Hasso Plattner Institut

IT Systems Engineering | Universität Potsdam

Efficient Java (with Stratosphere)

Arvid Heise,

Large Scale Duplicate Detection

Agenda

- Bottlenecks
- Mutable vs. Immutable
- Caching/Pooling
- Strings
- Primitives
- Final
- Classloaders
- Exception Handling
- Concurrency
- Debugging
- Network

Bottlenecks

- Java is not slow
  - But it is easier to write inefficient code
- Before tweaking
  - Make sure you have a good algorithm
  - Detect bottlenecks (are you CPU, memory, or I/O bound?)
  - Create (micro-)benchmarks to measure the effects
  - Use benchmarks to pinpoint the problem
- After tweaking
  - Very correctness with unit/integration tests

CPU-Bounds

- Inefficient algorithm

- Inefficient loops
- Garbage collector
- Unoptimizable code
- (Un-)Boxing
- Inefficient string handling

Memory bound

- Inefficient algorithm

- Caching unnecessary objects
- Objects too large
- Overallocated strings, collections, maps
- Oversized data types

I/O bound

- Inefficient algorithm

- Too many file/network accesses
- Sequential vs. random access
- Serialization inefficient
- Serialized objects too large
- Inefficient caching

Mutable vs. Immutable

- Immutable objects are needed for good API design
- Easy to use in defensive API design
- Address **immutable**

```
class Person {
    private final String name;

    private final Address address;

    public Person(String name, Address address) {
        this.name = name;
        this.address = address;
    }
}
```

- Person also immutable

Mutable vs. Immutable #2

- Mutable objects are better for fast code
- Harder to use in defensive API design
- Address **mutable**

```
class Person {
    private final String name;

    private final Address address;

    public Person(String name, Address address) {
        this.name = name;
        this.address = new Address(address);
    }
}
```

- What happens if we don't copy the address?

When to use mutable objects?

- Fetching data becomes expensive with immutable objects

```
Map<Person, Integer> personOccurences = new HashMap<>();
public void countOccurences(DataInput logFiles, int logCount)
  throws IOException {
    for (int index = 0; index < logCount; index++) {
        String name = logFiles.readUTF();
        String place = logFiles.readUTF();
        Person person = new Person(name, new Address(place));
        final Integer oldValue = this.personOccurences.get(person);
        this.personOccurences.put(person,
            oldValue == null ? 1 : (oldValue + 1));
    }
}
```

- Need to create a new Person and Address for each log entry

When to use mutable objects?

- Use lookup object

```
Map<Person, Integer> personOccurences = new HashMap<>();
public void countOccurences(DataInput logFiles, int logCount)
  throws IOException {
    Person person = new Person();
    for (int index = 0; index < logCount; index++) {
        String name = logFiles.readUTF();
        String place = logFiles.readUTF();
        person.setName(name);
        person.getAddress().setPlace(place);
       final Integer oldValue = this.personOccurences.get(person);
        this.personOccurences.put(person,
            oldValue == null ? 1 : (oldValue + 1));
    }
}
```

Benefits

- Constant number of objects
  - (Strings for name and address needed allocation in example)
- No memory congestion
- Slow performance gain for omitted object allocation
- Garbage collector will not reduce performance

- We can still get old behavior by object cloning
  - Remember this for PactRecords

Object Pools

- Favorite anti-pattern

```
String name = new String("Peter");
```

- String literals and interned strings are managed by string pool
  - Can be tested for equality with ==
- Similar Integer.valueOf maintains small pool
  - [-128, 127] by default
  - "java.lang.Integer.IntegerCache.high"
- Maintain pool if few different objects
  - That needs to be looked up often
  - XML attributes

External Caching

- EHCache provides map-like interface
  - Removes entries if a certain size is reached
  - Different strategies, LRU most often used
- Data is spilled to disk if configured
- Can use third tier caches as well
- Useful if you want to maintain object pool and you don’t know what is needed most

String Concatenation

- Second favorite anti-pattern

```
String alphabet = "";
for (char letter = 'a'; letter <= 'z'; letter++)
    alphabet += letter;
```

- Use String + only when you know what you are doing
  - Worst language decision in Java
  - **Never** use += inside a loop

- Remember String is immutable, needs lots of copying

- Use StringBuilder instead
  - Compiler does it on its own for

```
String name = firstName + " " + lastName;
```

(Un-)Boxing

- Beware of boxing and unboxing
  - Strongly degrades performance

```
Map<Person, Integer> personOccurences = new HashMap<>();
Person person = new Person(name, new Address(place));
final Integer oldValue = this.personOccurences.get(person);
this.personOccurences.put(person,
    oldValue == null ? 1 : (oldValue + 1));
```

- Use fastutil or trove instead

```
Object2IntMap<Person> personOccurences = new Object2IntOpenHashMap<>(
this.personOccurences.defaultReturnValue(0);
Person person = new Person(name, new Address(place));
final int oldValue = this.personOccurences.getInt(person);
this.personOccurences.put(person, oldValue + 1);
```

Double vs. Float

- Often double is not needed and float is sufficient
- Halves memory consumption
- CPUs usually can perform more floating operation or with less cycles

- Don’t ever use one of these types for currencies

Final

- Use final as often as possible
- Helps to find programming errors
- Helps compiler/JIT to inline

- Imho final parameters and variables should work most of the time
- Final classes are also good if you don’t devise APIs

Classloaders

- Classloaders load classes when needed
- During startup of Java program most time is spent here

- You can use your own Classloaders to load plugins later
- Saves startup time (less classes to manage)
- Cleaner, as you can then actually unload classes
- Look at URLClassLoader for more information

- Also used by Nephele to execute programs
- Be aware that sometimes classes don’t see each other, when in different classloaders

Exception Handling

- Anti-pattern

```
int index = 0;
List<String> strings;
try {
    while(true)
        System.out.println(strings.get(index++));
} catch(IndexOutOfBoundsException e) {
}
```

- To show errors, exceptions are essential and good
- Should not be part of normal workflow
- Primitive return times are better if the result is expected

- Most time is spent in creating stack trace

Debugging Tricks

- Always implement hashCode(), equals(), toString()
  - Eclipse helps to implement them (not easy manually)
- Use Logging, especially in a multithreading environment
- Use constant boolean expressions for debug statements

```
public final static boolean DEBUG = true;
```

  - Changing it to false allows compiler to remove all debug branches

jVisualVM

- Monitors your application
- Shows memory consumption
- Can be used for profiling (install sampler plugins)

- Very useful to create memory dumps and to query them
  - Finds overallocated strings and collections
  - Quickly shows you when your datastructures are larger than expected

- Can also be used for remote sessions

Concurrency

- Try java.util.concurrent package first before custom solution
- Use lock-free structures
  - ConcurrentLinkedQueue, ConcurrentHashMap
  - Note that size() is not constant
- Never use Vector, Hashtable
  - Synchronized versions of ArrayList, HashMap
  - But only for atomar operations
- Never use volatile as substitution for synchronized blocks
  - Does not help with write-write conflicts
  - Useful for stop flags

Network Traffic

- Most Stratosphere programs are network bound
- Combine where possible
- Try to minimize size of data structures
  - Always use more specific type instead of strings if possible
- Use dictionary encodings where possible
  - When processing RDF, replace URLs by IDs
- Use generic compression algorithms