EBook Gratis

APRENDIZAJE guava

Free unaffiliated eBook created from
Stack Overflow contributors.

#guava

Tabla de contenido

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: guava

It is an unofficial and free guava ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official guava.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Empezando con la guayaba

Observaciones

Esta sección proporciona una descripción general de qué es la guayaba y por qué un desarrollador puede querer usarla.

También debe mencionar cualquier tema importante dentro de la guayaba y vincular a los temas relacionados. Dado que la Documentación para guayaba es nueva, es posible que deba crear versiones iniciales de esos temas relacionados.

Versiones

Versión	Fecha de lanzamiento
r01%	2009-09-15
r02	2010-01-04
r03	2010-04-09
r04	2010-04-27
r05	2010-05-28
r06	2010-07-07
7.0	2010-09-22
8.0	2011-01-27
9.0	2011-04-07
10.0	2011-09-28
11.0	2011-12-18
12.0	2012-04-30
13.0	2012-08-03
14.0	2013-02-25
15.0	2013-09-06
16.0	2014-01-17
17.0	2014-04-22

Versión	Fecha de lanzamiento
18.0	2014-08-25
19.0	2015-12-09

% no incluyó Google Collections, que existía por separado en ese momento

Nota: Las versiones 1.0 a 11.0 requieren JDK 1.5 o más reciente. Las versiones 12.0 a 20.0 requieren JDK 1.6 o más reciente. Se espera que la versión 21.0 requiera JDK 1.8 o más reciente.

Examples

Preparar

La dependencia de la guayaba se puede agregar a su proyecto Java utilizando cualquier sistema de compilación.

Maven

```
<dependency>
  <groupId>com.google.guava</groupId>
  <artifactId>guava</artifactId>
  <version>19.0</version>
</dependency>
```

Gradle:

```
dependencies {
  compile 'com.google.guava:guava:19.0'
}
```

Hiedra

```
<dependency org="com.google.guava" name="guava" rev="19.0" />
```

Construir

```
compile.with 'com.google.guava:guava:jar:19.0'
```

Dependencia manual

También puedes descargar JARs manualmente desde la página de lanzamiento de Guava para las clases, fuentes y javadocs.

Tenga en cuenta que para Guava 12.0 a 20.0 se requiere JDK 1.6 o posterior. Ver la lista de versiones para más información. Los usuarios de Guayaba que se dirigen a Java 5 deben usar el puerto trasero Guava JDK5 . Esto incluye a los usuarios que se dirigen a Android Froyo y

anteriores.

Lea Empezando con la guayaba en línea: https://riptutorial.com/es/guava/topic/4391/empezando-con-la-guayaba

Capítulo 2: I / O

Examples

Manejo de InputStreams y OutputStreams existentes

Leyendo el contenido de un `InputStream` como una matriz de `byte` :

```
// Reading from a file
try (InputStream in = new FileInputStream("in.dat")) {
  byte[] content = ByteStreams.toByteArray(in);
  // do something with content
}
```

Copiando un `InputStream` a un `OutputStream` :

```
// Copying the content from a file in.dat to out.dat.
try (InputStream in = new FileInputStream("in.dat");
     OutputStream out = new FileOutputStream("out.dat")) {
  ByteStreams.copy(in, out);
}
```

Nota: para copiar archivos directamente, es mejor usar `Files.copy(sourceFile, destinationFile)` .

Leyendo una matriz de `byte` predefinida completa desde un `InputStream` :

```
try (InputStream in = new FileInputStream("in.dat")) {
  byte[] bytes = new byte[16];
  ByteStreams.readFully(in, bytes);
  // bytes is totally filled with 16 bytes from the InputStream.
} catch (EOFException ex) {
  // there was less than 16 bytes in the InputStream.
}
```

Saltando `n` bytes del `InputStream` :

```
try (InputStream in = new FileInputStream("in.dat")) {
  ByteStreams.skipFully(in, 20);
  // the next byte read will be the 21st.
  int data = in.read();
} catch (EOFException e) {
  // There was less than 20 bytes in the InputStream.
}
```

