HenCoder Android 开发进阶
自定义 View 1-5 绘制顺序

简介

前面几期讲的是「术」，是「用哪些 API 可以绘制什么内容」。到上一期为止，「术」已经讲完了，接下来要讲的是「道」，是「怎么去安排这些绘制」。

这期是「道」的第一期：绘制顺序。

Android 里面的绘制都是按顺序的，先绘制的内容会被后绘制的盖住。比如你在重叠的位置先画圆再画方，和先画方再画圆所呈现出来的结果肯定是不同的：

先画圆再画方

先画方再画圆

而在实际的项目中，绘制内容相互遮盖的情况是很普遍的，那么怎么实现自己需要的遮盖关系，就是这期要讲的内容。

1 super.onDraw() 前 or 后?

前几期我写的自定义绘制，全都是直接继承 `View` 类，然后重写它的 `onDraw()` 方法，把绘制代码写在里面，就像这样：

```
public class AppView extends View {
    ...

    protected void onDraw(Canvas canvas) {
        super.onDraw(canvas);
```

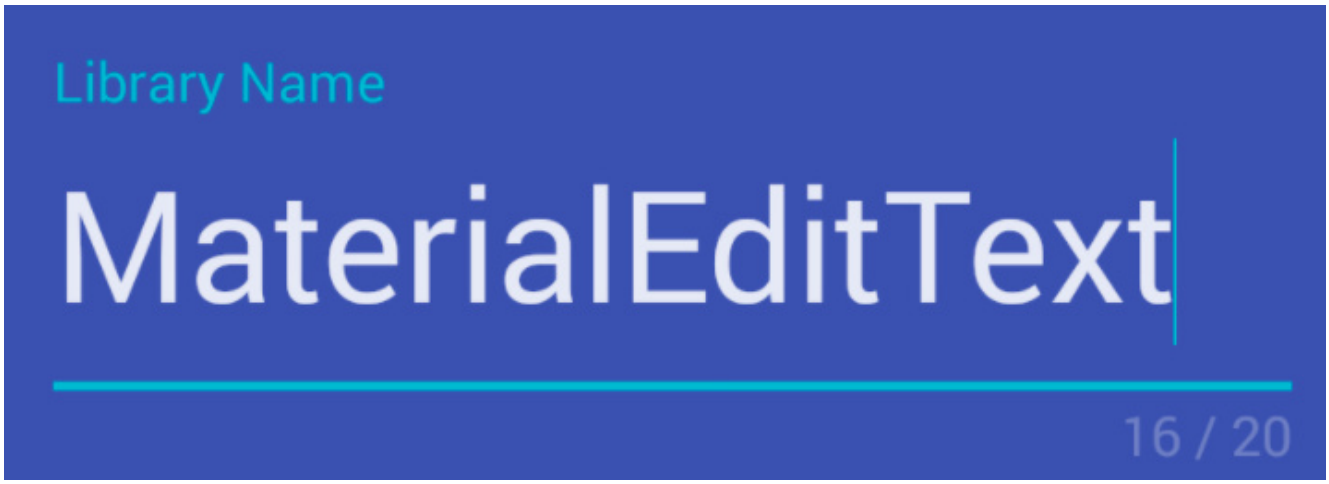
```
        ... // 自定义绘制代码
    }

    ...
}
```

这是自定义绘制最基本的形态：继承 View 类，在 onDraw() 中完全自定义它的绘制。

在之前的样例中，我把绘制代码全都写在了 super.onDraw() 的下面。不过其实，绘制代码写在 super.onDraw() 的上面还是下面都无所谓，甚至，你把 super.onDraw() 这行代码删掉都没关系，效果都是一样的——因为在 View 这个类里， onDraw() 本来就是空实现：

```
// 在 View.java 的源码中，onDraw() 是空的
// 所以直接继承 View 的类，它们的 super.onDraw() 什么也不会做
public class View implements Drawable.Callback,
        KeyEvent.Callback, AccessibilityEventSource {
    ...

    /**
     * Implement this to do your drawing.
     *
     * @param canvas the canvas on which the background will be dra
     */
    protected void onDraw(Canvas canvas) {
    }

    ...
}
```

