

Go Garbage Collector 이해하기

JVM, Python VM과의 비교와 함께

1. GC(쓰레기 수집기)란 무엇인가

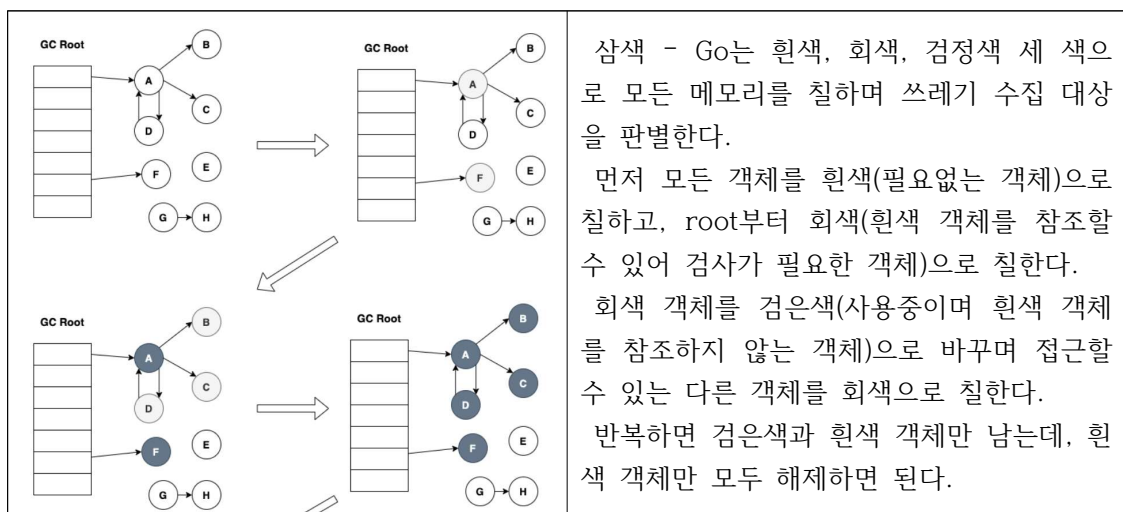
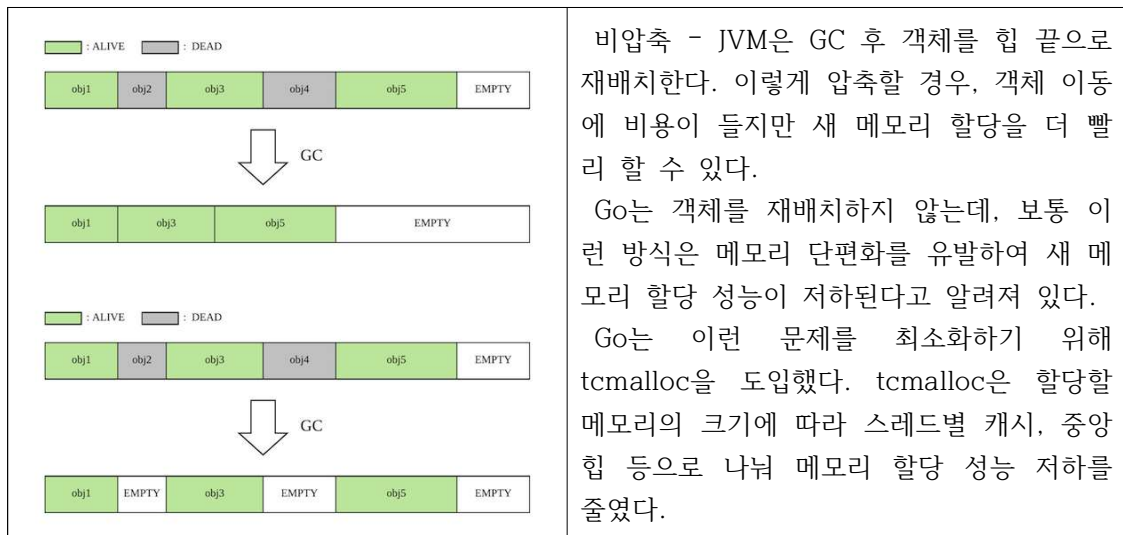
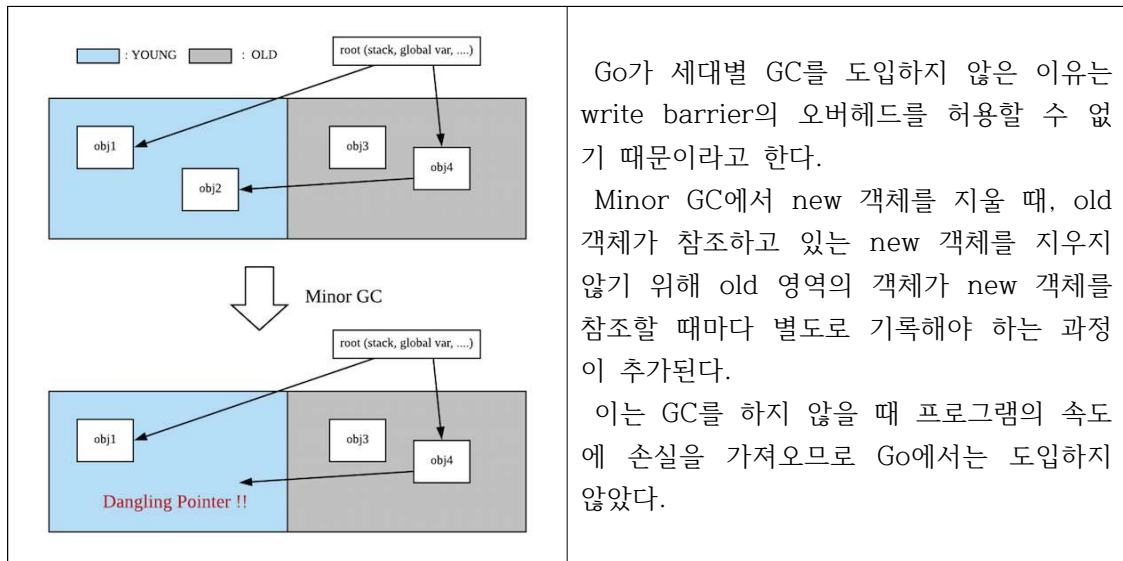
	<p>컴퓨터 프로그램은 동작 시 OS(운영체제)로부터 메모리를 할당받는데, 코드로 조작할 수 있는 메모리 영역은 크게 두 가지로 나뉜다.</p> <p>스택 영역은 컴파일 타임에 크기가 결정되는 (동적으로 크기가 바뀌지 않는) 데이터가 들어가고, 힙 영역은 컴파일 타임에 크기가 결정되지 않는 (동적으로 크기가 바뀌는) 데이터가 들어간다.</p> <p>스택 영역으로의 접근은 컴파일 타임에 미리 전부 계산되기 때문에, 힙 영역보다 더 빠르고 쓰레기 수집도 필요하지 않다.</p> <p>힙 영역은 사용자가 동적으로 메모리를 할당하지만, 해제할 책임도 사용자가 진다.</p>
--	--

