Optimización

Created	@March 14, 2022 6:07 PM
Tags	2doTrimestre
Formato	
Materia	
Tema	

Técnicas de optimización de código

¿Porqué optimizar código?

- La aplicación irá más rápida.
- Como efecto colateral, consumirá menos batería.
- Gastará menos memoria RAM.

Qué tenemos que optimizar

- **Coste temporal** (Ct): tiempo de ejecución.
- **Coste espacia**l (Ce): espacio de memoria RAM utilizado.

Optimizar preferiblemente

En las rutinas que se ejecutan con mayor frecuencia. Para identificar estas rutinas se pueden utilizar las siguientes técnicas:

- Sesiones de profiling: Procedimientos que se efectúan para conocer cuanto tiempo demora la ejecución de un programa en cada instrucción.
- Sentido común: Depende principalmente de los conocimientos y experiencia del programador.

Técnicas de optimización

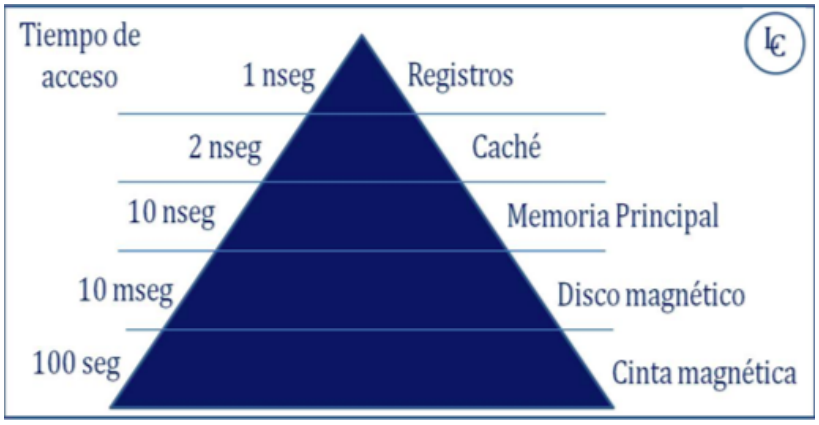
Pasar objetos por referencia mejor que por valor

En Java, los tipos primitivos (int, short, long, boolean, double, float) se pasan por valor y los objetos se pasan por referencia.

Al pasar un objeto por referencia damos la posibilidad a la función de cambiar el valor de la variable pasada, por lo tanto si pasamos los objetos por referencia ahorraremos a la máquina de copiar una y otra vez el valor de un objeto.

```
public void miFuncion(int parametro, Alumno alum) {
}
```

▼ Minimiza y optimiza el acceso a disco.



Reutilización de expresiones comunes

Las expresiones(cálculos) que se repiten pueden agruparse en una variable y hacer ese calculo una sola vez. (Ayudando así al Ct)

```
void ejemplo1SinOptimizar() {
    int b=0,c=0,d=0;
    int a= b +c;
    d=a-d;
    int e=(b+c)*d;
  }
```

```
void ejemplo1Optimizado() {
    int b=0,c=0,d=0;
    int a=b+c;
    d=a-d;
    int e=a*d;
  }
```

Reduce variables innecesarias.

Reducir declaración de variables innecesarias sobre todo dentro de bucles: tiende a favorecer tanto el Ct y Ce.

```
void ejemplo2SinOptimizar() {
        int valor=0,item;
        do {
            item=10;
            valor += valor + item;
        }while(valor<100);
    }
```

```
void ejemplo2Optimizado() {
        int valor=0,item;
        item=10;
        do {
            valor += valor + item;
        }while(valor<100);
    }
```

Elimina código innecesario de los bucles

```
void ejemplo3SinOptimizar() {
    String total="";
```

```
void ejemplo3Optimizado() {
    String total="";
```

```
    for(int i=0;i<10;i++) {
      // otras operaciones....
      String m = "Mensaje hola";
      int contador = i;
      total = m + contador;
    }
    System.out.println(total);
  }
```

```
    String m = "Mensaje hola";
    int i;
    for(i=0;i<10;i++) {
      // otras operaciones....
    }
    total = m + (i-1);
    System.out.println(total);
  }
```

Pre-calcular expresiones constantes

Tiende a favorecer el Ct.

```
void ejemplo4SinOptimizar() {
    int i=2+3;
    int j=4;
    float f = j + 2.5f;
/* aquí hay más código que usa i,j,f*/
  }
```

```
void ejemplo4Optimizado() {
    int i=5;
    int j=4;
    float f = 6.5f;
  }
```

Reducir propagación de copias

Tiende a favorecer el Ce. Ante instrucciones f = a, como se verá en el ejemplo básico, sustituir todos los usos de f por a.

```
void ejemplo5SinOptimizar() {
    int i=10, c=10, m=10;
    int a = 3 + i;
    int f=a;
    int b=f+c;
    int d=a+m;
    m=f+d;
  }
```

```
void ejemplo5Optimizado() {
    int i=10, c=10, m=10;
    int a = 3 + i;
    //int f=a;
    int b=a+c;
    int d=a+m;
    m=a+d;
  }
```

Métodos con pocos parámetros

Crear métodos con el menor número de parámetros posible. El código limpio dice que más de 2 puede que necesites revisar tu código.

