

! <https://zhuanlan.zhihu.com/p/643960658>

! <https://zhuanlan.zhihu.com/p/643960643>

PS图层混合模式超详细解答-图层混合模式的原理



前言

\$\color{red} \{本教程非常详细, 请用心看完\}\$
\$\color{red} \{本教程如果有任何问题, 欢迎评论区留言讨论\}\$
\$\color{red} \{本教程为了避免冗余, 一些不必要的截图就省略了\}\$
\$\color{blue} \{本教程只讨论8bit的情形下的混合\}\$

\$\color{blue} \{未经许可, 不可转载\}\$

饮茶

在一切开始之前, 我们先泡一杯茶

泡一杯茶需要一杯开水, 一袋茶叶, 如果太苦我们还需要一些水来兑一下

好的我们来温习一下泡茶的过程

- 1: 我们准备一壶开水
- 2: 我们准备一包茶叶, 放多少取决于个人口味
- 3: 我们把茶叶放到开水杯子中, 在这个过程中可以静置或者搅拌也可以煮一下, 等待茶叶和热水充分融合
- 4: 我们想喝茶就把茶从茶壶中倒到一个杯子中, 如果感觉苦了就加一点开水
- 5: 最后我们会得到一杯符合我们口味的茶

一般我们泡茶就是这几个步骤, 你会泡茶吗, 如果你会, 那么恭喜你, 你已经掌握了图层混合模式。

- 一壶开水相当于基础图层
- 一包茶叶相当于混合图层, 它将和一壶开水组成一个新的饮品-茶
- 茶叶放了多少就是填充, 可以是一包, 半包, 也可以不放, 放的越多, 得到的那壶茶就越浓郁
- 静置或者搅拌也可以煮一下就是混合模式, 这里的方式多种多样
- 喝茶的时候倒到杯子里面, 茶水会和开水混合, 茶水占据的比例就是不透明度
- 一杯符合我们口味的茶就是结果图层

图层混合模式在PS中一共有\$27\$种，对分组来说还有一种叫做穿透，另外在其他一些菜单中还有一些不常见的，比如笔刷菜单中的背后和清除，以及计算工具中的相加和相减。

由此，我们便可以得出，PS中一共有 $27+1+2+2=32$ 种图层混合模式。

接下来，我们将会对每一种模式进行详细解读。

什么是图层混合模式

图层混合模式究竟是什么，他的本质代表什么？

图层混合模式的本质就是对像素的运算，就像是泡茶的过程中的搅拌

比如 $x+y=z$ 或 $1+1=2$

一种图层混合模式，就代表一种运算方式，这种运算方式会把像素点 1 和像素点 2 融合成为像素点 3 。

在本教程中，对公式本身进行讨论，一些所谓的衍生概念我们尽量不提，因为会造成冗余，比如增色减色等衍生概念，我们只对本质问题讨论。如果某些过程在公式当中已经包含了，我们就不再截图了，因为不想冗余，比如图层混合模式之间的相互转换。

前置概念

像素点

像素点是PS可以处理的最小单元，为了方便我们使用\$Pix\$或者\$C\$来表示

图层

像素点可以组合形成图层也就是由点到面，图层我们使用\$Layer\$来表示

那么一个 $m \times n$ 的图层和像素的关系可以表示为

$$\text{\$\$Layer} = \left\{ \begin{aligned} & \text{\&Pix}_{(1,1)} \& \dots \& \text{\&Pix}_{(1,n)} \& \dots \& \text{\&Pix}_{(i,1)} \\ & \dots \& \dots \& \dots \& \dots \& \dots \end{aligned} \right\}$$

图层混合模式一共涉及三个图层分别是

1、基础图层或者叫做底图

就是下方的图层

我们使用英文单词below的开头字母B来表示，符号是\$Layer_B\$

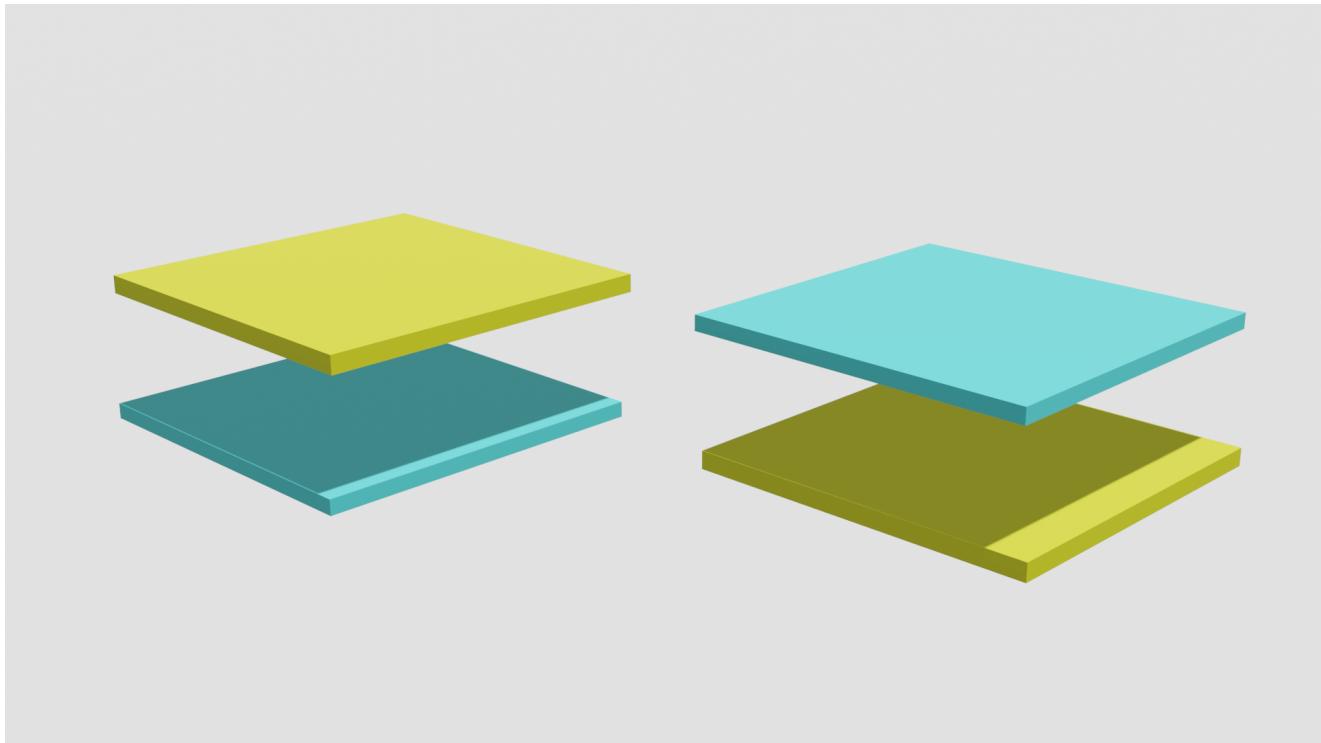
2、混合图层或者叫做调整图层或绘画图层

混合图层或者叫绘画图层，就是上方的图层

我们使用英文单词above的开头字母A来表示，符号是\$Layer_A\$

3、结果图层

就是以何种方式处理之后的结果，他是通过\$Layer_B\$和\$Layer_A\$的结合来表示。



图层大小

PS新建的过程中，我们可以看到一些图片属性，其中就有图片大小

这里我们讨论的图层大小都默认是\$m \times n\$，也就是高度是\$m\$宽度是\$n\$的图层大小

色彩空间

在图层混合模式中，涉及到的色彩空间一共有两种，第一是RGB，也就是红绿蓝三通道，第二是HSY，也就是色相，饱和度，明度。

如果是RGB空间则使用\$Pix\$来表示像素点，如果是HSY空间则使用\$C\$来表示像素点。

RGB

如果使用RGB来表示一个像素点那么像素点可以表示为

$\text{Pixel} = (\text{Red Channel}, \text{Green Channel}, \text{Blue Channel})$

为了便于和后面的简称区分开，我们使用颜色Red、Green、Blue和通道的英文字母channel的开头字母来表示各个通道。于是上面的公式也可以表示为

$\text{Pixel} = (R, G, B)$

在8bit的图像中，使用8个字节来表示一个像素，于是可以表示 $2^8 = 256$ 个数字，也就是 $[0, 255]$ 这个区间内的数字。如果数字越大那么这个通道就越亮。

几个典型的颜色

红色\$(255,0,0)\$, 黑色\$(0,0,0)\$, 白色\$(255,255,255)\$, 中性灰色\$(128,128,128)\$。

如果我们对每个通道都用归一化操作，

那我们可以得到 $\frac{R}{255} \rightarrow r \in [0, 1]$

那么以上的四种颜色就可以表示为

红色\$(1,0,0)\$, 黑色\$(0,0,0)\$, 白色\$(1,1,1)\$, 中性灰色\$(0.5,0.5,0.5)\$。

\$\$\text{Pix} = (\text{RC}, \text{GC}, \text{BC}) = (\text{rc}, \text{gc}, \text{bc})\$\$

一些运算规则

如果一个像素需要扩大或者缩小\$x\$倍

通道级别

通道数值乘以\$x\$

\$\$\text{RC} \times x

像素级别

如果一个像素点\$\text{Pix} \times x\$则表示其中所有通道数值都乘\$x\$

也就是说

\$\$\text{Pix} \times x = (\text{RC} \times x, \text{GC} \times x, \text{BC} \times x)\$\$

图层级别

如果是图层\$\text{Layer} \times x\$则表示所有像素点都乘\$x\$

也就是说

\$\$\begin{aligned} & \text{Layer} \times x \\ &= \left(\begin{aligned} & \text{Pix}_{(1,1)} \times x, \dots, \text{Pix}_{(1,n)} \times x, \dots, \text{Pix}_{(m,1)} \times x, \dots, \text{Pix}_{(m,n)} \times x \end{aligned} \right) \end{aligned}

也就是说

$$\begin{aligned} & \text{Layer} \times x \\ &= \left(\begin{aligned} & \text{rc}_{(1,1)} \times x, \dots, \text{rc}_{(1,n)} \times x, \dots, \text{rc}_{(m,1)} \times x, \dots, \text{rc}_{(m,n)} \times x \\ & \text{gc}_{(1,1)} \times x, \dots, \text{gc}_{(1,n)} \times x, \dots, \text{gc}_{(m,1)} \times x, \dots, \text{gc}_{(m,n)} \times x \\ & \text{bc}_{(1,1)} \times x, \dots, \text{bc}_{(1,n)} \times x, \dots, \text{bc}_{(m,1)} \times x, \dots, \text{bc}_{(m,n)} \times x \end{aligned} \right) \end{aligned}$$

辅助计算程序

为了辅助我们计算并且验证这些理论公式和PS中实际运行的结果是否一致，我们借助java代码来实现具体的计算过程，手动计算可以，但是浪费时间，我们有更简单的方式。

我们定义一个java类BlendColor,这个类用RGB作为基础，每次计算都以RGB存储，如果需要其他表达式，则我们通过RGB转换。

```
public class BlendColor {  
  
    public ColorItem red;  
    public ColorItem green;  
    public ColorItem blue;  
    public List<ColorItem> orderColorList;
```

```

public BlendColor(double redv, double greenv, double bluev) {
    ColorItem red = new ColorItem("red", redv);
    ColorItem green = new ColorItem("green", greenv);
    ColorItem blue = new ColorItem("blue", bluev);
    this.orderColorList = new ArrayList<ColorItem>();
    this.red = red;
    this.green = green;
    this.blue = blue;
    this.orderColorList.add(red);
    this.orderColorList.add(green);
    this.orderColorList.add(blue);

    // 排序
    Collections.sort(this.orderColorList);
}

...
}

```

HSY

如果使用HSY来表示一个像素点

$\text{C} = (\text{Hue}, \text{Stratuation}, \text{Luminosity}) = (H, S, Y)$

为了方便统一表述，在涉及到HSY时，如果我们需要表达RGB数值，我们使用 $\text{C} = (C_{\text{red}}, C_{\text{green}}, C_{\text{blue}}) \text{ iff } (RC, GC, BC)$

Hue也就是色相，取值范围是 $[0^{\circ}, 360^{\circ}]$ ，决定是什么颜色

Stratuation饱和度，取值范围是 $[0, 100]$ ，决定这种颜色鲜艳程度或者换一种说法-颜色有多少

Luminosity明度，取值范围是 $[0, 100]$ ，决定这种颜色有多亮

上述两种色彩空间的转换关系如下：

$\text{Hue} = \left\{ \begin{aligned} & 0 & \text{若 } \max = \min \& 60^{\circ} \times \frac{gc - bc}{max - min} & \text{若 } \\ & \max \geq bc \& 60^{\circ} \times \frac{gc - bc}{max - min} + 360^{\circ} & \text{若 } \\ & gc < bc \& 60^{\circ} \times \frac{bc - rc}{max - min} + 120^{\circ} & \text{若 } \\ & gc = max \& 60^{\circ} \times \frac{rc - gc}{max - min} + 240^{\circ} & \text{若 } \\ & bc = max \end{aligned} \right. \right.$

$\text{Stratuation} = \frac{\max - \min}{100}$

$\text{Luminosity} = 0.3rc + 0.59gc + 0.11bc$

代码实现如下

```

public double getHue() {
    ColorItem max = this.getMax();
    ColorItem mid = this.getMid();
    ColorItem min = this.getMin();
    if (Double.doubleToLongBits(max.value) ==
        Double.doubleToLongBits(min.value)) {

```

```

        return 0;
    }
    double hueAbs = (mid.value - min.value) / (max.value - min.value);
    if (max.name == "red") {
        if (this.green.value > this.blue.value) {
            return 60 *hueAbs;
        } else {
            return -60* hueAbs + 360;
        }
    }
    if (max.name == "green") {
        return 60 *hueAbs + 120;
    }
    if (max.name == "blue") {
        return 60* hueAbs + 240;
    }
    return 0;
}

public double getLum() {
    double lum = 0.3 * this.red.value + 0.59 * this.green.value + 0.11 *
this.blue.value;
    return lum;
}

public double getSat() {
    double max = this.getMax().value;
    double min = this.getMin().value;
    return max - min;
}

```

不透明度和填充

不透明度的英文单词是\$Opacity\$, 填充的英文单词是\$fill\$

两者在图层混合模式中有时相同有时不同。具体我们会在后面说明。

简单描述二者的区别，就是，不透明度就是将基础图层B和基础图层A，使用Opacity比例进行混合，公式如下：

$$\$\$Opacity(b,a)= op\backslash times b + (1-op)\backslash times BlendMode(b,a) \$\$$$

其中\$op\$代表不透明度

代码实现

```

public static BlendColor Opacity(BlendColor colorBase, BlendColor
colorResult, double opacity) {
    double redOpacity = colorResult.red.value *opacity +
colorBase.red.value* (1 - opacity);
    double greenOpacity = colorResult.green.value *opacity +
colorBase.green.value* (1 - opacity);
    double blueOpacity = colorResult.blue.value *opacity +

```

```

colorBase.blue.value* (1 - opacity);
    return new BlendColor(redOpacity, greenOpacity, blueOpacity);
}

```

如果是\$Fill\$则

$\$Fill(b,a) = BlendMode(b,a \times fill)$

这只是一个原则公式，具体情况要视具体模式确定，每一种计算方式都有细微区别。我们在此处不给出具体的公式和代码，但是我们在每一种中，我们会具体实现。

填充的计算方式有八个和其他的不同，并且此时和不透明度的计算方法不一样。除了这八个，不透明度和填充变化的结果是相同的。

他们分别是

- 线性加深
- 颜色加深
- 线性减淡
- 颜色减淡
- 线性光（线性减淡+线性加深）
- 亮光（颜色减淡+颜色加深）
- 实色混合（线性减淡+线性加深 或者 颜色减淡+颜色加深）
- 差值

并且，填充的优先级高于不透明度，如果让两者一起产生效果则公式为：

$\$Opacity(b,a) = op \times b + (1-op) \times Fill(b,a)$

如果使用一句话概括这两者的区别就是：

填充会让当前混合模式的效果弱化，弱化的程度取决于\$fill\$的大小。

不透明度决定了当前混合模式结果图层和基础图层混合的比例，*opacity*越大，结果图层占比越大。

结合前面的例子，填充越大放的茶叶就越多，不透明度越大，最后添加的水就越少。

混合模式的组合

一般来说，混合模式的组合会以基础图层B或者混合图层A的中性灰分割线为界，组合两种不同的混合模式。

例如，强光模式，它是由正片叠底和滤色两种模式以混合图层的中性灰分割线为界完成组合

正片叠底 $\$r= Multiply(b,a)=b \times a$

滤色 $\$r= Screen(b,a)=1-(1-b)(1-a)$

强光 $\$r=\begin{aligned} & r=HardLight(b,a) \\ & \&= \left\{ \begin{aligned} & \text{Multiply}(b,2a) & a \leq 0.5 \\ & \&= \left\{ \begin{aligned} & \text{Screen}(b,2(a-0.5)) & a > 0.5 \\ & \end{aligned} \right. \\ & \end{aligned} \right. \\ & \end{aligned}$

$r=HardLight(b,a) \&= \left\{ \begin{aligned} & \text{Multiply}(b,2a) & a \leq 0.5 \\ & \&= \left\{ \begin{aligned} & \text{Screen}(b,2(a-0.5)) & a > 0.5 \\ & \end{aligned} \right. \\ & \end{aligned} \right.$

$0.5 \leq a \leq 1 \Rightarrow r=2ba+a \leq 0.5 \& 1-2(1-b)(1-a) & a > 0.5$

$0.5 \leq a \leq 1 \Rightarrow r=2ba+a \leq 0.5 \& 1-2(1-b)(1-a) & a > 0.5$

$\end{aligned}$

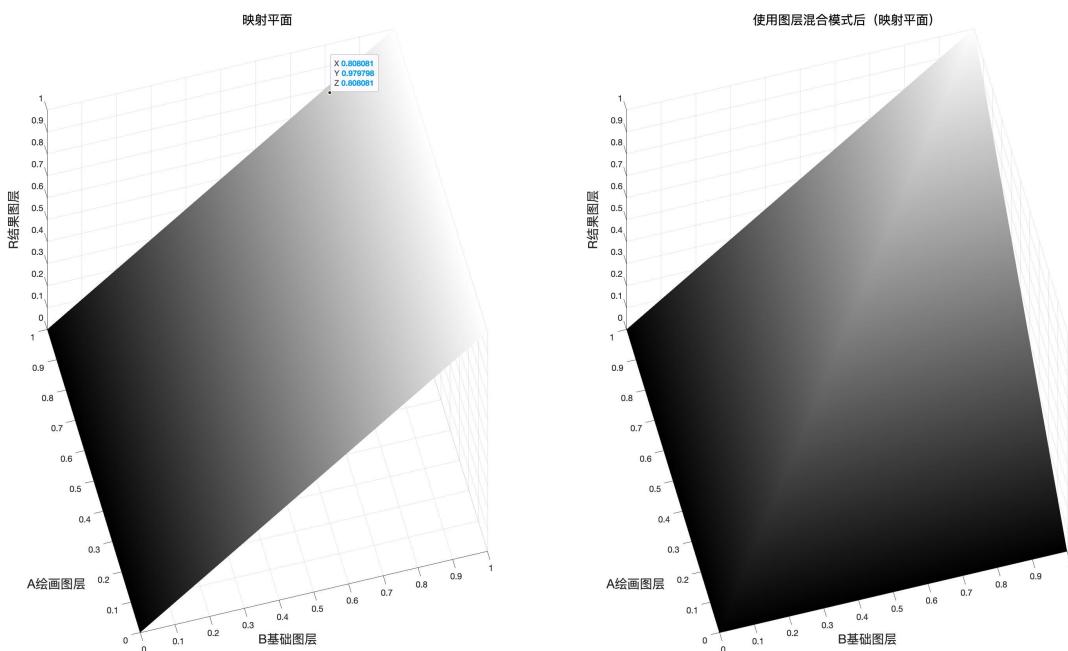
映射面

对于任意的一个混合模式，如果我们将它产生的所有基础图层和混合图层的结果，都在一个三维坐标中标记出来，则我们就会得到一个由 $256 \times 256 = 65536$ 个点组成的平面，换句话说，不管什么混合模式，基础图层和混合图层是什么值，在8bit的模式下，都只有65536种可能结果。

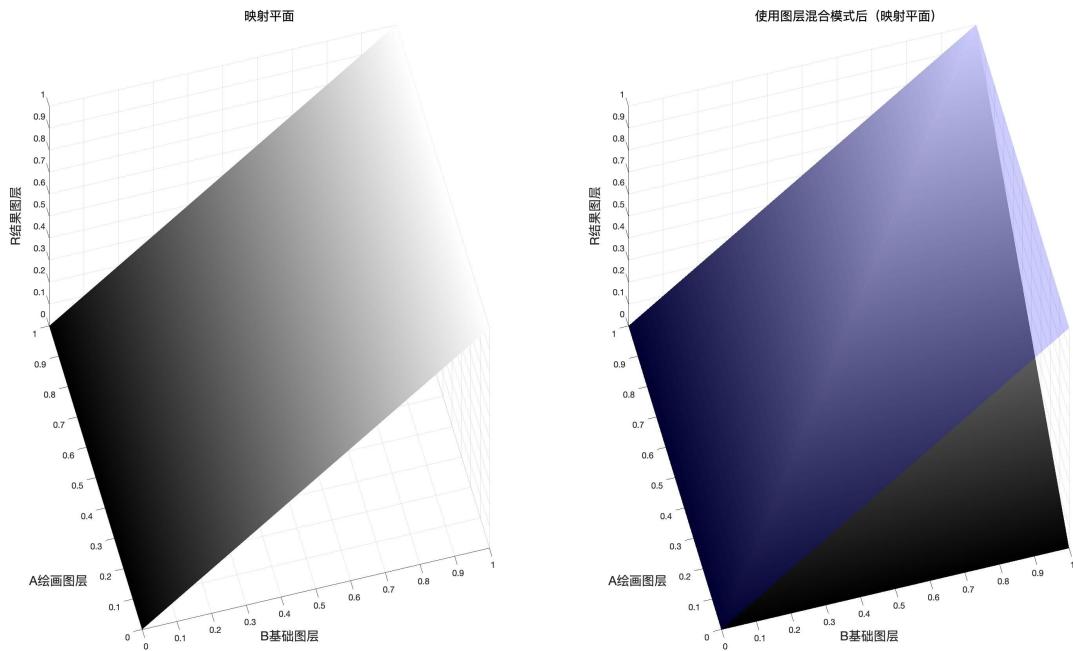
这个面就是映射面。

下图是我们将每一种图层混合模式表达式输入Matlab之后为我们呈现的结果， x 轴为基础图层， y 轴为混合图层， z 轴为结果图层，我们可以看到越大的带你则越白，**1最白**，越小的点越黑，**0最黑**。于是我们就可以通过黑白程度来判断结果大小。

映射面示例



映射面和原图面的对比



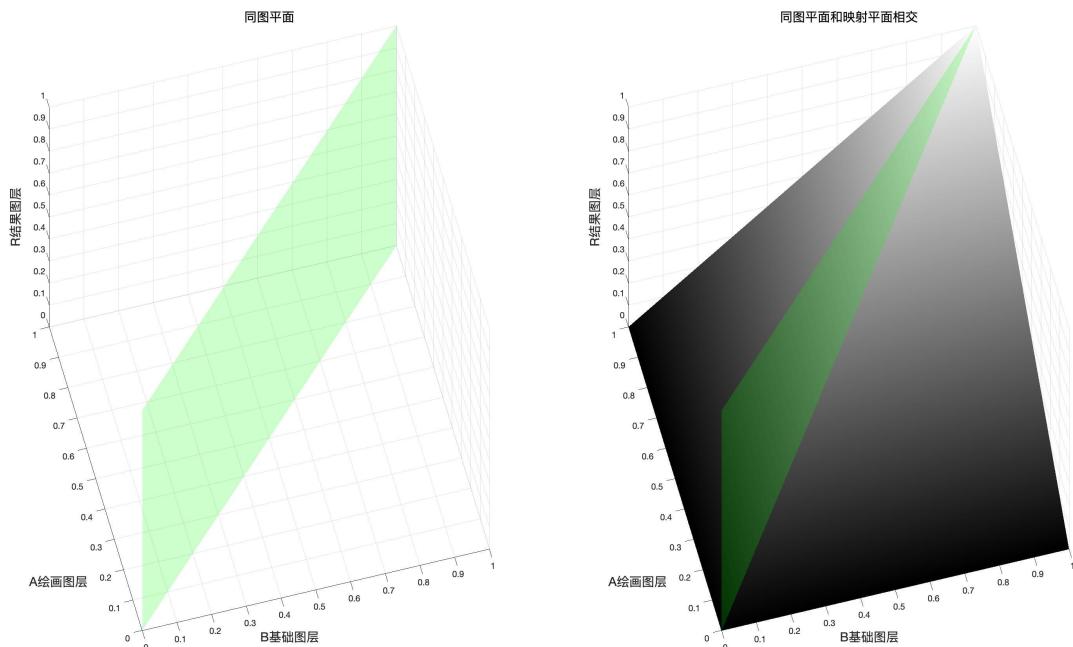
使用混合模式后的映射面和原图映射面的对比

由上图可以得知，这是一种变暗模式并且在混合图层和基础图层中选择了比较小的作为结果，因为可以明显看到一部分平面在原平面（图中的蓝色平面）的下方

同图平面

如果我们把基础图层和混合图层取值相同的点都标记出来，我们会得到一个对角平面，这个平面就是我们所说的同图平面

同图平面以及其和映射面相交的图像



图中绿色的平面就是同图平面，他和映射平面的交线就是我们可以使用曲线工具模拟的曲线。

同图混合曲线

此时相当于 $a=b$ 带入到某个图层混合模式的表达式中，

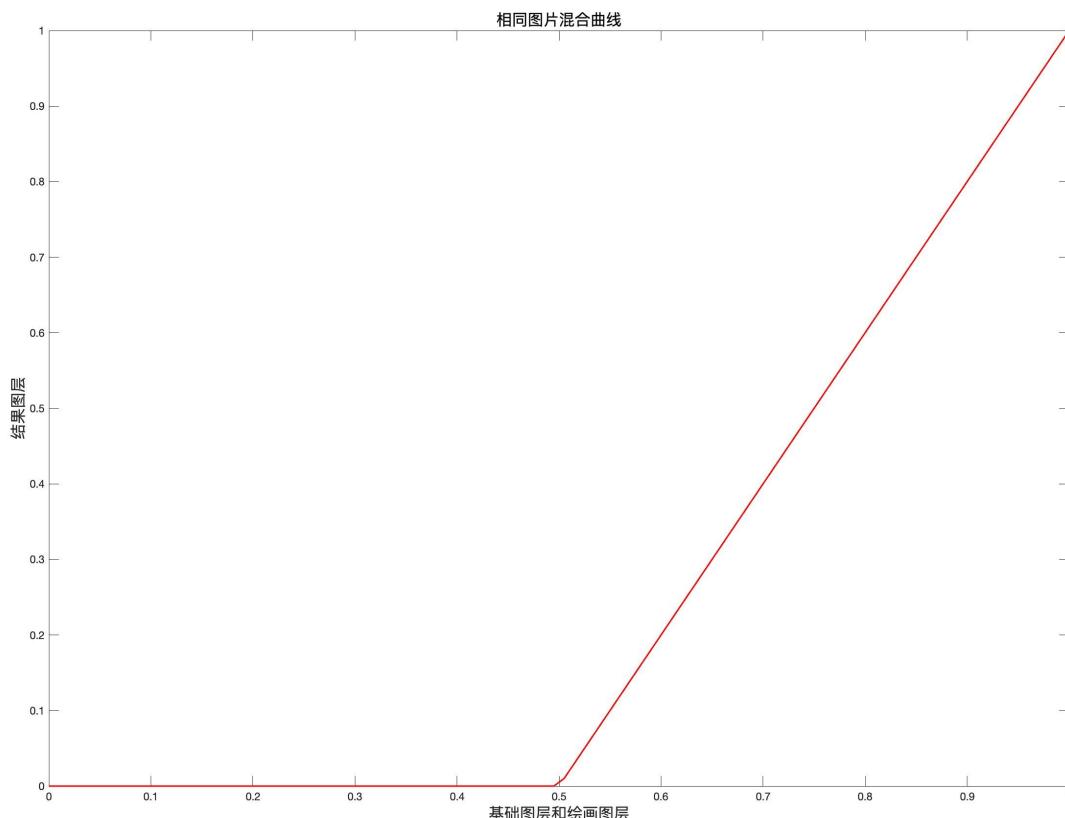
我们此处以线性加深为例

线性加深原本的表达式是

$\$r=LinearBurn(b,a)=b-(1-a)=b+a-1\$$

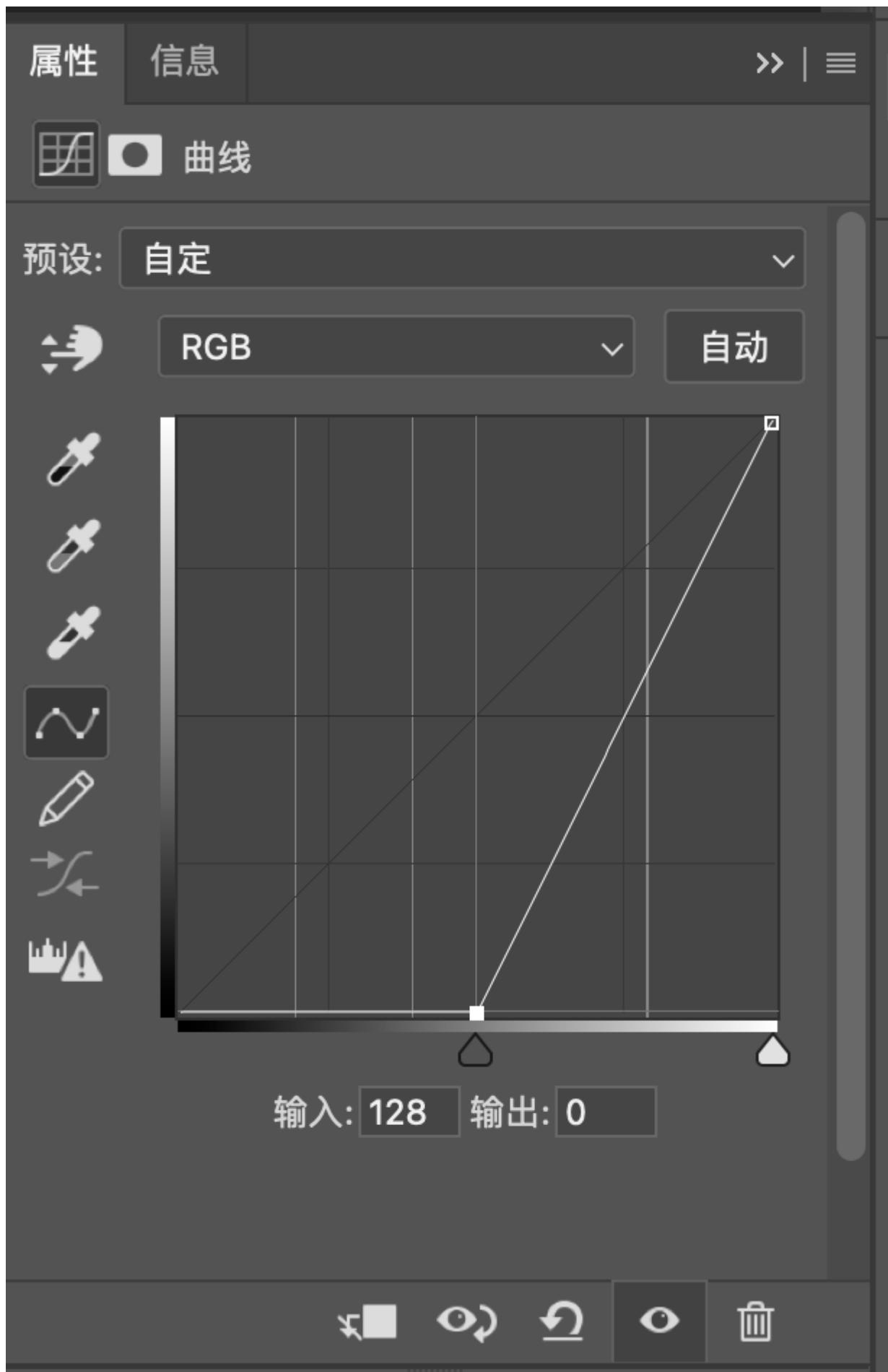
若 $b=a$ 则表达式可以写成

```
 $$\begin{aligned} r &= (b,b) \\ &= b - (1-b) \\ &= b + b - 1 \\ &= 2b - 1 \end{aligned}$$
```



