ПРОФИЛИРОВАНИЕ JAVA ПРИЛОЖЕНИЯ

Праметры 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 отчётом

Стадии экспериментов помечены тегами `stage_<#стадии>`. Дампы памяти и записи параметров на github не выложены, они знимают 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 3 стадии :

- Стадия 0 (тег `stage_0`): код без изменений. На этой стадии выделяются основные ошибки, которые удается 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 на `HashMap`.

Инструменты

Для замеров используются следующие инструменты:

- Самописный bash скрипт `profile.sh` для автоматизации замеров и обеспечения некоторой 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, заметен 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 помошью `analyze.py`
- Какие операции из значимых (т.е. без учёта 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 `jfr` файлы и делает textttmvn clean install
- `./profile.sh --dump` — запускает jar-ник и раз в 2 мин делает дамп 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` файла — один тот, в процессе записи которого не создавались дампы, а второй записывался одновременно с созданием дампов. Для анализа интересен первый
- два `plg` файла — файлы output потока python скрипта `run_test.py`, в который ещё добавлено время для каждой итерации. Один во время "чистого"прогона, а второй во время сохранения дампов памяти. Интересен первый.
- 5 дампов памяти, так как их пишем через каждые 2 мин в течении 10 минут . 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)

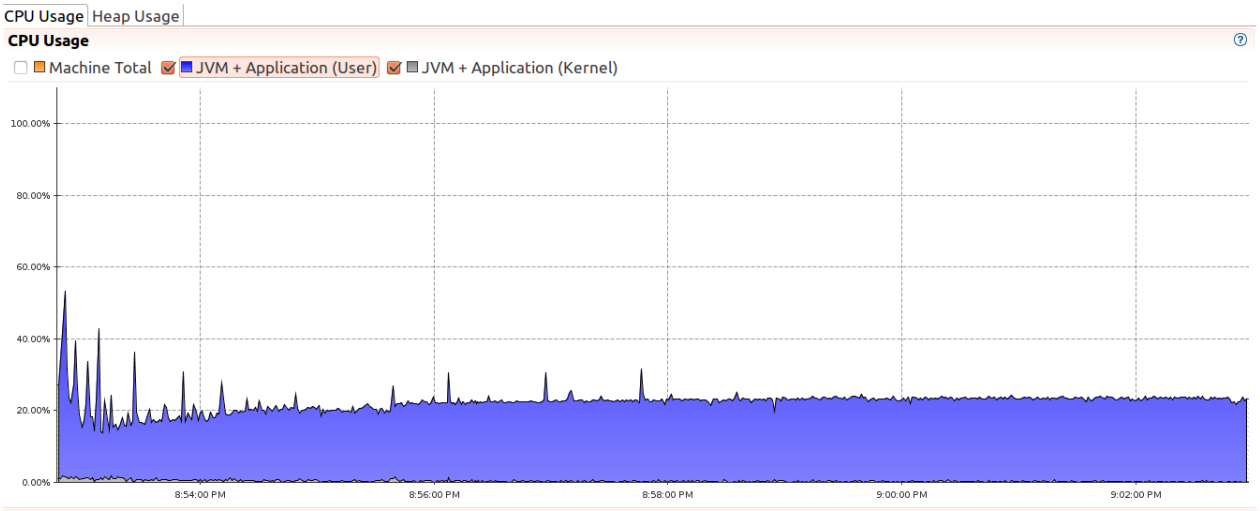
Стадия 0

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.

