**Phần 1: Giới thiệu Regular Expression**

Regular expression (tạm dịch: biểu thức có quy tắc) là một mẫu chuỗi mà dựa vào đó chúng ta có thể dùng để tìm kiếm dữ liệu, trích xuất một chuỗi con từ chuỗi ban đầu, kiểm tra một chuỗi tuân thủ theo một định dạng được quy định trước…..

Cho dù các bạn xây dựng loại ứng dụng gì, thì ở một thời điểm nào đó, việc sử dụng regular expression là khó tránh khỏi.

Các ứng dụng phổ biến của regular expression:

Kiểm tra định dạng số điện thoại: ví dụ mã quốc gia – mã vùng – số điện thoại

Kiểm tra định dạng email: ví dụ [email@address.com.vn](mailto:email@address.com.vn)

Kiểm tra định dạng ID: ví dụ ID98987, hoặc ISBN-123-12345

Kiểm tra một chuỗi có chứa hoặc không chứa một số kí tự nào đó hay không

Kiểm tra một password là mạnh hay yếu

Và đặc biệt là các search engine (google, bing, yahoo) cũng sử dụng regular expression để lọc kết quả tốt hơn.

Không chỉ trong Java (từ Java 1.5+), mà trong các ngôn ngữ lập trình khác, hầu hết đều hỗ trợ Regular Expression.

Regular Expression quy định một tập các kí hiệu và cú pháp, vì vậy chúng ta phải nắm các kí hiệu và cú pháp này trước khi có thể sử dụng.

**Các kí hiệu thường dùng(còn gọi là character classes):**

|  |  |
| --- | --- |
| . | Đại diện cho kí tự bất kỳ |
| \d | Đại diện cho một số từ 0 - 9 ; cú pháp tương tự [0-9] |
| \D | Đại diện cho một kí tự KHÔNG phải là số từ 0 – 9 ; cú pháp tương tự [^0-9] |
| \s | Đại diện cho một kí tự khoảng trắng (whitespace character), bao gồm \t, \n, \f, \r |
| \S | Đại diện cho một kí tự KHÔNG phải là khoảng trắng; cú pháp tương tự [^\s] |
| \w | Đại diện cho một kí tự từ (word character) (chữ hoa hoặc thường) từ a-z, 0-9, \_; cú pháp tương tự [a-zA-Z\_0-9] |
| \W | Đại diện cho một kí tự KHÔNG phải kí tự từ (non-word character); cú pháp tương tự [^\w] |

**Các kí hiệu liên quan về số lượng (Quantifiers)**

|  |  |
| --- | --- |
| \* | Trùng khớp 0 hoặc nhiều lần. ví dụ:  \d\*: chuỗi trùng có thể có 0 hoặc nhiều số từ 0 – 9 .  \w\*: chuỗi trùng có thể có 0 hoặc nhiều kí tự từ (word character) từ a to z, hoặc số từ 0 - 9 |
| + | Trùng khớp 1 hoặc nhiều lần. ví dụ:  \d+: chuỗi trùng phải có ít nhất 1 số từ 0 - 9  \w+: chuỗi trùng phải chứa ít nhất một kí tự từ (word character) từ a – z (hoa hoặc thường) hoặc số 0 – 9 |
| ? | Trùng khớp 0 hoặc 1 lần. ví dụ:  \d?: chuỗi trùng có thể có 0 hoặc 1 số từ 0 – 9  \w?: chuỗi trúng có thể chứa 0 hoặc 1 kí tự từ (word character) từ a – z, hoặc số từ 0 – 9 |
| {n} | Trùng khớp đúng **n** lần. ví dụ:  \d{5}: chuỗi trùng phải chứa đúng 5 số từ 0 – 9  \w{3}: chuỗi trùng phải chứa đúng 3 kí tự từ (word character) từ a – z, 0 – 9 |
| {n,} | Trùng khớp ít nhất **n** lần. ví dụ:  \d{3,}: chuỗi trùng phải chứa ít nhất 3 số từ 0 – 9  \w{4,}: chuỗi trùng phải chứa ít nhất 4 kí tự từ (word character) từ a – z , 0 – 9 |
| {n,m} | Trùng khớp tối thiểu **n** lần, tối đa **m** lần. ví dụ:  \d{3,5}: chuỗi trùng chứ ít nhất 3 số từ 0 – 9, nhưng không quá 5 số  \w{1,7}: chuỗi trùng chứa ít nhất 1 kí tự từ (word character) từ a – z, 0 – 9, nhưng không quá 7 kí tự từ (word character) . |

Hầu hết các ngôn ngữ lập trình ngày nay từ Java, C# cho đến JavaScript đều hỗ trợ Regular Expression. Mặc dù các ngôn ngữ lập trình khác nhau có thể có một số hỗ trợ khác nhau về kí hiệu và cú pháp, nhưng phần lớn là giống nhau về các tập kí kiệu như trên.

>> Xem tiếp Phần 2

**Phần 2: Kiểm tra tính hợp lệ của dữ liệu – các kiểm tra cơ bản**

Chú ý: các đoạn code sau đây được thực thi trên JDK 8

Trong Phần 1, tôi đã giới thiệu với các bạn về regular expression và các tập kí hiệu cơ bản, trong phần này, chúng ta sẽ ứng dụng regular expression vào kiểm tra một số dữ liệu cơ bản như sau:

Kiểm tra dữ liệu người dùng nhập vào có phải là một số nguyên (integer) hay không

Kiểm tra một số nguyên với số lượng kí số cố định. Ví dụ: phải đủ 4 số như năm 2015, 1980

Kiểm tra một số nguyên với số lượng kí số tối thiểu và tối đa. Ví dụ: nhập tuổi: tối thiểu là 2 số (trên 10 tuổi), đố đa là 3 số (<= 100 tuổi)

Kiểm tra một chuỗi bắt đầu bằng một tập kí tự hoặc số. Ví dụ ID1234, ISBN12312345

Kiểm tra một chuỗi không chứa các kí tự đặc biệt như !, @, # $,….

**Trường hợp 1: Kiểm tra một số nguyên**

Đây là đoạn chương trình chúng ta có thể dùng để kiểm tra dữ liệu người dùng nhập vào có phải là một số nguyên dương hay không:

|  |
| --- |
| import java.util.Scanner;  public class Demo {  public static void main(String[] args) {  boolean flag;  Scanner sc = new Scanner(System.in);  do {  // must be a digit from 0 - 9  String digit = "\\d";  System.out.print("Input an integer: ");  String input = sc.next();  flag = input.matches(digit);  if (!flag) System.out.println("You must enter a number!");  } while (!flag);  System.out.println("Valid data");  }  } |

Trong chương trình này, chúng ta sẽ dùng một vòng lặp do..while để yêu cầu người dùng nhập vào một giá trị. Lý do dùng do..while là nếu người dùng nhập sai (không phải là số nguyên dương) thì chúng ta yêu cầu người dùng nhập lại. Do vậy chúng ta cần một biến flag kiểu boolean để giữ cho vòng lặp tiếp tục chạy hoặc thoát ra nếu người dùng nhập đúng dữ liệu.

Chúng ta dùng lớp Scanner để lấy dữ liệu người dùng nhập vào, do đó các bạn lưu ý phải import class Scanner từ package java.util.

Do yêu cầu là người dùng phải nhập vào là một số nguyên, chúng ta định nghĩa mẫu là:

String digit = “\\d”;

Chú ý chúng ta phải dùng 2 dấu \\, vì kí hiệu quy định của regular expression là \d, nhưng dấu \ là ký tự đặc biệt trong Java, do vậy chúng ta phải dùng thêm một dấu \ nữa để trình biên dịch của Java bỏ qua dấu này. Nếu không chúng ta sẽ có lỗi biên dịch

Trong Phần trước, tôi đã giới thiệu là \d đại diện cho một số nguyên từ 0 – 9. Do vậy, chúng ta chỉ cần mẫu này là đủ để kiểm tra một giá trị có phải là một số nguyên hay không.

Sau khi người dùng nhập dữ liệu vào, chúng ta gọi phương thức next() của lớp Scanner để lấy dự liệu vào và lưu vào biến String input;

Chú ý ở đây chúng ta không dùng phương thức nextInt() để lấy dữ liệu, vì nếu người dùng nhập vào một ký tự thì sẽ có exception làm cho chương trình bị crashed

Lớp String trong Java hỗ trợ một phương thức là matches(). Phương thức này nhận vào tham số là một mẫu cần kiểm tra, và trả về giá trị true hoặc false tương ứng với kết quả kiểm tra.

Dòng code:

flag = input.matches(digit);

Có nghĩa là phương thức **matches**() sẽ kiểm tra giá trị của biến input xem có khớp với mẫu qui định của biến digit hay không: nếu khớp, tức là một số nguyên, thì trả về true và lưu vào biến flag; ngược lại, tức là giá trị của input không phải là một số, thì sẽ trả về false.

Tiếp theo chúng ta chỉ cần kiểm tra nếu giá trị biến flag là false thì chúng ta in ra câu thông báo: “You must enter a number!”

Và do biến flag lúc này đang là false, và chúng ta muốn người dùng nhập lại giá trị khác, nghĩa là vòng lặp do..while phải tiếp tục chạy, do vậy chúng ta phải đảo giá trị biến flag thành true bằng cách sử dụng toán tử NOT (!), vì điều kiện để vòng lặp tiếp tục chạy là điều kiện lặp phải là true.

Còn nếu trường hợp người dùng nhập đúng thì giá trị của biến flag sẽ là true, trong điều kiện while sẽ bị đảo lại là false, vì vậy vòng lặp do while cũng sẽ dừng lại.

Chạy chương trình, chúng ta có kết quả như sau:

|  |
| --- |
| Input an integer: a  You must enter a number!  Input an integer: abc  You must enter a number!  Input an integer: 1a  You must enter a number!  Input an integer: 3  Valid data |

Nếu người dùng không nhập 1 số nguyên dương thì chương trình sẽ tiếp tục hỏi cho đến khi nào nhập đúng thì thôi.

Chạy lại chương trình:

|  |
| --- |
| Input an integer: -1  You must enter a number!  Input an integer: 12  You must enter a number!  Input an integer: |

Chú ý nếu người dùng nhập vào là số âm, ví dụ -1, thì cũng không hợp lệ; hoặc nhập vào 2 chữ số, ví dụ 12, thì cũng không hợp lệ, vì kí hiệu [\\d](file:///\\d) chỉ chấp nhận một số nguyên từ 0 đến 9 mà thôi.

Muốn nhập nhiều hơn một số thì chúng ta phải thay đổi mẫu.

**Trường hợp 2: Kiểm tra một số nguyên với số lượng kí số cố định**

Trường hợp này tôi muốn người dùng nhập vào phải một số nguyên, tuy nhiên phải đúng số lượng kí số.

Ví dụ tôi muốn người dùng nhập vào năm và phải đủ 4 số như 2013, 2015, 1998….

Chúng ta có thể dùng đoạn chương trình sau:

|  |
| --- |
| import java.util.Scanner;  public class Demo {  public static void main(String[] args) {  boolean flag;  Scanner sc = new Scanner(System.in);  do {  String yearPattern = "\\d{4}";  System.out.print("Input a year [4 digits]: ");  String input = sc.next();  flag = input.matches(yearPattern);  if (!flag) System.out.println("Invalid data!");  } while (!flag);  System.out.println("Valid data");  }  } |

Chương trình này cũng giống như ví dụ trước chúng ta vừa xem. Tôi chỉ thay đổi mẫu và lưu vào biến:

String yearPattern = "\\d{4}";

Chú ý, chúng ta đặt số 4 vào cặp dấu ngoặc nhọn theo sau kí tự [\\d](file:///\\d); và không có khoảng trắng nào.

Kí hiệu \d có nghĩa là người dùng phải nhập vào một số nguyên từ 0 đến 9 như các bạn đã biết

Số 4 trong ngoặc nhọn có nghĩa là người dùng phải nhập đủ 4 số, không ít hơn, không nhiều hơn. Các bạn có thể thay bằng một số mong muốn khác của mình.

Các đoạn code còn lại không có gì khác so với ví dụ trước nên tôi không giải thích lại.

Chạy chương trình, chúng ta được kết quả như sau:

|  |
| --- |
| Input a year [4 digits]: 12  Invalid data!  Input a year [4 digits]: 123  Invalid data!  Input a year [4 digits]: fgd  Invalid data!  Input a year [4 digits]: 2015  Valid data |

Số12: không hợp lệ vì chỉ có 2 chữ số

Số 123: không hợp lệ vì chỉ có 3 chữ số

Chuỗi “fgd”: không hợp lệ vì không phải là số

Số 2015: hợp lệ vì đủ 4 chữ số

**Trường hợp 3: Kiểm tra một số nguyên với số lượng kí số tối thiểu và tối đa.**

Trường hợp này chúng ta muốn người dùng nhập vào một số nhưng có thể linh hoạt nhập tối thiểu hoặc tối đa bao nhiêu kí số.

Ví dụ: nhập tuổi: tối thiểu là 2 số (trên 10 tuổi), đố đa là 3 số (<= 100 tuổi)

|  |
| --- |
| import java.util.Scanner;  public class Demo {  public static void main(String[] args) {  boolean flag;  Scanner sc = new Scanner(System.in);  do {  String agePattern = "\\d{2,3}";  System.out.print("Input your age: ");  String input = sc.next();  flag = input.matches(agePattern);  if (!flag) System.out.println("Invalid data!");  } while (!flag);  System.out.println("Valid data");  }  } |

Tôi định nghĩa mẫu như sau:

String agePattern = "\\d{2,3}";

Chúng ta chỉ cần thêm dấy phẩy (,) và theo sau là 1 số chỉ định số lượng kí số tối đa. Còn số phía trước dấu phẩy (,) chỉ định kí số tối thiểu mà người dùng phải nhập.

Trong ví dụ này thì người dùng phải nhập vào 1 số với tối thiểu là 2 chữ số, và tối đa là 3 chữ số.

Chạy chương trình, chúng ta được kết quả như sau:

|  |
| --- |
| Input your age: 1  Invalid data!  Input your age: 1001  Invalid data!  Input your age: 33  Valid data |

Số 1: không hợp lệ vì tối thiểu phải là 2 chữ số

Số 1001: không hợp lệ vì tối đa là 3 chữ số

Số 33: hợp lệ vì nằm trong khoảng tối thiểu và tối đa

**Trường hợp 4: Kiểm tra một chuỗi bắt đầu bằng một tập kí tự hoặc số.**

Trong trường hợp này, chúng ta yêu cầu người dùng nhập vào một chuỗi bắt đầu bằng một số kí tự, và theo sau là các chữ số nào đó.

Ví dụ chúng ta yêu cầu người dùng nhập vào số ISBN của một cuốn sách với định dạng sau:

* Bắt đầu bằng các kí tự ISBN, viết hoa
* Theo sau là dấu gạch nối: -
* Theo sau là 5 số nguyên

Ví dụ: ISBN-12345, ISBN-98765

Chúng ta có thể dùng đoạn code như sau:

|  |
| --- |
| import java.util.Scanner;  public class Demo {  public static void main(String[] args) {  boolean flag;  Scanner sc = new Scanner(System.in);  do {  String isbnPattern = "ISBN-\\d{5}";  System.out.print("Input ISBN: ");  String input = sc.next();  flag = input.matches(isbnPattern);  if (!flag) System.out.println("Invalid data!");  } while (!flag);  System.out.println("Valid data");  }  } |

Tôi định nghĩa mẫu cho số ISBN như sau:

|  |
| --- |
| String isbnPattern = "ISBN-\\d{5}"; |

Do yêu cầu là phải bắt đầu bằng ISBN viết hoa, do vậy, chúng ta để nguyên các kí tự ISBN viết hoa vào trong mẫu. Điều này có nghĩa là người dùng phải nhập đúng ISBN, theo sau là dấu gạch nối (-)

Kế tiếp, theo sau phải là 5 số, thì chúng ta dùng \\d{5}.

Chạy chương trình, chúng ta có kết quả như sau

|  |
| --- |
| Input ISBN: ISBN12345  Invalid data!  Input ISBN: isbn-12345  Invalid data!  Input ISBN: ISBN-12345  Valid data |

Chuỗi ISBN12345: không hợp lệ vì không có dấu gạch nối (-) sau chuỗi ISBN

Chuỗi isbn-12345: không hợp lệ vì isbn viết thường, yêu cầu phải viết hoa tất cả kí tự

Chuỗi ISBN-12345: hoàn toàn hợp lệ vì khớp với mẫu quy định

**Trường hợp 5: Kiểm tra một chuỗi không chứa các kí tự đặc biệt như !, @, # $,….**

Đôi khi chúng ta cần kiểm tra một chuỗi có chứa các kí tự đặc biệt hay không. Ví dụ như nhập user name thì thường chỉ chứa kí tự alphabe và số, mà không được có kí tự đặc biệt.

Ví dụ user name hợp lệ: user1234, user9adj

Vi dụ user name không hợp lệ: user@!123

Chúng ta xem chương trình sau:

|  |
| --- |
| import java.util.Scanner;  public class Demo {  public static void main(String[] args) {  boolean flag;  Scanner sc = new Scanner(System.in);  do {  String usernamePattern = "\\w+";  System.out.print("Input user name: ");  String input = sc.next();  flag = input.matches(usernamePattern);  if (!flag) System.out.println("Invalid data!");  } while (!flag);  System.out.println("Valid data");  }  } |

Tôi định nghĩa mẫu như sau:

**String usernamePattern = "\\w+";**

Do yêu cầu là không chứa kí tự đặc biệt, do vậy tôi dùng [\\w](file:///\\w).

Kí tự [\\w](file:///\\w) đại diện cho các kí tự từ a – z, có thể chữ thường hoặc chữ hoa, và số từ 0 đến 9

Theo sau [\\w](file:///\\w) là dấu cộng (+). Dấu cộng (+) có nghĩa là ít nhất phải có 1 kí tự.

Như vậy, mẫu của tôi có nghĩa là: người dùng có thể nhập vào một hoặc nhiều kí tự. Các kí tự này phải từ a – z, có thể hoa hoặc thường, hoặc có thể là số từ 0 – 9 .

Chạy chương trình, chúng ta có kết quả như sau:

|  |
| --- |
| Input user name: user@  Invalid data!  Input user name: 123#user  Invalid data!  Input user name: user1234  Valid data |

Chuỗi user@: không hợp lệ vì có chứa kí tự đặc biệt @

Chuỗi 123#user: không hợp lệ vì có chứa kí tự đặc biệt #

Chuỗi user1234: hoàn toàn hợp lệ vì chỉ chứa kí tự từ a – z, và số từ 0 – 9

>> Xem tiếp Phần 3

**Phần 3: Kiểm tra tính hợp lệ của dữ liệu – Kiểm tra phạm vi các kí tự**

Trong Phần 2, chúng ta đã sử dụng regular expression để kiểm tra tính hợp lệ của dữ liệu trong một số trường hợp cơ bản. Trong phần này, chúng ta sẽ sử dụng một số mẫu phức tạp hơn như sau:

Kiểm tra một số nằm trong một phạm vi quy định trước, không nhất thiết phải từ 0 đến 9

Kiểm tra một kí tự nằm trong một phạm vi các kí tự quy định trước, không nhất thiết phải từ a đến z.

Kiểm tra cả số và kí tự nằm trong các phạm vi được qui định trước

Kiểm tra một chuỗi bao gồm các kí tự được qui định trước

Kiểm tra một chuỗi KHÔNG bao gồm các kí tự được qui định trước

**Trường hợp 1: Kiểm tra một số nằm trong một phạm vi quy định trước**

Ví dụ trong trường hợp này, chúng ta yêu cầu người dùng nhập vào số nhà. Tuy nhiên, số nhà phải tối thiểu là 1 số, tối đa là 3 số, và các số chỉ nằm trong phạm vi từ số 5 đến số 9 mà thôi. Nghĩa là chỉ được nhập các chữ số 5, 6, 7, 8, và 9. Không được nhập các chữ số 0, 1, 2, 3, và 4.

