Groovy Performance: Avoid the Bad, Use the Good, Create the Great

David Clark

September 7, 2016

Contents

1 2 Fast 2 Furious [0/5]

- ☐ Objection #1: Why should I care about this stuff? Computers just get faster!
  - The free lunch is over. Free speedups come from increasing clock speeds, optimizing execution inside the CPU, and increasing the size of the CPU cache. Only CPU caches are getting bigger.

  - When was the last time you bought a computer that was faster than the one you last bought?

- ☐ Objection #2: Isn't this just a matter of choosing the right algorithm?

  - Algorithm complexity does determine execution speed to a large degree
  - However, algorithm complexity always ignores constant factors
  - Those constant factors can be really high
  - A lot of this talk is about reducing those constant factors

- ☐ Objection #3: Won't the JVM optimize this stuff for me?

  - The JVM will optimize a lot of stuff, but it can't work miracles
  - A lot of this talk is about getting groovy out of the way of the JVM so it can optimize
  - A lot of the rest of this talk is about using groovy to help the JVM do its job

- ☐ Objection #4: My application doesn't need to go faster!

  - If this is the case then by all means ignore everything in this talk, move on to your next task
  - But, a lot of applications that are "fast enough" really are not. "Fast enough" is often a lie we tell ourselves because we don't know how to solve the problem.
  - Faster often opens up new use cases for your code. Everyone hated source code control before git. Now git is used by everyone and it has changed how we as programmers work. A lot of this has to do with how fast git is. The concepts of merging and branching were well understoof before git, but people avoided them because they were slow. Now everyone branches and merges several times a day, mostly because git is fast.

- ☐ Objection #5: Groovy is just inherently slow, don't bother.

- Claim: If you give me a java project I can make it at least as fast or faster by adding Groovy. In some sense this is an empty claim, I can give you back the project as is and it's just as fast as it was. However, I do think that adding Groovy can increase both clarity and speed.
- Most of what I have done over the past 3 years is write performance critical code, and I always reach for Groovy.

2 Assumptions [0/5]

- ☐ You actually have a performance problem
- ☐ You know how to measure application performance and how to measure improvements
- ☐ You know the basics of algorithm complexity (O(n) notation)
- ☐ You know the basics of how the JVM operates and what makes for fast executing code
- ☐ If you don't know these things no worries, you will still hopefully get a lot out of this talk, just be careful about applying these techniques. In other words, don't run out and add @CompileStatic to all of your code. Depending on your application it may not actually go any faster and may make your code worse.
- ☐ However, even if these things don't apply, you should still be able to get a lot out of this talk

3 Performance?? [0/3]

- ☐ This is a hard talk to give
  - Performance is a black art
  - Everything has an exception
  - Intuitions are wrong very often
  - Every application is different, very hard to generalize anything
- ☐ We are going to be talking about computation based performance. We want to squeeze as much performance out of the CPU as possible.

- ☐ This assumes you are doing no I/O operations: no sockets, no DB calls, no file operations
  - Performance problems here are usually easier to diagnose and fix
  - Usually it means using threads to parallelize operations and then minimizing thread context switches
  - Alternately, it means using asynchronous I/O will callbacks or promises to parallelize operations

4 Groovy Gotchas (Avoid the Bad) [0/9]

- ☐ Much fewer than there used to be. For the most part compiled static Groovy code is about as performant as Java code.
  - Java 8 is a big win for groovy. Java 8 is very agressive about inlining code.
  - I used to recommend against using arrays directly in Groovy. Once hotspot has optimized a method there is no difference between Groovy and Java array access. This is acutally quite shocking.
  - I used to also have concerns about reference equality in Groovy. I don't remember why, but it also is compiled to the same byte codes as on the JVM.
- ☐ Gotcha #1: Boxing/Unboxing
  - Implicit box/unbox operations can really slow down a hot inner loop.
  - How much of a slow down? I've seen 3x slowdown.
  - However, this isn't really Groovy specific because Java has the same problem.
  - See fastUtils in Groovy class.
- ☐ Gotcha #2: Implcit casting
  - Boxing/unboxing is really just a special case of this.
  - Groovy tends to be much slower about casts because it has to hook into asType().

  - This applies even to primitive casts. Bottom line: avoid casts in hot spots if necessary.
  - Amazingly enough, any type of non-exact match causes this problem. Even assigning to references of a super class, for instance see Groovy.referenceEquals()

- ☐ Gotcha #3: Meta Class based dispatch

  - propertyMissing/methodMissing/invokeMethod are just plain slow. You also can't use them with @CompileStatic or @TypeChecked
  - This also shows up in builders that tend to be implemented through propertyMissing/methodMissing/invokeMethod. However this is usually a case of it not mattering. 99% of the time time after I use a builder I save a the data to disk or send it out of a socket. I/O is orders of magnitude slower than in VM processing and will usually swamp the execution time of the builder.
  - Grails and Gradle have both moved away from these implementation techniques to increase performance

- ☐ Gotcha #4: Inner classes

  - Non-static inner classes "work" but are not natural in groovy and lead to surprising behavior. Some of these are lower performance due to accessing methods and properties via reflection, i.e. slow code.
  - Solution: Use static inner classes for speed and encapsulation or use closures.

