Výkon aplikace a profilování na JVM

Antonín Komenda

Katedra počítačů

Fakulta elektrotechnická

České vysoké učení technické v Praze

Přednáška 11

B0B36PJV – Programování v JAVA

Osnova 90 minut

1. Co na JVM znamená výkon a proč jedno číslo nestačí
2. Jak nás snadno oklame benchmark bez warmupu a bez profileru
3. Praktický workflow: VisualVM nejdřív, CLI nástroje hned vedle
4. Tři dema: CPU hotspot, memory allocation pressure, lock contention
5. Jak ověřit, že fix opravdu pomohl

Výkon není jedno číslo

- **Latency**: doba jedné odpovědi
- **Throughput**: počet odpovědí za sekundu
- **Allocation rate**: kolik “garbage” se vytváří za sekundu
- **Footprint**: kolik paměti se používá

> *Když zlepšíte throughput za cenu dvojnásobného heapu nebo horší latency, není to automaticky výhra.*

Co se často měří špatně

- První běh zahrnuje **class loading**, **JIT warmup** a častěji i víc práce GC.
- Mikrotest bez ochrany proti **dead-code elimination** může být zavádějící.
- `System.nanoTime()` uvnitř jedné smyčky je užitečný, ale jen opatrně.
- Často nevíte, jestli jste CPU-bound, memory-bound nebo lock-bound.

Problém	Typická chyba
Porovnání dvou implementací	pouze jeden průchod bez warmupu
Ladění produkční latence	microbenchmark místo profilace reálného běžícího procesu
GC tuning	přepínání flagů bez znalosti allocation rate a live setu

JVM HotSpot není jen interpreter

- Kód začíná interpretovaně a postupně se profiluje a na základě toho případně kompiluje.
- Pro “hot” metody proběhne **C1/C2 kompilace**, inlining a další optimalizace.
- Část objektů může díky **escape analysis** vypadnout z heapu úplně.
- Proto se chování po 5 sekundách a po 60 sekundách může zásadně lišit.

```
jcmd <pid> Compiler.codecache
jcmd <pid> Compiler.queue
```

Optimalizace startupu: GraalVM Native Image

- GraalVM umí přes `native-image` přeložit Java bytecode **ahead-of-time** do nativního executable.
- Výsledný program neběží na klasické JVM, ale na menším runtime přibaleném do binárky.
- Silná stránka je **start v milisekundách** a **žádný JIT warmup**.
- Dává smysl pro CLI utility, krátce žijící procesy, serverless a služby citlivé na cold start.

```
native-image -jar app.jar
./app
```

