Garbage Collector

🗑️ أساسيات الـ Garbage Collector (GC)

الـ GC هو مدير الذاكرة التلقائي في بيئة **CLR (Common Language Runtime)**. وظيفته الأساسية هي تحرير الذاكرة التي لم يعد البرنامج بحاجة إليها، مما يمنع مشكلة تسمى **Memory Leaks (تسرّب الذاكرة)**.

1. ما يفعله الـ GC:

- **يتعقب (Tracking):** يراقب كل **Reference Type** تم إنشاؤه في الـ **Heap**.
- **يحرر (Deallocating):** يزيل الكائنات التي لم يعد **أحد يشير إليها (Referencing)** لتحرير مساحتها.

2. الكائنات التي يراقبها الـ GC:

- الـ GC يتعامل فقط مع الكائنات المخزنة في الـ **Heap**، وهي:
  - **الكلاسات (Classes)**
  - **المصفوفات (Arrays)**
  - **الـ Strings**
  - **الـ Boxing** (عندما يتم تحويل Value Type إلى Reference Type)

> ملاحظة مهمة: الـ Value Types (مثل int و struct) المخزنة في الـ Stack لا يراقبها الـ GC. يتم إزالتها تلقائيًا وبسرعة بمجرد خروج النطاق (Scope) الذي أُعلنت فيه.

🌳 تحليل مفهوم الجذور (Roots) في الـ GC

الـ Roots (الجذور) هي نقطة البداية التي يعتمد عليها الـ Garbage Collector لتحديد الكائنات "الحية" (Live Objects) التي لا يجب مسحها.
الجذر هو أي **مرجع (Reference)** يشير إلى كائن في الـ **Heap** ولكنه موجود **خارج الـ Heap**.

الكائن "الحي" هو أي كائن يمكن الوصول إليه عن طريق تتبع سلسلة من المراجع بدءاً من أحد هذه الجذور.

إليك أمثلة تفصيلية لأنواع الجذور:

نوع الجذر	الشرح التفصيلي	مثال عملي
المتغيرات على Stack	المتغيرات المحلية (Local Variables) داخل الدوال. هذه المتغيرات هي مراجع تشير إلى كائنات في الـ Heap.	`Person p = new Person();` هو جذر موجود في المتغير `p` الاستاك
المتغيرات Static	الحقول الثابتة (Static Fields) في الكلاسات. هذه الكائنات تعيش طوال عمر التطبيق تقريباً.	`public static DatabaseContext DB;` هو مرجع ثابت يعيش الحقل `DB` في منطقة بيانات التطبيق.
CPU Registers	مسجلات المعالج (CPU Registers) هي مواقع تخزين سريعة جداً داخل المعالج نفسه. يتم استخدامها لحفظ المراجع مؤقتاً أثناء تنفيذ الكود.	`Person p = new Person();` عند استدعاء `Console.WriteLine(p.Name)` إلى `p` قد يتم نقل مرجع ، Register مؤقت لتسريع الوصول. إذا كان الـ Register يحمل مرجعاً، فهو يُعتبر جذراً.
Pinned Pointers	عادةً في) مراجع يتم "تثبيتها" بشكل صريح من GC لمنع الـ (`unsafe` الكود غير الآمن أثناء عملية Heap تحريك الكائن في الـ الضغط (Compaction). يُستخدم هذا عند التعامل مع مكتبات C/C++ الخارجية (Native Code).	`fixed (byte* ptr = &byteArray[0]) { ... }` هنا هو الـ `ptr` يمنع تحرك Pinned Pointer `byteArray` مؤقتاً.
Handles (GCHandles)	تُستخدم لإنشاء مراجع قوية (Strong References) من الكود المدار إلى الكود التي لها Timers والـ Threads الأصلي، أو للـ مراجع للكائنات.	**Thread:** إذا كان خيط تشغيل (Thread) يحمل مرجعاً لكائن، فهو يمنع مسحه.

🧠 آلية عمل الـ GC (المراحل الثلاث)

يعمل الـ GC من خلال عملية دورية تتكون من ثلاث مراحل أساسية:

أ. مرحلة التوقف والتعليم (Marking)

1. **توقيف التطبيق (Suspend Threads):** يوقف الـ GC جميع (Threads) مؤقتاً. هذا التوقف يسمى **Stop-The-World Pause**.
2. **تحديد الجذور (Root Determination):** يحدد الـ GC **الجذور (Roots)**. الجذر هو أي مرجع (Reference) يشير إلى كائن في الـ Heap من خارج الـ Heap. تشمل الجذور الشائعة:
   - المتغيرات المحلية في الـ Stack.

