Повесть о том,
как один инженер HTTP/2 Client разгонял

Sergey Kuksenko
sergey.kuksenko@oracle.com, @kuksenk0

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.

Начало

Sergey Kuksenko

October 25, 2016 · Twitter ·

А вообще коммюнити будет интересен доклад в стиле live story: вот мы взяли проект и так так и сяк его заоптимизяли?

Начало

Sergey Kuksenko

October 25, 2016 · Twitter ·

А вообще коммюнити будет интересен доклад в стиле live story: вот мы взяли проект и так так и сяк его заоптимизяли?

Aleksey Shipilev Доклад про оптимизацию проекта X.
Ожидание: тридцать трюков, которые можно тут же применить во всех проектах.
Реальность: пять фейспалмяще-очевидных переделок, и ещё два мегафейспалмящих хака.

Like · Reply · October 26, 2016 at 7:17am · Edited

HTTP/2

(a.k.a. RFC 7540)

HTTP/2 vs HTTP/1.1

Что нового (и важно):

- Бинарный формат (Frames)
- Мультиплексирование множества запросов в **единственном** TCP соединении
  - <Request>⇒<Stream>⇒<Frame>...<Frame>
- Сжатие заголовков (HPACK a.k.a. RFC 7541)

HTTP/2 vs HTTP/1.1

Что нового (и не важно):

- Приоритизация запросов
- Server push
- Ping
- ...

HTTP/2 vs HTTP/1.1

Что сохранилось:

- Значение HTTP методов (GET, POST, ...)
- Значение полей HTTP заголовка
- Коды статуса
- Структура: запрос $\rightarrow$ (ответ)* $\rightarrow$ финальный ответ

HTTP API

(a.k.a. JEP-110)

HTTP API a.k.a. HttpClient

- Появится в JDK 9, но не будет частью Java SE
  - `module:  jdk.incubator.httpclient`
  - `package: jdk.incubator.http`

HTTP API a.k.a. HttpClient

- Появится в JDK 9, но не будет частью Java SE
  - `module: jdk.incubator.httpclient`
  - `package: jdk.incubator.http`
- Incubator Modules a.k.a. JEP-11
  - песочница для "поиграться"
  - *«The incubation lifetime of an API is limited: It is expected that the API will either be standardized or otherwise made final in the next release, or else removed»*

HTTP API a.k.a. HttpClient

Основные классы:

- `HttpClient(Builder)`
- `HttpRequest(Builder)`
- `HttpResponse`

Примеры

```
HttpRequest getRequest = HttpRequest
    .newBuilder(URI.create("https://jpoint.ru/"))
    .header("X-header", "value")
    .GET()
    .build();
```

```
HttpRequest postRequest = HttpRequest
    .newBuilder(URI.create("https://jpoint.ru/"))
    .POST(fromFile(Paths.get("/abstract.txt")))
    .build();
```

Примеры

```
HttpClient client = HttpClient.newHttpClient();

HttpRequest request = ...;

HttpResponse<String> response =
        // synchronous/blocking
        client.send(request, BodyHandler.asString());

if (response.statusCode() == 200) {
    String body = response.body();
    ...
}
...
```

Примеры

```
HttpClient client = HttpClient.newHttpClient();

HttpRequest request = ...;

CompletableFuture<HttpResponse<String>> responseFuture =
        // asynchronous
        client.sendAsync(request, BodyHandler.asString());
...
```

Примеры

```
HttpClient client = HttpClient.newBuilder()
        .authenticator(someAuthenticator)
        .sslContext(someSSLContext)
        .sslParameters(someSSLParameters)
        .proxy(someProxySelector)
        .executor(someExecutorService)
        .followRedirects(HttpClient.Redirect.ALWAYS)
        .cookieManager(someCookieManager)
        .version(HttpClient.Version.HTTP_2)
        .build();
```

Бенчмаркаем

- Сервер:
  - локальный сервер (localhost)
  - Jetty Server 9.3.7.v20160115
- Клиент:
  - GET, POST запросы произвольного размера
  - Plain & SSL соединения
  - измеряем «throughput» a.k.a. («average latency»)$^{-1}$
  - «конкурент» - Jetty Client 9.3.7.v20160115

Переделка №1

~~Первая кровь~~ Первый запуск

GET 1 byte:

Linux, Intel Core i5-5300U, [1x2x2], 2.30GHz)	
HttpClient:	**25** ops/sec

~~Первая кровь~~ Первый запуск

GET 1 byte:

Linux, Intel Core i5-5300U, [1x2x2], 2.30GHz)	
HttpClient:	**25** ops/sec

~~Первая кровь~~ Первый запуск

GET 1 byte:

Linux, Intel Core i5-5300U, [1x2x2], 2.30GHz)	
HttpClient:	**25** ops/sec
JettyClient:	11688 ops/sec

~~Первая кровь~~ Первый запуск

GET 1 byte:

Linux, Intel Core i5-5300U, [1x2x2], 2.30GHz)	
HttpClient:	**25** ops/sec
JettyClient:	11688 ops/sec

HttpClient:

Linux	Intel Xeon E5-2690, [2x8x2], 2.90GHz)	25 ops/sec
Win64	Intel Xeon E5-2690, [2x8x2], 2.90GHz)	6091 ops/sec
Solaris	Intel Xeon E5-2690, [2x8x2], 2.90GHz)	6551 ops/sec
MacOS	Intel i7-3615, [1x4x2], 2.30GHz)	490 ops/sec

Проблема тут

```
SocketChannel chan;
...
try {
        chan = SocketChannel.open();
        int bufsize = client.getReceiveBufferSize();
        chan.setOption(StandardSocketOptions.SO_RCVBUF, bufsize);

    } catch (IOException e) {
        throw new InternalError(e);
    }
```

Проблема тут

```
SocketChannel chan;
...
try {
        chan = SocketChannel.open();
        int bufsize = client.getReceiveBufferSize();
        chan.setOption(StandardSocketOptions.SO_RCVBUF, bufsize);
        chan.setOption(StandardSocketOptions.TCP_NODELAY, true);
    } catch (IOException e) {
        throw new InternalError(e);
    }
```

Почему? (*привет из 1980-х*)

"Nagle's Algorithm" vs "Delayed ACK"

Stuart Cheshire

"TCP Performance problems caused by interaction between Nagle's Algorithm and Delayed ACK"

http://www.stuartcheshire.org/papers/NagleDelayedAck/

А что там с производительностью?