Creando un `OutputStream` que descarta todo lo que se escribe en él:

```
try (InputStream in = new FileInputStream("in.dat");
     OutputStream out = ByteStreams.nullOutputStream()) {
  ByteStreams.copy(in, out);
  // The whole content of in is read into... nothing.
}
```

Manejo de lectores y escritores existentes

Leyendo el contenido de un `Reader` como una `String` :

```
// Reading from a file
try (Reader reader = new FileReader("in.txt")) {
  String content = CharStreams.toString(reader);
  // do something with content
}
```

Leyendo el contenido de un `Reader` como una lista de contenidos de línea:

```
try (Reader reader = new FileReader("in.txt")) {
  List<String> lines = CharStreams.readLines(reader);
  for (String line: lines) {
    // Do something with line
  }
}
```

Copiando un `Reader` a un `Writer` :

```
try (Reader reader = new FileReader("in.txt");
     Writer writer = new FileWriter("out.txt")) {
  CharStreams.copy(reader, writer);
}
```

Nota: para copiar archivos directamente, es mejor usar Files.copy (sourceFile, destinationFile).

Saltando `n` bytes del `Reader` :

```
try (Reader reader = new FileReader("in.txt")) {
  CharStreams.skipFully(reader, 20);
  // The next char read will be the 21st.
} catch (EOFException e) {
  // There was less than 20 chars in the Reader.
}
```

Creando un `Writer` que descarta todo lo que está escrito en él:

```
try (Reader reader = new FileReader("in.txt");
     Writer writer = CharStreams.nullWriter()) {
  CharStreams.copy(reader, writer);
  // The whole content of reader is read into... nothing.
}
```

Fuentes y sumideros

Las fuentes y los sumideros son objetos que saben cómo abrir corrientes.

	Bytes	Los caracteres
Leyendo	`ByteSource`	`CharSource`

	Bytes	Los caracteres
Escritura	`ByteSink`	`CharSink`

Creando fuentes y sumideros.

Nota: para todos los ejemplos, considere `UTF_8` como si se estableciera la siguiente importación:

```
import static java.nio.charset.StandardCharsets.UTF_8;
```

Leyendo de un archivo

```
ByteSource dataSource = Files.asByteSource(new File("input.dat"));
CharSource textSource = Files.asCharSource(new File("input.txt"), UTF_8);
```

Escribiendo en un archivo

```
ByteSink dataSink = Files.asByteSink(new File("output.dat"));
CharSink textSink = Files.asCharSink(new File("output.txt"), UTF_8);
```

Leyendo desde una URL

```
ByteSource dataSource = Resources.asByteSource(url);
CharSource textSource = Resources.asCharSource(url, UTF_8);
```

Lectura de datos en memoria

```
ByteSource dataSource = ByteSource.wrap(new byte[] {1, 2, 3});
CharSource textSource = CharSource.wrap("abc");
```

Convertir de bytes a caracteres

```
ByteSource originalSource = ...
CharSource textSource = originalSource.asCharSource(UTF_8);
```

Convertir de caracteres a bytes

(De guayaba 20 en adelante)

```
CharSource originalSource = ...
ByteSource dataSource = originalSource.asByteSource(UTF_8);
```

Usando fuentes y sumideros

Operaciones comunes

Abriendo un arroyo

```
InputStream inputStream = byteSource.openStream();
OutputStream outputStream = byteSink.openStream();
Reader reader = charSource.openStream();
Writer writer = charSink.openStream();
```

Abriendo un flujo en buffer

```
InputStream bufferedInputStream = byteSource.openBufferedStream();
OutputStream bufferedOutputStream = byteSink.openBufferedStream();
BufferedReader bufferedReader = charSource.openBufferedStream();
Writer bufferedWriter = charSink.openBufferedStream();
```

Operaciones de fuente

Leyendo de una fuente:

```
ByteSource source = ...
byte[] bytes = source.read();

CharSource source = ...
String text = source.read();
```

Leyendo líneas de una fuente:

```
CharSource source = ...
ImmutableList<String> lines = source.readLines();
```

Leyendo la primera línea de una fuente:

```
CharSource source = ...
String firstLine = source.readFirstLine();
```

Copiando de una fuente a un sumidero:

```
ByteSource source = ...
ByteSink sink = ...
source.copyTo(sink);

CharSource source = ...
CharSink sink = ...
source.copyTo(sink);
```

Uso tipico

```
CharSource source = ...
try (Reader reader = source.openStream()) {
  // use the reader
}
```

Lea I / O en línea: https://riptutorial.com/es/guava/topic/6929/i---o

Capítulo 3: Instrumentos de cuerda

Examples

Comprobando una cadena de caracteres no deseados

Como desarrollador, con frecuencia te encuentras tratando con cadenas que no son creadas por tu propio código.

