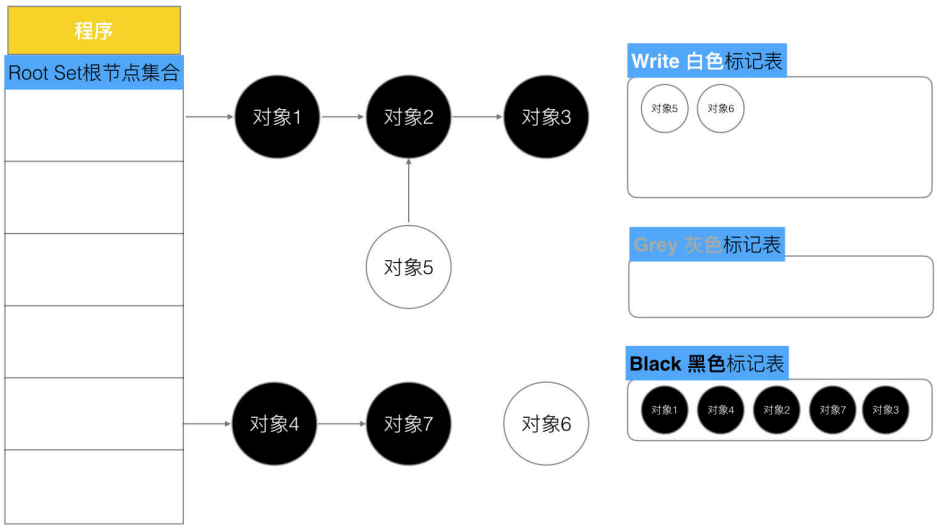


遍历Grey 灰色标记表.将可达的对象,从白色标记为灰色
遍历之后的灰色,标记为**黑色**

第四步, 重复**第三步**, 直到灰色中无任何对象.

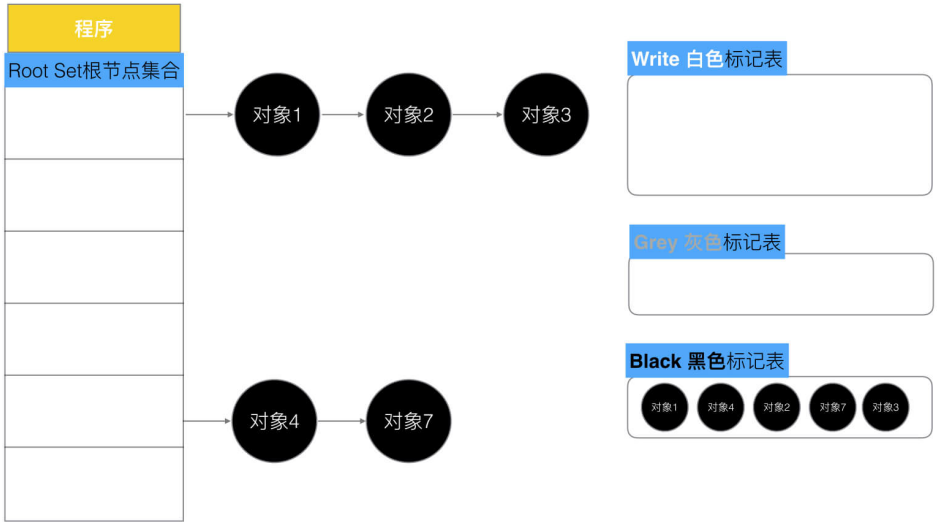


重复上一步, 直到灰色标记表中无任何对象



重复上一步, 直到灰色标记表中无任何对象

第五步: 回收所有的白色标记表的对象. 也就是回收垃圾.



收集所有白色对象（垃圾）

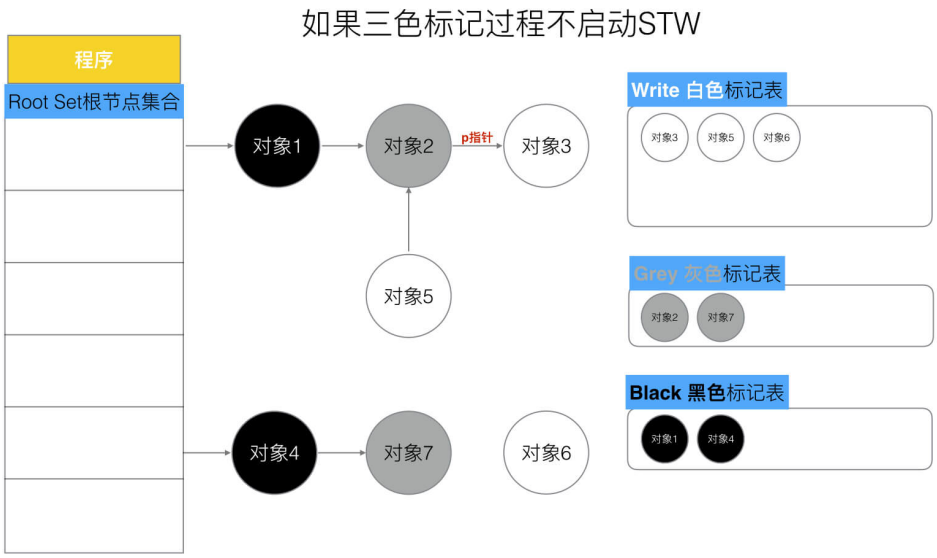
以上便是 三色并发标记法 , 不难看出,我们上面已经清楚的体现 三色 的特性, 那么又是如何实现并行的呢?

> Go是如何解决标记-清除(mark and sweep)算法中的卡顿(stw，stop the world)问题的呢？

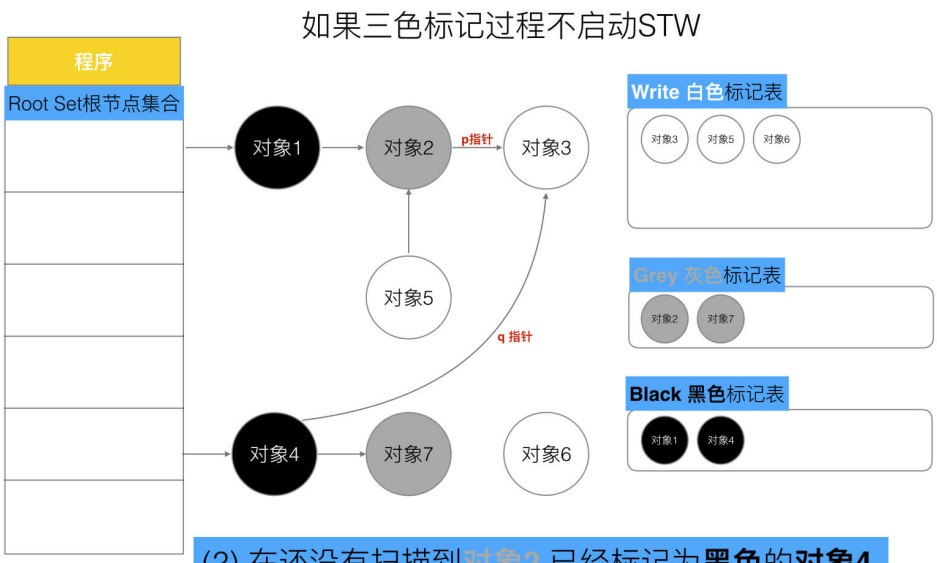
四、没有STW的三色标记法

我们还是基于上述的三色并发标记法来说, 他是一定要依赖STW的. 因为如果不暂停程序, 程序的逻辑改变对象引用关系, 这种动作如果在标记阶段做了修改，会影响标记结果的正确性。我们举一个场景.

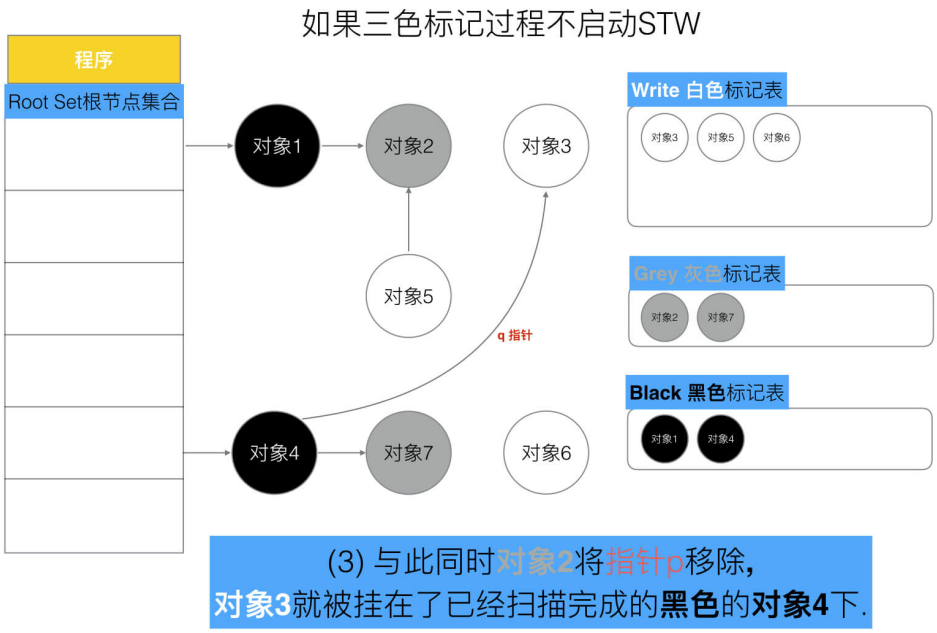
如果三色标记法, 标记过程不使用STW将会发生什么事情?



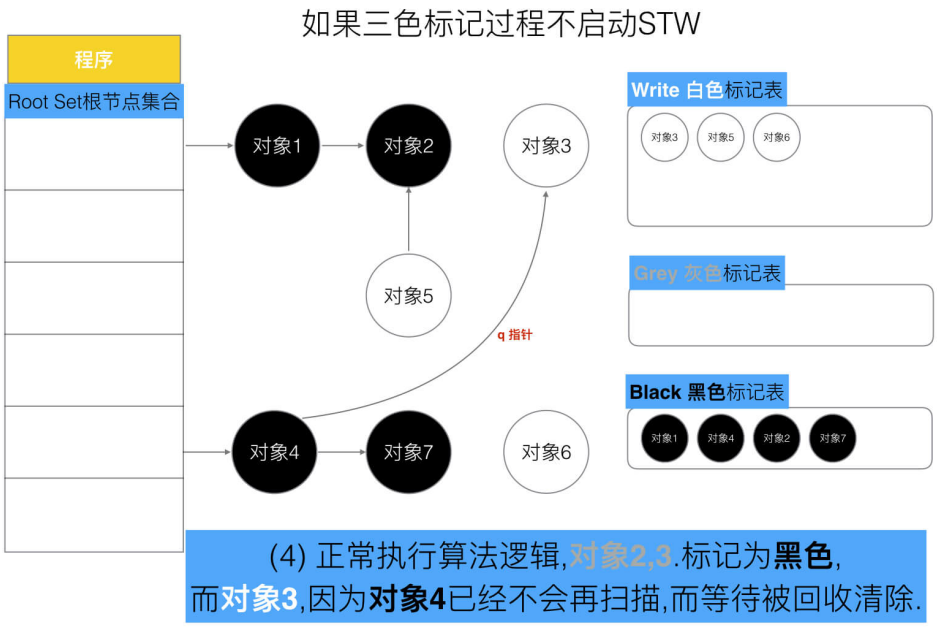
(1) 已经标记为灰色的对象2, 有指针p指向白色的对象3.



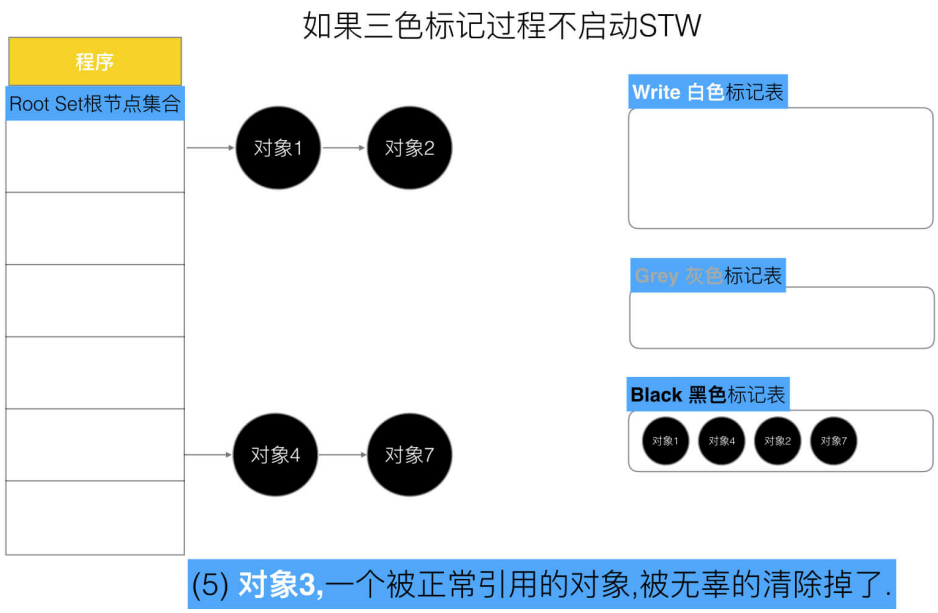
(2) 在还没有扫描到对象2,已经标记为**黑色**的**对象4**, 创建指针q,指向**对象3**.



(3) 与此同时对象2将指针p移除,
对象3就被挂在了已经扫描完成的**黑色**的**对象4**下.



(4) 正常执行算法逻辑,对象2,3.标记为**黑色**,
而**对象3**,因为**对象4**已经不会再扫描,而等待被回收清除.



可以看出，有两个问题，在三色标记法中,是不希望被发生的

- 条件1: 一个白色对象被黑色对象引用**(白色被挂在黑色下)**
- 条件2: 灰色对象与它之间的可达关系的白色对象遭到破坏**(灰色同时丢了该白色)**

当以上两个条件同时满足时，就会出现对象丢失现象!

当然，如果上述中的白色对象3，如果他还有很多下游对象的话，也会一并都清理掉.

为了防止这种现象的发生，最简单的方式就是STW，直接禁止掉其他用户程序对对象引用关系的干扰，但是**STW的过程有明显的资源浪费，对所有的用户程序都有很大影响**，如何能在保证对象不丢失的情况下合理的尽可能的提高GC效率，减少STW时间呢？

答案就是，那么我们只要使用一个机制,来破坏上面的两个条件就可以了.

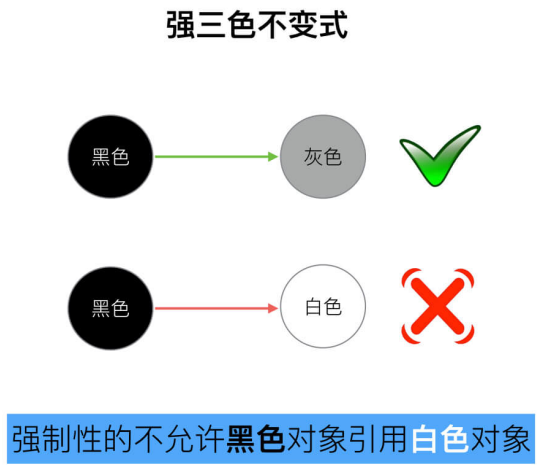
五、屏障机制

我们让GC回收器,满足下面两种情况之一时,可保对象不丢失. 所以引出两种方式.

(1)“强-弱”三色不变式

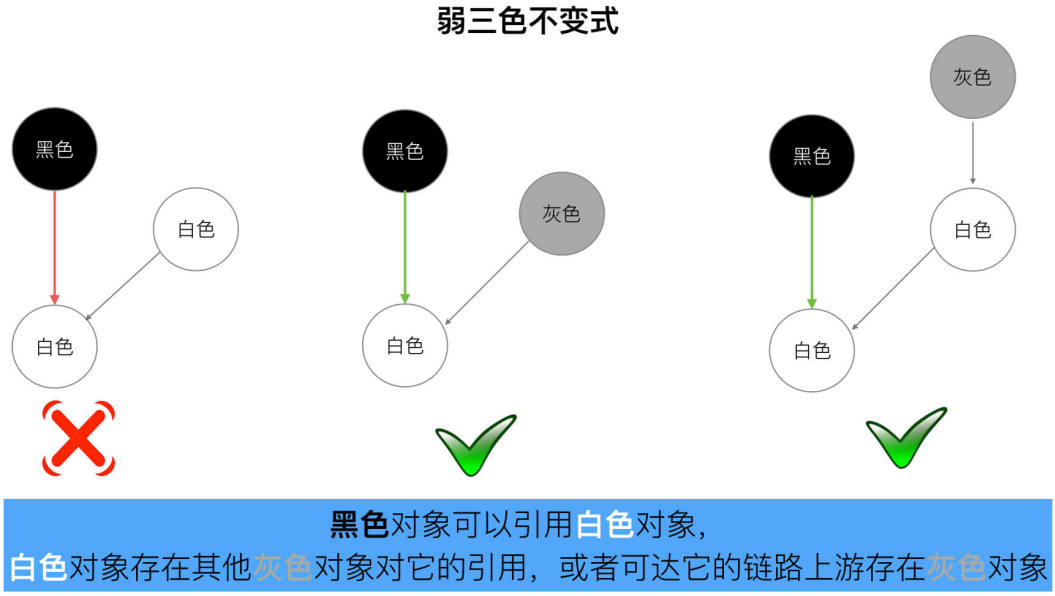
- 强三色不变式

不存在黑色对象引用到白色对象的指针。



- 弱三色不变式

所有被黑色对象引用的白色对象都处于灰色保护状态.



为了遵循上述的两个方式,Golang团队初步得到了如下具体的两种屏障方式“插入屏障”, “删除屏障”.

(2) 插入屏障

具体操作 ：在A对象引用B对象的时候，B对象被标记为灰色。(将B挂在A下游，B必须被标记为灰色)

满足 ：**强三色不变式**. (不存在黑色对象引用白色对象的情况了， 因为白色会强制变成灰色)

伪码如下:

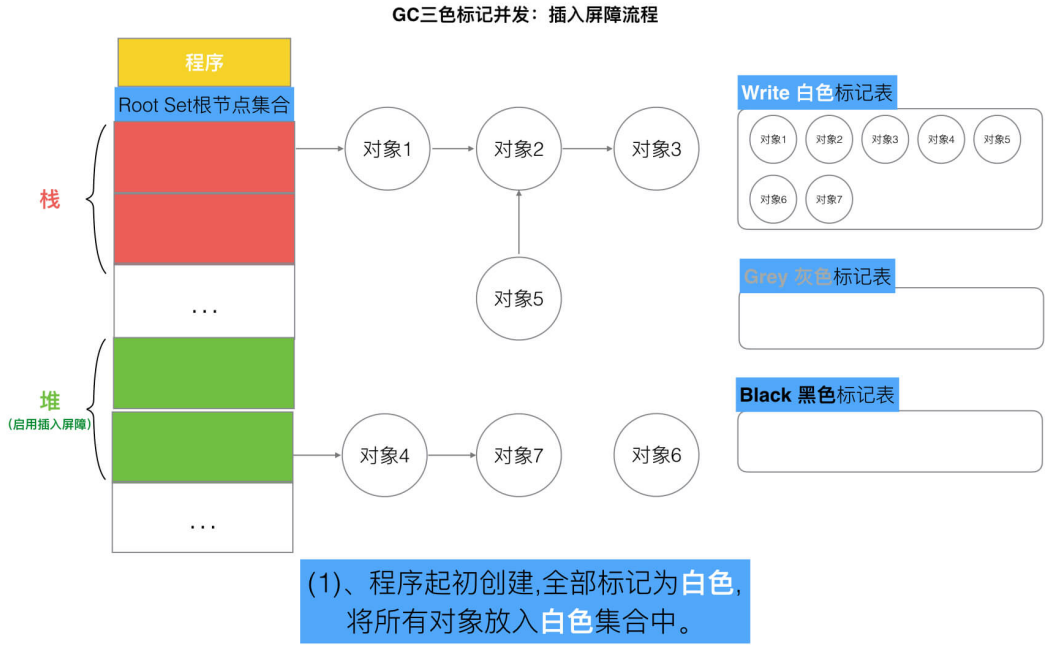
```
添加下游对象(当前下游对象slot, 新下游对象ptr) {
  //1
  标记灰色(新下游对象ptr)

  //2
  当前下游对象slot = 新下游对象ptr
}
```

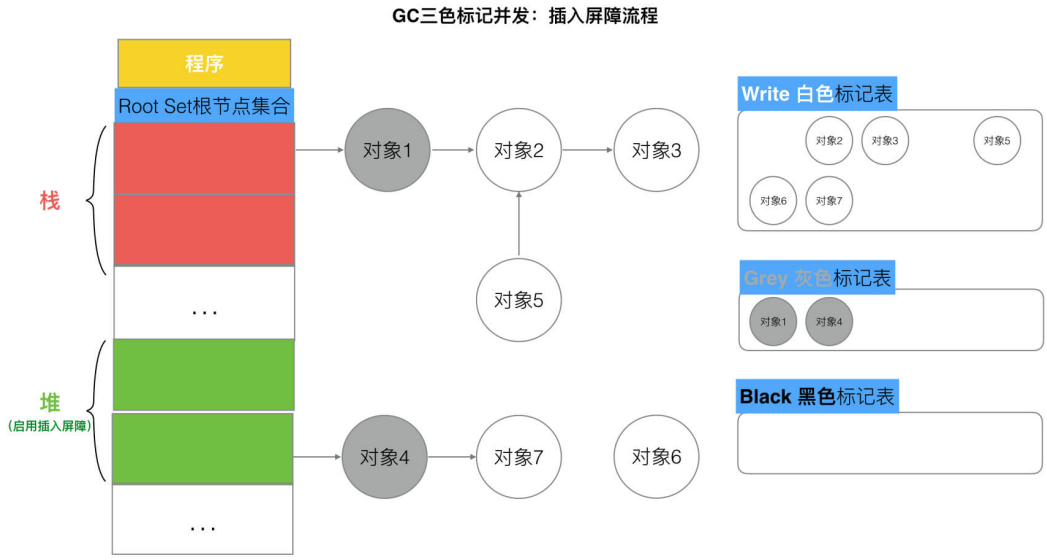
场景：

```
A.添加下游对象(nil, B)    //A 之前没有下游，  新添加一个下游对象B，  B被标记为灰色
A.添加下游对象(C, B)      //A 将下游对象C 更换为B，   B被标记为灰色
```

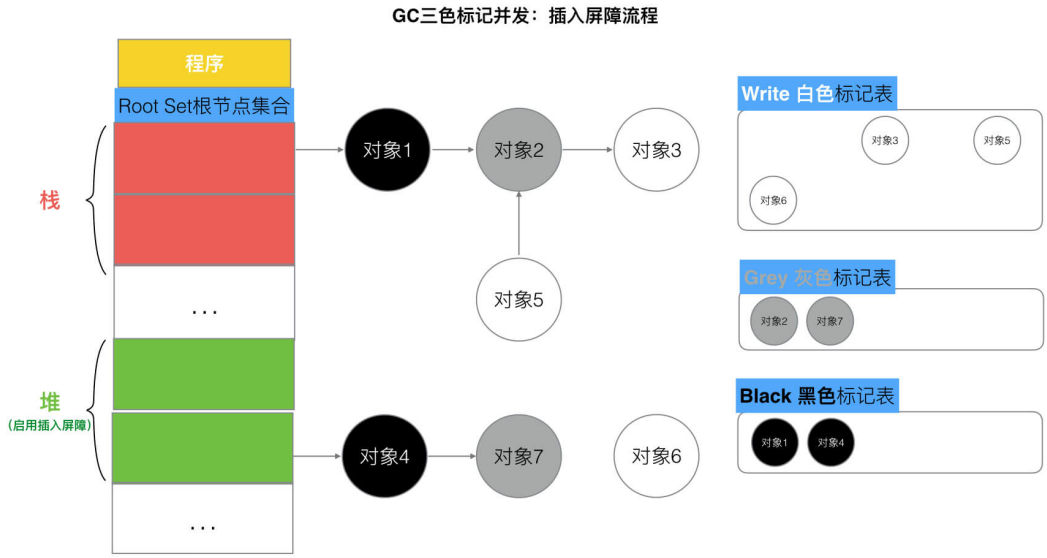
这段伪码逻辑就是写屏障,. 我们知道,黑色对象的内存槽有两种位置， 栈 和 堆 . 栈空间的特点是容量小,但是要求相应速度快,因为函数调用弹出频繁使用, 所以“插入屏障”机制,在**栈空间的对象操作中不使用**. 而仅仅使用在堆空间对象的操作中.

接下来，我们用几张图，来模拟整个一个详细的过程， 希望您能够更可观的看清晰整体流程。

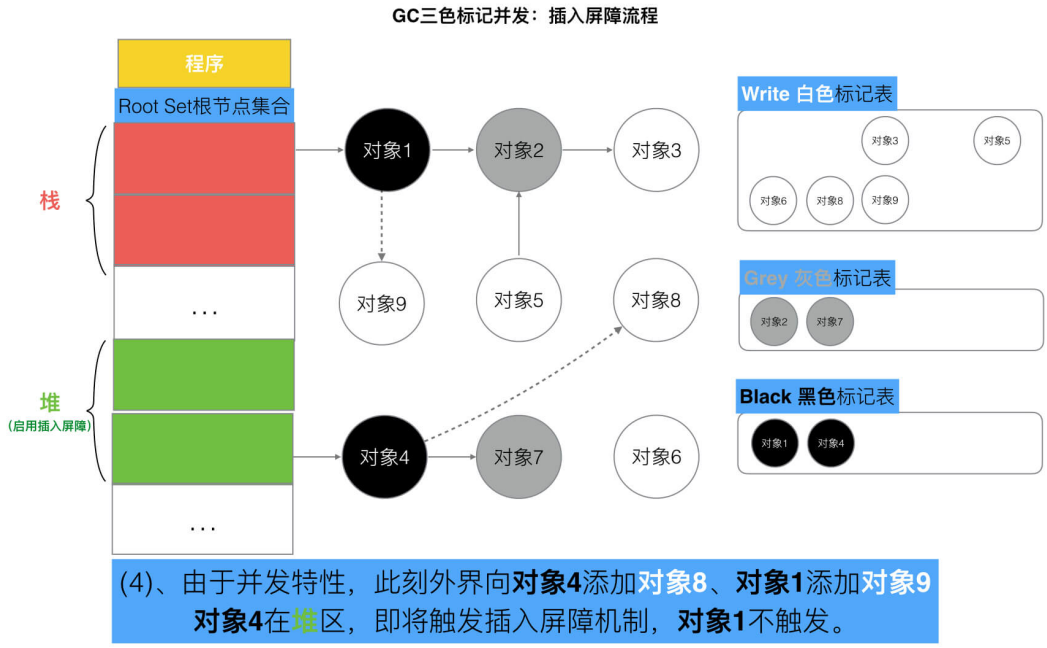




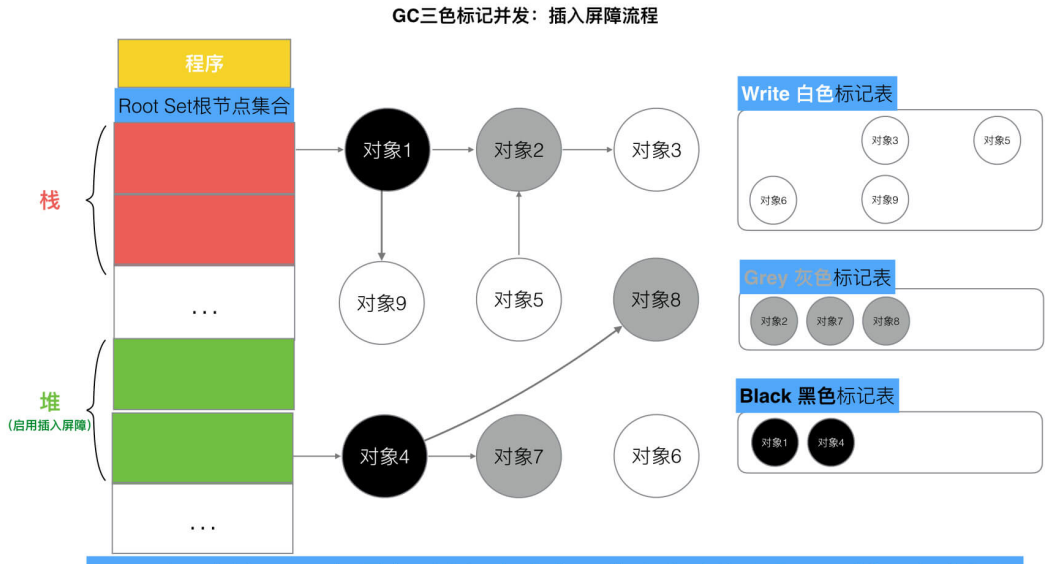
(2)、遍历Root Set (非递归形式,只遍历一次).得到灰色节点.



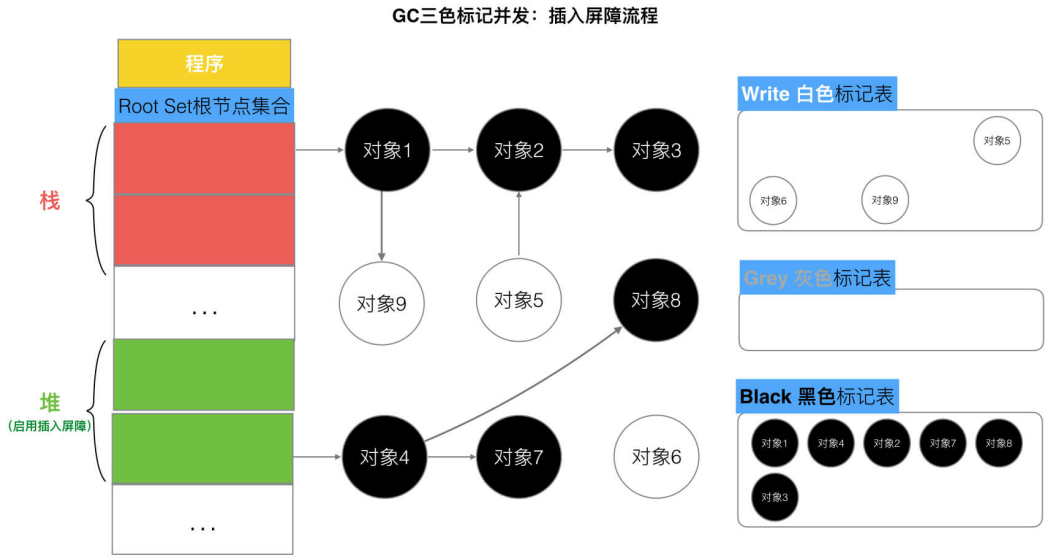
(3)、遍历Grey 灰色标记表.将可达的对象,从白色标记为灰色
遍历之后的灰色,标记为**黑色**



(4)、由于并发特性，此刻外界向**对象4**添加**对象8**、**对象1**添加**对象9**
对象4在堆区，即将触发插入屏障机制，**对象1**不触发。

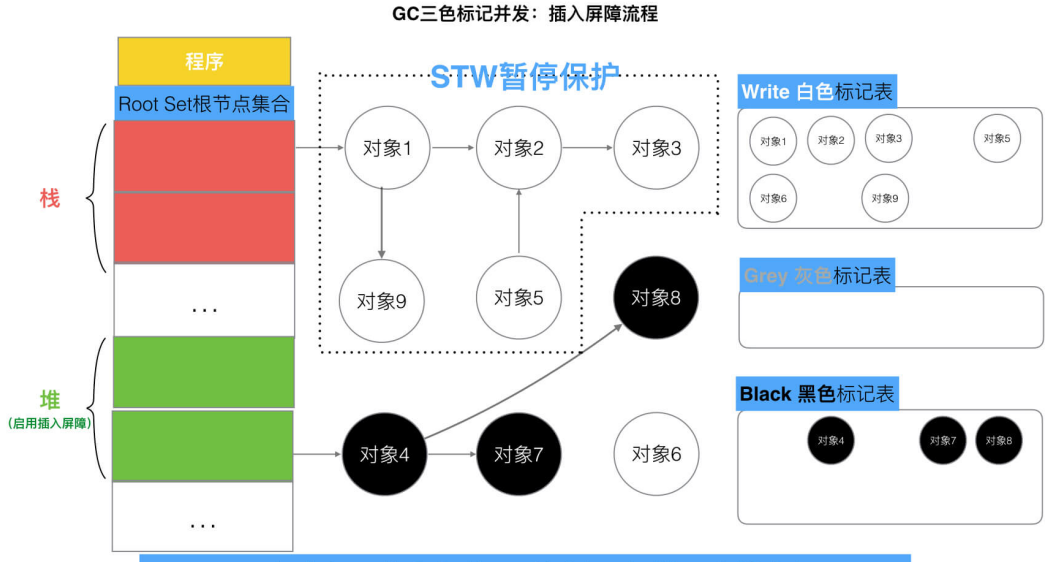


(5)、由于插入写屏障(**黑色**对象添加**白色**，将**白色**改为灰色)，**对象8**变成灰色
对象9依然为**白色**

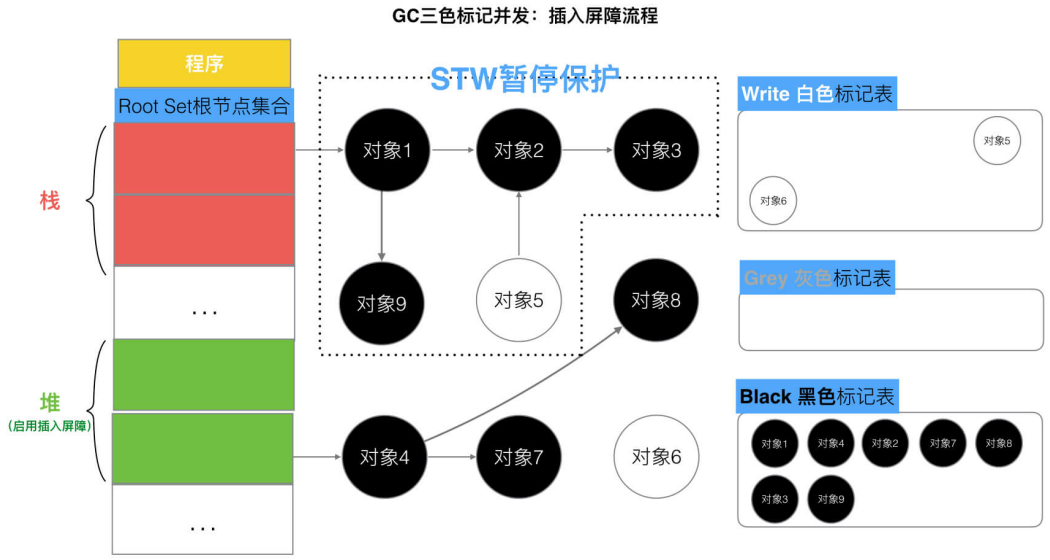


(6)、继续循环上述流程进行三色标记，直到没有灰色节点

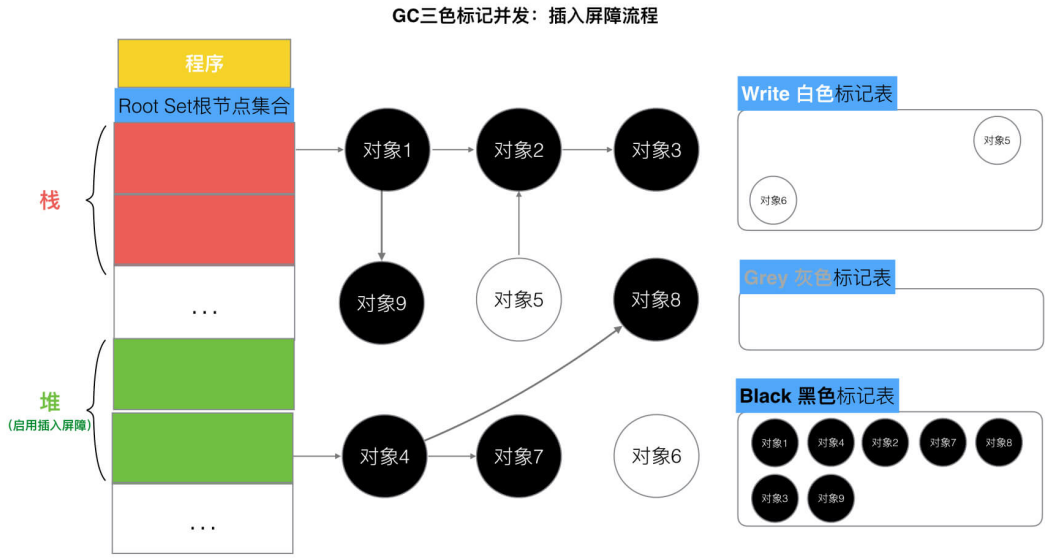
但是如果栈不添加,当全部三色标记扫描之后,栈上有可能依然存在白色对象被引用的情况(如上图的对象9). 所以要对栈重新进行三色标记扫描, 但这次为了对象不丢失, 要对本次标记扫描启动STW暂停. 直到栈空间的三色标记结束.



(7)、在准备回收白色前，重新遍历扫描一次栈空间。
此时加STW暂停保护栈，防止外界干扰(有新的白色被黑色添加)

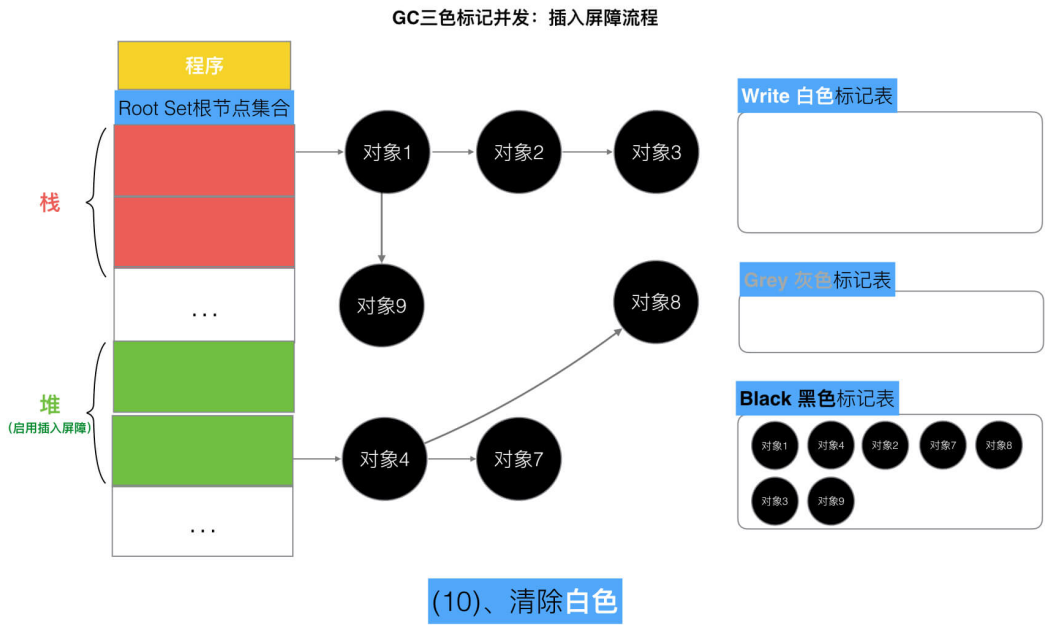


(8)、在STW中，将栈中的对象一次三色标记，直到没有灰色节点



(9)、停止STW

最后将栈和堆空间 扫描剩余的全部 白色节点清除. 这次STW大约的时间在10~100ms间.



