Работа с файлами

java.io.File (класс)

java.nio.file: Path (интерфейс) и Files (класс)

Создание файла (File):

|  |  |
| --- | --- |
| 1.  2.  3. | boolean success = file.createNewFile();  Files.creatFile([переменная]); |

Создание директории (Files):

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | boolean success = dir.mkdir();  boolean success2 = dir.mkdirs();  Files.createDirectory(dir);  Files.createDirectories(dir); |

Удаление файла/директории (File):

|  |  |
| --- | --- |
| 1. | boolean success = file.delete(); |

Рекурсивный обход для удаления вложенных файлов и поддиректорий.

Удаление файла/директории (Files) (копирование похоже):

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27. | Path directory = Paths.get("[путь]");  Files.walkFileTree(directory,  new SimpleFileVisitor<Path>() {  // класс SimpleFileVisitor<T> implements FileVisitor<T>  @Override  public FileVisitResult visitFile(  Path file,  BasicFileAttributes attrs)  throws IOException {  Files.delete(file);  return FileVisitResult.CONTINUE;  }  @Override  public FileVisitResult  postVisitDirectory(Path dir,  IOExeotion exc)  throws IOException {  if (exc == null) {  Files.delete(dir);  return FileVisitResult.CONTINUE;  } else {  throw exc;  }  }  }); |

Переименовывает файл (File):

|  |  |
| --- | --- |
| 1. | boolean success = file.renameTo(targetFile); |

/\*

Новые классы живут в пакете java.nio.file. Новый API более стройный. Нет булевских флажков, нет дабросайт исключения. Кроме того, новые классы предоставляют более широкий набор возможностей. Можно работать с расширенными атрибутами файлов, например, с правами доступа или с владельцем. Можно работать с символическими ссылками, в частности, создавать их или узнавать куда ссылка ведет. Ну и конечно появились человеческие методы перемещения и копирования файла.

В интерфейсе Path, в отличии от класса File, нет методов доступа к файловой системе. Когда надо залезть на диск, то это делается через и его статические методы. Там есть аналоги всех операций, которые были в классе File (2‑5) и даже больше. Наконец-то, есть человеческие методы copy() (8) и move(), не имеющие ограничений, связанных с нахождением файлов на одной файловой системе. Опционально туда можно передать дополнительные параметры (10). Например, что делать, если целевой файл уже существует.

InputStream и OutputStream

InputStream представляет собой поток байтов, из которого можно читать по одному байту или блоками. Класс OutputStream представляет собой поток байтов, в который можно писать.

byte[] - это просто примитивный массив, содержащий только необработанные данные.

InputStream:

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27. | package java.io;  public abcstract class InputStream  implements Closeable {  public abstract int read()  throws IOExeption;  public int read(byte b[])  throws IOExeption {  return read(b, 0, b.length);  }  public int read(byte b[], int off,  int len) throws IOExeption {  // ...  }  public long skip(long n)  throws IOExeption {  // ...  }  public void close() throws IOExeprtion {}  // ...  } |

OutputStream:

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26. | package java.io;  public abstract class OutputStream  implements Closeable, Flushable {  public abstract void write(int b)  throws IOExeption;  public void write(byte b[])  throws IOExeption {  write(b, 0, b.length);  }  public void write(byte b[], int off,  int len) throws IOExeption {  // ...  }  public void flush() throws IOExeption {  // ...  } // сбрасывает возможные промежуточные буфера, где конкретная реализация OutputSream может накапливать данные перед передачей операционной системе  public void close() throws IOExeprtion {}  // ...  } |

Копирование:

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8. | int totalBytesWritten = 0;  byte[] buf = new byte[1024];  int blockSize;  while ((blockSize = inputStream.read(buf))  > 0) {  outputStrem.write(buf, 0, blockSize);  totalBytesWritten += blockSize;  } |

Наследники InputStream и OutputStream

Работа с файлами:

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16. | InputStream inputStream =  new FileInputStream(  new File("[путь]"));  OutputStream outputStream =  new FileOutputStream(  new File("[путь]"));  // экземпляры класса Path  InputStream inputStream =  Files.newInputStream(  Paths.get("in.txt"));  OutputStream outputStream =  Files.newOutputStream(  Paths.get("out.txt")); |

Обмен данными по сети (Socket — сетевое соединение):