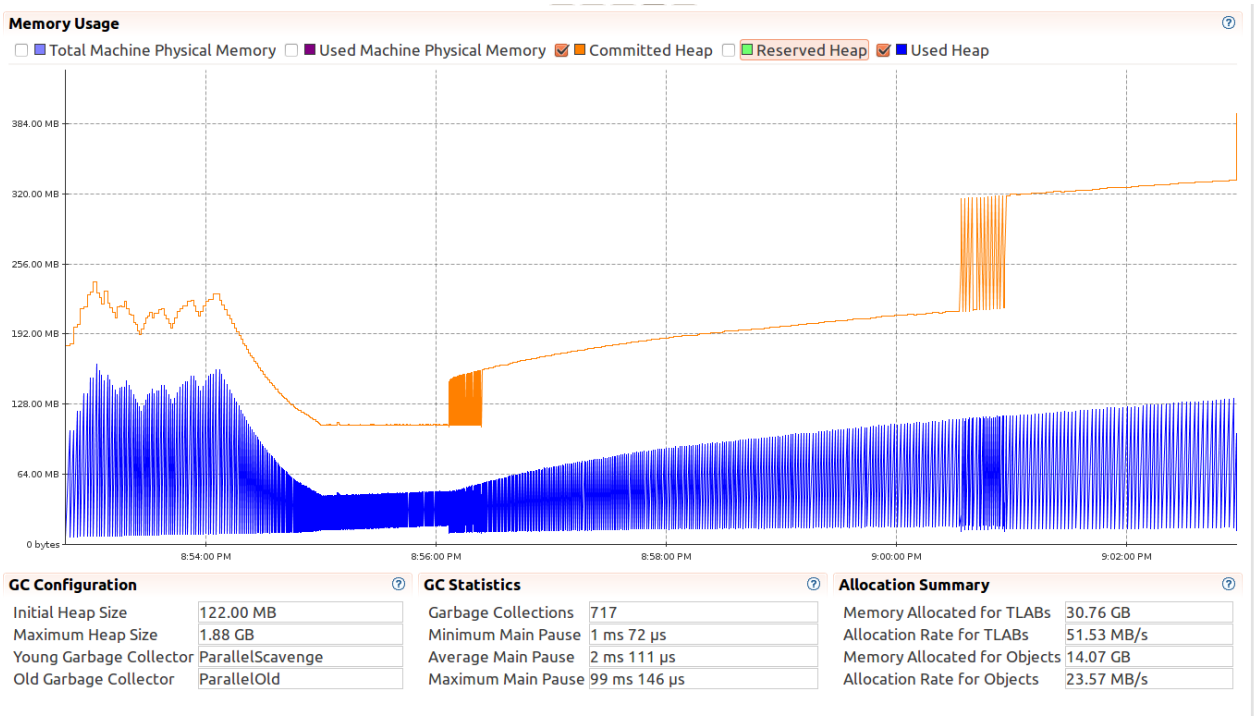
- *На сколько загружен CPU?*

  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

Classes — Compare with another heap dump

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$$L...**		4	(0%)	128	(0%)
com.moneytransfer.service.**UserService**		4	(0%)	128	(0%)
com.moneytransfer.model.**UserTransaction**		2	(0%)	96	(0%)
com.moneytransfer.service.**TransactionService**		2	(0%)	48	(0%)
com.moneytransfer.service.**AccountService$$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%)