---

(3) 删除屏障

具体操作：被删除的对象，如果自身为灰色或者白色，那么被标记为灰色。

满足：**弱三色不变式**. (保护灰色对象到白色对象的路径不会断)

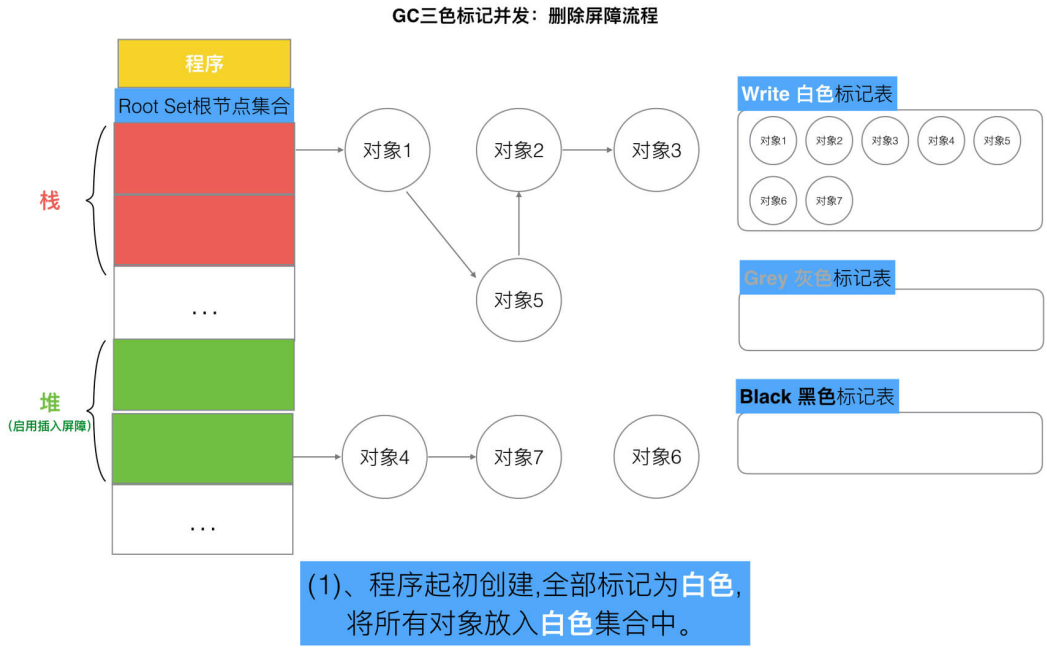
伪代码：

```
添加下游对象(当前下游对象slot， 新下游对象ptr) {
  //1
  if (当前下游对象slot是灰色 || 当前下游对象slot是白色) {
          标记灰色(当前下游对象slot)     //slot为被删除对象， 标记为灰色
  }

  //2
  当前下游对象slot = 新下游对象ptr
}
```

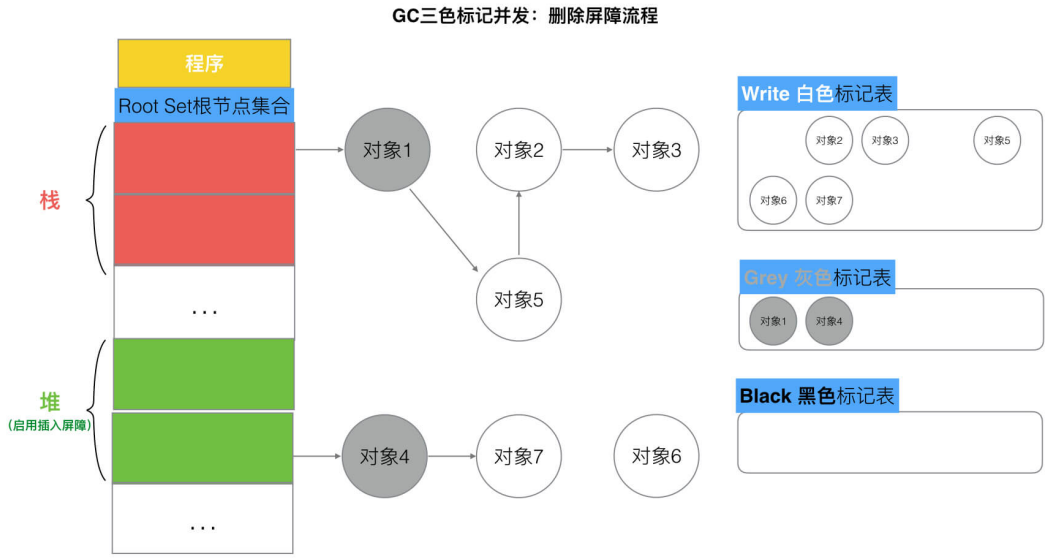
场景：

```
A.添加下游对象(B, nil)   //A对象，删除B对象的引用。   B被A删除，被标记为灰(如果B之前为白)
A.添加下游对象(B, C)          //A对象，更换下游B变成C。    B被A删除，被标记为灰(如果B之前为白)
```

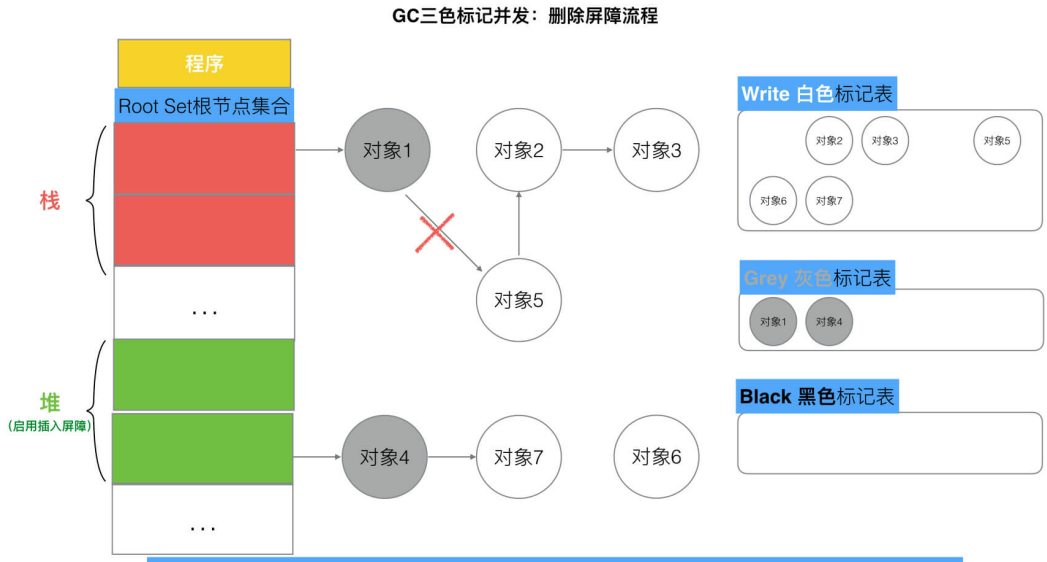
接下来，我们用几张图，来模拟整个一个详细的过程， 希望您能够更可观的看清晰整体流程。



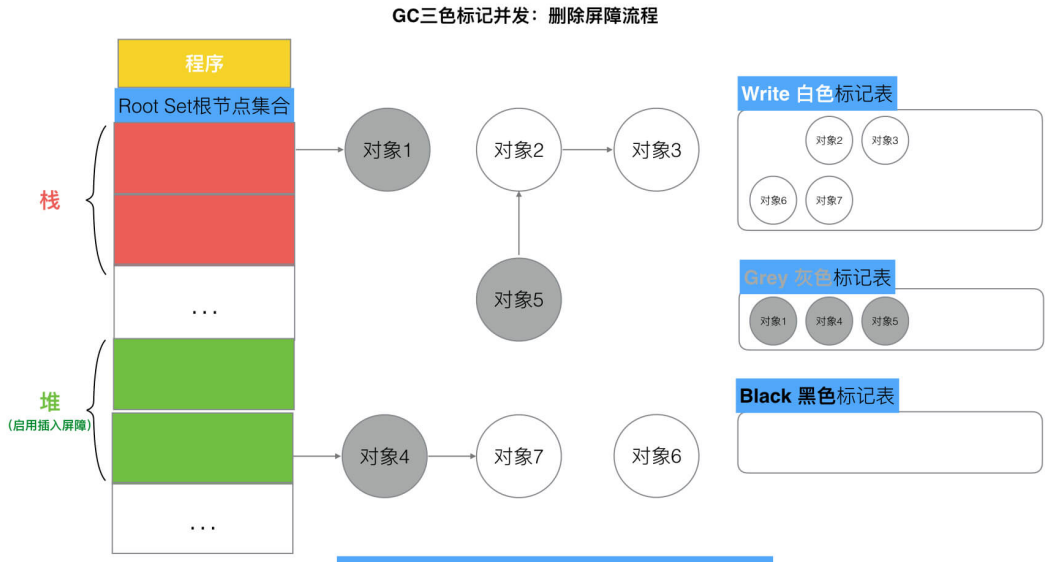
(1)、程序起初创建,全部标记为白色,
将所有对象放入白色集合中。



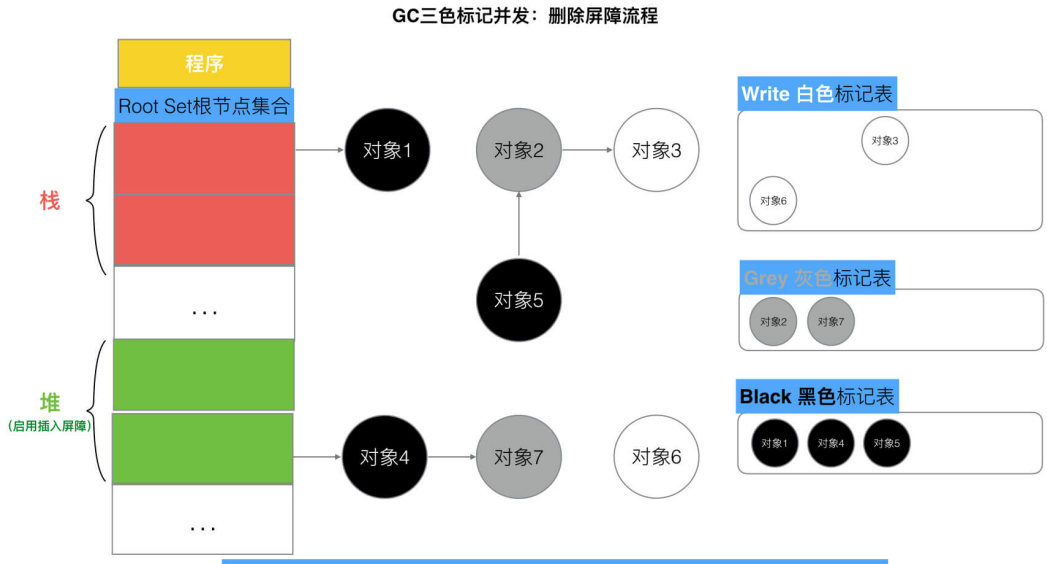
(2)、遍历Root Set (非递归形式,只遍历一次).得到灰色节点.



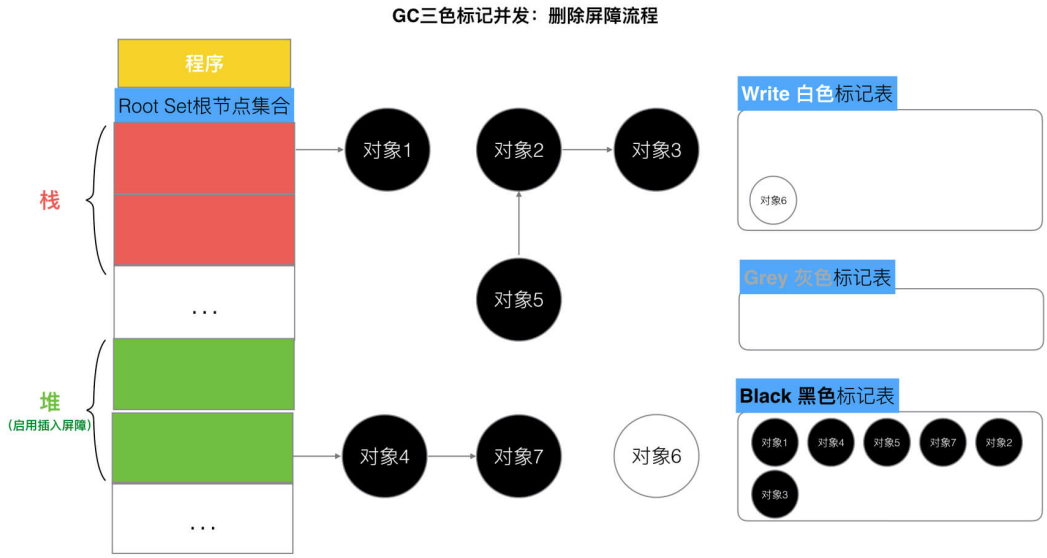
(3)、灰色对象1删除对象5
如果不触发删除写屏障，5-2-3路径与主链路断开，最后均会被清楚



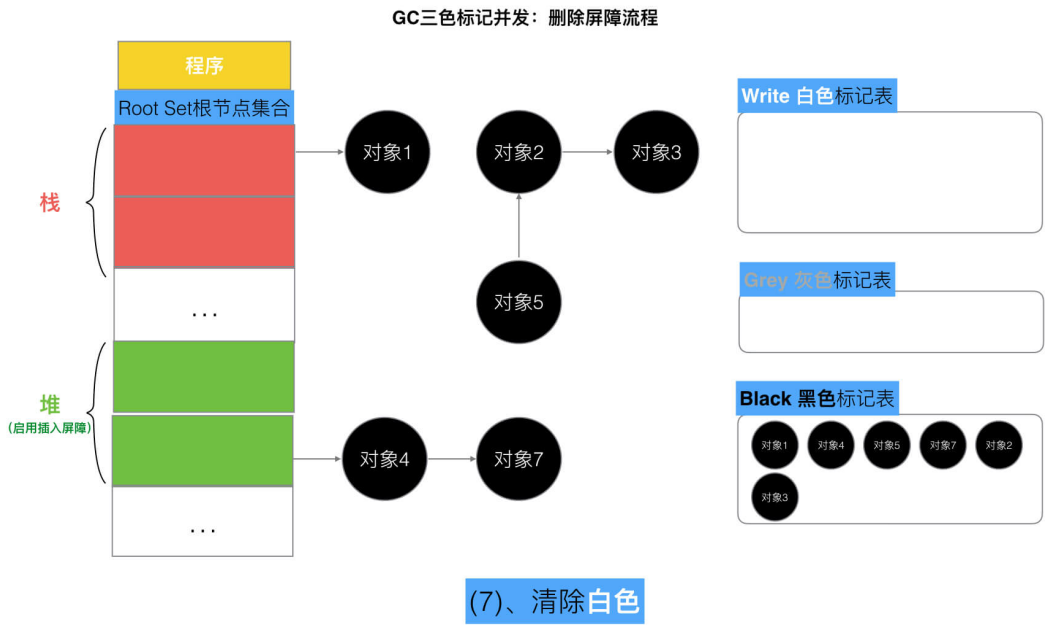
(4)、触发删除写屏障
被删除的对象5，自身被标记为灰色



(5)、遍历Grey 灰色标记表.将可达的对象,从白色标记为灰色
遍历之后的灰色,标记为**黑色**



(6)、继续循环上述流程进行三色标记，直到没有灰色节点



这种方式的回收精度低，一个对象即使被删除了最后一个指向它的指针也依旧可以活过这一轮，在下一轮GC中被清理掉。

六、Go V1.8的混合写屏障(hybrid write barrier)机制

插入写屏障和删除写屏障的短板：

- 插入写屏障：结束时需要STW来重新扫描栈，标记栈上引用的白色对象的存活；
- 删除写屏障：回收精度低，GC开始时STW扫描堆栈来记录初始快照，这个过程会保护开始时刻的所有存活对象。

Go V1.8版本引入了混合写屏障机制（hybrid write barrier），避免了对栈re-scan的过程，极大的减少了STW的时间。结合了两者的优点。

---

(1) 混合写屏障规则

具体操作：

1、GC开始将栈上的对象全部扫描并标记为黑色(之后不再进行第二次重复扫描，无需STW)，

2、GC期间，任何在栈上创建的新对象，均为黑色。

3、被删除的对象标记为灰色。

4、被添加的对象标记为灰色。

满足：变形的**弱三色不变式**.

伪代码：

```
添加下游对象(当前下游对象slot, 新下游对象ptr) {
    //1
    标记灰色(当前下游对象slot)    //只要当前下游对象被移走，就标记灰色

    //2
    标记灰色(新下游对象ptr)
```

```
        //3
        当前下游对象slot = 新下游对象ptr
}
```

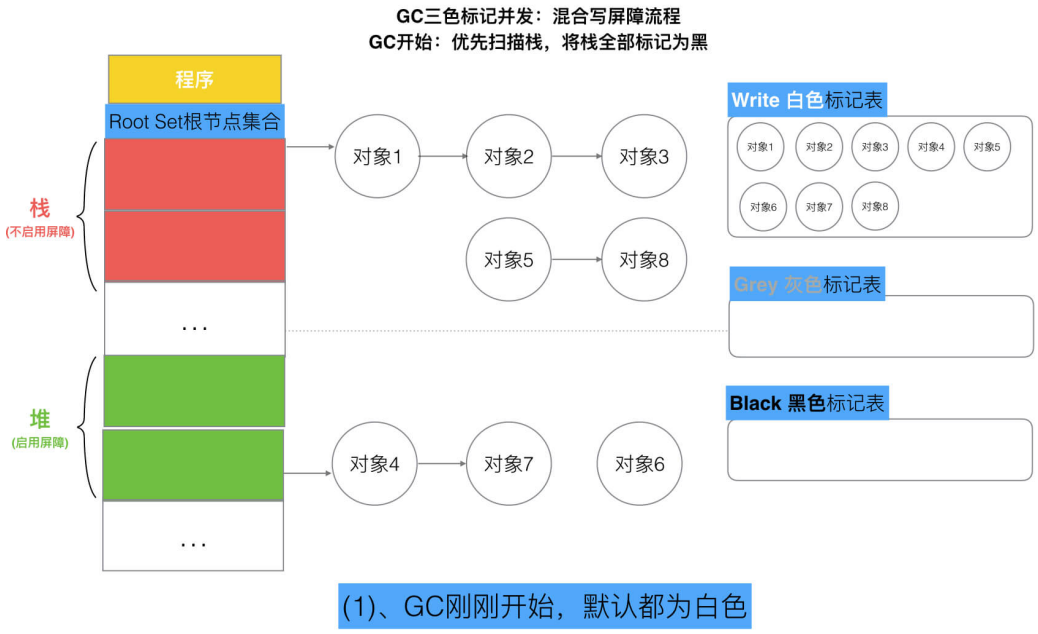
> 这里我们注意， 屏障技术是不在栈上应用的，因为要保证栈的运行效率。

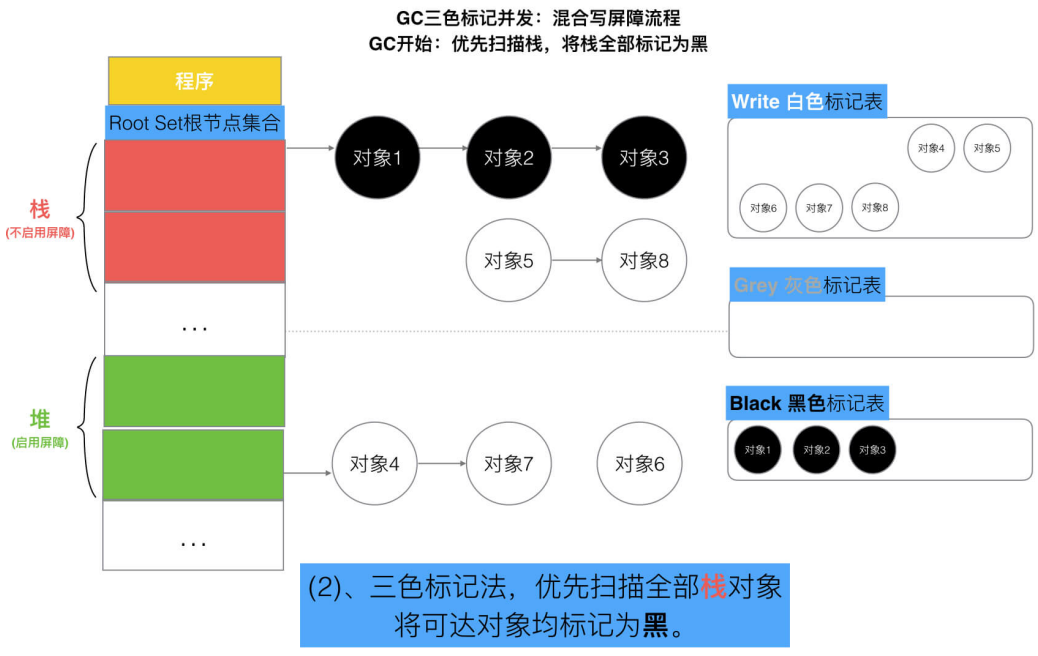
(2) 混合写屏障的具体场景分析

接下来，我们用几张图，来模拟整个一个详细的过程， 希望您能够更可观的看清晰整体流程。

> 注意混合写屏障是Gc的一种屏障机制，所以只是当程序执行GC的时候，才会触发这种机制。

GC开始：扫描栈区，将可达对象全部标记为黑

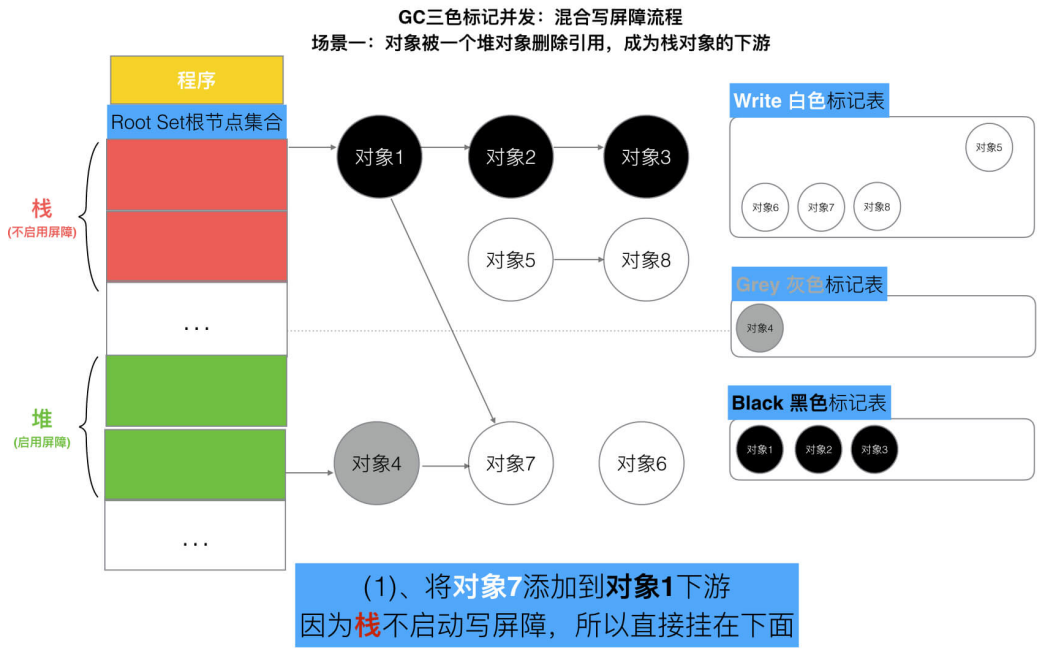


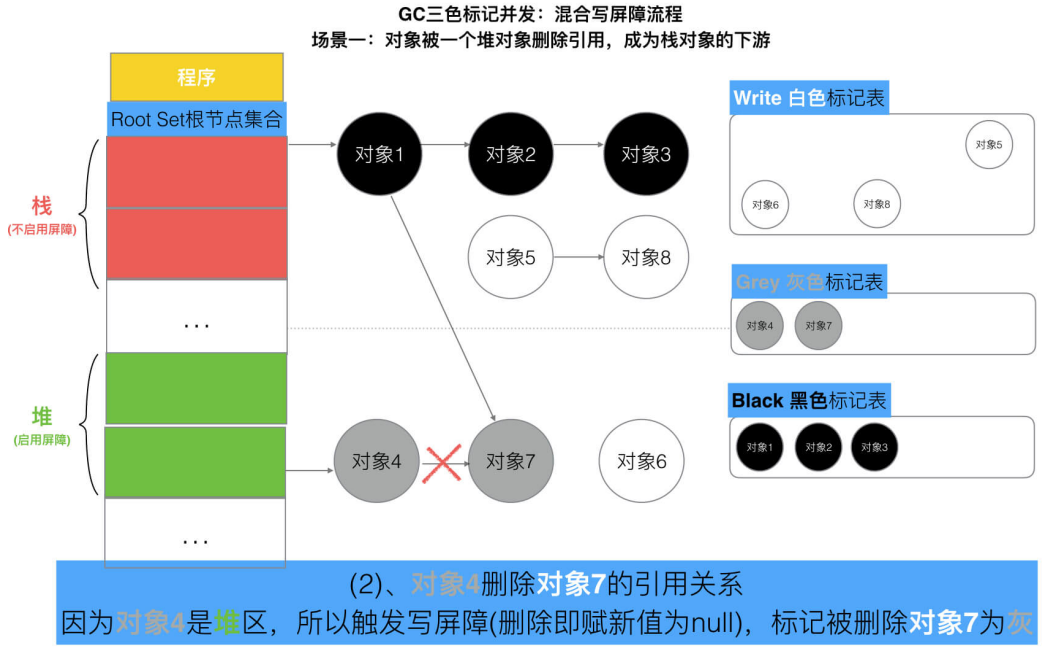


场景一： 对象被一个堆对象删除引用，成为栈对象的下游

> 伪代码

```
//前提：堆对象4->对象7 = 对象7;   //对象7 被 对象4引用
栈对象1->对象7 = 堆对象7;   //将堆对象7 挂在 栈对象1 下游
堆对象4->对象7 = null;      //对象4 删除引用 对象7
```

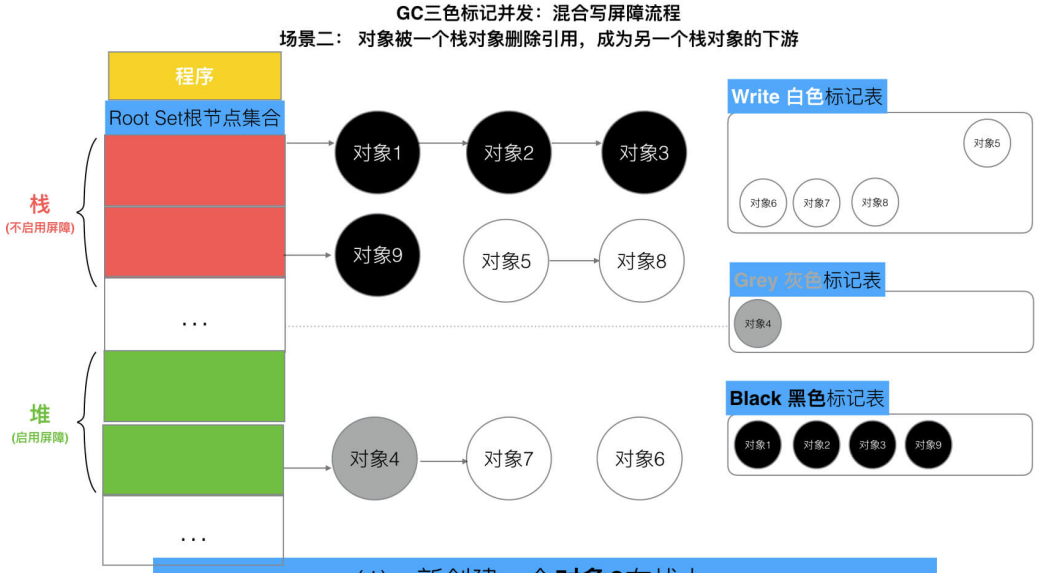




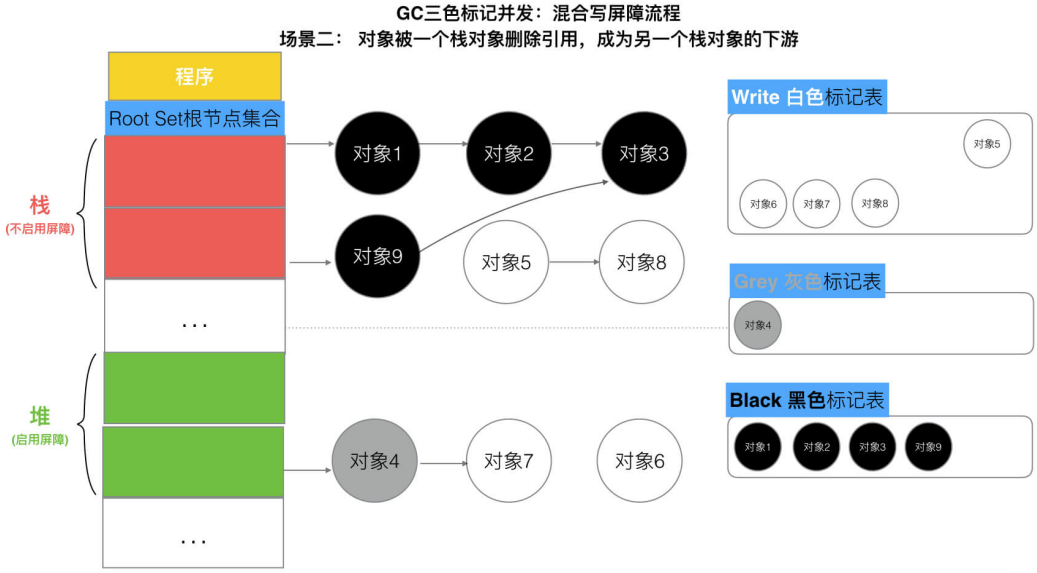
场景二： 对象被一个栈对象删除引用，成为另一个栈对象的下游

> 伪代码

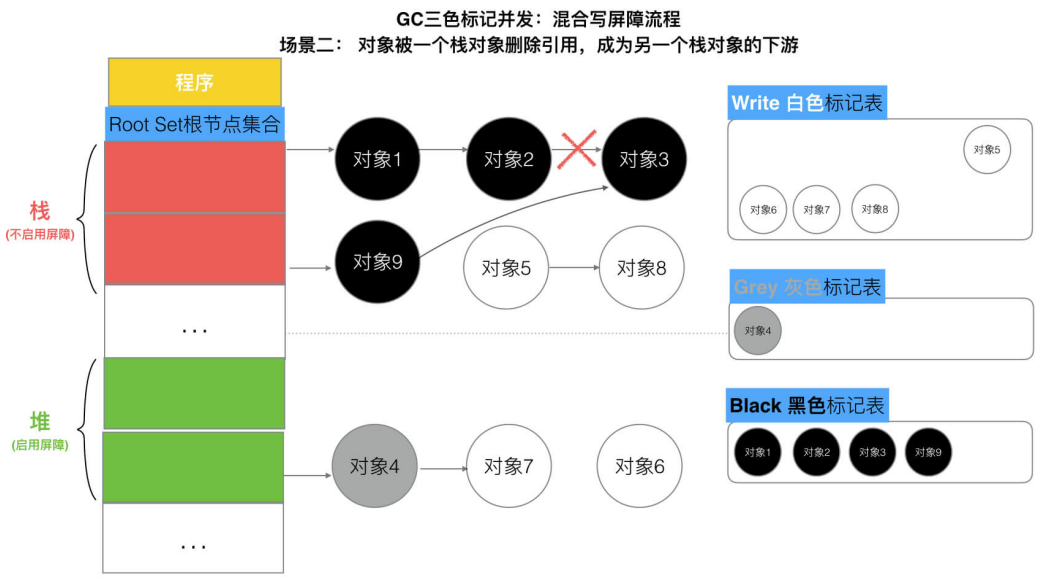
```
new 栈对象9;
对象8->对象3 = 对象3;        //将栈对象3 挂在 栈对象9 下游
对象2->对象3 = null;         //对象2 删除引用 对象3
```



(1)、新创建一个**对象9**在栈上。
(混合写屏障模式中，GC过程中任何新创建的对象均标记为**黑色**)



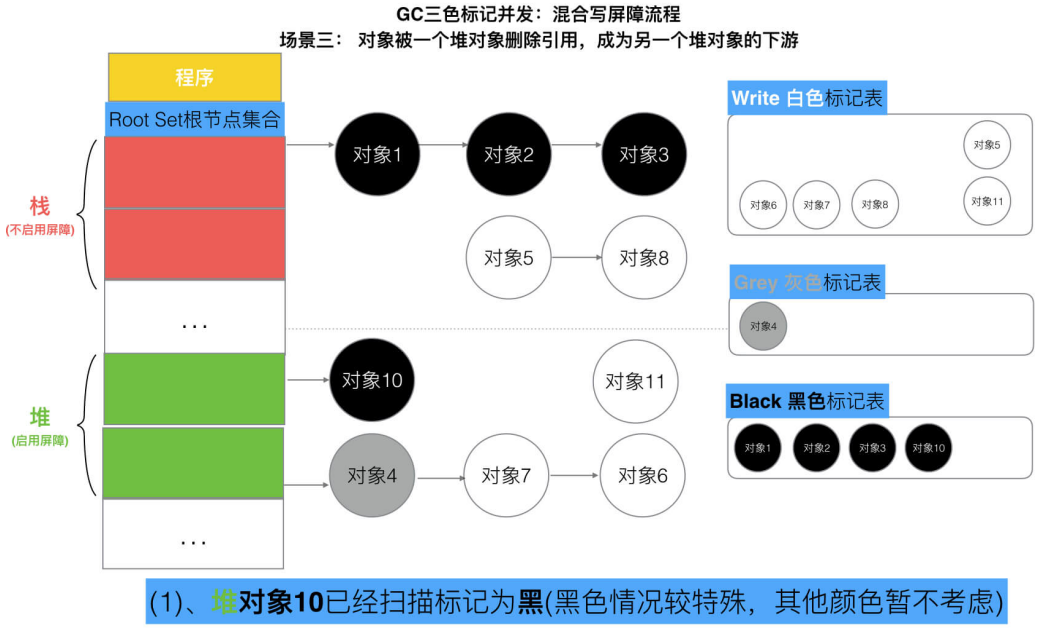
(2)、**对象9**添加下游引用栈**对象3**(直接添加，栈不启动屏障，无屏障效果)

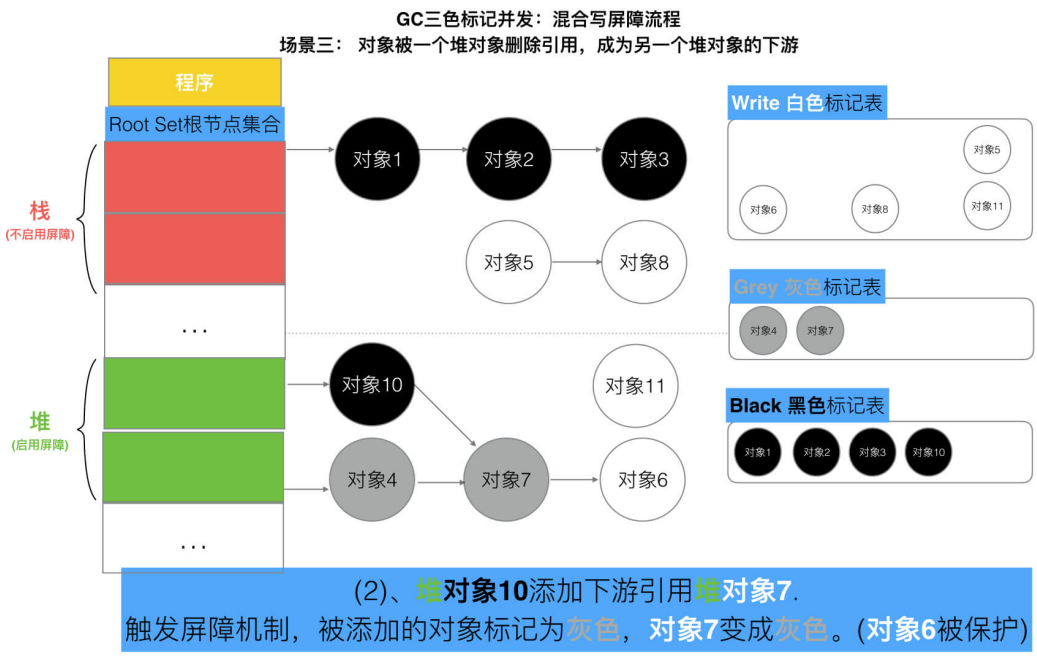


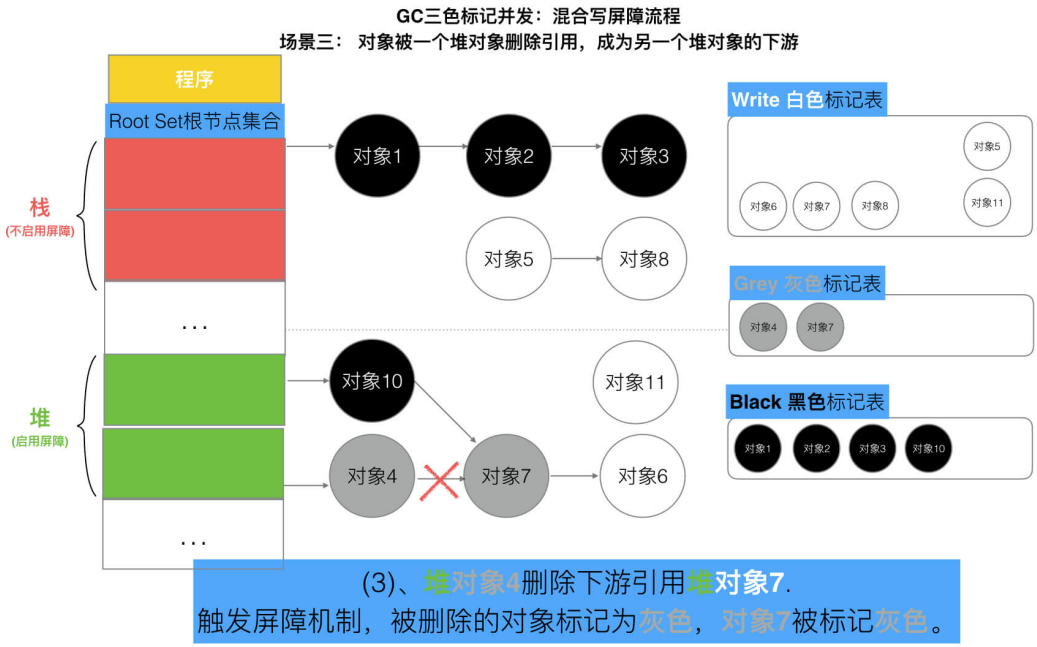
场景三：对象被一个堆对象删除引用，成为另一个堆对象的下游

> 伪代码

```
堆对象10->对象7 = 堆对象7;          //将堆对象7 挂在 堆对象10 下游
堆对象4->对象7 = null;              //对象4 删除引用 对象7
```



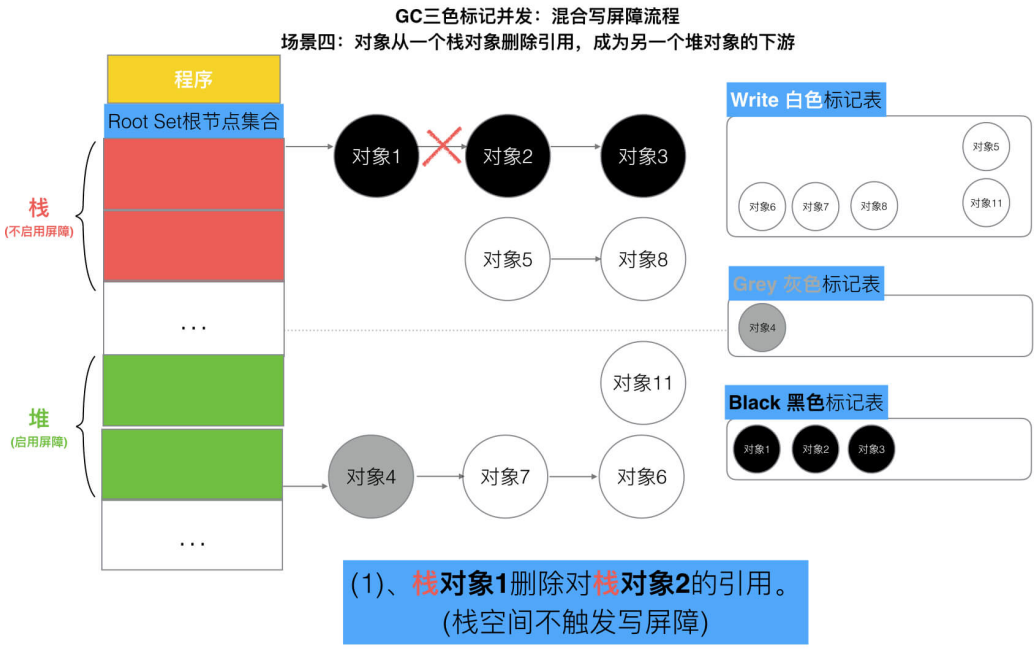


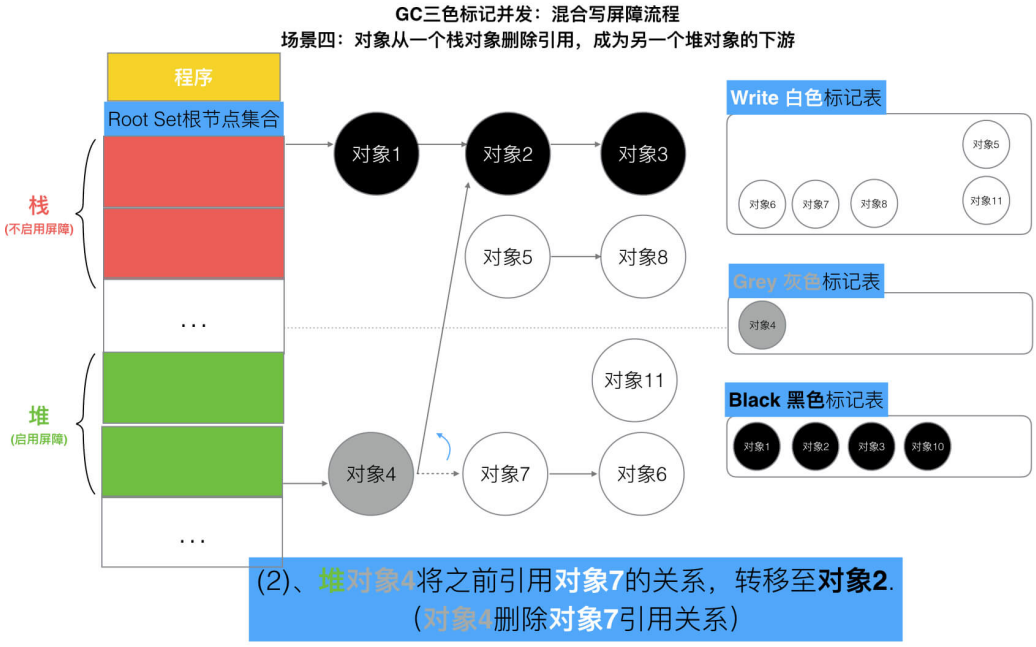


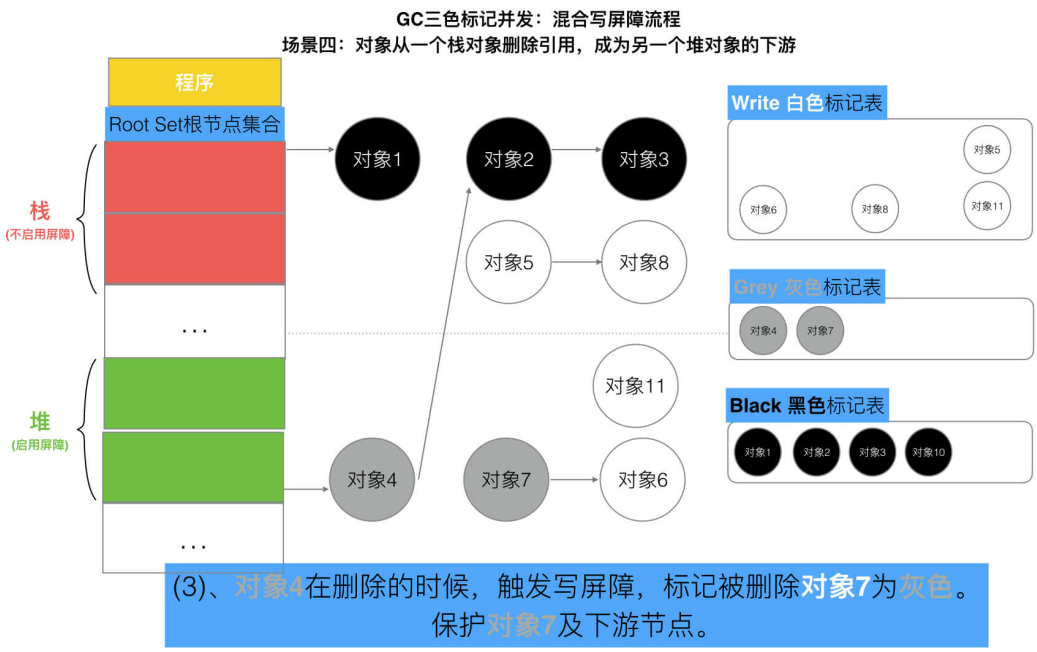
场景四：对象从一个栈对象删除引用，成为另一个堆对象的下游

> 伪代码

```
堆对象10->对象7 = 堆对象7;          //将堆对象7 挂在 堆对象10 下游
堆对象4->对象7 = null;              //对象4 删除引用 对象7
```


