

05. 프로그래밍 기초 with C#

23. 가비지 컬렉션

학습목표

- 가비지 컬렉션의 동작 원리를 이해하고, 성능 최적화에 활용할 수 있다.

들어가며

본디 메모리는 리소스다. 즉, **동적 할당된 메모리를 전부 사용했다면 시스템에 돌려줘야 한다.** 그래서 C++에서는 메모리를 프로그래머가 직접 사용이 끝난 메모리를 해제해야 했다.* 이 과정에서 여러 가지 실수가 많이 발생하는 데 아래와 같다.

* C++11부터 스마트 포인터 라이브러리가 추가되어 모던 C++에서는 명시적 해제보다는 스마트 포인터 사용을 권장하고 있다.

- **메모리 누수**(Memory Leak)
 - 메모리 사용이 끝났음에도 불구하고 **해제하지 않은 것**이다.
- **이중 해제**(Double Free)
 - 이미 해제가 된 메모리임에도 불구하고 **또 해제하는 것**이다.
 - 이미 해제가 된 메모리를 가리키는 포인터/레퍼런스를 **댕글링 포인터/레퍼런스**(Dangling Pointer/Reference)라 한다.
- **섣부른 해제**(Premature Free)
 - **아직 사용이 끝나지 않았음에도 불구하고 해제하는 것**이다.

이런 불편함을 해결하기 위해 나온 기술이 자동 메모리 관리(Automatic Memory Management) 기술인 **가비지 컬렉션**(Garbage Collection)이다. 이번 시간에는 가비지 컬렉션에 대해서 알아보고 .NET 아키텍처에서는 어떻게 동작하고 있는지 알아보도록 하자.

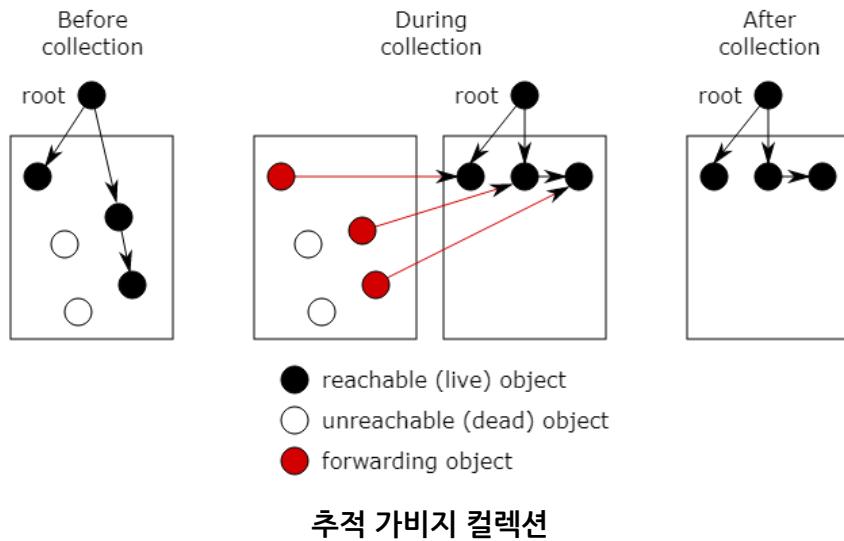
동작 원리

가비지 컬렉션은 **가비지 컬렉터**(Garbage Collector)가 **더이상 사용하지 않는 메모리*를 재사용**하는 것이다. 하지만 애석하게도 어떤 객체가 아직 사용되고 있고, 사용되지 않는지(which object is still lived) 정확하게 판별할 수 있는 알고리즘이 없다.** 그래서 다음과 같은 2가지 방법으로 객체의 사용 유무(Liveness)를 가정한다.

* 이를 가비지(Garbage)라고 한다.

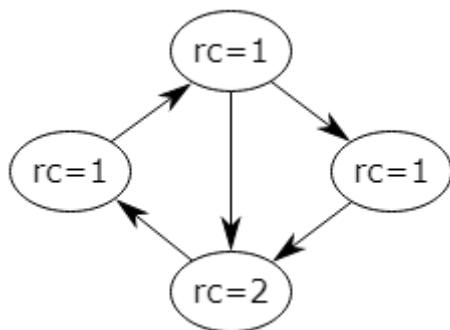
** 정지 문제(Halting Problem)와 관련이 있다.

첫 번째는 **추적 가비지 컬렉션**(Tracing Garbage Collection)이다. 추적 방식에서는 **도달 가능성**(Reachability)으로 생존을 가정하는데 루트(Root)에서 출발해 해당 메모리까지 도달할 수 있는지 보고, 도달되지 못한 메모리는 가비지로 가정한다.