然而，除了继承 View 类，自定义绘制更为常见的情况是，继承一个具有某种功能的控件，去重写它的 onDraw() ，在里面**添加**一些绘制代码，做出一个「进化版」的控件：



> 基于 `EditText`，在它的基础上增加了顶部的 Hint Text 和底部的字符计数。

而这种基于已有控件的自定义绘制，就不能不考虑 `super.onDraw()` 了：你需要根据自己的需求，判断出你绘制的内容需要盖住控件原有的内容还是需要被控件原有的内容盖住，从而确定你的绘制代码是应该写在 `super.onDraw()` 的上面还是下面。

1.1 写在 super.onDraw() 的下面

把绘制代码写在 `super.onDraw()` 的下面，由于绘制代码会在原有内容绘制结束之后才执行，所以绘制内容就会盖住控件原来的内容。

这是最为常见的情况：为控件增加点缀性内容。比如，在 Debug 模式下绘制出 ImageView 的图像尺寸信息：

```
public class AppImageView extends ImageView {
    ...

    protected void onDraw(Canvas canvas) {
        super.onDraw(canvas);

        if (DEBUG) {
            // 在 debug 模式下绘制出 drawable 的尺寸信息
            ...
        }
    }
}
```

> 这招很好用的，试过吗？

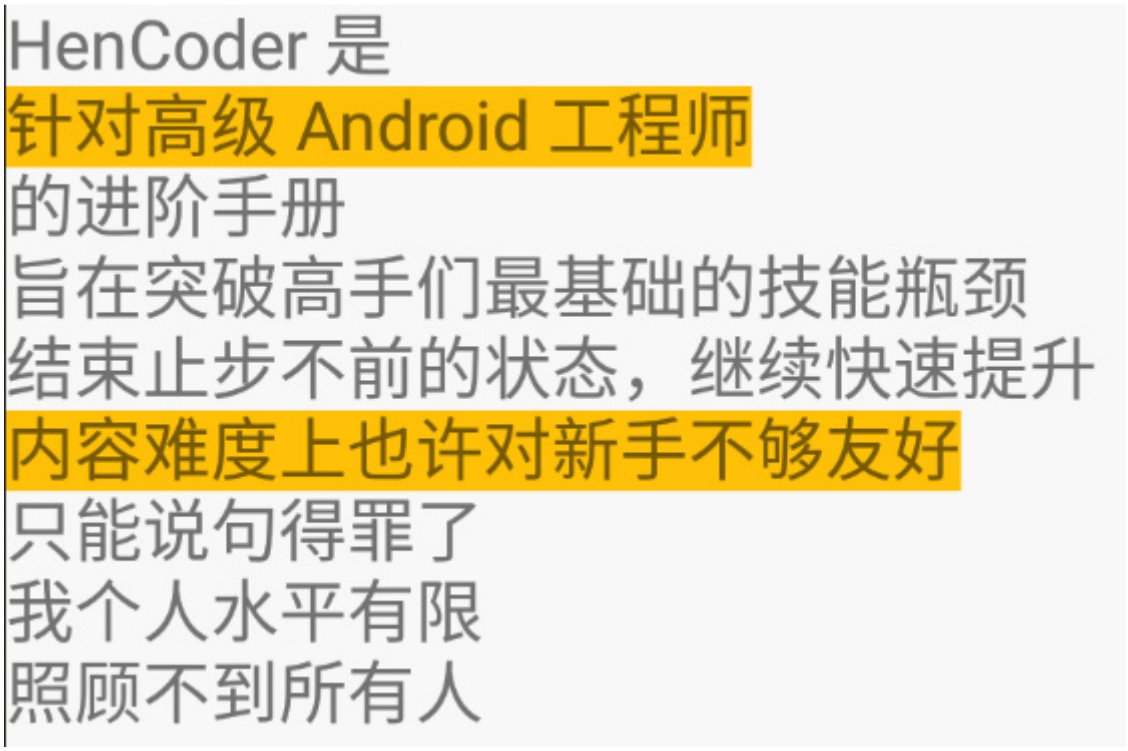
当然，除此之外还有其他的很多用法，具体怎么用就取决于你的需求、经验和想象力了。

1.2 写在 super.onDraw() 的上面

如果把绘制代码写在 `super.onDraw()` 的上面，由于绘制代码会执行在原有内容的绘制之前，所以绘制的内容会被控件的原内容盖住。

相对来说，这种用法的场景就会少一些。不过只是少一些而不是没有，比如你可以通过在文字的下层绘制纯色矩形来作为「强调色」：

```
public class AppTextView extends TextView {
    ...

    protected void onDraw(Canvas canvas) {
        ... // 在 super.onDraw() 绘制文字之前，先绘制出被强调的文字的背景

        super.onDraw(canvas);
    }
}
```