A menudo, estos serán suministrados por bibliotecas de terceros, sistemas externos o incluso usuarios finales. La validación de cadenas de procedencia poco clara se considera una de las características distintivas de la programación defensiva y, en la mayoría de los casos, deseará rechazar la entrada de cadenas que no cumpla con sus expectativas.

Un caso bastante común es en el que solo querría permitir caracteres alfanuméricos en una cadena de entrada, así que usaremos eso como ejemplo. En Java simple, los dos métodos siguientes sirven para el mismo propósito:

```
public static boolean isAlphanumeric(String s) {
    for (char c : s.toCharArray()) {
        if (!Character.isLetterOrDigit(c)) {
            return false;
        }
    }

    return true;
}
```

```
public static boolean isAlphanumeric(String s) {
    return s.matches("^[0-9a-zA-Z]*$");
}
```

La primera versión convierte la cadena en una matriz de caracteres, y luego utiliza el método estático `isLetterOrDigit` clase `Character` para determinar si los caracteres contenidos en la matriz son alfanuméricos o no. Este enfoque es predecible y legible, aunque un poco detallado.

La segunda versión utiliza una expresión regular para lograr el mismo propósito. Es más conciso, pero puede ser un tanto enigmático para los desarrolladores con un conocimiento limitado o nulo de expresiones regulares.

Guava presenta la clase `CharMatcher` para tratar este tipo de situaciones. Nuestra prueba alfanumérica, utilizando guayaba, se vería de la siguiente manera:

```
import static com.google.common.base.CharMatcher.javaLetterOrDigit;

/* ... */

public static boolean isAlphanumeric(String s) {
    return javaLetterOrDigit().matchesAllOf(s);
```

```
}
```

El cuerpo del método contiene solo una línea, pero en realidad hay muchas cosas aquí, así que vamos a desglosar un poco más las cosas.

Si echa un vistazo a la API de la clase `CharMatcher` de Guava, notará que implementa la interfaz `Predicate<Character>` . Si creara una clase que implementa `Predicate<Character>` usted mismo, podría verse algo como esto:

```
import com.google.common.base.Predicate;

public class AlphanumericPredicate implements Predicate<Character> {
    @Override
    public boolean apply(Character c) {
        return Character.isLetterOrDigit(c);
    }
}
```

En Guava, al igual que en otros lenguajes de programación y bibliotecas que se adaptan a un estilo funcional de programación, un predicado es una construcción que evalúa una entrada dada a verdadero o falso. En la interfaz `Predicate<T>` Guava, esto se hace evidente por la presencia del único método `boolean apply(T t)` . La clase `CharMatcher` se basa en este concepto y evaluará un carácter o secuencia de caracteres para verificar si coinciden o no con los criterios establecidos por la instancia de `CharMatcher` utilizada.

Guava actualmente proporciona los siguientes coincidencias de caracteres predefinidos:

Matcher	Descripción
`any()`	Coincide con cualquier personaje.
`none()`	No coincide con los personajes.
`javaDigit()`	Coincide con los dígitos, de acuerdo con la definición de Java.
`javaUpperCase()`	Coincide con cualquier carácter en mayúsculas, de acuerdo con la definición de Java.
`javaLowerCase()`	Coincide con cualquier carácter en minúscula, según la definición de Java.
`javaLetter()`	Coincide con cualquier letra, de acuerdo con la definición de Java.
`javaLetterOrDigit()`	Coincide con cualquier letra o dígito, de acuerdo con la definición de Java.
`javaIsoControl()`	Coincide con cualquier carácter de control ISO, de acuerdo con la definición de Java.
`ascii()`	Coincide con cualquier carácter en el conjunto de caracteres ASCII.