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17. | try (Socket socket = new Socket(  "ya.ru", 80)) {  OutputStream outputStream =  socket.getOutputStream();  outputStream.write(  "GET / HTTP/1.0\r\n\r\n".  getBytes());  outputStream.flush();  InputStream inputStream =  socket.getInputStream();  int read = inputStream.read();  while (read >= 0) {  System.out.println((char) read);  read = inputStream.read();  }  } |

Работа с памятью виртуальной машины:

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8. | byte[] data = {1, 2, 3, 4, 5};  InputStream inputStream =  new ByteArrayInputStream(data);  ByteArrayOutputStream outputStream =  new ByteArrayOutputStream();  // ...  byte[] result = outputStream.toByteArray(); |

Обертки InputStream и OutputStream

Чтение и запись примитивных типов и строк.

Класс DataOutputStream (writeInt()) и writeUTF()):

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22. | package java.io;  public class DataOutputStream  extends FilterOutputStream  implements DataOutput {  public DataOutputStream(  OutputStream out) {  // ...  }  public final void writeInt(int v)  throws IOExeption {  out.write((v >>> 24) & 0xFF);  out.write((v >>> 16) & 0xFF);  out.write((v >>> 8) & 0xFF);  out.write((v >>> 0) & 0xFF);  intCount(4);  }  // ...  } |

Класс DataInputStream (readInt()):

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25. | package java.io;  public class DataInputStream  extends FilterInputStream  implements DataInput {  public DataInputStream(  InputStream in) {  // ...  }  public final int readInt()  throws IOExeption {  int ch1 = in.read();  int ch2 = in.read();  int ch3 = in.read();  int ch4 = in.read();  if ((ch1 | ch2 | ch3 | ch4) < 0) {  throw new EOFExeption();  return ((ch1 << 24) + (ch2 << 16) +  (ch3 << 8) + (ch4 << 0));  }  // ...  } |

Сжатие (класс DeflaterOutputStream) и распаковка (класс InflaterInputStream) данных (алгоритм Deflater):

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21. | byte[] originalData = {1, 2, 3, 4, 5};  ByteArrayOutputSream os =  new ByteArrayOutputStream();  try (OutputStream dos =  new DeflaterOutputStream(os)) {  dos.write(otiginalData);  }  byte[] defaultData = os.toByteArray();  try (InflaterInputStream iis =  new InflaterInputStream(  new ByteArrayInputStream(  dafaultedData))) {  int read = iis.read();  while (read >= 0) {  System.out.prinln("%02x", read);  read = iis.read();  }  } |

Reader и Writer

Для работы с текстовыми данными Reader (ввод данных java.io) и Writer (вывод данных java.io).

Reader:

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27.  28. | package java.io;  public abstract class Reader  implements Readable, Closeable {  public int read() throws IOExeption {  // ...  }  public int read(char cbuf[])  throws IOExeption {  return read(cbuf, 0, cbuf.length);  }  public abstract int read(char cbuf[],  int off, int len)  throws IOExeption;  public long skip(long n)  throws IOExeption {  // ...  }    public abstract void close()  throws IOExeption;  // ...  } |

Writer:

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27.  28. | package java.io;  public abstract class Writer  implements Appendable, Closeable,  Flushable {  public void write(int c)  throws IOExeption {  // ...  }  public void write(char cbuf[])  throws IOExeption {  write(sbuf, 0, sbuf.length);  }  public abstract void write(char sbuf[],  int off, int len)  throws IOExeption;  public abstract void flush()  throws IOExeption;  public abstract void close()  throws IOExeprtion;  // ...  } |

Наследники Reader и Writer

Завернуть в InputStreamReader или OutputStreamWriter:

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6. | Reader reader = new InputStreamReader(  inputStream, “Utf-8”);  Charset charset = StandardCharsets.UTF\_8;  Writer writer = new OutputStreamWriter(  outputStream, charset); |

Чтение и запись текстовых файлов без кодировки и с ней:

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9. | Reader reader = new FileReader("in.txt");  Writer writer = new FileWriter("out.txt");  Reader reader2 = new InputStreamReader(  new FileInputStream("in.txt"),  StandardCharsets.UTF\_8);  Writer writer2 = new OutputStreamWriter(  new FileOutputStream("out.txt"),  StandardCharsets.UTF\_8); |