2 dispatchDraw()：绘制子 View 的方法

讲了这几期，到目前为止我只提到了 `onDraw()` 这一个绘制方法。但其实绘制方法不是只有一个的，而是有好几个，其中 `onDraw()` 只是负责自身主体内容绘制的。而有的时候，你想要的遮盖关系无法通过 `onDraw()` 来实现，而是需要通过别的绘制方法。

例如，你继承了一个 `LinearLayout`，重写了它的 `onDraw()` 方法，在 `super.onDraw()` 中插入了你自己的绘制代码，使它能够在内部绘制一些斑点作为点缀：

```
public class SpottedLinearLayout extends LinearLayout {
    ...

    protected void onDraw(Canvas canvas) {
        super.onDraw(canvas);

        ... // 绘制斑点
    }
}
```

看起来没问题对吧?

但是你会发现，当你添加了子 View 之后，你的斑点不见了：

```
<SpottedLinearLayout
    android:orientation="vertical"
    ... >

    <ImageView ... />

    <TextView ... />

</SpottedLinearLayout>
```

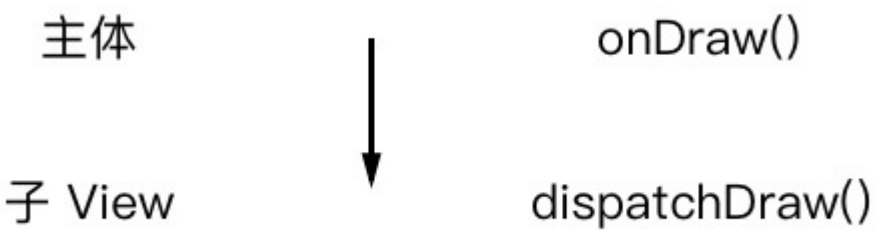
期望的效果

实际效果

造成这种情况的原因是 Android 的绘制顺序：在绘制过程中，每一个 ViewGroup 会先调用自己的 `onDraw()` 来绘制完自己的主体之后再去绘制它的子 View。对于上面这个例子来说，就是你的 `LinearLayout` 会在绘制完斑点后再去绘制它的子 View。那么在子 View 绘制完成之后，先前绘制的斑点就被子 View 盖住了。

具体来讲，这里说的「绘制子 View」是通过另一个绘制方法的调用来发生的，这个绘制方法叫做： `dispatchDraw()` 。也就是说，在绘制过程中，每个 View 和 ViewGroup 都会先调用 `onDraw()` 方法来绘制主体，再调用 `dispatchDraw()` 方法来绘制子 View。

> 注：虽然 `View` 和 `ViewGroup` 都有 `dispatchDraw()` 方法，不过由于 View 是没有子 View 的，所以一般来说 `dispatchDraw()` 这个方法只对 `ViewGroup` （以及它的子类）有意义。



