

ПРОФИЛИРОВАНИЕ JAVA ПРИЛОЖЕНИЯ

Параметры системы

Название	Значение
ПК	Lenovo ThinkPad T570
Процессор	Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz
ОЗУ	8192 MB DDR4 @ 2133 MHz
ОС	16.04.1-Ubuntu x86_64
Java	Java(TM) SE Runtime Environment (build 1.8.0_144-b01)
JVM	Java HotSpot(TM) 64-Bit Server VM (build 25.144-b01, mixed mode)
JMC	5.5.1
jmap	из JDK 1.8.0_144
maven	3.3.9

Репозиторий

В репозитории [hh-jvm](#) есть четыре ветки:

- **master** - ветка исходного репозитория [money-transfer](#)
- **automation** - ветка автоматизации профилирования, тут созданы файлы `profile.sh` скрипт автоматизирующий профилирование,
- `flight-recorder.jfc` файл с настройками для FlightRecorder и `analyze.py` файл для анализа вывода скрипта `run_tests.py`
- **develop** - ветка с изменениями в java коде для улучшения производительности.
- **report** - ветка с этим отчётом

Стадии экспериментов помечены тегами `stage_<#стадии>`. Дампы памяти и записи параметров на github не выложены, они занимают много места.

Стадии экспериментов

Эксперименты проводятся в несколько стадий, каждая стадия отличается от предыдущей изменениями в коде и на каждой стадии снимаются показатели производительности текущей версии кода. Всего есть 4 стадии :

- Стадия 0 (тег `stage_0`): код без изменений. На этой стадии выделяются основные ошибки, которые удастся найти с помощью профилировщика.
- Стадия 1 (тег `stage_1`): код содержит исправления, найденные на предыдущей стадии, а также при инспекции кода
- Стадия 2 (тег `stage_2`): реализация DAO переписана, хранение реализовано в структурах данных Java.
- Стадия 3 (тег `stage_3`): анализ различных GC

Инструменты

Для замеров используются следующие инструменты:

- Самописный bash скрипт `'profile.sh'` для автоматизации замеров и обеспечения хотябы небольшой степени воспроизводимости экспериментов.
- Самописный python скрипт `'analyze.py'` для построения графика времени выполнения сценария 1.
- [FlightRecorder](#) для записи `'jfr'` файлов с различными измеряемыми параметрами, перечисленными в файле `'flight-recorder.jfc'`
- [Java Mission Control](#) для анализа `'jfr'` файлов. Файлы `'jfr'` объемные и в репозиторий не входят.
- `jmap` для сохранения бинарный дампов памяти. Дампы большие, в репозиторий не входят.
- `jvisualvm` для анализа дампов памяти и фильтарции объектов по пакету

Замеряемые величины

Согласно заданию, требуется контролировать следующие параметры:

- На сколько загружен CPU - оценивается по записям FlightRecorder
- Сколько в среднем потребляется памяти, замечен ли в программе memory leak - оценивается по записям FlightRecorder
- Как часто происходит сборка мусора - оценивается по записям FlightRecorder, тут же оцениваются дампы памяти 'hprof' с помощью jvisualvm
- Сколько в среднем выполняется запуск сценария 1, как быстро увеличивается это время - оценивается по графику, построенному с помощью 'analyze.py'
- Какие операции из значимых (т.е. без учета работы системных функций, в т.ч. веб сервера) занимают больше всего процессорного времени - оценивается по записям FlightRecorder

Методика

В `run_test.py` добавлен вывод прошедшего с момента запуска времени, этот вывод мы будем перенаправлять в файл `.plg` (python log) и потом анализировать. На каждой стадии запускаем самописный `profile.sh` скрипт командой `./profile.sh -clear && ./profile.sh -dump && sleep 30 && ./profile.sh`

- `./profile.sh -clear` чистит старые дампы и `textttjfr` файлы и делает `textttmvn clean install`
- `./profile.sh -dump` запускает `jar`-ник и раз в 2 мин делает дампы через `textttjmap` в течении 10 мин
- `./profile.sh` - записывает `jfr` файл в течении 10 мин с помощью FlightRecorder и JMC .