- ☐ Gotcha #5: Switch statements

  - Groovy switch statements are SIGNIFICANTLY slower than Java versions. In simple tests. dispatching on integers, Java is 50x faster. Why?
  - Java switches always jump to computed offsets. If you can't jump based on an integer Java won't support your type. This is why it took so long to get String based switches. They had to set in stone how Strings compute hash codes so that offsets could be pre-computed based on absolutely known string hash codes.

  - Groovy switches are driven off of the isCase operator. This means a switch statement is just a series of calls to the isCase method. This is really slow.
  - Solution: Use if/else and fast tests for a small number of tests. For large numbers of tests use fast value types and maps.

- ☐ Gotcha #6: Using ==

  - Calling '==' is very slow, but oddly enough calling equals() is very fast. Huh?
  - Calling '==' results in a call to ScriptBytecodeAdapter.compareEqual() which appears to be the culprit in making things slow.
  - This appears to be a change from a few years ago. If I remember correctly calling equals() explicitly used to result in a call to ScriptBytecodeAdapter.compareEqual(), even in @CompileStatic mode. At least now there is this escape hatch if needed, though I probably won't remember to use it.
  - The actual implementation in Groovy of equals() is quite performant, meaning Groovy classes will be performant in collection types.

- ☐ Gotcha #7: Using out of bounds indexes for arrays/lists

  - This is implemented by catching ArrayIndexOutOfBoundsException, then re-trying with a re-mapped index.
  - I didn't measure this but it just looks slow. In fact I don't see how could it be anything but slower than normal array/list access.

- ☐ Audience participation

  - What have you seen?
  - Acutal measured slowness is preferred to vague intuitions, which are often wrong in dealing with code.

5 Groovy Annotations for Speed (Use the Good) [0/8]

- ☐ @CompileStatic

  - If there is a single thing you remember from this talk it should be use @CompileStatic as a first step when you have an application performance problem.
  - This annotation removes a lot of the dynamicity of Groovy. Properties and methods must be resolvable at compile time, no use of propertyMissing/methodMissing. Types must match. Groovy extension methods such as find/findAll/each are legal. Groovy style casting is legal. Groovy code with @CompileStatic enabled looks a lot like Kotlin code.

- ☐ @TypeChecked
  - Does all of the type checking @CompileStatic does and the same rules apply.
  - However it doesn't do static compilation, just the type checking part, so @TypeCheck'ed code won't actually run any faster than normal Groovy code. So why did I include this?
  - @TypeChecked does prevent dispatch based on propertyMissing/methodMissing which is extraordinarily slow.

- ☐ @Lazy
  - Only initializes a property if it is called; it's a simple one item cache.
  - With volatile it correctly implements double checked locking, which most people don't get right.
  - Very useful for objects which have expensive calls which are not always needed. I've used this extensively for templates with conditional logic. The conditional logic means that some properties are not needed and should not be computed.

- ☐ @Memoized
  - Caches invocations of your method by adding a hidden map to your class
  - If you have not invoked your method with a particular set of parameters, the logic of your method is called, the parameters are added as keys of the hidden map, and the returned value is added as the value of those keys

  - If you have invoked your method with a particular set of parameters, the parameters are used as the key to look up the correct return value in the cache
  - Is a simple, somewhat tunable cache. For simple use cases it gets the job done

- ☐ @Immutable
  - Doesn't make your code faster by itself, but does allow you to do fast things with your class
  - @Immutable classes are inherently thread safe, multi-threading can substantially improve performance
  - @Immutable classes also have correct equals() and hashCode() methods. This means they can be used as keys in maps or added to sets. This means that you can eliminate linear search algorithms O(n) with hash based O(1) algorithms
- ☐ @Sortable
  - Again, it doesn't make your code faster by itself, by does allow you to do fast things with your class
  - @Sortable code is usable in SortedSet, meaning O(n/2) searches become O(ln n) searches
  - @Sortable code is usable in NavigableSet, meaning range searches are now cheap and easy
- ☐ @TailRecursive
  - Specialized, if you don't know what tail recursion is, ignore this for now.
  - If your method is most naturally expressed as a recursion, use this to convert the method to iteration
- ☐ @Slf4j and friends in groovy.util.logging
  - Lots of logging can lead to performance degradation if the logging is done incorrectly
  - Parametrized logging is NOT a valid solution IMNSHO. It doesn't work in all cases and can lead to unnecessary array creation

  - The only way to log correctly is to use log guard statements consistently, like: if(log.isDebugEnabled()) { log.debug(...) }, this is of course a lot of typing, easy to forget, and just plain ugly
  - The Groovy logging annotations give you that for free, they do the right thing every time.

