第一章

Java™ 2D API 概述

Java™ 2D API 增强了抽象窗口工具包（AWT）的图形、文本和图像功能，可以开发更为强大的用户接口和新型的 Java 应用程序。

除了更为强大的图形、字体和图像 API 外，Java 2D API 还改进了颜色的定义与复合及对任意几何形状和文本的选中检测，并为打印机和显示设备提供了统一的绘制模式。

Java 2D API 还可以创建高级图形库（例如 CAD-CAM 库和图形或图像特殊效果库），并可创建图像和图形文件读/写过滤器。

当与 Java 媒体框架（JMF）和其它 Java 媒体应用程序配合使用时，Java 2D API 还可用来创建和显示动画和其它多媒体演示稿。Java 动画和 Java 媒体框架 API 依赖 Java 2D API 提供支持。

1.1 图形、文本和图像增强功能

早期版本的 AWT 提供了简单的绘制包，因此仅适用于普通的 HTML 页。对于复杂的图形、文本或图像，其功能却不够全面。作为一种简化的绘制包，早期版本的 AWT 仅可实现许多常规绘制概念的特例。Java 2D API 对 AWT 进行了扩展，提供了更加灵活、功能更全面的绘制包，使其支持更多的一般图形和绘制操作。

例如，用户通过 Graphics 类可以画矩形、椭圆和多边形。Graphics2D 还提供了一种可绘制任意几何形状的机制，从而增强了几何绘制功能。类似地，利用 Java 2D API 还可绘制任意宽度的线条，并可用任意纹理填充几何形状。

几何形状是通过 Shape 接口的实现提供的（例如 Rectangle2D 和 Ellipse2D）。曲线和弧也是 Shape 的特定实现。

填充和画笔样式由 Paint 和 Stroke 接口的实现提供（例如 BasicStroke、GradientPaint、TexturePaint 和 Color）。

AffineTransform 定义二维坐标的线性转换，包括缩放、平移、旋转和修剪。

剪切区域由用来定义一般剪切区域的 Shape 接口的同一实现来定义（例如 Rectangle2D 和 GeneralPath）。

颜色复合由 Composite 接口的实现提供（例如 AlphaComposite）。

Font 由 Glyphs 集定义，而 Glyphs 集由单个 Shape 定义。

1.2 绘制的模型

Java 2D API 新增的功能中并未改变基本图形绘制模型。绘制图形时，应设置图形上下文，然后调用 Graphics 对象的绘制方法。

Java 2D API 类 Graphics2D 对 Graphics 进行了扩展，可支持更多的图形属性，并可提供新的绘制方法。第 15 页的“用 Graphics2D 绘制”中说明了设置 Graphics2D 上下文的方法。

Java 2D API 自动补偿不同绘制设备之间的差异，可为不同类型设备提供统一的绘制模型。在应用程序级上，无论目标绘制设备是显示器还是打印机，绘制过程都是相同的。

1.2.1 坐标系统

Java 2D API 有两种坐标系统：

- *用户空间*是一种与设备无关的逻辑坐标系统。应用程序独占使用此坐标系统。所有传入 Java 2D 绘制例程中的几何图形均在用户空间中定义。
- *设备空间*是一种与设备有关的坐标系统，它根据目标绘制设备的不同而变化。

Java 2D 系统自动在用户空间和目标绘制设备的设备空间之间执行必要的转换。虽然显示器的坐标系统和打印机的坐标系统有很大的差别，但这些差别对应用程序并无影响。

1.2.1.1 用户空间

如图 1-1 所示，用户空间的原点位于空间左上角，*x* 值向右递增，*y* 值向下递增。

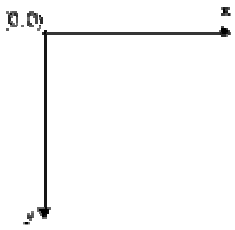


图 1-1 用户空间坐标系统

用户空间是对所有可能的设备坐标系统的一种抽象。某一设备的设备空间与用户空间的原点和方向可能相同，也可能不同。无论怎样，当绘制图形对象时，用户空间坐标将自动变换为相应的设备空间。通常由内部平台设备驱动程序执行这项转换。

1.2.1.2 设备空间

Java 2D API 定义了三级配置信息，用来支持从用户空间到设备空间的转换。该信息封装在三个类中：

- GraphicsEnvironment
- GraphicsDevice
- GraphicsConfiguration

GraphicsEnvironment 描述了特定平台上对 Java 应用程序可见的绘制设备集。绘制设备包括屏幕、打印机和图像缓冲区。GraphicsEnvironment 还包括平台上所有可用字体的列表。

GraphicsDevice 描述了对应用程序可见的绘制设备（例如屏幕或打印机）。该设备每一种可能的配置都由一个 GraphicsConfiguration 表示。例如，SVGA 显示设备可以在以下几种模式下操作：640x480x16 色、640x480x256 色和 800x600x256 色。SVGA 屏幕由 GraphicsDevice 对象表示，每种模式由一个 GraphicsConfiguration 对象表示。

每个 GraphicsEnvironment 可以包含一个或多个 GraphicsDevices；同样，每个 GraphicsDevice 也可以包含一个或多个 GraphicsConfigurations。

其中，GraphicsEnvironment、GraphicsDevice 和 GraphicsConfiguration 提供定位 Java 平台绘制设备或字体的信息，同时也提供从用户空间到设备空间坐标转换时所需的所有信息。应用程序可以访问该信息，但无需执行用户空间和设备空间之间的任何变换。

1.2.2 变换

Java 2D API 有一种统一的坐标变换方式。所有坐标变换（包括从用户到设备空间的变换）都由 AffineTransform 对象表示。AffineTransform 定义了用矩阵绘制坐标的规则。

可以将 AffineTransform 添加到图形上下文中，以便旋转、平移或修剪需要绘制的几何形状、文本或图像。加入的变换将应用到上下文中的任意图形对象中。用户空间坐标向设备空间坐标转换时将执行此变换。

1.2.3 字体

字符串通常按组成字符串的字符进行考虑。绘制字符串时，它的外观由所选字体决定。然而，被字体用来显示字符的形状并不一定适合每个字符。例如，在专业出版物中，两个或多个字符的某种组合通常被一种称为*连字*的单个字符取代。

被字体用来表示字符串中字符的形状称为*字形*。字体可能会用多个字形表示一个字符（例如 acute 中的小写 *a*），或用一个字形表示某种字符组合（例如 final 中的 *fi*）。在 Java 2D API 中，字形与其它 shape（形状） 相同，也是一种 shape，可以进行同样的操作和绘制。

*字体*可以看作字形的集合。一种字体可以有多种形式，例如大号、中号、倾斜、哥特式和正常。 这些不同的形式被称为*字样*。一种字体的所有字样都有类似的排版设计，可以看作同一*系列*的成员。换言之，带有特殊样式的字形集组成字样，字样的集合组成一个字体系列，而字体系列的集合组成特定 GraphicsEnvironment 内的可用字体集。

在 Java 2D API 中，字体用字样来描述 -- 例如 Helvetica Bold。这与 JDK 1.1 软件不同。在 JDK 1.1 软件中，字体用映射到不同字样（与特定平台上的可用字体有关）的逻辑名来描述。为保证向后兼容性，Java 2D API 既支持字样名的字体规范，也支持逻辑名的字体规范。

利用 Java 2D API，用户可以制作和绘制包含不同系列、字样、大小甚至不同语种等的多种字体的字符串。文本的外观在逻辑上与文本布局相独立。Font 对象用来描述 TextLayout 和 TextAttributeSet 对象中存储的外观和布局信息。将字体和布局信息分开存放将便于在不同布局配置中使用相同字体。

1.2.4 图像

图像是按空间位置组织的像素集合。*像素*则定义某个显示位置的图像外观。像素的二维数组称为*光栅*。

像素的外观既可直接定义，也可作为图像颜色表的索引进行定义。

在包含多种颜色的图像中（多于 256 种），像素通常直接表示每个屏幕位置的颜色、alpha 和其它显示特性。这种图像通常比索引颜色图像大得多，但看上去更为真实。

在索引颜色图像中，图像中的颜色仅限于颜色表中指定的颜色，因而图像中可用的颜色变少。然而，索引通常比颜色值占用更少的存储空间，因此以索引颜色集方式存储的图像通常较小。这种像素格式通常用于 16 或 256 色图像。

Java 2D API 中的图像主要有两个组件：

- 原始图像数据（像素）
- 解释像素的必要信息

解释像素的规则封装在 ColorModel 对象中。例如，值是应解释为直接颜色还是应解释为索引颜色。要显示某一像素，则必须有一种颜色模型与之相配。

带是图像颜色空间的一个组件。例如，红色、绿色和蓝色组件是 RGB 图像中的三个带。直接颜色模型图像中的像素可以看作单个屏幕位置的带值的集合。

java.awt.image 包中包含若干 ColorModel 实现，包括用于打包的实现和组件像素表示。

ColorSpace 对象中封装了一些控制一组数值度量如何与某种颜色相关联的规则。java.awt.color 中的 ColorSpace 实现提供了最常用的颜色空间，包括 RGB 和灰度。请注意，颜色空间不是颜色的集合，它只定义了如何解释单个颜色值的规则。

颜色空间与颜色模型的分离，为颜色表示方式及颜色表示之间的转换带来了更大的灵活性。

1.2.5 填充和笔画

有了 Java 2D API，就可以用不同的画笔样式和填充图案绘制 shape。由于文本最终是由字形集合表示的，因此文本字符串也可进行绘制和填充。

画笔样式由实现 Stroke 接口的对象定义。笔画的引入使用户可以为直线和曲线指定不同宽度和短划线图案。

填充图案由实现 Paint 接口的对象定义。早期版本 AWT 中的 Color 类是 Paint 对象的一种简单类型，用来定义纯色填充。Java 2D API 提供了另外两种 Paint 实现：TexturePaint 和 GradientPaint。TexturePaint 利用重复图像片段的方式定义一种填充图案，而 GradientPaint 则定义在两种颜色间渐变的填充图案。

在 Java 2D 中，绘制形状的轮廓和用图案填充颜色是两个独立的操作。

- 使用一种 draw 方法即可绘制由 Stroke 属性指定的形状轮廓或轮廓线及由 Paint 属性指定的填充图案。
- 使用 fill 方法可以用 Paint 属性指定的图案填充形状内部。

绘制文本字符串时，当前 Paint 属性将应用到组成该字符串的字形中。但注意，实际上是由 drawString 来填充所绘制的字形。要绘制文本字符串中字形的轮廓线，需要找出轮廓线并利用 draw 方法用与形状相同的方式对其进行绘制。

1.2.6 复合

当绘制的对象与已有对象重叠时，需要确定如何将新对象的颜色与所要绘制的已有区域的颜色相复合。Java 2D API 在 Composite 对象中封装了颜色复合规则。

基本绘制系统仅提供复合颜色的基本布尔运算符。例如，布尔复合规则可能允许源颜色值和目标颜色值为 ANDed、ORed 或 XORed。这种方法有一些问题。

- 它不具有“用户友好性”-- 当红色和蓝色进行 ANDed 操作而不是相加时，很难判断得到什么颜色。
- 布尔复合不支持不同颜色空间的精确颜色复合。
- 直接布尔复合不考虑颜色的颜色模型。例如，在索引颜色模型中，图像中两个像素值的布尔运算结果是两个索引的复合，而不是两种颜色的复合。

Java 2D API 通过使用 alpha-blending [1] 规则（进行复合颜色时考虑颜色模型信息）而避免了这些缺陷。 AlphaComposite 对象中包括源颜色和目标颜色的颜色模型。

1.3 向后兼容性和平台无关性

Java 2D API 保持了与 JDK 1.1 软件的向后兼容性。其体系结构的设计也可确保应用程序的平台无关性。

1.3.1 向后兼容性

为了保证向后兼容性，原有 JDK 图形和图像处理功能的类和接口被保留了下来。已有功能并未删除，也未改变已有类的包命名。通过实现已有类中的新方法、扩展已有类以及在不影响原 API 的前提下添加新的类和接口，Java 2D API 增强了 AWT 的功能。

例如，许多 Java 2D API 功能均是通过扩展的图形上下文 Graphics2D 来实现的。为了在保持向后兼容性的前提下提供此扩展的图形上下文，Graphics2D 扩展了 JDK 1.1 版本的 Graphics 类。

图形上下文的用法模型仍保持不变。AWT 通过以下方法将图形上下文传给 AWT 组件：

- paint
- paintAll
- update
- print
- printAll
- getGraphics

JDK 1.1 applet 将对以 Graphics 实例形式传入的图形上下文进行解释。为了使用 Graphics2D 中实现的新功能，我们提供了一个与 Java 2D API 兼容的 applet 对图形上下文进行强制类型转换，使其成为 Graphics2D 对象：

```
public void Paint (Graphics g) {       Graphics2D g2 = (Graphics2D) g;       ...     ...
```

在某些情况下，Java 2D API 对原有类进行了广义化，但没有进行扩展。对原有类进行广义化的技术有两种：

- 在层次中插入一个或多个父类，然后更新原有类以扩展新的父类。此项技术用来将一般实现的方法和实例数据添加到原有类中。
- 向原有类中添加一个或多个接口实现。此项技术用来将一般抽象方法添加到原有类中。

例如，Java 2D API 使用这两种技术对 AWT Rectangle 类进行广义化。矩形的层次如下所示：

```
java.lang.object|+-------java.awt.geom.RectangularShape                    |
+---------java.awt.geom.Rectangle2D                                        |
+-------java.awt.Rectangle
```

在 JDK 1.1 软件中，Rectangle 仅扩展 Object。现在它还可扩展新的 Rectangle2D 类，并可实现 Shape 和 Serializable。Rectangle 层次中新增了两个父类：RectangularShape 和 Rectangle2D。为 JDK 1.1 软件编写的 applet 并不知道新的父类和接口实现，但由于 Rectangle 仍包含早期版本中的方法和成员，因此将不受影响。

Java 2D API 在原 API 中添加了若干与之“直交”的新类和接口。这些增补不是对已有类进行扩展或广义化。它们与已有类完全不同。这些新类和接口使原 API 中没有明确表示法的概念更加具体化。

例如，Java 2D API 实现了若干新 Shape 类，包括 Arc2D、CubicCurve2D 和 QuadCurve2D。虽然 AWT 的早期版本可以使用 drawArc 和 fillArc 方法绘制弧，但并无一般曲线抽象和各个类来使弧具体化。将这些离散的类添加到 Java 2D API 中并不会扰乱原 applet，因为 drawArc 和 fillArc 仍受 Graphics 类的支持。

1.3.2 平台无关性

为了促进平台无关性应用程序的发展，Java 2D API 不考虑目标绘制设备的分辨率、颜色空间或颜色模型。Java 2D API 也不考虑特定的图像文件格式。

只有当字体是内置的（作为 JDK 软件的一部分），或是由数学方式或程序生成时，才可能实现真正的平台无关性。Java 2D API 当前不支持内置或数学方式生成的字体，但它支持全部字体通过其字形集进行程序定义。同时，每种字形可以通过由线段和曲线组成的 Shape 进行定义。 从一个字形集中可以导出许多特殊样式和大小的字体。

1.4 Java 2D API 包

Java 2D API 类分为以下几个包：

- java.awt
- java.awt.geom
- java.awt.font
- java.awt.color
- java.awt.image
- java.awt.image.renderable
- java.awt.print

包 java.awt 中包含一般的或比原有类增强的 Java 2D API 类和接口（显然，并非所有 java.awt 中的类都是 Java 2D 类）。

AlphaComposite	BasicStroke	Color
Composite	CompositeContext	Font
GradientPaint	Graphics2D	GraphicsConfiguration
GraphicsDevice	GraphicsEnvironment	Paint
PaintContext	Rectangle	Shape
Stroke	TexturePaint	Transparency

包 java.awt.geom 中包含与几何元素定义有关的类和接口：

AffineTransform	Arc2D	Arc2D.Double
Arc2D.Float	Area	CubicCurve2D
CubicCurve2D.Double	CubicCurve2D.Float	Dimension2D
Ellipse2D	Ellipse2D.Double	Ellipse2D.Float
FlatteningPathIterator	GeneralPath	Line2D
Line2D.Double	Line2D.Float	PathIterator
Point2D	Point2D.Double	Point2D.Float
QuadCurve2D	QuadCurve2D.Double	QuadCurve2D.Float
Rectangle2D	Rectangle2D.Double	Rectangle2D.Float
RectangularShape	RoundRectangle2D	RoundRectangle2D.Double
RoundRectangle2D.Float		

许多几何元素都有相应的 .Float 和 .Double 实现。这样就可以启用浮点数单、双精度实现。双精度实现的绘制精度较高，但在某些平台上会降低性能。

包 java.awt.font 中包含用于文本布局和字体定义的类和接口：

FontRenderContext	GlyphJustificationInfo	GlyphMetrics
GlyphVector	GraphicAttribute	ImageGraphicAttribute
LineBreakMeasurer	LineMetrics	MultipleMaster
OpenType	ShapeGrapicAttribute	TextAttribute

TextHitInfo	TextLayout	TransformAttribute

包 java.awt.color 中包含用于颜色空间定义和颜色监视的类和接口：

ColorSpace	ICC_ColorSpace	ICC_Profile
ICC_ProfileGray	ICC_ProfileRGB	

java.awt.image 和 java.awt.image.renderable 包中包含用于图像定义和绘制的类和接口：

AffineTransformOp	BandCombineOp	BandedSampleModel
BufferedImage	BufferedImageFilter	BufferedImageOp
ByteLookupTable	ColorConvertOp	ColorModel
ComponentColorModel	ComponentSampleModel	ConvolveOp
ContextualRenderedImageFactory		DataBuffer
DataBufferByte	DataBufferInt	DataBufferShort
DataBufferUShort	DirectColorModel	IndexColorModel
Kernel	LookupOp	LookupTable
MultiPixelPackedSampleModel	PackedColorModel	ParameterBlock
PixelInterleavedSampleModel	Raster	RasterImageConsumer
RasterOp	RenderableImage	RenderableImageOp
RenderableImageProducer	RenderContext	RenderedImageFactory
RenderedImage	RescaleOp	SampleModel
ShortLookupTable	SinglePixelPackedSampleModel	TileObserver
WritableRaster	WritableRenderedImage	

包 java.awt.image 在 AWT 的早期版本中存在。Java 2D API 对下列原有 AWT 图像类进行了功能增强：

- ColorModel
- DirectColorModel

- IndexColorModel

java.awt.image 包中保留了这些颜色模型类，以保证向后兼容性。为保证一致性，新的颜色模型类也位于 java.awt.image 包中。

包 java.awt.print 中包含用于打印所有基于 Java 2D 的文本、图形和图像的类和接口。

Book	Pageable	PageFormat
Paper	Printable	PrinterGraphics
PrinterJob		

第二章

用 Graphics2D 绘制

Graphics2D 扩展了 java.awt.Graphics，以便对形状、文本和图像的展示提供更加完善的控制。Java 2D 绘制进程是通过 Graphics2D 对象及其状态属性来控制的。

在绘制图形对象时，Graphics2D 状态属性（如线条样式和变换）将应用于图形对象。与 Graphics2D 有关的状态属性集被称为 Graphics2D 上下文。要绘制文本、形状或图像，请设置 Graphics2D 上下文，然后调用一种 Graphics2D 绘制方法（例如 draw 或 fill）。

2.1 接口和类

下表列出了与 Graphics2D 上下文一起使用的接口和类，包括代表状态属性的类。这些类中的大多数都是 java.awt 包的组成部分。

接口	说明
Composite	定义基本图形元素和底层图形区域的复合方法。由 AlphaComposite 实现。
CompositeContext	定义复合操作的封装和优化环境。用于程序员实现自定义复合规则。
Paint	扩展：Transparency 定义 draw 或 fill 操作的颜色。由 Color、GradientPaint 和 TexturePaint 实现。
PaintContext	定义绘画操作的封装和优化环境。程序员用来实现自定义绘画操作。
Stroke	生成 Shape 并包含要绘制的 Shape 的轮廓。由 BasicStroke 实现。