此时，在坐标系中画出该图像就如同上图所示。

这时候我们只要打开曲线工具，然后拉一条曲线，和上图相同，就可以得到和这种混合模式在基础图层和混合图层相同的情况下得到结果图层相同的效果。



同图平面和映射面的交线就是可以使用曲线工具模拟这种混合模式时画出的曲线。

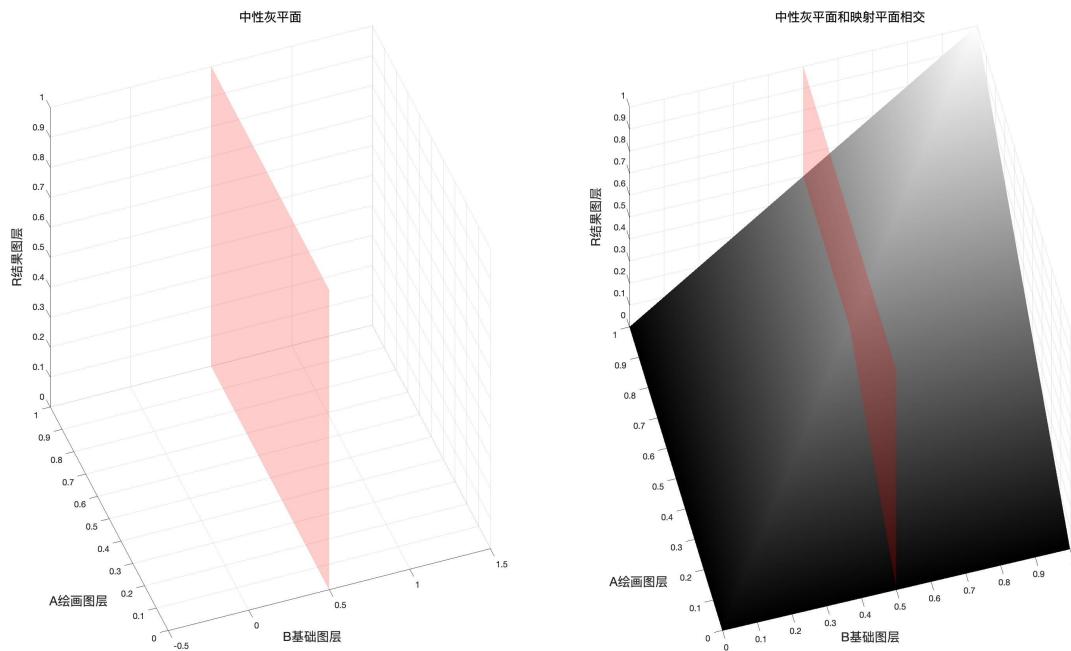
同图平面有时候也可以作为分割平面，比如在实色混合模式的时候。

中性灰平面

如果我们把基础图层或者绘画图层中128中性灰的点都标记出来，我们会得到一个中性灰平面
这个平面一般作为结合两种混合模式的分割面

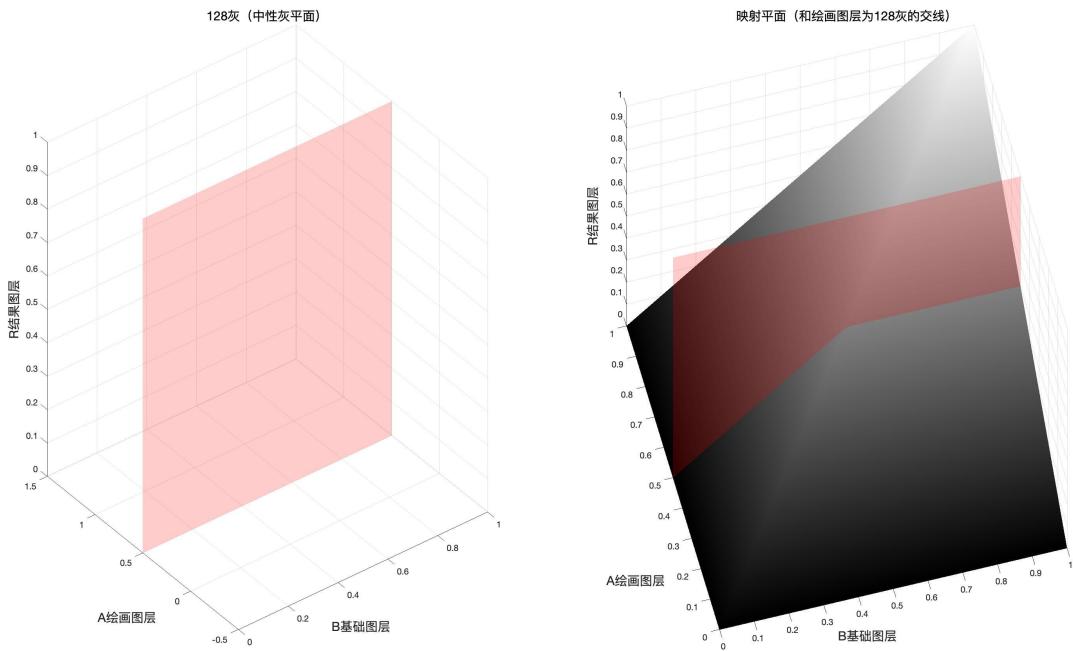
基础图层的中性灰平面

如果两种混合模式是以基础图层的取值作为分界依据的时候，使用此平面作为分割和结合依据。



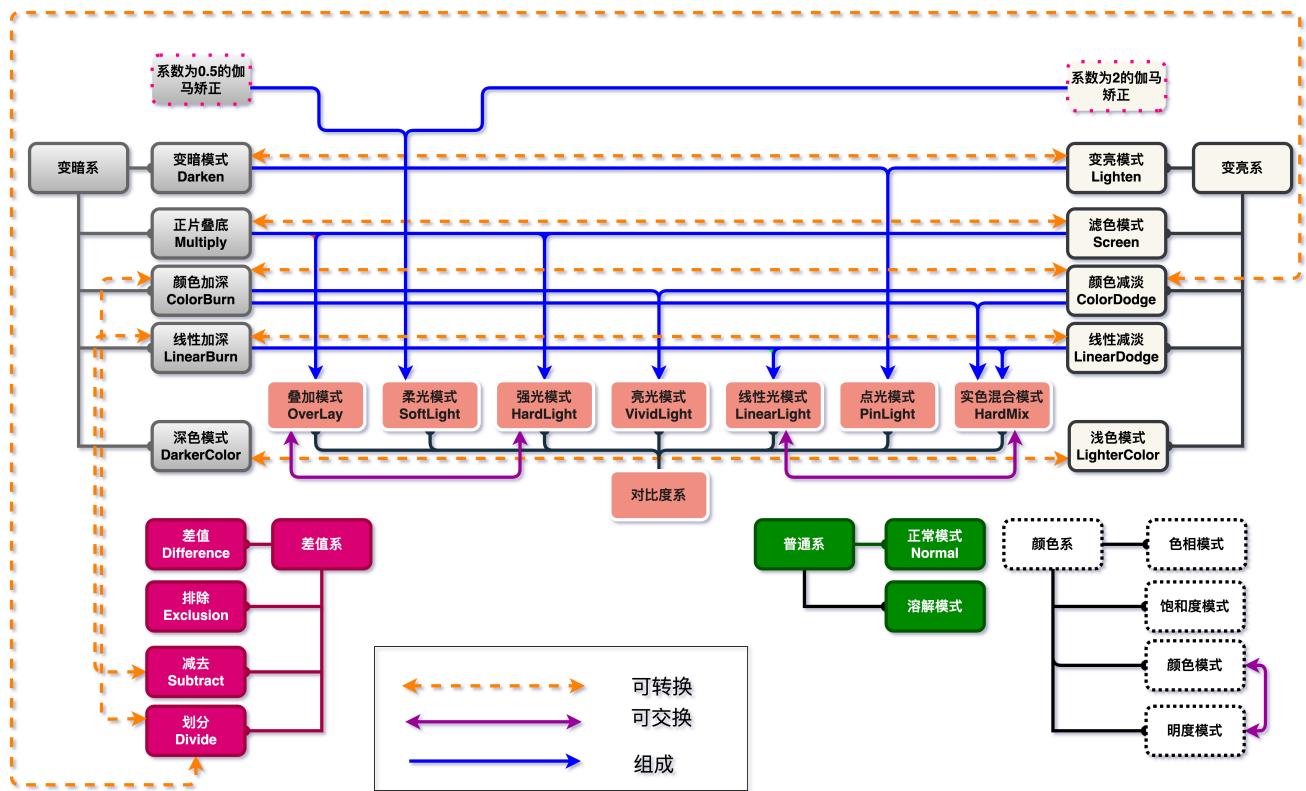
混合图层的中性灰平面

如果两种混合模式是以混合图层的取值作为分界依据的时候，使用此平面作为分割和结合依据。
同图平面，中性灰平面（基础中性灰平面，混图层中性灰平面）都是结合两种混合模式的依据。



关系总览

为了方便大家理解，我们在此给出27中图层混合模式的总览



- **可转换**: 代表两种混合模式可以在一定条件下等效另外一种
- **可交换**: 底图和绘画图层交换，两种混合模式产生的结果相同,也就是互逆
- **组成**: 一种混合模式可以由另外一种组成

特殊关系

从总览那幅图我们可以得出一些有趣的结论

就是有的混合模式在结合一些操作之后，结果等价于另外一种

有的图层混合模式和另外一种是互逆的关系

有些可以组合，有些对黑色无效，有些对白色无效，有些对中性灰无效

总之，他们的一切都蕴含在了公式中，如果你想搞清楚，就仔细研究一下。

可转换

可转换可能有些在实际操作中有偏差，因为在计算过程中可能出现大于1或者小于0的情况，导致被取舍。以下都是理论存在的情况。

- 变亮\$ \Leftrightarrow \$变暗 (三次负片)
- 滤色\$ \Leftrightarrow \$正片叠底 (三次负片)
- 线性减淡\$ \Leftrightarrow \$线性加深 (三次负片)
- 颜色减淡\$ \Leftrightarrow \$颜色加深 (三次负片)
- 浅色\$ \Leftrightarrow \$深色 (三次负片)
- 线性加深\$ \Leftrightarrow \$减去 (一次负片)
- 颜色减淡\$ \Leftrightarrow \$划分 (一次负片)
- 颜色加深\$ \Leftrightarrow \$划分 (两次负片)

互逆 (可交换)

- 叠加\$ \Leftrightarrow \$强光
- 颜色\$ \Leftrightarrow \$明度
- 实色混合\$ \Leftrightarrow \$线性光 (当实色混合填充设置为0.5也就是50%)

组成

- 叠加 \$=\$ 正片叠底\$ + \$滤色
- 强光 \$=\$ 正片叠底\$ + \$滤色
- 线性光 \$=\$ 线性加深\$ + \$线性减淡
- 实色混合 \$=\$ 线性加深\$ + \$线性减淡
- 亮光 \$=\$ 颜色加深\$ + \$颜色减淡
- 点光 \$=\$ 变亮\$ + \$变暗
- 柔光\$ = \$系数2的伽马矫正 \$ + \$ 系数为0.5的伽马矫正

另外五种

- 穿透
- 背后
- 擦除
- 相加
- 相减

图层混合模式的种类

色彩空间分类

分类方法有很多，如果按照色彩空间，可以分成两类，RGB和HSY

RGB

- 除颜色组

HSY

- 颜色组

是否产生新的数值

如果根据运算方式，可以分成两类，一类是替换式，一类是融合式

替换式

包括：

- 正常组（正常，溶解）
- 变暗组（变暗，深色）
- 变亮组（变亮，浅色）
- 颜色组（色相，饱和度，颜色，明度）

这种式的特点是，不会产生新的东西，只会复用之前的东西

比如正常组，正常就是直接使用混合图层作为结果

变暗组的变暗是使用基础图层和混合图层中通道较小的值组成新的像素，深色则是比较基础图层和混合图层通道值的和取小的作为结果

颜色组，则是根据HSY色彩空间，直接替换 HSY的数值，并且再转换为RGB，本质也是没有新的东西产生。

融合式

包括：

- 变暗组（正片叠底，颜色加深，线性加深）
- 变亮组（滤色，颜色减淡，线性减淡）
- 对比度组（叠加，柔光，强光，亮光，线性光，点光，实色混合）
- 差值组（差值，排除，减去，划分）

虽然点光是变暗和变亮的混合，但是混合过程中做了处理，所以此处我们也按照融合式对待
此类的特点是，会产生新的数值。

融合式就是产生了新的数值，比如正片叠底，结果图层的数值由基础图层和混合图层的乘积得到，他是不同于基础图层和混合图层的值。

是否顺序相关

如果根据是否受到图层顺序影响

如果 \$\$BlendMode(b,a)=BlendMode(a,b)\$\$

则说明结果不受图层顺序影响，前提是不调节\$fill\$和\$Opacity\$

顺序无关

- **变暗组** (变暗, 线性加深, 正片叠底, 深色)
- **变亮组** (变亮, 线性减淡, 滤色, 浅色)
- **对比度组** (实色混合)
- **差值组** (差值, 排除)

顺序相关

- **正常组** (正常, 溶解)
- **变暗组** (颜色加深)
- **变亮组** (颜色减淡)
- **对比度组** (叠加, 柔光, 强光, 亮光, 线性光, 点光)
- **差值组** (减去, 划分)
- **颜色组** (色相, 饱和度, 颜色, 明度)

是否fill和Opacity不同

也就是说, 调节填充的百分比, 和不透明度的百分比, 即使数值相同, 结果也有可能不同。

fill和Opacity产生不同影响

- **变暗组** (线性加深, 颜色加深)
- **变亮组** (线性减淡, 颜色减淡)
- **对比度组** (亮光, 线性光, 实色混合)
- **差值组** (差值)

fill和Opacity产生相同影响

- **正常组** (正常, 溶解)
- **变暗组** (变暗, 正片叠底, 深色)
- **变亮组** (变亮, 滤色, 浅色)
- **对比度组** (叠加, 柔光, 强光, 点光)
- **差值组** (排除, 减去, 划分)
- **颜色组** (色相, 饱和度, 颜色, 明度)

混合模式相关符号

图层混合模式我们使用\$BlendMode\$来表示, 这里是一个泛指, 如果涉及到具体的混合模式, 有专门的表示符号, 比如正常模式使用\$Normal\$

使用矩阵我们有:

```

$$LayerA=\left\{ \begin{aligned} &PixA_{(1,1)} \&\dots \&PixA_{(1,n)} \&\vdots \&\dots \&PixA_{(i,1)} \\ &\dots \&PixA_{(i,n)} \&\vdots \&\dots \&PixA_{(m,1)} \&\dots \&PixA_{(m,n)} \end{aligned} \right\}
\right. \\

$$LayerB=\left\{ \begin{aligned} &PixB_{(1,1)} \&\dots \&PixB_{(1,n)} \&\vdots \&\dots \&PixB_{(i,1)} \\ &\dots \&PixB_{(i,n)} \&\vdots \&\dots \&PixB_{(m,1)} \&\dots \&PixB_{(m,n)} \end{aligned} \right\}
\right. \\

$$LayerR=\left\{ \begin{aligned} &PixR_{(1,1)} \&\dots \&PixR_{(1,n)} \&\vdots \&\dots \&PixR_{(i,1)} \\ &\dots \&PixR_{(i,n)} \&\vdots \&\dots \&PixR_{(m,1)} \&\dots \&PixR_{(m,n)} \end{aligned} \right\}
\right. \\

```

再结合上面的三者关系公式 \$\$LayerR=BlendMode(LayerB,LayerA)\$\$

我们得到

```

$$LayerR=\left\{ \begin{aligned} &\text{BlendMode}(PixB_{(1,1)}, PixA_{(1,1)}) \&\dots \&\text{BlendMode}(PixB_{(1,n)}, PixA_{(1,n)}) \&\vdots \&\dots \&\text{BlendMode}(PixB_{(i,1)}, PixA_{(i,1)}) \&\dots \&\text{BlendMode}(PixB_{(i,n)}, PixA_{(i,n)}) \&\vdots \&\dots \&\text{BlendMode}(PixB_{(m,1)}, PixA_{(m,1)}) \&\dots \&\text{BlendMode}(PixB_{(m,n)}, PixA_{(m,n)}) \end{aligned} \right\}
\right. \\

```

对于其中任意项

\$\$Pix_R=BlendMode(Pix_B,Pix_A)\$\$

如果这种混合模式基于RGB色彩空间则

上述表达式可以写为

\$(rc_{\{R\}}, gc_{\{R\}}, bc_{\{R\}})=BlendMode((rc_{\{B\}}, gc_{\{B\}}, bc_{\{B\}}), (rc_{\{A\}}, gc_{\{A\}}, bc_{\{A\}}))\$\$

或者未归一化形式

\$(RC_{\{R\}}, GC_{\{R\}}, BC_{\{R\}})=BlendMode((RC_{\{B\}}, GC_{\{B\}}, BC_{\{B\}}), (RC_{\{A\}}, GC_{\{A\}}, BC_{\{A\}}))\$\$

如果这种混合模式基于HSY色彩空间

\$(H_{\{R\}}, S_{\{R\}}, Y_{\{R\}})=BlendMode((H_{\{B\}}, S_{\{B\}}, Y_{\{B\}}), (H_{\{A\}}, S_{\{A\}}, Y_{\{A\}}))\$\$

在下面的所有组中，如果不特别说明基于RGB色彩空间的混合模式都是使用归一化的数值进行计算。

并且，为了方便大家验证计算的正确性，这里提供一个java程序的链接给各位，GitHub的地址如下，输入你要混合的像素的数值，就可以得到对应混合模式混合的结果数值。

普通组

正常Normal

正常模式是一切的基础，也是我们理解和掌握混合模式的基础，

正常模式可以看作是基于RGB颜色空间，也可以看作是基于HSY色彩空间

公式(泡茶的方式，比如搅拌、静置)

对于像素维度公式

\$\$Pix_r=Normal(Pix_b,Pix_a)=Pix_a\$\$

正常模式通道维度初始公式

\$\$r=Normal(b,a)=a\$\$

融合填充(放多少茶叶)

$\$r=Fill(b,a)= fill\times a + (1-fill)\times b$

融合不透明度(太苦了， 最后加点水)

$\$r=Opacity(b,a)= op\times Fill(b,a) + (1-fill)\times b$

对于整个像素

$\$(r_{rc},r_{gc},r_{bc})=Normal((b_{rc},b_{gc},b_{bc}),(a_{rc},a_{gc},a_{bc}))=(r_{rc},r_{gc},r_{bc})$

后面的混合模式我们不再讨论整个像素的公式， 我们只讨论某个通道的结果。

整个像素融合填充

这里不同的混合模式公式不一定相同。

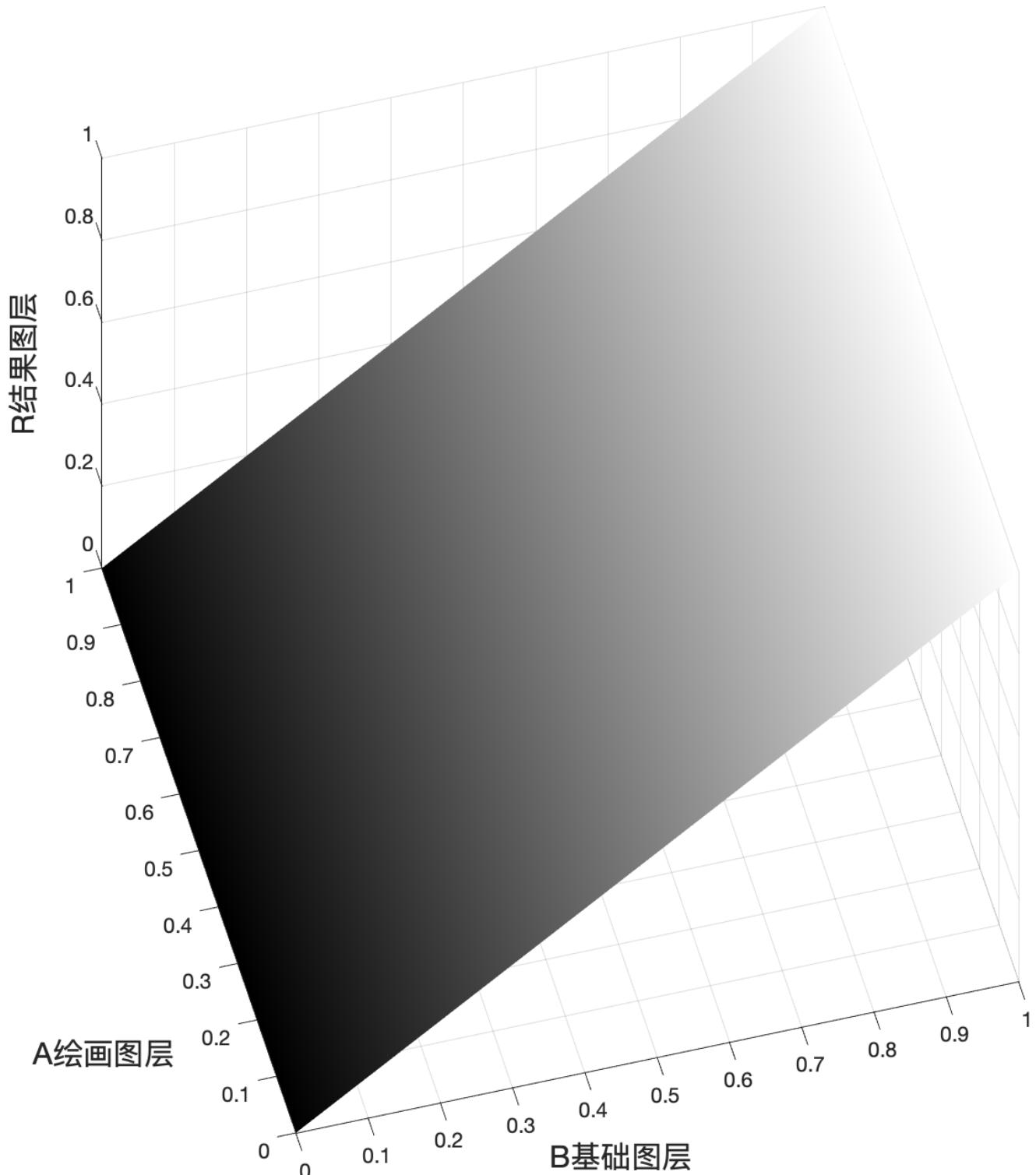
$\$\begin{aligned}(r_{rc},r_{gc},r_{bc})&=Fill((b_{rc},b_{gc},b_{bc}),\\(a_{rc},a_{gc},a_{bc}))&=\left((fill\times a_{rc} + (1-fill)\times b_{rc}),(fill\times a_{gc} + (1-fill)\times b_{gc}),(fill\times a_{bc} + (1-fill)\times b_{bc})\right)\end{aligned}$

整个像素融合不透明度

$\$\begin{aligned}(r_{rc},r_{gc},r_{bc})&= Opacity((b_{rc},b_{gc},b_{bc}),\\(a_{rc},a_{gc},a_{bc}))&=\left((op\times Fill(a_{rc},b_{rc}) + (1-op)\times b_{rc}),(op\times Fill(a_{gc},b_{gc}) + (1-op)\times b_{gc}),(op\times Fill(a_{bc},b_{bc}) + (1-op)\times b_{bc})\right)\end{aligned}$

映射面和相关拓展

映射平面



程序模拟该模式计算结果

```
// 正常模式
public static BlendColor Normal(BlendColor colorBase, BlendColor
colorBlend, double fill, double opacity) {
    double red = colorBlend.red.get01Value() *fill +
colorBase.red.get01Value()* (1 - fill);
    double green = colorBlend.green.get01Value() *fill +
colorBase.green.get01Value()* (1 - fill);
```

```

        double blue = colorBlend.blue.get01Value() *fill +
colorBase.blue.get01Value()*(1 - fill);
        return ColorUtils.Opacity(colorBase, new BlendColor(red *255, green*
255, blue * 255), opacity);
    }

```

借助正常模式

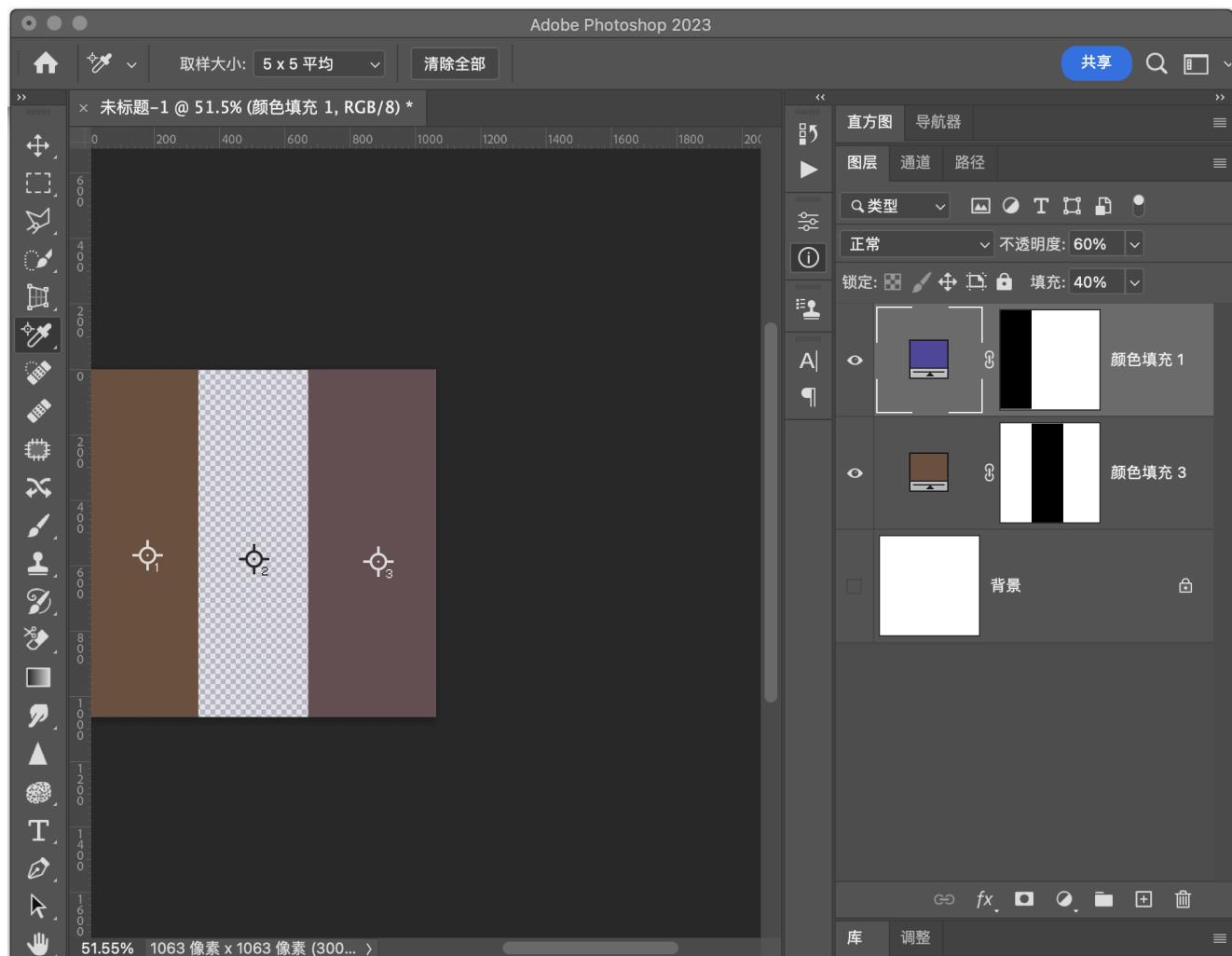
我们可以看到不透明度和填充的关系

后面的模式讨论我们都不再说明像素级别的公式，因为没有必要，我们只讨论通道级别的公式。

我们设定\$fill\$为\$40%\$, \$Opacity\$ 为\$60%\$

基础图层	RGB [111.00, 80.00, 60.00] ~ HSY [23.53, 51.00, 87.10] ~ HSB [23.53, 45.95, 43.53]
混合图层	RGB [80.00, 70.00, 156.00] ~ HSY [246.98, 86.00, 82.46] ~ HSB [246.98, 55.13, 61.18]
=====正常组=====	
正 常(Normal)	RGB [103.56, 77.60, 83.04] ~ HSY [347.43, 25.96, 85.99] ~ HSB [347.43, 25.07, 40.61]

我们在PS中使用这两种颜色进行验证，发现符合我们的算法。



用途示例

该模式是默认模式，在不调节填充和不透明度的情况下，就是上方像素点覆盖下方的像素点。结合不透明度或者填充，可以实现和下方图层的简单混色。

溶解Dissolve

溶解模式初始公式

$\$r=Dissolve(b,a)=a\$$

如果融合了填充

$\$r=Fill(b,a)=\text{Random}_{\{\text{fill}\}}(\text{Dissolve}(b,a),b)\$$

不透明度对溶解模式是无效的。

$\$fill\$$ 的值越大，则下层像素暴露的可能性越小

程序模拟该模式计算结果

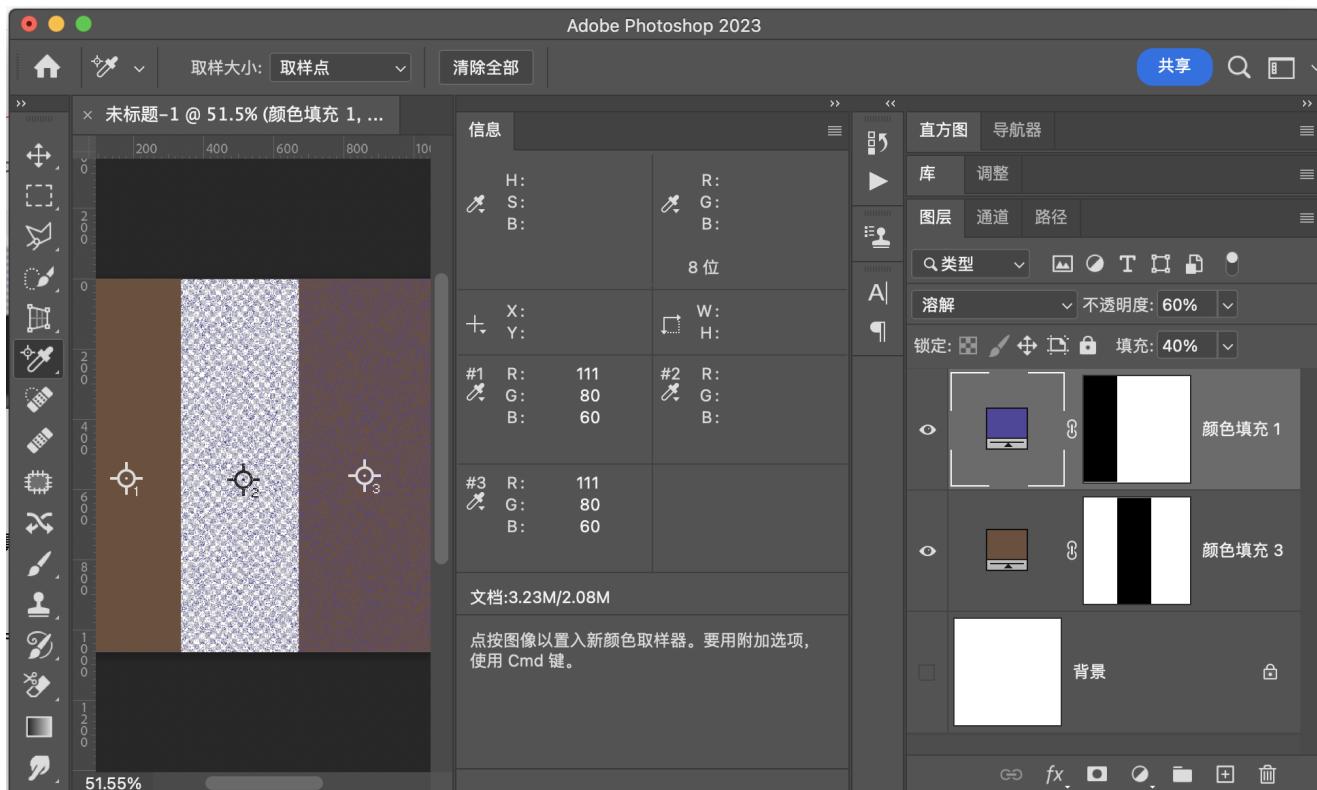
```
// 溶解模式
public static BlendColor Dissolve(BlendColor colorBase, BlendColor
colorBlend, double fill, double opacity) {
    double rand = Math.random(); // 产生随机数
    if (rand < fill) {
        return colorBlend;
    } else {
        return colorBase;
    }
}
```

溶解(Dissolve) RGB[111.00, 80.00, 60.00] ~ HSY[23.53, 51.00,
87.10] ~ HSB[23.53, 45.95, 43.53]

验证

溶解模式是根据概率来实现决定显示下方像素还是上方像素

于是取样点不同，则结果可以是上方的像素也可以是下方的像素



用途示例

可以通过该模式实现一下粒子效果

变暗组

这一组是基于RGB色彩空间，RGB三个通道的数值取值范围是 $[0,255]$ ，归一化之后取值范围是 $[0,1]$ 。这一组就是利用一系列运算让基础图层的三个通道的数值或者他们的和小于等于原值。这也是变暗组的本质。

和下面的变亮组一样，变暗组有两种方式将通道值或者通道值的和变小，那就是替换和运算，替换包括变暗和深色，运算包括其他三种。

变暗模式

如果将基础图层每个像素每个通道的数值都变小，最简单的方式就是选择原图图层像素通道值和混合图层像素通道值中比较小的那个作为结果图层像素通道值，这样的图层中每个像素点的通道值都小于等于原值，图像自然就会变暗。

公式

$$r = \text{Darken}(b, a) = \min(b, a)$$

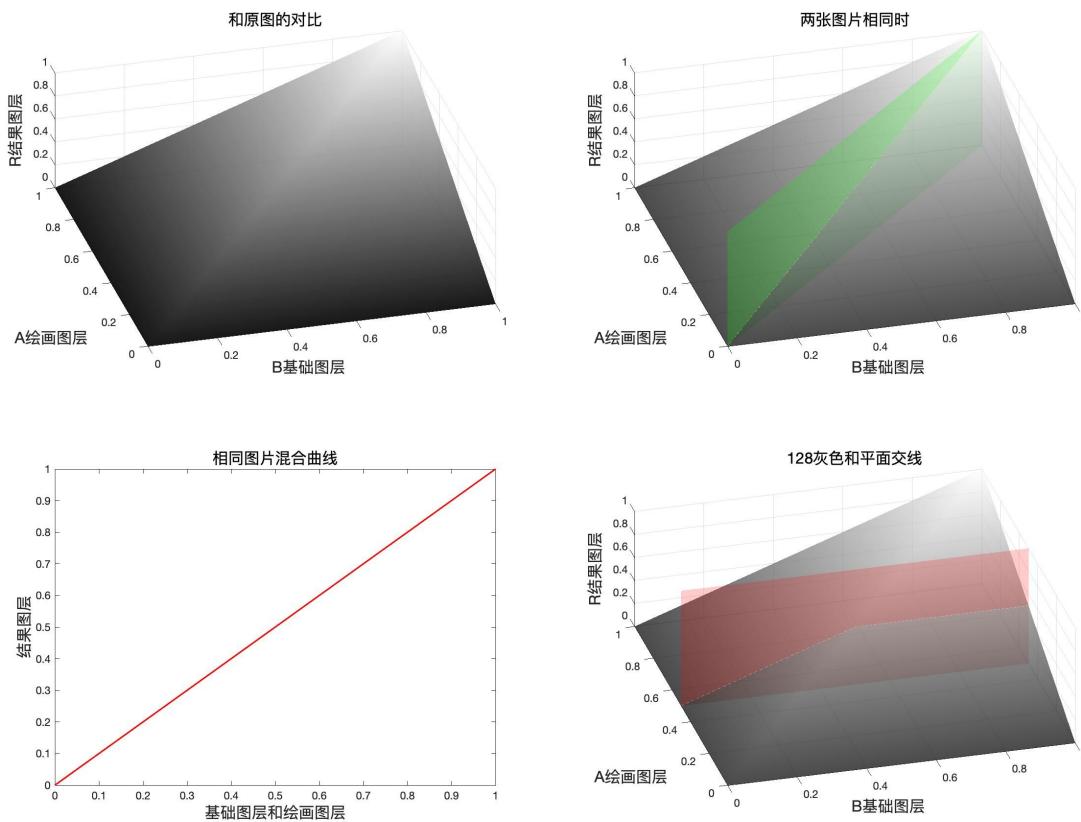
融合填充

$$r = \text{Fill}(b, a) = \text{fill} \times \text{Darken}(b, a) + (1 - \text{fill}) \times b$$

融合不透明度

$\$r = \text{Opacity}(b, a) = op \times \text{Fill}(b, a) + (1 - op) \times b$

映射面和同图等效曲线



同图曲线表达式

$\$r = \text{Darken}(b, b) = b$

这里我们简单认为fill和opacity都是100，因为我们日常使用中几乎用不到改变这两个值并且需要模拟同图曲线的情况，使用这里只讨论最简单的类型。但是其他类型可以通过我们提供的公式自行推导，但是这里没有必要写出来。下同。

程序模拟该模式计算结果

```
// 变暗
public static BlendColor Darken(BlendColor colorBase, BlendColor
colorBlend, double fill, double opacity) {
    double red = DarkenChannel(colorBase.red.get01Value(),
colorBlend.red.get01Value(), fill);
    double green = DarkenChannel(colorBase.green.get01Value(),
colorBlend.green.get01Value(), fill);
    double blue = DarkenChannel(colorBase.blue.get01Value(),
colorBlend.blue.get01Value(), fill);
    return ColorUtils.Opacity(colorBase, new BlendColor(red * 255, green*
255, blue * 255), opacity);
}
```