回到刚才的问题：怎样才能让 `LinearLayout` 的绘制内容盖住子 View 呢？只要让它的绘制代码在子 View 的绘制之后再执行就好了。

2.1 写在 super.dispatchDraw() 的下面

只要重写 `dispatchDraw()` ，并在 `super.dispatchDraw()` 的下面写上你的绘制代码，这段绘制代码就会发生在子 View 的绘制之后，从而让绘制内容盖住子 View 了。

```
public class SpottedLinearLayout extends LinearLayout {
    ...

    // 把 onDraw() 换成了 dispatchDraw()
    protected void dispatchDraw(Canvas canvas) {
        super.dispatchDraw(canvas);
```

```
        super.dispatchDraw(canvas);

        ... // 绘制斑点
    }
}
```

> 好萌的蝙蝠侠啊

2.2 写在 super.dispatchDraw() 的上面

同理，把绘制代码写在 `super.dispatchDraw()` 的上面，这段绘制就会在 `onDraw()` 之后、 `super.dispatchDraw()` 之前发生，也就是绘制内容会出现在主体内容和子 View 之间。而这个……

其实和前面 1.1 讲的，重写 `onDraw()` 并把绘制代码写在 `super.onDraw()` 之后的做法，效果是一样的。

能想明白为什么吧？图就不上了。

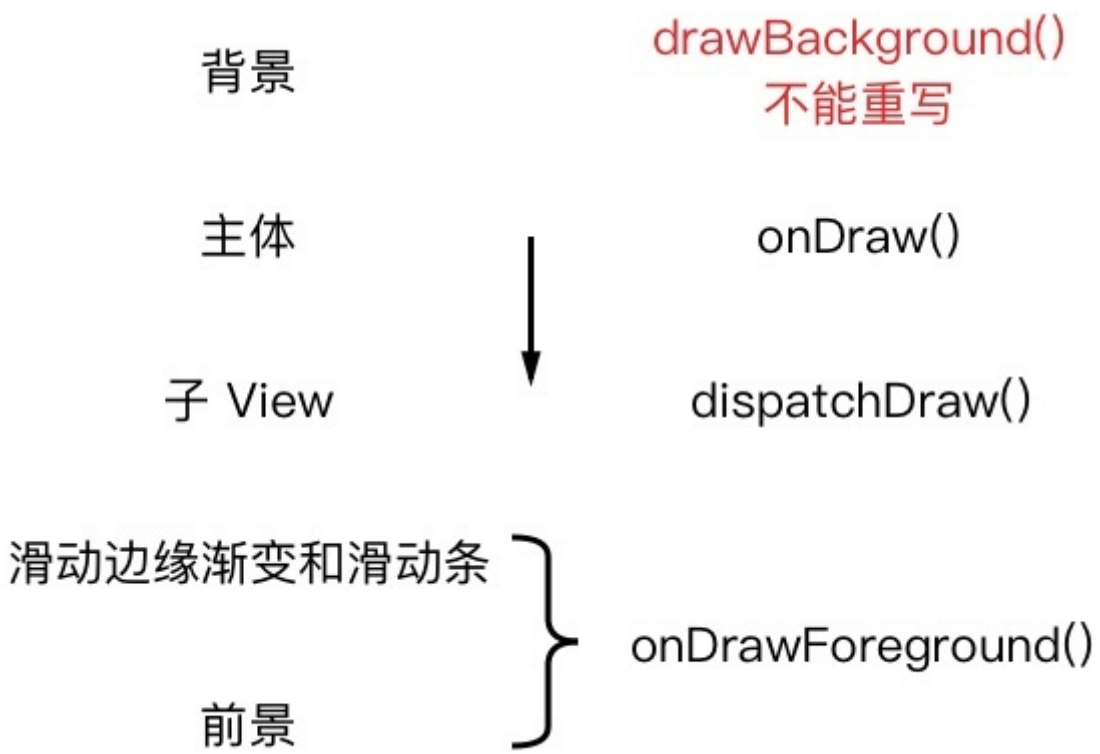
3 绘制过程简述

绘制过程中最典型的两个部分是上面讲到的主体和子 View，但它们并不是绘制过程的全部。除此之外，绘制过程还包含一些其他内容的绘制。具体来讲，一个完整的绘制过程会依次绘制以下几个内容：

