Article

Painting in AWT and Swing

AWT 与 Swing 中的绘制

Painting in AWT and Swing

AWT 与 Swing 中的绘制

Good Painting Code Is the Key to App Performance

好的绘制代码是决定应用程序性能的关键

By Amy Fowler

翻译：Azure

*原文：**http://java.sun.com/products/jfc/tsc/articles/painting/***

In a graphical system, a *windowing toolkit* is usually responsible for providing a framework to make it relatively painless for a graphical user interface (GUI) to render the right bits to the screen at the right time.

在图形系统中，一个窗口工具包通常负责提供一个框架，为用户图形接口（GUI）在恰当的时间在屏幕上渲染恰当的色位降低难度。

Both the AWT (abstract windowing toolkit) and Swing provide such a framework. But the APIs that implement it are not well understood by some developers -- a problem that has led to programs not performing as well as they could.

AWT（抽象窗口工具包）和 Swing 都提供了这样一个框架。但是有些开发人员并不太明白它的 API 实现 — 以致程序表现的没有它们可以做到的那么好。

This article explains the AWT and Swing paint mechanisms in detail. Its purpose is to help developers write correct and efficient GUI painting code. While the article covers the general paint mechanism (where and when to render), it does not tell how to use Swing's graphics APIs to render a correct output. To learn how to render nice graphics, visit the "Java 2D Web site".

本文详细的描述了 AWT 和 Swing 的绘制机制，目的是帮助开发人员编写出合理且高效的 GUI 绘制代码。本文描述了绘制机制的概要（即在什么时候和什么地方来做渲染），但并没有讲述如何使用图形 API 来渲染合理的输出。如果想学习如何渲染出出色的图形，请访问"Java 2D Web site"。

The main topics covered in this article are:

本文的主题包括：

- Evolution of Paint System

  绘制系统的发展

  - Painting in the AWT

Evolution of the Swing Paint System　Swing 绘制系统的发展

When the original AWT API was developed for JDK 1.0, only heavyweight components existed ("heavyweight"

means that the component has it's own opaque native window). This allowed the AWT to rely heavily on the paint subsystem in each native platform. This scheme took care of details such as damage detection, clip calculation, and z-ordering. With the introduction of lightweight components in JDK 1.1 (a "lightweight" component is one that reuses the native window of its closest heavyweight ancestor), the AWT needed to implement the paint processing for lightweight components in the shared Java code. Consequently, there are subtle differences in how painting works for heavyweight and lightweight components.

最初 JDK1.0 开发的 AWT API 中只有重量级组件（“重量级组件”是指组件有自己的不透明本地窗口）。这使得 AWT 严重依赖每个本地平台的子绘制系统。这个方案需要考虑很多细节，比如破坏保护，剪裁计算和 Z 轴次序等。在 JDK1.1 版本引入了轻量级组件（“轻量级”组件是指重用本地窗口最接近的重量级祖先），AWT 需要在公用 Java 代码中为轻量级组件实现绘制处理。因此，轻量级组件和重量级组件之间的绘制区别很小。

After JDK 1.1, when the Swing toolkit was released, it introduced its own spin on painting components. For the most part, the Swing painting mechanism resembles and relies on the AWT's. But it also introduces some differences in the mechanism, as well as new APIs that make it easier for applications to customize how painting works.

Swing 工具包在 JDK1.1 之后发布，它绘制组件时引入了自己的机制。大部分的 Swing 绘制机制都依赖和类似 AWT。但是它还引入了一些不同的机制，比如新的 API，它们可以使应用程序的绘制工作定制起来更简单。

Painting in AWT　　AWT 的绘制

To understand how AWT's painting API works, helps to know what triggers a paint operation in a windowing environment. In AWT, there are two kinds of painting operations: system-triggered painting, and application-triggered painting.

弄明白 AWT 的绘制 API 是如何工作可以帮助理解在窗口环境中是什么触发了绘制操作。在 AWT 中，有两种绘制方式：系统触发绘制与应用触发绘制。

System-triggered Painting　　系统触发绘制

In a system-triggered painting operation, the system requests a component to render its contents, usually for one of the following reasons:

在系统触发绘制操作中，系统要渲染组件的内容，一般有下面的某个原因：

- The component is first made visible on the screen.

  组件第一次在屏幕上显示。

- The component is resized.

  组件改变了大小。

- The component has damage that needs to be repaired. (For example, something that previously obscured the component has moved, and a previously obscured portion of the component has become exposed).

  组件破坏之后需要修复。（比如某些原先在组件上不明显的部分被移动了，组件的不明显的部分被显示出来了等）。

App-triggered Painting　　应用触发绘制

In an application-triggered painting operation, the component decides it needs to update its contents because its internal state has changed. (For example,. a button detects that a mouse button has been pressed and determines that it needs to paint a "depressed" button visual).

在应用触发绘制操作中，组件会根据需要更新它的内容，因为它的内部状态被改变了。（比如，一个按钮发现鼠标按下了，然后它需要绘制一个“按下的”按钮并显现出来）。

The Paint Method　　绘制方法

Regardless of how a paint request is triggered, the AWT uses a "callback" mechanism for painting, and this mechanism is the same for both heavyweight and lightweight components. This means that a program should place the component's rendering code inside a particular overridden method, and the toolkit will invoke this method when it's time to paint. The method to be overridden is in java.awt.Component:

不管绘制请求是怎么触发的，AWT 都使用“回调”机制来绘制，轻量级组件与重量级组件都是这样。所以程序应该把组件渲染代码放在一个特定的重置方法里，然后工具包将会在合适的时候调用这个方法绘制。这个方法在 java.awt.Component 中被重置：

```
public void paint(Graphics g)
```

When AWT invokes this method, the Graphics object parameter is pre-configured with the appropriate state for drawing on this particular component:

当 AWT 调用这个方法（Graphics 对象参数是先前设置并带有状态，用于在特定的组件上进行绘制）：

- The `Graphics` object's color is set to the component's `foreground` property.

  Graphics 对象的颜色设置成组件的前景色属性。

- The `Graphics` object's font is set to the component's `font` property.

  Graphics 对象的字体设置成组件的字体属性。

- The `Graphics` object's translation is set such that the coordinate (0,0) represents the upper left corner of the component.

  Graphics 对象转换坐标（0，0）为组件的左上角。

- The `Graphics` object's clip rectangle is set to the area of the component that is in need of repainting.

Graphics 对象的剪裁矩形设置为组件需要绘制的区域。

Programs must use this Graphics object (or one derived from it) to render output. They are free to change the state of the Graphics object as necessary.

