

HenCoder Android 开发进阶

自定义 View 1-3 drawText() 文字的绘制



这期是 HenCoder 自定义绘制的第三期：文字的绘制。

简介

上期的 Paint 详解里已经说过，文字的绘制所能控制的内容太多太细，必须拆成单独的一期专门来讲。今天这期，就是来把这些细节讲清楚的。

需要说明的有两点：

1. 和上期一样，这期讲的是细节，其中有一部分内容并不是很常用，所以这期你不必要求自己把内容全部背会，而只要做到全部理解，知道都有什么东西，大概怎么用就好，到你真正需要用的时候再拐回来看就是；
2. 除了常用和不常用的内容，本期还会讲到一些比较偏门的细节。这些偏门几乎永远不会用到，我讲这些偏门的目的是为了做到知识的全覆盖，帮你破开迷雾解开谜团，把那些「始终没有搞懂，也不知道有没有用」的 API 解释出来。有的时候，一样东西你确定了它确实没用，也就够了。所以如果遇到这些偏门的内容，你感兴趣，看看就好；不感兴趣，不看也罢——总之，不要把太多精力放在它们身上。

下面进入正题。

1 Canvas 绘制文字的方式

Canvas 的文字绘制方法有三个：`drawText()` `drawTextRun()` 和 `drawTextOnPath()`。

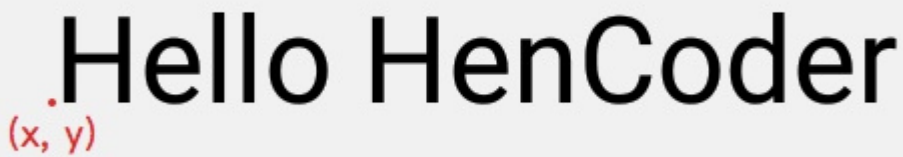
1.1 `drawText(String text, float x, float y, Paint paint)`

`drawText()` 是 Canvas 最基本的绘制文字的方法：给出文字的内容和位置，Canvas 按要求去绘制文字。

```
String text = "Hello HenCoder";  
  
...  
  
canvas.drawText(text, 200, 100, paint);
```

Hello HenCoder

方法的参数很简单：`text` 是文字内容，`x` 和 `y` 是文字的坐标。但需要注意：这个坐标并不是文字的左上角，而是一个与左下角比较接近的位置。大概在这里：



The diagram shows the text "Hello HenCoder" in a large, black, sans-serif font. A red dot is placed at the baseline of the first letter 'H'. Below this dot, the text "(x, y)" is written in red, indicating that these coordinates correspond to the baseline of the text.

而如果你像绘制其他内容一样，在绘制文字的时候把坐标填成 `(0, 0)`，文字并不会显示在 `View` 的左上角，而是会几乎完全显示在 `View` 的上方，到了 `View` 外部看不到的位置：

```
canvas.drawText(text, 0, 0, paint);
```

↑ 这里没有贴错图哦

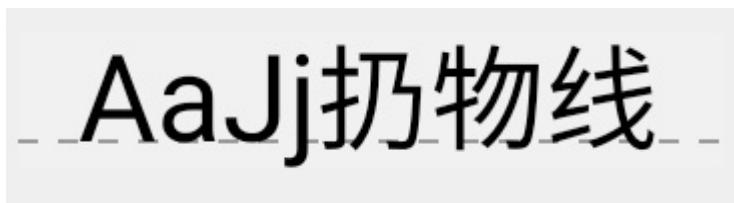


再附上一张图，应该能更清楚地表达：

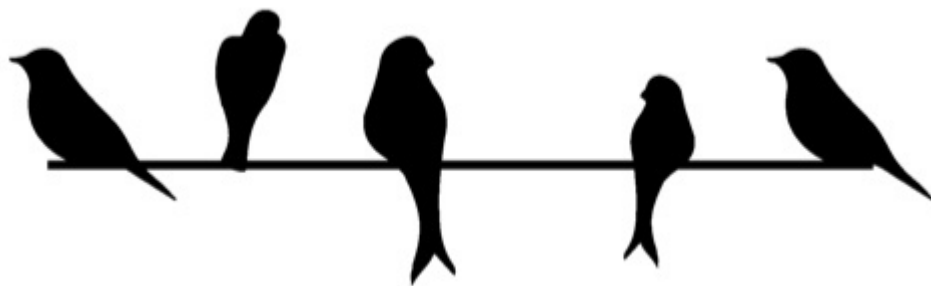


这是为什么？为什么其它的 `Canvas.drawXXX()` 方法，都是以左上角作为基准点的，而 `drawText()` 却是文字左下方？

先别觉得日了狗，这种设计其实是有道理的。`drawText()` 参数中的 `y`，指的是文字的**基线（baseline）**的位置。也就是这条线：



众所周知，不同的语言和文字，每个字符的高度和上下位置都是不一样的。要让不同的文字并排显示的时候整体看起来稳当，需要让它们上下对齐。但这个对齐的方式，不能是简单的「底部对齐」或「顶部对齐」或「中间对齐」，而应该是一种类似于「重心对齐」的方式。就像电线上的小鸟一样：



每只小鸟的最高点和最低点都不一样，但画面很平衡

而这个用来让所有文字互相对齐的基准线，就是**基线(baseline)**。`drawText()` 方法参数中的 `y` 值，就是指定的基线的位置。

说完 `y` 值，再说说 `x` 值。从前面图中的标记可以看出来，「Hello HenCoder」绘制出来之后的 `x` 点并不是字母 "H" 左边的位置，而是比它的左边再往左一点点。那么这个「往左的一点点」是什么呢？

它是字母 "H" 的左边的空隙。绝大多数的字符，它们的宽度都是要略微大于实际显示的宽度的。字符的左右两边会留出一部分空隙，用于文字之间的间隔，以及文字和边框的间隔。就像这样：

用竖线标记出边界后的文字。

所以，明白为什么 `x` 坐标在 "H" 的左边再往左一点点的位置，而不是紧贴着 "H" 的左边线了吗？就是因为 "H" 的这个留出的空隙。

除了 `drawText(text, x, y, paint)` 之外，`drawText()` 还有几个重载方法，使用方式跟这个都差不多，我就不说了，你自己看吧。

1.2 drawTextRun()

声明：这个方法对中国人没用。所以如果你有兴趣，可以继续看；而如果你想省时间，直接跳过这个方法看后面的就好了，没有任何毒副作用。

`drawTextRun()` 是在 API 23 新加入的方法。它和 `drawText()` 一样都是绘制文字，但加入了两项额外的设置——**上下文和文字方向**——用于辅助一些文字结构比较特殊的语言的绘制。