	<p>C/C++에서는 malloc, free 등으로 메모리를 수동으로 관리했다. 하지만 실수로 사용한 메모리를 해제하지 않거나, 이미 해제한 메모리를 참조하는 등 메모리 관리에서 실수가 많이 나왔고, 메모리 누수 등의 문제가 생겼다.</p>
--	--

쓰레기 수집기는 사용자가 메모리를 관리할 필요 없이, 자동으로 운영체제로부터 메모리를 할당받고, 필요없는 메모리를 반환한다.

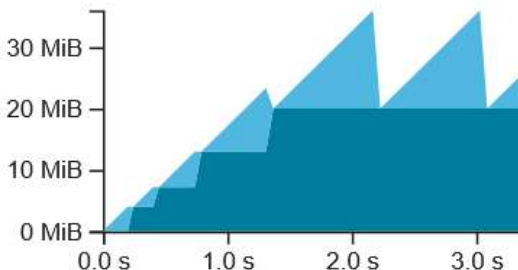
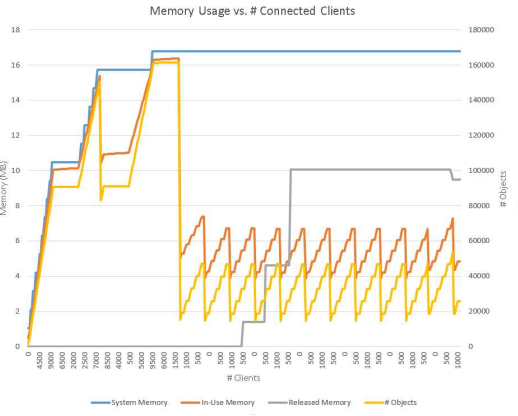
2. Go 쓰레기 수집기 (GOGC)의 특징

	<p>GOGC는 비세대적 비압축 탐색 쓰레기 수집기이다.</p> <p>비세대적 - JVM은 “대부분의 객체는 기대 수명이 짧고, 나머지는 훨씬 긴 기대 수명을 가진다”는 전제하에 세대를 나눠 Major GC를 적게, Minor GC를 많이 하도록 되어 있다. 하지만 GOGC는 세대를 나누지 않고 모두 한번에 GC를 한다.</p>
--	---



Naive Mark & Sweep은 메모리 정합성을 위해 GC를 제외한 다른 모든 스레드는 작업을 진행할 수 없다. 이를 Stop the world (STW)라고 하며, 프로그램이 중간중간 끊기는 원인이 된다. 하지만 Go는 삼색 기법을 사용하기 때문에 STW 기간을 짧고 가볍게 만들 수 있다.

3. Go runtime의 메모리 관리

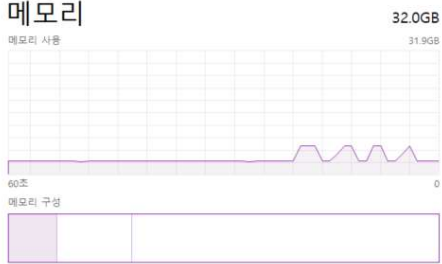
	<p>Go는 기본적으로 지난번 GC에서의 힙의 사용량에 2배만큼 메모리를 사용 중일 때 GC를 진행한다.</p> <p>GOGC 옵션을 통해 GC를 튜닝할 수 있다. GOGC는 지난번 힙 사용량에 몇 %를 추가로 사용중일 때 GC를 수행하는지를 결정하는데, 기본값은 100이다.</p>
	<p>Go가 GC로 사용하지 않는 메모리를 확인했다고 하더라도, 운영체제에 메모리를 반환하지 않을 수 있다.</p> <p>기본적으로 운영체제에서 자원을 할당받고 반환하는 것은 느리므로, Go는 사용하지 않는 메모리라 하더라도 다음 메모리 할당에 사용할 것에 대비해 OS에 메모리를 반환하지 않는다.</p> <p>약 5분간 신규 메모리 할당이 없다면 Go는 할당받았지만 사용 중이 아닌 메모리를 서서히 반환하기 시작한다.</p>
<pre> 1 package main 2 3 import (4 "runtime" 5 "runtime/debug" 6) 7 8 func main() { 9 temp := make([]byte, 100000) 10 temp[0] = 0 11 temp = nil 12 runtime.GC() 13 debug.FreeOSMemory() 14 } 15 </pre>	<p>GC를 위해 2가지 방법을 사용할 수 있는데, 첫 번째는 사용하지 않는 변수를 nil로 설정하는 것이고, 두 번째는 runtime.GC()를 호출하는 것이다.</p> <p>강제로 OS에 메모리를 반환시키려면 debug.FreeOSMemory()를 사용할 수 있다. 이 함수는 GC를 수행하고 사용하지 않는 메모리를 모두 OS에 반환한다.</p> <p>단, GC를 코드에서 직접 호출하는 행위는 성능상 비효율을 초래할 수 있으니 주의해야 한다.</p>

