ORACLE

Катехизис `java.lang.String`

Stay Awhile And Listen

Aleksey Shipilëv
aleksey.shipilev@oracle.com, @shipilev

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

Intro

Катехи́зис (из лат. *catechēsis* от др.-греч. κατηχισμός — поучение, наставление ← κατηχεῖν — внушать, отвечать = κάτω — вниз + ήχος — звук) — официальный вероисповедный документ какой-либо конфессии, огласительное наставление, книга, содержащая основные положения вероучения, часто изложенные в виде вопросов и ответов.

“Science replaces private prejudice with public, verifiable evidence.”

— Richard Dawkins

Intro: Disclaimers

Все тесты выполнены:

- ...профессиональными спортсменами: перепроверяйте[1] результаты перед тем, как их использовать
- ...на 1x2x4 i7-4790K (4.0 GHz, HSW): супер-быстрая машина, абсолютные числа значения не имеют
- ...на Linux x86_64, 3.13: не самом последнем ядре
- ...на 8u40 EA x86_64: на самой последней стабильной JDK
- ...на JMH[2]: ну, вы знаете...

[1] https://github.com/shipilev/article-string-catechism/

[2] http://openjdk.java.net/projects/code-tools/jmh/

Intro: Но зачем?!

- Человеки общаются через текст
- Машины обслуживают человека (пока), и поэтому тоже общаются как человеки: большая часть исходников – текст, большая часть данных – текст

- Мы прикидывали, в среднем 25% объектов – это `java.lang.String`
- В текстовых приложениях оптимизация `String`-ов даёт приросты

Понимание подходов и реализации – дорога к счастью!

Кишочки

Кишочки: java.lang.String навыверт

```
public final class String implements ... {
  private final char[] value;
  private int hash;
  ...
```

Строчки иммутабельны:

- Можно использовать без синхронизации, и ничего не сломается
- Можно ссылаться на один и тот же char[], скрытно от пользователя

Кишочки: java.lang.String навыверт

Строчки на 90% состоят из воды:

```
java.lang.String object internals:
 OFFSET   SIZE   TYPE DESCRIPTION
      0     12        (object header)
     12      4 char[] String.value
     16      4    int String.hash
     20      4        (alignment loss)
Instance size: 24 bytes
```

- 8..16 bytes: `String` header
- 4..4 bytes: `String` hashcode
- 12..16 bytes: `char[]` header
- 0..8 bytes: alignment losses

12..24 байт против `char[]`, 24..44 против `wchar_t*`

Кишочки: Катехизис

Q: Ну и что, надо вообще использовать `String`?
A: Да, если работаете с текстовыми данными!

Q: А что если у меня проблемы с памятью?
A: Есть способы извернуться, смотрите дальше.

Q: Ну а я могу сделать свой собственный `String` поверх `char[]`!
A: Конечно, можете! Смотрите дальше по поводу граблей.

Q: *Должен* ли я делать свою собственную реализацию `String`?
A: *(В ответ была тишина, и Инженер ушёл просветлённым)*

Иммутабельность

Иммутабельность: строки равнее других

15.18.1 String Concatenation Operator +

If only one operand expression is of type `String`, then string conversion (§5.1.11) is performed on the other operand to produce a string at run-time.

The result of string concatenation is a reference to a `String` object that is the concatenation of the two operand strings. The characters of the left-hand operand precede the characters of the right-hand operand in the newly created string.

The `String` object is newly created (§12.5) unless the expression is a compile-time constant expression (§15.28).

Иммутабельность: погрязли в циклах

```
@Benchmark
public String string() {
  String s = "Foo";
  for (int c = 0; c < 1000; c++) {
    s += "Bar";
  }
  return s;
}
```


Иммутабельность: погрязли в циклах

```
@Benchmark
public String string() {
  String s = "Foo";
  for (int c = 0; c < 1000; c++) {
    s += "Bar"; // newly created String here
  }
  return s;
}
```


Иммутабельность: погрязли в циклах

```
@Benchmark
public String stringBuilder() {
  StringBuilder sb = new StringBuilder();
  for (int c = 0; c < 1000; c++) {
    sb.append("Bar");
  }
  return sb.toString();
}
```


Иммутабельность: погрязли в циклах

Да ладно, ничего плохого не будет:

Benchmark	Throughput, ops/s	
string	3250	± 18
stringBuffer	125270	± 1005
stringBuilder	116173	± 423

Страшная боль: просадили 30x, добавляя каких-то вшивых тысячу строчек.
Компиляторы могут помочь, но не до конца (смотрим дальше).
Всё время сожрано GC и переливанием из пустого в порожнее.

Иммутабельность: катехизис

Q: Почему всё это так больно?
A: Иммутабельность **почти всегда** сопровождается накладными расходами.

Q: Но мне нравится иммутабельность, как облегчить мои страдания?
A: Использовать билдеры! Они умеют быстро конструировать иммутабельные объекты.

