garbage collector

서문(introduction)

이 가이드는 Go 가비지 컬렉터에 대한 통찰력을 제공하여 고급 Go 사용자가*애플리케이션 비용*을 더 잘 이해할 수 있도록 돕기 위한 것이다. 또한 Go 사용자가 이러한 통찰력을 사용하여 *애플리케이션의 리소스 활용도를 향상*시킬 수 있는 방법에 대한 지침도 제공한다.

Go 언어는 Go 값들의 저장 공간을 정리(arrange)하는 책임을 진다. 대부분의 경우, Go 개발자는 이러한 값들이 어디에 저장되는지, 또는 왜 저장되어야 하는지(만약 저장되어야 한다면)에 대해 신경 쓸 필요가 없다. 그러나 실제로, 이러한 값들은 종종 컴퓨터의 물리적 메모리에 저장되어야 하며, 물리적 메모리는 한정된 자원이다. 메모리는 유한하기 때문에 Go 프로그램을 실행하는 동안 메모리가 부족해지는 것을 방지하려면 메모리가 *신중하게 관리되고 재활용*돼야 한다. 필요에 따라 메모리를 할당하고 회수하는 것은 Go 구현체(Go 컴파일러와 런타임 시스템)의 역할이다.

자동으로 메모리를 재활용하는 것을 다른 말로 하면 가비지 수집(garbage collection)이다. 높은 수준에서 가비지 수집기는 *메모리의 어느 부분이 더 이상 필요하지 않은지 식별하여 애플리케이션을 대신하여 메모리를 재활용하는 시스템*이다. Go 표준 툴체인은 모든 애플리케이션과 함께 포함되어 배포되는 런타임 라이브러리를 제공하며, 이 런타임 라이브러리에는 가비지 수집기가 포함되어 있다.

이 가이드에 설명된 가비지 컬렉터의 존재는 Go의 공식 사양에서 명시적으로 보장되지는 않는다. Go 사양은 단지 *Go 값들이 언어에 의해 관리되는 기본적인 저장소(underlying storage for Go values)를 갖고 있다*는 것만을 명시한다. 이러한 명시적인 언급의 생략은 의도된 것으로, 이는 다양하고 근본적으로 다른 메모리 관리 방식을 가능하게 한다.

> underlying storage for Go values?
>
> Go 언어에서 값들이 저장되는 기본적인 메모리 공간을 의미한다. 여기서 "Go 값"이란 변수, 객체, 구조체 등 Go 프로그램에서 사용되는 다양한 데이터 유형을 말한다. "underlying storage"는 이러한 값들이 실제로 메모리 상에 배치되고 관리되는 방식을 가리킨다.

따라서 이 가이드는 Go 프로그래밍 언어의 특정 구현에 관한 것이며 다른 구현에는 적용되지 않을 수 있다. 특히 다음 가이드는 표준 도구 체인(gc Go 컴파일러 및 도구)에 적용된다. `Gccgo`와 `Gollvm`은 모두 매우 유사한 GC 구현을 사용하므로 동일한 개념이 많이 적용되지만 세부 사항은 다를 수 있다.

게다가 이 문서는 살아있는 문서이며 Go의 최신 릴리스를 가장 잘 반영하기 위해 시간이 지남에 따라 변경된다. 이 문서에서는 현재 Go 1.19 기준의 가비지 수집기에 대해 설명한다.

Go 값이 저장되는 위치

GC에 대해 자세히 알아보기 전에, 먼저 GC의 관리가 필요가 없는 메모리에 대해 살펴본다.

예를 들어, *지역 변수에 저장된 포인터가 아닌 Go 값*은 Go GC에 의해 관리될 가능성이 거의 없으며, 대신 Go는 그 *지역 변수에 저장된 포인터가 아닌 Go 값*이 생성된 어휘 범위(lexical scope)에 맞게 메모리를 할당하도록 준비한다.

> **Lexical Scope(어휘 범위)?**
>
> 어휘 범위란 변수가 프로그램의 코드에서 직접적으로 접근할 수 있는 범위를 말하며, 코드를 작성하는 시점에 변수의 유효 범위가 결정된다는 것을 의미한다. 프로그램의 소스 코드에서 변수가 선언된 위치에 따라 그 변수의 유효 범위가 결정된다. 예를 들어, 함수 내부에 선언된 지역 변수는 그 함수 내부에서만 접근 가능하며, 이러한 변수의 범위를 어휘 범위라고 한다.
>
> ```
> func myFunction() {
>     var localVariable int = 10 // localVariable은 myFunction의 어휘 범위 내에서만 접근 가능하다.
>     fmt.Println(localVariable) // 여기서는 localVariable을 사용할 수 있다.
> }
>
> func main() {
>     myFunction()
>     // fmt.Println(localVariable) // myFunction의 어휘 범위 밖이기 때문에 여기서는 localVariable에 접근할 수 없다.
> }
> ```
>
> **어휘 범위에 맞게 메모리를 할당?**
>
> Go 언어에서 변수가 함수 내부 등 특정 어휘 범위 안에서 선언되면, 그 변수의 메모리 할당 및 생명주기는 그 어휘 범위에 따라 결정된다. 즉, 변수가 해당 범위(예를 들어, 함수 또는 블록) 내에서만 존재하며, 그 범위를 벗어나면 해당 변수에 할당된 메모리가 자동으로 해제될 수 있도록 Go가 메모리 할당을 관리한다. 어휘 범위 내에 있는 지역 변수는 해당 범위의 실행 컨텍스트가 시작될 때 메모리가 할당되고, 그 범위의 실행이 종료될 때 자동으로 메모리가 해제된다. 이는 특히 스택 메모리 할당에 해당하는 작업으로, 해당 변수의 메모리 할당 및 해제가 런타임 시에 자동으로 이루어진다. 이처럼 포인터가 아닌 Go 값들은 가비지 컬렉션(GC)에 의존하지 않는 방식으로 관리된다.

Go 컴파일러는 해당 메모리를 언제 해제할 수 있는지 미리 결정하고 정리하는 머신 명령을 발행할 수 있기 때문에, 일반적으로 GC에 의존하는 것보다 더 효율적이다. 이러한 방식으로 Go 값의 메모리를 할당하는 과정은 '스택 할당'(stack allocation)이

라고 불린다. 해당 메모리 공간이 고루틴의 스택 영역 내에 위치하기 때문이다.

> **스택 할당?**
>
> 함수 내에서 선언된 지역 변수들은 함수가 호출될 때 함수의 호출 스택에 할당되고, 함수가 반환될 때 자동으로 메모리에서 해제된다. 이 과정은 Go 컴파일러에 의해 처리되며, 메모리 할당과 해제가 명시적으로 코드에 작성될 필요가 없다. Go 컴파일러는 변수가 함수나 블록 내에서 어떻게 사용되는지 분석하고, 이를 바탕으로 메모리 할당 및 해제 시점을 결정한다. 즉, 포인터가 아닌 Go 값들, 특히 지역 변수들은 대체로 가비지 컬렉터(GC)가 아닌 Go 언어의 컴파일러에 의해 관리된다. 따라서 포인터가 아닌 Go 값들이나 지역 변수들은 가비지 컬렉터(GC)가 아닌 Go 컴파일러에 의해 효율적으로 관리되며, 이는 프로그램의 성능 및 메모리 효율성을 향상시키는 데 중요한 역할을 한다.

Go 컴파일러가 그 수명을 결정할 수 없어 이러한 방식으로 메모리를 할당할 수 없는 Go 값은 *힙으로 이스케이프된다*고 한다.

> **이스케이프된다?**
>
> 이 표현은 주로 변수가 선언된 범위(예: 함수)에 묶여 있지 않고, 더 넓은 범위(예: 전체 프로그램)에서 접근 가능해질 때 사용된다. Go에서는 변수가 함수의 호출 스택보다 긴 생명주기를 가지는 경우(예: 함수 밖으로 반환되거나 다른 고루틴에서 참조될 때)처럼, 변수가 함수의 지역 스택 대신 힙에 저장될 필요가 있을 때 발생한다.

"힙"은 Go 값이 어딘가에 배치될 필요가 있을 때, 메모리 할당을 위한 포괄적인 개념(catch-all)으로 생각할 수 있다. 컴파일러와 런타임 모두 이 메모리가 어떻게 사용되고 언제 정리될 수 있는지에 대해 거의 가정할 수 없기 때문에, 힙에 메모리를 할당하는 행위는 일반적으로 "동적 메모리 할당"이라고 한다. 이때 *동적 메모리 할당을 구체적으로 식별하고 정리하는 시스템 GC*가 등장한다.

Go 값이 힙으로 이스케이프되어야 하는 데에는 여러 가지 이유가 있다. 한 가지 이유는 *크기가 동적으로 결정*될 수 있기 때문이다. 예를 들어 초기 크기가 상수가 아닌 변수에 의해 결정되는 슬라이스의 백업 배열(backing array)을 생각해보자.

> **backing array?**
>
> 슬라이스가 참조하는 배열을 의미한다. 슬라이스는 배열의 연속된 부분을 표현(일종의 뷰, window)하며, 배열을 통해 메모리에 저장된 데이터에 접근한다. 이 배열을 'backing array'라고 부르는 이유는 슬라이스가 배열의 일부분에 대한 참조를 제공하기 때문. 즉, 어떤 슬라이스가 참조하는 배열이 그 슬라이스의 데이터를 '지지' 또는 '지원'한다는 의미로 해석할 수 있다.

```
package main

import "fmt"

func main() {
    original := [5]int{1, 2, 3, 4, 5} // 고정된 길이를 갖는 연속적인 메모리 공
간 배열 생성

    // 배열의 일부분을 참조하는 가변 길이의 구조인 슬라이스 `a`를 만들어 배열 전체를 참조
    a := original[:] // `a`의 `backing array`는 원본 배열 `original`

    // 슬라이스 `a`의 일부분을 참조하는 새로운 슬라이스 `b`를 생성
    b := a[1:4] // `b`의 `backing array`는 `a`가 참조하는 같은 원본 배열
`original`

    // `b`의 첫 번째 요소를 변경
    b[0] = 20 // `b`의 변경이 `a`와 `original`에도 영향을 미친다
```

```
    fmt.Println(original) // [1 20 3 4 5]
    fmt.Println(a)        // [1 20 3 4 5]
    fmt.Println(b)        // [20 3 4]
}
```

references:

- Understanding Backing Array— Golang
- Build your own slice: Arrays and slices

힙으로의 이스케이프도 전이적(transitive)이어야 한다.

전이적(transitive)?

한 요소의 특성이나 상태가 다른 요소에 영향을 미칠 때 사용된다. "힙으로의 이스케이프가 전이적이어야 한다"는 말은 하나의 Go 값이 힙으로 이스케이프되었을 때, 이 값에 대한 참조를 갖는 다른 Go 값도 동일한 방식으로 이스케이프되어야 함을 의미한다. 즉, Go 언어의 메모리 관리 방식에서 *한 변수의 메모리 할당 상태가 다른 변수에 연쇄적으로 영향*을 미칠 수 있음을 나타낸다 예를 들어, 어떤 변수가 힙에 저장되어야 하는 경우, 이 변수를 참조하는 다른 모든 변수들도 자동으로 힙에 저장되어야 한다. 힙에 저장되는 것을 피하려면, **변수의 참조를 불필요하게 외부로 노출하지 않도록 설계하는 것이 중요**하다.