1. 背景
2. 主体（`onDraw()`）
3. 子 View（`dispatchDraw()`）
4. 滑动边缘渐变和滑动条
5. 前景

一般来说，一个 View（或 ViewGroup）的绘制不会这几项全都包含，但必然逃不出这几项，并且一定会严格遵守这个顺序。例如通常一个 `LinearLayout` 只有背景和子 View，那么它会先绘制背景再绘制子 View；一个 `ImageView` 有主体，有可能会再加上一层半透明的前景作为遮罩，那么它的前景也会在主体之后进行绘制。需要注意，前景的支持是在 Android 6.0（也就是 API 23）才加入的；之前其实也有，不过只支持 `FrameLayout`，而直到 6.0 才把这个支持放进了 `View` 类里。

这其中的第 2、3 两步，前面已经讲过了；第 1 步——背景，它的绘制发生在一个叫 `drawBackground()` 的方法里，但这个方法是 `private` 的，不能重写，你如果要设置背景，只能用自带的 API 去设置（xml 布局文件的 `android:background` 属性以及 Java 代码的 `View.setBackgroundXxx()` 方法，这个每个人都用得很 6 了），而不能自定义绘制；而第 4、5 两步——滑动边缘渐变和滑动条以及前景，这两部分被合在一起放在了 `onDrawForeground()` 方法里，这个方法是可以重写的。



滑动边缘渐变和滑动条可以通过 xml 的 `android:scrollbarXXX` 系列属性或 Java 代码的 `View.setXXXScrollbarXXX()` 系列方法来设置；前景可以通过 xml 的 `android:foreground` 属性或 Java 代码的 `View.setForeground()` 方法来设置。而重写 `onDrawForeground()` 方法，并在它的 `super.onDrawForeground()` 方法的上面或下面插入绘制代码，则可以控制绘制内容和滑动边缘渐变、滑动条以及前景的遮盖关系。

4 onDrawForeground()

> 首先，再说一遍，这个方法是 API 23 才引入的，所以在重写这个方法的时候要确认你的 `minSdk` 达到了 23，不然低版本的手机装上你的软件会没有效果。

在 `onDrawForeground()` 中，会依次绘制滑动边缘渐变、滑动条和前景。所以如果你重写 `onDrawForeground()` ：

4.1 写在 super.onDrawForeground() 的下面

如果你把绘制代码写在了 `super.onDrawForeground()` 的下面，绘制代码会在滑动边缘渐变、滑动条和前景之后被执行，那么绘制内容将会盖住滑动边缘渐变、滑动条和前景。

```
public class AppImageView extends ImageView {
    ...

    public void onDrawForeground(Canvas canvas) {
        super.onDrawForeground(canvas);

        ... // 绘制「New」标签
    }
}
```

```
<!-- 使用半透明的黑色作为前景，这是一种很常见的处理 -->
<AppImageView
    ...
    android:foreground="#88000000" />
```

原效果

绘制后的效果

> 左上角的标签并没有被黑色遮罩盖住，而是保持了原有的颜色。

4.2 写在 super.onDrawForeground() 的上面

如果你把绘制代码写在了 `super.onDrawForeground()` 的上面，绘制内容就会在 `dispatchDraw()` 和 `super.onDrawForeground()` 之间执行，那么绘制内容会盖住子 View，但被滑动边缘渐变、滑动条以及前景盖住：

```
public class AppImageView extends ImageView {
    ...

    public void onDrawForeground(Canvas canvas) {
        ... // 绘制「New」标签

        super.onDrawForeground(canvas);
    }
}
```

原效果

绘制后的效果

> 由于被半透明黑色遮罩盖住，左上角的标签明显变暗了。

这种写法，和前面 2.1 讲的，重写 `dispatchDraw()` 并把绘制代码写在 `super.dispatchDraw()` 的下面的效果是一样的：绘制内容都会盖住子 View，但被滑动边缘渐变、滑动条以及前景盖住。

4.3 想在滑动边缘渐变、滑动条和前景之间插入绘制代码?

