alexVengrovsk 4 сентября 2014 в 11:10

Основные отличия Java IO и Java NIO

Программирование, Java

Из песочницы

Когда я начал изучать стандартный ввод/вывод в Java, то первое время был немного шокирован обилием интерфейсов и классов пакета **java.io.***, дополненных еще и целым перечнем специфических исключений.

Потратив изрядное количество часов на изучение и реализацию кучи разнообразных туториалов из Интернета, начал чувствовать себя уверенно и вздохнул с облегчением. Но в один прекрасный момент понял, что для меня все только начинается, так как существует еще и пакет **java.nio.***, известный ещё под названием Java NIO или Java New IO. Вначале казалось, что это тоже самое, ну типа вид сбоку. Однако, как оказалось, есть существенные отличия, как в принципе работы, так и в решаемых с их помощью задачах.

Разобраться во всем этом мне здорово помогла статья Джакоба Дженкова (Jakob Jenkov) – **"Java NIO vs. IO"**. Ниже она приводиться в адаптированном виде.

Поспешу заметить, что статья не является руководством по использованию Java IO и Java NIO. Её цель – дать людям, начинающим изучать Java, возможность понять концептуальные отличия между двумя указанными инструментами организации ввода/вывода.

Основные отличия между Java IO и Java NIO

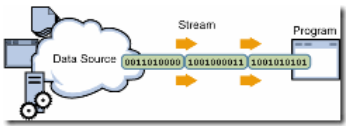
IO	NIO
Потокоориентированный	Буфер-ориентированный
Блокирующий (синхронный) ввод/вывод	Неблокирующий (асинхронный) ввод/вывод
	Селекторы

Потокоориентированный и буфер-ориентированный ввод/вывод

Основное отличие между двумя подходами к организации ввода/вывода в том, что Java IO является потокоориентированным, а Java NIO – буфер-ориентированным. Разберем подробней.

Потокоориентированный ввод/вывод подразумевает чтение/запись из потока/в поток одного или нескольких байт в единицу времени поочередно. Данная информация нигде не кэшируются. Таким образом, невозможно произвольно двигаться по потоку данных вперед или назад. Если вы хотите произвести подобные манипуляции, вам придется сначала кэшировать данные в буфере.

Потокоориентированный ввод:



Потокоориентированный вывод:

Подход, на котором основан Java NIO немного отличается. Данные считываются в буфер для последующей обработки. Вы

можете двигаться по буферу вперед и назад. Это дает немного больше гибкости при обработке данных. В то же время, вам необходимо проверять содержит ли буфер необходимый для корректной обработки объем данных. Также необходимо следить, чтобы при чтении данных в буфер вы не уничтожили ещё не обработанные данные, находящиеся в буфере.

Блокирующий и неблокирующий ввод/вывод

Потоки ввода/вывода (streams) в Java IO являются блокирующими. Это значит, что когда в потоке выполнения (tread) вызывается read() или write() метод любого класса из пакета java.io.*, происходит блокировка до тех пор, пока данные не будут считаны или записаны. Поток выполнения в данный момент не может делать ничего другого.

Неблокирующий режим Java NIO позволяет запрашивать считанные данные из канала (channel) и получать только то, что доступно на данный момент, или вообще ничего, если доступных данных пока нет. Вместо того, чтобы оставаться заблокированным пока данные не станут доступными для считывания, поток выполнения может заняться чем-то другим.

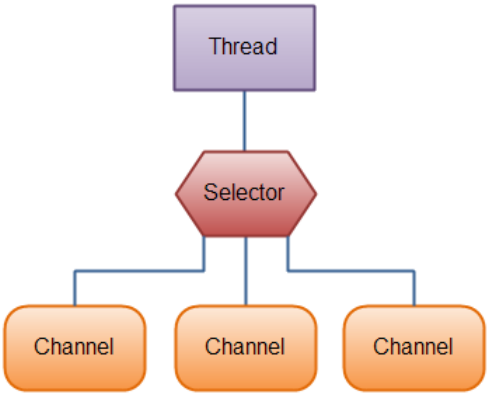
Каналы (channels)

Тоже самое справедливо и для неблокирующего вывода. Поток выполнения может запросить запись в канал некоторых данных, но не дожидаться при этом пока они не будут полностью записаны.

