

Správa paměti a výkonnost kódu

Zadání

Zjistěte, zda testovaný kód neobsahuje chyby ve správě paměti. Doporučujeme použít program Purify. Pokud by program takové problémy náhodou neobsahoval, zkuste je uměle vyvolat.

Zjistěte, zda by testovaný kód nemohl být výkonnější. Doporučujeme použít program Quantify.

Řešení

Jelikož se jedná o webovou aplikaci napsanou v jazyce Java a běžící v rámci aplikačního kontejneru (v tomto případě Apache Tomcat), nemohli jsme využít nástroje Purify. Museli jsme se poohlédnout po jiném řešení. Navrhovaný byl Profiler v rámci Netbeans IDE. Samotný projekt byl napsán v prostředí Eclipse, museli jsme tedy převést projekt do Netbeans (což bylo překvapivě snadné) a zprovoznit spouštění aplikace v prostředí Netbeans a možnost jeho profilování. Opět se nám to podařilo. Přesto jsme chtěli zjistit, zda neexistuje profiler i pro Eclipse, našli jsme některá řešení, ale nakonec jsme konečné výsledky zaznamenali s jProfilerem, který poskytl velké možnosti měření různých zátěží systému.

Nestihli jsme však zachytit zajímavé výsledky měření správy paměti přímo na aplikaci. Aplikace se tvářila, že nemá problém se správou paměti. Jistě má na tom lví podíl garbage collector v Javě, o rušení referencí na objekty se programátoři nemuseli u této aplikace starat. Pokoušeli jsme se tedy vyvolat zátěž uměle.

Následuje ukázka kódu, který jsme začlenili do jednoho z Controllerů a snažili se ho vyvolat. Jedná se o cca 2000 dotazů na databázi, ještě s následným dotahováním (bindováním) přidružených dotazů, takže se dostáváme na cca 10000 dotazů na databázi. Ve výsledku trvalo načtení stránky cca 20 vteřin.

```

80         if (errId == null) {
81             // edit a new Err
82             Person loggedPerson = userContextHelper.getContext(request).getLoggedPerson();
83             // edit a new Err without specified ticket
84             form.setErr(service.createErr(loggedPerson));
85         } else {
86             // edit an existing Err
87             form.setErr(service.getErr(errId));
88         }
89
90         form.setAllowedActions(errStateGraph.getValidActions(form.getErr().getState()));
91
92         for (long i = 1; i <= 2000; i++) {
93             try {
94                 Err e = service.getErr(i);
95                 System.out.println("Err: " + e.getId() + " : " + e.getCreator().getPerson().getName() );
96             } catch (DataRetrievalFailureException e) {
97                 System.out.println("Err: " + i + " NOT FOUND.");
98             }
99         }
100
101         return form;
102     }

```

Obrázek 8 - Ukázka kódu vytvářející zvýšenou zátěž

































































Nejdříve jsme zkoumali množství alokované paměti. Největší podíl dle očekávání zabrala vrstva zajišťující přístup k databázi. Chceme-li být konkrétnější, tak metoda, která získává ERR objekt z databáze. Opět zcela očekávané. Zajímavá je metoda, která se umístila jako třetí v pořadí – je to metoda Err.hashCode, kterou my explicitně nevoláme. Zde je nutné vědět, že aplikace využívá framework

Hibernate, který při sestavování kolekce vybraných objektů z databáze velmi často využívá metody hashCode na vybraných objektech, je tedy nutné tyto metody psát s velkou opatrností.