Таким образом, сохранение дампов и запись `jfr` файлов происходит на разных запусках `jar`-ника. После выполнения скрипта получаем файлы:

- два `jfr` - один тот, в процессе записи которого не создавались дампы, а второй записывался одновременно с созданием дампов. Для анализа интересен первый
- два `plg` файла, это файлы output потока python скрипта `run_test.py`, в который мы ещё добавили время для каждой итерации. Их также два так как один во время "чистого" прогона, а второй во время сохранения дампов памяти. Интересен первый.
- дампы памяти, их как правило 5, так как 10 минут пишем через каждые 2 мин. Анализируем средний, третий, созданный через 6 мин после начала записи

Полученные файлы анализируются с помощью `jvisualvm` (для дампов) и JMC (для `.jfr`), а также самописным python скриптом 'analyze.py' (для построения графика времени выполнения сценария 1 по `.plg` файлу)

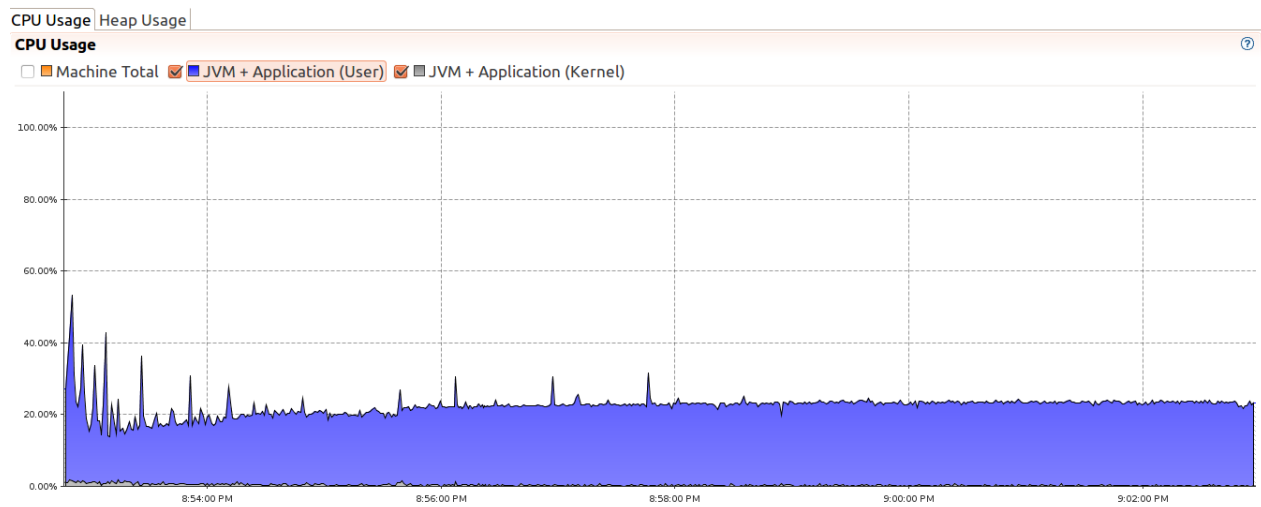
Стадия 0

Анализ производительности приложения до изменений кода.

- Выясняется, что сохранение дампов во время работы практически не сказывается на производительности, поэтому на самом деле можно запускать один раз на 10 мин `./profile.sh -dump`, вместо того чтобы за первый проход собирать дампы, а за второй писать события FlightRecorder'ом.