Чтение/запись символов из (в) массива или строки:

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14. | Reader reader = new CharArrayReader(  new char[] {‘a’, ‘b’, ‘c’});  Reader reader2 = new StringReader(  “Hellow World!”);  CharArrayWriter writer =  new CharArratyWriter();  writer.write(“Test”);  char[] resultArray = writer.toCharArray();  StringWriter writer2 = new StringWriter();  writer2.write(“Test”);  String relultString = writer2.toString(); |

Обертки Reader и Writer

Копит большие куски данных и сбрасывает их в writer. Бесполезный метод newLine()(выводит разделитель строк):

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13. | try (BufferedWriter writer =  Files.newBufferedWriter(  Paths.get(“out.txt”),  StandartCharsets.UTF\_8)) {  writer.write(“Hello”);  writer.newLine();  }  List<String> lines = Arrays.asList(  “Hellow”, “world”);  Files.write(Paths.get(“out.txt”), lines,  StandartCharsets.UTF\_8); |

Для небольших объемов данных можно воспользоваться удобным методом Files.write(), который принимает список строк и внутри себя делает все необходимое для его записи в выходной текстовый файл.

Класс BufferedReader запрашивает символы у Reader большими блоками и сохраняет в своем внутреннем буфере. Преимущества: положительно сказываться на производительности в случае больших объемов данных и есть удобный метод readLine() (9), считывающий построчно (сам символ, разделяющий строки, не возвращается; дочитали до конца — null).

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15. | package java.io;  public class BufferedReader extends Reader {  public BufferedReader(Reader in) {  // ...  }  public String readLine()  throws IOExeption {  // ...  }  // ...  } |

Удобный метод Files.newBufferedReader() (2), который скрывает внутри себя многоэтажную конструкцию с вложенными стримами. Если файл небольшой, можно сделать еще проще — воспользоваться другим удобным методом Files.readAllLines (13), который вернет строки в виде списка сразу все.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19. | try (BufferedReader reader =  Files.newBufferedReader(  Paths.get(“in.txt”),  StandartCharsets.UTF\_8)) {  String line;  while ((line = reader.readLine()) !=  null) {  // process line  }  }  List<String> lines = Files.readAllLines(  Paths.get(“in.txt”),  StandartCharsets.UTF\_8);  for (String line : lines) {  // process line  } |

Scanner

Класс java.util.Scanner:

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11. | Reader reader = new StringReader(  “abc|true|1,1e3|-42”);  Scanner csanner = new Scanner(reader).  useDelimiter(“\\|”).useLocale(  Locale.forLanguageTag(“ru”));  String token = scanner.next();  boolean bool = scanner.nextBoolean();  double dbl = scanner.nextDouble();  int integer = scanner.nextInt(); |

На основе Reader (есть и другие конструкторы).

Преимущества: парсить все примитивные типы и строки, указать шаблон разделителя и региональные настройки (Locale).

5.3.4 Форматированный вывод

Классы Reader и Writer универсальны, но не слишком удобны для повседневного использования. То есть ввод/вывод значений разных типов требует от программиста написание кода по конвертации в строку или массив символов и обратно. К счастью, такой код уже написан и добавлен в стандартную библиотеку. Форматированный ввод/вывод значений разных типов реализован в Java как надстройка, следующий уровень над Reader и Writer.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27. | package java.io;  public class PrintWriter extends Writer {  public PrintWriter(Writer out) {  // ...  }  public void print(int i) {  // ...  }  public void println(Object obj) {  // ...  }  public PrintWriter ptintf(  String format, Object ... args) {  // ...  }  public boolean checkError() {  // ...  }  // ...  } |