$$25 \ \frac{ops}{sec} \Rightarrow 9600 \ \frac{ops}{sec}$$

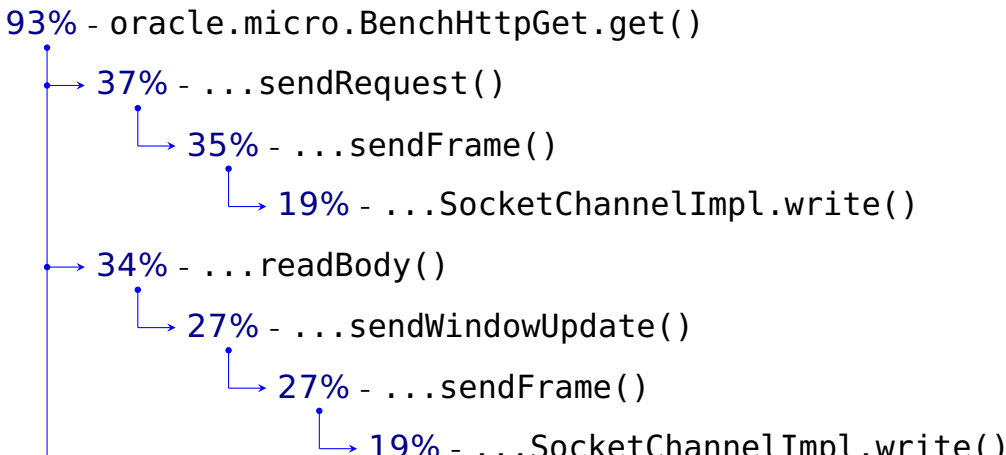
Мораль

Необходимо знать (и не забывать) список общеизвестных косяков для вашей предметной области.