很简单：不行。

虽然这三部分是依次绘制的，但它们被一起写进了 `onDrawForeground()` 方法里，所以你要么把绘制内容插在它们之前，要么把绘制内容插在它们之后。而想往它们之间插入绘制，是做不到的。

5 draw() 总调度方法

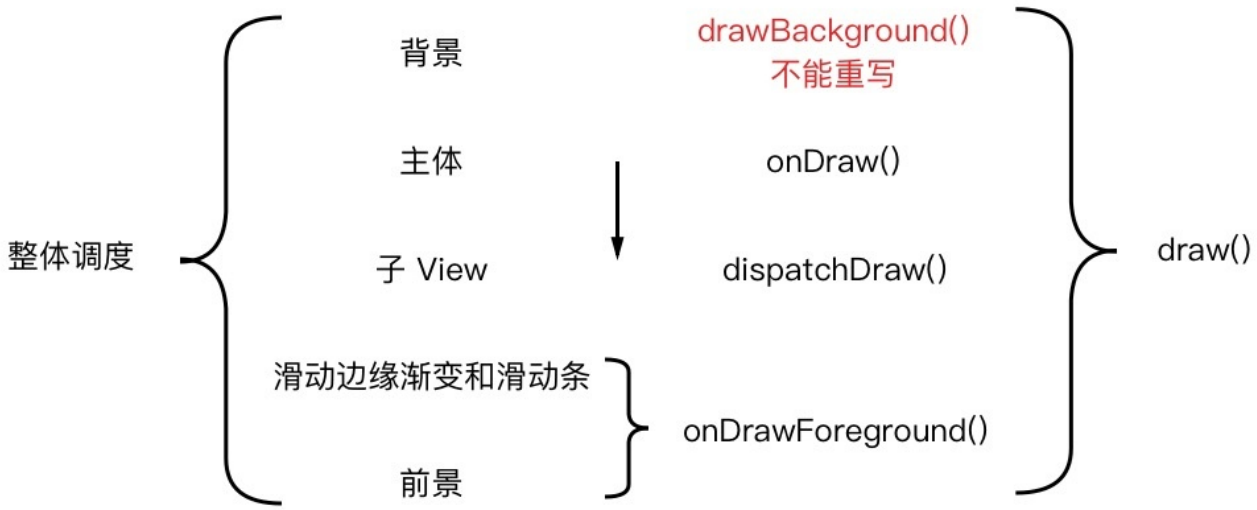
除了 onDraw() dispatchDraw() 和 onDrawForeground() 之外，还有一个可以用来实现自定义绘制的方法： draw() 。

draw() 是绘制过程的总调度方法。一个 View 的整个绘制过程都发生在 draw() 方法里。前面讲到的背景、主体、子 View 、滑动相关以及前景的绘制，它们其实都是在 draw() 方法里的。

```
// View.java 的 draw() 方法的简化版大致结构（是大致结构，不是源码哦）：

public void draw(Canvas canvas) {
    ...

    drawBackground(Canvas); // 绘制背景（不能重写）
    onDraw(Canvas); // 绘制主体
    dispatchDraw(Canvas); // 绘制子 View
    onDrawForeground(Canvas); // 绘制滑动相关和前景

    ...
}
```

从上面的代码可以看出， onDraw() dispatchDraw() onDrawForeground() 这三个方法在 draw() 中被依次调用，因此它们的遮盖关系也就像前面所说的

——`dispatchDraw()` 绘制的内容盖住 `onDraw()` 绘制的内容；`onDrawForeground()` 绘制的内容盖住 `dispatchDraw()` 绘制的内容。而在它们的外部，则是由 `draw()` 这个方法作为总的调度。所以，你也可以重写 `draw()` 方法来做自定义的绘制。



5.1 写在 super.draw() 的下面

由于 `draw()` 是总调度方法，所以如果把绘制代码写在 `super.draw()` 的下面，那么这段代码会在其他所有绘制完成之后再执行，也就是说，它的绘制内容会盖住其他的所有绘制内容。