- المعاملات التي تمرر للدوال (Parameters).
- الـ Static Fields.

3. **تعليم الكائنات الحية (Marking Live Objects):** يبدأ الـ GC من هذه الجذور ويتبع كل مرجع (Pointer) يشير إلى كائن آخر. أي كائن يمكن الوصول إليه من الجذر يعتبر **"حي" (Live)** ويتم تعليمه (Marked).

ب. مرحلة المسح (Sweeping)

- يقوم الـ GC بفحص الـ Heap بالكامل.
- أي كائن **لم يتم تعليمه** (أي لا يمكن الوصول إليه من أي جذر)، يعتبر **"ميت" (Dead)**، ويتم تحرير مساحته.

ج. مرحلة الضغط (Compacting)

- **لماذا الضغط؟** بعد تحرير مساحات الكائنات الميتة، يصبح هناك فجوات ومساحات فارغة متفرقة في الـ Heap (تسمى **Fragmentation**).
- يقوم الـ GC بـ **تحريك (Compacting)** الكائنات الحية المتبقية إلى بداية الـ Heap لسد هذه الفجوات.
- **النتيجة:** إنشاء كتلة واحدة كبيرة من الذاكرة الحرة، وهذا يسرّع عملية تخصيص الذاكرة الجديدة في المستقبل.
- **ملاحظة هامة:** أثناء الضغط، يجب على الـ GC تحديث جميع **المراجع (References)** التي كانت تشير إلى المواقع القديمة للكائنات الحية لتشير الآن إلى مواقعها الجديدة.

🚦 متى يعمل الـ GC؟

الـ GC لا يعمل بشكل مستمر، بل يعمل في أوقات محددة يقررها الـ CLR تلقائيًا. أهم محفزات عمل الـ GC هي:

1. **نقص الذاكرة (Memory Allocation Request):** عندما يحاول البرنامج تخصيص كائن جديد في الـ Heap ولا يجد مساحة كافية، يتم تشغيل الـ GC لمحاولة تحرير مساحة.
2. **الوصول إلى عتبة محددة (Threshold Reached):** عندما يصل حجم الـ Heap إلى عتبة معينة يحددها النظام.
3. **الاستدعاء الصريح (Manual Call):** نادراً ما يتم استخدامها، ولكن يمكن للمطور استدعاء الـ GC يدوياً باستخدام `GC.Collect()` . (لا ينصح بها للمطورين، لأن الـ GC التلقائي عادة ما يكون أفضل في تحديد التوقيت).

👶 مفهوم أجيال الـ GC

الـ GC في .NET لا يتعامل مع الذاكرة ككتلة واحدة؛ بل يقسمها إلى ثلاثة أجيال منفصلة.

هذا التقسيم مبني على مبدأ إحصائي أساسي في البرمجة:
> **القاعدة الذهبية:** "معظم الكائنات تموت صغيرة (Most objects die young)."

أي أن غالبية الكائنات التي يتم إنشاؤها (مثل المتغيرات المحلية المؤقتة في حلقة تكرارية) تصبح غير ضرورية ويتم تحريرها بعد فترة قصيرة جدًا من إنشائها.

من خلال تقسيم الذاكرة، يمكن للـ GC أن يركز مجهوده على الأجيال التي يُتوقع أن تحتوي على أكبر عدد من الكائنات الميتة (الأجيال الأصغر)، مما يقلل من وقت توقف التطبيق (Stop-the-World Pause).

1. 👶 الجيل الصفري (Generation 0)

- **المحتوى:** جميع الكائنات **الجديدة** التي تم تخصيصها للتو.
- **الحجم:** هذا هو أصغر الأجيال.
- **استراتيجية الـ GC:** يتم فحص هذا الجيل **بشكل متكرر جدًا**، وهو الأسرع في المسح. السبب هو أن معظم الكائنات التي ستموت بسرعة موجودة هنا.
- **الهدف:** تحرير مساحة كافية لتلبية طلبات الـ Allocation الجديدة. إذا نجح الـ GC في Gen 0، فإنه لا يحتاج للانتقال للأجيال الأخرى.