GC三色标记并发：混合写屏障流程
场景四：对象从一个栈对象删除引用，成为另一个堆对象的下游
程序
Root Set根节点集合
栈
(不启用屏障)
堆
(启用屏障)
对象1
对象2
对象3
对象5
对象8
对象11
对象4
对象7
对象6
Write 白色标记表
Grey 灰色标记表
Black 黑色标记表
(1)、栈对象1删除对栈对象2的引用。
(栈空间不触发写屏障)


GC三色标记并发：混合写屏障流程
场景四：对象从一个栈对象删除引用，成为另一个堆对象的下游
程序
Root Set根节点集合
栈
(不启用屏障)
堆
(启用屏障)
对象1
对象2
对象3
对象5
对象8
对象11
对象4
对象7
对象6
Write 白色标记表
Grey 灰色标记表
Black 黑色标记表
(2)、堆对象4将之前引用对象7的关系，转移至对象2.
(对象4删除对象7引用关系)



Golang中的混合写屏障满足 弱三色不变式 ，结合了删除写屏障和插入写屏障的优点，只需要在开始时并发扫描各个goroutine的栈，使其变黑并一直保持，这个过程不需要STW，而标记结束后，因为栈在扫描后始终是黑色的，也无需再进行re-scan操作了，减少了STW的时间。

七、总结

以上便是Golang的GC全部的标记-清除逻辑及场景演示全过程。

GoV1.3- 普通标记清除法，整体过程需要启动STW，效率极低。

GoV1.5- 三色标记法， 堆空间启动写屏障，栈空间不启动，全部扫描之后，需要重新扫描一次栈(需要STW)，效率普通

GoV1.8-三色标记法，混合写屏障机制， 栈空间不启动，堆空间启动。整个过程几乎不需要STW，效率较高。

面向对象的编程思维理解interface

一、 interface接口

interface 是GO语言的基础特性之一。可以理解为一种类型的规范或者约定。它跟java，C# 不太一样，不需要显示说明实现了某个接口，它没有继承或子类或“implements”关键字，只是通过约定的形式，隐式的实现interface 中的方法即可。因此，Golang 中的 interface 让编码更灵活、易扩展。

如何理解go 语言中的**interface** ？ 只需记住以下三点即可：

1. interface 是方法声明的集合
2. 任何类型的对象实现了在interface 接口中声明的全部方法，则表明该类型实现了该接口。
3. interface 可以作为一种数据类型，实现了该接口的任何对象都可以给对应的接口类型变量赋值。

> 注意：
> a. interface 可以被任意对象实现，一个类型/对象也可以实现多个 interface
> b. 方法不能重载，如 `eat(), eat(s string)` 不能同时存在

```
package main

import "fmt"

type Phone interface {
    call()
}

type NokiaPhone struct {
}

func (nokiaPhone NokiaPhone) call() {
    fmt.Println("I am Nokia, I can call you!")
}

type ApplePhone struct {
}

func (iPhone ApplePhone) call() {
    fmt.Println("I am Apple Phone, I can call you!")
}

func main() {
    var phone Phone
    phone = new(NokiaPhone)
    phone.call()

    phone = new(ApplePhone)
    phone.call()
}
```

上述中体现了 `interface` 接口的语法，在 `main` 函数中，也体现了 `多态` 的特性。
同样一个 `phone` 的抽象接口，分别指向不同的实体对象，调用的call()方法，打印的效果不同，那么就是体现出了多态的特性。

二、 面向对象中的开闭原则

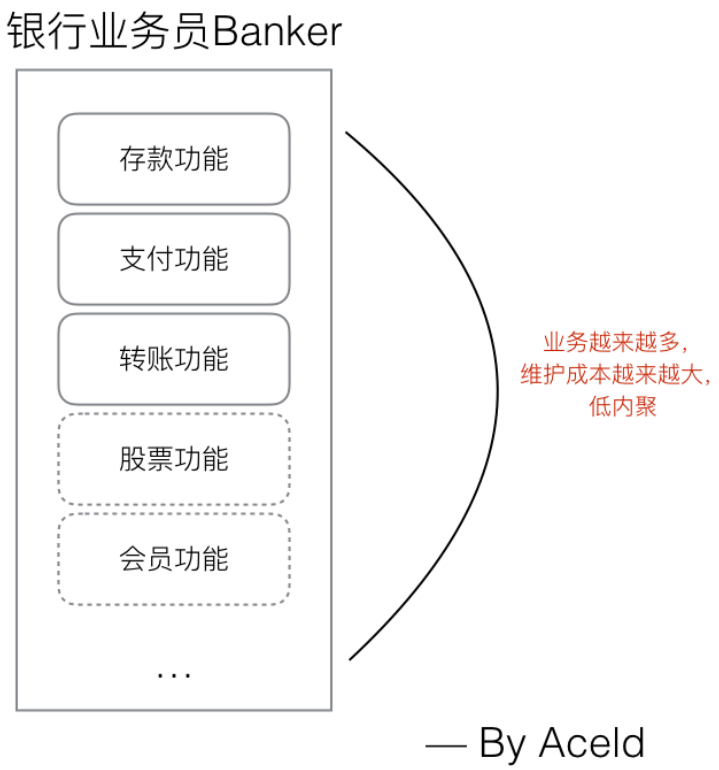
2.1 平铺式的模块设计

那么作为 `interface` 数据类型，他存在的意义在哪呢？ 实际上是为了满足一些面向对象的编程思想。我们知道，软件设计的最高目标就是 `高内聚，低耦合` 。那么其中有一个设计原则叫 `开闭原则` 。什么是开闭原则呢，接下来我们看一个例子：

```
package main

import "fmt"

//我们要写一个类,Banker银行业务员
type Banker struct {
}

//存款业务
func (this *Banker) Save() {
    fmt.Println("进行了 存款业务...")
}

//转账业务
func (this *Banker) Transfer() {
    fmt.Println("进行了 转账业务...")
}

//支付业务
func (this *Banker) Pay() {
    fmt.Println("进行了 支付业务...")
}

func main() {
    banker := &Banker{}

    banker.Save()
    banker.Transfer()
    banker.Pay()
}
```

代码很简单，就是一个银行业务员，他可能拥有很多的业务，比如 `Save()` 存款、 `Transfer()` 转账、 `Pay()` 支付等。那么如果这个业务员模块只有这几个方法还好，但是随着我们的程序写的越来越复杂，银行业务员可

能就要增加方法，会导致业务员模块越来越臃肿。

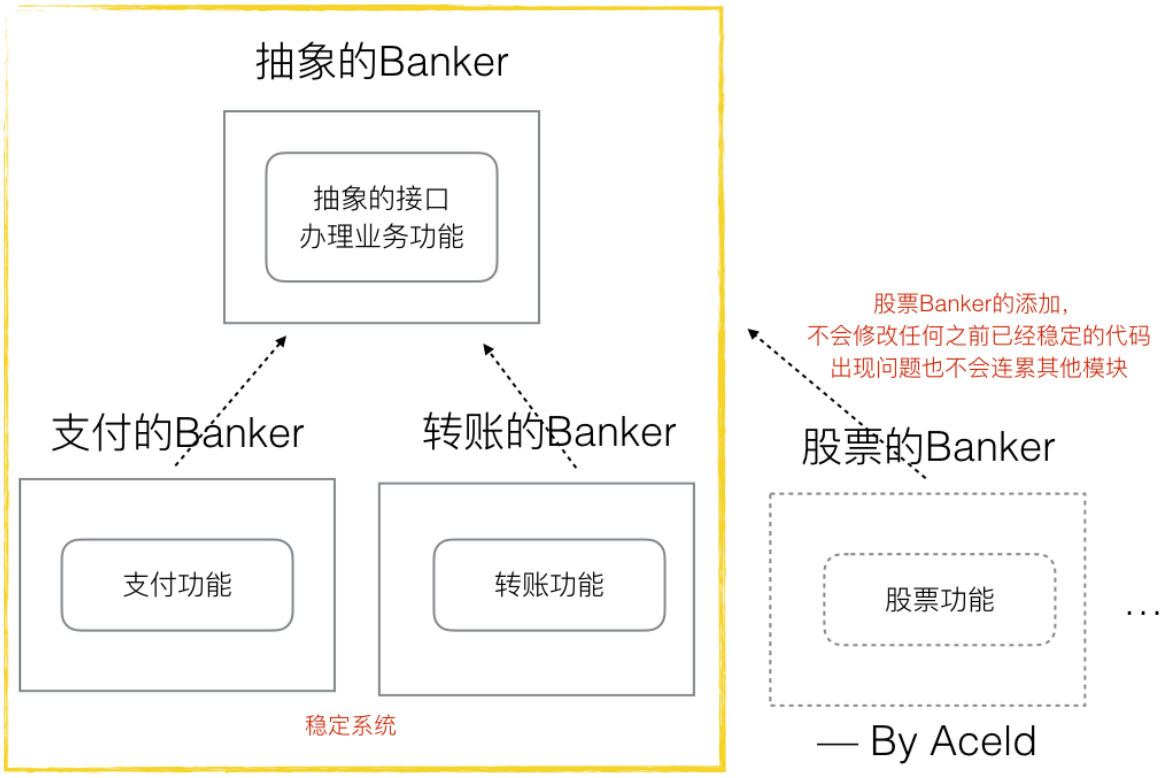


这样的设计会导致，当我们去给Banker添加新的业务的时候，会直接修改原有的Banker代码，那么Banker模块的功能会越来越多，出现问题的几率也就越来越大，假如此时Banker已经有99个业务了，现在我们要添加第100个业务，可能由于一次的不小心，导致之前99个业务也一起崩溃，因为所有的业务都在一个Banker类里，他们的耦合度太高，Banker的职责也不够单一，代码的维护成本随着业务的复杂正比成倍增大。

2.2 开闭原则设计

那么，如果我们拥有接口，`interface`这个东西，那么我们就可以抽象一层出来，制作一个抽象的Banker模块，然后提供一个抽象的方法。 分别根据这个抽象模块，去实现`支付Banker（实现支付方法）`，`转账Banker（实现转账方法）`

如下：



那么依然可以搞定程序的需求。 然后，当我们想要给Banker添加额外功能的时候，之前我们是直接修改Banker的内容，现在我们可以单独定义一个 `股票Banker(实现股票方法)` ，到这个系统中。 而且股票Banker的实现成功或者失败都不会影响之前的稳定系统，他很单一，而且独立。

所以以上，当我们给一个系统添加一个功能的时候，不是通过修改代码，而是通过增添代码来完成，那么就是开闭原则的核心思想了。所以要想满足上面的要求，是一定需要interface来提供一层抽象的接口的。

golang代码实现如下:

```
package main

import "fmt"

//抽象的银行业务员
type AbstractBanker interface{
    DoBusi()     //抽象的处理业务接口
}

//存款的业务员
type SaveBanker struct {
    //AbstractBanker
}

func (sb *SaveBanker) DoBusi() {
    fmt.Println("进行了存款")
}

//转账的业务员
type TransferBanker struct {
```

```
    //AbstractBanker
}

func (tb *TransferBanker) DoBusi() {
    fmt.Println("进行了转账")
}

//支付的业务员
type PayBanker struct {
    //AbstractBanker
}

func (pb *PayBanker) DoBusi() {
    fmt.Println("进行了支付")
}


func main() {
    //进行存款
    sb := &SaveBanker{}
    sb.DoBusi()

    //进行转账
    tb := &TransferBanker{}
    tb.DoBusi()

    //进行支付
    pb := &PayBanker{}
    pb.DoBusi()

}
```

当然我们也可以根据 `AbstractBanker` 设计一个小框架

```
//实现架构层(基于抽象层进行业务封装-针对interface接口进行封装)
func BankerBusiness(banker AbstractBanker) {
    //通过接口来向下调用，(多态现象)
    banker.DoBusi()
}
```

那么main中可以如下实现业务调用:

```
func main() {
    //进行存款
    BankerBusiness(&SaveBanker{})

    //进行存款
    BankerBusiness(&TransferBanker{})

    //进行存款
    BankerBusiness(&PayBanker{})
}
```

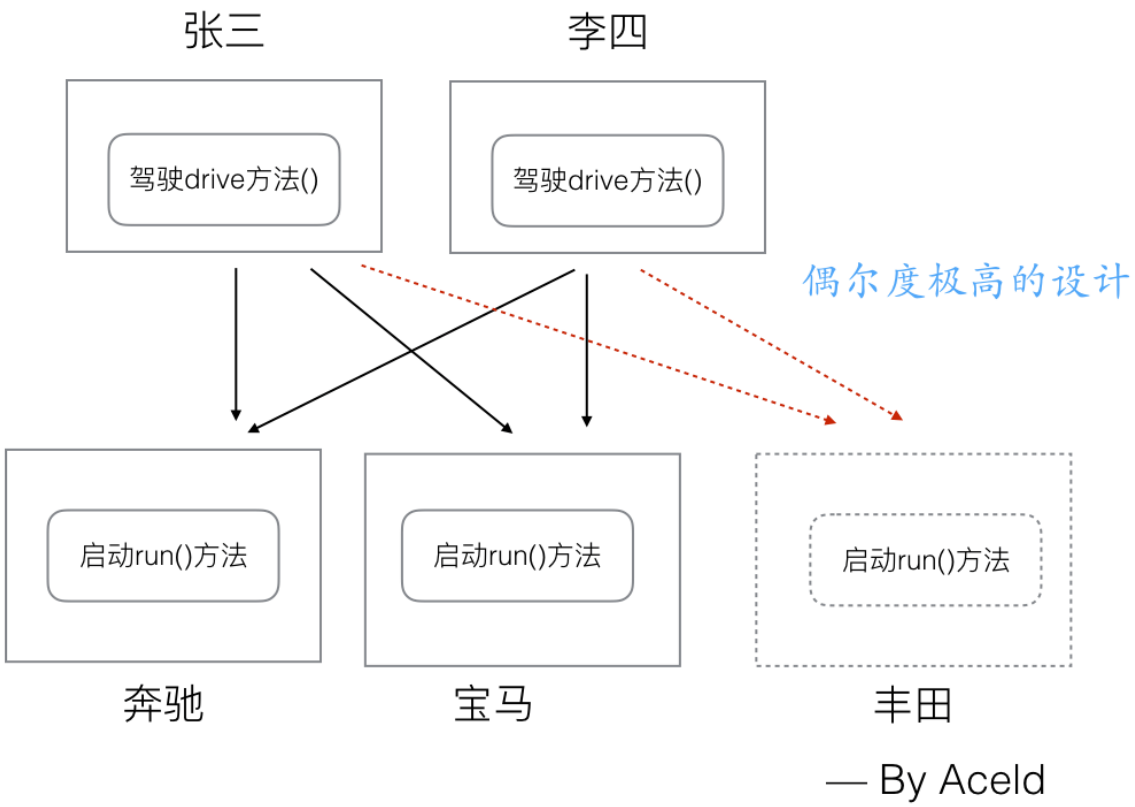
> 再看开闭原则定义:
> 开闭原则:一个软件实体如类、模块和函数应该对扩展开放，对修改关闭。
> 简单的说就是在修改需求的时候，应该尽量通过扩展来实现变化，而不是通过修改已有代码来实现变化。

三、 接口的意义

好了，现在interface已经基本了解，那么接口的意义最终在哪里呢，想必现在你已经有了一个初步的认知，实际上接口的最大的意义就是实现多态的思想，就是我们可以根据interface类型来设计API接口，那么这种API接口的适应能力不仅能适应当下所实现的全部模块，也适应未来实现的模块来进行调用。 `调用未来` 可能就是接口的最大意义所在吧，这也是为什么架构师那么值钱，因为良好的架构师是可以针对interface设计一套框架，在未来许多年却依然适用。

四、 面向对象中的依赖倒转原则

4.1 耦合度极高的模块关系设计

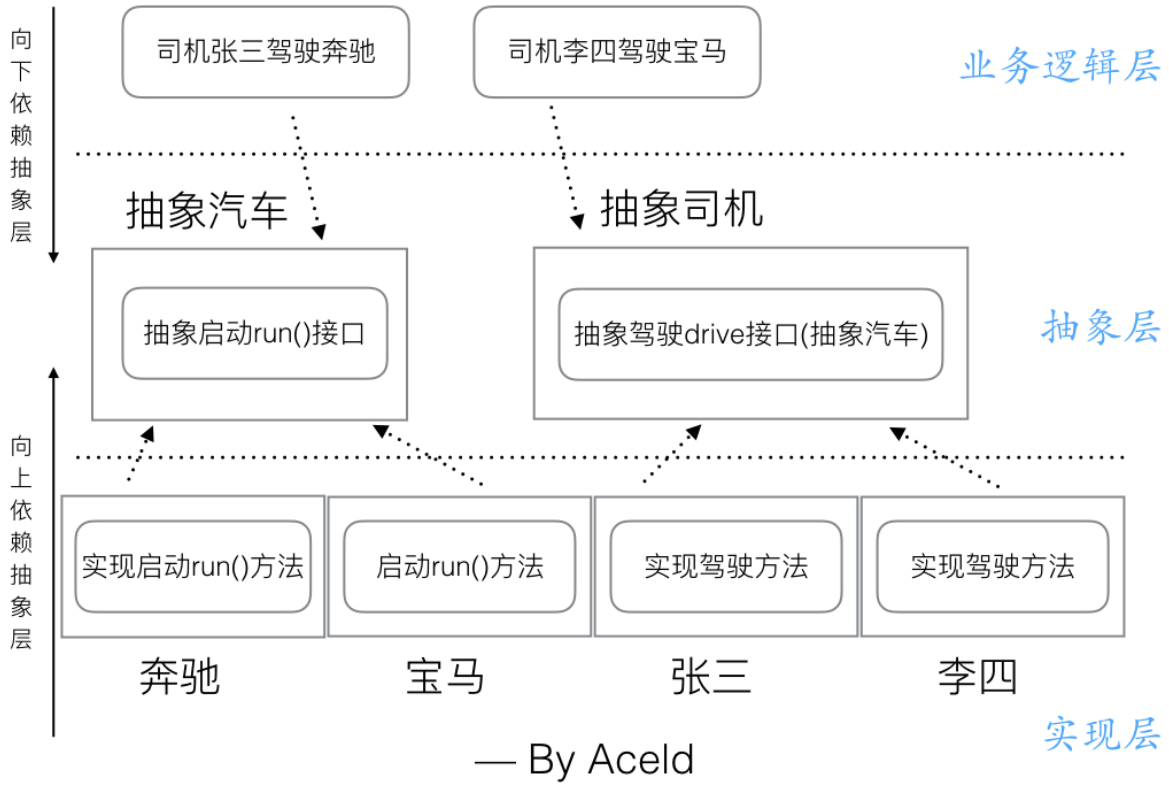



```
package main

import "fmt"

// ===> 奔驰汽车 <===
type Benz struct {

}

func (this *Benz) Run() {
    fmt.Println("Benz is running...")
}

// ===> 宝马汽车 <===
type BMW struct {

}
```

```
func (this *BMW) Run() {
    fmt.Println("BMW is running ...")
}


//===> 司机张三 <===
type Zhang3 struct {
    //...
}

func (zhang3 *Zhang3) DriveBenZ(benz *Benz) {
    fmt.Println("zhang3 Drive Benz")
    benz.Run()
}

func (zhang3 *Zhang3) DriveBMW(bmw *BMW) {
    fmt.Println("zhang3 drive BMW")
    bmw.Run()
}

//===> 司机李四 <===
type Li4 struct {
    //...
}

func (li4 *Li4) DriveBenZ(benz *Benz) {
    fmt.Println("li4 Drive Benz")
    benz.Run()
}

func (li4 *Li4) DriveBMW(bmw *BMW) {
    fmt.Println("li4 drive BMW")
    bmw.Run()
}

func main() {
    //业务1 张3开奔驰
    benz := &Benz{}
    zhang3 := &Zhang3{}
    zhang3.DriveBenZ(benz)

    //业务2 李四开宝马
    bmw := &BMW{}
    li4 := &Li4{}
    li4.DriveBMW(bmw)
}
```

我们来看上面的代码和图中每个模块之间的依赖关系，实际上并没有用到任何的 `interface` 接口层的代码，显然最后我们的两个业务 `张三开奔驰` ， `李四开宝马` ，程序中也都实现了。但是这种设计的问题就在于，小规模没什么问题，但是一旦程序需要扩展，比如我现在要增加一个 `丰田汽车` 或者 司机 `王五` ， 那么模块和模块的依赖关系将成指数级递增，想蜘蛛网一样越来越难维护和捋顺。

4.2 面向抽象层依赖倒转



如上图所示，如果我们在设计一个系统的时候，将模块分为3个层次，抽象层、实现层、业务逻辑层。那么，我们首先将抽象层的模块和接口定义出来，这里就需要了 `interface` 接口的设计，然后我们依照抽象层，依次实现每个实现层的模块，在我们写实现层代码的时候，实际上我们只需要参考对应的抽象层实现就好了，实现每个模块，也和其他的实现的模块没有关系，这样也符合了上面介绍的开闭原则。这样实现起来每个模块只依赖对象的接口，而和其他模块没关系，依赖关系单一。系统容易扩展和维护。

我们在指定业务逻辑也是一样，只需要参考抽象层的接口来业务就好了，抽象层暴露出来的接口就是我们业务层可以使用的方法，然后可以通过多态的线下，接口指针指向哪个实现模块，调用了就是具体的实现方法，这样我们业务逻辑层也是依赖抽象成编程。

我们就将这种的设计原则叫做 `依赖倒转原则` 。

来一起看一下修改的代码：

```
package main

import "fmt"

// =====>  抽象层  <========
type Car interface {
    Run()
}

type Driver interface {
    Drive(car Car)
}

// =====>  实现层  <========
type BenZ struct {
    //...
```

```
}

func (benz * BenZ) Run() {
    fmt.Println("Benz is running...")
}

type Bmw struct {
    //...
}

func (bmw * Bmw) Run() {
    fmt.Println("Bmw is running...")
}

type Zhang_3 struct {
    //...
}

func (zhang3 *Zhang_3) Drive(car Car) {
    fmt.Println("Zhang3 drive car")
    car.Run()
}

type Li_4 struct {
    //...
}

func (li4 *Li_4) Drive(car Car) {
    fmt.Println("li4 drive car")
    car.Run()
}


// =====>  业务逻辑层  <========
func main() {
    //张3 开 宝马
    var bmw Car
    bmw = &Bmw{}

    var zhang3 Driver
    zhang3 = &Zhang_3{}

    zhang3.Drive(bmw)

    //李4 开 奔驰
    var benz Car
    benz = &BenZ{}

    var li4 Driver
    li4 = &Li_4{}

    li4.Drive(benz)
}
```

4.3 依赖倒转小练习

> 模拟组装2台电脑，
> — 抽象层 —有显卡Card 方法display，有内存Memory 方法storage，有处理器CPU 方法calculate
> — 实现层层 —有 Intel因特尔公司 、产品有(显卡、内存、CPU)，有 Kingston 公司， 产品有(内存3)，有 NVIDIA 公司， 产品有(显卡)

> — 逻辑层 —1. 组装一台Intel系列的电脑，并运行，2. 组装一台 Intel CPU Kingston内存 NVIDIA显卡的电脑，并运行

```
/*
    模拟组装2台电脑
    --- 抽象层 ---
    有显卡Card  方法display
    有内存Memory 方法storage
    有处理器CPU   方法calculate

    --- 实现层层 ---
    有 Intel因特尔公司 、产品有(显卡、内存、CPU)
    有 Kingston 公司， 产品有(内存3)
    有 NVIDIA 公司， 产品有(显卡)

    --- 逻辑层 ---
    1. 组装一台Intel系列的电脑，并运行
    2. 组装一台 Intel CPU  Kingston内存 NVIDIA显卡的电脑，并运行
*/
package main

import "fmt"

//------  抽象层 -----
type Card interface{
    Display()
}

type Memory interface {
    Storage()
}

type CPU interface {
    Calculate()
}

type Computer struct {
    cpu CPU
    mem Memory
    card Card
}

func NewComputer(cpu CPU, mem Memory, card Card) *Computer{
    return &Computer{
        cpu:cpu,
        mem:mem,
        card:card,
    }
}

func (this *Computer) DoWork() {
    this.cpu.Calculate()
    this.mem.Storage()
    this.card.Display()
}

//------  实现层 -----
//intel
type IntelCPU struct {
    CPU
}
```

```
func (this *IntelCPU) Calculate() {
    fmt.Println("Intel CPU 开始计算了...")
}

type IntelMemory struct {
    Memory
}

func (this *IntelMemory) Storage() {
    fmt.Println("Intel Memory 开始存储了...")
}

type IntelCard struct {
    Card
}

func (this *IntelCard) Display() {
    fmt.Println("Intel Card 开始显示了...")
}

//kingston
type KingstonMemory struct {
    Memory
}

func (this *KingstonMemory) Storage() {
    fmt.Println("Kingston memory storage...")
}

//nvidia
type NvidiaCard struct {
    Card
}

func (this *NvidiaCard) Display() {
    fmt.Println("Nvidia card display...")
}



//------  业务逻辑层 -----
func main() {
    //intel系列的电脑
    com1 := NewComputer(&IntelCPU{}, &IntelMemory{}, &IntelCard{})
    com1.DoWork()

    //杂牌子
    com2 := NewComputer(&IntelCPU{}, &KingstonMemory{}, &NvidiaCard{})
    com2.DoWork()
}
```

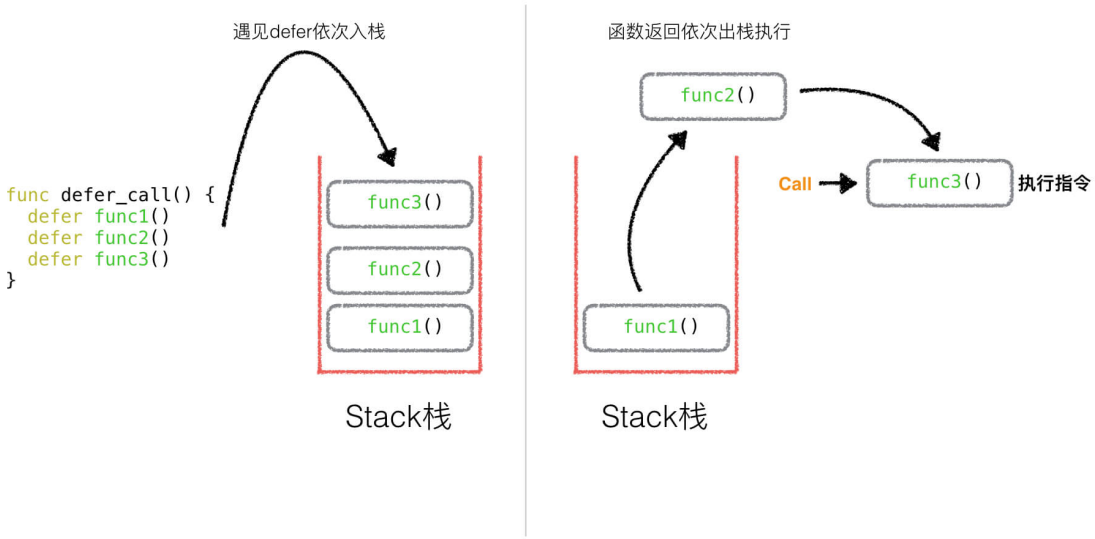
Golang中的Defer必掌握的7知识点

知识点1：defer的执行顺序

多个defer出现的时候，**它是一个“栈”的关系，也就是先进后出**。一个函数中，写在前面的defer会比写在后面的defer调用的晚。

> 示例代码

```
package main

import "fmt"

func main() {
    defer func1()
    defer func2()
    defer func3()
}

func func1() {
    fmt.Println("A")
}

func func2() {
    fmt.Println("B")
}

func func3() {
    fmt.Println("C")
}
```



输出结果：

```
C
B
A
```

知识点2: defer与return谁先谁后

> 示例代码

```
package main

import "fmt"

func deferFunc() int {
    fmt.Println("defer func called")
    return 0
}

func returnFunc() int {
    fmt.Println("return func called")
    return 0
}

func returnAndDefer() int {

    defer deferFunc()

    return returnFunc()
}

func main() {
    returnAndDefer()
}
```

执行结果为：

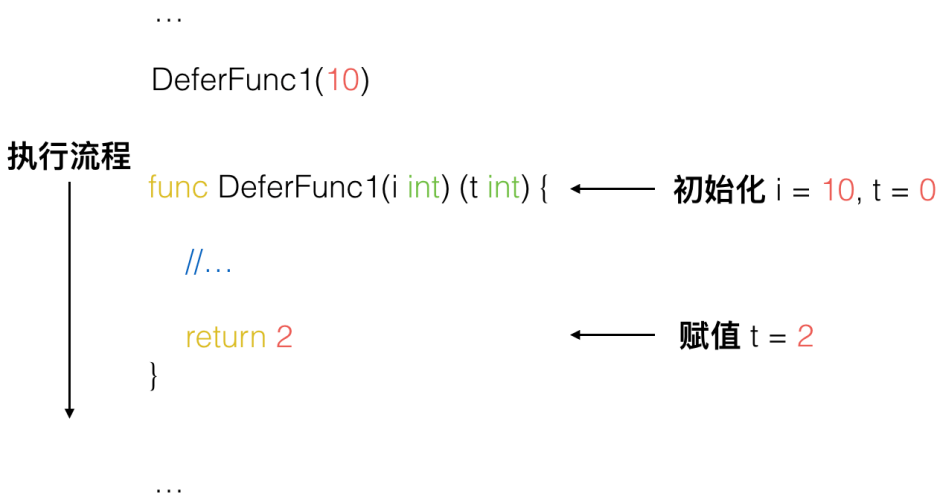
```
return func called
defer func called
```

结论为：**return之后的语句先执行，defer后的语句后执行**

知识点3：函数的返回值初始化

该知识点不属于defer本身，但是调用的场景却与defer有联系，所以也算是defer必备了解的知识点之一。

如： `func DeferFunc1(i int) (t int) {}`
其中返回值 `t int` ，这个 `t` 会在函数起始处被初始化为对应类型的零值并且作用域为整个函数。



> 示例代码

```
package main

import "fmt"

func DeferFunc1(i int) (t int) {

    fmt.Println("t = ", t)

    return 2
}

func main() {
    DeferFunc11(10)
}
```

结果

```
t =  0
```

证明，**只要声明函数的返回值变量名称，就会在函数初始化时候为之赋值为0，而且在函数体作用域可见**。

知识点4: 有名函数返回值遇见defer情况

在没有defer的情况下，其实函数的返回就是与return一致的，但是有了defer就不一样了。

我们通过**知识点2**得知，先return，再defer，所以在执行完return之后，还要再执行defer里的语句，依然可以修改本应该返回的结果。

```
package main

import "fmt"

func returnButDefer() (t int) {  //t初始化0，  并且作用域为该函数全域
```

```
    defer func() {
        t = t * 10
    }()

    return 1
}

func main() {
    fmt.Println(returnButDefer())
}
```

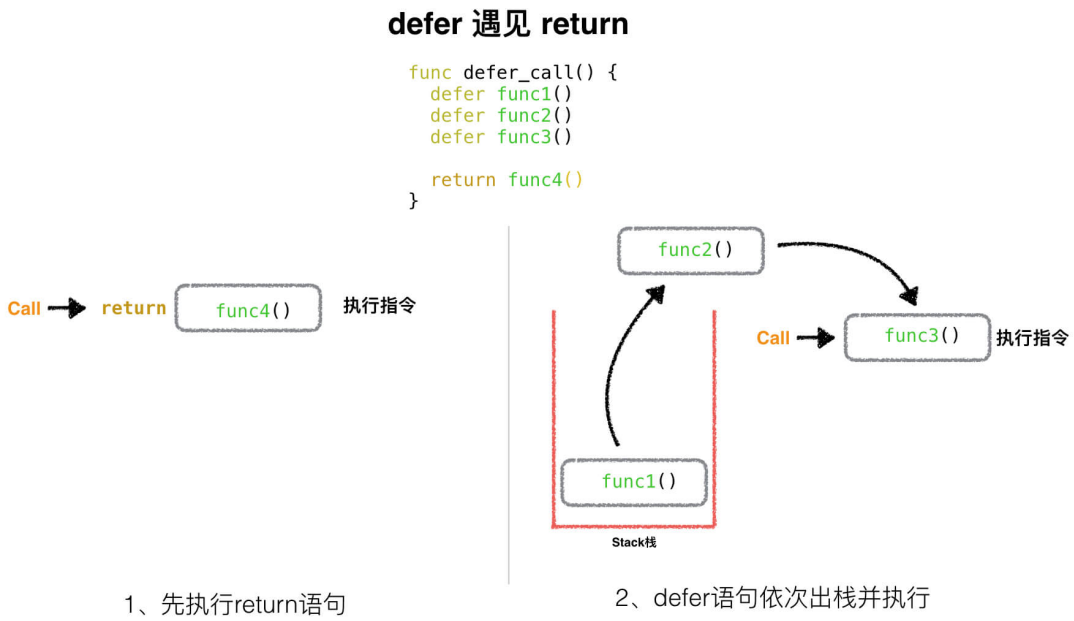
该 `returnButDefer()` 本应的返回值是 `1` ，但是在return之后，又被defer的匿名func函数执行，所以 `t=t*10` 被执行，最后 `returnButDefer()` 返回给上层 `main()` 的结果为 `10`

```
$ go run test.go
10
```

知识点5: defer遇见panic

我们知道，能够触发defer的是遇见return(或函数体到末尾)和遇见panic。

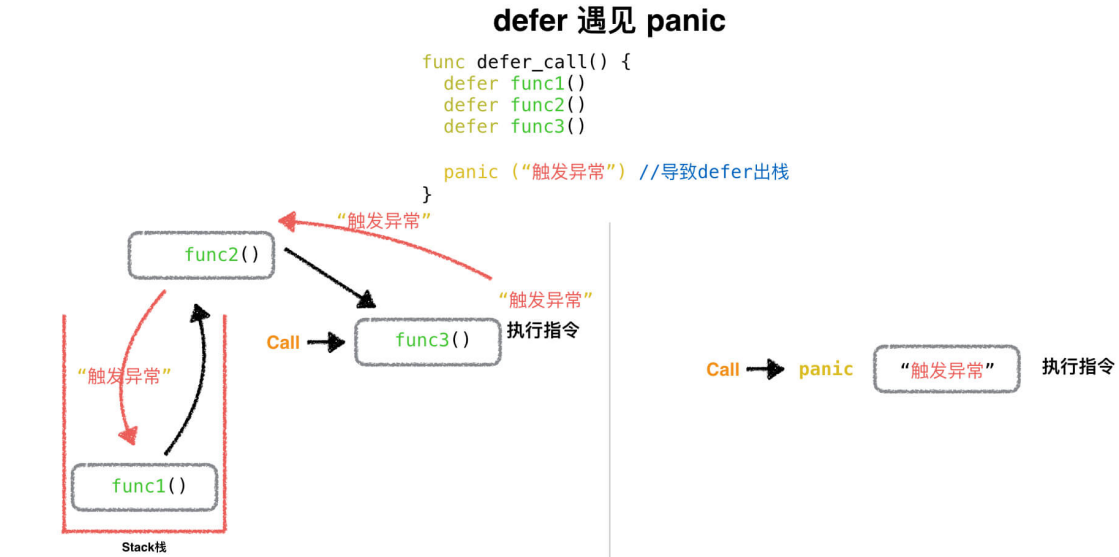
根据**知识点2**，我们知道，defer遇见return情况如下：



1、先执行return语句

2、defer语句依次出栈并执行

那么，遇到panic时，遍历本协程的defer链表，并执行defer。在执行defer过程中:遇到recover则停止panic，返回recover处继续往下执行。如果没有遇到recover，遍历完本协程的defer链表后，向stderr抛出panic信息。



1、遇见panic，强行defer出栈
并且每个defer均受到异常，不捕获则转移到下一个

2、直到defer捕获异常，或者上层显示异常

A. defer遇见panic，但是并不捕获异常的情况

test10.go

```
package main

import (
    "fmt"
)

func main() {
    defer_call()

    fmt.Println("main 正常结束")
}

func defer_call() {
    defer func() { fmt.Println("defer: panic 之前1") }()
    defer func() { fmt.Println("defer: panic 之前2") }()

    panic("异常内容")  //触发defer出栈

    defer func() { fmt.Println("defer: panic 之后，永远执行不到") }()
}
```

结果

```
defer: panic 之前2
defer: panic 之前1
panic: 异常内容
//... 异常堆栈信息
```

B. defer遇见panic，并捕获异常

```
package main

import (
    "fmt"
)

func main() {
    defer_call()

    fmt.Println("main 正常结束")
}

func defer_call() {

    defer func() {
        fmt.Println("defer: panic 之前1, 捕获异常")
        if err := recover(); err != nil {
            fmt.Println(err)
        }
    }()

    defer func() { fmt.Println("defer: panic 之前2, 不捕获") }()

    panic("异常内容")  //触发defer出栈

    defer func() { fmt.Println("defer: panic 之后, 永远执行不到") }()
}
```

结果

```
defer: panic 之前2, 不捕获
defer: panic 之前1, 捕获异常
异常内容
main 正常结束
```

defer 最大的功能是 panic 后依然有效
所以defer可以保证你的一些资源一定会被关闭，从而避免一些异常出现的问题。

知识点6: defer中包含panic

> 编译执行下面代码会出现什么?

> test16.go

```
package main

import (
    "fmt"
)

func main()  {

    defer func() {
        if err := recover(); err != nil{
            fmt.Println(err)
        }else {
```

```
            fmt.Println("fatal")
        }
    }()

    defer func() {
        panic("defer panic")
    }()

    panic("panic")
}
```

结果

```
defer panic
```

分析

panic仅有最后一个可以被revover捕获。

触发 `panic("panic")` 后defer顺序出栈执行，第一个被执行的defer中 会有 `panic("defer panic")` 异常语句，这个异常将会覆盖掉main中的异常 `panic("panic")` ，最后这个异常被第二个执行的defer捕获到。

知识点7: defer下的函数参数包含子函数

```
package main

import "fmt"

func function(index int, value int) int {

    fmt.Println(index)

    return index
}

func main() {
    defer function(1, function(3, 0))
    defer function(2, function(4, 0))
}
```

这里，有4个函数，他们的index序号分别为1，2，3，4。

那么这4个函数的先后执行顺序是什么呢？这里面有两个defer， 所以defer一共会压栈两次，先进栈1，后进栈2。 那么在压栈function1的时候，需要连同函数地址、函数形参一同进栈，那么为了得到function1的第二个参数的结果，所以就需要先执行function3将第二个参数算出，那么function3就被第一个执行。同理压栈function2，就需要执行function4算出function2第二个参数的值。然后函数结束，先出栈fuction2、再出栈function1.

