ANALYZING THE MOST COMMON PERFORMANCE AND MEMORY PROBLEMS IN JAVA

18 October 2017

Hotels.com

Who am I?

- Working in Performance and Reliability Engineering Team at Hotels.com
  - Part of Expedia Inc, handling $72billion in bookings last year
- Founder of JavaPerformanceTuning.com
- Author of Java Performance Tuning (O'Reilly)
- Published over 60 articles on Java Performance Tuning & a monthly newsletter for 15 years & around 10 000 tuning tips
- Also researched Black Hole Thermodynamics & published papers on Protein Structure Prediction with the UKs largest Cancer Research organisation

Before you start analyzing

Remember

- There Is ALWAYS A Bottleneck
  - Otherwise processing would take no time at all
- So having a bottleneck is NOT the problem
- Failing to achieve target times is the problem
  - Which means you need targets!
- For failing targets: find the bottleneck that applies, and fix them

Most Common Problems

- Resource leaks
- Slow DB queries
- Inefficient application code
- Too many DB queries
- Concurrency issues
- Memory leaks
- Configuration issues (pooling thresholds, request throttling)
- Slow DB
- GC pauses
- Memory churn

You might want to see another useful talk

I gave another talk at devoxx a few months ago:

"10,000 Java performance tips over 15 years - what did I learn?"

The talk today does NOT depend on you having seen that one,

but does lead on from that one

so I recommend watching it if you want to optimize yourself

https://www.youtube.com/watch?v=OYpTn0nWKR4

Most Common Problems

- 4. Resource leaks
- 2. Slow DB queries
- 3. Inefficient application code
- 2. Too many DB queries
- 6. Concurrency issues
- 4. Memory leaks
- 1. Configuration issues (pooling thresholds, request throttling)
- 2. Slow DB
- 5. GC pauses
- 4. Memory churn

Configuration Issues

Most Common Problems

- Resource leaks
- Slow database queries
- Inefficient application code
- Too many db queries
- Concurrency issues
- Memory leaks
- **1. Configuration issues (pooling thresholds, request throttling)**
- Slow DB
- GC pauses
- Memory churn

Configuration issues

Not a problem specific to Java, all projects have this

There is an awesome tool for this …

Configuration issues

diff

Configuration issues

Seriously, at least half the configuration issues can be fixed by diff'ing the config between the current and the target deployment

You need to do this in a structured and disciplined way

- Have the config in a diff'able format using whatever diff tool you prefer
- Including ALL configs that apply
  - Not just the top-level configs
- Never deploy to production without manually confirming all config changes
  - Using the PRODUCTION config, not dev or UAT or …
  - Diffing the wrong config and assuming that applies to the production diff is a common mistake

Datastore Issues

Most Common Problems

- Resource leaks
- **2. Slow DB queries**
- Inefficient application code
- **2. Too many DB queries**
- Concurrency issues
- Memory leaks
- 1. Configuration issues (pooling thresholds, request throttling)
- **2. Slow DB**
- GC pauses
- Memory churn

Datastores

At the Datastore side, there are always tools that can analyse performance, these are specific to each Datastore and there's little that I can say generically except

- Indexes always matter
- Caching in memory always makes it faster
- Faster disks always makes it faster
- The schema matters enormously

Datastore communications

On the Java side: the datastore request is always much more costly than a local file write - so you can log it; and always has a generic communication layer which either supports monitoring directly or is easily wrapped. You monitor the requests and results and analyse looking for

- Individual queries that take a long time (improve the query: likely an index or schema change)
- Multiple queries that are identical (is it an inefficiency? can they be cached or reduced in frequency?)
- Changes from baseline performance (is datastore load causing a slowdown?)
- Multiple queries that differ only in a parameter (can be combined or use a parametrized query?)