1. 가능하면 값을 직접 전달
2. 변수의 범위를 가능한 한 좁게 유지

즉, *어떤 Go 값에 대한 참조*가 *이미 힙으로 이스케이프되도록 결정된 다른 Go 값*에 기록되면(written into), 해당 값도 이스케이프되어야 한다.

어떤 Go 값에 대한 참조가 이미 힙으로 이스케이프 되도록 결정된 다른 Go 값에 기록?

```
func main() {
    a := 10 // 스택에 할당되는 지역 변수
    b := &a // `b`는 `a`의 스택 주소를 참조하는 포인터
    c := foo(b)
    // ... 나머지 코드 ...
}

func foo(x *int)*int {
    d := *x // `d`는 `foo` 함수의 지역 변수로, `x`(`&a`)의 값(`*x` == `a`)을 복사하여 저장한다
            // "기록된다"(written into)는 것은 일반적으로 변수에 값이 할당되거나 변수가 다른 변수를 참조할 때 발생
    return &d // `d`의 주소가 `main`으로 리턴되면서 `foo`의 스택 프레임을 벗어나므로,
              // 이때 `d`는 힙으로 이스케이프되기로 결정된다.
}
```

- 어떤 Go 값: a
- 어떤 Go 값에 대한 참조: &a
- 이미 힙으로 이스케이프되도록 결정된 다른 Go 값: d

- 이미 힙으로 이스케이프되도록 결정된 다른 Go 값에 기록되는 것: `d := *x`

```
package main

import "fmt"

type Data struct {
    Value *int
}

func main() {
    a := 10 // 어떤 Go 값
    b := &a // 어떤 Go 값에 대한 참조

    d := Data{Value: b} // 이미 힙으로 이스케이프된 값(`d`)에 기록되는 것
                        // `a`에 대한 참조가 힙으로 이스케이프된 `d` 내에 기록 기
록된다.
                        // 이로 인해서 `a`의 생명주기가 `d`에 의존하게 되고,
                        // 결과적으로 `a`가 힙으로 이스케이프하는 상황을 야기할 수
있다
    updateData(&d)
    fmt.Println(*d.Value)
}

func updateData(e *Data) { // `e`는 main 함수로부터 전달된 Data 구조체(`d`)의
주소를 참조(`&d`)
    newVal := 20 // 지역 변수
    e.Value = &newVal // 이미 힙으로 이스케이프된 값(`d`)에 기록되고,
                      // 따라서 `newVal`도 힙으로 이스케이프된다.
}
```

- 어떤 Go 값: `a`, `newVal`
- 어떤 Go 값에 대한 참조: `b`(`&a`), `&newVal`
- 이미 힙으로 이스케이프되도록 결정된 다른 Go 값: `d`
- 이미 힙으로 이스케이프되도록 결정된 다른 Go 값에 기록되는 것: `d := Data{Value: b}`, `e.Value = &newVal`

Go 값이 이스케이프되는지 여부는 *해당 값이 사용되는 컨텍스트*와 *Go 컴파일러의 이스케이프 분석 알고리즘*에 따라 달라진다. 알고리즘 자체는 상당히 정교하며 Go 릴리스 간에 변경되기 때문에, 값이 이스케이프되는 시점을 정확하게 열거하는 것은 취약하고 어렵다. 이스케이프되는 값과 그렇지 않은 값을 식별하는 방법에 대한 자세한 내용은 힙 할당 제거(eliminating heap allocations) 섹션을 참조.

Tracing Garbage Collection

가비지 컬렉션은, 예를 들어 참조 카운팅(reference counting) 같이, 자동으로 메모리를 재활용하는 다양한 방법을 의미할 수 있다. 이 문서에서 가비지 컬렉션은 포인터를 전이적으로 따라가면서 사용 중인 객체, 이른바 라이브 객체를 식별하는 **추적(tracing)** 가비지 컬렉션을 의미한다.

이러한 용어를 좀 더 엄격하게 정의해 보자.

- 객체: 동적으로 할당된 메모리 조각으로, 하나 이상의 Go 값들을 포함한다.
- 포인터:
    - 객체 내의 어떤 값을 참조하는 메모리 주소
    - 여기에는 당연히 *T 형식의 Go 값이 포함되지만 내장된 Go 값의 일부도 포함된다.
    - 가령 다음 항목들은 GC가 추적해야 하는 메모리 주소들을 포함한다.
        - 문자열: 문자열은 *내부적으로 문자 데이터가 저장된 메모리 영역을 가리키는 포인터*와 *문자열의 길이 정보*를 포함한다.
        - 슬라이스: 슬라이스는 배열의 일부분을 참조하며, 내부적으로 *배열의 메모리 주소*, *슬라이스의 길이*, 그리고 *용량*을 저장한다.
        - 채널: 채널은 고루틴 간의 통신을 위해 사용되며, 내부적으로 *메시지 큐의 메모리 주소*를 관리한다.
        - 맵: 내부적으로 *해시 테이블의 메모리 주소*를 포함하여, 동적으로 메모리를 할당하고 관리한다.
        - 인터페이스: 인터페이스는 다양한 타입의 값을 동적으로 저장할 수 있으며, 내부적으로 *구체적인 값의 타입 정보*와 *그 값을 가리키는 메모리 주소*를 저장한다.

> **GC가 추적해야 하는 메모리 주소들?**
>
> 동적으로 할당된 메모리 영역 내에서 아직 사용 중인 데이터를 가리키는 주소들을 의미한다. 이러한 메모리 주소들은 프로그램의 실행 도중 생성되고, 데이터에 접근하기 위해 사용된다. 가비지 컬렉터는 이 주소들을 추적하여, 더 이상 사용되지 않는 메모리(즉, 어떤 변수나 포인터에도 참조되지 않는 메모리 영역)를 식별하고 해제함으로써 메모리 누수를 방지하고 효율적인 메모리 사용을 유지한다.
>
> - 메모리 할당 추적: 슬라이스가 새로운 배열을 참조하게 되거나, 기존 배열에서 일부를 참조하도록 변경될 때, GC는 이러한 참조 정보를 업데이트하여 해당 배열의 메모리가 여전히 필요한지를 판단
> - 메모리 해제: 슬라이스가 더 이상 배열을 참조하지 않게 되거나(예: 슬라이스가 다른 배열을 가리키도록 변경되거나, 슬라이스 자체가 사용 범위를 벗어나게 되는 경우), 프로그램의 다른 부분에서 해당 배열에 대한 참조가 모두 사라지면, GC는 해당 배열의 메모리를 해제할 수 있다.

객체와 다른 객체에 대한 포인터가 함께 객체 그래프(object graph)를 형성한다. 라이브 메모리를 식별하기 위해 GC는 *프로그램에 의해 확실하게 사용중인 객체를 식별하는 포인터 집합*인 프로그램의 루트부터 시작하여 객체 그래프를 탐색한다.

> **프로그램 루트?**
>
> 프로그램의 루트는 가비지 컬렉터가 메모리를 탐색하고 어떤 객체가 여전히 사용 중인지(즉, '라이브' 상태인지)를 판단하는 출발점이다. 이러한 루트는 주로 전역 변수, 지역 변수, CPU 레지스터에 저장된 포인터 등 *프로그램 실행 중에 활성화된 메모리 영역을 가리키는 포인터들*로 구성된다. 이 포인터들은 프로그램이 확실히 사용하고 있는 객체들을 식별하며, 프로그램의 루트는 GC가 객체 그래프 탐색을 시작하는 지점이다
>
> 프로그램 루트에는 다음과 같은 요소들이 포함될 수 있다
>
> - 전역 변수: 프로그램 전역에서 사용되는 변수들로, 프로그램이 실행되는 동안 계속해서 접근 가능한 메모리 영역을 참조
> - 활성 스택 프레임의 로컬 변수 및 매개변수: 현재 실행 중인 함수의 스택 프레임에 포함된 변수들과 매개변수는 함수의 실행 컨텍스트 내에서 사용된다. 포인터가 아닌 기본 타입의 변수들은 GC의 직접적인 관리 대상이 아니나, 동적으로 할당된 메모리를 참조하는 포인터나 참조 타입의 변수는 GC에 의해 추적되며, 이러한 변수들을 통해 도달할 수 있는 객체들은 메모리 해제 대상이 될 수 있다.
> - 고루틴의 스택: 동시에 실행되고 있는 각 고루틴의 스택도 루트 집합의 일부로 간주된다. 각 고루틴은 자신만의 스택을 가지며, 이 스택 내의 포인터들도 GC에 의해 추적된다.
> - CPU 레지스터 내의 포인터
> - 기타 실행 중인 스레드의 스택 포인터

루트의 두 가지 예는 지역 변수와 전역 변수다. 객체 그래프를 탐색하는 과정은 스캔이라고 한다.

이 기본 알고리즘은 모든 추적 GC에 공통된다. 추적 GC는 메모리가 활성화된 것을 발견한 후 수행하는 작업이 다르다. Go의 GC는 마크 스윕(mark-sweep) 기술을 사용한다. 이는 진행 상황을 추적하기 위해 GC가 발견한 값을 실시간으로 *표시*한다는 의미. 추적이 완료되면, GC는 힙의 모든 메모리를 탐색해서 *표시*되지 않은 모든 메모리를 할당에 사용할 수 있도록 한다. 이 프로세스를 스위핑(sweeping)이라고 한다.

익숙한 한 가지 대체 기술은 실제로 객체를 메모리의 새로운 부분으로 이동시키고, 나중에 모든 애플리케이션의 포인터를 업데이트하는 데 사용되는 전달(forwarding) 포인터를 남겨 두는 것이다. 이런 방식으로 객체를 이동하는 GC를 **이동(moving)** GC라고 부른다. Go에는 **움직이지 않는(non-moving)** GC가 있다.

The GC cycle

Go GC는 마크-스윕 GC이기 때문에 크게 마크 단계와 스윕 단계의 두 단계로 작동한다. 이 설명은 동어반복으로 보일 수 있지만, 여기에는 중요한 통찰력이 담겨 있다. 스캔되지 않은 포인터가 여전히 객체를 유지하고 있을 수 있기 때문에, 모든 메모리를 추적되기 전에는 할당되도록 메모리를 해제하는 것이 불가능하다. 따라서 스위핑 행위는 마킹 행위와 완전히 분리되어야 한다. 또한 GC 관련 작업이 없을 때는 GC가 전혀 활성화되지 않을 수도 있다. GC는 이 세 단계의 스위핑, 꺼짐, 마킹을 통해 지속적으로 반복하는데, 이를 GC 사이클이라고 한다. 이 문서의 목적상 GC 주기는 스위핑, 꺼짐, 그리고 마킹 순으로 시작하는 것으로 간주한다.

