(https://adictosaltrabajo.com)

Visualización de rendimiento del Garbage Collector

En este tutorial se presentan una serie de ejemplos que demuestran el comportamiento del recolector de basura o Garbage Collector (GC) de Java en determinados escenarios.

0. Índice de contenidos

1. Introducción

En este tutorial se presentan una serie de ejemplos que demuestran el comportamiento del recolector de basura o Garbage Collector (GC) de Java en determinados escenarios que fuerzan al GC a limpiar el Heap de ejecución.

Además de los trozos de código sobre los que se ha trabajado, se invluyen capturas de pantalla de las salidas gráficas de las herramientas especializadas utilizadas durante el tutorial, como **VisualVM** con el plug-in **VisualGC**.

2. Entorno de desarrollo

- MacBook Pro Intel Core 2 Duo 8GB RAM
- SO: Yosemite
- IDE: Eclipse Mars 4.5.0
- Java JDK 1.7

Al IDE Eclipse se le ha instalado el plugin de **VisualVM** que permite la representación en tiempo real de espacio de memoria de la JVM, uso de procesador, carga del GC, etc. Conjunto a **VisualVM**, se ha añadido el plugin **VisualGC** que añade una serie de visualizaciones del recolector: espacio utilizado en Eden, en Survivor y en PermGen, entre otros.

Para más información sobre **VisualVM**, consúltese la página oficial (https://visualvm.github.io/).

3. Llamada explícita al GC

La forma más simple de forzar al colector a limpiar memoria es desreferenciando (haciendo apuntar a null) un objeto creado. Se ha sobreescrito el método Object#finalize para mostrar por consola que, efectivamente, se limpia cuando no hay referencias al objeto.

Es conveniente indicar que la llamada a Runtime#gc no garantiza al 100% que el GC haga una pasada de limpieza; es sólo una sugerencia del usuario a la JVM.

```
package es.autentia.gc_samples;

public class GCExplicito {

    public static void main(String[] args) {
        A a = new A("blanco");

        a = null;

        Runtime.getRuntime().gc();
    }

}

class A {
    private String color;

    public A(String color) {
        this.color = color;
    }

    @Override
    protected void finalize() throws Throwable {
        System.out.println(color + " eliminado");
    }
}
```

La salida que obtendremos por consola nos indicará que el objeto ha sido reciclado:

```
blanco eliminado
```

4. Referencias inalcanzables

Otro de los casos en los que el GC realiza una limpia del Heap es cuando existe una referencia circular entre objetos y no son alcanzables desde fuera.

En el siguiente caso se declaran tres objetos apuntándose entre ellos (mediante un campo de clase):

```
package es.autentia.gc_samples;

public class GCAmbito {

    GCAmbito t;
    int i;

    public GCAmbito(int i) {
        this.i = i;
    }

    public static void main(String[] args) {

        GCAmbito t1 = new GCAmbito(1);
        GCAmbito t2 = new GCAmbito(2);
        GCAmbito t3 = new GCAmbito(3);
        // Ningun objeto se debe reciclar

        t1.t = t2;
        t2.t = t3;
        t3.t = t1;
        // Ningun objeto se debe reciclar

        t1 = null;
        System.out.println("t1 = null");
        Runtime.getRuntime().gc();
        // No se debe reciclar: t3.t apunta aun a t1

        t2 = null;
        System.out.println("t2 = null");
        Runtime.getRuntime().gc();
        // No se debe reciclar: t3.t.t apunta aun a t2

        t3 = null;
        System.out.println("t3 = null");
        Runtime.getRuntime().gc();
        // Deben reciclarse los tres objetos: cada uno de ellos
        // apunta a otro de manera circular, sin ser referenciados
        // desde fuera

    }

    @Override
```