2. 👧 الجيل الأول (Generation 1)

- **المحتوى:** الكائنات التي **نجت** من عملية مسح واحدة للجيل الصفري (Gen 0).
- **الحجم:** متوسط.
- **استراتيجية الـ GC:** يتم مسحه **أقل تكرارًا** من Gen 0، وفقط عندما تفشل محاولة مسح Gen 0 في تحرير ذاكرة كافية.

3. 👴 الجيل الثاني (Generation 2)

- **المحتوى:** الكائنات التي **عاشت طويلاً** ونجت من عمليات مسح Gen 1. وتشمل الكائنات التي تعيش طوال عمر التطبيق (مثل الـ Singletons، الكائنات الثابتة، والتخزين المؤقت طويل الأجل).
- **الحجم:** أكبر الأجيال.
- **استراتيجية الـ GC:** يتم مسحه **نادراً جداً**، لأنه مكلف ويتطلب مسحاً لكامل الـ Heap، وعادةً ما يحدث فقط عندما يكون هناك نقص كبير في الذاكرة.

📈 كيف يتم ترقية الكائنات (Object Promotion)

عملية ترقية الكائن من جيل إلى آخر تحدث كالتالي:

1. يتم إنشاء كائن جديد في **Gen 0**.
2. يحدث **مسح للـ GC (Collection)**.
3. إذا كان الكائن لا يزال **حيًا** (أي ما زال يُشار إليه من قبل جذر ما)، يتم نقله (ترقيته) إلى **الجيل التالي**.
   - الكائنات التي نجت من **Gen 0** تنتقل إلى **Gen 1**.
   - الكائنات التي نجت من **Gen 1** تنتقل إلى **Gen 2**.

النتيجة العملية: الكائنات التي تعيش طويلاً تستقر بسرعة في **Gen 2** ولا يتم تضييع وقت الـ GC في فحصها بشكل متكرر.

⚡ الاستفادة من الأداء

تقسيم الذاكرة إلى أجيال يخدم هدفين رئيسيين للأداء:

1. **تقليل مدة التوقف (Shorter Pauses):** عندما يجمع الـ GC الذاكرة، فإنه يوقف جميع (Threads) (Stop-The-World). بما أن معظم الـ GC Collections هي من Gen 0 الصغيرة، فإن مدة التوقف تكون قصيرة جداً وغير ملحوظة للمستخدم.
2. **كفاءة الـ Caching:** الكائنات الجديدة (Gen 0) تكون متجاورة (Contiguous) في الذاكرة. عندما يتم فحصها، يمكن للـ CPU أن يجلب كتل كبيرة من الذاكرة إلى الـ Cache الخاص به دفعة واحدة، مما يزيد من سرعة الفحص.

🚶 مسار الكائنات عبر الأجيال (Generations)

1. مفهوم "يموت بسرعة" في الـ Gen 0

- **الكائنات قصيرة العمر:** هي الكائنات التي يتم إنشاؤها لغرض مؤقت داخل نطاق ضيق، مثل:
  - المتغيرات داخل حلقة `foreach` أو `for`.
  - سلاسل نصية مؤقتة (`string`) أثناء عمليات بناء سلاسل نصية أكبر.
  - الكائنات التي يتم إنشاؤها لتمريرها كمعاملات (Parameters) لدالة معينة.

مثال:

- عند الخروج من الدالة، تختفي المراجع لهذه الكائنات من الـ **Stack**.

- وعند تشغيل الـ GC (وهو ما يحدث بسرعة في Gen 0)
- يجد أن هذه الكائنات في الـ Heap لم يعد يُشار إليها من أي جذر (ماتت)، فيقوم بمسحها.
- هذا يحدث في غضون ثوانٍ أو أجزاء من الثانية من إنشائها.

2. مسار كائن الـ Singleton

الـ Singleton هو مثال مثالي لكائن **طويل العمر (Long-Lived Object)**، وعادةً ما يكون جذراً ثابتاً (Static Root).

1. **الإنشاء:** يتم إنشاء كائن الـ Singleton لأول مرة في **Gen 0**.
2. **النجاة الأول (Gen 1):** عند أول تشغيل للـ GC (مسح Gen 0)، يجد الـ GC أن الـ Singleton ما زال يُشار إليه بمرجع ثابت (Static Reference) يعتبر **جذراً**.
   - **النتيجة:** يتم ترقية الكائن من **Gen 0** إلى **Gen 1**.
3. **النجاة الثاني (Gen 2):** عند تشغيل الـ GC مرة أخرى (مسح Gen 1)، يجد الـ GC أن الـ Singleton لا يزال حياً.
   - **النتيجة:** يتم ترقيته من **Gen 1** إلى **Gen 2**.
4. **الاستقرار:** يستقر الـ Singleton في **Gen 2**، ونادراً ما يتم فحصه بعد ذلك (فقط عندما يتطلب الأمر مسح Gen 2 بالكامل).