它的效果和重写 `onDrawForeground()`，并把绘制代码写在 `super.onDrawForeground()` 下面时的效果是一样的：都会盖住其他的所有内容。

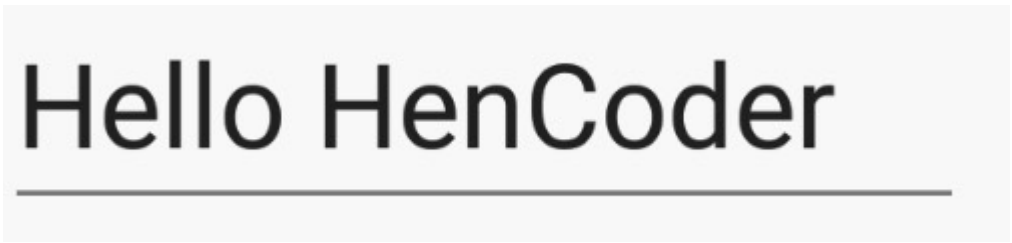
> 当然了，虽说它们效果一样，但如果你既重写 `draw()` 又重写 `onDrawForeground()`，那么 `draw()` 里的内容还是会盖住 `onDrawForeground()` 里的内容的。所以严格来讲，它们的效果还是有一点点不一样的。
>
> 但这属于抬杠……

5.2 写在 super.draw() 的上面

同理，由于 `draw()` 是总调度方法，所以如果把绘制代码写在 `super.draw()` 的上面，那么这段代码会在其他所有绘制之前被执行，所以这部分绘制内容会被其他所有的内容盖住，包括背景。是的，背景也会盖住它。

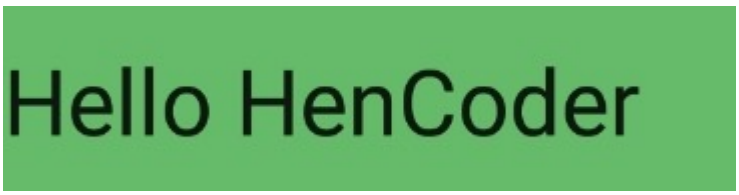
是不是觉得没用？觉得怎么可能会有谁想要在背景的下面绘制内容？别这么想，有的时候它还真的有用。

例如我有一个 `EditText`：



它下面的那条横线，是 `EditText` 的背景。所以如果我想给这个 `EditText` 加一个绿色的底，我不能使用给它设置绿色背景色的方式，因为这就相当于是把它的背景替换掉，从而会导致下面的那条横线消失：

```
<EditText
    ...
    android:background="#66BB6A" />
```



> EditText：我到底是个 EditText 还是个 TextView？傻傻分不清楚。

在这种时候，你就可以重写它的 draw() 方法，然后在 super.draw() 的上方插入代码，以此来在所有内容的底部涂上一片绿色：

```
public AppEditText extends EditText {
    ...

    public void draw(Canvas canvas) {
        canvas.drawColor(Color.parseColor("#66BB6A")); // 涂上绿色

        super.draw(canvas);
    }
}
```

当然，这种用法并不常见，事实上我也并没有在项目中写过这样的代码。但我想说的是，我们作为工程师，是无法预知将来会遇到怎样的需求的。我们能做的只能是尽量地去多学习一些、多掌握一些，尽量地了解我们能够做什么、怎么做，然后在需求到来的时候，就可以多一些自如，少一些束手无策。