所以顺序如下：

- defer压栈function1，压栈函数地址、形参1、形参2(调用function3) -> 打印3
- defer压栈function2，压栈函数地址、形参1、形参2(调用function4) -> 打印4
- defer出栈function2，调用function2 -> 打印2
- defer出栈function1，调用function1-> 打印1

```
3
4
2
1
```

练习：defer面试真题

了解以上6个defer的知识点，我们来验证一下网上的真题吧。

下面代码输出什么？

> test11.go

```
package main

import "fmt"

func DeferFunc1(i int) (t int) {
    t = i
    defer func() {
        t += 3
    }()
    return t
}

func DeferFunc2(i int) int {
    t := i
    defer func() {
        t += 3
    }()
    return t
}

func DeferFunc3(i int) (t int) {
    defer func() {
        t += i
    }()
    return 2
}

func DeferFunc4() (t int) {
    defer func(i int) {
        fmt.Println(i)
        fmt.Println(t)
    }(t)
    t = 1
    return 2
}

func main() {
    fmt.Println(DeferFunc1(1))
    fmt.Println(DeferFunc2(1))
    fmt.Println(DeferFunc3(1))
    DeferFunc4()
}
```

练习题分析

DeferFunc1

```
func DeferFunc1(i int) (t int) {
    t = i
    defer func() {
        t += 3
    }()
    return t
}
```

1. 将返回值t赋值为传入的i，此时t为1
2. 执行return语句将t赋值给t（等于啥也没做）
3. 执行defer方法，将t + 3 = 4
4. 函数返回 4
   因为t的作用域为整个函数所以修改有效。

DeferFunc2

```
func DeferFunc2(i int) int {
    t := i
    defer func() {
        t += 3
    }()
    return t
}
```

1. 创建变量t并赋值为1
2. 执行return语句，注意这里是将t赋值给返回值，此时返回值为1（这个返回值并不是t）
3. 执行defer方法，将t + 3 = 4
4. 函数返回返回值1

也可以按照如下代码理解

```
func DeferFunc2(i int) (result int) {
    t := i
    defer func() {
        t += 3
    }()
    return t
}
```

上面的代码return的时候相当于将t赋值给了result，当defer修改了t的值之后，对result是不会造成影响的。

DeferFunc3

```
func DeferFunc3(i int) (t int) {
    defer func() {
        t += i
    }()
    return 2
}
```

1. 首先执行return将返回值t赋值为2
2. 执行defer方法将t + 1

3. 最后返回 3

DeferFunc4

```
func DeferFunc4() (t int) {
    defer func(i int) {
        fmt.Println(i)
        fmt.Println(t)
    }(t)
    t = 1
    return 2
}
```

1. 初始化返回值t为零值 0
2. 首先执行defer的第一步，赋值defer中的func入参t为0
3. 执行defer的第二步，将defer压栈
4. 将t赋值为1
5. 执行return语句，将返回值t赋值为2
6. 执行defer的第三步，出栈并执行
   因为在入栈时defer执行的func的入参已经赋值了，此时它作为的是一个形式参数，所以打印为0；相对应的因为最后已经将t的值修改为2，所以再打印一个2

结果

```
4
1
3
0
2
```

精通Golang项目依赖Go modules

一、什么是Go Modules?

Go modules 是 Go 语言的依赖解决方案，发布于 Go1.11，成长于 Go1.12，丰富于 Go1.13，正式于 Go1.14 推荐在生产上使用。

Go moudles 目前集成在 Go 的工具链中，只要安装了 Go，自然而然也就可以使用 Go moudles 了，而 Go modules 的出现也解决了在 Go1.11 前的几个常见争议问题：

1. Go 语言长久以来的依赖管理问题。
2. “淘汰”现有的 GOPATH 的使用模式。
3. 统一社区中的其它的依赖管理工具（提供迁移功能）。

二、GOPATH的工作模式

Go Modoules的目的之一就是淘汰GOPATH, 那么GOPATH是个什么?

为什么在 Go1.11 前就使用 GOPATH，而 Go1.11 后就开始逐步建议使用 Go modules，不再推荐 GOPATH 的模式了呢？

(1) Wait is GOPATH?

```
$ go env

GOPATH="/home/itheima/go"
...
```

我们输入 `go env` 命令行后可以查看到 GOPATH 变量的结果，我们进入到该目录下进行查看，如下：

```
go
├── bin
├── pkg
└── src
    ├── github.com
    ├── golang.org
    ├── google.golang.org
    ├── gopkg.in
   ....
```

GOPATH目录下一共包含了三个子目录，分别是：

- bin：存储所编译生成的二进制文件。
- pkg：存储预编译的目标文件，以加快程序的后续编译速度。
- src：存储所有 `.go` 文件或源代码。在编写 Go 应用程序，程序包和库时，一般会以 `$GOPATH/src/github.com/foo/bar` 的路径进行存放。

因此在使用 GOPATH 模式下，我们需要将应用代码存放在固定的 `$GOPATH/src` 目录下，并且如果执行 `go get` 来拉取外部依赖会自动下载并安装到 `$GOPATH` 目录下。

(2) GOPATH模式的弊端

在 GOPATH 的 `$GOPATH/src` 下进行 `.go` 文件或源代码的存储，我们可以称其为 GOPATH 的模式，这个模式拥有一些弊端.

- **A. 无版本控制概念.** 在执行`go get`的时候，你无法传达任何的版本信息的期望，也就是说你也无法知道自己当前更新的是哪一个版本，也无法通过指定来拉取自己所期望的具体版本。

- **B.无法同步一致第三方版本号.** 在运行 Go 应用程序的时候，你无法保证其它人与你所期望依赖的第三方库是相同的版本，也就是说在项目依赖库的管理上，你无法保证所有人的依赖版本都一致。
- *C.无法指定当前项目引用的第三方版本号.* 你没办法处理 v1、v2、v3 等等不同版本的引用问题，因为 GOPATH 模式下的导入路径都是一样的，都是`github.com/foo/bar`。

三、Go Modules模式

我们接下来用Go Modules的方式创建一个项目, 建议为了与GOPATH分开,不要将项目创建在`GOPATH/src`下.

(1) go mod命令

命令	作用
go mod init	生成 go.mod 文件
go mod download	下载 go.mod 文件中指明的所有依赖
go mod tidy	整理现有的依赖
go mod graph	查看现有的依赖结构
go mod edit	编辑 go.mod 文件
go mod vendor	导出项目所有的依赖到vendor目录
go mod verify	校验一个模块是否被篡改过
go mod why	查看为什么需要依赖某模块

(2) go mod环境变量

可以通过`go env`命令来进行查看

```
$ go env
GO111MODULE="auto"
GOPROXY="https://proxy.golang.org,direct"
GONOPROXY=""
GOSUMDB="sum.golang.org"
GONOSUMDB=""
GOPRIVATE=""
...
```

GO111MODULE

Go语言提供了`GO111MODULE`这个环境变量来作为 Go modules 的开关，其允许设置以下参数：

- auto：只要项目包含了 go.mod 文件的话启用 Go modules，目前在 Go1.11 至 Go1.14 中仍然是默认值。

- on：启用 Go modules，推荐设置，将会是未来版本中的默认值。
- off：禁用 Go modules，不推荐设置。

可以通过来设置

```
$ go env -w GO111MODULE=on
```

GOPROXY

这个环境变量主要是用于设置 Go 模块代理（Go module proxy）,其作用是用于使 Go 在后续拉取模块版本时直接通过镜像站点来快速拉取。

GOPROXY 的默认值是： `https://proxy.golang.org,direct`

`proxy.golang.org` 国内访问不了,需要设置国内的代理.

- 阿里云

  https://mirrors.aliyun.com/goproxy/

- 七牛云

  https://goproxy.cn,direct

如:

```
$ go env -w GOPROXY=https://goproxy.cn,direct
```

GOPROXY 的值是一个以英文逗号 “,” 分割的 Go 模块代理列表，允许设置多个模块代理，假设你不想使用，也可以将其设置为 “off” ，这将会禁止 Go 在后续操作中使用任何 Go 模块代理。

如:

```
$ go env -w GOPROXY=https://goproxy.cn,https://mirrors.aliyun.com/goproxy/,direct
```

> direct

而在刚刚设置的值中，我们可以发现值列表中有 “direct” 标识，它又有什么作用呢？

实际上 “direct” 是一个特殊指示符，用于指示 Go 回源到模块版本的源地址去抓取（比如 GitHub 等），场景如下：当值列表中上一个 Go 模块代理返回 404 或 410 错误时，Go 自动尝试列表中的下一个，遇见 “direct” 时回源，也就是回到源地址去抓取，而遇见 EOF 时终止并抛出类似 “invalid version: unknown revision...” 的错误。

GOSUMDB

它的值是一个 Go checksum database，用于在拉取模块版本时（无论是从源站拉取还是通过 Go module proxy 拉取）保证拉取到的模块版本数据未经过篡改，若发现不一致，也就是可能存在篡改，将会立即中止。

GOSUMDB 的默认值为： `sum.golang.org` ，在国内也是无法访问的，但是 GOSUMDB 可以被 Go 模块代理所代理（详见：Proxying a Checksum Database）。

因此我们可以通过设置 GOPROXY 来解决，而先前我们所设置的模块代理 `goproxy.cn` 就能支持代理 `sum.golang.org` ，所以这一个问题在设置 GOPROXY 后，你可以不需要过度关心。

另外若对 GOSUMDB 的值有自定义需求，其支持如下格式：

- 格式 1：`<SUMDB_NAME>+<PUBLIC_KEY>`。
- 格式 2：`<SUMDB_NAME>+<PUBLIC_KEY> <SUMDB_URL>`。

也可以将其设置为“off”，也就是禁止 Go 在后续操作中校验模块版本。

GONOPROXY/GONOSUMDB/GOPRIVATE

这三个环境变量都是用在当前项目依赖了私有模块，例如像是你公司的私有 git 仓库，又或是 github 中的私有库，都是属于私有模块，都是要进行设置的，否则会拉取失败。

更细致来讲，就是依赖了由 GOPROXY 指定的 Go 模块代理或由 GOSUMDB 指定 Go checksum database 都无法访问到的模块时的场景。

而一般**建议直接设置 GOPRIVATE，它的值将作为 GONOPROXY 和 GONOSUMDB 的默认值，所以建议的最佳姿势是直接使用 GOPRIVATE**。

并且它们的值都是一个以英文逗号 “,” 分割的模块路径前缀，也就是可以设置多个，例如：

```
$ go env -w GOPRIVATE="git.example.com,github.com/eddycjy/mquote"
```

设置后，前缀为 git.xxx.com 和 github.com/eddycjy/mquote 的模块都会被认为是私有模块。

如果不想每次都重新设置，我们也可以利用通配符，例如：

```
$ go env -w GOPRIVATE="*.example.com"
```

这样子设置的话，所有模块路径为 example.com 的子域名（例如：git.example.com）都将不经过 Go module proxy 和 Go checksum database，**需要注意的是不包括 example.com 本身**。

四、使用Go Modules初始化项目

(1) 开启Go Modules

```
$ go env -w GO111MODULE=on
```

又或是可以通过直接设置系统环境变量（写入对应的~/.bash_profile 文件亦可）来实现这个目的：

```
$ export GO111MODULE=on
```

(2) 初始化项目

创建项目目录

```
$ mkdir -p $HOME/aceld/modules_test
$ cd $HOME/aceld/modules_test
```

执行Go modules 初始化

```
$ go mod init github.com/aceld/modules_test
go: creating new go.mod: module github.com/aceld/modules_test
```

在执行 `go mod init` 命令时，我们指定了模块导入路径为 `github.com/aceld/modules_test` 。接下来我们在该项目根目录下创建 `main.go` 文件，如下：

```
package main

import (
    "fmt"
    "github.com/aceld/zinx/znet"
    "github.com/aceld/zinx/ziface"
)

//ping test 自定义路由
type PingRouter struct {
    znet.BaseRouter
}

//Ping Handle
func (this *PingRouter) Handle(request ziface.IRequest) {
    //先读取客户端的数据
    fmt.Println("recv from client : msgId=", request.GetMsgID(),
            ", data=", string(request.GetData()))

    //再回写ping...ping...ping
    err := request.GetConnection().SendBuffMsg(0, []byte("ping...ping...ping"))
    if err != nil {
      fmt.Println(err)
    }
}

func main() {
    //1 创建一个server句柄
    s := znet.NewServer()

    //2 配置路由
    s.AddRouter(0, &PingRouter{})

    //3 开启服务
    s.Serve()
}
```

OK, 我们先不要关注代码本身,我们看当前的main.go也就是我们的 `aceld/modules_test` 项目,是依赖一个叫 `github.com/aceld/zinx` 库的. `znet` 和 `ziface` 只是 `zinx` 的两个模块.

接下来我们在 `$HOME/aceld/modules_test` ,本项目的根目录执行

```
$ go get github.com/aceld/zinx/znet

go: downloading github.com/aceld/zinx v0.0.0-20200221135252-8a8954e75100
go: found github.com/aceld/zinx/znet in github.com/aceld/zinx v0.0.0-20200221135252-8a8954e75100
```

我们会看到 我们的 `go.mod` 被修改,同时多了一个 `go.sum` 文件.

(3) 查看go.mod文件

> aceld/modules_test/go.mod

```
module github.com/aceld/modules_test
```

```
go 1.14

require github.com/aceld/zinx v0.0.0-20200221135252-8a8954e75100 // indirect
```

我们来简单看一下这里面的关键字

`module` ：用于定义当前项目的模块路径

`go` :标识当前Go版本.即初始化版本

`require` ：当前项目依赖的一个特定的必须版本

`// indirect` ：示该模块为间接依赖，也就是在当前应用程序中的 import 语句中，并没有发现这个模块的明确引用，有可能是你先手动 `go get` 拉取下来的，也有可能是你所依赖的模块所依赖的.我们的代码很明显是依赖的 `"github.com/aceld/zinx/znet"` 和 `"github.com/aceld/zinx/ziface"` ,所以就间接的依赖了 `github.com/aceld/zinx`

(4) 查看go.sum文件

在第一次拉取模块依赖后，会发现多出了一个 go.sum 文件，其详细罗列了当前项目直接或间接依赖的所有模块版本，并写明了那些模块版本的 SHA-256 哈希值以备 Go 在今后的操作中保证项目所依赖的那些模块版本不会被篡改。

```
github.com/aceld/zinx v0.0.0-20200221135252-8a8954e75100 h1:Ez5iM6cKGMtqvIJ8nvR9h74Ln8FvFDgfb7bJIbrKv54=
github.com/aceld/zinx v0.0.0-20200221135252-8a8954e75100/go.mod h1:bMiERrPdR8FzpBOo86nhWWmeHJ1cCaqVvWKCGcDVJ5M=
github.com/golang/protobuf v1.3.3/go.mod h1:vzj43D7+SQXF/4pzW/hwtAqwc6iTitCiVSaWz5lYuqw=
```

我们可以看到一个模块路径可能有如下两种：

h1:hash情况

```
github.com/aceld/zinx v0.0.0-20200221135252-8a8954e75100 h1:Ez5iM6cKGMtqvIJ8nvR9h74Ln8FvFDgfb7bJIbrKv54=
```

go.mod hash情况

```
github.com/aceld/zinx v0.0.0-20200221135252-8a8954e75100/go.mod h1:bMiERrPdR8FzpBOo86nhWWmeHJ1cCaqVvWKCGcDVJ5M=
github.com/golang/protobuf v1.3.3/go.mod h1:vzj43D7+SQXF/4pzW/hwtAqwc6iTitCiVSaWz5lYuqw=
```

h1 hash 是 Go modules 将目标模块版本的 zip 文件开包后，针对所有包内文件依次进行 hash，然后再把它们的 hash 结果按照固定格式和算法组成总的 hash 值。

而 h1 hash 和 go.mod hash 两者，要不就是同时存在，要不就是只存在 go.mod hash。那什么情况下会不存在 h1 hash 呢，就是当 Go 认为肯定用不到某个模块版本的时候就会省略它的 h1 hash，就会出现不存在 h1 hash，只存在 go.mod hash 的情况。

五、修改模块的版本依赖关系

为了作尝试,假定我们现在都zinx版本作了升级，由 `zinx v0.0.0-20200221135252-8a8954e75100` 升级到 `zinx v0.0.0-20200306023939-bc416543ae24` (注意zinx是一个没有打版本tag打第三方库,如果有的版本号是有tag的,那么可以直接对应v后面的版本号即可)

那么,我们是怎么知道zinx做了升级呢，我们又是如何知道的最新的 `zinx` 版本号是多少呢?

先回到 `$HOME/aceld/modules_test` ,本项目的根目录执行

```
$ go get github.com/aceld/zinx/znet
go: downloading github.com/aceld/zinx v0.0.0-20200306023939-bc416543ae24
go: found github.com/aceld/zinx/znet in github.com/aceld/zinx v0.0.0-20200306023939-bc416543ae24
go: github.com/aceld/zinx upgrade => v0.0.0-20200306023939-bc416543ae24
```

这样我们,下载了最新的zinx, 版本是 `v0.0.0-20200306023939-bc416543ae24`

然后,我么看一下go.mod

```
module github.com/aceld/modules_test

go 1.14

require github.com/aceld/zinx v0.0.0-20200306023939-bc416543ae24 // indirect
```

我们会看到,当我们执行 `go get` 的时候, 会自动的将本地将当前项目的 `require` 更新了.变成了最新的依赖.

好了, 现在我们就要做另外一件事,就是,我们想用一个旧版本的zinx. 来修改当前 `zinx` 模块的依赖版本号.

目前我们在 `$GOPATH/pkg/mod/github.com/aceld` 下,已经有了两个版本的zinx库

```
/go/pkg/mod/github.com/aceld$ ls
zinx@v0.0.0-20200221135252-8a8954e75100
zinx@v0.0.0-20200306023939-bc416543ae24
```

目前,我们 `/aceld/modules_test` 依赖的是zinx@v0.0.0-20200306023939-bc416543ae24"">`zinx@v0.0.0-20200306023939-bc416543ae24` 这个是最新版, 我们要改成之前的版本 zinx@v0.0.0-20200306023939-bc416543ae24`.

回到 `/aceld/modules_test` 项目目录下,执行

```
$ go mod edit -replace=zinx@v0.0.0-20200306023939-bc416543ae24=zinx@v0.0.0-20200221135252-8a8954e75100
```

然后我们打开go.mod查看一下

```
module github.com/aceld/modules_test

go 1.14

require github.com/aceld/zinx v0.0.0-20200306023939-bc416543ae24 // indirect

replace zinx v0.0.0-20200306023939-bc416543ae24 => zinx v0.0.0-20200221135252-8a8954e75100
```

这里出现了 `replace` 关键字.用于将一个模块版本替换为另外一个模块版本。

Golang面试之路

数据定义

(1).函数返回值问题

> 下面代码是否可以编译通过?

> test1.go

```
package main

/*
    下面代码是否编译通过?
*/
func myFunc(x,y int)(sum int,error){
    return x+y,nil
}

func main() {
    num, err := myFunc(1, 2)
    fmt.Println("num = ", num)
}
```

答案:

编译报错理由:

```
# command-line-arguments
./test1.go:6:21: syntax error: mixed named and unnamed function parameters
```

> 考点：函数返回值命名

> 结果：编译出错。

> 在函数有多个返回值时，只要有一个返回值有指定命名，其他的也必须有命名。 如果返回值有有多个返回值必须加上括号； 如果只有一个返回值并且有命名也需要加上括号； 此处函数第一个返回值有sum名称，第二个未命名，所以错误。

(2).结构体比较问题

> 下面代码是否可以编译通过？为什么？

> test2.go

```
package main

import "fmt"

func main() {

    sn1 := struct {
```

```
        age  int
        name string
    }{age: 11, name: "qq"}

    sn2 := struct {
        age  int
        name string
    }{age: 11, name: "qq"}

    if sn1 == sn2 {
        fmt.Println("sn1 == sn2")
    }

    sm1 := struct {
        age int
        m   map[string]string
    }{age: 11, m: map[string]string{"a": "1"}}

    sm2 := struct {
        age int
        m   map[string]string
    }{age: 11, m: map[string]string{"a": "1"}}

    if sm1 == sm2 {
        fmt.Println("sm1 == sm2")
    }
}
```

结果

编译不通过

```
./test2.go:31:9: invalid operation: sm1 == sm2 (struct containing map[string]string cannot be compared)
```

考点:**结构体比较**

> **结构体比较规则注意1**：只有相同类型的结构体才可以比较，结构体是否相同不但与属性类型个数有关，还与属性顺序相关.

比如：

```
sn1 := struct {
    age  int
    name string
}{age: 11, name: "qq"}

sn3:= struct {
    name string
    age  int
}{age:11, name:"qq"}
```

`sn3` 与 `sn1` 就不是相同的结构体了，不能比较。

> **结构体比较规则注意2**：结构体是相同的，但是结构体属性中有不可以比较的类型，如 `map` ，`slice` ，则结构体不能用 `==` 比较。

可以使用reflect.DeepEqual进行比较

```
if reflect.DeepEqual(sm1, sm2) {
        fmt.Println("sm1 == sm2")
} else {
        fmt.Println("sm1 != sm2")
}
```

(3).string与nil类型

> 下面代码是否能够编译通过？为什么？

> test3.go

```
package main

import (
    "fmt"
)

func GetValue(m map[int]string, id int) (string, bool) {
    if _, exist := m[id]; exist {
        return "存在数据", true
    }
    return nil, false
}

func main()  {
    intmap:=map[int]string{
        1:"a",
        2:"bb",
        3:"ccc",
    }

    v,err:=GetValue(intmap,3)
    fmt.Println(v,err)
}
```

考点：**函数返回值类型**

答案：编译不会通过。

分析：

nil 可以用作 interface、function、pointer、map、slice 和 channel 的“空值”。但是如果不特别指定的话，Go 语言不能识别类型，所以会报错。通常编译的时候不会报错，但是运行是时候会报：`cannot use nil as type string in return argument` .

所以将 `GetValue` 函数改成如下形式就可以了

```
func GetValue(m map[int]string, id int) (string, bool) {
    if _, exist := m[id]; exist {
        return "存在数据", true
    }
    return "不存在数据", false
}
```

(4) 常量

> 下面函数有什么问题？

> test4.go

```
package main

const cl = 100

var bl = 123

func main()  {
    println(&bl,bl)
    println(&cl,cl)
}
```

解析

考点:**常量**
常量不同于变量的在运行期分配内存，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，

```
cannot take the address of cl
```

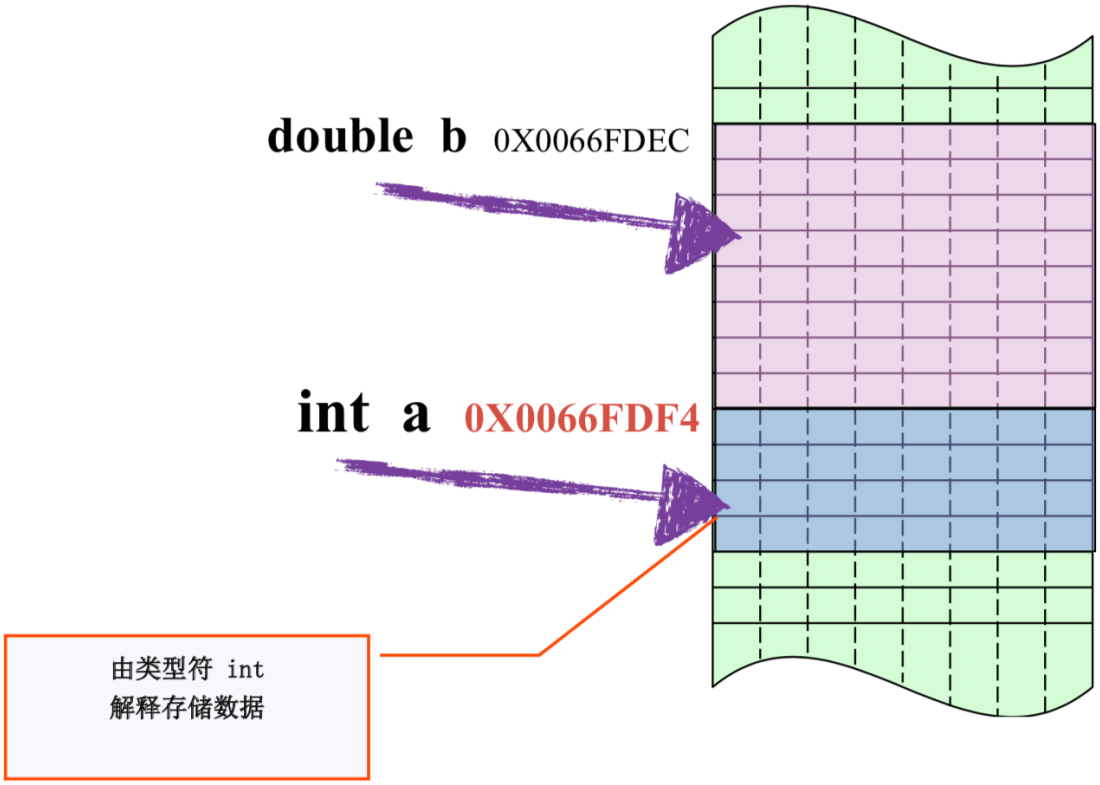
内存四区概念：

A.数据类型本质：

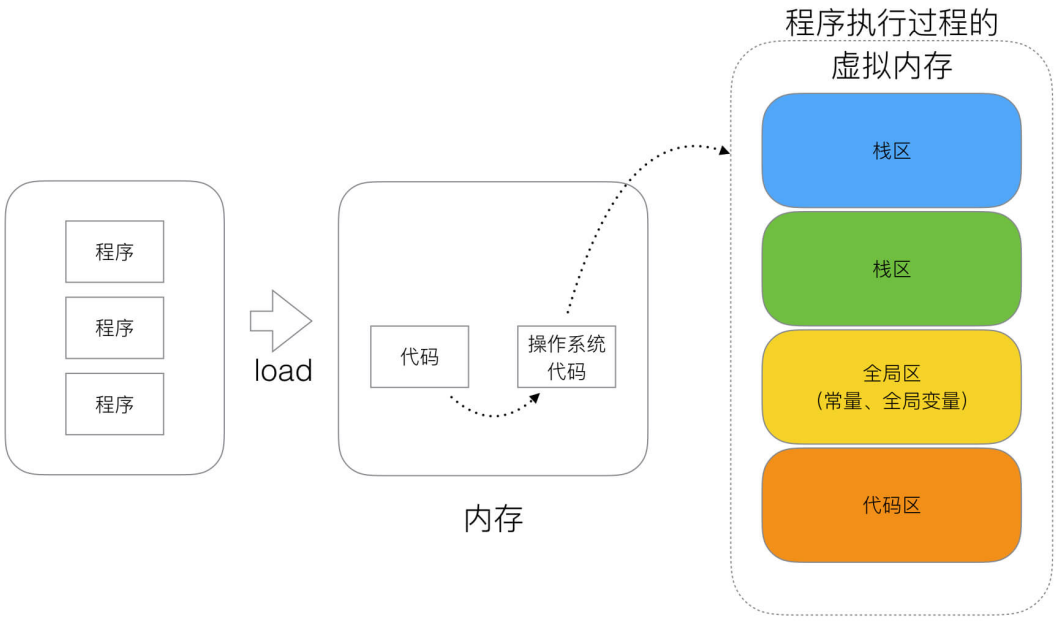
固定内存大小的别名

B. 数据类型的作用：

编译器预算对象(变量)分配的内存空间大小。



C. 内存四区



流程说明

1、操作系统把物理硬盘代码load到内存

2、操作系统把c代码分成四个区

3、操作系统找到main函数入口执行

栈区(Stack)：

空间较小，要求数据读写性能高，数据存放时间较短暂。由编译器自动分配和释放，存放函数的参数值、函数的调用流程方法地址、局部变量等(局部变量如果产生逃逸现象，可能会挂在在堆区)

堆区(heap):

空间充裕，数据存放时间较久。一般由开发者分配及释放(但是Golang中会根据变量的逃逸现象来选择是否分配到栈上或堆上)，启动Golang的GC由GC清除机制自动回收。

全局区-静态全局变量区:

全局变量的开辟是在程序在 `main` 之前就已经放在内存中。而且对外完全可见。即作用域在全部代码中，任何同包代码均可随时使用，在变量会搞混淆，而且在局部函数中如果同名称变量使用 `:=` 赋值会出现编译错误。

全局变量最终在进程退出时，由操作系统回收。

> 我么在开发的时候，尽量减少使用全局变量的设计

全局区-常量区：

常量区也归属于全局区，常量为存放数值字面值单位，即不可修改。或者说的有的常量是直接挂钩字面值的。

比如:

```
const cl = 10
```

cl是字面量10的对等符号。

所以在golang中，常量是无法取出地址的，因为字面量符号并没有地址而言。

数组和切片

(1) 切片的初始化与追加

> 1.2 写出程序运行的结果

```
package main

import (
    "fmt"
)

func main(){
    s := make([]int, 10)

    s = append(s, 1, 2, 3)

    fmt.Println(s)
}
```

考点

切片追加, make初始化均为0

结果

```
[0 0 0 0 0 0 0 0 0 0 1 2 3]
```

(2) slice拼接问题

> 下面是否可以编译通过?

> test6.go

```
package main

import "fmt"

func main() {
    s1 := []int{1, 2, 3}
    s2 := []int{4, 5}
    s1 = append(s1, s2)
    fmt.Println(s1)
}
```

结果

编译失败

两个slice在append的时候，记住需要进行将第二个slice进行 `...` 打散再拼接。

```
s1 = append(s1, s2...)
```

(3) slice中new的使用

> 下面代码是否可以编译通过？

> test9.go

```
package main

import "fmt"

func main() {

    list := new([]int)

    list = append(list, 1)

    fmt.Println(list)
}
```

结果：

编译失败，`./test9.go:9:15: first argument to append must be slice; have *[]int`

分析：

> 切片指针的解引用。

> 可以使用list:=make([]int,0) list类型为切片

> 或使用*list = append(*\list, 1) list类型为指针

new和make的区别：

二者都是内存的分配（堆上），但是make只用于slice、map以及channel的初始化（非零值）；而new用于类型的内存分配，并且内存置为零。所以在我们编写程序的时候，就可以根据自己的需要很好的选择了。

make返回的还是这三个引用类型本身；而new返回的是指向类型的指针。

Map

(1) Map的Value赋值

> 下面代码编译会出现什么结果？

> test7.go

```
package main

import "fmt"

type Student struct {
    Name string
}

var list map[string]Student

func main() {

    list = make(map[string]Student)

    student := Student{"Aceld"}

    list["student"] = student
    list["student"].Name = "LDB"

    fmt.Println(list["student"])
}
```

结果

编译失败， `./test7.go:18:23: cannot assign to struct field list["student"].Name in map`

分析

`map[string]Student` 的value是一个Student结构值，所以当 `list["student"] = student` ,是一个值拷贝过程。而 `list["student"]` 则是一个值引用。那么值引用的特点是 `只读` 。所以对 `list["student"].Name = "LDB"` 的修改是不允许的。

方法一：

```
package main

import "fmt"

type Student struct {
    Name string
}

var list map[string]Student

func main() {

    list = make(map[string]Student)
```

```
    student := Student{"Aceld"}

    list["student"] = student
    //list["student"].Name = "LDB"

    /*
        方法1:
    */
    tmpStudent := list["student"]
    tmpStudent.Name = "LDB"
    list["student"] = tmpStudent

    fmt.Println(list["student"])
}
```

其中

```
    /*
        方法1:
    */
    tmpStudent := list["student"]
    tmpStudent.Name = "LDB"
    list["student"] = tmpStudent
```

是先做一次值拷贝，做出一个 `tmpStudent副本` ,然后修改该副本，然后再次发生一次值拷贝复制回去， `list["student"] = tmpStudent` ,但是这种会在整体过程中发生2次结构体值拷贝，性能很差。

方法二：

```
package main

import "fmt"

type Student struct {
    Name string
}

var list map[string]*Student

func main() {

    list = make(map[string]*Student)

    student := Student{"Aceld"}

    list["student"] = &student
    list["student"].Name = "LDB"

    fmt.Println(list["student"])
}
```

我们将map的类型的value由Student值，改成Student指针。

```
var list map[string]*Student
```

这样，我们实际上每次修改的都是指针所指向的Student空间，指针本身是常指针，不能修改， `只读` 属性，但是指向的Student是可以随便修改的，而且这里并不需要值拷贝。只是一个指针的赋值。

(2) map的遍历赋值

> 以下代码有什么问题，说明原因

> test8.go

```
package main

import (
    "fmt"
)

type student struct {
    Name string
    Age  int
}

func main() {
    //定义map
    m := make(map[string]*student)

    //定义student数组
    stus := []student{
        {Name: "zhou", Age: 24},
        {Name: "li", Age: 23},
        {Name: "wang", Age: 22},
    }

    //将数组依次添加到map中
    for _, stu := range stus {
        m[stu.Name] = &stu
    }

    //打印map
    for k,v := range m {
        fmt.Println(k ,"=>", v.Name)
    }
}
```

结果

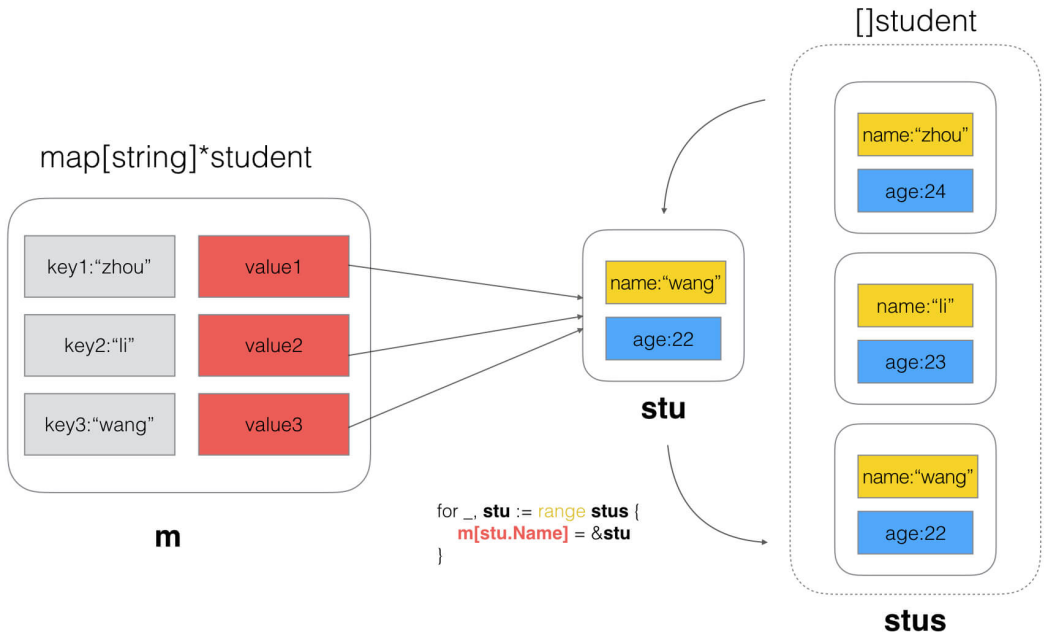
遍历结果出现错误，输出结果为

```
zhou => wang
li => wang
wang => wang
```

map中的3个key均指向数组中最后一个结构体。

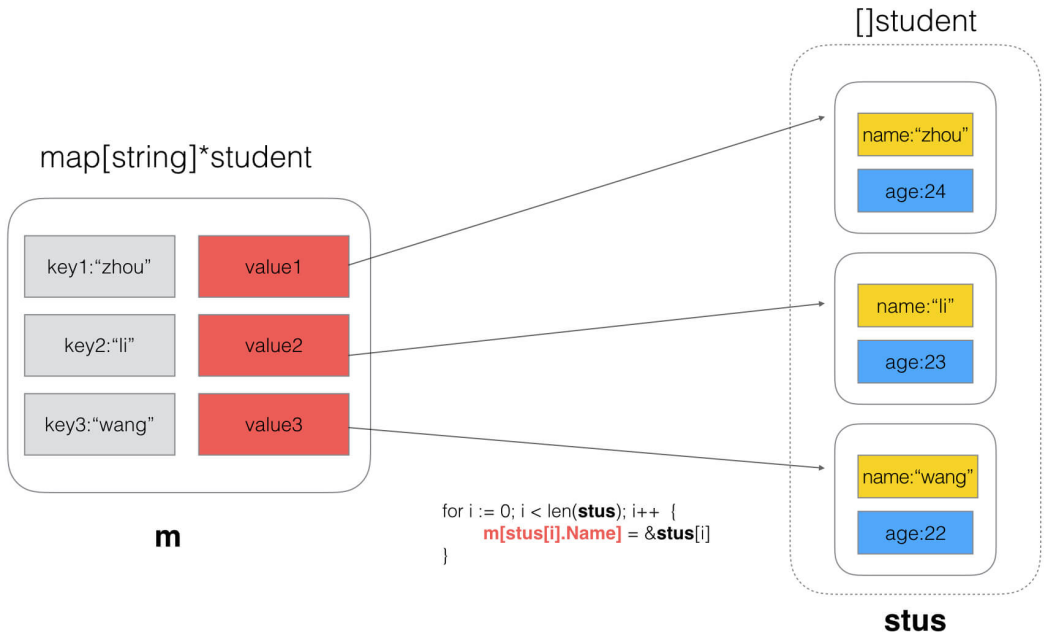
分析

foreach中，stu是结构体的一个拷贝副本，所以 `m[stu.Name]=&stu` 实际上一致指向同一个指针， 最终该指针的值为遍历的最后一个 `struct的值拷贝` 。



正确写法

```
package main

import (
    "fmt"
)

type student struct {
    Name string
    Age  int
}

func main() {
    //定义map
    m := make(map[string]*student)

    //定义student数组
    stus := []student{
        {Name: "zhou", Age: 24},
        {Name: "li", Age: 23},
        {Name: "wang", Age: 22},
    }

    // 遍历结构体数组，依次赋值给map
    for i := 0; i < len(stus); i++  {
        m[stus[i].Name] = &stus[i]
    }

    //打印map
    for k,v := range m {
        fmt.Println(k ,"=>", v.Name)
    }
}
```



运行结果

```
zhou => zhou
li => li
wang => wang
```

interface

(1) interface的赋值问题

> 以下代码能编译过去吗？为什么？

> test12.go

```
package main

import (
    "fmt"
)

type People interface {
    Speak(string) string
}

type Stduent struct{}

func (stu *Stduent) Speak(think string) (talk string) {
    if think == "love" {
        talk = "You are a good boy"
    } else {
        talk = "hi"
    }
    return
}

func main() {
    var peo People = Stduent{}
    think := "love"
    fmt.Println(peo.Speak(think))
}
```

继承与多态的特点

在golang中对多态的特点体现从语法上并不是很明显。

我们知道发生多态的几个要素：

1、有interface接口，并且有接口定义的方法。

2、有子类去重写interface的接口。

3、有父类指针指向子类的具体对象

那么，满足上述3个条件，就可以产生多态效果，就是，父类指针可以调用子类的具体方法。

所以上述代码报错的地方在 `var peo People = Stduent{}` 这条语句， `Student{}` 已经重写了父类 `People{}` 中的 `Speak(string) string` 方法，那么只需要用父类指针指向子类对象即可。

所以应该改成 `var peo People = &Student{}` 即可编译通过。（People为interface类型，就是指针类型）

(2) interface的内部构造(非空接口iface情况)

> 以下代码打印出来什么内容，说出为什么。

> test14.go

```
package main

import (
    "fmt"
)

type People interface {
    Show()
}

type Student struct{}

func (stu *Student) Show() {

}

func live() People {
    var stu *Student
    return stu
}

func main() {
    if live() == nil {
        fmt.Println("AAAAAAA")
    } else {
        fmt.Println("BBBBBBB")
    }
}
```

结果

```
BBBBBBB
```

分析：

我们需要了解 `interface` 的内部结构，才能理解这个题目的含义。

interface在使用的过程中，共有两种表现形式

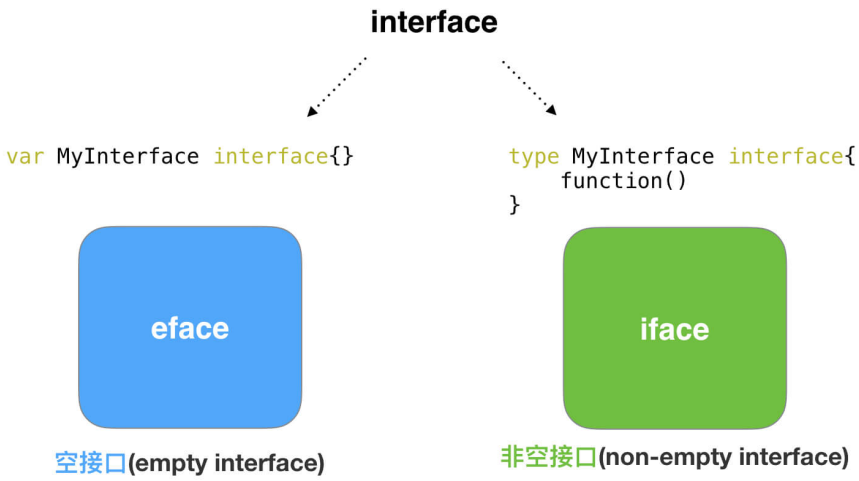
一种为**空接口(empty interface)**，定义如下：

```
var MyInterface interface{}
```

另一种为**非空接口(non-empty interface)**，定义如下：

```
type MyInterface interface {
        function()
}
```

这两种interface类型分别用两种 `struct` 表示，空接口为 `eface` ，非空接口为 `iface` .