---

3. مسار المسح العملي للأجيال

السيناريو	الكائن	مكان البدء	المسار	متى يُمسح؟
مسح Gen 0	كائن محلي (مثلاً `List<int>` داخل دالة)	Stack (الجذر)	يتم إنشاء الكائن في **Gen 0**. عند انتهاء الدالة، يختفي الجذر.	**مباشرة بعد اختفاء المرجع** (عادةً في أول مسح لـ Gen 0).
مسح Gen 1	كائن تم استخدامه لفترة قصيرة	Gen 0	ينجو الكائن من مسح Gen 0 وينتقل إلى **Gen 1**.	**عندما يفشل في مسح Gen 0** توفير ذاكرة كافية، يتم مسح Gen 1، وحينها يُمسح الكائن إذا لم يعد له مرجع.
مسح Gen 2	كائن ضخم في Cache	Gen 2	ينجو الكائن من Gen 0 و Gen 1 وينتقل إلى **Gen 2**.	**نادراً جداً** (غالباً عند نقص الذاكرة الشديد)

4. تحديد الاستحقاق للمسح

- **متى نقول يستحق المسح؟** عندما لا يمكن الوصول إليه من أي **جذر (Root)** في التطبيق، حتى عبر سلسلة طويلة من المراجع.
- **متى لا يستحق المسح؟** عندما يكون الكائن "حياً"، أي يمكن تتبع سلسلة المراجع المؤدية إليه بدءاً من أحد الجذور.

⏱️ اختلاف وقت الفحص والمسح بين الأجيال

نعم، **وقت الفحص (Scan Time)** و **وقت الضغط/المسح (Compaction/Sweep Time)** يختلف بشكل كبير بين الأجيال، وهذا هو جوهر استراتيجية الأجيال في تحسين الأداء:

الجيل (Generation)	تكرار الفحص	وقت التوقف (Pause Time)	لماذا يختلف الوقت؟
Gen 0	**متكرر جداً**	**قصير جداً (Millisecond)**	حجمه صغير، ومعظم كائناته تموت، ويستفيد من **CPU Cache** (كما ذكرت).
Gen 1	**متوسط**	**متوسط**	يتم فحصه عندما لا يكفي مسح Gen 0، أكبر من Gen 0.
Gen 2	**نادر جداً**	**طويل (قد يصل لثوانٍ)**	يمسح الـ Heap بالكامل، وحجمه كبير، وعملية الضغط (Compaction) فيه أكثر تكلفة.

الخلاصة: الـ GC يركز جهده على **Gen 0** لأنه يمنح أكبر عائد (تحرير مساحة كبيرة) بأقل تكلفة (أقصر وقت توقف).

🧠 آلية المسح في الـ Gen 0 والـ CPU Cache

1. دور الـ Cache في الفحص (Scanning)

- **الـ Cache ليس مكان العمل:** الـ CPU **لا يقوم** بعملية المسح (Marking) والتحرير (Sweeping) داخل الـ Cache. الذاكرة الحقيقية للكائنات هي الـ **Heap**.
- **تسريع القراءة:** الـ Cache هي مجرد **ذاكرة مؤقتة فائقة السرعة** لتقليل الوقت المستغرق لقراءة البيانات من الـ RAM (حيث يوجد الـ Heap).
- الـ **Locality of Reference:** نظراً لأن كائنات Gen 0 متراصة ومتجاورة في الـ Heap، عندما يقرأ الـ CPU أول كائن، فإنه يسحب معه كتلة كاملة من الذاكرة (Cache Line) إلى الـ Cache.

- عندما ينتقل الـ GC لفحص الكائن التالي (الذي هو مرصوص بجواره)، يجده **بالفعل موجوداً** في الـ Cache السريع، مما يلغي الحاجة للانتظار حتى يتم تحميله من الـ RAM البطيئة. هذا يسرع مرحلة **التعليم (Marking)** بشكل كبير.