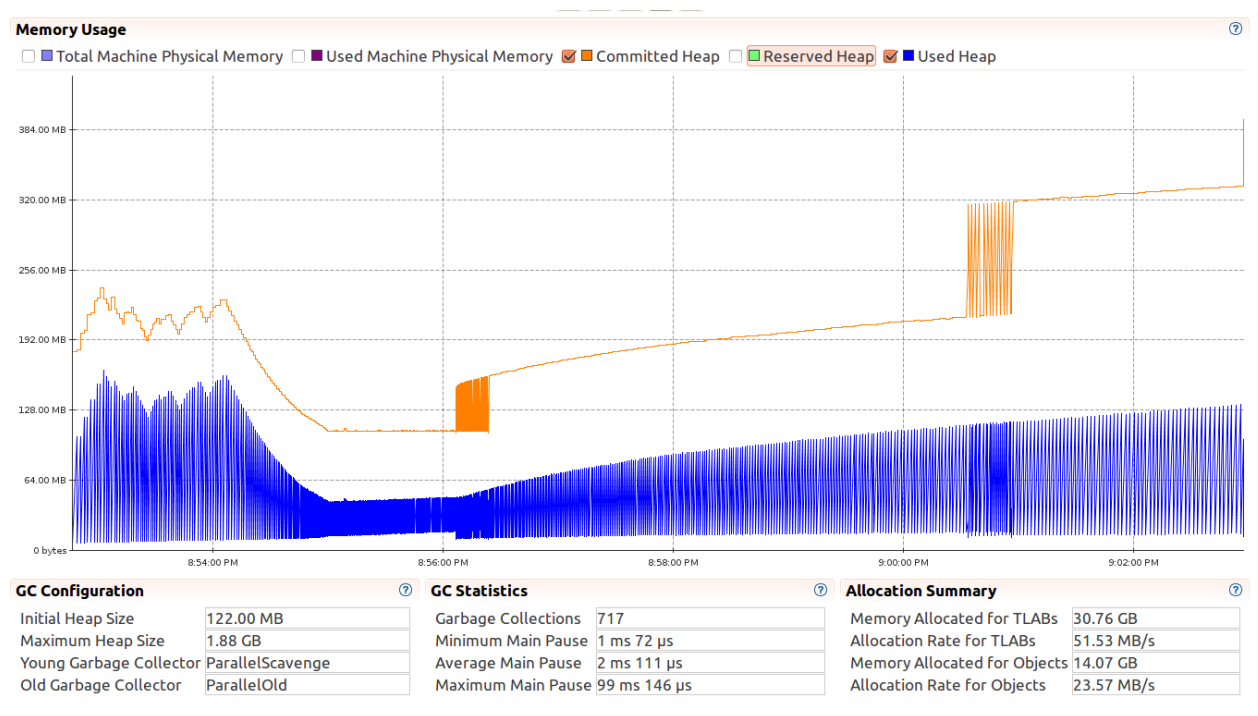
- На сколько загружен CPU?

Анализируем загрузку CPU из записанного jfr файла. Она в среднем составляет 23%.



- Сколько в среднем потребляется памяти, заметен ли в программе memory leak?

Использование heap от 50 Мб до 145Мб:



Дамп #3

Class Name	Instances [%]	Instances	Size
com.moneytransfer.model. Account		18,388 (3.2%)	882,624 (1.5%)
com.moneytransfer.model. User		10,620 (1.9%)	424,800 (0.7%)
com.moneytransfer.dao.impl. AccountDAOImpl		22 (0%)	352 (0%)
com.moneytransfer.dao. H2DAOFactory		20 (0%)	320 (0%)
com.moneytransfer.service. AccountService		13 (0%)	312 (0%)
com.moneytransfer.dao.impl. UserDAOImpl		9 (0%)	216 (0%)
com.moneytransfer.dao.impl. AccountDAOImpl\$\$L...		4 (0%)	128 (0%)
com.moneytransfer.service. UserService		4 (0%)	128 (0%)
com.moneytransfer.model. Transaction		2 (0%)	96 (0%)
com.moneytransfer.service. TransactionService		2 (0%)	48 (0%)
com.moneytransfer.service. AccountService\$\$La...		1 (0%)	16 (0%)
com.moneytransfer.service. UserService\$\$Lambd...		1 (0%)	16 (0%)
com.moneytransfer.dao.impl. UserDAOImpl\$\$Lam...		1 (0%)	16 (0%)
com.moneytransfer.model. MoneyUtil		1 (0%)	28 (0%)
com.moneytransfer.model. MoneyUtil[]		1 (0%)	32 (0%)
com.moneytransfer.service. ServiceExceptionMap...		1 (0%)	16 (0%)
com.moneytransfer.exception. CustomException		0 (0%)	0 (0%)
com.moneytransfer.utils. Utils		0 (0%)	0 (0%)
com.moneytransfer.dao. AccountDAO		0 (0%)	0 (0%)
com.moneytransfer.dao. UserDAO		0 (0%)	0 (0%)
com.moneytransfer.dao. DAOFactory		0 (0%)	0 (0%)
com.moneytransfer. Application		0 (0%)	0 (0%)

