Как устроена память в Go

#memory #stack #heap #go

Если есть указатель, это не значит, что переменная в куче. Будет ли она в куче решает `escape analysis`.

Он в свою очередь устроен через граф весов. Если кратко, то алгоритм представлен ниже:

1. Вначале строится граф
2. Потом происходит проверка на размер переменных. Дело в том, что у каждого типа переменной есть свой лимит на размер.
3. Наконец, обход графа (walkAll) и выдвижение решения того, уходит ли в хип переменная или нет.

Аллокатор хипа

Он основан на TCMalloc (Thread-Caching Allocator). Главная идея заключается в «слоистом» представлении памяти и в отдельных блоках памяти для каждого треда, в которые он обращается без lock.

Вместо частых малых запросов памяти, go просит сразу большой кусок, называемый ареной. Табличка размера арены для каждой ОС:

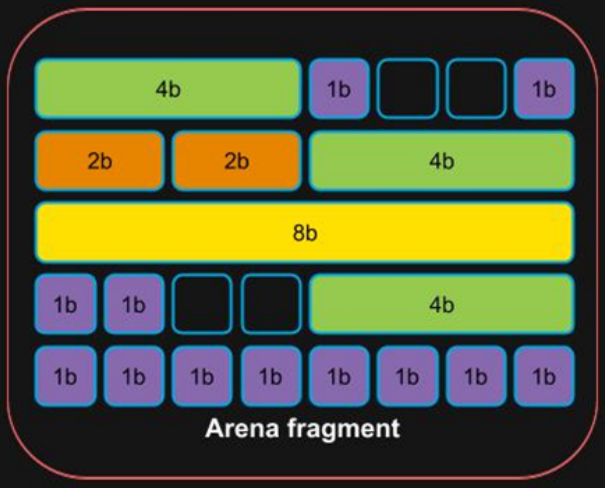
Go allocator

Platform	Arena size
*/64-bit	64Mb
windows/64-bit	4Mb
ios/arm64	4Mb
*/32-bit	4Mb
*/mips(le)	4Mb

Но арена это не просто кусок памяти, она делится на странички. Всем этим добром управляет структура в аллокаторе, mheap.

Однако у такого подхода есть проблема фрагментации,когда переменная весом в n байта не может быть записана:

Go allocator. Фрагментация



То есть у нас образовалась внешняя фрагментация.

Для решения данной проблемы был введен Pool объектовю

Pool объектов

У нас есть арена. На этой арене выделим пулы для разных размеров переменных. То есть, если есть переменные в 16b, сразу выделим пул в 16b. Если есть переменная в 32b, сразу выделим пул в 32b, и т.д. Такой пул называется в аллокаторе mspan.

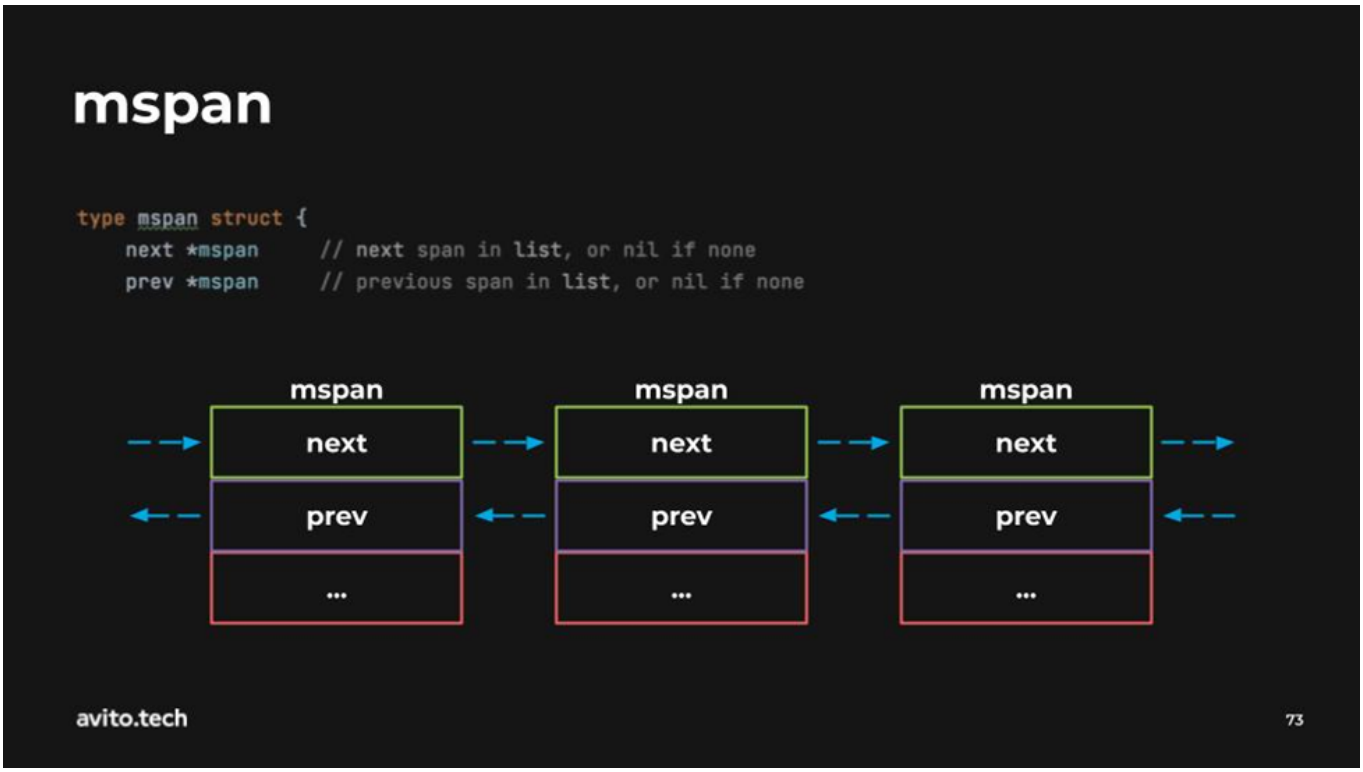
mspan — это минимальный юнит в аллокаторе, который уже имеет метаинформацию о переменных. У mspan есть поле spanClass, которое хранит тип класса для данного пула. Класс определяет для какого размера создан этот пул. Mspan 1-го класса означает, что он создан для переменных 8b, а если 3-го класса, то для переменных 24b.