- 额外设置一：上下文。

有些语言的文字，字符的形状会互相之间影响：一个字你单独写是一个样，和别的字放在一起写又是另外一个样。不过由于我们最熟悉的语言——汉语和英语——都没有这种情况，所以只靠说可能不太好理解，我就用图说明一下吧。

以阿拉伯文为例。阿拉伯文里的「عربي (阿拉伯)」是一个四字词，它的中间两个字符「رب」在这个词里的样子，和单独写的时候的样子是不同的。也就是说，当这四个字写在一起的时候，中间两个字由于受到两边的字的影响，形状被改变了。看图吧：

عربي

原字符串

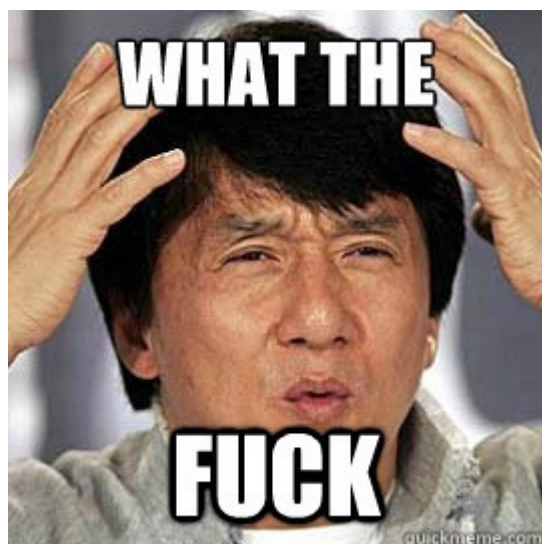
رب

中间两个字符

رب

中间两个字符（单独写）

上面第二行和第三行的文字是完全一样的俩字，你敢信？



哇塞，是不是特别神奇？

不过我们就不用管它为什么这么神奇了，也不用替阿拉伯人操心这么复杂的文字他们使用起来会不会很痛苦，人家都已经用了几百上千年了。我还说回到 `drawTextRun()`。`drawTextRun()` 除了文字的内容和位置之外，还可以设置文字的上下文（也就是要绘制的文字的左边和右边是什么文字，虽然这些文字并不会被绘制出来），从而让同样的文字可以按需表现出不同的显示效果。

- 额外设置二：文字方向。

除了上下文，`drawTextRun()` 还可以设置文字的方向，即文字是从左到右还是从右到左排列的。

介绍完这两类额外设置，来看一下具体的方法吧：

`drawTextRun(CharSequence text, int start, int end, int contextStart, int contextEnd, float x, float y, boolean isRtl, Paint paint)`

参数：

`text`：要绘制的文字

`start`：从那个字开始绘制

`end`：绘制到哪个字结束

`contextStart`：上下文的起始位置。`contextStart` 需要小于等于 `start`

`contextEnd`：上下文的结束位置。`contextEnd` 需要大于等于 `end`

`x`：文字左边的坐标

`y`：文字的基线坐标

`isRtl`：是否是 RTL (Right-To-Left, 从右向左)

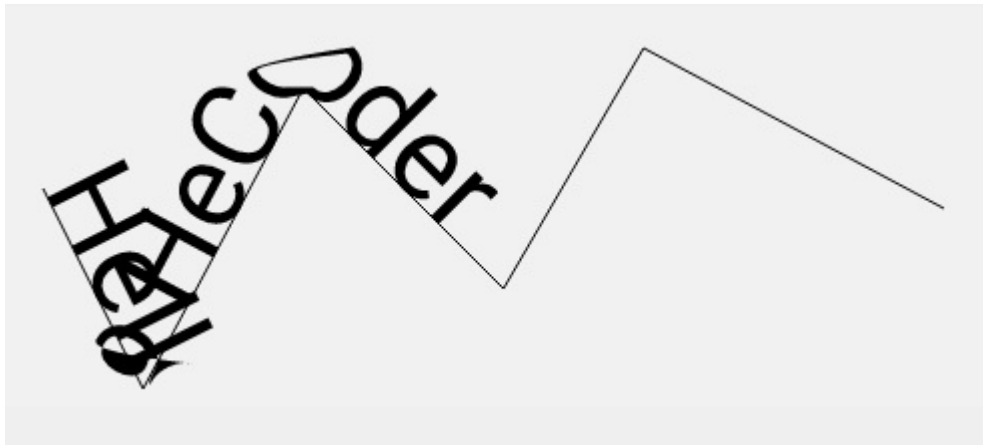
要实现上面图中的「同样的字有不同的显示」效果，调节 `contextStart` 和 `contextEnd` 就可以了，至于具体的实现，你有兴趣的话就自己试试吧。

这就是 `drawTextRun()`，一个增加了「上下文」和「RTL」支持的增强版本的 `drawText()`。不过就像刚才说过的，这个方法对中国人其实没什么用……

1.3 drawTextOnPath()

沿着一条 `Path` 来绘制文字。这是一个耍杂技的方法。

```
canvas.drawPath(path, paint); // 把 Path 也绘制出来，理解起来更方便
canvas.drawTextOnPath("Hello HeCoder", path, 0, 0, paint);
```



吁，拐角处的文字怎么那么难看？

所以记住一条原则：`drawTextOnPath()` 使用的 `Path`，拐弯处全用圆角，别用尖角。

具体的方法很简单：

```
drawTextOnPath(String text, Path path, float hOffset, float vOffset, Pai
```

参数里，需要解释的只有两个：`hOffset` 和 `vOffset`。它们是文字相对于 `Path` 的水平偏移量和竖直偏移量，利用它们可以调整文字的位置。例如你设置 `hOffset` 为 5，`vOffset` 为 10，文字就会右移 5 像素和下移 10 像素。

1.4 StaticLayout

额外讲一个 `StaticLayout`。这个也是使用 `Canvas` 来进行文字的绘制，不过并不是使用 `Canvas` 的方法。

`Canvas.drawText()` 只能绘制单行的文字，而不能换行。它：

- 不能在 View 的边缘自动折行

```
String text = "Lorem Ipsum is simply dummy text of the printing a  
...  
canvas.drawText(text, 50, 100, paint);
```