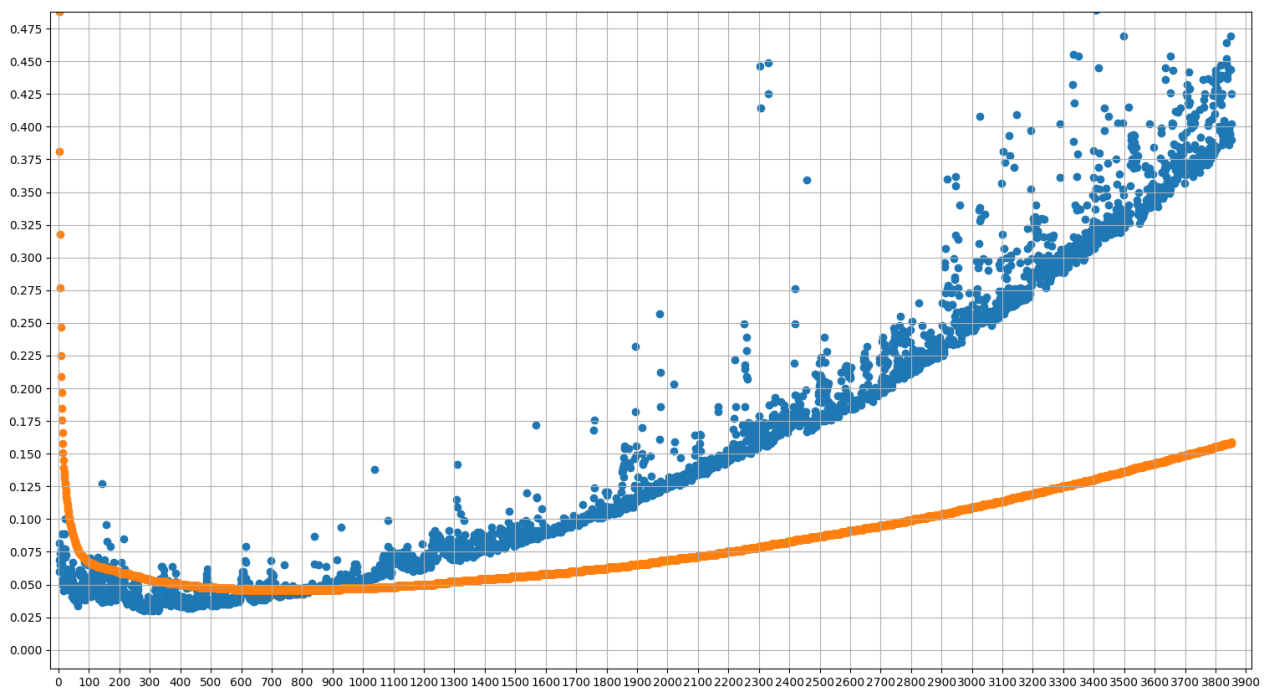
Относительно много объектов **Account** и **User**, почему-то создано несколько объектов типа **DAOImpl**.

- Как часто происходит сборка мусора?

За время эксперимента (10мин 11с) произошло 717 раз. Средняя пауза 2 ms 111 us, максимальная 99 ms 146 us.

- Сколько в среднем выполняется запуск сценария 1, как быстро увеличивается это время?

На графике показано среднее время **Execution time average** и текущее время **Execution time last** в секундах зависимости от числа выполнения сценария 1 **Function play_scenario_1 called**. Текущее время выросло с 0.03 с до 0.4с за 3850 выполнений сценария.



- Какие операции из значимых (т.е. без учёта работы системных функций, в т.ч. веб сервера) занимают больше всего процессорного времени?

Семплирование, применённое при записи, имеет ограниченную точность и общее число вызова значимых функций не велико, в основном процессор выполняет функции из jdk или веб сервера, но оценить узкие места можно при помощи таблицы:

Stack Trace	Sample Count	Percentage
com.moneytransfer.dao.impl.AccountDAOImpl.getAllAccounts()	102	0.31%
com.moneytransfer.dao.impl.UserDAOImpl.getAllUsers()	52	0.16%
com.moneytransfer.model.Account.toString()	12	0.04%
com.moneytransfer.dao.impl.AccountDAOImpl.transferAccountBalance(UserTransaction)	2	0.01%
com.moneytransfer.service.AccountService.withdraw(long, BigDecimal)	1	0.00%
com.moneytransfer.service.AccountService.createAccount(Account)	1	0.00%
com.moneytransfer.dao.impl.AccountDAOImpl.getAccountByUser(String, String)	1	0.00%
com.moneytransfer.dao.H2DAOFactory.getConnection()	1	0.00%
com.moneytransfer.model.Account.equals(Object)	1	0.00%
com.moneytransfer.service.UserService.<init>()	1	0.00%
com.moneytransfer.dao.H2DAOFactory.getAccountDAO()	1	0.00%
com.moneytransfer.dao.impl.AccountDAOImpl.createAccount(Account)	1	0.00%
com.moneytransfer.dao.impl.UserDAOImpl.insertUser(User)	1	0.00%