注意

关于绘制方法，有两点需要注意一下：

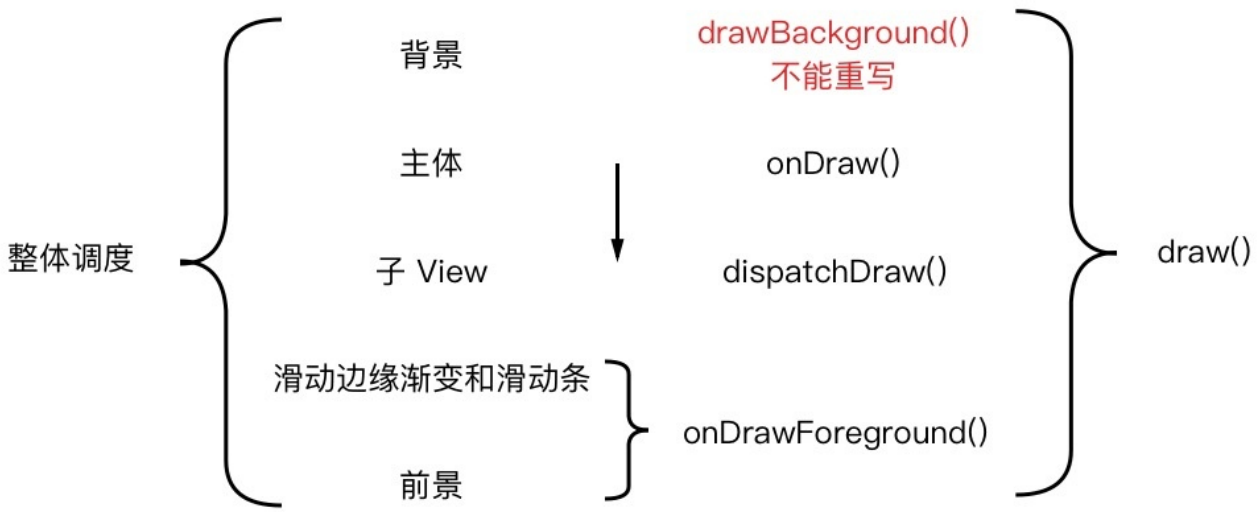
1. 出于效率的考虑，ViewGroup 默认会绕过 draw() 方法，换而直接执行 dispatchDraw()，以此来简化绘制流程。所以如果你自定义了某个 ViewGroup 的子类（比如 LinearLayout）并且需要在它的除 dispatchDraw() 以外的任何一个绘制方法内绘制内容，你**可能**会需要调用 View.setWillNotDraw(false) 这行代码来切换到完整的绘制流程（是「可

能」而不是「必须」的原因是，有些 ViewGroup 是已经调用过 `setWillNotDraw(false)` 了的，例如 `ScrollView`）。

2. 有的时候，一段绘制代码写在不同的绘制方法中效果是一样的，这时你可以选一个自己喜欢或者习惯的绘制方法来重写。但有一个例外：如果绘制代码既可以写在 `onDraw()` 里，也可以写在其他绘制方法里，那么优先写在 `onDraw()`，因为 Android 有相关的优化，可以在不需要重绘的时候自动跳过 `onDraw()` 的重复执行，以提升开发效率。享受这种优化的只有 `onDraw()` 一个方法。

总结

今天的内容就是这些：使用不同的绘制方法，以及在重写的时候把绘制代码放在 `super.绘制方法()` 的上面或下面不同的位置，以此来实现需要的遮盖关系。下面用一张图和一个表格总结一下：



> 嗯，上面这张图在前面已经贴过了，不用比较了完全一样的。

重写的方法	绘制代码的位置	绘制内容出现的位置
onDraw()	super.onDraw() 之前	背景和原主体内容之间
	super.onDraw() 之后	原主体内容和子 View 之间
dispatchDraw()	super.dispatchDraw() 之前	
	super.dispatchDraw() 之后	子 View 和前景之间
onDrawForeground()	super.onDrawForeground() 之前	
	super.onDrawForeground() 之后	盖住前景
draw()	super.draw() 之前	被背景盖住
	super.draw() 之后	盖住前景

另外别忘了上面提到的那两个注意事项：

1. 在 `ViewGroup` 的子类中重写除 `dispatchDraw()` 以外的绘制方法时，可能需要调用 `setWillNotDraw(false)`；
2. 在重写的方法有多个选择时，优先选择 `onDraw()`。

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：
HenCoderPracticeDraw5