Chúng ta xem chương trình sau:

|  |
| --- |
| import java.util.Scanner;  public class Demo {  public static void main(String[] args) {  boolean flag;  Scanner sc = new Scanner(System.in);  do {  String addressPattern = "[5-9]{1,3}";  System.out.print("Input your house number: ");  String input = sc.next();  flag = input.matches(addressPattern);  if (!flag) System.out.println("Invalid data!");  } while (!flag);  System.out.println("Valid data");  }  } |

Trong chương trình trên, tôi định nghĩa một mẫu như sau:

String addressPattern = "[5-9]{1,3}";

Trong mẫu này, tôi dùng cặp dấu ngoặc vuông ([]), bên trong dấu ngoặc vuông này, tôi chỉ định 5-9. Có nghĩa là chuỗi trùng chỉ được chứa các số từ 5 đến 9 mà thôi.

Theo sau dấu ngoặc vuông là cặp dấu ngoặc nhọn ({}), bên trong dấu ngoặc nhọn này, tôi chỉ định số lượng kí tự phải nhập tối thiểu là 1, và tối đa là 3. Điều này thì các bạn đã biết trong Phần 2.

Như vậy, mẫu của tôi có nghĩa như sau: chuỗi trùng phải có ít nhất 1 kí tự, tố đa 3 kí tự, và các kí tự này phải là các chữ số từ 5 đến 9. Và chú ý, không có khoảng trắng giữa các kí tự này.

Chạy chương trình, chúng ta có kết quả như sau:

|  |
| --- |
| Input your house number: a  Invalid data!  Input your house number: 156  Invalid data!  Input your house number: 78  Valid data |

Kí tự a: không hợp lệ, vì yêu cầu phải là số từ 5 đến 9

Số 156: không hợp lệ, vì có số 1 không nằm trong phạm vi từ 5 đến 9; mặc dù có 5, 6 nằm trong phạm vi này

Số 78: hoàn toàn hợp lệ, vì tất cả chữ số (7, 8) đều nằm trong phạm vi từ 5 đến 9.

**Trường hợp 2: Kiểm tra một kí tự nằm trong một phạm vi các kí tự quy định trước**

Tương tự trong trường hợp qui định phạm vi cho số, chúng ta cũng có thể qui định phạm vi cho kí tự alphabe.

Ví dụ, chúng ta yêu cầu người dùng nhập vào tên đường trong địa chỉ nhà. Các kí tự cho phép nhập là từ A – E, và từ G – N mà thôi. Và tất cả kí tự đều phải là chữ in hoa. Số lượng kí tự cho phép tối thiểu là 2, và tối đa là 8.

Như vậy:

* Các kí tự cho phép nhập từ A – E bao gồm: A, B, C, D, E
* Các kí tự cho phép nhập từ G – N bao gồm: G, H, I, J, K, L, M, N

Chúng ta xem đoạn chương trình sau:

|  |
| --- |
| import java.util.Scanner;  public class Demo {  public static void main(String[] args) {  boolean flag;  Scanner sc = new Scanner(System.in);  do {  String addressPattern = "[A-EG-N]{2,8}";  System.out.print("Input your street name: ");  String input = sc.next();  flag = input.matches(addressPattern);  if (!flag) System.out.println("Invalid data!");  } while (!flag);  System.out.println("Valid data");  }  } |

Trong chương trình trên, tôi định nghĩa mẫu như sau:

String addressPattern = "[A-EG-N]{2,8}";

Để chỉ định một phạm vi các kí tự nào đó, chúng ta vẫn sử dụng dấu ngoặc vuông ([]) như trong ví dụ trước. Tuy nhiên trong trường hợp này, do yêu cầu bài toán, tôi chỉ định 2 phạm vi:

* Phạm vi đầu tiên là từ A đến E
* Phạm vi thứ 2 là từ G đến N

Chú ý không có khoảng trắng giữa các kí tự trong cặp dấu ngoặc vuông.

Và thứ tự các kí tự cũng không quan trọng. Có nghĩa là người dùng có thể nhập vào các kí tự trong khoảng từ A đến E trước, sau đó mới nhập các kí tự trong khoảng từ G đến N; hoặc ngược lại; hoặc nhập xen kẽ nhau; miễn sao các kí tự nằm trong các phạm vi từ A đến E, và từ G đến N.

Số lượng kí tự cho phép trong ví dụ này tối thiểu là 2, tối đa là 8. Điều này thì các bạn đã biết.

Chạy chương trình, chúng ta được kết quả như sau:

|  |
| --- |
| Input your street name: AEF  Invalid data!  Input your street name: OP  Invalid data!  Input your street name: gek  Invalid data!  Input your street name: GEK  Valid data |

Chuỗi AEF: không hợp lệ vì có chứa kí tự F không nằm trong phạm vi cho phép

Chuỗi OP: không hợp lệ vì có các kí tự O, P không nằm trong phạm vi cho phép

Chuỗi gek: không hợp lệ vì mặc dù tất cả các kí tự đề nằm trong phạm vi cho phép, nhưng lại là chữ thường, trong khi mẫu qui định tất cả phải là chữ IN HOA

Chuỗi GEK: hoàn toàn hợp lệ vì tất cả kí tự đều nằm trong phạm vi cho phép và đều là chữ IN HOA

**Trường hợp 3: Kiểm tra cả số và kí tự nằm trong các phạm vi được qui định trước**

Tương tự trong 2 trường hợp ở trên, chúng ta có thể kết hợp các phạm vi dành cho số và kí tự alphane.

Ví dụ giống như trong trường hợp trên là chúng ta cho phép người dùng nhập vào địa chỉ nhà với các kí tự từ A đến E, và từ G đến N. Để tạo sự thuận tiện cho người dùng, chúng ta cho phép nhập thêm vào các chữ số từ 5 đến 9.

Chúng ta xem đoạn chương trình sau:

|  |
| --- |
| import java.util.Scanner;  public class Demo {  public static void main(String[] args) {  boolean flag;  Scanner sc = new Scanner(System.in);  do {  String addressPattern = "[1-5A-EG-N]{2,8}";  System.out.print("Input your street name: ");  String input = sc.next();  flag = input.matches(addressPattern);  if (!flag) System.out.println("Invalid data!");  } while (!flag);  System.out.println("Valid data");  }  } |

Trog đoạn chương trình trên, tôi định nghĩa mẫu như sau:

String addressPattern = "[1-5A-EG-N]{2,8}";

Trong mẫu này, phần phạm vi các kí tự giống như mẫu trong trường hợp vừa rồi, tôi chỉ chèn thêm phạm vi dành cho số từ 1 đến 5.

Như vậy, mẫu này có nghĩa là:

* Người dùng được nhập các kí tự số từ 1 đến 5, bao gồm: 1, 2, 3, 4, 5
* Người dùng được nhập các kí tự chữ IN HOA từ A đến E, bao gồm: A, B, C, D, E
* Người dùng được nhập các kí tự chữ IN HOA từ G đến N, bao gồm: G, H, I, J, K, L, M, N
* Người dùng phải nhập tối thiểu 2 kí tự, tối đa 8 kí tự

Chú ý, không có khoảng trắng trong mẫu, có nghĩa là người dùng không được nhập kí tự khoảng trắng.

Chạy chươn trình, chúng ta được kết quả như sau:

|  |
| --- |
| Input your street name: 8ACD  Invalid data!  Input your street name: Gc123  Invalid data!  Input your street name: GC123  Valid data |

Chuỗi 8ACD: không hợp lệ vì có chứa số 8, không nằm trong phạm vi cho phép từ 1 đến 5

Chuỗi Gc123: không hợp lệ vì có chứa kí tự c KHÔNG phải kí tự IN HOA

Chuỗi GC123: hoàn toàn hợp lệ vì tất cả kí tự IN HOA và số đều nằm trong phạm vi qui định trong mẫu

**Trường hợp 4: Kiểm tra một chuỗi bao gồm các kí tự được qui định trước**

Ngoài các trường hợp ràng buộc phạm vi cho kí tự alphabe và số, chúng ta có thể quy định mẫu khớp với từng kí tự riêng lẻ bằng cách sử dụng cặp dấu vuông ([])

Chúng ta xem ví dụ sau:

|  |
| --- |
| import java.util.Scanner;  public class Demo {  public static void main(String[] args) {  boolean flag;  Scanner sc = new Scanner(System.in);  do {  String addressPattern = "[1357ASDF]+";  System.out.print("Input your street name: ");  String input = sc.next();  flag = input.matches(addressPattern);  if (!flag) System.out.println("Invalid data!");  } while (!flag);  System.out.println("Valid data");  }  } |

Trong ví dụ trên, tôi định nghĩa một mẫu như sau:

String addressPattern = "[1357ASDF]+";

Mẫu này có nghĩa là:

* Chuỗi trùng chỉ được chứa các số: 1, 3, 5, 6 và các kí tự IN HOA: A, S, D, F.
* Số lượng kí tự tối thiểu phải là 1, vì tôi sử dụng dấu cộng (+) trong mẫu.

Chạy chương trình, chúng ta được kết quả như sau:

|  |
| --- |
| Input your street name: 12ASD  Invalid data!  Input your street name: 13ACB  Invalid data!  Input your street name: AD57  Valid data |

Chuỗi 12ASD: không hợp lệ vì có số 2 không phải là một trong các số được quy định trong mẫu

Chuỗi 13ACB: không hợp lệ vì có 2 kí tự C và B không phải là một trong các kí tự được quy định trong mẫu

Chuỗi AD57: hoàn toàn hợp lệ vì các số và kí tự IN HOA khớp với mẫu qui định

**Trường hợp 5: Kiểm tra một chuỗi KHÔNG bao gồm các kí tự được qui định trước**

Thay vì kiểm tra một chuỗi có bao gồm các kí tự cụ thể được qui định trước hay không, chúng ta có thể kiểm tra một chuỗi KHÔNG được chứa các kí tự được qui định sẵn.

Chúng ta xem ví dụ sau:

|  |
| --- |
| import java.util.Scanner;  public class Demo {  public static void main(String[] args) {  boolean flag;  Scanner sc = new Scanner(System.in);  do {  String addressPattern = "[^1357ASDF]+";  System.out.print("Input your street name: ");  String input = sc.next();  flag = input.matches(addressPattern);  if (!flag) System.out.println("Invalid data!");  } while (!flag);  System.out.println("Valid data");  }  } |

Trong ví dụ trên, tôi định nghĩa mẫu như sau:

String addressPattern = "[^1357ASDF]+";

Mẫu này gần như tương tự như trong ví dụ trước, và tôi sử dụng thêm dấu mũ (^) phía trước các kí tự. Mẫu này có nghĩa là:

* Chuỗi trùng KHÔNG được chứa các số 1, 3, 5, 7; và KHÔNG được chứa các kí tự IN HOA: A, S, D, F.
* Số lượng kí tự tối thiểu phải là 1, vì tôi vẫn dùng dấu cộng (+) trong mẫu

Chạy chương trình, chúng ta được kết quả như sau:

|  |
| --- |
| Input your street name: 135ASDF  Invalid data!  Input your street name: QWE  Valid data |

Chuỗi 135ASDF: không hợp lệ vì chứa các kí tự KHÔNG cho phép trong mẫu

Chuỗi QWE: hoàn toàn hợp lệ ví chứa các kí tự không bị cấm trong mẫu.

**Phần 4: Kiểm tra tính hợp lệ của dữ liệu – Kiểm tra các mẫu phổ biến: số điện thoại, email**

Trong phần này, chúng ta sẽ sử dụng regular expression để kiểm tra một số định dạng dữ liệu phổ biến: số điện thoại và địa chỉ email.

**Trường hợp 1: Kiểm tra số điện thoại**

Chúng ta có rất nhiều định dạng số điện thoại như điện thoại bàn, điện thoại di động, có mã quốc gia, có mã vùng…Tùy vào yêu cầu của mỗi trường hợp mà chúng ta sẽ có những mẫu tương ứng kiểm tra.

Trong phần này, tôi sẽ tạo ra một mẫu để kiểm tra định dạng của số điện thoại cơ bản nhất. Khi đã nắm được ý tưởng cơ bản, các bạn có thể tự mình định nghĩa ra mẫu cho từng yêu cầu cụ thể của mình.

Ví dụ, tôi cần kiểm tra một số điện thoại với mẫu như sau:

Mã quốc gia (3 số) – mã vùng (2 số) – số điện thoại (7 số)

Như vậy, tôi yêu cầu người dùng phải nhập một số điện thoại với đầy đủ các yếu tố như sau:

* Mã quốc gia: phải nhập đủ 3 chữ số, theo sau là dấu gạch nối (-)
* Mã vùng: phải nhập đủ 2 chữ số, theo sau là dấu gạch nối (-)
* Số điện thoại: phải nhập đủ 7 chữ số

Chúng ta xem đoạn chương trình sau:

|  |
| --- |
| import java.util.Scanner;  public class Demo {  public static void main(String[] args) {  boolean flag;  Scanner sc = new Scanner(System.in);  do {  String phonePattern = "\\d{3}-\\d{2}-\\d{7}";  System.out.print("Input your phone(xxx-xx-xxxxxxx): ");  String input = sc.next();  flag = input.matches(phonePattern);  if (!flag) System.out.println("Invalid data!");  } while (!flag);  System.out.println("Valid data");  }  } |

Dựa vào yêu cầu bài toán, tôi định nghĩa ra mẫu như sau:

String phonePattern = "\\d{3}-\\d{2}-\\d{7}";

* Phần đầu yêu cầu là 3 số cho mã quốc gia, do vậy, tôi dùng \\d{3}, và theo sau là dấu gạch nối
* Phần tiếp theo là 2 số cho mã vùng, do vậy, tôi dùng \\d{2}, và theo sau là dấu gạch nối
* Phần cuối cùng là 7 số cho số điện thoại, tôi dùng \\d{7}

Chạy chương trình, chúng ta có kết quả như sau:

|  |
| --- |
| Input your phone(xxx-xx-xxxxxxx): 084-888-1234567  Invalid data!  Input your phone(xxx-xx-xxxxxxx): 084-88-123456  Invalid data!  Input your phone(xxx-xx-xxxxxxx): 084-38-1234567  Valid data |

Chuỗi 084-888-1234567: không hợp lệ vì phần mã vùng có tới 3 số (888), trong khi yêu cầu chỉ có 2 số

Chuỗi 084-88-123456: không hợp lệ vì phần số điện thoại chỉ có 6 số (123456), trong khi yêu cầu là 7 số

Chuỗi 084-38-1234567: hoàn toàn hợp lệ vì đáp ứng tất cả yêu cầu qui định trong mẫu

**Trường hợp 2: Kiểm tra số điện thoại với kỹ thuật phân nhóm (grouping)**

Tuy nhiên, thông thường nếu gọi điện thoại trong nước thì chúng ta không cần phải nhập mã quốc gia. Do vậy, để tạo sự thuận tiện cho người sử dụng, thì chúng ta cho tùy chọn: có thể nhập mã quốc gia hoặc không.

Để thực hiện điều này thì chúng ta sử dụng một kỹ thuật gọi là phân nhóm (grouping) để nhóm 3 số mã quốc gia lại. Nghĩa là người dùng có thể bỏ qua hoàn toàn, không cần nhập 3 số này; hoặc nếu nhập thì phải nhập đủ 3 số

Chúng ta xem đoạn chương trình sau:

|  |
| --- |
| import java.util.Scanner;  public class Demo {  public static void main(String[] args) {  boolean flag;  Scanner sc = new Scanner(System.in);  do {  String phonePattern = "(\\d{3}-)?\\d{2}-\\d{7}";  System.out.print("Input your phone(xxx-xx-xxxxxxx): ");  String input = sc.next();  flag = input.matches(phonePattern);  if (!flag) System.out.println("Invalid data!");  } while (!flag);  System.out.println("Valid data");  }  } |

Trong đoạn chương trình trên, tôi định nghĩa mẫu như sau:

String phonePattern = "(\\d{3}-)?\\d{2}-\\d{7}";

Vẫn là mẫu trước đó, và tôi đặt phần mã quốc gia và dấu gạch nối (-) vào trong cặp dấu ngoặc tròn (()), theo sau là dấu chấm hỏi (?).

Tất cả mọi kí tự nào đặt trong cặp dấu ngoặc tròn (()) đều thuộc về một nhóm.

Và chú ý, dấu chấm hỏi (?) theo sau rất quan trọng, vì dấu này có nghĩa là có thể không có, hoặc nếu có chỉ xuất hiện 1 lần.

Như vậy, trong mẫu của tôi, người dùng hoặc có thể bỏ qua hoàn toàn mã quốc gia và dấu gạch nối (-); hoặc phải nhập vào đủ 3 số cho mã quố gia và dấu gạch nối (-).

Chạy chương trình, chúng ta được kết quả như sau:

|  |
| --- |
| Input your phone(xxx-xx-xxxxxxx): 123-1234567  Invalid data!  Input your phone(xxx-xx-xxxxxxx): 12-12-1234567  Invalid data!  Input your phone(xxx-xx-xxxxxxx): 084-38-1234567  Valid data |

Chuỗi 123-1234567: không hợp lệ, vì mặc dù không nhập mã quốc gia, nhưng có tới 3 số cho mã vùng, trong khi yêu cầu chỉ có 2 số

Chuỗi 12-12-1234567: không hợp lệ, vì chỉ có 2 số cho mã quốc gia, trong khi yêu cầu là phải có 3 số

Chuỗi 084-38-1234567: hoàn toàn hợp lệ vì có đầy đủ các chữ số cho cả 3 phần

Chạy lại chương trình:

|  |
| --- |
| Input your phone(xxx-xx-xxxxxxx): 38-1234567  Valid data |

Chuỗi 38-1234567: hoàn toàn hợp lệ, vì chúng ta có thể bỏ qua phần mã quốc gia, chỉ cung cấp mã vùng và số điện thoại.

**Trường hợp 3: Kiểm tra định dạng địa chỉ email**

Tương tự như các trường hợp số điện thoại, chúng ta cũng có rất nhiều các định dạng địa chỉ email khác nhau. Không thể có một mẫu mà có thể dùng để kiểm tra tất cả mọi định dạng địa chỉ email trên thế giới, và điều này cũng hoàn toàn không cần thiết, vì nếu một mẫu quá phức tạp sẽ làm chậm quá trình so trùng chuỗi với mẫu qui định trước.

Tùy vào các yêu cầu cụ thể mà chúng ta định nghĩa các mẫu kiểm tra tương ứng.

Ví dụ, chúng ta cần kiểm tra mẫu email với định dạng như sau:

[email@address.com](mailto:email@address.com)

Trong mẫu này, chúng ta chia làm 5 phần như sau:

* Phần đầu là tên tài khoản (chuỗi email): phần này có thể chứa kí tự alphabe hoặc số
* Phần thứ 2 là dấu @
* Phần thứ 3 là tên miền (domain, chuỗi address): phần này có thể chứa kí tự alphabe hoặc số
* Phần thứ 4 là dấu chấm (.).
* Phần thứ 5 là cấp độ của tên miền (chuỗi com): phần này có thể chứa kí tự alphabe hoặc số

Chúng ta xem đoạn chương trình sau:

|  |
| --- |
| import java.util.Scanner;  public class Demo {  public static void main(String[] args) {  boolean flag;  Scanner sc = new Scanner(System.in);  do {  String emailPattern = "\\w+@\\w+[.]\\w+";  System.out.print("Input your email(email@address.com): ");  String input = sc.next();  flag = input.matches(emailPattern);  if (!flag) System.out.println("Invalid data!");  } while (!flag);  System.out.println("Valid data");  }  } |

Trong đoạn chương trình trên, dựa vào yêu cầu ở trên, tôi định nghĩa mẫu như sau:

String emailPattern = "\\w+@\\w+[.]\\w+";

* Do phần đầu là tên tài khoản có thể chứa kí tự alphabe hoặc số, và có thể có nhiều hơn 1 kí tự, do vậy tôi dùng [\\w](file:///\\w)+. Vì [\\w](file:///\\w) là đại diện co các kí tự aphabe từ a đến z, và các chữ số từ 0 đến 9.
* Phần thứ 2 là dấu @, do vậy tôi chỉ định cụ thể dấu @ ở đây
* Phần thứ 3 là tên miền, có thể có kí tự hoặc số, và có thể có nhiều hơn 1 kí tự, do vậy tôi cũng dùng [\\w](file:///\\w)+ ở đây
* Phần thứ 4 là dấu chấm (.). Lưu ý ở đây, trong các kí hiệu của Regular Expression thì dấu chấm (.) là đại diện cho kí tự bất kì. Tuy ở đây, chúng ta lại muốn có dấu chấm (.) chứ không phải bất kỳ kí tự nào cũng được. Vì vậy, chúng ta có 2 cách để sử dụng dấu chấm (.) ở đây: hoặc đặt dấu chấm (.) vào cặp dấu ngoặc vuông, như tôi sử dụng trong chương trình; hoặc có thể dùng 2 dấu **\\** phía trước: **\\.**
* Phần thứ 5 là cấp độ tên miền, cũng có thể chứa kí tự và số, và có thể có nhiều hơn 1 kí tự; do vậy, tôi cũng dùng [\\w](file:///\\w)+

Chạy chương trình, chúng ta được kết quả như sau:

|  |
| --- |
| Input your email(email@address.com): hung@  Invalid data!  Input your email(email@address.com): hung@fpt.  Invalid data!  Input your email(email@address.com): hung@fpt.com  Valid data |

Chuỗi hung@: không hợp lệ vì thiếu phần tên miền và cấp độ tên miền

Chuỗi hung@fpt.: không hợp lệ vì thiếu phần cấp độ tên miền

Chuỗi: [hung@fpt.com](mailto:hung@fpt.com): hoàn toàn hợp lệ vì có đầy đủ các phần yêu cầu.