类	说明
AffineTransform (java.awt.geom)	代表 2D 仿射变换，用于执行从一种二维坐标到其它二维坐标的线性映射。
AlphaComposite	实现：Composite 实现形状、文本和图像的基本 alpha 复合规则。
BasicStroke	实现：Stroke 定义应用于 Shape 轮廓的“画笔样式”。
Color	实现：Paint 定义 Shape 的实心填充颜色。
GradientPaint	实现：Paint 定义 Shape 的线性颜色渐变填充图案。该填充图案将从点 P1 的 C1 颜色逐渐变为点 P1 的 C2 颜色。
Graphics2D	扩展：Graphics 2D 绘制的基本类。是原来的 java.awt.Graphics 类的扩展。
TexturePaint	实现： Paint 定义 Shape 的纹理或图案填充。纹理或图案由 BufferedImage 生成。

2.2 绘制的概念

要使用 Java 2D API 绘制图形对象，请设置 Graphics2D 上下文，并将该图形对象传给 Graphics2D 绘制方法。

可以修改组成 Graphics2D 上下文的状态属性，以：

- 改变笔画宽度。
- 改变笔画的连接方式。
- 设置剪切路径以限定绘制区域。
- 在绘制时平移、旋转、缩放或修剪对象。
- 定义用来填充形状的颜色和图案。
- 指定复合多个图形对象的方式。

Graphics2D 定义了几种方法，用于添加或改变图形上下文中的属性。其中的大多数都采用某一代表特定属性的对象，例如 Paint 或 Stroke 对象。

Graphics2D 上下文持有对这些属性的*引用*：它们并不会被复制。如果改变作为 Graphics2D 上下文一部分的属性对象，则需要调用相应的 set 方法来通知上下文。在绘制期间修改属性对象将导致不可预测且可能不稳定的行为。

2.2.1 绘制过程

当绘制图形对象时，几何形状、图像和属性信息将组合起来，以计算显示时必须改变哪些像素值。

Shape 的处理过程可分为四步：

1. 如果要绘制 Shape，则 Graphics2D 上下文中的 Stroke 属性将用于生成包含绘制路径的新 Shape。
2. 根据 Graphics2D 上下文中的 transform 属性，将 Shape 路径的坐标从用户空间坐标变换成设备空间坐标。
3. 使用 Graphics2D 上下文中的 clip 属性剪切 Shape 的路径。
4. Shape 的其余部分（如果有）使用 Graphics2D 上下文中的 Paint 和 Composite 属性填充。

绘制文本类似于绘制 Shape，因为文本是按单个的字形来绘制的，而每种字形都是一个 Shape。其唯一差别是 Java 2D API 必须在绘制前确定对文本应用什么 Font，并且在处理前从 Font 中获取相应的字形。

图像的处理方式则有所不同，即变换和剪切操作都在图像的边框内执行。颜色信息取自于图像本身，并且在将图像像素复合到处理表面上时，该图像的 alpha 通道将与当前 Composite 属性联合使用。

2.2.2 控制绘制质量

Java 2D API 允许用户选择是尽快绘制对象，还是倾向于以尽可能高的质量绘制对象。通过指定 Graphics2D 上下文中的 RenderingHints 属性来反映用户的选择。并非所有的平台都支持对绘制模式的修改，因此指定绘制模式的建议并不能保证一定会被使用。

RenderingHints 类支持下列类型的建议：

- Alpha interpolation--可设置为缺省、质量或速度。
- Antialiasing--可设置为缺省、打开或关闭。
- Color Rendering--可设置为缺省、质量或速度。
- Dithering--可设置为缺省、禁用或启用。
- Fractional Metrics--可设置为缺省、打开或关闭。
- Interpolation--可设置为最近邻域、双线性或双三次。
- Rendering--可设置为缺省、质量或速度。
- Text antialiasing--可设置为缺省、打开或关闭。

要设置或改变 Graphics2D 上下文中的 RenderingHints 属性，请调用 setRenderingHints。当提示设置为缺省值时，将使用平台的绘制缺省值。

反走样
当基本图形在光栅图形显示设备上显示时，其边缘可能会由于*走样*而参差不齐。因为圆弧和对角线是通过打开距离该曲线路径最近的像素进行逼近而得到的，因此它们可能呈现锯齿状。这种问题在低分辨率设备上尤其显著，其中锯齿边缘与水平线或垂直线的平滑边缘形成了强烈的对比。 *反走样*是一种用于平滑边缘显示对象的技术。它不是简单地显示距离线条或曲线最近的像素，而是还按要显示的几何形状所覆盖范围的比例来设置周围像素的亮度，从而软化了边缘并且将开-关转变延伸到多个像素上。但是，反走样需要更多的计算资源，而且会降低处理速度。    走样　　反走样

2.2.3 Stroke 属性

绘制 Shape（如 GeneralPath 对象）等于沿着 GeneralPath 的线段移动一支逻辑画笔。Graphics2D 的 Stroke（画笔）属性定义了该画笔所绘制标记的特性。

BasicStroke 对象用于定义 Graphics2D 上下文的 stroke 属性。BasicStroke 定义的特性包括线条宽度、笔形样式、线段连接样式和短划线图案等。要设置或改变 Graphics2D 上下文中的 Stroke 属性，请调用 setStroke。

图 **2-1** BasicStroke 支持的笔形样式

图 **2-2** BasicStroke 支持的连接样式

例如，图 2-3 中的第一幅图像使用斜角连接样式；第二幅图像使用圆角连接样式、圆角笔形样式和一种短划线图案。

图 **2-3** Stroke 样式

使用 Stroke 属性的 Graphics2D 绘制方法包括 draw、drawArc、drawLine、drawOval、drawPolygon、drawPolyline、drawRect 和 drawRoundRect。当调用这些方法时，将绘制指定 Shape 的轮廓。Stroke 属性定义线条特性，而 Paint 属性定义画笔所绘标记的颜色或图案。

例如，调用 draw(myRectangle) 时：

1.Stroke 属性将应用于矩形轮廓。
2.绘制的轮廓将转换为 Shape 对象。
3.Paint 属性将应用于 Shape 轮廓内的像素。

图 2-4 例示了该过程：

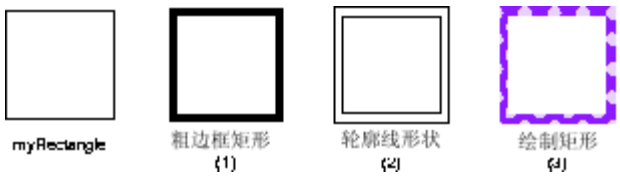


图 **2-4** 绘制形状

2.2.4 Fill 属性

Graphics2D 上下文中的 fill（填充）属性用 Paint 对象表示。可以通过调用 setPaint 来将 Paint 添加到 Graphics2D 上下文中。

在绘制 Shape 或字形（Graphics2D.draw、Graphics2D.drawString）时，Paint 属性将应用于代表该对象绘制轮廓的 Shape 内的所有像素。填充 Shape （Graphics2D.fill）时，Paint 属性将应用于 Shape 轮廓内的像素。

使用 setColor 方法可以设置简单的纯色填充。Color 是 Paint 接口的最简单实现。

要使用更复杂的图形样式（例如渐变和纹理）填充 Shape，请使用 Java 2D Paint 类的 GradientPaint 和 TexturePaint。这些类消除了相当耗时的使用简单纯色图形创建复杂填充的任务。图 2-5 例示了这两种填充，它们可由 GradientPaint 和 TexturePaint 进行定义。

图 **2-5** 复杂填充样式

当调用 fill 绘制 Shape 时，系统将：

1. 确定组成该 Shape 的像素。
2. 从 Paint 对象中获得每个像素的颜色。
3. 将颜色转换为输出设备上的相应像素值。
4. 将像素写到输出设备上。

批处理
为使像素处理成为流水作业，Java 2D API 将成批地处理像素。批任务可以是给定扫描线上的相邻像素集，也可以是像素块。批处理分两步完成： 1. 调用 Paint 对象的 createContext 方法创建 PaintContext。 PaintContext 将储存有关当前处理操作的上下文信息和生成颜色所需的信息。createContext 方法传递要在用户空间和设备空间中填充的图形对象的边框（其中生成颜色的 ColorModel）和用于将用户空间映射到设备空间的变换。ColorModel 被视为建议，因为并非所有的 Paint 对象都支持任意的 ColorModel（有关 ColorModels 的详细信息，参见第 89 页的“颜色”）。

2.调用 getColorModel 方法,以便从 PaintContext 获取生成图形颜色的 ColorModel。

然后重复调用 getRaster 方法，以获得包含每个批任务的实际颜色数据的 Raster。 此信息被传给处理流水线的下一阶段，在该阶段中将使用当前 Composite 对象绘制生成的颜色。

2.2.5 剪切路径

*剪切路径*标识了 Shape 或 Image 中需处理的部分。当剪切路径是 Graphics2D 上下文的一部分时，只有位于该路径之内的 Shape 或 Image 部分才会得到处理。

要将剪切路径添加到 Graphics2D 上下文，请调用 setClip。任何 Shape 都可用来定义剪切路径。

要改变剪切路径，可以用 setClip 指定新路径，也可调用 clip 将剪切路径改变为原剪切路径与一个新 Shape 的相交部分。

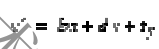
2.2.6 变换

Graphics2D 上下文包含变换，用于在绘制期间将对象从用户空间变换到设备空间。要执行其它变换（例如旋转或缩放），请将其它变换添加到 Graphics2D 上下文中。这些额外的变换将成为绘制期间变换流水线的一部分。

Graphics2D 提供了多种 Graphics2D 上下文变换的修改方法。最简单的方法是调用下列 Graphics2D 变换方法：rotate（旋转）、scale（缩放）、shear（修剪）或 translate（平移）。用户可以指定想要在绘制期间应用的变换，Graphics2D 将自动做出相应改变。

也可明确地将某一 AffineTransform 与当前 Graphics2D 变换相*连接*。AffineTransform 对一系列基本图形执行线性变换（例如平移、缩放、旋转或修剪）。当某一变换与已有变换连接时，最后指定的变换将*首先*应用。要将某一变换与当前变换相连接，请向 Graphics2D.transform 传递 AffineTransform。

Graphics2D 还提供了一种将 AffineTransform 作为参数接受的 drawImage 版本。因此，用户可以在图像绘制过程中应用变换，而不必永久修改变换流水线。绘制图像时，就象已经将该变换与 Graphics2D 上下文中的当前变换相连接。

仿射变换
Java 2D API 提供了一个变换类，即 AffineTransform。AffineTransforms 用于在处理文本、形状和图像时对其进行变换。还可对 Font 对象应用变换以创建新的衍生字体，说明见第 65 页的“创建衍生字体”。 仿射变换是对一系列基本图形执行线性变换。它总是将直线变换成直线，平行线变换成平行线；但是，点与点之间的距离和非平行线之间的角度可能会改变。 仿射变换建立在如下形式的二维矩阵基础上：      其中      和      变换可以组合，从而创建可应用于对象的变换系列或*流水线*。这种组合称为*连接*。当某一变换与已有变换（例如 AffineTransform）连接时，最后指定的变换将*首先*应用。变换还可以与已有变换进行*预连接*。这种情况下，最后指定的变换将*最后*应用。 预连接用于执行相对于设备空间（而不是用户空间）的变换。例如，可以使用 AffineTransform.preConcatenate 来执行相对于绝对像素空间的平移。

2.2.6.1 构造 AffineTransform

AffineTransform 提供了一套用于构造 AffineTransform 对象的便利方法：

- getTranslateInstance
- getRotateInstance
- getScaleInstance
- getShearInstance

要使用这些方法，请指定所要创建的变换的特性，同时 AffineTransform 将生成相应的变换矩阵。也可通过直接指定变换矩阵的元素来构造 AffineTransform。

2.2.7 Composite 属性

当两个图形对象重叠时，有必要确定使用哪些颜色来处理重叠像素。例如，如果红色矩形与蓝色矩形相重叠，则其重叠区域应该处理为红色、蓝色或二者的某种组合。重叠区域中像素的颜色决定了哪一个矩形位于上面及其透明程度。确定使用何种颜色处理重叠对象所共有像素的过程称为*复合*。

两个接口构成了 Java 2D 复合模型的基础：Composite 和 CompositeContext。

要指定应该使用的复合样式，请调用 setComposite，将 AlphaComposite 对象添加到 Graphics2D 上下文中。AlphaComposite（Composite 接口的一种实现）支持大量不同的复合样式。该类的实例包含了描述如何混合新颜色与已有颜色的复合规则。

在 AlphaComposite 类中，最常用的复合规则是 SRC_OVER，表示混合时新颜色（源色）应覆盖在已有颜色（目标色）之上。

AlphaComposite 复合规则	说明	示例
CLEAR	清除	
DEST_IN	加入目标色	
DEST_OUT	去除目标色	
DEST_OVER	用目标色覆盖	
SRC	源色	
SRC_IN	加入源色	
SRC_OUT	去除源色	
SRC_OVER	用源色覆盖	

2.2.7.1 透明度管理

颜色的 *alpha* 值是其透明度的一种量度：它用百分比表示在颜色重叠时以前处理过的颜色应显示的量。不透明颜色（alpha=1.0）不能显示任何下层颜色，而透明颜色（alpha=0.0）可以将所有下层颜色显示出来。

在绘制文本和 Shape 时，alpha 值来源于 Graphics2D 上下文中的 Paint 属性。在消除 Shape 和文本的走样时，Graphics2D 上下文中 Paint 属性的 alpha 值将与光栅化路径中的像素覆盖信息相结合。图像保存自己的 alpha 信息--有关详细信息，参见第 26 页的“透明度和图像”。

在构造 AlphaComposite 对象时，可以指定额外的 alpha 值。将该 AlphaComposite 对象添加到 Graphics2D 上下文中时，此额外 alpha 值将增加任何所要处理的图形对象的透明度--每个图形对象的 alpha 值都将乘以 AlphaComposite 的 alpha 值。

2.2.7.2 透明度和图像

图像可以携带本图像中每个像素的透明度信息。该信息称为 *alpha 通道*，用于与 Graphics2D 上下文中的 Composite 对象联合使用，以将该图像与已有图形混合。

例如，图 2-6 包含有三幅具有不同透明度信息的图像。每种情况下，图像都将显示在蓝色矩形之上。此例假定 Graphics2D 上下文包含使用 SRC_OVER 作为复合操作的 AlphaComposite 对象。

图 **2-6** 透明度和图像

在第一幅图像中，所有像素都是完全不透明（狗的身体）或完全透明的（背景）。这种效果经常用于网页上。在第二幅图像中，狗身体中的所有像素都使用统一的透明 alpha 值处理，允许显示出蓝色背景。在第三幅图像中，狗脸周围的像素是完全不透明的（alpha=1.0），但是随着离狗脸距离的增加，像素的 alpha 值递减。

2.3 设置 Graphics2D 上下文

要配置 Graphics2D 上下文以进行绘制，应使用 Graphics2D set 方法来指定属性，例如 RenderingHints、Stroke、Paint、剪切路径、Composite 和 Transform。

2.3.1 设置绘制建议

RenderingHints 对象封装了与绘制对象有关的所有参数。要设置 Graphics2D 上下文中的绘制建议，请创建 RenderingHints 对象并将它传递给 Graphics2D.setRenderingHints。

设置绘制建议并不能保证一定会使用特定的绘制算法：并非所有平台都支持对绘制模式的修改。

下例中，我们启用了反走样并且将质量设置成绘制的首选项：

```
qualityHints = new
               RenderingHints(RenderingHints.KEY_ANTIALIASING,
               RenderingHints.VALUE_ANTIALIAS_ON);
qualityHints.put(RenderingHints.KEY_RENDERING,
               RenderingHints.VALUE_RENDER_QUALITY);
g2.setRenderingHints(qualityHints);
```


2.3.2 指定 Stroke 属性

BasicStroke 定义了应用于 Shape 轮廓的特性，包括其宽度和短划线图案、线段连接方式及线条末端所用的装饰（如果有）。要设置 Graphics2D 上下文中的 stroke 属性，请创建 BasicStroke 对象并将它传给 setStroke。

2.3.2.1 设置 Stroke 宽度

要设置 stroke 宽度，请创建一个具有适当宽度的 BasicStroke 对象，然后调用 setStroke。

下例中，我们将 stroke 宽度设置成 12 磅，而对连接和笔形修饰则使用缺省值：

```
wideStroke = new BasicStroke(12.0f);
g2.setStroke(wideStroke);
```

2.3.2.2 指定连接和笔形样式

要设置连接和笔形样式，请创建一个具有相应属性的 BasicStroke 对象。

下例中，我们将 stroke 宽度设置成 12 磅，同时使用圆角连接和笔形样式（而不用缺省值）：

```
roundStroke = new BasicStroke(4.0f, BasicStroke.CAP_ROUND,
              BasicStroke.JOIN_ROUND);
g2.setStroke(roundStroke);
```

2.3.2.3 设置短划线图案

使用 BasicStroke 对象可以很容易地定义复杂的短划线图案。创建 BasicStroke 对象时，可以指定两个用于控制短划线图案的参数：

- dash--代表短划线图案的数组。数组中的元素交替代表短划线的尺寸和短划线之间的间距。元素 0 代表第一个短划线，元素 1 代表第一个间距。
- dash_phase--定义短划线图案开始位置的偏移量。

下例中，我们对同一线条应用了两种不同的短划线图案。在前一种线型中，短划线的长度和它们之间的间距为常数。第二种短划线图案要复杂一些，它使用了一个六元素数组来定义短划线图案。

```
float dash1[] = {10.0f};
BasicStroke bs = new BasicStroke(5.0f, BasicStroke.CAP_BUTT,
                 BasicStroke.JOIN_MITER, 10.0f, dash1, 0.0f);
g2.setStroke(bs);
Line2D line = new Line2D.Float(20.0f, 10.0f, 100.0f, 10.0f);
g2.draw(line);

float[] dash2 = {6.0f, 4.0f, 2.0f, 4.0f, 2.0f, 4.0f};
bs = new BasicStroke(5.0f, BasicStroke.CAP_BUTT,
     BasicStroke.JOIN_MITER, 10.0f, dash2, 0.0f);
g2.setStroke(bs);
g2.draw(line);
```

两种短划线图案的短划线相位值都为零，表示从短划线图案的开头部分开始绘制短划线。这两种短划线图案如图 2-7 所示。

图 2-7 短划线图案

2.3.3 指定 **Fill** 属性

Graphics2D 上下文中的 Paint 属性决定了绘制文本和 Shape 时所使用的填充颜色和图案（即 Fill 属性）。

2.3.3.1 使用渐变填充形状

GradientPaint 类提供了使用一种颜色到另一种颜色的渐变填充形状的方法。在创建 GradientPaint 时，可指定开始位置和颜色及结束位置和颜色。填充颜色将沿着两个位置之间的直线成比例地从一种颜色变为另一种颜色，如图 2-8 所示。

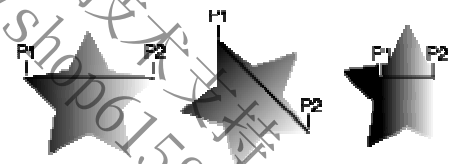


图 2-8 创建渐变填充

图 2-8 的第三颗星中，两个点都位于形状内。沿渐变线所有位于 P1 之外的点使用开始颜色，而沿渐变线所有位于 P2 之外的点都使用结束颜色。

要使用颜色渐变填充形状：

1. 创建 GradientPaint 对象。
2. 调用 Graphics2D.setPaint。
3. 创建 Shape。
4. 调用 Graphics2D.fill(shape)。

下例中，我们用蓝-绿色渐变来填充矩形。

```
GradientPaint gp = new GradientPaint(50.0f, 50.0f, Color.blue
                   50.0f, 250.0f, Color.green);
g2.setPaint(gp);
g2.fillRect(50, 50, 200, 200);
```

2.3.3.2 使用纹理填充形状

TexturePaint 类提供了使用重复图案填充形状的方法。创建 TexturePaint 时，应指定 BufferedImage 作为重复图案。也可为构造函数传递一个矩形来定义图案的重复频率，如图 2-9 所示。