Q: Почему JDK/JVM за нас это не сделает?
A: Они порядочно делают! Но если вам нужна пиковая производительность во всех ваших случаях, то придётся поработать ручками. (No Free Lunch)

Q: Нужна ли мне пиковая производительность?
A: *(В ответ была тишина, и Инженер ушёл просветлённым)*

Concat

Concat: декомпилируем

```
@Benchmark
public String string_2() {
  return s1 + s2;
}
```

...скомпилируется в:

```
public String string_2();
  Code:
     0: new           #14   // java.lang.StringBuilder
     3: dup
     4: invokespecial #15   // StringBuilder.new()
     7: aload_0
     8: getfield      #3    // s1:String;
    11: invokevirtual #16   // StringBuilder.append(String);
    14: aload_0
    15: getfield      #5    // s2:String;
    18: invokevirtual #16   // StringBuilder.append(String);
    21: invokevirtual #17   // StringBuilder.toString();
    24: areturn
```


SB: декомпилируем

Не мудрено, что цепочки
`StringBuilder.append` хорошо оптимизируются:

```
@Benchmark
public String sb_6() {
  return new StringBuilder()
      .append(s1).append(s2).append(s3)
      .append(s4).append(s5).append(s6)
      .toString();
}

@Benchmark
public String string_6() {
  return s1 + s2 + s3 + s4 + s5 + s6;
}
```

Попробуем с `-XX:±OptimizeStringConcat`...

SB: оптимизации в StringBuilder – крутые

Benchmark	N	Score, ns/op -Opt		Score, ns/op +Opt		Impr
stringBuilder	1	14.0	± 0.1	8.7	± 0.1	+61%
stringBuilder	2	20.3	± 0.2	12.1	± 0.4	+68%
stringBuilder	3	27.0	± 0.2	14.8	± 0.1	+82%
stringBuilder	4	33.3	± 0.5	21.1	± 0.1	+58%
stringBuilder	5	38.6	± 0.2	25.4	± 0.1	+50%
stringBuilder	6	69.6	± 1.0	29.9	± 0.2	+133%
string	1	2.3	± 0.1	2.3	± 0.1	0%
string	2	20.4	± 0.2	11.8	± 0.1	+73%
string	3	27.1	± 0.3	14.9	± 0.1	+82%
string	4	33.0	± 0.4	21.1	± 0.1	+56%
string	5	38.0	± 0.3	25.3	± 0.1	+50%
string	6	70.1	± 0.7	29.9	± 0.3	+135%

SB: имплицитность vs. эксплицитность

Из-за таких фокусов люди удивляются, как работает этот бенчмарк:

```
private int x;

@Setup
void setup() { x = 1709; }

@Benchmark
String concat_Pre()       { return "" + x; }

@Benchmark
String concat_Post()      { return x + ""; }

@Benchmark
String integerToString() { return Integer.toString(x); }

@Benchmark
String stringValueOf()    { return String.valueOf(x); }
```


SB: имплицитность vs. эксплицитность, #2

Benchmark	Score, ns/op	
concat_Post	14.9	± 0.1
concat_Pre	15.0	± 0.1
integerToString	21.8	± 0.1
stringValueOf	21.9	± 0.3

Имплицитная конкатенация быстрее, чем **эксплицитная** конверсия?

- Оптимизации в `StringBuilder` делают `append(int)` быстрее!
- Будет медленее с `-XX:-OptimizeStringConcat`

SB: Side Effects

Немножко усложним:

```
private int x;

@Setup
void setup() { x = 1709; }

@Benchmark
String concat_just()           { return "" + x; }

@Benchmark
String concat_side()           { x--; return "" + (x++); }

@Benchmark
String integerToString_just() { return Integer.toString(x); }

@Benchmark
String integerToString_side() { x--; return Integer.toString(x++); }
```


SB: Side Effects, #2

Benchmark	Score, ns/op	
concat_just	14.8	± 0.1
integerToString_just	21.6	± 0.1
stringValueOf_just	21.6	± 0.1
concat_side	27.2	± 0.3
integerToString_side	21.6	± 0.1
stringValueOf_side	21.6	± 0.2

- Как только есть сайд-эффект в аргументах append(), оптимизация ломается[3]: если нужна будет деоптимизация, потребуется «откручивать» исполнение, но уже поздно
- Вытаскивание сайд-эффектов из append() помогает

[3] https://bugs.openjdk.java.net/browse/JDK-8043677

Ленивые логгеры: проблема

```
private int x;
private boolean enabled;

void log(String msg) {
  if (enabled) {
    System.out.println(msg);
  }
}

@Benchmark
void heap_string() {
  log("Wow, x is such " + x + "!");
}

@Benchmark
void heap_string_guarded() {
  if (enabled) {
    log("Wow, x is such " + x + "!");
  }
}
```

- Конкатенация случится *до* проверки `enabled`
- Теряем время на построение строк, которые нам не нужны
- Поэтому, многие вытаскивают проверки за конкатенацию

Ленивые логгеры: проблема

```
private int x;
private boolean enabled;

@Benchmark
void heap_lambda() {
  log(() -> "Wow, such " + x + "!");
}

@Benchmark
void noArg_lambda() {
  log(() -> "Such message, wow.");
}

@Benchmark
public void local_lambda() {
  int lx = x;
  log(() -> "Wow, such " + lx + "!");
}
```

- Куда лучше с лямбдами: отложенное исполнение без синтаксического ада
- (Есть небольшая разница между захватом локалов, полей, и вообще не захвата)

Java

Ленивые логгеры: нате

Method	Time, ns/op					
	heap		local		noArgs	
string	19.3	± 0.4	17.7	± 0.2	0.4	± 0.1
lambda	1.8	± 0.1	1.8	± 0.1	0.4	± 0.1
string_guarded	0.4	± 0.1	0.4	± 0.1	0.4	± 0.1

Лямбды крутые! Эксплицитная проверка всё равно побеждает, но не намного: лямбды с захватом контекста (всё ещё) требуют аллокации.

