Java游戏开发中应始终坚持的10项基本原则Java认证考试 PDF转换可能丢失图片或格式，建议阅读原文 https://www.100test.com/kao_ti2020/559/2021_2022_Java_E6_B8_B8_E6_88_8F_c104_559592.htm 关于文章中涉及的两个杜撰概念: 一、绘图器：众所周知，Java GUI以paint进行绘图，以repaint进行图像刷新，而完成repaint及paint这一连贯过程中所用到绘图组件，我将其称为绘图器。就我个人的体会，绘图器的调用时机应始终处于repaint之后paint之前，即通过repaint触发刷新后执行，当其中的具体逻辑完成其对应的图像绘制后，再通过统一接口将其图像插入paint中，为了匹配需要，绘图器应始终以接口方式实现。 二、监听器：这里所说的监听器，并不是特指某个Listener组件，而是包括Java游戏中所需的所有监听器集合。由于Java游戏中很可能会切换不同的游戏模式，而不同模式游戏中需要处理的鼠标或键盘事件也不尽相同。所以在Java游戏开发中，我们需要一个可替换的监听器集合，用以变更不同游戏模式下的不同监听事件，为了匹配需要，监听器应始终以接口方式实现。 正文，关于Java游戏开发中应始终坚持的10项基本原则： 1、始终保持画布的唯一性。 现实生活中，人类通过口腔及消化道摄取的营养物质可以被心、肝、脾、肺、肾等内脏吸收，却没有人会想给自己的心、肝、脾、肺、肾上也弄个嘴，因为一致性的功能实现只要有一个就足够了。但是，有时我们不经意的在游戏中add、remove不同panel或canvas以求转换画面的行为，无异于是想给游戏的心、肝、脾、肺、肾上装嘴的不智之举，切忌Java的GUI都是画出来的，重绘就好，没有切换组件的必要，否则费力不讨好。 2、始终以接口方式转换监听及

处理图像绘制，务必将视图及逻辑层分开。 针对一些混合类型的游戏，比如SLG AVG、RPG STG，我们将面临不同模式下游戏的监听器及绘图器切换问题。 这时最简单的抉择莫过于为每一个游戏类型都订制一个对应的面板进行切换，这样虽表面上方省心，但却也是最费力而不讨好的，且不说闪烁问题需要单独解决，资源占用问题，光冗余代码就够人头痛了。 其次就是在一个面板中针对不同游戏类型使用switch判断以切换监听及绘图，事件数量少时固然可以，效率也不错，但稍微多一点恐怕就不那么简单，更多时则仅余郁闷，同样不建议使用。 就我个人所见，解决这一问题的最好方法莫过于沿用MVC模式，以接口方式构建绘图器及监听器，当游戏出现变更时，我们仅仅需要切换监听及绘图接口，就可以迅速转变游戏内容，而无需区别对待不同的实现，这样即避免了组件切换的闪烁及延迟，也精简了代码，更有利于开发时的模块划分。 3、始终以静态方式加载游戏常用资源，缓存常用对象，并及时释放无用资源。 即使历史发展到今天，Java依旧没有彻底摆脱其系统资源杀手的可憎面目，GC机制也导致我们无法适时地释放资源，new的越多，系统也变得越慢，这对于大量使用图形资源的游戏来讲尤其要命。所以我们要尽一切可能令常用资源静态化为唯一实例以避免反复调用，而将一些调用后不会再使用或很少使用的资源迅速null以等待GC自动回收。否则，你将发现你的游戏距离内存溢出是那样的近…… 4、始终以循环方式展开游戏，利用线程控制游戏流程，避免出现僵直现象。 事实上所谓的游戏开发，在某种程度上不过是由程序员制作出的一种夹杂着各类图形算法，用以适时地展示各种资源的幻灯程序；唯一的区