Và tương tự như trong phần kiểm tra số điện thoại, chúng ta có thể sử dụng kỹ thuật grouping để tạo sự linh hoạt hơn cho người dùng.

Ví dụ, tôi cho phép người dùng có thể nhập vào địa chỉ email:

[email@address.com.vn](mailto:email@address.com.vn)

tức là người dùng có thể nhập thêm vào một cấp độ domain nữa (phần .vn); hoặc có thể không nhập.

Chúng ta xem đoạn chương trình sau:

|  |
| --- |
| import java.util.Scanner;  public class Demo {  public static void main(String[] args) {  boolean flag;  Scanner sc = new Scanner(System.in);  do {  String emailPattern = "\\w+@\\w+[.]\\w+([.]\\w+)?";  System.out.print("Input your email(email@address.com): ");  String input = sc.next();  flag = input.matches(emailPattern);  if (!flag) System.out.println("Invalid data!");  } while (!flag);  System.out.println("Valid data");  }  } |

Trong chương trình trên, tôi định nghĩa mẫu như sau:

String emailPattern = "\\w+@\\w+[.]\\w+([.]\\w+)?";

Phần đầu thì cũng giống như ví dụ trên, tôi thêm vào phần sau của mẫu ban đầu một mẫu con:

([.][\\w](file:///\\w)+)?

Tôi đặt trong cặp dấu ngoặc tròn (()) có nghĩa là phân nhóm như chúng ta đã biết.

Trong cặp ngoặc nhọn này, tôi chỉ định người cùng có thể nhập thêm dấu chấm (.) và các kí tự từ (word character)

Và chú ý theo sau phải là dấu chấm hỏi (?), cho phép người dùng có thể tùy chọn nhập hoặc không.

Chạy chương trình, chúng ta có kết quả như sau:

|  |
| --- |
| Input your email(email@address.com): hung@fpt.com.vn.vn  Invalid data!  Input your email(email@address.com): hung@fpt.com.vn  Valid data |

Chuỗi [hung@fpt.com.vn.vn](mailto:hung@fpt.com.vn.vn): không hợp lệ, vì có tới 2 phần .vn; trong khi chúng ta chỉ cho phép hoặc không nhập, hoặc nếu có thì chỉ có một .vn (hoặc chấm cái gì đó)

Chuỗi [hung@fpt.com.vn](mailto:hung@fpt.com.vn): hoàn toàn hợp lệ, vì có tất cả các thành phần quy định trong mẫu.

>> Xem tiếp Phần 5

**Phần 5: Sử dụng kĩ thuật nhóm (Grouping) cho phép lựa chọn các mục được quy định trước và kiểm tr định dạng ngày tháng năm**

Trong các phần trước chúng ta đã sử dụng kỹ thuật grouping để cho phép người dùng có thể tùy chọn nhập vào một số kí tự hay không. Trong phần này chúng ta sẽ sử dụng kỹ thuật grouping để quy định sẵn một số mục dữ liệu mà người dùng bắt buộc phải nhập vào.

Chúng ta xem đoạn chương trình sau:

|  |
| --- |
| import java.util.Scanner;  public class Demo {  public static void main(String[] args) {  boolean flag;  Scanner sc = new Scanner(System.in);  do {  String favorPattern = "(sport|music|book|movie)";  System.out.print("Input a favorite (sport,music,book,movie): ");  String input = sc.next();  flag = input.matches(favorPattern);  if (!flag) System.out.println("Invalid data!");  } while (!flag);  System.out.println("Valid data");  }  } |

Trong đoạn chương trình trên, tôi định nghĩa mẫu như sau:’

String emailPattern = "(sport|music|book|movie)";

Để sử dụng kỹ thuật grouping, chúng ta vẫn dùng cặp dấu ngoặc nhọn (()).

Bên trong dấu ngoặc nhọn tôi chỉ định một danh sách các mục mà người dùng phải nhập, mỗi một mục cách biệt bởi dấu gạch đứng (|).

Như vậy, mẫu này có nghĩa, người dùng phải nhập những giá trị nằm trong danh sách này mà thôi.

Chạy chương trình, chúng ta có kết quả như sau:

|  |
| --- |
| Input a favorite (sport,music,book,movie): reading  Invalid data!  Input a favorite (sport,music,book,movie): sport  Valid data |

Chuỗi “reading”: không hợp lệ vì không nằm trong danh sách quy định trước

Chuỗi “sport”: hoàn toàn hợp lệ vì nằm trong danh sách quy định

Ngoài việc sử dụng grouping để quy định trước một danh sách các mục mà người dùng phải nhập, chúng ta còn có thể dùng kỹ thuật grouping để kiểm tra các định dạng ngày tháng, ví dụ như kiểm tra định dạng ngày sinh nhật.

Chúng ta xem ví dụ sau:

|  |
| --- |
| import java.util.Scanner;  public class Demo {  public static void main(String[] args) {  boolean flag;  Scanner sc = new Scanner(System.in);  do {  String dobPattern = "([1-9]{1,2}/)?([1-9]{1,2})/([0-9]{4})";  System.out.print("Input your day of birth: ");  String input = sc.next();  flag = input.matches(dobPattern);  if (!flag) System.out.println("Invalid data!");  } while (!flag);  System.out.println("Valid data");  }  } |

Trong đoạn chương trình trên, tôi định nghĩa mẫu như sau:

String dobPattern = "([1-9]{1,2}/)?([1-9]{1,2})/([0-9]{4})";

Trong mẫu trên tôi tách làm 3 nhóm:

Nhóm 1: ([1-9]{1,2}/)?

Nhóm này người dùng phải nhập tối thiểu 1 số, tối đa 2 số; và các con số chỉ từ 1 đến 9, không được chứa số 0; theo sau là dấu / để phân cách giữa ngày và tháng. Tôi dùng dấu chấm hỏi (?) theo sau nhóm này để người dùng có thể nhập ngày hoặc không

Nhóm 2: ([1-9]{1,2})

Nhóm này yêu cầu người dùng phải nhập tối thiểu 1 số, tối đa 2 chữ số từ 1 đến 9. Và cũng phân cách với năm bằng dấu /

Nhóm 3: ([0-9]{4})

Nhóm này yêu cầu người dùng phải nhập đủ 4 chữ số từ 0 đến 9. So năm sinh có thể chứa số 0, ví dụ 1990, 1980

Chạy chương trình, chúng ta được kết quả như sau:

|  |
| --- |
| Input day of birth: 1212/1980  Invalid data!  Input day of birth: 12/1980  Valid data  Input day of birth: 1/12/1980  Valid data |

Chuỗi 1212/1980: không hợp lệ, vì không có dấu / phân cách giữa ngày và tháng

Chuỗi 12/1980: hoàn toàn hợp lệ vì chúng ta có thể bỏ qua ngày

Chuỗi 1/12/1980: hoàn toàn hợp lệ vì các thành phần khớp với các qui định trong mẫu.

Như vậy qua 5 phần vừa rồi, chúng ta đã ứng regular expression vào việc kiểm tra tính hợp lệ của dữ liệu. Và như chúng ta cũng nhận thấy, tùy thuộc vào yêu cầu cụ thể là gì, chúng ta cần phân tích từng thành phần trong yêu cầu và đưa ra các mẫu kiểm tra tương ứng.

Hãy xem các phần kế tiếp, chúng ta sẽ ứng dụng Regular Expression vào việc thay thế chuỗi con trong chuỗi cha, và ngắt các chuỗi con từ chuỗi cha.

**Phần 6: Thay thế chuỗi**

Chúng ta có rất nhiều các trường hợp cần thay thế một chuỗi hoặc một số kí tự nào đó trong chuỗi ban đầu (gọi là chuỗi cha). Ví dụ như thay thế các kí tự lỗi hoặc dư thừa khi gõ văn bản chẳng hạn.

Lớp java.lang.String trong Java hỗ trợ 2 phương thức cho phép chúng ta có thể thay thế chuỗi con trong một chuỗi cha bằng một chuỗi mới.

Ví dụ tôi có một chuỗi như sau:

“Đây là một văn bảN có nhiều lỗi đáNh máy. Chúng ta cần thay thế các kí tự N hoa thành N thường”

Trong chuỗi trên, thì như chúng ta thấy, có một số chữ N viết hoa do lỗi đánh máy, và chúng ta cần thay thế các chữ N hoa này thành n viết thường.

Chúng ta xem đoạn chương trình sau:

|  |
| --- |
| public class Demo {  public static void main(String[] args) {  String text = "Đây là một văn bảN có nhiều lỗi đáNh máy. Chúng ta cần thay thế các kí tự N hoa thành N thường";  text = text.replace("N", "n");  System.out.println(text);  }  } |

Ttrong đoạn chương trình trên, tôi sử dụng phương thức replace() của lớp String như sau:

text = text.replace("N", "n");

Phương thức replace() nhận vào 2 tham số:

* Tham số thứ 1 là chuỗi hoặc kí tự cần thay thế
* Tham số thứ 2 là chuỗi hoặc kí tự mới sẽ thay thế vào chuỗi ban đầu

Ví dụ trong trường hợp này, phương thức replace() sẽ tìm tất cả các chữ N hoa và thay thết bằng chữ n viết thường.

Do phương thức replace() trả về một chuỗi mới đã được thay thế, do vậy chúng ta có thể gán lại vào biết String text, hoặc các bạn có thể gán vào một biến khác nếu cần thiết.

Chạy chương trình, chúng ta có chuỗi mới đã được thay thế như sau:

|  |
| --- |
| Đây là một văn bản có nhiều lỗi đánh máy. Chúng ta cần thay thế các kí tự n hoa thành n thường |

Như chúng ta cũng thấy, phương thức replace() khá thuận tiện torng việc thay thế một chuỗi con bằng một chuỗi mới.

Tuy nhiên, điểm bất lợi của phương thức này là không hỗ trợ các cú pháp của Regular Expression, mà chúng ta phải chỉ định cụ thể kí tự, hoặc chuỗi con cần thay thế là gì.

Ví dụ như tôi có chuỗi văn bản sau:

“Đây là một văn 3bản có 6nhiều lỗi đ67ánh máy vì có chứa các kí tự số”

Nhưng chúng ta thấy trong chuỗi trên, có một số kí tự số và chúng ta cần phải xóa các kí tự số này ra khỏi văn bản.

Trong trường hợp này, chúng ta có thể sử dụng phương thức String.replace(), nhưng phải gọi nhiều lần để chỉ định cụ thể số cần xóa là số nào. Điều này rất không hiệu quả về mặt lập trình.

Thay vào đó, chúng ta sẽ sử dụng phương thứ String.replaceAll().

Phương thức String.replaceAll() này cũng giống như phương thức String.replace(), tuy nhiên có hỗ trợ cú pháp của Regular Expression.Do vậy, trong trường hợp này, chúng ta sẽ dùng phương thức String.replaceAll() để chỉ định thay thế tất cả các kí tự số, mà không cần quan tâm đó là số mấy, bằng các kí tự rỗng. Tức là xóa đi các kí tự số trong chuỗi.

Chúng ta xem đoạn chương trình sau:

|  |
| --- |
| public class Demo {  public static void main(String[] args) {  String text = "Đây là một văn 3bản có 6nhiều lỗi đ67ánh máy vì có chứa các kí tự số";  text = text.replaceAll("\\d", "");  System.out.println(text);  }  } |

Trong đoạn chương trình trên tôi sử dụng phương thức String.replaceAll() như sau:

text = text.replaceAll("\\d", "");

Phương thức String.replaceAll() này cũng nhận vào 2 tham số

* Tham số thứ nhất là chuỗi hoặc kí tự cần thay thế. Và chúng ta có thể sử dụng Regular Expression tại tham số này. Do vậy tôi chỉ định [\\d](file:///\\d), vì tôi muốn thay thế tất cả các kí tự số
* Tham số thứ 2 là chuỗi hoặc kí tự mới sẽ thay thế. Do muốn xóa các kí tự số, do vậy tôi truyền vào tham số này là một kí tự rỗng. Vì vậy, tất cả các kí tự số sẽ được thay thế bằng kí tự rỗng, tương đương với việc các kí tự số sẽ bị xóa đi.

Chạy chương trình, chúng ta được kết quả như sau:

|  |
| --- |
| Đây là một văn bản có nhiều lỗi đánh máy vì có chứa các kí tự số |

**Phần 7: Tiếng việt (hay Unicode nói chung) trong regular expression**

Đôi khi sử dụng Regular Expression với các kí tự unicode, như tiếng việt, sẽ có một số bất tiện mà nếu không chú ý, sẽ dẫn tới kết quả không như chúng ta mong muốn.

Chúng ta xem ví dụ sau:

|  |
| --- |
| public class Demo {  public static void main(String[] args) {  String text = "Đây là một văn# có% c&hứa c@ác kí tự đặc biệt";  text = text.replaceAll("[^\\w\\s]", "");  System.out.println(text);  }  } |

Trong đoạn chương trình trên, tôi có một chuỗi văn bản chứa các kí tự đặc biệt như sau:

|  |
| --- |
| String text = "Đây là một văn# có% c&hứa c@ác kí tự đặc biệt"; |

Và tôi muốn xóa tất cả các kí tự đặc biệt này ra khỏi chuỗi văn bản bằng cách thay thế các kí tự đặc biệt bằng các kí tự rỗng như chúng ta thực hiện trong phần trước.

Tuy nhiên, trong các kí tự của Regular Expression chúng ta lại không có kí hiệu nào đại diện cho tất cả các kí tự đặc biệt, giống như kí hiệu \d là đại diện cho số.

Do vậy, trong phương thức replaceAll(), tôi chọn giải pháp như sau:

|  |
| --- |
| text = text.replaceAll("[^\\w\\s]", ""); |

Trong mẫu trên, tôi sử dụng dấu mũ (^) theo sau là [\\w](file:///\\w), theo sau là [\\s](file:///\\s):

* Dấu mũ (^) như các bạn đã biết có nghĩa là “không”
* Kí hiệu [\\w](file:///\\w) là đại diện cho các kí tự từ a đến z, và số từ 0 đến 9
* Kí hiệu [\\s](file:///\\s) là đại diện cho các kí tự khoảng trắng. Vì tôi muốn giữ lại khoảng trắng trong chuỗi ban đầu

Như vậy, phương thức replaceAll() có nghĩa là: thay thế tất cả mọi kí tự bằng các kí tự rỗng, ngoại trừ các kí tự đại diện bởi [\\w](file:///\\w) và kí tự khoảng trắng.

Như vậy, chúng ta vẫn có thể xóa các kí tự đặc biệt mà vẫn giữ lại các kí tự thông thường.

Tuy nhiên, chạy chương trình lên, chúng ta có kết quả như sau:

|  |
| --- |
| y l mt vn c cha cc k t c bit |

Chương trình không chạy sai, mà chạy đúng theo những gì chúng ta yêu cầu.

Vì chúng ta yêu cầu giữ lại các kí tự đại diện bởi [\\w](file:///\\w) và khoảng trắng.

Tuy nhiên do [\\w](file:///\\w) chỉ đại diện cho các kí tự từ a đến z, và số từ 0 đến 9, trong khi chuỗi văn bản ban đầu có các kí tự Unicode là tiếng việt như â, à, ột, …..Nên phương thức replaceAll() cũng sẽ thay thế các kí tự Unicode này bằng các kí tự khoảng trống.

Như vậy, trong các trường hợp văn bản của chúng ta có chứa các kí tự Unicode như tiếng Việt, tiếng Nhật, Hàn Quốc…thì chúng ta phải chỉ định trong mẫu Regular Expression là xử lý kí tự Unicode.

Chúng ta sử dụng như sau:

|  |
| --- |
| text = text.replaceAll("[^\\p{L}\\s]", ""); |

Để chỉ định sử dụng Unicode chúng ta dùng kí hiệu [\\p](file:///\\p), theo sau là kí tự đại diện nằm trong cặp dấu ngoặc nhọn.

Chú ý, vì chúng ta đang chỉ định xử lý các kí tự Unicode, nên các kí tự thông thường từ a đến z cũng sẽ khác, không còn là [\\w](file:///\\w) nữa, mà là kí tự L.

L có nghĩa là đại diện cho các kí tự Unicode Alphabe.

Ngoài ra còn các kí tự đại diện khác như số ([\\d](file:///\\d)), khoảng trắng ([\\s](file:///\\s)) vẫn sử dụng như bình thường.

Như vậy, chương trình hoàn chỉnh của chúng ta như sau:

|  |
| --- |
| public class Demo {  public static void main(String[] args) {  String text = "Đây là một văn# có% c&hứa c@ác kí tự đặc biệt";  text = text.replaceAll("[^\\p{L}\\s]", "");  System.out.println(text);  }  } |

Chạy chương trình, chúng ta được kết quả mong muốn như sau:

|  |
| --- |
| Đây là một văn có chứa các kí tự đặc biệt |

**Phần 8: Ngắt chuỗi với lớp java.util.Scanner**

Trong 2 phần trước, chúng ta đã tìm hiểu việc ứng dụng Regular Expression trong việc thay thế chuỗi con hoặc kí tự trong một chuỗi cha.

Trong các phần tiếp theo này, chúng ta sẽ tìm hiểu việc ứng dụng Regular Expression để ngắt các chuỗi con, hay còn gọi là token từ chuỗi cha.

Có 2 thuật ngữ chúng ta lưu ý trước khi thực hiện công việc ngắt chuỗi:

Token: là các từ có nghĩa

Delimiter: các kí tự phân cách các token

Ví dụ chúng ta có chuỗi sau:

|  |
| --- |
| I love you so much |

Trong chuỗi trên:

Token là các từ: I, love, you, so, much

Delimiter: là các kí tự khoảng trắng dùng phân cách các từ (token)

Tuy nhiên, chúng ta có thể linh hoạt chỉ định kí tự dùng làm delimiter là gì, không nhất thiết phải là kí tự khoảng trắng. Mặc dù, mặc định hầu hết các hàm trong Java đều dùng kí tự khoảng trắng làm delimiter.

Có lẽ chúng ta đã quen thuộc với lới java.util.Scanner dùng để lấy dữ liệu người dùng nhập vào từ cửa sổ dòng lệnh (console interface).

Tuy nhiên lớp java.util.Scanner còn có thể nhận đầu vào là một biến String, hoặc một tập tin dữ liệu (file).

Ví dụ, tôi có chuỗi sau:

|  |
| --- |
| I love you so much. I want to marry you |

Và tôi muốn ngắt từ từ trong chuỗi trên ra.

