YYText はどのように動作するのか

上の影の効果は、以下のコードで実現されています：

先生が YYTextShadow を生成し、それを attributedString の yy_textShadow に代入し、その後 attributedString を YYLabel に代入し、さらに YYLabel を UIView に追加して表示していることがわかります。yy_textShadow を追跡すると、主に textShadow が NSAttributedString の attribute にバインドされており、キーは YYTextShadowAttributeName、値は textShadow であることがわかります。つまり、最初に shadow を保存し、後で使用するという流れです。Shift + Command + J を使って定義に素早くジャンプできます。

ここに addAttribute というものがあります。これは NSAttributedString.h で定義されています：

```
- (void)addAttribute:(NSString *)name value:(id)value range:(NSRange)range;
```

このメソッドは、指定された属性を指定された範囲のテキストに追加します。name は属性の名前を表し、value はその属性の値です。range は、属性を適用するテキストの範囲を指定します。

説明によると､任意のキーと値のペアを代入することができます｡そして､YYTextShadowAttributeName の定義は普通の文字列であり、これは最初に shadow の情報を保存し、後で使用することを意味します。YYTextShadowAttributeName をグローバルに検索してみましょう。

次に、YYTextLayout 内の YYTextDrawShadow 関数を見ていきましょう：

CGContextTranslateCTM は、コンテキスト内の原点座標を変更することを意味します。したがって、

```
CGContextTranslateCTM(context, point.x, point.y);
```

このコードは、Core Graphics のコンテキスト（context）に対して、指定された点（point.x と point.y）に基づいて座標系を平行移動（translate）します。具体的には、描画コンテキストの原点が point.x と point.y だけ移動されます。これにより、その後の描画操作は新しい原点を基準として行われます。

描画のコンテキストを point 点に移動することを意味します。まず、どこで YYTextDrawShadow が呼び出されているのかを確認し、それが drawInContext 内で呼び出されていることを発見しました。

drawInContext 内では、順番にブロックの枠線を描画し、その後、背景の枠線、影、下線、テキスト、装飾、内側の影、取り消し線、テキストの枠線、デバッグ用の線を描画します。

では、実際にどこで drawInContext が使われているのでしょうか？その中に YYTextDebugOption というパラメータがあることがわかります。したがって、この関数はシステムのコールバックではなく、YYText 内部で独自に呼び出されていることが確実です。

Ctrl + 1 を押してショートカットを表示すると、4 つの場所で呼び出されていることがわかります。

drawInContext:size:debug は依然として YYText 自身の呼び出しです。なぜなら、debug の型は YYTextDebugOption * であり、これは YY 自身のものだからです。newAsyncTask はシステムの呼び出しには見えず、addAttachmentToView:layer: も同様です。したがって、これらはおそらく drawRect: の呼び出しである可能性が高いです。

確かに、右側のクイックヘルプを見ると、詳細な説明があり、ヘルプの下には UIView で定義されていると書かれています。さらに YYTextContainerView を見ると、これは UIView を継承しています。

YYLabel は YYTextContainerView を使っているのですか？そしてシステムに YYTextContainerView 内の drawRect: を呼び出させて描画させているのですか？

奇妙ですね、YYLabel は UIView を継承しています。つまり、YYText には 2 つのセットが存在するはずです！1 つは YYLabel、もう 1 つは YYTextView で、UILabel と UITextView のように。それでは、先ほどの YYLabel の newAsyncDisplayTask をもう一度見てみましょう。

長いコードの中間で YYTextLayout の drawInContext が呼び出されています。newAsyncDisplayTask はどこで呼び出されているのでしょうか？

2 行目で呼び出されました。したがって、簡単に理解すると、YYLabel はテキストを描画するために非同期を使用していると言えます。そして、_displayAsync は上記の display によって呼び出されています。display のドキュメントを見ると、システムが適切なタイミングで呼び出してレイヤーの内容を更新すると書かれています。直接呼び出すべきではありません。また、ブレークポイントを設定することもできます。

display は CALayer のトランザクション中に呼び出されることを説明します。なぜトランザクションを使うのかというと、おそらく更新をバッチ処理して効率を上げるためでしょう。データベースのようなロールバックの必要性はなさそうです。

display のシステムドキュメントには、もしあなたのレイヤーを異なる方法で描画したい場合、このメソッドをオーバーライドして独自の描画を実装できると書かれています。

したがって、簡単にいくつかのアイデアを得ました。YYLabel は UIView の display メソッドをオーバーライドして、自身の影などの効果を非同期に描画します。影の効果はまず YYLabel の attributedText の属性に保存され、display メソッドで描画する際に取り出されます。描画時にはシステムの CoreGraphics フレームワークが使用されます。

いくつかの考えを整理した後、本当に強力なものは何かがわかります。一方では、これだけの効果や非同期呼び出しなどを組織化する能力であり、もう一方では、基盤となる CoreGraphics フレームワークの熟練した運用です。したがって、前のコードの組織化について少し理解した後、CoreGraphics フレームワークに深く入り込んでいきます。どのように描画されているのかを見てみましょう。