„Zero-overhead” startup

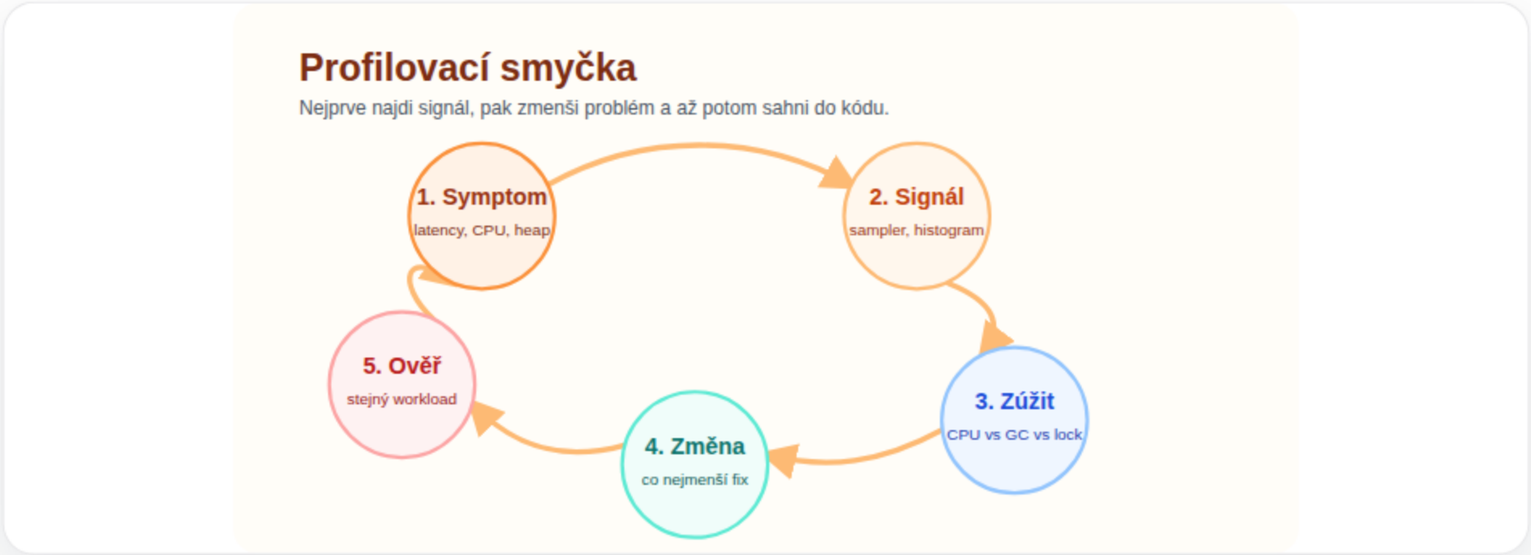
- Přesnější tvrzení je: **velmi nízký startup overhead** a **peak performance hned**, bez JITu.
- Není to doslova nula: pořád startuje proces, inicializuje se framework, čtou se konfigurace a běží I/O.
- Cena za to je **build-time kompilace** a tzv. **closed-world assumption**.
- U reflexe, proxy, resource lookupu nebo jiných dynamických features může být potřeba dodatečná konfigurace.
- U dlouho běžících služeb nemusí Native Image vždy vyhrát nad už “hot” HotSpotem na absolutní throughput.

> *GraalVM přesouvá část práce z runtime do compilation/build fáze.*

Polyglot, jiné jazyky a JNI

- GraalVM Polyglot umí sdílet hodnoty mezi jazyky v jednom procesu a ve stejné paměťové oblasti.
- Dokumentace běžně ukazuje kombinace Java s **JavaScriptem, Pythonem, Ruby, R a LLVM (C, C++, Rust)** kódem.
- V přehledu jazykových runtime dnes nejvíc uvidíte **GraalJS, GraalPy, GraalWasm** a **Espresso (Java on Truffle)**.
- **JNI** tím nezmizelo: když potřebujete existující nativní knihovnu z C/C++, pořád je to relevantní rozhraní.
- V Native Image JNI funguje, ale přístupné třídy, metody a fieldy je potřeba zahrnout do build-time konfigurace.