> **스위핑, 꺼짐, 마킹**
>
> - 스위핑(Sweeping): 가비지 컬렉터가 더 이상 사용되지 않는 메모리 영역을 식별하고 해제하는 단계. 마킹 단계에서 식별된 가비지들이 메모리에서 제거된다.
> - 꺼짐(Off): GC 작업이 일시적으로 중단되어, 프로그램의 실행에 영향을 주지 않는 단계. GC가 활성화되지 않으며, 프로그램의 다른 작업이 우선적으로 수행된다.
> - 마킹(Marking): 가비지 컬렉터가 프로그램에서 여전히 사용 중인 객체를 식별하는 단계. 프로그램의 루트부터 시작하여 객체 그래프를 탐색하고, 도달 가능한 모든 객체를 "라이브"로 마킹한다.

다음 몇 섹션에서는 사용자가 자신에게 유리하도록 GC 매개변수를 조정(tweaking)하는 데 도움이 되도록 GC 비용에 대한 직관력을 키우는 데 중점을 둘 것이다.

Understanding costs

GC는 본질적으로 훨씬 더 복잡한 시스템을 기반으로 구축된 복잡한 소프트웨어이기 때문에, GC를 이해하고 그 동작을 조정하려고 하면 세부적인 수렁에 빠지기 쉽다. 이 섹션은 Go GC의 비용과 튜닝 매개변수에 대한 추론의 틀을 제공하기 위한 것이다.

우선 세 가지 간단한 공리(axioms)를 기반으로 하는 GC 비용 모델을 고려해보자.

1. GC는 두 가지 리소스만 사용한다.
    - *CPU 시간*
    - *물리적 메모리*
2. GC의 메모리 비용 구성
    - 라이브 힙 메모리

        > 라이브 힙 메모리는 *이전* GC 사이클에 의해 라이브로 결정된 메모리 인 반면,새로운 힙 메모리는 *현재* GC 사이클에 할당된 메모리로, 마지막까지 라이브일 수도 있고 아닐 수도 있다.

- 표시 단계 전에 할당된 새로운 힙 메모리
- GC 프로세스 위해 필요한 메타데이터 공간(힙 메모리의 실제 사용량에 비교해 상대적으로 작은 양을 차지)

> GC 프로세스 메타데이터
>
> Go의 가비지 컬렉션 구현에서 ~~객체 자체에 대한 메타데이터~~를 별도로 저장하지 않는다. 대신, Go는 효율적인 메모리 관리를 위해 메타데이터를 별도의 구조에 저장한다.
>
> - Mark bit: 객체가 가비지 컬렉션 과정에서 '살아있는' 상태로 마크되었는지 여부를 나타내는 비트
> - Type Information: 각 객체의 타입에 대한 정보. 타입별 메모리 레이아웃을 파악하고, 포인터가 포함된 메모리 영역을 식별하는 데 필요
> - Extra Bits for Debugging: 디버깅 목적으로 사용

3. GC의 CPU 비용 모델링

- 사이클당 고정 비용
- 라이브 힙 크기에 비례하여 확장되는 한계 비용(margin cost)

> **한계 비용(margin cost)?**
>
> 추가적인 단위 생산이나 활동이 발생할 때 그것이 총 비용에 추가하는 비용
>
> Go GC에서는 라이브 힙의 크기가 증가할 때마다 추가되는 CPU 비용을 의미. 메모리에 존재하는 살아있는 객체들의 양(즉, 라이브 힙의 크기)에 비례하여 증가하거나 감소한다.
>
> - 라이브 힙이 크다(=더 많은 메모리 사용중) -> GC는 더 많은 객체를 스캔하고 마킹 -> CPU 부담 커짐 -> 한계 비용 증가
> - 라이브 힙이 작다 -> GC는 더 적은 객체를 스캔하고 마킹 -> CPU 부담 감소 -> 한계 비용 감소
>
> 점근적으로 말하면, 스윕은 라이브가 아닌 것으로 확인된(예: "죽은") 메모리를 포함한 전체 힙의 크기에 비례하여 작업을 수행해야 하기 때문에, 크기가 커질수록 스윕은 마킹 및 스캐닝보다 더 나쁘다. 그러나 현재 구현에서는 스윕이 마킹 및 스캐닝보다 훨씬 빠르므로, 이 논의에서는 관련 비용을 무시할 수 있다. 고정 비용은 고정 되어 있으므로 관련이 없고, 한계 비용에 영향을 주지 않는다.

이 모델은 간단하지만 효과적으로 GC의 주요 비용을 정확하게 분류한다. 그러나 이 모델은 이러한 비용의 규모나 어떻게 상호작용 하는지에 대해서는 아무 것도 알려주지 않는다. 이를 모델링하려면 여기부터 *정상 상태*라고 하는 다음 상황을 고려해보자.

> *정상 상태*는 인위적으로 보일 수 있지만, 일정한 워크로드 하에서 애플리케이션의 동작을 나타낸다 물론 애플리케이션이 실행되는 동안에도 워크로드가 변경될 수 있지만, 그러나 일반적으로 애플리케이션 동작은 이러한 *정상 상태*가 여러 개 뭉쳐져 있고 그 사이에 일시적인 동작이 있는 것처럼 보인다.
>
> *정상 상태*에서는 라이브 힙에 대해 어떠한 가정도 하지 않는다. 후속 GC 사이클마다 증가할 수도, 줄어들 수도, 동일하게 유지될 수도 있다. 하지만, 앞으로 이어질 설명에서 이러한 모든 상황을 포함하려고 하면 지루하고 설명이 잘 되지 않으므로, 이 가이드에서는 라이브 힙이 일정하게 유지되는 예시를 중심으로 설명하고, GOGC 섹션에서는 라이브 힙이 일정하지 않은 시나리오를 좀 더 자세히 살펴본다.

- 애플리케이션이 새 메모리를 할당하는 속도(초당 바이트 수, bps)는 일정하다.

> 이 할당 속도는 *새 메모리가 살아 있는지 여부와 완전히 별개*라는 점을 이해하는 것이 중요하다. 모두 라이브가 아닐 수 있고, 모두 라이브일 수도 있고, 일부만 라이브일 수도 있다. 게다가 일부 오래된 힙 메모리가 죽을 수도

> 있으므로, 해당 메모리가 활성화된 경우 라이브 힙 크기가 반드시 커지는 것은 아니다.
>
> 이를 보다 구체적으로 설명하려면 처리하는 각 요청에 대해 총 2MiB의 힙 메모리를 할당하는 웹 서비스를 고려해보자. 요청이 전송되는 동안 2MiB 중 최대 512KiB는 라이브 상태로 유지되며, 서비스가 요청 처리를 완료하면 해당 메모리는 모두 죽는다. 이제 단순화를 위해 각 요청을 처음부터 끝까지 처리하는 데 약 1초가 걸린다고 가정해보자. 초당 100개의 요청이 꾸준히 발생하면, 할당 속도는 200MiB/s가 되고, 피크 라이브 힙은 50MiB(512KiB * 100)가 된다.

- 애플리케이션의 객체 그래프는 매번 거의 동일하게 보인다
    - 객체의 크기가 비슷하고
    - 포인터 수가 거의 일정하며
    - 그래프의 최대 깊이는 거의 일정하다

*정상 상태*에서 라이브 힙 크기가 일정할 때, 같은 시간이 지난 후에 GC가 실행된다면, 비용 모델에서 모든 GC 사이클은 동일하게 보일 것이다. *고정된 시간* 동안, 애플리케이션에 의해 *고정된 할당 비율*로, *고정된 양*의 새로운 힙 메모리가 할당되기 때문이다

따라서 라이브 힙 크기가 일정하고, 새로운 힙 메모리가 일정하면, 메모리 사용량은 항상 동일하게 유지된다. 그리고 라이브 힙의 크기가 동일하기 때문에, GC CPU 한계 비용도 동일하며, 고정 비용은 일정한 간격으로 발생한다.

시간상 GC가 나중에 실행된다고 생각해 보자. 그러면 더 많은 메모리가 할당되지만, 각 GC 사이클마다 여전히 동일한 CPU 비용이 발생한다.

> *정상 상태*에서 라이브 힙의 크기가 일정하게 유지된다는 전제 -> 각 GC 사이클마다 동일한 CPU 비용이 발생한다 각 사이클에서 라이브 힙을 스캔하고 정리하는 작업은 비슷하게 유지되므로, GC의 CPU 비용도 대체로 일정하다 그러나 다른 고정된 시간 동안에는 더 적은 수의 GC 사이클이 완료될 수 있고, 전체 CPU 비용이 낮아진다. 더 긴 간격으로 GC가 실행됨 -> 더 적은 수의 GC 사이클 GC가 더 일찍 시작하기로 결정하면 그 반대가 된다. 즉, 더 적은 메모리가 할당되고 CPU 비용이 더 자주 발생한다. GC가 더 일찍 시작 -> 자주 GC 사이클이 발생 -> 전체적으로 더 많은 CPU 비용이 발생 더 작은 메모리 할당에 대해 더 자주 GC를 실행 -> 시간당 GC 실행 횟수가 증가 -> 전체 CPU 비용의 증가

이 상황은 GC가 실제로 실행하는 빈도에 따라 제어되는, GC가 만들 수 있는 *CPU 시간*과 *메모리* 간의 근본적인 트레이드 오프를 나타낸다. 즉, 트레이드오프는 전적으로 GC 빈도에 의해 정의된다.

정의해야 할 세부 사항이 하나 더 남아 있는데, 바로 GC가 언제 시작될지 결정하는 것이다. 이는 특정 *정상 상태*의 GC 빈도(frequency)를 직접 설정하여 *GC가 만들 수 있는 CPU 시간과 메모리 간의 근본적인 트레이드 오프*를 정의한다. Go에서 GC가 언제 시작될지 결정하는 것은 사용자가 제어할 수 있는 주요 매개변수다.

GOGC

높은 수준에서 `GOGC`는 GC CPU와 메모리 간의 균형(trade-off)을 결정한다

각 GC 사이클 이후 목표 힙 크기를 결정함으로써 작동하며, 이는 다음 사이클(현재 GC 사이클 완료 후 다음 GC 사이클 시작 전)에 대한 전체 힙 크기의 목표 값을 설정한다. GC의 목표는 총 힙 크기가 목표 힙 크기를 초과하기 전에 수집 사이클을 완료하는 것이다.

> `GOGC=100`는 프로그램의 활성 메모리 크기가 마지막 GC 사이클 이후 100% 증가했을 때 다음 GC 사이클이 시작됨을 의미한다. 가령, 마지막 GC 후 활성 메모리가 30MiB였다면, 총 메모리 사용량이 60MiB에 도달했을 때 다음 GC가 트리거된다.

> GOGC=off는 GC가 자동으로 실행되지 않음을 의미하며, 이는 메모리 사용량이 메모리 제한까지 증가할 수 있음을 의미한다.