2. عملية المسح والتحرير (Sweeping and Compacting)

عملية المسح والضغط تتم مباشرة على الـ **Heap** في الذاكرة الرئيسية (RAM)، وليست في الـ Cache:

1. **التعليم (Marking):** يحدد الـ GC الكائنات الحية والـ Dead.
2.
3. **الضغط (Compacting):**

- إذا كان الكائن **Dead**، يتم تجاهله.
- إذا كان الكائن **Live**، يقوم الـ GC **بتحريكه** إلى مكان جديد في بداية الـ Heap (لسد الفجوات).
- بعد التحريك، يقوم الـ GC بتحديث **جميع المراجع (References)** التي كانت تشير إلى الموقع القديم للكائن لتشير إلى موقعه الجديد.

الخلاصة: الـ Cache هي أداة **لتسريع القراءة والفحص** (Marking)، لكن **العملية الفعلية للمسح والتحريك** (Compaction) التي تحرر الذاكرة وتغير عناوين الكائنات تتم على **الـ Heap** مباشرة.

🔄 1. Compacting GC vs. Non-Compacting GC

الفرق بين آليتي عمل الـ GC يكمن في كيفية تعامله مع **تفتت الذاكرة (Fragmentation)**.

أ. Compacting GC (آلية الـ GC الضاغطة)

هذه هي الآلية القياسية في **الأجيال الصغيرة (Gen 0 و Gen 1)**، وكذلك في **Gen 2** عند استخدام وضع Workstation GC.

الميزة	الشرح	التأثير على الأداء
آلية العمل	بعد تحديد الكائنات الحية والمسح، يقوم الـ GC بـ **تحريك** جميع الكائنات الحية إلى بداية الذاكرة المخصصة، ملغياً الفجوات.	**الإيجابيات:** تمنع تفتت الذاكرة، وتضمن وجود مساحة حرة **متجاورة كبيرة** للتخصيص المستقبلي (مما يسرّع التخصيص).
التكلفة	**مكلفة في الوقت:** يجب أن يقوم بتحديث **جميع المراجع** التي كانت تشير إلى المواقع القديمة للكائنات .	

ب. Non-Compacting GC (آلية الـ GC غير الضاغطة)

هذه الآلية تُستخدم بشكل أساسي في **Large Object Heap (LOH)**، وتُستخدم أحياناً في Gen 2 عند وضع Server GC.

الميزة	الشرح	التأثير على الأداء
آلية العمل	يكتفي الـ GC بـ **تعليم** و **مسح** الكائنات الميتة لتحرير مساحتها، لكنه **لا يحرك** الكائنات الحية.	**الإيجابيات:** أسرع بكثير في وقت التوقف (Pause Time) لأن لا حاجة لتحديث أي مراجع أو نقل بيانات كبيرة.
التكلفة	**تسبب التفتت:** تؤدي إلى بقاء فجوات في الذاكرة مما قد يجعل تخصيص كائنات ضخمة جديدة أمراً صعباً في المستقبل.	

🏗️ 2. Large Object Heap (LOH)

الـ **LOH (Large Object Heap)** هو منطقة خاصة في الذاكرة مخصصة للكائنات **الضخمة** فقط.

أ. تعريف الكائن الضخم

يعتبر الكائن "ضخماً" إذا كان حجمه **85,000 بايت (حوالي 83 كيلوبايت) أو أكثر**.

- **أمثلة:** مصفوفات ضخمة (مثل `[]byte` أو `[]double`)، سلاسل نصية طويلة جداً.

ب. مكان التخزين والعمر

- يتم تخصيص الكائنات في الـ LOH مباشرة في منطقة منفصلة.
- من منظور الأجيال، تعتبر كائنات الـ LOH جزءاً من **الجيل الثاني (Gen 2)**. هذا يعني أنها لا تُجمع إلا نادراً، فقط عندما يتم جمع Gen 2.