두 번째는 **참조 카운팅**(Reference Counting)*이다. 이 방식에서는 해당 메모리에 참조하는 것이 없을 때 가비지로 가정한다. 참조 카운팅 방법은 **순환 참조**(Circular Reference)**를 주의해야 한다. 순환 참조란 서로 다른 두 메모리가 서로를 참조하는 것을 의미한다. 이를 방지하기 위해 참조 횟수에 영향을 주지 않고 참조를 하는 **약한 참조**(Weak Reference)라는 개념을 사용한다. 이 두 방법은 하이브리드 형식으로 같이 사용될 수 있다.

* 앞서 말한 스마트 포인터 라이브러리가 이 방식으로 동작한다.



참조 카운팅

가비지 컬렉션에도 여러 가지 종류가 있는데, 보수적 가비지 컬렉션(Conservative Garbage Collection), 복제 가비지 컬렉션(Copying Garbage Collection), 분산 가비지 컬렉션(Distributed Garbage Collection), 증분 가비지 컬렉션(Incremental Garbage Collection) 등등이 있다.

.NET 아키텍처의 가비지 컬렉션

C#은 가비지 컬렉션을 지원한다.* 가비지 컬렉션을 지원하는 언어를 **매니지드 언어**(Managed Language)라고 한다.** C#에서 사용하는 방식은 **세대별 가비지 수집**(Generational Garbage Collection)이다. 어떻게 동작하는지 살펴보자.

* 엄밀히는 CLR이 지원한다.

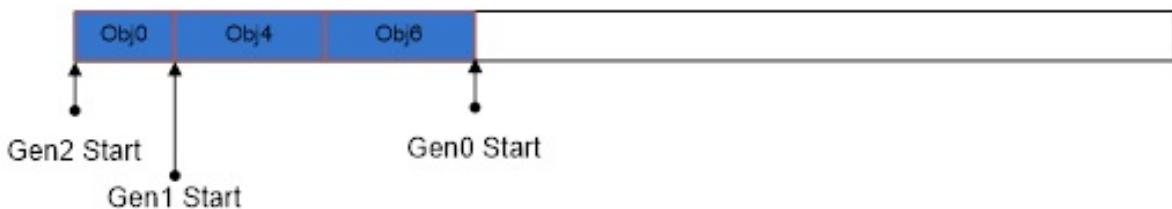
** 반대를 **언매니지드 언어**(Unmanaged Language)라고 한다. 언매니지드 언어는 프로그래머가 거의 모든 작업을 수행하고, 실제 프로그램이 운영체제에 의해 로드되므로 **네이티브 언어**(Native Language)라고도 한다.

세대

먼저 **세대**(Generation)에 대한 이해가 필요하다. 가비지 컬렉터가 관리하는 메모리를 **매니지드 힙**(Managed Heap)이라고 하는데, 이 힙을 0세대, 1세대 및 2세대의 총 3개 세대로 나눠 관리한다. **세대를 나눈 이유는 메모리를 재사용하기 용이하기 때문이다.** 이유는 아래와 같다.

- 가비지 컬렉션이 일어날 때 파편화를 방지하기 위해 메모리를 압축하는 데, **힙 전체를 대상으로 하기보다 일부분에서만 수행 하는 게 더 빠르다.**
- 최근에 만들어진 객체일 수록 수명이 짧고 오래 사용된 객체일 수록 수명이 길어 **재사용할 메모리를 빠르게 분류할 수 있다.**
- 메모리 할당은 0세대에서만 일어나는데 최근에 만들어진 객체끼리 서로 연관되는 경향이 있어 **캐싱 측면에서 좋다.**

매니지드 힙에는 여러 개의 포인터가 있으며 이를 이용해 세대를 구별한다.



매니지드 힙에는 각 세대의 시작을 가리키는 포인터가 있다.

메모리 할당

C#에서 모든 참조 타입의 객체는 매니지드 힙의 **0세대에 할당되며, 연속적으로 배치된다.** 매니지드 힙은 메모리를 미리 시스템으로부터 할당 받아 놓기 때문에* 스택에서 메모리를 할당하는 속도만큼 빠르게 할당할 수 있고, 접근도 빠르게 할 수 있다. 단, 85KB 이상의 크기를 가지는 큰 객체는 **LOH**(Large Object Heap)라는 2세대 메모리에 할당된다.

*메모리 폴링(Memory Pooling)을 떠올리면 된다.

