Асинхронный HTTP с асинхронным http-клиентом в Java

* [HTTP Client](https://www.codeflow.site/ru/tag/HTTP%20Client) [Library](https://www.codeflow.site/ru/tag/Library)

1. Обзор

[AsyncHttpClient](https://github.com/AsyncHttpClient/async-http-client)(AHC) - это сборка библиотеки поверх ссылки:/netty[Netty]с целью простого выполнения HTTP-запросов и асинхронной обработки ответов.

В этой статье мы расскажем, как настроить и использовать клиент HTTP, как выполнить запрос и обработать ответ с помощью AHC.

2. Настройка

Последнюю версию библиотеки можно найти в [Maven репозитории](https://mvnrepository.com/artifact/org.asynchttpclient/async-http-client). Мы должны быть осторожны, чтобы использовать зависимость с идентификатором группы *org.asynchttpclient*, а не с *com.ning:*

<**dependency**>

<**groupId**>org.asynchttpclient</**groupId**>

<**artifactId**>async-http-client</**artifactId**>

<**version**>2.2.0</**version**>

</**dependency**>

3. Конфигурация клиента HTTP

Самый простой способ получения HTTP-клиента - использование класса *Dsl*. Статический метод *asyncHttpClient ()*возвращает объект *AsyncHttpClient*:

AsyncHttpClient client = Dsl.asyncHttpClient();

Если нам нужна настраиваемая конфигурация HTTP-клиента, мы можем создать объект *AsyncHttpClient*с помощью построителя *DefaultAsyncHttpClientConfig.Builder*:

DefaultAsyncHttpClientConfig.Builder clientBuilder = Dsl.config()

Это дает возможность настроить тайм-ауты, прокси-сервер, HTTP-сертификаты и многое другое:

DefaultAsyncHttpClientConfig.Builder clientBuilder = Dsl.config()

.setConnectTimeout(500)

.setProxyServer(**new** ProxyServer(...));

AsyncHttpClient client = Dsl.asyncHttpClient(clientBuilder);

После того, как мы настроили и получили **экземпляр HTTP-клиента, мы можем повторно использовать его во всем приложении**. Нам не нужно создавать экземпляр для каждого запроса, потому что внутри он создает новые потоки и пулы соединений, что приведет к проблемам с производительностью.

Кроме того, важно отметить, что **когда мы закончили использовать клиент, мы должны вызывать метод *close ()*, чтобы предотвратить любые утечки памяти**или зависание ресурсов.

4. Создание HTTP-запроса

Есть два метода, в которых мы можем определить HTTP-запрос, используя AHC:

* связаны
* несвязанный

Нет большой разницы между двумя типами запросов с точки зрения производительности. Они представляют только два отдельных API, которые мы можем использовать для определения запроса. \*\* Связанный запрос привязан к клиенту HTTP, из которого он был создан, и по умолчанию будет использовать конфигурацию этого конкретного клиента, если не указано иное.

Например, при создании связанного запроса флаг *disableUrlEncoding*считывается из конфигурации клиента HTTP, а для несвязанного запроса по умолчанию устанавливается значение false. Это полезно, потому что конфигурацию клиента можно изменить без перекомпиляции всего приложения, используя системные свойства, передаваемые в качестве аргументов виртуальной машины:

java -jar -Dorg.asynchttpclient.disableUrlEncodingForBoundRequests=**true**

Полный список свойств можно найти в файле *ahc-default.properties*.

4.1. Связанный запрос

Для создания связанного запроса мы используем вспомогательные методы из класса *AsyncHttpClient*, которые начинаются с префикса *«prepare»*. Также мы можем использовать метод *prepareRequest ()*, который получает уже созданный объект *Request*.

Например, метод *prepareGet ()*создаст HTTP-запрос GET:

BoundRequestBuilder getRequest = client.prepareGet("http://www.baeldung.com");

4.2. Несвязанный запрос

Несвязанный запрос может быть создан с помощью класса *RequestBuilder*:

Request getRequest = **new** RequestBuilder(HttpConstants.Methods.GET)

.setUrl("http://www.baeldung.com")

.build();

или с помощью вспомогательного класса *Dsl*, который фактически использует *RequestBuilder*для настройки метода HTTP и URL-адреса запроса:

Request getRequest = Dsl.get("http://www.baeldung.com").build()

5. Выполнение HTTP-запросов

Название библиотеки дает нам подсказку о том, как запросы могут быть выполнены. AHC поддерживает как синхронные, так и асинхронные запросы.