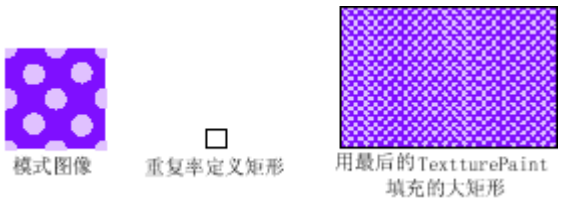


图 2-9 创建纹理画图

要使用纹理填充形状：

1.创建 TexturePaint 对象。
2.调用 Graphics2D.setPaint。
3.创建 shape。
4.调用 Graphics2D.fill(shape)。

下例中，我们将使用由缓冲区图像创建的简单纹理来填充矩形。

```
// 创建大小为 5x5 的缓冲区纹理图像片
BufferedImage bi = new BufferedImage(5, 5,
                        BufferedImage.TYPE_INT_RGB);
Graphics2D big = bi.createGraphics();
// 绘制 BufferedImage 图形以创建纹理
big.setColor(Color.green);
big.fillRect(0,0,5,5);
big.setColor(Color.lightGray);
big.fillOval(0,0,5,5);

// 由缓冲区图像创建纹理画图
Rectangle r = new Rectangle(0,0,5,5);
TexturePaint tp = new
TexturePaint(bi,r,TexturePaint.NEAREST_NEIGHBOR);

// 将纹理画图添加到图形上下文中。
g2.setPaint(tp);

// 创建并绘制使用该纹理填充的矩形。
g2.fillRect(0,0,200,200);
}
```

2.3.4 设置剪切路径

要定义剪切路径：

1.创建一个 Shape，代表想要绘制的区域。
2.调用 Graphics2D.setClip，将该形状用作 Graphics2D 上下文的剪切路径。

要缩小剪切路径：

1.创建一个与当前剪切路径相交的 Shape。
2.调用 clip，将剪切路径更改为当前剪切路径与新 Shape 相交的部分。

下例中，我们先用椭圆创建剪切路径，然后通过调用 clip 来修改剪切路径。

```
public void paint(Graphics g) {
  Graphics2D g2 = (Graphics2D) g;

// 画布的宽度和高度
  int w = getSize().width;
  int h = getSize().height;
  // 创建一个椭圆并用作剪切路径
  Ellipse2D e = new Ellipse2D.Float(w/4.0f,h/4.0f,
                                    w/2.0f,h/2.0f);
  g2.setClip(e);

  // 填充画布。仅绘制位于剪切路径内的区域
  g2.setColor(Color.cyan);
  g2.fillRect(0,0,w,h);

  // 更改剪切路径，将它设置成当前剪切路径
  // 与新矩形相交的部分。
  Rectangle r = new Rectangle(w/4+10,h/4+10,w/2-20,h/2-20);
  g2.clip(r);

  // 填充画布。 仅绘制位于新剪切路径内
  // 的区域
  g2.setColor(Color.magenta);
  g2.fillRect(0,0,w,h);
}
```

2.3.5 设置 **Graphics2D** 变换

要变换 Shape、文本字符串或 Image，请在绘制前将新 AffineTransform 添加到 Graphics2D 上下文的变换流水线中。绘制图形对象时将应用该变换。

例如，要绘制一个旋转 45 度的矩形：

1.通过调用 AffineTransform. getRotateInstance 获得一次旋转变换。
2.调用 Graphics2D.setTransform 将新变换添加到变换流水线中。
3.创建 Rectangle2D.Float 对象。
4.调用 Graphics2D.draw 绘制该矩形。

下例中，我们将在绘制矩形时使用 AffineTransform 实例将它旋转 45 度。

```
Rectangle2D rect = new Rectangle2D.Float(1.0,1.0,2.0,3.0);
AffineTransform rotate45 =
  AffineTransform.getRotateInstance(Math.PI/4.0,0.0,0.0)
g2.setTransform(rotate45);
g2.draw(rect);
```

下例中，我们使用 AffineTransform 沿一个中心点旋转文本字符串：

```
// 定义绘制变换
AffineTransform at = new AffineTransform();
// 应用平移变换以便为旋转文本
// 留出空间。
at.setToTranslation(400.0, 400.0);
g2.transform(at);
// 创建旋转变换以旋转文本
at.setToRotation(Math.PI / 2.0);
// 以 90 度角分别绘制四个“Java”字符串副本
for (int i = 0; i < 4; i++) {
    g2.drawString("Java", 0.0f, 0.0f);
    g2.transform(at);
}
```

用户可以用相同的方法来变换图像 -- 在绘制期间将应用 Graphics2D 上下文中的变换，而不管要绘制的图形对象是什么类型。

要对图像应用变换*而不*改变 Graphics2D 上下文中的变换，可以将 AffineTransform 传递给 drawImage：

```
AffineTransform rotate45 =
  AffineTransform.getRotateInstance(Math.PI/4.0,0.0,0.0)
g2.drawImage(myImage, rotate45);
```

还可对 Font 应用变换以创建衍生 Font。有关详细信息，参见第 65 页的“创建衍生字体”。

2.3.6 指定复合样式

AlphaComposite 封装了复合规则，可确定在对象相互重叠时如何绘制颜色。要指定 Graphics2D 上下文的复合样式，请创建 AlphaComposite 并将它传给 setComposite。最常用的复合样式是 SRC_OVER。

2.3.6.1 使用“用源色覆盖”复合规则

SRC_OVER 复合规则将源像素复合到目的像素上，因此复合像素呈现源像素的颜色。例如，如果先绘制蓝色矩形，然后绘制与其部分重叠的红色矩形，则重叠区域将为红色。也就是说，后绘制的对象将显示在上面。

要使用 SRC_OVER 复合规则：

1.通过调用 getInstance 并指定 SRC_OVER 规则，创建 AlphaComposite 对象。

```
AlphaComposite ac =
AlphaComposite.getInstance(AlphaComposite.SRC_OVER);
```

1.调用 setComposite，将 AlphaComposite 对象添加到 Graphics2D 上下文中。

```
g2.setComposite(ac);
```

一旦设置完复合对象，就将使用指定的复合规则来绘制重叠对象。

2.3.6.2 增加复合对象的透明度

AlphaComposite 允许用户指定一个额外的 alpha 常量来乘以源像素的 alpha 值，以增加透明度。

例如，要创建以 50% 透明度绘制源对象的 AlphaComposite 对象，请指定 alpha 为 .5：

```
AlphaComposite ac =
AlphaComposite.getInstance(AlphaComposite.SRC_OVER, .5f);
```

下例中，我们将创建 alpha 为 .5 的 alpha 复合对象源，然后将其添加到图形上下文中，使得以后的形状均绘制为 50% 的透明度。

```
public void paint(Graphics g) {
  Graphics2D g2 = (Graphics2D) g;

  g2.setColor(Color.red);
  g2.translate(100,50);
  // 弧度 = 度 * pie / 180
  g2.rotate((45*java.lang.Math.PI)/180);
  g2.fillRect(0,0,100,100);
  g2.setTransform(new AffineTransform());  // 设置标识
  // 创建新 alpha 复合
  AlphaComposite ac =
      AlphaComposite.getInstance(AlphaComposite.SRC_OVER,0.5f);
  g2.setComposite(ac);
  g2.setColor(Color.green);
  g2.fillRect(50,0,100,100);
  g2.setColor(Color.blue);
  g2.fillRect(125,75,100,100);
  g2.setColor(Color.yellow);
  g2.fillRect(50,125,100,100);
  g2.setColor(Color.pink);
  g2.fillRect(-25,75,100,100);
}
```


2.4 绘制图形元素

Graphics2D 提供绘制 Shape，Text 和 Image 的方法：

- draw-- 利用 Graphics2D 上下文中的 Stroke 和 Paint 对象描绘 Shape 的路径。
- fill-- 利用 Graphics2D 上下文中的 Paint 填充 Shape。
- drawString-- 利用 Graphics2D 上下文中的 Paint 绘制指定的文本串。
- drawImage-- 绘制指定的图像。

要描绘和填充形状，必须调用 draw 和 fill 两种方法。

同时,Graphics2D 也支持 JDK 早期版本中的描绘和填充方法,例如 drawOval 和 fillRect。

2.4.1 绘制形状

所有 Shape 的轮廓线都可用 Graphics2D.draw 方法进行绘制。同时，下列 JDK 早期版本中的描绘方法也受到支持：drawLine、drawRect、drawRoundRect、drawOval、drawArc、drawPolyline、drawPolygon、draw3DRect。

绘制 Shape 时，其路径由 Graphics2D 上下文中的 Stroke 对象描绘（有关详细信息，参见第 19 页的“Stroke 属性”）。通过设置 Graphics2D 上下文中的相应 BasicStroke 对象，用户即可绘制任意宽度或图案的线条。BasicStroke 对象也可定义线条的笔形和交点属性。

要绘制 Shape 的轮廓线：

1.创建 BasicStroke 对象
2.调用 Graphics2D.setStroke
3.创建 Shape。
4.调用 Graphics2D.draw(shape)。

下例中，我们使用 GeneralPath 对象定义星形，然后将 BasicStroke 对象添加到 Graphics2D 上下文中以定义星形线的 with 和 join 属性。

```
public void paint(Graphics g) {
  Graphics2D g2 = (Graphics2D) g;

  // 创建并设置绘图
  g2.setStroke(new BasicStroke(4.0f));

  //用通用路径对象创建星形
  GeneralPath p = new GeneralPath(GeneralPath.NON_ZERO);
  p.moveTo(- 100.0f, - 25.0f);
  p.lineTo(+ 100.0f, - 25.0f);
  p.lineTo(- 50.0f, + 100.0f);
  p.lineTo(+ 0.0f, - 100.0f);
  p.lineTo(+ 50.0f, + 100.0f);
  p.closePath();

  // 将原图向画布中心位置变换
  g2.translate(100.0f, 100.0f);

  // 绘制星形的路径
  g2.draw(p);
}
```

2.4.2 填充形状

Graphics2D.fill 方法可用于*填充*任一 Shape。Shape 被填充完毕，系统就会用 Graphics2D 上下文的当前 Paint 属性（包括 Color、TexturePaint 或 GradientPaint）对其路径以内的区域进行绘制。

同时，下列 JDK 早期版本中的填充方法也受到支持：fillRect、fill3DRect、fillRoundRect、fillOval、fillArc、fillPolygon、clearRect。

要填充一个 Shape：

1. 利用 Graphics2D.setColor 或 Graphics2D.setPaint 在图形上下文中设置填充颜色或图案。
2. 创建 Shape。
3. 调用 Graphics2D.fill 绘制上述 Shape。

下例中，系统调用 setColor 定义 Rectangle2D 的绿色填充。

```
public void paint(Graphics g) {
  Graphics2D g2 = (Graphics2D) g;

  g2.setPaint(Color.green);
  Rectangle2D r2 = new Rectangle2D.Float(25,25,150,150);

  g2.fill(r2);
}
```


2.4.3 绘制文本

要绘制文本串，需要调用 Graphics2D.drawString 以传入要绘制的字符串。有关绘制文本和选择字体的详细信息，参见第 45 页的“字体和文本布局”。

2.4.4 绘制图像

要绘制一个 Image，需要创建该 Image 并调用 Graphics2D.drawImage。有关图像绘制的详细信息，参见第 67 页的“图像处理”。

2.5 定义自定义复合规则

通过实现 Composite 和 CompositeContext 接口程序，用户可以创建全新的复合操作类型。Composite 对象提供的 CompositeContext 对象实际上记录着状态并执行复合操作。利用一个 Composite 对象可创建多个 CompositeContext 对象，以记录多线程环境中的各个状态。

第 3 章

几何形状

Java 2D API 提供了几种定义诸如点、直线、曲线和矩形等常用几何对象的类。这些新几何类是 java.awt.geom 包的组成部分。为保持向后兼容性，以前版本的 JDK 软件的几何类（例如 Rectangle 、 Point 和 Polygon）仍在 java.awt 包中。

Java 2D API 几何类（例如 GeneralPath、Arc2D 和 Rectangle2D）实现 java.awt 定义的 Shape 接口。Shape 提供说明、检查几何路径对象的公共协议。新接口 PathIterator 定义了从几何形状检索元素的方法。

用几何类可以很容易定义和处理几乎所有二维对象。

3.1 接口和类

下表列出了重要的几何接口和类。这些接口和类多数是 java.awt.geom 包的组成部分。其它如 Shape 是 java.awt 包的组成部分，主要为了保持与以前版本 JDK 软件的兼容性。

接口	说明
PathIterator	定义从路径检索元素的方法。
Shape (java.awt)	提供说明和检查几何路径对象的公共方法集。由 GeneralPath 和其它几何类实现。

类	说明
Arc2D Arc2D.Double Arc2D.Float	扩展：RectangularShape 表示用外切矩形、起始角、角度和闭合类型定义的圆弧。用来指定圆弧，包括浮点和双精度两种方式：Arc2D.Float 和 Arc2D.Double。
Area	实现：Shape, Cloneable 表示支持布尔操作的区域几何形状。
CubicCurve2D CubicCurve2D.Double CubicCurve2D.Float	实现：Shape 表示 (w) 坐标系中一段三次方程曲线。用来指定三次曲线，包括浮点和双精度两种方式：CubicCurve2D.Float 和 CubicCurve2D.Double。
Dimension2D	封装宽和高。所有存储两维的对象的抽象父类。
Ellipse2D Ellipse2D.Double Ellipse2D.Float	扩展：RectangularShape 表示用外切矩形定义的椭圆。用来指定椭圆，包括浮点和双精度两种方式：Ellipse2D.Float 和 Ellipse2D.Double。
FlatteningPathIterator	返回 PathIterator 对象的整平视图。 可用来为自身不能进行插值计算的 Shapes 提供整平操作。
GeneralPath	实现：Shape 表示由直线及二次、三次曲线形成的几何路径。
Line2D Line2D.Double Line2D.Float	实现：Shape 表示 (x, y) 坐标空间的线段。用来指定直线，包括浮点和双精度两种：Line2D.Float 和 Line2D.Double。
Point2D Point2D.Double Point2D.Float	代表 (x, y) 坐标空间中某个位置的点。用来指定点，包括浮点和双精度两种方式：Point2D.Float 和 Point2D.Double。
QuadCurve2D QuadCurve2D.Double	实现：Shape 表示 (x, y) 坐标空间中一段二次方程曲线。用来指定二次曲线，包括浮点和双精度两种方式：QuadCurve2D.Float 和

QuadCurve2D.Float	QuadCurve2D.Double。
Rectangle2D Rectangle2D.Double Rectangle2D.Float	扩展：RectangularShape 表示由 (x, y) 位置和 (w x h) 维定义的矩形。用来指定矩形，包括浮点和双精度两种方式：Rectangle2D.Float 和 Rectangle2D.Double。
RectangularShape	实现：Shape 为处理有矩形边界的形状提供公共处理例程。
RoundRectangle2D RoundRectangle2D.Double RoundRectangle2D.Float	扩展：RectangularShape 表示由 (x, y) 位置、(w x h) 维及角弧的宽和高定义的圆角矩形。用来指定圆角矩形，包括浮点和双精度两种方式：RoundRectangle2D.Float 和 RoundRectangle2D.Double。

3.2 几何概念

Shape（形状）是任何实现 Shape 接口的类（例如 GeneralPath 或 Rectangle2D.Float）的一个实例。 Shape 的轮廓（轮廓线）是指它的*路径*。

所谓绘制 Shape，即对 Shape 的路径应用 Stroke 对象在 Graphics2D 上下文中定义的画笔样式。 而所谓填充 Shape，即在它的路径区里应用 Graphics2D 上下文中的 Paint。有关详细信息，参见第 15 页“用 Graphics2D 绘制”。

Shape 的路径也可用来定义*剪切路径*。剪切路径决定了所要渲染的像素，即只渲染位于剪切路径定义的区域内的像素。剪切路径是 Graphics2D 上下文的组成部分。有关详细信息，参见第 32 页“设置剪切路径”。

GeneralPath 是一种可用来表示能用直线和二次或三次曲线构造的二维对象的形状。为方便起见，java.awt.geom 还提供了表示矩形、椭圆、圆弧和曲线等常用几何对象的 Shape 接口的实现。Java 2D API 还提供了支持构造几何区域的形状的特殊类型。

3.2.1 构造几何区域

构造几何区域 (CAG) 是通过对现有对象进行布尔运算以创建新几何对象的过程。在 Java 2D API 中，一种名为 Area 的特殊 Shape 类型支持布尔运算。用户可从任何 Shape 构造 Area。

Area 支持下面的布尔运算：

- Union（并）
- Intersection（交）
- Subtraction（差）
- Exclusive OR（异或 XOR）

图 3-1 说明了这些运算。

图 3-1 布尔运算

3.2.2 边界与选中测试

*边框*是最接近几何形状的矩形。边框用来决定用户是否选择了对象或“选中” (hit)。

Shape 接口定义了两种检索形状的边框的方法：getBounds 和 getBounds2D。getBounds2D 方法返回 Rectangle2D 而不是 Rectangle，可以提供对形状的边框更精确的描述。

Shape 还提供了一些方法来判断：

- 指定点是否位于该形状的边界内 (contains)
- 指定矩形是否整体位于该形状的边界内 (contains)
- 指定矩形是否与该形状相交 (intersects)

3.3 通过区域组合创建新形状

Area(区域)可用来从简单的形状(例如圆和正方形)快速构造复杂的 Shape。要通过组合 Area 来创建新的复杂 Shape：

1.利用 Shape 构造要组合的 Area。

调用相应的布尔运算符： add、subtract、intersect、exclusiveOr。

例如，CAG 可以用来创建图 3-2 中所示的梨。

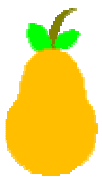


图 3-2 从圆构造的梨

梨的主体通过对下列两个重叠 Area 执行 Union（并）运算构造而成：圆和椭圆。每片叶子是通过在两个重叠的圆上执行 intersection（交）运算，然后将它们通过 union（并）运算连接为单个 Shape 而成。还可以对重叠的圆执行两次 subtraction（差）运算来构造叶柄。

3.4 创建自定义形状

通过实现 Shape 接口，用户可以创建定义新形状的类。只要可以实现 Shape 接口方法，如何在内部表示形状无关紧要。Shape 必须可以生成指定其轮廓的路径。

例如，可以创建 Shape 的一个简单实现，将多边形表示为点阵列。构建多边形后，可以将其传给 draw、setClip 或任意需要将 Shape 对象作为参数的方法。

PolygonPath 类必须实现 Shape 接口方法：

- contains
- getBounds
- getBounds2D
- getPathIterator
- intersects

可以对文本字符串使用 Java 2D API 变换和绘制机制。另外，Java 2D API 提供与文本相关的类，它们支持划分细致的字体控制和复杂的文本布局。其中包括增强的 Font 类和新的 TextLayout 类。

本章集中介绍 java.awt 和 java.awt.font 中的接口和类所支持的新字体和文本布局功能。有关使用这些功能的详细信息，参见 Java 开发人员连接 http://developer.java.sun.com/developer/onlineTraining/Graphics/2DText/ 中的“2D 文本教程”。

有关文本分析和国际化的信息，参见“Java 教程”中的 java.text 文档和“编写全球化程序”。 有关使用 Swing 中实现的文本布局机制的信息，参见“Java 教程”中的 java.awt.swing.text 文档和“使用 JFC/Swing 包”。

注意：本章所包含的多语种文本布局方面的信息是根据 Mark Davis、Doug Felt 和 John Raley 的论文 *“International Text in JDK 1.2”* 编写的(版权所有 1997, Taligent, Inc.)。

4.1 接口和类

下表列出了主要的字体及文本布局接口和类。这些接口和类多数是 java.awt.font 包的组成部分。其它（例如 Font）是 java.awt 包的组成部分，用于保持与早期版本 JDK 的兼容性。

接口	说明
MultipleMaster	代表 Type 1 Multiple Master 字体。通过 Font 对象实现。后者包含多个主字体，允许对多个主设计控制进行访问。
OpenType	代表 Open Type 和 True Type 字体。通过 Font 对象实现。后者包括 Open Type 和 True Type 字体，允许对字体的 sfnt 表进行访问。