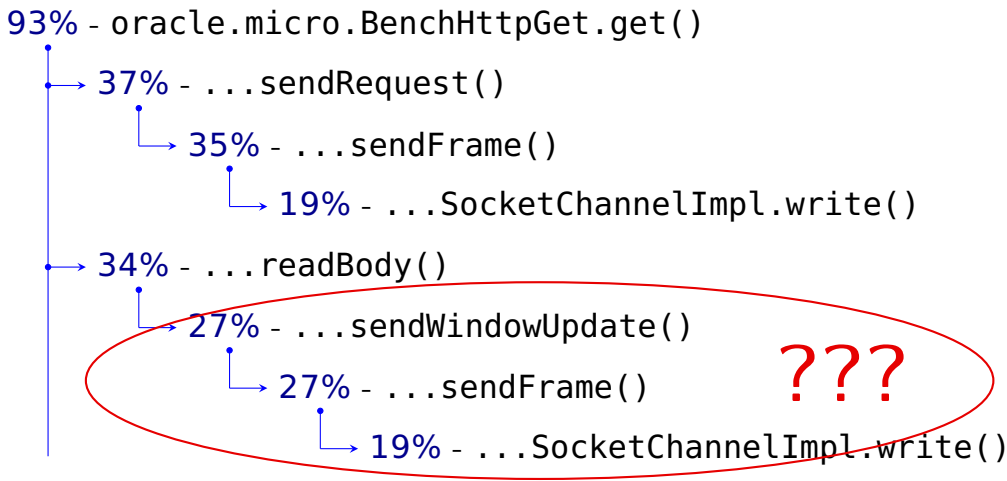
Переделка №2

Первое профилирование

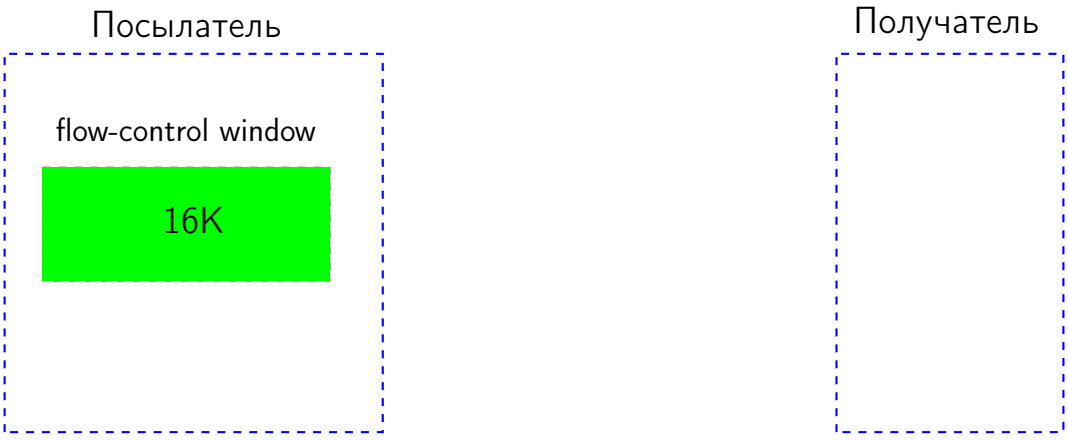
Первое профилирование



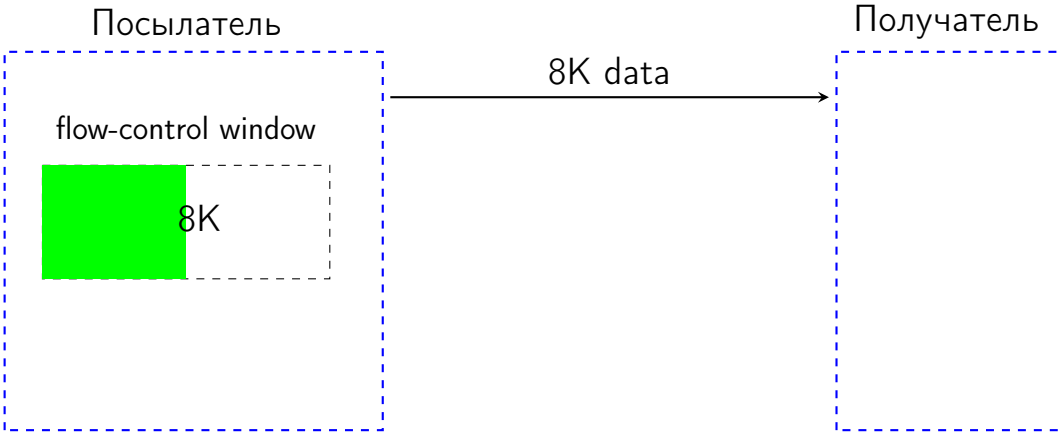
Первое профилирование



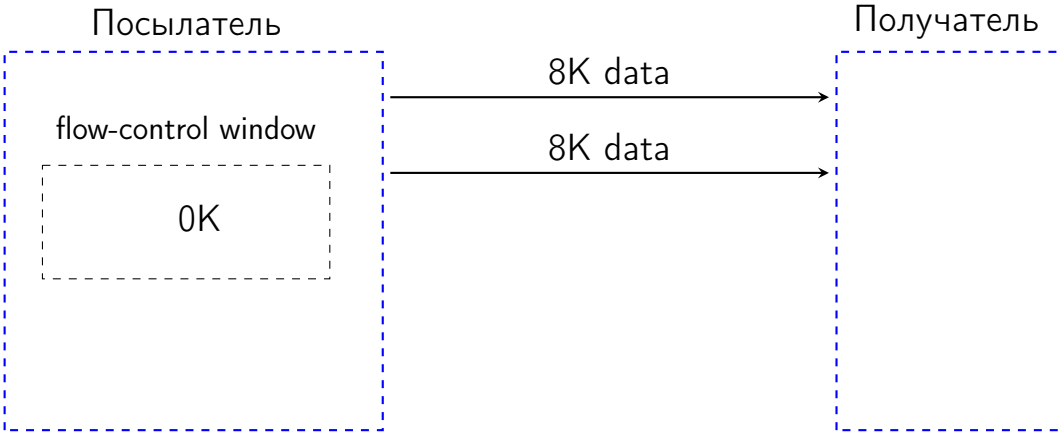
HTTP/2 Flow Control



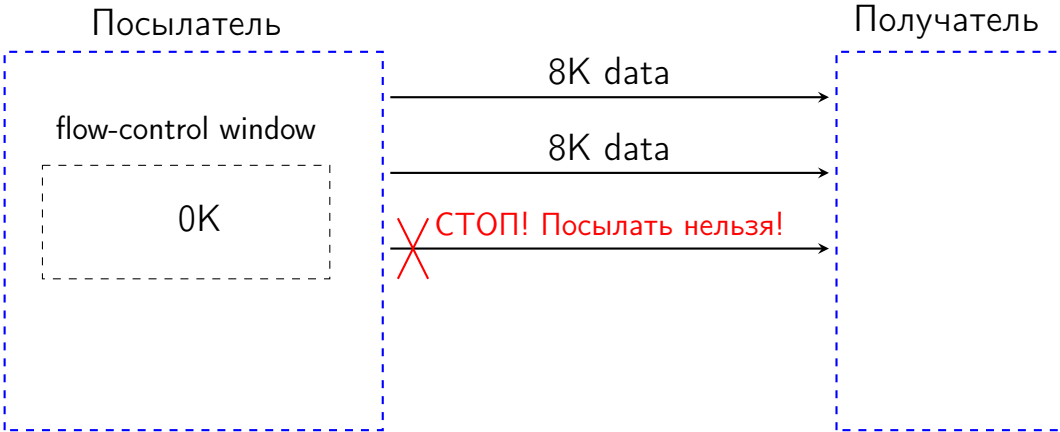
HTTP/2 Flow Control



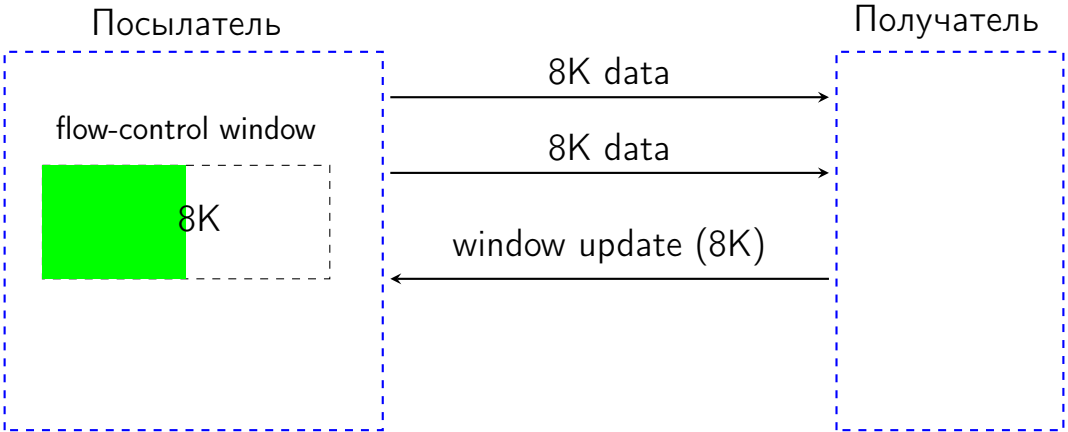
HTTP/2 Flow Control



HTTP/2 Flow Control



HTTP/2 Flow Control



Было (обработка входящих **DataFrame'ов**)

```
// process incoming data frames
...
DataFrame dataFrame;
do {
    DataFrame dataFrame = inputQueue.take();
    ...
    int len = dataFrame.getDataLength();
    sendWindowUpdate(0, len);        // update connection window
    sendWindowUpdate(streamid, len); // update stream window
} while (!dataFrame.getFlag(END_STREAM));
...
```

Стало (обработка входящих **DataFrame'ов**)

```
// process incoming data frames
...
DataFrame dataFrame;
do {
    DataFrame dataFrame = inputQueue.take();
    ...
    int len = dataFrame.getDataLength();
    connectionWindowUpdater.update(len);
    if (dataFrame.getFlag(END_STREAM)) {
        break;
    }
    streamWindowUpdater.update(len);
} while (true);
...
```

Стало (`WindowUpdater`)

```
final AtomicInteger received;
final int threshold;
...
void update(int delta) {
    if (received.addAndGet(delta) > threshold) {
        synchronized (this) {
            int tosend = received.get();
            if( tosend > threshold) {
                received.getAndAdd(-tosend);
                sendWindowUpdate(tosend);
            }
        }
    }
}
```

А что там с производительностью?

$$9600 \ \frac{ops}{sec} \Rightarrow 11850 \ \frac{ops}{sec}$$

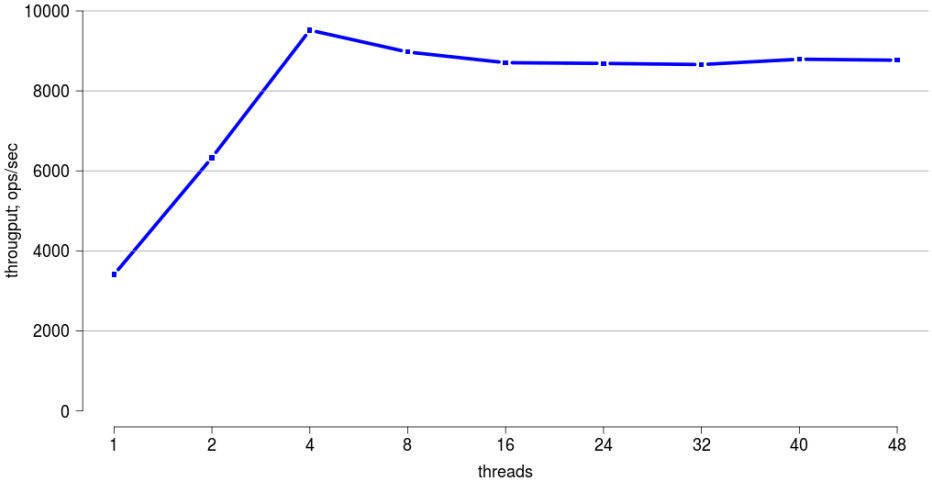
$$+23\%$$

Мораль

Аккуратное чтение спецификации и логика - ваши друзья.

