5 coding hacks to reduce GC overhead

www.takipi.com

1. Avoid Implicit String Allocations

Strings are an integral part of almost every data structure we manage.

Heavier than other primitive values, they have a very **strong impact** on memory usage.

Strings are immutable - they cannot be modified after allocation.

Operators such as “+” allocate a new String to hold the resulting value.

In addition, an **implicit StringBuilder** is allocated to do the actual work of combining them.

For this statement:

```
a = a + b; //a and b are Strings
```

The compiler generates comparable code behind the scenes:

```
StringBuilder temp = new StringBuilder(a).
temp.append(b);
a = temp.toString(); // a new String is allocated. The previous "a" is now garbage.
```

But it can get worse. These simple statements makes 5 implicit allocations (3 Strings and 2 StringBuilders) –

```
String result = foo() + arg;
result += boo();
System.out.println("result = " + result);
```

The Solution

Reducing implicit String allocations in high-scale code by proactively using StringBuilders.

This example achieves the same result , while allocating only **1 StringBuilder + 1 String**, instead of the original 5 allocations.

```
StringBuilder value = new StringBuilder(“result = “);
value.append(foo()).append(arg).append(boo());
System.out.println(value);
```

2. Plan List Capacities

Collections such as ArrayLists, HashMaps and TreeMaps are implemented using underlying Object[] arrays.

Like Strings (themselves wrappers over char[] arrays), arrays are also immutable.

The obvious question then becomes - how can we add/put items in collections if their underlying array's size is immutable?

The answer is obvious as well - by **allocating more implicit arrays**.

Let's look at this example -

```
List<Item> items = new ArrayList<Item>();

for (int i = 0; i < len; i++)
{
     Item item = readNextItem();
     items.add(item);
}
```

The value of `len` determines the ultimate size of `items` once the loop finishes.

`len` is unknown to the constructor of the `ArrayList` which allocates an initial `Object[]` with a default size.

Whenever the capacity of the array is exceeded, it's replaced with a new array of sufficient length, making the previous array garbage.

If the loop executes > 100 times you may be forcing multiple arrays allocations + collections.

The Solution

Whenever possible, allocate lists and maps with an initial capacity:

```
List<MyObject> items = new ArrayList<MyObject>(len);
```

This ensures no unnecessary allocations and collection of arrays occur at runtime.

If you don’t know the exact size, go with an estimate (e.g. 1024, 4096) of what an average size would be, and add some buffer to prevent accidental overflows.

3. Use Efficient Primitive Collections

Java currently support s collections with a primitive key or value through the use of “boxing” - wrapping a primitive value in a standard object which can be allocated and collected by the GC.

For each key / value entry added to a HashMap an internal object is allocated to hold the pair.

A necessary evil, this means an **extra allocation** and possible deallocation every time you put an item in a map.

Use Efficient Primitive Collections (2)

A very common case is to hold a map between a primitive value (such as an Id) and an object.

Since Java's HashMap is built to hold object types (vs. primitives), for every insertion into the map can potentially **allocate yet another object** to hold the primitive value ("boxing" it).

This can potentially more than **triple GC overhead** for the map.

For those coming from a C++ background this can really be troubling news, where STL templates solve this problem very efficiently.

The Solution

Luckily, this problem is being worked on for next versions of Java.

Until then, it's been dealt with quite efficiently by some great libraries which provide primitive trees, maps and lists for each of Java's primitive types.

We strongly recommend Trove, which we've worked with for quite a while and found that can really reduce GC overhead in high scale code.

4. Use Streams Instead of In-memory Buffers

Most of the data we manipulate in server applications comes to us as files or data streamed over the network from another web service or DB.

In most cases, the data is in serialized form, and needs to be deserialized objects before we can begin operating on it.

This stage is very prone to **large implicit allocations.**

Use Streams Instead of In-memory Buffers (2)

A common thing to do is read the data into memory using a ByteArrayInputStream or ByteBuffer and pass that on to the deserialization code.

This can be a **bad move,** as you'd need to allocate and later deallocate room for that data in its entirety while constructing objects out of it .

Since the size of the data can be of unknown size, you'll have to allocate and deallocate internal byte[] arrays to hold the data as it grows beyond the initial buffer's capacity.

The Solution

Most persistence libraries such as Java's native serialization, Google's Protocol Buffers, etc. are built to deserialize data directly from the incoming file or network stream.

This is done without ever having to keep it in memory, and without having to allocate new internal byte arrays to hold the data as it grows.

If available, go for that approach vs. loading binary data into memory.

5. Aggregate Lists

Immutability is a beautiful thing, but in some high scale situations it can have some **serious drawbacks**.

When returning a collection from a function, it's usually advisable to create the collection (e.g. ArrayList) within the method, fill it and return it in the form of a Collection<> interface.

A noticeable cases where this doesn't work well is when collections are aggregated from multiple method calls into a final collection.

This can mean massive allocation of interim collections.

The Solution

Aggregate values into a single collection that's passed into those methods as a parameter.

Example 1 (inefficient) -

```
List<Item> items = new ArrayList<Item>();

for (FileData fileData : fileDatas)
{
     items.addAll(readFileItem(fileData)); //Each invocation creates a new interim list
     //with internal interim arrays
}
```

Example 2 (efficient):

```
List<Item> items = new ArrayList<Item>(fileDatas.size() * averageFileDataSize * 1.5);

for (FileData fileData : fileDatas)
{
     readFileItem(fileData, items); //fill items inside, no interim allocations

}
```

Additional reading -

String interning - http://plumbr.eu/blog/reducing-memory-usage-with-string-intern

Efficient wrappers - http://vanillajava.blogspot.co.il/2013/04/low-gc-coding-efficient-listeners.html

Using Trove - http://java-performance.info/primitive-types-collections-trove-library/

www.takipi.com

God Mode in Production Code

@takipid