# Go sync.Pool и механика, лежащая в его основе

ноября 11, 2024 · 19 минут · German Gorelkin



---

Перевод/заметки Go sync.Pool and the Mechanics Behind It

---

`sync.Pool` хорошо подходит для работы с byte buffers или слайсами.

Он часто используется в стандартной библиотеке. Например, в пакете `encoding/json`:

```
package json

var encodeStatePool sync.Pool

// An encodeState encodes JSON into a bytes.Buffer.
type encodeState struct {
    bytes.Buffer // accumulated output

    ptrLevel uint
    ptrSeen  map[any]struct{}
}
```

В данном случае `sync.Pool` используется для повторного использования объектов `*encodeState`, которые обрабатывают процесс кодирования JSON в `bytes.Buffer`.

Вместо того чтобы просто выбрасывать такие объекты после каждого использования, что только прибавит работы сборщику мусора, мы складываем их в пул ( `sync.Pool` ). В следующий раз, когда они нам понадобится, мы просто возьмем их из пула, вместо того чтобы создавать новые с нуля.

В пакете `net/http` вы также найдете несколько экземпляров `sync.Pool`, которые используются для оптимизации операций ввода-вывода:

```
package http

var (
    bufioReaderPool   sync.Pool
    bufioWriter2kPool sync.Pool
    bufioWriter4kPool sync.Pool
)
```

Когда сервер читает тело запроса или пишет ответ, он может быстро извлечь из этих пулов заранее созданный reader или writer, минуя дополнительные аллокации. Кроме того, `*bufioWriter2kPool` и `*bufioWriter4kPool`, настроенны для разных сценариев записи.

```
func bufioWriterPool(size int) *sync.Pool {
    switch size {
    case 2 << 10:
        return &bufioWriter2kPool
    case 4 << 10:
        return &bufioWriter4kPool
    }
    return nil
}
```

## Что такое `sync.Pool`?

`sync.Pool` - это место, где вы можете хранить временные объекты для последующего использования.

Но важно помнить: **вы не контролируете, сколько объектов останется в пуле, и все, что вы туда поместите, может быть удалено в любой момент без предупреждения**.

Плюс в том, что пул создан как потокобезопасный, поэтому несколько горутин могут использовать его одновременно. Это неудивительно, учитывая, что он является частью пакета синхронизации.

> *Но зачем мы используем объекты повторно?*

Когда одновременно выполняется множество горутин, им часто требуются похожие объекты. Представьте себе, что `go f()` выполняется одновременно много раз.

Если каждая горутина создает собственные объекты, использование памяти может быстро увеличиться, и это создает нагрузку на сборщик мусора, поскольку ему приходится очищать все эти объекты, когда они больше не нужны.

*Такая ситуация создает цикл, в котором высокий параллелизм приводит к высокому использованию памяти*, что замедляет работу сборщика мусора. `sync.Pool` решает эту проблему.

```
type Object struct {
```

```
    Data []byte
}

var pool sync.Pool = sync.Pool{
    New: func() any {
        return &Object{
            Data: make([]byte, 0, 1024),
        }
    },
}
```

Чтобы создать пул, вы можете определить функцию `New()`, которая возвращает новый объект, когда пул пуст. Эта функция необязательна, без нее пул просто вернет `nil`, если он пуст.

В приведенном выше фрагменте цель состоит в том, чтобы повторно использовать экземпляр структуры `Object`, а именно слайс внутри нее.

Например, если в процессе использования слайс вырастет до 8192 байт, вы можете сбросить его длину до нуля, прежде чем поместить его обратно в пул. Базовый массив по-прежнему имеет емкость 8192, поэтому в следующий раз, когда он вам понадобится, эти 8192 байта будут готовы к повторному использованию.

```
func (o *Object) Reset() {
    o.Data = o.Data[:0]
}

func main() {
    testObject := pool.Get().(*Object)

    // do something with testObject

    testObject.Reset()
    pool.Put(testObject)
}
```

Процесс довольно прост: вы берете объект из пула, используете его, сбрасываете, а затем кладете обратно в пул. Сбросить объект можно как перед тем, как положить его обратно, так и сразу после того, как вы взяли его из пула.

