

# Использование `select` для высокоскоростного сетевого взаимодействия

*Новые библиотеки ввода/вывода ускорят ваш сервер.*

J2SE

Грэг Тревис

Стандартная редакция Java 2 (J2SE) в версии 1.4 ввела Новые библиотеки Ввода/Вывода (New Input/Output — NIO), предназначенные для реализации высокопроизводительного ввода/вывода в Java приложениях. NIO использует модель ввода/вывода значительно отличающуюся от использованной в первоначальных библиотеках ввода/вывода. Эта статья пошагово продемонстрирует использование службы `select` предоставляемой NIO. `select` даёт вашему серверу возможность обрабатывать огромное количество данных поступающих от множества соединений. После короткого введения в библиотеки NIO, эта статья рассматривает теорию, лежащую в основе службы `select` и разбирает исходный код работающего сервера, использующего `select`. *(В оригинальной версии на английском языке 2000 слов;)*

Java использует очень изящную модель ввода/вывода, основанную на идее потоков (*stream*). Поток является объектом, который порождает или потребляет строки или байты. Потоки могут быть соединены вместе для реализации операций фильтрации и расширены для обработки других типов данных. Поточная модель очень гибкая, но не слишком быстрая. Это подходит для большинства приложений, но некоторые системы требуют максимальной скорости, какую только может обеспечить аппаратное обеспечение. Иногда поточная модель не может обеспечить этого.

Новые библиотеки ввода/вывода (NIO) введённые в стандартной редакции Java 2 (J2SE) версии 1.4 нацелены на решение данной проблемы. NIO использует основанную

на буферах (*buffer*) модель. Это значит, что NIO работает с данными в основном большими блоками. Это исключает издержки, вызываемые использованием поточной модели, и позволяет использовать службы уровня операционной системы, когда это возможно, для достижения максимальной пропускной способности.

Для начала, рассмотрим, как NIO работает, а затем внедрим его в высокоскоростном серверном приложении.

**Замечание:**

Вы можете скачать исходные коды этой статьи из [Ресурсов](#).

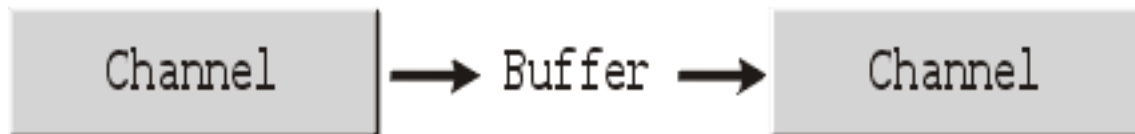
## 1. Система NIO

NIO основывается на двух понятиях, каналах (*channels*) и буферах (*buffers*). Каналы являются примерным аналогом потоков, используемых в поточной модели. Буфера не имеют аналогов в поточной модели.

Основные потоки, `InputStream` и `OutputStream`, могут читать и записывать байты; подклассы этих поточных классов могут читать и записывать другие типы данных. В NIO все данные читаются и записываются через буфера. Смотрите сравнение двух моделей на Рисунке 1.



## The stream model



## The NIO model

Рисунок 1. Поточная модель использует потоки и байты; NIO модель использует каналы и буфера.

Заметим также, что когда поточная модель проводит различие между `InputStream` и `OutputStream`, NIO использует один тип объектов канала (`Channel`) вместо обоих.

Основным преимуществом буферов является то, что они работают с данными блоками. Вы можете читать и записывать большие блоки данных, и размер буферов, используемых при этом, будет ограничен только размером памяти, который вы готовы выделить для них.

Другим более хитрым преимуществом буферов является то, что они могут представлять буфера системного уровня. Некоторые операционные системы используют унифицированную схему памяти, что позволяет осуществлять ввод/вывод без копирования данных из системной памяти в память приложения. Некоторые реализации предоставляют объекты буфера (`Buffer`), представляющие непосредственно системный буфер, что позволяет вам читать и записывать данные с минимальным копированием данных (смотрите Рисунок 2).