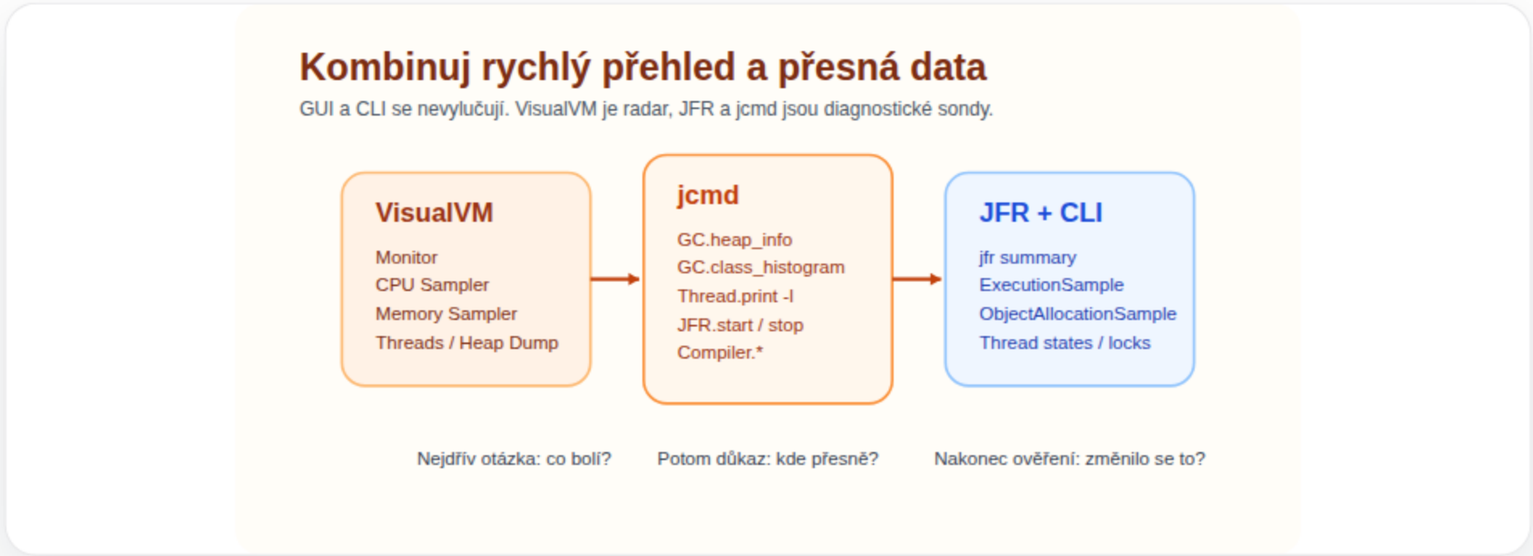
Profilovací smyčka



1. Začněte od symptomu: vysoká latence, CPU, heap nebo timeouty.
2. Najděte **signál**, ne domněnku.
3. Omezte prostor problému: CPU, alokace, GC, IO, contention.
4. Udělejte co nejmenší změnu.
5. Znovu změřte stejný scénář.

> *Profilování není lov náhodných mikrooptimalizací. Je to eliminace největšího bottlenecku.*

Nástroje: rychlá mapa



- **VisualVM**: rychlý a vhodný pro první orientaci.
- **`jcmd` a `jfr` (Java Flight Recorder)**: přesnější analýzy, když už víte, co hledáte.

Nástroje: kdy který použít

Nástroj	Na co je nejlepší
VisualVM	rychlý přehled CPU, heapu, threadů, sampleru
jcmd	stav JVM, histogram tříd, thread dump, start JFR
jfr	lehčí dlouhodobější profiling a export dat
GC log	frekvence kolekcí, pauzy, růst live setu
jstat	rychlý textový pohled na GC trend v čase

CLI quickstart bez IDE

```
jcmd <pid> VM.version
jcmd <pid> GC.heap_info
jcmd <pid> GC.class_histogram
jcmd <pid> Thread.print -l
jcmd <pid> JFR.start name=perf settings=profile duration=30s filename=app.jfr
jfr summary app.jfr
jfr print --events jdk.ExecutionSample,jdk.ObjectAllocationSample app.jfr
```

- `GC.heap_info`: kolik heapu máte a co se děje s generacemi.
- `GC.class_histogram`: které třídy drží nejvíc objektů.
- `Thread.print -l`: kdo čeká, na jakém monitoru a proč.
- `JFR.start`: záznam bez potřeby IDE nebo agentu třetí strany.

VisualVM workflow pro výuku i praxi

- Začněte na **Monitor** tabu: CPU, heap, GC activity, thread count.
- Když je proces zřetelně CPU-heavy, otevřete **Sampler -> CPU**.
- Když roste heap nebo GC, otevřete **Sampler -> Memory**.
- Když aplikace “visí”, podívejte se na **Threads** a pak si vytáhněte `jcmd Thread.print -l`.
- Heap dump berte až ve chvíli, kdy už víte, co hledáte.