刘丹冰Aceld

空接口eface

空接口eface结构，由两个属性构成，一个是类型信息_type，一个是数据信息。其数据结构声明如下：

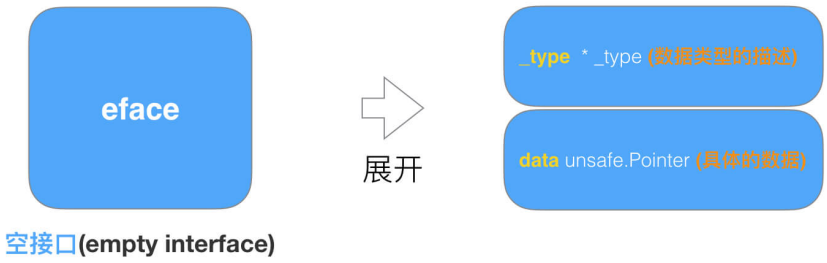
```
type eface struct {      //空接口
    _type *_type         //类型信息
    data  unsafe.Pointer //指向数据的指针(go语言中特殊的指针类型unsafe.Pointer类似于c语言中的void*)
}
```

_type属性：是GO语言中所有类型的公共描述，Go语言几乎所有的数据结构都可以抽象成 _type，是所有类型的公共描述，**type负责决定data应该如何解释和操作**，type的结构代码如下：

```
type _type struct {
    size       uintptr  //类型大小
    ptrdata    uintptr  //前缀持有所有指针的内存大小
    hash       uint32   //数据hash值
    tflag      tflag
    align      uint8    //对齐
    fieldalign uint8    //嵌入结构体时的对齐
    kind       uint8    //kind 有些枚举值kind等于0是无效的
    alg        *typeAlg //函数指针数组，类型实现的所有方法
    gcdata    *byte
    str       nameOff
    ptrToThis typeOff
}
```

data属性: 表示指向具体的实例数据的指针，他是一个 `unsafe.Pointer` 类型，相当于一个C的万能指针 `void*` 。

```
var MyInterface interface{}
```



刘丹冰Aceld

非空接口iface

iface 表示 non-empty interface 的数据结构，非空接口初始化的过程就是初始化一个iface类型的结构，其中 `data` 的作用同 `eface` 的相同，这里不再多加描述。

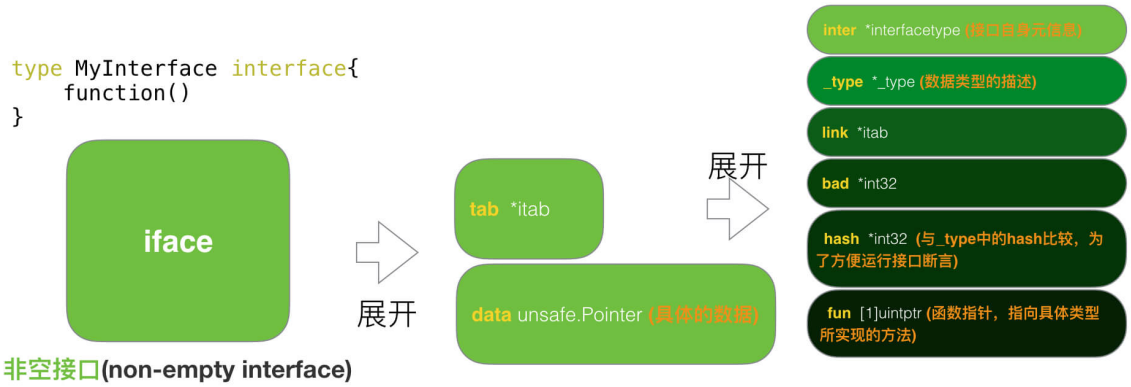
```
type iface struct {
  tab  *itab
  data unsafe.Pointer
}
```

iface结构中最重要的是itab结构（结构如下），每一个 `itab` 都占 32 字节的空间。itab可以理解为 `pair<interface type, concrete type>` 。itab里面包含了interface的一些关键信息，比如method的具体实现。

```
type itab struct {
  inter  *interfacetype   // 接口自身的元信息
  _type  *_type           // 具体类型的元信息
  link   *itab
  bad    int32
  hash   int32            // _type里也有一个同样的hash，此处多放一个是为了方便运行接口断言
  fun    [1]uintptr       // 函数指针，指向具体类型所实现的方法
}
```

其中值得注意的字段，个人理解如下：

1. `interface type` 包含了一些关于interface本身的信息，比如 `package path` ，包含的 `method` 。这里的interfacetype是定义interface的一种抽象表示。
2. `type` 表示具体化的类型，与eface的 *type类型相同*。
3. `hash` 字段其实是对 `_type.hash` 的拷贝，它会在interface的实例化时，用于快速判断目标类型和接口中的类型是否一致。另，Go的interface的Duck-typing机制也是依赖这个字段来实现。
4. `fun` 字段其实是一个动态大小的数组，虽然声明时是固定大小为1，但在使用时会直接通过fun指针获取其中的数据，并且不会检查数组的边界，所以该数组中保存的元素数量是不确定的。



刘丹冰Aceld

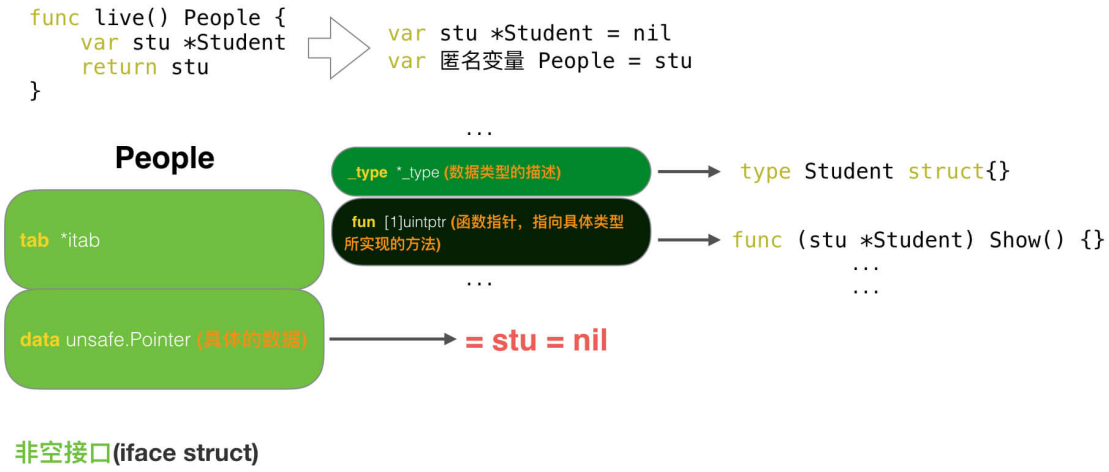
所以，People拥有一个Show方法的，属于非空接口，People的内部定义应该是一个 `iface` 结构体

```
type People interface {
    Show()
}
```

刘丹冰Aceld

```
func live() People {
    var stu *Student
    return stu
}
```

stu是一个指向nil的空指针，但是最后 `return stu` 会触发 `匿名变量 People = stu` 值拷贝动作，所以最后 `live()` 放回给上层的是一个 `People insterface{}` 类型，也就是一个 `iface struct{}` 类型。 stu为nil，只是 `iface` 中的data 为nil而已。 但是 `iface struct{}` 本身并不为nil.



刘丹冰Aceld

所以如下判断的结果为 `BBBBBBB` ：

```
func main() {
    if live() == nil {
        fmt.Println("AAAAAAA")
    } else {
        fmt.Println("BBBBBBB")
    }
}
```

(3) interface内部构造(空接口eface情况)

> 下面代码结果为什么？

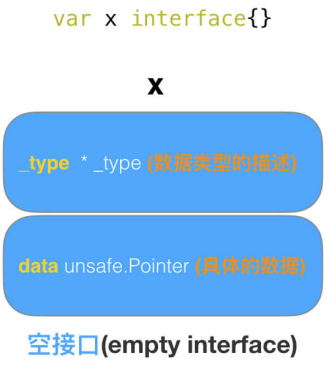
```
func Foo(x interface{}) {
    if x == nil {
        fmt.Println("empty interface")
        return
    }
    fmt.Println("non-empty interface")
}
func main() {
    var p *int = nil
    Foo(p)
}
```

结果

```
non-empty interface
```

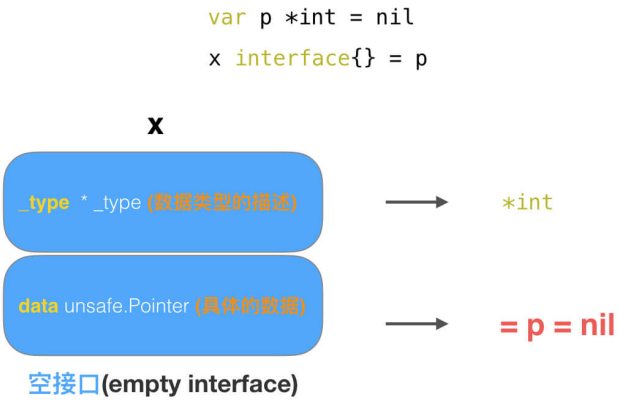
分析

不难看出， `Foo()` 的形参 `x interface{}` 是一个空接口类型 `eface struct{}` 。



刘丹冰Aceld

在执行 `Foo(p)` 的时候，触发 `x interface{} = p` 语句，所以此时 x结构如下。



刘丹冰Aceld

所以 x 结构体本身不为nil，而是data指针指向的p为nil。

---

(4) inteface{}与*interface{}

> ABCD中哪一行存在错误？

> test15.go

```
type S struct {
}

func f(x interface{}) {
}

func g(x *interface{}) {
}

func main() {
    s := S{}
    p := &s
    f(s) //A
    g(s) //B
    f(p) //C
    g(p) //D
}
```

结果

```
B、D两行错误
B错误为: cannot use s (type S) as type *interface {} in argument to g:
    *interface {} is pointer to interface, not interface

D错误为: cannot use p (type *S) as type *interface {} in argument to g:
    *interface {} is pointer to interface, not interface
```

看到这道题需要第一时间想到的是Golang是强类型语言，interface是所有golang类型的父类 函数中 `func f(x interface{})` 的 `interface{}` 可以支持传入golang的任何类型，包括指针，但是函数 `func g(x *interface{})` 只能接受 `*interface{}`

channel

(1)Channel读写特性(15字口诀)

首先，我们先复习一下Channel都有哪些特性？

- 给一个 nil channel 发送数据，造成永远阻塞
- 从一个 nil channel 接收数据，造成永远阻塞
- 给一个已经关闭的 channel 发送数据，引起 panic
- 从一个已经关闭的 channel 接收数据，如果缓冲区中为空，则返回一个零值
- 无缓冲的channel是同步的，而有缓冲的channel是非同步的

以上5个特性是死东西，也可以通过口诀来记忆：“空读写阻塞，写关闭异常，读关闭空零”。

> 执行下面的代码发生什么？

> test17.go

```
package main

import (
    "fmt"
    "time"
)

func main() {
    ch := make(chan int, 1000)
    go func() {
        for i := 0; i < 10; i++ {
            ch <- i
        }
    }()
    go func() {
        for {
            a, ok := <-ch
            if !ok {
                fmt.Println("close")
                return
            }
            fmt.Println("a: ", a)
        }
    }()
    close(ch)
    fmt.Println("ok")
    time.Sleep(time.Second * 100)
}
```

15字口诀：“空读写阻塞，写关闭异常，读关闭空零”，往已经关闭的channel写入数据会panic的。因为main在开辟完两个goroutine之后，立刻关闭了ch， 结果：

```
panic: send on closed channel
```

WaitGroup

(1) WaitGroup与goroutine的竞速问题

> 编译并运行如下代码会发生什么？

> test18.go

```
package main

import (
    "sync"
    //"time"
)

const N = 10

var wg = &sync.WaitGroup{}

func main() {

    for i := 0; i < N; i++ {
        go func(i int) {
            wg.Add(1)
            println(i)
            defer wg.Done()
        }(i)
    }
    wg.Wait()

}
```

结果

```
结果不唯一，代码存在风险，所有go未必都能执行到
```

这是使用WaitGroup经常犯下的错误！请各位同学多次运行就会发现输出都会不同甚至又出现报错的问题。 这是因为 `go` 执行太快了，导致 `wg.Add(1)` 还没有执行main函数就执行完毕了。 改为如下试试

```
package main

import (
    "sync"
)

const N = 10

var wg = &sync.WaitGroup{}

func main() {

    for i:= 0; i< N; i++ {
        wg.Add(1)
        go func(i int) {
```

```
            println(i)
            defer wg.Done()
        }(i)
    }

    wg.Wait()
}
```

Golang编程设计与通用之路

流？I/O操作？阻塞？epoll?

一、流？I/O操作? 阻塞？

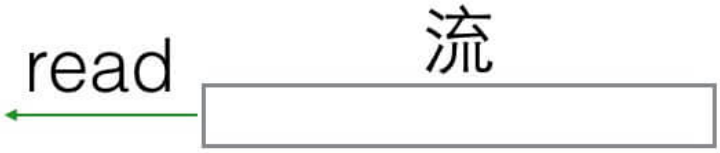
(1) 流

- 可以进行I/O操作的内核对象
- 文件、管道、套接字.....
- 流的入口：文件描述符(fd)

(2) I/O操作

所有对流的读写操作，我们都可以称之为IO操作。

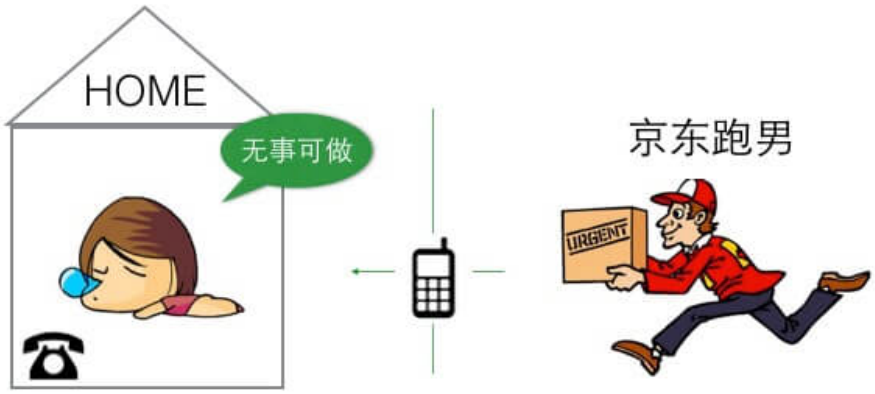
当一个流中， 在没有数据read的时候，或者说在流中已经写满了数据，再write，我们的IO操作就会出现一种现象，就是阻塞现象，如下图。



---

(3) 阻塞

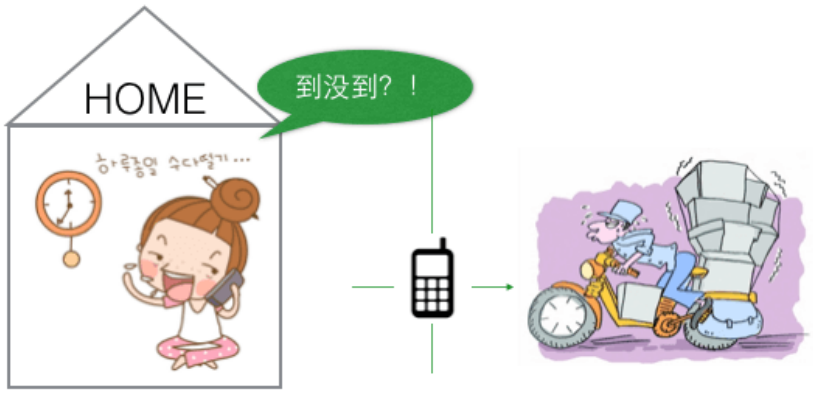
阻塞?



阻塞等待快递

阻塞场景: 你有一份快递，家里有个座机，快递到了主动给你打电话，期间你可以休息。

非阻塞，忙轮询



每隔一分钟催一次

非阻塞，忙轮询场景: 你性子比较急躁， 每分钟就要打电话询问快递小哥一次， 到底有没有到，快递员接你电话要停止运输，这样很耽误快递小哥的运输速度。

- 阻塞等待

空出大脑可以安心睡觉, 不影响快递员工作（不占用CPU宝贵的时间片）。

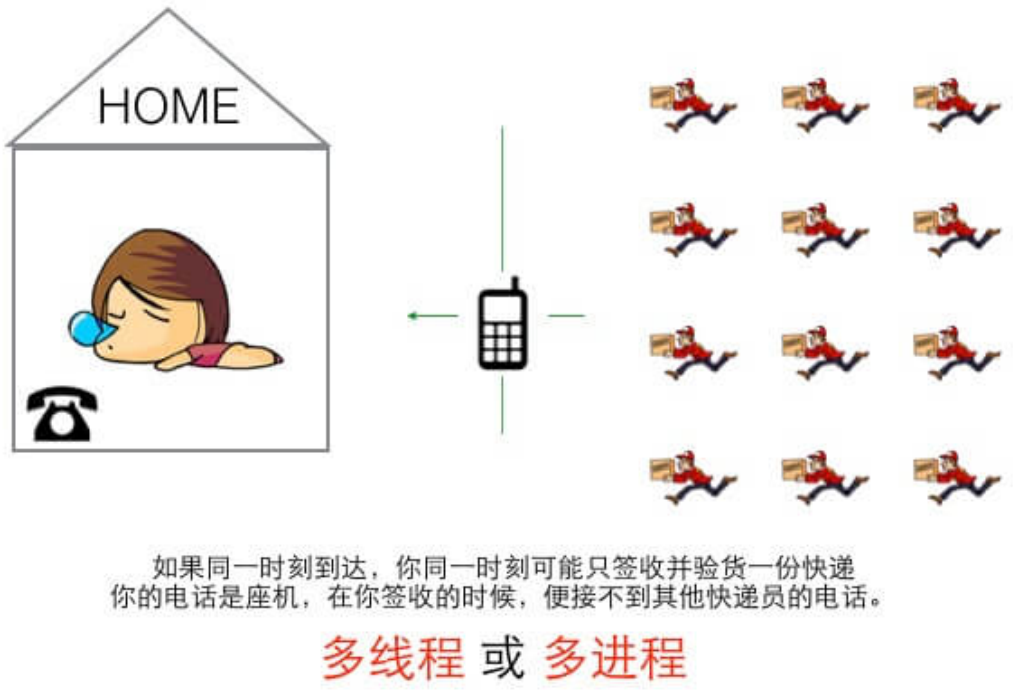
- 非阻塞，忙轮询

浪费时间，浪费电话费，占用快递员时间（占用CPU，系统资源）。

很明显，阻塞等待这种方式，对于通信上是有明显优势的， 那么它有哪些弊端呢？

二、解决阻塞死等待的办法

阻塞死等待的缺点

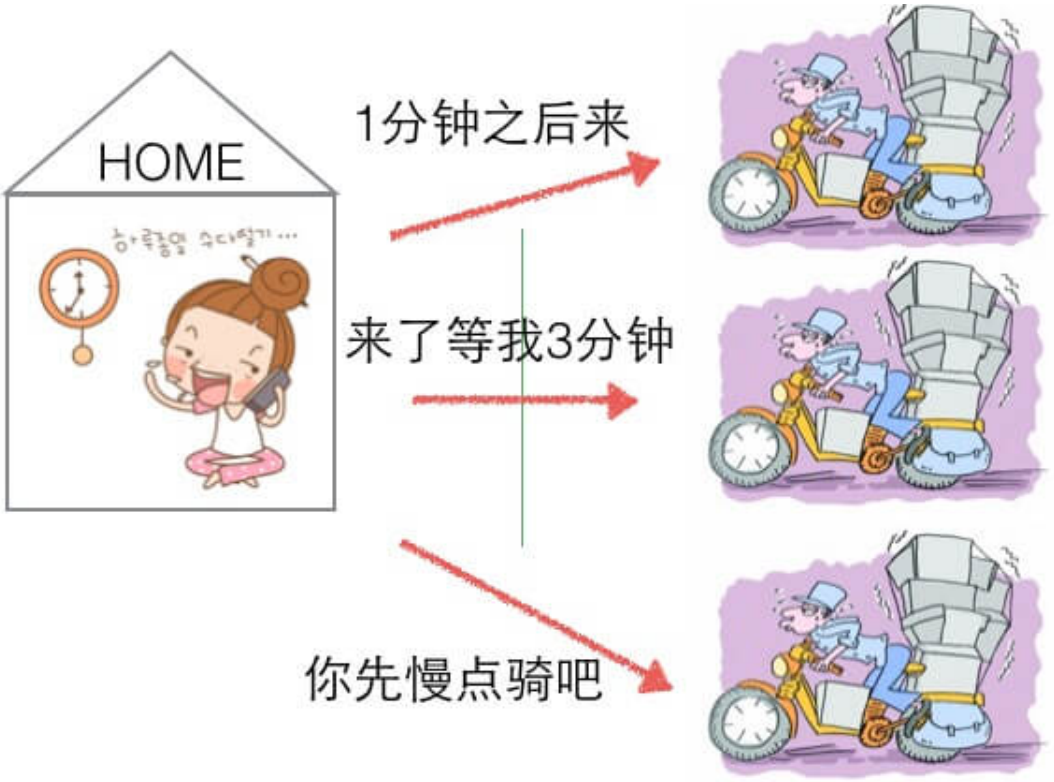


也就是同一时刻，你只能被动的处理一个快递员的签收业务，其他快递员打电话打不进来，只能干瞪眼等待。那么解决这个问题，家里多买N个座机， 但是依然是你一个人接，也处理不过来，需要用影分身术创建都个自己来接电话(采用多线程或者多进程）来处理。

这种方式就是没有多路IO复用的情况的解决方案， 但是在单线程计算机时代(无法影分身)，这简直是灾难。

那么如果我们不借助影分身的方式(多线程/多进程)，该如何解决阻塞死等待的方法呢？

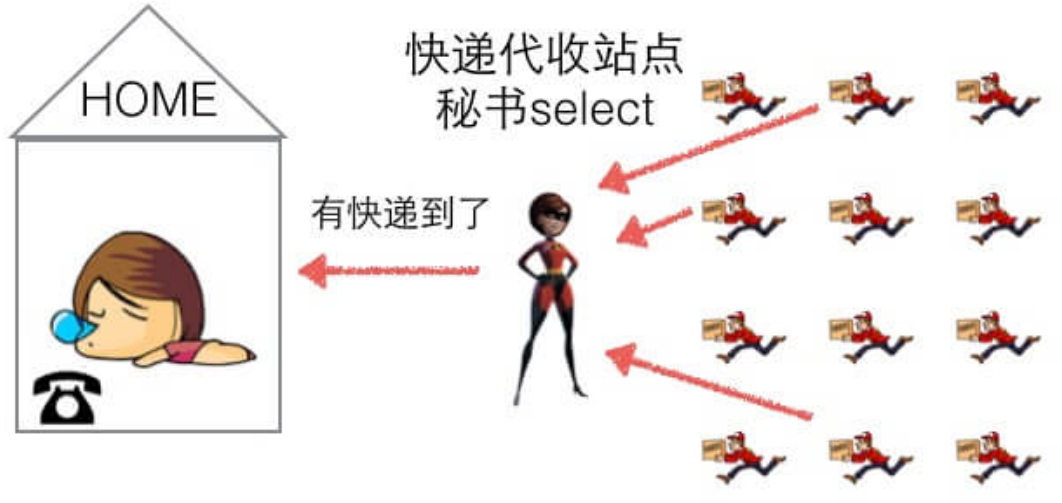
办法一：非阻塞、忙轮询




```
while true {
    for i in 流[] {
        if i has 数据 {
            读 或者 其他处理
        }
    }
}
```

非阻塞忙轮询的方式，可以让用户分别与每个快递员取得联系，宏观上来看，是同时可以与多个快递员沟通(并发效果)、 但是快递员在于用户沟通时耽误前进的速度(浪费CPU)。

办法二：select

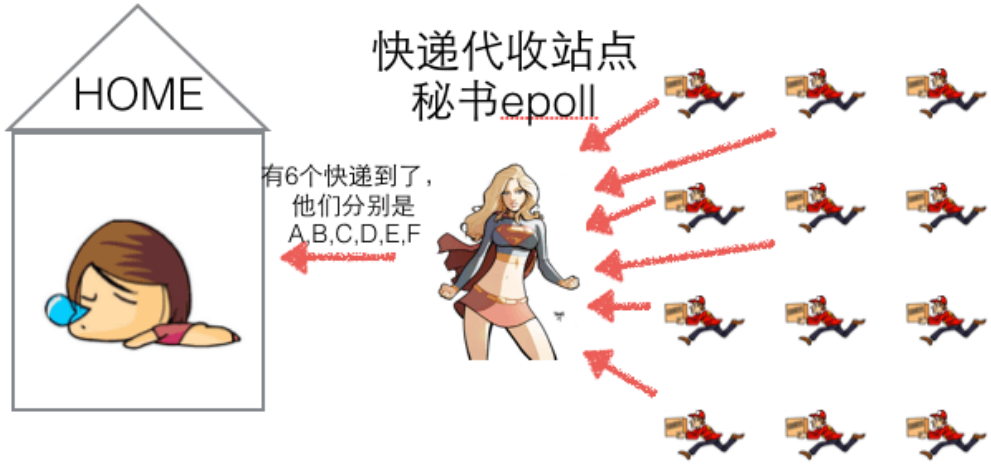


我们可以开设一个代收网点，让快递员全部送到代收点。这个网店管理员叫select。这样我们就可以在家休息了，麻烦的事交给select就好了。当有快递的时候，select负责给我们打电话，期间在家休息睡觉就好了。

但select 代收员比较懒，她记不住快递员的单号，还有快递货物的数量。她只会告诉你快递到了，但是是谁到的，你需要挨个快递员问一遍。

```
while true {
    select(流[]); //阻塞

  //有消息抵达
    for i in 流[] {
        if i has 数据 {
            读 或者 其他处理
        }
    }
}
```

办法三：epoll



epoll的服务态度要比select好很多，在通知我们的时候，不仅告诉我们有几个快递到了，还分别告诉我们是谁谁谁。我们只需要按照epoll给的答复，来询问快递员取快递即可。

```
while true {
    可处理的流[] = epoll_wait(epoll_fd); //阻塞

  //有消息抵达，全部放在 “可处理的流[]”中
    for i in 可处理的流[] {
        读 或者 其他处理
    }
}
```

三、epoll？

- 与select，poll一样，对I/O多路复用的技术
- 只关心“活跃”的链接，无需遍历全部描述符集合
- 能够处理大量的链接请求(系统可以打开的文件数目)

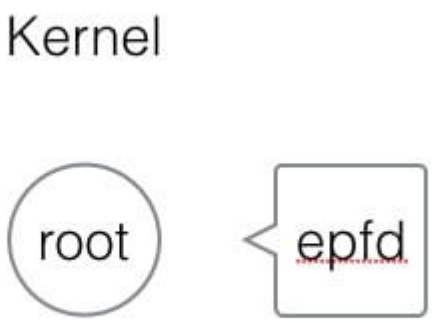
四、epoll的API

(1) 创建EPOLL

```
/**
 * @param size 告诉内核监听的数目
 *
 * @returns 返回一个epoll句柄（即一个文件描述符）
 */
int epoll_create(int size);
```

使用

```
int epfd = epoll_create(1000);
```



创建一个epoll句柄，实际上是在内核空间，建立一个root根节点，这个根节点的关系与epfd相对应。

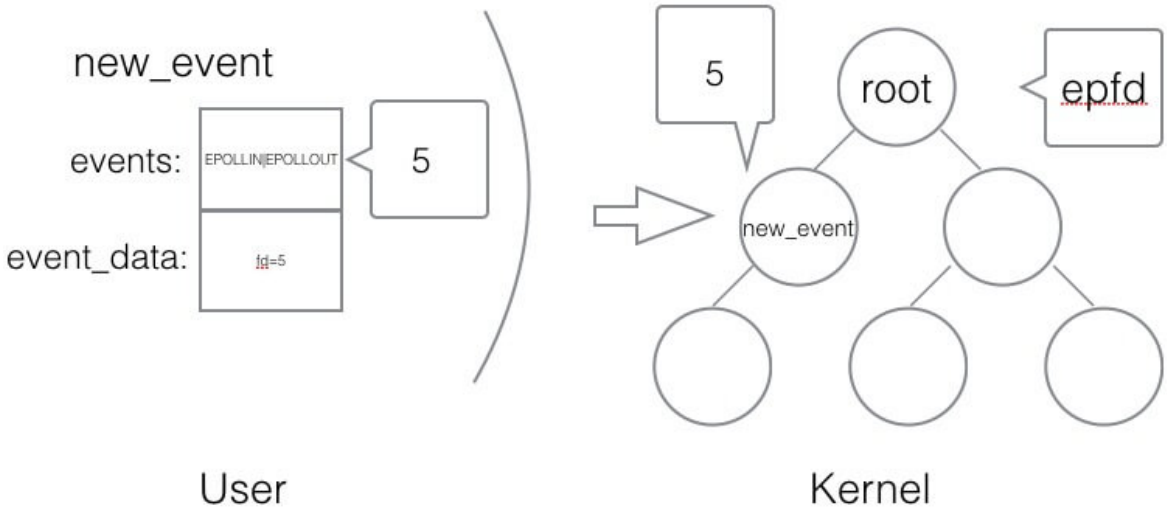
(2) 控制EPOLL

```
/**
* @param epfd 用epoll_create所创建的epoll句柄
* @param op 表示对epoll监控描述符控制的动作
*
* EPOLL_CTL_ADD(注册新的fd到epfd)
* EPOLL_CTL_MOD(修改已经注册的fd的监听事件)
* EPOLL_CTL_DEL(epfd删除一个fd)
*
* @param fd 需要监听的文件描述符
* @param event 告诉内核需要监听的事件
*
* @returns 成功返回0，失败返回-1, errno查看错误信息
*/
int epoll_ctl(int epfd, int op, int fd,
struct epoll_event *event);


struct epoll_event {
    __uint32_t events; /* epoll 事件 */
    epoll_data_t data; /* 用户传递的数据 */
}

/*
 * events : {EPOLLIN, EPOLLOUT, EPOLLPRI,
                        EPOLLHUP, EPOLLET, EPOLLONESHOT}
 */
typedef union epoll_data {
    void *ptr;
    int fd;
    uint32_t u32;
    uint64_t u64;
} epoll_data_t;
```

使用

```
struct epoll_event new_event;

new_event.events = EPOLLIN | EPOLLOUT;
new_event.data.fd = 5;

epoll_ctl(epfd, EPOLL_CTL_ADD, 5, &new_event);
```

创建一个用户态的事件，绑定到某个fd上，然后添加到内核中的epoll红黑树中。



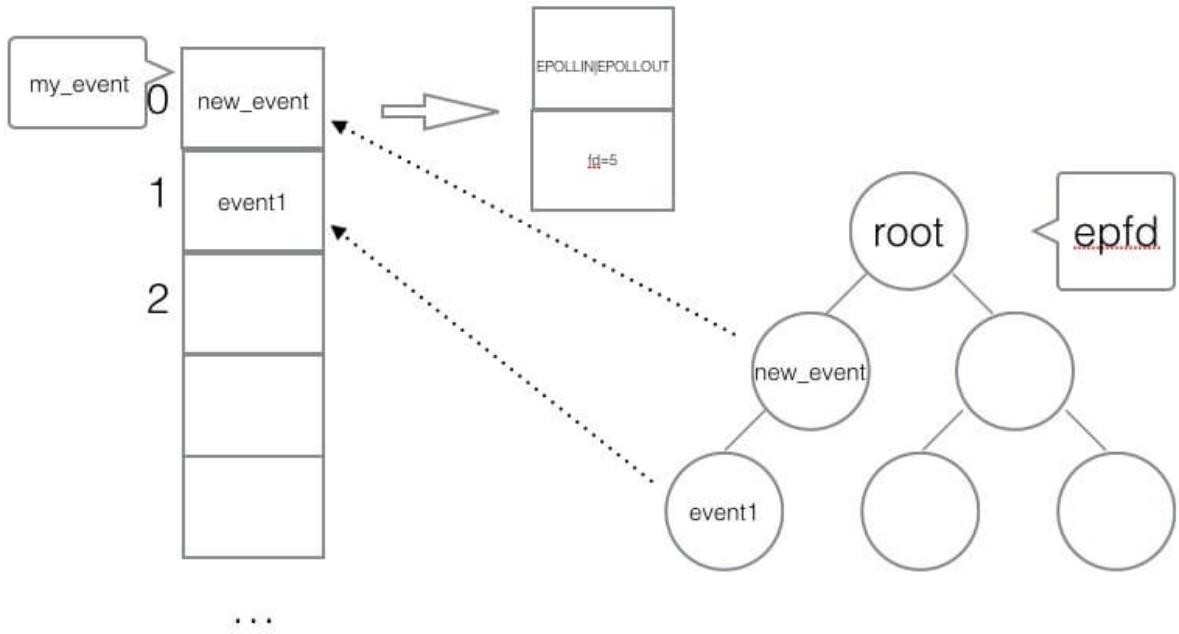
(3) 等待EPOLL

```
/**
*
* @param epfd 用epoll_create所创建的epoll句柄
* @param event 从内核得到的事件集合
* @param maxevents 告知内核这个events有多大,
* 注意: 值 不能大于创建epoll_create()时的size.
* @param timeout 超时时间
* -1: 永久阻塞
* 0: 立即返回，非阻塞
* >0: 指定微秒
*
* @returns 成功: 有多少文件描述符就绪,时间到时返回0
* 失败: -1, errno 查看错误
*/
int epoll_wait(int epfd, struct epoll_event *event,
                            int maxevents, int timeout);
```

使用

```
struct epoll_event my_event[1000];

int event_cnt = epoll_wait(epfd, my_event, 1000, -1);
```

`epoll_wait` 是一个阻塞的状态，如果内核检测到IO的读写响应，会抛给上层的epoll_wait, 返回给用户态一个已经触发的事件队列，同时阻塞返回。开发者可以从队列中取出事件来处理，其中事件里就有绑定的对应fd是哪个(之前添加epoll事件的时候已经绑定)。

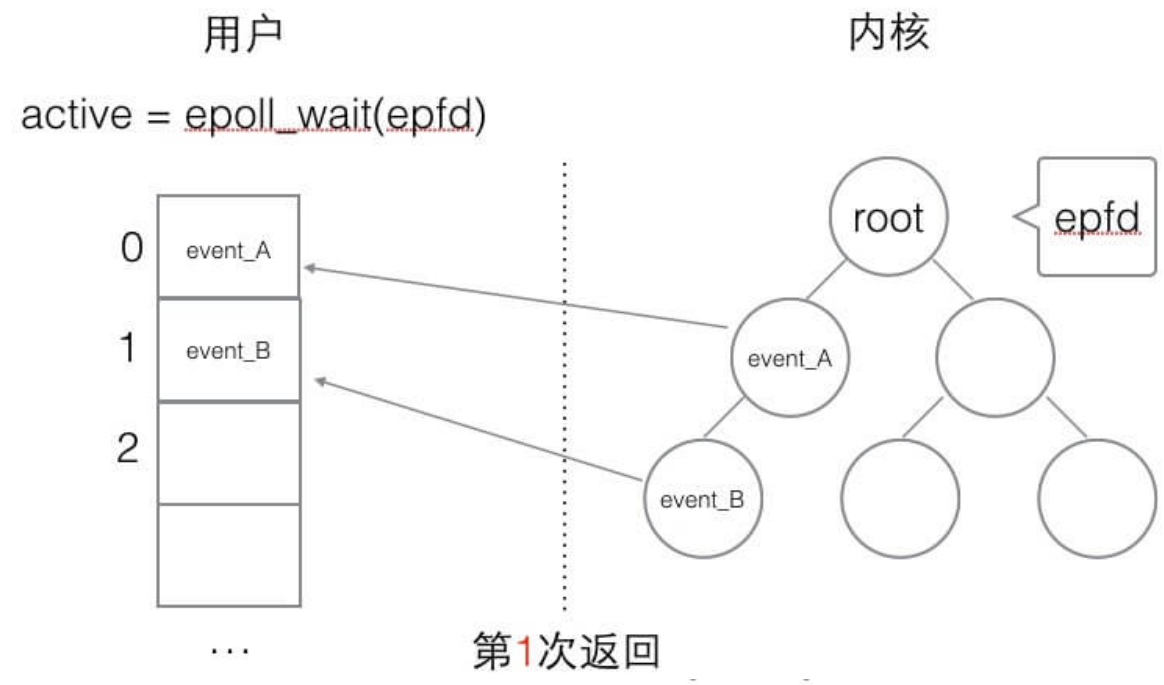


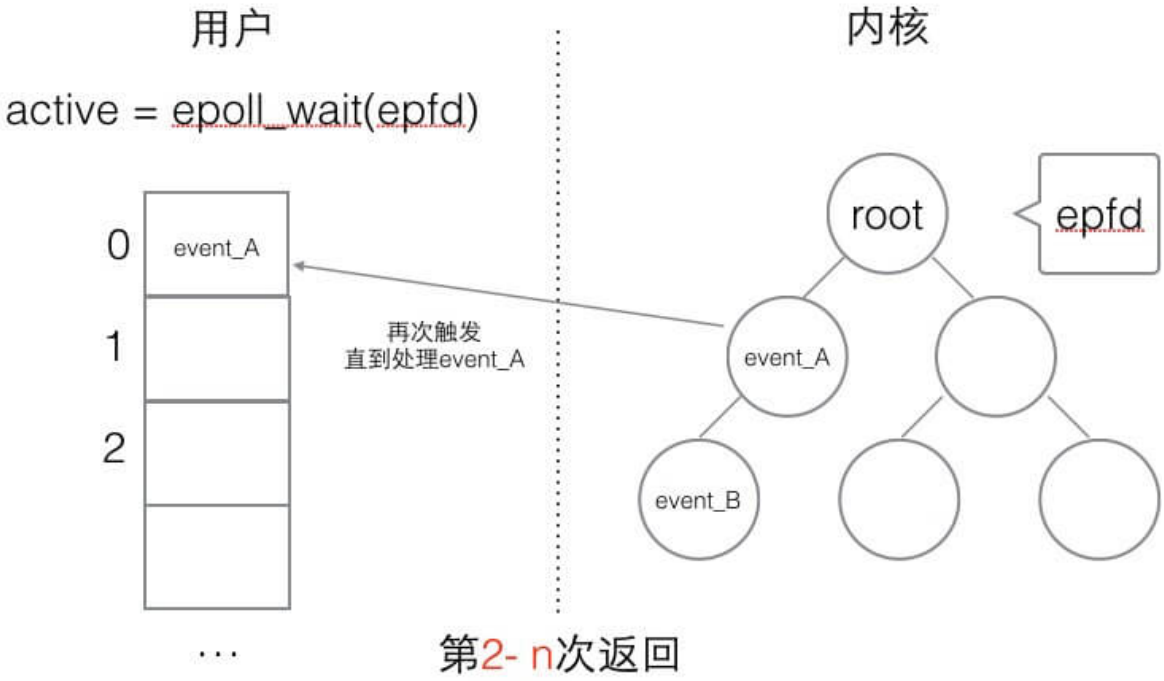
(4) 使用epoll编程主流程骨架

```
int epfd = epoll_crete(1000);

//将 listen_fd 添加进 epoll 中
epoll_ctl(epfd, EPOLL_CTL_ADD, listen_fd,&listen_event);

while (1) {
    //阻塞等待 epoll 中 的fd 触发
    int active_cnt = epoll_wait(epfd, events, 1000, -1);

    for (i = 0 ; i < active_cnt; i++) {
        if (evnets[i].data.fd == listen_fd) {
            //accept. 并且将新accept 的fd 加进epoll中.
        }
        else if (events[i].events & EPOLLIN) {
            //对此fd 进行读操作
        }
        else if (events[i].events & EPOLLOUT) {
            //对此fd 进行写操作
        }
    }
}
```