Datastore connection pool

If you have connection pools, you absolutely MUST monitor the waiting time to acquire the connection from the pool

Wrapping Interfaces

```
public interface Say {
    public boolean sayHi();
    public boolean sayBye();
}
```

Wrapping Interfaces

```
public class SayToOut implements Say {
    public boolean sayHi() {print(500, "hello"); return false;}
    public boolean sayBye() {print(800, "goodbye"); return false;}

    private void print(long pause, String s) {
        try {Thread.sleep(pause);} catch (InterruptedException e) {};
        System.out.println(s);
    }
}
```

Wrapping Interfaces

```
public class SayWrapper implements Say {
    private Say say;
    public SayWrapper(Say say) {this.say = say;}

    public boolean sayHi() {return log(System.nanoTime(), say.sayHi());}
    public boolean sayBye() {return log(System.nanoTime(), say.sayBye());}

    private boolean log(long start, boolean ret) {
        System.out.println("sayHi took: "+(System.nanoTime()-start)+ " ns");
        return ret;
    }
```

Wrapping Interfaces

```
say.sayHi()
```

is transparent to the system whether it is a

```
say = new SayToOut()
```

or

```
say = new SayWrapper(new SayToOut())
```

JDBC Wrapper

JDBC API is implemented entirely with interfaces, so perfect candidate for wrapping in a logging layer

Consequently, there are many available, (and very easy to roll your own)

I use p6spy, https://github.com/p6spy/p6spy

P6Spy

Example log entry

- *current time*|execution time|*category*|connection id|*statement SQL String*|effective SQL string (if different)

- *1494281488717*|91|*statement*|connection 1|*SELECT * FROM TIPS WHERE KEYWORD='P6SPY'*

Easily processed using your preferred tool for processing field separated data

P6Spy – slow DB Queries

Example log entry

- *1494281488717*|**911**|*statement*|connection 1|*SELECT * FROM TIPS WHERE KEYWORD='P6SPY'*

- *Look for individual big times*
- *But note that result sets are batched (pagination) so need to combine multiple log entries with the same connection and query with further results*

P6Spy – too many DB queries

Example log entry

- ***1494281488717***|91|*statement*|**connection 1**|*SELECT * FROM TIPS WHERE KEYWORD='P6SPY'*
- ***1494281488817***|91|*statement*|**connection 1**|*SELECT * FROM TIPS WHERE KEYWORD='P6SPY1'*

- *Look for lots of entries with the same connection and with very close timestamps, "chatty requests"*

P6Spy – slow DB

Example log entry

- *1494281488717*|**911**|*statement*|connection 1|*SELECT * FROM TIPS WHERE KEYWORD='P6SPY'*
- *1494281488917*|**1211**|*statement*|connection 1|*SELECT * FROM TIPS WHERE KEYWORD='P6SPY'*

- *Look for lots of individual big times – DB is overloaded*

Inefficient application code

Most Common Problems

- Resource leaks
- 2. Slow DB queries
- **3. Inefficient application code**
- 2. Too many DB queries
- Concurrency issues
- Memory leaks
- 1. Configuration issues (pooling thresholds, request throttling)
- 2. Slow DB
- GC pauses
- Memory churn

Execution profiling

Execution profiling looks at what is taking the most time running in your application

There are two types of execution profiles

- Sampling
  - Takes a sample every N milliseconds
  - Low overhead but misses out what's happening between the samples
- Instrumented
  - Changes the bytecode to wrap every method with the time it takes
  - Significant overhead for small methods – which is of course the recommended way to code – but doesn't miss anything

Execution profiling

In practice, you almost always want to use the sampling profiler unless you know exactly what you are doing

I'll give an example with the VisualVM profiler that is included with the JDK distribution

Execution profiling

DEMO

Resources and Memory

Most Common Problems

- **4. Resource leaks**
- 2. Slow DB queries
- 3. Inefficient application code
- 2. Too many DB queries
- Concurrency issues
- **4. Memory leaks**
- 1. Configuration issues (pooling thresholds, request throttling)
- 2. Slow DB
- GC pauses
- **4. Memory churn**

Always use try with resources

```
try (InputStream in = new InputStream(x);
        OutputStream in = new OutputStream(y))
{
    //do stuff
}
```

try with resources will always remember to close all resources regardless of when exceptions happen

Handles an Exception being thrown in any constructor

- Still correctly closes everything that has been opened prior to the exception
- AND won't throw any NPE for variables not yet initialized
- AND you'll get the constructor Exception, not one thrown during close() calls