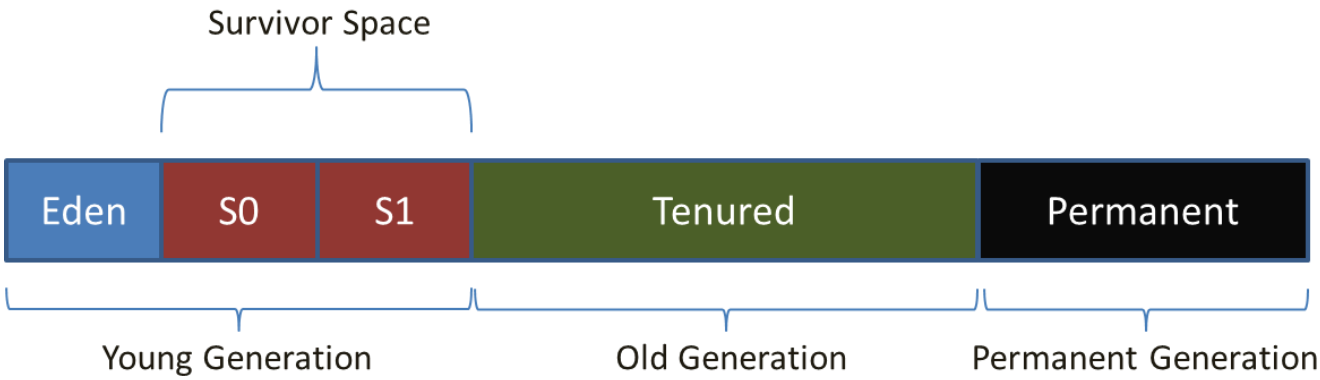
```
    protected void finalize() throws Throwable {
        System.out.println("Finalizado objeto " + i);
    }
}
```

Como se desprende de los comentarios en código, cada uno de los objetos son desreferenciados en orden, pero no son liberados por el GC hasta que la referencia circular es inalcanzable (t3 es desreferenciado).

La salida por consola nos demuestra éste comportamiento (nótese que el orden de reciclaje no es determinista y depende de muchos factores como la carga actual de la JVM):

```
t1 = null
t2 = null
t3 = null
Finalizado objeto 1
Finalizado objeto 3
Finalizado objeto 2
```

5. Estructura del Heap de Java – Hotspot Heap



Hotspot VM Structure

(https://adictosaltrabajo.com/wp-content/uploads/2015/07/2-vmstructure.png)

Conviene recordar cómo se produce el envejecimiento de los objetos en Java mediante la denominada “Weak Generational Hypothesis”. Para más información sobre el funcionamiento y los tipos de GC de Java conviene echarle un vistazo al tutorial sobre Garbage Collector (https://adictosaltrabajo.com/tutoriales/garbage-collector/) de Jose Luis Rodríguez.

En el tipo de GC por defecto de la JVM (Serial GC), la política de limpieza y envejecimiento sigue estos pasos:

- Cualquier objeto nuevo es colocado en Eden.

- Cuando el Eden se llena, se realiza un reciclaje menor (*minor garbage collection*).
- Los objetos con referencias se mueven al primer espacio de supervivientes (S0). El resto serán eliminados en la siguiente limpieza de Eden.
- En el siguiente reciclaje menor, Eden sufre el mismo proceso antes descrito. Los objetos no referenciados son eliminados de Eden; sin embargo, los objetos viejos tanto de Eden como de S0 son movidos al segundo espacio de supervivientes (S1). Una vez terminada la copia, tanto Eden como S0 son limpiados.
- En el siguiente reciclaje menor, los espacios de supervivientes (S0 y S1) se intercambian los roles y vuelven a repetir el proceso.
- Cuando algún objeto de cualquier de los espacios de supervivientes alcanza un determinado tiempo de vida, es promocionado a la *Old Generation*.
- Eventualmente, se producirá un reciclaje amplio (*major garbage collection*) que limpiará y compactará este espacio de la *Old Generation*.