Lorem Ipsum is simply dummy text of

到了 View 的边缘处，文字继续向后绘制到看不见的地方，而不是自动换行

- 不能在换行符 `\n` 处换行

```
String text = "a\nbc\ndefghi\njklm\nnopqrst\nuvwxy\nz";  
...  
canvas.drawText(text, 50, 100, paint);
```

a bc defghi jklm nopqrst uvwx yz

在换行符 `\n` 的位置并没有换行，而只是加了个空格

如果需要绘制多行的文字，你必须自行把文字切断后分多次使用 `drawText()` 来绘制，或者——使用 `StaticLayout` 。

StaticLayout 并不是一个 view 或者 ViewGroup，而是 android.text.Layout 的子类，它是纯粹用来绘制文字的。StaticLayout 支持换行，它既可以为文字设置宽度上限来让文字自动换行，也会在 \n 处主动换行。

```
String text1 = "Lorem Ipsum is simply dummy text of the printing and  
StaticLayout staticLayout1 = new StaticLayout(text1, paint, 600,  
        Layout.Alignment.ALIGN_NORMAL, 1, 0, true);  
String text2 = "a\nbc\ndefghi\njklm\nnopqrst\nuvwxy\nz";  
StaticLayout staticLayout2 = new StaticLayout(text2, paint, 600,  
        Layout.Alignment.ALIGN_NORMAL, 1, 0, true);  
  
...  
  
canvas.save();  
canvas.translate(50, 100);  
staticLayout1.draw(canvas);  
canvas.translate(0, 200);  
staticLayout2.draw(canvas);  
canvas.restore();
```

上面代码中出现的 Canvas.save() Canvas.translate() Canvas.restore() 配合起来可以对绘制的内容进行移动。它们的具体用法我会在下期讲，这期你就先依葫芦画瓢照搬着用吧。

Lorem Ipsum is simply dummy
text of the printing and
typesetting industry.

a
bc
defghi
jklm
nopqrst
uvwxy
z

自动换行
\n 换行

`StaticLayout` 的构造方法是 `StaticLayout(CharSequence source, TextPaint paint, int width, Layout.Alignment align, float spacingmult, float spacingadd, boolean includepad)`，其中参数里：

`width` 是文字区域的宽度，文字到达这个宽度后就会自动换行；

`align` 是文字的对齐方向；

`spacingmult` 是行间距的倍数，通常情况下填 1 就好；

`spacingadd` 是行间距的额外增加值，通常情况下填 0 就好；

`includeadd` 是指是否在文字上下添加额外的空间，来避免某些过高的字符的绘制出现越界。

如果你需要进行多行文字的绘制，并且对文字的排列和样式没有太复杂的花式要求，那么使用 `StaticLayout` 就好。

2 Paint 对文字绘制的辅助

`Paint` 对文字绘制的辅助，有两类方法：设置显示效果的和测量文字尺寸的。

2.1 设置显示效果类

2.1.1 setTextSize(float textSize)

设置文字大小。

```
paint.setTextSize(18);  
canvas.drawText(text, 100, 25, paint);  
paint.setTextSize(36);  
canvas.drawText(text, 100, 70, paint);  
paint.setTextSize(60);  
canvas.drawText(text, 100, 145, paint);  
paint.setTextSize(84);  
canvas.drawText(text, 100, 240, paint);
```

Hello HenCoder

Hello HenCoder

Hello HenCoder

Hello HenCoder

很简单，不再详细解释。

2.1.2 setTypeface(Typeface typeface)

设置字体。

```
paint.setTypeface(Typeface.DEFAULT);  
canvas.drawText(text, 100, 150, paint);  
paint.setTypeface(Typeface.SERIF);
```

```
canvas.drawText(text, 100, 300, paint);  
paint.setTypeface(Typeface.createFromAsset(getContext().getAssets()  
canvas.drawText(text, 100, 450, paint);
```



Hello HenCoder
Hello HenCoder
Hello HenCoder

设置不同的 `Typeface` 就可以显示不同的字体。我们中国人谈到「字体」，比较熟悉的词是 `font`，`typeface` 和 `font` 是一个意思，都表示字体。`Typeface` 这个类的具体用法，需要了解的话可以[直接看文档](#)，很简单。

严格地说，其实 `typeface` 和 `font` 意思不完全一样。`typeface` 指的是某套字体（即 `font family`），而 `font` 指的是一个 `typeface` 具体的某个 `weight` 和 `size` 的分支。不过无所谓啦~做人最紧要系开心啦。

2.1.3 `setFakeBoldText(boolean fakeBoldText)`

是否使用伪粗体。

```
paint.setFakeBoldText(false);  
canvas.drawText(text, 100, 150, paint);  
paint.setFakeBoldText(true);  
canvas.drawText(text, 100, 230, paint);
```

Hello HenCoder

Hello HenCoder

之所以叫伪粗体（fake bold），因为它并不是通过选用更高 weight 的字体让文字变粗，而是通过程序在运行时把文字给「描粗」了。

2.1.4 setStrikeThruText(boolean strikeThruText)

是否加删除线。

```
paint.setStrikeThruText(true);  
canvas.drawText(text, 100, 150, paint);
```

~~Hello HenCoder~~

2.1.5 setUnderlineText(boolean underlineText)

是否加下划线。

```
paint.setUnderlineText(true);  
canvas.drawText(text, 100, 150, paint);
```

Hello HenCoder
yummy juicy baby

2.1.6 setTextSkewX(float skewX)

设置文字横向错切角度。其实就是文字倾斜度的啦。

```
paint.setTextSkewX(-0.5f);  
canvas.drawText(text, 100, 150, paint);
```

Hello HenCoder

2.1.7 setTextScaleX(float scaleX)

设置文字横向放缩。也就是文字变胖变瘦。

```
paint.setTypeface(Typeface.DEFAULT);  
canvas.drawText(text, 100, 150, paint);  
paint.setTypeface(Typeface.SERIF);  
canvas.drawText(text, 100, 300, paint);  
paint.setTypeface(Typeface.createFromAsset(getContext().getAssets(),  
canvas.drawText(text, 100, 450, paint);
```


Hello HenCoder
Hello HenCoder
Hello HenCoder

2.1.8 setLetterSpacing(float letterSpacing)

设置字符间距。默认值是 0。

```
paint.setLetterSpacing(0.2f);  
canvas.drawText(text, 100, 150, paint);
```

H e l l o H e n C o d e r

为什么在默认的字符间距为 0 的情况下，字符和字符之间也没有紧紧贴着，这个我在前面讲 `Canvas.drawText()` 的 `x` 参数的时候已经说过了，在这里应该没有疑问吧？

2.1.9 setFontFeatureSettings(String settings)

用 CSS 的 `font-feature-settings` 的方式来设置文字。

```
paint.setFontFeatureSettings("smcp"); // 设置 "small caps"  
canvas.drawText("Hello HenCoder", 100, 150, paint);
```

HELLO HENCODER

CSS 全称是 Cascading Style Sheets，是网页开发用来设置页面各种元素的样式的。咦，网页开发的设置怎么会出现在 Android 的 API 里？