총 힙 크기 = 이전 주기 종료 시의 활성 힙 크기 + 이전 주기 이후 애플리케이션에 의해 할당된 새로운 힙 메모리

한편, 대상 힙 메모리는 다음과 같이 정의된다:

$\text{Target heap memory} = \text{Live heap} + (\text{Live heap} + \text{GC roots}) * GOGC / 100$

> **총 힙 크기 (Total heap size)**
>
> 프로그램 실행 동안 메모리 할당에 따라 변동되는, 실제로 사용 중인 힙 메모리의 현재 크기를 의미. 현재 활성 힙 + 새로 할당된 힙 메모리.
>
> **목표 힙 크기 (Target heap memory)**
>
> GC가 다음 수집 사이클을 시작하기 전에 허용되는 힙 메모리의 최대 크기를 의미. GC가 다음 사이클을 시작하기 전까지 힙이 얼마나 커질 수 있는지를 결정하는 값.
>
> **GC 사이클과 목표 힙 크기**
>
> 1. GC 사이클의 종료: 현재 GC 사이클이 완료되면, 사용중인 힙 메모리의 크기와 GC 루트 집합의 크기를 기반으로 다음 사이클의 목표 힙 크기 계산
> 2. 목표 힙 크기의 설정: GOGC 값에 따라 계산된 목표 힙 크기는 **현재 사용 중인 힙 크기에 특정 비율을 더한 값**이다. 예를 들어, GOGC가 100이라면 목표 힙 크기는 현재 라이브 힙 크기의 2배가 된다.(100%)
> 3. 다음 GC 사이클 시작 전: 이 목표 힙 크기는 다음 GC 사이클이 시작하기 전까지의 기간 동안 적용된다. 즉, 프로그램이 메모리를 계속 할당하면서 실제 힙 크기가 이 목표치에 도달하거나 초과할 때까지 적용된다.
> 4. 다음 GC 사이클의 시작: 실제 힙 크기가 설정된 목표 힙 크기에 도달하면, 다음 GC 사이클이 시작된다. 이때 새로운 목표 힙 크기가 다시 계산되어 다음 사이클을 위한 상한선으로 설정된다.
>
> 따라서 목표 힙 크기는 메모리 사용량 증가에 따라 *GC의 빈도*와 *GC의 효율성*을 조절하는 데 중요한 역할을 한다.
>
> 여기서 목표 힙 크기는 단지 목표일 뿐이며, GC 주기가 해당 대상에서 바로 끝나지 않을 수 있는 몇 가지 이유가 있다.
>
> 1. 우선, 충분히 큰 힙 할당은 단순히 목표를 초과할 수 있다
> 2. 이 가이드에서의 GC 모델을 넘어서, 실제 GC 구현 상세에 따라 다를 수 있다.(지연 시간, 추가 리소스섹션 참고)

예를 들어,

- 라이브 힙 크기: 8MiB
- 고루틴 스택: 1MiB
- 전역 변수에서 포인터 크기: 1MiB
- GOGC -> 다음 GC가 실행되기 전에 할당될 새 메모리 양 계산에 사용된다.
    - 50: 50%, 즉 5MiB가 되고, 총 힙 공간은 15MiB가 된다
    - 100: 100%, 즉 10MiB가 되고, 총 힙 공간은 18MiB가 된다
    - 200: 200%, 즉 20MiB가 된다, 총 힙 공간은 28MiB가 된다

> 이전에는 라이브 힙만 계산했지만, 1.18부터 GOGC는 **루트 집합만 포함**한다. 고루틴 스택의 메모리 양이 매우 적고 라이브 힙 크기가 GC 작업의 다른 모든 소스를 지배하는 경우가 많지만, 프로그램에 수십만 개의 고루틴이 있는 경우 GC가 잘못된 판단을 내리는 경우가 많았다.

> **루트 집합?**
>
> 가비지 컬렉터가 추적하는 객체들의 집합을 의미. GC가 시작할 때 살아있는 객체를 찾기 위한 시작점을 제공하는 객체의 집합으로, 고루틴 스택, 전역 변수, 그리고 현재 실행 중인 고루틴의 스택 프레임에서 접근 가능한 모든 객체를 포함한다.

힙 목표는 GC 빈도를 제어한다. 목표가 클수록 GC가 다른 마크 단계를 시작하기까지 기다릴 수 있는 시간이 길어지고, 그 반대도 마찬가지입니다. 정확한 공식은 추정에 유용하지만 GOGC의 기본 목적, 즉 GC CPU와 메모리 균형의 지점을 선택하는 매개변수 관점에서 생각하는 게 가장 좋다. 핵심 내용은

- **GOGC를 두 배로 늘리면 힙 메모리 오버헤드가 두 배로 늘어나고 GC CPU 비용이 대략 절반으로 줄어든다**
- GOGC를 절반으로 줄이면 힙 메모리 오버헤드가 절반으로 줄어들고 GC CPU 비용이 대략 두배로 늘어난다 전체 설명은 Additional notes on GOGC을 참조.

> **GOGC = 100 (기준 값)**
>
> - 기본 설정
> - 이 설정에서 힙 메모리 오버헤드와 GC CPU 비용은 표준 수준이다
> - 이 상태에서 힙의 크기는 마지막 GC 이후의 라이브 힙 크기에 기반하여 결정된다
>
> **GOGC = 50 (기준의 절반)**
>
> - 힙 메모리 오버헤드가 기준값(100)의 절반으로 줄어들고, GC가 더 자주 발생하게 하여 사용되지 않는 메모리를 더 빠르게 회수하도록 한다
> - 하지만, GC CPU 비용은 기준값(100)의 대략 두 배로 증가한다. 더 자주 발생하는 GC는 더 많은 CPU 시간을 요구하기 때문.
> - 이 설정은 메모리 사용량을 줄이고자 할 때 유용하지만, CPU 리소스 사용이 증가하는 점을 고려해야 한다.
>
> **GOGC = 200 (기준의 두 배)**
>
> - 힙 메모리 오버헤드가 기준값(100)의 두 배로 증가하고, GC가 덜 자주 발생하게 하여 프로그램이 더 많은 메모리를 사용할 수 있도록 한다
> - 반면, GC CPU 비용은 기준값(100)의 대략 절반으로 감소한다. 덜 자주 발생하는 GC는 전체적으로 더 적은 CPU 시간을 소모하기 때문.
> - 이 설정은 CPU 사용을 최소화하려는 경우에 적합하지만, 힙 메모리 사용량이 증가한다는 점을 감안해야 한다.

GOGC는 (모든 Go 프로그램이 인식하는) GOGC 환경 변수를 통해 설정하거나 `runtime/debug` 패키지의 `SetGCPercent` API를 통해 구성할 수 있다.

GOGC=off 설정을 사용하거나 SetGCPercent(-1)을 호출함으로써 완전히 GC를 비활성화할 수 있습니다 (단, 메모리 제한이 적용되지 않는 경우에 한함). 개념적으로, 이 설정은 GOGC를 무한대의 값으로 설정하는 것과 동등하며, GC가 트리거되기 전에 새로 할당되는 메모리의 양에 제한이 없습니다.

(메모리 제한이 적용되지 않는 경우) `GOGC=off`를 설정하거나 `SetGCPercent(-1)`를 호출하여 GC를 완전히 비활성화할 수 있다. GC가 작동하기 전에 할당할 수 있는 새로운 메모리의 양에 제한이 없기 때문에, 이 설정은 개념적으로 GOGC를 무한대로 설정하는 것과 동일하다.

> **메모리 제한이 설정된 경우라면?**
>
> Go의 가비지 컬렉터(GC)는 메모리 사용량을 제한하는 역할을 한다. 즉 프로그램이 사용할 수 있는 최대 메모리 양이 명시적으로 제한되어 있는 경우, GOGC=off로 설정하더라도 Go 런타임은 메모리 사용량이 이 제한을 초과하지 않도

> 록 관리해야 할 수 있다. 이 상황에서 GC가 완전히 비활성화되면, 프로그램이 계속해서 메모리를 할당만 하고 해제하지 않게 되어 결국에는 설정된 메모리 제한에 도달하게 된다. 이러한 상황을 방지하기 위해, Go 런타임이 프로그램의 메모리 사용을 적절히 관리하기 위해 여전히 GC를 활용할 수 있다.

지금까지 논의한 모든 내용을 더 잘 이해하려면 앞서 논의한 GC 비용 모델을 기반으로 구축된 아래의 대화형 시각화를 사용해 보자.

- 이 시각화는 GC가 아닌 작업을 완료하는 데 10초의 CPU 시간이 걸리는 프로그램의 실행을 보여준다.
- 첫 번째 초에는 일부 초기화 단계(라이브 힙을 증가시킴)를 수행한 후 안정된 상태(steady-state)로 정착한다.
- 애플리케이션은 한 번에 20MiB씩 라이브로, 10초 동안 총 200MiB를 할당한다 완료할 유일한 관련 GC 작업은 라이브 힙에서 나오며, 비현실적으로 애플리케이션이 추가 메모리를 사용하지 않는다고 가정한다.

슬라이더를 사용하여 GOGC 값을 조정하여 총 실행 시간과 GC 오버헤드 측면에서 애플리케이션이 어떻게 반응하는지 확인해 보자. 각 GC 사이클은 새로운 힙의 크기가 0이 될 때 종료된다.

> **새로운 힙?**
>
> GC 사이클이 시작된 후에 프로그램에 의해 할당된 메모리를 의미한다. 이 메모리는 아직 가비지 컬렉터에 의해 검사되지 않았으므로, "새로운" 것으로 간주된다.

새 힙이 0으로 떨어지는 데 걸리는 시간은 N번째 사이클의 마킹(사용 중인 메모리를 식별) 단계와 N+1번째 사이클의 스위핑(마킹 단계에서 사용 중이지 않다고 식별된 메모리를 실제로 해제) 단계를 합친 시간이다. 이 가이드에 있는 모든 시각화는 GC가 실행되는 동안 애플리케이션이 일시 중단되었다고 가정하므로, GC의 CPU 비용은 새로운 힙 메모리가 0으로 줄어드는 데 걸리는 시간으로 완전히 표현된다.

> **새로운 힙 메모리가 0으로 줄어드는 시간?**
>
> 마킹과 스윕 단계를 완료하는 데 걸리는, GC가 시작되어 모든 쓰레기 객체를 처리하고 메모리를 정리하는 데까지 걸리는 전체 시간을 의미. 이 시간은 *GC가 수행하는 작업의 양과 복잡성을 반영*한다. 새로운 힙 메모리가 클수록 더 많은 메모리가 GC 사이클 사이에 할당 됨을 의미하고, GC는 더 많은 메모리를 검사하고 처리해야 한다. 이는 CPU가 더 많은 시간을 GC 작업에 할당해야 함을 의미한다. *새로운 힙 메모리가 0으로 줄어드는 시간*이 길수록, GC는 더 많은 CPU 시간을 소모했다고 볼 수 있으므로, GC가 CPU 자원을 사용하는 정도를 나타내는 좋은 지표가 된다.