Переделка №3

Скалируемость (Intel Xeon E5-2697, [2x12x2], 2.7GHz)



Ищем виновника

```
#!/bin/bash

(java  -jar benchmarks.jar BenchHttpGet.get -t 4 -f 0 &> log.log) &
JPID=$!
sleep 5
while kill -3 $JPID;
do
:
done
```

Посмотрим состояния потоков

```
$ grep java.lang.Thread.State log.log | sort | uniq -c | sort -g

     1    java.lang.Thread.State: NEW
   628    java.lang.Thread.State: TIMED_WAITING (parking)
 22990    java.lang.Thread.State: TIMED_WAITING (on object monitor)
 24058    java.lang.Thread.State: TIMED_WAITING (sleeping)
 30262    java.lang.Thread.State: BLOCKED (on object monitor)
 37725    java.lang.Thread.State: WAITING (parking)
 46364    java.lang.Thread.State: WAITING (on object monitor)
205420    java.lang.Thread.State: RUNNABLE
```

Кто блокирует?

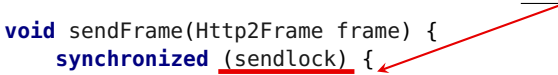
```
$ grep -A 1 BLOCKED log.log | sort | uniq -c | sort -g

      ...
  276  at sun.nio.ch.SocketChannelImpl.readerCleanup(SocketChannelImpl.java:281)
  328  at sun.nio.ch.SocketChannelImpl.writerCleanup(SocketChannelImpl.java:289)
  580  at sun.nio.ch.SocketChannelImpl.ensureWriteOpen(SocketChannelImpl.java:270)
  591  at sun.nio.ch.SocketChannelImpl.ensureReadOpen(SocketChannelImpl.java:257)
27915  at java.net.http.Http2Connection.sendFrame(Http2Connection.java:829)

30262  java.lang.Thread.State: BLOCKED (on object monitor)
```

Изучаем дальше

```
void sendFrame(Http2Frame frame) {
    synchronized (sendlock) {
        try {
            if (frame instanceof OutgoingHeaders) {
                OutgoingHeaders<Stream> oh = (OutgoingHeaders<Stream>) frame;
                Stream stream = registerNewStream(oh);
                List<Http2Frame> frames = encodeHeaders(oh, stream);
                writeBuffers(encodeFrames(frames));
            } else {
                writeBuffers(encodeFrame(frame));
            }
        } catch (IOException e) {
            ...
        }
    }
}
```

Изучаем дальше

Глобальный монитор



```
void sendFrame(Http2Frame frame) {
    synchronized (sendlock) {
        try {
            if (frame instanceof OutgoingHeaders) {
                OutgoingHeaders<Stream> oh = (OutgoingHeaders<Stream>) frame;
                Stream stream = registerNewStream(oh);
                List<Http2Frame> frames = encodeHeaders(oh, stream);
                writeBuffers(encodeFrames(frames));
            } else {
                writeBuffers(encodeFrame(frame));
            }
        } catch (IOException e) {
            ...
        }
    }
}
```

Изучаем дальше

Начало запроса (отправка хедера)

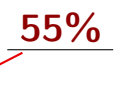
```
void sendFrame(Http2Frame frame) {
    synchronized (sendlock) {
        try {
            if (frame instanceof OutgoingHeaders) {
                OutgoingHeaders<Stream> oh = (OutgoingHeaders<Stream>) frame;
                Stream stream = registerNewStream(oh);
                List<Http2Frame> frames = encodeHeaders(oh, stream);
                writeBuffers(encodeFrames(frames));
            } else {
                writeBuffers(encodeFrame(frame));
            }
        } catch (IOException e) {
            ...
        }
    }
}
```

Изучаем дальше

Отправка всего остального

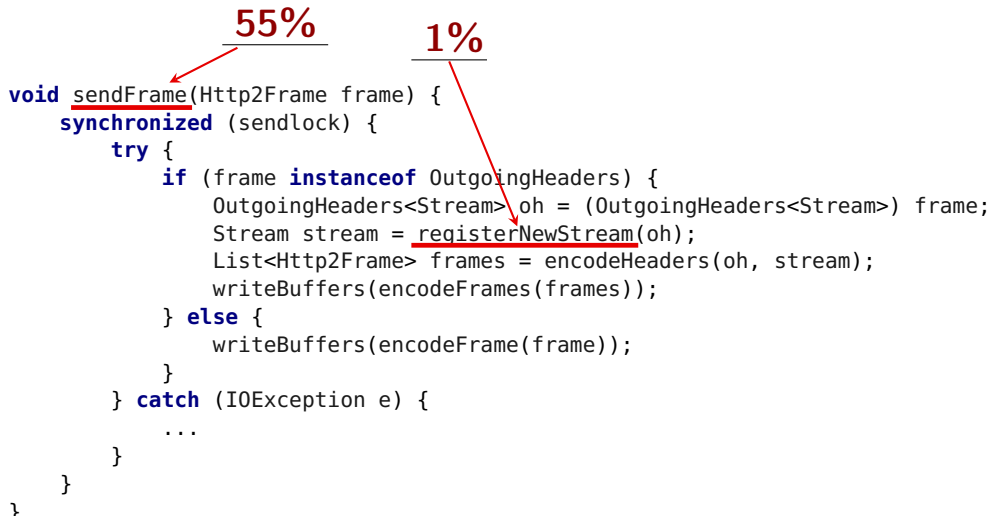
```
void sendFrame(Http2Frame frame) {
    synchronized (sendlock) {
        try {
            if (frame instanceof OutgoingHeaders) {
                OutgoingHeaders<Stream> oh = (OutgoingHeaders<Stream>) frame;
                Stream stream = registerNewStream(oh);
                List<Http2Frame> frames = encodeHeaders(oh, stream);
                writeBuffers(encodeFrames(frames));
            } else {
                writeBuffers(encodeFrame(frame));
            }
        } catch (IOException e) {
            ...
        }
    }
}
```