Chúng ta xem đoạn chương trình sau:

|  |
| --- |
| import java.util.Scanner;  public class Demo {  public static void main(String[] args) {  Scanner sc;  String s = "I love you so much. I want to marry you";  sc = new Scanner(s);  while (sc.hasNext()) {  String token = sc.next();  System.out.println(token);  }  }  } |

Trong đoạn chương trình trên, trước tiên tôi khai báo đối tượng lớp Scanner. Và thay vì truyền tham số cho constructor là System.in để chúng ta lấy dữ liệu từ cửa sổ dòng lệnh, thì tôi truyền vào là biến:

|  |
| --- |
| String s = "I love you so much. I want to marry you"; |

Như vậy, lớp Scanner sẽ đọc dữ liệu từ biến s này:

|  |
| --- |
| sc = new Scanner(s); |

Tiếp theo, tôi dùng vòng lặp while để duyệt qua tất cả các token trong chuỗi.

Trong điều kiện lặp của while(), tôi sử dụng phương thức hasNext() để kiểm tra có còn token nào trong chuỗi hay không:

|  |
| --- |
| while (sc.hasNext()) |

Phương thức hasNext() này sẽ trả về giá trị true nếu trong chuỗi vẫn còn token, ngược lại trả về false nếu đã hết token, tức là kết thúc chuỗi.

Mặc định phương thức hasNext() sẽ dùng kí tự khoảng trắng làm kí tự phân cách (delimiter) để phân biệt giữa các token.

Phương thức hasNext() sẽ đọc một token, và dừng lại nếu gặp kí tự delimter (trong trường hợp này là kí tự khoảng trắng). Nếu tiếp tục được gọi, phương thức hasNext() sẽ đọc token kế tiếp, và dừng lại nếu gặp kí tự delimiter. Và tiếp tục như vậy cho đến khi nào kết token, tức là kết thúc chuỗi thì thôi.

Thực ra phương thức hasNext() không thật sự đọc token, mà chỉ kiểm tra xem còn token hay không.

Phương thức thật sự đọc và trả về token là phương thức next():

|  |
| --- |
| String token = sc.next(); |

Và cuối cùng, chúng ta chỉ việc xử lý các token này. Trong trường hợp ví dụ ở đây, tôi chỉ in ra màn hình:

|  |
| --- |
| System.out.println(token); |

Chạy chương trình, chúng ta được kết quả như sau:

|  |
| --- |
| I  love  you  so  much.  I  want  to  marry  you |

Chúng ta được kết quả như trên là do mặc định, phương thức hasNext() sử dụng khoảng trắng làm kí tự phân cách.

Tuy nhiên chúng ta có thể chỉ định bất kì kí tự nào làm kí tự phân cách cũng được.

Ví dụ, tôi muốn sử dụng 2 kí tự là khoảng trắng và dấu chấm (.) làm kí tự phân cách thì tôi làm như sau:

|  |
| --- |
| import java.util.Scanner;  public class Demo {  public static void main(String[] args) {  Scanner sc;  String s = "I love you so much. I want to marry you";  sc = new Scanner(s);  sc.useDelimiter("[ .]");  while (sc.hasNext()) {  String token = sc.next();  System.out.println(token);  }  }  } |

Trong đoạn chương trình trên, tôi thêm vào dòng code sau:

|  |
| --- |
| sc.useDelimiter("[ .]"); |

Tôi sử dụng phương thức useDelimiter() để chỉ định các kí tự dùng làm kí tự phân cách. Tôi sử dụng kí tự khoảng trắng và dấu chấm (.).

Chú ý khi chúng ta đã sử dụng phương thức useDelimiter() để tùy chọn các kí tự delimiter, thì kí tự khoảng trắng không còn được sử dụng mặc định nữa. Do vậy, nếu vẫn muốn dùng kí tự khoảng trắng làm delimiter thì chúng ta phải chỉ định tường minh ở đây.

Chạy chương trình lên, chúng ta được kết quả như sau:

|  |
| --- |
| I  love  you  so  much  I  want  to  marry  you |

Trong kết quả trên, chúng ta thấy có một dòng trống. Sở dĩ có dòng trống này là do chúng ta sử dụng 2 kí tự delimiter là khoảng trắng và dấu chấm (.); và có thời điểm 2 kí tự này nằm liền kề nhau (giữa từ **much** và từ **I**).

Nếu chúng ta muốn xử lý 2 (hoặc nhiều hơn) kí tự delimiter nằm liền kề nhau như một kí tự delimiter thì chúng ta làm như sau:

|  |
| --- |
| sc.useDelimiter("[ .]+"); |

Chúng ta chỉ cần thêm dấu cộng (+) ngay phía sau dấu ngoặc vuông ([]).

Như vậy, trong trường hợp nếu có 2 hay nhiều kí tự delimiter nằm liền kề nhau, thì phương thức hasNext() sẽ xử lý như một kí tự delimiter mà thôi.

Chạy chương trình, chúng ta được kết quả như sau:

|  |
| --- |
| I  love  you  so  much  I  want  to  marry  you |

Như vậy, chúng ta không còn dòng trống nữa.

Ngoài việc chỉ định các kí tự cụ thể làm delimiter, thì chúng ta còn có thể chỉ định một mẫu của Regular Expression làm các kí tự delimiter.

Ví dụ tôi có chuỗi văn bản sau:

|  |
| --- |
| I love you 4 so much. 34 I 23 want to marry you |

Có các số trong chuỗi văn bản trên, và tôi muốn ngắt các chuỗi con dựa trên các số này.

Chúng ta xem đoạn chương trình sau:

|  |
| --- |
| import java.util.Scanner;  public class Demo {  public static void main(String[] args) {  Scanner sc;  String s = "I love you 4 so much. 34 I 23 want to marry you";  sc = new Scanner(s);  sc.useDelimiter("\\d+");  while (sc.hasNext()) {  String token = sc.next();  System.out.println(token);  }  }  } |

Trong đoạn chương trình trên, tôi chỉ thay đổi mẫu trong phương thức useDelimiter() như sau:

|  |
| --- |
| sc.useDelimiter("\\d+"); |

Tôi chỉ định tất cả các kí tự số từ 0 đến 9 làm kí tự delimiter. Chú ý chúng ta cần sử dụng dấu cộng (+) theo sau [\\d](file:///\\d), để phương thức hasNext() xử lý các kí tự số nằm gần nhau như một kí tự delimiter mà thôi.

Chạy chương trình lên, chúng ta có kết quả như sau:

|  |
| --- |
| I love you  so much.  I  want to marry you |

Như vậy, chuỗi ban đầu đã được ngắt ra thành các chuỗi con dựa vào các kí tự số.

Hoặc cũng trong trường hợp trên, tôi muốn lấy ra các chữ số 4, 34, và 23. Tức là tôi sử dụng tất cả các kí tự khác, ngoại trừ kí tự số từ 0 đến 9, làm kí tự delimiter để ngắt các số ra.

Chúng ta xem đoạn chương trình sau:

|  |
| --- |
| import java.util.Scanner;  public class Demo {  public static void main(String[] args) {  Scanner sc;  String s = "I love you 4 so much. 34 I 23 want to marry you";  sc = new Scanner(s);  sc.useDelimiter("[^\\d]+");  while (sc.hasNext()) {  String token = sc.next();  System.out.println(token);  }  }  } |

Trong đoạn chương trình trên, tôi thay đổi mẫu như sau:

|  |
| --- |
| sc.useDelimiter("[^\\d]+"); |

Tôi sử dụng dấu mũ (^) phía trước kí tự [\\d](file:///\\d) để chỉ định tất cả các kí tự khác, ngoại trừ các kí tự số từ 0 đến 9, sẽ được sử dụng như các kí tự delimiter.

Chú ý, do chúng ta sử dụng dấu mũ (^) nên chúng ta phải đặt toàn bộ mẫu trong cặp dấu ngoặc vuông ([]).

Chạy chương trình, chúng ta có kết quả như sau:

|  |
| --- |
| 4  34  23 |

Như vậy là chúng ta đã tách ra được các chữ số từ chuỗi ban đầu.

>> Xem tiếp **Phần 9**

**Phần 9: Ngắt Chuỗi Với Phương Thức String.split()**

Trong **Phần 8**, chúng ta đã tìm hiểu cách thức ngắt một chuỗi cha thành các chuỗi con sử dụng các phương thức trong lớp java.util.Scanner.

Trong phần này, chúng ta sẽ tìm hiểu việc tách các chuỗi con bằng cách sử dụng phương thức split() trong lớp String.

Có 2 sự khác biệt cơ bản trong việc ngắt chuỗi bằng cách sử dụng phương thức String.split() và lớp java.util.Scanner như sau:

* Phương thức String.split() không sử dụng kí tự khoảng trắng làm kí tự phân cách (delimiter) để ngắt chuỗi. Do vậy, chúng ta phải chỉ định tường minh các kí tự nào dùng làm kí tự phân cách cho phương thức String.split()
* Phương thức String.split() trả về một mảng kiểu String lưu trữ tất cả các chuỗi con đã tách ra từ chuỗi ban đầu.

Chúng ta xem ví dụ sau:

|  |
| --- |
| public class Demo {  public static void main(String[] args){  String tokens[];  String s = "I love you so much! But I cannot marry you.";  tokens = s.split("[ ]");  for (String token : tokens) {  System.out.println(token);  }  }  } |

Trong đoạn chương trình trên, tôi có chuỗi như sau:

|  |
| --- |
| String s = "I love you so much! But I cannot marry you."; |

Tôi muốn ngắt chuỗi này thành các chuỗi con (token) dựa vào kí tự khoảng trắng, tôi sử dụng phương thức String.split() như sau:

|  |
| --- |
| tokens = s.split("[ ]"); |

Có 2 điểm lưu ý về dòng code trên:

* Phương thức String.split() trả về một mảng kiểu String chức các tokens đã được tách ra, do vậy chúng ta cần gán vào một mảng String để lưu lại các tokens. Điều này rất thuận tiện cho việc xử lý các tokens này về sau.
* Do phương thức String.split() không sử dụng bất kì kí tự nào làm kí tự phân cách (delimiter), do vậy chúng ta phải chỉ định tường minh. Trong trường hợp này, tôi chỉ định kí tự khoảng trắng làm kí tự phân cách.

Sauk hi thực hiện xong việc ngắt chuỗi, tôi đã có một mảng lưu các chuỗi con, tôi sẽ in ra các chuỗi con torng mảng này:

|  |
| --- |
| for (String token : tokens) {  System.out.println(token);  } |

Chạy chương trình, chúng ta có kết quả như sau:

|  |
| --- |
| I  love  you  so  much!  But  I  cannot  marry  you. |

Không chỉ chấp nhận một kí tự, mà phương thức String.split() có thể chấp nhận một danh sách các kí tự dùng làm kí tự phân cách.

Ví dụ trong trường hợp trên, tôi muốn ngắt chuỗi dựa vào 2 kí tự: kí tự khoảng trắng và dấu chấm than (!); tôi chỉ cần thay đổi tham số trong phương thức String.split() như sau:

|  |
| --- |
| tokens = s.split("[ !]"); |

Chương trình hoàn chỉnh:

|  |
| --- |
| public class Demo {  public static void main(String[] args){  String tokens[];  String s = "I love you so much! But I cannot marry you.";  tokens = s.split("[ !]");  for (String token : tokens) {  System.out.println(token);  }  System.out.println("Number of tokens: " + tokens.length);  }  } |

Chú ý, tôi có in ra số lượng phần tử của mảng **tokens**, chính là số lượng chuỗi con đã tách ra từ chuỗi ban đầu.

Chạy chương trình, chúng ta được kết quả như sau:

|  |
| --- |
| I  love  you  so  much  But  I  cannot  marry  you.  Number of tokens: 11 |

Tuy nhiên, có tổng cộng 11 tokens, và một trong các token đó là token rỗng.

Sở dĩ có điều này là do tôi sử dụng 2 kí tự khoảng trắng và dấu chấm than (!) dùng là kí tự phân cách, và có thời điểm 2 kí tự này nằm liền kề nhau (giữa từ much và But). Do vậy, phương thức String.split() xem như đây là một kí tự rỗng.

Nếu chúng ta muốn phương thức String.split() xử lý 2 hay nhiều kí tự phân cách nằm kiền kề nhau như 1 kí tự phân cách, tức là trong ví dụ trên, chỉ trả về 10 token, thay vì 11 token, thì chúng ta thêm vào dấu cộng (+) ngay phía sau cặp dấu ngoặc vuông ([]):

|  |
| --- |
| tokens = s.split("[ !]+"); |

Chạy chương trình, chúng ta có kết quả như sau:

|  |
| --- |
| I  love  you  so  much  But  I  cannot  marry  you.  Number of tokens: 10 |

Và như chúng ta thấy trong phần kết quả in ra, chúng ta chỉ có 10 token, và cũng không còn kí tự rỗng nữa.

Ngoài việc chỉ định các kí tự cụ thể làm kí tự delimiter, chúng ta có thể chỉ định một mẫu dùng làm kí tự delimiter.

Chúng ta xem đoạn chương trình sau:

|  |
| --- |
| public class Demo {  public static void main(String[] args){  String tokens[];  String s = "I love you 4 so much. 34 I 23 want to marry you";  tokens = s.split("[\\s\\d]+");  for (String token : tokens) {  System.out.println(token);  }  System.out.println("Number of tokens: " + tokens.length);  }  } |

Trong đoạn chương trình trên, tôi có một chuỗi chứa các kí tự số như sau:

|  |
| --- |
| String s = "I love you 4 so much. 34 I 23 want to marry you"; |

Tôi muốn ngắt các token dựa vào các số và khảng trắng, tôi chỉ định mẫu trong phương thức String.split() như sau:

|  |
| --- |
| tokens = s.split("[\\s\\d]+"); |

* Thứ nhất, tôi chỉ định dùng kí tự khoảng trắng [\\s](file:///\\s) làm kí tự phân cách. Chúng ta có thể chỉ định cụ thể kí tự khảng trắng ( ), hoặc sử dụng kí tự đại diện [\\s](file:///\\s). Kí tự [\\s](file:///\\s) đại diện cho tất cả các kí tự khoảng trắng (whitespace characters) bao gồm: khoảng trắng, tab, new line (\n), line feed (\f), carriage return (\r)
* Thứ hai, tôi chỉ định kí tự [\\d](file:///\\d) đại diện cho các số từ 0 đến 9 làm kí tự phân cách. Điều này thì các bạn đã biết.

Chạy chương trình, chúng ta có kết quả như sau:

|  |
| --- |
| I  love  you  so  much.  I  want  to  marry  you  Number of tokens: 10 |

Như vậy, chúng ta cũng được các chuỗi con được tách ra dựa trên các kí tự khoảng trắng và các số.

>> Xem tiếp Phần 10

**Phần 10: Ngắt Chuỗi Với lớp java.util.StringTokenizer**

Trong Phần 8 và Phần 9, chúng ta đã tìm hiểu các cách thức để tách các chuỗi con từ chuỗi cha ban đầu bằng cách sử dụng lớp java.util.Scanner và phương thức String.split()

Chúng ta có một cách thứ 3 cũng được dùng để ngắt một chuỗi ban đầu thành các chuỗi con. Đó là lớp java.util.StringTokenizer.

Lớp java.util.StringTokenizer là một trong những lớp được tạo ra lâu đời nhất trong các lớp thư viện của Java (từ JDK 1.0).

Mặc định, lớp java.util.StringTokenizer cũng sử dụng kí tự khoảng trắng làm kí tự phân cách.

Chúng ta sử dụng lớp java.util.StringTokenizer như sau:

|  |
| --- |
| import java.util.StringTokenizer;  public class Demo {  public static void main(String[] args) {  StringTokenizer stk;  String s = "I love you so much! But I cannot marry you.";  stk = new StringTokenizer(s);  while (stk.hasMoreTokens()) {  System.out.println(stk.nextToken());  }  }  } |

Trong đoạn chương trình trên, tôi có chuỗi như sau:

|  |
| --- |
| String s = "I love you so much! But I cannot marry you."; |

Sau đó, tôi truyền chuỗi này như tham số vào constructor của lớp java.util.StringTokenizer:

|  |
| --- |
| stk = new StringTokenizer(s); |

Tiếp theo, tôi sử dụng vòng lặp while để duyệt tất cả các token hiện có trong chuỗi bằng cách gọi phương thức:

|  |
| --- |
| stk.hasMoreTokens() |

Để kiểm tra còn token nào trong chuỗi không. Mặc định, phương thức này sẽ sử dụng kí tự khoảng trắng làm kí tự phân cách.

Nếu còn token, tôi gọi tiếp phương thức:

|  |
| --- |
| stk.nextToken() |

Để lấy giá trị token đó và in ra ngoài màn hình:

|  |
| --- |
| System.out.println(stk.nextToken()); |

Chạy chương trình, chúng ta có kết quả như sau:

|  |
| --- |
| I  love  you  so  much!  But  I  cannot  marry  you. |

Nếu muốn chỉ định một danh sách các kí tự làm kí tự phân cách (delimiter), chúng ta thêm vào tham số thứ 2 cho constructor của lớp java.util.StringTokenizer:

|  |
| --- |
| import java.util.StringTokenizer;  public class Demo {  public static void main(String[] args) {  StringTokenizer stk;  String s = "I love you so much! But I cannot marry you.";  stk = new StringTokenizer(s, " !");  while (stk.hasMoreTokens()) {  System.out.println(stk.nextToken());  }  }  } |

Trong đoạn chương trình trên, tôi chỉ định kí tự khoảng trắng và dấu chấm than (!) làm kí tự phân cách như sau:

|  |
| --- |
| stk = new StringTokenizer(s, " !"); |

Chạy chương trình, chúng ta có kết quả như sau:

|  |
| --- |
| I  love  you  so  much  But  I  cannot  marry  you. |

Chú ý trong kết quả trên, mặc dù có thời điểm 2 kí tự khoảng trắng và dấu chấm than (!) nằm liền kề nhau (giữa từ much và But), nhưng lớp java.uti.StringTokenizer xem như một kí tự phân cách mà chúng ta không cần phải chỉ định thêm dấu cộng (+) như trong trường hợp lớp java.util.Scanner và phương thức String.split().

Và cho dù có chỉ định thêm dấu cộng (+), thì lớp java.util.StringTokenizer cũng xem dấu cộng (+) là kí tự phân cách, chứ không phải gộp 2 kí tự khoảng trắng và dấu chấm than (!) thành một kí tự phân cách.

Có điều này vì thật sự lớp java.util.StringTokenizer không hỗ trợ các biểu thức mẫu của Regular Expression.

Và đây cũng chính là sự khác biệt lớn nhất giữa lớp java.util.StringTokenizer và 2 cách thức còn lại.

Lớp java.util.StringTokenizer không hỗ trợ Regular Expression vì lớp này được đưa vào từ JDK 1.0. Trong khi đó, phải tới JDK 1.5 thì Regular Expression mới được hỗ trợ trong Java.

Và chính vì không hỗ trợ Regular Expression, do vậy lớp java.util.StringTokenizer không mất thời gian để kiểm tra các kí tự chỉ định làm kí tự phân cách là các kí tự cụ thể hay là thuộc về kí hiệu mẫu qui định trong Regular Expression.

Điều này cho phép nếu chúng ta ngắt các chuỗi con trên một chuỗi ban đầu rất dài thì sử dụng lớp java.util.StringTokenizer sẽ cho hiệu suất tốt hơn nhiều so với lớp java.util.Scanner và phương thức String.split().

Do vậy, chúng ta lưu ý các sự khác biệt giữa các cách thức ngắt chuỗi khác nhau để có sự sử dụng thích hợp cho từng trường hợp cụ thể.

>> Xem tiếp

**Phần 11: Java Regular Expression Engine (java.util.regex.\*): matches(), lookingAt(), find()**

Trong các phần trước, chúng ta đã sử dụng các biểu thức của Regular Expression và phương thức String.matches() để kiểm tra trùng khớp của chuỗi đầu vào với mẫu qui định trước.