Matcher	Descripción
`invisible()`	Coincide con los caracteres que no son visibles, de acuerdo con el estándar de Unicode.
`digit()`	Coincide con cualquier dígito, de acuerdo con la especificación Unicode.
`whitespace()`	Coincide con cualquier carácter de espacio en blanco, de acuerdo con la especificación de Unicode.
`breakingWhitespace()`	Coincide con cualquier carácter de espacio en blanco de última hora, según la especificación de Unicode.
`singleWidth()`	Coincide con cualquier carácter de ancho único.

Si ha leído la tabla anterior, indudablemente ha notado la cantidad de definición y especificación involucrada en la determinación de qué caracteres pertenecen a una determinada categoría. El enfoque de Guava, hasta ahora, ha sido proporcionar envoltorios de `CharMatcher` para una serie de categorías de caracteres definidas por Java, y puede consultar la API de la clase de `Character` de Java para obtener más información sobre estas categorías. Por otro lado, Guava intenta proporcionar una cantidad de instancias de `CharMatcher` que están en línea con la especificación actual de Unicode. Para los detalles esenciales, consulte la documentación de la API de `CharMatcher` .

Volviendo a nuestro ejemplo de verificación de una cadena en busca de caracteres no deseados, los siguientes métodos de `CharMatcher` proporcionan las capacidades que necesita para verificar si el uso del carácter de una cadena determinada cumple con sus requisitos:

- `boolean matchesNoneOf(CharSequence sequence)`
  Devuelve verdadero si ninguno de los caracteres en la cadena del argumento coincide con la instancia de `CharMatcher` .

- `boolean matchesAnyOf(CharSequence sequence)`
  Devuelve verdadero si al menos un carácter en la cadena del argumento coincide con la instancia de `CharMatcher` .

- `boolean matchesAllOf(CharSequence sequence)`
  Devuelve verdadero si todos los caracteres en la cadena del argumento coinciden con la instancia de CharMatcher.

Encontrar y contar caracteres en una cadena

Para ayudarlo a encontrar y contar caracteres en una cadena, `CharMatcher` proporciona los siguientes métodos:

- `int indexIn(CharSequence sequence)`
  Devuelve el índice del primer carácter que coincide con la instancia de `CharMatcher` .
  Devuelve -1 si no coincide el carácter.

- `int indexIn(CharSequence sequence, int start)`
  Devuelve el índice del primer carácter después de la posición de inicio especificada que coincide con la instancia de `CharMatcher` . Devuelve -1 si no coincide el carácter.

- `int lastIndexIn(CharSequence sequence)`
  Devuelve el índice del último carácter que coincide con la instancia de `CharMatcher` . Devuelve -1 si no coincide el carácter.

- `int countIn(CharSequence sequence)`
  Devuelve el número de caracteres que coinciden con la instancia de `CharMatcher` .

Usando estos métodos, aquí hay una aplicación de consola simple llamada `NonAsciiFinder` que toma una cadena como un argumento de entrada. Primero, imprime el número total de caracteres no ASCII contenidos en la cadena. Posteriormente, imprime la representación Unicode de cada carácter no ASCII que encuentra. Aquí está el código:

```
import com.google.common.base.CharMatcher;

public class NonAsciiFinder {
    private static final CharMatcher NON_ASCII = CharMatcher.ascii().negate();

    public static void main(String[] args) {
        String input = args[0];
        int nonAsciiCount = NON_ASCII.countIn(input);

        echo("Non-ASCII characters found: %d", nonAsciiCount);

        if (nonAsciiCount > 0) {
            int position = -1;
            char character = 0;

            while (position != NON_ASCII.lastIndexIn(input)) {
                position = NON_ASCII.indexIn(input, position + 1);
                character = input.charAt(position);

                echo("%s => \\u%04x", character, (int) character);
            }
        }
    }

    private static void echo(String s, Object... args) {
        System.out.println(String.format(s, args));
    }
}
```