Таким образом неблокирующий режим Java NIO позволяет использовать один поток выполнения для решения нескольких задач вместо пустого прожигания времени на ожидание в заблокированном состояний. Наиболее частой практикой является использование сэкономленного времени работы потока выполнения на обслуживание операций ввода/вывода в другом или других каналах.

Селекторы

Селекторы в Java NIO позволяют одному потоку выполнения мониторить несколько каналов ввода. Вы можете зарегистрировать несколько каналов с селектором, а потом использовать один поток выполнения для обслуживания каналов, имеющих доступные для обработки данные, или для выбора каналов, готовых для записи.



Чтобы лучше понять концепцию и выгоду от применения селекторов, давайте абстрагируемся от программирования и представим себе железнодорожный вокзал. Вариант без селектора: есть три железнодорожных пути (каналы), на каждый из них в любой момент времени может прибыть поезд (данные из буфера), на каждом пути постоянно ожидает сотрудник вокзала (поток выполнения), задача которого – обслуживание прибывшего поезда. В результате трое сотрудников постоянно находятся на вокзале даже если там вообще нет поездов. Вариант с селектором: ситуация та же, но для каждой платформы есть индикатор, сигнализирующий сотруднику вокзала (поток выполнения) о прибытии поезда. Таким образом на вокзале достаточно присутствия одного сотрудника.

Влияние Java NIO и Java IO на дизайн приложения

Выбор между Java NIO и Java IO может на следующие аспекты дизайна вашего приложения:
1. API обращений к классам ввода/вывода;
2. Обработка данных;
3. Количество потоков выполнения, использованных для обработки данных.

API обращений к классам ввода/вывода

Естественно, использование Java NIO серьезно отличается от использования Java IO. Так как, вместо чтения данных байт за байтом с использованием, например InputStream, данные для начала должны быть считаны в буфер и браться для обработки

уже оттуда.

Обработка данных

Обработка данных при использовании Java NIO тоже отличается.
Как уже упоминалось, при использовании Java IO вы читаете данные байт за байтом с InputStream или Reader. Представьте, что вы проводите считывание строк текстовой информации:

Name: Anna
Age: 25
Email: anna@mailserver.com
Phone: 1234567890

Этот поток строк текста может обрабатываться следующим образом:

```
InputStream input = ... ;
BufferedReader reader = new BufferedReader(new InputStreamReader(input));
String nameLine   = reader.readLine();
String ageLine    = reader.readLine();
String emailLine  = reader.readLine();
String phoneLine  = reader.readLine();
```

Обратите внимание, как состояние процесса обработки зависит от того, насколько далеко продвинулось выполнение программы. Когда первый метод readLine() возвращает результат выполнения, вы уверенны – целая строка текста была считана. Метод является блокирующим и действие блокировки продолжается до тех пор, пока вся строка не будет считана. Вы также четко понимаете, что данная строка содержит имя. Подобно этому, когда метод вызывается во второй раз, вы знаете, что в результате получите возраст.

Как вы видите, прогресс в выполнении программы достигается только тогда, когда доступны новые данные для чтения, и для каждого шага вы знаете что это за данные. Когда поток выполнения достигает прогресса в считывании определенной части данных, поток ввода (в большинстве случаев) уже не двигает данные назад. Данный принцип хорошо демонстрирует следующая схема:

Имплементация с использованием Java IO будет выглядеть несколько иначе:

```
ByteBuffer buffer = ByteBuffer.allocate(48);
int bytesRead = inChannel.read(buffer);
```

Обратите внимание на вторую строчку кода, в которой происходит считывание байтов из канала в ByteBuffer. Когда возвращается результат выполнения данного метода, вы не можете быть уверенны, что все необходимые вам данные находятся внутри буфера. Все, что вам известно, так это то, что буфер содержит некоторые байты. Это немного усложняет процесс обработки.

Представьте, что после первого вызова метода read(buffer), в буфер было считано только половину строки. Например, “Name: An”. Сможете ли вы обработать такие данные? Наверное, что нет. Вам придется ждать пока, по крайней мере, одна полная

строка текста не будет считана в буфер.