Android 开发没人
要了

大多数 Android 开发者都不了解这个 CSS 的 `font-feature-settings` 属性，不过没关系，这个属性设置的都是文字的一些次要特性，所以不用着急了解这个方法。当然有兴趣的话也可以看一看哈，文档在[这里](#)。

2.1.10 setTextAlign(Paint.Align align)

设置文字的对齐方式。一共有三个值：LEFT CENTER 和 RIGHT。默认值为 LEFT。

```
paint.setTextAlign(Paint.Align.LEFT);
canvas.drawText(text, 500, 150, paint);
paint.setTextAlign(Paint.Align.CENTER);
canvas.drawText(text, 500, 150 + textHeight, paint);
paint.setTextAlign(Paint.Align.RIGHT);
canvas.drawText(text, 500, 150 + textHeight * 2, paint);
```

Hello HenCoder

Hello HenCoder

Hello HenCoder

2.1.11 setTextLocale(Locale locale) / setTextLocales(LocaleList locales)

设置绘制所使用的 `Locale`。

`Locale` 直译是「地域」，其实就是你在系统里设置的「语言」或「语言区域」（具体名称取决于你用的是什么手机），比如「简体中文（中国）」「English (US)」「English (UK)」。有些同源的语言，在文化发展过程中对一些相同的字衍生出了不同的写法（比如中国大陆和日本对于某些汉字的写法就有细微差别。注意，不是繁体和简体这种同音同义不同字，而真的是同样的一个字有两种写法）。系统语言不同，同样的一个字的显示就有可能不同。你可以试一下把自己手机的语言改成日文，然后打开微信看看聊天记录，你会明显发现文字的显示发生了很多细微的变化，这就是由于系统的 `Locale` 改变所导致的。

`Canvas` 绘制的时候，默认使用的是系统设置里的 `Locale`。而通过 `Paint.setTextLocale(Locale locale)` 就可以在不改变系统设置的情况下，直接修改绘制时的 `Locale`。

```
paint.setTextLocale(Locale.CHINA); // 简体中文
canvas.drawText(text, 150, 150, paint);
paint.setTextLocale(Locale.TAIWAN); // 繁体中文
canvas.drawText(text, 150, 150 + textHeight, paint);
paint.setTextLocale(Locale.JAPAN); // 日语
canvas.drawText(text, 150, 150 + textHeight * 2, paint);
```

雨骨底条今直沿微写

Locale.CHINA

雨骨底条今直沿微写

Locale.TAIWAN

雨骨底条今直沿微写

Locale.JAPAN

有意思吧？

另外，由于 Android 7.0 (API v24) 加入了多语言区域的支持，所以在 API v24 以及更高版本上，还可以使用 `setTextLocales(LocaleList locales)` 来为绘制设置多个语言区域。

2.1.12 setHinting(int mode)

设置是否启用字体的 hinting （字体微调）。

现在的 Android 设备大多数都是用的[矢量字体](#)。矢量字体的原理是对每个字体给出一个字形的矢量描述，然后使用这一个矢量来对所有的尺寸的字体来生成对应的字形。由于不必为所有字号都设计它们的字体形状，所以在字号较大的时候，矢量字体也能够保持字体的圆润，这是矢量字体的优势。不过当文字的尺寸过小（比如高度小于 16 像素），有些文字会由于失去过多细节而变得不太好看。hinting 技术就是为了解决这种问题的：通过向字体中加入 hinting 信息，让矢量字体在尺寸过小的时候得到针对性的修正，从而提高显示效果。效果图盗一张[维基百科](#)的：

abcfgop AO abcfgop 維基百科
abcfgop AO abcfgop 維基百科國際
維基百科國際

abcfgop 維基百科
維基百科國際
abcfgop 維基百科
維基百科國際

不带微调（上、下部分位于上边的行）和带微调（上、下部分位于下边的行）的字体测试。上部分为100%缩放，下部分为400%缩放。微调造成了边角对比度提升，但损失了忠实表现和自然间隔。

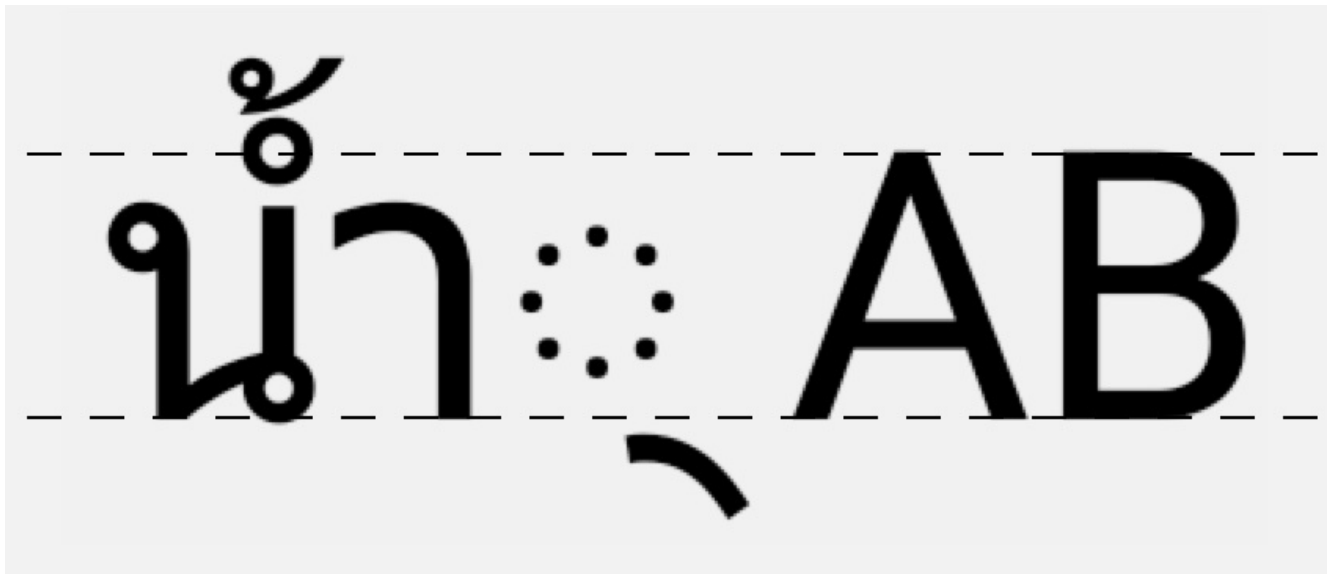
功能很强，效果很赞。不过在现在（2017年），手机屏幕的像素密度已经非常高，几乎不会再出现字体尺寸小到需要靠 hinting 来修正的情况，所以这个方法其实……没啥用了。可以忽略。