Concat: катехизис

Q: Надо ли мне беспокоиться о стоимости конкатенаций?
A: Да, во всех нетривиальных случаях. В тривиальных случаях помочь толком нельзя.

Q: Какие случаи считать нетривиальными?
A: Все, что включают в себя control flow, side effects, etc.

Q: То есть, со `StringBuilder`-ами вообще всё хорошо?
A: Они хорошо оптимизированы, но «и на старуху бывает проруха».

Q: Отвалите! Я профессионал, дайте мне lazy-val, call-by-name...
A: *(Тишина была ему ответом, и профессионал ушёл просветлённым)*

Хэшкоды

Нули: P(31) хэшкод

Спецификация на `String.hashCode` говорит, что используется полиномиальный хэшкод: P(31)

$$h(s) = \sum_{k=0}^{n-1} 31^{n-k-1} s_k$$

```
public int hashCode() {
   ...
   int h = 0;
   for (char v : value) {
     h = 31 * h + v;
   }
   hash = h;
}
```

Сложность: $\Omega(N)$ и $O(N)$.

Нули: попробуем...

```
String str1, str2;

@Setup
public void setup() {
  str1 = "сверхинструментом␣пренебрегшая"; // same length
  str2 = "пренебрегшая␣сверхинструментом"; // same length
}

@Benchmark
int test1() { return str1.hashCode(); }

@Benchmark
int test2() { return str2.hashCode(); }
```


Нули: попробуем...

```
String str1, str2;

@Setup
public void setup() {
  str1 = "сверхинструментом␣пренебрегшая"; // same length
  str2 = "пренебрегшая␣сверхинструментом"; // same length
}

@Benchmark
int test1() { return str1.hashCode(); } // 24.6 ± 0.1 ns/op

@Benchmark
int test2() { return str2.hashCode(); } // 2.0 ± 0.1 ns/op
```


Нули: настоящая реализация

```
public int hashCode() {
  int h = hash;
  if (h == 0) {
    for (char v : value) {
      h = 31 * h + v;
    }
    hash = h;
  }
  return h;
}
```

- Настоящая реализация кеширует хэшкоды: огромные приросты в большинстве сценариев, поэтому можно потратить 4 байта на объект
- Согласно *принципу Дирихле*, у некоторых строчек будет $hs(s) = 0$, выпьем за их светлую память
- Вменяемый инженерный tradeoff: перформансная аномалия с вероятностью 2^{-32}

Коллизии: Walking on a Sunshine

```
// carefully populated with unicorn dust:
HashMap<String, String> sunshine;

@Benchmark void keySet(Blackhole bh) {
  for (String key : sunshine.keySet()) {
    bh.consume(sunshine.get(key));
  }
}

@Benchmark void entrySet(Blackhole bh) {
  for (Map.Entry<String, String> e : sunshine.entrySet()) {
    bh.consume(e);
  }
}
```


Коллизии: включим JDK 7u0...

Benchmark	Size	Time, ns/op		ns/key
entrySet	1	14.1	± 0.1	14.1
entrySet	10	47.4	± 0.2	4.7
entrySet	100	294.1	± 0.9	2.9
entrySet	1000	5366.9	± 802.8	5.4
entrySet	10000	67394.4	± 456.5	6.7
keySet	1	18.4	± 0.5	18.4
keySet	10	279.8	± 6.7	27.8
keySet	100	22266.6	± 179.6	222.7
keySet	1000	2716486.4	± 10145.7	2716.5
keySet	10000	355309390.2	± 1214802.8	355309.4

Производительность keySet резко падает: $O(N^2)$

Коллизии: алгоритмические атаки

На полиномиальных хеш-функциях тривиально генерируются коллизии.
Это довольно просто следует из раскрытия:

$$h(s) = \sum_{k=0}^{n-1} 31^{n-k-1} s_k = [\sum_{k=0}^{n-3} 31^{n-k-1} s_k] + 31 s_{n-2} + s_{n-1}$$

Тогда, если строки a и b имеют общий префикс в $[0..n-3]$:

$$h(a) = h(b) \Leftrightarrow 31(a_{n-2} - b_{n-2}) = (a_{n-1} - b_{n-1})$$

...и это супер-просто. Положим, $a =$ "$...Aa$" and $b =$ "$...BB$".

Коллизии: мы-то тут при чём?

