作者 __微凉 (/users/8179cea7ee37) 2015.08.18 12:38*
写了21323字，被536人关注，获得了437个喜欢
(/users/8179cea7ee37)

源码笔记---MBProgressHUD

字数6320 阅读10131 评论8 喜欢86

前言

作为初学者,想要快速提高自己的水平,阅读一些优秀的第三方源代码是一个非常好的途径.通过看别人的代码,可以学习不一样的编程思路,了解一些没有接触过的类和方法.MBProgressHUD是一个非常受欢迎的第三方库,其用法简单,代码朴实易懂,涉及的知识点广而不深奥,是非常适合初学者阅读的一份源码.

一. 模式

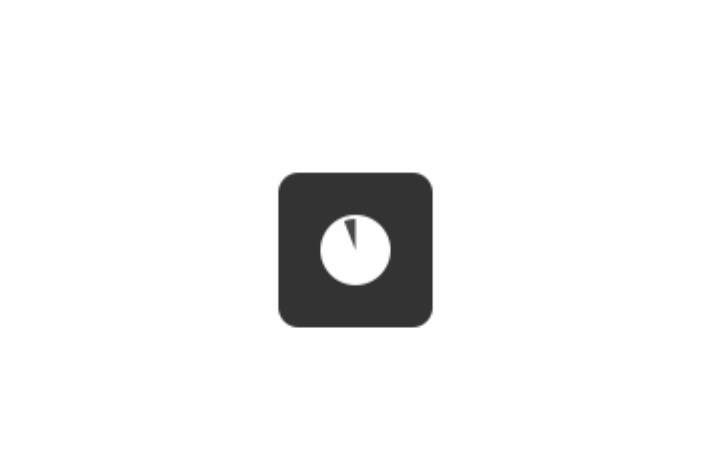
首先，`MBProgressHUD` 有以下几种视图模式.

```
typedef enum {
    /** 默认模式,使用系统自带的指示器 ,不能显示进度,只能不停地转呀转*/
    MBProgressHUDModeIndeterminate,
    /** 用饼图显示进度 */
    MBProgressHUDModeDeterminate,
    /** 进度条 */
    MBProgressHUDModeDeterminateHorizontalBar,
    /** 圆环 */
    MBProgressHUDModeAnnularDeterminate,
    /** 自定义视图 */
    MBProgressHUDModeCustomView,
    /** 只显示文字 */
    MBProgressHUDModeText
} MBProgressHUDMode;
```

`mode` 属性指定显示模式

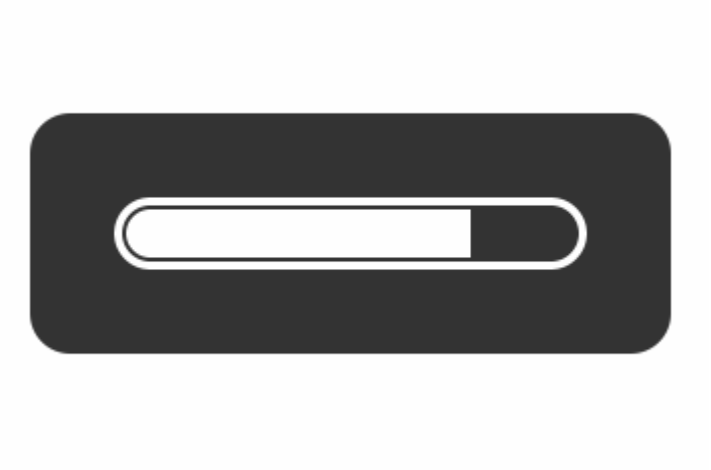
默认使用的系统自带指示器

```
hud.mode = MBProgressHUDModeIndeterminate;
```



饼图

```
hud.mode = MBProgressHUDModeDeterminate;
```



进度条

```
    hud.mode = MBProgressHUDModeDeterminateHorizontalBar;
```

圆环

```
    hud.mode = MBProgressHUDModeAnnularDeterminate;
```

只显示文字

```
    hud.mode = MBProgressHUDModeText;
    hud.labelText = @"MBProgressHUDModeText";
```

二. 结构

MBProgressHUD 由指示器,文本框,详情文本框,背景框4个部分组成.

结构组成

```
// 文本框和其相关属性
@property (copy) NSString *labelText;
@property (MB_STRONG) UIFont* labelFont;
@property (MB_STRONG) UIColor* labelColor;

//详情文本框和其相关属性
@property (copy) NSString *detailsLabelText;
@property (MB_STRONG) UIFont* detailsLabelFont;
@property (MB_STRONG) UIColor* detailsLabelColor;

// 背景框的透明度，默认值是0.8
@property (assign) float opacity;
// 背景框的颜色，如果设置了这个属性，则opacity属性会失效，即不会有半透明效果
@property (MB_STRONG) UIColor *color;
// 背景框的圆角半径。默认值是10.0
@property (assign) float cornerRadius;
// 菊花的颜色，默认是白色
@property (MB_STRONG) UIColor *activityIndicatorColor;
```

三. 初始化方法

```
- (id)initWithFrame:(CGRect)frame {
    self = [super initWithFrame:frame];
    if (self) {

        // 显示隐藏时的动画模式
        self.animationType = MBProgressHUDAnimationFade;
          // 默认指示器是菊花
        self.mode = MBProgressHUDModeIndeterminate;
        ......

        // 关闭绘制的"性能开关",如果alpha不为1,最好将opaque设为NO,让绘图系统优化性能
        self.opaque = NO;

          // 使背景颜色为透明
        self.backgroundColor = [UIColor clearColor];

          // 即使用户创建了一个hud,并调用了addSubview方法
          // 没有调用show也是不能显示的.在这之前要使hud隐藏并且不能接受触摸事件
        // 透明度为0(小于等于0.01),相当于hidden,无法响应触摸事件
        self.alpha = 0.0f;

        rotationTransform = CGAffineTransformIdentity;

          // 设置label和detailLabel
        [self setupLabels];
          // 设置指示器
        [self updateIndicators];

    }
    return self;
}
```

至于 opaque 这个属性,着实让我纠结了好一阵子,不过暂时先不纠结那么多,以苹果官方文档为参考:

> This property provides a hint to the drawing system as to how it should treat the view. If set to YES, the drawing system treats the view as fully opaque, which allows the drawing system to optimize some drawing operations and improve performance. If set to NO, the drawing system composites the view normally with other content. The default value of this property is YES.
>
> An opaque view is expected to fill its bounds with entirely opaque content—that is, the content should have an alpha value of 1.0. If the view is opaque and either does not fill its bounds or contains wholly or partially transparent content, the results are unpredictable. You should always set the value of this property to NO if the view is fully or partially transparent.