이는 시각화를 더 단순하게 하기 위한 것일 뿐, 동일한 직관이 여전히 적용된다. X축은 항상 프로그램의 전체 CPU 시간을 표시하도록 조정된다. GC에 의해 사용되는 추가 CPU 시간은 전체 실행 시간을 늘린다.

- `GOGC=50` 경우(GC가 더 발생)
  - GC CPU = 11.2%
  - Peak Mem = 30.0 MiB
  - Peak Live Mem = 20.0 MiB
  - Total: 11.26 s
- `GOGC=100` 경우
  - GC CPU = 6.4%
  - Peak Mem = 40.0 MiB
  - Peak Live Mem = 20.0 MiB
  - Total: 10.68s
- `GOGC=200` 경우(GC가 덜 발생)
  - GC CPU = 3.8%
  - Peak Mem = 60.0 MiB
  - Peak Live Mem = 20.0 MiB

    - Total: 10.39 s

GC는 항상 약간의 CPU 및 최대(peak) 메모리 오버헤드가 발생한다.

- `GOGC` 값 증가?
    - 더 많은 메모리 사용으로 더 적은 횟수의 GC
    - CPU 오버헤드는 감소
    - 최대(peak) 메모리는 라이브 힙 크기에 비례하여 증가
- `GOGC` 값 감소?
    - 더 적은 메모리 사용하지만 잦은 GC
    - CPU 오버헤드는 증가
    - 최대(peak) 메모리 오버헤드 요구량은 감소

> 그래프는 프로그램을 완료하는 데 걸리는 월 클럭 시간(wall-clock time)이 아닌 CPU 시간을 표시한다. 프로그램이 1개의 CPU에서 실행되고 리소스를 최대한 활용한다면 이 수치는 동일하다. 실제 프로그램은 멀티코어 시스템에서 실행되며 항상 CPU를 100% 활용하지 않을 가능성이 높다. 이러한 경우 GC의 월타임 영향(wall-time impact)은 더 낮을 것이다.
>
> Go GC의 최소 총 힙 크기는 4MB이므로, `GOGC`에서 설정한 목표가 이보다 작으면 반올림 된다. 시각화에는 이 세부 사항이 반영되어 있다.
>
> **Wall Clock Time**?
>
> 프로그램이 시작되어 완료될 때까지 **실제 경과한 시간**을 의미하며, 실제로 사용하는 CPU 시간과 다를 수 있다. 특히 요즘에는 여러 프로세스나 스레드가 동시에 실행되며, 이 때문에 한 프로그램이 CPU를 독점적으로 사용하지 않기 때문이다. 따라서 CPU 시간과 실제 경과 시간 사이에 차이가 발생할 수 있다.
>
> **Wall?**
>
> 프로그램이나 작업의 실행에 실제로 걸리는 시간을 측정하는 방식 "월"은 실제 세계의 벽에 걸린 시계를 의미한다. 프로그램이 시작되어 완료되기까지 벽시계로 측정할 수 있는 실제 경과 시간을 가리킨다

다음은 좀 더 동적이고 현실적인 또 다른 예시다. 이 애플리케이션도 GC 없이 완료하는 데 10 CPU 초가 걸리지만, 중간에 정상 상태(steady-state) 할당률이 급격히 증가하고, 첫 번째 단계에서 라이브 힙 크기가 약간 이동한다. 이 예는 라이브 힙 크기가 실제로 변경될 때 정상 상태(steady-state)가 어떻게 보일 수 있는지, 그리고 어떻게 할당률이 높을수록 GC 주기가 더 자주 발생하는지 보여준다.

- `GOGC=50`: 활성 메모리의 50% 만큼의 메모리를 추가로 할당할 수 있음
    - GC CPU = 40.3%
    - Peak Mem = 30.0 MiB = 20.0 MiB + (20.0 MiB * 0.5)
    - Peak Live Mem = 20.0 MiB
    - Total: 16.74 s
- `GOGC=100`: 활성 메모리의 100% 만큼의 메모리를 추가로 할당할 수 있음
    - GC CPU = 25.3%
    - Peak Mem = 40.0 MiB = 20.0 MiB + (20.0 MiB * 1)
    - Peak Live Mem = 20.0 MiB
    - Total: 13.39 s
- `GOGC=200`: 활성 메모리의 200% 만큼의 메모리를 추가로 할당할 수 있음
    - GC CPU = 14.5%
    - Peak Mem = 60.0 MiB = 20.0 MiB + (20.0 MiB * 2)

    - Peak Live Mem = 20.0 MiB
    - Total: 11.70 s

Memory limit

Go 1.19까지 GOGC는 GC의 동작을 수정하는 데 사용할 수 있는 유일한 매개변수였다. 절충점(trade-off)을 설정하는 방법으로는 훌륭하게 작동하지만, 사용 가능한 메모리가 유한하다는 점을 고려하지 않았다. 가령 라이브 힙 크기에 일시적인 급증이 있을 때 어떤 일이 발생하는지 생각해 보자. GC는 라이브 힙 크기에 비례하여 전체 힙 크기를 결정하기 때문에, 일반적으로 높은 GOGC 값이 더 나은 절충안(trade-off)을 제공할 수 있더라도, GOGC는 반드시 그 최대 라이브 힙(가령 일시적인 급증한 순간의 힙) 크기에 맞게 구성되어야 한다 .

> **GC는 라이브 힙 크기에 비례하여 전체 힙 크기를 결정한다**
>
> GC는 프로그램이 사용하는 메모리(힙 메모리)를 관리한다.
>
> - 라이브 힙 크기: 프로그램이 실제로 사용하고 있는 메모리의 크기를 의미
> - 전체 힙 크기: GC가 관리하는 메모리의 전체 크기를 의미
>
> 프로그램이 사용하는 메모리가 늘어날수록 GC가 관리해야 할 전체 메모리 영역도 커진다. 예를 들어, 프로그램이 10MB의 라이브 힙을 사용하고 있다면, 예를 들어 GC는 이에 비례하여 20MB의 전체 힙 크기를 할당할 수 있다.
>
> **일반적으로 높은 GOGC 값이 더 나은 절충안(trade-off)을 제공할 수 있더라도**
>
> GC는 메모리를 관리하는 과정에서 CPU 자원을 사용한다.
>
> 1. GOGC 값이 클수록 사용할 수 있는 힙 메모리 크기가 커지고
> 2. 사용할 수 있는 힙 메모리가 커지면서 GC 주기가 길어짐에 따라 GC가 더 적게 실행되고
> 3. GC가 더 적게 실행되면 CPU 사용량이 줄어들고
> 4. 그만큼 CPU 시간을 다른 작업에 할당할 수 있게 된다
>
> 특히 CPU 사용이 중요한 작업이나 고성능을 요구하는 애플리케이션에서는 GC로 인한 CPU 사용 시간을 최소화하는 것이 중요할 수 있다.
>
> 반면, GOGC 값이 클수록 사용되지 않는 메모리가 더 오랫동안 시스템에 남아 있게 되므로 메모리 사용량은 증가한다. 메모리 리소스가 충분하고, 메모리 사용량이 성능에 미치는 영향이 CPU 사용량보다 낮은 경우에는 큰 GOGC 값이 효과적일 수 있다.

아래 시각화는 이러한 일시적인 힙 스파이크 상황을 보여준다.

- GOGC=50: 활성 메모리의 50% 만큼의 메모리를 추가로 할당할 수 있음
    - GC CPU = 11.9%
    - Peak Mem = 45.0 MiB = 30.0 MiB + (30.0 MiB * 0.5)
    - Peak Live Mem = 30.0 MiB
    - Total: 11.35 s
- GOGC=100: 활성 메모리의 100% 만큼의 메모리를 추가로 할당할 수 있음
    - GC CPU = 6.3%
    - Peak Mem = 60.0 MiB = 30.0 MiB + (30.0 MiB * 1)
    - Peak Live Mem = 30.0 MiB
    - Total: 10.67 s
- GOGC=200: 활성 메모리의 200% 만큼의 메모리를 추가로 할당할 수 있음
    - GC CPU = 3.3%

    - Peak Mem = 90.0 MiB = 30.0 MiB + (30.0 MiB * 2)
    - Peak Live Mem = 30.0 MiB
    - Total: 10.34 s

예시 워크로드가 60MiB가 조금 넘는 메모리를 사용할 수 있는 컨테이너에서 실행되는 경우, 나머지 GC 사이클에서 여분의 메모리를 사용할 수 있는 가용 메모리가 있더라도 `GOGC`를 100 이상으로 늘릴 수 없다. 또한 일부 애플리케이션에서는 이러한 일시적인 피크가 드물고 예측하기 어려워서, 때때로 피할 수 없고 잠재적으로 비용이 많이 드는 메모리 부족 상태가 발생할 수 있다.

이것이 바로 Go가 1.19 릴리스에서 런타임 `메모리 제한` 설정에 대한 지원을 추가한 이유다. `메모리 제한`은 모든 Go 프로그램이 인식하는 `GOMEMLIMIT` 환경 변수를 통해 구성하거나 `runtime/debug` 패키지에서 사용할 수 있는 `SetMemoryLimit` 함수를 통해 구성할 수 있다.

이 `메모리 제한`은 *Go 런타임이 사용할 수 있는 전체 메모리*에 대한 최대값을 설정한다. 즉 Go 런타임에 의해 관리되는 메모리 영역의 상한선을 설정한다. `메모리 제한`에 포함된 특정 메모리 집합은 `runtime.MemStats` 타입을 사용하여 다음 표현식으로 정의된다.

$\text{Sys(=runtime.MemStats.Sys)} - \text{HeapReleased(=runtime.MemStats.HeapReleased)}$

> **포함된 특정 메모리 집합?**
>
> Go 런타임이 관리하는 메모리 영역 중 `메모리 제한`에 영향을 받는 부분을 의미한다. 이는 힙 메모리, 스택 메모리, 시스템 메모리 등 Go 런타임에 의해 관리되는 주요 메모리 영역을 포함한다. `메모리 제한`은 특히 `runtime` 패키지의 `Sys` 필드에 영향을 미친다. `Sys`는 Go 런타임이 운영체제로부터 얻은 총 메모리 양을 나타내며, 힙, 스택, 그리고 다른 내부 데이터 구조들을 포함한다.
>
> **`runtime.MemStats`?**
>
> Go의 runtime 패키지에 있는 구조체로, Go 프로그램에 의해 할당된 메모리의 통계를 제공한다. 할당된 메모리, 사용 중인 메모리, 가용 메모리 등 Go 프로그램의 메모리 사용 패턴에 대한 세부 정보를 제공한다.
>
> - `Alloc`: 현재 할당된 힙 메모리의 크기. `HeapAlloc`과 같다.
> - `TotalAlloc`: 프로그램 시작 이후 할당된 총 누적 메모리의 크기. `Alloc` 그리고 `HeapAlloc`과는 달리 오브젝트가 해제되어도 감소하지 않는다.
> - `Sys`: Go 프로그램에 의해 사용되는 총 시스템 메모리의 양. `*Sys` 항목들(`StackSys`, `MSpanSys` 등)의 합과 같다. heap, stack 그리고 다른 내부 자료 구조를 위해 고 런타임에 의해 예약된 가상 주소 공간을 측정한다.
> - `Lookups`: 힙 메모리 할당을 위해 수행된 조회(lookups) 횟수
> - `Mallocs`: 할당된 힙 객체의 누적 개수. 라이브 오브젝트 수는 `Mallocs` - `Frees`
> - `Frees`: 해제된 힙 객체의 누적 개수.
> - `HeapAlloc`: 할당된 힙 객체의 바이트.
> - `HeapSys`: OS에서 얻은 힙 메모리의 바이트. 힙을 위해 예약된 가상 주소 공간의 크기를 측정한다. 예약되었지만 사용되지 않은 공간을 포함한다.
> - `HeapReleased`: OS로 반환된 물리 메모리의 바이트. OS로 반환되었지만 아직 힙으로 확보되지 않은 유휴 스팬의 힙 메모리가 포함된다.