程序必须使用这个 Graphics 对象（或来源于它）来渲染输出。如果有必要可以任意改变 Graphics 对象的状态。

Here is a simple example of a paint callback which renders a filled circle in the bounds of a component:

下面是一个绘制回调的简单例子，它在一个组件范围内渲染填充了一个圆：

```
public void paint(Graphics g) {
    // Dynamically calculate size information
    Dimension size = getSize();
    // diameter
    int d = Math.min(size.width, size.height);
    int x = (size.width - d)/2;
    int y = (size.height - d)/2;

    // draw circle (color already set to foreground)
    g.fillOval(x, y, d, d);
    g.setColor(Color.black);
    g.drawOval(x, y, d, d);
}
```

Developers who are new to AWT might want to take a peek at the PaintDemo example, which provides a runnable program example of how to use the paint callback in an AWT program.

AWT 开发新手可以看一看 PaintDemo（jdk 源代码所带）例子，它提供了一个如何在 AWT 中使用绘制回调的可运行的程序例子。

In general, programs should avoid placing rendering code at any point where it might be invoked outside the scope of the paint callback. Why? Because such code may be invoked at times when it is not appropriate to paint -- for instance, before the component is visible or has access to a valid Graphics object. It is not recommended that programs invoke paint() directly.

一般来说，程序应该避免将渲染代码放在超出绘制回调范围的地方，因为这些代码可能在不应当绘制的时候被调用－举个例子，在组件显现出来之前或访问一个有效的 Graphics 对象，不建议直接调用 paint()。

To enable app-triggered painting, the AWT provides the following java.awt.Component methods to allow programs to asynchronously request a paint operation:

为了实现应用触发绘制，AWT 提供了下面的 java.awt.Component 方法来允许程序异步请求绘制操作。

```
public void repaint()
public void repaint(long tm)
public void repaint(int x, int y, int width, int height)
public void repaint(long tm, int x, int y, int width, int height)
```

The following code shows a simple example of a mouse listener that uses repaint() to trigger updates on a theoretical button component when the mouse is pressed and released:

下面的代码演示了一个简单的鼠标监听器，当鼠标按下和松开的时候使用 repaint()来触发按钮组件更新。

```
MouseListener l = new MouseAdapter() {
    public void mousePressed(MouseEvent e) {
        MyButton b = (MyButton)e.getSource();
        b.setSelected(true);
        b.repaint();
    }

    public void mouseReleased(MouseEvent e) {
        MyButton b = (MyButton)e.getSource();
        b.setSelected(false);
        b.repaint();
    }
};
```

Components that render complex output should invoke repaint() with arguments defining only the region that requires updating. A common mistake is to always invoke the no-arg version, which causes a repaint of the entire component, often resulting in unnecessary paint processing.

渲染复杂输出的组件应该通过调用带需要更新区域的参数的 repaint()。一个普遍的错误是调用无参数的版本，它会让整个组件被重绘，通常会导致做不必要的绘制处理。

paint() vs. update()　　paint()方法与 update()方法

Why do we make a distinction between "system-triggered" and. "app-triggered" painting? Because AWT treats each of these cases slightly differently for heavyweight components (the lightweight case will be discussed later), which is unfortunately a source of great confusion.

我们之所以要区分“系统触发”与“应用触发”绘制，是因为对于 AWT 的重量级组件来说这两种情况是有区别的（轻量级组件稍后再作讨论），不幸的是，这是很大的混乱之源。

For heavyweight components, these two types of painting happen in the two distinct ways, depending on whether a painting operation is system-triggered or app-triggered.

对重量级组件来说，这两种类型的绘制发生在两种不同的场合，即绘制操作是由系统触发的还是由应用触发的。

System-triggered painting　　系统触发绘制

This is how a system-triggered painting operation takes place:

下面是系统触发绘制操作如何发生的：

1. The AWT determines that either part or all of a component needs to be painted

   AWT 决定组件是部分需要绘制还是全部需要绘制。

2. The AWT causes the event dispatching thread to invoke `paint()` on the component.

   AWT 让事件派发线程调用组件的 `paint()`。

App-triggered painting　　应用触发绘制

An app-triggered painting operation takes place as follows:

下面是应用触发绘制操作如何发生的：

1. The program determines that either part or all of a component needs to be repainted in response to some internal state change.

   程序根据内部状态的变化决定组件是部分需要绘制还是全部需要绘制。

2. The program invokes `repaint()` on the component, which registers an asynchronous request to the AWT that this component needs to be repainted.

   程序调用组件的 `repaint()` 方法，注册一个重绘异步请求到 AWT。

   1. The AWT causes the event dispatching thread to invoke `update()` on the component.

      AWT 使得事件派发线程调用组件的 `update()` 方法。

NOTE: *If multiple calls to repaint() occur on a component before the initial repaint request is processed, the multiple requests may be collapsed into a single call to update(). The algorithm for determining when multiple requests should be collapsed is implementation-dependent. If multiple requests are collapsed, the resulting update rectangle will be equal to the union of the rectangles contained in the collapsed requests.*

注意：*如果在重绘请求初始化处理开始之前，组件的 repaint()被多次被调用，多个调用请求可能会被组合成一个 update()调用，是否决定将多个请求合成取决于具体实现，如果多个请求组合，矩形更新的结果等同于所有组合的矩形之和。*

3. If the component did not override `update()`, the default implementation of `update()` clears the component's background (if it's not a lightweight component) and simply calls `paint()`.

   如果组件没有重置 update()，update()的缺省实现会清空组件的背景（如果不是轻量级组件），然后简单的调用一下 paint()。

Since by default, the final result is the same (paint() is called), many people don't understand the purpose of having a separate update() method at all. While it's true that the default implementation of update() turns around and calls paint(), this update "hook" enables a program to handle the app-triggered painting case differently, if desired. A program must assume that a call to paint() implies that the area defined by the graphic's clip rectangle is "damaged" and must be completely repainted, however a call to update() does not imply this, which enables a program to do incremental painting.

通过缺省处理后的结果一样（paint()被调用），很多人都不明白为什么会有一个单独的 update()方法，这就是答案。update()缺省实现回过头来调用 paint()，如果需要，这个更新“钩子”可以使程序处理应用触发绘制方案有所不同。如果图形的剪裁矩形定义的区域被“破坏”，需要进行完全绘制，程序必须调用 paint()，而不必调用 update()，这样会让程序做多余的绘制。

Incremental painting is useful if a program wishes to layer additional rendering on top of the existing bits of that component. The “UpdateDemo example” demonstrates a program which benefits from using update() to do incremental painting.

如果程序希望在组件已有的图形顶部再做渲染，那么进一步的绘制是有用的。UpdateDemo example 示范了程序使用 update()来做进一步的绘制的好处。

In truth, the majority of GUI components do not need to do incremental drawing, so most programs can ignore the update() method and simply override paint() to render the component in it's current state. This means that both system-triggered and app-triggered rendering will essentially be equivalent for most component implementations.

事实上，大部分 GUI 组件都不需要做进一步的绘制，所以大多程序可以忽略 update()方法，只需简单的重置 paint()来在组件的当前状态上进行渲染，这意味着系统触发渲染和应用触发渲染对大多组件的实现来说在本质上都是一样的。

Painting & Lightweight Components 轻量级组件绘制

From an application developer's perspective, the paint API is basically the same for lightweights as it is for heavyweights (that is, you just override paint() and invoke repaint() to trigger updates). However, since AWT's lightweight component framework is written entirely in common Java code, there are some subtle differences in the way the mechanism is implemented for lightweights.