2.1.13 setElegantTextHeight(boolean elegant)

声明：这个方法对中国人没用，不想看的话可以直接跳过，无毒副作用。

设置是否开启文字的 elegant height。开启之后，文字的高度就变优雅了（误）。下面解释一下所谓的 elegant height：

在有些语言中，可能会出现一些非常高的字形：

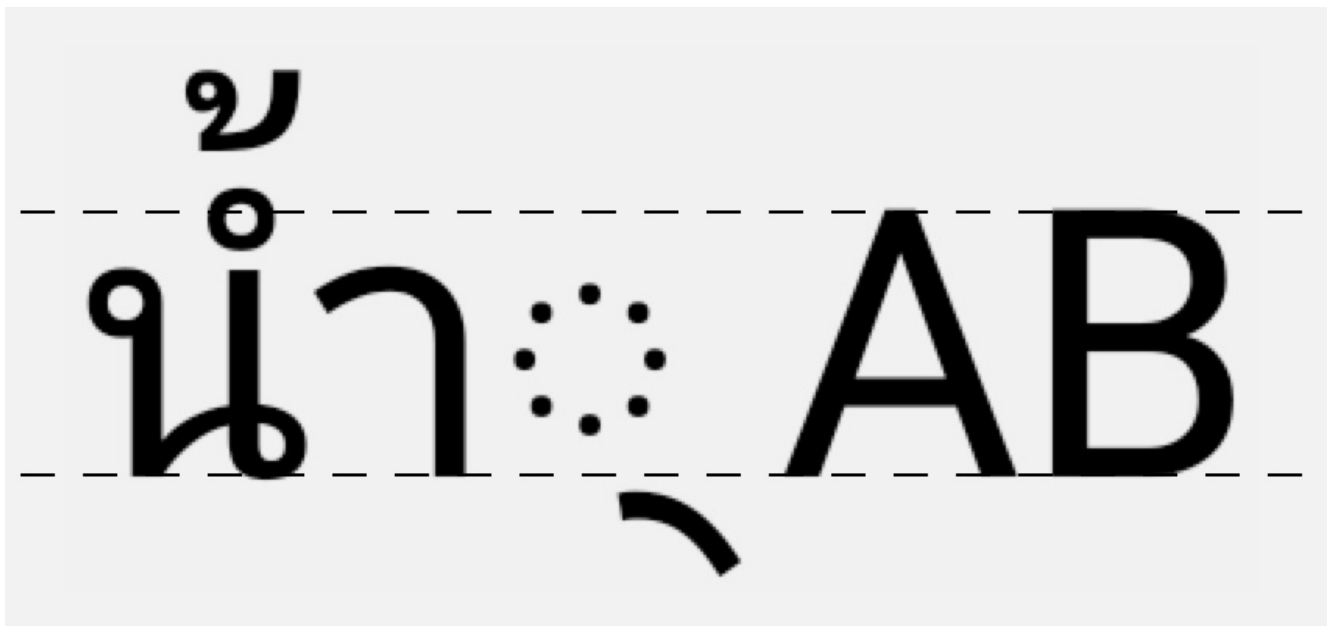


左边那几个泰文文字，挺高的吧？但其实它们已经是被压缩过了的，它们本来比这还要高。

这些比较高的文字，通常都有两个版本的字体：一个原始版本，一个压缩了高度的版本。压缩版本可以保证让这些「大高个」文字在和普通文字（例如拉丁文字）放在一起的时候看起来不会显得太奇怪。事实上，`Paint` 绘制文字时是用的默认版本就是压缩版本，就像上图这样。

不过有的时候，开发者会需要使用它们的原始（优雅）版本。使用 `setElegantTextHeight()` 就可以切换到原始版本：

```
paint.setElegantTextHeight(true);
```



这字得有多高？2 米 26？

那么，`setElegantTextHeight()` 的作用到这里就很清晰了：

1. 把「大高个」文字的高度恢复为原始高度；
2. 增大每行文字的上下边界，来容纳被加高的文字。

其实这个问题我已经在 `stackoverflow` 回答过一次，原回答在[这里](#)。

不过就像前面说的，由于中国人常用的汉语和英语的文字并不会达到这种高度，所以这个方法对于中国人基本上是没用的。

2.1.14 `setSubpixelText(boolean subpixelText)`

是否开启次像素级的抗锯齿（sub-pixel anti-aliasing）。

次像素级抗锯齿这个功能解释起来很麻烦，简单说就是根据程序所运行的设备的屏幕类型，来进行针对性的次像素级的抗锯齿计算，从而达到更好的抗锯齿效果。更详细的解释可以看[这篇文章](#)。

不过，和前面讲的字体 hinting 一样，由于现在手机屏幕像素密度已经很高，所以默认抗锯齿效果就已经足够好了，一般没必要开启次像素级抗锯齿，所以这个方法基本上没有必要使用。

2.1.15 `setLinearText(boolean linearText)`

这个方法老实说我从没用过，也始终没有搞懂它是什么意思，就不强行装逼了。把文档中的解释照搬过来，各位自己研究吧。

Helper for `setFlags()`, setting or clearing the `LINEARTEXTFLAG` bit

上面这句中提到的 `LINEAR_TEXT_FLAG`：

Paint flag that enables smooth linear scaling of text.

Enabling this flag does not actually scale text, but rather adjusts text draw operations to deal gracefully with smooth adjustment of scale. When this flag is enabled, font hinting is disabled to prevent shape deformation between scale factors, and glyph caching is disabled due to the large number of glyph images that will be generated.

SUBPIXELTEXTFLAG should be used in conjunction with this flag to prevent glyph positions from snapping to whole pixel values as scale factor is adjusted.