四. 动画效果

在HUD `show` 或者 `hide` 的时候会显示的动画效果,默认的是 `MBProgressHUDAnimationFade` .

```
self.animationType = MBProgressHUDAnimationFade;
```

动画效果 `MBProgressHUDAnimation` 是一个枚举.

```
typedef NS_ENUM(NSInteger, MBProgressHUDAnimation) {
    // 默认效果,只有透明度变化的动画效果
    MBProgressHUDAnimationFade,
    // 透明度变化+形变效果,其中MBProgressHUDAnimationZoom和
    // MBProgressHUDAnimationZoomOut的枚举值都为1
    MBProgressHUDAnimationZoom,
    MBProgressHUDAnimationZoomOut = MBProgressHUDAnimationZoom,
    MBProgressHUDAnimationZoomIn
};
```

动画效果是在这两个方法中实现的:

```
// 显示HUD
- (void)showUsingAnimation:(BOOL)animated {
    // Cancel any scheduled hideDelayed: calls
    [NSObject cancelPreviousPerformRequestsWithTarget:self];
    [self setNeedsDisplay];

    // ZoomIn,ZoomOut分别理解为`拉近镜头`,`拉远镜头`
    // 因此MBProgressHUDAnimationZoomIn先把形变缩小到0.5倍,再恢复到原状,产生放大效果
```

```
    // 反之MBProgressHUDAnimationZoomOut先把形变放大到1.5倍,再恢复原状,产生缩小效果
    // 要注意的是,形变的是整个`MBProgressHUD`,而不是中间可视部分
    if (animated && animationType == MBProgressHUDAnimationZoomIn) {
    // 在初始化方法中, 已经定义了rotationTransform = CGAffineTransformIdentity.
    // CGAffineTransformIdentity也就是对view不进行变形,对view进行仿射变化总是原样

    // CGAffineTransformConcat是两个矩阵相乘,与之等价的设置方式是:
    // self.transform = CGAffineTransformScale(rotationTransform, 0.5f, 0.5f);
        self.transform = CGAffineTransformConcat(rotationTransform, CGAffineTransfor
    } else if (animated && animationType == MBProgressHUDAnimationZoomOut) {
    // self.transform = CGAffineTransformScale(rotationTransform, 1.5f, 1.5f);
        self.transform = CGAffineTransformConcat(rotationTransform, CGAffineTransfor
    }

    self.showStarted = [NSDate date];

    // 开始做动画
    if (animated) {
    // 在初始化方法或者`hideUsingAnimation:`方法中,alpha被设置为0.f,在该方法中完成0.f~1.f的动
        [UIView beginAnimations:nil context:NULL];
        [UIView setAnimationDuration:0.30];
        self.alpha = 1.0f;
    // 从形变状态回到初始状态
        if (animationType == MBProgressHUDAnimationZoomIn || animationType == MBProg
            self.transform = rotationTransform;
        }
        [UIView commitAnimations];
    }
    else {
        self.alpha = 1.0f;
    }
}

// 隐藏HUD
- (void)hideUsingAnimation:(BOOL)animated {
    // Fade out
    if (animated && showStarted) {
        [UIView beginAnimations:nil context:NULL];
        [UIView setAnimationDuration:0.30];
        [UIView setAnimationDelegate:self];
        [UIView setAnimationDidStopSelector:@selector(animationFinished:finished:con
          // 当alpha小于0.01时,就会被当做全透明对待,全透明是接收不了触摸事件的.
          // 所以设置0.02防止hud在还没结束动画并调用done方法之前传递触摸事件.
          // 在完成的回调animationFinished:finished:context:才设为0
        if (animationType == MBProgressHUDAnimationZoomIn) {
            self.transform = CGAffineTransformConcat(rotationTransform, CGAffineTran
        } else if (animationType == MBProgressHUDAnimationZoomOut) {
            self.transform = CGAffineTransformConcat(rotationTransform, CGAffineTran
```

```
        }

        self.alpha = 0.02f;
        [UIView commitAnimations];
    }
    else {
        self.alpha = 0.0f;
        [self done];
    }
    self.showStarted = nil;
}
```

接下来 `-initWithFrame:` 中又调用 `[self setupLabels]` 设置了两个 `label` 的相关初始化设置(除了 `frame` 的设置--这应该是在 `layoutSubviews` 里面做的事情).然后开始设置指示器.

```
- (void)updateIndicators {

    // 读源码的时候,类似这种局部变量直接忽略,等代码用到它,我们再"懒加载"
    BOOL isActivityIndicator = [indicator isKindOfClass:[UIActivityIndicatorView cla
    BOOL isRoundIndicator = [indicator isKindOfClass:[MBRoundProgressView class]];

    // 如果模式是MBProgressHUDModeIndeterminate,将使用系统自带的菊花系列指示器
    if (mode == MBProgressHUDModeIndeterminate) {
            // 再看回最上面的两条语句
              // 初始化的时候进来,indicator是空的,对空对象发送消息返回的布尔值是NO
            // 因为在初始化完毕后,用户可能会设置mode属性,那时还会进入这个方法,所以这两个布尔变量
        if (!isActivityIndicator) {
            // 默认第一次会进入到这里,对nil发送消息不会发生什么事
            // 为什么要removeFromSuperview呢,因为这方法并不会只进入一次
            // 不排除有些情况下先改变了mode到其他模式,之后又改回来了,这时候如果不移除
            // MBProgressHUD就会残留子控件在subviews里,虽然界面并不会显示它
            [indicator removeFromSuperview];
            // 使用系统自带的巨大白色菊花
            // 系统菊花有三种
            //typedef NS_ENUM(NSInteger, UIActivityIndicatorViewStyle) {
                    //    UIActivityIndicatorViewStyleWhiteLarge, // 大又白
                    //    UIActivityIndicatorViewStyleWhite, // 小白
                    //    UIActivityIndicatorViewStyleGray,  // 小灰
                //};
            self.indicator = MB_AUTORELEASE([[UIActivityIndicatorView alloc]
                                             initWithActivityIndicatorStyle:UIActivi
            [(UIActivityIndicatorView *)indicator startAnimating];
            [self addSubview:indicator];
        }
        // 系统菊花能设置颜色是从iOS5开始(NS_AVAILABLE_IOS(5_0)),这里用宏对手机版本进行了判断
#if __IPHONE_OS_VERSION_MIN_REQUIRED >= 50000
```

```
        [(UIActivityIndicatorView *)indicator setColor:self.activityIndicatorColor];
#endif
    }
            // 源码实现了两种自定义视图
            // 一种是MBBarProgressView(进度条),另一种是MBRoundProgressView(圆饼or圆环)
    else if (mode == MBProgressHUDModeDeterminateHorizontalBar) {
            // 进度条样式
        [indicator removeFromSuperview];
        self.indicator = MB_AUTORELEASE([[MBBarProgressView alloc] init]);
        [self addSubview:indicator];
    }
    else if (mode == MBProgressHUDModeDeterminate || mode == MBProgressHUDModeAnnula
            // 这两种mode都产生MBRoundProgressView视图,MBRoundProgressView又分两种样式
            // 如果你设置了mode为MBProgressHUDModeDeterminate,那么流程是这样子的
            // 1)alloc init先生成系统的MBProgressHUDModeIndeterminate模式->
            // 2)设置了mode为饼图,触发KVO,又进入了updateIndicators方法->
            // 3)由于isRoundIndicator是No,产生饼状图

            // 如果设置了MBProgressHUDModeAnnularDeterminate,那么步骤比它多了一步,
            // 1)alloc init先生成系统的MBProgressHUDModeIndeterminate模式->
            // 2)设置了mode为圆环,触发KVO,又进入了updateIndicators方法->
            // 3)由于isRoundIndicator是No,产生饼状图->
            // 4)设置[(MBRoundProgressView *)indicator setAnnular:YES]触发MBRoundProgr
            // KVO进行重绘视图产生圆环图
        if (!isRoundIndicator) {
            // 个人认为这个isRoundIndicator变量纯属多余
            // isRoundIndicator为Yes的情况只有从MBProgressHUDModeDeterminate换成MBProgr
            // 或者MBProgressHUDModeAnnularDeterminate换成MBProgressHUDModeDeterminate
            // 而实际上这两种切换方式产生的视图都是圆环,这是由于没有让annular设置成No
            [indicator removeFromSuperview];
            self.indicator = MB_AUTORELEASE([[MBRoundProgressView alloc] init]);
            [self addSubview:indicator];
        }
        if (mode == MBProgressHUDModeAnnularDeterminate) {
            [(MBRoundProgressView *)indicator setAnnular:YES];
        }
    }
    else if (mode == MBProgressHUDModeCustomView && customView != indicator) {
        // 自定义视图
        [indicator removeFromSuperview];
        self.indicator = customView;
        [self addSubview:indicator];
    } else if (mode == MBProgressHUDModeText) {
          // 只有文字的模式
        [indicator removeFromSuperview];
        self.indicator = nil;
    }
}
```

