
ORACLE®

Делаем CompletableFuture быстрее

советы и трюки по производительности

Сергей Куксенко
Oracle
Февраль, 2020

Safe Harbor Statement

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.

Кто я?

- Java Performance Engineer at Oracle, @since 2010
- Java Performance Engineer, @since 2005
- Java Engineer, @since 1996

j.u.c.CompletableFuture

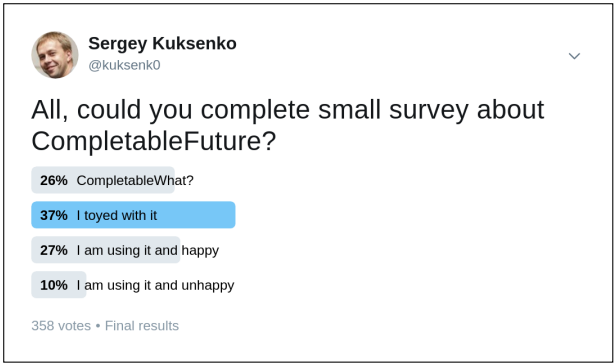
- появился в Java8

j.u.c.CompletableFuture

- появился в Java8
- не использовался в Java8

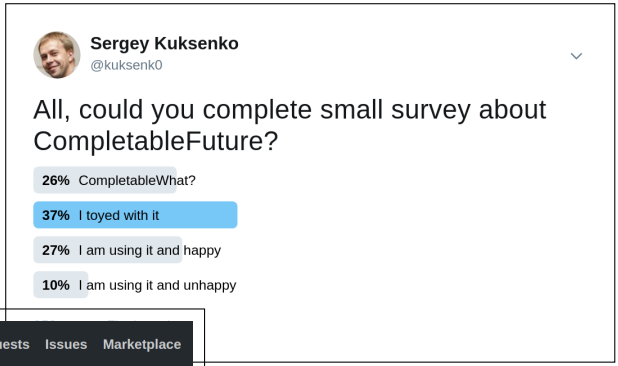
j.u.c.CompletableFuture

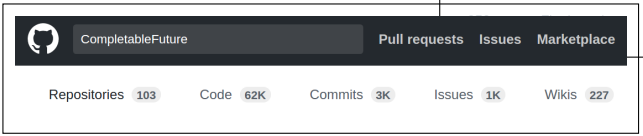
- появился в Java8
- не использовался в Java8



j.u.c.CompletableFuture

- появился в Java8
- не использовался в Java8





j.u.c.CompletableFuture

- Начиная с Java9:
  - Process API
  - HttpClient (до 11 в инкубаторе)*

*основная часть советов отсюда

HttpClient

(a.k.a. JEP-110)

HttpClient

Обработка запросов:

- синхронная (блокирующая)
- асинхронная

синхронная

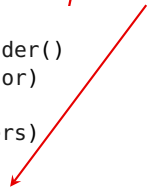
```
HttpClient client = «create client»;
HttpRequest request = «create request»;

HttpResponse<String> response =

                    client.send(request, BodyHandler.asString());

if (response.statusCode() == 200) {
    System.out.println("We've got: " + response.body());
}


...
```

асинхронная

```
HttpClient client = «create client»;
HttpRequest request = «create request»;

CompletableFuture<HttpResponse<String>> futureResponse =

                    client.sendAsync(request, BodyHandler.asString());

futureResponse.thenAccept( response -> {
           if (response.statusCode() == 200) {
                  System.out.println("We've got: " + response.body());
           }
        });
...
```