类	说明
Font	代表主机系统上可用字样集中的字样实例。提供详细的字体信息规范，并允许访问字体及其字形的信息（在 java.awt 包中）。
FontRenderContext	封装正确度量文本所需的信息。
GlyphJustificationInfo	提供有关字形对齐属性的信息（例如粗细度、优先级、吸收率和界限）。
GlyphMetrics	为单个字形提供度量。
GlyphVector	字形及其位置的集合。
GraphicAttribute	TextLayout 属性的基本类，指定嵌入文本的图形。通过 ShapeGraphicAttribute 和 ImageGraphicAttribute 来实现。它们可实现在 TextLayout 中嵌入 Shape（形状）和 Image（图像）。可将其子类化以实现自定义字符替换图形的功能。
ImageGraphicAttribute	扩展：GraphicAttribute GraphicsAttribute 用于在 TextLayout 内绘制 Image。
LineBreakMeasurer	将跨行的文本块断开以适应指定行长的 TextLayout 对象。
LineMetrics	提供对沿一行和多行排定字符布局时所需字体度量的访问。这些度量包括提升、降低、前导、高度和基线信息。
ShapeGraphicAttribute	扩展：GraphicAttribute GraphicsAttribute 用于在 TextLayout 内绘制 Shape。
TextAttribute	定义用于文本处理的属性关键字和值。
TextHitInfo	提供 TextLayout 中字符的选中测试信息。
TextLayout	实现：Cloneable 提供有一定样式的字符数据（包括双向文本）的不可变图形表示。

4.2 字体概念

Font 类的功能已得到增强，可支持详细的字体信息规范和复杂印刷特性的使用。

Font 对象代表系统上可用字样集的字样实例。通用字样实例包括 Helvetica Bold 和 Courier Bold Italic。

与 Font 关联的名称有三个 -- 逻辑名、系列名和字样名：

- Font 对象的*逻辑名*是映射为平台上可用的某个特定字体的名称。在 JDK 1.1 及更早版本中，逻辑字体名是用来指定字体的名称。JDK 1.2 中指定字体时应使用*字样名*代替逻辑名。可通过调用 getName 从字体中得到逻辑名。对于映射为平台上可用的特定字体，要得到其逻辑名表，请调用 java.awt.Toolkit.getFontList。
- Font 对象的*系列名*是字体系列的名称。它决定了用多种字样（例如 Helvetica）时的版式设计。可通过 getFamily 方法检索系列名。
- Font 对象的*字样名*指的是安装在系统上的实际字体名。在 JDK 1.2 中指定字体时应使用该名称。通常仅称为*字体名*。可通过调用 getFontName 来检索字体名。要确定系统上可用的字样，可调用 GraphicsEnvironment.getAllFonts。

可通过 getAttributes 方法来访问有关 Font 的信息。Font 的属性包括名称、大小、变换和字体特征（例如粗细和形态）。

LineMetrics 对象封装了与 Font 相关联的度量信息，例如上升值、下降值和前置量：

- *上升值*是从基线到提升线的距离。该距离通常为大写字母的常规高度，但有一些字符可能会超出提升线。
- *下降值*是从基线到降低线的距离。大部分字符的最低点位于降低线以内，但有一些字符可能会超出降低线。
- *前置量*是从降低线到下一行顶部的缺省距离。

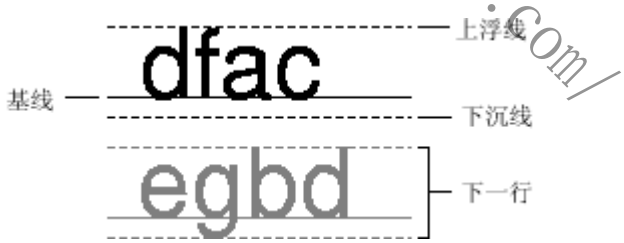


图 4-1 线度量

本信息用于沿一行准确确定字符的位置，并可按与其它线的相对关系定位行。可通过 getAscent 、getDescent 和 getLeading 方法访问这些线度量。也可通过 LineMetrics 访问有关 Font 高度、基线、下划线和删除线等特性信息。

4.3 文本布局概念

在文本能被显示前，必须利用一定的字形和连字进行相应地定形和定位。这个过程称为*文本布局*。文本布局过程包括：

- 利用适当的字形和连字使文本定形。
- 文本的正确排序。
- 文本的度量和定位。

用来排定文本布局的信息对进行文本操作（例如插入记号定位、识别和高亮显示）也是必需的。

要开发能在国际市场上销售的软件，文本必须以符合一般书写系统规则的方式用不同的语种排定布局。

4.3.1 文本定形

*字形*是一个或多个字符的形象化表示。字形的形状、大小和位置取决于它的上下文。可用多个不同的字形来表示单个的字符或字符组合，这要取决于字体和样式。

例如，在手写的草书文本中，某个特定字符可以表现为不同的形状，这要取决于它与邻近字符的连接方式。

在某些书写系统中（尤其是在阿拉伯语中），必须始终考虑字形的上下文。与英语不同，阿拉伯语中必须遵循草书格式，而不使用草书格式的文本是无法接受的。

这些草书格式在形状上会因上下文的不同而完全不同。例如，阿拉伯字母 *heh* 有四种草书格式，如图 4-2 所示。

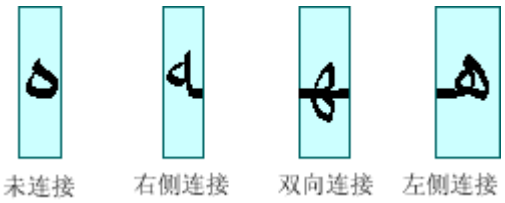


图 4-2 阿拉伯语中的草书格式

尽管这四种格式彼此极不相同，但其形状变化与英语中的草书没有根本的区别。

在某些上下文中，两个字形可能更为彻底地改变形状，合并为一种字形。这种合并而成的字形称为*连字*。例如，很多英文文本内含连字 *fi*，如图 4-3 所示。合并后的字形考虑了字符 *f* 的外伸部分，以一种看起来简单自然的方式创建了字符组合，而不是简单地使字符相连。

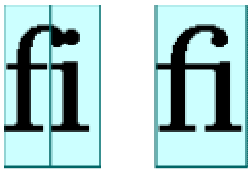


图 4-3 英文连字

阿拉伯语中也使用连字，并且某些连字的用法具有强制性 -- 给出某个字母组合而不使用相应的连字是令人无法接受的。用阿拉伯字符形成连字时，形状的变化甚至比在英文中还要彻底。例如，图 4-4 演示了两个阿拉伯字符在一起出现时组合为单个连字的方式。

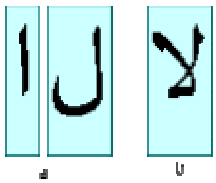


图 4-4 阿拉伯语的连字

4.3.2 文本排序

Java 编程语言中使用 Unicode 字符编码方式对文本进行编码。采用 Unicode 字符编码的文本按*逻辑顺序*存储在内存中。所谓逻辑顺序，就是字符和单词的读写顺序。逻辑顺序无需与相应字形显示的*视觉顺序*相同。

在某些书写系统中，字形的视觉顺序称为*脚本顺序*。例如，罗马体文本的脚本顺序为从左至右，而阿拉伯语和希伯来语的脚本顺序为从右至左。

除脚本顺序外，某些书写系统还存在文本行中字形和单词的排列规则。例如，尽管阿拉伯语和希伯来语的字母从右至左排列，但其数字却也是从左至右排列的（这意味着即使没有嵌入英语文本，阿拉伯语和希伯来语实际上也是双向的）。

即使存在语言混合现象，也必须保持书写系统的视觉顺序。图 4-5 对此有所说明。该图显示了一个嵌入英文句子中的阿拉伯词组。

注意：在此例及后面的示例中，我们将用大写字母来表示阿拉伯语和希伯来语文本，而用下划线来表示空格。每个示例包括两部分：一部分表示存储在内存中的字符（逻辑顺序的字符），另一部分表示这些字符的显示方式（视觉顺序的字符）。字符框下面的数字表示插入点偏移量。

图 4-5 双向文本

即使阿拉伯单词是英文语句的一部分，它们也要按从右至左的阿拉伯语脚本顺序显示。因为在逻辑上纯文本中的斜体阿拉伯单词位于阿拉伯文之后，所以在外观上它位于纯文本的左边。

当显示一行从左至右和从右至左的顺序混合的文本时，*基准方向*尤为重要。基准方向是主要书写系统的脚本顺序。例如，如果该文本主要是英文而内含少许阿拉伯语，则基准方向为从左到右。 如果该文本主要是阿拉伯语而内含少许英文，则基准方向为从右到左。

基准方向决定了方向相同的文本段的显示顺序。在图 4-5 所示的示例中，基准方向为从左至右。该示例中有三种方向：语句开始部分的英文文本为从左至右，阿拉伯文本为从右至左，而句点为从左至右。

文本流中经常需要嵌入图形。就它们对文本流和包线的影响而言，这些内联图形与字形的行为特征类似。内联图形需要采用同样的双向布局算法来定位，从而保证其在字符流中出现的位置正确。

有关在一行内实现字形准确排序的算法的详细信息，参看 Unicode Standard 中对双向算法的说明（2.0 版，第 3.11 节）。

4.3.3 文本的度量和定位

除非是使用等宽字体，否则字体中不同的字符将有不同的宽度。这意味着所有文本的定位和量度必须考虑要用哪些字符，而不仅是多少字符。例如，要准确对齐一列采用比例字体显示的数字，就不能仅用附加空格来定位文本。要准确对齐该列，需要知道每个数字的确切宽度以加以调整。

文本通常使用多种字体和样式来显示（例如粗体和斜体）。此时，即使是同一个字符也会有不同的形状和宽度，这要取决于它的字体样式。要准确对文本进行定位、度量和绘制，需要明了每一个字符*及*该字符所应用的样式。幸运的是，TextLayout 可完成这个任务。

在诸如希伯来语和阿拉伯语这样的语言中，要准确显示文本，需要在相邻字符的上下文之内度量和定位每个字符。因为字符的形状和位置可能因上下文的变化而变化，所以未考虑上下文的文本度量和定位会产生不可接受的后果。

4.3.4 支持文本操作

要实现用户对所显示文本的可编辑性，必须能：

- 显示一个插入记号，用来指示用户输入文本时新字符将插入到何处。
- 移动插入记号和插入点以响应用户输入。
- 检测用户选定的内容（选中测试）。

- 高亮显示所选文本。

4.3.4.1 显示插入记号

在可编辑的文本中，我们用*插入记号*来形象表示当前插入点，即文本中将插入新字符的位置。一般将插入记号显示为在两个字形间闪烁的竖条。新字符被插入并显示在插入记号的位置。

插入记号位置的计算可能很复杂，尤其是存在双向文本的情况下。在有方向性的分界线上的插入点偏移量有两种可能的插入记号位置，这是因为与字符偏移量相对应的两个字形在显示时彼此不相邻。见图 4-6。在该图中，插入记号显示为方括号，用来指示与插入记号相对应的字形。

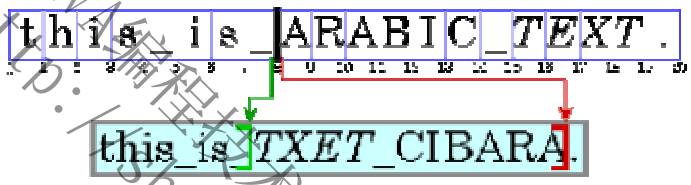


图 4-6 双插入记号

字符偏移量 8 对应 _ 之后和 A 之前的位置。如果用户输入一个阿拉伯字符，则其字形显示在 A 的右边（之前）；如果用户输入一个英文字符，则其字形显示在 _ 的右边（之后）。

为处理这种情形，一些系统显示双插入记号，即一个强（主要）插入记号和一个弱（辅助）插入记号。强插入记号指示所插入字符的方向与文本基准方向相同时该字符显示的位置。弱插入记号指示所插入字符与基住方向相反时该字符显示的位置。TextLayout 自动支持双插入记号；而 JTextComponent 不支持。

处理双向文本时，不要在字符偏移量前简单地增加字形宽度来计算插入记号的位置。否则，插入记号会出现在错误的位置，如图 4-7 所示。

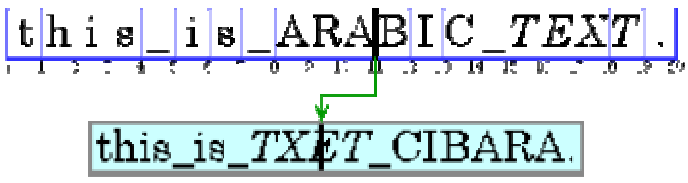


图 4-7 插入记号绘制错误

为正确给插入记号定位，需要增加偏移量左侧的字形宽度，并且考虑当前的上下文。除非已经考虑到上下文，否则字形度量并不需要与显示匹配 （上下文会影响所用的字形）。

4.3.4.2 插入记号的移动

所有的文本编辑器都允许用户通过方向键移动插入记号。用户希望插入记号向所按方向键的方向移动。在从左至右的文本中，插入点偏移量的移动很简单：右方向键逐一增加插入点偏移量，而方向头键逐一减少插入点偏移量。在双向文本或带连字的文本中，该操作会导致插入记号在有方向性的分界线上跳过字形，而在另一个方向上则向反向移动。

为平稳地移动插入记号通过双向文本，需要考虑文本的方向。不要简单地认为在按下右方向键时增加插入点偏移量，而在按下左方向键时会减少插入点偏移量。如果当前插入点偏移量处于从右至左进行的字符中，则右方向键将减少插入点偏移量，而左方向键会增加之。

移动插入记号通过有方向性的分界线时情形非常复杂。图 4-8 演示了用户操作方向键通过有方向性的分界线时的情形。在所显示的文本中向右侧移动三个位置对应着移动字符偏移量 7、19 和 18。

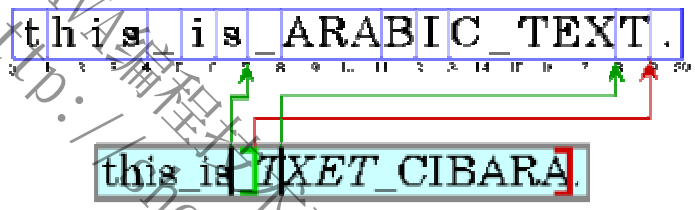


图 4-8 插入记号的移动

某些字形之间决不应出现插入记号；相反，插入记号移动时应把这类字形当作单个字符。例如，如果 *o* 和元音变音由两个相互独立的字符表示，则它们之间决不应存在插入记号（有关详细信息，参看 Unicode Standard 2.0 版，第 5 章）。

TextLayout 提供了可以在双向文本中简单而平稳地移动插入记号的方法（getNextRightHit 和 getNextLeftHit）。

4.3.4.3 选中测试

通常，设备空间的位置应转换为文本偏移量。例如，当用户单击可选文本时，鼠标的位置将转换为文本偏移量，并成为选择范围的终点。逻辑上，这是定位插入记号的逆过程。

当使用双向文本时，显示内容中的单个视觉位置可对应源文本中两个不同的偏移量，如图 4-9 所示。

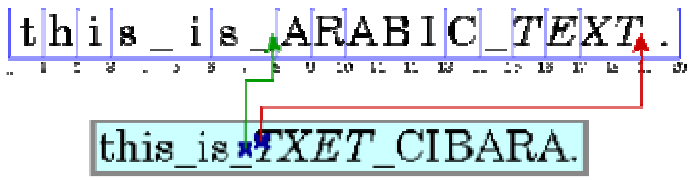


图 4-9 双向文本的选中测试

因为单个视觉位置可对应两个不同的偏移量，所以双向文本的选中测试已不仅是度量字形宽度，直到发现正确位置的字形，然后将该位置映射为字符偏移量的问题，而是一定要检测到边，这有助于区分两种可能的偏移量。

可使用 TextLayout.hitTestChar 进行检测。选中测试信息被封装到 TextHitInfo 对象中，并且其中包含测试到的边的信息。

4.3.4.4 高亮显示选择内容

高亮显示区可以形象地表示字符选择范围。高亮显示区中字形显示为反色视频或不同的背景色。

与插入记号类似，高亮显示区对于双向文本要比单向文本复杂得多。在双向文本中，相邻范围内的字符在显示时高亮显示区可能并不连续。反之，外观上显示为连续字形范围的高亮显示区可能并不对应单个、连续的字符范围。

这导致双向文本中存在两种高亮显示选择内容的方式：

- *逻辑高亮显示* -- 对于逻辑高亮显示，选定的字符在文本模型中总是连续的，而高亮显示区域允许不连续。有关逻辑高亮显示的示例，参看图 4-10。
- *视觉高亮显示* -- 对于视觉高亮显示，可能存在多个选定字符范围，但高亮显示区总是连续的。有关视觉高亮显示的示例，参看图 4-11。

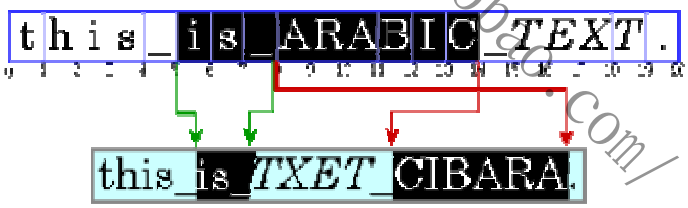


图 4-10 逻辑高亮显示（连续字符）

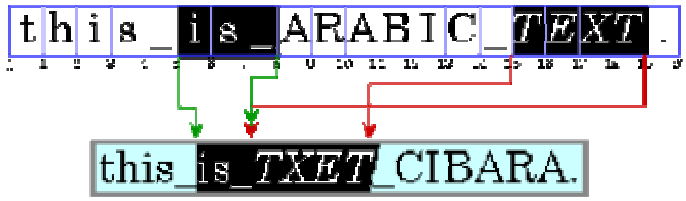


图 4-11 视觉高亮显示（连续的高亮显示区）

因为所选的字符在文本中总是连续的，所以逻辑高亮显示实现起来更为简单。

4.3.5 在 **Java** 应用程序中确定文本布局

取决于所用的 Java API，可以根据需要不同程度地控制文本布局：

- 如果只需显示一个文本块或可编辑的文本控制，可使用 JTextComponent，它会自动排定文本布局。JTextComponent 可满足很多多语种应用程序的需要，并且支持双向文本。有关 JTextComponent 的详细信息，参看《Java 教程》中的“使用 JFC/Swing 包”。
- 如果要显示简单的文本字符串，可调用 Graphics2D.drawString ，让 Java 2D 确定字符串的布局。也可使用 drawString 来绘制有一定样式的字符串和内含双向文本的字符串。有关用 Graphics2D 绘制文本的详细信息，参看第 36 页的“绘制图形元素”。
- 如果要实现自己的文本编辑例程，可使用 TextLayout 来控制文本布局、高亮显示和选中检测。TextLayout 提供的工具可处理很多常见问题，包括有多字体、多语种和双向文本的字符串。有关使用 TextLayout 的详细信息，参看第 58 页的“管理文本布局”。
- 如果需要完全控制文本的形状和位置，可使用 Font 构造自己的 GlyphVectors，然后用 Graphics2D 进行绘制。有关实施自己的文本布局机制的详细信息，参看第 64 页的“实现自定义文本布局机制”。

通常，用户无需自己执行文本布局操作。对于多数应用程序，JTextComponent 是显示静态可编辑文本的最佳解决方案。但是，JTextComponent 不支持双插入记号和双向文本中不连续选择内容的显示。如果应用程序需要上述功能，或用户倾向于实施自己的文本编辑例程，即可使用 Java 2D 文本布局 API。

4.4 管理文本布局

TextLayout 类支持含有不同书写系统下多种样式和字符的文本，包括 Arabic 和 Hebrew（由于必须对文本重新进行定形和排序才能得到满意的 Arabic 和 Hebrew 表示法，因此它们特别难于显示）。