• ВЫВОДЫ СТАДИИ 0

1. Проверить как создаются и куда сохраняются объекты `Account` и `User`.
2. Проверить почему создаётся несколько объектов типа `*DAOImpl` и `H2DAOFactory`
3. Проверить вызов методов `AccountDAOImpl.getAllAccounts()`, `UserDAOImpl.getAllUsers()`

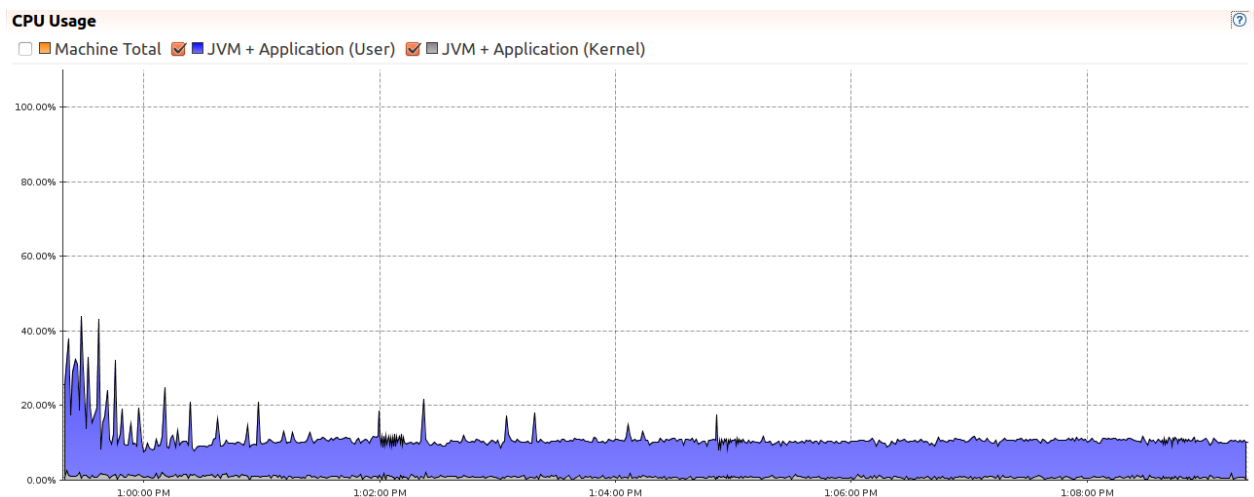
Стадия 1

При анализе найдено, что

1. Можно переиспользовать `DAOImpl` так как это объекты без состояния
2. Есть утечка памяти, когда в объекте `UserDAOImpl` сохраняются неиспользуемые объекты `User`