五. KVO

初始化时,设置完指示器就开始注册KVO和通知.

```
        .....
        [self registerForKVO];
        [self registerForNotifications];
        .....
```

具体代码实现:

```
// 注册KVO,遍历从[self observableKeypaths]返回的字符串,观察这些属性的变化
- (void)registerForKVO {
    for (NSString *keyPath in [self observableKeypaths]) {
        [self addObserver:self forKeyPath:keyPath options:NSKeyValueObservingOptionN
    }
}

- (NSArray *)observableKeypaths {
    return [NSArray arrayWithObjects:@"mode", @"customView", @"labelText", @"labelFo
            @"detailsLabelText", @"detailsLabelFont", @"detailsLabelColor", @"progre
}

// 在delloc的时候,需要将观察解除
- (void)unregisterFromKVO {
    for (NSString *keyPath in [self observableKeypaths]) {
        [self removeObserver:self forKeyPath:keyPath];
    }
}

// 触发KVO
- (void)observeValueForKeyPath:(NSString *)keyPath ofObject:(id)object change:(NSDic
    if (![NSThread isMainThread]) {
        // 当前是子线程,那么切换到主线程进行UI更新
        [self performSelectorOnMainThread:@selector(updateUIForKeypath:) withObject:
    } else {
          // 当前线程为主线程,直接更新
        [self updateUIForKeypath:keyPath];
    }
}

- (void)updateUIForKeypath:(NSString *)keyPath {
  ........
```

```
  // 以上省略一万行
          else if ([keyPath isEqualToString:@"progress"]) {
          // 除了系统指示器和自定义视图,MB给我们提供的三种形状的指示器都带有progress属性
        if ([indicator respondsToSelector:@selector(setProgress:)]) {
        // 触发该视图的KVO更新指示器视图
            [(id)indicator setValue:@(progress) forKey:@"progress"];
        }
        // 绘制交给视图内部处理
        return;
        }
    // 如果更改了label的字体,需要重新调用layoutSubviews
    [self setNeedsLayout];
    // 设置标记,在下一个周期调用drawRect:方法重绘
    [self setNeedsDisplay];
}
```

六. 布局与绘制

布局

子控件的布局计算没什么复杂的地方,为了方便理解,我画了两幅图

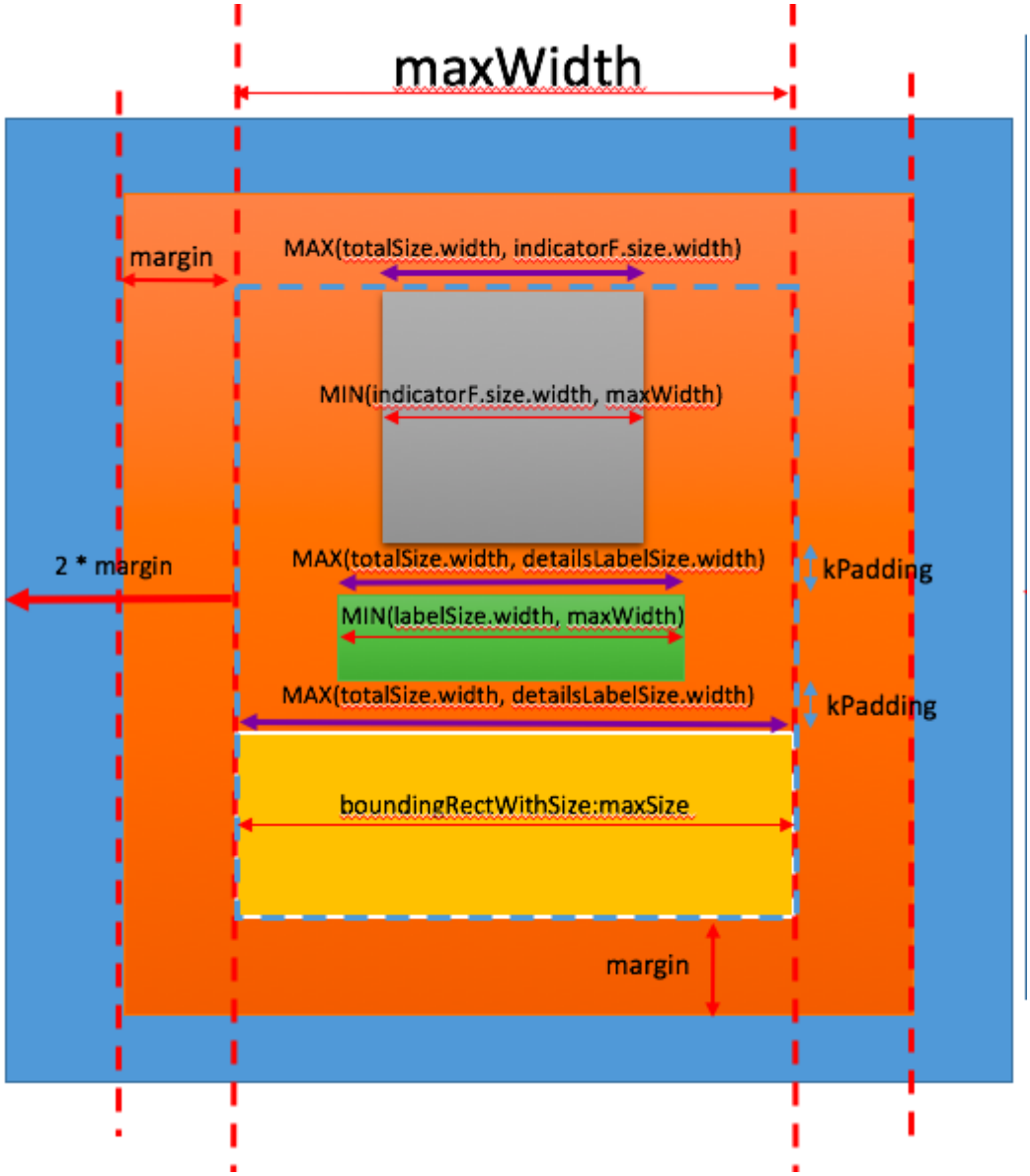
```
- (void)layoutSubviews {
    [super layoutSubviews];

    // MBProgressHUD是一个充满整个父控件的控件
    // 使得父控件的交互完全被屏蔽
    UIView *parent = self.superview;
    if (parent) {
        self.frame = parent.bounds;
    }
    CGRect bounds = self.bounds;

    .......

    // 如果用户设置了square属性,就会尽量让它显示成正方形
    if (square) {
    // totalSize为下图蓝色框框的size
        CGFloat max = MAX(totalSize.width, totalSize.height);
        if (max <= bounds.size.width - 2 * margin) {
            totalSize.width = max;
        }
        if (max <= bounds.size.height - 2 * margin) {
            totalSize.height = max;
        }
    }
    if (totalSize.width < minSize.width) {
        totalSize.width = minSize.width;
    }
    if (totalSize.height < minSize.height) {
        totalSize.height = minSize.height;
    }

    size = totalSize;
}
```