从应用开发人员的角度上来看，绘制 API 对轻量级组件和重量级组件来说基本上是一样的（也就是说，你可以重置 paint()，然后调用 repaint()来触发更新），然而，由于 AWT 的轻量级组件框架全部写在 Java 公用代码里，轻量级组件的实现机制与重量级组件还是有些微小的不同之处的。

How Lightweights Get Painted 轻量级组件是如何绘制的

For a lightweight to exist, it needs a heavyweight somewhere up the containment hierarchy in order to have a place

to paint. When this heavyweight ancestor is told to paint its window, it must translate that paint call to paint calls on all of its lightweight descendents. This is handled by java.awt.Container's paint() method , which calls paint() on any of its visible, lightweight children which intersect with the rectangle to be painted. So it's critical for all Container subclasses (lightweight or heavyweight) that override paint() to do the following:

轻量级组件需要重量级组件建立包含层次结构来让其有一个地方进行绘制。当这个重量级组件祖先被要求绘制它的窗口，它必须将绘制调用转换为对它所有的派生的轻量级组件的绘制调用。这通过 java.awt.Container's paint()处理，与其矩形有交错的轻量级组件子类都将调用 paint()进行绘制。所以不建议对所有的容器子类（轻量级组件或重量级组件）像下面一样重置 paint()：

```
public class MyContainer extends Container {
    public void paint(Graphics g) {
        // paint my contents first...
        // then, make sure lightweight children paint
        super.paint(g);
    }
}
```

If the call to super.paint() is missing, then the container's lightweight descendents won't show up (a very common problem when JDK 1.1 first introduced lightweights).

如果 super.paint()调用丢失，容器的派生轻组件就不会显示（当 JDK1.1 第一次引入轻量级组件的时候这是一个非常普遍的问题）。

It's worth noting that the default implementation of Container.update() does not use recursion to invoke update() or paint() on lightweight descendents. This means that any heavyweight Container subclass that uses update() to do incremental painting must ensure that lightweight descendents are recursively repainted if necessary. Fortunately, few heavyweight container components need incremental painting, so this issue doesn't affect most programs.

缺省的 Container.update()实现如果不使用递归调用派生轻量级组件上 update()或 paint()，那它不值一提。所以如果必要，任何 Container 组件子类使用 update()做进一步的绘制必须保证派生轻组件被递归调用。幸运的是，一些重量级容器组件都需要进一步绘制，所以这个问题不会影响大多数程序。

Lightweights & System-triggered Painting　　轻量级组件的系统触发绘制

The lightweight framework code that implements the windowing behaviors (showing, hiding, moving, resizing, etc.) for lightweight components is written entirely in Java. Often, within the Java implementation of these functions, the AWT must explicitly tell various lightweight components to paint (essentially system-triggered painting, even though it's no longer originating from the native system). However, the lightweight framework uses repaint() to tell components to paint, which we previously explained results in a call to update() instead of a direct call to paint(). Therefore, for lightweights, system-triggered painting can follow two paths:

轻量级组件实现窗口行为（显示，隐藏，移动，改变大小等）的轻量级框架代码全部都是用 Java 写的。经常，这些功能的 Java 实现中，AWT 必须清楚的告诉各种轻量级组件如何绘制（本质上是系统触发绘制，虽然它不再是来自本地 系

统）。然而，轻量级框架使用 repaint()告诉组件如何绘制，这是我们上文所提到的直接调用 update()来代替 paint()。所以，对于轻量级组件，系统触发绘制遵循下面两种途径：

- The system-triggered paint request originates from the native system (i.e. the lightweight's heavyweight ancestor is first shown), which results in a direct call to `paint()`.

  系统触发绘制请求来源于本地系统（比如：轻量级组件的祖先第一次显示），它使得 paint()被直接调用。

- The system-triggered paint request originates from the lightweight framework (i.e., the lightweight is resized), which results in a call to `update()`, which by default is forwarded to `paint()` .

  系统触发绘制请求来源于轻量级框架，它会调用 update()（比如，轻量级组件改变了大小），update()缺省情况下会转向 paint()。

In a nutshell, this means that for lightweight components there is no real distinction between update() and paint(), which further implies that the incremental painting technique should not be used for lightweight components.

总的来说，对于轻量级组件 update()和 paint()之间没有真正的区别，从长远来说进一步的绘制技术不应该用于轻量级 组件。

Lightweights and Transparency　轻量级组件和透明性

Since lightweight components "borrow" the screen real estate of a heavyweight ancestor, they support the feature of transparency. This works because lightweight components are painted from back to front and therefore if a lightweight component leaves some or all of its associated bits unpainted, the underlying component will "show through." This is also the reason that the default implementation of update() will not clear the background if the component is lightweight.

从轻量级组件“借用”重量级祖先的屏幕状态之后，它们支持透明度特性。这是可行的，因为轻量级组件是从后向前 绘制的，然而如果一个轻量级组件留下部分或所有的相关色位没有绘制，下面的组件将会“穿过显示”，这也是轻量级组件中缺省的 update()实现不会清除背景的原因。

The LightweightDemo sample program demonstrates the transparency feature of lightweight components.

LightweightDemo（JDK 中的例子）程序展示了轻量级组件的透明度特性。

"Smart" Painting　“灵巧的”绘制

While the AWT attempts to make the process of rendering components as efficient as possible, a component's paint() implementation itself can have a significant impact on overall performance. Two key areas that can affect this process are:

当 AWT 尝试让渲染组件的过程变的尽可能的有效率，组件的 paint()实现能对全部性能产生重大的影响。影响这个处理的两个关键的区域是：

- Using the clip region to narrow the scope of what is rendered.

  使用剪裁区域来使渲染的范围变小。

- Using internal knowledge of the layout to narrow the scope of what children are painted (lightweights only).

  使用布局的内部知识来使子组件的绘制区域变小（只适用于轻量级组件）。

If your component is simple -- for example, if it's a pushbutton -- then it's not worth the effort to factor the rendering in order to only paint the portion that intersects the clip rectangle; it's preferable to just paint the entire component and let the graphics clip appropriately. However, if you've created a component that renders complex output, like a text component, then it's critical that your code use the clip information to narrow the amount of rendering.

如果你的组件很简单 -- 比如一个按钮 – 那么不必只对与剪裁矩形的交错部分做渲染;更好的做法是绘制整个组件，然后让图形做适当的剪裁。如果你需要创建一个渲染复杂输出的组件，如文本组件，不建议在你的代码中使用剪裁信息 来减少渲染的工作量。

Further, if you're writing a complex lightweight container that houses numerous components, where the component and/or its layout manager has information about the layout, then it's worth using that layout knowledge to be smarter about determining which of the children must be painted. The default implementation of Container.paint() simply looks through the children sequentially and tests for visibility and intersection -- an operation that may be unnecessarily inefficient with certain layouts. For example, if a container layed out the components in a 100x100 grid, then that grid information could be used to determine more quickly which of those 10,000 components intersect the clip rectangle and actually need to be painted.