- Дядя Фёдор обслуживает супер-защищённый HTTP сервер, пропатчен под Heartbleed, Shellshock, и прочие именные эксплойты.
- Почтальон Печкин хихикает в усы и посылает HTTP запрос с такими заголовками:

```
"X-Achtung-AaAaAaAa: Сейчас я буду вашего мальчика измерять"
"X-Achtung-AaAaAaBB: Сейчас я буду вашего мальчика измерять"
"X-Achtung-AaAaBBAa: Сейчас я буду вашего мальчика измерять"
"X-Achtung-AaAaBBBB: Сейчас я буду вашего мальчика измерять"
```

- Web-сервер дяди Фёдора принимает запрос, сохраняет заголовки в `Map<String, String>`, и потом пытается их запроцессить. Бабах, denial of service.

Коллизии: включаем JDK 8

Benchmark	Size	Time, ns/op		ns/key
entrySet	1	11.6	± 0.1	11.7
entrySet	10	36.3	± 0.1	3.6
entrySet	100	278.1	± 0.7	2.8
entrySet	1000	3606.7	± 21.4	3.6
entrySet	10000	86459.5	± 626.4	8.6
keySet	1	15.1	± 0.1	15.0
keySet	10	253.2	± 0.6	2.5
keySet	100	10072.5	± 144.4	100.7
keySet	1000	158591.7	± 1202.4	158.6
keySet	10000	2355039.3	± 12087.3	235.3

keySet теперь «всего» $O(NlogN)$ – неплохо

Коллизии: другие грабли

`http://www.zlib.net/crc_v3.txt`

> *In particular, any CRC algorithm that initializes its register to zero will have a blind spot of zero when it starts up and will be unable to "count"a leading run of zero bytes. As a leading run of zero bytes is quite common in real messages, it is wise to initialize the algorithm register to a non-zero value.*

Та же самая история с `String.hashCode`.
Слава всем богам, строчки с нулевыми префиксами очень редки.

Хэшкоды: катехизис

Q: Надо ли мне волноваться насчёт `String.hashCode`?
A: Скорее всего нет, кроме случаев, когда вы голыми `Map`-ами подставляетесь под пользовательский ввод.

Q: Надо ли оборачивать `String` в адаптер со своим `hashCode`?
A: В некоторых редких случаях приходится.

Q: Почему бы просто не изменить вычисление `String.hashCode`?
A: Хэшкод для `String` специфицирован уже в туче мест, его нельзя сменить.

Q: Штука про нули `String.hashCode` меня беспокоит, сделаем `boolean`?
A: Не стоит: потенциально взрывает `String` на 8 байт.

Подстроки

Подстроки: JDK 8

```
java.lang.String object internals:
 OFFSET  SIZE   TYPE DESCRIPTION
      0    12        (object header)
     12     4 char[] String.value
     16     4    int String.hash
     20     4        (alignment loss)
Instance size: 24 bytes
```

Опытные разработчики спросят...

Подстроки: JDK 8

```
java.lang.String object internals:
 OFFSET  SIZE   TYPE DESCRIPTION
      0    12        (object header)
     12     4 char[] String.value
     16     4    int String.hash
     20     4        (alignment loss)
Instance size: 24 bytes
```

Опытные разработчики спросят... а где поля `offset` и `count`?

Подстроки: JDK $<$ 7u6

```
java.lang.String object internals:
 OFFSET  SIZE   TYPE DESCRIPTION
      0    12        (object header)
     12     4 char[] String.value
     16     4    int String.offset
     20     4    int String.count
     24     4    int String.hash
     28     4        (alignment loss)
Instance size: 32 bytes
```

Вот они! Остались в прошлом, JDK $<$ 7u6.

Подстроки: побенчмаркаемЪ

```
@Param({"0", "30", "60", "90", "120"})
int limit;

String str;

@Setup
public void setup() {
  str = "JPoint 2015: Public. Static. Void." +
        "JPoint 2015: Public. Static. Void." +
        "JPoint 2015: Public. Static. Void." +
        "JPoint 2015: Public. Static. Void.";
}

@Benchmark
String head() { return str.substring(limit); }

@Benchmark
String tail() { return str.substring(0, limit); }
```


Подстроки: JDK < 7u6: sharing

Limit	Time, ns/op			
	head		tail	
0	2.2	± 0.1	3.7	± 1.1
30	3.5	± 0.2	3.6	± 0.9
60	3.5	± 0.2	3.4	± 0.2
90	3.7	± 0.4	3.4	± 0.1
120	3.7	± 1.0	3.4	± 0.1

- `substring()` только инстанциирует `String`-и, шарит `char[]`
- Известно, что это приводит к утечкам памяти: представьте огромный XML и маленький substring

Подстроки: JDK 8: Copying

Limit	Time, ns/op			
	head		tail	
0	2.2	± 0.1	19.4	± 0.3
30	22.9	± 0.1	10.1	± 0.0
60	16.8	± 0.1	15.2	± 0.1
90	12.7	± 0.1	21.7	± 0.5
120	11.1	± 0.3	26.6	± 0.1

- `substring()` копирует нужну ему часть массива `char[]`
- Работает нормально для мелких строчек, не допускает утечек памяти

Подстроки: катехизис

Q: Этот ваш новый `substring` выглядит плохо, можно мне обратно?
A: Нет, нельзя.