Observe en el ejemplo anterior cómo puede simplemente invertir un `CharMatcher` llamando a su método de `negate` . De manera similar, el `CharMatcher` continuación coincide con todos los caracteres de doble ancho y se crea negando el `CharMatcher` predefinido para los `CharMatcher` de un solo ancho.

```
final static CharMatcher DOUBLE_WIDTH = CharMatcher.singleWidth().negate();
```

La ejecución de la aplicación `NonAsciiFinder` produce el siguiente resultado:

```
$> java NonAsciiFinder "Maître Corbeau, sur un arbre perché"
Non-ASCII characters found: 2
î => \u00ee
é => \u00e9
```

```
$> java NonAsciiFinder "□□や□□び□む□の□"
NonASCII characters found: 11
□ => \u53e4
□ => \u6c60
や => \u3084
□ => \u86d9
□ => \u98db
び => \u3073
□ => \u8fbc
む => \u3080
□ => \u6c34
の => \u306e
□ => \u97f3
```

Eliminar caracteres no deseados de una cadena

El ejemplo Comprobación de una cadena en busca de caracteres no deseados , describe cómo probar y rechazar cadenas que no cumplen ciertos criterios. Obviamente, rechazar la entrada directamente no siempre es posible, y algunas veces solo tienes que conformarte con lo que recibes. En estos casos, un desarrollador cauteloso intentará desinfectar las cadenas proporcionadas para eliminar cualquier carácter que pueda hacer que el procesamiento continúe.

Para eliminar, recortar y reemplazar los caracteres no deseados, el arma elegida nuevamente será la clase `CharMatcher` de Guava.

Quitando personajes

Los dos métodos de `CharMatcher` de interés en esta sección son:

- `String retainFrom(CharSequence sequence)`
  Devuelve una cadena que contiene todos los caracteres que coincidieron con la instancia de `CharMatcher` .

- `String removeFrom(CharSequence sequence)`
  Devuelve una cadena que contiene todos los caracteres que no coincidieron con la instancia de `CharMatcher` .

Como ejemplo, usaremos `CharMatcher.digit()` , una instancia de `CharMatcher` predefinida que, como era de esperar, solo coincide con los dígitos.

```
String rock = "1, 2, 3 o'clock, 4 o'clock rock!";

CharMatcher.digit().retainFrom(rock); // "1234"
CharMatcher.digit().removeFrom(rock); // ", , o'clock, o'clock rock!"
CharMatcher.digit().negate().removeFrom(rock); // "1234"
```

La última línea en este ejemplo ilustra que `removeFrom` es en realidad la operación inversa de `retainFrom` . Invocar `retainFrom` en un `CharMatcher` tiene el mismo efecto que invocar `removeFrom` en una versión negada de ese `CharMatcher` .

Recorte de caracteres iniciales y finales

La eliminación de caracteres iniciales y finales es una operación muy común, que se utiliza con mayor frecuencia para recortar espacios en blanco de cadenas. El `CharMatcher` de Guava ofrece estos métodos de recorte:

- `String trimLeadingFrom(CharSequence sequence)`
  Elimina todos los caracteres `CharMatcher` que coincidan con la instancia de `CharMatcher` .
- `String trimTrailingFrom(CharSequence sequence)`
  Elimina todos los caracteres finales que coinciden con la instancia de `CharMatcher` .
- `String trimFrom(CharSequence sequence)`
  Elimina todos los caracteres `CharMatcher` y finales que coinciden con la instancia de `CharMatcher` .

Cuando se usa con `CharMatcher.whitespace()` , estos métodos se ocuparán de todas sus necesidades de recorte de espacios en blanco:

```
CharMatcher.whitespace().trimFrom("   Too much space   "); // returns "Too much space"
```

Reemplazo de personajes

A menudo, las aplicaciones reemplazarán los caracteres que no están permitidos en una situación determinada con un carácter de marcador de posición. Para reemplazar los caracteres en una cadena, la API de `CharMatcher` proporciona los siguientes métodos:

- `String replaceFrom(CharSequence sequence, char replacement)`
  Reemplaza todas las apariciones de caracteres que coincidan con la instancia de `CharMatcher` con el carácter de reemplazo proporcionado.
- `String replaceFrom(CharSequence sequence, CharSequence replacement)` Reemplaza todas las apariciones de caracteres que coinciden con la instancia de `CharMatcher` con la secuencia de caracteres de reemplazo proporcionada (cadena).
- `String collapseFrom(CharSequence sequence, char replacement)`
  Reemplaza grupos de caracteres consecutivos que coinciden con la instancia de `CharMatcher` con una sola instancia del carácter de reemplazo proporcionado.
- `String trimAndCollapseFrom(CharSequence sequence, char replacement)`
  Se comporta igual que `collapseFrom` , pero los grupos coincidentes al principio y al final se eliminan en lugar de reemplazarse.