以上就是 `Paint` 的对文字的显示效果设置类方法。下面介绍它的第二类方法：测量文字尺寸类。

2.2 测量文字尺寸类

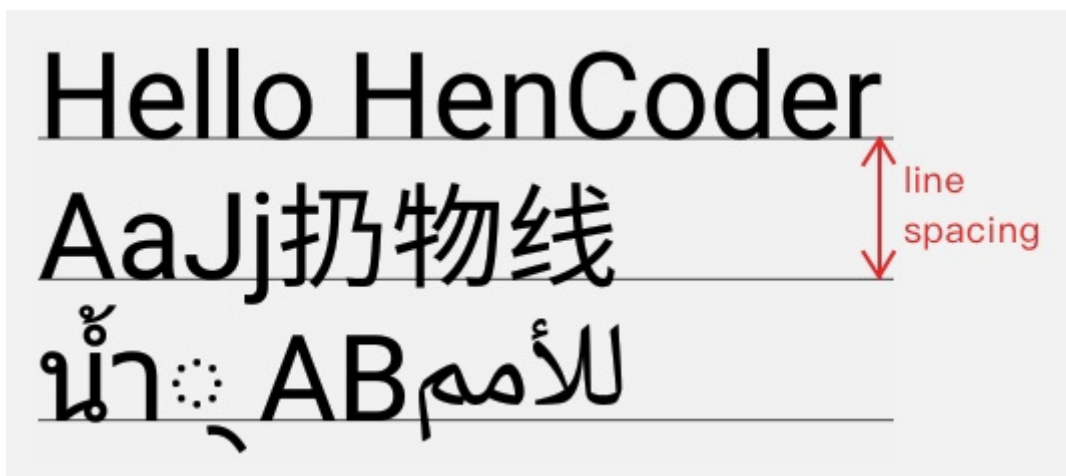
不论是文字，还是图形或 `Bitmap`，只有知道了尺寸，才能更好地确定应该摆放的位置。由于文字的绘制和图形或 `Bitmap` 的绘制比起来，尺寸的计算复杂得多，所以它有一整套的方法来计算文字尺寸。

2.2.1 float getFontSpacing()

获取推荐的行距。

即推荐的两行文字的 `baseline` 的距离。这个值是系统根据文字的字体和字号自动计算的。它的作用是当你要手动绘制多行文字（而不是使用 `StaticLayout`）的时候，可以在换行的时候给 `y` 坐标加上这个值来下移文字。

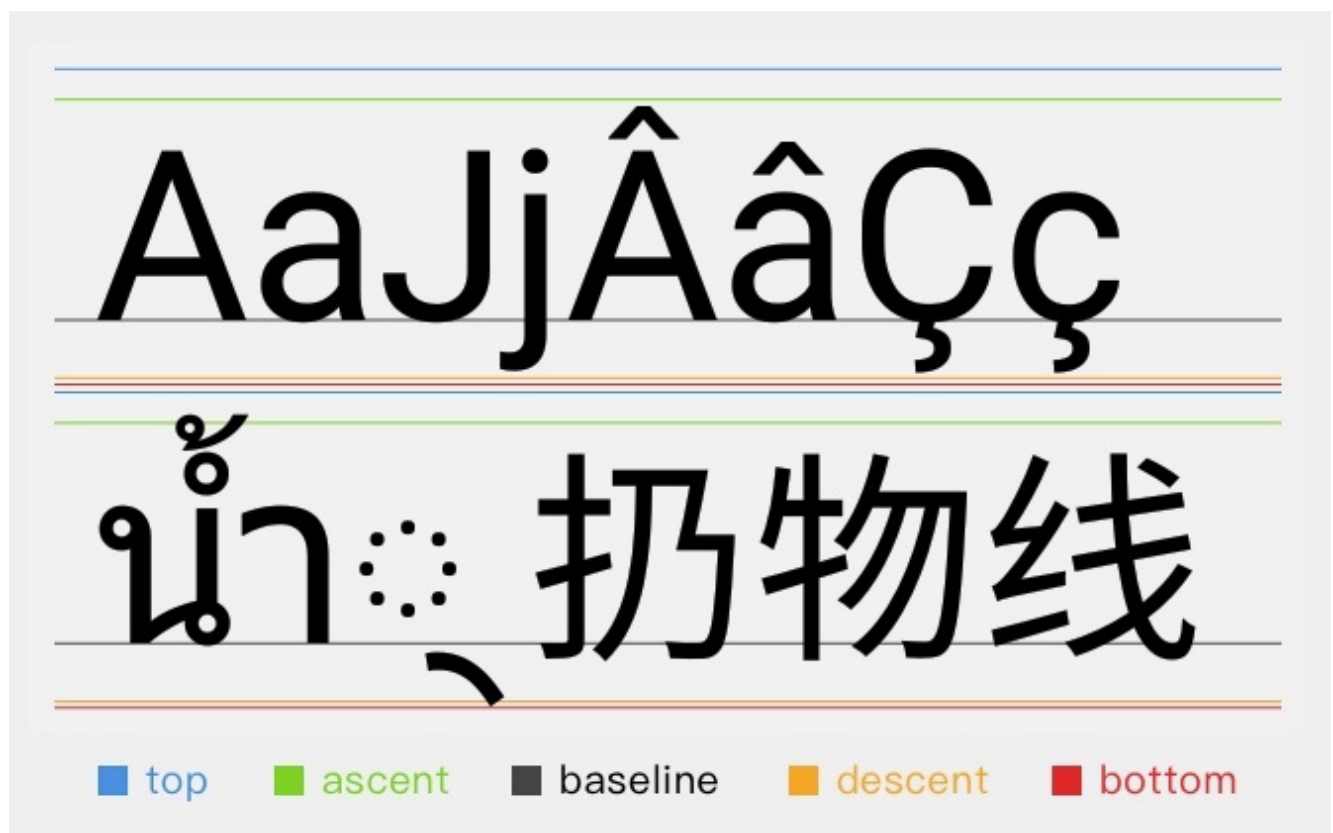
```
canvas.drawText(texts[0], 100, 150, paint);
canvas.drawText(texts[1], 100, 150 + paint.getFontSpacing(), paint);
canvas.drawText(texts[2], 100, 150 + paint.getFontSpacing() * 2, paint);
```

2.2.2 FontMetrics.getFontMetrics()

获取 Paint 的 FontMetrics。

FontMetrics 是个相对专业的工具类，它提供了几个文字排印方面的数值：ascent, descent, top, bottom, leading。



如图，图中有两行文字，每一行都有 5 条线：top, ascent, baseline, descent, bottom。（leading 并没有画出来，因为画不出来，下面会给出解释）

- `baseline`: 上图中黑色的线。前面已经讲过了, 它的作用是作为文字显示的基准线。
- `ascent / descent`: 上图中绿色和橙色的线, 它们的作用是限制普通字符的顶部和底部范围。

普通的字符, 上不会高过 `ascent`, 下不会低过 `descent`, 例如上图中大部分的字形都显示在 `ascent` 和 `descent` 两条线的范围内。具体到 Android 的绘制中, `ascent` 的值是图中绿线和 `baseline` 的相对位移, 它的值为负 (因为它在 `baseline` 的上方); `descent` 的值是图中橙线和 `baseline` 相对位移, 值为正 (因为它在 `baseline` 的下方)。
- `top / bottom`: 上图中蓝色和红色的线, 它们的作用是限制所有字形 (`glyph`) 的顶部和底部范围。