создание клиента

Хорошее правило для асинхронного API

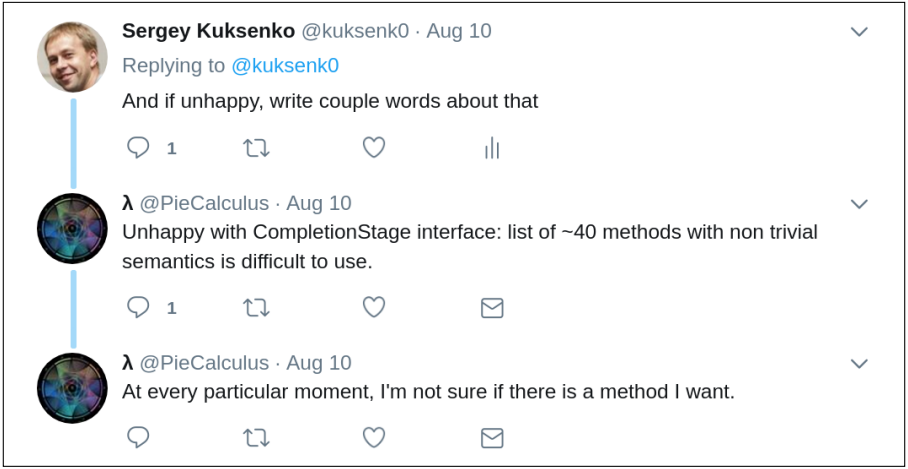


```
HttpClient client = HttpClient.newBuilder()
        .authenticator(someAuthenticator)
        .sslContext(someSSLContext)
        .sslParameters(someSSLParameters)
        .proxy(someProxySelector)
        .executor(someExecutorService)
        .followRedirects(HttpClient.Redirect.ALWAYS)
      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

CompletableFuture

CompletionStage

Про производительность разработчиков



43 метода **CompletionStage**

43 метода **CompletionStage**

43 метода **CompletionStage**

42 имеют вид:

- `somethingAsync(..., executor)`
- `somethingAsync(...)`
- `something(...)`

j.u.c.CompletionStage

- `somethingAsync(..., executor)`
  - выполняем действия в `executor`
- `somethingAsync(...)`
  - `somethingAsync(..., ForkJoinPool.commonPool())`
- `something(...)`
  - по умолчанию *

*поговорим позже

j.u.c.CompletionStage

CompletionStage<T>

- apply(Function<T, R>) ⇒ CompletionStage<R>
  - combine(BiFunction<T, U, R>)
- accept(Consumer<T>) ⇒ CompletionStage<Void>
- run(Runnable) ⇒ CompletionStage<Void>

j.u.c.CompletionStage

- унарные
  - `thenApply, thenAccept, thenRun`
- «OR»
  - `applyToEither, acceptEither, runAfterEither`
- «AND»
  - `thenCombine, thenAcceptBoth, runAfterBoth`

j.u.c.CompletionStage

Осталось:

- thenCompose *(flatMap в мире CF)*
- handle/whenComplete
- exceptionally/exceptionallyCompose
- toCompletableFuture

j.u.c.CompletableFuture

Завершить разными способами

- complete/completeAsync/completeExceptionally
- cancel
- obtrudeValue/obtrudeException
- completeOnTimeout/orTimeout

j.u.c.CompletableFuture

Получить значение

- `get/join` – блокируемся
- `get(timeout, timeUnit)` – чуть-чуть блокируемся
- `getNow(valueIfAbsent)` – не блокируемся

j.u.c.CompletableFuture

Подглядеть статус

- isDone
- isCompletedExceptionally
- isCancelled

j.u.c.CompletableFuture

Создать future

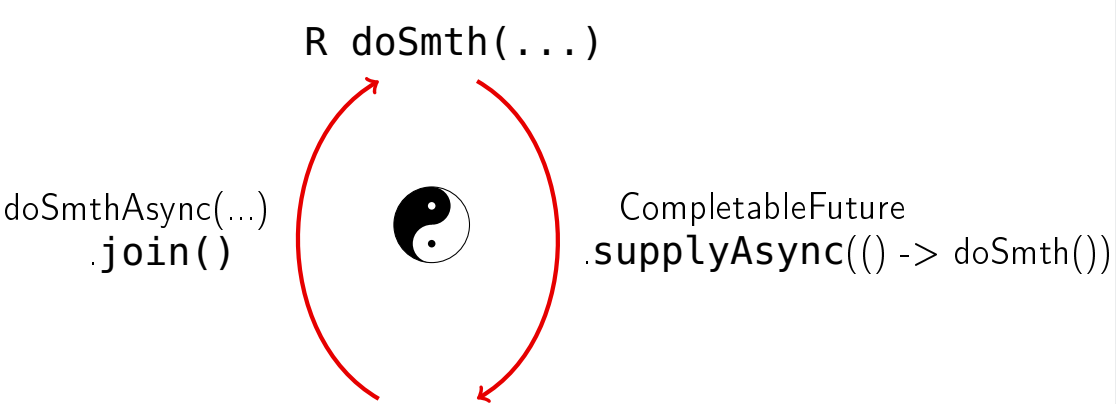
- `completedFuture/completedStage`
- `failedFuture/failedStage`
- `runAsync(Runnable)` → `CompletableFuture<Void>`
- `supplyAsync(Supplier<U>)` → `CompletableFuture<U>`

Блокирующий
или
Асинхронный?

Блокирующий или Асинхронный

- Блокирующий:
  - `R doSmth(...);`
- Асинхронный:
  - `CompletableFuture<R> doSmthAsync(...);`

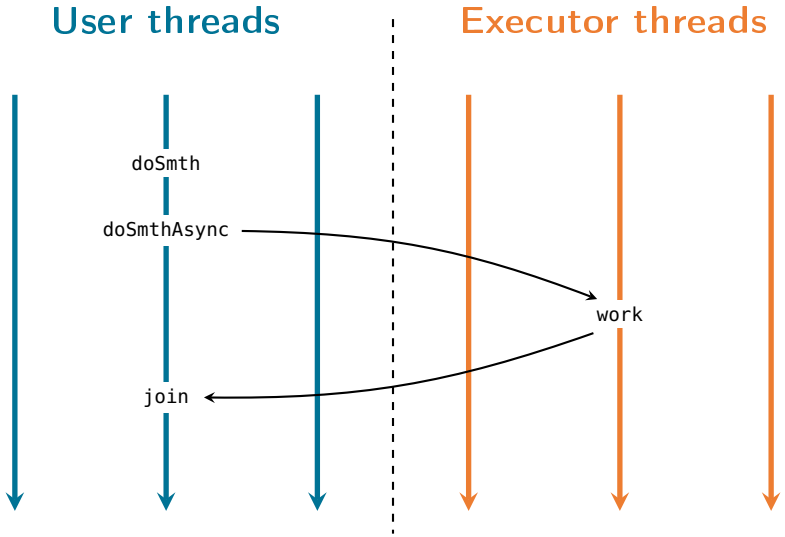
Блокирующий ⟺ Асинхронный



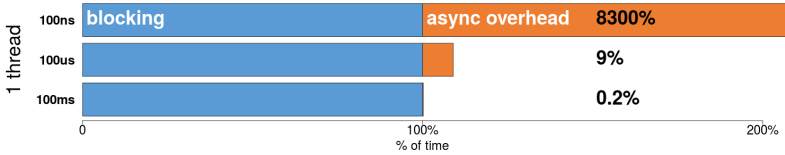
Blocking via async