Go의 Mark & Sweep은 세 단계로 구성된다. 먼저 Write barrier를 켜고, 이 단계는 STW가 진행되며, 마킹이 진행되는 동안 고루틴이 힙 영역에 데이터를 안전하게 다룰 수 있게 한다. 다음으로 마킹이 진행되는데, 이 과정은 concurrent하게 진행된다. 가용 CPU 자원의 30%를 차지하기에 성능 손실이 있다. 마지막으로 STW를 일으키며 마킹을 종료하고 다음 GC가 동작할 목표를 정한다.

실제 메모리를 정리하는 sweep 과정은 mark 바로 뒤에 일어나지 않고, lazy 하게 처리된다. 즉, GC와 상관없이 힙 영역에 새롭게 할당이 필요할 때 일어난다. 또한 새롭게 GC가 시작될 때 아직 sweep 되지 않은 영역을 정리한다.

4. 예시 : Python

4GB의 큰 파일을 읽고, 체크섬을 구하는 간단한 작업을 반복하면서 각 언어별 GC의 작동의 차이를 알아보자.



메모리 사용량 그래프: 32.0GB, 31.9GB. 사용 중인 메모리: 3.5GB (261MB). 사용 가능 메모리: 28.3GB. 커밋된 메모리: 8.1/38.5GB. 비커밋된 메모리: 5.5GB. 페이지징 풀: 661MB. 비페이지징 풀: 536MB.

```
# def read(path: str) -> bytes: (...)  
# def checksum(data: bytes) -> list[int]: (...)  
  
for i in range(0, 4):  
    print( checksum( read("./big.bin") ) )  
    time.sleep(2)
```

파이썬은 사용하지 않는 메모리가 있다면 바로바로 OS로 반환하는 것을 볼 수 있다.

5. 예시 : Golang

위와 마찬가지로 상황이다.



메모리 사용량 그래프: 32.0GB, 31.9GB. 사용 중인 메모리: 4.1GB (232MB). 사용 가능 메모리: 27.7GB. 커밋된 메모리: 8.9/38.5GB. 비커밋된 메모리: 5.7GB. 페이지징 풀: 667MB. 비페이지징 풀: 533MB.

```
21 // func read(path string) []byte {...}  
22 // func checksum(data []byte) []byte {...}  
23  
24 func main() {  
25     for i := 0; i < 4; i++ {  
26         fmt.Println(checksum(read("./big.bin")))  
27         time.Sleep(2 * time.Second)  
28     }  
29 }  
30
```

힙 사용량이 2배가 될 때 GC가 수행된다. 사용중이지 않은 메모리는 OS에 반환되지 않으며 다음 메모리 할당에 사용된다.



메모리 사용량 그래프: 32.0GB, 31.9GB. 사용 중인 메모리: 4.2GB (218MB). 사용 가능 메모리: 27.6GB. 커밋된 메모리: 8.8/38.5GB. 비커밋된 메모리: 5.8GB. 페이지징 풀: 665MB. 비페이지징 풀: 532MB.

```
22 // func read(path string) []byte {...}  
23 // func checksum(data []byte) []byte {...}  
24  
25 func main() {  
26     for i := 0; i < 4; i++ {  
27         fmt.Println(checksum(read("./big.bin")))  
28         runtime.GC()  
29         time.Sleep(2 * time.Second)  
30     }  
31 }  
32
```