Veamos un ejemplo que demuestra cómo difiere el comportamiento de estos métodos. Digamos que estamos creando una aplicación que le permite al usuario especificar nombres de archivos de salida. Para sanear la entrada proporcionada por el usuario, creamos una instancia de `CharMatcher` que es una combinación del espacio en blanco predeterminado `CharMatcher` y un `CharMatcher` personalizado que especifica un conjunto de caracteres que preferiríamos evitar en nuestros nombres de archivo.

```
CharMatcher illegal = CharMatcher.whitespace().or(CharMatcher.anyOf("<>:|?*\"/\\"));
```

Ahora, si invocamos los métodos de reemplazo discutidos de la siguiente manera en un nombre de archivo que está en extrema necesidad de limpieza:

```
String filename = "<A::12> first draft???";

System.out.println(illegal.replaceFrom(filename, '_'));
System.out.println(illegal.collapseFrom(filename, '_'));
System.out.println(illegal.trimAndCollapseFrom(filename, '_'));
```

Veremos la salida a continuación en nuestra consola.

```
_A__12___first_draft___
_A_12_first_draft_
A_12_first_draft
```

Dividir una cadena en una lista

Para dividir cadenas, Guava introduce la clase `Splitter` .

¿Por qué no usar las capacidades de división de Java?

Como regla general, Guava no duplica la funcionalidad que está disponible en Java. ¿Por qué entonces necesitamos una clase `Splitter` adicional? ¿Los métodos de división en la clase `String` de Java no nos proporcionan todas las mecánicas de división de cadenas que necesitaremos?

La forma más fácil de responder a esa pregunta es con un par de ejemplos. En primer lugar, trataremos con el siguiente dúo de pistoleros:

```
String gunslingers = "Wyatt Earp+Doc Holliday";
```

Para tratar de dividir al legendario representante de la ley y su amigo dentista, podemos intentar lo siguiente:

```
String[] result = gunslingers.split("+"); // wrong
```

En el tiempo de ejecución, sin embargo, nos enfrentamos a la siguiente excepción:

```
Exception in thread "main" java.util.regex.PatternSyntaxException:
Dangling meta character '+' near index 0
```

Después de una máscara facial involuntaria, recordamos rápidamente que `String` método de división de `String` toma una expresión regular como un argumento, y que el carácter `+` se usa como cuantificador en expresiones regulares. La solución es entonces escapar del carácter `+` , o encerrarlo en una clase de carácter.

```
String[] result = gunslingers.split("\\+");
String[] result = gunslingers.split("[+]");
```

Habiendo resuelto exitosamente el problema, pasamos a los tres mosqueteros.

```
String musketeers = ",Porthos , Athos ,Aramis,";
```

La coma no tiene un significado especial en las expresiones regulares, por lo que vamos a contar los mosqueteros aplicando el método `String.split()` y obteniendo la longitud de la matriz resultante.

```
System.out.println(musketeers.split(",").length);
```

Lo que produce el siguiente resultado en la consola:

```
4
```

Cuatro? Dado el hecho de que la cadena contiene una coma al principio y una al final, un resultado de cinco habría estado dentro del ámbito de las expectativas normales, ¿pero cuatro? Como resultado, el comportamiento del método de `split` de Java es preservar el inicio, pero descartar las cadenas vacías, por lo que el contenido real de la matriz es `["", "Porthos ", " Athos ", "Aramis"]` .