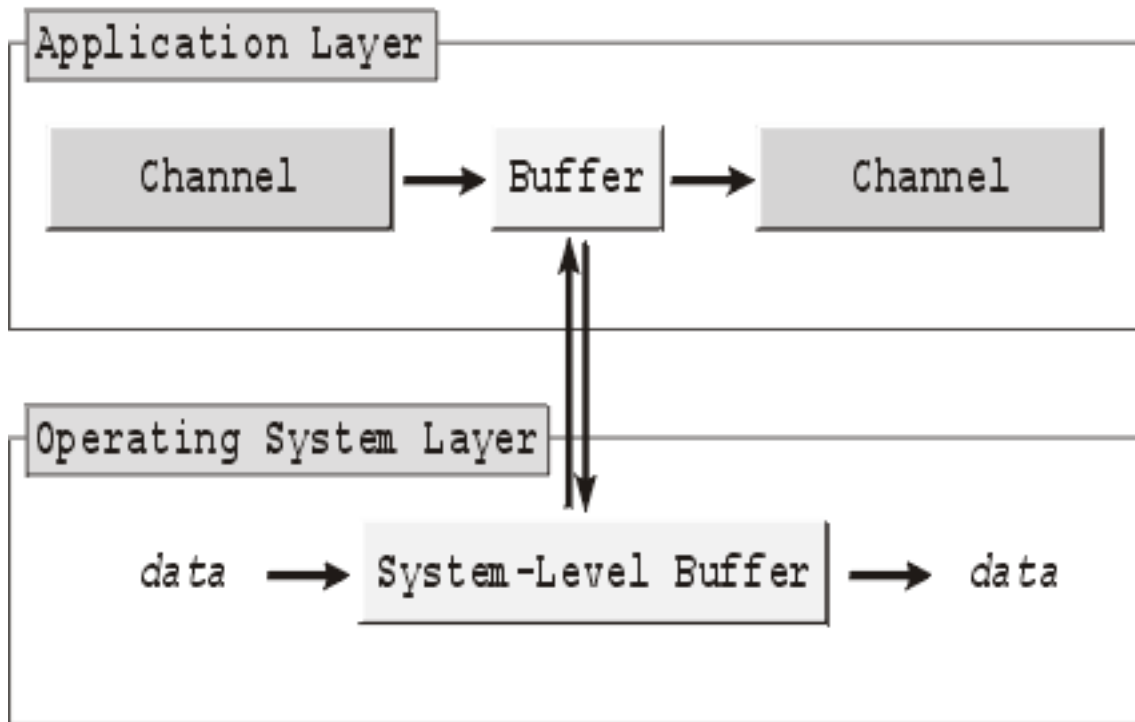


Рисунок 2. Системные буфера позволяют вам непосредственно использовать буфера системного уровня, избегая дополнительного копирования данных.

## 2. Служба *select*

Служба *select* являются прекрасным способом для работы одновременно с большим количеством источников данных. Она получила своё название от системного вызова Unix — *select()*, предоставляющего схожие возможности программам на C, запущенным на Unix системах.

Обычно, ввод/вывод осуществляется через блокирующие системные вызовы. Когда вы вызываете метод *read()* для входного потока, метод блокируется до появления некоторых данных. Если вы читаете из локального файла, вам не придётся долго ждать до появления данных. С другой стороны, если вы осуществляете чтение с сетевого файлового сервера или сетевого сокетного соединения, вы можете ждать долго. Пока вы ожидаете, ваш читающий поток не может делать что-то ещё.

В Java, конечно, можно легко создать много нитей (*thread*), которые будут читать из

множества потоков. Однако, нити могут быть ресурсо-затратными. Во множестве реализаций, каждая нить может занимать немалый размер памяти, даже если она при этом ничего не делает. В итоге, использование большого количества нитей может снизить производительность.

Служба `select` работает иначе. При использовании `select` вы регистрируете множество входных потоков в объекте `Selector`. Когда происходит работа ввода/вывода с любым из потоков, `Selector` оповестит вас об этом. Таким образом, становится возможно чтение из большого количества источников в одной нити. Заметьте также, что `Selector`'ы не просто помогают вам читать данные, они могут также ожидать входящие сетевые соединения и записывать данные через медленные каналы.