除了普通字符, 有些字形的显示范围是会超过 `ascent` 和 `descent` 的, 而 `top` 和 `bottom` 则限制的是所有字形的显示范围, 包括这些特殊字形。例如上图的第二行文字里, 就有两个泰文的字形分别超过了 `ascent` 和 `descent` 的限制, 但它们都在 `top` 和 `bottom` 两条线的范围内。具体到 Android 的绘制中, `top` 的值是图中蓝线和 `baseline` 的相对位移, 它的值为负 (因为它在 `baseline` 的上方); `bottom` 的值是图中红线和 `baseline` 相对位移, 值为正 (因为它在 `baseline` 的下方)。
- `leading`: 这个词在上图中没有标记出来, 因为它并不是指的某条线和 `baseline` 的相对位移。 `leading` 指的是行的额外间距, 即对于上下相邻的两行, 上行的 `bottom` 线和下行的 `top` 线的距离, 也就是上图中**第一行的红线**和**第二行的蓝线**的距离 (对, 就是那个小细缝)。

`leading` 这个词的本意其实并不是行的额外间距, 而是行距, 即两个相邻行的 `baseline` 之间的距离。不过对于很多非专业领域, `leading` 的意思被改变了, 被大家当做行的额外间距来用; 而 Android 里的 `leading`, 同样也是行的额外间距的意思。

另外, `leading` 在这里应该读作 "ledding" 而不是 "leeding" 哦。原因就不说了, 我这越扯越远没边了。

FontMetrics 提供的就是 Paint 根据当前字体和字号，得出的这些值的推荐值。它把这些值以变量的形式存储，供开发者需要时使用。

- FontMetrics.ascent：float 类型。
- FontMetrics.descent：float 类型。
- FontMetrics.top：float 类型。
- FontMetrics.bottom：float 类型。
- FontMetrics.leading：float 类型。

另外，ascent 和 descent 这两个值还可以通过 Paint.ascent() 和 Paint.descent() 来快捷获取。

FontMetrics 和 getFontSpacing():

从定义可以看出，上图中两行文字的 font spacing (即相邻两行的 baseline 的距离) 可以通过 $\text{bottom} - \text{top} + \text{leading}$ (top 的值为负，前面刚说过，记得吧?) 来计算得出。

但你真的运行一下会发现， $\text{bottom} - \text{top} + \text{leading}$ 的结果是要大于 getFontSpacing() 的返回值的。

两个方法计算得出的 font spacing 竟然不一样?

这并不是 bug，而是因为 getFontSpacing() 的结果并不是通过 FontMetrics 的标准值计算出来的，而是另外计算出来的一个值，它能够做到在两行文字不显得拥挤的前提下缩短行距，以此来得到更好的显示效果。所以如果你要对文字手动换行绘制，多数时候应该选取 getFontSpacing() 来得到行距，不但使用更简单，显示效果也会更好。

getFontMetrics() 的返回值是 FontMetrics 类型。它还有一个重载方法 getFontMetrics(FontMetrics fontMetrics)，计算结果会直接填进传入的 FontMetrics 对象，而不是重新创建一个对象。这种用法在需要频繁获取 FontMetrics 的时候性能会好些。

另外，这两个方法还有一对同样结构的对应的方法 `getFontMetricsInt()` 和 `getFontMetricsInt(FontMetricsInt fontMetrics)`，用于获取 `FontMetricsInt` 类型的结果。

2.2.3 `getTextBounds(String text, int start, int end, Rect bounds)`

获取文字的显示范围。

参数里，`text` 是要测量的文字，`start` 和 `end` 分别是文字的起始和结束位置，`bounds` 是存储文字显示范围的对象，方法在测算完成之后会把结果写进 `bounds`。

```
paint.setTextAlign(Paint.Align.LEFT);
canvas.drawText(text, 500, 150, paint);
paint.setTextAlign(Paint.Align.CENTER);
canvas.drawText(text, 500, 150 + textHeight, paint);
paint.setTextAlign(Paint.Align.RIGHT);
canvas.drawText(text, 500, 150 + textHeight * 2, paint);
```



它有一个重载方法

`getTextBounds(char[] text, int index, int count, Rect bounds)`，用法非常相似，不再介绍。

2.2.4 `float measureText(String text)`

测量文字的宽度并返回。

```
canvas.drawText(texts[0], 100, 150, paint);
canvas.drawText(texts[1], 100, 150 + paint.getFontSpacing(), paint);
```

```
canvas.drawText(texts[2], 100, 150 + paint.getFontSpacing * 2, pain
```

Hello HenCoder

咦，前面有了 `getTextBounds()`，这里怎么又有一个 `measureText()`？

如果你用代码分别使用 `getTextBounds()` 和 `measureText()` 来测量文字的宽度，你会发现 `measureText()` 测出来的宽度总是比 `getTextBounds()` 大一点。这是因为这两个方法其实测量的是两个不一样的东西。

- `getTextBounds`：它测量的是文字的显示范围（关键词：显示）。形象点来说，你这段文字外放置一个可变的矩形，然后把矩形尽可能地缩小，一直小到这个矩形恰好紧紧包裹住文字，那么这个矩形的范围，就是这段文字的 `bounds`。
- `measureText()`：它测量的是文字绘制时所占用的宽度（关键词：占用）。前面已经讲过，一个文字在界面中，往往需要占用比他的实际显示宽度更多一点的宽度，以此来让文字和文字之间保留一些间距，不会显得过于拥挤。上面的这幅图，我并没有设置 `setLetterSpacing()`，这里的 `letter spacing` 是默认值 0，但你可以看到，图中每两个字母之间都是有空隙的。另外，下方那条用于表示文字宽度的横线，在左边超出了第一个字母 `H` 一段距离的，在右边也超出了最后一个字母 `r`（虽然右边这里用肉眼不太容易分辨），而就是两边的这两个「超出」，导致了 `measureText()` 比 `getTextBounds()` 测量出的宽度要大一些。

在实际的开发中，测量宽度要用 `measureText()` 还是 `getTextBounds()`，需要根据情况而定。不过你只要掌握了上面我所说的它们的本质，在选择的时候就不会为难和疑惑了。

`measureText(String text)` 也有几个重载方法，用法和它大同小异，不再介绍。

2.2.5 `getTextWidths(String text, float[] widths)`

获取字符串中每个字符的宽度，并把结果填入参数 `widths`。

这相当于 `measureText()` 的一个快捷方法，它的计算等价于对字符串中的每个字符分别调用 `measureText()`，并把它们的计算结果分别填入 `widths` 的不同元素。