更进一步的来说，如果你写一个有大量组件的复杂的轻量级容器，组件和它的布局管理器有布局的信息，那么值得使 用布局知识来灵活决定哪个子组件需要绘制。 Container.paint()的缺省实现简单的穿过了子组件并检验可见性与交互性 -- 某些布局操作可能是不必要的低效率操作。比如，如果容器在 100x100 的格子中布局组件，格子信息会被用于更快确 定这 10,000 个与剪裁矩形交互的组件中哪些确实需要绘制。

AWT Painting Guidelines AWT 绘制指南

The AWT provides a simple callback API for painting components. When you use it, the following guidelines apply:

AWT 为组件绘制提供了简单的回调 API。当你使用的时候，下面的指南比较实用：

1. For most programs, all client paint code should be placed within the scope of the component's `paint()` method.

   对于大多程序，所有客户端绘制代码应该放在组件的 `paint()`方法内部。

2. Programs may trigger a future call to `paint()` by invoking `repaint()`, but shouldn't call `paint()` directly.

   程序可能将来会通过 `repaint()`触发 `paint()`，而不应该直接调用 `paint()`。

3. On components with complex output, repaint() should be invoked with arguments which define only the rectangle that needs updating, rather than the no-arg version, which causes the entire component to be repainted.

在有复杂输出的组件，repaint()应该带参数调用，因为它只定义了需要更新的矩形。如果使用不带参数的版本，它将会导致整个组件都被绘制。

4. Since a call to repaint() results first in a call to update(), which is forwarded to paint() by default, heavyweight components may override update() to do incremental drawing if desired (lightweights do not support incremental drawing)

在 repaint()被调用时会首先调用 update()，缺省情况下 update()会转向 paint()，如果需要，重量级组件可以重置 update()来做进一步的绘画（轻量级组件不支持进一步的绘画）。

5. Extensions of java.awt.Container which override paint() should always invoke super.paint() to ensure children are painted.

如果 java.awt.Container 的扩展重置了 paint()方法，在该重置的方法中应该调用 super.paint()来保证子组件都被绘制。

6. Components which render complex output should make smart use of the clip rectangle to narrow the drawing operations to those which intersects with the clip area.

渲染复杂输出的组件应该灵活的应用剪裁矩形来减少与剪裁区域交互的绘画操作。

Painting in Swing　　Swing 中的绘制

Swing starts with AWT's basic painting model and extends it further in order to maximize performance and improve extensibility. Like AWT, Swing supports the paint callback and the use of repaint() to trigger updates. Additionally, Swing provides built-in support for double-buffering as well as changes to support Swing's additional structure (like borders and the UI delegate). And finally, Swing provides the RepaintManager API for those programs who want to customize the paint mechanism further.

Swing 从 AWT 的基本绘制模型开始，然后为了使性能最大化和增强可扩展性，它做了很多延伸。和 AWT 一样，Swing 支持绘制回调与使用 repaint()来触发更新。Swing 提供了对双缓冲绘制的内置支持，同时还对 Swing 附加结构的改变给予支持（比如边框和 UI 代理）。最后，Swing 为那些希望定制绘制机制的程序提供了 RepaintManager API 来给予更多支持。

Double Buffering Support　　双缓冲支持

One of the most notable features of Swing is that it builds support for double-buffering right into the toolkit. It does the by providing a "doubleBuffered" property on javax.swing.JComponent:

一个 Swing 非常值得注意的特性是它在工具中建立了对双缓冲的完全支持。它通过在 javax.swing.JComponent 上提供“双缓冲”属性来做到这点的。

```
public boolean isDoubleBuffered()
public void setDoubleBuffered(boolean o)
```

Swing's double buffer mechanism uses a single offscreen buffer per containment hierarchy (usually per top-level window) where double-buffering has been enabled. And although this property can be set on a per-component basis, the result of setting it on a particular container will have the effect of causing all lightweight components underneath that container to be rendered into the offscreen buffer, regardless of their individual "doubleBuffered" property values.

Swing 的双缓冲机制在双缓冲可用的每个包含层次结构都使用了一个画面以外的缓冲（通常每个顶层窗口），虽然这个属性是建立在每个组件的基础上，在一个特定的容器上设置双缓冲将会使该容器上的所有轻量级组件都被在画面之外 渲染，不管它们单个的“双缓冲”属性值如何。

By default, this property is set to true for all Swing components. But the setting that really matters is on JRootPane, because that setting effectively turns on double-buffering for everything underneath the top-level Swing component. For the most part, Swing programs don't need to do anything special to deal with double-buffering, except to decide whether it should be on or off (and for smooth GUI rendering, you'll want it on!). Swing ensures that the appropriate type of Graphics object (offscreen image Graphics for double-buffering, regular Graphics otherwise) is passed to the component's paint callback, so all the component needs to do is draw with it. This mechanism is explained in greater detail later in this article, in the section on Paint Processing.

缺省情况下，所有的 Swing 组件的这个属性都被设置为 true，但是真正起作用的是在 JRootPane 中的设置，因为它为顶层 Swing 组件之下所有的组件都有效启动了双缓冲。在大多数情况之下，Swing 程序不需要为双缓冲做任何特别的处理，除了决定它是否需要开启或关闭（对于滑动 GUI 渲染，你将需要打开它）。Swing 保证了图形对象的准确类型（后台双缓冲图形，矩形图形等）被传送给组件绘制回调，所以所有组件需要做的是使用该图形对象来绘图。这个机制迟 些将会在本文的绘制处理小节中做非常详细的描述。

Additional Paint Properties　　附加的绘制属性

Swing introduces a couple of additional properties on JComponent in order to improve the efficiency of the internal paint algorithms. These properties were introduced in order to deal with the following two issues, which can make painting lightweight components an expensive operation:

Swing 在 JComponent 上引进一些附加属性来改进内部绘制算法的性能。这些属性被引进来处理下面两个问题，这两 个问题使轻量级组件的绘制操作花销特别大：

- *Transparency*: If a lightweight component is painted, it's possible that the component will not paint all of its associated bits if partially or totally transparent; this means that whenever it is repainted, whatever lies underneath it must be repainted first. This requires the system to walk up the containment hierarchy to find the first underlying heavyweight ancestor from which to begin the back-to-front paint operation.

  透明度处理：如果一个轻量级组件被绘制，这意味着不管什么时候和什么前提下，如果部分或全部都是透明的，组件可能不会绘制所有与它相关的色位，这需要系统沿着包含层次结构来找到第一个重量级组件祖先，然后开始由后向前进行绘制。

- *Overlapping components:* If a lightweight component is painted, its possible that some other lightweight component partially overlaps it; this means that whenever the original lightweight component is painted, any

components which overlap the original component (where the clip rectangle intersects with the overlapping area) the overlapping component must also be partially repainted. This requires the system to traverse much of the containment hierarchy, checking for overlapping components on each paint operation.