ج. لماذا الـ LOH لا يتم ضغطه (Non-Compacting)؟

هذا هو السبب الجوهري وراء تصميم الـ LOH:

1. **التكلفة الهائلة للنقل (High Cost of Movement):** تخيل محاولة تحريك مصفوفة بحجم 10 ميجابايت في الذاكرة. يتطلب ذلك قدراً كبيراً من وقت المعالج (CPU Time) وعرض النطاق الترددي للذاكرة.
2. **زيادة وقت التوقف (Increased Pause Time):** إذا تم نقل كائن ضخم أثناء عملية الضغط، فإن وقت "Stop-The-World Pause" سيزداد بشكل كبير (ربما يصل إلى مئات المللي ثانية أو أكثر)، مما يؤدي إلى تجربة مستخدم سيئة وتأخر في استجابة الخوادم.
3. **تقليل عائد الضغط:** الـ LOH يحتوي على عدد قليل نسبياً من الكائنات، وإذا مات أحدها فإنه يترك فجوة كبيرة (حوالي 85 كيلوبايت على الأقل). هذه الفجوة الكبيرة يمكن إعادة استخدامها بسهولة لتخزين كائن ضخم جديد دون الحاجة للضغط.

> الخلاصة: يتسامح الـ LOH مع تفتت الذاكرة (Fragmentation) مقابل الحصول على أسرع وقت توقف ممكن أثناء عملية الـ GC.

⚠️ ملاحظة (LOH Fragmentation)

بما أن الـ LOH لا يضغط الذاكرة، فمع مرور الوقت وتخصيص وتحرير الكائنات الضخمة، قد يتفتت الـ LOH

إذا كان لديك الفجوات التالية في الـ LOH:

- فجوة 1: 250 كيلوبايت
- فجوة 2: 250 كيلوبايت
- فجوة 3: 250 كيلوبايت
- فجوة 4: 250 كيلوبايت

هذه الفجوات مجتمعة تمنحك **1 ميجابايت من الذاكرة الحرة**، ولكن:

- إذا طلبت تخصيص كائن بحجم 500 كيلوبايت، فالـ GC لا يستطيع دمج الفجوات الصغيرة ليعطيك مساحة واحدة بحجم 500 كيلوبايت.
- في هذه الحالة، حتى لو كانت الذاكرة الإجمالية متاحة، ستفشل عملية تخصيص الكائن.

الـ GC سيقوم بـ:

- محاولة **توسيع الـ LOH** (أي تخصيص مساحة أكبر).
- لكن لا يمكنه استخدام الفجوات الصغيرة بشكل فعال.

الحل الحديث (منذ .NET Core): أضافت مايكروسوفت خياراً لتمكين **ضغط الـ LOH عند الطلب** (بالإعدادات)، للاستفادة من الضغط في الحالات التي يكون فيها التفتت مشكلة كبيرة.

يوجد وضعيتان رئيسيتان لعمل الـ Garbage Collector:

1. الـ **Workstation GC :** مصمم لواجهة المستخدم والتطبيقات المكتبية.
2. الـ **Server GC :** مصمم لبيئات الخوادم عالية الأداء ومتعددة المعالجات.

🖥️ التعمق في Workstation GC وآلية التزامن

الفكرة الأساسية في **Workstation GC** هي أن **تجربة المستخدم أهم من الأداء الإجمالي للـ Server**. يجب أن تظل الشاشة تستجيب لأزرار الماوس والكتابة دون تجميد (Freezing).

1. 🔄 العمل المتزامن (Concurrent GC)

الوضع المتزامن هو الآلية الأساسية التي يستخدمها Workstation GC للحفاظ على الاستجابة:

الميزة	الشرح التفصيلي	التشبيه العملي
التزامن (Concurrency)	GC يعني أن خيط الـ يعمل في نفس الوقت مع خيوط التطبيق (UI Thread).	تخيل أنك تقوم ببناء جدار (تطبيقك) بينما عامل النظافة (GC الـ) يعمل يقوم بجمع النفايات المتراكمة حولك.
الحد من التوقف (Minimize Stop)	خاصة) إلى إجراء أغلب عمله GC يهدف الـ في الخلفية (**Marking**/مرحلة **التعليم**) **دون إيقاف** التطبيق.	عامل النظافة يقوم بجرد القمامة وتعليمها دون مطالبتك بالتوقف عن البناء.
الـ "Stop-The-World" القصيرة	يحدث توقفان قصيران جداً: الأول لبدء عملية التعليم، والثاني والأهم في النهاية للضغط والتحرير. مدة التوقف الثانية هي الأهم وتكون قصيرة جداً.	تتوقف للحظة واحدة فقط لـ "تأكيد" مواقع القمامة، ثم تتوقف للحظة ثانية لـ "نقل" القمامة المتبقية إلى مكان آخر.
النتيجة	المستخدم لا يلاحظ تأخراً كبيراً في واجهة (UI)، حتى أثناء عملية جمع المستخدم القمامة.	تشعر ببطء طفيف، لكن الشاشة لا تتجمد.