Так как же вам узнать, достаточно ли данных для корректной обработки содержит буфер? А никак. Единственный вариант узнать, это посмотреть на данные, содержащиеся внутри буфера. В результате вам придется по нескольку раз проверять данные в буфере, пока они не станут доступными для корректной обработки. Это неэффективно и может негативно сказаться на дизайне программы. Например:

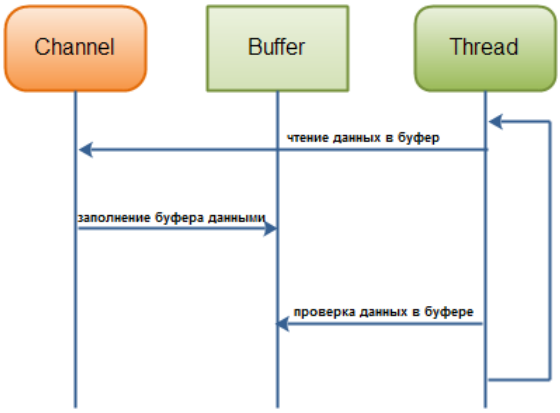
```
ByteBuffer buffer = ByteBuffer.allocate(48);
        int bytesRead = inChannel.read(buffer);
                while(! bufferFull(bytesRead) ) {
                    bytesRead = inChannel.read(buffer);
                }
```

Метод bufferFull() должен следить за тем, сколько данных считано в буфер и возвращать true или false, в зависимости от того, заполнен буфер или нет. Другими словами, если буфер готов к обработке, то он считается заполненным.

Также метод bufferFull() должен оставлять буфер в неизмененном состоянии, поскольку в противном случае следующая порция считанных данных может быть записана в неправильное место.

Если буфер заполнен, данные из него могут быть обработаны. Если он не заполнен вы все же будете иметь возможность обработать уже имеющиеся в нем данные, если это имеет смысл в вашем конкретном случае. В большинстве случаев – это бессмысленно.

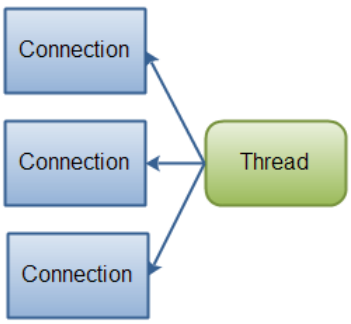
Следующая схема демонстрирует процесс определения готовности данных в буфере для корректной обработки:



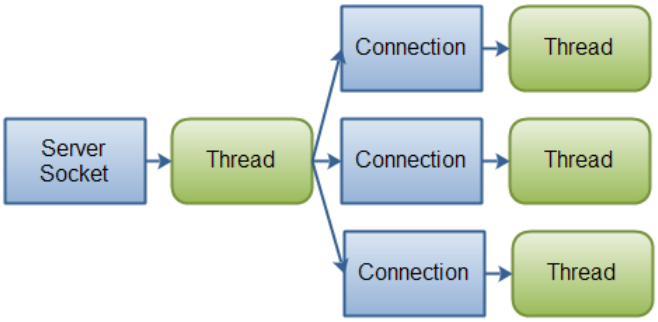
Итоги

Java NIO позволяет управлять несколькими каналами (сетевыми соединениями или файлами) используя минимальное число потоков выполнения. Однако ценой такого подхода является более сложный, чем при использовании блокирующих потоков, парсинг данных.

Если вам необходимо управлять тысячами открытых соединений одновременно, причем каждое из них передает лишь незначительный объем данных, выбор Java NIO для вашего приложения может дать преимущество. Дизайн такого типа схематически изображен на следующем рисунке:



Если вы имеете меньшее количество соединений, по которым передаются большие объемы данных, то лучшим выбором станет классический дизайн системы ввода/вывода:



Важно понимать, что Java NIO отнюдь не является заменой Java IO. Его стоит рассматривать как усовершенствование – инструмент, позволяющий значительно расширить возможности по организации ввода/вывода. Грамотное использование мощи обоих подходов позволит вам строить хорошие высокопроизводительные системы.

Стоит заметить, что с выходом версии Java 1.7 появился еще и Java NIO.2, но присущие ему новшества касаются, в первую очередь, работы с файловым вводом/выводом, поэтому выходят за рамки этой статьи.