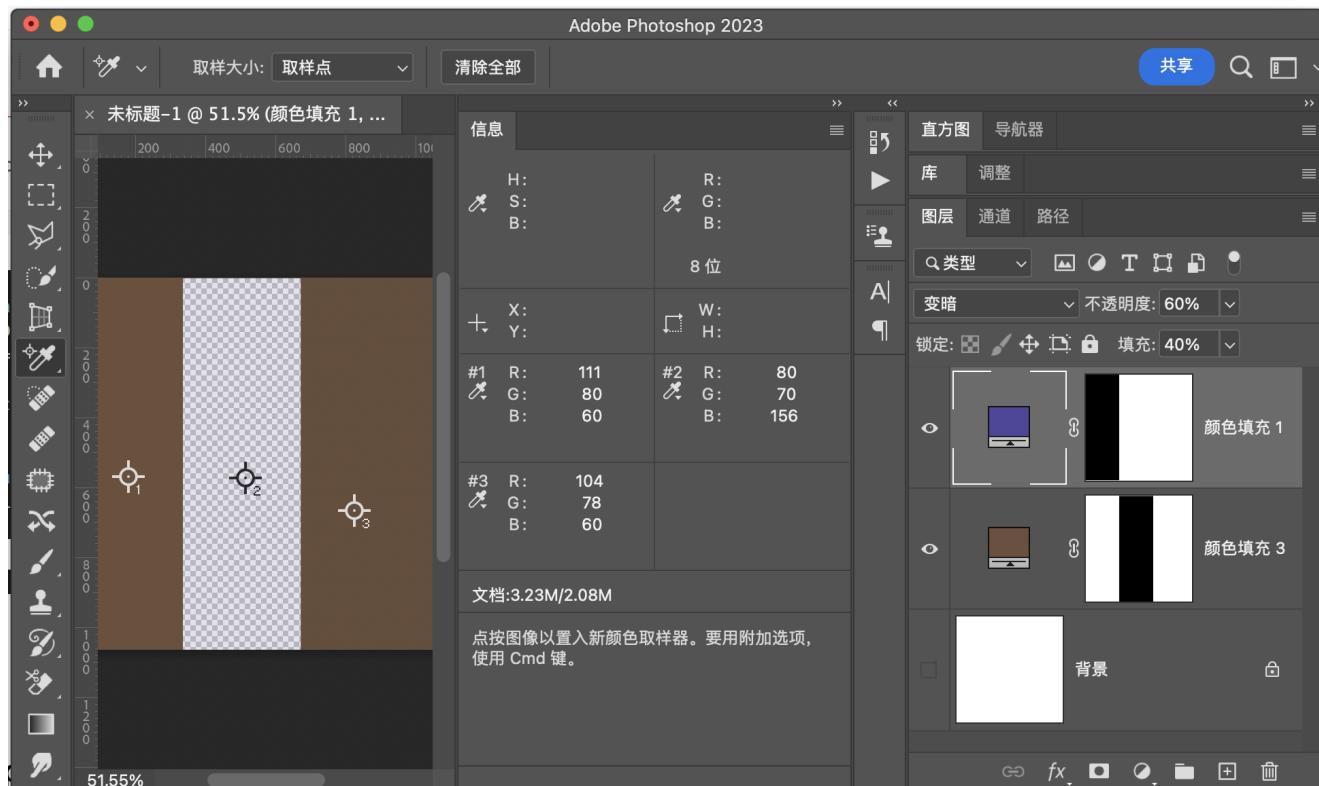
```

private static double DarkenChannel(double base, double blend, double fill) {
    return Math.min(base, blend) * fill + (1 - fill) * base;
}

```

变 暗(Darken) RGB [103.56, 77.60, 60.00] ~ HSY [24.24, 43.56, 83.45] ~ HSB [24.24, 42.06, 40.61]

验证



用途示例

1:组合成为对比度组的点光Pinlight模式

2:此处是深色模式的简化版本，但是我们还是可以通过它来实现一些溶图操作

正片叠底Multiply

如果将混合图层像素的通道数值和原图像素通道数值相乘，则因为归一化之后的数值都是小于等于1的所以，原值乘一个小于1的数值一定小于等于原来的值。所以图像会变暗。

公式

$\$r = \text{Multiply}(b, a) = b \times a$

融合填充

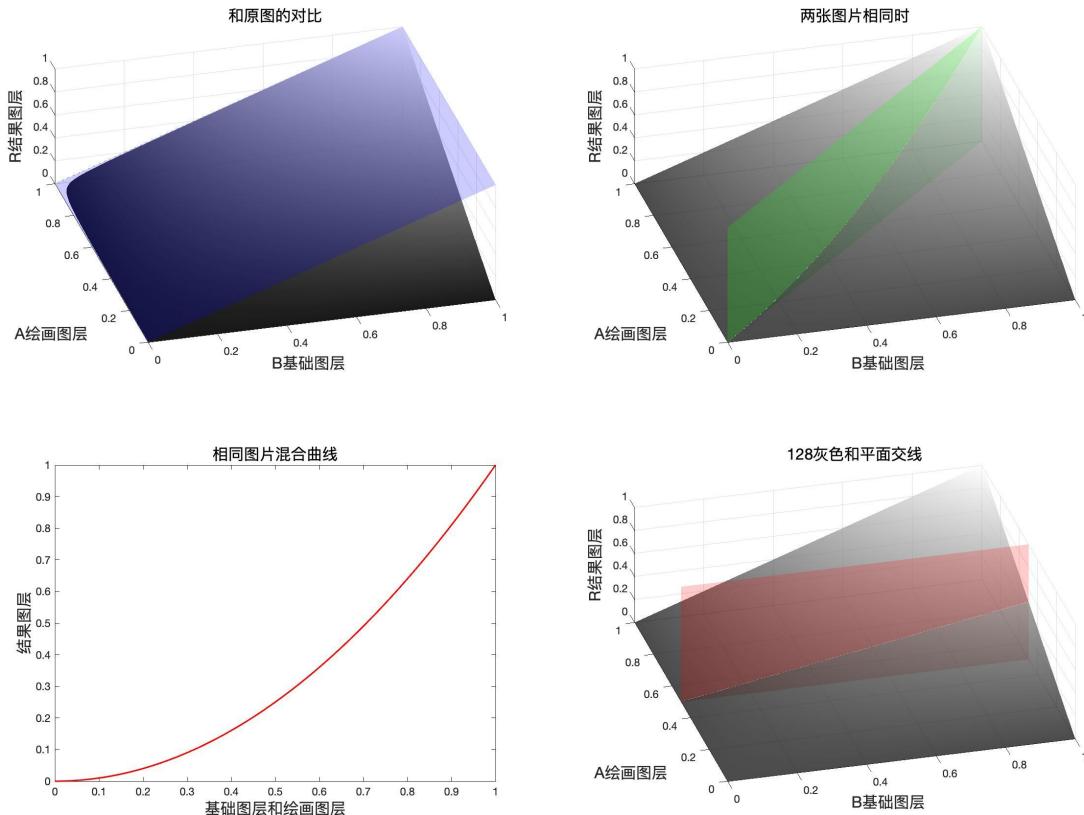
$\$r = \text{Fill}(b, a) = \text{fill} \times \text{Multiply}(b, a) + (1 - \text{fill}) \times b$

融合不透明度

$\$r = \text{Opacity}(b, a) = \text{op} \times \text{Fill}(b, a) + (1 - \text{op}) \times b$

映射面和同图等效曲线

正片叠底的映射面，同图面，同图曲线和中性灰平面



同图曲线表达式

$\$r = \text{Multiply}(b, b) = b^2$

程序模拟该模式计算结果

```
// 正片叠底
public static BlendColor Multiply(BlendColor colorBase, BlendColor
colorBlend, double fill, double opacity) {
    double red = MultiplyChannel(colorBase.red.get01Value(),
colorBlend.red.get01Value(), fill);
    double green = MultiplyChannel(colorBase.green.get01Value(),
colorBlend.green.get01Value(), fill);
    double blue = MultiplyChannel(colorBase.blue.get01Value(),
colorBlend.blue.get01Value(), fill);
    return ColorUtils.Opacity(colorBase, new BlendColor(red * 255, green
* 255, blue * 255), opacity);
```

```

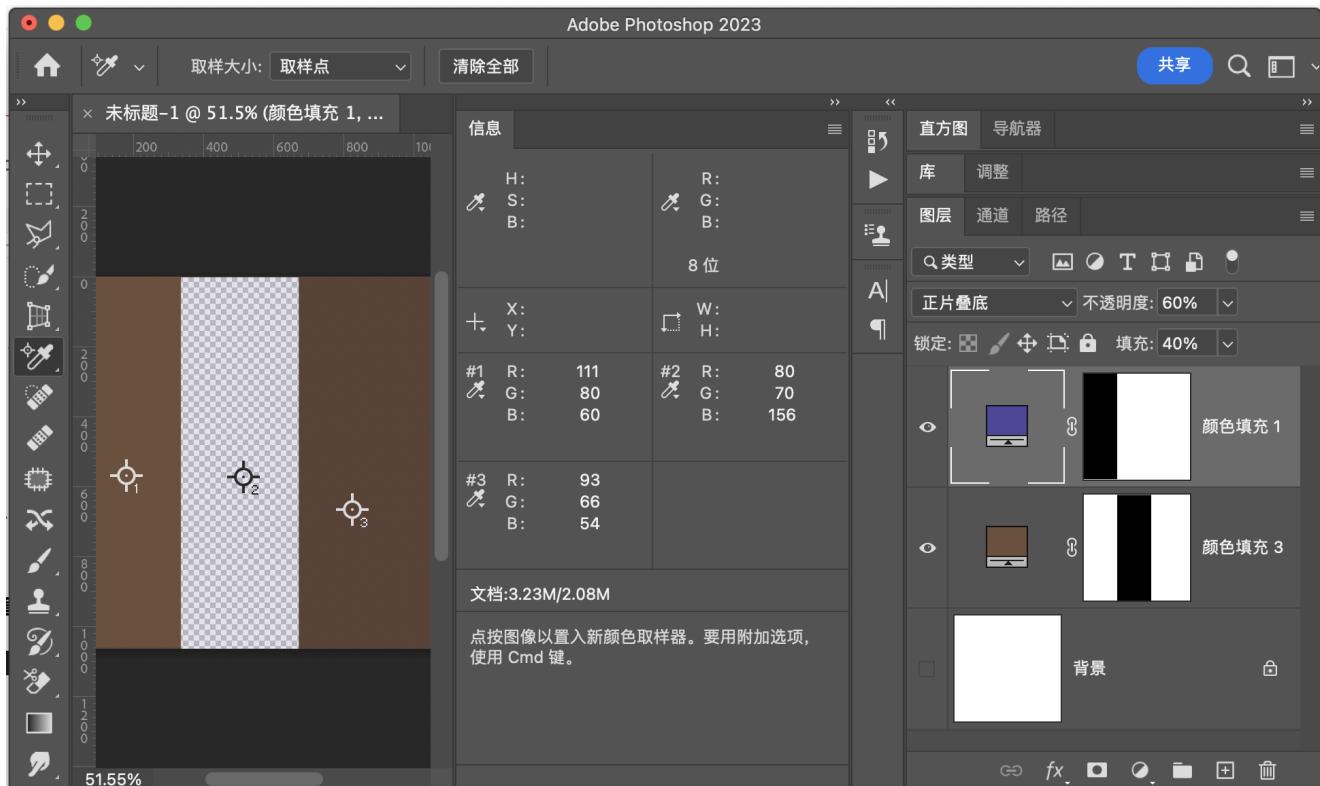
    }

private static double MulitplyChannel(double base, double blend, double fill) {
    return ColorUtils.round((base * blend) * fill + (1 - fill) * (base),
1, 0);
}

```

正片叠底(Mulitply) RGB [92.72, 66.07, 54.41] ~ HSY [18.26, 38.31, 72.78] ~ HSB [18.26, 41.32, 36.36]

验证



用途示例

- 1: 和滤色模式组合成强光和叠加模式
- 2: 给比较亮的图片添加纹理
- 3: 扣除白色背景

线性加深LinearBurn

此模式本质是减法，就是使用混合图层像素通道数值的补也就是附片和原图相减，如果大于1则取1小于0则取0，并且此模式需要对填充特殊处理。线性加深可以通过划分和颜色加深转化。

公式

$$r = \text{LinearBurn}(b, a) = b - (1-a) = b + a - 1$$

融合填充

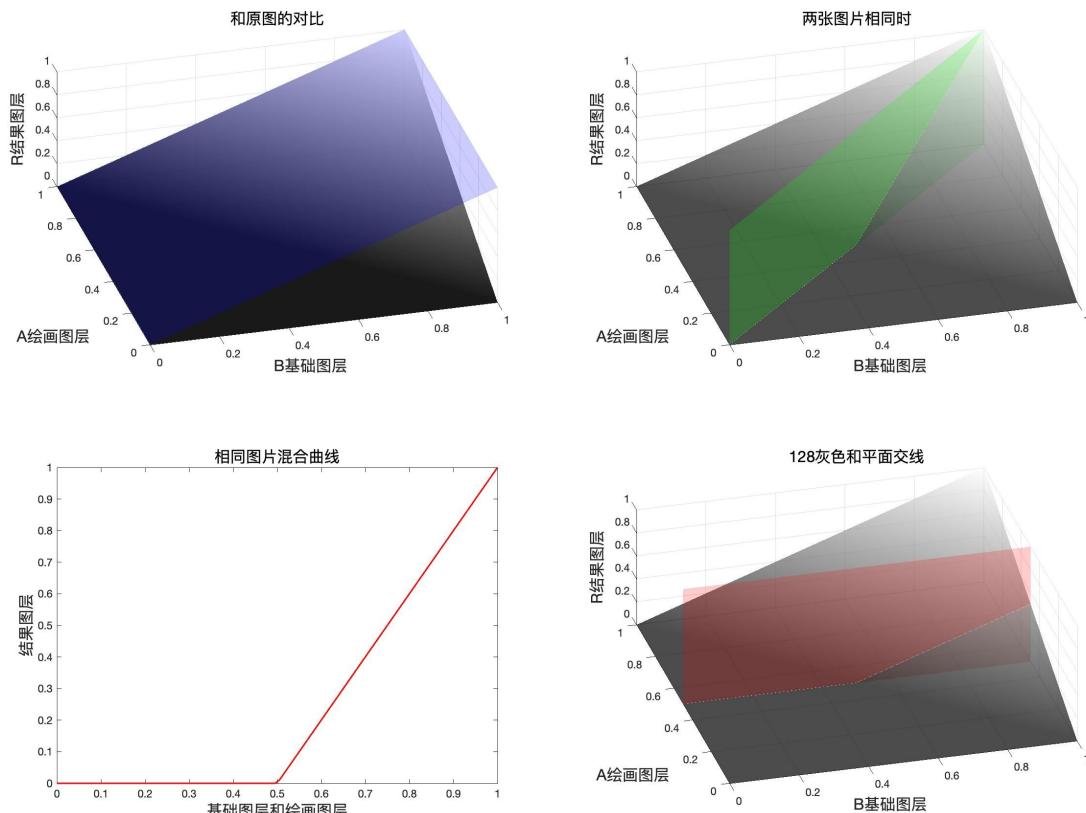
$\$r = \text{Fill}(b, a) = b - (1-a) \times \text{fill}$

融合不透明度

$\$r = \text{Opacity}(b, a) = op \times \text{Fill}(b, a) + (1-op) \times b$

映射面和同图等效曲线

线性加深的映射面，同图面，同图曲线和中性灰平面



同图曲线表达式

$\$r = \text{LinearBurn}(b, a) = 2 \times b - 1$

程序模拟该模式计算结果

```
// 线性加深
public static BlendColor LinearBurn(BlendColor colorBase, BlendColor
colorBlend, double fill, double opacity) {
    double red = LinearBurnChannel(colorBase.red.get01Value(),
colorBlend.red.get01Value(), fill);
    double green = LinearBurnChannel(colorBase.green.get01Value(),
colorBlend.green.get01Value(), fill);
    double blue = LinearBurnChannel(colorBase.blue.get01Value(),
colorBlend.blue.get01Value(), fill);
```

```

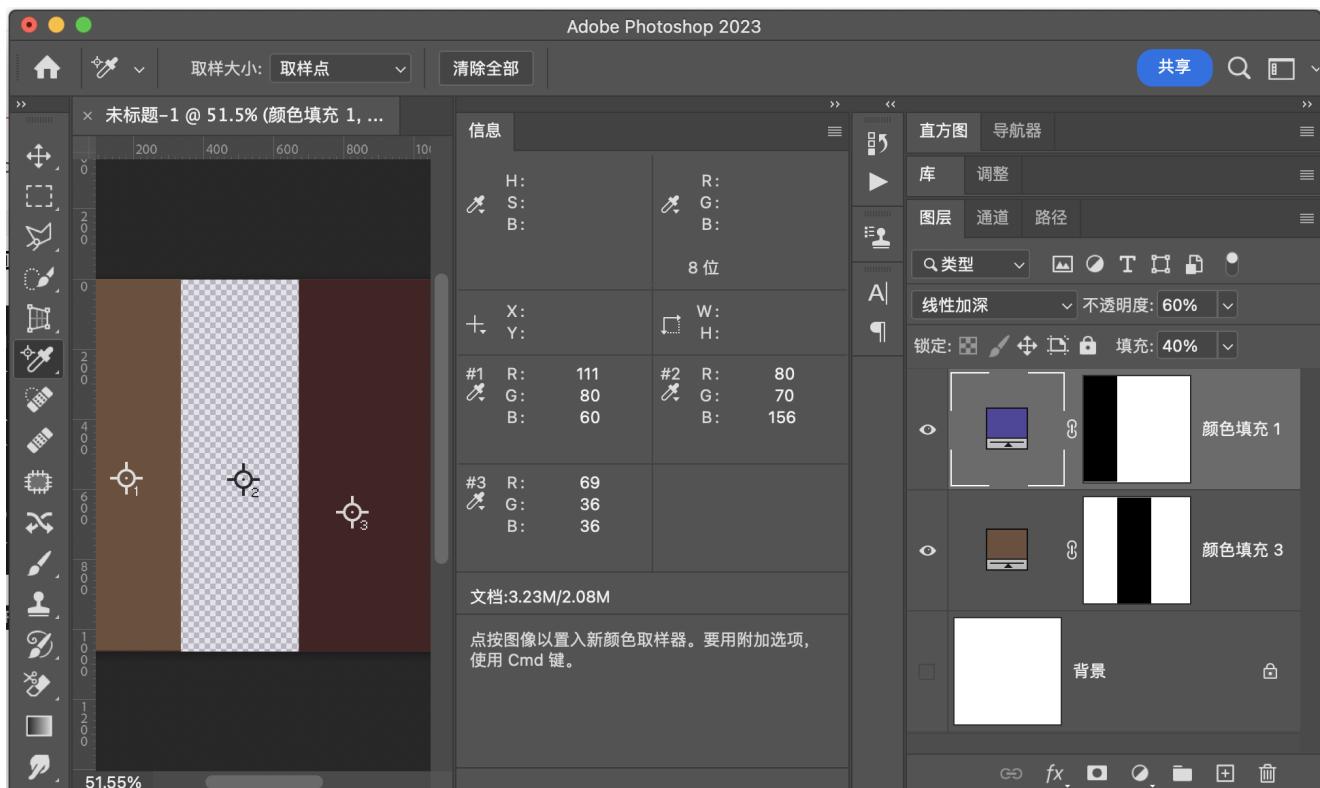
        return ColorUtils.Opacity(colorBase, new BlendColor(red *255, green*255, blue * 255), opacity);
    }

private static double LinearBurnChannel(double base, double blend,
double fill) {
    return ColorUtils.round(base - (1 - blend) * fill, 1, 0);
}

```

线性加深(LinearBurn) RGB[69.00, 35.60, 36.24]~ HSY[358.85, 33.40, 45.69]~ HSB[358.85, 48.41, 27.06]

验证



用途示例

- 1:组成线性光
- 2:添加光效可做滤镜

颜色加深ColorBurn

此模式的本质是线性加深和混合图层相除，由此可以得知他可以和线性加深通过正片叠底转化。

公式

$\begin{aligned} r = \text{ColorBurn}(b, a) &= 1 - \frac{1-b}{1-(1-a)} \end{aligned}$

这里之所以这么写，是因为后面要加入 fill，也就是填充。

推导一下

$\begin{aligned} r = \text{ColorBurn}(b, a) &= 1 - \frac{1-b}{1-(1-a)} = \frac{1-(1-a)-(1-b)}{1-(1-a)} \\ &\&= \frac{b-(1-a)}{1-(1-a)} \&= \frac{\text{LinearBurn}(b, a)}{1-(1-a)} = \frac{\text{LinearBurn}(b, a)}{a} \end{aligned}$

从这里可以看出，线性加深可以和颜色加深相互转换，通过划分和正片叠底。但是这里涉及到其他的混合模式，而非单纯负片，所以不把他们放到可以互相转化的分类中

融合填充

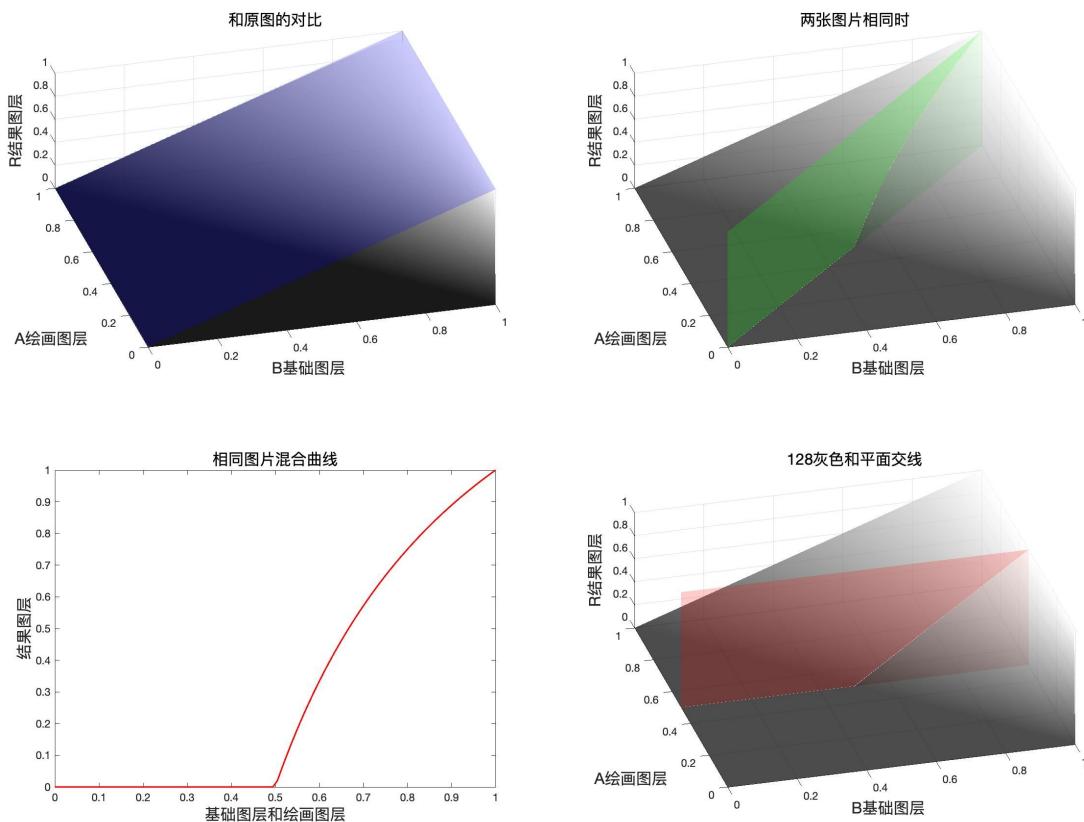
$r = \text{Fill}(b, a) = 1 - \frac{1-b}{1-(1-a)} \times \text{fill}$

融合不透明度

$r = \text{Opacity}(b, a) = \text{op} \times \text{Fill}(b, a) + (1-\text{op}) \times b$

映射面和同图等效曲线

颜色加深的映射面，同图面，同图曲线和中性灰平面



同图曲线表达式

$r = \text{ColorBurn}(b, b) = 2 - \frac{1}{b}$

两次负片转划分

\$\$\begin{aligned} &r=1-\text{ColorBurn}(1-b,a) \\ &=1-(1-\frac{1-(1-b)}{a}) \\ &=\frac{b}{a} \end{aligned}\$\$

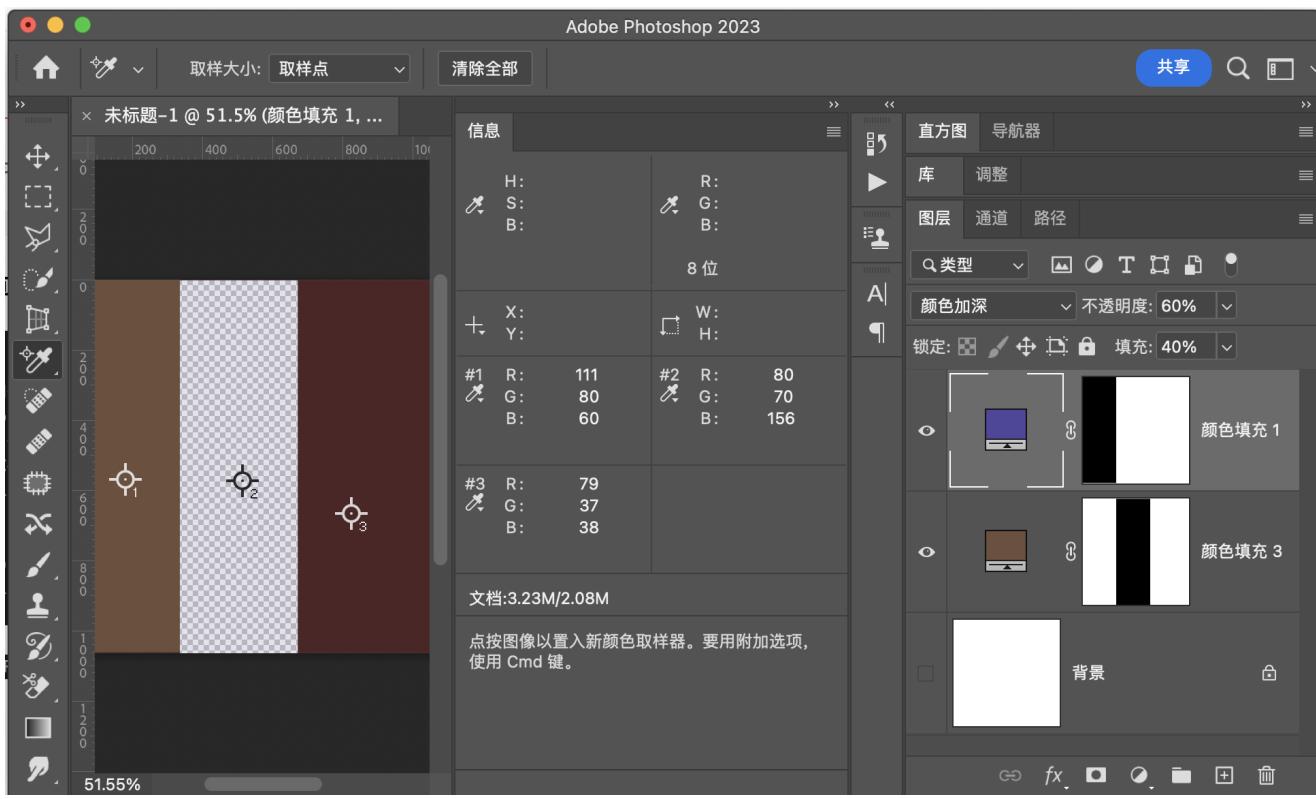
程序模拟该模式计算结果

```
// 颜色加深
public static BlendColor ColorBurn(BlendColor colorBase, BlendColor
colorBlend, double fill, double opacity) {
    double red = ColorBurnChannel(colorBase.red.get01Value(),
colorBlend.red.get01Value(), fill);
    double green = ColorBurnChannel(colorBase.green.get01Value(),
colorBlend.green.get01Value(), fill);
    double blue = ColorBurnChannel(colorBase.blue.get01Value(),
colorBlend.blue.get01Value(), fill);
    return ColorUtils.Opacity(colorBase, new BlendColor(red * 255, green
* 255, blue * 255), opacity);
}

private static double ColorBurnChannel(double base, double blend, double
fill) {
    return ColorUtils.round(1 - Math.min(1, (1 - base) / ((1 - (1 -
blend) * fill))), 1, 0);
}
```

颜色加深(ColorBurn) RGB[78.31, 37.07, 38.49] ~ HSY[357.94, 41.24,
49.60] ~ HSB[357.94, 52.66, 30.71]

验证



用途示例

1:组合成为亮光模式

2:完成特殊光影效果，例如给太阳添加一些颜色或光晕，并且保留亮部细节,该模式的特点也是相对线性加深，可以保留底图的亮部细节。

深色Darker

深色模式可以理解为变暗模式的加强变暗模式，或者是粗略的变暗模式，因为其不产生新的像素，就像溶解模式一样。

计算细节简单来说就是求和，比较大小，小的留下了，若求和的结果一样，就计算明度，明度小的留下了。

公式

```
 $$\begin{aligned}
 & \text{Pix\_r} \&= \text{Darker}(\text{Pix\_b}, \text{Pix\_a}) \&= \left\{ \begin{aligned}
 & \text{Pix\_a} \&& \text{Sum}(\text{Pix\_a}) < \text{Sum}(\text{Pix\_b}) \& \text{Pix\_b} \&& \\
 & \text{Sum}(\text{Pix\_a}) > \text{Sum}(\text{Pix\_b}) \& \& \text{Sum}(\text{Pix\_a}) = \text{Sum}(\text{Pix\_b}) \& \text{且} \text{Lum}(\text{Pix\_b}) < \text{Lum}(\text{Pix\_a}) \& \text{Pix\_a} \&& \\
 & \text{Sum}(\text{Pix\_a}) = \text{Sum}(\text{Pix\_b}) \& \text{且} \text{Lum}(\text{Pix\_b}) > \text{Lum}(\text{Pix\_a}) \end{aligned} \right. \\
 & \end{aligned} $$
 
$$\text{Pix\_r} = \text{Fill}(\text{Pix\_b}, \text{Pix\_a}) = \text{fill} \times \text{Pix\_r} + \text{Pix\_b}$$


$$\text{Lum}(\text{Pix}) = 0.3\text{R} + 0.59\text{G} + 0.11\text{B}$$


$$\text{Sum}(\text{Pix}) = \text{RC} + \text{GC} + \text{BC}$$


```

融合填充

$\text{r} = \text{Fill}(\text{Pix_b}, \text{Pix_a}) = \text{fill} \times \text{Pix_r} + \text{Pix_b}$

融合不透明度

\$\$r=Opacity(Pix_b,Pix_a)=op\times Fill(Pix_b,Pix_a)+(1-op)\times Pix_b\$\$

映射面和同图等效曲线

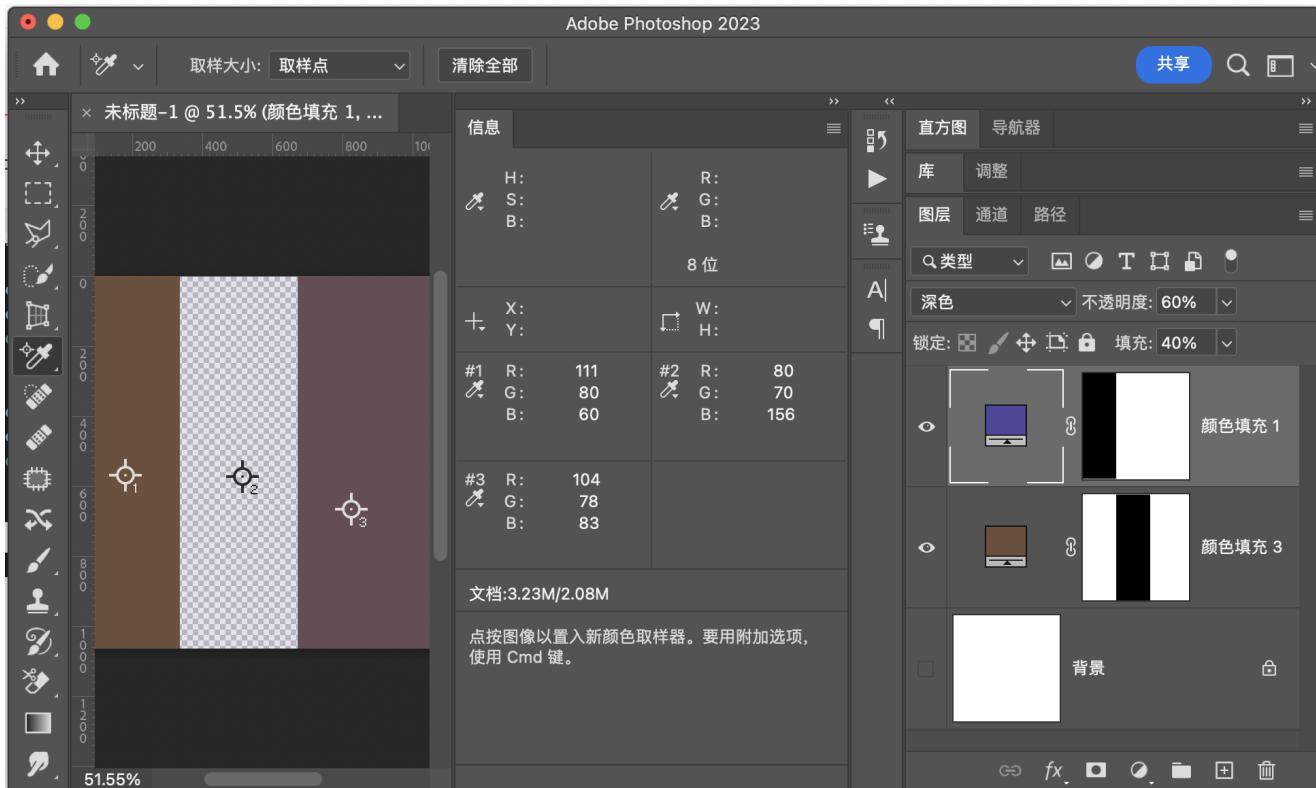
\$\$Pix_r=Pix_b\$\$

程序模拟该模式计算结果

```
// 深色
public static BlendColor Darker(BlendColor colorBase, BlendColor
colorBlend, double fill, double opacity) {
    double sumBase = colorBase.red.value + colorBase.green.value +
colorBase.blue.value;
    double sumBlend = (colorBlend.red.value + colorBlend.green.value +
colorBlend.blue.value) * fill;
    if (sumBase == sumBlend) {
        if (colorBase.getLum() < colorBlend.getLum()) {
            double red = colorBase.red.get01Value();
            double green = colorBase.green.get01Value();
            double blue = colorBase.blue.get01Value();
            return ColorUtils.Opacity(colorBase, new BlendColor(red *
255, green * 255, blue * 255), opacity);
        } else {
            double red = colorBase.red.get01Value() * (1 - fill) +
colorBlend.red.get01Value() * fill;
            double green = colorBase.green.get01Value() * (1 - fill) +
colorBlend.green.get01Value() * fill;
            double blue = colorBase.blue.get01Value() * (1 - fill) +
colorBlend.blue.get01Value() * fill;
            return ColorUtils.Opacity(colorBase, new BlendColor(red *
255, green * 255, blue * 255), opacity);
        }
    }
    if (sumBase < sumBlend) {
        double red = colorBase.red.get01Value();
        double green = colorBase.green.get01Value();
        double blue = colorBase.blue.get01Value();
        return ColorUtils.Opacity(colorBase, new BlendColor(red *
255, green * 255, blue * 255), opacity);
    }
    double red = colorBase.red.get01Value() * (1 - fill) +
colorBlend.red.get01Value() * fill;
    double green = colorBase.green.get01Value() * (1 - fill) +
colorBlend.green.get01Value() * fill;
    double blue = colorBase.blue.get01Value() * (1 - fill) +
colorBlend.blue.get01Value() * fill;
    return ColorUtils.Opacity(colorBase, new BlendColor(red * 255,
green * 255, blue * 255), opacity);
}
```