即使是在处理纯英文，TextLayout 仍可简化文本的显示和度量过程。利用 TextLayout，用户毫不费力即可获得高质量的字样。

文本布局的性能
将 TextLayout 用于显示简单的单向文本并不能充分展示其主要性能。当 TextLayout 用于显示 Arabic 或 Hebrew 文本时，会多一些处理时间开销。但通常显示每个字符只需几个微秒，而大多数时间被用于执行一般的绘制代码。

TextLayout 类负责管理字形的定位和排序。其用途如下：

- 确定单向和双向文本的布局
- 显示和移动插入记号
- 对文本执行选中测试
- 高亮显示所选文本

某些情况下，用户可能想自己计算文本的布局，以便准确控制所用的字形及其位置。利用诸如字形大小、字距调整表及连字等信息，即可自己编写文本布局的算法，而不使用系统的布局计算机制。有关详细信息，参见第 64 页的“实现自定义文本布局机制”。

4.4.1 确定文本布局

TextLayout 可自动确定文本的布局（包括双向 (BIDI) 文本），使之外形和次序正确。要准确确定代表文本行的字样的形状和次序，TextLayout 必须知道完整的文本上下文：

- 如果文本需要位于一行上（例如按钮上单个字组成的标签或对话框中的文本行），就可以直接利用文本编写 TextLayout。
- 如果文本太大而无法位于一行内，或者想将一行上的文本分为若干由制表符分隔的片段，就不能直接编写 TextLayout， 而是必须用 LineBreakMeasurer 提供足够的上下文。

文本的基准方向通常由文本的属性（样式）设置。如果丢失了该属性，TextLayout 就会遵循 Unicode 双向算法，沿用段落起始字符的基准方向。

4.4.2 显示双插入记号

TextLayout 可以保留插入记号的信息，诸如插入记号的 Shape、位置及角度。使用该信息可以很方便地在单向及双向文本中显示插入记号。在为双向文本绘制插入记号时，使用 TextLayout 可确保插入记号准确定位。

TextLayout 提供缺省插入记号 Shape 并自动支持双插入记号。对于斜体和倾斜字形，TextLayout 将生成有一定角度的插入记号，如图 4-12 所示。这些插入记号的位置也用作字形间的边界，用于高亮显示及选中测试，使不同的用户获得一致的感受。

图 4-12 有一定角度的插入记号

在给定插入点偏移量的前提下，getCaretShapes 方法将返回一个双元素 Shape 数组：元素 0 是强插入记号，而元素 1 是弱插入记号（如果有）。要显示双插入记号，只需绘制两个插入记号 Shape，插入记号将自动准确定位。

如果想使用自定义插入记号 Shape，可从 TextLayout 中检索插入记号的位置和角度，然后自己绘制。

下例中，缺省的强、弱插入记号 Shape 使用了不同的颜色加以绘制。这是一种区分双插入记号的常见方式。

```
Shape[] caretShapes = layout.getCaretShapes(hit);
g2.setColor(PRIMARY_CARET_COLOR);
g2.draw(caretShapes[0]);
if (caretShapes[1] != null){
  g2.setColor(SECONDARY_CARET_COLOR);
  g2.draw(caretShapes[1]);
}
```

4.4.3 移动插入记号

利用 TextLayout 也可确定按左右方向键时最终的插入点偏移量。在给定一个代表当前插入点偏移量的 TextHitInfo 对象的前提下，如果按的是在方向键，getNextRightHit 方法就会返回一个 TextHitInfo 对象，给出正确的插入点偏移量。getNextLeftHit 方法提供按左方向键时的类似信息。

在下例中，当前插入点偏移量是对应于右方向键的。

```
TextHitInfo newInsertionOffset =
            layout.getNextRightHit(insertionOffset);
if (newInsertionOffset != null) {
  Shape[] caretShapes =
          layout.getCaretShapes(newInsertionOffset);
  // 绘制插入记号
  ...
  insertionOffset = newInsertionOffset;
}
```

4.4.4 选中测试

TextLayout 提供文本选中测试的简单机制。hitTestChar 方法将鼠标的 *x* 和 *y* 坐标作为参数，并返回 TextHitInfo 对象。TextHitInfo 中包含指定的测试点位置和边缘的插入点偏移量。插入点偏移量是距离测试点最近的偏移量：如果测试点超出行尾，则返回行尾的偏移量。

下例中，TextLayout 上首先调用 hitTestChar，随后用 getInsertionOffset 检索偏移量。

```
TextHitInfo hit = layout.hitTestChar(x, y);
int insertionOffset = hit.getInsertionOffset();
```

4.4.5 高亮显示所选内容

从 TextLayout 中也可获得代表高亮显示区域的 Shape。TextLayout 会在计算高亮显示区域的大小时自动考虑上下文。TextLayout 既支持逻辑意义上的高亮显示，也支持视觉上的高亮显示。

下例中，高亮显示区域被高亮显示色所充满，随即在填满的区域上绘制 TextLayout。这是一种显示高亮显示文本的常见方式。

```
Shape highlightRegion = layout.getLogicalHighlightShape(hit1,
      hit2);
graphics.setColor(HIGHLIGHT_COLOR);
graphics.fill(highlightRegion);
graphics.drawString(layout, 0, 0);
```


4.4.6 查询布局度量

TextLayout 可提供对其所表示的整个文本范围内图形度量的访问。可从 TextLayout 中获得的度量包括：上升值、下降值、前置量、加宽、可见加宽及闭合矩形。

多种字体可同时与 TextLayout 关联：不同的样式将使用不同的字体。TextLayout 的上升值和下降值是 TextLayout 中所用字体的最大值。TextLayout 的前置量计算较为复杂，而非简单的最大前置量。

TextLayout 的加宽值就是其长度：从最左侧字形的左边缘到最右侧字形的右边缘。加宽有时也称为*总加宽*。*可见加宽*是指 TextLayout 不带尾部空白的长度。

TextLayout 的边框将布局中的全部文本都包含在其中。其中既包含全部可见字形，也包含插入记号边界（其中一些可能会延伸到初值或初值与加宽值之和的范围以外）。边框是相对于 TextLayout 初值（而非某一特定屏幕位置）而言的。

下例中，将在布局的边框范围内绘制 TextLayout 中的文本。

```
graphics.drawString(layout, 0, 0);
Rectangle2D bounds = layout.getBounds();
graphics.drawRect(bounds.getX()-1, bounds.getY()-1,
         bounds.getWidth()+2, bounds.getHeight()+2);
```

4.4.7 跨行绘制文本

TextLayout 也可用于跨行显示一串文本。例如，可以将某一文本段按一定宽度换行，从而在数行上进行显示。

为此，用户无需直接创建代表每一文本行的 TextLayouts --LineBreakMeasurer 可为用户代劳。除非段落中的文本均可用，否则将无法保证双向排序始终能正确执行。LineBreakMeasurer 封装的上下文信息足够生成正确的 TextLayouts。

跨行显示文本时，行的长度通常由显示区的宽度决定。在给定行必须适应的图形宽度后，换行操作决定了行的起止位置。

最常用的策略就是在每行上放置所能适应的尽量多的字。这可由 LineBreakMeasurer 实现。其它更复杂的换行方式包括使用连字符，或者试着将段内不同行间的长度差距降到最小。Java 2D API 不提供这些方式。

要将文本段换到多行上，需要对整个段落编写 LineBreakMeasurer，然后调用 nextLayout 以在文本中移动并生成各行的 TextLayouts。

为此，LineBreakMeasurer 保留有文本内的偏移量。初始情况下，该偏移量位于文本的起始位置。每次调用 nextLayout 都会按所建 TextLayout 的字符数移动一定的偏移量。当该偏移量到达文本末尾时，nextLayout 返回 null。

LineBreakMeasurer 所创建的各个 TextLayout 的可见加宽不会超过指定的行宽。调用 nextLayout 时，通过更改所指定的行宽可对文本进行换行以适应复杂的区域，例如 HTML 网页中有时需要图像位于某固定区域内或含制表符域。也可通过传递 BreakIterator 来通知 LineBreakMeasurer 有效断点的位置。如果用户不提供，就将使用缺省 locale 机制的某个 BreakIterator。

下例中将按行绘制多语种文本片段。取决于文本基准方向是从左到右还是从右到左，行将对齐左边缘或右边缘。

```
Point2D pen = initialPosition;
LineBreakMeasurer measurer = new LineBreakMeasurer(styledText,
myBreakIterator);
while (true) {
  TextLayout layout = measurer.nextLayout(wrappingWidth);
  if (layout == null) break;
    pen.y += layout.getAscent();
    float dx = 0;
    if (layout.isLeftToRight())
      dx = wrappingWidth - layout.getAdvance();
    layout.draw(graphics, pen.x + dx, pen.y);
    pen.y += layout.getDescent() + layout.getLeading();
}
```

4.5 实现自定义文本布局机制

GlyphVector 类提供了一种显示自定义布局机制效果的途径。GlyphVector 对象可看作是这样一种算法的输出：首先取一个字符串，然后计算如何对其进行精确显示。系统中有一种内置算法，而 Java 2D API 也允许高级用户定义自己的算法。

GlyphVector 对象其实是一个字形和字形位置数组。对布局特性（例如字距调整和连字）进行完全控制的方法是使用字形而不是字符。例如，当显示字符串 "final" 时，用户可能想将前导子串 fi 替换为连字 *fi*。此时，GlyphVector 对象中的字形数就少于原始字符串中的字符数。

图 4-13 和 图 4-14 展示了布局机制如何使用 GlyphVector 对象。图 4-13 显示缺省布局机制。当 String 上调用 drawString 时，内置布局算法将：

- 利用 Graphics2D 上下文中的当前 Font 确定所用字形。
- 计算各个字形所应放置的位置。
- 将最终字形及其位置信息存储于 GlyphVector。
- 将 GlyphVector 传递给实际绘制字形的字形绘制例程。

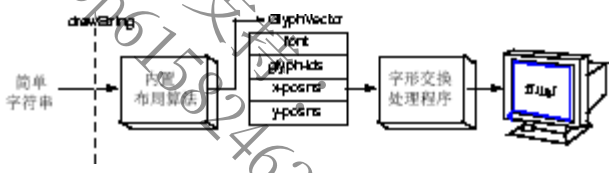


图 4-13 使用内置布局算法

图 4-14 显示使用自定义布局算法的过程。要使用自定义布局算法，用户必须汇集确定文本布局所需的全部信息。基本过程类似：

- 利用 Font 确定所用字形
- 确定字形的放置位置
- 将布局信息存储于 GlyphVector

要绘制文本，需要将 GlyphVector 传递给 drawString，后者将之传递给字形变换绘制程序。在图 4-14 中，自定义布局算法将 fi 子串替换为连字 *fi*。

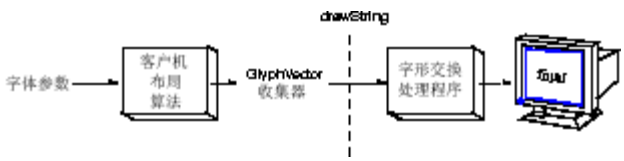


图 4-14 使用自定义布局算法

4.6 创建衍生字体

用 Font.deriveFont 方法可创建与现有 Font 对象属性不同的新 Font 对象。通常对现有 Font 进行变换后，可以创建一种新衍生 Font。为此，您必须：

1.创建 Font 对象。
2.创建要用于该 Font 的 AffineTransform。
3.调用 Font.deriveFont，传入 AffineTransform。

这样就可以很容易地利用已有 Font 创建自定义大小的 Font 或扭曲 Font。

在下面的代码引用中，我们将用 AffineTransform 创建 Helvetica 字体的扭曲形式，然后用该新字体绘制字符串。

```
// 创建字体的变换。
AffineTransform fontAT = new AffineTransform();
fontAT.setToShear(-1.2, 0.0);
// 创建 Font 对象。
Font theFont = new Font("Helvetica", Font.PLAIN, 1);
// 用修剪变换衍生新字体
theDerivedFont = theFont.deriveFont(fontAT);
// 将衍生字体添加到 Graphics2D 上下文中
g2.setFont(theDerivedFont);
// 用该衍生字体绘制字符串
g2.drawString("Java", 0.0f, 0.0f);
```


第 **5** 章

图像处理

Java 2D API 支持三种图像处理模型：

- JDK 软件的早期版本所提供的生产者/消费者（推）模型。
- JDK 1.2 软件版本中引入的即时模式模型。
- 与即时模式模型兼容的流水线（拉）模型，它将在即将面世的 Java 高级图像处理 API 中获得完全的实现。

下表比较了每种图像处理模型的特性。

本章主要介绍即时模式图像处理模型的对象和技术。Java 2D API 的即时模式图像处理类和接口提供处理像素映射图像的技术，其中的像素映射图像的数据存储在内存中。该 API 支持用多种存储格式访问数据，也支持通过几种过滤操作来操作图像。

5.1 接口和类

Java 2D API 中的即时模式图像处理 API 可以分为六组：接口、图像数据类、图像操作类、样本模型类、颜色模型类和异常。

5.1.1 图像处理接口

接口	说明
BufferedImageOp	描述 BufferedImage 对象上执行的单输入/单输出操作。由 AffineTransformOp、ColorConvertOp、ConvolveOp、LookupOp 和 RescaleOp 实现。
RasterOp	定义 Raster 对象上执行的单输入/单输出操作。由 AffineTransformOp、BandCombineOp、ColorConvertOp、ConvolveOp、LookupOp 和 RescaleOp 实现。
RenderedImage	定义对象的公共协议，这些对象包含或可以生成 Raster 格式的图像数据。

WritableRenderedImage	扩展：RenderedImage 定义对象的公共协议，这些对象包含或可以生成可修改的 Raster 格式的图像数据。
TileObserver	定义当 WritableRenderedImage 的修改状态改变时要通知的对象的协议。

5.1.2 图像数据类

类	说明
BufferedImage	继承：Image 实现：WriteableRenderedImage 带有可访问数据缓冲的图像。BufferedImage 有一个 ColorModel 和装有图像数据的 Raster。
ByteLookupTable	继承：LookupTable 包含字节数据的 LookupTable。
DataBuffer	保存包含像素数据的一个或多个数据数组。每个数据数组称为一个*组*(bank)。
DataBufferByte	继承：DataBuffer（终态） 按字节存储数据的数据缓冲区（在 Java 高级图像处理 API 中使用）。
DataBufferInt	继承：DataBuffer（终态） 存储整型数据的数据缓冲区（在 Java 高级图像处理 API 中使用）。
DataBufferShort	继承：DataBuffer（终态） 存储短数据的数据缓冲区（在 Java 高级图像处理 API 中使用）。
DataBufferUShort	继承：DataBuffer（终态）

	存储无符号短数据的数据缓冲区。
Kernel	描述在 ConvolveOp 过滤操作中输入像素及其周围像素如何影响输出像素值的矩阵。
LookupTable	继承：Object 将单带像素数据值映射为颜色值的表。
Raster	像素的矩形数组，可以从中检索图像数据。Raster 包含 DataBuffer 和 SampleModel。
ShortLookupTable	继承：LookupTable 包含短数据的查询表。
WritableRaster	继承：Raster 可以修改的 Raster。

5.1.3 图像操作类

类	说明
AffineTransformOp	实现：BufferedImageOp、RasterOp 定义仿射变换的类，用于执行从源 Image 或 Raster 中的二维坐标到目标图像或 Raster 中的二维坐标的线性映射。该类可以执行双线性或最近邻域仿射变换操作。
BandCombineOp	实现：RasterOp 该操作使用指定矩阵执行 Raster 中的带的任意线性组合。
BufferedImageFilter	继承：ImageFilter 提供使用 BufferedImageOp（一种单源/单目标图像操作符）过滤 BufferedImage 或 Raster 的一种简单方式的 ImageFilter。

ColorConvertOp	实现：BufferedImageOp、RasterOp 执行源图像中数据的逐像素颜色转换。
ConvolveOp	实现：BufferedImageOp、RasterOp 使用 Kernel（内核）在源图像上执行变换。变换是一种空间操作，将输入像素周围的像素乘以内核值以生成输出像素值。Kernel 从数学上定义输入像素的直接相邻像素和输出像素之间的关系。
LookupOp	实现：BufferedImageOp、RasterOp 执行从源到目标的查询操作。对于 Rasters，查询将在样本值上执行。对于 BufferedImages，查询将在颜色和 alpha 组件上执行。
RescaleOp	实现：BufferedImageOp、RasterOp 通过将每个像素值乘以缩放因子并加上偏移量，对源图像中的数据执行逐像素缩放。

5.1.4 样本模型类

类	说明
BandedSampleModel	继承：ComponentSampleModel（终态） 提供对某些图像数据的访问，这些图像数据的存储方式就象将样本存储为 DataBuffer 各个组中的带一样。一个像素由每个带的一个样本组成。
ComponentSampleModel	继承：SampleModel 提供对某些图像数据的访问，这些图像数据的存储方式为：像素的每个样本都保存在 DataBuffer 的不同元素中。 支持不同类型像素的交叉存取。
MultiPixelPackedSampleModel	继承：SampleModel 提供对某些图像数据的访问，这些图像数据的存储方式为：

	多个单样本像素打包成 DataBuffer 的一个元素。
PixelInterleavedSampleModel	继承：ComponentSampleModel 提供对某些图像数据的访问，这些图像数据的存储方式为：在数据数组的相邻元素中存放每个像素的样本数据，而所有元素最后存放在一个 DataBuffer 组中。
SampleModel	定义某一机制的抽象类，用该机制可从图像中抽取样本数据而无需了解基本数据在 DataBuffer 中如何存放。
SinglePixelPackedSampleModel	继承：SampleModel 提供对某些图像数据的访问，这些图像数据的存储方式为：将属于某一像素的所有样本都打包成 DataBuffer 的一个元素。

5.1.5 颜色模型类

类	说明
ColorModel	实现：Transparency JDK1.1 类。 定义从图像像素值向颜色组件（例如红色、绿色和蓝色）进行转换的方法的抽象类。
ComponentColorModel	继承：ColorModel 一种可以处理任意 ColorSpace 和颜色组件数组以匹配 ColorSpace 的 ColorModel。该类可以用来表示多数 GraphicsDevices 上的颜色模型。
DirectColorModel	继承：PackedColorModel JDK1.1 类，是一种 ColorModel，表示具有 RGB 颜色组件的像素值，这些 RGB 颜色组件直接嵌入像素本身的位中。该颜色模型类似于 X11 TrueColor 视觉效果。ColorModel.getRGBdefault 返回的缺省 RGB ColorModel 是 DirectColorModel。

IndexColorModel	继承：ColorModel JDK1.1 类，是一种 ColorModel，代表 sRGB ColorSpace 中固定颜色映射索引的像素值。
PackedColorModel	继承：ColorModel 一种抽象 ColorModel，代表具有颜色组件的的像素值，这些颜色组件直接嵌入像素的位中。DirectColorModel 将 PackedColorModel 扩展为支持包含 RGB 颜色组件的像素。

5.1.6 异常类

类	说明
ImagingOpException	继承：RuntimeException 当 BufferedImageOp 或 RasterOp 过滤器方法不能处理图像时抛出。
RasterFormatException	继承：RuntimeException 当 Raster 中有无效布局信息时抛出。

5.2 即时模式图像处理的概念

即时模式图像处理模型支持在内存中存储固定分辨率的图像。该模型也支持对图像数据的过滤操作。它使用了许多类和接口。

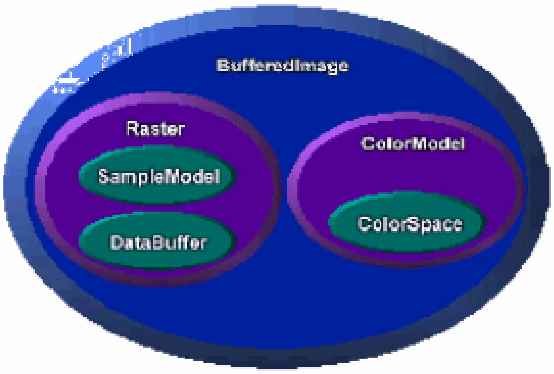


图 5-1 BufferedImage 和支持的类

如图 5-1 中所示，BufferedImage 提供一般图像管理。BufferedImage 可直接在内存中创建，用来保存和操作从文件或 URL 中检索的图像数据。BufferedImage 可以用屏幕设备的 Graphics2D 对象显示，或用相应的 Graphics2D 上下文绘制到其它目标上。BufferedImage 对象包括另外两个对象：Raster 和 ColorModel。