```
R doSmth(...) {
    return doSmthAsync(...).join();
}
```

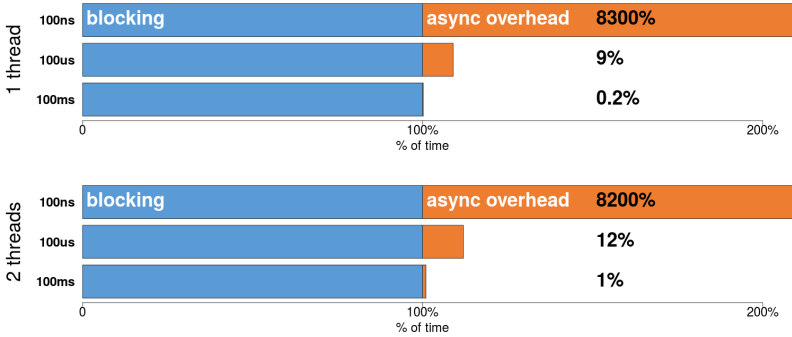
Это вообще работает?


User threads
Executor threads
doSmth
doSmthAsync
work
join

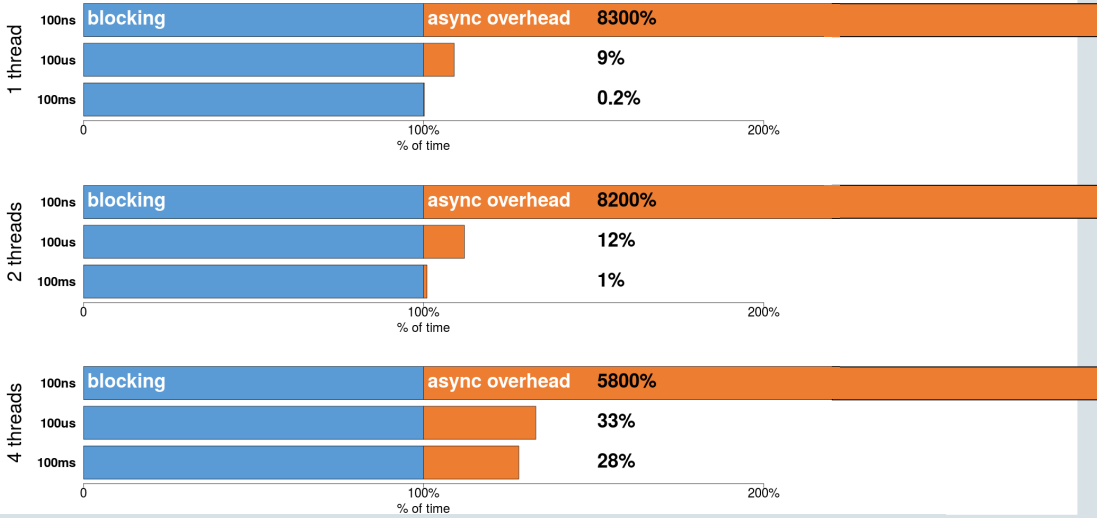
А померить?


1 thread
100ns
blocking
async overhead
8300%
100us
9%
100ms
0.2%
0
100%
200%
% of time

А померить?


1 thread
100ns
blocking
async overhead
8300%
100us
9%
100ms
0.2%
0
100%
200%
% of time
2 threads
100ns
blocking
async overhead
8200%
100us
12%
100ms
1%
0
100%
200%
% of time

А померить?


1 thread
100ns
blocking
async overhead
8300%
100us
9%
100ms
0.2%
0
100%
200%
% of time
2 threads
100ns
blocking
async overhead
8200%
100us
12%
100ms
1%
0
100%
200%
% of time
4 threads
100ns
blocking
async overhead
5800%
100us
33%
100ms
28%
0
100%
200%
% of time

Мораль

Перемещение работы из потока в поток снижает производительность

Async via blocking

```
CompletableFuture<R> doSmthAsync(...) {
    return CompletableFuture.supplyAsync(()->doSmth(...), executor);
}
```

Это вообще работает?

Вернемся к HttpClient'у

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

public <T> CompletableFuture<HttpResponse<T>> sendAsync
                              (HttpRequest req, BodyHandler<T> responseHandler) {

  return CompletableFuture.supplyAsync(() -> send(req, responseHandler), executor);

}
```

Это вообще работает?

Вернемся к HttpClient'у

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

public <T> CompletableFuture<HttpResponse<T>> sendAsync
                            (HttpRequest req, BodyHandler<T> responseHandler) {

  return CompletableFuture.supplyAsync(() -> send(req, responseHandler), executor);