组件重叠：如果一个轻量级组件被绘制，可能会有一些其它轻量级组件与其有部分重叠，不论什么时候原轻量级组件被绘制，任何与其重叠的组件（在剪裁矩形交错的重叠区域）也必须部分绘制，这需要系统在做每个绘制操作时都穿过很多包含层次结构，检查组件的重叠。

Opacity　不透明度

To improve performance in the common case of opaque components, Swing adds a read-write opaque property to javax.swing.JComponent:

为了改进不透明组件共用情况的性能， Swing 给 javax.swing.JComponent 添加了 opaque 读－写属性：

```
public boolean isOpaque()
public void setOpaque(boolean o)
```

The settings are:

设置说明：

- `true`: The component agrees to paint all of the bits contained within its rectangular bounds.

  `true`：组件同意绘制包含在其矩形范围的所有的色位

- `false`: The component makes no guarantees about painting all the bits within its rectangular bounds.

  `false`：组件不保证绘制包含在其矩形范围的所有的色位

The opaque property allows Swing's paint system to detect whether a repaint request on a particular component will require the additional repainting of underlying ancestors or not. The default value of the opaque property for each standard Swing component is set by the current look and feel UI object. The value is true for most components.

opaque 属性让 Swing 绘制系统察觉到某一组件上的绘制请求是否会要求在底层祖先上做进一步的绘制。每个标准 Swing 组件的 opaque 属性缺省值由当前感官 UI 对象来设置。大多数组件都是 true。

One of the most common mistakes component implementations make is that they allow the opaque property to default to true, yet they do not completely render the area defined by their bounds, the result is occasional screen garbage in the unrendered areas. When a component is designed, careful thought should be given to its handling of the opaque property, both to ensure that transparency is used wisely, since it costs more at paint time, and that the contract with the paint system is honored.

一个非常普遍的错误是将组件实现的 opaque 属性缺省设为 true，这样它们不会完全绘制它们定义范围的所有区域，结果是因有些区域没有渲染有时会产生屏幕垃圾。经过认真思考后的组件的设计应该对 opaque 属性做出处理，还要保证对透明度属性有合理运用，这样的绘制系统才更优秀。

The meaning of the opaque property is often misunderstood. Sometimes it is taken to mean, "Make the component's background transparent." However, this is not Swing's strict interpretation of opacity. Some components, such as a pushbutton, may set the opaque property to false in order to give the component a non-rectangular shape, or to leave room around the component for transient visuals, such as a focus indicator. In these cases, the component is not opaque, but a major portion of its background is still filled in.

Opaque 属性的意义常常被误解，有时它被理解为“使组件的背景透明”，这不是对 Swing 的不透明度的严格解释。某些组件，比如一个按钮，也许会设置不透明度属性为 false 来给组件一个非矩形形状，或者在组件的周围留下暂时的外观空间，比如焦点指示器，在这些情况下组件是透明的，但它的背景的主要部分总被填充。

As defined previously, the opaque property is primarily a contract with the repaint system. If a component also uses the opaque property to define how transparency is applied to a component's visuals, then this use of the property should be documented. (It may be preferable for some components to define additional properties to control the visual aspects of how transparency is applied. For example, javax.swing.AbstractButton provides the ContentAreaFilled property for this purpose.)

前文定义过，不透明度属性是绘制系统的一个基本约定。如果一个组件使用不透明度属性来定义组件的透明度如何依赖其外观，那么这个属性的使用必须做文档说明。（在透明度是如何依赖的等透明度视觉方面，可能为某些组件定义其它属性来控制会更好一些，比如 javax.swing.AbstractButton 对此提供了 ContentAreaFilled 属性）。

Another issue worth noting is how opacity relates to a Swing component's border property. The area rendered by a Border object set on a component is still considered to be part of that component's geometry. This means that if a component is opaque, it is still responsible for filling the area occupied by the border. (The border then just layers its rendering on top of the opaque component).

还有值得一提的是不透明度属性与 Swing 组件边框的属性的联系。在组件上设置的边框对象渲染的区域也被认为是组件几何学的一部分。这意味着如果组件是不透明的，它还负责填充边框区域（边框在不透明组件的顶层做渲染）。

If you want a component to allow the underlying component to show through its border area -- that is, if the border supports transparency via isBorderOpaque() returning false -- then the component must define itself to be non-opaque and ensure it leaves the border area unpainted.

如果你希望一个组件允许底层组件穿过它的边框区域来显示 – 也就是说，如果边框支持通过方法 isBorderOpaque()返回 false 设置为透明 – 这样组件必须定义自身为透明来使边框区域不进行绘制。

"Optimized" Drawing “最优化”绘制

The overlapping component issue is more tricky. Even if none of a component's immediate siblings overlaps the component, it's always possible that a non-ancestor relative (such as a "cousin" or "aunt") could overlap it. In such a case the repainting of a single component within a complex hierarchy could require a lot of treewalking to ensure 'correct' painting occurs. To reduce unnecessary traversal, Swing adds a read-only isOptimizedDrawingEnabled property to javax.swing.JComponent:

组件重叠问题更敏感，即使没有其它兄弟级组件与某个组件重叠，它也可能会有非祖宗级的亲属关系的（比如“堂兄”或“阿姨”）组件与其重叠。这样在复杂的层次中绘制某一组件需要做很多向上追溯的工作来保证绘制的正确性。为了减少不必要的来回移动，Swing 为 javax.swing.JComponent 增加了一个只读的属性 isOptimizedDrawingEnabled：

```
public boolean isOptimizedDrawingEnabled()
```

The settings are:

设置为：

`true`: The component indicates that none of its immediate children overlap.

`true`:组件指示它所有的子组件不会重叠

`false`: The component makes no guarantees about whether or not its immediate children overlap

`false`:组件不保证它的子组件是否被重叠

By checking the isOptimizedDrawingEnabled property, Swing can quickly narrow its search for overlapping components at repaint time.

通过检查 isOptimizedDrawingEnabled 属性，Swing 在绘制的时候可以快速缩小重叠组件的搜索范围。

Since the `isOptimizedDrawingEnabled` property is read-only, so the only way components can change the default value is to subclass and override this method to return the desired value. All standard Swing components return `true` for this property, except for `JLayeredPane`, `JDesktopPane`, and `JViewPort`.