### **3. Пример приложения**

Для демонстрации использования службы `select`, мы создадим простой шифрующий сервер. Эта программа получает данные от клиента, шифрует их, и посылает обратно зашифрованные данные. Служба `select` используется для принятия входящих соединений и чтения поступающих данных из существующих соединений.

Так как данная статья не о шифровании, шифрование, используемое в нашей программе, будет тривиальным. Мы просто применим фильтр `rot13` к данным и пошлём их обратно. `rot13` это простой фильтр, который циклически сдвигает каждый буквенный символ на 13 позиций, превращая *a* в *n*, *b* в *o*, и так далее. Буквы в конце алфавита зацикливаются на начало, так *w* становится *j*.

Не хотелось бы слишком подробно останавливаться на клиенте, взаимодействующем с данным сервером. Вы можете использовать телнет, имеющийся в вашей системе, для соединения с сервером. Также будет предоставлена тестовая программа, которая нагружает сервер множеством соединений, что позволит вам увидеть, как он функционирует под нагрузкой.

### **4. Использование Selector**

Давайте рассмотрим, как используется `Selector`. В примере ниже, `Selector`

## Использование *select* для высокоскоростного сетевого взаимодействия

используется для двух задач: принятия входящих соединений и получения данных из существующих соединений.

### Замечание:

Приведённый фрагмент кода сокращён для экономии места; вы можете скачать полный исходный код из [Ресурсов](#).

### 4.1. Ожидание входящих соединений

Сначала, мы создаём объект `Selector`. `Selector` является основным объектом в данном процессе; каждое соединение, которое ожидается, в любом случае, должно быть зарегистрировано в данном объекте. Статический метод `Selector.open()` создаёт `Selector`:

```
Selector selector = Selector.open();
```

При создании клиент-серверной системы, мы должны прослушивать `ServerSocketChannel`. Необходимо сконфигурировать его как *nonblocking*, что бы можно было использовать его совместно с `Selector`:

```
ServerSocketChannel ssc = ...
ssc.configureBlocking( false );
ssc.register( selector, SelectionKey.OP_ACCEPT );
```

Параметр `SelectionKey.OP_ACCEPT` сообщает `Selector`'у, что мы хотим ожидать только входящие соединения, а не обычные данные. Так как серверные сокеты не получают обычных данных, то это то, что нам надо.

### 4.2. Основной цикл

Теперь, после того, как мы зарегистрировали наш `Selector`, давайте заставим его работать. Мы будем использовать метод `select()` `Selector`'а и поместим его внутрь бесконечного цикла, чтобы повторно ожидать новых активностей:

```
while (true) {
    // Проверяем, если ли какие-либо активности -
    // входящие соединения или входящие данные в
    // существующем соединении.
    int num = selector.select();
```

## Использование *select* для высокоскоростного сетевого взаимодействия

```
// Если никаких активностей нет, выходим из цикла
// и снова ждём.
if (num == 0) {
    continue;
}

// Получим ключи, соответствующие активности,
// которые могут быть распознаны и обработаны один за другим.
Set keys = selector.selectedKeys();

Iterator it = keys.iterator();

while (it.hasNext()) {
    // Получим ключ, представляющий один из битов
    // активности ввода/вывода.
    SelectionKey key = (SelectionKey)it.next();

    // ... работаем с SelectionKey ...
}

// Удаляем выбранные ключи, поскольку уже отработали с ними.
keys.clear();
}
```

Заметьте, что эта схема похожа на *событийный цикл (event loop)*, используемый при создании графического пользовательского интерфейса. Такая аналогия возникает из-за того, что событийный цикл используется, когда входное событие может придти от различных объектов и мы заранее не знаем, когда придёт первое из них.

Внутри цикла, метод `select()` возвращает количество каналов, имеющих активность ввода/вывода. Если он возвращает 0, мы просто возвращаемся в начало цикла и снова ждём. Если нет, то мы должны обработать активность ввода/вывода.

Активность представляется одним или более объектом `SelectionKey`. `SelectionKey` представляет регистрацию одного `Channel` в одном `Selector`. Когда `Selector` определяет, что какой-то `Channel` имеет некую активность, он возвращает `SelectionKey`, соответствующий данному `Channel`.