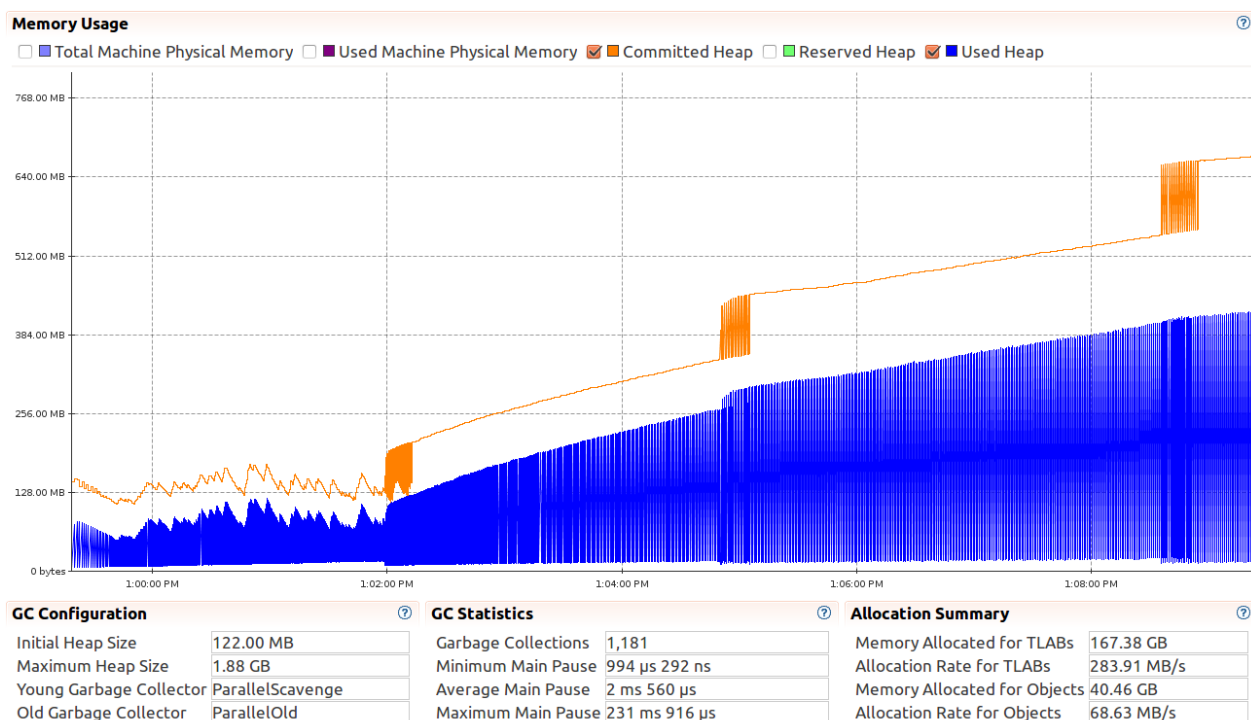
- На сколько загружен CPU?

Загрузка ЦП уменьшилась до 15% в среднем.



- Сколько в среднем потребляется памяти, заметен ли в программе memory leak?

Использование heap составляет от 50 Мб до 400Мб:



Дамп #3

Classes [Compare with another heap dump](#)

Class Name	Instances [%]	Instances	Size
com.moneytransfer.model. Account		10,725 (0.7%)	514,800 (0.4%)
com.moneytransfer.model. User		10,391 (0.7%)	415,640 (0.4%)
com.moneytransfer.service. AccountService		223 (0%)	5,352 (0%)
com.moneytransfer.service. UserService		76 (0%)	1,824 (0%)
com.moneytransfer.model. UserTransaction		37 (0%)	1,776 (0%)
com.moneytransfer.service. TransactionService		37 (0%)	888 (0%)
com.moneytransfer.model. MoneyUtil		1 (0%)	28 (0%)
com.moneytransfer.model. MoneyUtil[]		1 (0%)	32 (0%)
com.moneytransfer.service. ServiceException...		1 (0%)	16 (0%)
com.moneytransfer.dao.impl. AccountDAOImpl		1 (0%)	16 (0%)
com.moneytransfer.dao.impl. UserDAOImpl		1 (0%)	16 (0%)
com.moneytransfer.dao. H2DAOFactory		1 (0%)	16 (0%)
com.moneytransfer.exception. CustomException		0 (0%)	0 (0%)
com.moneytransfer.utils. Utils		0 (0%)	0 (0%)
com.moneytransfer.dao. UserDAO		0 (0%)	0 (0%)
com.moneytransfer.dao. AccountDAO		0 (0%)	0 (0%)
com.moneytransfer.dao. DAOFactory		0 (0%)	0 (0%)
com.moneytransfer. Application		0 (0%)	0 (0%)