由于 isOptimizedDrawingEnabled 属性是只读的， 所以组件改变缺省值的唯一方法在子类覆盖这个方法并返回需要的值。所有标准 Swing 组件的这个属性都返回 true，除了 JLayeredPane, JDesktopPane, 和 JViewPort

The Paint Methods 绘制方法

The rules that apply to AWT's lightweight components also apply to Swing components -- for instance, paint() gets called when it's time to render -- except that Swing further factors the paint() call into three separate methods, which are invoked in the following order:

AWT 轻量级组件的规则同样使用于 Swing 组件 – 比如当绘制的时候 paint()会被调用 – 除了 Swing 进一步将 paint()调用分成三个方法，调用的次序如下：

```
protected void paintComponent(Graphics g)
protected void paintBorder(Graphics g)
protected void paintChildren(Graphics g)
```

Swing programs should override paintComponent() instead of overriding paint(). Although the API allows it, there is generally no reason to override paintBorder() or paintComponents() (and if you do, make sure you know what you're doing!). This factoring makes it easier for programs to override only the portion of the painting which they need to extend. For example, this solves the AWT problem mentioned previously where a failure to invoke super.paint() prevented any lightweight children from appearing.

Swing 程序应该覆盖 paintComponent()来代替 paint()方法，虽然 API 允许这么做，但一般没有理由来覆盖 paintBorder()或 paintComponents()（如果你这么做了，你得要明白你在做什么）。这个原因使得程序只需简单的覆盖它们需要扩展的绘制部分。比如，这解决了前面提到的 AWT 中调用 super.paint()失败导致所有轻量级子组件无法显示的问题。

The SwingPaintDemo sample program demonstrates the simple use of Swing's paintComponent() callback.

在 ~~SwingPaintDemo~~ 例子程序中简单示范了 Swing 的 paintComponent()方法的回调。

Painting and the UI Delegate绘制和 UI 代理

Most of the standard Swing components have their look and feel implemented by separate look-and-feel objects (called "UI delegates") for Swing's Pluggable look and feel feature. This means that most or all of the painting for the standard components is delegated to the UI delegate and this occurs in the following way:

大部分 swing 标准组件都通过独立的感官对象（称谓“UI 代理”）为 Swing 的可拔插特性做了感官实现。这意味着标准组件的所有或大部分绘制都按下面的方式由 UI 代理来代理：

1. `paint()` invokes `paintComponent()`.

   `paint()`调用 `paintComponent()`

2. If the `ui` property is non-null, `paintComponent()` invokes `ui.update()`.

   如果 ui 属性不为空，`paintComponent()`调用 `ui.update()`

3. If the component's `opaque` property is true, `ui.udpate()` fills the component's background with the background color and invokes `ui.paint()`.

   如果组件的 `opaque` 属性为 `true`，`ui.udpate()`使用背景填充组件的背景并调用 `ui.paint()`方法

4. `ui.paint()` renders the content of the component.

   `ui.paint()` 方法渲染组件的内容。

This means that subclasses of Swing components which have a UI delegate (vs. direct subclasses of JComponent), should invoke super.paintComponent() within their paintComponent override:

这意味着有 UI 代理的 Swing 组件的子类（对 JComponent 的直接子类），应该在它们重写的 paintComponent 中调用 super.paintComponent()：

```
public class MyPanel extends JPanel {
    protected void paintComponent(Graphics g) {
        // Let UI delegate paint first
        // (including background filling, if I'm opaque)
        super.paintComponent(g);
        // paint my contents next....
    }
}
```

If for some reason the component extension does not want to allow the UI delegate to paint (if, for example, it is completely replacing the component's visuals), it may skip calling super.paintComponent(), but it must be responsible for filling in its own background if the opaque property is true, as discussed in the section on the opaque property.

如果由于某种原因组件的扩展不希望让 UI 代理绘制（比如， 完全取代组件的视觉），可以跳过 super.paintComponent() 的调用,但是如果 opaque 属性为 true，它必须负责像上文对 opaque 属性的讨论中提到的那样填充自己的背景。

Paint Processing　绘制处理

Swing processes "repaint" requests in a slightly different way from the AWT, although the final result for the application programmer is essentially the same -- paint() is invoked. Swing does this to support its RepaintManager API (discussed later), as well as to improve paint performance. In Swing, painting can follow two paths, as described below:

Swing 处理“repaint”与 AWT 有一些细微的差别，尽管对应用程序员来说最后的结果本质上都一样 – 调用 paint()方法。Swing 这样做的目的是支持它的 RepaintManager API（迟些再做讨论），也为了改进绘制的性能。在 Swing 中，绘制有如下两种途经：

(A) The paint request originates on the first heavyweight ancestor (usually JFrame, JDialog, JWindow, or JApplet):

绘制请求起源于第一个重量级祖先（一般是 JFrame, JDialog, JWindow, 或 JApplet）：

1. the event dispatching thread invokes paint() on that ancestor

事件派发线程调用祖先的 paint()方法

2. The default implementation of Container.paint() recursively calls paint() on any lightweight descendents

Container.paint()方法的缺省实现递归调用所有派生的轻量级组件的 paint()方法

3. When the first Swing component is reached, the default implementation of JComponent.paint() does the following:

当到达第一个 Swing 组件的时候，JComponent.paint()的缺省实现会做如下事情：

1. if the component's doubleBuffered property is true and double-buffering is enabled on the component's RepaintManager, will convert the Graphics object to an appropriate offscreen graphics.

   如果组件的 doubleBuffered 属性为 true，并且组件的 RepaintManager 启用了双缓冲，将会把 Graphics 对象转换为相应的后台图形。

2. invokes paintComponent() (passing in offscreen graphics if doubled-buffered)

   调用 paintComponent()方法（如果有双缓冲则通过后台图形来传递）

3. invokes paintBorder() (passing in offscreen graphics if doubled-buffered)

   调用 paintBorder()方法（如果有双缓冲则通过后台图形来传递）

4. invokes paintChildren() (passing in offscreen graphics if doubled-buffered), which uses the clip and the opaque and optimizedDrawingEnabled properties to determine exactly which descendents to recursively invoke paint() on.

   调用 paintChildren()（如果进行了双缓冲则通过后台图形来传递），使用剪裁， opaque 和 optimizedDrawingEnabled 属性来严格限定该调用哪些派生类的 paint()

5. if the component's doubleBuffered property is true and double-buffering is enabled on the component's RepaintManager, copies the offscreen image to the component using the original on-screen Graphics object.

   如果组件的 doubleBuffered 属性为 true，而且组件的 RepaintManager 启用了双缓冲，根据组件的前台图形对象复制后台图形

   Note: the JComponent.paint() steps #1 and #5 are skipped in the recursive calls to paint() (from paintChildren(), described in step#4) because all the lightweight components within a Swing window hierarchy will share the same offscreen image for double-buffering.

   说明：JComponent.paint()的第 1 步到第 5 步跳过了对 paint()的递归调用（从 paintChildren()，在第 4 步中描述）因为在 Swing 窗口层次关系中所有的轻量级组件都将共享同一个双缓冲的后台图形。

(B) The paint request originates from a call to repaint() on an extension of javax.swing.JComponent:

请求来源于 javax.swing.JComponent 派生类中对 repaint()的调用：

1. JComponent.repaint() registers an asynchronous repaint request to the component's RepaintManager, which uses invokeLater() to queue a Runnable to later process the request on the event dispatching thread.