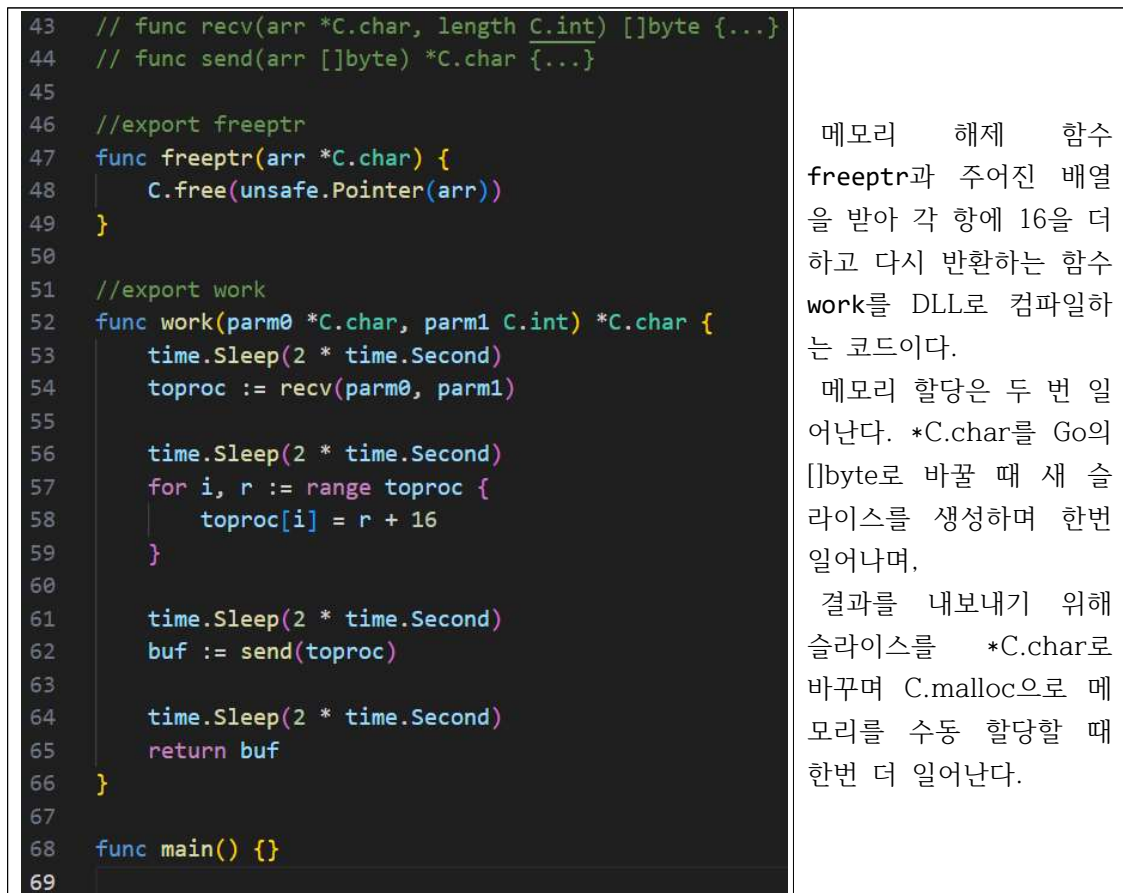
강제로 GC를 수행하면 메모리를 OS에 반환하진 않으나, 힙 사용량이 늘어나기 전에 사용하지 않은 메모리 영역에 새 메모리를 할당할 수 있다.



6. 예시 : CGo 공유 라이브러리 (DLL)

DLL 등 공유 라이브러리로 컴파일한 경우에도, Go는 호출된 작업을 하는 동안 사용한 메모리를 바로 반환하지 않는다. 함수 호출이 끝나도, Go가 내부적으로 사용한 메모리는 남아 있으며, 별도의 설정 없이는 약 5분이 지나야 OS에 반환되기 시작한다. Go가 반환하지 않고 남겨둔 메모리는 나중에 DLL의 다른 함수를 호출했을 때 사용될 수 있다.