YYTextDrawShadow に戻りましょう。

ここでは、CGContextSaveGState と CGContextRestoreGState が描画コードの一部を囲んでいます。CGContextSaveGState の意味は、現在の描画状態をコピーして、描画スタックにプッシュすることです。各描画コンテキストは、描画スタックを維持しています。スタックの内部操作については詳しくはわかりませんが、とりあえず、描画コンテキストの前に CGContextSaveGState を呼び出し、描画コンテキストの後に CGContextRestoreGState を呼び出すことで、その間の描画がコンテキストに有効に表示されると理解しておきます。CGContextTranslateCTM は、コンテキストを対応する位置に移動します。まず point.x と point.y に移動し、描画の対応位置に移動します。その後、0 と size.height に移動する理由はまだわからないので、後でまた確認します。次に lines を取り出し、for ループを実行します。

lines とは何ですか？YYTextLayout 内の (YYTextLayout *)layoutWithContainer:(YYTextContainer *)container text:(NSAttributedString *)text range:(NSRange)range で値が設定されているのを見つけました。

次に、この関数の定義部分に移動します：

この関数は非常に長く、367 行から 861 行まで、500 行ものコードがあります！最初と最後を見ると、その目的はこれらの変数を取得することだとわかります。lines はどのように取得されるのでしょうか？

大きな for ループの中で、1 行ずつ line を lines に追加しているのが見られます。では、lineCount はどのように得られるのでしょうか？

472 行目では、framesetter オブジェクトが作成され、text パラメータは NSAttributedString です。その後、frameSetter オブジェクト内に CTFrameRef が作成され、CTFrameRef から lines が取得されます。line とは一体何でしょうか？ここにブレークポイントを設定して確認してみましょう。

発見しましたが、shadow という単語の lineCount = 2 は、私たちが想像していた文字数ではありませんでした。

なので、白い Shadow 全体が一つの line で、影も一つの line だと推測していますか？

YYText にはいくつかの例がありますが、そのうちの 1 つの効果だけを表示し、他のコードをコメントアウトしています。奇妙なことに、Shadow の lineCount = 2 で、Multiple Shadows の

lineCount も 2 です。しかし、Multiple Shadows には内側の影もあるので、3 つになるはずですよね？

CTLine の Apple ドキュメントを調べると、CTLine は一行のテキストを表し、CTLine オブジェクトは一連の glyph runs を含んでいると書かれています。つまり、単純に行数のことです！上のブレークポイントのスクリーンショットを見ると、先ほど shadow が 2 だったのは、そのテキストが shadow\n\n だったからです。先ほど、\n\n は意図的に追加され、見た目を良くするためでした：

したがって、shadow\n\n は 2 行のテキストです。CTLine は、私たちが普段「行」と呼んでいるものです。次に、lineCount を見てみましょう：

ここで CTLines 配列を取得し、その中の要素数を取得します。そして、lineCount が 0 より大きい場合、各行の座標原点を取得します。さて、lineCount が得られたので、次に for ループを見ていきましょう。

ctLines 配列から CTLine を取得し、次に YYTextLine オブジェクトを取得して、それを lines 配列に追加します。その後、line のフレーム計算を行います。YYTextLine のコンストラクタはシンプルで、位置、垂直レイアウトかどうか、CTLine オブジェクトを保存します：

lines を理解した後、以前の YYTextDrawShadow に戻りましょう：

これでコードがシンプルになりました。まず行数を取得し、それを走査します。次に GlyphRuns 配列を取得し、それを走査します。GlyphRun は図形や描画単位と理解できます。その後、attributes 配列を取得し、以前の YYTextShadowAttributeName を使用して、最初に割り当てた shadow を取得します。そして、影の描画を開始します：

while ループを使用して、サブシャドウを繰り返し描画します。CGContextSetShadowWithColor を呼び出して、シャドウのオフセット、半径、色を設定します。その後、YYTextDrawRun を呼び出して実際に描画を行います。YYTextDrawRun は 3 つの場所で呼び出されます：

内側の影やテキストの影、テキスト自体を描画するために使用されます。これは、Run オブジェクトを描画するための汎用的なメソッドであることを示しています。

最初にテキストの変換行列を取得し、runTextMatrixIsID を使用してそれが変更されていないかどうかを確認します。もし垂直レイアウトでないか、または図形変換が設定されていない場合は、直接描画を開始します。CTRunDraw を呼び出して run オブジェクトを描画します。その後、ブレークポイントを設定して、最初の影を描画する際に if 文の中に入るが、else 文には入らないことを確認します。

以上で、私たちの影の描画は終了です！

まとめると、YYLabel はまず、影などの効果を attributedText の attributes に保存し、UIView の display メソッドをオーバーライドします。display メソッド内で非同期描画を行い、CoreText

フレームワークを使用して CTLine や CTRun オブジェクトを取得します。CTRun から attributes を取得し、その後、attributes 内の各プロパティに基づいて、CoreGraphics フレームワークを使用して CTRun オブジェクトを Context に描画します。

理解がまだ十分でないので、後でまた読み返してみようと思います。YY の強さに改めて感嘆せざるを得ません！今日は考えを整理し、コードを書きながら読み進めることで、単調さを避けつつ、皆さんの参考にもなるようにしました。そろそろ寝る時間です。