别在于，普通幻灯程序中人机交互性较弱，而游戏的人机交互性较强罢了。 我们都知道，幻灯程序在展示中无论如何跳转展示页，也必然有其固定的begin与end页面，而且也势必能重复从头至尾顺序循环其begin与end，以此构成一个幻灯片。实际上游戏制作也一样，无论游戏流程如何转变，游戏都会有也必然会有一个主流程，或者说一个主循环体，这样我们才能由游戏开始进行到游戏结束，而不是从一个结束到另一个结束，也就是说无论游戏中细节分支有多少，它的主流程处理及判定也必然是顺序的。针对这一特性，决定了我们应将游戏主体代码至于一个大的循环体之内，再利用线程控制循环体中的游戏进度，从而更好的顺应这一流程。简单的说，我们应将循环体中每一个使用到的绘图器都当作于flash中的一桢,而线程的各种控制当作时间轴,用以调节不同桢的播放速度及调用时机，以此完成各种不同的事件交互。 5、始终在处理复杂绘图时直接准备贴图而非由程序绘制。 我们都知道Java绘图事实上是GDI实现，因此其绘制复杂画面的效率也就可想而知。通常强况下，除非当前的效果非编程不能实现，或者其所造成的资源损耗确实微小到可以忽略不计，否则最好的方法就是用空间换效率，准备好图片直接贴上去吧，宁可增加些程序体积，也不要让玩家因等待的愤怒而问候你祖宗八辈。 6、始终保证repaint仅刷新需要部分，避免无谓的全局重绘。 每repaint一次，事实上就是将paint中的图形打印到窗体上一次，窗体越大，处理的图像越复杂，repaint所造成的资源损耗也势必越多，运行效率也势必越低。但反过来说，由于Java允许我们限定repaint的范围，因而我们可以将刷新限定在某一特定区域内，更准确地说我们可以仅在需要变

更画面的位置上才进行刷新，以此将损耗降低到最低限度，总体上说，即使我们会因为计算刷新区域额外花费些许时间，总体上讲也比全局repaint要快得多。 7、始终双缓冲游戏图像避免闪烁现象发生。 对于Java绘图而言，每次调用repaint方法时都会清除整个屏幕，然后paint才显示画面。而万一系统速度不够，在清除背景和绘制图像间的短暂间隔内被用户看见，就出现了所谓的闪烁现象；简单来讲闪烁的成因就是运算效率不足，使得repaint与paint不连贯造成的。 针对这种情况，我们需要利用双缓冲技术加以解决。 双缓冲实现其实简单至极，主要过程就是先创建一个等大小于希望绘制图形的Image，而后取得其Graphics，每当paint绘图时我们不直接将图像绘制于paint函数的Graphics上，而是绘制于我们创建的缓冲图像的Graphics上，当绘制完成后再调用paint函数提供的drawImage方法，将整个后台图像一次画到屏幕上去。这种方法的优点在于大部分绘制是在后台进行的。将后台绘制的图像一次绘制到屏幕上。 这时只要系统速度正常，我们所看到的绘图将不再有闪烁现象发生。 8、始终在自绘组件的桌面游戏中应用AWT或SWT而非Swing。 众所周知，Swing(JFC)的GUI是以AWT为基础在本地窗体绘制而成，相较AWT虽然提供了更为丰富的组件，但也意味着它占用了更多的资源。而事实上，大多数Java桌面游戏组件是由开发者所针对性绘制，并非Swing库提供，也不需要Swing库支持，我们完全可以放弃Swing而选择AWT或 SWT（SWT绘图与AWT/Swing绘图在方法上略有区别，但本质一样）这种直接Native而来的界面，实在不需劳动Swing他老人家，平白的耗费掉那些本就因使用Java应用而稀缺的系统资源。 9、始终别忘了在Graphics处

理完毕后dispose。 Graphics的dispose与数据库Connection的close可谓异曲同工。为此我特意做了一个实验，在死循环中无间隔无优化的反复 repaint一幅2000X2000的大图，应用dispose时虽然刷新很慢并伴随闪烁但总体讲正常，而去掉dispose运行大约一分钟后万恶的溢出大神降临…… 当然，就像Connection应在全部操作完成后才close一样，Graphics也仅在全部绘图完毕后才需要dispose，也就是当最后一个paint最后一次draw后，别忘了留个dispose关门，除非你很想看见溢出大神…… 10、始终以运算效率为第一优先，可适当放弃代码可读性，可适当违背OO原则。 以Java进行游戏开发，最大的问题莫过于系统资源的损耗，在关键问题上，就别死抱着OO不放了。 若你能始终坚持以上十点，虽然别指望就此超越C/C 游戏的运行效率，但已能与Delphi游戏争锋而无愧色，傲视于vb6、Flash、rmxp、rmvx等工具开发的游戏而鄙夷之。 