메모리 해제

메모리를 해제할 때는 추적 방식을 사용한다. 루트에는 **스택 루트**, **CPU 레지스터**, **정적 필드** 등이 있으며*, 메모리 해제는 가비지 컬렉터가 가장 적합한 때**에 자동으로 일어나게 된다. 메모리를 해제한다고 했지만 정말로 시스템에 돌려주는 것은 아니다. 실제로는 다른 메모리에 의해 덮어쓰여진다***. 이 과정에서 참조 변수의 주소값을 모두 수정하며, 각 세대의 시작을 가리키는 포인터 또한 수정한다.

* 이외에도 GC 핸들이나 Finalize 큐가 있으나, 일반적으로 고려할 부분은 아니다.

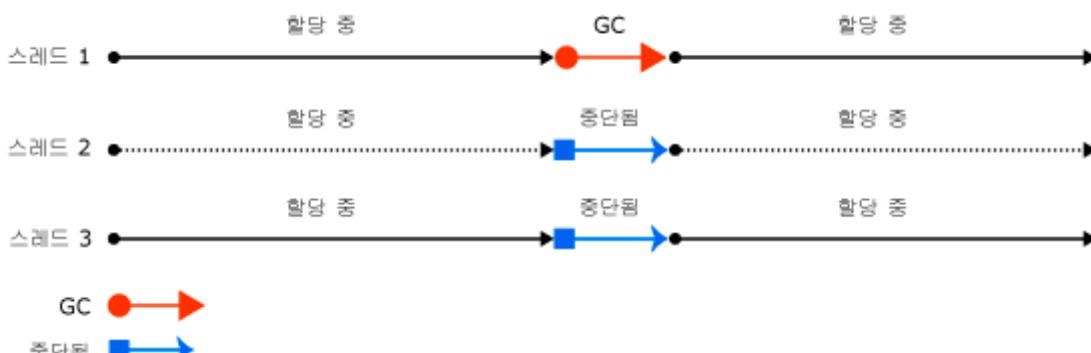
** 상세한 조건을 알고 싶다면 [여기](#)를 확인하라.

*** 단, LOH에서는 덮어쓰는 과정이 생략된다.

가비지 컬렉션이 일어나는 순서도 정해져 있다. 가장 먼저 가비지 컬렉션이 일어나는 세대는 0세대다. 이 과정에서 **가비지가 아닌 메모리는 윗 세대로 승격(Promotion)시킨다**. 만약 0세대에서 가비지 컬렉션을 수행했음에도 불구하고 새로운 객체를 만들기 위한 메모리 공간이 충분하지 않다면, 먼저 1, 2세대 순으로 수집을 수행한다. 그럼에도 또 부족하다면 세대 2, 1, 0의 순서로 수집을 수행한다. 이 경우에도 2세대를 제외하곤 세대 승격은 일어난다.

주의 사항

우리가 가비지 컬렉션에 대해서 명확히 이해해야 하는 이유는 성능과 직결되기 때문이다. **가비지 컬렉션은 결코 자원을 적게 소모하는 연산이 아니며, 멀티스레드 환경인 경우 가비지 컬렉션이 수행되는 동안 다른 스레드가 중단(Suspended)된다**.



멀티스레드 환경에서 가비지 컬렉션이 수행되면 다른 스레드가 중단된다.

따라서 아래의 사항을 주의하자.

참조 카운팅 방식으로 가비지 수집이 일어나지 않는다.

세대별 가비지 컬렉션은 추적 방식이다. 이를 눈으로 확인해보자.

```
using System;

class Program
{
    static readonly int MAX_COUNT = 10000;

    class A { public B b; }
    class B { public A a; }

    static void Main()
    {
        Console.WriteLine($"할당 전 총 메모리 {GC.GetTotalMemory(false)}");

        Foo();

        Console.WriteLine($"할당 후 총 메모리 {GC.GetTotalMemory(false)}");

        GC.Collect();

        Console.WriteLine($"수집 후 총 메모리 {GC.GetTotalMemory(false)}");
    }

    static void Foo()
    
```

```
{
    // 객체를 만들어 서로 순환 참조를 걸어준다.

    for (int i = 0; i < MAX_COUNT; ++i)
    {
        A a = new A();
        B b = new B();

        a.b = b;
        b.a = a;
    }
}
}
```

결과는 아래와 같다.

	할당	전	총	메모리
할당	전	총	메모리	59688
수집	후	총	메모리	554088
			메모리	69192

가비지 컬렉션이 잘 수행되었다.