money

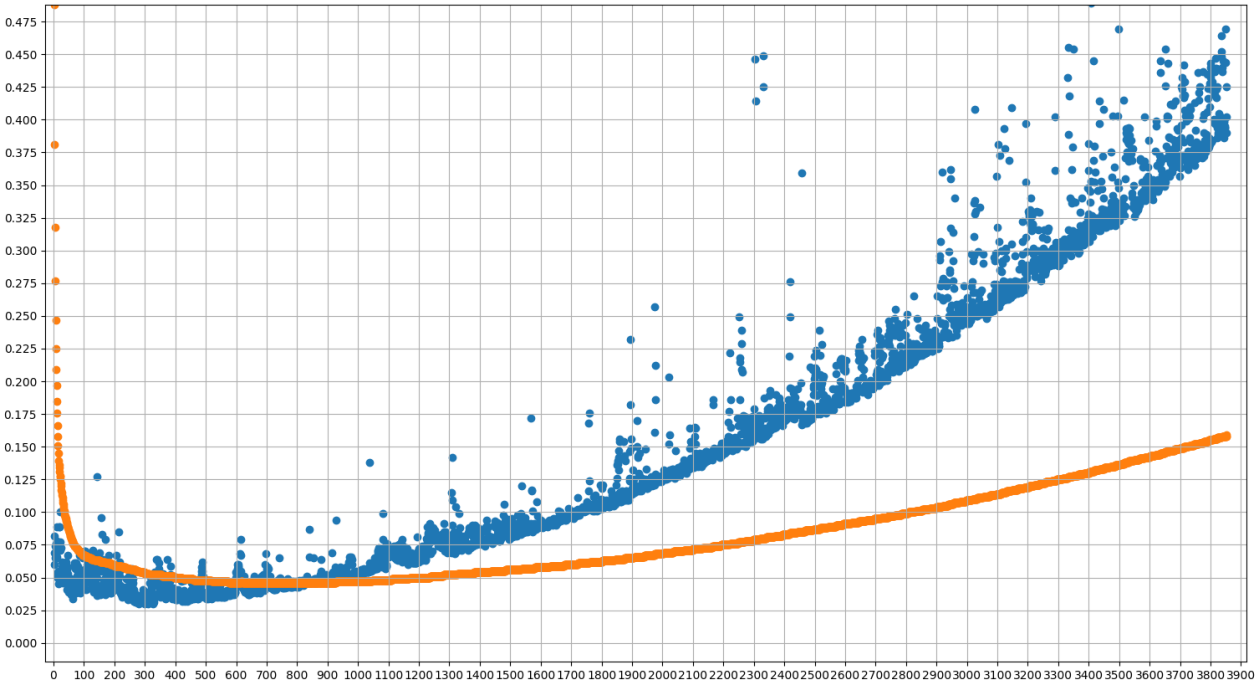
Относительно много объектов `Account` и `User`, почему-то создано несколько объектов типа `DAOImpl`.

- *Как часто происходит сборка мусора?*

  За время эксперимента (10мин 11с) произошла 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 текщее 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 0.4 с 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()`