Con este proceso en mente, el siguiente ejemplo ilustra cómo se produce este llenado y consecuente vaciado de ambas generaciones del Heap:

```
package es.autentia.gc_samples;

import java.util.ArrayList;

public class GCDesreferencia {

    public static void main(String[] args) {
        try {
            Thread.sleep(10000);
        } catch (InterruptedException e1) {
            e1.printStackTrace();
        }

        ArrayList arrayStrings = new ArrayList();

        for (int j = 0; j < 100000000; j++) {
            for (int i = 0; i < 1000000; i++) {
                try {
                    Thread.sleep(1);

                    String s = i + j + "Lorem ipsum dolor sit amet,
consectetur adipiscing elit. Fusce"
                            + " viverra semper ipsum, in mollis sem
pellentesque eu. Donec ferment"
                            + "um erat sed mattis facilisis. Praesent
porttitor purus at libero ti"
                            + "ncidunt laoreet. Nam posuere dolor tellus,
vel laoreet sem rutrum eu"
                            + ". Cras posuere dui sit amet lectus
pellentesque accumsan. Nulla auctor cu"
                            + "rsus dolor a efficitur. Cras auctor ultrices
fringilla. Mauris pretium ris"
                            + "us orci, vitae scelerisque lorem blandit at.
Nunc semper pulvinar facilisis."
                            + " Etiam bibendum blandit velit. Mauris
commodo sapien in velit pretium"
                            + ", nec varius nulla porttitor. Aliquam congue
facilisis convallis. Pel"
                            + "lentesque enim orci, sagittis sed cursus
porta, dictum eget massa. ";

                    arrayStrings.add(s);
                } catch (InterruptedException e1) {
```

```
                    e1.printStackTrace();
                }
            }
        }
    }
}
```

A modo de resumen, la aplicación espera unos 10 segundos antes de comenzar la funcionalidad para dar tiempo a lanzar VisualVM empezar a monitorizar el GC. Dentro de los bucles se ha añadido la creación de una serie de Strings lo suficientemente grandes como para que el GC tenga que saltar en un tiempo de simulación razonable. Un pequeño retardo de 1 ms entre cada creación permite un espaciado temporal mayor y aumenta la legibilidad de las gráficas generadas por VisualGc que veremos más adelante.

La creación de un array que vaya almacenando los String generados permite que los objetos sean movidos a la *Old Gen* cuando se cumpla el criterio de envejecimiento.

Entre los muchos flags que podemos añadir a la ejecución del programa para visualizar el trabajo del GC, los básicos son los siguientes:

- **-XX:+PrintGCDetails**: Obtenemos mensajes cada vez que actúa el GC.
- **-Xloggc:<fichero.log>**: Redirige el resultado de los mensajes del GC a un fichero externo en vez de por salida estándar.

Si nos fijamos en el *log* generado, podemos diferenciar dos partes:

```
[ ... ]

0.822: [GC [PSYoungGen: 38912K→5104K(72704K)] 66872K→63644K(159744K),
0.1651680 secs] [Times: user=0.20 sys=0.03, real=0.16 secs]
0.988: [Full GC [PSYoungGen: 5104K→0K(72704K)] [ParOldGen: 58540K-
>63275K(151040K)] 63644K→63275K(223744K) [PSPermGen: 2631K-
>2631K(21504K)], 1.4727490 secs] [Times: user=2.58 sys=0.01, real=1.47
secs]
3.172: [GC [PSYoungGen: 67584K→5120K(72704K)] 130859K-
>119483K(223744K), 0.2449040 secs] [Times: user=0.29 sys=0.05,
real=0.24 secs]
3.418: [Full GC [PSYoungGen: 5120K→0K(72704K)] [ParOldGen: 114363K-
>116723K(265728K)] 119483K→116723K(338432K) [PSPermGen: 2631K-
>2631K(21504K)], 1.3292800 secs] [Times: user=2.42 sys=0.01, real=1.33
secs]
5.192: [GC [PSYoungGen: 67584K→5120K(91136K)] 184307K-
>176627K(356864K), 0.2886140 secs] [Times: user=0.31 sys=0.04,
real=0.28 secs]

[ ... ]
```