Вы можете сами посмотреть какие есть классы так как разработчиками сгенерирована табличка со всеми этим классами и информацией по ним. Она находится в пакете runtime/sizeclasses.go

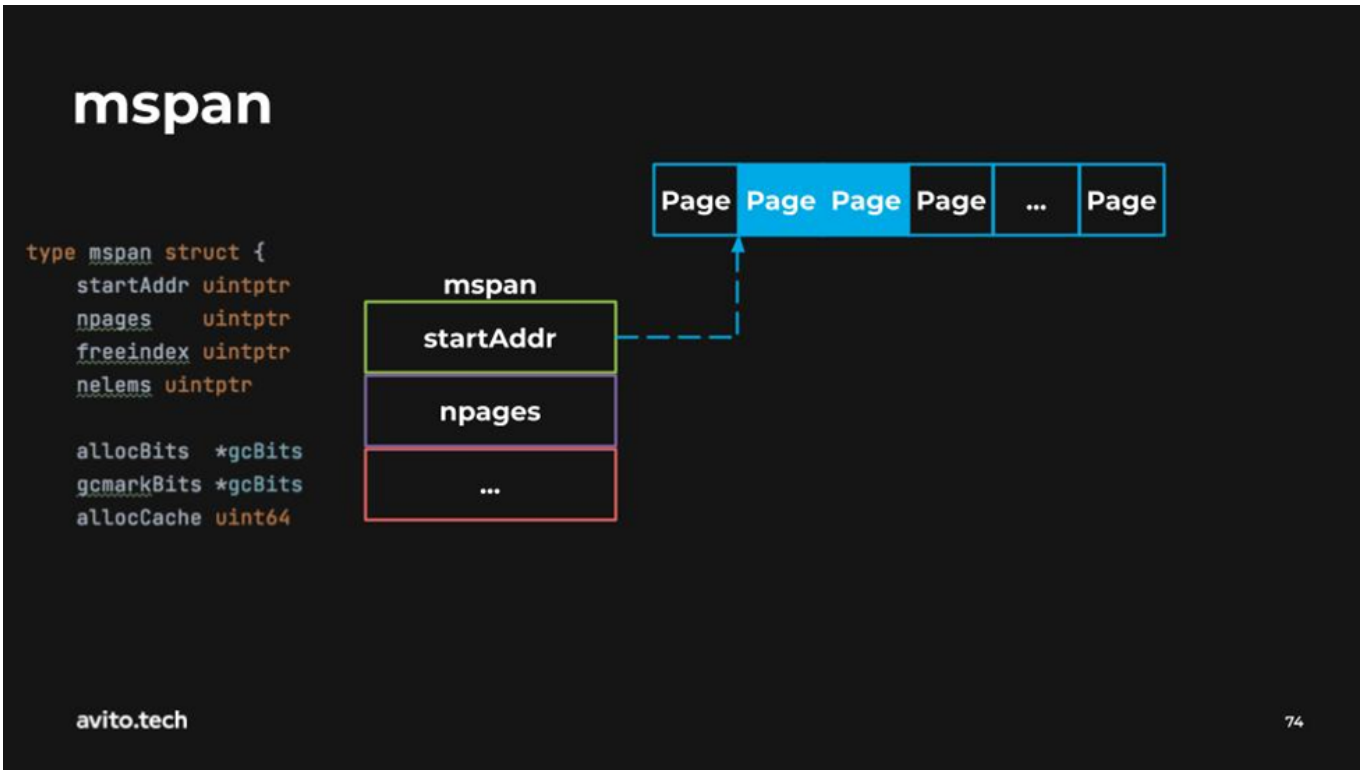
Третий столбец говорит о том, сколько страниц тратится на данный пул. Видно, что для mspan класса 1 тратится 1 страница, потому что в этом третьем столбце написано 8192 байта. В Golang всего 67 классов. Для 67-го класса тратится 4 страницы.