上图 蓝色虚线部分 代表子控件们能够展示的区域,其中宽度是被限制的,其中定义了 `maxWidth` 让3个子控件中的最大宽度都不得超过它.值得注意的是,源码并没设置最大高度,如果我们使用自定义的视图,高度够大就会使 蓝色虚线部分 的上下底超出屏幕范围.某种程度上来讲也是设计上的一种bug,但我认为作者肯定意识到了这点---- `label\detailLabel` 中有很多文字导致换行是很常见的情况,因此需要限制它的最大宽度,但没人会使用一个非常大的指示器,所以通过额外的计算来考虑因为这种情况超出屏幕上下边界是毫无必要的.

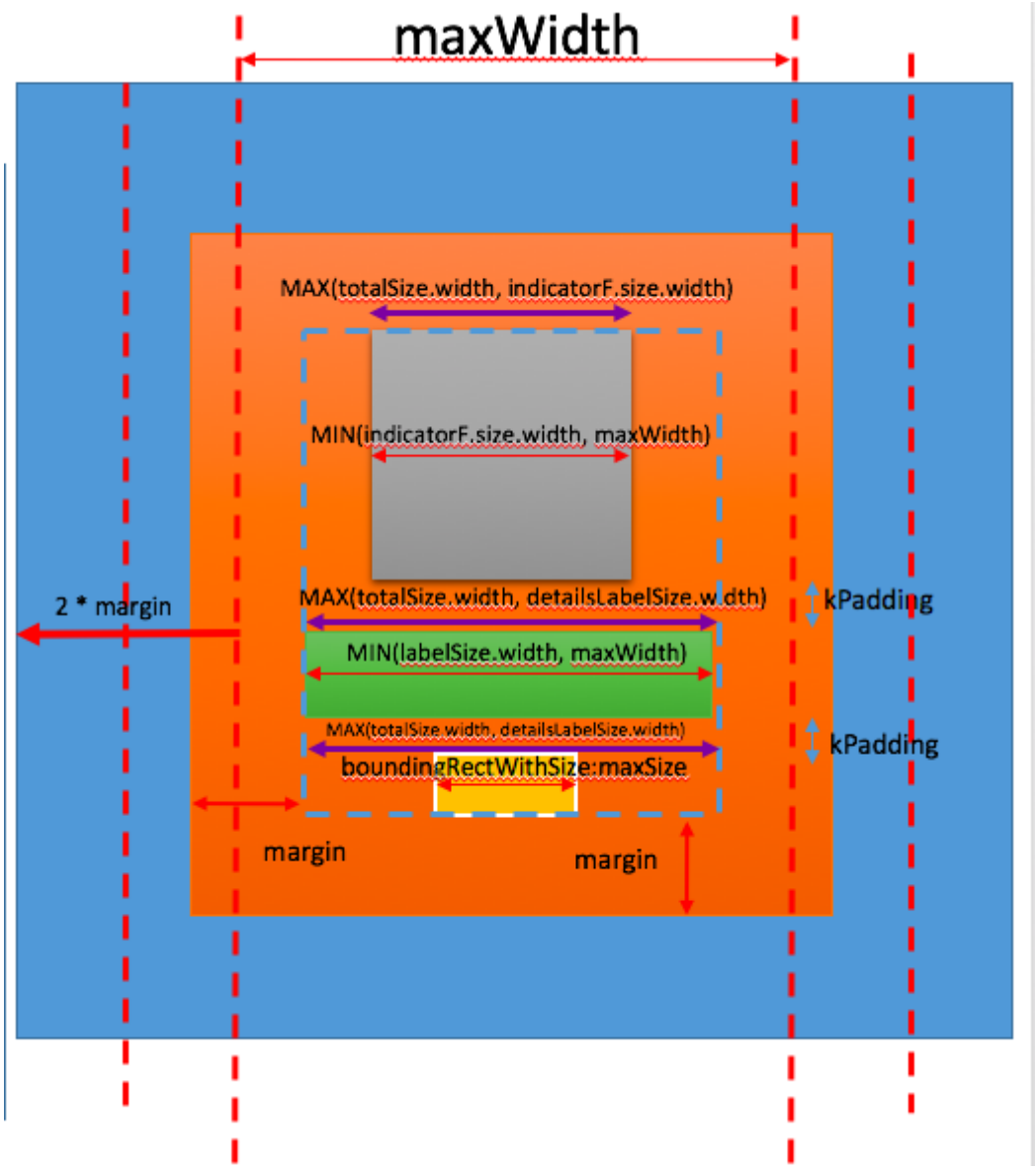
此外,绿色的 `label` 被限制为只能显示一行,黄色的 `detailLabel` 通过下面的代码来限制它不能超出屏幕上下.

```
// 计算出屏幕剩下的高度
// 其中减去了4个margin大小,保证了子空间和HUD的边距,HUD和屏幕的距离
CGFloat remainingHeight = bounds.size.height - totalSize.height - kPadding - 4 * mar

// 将文字内容限制在这个size中,超出部分省略号
CGSize maxSize = CGSizeMake(maxWidth, remainingHeight);

CCGSize detailsLabelSize = MB_MULTILINE_TEXTSIZE(detailsLabel.text, detailsLabel.fon

// 7.0开始使用boundingRectWithSize:options:attributes:context:方法计算
// 7.0以前使用sizeWithFont:constrainedToSize:lineBreakMode:计算
#if __IPHONE_OS_VERSION_MIN_REQUIRED >= 70000
#define MB_MULTILINE_TEXTSIZE(text, font, maxSize, mode) [text length] > 0 ? [text \
boundingRectWithSize:maxSize options:(NSStringDrawingUsesLineFragmentOrigin) \
attributes:@{NSFontAttributeName:font} context:nil].size : CGSizeZero;
#else
#define MB_MULTILINE_TEXTSIZE(text, font, maxSize, mode) [text length] > 0 ? [text \
sizeWithFont:font constrainedToSize:maxSize lineBreakMode:mode] : CGSizeZero;
#endif
```