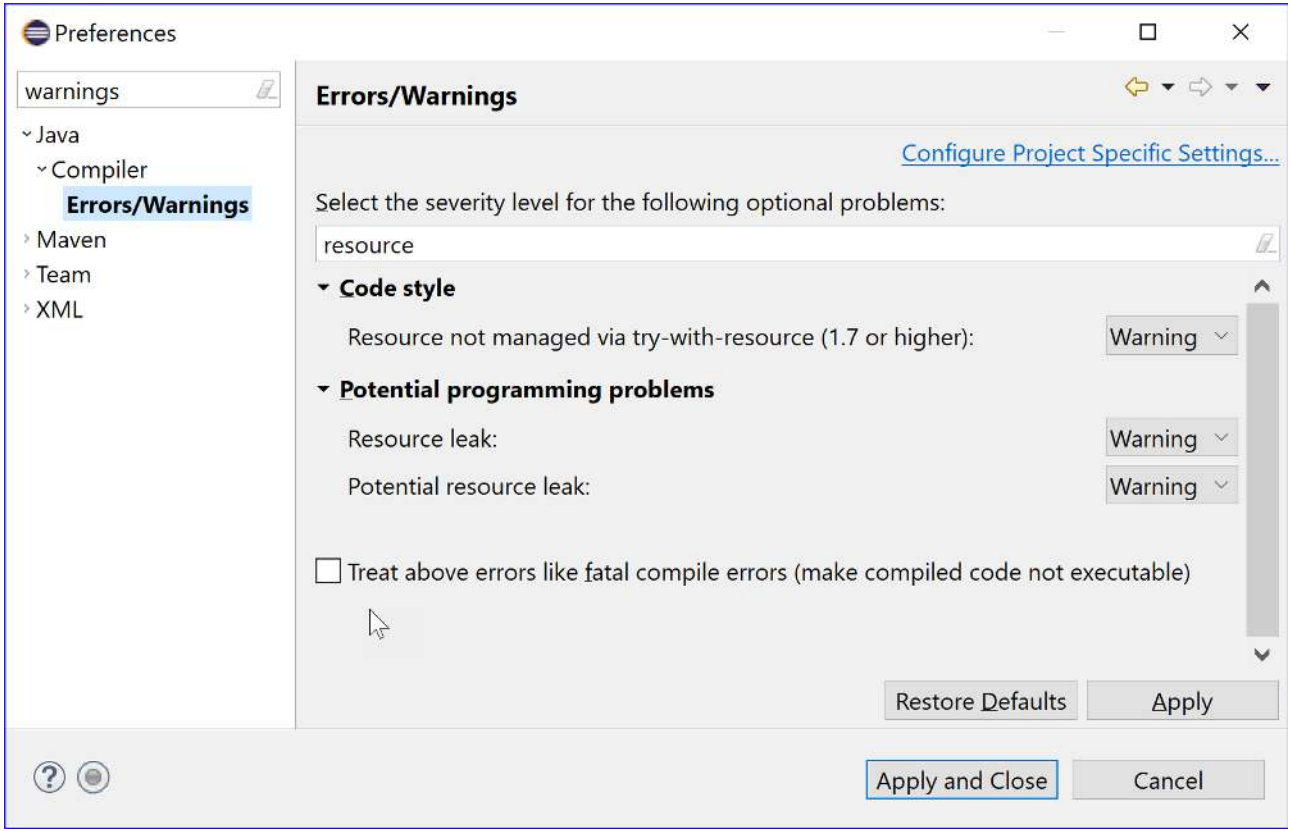
Handles an Exception being thrown while using any of the resources

- Still correctly closes all open resources
- AND you'll get the right Exception, not one thrown during close() calls

Handles Exceptions being thrown while closing

- And will rethrow the first exception so you know there's an issue

Resources without try-with-resources is code smell



Memory profiling & analysis

DEMO

Tools

- P6Spy (JDBC logging)
  - Suitable for production
- GC Logging
  - Suitable for production
- GCViewer
  - Suitable for production
- Heap Dumping
  - Suitable for production – but! freezes the JVM so only when necessary
- Eclipse MAT
  - Suitable for production
- VisualVM
  - **NOT** Suitable for production