}
```

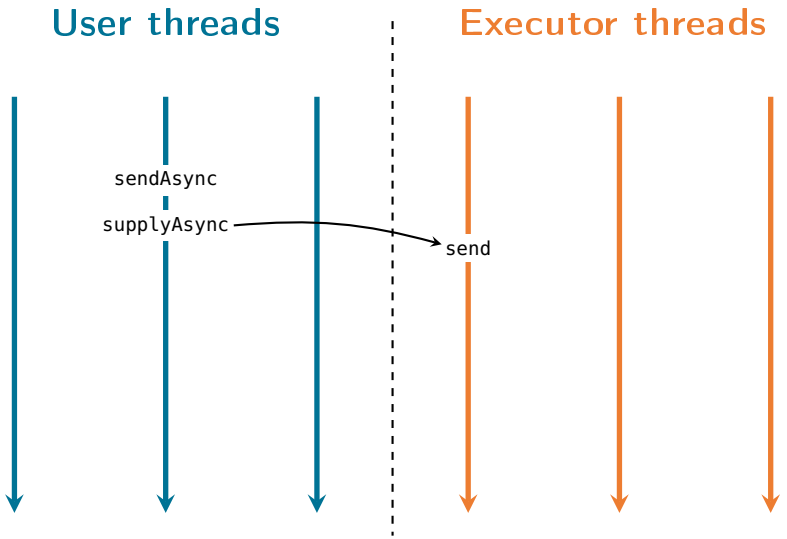
Иногда

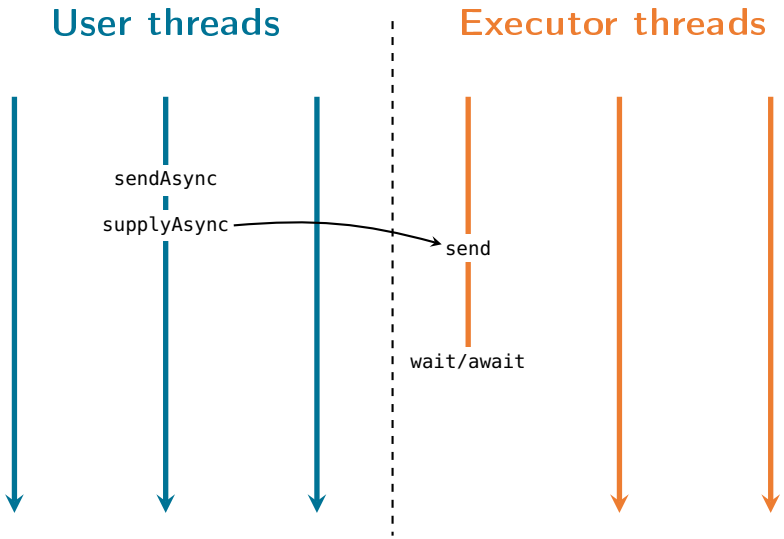
Нельзя так просто взять и сделать «sendAsync»

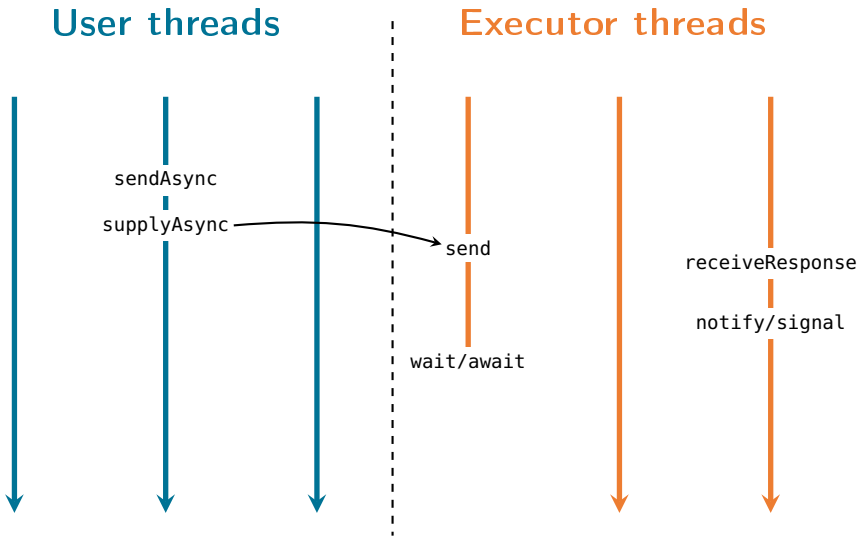
- послать «header»
- послать «body»
- получить «header» от сервера
- получить «body» от сервера

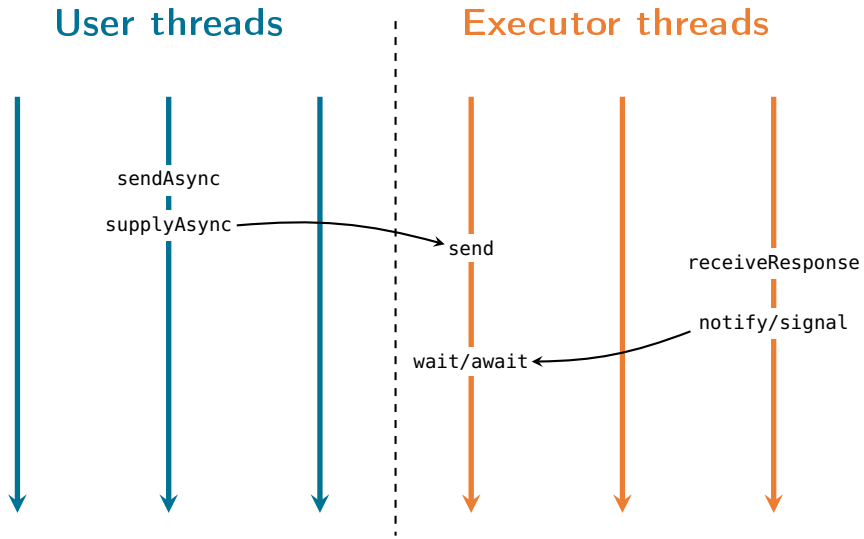
Нельзя так просто взять и сделать «sendAsync»

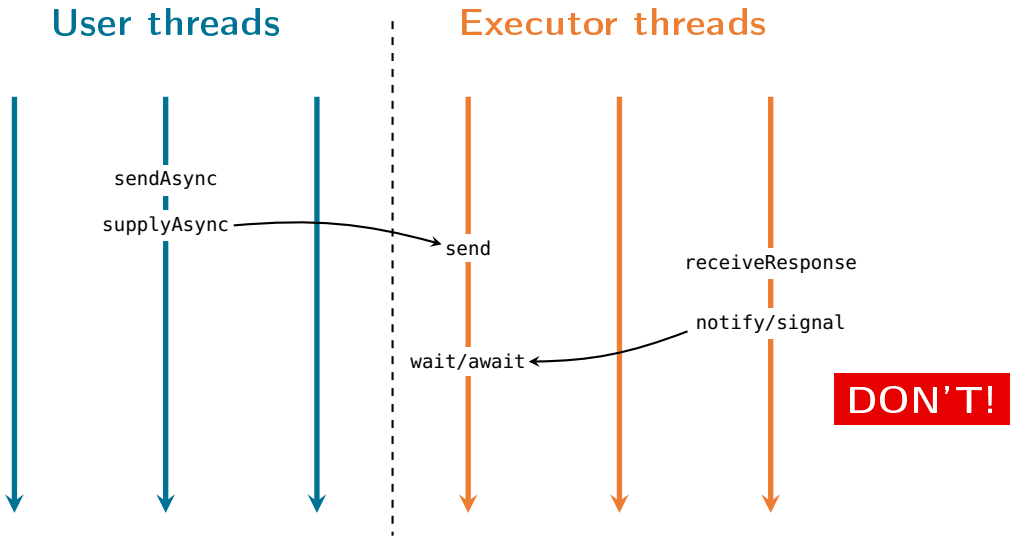
- послать «header»
- послать «body»
- ждать «header» от сервера
- ждать «body» от сервера

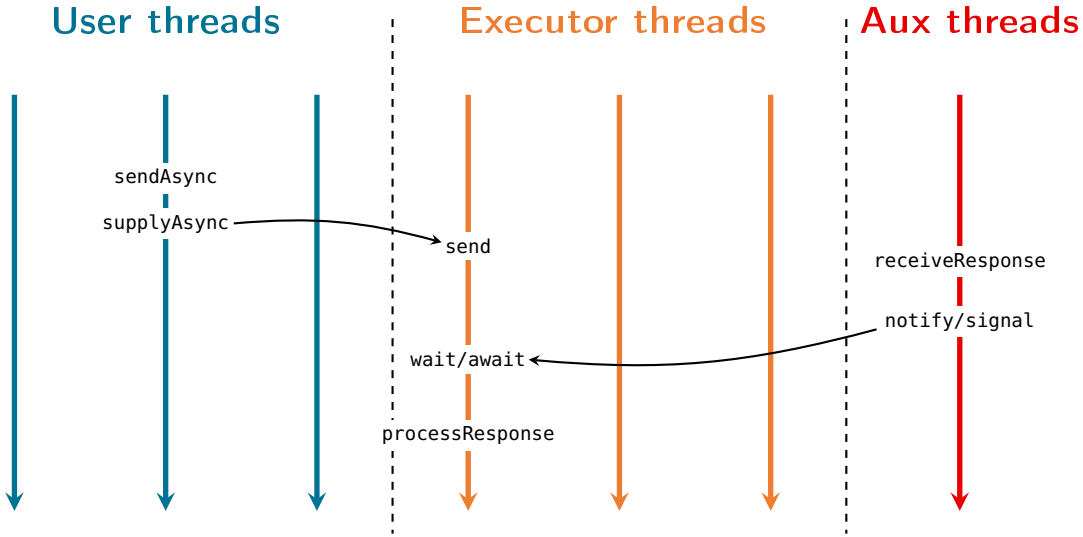

User threads
Executor threads
sendAsync
supplyAsync
send


User threads
Executor threads
sendAsync
supplyAsync
send
wait/await


User threads
Executor threads
sendAsync
supplyAsync
send
wait/await
receiveResponse
notify/signal


User threads
Executor threads
sendAsync
supplyAsync
send
receiveResponse
notify/signal
wait/await


User threads
Executor threads
sendAsync
supplyAsync
send
receiveResponse
notify/signal
wait/await
DON'T!


User threads
Executor threads
Aux threads
sendAsync
supplyAsync
send
receiveResponse
notify/signal
wait/await
processResponse

RTFM (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);
```