6 What makes these annotations work and can I do the same thing [0/2]

- ☐ The secret is code injection, re-arranging, and re-writing your code
  - @Lazy, @Memoized, and @Slf4j wrap code around your code. The semantics are the same, the JVM will just execute it more efficiently
  - @Sortable and @Immutable add code to your code to enforce semantics of immutability and comparability
  - @TailRecursive re-writes your code to be iterative instead of recursive
  - @CompileStatic generates different byte code than normal groovy code
- ☐ You can absolutely do the same thing

7 Let's do what Groovy does! (Create the Great) [0/4]

- ☐ Re-arrange code at runtime with builders
  - See Grades.groovy
  - Dynamic groovy code is usually the easiest and most natural way to express something
  - However, it may not lead to the most efficient execution
  - Solution: combine easy Groovy syntax with uglier execution, using a builder as your bridge between the two
- ☐ Compile using the Groovy Class Loader (GCL)

  - See Functions.compile
  - Basic idea is that you have a string representation of code and you also have a groovy compiler at all times, make use of both to turn strings into executable code
  - Can also be used to load/reload and optimize scripts at runtime
  - Use this version if you need access to actual class produced

- ☐ Compile using the Groovy Shell
  - See Functions.fromScript
  - Similar to GCL trick. Really it's exactly the same thing since they underneath the covers are doing the same thing.
  - Like GCL the basic problem you are trying to solve is that you need to defer optimization to runtime, but you do want to use the static compiler.
  - Safer than GCL since you can use Secure AST transformations to restrict code being run
  - Don't write parsers, write Groovy DSL's, compile them using the Groovy Shell and write your DSL engines to optimize the code that executes.

- ☐ Transform your code using AST Transformations
  - Beyond the scope of this presentation. I have a presentation I have given in the past, check out my github repo on AST transformations.
  - Basic idea: You want to give the Groovy compiler code other than the one you wrote. Maybe you want to pre-compute something or you want the compiler to generate code based on the code you have written (compile time meta-programming).
  - Idea #1: A better caching library than @Memoized.
  - Idea #2: Minimal Perfect Hashing: Guarantee that hashCode() produces collision free hash codes if the set of objects is known at compile time.
  - Idea #3: Binary parsers/encoder generated based on data known at compile time.
  - Idea #4: Go crazy with Cedric, embed java byte code via Groovy AST transforms: `https://github.com/melix/groovy-bytecode-ast`. I've been wanting to try this myself.

8 Tools For Diagnosing Groovy Performance Problems [0/5]

- ☐ YourKit `https://www.yourkit.com`
  - Best JVM profiler out there.
  - Tells you exactly where your code is slow
  - However, it's pricey, but free for open source projects
- ☐ VisualVM `https://visualvm.java.net/`
  - If you don't have access to YourKit, this gives basic information about in memory JVM behavior
  - Tells you your code is slow, doesn't help as much in locating where it is slow
  - Generally comes with Oracle JDK's these days. OpenJDK may or may not have it.
- ☐ JD-GUI `http://jd.benow.ca/`
  - Best Java bytecode decompiler out there
  - Tells you what a JVM thinks your code look like
  - Great a looking for extra/slow code the Groovy compiler is injecting. As a generalization, the less comprehensible and the uglier your decompiled code is, the worse it will perform. However, this is not a guarantee and exceptions abound.
- ☐ JIT Watch `https://github.com/AdoptOpenJDK/jitwatch`
  - Tells you what the JVM is doing to your code at runtime.
  - Is your code being compiled to native code? Is your code getting inlined? What's the runtime call graph of your methods? JIT Watch can answer all of these questions, which are critical in understanding why your code is fast or slow.
- ☐ Java Bytecode Editor `http://set.ee/jbe/`
  - Valuable for looking at what bytecode is in your class files
  - If you do performance work, you eventually have to start looking at bytecode

9 How do I apply this stuff? [0/2]

- ☐ A lot of these tricks remove a lot of the "Groovyness" out of Groovy. Your code can end up looking like mostly Java code.
- ☐ Good application architecture
  - Bottom layer of your application/library should be a performant engine that makes heavy use of @CompileStatic or Java code
    - This layer should be mostly hidden from users of your library
    - If you need to expose pieces of this part to users, only expose interfaces or abstract classes, you need the ability to change the internals without people depending on details. It's also best if what you expose it immutable.
    - Unit test this layer like crazy, if something goes wrong here, it will be a huge problem
  - Middle layer is for business logic and I/O code
    - No need to @CompileStatic, but @TypeChecked code can be useful here
    - I/O is generally an order of magnitude slower than the slowest Groovy code, so don't bother speeding things up
    - Write clean and idiomatic Groovy here
    - This is the layer an application developer should work at
  - Top Layer is for scripting, DSL's, configuration, and more idiomatic Groovy
    - This is the public API for your application, this is what people should think of when they think about your application
    - Use any dynamic tricks your want here, this code gets executed very few times
    - This is where you show how cool Groovy is