简
(/)

(/collections)

(/apps)

上图是另一种没达到 `maxSize` 的情况.

绘制

下面看绘制部分,这是MBProgreeHUD中比较重要的内容.

```
- (void)drawRect:(CGRect)rect {
    // 拿到当前的绘图上下文
    CGContextRef context = UIGraphicsGetCurrentContext();
    UIGraphicsPushContext(context);

    // 默认中间的HUD外是透明的,可以看到父控件,设置了dimBackground这个属性可以让HUD周围是一个渐变
    // 这里用了一个渐变层,颜色是写死的
    if (self.dimBackground) {
        //Gradient colours
        size_t gradLocationsNum = 2;
        CGFloat gradLocations[2] = {0.0f, 1.0f};
```

```
        CGFloat gradColors[8] = {0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.75f};
        CGColorSpaceRef colorSpace = CGColorSpaceCreateDeviceRGB();
        CGGradientRef gradient = CGGradientCreateWithColorComponents(colorSpace, gra
        CGColorSpaceRelease(colorSpace);
        //Gradient center
        CGPoint gradCenter= CGPointMake(self.bounds.size.width/2, self.bounds.size.h
        //Gradient radius
        float gradRadius = MIN(self.bounds.size.width , self.bounds.size.height) ;
        //Gradient draw
        CGContextDrawRadialGradient (context, gradient, gradCenter,
                                     0, gradCenter, gradRadius,
                                     kCGGradientDrawsAfterEndLocation);
        CGGradientRelease(gradient);
    }

    // 用户有设置颜色就使用设置的颜色,没有的话默认灰色
    // 从下面代码可以看出,自定义HUD背景颜色是没有透明度的
    if (self.color) {
        CGContextSetFillColorWithColor(context, self.color.CGColor);
    } else {
        CGContextSetGrayFillColor(context, 0.0f, self.opacity);
    }

    CGRect allRect = self.bounds;
    // 画出一个圆角的HUD
    // size在layoutSubviews中被计算出来,是HUD的真实size
    CGRect boxRect = CGRectMake(round((allRect.size.width - size.width) / 2) + self.
                                round((allRect.size.height - size.height) / 2) + sel
    float radius = self.cornerRadius;
    //开始绘制路径
    CGContextBeginPath(context);
    // 起始点
    CGContextMoveToPoint(context, CGRectGetMinX(boxRect) + radius, CGRectGetMinY(box
    // 依次画出右上角、右下角,左下角,左上角的四分之一圆弧
    // 注意,虽然没有显式地调用CGContextAddLineToPoint函数
    // 但绘制圆弧时每一次的起点都会和上一次的终点连接,生成线段
    CGContextAddArc(context, CGRectGetMaxX(boxRect) - radius, CGRectGetMinY(boxRect)
    CGContextAddArc(context, CGRectGetMaxX(boxRect) - radius, CGRectGetMaxY(boxRect)
    CGContextAddArc(context, CGRectGetMinX(boxRect) + radius, CGRectGetMaxY(boxRect)
    CGContextAddArc(context, CGRectGetMinX(boxRect) + radius, CGRectGetMinY(boxRect)
    CGContextClosePath(context);
    CGContextFillPath(context);

    //
    UIGraphicsPopContext();

}
```

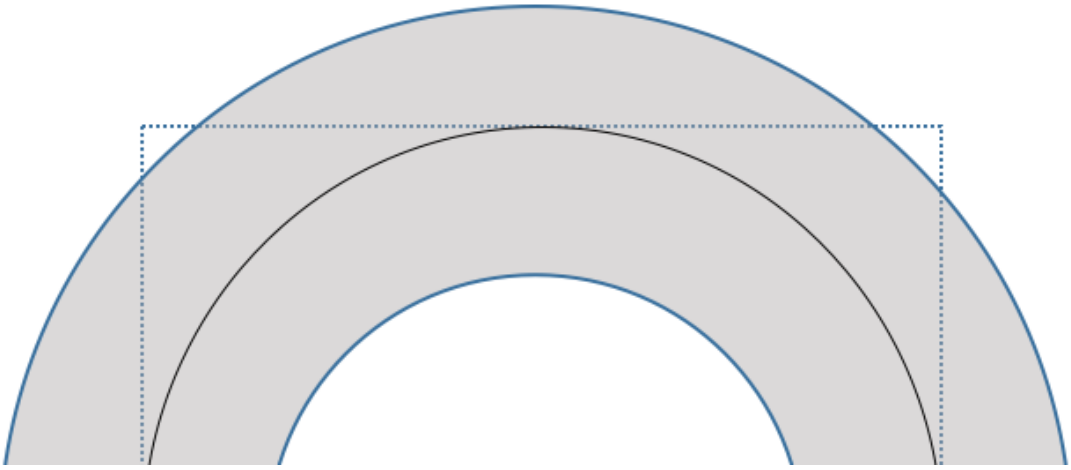
indicator的绘制

MBRoundProgressView

当我们绘制路径时,描述的路径如果宽度大于1,描边的时候是向路径宽度是以路径为中点的.

举个例子,如果从 `(0,0)` 向 `(100,0)` 画一条宽度为 `X` 的线,那么显示的宽度实际只有 `X/2` ,因为还有一半因为超出了绘图区域而没有被绘制.