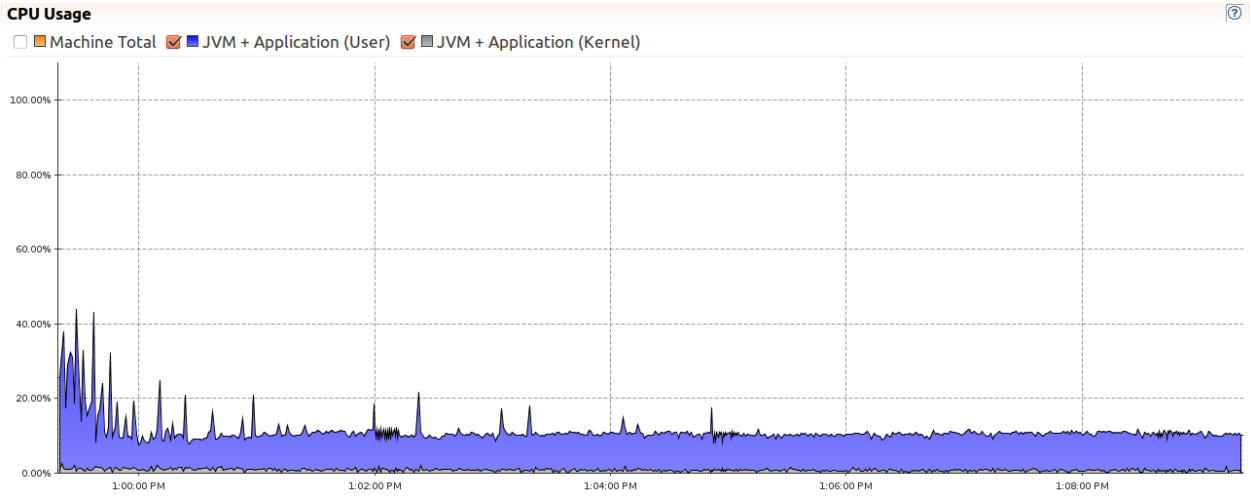
Стадия 1

При анализе найдено, что

1. `9b81cd4` В классе есть `UserDAOImpl` есть неиспользуемое поле `fatched`
2. `fa418f8` Объекты типа `DAO` можно переиспользовать, не создавая заново новые объекты, так как они не имеют состояния
3. `501dd00` В классе `UserTransaction` в методе `equals()` можно переупорядочить порядок сравнения полей
4. `e9eed2d` Обнаружена ещё одна утечка памяти, экземпляры `Pattern` можно переиспользовать
5. `d78be31` В методах `getDAO` можно убрать `loadDriver`
6. `42a5108` В классе `Account` `hashCode` возвращал константу
7. `d6ee875` В методе `getAccountByUser()` неправильно возвращается значение
8. `9cb79c3` В таблице `User` можно установить `AUTO_INCREMENT` и возвращать значение, сгенерированное базой данных
9. `83dd1f1` Можно переиспользовать объекты `H2DAOFactory`
10. `68a205b` Переписаны методы `getAllAccounts` и `getAllUsers` для использования `StringBuilder` (не уверен, не будет ли старая реализация создавать и соединять лишние строки)

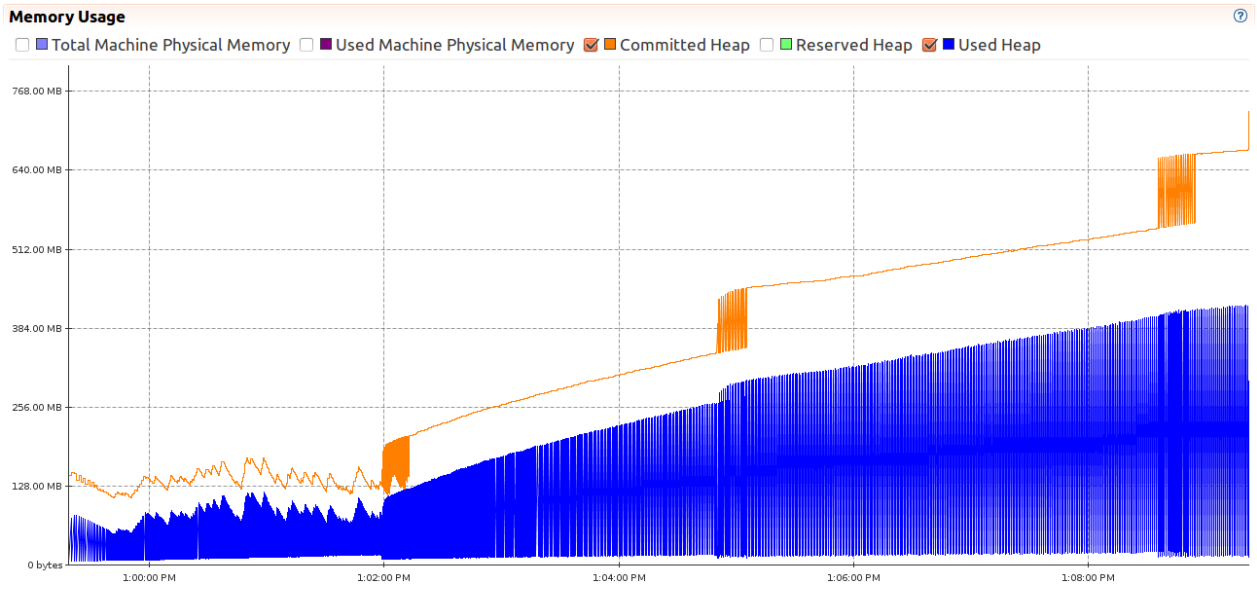
- *На сколько загружен CPU?*

  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:



GC Configuration

Initial Heap Size	122.00 MB
Maximum Heap Size	1.88 GB
Young Garbage Collector	ParallelScavenge
Old Garbage Collector	ParallelOld

GC Statistics

Garbage Collections	1,181
Minimum Main Pause	994 µs 292 ns
Average Main Pause	2 ms 560 µs
Maximum Main Pause	231 ms 916 µs

Allocation Summary

Memory Allocated for TLABs	167.38 GB
Allocation Rate for TLABs	283.91 MB/s
Memory Allocated for Objects	40.46 GB
Allocation Rate for Objects	68.63 MB/s