Q: Но почему?!
A: Потому что реальные утечки памяти куда хуже потенциальных перформанс-проблем.

Q: Но что если мне нужен $O(1)$ `substring`?
A: Это значит, что ваш случай специальный, делайте свою собственную реализацию.

Q: Но моё приложение так надеялось на производительность `substring`!
A: *(показывает на javadoc к `String.substring`, и инженер уходит просветлённым)*

Intern

Intern: интернирование vs. дедупликация

Дедупликация:
Уменьшение количества объектов в одном классе эквивалентности

Интернирование (каноникализация):
Уменьшение количества объектов в одном классе эквивалентности до одного (канонического) объекта

- Как обычно, поддерживать *более строгий инвариант* стоит дороже
- Во многих случаях вы хотите *дедупликацию*, не *интернирование*

Intern: пользовательские интернеры

Его реально просто сделать руками:

```
public class CHMInterner<T> {
  private final Map<T, T> map;

  public CHMInterner() {
    map = new ConcurrentHashMap<>();
  }

  public T intern(T t) {
    T exist = map.putIfAbsent(t, t);
    return (exist == null) ? t : exist;
  }
}
```


Intern: пользовательские интернеры, #2

strings	Time, us/op					
	chm		hm		intern	
100	2.4	± 0.1	0.9	± 0.1	8.0	± 0.3
10000	242.9	± 0.944	133.8	± 0.8	891.8	± 13.6
1000000	47537.0	± 2123.8	35349.2	± 1188.8	315664.8	± 17821.4

(Временный) (Concurrent)HashMap на порядок быстрее!

Intern: а всё потому, что...

`String.intern()` – это маленькая дверца во внутренний VM-ный StringTable. Он постоянного размера, и почти всегда перегружен:

```
-XX:+PrintStringTableStatistics
StringTable statistics:
Number of buckets       :       60013 =     480104 bytes, avg   8.000
Number of entries       :     1002451 =   24058824 bytes, avg  24.000
Number of literals      :     1002451 =   64168512 bytes, avg  64.012
Total footprint         :             =   88707440 bytes
Average bucket size     :      16.704
Variance of bucket size :       9.731
Std. dev. of bucket size:       3.119
Maximum bucket size     :          27
```

Пользовательские `String.intern()` всё делают только хуже!

Intern: вероятностные дедупликаторы

Ослабление требования каноникализации приносит производительность:

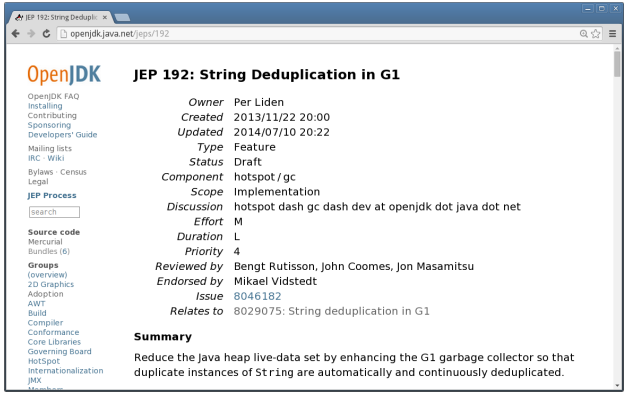
```
public class CHMDeduplicator<T> {
  private final int prob;
  private final Map<T, T> map;

  public CHMDeduplicator(double prob) {
    this.prob = (int) (Integer.MIN_VALUE + prob * (1L << 32));
    this.map = new ConcurrentHashMap<>();
  }

  public T dedup(T t) {
    if (ThreadLocalRandom.current().nextInt() > prob) {
      return t;
    }
    T exist = map.putIfAbsent(t, t);
    return (exist == null) ? t : exist;
  }
}
```


Intern: вероятностные дедупликаторы

	time, us/op					
Prob	chm		hm		intern	
0.0	3.2	± 0.1	3.3	± 0.1	3.3	± 0.1
0.1	6.9	± 0.1	7.3	± 0.7	13.1	± 0.1
0.2	10.4	± 0.4	9.7	± 0.7	22.4	± 0.1
0.3	13.4	± 0.2	12.1	± 0.2	31.9	± 0.3
0.4	16.4	± 0.1	14.2	± 0.1	40.3	± 0.3
0.5	19.1	± 0.1	15.9	± 0.1	49.3	± 0.8
0.6	21.7	± 1.1	16.7	± 0.2	56.6	± 0.6
0.7	22.4	± 0.2	16.0	± 0.1	63.3	± 1.1
0.8	23.7	± 0.5	15.4	± 0.1	70.7	± 2.5
0.9	25.7	± 0.9	14.0	± 0.1	76.4	± 0.7
1.0	26.1	± 0.1	11.5	± 0.1	118.5	± 30.1

Intern: GC Deduplication

Почему JVM за нас это не сделает?