Давайте сначала о выводе. Форматированный вывод обеспечивает класс PrintWriter (3). В нем есть методы print() (9) и println() (13), перегруженные для всех примитивных типов, а также строк и объектов. PrintWriter конвертирует все это в строки и спускает в нижележащий Writer (5: Writer out) для вывода. Объекты преобразуются в строки при помощи вызова метода toString(). А еще здесь есть метод ptintf() (17), знакомый по языку C. Он принимает строку шаблон (18: String format) и набор параметров (18: Object ... args), которые в эту строку будут подставляется. Места подстановки параметров обозначаются специальными последовательностями символов. Например, %s — это место подстановки строки, %d — это место подстановки целого числа, %3f — это место подстановки вещественного числа, которое будет округленно, либо дополнено нулями до 3‑х цифр после запятой. Полное описание формата ищите в Javadoc. Существенное отличие методов PrintWriter состоит в том, что они не бросают IOExeption. Вместо этого они молча устанавливают внутренний флаг ошибки, который можно проверить при помощи метода checkError() (22). С одной стороны, это удобно, потому что не надо возиться с проверяемым IOExeption. С другой стороны, мы так легко можем пропустить ошибку.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27.  28.  29. | package java.io;  public class PrintStream  extends FileterOutputSream  implements Appendable, Closeable {  public PrintStream(OutputStream out) {  // ...  }  public void print(int i) {  // ...  }  public void println(Object obj) {  // ...  }  public PrintWriter ptintf(  String format, Object ... args) {  // ...  }  public boolean checkError() {  // ...  }  // ...  } |

У класса PrintWriter есть близкий родственник класс PrintStream (3). На вид он очень похож, так как предоставляет те же самые методы print (11), println (15) и ptintf (19) и не бросает исключений. Но он оборачивает OutputStream (7: OutputStream out), а не Writer и сам является OutputSream (4: FileterOutputSream). Внутри своих методов print он делает сразу два преобразования. Сначала из примитивных типов и объектов в строку, в потом сразу в байты, которые передаются нижележащему OutputStream для вывода. То есть получается этакий гибрид OutputStream и PrintWriter.

5.3.6 Стандартные потоки

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12. | package java.lang;  public final class System {  public static final InputStream in =  null;  public static final PrintStream out =  null;  public static final PrintStream err =  null;  // ...  } |

В заключение давайте заглянем в класс System и посмотрим, что из себя представляют объекты in, out и err, соответствующие стандартным потоком ввода, вывода и ошибок, при помощи которых программа взаимодействует с внешним миром и с консолью. System.in (5) — это InputStream, поток байт. Если программе на вход подается текстовая информация, то удобно обернуть этот InputStream в InputStreamReader плюс BufferedReader или сразу в Scanner. System.out (7) и System.err (9) — это PrintStream, то есть можно выводить в них как двоичные данные при помощи метода write(), так и текстовые при помощи методов print, println и printf. Чем мы, кстати, и пользовались, начиная с нашей первой программы HelloWorld. Любопытно, что поля объявлены как финальные (5,7,9: final) со значением null. На самом деле они конечно же будут инициализированы какими-то конкретными объектами. Это сделает виртуальная машина и ей final не указ, но не пытайтесь повторить это у себя дома.

Сериализация

Сериализация: взять объект (из памяти виртуальной машины), превратить его в набор байтов (вместе со всеми значениями его полей) и (с этим набором байтов) сделать все что угодно (записать в файл или отправить по сети). А потом этот набор байтов (можно будет прочитать) декодировать и получить (точно) такой же объект.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27.  28.  29.  30.  31.  32.  33.  34.  35.  36.  37.  38.  39.  40.  41.  42.  43.  44.  45.  46.  47.  48.  49.  50.  51.  52.  53.  54.  55.  56.  57.  58.  59.  60.  61.  62.  63.  64.  65.  66.  67.  68.  69.  70.  71.  72.  73.  74.  75.  76.  77.  78.  79.  80.  81.  82.  83.  84.  85.  86.  87.  88.  89. | package org.stepic.java.io;  import java.io.\*;  import java.nio.file.Files;  import java.nio.file.Path;  import java.nio.file.Paths;  import java.nio.time.LocalDate;  public class SerializationDemo {  public static void main(String[] args)  throws Exception {  Client orginalClient = new Client();  originalClient.setId(1);  originalClient.setName(“Chuck  Norris”);  otriginalClient.setBirDate(  LocalDate.of(1940, 3, 10));  Path path = Paths.get(“object.bin”);  try (ObjectOutputStream oos =  new ObjectOutputStream(  Files.newOutputStream(  path))) {  oos.writeObject(originalClient);  }  Client deserializedClient;  try (ObjectIntputStream ois =  new ObjectInputStream(  Files.newInputStream(  path))) {  deseralizedClient = (Client)  ois.readObject();  }  System.out.println(  deserializedClient.getId());  System.out.println(  deserializedClient.getName(  ));  System.out.println(  deserializedClient.  getBirthDate());  System.out.println(  deserializedClient.  getAgeInYears());  }  public static class Client  implements Serializable {  private long id;  private String name;  private LocalDate birthDate;  private transient int ageInYears;  public long getId() { return id; }  public void setId(long id) {  this.id = id;  }  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  public LocalDate getBirthDate() {  return birthDate;  }  public void setBirthDate(  LocalDate birthDate) {  this.birthDate = birthDate;  }  public int getAgeInYears() {  if (ageInYears == 0) {  ageInYears = birthDate.until(  LocalDate.now()).  getYears();  }  return ageInYears;  }  }  } |