Raster 类提供图像数据管理。它可以表示图像的矩形坐标，在内存中保存图像数据，也可以提供从一个图像数据缓冲区创建多个子图像的机制。此外，它还提供访问图像中特定像素的方法。 Raster 对象包含另外两个对象：DataBuffer 和 SampleModel。

DataBuffer 类保存内存中的像素数据。

SampleModel 类解释缓冲区中的数据，并将其作为单个像素或矩形区域像素提供。

ColorModel 类提供对图像样本模型中的像素数据的颜色解释。

图像包提供定义 BufferedImage 和 Raster 对象上的过滤操作的其它类。每种图像处理操作都在实现 BufferedImageOp 接口或 RasterOp 接口（或同时实现这两种接口）的类中进行了具体描述。操作类定义 filter 方法，用于执行实际图像操作。

图 5-2 说明了用于 Java 2D API 图像处理的基本模型：

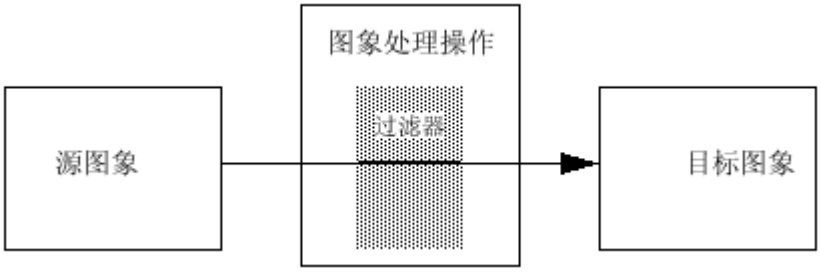


图 5-2 图像处理模型

支持的操作包括：

- 仿射变换
- 放大
- 查询表修改
- 带的线性组合
- 颜色转换
- 变换

注意：如果仅对显示和操作图像感兴趣，则只需了解 BufferedImage 类和过滤操作类。另一方面，如果要编写过滤器程序或直接访问图像数据，则需要了解与 BufferedImage 有关的类。

5.2.1 术语

下面是讨论中使用的一些术语：

数据元素：存储图像数据的基本单位。数据元素是 DataBuffer 数组的单个成员。数据缓冲区中元素的布局与图像的 SampleModel 将数据作为像素解释没有关系。

样本：图像像素的不同成员。SampleModel 提供将 DataBuffer 中的元素转换为像素及其样本的机制。像素的样本可以表示特定颜色模型中的基本值。例如，RGB 颜色模型中的像素由三个样本组成：红色、绿色和蓝色。

组件：与颜色解释无关的像素值。组件和样本的差别可用于 IndexColorModel，其中像素组件是 LookupTable 的索引。

带：图像中某种类型的所有样本集合，例如所有红色样本或所有绿色样本。可以用多种方法存储像素数据。Java 2D API 所提供的两种是带形和像素交叉存取。带形存储按带组织图像数据：一个像素由每个带中相同位置的样本数据组成。像素交叉存取存储用一个包含所有像素的数组和由每个像素中相同索引位置的样本集合组成的带按像素组织图像数据。

主值：某一特定颜色模型中颜色值的不同成员；例如 RGB 模型由主值红色、绿色和蓝色构成颜色值。

5.3 使用 BufferedImage

BufferedImage 类是支持即时图像处理模式的主要类。它管理内存中的图像，提供存储像素数据、解释像素数据以及将像素数据绘制到 Graphics 或 Graphics2D 上下文中的方法。

5.3.1 创建 BufferedImage

要创建 BufferedImage，需调用 Component.createImage 方法；它将返回 BufferedImage 实例，该实例的绘制特性与用来创建它的组件的绘制特性相匹配--创建的图像不透明，有 Component 的前景和背景色，而且不能调整图像的透明度。当要在组件中用双缓冲绘制动画时，可以使用该技术；有关详细信息，参见第 78 页的“在屏外缓冲内绘制”。

```
public Graphics2D createDemoGraphics2D(Graphics g) {
        Graphics2D g2 = null;
        int width = getSize().width;
        int height = getSize().height;

        if (offImg == null || offImg.getWidth() != width ||
                                    offImg.getHeight() != height) {
                offImg = (BufferedImage) createImage(width, height);
        }

        if (offImg != null) {
                g2 = offImg.createGraphics();
                g2.setBackground(getBackground());
        }

        // ..清除画布..
        g2.clearRect(0, 0, width, height);

        return g2;
}
```

也可以使用所提供的某种构造函数方法在内存中创建空白 BufferedImage。

5.3.2 在屏外缓冲内绘制

BufferedImage 类可以用来准备屏外图形元素，然后将其复制到屏幕上。当图形很复杂或需要重复使用时，这项技术非常有用。例如，如果要使一个复杂形状显示数次，可以一次将其绘制到屏外缓冲中，然后再复制到窗口中的不同位置。由于形状仅绘制一次，然后进行复制，因此可以更快地显示图形。

java.awt 包允许按与绘制到窗口中相同的方式绘制到 Image 对象中，从而使屏外缓冲功能更易于使用。绘制到屏外图像时，可以使用所有 Java 2D API 绘制功能。

屏外缓冲经常用于动画。例如，可以使用屏外缓冲绘制对象一次，然后在窗口内移动。同样，当用户用鼠标移动图形时，可以使用屏外缓冲提供反馈。可以将图形一次绘制到屏外缓冲，然后在拖动鼠标时将其复制到鼠标位置，而无须在每个鼠标位置重新绘制图形。[1]

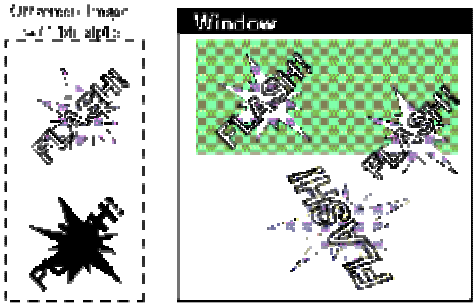


图 5-3 使用屏外缓冲

图 5-3 演示了程序如何绘制屏外图像，然后将该图像多次复制到窗口中的过程。图像最后一次复制时将进行变换。注意：变换图像时若不重新绘制，可能会导致不满意的效果。

5.3.2.1 创建屏外缓冲

创建屏外缓冲图像最简单的方法是使用 Component.createImage 方法。

通过创建颜色空间、深度和像素布局恰好与要用的窗口相匹配的图像，可以高效地将图像送入图形设备中。这就使得 drawImage 的效率得到提高。

也可以直接构造 BufferedImage 对象作为屏外缓冲。它便于控制屏外图像的类型或透明度。

BufferedImage 支持几种预定义的图像类型：

- TYPE_3BYTE_BGR
- TYPE_4BYTE_ABGR
- TYPE_4BYTE_ABGR_PRE
- TYPE_BYTE_BINARY
- TYPE_BYTE_GRAY
- TYPE_BYTE_INDEXED
- TYPE_CUSTOM
- TYPE_INT_ARGB_PRE
- TYPE_INT_ARGB
- TYPE_INT_BGR
- TYPE_INT_RGB
- TYPE_USHORT_555_RGB
- TYPE_USHORT_565_RGB
- TYPE_INT_GRAY

BufferedImage 对象可以包含 alpha 通道。图 5-3 中，alpha 通道用来区分已绘和未绘区域，允许已绘图形（在本例中为阴影矩形）上出现不规则形状。其它情况下，可以使用 alpha 通道将新图像中的颜色与已有图像中的颜色相混合。

注意：除非需要 alpha 图像数据以处理透明度（如图 5-2 所示的不规则形状图像），否则应避免创建带有 alpha 的屏外缓冲。在不需要的地方使用 alpha 将降低绘制性能。

GraphicsConfiguration 提供了一些便利的方法，可以自动创建格式与您的配置兼容的缓冲图像。也可以查询与窗口所驻留的图形设备相关联的图形配置，以获得构造兼容 BufferedImage 对象所需的信息。

5.3.2.2 在屏外缓冲内绘制

要在缓冲图像内绘制，需调用它的 BufferedImage.createGraphics 方法，该方法返回 Graphics2D 对象。可以用此对象调用所有 Graphics2D 方法，从而在图像中画基本图形、放置文本以及绘制其它图像。这种绘制技术支持 2D 图像处理包所提供的抖动和其它增强功能。以下代码说明了屏外缓冲的用法：

```
public void update(Graphics g){
        Graphics2D g2 = (Graphics2D)g;
        if(firstTime){
                Dimension dim = getSize();
                int w = dim.width;
                int h = dim.height;
                area = new Rectangle(dim);
                bi = (BufferedImage)createImage(w, h);
                big = bi.createGraphics();
                rect.setLocation(w/2-50, h/2-25);
                big.setStroke(new BasicStroke(8.0f));
                firstTime = false;
        }

        // 清除以前绘制的矩形。
        big.setColor(Color.white);
        big.clearRect(0, 0, area.width, area.height);

        // 在缓冲中绘制和填充新定位的矩形。
        big.setPaint(strokePolka);
        big.draw(rect);
        big.setPaint(fillPolka);
        big.fill(rect);

        // 将缓冲图形绘制到屏幕。
```

```
        _g2.drawImage(bi, 0, 0, this);
}
```

5.3.3 直接操作 **BufferedImage** 数据

除了可在 BufferedImage 中直接绘制外，还可以用几种方法直接访问和操作图像的像素数据。 它便于实现 BufferedImageOp 过滤接口，如第 83 页的“图像处理与增强功能”中所述。

可以使用 BufferedImage.setRGB 方法直接将像素或像素数组的值设置为特定的 RGB 值。注意：直接修改像素时不会产生抖动。也可以通过操作与 BufferedImage 关联的 WritableRaster 对象来操作像素数据（参见第 79 页的“管理和操作光栅”）。

5.3.4 过滤 **BufferedImage**

可以使用实现 BufferedImageOp 接口的对象将过滤操作应用到 BufferedImage 中。第 83 页的“图像处理与增强功能” 中讨论了过滤和提供此过滤接口的类。

5.3.5 绘制 **BufferedImage**

要将缓冲图像绘制到特定上下文中，需调用上下文的 Graphics 对象的 drawImage 方法。例如，在 Component.paint 方法内绘制时，需要在传递给该方法的 graphics 对象上调用 drawImage。

```
public void paint(Graphics g) {

        if (getSize().width <= 0 || getSize().height <= 0)
                return;
        Graphics2D g2 = (Graphics2D) g;

        if (offImg != null && isShowing())  {
                g2.drawImage(offImg, 0, 0, this);
        }
}
```

5.4 管理和操作光栅

BufferedImage 对象使用 Raster 来管理其像素数据的矩形数组。Raster 类定义图像坐标系统的域--宽度、高度和原点。Raster 对象本身使用两个对象管理像素数据：DataBuffer 和 SampleModel。DataBuffer 是存储光栅的像素数据的对象(如第 81 页所述)，而 SampleModel 用于解释 DataBuffer 中的像素数据（如第 81 页所述）。

5.4.1 创建光栅

大多数情况下无需直接创建 Raster，因为在内存中创建的 BufferedImage 已经提供了一个。然而，一种 BufferedImage 构造函数方法允许通过传入 WritableRaster 来创建 Raster。

Raster 类提供了许多静态工厂方法，以便用指定的 DataBuffer 和 SampleModel 创建 Raster。当实现 RasterOp 过滤类时，可以使用这些工厂。

5.4.2 父光栅和子光栅

Raster 类中包含了父光栅和子光栅的概念。这样即可从同一父光栅构造任意数目的缓冲图像，从而提高存储效率。父光栅及其子光栅指向同一数据缓冲区，而且每个子光栅都有特定的偏移量和边界来标识它在缓冲区中的图像位置。子光栅通过其 getParent 方法来标识它的从属关系。

要创建子光栅，可以使用 Raster.createSubRaster 方法。创建子光栅时，需要标识它所覆盖的父光栅区域及其相对于父光栅原点的偏移量。

5.4.3 光栅操作

Raster 类定义了多种访问像素和像素数据的方法。它们可以用于实现需要执行低级像素操作的方法，或者用于实现提供对图像数据的光栅低级过滤和操作的 RasterOp 接口。

Raster.getPixel 方法将单个像素作为数组中的单个样本返回，从而使用户可以获得该像素。Raster.getDataElements 方法返回 DataBuffer 中指定排列方式的未解释图像数据。Raster.getSample 方法返回单个像素的样本。getSamples 方法返回图像的特定区域带。

除这些方法外，还可通过 Raster 类的实例变量访问数据缓冲区和样本模型。这些对象提供访问和解释 Raster 像素数据的其它方法。

5.4.4 WritableRaster 子类

WritableRaster 子类提供设置像素数据和样本的方法。与 BufferedImage 关联的 Raster 实际上是 WritableRaster，因而可提供操作其像素数据的完全访问权限。

5.5 图像数据和 DataBuffer

从属于 Raster 的 DataBuffer 代表一个图像数据数组。当直接创建 Raster 或通过 BufferedImage 构造函数创建 Raster 时，应指定以像素为单位的高度和宽度，同时指定图像数据的 SampleModel。 这些信息将用于创建相应数据类型和大小的 DataBuffer。

DataBuffer 有三个子类，分别代表不同类型的数据单元：

- DataBufferByte（表示 8 位的值）
- DataBufferInt（表示 32 位的值）
- DataBufferShort（表示 16 位的值）
- DataBufferUShort（表示无符号短值）

如前所述，单元是数据缓冲区数组的非连续成员，而组件或样本则是共同构成像素的非连续值。 在 DataBuffer 的特定单元类型和 SampleModel 代表的特定类型像素间可以有不同的映射关系。不同的 SampleModel 子类的职责就是实现上述映射关系，并提供从特定 DataBuffer 中获取指定像素的方法。

DataBuffer 构造函数提供了创建特定大小和组数的缓冲区的方法。

尽管可以直接从 DataBuffer 中访问图像数据，但利用 Raster 方法和 WritableRaster 类时通常更为便利。

5.6 从 SampleModel 中抽取像素数据

抽象的 SampleModel 类定义的方法可在不知道基本数据储存方式的情况下抽取图像样本。该类所提供的域可跟踪相关联的 DataBuffer 中图像数据的高度和宽度，同时可描述该缓冲区中带的数量和数据类型。SampleModel 方法提供的图像数据表现为像素集，其中每个像素由许多样本或组件构成。

java.awt.image 包提供五种类型的样本模型：

- ComponentSampleModel -- 用于从满足下列条件的图像中抽取像素：该图像将样本数据存储于一组 DataBuffer 中各自独立的数组单元中。
- BandedSampleModel -- 用于从满足下列条件的图像中抽取像素：该图像将每个样本存储于各自独立的数据单元中，而图像带则存储于一组连续数据单元中。
- PixelInterleavedSampleModel -- 用于从满足下列条件的图像中抽取像素：该图像将每个样本存储于各自独立的数据单元中，而像素则存储于一组连续数据单元中。

- MultiPixelPackedSampleModel -- 用于从满足下列条件的单个带形图像中抽取像素：该图像将多个单样本像素存储于一个数据单元中。
- SinglePixelPackedSampleModel -- 用于从满足下列条件的图像中抽取样本：该图像在第一组 DataBuffer 的某个数据数组单元中存储单个像素的样本数据。

取决于数据源，SampleModel 提供的像素数据可能与特定颜色模型的颜色数据表示法直接相关或无关。例如在摄影图像数据中，样本可能代表 RGB 数据；而在医学图像数据中，样本可能代表不同类型的数据（例如温度或骨骼密度）。

访问图像数据的方法有三种。getPixel 方法将整个像素以一个数组的形式返回，每个样本一项。getDataElement 方法可以访问 DataBuffer 中所存的原始未解释数据。getSample 方法可以访问特定带的像素组件。

5.7 ColorModel 和颜色数据

除了包括用于管理图像数据的 Raster 对象之外，BufferedImage 类还包括 ColorModel，用于将该数据解释为颜色像素值。抽象 ColorModel 类定义了将图像的像素数据转换为颜色值的方法，该颜色值在与其关联的 ColorSpace 中。

java.awt.image 包提供四种类型的颜色模型：

- PackedColorModel--抽象 ColorModel 类，代表具有颜色组件的的像素值，这些颜色组件直接嵌入整型像素的位中。DirectColorModel 是 PackedColorModel 的子类。
- DirectColorModel--一种 ColorModel，代表具有 RGB 颜色组件的像素值，这些 RGB 颜色组件直接嵌入像素本身的位中。DirectColorModel 模型类似于 X11 TrueColor 视觉效果。
- ComponentColorModel--一种 ColorModel，可以处理任意 ColorSpace 和匹配该 ColorSpace 的颜色组件数组。
- IndexColorModel--一种 ColorModel，代表 sRGB 颜色空间中固定颜色映射索引的像素值。

ComponentColorModel 和 PackedColorModel 在 JDK 1.2 版中是新增的。

SampleModel 根据 DataBuffer 中的数据将像素提供给 ColorModel，然后由 ColorModel 将该像素解释为某种颜色。

5.7.1 查询表

查询表包含一个或多个通道或图像组件的数据，例如独立的 R（红）、G（绿）和 B（蓝）数组。java.awt.image 包定义了两种类型的查询表，二者均扩展抽象 LookupTable 类：一种查询表包含 byte 型数据，另一种包含 short 型数据（ByteLookupTable 和 ShortLookupData）。

5.8 图像处理与增强功能

图像程序包提供一对接口，用来定义 BufferedImage 和 Raster 对象上的操作：BufferedImageOp 和 RasterOp。

实现这些接口的类包括 AffineTransformOp、BandCombineOp、ColorConvertOp、ConvolveOp 和 LookupOp、RescaleOp。这些类用于图像的几何变换、钝化、锐化、增强对比、阈值和图像颜色校正。

图 5-4 演示了边缘检测和增强功能。该操作强调了图像内亮度的强烈变化。边缘检测常用于医学图像和映射，可用来将图像中相邻部分的对比度增强，从而使观察者可以看到更细微的差异。

Figure 5-4 边缘检测与增强功能

下列代码说明如何实现边缘检测：

```
float[] elements = { 0.0f, -1.0f, 0.0f,
                     -1.0f,  4.f, -1.0f,
                      0.0f, -1.0f, 0.0f};
...

BufferedImage bimg = new
BufferedImage(bw,bh,BufferedImage.TYPE_INT_RGB);
Kernel kernel = new Kernel(3, 3, elements);
ConvolveOp cop = new ConvolveOp(kernel, ConvolveOp.EDGE_NO_OP, null);
cop.filter(bi,bimg);
```

图 5-5 演示了查询表操作。查询操作可用来改变像素内的单个组件。

图 5-5 查询表操作

下列代码说明如何实现查询表操作：

```
byte reverse[] = new byte[256];
for (int j=0; j<200; j++){
        reverse[j]=(byte)(256-j);}
ByteLookupTable blut=new ByteLookupTable(0, reverse);
LookupOp lop = new LookupOp(blut, null);
lop.filter(bi,bimg);
```


图 5-6 演示了缩放功能。缩放功能可以增加或减少图像中所有像素点的强度，可用来增加一幅模糊图像的动态范围，将看上去模糊而缺乏层次的区域中的细节显示出来。

图 5-6 缩放

下列代码段说明如何实现缩放：

```
RescaleOp rop = new RescaleOp(1.5f, 1.0f, null);
rop.filter(bi,bimg);
```

5.8.1 使用图像处理操作

变换是构成空间过滤算法的基础。变换将图像中每个像素的值与相邻像素的值进行加权或平均，从而使每个输出像素按照内核算法指定的方式被相邻像素所影响。图 5-7 演示了变换。