`-XX:+UseG1GC -XX:+UseStringDeduplication`

Intern: GC Deduplication

```
public static void main(String... args) {
  List<String> strs = ...;

  String last = GraphLayout.parseInstance(strs).toFootprint();
  System.out.println("*** Original: " + last);

  for (int gc = 0; gc < 100; gc++) {
    String cur = GraphLayout.parseInstance(strs).toFootprint();

    if (!cur.equals(last)) {
      System.out.println("*** GC changed: " + cur);
      last = cur;
    }

    System.gc();
  }
}
```

Используем JOL[4], чтобы посмотреть на занимаемую память.

[4] http://openjdk.java.net/projects/code-tools/jol/

Intern: GC Deduplication

```
*** Original:
java.util.ArrayList instance footprint:
     COUNT       AVG       SUM   DESCRIPTION
     10000        47    472000   [C
         1     56232     56232   [Ljava.lang.Object;
     10000        24    240000   java.lang.String
         1        24        24   java.util.ArrayList
     20002             768256   (total)

*** GC changed:
java.util.ArrayList instance footprint:
     COUNT       AVG       SUM   DESCRIPTION
       100        47      4720   [C
         1     56232     56232   [Ljava.lang.Object;
     10000        24    240000   java.lang.String
         1        24        24   java.util.ArrayList
     10102             300976   (total)
```

Вуаля, `char[]` дедуплицированы!

Intern: GC Deduplication

```
*** GC changed:
java.util.ArrayList instance footprint:
     COUNT        AVG        SUM   DESCRIPTION
       100         47       4720   [C
         1      56232      56232   [Ljava.lang.Object;
     10000         24     240000   java.lang.String
         1         24         24   java.util.ArrayList
     10102             300976   (total)

*** Dedup:
java.util.ArrayList instance footprint:
     COUNT        AVG        SUM   DESCRIPTION
       100         47       4720   [C
         1      56232      56232   [Ljava.lang.Object;
       100         24       2400   java.lang.String
         1         24         24   java.util.ArrayList
       202             63376   (total)
```

Ручной дедупликатор может уменьшить число самих String-ов.

Intern: катехизис

Q: Но я столько читал про использование `String.intern` для оптимизации!
A: `http://en.wikipedia.org/wiki/Hanlon's_razor`

Q: Не страшно, я использую `String.intern` в этом незначительном месте.
A: Отлично, мы уже знаем, куда смотреть для оптимизации.

Q: Почему бы просто не улучшить `String.intern`?
A: Мы **улучшаем**! Но это не спасает от долбанутых сценариев использования.

Q: Могу ли я надеяться на GC для максимальной дедупликации?
A: Правила identity в Java запрещают нам клеить объекты, так что вам придётся делать это самим.

Equals

Equals: проверим базовые вещи

```
String bar10_0 = "BarBarBarA", bar10_1 = "BarBarBarA";
String bar10_2 = "BarBarBarB", bar10_3 = "ABarBarBar";
String bar11   = "BarBarBarAB";

@Benchmark
boolean sameChar()          { return bar10_0.equals(bar10_1); }

@Benchmark
boolean sameLen_diffEnd()   { return bar10_0.equals(bar10_2); }

@Benchmark
boolean sameLen_diffStart() { return bar10_0.equals(bar10_3); }

@Benchmark
boolean differentLen()      { return bar10_0.equals(bar11); }
```


Equals: базовые свойства

Benchmark	Score, ns/op
sameChar	1.0 ± 0.1
differentLen	1.3 ± 0.1
sameLen_diffEnd	4.6 ± 0.1
sameLen_diffStart	2.6 ± 0.1

- Строки, взятые из констант, проверяются на ==, и всё
- Строки с различной длиной не надо сравнивать
- Остальные строки проверяются с начала и до конца

Equals: реализация

```
public boolean equals(Object anObject) {
  if (this == anObject) {
    return true;
  }
  if (anObject instanceof String) {
    String anotherString = (String)anObject;
    int n = value.length;
    if (n == anotherString.value.length) {
      char v1[] = value;
      char v2[] = anotherString.value;
      int i = 0;
      while (n-- != 0) {
        if (v1[i] != v2[i])
          return false;
        i++;
      }
      return true;
    }
  }
  return false;
}
```

«I think this version is well-optimized, and you can gain nothing here...»
(somebody on StackOverflow)

Equals: интринзики

Benchmark	Score, ns/op			
	default		disabled[5]	
sameChar	1.0	± 0.1	1.1	± 0.1
differentLen	1.3	± 0.1	1.3	± 0.1
sameLen_diffEnd	4.6	± 0.1	9.7	± 0.1
sameLen_diffStart	2.6	± 0.1	3.0	± 0.1

- Настоящая реализация `equals()` подставляется компилятором
- Слепое переписывание Java-кода не даст приростов
- Как интринзик-версия побила «оптимальный» Java-овый код?