Четвертый столбец (objects) говорит, сколько, максимум, объектов помещается в этот пул.

Естественно, эти пулы заканчиваются, и когда это происходит, создаётся ещё один пул (ещё один mspan). Все эти mspan соединены друг с другом в двусторонний связанный список.



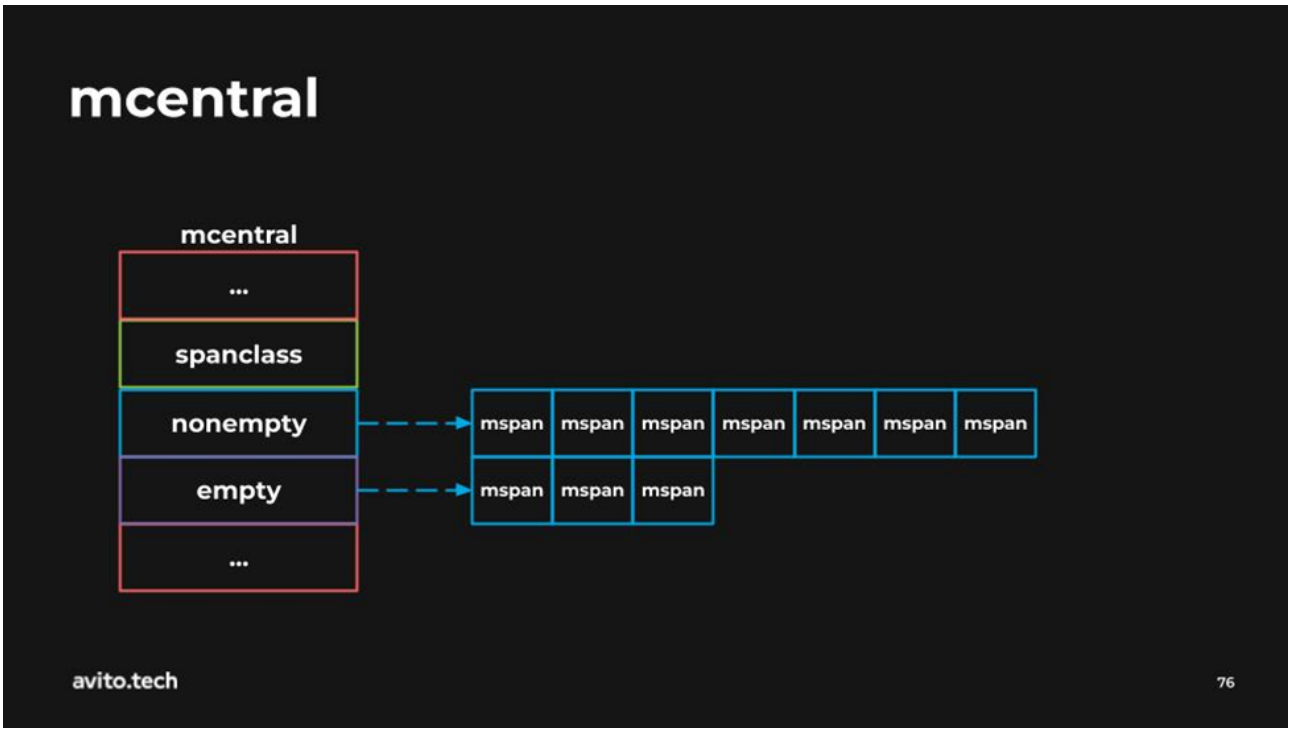
Также из интересного — поле startAddr, которое говорит, где именно на арене начинается данный пул.