为了防止绘制内容的丢失,半径 `radius` 的计算是 `(self.bounds.size.width - lineWidth)/2` ,而并不是 `self.bounds.size.width/2` .更不是 `(self.bounds.size.width -2*lineWidth)/2` ,借助下图理解:



```
// 圆环绘制
if (_annular) {
    // iOS7.0以后的圆环描边风格变了,变成了2.f
      // 7.0之前的还是5.f.主要是为了迎合扁平的风格我觉得
    BOOL isPreiOS7 = kCFCoreFoundationVersionNumber < kCFCoreFoundationVersionNu
    CGFloat lineWidth = isPreiOS7 ? 5.f : 2.f;
    ......
    CGFloat radius = (self.bounds.size.width - lineWidth)/2;
}
```

在圆饼的绘制过程中,圆饼外层的圆环是通过 `CGContextStrokeEllipseInRect(CGContextRef, CGRect)` 进行描边的,根据上面的结论,圆饼绘制区域(circleRect)和上下文提供的绘制区域(allRect)应该宽高都相差 `1.f` 就够圆饼外层的圆环的正确绘制.作者在这里用了 `2.f` ,实际上 `1.f` 就够了.

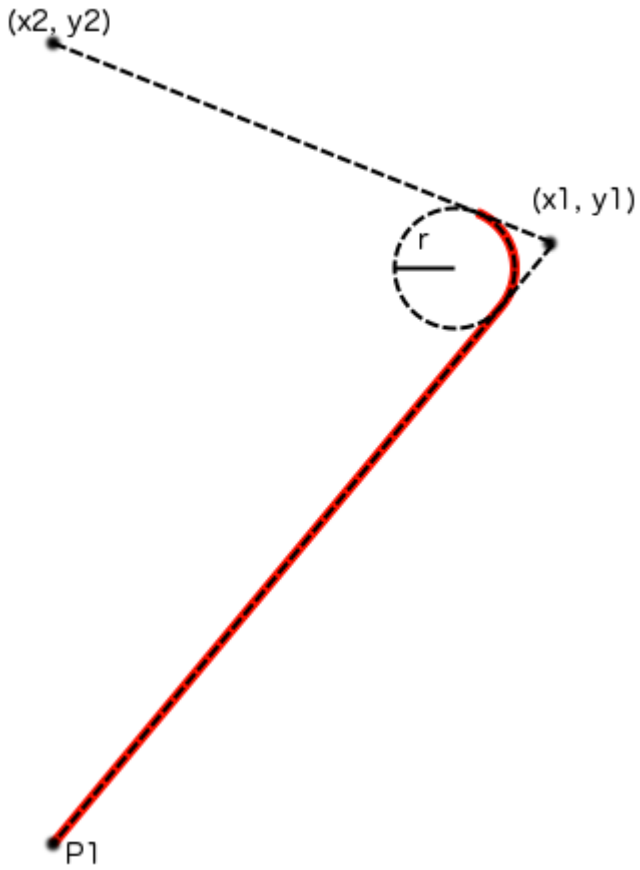
```
    CGRect allRect = self.bounds;
    CGRect circleRect = CGRectInset(allRect, 2.0f, 2.0f);
```

接下来是 MBBarProgressView 的绘制.

MBBarProgressView

MBBarProgressView与MBRoundProgressView的绘制类似,都是使用Quartz2D进行绘图.使用的都是很基础很常用的API,所以阅读难度并不大.唯一让人困惑的可能是这个 CGContextAddArcToPoint(CGContextRef c, CGFloat x1, CGFloat y1,CGFloat x2, CGFloat y2, CGFloat radius) 了,另一个画弧的函数则简单很多: CGContextAddArc(CGContextRef c, CGFloat x, CGFloat y, CGFloat radius, CGFloat startAngle, CGFloat endAngle, int clockwise) .

结合下图,我的理解方式是: P1 为绘图的当前点, x1 ，y1， x2， y2 表示了两个定点.通过当前点 P1 ,点 (x1,y1) 和 (x2,y2) ,可以表示一个确定的角度,这时一个任意半径的圆都能与图中的两条射线相切.不同半径的圆,圆心角都不同,两个切点之间的弧也不相同.举个例子,我们拿不同半径的球体去贴到两面墙的相交处,两个切点之间有段弧线,球越大弧越长,但是圆心角大小都是一样的.控制圆心角大小由这三个点决定,能够获得的最大圆心角是90度.



函数示意图

两个画弧的函数差别有点大, `CGContextAddArcToPoint` 分为两步:

1. 从当前点 `P1` 开始,沿着 `(x1,y1)` 方向画线段.
2. 线段一直画到 `圆` 与虚线相切的地方.
3. 这是圆被分成了两段弧线,绘制短的那条(即圆心对着的那段弧).

我们还可以得到其他的结论:

1. `(x2,y2)` 的作用只是为了确定与另一条射线形成的角度,只要 `(x2,y2)` 是在 `(x1,y1)` -> `(x2,y2)` 射线方向上的任意一点就可以了.
2. 当 `P1` 点刚好为切点时,画出来的仅仅是一条弧线而不是线段加弧线.
3. `CGContextAddArcToPoint` 功能比 `CGContextAddArc` 强大,后者需要起始角度和终止角度. 有些情况下,是很难算出这两个角度的.

当利用上面的结论2时,画出来的弧和使用 `CGContextAddArc` 函数画出的弧效果相当.如果三个点形成的角度为直角,那么刚好是1/4圆弧.

遗憾的是,源码并没有发挥该函数强大的一面,使用了 `CGContextAddLineToPoint` 来画蛇添足.将它们注释掉,结果并没有什么不同,读者可以继续注释后三条 `CGContextAddArcToPoint` ,可以验证该函数已经帮我们画好线段了.

```
.....
// Draw background
float radius = (rect.size.height / 2) - 2;
CGContextMoveToPoint(context, 2, rect.size.height/2);
CGContextAddArcToPoint(context, 2, 2, radius + 2, 2, radius);
//CGContextAddLineToPoint(context, rect.size.width - radius - 2, 2);
CGContextAddArcToPoint(context, rect.size.width - 2, 2, rect.size.width - 2, rec
CGContextAddArcToPoint(context, rect.size.width - 2, rect.size.height - 2, rect.
//CGContextAddLineToPoint(context, radius + 2, rect.size.height - 2);
CGContextAddArcToPoint(context, 2, rect.size.height - 2, 2, rect.size.height/2,
CGContextFillPath(context);
```

画完背景后,继续进行了描边,描边的代码和上面几乎一模一样,作者之所以这样做,是因为一个子路径的 `fill` 和 `stroke` 效果是不能同时产生的,哪个先调用,就只会出现它产生的效果.如果源码是这样写的:

```
// Draw background
.....
//CGContextAddLineToPoint(context, radius + 2, rect.size.height - 2);
CGContextAddArcToPoint(context, 2, rect.size.height - 2, 2, rect.size.height/2,
// 先调用fill,就只有填充效果,如果调换CGContextFillPath和CGContextStrokePath的调用顺序呢
// 那么就只有描边效果
CGContextFillPath(context);
CGContextStrokePath(context);
```