Если вы не любите использовать type assertions `pool.Get().(*Object)`, есть

несколько способов избежать этого.

1. **Используйте специальную функцию для получения объекта из пула**:

```
func getObjectFromPool() *Object {
    obj := pool.Get().(*Object)
    return obj
}
```

2. **Создайте собственную дженерик версию** `sync.Pool` :

```
type Pool[T any] struct {
    sync.Pool
}

func (p *Pool[T]) Get() T {
    return p.Pool.Get().(T)
}

func (p *Pool[T]) Put(x T) {
    p.Pool.Put(x)
}

func NewPool[T any](newF func() T) *Pool[T] {
    return &Pool[T]{
        Pool: sync.Pool{
            New: func() interface{} {
                return newF()
            },
        },
    }
}
```

Дженерик-обертка дает вам более безопасный с точки зрения типов способ работы с пулом, позволяющий избежать type assertions.

Только учтите, что она добавляет немного накладных расходов из-за дополнительного уровня косвенности. В большинстве случаев эти накладные расходы минимальны, но если вы работаете в среде с высокой нагрузкой на процессор, стоит провести сравнительные тесты, чтобы понять, стоит ли оно того.

## sync.Pool и Ловушка аллокаций

В пуле обычно хранится не сам объект, а указатель на него.

Сейчас я объясню это на примере:

```
var pool = sync.Pool{
    New: func() any {
        return []byte{}
    },
}

func main() {
    bytes := pool.Get().([]byte)

    // do something with bytes
    _ = bytes

    pool.Put(bytes)
}
```

Мы используем пул для `[]byte`. Обычно (хотя и не всегда), когда вы передаете значение интерфейсу, это может привести к тому, что значение будет помещено в кучу. Это происходит и здесь, причем не только со слайсами, но и со всем, что вы передаете в `pool.Put()` и что не является указателем.

Если вы проведете проверку с помощью escape analysis:

```
// escape analysis
$ go build -gcflags=-m

bytes escapes to heap
```

Но я не говорю, что наша переменная bytes перемещается в кучу, я бы сказал «значение bytes перемещается в кучу через интерфейс».

Однако если мы передаем указатель в `pool.Put()`, то никакого дополнительного аллокаций не происходит:

```
var pool = sync.Pool{
    New: func() any {
        return new([]byte)
    },
}
```

```
func main() {
    bytes := pool.Get().(*[]byte)

    // do something with bytes
    _ = bytes

    pool.Put(bytes)
}
```

# sync.Pool Internals

Прежде чем мы перейдем к рассмотрению того, как работает `sync.Pool`, стоит ознакомиться с основами модели планирования PMG в Go - это действительно основа того, почему `sync.Pool` так эффективен.

PMG расшифровывается как P(logical **p**rocessors), M (**m**achine threads) и G (**g**oroutines). Ключевым моментом является то, что на каждом логическом процессоре (P) в любой момент времени может быть запущен только один машинный поток (M). А для выполнения горутины (G) она должна быть привязана к потоку (M).

Это сводится к двум ключевым моментам:

- Если у вас есть n логических процессоров (P), вы можете параллельно выполнять до n горутин при условии, что у вас есть как минимум n машинных потоков (M).
- В каждый момент времени на одном процессоре (P) может выполняться только одна горутина (G). Поэтому, когда P1 занят G, никакая другая G не может выполняться на этом P1, пока текущая G не будет заблокирована, не завершится или не произойдет что-то еще, что освободит ее.

Но дело в том, что `sync.Pool` в Go - это не просто один большой пул, он состоит из нескольких «локальных» пулов, каждый из которых привязан к определенному процессорному контексту, или P, которым управляет среда выполнения Go в каждый конкретный момент времени.

Когда горутине, выполняющейся на процессоре (P), нужен объект из пула, она

сначала проверит свой собственный P-локальный пул, прежде чем искать его в другом месте.