五、epoll的触发模式

(1) 水平触发

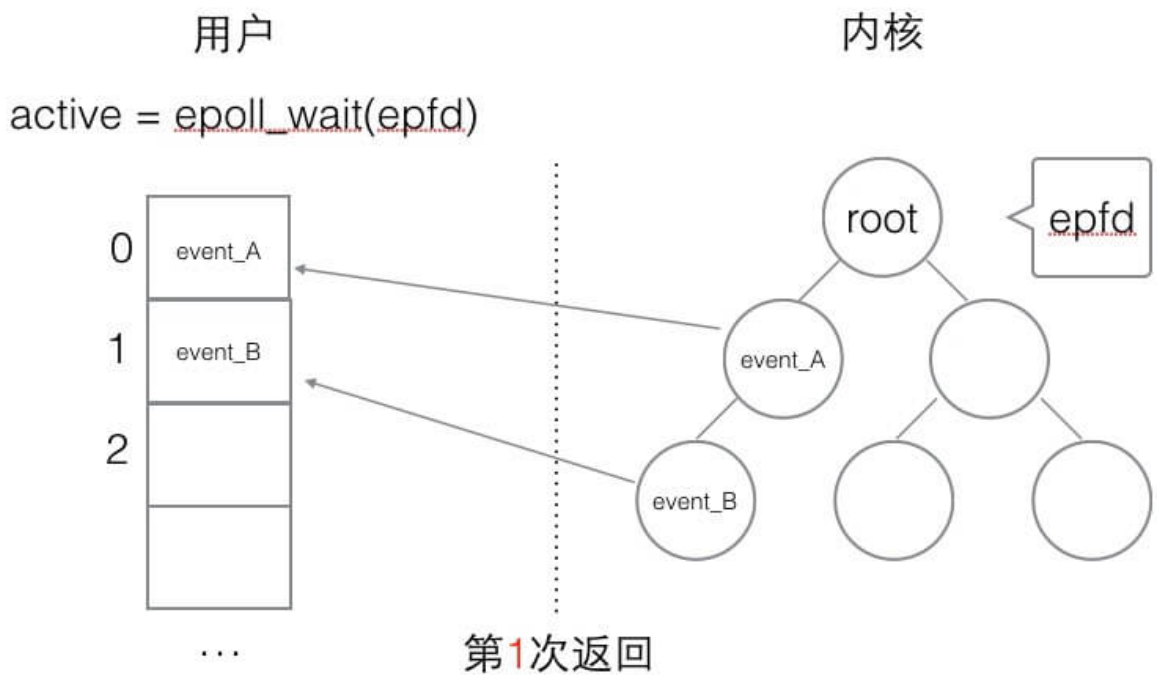


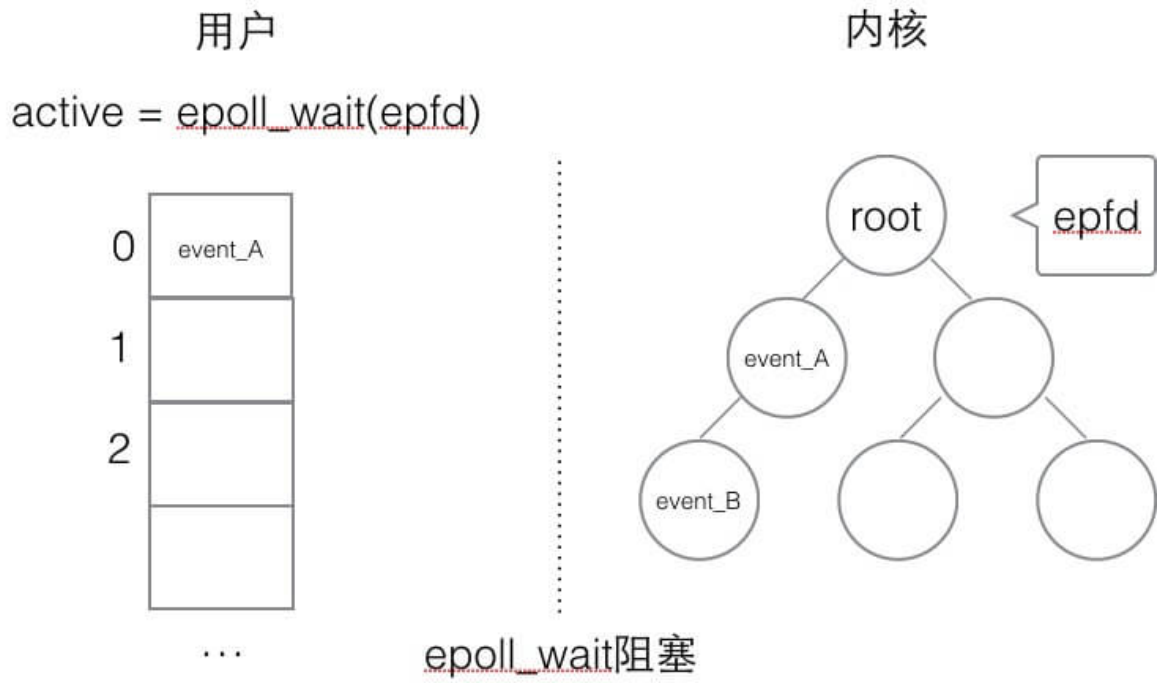


水平触发的主要特点是，如果用户在监听 `epoll` 事件，当内核有事件的时候，会拷贝给用户态事件，但是**如果用户只处理了一次，那么剩下没有处理的会在下一次epoll_wait再次返回该事件**。

这样如果用户永远不处理这个事件，就导致每次都会有该事件从内核到用户的拷贝，耗费性能，但是水平触发相对安全，最起码事件不会丢掉，除非用户处理完毕。

(2) 边缘触发





边缘触发，相对跟水平触发相反，当内核有事件到达， 只会通知用户一次，至于用户处理还是不处理，以后将不会再通知。这样减少了拷贝过程，增加了性能，但是相对来说，如果用户马虎忘记处理，将会产生事件丢的情况。

六、简单的epoll服务器(C语言)

(1) 服务端

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
```

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>

#include <sys/epoll.h>

#define SERVER_PORT (7778)
#define EPOLL_MAX_NUM (2048)
#define BUFFER_MAX_LEN (4096)

char buffer[BUFFER_MAX_LEN];

void str_toupper(char *str)
{
    int i;
    for (i = 0; i < strlen(str); i ++) {
        str[i] = toupper(str[i]);
    }
}

int main(int argc, char **argv)
{
    int listen_fd = 0;
    int client_fd = 0;
    struct sockaddr_in server_addr;
    struct sockaddr_in client_addr;
    socklen_t client_len;

    int epfd = 0;
    struct epoll_event event, *my_events;

    / socket
        listen_fd = socket(AF_INET, SOCK_STREAM, 0);

    // bind
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    server_addr.sin_port = htons(SERVER_PORT);
    bind(listen_fd, (struct sockaddr*)&server_addr, sizeof(server_addr));

    // listen
    listen(listen_fd, 10);

    // epoll create
    epfd = epoll_create(EPOLL_MAX_NUM);
    if (epfd < 0) {
        perror("epoll create");
        goto END;
    }

    // listen_fd -> epoll
    event.events = EPOLLIN;
    event.data.fd = listen_fd;
    if (epoll_ctl(epfd, EPOLL_CTL_ADD, listen_fd, &event) < 0) {
        perror("epoll ctl add listen_fd ");
        goto END;
    }

    my_events = malloc(sizeof(struct epoll_event) * EPOLL_MAX_NUM);
```

```
    while (1) {
        // epoll wait
        int active_fds_cnt = epoll_wait(epfd, my_events, EPOLL_MAX_NUM, -1);
        int i = 0;
        for (i = 0; i < active_fds_cnt; i++) {
            // if fd == listen_fd
            if (my_events[i].data.fd == listen_fd) {
                //accept
                client_fd = accept(listen_fd, (struct sockaddr*)&client_addr, &client_len);
                if (client_fd < 0) {
                    perror("accept");
                    continue;
                }

                char ip[20];
                printf("new connection[%s:%d]\n", inet_ntop(AF_INET, &client_addr.sin_addr, ip, sizeof(ip)),
ntohs(client_addr.sin_port));

                event.events = EPOLLIN | EPOLLET;
                event.data.fd = client_fd;
                epoll_ctl(epfd, EPOLL_CTL_ADD, client_fd, &event);
            }
            else if (my_events[i].events & EPOLLIN) {
                printf("EPOLLIN\n");
                client_fd = my_events[i].data.fd;

                // do read

                buffer[0] = '\0';
                int n = read(client_fd, buffer, 5);
                if (n < 0) {
                    perror("read");
                    continue;
                }
                else if (n == 0) {
                    epoll_ctl(epfd, EPOLL_CTL_DEL, client_fd, &event);
                    close(client_fd);
                }
                else {
                    printf("[read]: %s\n", buffer);
                    buffer[n] = '\0';
#if 1
                    str_toupper(buffer);
                    write(client_fd, buffer, strlen(buffer));
                    printf("[write]: %s\n", buffer);
                    memset(buffer, 0, BUFFER_MAX_LEN);
#endif

                    /*
                       event.events = EPOLLOUT;
                       event.data.fd = client_fd;
                       epoll_ctl(epfd, EPOLL_CTL_MOD, client_fd, &event);
                       */
                }
            }
            else if (my_events[i].events & EPOLLOUT) {
                printf("EPOLLOUT\n");
                /*
                   client_fd = my_events[i].data.fd;
                   str_toupper(buffer);
                   write(client_fd, buffer, strlen(buffer));
                   printf("[write]: %s\n", buffer);
```

```
                    memset(buffer, 0, BUFFER_MAX_LEN);

                    event.events = EPOLLIN;
                    event.data.fd = client_fd;
                    epoll_ctl(epfd, EPOLL_CTL_MOD, client_fd, &event);
                    */
                }
            }
        }

END:
    close(epfd);
    close(listen_fd);
    return 0;
}
```

(2) 客户端

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>

#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <fcntl.h>

#define MAX_LINE (1024)
#define SERVER_PORT (7778)

void setnoblocking(int fd)
{
    int opts = 0;
    opts = fcntl(fd, F_GETFL);
    opts = opts | O_NONBLOCK;
    fcntl(fd, F_SETFL);
}

int main(int argc, char **argv)
{
    int sockfd;
    char recvline[MAX_LINE + 1] = {0};

    struct sockaddr_in server_addr;

    if (argc != 2) {
        fprintf(stderr, "usage ./client <SERVER_IP>\n");
        exit(0);
    }


    // 创建socket
    if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        fprintf(stderr, "socket error");
        exit(0);
    }
```

```
    // server addr 赋值
    bzero(&server_addr, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(SERVER_PORT);

    if (inet_pton(AF_INET, argv[1], &server_addr.sin_addr) <= 0) {
        fprintf(stderr, "inet_pton error for %s", argv[1]);
        exit(0);
    }


    // 链接服务端
    if (connect(sockfd, (struct sockaddr*) &server_addr, sizeof(server_addr)) < 0) {
        perror("connect");
        fprintf(stderr, "connect error\n");
        exit(0);
    }

    setnoblocking(sockfd);

    char input[100];
    int n = 0;
    int count = 0;



    // 不断的从标准输入字符串
    while (fgets(input, 100, stdin) != NULL)
    {
        printf("[send] %s\n", input);
        n = 0;
        // 把输入的字符串发送 到 服务器中去
        n = send(sockfd, input, strlen(input), 0);
        if (n < 0) {
            perror("send");
        }

        n = 0;
        count = 0;


        // 读取 服务器返回的数据
        while (1)
        {
            n = read(sockfd, recvline + count, MAX_LINE);
            if (n == MAX_LINE)
            {
                count += n;
                continue;
            }
            else if (n < 0){
                perror("recv");
                break;
            }
            else {
                count += n;
                recvline[count] = '\0';
                printf("[recv] %s\n", recvline);
                break;
            }
        }
    }
```

```
    return 0;
}
```

分布式从ACID、CAP、BASE的理论推进

2、分布式从ACID、CAP、BASE的理论推进

分布式实际上就是单一的本地一体解决方案，在硬件或者资源上不够业务需求，而采取的一种分散式多节点，可以扩容资源的一种解决思路。它研究如何把一个需要非常巨大的计算能力才能解决的问题分成许多小的部分，然后把这些部分分配给多个计算机进行处理，最后把这些计算结果综合起来得到最终的结果。

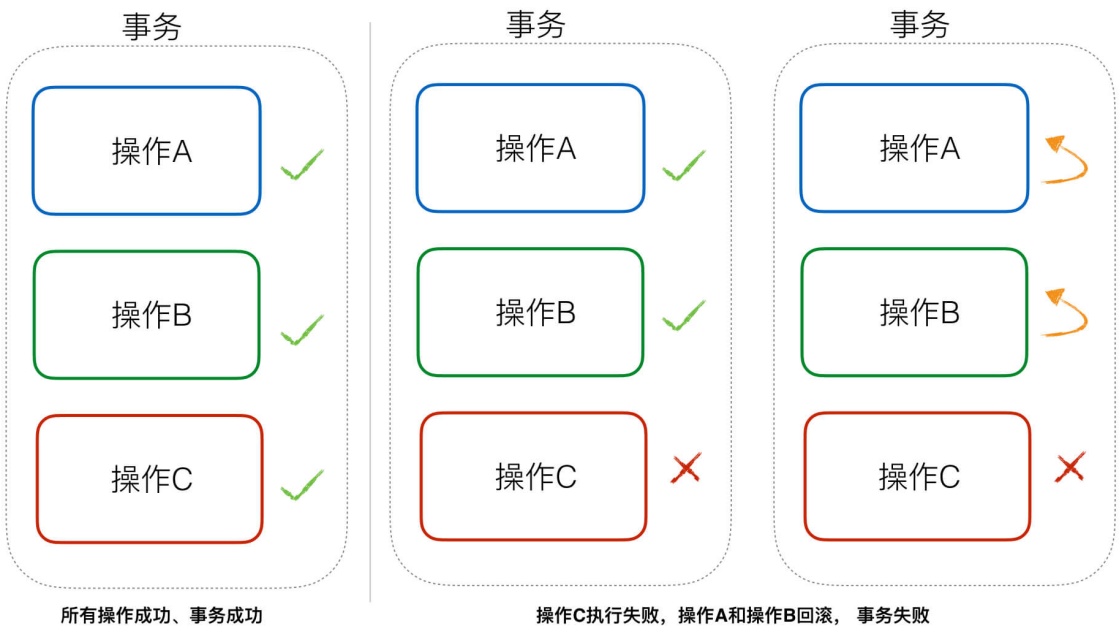
那么在了解分布式之前，我们应该从一体式的构造开始说明。

一、从本地事务到分布式理论

理解分布式之前，需要理解一个问题就是"事务"

> 事务提供一种机制将一个活动涉及的所有操作纳入到一个不可分割的执行单元，组成事务的所有操作只有在所有操作均能正常执行的情况下方能提交，只要其中任一操作执行失败，都将导致整个事务的回滚。

简单地说，事务提供一种"**要么什么都不做，要么做全套（All or Nothing）**"机制。



二、ACID理论

事务是基于数据进行操作，需要保证事务的数据通常存储在数据库中，所以介绍到事务，就不得不介绍数据库事务的 `ACID` 特性，指数据库事务正确执行的四个基本特性的缩写。包含：

- **原子性（Atomicity）**
- **一致性（Consistency）**
- **隔离性（Isolation）**
- **持久性（Durability）**

(1) 原子性（Atomicity）

整个事务中的所有操作，要么全部完成，要么全部不完成，不可能停滞在中间某个环节。

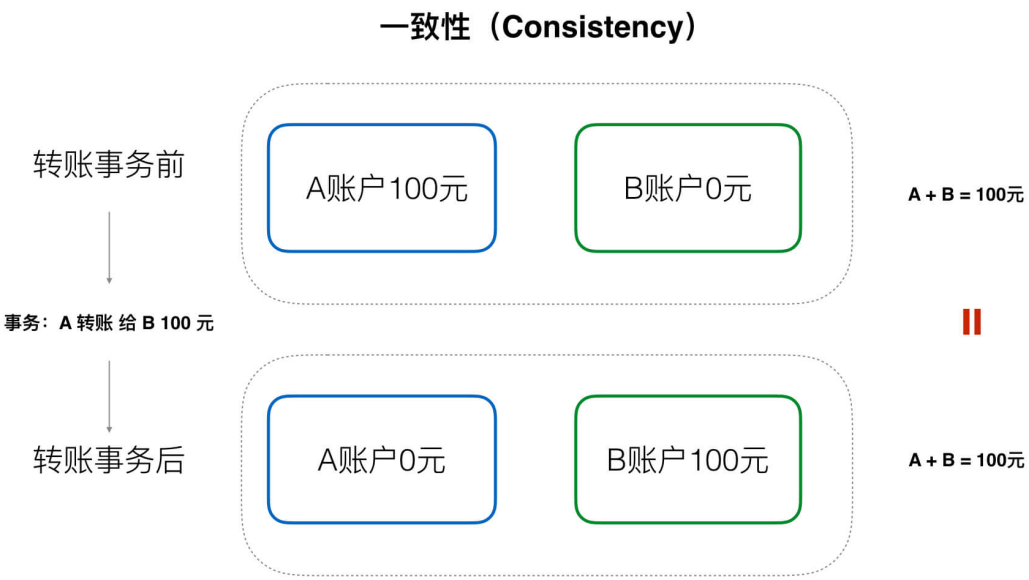
例如：银行转账，从A账户转100元至B账户：

A、从A账户取100元

B、存入100元至B账户。 这两步要么一起完成，要么一起不完成，如果只完成第一步，第二步失败，钱会莫名其妙少了100元。

(2) 一致性（Consistency）

在事务开始之前和事务结束以后，数据库数据的一致性约束没有被破坏。



例如：现有完整性约束A+B=100，如果一个事务改变了A，那么必须得改变B，使得事务结束后依然满足A+B=100，否则事务失败。

(3) 隔离性（Isolation）

数据库允许多个并发事务同时对数据进行读写和修改的能力，如果一个事务要访问的数据正在被另外一个事务修改，只要另外一个事务未提交，它所访问的数据就不受未提交事务的影响。隔离性可以防止多个事务并发执行时由于交叉执行而导致数据的不一致。

例如：现有有个交易是从A账户转100元至B账户，在这个交易事务还未完成的情况下，如果此时B查询自己的账户，是看不到新增加的100元的。

(4) 持久性（Durability）

事务处理结束后，对数据的修改就是永久的，即便系统故障也不会丢失。

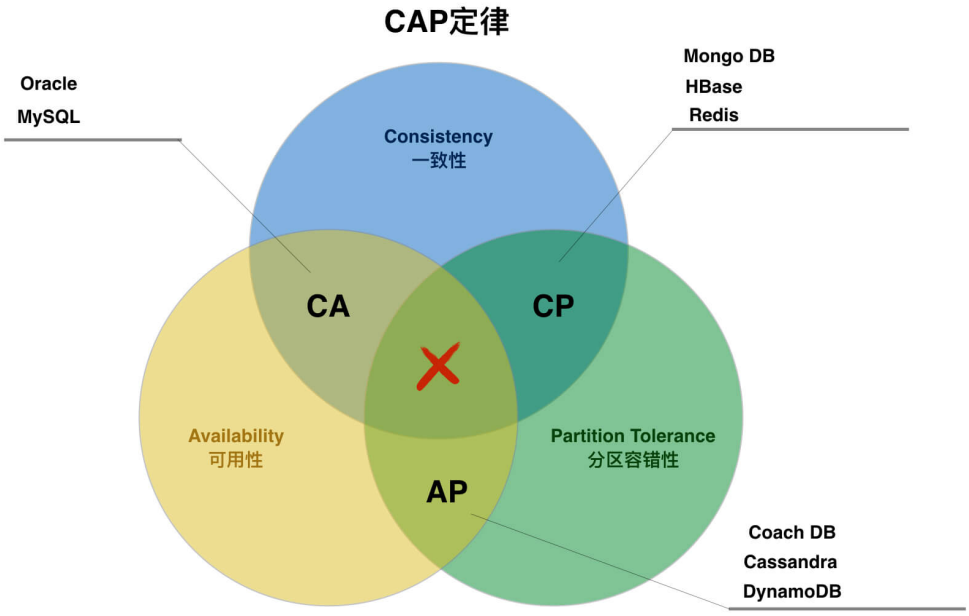
本地事务ACID实际上可用”统一提交，失败回滚“几个字总结，严格保证了同一事务内数据的一致性！

而分布式事务不能实现这种 `ACID` 。因为有CAP理论约束。接下来我们来了解一下，分布式中是如何保证以上特性的，那么就有了一个著名的CAP理论。

三、CAP理论

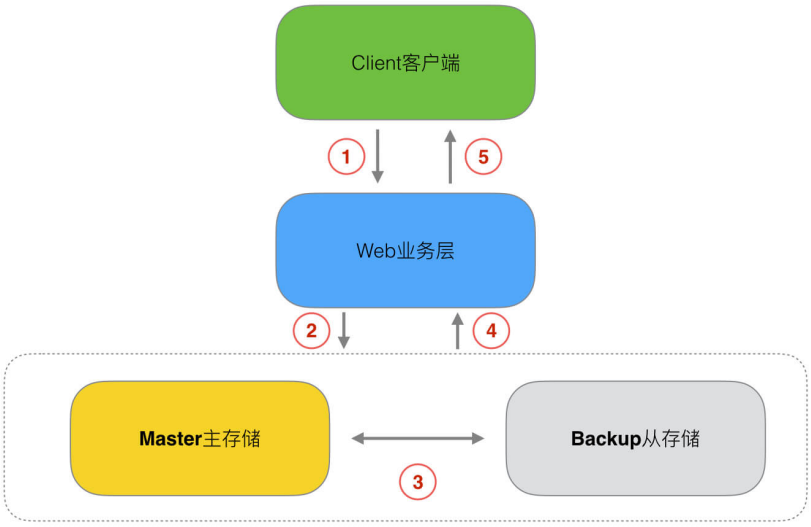
在设计一个大规模可扩放的网络服务时候会遇到三个特性：一致性（consistency）、可用性（Availability）、分区容错（partition-tolerance）都需要的情景.

CAP定律说的是在一个分布式计算机系统中，一致性，可用性和分区容错性这三种保证无法同时得到满足，最多满足两个。



如上图，CAP的三种特性只能同时满足两个。而且在不同的两两组合，也有一些成熟的分布式产品。

接下来，我们来介绍一下CAP的三种特性，我们采用一个应用场景来分析CAP中的每个特点的含义。



该场景整体分为5个流程：

流程一、客户端发送请求(如:添加订单、修改订单、删除订单)

流程二、Web业务层处理业务，并修改存储成数据信息

流程三、存储层内部Master与Backup的数据同步

流程四、Web业务层从存储层取出数据

流程五、Web业务层返回数据给客户端

(1) 一致性Consistency

> “ `all nodes see the same data at the same time` ”

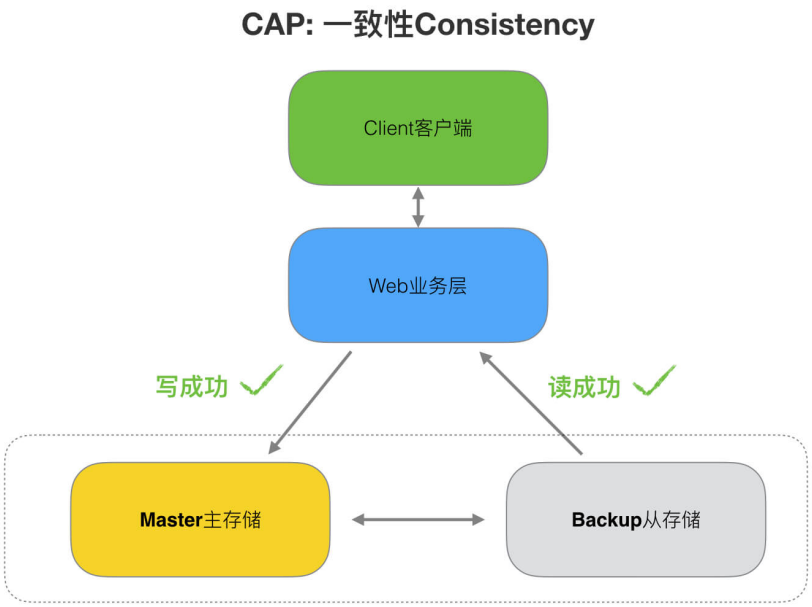
一旦数据更新完成并成功返回客户端后，那么分布式系统中所有节点在同一时间的数据完全一致。

> 在CAP的一致性中还包括强一致性、弱一致性、最终一致性等级别，稍后我们在后续章节介绍。

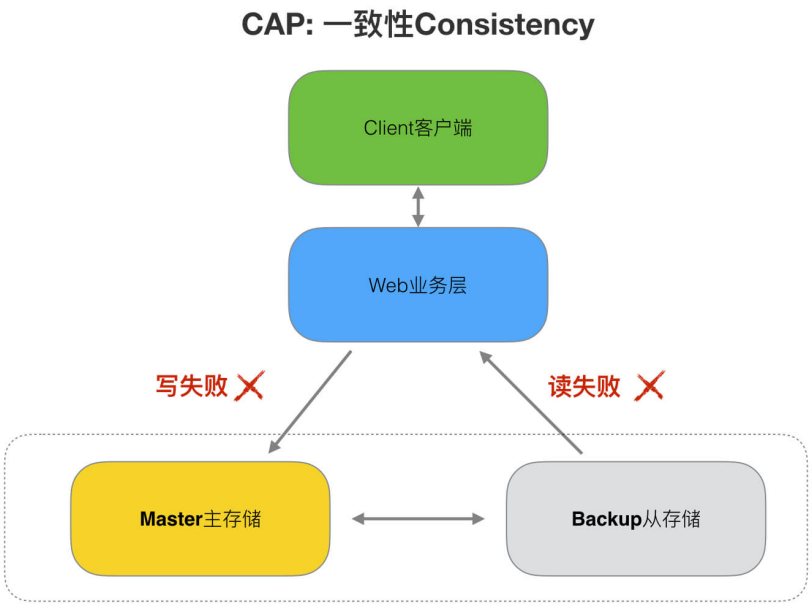
一致性是指写操作后的读操作可以读取到最新的数据状态，当数据分布在多个节点上，从任意结点读取到的数据都是最新的状态。

一致性实现目标：

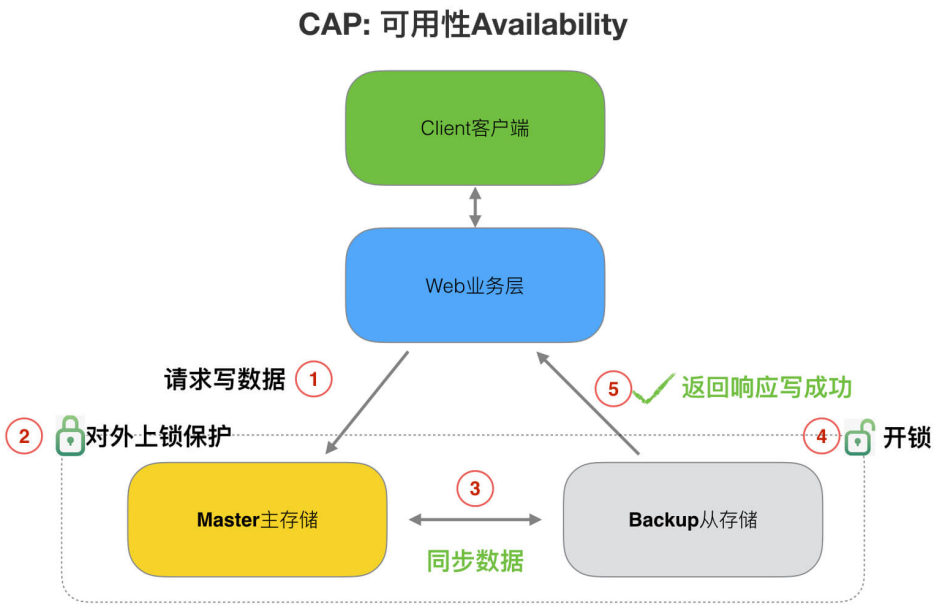
- Web业务层向主Master写数据库成功，从Backup读数据也成功。



- Web业务层向主Master读数据库失败，从Backup读数据也失败。



必要实现流程：



写入主数据库后，在向从数据库同步期间要将从数据库锁定，待同步完成后再释放锁，以免在新数据写入成功后，向从数据库查询到旧的数据。

分布式一致性特点：

1. 由于存在数据同步的过程，写操作的响应会有一定的延迟。
2. 为了保证数据一致性会对资源暂时锁定，待数据同步完成释放锁定资源。
3. 如果请求数据同步失败的结点则会返回错误信息，一定不会返回旧数据。

(2) 可用性(Availability)

> “ `Reads and writes always succeed` ”

服务一直可用，而且是正常响应时间。

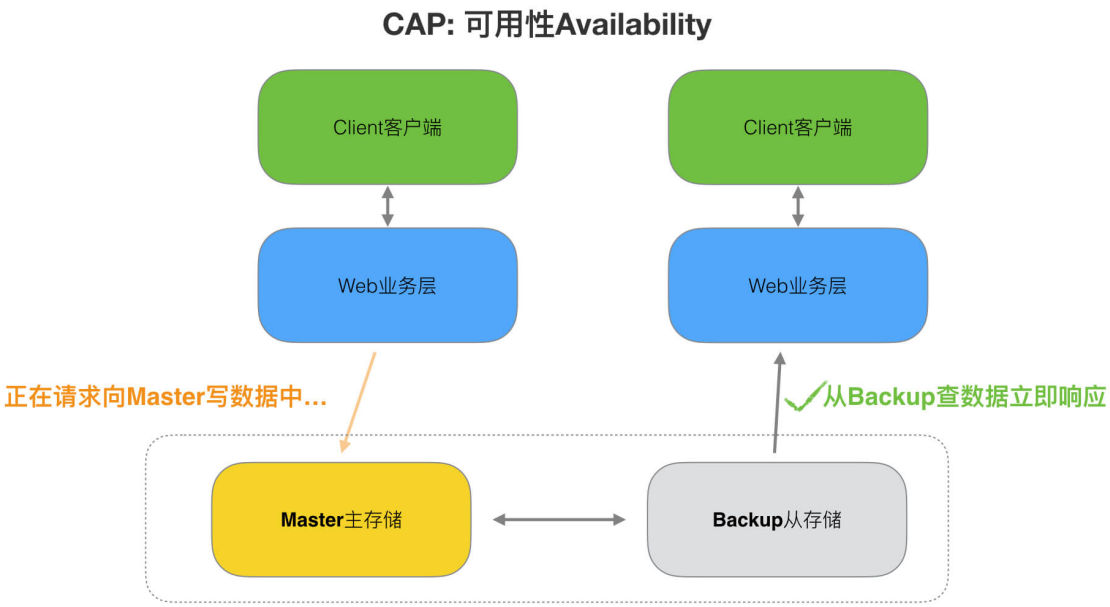
对于可用性的衡量标准如下：

可用性分类	可用水平（%）	一年中可容忍停机时间
容错可用性	99.9999	<1 min
极高可用性	99.999	<5 min
具有故障自动恢复能力的可用性	99.99	<53 min
高可用性	99.9	<8.8h
商品可用性	99	<43.8 min

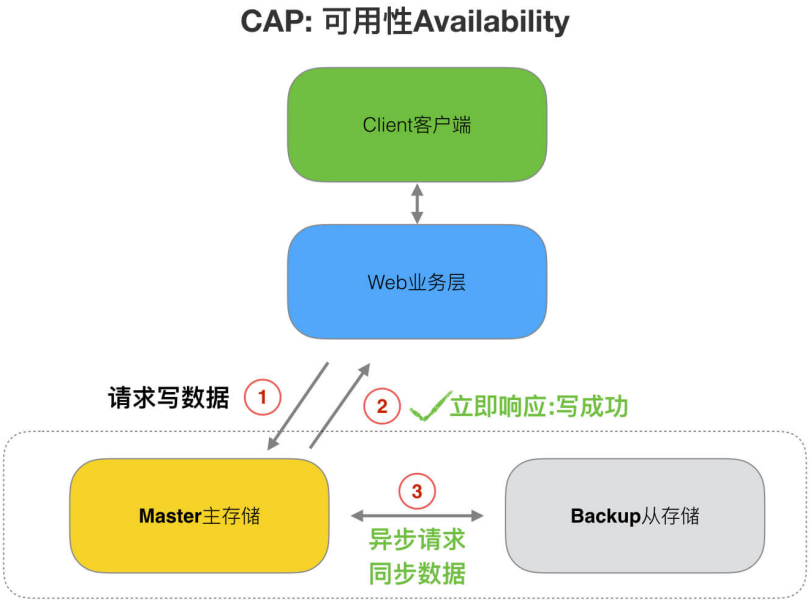
可用性实现目标：

- 当Master正在被更新，Backup数据库接收到数据查询的请求则立即能够响应数据查询结果。

- backup数据库不允许出现响应超时或响应错误。



必要实现流程：



1. 写入Master主数据库后要将数据同步到从数据库。
2. 由于要保证Backup从数据库的可用性，不可将Backup从数据库中的资源进行锁定。
3. 即时数据还没有同步过来，从数据库也要返回要查询的数据，哪怕是旧数据/或者默认数据，但不能返回错误或响应超时。

分布式可用性特点：

所有请求都有响应，且不会出现响应超时或响应错误。

(3) 分区容错性(Partition tolerance)

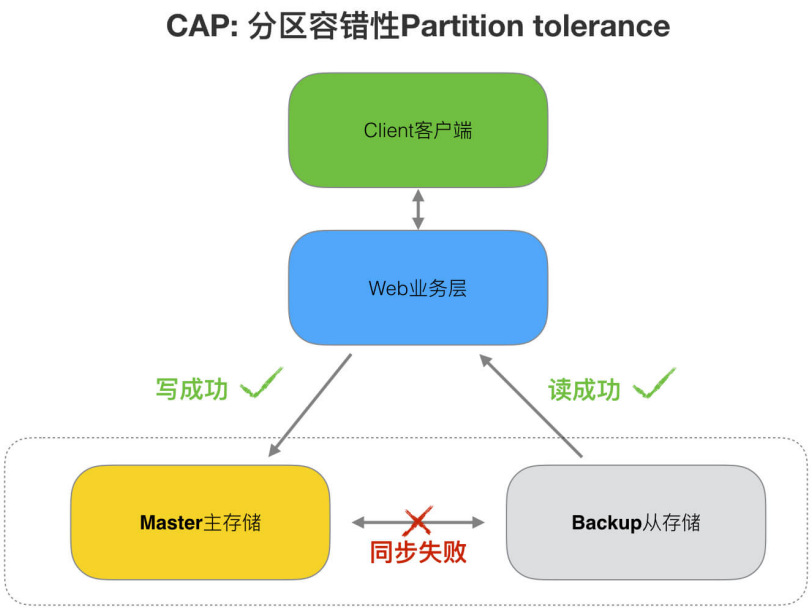
> “ `the system continues to operate despite arbitrary message loss or failure of part of the system` ”

分布式系统中，尽管部分节点出现任何消息丢失或者故障，系统应继续运行。

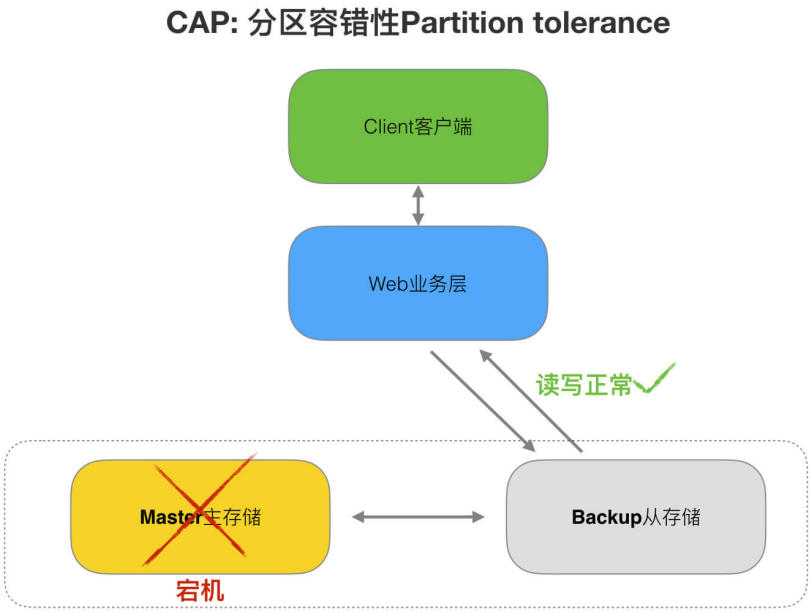
通常分布式系统的各各结点部署在不同的子网，这就是网络分区，不可避免的会出现由于网络问题而导致结点之间通信失败，此时仍可对外提供服务。

分区容错性实现目标：

- 主数据库向从数据库同步数据失败不影响读写操作。

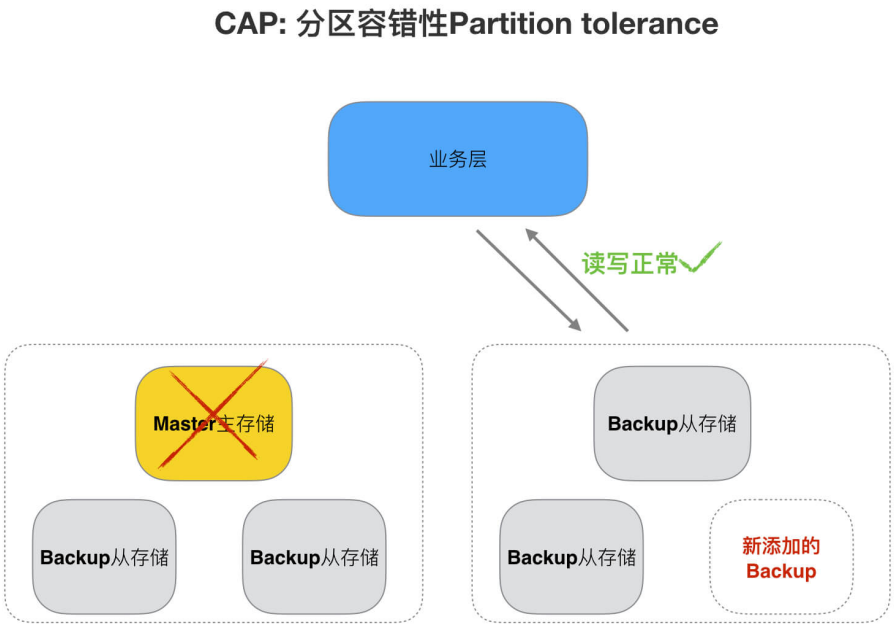


- 其一个结点挂掉不影响另一个结点对外提供服务。



必要实现流程：

1. 尽量使用异步取代同步操作，例如使用异步方式将数据从主数据库同步到从数据，这样结点之间能有效的实现松耦合。
2. 添加Backup从数据库结点，其中一个Backup从结点挂掉其它Backup从结点提供服务。



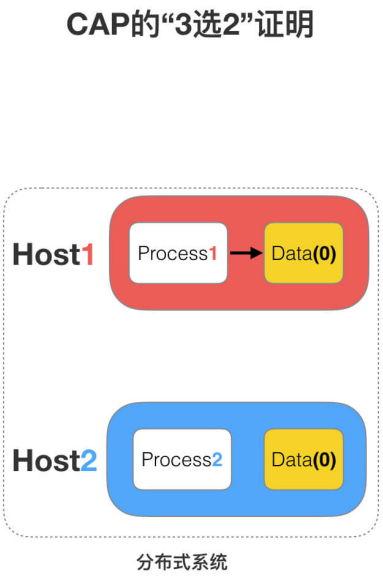
分区容错性特点：

分区容忍性分是布式系统具备的基本能力。

四、CAP的”3选2“证明

(1) 基本场景

在小结中，我们主要介绍CAP的理论为什么不能够3个特性同时满足。



如上图，是我们证明CAP的基本场景，分布式网络中有两个节点Host1和Host2，他们之间网络可以连通，Host1中运行Process1程序和对应的数据库Data，Host2中运行Process2程序和对应数据库Data。

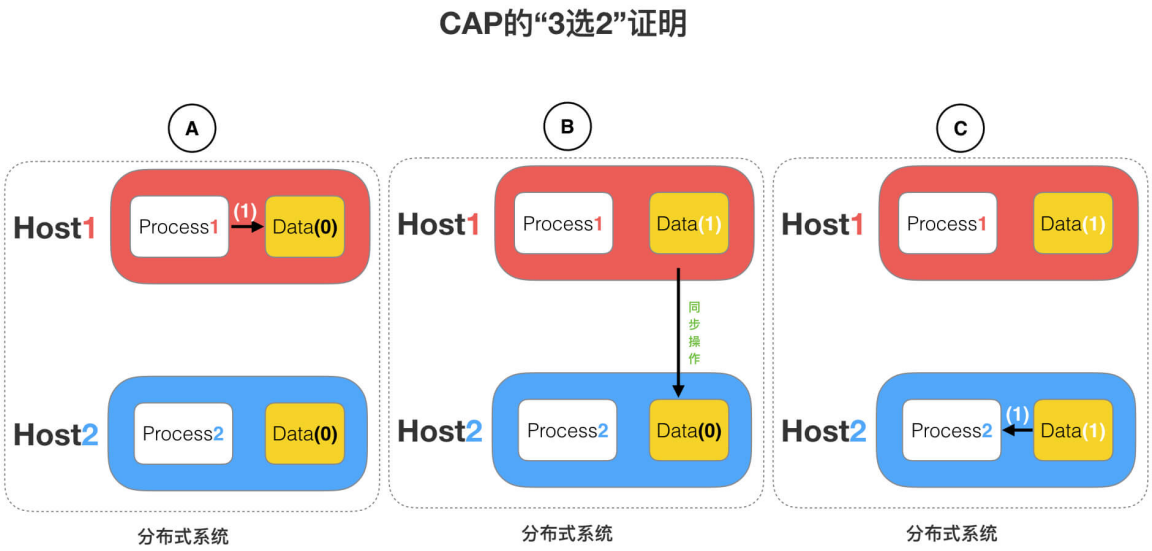
(2) CAP特性

如果满足一致性(C)：那么 Data(0) = Data(0) .

如果满足可用性(A)：用户不管请求Host1或Host2，都会立刻响应结果。

如果满足分区容错性(P)：Host1或Host2有一方脱离系统(故障)， 都不会影响Host1和Host2彼此之间正常运作。

(3) 分布式系统正常运行流程



如上图，是分布式系统正常运转的流程。

A、用户向 Host1 主机请求数据更新，程序 Process1 更新数据库 Data(0) 为 Data(1)

B、分布式系统将数据进行同步操作，将 Host1 中的 Data(1) 同步的 Host2 中 Data(0) ，使 Host2 中的数据也变为 Data(1)`

C、当用户请求主机 Host2 时，则 Process2 则响应最新的 Data(1) 数据

根据CAP的特性：

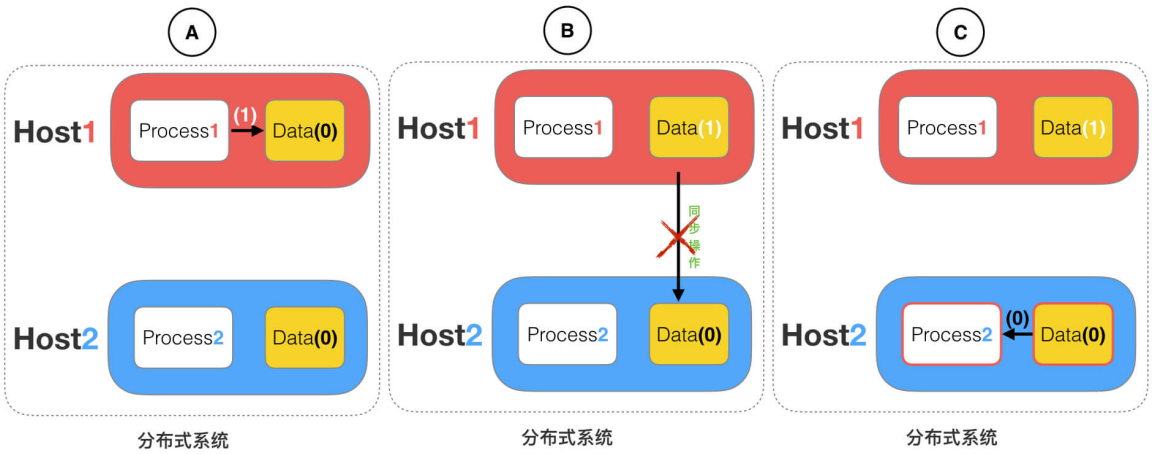
- Host1 和 Host2 的数据库 Data 之间的数据是否一样为一致性(C)
- 用户对 Host1 和 Host2 的请求响应为可用性(A)
- Host1 和 Host2 之间的各自网络环境为分区容错性(P)

当前是一个正常运作的流程，目前CAP三个特性可以同时满足，也是一个 理想状态 ,但是实际应用场景中，发生错误在所难免，那么如果发生错误CAP是否能同时满足，或者该如何取舍？

---

(4) 分布式系统异常运行流程

假设 Host1 和 Host2 之间的网络断开了，我们要支持这种网络异常，相当于要满足 分区容错性(P) ，能不能同时满足 一致性(C) 和 可用响应性(A) 呢？



假设在N1和N2之间网络断开的时候，

A、用户向 `Host1` 发送数据更新请求，那 `Host1` 中的数据 `Data(0)` 将被更新为 `Data(1)`

B、弱此时 `Host1` 和 `Host2` 网络是断开的，所以分布式系统同步操作将失败， `Host2` 中的数据依旧是 `Data(0)`

C、有用户向 `Host2` 发送数据读取请求，由于数据还没有进行同步， `Process2` 没办法立即给用户返回最新的数据V1，那么将面临两个选择。

第一，牺牲 `数据一致性(c)` ，响应旧的数据 `Data(0)` 给用户；

第二，牺牲 `可用性(A)` ，阻塞等待，直到网络连接恢复，数据同步完成之后，再给用户响应最新的数据 `Data(1)` 。

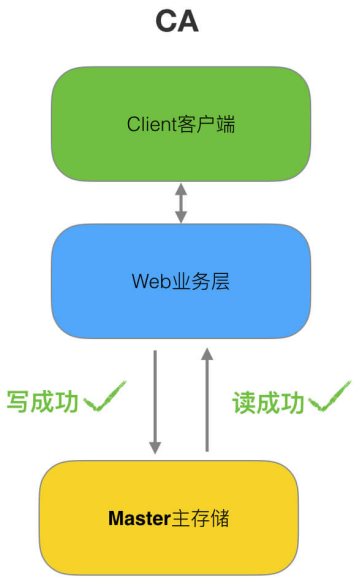
这个过程，证明了要满足 `分区容错性(p)` 的分布式系统，只能在 `一致性(C)` 和 `可用性(A)` 两者中，选择其中一个。

(5) “3选2”的必然性

通过CAP理论，我们知道无法同时满足 `一致性` 、 `可用性` 和 `分区容错性` 这三个特性，那要舍弃哪个呢？

CA 放弃 P：

一个分布式系统中，不可能存在不满足P，放弃 `分区容错性(p)` ，即不进行分区，不考虑由于网络不通或结点挂掉的问题，则可以实现一致性和可用性。那么系统将不是一个标准的分布式系统。我们最常用的关系型数据就满足了CA，如下：



主数据库和从数据库中间不再进行数据同步，数据库可以响应每次的查询请求，通过事务(原子性操作)隔离级别实现每个查询请求都可以返回最新的数据。

注意：

> 对于一个分布式系统来说。P是一个基本要求，CAP三者中，只能在CA两者之间做权衡，并且要想尽办法提升P。

CP 放弃 A

如果一个分布式系统不要求强的可用性，即容许系统停机或者长时间无响应的话，就可以在CAP三者中保障CP而舍弃A。

放弃可用性，追求一致性和分区容错性，如Redis、HBase等，还有分布式系统中常用的Zookeeper也是在CAP三者之中选择优先保证CP的。

场景：

跨行转账，一次转账请求要等待双方银行系统都完成整个事务才算完成。

AP 放弃 C

放弃一致性，追求分区容忍性和可用性。这是很多分布式系统设计时的选择。实现AP，前提是只要用户可以接受所查询的到数据在一定时间内不是最新的即可。

通常实现AP都会保证最终一致性，后面讲的BASE理论就是根据AP来扩展的。

场景1：

淘宝订单退款。今日退款成功，明日账户到账，只要用户可以接受在一定时间内到账即可。

场景2：

12306的买票。都是在可用性和一致性之间舍弃了一致性而选择可用性。

你在12306买票的时候肯定遇到过这种场景，当你购买的时候提示你是有票的（但是可能实际已经没票了），你也正常的去输入验证码，下单了。但是过了一会系统提示你下单失败，余票不足。这其实就是先在可用性方面保证系统可以正常的服务，然后在数据的一致性方面做了些牺牲，会影响一些用户体验，但是也不至于造成用户流程的严重阻塞。

但是，我们说很多网站牺牲了一致性，选择了可用性，这其实也不准确的。就比如上面的买票的例子，其实舍弃的只是强一致性。退而求其次保证了最终一致性。也就是说，虽然下单的瞬间，关于车票的库存可能存在数据不一致的情况，但是过了一段时间，还是要保证最终一致性的。

(6) 总结:

> CA 放弃 P：如果不要求P（不允许分区），则C（强一致性）和A（可用性）是可以保证的。这样分区将永远不会存在，因此CA的系统更多的是允许分区后各子系统依然保持CA。

> CP 放弃 A：如果不要求A（可用），相当于每个请求都需要在Server之间强一致，而P（分区）会导致同步时间无限延长，如此CP也是可以保证的。很多传统的数据库分布式事务都属于这种模式。

> AP 放弃 C：要高可用并允许分区，则需放弃一致性。一旦分区发生，节点之间可能会失去联系，为了高可用，每个节点只能用本地数据提供服务，而这样会导致全局数据的不一致性。现在众多的NoSQL都属于此类。

五、思考

思考：按照CAP理论如何设计一个电商系统？

- 首先个电商网站核心模块有**用户，订单，商品，支付，促销管理**等

> 1、对于用户模块，包括登录，个人设置，个人订单，购物车，收藏夹等，这些模块保证AP，数据短时间不一致不影响使用。
> 2、订单模块的下单付款扣减库存操作是整个系统的核心，CA都需要保证，极端情况下面牺牲A保证C
> 3、商品模块的商品上下架和库存管理保证CP
> 4、搜索功能因为本身就不是实时性非常高的模块，所以保证AP就可以了。
> 5、促销是短时间的数据不一致，结果就是优惠信息看不到，但是已有的优惠要保证可用，而且优惠可以提前预计算，所以可以保证AP。
> 6、支付这一块是独立的系统，或者使用第三方的支付宝，微信。其实CAP是由第三方来保证的，支付系统是一个对CAP要求极高的系统，C是必须要保证的，AP中A相对更重要，不能因为分区，导致所有人都不能支付

六、分布式BASE理论

CAP 不可能同时满足，而 `分区容错性(P)` 是对于分布式系统而言是必须的。如果系统能够同时实现 CAP 是再好不过的了，所以出现了 BASE 理论。

(1) BASE理论

> 通用定义

BASE是**Basically Available(基本可用）**、**Soft state(软状态）**和**Eventually consistent(最终一致性）**三个短语的简写。

BASE是对CAP中一致性和可用性权衡的结果，其来源于对大规模互联网系统分布式实践的总结，是**基于CAP定理逐步演化**而来的，其核心思想是即使无法做到强一致性，但每个应用都可以根据自身的业务特点，采用适当的方法来使系统达到**最终一致性**。

> 两个对冲理念：ACID和BASE

`ACID` 是传统数据库常用的设计理念， `追求强一致性` 模型。

`BASE` 支持的是大型分布式系统，提出通过 `牺牲强一致性` 获得 `高可用性` 。

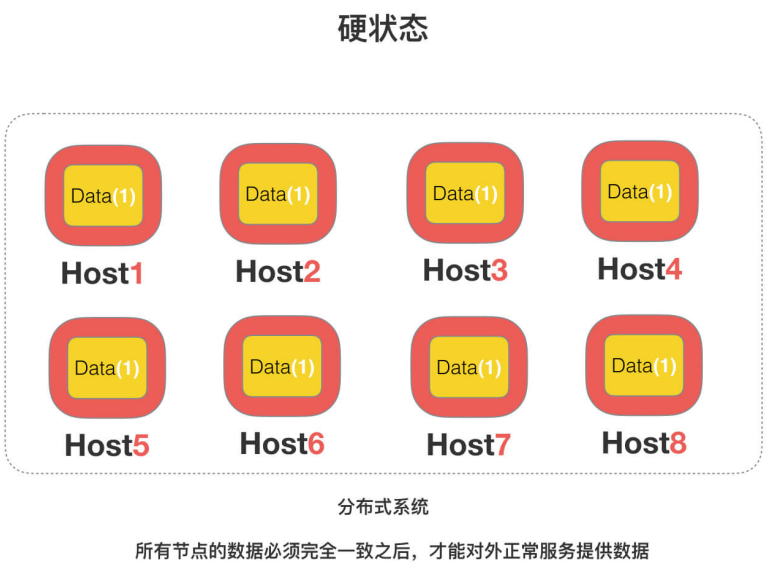
(2) Basically Available(基本可用)

实际上就是两个妥协。

- 对响应上时间的妥协：正常情况下，一个在线搜索引擎需要在0.5秒之内返回给用户相应的查询结果，但由于出现故障（比如系统部分机房发生断电或断网故障），查询结果的响应时间增加到了1~2秒。
- 对功能损失的妥协：正常情况下，在一个电子商务网站（比如淘宝）上购物，消费者几乎能够顺利地完成每一笔订单。但在一些节日大促购物高峰的时候（比如双十一、双十二），由于消费者的购物行为激增，为了保护系统的稳定性（或者保证一致性），部分消费者可能会被引导到一个降级页面，如下：

(3) Soft state（软状态）

- 原子性（硬状态） -> 要求多个节点的数据副本都是一致的,这是一种”硬状态”



- 软状态（弱状态） -> 允许系统中的数据存在中间状态,并认为该状态不影响系统的整体可用性,即允许系统在多个不同节点的数据副本存在数据延迟。

(4) Eventually consistent（最终一致性）

上面说软状态，然后不可能一直是软状态，必须有个时间期限。在期限过后，应当保证所有副本保持数据一致性。从而达到数据的最终一致性。这个时间期限取决于网络延时，系统负载，数据复制方案设计等等因素。

稍微官方一点的说法就是：

> 系统能够保证在没有其他新的更新操作的情况下，数据最终一定能够达到一致的状态，因此所有客户端对系统的数据访问最终都能够获取到最新的值。

(5) BASE总结

总的来说，BASE 理论面向的是大型高可用可扩展的分布式系统，和传统事务的 ACID 是**相反的**，它完全不同于 ACID 的强一致性模型，而是**通过牺牲强一致性**来获得可用性，并允许数据在一段时间是不一致的。

参考：

https://blog.csdn.net/weixin_44062339/article/details/99710968

https://blog.csdn.net/w372426096/article/details/80437198

https://www.solves.com.cn/it/cxkf/bk/2019-09-24/5229.html

https://www.jianshu.com/p/46b90dfc7c90

https://www.jianshu.com/p/9cb2a6fa4e0e

https://www.jianshu.com/p/68c7c16b3fbd

对于操作系统而言进程、线程以及Goroutine协程的区别

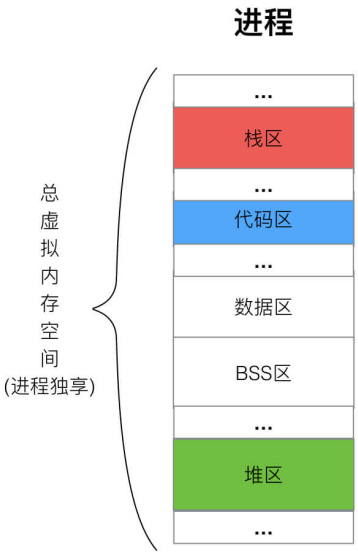
进程、线程、协程实际上都是为并发而生。

但是他们的各自的模样是完全不一致的，下面我们来分析一下他们各自的特点和关系。

> 本文不重点介绍什么是进程和线程，而是提炼进程、线程、协程干货。且是基于Linux下的进程、线程解释

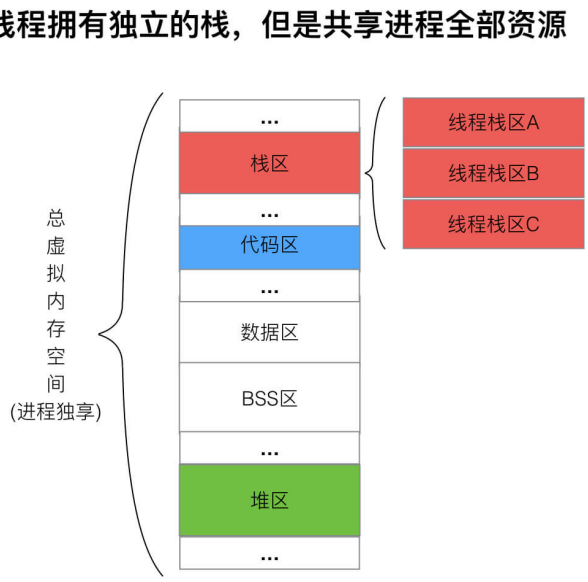
一、进程内存

进程，可执行程序运行中形成一个独立的内存体，这个内存体**有自己独立的地址空间(Linux会给每个进程分配一个虚拟内存空间32位操作系统为4G, 64位为很多T)，有自己的堆**，上级挂靠单位是操作系统。**操作系统会以进程为单位，分配系统资源（CPU时间片、内存等资源），进程是资源分配的最小单位**。



二、线程内存

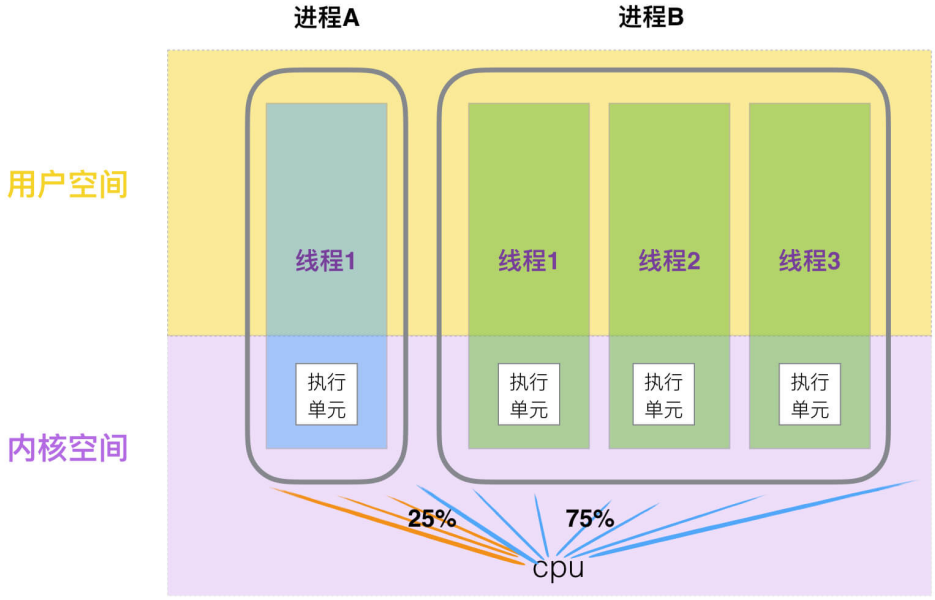
线程，有时被称为轻量级进程(Lightweight Process，LWP），是操作系统调度（CPU调度）执行的最小单位。



多个线程共同“寄生”在一个进程上，除了拥有各自的栈空间，其他的内存空间都是一起共享。所以由于这个特性，使得线程之间的内存关联性很大，互相通信就很简单(堆区、全局区等数据都共享，需要加锁机制即可完成同步通信)，但是同时也让线程之间生命体联系较大，比如一个线程出问题，到底进程问题，也就导致了其他线程问题。

三、执行单元

对于Linux来讲，不区分进程还是线程，他们都是一个单独的执行单位，CPU一视同仁，均分配时间片。

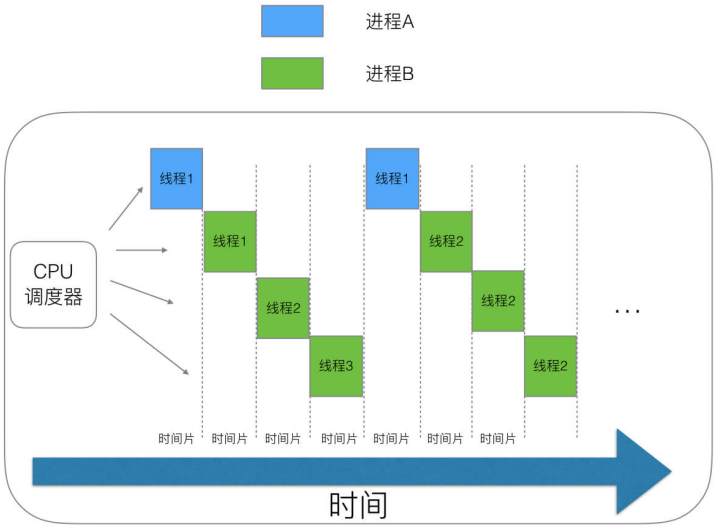


所以，如果一个进程想更大程度的与其他进程抢占CPU的资源，那么多开线程是一个好的办法。

如上图，进程A没有开线程，那么默认就是 `1个线程` ，对于内核来讲，它只有1个 `执行单元` ,进程B开了 `3个线程` ，那么在内核中，该进程就占有3个 `执行单元` 。CPU的视野是只能看见内核的，它不知晓谁是进程和谁是线程，谁和谁是一家人。时间片轮询平均调度分配。那么进程B拥有的3个单元就有了资源供给的优势。

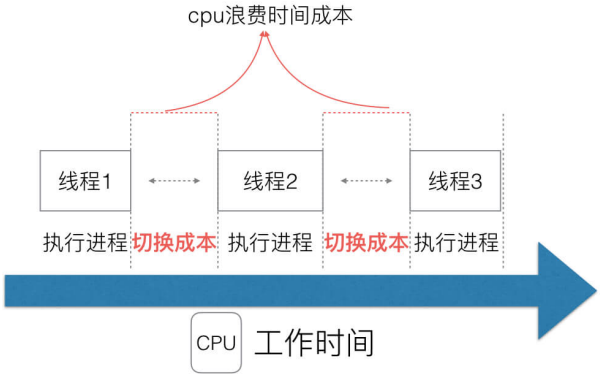
四、切换问题与协程

我们通过上述的描述，可以知道，线程越多，进程利用(或者)抢占的cpu资源就越高。



那么是不是线程可以无限制的多呢？

答案当然不是的，我们知道，当我们cpu在内核态切换一个 执行单元 的时候，会有一个时间成本和性能开销



其中性能开销至少会有两个开销

- 切换内核栈
- 切换硬件上下文

> 这两个切换，我们没必要太深入研究，可以理解为他所带来的后果和影响是

- 保存寄存器中的内容

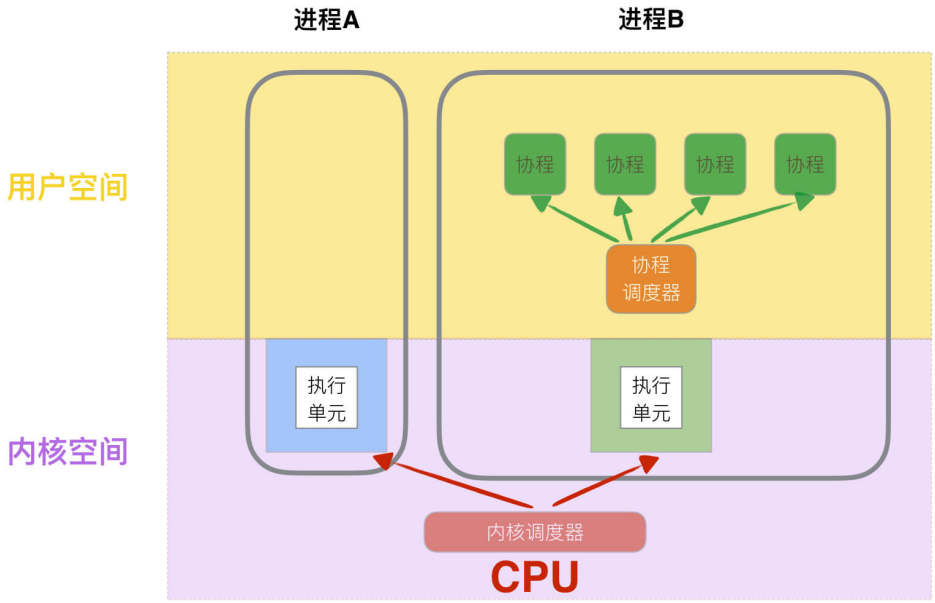
  将之前执行流程的状态保存。

- CPU高速缓存失效

  页表查找是一个很慢的过程，因此通常使用Cache来缓存常用的地址映射，这样可以加速页表查找，这个cache就是TLB.当进程切换后页表也要进行切换，页表切换后TLB就失效了，cache失效导致命中率降低，那么虚拟地址转换为物理地址就会变慢，**表现出来的就是程序运行会变慢**。

综上，我们不能够大量的开辟，因为 线程执行流程 越多，cpu在切换的时间成本越大。很多编程语言就想了办法，既然我们不能左右和优化cpu切换线程的开销，那么，我们能否让cpu内核态不切换 执行单元 ， 而是在用户态切换执行流程呢？

很显然，我们是没权限修改操作系统内核机制的，那么只能在用户态再来一个 伪执行单元 ,那么就是 协程 了。



五、协程的切换成本

协程切换比线程切换快主要有两点：

（1）协程切换**完全在用户空间进行**线程切换涉及**特权模式切换，需要在内核空间完成**;

（2）协程切换相比线程切换**做的事情更少**，线程需要有内核和用户态的切换,系统调用过程。

协程切换成本：

协程切换非常简单，就是把**当前协程的 CPU 寄存器状态保存起来，然后将需要切换进来的协程的 CPU 寄存器状态加载的 CPU 寄存器上**就 ok 了。而且**完全在用户态进行**，一般来说一次协程上下文切换最多就是**几十ns** 这个量级。

线程切换成本：

系统内核调度的对象是线程，因为线程是调度的基本单元（进程是资源拥有的基本单元，进程的切换需要做的事情更多，这里占时不讨论进程切换），而**线程的调度只有拥有最高权限的内核空间才可以完成**，所以线程的切换涉及到**用户空间和内核空间的切换**，也就是特权模式切换，然后需要操作系统调度模块完成**线程调度（task*struct）**，*而且除了和协程相同基本的 CPU 上下文，还有线程私有的栈和寄存器等，说白了就是上下文比协程多一些，其实简单比较下 task_strcut 和 任何一个协程库的 coroutine 的 struct 结构体大小就能明显区分出来。而且特权模式切换的开销确实不小，随便搜一组测试数据 [3]，随便算算都比协程切换开销大很多。

进程占用多少内存

4g

线程占用多少内存

线程跟不同的操作系统版本有有差异