深 色(Darker) RGB [103.56, 77.60, 83.04] ~ HSY [347.43, 25.96, 85.99] ~ HSB [347.43, 25.07, 40.61]

验证



用途示例

用于替换图中某些像素点，一般用于两张十分相似的图片的融合，比如连拍，延时摄影的一组照片融合成一张图。

变亮组

变亮组是变暗组的相反模式，并且都可以通过负片操作来实现相互转换

变亮组的本质就是以混合图层的像素为参数，对原图层像素的数值进行增大，数值增大了，图片就变亮了。如果混合图层中像素点的通道值为0，则此时该通道的计算结果和原图一致。

变亮 Lighten

变亮模式和变暗模式相反，变暗时取最小值，变亮就是取最大值，具体做法就是取原图层和混合图层像素点中三个通道各自的最大值，保留最大值组成的像素作为结果像素。

公式

$$r = \text{Lighten}(b, a) = \text{Max}(b, a)$$

融合填充

$\$r = \text{Fill}(b, a) = \text{fill} \times \text{Lighten}(b, a) + (1 - \text{fill}) \times b$

融合不透明度

$\$r = \text{Opacity}(b, a) = \text{op} \times \text{Fill}(b, a) + (1 - \text{op}) \times b$

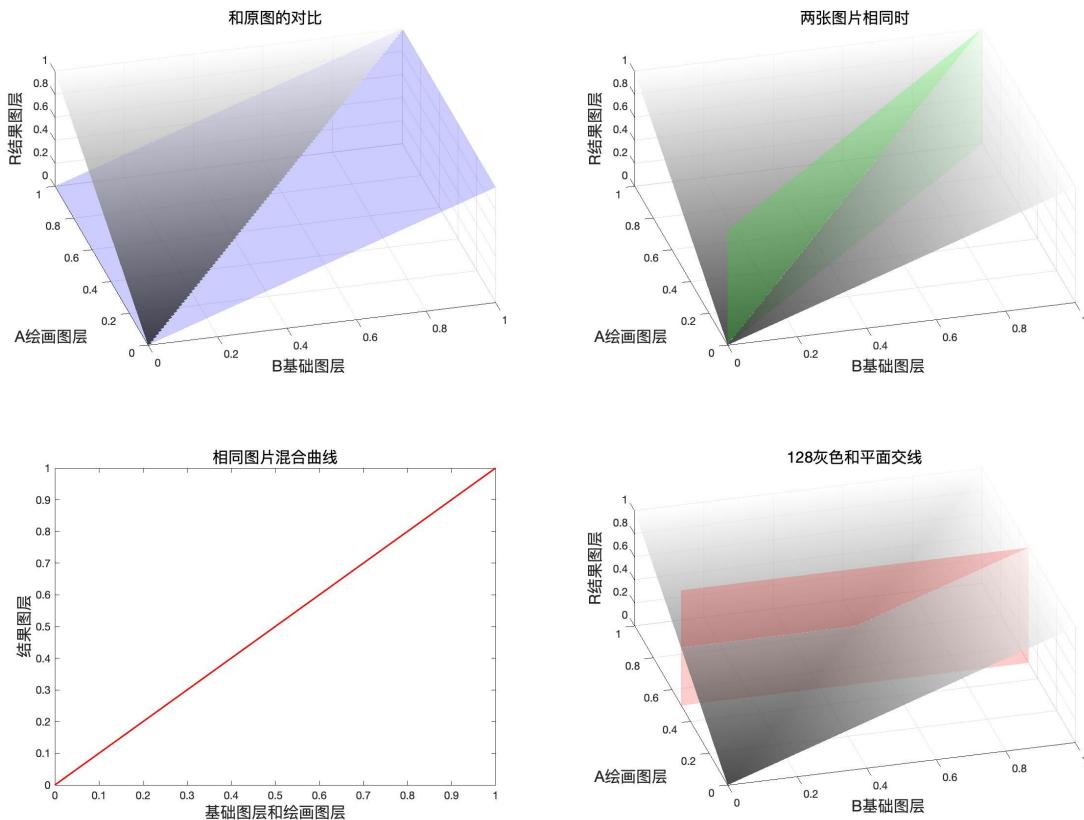
可以写作

三次负片操作相互转换

```
 $\$r \begin{aligned} &= \text{Lighten}(b, a) \\ &= 1 - \text{Min}(1 - b, 1 - a) \\ &= 1 - \text{Darken}(1 - b, 1 - a) \end{aligned}$ 
```

也就是说，三次负片操作可以实现变暗模式和变亮模式的相互转换

映射面和同图等效曲线



程序模拟该模式计算结果

```
// 变亮模式
public static BlendColor Lighten(BlendColor colorBase, BlendColor
colorBlend, double fill, double opacity) {
    double red = LightenChannel(colorBase.red.get01Value(),
colorBlend.red.get01Value(), fill);
    double green = LightenChannel(colorBase.green.get01Value(),
colorBlend.green.get01Value(), fill);
```

```

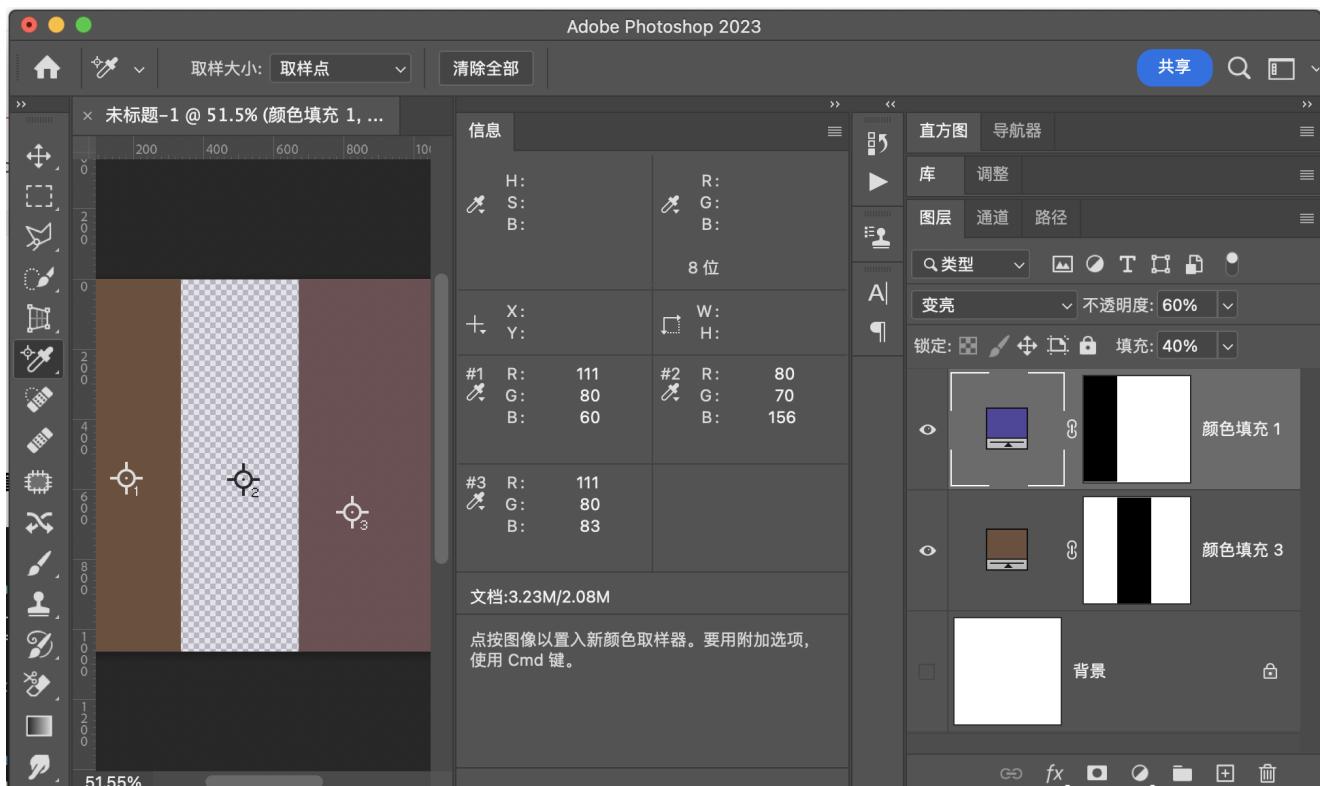
        double blue = LightenChannel(colorBase.blue.get01Value(),
colorBlend.blue.get01Value(), fill);
        return ColorUtils.Opacity(colorBase, new BlendColor(red * 255,
green * 255, blue * 255), opacity);
    }

    private static double LightenChannel(double base, double blend,
double fill) {
        return Math.max(base, blend * fill) * fill + (1 - fill) * base;
}

```

变亮(Lighten) RGB [111.00, 80.00, 83.04] ~ HSY [354.12, 31.00, 89.63] ~ HSB [354.12, 27.93, 43.53]

验证



用途示例

1:组成叠加和强光

2:保留通道中较大的值，并且组成新的像素，可用于将背景较暗或相似但是部分不相同的图片融合。相当于做了一个复杂的通道蒙版，并且这个蒙版非常精确，精确到通道。

滤色 Screen

滤色时正片叠底的一种负片组合，和变暗变亮一样可以通过三次负片操作相互转换。

公式

$\$r=Screen(b,a)=1-(1-b)(1-a)\$\$$

融合填充

$\$r= Fill(b,a) =fill\times Screen(b,a)+(1-fill)\times b\$\$$

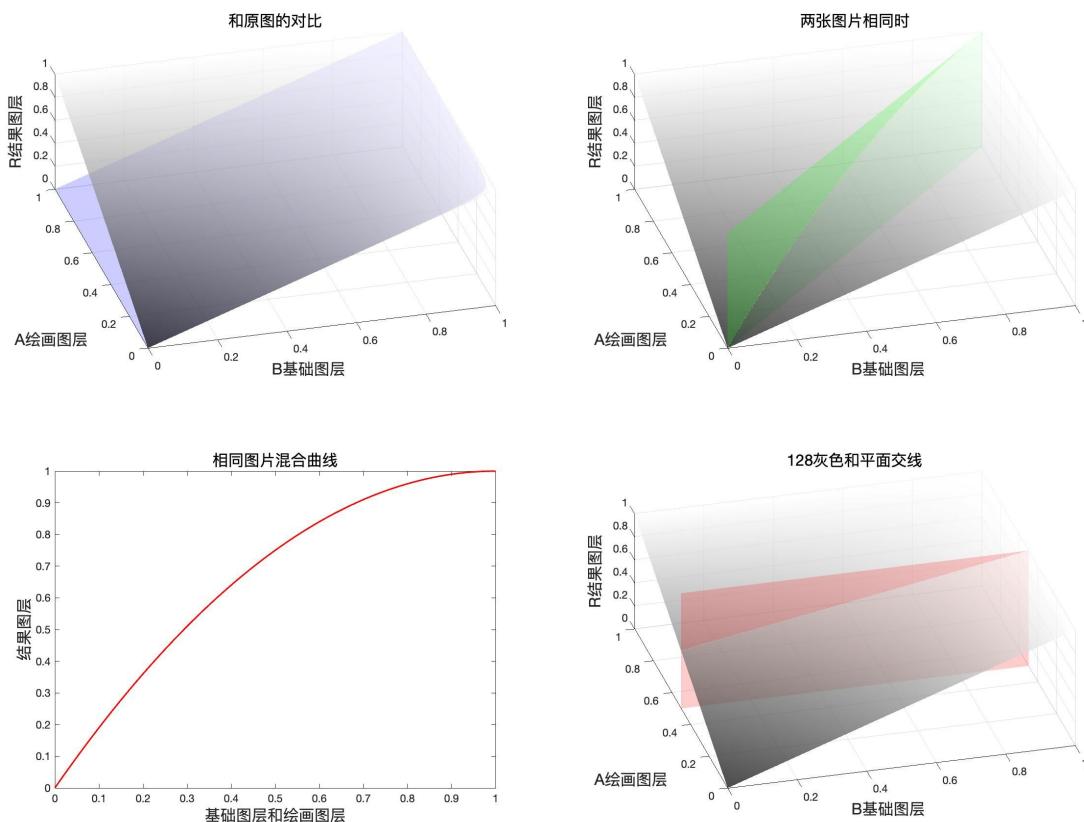
融合不透明度

$\$r=Opacity(b,a)=op\times Fill(b,a)+(1-op)\times b\$\$$

三次负片操作相互转换

```
 $$\begin{aligned} r &= Screen(b,a) \\ &= 1 - (1-b)(1-a) \\ &= 1 - \text{Multiply}(1-b, 1-a) \end{aligned}$$
```

映射面和同图等效曲线



程序模拟该模式计算结果

```
// 滤色  
public static BlendColor Screen(BlendColor colorBase, BlendColor  
colorBlend, double fill, double opacity) {
```

```

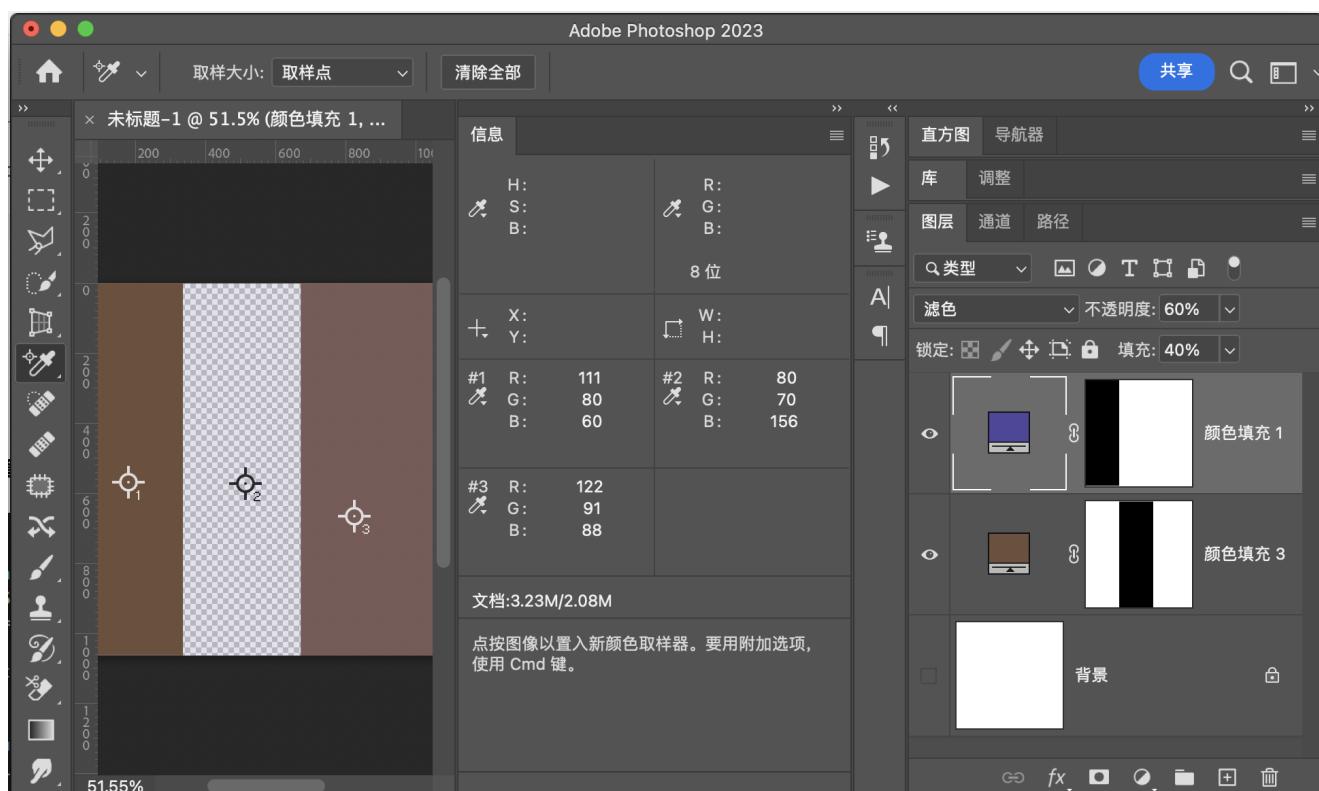
        double red = ScreenChannel(colorBase.red.get01Value(),
colorBlend.red.get01Value(), fill);
        double green = ScreenChannel(colorBase.green.get01Value(),
colorBlend.green.get01Value(), fill);
        double blue = ScreenChannel(colorBase.blue.get01Value(),
colorBlend.blue.get01Value(), fill);
        return ColorUtils.Opacity(colorBase, new BlendColor(red * 255,
green * 255, blue * 255), opacity);
    }

    private static double ScreenChannel(double base, double blend,
double fill) {
    return (1 - (1 - base) * (1 - blend)) * fill + (1 - fill) *
base;
}

```

滤色(Screen) RGB[121.84, 91.53, 88.63]~ HSY[5.24, 33.21, 100.30]~ HSB[5.24, 27.26, 47.78]

验证



用途示例

1:组成叠加模式和强光模式

2:扣除黑色背景

线性减淡LinearDodge

线性减淡就是让原图和混合图层做加法
他和线性加深也是可以互相转化的

公式

$$r = \text{LinearDodge}(b, a) = b + a$$

融合填充

$$r = \text{Fill}(b, a) = b + a \times \text{fill}$$

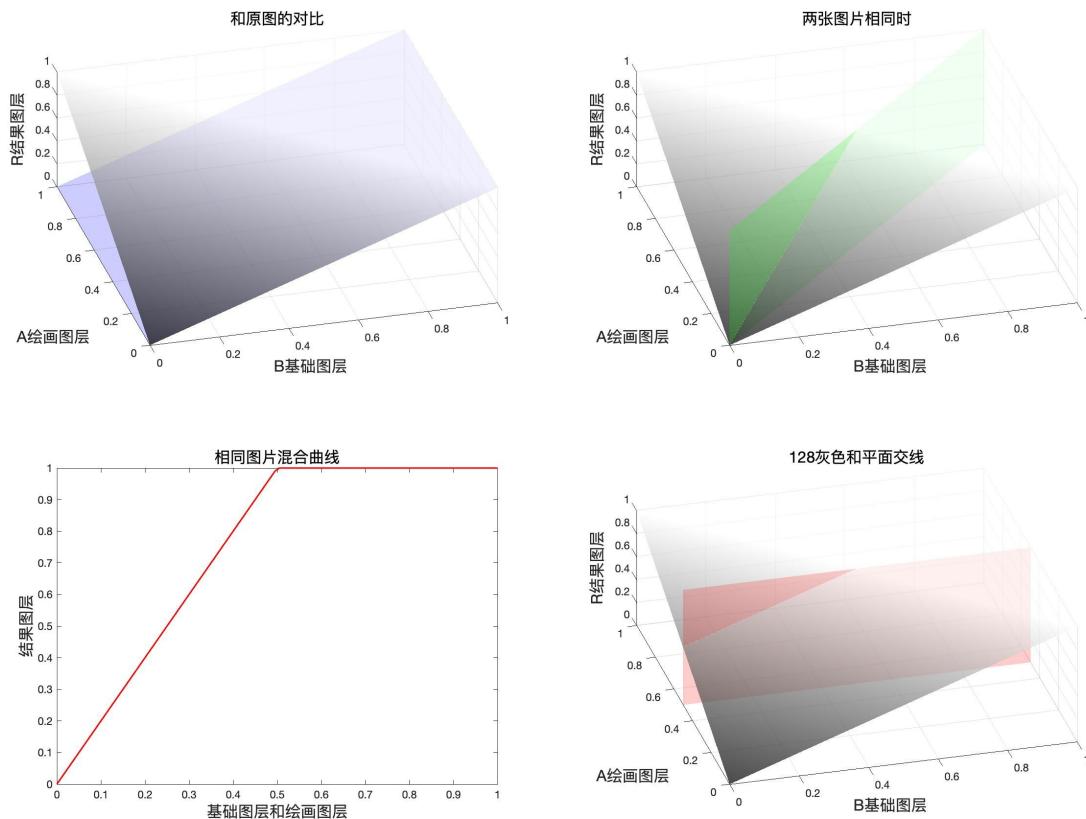
融合不透明度

$$r = \text{Opacity}(b, a) = \text{op} \times \text{Fill}(b, a) + (1 - \text{op}) \times b$$

三次负片操作相互转换

$$\begin{aligned} r &= \text{LinearDodge}(b, a) = 1 - \text{LinearBurn}(1-b, 1-a) = 1 - (1-b+1-a-1) = b+a \\ &\end{aligned}$$

映射面和同图等效曲线



程序模拟该模式计算结果

```

// 线性减淡
public static BlendColor LinearDodge(BlendColor colorBase,
BlendColor colorBlend, double fill, double opacity) {

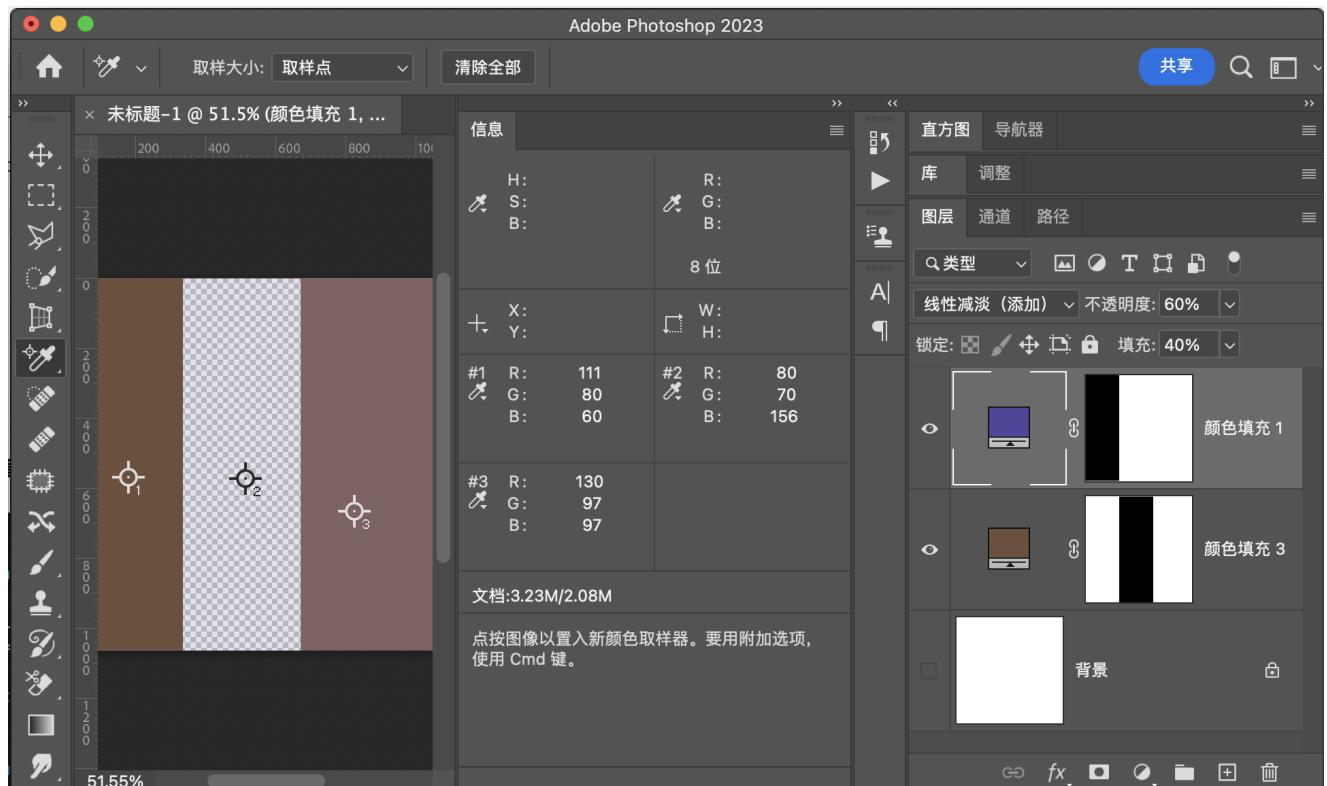
    double red = LinearDodgeChannel(colorBase.red.get01Value(),
colorBlend.red.get01Value(), fill);
    double green = LinearDodgeChannel(colorBase.green.get01Value(),
colorBlend.green.get01Value(), fill);
    double blue = LinearDodgeChannel(colorBase.blue.get01Value(),
colorBlend.blue.get01Value(), fill);
    return ColorUtils.Opacity(colorBase, new BlendColor(red * 255,
green * 255, blue * 255), opacity);
}

private static double LinearDodgeChannel(double base, double blend,
double fill) {
    return ColorUtils.round(base + blend * fill, 1, 0);
}

```

线性减淡(LinearDodge) RGB[130.20, 96.80, 97.44]~ HSY[358.85, 33.40, 106.89]~ HSB[358.85, 25.65, 51.06]

验证



用途示例

- 1: 组成线性光
- 2: 制作特殊光效

颜色减淡ColorDodge

颜色减淡可以由颜色加深通过三次负片转化，颜色减淡也可以转化为划分，通过一次负片可以实现

公式

$$r = \text{ColorDodge}(b, a) = \frac{b}{1-a}$$

融合填充

$$r = \text{Fill}(b, a) = \frac{b}{1-a} \times \text{fill}$$

融合不透明度

$$r = \text{Opacity}(b, a) = \text{op} \times \text{Fill}(b, a) + (1 - \text{op}) \times b$$

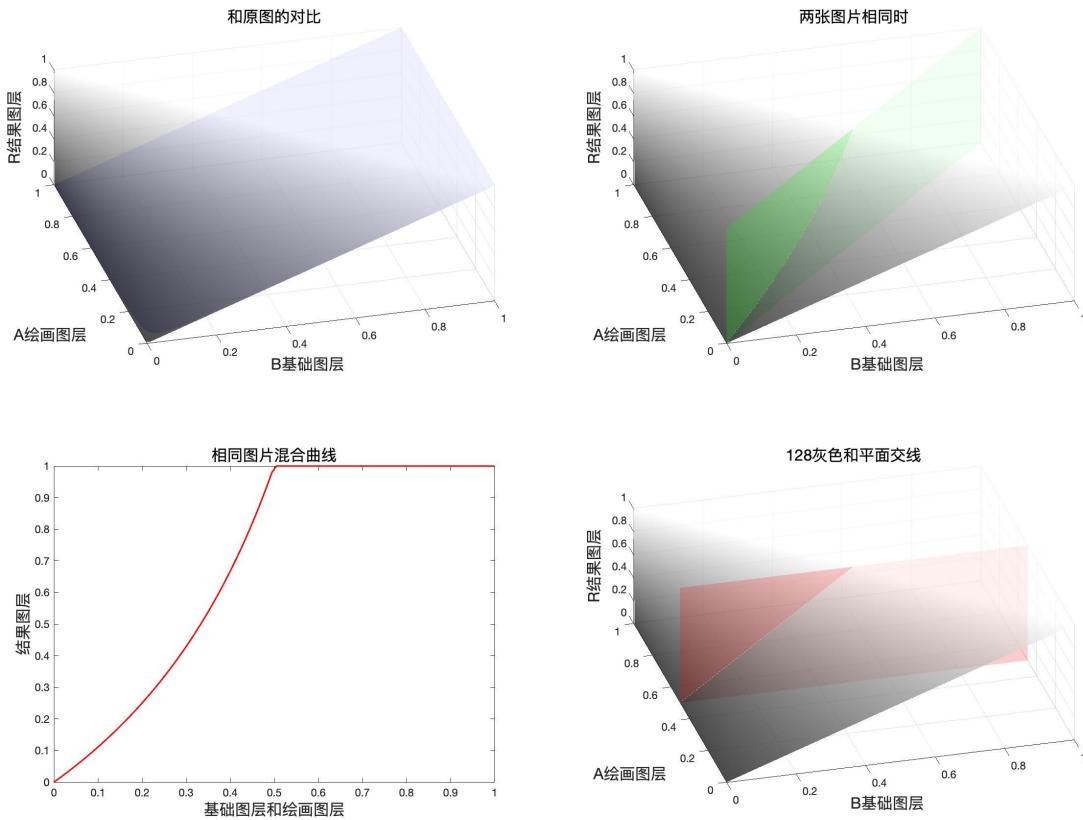
三次负片操作相互转换

$$\begin{aligned} r &= \text{ColorDodge}(b, a) \\ &= 1 - \text{ColorBurn}(1-b, 1-a) \\ &= 1 - (1 - \frac{1-(1-b)}{1-(1-a)}) \\ &= \frac{b}{1-a} \end{aligned}$$

一次负片转为划分

$$\begin{aligned} r &= \text{ColorDodge}(b, 1-a) \\ &= \frac{b}{1-1+a} \\ &= \frac{b}{a} \end{aligned}$$

映射面和同图等效曲线



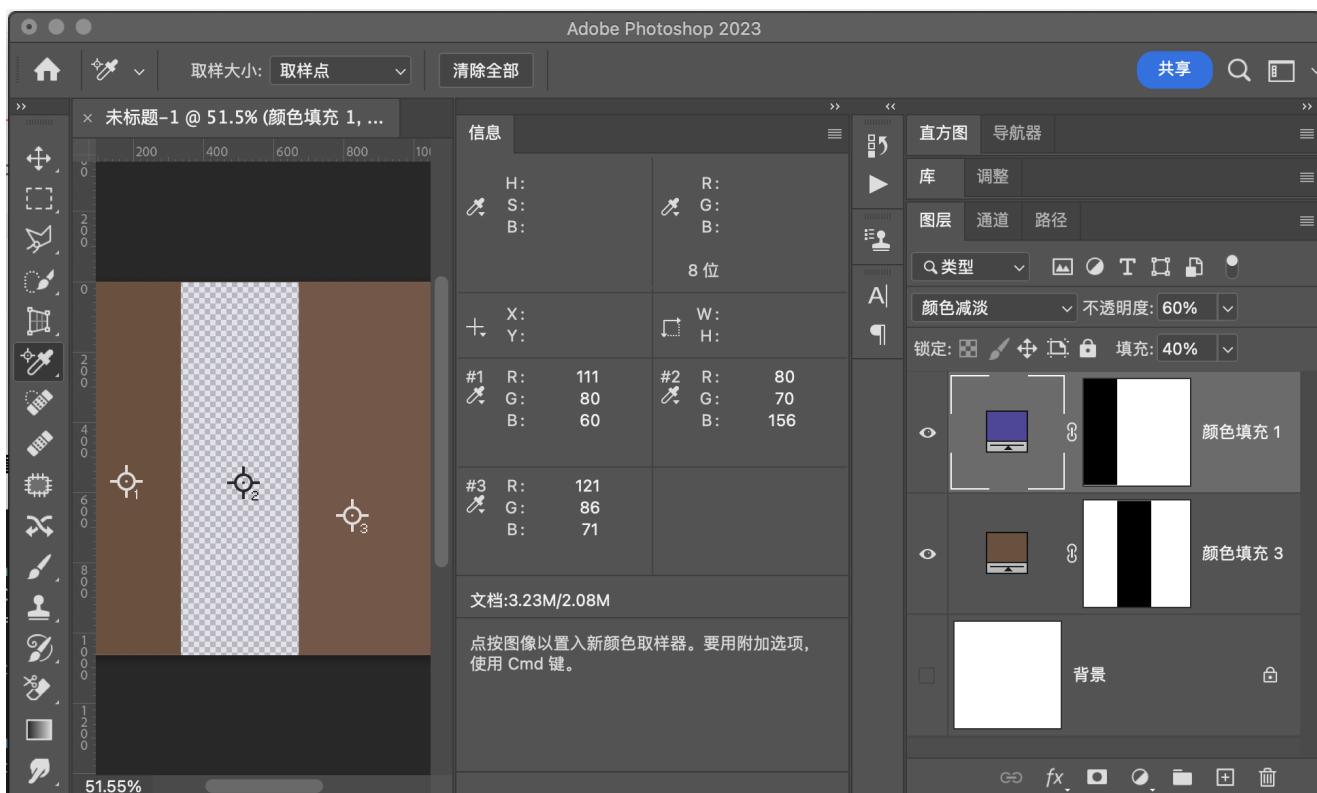
程序模拟该模式计算结果

```
// 颜色减淡
public static BlendColor ColorDodge(BlendColor colorBase, BlendColor
colorBlend, double fill, double opacity) {
    double red = ColorDodgeChannel(colorBase.red.get01Value(),
colorBlend.red.get01Value(), fill);
    double green = ColorDodgeChannel(colorBase.green.get01Value(),
colorBlend.green.get01Value(), fill);
    double blue = ColorDodgeChannel(colorBase.blue.get01Value(),
colorBlend.blue.get01Value(), fill);
    return ColorUtils.Opacity(colorBase, new BlendColor(red * 255,
green * 255, blue * 255), opacity);
}

private static double ColorDodgeChannel(double base, double blend,
double fill) {
    return ColorUtils.round((base) / ((1 - blend * fill)), 1, 0);
}
```

颜色减淡(ColorDodge) RGB[120.56, 85.92, 71.66] ~ HSY[17.50, 48.89, 94.74] ~ HSB[17.50, 40.56, 47.28]

验证



用途示例

1:组合成为亮光模式

2:制造光线,同时保护暗部细节

浅色Lighter

浅色模式是深色模式的相反模式，可以通过深色三次负片操作得到

计算细节简单来说就是求和，比较大小，大的留下了，若求和的结果一样，就计算明度，明度大的留下了。

公式

```
 $$\begin{aligned}
 \text{Pix\_r} &= \text{Lighter}(\text{Pix\_b}, \text{Pix\_a}) \\
 &= \left\{ \begin{aligned}
 &\text{Pix\_a} \&& \text{Sum}(\text{Pix\_a}) > \text{Sum}(\text{Pix\_b}) \&& \text{Pix\_b} \&& \\
 &\text{Sum}(\text{Pix\_a}) < \text{Sum}(\text{Pix\_b}) \&& \text{Sum}(\text{Pix\_a}) = \text{Sum}(\text{Pix\_b}) \&& \text{Lum}(\text{Pix\_b}) > \text{Lum}(\text{Pix\_a}) \&& \\
 &\text{Sum}(\text{Pix\_a}) = \text{Sum}(\text{Pix\_b}) \&& \text{Lum}(\text{Pix\_b}) < \text{Lum}(\text{Pix\_a}) \end{aligned} \right. \\
 &\end{aligned} $$
 $$\text{Lum}(\text{Pix}) = 0.3\text{R} + 0.59\text{G} + 0.11\text{B}$$
 $$\text{Sum}(\text{Pix}) = \text{R} + \text{G} + \text{B}$$
```

融合填充

\$\$\text{r} = \text{Fill}(\text{Pix_b}, \text{Pix_a} \times \text{fill}) = \text{fill} \times \text{Pix_r} + \text{Pix_b}\$\$

融合不透明度

\$\$r=Opacity(Pix_b,Pix_a)=op\times Fill(Pix_b,Pix_a)+(1-op)\times Pix_b\$\$

三次负片操作相互转换

\$\$\begin{aligned} & \text{begin}\{\text{aligned}\} \\ & \quad \text{Pix_r} = \text{Lighter}(\text{Pix_b}, \text{Pix_a}) \&= \left(\begin{aligned} & \text{Pixel}_b \&\& \text{当} \\ & \text{a}_{rc} + a_{gc} + a_{bc} < b_{rc} + b_{gc} + b_{bc} \&\& \text{当} \\ & \text{a}_{rc} + a_{gc} + a_{bc} > b_{rc} + b_{gc} + b_{bc} \end{aligned} \right) \text{right.} \\ & \quad \&= 1 - \text{Darker}(1 - \text{Pixel}_b, 1 - \text{Pixel}_a) \\ & \&= \left(\begin{aligned} & 1 - (\text{Pixel}_b + \text{Pixel}_a) \&\& \text{当} \\ & 1 - \text{a}_{rc} - \text{a}_{gc} - \text{a}_{bc} > 1 - \text{b}_{rc} - \text{b}_{gc} - \text{b}_{bc} \end{aligned} \right) \text{right.} \\ & \quad \&= \left(\begin{aligned} & \text{Pixel}_b + \text{Pixel}_a < \text{b}_{rc} + \text{b}_{gc} + \text{b}_{bc} \&\& \text{当} \\ & \text{a}_{rc} + \text{a}_{gc} + \text{a}_{bc} > \text{b}_{rc} + \text{b}_{gc} + \text{b}_{bc} \end{aligned} \right) \text{right.} \end{aligned} \} \text{end}\{\text{aligned}\} \\ & \text{相等的情况同理可得} \end{aligned}

