Others

Lập trình bất đồng bộ

Lập trình bất đồng bộ là kỹ thuật lập trình cho phép dừng tiến trình mà không chặn luồng chính.

- **async** là từ khóa dùng để khai báo phương thức bất đồng bộ. Phương thức bất đồng bộ có thể trả về giá trị hoặc không.
- **await** cho phép nhường quyền điều khiển và tiếp tục thực thi khi tác vụ hoàn tất.

Tác dụng:

- Giữ UI/Gameplay mượt khi chờ I/O (HTTP, file, DB).
- Dễ chạy **đồng thời** nhiều I/O với `Task.WhenAll(...)`.
- Đẩy việc **CPU-bound** ra nền bằng `Task.Run(...)`.

Ưu điểm:

- Code tuyến tính, dễ đọc.
- Không block thread → scalable.
- Hỗ trợ hủy (`CancellationToken`) và gom lỗi chuẩn.

Nhược điểm / Lỗi hay gặp:

- `.Result`/`.Wait()` trên main/UI → deadlock.
- `async void` (trừ event) khó bắt lỗi.
- Đồng bộ hoá context phức tạp (cẩn trọng `ConfigureAwait(false)` trong lib).

UniTask

UniTask là thư viện async tối ưu GC cho Unity. Cung cấp awaiter cho `AsyncOperation`, `UnityWebRequest`, Addressables... và tích hợp PlayerLoop.

Tác dụng:

- `await` trực tiếp Unity API (load scene, asset, web request).
- Giảm cấp phát so với `Task`.
- Điều khiển thời điểm tiếp tục (Update/LateUpdate/FixedUpdate).
- Hủy theo vòng đời object: `GetCancellationTokenOnDestroy()`.

Ưu điểm:

- Hiệu năng/GC tốt, sát Unity main thread.
- API phong phú: `UniTask.WhenAll`, `Delay`, `Yield`, `.Forget(handler)`.

Nhược điểm / Lưu ý:

- Thêm phụ thuộc vào bên thứ ba.
- Cần quản lý PlayerLoop & hủy đúng cách để tránh leak.
- Fire-and-forget phải **log/bắt lỗi** (`.Forget(ex => LogException)`).

Gợi ý nhanh:

- I/O: tạo nhiều tác vụ → `WhenAll`.
- CPU: `Task.Run`.
- Tránh `.Result`/`.Wait()`; tránh `async void`.
- Unity: ưu tiên **UniTask**, luôn dùng `CancellationToken` gắn vòng đời object.

Garbage Collection (GC)

GC là bộ thu gom rác tự động của .NET quản lý **vòng đời đối tượng** trên **managed heap. GC** tự phát hiện đối tượng **không còn tham chiếu**, thu hồi bộ nhớ, và (thường) **nén** vùng heap để giảm phân mảnh.

Cách ho?t ??ng

- **Phân tầng**:
  - Gen0: ngắn hạn, được dọn dẹp thường xuyên và rất nhanh
  - Gen1, Gen2: trung hạn và dài hạn, dữ liệu không bị dọn dẹp ở Gen0 sẽ được chuyển lên Gen1 và Gen2. Những tầng này ít bị dọn dẹp hơn. Khi dọn dẹp sẽ tốn thời gian hơn.
- **LOH (Large Object Heap)**:
  - Dữ liệu ≥ ~85KB cấp phát ở LOH;
  - Dữ liệu này thường **không nén** mỗi lần GC. Được sắp xếp lại mỗi lần dọn dẹp.
  - Có thể gây phân mảnh
- **Tạm dừng (STW)**: Để đảm bảo dọn dẹp hiệu quả GC sẽ yêu cầu chương trình tạm dừng để dọn dẹp.
- **Thời điểm kích hoạt**:
  - Khi thiếu bộ nhớ để cấp phát, áp lực bộ nhớ cao.
  - Khi gọi `GC.Collect()` (không khuyến khích).