Ya que no necesitamos cadenas vacías, al principio ni al final, vamos a filtrarlas con un bucle:

```
for (String musketeer : musketeers.split(",")) {
    if (!musketeer.isEmpty()) {
        System.out.println(musketeer);
    }
}
```

Esto nos da la siguiente salida:

```
Porthos
 Athos
Aramis
```

Como puede ver en la salida anterior, los espacios adicionales antes y después de los separadores de coma se han conservado en la salida. Para evitar eso, podemos recortar los espacios innecesarios, que finalmente darán el resultado deseado:

```
for (String musketeer : musketeers.split(",")) {
    if(!musketeer.isEmpty()) {
        System.out.println(musketeer.trim());
    }
}
```

(Alternativamente, también podríamos adaptar la expresión regular para incluir los espacios en blanco que rodean los separadores de coma. Sin embargo, tenga en cuenta que los espacios iniciales antes de la primera entrada o los espacios finales después de la última entrada aún se conservarán).

Después de leer los ejemplos anteriores, no podemos dejar de concluir que dividir cadenas con Java es un poco molesto en el mejor de los casos.

Dividir cuerdas con guayaba

La mejor manera de demostrar cómo Guava convierte las cuerdas divididas en una experiencia relativamente sin dolor, es tratar las mismas dos cadenas nuevamente, pero esta vez utilizando la clase `Splitter` de Guava.

```
List<String> gunslingers = Splitter.on('+')
        .splitToList("Wyatt Earp+Doc Holliday");
```

```
List<String> musketeers = Splitter.on(",")
        .omitEmptyStrings()
        .trimResults()
        .splitToList(",Porthos , Athos ,Aramis,");
```

Como puede ver en el código anterior, `Splitter` expone una API fluida y le permite crear instancias a través de una serie de métodos de fábrica estáticos:

- `static Splitter on(char separator)`
  Le permite especificar el separador como un carácter.
- `static Splitter on(String separator)`
  Le permite especificar el separador como una cadena.
- `static Splitter on(CharMatcher separatorMatcher)`
  Le permite especificar el separador como Guava `CharMatcher` .
- `static Splitter on(Pattern separatorPattern)`
  Le permite especificar el separador como un `Pattern` expresión regular de Java.
- `static Splitter onPattern(String separatorPattern)`
  Le permite especificar el separador como una cadena de expresión regular.

Además de estos métodos de fábrica basados en separadores, también hay un método `static Splitter fixedLength(int length)` para crear instancias `Splitter` que dividen cadenas en trozos de la longitud especificada.

Después de crear la instancia de `Splitter` , se pueden aplicar varios modificadores:

- `Splitter omitEmptyStrings()`
  Indica al `Splitter` que excluya cadenas vacías de los resultados.

- `Splitter trimResults()`
  `CharMatcher` al `Splitter` que recorte los resultados usando el `CharMatcher` espacios en blanco predefinido.

- `Splitter trimResults(CharMatcher trimmer)`
  Indica al `Splitter` que recorte los resultados utilizando el `CharMatcher` especificado.

Después de crear (y opcionalmente modificar) un `Splitter` , se puede invocar en una secuencia de caracteres invocando su método de `split` , que devolverá un objeto de tipo `Iterable<String>` , o su método `splitToList` , que devolverá un objeto (inmutable) de tipo `List<String>` .

Podría preguntarse en qué casos sería beneficioso utilizar el método de `split` (que devuelve un `Iterable`) en lugar del método `splitToList` (que devuelve el tipo de `List` más comúnmente utilizado). La respuesta breve a esto es: probablemente desee utilizar el método de `split` solo para procesar cadenas muy grandes. La respuesta ligeramente más larga es que debido a que el método de `split` devuelve un `Iterable` , las operaciones de división pueden evaluarse perezosamente (en el momento de la iteración), eliminando así la necesidad de mantener todo el resultado de la operación de división en la memoria.

Lea Instrumentos de cuerda en línea: https://riptutorial.com/es/guava/topic/4576/instrumentos-de-cuerda

Creditos

S. No	Capítulos	Contributors
1	Empezando con la guayaba	Community, Daniel Käfer, jayantS, Omar Hrynkiewicz
2	I / O	Olivier Grégoire
3	Instrumentos de cuerda	jbduncan, Robby Cornelissen, Xaerxess