Изучаем дальше

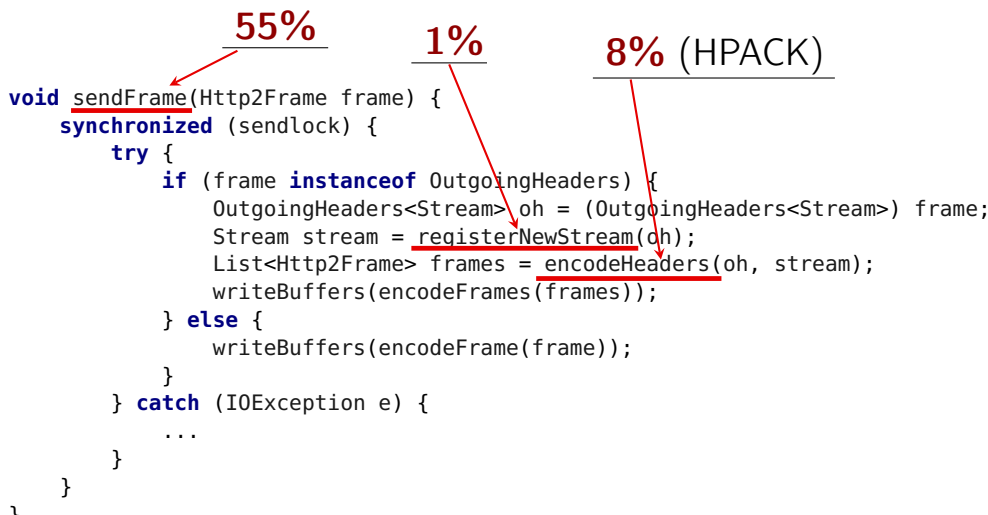



```
void sendFrame(Http2Frame frame) {
    synchronized (sendlock) {
        try {
            if (frame instanceof OutgoingHeaders) {
                OutgoingHeaders<Stream> oh = (OutgoingHeaders<Stream>) frame;
                Stream stream = registerNewStream(oh);
                List<Http2Frame> frames = encodeHeaders(oh, stream);
                writeBuffers(encodeFrames(frames));
            } else {
                writeBuffers(encodeFrame(frame));
            }
        } catch (IOException e) {
            ...
        }
    }
}
```

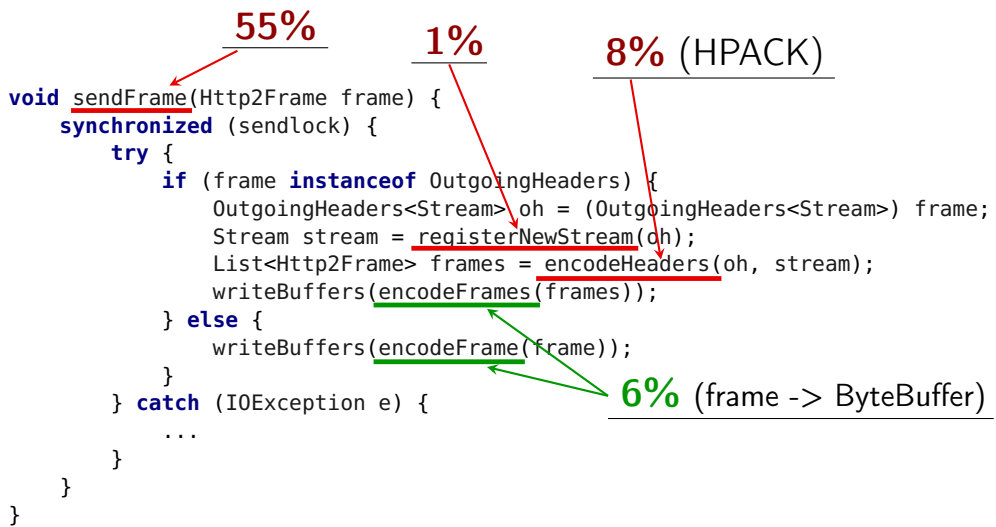
Изучаем дальше



Изучаем дальше

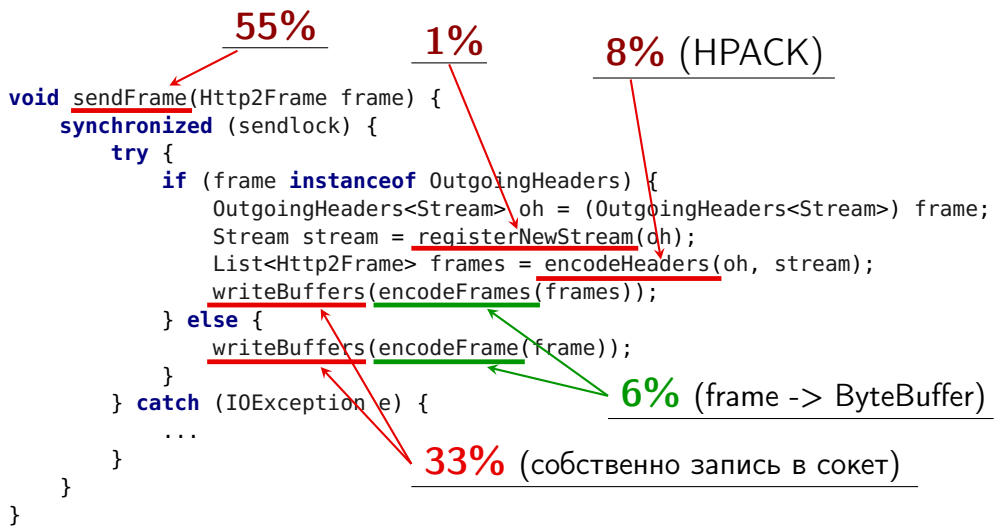


Изучаем дальше




```
void sendFrame(Http2Frame frame) {
    synchronized (sendlock) {
        try {
            if (frame instanceof OutgoingHeaders) {
                OutgoingHeaders<Stream> oh = (OutgoingHeaders<Stream>) frame;
                Stream stream = registerNewStream(oh);
                List<Http2Frame> frames = encodeHeaders(oh, stream);
                writeBuffers(encodeFrames(frames));
            } else {
                writeBuffers(encodeFrame(frame));
            }
        } catch (IOException e) {
            ...
        }
    }
}
```

Изучаем дальше



Делаем раз

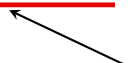
- делаем `Queue<ByteBuffer>`
  - `sendFrame` складывает в очередь (под глобальной блокировкой)
  - новый, выделенный поток, читает из очереди и отсылает
- плюсы:
  - размер критической секции стал меньше на 60%
- минусы:
  - некоторые фреймы по спеке можно отправить раньше других, но нельзя по реализации
  - отсылающий поток часто засыпает и долго просыпается