Хороший асинхронный API должен работать с любым executor'ом.

RTFM (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

- Что хорошо:
  - если все потоки заняты, задача будет запущена в новом потоке
- Что плохо:
  - если все потоки заняты, новый поток будет создан

sendAsync via send

Один HttpRequest порождал ~ 20 потоков.

Значит ли, что

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

sendAsync via send

Один HttpRequest порождал ~ 20 потоков.

Значит ли, что

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

100 одновременных запросов $\Rightarrow$ OutOfMemoryError!

Удаляем ожидание (шаг 1)

Executor thread

```
Condition responseReceived;

R send(...) {
   sendRequest(...);
   responseReceived.await();
   processResponse();
   ...
}
```

Aux thread

```
... receiveResponse(...) {
   ...
   responseReceived.signal();
   ...
}
```

Удаляем ожидание (шаг 1)

«CompletableFuture» как одноразовый «Condition»

Executor thread

```
CompletableFuture<...> responseReceived;

R send(...) {
   sendRequest(...);
   responseReceived.join();
   processResponse();
   ...
}
```

Aux thread

```
... receiveResponse(...) {
   ...
   responseReceived.complete();
   ...
}
```

Удаляем ожидание (шаг 2)

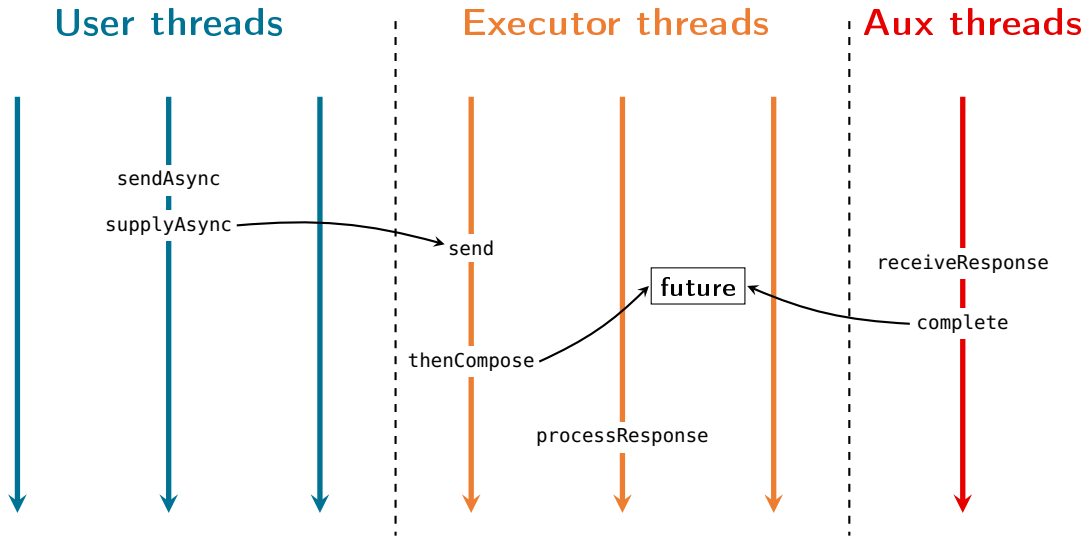
```
CompletableFuture<...> sendAsync(...) {
    return CompletableFuture.supplyAsync(() -> send(...));
}

R send(...) {
   sendRequest(...);
   responseReceived.join();
   return processResponse();
}
```

Удаляем ожидание (шаг 2)

```
CompletableFuture<...> sendAsync(...) {
    return CompletableFuture.supplyAsync(() -> sendRequest(...))
                            .thenApply((...) -> responseReceived.join())
                            .thenApply((...) -> processResponse());
}
```

Удаляем ожидание (шаг 2)

```
CompletableFuture<...> sendAsync(...) {
    return CompletableFuture.supplyAsync(() -> sendRequest(...))
                            .thenCompose((...) -> responseReceived)
                            .thenApply((...) -> processResponse());
}
```


User threads
Executor threads
Aux threads
sendAsync
supplyAsync
send
future
receiveResponse
complete
thenCompose
processResponse

А что там с производительностью?

Удаление `wait()/await()`

+40%

Мораль

Блокировки внутри «CompletableFuture» цепочки снижают производительность

Задачка

Поток 1

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

Поток 2

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

В каком потоке будет выполняться `foo()`?

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

Задачка

Поток 1	Поток 2
`future.thenApply((...) -> foo());`	`future.complete(...);`

В каком потоке будет выполняться **`foo()`**?

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

Так где же CompletableFuture

выполняет действия по умолчанию?

Это очень просто

- Завершающий поток выполняет действия, привязанные «**до**» завершения.
- Конструирующий поток выполняет действия, если CompletableFuture завершен «**до**» конструирования.

Это не всегда просто

- Завершающий поток выполняет действия, привязанные «**до**» завершения.
- Конструирующий поток выполняет действия, если CompletableFuture завершен «**до**» конструирования.

Это параллелизм, тут гонки.

Действия могут быть выполнены в:

- завершающем потоке
  - `complete, completeExceptionally` …
- конструирующем потоке
  - `thenApply, thenCompose` …
- запрашивающем потоке
  - `get, join` …

Где пруфы?

jcstress (все уже написано до нас)

http://openjdk.java.net/projects/code-tools/jcstress/

The Java Concurrency Stress tests (jcstress) is an experimental harness and a suite of tests to aid the research in the correctness of concurrency support in the JVM, class libraries, and hardware.

Пример 1

```
CompletableFuture<...> f = new CompletableFuture<>();
```

`f.complete(...);`	`f.thenApply(a -> action());`

Результаты:

```
  Occurrences   Expectation  Interpretation
1,630,058,138    ACCEPTABLE  action in chain construction thread
  197,470,850    ACCEPTABLE  action in completion thread
```

Пример 2

```
CompletableFuture<...> f = new CompletableFuture<>();

                f.thenApply(a -> action());
```

`f.complete(...);`	`f.complete(...);`

Результаты:

```
Occurrences   Expectation  Interpretation
819,755,198    ACCEPTABLE  action in successful completion thread
163,205,510    ACCEPTABLE  action in failed completion thread
```

Пример 3

```
CompletableFuture<...> f = new CompletableFuture<>();

                f.thenApply(a -> action());
```

f.complete(...);	f.join();

Результаты:

```
Occurrences   Expectation  Interpretation
904,651,258    ACCEPTABLE  action in completion thread
300,524,840    ACCEPTABLE  action in join thread
```

Пример 4

```
CompletableFuture<...> f = new CompletableFuture<>();
            f.thenApply(a -> action1());
            f.thenApply(a -> action2());
```

`f.complete(...);`	`f.join();`

Результаты:

Occurrences	Expectation	Interpretation
179,525,918	ACCEPTABLE	both actions in the same thread
276,608,380	ACCEPTABLE	actions in different threads

Что быстрее?