图 5-7 变换形成的钝化效果

下列代码段说明如何使用图像处理类 ConvolveOp。该例中，源图像中每个像素的值均由其周围八个像素的值平均而来。

```
float weight = 1.0f/9.0f;
float[] elements = new float[9]; // create 2D array

// 将九个相等元素添入矩阵
for (i = 0; i < 9; i++) {
        elements[i] = weight;}

// 使用矩阵元素作为参数创建一个 Kernel（内核）
private Kernel myKernel = new Kernel(3, 3, elements);
public ConvolveOp simpleBlur = new ConvolveOp(myKernel);

//sourceImage 和 destImage 是 BufferedImage 的实例
simpleBlur.filter(sourceImage, destImage) // 钝化图像
```

变量 simpleBlur 包含一个新 ConvolveOp 实例，后者在 BufferedImage 或 Raster 上实现钝化操作。不妨假设 sourceImage 和 destImage 是 BufferedImage 的两个实例。如果调用 filter（ConvolveOp 类的内核算法），它将确定目标图像中每个像素的值，方法是将源图像中相应像素的值与其周围八个像素的值一起平均。

在该例中，变换内核可以表示为下列矩阵，其中矩阵元素被指定为四个有意义的数字：

$$K = \begin{bmatrix} 0.1111 & 0.1111 & 0.1111 \\ 0.1111 & 0.1111 & 0.1111 \\ 0.1111 & 0.1111 & 0.1111 \end{bmatrix}$$

当图像变换时，目标图像中每个像素的值都通过以内核矩阵为权数将该像素值与周围像素的值平均而计算出来。图像的每个通道中都执行此操作。

下列公式显示当执行变换操作时内核中的权数如何与源图像中的像素相关联。内核中每个值都与图像中某一空间位置有关。

$$K = \begin{bmatrix} i-1,j-1 & i,j-1 & i+1,j-1 \\ i-1,j & i,j & i+1,j \\ i-1,j+1 & i,j+1 & i+1,j+1 \end{bmatrix}$$

内核中权数与相应源像素值的乘积之和就是目标像素值。对于许多简单操作而言，内核矩阵是方形对称阵，而且它的权数和为一。[1]

该例中的变换矩阵较为简单。它赋予源图像中每个像素以相同权数。通过选择对源图像进行较高或较低水平加权的内核，程序可以增加或降低目标图像的亮度。在 ConvolveOp 构造函数中进行设置的 Kernel 对象将确定所执行的过滤类型。通过设置其它值即可执行不同类型的变换，包括钝化（如高斯钝化、放射状钝化和运动钝化）、锐化和平滑操作。 图 5-8 演示了利用变换进行锐化的过程。

Figure 5-8 用变换实现锐化

下列代码段说明如何用变换实现锐化：

```
float[] elements = { 0.0f, -1.0f, 0.0f,
                     -1.0f,  5.f, -1.0f,
                      0.0f, -1.0f, 0.0f};
...

Kernel kernel = new Kernel(3,3,elements);
ConvolveOp cop = new ConvolveOp(kernel, ConvolveOp.EDGE_NO_OP, null);
cop.filter(bi,bimg);
```

第 6 章

颜色

彩色图像处理是任何图形系统的基本组件之一，而且经常是图像模型极为复杂的原因之一。Java 2D API 提供对高质量颜色输出的支持，不仅易于使用而且允许高级用户进行复杂的颜色处理。

Java 2D API 中的主要颜色管理类是 ColorSpace、Color 和 ColorModel：

- ColorSpace 类表示颜色度量系统，通常使用三个独立的数字值或组件。ColorSpace 类中包含在当前颜色空间和两个标准颜色空间 CIEXYZ 和 RGB 之间进行转换的方法。
- Color 对应一种固定颜色，并根据其在特定 ColorSpace 中的组件进行定义。要绘制某种颜色的形状（例如红色），应将代表该颜色的 Color 对象传给 Graphics2D 上下文。Color 在 java.awt 包中进行了定义。
- ColorModel 描述了将像素值映射到颜色上的特定方法。通常，ColorModel 与 Image 或 BufferedImage 有关，而且提供必要的信息来正确解释像素值。Color 在 java.awt.image 包中进行了定义。

6.1 类

类	说明
ColorSpace	识别 Color 对象、Image、BufferedImage 或 GraphicsDevice 的颜色空间。提供在 RGB 和 CIEXYZ 颜色空间之间变换的方法。
ICC_ColorSpace	扩展：ColorSpace 表示基于“ICC 配置格式规范”的、与设备无关和与设备相关的颜色空间。
ICC_Profile	表示基于“ICC 配置格式规范”的、与设备无关和与设备相关的颜色空间的颜色监视数据。
ICC_ProfileGray	扩展：ICC_Profile

	代表颜色空间中的灰阶类型。
ICC_ProfileRGB	扩展：ICC_Profile 代表颜色空间中的 RGB 类型。

6.2 颜色概念

ColorModel 用来解释图像中的像素数据，包括将图像带中的组件映射到一个特定颜色空间的组件。它也可实现从已打包的像素数据中抽取像素组件、使用掩码从单个带中检索多个组件及通过查询表转换像素数据。

要确定图像中特定像素的颜色值，需要知道每个像素中颜色信息的编码方式。与图像相关联的 ColorModel 中封装了在像素值与颜色组件之间转换所需的数据和方法。

在 JDK 1.1 中，除了 DirectColorModel 和 IndexColorModel 之外，Java 2D API 还提供了两种颜色模型：

- ComponentColorModel 可处理任意 ColorSpace 和颜色组件的数组，从而与 ColorSpace 相匹配。该模型可用来表示多种 GraphicsDevices 上的颜色模型。
- 对于表示将颜色组件直接嵌入整数像素位的像素值的模型而言，PackedColorModel 是一个基本类。PackedColorModel 类存储着打包信息，用于描述从颜色通道中抽取颜色和 alpha 组件的方式。JDK 1.1 中的 DirectColorModel 即为一个 PackedColorModel。

6.2.0.1 ColorSpace

ColorSpace 对象代表一个颜色度量系统，通常，它采用三个独立的数值（例如 RGB 和 CMYK 都是颜色空间）。ColorSpace 对象用作 colorspace 标记，标识 Image、BufferedImage 或 GraphicsConfiguration 中 Color 或 ColorModel 对象的特定颜色空间。ColorSpace 提供在特定颜色空间与 sRGB 和已定义的 CIEXYZ 颜色空间中的 Colors 之间互相转换的方法。

所有 ColorSpace 对象都必须能将所代表的颜色空间中的颜色映射到 sRGB 中，并将 sRGB 颜色变换为所代表的颜色空间。由于每个 Color 都包含 ColorSpace 对象（显式设置或缺省设置），因此每个 Color 也可转换为 sRGB。每个 GraphicsConfiguration 都与一个 ColorSpace 对象相关联，而后者则与 ColorSpace 相关联。通过使用过渡颜色空间 sRGB 进行映射，任何颜色空间所指定的颜色都可在任意设备上显示。

该过程所用的方法为 toRGB 和 fromRGB：

- toRGB 将所表示的颜色空间中的 Color 变换为 sRGB 中的 Color。

- fromRGB 抽取 sRGB 中的 Color 并将其变换到所表示的颜色空间中。

虽然 sRGB 之间的映射总可起作用，但这并非总是最好的解决方案。首先，sRGB 不能表示 CIEXYZ 全部色阶中的每种颜色。如果在某种与 sRGB 有不同色阶（可表示颜色的范围）的颜色空间中指定了一种颜色，则采用 sRGB 作为过渡颜色空间将导致信息的丢失。为解决该问题，ColorSpace 类可使颜色与另一种颜色空间相互映射，即“转换空间”CIEXYZ。

toCIEXYZ 和 fromCIEXYZ 方法将颜色值从所表示的颜色空间映射到转换空间。这些方法以相当高的精确度支持任两种颜色空间之间的转换，每次转换一个 Color。但将来 Java 2D API 实现有望支持基于基本平台颜色管理系统的高性能转换功能，从而可处理整个图像（参见第 67 页“图像处理”中的 ColorConvertOp）。

图 6-1 和图 6-2 说明了转换 CMYK 颜色空间中的指定颜色以使之能在 RGB 显示器上显示出来的过程。图 6-1 显示了一种通过 sRGB 进行的映射。正如该图中所说明的，由于色阶不匹配，从 CMYK 颜色到 RGB 颜色的转换不够精确。

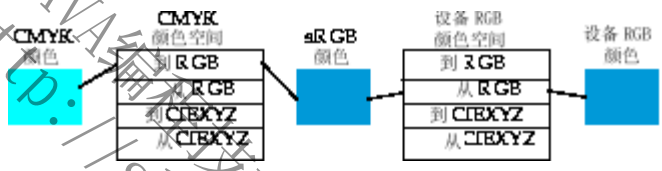


图 6-1 通过 sRGB 进行映射

图 6-2 显示了采用 CIEXYZ 作为转换空间时的同一过程。使用 CIEXYZ 时，颜色转换较为精确。

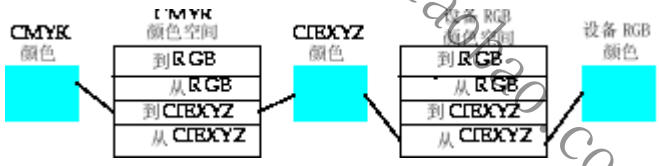


图 6-2 通过 CIEXYZ 进行映射

6.2.0.2 ICC_Profile 和 ICC_ColorSpace

ColorSpace 实质上是一个抽象类。Java 2D API 基于 ICC_Profile 类所代表的 ICC 描述数据而提供了一个实现（即 ICC_ColorSpace）。只要已经实现上文所讨论的方法，就可以定义自己的子类来表示任何颜色空间。然而，很多开发人员可使用缺省 sRGB ColorSpace 或 ICC 监视程序所表示的颜色空间（例如显示器和打印机的监视程序），也可使用已嵌入图像数据的监视程序。

第 90 页的“ColorSpace” 说明了 ColorSpace 对象表示颜色空间的方式以及其中的颜色与转换空间相互映射的过程。颜色管理系统经常用于处理颜色空间之间的映射。颜色管理系统 (CMS) 一般管理 ICC 监视程序，而后者与 ColorSpace 对象相似；ICC 监视程序对输入空间和连接空间进行监视，同时定义它们之间的映射关系。颜色管理系统非常擅长于计算如何将标为某种监视程序的颜色映射到标为另一种监视程序的颜色空间。

Java 2D API 定义了一个名为 ICC_Profile 的类，可以存储 ICC 监视程序的数据。ICC_ColorSpace 是抽象类 ColorSpace 的实现。用户可以利用 ICC_Profiles 构造 ICC_ColorSpace 对象（限制：并非所有 ICC 监视程序都适合于定义 ICC_ColorSpace）。

ICC_Profile 提供几个与特定颜色空间类型相对应的子类（例如 ICC_ProfileRGB 和 ICC_ProfileGray）。ICC_Profile 的每个子类都有已定义的输入空间（例如 RGB 空间）和已定义的的连接空间（例如 CIEXYZ）。对于不同的设备（例如扫描仪、打印机和显示器），Java 2D API 可使用平台的 CMS 来访问监视程序。也可用 CMS 来查找监视程序之间的最佳映射。

6.2.1 颜色描述

Color 类可提供特定颜色空间中颜色的描述。Color 的实例包含颜色组件的值和 ColorSpace 对象。由于在创建新的 Color 实例时除了颜色组件外还可指定 ColorSpace 对象，因此 Color 类能处理任何颜色空间中的颜色。

Color 类有许多支持名为 sRGB 的标准 RGB 颜色空间（参见 http://www.w3.org/pub/WWW/Graphics/Color/sRGB.html）的方法。sRGB 是 Java 2D API 的缺省颜色空间。Color 类定义的几个构造函数省略了 ColorSpace 参数。这些构造函数假定在 sRGB 中定义颜色的 RGB 值，并使用 ColorSpace 的缺省实例来表示该空间。

Java 2D API 将 sRGB 视为编程人员的便利工具，而不是颜色转换的参考颜色空间。 很多应用程序主要与 RGB 图像和显示器有关，因此定义标准 RGB 颜色空间将使这类应用程序的编写更为简单。ColorSpace 类定义了方法 toRGB 和 fromRGB，可以帮助开发人员方便地在此标准空间中检索颜色。这些方法并不打算用于高精度的颜色修正和转换。有关详细信息，参见第 90 页“ColorSpace”。

要创建 sRGB 以外的颜色空间中的颜色，请使用 Color 构造函数来获取 ColorSpace 对象和浮点数组。它们可用来表示该空间中相应的颜色组件。ColorSpace 对象标识颜色空间。

要显示某种颜色的矩形（例如青色），需要一种方法来向系统描述该颜色。描述颜色有多种方法。例如，可以将某种颜色描述为红色、绿色和蓝色 (RGB) 组件的集合，或者是青色、品红色、黄色和黑色 (CMYK) 组件的集合。我们将这些指定颜色的技术称之为*颜色空间*。

或许您已知道，计算机屏幕上的颜色是通过混合不同数量的红光、绿光和蓝光而生成的。因此，对于计算机显示器上的图像处理，使用 RGB 颜色空间比较标准。同样，四色印刷采用青色、品红色、黄色和黑色的油墨在印刷页上生成颜色，印刷色按 CMYK 颜色空间的百分率指定。

随着计算机显示器和彩色打印的日渐流行，RGB 和 CMYK 颜色空间被普遍用来描述颜色。但是，两种颜色空间都有一个重要缺陷：与设备相关。打印机所用的青色油墨可能与另一台所用的青色油墨不完全匹配。同样，以 RGB 颜色描述的某种颜色在一台显示器上可能呈现蓝色，而在另一台上则会略带紫色。

6.2.2 通过 **sRGB** 和 **CIEXYZ** 进行颜色映射

Java 2D API 将 RGB 和 CMYK 视为颜色空间。特定的显示器及其特定的磷光体决定了其自身的 RGB 颜色空间。同样，特定的打印机有其自身的 CMYK 颜色空间。通过与设备无关的颜色空间，可以将不同的 RGB 或 CMYK 颜色空间互相联系起来。

国际照明委员会（CIE）已经制定了颜色与设备无关的规格标准。最常用的与设备无关的颜色空间是由 CIE 开发的三元 XYZ 颜色空间。采用 CIEXYZ 指定颜色时，将不再与设备相关。

不幸的是，用 CIEXYZ 颜色空间描述颜色并非总是实用 -- 用其它颜色空间描述颜色有时会更有优势。当用与设备相关的颜色空间（例如 RGB 空间）来表示颜色时，为获得一致的结果，需要显示出 RGB 空间和与设备无关的空间（例如 CIEXYZ）之间的联系。

颜色空间之间相互映射的办法之一，就是将描述与设备相关和与设备无关空间之间关系的信息附加到空间中。该附加信息称为*监视程序*。颜色描述的一个常用类型是国际颜色协会开发的“ICC 颜色监视程序”。有关详细信息，参见“ICC 监视程序格式规范（3.4 版）”。用户可从下面的网址得到：http://www.color.org。

图 6-3 说明了如何将一种纯色和一幅扫描图像传送到 Java 2D API 中，以及不同输出设备对它们的显示效果。如图 6-3 所示，输入颜色和图像都有附加的监视程序。

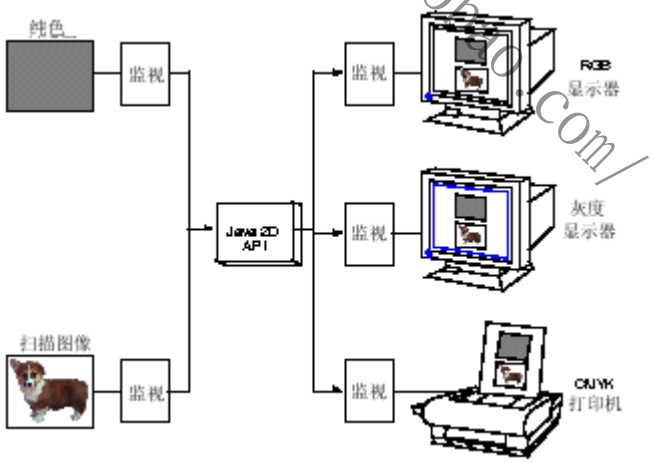


图 6-3 使用监视程序在颜色空间之间映射

6.2.2.1 颜色匹配

只要 API 有正确指定的颜色，它就必然会将该颜色复制到输出设备上（例如显示器或打印机）。这些设备有其自身的图像处理特征，因此必须加以考虑以确保得到正确的结果。还有一个与各输出设备相关联的监视程序，描述为得到正确结果而需要进行的颜色转换。

为获得一致而精确的颜色，要求输入颜色和输出设备以标准的颜色空间来描述。例如，可将某种输入颜色由其原来的颜色空间映射到标准的、与设备无关的空间，然后由该空间映射到输出设备的颜色空间。很多情况下，颜色变换模拟 (*x*, *y*) 坐标空间中图形对象的变换。在两种情况下，都使用变换来指定“标准”空间中的坐标，然后将该坐标映射到特定设备空间进行输出。

Java 打印 API 允许应用程序：

- 打印所有的 AWT 和 Java 2D 图形，包括复合图形与图像。
- 控制文档复合功能，如软核对、逆序打印和按册打印。
- 调用打印机指定功能，例如双面打印和装订。
- 在任意操作平台上打印（例如 Windows 和 Solaris 操作系统）。这包括直接连接到计算机的打印机以及平台软件可通过网络打印协议进行访问的打印机。

JDK 1.2 打印 API 及其实现中并不支持以上所有功能。API 未来的版本中将支持所有功能。例如，通过扩大应用程序所控制打印作业的指定属性集，可添加额外的打印机控制。

7.1 接口和类

接口	说明
Printable	Printable 接口由每个 *page painter* （页画笔）来实现，页画笔是被打印系统调用来绘制页的应用程序类。系统调用页画笔中的 print 方法来请求对页进行绘制。
Pageable	Pageable 接口的实现是通过打印系统要打印的文档来完成的。通过 Pageable 方法，系统可以确定文档的页数、每页的格式以及绘制各页所用的页画笔。

PrinterGraphics	实现 PrinterGraphics 接口的是被页画笔用来对页进行绘制的 Graphics2D 对象。 这样，应用程序可以获取正在控制打印的 PrinterJob 对象。

类	说明
Book	实现：Pageable 代表具有不同的页面格式及页画笔的文档。该类使用 Pageable 接口与 PrinterJob 交互。
PageFormat	描述要打印页面的大小和方向及打印所用的 Paper。例如，*纵向*和*横向*纸张方向由 PageFormat 代表。
Paper	描述打印纸的物理特征。
PrinterJob	负责控制打印的类。应用程序调用 PrinterJob 方法来建立打印作业、向用户显示打印对话（可选）以及打印列于作业中的页。

7.2 打印的概念

Java 打印 API 是以*回调*打印模型为基础的，打印系统（而不是应用程序）即在该模型中控制打印。应用程序提供要打印文档的信息，而打印系统则根据需要请求应用程序对要打印的每一页进行绘制。

打印系统可能请求多次绘制同一页或绘制时不按页的顺序进行。应用程序必须能够生成正确的页面图像，而无论打印系统请求的是哪一页。由此看来，打印系统类似于窗口工具包，它可以在任一时间、按任何顺序重新布置组件。

回调打印模型比传统的由应用程序驱动的打印模型更为灵活，而且所支持的系统和打印机范围更广。例如，如果打印机堆栈以逆序输出打印页，则打印系统也可以要求应用程序生成逆序页，从而使最终堆栈的读取顺序正确。

该模型还允许应用程序在内存或硬盘空间不足时将全页位图从计算机打印到位图打印机，以形成缓冲。此时，页面将分为一系列小位图或*带*。例如，如果内存只够存储十分之一的页，则该页将分为十个带。打印系统要求应用程序将该页绘制十次，每次填充一个带。应用程序无需注意带的数目或大小，而只须被请求时能够绘制每一页。

7.2.1 支持打印

为支持打印，应用程序必须执行两项任务：

- 作业控制--初始化及管理打印作业。
- 图像处理--在打印系统请求时绘制每一页。

作业控制

用户经常通过在应用程序中单击按钮或选择菜单项来初始化打印作业。当用户触发打印操作时，应用程序创建 PrinterJob 对象并利用它来管理打印进程。

应用程序负责设置打印作业、向用户显示打印对话框，以及启动打印进程。

图像处理

打印文档时，一旦打印系统提出请求，应用程序就必须对所需的每一页进行绘制。为了支持该机制，应用程序提供了实现 Printable 接口的*页画笔*。当打印系统需要绘制某页时，将调用页画笔的 print 方法。

print 方法被调用时将获得 Graphics 上下文，以便用来绘制页面图像。该方法还将获得用于指定页面几何布局 PageFormat 对象，以及用于标识该页在打印作业中所处位置的整数*页索引*。

