<http://qinjiangbo.com/2016/11/06/Guava%E4%BC%98%E7%BE%8E%E4%BB%A3%E7%A0%81-14-Strings/>

Guava字符串处理

字符串不管在什么时候都是我们最为关心的，尤其是从事Web开发的朋友，字符串的各种操作也都是工作中必备的技能。Google也看到了我们广大的开发者处理字符串的痛点，于是他们为我们带来了Strings的工具类，这个工具类套件包含了很多有用的类，比如Strings类本身，还有用于分割的Splitter类，用于连接的Joiner类以及用于字符串匹配的CharMatcher类等等。下面我们就针对这些类进行详细的说明。

Strings类使用实例

package com.qinjiangbo;

import com.google.common.base.Strings;

import org.junit.Test;

/\*\*

\* Date: 9/12/16

\* Author: qinjiangbo@github.io

\*/

public class StringsTest {

@Test

public void testNullOrEmptyString() {

System.out.println(**Strings.isNullOrEmpty("")**); // true

}

@Test

public void testNullOrEmptyString2() {

System.out.println(**Strings.isNullOrEmpty(null)**); // true

}

@Test

public void testStringsPadEnd() {

System.out.println(**Strings.padEnd**("Hello World", 20, '-'));

// Hello World---------

}

@Test

public void testStringsPadStart() {

System.out.println(**Strings.padStart**("Hello World", 20, '\*'));

// \*\*\*\*\*\*\*\*\*Hello World

}

@Test

public void testStringsRepeat() {

System.out.println(**Strings.repeat**("I love you!\n", 20));

// I love you!

// I love you!

// ......

// I love you!

}

}

上面的代码中的测试方法都是自解释型的，所以我就不一一做详细的介绍了。不过需要解释一下其中的pad相关的方法，pad就是padding的简写，有两个方法padStart和padEnd，这两个方法说的是字符串的偏移，案例中Strings.padStart("Hello World", 20, '\*')是从起始位置向右偏移，左边空出来的位置使用\*号填充，同理，Strings.padEnd("Hello World", 20, '-')是从起始位置向左边偏移，右边空出来的部分使用-填充。

Splitter类使用实例

package com.qinjiangbo;

import com.google.common.base.CharMatcher;

import com.google.common.base.Splitter;

import com.google.common.collect.Lists;

import org.junit.Test;

import java.util.List;

/\*\*

\* Date: 9/10/16

\* Author: qinjiangbo@github.io

\*/

public class SplitterTest {

@Test

public void testSplitOnSemicolons() {

Iterable<String> iterable = Splitter.on(";").split("Java;Scala;Php;Haskell");

List<String> splittedList = Lists.newArrayList(iterable);

System.out.println(splittedList.get(2).equals("Php")); // true

}

@Test

public void testSplitOnRegExp() {

//onPattern按正则表达式拆分

Iterable<String> iterable

= **Splitter.onPattern("\\d+").split("Java13Scala41Php5C#6")**;

List<String> splittedList = Lists.newArrayList(iterable);

System.out.println(splittedList.get(2).equals("Php")); // true

}

@Test

public void testSplitUsingCharMatcher() {

Iterable<String> iterable

= **Splitter**

**.on(CharMatcher.inRange('3', '8'))**

**.split("Java3Scala4Haskell7Brain9Kotlin")**;

List<String> splittedList = Lists.newArrayList(iterable);

System.out.println(splittedList.get(3)); // Brain9Kotlin

}

@Test

public void testSplitOmitEmptyStrings() {

//omitEmptyStrings()从结果中自动忽略空字符串

Iterable<String> iterable

= **Splitter.on(";")**

**.omitEmptyStrings()**

**.split("Java;Scala; ;;Haskell;;Kotlin")**;

List<String> splittedList = Lists.newArrayList(iterable);

System.out.println(splittedList.get(2)); // [说明一下,这里输出是空格]

}

@Test

public void testSplitTrimResults() {

//trimResults()移除结果字符串的前导空白和尾部空白

Iterable<String> iterable

= **Splitter.on(";")**

**.trimResults()**

**.omitEmptyStrings()**

**.split("Java;Scala; ;;Haskell;;Kotlin")**;

List<String> splittedList = Lists.newArrayList(iterable);

System.out.println(splittedList.get(2)); // Haskell

}

@Test

public void testSplitOnFixedLength() {

//fixedLength() 按固定长度拆分

Iterable<String> iterable

= **Splitter.fixedLength(7)**

**.split("Someone once told me that I was lucky!")**;

List<String> splittedList = Lists.newArrayList(iterable);