이번 예시는 시간이 오래 걸리는 이유로 메모리 사용 그래프 사진이 2개이다.




```

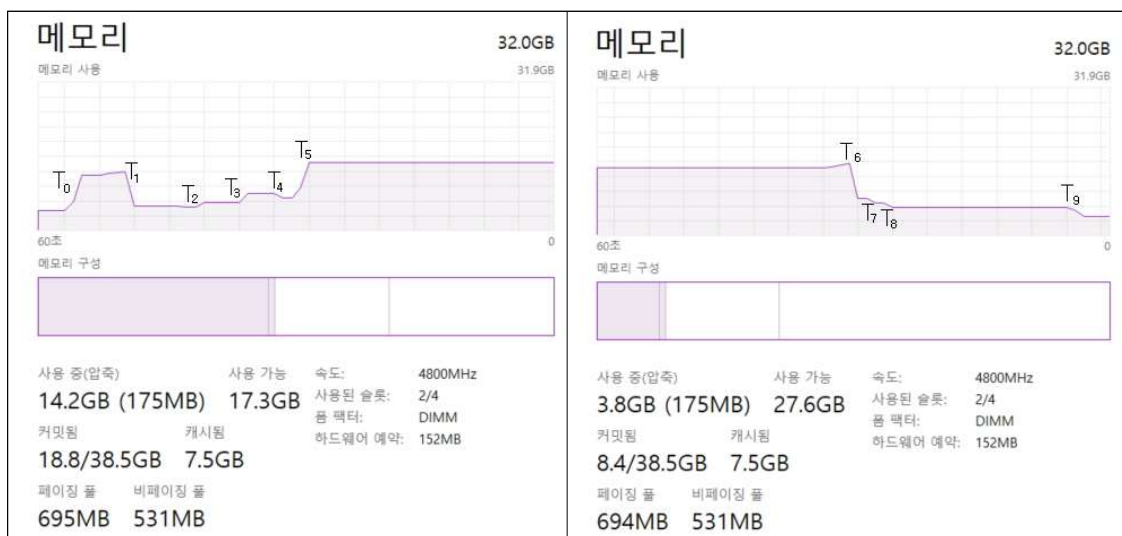
44 # def send(data: bytes) -> c.char.ptr: (...)
45 # def recv(ptr: c.char.ptr, length: int) -> bytes: (...)
46 # dll: ctypes.dll; dll.freeptr: ctypes.Func; dll.work: ctypes.Func
47
48 inarr = [0, 1, 2, 3, 4, 5, 6, 7] * (1024 * 1024 * 120)
49 time.sleep(2)
50
51 inarr = bytes(inarr)
52 time.sleep(2)
53
54 parm0, parm1 = send(inarr), len(inarr)
55 time.sleep(2)
56
57 del inarr
58 time.sleep(2)
59
60 ptr = dll.work(parm0, parm1)
61 time.sleep(2)
62
63 del parm0
64 time.sleep(2)
65
66 resarr = recv(ptr, parm1)
67 print( resarr[0:32] )
68 time.sleep(2)
69
70 del resarr
71 time.sleep(2)
72
73 dll.freeptr(ptr)
74 time.sleep(2)
75

```

파이썬 ctypes에서 바이트 배열은 파이썬 bytes 객체를 이용하기 때문에 추가 메모리 소모가 없다.

하지만 리스트를 바이트 객체로 바꾸거나 ctypes의 C char 포인터에서 리스트로 값을 읽어올 때 메모리 사용량이 많다.

ctypes의 C char 포인터는 파이썬에서 만들어진 경우 파이썬 VM의 GC가 관리한다는 것에 주의하라.



T0 : 파이썬 프로그램 시작.

T0 ~ T1 : 리스트 객체를 생성하며 많은 메모리 사용.

T1 : 리스트 객체에서 바이트 객체가 생성된다. 필요 없어진 리스트 객체의 메모리는 OS에 반환된다.

T1 ~ T2 : parm0라는 ctypes c char 포인터가 생성되고 inarr가 삭제된다. 바이트 객체는 parm0가 참조하고 있기에 해제되지 않는다. inarr, parm0 모두 파이썬 VM의 GC가 관리하고 있다.