2. التحدي: الضغط المتزامن لا يمكن أن يكون مثالياً

> المشكلة: إذا كان الـ GC يعمل في الخلفية أثناء تحريك الكائنات، كيف يمكنه تحديث المراجع (Pointers) بينما قد يقوم التطبيق بتغيير هذه المراجع في نفس اللحظة؟

- **الحل:** لهذا السبب، لا يمكن أن تتم عملية **الضغط (Compaction)** التي تحرك الكائنات بالكامل بشكل متزامن.
- **عملية الضغط متزامنة (Synchronous Compaction):** عندما يحين وقت الضغط (التحريك)، يجب على Workstation GC أن يقوم بـ **تجميد جميع الخيوط** (Stop-The-World Pause) لضمان أن عملية التحريك وتحديث العناوين تتم بشكل آمن ودون تعارض مع الكود الذي يتم تشغيله.

- **الفرق مع Server GC:** وضع Server GC لا يهتم بهذه التكاليف الزمنية بنفس القدر، بل يفضل السرعة الإجمالية. Workstation GC يضحي بالسرعة الإجمالية لتقليل مدة التوقف، حتى لو كان التجميع يأخذ وقتاً أطول بشكل عام.

🛠️ مثال عملي لـ Workstation GC

تخيل تطبيقاً لمحرر نصوص (مثل Notepad):

الحالة	السيناريو	سلوك Workstation GC
الاستجابة	يقوم المستخدم بكتابة جملة بسرعة.	يتم إنشاء كائنات `string` مؤقتة في الـ Gen 0. يجمع GC Gen 0 بسرعة وبشكل متزامن (Concurrent) دون توقيف الكتابة.
التجميد	في منتصف الكتابة، يتم تشغيل عملية GC كبيرة لـ Gen 2.	ا يضطر لإيقاف التطبيق للحظة (Stop-The-World) قصيرة جداً لإجراء الضغط. يشعر المستخدم بتأخر **جزء من الثانية**، لكن التطبيق لا يتجمد لعدة ثوانٍ.
الأولوية	يضمن أن GC الـ يعمل (خيط واجهة المستخدم) الـ UI Thread بأقل قدر الإمكان.	إذا كان هناك خيط GC متزامن وخيط UI يعملان، يتم إعطاء الأولوية القصوى لخيط الـ UI لمنع التجمد.

⚙️ التعمق في Server GC وآلية التوازي

تم تصميم **Server GC** خصيصًا للتطبيقات التي تحتوي على معالجات متعددة (Multi-core CPUs) وذاكرة كبيرة، ويكون الهدف هو **إنجاز أكبر قدر من العمل في أقل وقت ممكن (Maximum Throughput)**.

1. 🔀 مبدأ التوازي (Parallelism)

بدلاً من استخدام خيط GC واحد (كما في Workstation GC)، يستخدم Server GC خيوط GC متعددة تعمل في وقت واحد:

- **خيط لكل نواة (Thread Per Core):** يتم إنشاء خيط GC واحد لكل معالج منطقي (CPU Core) على الجهاز. إذا كان لديك خادم بـ 8 أنوية (Cores)، سيكون لديك 8 خيوط GC تعمل في نفس الوقت.
- **توزيع العمل:** تقوم هذه الخيوط الـ 8 بتنفيذ مراحل **التعليم** و **المسح** و **الضغط** بالتوازي.
- **الهدف:** تقليل الوقت **الإجمالي** الذي يستغرقه التطبيق بالكامل لإجراء عملية GC.

2. 🧱 الذاكرة المخصصة (Dedicated Heaps)

هذه هي الميزة الأكثر أهمية في Server GC:

- وجود **Heap لكل Thread GC :** بدلاً من وجود Heap واحدة كبيرة مشتركة بين جميع خيوط التطبيق (Shared Heap)، يقوم Server GC بإنشاء **Heap منفصلة** لكل خيط GC.
    - إذا كان لديك 8 أنوية، سيكون لديك 8 Heaps منفصلة، وكل خيط GC يدير Heap خاصة به (منطقة الذاكرة الخاصة به).
- **تقليل النزاع (Reduced Contention):**
    - عندما يقوم خيط GC بمسح أو ضغط Heap الخاصة به، فإنه **لا يتنافس** مع خيوط GC الأخرى أو خيوط التطبيق على الموارد في الـ Heap الرئيسية.
    - هذا يقلل من ظاهرة **Locking** ويجعل عملية تخصيص الكائنات الجديدة أسرع بكثير.