La primera parte detalla el estado de ocupación del Heap de forma periódica cada vez que actúa el GC, indicando los tres espacios en KB: la *Young*, la *Old* y la *Permanent Generation*, así como los tiempos en los que se han tomado las medidas. El indicador **GC** se imprime cuando se realiza un *minor collection* y **Full GC** cuando es un *major collection*.

```
[ ... ]

Heap
 PSYoungGen      total 238592K, used 194302K [0x00000007d5500000,
0x00000007f3e00000, 0x0000000800000000)
  eden space 97280K, 54% used
[0x00000007d5500000,0x00000007d88bf808,0x00000007db400000)
  from space 141312K, 100% used
[0x00000007e6400000,0x00000007eee00000,0x00000007eee00000)
  to   space 180224K, 0% used
[0x00000007db400000,0x00000007db400000,0x00000007e6400000)
 ParOldGen       total 450560K, used 352260K [0x0000000780000000,
0x000000079b800000, 0x00000007d5500000)
  object space 450560K, 78% used
[0x0000000780000000,0x0000000795801030,0x000000079b800000)
 PSPermGen       total 21504K, used 2637K [0x000000077ae00000,
0x000000077c300000, 0x0000000780000000)
  object space 21504K, 12% used
[0x000000077ae00000,0x000000077b093790,0x000000077c300000)
```

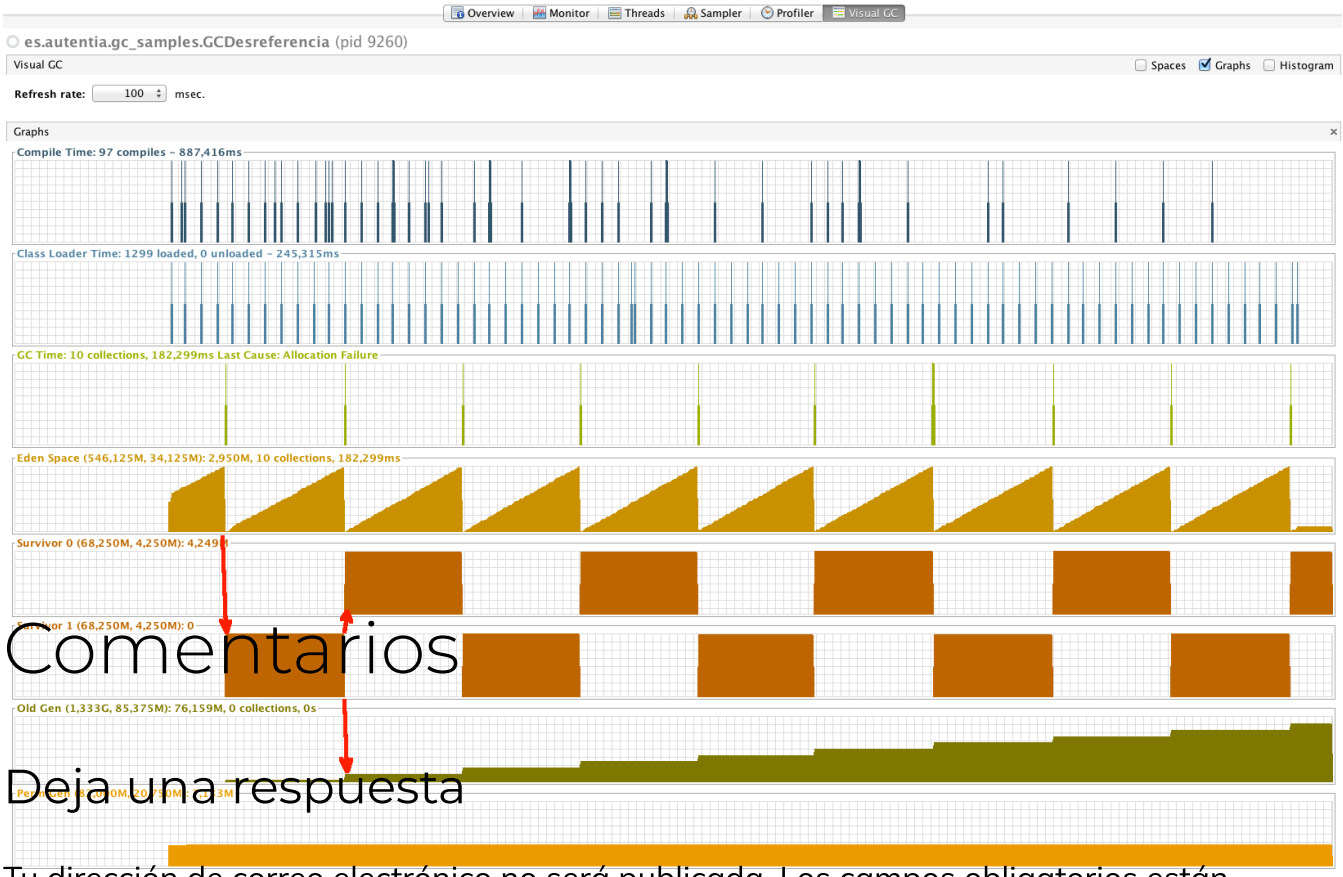
Al final del log se detalla el nivel de ocupación del Heap de manera estratificada al finalizar la ejecución. En la *Young Generation* o **PSYoungGen**, **from space** se refiere a S0 y **to space** a S1.