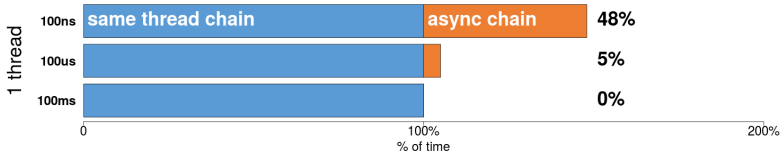
По умолчанию

```
future
  .thenApply((...) -> foo1())
  .thenApply((...) -> foo2())
```

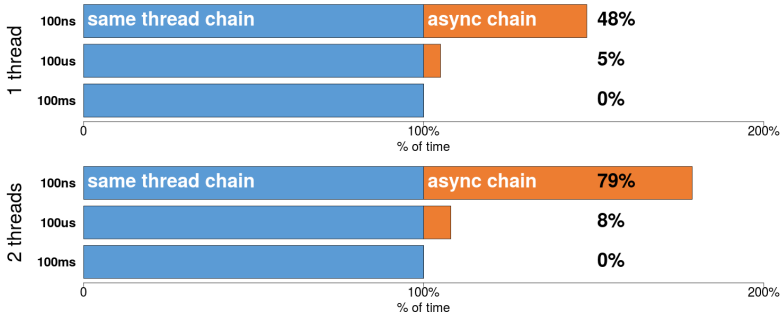
Async

```
future
  .thenApplyAsync((...) -> foo1(), executor)
  .thenApplyAsync((...) -> foo2(), executor);
```

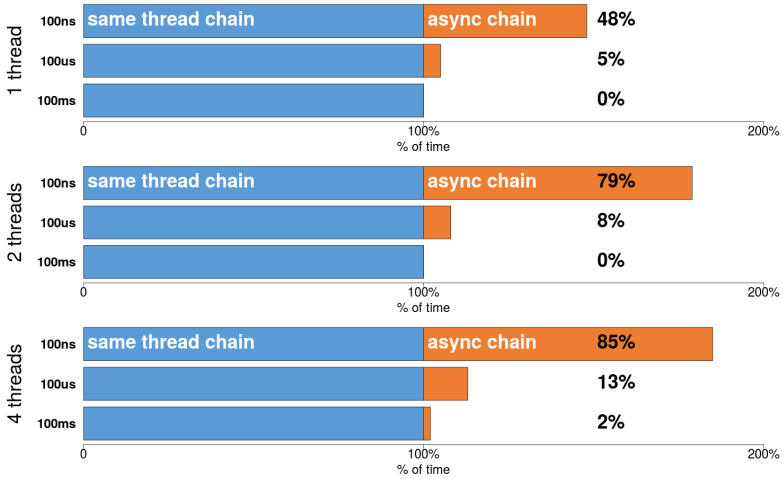
А померить?


1 thread
100ns
same thread chain
async chain
48%
100us
5%
100ms
0%
0
100%
200%
% of time

А померить?


1 thread
100ns
same thread chain
async chain
48%
100us
5%
100ms
0%
0
100%
200%
% of time
2 threads
100ns
same thread chain
async chain
79%
100us
8%
100ms
0%
0
100%
200%
% of time

А померить?


1 thread
100ns
same thread chain
async chain
48%
100us
5%
100ms
0%
0
100%
200%
% of time
2 threads
100ns
same thread chain
async chain
79%
100us
8%
100ms
0%
0
100%
200%
% of time
4 threads
100ns
same thread chain
async chain
85%
100us
13%
100ms
2%
0
100%
200%
% of time

CompletableFuture

- `thenSomethingAsync(...)` – предсказуемость
- `thenSomething(...)` – производительность