```
$ulimit -s
8192
```

单位 `kb`

但线程基本都是维持Mb的量级单位，一般是4~64Mb不等， 多数维持约10M上下

协程占用多少内存

测试环境

```
$ more /proc/cpuinfo | grep "model name"
model name      : Intel(R) Core(TM) i7-5775R CPU @ 3.30GHz
model name      : Intel(R) Core(TM) i7-5775R CPU @ 3.30GHz

(2个CPU )

$ grep MemTotal /proc/meminfo
MemTotal:        2017516 kB

(2G内存)

$ getconf LONG_BIT
64

(64位操作系统)

$ uname -a
Linux ubuntu 4.15.0-91-generic #92-Ubuntu SMP Fri Feb 28 11:09:48 UTC 2020 x86_64 x86_64 x86_64 GNU/Linux
```

测试程序

```
package main

import (

    "time"
)

func main() {

    for i := 0; i < 200000; i++ {
```

```
        go func() {

            time.Sleep(5 * time.Second)

        }()

    }

    time.Sleep(10 * time.Second)
}
```

程序运行前

```
top - 00:16:24 up  7:08,  1 user,  load average: 0.08, 0.03, 0.01
任务: 288 total,   1 running, 218 sleeping,   0 stopped,   0 zombie
%Cpu0  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu1  :  0.3 us,  0.3 sy,  0.0 ni, 99.3 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  2017516 total,   593836 free,  1163524 used,   260156 buff/cache
KiB Swap:   969960 total,   574184 free,   395776 used.   679520 avail Mem
```

free的mem为1163524，

程序运行中

```
top - 00:17:12 up  7:09,  1 user,  load average: 0.04, 0.02, 0.00
任务: 290 total,   1 running, 220 sleeping,   0 stopped,   0 zombie
%Cpu0  :  4.0 us,  1.0 sy,  0.0 ni, 95.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu1  :  8.8 us,  1.4 sy,  0.0 ni, 89.9 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  2017516 total,    89048 free,  1675844 used,   252624 buff/cache
KiB Swap:   969960 total,   563688 free,   406272 used.   168812 avail Mem
```

free的mem为1675844，

所以**20万个协程占用了约 50万KB****平均一个协程占用约2.5KB****

那么，go的协程切换成本如此小，占用也那么小，是否可以无限开辟呢？

Go是否可以无限go？ 如何限定数量？

一、不控制goroutine数量引发的问题

我们都知道Goroutine具备如下两个特点

- 体积轻量
- 优质的GMP调度

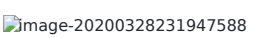
那么goroutine是否可以无限开辟呢，如果做一个服务器或者一些高业务的场景，能否随意的开辟goroutine并且放养不管呢？让他们自生自灭，毕竟有强大的GC和优质的调度算法支撑？

那么我可以先看如下一个问题。

> code1.go

```
package main

import (
    "fmt"
    "math"
    "runtime"
)

func main() {
    //模拟用户需求业务的数量
    task_cnt := math.MaxInt64

    for i := 0; i < task_cnt; i++ {
        go func(i int) {
            //... do some busi...

            fmt.Println("go func ", i, " goroutine count = ", runtime.NumGoroutine())
        }(i)
    }
}
```

结果



最后被操作系统以kill信号，强制终结该进程。

```
signal: killed
```

所以，我们迅速的开辟goroutine(**不控制并发的 goroutine 数量**)会在短时间内占据操作系统的资源(CPU、内存、文件描述符等)。

- CPU 使用率浮动上涨
- Memory 占用不断上涨。
- 主进程崩溃（被杀掉了）

这些资源实际上是所有用户态程序共享的资源，所以大批的goroutine最终引发的灾难不仅仅是自身，还会关联其他运行的程序。

所以在编写逻辑业务的时候，限制goroutine是我们必须要重视的问题。

二、一些简单方法控制goroutines数量

方法一：只是用有buffer的channel来限制

> code2.go

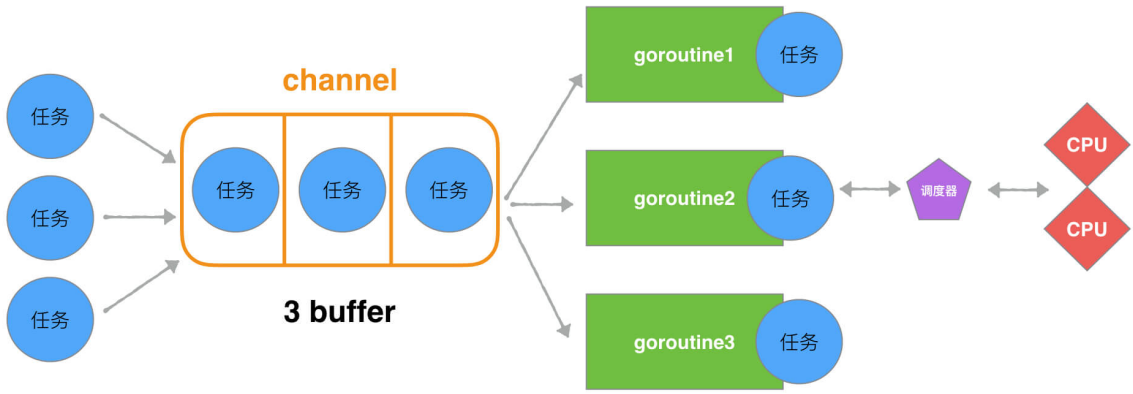
```
package main

import (
    "fmt"
    "math"
    "runtime"
)

func busi(ch chan bool, i int) {

    fmt.Println("go func ", i, " goroutine count = ", runtime.NumGoroutine())
    <-ch
}

func main() {
    //模拟用户需求业务的数量
    task_cnt := math.MaxInt64
    //task_cnt := 10

    ch := make(chan bool, 3)

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

        ch <- true

        go busi(ch, i)
    }

}
```

结果

```
...
go func  352277  goroutine count =  4
go func  352278  goroutine count =  4
go func  352279  goroutine count =  4
go func  352280  goroutine count =  4
go func  352281  goroutine count =  4
go func  352282  goroutine count =  4
go func  352283  goroutine count =  4
go func  352284  goroutine count =  4
go func  352285  goroutine count =  4
go func  352286  goroutine count =  4
go func  352287  goroutine count =  4
go func  352288  goroutine count =  4
go func  352289  goroutine count =  4
go func  352290  goroutine count =  4
go func  352291  goroutine count =  4
go func  352292  goroutine count =  4
```

```
go func  352293  goroutine count =  4
go func  352294  goroutine count =  4
go func  352295  goroutine count =  4
go func  352296  goroutine count =  4
go func  352297  goroutine count =  4
go func  352298  goroutine count =  4
go func  352299  goroutine count =  4
go func  352300  goroutine count =  4
go func  352301  goroutine count =  4
go func  352302  goroutine count =  4
...
```

从结果看，程序并没有出现崩溃，而是按部就班的顺序执行，并且go的数量控制在了3，(4的原因是因为还有一个main goroutine)那么从数字上看，是不是在跑的goroutines有几十万个呢？



这里我们用了，buffer为3的channel, 在写的过程中，实际上是限制了速度。限制的是

```
for i := 0; i < go_cnt; i++ { //循环速度

        ch <- true

        go busi(ch, i)
}
```

`for` 循环的速度，因为这个速度决定了go的创建速度，而go的结束速度取决于 `busi()` 函数的执行速度。 这样实际上，我们就能够保证了，同一时间内运行的goroutine的数量与buffer的数量一致。从而达到了限定效果。

但是这段代码有一个小问题，就是如果我们把go_cnt的数量变的小一些，会出现打出的结果不正确。

```
package main

import (
    "fmt"
    //"math"
    "runtime"
)
```

```
func busi(ch chan bool, i int) {

    fmt.Println("go func ", i, " goroutine count = ", runtime.NumGoroutine())
    <-ch
}

func main() {
    //模拟用户需求业务的数量
    //task_cnt := math.MaxInt64
    task_cnt := 10

    ch := make(chan bool, 3)

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

        ch <- true

        go busi(ch, i)
    }

}
```

结果

```
go func  2  goroutine count =  4
go func  3  goroutine count =  4
go func  4  goroutine count =  4
go func  5  goroutine count =  4
go func  6  goroutine count =  4
go func  1  goroutine count =  4
go func  8  goroutine count =  4
```

是因为 `main` 将全部的go开辟完之后，就立刻退出进程了。所以想全部go都执行，需要在main的最后进行阻塞操作。

方法二：只使用sync同步机制

> code3.go

```
import (
    "fmt"
    "math"
    "sync"
    "runtime"
)

var wg = sync.WaitGroup{}

func busi(i int) {

    fmt.Println("go func ", i, " goroutine count = ", runtime.NumGoroutine())
    wg.Done()
}

func main() {
    //模拟用户需求业务的数量
    task_cnt := math.MaxInt64
```

```
    for i := 0; i < task_cnt; i++ {
        wg.Add(1)
        go busi(i)
    }

    wg.Wait()
}
```

很明显，单纯的使用 `sync` 依然达不到控制goroutine的数量，所以最终结果依然是崩溃。

结果

```
...
go func  7562  goroutine count =  7582
go func  24819  goroutine count =  17985
go func  7685  goroutine count =  7582
go func  24701  goroutine count =  17984
go func  7563  goroutine count =  7582
go func  24821  goroutine count =  17983
go func  24822  goroutine count =  17983
go func  7686  goroutine count =  7582
go func  24703  goroutine count =  17982
go func  7564  goroutine count =  7582
go func  24824  goroutine count =  17981
go func  7687  goroutine count =  7582
go func  24705  goroutine count =  17980
go func  24706  goroutine count =  17980
go func  24707  goroutine count =  17979
go func  7688  goroutine count =  7582
go func  24826  goroutine count =  17978
go func  7566  goroutine count =  7582
go func  24709  goroutine count =  17977
go func  7689  goroutine count =  7582
go func  24828  goroutine count =  17976
go func  24829  goroutine count =  17976
go func  7567  goroutine count =  7582
go func  24711  goroutine count =  17975
//操作系统停止响应
```

方法三：channel与sync同步组合方式

> code4.go

```
package main

import (
    "fmt"
    "math"
    "sync"
    "runtime"
)

var wg = sync.WaitGroup{}

func busi(ch chan bool, i int) {

    fmt.Println("go func ", i, " goroutine count = ", runtime.NumGoroutine())
```

```
    <-ch

    wg.Done()
}

func main() {
    //模拟用户需求go业务的数量
    task_cnt := math.MaxInt64

    ch := make(chan bool, 3)

    for i := 0; i < task_cnt; i++ {
        wg.Add(1)

        ch <- true

        go busi(ch, i)
    }

     wg.Wait()
}
```

结果

```
//...
go func  228851  goroutine count =  4
go func  228852  goroutine count =  4
go func  228853  goroutine count =  4
go func  228854  goroutine count =  4
go func  228855  goroutine count =  4
go func  228856  goroutine count =  4
go func  228857  goroutine count =  4
go func  228858  goroutine count =  4
go func  228859  goroutine count =  4
go func  228860  goroutine count =  4
go func  228861  goroutine count =  4
go func  228862  goroutine count =  4
go func  228863  goroutine count =  4
go func  228864  goroutine count =  4
go func  228865  goroutine count =  4
go func  228866  goroutine count =  4
go func  228867  goroutine count =  4
//...
```

这样我们程序就不会再造成资源爆炸而崩溃。而且运行go的数量控制住了在buffer为3的这个范围内。

方法四：利用无缓冲channel与任务发送/执行分离方式

> code5.go

```
package main

import (
    "fmt"
    "math"
    "sync"
    "runtime"
```

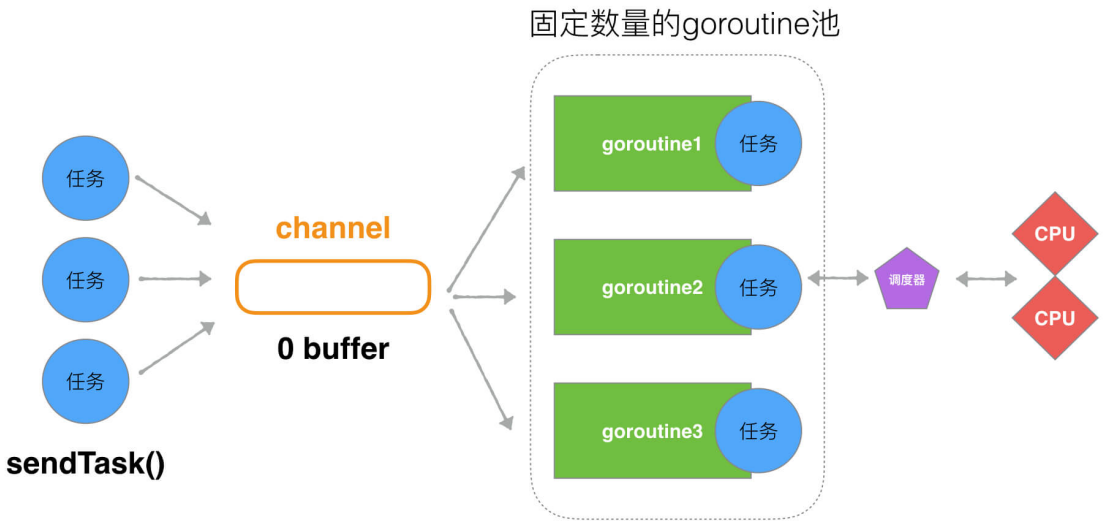
```
)

var wg = sync.WaitGroup{}

func busi(ch chan int) {

    for t := range ch {
        fmt.Println("go task = ", t, ", goroutine count = ", runtime.NumGoroutine())
        wg.Done()
    }
}

func sendTask(task int, ch chan int) {
    wg.Add(1)
    ch <- task
}

func main() {

    ch := make(chan int)   //无buffer channel

    goCnt := 3              //启动goroutine的数量
    for i := 0; i < goCnt; i++ {
        //启动go
        go busi(ch)
    }

    taskCnt := math.MaxInt64 //模拟用户需求业务的数量
    for t := 0; t < taskCnt; t++ {
        //发送任务
        sendTask(t, ch)
    }

      wg.Wait()
}
```

结构

```
//...
go task =  130069 , goroutine count =  4
go task =  130070 , goroutine count =  4
go task =  130071 , goroutine count =  4
go task =  130072 , goroutine count =  4
go task =  130073 , goroutine count =  4
go task =  130074 , goroutine count =  4
go task =  130075 , goroutine count =  4
go task =  130076 , goroutine count =  4
go task =  130077 , goroutine count =  4
go task =  130078 , goroutine count =  4
go task =  130079 , goroutine count =  4
go task =  130080 , goroutine count =  4
go task =  130081 , goroutine count =  4
go task =  130082 , goroutine count =  4
go task =  130083 , goroutine count =  4
go task =  130084 , goroutine count =  4
go task =  130085 , goroutine count =  4
go task =  130086 , goroutine count =  4
go task =  130087 , goroutine count =  4
go task =  130088 , goroutine count =  4
go task =  130089 , goroutine count =  4
```