Делаем два

- делаем `Deque<ByteBuffer[]>`
  - `ByteBuffer[]` - атомарная последовательность буферов
  - `WindowUpdateFrame` - без блокировки, в начало очереди
  - `DataFrame` - без блокировки, в конец очереди
- плюсы:
  - меньше блокировок
  - ранняя отправка `WindowUpdate` позволяет серверу раньше отсылать данные
- минусы:
  - отсылающий поток часто засыпает и долго просыпается

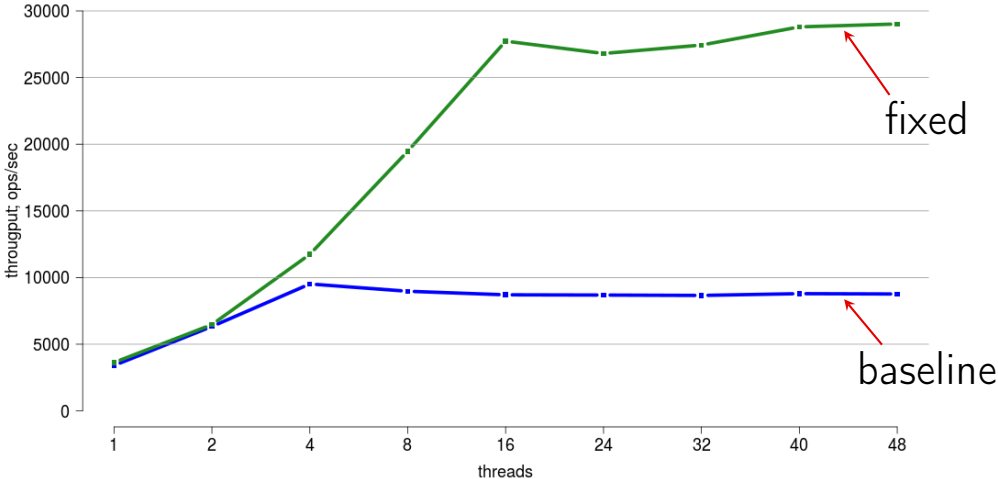
Делаем три

```
...
synchronized (sendlock) {
    if (frame instanceof OutgoingHeaders) {
        OutgoingHeaders<Stream> oh = (OutgoingHeaders<Stream>) frame;
        Stream stream = registerNewStream(oh);
        asyncOutputQueue.put(encodeHeaders(oh, stream));
    } else {
        asyncOutputQueue.put(encodeFrame(frame));
    }
}
asyncOutputQueue.flush();
...
```



- потоки соревнуются за право писать
- победитель пишет
- проигравшие работают дальше

А что там со скалируемостью?



Мораль

Блокировки - зло!

Переделка №4

Вопрос: Нужно ли делать ByteBuffer пул?

Вопрос: Нужно ли делать ByteBuffer пул?

Имеем:

- Архитектура HttpClient'а заточена на 100%-е использование пула буферов *(что даже пролезло в public API)*

Вопрос: Нужно ли делать ByteBuffer пул?

Имеем:

- Архитектура HttpClient'а заточена на 100%-е использование пула буферов *(что даже пролезло в public API)*

Имеем (и плачем):

- Только 20% буферов возвращается в пул
- `ByteBufferPool.getBuffer()` занимает **12%** времени

Что говорит народ?

- **Пул не нужен!**
  e.g. Dr. Cliff Click, Brian Goetz, Sergey Kuksenko, Aleksey Shipilëv, ...

- **Пул нужен!**
  e.g. Netty*, ...

*blog.twitter.com/2013/netty-4-at-twitter-reduced-gc-overhead

Подвопрос: `DirectByteBuffer` или `HeapByteBuffer`?

Подвопрос: `DirectByteBuffer` или `HeapByteBuffer`?

`DirectByteBuffer` лучше для I/O

- `sun.nio.*` копирует `HeapByteBuffer` в `DirectByteBuffer`

`HeapByteBuffer` лучше для SSL

- `SSLEngine` работает напрямую с `byte[]` в случае `HeapByteBuffer`

Проверяем: `DirectByteBuffer` vs `HeapByteBuffer`

`PlainConnection`

- `HeapByteBuffer` «быстрее» на **0%-1%**

`SSLConnection`

- `HeapByteBuffer` быстрее на **2%-3%**

`HeapByteBuffer` - наш выбор!

Делаем раз: «все в пул»

- не теряем ByteBuffer'а
- оптимизируем сам пул (на базе `ConcurrentLinkedQueue`)
- разделяем пулы (по размеру буфера):
  - SSL пакеты (`SSLSession.getPacketBufferSize()`)
  - кодирование заголовков (`MAX_FRAME_SIZE`)
  - все остальное

Делаем раз: «все в пул»

- плюсы:
  - меньше «allocation pressure»
- минусы:
  - сложный код
  - плохая «локальность данных»
  - затраты на пул (а еще мешает скалируемости)
  - частое копирование данных
    - Практическая невозможность использования `ByteBuffer.slice()` и `ByteBuffer.wrap()`

Делаем два: «никаких пулов»

- выпиливаем пулы
- активно используем `ByteBuffer.slice()/wrap()`

Делаем два: «никаких пулов»

- плюсы:
  - простой код
  - нет пулов в «public API»
  - хорошая «локальность данных»
  - значительное сокращение затрат на копирование
  - нет затрат на пул
- минусы:
  - выше «allocation pressure»
  - неэффективное использование памяти*

*когда требуемый размер буфера неизвестен

Делаем три: «смешанный вариант»

- Исходящие:
  - пользовательские данные - `wrap()/slice()` (GC соберет)
  - кодирование HTTP заголовков - свой пул
  - все остальное - буфера требуемого размера (GC соберет)

Делаем три: «смешанный вариант»

- Входящие:
  - чтение из сокета - буфер из пула
  - пришли данные (`DataFrame`) - `slice()` (GC соберет)
  - все остальное - возвращаем в пул

Делаем три: «смешанный вариант»

- SSL:
  - зашифрованные буфера - свой пул

Делаем три: «смешанный вариант»

- плюсы:
  - средняя сложность кода
  - нет пулов в «public API»
  - хорошая «локальность данных»
  - нет затрат на копирование
  - приемлимые затраты на пул
  - приемлимое использование памяти
- минусы:
  - выше «allocation pressure»

А что там с потреблением памяти?

		Plain	SSL
POST	«все в пул»	3% — 5%	4% — 10%
	«никаких пулов»	10% — 40%	12% — 40%
	«смешанный вариант»	3% — 11%	4% — 16%
GET	«все в пул»	4% — 20%	5% — 30%
	«никаких пулов»	5% — 60%	10% — 60%
	«смешанный вариант»	4% — 40%	5% — 40%

allocation rate(bytes/op); baseline = 100%

А это вообще играет роль?