또는 마찬가지로 `runtime/metrics` 패키지 경우 다음 표현식으로 정의된다.

$\text{/memory/classes/total:bytes} - \text{/memory/classes/heap/released:bytes}$

Go GC는 사용하는 힙 메모리의 양을 명시적으로 제어할 수 있으므로, 이 메모리 제한과 Go 런타임이 사용하는 다른 메모리의 양에 기반하여 총 힙 크기를 설정한다.

아래의 시각화는 GOGC 섹션에서 설명된 동일한 단일 단계(single-phase)의 정상 상태(steady-state)를 유지하는 작업 부하를 보여준다. 하지만 이번에는 Go 런타임에서 발생하는 추가적인 10 MiB의 오버헤드와 조절 가능한 메모리 제한이 포함되어 있다. GOGC와 메모리 제한을 조정해보고 어떤 변화가 일어나는지 살펴보자.

> **the same single-phase steady-state workload from the GOGC section?**
>
> 앞서 GOGC 섹션에서 전제했던, "변동 없이 일정한 상태를 유지하는 하나의 연속된 작업 부하"를 의미한다.

- GOGC=100: 활성 메모리의 100% 만큼의 메모리를 추가로 할당할 수 있음
  - Memory Limit: 100 MiB
    - GC CPU = 6.4%
    - Peak Mem = 50.0 MiB => (20 MiB * 2) + 10 MiB
    - Peak Live Mem = 20.0 MiB => GOGC는 100이므로 허용되는 총 힙 메모리 크기는 두 배인 40 MiB가 된다
    - Other Mem = 10.0 MiB
  - Memory Limit: 50 MiB
    - GC CPU = 6.4%
    - Peak Mem = 50.0 MiB => (20 MiB * 2) + 10 MiB
    - Peak Live Mem = 20.0 MiB => GOGC는 100이므로 허용되는 총 힙 메모리 크기는 두 배인 40 MiB가 된다
    - Other Mem = 10.0 MiB

메모리 제한(memory limit)이 GOGC에 의해 결정된 최대 메모리(peak memory, GOGC 100의 경우 42MiB) 아래로 낮아지면 최대 메모리를 메모리 제한 내로 유지하기 위해 GC가 더 자주 실행된다.

> **계산식**
>
> - peak_live_mem: 활성 메모리 MiB
> - other_mem: 기타 메모리 MiB
> - gogc: GOGC 값
>
> 피크 메모리 = 활성 메모리(peak_live_mem) + 활성 메모리의 100%(gogc / 100 *peak_live_mem) + 기타 메모리(other_mem)

일시적인 힙 스파이크에 대한 이전 예로 돌아가서, 메모리 제한을 설정하고 GOGC를 켜면, *메모리 제한 위반 없음*과 *리소스 경제성이 향상*되는 두 가지 장점을 모두 얻을 수 있다.

- GOGC=100: 활성 메모리의 100% 만큼의 메모리를 추가로 할당할 수 있음
  - Memory Limit: 100 MiB
    - GC CPU = 6.3%
    - Peak Mem = 60.0 MiB => (30 MiB * 2) + 0 MiB
    - Peak Live Mem = 30.0 MiB => GOGC는 100이므로 허용되는 총 힙 메모리 크기는 두 배인 60 MiB가 된다
  - Memory Limit: 50 MiB
    - GC CPU = 6.3%
    - Peak Mem = 50.0 MiB => (30 MiB * 2) + 0 MiB => 60 MiB, 하지만 메모리 제한이 50 MiB이므로 50 MiB가 된다

        - Peak Live Mem = 30.0 MiB => GOGC는 100이므로 허용되는 총 힙 메모리 크기는 두 배인 60 MiB가 된다
- GOGC=off: GC를 완전히 꺼버린다. 메모리 사용량은 메모리 제한까지 증가할 수 있다.
    - Memory Limit: 100 MiB
        - GC CPU = 1.1%
        - Peak Mem = Memory Limit = 100.0 MiB
        - Peak Live Mem = 20.0 MiB
    - Memory Limit: 50 MiB
        - GC CPU = 2.9%
        - Peak Mem = Memory Limit = 50.0 MiB
        - Peak Live Mem = 30.0 MiB
    - Memory Limit: 32 MiB
        - GC CPU = 10.0%
        - Peak Mem = 32.0 MiB
        - Peak Live Mem = 30.0 MiB
    - Memory Limit: 30 MiB
        - GC CPU = 55.2%
        - Peak Mem = 30.1 MiB
        - Peak Live Mem = 30.0 MiB

> **GOGC=off 경우 "Peak Live Mem" 값은 왜 다른가?**

GOGC와 메모리 제한의 특정 값에 따라, 최대 메모리 사용량은 설정된 메모리 제한치에서 멈추지만, 프로그램의 나머지 실행 부분은 여전히 GOGC에 의해 설정된 전체 힙 크기 규칙을 따른다.

이 관찰로부터 또 다른 흥미로운 사실이 나타나는데,GOGC가 꺼져 있을 때에도 메모리 제한은 여전히 존중된다는 것이다. 사실, 이 특정 설정(GOGC=off)은 일정한 메모리 제한을 유지하기 위해 필요한 최소한의 가비지 컬렉터 빈도를 설정하므로, *리소스 경제성을 극대화(maximization of resource economy)*한다. 이 경우, 프로그램의 *모든* 실행 과정에서 힙 크기가 메모리 제한에 도달할 때까지 증가한다.

> **GOGC가 꺼져 있을 때에도 메모리 제한은 여전히 존중된다?**
>
> Go 런타임은 GOGC가 꺼져 있어도 메모리 사용량을 모니터링하고, 설정된 메모리 제한을 넘지 않도록 관리한다.
>
> **자원의 경제성을 극대화(maximization of resource economy)?**
>
> GOGC 값을 낮추면? GC가 더 자주 실행되어 불필요한 메모리를 더 적극적으로 회수하므로 CPU 사용량이 증가할 수 있다. GOGC 값을 높이면? CPU 시간을 소모하는 GC의 실행 빈도가 감소하여 CPU 사용량을 절약할 수 있지만, 메모리 사용량이 증가할 수 있다. 즉, 적절한 GOGC 설정은 프로그램의 메모리 사용과 CPU 사용 사이의 균형을 최적화하여 리소스 경제성을 향상시킬 수 있다는 의미.
>
> **GOGC=off 장점**
>
> 가비지 컬렉션에 의한 성능 저하를 줄일 수 있으므로, 성능이 중요한 애플리케이션에 유리하다.
>
> **GOGC=off 단점**
>
> 메모리 사용량이 많은 애플리케이션에서는 메모리 제한에 더 빨리 도달할 수 있으며, 이로 인해 메모리 부족 문제가 발생할 수 있다. 또한 정기적인 가비지 컬렉션은 메모리에서 더 이상 사용되지 않는 객체들을 정리하는데, 메모리 누수가 있을 경우 GC 빈도가 줄어들면 사용되지 않는 메모리가 점차 증가하는 경향을 보이게 된다.

메모리 제한이 강력한 도구인 것은 분명하지만, 메모리 제한을 사용하는 데는 대가가 따르며, 그렇다고 해서 GOGC의 유용성이 무효화되는 것도 아니다.

라이브 힙이 충분히 커져서 총 메모리 사용량이 메모리 제한에 가까워지면 어떤 일이 발생하는지 생각해 보자. 위의 정상 상태(steady-state) 시각화에서 GOGC를 끈 다음 메모리 제한을 점점 더 낮춰서 어떤 일이 일어나는지 살펴보자. GC가 불가능한 메모리 제한(Memory Limit 값이 Peak Mem 값보다 작은 경우)을 유지하기 위해 지속적으로 실행되므로, 애플리케이션에 걸리는 총 시간이 무제한으로 증가하기 시작하는 것을 알 수 있다.

지속적인 GC 사이클로 인해 프로그램이 합리적인 진전을 이루지 못하는 이러한 상황을 쓰레싱(thrashing)이라고 한다. 이는 프로그램을 사실상(effectively, 실질적으로) 정지시키기 때문에 특히 위험하다. 더 심각한 문제는, 충분히 큰 일시적인 힙 스파이크로 인해 프로그램이 무한정 멈출 수 있다는, GOGC를 사용해서 피하려고 했던 것과 똑같은 상황이 발생할 수 있다는 것이다. 일시적 힙 스파이크 시각화에서 메모리 제한(약 30MB 이하)을 줄여보고, 최악의 동작이 힙 스파이크에서 구체적으로 어떻게 시작되는지 확인해 보자.

> **스래싱(thrashing)?**
>
> 프로그램이 가비지 컬렉션(GC) 등의 메모리 관리 작업에 지나치게 많은 시간을 소비하여 실질적인 작업 수행이 어려워지는 상태를 의미

많은 경우, 무기한 중단은 훨씬 더 빠른 장애를 초래하는 메모리 부족 상태보다 더 나쁘다.

이러한 이유로 메모리 제한은 **유연(soft)**하게 정의된다. Go 런타임은 모든 상황에서 이 **메모리 제한을 유지한다는 것을 보장하지 않으며, 합리적인 노력**만을 약속한다.

> 런타임은 항상 이 메모리 제한을 엄격하게 준수하지 않고, 메모리 제한에 도달했을 때 적절한 조치를 취하되, 특정 상황에서는 이 제한을 초과할 수도 있다.

이러한 메모리 제한의 완화는 스래싱 현상을 피하는 데 중요한데, 이는 GC에 탈출구를 제공하기 때문이다. GC에 너무 많은 시간을 소비하는 것을 피하기 위해 메모리 사용이 메모리 제한을 초과하도록 허용한다.