所以作者的做法是——又画了一个路径.

```
// Draw border
    CGContextMoveToPoint(context, 2, rect.size.height/2);
    CGContextAddArcToPoint(context, 2, 2, radius + 2, 2, radius);
    CGContextAddLineToPoint(context, rect.size.width - radius - 2, 2);
    CGContextAddArcToPoint(context, rect.size.width - 2, 2, rect.size.width - 2, rec
    CGContextAddArcToPoint(context, rect.size.width - 2, rect.size.height - 2, rect.
    CGContextAddLineToPoint(context, radius + 2, rect.size.height - 2);
    CGContextAddArcToPoint(context, 2, rect.size.height - 2, 2, rect.size.height/2,
    CGContextStrokePath(context);
```

事实上,可以使用 `CGContextDrawPath(CGContextRef c, CGPathDrawingMode mode)` 函数解决这个问题.这样就能省略很多的重复代码.

```
       // Draw background
    .....
    //CGContextAddLineToPoint(context, radius + 2, rect.size.height - 2);
    CGContextAddArcToPoint(context, 2, rect.size.height - 2, 2, rect.size.height/2,
    // 这两句被替换
     // CGContextFillPath(context);
    // CGContextStrokePath(context);
    // kCGPathFillStroke参数告诉函数进行描边和填充
    CGContextDrawPath(context, kCGPathFillStroke);
```

progress进度的更新

1.用户更新 `progress` 属性

2.由于 `progress` 被监听,触发 `KVO` ,调用 `- observeValueForKeyPath:ofObject:change:context:`

3. `observeValueForKeyPath:ofObject:change:context:` 中调用了 `setNeedsDisplay` ,标识视图为需要重新绘制.

4.调用 `drawRect:` 重绘,进度条更新

七. 显示与隐藏

显示

显示过程中,源码提供了给 `hud` "绑定"后台任务的方法.

```
- (void)showWhileExecuting:(SEL)method onTarget:(id)target withObject:(id)object ani
    methodForExecution = method;
      // 对于MRC来说,要保留target和object对象
      // ARC会自动保留这两个对象
      // 不管是ARC还是MRC,都要注意引用循环的问题,因此下面有个-cleanUp方法用来释放强引用
    targetForExecution = MB_RETAIN(target);
    objectForExecution = MB_RETAIN(object);

    self.taskInProgress = YES;
      // detachNewThreadSelector是NSThread的类方法,开启一个子线程执行任务,线程默认start
    [NSThread detachNewThreadSelector:@selector(launchExecution) toTarget:self withO
    // Show HUD view
    [self show:animated];
}

- (void)showAnimated:(BOOL)animated whileExecutingBlock:(dispatch_block_t)block onQu
     completionBlock:(MBProgressHUDCompletionBlock)completion {
    // 标记任务标识
    self.taskInProgress = YES;
    // 将block先引用起来,在隐藏完之后执行block
    self.completionBlock = completion;
    // 在队列上异步执行,更新UI在主线程进行
    dispatch_async(queue, ^(void) {
        block();
        dispatch_async(dispatch_get_main_queue(), ^(void) {
              // 方法中有隐藏HUD这一更新UI的操作
            [self cleanUp];
        });
    });
    // 在任务执行的过程中进行动画
    [self show:animated];
}

- (void)launchExecution {
    // 对于多线程操作建议把线程操作放到@autoreleasepool中
    @autoreleasepool {
      // 忽略警告的编译器指令
#pragma clang diagnostic push
#pragma clang diagnostic ignored "-Warc-performSelector-leaks"
        // 究其原因,编译期时编译器并不知道methodForExecution是什么
          // ARC的内存管理是建立在规范的命名规则之上的,不知道方法名是什么就不知道如何处理返回值
          // 如果该方法有返回值,就不知道返回值是加入了自动释放池的还是需要ARC释放的对象
          // 因此ARC不对返回值执行任何操作,如果返回值并不是加入自动释放池的对象,这时就内存泄露了
        [targetForExecution performSelector:methodForExecution withObject:objectForE
#pragma clang diagnostic pop

        [self performSelectorOnMainThread:@selector(cleanUp) withObject:nil waitUnti
    }
```

```
}

- (void)cleanUp {
    // 任务标识重置
    taskInProgress = NO;
#if !__has_feature(objc_arc)
    [targetForExecution release];
    [objectForExecution release];
#else
    targetForExecution = nil;
    objectForExecution = nil;
#endif
    [self hide:useAnimation];
}
```

taskInProgress 的意思要结合 graceTime 来看. graceTime 是为了防止hud只显示很短时间(一闪而过)的情况,给用户设定的一个属性,如果任务在 graceTime 内完成,将不会 show hud. 所以 graceTime 这个属性离开了赋给hud的任务就没意义了.因此, taskInProgress 用来标识是否带有执行的任务.

```
- (void)handleGraceTimer:(NSTimer *)theTimer {
    // 如果没有任务,设置了graceTime也没有意义
    if (taskInProgress) {
        [self showUsingAnimation:useAnimation];
    }
}
```

值得注意的是,通过 showWhileExecuting:onTarget:withObject:animated: 等方法时,会自动将 taskInProgress 置为 yes ,其他情况(任务所在的线程不是由hud内部所创建的)需手动设置这个属性.

```
- (void)show:(BOOL)animated {
    ......
      // 进行self.graceTime的延时之后,才调用handleGraceTimer:显示hud
      // 如果没到时间就执行完了,那么完成任务调用的done方法会把taskInProgress设为NO,那么就不会显
    if (self.graceTime > 0.0) {
        NSTimer *newGraceTimer = [NSTimer timerWithTimeInterval:self.graceTime targe
        [[NSRunLoop currentRunLoop] addTimer:newGraceTimer forMode:NSRunLoopCommonMo
        self.graceTimer = newGraceTimer;
    }
   ......
}
```

隐藏

```
- (void)hide:(BOOL)animated afterDelay:(NSTimeInterval)delay {
    [self performSelector:@selector(hideDelayed:) withObject:[NSNumber numberWithBoo
}

- (void)hideDelayed:(NSNumber *)animated {
    [self hide:[animated boolValue]];
}

- (void)hide:(BOOL)animated {
    NSAssert([NSThread isMainThread], @"MBProgressHUD needs to be accessed on the ma
    useAnimation = animated;
    // 设置一个最短的显示时间
      // showStarted在显示的时候被设置了,用当前的时间算出距离showStarted过了多少时间
      // 得出interv.如果没有达到minShowTimer所要求的时间,就开启定时器等待到指定的最短时间
    if (self.minShowTime > 0.0 && showStarted) {
        NSTimeInterval interv = [[NSDate date] timeIntervalSinceDate:showStarted];
        if (interv < self.minShowTime) {
            self.minShowTimer = [NSTimer scheduledTimerWithTimeInterval:(self.minSho
                                                                selector:@selector(ha
            return;
        }
    }
    // ... otherwise hide the HUD immediately
    [self hideUsingAnimation:useAnimation];
}
```