程序模拟该模式计算结果

```
// 浅色
public static BlendColor Lighter(BlendColor colorBase, BlendColor
colorBlend, double fill, double opacity) {
    double sumBase = colorBase.red.value + colorBase.green.value +
colorBase.blue.value;
    double sumBlend = (colorBlend.red.value + colorBlend.green.value +
colorBlend.blue.value);
    if (sumBase == sumBlend) {
        if (colorBase.getLum() > colorBlend.getLum()) {
            double red = colorBase.red.get01Value();
            double green = colorBase.green.get01Value();
            double blue = colorBase.blue.get01Value();
            return ColorUtils.Opacity(colorBase, new BlendColor(red *
255, green * 255, blue * 255), opacity);
        } else {
            double red = colorBase.red.get01Value() * (1 - fill) +
colorBlend.red.get01Value() * fill;
            double green = colorBase.green.get01Value() * (1 - fill) +
colorBlend.green.get01Value() * fill;
            double blue = colorBase.blue.get01Value() * (1 - fill) +
colorBlend.blue.get01Value() * fill;
            return ColorUtils.Opacity(colorBase, new BlendColor(red *
255, green * 255, blue * 255), opacity);
        }
    }
    if (sumBase > sumBlend) {
        double red = colorBase.red.get01Value();
        double green = colorBase.green.get01Value();
        double blue = colorBase.blue.get01Value();
        return ColorUtils.Opacity(colorBase, new BlendColor(red *
255, green * 255, blue * 255), opacity);
    }
    double red = colorBase.red.get01Value() * (1 - fill) +
colorBlend.red.get01Value() * fill;
```

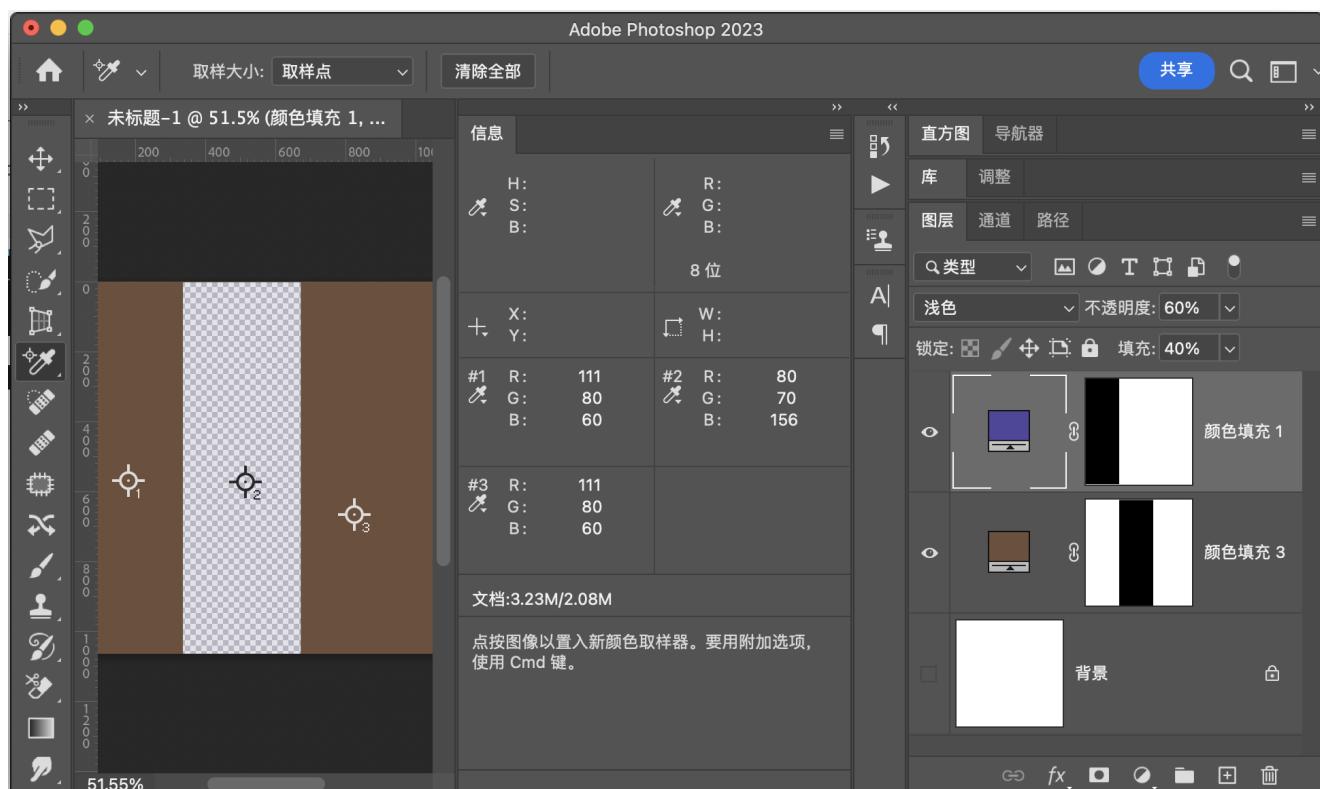
```

        double green = colorBase.green.get01Value() * (1 - fill) +
colorBlend.green.get01Value() * fill;
        double blue = colorBase.blue.get01Value() * (1 - fill) +
colorBlend.blue.get01Value() * fill;
        return ColorUtils.Opacity(colorBase, new BlendColor(red * 255,
green * 255, blue * 255), opacity);
    }
}

```

浅 色(Lighter) RGB[111.00, 80.00, 60.00]~ HSY[23.53, 51.00, 87.10]~ HSB[23.53, 45.95, 43.53]

验证



用途示例

保留像素中通道和较大的像素，可用于将背景较暗或相似但是部分不相同的图片融合。相当于做了一个复杂的通道蒙版，并且这个蒙版非常精确，精确到像素。

对比度组

对比度组都是上述两组某两项的组合，组合的分割界限是基础图层或者混合图层中性灰平面，并且他们之间也有一些关系。

对比度组

叠加Overlay

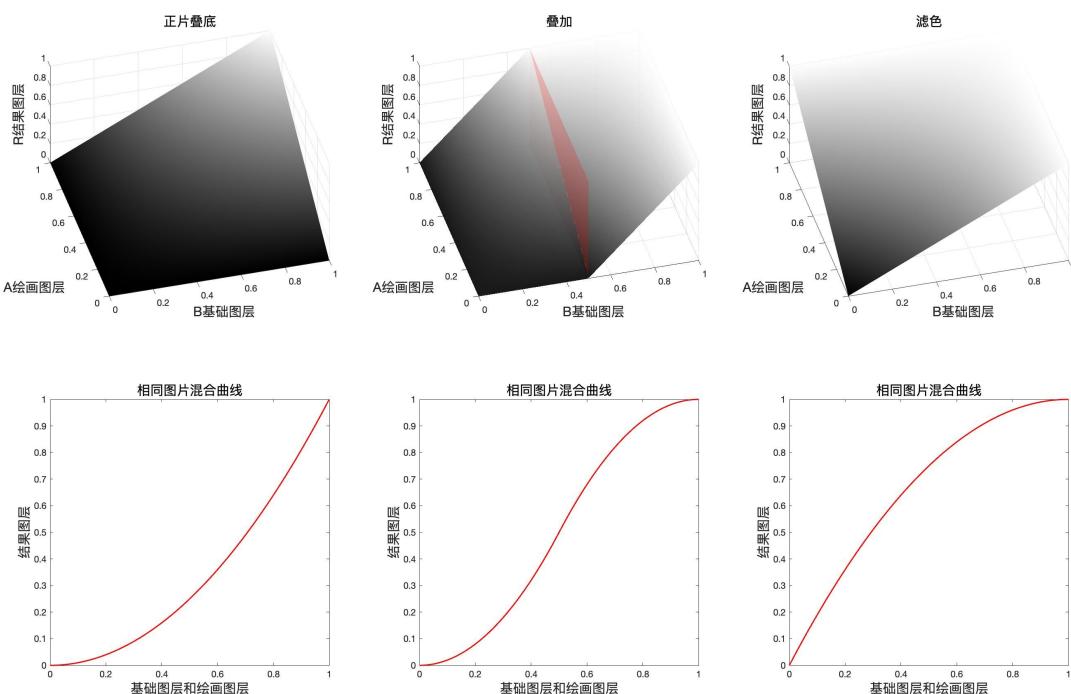
此模式是正片叠底和滤色的组合，组合依据是底图的中性灰平面，如果在 $[0,128]$ 则使用正片叠底，若是在 $(128,255]$ 之间，则是实用滤色。

公式

$$\begin{aligned} \text{OverLay}(b,a) &= \left\{ \begin{aligned} &\text{Multiply}(2b,a) \quad \text{当 } 0 \leq b \leq 0.5 \\ &\text{Screen}(2(b-0.5),a) \quad \text{当 } 0.5 < b \leq 1 \end{aligned} \right. \\ &= \left\{ \begin{aligned} &2ba \quad \text{当 } 0 \leq b \leq 0.5 \\ &1 - 2(1-b)(1-a) \quad \text{当 } 0.5 < b \leq 1 \end{aligned} \right. \end{aligned}$$

和填充还有不透明度的关系，以及公式可以参考正片叠底和滤色

映射面和同图等效曲线



程序模拟该模式计算结果

```
// 叠加
public static BlendColor Overlay(BlendColor colorBase, BlendColor
colorBlend, double fill, double opacity) {
    double red = OverlayChannel(colorBase.red.get01Value(),
colorBlend.red.get01Value(), fill);
    double green = OverlayChannel(colorBase.green.get01Value(),
colorBlend.green.get01Value(), fill);
    double blue = OverlayChannel(colorBase.blue.get01Value(),
colorBlend.blue.get01Value(), fill);
    return ColorUtils.Opacity(colorBase, new BlendColor(red *255, green*
255, blue * 255), opacity);
}

private static double OverlayChannel(double baseValue, double
```

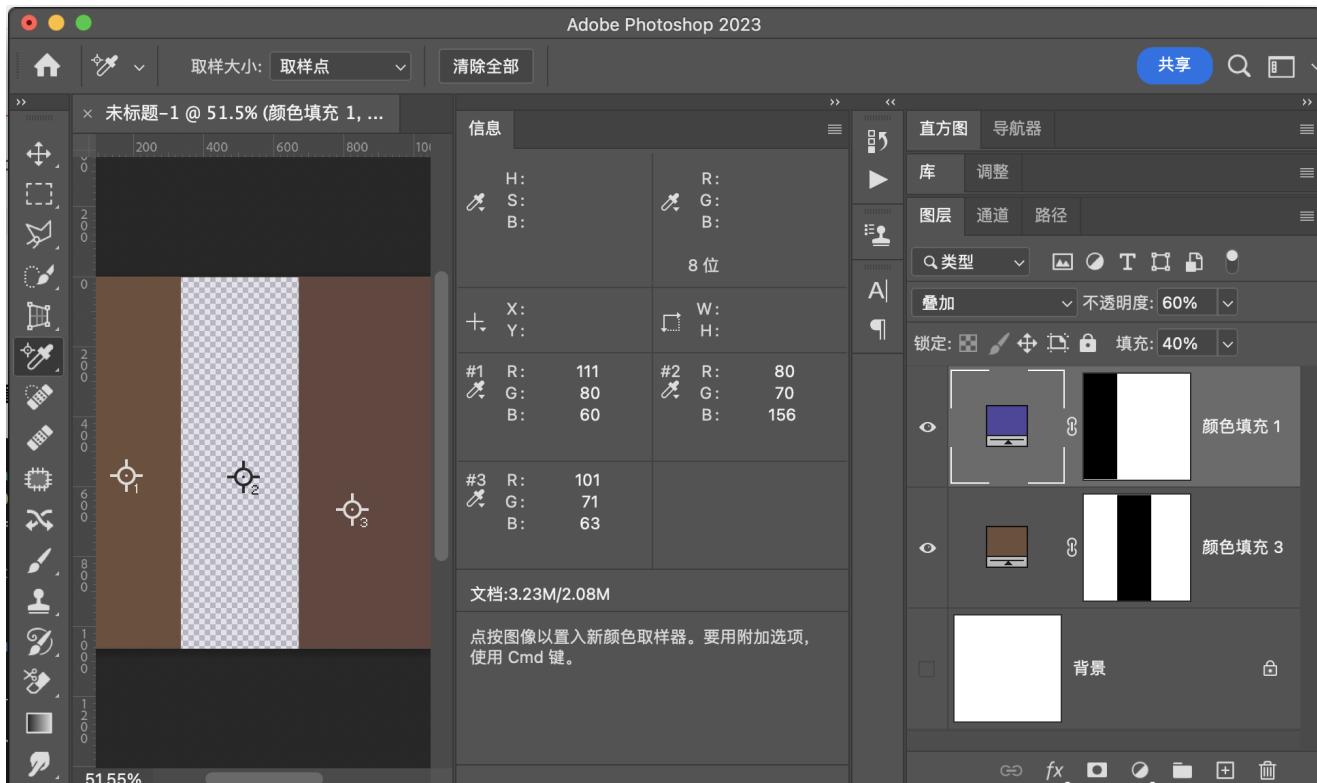
```

blendValue, double fill) {
    if (baseValue <= 0.5) {
        return MulitplyChannel(baseValue, 2 * blendValue, fill);
    } else {
        return ScreenChannel(baseValue, 2 * (blendValue - 0.5), fill);
    }
}

```

叠 加(Overlay) RGB [101.08, 71.34, 63.22] ~ HSY [12.87, 37.86, 79.37] ~ HSB [12.87, 37.45, 39.64]

验证



用途示例

- 1: 同图混合增加图片对比度
- 2: 和中性灰混合，达到局部提亮或者压黑

柔光SoftLight

柔光模式是最复杂的一种混合模式，也是最巧妙的一种混合模式，柔光模式的本质是伽马矫正(gamma correction)。配合混合图层的像素点的通道数值，再对原图层使用伽马矫正，二者通过配合就可以得到柔光模式。

如果我们想了解柔光模式，首先必须了解什么是伽马矫正，伽马矫正，简单来说就是，将原像素通道数值通过幂次方的方式进行修改，比如平方和根号，例如我们由一个归一化之后为0.5的通道数值，我们对其

进行系数为2的伽马矫正，则结果是 $0.5^{\frac{1}{2}} = \sqrt{0.5}$ ，如果进行系数为 $\frac{1}{2}$ 的伽马矫正，则结果为 0.5^2

公式

$$\begin{aligned} r &= \text{SoftLight}(b, a) \\ &\Leftarrow \left(\begin{aligned} &(2a-1)(b^2-b)+b \leq 0.5 \wedge (2a-1) \\ &(\sqrt{b}-b)+b > 0.5 \end{aligned} \right) \end{aligned}$$

伽马矫正的数学表达式 $\text{output} = \text{input}^{\frac{1}{gama}}$

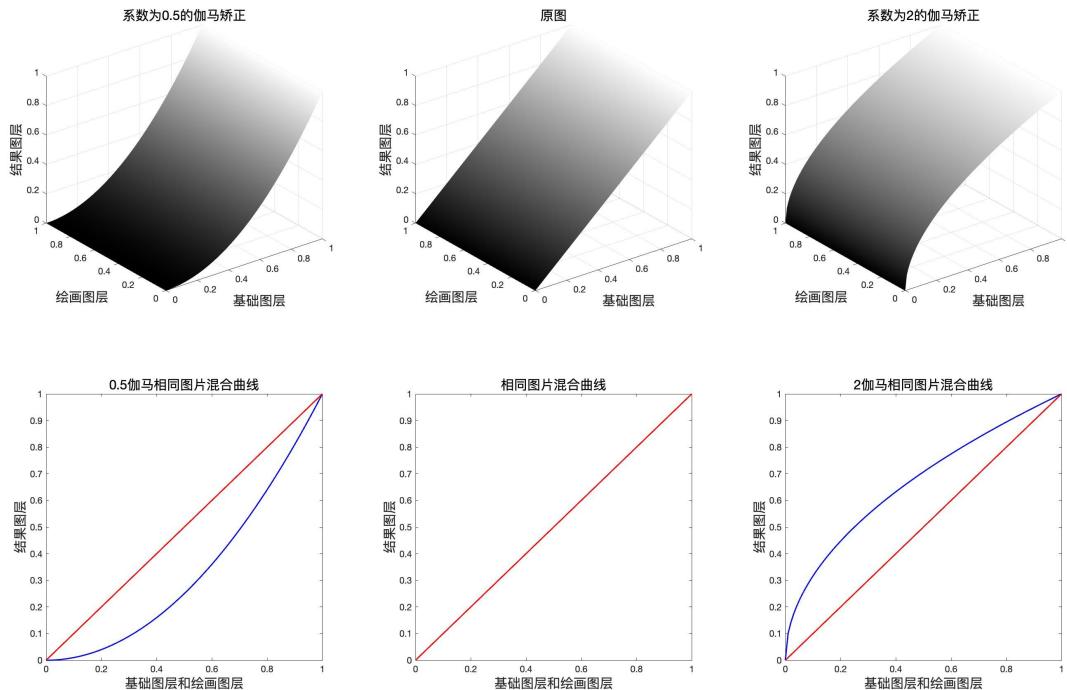
其中 input 代表输入信号， output 代表输出， $gama$ 代表伽马系数

系数为 2 的伽马矫正

$$\text{output} = \text{input}^{\frac{1}{2}}$$

系数为 $\frac{1}{2}$ 的伽马矫正

$$\text{output} = \text{input}^2$$

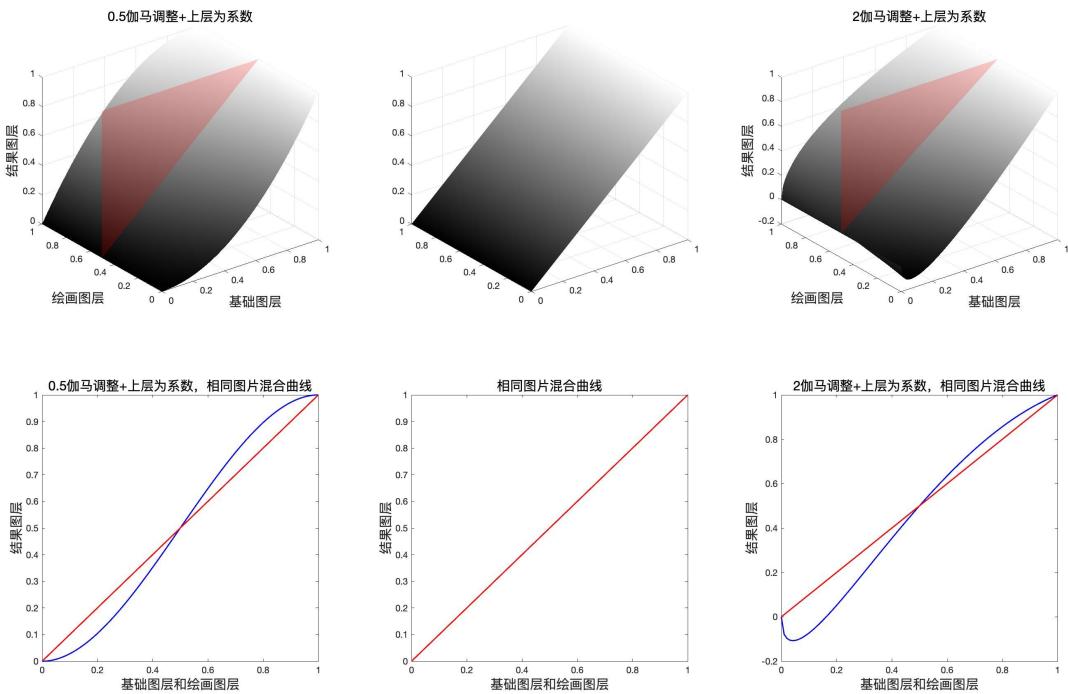


于是对于系数为 $\frac{1}{2}$ 的伽马矫正，稍微变换一下表达式

$$r = b^2 - (b - b^2) + b$$

然后再使用 $(2a-1)$ 作为系数乘以差值项

$$r = (2a-1)(b - b^2) + b$$



于是对于系数为\$2\$的伽马矫正，稍微变换一下表达式

$$\$\$r = \sqrt{b} = (\sqrt{b} - b) + b\$\$$$

然后再使用\$(2a-1)\$作为系数乘以差值项

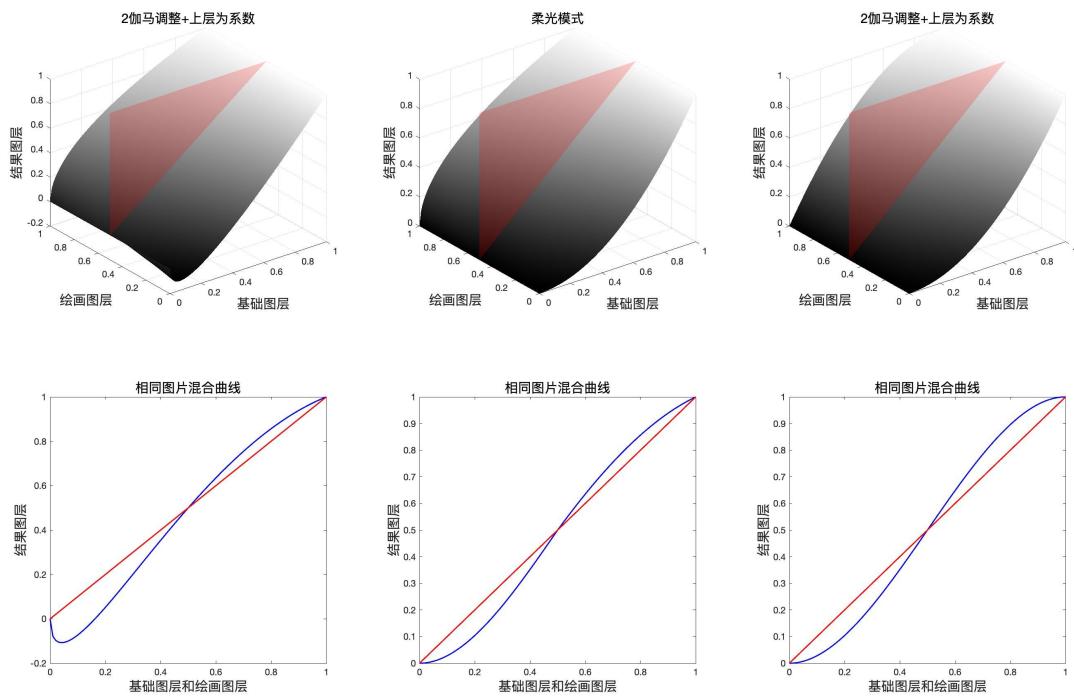
$$\$\$r = (2a-1)(\sqrt{b} - b) + b\$\$$$

再将结果合并，我们就可以得到柔光模式的表达式。

如果使用一句话概括柔光模式的数学表达式，就是“以混合图层为系数的系数为\$\frac{1}{2}\$和\$2\$的伽马矫正”

在 PS 中伽马矫正可以在色阶工具和曝光度工具中找到

映射面和同图等效曲线



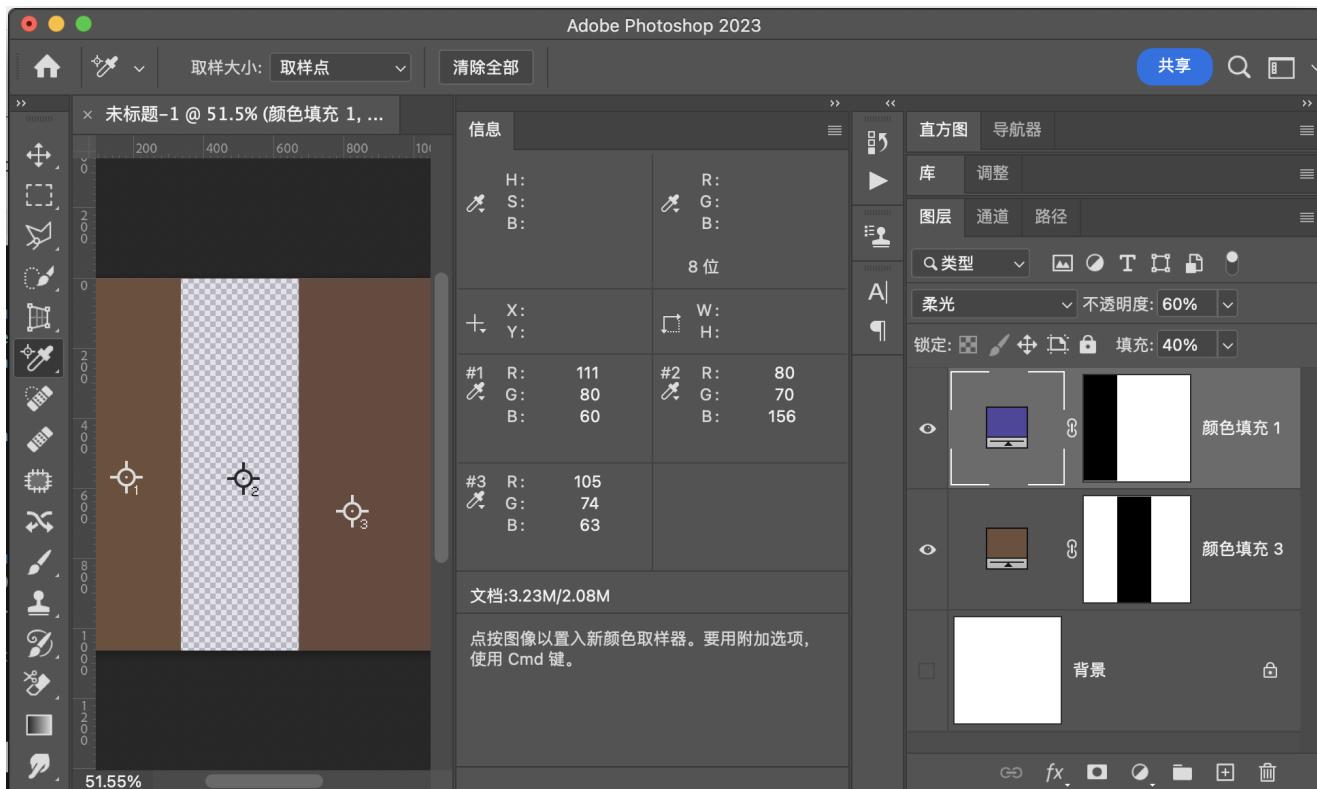
程序模拟该模式计算结果

```
// 柔光
public static BlendColor SoftLight(BlendColor colorBase, BlendColor
colorBlend, double fill, double opacity) {
    double red = SoftLightChannel(colorBase.red.get01Value(),
colorBlend.red.get01Value(), fill);
    double green = SoftLightChannel(colorBase.green.get01Value(),
colorBlend.green.get01Value(), fill);
    double blue = SoftLightChannel(colorBase.blue.get01Value(),
colorBlend.blue.get01Value(), fill);
    return ColorUtils.Opacity(colorBase, new BlendColor(red * 255, green
* 255, blue * 255), opacity);
}

private static double SoftLightChannel(double baseValue, double
blendValue, double fill) {
    if (blendValue <= 0.5) {
        return (baseValue + (2 * blendValue - 1) * (baseValue -
baseValue * baseValue)) * fill
                + (1 - fill) * baseValue;
    } else {
        return (baseValue + (2 * blendValue - 1) * (Math.sqrt(baseValue)
- baseValue)) * fill
                + (1 - fill) * baseValue;
    }
}
```

柔光(SoftLight) RGB [105.40, 74.06, 63.42] ~ HSY [15.21, 41.98, 82.29] ~ HSB [15.21, 39.83, 41.33]

验证



用途示例

- 1: 同图混合增加图片对比度
- 2: 配合中性灰平面，实现局部提亮和压暗 (dodge and burn)

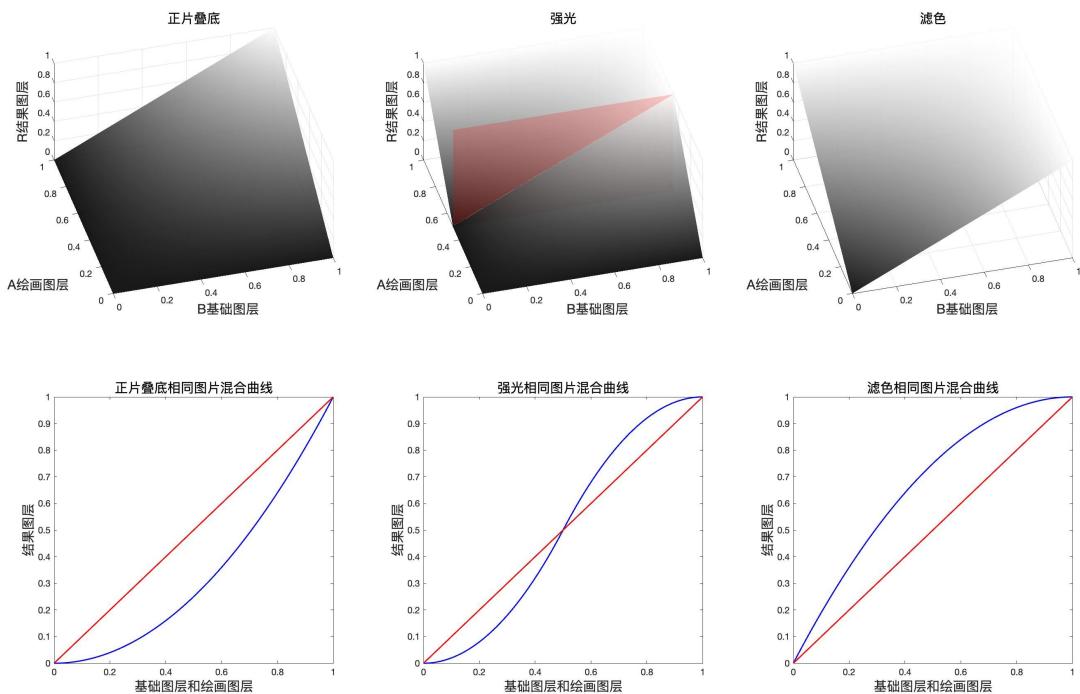
强光HardLight

此模式也是正片叠底和滤色的组合，组合的分割界限是混合图层中性灰平面，并且它和叠加模式是互逆的关系，也就是说，如果在强光模式下把基础图层和混合图层顺序调换，可以得到原顺序下叠加模式的效果

公式

$$\begin{aligned} r &= \text{HardLight}(b, a) \\ &= \begin{cases} \text{Multiply}(b, 2a) & \text{当 } 0 \leq a \leq 0.5 \\ \text{Screen}(b, 2(a-0.5)) & \text{当 } 0.5 < a \leq 1 \end{cases} \\ &= \begin{cases} 2ba & \text{当 } 0 \leq a \leq 0.5 \\ 1-2(1-b)(1-a) & \text{当 } 0.5 < a \leq 1 \end{cases} \end{aligned}$$

映射面和同图等效曲线



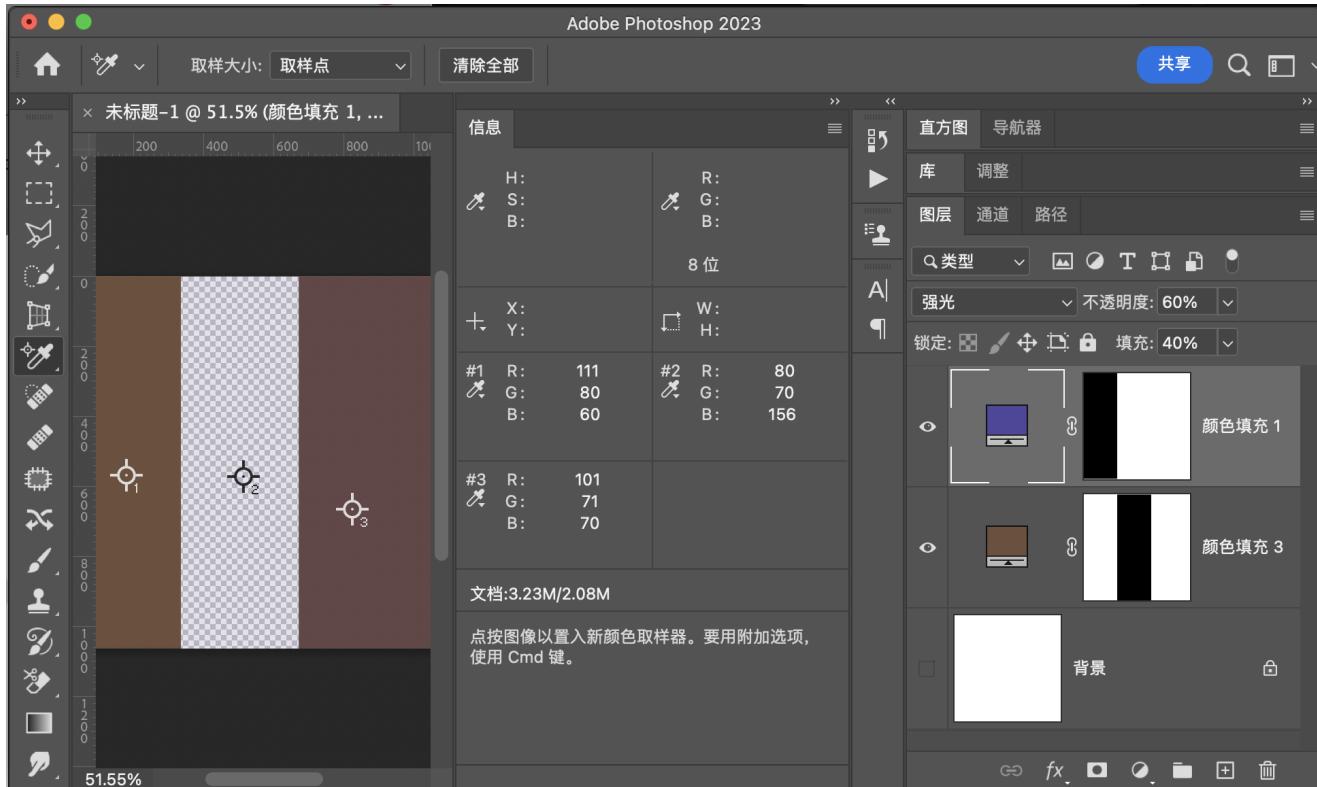
程序模拟该模式计算结果

```
// 强光
public static BlendColor HardLight(BlendColor colorBase, BlendColor
colorBlend, double fill, double opacity) {
    double red = HardLightChannel(colorBase.red.get01Value(),
colorBlend.red.get01Value(), fill);
    double green = HardLightChannel(colorBase.green.get01Value(),
colorBlend.green.get01Value(), fill);
    double blue = HardLightChannel(colorBase.blue.get01Value(),
colorBlend.blue.get01Value(), fill);
    return ColorUtils.Opacity(colorBase, new BlendColor(red * 255,
green * 255, blue * 255), opacity);
}

private static double HardLightChannel(double baseValue, double
blendValue, double fill) {
    if (blendValue <= 0.5) {
        return MulitplyChannel(baseValue, 2 * blendValue, fill);
    } else {
        return ScreenChannel(baseValue, 2 * (blendValue - 0.5),
fill);
    }
}
```

强光(HardLight) RGB [101.08, 71.34, 70.46] ~ HSY [1.72, 30.61, 80.16] ~ HSB [1.72, 30.29, 39.64]

验证



用途示例

1: 同图混合增加图片对比度

2: 配合中性灰平面, 实现dodge and burn.