Это разумный выбор, поскольку он означает, что каждый логический процессор (P) имеет свой собственный набор объектов для работы. Это уменьшает количество конфликтов между горутинами, поскольку только одна горутина может одновременно обращаться к своему P-локальному пулу.

Таким образом, процесс выполняется очень быстро, потому что нет вероятности того, что две горутины одновременно попытаются взять один и тот же объект из одного и того же локального пула.

## Локальность пула и False Sharing проблема

Ранее мы упоминали, что «только одна горутина может одновременно обращаться к пулу P-local», но на самом деле все немного сложнее.

Посмотрите на приведенную ниже диаграмму: каждый P-локальный пул состоит из двух основных частей: общей цепочки пулов ( `shared` ) и приватного объекта ( `private` ).

Вот определение локального пула в исходном коде Go:

```
type poolLocalInternal struct {
    private any
    shared  poolChain
}

type poolLocal struct {
    poolLocalInternal
    pad [128 - unsafe.Sizeof(poolLocalInternal{})%128]byte
}
```

В `private` хранится один объект, доступ к которому может получить только P, владеющий этим P-локальным пулом, назовем его приватным объектом.

Он создан для того, чтобы горутина могла быстро получить многократно используемый объект без необходимости возиться с мьютексом или трюками синхронизации. Другими словами, только одна горутина может получить доступ к

своему приватному объекту, никакая другая горутина не может конкурировать с ней.

Но если приватный объект недоступен, в дело вступает цепочка общего пула ( `shared` ).

> *Почему объект не будет доступен? Я думал, что только одна горутина может получить и поместить приватный объект обратно в пул. Так кто же тогда конкурент?*

Хотя верно, что только одна горутина может одновременно получить доступ к приватному объекту P, есть одна загвоздка. Если горутина A захватит приватный объект, а затем будет заблокирована или вытеснена, на том же P может начать работать горутина B. Когда это произойдет, горутина B не сможет получить доступ к приватному объекту, потому что он все еще у горутины A.

В отличие от простого приватного объекта, цепочка общего пула ( `shared` ) немного сложнее.

Процесс `Get()` можно представить следующим образом:



Приведенная выше диаграмма не совсем точна, поскольку в ней не учитывается *victim pool*.

Если цепочка общего(shared) пула тоже пуста, `sync.Pool` либо создаст новый объект (при условии, что вы предоставили функцию `New()` ), либо просто вернет `nil` .

> *Подождите, я вижу поле `pad` в P-локальном пуле. Что это такое?*

Одна вещь, которая бросается в глаза при взгляде на структуру пула *P-local*, - это атрибут `pad` .

```
type poolLocal struct {
    poolLocalInternal
    pad [128 - unsafe.Sizeof(poolLocalInternal{})%128]byte
}
```

Этот `pad` может показаться немного странным, поскольку он не добавляет никакой прямой функциональности, но на самом деле он предназначен для предотвращения проблемы, которая может возникнуть на современных многоядерных процессорах, называемой *false sharing*.

Чтобы понять, почему это важно, нам нужно разобраться в том, как процессоры работают с памятью и кэшем.

Современные процессоры используют компонент, называемый кэшем процессора, для ускорения доступа к памяти. Этот кэш разделен на блоки, называемые линиями кэша, которые обычно содержат 64 или 128 байт данных. Когда процессору нужно получить доступ к памяти, он не просто забирает один байт или слово - он загружает целую кеш-линию.

Это означает, что если два фрагмента данных находятся в памяти близко друг к другу, они могут оказаться в одной строке кэша, даже если логически они разделены.

Теперь, в контексте `sync.Pool`, каждый логический процессор (P) имеет свой собственный `poolLocal`, который хранится в массиве. Если структура `poolLocal` меньше размера строки кэша, то несколько экземпляров `poolLocal` из разных P могут оказаться в одной строке кэша. Именно здесь все может пойти не так. Если два Ps, работающие на разных ядрах процессора, попытаются одновременно получить доступ к собственному `poolLocal`, они могут непреднамеренно наступить друг другу на пятки.

Даже если каждый P имеет дело только со своим собственным `poolLocal`, эти структуры могут использовать одну и ту же кэш-линию.