打印系统同时支持 Graphics 绘制和 Graphics2D 绘制。为打印 Java 2D Shape、Text 和 Image，需要将传给 print 方法的 Graphics 对象进行强制类型转换，使其成为 Graphics2D。

要打印带有不同页面格式和页画笔的文档，请使用*可分页作业*。要创建可分页作业，可以使用 Book 类或自己的 Pageable 接口实现。要实现简单的打印操作，无需使用可分页的打印作业；只要所有页的页面格式和画笔都相同，就只需用 Printable。

7.2.2 页画笔

页画笔的主要任务是利用打印系统提供的图形上下文去绘制页。页画笔实现 Printable.print 方法：

```
public int print(Graphics g, PageFormat pf, int pageIndex)
```

传递给 print 方法的图形上下文或者是 Graphics 的实例，或者是 Graphics2D 的实例，这取决于 Java 虚拟机中加载的程序包。要使用 Graphics2D 功能，可将 Graphics 对象强制转换为 Graphics2D。传递给 print 的 Graphics 实例还需实现 PrinterGraphics 接口。

传递给 Printable 的 PageFormat 描述了要打印页的几何特征。传递给 print 的图形上下文坐标系被固定于页面上：坐标原点位于纸张左上角，*X* 轴向右递增，*Y* 轴向下递增，单位是 1/72 英寸。如果是纵向打印，则 x 轴对应于纸张的“宽度”而 y 轴对应于“高度”（纸张的高度通常但非绝对会超过其宽度）。横向打印则相反：x 轴对应于“高度”而 y 轴对应于“宽度”。

由于许多打印机都无法打印整个页面，因此 PageFormat 将指定打印页的*可成像区*：该区域是打印页中能够被安全绘制的部分。可成像区的指定并不会改变坐标系，其目的是使页的内容能够得到绘制，而不是落到打印机无法打印的区域中。

传递给 print 的图形上下文中有一个剪切区域，用于描述应绘制的可成像区。在上下文中描绘整个页总是安全的，因为打印系统将处理必要的剪切操作。要消除页面中无需打印的部分，可以使用剪切区域来限制要绘制的可成像区。要从图形上下文中获得剪切区域，请调用 Graphics.getClip。最好使用剪切区域来减少页面中无需绘制的部分。

有时，将整个打印操作放置到“后台打印”比较好，这样用户就可以在绘制打印页的同时与应用程序进行交互。要实现这一点，请在另一个的线程中调用 PrinterJob.print。

如果可能，应避免进行需要知道以前图像内容的图形操作，例如 copyArea、setXOR 和图形复合。这些操作可能降低绘制速度并产生不一致的效果。

7.2.3 可打印的作业和可分页的作业

Printable（可打印的）作业是最简单的打印方式。页画笔只需要一种；应用程序提供一个类来实现 Printable 接口。打印时，打印系统调用该页画笔的 print 方法来绘制每一页。打印页按顺序被请求，以页索引 0 为开始。但是，页画笔可能被要求在进入下一页之前对每页都进行多次绘制。最后一页打印完毕后，页画笔的 print 方法返回 NO_SUCH_PAGE。

在 Printable 作业中：

- 所有页都使用相同的页画笔和 PageFormat。显示打印对话框时将不显示文档中的页数，因为打印系统中没有得到该信息。
- 打印系统始终要求页画笔按照索引顺序打印每一页，以索引 0 处的页开始，不跳过任一页。例如，如果用户要求打印文档的第 2 页和第 3 页，页画笔将打印索引 0、1 和 2 处的页。打印系统可能请求在绘制下一页之前多次绘制当前页。
- 当文档处理结束时，页画笔将通知打印系统。
- 所有页画笔都将在相同的线程中被调用。
- 某些打印系统可能无法完成理想的输出。例如，打印机中的页堆栈可能出现顺序错误，或者因请求多个副本而无法对其进行核对。

Pageable（可分页的）作业比 Printable 作业更为灵活。与 Printable 作业中的页不同，Pageable 作业中的页在布局和实现上可以互不相同。要管理 Pageable 作业，可使用 Book 类或自己的 Pageable 类实现。通过 Pageable，打印系统可以确定要打印页的数目、每页所

用的页画笔以及每页的 PageFormat。应用程序如果需要打印结构与格式都设计好的文档，可以使用 Pageable 作业。

在 Pageable 作业中：

- 不同的页可以使用不同的页画笔和 PageFormats。
- 打印系统可以要求页画笔以任意顺序打印，还可以跳过几页。 例如，用户要打印文档的第 2 页和第 3 页，页画笔将打印索引 1 和 2 处的页，而跳过索引 0。
- Pageable 作业不必知道文档中有多少页要打印。与 Printable 作业不同，该作业中的页可以按任意顺序绘制。打印序列中可能出现间隔，而且打印系统可能请求在打印下一页之前对当前页进行多次绘制。例如，打印文档第 2 页和第 3 页的请求可能导致请求按如下调用序列的打印页：索引 2、2、1、1 和 1。

7.2.4 PrinterJob 的一般生命期

应用程序通过一系列步骤来控制 PrinterJob 对象完成打印作业。最简单的步骤如下所示：

1. 通过调用 PrinterJob.getPrinterJob 来获得新的 PrinterJob 对象。
2. 确定打印时使用何种 PageFormat。可以通过调用 defaultPage 来获得缺省的 PageFormat，或者可以通过调用 pageDialog 来显示对话框。用户可在对话框中指定一种格式。
3. 指定要打印到 PrinterJob 的作业的特性。对于 Printable 作业，调用 setPrintable；对于 Pageable 作业，调用 setPageable。 注意：Book 对象最适于传递给 setPageable。
4. 指定附加的打印作业属性，例如要打印的副本数目或者要打印在标题页上的作业名。
5. 调用 printDialog 来显示对话框。此项可选。此对话框的内容和外观随系统平台和打印机的不同而有所不同。在大多数系统平台上，用户可使用此对话框来改变打印机选项。如果用户取消打印作业，则 printDialog 方法返回 FALSE。
6. 调用 Printerjob.print 来打印作业。该方法根据相应的页画笔调用 print 。

如果发生下列情况，可中断打印作业：

- PrinterException 被抛出--该异常被 print 方法捕捉，同时作业中断。如果页画笔检测到致命错误，则抛出异常 PrinterException。
- PrinterJob.cancel 被调用--打印循环终止，同时打印作业取消。cancel 方法可由另一个线程调用，该线程显示对话框并允许用户通过单击按钮来取消打印。

在打印作业停止前生成的页可能被打印也可能不被打印。

当 print 方法返回时，打印作业通常没有结束。打印机驱动器、打印服务器或打印机本身仍然在工作。PrinterJob 对象的状态可能并不反映实际打印作业的状态。

由于 PrinterJob 的状态会在它的生命期内发生变化，因此有时激活某些方法是非法的。例如，调用了 print 之后再调用 setPageable 没有任何意义。检测到非法调用后，PrinterJob 将抛出 java.lang.IllegalStateException 异常。

7.2.5 对话框

Java 打印 API 需要应用程序显式调用用户界面对话框。这些对话框可能由平台软件（例如 Windows）提供，或者由 JDK 软件实现。对于交互型应用程序，一般习惯上使用这种对话框。对于打印型应用程序，对话框则不是必需的。例如，当自动生成并打印晚间数据库报告时不需要显示对话框。不需要用户交互的打印作业有时称为*静默*打印作业。

页面设置对话框

通过显示页面设置对话框，用户可以更改 PageFormat 中的页面设置信息。要显示页面设置对话框，请调用 PrinterJob.pageDialog。该对话框用传递给 pageDialog 的参数进行初始化。如果用户对话框中的单击“确定”按钮，则 PageFormat 实例将被复制并按用户的选择更改，然后被返回。如果用户取消对话框，则 pageDialog 将返回原来未被更改的 PageFormat。

打印对话框

通常，在打印菜单项或按钮被激活时，应用程序会显示对话框。要显示该对话框，请调用 PrinterJob 类的 printDialog 方法。对话框中的用户选择项主要是限制在 PrinterJob 中已配备的 Printable 或 Pageable 中的页数和页面格式。如果用户单击打印对话框中的“确定”，则 printDialog 返回 TRUE。如果用户取消打印对话框，则返回 FALSE，而且打印作业被视为已取消。

7.3 打印 **Printable** 作业

要提供基本打印支持，需要：

1.实现 Printable 接口以提供页画笔来绘制要打印的每一页。
2.创建 PrinterJob。
3.调用 setPrintable 来通知 PrinterJob 应该如何打印文档。
4.调用 PrinterJob 对象上的 print，开始打印作业。

在下例中，Printable 作业打印五页，每页显示一个绿色页号。作业控制的管理在 main 方法中进行，该方法获得并控制 PrinterJob。页画笔中的 print 方法执行打印绘制。

```
import java.awt.*; import java.awt.print.*;
public class SimplePrint implements Printable
{

  private static Font fnt = new Font("Helvetica",Font.PLAIN,24);


  public static void main(String[] args)
  {
    // 获取 PrinterJob
    PrinterJob job = PrinterJob.getPrinterJob();
    // 指示 Printable 是 SimplePrint 的实例
    job.setPrintable(new SimplePrint());
    // 提供对话框
    if (job.printDialog())
    {
      // 如果用户未取消打印，则打印作业
      try { job.print(); }
      catch (Exception e)
        { /* 处理异常 */ }
    }
    System.exit(0);
  }


  public int print(Graphics g, PageFormat pf, int pageIndex)
  throws PrinterException
  {
    // 页索引 0 至 4 对应页号 1 至 5。
    if (pageIndex >= 5) return Printable.NO_SUCH_PAGE;
    g.setFont(fnt);
    g.setColor(Color.green);
    g.drawString("Page " + (pageIndex+1), 100, 100);
    return Printable.PAGE_EXISTS;
  }
}
```


7.3.1 使用 **Graphics2D** 绘制

通过首先将 Graphics 上下文强制转换为 Graphics2D，可以调用页画笔 print 方法中的 Graphics2D 功能。

下例中，页号将被绘制成红绿渐变。为此，Graphics2D 上下文中设置了 GradientPaint。

```
import java.awt.*; import java.awt.print.*;
public class SimplePrint2D implements Printable
{
  private static Font fnt = new Font("Helvetica",Font.PLAIN,24);

  private Paint pnt = new GradientPaint(100f, 100f, Color.red,
136f, 100f, Color.green, true);

  public static void main(String[] args)
  {
    // 获取 PrinterJob
    PrinterJob job = PrinterJob.getPrinterJob();
    // 指示 Printable 是 SimplePrint2D 的实例
    job.setPrintable(new SimplePrint2D());
    // 提供对话框
    if (job.printDialog())
    {
      // 如果用户未取消打印，则打印作业
      try { job.print(); }
      catch (Exception e) { /* 处理异常 */ }
    }
  System.exit(0);
  }


  public int print(Graphics g, PageFormat pf, int pageIndex)
  throws PrinterException
  {
    // 页索引 0 至 4 对应页号 1 至 5。
    if (pageIndex >= 5) return Printable.NO_SUCH_PAGE;
    Graphics2D g2 = (Graphics2D) g;
    // 使用上面定义的字体
    g2.setFont(fnt);
    // 使用上面定义的渐变颜色
    g2.setPaint(pnt);
    g2.drawString("Page " + (pageIndex+1), 100f, 100f);
    return Printable.PAGE_EXISTS;
  }
}
```


7.3.2 打印文件

如果页画笔 print 方法对同一页作用多次，则每次都生成相同的输出。

有多种方法可确保对页的重复绘制请求会生成相同的输出。例如，为确保打印系统每次请求文本文件中某页时都生成相同的输出，页画笔可以存储并重复使用各页的文件指针，或者存储实际的页数据。

在下例中将打印文本文件的“列表”。文件名作为参数传给 main 方法。PrintListingPainter 类存储文件指针（位于被要求绘制的每一新页的开始处）。当同一页被反复绘制时，文件指针将被重置到原先的位置。

```
import java.awt.*;
import java.awt.print.*;
import java.io.*;

public class PrintListing
{
  public static void main(String[] args)
  {
    // 获取 PrinterJob
    PrinterJob job = PrinterJob.getPrinterJob();
    // 询问用户页面格式（如纵向/横向）
    PageFormat pf = job.pageDialog(job.defaultPage());
    // 指示 Printable 是
    // PrintListingPainter 的实例，同时提供给定的 PageFormat
    job.setPrintable(new PrintListingPainter(args[0]), pf);
    // 打印一份
    job.setCopies(1);
    // 提供对话框
    if (job.printDialog())
    {
      // 如果用户未取消打印，则打印作业
      try { job.print(); }
      catch (Exception e) { /* 处理异常 */ }
    }
    System.exit(0);
  }
}

class PrintListingPainter implements Printable
{
  private RandomAccessFile raf;
  private String fileName;
  private Font fnt = new Font("Helvetica", Font.PLAIN, 10);
  private int rememberedPageIndex = -1;
  private long rememberedFilePointer = -1;
  private boolean rememberedEOF = false;

  public PrintListingPainter(String file)
```

```
{
  fileName = file;
  try
  {
    // 打开文件
    raf = new RandomAccessFile(file, "r");
  }
  catch (Exception e) { rememberedEOF = true; }
}

public int print(Graphics g, PageFormat pf, int pageIndex)
throws PrinterException
{
try
{
  // 用于捕获 IOException
  if (pageIndex != rememberedPageIndex)
  {
    // 第一次访问该页
    rememberedPageIndex = pageIndex;
    // 如果遇到上一页的 EOF，则结束
    if (rememberedEOF) return Printable.NO_SUCH_PAGE;
    // 保存输入文件的当前位置
    rememberedFilePointer = raf.getFilePointer();
  }
  else raf.seek(rememberedFilePointer);
  g.setColor(Color.black);
  g.setFont(fnt);
      int x = (int) pf.getImageableX() + 10;
      int y = (int) pf.getImageableY() + 12;
  // 标题行
  g.drawString("File: " + fileName+", page: " + (pageIndex+1),
                x, y);
  // 生成尽可能多的行填满可图像区
  y += 36;
  while (y + 12 < pf.getImageableY()+pf.getImageableHeight())
  {
    String line = raf.readLine();
    if (line == null)
    {
      rememberedEOF = true;
      break;
             }
      g.drawString(line, x, y);
      y += 12;
    }
```

```
      return Printable.PAGE_EXISTS;
    }
    catch (Exception e) { return Printable.NO_SUCH_PAGE;}
  }
}
```

7.4 打印 **Pageable** 作业和 **Book**

Pageable 作业便于应用程序对文档中的每页都建立清晰的表示。Book 类是 Pageables 一种便利的使用方式，但是如果 Book 不能满足要求，用户即可建立自己的 Pageable 结构。本节讲解如何使用 Book。

虽然 Pageable 作业比 Printable 作业稍复杂，但却更受欢迎，因为它使打印系统更为灵活。Pageable 的一个主要优势是文档中的打印页数可知，而且能够在打印对话框中显示给用户，因此可帮助用户选择打印页的范围或者确认指定作业是否正确。

Book 代表打印页的集合。这些页不必拥有相同的大小、方向或页画笔。例如，Book 可能包括两页信纸大小的纵向页和一页信纸大小的横向页。

Book 在最初建立时是空的。要将页添加到 Book 中，请使用 append 方法。该方法使用 PageFormat 对象，用来定义页面大小、可打印区域及实现 Printable 接口的页画笔和方向。

同一个 Book 中的多个页可以共享相同的页面格式及页画笔。append 方法被重载，以便可以通过指定另一个参数（即页数）来添加一系列具有相同属性的页。

如果不知道 Book 中的总页数，可以将 UNKNOWN_NUMBER_OF_PAGES 传递给 append 方法。然后，打印系统将以页索引递增的顺序调用页画笔，直到返回 NO_SUCH_PAGE。

setPage 方法可用来更改某一页的页面格式或页画笔。要更改的页由页索引标识其在 Book 中的位置。

请调用 setPageable 并将 Book 传入，以准备打印作业。setPageable 和 setPrintable 方法是互斥的，也就是说在准备 PrinterJob 时只能选择二者之一。

7.4.1 使用 **Pageable** 作业

下例中，Book 用来再现第一个简单打印示例（因为此例太简单，所以没必要用 Pageable 作业代替 Printable 作业，但它演示了 Book 的基本用法）。注意，您仍然必须实现 Printable 接口并执行页画笔 print 方法中的页面绘制。

```
import java.awt.*;
import java.awt.print.*;
```

```
public class SimplePrintBook implements Printable
{
  private static Font fnt = new Font("Helvetica",Font.PLAIN,24);
  public static void main(String[] args)
  {
    // 获取 PrinterJob
    PrinterJob job = PrinterJob.getPrinterJob();
    // 建立 book
    Book bk = new Book();
    bk.append(new SimplePrintBook(), job.defaultPage(), 5);
    // 将 book 传给 PrinterJob
    job.setPageable(bk);
    //提供对话框
    if (job.printDialog())
    {
      // 如果用户未取消打印，则打印作业
      try { job.print(); }
      catch (Exception e) { /* 处理异常 */ }
    }
    System.exit(0);
  }

  public int print(Graphics g, PageFormat pf, int pageIndex)
  throws PrinterException
  {
    g.setFont(fnt);
    g.setColor(Color.green);
    g.drawString("Page " + (pageIndex+1), 100, 100);
    return Printable.PAGE_EXISTS;
  }
}
```


7.4.2 使用多个页画笔

下例中使用了两个不同的页画笔：一个用于封面页而另一个用于内容页。封面页横向打印而内容页纵向打印。

```
import java.awt.*;
import java.awt.print.*;

public class PrintBook
{
  public static void main(String[] args)
```

```
  {
    // 获取 PrinterJob
    PrinterJob job = PrinterJob.getPrinterJob();
    // 创建横向格式
    PageFormat pfl = job.defaultPage();
    pfl.setOrientation(PageFormat.LANDSCAPE);
    // 建立 book
    Book bk = new Book();
    bk.append(new PaintCover(), pfl);
    bk.append(new PaintContent(), job.defaultPage(), 2);
    // 将 book 传给 PrinterJob
    job.setPageable(bk);
    // 提供对话框
    if (job.printDialog())
    {
      // 如果用户未取消打印，则打印作业
      try { job.print(); }
      catch (Exception e) { /* 处理异常 */ }
    }
  System.exit(0);
  }
}

class PaintCover implements Printable
{
  Font fnt = new Font("Helvetica-Bold", Font.PLAIN, 72);

  public int print(Graphics g, PageFormat pf, int pageIndex)
  throws PrinterException
  {
    g.setFont(fnt);
    g.setColor(Color.black);
         int yc = (int) (pf.getImageableY() +
              pf.getImageableHeight()/2);
    g.drawString("Widgets, Inc.", 72, yc+36);
    return Printable.PAGE_EXISTS;
  }
}
class PaintContent implements Printable
{
  public int print(Graphics g, PageFormat pf, int pageIndex)
  throws PrinterException
  {
    Graphics2D g2 = (Graphics2D) g;
    int useRed = 0;
   int xo = (int) pf.getImageableX();
```

```
        int yo = (int) pf.getImageableY();
    // 用圆环或正方形填充页（红色和绿色交替）
        for (int x = 0; x+28 < pf.getImageableWidth(); x += 36)
    for (int y = 0; y+28 < pf.getImageableHeight(); y += 36)
    {
      if (useRed == 0) g.setColor(Color.red);
      else g.setColor(Color.green);
      useRed = 1 - useRed;
      if (pageIndex % 2 == 0) g.drawRect(xo+x+4, yo+y+4, 28, 28);
      else g.drawOval(xo+x+4, yo+y+4, 28, 28);
    }
    return   Printable.PAGE_EXISTS;
  }
}
```