### 4.3. Получение соединения

Когда `SelectionKey` представляет некоторую активность, вы должны определить тип этой активности. В этом месте кода мы зарегистрировали только один `Channel` — `ServerSocketChannel`, поэтому нам нужно работать с входящими соединениями в данном `Channel`.

Используйте метод `readyOps()` объекта типа `SelectionKey` для определения типа активности. Этот метод возвращает битовую маску, которая предоставляет тип (или типы) входящих событий, произошедших для данного `Channel`. Проверьте полученный результат, содержит ли он `OP_ACCEPT` бит, который представляет входящее соединение:

```
if ((key.readyOps() & SelectionKey.OP_ACCEPT) ==
    SelectionKey.OP_ACCEPT) {

    // Принимаем входящее соединение
    Socket s = serverSocket.accept();

    // ... работаем с входящим соединением ...
}
```

Если мы действительно имеем входящее соединение, то используем традиционный блокирующий вызов `accept()`, для получения соединения. Вызов `accept()` не блокирует выполнение, потому что `Selector` уже сообщил нам о входящем соединении, ожидающем обработки.

Теперь, когда соединение установлено, используем его для получения данных.

### 4.4. Ожидание входящих данных

После принятия соединения, мы будем ожидать данных, передаваемых по нему. Так же, как мы зарегистрировали серверный сокет для ожидания входящих соединений, мы регистрируем только что соединённый сокет для ожидания входящих данных. Мы конфигурируем новый сокет неблокирующим, так же, как делали это для серверного сокета:

```
// Необходимо сделать его неблокирующим,
// чтобы использовать Selector для него.
SocketChannel sc = socket.getChannel();
sc.configureBlocking( false );
```

```
// Регистрируем его в Selector для чтения.  
sc.register( selector, SelectionKey.OP_READ );
```

Мы установили Selector для ожидания активности типа OP\_READ, а не для OP\_ACCEPT, что означает, что мы ожидаем входящие данные, а не соединения.

## 4.5. Возвращаемся в начало

Два сокета зарегистрированы в Selector: серверный сокет и обычный сокет. В начале цикла мы опять вызываем select:

```
int num = selector.select();
```

Теперь мы получим уведомление, если любой из сокетов будет иметь активность; то есть, если серверный сокет примет другое соединение, или обычный сокет получит данные, или если одновременно произойдет и то и другое. Когда подключаются другие соединения, они тоже будут зарегистрированы в Selector.

## 4.6. Входящие данные

Поскольку минимум один обычный сокет зарегистрирован, мы можем получить по нему некоторые данные. Вы можете определить это событие, когда вы получаете ключ с установленным битом OP\_READ в битовой маске сокета, получаемой вызовом readyOps():

```
} else if ((key.readyOps() & SelectionKey.OP_READ) ==  
    SelectionKey.OP_READ) {  
  
    SocketChannel sc = (SocketChannel)key.channel();  
    processInput( sc );  
  
    // ...  
}
```

Когда это происходит, мы передаём сокет, а вернее SocketChannel сокета в шифрующую процедуру. Шифрующая процедура, которая может быть найдена в [исходном коде](#), просто шифрует переданные данные и возвращает зашифрованные данные клиенту.

#### 4.7. Конечный результат

Это серия шагов, но все они выполняются в рамках парадигмы *select*. Все входящие источники зарегистрированы в *Selector*. Внутри бесконечного цикла вызывается метод *select()* объекта *Selector*. Каждый раз он возвращает, сколько входных источников имели активность. Мы обрабатываем каждое входящее событие и повторяем цикл. Рисунок 3 иллюстрирует этот процесс.