PS: также в данном посте использованы материалы очень достойной статьи Nino Guarnacci – «Java.nio vs Java.io»

Теги: java nio, java io

Хабы: Программирование, Java

Комментарии 17

pyatigil 4 сентября 2014 в 12:09 +5

Глаз режет расшифровка nio как new io. С чего бы это, если nio означает non-blocking io?

Crazybot 4 сентября 2014 в 14:31 0

Я сморозил

kozhevnikovv 4 сентября 2014 в 20:31 0

Я думаю ноги растут из википедии.

QtRoS 4 сентября 2014 в 20:33 +4

Вот это поворот, всю жизнь думал, что это new io (и мысленно ругал, ибо что потом будет, newest io и тд?)

pyatigil 5 сентября 2014 в 15:50 +1

потом, как теперь известно, стало NIO2: jcp.org/en/jsr/detail?id=203

drJonnie 4 сентября 2014 в 21:07 +16

Нет, в джаве это именно расшифровывется как new io. Вот полное название JSR 51: New I/O APIs for the JavaTM Platform. В седьмой добавили NIO.2 JSR 203: More New I/O APIs for the JavaTM Platform («NIO.2»)

pyatigil 4 сентября 2014 в 23:31 0

Спасибо, не знал!

alexVengrovsk 5 сентября 2014 в 08:30 0

Спасибо за комментарий. Старое доброе правило работает: «Если сомневаешься, то смотри стандартную документацию»

fshp 5 сентября 2014 в 12:57 0

nbio тогда назывался бы.

Avers 8 сентября 2014 в 16:33 +1

Версию new IO предлагает также Греберт Шилдт (Java Полное руководство, у меня 8е издание стр. 641)

in3gant 4 сентября 2014 в 13:54 −1

Java IO было с первых версий, Java NIO появилось только с 1.4, отсюда можно вывести, что Новая

Cupper 5 сентября 2014 в 01:29 +2

```
ByteBuffer buffer = ByteBuffer.allocate(48);
int bytesRead = inChannel.read(buffer);
while(! bufferFull(bytesRead) ) {
bytesRead = inChannel.read(buffer);
}
```

are you kidding me? Never use this approach, it's way to nowhere. You need to better learn what selector is and how to use non-blocking operations.

alexVengrovsk 5 сентября 2014 в 08:27 0

You are right! But in this article this source code is not a tutorial of using Selector, it is an example of wrong desigh approach. That is why before this code I wrote: «Это неэффективно и может негативно сказаться на дизайне программы. Например:». The translation of this phrase is — «It isn't effective and can have a negative impact on app design. For example:»

Cupper 5 сентября 2014 в 10:13 0

Oh, I've missed it :) But I still think it's not good way to present non-bloking part of library. It's better to say that we could do something in blocking mode but in some cases it's bad solutions and at this time we can use non-bloking IO and bla bla bla.
I'm writing on english just because I don't have rus keyboard at work and I still can read russian :)

eyeless_watcher 5 сентября 2014 в 10:44 +6

translit.net/

Slider_X 6 сентября 2014 в 00:12 +2

www.google.com/search?q=обучение%20слепой%20печати

R_Voland 8 апреля 2020 в 15:05 0

Вот интересно что будет если прибудут сразу три поезда? Куда кинется сотрудник вокзала?
В целом же — в каждом из вариантов свои проблемы. IO тратит больше ресурсов на сетевую подсистему и меньше на обработку данных, NIO — наоборот.

Только полноправные пользователи могут оставлять комментарии. Войдите, пожалуйста.

САМОЕ ЧИТАЕМОЕ

Сутки | Неделя | Месяц

В IT растет цензура, а мы не замечаем — разрешают только улыбаться и молчать

+339 55,2k 151 340

Стивен Вольфрам: кажется, мы близки к пониманию фундаментальной теории физики, и она прекрасна

+116 30k 197 103

В сеть утекли исходные коды операционной системы Windows XP и Server 2003

+73 33,2k 43 189

Как я избавлялся от Google на Android

+43 23,5k 114 70

Фриланс с соцпакетом – дело недалёкого будущего. Или нет?

Мегапост