1) Пометить ваш класс маркерным интерфейсом Serializable (маркерный — нет ни одного метода). Вот так и Serializable ни одного метода не объявлено. Это просто признак, по которому JVM понимает, что данный объект можно сериализовывать. И эту сериализацию JVM будет делать сама по своим правилам. Писать дополнительный код для этого не требуется. Иногда бывает так, что в нашем объекте есть поля, которые не надо сохранять. Это может быть какой-то кеш или другие данные, которые легко вычисляются по тому, что есть в объекте. Их потеря ничем не грозит. Зато позволит сохранить размер, занимаемый объектом при сериализации. Такие поля примечаются ключевым словом transient (55). JVM будет их игнорировать при записи. Объекты можно записывать в ObjectOutputStream (21), который может построить из любого другого OutputStream (23). Запись осуществляется методом writeObject() (25). Объект сериализуется вместе с другими объектами, на которые он ссылается. Разумеется, они тоже должны быть помечены как Serializable. Иначе при попытке их записи будет брошено исключение. Для чтения десириализации предназначен симметричный класс ObjectIntputStream (29) и его метод readObject() (34). При этом мы получаем объект и все то, на что он ссылался. Конечно, тут все не так просто, есть много нюансов и подводных камней, о которых можно рассказывать очень долго, но давайте остановимся на этом минимально работающем примере и убедимся, что Чаку Норрису никакая сериализация нипочем. Вывод в консоль:

|  |  |
| --- | --- |
| 1.  2.  3.  4. | 1  Chuck Norris  1940-03-10  75 |

Externalizable

Externalizable — расширение интерфейса Serializable.

При реализации интерфейса Externalizable нам необходимо переопределить два метода: writeExternal() и readExternal(). В методе writeExternal() мы указываем какие поля будут сериализованы и как, в readExternal() как их прочитать. При использовании слова transient мы явно указываем, какое поле или поля не нужно сериализовывать. Так же заметим, что во втором способе мы явно создали конструктор по умолчанию, причем публичный. Зачем это сделано? Давайте попробуем запустить код без этого конструктора.

Недостатки Serializable:

1) Производительность.

Во-первых, внутренний механизм Serializable во время работы генерирует большой объем служебной информации и разного рода временных данных.

Во-вторых, работа Serializable основана на использовании Reflection API.

2) Гибкость.

Не управляем процессом сериализации-десериализации при использовании интерфейса Serializable.

3) Безопасность (вытекает из предыдущего)

Что ж, давай наконец посмотрим, как будет выглядеть класс с использованием интерфейса Externalizable.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27. | import java.io.Externalizable;  import java.io.IOException;  import java.io.ObjectInput;  import java.io.ObjectOutput;  public class UserInfo implements Externalizable {  private String firstName;  private String lastName;  private String superSecretInformation;  private static final long serialVersionUID = 1L;  //...конструктор, геттеры, сеттеры, toString()...  @Override  public void writeExternal(ObjectOutput out) throws IOException {  }  @Override  public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {  }  } |

Как видишь, у нас появились существенные изменения!

Главное из них очевидно: при имплементации интерфейса Externalizable ты должен реализовать два обязательных метода — writeExternal() и readExternal().

Как мы и говорили ранее, вся ответственность за сериализацию и десериализацию будет лежать на программисте.

Однако теперь ты можешь решить проблему отсутствия контроля над этим процессом! Весь процесс программируется напрямую тобой, что, конечно, создает гораздо более гибкий механизм.

Кроме того, решается проблема и c безопасностью. Как видишь, у нас в классе есть поле: персональные данные, которые нельзя хранить в незашифрованном виде.

Теперь мы легко можем написать код, соответствующий этому ограничению.