线性光LinearLight

线性光是线性减淡和线性加深的结合, 组合的分割界限是混合图层中性灰平面, 具体公式如下

公式

$$\begin{aligned} \text{r} &= \text{LinearLight}(b, a) \\ &\& \left\{ \begin{aligned} &\text{当 } 0 \leq a \leq 0.5 && \text{LinearBurn}(b, 2a) \\ &\& \text{当 } 0.5 < a \leq 1 && \text{LinearDodge}(b, 2(a-0.5)) \end{aligned} \right. \\ &\& \left. \begin{aligned} &\& \& \& b + (2a - 1) \\ &\& \& \& \end{aligned} \right. \\ &\& \left. \begin{aligned} &\& \& \& b + 2a - 1 \end{aligned} \right. \end{aligned}$$

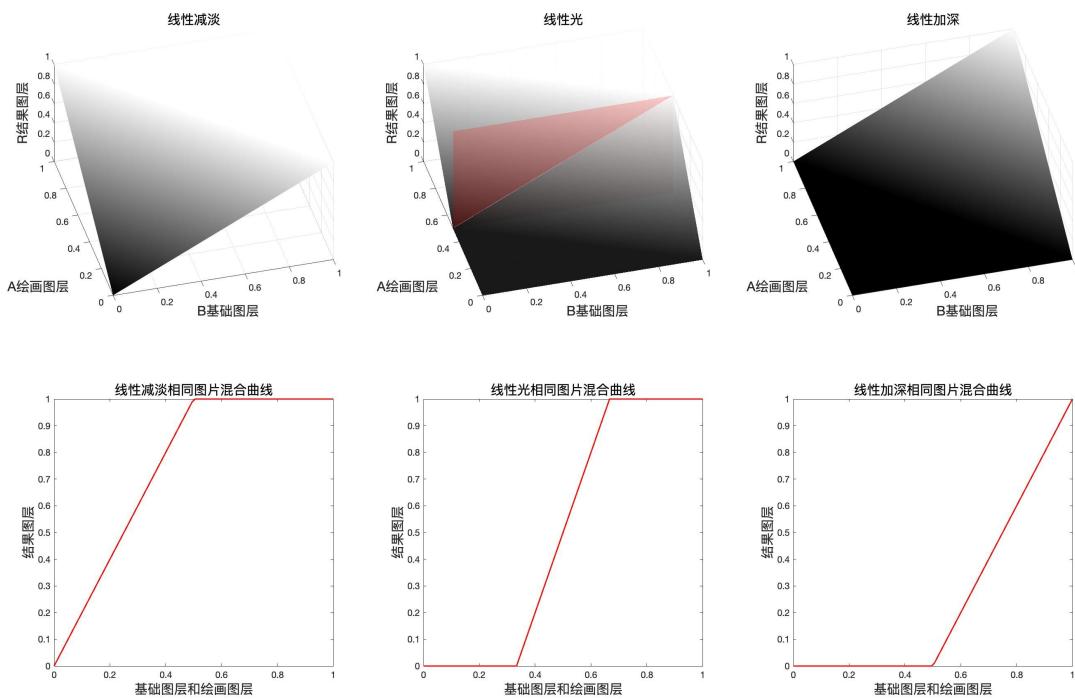
和填充结合

$$\begin{aligned} \text{r} &= \text{Fill}(b, a) \\ &\& \left\{ \begin{aligned} &\text{当 } 0 \leq a \leq 0.5 && \text{LinearBurn}(b, 2a \times \text{fill}) \\ &\& \text{当 } 0.5 < a \leq 1 && \text{LinearDodge}(b, 2(a-0.5) \times \text{fill}) \\ &\& \& \& b + (2a - 1) \times \text{fill} \\ &\& \& \& \end{aligned} \right. \\ &\& \left. \begin{aligned} &\& \& \& b + 2a \times \text{fill} - 1 \\ &\& \& \& \end{aligned} \right. \end{aligned}$$

融合不透明度

$\$r = \text{Opacity}(b, a) = op \times \text{Fill}(b, a) + (1 - op) \times b$

映射面和同图等效曲线



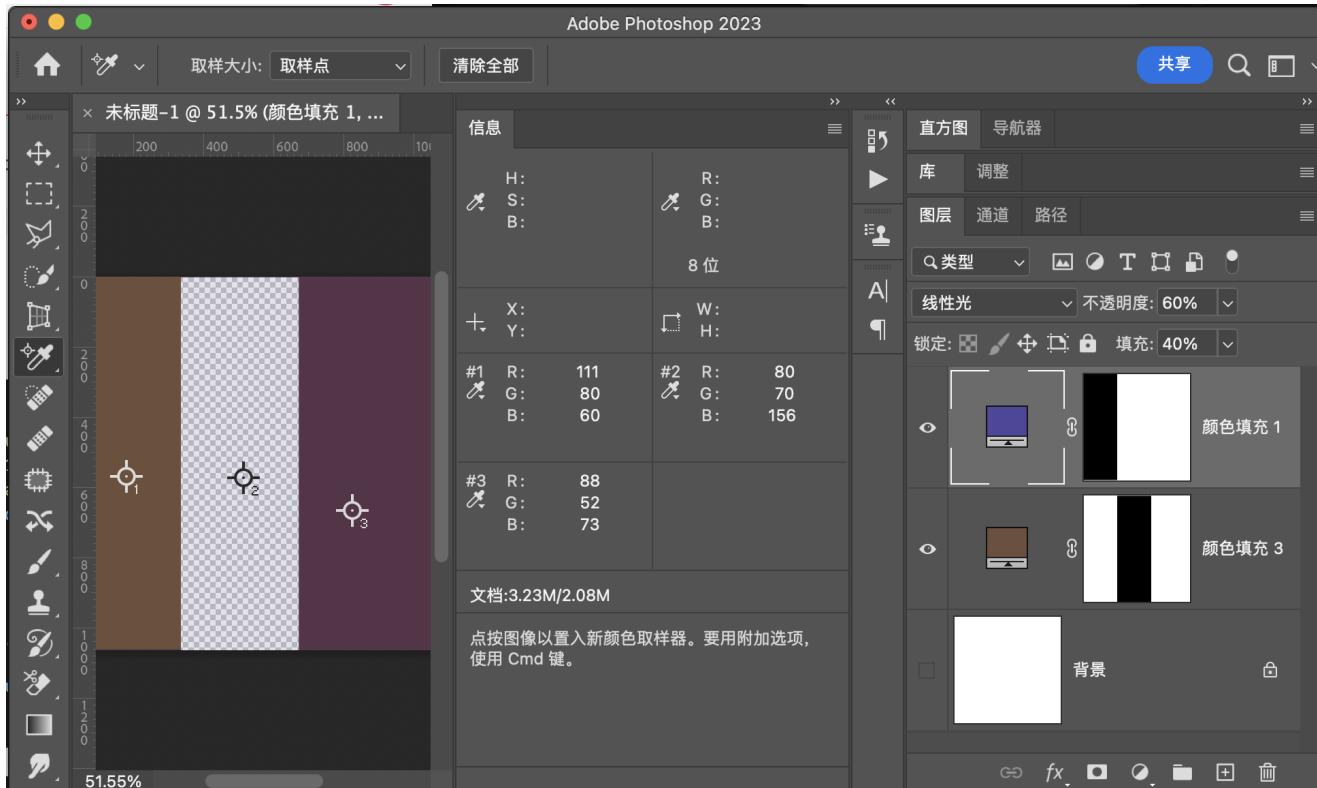
程序模拟该模式计算结果

```
public static BlendColor LinearLight(BlendColor colorBase, BlendColor colorBlend, double fill, double opacity) {
    double red = LinearLightChannel(colorBase.red.get01Value(),
colorBlend.red.get01Value(), fill);
    double green = LinearLightChannel(colorBase.green.get01Value(),
colorBlend.green.get01Value(), fill);
    double blue = LinearLightChannel(colorBase.blue.get01Value(),
colorBlend.blue.get01Value(), fill);
    return ColorUtils.Opacity(colorBase, new BlendColor(red * 255, green
* 255, blue * 255), opacity);
}

private static double LinearLightChannel(double baseValue, double
blendValue, double fill) {
    if (blendValue <= 0.5) {
        return LinearBurnChannel(baseValue, 2 * blendValue, fill);
    } else {
        return LinearDodgeChannel(baseValue, 2 * (blendValue - 0.5),
fill);
    }
}
```

线性光(LinearLight) RGB [88.20, 52.40, 73.68] ~ HSY [324.34, 35.80, 65.48] ~ HSB [324.34, 40.59, 34.59]

验证



用途示例

1: 组成中性灰磨皮

点光 PinLight

点光是变暗模式和变亮模式的组合

公式

$$\begin{aligned} r = \text{PinLight}(b, a) &= \left\{ \begin{aligned} &\text{Darken}(b, 2a) \quad &a \leq 0.5 \\ &\text{Lighten}(b, 2(a - 0.5)) \quad &a > 0.5 \end{aligned} \right. \\ &\&= \left\{ \begin{aligned} &\text{Min}(b, 2a) \quad &a \leq 0.5 \\ &\text{Max}(b, 2(a - 0.5)) \quad &a > 0.5 \end{aligned} \right. \end{aligned}$$

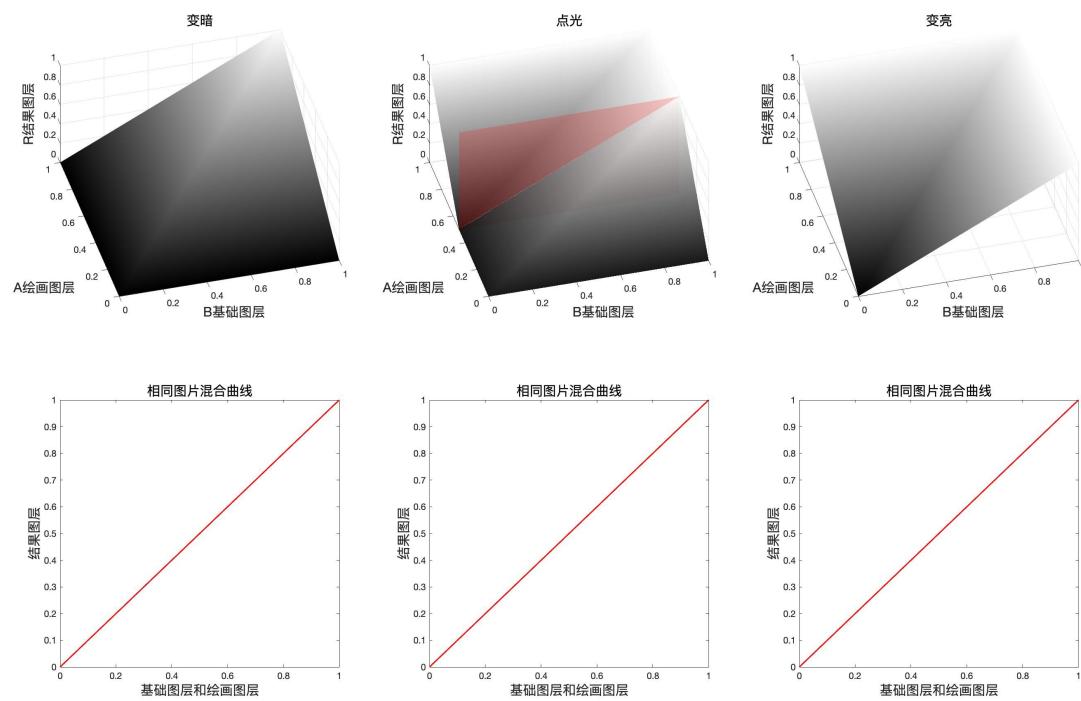
和填充结合

$$\begin{aligned} r = \text{Fill}(b, a) &= \left\{ \begin{aligned} &\text{Min}(b, 2a \times \text{fill}) \quad &a \leq 0.5 \\ &\text{Max}(b, 2(a - 0.5) \times \text{fill}) \quad &a > 0.5 \end{aligned} \right. \end{aligned}$$

融合不透明度

$$r = \text{Opacity}(b, a) = op \times \text{Fill}(b, a) + (1 - op) \times b$$

映射面和同图等效曲线



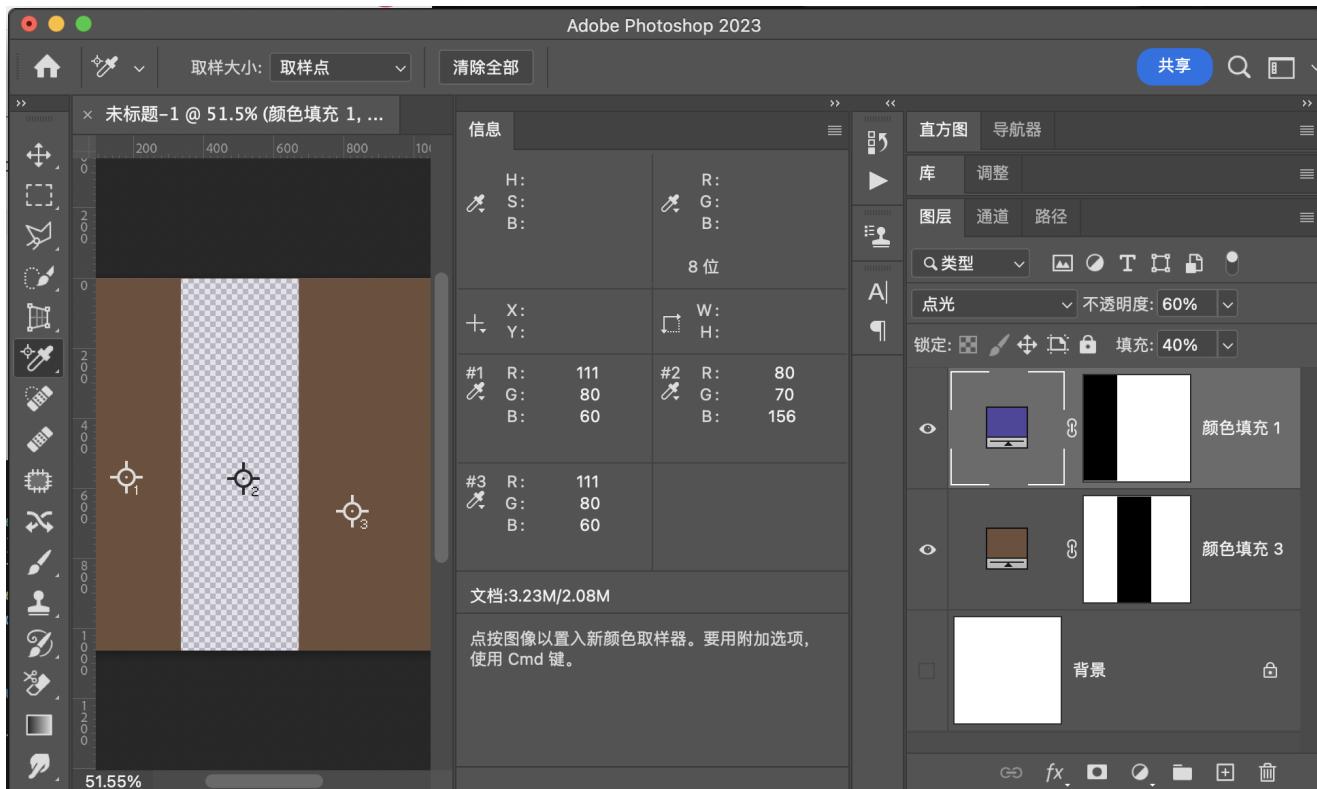
程序模拟该模式计算结果

```
// 点光
public static BlendColor PinLight(BlendColor colorBase, BlendColor
colorBlend, double fill, double opacity) {
    double red = PinLightChannel(colorBase.red.get01Value(),
colorBlend.red.get01Value(), fill);
    double green = PinLightChannel(colorBase.green.get01Value(),
colorBlend.green.get01Value(), fill);
    double blue = PinLightChannel(colorBase.blue.get01Value(),
colorBlend.blue.get01Value(), fill);
    return ColorUtils.Opacity(colorBase, new BlendColor(red * 255, green
* 255, blue * 255), opacity);
}

private static double PinLightChannel(double baseValue, double
blendValue, double fill) {
    if (blendValue <= 0.5) {
        return DarkenChannel(baseValue, 2 * blendValue, fill);
    } else {
        return LightenChannel(baseValue, 2 * (blendValue - 0.5), fill);
    }
}
```

点 光(PinLight) RGB [111.00, 80.00, 60.00] ~ HSY [23.53, 51.00, 87.10] ~ HSB [23.53, 45.95, 43.53]

验证



用途示例

1:增加图片对比度

亮光 VividLight

亮光模式是颜色加深和颜色减淡的组合

公式

$$\begin{aligned} r = & \text{VividLight}(b, a) = \left\{ \begin{aligned} & \text{ColorBurn}(b, 2a) \leq a \\ & 0.5 \& \text{ColorDodge}(b, 2(a-0.5)) & a > 0.5 \end{aligned} \right. \& \left. \begin{aligned} & \frac{1 - (1-b)}{2a} \leq a \leq 0.5 \\ & \frac{b}{1 - 2(a-0.5)} & a > 0.5 \end{aligned} \right. \end{aligned}$$

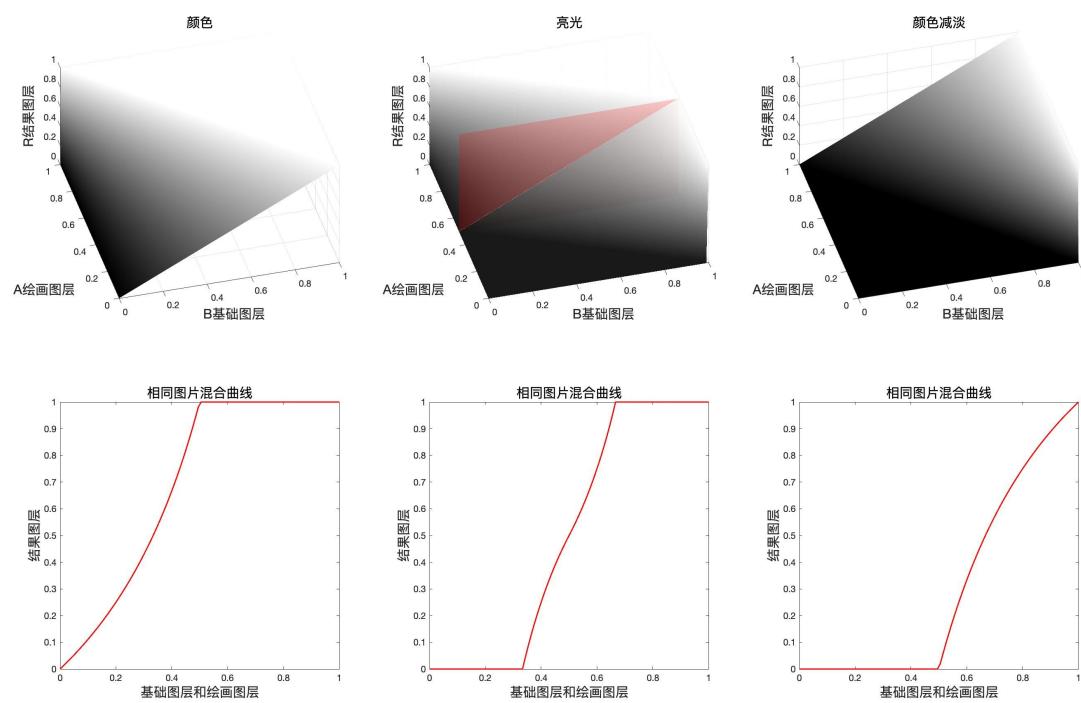
加上fill

$$\begin{aligned} r = & \text{Fill}(b, a) = \left\{ \begin{aligned} & \text{ColorBurn}(b, 2a \times \text{fill}) \leq a \\ & 0.5 \& \text{ColorDodge}(b, 2(a-0.5) \times \text{fill}) & a > 0.5 \end{aligned} \right. \& \left. \begin{aligned} & \frac{1 - (1-b)}{2a \times \text{fill}} \leq a \leq 0.5 \\ & \frac{b}{1 - 2(a-0.5) \times \text{fill}} & a > 0.5 \end{aligned} \right. \end{aligned}$$

融合不透明度

$$r = \text{Opacity}(b, a) = op \times \text{Fill}(b, a) + (1 - op) \times b$$

映射面和同图等效曲线



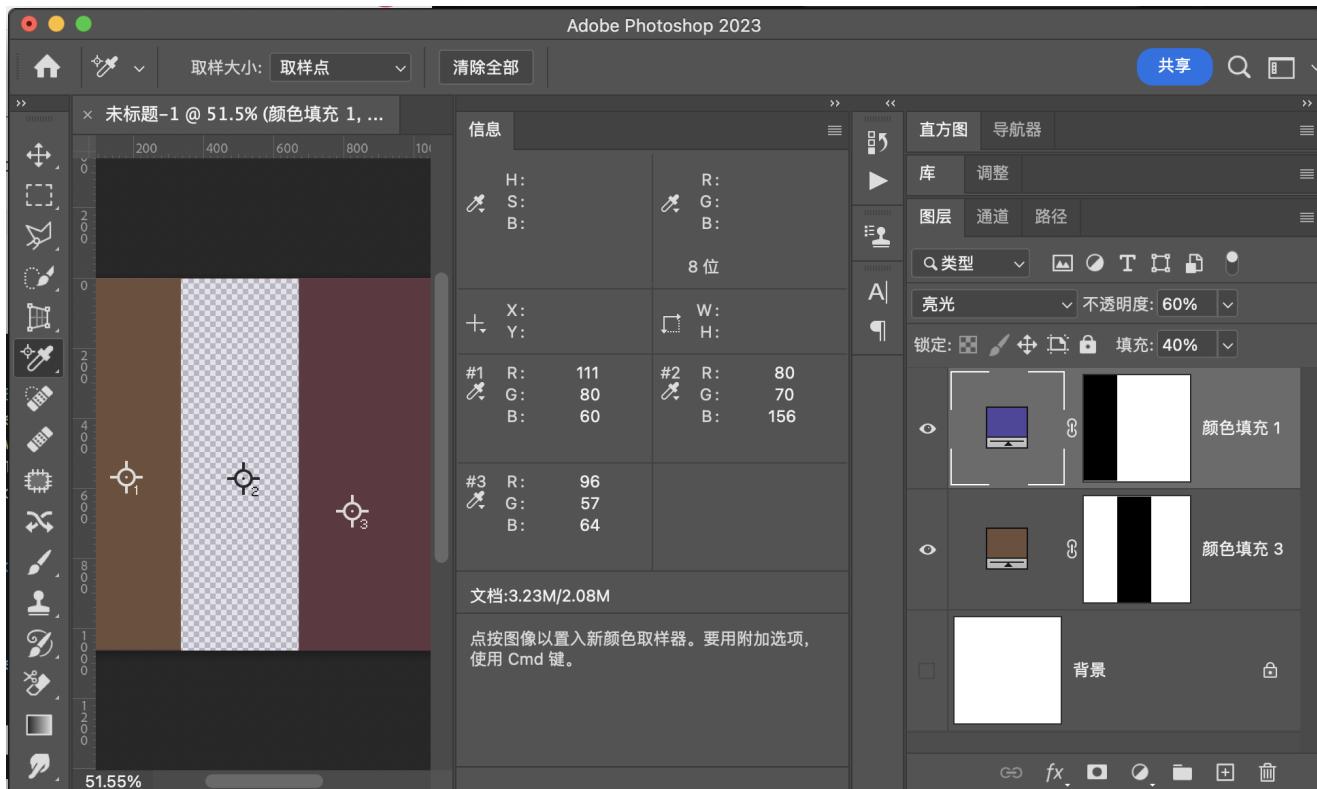
程序模拟该模式计算结果

```
// 亮光
public static BlendColor VividLight(BlendColor colorBase, BlendColor
colorBlend, double fill, double opacity) {
    double red = VividLightChannel(colorBase.red.get01Value(),
colorBlend.red.get01Value(), fill);
    double green = VividLightChannel(colorBase.green.get01Value(),
colorBlend.green.get01Value(), fill);
    double blue = VividLightChannel(colorBase.blue.get01Value(),
colorBlend.blue.get01Value(), fill);
    return ColorUtils.Opacity(colorBase, new BlendColor(red * 255, green
* 255, blue * 255), opacity);
}

private static double VividLightChannel(double minValue, double
blendValue, double fill) {
    if (blendValue <= 0.5) {
        return ColorBurnChannel(minValue, 2 * blendValue, fill);
    } else {
        return ColorDodgeChannel(minValue, 2 * (blendValue - 0.5),
fill);
    }
}
```

亮光(VividLight) RGB [95.87, 56.89, 63.53] ~ HSY [349.77, 38.98, 69.31] ~ HSB [349.77, 40.66, 37.60]

验证



用途示例

1: 同图混合增加图片对比度

2: 通道抠图法，去除白边

实色混合HardMix

实色混合是一种极端的混合方式，但是如果它合填充结合，则会产生一些意想不到的效果，并且我们还能找到它合线性光的关系

公式

$$r = \text{HardMix}(b, a) = \begin{cases} b + a & \text{if } b + a \geq 1 \\ 0 & \text{else} \end{cases}$$

由公式我们可以看出，最后的结果只有两个，所以最后之后保留 $2^3=8$ 种颜色，也就是

$\begin{aligned} &\text{黑} (0,0,0) \& \text{红} (1,0,0) \& \text{黄} (1,1,0) \& \text{白} (0,1,0) \& \text{绿} (0,1,1) \& \text{青} (1,0,1) \& \text{品红} (0,0,1) \& \text{蓝} (0,1,0) \end{aligned}$

加上fill

但是如果填充介入表达式，则结果将合线性光类似

$$r = \text{HardMix}_{\{\text{fill}\}}(b, a) = \begin{cases} b + a - \text{fill} & \text{if } (b + a - \text{fill}) < 0 \\ \text{fill} & \text{else} \end{cases}$$

$a+b-\text{fill}\} \{(1-\text{fill})\} & 0 \leq \frac{\text{fill}}{a+b-\text{fill}} \{(1-\text{fill})\} \leq 1 \& \frac{\text{fill}}{a+b-\text{fill}} \{(1-\text{fill})\} > 1$
 $\end{aligned}\right. \text{right.} \text{\$\$}$

如果 fill 的取值是 0.5 则

$\begin{aligned} r = \text{HardMix}_{\{\text{fill}\}}(b, a) = & \left\{ \begin{aligned} & 0 & \frac{0.5 \times a + 0.5}{(1 - 0.5)} < 0 \\ & 1 & \frac{0.5 \times a + 0.5}{(1 - 0.5)} \leq 1 \& \frac{0.5 \times a + 0.5}{(1 - 0.5)} > 1 \end{aligned} \right. \end{aligned} \& \left\{ \begin{aligned} & a + 2b - 1 < 0 \& a + 2b - 1 & 0 \leq a + 2b - 1 \leq 1 \& a + 2b - 1 > 1 \end{aligned} \right. \end{aligned} \text{\$\$}$

上面的结果就是线性光的表达式，也就是说此时二者等价，或者说是互逆，也就是说，实色混合其实是线性光的强化版本，可以实现线性光的功能而去变化更多。

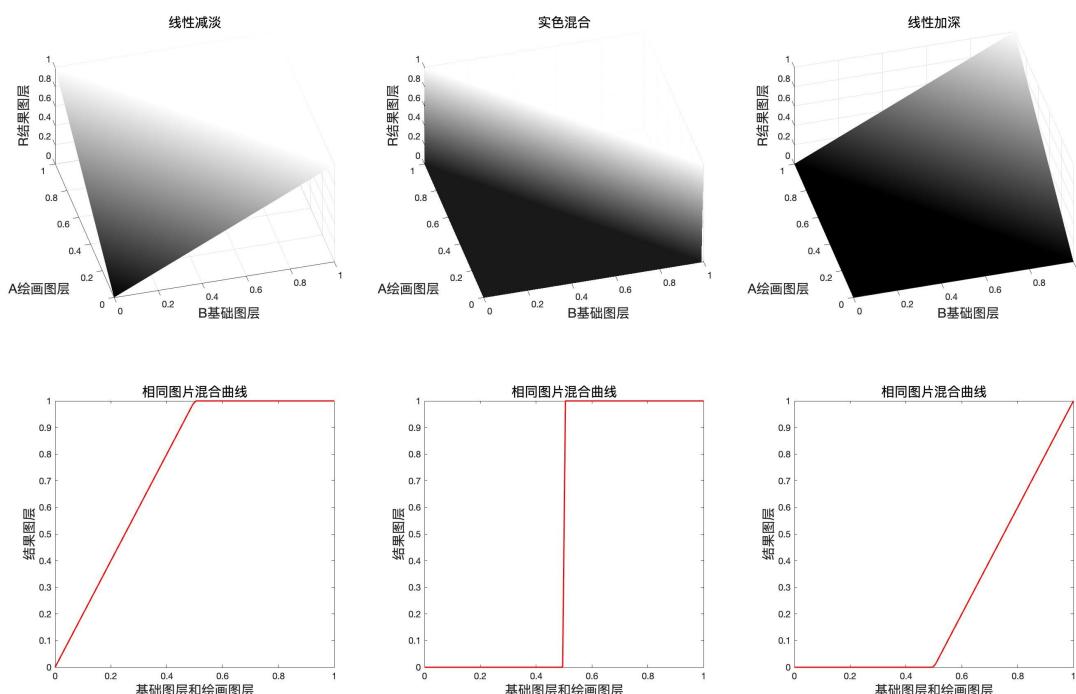
融合不透明度

$r = \text{Opacity}(b, a) = \text{op} \times \text{HardMix}_{\{\text{fill}\}}(b, a) + (1 - \text{op}) \times b$

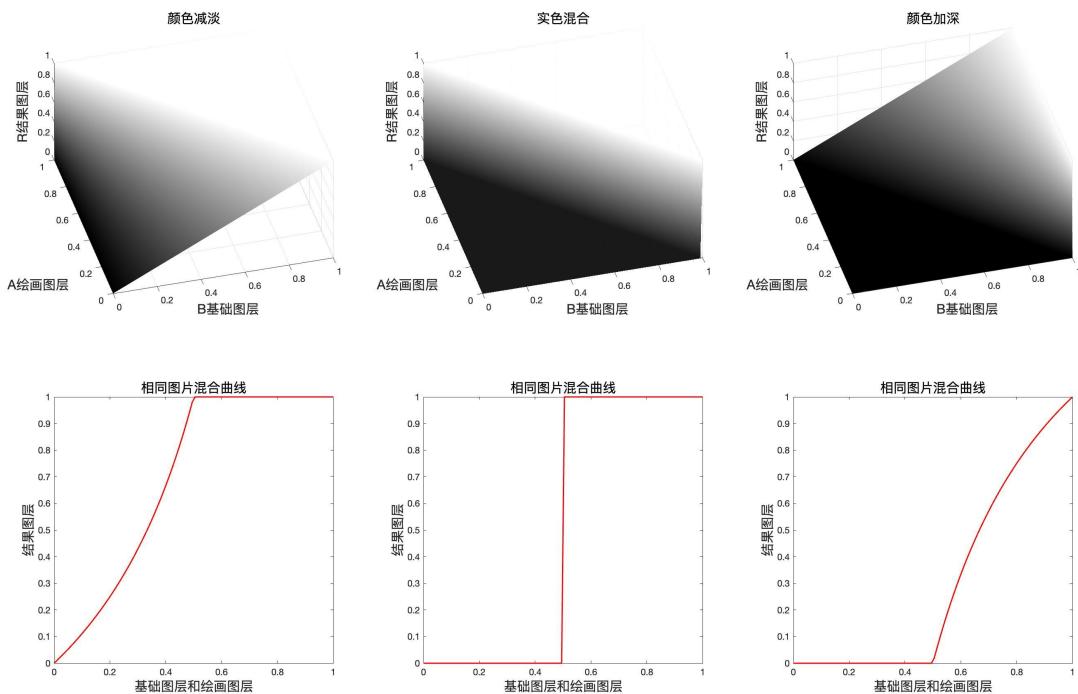
映射面和同图等效曲线

由此我们可以看出，实色混合可以看作线性减淡和线性加深的组合，也可以看作是颜色减淡和颜色加深的组合，注意这里是可以看作，并不是真的。

但是由我们刚才推导出当填充等于 50% 的时候，他可以和线性光互逆，此时我们也可以得出结论，实色混合本质是一种特殊的线性减淡和线性加深的组合，并且线性光是实色混合的特殊形式。



线性减淡和线性加深组合方式（真的）



颜色减淡和颜色加深的组合方式（可以看作是这样）

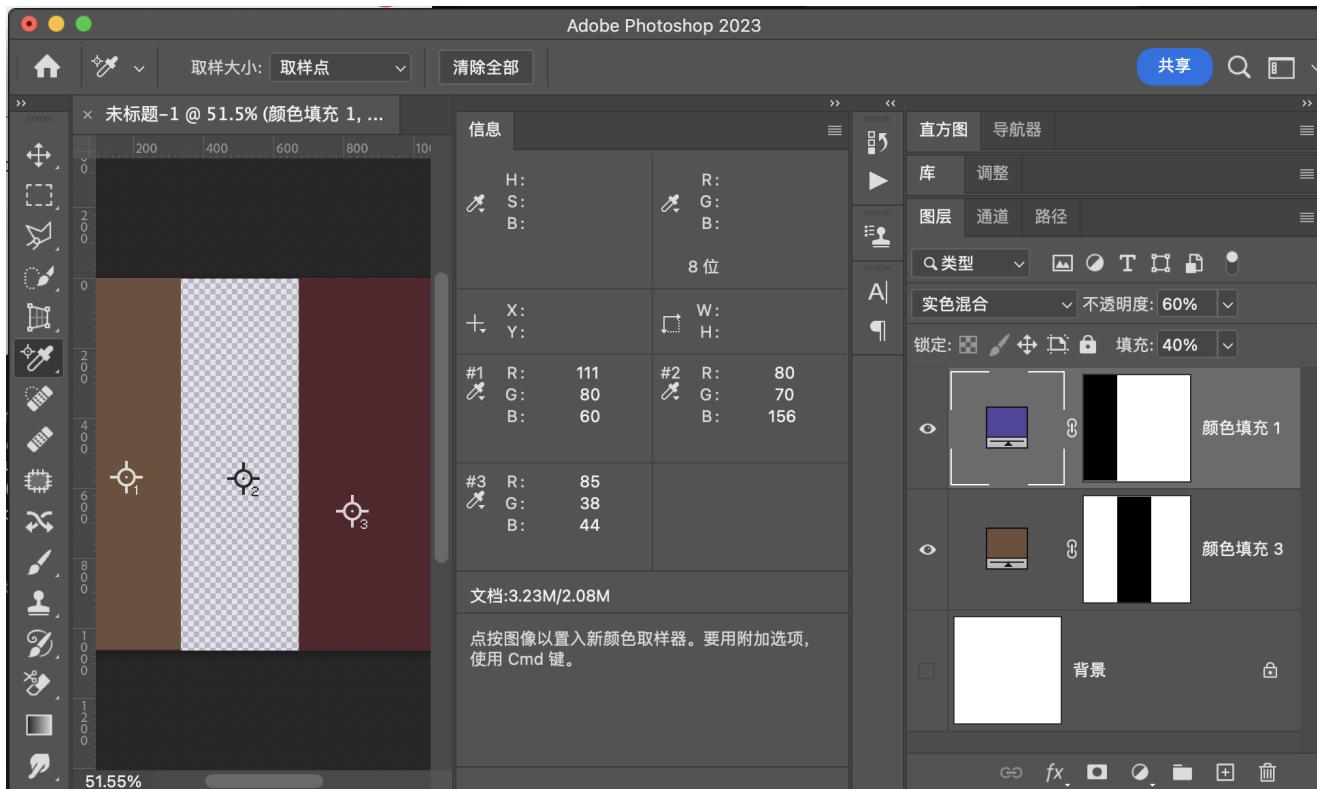
程序模拟该模式计算结果

```
// 实色混合
public static BlendColor HardMix(BlendColor colorBase, BlendColor
colorBlend, double fill, double opacity) {
    double red = HardMixChannel(colorBase.red.get01Value(),
colorBlend.red.get01Value(), fill);
    double green = HardMixChannel(colorBase.green.get01Value(),
colorBlend.green.get01Value(), fill);
    double blue = HardMixChannel(colorBase.blue.get01Value(),
colorBlend.blue.get01Value(), fill);
    return ColorUtils.Opacity(colorBase, new BlendColor(red * 255, green
* 255, blue * 255), opacity);
}

private static double HardMixChannel(double baseValue, double
blendValue, double fill) {
    if (fill == 1) {
        if (baseValue + blendValue >= 0.5) {
            return 1;
        }
        return 0;
    }
    return ColorUtils.round((fill * blendValue + baseValue - fill) /
(1 - fill), 1, 0);
}
```