Destructor & `IDisposable`

- **destructor** (`~Type()`): chạy **không xác định thời điểm** trên thread GC; chỉ dùng khi giữ **tài nguyên unmanaged**.

- **IDisposable/using**: giải phóng tài nguyên **đúng lúc** (socket, file, handle…). Với unmanaged, dùng pattern `Dispose` + `SafeHandle`.
- Quy tắc: **Ưu tiên `Dispose`**; chỉ dùng destructor như “lưới an toàn”.

Nh?ng hi?u l?m th??ng g?p

- “Dùng `GC.Collect()` là tối ưu” → **Sai**: thường làm **tệ hơn** vì tăng STW.
- “Dùng List/Dictionary để tránh phân mảnh” → **Không chính xác**: GC mới xử lý phân mảnh; vấn đề lớn ở **LOH** và **pinning**.
- “GC luôn chậm” → **Phụ thuộc mẫu cấp phát**; tối ưu **mẫu cấp phát** thường hiệu quả hơn “tắt/bắt” GC.

Best practices

- **Giảm cấp phát**: tái sử dụng buffer; dùng `ArrayPool<T>`, `StringBuilder`, cache hợp lý.
- **Tránh boxing** (struct → object), tránh LINQ tạo rác trong hot path.
- **Chú ý vòng đời tham chiếu**: đừng giữ tham chiếu lâu “vô tình” (static cache, event không hủy đăng ký) → đối tượng lên **Gen2**.
- **LOH**: hạn chế tạo nhiều mảng lớn; chia nhỏ/thuê từ `ArrayPool<T>`.
- **Pinning** (`fixed`, `GCHandleType.Pinned`): dùng ít – pin lâu gây phân mảnh.
- **API hiện đại**: `Span<T>`, `Memory<T>`, `ReadOnlySpan<T>` cho xử lý dữ liệu không cấp phát.
- **Thiết lập GC**:
  - **Server GC** (dịch vụ/đa nhân nặng) vs **Workstation GC** (ứng dụng desktop).
  - `GCLatencyMode.LowLatency`/`TryStartNoGCRegion` chỉ dùng ngắn hạn, có kiểm soát.

V?i Unity

- **Incremental GC** (Unity 2019.3+): giảm thời gian dừng bằng cách chia nhỏ công việc GC. Bật trong **Project Settings > Player > Garbage Collector**.
- **Tránh cấp phát mỗi frame**:
  - Không tạo string/`new` trong `Update()`; tránh LINQ/closure/boxing trong `Update()`.
  - Dùng **object pooling** cho bullets, VFX, particle, v.v.
  - Ưu tiên **struct**/Burst/DOTS cho luồng hiệu năng cao.
- **Profiler**: theo dõi `GC.Alloc` và spikes; kiểm tra callsite gây rác.
- **Addressables/Assets**: giải phóng đúng cách (`Release`/`UnloadUnusedAssets`) để giảm áp lực heap.

Checklist t?i ?u GC

- Không gọi [illegible] trừ khi thật cần.
- Dọn [illegible] (unsubscribe) để tránh giữ tham chiếu lâu.
- Dùng [illegible] cho tài nguyên ngoài managed.
- Pool hóa bộ nhớ & đối tượng “ngắn hạn nóng”.
- Tránh cấp phát trong [illegible]/hot path; kiểm tra bằng [illegible].
- Cẩn trọng với [illegible] & [illegible]; ưu tiên chia nhỏ hoặc [illegible].
- Với Unity: bật [illegible] khi phù hợp, tránh tạo rác frame-by-frame.

> *Tối ưu GC = tối ưu mẫu cấp phát + quản vòng đời tham chiếu*, không phải “gọi GC nhiều hơn”.

Revision #2
Created 2025-10-14 18:45:11 UTC by ThanhDV
Updated 2025-10-15 17:57:22 UTC by ThanhDV