К примеру, добавить в наш класс два простых приватных метода для шифрования и дешифрования секретных данных. Записывать их в файл и вычитывать из файла мы будем именно в зашифрованном виде. А остальные данные будем записывать и считывать как есть :)

Копирование

Cloneable (маркерный интерфейс), метод clone()

Поверхностное — только примитивы, глубокое — еще и объекты.

в

5.4 Продвинутые возможности

5.4.1 Запуск внешних процессов

Мы с вами рассмотрели уже целую кучу классов, отвечающих за ввод/вывод. Однако, как ни странно, это еще не все. В стандартной библиотеке Java есть еще не мало классов, имеющих отношение к вводу/выводу. Хотя они уже, конечно, нужны далеко не в каждой программе и предоставляют какие-то специальные продвинутые возможности. Подробный разговор о таких возможностях выходят за рамки базового курса, но небольшой обзор будет уместен.

Из Java‑программы можно запускать другие процессы на вашем компьютере. Достаточно знать путь у исполняемому файлу на файловой системе и параметр, который ему можно передать для запуска. Зачем это может быть нужно? Ну, например, может стоять задача организовать наблюдение за каким-то серверным процессом и перезапускать его, если он вдруг зависнет или упадет. Ничто не мешает реализовать такое наблюдение в виде Java‑программы.

Или другой пример. До выхода Java 7 с ее новым API доступа к файловой системе java.nio.file нельзя было из Java создать, например, символическую ссылку в файловой системе или получить расширенные атрибуты файла или директории, такие как владельца и права доступа. Реализовывать это приходилось при помощи внешних средств: запуска внешней команды создания симлинка или внешней команды вывода расширенной информации о файле и директории с последующим разбором того, что она вернула. И хотя сейчас в Java 8 благодаря этому API java.nio.file конкретно эти две вещи можно уже делать, но всегда останутся задачи, которые для Java слишком низкоуровневы или платформенно-специфичны. И поэтому, в стандартной библиотеке их не будет. Например, форматирование диска или настройка параметров сетевого интерфейса, или изменения других параметров операционной системы. Это всегда будет требовать выхода за рамки Java. И поэтому время от времени приходится запускать внешние процессы.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27.  28.  29.  30.  31.  32.  33.  34.  35.  36.  37.  38. | package org.stepic.java.io;  import java.io.\*;  import java.lang.ProcessBuilder.Redirect;  public class ProcessDemo {  public static void main(String[] args)  throws Exception {  ProcessBuilder processBuilder =  new ProcessBuilder();  processBuilder.command("ls", "-l").  directory(new File(  "/home/stepic/Vladykin")).  redirectInput(Redirect.from(  newFile("/dev/null"))).  redirectOutput(  Redirect.PIPE).  redirectError(  Redirect.INHERIT);  Process process =  processBuilder.start();  try (BufferedReader reader =  new BufferedReader(  newInputStreamReader(  process.getInputStream()))) {  reader.lines().forEach(  System.out::println);  }  int exitValue = process.waitFor();  if (exitValue != 0) {  System.err.println("Subprocess  terminated abnormally");  }  }  } |

Давайте посмотрим, как это делается. Всю подготовительную работу по запуску внешнего процесса делает класс ProcessBuilder (10). Ему можно указать запускаемую команду (12: command()) с параметрами (12: "ls", "-l"), рабочую директорию (13: new File(“/home/stepic/Vladykin”)), в которой будет запускаться новый процесс, а также ему можно сказать к чему будут подключаться стандартные потоки ввода (15), вывода (17) и ошибок (19) нового процесса. Можно, например, организовать PIPE (18: Redirect.PIPE) между Java процессом и запускаемым процессом. Тогда мы со стороны Java‑программы сможем явно получить этот поток Stream и сможем в него писать или из него читать. Или новый процесс может унаследовать поток Java‑процесса (20: Redirect.INHERIT). И все что тот процесс выведет в свой стдЭрр, попадет в стдЭрр нашего Java‑процесса. Или можно сделать даже так: перенаправить поток в файл или из файла. В данном случае мы направляем на вход запущенному процессу содержимое файла dev/null (15: Redirect.from(newFile("/dev/null"))). В Unix системах это такой специальный виртуальный файл, в который можно выводить все что угодно, и оно там исчезает как в черной дыре. А при попытке чтении их этого файла он оказывается пустым. После того, как параметры запуска настроены, можно стартовать новый процесс методом start() (23). В результате мы получим экземпляр класса Process (22). Если какие‑то из потоков ввода/вывода настроены как PIPE (18), то мы можем взять у процесса этот поток (27: getInputStream()) и поработать с ним. Надо только помнить: то, что для запускаемого процесса является потоком вывода, со стороны Java‑процесса является потоком ввода и наоборот. В данном примере мы принимаем все (28), что запущенный процесс выводит, и записываем это в поток стандартного вывода нашего Java процесса, то есть, по сути, руками реализуем режим INHERIT (20). При помощи метода waitFor() (32) можно дождаться завершение дочернего процесса и получить его код возврата. Ноль соответствует успешному завершению (33), а любые другие значения — какой-то ошибки. Ну и давайте запусти, чтобы добиться что оно работает.