[5] `-XX:+UnlockDiagnosticVMOptions -XX:DisableIntrinsic=::_equals`

Equals: интринзики, #2

Интринзик-версия векторизована:

```
  5.23%     3.42%     0x00007f1b8c93de95: mov     (%rdi,%rcx,1),%ebx
 14.73%     4.01%     0x00007f1b8c93de98: cmp     (%rsi,%rcx,1),%ebx
                      0x00007f1b8c93de9b: jne     0x00007f1b8c93debb
 26.39%    27.41%     0x00007f1b8c93de9d: add     $0x4,%rcx
                      0x00007f1b8c93dea1: jne     0x00007f1b8c93de95
```

- Сравниваем кусками сразу по 4 байта
- Работает вне зависимости от штатных возможностей компилятора по векторизации
- VM сама выберет, использовать ли ей SSE, AVX, AVX2 и т.п.

Equals: катехизис

Q: У меня есть классная идея, как оптимизировать `String.equals`...
A: Если вы не готовы копаться в компиляторном коде, даже не начинайте.

Q: Зачем тогда вообще смущать народ Java-овой версией?
A: Интепретатор, C1, и прочие компиляторы используют её как fallback.

Q: Но я могу просто заинтернить строчки и сравнивать их на ==?
A: Ну да, только ещё интернирование будет стоить времени и памяти.

Q: Но интернирование настолько проще!
A: *(тишина была ему ответом, и Инженер ушёл просвещённым)*

Регулярки

Регулярки: разорвать на куски

```
String text = "Глокая␣куздра␣штеко␣будланула␣бокра␣и␣курдячит␣бокрёнка
String textDup = text.replaceAll("␣", "␣␣");
Pattern pattern = Pattern.compile("␣␣");

@Benchmark
String[] charSplit()         { return text.split("␣"); }

@Benchmark
String[] strSplit()          { return textDup.split("␣␣"); }

@Benchmark
String[] strSplit_pattern() { return pattern.split(textDup); }
```


Регулярки: разорвать на куски

Benchmark	Time, ns/op	
charSplit	191.6	± 1.8
strSplit	527.9	± 5.6
strSplit_pattern	416.2	± 4.1

- `charSplit`: срабатывает быстрый путь для однобайтовых символов
- `strSplit`: использует `Pattern`, не удивляйтесь «скорости»
- `strSplit_pattern`: переиспользует `Pattern`, спасает чуть-чуть

Регулярки: другие методы

Куча других методов String-а скрытно используют Pattern:

- `matches(String regex)`
- `replaceFirst(String regex, String replacement)`
- `replaceAll(String regex, String replacement)`
- `replace(CharSequence target, CharSequence replacement)`
- `split(String regex)`
- `split(String regex, int limit)`

Иногда стоит кэшировать скомпилированный Pattern!

Регулярки: backtracking

Ищем при помощи `Pattern.compile("(x+x+)+y")`:

Text size	Time, ns/op "xx...xxy"	"xx..xx"
4	94.5 ± 1.3	
6	96.8 ± 1.0	
8	102.7 ± 1.6	
10	106.5 ± 5.1	
12	106.7 ± 1.5	
14	111.9 ± 1.5	
16	115.6 ± 2.1	

Регулярки: backtracking

Ищем при помощи `Pattern.compile("(x+x+)+y")`:

Text size	Time, ns/op "xx...xxy"		"xx..xx"	
4	94.5	± 1.3	291.8	± 9.2
6	96.8	± 1.0	1049.5	± 7.2
8	102.7	± 1.6	4028.0	± 49.9
10	106.5	± 5.1	15900.0	± 263.3
12	106.7	± 1.5	61694.5	± 704.4
14	111.9	± 1.5	245397.2	± 1528.4
16	115.6	± 2.1	989130.3	± 11201.7

На неподходящем тексте регулярка катастрофически backtrack-ится.

Регулярки: катехизис

Q: Я никогда не использую регулярки. Мне должно быть пофиг?
A: Нет, не должно. Учитесь их использовать, пока не слишком поздно.

Q: Окей, какие основные направления улучшений в регулярках?
A: Упрощать и кэшировать `Pattern`-ы.

Q: Катастрофический backtracking выглядит теоретически, забить?
A: Нельзя забивать, если пользователь вводит тексты или сами регулярки.

Q: Stand back! I know Regular Expressions!
A: *(stands back, и Инженер впиливается в стену, обретая просветление)*

Гуляем

Гуляем: charAt vs toCharArray

```
@Benchmark
public int charAt() {
  int r = 0;
  for (int c = 0; c < text.length(); c++) {
    r += text.charAt(c);
  }
  return r;
}

@Benchmark
public int toCharArray() {
  int r = 0;
  char[] chars = text.toCharArray();
  for (int c = 0; c < text.length(); c++) {
    r += chars[c];
  }
  return r;
}
```


Гуляем: charAt vs toCharArray

Benchmark	Size	Time, ns/op	
charAt	1	2.1	± 0.1
charAt	10	4.8	± 0.1
charAt	100	51.6	± 0.1
charAt	1000	734.6	± 0.3
toCharArray	1	6.5	± 0.1
toCharArray	10	9.6	± 0.1
toCharArray	100	61.2	± 1.2
toCharArray	1000	1242.2	± 4.6

- `charAt` делает проверки границ, но они хорошо оптимизируются
- `toCharArray` платит за дополнительное копирование