	Plain	SSL
«baseline»	1.50%	1.40%
«все в пул»	0.18%	0.10%
«никаких пулов»	0.70%	0.65%
«смешанный вариант»	0.29%	0.15%

GC паузы; % от общего времени исполнения

А с производительностью то что?

		Plain	SSL
POST	«все в пул»	3% — 25%	2% — 12%
	«никаких пулов»	3% — 27%	2% — 14%
	«смешанный вариант»	6% — 36%	4% — 16%
GET	«все в пул»	2% — 30%	2% — 14%
	«никаких пулов»	2% — 30%	2% — 15%
	«смешанный вариант»	2% — 32%	4% — 17%

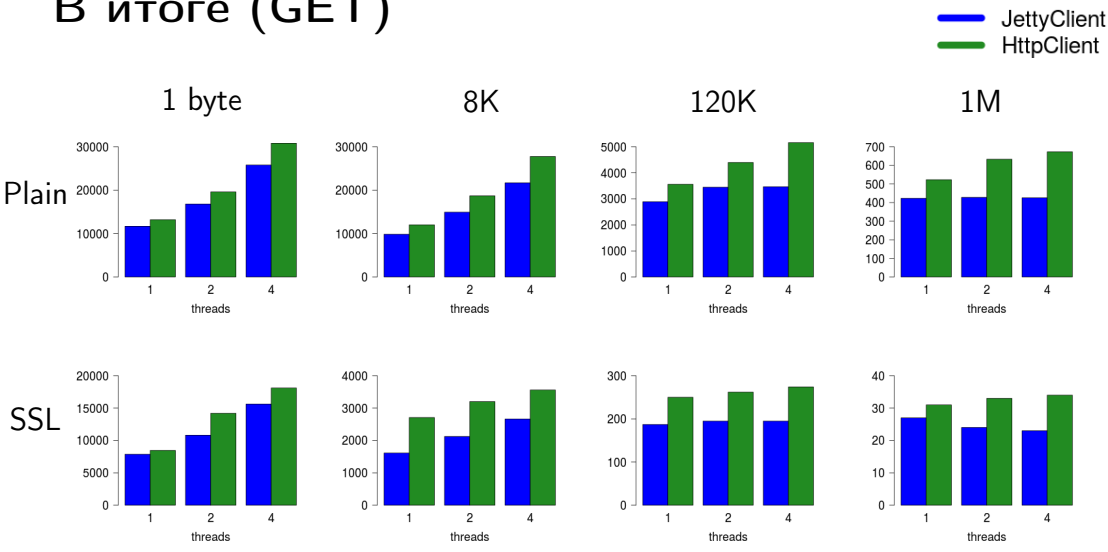
ускорение к базовой версии

Мораль

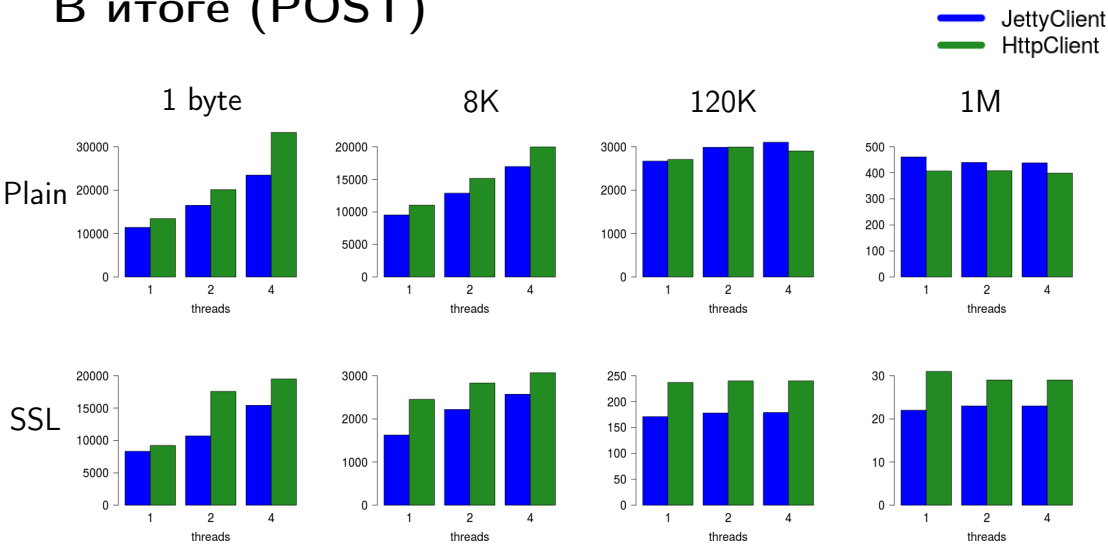
- **Не ориентироваться на «urban legends»**
- **Знать мнения авторитетов**
- **Но нередко «истина где-то рядом»**

В итоге

В итоге (GET)


JettyClient
HttpClient
1 byte
8K
120K
1M
Plain
SSL
threads
30000
20000
10000
0
5000
4000
3000
2000
1000
700
600
500
400
300
200
100
20000
15000
40
30
10
1
2
4

В итоге (POST)


JettyClient
HttpClient
1 byte
8K
120K
1M
Plain
SSL
threads
30000
20000
10000
0
15000
5000
3000
2000
1000
500
400
300
200
100
250
150
50
30
10
1
2
4

А что там с асинхронными запросами?

Переделка №5

Наивный подход

```
public <T> HttpResponse<T>
send(HttpRequest req, HttpResponse.BodyHandler<T> responseHandler) {
    ...
}

public <T> CompletableFuture<HttpResponse<T>>
sendAsync(HttpRequest req, HttpResponse.BodyHandler<T> responseHandler) {
    return CompletableFuture.supplyAsync(() -> send(req, responseHandler), executor);
}
```

Наивный подход

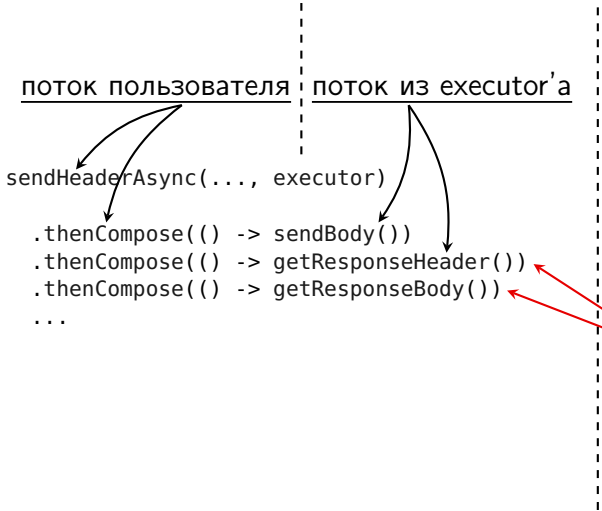
```
public <T> HttpResponse<T>
send(HttpRequest req, HttpResponse.BodyHandler<T> responseHandler) {
    ...
}
```

```
public <T> CompletableFuture<HttpResponse<T>>
sendAsync(HttpRequest req, HttpResponse.BodyHandler<T> responseHandler) {
    return CompletableFuture.supplyAsync(() -> send(req, responseHandler), executor);
}
```

Нельзя просто так взять и сделать «sendAsync»

- отправляем header
- отправляем body
- ждём header от сервера
- ждём body

CompletableFuture - наше все



поток «SelectorManager»

- спит на селекторе
- читает из сокета
- декодирует фреймы
- распределяет фреймы по получателям

`future.complete()`

А что там с производительностью?