Tuy nhiên, phương thức String.matches() chỉ hỗ trợ kiểm tra toàn bộ chuỗi đầu vào xem có khớp với mẫu hay không, chứ không hỗ trợ kiểm tra trùng với một chuỗi con bên trong chuỗi ban nhập vào ban đầu.

Ví dụ như trong các chức năng tìm kiếm, thay thế của các trình soạn thảo văn bản như Microsoft Notepad, Word…chúng ta cần tìm một chuỗi con bên trong toàn bộ chuỗi văn bản dựa trên một tiêu chí được qui định trước. Trong các trường hợp này, thì phương thức String.matches() không đáp ứng được yêu cầu.

Để linh hoạt trong các tình huống như tìm kiếm dữ liệu, kiểm tra thành phần chuỗi con trong chuỗi ban đầu, kiểm tra sự tồn tại của một chuỗi trước khi thực hiện so trùng….thì chúng ta phải sử dụng đến bộ Regular Expression Engine của Java nằm trong package: java.util.regex.\*.

Trong phần này, chúng ta sẽ tìm hiểu 2 lớp là Pattern và Matcher; và 3 phương thức cơ bản nhất trong lớp Matcher của bộ Regular Expression Engine này: matches(), lookingAt(), và find()

Chức năng của 3 phương thức này như sau:

|  |  |
| --- | --- |
| **Phương thức** | **Chức năng** |
| matches() | Kiểm tra một chuỗi xem có trùng với một mẫu qui định trước hay không. Phương thức này hoạt động giống như phương thức String.matches() |
| lookingAt() | Kiểm tra một chuỗi có bắt đầu bằng một chuỗi con nào đó hay không |
| find() | Kiểm tra một chuỗi se có chứa một chuỗi con nào đó hay không. Đây cũng là phương thức thường được sử dụng nhất. |

Trước tiên, chúng ta xem cách sử dụng các class Pattern, Matcher, và phương thức matches() như thế nào.

Chúng ta xem đoạn chương trình sau:

|  |
| --- |
| import java.util.Scanner;  import java.util.regex.\*;  public class Demo {  public static void main(String[] args) {  boolean flag;  Scanner sc = new Scanner(System.in);  Pattern pattern;  Matcher matcher;  do {  String namePattern = "[a-zA-Z\\s]+";  pattern = Pattern.compile(namePattern);    System.out.print("Enter your name: ");  String name = sc.nextLine();  matcher = pattern.matcher(name);    flag = matcher.matches();  if (!flag) System.out.println("Invalid Input!");  } while (!flag);  System.out.println("Valid input");  }  } |

Trong đoạn chương trình trên, chúng ta yêu cầu người dùng nhập vào tên, và kiểm tra xem trên có hợp lệ hay không dựa vào mẫu sau:

|  |
| --- |
| String namePattern = "[a-zA-Z\\s]+"; |

Như chúng ta đã biết, mẫu trên qui định chuỗi đầu vào chỉ chứa các kí tự khoảng trắng, và các kí tự từ a – z có thể chữ HOA hoặc chữ thường. Chuỗi đầu vào phải chứa ít nhất một kí tự.

Tuy nhiên, thay vì sử dụng phương thức String.matches(), thì trong trường hợp này, tôi sử dụng phương thức matches trong lớp Matcher.

Trước tiên, tôi cần khai báo 2 đối tượng của 2 lớp Pattern và Macher:

|  |
| --- |
| Pattern pattern;  Matcher matcher; |

Trước khi có thể sử dụng mẫu để kiểm tra trùng khớp, chúng ta cần đưa mẫu vào phương thức compile() của lớp Pattern:

|  |
| --- |
| pattern = Pattern.compile(namePattern); |

Giống như tên gọi của nó, phương thức compile() này sẽ tiến hành kiểm tra xem mẫu chúng ta đưa vào có chứa các kí tự hợp lệ qui định bởi Regular Expression hay không.

Điều này cho phép tiết kiệm rất nhiều thời gian trong công việc so trùng, tìm kiếm sau này. Vì nếu một mẫu chứa các kí tự không hợp lệ, hoặc sai cú pháp của Regular Expression thì phương thức compile() sẽ thông báo một exception và dừng chương trình lại; mà không cần cố thử so trùng ở các tác vụ sau đó.

Đây là sự khác biệt cơ bản nhất giữ việc sử dụng lớp Pattern và Matcher này so với sử dụng trực tiếp phương thức String.matches().

Phương thức String.matches() vừa kiểm tra cú pháp-kí tự hợp lệ của Regular Expression, vừa kiểm tra so trùng chuỗi mỗi lần chúng ta gọi phương thức String.matches(). Điều này gây mất thời gian, vì thông thường mẫu chỉ qui định một lần ngay từ ban đầu, mà ít khi thay đổi.

Sau khi mẫu đã được kiểm tra xong, chúng ta đưa chuỗi cần kiểm tra vào phương thức matcher() của lớp Pattern:

|  |
| --- |
| matcher = pattern.matcher(name); |

Và để kiểm tra trùng khớp hay không, chúng ta gọi phương thức matches() của lớp Matcher:

|  |
| --- |
| flag = matcher.matches(); |

Phương thức matches() này cũng trả về một giá trị Boolean xác định mẫu có trùng khớp hay không. Do vậy, chúng ta cần gán vào một biến kiểu Boolean để nhận kết quả, và có các xử lý tiếp theo.

Và chú ý, mặc định phương thức matches() này sẽ kiểm tra toàn bộ chuỗi đầu vào xem có khớp hay không.

Các phần còn lại của chương trình thì cũng giống như trong các ví dụ trước chúng ta đã xem.

Chạy chương trình, chúng ta được kết quả như sau:

|  |
| --- |
| Enter your name: nguyen huy 33 hung  Invalid Input!  Enter your name: nguyen huy hung  Valid input |

Chuỗi “nguyen huy 33 hung”: không hợp lệ vì có chứa số 33

Chuỗi “nguyen huy hung”: hoàn toàn hợp lệ vì thỏa mãn các yêu cầu của mẫu qui định ban đầu.

Kế tiếp chúng ta xem cách sử dụng phương thức lookingAt() như thế nào.

Phương thức lookingAt() chỉ kiểm tra xem chuỗi đầu vào có bắt đầu bằng một chuỗi con được qui định trước hay không.

Ví dụ, tôi muốn tên người dùng nhập vào phải bắt đầu bằng chuỗi “nguyen” thì mới hợp lệ. Tôi sử dụng phương thức lookingAt() như sau:

|  |
| --- |
| import java.util.Scanner;  import java.util.regex.\*;  public class Demo {  public static void main(String[] args) {  boolean flag;  Scanner sc = new Scanner(System.in);  Pattern pattern;  Matcher matcher;  do {  String namePattern = "nguyen";  pattern = Pattern.compile(namePattern);    System.out.print("Enter your name: ");  String name = sc.nextLine();  matcher = pattern.matcher(name);    flag = matcher.lookingAt();  if (!flag) System.out.println("Invalid Input!");  } while (!flag);  System.out.println("Valid input");  }  } |

Trước tiên, tôi qui định chuỗi con bắt đầu là “nguyen”

|  |
| --- |
| String namePattern = "nguyen"; |

Sau đó, tôi chỉ cần thay đổi từ phương thức matches() thành phương thức lookingAt():

|  |
| --- |
| flag = matcher.lookingAt(); |

Như vậy, phương thức lookingAt() sẽ kiểm tra chuỗi đầu vào xem có bắt đầu bằng chuỗi con “nguyen” hay không, và trả về giá trị true/false tương ứng.

Chạy chương trình, chúng ta có kết quả như sau:

|  |
| --- |
| Enter your name: huy hung  Invalid Input!  Enter your name: nguyen huy hung  Valid input |

Chuỗi “huy hung”: không hợp lệ, vì không bắt đầu bằng “nguyen”

Chuỗi “nguyen huy hung”: hoàn toàn hợp lệ, vì bắt đầu bằng “nguyen”

Kế tiếp, chúng ta hãy xem cách sử dụng phương thức find() như thế nào.

Chúng ta xem ví dụ sau:

|  |
| --- |
| import java.util.regex.\*;  public class Demo {  public static void main(String[] args) {  Pattern pattern;  Matcher matcher;  String text = "I love you so much! However, I cannot marry you because you are not a human!";  String searchString = "not";  pattern = Pattern.compile(searchString, Pattern.CASE\_INSENSITIVE);  matcher = pattern.matcher(text);  int count = 0;  while (matcher.find()) {  count++;  }  System.out.println("The word was found: " + count + " times");  }  } |

Trong ví dụ trên, tôi có chuỗi văn bản sau:

|  |
| --- |
| String text = "I love you so much! However, I cannot marry you because you are not a human!"; |

Và tôi muốn tìm xem trong chuỗi văn bản trên có chứa bao nhiều từ “not”.

Do vậy, tôi chỉ định mẫu cần tìm:

|  |
| --- |
| String searchString = "not"; |

Sau đó, tôi dùng phương thức find() để tiến hành tìm trong chuỗi văn bản ban đầu. Do phương thức find() sẽ bắt đầu tìm từ đầu cho đến khi kết thúc chuỗi, do vậy chúng ta cần sử dụng vòng lặp while để tìm toàn bộ chuỗi ban đầu:

|  |
| --- |
| while (matcher.find()) {  count++;  } |

Phương thức find() sẽ tiến hành tìm chuỗi con, trong trường hợp của chúng ta là từ “not”, và trả về giá trị true. Sau đó, tiếp tục tìm cho đến khi nào hết chuỗi thì dừng lại, tức là trả về giá trị false.

Trong vòng lặp while, chúng ta sử dụng một biến đếm count để đếm xem phương thức find() tìm thấy từ “not” bao nhiêu lần.

Chạy chương trình, chúng ta có kết quả như sau:

|  |
| --- |
| The word was found: 2 times |

Thay vì chỉ định một chuỗi con cụ thể để tìm kiếm như trong ví dụ trên, chúng ta có thể chỉ định một mẫu để tìm kiếm.

Chúng ta xem ví dụ sau:

|  |
| --- |
| import java.util.regex.\*;  public class Demo {  public static void main(String[] args) {  Pattern pattern;  Matcher matcher;  String text = "I love you 34 so much! However, 24 I cannot marry 45 you because 4 you are not a human!";  String searchString = "\\d";  pattern = Pattern.compile(searchString, Pattern.CASE\_INSENSITIVE);  matcher = pattern.matcher(text);  int count = 0;  while (matcher.find()) {  count++;  }  System.out.println("Found: " + count + " times");  }  } |

Trong đoạn chương trình trên, chuỗi văn bản có thêm các kí tự số:

|  |
| --- |
| String text = "I love you 34 so much! However, 24 I cannot marry 45 you because 4 you are not a human!"; |

Và tôi muốn tìm xem, trong chuỗi văn bản này có chứa các kí tự số hay không, tôi định nghĩa mẫu tìm kiếm đơn giản như sau:

|  |
| --- |
| String searchString = "\\d"; |

Như vậy, phương thức find() sẽ tìm trong chuỗi xem có kí tự số nào không và trả về kết quả true/false tương ứng.

Chạy chương trình, chúng ta có kết quả như sau:

|  |
| --- |
| Found: 7 times |

Có tổng cộng 7 kí tự số được tìm thấy là 3, 4, 2, 4, 4, 5, 4.

Nếu chúng ta muốn phương thức find() xem 2 hay nhiều kí tự số nằm liền kề nhau như một số, nghĩa là cho ra kết quả là 4 số, bao gồm: 43, 24, 45, và 4; thì chúng ta chỉ cần thay đổi mẫu nhu sau:

|  |
| --- |
| String searchString = "\\d+"; |

Chạy chương trình, chúng ta có kết quả như sau:

|  |
| --- |
| Found: 4 times |

Và cuối cùng, nếu chúng ta muốn lấy ra các số đó là số nào khi phương thức find() tìm thấy, thì chúng ta chỉ cần gọi phương thức group() của lớp Matcher trong vòng lặp while:

|  |
| --- |
| while (matcher.find()) {  count++;  System.out.println("Found: " + matcher.group());  } |

Chạy chương trình, chúng ta có kết quả như sau:

|  |
| --- |
| Found: 34  Found: 24  Found: 45  Found: 4  Found: 4 times |

>> Xem tiếp

**Phần 12: Java Regular Expression Engine (java.util.regex.\*): Group và Subgroup**

Trong các phần trước, chúng ta đã làm quen với khái niệm group cho phép gom nhóm một biểu thức mẫu con trong một mẫu cha để người dùng có thể tùy chọn hoặc phải nhập dữ liệu đúng với toàn bộ nhóm, hoặc có thể bỏ qua toàn bộ nhóm kí tự đó.

Hoặc chúng ta đã sử dụng qua khái niệm group để qui định trước danh sách các mục mà người dùng phải nhập, ví dụ như sau:

|  |
| --- |
| String pattern = (music|sport|book|movie) |

Chúng ta dùng cặp dấu ngoặc tròn (()) và chỉ định một danh sách các mục, cách nhau bởi dấu gạch đứng (|). Như vậy, với mẫu trên, người dùng bắt buộc phải nhập một trong các giá trị: music, sport, book, hoặc movie.

Trong phần này, chúng ta sẽ tìm hiểu chi tiết hơn về khái niệm group (nhóm) và sub-group (nhóm con)

Group là một tập hợp các kí tự mẫu tuần tự. Hay nói một cách khác, group cho phép chúng ta xử lý một tập hợp các kí tự mẫu như một đơn vị duy nhất.

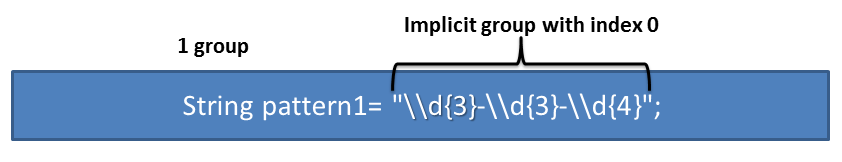
Trong java, một biểu thức Regular Expression có thể có nhiều group. Mỗi group đượ đánh một chỉ số (index) giống như trong mảng (array). Group đầu tiên cũng sẽ có chỉ số là 0.

Group được chia làm 2 loại: group ngầm định (implicit group) và group tường minh (explicit group):

* Các kí tự được đặt trong cặp dấu ngoặc tròn (()) được gọi là group tường minh, như chúng ta đã sử dụng trong các phần trước
* Các kí tự **không** nằm trong cặp dấu ngoặc tròn (()) được gọi là group ngầm định

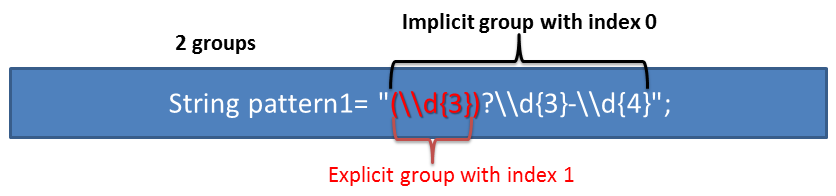
Như vậy, khi chúng ta tạo ra một mẫu và không đặt vào trong cặp dấu ngoặc tròn (()) thì mẫu này thuộc về một nhóm ngầm định, và có chỉ số (index) là 0.

Ví dụ một mẫu trong hình sau:



Trong mẫu trên, chúng ta có 1 group ngầm định với chỉ số là 0.

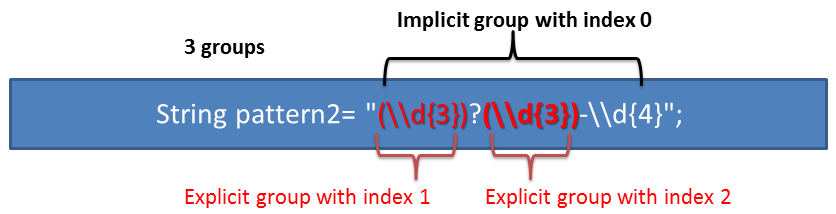
Chúng ta xem ví dụ sau:



Trong mẫu trên, chúng ta có 2 group:

* 1 group ngầm định có chỉ số 0, chính là nhóm chứa toàn bộ mẫu
* 1 group tường minh có chỉ số 1, chính là phần đặt trong cặp dấu ngoặc tròn ((\\d{3}))

Chúng ta xem ví dụ sau:



Trong mẫu trên, chúng ta có 3 group:

* Một group ngầm định có chỉ số 0, chính là nhóm chứa toàn bộ mẫu
* Một group tường minh có chỉ số 1, chính là phần đặt trong cặp dấu ngoặc tròn đầu tiên ((\\d{3}))
* Một group tường minh có chỉ số 2, chính là phần đặt trong cặp dấu ngoặc tròn thứ 2 ((\\d{3}))

Như vậy, chú ý chúng ta có thể nói mẫu trên có tổng cộng 3 nhóm: 1 nhóm ngầm định, và 2 nhóm tường minh.

Chúng ta xem đoạn chương trình sau:

|  |
| --- |
| import java.util.regex.\*;  public class Demo {  public static void main(String[] args) throws Exception {  Pattern pattern;  Matcher matcher;  String searchString;  String text;  text = "Monday 12-9-2013";  searchString = "\\w+ \\d{1,2}-\\d{1,2}-\\d{4}";  pattern = Pattern.compile(searchString, Pattern.CASE\_INSENSITIVE);  matcher = pattern.matcher(text);    while (matcher.find()) {  System.out.println("found: " + matcher.group(0));  }  }  } |

Trong đoạn chương trình trên, tôi định nghĩa một mẫu như sau:

|  |
| --- |
| searchString = "\\w+ \\d{1,2}-\\d{1,2}-\\d{4}"; |

Mẫu này khớp với bất kì định dạng ngày tháng:

* Bắt đầu bằng các kí tự, theo sau là một khoảng trắng
* Theo sau là 1 hoặc 2 số đại diện cho ngày, theo sau là dấu gạch nối (-)
* Theo sau là 1 hoặc 2 số đại diện cho tháng, theo sau là dấu gạch nối (-)
* Theo sau là 4 số đại diện cho năm

Sau đó tôi truyền mẫu này vào cho phương thức Pattern.compile() để tiến hành “biên dịch” mẫu, nghĩa là kiểm tra mẫu có hợp lệ hay không:

|  |
| --- |
| pattern = Pattern.compile(searchString, Pattern.CASE\_INSENSITIVE); |

Chú ý, tôi chỉ định thêm tham số thứ 2 cho phương thức Pattern.compile():

|  |
| --- |
| Pattern.CASE\_INSENSITIVE |

Giá trị này có nghĩa là các kí tượng trong chuỗi đầu vào có thể là chữ HOA hoặc chữ thường.

Tôi để sẵn một chuỗi ngày tháng khớp với mẫu như sau:

|  |
| --- |
| text = "Monday 12-9-2013"; |

Va trong vòng lặp while() tôi sử dụng phương thức Matcher.find() để tìm mẫu trùng trong chuỗi ban đầu, nếu tìm thấy, thì tôi gọi phương thức Matcher.group() để lấy ra nhóm trùng với mẫu:

|  |
| --- |
| while (matcher.find()) {  System.out.println("found: " + matcher.group(0));  } |

Chú ý, trong phương thức Matcher.group() tôi truyền vào giá trị 0, chính là muốn lấy ra chuỗi khớp với group có chỉ số là 0.

Hoặc nếu tôi tôi không truyền giá trị chỉ số cho phương thức Matcher.group() thì mặc định, phương thức này cũng sẽ trả về chuỗi khớp với group có chỉ số là 0.

Do vậy, dòng code:

|  |
| --- |
| System.out.println("found: " + matcher.group(0)); |

Tương đương với:

|  |
| --- |
| System.out.println("found: " + matcher.group()); |

Chạy chương trình, chúng ta có kết quả như sau:

|  |
| --- |
| found: Monday 12-9-2013 |

Chúng ta được toàn bộ chuỗi ban đầu, vì toàn bộ chuỗi này khớp với mẫu nằm trong group ngầm định có chỉ số 0.

Giả sử bây giờ tôi muốn lấy ra từng thành phần:

* Phần ngày, bao gồm cả thứ
* Phần tháng
* Phần năm

Trong trường hợp này, tôi cần đặt từng thành phần trong một group để tôi có thể tách ra khi so trùng mẫu.