   JComponent.repaint()在组件的 RepaintManager 中注册一个异步的绘制请求，使用了 invokeLater()把一个 Runnable 放到事件派发线程队列中稍后处理

2. The runnable executes on the event dispatching thread and causes the component's RepaintManager to invoke paintImmediately() on the component, which does the following:

事件派发线程执行 Runnable 来使组件的 RepaintManager 调用组件上的 paintImmediately()，它所作如下：

1. uses the clip rectangle and the opaque and optimizedDrawingEnabled properties to determine the 'root' component from which the paint operation must begin (to deal with transparency and potentially overlapping components).

   使用剪裁矩形，opaque 和 optimizedDrawingEnabled 属性来决定从哪里是开始绘制操作的“根”组件（处理透明性和潜在的重叠组件）

2. if the root component's doubleBuffered property is true, and double-buffering is enabled on the root's RepaintManager, will convert the Graphics object to an appropriate offscreen graphics.

   如果根组件的 doubleBuffered 属性为 true，且 RepaintManager 启用了双缓冲，将会把图形对象转换成合适的后台图形

3. invokes paint() on the root component (which executes (A)'s JComponent.paint() steps #2-4 above), causing everything under the root which intersects with the clip rectangle to be painted.

   调用根组件的 paint()方法（执行上面(A)部分中的 JComponent.paint()的第 2－4 步），使得所有与剪裁矩形交互的组件都被绘制

4. if the root component's doubleBuffered property is true and double-buffering is enabled on the root's RepaintManager, copies the offscreen image to the component using the original on-screen Graphics object.

   如果根组件的 doubleBuffered 属性为 true，且 RepaintManager 启用了双缓冲，将会使用原始前台图形对象把后台图形复制到组件上

   NOTE: if multiple calls to repaint() occur on a component or any of its Swing ancestors before the repaint request is processed, those multiple requests may be collapsed into a single call back to paintImmediately() on the topmost Swing component on which repaint() was invoked. For example, if a JTabbedPane contains a JTable and both issue calls to repaint() before any pending repaint requests on that hierarchy are processed, the result will be a single call to paintImmediately() on the JTabbedPane, which will cause paint() to be executed on both components.

   说明：如果组件或任何它的 swing 祖先对 repaint()在绘制请求处理之前进行多个调用，这些多个请求可能会在调用了 repaint()方法的最顶层的 Swing 组件中组合成一个 paintImmediately()回调。

This means that for Swing components, update() is never invoked.

这就是说对于 Swing 组件来说，update()方法从不被调用。

Although repaint() results in a call to paintImmediately(), it is not considered the paint "callback", and client paint code should not be placed inside of a paintImmediately(). In fact, there is no common reason to override paintImmediately() at all.

虽然 repaint()引起了对 paintImmediately()方法的调用，它并不被认为是绘制“回调”，而且客户端绘制代码也不应该放在 paintImmediately()方法里面。事实上，一般没有什么理由覆盖 paintImmediately()。

Synchronous Painting 同步绘制

As described in the previous section, paintImmediately() acts as the entry point for telling a single Swing component to paint itself, making sure that all the required painting occurs appropriately. This method may also be used for making synchronous paint requests, as its name implies, which is sometimes required by components which need to ensure their visual appearance 'keeps up' in real time with their internal state (e.g. this is true for the JScrollPane during a scroll operation).

上文提到，paintImmediately()方法表现的像一个入口点来告诉 Swing 绘制自身，保证所有的绘制请求正确发生。该方法还被用于做同步的绘制请求，像它的名字那样，有时候组件要求保证它们的可视化表现与它们的内部状态“保持”实时性。

Programs should not invoke this method directly unless there is a valid need for real-time painting. This is because the asynchronous repaint() will cause multiple overlapping requests to be collapsed efficiently, whereas direct calls to paintImmediately() will not. Additionally, the rule for invoking this method is that it must be invoked from the event dispatching thread; it's not an api designed for multi-threading your paint code!. For more details on Swing's single-threaded model, see the archived article "Threads and Swing."

程序不应该直接调用这个方法，除非有实时绘制的正当请求。这是因为异步 repaint()可能会导致多个重叠请求失效，而直接调用 paintImmediately()方法就不会这样。进一步来说，调用该方法的规则是它必须从事件派发线程中触发；它不是针对你的多线程绘制代码设计的 API。如果想知道更多关于 Swing 的单线程模型，请看文档"Threads and Swing."

The RepaintManager 绘制管理器

The purpose of Swing's RepaintManager class is to maximize the efficiency of repaint processing on a Swing containment hierarchy, and also to implement Swing's 'revalidation' mechanism (the latter will be a subject for a separate article). It implements the repaint mechanism by intercepting all repaint requests on Swing components (so they are no longer processed by the AWT) and maintaining its own state on what needs to be updated (known as "dirty regions"). Finally, it uses invokeLater() to process the pending requests on the event dispatching thread, as described in the section on "Repaint Processing" (option B).

Swing 中的 RepaintManager 类的作用是最大化 Swing 层次结构中绘制处理的效率，还实现 Swing 中的“重新生效”机制（这是在迟些时候发表的另外一篇文档中的主题）。它通过中途截取组件所有的绘制请求来实现绘制机制（所以它们将不用在 AWT 中处理），同时保持它自身需要更新的状态（像“脏区域”等）。最后，它使用 invokeLater()来在事件派发线程中处理未处理的请求，见“绘制处理”部分（选项 B）。

For most programs, the RepaintManager can be viewed as part of Swing's internal system and can virtually be ignored. However, its API provides programs the option of gaining finer control over certain aspects of painting.

对于大多数程序，RepaintManager 可以被当作是 Swing 的内部系统事实上可以被忽视。然而，它的 API 为程序的某些绘制方面做更好的控制提供了选择。

The "Current" RepaintManager “当前”绘制管理器

The RepaintManager is designed to be dynamically plugged, although by default there is a single instance. The following static methods allow programs to get and set the "current" RepaintManager:

RepaintManager 被设计为可动态拔插，虽然缺省只有一个实例。下面的静态方法允许获得和设置“当前”RepaintManager：

```
public static RepaintManager currentManager(Component c)
public static RepaintManager currentManager(JComponent c)
public static void setCurrentManager(RepaintManager aRepaintManager)
```

Replacing The "Current" RepaintManager

取代“当前”绘制管理器

A program would extend and replace the RepaintManager globally by doing the following:

程序可能会通过下面的方式来全局取代或扩展 RepaintManager：

```
RepaintManager.setCurrentManager(new MyRepaintManager());
```

You can also see RepaintManagerDemo for a simple running example of installing a RepaintManager which prints out information about what is being repainted.