> *Nejdřív tvar problému, teprve potom detail. Nezačínejte dumpem nebo 500 řádky logu.*

Jak poznat CPU pressure

- Proces má vysokou CPU zátěž a málo threadů je skutečně blokovaných.
- Heap nemusí růst dramaticky, ale odpovědi jsou pomalé.
- Ve VisualVM CPU sampleru dominuje několik metod.
- V JFR uvidíte hlavně `jdk.ExecutionSample` z jedné části kódu.

Častý pachatel	Typický signál
regex a `String.split()`	hodně CPU zátěže ve `Pattern` / `Matcher`
boxing a stream pipeline	hodně malých objektů a rozsekaný call stack
zbytečné třídění nebo kopie	vysoká CPU zátěž bez výsledného užitku

Demo 1: LogParserDemo

```
if (slowMode) {
    String[] parts = line.split("\\|");
    ParsedRow row = new ParsedRow(parts[1], parts[6],
        Integer.parseInt(parts[4]), Long.parseLong(parts[5]));
    checksum += row.score();
} else {
    checksum += parseFast(line);
}
```

- `slow`: regex split, pomocný record, víc mezikroků a alokací.
- `fast`: jedna lineární parsovací smyčka bez regexu a bez dočasného objektu.
- Oba režimy dělají stejnou logickou práci nad stejnou sadou řádků.

Demo 1: jak ho spustit a co čekat

```
cd examples
./build.sh

java -Xms256m -Xmx256m -cp out demo.perf.LogParserDemo --mode slow
java -Xms256m -Xmx256m -cp out demo.perf.LogParserDemo --mode fast
```

- **VisualVM CPU**: v `slow` režimu uvidíte `String.split`, `Pattern.split`, parsování a record konstrukci.
- **VisualVM Memory**: víc krátkodobých `String[]`, `String` a pomocných objektů.
- **JFR CLI**: `jdk.ExecutionSample` a `jdk.ObjectAllocationSample` ukážou rozdíl bez GUI.

Jak poznat paměťově alokační pressure

- Heap má pilovitý tvar a GC je častý, i když aplikace “nic nedrží”.
- CPU může být část ztraceného času v GC nebo v samotných alokacích.
- Memory sampler ukazuje mnoho krátkodobých objektů stejných tříd.
- JFR typicky ukáže vysoké `jdk.ObjectAllocationSample` a GC události.

> *Alokace v HotSpotu umí být levná. Problém je, když je jejich tempo dlouhodobě vyšší než to, co GC stíhá uklízet.*

Demo 2: `AllocationBurstDemo`

```
if (reuseMode) {
    byte[][] buffers = reusable;
    touch(buffers);
} else {
    byte[][] buffers = new byte[batchSize][];
    for (int i = 0; i < batchSize; i++) {
        buffers[i] = new byte[payloadSize];
    }
    touch(buffers);
}
```

- `churn`: každý batch vytvoří novou sadu `byte[]` bufferů.
- `reuse`: stejné buffery se recyklují mezi batchi.
- V obou případech aplikace počítá checksum, aby se práce neoptimalizovala pryč.

Demo 2: co čekat ve VisualVM a CLI

```
java -Xms128m -Xmx128m -Xlog:gc*:file=alloc.log:time,uptime,level,tags \
  -cp out demo.perf.AllocationBurstDemo --mode churn

jcmd <pid> GC.heap_info
jcmd <pid> JFR.start name=alloc settings=profile duration=20s filename=alloc.jfr
```

- **Monitor**: `churn` má výrazný pilovitý tvar a častější GC.
- **Memory sampler**: dominantní jsou `byte[]`.
- **GC log**: krátké, ale časté GC-young kolekce.
- **reuse** varianta má podobný výsledek a řádově nižší allocation rate.