八. 用法

用法示例代码来自该源码的github上.

```
// 使用MBProgressHUD最重要的准则是当要执行一个耗时任务时,不能放在主线程上影响UI的刷新
// 正确地使用方式是在主线程上创建MBProgressHUD,然后在子线程上执行耗时操作,执行完再在主线程上刷新UI
[MBProgressHUD showHUDAddedTo:self.view animated:YES];
dispatch_async(dispatch_get_global_queue( DISPATCH_QUEUE_PRIORITY_LOW, 0), ^{
    // Do something...
    dispatch_async(dispatch_get_main_queue(), ^{
        [MBProgressHUD hideHUDForView:self.view animated:YES];
    });
});
```

如果你想要对 `MBProgressHUD` 进行额外的配置,需要将 `showHUDAddedTo:animated:` 的返回的实例进行设置.

```
// 通过这个类方法生成的hud是加在传进去的view上的
MBProgressHUD *hud = [MBProgressHUD showHUDAddedTo:self.view animated:YES];
hud.mode = MBProgressHUDModeAnnularDeterminate;
hud.labelText = @"Loading";
[self doSomethingInBackgroundWithProgressCallback:^(float progress) {
    hud.progress = progress;
} completionCallback:^{
    [hud hide:YES];
}];
```

UI的更新应当总是在主线程上完成的,一些 `MBProgressHUD` 上的属性的setter方法考虑到了线程安全,可以被后台线程安全地调用.这些setter包括setMode:, setCustomView:, setLabelText:, setLabelFont:, setDetailsLabelText:, setDetailsLabelFont: 和 setProgress:.

如果你需要在主线程上执行一个耗时的操作,你需要在执行前稍微延时一下,以使得在阻塞主线程之前,UIKit有足够的时间去更新UI(即绘制HUD).

```
[MBProgressHUD showHUDAddedTo:self.view animated:YES];
// 如果上面那句话之后就要在主线程执行一个长时间操作,那么要先延时一下让HUD先画好
// 不然在执行任务前没画出来就显示不出来了
dispatch_time_t popTime = dispatch_time(DISPATCH_TIME_NOW, 0.01 * NSEC_PER_SEC);
dispatch_after(popTime, dispatch_get_main_queue(), ^(void){
    // Do something...
    [MBProgressHUD hideHUDForView:self.view animated:YES];
});
```

推荐拓展阅读 (/sign_in)

如果觉得我的文章对您有用，请随意打赏。您的支持将鼓励我继续创作！

¥ 打赏支持

♡ 喜欢 | 86

分享到微博 分享到微信
更多分享

8条评论 （ 按时间正序 · 按时间倒序 · 按喜欢排序 ） 添加新评论 (/sign_in)

哟_Json (/users/22ee0fa80171)
2楼 · 2015.09.15 18:48 (/p/485b8d75ccd4/comments/607421#comment-607421)
(/users/22ee0fa80171)
好长，慢慢看⊙﹏⊙b

♡ 喜欢(0) 回复

gscc (/users/09d0f0a212f9)
3楼 · 2015.09.21 09:34 (/p/485b8d75ccd4/comments/634492#comment-634492)
(/users/09d0f0a212f9)
文章写得不错，不过希望在上传项目钱，先跑一下，那个重构的demo，明显报错了。

♡ 喜欢(0) 回复

> __微凉 (/users/8179cea7ee37)： @gscc (/users/09d0f0a212f9) 非常抱歉,已经修正了
> 2015.09.21 09:57 (/p/485b8d75ccd4/comments/634573#comment-634573) 回复
>
> gscc (/users/09d0f0a212f9)： @__微凉 (/users/8179cea7ee37) 加油，多出好文章
> 2015.09.21 11:43 (/p/485b8d75ccd4/comments/634913#comment-634913) 回复
>
> 添加新回复

巩红霞 (/users/f92563c2e52b)

4 楼 · 2016.02.01 17:08 (/p/485b8d75ccd4/comments/1369892#comment-1369892)

(/users/f92563c2e52b)

线上运行过程中，发现在iphone6 plus,8.1系统上，会崩溃到registerForKVO方法

♡ 喜欢(0) 回复

偶系随便先生 (/users/3e8c7d054640)

5 楼 · 2016.03.28 18:24 (/p/485b8d75ccd4/comments/1858226#comment-1858226)

(/users/3e8c7d054640)

留着...慢慢看 谢

♡ 喜欢(0) 回复

蓝新 (/users/23bd8d4cc8bf)

6 楼 · 2016.10.09 21:21 (/p/485b8d75ccd4/comments/4704372#comment-4704372)

(/users/23bd8d4cc8bf)

关于opaque 这个属性，https://objccn.io/issue-3-1/ (https://objccn.io/issue-3-1/)这篇文章有原理解释。我们都知道有view图层树的概念. 屏幕上每个像素点的颜色都用一个计算公式，R = S + D * (1 – Sa)，源色彩(顶端纹理)+目标颜色(低一层的纹理)*(1-源颜色的透明度)。 但是，当源纹理是完全不透明的时候，目标像素就等于源纹理，计算公式就变成 R = S. 这可以省下 GPU 很大的计算量。所以CALayer有个属性叫 opaque , 由开发者告诉 GPU 纹理上的像素是透明还是不透明的。

♡ 喜欢(0) 回复

懵然中寻找光芒 (/users/08e77db408f7)

7 楼 · 2016.10.26 12:01 (/p/485b8d75ccd4/comments/5115809#comment-5115809)

(/users/08e77db408f7)

关于MBProgressHUD 怎么样只用来显示，不阻挡点击效果

♡ 喜欢(0) 回复

加载更多 ⬇ (/notes/1740800/comments?max_id=5115809&order=asc&page=2)

登录后发表评论 (/sign_in)

被以下专题收入，发现更多相似内容：

iOS 大神班 (/collection/5aac963ca52d)

收集进阶知识、编程干货 ，每天会更新四篇。对于接收的文章，伯乐在线（@iOS大

(/collection/5aac963ca52d)

988篇文章 (/collection/5aac963ca52d) · 3237人关注

iOS开发 (/collection/152889fde467)

680篇文章 (/collection/152889fde467) · 1552人关注

添加关注 (/sign_in)

(/collection/152889fde467)

移动开发 (/collection/908d710faefe)

分享移动开发技术 关注移动开发趋势 Happy Coding Happy Life

1278篇文章 (/collection/908d710faefe) · 374人关注

添加关注 (/sign_in)

(/collection/908d710faefe)