`getTextWidths()` 同样也有好几个变种，使用大同小异，不再介绍。

2.2.6 `int breakText(String text, boolean measureForwards, float maxWidth, float[] measuredWidth)`

这个方法也是用来测量文字宽度的。但和 `measureText()` 的区别是，`breakText()` 是在给出宽度上限的前提下测量文字的宽度。如果文字的宽度超出了上限，那么在临近超限的位置截断文字。

```
int measuredCount;
float[] measuredWidth = {0};

// 宽度上限 300 （不够用，截断）
measuredCount = paint.breakText(text, 0, text.length(), true, 300, 1, 1);
canvas.drawText(text, 0, measuredCount, 150, 150, paint);

// 宽度上限 400 （不够用，截断）
measuredCount = paint.breakText(text, 0, text.length(), true, 400, 1, 1);
canvas.drawText(text, 0, measuredCount, 150, 150 + fontSpacing, paint);

// 宽度上限 500 （够用）
measuredCount = paint.breakText(text, 0, text.length(), true, 500, 1, 1);
canvas.drawText(text, 0, measuredCount, 150, 150 + fontSpacing * 2, paint);

// 宽度上限 600 （够用）
measuredCount = paint.breakText(text, 0, text.length(), true, 600, 1, 1);
canvas.drawText(text, 0, measuredCount, 150, 150 + fontSpacing * 3, paint);
```

Hello Hen

截取文字个数：9

Hello HenCod

截取文字个数：12

Hello HenCoder

截取文字个数：14

Hello HenCoder

截取文字个数：14

宽度上限

测得宽度

`breakText()` 的返回值是截取的文字个数（如果宽度没有超限，则是文字的总个数）。参数中，`text` 是要测量的文字；`measureForwards` 表示文字的测量方向，`true` 表示由左往右测量；`maxWidth` 是给出的宽度上限；`measuredWidth` 是用于接受数据，而不是用于提供数据的：方法测量完成后会把截取的文字宽度（如果宽度没有超限，则为文字总宽度）赋值给 `measuredWidth[0]`。

这个方法可以用于多行文字的折行计算。

`breakText()` 也有几个重载方法，使用大同小异，不再介绍。

2.2.7 光标相关

对于 `EditText` 以及类似的场景，会需要绘制光标。光标的计算很麻烦，不过 API 23 引入了两个新的方法，有了这两个方法后，计算光标就方便了很多。

2.2.7.1 `getRunAdvance(CharSequence text, int start, int end, int contextStart, int contextEnd, boolean isRtl, int offset)`

对于一段文字，计算出某个字符处光标的 `x` 坐标。`start` `end` 是文字的起始和结束坐标；`contextStart` `contextEnd` 是上下文的起始和结束坐标；`isRtl` 是文字的方向；`offset` 是字数的偏移，即计算第几个字符处的光标。

```
int length = text.length();
float advance = paint.getRunAdvance(text, 0, length, 0, length, false);
canvas.drawText(text, offsetX, offsetY, paint);
canvas.drawLine(offsetX + advance, offsetY - 50, offsetX + advance,
```

Hello HenCoder

其实，说是测量光标位置的，本质上这这也是一个测量文字宽度的方法。上面这个例子中，start 和 contextStart 都是 0，end contextEnd 和 offset 都等于 text.length()。在这种情况下，它是等价于 measureText(text) 的，即完整测量一段文字的宽度。而对于更复杂的需求，getRunAdvance() 能做的事就比 measureText() 多了。


```
// 包含特殊符号的绘制（如 emoji 表情）
String text = "Hello HenCoder \uD83C\uDDE8\uD83C\uDDF3" // "Hello H

...

float advance1 = paint.getRunAdvance(text, 0, length, 0, length, false);
float advance2 = paint.getRunAdvance(text, 0, length, 0, length, false);
float advance3 = paint.getRunAdvance(text, 0, length, 0, length, false);
float advance4 = paint.getRunAdvance(text, 0, length, 0, length, false);
float advance5 = paint.getRunAdvance(text, 0, length, 0, length, false);
float advance6 = paint.getRunAdvance(text, 0, length, 0, length, false);

...
```


Hello HenCoder		offset = length
Hello HenCoder		offset = length - 1
Hello HenCoder		offset = length - 2
Hello HenCoder		offset = length - 3
Hello HenCoder		offset = length - 4
Hello HenCoder		offset = length - 5

如上图， 虽然占了 4 个字符（`\uD83C\uDDE8\uD83C\uDDF3`），但当 `offset` 是表情中间处时，`getRunAdvance()` 得出的结果并不会在表情的中间处。为什么？因为这是用来计算光标的方法啊，光标当然不能出现在符号中间啦。

2.2.7.2 `getOffsetForAdvance(CharSequence text, int start, int end, int contextStart, int contextEnd, boolean isRtl, float advance)`


给出一个位置的像素值，计算出文字中最接近这个位置的字符偏移量（即第几个字符最接近这个坐标）。

方法的参数很简单：`text` 是要测量的文字；`start` `end` 是文字的起始和结束坐标；`contextStart` `contextEnd` 是上下文的起始和结束坐标；`isRtl` 是文字方向；`advance` 是给出的位置的像素值。填入参数，对应的字符偏移量将作为返回值返回。

`getOffsetForAdvance()` 配合上 `getRunAdvance()` 一起使用，就可以实现「获取用户点击处的文字坐标」的需求。

2.2.8 `hasGlyph(String string)`

检查指定的字符串中是否是一个单独的字形 (glyph) 。最简单的情况是，string 只有一个字母（比如 a）。

原字符串	对应的显示	hasGlyph() 返回值
a	a	true
b	b	true
ab	ab（两个字符不算一个字形哦）	false
\uD83C\uDDE8\uD83C\uDDF3		true

以上这些内容，就是文字绘制的相关知识。它们有的常用，有的不常用，有的甚至可以说是在某些情况下没用，不过你把它们全部搞懂了，在实际的开发中，就知道哪些事情可以做到，哪些事情做不到，以及应该怎么做了。

练习项目

为了避免转头就忘，强烈建议你趁热打铁，做一下这个练习项目：

[HenCoderPracticeDraw3](#)

HenCoder 绘制 3

setTextSize()

setTypeface()

setFakeBoldText()

Hello HenCoder

Hello HenCoder

Hello HenCoder

HenCoder 绘制 3

setFontSpacing()

measureText()

getTextBounds()

三个月内你胖了4.5公斤

HenCoder 绘制 3

measureText()

getTextBounds()

getFontMetrics()

AaJj Ââ

HenCoder 绘制 3

getTextBounds()

getFontMetrics()

AaJj Ââ