System.out.println(splittedList.get(1)); // **空格once t** 7个字符

}

}

我们很容易从这个类的名字知道，这个类是用来分割字符串的，其实Java中的String类也可以分割字符串，比如String.split()方法，但是为什么我们要使用Guava的Splitter类呢？原因很简单，因为它更简单而且更强大！上面的测试方法也都是自解释型的，所以我还是只打算解释其中一两个测试方法的使用。testSplitUsingCharMatcher这个方法使用了我们即将要介绍的CharMatcher类，这个类能极大地提升这个分割的灵活性和多样性，基本上能涵盖我们平时开发中的所有要求。关于CharMatcher类具体的使用方式我们接下来介绍。

CharMatcher类使用实例

package com.qinjiangbo;

import com.google.common.base.CharMatcher;

import org.junit.Test;

/\*\*

\* Date: 9/10/16

\* Author: qinjiangbo@github.io

\*/

public class CharMatcherTest {

@Test

public void testNotMatchChar() {

System.out.println(**CharMatcher.noneOf("xZ").matchesAnyOf("anything")**);

} // true

@Test

public void testMatchAny() {

System.out.println(**CharMatcher.ANY.matchesAllOf("anything")**);

} // true

@Test

public void testMatchBreakingWhiteSpace() {

System.out.println(**CharMatcher.BREAKING\_WHITESPACE.matchesAllOf("\r\n\r\n")**);

} // true

@Test

public void testMatchDigits() {

System.out.println(**CharMatcher.DIGIT.matchesAllOf("1212121")**);

} // true

@Test

public void testMatchDigits2() {

System.out.println(**CharMatcher.DIGIT.matchesAnyOf("123abc123")**);

} // true

@Test

public void testMatchJavaDigits() {

System.out.println(**CharMatcher.JAVA\_DIGIT.matchesAllOf("123456")**);

} // true

@Test

public void testMatchJavaLetter() {

System.out.println(**CharMatcher.JAVA\_LETTER.matchesAllOf("Opera")**);

} // true

@Test

public void testMatchAscii() {

System.out.println(**CharMatcher.ASCII.matchesAllOf("azt\*1")**);

} // true

@Test

public void testMatchUpperCase() {

System.out.println(CharMatcher.JAVA\_UPPER\_CASE.matchesAllOf("JAVAC"));

} // true

@Test

public void testMatchDigitsWithWhiteSpaces() {

System.out.println(CharMatcher.DIGIT.matchesAnyOf("1111 abc"));

} // true

@Test

public void testMatchRetainsDigits() {

System.out.println(CharMatcher.DIGIT.retainFrom("123gb6789"));

} // 1236789

@Test

public void testMatchRetainsDigitsOrWhiteSpaces() {

System.out.println(CharMatcher.DIGIT.or(CharMatcher.WHITESPACE).retainFrom("Hello world 123 javac!"));

} // 123 [注意123前后都变成空格了]

@Test

public void testMatchRetainsNothingAsConstrainsAreExcluding() {

System.out.println(CharMatcher.DIGIT.and(CharMatcher.JAVA\_LETTER).retainFrom("hello 123 abc!"));

} // [这里是空格]

@Test

public void testMatchRetainsDigitsAndLetters() {

System.out.println(CharMatcher.DIGIT.or(CharMatcher.JAVA\_LETTER).retainFrom("hello 123 abc!"));

} // hello123abc

@Test

public void testMatchCollapseAllDigitsByX() {

System.out.println(CharMatcher.DIGIT.collapseFrom("Hello 167 j176", 'x'));

} // Hello x jx

@Test

public void testMatchReplaceAllDigitsByX() {

System.out.println(CharMatcher.DIGIT.replaceFrom("Hello 17689 jik009", 'x'));

} // Hello xxxxx jikxxx

@Test

public void testMatchReplaceAllLettersByX() {

System.out.println(CharMatcher.JAVA\_LETTER.or(CharMatcher.is('\*')).replaceFrom("password 97321321 \*\*65", 'X'));

} // XXXXXXXX 97321321 XX65

@Test

public void testMatchCountIn() {

System.out.println(CharMatcher.DIGIT.countIn("\*\*\* 121 \* a \*\* b"));

} // 3

@Test

public void testMatchCountIn2() {

System.out.println(CharMatcher.is('\*').countIn("\*\*\* 121 \* a \*\* b"));