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

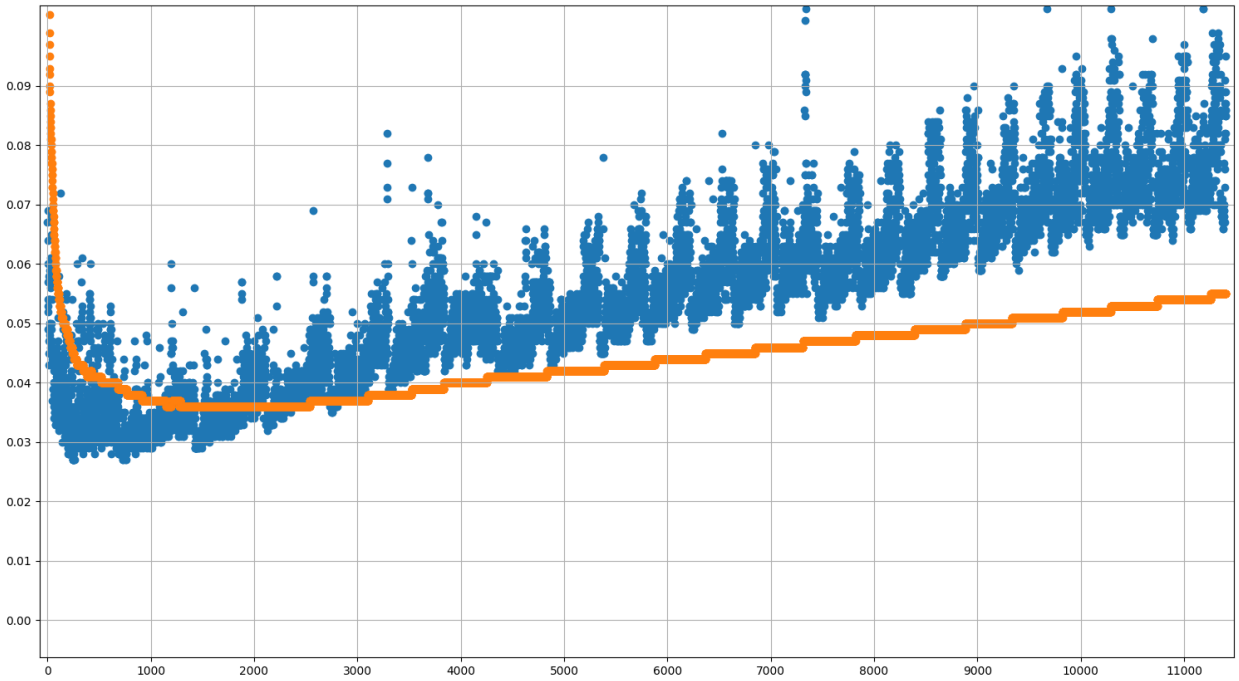
Число объектов сократилось, но не радикально, такие изменения могут быть вызваны временем

- *Как часто происходит сборка мусора?*

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

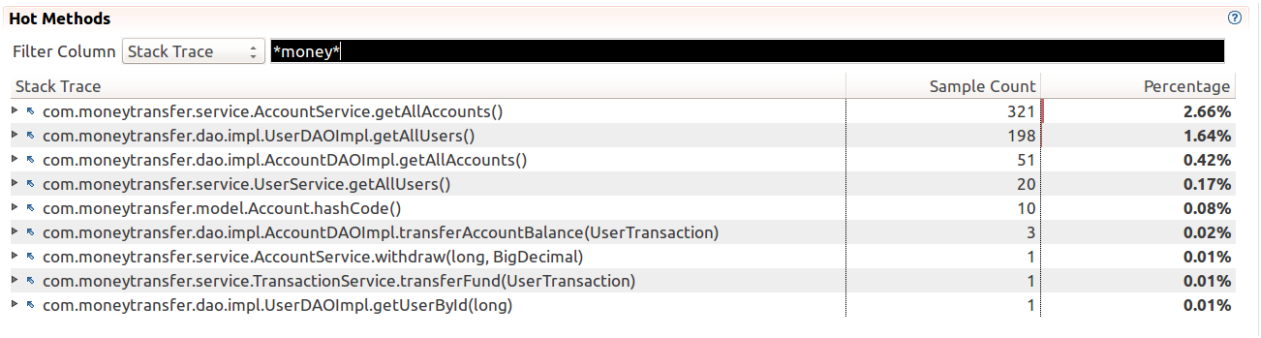
- *Сколько в среднем выполняется запуск сценария 1, как быстро увеличивается это время?*