Eliminar redundancias en acceso matrices

Eliminar redundancias en acceso matrices: tiende a favorecer el Ct.
Siempre que vayamos a acceder a la misma posición de un array varias veces, es mejor guardar esa posición en una variable local y así evitar el acceso repetido al índice del array.

```
void ejemplo7SinOptimizar() {
    int i=1;
    float array[] = new float[5];
    array[i] = array[8+i] + (i+1)*5*8 + (5+1);
```

```
void ejemplo7Optimizado() {
    int i=1;
    float array[] = new float[5];
    float temporal = array[8+i];
    //tofix (i+1)*5*8 se puede a�adir a temporal
```

```
    array[i-1] = array[8+i] + (i+1) *5*8 + (6+1);
  }
```

```
    //tofix potimizar calculos constantes 5+1 y 6+1
    array[i] = temporal + (i+1)*5*8 + (5+1);
    array[i-1] = temporal + (i+1) *5*8 + (6+1);
  }
```

```
void ejemplo8SinOptimizar() {
    int pos = 3;
    int contador = 0;
    int array[] = new int[5];
    for(int i=0;i<2000;i++) {
      //cosas
      contador = array[pos] + i;
      //cosas
    }
    System.out.println(contador);
    //cosas
  }
```

```
void ejemplo8Optimizado() {
    int pos = 3;
    int contador = 0;
    int array[] = new int[5];
    int temporal = array[pos];
    for(int i=0;i<2000;i++) {
      //cosas
      contador = temporal + i;
      //cosas
    }
    System.out.println(contador);
    //cosas
  }
```

Mejor multiplicar que dividir

El ordenador es capaz de hacer más rápido las multiplicaciones que las divisiones.

```
void ejemplo9SinOptimizar() {
    int a = 10;
    float b = a / 2;
  }
```

```
void ejemplo9Optimizado() {
    int a = 10;
    float b = a * 0.5f;
  }
```

Código muerto

Eliminar el código muerto: tiende a favorecer tanto el Ct y Ce.

- O nunca ejecutados o inalcanzable,
- O si se ejecuta, su producción nunca se utiliza.

```
int ejemplo10SinOptimizar() {
    int x =10, y =20;
    int z = x/y;
    return x/y;
  }
```

```
int ejemplo10Optimizado() {
    int x =10, y =20;
    return x/y;
  }
```

Reutilización de variables

En ocasiones puedes reutilizar variables que has definido antes, en lugar de volver a crear nuevas variables.

```
void ejemplo11SinOptimizar() {
    int i = 0;
    while(i<10) {
```

```
void ejemplo11Optimizado() {
    int i = 0;
    while(i<10) {
```

```
      System.out.println("hola:"+i);
      i++;
    }
    int j = 0;
    while(j<10) {
      System.out.println("adios:"+j);
      j++;
    }
  }
```

```
      System.out.println("hola:"+i);
      i++;
    }
    i=0;
    while(i<10) {
      System.out.println("adios:"+i);
      i++;
    }
  }
```

Nombrar como corresponde

Poner nombres largos a las variables y métodos no hace que el programa sea más lento.

▼ **camelCase**

En esta, la primera letra de cada palabra nueva lleva mayúscula, a excepción de la primera. Ej unaFuncionCualquiera

▼ **Underscore**

En la cual cada palabra es separada por un underscore o guión bajo. Ej una_funcion_cualquiera

▼ **KebabCase**

Separamos cada palabra por un guión medio. Ej una-funcion-cualquiera

▼ **snake_case**

Las palabras se separan por guiones bajo (_) sin espacios, con normalmente todas las palabras en minúscula. Usado en ocasiones en Ruby en la definición de variables y métodos. En las librerias estandar de C y C++

▼ **Hungarian** (**Systems**) notation: iNumberOfPeople

En esta notación precedemos con un caracter para indicar el tipo de variable (Ej i=integer)