T2 : Go DLL 호출 시작.

T2 ~ T3 : Go가 C char 포인터를 받는다. Go []byte로 반환하기 위해 새 메모리를 할당받고 C char 포인터의 내용을 슬라이스에 복사한다. Go는 바이트 슬라이스의 내용을 수정한다.

T3 ~ T4 : 수정된 슬라이스를 반환하기 위해 CGo 함수 malloc으로 힙에 동적할당한다. 바이트 슬라이스의 내용을 힙에 할당된 공간에 복사한다.

T4 : Go의 C char 포인터가 반환된다. 이전에 사용했던 바이트 슬라이스는 이제 쓰레기라서 수집되지만, 다음 DLL 함수 호출을 위해 OS에 반환되지는 않는다.

T4 ~ T5 : parm0 까지 지워지며 파이썬 바이트 객체에 접근할 수 있는 변수가 사라진다. 파이썬 VM은 이를 GC 후 OS에 반환한다.

T5 ~ T6 : 반환된 C char 포인터를 읽는데 리스트 변수가 생성되며 많은 메모리를 사용한다. C char 포인터의 내용이 리스트로 복사된다.

T6 : 리스트 객체에서 바이트 객체가 생성되고, 필요 없어진 리스트 객체의 메모리는 수집되고 OS로 반환된다.

T7 : 바이트 객체의 앞 32 바이트가 출력되고, resarr 변수가 삭제되며 바이트 객체의 참조가 끊긴다. 이 메모리는 GC 되어 OS로 반환된다.

T8 : CGo로 힙에 동적할당된 메모리를 free 하는 DLL 함수가 실행된다.

T8 ~ T9 : 대부분의 메모리가 해제되었지만, Go 내부적으로 사용했던 메모리는 아직 OS 반환되지 않았다.

T9 : 프로그램 창을 닫으며 OS에 의해 강제로 메모리가 회수되었다. 만약 프로그램이 계속 돌아갔다면, Go에 의해 할당되었던 메모리는 약 5분 후 OS에 반환된다.

이 예시에 쓰인 상세 코드를 추가한다.

<pre>14 // (c_char_p, length int) -> []byte 15 func recv(arr *C.char, length C.int) []byte { 16 // C char array -> Go slice 17 gs := (*[1 << 30]C.char)(unsafe.Pointer(arr))[:length:length] 18 19 // convert to []byte 20 bs := make([]byte, len(gs)) 21 for i := 0; i < len(bs); i++ { 22 bs[i] = byte(gs[i]) 23 } 24 25 return bs 26 } 27 28 // []byte -> cptr(nB data) 29 func send(arr []byte) *C.char { 30 length := len(arr) 31 tb := make([]C.char, length) 32 for i, r := range arr { 33 tb[i] = C.char(r) 34 } 35 36 // make C char array 37 na := (*C.char)(C.malloc(C.size_t(length) * C.sizeof_char)) 38 copy((*[1 << 30]C.char)(unsafe.Pointer(na))[:length:length], tb) 39 40 return na 41 }</pre>	<p>*C.char를 슬라이스로 변환하기 위해 먼저 슬라이스를 생성한다. 그 후 포인터의 값을 슬라이스에 복사해온다.</p> <p>C.char의 배열을 생성하고, 반환하려는 슬라이스의 값을 복사한다. []C.char는 C.malloc을 통해 동적할당된 힙 공간의 *C.char로 복사된다.</p>
---	---

<pre>def send(data): arr = ctypes.c_char_p(data) return arr def recv(ptr, length): temp = [0] * length for i in range(0, length): temp[i] = ptr[i][0] return bytes(temp)</pre>	<p>ctypes을 통해 파이썬의 바이트 객체를 C char 포인터로 변환한다. 이 과정은 추가 메모리를 소모하지 않는다.</p> <p>C char 포인터의 값을 하나씩 읽어온다. 리스트를 하나 생성하여, 각 값을 복사한 후 바이트 객체로 변환한다.</p>
---	---