  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?*

  Статистика использования методов изменилась не сильно, возможно она зависит только от характера загрузки сервера.



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**

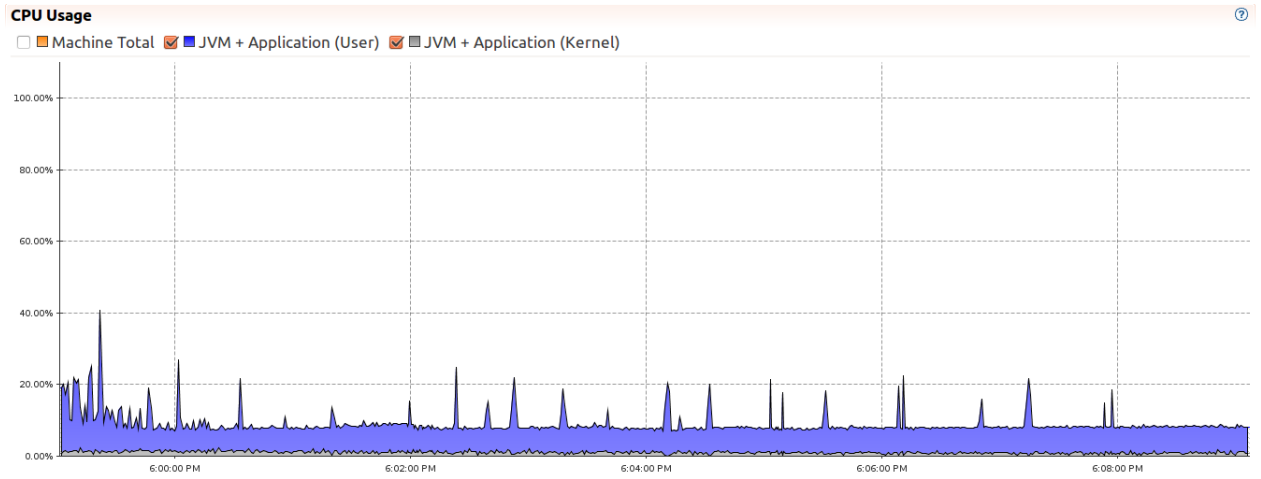
  1. Загрузка процессора снизилась с 25% до 15%.
  2. Потребление памяти увеличилось 50–145 Мб до 50–400 Мб, возможно это связано с общим ростом производительности, а не с утечкой памяти. Мне не удалось найти больше кода, который мог бы вызывать утечку памяти. Анализ дампов также не проясняет этот вопрос.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