▼ **Hungarian** (**Apps**) notation: cntNumberOfPeople

En este caso precedemos la variable de un prefijo indicando su aplicación: (Ej cnt = variable usada como contador)

Demasiado a menudo intenta evitar la creación de objetos de Java

Trata de usar variables locales

Es mejor usar variable locales que variables globales.

```
public class Ejemplo13 {
  int j=5;
  void ejemplo13SinOptimizar() {
    int i = 10;
    int v = i+j;
  }
  void ejemplo13Optimizado() {
      int j=5;
      int i = 10;
      int v = i+j;
```

```
    }
}
```

Evita crear variables innecesarias

```
void ejemplo15SinOptimizar() {
    int i=5;
    ArrayList lista = new ArrayList();
    Elemento e = new Elemento()
    if(i==1) {
      lista.add(v);
    }
  }
```

```
void ejemplo15Optimizado() {
    int i=5;
    ArrayList lista = new ArrayList();
    if(i==1) {
      lista.add( new Elemento() );
    }
  }
```

Evita el uso de expresiones complejas en las condiciones del bucle

```
void ejemplo17SinOptimizar() {
    ArrayList lista = new ArrayList();
    for(int i=0;i< lista.size();i++) {
      // hacemos cosas
    }
  }
```

```
/*TOFIX: usar metodos mejores para recorrer colecciones
  void ejemplo17Optimizado() {
    ArrayList lista = new ArrayList();
    int tamannoLista = lista.size();
    for(int i=0;i< tamannoLista;i++) {
      // hacemos cosas
    }
  }
```

Usar las clases adecuadas en cada momento

Por ejemplo, para guardar algo en una colección, existen multitud de clases que puedes usar, es muy recomendable usar la adecuada, porque unas son más lentas que otras; o tienen unas características que otras no tienen.

```
void ejemplo18SinOptimizar() {
    ArrayList datos = new ArrayList();
    for(int i=0;i<10000;i++) {
      datos.add(i);
    }
  }

  /*El objetivo es almacenar cosas lo mas rapido posible*/
  void ejemplo18OptimizadoA() {
    HashSet datos = new HashSet();
    for(int i=0;i<10000;i++) {
      datos.add(i);
    }
  }
  /*Acceder a la informacion lo mas rapido posible*/
  void ejemplo18OptimizadoB() {
    TreeSet datos = new TreeSet();
    for(int i=0;i<10000;i++) {
      datos.add(i);
    }
  }
```

Usar desplazamiento de bits en lugar de divisiones

Por ejemplo:

x >> 2 es equivalente a x / 4

x << 10 es equivalente a x * 1024

1 << 20 es equivalente a Math.pow(2, 20).

```
void ejemplo11(){
    int a = 44;
    int num1 = a / 2;
    int num1 = a / 4;
    int num2 = a / 8;
    int num3 = a * 4;
    int num4 = a * 8;
  }
```

```
void ejemplo11Optimizado(){
    int a = 44;
    int num1 = a >> 1;
    int num1 = a >> 2;
    int num2 = a >> 3;
    int num3 = a << 2;
    int num4 = a << 3;
  }
```

Salir del bucle cuando sea necesario

```
boolean ejemplo20SinOptimizar() {
    ArrayList palabras = new ArrayList();
    String palabraABuscar = "Ana";

    boolean encontrado = false;
    for(int i=0;i<palabras.size();i++) {
      if(palabras.get(i) == palabraABuscar) {
        encontrado = true;
      }
    }
    return encontrado;
  }
```

```
boolean ejemplo20Optimizado() {
    ArrayList palabras = new ArrayList();
    String palabraABuscar = "Ana";

    //boolean encontrado = false;
    for(int i=0;i<palabras.size();i++) {
      if(palabras.get(i).equals(palabraABuscar) ) {
        return true;
      }
    }
    return false;
  }
```

Evita el uso de variables innecesarias dentro de los bucles

```
void ejemplo21SinOptimizar() {
    int item = 0;
    int valor = 0;
    do {
      item=10;
      valor = valor + item;
    }while(valor<100);
  }
```

```
void ejemplo21Optimizado() {
    int item = 0;
    int valor = 0;
    item=10;
    do {
      valor = valor + item;
    }while(valor<100);
  }
```