이는 내부적으로 GC가 일정 시간 동안 사용할 수 있는 CPU 시간에 상한을 설정함으로써 동작한다.(매우 짧은 CPU 사용량의 급격한 변동에 대해 일부 *후행성*이 적용된다)

> **GC CPU 시간에 상한을 설정?**
>
> 시스템이 너무 많은 시간을 GC에 할애하는 것을 방지하여 GC가 프로그램 전체 성능에 지나치게 영향을 미치지 않도록 하기 위함. GC가 설정된 CPU 시간을 초과하면, GC 작업이 일시적으로 지연될 수 있다.
>
> **후행성(hysteresis)?**
>
> 시스템이나 장치가 외부 환경의 변화에 따라 자신의 상태를 변화시키는 데 있어서, 그 변화의 속도가 너무 빠르지 않도록 하는 것. 즉, 변화에 즉각적으로 반응하지 않고 *일정한 지연이나 편차를 가지고 반응하는 특성*을 말한다. 예를 들어, 온도 조절 시스템에서 설정 온도에 도달했을 때 즉시 반응하지 않고, 일정한 범위 내에서 온도가 변동하는 것을 허용하며, 이로 인해 시스템이 자주 켜지고 꺼지는 것을 방지한다.

이 제한(GC의 CPU 시간 제한)은 현재 2 * GOMAXPROCS CPU 초 동안 대략 50% 정도로 설정되어 있다.

> **GC의 CPU 시간을 2 * GOMAXPROCS CPU 초 동안 대략 50% 정도로 설정**
>
> GC는 프로그램이 사용할 수 있는 CPU 리소스의 절반까지만 사용할 수 있으며, 이는 GOMAXPROCS에 따라 달라진다

> - 2 * GOMAXPROCS CPU 초: *가비지 컬렉터(GC)가 사용할 수 있는 CPU 시간의 총 기간*으로, CPU 시간 제한이 적용되는 기간이다.
> - GOMAXPROCS: *Go 프로그램이 사용할 수 있는 CPU 코어의 최대 수*를 의미하며, 이 값을 2배한 시간 동안 GC가 CPU 자원을 사용할 수 있음을 나타낸다.
> - 대략 50% 정도로 설정: 는 이 시간 동안 GC가 사용할 수 있는 CPU 자원의 최대 비율을 의미한다.

GC CPU 시간 제한의 결과 GC 작업은 지연되는 반면, Go 프로그램은 메모리 제한을 넘어서 새로운 힙 메모리 할당을 지속할 수 있다.

> **Go 프로그램은 메모리 제한을 넘어서 새로운 힙 메모리 할당을 지속?**
>
> GC가 충분히 빠르게 작동하지 않아 메모리가 충분히 정리되지 않는 경우, 메모리 사용량은 설정된 제한을 넘어서게 된다. 이는 Go 런타임이 프로그램의 성능을 유지하기 위해 일시적으로 메모리 제한을 초과하도록 허용하는 것을 의미한다.(후행성) 이러한 전략은 스래싱 현상을 방지하고, 프로그램이 GC에 의해 과도하게 영향을 받지 않도록 하기 위함이다.

50% GC CPU 제한의 배경에는 사용 가능한 메모리가 충분한 프로그램에 대한 최악의 영향을 고려한 직관이 있다. 메모리 제한을 실수로 너무 낮게 설정한 경우, GC가 CPU 시간의 50% 이상을 사용할 수 없기 때문에 프로그램의 속도는 최대 2배까지 느려질 수 있다. 이는 메모리 제한을 너무 낮게 설정한 경우에도 프로그램이 계속 실행되도록 보장한다.

Suggested uses

메모리 제한은 강력한 도구이고, Go 런타임은 오용으로 인한 최악의 동작을 완화하기 위한 조치를 취하지만, 여전히 신중하게 사용하는 것이 중요하다. 다음은 메모리 제한이 가장 유용하고 적용 가능한 경우와 득보다 실이 큰 경우에 대한 간단한 조언 모음이다.

메모리 제한이 유용한 경우

1. Go 프로그램의 실행 환경이 전적으로 사용자가 제어할 수 있고, Go 프로그램만이 특정 리소스 집합(예: 컨테이너 메모리 제한과 같은 일종의 메모리 예약)에 액세스할 수 있는 유일한 프로그램인 경우. 예를 들어 사용 가능한 메모리가 고정된 컨테이너에 웹 서비스를 배포하는 경우. 이 경우, Go 런타임이 인식하지 못하는 메모리 소스를 위해 추가로 5~10%의 헤드룸(headroom)을 남겨두는 것이 경험상 좋다.

2. 변화하는 조건에 맞춰 메모리 제한을 실시간으로 자유롭게 조정할 수 있는 경우. 예를 들어, C 라이브러리가 실질적으로 훨씬 더 많은 메모리를 사용해야 하는 cgo 프로그램.

메모리 제한이 유용하지 않는 경우

1. Go 프로그램이 제한된 메모리의 일부를 다른 프로그램과 공유해야 하고, 그 프로그램들이 일반적으로 Go 프로그램과 독립적일 경우, GOGC=off와 메모리 제한을 함께 설정하지 않아야 한다. 대신 바람직하지 않은 일시적인 동작을 억제하는 데 도움이 될 수 있는 메모리 제한을 유지하되, 평균적인 경우에 합리적인 작은 값으로 GOGC를 설정한다.
    - 다른 프로그램들을 위해 메모리를 "예약"하려는 유혹이 있을 수 있지만, 프로그램이 완전히 동기화되지 않는 한(예: Go 프로그램이 일부 하위 프로세스를 호출하고 수신자 실행되는 동안 블록되는 경우) 두 프로그램 모두 더 많은 메모리를 필요로 하므로 결과의 안정성이 떨어진다. Go 프로그램이 메모리가 필요하지 않을 때 메모리를 적게 사용하도록 하면 전체적으로 더 안정적인 결과를 생성할 수 있다.
    - 이 조언은 한 머신에서 실행 중인 컨테이너의 메모리 제한 합계가 머신에서 사용할 수 있는 실제 물리적 메모리를 초과할 수 있는 *과다할당(오버커밋)* 상황에도 적용된다.

> 예를 들어, 한 서버에서 여러 컨테이너화된 애플리케이션을 실행하고 있고, 각각의 애플리케이션이 고유한 메모리 제한을 가지고 있다고 가정해 보자. 어떤 Go 애플리케이션을 `GOGC=off`로 설정하면 Go 프로그램이 GC를 수행하지 않아 메모리 사용량이 점점 증가하고, 다른 애플리케이션의 성능에 부정적인 영향을 줄 수 있다.
>
> 가령, 서버 메모리가 4GB, `GOGC=off`이고 메모리 제한이 2GB인 경우:
>
> 1. 메모리 사용 시작: 애플리케이션 시작 시, 필요한 메모리를 할당받기 시작한다. 초기 로딩, 데이터 처리 등으로 메모리 사용량이 점차 증가할 수 있다.
> 2. 메모리 사용량 증가: `GOGC=off`로 설정되어 있기 때문에, 사용되지 않는 객체에 대한 **메모리 해제가 자동으로 이루어지지 않는다**. 따라서 애플리케이션에 의해 메모리 할당이 계속되면 메모리 사용량은 점점 증가한다.
> 3. 메모리 제한 도달: 애플리케이션의 메모리 사용량이 2GB에 도달하면, 더 이상 새로운 메모리를 할당받을 수 없다. 이 시점에서 추가 메모리 할당 요청이 있을 경우, 메모리 할당 실패 오류가 발생할 수 있다.
> 4. 성능 영향: 메모리 제한에 도달하면, 애플리케이션은 필요한 작업을 수행하기 위한 충분한 메모리를 확보하지 못할 수 있다. 이는 성능 저하 또는 애플리케이션 오류로 이어질 수 있다.

2. 제어할 수 없는 실행 환경에 배포할 때, 특히 프로그램의 메모리 사용량이 입력에 비례하는 경우. 좋은 예로 CLI 도구나 데스크톱 애플리케이션을 들 수 있다. 어떤 종류의 입력이 제공될지, 시스템에서 사용 가능한 메모리가 얼마나 될지 불분명한 상황에서 프로그램에 메모리 제한을 설정하면 혼란스러운 충돌과 성능 저하로 이어질 수 있다. 또한 고급 최종 사용자는 원하는 경우 언제든지 메모리 제한을 설정할 수 있다.

3. 프로그램이 이미 환경의 메모리 제한에 근접했을 때, 메모리 부족 상태를 피하기 위해 메모리 제한을 설정하지 말 것. 메모리 부족 위험을 애플리케이션의 심각한 속도 저하 위험으로 효과적으로 대체할 수 있지만, 스래싱을 완화하기 위한 Go의 노력에도 불구하고, 좋은 거래가 되지 않는 경우가 많다. 이러한 경우, 환경의 메모리 제한을 늘리거나(그리고 잠재적으로 메모리 제한을 설정) `GOGC`를 줄이는 것이 훨씬 더 효과적이다.(스래싱 완화보다 훨씬 더 깔끔한 절충안을 제공한다).

Latency

이 문서의 시각화에서는 GC가 실행되는 동안 애플리케이션이 일시 중지된 것으로 모델링 됐다. 이러한 방식으로 동작하는 GC 구현이 실제로 존재하며, 이를 "중지형"(stop-the-world) GC라고 한다.

그러나 Go의 GC는 완전한 '세계를 멈추는(stop-the-world)' 방식이 아니며, 대부분의 작업을 애플리케이션과 동시에 수행한다. 이렇게 동시에 수행하는 목적은 애플리케이션의 지연시간을 줄이기 위함이다. 구체적으로, 단일 계산 단위(가령 웹 요청 등)의 시작부터 끝까지 걸리는 시간을 줄인다. 지금까지 이 문서는 주로 애플리케이션의 처리량(가령 초당 처리되는 웹 요청 수 등)을 고려했다.

GC 사이클 섹션의 각 예제는 실행 중인 프로그램의 총 CPU 기간에 초점을 맞췄다. 그러나 이러한 기간(프로그램의 실행 중인 총 CPU 기간)은, 예를 들어 웹 서비스의 경우 의미가 훨씬 덜하다. 웹 서비스의 경우 처리량(즉, 초당 쿼리 수)도 여전히 중요하지만, 주로 각 개별 요청의 지연 시간이 더 중요하다.

> 사용자는 자신의 요청에 대한 응답을 가능한 빨리 받기를 원하기 때문에, 지연 시간이 길어지면 사용자 만족도가 떨어질 수 있다. 따라서 각 요청의 지연 시간을 최소화하는 것이 종종 더 중요한 고려 사항이 된다.