Стадия 2

1. `9b81cd4` В классе есть `UserDAOImpl` есть неиспользуемое поле `fatched`

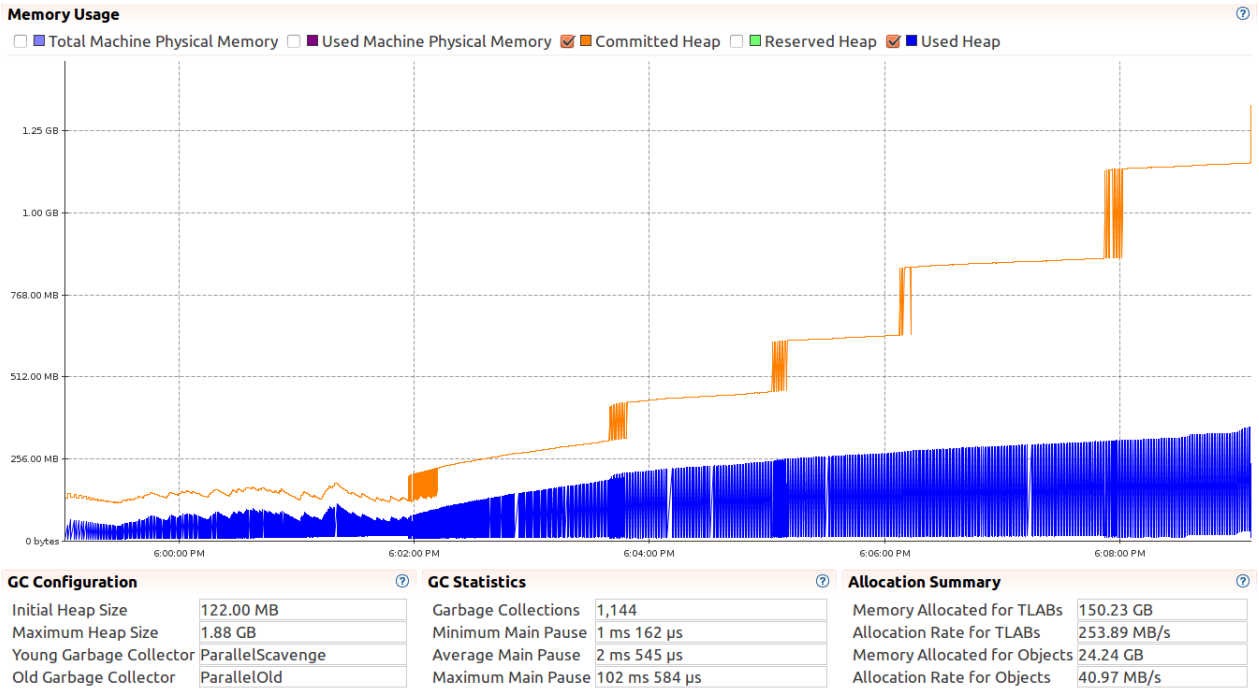
- *На сколько загружен CPU?*

  Загрузка ЦП уменьшилась до $\sim$ 12%в среднем.

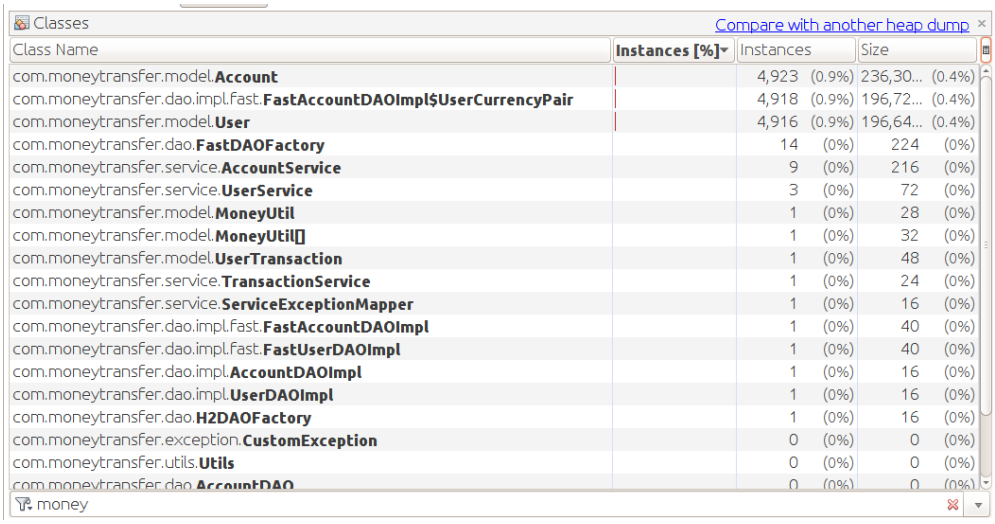


- *Сколько в среднем потребляется памяти, заметен ли в программе memory leak?*

  Использование heap составляет от 50 Мб до 370Мб:



Дамп #3



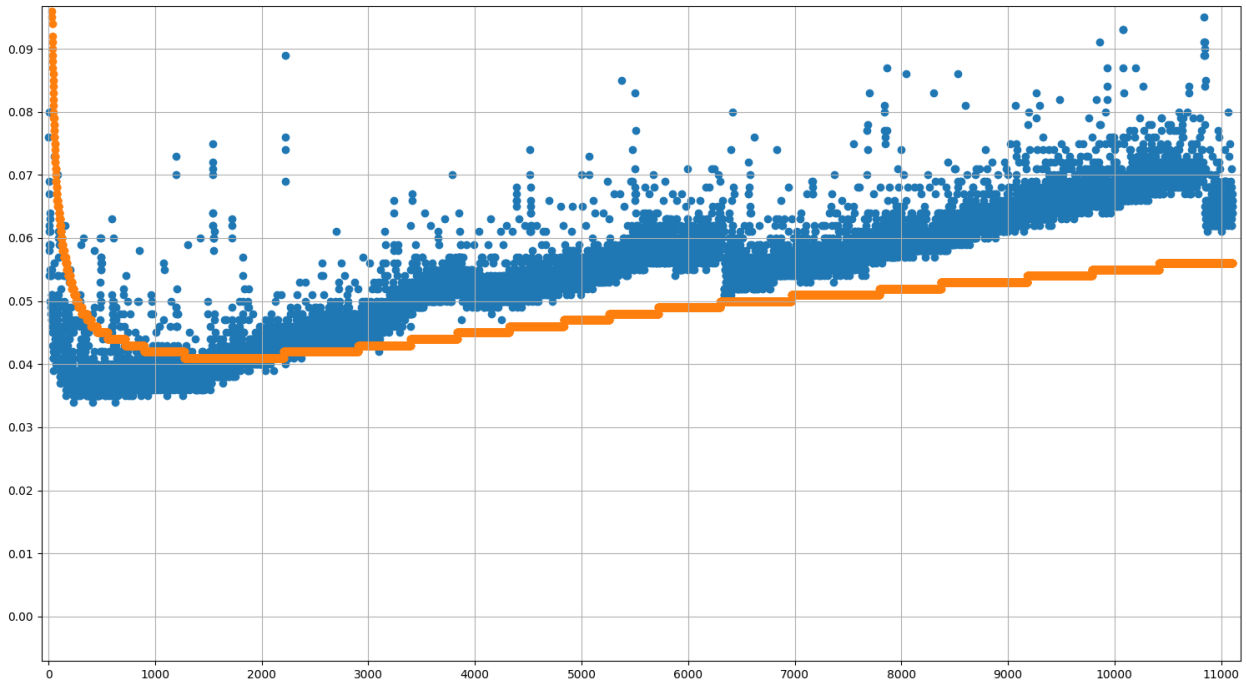
Судя по дампу объекты создаются и уничтожаются правильно, за исключением `FastDAOFactory`.

- *Как часто происходит сборка мусора?*

  За время эксперимента произошла 1144 раз. Средняя пауза 2 ms 545 us, максимальная 102 ms 584 us.

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

  Текущее время выросло с 0.04 с до 0.57 с за 11200 выполнений сценария 1. По сравнению с предыдущим этапом, общая производительность немного ниже.



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

  Чаще всего CPU обрабатывает те же методы, что и на предыдущем этапе.

Stack Trace	Sample Count	Percentage
com.moneytransfer.service.AccountService.getAllAccounts()	536	**7.55%**
com.moneytransfer.service.UserService.getAllUsers()	59	**0.83%**
com.moneytransfer.model.Account.hashCode()	10	**0.14%**

- **ВЫВОДЫ СТАДИИ 2**

  1. Изменения позволили снизить загрузку процессора ещё на 3-4%
  2. Потребление памяти изменилось незначительно
  3. Общая производительность ухудшилась на несколько процентов
  4. in memory СУБД обладает не худшими характеристиками по сравнению с самописным хранилищем на `HashMap`