Concatenar String

Evitar concatenaciones de Strings, utilizar StringBuffer(sincronizado) o StringBuilder para tal caso. La concatenación de Strings produce cada vez un nuevo objeto y por tanto, mayor consumo de memoria y mayor recolección de basura. (Por favor, esta hacedla siempre)

```
void ejemplo22SinOptimizar() {
    String nombres = "";
    nombres = nombres + "Angel";
    nombres = nombres + "Rosa";
    nombres = nombres + "Paula";
    nombres += "Juan";
  }
```

```
// Mejoramos el con StringBuffer (sincroniced).
  // Lo usaremos solo si estoy creando aplicaciones con
  void ejemplo22OptimizadoA() {
    StringBuffer sb = new StringBuffer();
    sb.append("Angel");
    sb.append("Rosa");
    sb.append("Paula");
    sb.append("Juan");
  }
  // Mejoramos el con StringBuffer (no sincroniced)
  // Lo usaremos si no estoy usando hilos
  void ejemplo22OptimizadoB() {
    StringBuilder sb = new StringBuilder();
    sb.append("Angel");
    sb.append("Rosa");
    sb.append("Paula");
    sb.append("Juan");
  }
```

Referenciar a null

Referenciar a null instancias de objetos que ya no se van a usar, para que el recolector de basura libere memoria.

```
public class ejemplo24 {
  void ejemplo24SinOptimizar() {
    Alumno pepe = new Alumno();
    // trabajo con pepe
    Profesor juan = new Profesor();
    // trabajo con juan
  }
```

```
void ejemplo24Optimizado() {
    Alumno pepe = new Alumno();
    // trabajo con pepe
    pepe=null;
    Profesor juan = new Profesor();
    // trabajo con juan
    juan=null;

  }
```

Atributos VS getters y setters

El acceso a los atributos de una clase es más rápido que encapsular con getter y setter.

Contar hacia atrás

Contar hacia atrás es más rápido en los bucles.

Porque el ciclo ya no tiene que evaluar una propiedad cada vez que se comprueba para ver si está terminado y simplemente se compara con el valor numérico final.

```
//Evalúa .length solo una vez, cuando declara i
for(int i = array.length-1; i!=-1 ;i--)

//evalúa .length cada vez que incrementa i , cuando comprueba si i <= array.length
for(int i = 0; i < array.length; i++)
```

Evitar bucles

Cuando sea posible, evitar bucles, ya que evitaremos toda la sobrecarga de control de flujo en cada iteración.

Por ejemplo, si tenemos una operación que se va a realizar 5 veces, en vez de utilizar un bucle podemos realizar las 5 operaciones secuencialmente.

(nota: esto ensuciará mucho nuestro código)

Usar tipos básico de datos

(int, float, boolean, ...) en lugar de objetos Java (Integer, Float, Boolean, ...) siempre que sea posible.

▼ Tiempo de acceso

La encriptación y conexiones https tienen peor rendimiento.
La apertura de las bases de datos también tienen un rendimiento pobre.
El acceso al disco duro también tiene un alto coste

▼ Evita el uso de métodos sincronizados (si es posible)

Usamos métodos sincronizados cuando creamos aplicaciones multi-hilo y así evitamos que 2 o más hilos accedan a la vez a un método (con todos los problemas que puede acarrear esto).
Pero sincronizar método penaliza haciendo que se ejecute más lentamente el código, así que usa métodos sincronizados solamente cuando sea necesario.

▼ Apertura y cierre de ficheros, conexiones de base de datos y de red.

Puedes optimizar mucho una aplicación abriendo y cerrando en el momento adecuado las conexiones a bases de datos, recursos de red y ficheros.

▼ Pool de objetos

Reutilizar y hacer pool de objetos siempre que sea posible para evitar crear nuevas instancias.
Ver el patrón de diseño objet pool

▼ Liberar recursos tan pronto como sea posible

Liberar recursos tan pronto como sea posible, como conexiones de red, a streams o a ficheros.
Normalmente, se suele liberar este tipo de recursos dentro de la cláusula finally para asegurarnos de que los recursos se liberan aún cuando se produzca alguna excepción.

▼ Variables, arrays o colecciones

Usar variables es más eficiente que arrays.

Los arrays son más eficientes que Vector o HashTable y en cualquier caso, arrays unidimensionales siempre mejor que bidimensionales.

Tener en cuenta también que hay que inicializar la clase Vector y HashTable con un tamaño que se ajuste a nuestras necesidades.

▼ Usar buffers

Utilizar buffers para leer datos a través de la red y leer los datos en porciones en lugar de byte a byte que es más lento.

▼ Evitar métodos sincronizados, usar métodos estáticos

Los métodos sincronizados son los más lentos, a continuación los métodos de interfaz, los métodos de instancia, los métodos finales y por último los métodos estáticos son los más rápidos. Hay que tener en cuenta esta clasificación para evitar siempre que sea posible la sincronización e interfaces.

▼ Evitar métodos sincronizados en los bucles

Evitar en cualquier caso sincronización dentro de bucles.