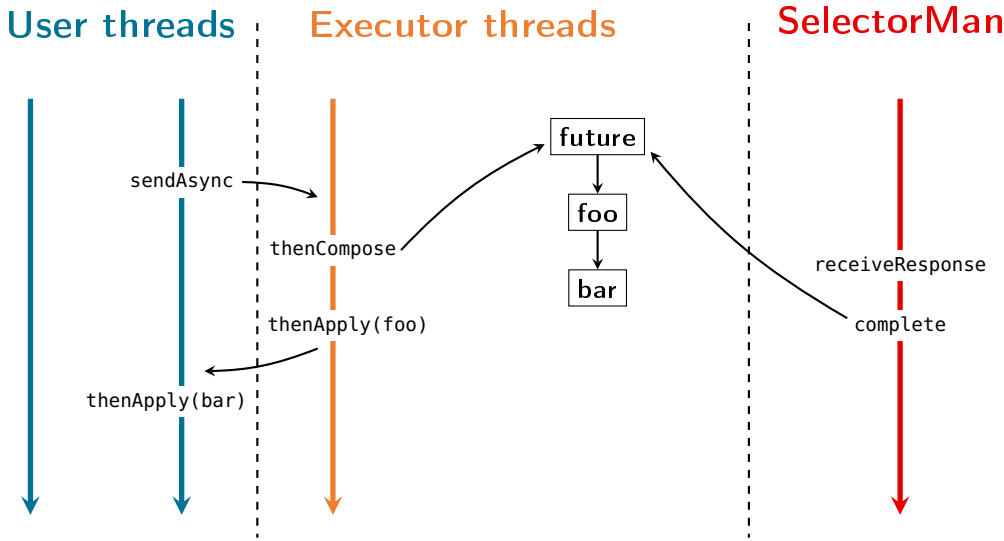
Мораль

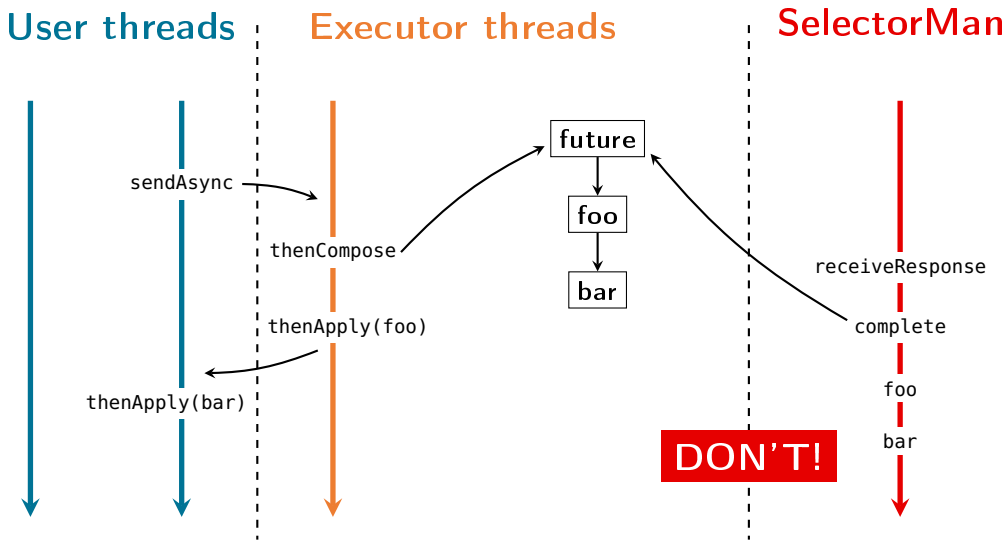
Перемещение работы из потока в поток снижает производительность

Когда необходима предсказуемость

`HttpClient`, вспомогательный поток «SelectorManager»:

- ждет на `Selector.select`
- читает из `Socket`
- выделяет HTTP2 фреймы
- распределяет фреймы получателям


User threads
Executor threads
SelectorManager
sendAsync
thenCompose
future
foo
bar
thenApply(foo)
thenApply(bar)
receiveResponse
complete


User threads
Executor threads
SelectorManager
sendAsync
thenCompose
future
foo
bar
thenApply(foo)
thenApply(bar)
receiveResponse
complete
foo
bar
DON'T!

Когда необходима предсказуемость

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

Executor thread

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

«SelectorManager»

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

Можно так (@since 9)

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

Executor thread

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

«SelectorManager»

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

Или так

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

Executor thread

```
...
.thenComposeAsync(() -> response, executor)
...
```

«SelectorManager»

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

Что мы имеем (в обоих случаях)

- Плюсы:
  - «SelectorManager» защищен
- Минусы:
  - Перемещаем работу из потока в поток

Еще вариант

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

Executor thread

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

«SelectorManager»

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

А что там с производительностью?

Подкрутили `complete()`

⇓

+16%

Мораль

Перемещение работы из потока в поток снижает производительность

А что если ответ приходит очень быстро?

```
CompletableFuture<...> sendAsync(...) {
  return
   sendHeaderAsync(..., executor)
   .thenCompose(() -> sendBody())
   .thenCompose(() -> getResponseHeader())
   .thenCompose(() -> getResponseBody())
   ...
}
```

Иногда (3% запросов) CompletableFuture уже завершен

`getResponseBody()` выполняется в пользовательском потоке

Есть же `thenComposeAsync()`

- Плюсы:
  - Пользовательский поток защищен
- Минусы:
  - Перемещаем работу из потока в поток

Сделаем так

```
CompletableFuture<...> sendAsync(...) {
        CompletableFuture<Void> start = new CompletableFuture<>();

        CompletableFuture<...> end = start.thenCompose(v -> sendHeader())
        .thenCompose(() -> sendBody())
        .thenCompose(() -> getResponseHeader())
        .thenCompose(() -> getResponseBody())
        ...;

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

А что там с производительностью?

Задержанный старт

⇓

+10%

Мораль

Может быть полезно сначала сконструировать, а потом исполнять

Вернемся к `CachedThreadPool`

- Что хорошо:
  - если все потоки заняты, задача будет запущена в новом потоке
- Что плохо:
  - если все потоки заняты, новый поток будет создан

Что выбрать executor'ом по умолчанию?

Попробуем разные

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

+72%

Мораль

Не все ThreadPool'ы одинаково ~~полезны~~ быстры

Золотое правило производительности

Забудьте про черный ящик

**Если вы что-то используете,
то чтобы оно работало быстро,
вы должны знать, как оно устроено внутри**

Q & A ?

Appendix

просто пример thenCompose

```
// e.g. how to make recursive CompletableFuture chain

CompletableFuture<...> makeRecursiveChain(...) {
    if(«recursion ends normally») {
        return CompletableFuture.completedFuture(...);
    else if(«recursion ends abruptly») {
        return CompletableFuture.failedFuture(...); // appeared in Java9
    }
    return CompletableFuture.supplyAsync(() -> doSomething(...))
                            .thenCompose((...) -> makeRecursiveChain(...));
}
```


ORACLE®