Tôi thay đổi chương trình như sau:

|  |
| --- |
| import java.util.regex.\*;  public class Demo {  public static void main(String[] args) throws Exception {  Pattern pattern;  Matcher matcher;  String searchString;  String text;  text = "Monday 12-9-2013";  searchString = "(\\w+ \\d{1,2})-(\\d{1,2})-(\\d{4})";  pattern = Pattern.compile(searchString, Pattern.CASE\_INSENSITIVE);  matcher = pattern.matcher(text);    while (matcher.find()) {  System.out.println("found: " + matcher.group(1));  System.out.println("found: " + matcher.group(2));  System.out.println("found: " + matcher.group(3));  }  System.out.println("There are " + matcher.groupCount() + " groups in the pattern!");  }  } |

Chú ý trong đoạn chương trình trên, tôi đã đặt từng thành phần muốn lấy ra trong từng nhóm:

|  |
| --- |
| searchString = "(\\w+ \\d{1,2})-(\\d{1,2})-(\\d{4})"; |

Như vậy, trong mẫu trên, tôi có tổng cộng 4 nhóm:

* Nhóm đầu tiên, có chỉ số 0, luôn là nhóm ngầm định không nằm trong dấu ngoặc tròn nào
* Nhóm thứ 2, có chỉ số 1, chính là phần nằm trong cặp dấu ngoặc tròn đầu tiên: (\\w+ \\d{1,2})
* Nhóm thứ 3, có chỉ số 2, chính là phần nằm trong cặp dấu ngoặc tròn thứ 2: (\\d{1,2})
* Nhóm thứ 4, có chỉ số 3, chính là phần nằm trong cặp dấu ngoặc tròn thứ 3: (\\d{4})

Như vậy, sau khi so trùng khớp với mẫu, chúng ta muốn lấy ra từng thành phần, tức là từng nhóm, chúng ta chỉ cần cung cấp chỉ số tương ứng cho phương thức Matcher.group() như sau:

|  |
| --- |
| while (matcher.find()) {  System.out.println("found: " + matcher.group(1));  System.out.println("found: " + matcher.group(2));  System.out.println("found: " + matcher.group(3));  } |

Và để đếm xem có bao nhiêu group trong mẫu, chúng ta gọi phương thức Matcher.groupCount():

|  |
| --- |
| System.out.println("There are " + matcher.groupCount() + " groups in the pattern!"); |

Tuy nhiên, lưu ý là phương thức Matcher.groupCount() chỉ trả về số lượng group tường minh, tức 3 group trong ví dụ hiện tại.

Chạy chương trình, chúng ta được kết quả như sau:

|  |
| --- |
| found: Monday 12  found: 9  found: 2013  There are 3 groups in the pattern! |

Như vậy là chúng ta đã tách ra được từng thành phần trong chuỗi ban đầu.

Chúng ta có thể lồng một group vào trong một group khác. Trong trường hợp này, group nằm trong được gọi là sub group (group con).

Ví dụ trong trường hợp trên, tôi muốn tách riêng phần thứ trong phần ngày ra, thì tôi cần đặt toàn bộ phần mẫu khớp ngày vào một group nữa:

|  |
| --- |
| searchString = "((\\w+) \\d{1,2})-(\\d{1,2})-(\\d{4})"; |

Như vậy, hiện tại trong mẫu của tôi đang có tổng cộng 5 nhóm:

* Nhóm đầu tiên, có chỉ số 0, luôn là nhóm ngầm định không nằm trong dấu ngoặc tròn nào
* Nhóm thứ 2, có chỉ số 1, chính là phần nằm trong cặp dấu ngoặc tròn đầu tiên: ((\\w+) \\d{1,2})
* Nhóm thứ 3, có chỉ số 2, bây giờ chính là nhóm con nằm trong nhóm thứ 2: ([\\w](file:///\\w)+)
* Nhóm thứ 4, có chỉ số 3,chính là nhóm nằm trong cặp dấu ngoặc tròn thứ 3: (\\d{1,2})
* Nhóm thứ 5, có chỉ số 4, chính là nhóm nằm trong cặp dấu ngoặc tròn cuối cùng: (\\d{4})

Do chúng ta đã có thêm một nhóm con, vì vậy chúng ta cần gọi thêm phương thức Matcher.group() và cung cấp thêm chỉ số thứ 4 để lấy ra tất cả các nhóm:

|  |
| --- |
| while (matcher.find()) {  System.out.println("found: " + matcher.group(1));  System.out.println("found: " + matcher.group(2));  System.out.println("found: " + matcher.group(3));  System.out.println("found: " + matcher.group(4));  } |

Chạy chương trình, chúng ta có kết quả như sau:

|  |
| --- |
| found: Monday 12  found: Monday  found: 9  found: 2013  There are 4 groups in the pattern! |

Như vậy, chúng ta đã tách ra được từng thành phần mong muốn. Và số lượng nhóm tường minh hiện tại là 4 nhóm.

>> Xem tiếp

**Phần 13: Java Regular Expression Engine (java.util.regex.\*): Boundary Matcher**

Trong các ví dụ trước, khi so trùng một chuỗi, chúng ta chỉ so trùng toàn bộ chuỗi với mẫu qui định trước. Tuy nhiên trong thực tế, đôi khi có các tác vụ mà việc so trùng toàn bộ chuỗi là không khả thi, mà đòi hỏi việc so trùng phải linh hoạt hơn.

Ví dụ như trong các trường hợp:

* Chúng ta muốn tìm một chuỗi con, nhưng chỉ khi chuỗi con này xuất hiện tại vị trí đầu hoặc cuối của chuỗi văn bản
* Hoặc chúng ta muốn so trùng toàn bộ một từ nào đó, chứ không phải một chuỗi con trong 1 từ nào đó. Ví dụ chúng ta muốn tìm từ “is”, chứ không phải chuỗi con “is” trong từ “th**is**”.

Trong các trường hợp như trên, chúng ta phải sử dụng kỹ thuật gọi là boundary matcher (tạm dịch: so trùng biên) thì mới có thể thực hiện tốt tác vụ của chúng ta.

Có một số kí tự mẫu được dùng phổ biến trong kĩ thuật boundary matcher như sau:

|  |  |
| --- | --- |
| **Kí tự mẫu** | **Ý nghĩa** |
| **^** | So trùng bắt đầu từ đầu chuỗi |
| **$** | So trùng cho đến khi kết thúc chuỗi |
| **\b** | So trùng toàn bộ một từ trong một chuỗi. ví dụ như tính năng “Match whole word only” trong các chức năng “Find and replace” của các trình soạn thảo văn bản.  D:\Work\PersonalProject\Block\Tutorial\JavaRegularExpression\boudary_matcher.png |
| **\B** | So trùng một từ bắt đầu bằng một số kí tự nào đó |

Chúng ta xem ví dụ sau:

|  |
| --- |
| import java.util.Scanner;  import java.util.regex.\*;  public class Demo {  public static void main(String[] args) throws Exception {  Pattern pattern;  Matcher matcher;  boolean flag;  String searchString;  Scanner sc = new Scanner(System.in);  do {  String searchPattern = "^is$";  pattern = Pattern.compile(searchPattern);  System.out.print("Enter a string to search: ");  searchString = sc.nextLine();  matcher = pattern.matcher(searchString);  flag = matcher.find();  if (!flag) System.out.println("Not found in the input string!");  if (flag) System.out.println("Found: " + matcher.group());  } while (!flag);  }  } |

Trong đoạn chương trình trên, tôi định nghĩa một mẫu như sau:

|  |
| --- |
| String searchPattern = "^is$"; |

Trong mẫu trên, tôi bắt đầu mẫu bằng dấu mũ (^), theo sau là chuỗi “is”, và theo sau là dấu đô la ($):

* Dấu mũ (^) có nghĩa là so trùng bắt đầu từ đầu chuỗi. Có nghĩa là chuỗi trùng phải bắt đầu bằng chuỗi “is”
* Dấu đô la ($) có nghĩa là so trùng cho đến khi kết thúc chuỗi. Có nghĩa là chuỗi trùng cũng phải kết thúc bằng chuỗi “is”

Như vậy, toàn bộ mẫu của chúng ta qui định toàn bộ chuỗi trùng phải là chuỗi “is”. Và điều này tương đương với việc chúng ta sử dụng phương thức Matcher.matches().

Và chú ý, chúng ta cần sử dụng phương thức find() để tiến hành kiểm tra toàn bộ chuỗi.

Chạy chương trình, chúng ta có kết quả như sau:

|  |
| --- |
| Enter a string to search: this is a string  Not found in the input string!  Enter a string to search: is a string  Not found in the input string!  Enter a string to search: is  Found: is |

Mẫu “is” không được tìm thấy trong chuỗi “this is a string”,vì chúng ta qui định chuỗi trùng phải bắt đầu và kết thúc bằng “is”

Mẫu “is” không được tìm thấy trong chuỗi “is a string”, vì mặc dù chuỗi này bắt đầu bằng “is”, nhưng lại không kết thúc bằng “is”

Mẫu “is” hoàn toàn trùng khớp với chuỗi đầu vào “is”

Giả sử bây giờ tôi thay đổi mẫu tìm kiếm như sau:

|  |
| --- |
| String searchPattern = "^is[\\s\\w]\*"; |

Trong mẫu này, tôi thay phần dấu đo la ($) bằng tổ hợp các kí tự mẫu [\\s\\w]\*.

Như vậy mẫu tìm kiếm này sẽ trùng khớp với bất kì chuỗi nào bắt đầu bằng chuỗi “is” và có thể theo sau là các kí tự khoảng trắng ([\\s](file:///\\s)), hoặc các kí tự từ a đến z, hoặc số từ 0 đến 9.

Chạy chương trình, chúng ta có kết quả như sau:

|  |
| --- |
| Enter a string to search: A string  Not found in the input string!  Enter a string to search: is a string  Found: is a string |

Chuỗi “A string” không khớp với mẫu vì không bắt đầu bằng chuỗi “is”

Chuỗi “is a string” hoàn toàn trùng khớp vì bắt đầu bằng chuỗi “is” và theo sau là các kí tự hợp lệ.

Tôi tiếp tục thay đổi chuỗi tìm kiếm thành mẫu như sau:

|  |
| --- |
| String searchPattern = "[\\s\\w]\*is$"; |

Trong mẫu này, tôi di chuyển tổ hợp kí tự mẫu [\\s\\w]\* ra phía trước và theo sau là chuỗi “is”, cuối cùng là dấu đô la ($). Chú ý, chúng ta không còn kí tự dấu mũ (^) phía trước nữa.

Như vậy, mẫu này sẽ khớp với bất kì chuỗi nào mà kết thúc bằng chuỗi “is”.

Chạy chương trình, chúng ta có kết quả như sau:

|  |
| --- |
| Enter a string to search: This is a string  Not found in the input string!  Enter a string to search: This is  Found: This is |

Chuỗi “This is a string” không trùng khớp vì không kết thúc bằng chuỗi “is”.

Chuỗi “This is” hoàn toàn trùng khớp vì kết thúc bằng chuỗi con “is”.

Kế tiếp, tôi thay đổi chuỗi tìm kiếm như sau:

|  |
| --- |
| String searchPattern = "\\bis\\b"; |

Trong mẫu hiện tại, tôi sử dụng kí hiệu [\\b](file:///\\b) cho phía trước và phía sau chuỗi “is”.

Điều này có nghĩa là trong chuỗi trùng phải chứa chuỗi con độc lập, có thể xem là một từ, là “is”.

Chạy chương trình, chúng ta có kết quả như sau:

|  |
| --- |
| Enter a string to search: This does not have the one  Not found in the input string!  Enter a string to search: This is a string  Found: is |

Chuỗi “This does not have the one” không có chuỗi con nào khớp với mẫu tìm kiếm. Vì chuỗi này không chứa bất kì từ “is” độc lập nào. Chú ý, chuỗi con “is” trong từ “This” không được xem là độc lập, vì trong mẫu tìm kiếm, tôi chỉ định [\\b](file:///\\b) cho cả 2 đầu chuỗi “is”

Chuỗi “This is a string” hoàn toàn hợp lệ vì có chứa toàn bộ từ “is” độc lập, khớp với mẫu chúng ta cần tìm.

Tuy nhiên, nếu tôi thay đổi mẫu tìm kiếm như sau:

|  |
| --- |
| String searchPattern = "\\Bis\\b"; |

Phía trước chuỗi “is”, tôi thau [\\b](file:///\\b) (viết thường) thành [\\B](file:///\\B) (viết hoa).

Như vậy, mẫu của tôi có nghĩa là trong chuỗi đầu vào chỉ cần có từ nào kết thúc là “is”, còn bắt đầu bằng kí tự nào cũng được.

Chạy chương trình, chúng ta có kết quả như sau:

|  |
| --- |
| Enter a string to search: This does not have the one  Found: is |

Chuỗi “This does not have the one” hoàn toàn hợp lệ, vì có từ “This” kết thúc bằng chuỗi con “is”.

Và tôi cũng kết thúc phần boundary matcher ở đây.

Hy vọng các bạn đã hiểu cách sử dụng của các kí tự mẫu trong boundary matcher.

Mặc dù trong tập kí tự mẫu của boundary matcher còn một số kí tự mẫu khác, tuy nhiên các kí tự mẫu trên là các kí tự thường dùng phổ biến nhất.

>> Xem tiếp

**Phần 14: Java Regular Expression Engine (java.util.regex.\*): Backreferences (tạm dịch: tham chiếu ngược lại)**

Trong quá trình sử dụng group trong một biểu thức mẫu của Regular Expression, chúng ta có thể sử dụng cơ chế backreference để so trùng với một mẫu trước đó mà không cần phải viết lại toàn bộ mẫu.

Trong một biểu thức Regular Expression, backreference được chỉ định thông qua kí tự [\\x](file:///\\x) ngay sau một group nào đó. Giá trị x chính là chỉ số của group.

Một trong các ứng dụng của backreference là cho phép chúng ta có thể xác định trong một chuỗi văn bản nào đó, có chứa các từ bị trùng lắp hay không.

Chúng ta xem đoạn chương trình sau:

|  |
| --- |
| import java.util.Scanner;  import java.util.regex.\*;  public class Demo {  public static void main(String[] args) throws Exception {  Pattern pattern;  Matcher matcher;  boolean flag = false;  String searchString;  Scanner sc = new Scanner(System.in);  do {  String searchPattern = "(\\w\\w)\\1";  pattern = Pattern.compile(searchPattern);  System.out.print("Enter a string to search: ");  searchString = sc.nextLine();  matcher = pattern.matcher(searchString);  while(matcher.find()) {  System.out.println("Found duplicate: " + matcher.group());  flag = true;  }  if (!flag) System.out.println("Not found duplicate word");  } while (!flag);  }  } |

Trong đoạn chương trình trên, tôi định nghĩa một mẫu có sử dụng backreference như sau:

|  |
| --- |
| String searchPattern = "(\\w\\w)\\1"; |

Trong mẫu trên tôi định nghĩa một group với 2 kí tự từ (a-z, 0-9); theo sau là [\\1](file:///\\1). Số 1 ở đây có nghĩa là chỉ số của group.

Vì trong mẫu hiện tại có 2 group:

* Luôn luôn có 1 group ngầm định có chỉ số là 0
* Group tường minh nằm trong cặp dấu ngoặc tròn (()) có chỉ số 1

Như vậy, hiện tại tôi đang sử dụng một tham chiếu ngược lại (backreference) đến group có chỉ số 1. Điều này có nghĩa là mẫu này sẽ trùng khớp với bất kì chuỗi nào bắt đầu bằng 2 kí tự từ (a-z, 0-9) và theo sau chính xác cũng là 2 kí tự từ (a-z, 0-9).

Chạy chương trình, chúng ta có kết quả như sau:

|  |
| --- |
| Enter a string to search: This is a string  Not found duplicate word  Enter a string to search: This isis a string  Found duplicate: isis  Enter a string to search: This is abab string  Found duplicate: abab |

Chuỗi “This is a string” không trùng khớp vì không chứa bất kì chuỗi con nào bắt đầu bằng 2 kí tự từ (a-z, 0-9) và theo sau chính xác cũng là 2 kí tự đó.

Chuỗi “This isis a string”: hoàn toàn trùng khớp với mẫu backreference, vì có chứa một chuỗi con “isis” bắt đầu bằng “is”, và theo sau cũng chính xác là 2 từ “is”.

Chuỗi “This is abab string”: cũng hoàn toàn trùng khớp với mẫu backreference, vì có chứa chuỗi con “abab” bắt đầu bằng “ab”, và theo sau cũng chính xác là 2 từ “ab”.

Chúng ta xem một mẫu có sử dụng backreference khác.

Giả sử tôi thay đổi mẫu thành:

|  |
| --- |
| String searchPattern = "(\\w\\w)(\\d\\d)\\2"; |

Trong mẫu này, tôi thêm vào một group tường minh thứ 2 bao gồm 2 kí tự số (([\\d\\d)](file:///\\d\\d))); và theo sau là backreference vào group này ([\\2](file:///\\2)); vì group này có chỉ số là 2.

Điều này có nghĩa mẫu này sẽ khớp với bất kì chuỗi nào:

* Bắt đầu bằng 2 kí tự từ (a-z, 0-9)
* Theo sau là 2 số
* Theo sau chính xác là 2 số trước đó; vì chúng ta sử dụng backreference vào group 2 kí tự số này.

Chạy chương trình, chúng ta có kết quả như sau:

|  |
| --- |
| Enter a string to search: This is23 a string  Not found duplicate word  Enter a string to search: This is2323 a string  Found duplicate: is2323 |

Chuỗi “This is23 a string” không trùng khớp, vì không có chứa bất kì chuỗi con nào bắt đầu bằng 2 kí tự từ, theo sau là 2 số, theo sau chính xác cũng là 2 số đó.

Chuỗi “This is2323 a string”: hoàn toàn trùng khớp, vì có chứa chuỗi con “is2323”, bắt đầu bằng 2 kí tự từ “is”, theo sau là 2 kí tự số “23”, và theo sau cũng chính xác là 2 kí tự số “23”.

Như vậy, bằng việc sử dụng cơ chế backreference thích hợp, chúng ta có thể tránh được việc khai báo lại các mẫu group lặp đi lặp lại.

Chúng ta xem một ví dụ khác:

|  |
| --- |
| import java.util.Scanner;  import java.util.regex.\*;  public class Demo {  public static void main(String[] args) throws Exception {  Pattern pattern;  Matcher matcher;  boolean flag = false;  String searchString;  Scanner sc = new Scanner(System.in);  do {  String searchPattern = "\\b(\\w+)\\s+\\1\\b";  pattern = Pattern.compile(searchPattern);  System.out.print("Enter a string to search: ");  searchString = sc.nextLine();  matcher = pattern.matcher(searchString);  while(matcher.find()) {  System.out.println("Found duplicate: " + matcher.group());  flag = true;  }  if (!flag) System.out.println("Not found duplicate word");  } while (!flag);  }  } |

Trong đoạn chương trình trên, tôi định nghĩa một mẫu có sử dụng backreference và kí tự boundary matcher:

|  |
| --- |
| String searchPattern = "\\b(\\w+)\\s+\\1\\b"; |

Kí tự boundary [\\b](file:///\\b) như chúng ta đã biết là so trùng toàn bộ một nhóm các kí tự nào đó. Trong trường hợp này là các kí tự từ (a-z, 0-9) và theo sau là khác kí tự khoảng trắng.

Backreference trong mẫu của tôi tham chiếu đến group 1, tức là group chứa các kí tự từ (a-z, 0-9).

Như vậy, mẫu này sẽ trùng khớp với bất kì chuỗi nào bắt đầu là các kí tự từ (z-a, 0-9), theo sau là các khoảng trắng, và theo sau chính xác là các nhóm kí tự trước đó.