Гуляем: charAt vs toCharArray (spoiled)

```
@Benchmark
public int charAt_spoil() {
  int r = 0;
  for (int c = 0; c < text.length(); c++) {
    spoiler(); // empty non-inlineable
    r += text.charAt(c);
  }
  return r;
}

@Benchmark
public int toCharArray_spoil() {
  int r = 0;
  char[] chars = text.toCharArray();
  for (char c : chars) {
    spoiler(); // empty non-inlineable
    r += c;
  }
  return r;
}
```


Гуляем: charAt vs toCharArray (spoiled)

Benchmark	size	Score, ns/op	
charAt_spoil	1	4.7	± 1.1
charAt_spoil	10	32.3	± 0.1
charAt_spoil	100	607.9	± 0.2
charAt_spoil	1000	10247.5	± 1552.4
toCharArray_spoil	1	8.9	± 0.1
toCharArray_spoil	10	28.5	± 0.1
toCharArray_spoil	100	435.4	± 3.3
toCharArray_spoil	1000	6559.9	± 22.7

- Когда VM не может уследить за `text`, ломается девирт и оптимизация проверки границ
- С локальным массивом всё отлично

Гуляем: катехизис

Q: Так надо делать `toCharArray` или нет?
A: Если вам не нужен особенный перформанс, то выбор – вопрос стиля.

Q: Мне важен перформанс, надо делаеть `toCharArray`?
A: Да, в нетривиальных случаях.

Q: Что является нетривиальным случаем?
A: Нелокальный control flow, volatile-чтения, и прочие делающие оптимизаторную жизнь хуже вещи.

Q: Ну это фигня какая-то. Почему нет нормального общего способа?
A: *(Тишина была ему ответом, и Инженер ушёл просветлённым)*

Поиск

Поиск: ...символов

Ищем в "abcdefghiklmnopqrstuvxyz":

image	Time, ns/op	
	indexOf	lastIndexOf
a	1.3 ± 0.1	8.5 ± 0.1
m	4.8 ± 0.1	5.7 ± 0.1
z	7.3 ± 0.1	1.6 ± 0.1

- Что `indexOf`, что `lastIndexOf` работают за $O(n)$
- Каждый из них быстр, смотря с какого конца начинать

Поиск: интринзики

Benchmark	Image	Score, ns/op +Opt	Score, ns/op -Opt[6]
indexOf	abc	5.0 ± 0.1	4.9 ± 0.1
indexOf	mno	7.0 ± 0.1	9.8 ± 0.1
indexOf	xyz	11.5 ± 0.1	12.8 ± 0.1
lastIndexOf	abc	13.9 ± 0.1	13.9 ± 0.1
lastIndexOf	mno	10.5 ± 0.1	10.5 ± 0.1
lastIndexOf	xyz	5.3 ± 0.1	5.3 ± 0.1

- Настоящая реализация `indexOf` интринсифицирована
- Использует SSE/AVX для поиска совпадений

[6] `-XX:+UnlockDiagnosticVMOptions -XX:DisableIntrinsic=::_indexOf`

Поиск: поиск в геноме

Поиск последовательности кодонов в человеческой хромосоме «Y»:

Benchmark	Time, ms/op
indexOf	48.2 ± 0.4

- `str.indexOf(im)`: наивный поиск

Поиск: поиск в геноме

Поиск последовательности кодонов в человеческой хромосоме «Y»:

Benchmark	Time, ms/op	
indexOf	48.2	± 0.4
wikipediaBM	16.7	± 0.4

- `str.indexOf(im)`: наивный поиск
- `wikipediaBM`: копипаст из википедийной статьи про Бойера-Мура[7]

[7] http://en.wikipedia.org/wiki/Boyer-Moore_string_search_algorithm

Поиск: поиск в геноме

Поиск последовательности кодонов в человеческой хромосоме «Y»:

Benchmark	Time, ms/op	
indexOf	48.2	± 0.4
wikipediaBM	16.7	± 0.4
matcherFind	21.2	± 0.4

- `str.indexOf(im)`: наивный поиск
- `wikipediaBM`: копипаст из википедийной статьи про Бойера-Мура[7]
- `pattern(im).matcher(str).find()` тоже использует BM

[7] `http://en.wikipedia.org/wiki/Boyer-Moore_string_search_algorithm`

Поиск: катехизис

Q: Ну и почему в JDK нет оптимального способа поиска подстрок?
A: «Оптимального» для всех не существует.

Q: Зачем вообще держать тривиальный `indexOf`?
A: Поиск мелких подстрок работает лучше с тривиальным поиском. И подставляется под векторные инструкции.[8]

Q: Мы забили на Java в <insert domain here> из-за поиска подстрок.
A: *(показывает в сторону третьесторонних библиотек, и Инженер уходит просвещённым)*

[8] `https://twitter.com/shipilev/status/560065173407662080`

Выводы

Выводы: ...

- `String` хорошо оптимизирован:
  - Понимание, что и как оптимизируется, помогает в низкоуровневом тюнинге
  - Понимание, как работает, JDK/VM приближает вас к Правде^{TM}
- Перформансные советы быстро протухают:
  - Регулярно переучивайтесь
  - Не верьте фольклору