} // 6

@Test

public void testMatchIndexIn() {

System.out.println(CharMatcher.is('\*').indexIn("666 \*\*\* 121 \* a \*\* b"));

} // 4

@Test

public void testMatchLastIndexIn() {

System.out.println(CharMatcher.is('\*').lastIndexIn("666 \*\*\* 121 \* a \*\* b"));

} // 17

@Test

public void testMatchRemoveDigitsBetween3And6() {

System.out.println(CharMatcher.inRange('3', '8').removeFrom("117787321daa096aa453aa299"));

} // 1121daa09aaaa299

@Test

public void testNegateMatchingAbove() {

System.out.println(CharMatcher.inRange('3', '8').negate().removeFrom("117787321daa096aa453aa299"));

} // 778736453

@Test

public void testRemoveStartingAndEndingDollarsAndKeepOthersUnchanged() {

System.out.println(CharMatcher.is('$').trimFrom("$$$ This is a $ sign $$$"));

} // This is a $ sign [前后都有空格]

@Test

public void testRemoveOnlyStartingDollarsAndKeepOthersUnchanged() {

System.out.println(CharMatcher.is('$').trimLeadingFrom("$$$ This is a $ sign $$$"));

} // This is a $ sign $$$

@Test

public void testRemoveOnlyEndingDollarsAndKeepOthersUnchanged() {

System.out.println(CharMatcher.is('$').trimTrailingFrom("$$$ This is a $ sign $$$"));

} // $$$ This is a $ sign

@Test

public void testRemoveStartingAndEndingDollarsAndReplaceOtherDollarsWithX() {

System.out.println(CharMatcher.is('$').trimAndCollapseFrom("$$$ This is a $$ and $ sign $$$", 'X'));

} // This is a X and X sign

}

CharMatcher为我们提供了非常强大的处理字符串的能力，可以看到CharMatcher类基本上涵盖了我们操作字符串的各种方式。大家可以仔细看看上面的测试实例，并且好好琢磨琢磨它的使用方式。

Joiner类使用实例

package com.qinjiangbo;

import com.google.common.base.Joiner;

import com.google.common.collect.Maps;

import org.junit.Test;

import java.util.Arrays;

import java.util.List;

import java.util.Map;

/\*\*

\* Date: 9/10/16

\* Author: qinjiangbo@github.io

\*/

public class JoinerTest {

@Test

public void testJoinerOn() {

List<String> languages = Arrays.asList("Java", "Haskell", "Scala", "Brainfuck");

System.out.println(Joiner.on(',').join(languages));

} // Java,Haskell,Scala,Brainfuck

@Test

public void testJoinerJoinWithCommasAndOmitNulls() {

List<String> countriesWithNullValue = Arrays.asList("Poland", "Brazil", "Ukraine", null, "China");

System.out.println(Joiner.on(',').skipNulls().join(countriesWithNullValue));

} // Poland,Brazil,Ukraine,China

@Test

public void testJoinerJoinWithCommasAndReplaceNullWithDefaultValue() {

List<String> countriesWithNullValue = Arrays.asList("Poland", "Brazil", "Ukraine", null, "China");

System.out.println(Joiner.on(',').useForNull("NONE").join(countriesWithNullValue));

} // Poland,Brazil,Ukraine,NONE,China

@Test

public void testJoinerJoinMap() {

Map<Integer, String> numberWords = Maps.newHashMap();

numberWords.put(1, "one");

numberWords.put(2, "two");

numberWords.put(3, null);

numberWords.put(4, "four");

System.out.println(Joiner.on(" | ").withKeyValueSeparator(" -> ").useForNull("Unknown").join(numberWords));

} // 1 -> one | 2 -> two | 3 -> Unknown | 4 -> four

}

需要说明一下最后一个测试方法，这个是将Joiner类应用到了Map上面，我们可以利用Joiner类将Map表现形式转换成为我们想要的形式，这样更加易于读和理解。

总结

好了，Guava中的Strings相关的操作类在上面已经用实例代码进行了说明，每段代码都给出了相应的输出，测试全部通过。大家可以将上面的代码拷贝下来，自己到本机上跑一下，感受一下这段代码给我们带来的便捷与强大的操作能力。总的来说，Guava中涉及到字符串操作相关的类还有很多，这里不一一列出了。因为那些都没有上面列出的这些类方便和强大。我们主要是关注主要的类，我建议读者朋友们有时间自己还是多跑一跑上面的这些代码，更好地就是自己能动手敲一遍，这样更能加深自己的理解。