Когда один процессор изменяет что-то в кэш-линии, вся кэш-линия становится недействительной в кэшах других процессоров, даже если они работают с разными переменными в этой же линии. Это может привести к серьезному снижению производительности из-за ненужных валидаций кэша и дополнительного трафика памяти.

Вот тут-то и появляется `128 - unsafe.Sizeof(poolLocalInternal{})%128`.

Он вычисляет количество байт, необходимых для заполнения пула P-local, чтобы его общий размер был кратен 128 байтам. Благодаря такому заполнению каждый `poolLocal` получает свою собственную кэш-линию, предотвращая *false sharing* и обеспечивая более быструю и бесконфликтную работу.

## Pool Chain и Pool Dequeue

Цепочка общих пулов в `sync.Pool` представлена типом под названием `poolChain`.

Это цепочка структур данных dequeue ( `poolDequeue` ).

```
type poolChain struct {
    head *poolChainElt
    tail atomic.Pointer[poolChainElt]
}

type poolChainElt struct {
    poolDequeue
    next, prev atomic.Pointer[poolChainElt]
}
```

Когда текущий пул dequeue (тот, что находится в начале списка) переполняется, создается новый пул dequeue, вдвое больший предыдущего. Этот новый, более крупный пул добавляется в цепочку.

Если вы посмотрите на структуру poolChain, то заметите, что в ней есть два поля: указатель `head *poolChainElt` и атомарный указатель `tail atomic.Pointer[poolChainElt]`.

Эти поля показывают, как работает механизм:

- *Producer* (P владеющий текущим P-локальным пулом) добавляет новые элементы только в последний пул dequeue, который мы называем *head*. Поскольку к head имеет доступ только *producer*, нет необходимости в блокировках или каких-либо сложных трюках синхронизации, поэтому все происходит очень быстро.
- *Consumers* (другие Ps) берут элементы из пула dequeue с хвоста списка.

Поскольку несколько *consumer* могут пытаться получить элементы одновременно, доступ к *tail* синхронизируется с помощью атомарных операций для поддержания порядка.

**Ключевой момент**:

- Когда пул dequeue в хвосте полностью освобождается, он удаляется из списка, а следующий по очереди пул dequeue становится новым хвостом. Но в голове ситуация немного другая.
- Когда в головном пуле dequeue заканчиваются элементы, он не удаляется. Вместо этого он остается на месте, готовый к пополнению при добавлении новых элементов.

Теперь давайте рассмотрим, как определяется пул dequeue. Как следует из названия «dequeue», это двусторонняя очередь.

В отличие от обычной очереди, где вы можете добавлять элементы только в конец, а удалять их из начала, dequeue позволяет вставлять и удалять элементы как в конец, так и в начала.

Ее механизм на самом деле очень похож на цепочку пулов. Она устроена так, что один производитель может добавлять или удалять элементы из головы, а несколько потребителей могут забирать элементы из хвоста.

```
type poolDequeue struct {
    headTail atomic.Uint64
    vals []eface
}
```

Производитель (который является текущим P) может добавлять новые элементы в переднюю часть очереди или забирать их из нее.

В то же время потребители могут брать элементы только из хвоста очереди. Эта очередь свободна от блокировок, что означает, что она не использует блокировки для управления координацией между производителем и потребителями, а только использует атомарные операции.

Вы можете представить себе эту очередь как своего рода кольцевой буфер.

*Кольцевой буфер(ring buffer) или круговой буфер(circular buffer)*, - это структура данных, которая использует массив фиксированного размера для хранения элементов в циклическом режиме. Она называется «кольцевой», потому что конец буфера в некотором смысле заворачивается, чтобы встретиться с началом, что делает его похожим на круг.

В контексте пула dequeue, о котором мы говорим, поле `headTail` - это 64-битное целое число, которое упаковывает два 32-битных индекса в одно значение.

Эти индексы являются головой и хвостом очереди и помогают отслеживать, где в буфере хранятся данные и как к ним обращаться.