6. Tipos de GC y su rendimiento comparado

Mediante las herramientas **VisualVM** con el plugin **VisualGC**, podemos realizar una pequeña comparación de rendimiento y comportamiento de los diferentes tipos de GC al ejecutar el ejemplo anterior.

6.1. GC Serie

Flag de ejecución: **-XX:+UseSerialGC**



(https://adictosaltrabajo.com/wp-content/uploads/2015/07/GC_Serie.png)

Este primer ejemplo usa el recolector en serie, que sigue el proceso descrito en el apartado del Hotspot de Java. En la zona resaltada de las gráficas se puede apreciar cómo cada vez que alguna de las zonas de supervivientes son vaciadas, o bien pasa a la siguiente zona, o, si se cumplen los criterios de envejecimiento, se copian a la *Old Gen*.

6.2. GC en Paralelo con 2 hilos

Flags de ejecución: **-XX:+UseParallelGC -XX:ParallelGCThreads=2**

Comentarios

Deja una respuesta

Tu dirección de correo electrónico no será publicada. Los campos obligatorios están marcados con *

Comentario *

☐ He leído y acepto la política de privacidad (https://adictosaltrabajo.com/texto-legal-y-condiciones-de-uso/)

Nombre *

Correo electrónico *

☐ He leído la política de privacidad (http://adictosaltrabajo.com/texto-legal-y-condiciones-de-uso/) y acepto recibir la newsletter con las últimas novedades vía email.

Publicar el comentario

(https://adictosaltrabajo.com/wp-content/uploads/2015/07/GC_Paralelo.png)

Alberto31 agosto, 2016 a las 10:53 pm
(https://adictosaltrabajo.com/2015/07/13/visualizacion-de-rendimiento-del-garbage-collector/#comment-3553)

Hola gracias por este tutorial, me ayudo a aclarar el temaDel GC, tengo una duda, cuando en los logs aparece una entrada parecida a [Full GC [PSYoungGen: ..... y algunas veces sale [Full GC System [PSYoungGen: ..... en que se diferencian? que es lo que libera el que hace referencia a System.
un saludo.

Responder

mukeko1 septiembre, 2016 a las 11:41 am
(https://adictosaltrabajo.com/2015/07/13/visualizacion-de-rendimiento-del-garbage-collector/#comment-3554)

Hola, en primer lugar gracias por este manual para entender el mudo del GC.
Una duda, que diferencia existe en los los, entre [Full GC [PSYoungGen: y [Full GC System [PSYoungGen:..., se libera la misma memoria o que diferencia existe entre los dos FULL GC.
Un saludo.