你还可以看看一个简单的例子 RepaintManagerDemo 来说明如何安装一个 RepaintManager，它会输出一些有什么被绘制的信息。

A more interesting reason for extending and replacing the RepaintManager would be to change how it processes repaint requests. Currently the internal state used by the default implementation to track dirty regions is package private and therefore not accessible by subclasses. However, programs may implement their own mechanisms for tracking dirty regions and for collapsing requests by overriding the following methods:

一个更有趣的扩展和取代 RepaintManager 的理由是改变它对绘制请求的处理。当前缺省实现中用来跟踪脏区域的内部状态是包私有的，并且不可以被子类访问。但是程序可以通过覆盖下面的方法使其失效来实现自己的脏区域跟踪机制。

```
public synchronized void addDirtyRegion(JComponent c, int x, int y, int w, int h)
public Rectangle getDirtyRegion(JComponent aComponent)
public void markCompletelyDirty(JComponent aComponent)
public void markCompletelyClean(JComponent aComponent)
```

The addDirtyRegion() method is the one which is invoked when repaint() is called on a Swing component, and thus can be hooked to catch all repaint requests. If a program overrides this method (and does not call super.addDirtyRegion()) then it becomes its responsibility to use invokeLater() to place a Runnable on the EventQueue which will invoke paintImmediately() on an appropriate component (translation: not for the faint of heart).

addDirtyRegion()是在 Swing 组件 repaint()方法被调用之后触发的方法之一，它可以捕捉所有的绘制请求。如果程序覆盖了该方法（并没有调用 super.addDirtyRegion()），接下来它会使用 invokeLater()来放置一个 Runnable 在事件队列中，然后会在合适的组件上调用 paintImmediately()（说明：胆小慎用）。

Global Control Over Double-Buffering　双缓冲的全局控制

The RepaintManager provides an API for globally enabling and disabling double-buffering:

RepaintManager 提供了全局启用和禁用双缓冲的 API：

```
public void setDoubleBufferingEnabled(boolean aFlag)
public boolean isDoubleBufferingEnabled()
```

This property is checked inside of JComponent during the processing of a paint operation in order to determine whether to use the offscreen buffer for rendering. This property defaults to true, but programs wishing to globally disable double-buffering for all Swing components can do the following:

该属性在 JComponent 的绘制操处理中用于做检查，以决定是否使用后台缓冲来渲染。这个属性的缺省值为 true，但是如果程序想对所有 Swing 组件进行全局禁用双缓冲，可以这么做：

```
RepaintManager.currentManager(mycomponent).setDoubleBufferingEnabled(false);
```

Note: since Swing's default implementation instantiates a single RepaintManager instance, the mycomponent argument is irrelevant.

说明：Swing 的缺省实现实例化了一个 RepaintManager 实例，与 mycomponent 参数无关。

Swing Painting Guidelines　Swing 绘制指南

Swing programs should understand these guidelines when writing paint code:

Swing 程序在写绘制代码的时候应该记住下面的这些指南：

1. For Swing components, `paint()` is always invoked as a result of both system-triggered and app-triggered paint requests; `update()` is never invoked on Swing components.

   对于 Swing 组件，paint()在系统触发和应用触发绘制请求中都会被调用，update()从不在 Swing 组件中调用。

2. Programs may trigger a future call to `paint()` by invoking `repaint()`, but shouldn't call `paint()` directly.

程序应该通过调用 repaint()在将来某一时间触发 paint()，不应该直接调用 paint()。

3. On components with complex output, repaint() should be invoked with arguments which define only the rectangle that needs updating, rather than the no-arg version, which causes the entire component to be repainted.

对于有复杂输出的组件，repaint()调用时应该带上指定更新矩形的参数，而不要用无参数的版本，无参数的版本会让整个组件都被绘制。

4. Swing's implementation of paint() factors the call into 3 separate callbacks:

Swing 的 paint()实现被分成如下 3 个回调：

1. paintComponent()
2. paintBorder()
3. paintChildren()

Extensions of Swing components which wish to implement their own paint code should place this code within the scope of the paintComponent() method (not within paint()).

希望实现自己的绘制代码的 Swing 组件的扩展应该把代码放在 paintComponent()方法的区域内（而不是在 paint()中）

5. Swing introduces two properties to maximize painting efficiency:

Swing 中引入了两个属性来最大化绘制的效率：

- opaque: will the component paint all its bits or not?

  opaque: 组件是否绘制所有的色位？

- optimizedDrawingEnabled: may any of this component's children overlap?

  optimizedDrawingEnabled: 该组件是否与它的子组件重叠？

6. If a Swing component's opaque property is set to true, then it is agreeing to paint all of the bits contained within its bounds (this includes clearing it's own background within paintComponent()), otherwise screen garbage may result.

如果 Swing 组件的 opaque 属性设置为 true，它会同意绘制它范围内的所有色位（这包括在 paintComponent()中清除它的背景色），否则会产生屏幕垃圾。

7. Setting either the opaque or optimizedDrawingEnabled properties to false on a component will cause more processing on each paint operation, therefore we recommend judicious use of both transparency and overlapping components.

将组件的 opaque 或 optimizedDrawingEnabled 属性设置为 false 都会产生更多绘制处理的操作，所以我们建议合理运用透明性和组件重叠。

8. Extensions of Swing components which have UI delegates (including JPanel), should typically invoke super.paintComponent() within their own paintComponent() implementation. Since the UI delegate will take responsibility for clearing the background on opaque components, this will take care of #5.

拥有 UI 代理的 Swing 组件的扩展（包括 JPanel），应该在它们自己的 paintComponent()实现中调用 super.paintComponent()，UI 代理将会负责清除不透明组件的背景色，这会涉及到第 5 步。

9. Swing supports built-in double-buffering via the JComponent doubleBuffered property, and it defaults to true for all Swing components, however setting it to true on a Swing container has the general effect of turning it on for all lightweight descendents of that container, regardless of their individual property settings.

Swing 通过 JComponent doubleBuffered 属性内置支持双缓冲，所有 Swing 组件中它都缺省为 true，然而在一个 Swing 容器上将其设置为 true 会让该容器上的所有轻量级派生都启用双缓冲的功效，不管它们单独做了怎样的属性设置。

10. It is strongly recommended that double-buffering be enabled for all Swing components.

强烈建议让所有的 Swing 组件都启用双缓冲功能。

11. Components which render complex output should make smart use of the clip rectangle to narrow the drawing operations to those which intersect with the clip area.

有着复杂渲染输出的组件应该灵活使用剪裁矩形来减少与剪裁区域打交道的绘制操作

Summary 总结

Both the AWT and Swing provide APIs to make it easy for programs to render content to the screen correctly. Although we recommend that developers use Swing for most GUI needs, it is useful to understand AWT's paint mechanism because Swing's is built on top of it.

AWT 和 Swing 都提供了 API 来让程序更轻松的在屏幕上渲染出恰当的内容。虽然我们推荐开发人员使用 Swing 来开发大多数需要 GUI，明白 AWT 的绘制机制也是很有用的，因为 Swing 是基于它的。

To get the best performance from these APIs, application programs must also take responsibility for writing programs which use the guidelines outlined in this document.

为了从这些 API 中获取最好的表现，应用程序必须按照本文提出的这些指南来编写。