- **tail index** указывает на место, где находится самый старый элемент в буфере, и когда потребители (например, другие горутины) читают из буфера, они начинают с этого места и продвигаются вперед.
- **head index** - это место, куда будет записан следующий фрагмент данных. Когда поступают новые данные, они помещаются в этот головной индекс, а затем индекс перемещается в следующий доступный слот.

*Упаковывая два индекса в одно 64-битное значение, можно обновить оба индекса за один проход, что делает операцию атомарной.*

Это особенно полезно, когда два потребителя (или потребитель и производитель) пытаются одновременно извлечь элемент из очереди. Операция *CompareAndSwap (CAS)*, `d.headTail.CompareAndSwap(ptrs, ptrs2)` , гарантирует, что только один из них достигнет успеха. Другой терпит неудачу и повторяет попытку, обеспечивая порядок без сложной блокировки.

Фактические данные в очереди хранятся в круговом буфере под названием `vals` , размер которого должен быть равен степени двойки.

Такой выбор дизайна позволяет легче справляться с разворачиванием очереди, когда она достигает конца буфера. Каждый слот в этом буфере - это значение `eface` , именно так Go представляет пустой интерфейс ( `interface{}` ) под

капотом.

```
type eface struct {
    typ, val unsafe.Pointer
}
```

*Слот в буфере остается «in use» до тех пор, пока не произойдут две вещи*:

- Хвостовой индекс перемещается за пределы слота, что означает, что данные в этом слоте были использованы одним из потребителей.
- Потребитель, получивший доступ к этому слоту, устанавливает его в `nil`, сигнализируя, что производитель теперь может использовать этот слот для хранения новых данных.

Вкратце, цепочка пула объединяет связанный список и кольцевой буфер для каждого узла. Когда один dequeue заполняется, создается новый, больший по размеру, и он связывается с головной частью цепочки. Такая настройка помогает эффективно управлять большим количеством объектов.

# Pool.Put()

Начнем с `Put()`, потому что он немного проще, чем `Get()`, а также связан с другим процессом: *прикреплением горутины к P*.

Когда горутина вызывает `Put()` для `sync.Pool`, первое, что она пытается сделать, это сохранить объект в приватном сегменте P-local пула для текущего P. Если этот приватный сегмент уже занят, объект перемещается в начало цепочки пула, то есть в общую часть.

```
func (p *Pool) Put(x interface{}) {
    // If the object is nil, it will do nothing
    if x == nil {
        return
    }

    // Pin the current P's P-local pool
    l, _ := p.pin()

    // If the private pool is not there, create it and set the object to it
```

```
    if l.private == nil {
        l.private = x
        x = nil
    }

    // If the private object is there, push it to the head of the shared chain
    if x != nil {
        l.shared.pushHead(x)
    }

    // Unpin the current P
    runtime_procUnpin()
}
```

Начиная с Go 1.14, в Go появилось **вытесняющее планирование(preemptive scheduling)**, которое означает, что рантайм может приостановить горутину, если она работает на процессоре P слишком долго, обычно около 10мс, чтобы дать другим горутинам шанс на выполнение.

В целом это хорошо для обеспечения справедливости и отзывчивости, но при работе с `sync.Pool` могут возникнуть проблемы.

Такие операции, как `Put()` и `Get()` в `sync.Pool`, предполагают, что горутина остается на одном процессоре (скажем, P1) на протяжении всей работы. Если горутина будет вытеснена в середине этих операций, а затем возобновлена на другом процессоре (P2), локальные данные, с которыми она работала, могут оказаться на другом процессоре.

Итак, что же делает функция `pin()`? Вот комментарий из исходного кода Go, который объясняет это:

```
// pin pins the current goroutine to P, disables preemption and
// returns poolLocal pool for the P and the P's id.
// Caller must call runtime_procUnpin() when done with the pool.
func (p *Pool) pin() (*poolLocal, int) { ... }
```

По сути, `pin()` временно лишает планировщик возможности вытеснить горутину, пока она помещает объект в пул.

Хотя здесь сказано «прикрепить текущую горутину к P», на самом деле происходит то, что текущий поток (M) зафиксирован на процессоре (P), что не

позволяет ему быть вытесненным. В результате горутина, выполняющаяся в этом потоке, также не будет вытеснена.