Выпиливание `wait()`

+40%

Вопрос

поток 1

```
future.thenSomething((...) -> foo());
```

поток 2

```
future.complete(...);
```

В каком потоке выполнится `foo()`?

A) поток 1
B) поток 2
C) поток 1 или поток 2
D) поток 1 и поток 2

Вопрос

поток 1	поток 2
`future.thenSomething((...) -> foo());`	`future.complete(...);`

В каком потоке выполнится `foo()`?

A) поток 1
B) поток 2
C) поток 1 или поток 2 ← **правильный ответ**
D) поток 1 и поток 2

А что если ответ очень быстр?

```
sendHeaderAsync(..., executor)

  .thenCompose(() -> sendBody())
  .thenCompose(() -> getResponseHeader())
  .thenCompose(() -> getResponseBody())
  ...
```

иногда (3% случаев)
CompletableFuture
уже завершен

`getResponseBody()` запускается из потока пользователя

А есть же `thenComposeAsync()`

А есть же `thenComposeAsync()`

- плюсы:
  - ничего не попадает в пользовательский поток
- минусы:
  - слишком частое переключение с одного потока из executor'а на другой поток из executor'а (дорого)

Делаем раз

```
CompletableFuture<Void> start = new CompletableFuture<>();

start.thenCompose(v -> sendHeader())
.thenCompose(() -> sendBody())
.thenCompose(() -> getResponseHeader())
.thenCompose(() -> getResponseBody())
...;

start.completeAsync( () -> null, executor); // trigger execution
```

А что там с производительностью?

Задержанный запуск

+10%

Тут есть проблема

```
CompletableFuture<...> response;
```

поток из executor'a

```
response
.thenCompose(() -> something());
...;
```

поток «SelectorManager»

```
response.complete(...);
```

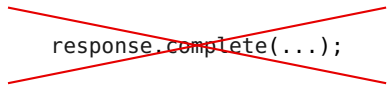
Тут есть проблема

```
CompletableFuture<...> response;
```

поток из executor'a

```
response
.thenCompose(() -> something());
...;
```

поток «SelectorManager»



Делаем раз

```
CompletableFuture<...> response;
```

поток из executor'a

```
response
.thenCompose(() -> something());
...;
```

поток «SelectorManager»

```
response.completeAsync(..., executor);
```

Или два

```
CompletableFuture<...> response;
```

поток из executor'a

```
CompletableFuture<...> cf = response;
cf = cf.thenApplyAsync( x -> x, executor);
...
cf.thenCompose(() -> something());
...;
```

поток «SelectorManager»

```
response.complete(...);
```

Получаем (для обоих вариантов)

Получаем (для обоих вариантов)

- плюсы:
  - ничего не попадает в поток «SelectorManager»
- минусы:
  - частое переключение с одного потока из executor'а на другой поток из executor'а (дорого)

Делаем три

```
CompletableFuture<...> response;
```

поток из executor'а

```
CompletableFuture<...> cf = response;
if(!cf.isDone())
    cf = cf.thenApplyAsync( x -> x, executor);
...
cf.thenCompose(() -> something());
...;
```

поток «SelectorManager»

```
response.complete(...);
```

А что там с производительностью?

Трюки с `complete()`

+16%

А что там с производительностью?

Подкрутка `CompletableFuture`

+80%

Мораль

Изучать новое.
CompletableFuture (@since 1.8)

Переделка №6

Читая доки (HttpClient.Builder)

```
/**
 * Sets the executor to be used for asynchronous tasks. If this method is
 * not called, a default executor is set, which is the one returned from
 * {@link java.util.concurrent.Executors#newCachedThreadPool()
 * Executors.newCachedThreadPool}.
 *
 * @param executor the Executor
 * @return this builder
 */
public abstract Builder executor(Executor executor);
```

Читая доки (java.util.concurrent.Executors)

```
/**
 * Creates a thread pool that creates new threads as needed, but
 * will reuse previously constructed threads when they are
 * available.  These pools will typically improve the performance
 * of programs that execute many short-lived asynchronous tasks.
 * Calls to {@code execute} will reuse previously constructed
 * threads if available. If no existing thread is available, a new
 * thread will be created and added to the pool. Threads that have
 * not been used for sixty seconds are terminated and removed from
 * the cache. Thus, a pool that remains idle for long enough will
 * not consume any resources. Note that pools with similar
 * properties but different details (for example, timeout parameters)
 * may be created using {@link ThreadPoolExecutor} constructors.
 *
 * @return the newly created thread pool
 */
public static ExecutorService newCachedThreadPool()
```

CachedThreadPool

- плюсы:
  - если все потоки заняты, то задача запускается в новом потоке
- минусы:
  - если все потоки заняты, то создается новый поток

Что было до «переделки №5»

На один запрос создается ~ 20 потоков.

100 одновременных запросов $\Rightarrow \sim 2000$ потоков?

Что было до «переделки №5»

На один запрос создается ~ 20 потоков.

100 одновременных запросов $\Rightarrow$ ~~~ 2000 потоков~~?

100 одновременных запросов $\Rightarrow$ OoME!

Что стало после «переделки №5»

На один запрос создается ~ 12 потоков.

100 одновременных запросов $\Rightarrow$ OoME?

Что стало после «переделки №5»

На один запрос создается ~ 12 потоков.

100 одновременных запросов $\Rightarrow$ ~~OoME?~~

100 одновременных запросов $\Rightarrow \sim 800$ потоков*!

*скрипим, но работаем

Перебираем executior'ы, находим лучший

CachedThreadPool	35500 ops/sec
FixedThreadPool(2)	61300 ops/sec

+72%

Мораль

Не все ThreadPool'ы одинаково полезны.

Для тех кто хочет копнуть глубже

Norman Maurer

“Writing Highly Performant Network Frameworks on the JVM - A Love-Hate Relationship”

https://speakerdeck.com/normanmaurer/writing-highly-performant-network-frameworks-on-the-jvm-a-love-hate-relationship

Q & A ?