Кроме этого здесь есть:

1. Поле npages. Оно показывает из какого количества страниц состоит данный пул.
2. Поле freeindex. Оно нужно для того, чтобы моментально находить свободный блок в данном пуле.

3. Поле nelems. Оно говорит о том, сколько всего элементов в данном пуле
4. Также есть интересное поле allocBits. Оно показывает свободные ячейки в данном пуле. Нужно для поиска и для garbage collector.
   Для управления двустороннего связанного списка должны быть центральные структуры и они есть. За это отвечает структура mcentral. Один mcentral закрепляется за одним классом. Соответственно, всего будет 67 mcentral, так как всего 67 классов.

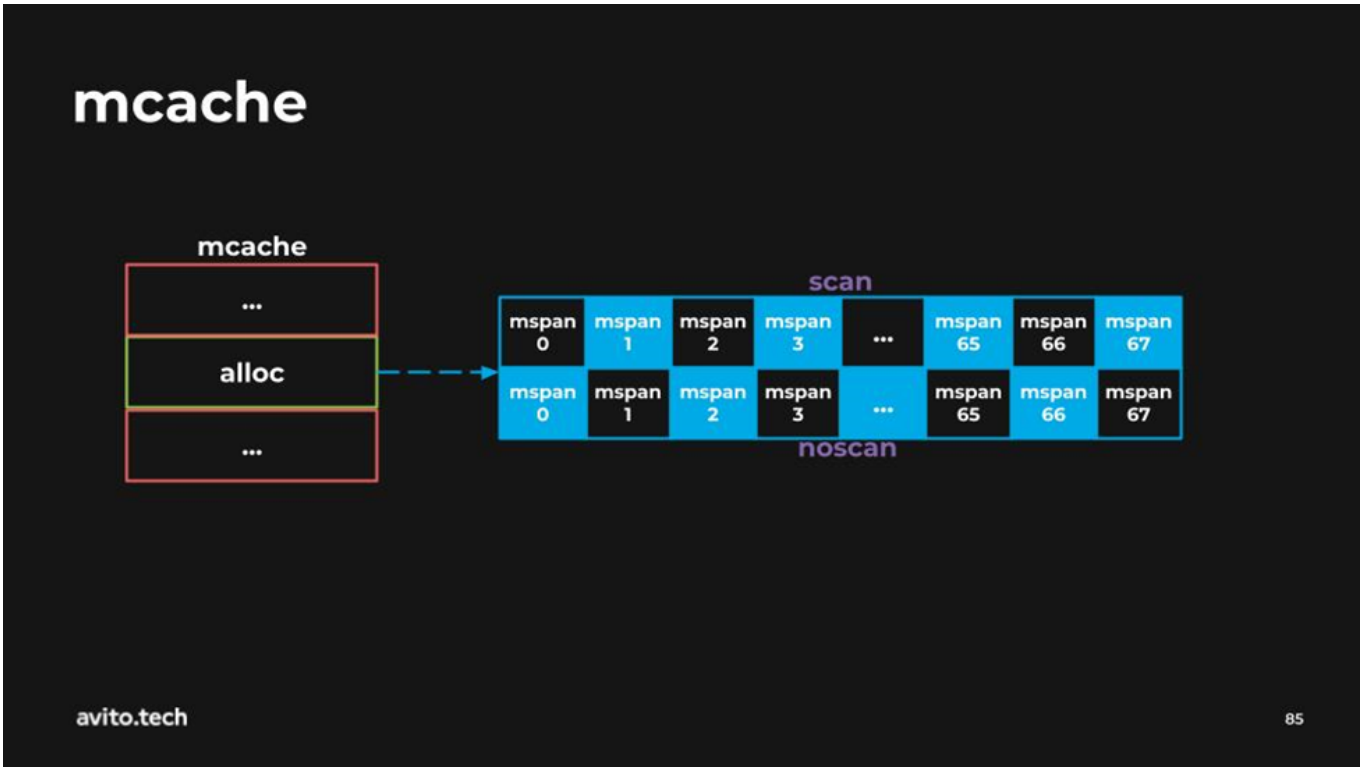


Go Concurrency model. Работа с несколькими тредами

Что, если у нас несколько тредов и один блок памяти? Если треды начнут запрашивать память, мы получим race condition. Как с этим можно бороться?

- Вводом лока и кэша.

Кэш создается для каждого треда, за это отвечает структура mcache.



Подводя итоги

Есть три неизменяемых правила выделения памяти от версии к версии Golang. У вас 100% выделится значение на хипе, если:

1. Возврат результата происходит по ссылке;
2. Значение передается в аргумент типа interface{} — аргумент fmt.Println;
3. Размер значения переменной превышает лимиты стека.