참조 카운팅 방식이라면 할당 후 총 메모리와 수집 후 총 메모리가 같아야 한다. 하지만 그렇지 않음을 볼 수 있다. 왜냐하면 Foo()에서 생성된 객체에 루트를 통해서 접근할 수 없기 때문이다. 다시 말해 도달할 수 없다.

필요하다면 약한 참조를 사용할 수 있다.

어떤 객체에 도달 가능할 때, 이를 객체에 대한 **강한 참조**(Strong Reference)를 갖는다고 표현한다. 하지만 객체의 도달 가능성에 영향을 주지 않으면서(객체의 생존에 영향을 주지 않으면서) 해당 객체를 참조하고 싶은 때도 있을 것이다. 이때 **약한 참조**(Weak Reference)를 사용할 수 있다. 아래는 예시 코드다.

```
using System;
```

```
class A {}
```

```
class Program
{
    static void Main(string[] args)
    {
        A a;
        TestStrongReference(out a);

        WeakReference aw;
        TestWeakReference(out aw);

        GC.Collect();

        // TestStrongReference()에서 생성된 A 객체는 Main()의 a에 의해 도달될 수 있어 수집되지 않는다.
        Console.WriteLine("TestStrongReference A 강한 참조: {0}", (a == null ? "null" : a.ToString()));

        // 약한 참조를 이용하면 정상적으로 객체가 수집된 걸 확인할 수 있다.
        Console.WriteLine("TestWeakReference A 약한 참조: {0}", (aw.Target == null ? "null" :
aw.Target.ToString()));
    }
}
```

```
static void TestStrongReference(out A ast)
{
    // 객체를 생성한다.
    A a = new A();

    // 강한 참조가 생긴다.
    ast = a;

    // a, b를 null로 만들어준다.
    a = null;

    Console.WriteLine("TestStrongReference A 원본: {0}", (a == null ? "null" : a.ToString()));
}
```

```
static void TestWeakReference(out WeakReference aw)
{
    // 객체를 생성한다.
    A a = new A();

    // 약한 참조가 생긴다.
    aw = new WeakReference(a);

    // a, b를 null로 만들어준다.
    a = null;
```

```
Console.WriteLine("TestWeakReference A 원본: {0}", (a == null ? "null" : a.ToString()));
}
}
```

약한 참조에는 짧은 참조와 긴 참조가 있는데 이에 대해서는 [여기](#)를 참고하자.

빈번한 할당을 조심하자.

가비지 컬렉션이 일어나는 조건 중 하나는 객체를 할당할 충분한 공간이 없을 때다. 즉, 0세대가 가득 찼을 때를 의미한다. 객체를 빈번하게 생성하면 0세대에 여유 공간이 부족해 가비지 컬렉션이 일어날 수 있다.

너무 큰 객체 할당은 피하도록 하자.

앞서 말했듯 85KB 이상의 크기를 가지는 객체는 LOH에 할당되며, LOH에서는 메모리 압축이 일어나지 않아 내부 단편화가 발생할 수 있다.

복잡한 참조 관계를 피하자.

가독성도 문제지만 가비지 컬렉션 후에 메모리 주소 관리를 어렵게 한다. 특히나 오래된 세대에서 새로운 세대에 대한 메모리를 참조하게 될 때 수집을 방지하기 위해 쓰기 장벽(Write Barrier)을 만드는데 이는 많은 성능을 필요로 한다.

관리되지 않는 리소스도 있다.

메모리 외에 파일 핸들, 윈도우 핸들, 네트워크 연결 등의 운영체제 리소스를 래핑하는 경우 제대로 정리가 되지 않는다. 이런 경우에는 [IDisposable](#)와 [using 문](#)을 이용할 수 있다. 자세한 것은 [여기](#)를 참고하자.

더해보기

1. 수업 내용을 복기하면서 아래 내용을 정리해봅시다.
 - a. 메모리를 직접 관리할 때 어떤 문제가 발생할 수 있나요?
 - b. 가비지 컬렉션의 2가지 동작 방법에 대해서 설명해주세요.
 - c. .NET에서는 메모리 할당 과정이 어떻게 되나요?
 - d. .NET에서는 메모리 해제 과정이 어떻게 되나요?

참고자료

- [Automatic Memory Management | Microsoft Learn](#)
- [NET 가비지 수집](#)
- [LOH](#)

- [약한 참조](#)
- [.NET Framework: 561. null 처리된 객체가 왜 GC에 의해 수집되지 않을까요?](#)
- [The Memory Management Reference](#)
- [Understanding Garbage Collection in .NET - Simple Talk](#)