实色混合(HardMix) RGB [85.40, 38.00, 44.40] ~ HSY [351.90, 47.40, 52.92] ~ HSB [351.90, 55.50, 33.49]

验证



用途示例

- 1: 同图混合增加图片对比度
- 2: 特殊光效，可以根据fill调节

差值组

差值Difference

差值就是基础图层和混合图层的差值的绝对值

公式

$$r = \text{Difference}(b, a) = |b - a|$$

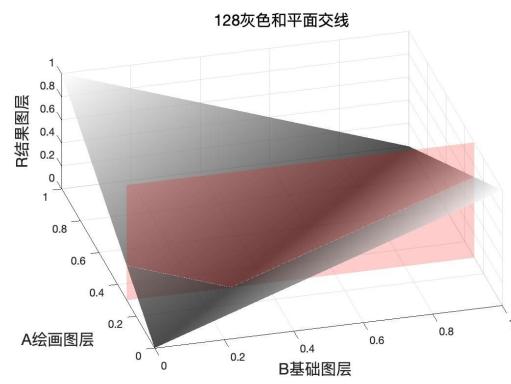
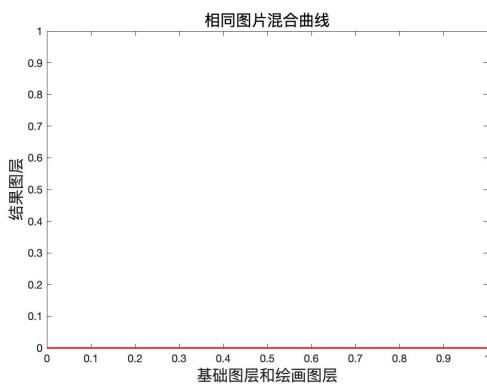
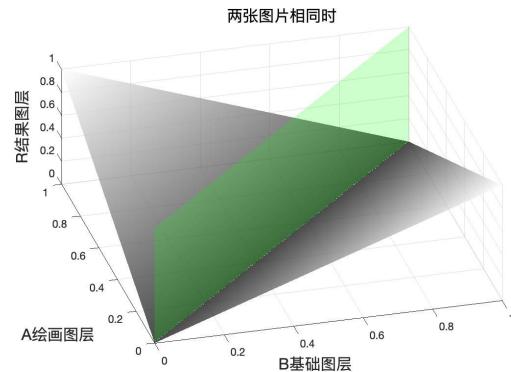
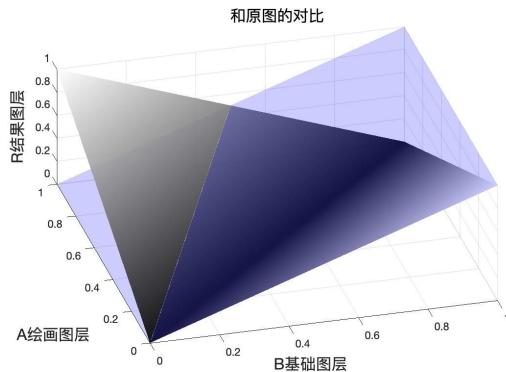
结合填充

$$r = \text{Fill}(b, a) = |b - a| \times \text{fill} + (1 - \text{fill}) \times b$$

融合不透明度

$$r = \text{Opacity}(b, a) = \text{op} \times \text{Fill}(b, a) + (1 - \text{op}) \times b$$

映射面和同图等效曲线



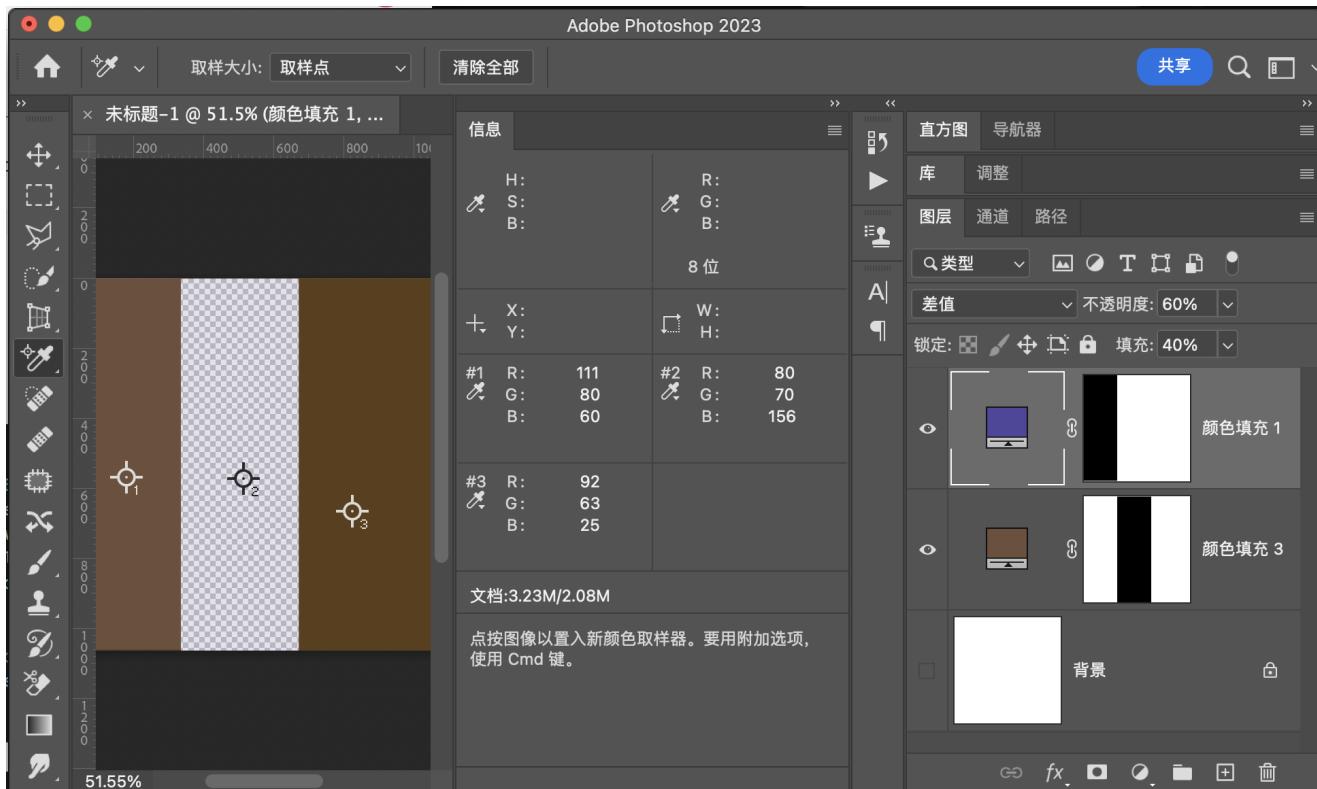
程序模拟该模式计算结果

```
// 差值
public static BlendColor Difference(BlendColor colorBase, BlendColor
colorBlend, double fill, double opacity) {
    double red = DifferenceChannel(colorBase.red.get01Value(),
colorBlend.red.get01Value(), fill);
    double green = DifferenceChannel(colorBase.green.get01Value(),
colorBlend.green.get01Value(), fill);
    double blue = DifferenceChannel(colorBase.blue.get01Value(),
colorBlend.blue.get01Value(), fill);
    return ColorUtils.Opacity(colorBase, new BlendColor(red * 255, green
* 255, blue * 255), opacity);
}

private static double DifferenceChannel(double baseValue, double
blendValue, double fill) {
    return ColorUtils.round(Math.abs(baseValue - blendValue * fill), 1,
0);
}
```

差 值(Difference) RGB [91.80, 63.20, 25.44] ~ HSY [34.14, 66.36, 67.63] ~ HSB [34.14, 72.29, 36.00]

验证



用途示例

1:对齐图片

2:颜色矫正

排除Exclusion

公式

$$r = \text{Exclusion}(b, a) = b + a - 2ba$$

结合填充

$$r = \text{Fill}(b, a) = (b + a - 2ba) \times \text{fill} + (1 - \text{fill}) \times b$$

融合不透明度

$$r = \text{Opacity}(b, a) = \text{op} \times \text{Fill}(b, a) + (1 - \text{op}) \times b$$

如果在该模式下，混合图层是白色，黑色或者中性灰色

白色

$\$r = \text{Exclusion}(b, 1) = b + 1 - 2b \times 1 = 1 - b$

等于负片

黑色

$\$r = \text{Exclusion}(b, 0) = b + 1 - 2b \times 0 = b$

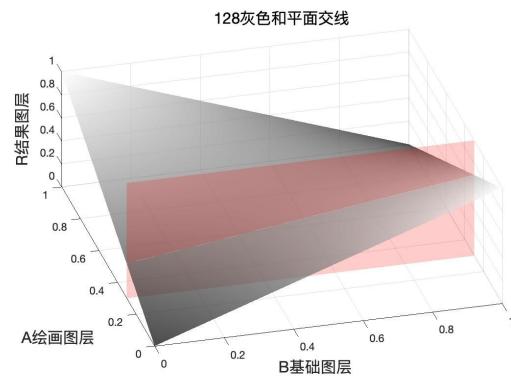
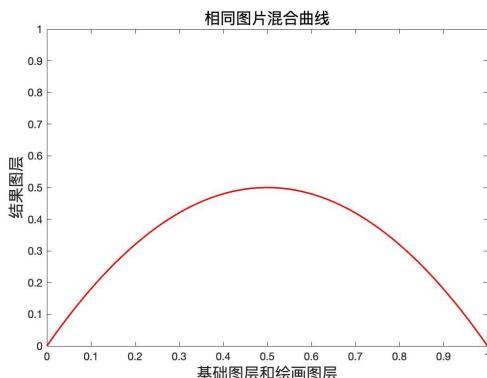
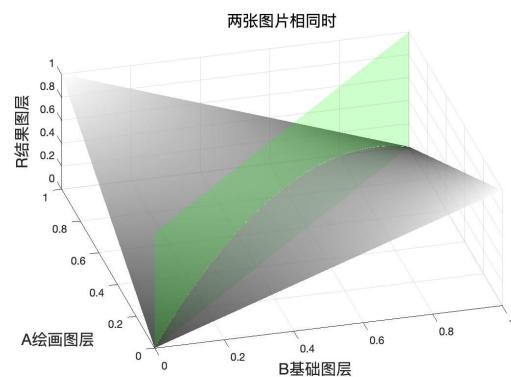
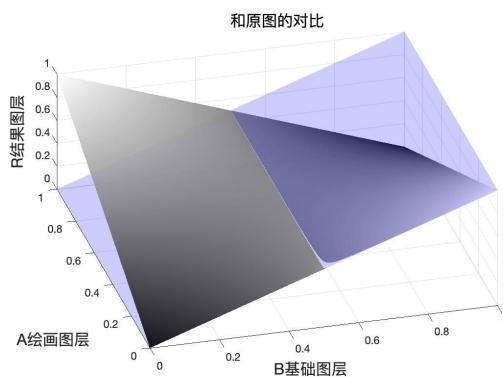
等于原图

中性灰

$\$r = \text{Exclusion}(b, \frac{1}{2}) = b + \frac{1}{2} - 2b \times \frac{1}{2} = \frac{1}{2}$

依然是中性灰

映射面和同图等效曲线



程序模拟该模式计算结果

```
// 排除
public static BlendColor Exclusion(BlendColor colorBase, BlendColor
colorBlend, double fill, double opacity) {
    double red = ExclusionChannel(colorBase.red.get01Value(),
colorBlend.red.get01Value(), fill);
    double green = ExclusionChannel(colorBase.green.get01Value(),
colorBlend.green.get01Value(), fill);
    double blue = ExclusionChannel(colorBase.blue.get01Value(),
colorBlend.blue.get01Value(), fill);
    return new BlendColor(red, green, blue, opacity);
}
```

```

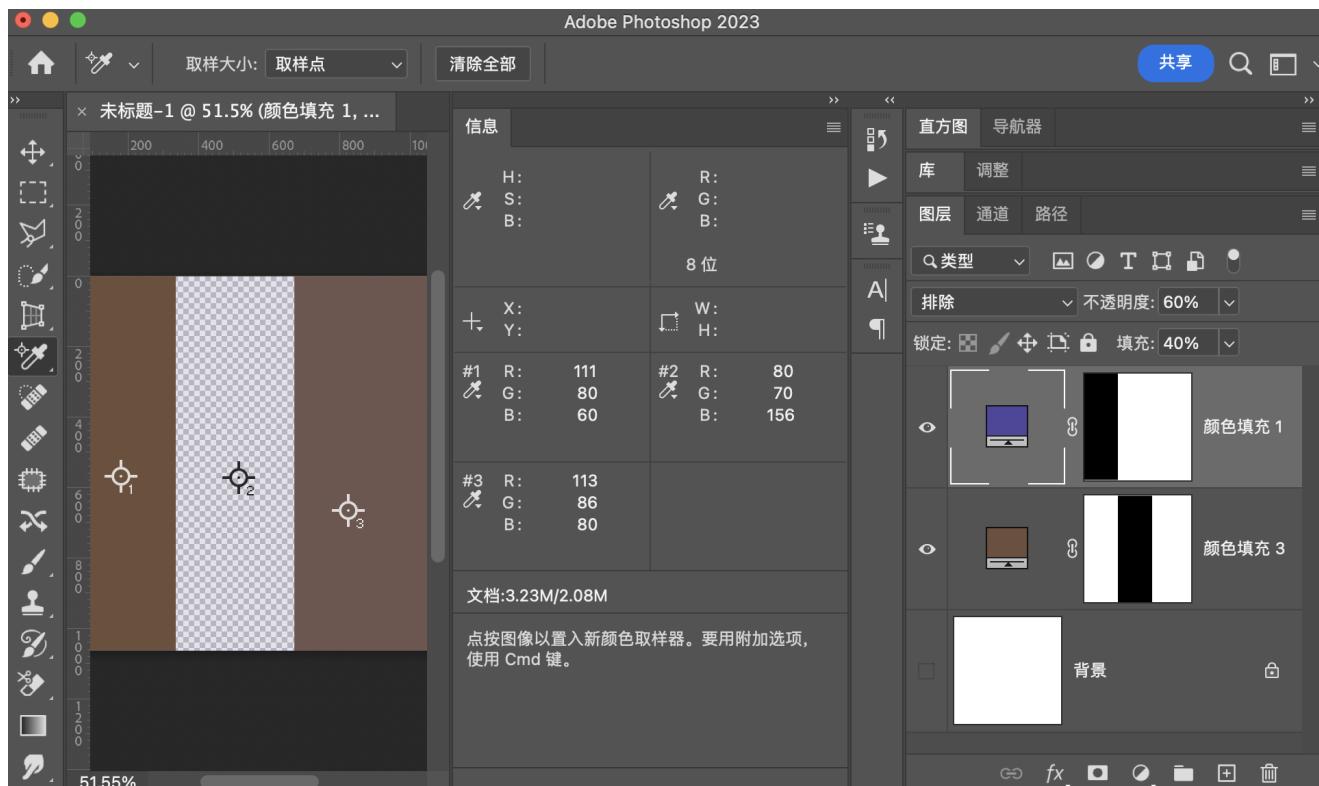
        colorBlend.blue.get01Value(), fill);
    return ColorUtils.Opacity(colorBase, new BlendColor(red *255, green*
255, blue * 255), opacity);
}

private static double ExclusionChannel(double baseValue, double
blendValue, double fill) {
    return ColorUtils.round((baseValue + blendValue - 2 * baseValue *
blendValue) * fill + (1 - fill) * baseValue,
                           1, 0);
}

```

排除 除(Exclusion) RGB [113.48, 86.26, 79.82] ~ HSY [11.47, 33.66, 93.72] ~ HSB [11.47, 29.66, 44.50]

验证



用途示例

制作特殊光效，比如人物肖像

减去Subtract

公式

$$r = \text{Subtract}(b, a) = b - a$$

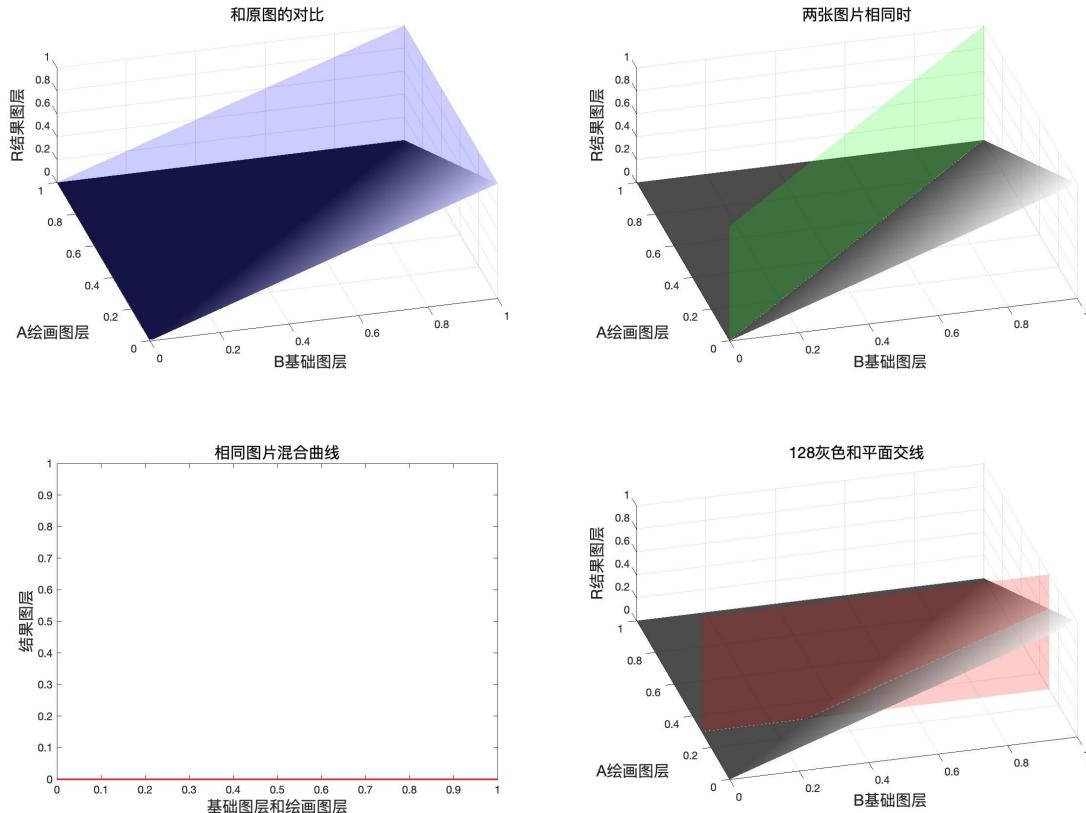
结合填充

$\$r=Fill(b,a)=round(b-a)\times fill + (1-fill)\times b\$$

融合不透明度

$\$r=Opacity(b,a)=op\times Fill(b,a)+(1-op)\times b\$$

映射面和同图等效曲线



程序模拟该模式计算结果

```
// 减去
public static BlendColor Substact(BlendColor colorBase, BlendColor
colorBlend, double fill, double opacity) {
    double red = SubstactChannel(colorBase.red.get01Value(),
colorBlend.red.get01Value(), fill);
    double green = SubstactChannel(colorBase.green.get01Value(),
colorBlend.green.get01Value(), fill);
    double blue = SubstactChannel(colorBase.blue.get01Value(),
colorBlend.blue.get01Value(), fill);
    return ColorUtils.Opacity(colorBase, new BlendColor(red *255, green*
255, blue * 255), opacity);
}

private static double SubstactChannel(double minValue, double
```

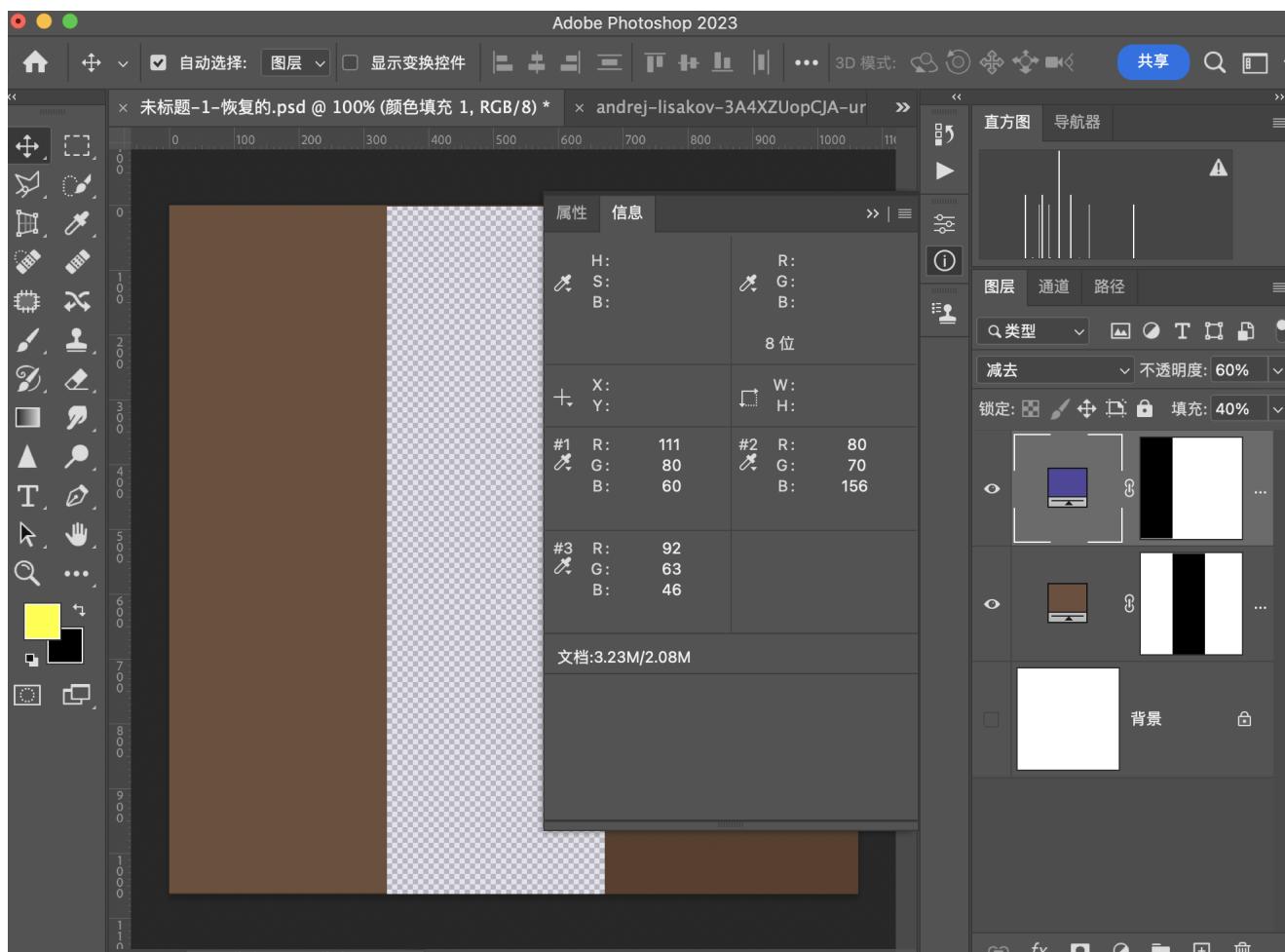
```

blendValue, double fill) {
    return ColorUtils.round(ColorUtils.round((baseValue - blendValue),
1, 0) *fill + (1 - fill)* baseValue, 1, 0);
}

```

减去(Subtract) RGB [91.80, 63.20, 45.60] ~ HSY [22.86, 46.20, 69.84] ~ HSB [22.86, 50.33, 36.00]

验证



制作线稿效果

划分Divide

公式

$$r = \text{Divide}(b, a) = \frac{b}{a}$$

结合填充

$$r = \text{Fill}(b, a) = \frac{b}{a} \times \text{fill} + (1 - \text{fill}) \times b$$

融合不透明度

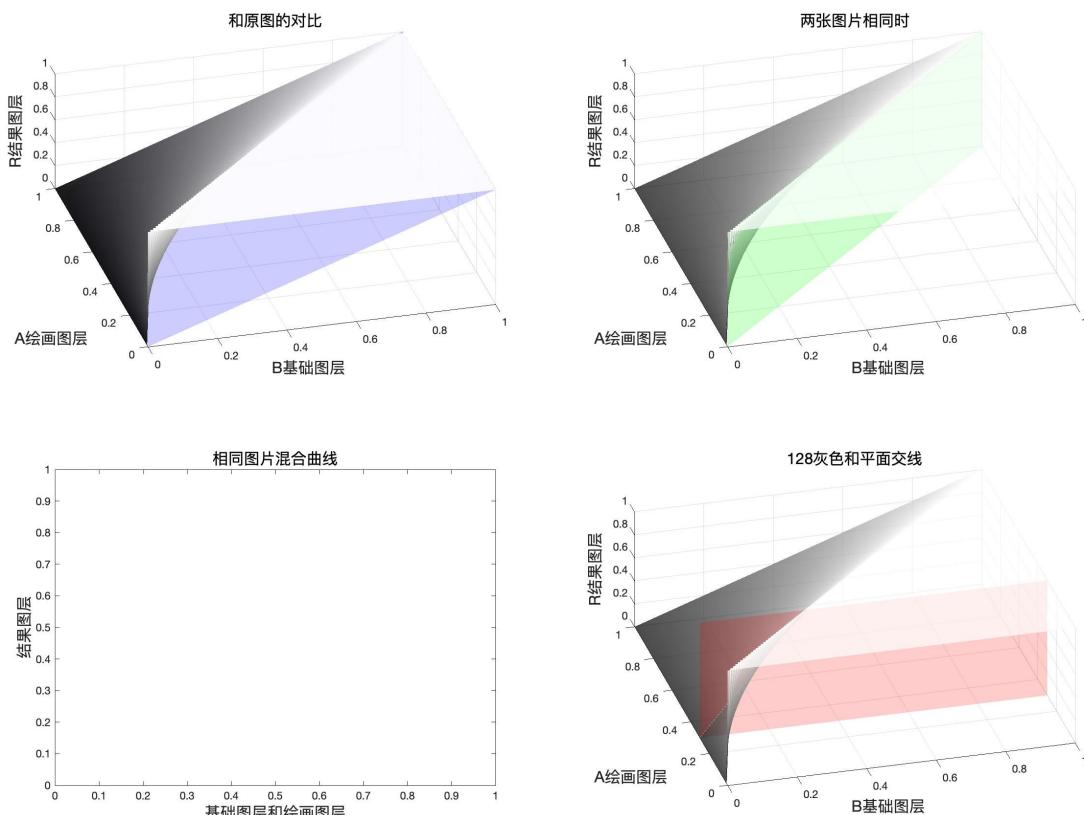
$\$r=Opacity(b,a)=op\times Fill(b,a)+(1-op)\times b$

划分和颜色减淡可以通过一次负片操作转换

一次负片

$\$r=Divide(b,1-a)=\frac{b}{1-a}=ColorDodge(b,a)$

映射面和同图等效曲线



程序模拟该模式计算结果

```
// 划分
public static BlendColor Divide(BlendColor colorBase, BlendColor
colorBlend, double fill, double opacity) {
    double red = DivideChannel(colorBase.red.get01Value(),
colorBlend.red.get01Value(), fill);
    double green = DivideChannel(colorBase.green.get01Value(),
colorBlend.green.get01Value(), fill);
    double blue = DivideChannel(colorBase.blue.get01Value(),
colorBlend.blue.get01Value(), fill);
    return ColorUtils.Opacity(colorBase, new BlendColor(red * 255, green
* 255, blue * 255), opacity);
}
```

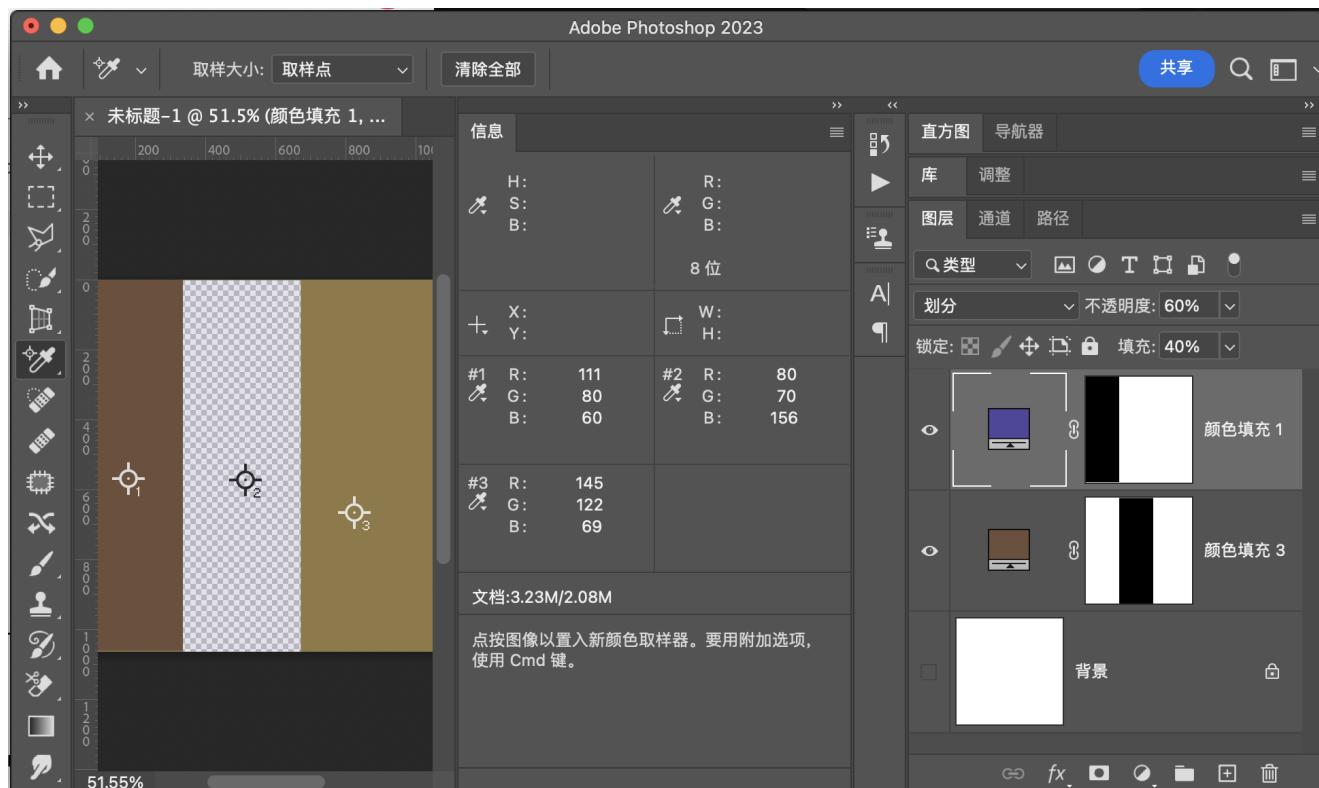
```

private static double DivideChannel(double baseValue, double blendValue,
double fill) {
    return ColorUtils.round(Math.min(1, baseValue / blendValue) * fill +
(1 - fill) * baseValue, 1, 0);
}

```

划 分(Divide) RGB[145.56, 122.00, 69.14] ~ HSY[41.50, 76.42, 123.25] ~ HSB[41.50, 52.50, 57.08]

验证



用途示例

颜色矫正

颜色组

这一组和其他都不同，这一组是基于HSY颜色空间，并且设计的计算都是方程组
转化伪代码

计算明度

$\text{Lum}(C) = 0.3 \times C_{\text{red}} + 0.59 \times C_{\text{green}} + 0.11 \times C_{\text{blue}}$

计算饱和度

$\text{Sat}(C) = \max(C_{\text{red}}, C_{\text{green}}, C_{\text{blue}})$

这里采用了比较取巧的做法，就是只涉及饱和度和明度的改变，不涉及直接改变色相，因为改变色相相当于直接同时改变饱和度和明度。

下面是改变明度的伪代码

```
 $$\begin{aligned} \text{SetLum}(C, lum) &= \& \begin{aligned} &\text{lum} - \text{Lum}(C) \& C_{\text{red}} = C_{\text{red}} + d \\ &C_{\text{green}} = C_{\text{green}} + d \& C_{\text{blue}} = C_{\text{blue}} + d \end{aligned} \\ &\& \text{return} \space \end{aligned} $$
```

ClipColor(C) \end{aligned} \end{aligned} \$\$

这是修改饱和度的伪代码

```
 $$\begin{aligned} \text{SetSat}(C, sat) &= \& \begin{aligned} &\text{if } C_{\text{max}} > C_{\text{min}} \& \quad \\ &C_{\text{mid}} = \frac{(C_{\text{mid}} - C_{\text{min}})}{\text{times sat}} \{C_{\text{max}} - C_{\text{min}}\} \& \quad \\ &C_{\text{max}} = \text{sat} \& \text{else} \& \quad \\ &C_{\text{mid}} = C_{\text{max}} = 0 \& C_{\text{min}} = 0 \& \text{return} \quad \end{aligned} \end{aligned} $$
```