Responder

6.3. GC con Concurrent Mark Sweep (GCCMS)

En el caso del recolector en paralelo, aunque las pasadas del GC sigan siendo *Stop-the-world*, al aprovechar el uso de varias CPUs, los *minor collections* pueden ser mucho más frecuentes, resultando en tamaños de Eden y Survivor (*Young Generation*) sensiblemente mayores que el GC en serie. Como las *minor collections* son más frecuentes, es necesario un *major collection* antes que con el método anterior.

Flags de ejecución: **-XX:+UseConcMarkSweepGC**

(https://adictosaltrabajo.com/wp-content/uploads/2015/07/GC_CMS.png)

(https://adictosaltrabajo.com/2025/09/09/automatiza-tareas-con-chatgpt-y-zapier/)

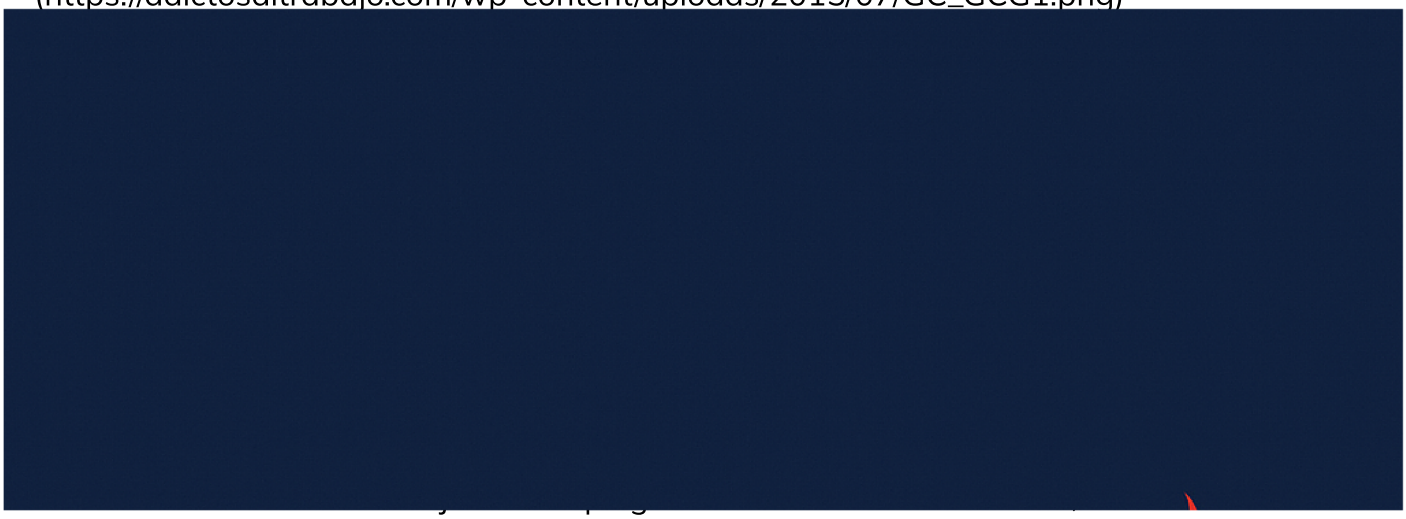
Tutoriales (https://adictosaltrabajo.com/category/tutoriales/)

09/09/2025 Francisco Javier Martínez Páez

Automatiza publicaciones con ChatGPT y Zapier (https://adictosaltrabajo.com/2025/09/09/automatiza-tareas-con-chatgpt-y-zapier/)

Construye un GPT personalizado que publica en LinkedIn vía Zapier con revisión previa, combinando IA generativa y no-code.