Chạy chương trình, chúng ta có kết quả như sau:

|  |
| --- |
| Enter a string to search: This string contains contains some duplicate word word  Found duplicate: contains contains  Found duplicate: word word |

Tôi nhập vào một chuỗi “This string contains contains some duplicate word word”, và chương trình tìm thấy 2 chuỗi con có từ trùng lặp: “contains contains” và “word word”

>> Xem tiếp

**Phần 15: Java Regular Expression Engine (java.util.regex.\*): Positive Look-Ahead**

Trong một biểu thức Regular Expression, chúng ta có thể sử dụng kỹ thuật gọi là positive look-ahead để tăng hiệu suất trong quá trình so trùng, tìm kiếm dữ liệu.

Chúng ta hình dung một tình huống như sau: giả sử bạn muốn ghé thăm một người bạn của mình. Tuy nhiên, để chắc chắn bạn của mình có nhà để không mất không đến, rồi lại đi về vì không có bạn mình ở nhà; bạn gọi điện trước (positive look ahead) để chắc chắn người bạn mình định ghé thăm có nhà. Nếu người bạn này có nhà, bạn sẽ ghé thăm. Còn nếu không có nhà, bạn sẽ không mất công đi nữa.

Cơ chế positive look-ahead trong Java Regular Expression cũng hoạt động tương tự.

Cơ chế này cho phép chúng ta kiểm tra chuỗi đầu vào xem có trùng khớp với một mẫu nào đó hay không, trước khi quyết định có kiểm tra phần còn lại của chuỗi ban đầu. Điều này cho phép tiết kiệm nhiều thời gian và tăng hiệu suất vì không phải kiểm tra toàn bộ chuỗi nếu không thỏa mãn một số điều kiện ban đầu.

Ví dụ tôi muốn kiểm tra định dạng của một chuỗi giá trị ISBN của một cuốn sách. Chuỗi ISBN cần kiểm tra có mẫu như sau:

|  |
| --- |
| ISBN-\\d{3}-\\d{4} |

Như vậy, một chuỗi giá trị ISBN bắt buộc phải:

* Bắt đầu bằng chuỗi ISBN, theo sau là dấu gạch nối
* Theo sau là 3 số, theo sau là dấu gạch nối
* Theo sau là 4 số

Với kỹ thuật positive look ahead, mẫu này sẽ kiểm tra chuỗi ISBN đầu vào như sau:

* Nếu chuỗi đầu vào không bắt đầu bằng các kí tự in HOA ISBN, thì quá trình so trùng sẽ kết thúc mà không cần kiểm tra các phần còn lại của chuỗi
* Nếu chuỗi đầu vào bắt đầu bằng các kí tự chữ in HOA ISBN, mẫu sẽ tiếp tục kiểm tra các phần còn lại. Nếu một trong các thành phần tiếp theo không khớp với thành phần qui định trong mẫu, quá trình so trùng sẽ kết thúc mà không cần kiểm tra các phần còn lại trong chuỗi ban đầu.

Chúng ta xem ví dụ sau:

|  |
| --- |
| import java.util.regex.\*;  public class Demo {  public static void main(String[] args) throws Exception {  Pattern pattern;  Matcher matcher;  String searchString;  String isbn = "ISBN-123-4567";  searchString = "(?=ISBN)\\w{4}\\-\\d{3}-\\d{4}";  pattern = Pattern.compile(searchString, Pattern.CASE\_INSENSITIVE);  matcher = pattern.matcher(isbn);  while (matcher.find()) {  System.out.println("Matched: " + matcher.group());  }  }  } |

Trong đoạn chương trình trên, tôi định nghĩa một mẫu như sau:

|  |
| --- |
| searchString = "(?=ISBN)\\w{4}\\-\\d{3}-\\d{4}"; |

Trong mẫu tên, tôi sử dụng kĩ thuật positive look ahead cho chuỗi ISBN.

Kỹ thuật positive look ahead phải đặt trong 1 group, theo sau là dấu chấm hỏi (?), theo sau là dấu bằng (=), và theo sau là mẫu mà chúng ta muốn xử lý cho kỹ thuật positive look ahead.

Mẫu của chúng ta có sẽ trùng với các chuỗi:

* Bắt đầu bằng 4 kí tự từ (a-z, 0-9). Tuy nhiên, do chúng ta chỉ định các kí tự IN HOA trong positive look a head là ISBN, chuỗi trùng phải chứa các kí tự này. Nếu chuỗi đầu vào không bắt đầu hoặ có chứa các kí tự ISBN, thì quá trình so trùng sẽ kết thúc mà không cần kiểm tra các thành phần còn lại
* Theo sau là dấu gạch nối và 3 chữ số
* Theo sau là dấu gạch nối và 4 chữ số

Trog chương trình trên, tôi tạo sẵn một chuỗi ISBN đầu vào như sau:

|  |
| --- |
| String isbn = "ISBN-123-4567"; |

Chạy chương trình, chúng ta có kết quả như sau:

|  |
| --- |
| Matched: ISBN-123-4567 |

Chuỗi đầu vào “ISBN-123-4567” hoàn toàn trùng khớp với mẫu vì thỏa mãn đầy đủ các qui định sẵn.

Một trong các ứng dụng khác của positive look ahead là chúng ta có thể dùng để kiểm tra độ mạnh – yếu của một mật khẩu (password) của người dùng nhập vào.

Ví dụ, khi người dùng đăng kí một tài khoản mới và được yêu cầu cung cấp mật khẩu (password) đăng nhập. Để tăng độ bảo mật, tôi muốn mật khẩu người dùng phải thỏa mãn các tiêu chí sau:

* Ít nhất 4 kí tự
* Tối đa 8 kí tự
* Có ít nhất 1 kí tự số (0-9)
* Có ít nhất 1 kí tự IN HOA

Dựa trên các yêu cầu, tôi định nghĩa ra mẫu trong đoạn chương trình sau:

|  |
| --- |
| import java.util.Scanner;  import java.util.regex.\*;  public class Demo {  public static void main(String[] args) throws Exception {  Pattern pattern;  Matcher matcher;  boolean flag;  String password;  Scanner sc = new Scanner(System.in);  do {  String passwordPattern = "^(?=.\*\\d+.\*)(?=.\*[A-Z]+.\*)\\w{4,8}$";  pattern = Pattern.compile(passwordPattern);    System.out.print("Enter your password: ");  password = sc.nextLine();  matcher = pattern.matcher(password);    flag = matcher.matches();  if (!flag) System.out.println("Invalid password!");  if (flag) System.out.println("Password is valid!");  } while (!flag);  }  } |

Trong đoạn chương trình trên, tôi định nghĩa một mẫu dành cho password như sau:

|  |
| --- |
| String passwordPattern = "^(?=.\*\\d+.\*)(?=.\*[A-Z]+.\*)\\w{4,8}$"; |

Trong mẫu trên, tôi định nghĩa 2 mẫu positive look ahead:

* (?=.\*\\d+.\*): phần look ahead này qui định chuỗi đầu vào phải có ít nhất 1 kí tự số từ 0 đến 9
* (?=.\*[A-Z]+.\*): phần look ahead này qui định chuỗi đầu vào phải có ít nhất một kí tự IN HOA từ a đến z

Và để qui định chuỗi đầu vào ít nhất 4 kí tự, tối đa 8 kí tự, chúng ta dùng mẫu quen thuộc: \\w{4,8}

Chú ý, chúng ta cần dùng dấu mũ (^) ở đầu mẫu, và dấu đô la ($) ở cuối mẫu, để đảm bảo toàn bộ chuỗi đầu vào phải khớp với mẫu.

Chạy chương trình, chúng ta có kết quả như sau:

|  |
| --- |
| Enter your password: ajkd3  Invalid password!  Enter your password: Akijj  Invalid password!  Enter your password: 8Aj  Invalid password!  Enter your password: 8abCd  Password is valid! |

Mật khẩu “ajkd3” không hợp lệ vì không có kí tự IN HOA nào

Mật khẩu “Akijj” không hợp lệ vì có kí tự IN HOA nhưng không có kí tự số nào

Mật khẩu “8Aj” không hợp lệ vì có kí tự IN HOA, có kí tự số, nhưng không đủ số lượng kí tự tối thiểu là 4 kí tự

Mật khẩu “8abCd” hoàn toàn hợp lệ vì có kí tự số (8), có kí tự IN HOA (C), tối thiểu 4 kí tự, và không vượt quá 8 kí tự.

Thật ra với yêu cầu như trên chúng ta không cần sử dụng kỹ thuật positive look ahead mà vẫn có thể đảm bảo các qui định của bài toán ban đầu.

Tuy nhiên, với kỹ thuật positive look ahead, thì Java Regular Expression Engine sẽ ưu tiên kiểm tra trước các mẫu chúng ta chỉ định trong group của look ahead, nếu một trong các mẫu look ahead không thỏa mãn, thì quá trình so trùng, hoặc tìm kiếm sẽ dừng lại ngay. Điều này cho phép tiết kiệm thời gian xử lý và làm tăng hiệu suất chương trình.

>> Xem tiếp

**Phần 16:** **Java Regular Expression Engine (java.util.regex.\*): Negative Look Ahead**

Trong phần trước, chúng ta đã tìm hiểu về kỹ thuật positive look a head. Về cơ bản, kỹ thuật positive look ahead kiểm tra một chuỗi có khớp với một mẫu qui định hay không trước khi quyết định tiếp tục kiểm tra phần còn lại trong chuỗi ban đầu.

Nói một cách khác, kỹ thuật positive look ahead cho phép kiểm tra sự tồn tại của một chuỗi con (dựa trên một mẫu qui định sẵn) trong một chuỗi ban đầu. Nếu tồn tại, sẽ tiếp tục thực hiện các công việc kế tiếp. Ngược lại, toàn bộ quá trình kiểm tra, so trùng, tìm kiếm…sẽ dừng lại.

Chúng ta có một kỹ thuật khác liên quan gọi là negative look ahead.

Cùng thuộc về nhóm look ahead, do vậy, kỹ thuật negative look ahead giống với kỹ thuật positive look ahead ở điểm là cả 2 kỹ thuật này đều kiểm tra trước (look ahead) xem một chuỗi có khớp với một mẫu đã được qui định hay không, trước khi tiếp tục kiểm tra phần còn lại của chuỗi ban đầu.

Tuy nhiên, kỹ thuật negative look ahead khác với kỹ thuật positive look ahead ở điểm là thay vì kiểm tra sự tồn tại, thì kỹ thuật này kiểm tra sự không tồn tại, hay không trùng khớp, của một chuỗi con trước khi có tiếp tục kiểm tra phần còn lại trong chuỗi.

Một trong những ứng dụng phổ biến của kỹ thuật negative look ahead là chúng ta dùng để kiểm tra một văn bản nào đó có KHÔNG chứa một từ được qui định trước hay không.

Ví dụ như trong các phần mềm lọc từ khóa dành có các website có nội dung không lành mạnh. Chúng ta muốn kiểm tra nội dung một trang web:

* Nếu chỉ có chứa từ khóa “sex” thì không thông báo gì cả. Vì có thể các trang web lành mạnh như tin tức, mạng xã hội….cũng có chứa từ này.
* Nếu chứa từ khóa “porn” thì sẽ thông báo, vì tỷ lệ các trang web có chứa từ khóa này có các nội dung không lành mạnh là rất cao.

Chúng ta xem ví dụ sau:

|  |
| --- |
| import java.util.\*;  import java.util.regex.\*;  public class Demo {  public static void main(String[] args) throws Exception {  List<String> documentList = new ArrayList<>();  documentList.add("My sex is female");  documentList.add("Sex education can be taught in high school");  documentList.add("This sentence contain a porn word");  documentList.add("These are sex and porn websites");  Pattern pattern;  Matcher matcher;  String searchString;  searchString = "(?!.\*porn).\*sex.\*";  pattern = Pattern.compile(searchString, Pattern.CASE\_INSENSITIVE);  for (String text : documentList) {  matcher = pattern.matcher(text);  while (matcher.find()) {  System.out.println("Matched: " + matcher.group());  }  }  }  } |

Trong đoạn chương trình trên, tôi định nghĩa một mẫu có sử dụng kỹ thuật negative look ahead như sau:

|  |
| --- |
| searchString = "(?!.\*porn).\*sex.\*"; |

Để qui định một mẫu negative look ahead, chúng ta cũng đặt trong một group và bắt đầu là dấu chấm hỏi (?), theo sau là dấu chấm than (!), và theo sau là mẫu mà chúng ta KHÔNG muốn tồn tại trong chuỗi. Trong trường hợp này là từ khóa “porn”.

Chú ý, phía trước từ khóa “porn”, tôi có sử dụng dấu chấm (.) theo sau là dấu sao (\*) để chỉ định từ khóa “porn” có thể xuất hiện tại bất cứ vị trí nào trong chuỗi văn bản ban đầu.

Theo sau mẫu negative look ahead là mẫu thông thường: .\*sex.\*

Mẫu này có nghĩa là chuỗi văn bản phải có từ khóa “sex” và từ khóa này có thể xuất hiện tại bất kỳ vị trí nào trong văn bản.

Trong chương trình tôi sử dụng ArrayList để lưu một số chuỗi làm ví dụ:

|  |
| --- |
| List<String> documentList = new ArrayList<>();  documentList.add("My sex is female");  documentList.add("Sex education can be taught in high school");  documentList.add("This sentence contain a porn word");  documentList.add("These are sex and porn websites"); |

Trong sanh sách, chúng ta thấy có chuỗi chỉ có chứa từ khóa “sex”; có chuỗi chứa cà 2 từ khóa “sex” và “porn”.

Chạy chương trình, chúng ta có kết quả như sau:

|  |
| --- |
| Matched: My sex is female  Matched: Sex education can be taught in high school |

Chương trình chỉ lấy ra các chuỗi mà chỉ chứa từ khóa “sex” mà thôi. Còn các chuỗi có chứa thêm từ “porn” thì được loại bỏ vì không trùng với mẫu negative look ahead.

**Phần 17: Java Regular Expression Engine (java.util.regex.\*): Positive Look Behind**

Trong 2 phần trước, chúng ta đã tìm hiểu về các kỹ thuật look ahead để kiểm tra sự tồn tại (positive look ahead) hoặc không tồn tại (negative look ahead) của một chuỗi con nào đó trước khi kiểm tra phần còn lại của chuỗi ban đầu.

Trong phần này, chúng ta sẽ tìm hiểu về kỹ thuật look behind. Look behind cũng có 2 kỹ thuật: positive look behind và negative look behind.

Kỹ thuật positive look behind được dùng để kiểm tra sự tồn tại của một mẫu nào đó nằm phía trước của một vị trí hiện tại trong chuỗi ban đầu. Nói một cách khác, kỹ thuật này dùng để kiểm tra một chuỗi có bắt đầu bằng một mẫu chuỗi cụ thể nào đó hay không.

Kỹ thuật negative look behind thực hiện công việc ngược lại. Nghĩa là kiểm tra sự KHÔNG tồn tại của một chuỗi trước một chuỗi khác hay không.

Trong phần này, chúng ta sẽ xem các ví dụ sử dụng kỹ thuật positive look behind.

Giả sử tôi có một danh sách tên những người bạn như sau:

|  |
| --- |
| Nguyễn Huy Hùng  Trần nguyễn Phi Long  Nguyễn Hoài Nam  Nguyễn Trần  Peter Nguyễn |

Và tôi muốn tìm những người nào có họ là “Nguyễn” và in ra tên đầy đủ của những người này.

Chúng ta xem đoạn chương trình sau:

|  |
| --- |
| import java.util.\*;  import java.util.regex.\*;  public class Demo {  public static void main(String[] args) throws Exception {  List<String> friendList = new ArrayList<>();  friendList.add("Nguyễn Huy Hùng");  friendList.add("Trần nguyễn Phi Long");  friendList.add("Nguyễn Hoài Nam");  friendList.add("Nguyễn Trần");  friendList.add("Peter Nguyễn");  Pattern pattern;  Matcher matcher;  String searchString;  searchString = "(?<=^Nguyễn).\*";  pattern = Pattern.compile(searchString, Pattern.CASE\_INSENSITIVE);  for (String text : friendList) {  matcher = pattern.matcher(text);  while (matcher.find()) {  System.out.println("Matched: " + text);  }  }  }  } |

Trong đoạn chương trình trên, tôi dùng một ArrayList để lưu danh sách tên những người bạn:

|  |
| --- |
| List<String> friendList = new ArrayList<>();  friendList.add("Nguyễn Huy Hùng");  friendList.add("Trần nguyễn Phi Long");  friendList.add("Nguyễn Hoài Nam");  friendList.add("Nguyễn Trần");  friendList.add("Peter Nguyễn"); |

Sau đó, tôi định nghĩa một mẫu như sau:

|  |
| --- |
| searchString = "(?<=^Nguyễn).\*"; |

Trong mẫu trên, tôi có sử dụng kỹ thuật positive look behind: (?<=^Nguyễn)

Chúng ta cũng bắt đầu kỹ thuật positive look behind bằng việc sử dụng group, theo sau là dấu chấm hỏi (?), theo sau là dấu nhỏ hơn (<), theo sau là dấu bằng (=), cuối cùng theo sau là dấu mũ (^) và từ “Nguyễn”.

Tôi sử dụng dấu mũ (^) trước từ “Nguyễn” để đảm bảo không có bất kì từ nào khác phía trước từ “Nguyễn”, vì yêu cầu ban đầu là tìm ra những người có họ là “Nguyễn”, chứ không phải những người mà tên có chứa từ “Nguyễn”.

Theo sau mẫu positive look behind là một mẫu thông thường mà chúng ta đã quen thuộc: .\*

Mẫu này cho phép tên có thể có bất kỳ kí tự nào cũng được. Các bạn có thể thay đổi mẫu này cho phù hợp với yêu cầu của mình.

Chúng ta hết sức lưu ý cơ chế làm việc của positive look behind như sau:

* Tên đầu tiên trong danh sách được lấy ra để kiểm tra: Nguyễn Huy Hùng
* Từ đầu tiên trong trên này sẽ được lấy ra để kiểm tra là từ “Nguyễn”.
* Từ này không có từ “Nguyễn” nào trước đó, do đó nó được bỏ qua vì không khớp với mẫu
* Từ kế tiếp được lấy ra để kiểm tra là từ “Huy”. Từ này có từ phía trước là “Nguyễn”, như vậy, hoàn toàn trùng khớp với mẫu qui định trong positive look behind.
* Từ kế tiếp được lấy ra để kiểm tra là “Hùng”. Từ này không có từ “Nguyễn” phía trước, tức là không thỏa mãn mẫu qui định nên được bỏ qua
* Như vậy, sau khi kiểm tra tên đầu tiên, chúng ta tìm được một lần trùng khớp tại vị trí chuỗi con “Huy”.
* Và quá trình tương tự được lặp lại cho các tên còn lại trong danh sách.

Do vậy, nếu chạy chương trình, chúng ta có kết quả như sau:

|  |
| --- |
| Matched: Nguyễn Huy Hùng  Matched: Nguyễn Hoài Nam  Matched: Nguyễn Trần |

2 tên “Trần nguyễn Phi Long” và “Peter Nguyễn” không thỏa mãn mẫu positive look behind vì mặc dù có từ “Nguyễn” (không phân biệt hoa – thường), nhưng 2 từ “n/Nguyễn” này lại có các từ khác ở phía trước.

Do chúng ta sử dụng dấu mũ (^) phía trước từ “Nguyễn” trong mẫu nên không được có bất kỳ kí tự nào phía trước từ này.

Và nếu giả sử bây giờ tôi muốn tìm ra các tên có chứa từ “Nguyễn”, không phân biệt là tên, họ, hay tên đệm, thì tôi chỉ cần bỏ dấu mũ (^) trong mẫu positive look behind đi:

|  |
| --- |
| searchString = "(?<=Nguyễn).\*"; |

Chạy chương trình, chúng ta có kết quả như sau:

|  |
| --- |
| Matched: Nguyễn Huy Hùng  Matched: Trần nguyễn Phi Long  Matched: Nguyễn Hoài Nam  Matched: Nguyễn Trần  Matched: Peter Nguyễn |

>> Xem tiếp

**Phần 18: Java Regular Expression Engine (java.util.regex.\*): Negative Look Behind**

Trong phần trước, chúng ta đã sử dụng kĩ thuật positive look behind trong việc kiểm tra sự tồn tại của một chuỗi trước một chuỗi khác.