3. عملية التجميع (Collection Process)

على الرغم من التوازي، لا تزال Server GC تستخدم آلية **Stop-The-World** للعمليات التي تحتاج إلى الدقة (مثل الضغط)، ولكن بشكل متزامن:

1. الـ **Stop-The-World (الإيقاف):** يتم إيقاف جميع خيوط التطبيق.
2. **التجميع المتوازي:** تبدأ خيوط الـ GC المتعددة بالعمل بالتوازي:
    - الـ **Marking:** كل خيط GC يفحص الجذور (Roots) في منطقته الخاصة وفي الـ Heaps الأخرى.
    - الـ **Compacting:** كل خيط GC يقوم بعملية الضغط (Compacting) على الـ Heap المخصصة له.
3. **الاستئناف:** بعد الانتهاء من جميع المهام بالتوازي، يتم استئناف عمل خيوط التطبيق.

الميزة	Server GC (شرح أعمق)	التأثير على الأداء
التوازي	على جميع أنوية المعالج، GC توزيع عبء الـ مما يحول عملية التجميع المكلفة إلى عدة عمليات صغيرة تتم في نفس اللحظة.	**تقليل وقت التجميع الإجمالي (Total Collection Time)**، مما يعني أن التطبيق يقضي وقتاً أقل في انتظار تحرير الذاكرة.
الاستجابة	يكون **أطول** (Pause Time) زمن التوقف لأنه لا يركز Workstation GC (من **قليلاً** لكن السرعة، (بنفس القوة على التزامن الإجمالية تعوض ذلك.	على **الإنتاجية (Throughput)** يفضل الاستجابة الفورية، وهو مناسب للخوادم التي لا تتأثر بمللي ثانية إضافية في زمن التوقف.
الذاكرة	مخصصة Heap يستخدم ذاكرة أكبر لأن كل تحتاج إلى مساحة إضافية لتخزين البيانات الإدارية (Overhead).	مناسب فقط للأنظمة ذات الذاكرة الوفيرة (Memory-Rich Systems)، وهي ميزة متوفرة في معظم الخوادم الحديثة.

💡 الخلاصة: التضحية من أجل الإنتاجية

الـ Server GC يضحي ببعض الذاكرة الإجمالية وبزمن توقف (Pause Time) أقصر جداً (الذي يهم مستخدمي سطح المكتب) في حين أن **Server GC** يهدف إلى تقليل **الفترات الطويلة** التي يتوقف فيها التطبيق، إلا أن هذه الفترات قد تكون **أقصر وأقل وضوحًا** في الخوادم، لكن في **تطبيقات سطح المكتب**، حتى هذه الفترات القصيرة قد تكون **مزعجة**.

من أجل:

1. **استخدام قوة المعالج بالكامل (Full CPU Power).**
2. **إنجاز مهمة GC الضخمة في أسرع وقت ممكن.**

هذا يضمن أن الخادم يستطيع معالجة عدد أكبر من الطلبات في الثانية.

🧠 مقارنة الوضعيتين وتأثيرهما على الأداء

الميزة	Workstation GC	Server GC
البيئة المستهدفة	تطبيقات العميل/المستخدم	الخوادم/الإنتاجية العالية
عدد الـ Threads	خيط واحد	خيط واحد لكل CPU Core
التوازي	لا يوجد (يعمل بالتزامن)	**متوازي** (Concurrent)
عدد الـ Heaps	واحدة للتطبيق بأكمله Heap	منفصلة لكل CPU Core Heap
وقت التوقف (Pause Time)	**قصير جداً** (بسبب وضع التزامن)	**أطول قليلاً** (أثناء الضغط المتوازي)
استهلاك الذاكرة	أقل	أعلى (المتعددة Heaps بسبب الـ)

متى تستخدم أياً منهما؟

- إذا كان تطبيقك يتطلب واجهة مستخدم سريعة الاستجابة ، استخدم **Workstation GC** (الوضع الافتراضي).
- إذا كان تطبيقك يعمل كـ **خادم** ويخدم مئات أو آلاف الطلبات في نفس الوقت، يجب عليك تغيير الإعداد إلى **Server GC** للحصول على أعلى إنتاجية ممكنة.

لاتنسوني من دعائكم 💙

Follow Me : a7medsabrii