(https://adictosaltrabajo.com/wp-content/uploads/2015/07/GC_GCG1.png)



(https://adictosaltrabajo.com/2025/09/01/spring-boot-con-kotlin-frente-a-java-comparativa-practica-y-guia-de-integracion-empresarial/)

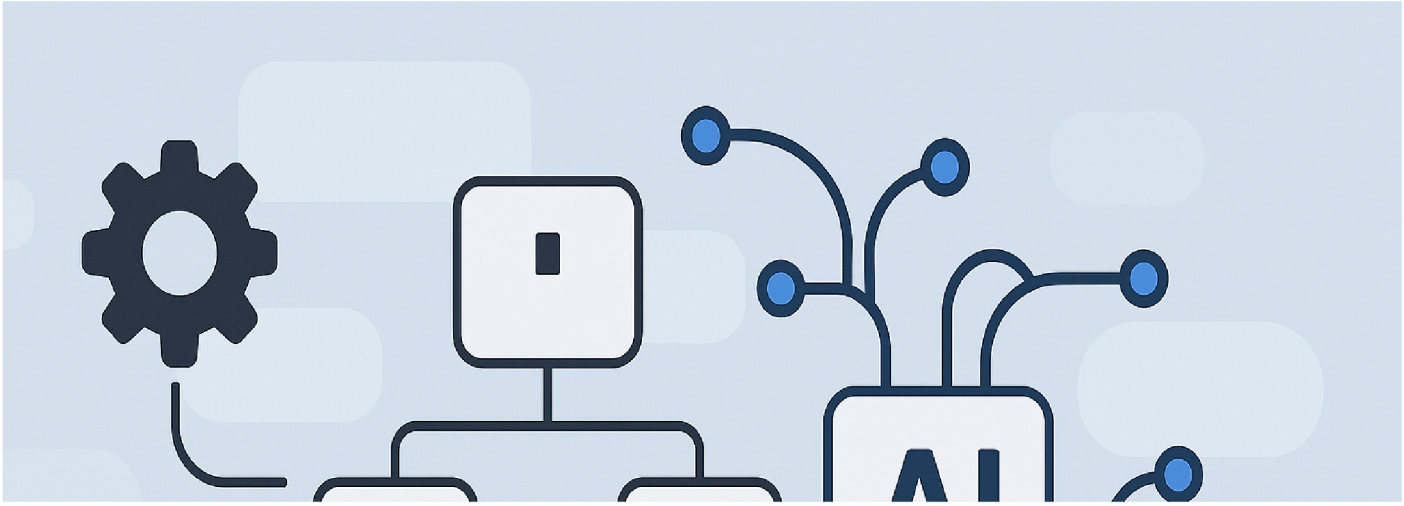
VisualGC, permiten recabar información sobre el rendimiento del GC en nuestras aplicaciones.

Tutoriales (https://adictosaltrabajo.com/category/tutoriales/)

La elección del mejor tipo de GC para un determinado problema es una tarea compleja que debe ser sopesada tanto en términos de uso de memoria, como de frecuencia de dicho uso.

01/09/2025 Iván García Sainz-Aja

Spring Boot con Kotlin frente a Java: Comparativa práctica y guía de integración empresarial (https://adictosaltrabajo.com/2025/09/01/spring-boot-con-kotlin-frente-a-java-comparativa-practica-y-guia-de-integracion-empresarial/)



(https://adictosaltrabajo.com/2025/08/08/automatiza-tareas-con-herramientas-no-code-y-agentes-de-ia-generativa-tutorial-con-zapier-y-google-sheets/)

Tutoriales (https://adictosaltrabajo.com/category/tutoriales/)

08/08/2025 Francisco Javier Martínez Páez

Automatiza tareas con herramientas no-code y agentes de IA generativa: tutorial con Zapier y Google Sheets (https://adictosaltrabajo.com/2025/08/08/automatiza-tareas-con-herramientas-no-code-y-agentes-de-ia-generativa-tutorial-con-zapier-y-google-sheets/)

Descubre cómo automatizar tareas repetitivas combinando herramientas no-code como Zapier, Make, n8n o Relevance AI con el poder de la IA generativa. El artículo incluye un ejemplo práctico donde un agente inteligente publica automáticamente en LinkedIn cada vez que se lanza un nuevo tutorial.

(https://adictosaltrabajo.com)