```
go task =  130090 , goroutine count =  4
go task =  130091 , goroutine count =  4
go task =  130092 , goroutine count =  4
go task =  130093 , goroutine count =  4
...
```

执行流程大致如下，这里实际上是将任务的发送和执行做了业务上的分离。使得消息出去，输入SendTask的频率可设置、执行Goroutine的数量也可设置。也就是既控制输入(生产)，又控制输出(消费)。使得可控更加灵活。这也是很多Go框架的Worker工作池的最初设计思想理念。



以上便是目前有关限定goroutine基础设计思路。

参考：

http://team.jiunile.com/blog/2019/09/go-control-goroutine-number.html

https://www.joyk.com/dig/detail/1547976674512705

单点Server的N种并发模型汇总

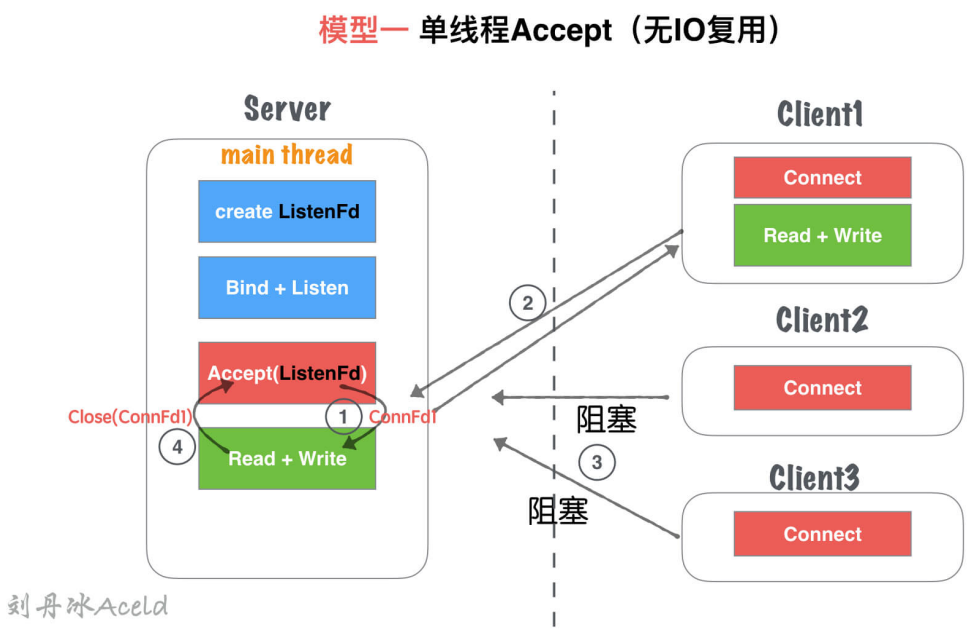
本文主要介绍常见的Server的并发模型，这些模型与编程语言本身无关，有的编程语言可能在语法上直接透明了模型本质，所以开发者没必要一定要基于模型去编写，只是需要知道和了解并发模型的构成和特点即可。

那么在了解并发模型之前，我们需要两个必备的前置知识：

- socket网络编程
- 多路IO复用机制
- 多线程/多进程等并发编程理论

模型一、单线程Accept（无IO复用）

(1) 模型结构图



(2) 模型分析

① 主线程 `main thread` 执行阻塞Accept，每次客户端Connect链接过来， `main thread` 中accept响应并建立连接

② 创建链接成功，得到 `Connfd1` 套接字后，依然在 `main thread` 串行处理套接字读写，并处理业务。

③ 在②处理业务中，如果有新客户端 `Connect` 过来， `Server` 无响应，直到当前套接字全部业务处理完毕。

④ 当前客户端处理完后，完毕链接，处理下一个客户端请求。

(3) 优缺点

优点：

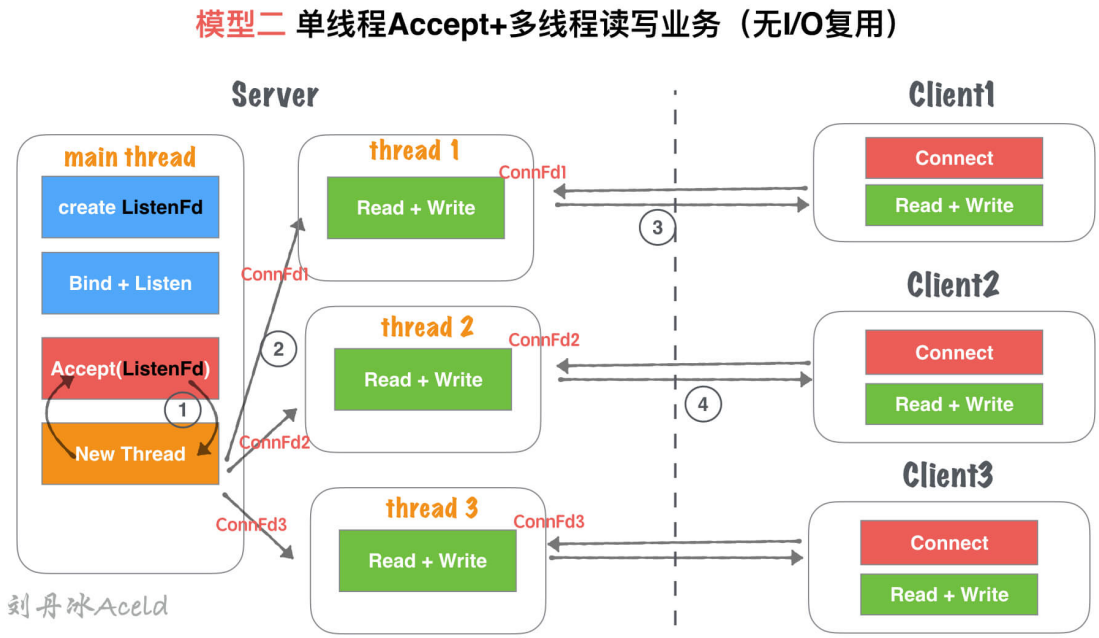
- socket编程流程清晰且简单，适合学习使用，了解socket基本编程流程。

缺点：

- 该模型并非并发模型，是串行的服务器，同一时刻，监听并响应最大的网络请求量为 `1` 。即并发量为 `1` 。
- 仅适合学习基本socket编程，不适合任何服务器Server构建。

模型二、单线程Accept+多线程读写业务（无IO复用）

(1) 模型结构图



(2) 模型分析

① 主线程 `main thread` 执行阻塞Accept，每次客户端Connect链接过来， `main thread` 中accept响应并建立连接

② 创建链接成功，得到 `Connfd1` 套接字后，创建一个新线程 `thread1` 用来处理客户端的读写业务。 `main thead` 依然回到 `Accept` 阻塞等待新客户端。

③ `thread1` 通过套接字 `Connfd1` 与客户端进行通信读写。

④ server在②处理业务中，如果有新客户端 `Connect` 过来， `main thread` 中 `Accept` 依然响应并建立连接，重复②过程。

(3) 优缺点

优点：

- 基于 `模型一：单线程Accept（无IO复用）` 支持了并发的特性。
- 使用灵活，一个客户端对应一个线程单独处理， `server` 处理业务内聚程度高，客户端无论如何写，服务端均会有一个线程做资源响应。

缺点：

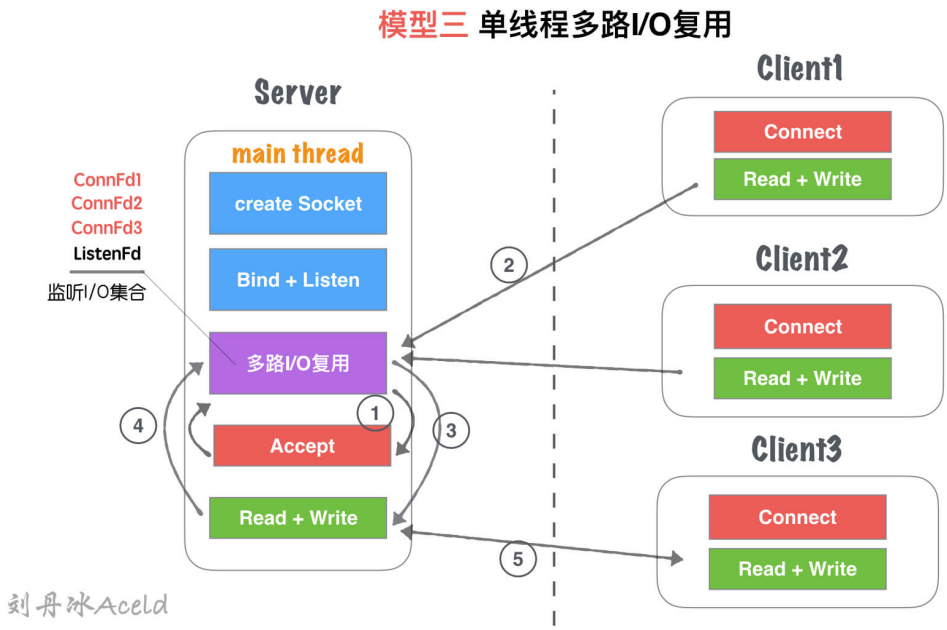
- 随着客户端的数量增多，需要开辟的线程也增加，客户端与server线程数量 `1:1` 正比关系，一次对于高并发场景，线程数量收到硬件上限瓶颈。

- 对于长链接，客户端一旦无业务读写，只要不关闭，server的对应线程依然需要保持连接(心跳、健康监测等机制)，占用连接资源和线程开销资源浪费。
- 仅适合客户端数量不大，并且数量可控的场景使用。

仅适合学习基本socket编程，不适合任何服务器Server构建。

模型三、单线程多路IO复用

(1) 模型结构图



(2) 模型分析

① 主线程 `main thread` 创建 `listenFd` 之后，采用多路I/O复用机制(如:select、epoll)进行IO状态阻塞监控。有 `Client1` 客户端 `Connect` 请求，I/O复用机制检测到 `ListenFd` 触发读事件，则进行 `Accept` 建立连接，并将新生成的 `connFd1` 加入到 `监听I/O集合` 中。

② `Client1` 再次进行正常读写业务请求， `main thread` 的 `多路I/O复用机制` 阻塞返回，会触该套接字的读/写事件等。

③ 对于 `Client1` 的读写业务，Server依然在 `main thread` 执行流程提继续执行，此时如果有新的客户端 `Connect` 链接请求过来，Server将没有即时响应。

④ 等到Server处理完一个连接的 `Read+Write` 操作，继续回到 `多路I/O复用机制` 阻塞，其他链接过来重复 ②、③流程。

(3) 优缺点

优点：

- 单流程解决了可以同时监听多个客户端读写状态的模型，不需要 `1:1` 与客户端的线程数量关系。
- 多路I/O复用阻塞，非忙询状态，不浪费CPU资源， CPU利用率较高。

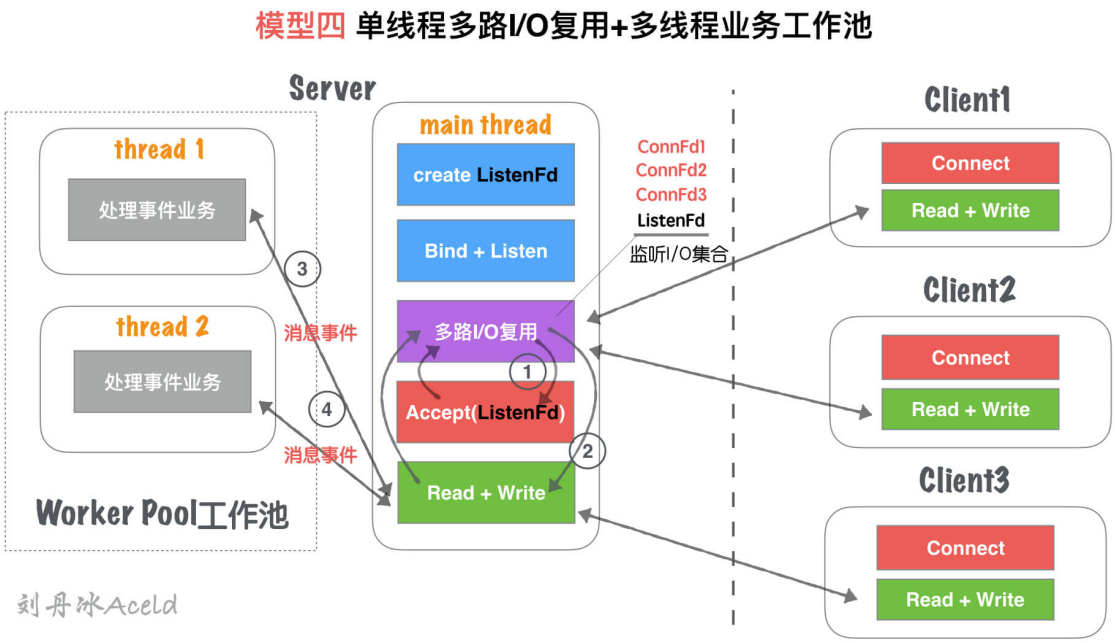
缺点：

- 虽然可以监听多个客户端的读写状态，但是同一时间内，只能处理一个客户端的读写操作，实际上读写的业务并发为1。

- 多客户端访问Server，业务为串行执行，大量请求会有排队延迟现象，如图中⑤所示，当 `Client3` 占据 `main thread` 流程时， `Client1,Client2` 流程卡在 `IO复用` 等待下次监听触发事件。

模型四、单线程多路IO复用+多线程读写业务(业务工作池)

(1) 模型结构图



(2) 模型分析

① 主线程 `main thread` 创建 `listenFd` 之后，采用多路I/O复用机制(如:select、epoll)进行IO状态阻塞监控。有 `Client1` 客户端 `Connect` 请求，I/O复用机制检测到 `ListenFd` 触发读事件，则进行 `Accept` 建立连接，并将新生成的 `connFd1` 加入到 `监听I/O集合` 中。

② 当 `connFd1` 有可读消息，触发读事件，并且进行读写消息

③ `main thread` 按照固定的协议读取消息，并且交给 `worker pool` 工作线程池， 工作线程池在server启动之前就已经开启固定数量的 `thread` ，里面的线程只处理消息业务，不进行套接字读写操作。

④ 工作池处理完业务，触发 `connFd1` 写事件，将回执客户端的消息通过 `main thead` 写给对方。

(3) 优缺点

优点：

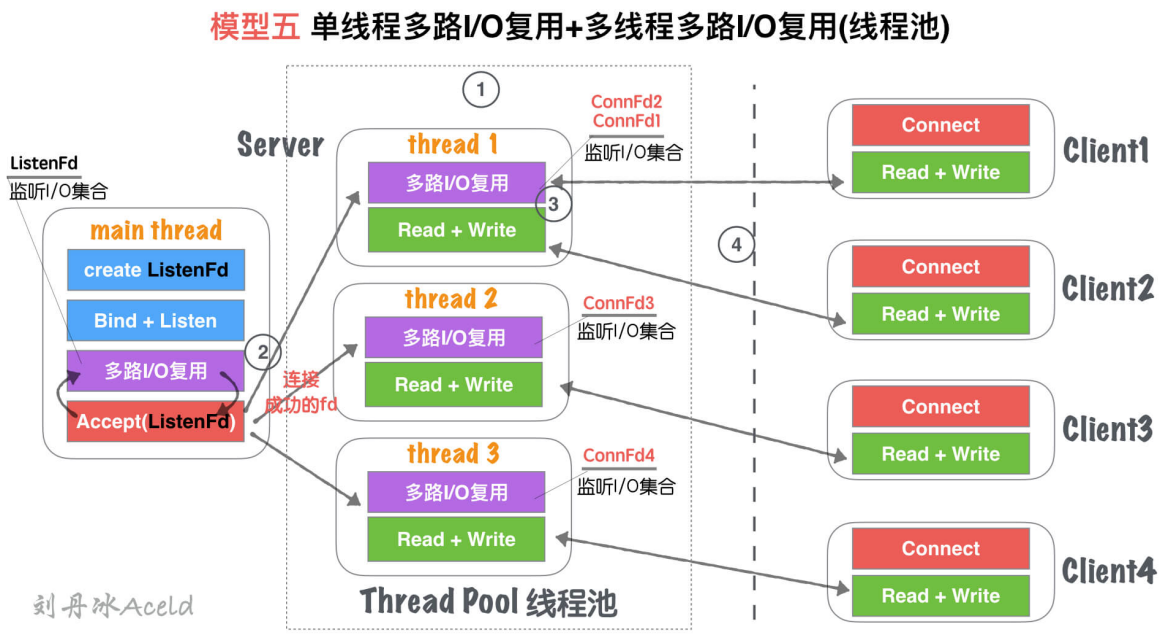
- 对于 `模型三` ，将业务处理部分，通过工作池分离出来，减少多客户端访问Server，业务为串行执行，大量请求会有排队延迟时间。
- 实际上读写的业务并发为1，但是业务流程并发为worker pool线程数量，加快了业务处理并行效率。

缺点：

- 读写依然为 `main thread` 单独处理，最高读写并行通道依然为1.
- 虽然多个worker线程处理业务，但是最后返回给客户端，依旧需要排队，因为出口还是 `main thread` 的 `Read + Write`

模型五、单线程IO复用+多线程IO复用(链接线程池)

(1) 模型结构图



(2) 模型分析

① Server在启动监听之前，开辟固定数量(N)的线程，用 `Thead Pool` 线程池管理

② 主线程 `main thread` 创建 `listenFd` 之后，采用多路I/O复用机制(如:select、epoll)进行IO状态阻塞监控。有 `Client1` 客户端 `Connect` 请求，I/O复用机制检测到 `ListenFd` 触发读事件，则进行 `Accept` 建立连接，并将新生成的 `connFd1` 分发给 `Thread Pool` 中的某个线程进行监听。

③ `Thread Pool` 中的每个 `thread` 都启动 `多路I/O复用机制(select、epoll)` ,用来监听 `main thread` 建立成功并且分发下来的socket套接字。

④ 如图， `thread` 监听 `ConnFd1、ConnFd2` ， `thread2` 监听 `ConnFd3` ， `thread3` 监听 `ConnFd4` . 当对应的 `ConnFd` 有读写事件，对应的线程处理该套接字的读写及业务。

(3) 优缺点

优点：

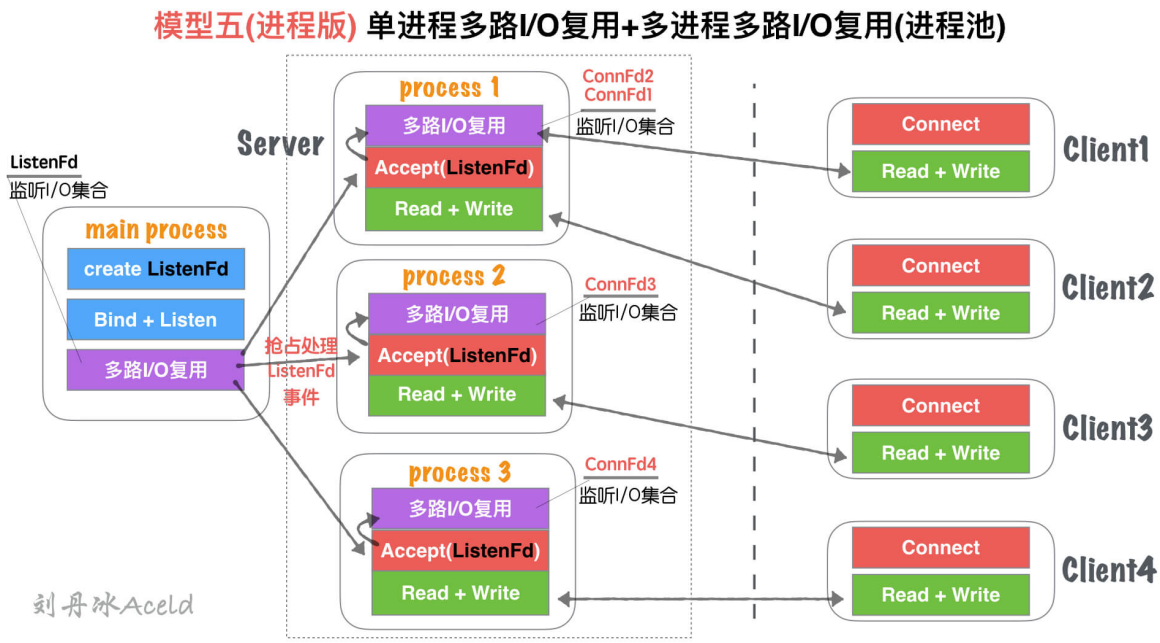
- 将 `main thread` 的单流程读写，分散到多线程完成，这样增加了同一时刻的读写并行通道，并行通道数量 `N` ， `N` 为线程池 `Thread` 数量。
- server同时监听的 `ConnFd套接字` 数量几乎成倍增大，之前的全部监控数量取决于 `main thread` 的 `多路I/O复用机制` 的最大限制***(select 默认为1024， epoll默认与内存大小相关，约3~6w不等)***，所以理论单点Server最高响应并发数量为 `N*(3~6W)` (`N` 为线程池 `Thread` 数量，建议与CPU核心成比例1:1)。
- 如果良好的线程池数量和CPU核心数适配，那么可以尝试CPU核心与Thread进行绑定，从而降低CPU的切换频率，提升每个 `Thread` 处理合理业务的效率，降低CPU切换成本开销。

缺点：

- 虽然监听的并发数量提升，但是最高读写并行通道依然为 `N` ，而且多个身处同一个Thread的客户端，会出现读写延迟现象，实际上每个 `Thread` 的模型特征与 `模型三：单线程多路IO复用` 一致。

模型五(进程版)、单进程多路I/O复用+多进程多路I/O复用(进程池)

(1) 模型结构图



(2) 模型分析

与 `五、单线程IO复用+多线程IO复用(链接线程池)` 无大差异。

不同处

- 进程和线程的内存布局不同导致， `main process` (主进程)不再进行 `Accept` 操作，而是将 `Accept` 过程分散到各个 `子进程(process)` 中.
- 进程的特性，资源独立，所以 `main process` 如果Accept成功的fd，其他进程无法共享资源，所以需要各子进程自行Accept创建链接
- `main process` 只是监听 `ListenFd` 状态，一旦触发读事件(有新连接请求). 通过一些IPC(进程间通信：如信号、共享内存、管道)等, 让各自子进程 `Process` 竞争 `Accept` 完成链接建立，并各自监听。

(3) 优缺点

与 `五、单线程IO复用+多线程IO复用(链接线程池)` 无大差异。

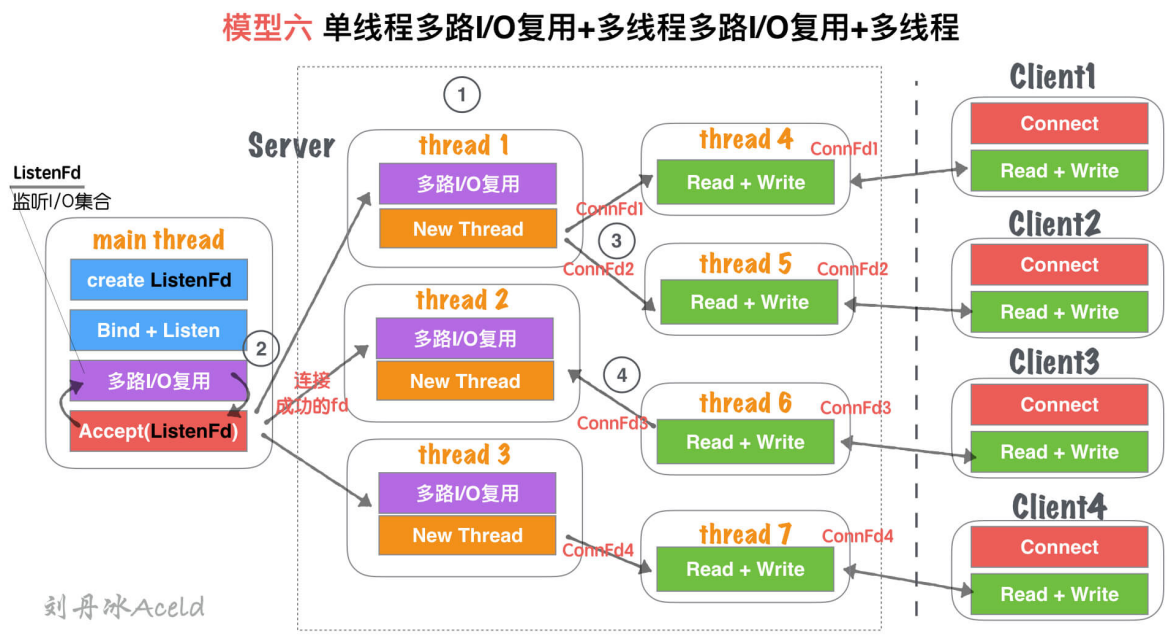
不同处:

多进程内存资源空间占用稍微大一些

多进程模型安全稳定型较强，这也是因为各自进程互不干扰的特点导致。

模型六、单线程多路I/O复用+多线程多路I/O复用+多线程

(1) 模型结构图



(2) 模型分析

① Server在启动监听之前，开辟固定数量(N)的线程，用 `Thead Pool` 线程池管理

② 主线程 `main thread` 创建 `listenFd` 之后，采用多路I/O复用机制(如:select、epoll)进行IO状态阻塞监控。有 `Client1` 客户端 `Connect` 请求，I/O复用机制检测到 `ListenFd` 触发读事件，则进行 `Accept` 建立连接，并将新生成的 `connFd1` 分发给 `Thread Pool` 中的某个线程进行监听。

③ `Thread Pool` 中的每个 `thread` 都启动 `多路I/O复用机制(select、epoll)` ,用来监听 `main thread` 建立成功并且分发下来的socket套接字。一旦其中某个被监听的客户端套接字触发 `I/O读写事件` ,那么，会立刻开辟一个新线程来处理 `I/O读写` 业务。

④ 但某个读写线程完成当前读写业务，如果当前套接字没有被关闭，那么将当前客户端套接字 `如:ConnFd3` 重新加回线程池的监控线程中，同时自身线程自我销毁。

(3) 优缺点

优点：

- 在 `模型五、单线程IO复用+多线程IO复用(链接线程池)` 基础上，除了能够保证同时响应的 `最高并发数` ，又能解决 `读写并行通道` 局限的问题。
- 同一时刻的读写并行通道，达到 `最大化极限` ，一个客户端可以对应一个单独执行流程处理读写业务，读写并行通道与客户端数量 `1:1` 关系。

缺点：

- 该模型过于理想化，因为要求CPU核心数量足够大。
- 如果硬件CPU数量可数(目前的硬件情况)，那么该模型将造成大量的CPU切换成本浪费。因为为了保证读写并行通道与客户端 `1:1` 的关系，那么Server需要开辟的 `Thread` 数量就与客户端一致，那么线程池中做 `多路I/O复用` 的监听线程池绑定CPU数量将变得毫无意义。
- 如果每个临时的读写 `Thread` 都能够绑定一个单独的CPU，那么此模型将是最优模型。但是目前CPU的数量无法与客户端的数量达到一个量级，目前甚至差的不是几个量级的事。

总结

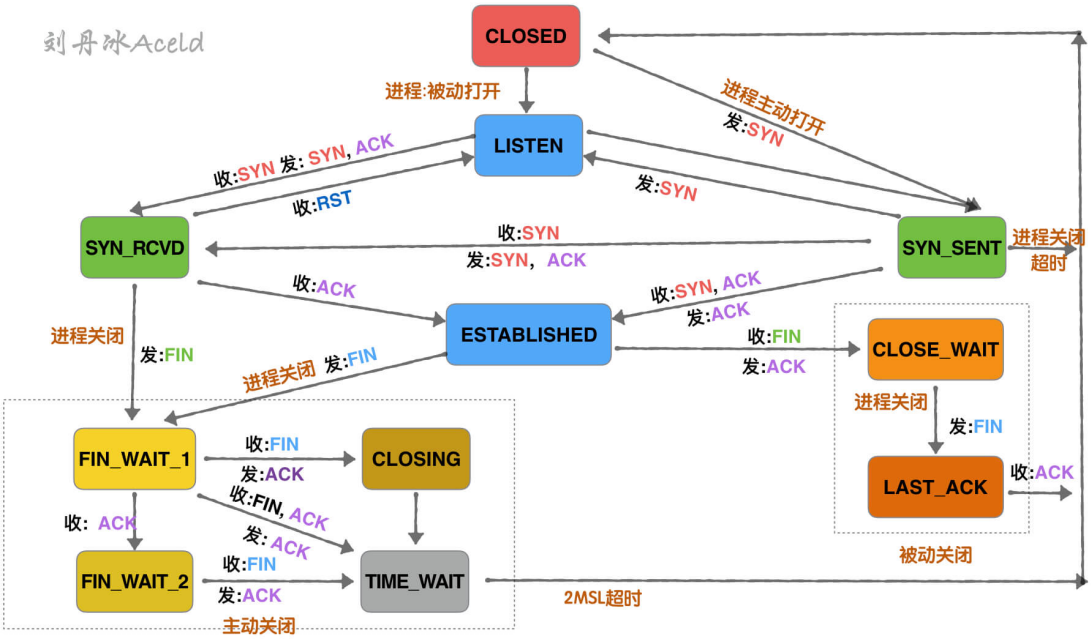
综上，我们整理了7中Server的服务器处理结构模型，每个模型都有各自的特点和优势，那么对于多少应付高并发和高CPU利用率的模型，目前多数采用的是模型五(或模型五进程版，如Nginx就是类似模型五进程版的改版)。

至于并发模型并非设计的约复杂越好，也不是线程开辟的越多越好，我们要考虑硬件的利用与和切换成本的开销。模型六设计就极为复杂，线程较多，但以当今的硬件能力无法支撑，反倒导致该模型性能极差。所以对于不同的业务场景也要选择适合的模型构建，并不是一定固定就要使用某个来应用。

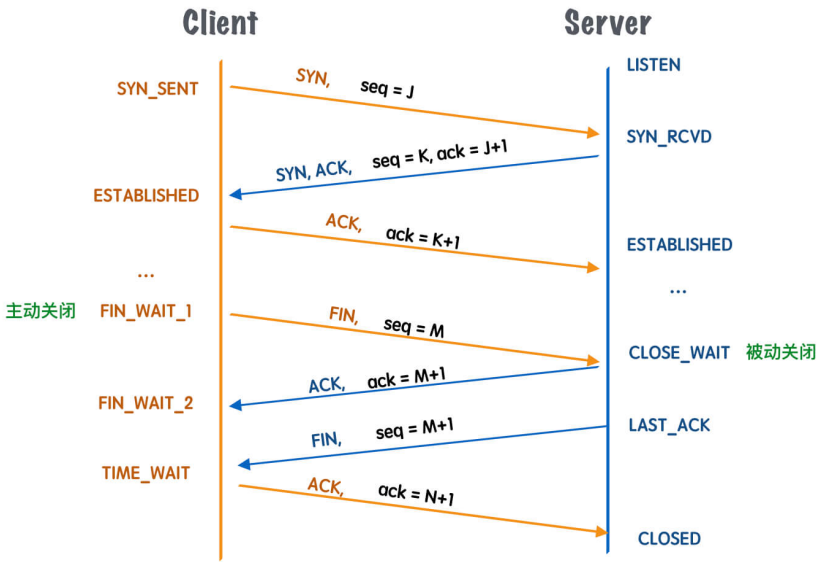
TCP中TIME_WAIT状态意义详解

一、何为TIME_WAIT?

我们在日常做服务器的研发中、或者面试网络部分知识的时候，会经常问到TIME_WAIT这个词，这个词作为服务端的开发者尤为重要。TIME_WAIT是TCP协议中断开连接所经历的一种状态。



上图是TCP连接的状态转换，包括了一些触发条件，如果不是很直观，可以对比看下面的简图。



这里面作为主动关闭的一方(Client)出现了 `TIME_WAIT` 状态，目的是告诉Server端，**自己没有需要发送的数据**，但是它仍然**保持了接收对方数据的能力**，一个常见的关闭连接过程如下：

1、当客户端没有待发送的数据时，它会向服务端发送 `FIN` 消息，发送消息后会进入 `FIN_WAIT_1` 状态；

2、服务端接收到客户端的 `FIN` 消息后，会进入 `CLOSE_WAIT` 状态并向客户端发送 `ACK` 消息，客户端接收到 `ACK` 消息时会进入 `FIN_WAIT_2` 状态；

3、当服务端没有待发送的数据时，服务端会向客户端发送 `FIN` 消息；

4、客户端接收到 `FIN` 消息后，会进入 `TIME_WAIT` 状态并向服务端发送 `ACK` 消息，服务端收到后会进入 `CLOSED` 状态；

5、客户端等待**两个最大数据段生命周期**（Maximum segment lifetime，MSL）的时间后也会进入 `CLOSED` 状态；

二、为什么需要TIME_WAIT

TIME_WAIT一定是发生在主动关闭一方

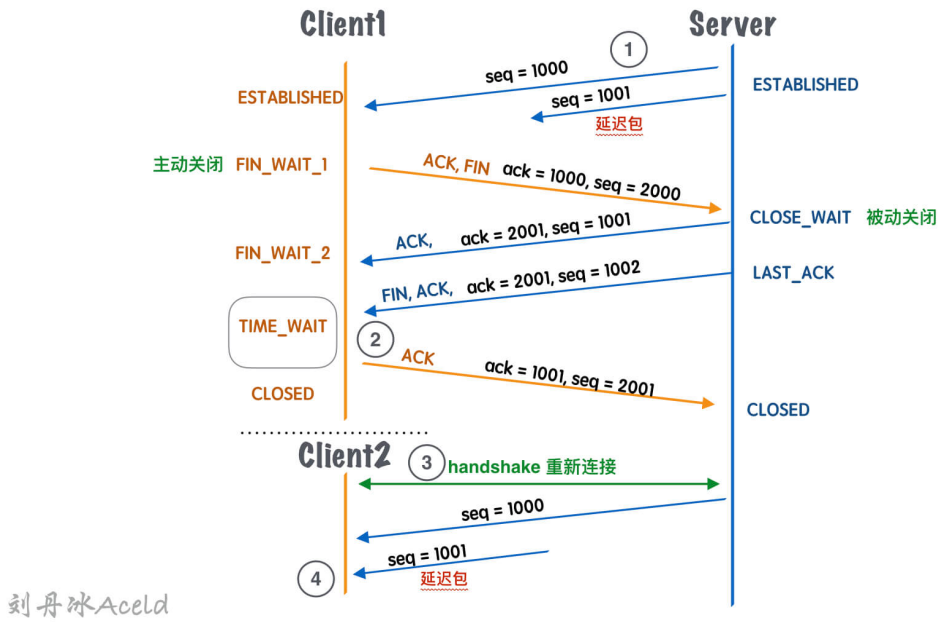
被动关闭一方，会直接进入 `CLOSED` 状态，而主动关闭一方需要等待2*MSL时间才会最终关闭。

原因：

1、防止被动关闭方的延迟数据被人窃取

2、防止被动关闭方没有收到最后的ACK

原因一：防止被动关闭方的延迟数据被人窃取



如上图所示，

1、在①中，服务端发送 `seq=1001` 的消息，由于网络延迟或其他原因，没有及时到达 `Client1` 客户端，导致整个包一直存留在网络环境的传输过程中。

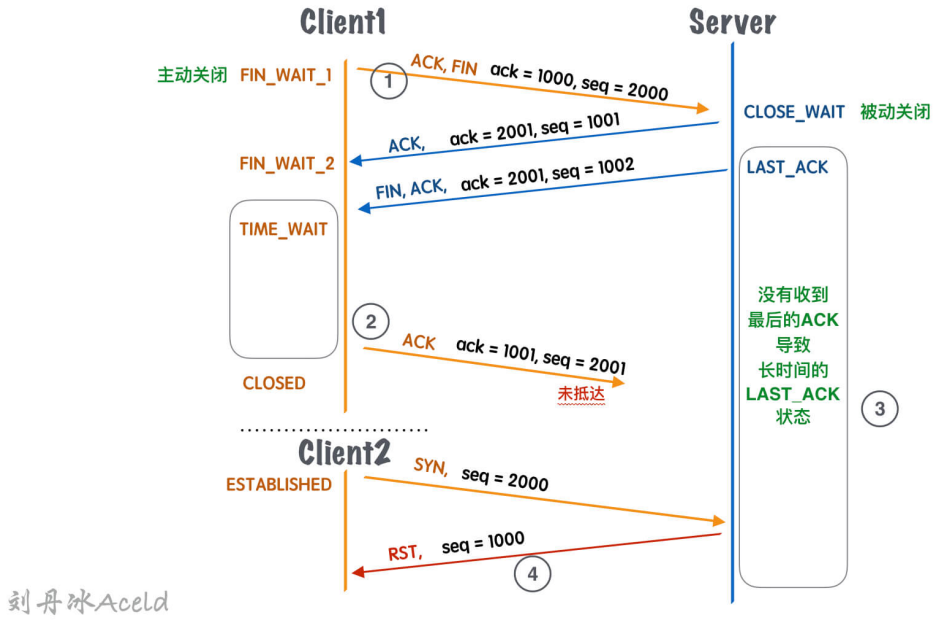
2、在②中，`Client1` 收到server的 `FIN` 包之后，变成了 `TIME_WAIT` 状态，这里假设 `TIME_WAIT` 等待的时间很短暂，那么，还没等之前的那个延迟包 `seq=1001` 到来，就回复给了 `Server` 最后一个 `ACK` 包。那么 `Server` 就会变成 `CLOSED` 状态。

3、在③中，相同的端口号的 `Client2` 的TCP链接被重用后

4、在④中，`seq=1001`的延迟包消息才发送给客户端，而这个延迟的消息却被`Client2`正常接收，主要就会给Client2带来严重的问题。所以`TIME_WAIT`不要轻易的调整，或者缩小时间，可能就会出现这种问题。

原因二：防止被动关闭方没有收到最后的ACK

该作用就是等待足够长的时间以确定远程的TCP链接收到了其发出的终止链接消息`FIN`包的回执消息`ACK`包。



如上图所示：

1、在①中，`CLient1`端主动发起关闭链接，`Server`针对`Client1`的`FIN`回执了`ACK`包，然后接着发送了自己的`FIN`包，等待`Client1`回执最终的`ACK`包。

2、在②中，这里假设`TIME_WAIT`的时间不足够充分，当`Server`还没有收到`ACK`消息时，`Client1`就主动变成`CLOSED`状态。

3、在③中，由于`Server`一直没有等到自己`FIN`包的`ACK`应答包，导致一直处于`LAST_ACK`状态。

4、在④中，因为 服务端因为没有收到`ACK`消息，当`Client2`重新与`Server`建立TCP链接，认为当前连接是合法的，`CLient2`重新发送`SYN`消息请求握手时会收到`Server`的`RST`消息，连接建立的过程就会被终止。

所以，我们在默认情况下，如果客户端等待足够长的时间就会遇到以下两种情况：

1. 服务端正常收到了`ACK`消息并关闭当前 TCP 连接；
2. 服务端没有收到`ACK`消息，重新发送`FIN`关闭连接并等待新的`ACK`消息；

只要客户端等待 2 MSL 的时间，客户端和服务端之间的连接就会正常关闭，新创建的 TCP 连接收到影响的概率也微乎其微，保证了数据传输的可靠性。

动态保活Worker工作池设计

一、我们如何知道一个Goroutine已经死亡？

实际上，Go语言并没有给我们暴露如何知道一个Goroutine是否存在的接口，如果要证明一个Go是否存在，可以在子Goroutine的业务中，定期写向一个keep live的Channel，然后主Goroutine来发现当前子Go的状态。Go语言在对于Go和Go之间没有像进程和线程一样有强烈的父子、兄弟等关系，每个Go实际上对于调度器都是一个独立的，平等的执行流程。

> PS: 如果你是监控子线程、子进程的死亡状态，就没有这么简单了，这里也要感谢go的调度器给我们提供的方便，我们既然用Go，就要基于Go的调度器来实现该模式。

那么，我们如何做到一个Goroutine已经死亡了呢？

子Goroutine

可以通过给一个被监控的Goroutine添加一个 `defer` ，然后 `recover()` 捕获到当前Goroutine的异常状态，最后给主Goroutine发送一个死亡信号，通过 `Channel` 。

主Goroutine

在 `主Goroutine` 上，从这个 `Channel` 读取内容，当读到内容时，就重启这个 `子Goroutine` ，当然 `主Goroutine` 需要记录 `子Goroutine` 的 `ID` ，这样也就可以针对性的启动了。

二、代码实现

我们这里以一个工作池的场景来对上述方式进行实现。

`WorkerManager` 作为 `主Goroutine` ， `worker` 作为子 `Goroutine`

> WorkerManager

```
type WorkerManager struct {
   //用来监控Worker是否已经死亡的缓冲Channel
   workerChan chan *worker
   // 一共要监控的worker数量
   nWorkers int
}

//创建一个WorkerManager对象
func NewWorkerManager(nworkers int) *WorkerManager {
   return &WorkerManager{
      nWorkers:nworkers,
      workerChan: make(chan *worker, nworkers),
   }
}

//启动worker池，并为每个Worker分配一个ID，让每个Worker进行工作
func (wm *WorkerManager)StartWorkerPool() {
   //开启一定数量的Worker
   for i := 0; i < wm.nWorkers; i++ {
      i := i
      wk := &worker{id: i}
      go wk.work(wm.workerChan)
   }
```

```
    //启动保活监控
    wm.KeepLiveWorkers()
}

//保活监控workers
func (wm *WorkerManager) KeepLiveWorkers() {
    //如果有worker已经死亡 workChan会得到具体死亡的worker然后 打出异常，然后重启
    for wk := range wm.workerChan {
        // log the error
        fmt.Printf("Worker %d stopped with err: [%v] \n", wk.id, wk.err)
        // reset err
        wk.err = nil
        // 当前这个wk已经死亡了，需要重新启动他的业务
        go wk.work(wm.workerChan)
    }
}
```

> worker

```
type worker struct {
    id  int
    err error
}

func (wk *worker) work(workerChan chan<- *worker) (err error) {
    // 任何Goroutine只要异常退出或者正常退出 都会调用defer 函数，所以在defer中想WorkerManager的WorkChan发送通知
    defer func() {
        //捕获异常信息，防止panic直接退出
        if r := recover(); r != nil {
            if err, ok := r.(error); ok {
                wk.err = err
            } else {
                wk.err = fmt.Errorf("Panic happened with [%v]", r)
            }
        } else {
            wk.err = err
        }

        //通知 主 Goroutine，当前子Goroutine已经死亡
        workerChan <- wk
    }()

    // do something
    fmt.Println("Start Worker...ID = ", wk.id)

    // 每个worker睡眠一定时间之后，panic退出或者 Goexit()退出
    for i := 0; i < 5; i++ {
        time.Sleep(time.Second*1)
    }

    panic("worker panic..")
    //runtime.Goexit()

    return err
}
```

三、测试

> main

```
func main() {
    wm := NewWorkerManager(10)

    wm.StartWorkerPool()
}
```

结果：

```
$ go run workmanager.go
Start Worker...ID =  2
Start Worker...ID =  1
Start Worker...ID =  3
Start Worker...ID =  4
Start Worker...ID =  7
Start Worker...ID =  6
Start Worker...ID =  8
Start Worker...ID =  9
Start Worker...ID =  5
Start Worker...ID =  0
Worker 9 stopped with err: [Panic happened with [worker panic..]]
Worker 1 stopped with err: [Panic happened with [worker panic..]]
Worker 0 stopped with err: [Panic happened with [worker panic..]]
Start Worker...ID =  9
Start Worker...ID =  1
Worker 2 stopped with err: [Panic happened with [worker panic..]]
Worker 5 stopped with err: [Panic happened with [worker panic..]]
Worker 4 stopped with err: [Panic happened with [worker panic..]]
Start Worker...ID =  0
Start Worker...ID =  2
Start Worker...ID =  4
Start Worker...ID =  5
Worker 7 stopped with err: [Panic happened with [worker panic..]]
Worker 8 stopped with err: [Panic happened with [worker panic..]]
Worker 6 stopped with err: [Panic happened with [worker panic..]]
Worker 3 stopped with err: [Panic happened with [worker panic..]]
Start Worker...ID =  3
Start Worker...ID =  6
Start Worker...ID =  8
Start Worker...ID =  7
...
...
```

我们会发现，无论子Goroutine是因为 panic()异常退出，还是Goexit()退出，都会被主Goroutine监听到并且重启。主要我们就能够起到保活的功能了. 当然如果线程死亡？进程死亡？我们如何保证？ 大家不用担心，我们用Go开发实际上是基于Go的调度器来开发的，进程、线程级别的死亡，会导致调度器死亡，那么我们的全部基础框架都将会塌陷。那么就要看线程、进程如何保活啦，不在我们Go开发的范畴之内了。