money

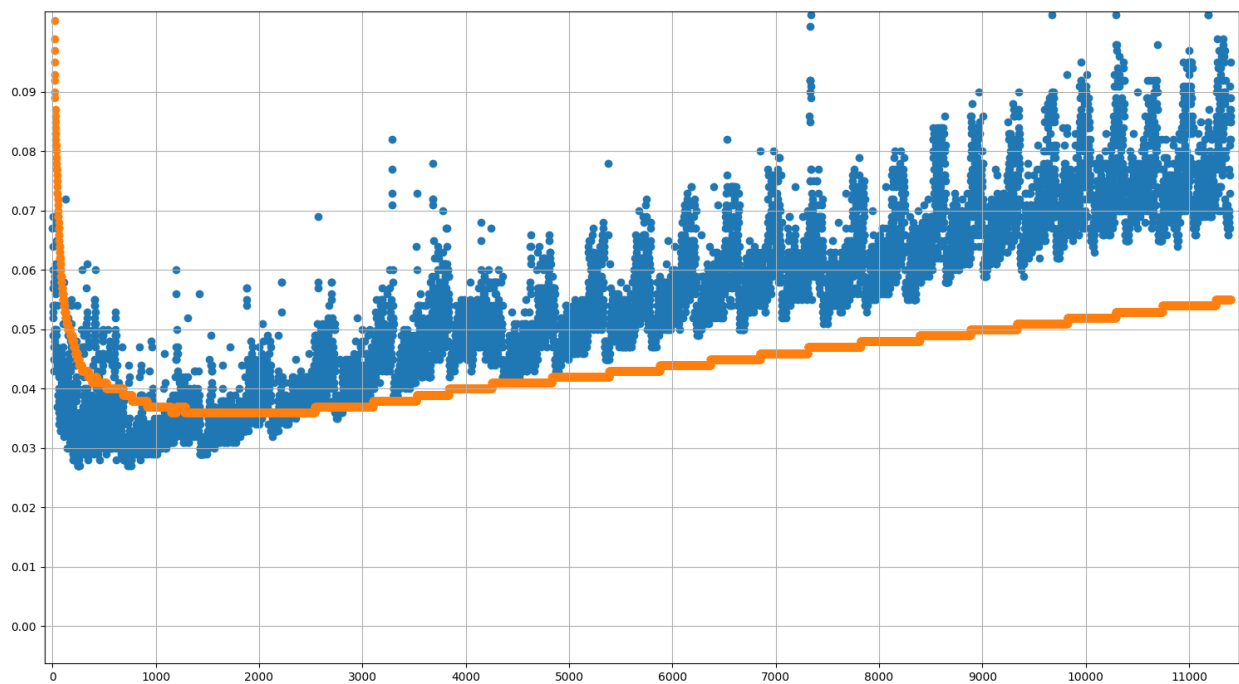
Число объектов сократилось, но не радикально, такие изменения могут быть вызваны временем

- Как часто происходит сборка мусора?

За время эксперимента произошла 1181 раз. Средняя пауза 2 ms 560 us, максимальная 231 ms 916 us.

- Сколько в среднем выполняется запуск сценария 1, как быстро увеличивается это время?

Текущее время выросло с 0.035 с до 0.48 с за 11300 выполнений сценария 1. Таким образом, производительность существенно возросла. На предыдущей стадии 0 сценарий 1 был выполнен всего 3850 раз. Рост времени выполнения сценария 1 существенно более медленный.



- Какие операции из значимых (т.е. без учёта работы системных функций, в т.ч. веб сервера) занимают больше всего процессорного времени?

Семплирование, применённое при записи, имеет ограниченную точность и общее число вызова значимых функций не велико, в основном процессор выполняет функции из jdk или веб сервера, но оценить узкие места можно при помощи таблицы:

Hot Methods		
Filter Column	Stack Trace	*money*
Stack Trace	Sample Count	Percentage
com.moneytransfer.service.AccountService.getAllAccounts()	321	2.66%
com.moneytransfer.dao.impl.UserDAOImpl.getAllUsers()	198	1.64%
com.moneytransfer.dao.impl.AccountDAOImpl.getAllAccounts()	51	0.42%
com.moneytransfer.service.UserService.getAllUsers()	20	0.17%
com.moneytransfer.model.Account.hashCode()	10	0.08%
com.moneytransfer.dao.impl.AccountDAOImpl.transferAccountBalance(UserTransaction)	3	0.02%
com.moneytransfer.service.AccountService.withdraw(long, BigDecimal)	1	0.01%
com.moneytransfer.service.TransactionService.transferFund(UserTransaction)	1	0.01%
com.moneytransfer.dao.impl.UserDAOImpl.getUserById(long)	1	0.01%

• ВЫВОДЫ СТАДИИ 1

1. Загрузка процессора снизилась с 25% до 15%.
2. Потребление памяти увеличилось 50–145 Мб до 50–400 Мб, возможно это связано с общим ростом производительности, а не с утечкой памяти. Мне не удалось найти больше кода, который мог бы вызывать утечку памяти. Анализ дампов также не проясняет этот вопрос.
3. Общая производительность существенно возросла согласно результатам замера времени выполнения сценария 1.