这是矫正误差的伪代码

```
 $$\begin{aligned} \text{ClipColor}(C) &= \& \begin{aligned} &\text{ClipColor}(C) \& \begin{aligned} &\text{lum} = \text{Lum}(c) \& \min = \\ &\text{Min}(C_{\text{red}}, C_{\text{green}}, C_{\text{blue}}) \& \max = \text{Max}(C_{\text{red}}, C_{\text{green}}, C_{\text{blue}}) \& \text{if } \min < 0 \& \quad \\ &C_{\text{red}} = \text{lum} + \frac{(C_{\text{red}} - \text{lum})}{\text{times lum}} \{ \text{lum} - \min \} \& \quad \\ &C_{\text{green}} = \text{lum} + \frac{(C_{\text{green}} - \text{lum})}{\text{times lum}} \{ \text{lum} - \min \} \& \quad \\ &C_{\text{blue}} = \text{lum} + \frac{(C_{\text{blue}} - \text{lum})}{\text{times lum}} \{ \text{lum} - \min \} \& \quad \\ &\text{if } \max > 1 \& \quad \\ &C_{\text{red}} = \text{lum} + \frac{(C_{\text{red}} - \text{lum})}{\text{times (1-lum)}} \{ \max - \text{lum} \} \& \quad \\ &C_{\text{green}} = \text{lum} + \frac{(C_{\text{green}} - \text{lum})}{\text{times (1-lum)}} \{ \max - \text{lum} \} \& \quad \\ &C_{\text{blue}} = \text{lum} + \frac{(C_{\text{blue}} - \text{lum})}{\text{times (1-lum)}} \{ \max - \text{lum} \} \& \quad \\ &\text{return} \space \end{aligned} \end{aligned} \end{aligned} $$
```

色相Hue

计算方法是基于这个公式

```
 $$\text{Hue}(H_r, S_r, Y_r) = \text{Hue}((H_a, S_a, Y_a), (H_b, S_b, Y_b)) = (H_a, S_b, Y_b) $$
```

算出HSY的值之后再把HSY转化为RGB的数值

此时我们想修改基础图层的色相，但是我们只有修改饱和度和明度的公式，于是我们直接对混合图层使用设置饱和度和明度于是我们得到

```
 $$\text{Hue}(C_b, C_a) = \text{SetLum}(\text{SetSat}(C_a, \text{Sat}(C_b)), \text{Lum}(C_b)) $$
```

结合填充

```
 $$r = \text{Fill}(C_b, C_a) = \text{Hue}(C_b, C_a) \times \text{fill} + (1 - \text{fill}) \times C_b $$
```

融合不透明度

```
 $$r = \text{Opacity}(C_b, C_a) = \text{op} \times \text{Fill}(C_b, C_a) + (1 - \text{op}) \times C_b $$
```

程序模拟该模式计算结果

```
// 色相模式

public static BlendColor HUE(BlendColor colorBase, BlendColor
colorBlend, double fill, double opacity) {
    BlendColor temp = HUE_Sub(colorBase, colorBlend);
    double red = temp.red.get01Value() * fill +
    (1 - opacity) * colorBlend.red.get01Value();
    BlendColor result = new BlendColor(red, colorBase.green.get01Value(),
    colorBase.blue.get01Value());
    return result;
}
```

```

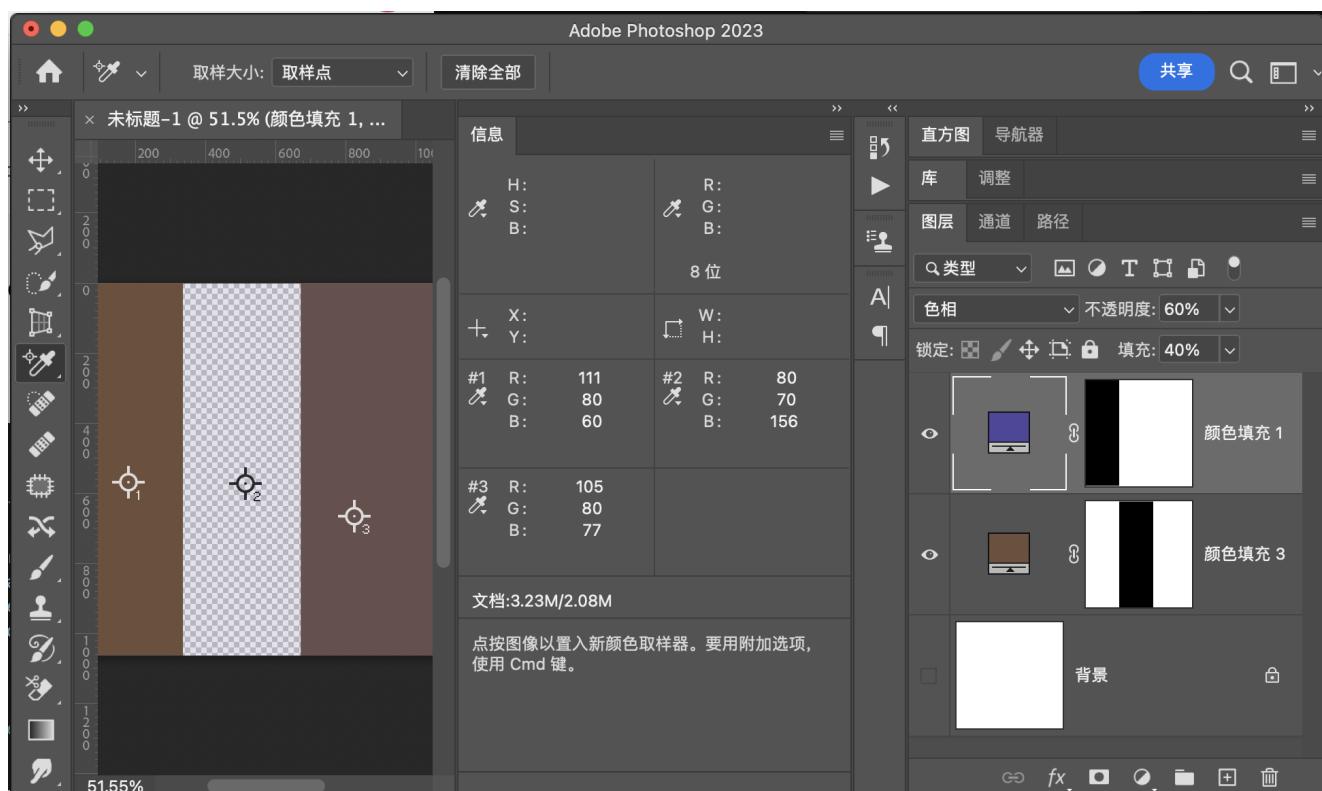
        colorBase.red.get01Value() * (1 - fill);
        double green = temp.green.get01Value() * fill +
colorBase.green.get01Value() * (1 - fill);
        double blue = temp.blue.get01Value() * fill +
colorBase.blue.get01Value() * (1 - fill);
        return ColorUtils.Opacity(colorBase, new BlendColor(red * 255, green
* 255, blue * 255), opacity);
    }

private static BlendColor HUE_Sub(BlendColor colorBase, BlendColor
colorBlend) {
    return ColorUtils.setLum(ColorUtils.setSat(colorBlend,
colorBase.getSat()), colorBase.getLum());
}

```

色 相(hue) RGB [104.91, 79.93, 76.97] ~ HSY [6.36, 27.94,
87.10] ~ HSB [6.36, 26.63, 41.14]

验证



用途示例

饱和度Saturation

公式

计算方法是基于这个公式

$\$(H_r, S_r, Y_r) = \text{Saturation}((H_a, S_a, Y_a), (H_b, S_b, Y_b)) = (H_b, S_a, Y_b) \$\$$

设置饱和度，就直接对基础图层使用设置饱和度

$\$ \$ \text{Saturation}(C_b, C_a) = \text{SetLum}(\text{SetSat}(C_b, \text{Sat}(C_s)), \text{Lum}(C_b)) \$ \$$

结合填充

$\$ \$ r = \text{Fill}(C_b, C_a) = \text{Saturation}(C_b, C_a) \text{ fill} + (1 - \text{fill}) \times C_b \$ \$$

融合不透明度

$\$ \$ r = \text{Opacity}(C_b, C_a) = \text{op} \times \text{Fill}(C_b, C_a) + (1 - \text{op}) \times C_b \$ \$$

程序模拟该模式计算结果

```
// 饱和度模式

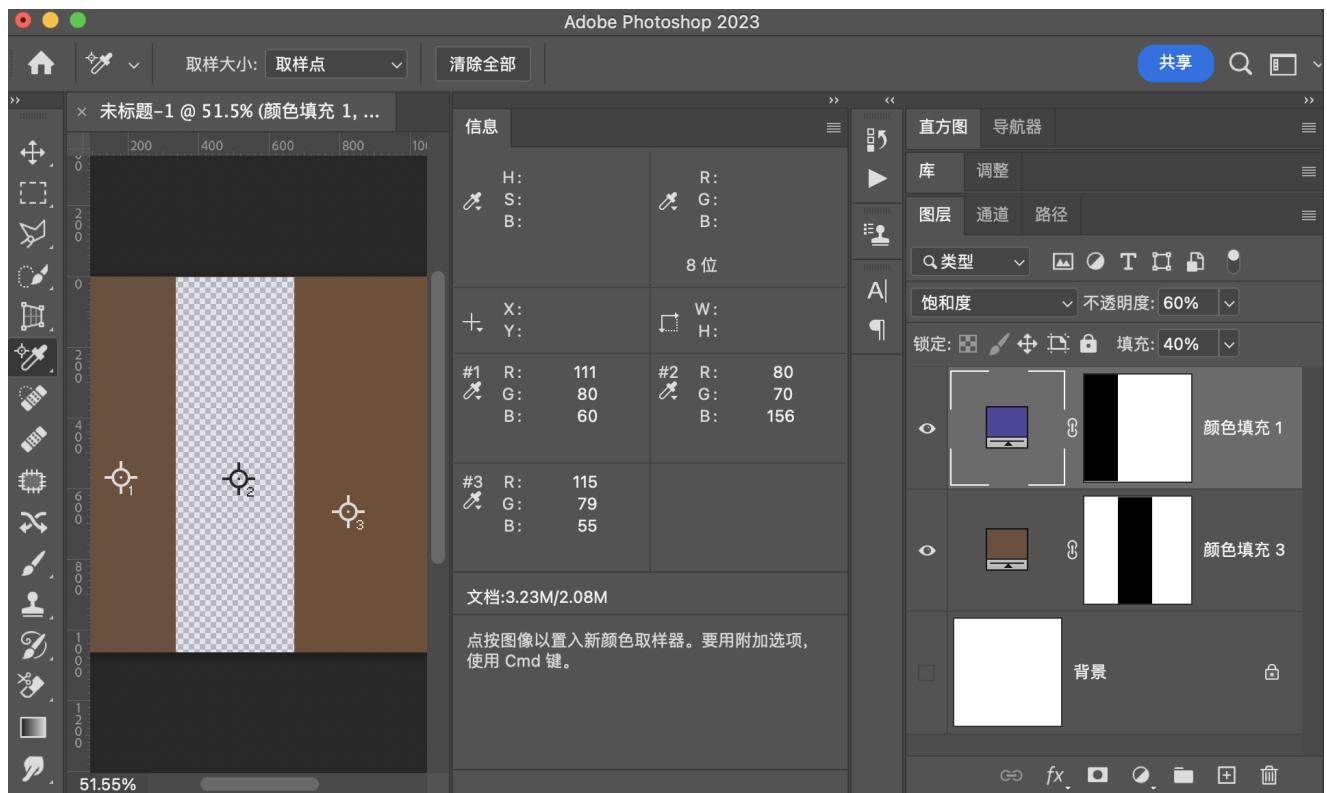
public static BlendColor Saturation(BlendColor colorBase, BlendColor
colorBlend, double fill, double opacity) {

    double redbase = colorBase.red.get01Value();
    double greenbase = colorBase.green.get01Value();
    double bluebase = colorBase.blue.get01Value();
    BlendColor temp = Saturation_Sub(colorBase, colorBlend);
    double red = temp.red.get01Value() * fill + redbase * (1 -
fill);
    double green = temp.green.get01Value() * fill + greenbase * (1 -
fill);
    double blue = temp.blue.get01Value() * fill + bluebase * (1 -
fill);
    return ColorUtils.Opacity(new BlendColor(redbase * 255,
greenbase * 255, bluebase * 255),
                           new BlendColor(red * 255, green * 255, blue * 255),
                           opacity);
}

private static BlendColor Saturation_Sub(BlendColor colorBase,
BlendColor colorBlend) {
    double sat = colorBlend.getSat();
    double lum = colorBase.getLum();
    return ColorUtils.setLum(ColorUtils.setSat(colorBase, sat),
lum);
}
```

饱和度(Saturation) RGB[114.94, 78.83, 55.54] ~ HSY[23.53, 59.40, 87.10] ~ HSB[23.53, 51.68, 45.07]

验证



用途示例

颜色匹配

颜色Color

计算方法是基于这个公式

$\$(\text{H}_r, \text{S}_r, \text{Y}_r) = \text{Color}((\text{H}_a, \text{S}_a, \text{Y}_a), (\text{H}_b, \text{S}_b, \text{Y}_b)) = (\text{H}_a, \text{S}_a, \text{Y}_b) \$\$$

直接对混合图层使用设置明度，则可以得到需要的等效结果

$\$Color(C_b, C_a) = SetLum(C_a, Lum(C_b)) \$\$$

结合填充

$\$r = Fill(C_b, C_a) = Saturation(C_b, C_a) \text{ fill} + (1 - fill) \times C_b \$\$$

融合不透明度

$\$r = Opacity(C_b, C_a) = op \times Fill(C_b, C_a) + (1 - op) \times C_b \$\$$

程序模拟该模式计算结果

```
// 颜色模式
public static BlendColor BlendColor(BlendColor colorBase, BlendColor
colorBlend, double fill, double opacity) {
    BlendColor temp = Color_Sub(colorBase, colorBlend);
    double red = temp.red.get01Value() * fill +
colorBase.red.get01Value() * (1 - fill);
```

```

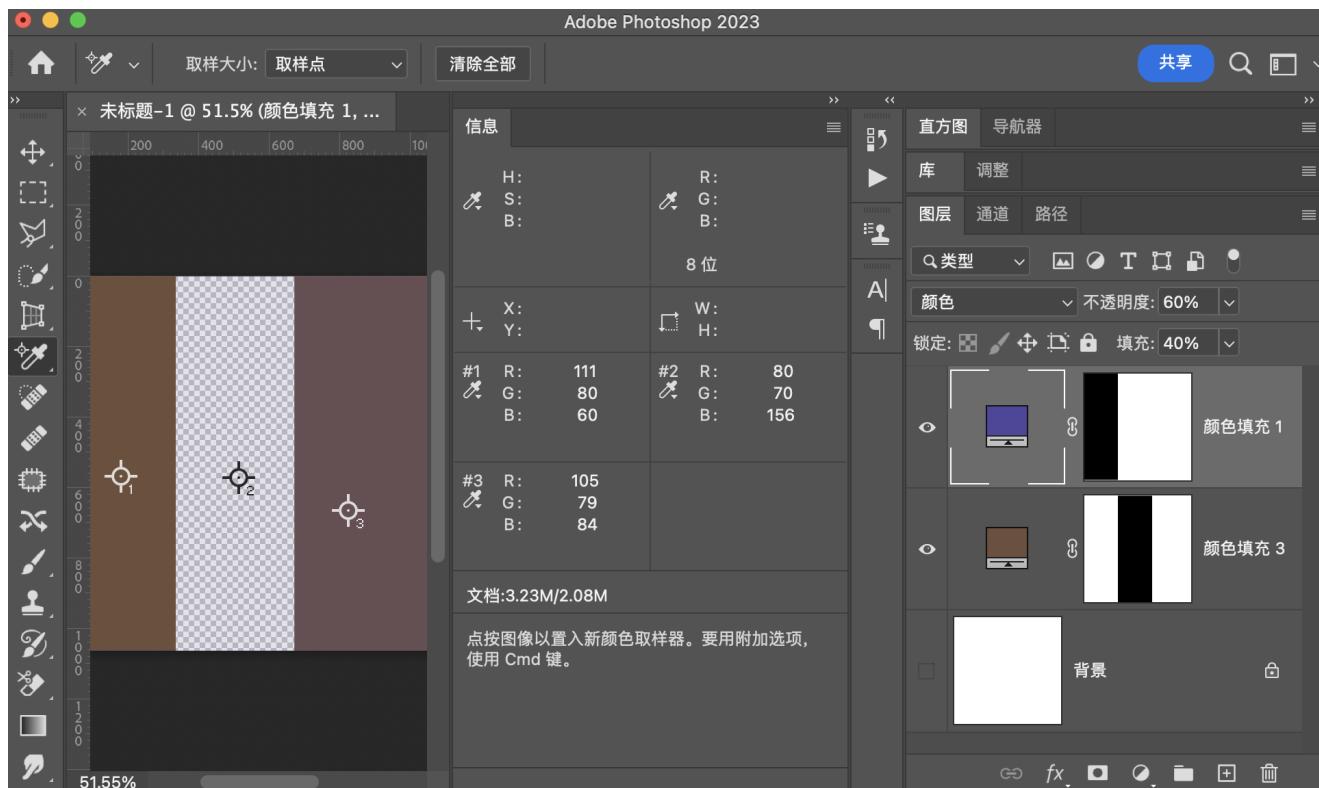
        double green = temp.green.get01Value() * fill +
colorBase.green.get01Value() * (1 - fill);
        double blue = temp.blue.get01Value() * fill +
colorBase.blue.get01Value() * (1 - fill);
        return ColorUtils.opacity(colorBase, new BlendColor(red * 255,
green * 255, blue * 255), opacity);
    }

    private static BlendColor Color_Sub(BlendColor colorBase, BlendColor
colorBlend) {
        return ColorUtils.setLum(colorBlend, colorBase.getLum());
}

```

颜色(BlendColor) RGB[104.67, 78.71, 84.15]~ HSY[347.43, 25.96, 87.10]~ HSB[347.43, 24.80, 41.05]

验证



用途示例

明度Luminosity

计算方法是基于这个公式

$$Luminosity((H_a, S_a, Y_a), (H_b, S_b, Y_b)) = (H_b, S_b, Y_a)$$

设置明度，就直接使用设置明度

\$\$Luminosity(C_b, C_a) = SetLum(C_b, Lum(C_a))\$\$

结合填充

\$\$r = Fill(C_b, C_a) = Luminosity(C_b, C_a) \cdot fill + (1 - fill) \cdot C_b\$\$

融合不透明度

\$\$r = Opacity(C_b, C_a) = op \cdot fill + (1 - op) \cdot C_b\$\$

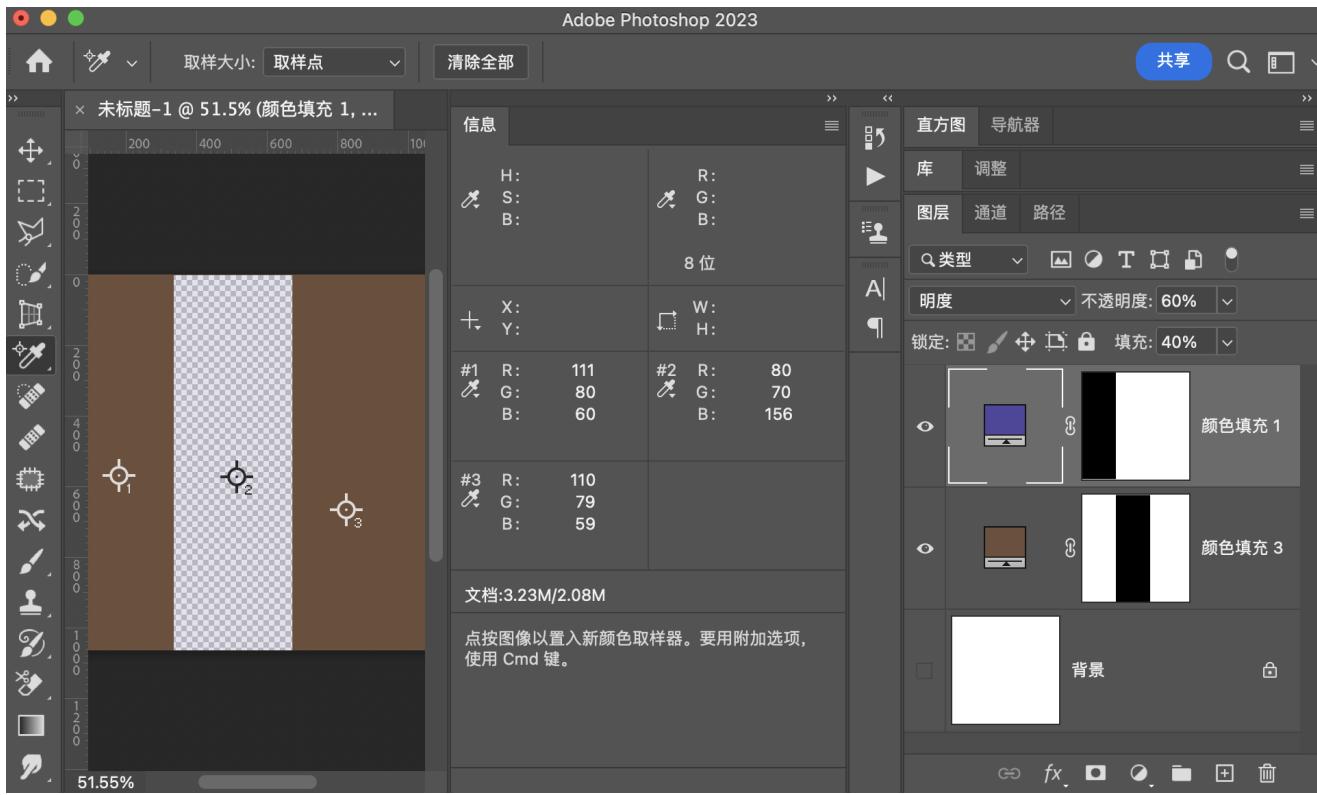
程序模拟该模式计算结果

```
// 明度模式
public static BlendColor Luminosity(BlendColor colorBase, BlendColor
colorBlend, double fill, double opacity) {
    BlendColor temp = Luminosity_Sub(colorBase, colorBlend);
    double red = temp.red.get01Value() * fill +
colorBase.red.get01Value() * (1 - fill);
    double green = temp.green.get01Value() * fill +
colorBase.green.get01Value() * (1 - fill);
    double blue = temp.blue.get01Value() * fill +
colorBase.blue.get01Value() * (1 - fill);
    return ColorUtils.Opacity(colorBase, new BlendColor(red * 255,
green * 255, blue * 255), opacity);
}

private static BlendColor Luminosity_Sub(BlendColor colorBase,
BlendColor colorBlend) {
    return ColorUtils.setLum(colorBase, colorBlend.getLum());
}
```

明 度(Luminosity) RGB[109.89, 78.89, 58.89] ~ HSY[23.53, 51.00,
85.99] ~ HSB[23.53, 46.41, 43.09]

验证



用途示例

和黑白调整图层配合，可以实现一些特殊效果

特殊的5种模式

穿透

穿透会出现在建立分组的时候，如果选择穿透，则此时效果和不建分组一样，但是如果修改为正常或者其他模式，则会先把这一组的图层计算出结果，然后用结果作为混合图层和下方图层进行运算。

相加

相加在计算和应用图像中，此时相当于强化的线性减淡，公式为

$$\text{Add}(b,a) = \frac{b+a}{\max(b,a)} + \text{补偿值}$$

缩放的取值范围是\$[1,2]\$

补偿值的取值范围是\$[0,255]\$

相减

相减在计算和应用图像中，此时相当于强化的减去，公式为

$$\text{Subtract}(b,a) = b - a + \text{补偿值}$$

缩放的取值范围是\$[1,2]\$

补偿值的取值范围是\$[0,255]\$

背后

背后模式

简单来说就是，有像素点则笔刷或油漆桶工具不能修改，没有像素点的透明像素可以被修改。

擦除

功能相当于橡皮擦

调整图层和图层混合模式

如果调整图层和混合图层混用会发生什么

我们假设调整图层为\$Adjustment(Layer)\$,

则对于像素点\$Adjustment(pix)\$,

对于通道\$Adjustment(channel)\$

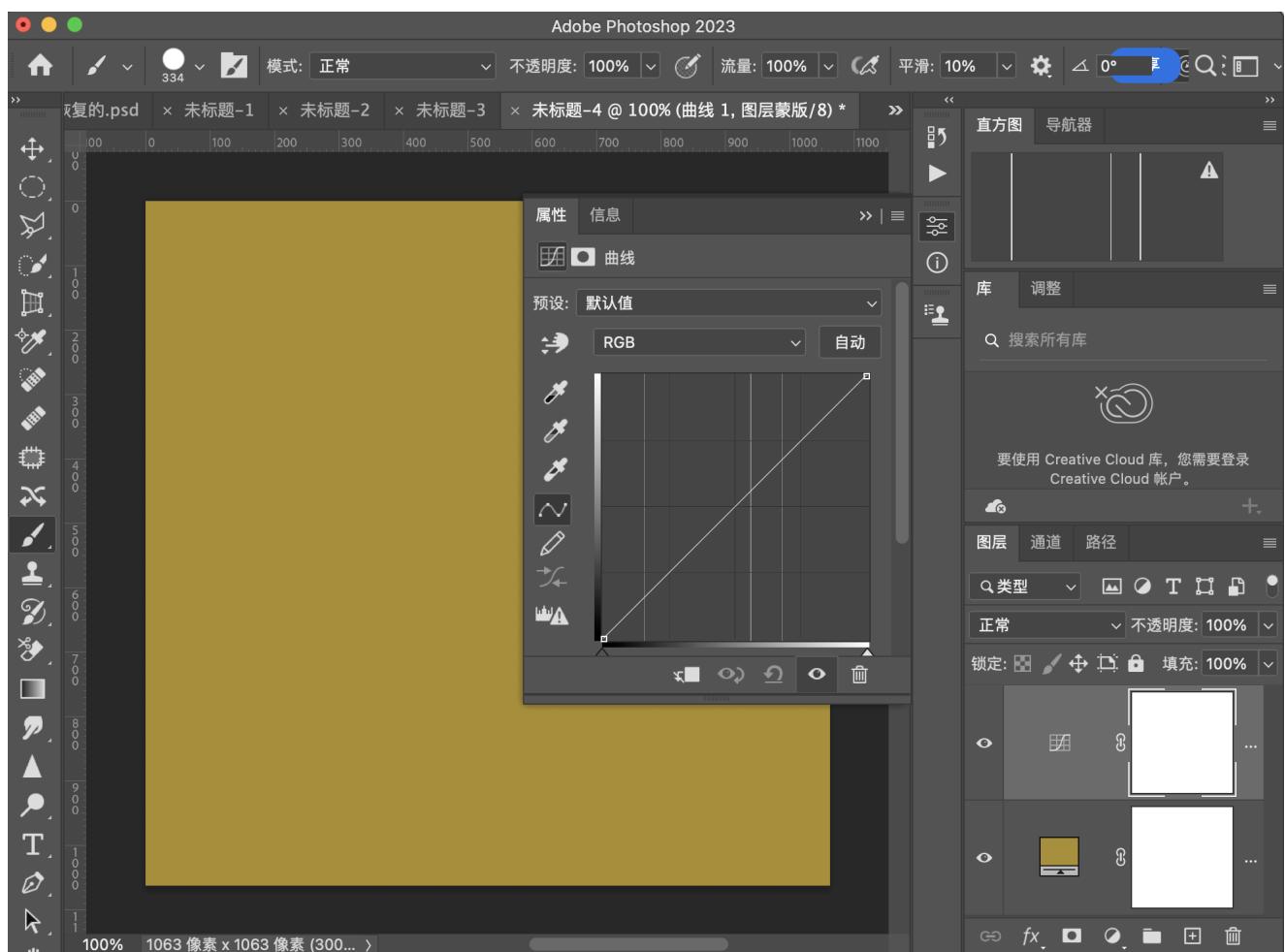
则结果公式可以写作

$\$r=BlendMode(b,Adjustment(b))\$$

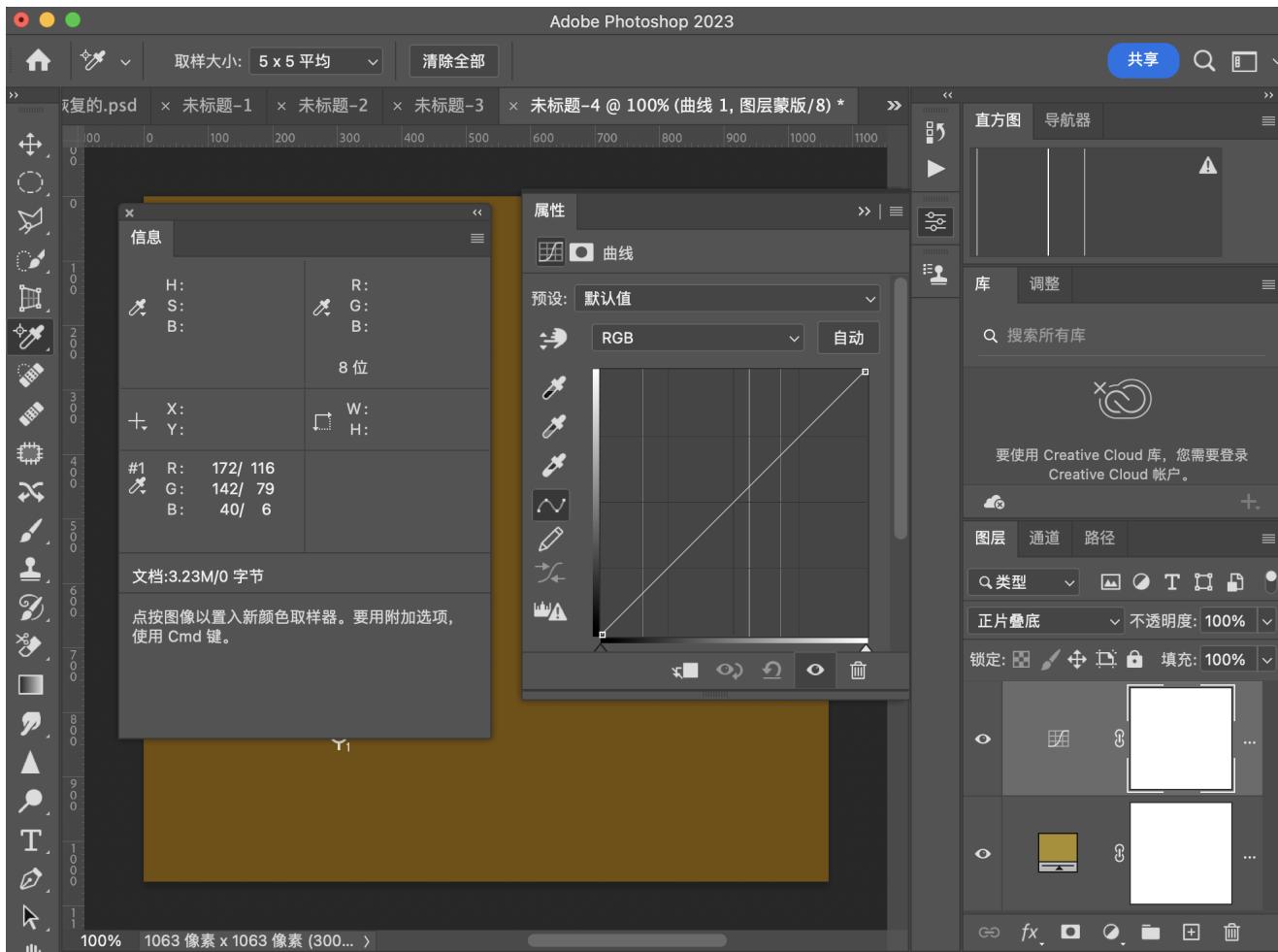
也就是说相当于，先使用调整图层产生基础图层调整之后的图层，再使用调整后的图层和原先的基础图层进行图层混合模式的操作。

此处我们以曲线调整图层为例

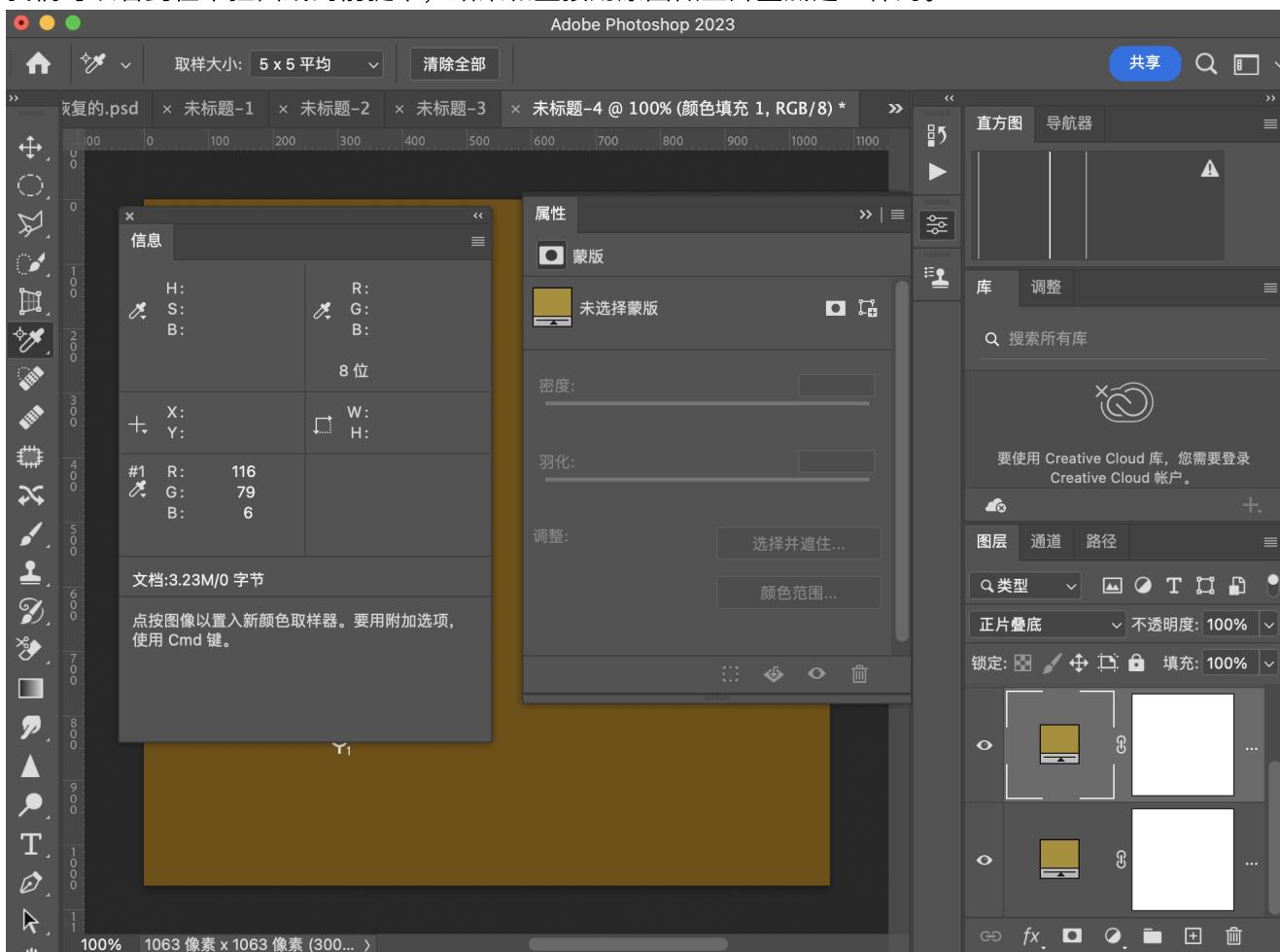
如果我们对原图层新建调整图层，并且对调整图层使用图层混合模式



如果是正常模式，则相当于原图，但是只要我们选择一个别的模式，或拉一下曲线，结果就会不同。



我们可以看到在不拉曲线的前提下，结果和直接用原图做正片叠底是一样的。



参考文档

<http://www.simplefilter.de/en/basics/mixmods.html>

<https://printtechnologies.org/wp-content/uploads/2020/03/pdf-reference-1.6-addendum-blend-modes.pdf>