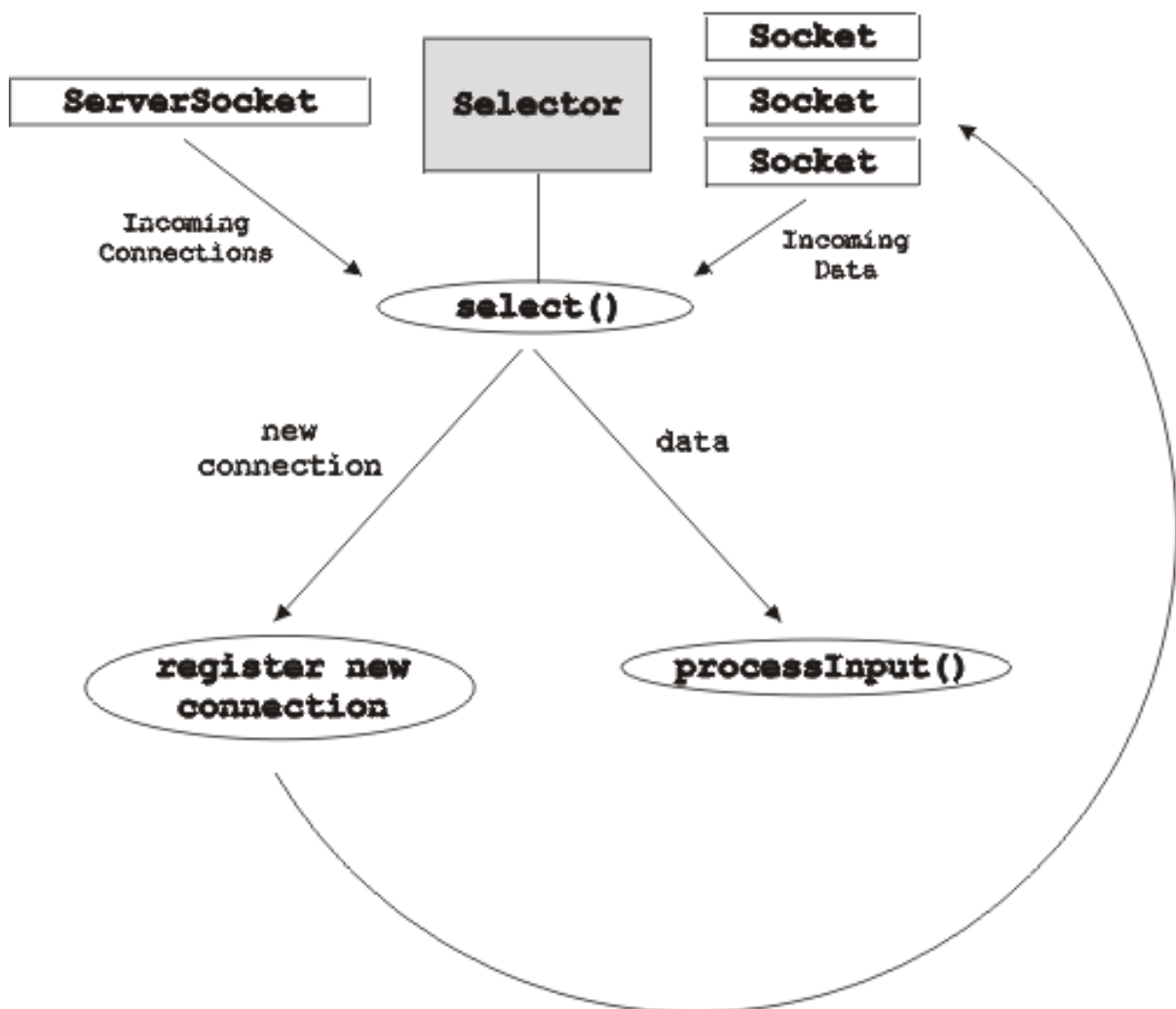


Рисунок 3. Полный цикл *select*.

## 5. Законченный сервер

Код, рассмотренный выше только часть сервера. Полный [исходный код](#) содержит законченный основной цикл, а также код для метода `processInput`. Для тестирования сервера, сначала запустите его из командной строки:

```
java Server [номер порта]
```

Затем, используйте телнет для соединения с сервером. Если вы запускаете телнет на той же машине, что и сервер, вы можете использовать `localhost` или `127.0.0.1` в качестве имени машины для подключения. Укажите номер порта, указанный в командной строке ранее.