지연 시간 측면에서 볼 때, stop-the-world GC는 마크와 스윕 단계를 모두 실행하는 데 상당한 시간이 소요될 수 있으며, 이 기간 동안 애플리케이션은 물론, 웹 서비스의 경우 진행 중(in-flight)인 모든 요청이 더 이상 진행될 수 없. 반면, Go의 GC는 전체 애플리케이션의 일시 정지 시간을 힙(heap)의 크기에 비례하게 만들지 않으며, 핵심 추적 알고리즘은 애플리케이션이 활발하게 실행 중일 때 수행된다. (일시 정지는 알고리즘적으로 `GOMAXPROCS`에 더 크게 연관되어 있지만, 대부분의 경우 실행 중인 고루틴을 멈추는 데 걸리는 시간에 의해 좌우된다) (애플리케이션 실행중에 가비지 컬렉션이)동시에 수집하는 것은 비용이 든다. 실제로는 동등한 stop-the-world 가비지 컬렉터보다 처리량이 낮을 수 있다. 하지만, 낮은 지연 시간이 반드시 처리

량 감소를 의미하는 것은 아니다. 중요한 점은, Go 가비지 컬렉터의 성능이 시간이 지남에 따라 지속적으로 향상되고 있으며, 이는 지연 시간뿐만 아니라 처리량에서도 개선되고 있다는 것이다.

Go의 현재 GC의 동시성은 이 문서에서 논의된 어떤 것도 무효화하지 않는다. 어떤 설명도 이 디자인 선택에 의존하지 않는다. GC 빈도는 여전히 GC가 처리량을 위해 CPU 시간과 메모리 사이에서 균형을 맞추는 주요 방식이며, 실제로 지연 시간에서도 이 역할을 담당한다. 이는 GC에 대한 대부분의 비용이 마크 단계가 활성화되어 있는 동안 발생하기 때문이다.

여기서 중요한 점은 *GC 빈도를 줄이면 지연 시간도 향상*될 수 있다는 것이다. 이는 `GOGC` 그리고/또는 메모리 제한 증가와 같은 튜닝 매개변수 수정을 통해 GC 빈도 감소뿐만 아니라, 최적화 가이드에 설명된 최적화에도 적용된다.

그러나 지연 시간은 단순한 비용의 합이 아니라 프로그램의 순간적인 실행의 산물이기 때문에 처리량보다 이해하기가 더 복잡하다. 결과적으로, 지연 시간과 GC 빈도 사이의 연관성은 덜 직접적이다. 더 자세히 알아보고자 하는 분들을 위해 지연 시간의 원인에 대한 목록은 다음과 같다.

- GC가 마크 단계와 스윕 단계 사이를 전환할 때 잠시 stop-the-world 발생
- GC가 마크 단계에 있을 때 CPU 리소스의 25%를 사용하기 때문에 스케줄링이 지연된다
- 높은 할당률에 대응하여 GC를 지원하는 사용자 고루틴
    - Go의 가비지 컬렉터는 메모리 할당률이 높을 때, 프로그램의 고루틴이 GC 작업에 참여하여 GC의 효율을 높이는 메커니즘을 가지고 있다.
    - 즉, 프로그램의 실행 중인 고루틴이 가비지 컬렉션 프로세스에 일정 부분 참여한다는 의미
    - 사용자가 직접적으로 이를 제어하거나 코드로 구현하는 것은 아니고, 메모리 할당률이 급격히 증가하면 Go 런타임이 자동으로 고루틴을 활용하여 GC 작업을 분산시켜 처리한다
    - 이는 고루틴이 직접적으로 가비지 컬렉션 작업을 수행한다는 의미가 아니다. 고루틴의 실행을 일시 중지하여 가비지 컬렉션(GC) 프로세스에 자원을 할당한다는 의미.
- GC가 마크 단계에 있는 동안 추가적인 작업이 필요한 포인터 쓰기(수정)
    - GC의 마크 단계에서는 메모리의 객체들이 여전히 사용 중인지(살아있는지) 여부를 체크하는데, 이 단계에서 포인터를 변경하는 경우 변경된 포인터가 가리키는 객체가 GC 과정에서 올바르게 처리되도록 추가적인 작업이 필요하다.
    - 포인터가 새로운 객체를 가리키게 되었을 때, 그 객체가 GC에 의해 잘못 회수되지 않도록 보장하기 위한 조치
- 루트를 검사하려면 실행 중인 고루틴을 일시 중단해야 한다.

이러한 지연 시간은 추가 작업이 필요한 포인터 쓰기를 제외하고 실행 추적에서 확인할 수 있다.

Additional resources

위에 제시된 정보는 정확하지만, Go GC 설계의 비용과 trade-off를 완전히 이해하기에는 세부적인 내용이 부족하다. 자세한 내용은 다음 추가 리소스를 참조.

- The GC Handbook: An excellent general resource and reference on garbage collector design.
- TCMalloc: Design document for the C/C++ memory allocator TCMalloc, which the Go memory allocator is based on.
- Go 1.5 GC announcement: The blog post announcing the Go 1.5 concurrent GC, which describes the algorithm in more detail.
- Getting to Go: An in-depth presentation about the evolution of Go's GC design up to 2018.
- Go 1.5 concurrent GC pacing: Design document for determining when to start a concurrent mark phase.
- Smarter scavenging: Design document for revising the way the Go runtime returns memory to the operating system.

- Scalable page allocator: Design document for revising the way the Go runtime manages memory it gets from the operating system.
- GC pacer redesign (Go 1.18): Design document for revising the algorithm to determine when to start a concurrent mark phase.
- Soft memory limit (Go 1.19): Design document for the soft memory limit.

A note about virtual memory

이 가이드는 주로 GC의 물리적 메모리 사용에 초점을 맞추었지만, 이것이 정확히 무엇을 의미하며 가상 메모리(일반적으로 `top` 같은 프로그램에서 VSS로 표시)와 어떻게 비교되는지에 대한 질문이 자주 제기된다.

물리적 메모리는 대부분의 컴퓨터에서 실제 물리적 RAM 칩에 들어 있는 메모리이다. 가상 메모리는 프로그램을 서로 격리하기 위해 운영 체제에서 제공하는 물리적 메모리에 대한 추상화이다. 또한 일반적으로 프로그램이 실제 주소에 전혀 매핑되지 않는 가상 주소 공간(virtual address space)을 예약하는 것도 허용된다.

가상 메모리는 운영 체제에서 유지 관리하는 매핑일 뿐이므로, 일반적으로 물리적 메모리와 매핑되지 않는 대용량 가상 메모리를 예약하는 것이 매우 저렴하다.

Go 런타임은 일반적으로 몇 가지 방식으로 가상 메모리 비용에 대한 이러한 관점에 의존한다.

- Go 런타임은 매핑된 가상 메모리를 절대 삭제하지 않는다. 대신 대부분의 운영 체제가 제공하는 특수 작업을 사용하여 일부 가상 메모리 범위와 관련된 물리적 메모리 리소스를 명시적으로 해제한다. 이 기술은 메모리 제한을 관리하고 Go 런타임에 더 이상 필요하지 않은 메모리를 운영 체제에 반환하기 위해 명시적으로 사용된다. 또한 Go 런타임은 더 이상 필요하지 않은 메모리를 백그라운드에서 지속적으로 해제한다. 자세한 내용은 추가 리소스를 참조.
- 32비트 플랫폼에서 Go 런타임은 메모리 단편화 문제를 제한하기 위해 힙을 위한 주소 공간을 최소 128 MiB에서 최대 512 MiB까지 미리 예약한다.
- Go 런타임은 여러 내부 데이터 구조들을 구현하는 과정에서 대규모의 가상 메모리 주소 공간 예약을 사용한다. 64비트 플랫폼에서는 이들 데이터 구조들이 일반적으로 약 700 MiB의 최소 가상 메모리 풋프린트를 가지며, 32비트 플랫폼에서는 그 메모리 사용량이 미미하다.

> **메모리 단편화 문제?**
>
> 사용 가능한 메모리가 충분함에도 불구하고 연속적인 공간의 부족으로 메모리 할당이 실패하는 현상. Go 런타임은 미리 큰 블록의 주소 공간을 예약함으로써 이러한 단편화를 줄이고 메모리 할당의 효율성을 높인다.

따라서 상단의 VSS와 같은 가상 메모리 메트릭은 일반적으로 Go 프로그램의 메모리 사용량을 이해하는 데 그다지 유용하지 않다. 대신 물리적 메모리 사용량을 보다 직접적으로 반영하는 RSS 및 이와 유사한 측정치에 집중하는 게 좋다.

> **VSS(Virtual Set Size)?**
>
> process와 관련된 virtual memory의 크기. VSS는 프로세스가 사용할 수 있는 가상 메모리의 총량을 나타낸다. 가상 메모리는 실제 물리적 메모리(RAM)와 디스크에 있는 스왑 공간을 포함하여 운영 체제가 프로세스에게 제공하는 주소 공간이다. VSS는 프로세스에 할당된 모든 메모리를 포함하며, 실제로 메모리에 적재되지 않은 부분도 포함한다. VSS는 프로세스가 요청한 전체 메모리 공간을 나타내므로, *프로세스의 메모리 요구량*을 파악하는 데 사용된다. 메모리 맵이 1M이면 프로세스가 어떤 resource 사용하지 않아도 VSS는 1MB를 출력하기 때문에 의미 있는 수치라고 볼 수 없다. VSS에는 다음 항목들을 포함된다.
>
> - 프로세스의 코드(실행 파일)
> - 데이터(전역 변수 등)
> - 힙(동적으로 할당된 메모리)

> - 스택(함수 호출과 지역 변수를 위한 메모리)
> - 메모리 매핑된 파일이나 디바이스
> - 공유 라이브러리
>
> **RSS(Resident Set Size)?**
>
> 프로세스와 관련된 물리적 페이지 수. RSS는 프로세스가 실제로 사용하고 있는 물리적 메모리(RAM)의 양을 나타낸다. 이는 운영 체제가 프로세스의 가상 메모리 중 일부를 물리적 메모리에 매핑한 부분으로, 현재 메모리에 적재되어 있고 접근 가능한 부분만을 포함한다. RSS는 실제로 물리적 메모리에 적재되어 있는 부분을 나타내므로, *시스템의 현재 메모리 부하*를 파악하고, *메모리가 부족할 때 어떤 프로세스가 메모리를 많이 사용하고 있는지*를 파악하는 데 사용된다. 멀티 프로세스 사이에서 shared page 수를 확인할 수 없어 그 값이 정확하지 않다. RSS는 다음을 포함된다.
>
> - 현재 메모리에 적재된 코드
> - 데이터
> - 힙
> - 스택
> - 메모리 매핑된 파일이나 디바이스의 일부
>
> **Shared Pages?**
>
> 여러 프로세스에 의해 공유되는 메모리 페이지. 이는 주로 공유 라이브러리(예: DLLs in Windows, `.so` files in Linux) 또는 프로세스 간 통신을 위해 공유된 메모리 영역에서 발생한다. 공유 페이지를 사용하면 여러 프로세스가 동일한 물리적 메모리 페이지를 읽고 쓸 수 있으므로 메모리 사용량을 효율적으로 줄일 수 있다.

Optimization guide

Identifying costs

Eliminating heap allocations

Implementation-specific optimizations

Linux transparent huge pages (THP)

Appendix

Additional notes on GOGC

기타

참고

- How does gc handle slice memory reclaim
- Go Data Structures
- [안드로이드] 프로세스별 메모리 사용량 분석