5.4.2 Классы java.nio

Вторая группа классов ввода/вывода, которые могут быть иногда полезны, это классы пакета java.nio и java.nio.chanels. Эти классы более низкоуровневые, чем знакомые нам классы пакета java.io, поэтому они не так удобны для повседневных задач вроде чтения текстовых файлов. Однако в определенных ситуациях их использование позволяет достичь лучшей производительности и большей масштабируемости.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27.  28.  29.  30.  31.  32.  33.  34.  35.  36.  37.  38.  39.  40.  41.  42.  43.  44.  45.  46.  47. | package org.stepic.java.io;  import java.io.IOExeption;  import java.nio.ByteBuffer;  import java.nio.chanels;  import java.nio.file.Path;  import java.nio.file.Paths;  public class NioDemo {  public static void main(String[] args)  throws IOException {  Path path Paths.get(“  src/org/stepic/java/io/  NoDemo.java”);  try (ReadableByteChannel in =  FileChannel.open(path);  WritableByteChanel out =  Chanels.newChanel(  System.out)) {  ByteBuffer buffer =  ByteBuffer.allocate(  1024);  while (in.read(buffer >= 0 ||  buffer.position() !=  0) {  buffer.flip();  out.wtite(buffer);  buffer.compact();  }  }  // try (InputStream in =  // Files.newInputStream(path);  // OutputStream out =  // System.out) {  //  // byte[] buffer = new byte[1024];  // int bytesRead;  // while ((bytesRead = in.read(  // buffer)) > 0) {  // out.write(buffer, 0,  // bytesRead);  // }  // }  }  } |

Основные сущности этого набора классов это Buffer (22), аналог массива, и Channel (18), канал, аналог Input- OutputStream. Ключевое свойство nio — неблокирующий ввод/вывод. Чтобы понять, что это такое, давайте вспомним знакомые нам классы InputStream и OutputStream (33-45). Когда мы вызываем на InputStream метод read() (40), любой из перегруженных вариантов, то программа как-бы становится на паузу, блокируется, в ожидании, пока из потока не будет прочитан хотя бы один байт. Если мы читаем, допустим, данные, которые нам посылают по сети с другого компьютера, да еще по не очень надежному каналу, то возможно ждать придется долго. Аналогично с методом write() (42). Он не вернет управление, пока все не запишет. Никакого способа прервать этого ожидания нет. И это называется блокирующий ввод/вывод. В противоположность ему неблокирующий ввод/вывод, предоставляемый классом из nio, — это возможность сразу прочитать или записать столько данных, сколько можно и ничего не ждать. Если в канале нет ни одного доступного для чтения байта или канал не готов ничего принять, значит метод сразу возвращает управление, ничего не делая и не ожидая. Располагая возможностью неблокирующего ввода/вывода можно, например, из одного треда работать сразу с многими каналами. Мы просто будем опрашивать их состояния и обрабатывать данные по мере поступления, то есть можно скажем написать программу сервер, которая в одном треде будет обслуживать сотни клиентов. При использовании классического java.io на каждого клиента потребовался бы свой тред.

Еще раз повторюсь, что данные классы представляют более низкоуровневый интерфейс ввода/вывода, который с одной стороны является более гибким и потенциально более производительным, но с другой стороны значительно менее удобным для повседневных задач. То есть прибегать к этим классам стоит в тех редких случаях, когда вы реально упираетесь в неэффективность блокирующего ввода/вывода.