Выполнение запроса зависит от его типа. При использовании **связанного запроса мы используем метод *execute ()*из класса *BoundRequestBuilder***, а когда у нас есть **несвязанный запрос, мы выполним его, используя одну из реализаций метода executeRequest () из интерфейса *AsyncHttpClient***.

5.1. Синхронно

Библиотека была разработана, чтобы быть асинхронной, но при необходимости мы можем симулировать синхронные вызовы, блокируя объект *Future*. Методы *execute ()*и *executeRequest ()*возвращают объект *ListenableFuture <Response>*. Этот класс расширяет интерфейс Java *Future*, таким образом наследуя метод *get ()*, который можно использовать для блокировки текущего потока до завершения HTTP-запроса и возврата ответа:

Future<Response> responseFuture = boundGetRequest.execute();

responseFuture.get();

Future<Response> responseFuture = client.executeRequest(unboundRequest);

responseFuture.get();

Использование синхронных вызовов полезно при попытке отладки частей нашего кода, но его не рекомендуется использовать в производственной среде, где асинхронное выполнение приводит к повышению производительности и пропускной способности.

5.2. Асинхронный

Когда мы говорим об асинхронных выполнениях, мы также говорим о слушателях для обработки результатов. Библиотека AHC предоставляет 3 типа слушателей, которые можно использовать для асинхронных HTTP-вызовов:

* *AsyncHandler*
* *AsyncCompletionHandler*
* *ListenableFuture*слушатели

Слушатель *AsyncHandler*предлагает возможность контролировать и обрабатывать HTTP-вызов до его завершения. С его помощью можно обработать серию событий, связанных с вызовом HTTP:

request.execute(**new** AsyncHandler<Object>() {

@Override

**public** State **onStatusReceived**(HttpResponseStatus responseStatus)

**throws** Exception {

**return** **null**;

}

@Override

**public** State **onHeadersReceived**(HttpHeaders headers)

**throws** Exception {

**return** **null**;

}

@Override

**public** State **onBodyPartReceived**(HttpResponseBodyPart bodyPart)

**throws** Exception {

**return** **null**;

}

@Override

**public** **void** **onThrowable**(Throwable t) {

}

@Override

**public** Object **onCompleted**() **throws** Exception {

**return** **null**;

}

});

Перечисление *State*позволяет нам контролировать обработку HTTP-запроса. Возвращая ***State.ABORT*, мы можем остановить обработку**в определенный момент, а с помощью *State.CONTINUE*мы дадим завершить обработку.

Важно отметить, что \*\* *AsyncHandler*не является потокобезопасным и не должен использоваться повторно при выполнении одновременных запросов.

*AsyncCompletionHandler*наследует все методы из интерфейса *AsyncHandler*и добавляет вспомогательный метод *onCompleted (Response)*для обработки завершения вызова. Все остальные методы прослушивателя переопределяются для возврата *\_State . \_*CONTINUE, что делает код более читабельным:

request.execute(**new** AsyncCompletionHandler<Object>() {

@Override

**public** Object **onCompleted**(Response response) **throws** Exception {

**return** response;

}

});

Интерфейс *ListenableFuture*позволяет нам добавлять прослушиватели, которые будут запускаться после завершения HTTP-вызова.

Кроме того, он позволяет выполнять код от слушателей - используя другой пул потоков:

ListenableFuture<Response> listenableFuture = client

.executeRequest(unboundRequest);

listenableFuture.addListener(() -> {

Response response = listenableFuture.get();

LOG.debug(response.getStatusCode());

}, Executors.newCachedThreadPool());

Кроме того, опция добавления слушателей, интерфейс *ListenableFuture*позволяет нам преобразовывать ответ *Future*в *CompletableFuture*.

7. Заключение

AHC - очень мощная библиотека с множеством интересных функций. Он предлагает очень простой способ настройки HTTP-клиента и возможность выполнения как синхронных, так и асинхронных запросов.

Как всегда, исходный код статьи доступен на [over на GitHub](https://github.com/eugenp/tutorials/tree/master/libraries).