Кроме того, `pin()` обновляет количество процессоров (Ps), если вы случайно изменили `GOMAXPROCS(n)` (который управляет количеством Ps) во время выполнения.

Когда вам нужно добавить элемент в цепочку, операция сначала проверяет голову цепочки. Помните указатель head `*poolChainElt` ? Это самый последний пул dequeue в списке.

В зависимости от ситуации, вот что может произойти:

- Если головной буфер цепочки равен `nil` , то есть в цепочке еще нет пула dequeue, создается новый пул dequeue с начальным размером буфера 8. Затем элемент помещается в этот совершенно новый пул dequeue.
- Если головной буфер цепочки не равен `nil` и этот буфер не заполнен, элемент просто добавляется в буфер в головной позиции.
- Если головной буфер цепочки не равен `nil` , но этот буфер заполнен, создается новый пул dequeue. Этот новый пул имеет размер буфера, вдвое превышающий размер текущего head. Элемент помещается в этот новый пул dequeue, и головная часть цепочки пулов обновляется, чтобы указывать на этот новый пул.

Это относительно простой процесс, поскольку он не предполагает взаимодействия с другими процессорами локального пула (Ps); все происходит в пределах текущего head цепочки пула.

# sync.Pool.Get()

`Pool.Get` начинается с фиксации текущей горутины в своем P, чтобы предотвратить вытеснение, затем проверяет и берет приватный объект из своего P-локального пула, не требуя синхронизации. Если приватного объекта там нет, он проверяет цепочку общего пула и переходит в начало цепочки.

Только горутина, выполняющаяся в текущем P-локальном пуле, может получить

доступ к началу цепочки, поэтому мы используем `popHead()` :

```
func (p *Pool) Get() interface{} {
    // Pin the current P's P-local pool
    l, pid := p.pin()

    // Get the private object from the current P-local pool
    x := l.private
    l.private = nil

    // If the private object is not there, pop the head of the shared pool chain
    if x == nil {
        x, _ = l.shared.popHead()

        // Steal from other P's cache
        if x == nil {
            x = p.getSlow(pid)
        }
    }
    runtime_procUnpin()

    // If the object is still not there, create a new object from the factory fun
    if x == nil && p.New != nil {
        x = p.New()
    }
    return x
}
```

Здесь мы также получаем `pid` - идентификатор P, на котором запущена текущая горутина. Он нужен нам для процесса *stealing*, который вступает в игру, если быстрый путь не работает.

Быстрый путь - это когда объект доступен в кэше текущего P. Но если это не удается, то есть приватный объект и head общей цепочки оказываются пустыми, на смену приходит медленный путь ( `getSlow` ).

В медленном пути мы пытаемся *похитить(steal)* объекты из кэш-пулов других процессоров (Ps).

*Идея похищения(stealing)* заключается в повторном использовании объектов, которые могут простаивать в кэшах других процессоров, вместо того чтобы создавать новые объекты с нуля. Если у другого P есть лишние объекты в его кэш-

пуле, текущий P может взять эти объекты и использовать их.

![[Pasted image 20241016165407.png]]

Процесс похищения(stealing), по сути, обходит все P, кроме текущего ( `pid` ), и пытается получить объект из цепочки общего пула каждого P:

```
for i := 0; i <int(size); i++ {
    l := indexLocal(locals, (pid+i+1)%int(size))
    if x, _ := l.shared.popTail(); x != nil {
        return x
    }
}
```

Как мы уже говорили, в `poolChain` поставщик (текущий P) работает с head, а многочисленные потребители (другие P) только получают объекты из хвоста.

Итак, `popTail` смотрит на последний пул dequeue и пытается получить данные из конца этого пула.

Если он не находит данных в этом пуле dequeue, хвостовой индекс увеличивается, и этот пул dequeue удаляется из цепочки.

Этот процесс продолжается до тех пор, пока не будет либо успешно получены данные, либо не закончатся варианты во всех цепочках пулов.

Если после всех попыток все еще не найдены никакие данных, функция пытается получить их от так называемой «*victim*».