Ngược lại với positive look behind, kỹ thuật negative look behind sẽ kiểm tra sự KHÔNG tồn tại của một chuỗi trước một chuỗi khác.

Chúng ta xem đoạn chương trình sau:

|  |
| --- |
| import java.util.\*;  import java.util.regex.\*;  public class Demo {  public static void main(String[] args) throws Exception {  List<String> friendList = new ArrayList<>();  friendList.add("David Anderson");  friendList.add("Jonathan David San");  friendList.add("David Gorge");  friendList.add("David M. Sandre");  friendList.add("Peter Tran");  Pattern pattern;  Matcher matcher;  String searchString;  searchString = "(?<![rs])an";  pattern = Pattern.compile(searchString, Pattern.CASE\_INSENSITIVE);  for (String text : friendList) {  matcher = pattern.matcher(text);  while (matcher.find()) {  System.out.println("Matched: " + text);  }  }  }  } |

Trong đoạn chương trình trên, tôi có một danh sách tên như sau:

|  |
| --- |
| List<String> friendList = new ArrayList<>();  friendList.add("David Anderson");  friendList.add("Jonathan David San");  friendList.add("David Gorge");  friendList.add("David M. Sandre");  friendList.add("Peter Tran"); |

Giả sử tôi muốn tìm tất cả các tên nào có chứa chuỗi con “**an**”. Nhưng chuỗi con “**an**” KHÔNG được có các kí tự phía trước là “**r**” hoặc “**s**”.

Dựa vào yêu cầu trên, tôi định nghĩa một mẫu negative look behind như sau:

|  |
| --- |
| searchString = "(?<![rs])an"; |

Để chỉ định một mẫu là negative look hehind thì chúng ta sử dụng dấu chấm than (!) thay vì dấu bằng (=) như trong positive look hehind.

Trong mẫu negative look behind, tôi dùng [rs], có nghĩa là KHÔNG được bắt đầu bằng kí tự hoặc “**r**” hoặc “**s**” nếu sau đó là chuỗi con “**an**”.

Chạy chương trình, chúng ta có kết quả như sau:

|  |
| --- |
| Matched: David Anderson  Matched: Jonathan David San |

2 tên này khớp với mẫu vì cùng chứa chuỗi con “**an**” (Anderson, Jonathan); và cả 2 chuỗi con này đều không có kí tự “**r**” hoặc “**s**” phía trước.

Tên “**David M. Sandre**” không thỏa mãn mẫu vì mặc dù có chứa chuỗi con “**an**”, nhưng chuỗi con này lại có kí tự “**S**” phía trước.

Tên “**Peter Tran**” cũng không thỏa mãn mẫu vì mặc dù có chứa chuỗi con “**an**”, nhưng chuỗi này lại có kí tự “**r**” phía trước.

**>> Xem tiếp**

**Phần 19: Java Regular Expression Engine (java.util.regex.\*): Greedy – Reluctant (Non-greedy) Quantifiers**

Trong phần này, chúng ta sẽ tìm hiểu về 2 kỹ thuật gọi là greedy và reluctant quantifiers.

Kỹ thuật greedy quantifier cho phép một biểu thức regular expression cố gắng so trùng các kí tự một cách tối đa nhất có thể.

Thật ra chúng ta đã sử dụng qua các kí tự mẫu trong kỹ thuật greedy quantifier.

Đây là các kí tự mẫu thuộc về nhóm greedy quantifier:

|  |  |
| --- | --- |
| Greedycharacters | Mô tả |
| ? | Trùng khớp 0 hoặc 1 lần |
| \* | Trùng khớp 0 hoặc nhiều lần |
| + | Trùng khớp 1 hoặc nhiều lần |
| {n} | Trùng khớp n lần |
| {n,} | Trùng khớp tối thiểu n lần |
| {n,m} | Trùng khớp tối thiểu n lần, tối đa m lần |

Chúng ta xem ví dụ để thấy rõ greedy và reluctant quantifier ảnh hưởng như thế nào đến các biểu thức mẫu:

|  |
| --- |
| import java.util.regex.\*;  public class Demo {  public static void main(String[] args) throws Exception {  String regex = "(.\*)(\\d+)";  Pattern pattern = Pattern.compile(regex);  String text = "The order number is 8983";  Matcher matcher = pattern.matcher(text);  while (matcher.find()){  System.out.println(matcher.group(1));  System.out.println(matcher.group(2));  }  }  } |

Trong đoạn chương trình trên, tôi có một chuỗi như sau:

|  |
| --- |
| String text = "The order number is 8983"; |

Tôi muốn tách phần số 8983 từ chuỗi ban đầu ra, vì vậy tôi định nghĩa một mẫu như sau:

|  |
| --- |
| String regex = "(.\*)(\\d+)"; |

Trong mẫu trên, tôi có 2 group tường minh:

Group 1: (.\*): trong group này tôi sử dụng dấu chấm (.) và kí tự greedy quantifier là dấu sao (\*), có nghĩa group này sẽ trùng với bất kỳ kí tự nào với số lương không cố định

Group 2: ([\\d](file:///\\d)+): group này sẽ trùng với các chữ số từ 0 đến 9. Trong group này, chúng ta cũng sử dụng một kí hiệu của greedy quantifier là dấu +, có nghĩa là trùng khớp nhiều hơn 1 kí tự số.

Trong vòng lặp while(), tôi gọi phương thức group() 2 lần với 2 chỉ số tương ứng để lấy ra 2 group trùng khớp:

|  |
| --- |
| while (matcher.find()){  System.out.println(matcher.group(1));  System.out.println(matcher.group(2));  } |

Tuy nhiên, chạy chương trình chúng ta có kết quả như sau:

|  |
| --- |
| The order number is 898  3 |

Group 1 trả về chuỗi con “The order number is 898”

Group 2 trả về chuỗ số 3, chứ không phải toàn bộ số 8983 như chúng ta mong đợi.

Chúng ta có kết quả này, vì cơ chế hoạt động của greedy quantifier như sau:

* Do trong group 1 có sử dụng greedy quantifier là dấu sao (\*), và dấu này khớp với tất cả mọi kí tự nên toàn bộ chuỗi đã được group 1 bắt lại.
* Sau khi group 1 giữ lại toàn bộ chuỗi, group 2 sẽ tiến hành so trùng với chuỗi. Và vì group 2 cũng có sử dụng một greedy quantifier là dấu cộng (+) do vậy, group 2 cũng sẽ cố gắng so trùng tối đa có thể.
* Tuy nhiên, do tại thời điểm này, tất cả mọi kí tự đều đang được giữ lại bởi group 1, do vậy, group 1 sẽ lần lượt nhả từng kí tự ra để group 2 có thể so trùng. Nếu group 2 trùng với bất kì kí tự nào, thì group 2 sẽ giữ lại kí tự đó.
* Tại thời điểm này có 2 tình huống có thể xảy ra:
* Tình huống thứ 1: group 2 tìm thấy 1 kí tự trùng khớp. Khi đó, group 1 sẽ dừng việc nhả kí tự ra và toàn bộ quá trình so trùng cũng sẽ kết thúc
* Tình huống 2: group 2 không trùng với bất kì kí tự nào nhả ra bởi group1. Khi đó group 1 sẽ thu lại tất cả các kí tự đã nhả ra và toàn bộ quá trình so trùng cũng sẽ kết thúc.
* Ví dụ của chúng ta rơi vào tình huống 1, vì khi group 1 nhả kí tự đầu tiên ra là số 3. Số 3 hoàn toàn trùng khớp với group 2. Do vậy, group 1 dừng việc nhả tiếp kí tự và toàn bộ quá trình so trùng cũng sẽ kết thúc.

Và đó là lý do tại sao chúng ta có kết quả như trên.

Tuy nhiên lưu ý là nếu giả sử group 2 tìm thấy một số kí tự trùng khớp và một số kí tự không trùng khớp, thì các kí tự không trùng khớp với group 2 này sẽ không được thu lại bởi group 1.

Chúng ta có thể kiểm tra điều này bằng cách thay đổi chuỗi đầu vào như sau:

|  |
| --- |
| String text = "The order 8983 number is"; |

Tôi di chuyển số 8983 ra phía trước chuỗi “number is”.

Chạy chương trình, chúng ta có kết quả như sau:

|  |
| --- |
| The order 898  3 |

Group 1 chỉ trả về chuỗi “The order 898”.

Group 2 vẫn trả về số “3”.

Group 1 chỉ trả về chuỗi như trên, vì khi group 1 nhả ra các kí tự trong chuỗi “ number is”, không có bất kỳ kí tự nào khớp với group 2 do group 2 chỉ khớp với kí tự số.

Nhưng khi group 1 nhả tới số 3, thì số 3 hoàn toàn khớp với group 2. Group 2 giữ lại số 3 và quá trình so trùng cũng kết thúc. Và group sẽ không thu lại chuỗi không trùng là “ number is” do group 2 đã tìm thấy 1 kí tự trùng khớp.

Như vậy, để giải quyết tình huống là chúng ta muốn group 2 lấy được toàn bộ số 8983, thì chúng ta cần:

* Chỉ định cho group 1 so trùng các kí tự ở mức tối thiểu nhất
* Chỉ định cho group 2 so trùng các kí tự ở mức tối đa nhất

Để chỉ định group 1 so trùng các kí tự ở mức tối thiểu nhất, chúng ta cần áp dụng kỹ thuật reluctant quantifier cho group 1, thay vì greedy quantifier.

Và group 2 chúng ta vẫn giữ nguyên là greedy quantifier do chúng ta muốn group 2 so trùng các kí tự ở mức nhiều nhất có thể.

Để sử dụng kỹ thuật reluctant quantifier, chúng ta thêm dấu chấm hỏi (?) ngay sau kí tự greedy quantifier như sau:

|  |
| --- |
| String regex = "(.\*?)(\\d+)"; |

Tôi thay đổi lại thành chuỗi ban đầu như sau:

|  |
| --- |
| String text = "The order number is 8983"; |

Mẫu của chúng ta lúc này sẽ hoạt động như sau:

* Trước tiên, group 1 cũng sẽ so trùng và giữ toàn bộ chuỗi ban đầu
* Sau đó group 2 sẽ tiến hành so trùng. Nhưng do tại thời điểm này, không còn kí tự nào do tất cả đang bị group 1 giữ lại, do vậy, group 1 sẽ lần lượt nhả từng kí tự để group 2 so trùng.
* Kí tự đầu tiên là số 3 được nhả ra, trùng khớp với group 2, do vậy group 2 giữ lại kí tự này.
* Và do group 1 bây giờ là reluctant quantifier, do vậy, group 1 tiếp tục nhả kí tự tiếp theo là số 8. Số 8 cũng trùng với group 2, do vậy group 2 cũng sẽ giữ lại số 8 này.
* Quá trình này tiếp tục cho đến khi group 1 nhả ra một kí tự mà không trùng với group 2, chính là kí tự “s”. Và lúc này, toàn bộ quá trình so trùng cũng sẽ kết thúc.

Do vậy, chạy chương trình chúng ta có kết quả như sau:

|  |
| --- |
| The order number is  8983 |

Và như vậy, chúng ta lưu ý cơ chế hoạt động của greedy và reluctant quantifiers để có thể sử dụng thích hợp trong từng tình huống cụ thể.

>> Xem tiếp

**Phần 20: Java Regular Expression Engine (java.util.regex.\*): Possessive Quantifiers**

Trong phần trước, chúng ta đã tìm hiểu về cơ chế hoạt động, cũng như sự khác biệt giữa greedy và reluctant quantifiers.

Trong phần này, chúng ta sẽ tìm hiểu về một kỹ thuật cũng có cơ chế hoạt động gần giống với 2 kỹ thuật trên gọi là possessive quantifier.

Để thấy sự khác biệt giữa possessive với 2 kỹ thuật greedy và reluctant quantifier, chúng ta xem ví dụ sau.

Đây là ví dụ tôi sử dụng trong phần trước:

|  |
| --- |
| import java.util.regex.\*;  public class Demo {  public static void main(String[] args) throws Exception {  String regex = "(.\*?)(\\d+)";  Pattern pattern = Pattern.compile(regex);  String text = "The order number is 8983";  Matcher matcher = pattern.matcher(text);  while (matcher.find()){  System.out.println(matcher.group(1));  System.out.println(matcher.group(2));  }  }  } |

Trước tiên, tôi nhắc lại cơ chế hoạt động của greedy và reluctant quantifier trong ví dụ trên:

* Trước tiên, group 1 sẽ so trùng và giữ toàn bộ chuỗi ban đầu
* Sau đó group 2 sẽ tiến hành so trùng. Nhưng do tại thời điểm này, không còn kí tự nào do tất cả đang bị group 1 giữ lại, do vậy, group 1 sẽ lần lượt nhả từng kí tự để group 2 so trùng.
* Kí tự đầu tiên là số 3 được nhả ra, trùng khớp với group 2, do vậy group 2 giữ lại kí tự này.
* Và do group 1 bây giờ là reluctant quantifier, do vậy, group 1 tiếp tục nhả kí tự tiếp theo là số 8. Số 8 cũng trùng với group 2, do vậy group 2 cũng sẽ giữ lại số 8 này.
* Quá trình này tiếp tục cho đến khi group 1 nhả ra một kí tự mà không trùng với group 2, chính là kí tự “s”. Và lúc này, toàn bộ quá trình so trùng cũng sẽ kết thúc.

Do vậy, chạy chương trình chúng ta có kết quả như sau:

|  |
| --- |
| The order number is  8983 |

Kỹ thuật possesstive quantifier giống với greedy ở chỗ là ban đầu cũng so trùng và giữ lại các kí tự trùng khớp. Nhưng khác ở chỗ là possessive không nhả các kí tự đã trùng khớp cho các group khác có cơ hội so trùng.

Chúng ta chỉ định một mẫu possessive bằng các thêm vào dấu cộng (+) ngay phía sau kí tự greedy:

|  |
| --- |
| String regex = "(.\*+)(\\d+)"; |

Như vậy, trong group 1 hiện tại chúng ta đang dùng possessive quantifier.

Và mẫu của chúng ta lúc này hoạt động như sau:

* Trước tiên group 1 sẽ so trùng và giữ lại toàn bộ chuỗi, vì group 1 khớp với tất cả các kí tự trong chuỗi.
* Sau đó, group 2 sẽ tiến hành so trùng với các kí tự trong chuỗi. Tuy nhiên, hiện tại không còn kí tự nào do tất cả đã bị giữ bởi group 1.
* Và do group 1 đang sử dụng possessive quantifier, do đó group 1 sẽ không nhả ra bất kì kí tự nào để group 2 so trùng.
* Group 2 không tìm được bất kì kí tự nào so trùng, do vậy group 2 sẽ trả về kết quả không trùng, và điều này dẫn đến toàn bộ biểu thức mẫu sẽ không trùng.

Do vậy, nếu chạy chương trình, chúng ta không nhận được chuỗi nào trả về cả.

Các bạn có thể hỏi là như vậy chúng ta dùng possessive quantifier để làm gì?

Một trong các ứng dụng của possessive quantifier là để tăng hiệu suất xử lý cho các chuỗi dài dựa trên một mẫu nào đó.

Chúng ta xem ví dụ sau:

|  |
| --- |
| import java.util.regex.\*;  public class Demo {  public static void main(String[] args) throws Exception {  String regex = "(.++)(\\d{4})";  Pattern pattern = Pattern.compile(regex);  String text = "This is a very long string that does not end with digits "  + "This is a very long string that does not end with digits "  + "This is a very long string that does not end with digits "  + "This is a very long string that does not end with digits "  + "This is a very long string that does not end with digits "  + "This is a very long string that does not end with digits "  + "This is a very long string that does not end with digits "  + "This is a very long string that does not end with digits "  + "This is a very long string that does not end with digits "  + "This is a very long string that does not end with digits "  + "This is a very long string that does not end with digits "  + "This is a very long string that does not end with digits "  + "This is a very long string that does not end with digits "  + "This is a very long string that does not end with digits "  + "This is a very long string that does not end with digits "  + "This is a very long string that does not end with digits "  + "This is a very long string that does not end with digits "  + "This is a very long string that does not end with digits "  + "This is a very long string that does not end with digits "  + "This is a very long string that does not end with digits "  + "This is a very long string that does not end with digits";  long start = System.currentTimeMillis();  Matcher matcher = pattern.matcher(text);  if (matcher.find()) {  System.out.println(matcher.group(1));  } else {  System.out.println("No match!");  }  long end = System.currentTimeMillis();  System.out.println("Time elapsed: " + (end - start) + " ms");  }  } |

Trong đoạn chương trình trên, tôi có một chuỗi khá dài.

Tôi định nghĩa một mẫu như sau:

|  |
| --- |
| String regex = "(.+)(\\d{4})"; |

Trong mẫu trên, tôi có 2 group tường minh:

* Group 1 sử dụng greedy quantifier; và nhóm này khớp với tất cả mọi kí tự
* Group 2 sử dụng kí tự \\d{4}, có nghĩa sẽ khớp với các số có 4 chữ số.

Trước khi chạy phần xử lý tìm kiếm, tôi lấy ra thời gian hiện tại:

|  |
| --- |
| long start = System.currentTimeMillis(); |

Sauk hi xử lý xong, tôi cũng lấy ra thời gian hiện tại và trừ đi thời gian trước đó để có được thời gian xử lý:

|  |
| --- |
| long end = System.currentTimeMillis();  System.out.println("Time elapsed: " + (end - start) + " ms"); |

Chạy chương trình, chúng ta có kết quả sau:

|  |
| --- |
| No match!  Time elapsed: 93 ms |

Không có kết quả trùng, và thời gian xử lý là 93 mili giây.

Có kết quả này là do:

* Trước tiên group 1 so trùng và giữ lại toàn bộ chuỗi.
* Sau đó group 2 sẽ tiến hành so trùng. Tại thời điểm này, không còn kí tự nào do toàn bộ chuỗi đang bị giữ bởi group 1
* Group 1 hiện đang là greedy, do vậy group 1 sẽ lần lượt nhả từng kí tự cho group 2 so trùng.
* Sau khi group 1 đã nhả hết các kí tự, vẫn không có kí tự nào trùng với group 2, do không có kí tự số nào.
* Do group 2 không tìm được kí tự trùng nào, group 1 cũng không thu lại tất cả các kí tự đã nhả ra. Và kết quả cuối cùng là không tìm được chuỗi trùng nào.

Tổng thời gian xử lý là 93 mili giây, do group 1 phải nhả ra từng kí tự nên mất khá nhiều thời gian.

Bây giờ tôi sẽ thay đổi mẫu thành possessive quantifier bằng cách thêm vào một dấu cộng:

|  |
| --- |
| String regex = "(.++)(\\d{4})"; |

Chạy chương trình, chúng ta có kết quả như sau:

|  |
| --- |
| No match!  Time elapsed: 31 ms |

Cũng không có kết quả trùng, nhưng thời gian xử lý chỉ có 31 mili giây.

Có kết quả này là do:

* Trước tiên group 1 so trùng và giữ lại toàn bộ chuỗi.
* Sau đó group 2 sẽ tiến hành so trùng. Tại thời điểm này, không còn kí tự nào do toàn bộ chuỗi đang bị giữ bởi group 1
* Group 1 hiện đang là possessive, do vậy group 1 sẽ KHÔNG nhả từng kí tự cho group 2 so trùng.
* Do group 2 không tìm được kí tự thỏa mãn nào, và kết quả cuối cùng là không tìm được chuỗi trùng nào.

Tổng thời gian xử lý là 31 mili giây, do group 1 KHÔNG phải nhả ra từng kí tự cho group 2 so trùng nên không mất nhiều thời gian.

Như vậy, chúng ta thấy mặc dù cho ra cùng kết quả, nhưng possessive cho thời gian xử lý nhanh hơn so với greedy quantifier.

Do vậy, chúng ta nên sử dụng possessive nếu có thể để tận dụng ưu điểm về xử lý của biểu thức mẫu.