Jak poznat contention a blokování

- CPU zátěž nemusí být vysoká, ale throughput padá.
- Ve VisualVM v **Threads** uvidíte mnoho vláken ve stavech `BLOCKED` nebo `WAITING`.
- `jcmd Thread.print -l` ukáže, na kterém monitoru nebo locku vlákna stojí.
- Typický problém není GC, ale moc dlouhý kritický úsek.

Symptom	Co hledat
hodně vláken, málo práce	jeden sdílený monitor
timeouty při malé CPU zátěži	fronty, serializace přes jeden lock
throughput neroste s počtem workerů	contention místo reálné paralelizace

Demo 3: LockContentionDemo

```
public synchronized void record(String key, long value) {
    totals.merge(key, value, Long::sum);
    history.addLast(key);
    if (history.size() > historyLimit) {
        history.removeFirst();
    }
    LockSupport.parkNanos(200_000);
}
```

- `synchronized`: všechny workery jdou přes jeden monitor.
- `striped`: `ConcurrentHashMap` + `LongAdder`, krátký nekonkurenční update.
- Záměrně držíme kritický úsek dost dlouho, aby bylo blokování citelné i ve výuce.

Demo 3: co čekat

```
java -cp out demo.perf.LockContentionDemo --mode synchronized --threads 8
java -cp out demo.perf.LockContentionDemo --mode striped --threads 8

jcmd <pid> Thread.print -l
jcmd <pid> JFR.start name=locks settings=profile duration=20s filename=locks.jfr
```

- **VisualVM Threads**: `synchronized` režim ukáže mnoho `BLOCKED` workerů.
- **CLI thread dump**: vlákna čekají na stejném monitoru uvnitř `record`.
- **JFR**: rozdíl mezi tím, kdy má aplikace CPU zátěž a kdy jen stojí ve frontě na lock.

Který nástroj vzít jako první

Matice symptomů a prvního nástroje

Cíl není otevřít vše. Cíl je otevřít správnou věc jako první.

Symptom	První pohled	Další krok	Co ověřit
Vysoké CPU	VisualVM CPU Sampler	JFR ExecutionSample	je hotspot stabilní po warmupu?
Pilovitý heap, časté GC	Monitor + Memory Sampler	GC log + class histogram	allocation rate vs live set
Timeouty, málo CPU	VisualVM Threads	Thread.print -l a JFR locks	který lock serializuje práci?
Paměť stále roste	Histogram tříd	Heap dump + dominators	kdo objekty drží naživu?

- Nevíte-li nic, začněte **VisualVM Monitor** nebo `jcmd`.
- Když proces zatěžuje CPU, jděte na **CPU sampler** nebo **JFR execution samples**.
- Když roste heap, jděte na **Memory sampler**, histogram tříd a GC log.
- Když throughput neroste s vlákny, běžte rovnou na **Threads** a `Thread.print -l`.

Co nedělat

- Neladťe flagy GC, když jste ještě neviděli allocation profile.
- Neřešte lock contention přidáním dalších threadů.
- Neporovnávejte dvě verze kódu bez stejného workloadu a warmupu.
- Nevyvozujte závěr z jediného snapshotu nebo jednoho peeku.
- Neoptimalizujte naslepo metody, které ani nejsou v top CPU zátěže.

> *Nejrychlejší cesta k chybné optimalizaci je opravit to, co jen vypadá podezřele.*

> *Neoptimalizujte předčasně (Premature Optimization).*

Shrnutí

- Výkon na JVM je kombinace **latency, throughputu, alokací a footprintu**.
- Bez warmupu a bez profileru se snadno zaměří špatné místo.
- **VisualVM** je rychlý na prvotní analýzu, **`jcmd`** a **JFR** dávají hlubší vhledy.
- Tři ukázky v `examples/` pokrývají CPU pressure, alokační pressure a contention.
- Nejdřív změřte, potom změňte, nakonec znovu ověřte.