![[Pasted image 20241016170124.png]]

Мы пытаемся получить объект всеми возможными способами, и если ничего не найдено, тогда создаем новый объект с помощью `New()` . Но если `New()` - это `nil` , то он просто возвращает `nil` . Все просто.

Теперь, после попытки работы с victim пулом, пул атомарно помечается как пустой. Последующие операции `Get()` будут пропускать проверку victim, пока он снова не заполнится.

# Victim Pool

Несмотря на то что `sync.Pool` создан для лучшего управления ресурсами, он не дает нам, разработчикам, прямых инструментов для очистки или управления жизненным циклом объектов. Вместо этого `sync.Pool` выполняет очистку за кулисами, чтобы избежать бесконтрольного роста, который может привести к утечкам памяти.

Помните, мы говорили о `pin()` ? Оказывается, у `pin()` есть еще один побочный эффект. Каждый раз, когда `sync.Pool` вызывает `pin()` в первый раз (или после изменения количества Ps через `GOMAXPROCS` ), он добавляется в глобальный слайс под названием `allPools` в пакете `sync` :

```
package sync

var (
    allPoolsMu Mutex

    // allPools is the set of pools that have non-empty primary
    // caches. Protected by either 1) allPoolsMu and pinning or 2)
    // STW.
    allPools []*Pool

    // oldPools is the set of pools that may have non-empty victim
    // caches. Protected by STW.
    oldPools []*Pool
)
```

Этот слайс `allPools []*Pool` отслеживает все активные экземпляры `sync.Pool` в вашем приложении.

Перед началом каждого цикла сборки мусора (GC) рантайм Go запускает процесс очистки, который очищает слайс `allPools` . Вот как это работает:

- Перед запуском GC вызывается `clearPool` , который переносит все объекты в `sync.Pool` , включая приватные объекты и цепочки общего пула, в так называемую *victim area*.
- Эти объекты не выбрасываются сразу, они пока хранятся в этом месте.
- При этом объекты, которые уже находились в области victim с последнего

цикла GC, полностью очищаются во время текущего цикла GC.

```
func poolCleanup() {
    // Drop victim caches from all pools.
    for _, p := range oldPools {
        p.victim = nil
        p.victimSize = 0
    }

    // Move primary cache to victim cache.
    for _, p := range allPools {
        p.victim = p.local
        p.victimSize = p.localSize
        p.local = nil
        p.localSize = 0
    }

    // The pools with non-empty primary caches now have non-empty
    // victim caches and no pools have primary caches.
    oldPools, allPools = allPools, nil
}
```

> Но зачем нам нужен этот механизм victim? Почему для очистки всех объектов в пуле требуется до двух циклов GC?

Причина использования механизма victim в `sync.Pool` заключается в том, чтобы избежать внезапного и полного очищения пула сразу после цикла GC. Если бы пул был очищен сразу, это могло бы привести к проблемам с производительностью, поскольку при любых новых запросах на объекты их пришлось бы создавать заново. Поэтому мы сначала перемещаем объекты в область-victim, а `sync.Pool` обеспечивает буферный период, в течение которого объекты еще могут быть использованы повторно, прежде чем они будут полностью удалены.

Итак, **для полного удаления объекта в `sync.Pool` требуется не менее 2 циклов GC**.

*Это может стать проблемой для программ с низким значением `GOGC`, которое контролирует частоту выполнения GC для очистки неиспользуемых объектов.* Если значение `GOGC` слишком мало, процесс очистки может удалять неиспользуемые объекты слишком быстро, что приведет к увеличению числа

промахов кэша.

Даже при использовании `sync.Pool`, если вы имеете дело с очень высоким параллелизмом и медленным GC, вы можете столкнуться с большими накладными расходами. В этом случае хорошим решением может стать использования *rate limiting* для `sync.Pool`.

---

Комментарии в Telegram-группе!

go concurrency

« ПРЕДЫДУЩАЯ
Range по функциям в Go 1.23

СЛЕДУЮЩАЯ »
Использование benchstat проекций в анализе Go бенчмарков