[Исходный код](#) также содержит `Client.java`, являющийся программой для тестирования сервера. Она запускает множество нитей выполнения, каждая из которых посылает данные серверу и получает ответ. Каждая нить потом приостанавливается и посылает данные снова, в бесконечном цикле. Вы можете запустить его следующим образом:

```
java Client [имя машины] [номер порта] [количество нитей]
```

## 6. Заключение

Как было рассмотрено выше, *select* модель ввода/вывода по сути является управляемой событиями. Все ваши входные источники регистрируются в едином объекте `Selector`, который ожидает активности любого из входных источников. Данная модель отличается от поточной модели, но тоже целостная модель. Фактически, при общем рассмотрении предмета, более корректно говорить о поточной модели как о слое, который работает поверх *select* модели. На аппаратном уровне, ввод/вывод по сути событийно-управляемый, когда периферийные устройства, такие как сетевые карты, клавиатуры и дисковые драйвера посылают свои данные.

Поточная модель использует буферизацию для сокрытия событийно-ориентированного ввода/вывода за блокирующими вызовами ввода/вывода, которые делают программирование с использованием ввода/вывода более простым. Но когда вам нужна максимальная скорость, пренебрегайте поточным слоем и работайте непосредственно с событиями ввода/вывода сами.

Библиотека NIO предоставляет элегантный интерфейс, основанный на буферах, для select модели. Он полностью совместим со старой поточной моделью; фактически, классы в традиционном пакете `java.io.*` теперь основаны на `java.nio.*`, так что эти две функциональные модели взаимосвязаны.

## 7. Об авторе

Greg Travis — Java программист и технический писатель, живущий в Нью-Йорке. После трёх лет, потраченных в мире высококачественных PC игр, он присоединился к EarthWeb, где он разрабатывал новые технологии на тогда новом языке Java. Начиная с 1997, он был консультантом в различных Web технологиях, специализировался на звуке и графике реального времени. Область его интересов включала алгоритмы оптимизации, проектирование языков программирования, обработку сигналов (с акцентом на музыку) и 3D графику реального времени. Его остальные статьи могут быть найдены на <http://www.panix.com/~mito/articles>. Он является автором *JDK 1.4 Tutorial*, опубликованного Manning Publications.

## 8. Ресурсы

- Скачайте исходный код к данной статье:  
<http://www.javaworld.com/javaworld/jw-04-2003/select/jw-0411-select.zip>
- Страница ссылок J2SE 1.4 содержит документацию по всем классам NIO, обсуждавшимся в данной статье:  
<http://java.sun.com/j2se/1.4/docs/api/>
- Документация по J2SE содержит обзор select или неблокирующей модели:  
<http://java.sun.com/j2se/1.4/docs/api/java/nio/channels/package-summary.html#multiplex>
- Обзор библиотеки NIO от Sun Microsystems:  
<http://java.sun.com/j2se/1.4/docs/guide/nio/>
- Книга Грега Тревиса *JDK 1.4 Tutorial* (Manning Publications, 2002; ISBN: 1930110456) содержит две главы, посвящённые библиотеке NIO, включая подробное обсуждение select модели:  
<http://www.manning.com/travis/>
- "Новые классы ввода/вывода Мага Мерлина", Michael Nygard (*JavaWorld*, Сентябрь 2001):

*Использование select для высокоскоростного сетевого взаимодействия*

<http://www.javaworld.com/javaworld/jw-09-2001/jw-0907-merlin.html>

- Просмотрите **Java 2 Platform, Standard Edition** раздел тематического индекса *JavaWorld*:

[http://www.javaworld.com/channel\\_content/jw-j2se-index.shtml](http://www.javaworld.com/channel_content/jw-j2se-index.shtml)

- Посетите форум JavaWorld:

<http://forums.devworld.com/webx?13@@.ee6b802>

- Подпишитесь на еженедельную новостную рассылку *JavaWorld*:

<http://www.javaworld.com/subscribe>

- Вы можете найти множество статей, связанных с информационными технологиями на дружественном нам сайте [IDG.net](http://www.idg.com)

Reprinted with permission from the April 2003 edition of JavaWorld magazine. Copyright © ITworld.com, Inc., an IDG Communications company.

View the original article at:

[http://www.javaworld.com/javaworld/jw-04-2003/jw-0411-select\\_p.html](http://www.javaworld.com/javaworld/jw-04-2003/jw-0411-select_p.html)

[Перевод на русский © Олег Лапшин, 2003](#)