Recorded allocations of: **All classes**

Liveness mode: **Live objects**

Aggregation level: **Methods**

	Hot spot	Allocated memory ▾	Allocations
 	err.dao.hibernate.HibernateErrDao.load	<div></div> 16,123 kB (39%)	291,503
 	org.apache.catalina.core.ContainerBase\$ContainerBackgroundProcessor.run	<div></div> 8,698 kB (21%)	131,985
 	err.domain.Err.hashCode	<div></div> 4,761 kB (11%)	80,465
 	org.apache.catalina.startup.Bootstrap.main	<div></div> 4,100 kB (10%)	91,526
 	org.apache.tomcat.util.threads.ThreadPool\$ControlRunnable.run	<div></div> 1,617 kB (3%)	29,240
 	com.lowagie.text.pdf.Type1Font.process	<div></div> 554 kB (1%)	23,156
 	err.web.controller.EditErrController.formBackingObject	<div></div> 545 kB (1%)	16,305
 	com.lowagie.text.pdf.RandomAccessFileOrArray.readLine	<div></div> 536 kB (1%)	8,854
 	URL: /err/view.html	<div></div> 513 kB (1%)	7,940
 	com.lowagie.text.pdf.GlyphList.<clinit>	<div></div> 499 kB (1%)	21,246
 	err.dao.hibernate.HibernateErrDao\$1.doInHibernate	<div></div> 444 kB (1%)	6,891
 	/mainPage/main.jsp [org.apache.jsp.mainPage.main_jsp]	<div></div> 413 kB (1%)	9,449
 	com.lowagie.text.pdf.BaseFont.getResourceStream	<div></div> 322 kB (0%)	32
 	/viewErr/main.jsp [org.apache.jsp.viewErr.main_jsp]	<div></div> 229 kB (0%)	1,132
 	com.lowagie.text.pdf.PdfEncodings.convertToString	<div></div> 197 kB (0%)	1
 	/errSearch/main.jsp [org.apache.jsp.errSearch.main_jsp]	<div></div> 180 kB (0%)	1,967
 	/errSearch/results.jsp [org.apache.jsp.errSearch.results_jsp]	<div></div> 113 kB (0%)	3,188
 	err.domain.Err.<init>	<div></div> 64,792 bytes (0%)	1,547
 	URL: /err/main.html	<div></div> 58,160 bytes (0%)	843
 	URL: /err/search.html	<div></div> 55,920 bytes (0%)	481
 	err.dao.hibernate.HibernateErrDao.find	<div></div> 48,120 bytes (0%)	868
 	err.service.pdf.ErrPdfGeneratorImpl.<init>	<div></div> 42,720 bytes (0%)	870
 	uk.ltd.getahead.dwr.Messages.<clinit>	<div></div> 42,688 bytes (0%)	711
 	uk.ltd.getahead.dwr.impl.FileProcessor.doFile	<div></div> 41,592 bytes (0%)	30
 	uk.ltd.getahead.dwr.util.JavascriptUtil.trimLines	<div></div> 39,744 bytes (0%)	2
 	commons.hibernate.support.EnumUserType.nullSafeGet	<div></div> 39,600 bytes (0%)	1,074
 	com.lowagie.text.pdf.PdfObject.<init>	<div></div> 36,864 bytes (0%)	1,536
 	com.lowagie.text.pdf.ByteBuffer.toByteArray	<div></div> 35,808 bytes (0%)	1,536
 	err.web.controller.SearchErrController.initBinder	<div></div> 35,576 bytes (0%)	964
 	commons.domain.Entity.<init>	<div></div> 35,152 bytes (0%)	763
 	err.web.controller.ERRMainController.handleRequestInt...	<div></div> 27,984 bytes (0%)	787
 	uk.ltd.getahead.dwr.impl.DefaultContainer.configurationFinished	<div></div> 21,456 bytes (0%)	579

Obrázek 9 - Alokace paměti pro jednotlivé třídy

Abychom si byli jistí, že se jedná opravdu o námi vyvolanou chybu, podívali jsme se ještě na tzv. Call tree. Tzn. strom volání, ze kterého jsme zjistili, že nadměrné volání dotazů na databázi pochází opravdu z námi upraveného Controller objektu.

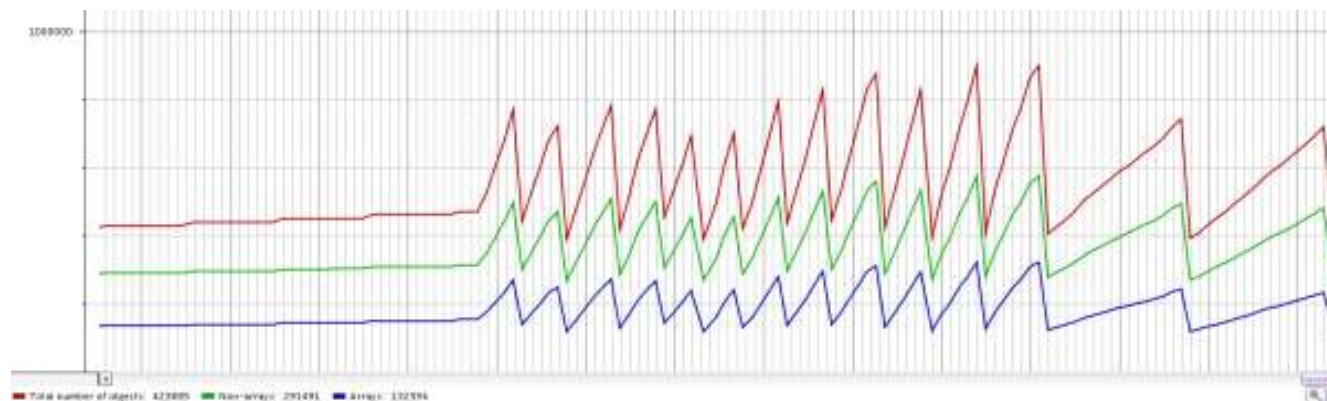


Obrázek 10 - Strom volání náročné části

Dále nás zajímalo vytížení procesoru při tak náročném požadavku. Zvýšený „hřeben“ na obrázku níže časově odpovídá generování stránky. Podobně vypadá i diagram alokované paměti. Je na něm dobře patrné, kdy začíná pracovat javovský garbage collector a uvolňuje již nepotřebnou paměť.



Obrázek 11 - Vytížení procesoru



Obrázek 12 - Počet objektů alokovaných v paměti